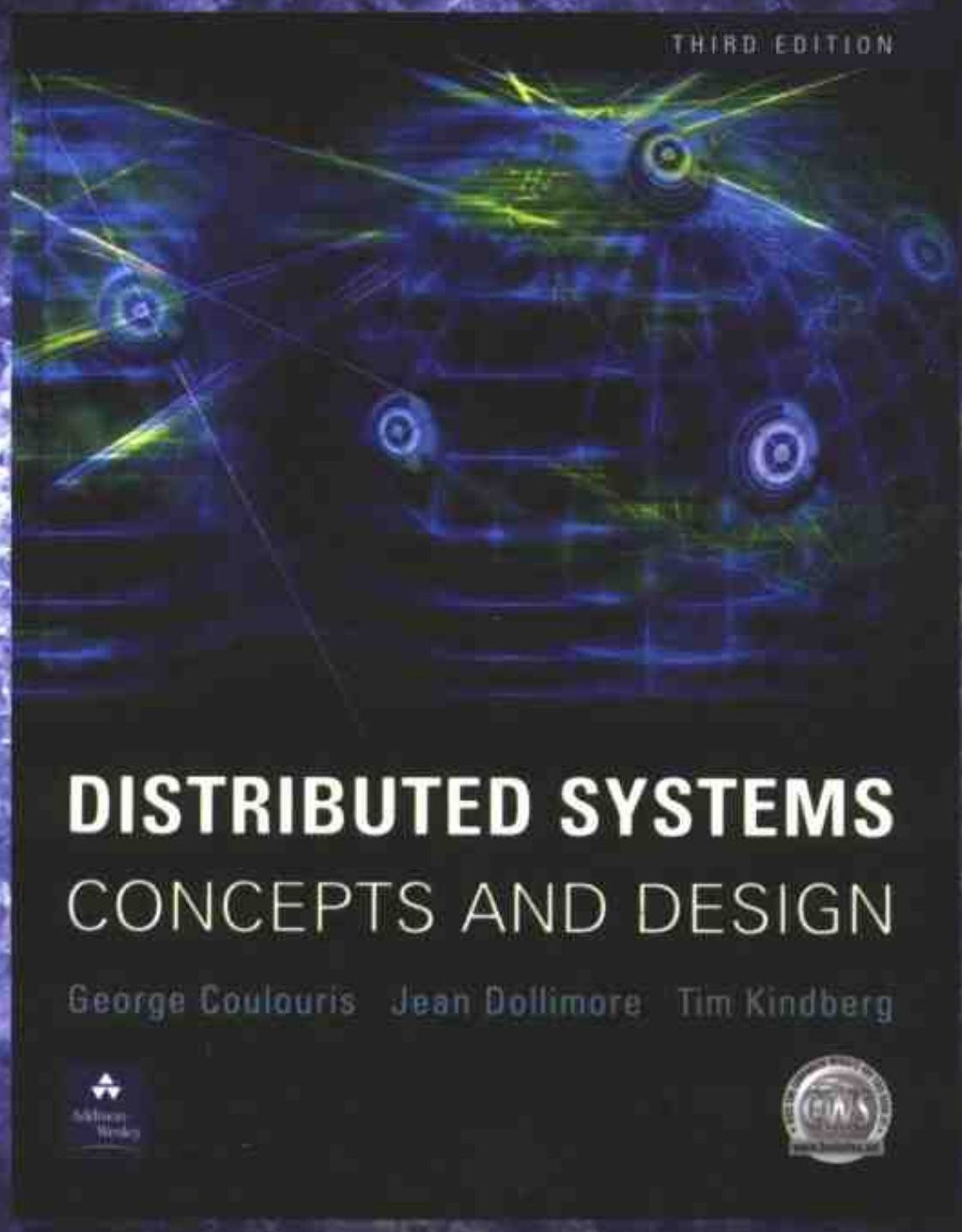
计 算 机 科 学 丛 书

原书第3版

分布式系统

概念与设计

(英) George Coulouris Jean Dollimore Tim Kindberg 著 金蓓弘 等译



Distributed Systems

Concepts and Design

Third Edition

机械工业出版社 China Machine Press
中信出版社 CITIC PUBLISHING HOUSE

"在分布式系统领域，我不知道还有没有更好的书，但我毫不犹豫地推荐这本书。"

——**Jan Madey**
《*IEEE Parallel and Distributed Technology*》杂志

"介绍分布式系统的最好教科书。"

——**E. Douglas Jensen**
Mitre公司首席科学家，世界分布式实时系统权威

本书旨在提供深入的分布式系统设计原理和实践方面的知识，读者通过学习可以掌握评价已有系统或设计新系统的方法。书中的实例研究阐述了每个主要论题的设计概念。

本书已被爱丁堡大学、伊利诺依大学、卡内基-梅隆大学、南加州大学、得克萨斯A&M大学、多伦多大学、罗彻斯特理工学院等世界众多名校采用为高级操作系统、计算机网络、分布式系统课程的教材。

综合性网站www.cdk3.net 和www.booksites.net/cdkbook 为读者提供了补充资料（勘误、源代码等）。

作者简介

George Coulouris 是伦敦大学Queen Mary and Westfield学院的荣誉教授，同时是剑桥大学通信工程实验室的资深客座研究员。他领导的项目研究有关可动态配置的多媒体系统的服务质量，最近他一直致力于计算机支持协同工作和应用、分布式多媒体中间件和群件安全模型方面的研究工作。

Jean Dollimore 在退休前是伦敦大学Queen Mary and Westfield学院的高级研究员，最近一直在从事有关计算机支持协同工作、分布式多媒体中间件和群件安全模型方面的研究。

Tim Kindberg 是惠普Palo Alto实验室的研究员，目前正在研究用于移动计算的基于Web的基础设施，目的是用Web资源扩展物理世界。之前，他领导过一个计算机支持协同工作的研究项目，并从事过分布式操作系统和分布式多媒体中间件的研究。

封面设计：陈子平

ISBN 7-111-12956-3
9 787111 129561

HZ BOOKS
华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259，68995264
读者服务信箱：hzedu@hzbook.com
http://www.hzbook.com

ISBN 7-111-12956-3/TP · 2904
定价：59.00 元

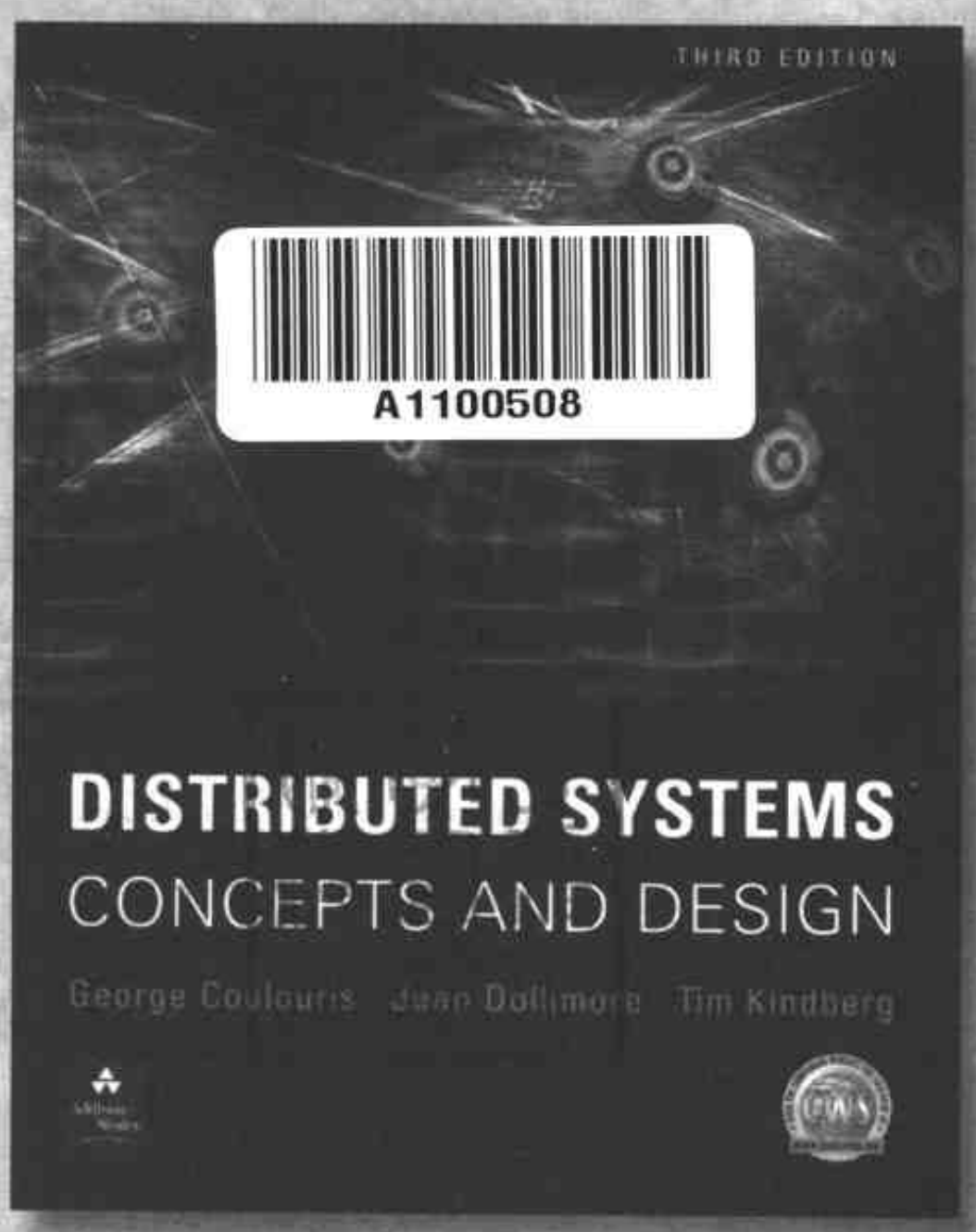
计 算 机 科 学 丛 书

原书第3版

分布式系统

概念与设计

（英）George Coulouris　Jean Dollimore　Tim Kindberg 著　金蓓弘　等译



Distributed Systems

Concepts and Design

Third Edition

机械工业出版社 China Machine Press
中信出版社 CITIC PUBLISHING HOUSE

本书旨在提供深入的分布式系统设计原理和实践方面的知识，读者通过学习可以掌握评价已有系统或设计新系统的方法。书中结合分布式系统技术主要的新进展，重点介绍了因特网、企业内部网、Web和中间件，还包括故障建模和容错、分布式对象和分布式多媒体系统。本书非常强调算法，并结合其他相关技术讨论了安全问题。

本书内容详实、覆盖面广且循序渐进，适合作为大中专院校计算机系高年级本科生及研究生的教科书和教学参考书，同时也可以作为计算机软件行业技术人员的参考书。

本书版权登记号：图字：01-2002-0609

图书在版编目（CIP）数据

分布式系统概念与设计（原书第3版）/（英）康乐瑞斯（Coulouris, G.）等著；金蓓弘等译. -北京：机械工业出版社，2004.1

（计算机科学丛书）

书名原文：Distributed Systems: Concepts and Design（Third Edition）

ISBN 7-111-12956-3

Ⅰ.分… Ⅱ.①康… ②金… Ⅲ.分布式操作系统 Ⅳ.TP316.4

中国版本图书馆CIP数据核字（2003）第076043号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：龙 燕

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2004年1月第1版第1次印刷

787mm×1092mm 1/16·38.25印张

印数：0 001-5 000册

定价：59.00元

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范、又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及收藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经二十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

前　言

在分布式系统，特别是Web和其他基于互联网的应用和服务变得备受关注和空前重要之时，这本教材的第3版问世了。本书旨在传授基于互联网（及其他）的分布式系统的设计原理和实践知识。全书提供了充足的信息，供读者评价已有的系统或设计--个新系统，并且还包含了详细的实例研究，用于进一步阐明每个主题的概念。

分布式系统技术，如进程间通信、远程调用、分布式命名、密码安全、分布式文件系统、数据复制和分布式事务机制，是在过去二三十年中发展起来的，它提供运行时的基础结构，以支持当今的网络计算机应用。

软件框架提供对诸如分布式共享对象的抽象，以及包括安全通信、认证和访问控制、移动代码、事务和永久存储机制等服务，软件框架的使用意味着分布式系统的开发日益依赖中间件支持。

在不久的将来，分布式应用将通过复制数据和多媒体数据流促进用户之间更加紧密的协作，同时，通过使用无线和自发网络支持用户和设备的移动性。

分布式系统和应用的开发者可从许多语言、工具和环境中受益。学生和专业开发人员可利用它们来构造可运行的分布式应用。

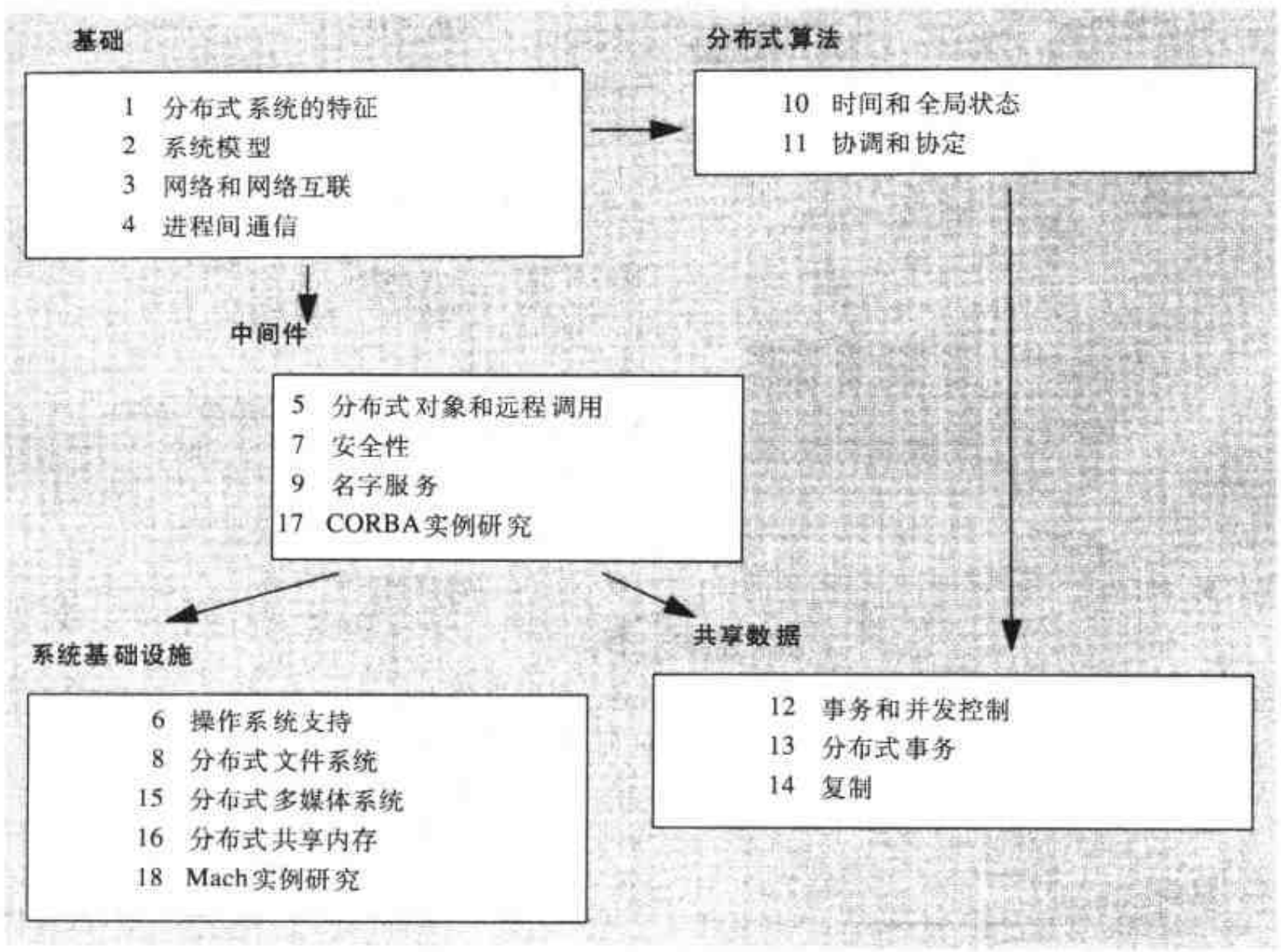
目的和读者对象

本书可用做本科生教材和研究生的入门教材，也可作为自学教材。本书采用自顶向下的方法，首先叙述在分布式系统设计中要解决的问题，然后，通过抽象模型、算法和对广泛使用的系统实例进行详细研究的方式，描述成功开发分布式系统的方法。本书覆盖的领域有足够的深度和广度，为读者继续研究大多数分布式系统文献提供了良好的基础。

本书针对具有面向对象编程、操作系统、初级计算机体系结构等基础知识的学生。全书覆盖与分布式系统有关的计算机网络部分，包括因特网、广域网、局域网和无线网的基本技术。本书中的算法和接口大部分用Java描述，少部分用ANSI C描述。为了表述上的简洁明了，还使用一种从Java/C中派生出来的伪代码。

本书的组织结构

下图表明本书的章节可归结在5个主题下。它提供了本书的--个结构指南，为教师、读者提供了一个推荐的导航路径，以便于他们理解分布式系统设计中的不同子领域。



参考文献

万维网的*存*在改变了书（比如本书）与资料（包括研究论文、技术规范和标准）的链接方式。许多源文件现在可从Web上获取；有一些甚至只能从Web上获取。出于简洁和可读性的考虑，本书对Web参考资料采用了一种特殊的类似URL的格式，诸如[www.omg.org]和[www.rsasecurity.com I]指的是仅能从Web上获得的文档。在书后的参考文献列表中可以找到它们，但是完整的URL仅在本书参考文献的联机版本（www.cdk3.net/refs）上给出。两个版本的参考文献列表都包含对这种机制的更详细的解释。

第3版所做的修改

第3版在第2版出版6年后问世。下表总结了这版书稿所做的工作。相对于第2版，我们已经重写了大量介绍性的章节和其他一些章节，新增的几章旨在反映新的观点和技术方向。有些章节进行了重组织，包括覆盖的主题、内容的深度、资料的位置。为了给新的主题留出篇幅，本书压缩了原来的内容。有些材料被移到更为显著的位置以突出它的重要性。可从本书的Web网站（见以下介绍）找到从第2版中移走的实例研究。

重写和扩展的章节：	1 分布式系统的特征
	3 网络和网络互联
	4 进程间通信
	5 分布式对象和远程调用
	7 安全性
	10 时间和全局状态
	14 复制
完全新增的章节：	2 系统模型
	11 协调和协定
	15 分布式多媒体系统
	17 CORBA实例研究
修改的章节：	6 操作系统支持
	8 分布式文件系统
	9 名字服务
	12 事务和并发控制
	13 分布式事务
	16 分布式共享内存
	18 Mach实例研究

致谢

感谢玛丽女王与韦斯特菲尔德学院的学生，他们在讲座和讨论会中提供的内容帮助我们形成此新版本。也感谢其他教师提出的有益的建议，Kohei Honda在QMW的教学中试用过该版本的草稿，提出了一些相当有价值的意见。

Angel Alvarez、Dave Bakken、John Barton、Simon Boggis、Brent Callaghan、Keith Clarke、Kurt Jensen、Roger Prowse倾注了大量的时间，复审了完成稿的关键部分，并给出了参考意见。

还要感谢下列人员，他们允许我们使用其资料，或对本书应该包括的主题给出了建议：Tom Berson、Antoon Bosselaers、Ralph Herrtwitch、Frederick Hirsch、Bob Hopgood、Ajay Kshemkalyan、Roger Needham、Mikael Pettersson、Rick Schantz、David Wheeler。

感谢玛丽女王与韦斯特菲尔德学院计算机科学系提供Web网站，感谢Keith Clarke和Tom King对建立网站的支持。

最后，感谢Pearson Education/Addison-Wesley的Keith Mansfield、Bridget Allen、Julie Knight以及Kristin Erickson为本书的出版、付印全过程提供的支持。

Web网站

我们维护的Web网站提供了大量的材料，可以帮助教师和读者更好地理解本书内容。可通过下列URL访问该网站：www.cdk3.net和www.booksites.net/cdkbook。

该Web网站包括：

- 参考文献列表：书后的参考文献列表也可在Web网站上找到。参考文献列表的Web版本

包含可联机获得的资料的Web链接。

- 勘误表：给出书中的错误和修正列表。
- 补充材料：我们计划为每一章保留一套补充材料。最初，包括书中程序的源代码和相关的阅读材料（主要是本书上一版本中的但在此版本中出于篇幅上的考虑而被删除的材料），该类的补充材料在本书中用类似www.cdk3.net/ipc的链接表示。
- 他人贡献的教学材料：我们希望扩展补充材料以涵盖在本书写作和出版后出现的新主题。我们邀请教师提供教学材料，包括讲座笔记和实验项目。Web网站给出了提交补充材料的方法，提交的材料将由使用本书的几位教师组成的一个小组进行复审。
- 使用本书的课程的网站链接：要求使用该书的教师告知我们他们的URL，以便包含在我们的链接列表中。
- 教师指南。

George Coulouris
Jean Dollimore
Tim Kindberg
2000年6月于伦敦&Palo Alto
<authors@cdk3.net>

目　录

出版者的话
专家指导委员会
前言
第1章　分布式系统的特征 ……1
1.1　简介 ……1
1.2　分布式系统实例 ……2
1.2.1　因特网 ……2
1.2.2　企业内部网 ……3
1.2.3　移动计算和无处不在的计算 ……4
1.3　资源共享和Web ……5
1.4　挑战 ……12
1.4.1　异构性 ……12
1.4.2　开放性 ……13
1.4.3　安全性 ……14
1.4.4　可伸缩性 ……14
1.4.5　故障处理 ……16
1.4.6　并发 ……17
1.4.7　透明性 ……17
1.5　小结 ……18
第2章　系统模型 ……21
2.1　简介 ……21
2.2　体系结构模型 ……22
2.2.1　软件层 ……22
2.2.2　系统体系结构 ……24
2.2.3　客户－服务器模型的变种 ……27
2.2.4　接口和对象 ……31
2.2.5　分布式体系结构的设计需求 ……32
2.3　基础模型 ……34
2.3.1　交互模型 ……35
2.3.2　故障模型 ……39
2.3.3　安全模型 ……42
2.4　小结 ……45
第3章　网络和网络互联 ……47
3.1　简介 ……47
3.2　网络类型 ……50
3.3　网络原理 ……52
3.3.1　数据包的传输 ……52
3.3.2　数据流 ……53
3.3.3　交换模式 ……53
3.3.4　协议 ……54
3.3.5　路由 ……59
3.3.6　拥塞控制 ……62
3.3.7　网络互联 ……62
3.4　因特网协议 ……65
3.4.1　IP寻址 ……67
3.4.2　IP协议 ……68
3.4.3　IP路由 ……70
3.4.4　IPv6 ……72
3.4.5　移动IP ……74
3.4.6　TCP和UDP ……75
3.4.7　域名 ……77
3.4.8　防火墙 ……77
3.5　网络实例研究：以太网、无线LAN和ATM ……80
3.5.1　以太网 ……81
3.5.2　IEEE 802.11无线LAN ……84
3.5.3　异步传输模式网络 ……86
3.6　小结 ……88
第4章　进程间通信 ……91
4.1　简介 ……91
4.2　因特网协议的API ……92
4.2.1　进程间通信的特征 ……92
4.2.2　套接字 ……93
4.2.3　UDP数据报通信 ……94
4.2.4　TCP流通信 ……96
4.3　外部数据表示和编码 ……101
4.3.1　CORBA的公共数据表示（CDR） …102
4.3.2　Java对象序列化 ……103

4.3.3 远程对象引用 ……105
4.4 客户－服务器通信……106
4.5 组通信……111
4.5.1 IP组播——组通信的实现 ……112
4.5.2 组播的可靠性和排序 ……113
4.6 实例研究：UNIX系统的进程间通信 …115
4.6.1 数据报通信 ……115
4.6.2 流通信 ……116
4.7 小结……117
第5章 分布式对象和远程调用 ……121
5.1 简介……121
5.2 分布式对象间的通信……124
5.2.1 对象模型 ……124
5.2.2 分布式对象 ……125
5.2.3 分布式对象模型 ……126
5.2.4 RMI的设计问题 ……127
5.2.5 RMI的实现 ……129
5.2.6 分布式无用单元回收 ……133
5.3 远程过程调用……134
5.4 事件和通知……137
5.4.1 分布式事件通知的参与者 ……138
5.4.2 Jini分布式事件规范……141
5.5 Java RMI实例研究 ……142
5.5.1 创建客户程序和服务器程序 ……145
5.5.2 Java RMI的设计和实现 ……147
5.6 小结……148
第6章 操作系统支持 ……153
6.1 简介……153
6.2 操作系统层……154
6.3 保护……156
6.4 进程和线程……157
6.4.1 地址空间 ……158
6.4.2 新进程的创建 ……160
6.4.3 线程 ……162
6.5 通信和调用……171
6.5.1 调用性能 ……172
6.5.2 异步操作 ……177
6.6 操作系统体系结构……179
6.7 小结……182
第7章 安全性……185
7.1 简介……185
7.1.1 威胁和攻击 ……187
7.1.2 保护电子事务 ……189
7.1.3 设计安全系统 ……190
7.2 安全技术概述……192
7.2.1 密码学 ……192
7.2.2 密码学的应用 ……192
7.2.3 证书 ……195
7.2.4 访问控制 ……196
7.2.5 凭证 ……198
7.2.6 防火墙 ……199
7.3 加密算法……200
7.3.1 保密密钥（对称）算法 ……203
7.3.2 公开密钥（非对称）算法 ……205
7.3.3 混合密码协议 ……207
7.4 数字签名……207
7.4.1 公开密钥数字签名 ……209
7.4.2 保密密钥数字签名——MAC ……209
7.4.3 安全摘要函数 ……210
7.4.4 证书标准和证书权威机构 ……212
7.5 密码实用学……213
7.5.1 加密算法的性能 ……213
7.5.2 密码学的应用和政治障碍 ……213
7.6 实例研究：Needham-Schroeder、Kerberos、SSL和Millicent ……215
7.6.1 Needham-Schroeder认证协议 ……215
7.6.2 Kerberos ……216
7.6.3 使用安全套接字确保电子交易安全…220
7.6.4 小额电子交易：Millicent协议 ……223
7.7 小结……226
第8章 分布式文件系统 ……229
8.1 简介……229
8.1.1 文件系统的特点 ……231
8.1.2 分布式文件系统的需求 ……233
8.1.3 实例研究 ……234
8.2 文件服务系统结构……235
8.3 Sun网络文件系统 ……239
8.4 Andrew文件系统 ……248

8.4.1 实现 ……250
8.4.2 缓存的一致性 ……253
8.4.3 其他方面 ……255
8.5 最新进展 ……255
8.6 小结 ……260
第9章 命名服务 ……263
9.1 简介 ……263
9.2 命名服务和域名系统 ……265
9.2.1 名字空间 ……266
9.2.2 名字解析 ……269
9.2.3 域名系统 ……271
9.3 目录服务和发现服务 ……277
9.4 实例研究：全局命名服务 ……279
9.5 实例研究：X.500目录服务 ……282
9.6 小结 ……285
第10章 时间和全局状态 ……289
10.1 简介 ……289
10.2 时钟、事件和进程状态 ……290
10.3 同步物理时钟 ……292
10.3.1 同步系统中的同步 ……293
10.3.2 同步时钟的Cristian方法 ……293
10.3.3 Berkeley算法 ……294
10.3.4 网络时间协议 ……295
10.4 逻辑时间和逻辑时钟 ……297
10.5 全局状态 ……300
10.5.1 全局状态和一致割集 ……301
10.5.2 全局状态谓词、稳定性、安全性和活性 ……303
10.5.3 Chandy和Lamport的“快照”算法 …303
10.6 分布式调试 ……307
10.6.1 观察一致的全局状态 ……308
10.6.2 求解可能的φ ……309
10.6.3 求解明确的φ ……310
10.6.4 在同步系统中求解可能的φ和明确的φ ……311
10.7 小结 ……311
第11章 协调和协定 ……315
11.1 简介 ……315
11.2 分布式互斥 ……318
11.3 选举 ……324
11.4 组播通信 ……327
11.4.1 基本组播 ……329
11.4.2 可靠组播 ……329
11.4.3 有序组播 ……332
11.5 共识和相关问题 ……339
11.5.1 系统模型和问题定义 ……339
11.5.2 同步系统中的共识问题 ……342
11.5.3 同步系统中的拜占庭将军问题 …343
11.5.4 异步系统的不可能性 ……346
11.6 小结 ……347
第12章 事务和并发控制 ……351
12.1 简介 ……351
12.1.1 简单的同步机制（无事务）……352
12.1.2 事务的故障模型 ……353
12.2 事务 ……354
12.2.1 并发控制 ……357
12.2.2 事务放弃时的恢复 ……360
12.3 嵌套事务 ……362
12.4 锁 ……364
12.4.1 死锁 ……369
12.4.2 在加锁机制中增加并发度 ……372
12.5 乐观并发控制 ……374
12.6 时间戳排序 ……377
12.7 并发控制方法的比较 ……383
12.8 小结 ……384
第13章 分布式事务 ……389
13.1 简介 ……389
13.2 平面分布式事务和嵌套分布式事务 …390
13.3 原子提交协议 ……392
13.3.1 两阶段提交协议 ……393
13.3.2 嵌套事务的两阶段提交协议 ……395
13.4 分布式事务的并发控制 ……398
13.4.1 锁 ……399
13.4.2 时间戳排序并发控制 ……399
13.4.3 乐观并发控制 ……400
13.5 分布式死锁 ……401
13.6 事务恢复 ……406
13.6.1 日志 ……408

13.6.2 影子版本 ……410
13.6.3 为何恢复文件需要事务状态和意图列表 ……411
13.6.4 两阶段提交协议的恢复 ……411
13.7 小结 ……414
第14章 复制 ……417
14.1 简介 ……417
14.2 系统模型和组通信 ……419
14.2.1 系统模型 ……419
14.2.2 组通信 ……421
14.3 容错服务 ……425
14.3.1 被动（主备份）复制 ……427
14.3.2 主动复制 ……429
14.4 高可用服务 ……430
14.4.1 gossip系统 ……431
14.4.2 Bayou系统的操作变换方法 ……438
14.4.3 Coda文件系统 ……439
14.5 复制数据上的事务 ……444
14.5.1 用于复制事务的体系结构 ……445
14.5.2 可用拷贝复制 ……446
14.5.3 网络分区 ……448
14.5.4 带验证的可用拷贝 ……449
14.5.5 法定数共识方法 ……449
14.5.6 虚拟分区算法 ……451
14.6 小结 ……453
第15章 分布式多媒体系统 ……457
15.1 简介 ……457
15.2 多媒体数据的特征 ……460
15.3 服务质量管理 ……461
15.3.1 服务质量协商 ……464
15.3.2 许可控制 ……468
15.4 资源管理 ……469
15.5 流适应 ……470
15.5.1 调整 ……471
15.5.2 过滤 ……471
15.6 实例研究：Tiger视频文件服务器 ……472
15.7 小结 ……475
第16章 分布式共享内存 ……477
16.1 简介 ……477
16.1.1 消息传递机制和DSM的比较 ……478
16.1.2 DSM的实现方法 ……479
16.2 设计问题和实现问题 ……481
16.2.1 结构 ……481
16.2.2 同步模型 ……482
16.2.3 一致性模型 ……483
16.2.4 更新选项 ……486
16.2.5 粒度 ……487
16.2.6 系统颠簸 ……488
16.3 顺序一致性和Ivy ……488
16.3.1 系统模型 ……488
16.3.2 写失效 ……489
16.3.3 失效协议 ……491
16.3.4 一个动态分布式管理器算法 ……492
16.3.5 系统颠簸 ……494
16.4 释放一致性和Munin ……494
16.4.1 内存访问 ……495
16.4.2 释放一致性 ……496
16.4.3 Munin ……497
16.5 其他一致性模型 ……499
16.6 小结 ……500
第17章 CORBA实例研究 ……503
17.1 简介 ……503
17.2 CORBA RMI ……504
17.2.1 CORBA客户和服务器举例 ……506
17.2.2 CORBA体系结构 ……509
17.2.3 CORBA接口定义语言 ……511
17.2.4 CORBA远程对象引用 ……515
17.2.5 CORBA语言映射 ……515
17.3 CORBA服务 ……516
17.3.1 CORBA命名服务 ……517
17.3.2 CORBA事件服务 ……519
17.3.3 CORBA通知服务 ……520
17.3.4 CORBA安全服务 ……522
17.4 小结 ……522
第18章 Mach实例研究 ……527
18.1 简介 ……527
18.1.1 设计目标和主要设计特点 ……528
18.1.2 Mach主要的抽象概述 ……529

18.2 端口、命名和保护 ……530
18.3 任务和线程 ……531
18.4 通信模型 ……533
18.4.1 消息 ……533
18.4.2 端口 ……534
18.4.3 mach_msg 系统调用 ……535
18.5 通信实现 ……535
18.5.1 透明消息传递 ……535
18.5.2 开放性：协议和驱动程序 ……537
18.6 内存管理 ……537
18.6.1 地址空间结构 ……538
18.6.2 内存共享：继承和消息传递 ……538
18.6.3 对写时复制的评价 ……539
18.6.4 外部分页 ……540
18.6.5 对访问内存对象的支持 ……541
18.7 小结 ……542
参考文献 ……545
索引 ……575

第1章　分布式系统的特征

1.1　简介
1.2　分布式系统实例
1.3　资源共享和Web
1.4　挑战
1.5　小结

分布式系统是组件分布在网络计算机上且通过消息传递进行通信和动作协调的系统。分布式系统具有下列特征：组件的并发性，缺乏全局时钟，组件故障的独立性。

我们给出分布式系统的3个实例：

- 因特网。
- 企业内部网，是因特网的一部分，一般由一个机构负责管理。
- 移动和无处不在的计算。

资源共享是形成分布式系统的主要动力。资源可以由服务器管理并由客户访问，或封装成对象，由其他客户对象访问。作为资源共享的例子，我们将讨论Web并介绍它的主要特征。

构造分布式系统的挑战是其组件的异构性、开放性（指允许增加或替换组件）、安全性、可伸缩性（指用户数量增加时能正常运行的能力）、故障处理以及组件的并发性和透明性。

1.1　简介

计算机网络无处不在。因特网是由许多网络组成的一个网络。移动电话网、公司网络、工厂网络、校园网、家庭网络、车内网络，所有这些（或单独或组合到一起）具有相同的本质特征，都可以放在分布式系统的标题下来研究。本书旨在解释影响系统设计者和实现者的网络化计算机的特征，并描述已有的可帮助完成设计和实现分布式系统任务的主要概念和技术。

我们定义分布式系统为一个硬件或软件组件分布在网络计算机上，仅仅通过消息传递进行通信和动作协调的系统。这个简单的定义覆盖了所有有效地部署了网络化计算机的系统。

同一网络中的计算机可能在空间上有距离，可能在不同的洲，也可能在同一个楼或同一个房间。我们关于分布式系统的定义产生下列重要的结论：

- *并发性*　在一个计算机网络中，程序并发执行是常见的行为。大家可以在各自的计算机上工作，在必要时共享诸如Web页面或文件之类的资源。系统处理共享资源的能力应该随着网络资源（例如，计算机）的增加而增加。在本书的许多地方将描述如何有效实施这种额外能力。如何协调并发执行的共享资源型程序也是一个重要的并经常被提及的话题。
- *缺乏全局时钟*　在程序需要协作时，它们通过交换消息来协调它们的动作。紧密的协调经常依赖于对程序动作发生时间的共识，但是，事实证明网络上计算机同步时钟的准确性会受到限制，即没有一个正确时间的全局概念。这是通过网络发送消息作为惟一的通信方式这一事实带来的直接结果。同步问题及其解决方案将在第10章描述。
- *故障独立性*　所有的计算机系统都可能发生故障，一般由系统设计者负责处理可能出现

的故障。分布式系统可能以新的方式出现故障。网络故障导致与之互联的计算机的隔离，但这并不意味着它们停止运行。事实上，计算机中的程序不能够检测网络是出现故障还是网络运行得比通常慢。同样，计算机的故障或系统中程序的异常终止并不能马上被与之通信的其他组件感知。系统的每个组件会单独地出现故障，而其他组件还在运行。分布式系统的这个特征所带来的结果将是本书中一个反复提及的主题。

构造和使用分布式系统的动力来源于对资源共享的愿望。“资源”一词是相当抽象的，但它很好地描述了能在网络化计算机系统中共享的事物的范围。它涉及的范围从硬件组件如硬盘、打印机到软件定义的实体如文件、数据库和所有数据对象，包括来自数字摄像机的视频流和移动电话表示的声频连接。

2

本章的目的是给出分布式系统的本质，叙述保证分布式系统设计成功所面临的挑战。1.2节给出了分布式系统的一些重要的实例，构造系统所需的组件和它们的目的。1.3节给出了在万维网环境中资源共享系统的设计。1.4节描述了分布式系统设计者要面临的挑战：异构性、开放性、安全性、可伸缩性、故障处理、并发性和对透明性的要求。

1.2 分布式系统实例

本节选用的实例都是大家熟悉和广泛使用的计算机网络：因特网、企业内部网和正在兴起的基于移动设备的网络技术，这些系统说明计算机网络支持的服务和应用的范围很广。随后我们将从这些系统展开，讨论支撑系统实现的技术问题。

1.2.1 因特网

因特网是一个巨大的多种类型计算机网络的互联集合。图1-1给出了因特网的典型组成部分。因特网上的计算机程序通过传递消息交互，采用了一种公共的通信手段。因特网通信机制（因特网协议）是一项重大技术成果，它使得一个在某地运行的程序能给在任何其他地方的程序发送消息。

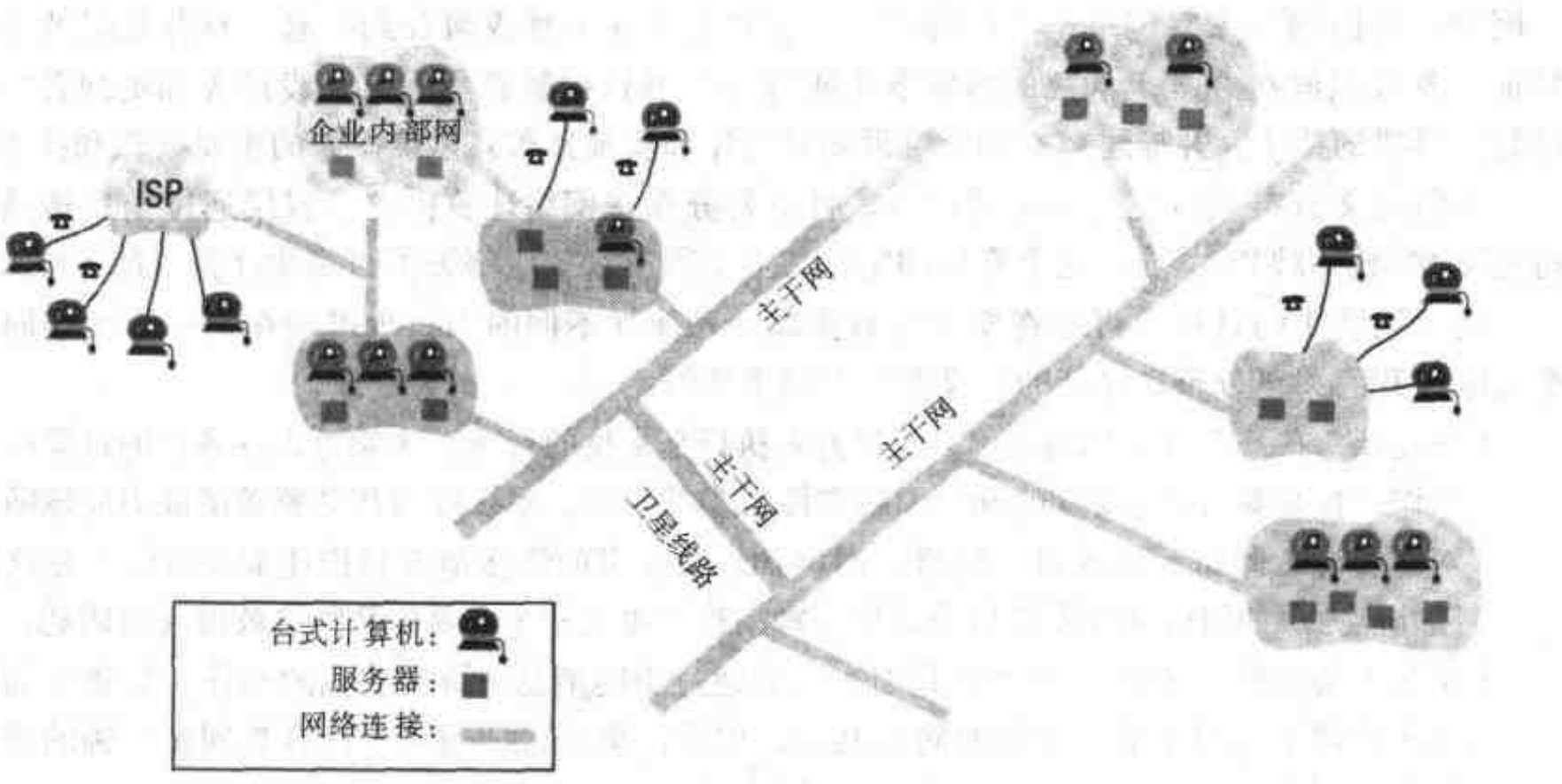


图1-1 因特网的典型组成部分

因特网也是一个非常大的分布式系统，它使得世界各地的用户能利用诸如万维网、电子

邮件和文件传输等服务。服务集是可伸缩的，它能够通过服务器计算机和新的服务类型的增加而扩展。图1-1还包含了许多企业内部网——由公司和其他组织控制的子网。因特网服务提供商（ISP）是为个体用户和小型组织提供调制解调器连接和其他类型连接的公司，通过ISP提供的连接，用户能够访问因特网上的服务，同时，ISP还提供诸如电子邮件和Web接入等本地服务。企业内部网通过主干网连接在一起。主干网是具有高传输性能的网络连接，通常采用卫星线路、光纤和其他宽带线路作为传输介质。

通过因特网还可以获得多媒体服务，它使用户可以访问包括音乐、电台和TV频道在内的音频和视频数据，还可以召开电话和视频会议。目前，因特网处理多媒体数据的特殊通信需求的能力还很有限，因为它没有提供必要的设施存储单个数据流的网络容量。第15章将讨论分布式多媒体系统的需求。

因特网和它支持的服务已经解决了许多分布式系统问题（包括在1.4节中定义的大多数问题）。本书将突出这些解决方案，并在适当的时候指出它们的适用范围和局限性。

1.2.2 企业内部网

企业内部网是因特网的一部分，实行独立管理，具有边界，通过配置能执行本地安全策略。图1-2给出了一个典型的企业内部网结构。它由几个通过主干网连接的局域网（LAN）组成。企业内部网的网络配置由管理企业内部网的组织负责，它的变化范围很广，可以仅仅由位于一个场所的LAN构成，也可以是由位于不同国家的同一公司的分支机构或公司之外的其他组织的多个LAN连接而成。

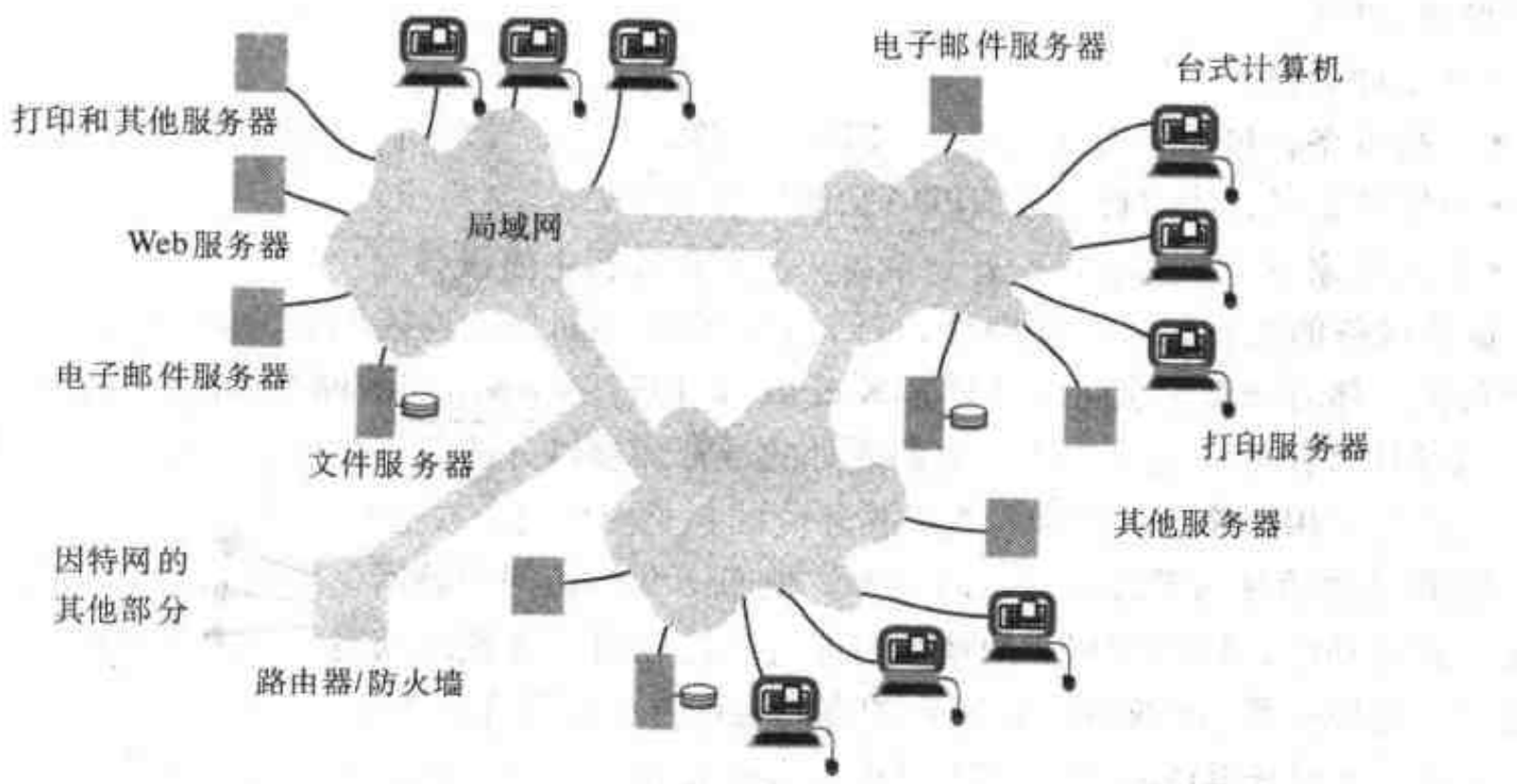


图1-2 典型的企业内部网

企业内部网通过路由器连接到因特网，这使得企业内部网内的用户能利用其他地方的服务如Web服务或电子邮件服务。它也允许其他企业内部网的用户访问它提供的服务。许多组织需要保护自己的服务，以免外部的恶意用户对其进行未经授权的使用。例如，公司不希望机密信息被竞争对手获取、医院不希望敏感的病人数据被曝光。公司还希望避免病毒等有害程序入侵并攻击企业内部网内的计算机，同时还要防止这些有害程序破坏有用的数据。

防火墙的作用是通过防止未授权消息发出或进入来保护企业内部网。防火墙是通过过滤进出消息实现其功能的，例如，根据消息的源地址或目的地址进行过滤。一个防火墙可能仅允许与电子邮件和Web访问有关的消息进出它保护的企业内部网。

一些组织并不希望将自己的内部网连接到因特网。例如，警察机构和其他安全法律执行机构可能有几个内部网与外部世界隔离，英国国家健康服务也选择了这样的观点，它保留了一个物理分离的内部网来保护敏感的病人数据。一些军事组织在战时会将它们的内部网与因特网断开，但这些组织仍然希望从大量的采用因特网通信协议的应用和系统软件中受益。它们通常采用的方案还是像上面描述的那样构造企业内部网，但不与因特网相连。这样一个企业内部网没有防火墙也行，或者说，它有最有效的防火墙，因为它与因特网没有任何物理连接。

在设计用于企业内部网的组件时，会遇到下列这些主要问题：

- 需要文件服务以便用户能共享数据，这方面的设计见第8章的讨论。
- 防火墙可能阻止对服务的合法访问。当需要在内部和外部之间共享资源时，防火墙必须增加小粒度的安全机制，详见第7章的讨论。
- 用于软件安装和支持的开销是一个重要的问题。通过使用诸如网络计算机和瘦客户机等系统体系结构，能减少这些开销，详细描述参见第2章。

3
~
5

1.2.3 移动计算和无处不在的计算

设备小型化和无线网络技术的进步已经逐步使小型和便携计算设备集成到分布式系统中。这些设备包括：

- 膝上计算机。
- 手持设备，包括个人数字助理（PDA）、移动电话、传呼机、摄像机和数字照相机。
- 可穿戴设备，诸如具有类似PDA功能的智能手表。
- 嵌入在家电，如洗衣机、音响系统、汽车和冰箱中的设备。

这些设备的大多数具有便携性，再加上它们在不同地方连接网络的能力，使得移动计算成为可能。移动计算（也叫游动计算[Kleinrock 1997, www.cooltown.hp.com]）是指用户在移动中执行计算任务的能力或访问他们所处的通常环境以外的位置的能力。在移动计算中，即使用户远离常用的企业内部网（指工作环境的企业内部网，或他们住处的企业内部网），他们仍然能够通过随身携带的设备访问资源。他们能继续访问因特网，继续访问企业内部网上的资源。在移动时，用户可利用的诸如打印机等能在附近方便地获得的资源的供应量正在不断地增加，这种在移动时使用邻近资源的方式也称为位置清楚的计算。

无处不在的计算[Weiser 1993]是指对多种在用户的物理环境（包括家庭、办公室和其他地方）中存在的小型的、便宜的计算设备的控制。术语“无处不在”意指小型计算设备最终将普及到现在的日常物品中而不被人注意，就是说，它们的计算行为将紧密地、透明地捆绑到它们的物理功能上。

各处的计算机只有在它们能相互通信时才变得有用。例如，用户通过家中的一个“通用远程控制设备”控制洗衣机和音响系统，而洗衣机在完成洗衣后能通过一个智能标记卡或手表呼叫用户。

无处不在的计算和移动计算有重叠的部分，因为从原理上讲，移动用户能从各处的计算

机中受益。但总的说来，它们是不同的。无处不在的计算只能在诸如家庭或医院这样单一的环境中使用户受益。类似地，移动计算可以在很多环境中发挥作用，即使环境中只涉及到常规的分立的计算机和设备，诸如膝上计算机和打印机等。

图1-3显示了一个用户正在访问一个组织。该图显示出用户本地的企业内部网和用户正在访问的企业内部网。两个企业内部网通过其他因特网相连。

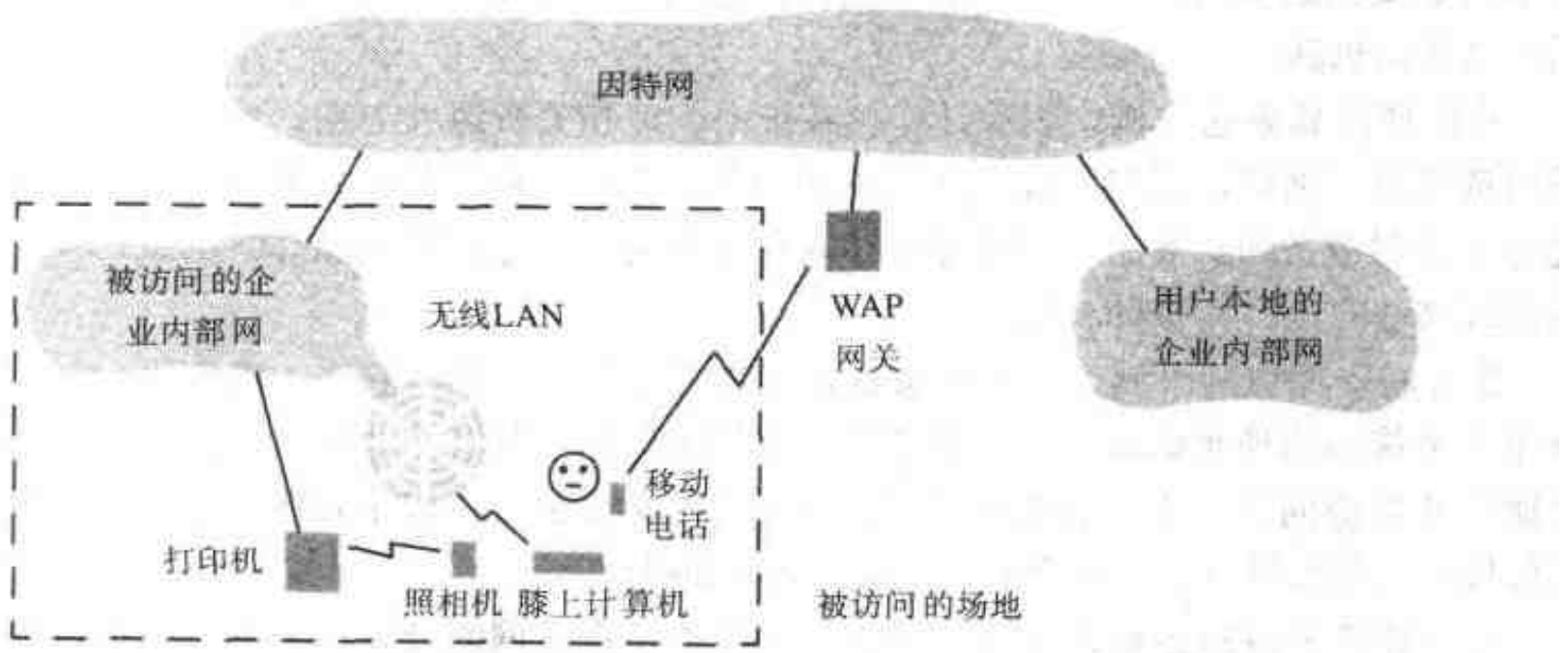


图1-3 分布式系统中的便携式设备和手持设备

用户可以使用3种方式进行无线连接。膝上计算机可以连接到被访问组织的无线LAN中。这个网络提供了几百米的覆盖范围（相当于建筑物的一层）。它通过网关连接到被访问的企业内部网的其余部分。用户还有一部移动电话，该电话利用无线应用协议（WAP）通过网关连接到因特网（见第3章）。电话可以访问简单的文本信息，按页显示在电话的显示屏上。用户携带有一台数字照相机，当它对准一个相应的设备，如打印机时，可以通过红外线连接与之通信。

在一个适宜系统基础构架内，用户能用他们携带的设备完成一些简单的任务。当用户到该地时，可以通过移动电话从Web服务器上取得最新的股票价格。在该地开会时，把数字照相机的照片直接发送到会议室一台可用的打印机上即可打印最近的照片，这仅仅要求在照相机和打印机之间具有红外连接。从原理上讲，不论利用无线LAN还是有线以太网连接，用户均能从膝上计算机把文件发送到同一台打印机。

移动计算和无处不在的计算产生许多值得注意的系统问题[Milojicic *et al.* 1999, p.266, Weiser 1993]。2.2.3节给出了移动计算的体系结构，列出了其中产生的问题，包括如何在主机环境中发现资源；如何避免用户在四处移动时需要重新配置移动设备；如何帮助用户解决旅游时连接受限的问题；如何为用户和他们访问的环境提供私密性和其他安全保证。

1.3 资源共享和Web

用户已经习惯了资源共享带来的好处，以致于很容易忽视它们的重要性。大家平常共享硬件资源如打印机，数据资源如文件，以及具有特定功能的资源如搜索引擎等。

从硬件资源的观点来看，大家共享设备如打印机和磁盘以减少花费。但对于用户来说，与他们的应用程序、日常工作和社会活动有关的更高层次的资源共享具有更重大的意义。例如，

用户关心共享数据库或Web页面方式的共享数据，而不是实现上述服务的硬盘和处理器。类似地，用户考虑诸如搜索引擎或货币转换器的共享资源，而不是关心提供这些服务的服务器。

实际上，资源共享的模式在工作范围和用户工作方式上变化很大。一个极端的例子是，Web上的搜索引擎给全世界的用户提供工具，使得用户并不需要直接接触。另一个极端的例子是，计算机支持协同工作（CSCW）中，若干直接进行合作的用户在一个小型封闭的组中共享诸如文档之类的资源。共享的模式和特定用户在地理上的分布决定了系统必须提供协调用户动作的机制。

我们使用服务这一概念表示计算机系统中管理相关资源并为用户和应用提供功能的单独的组成部分。例如，我们通过文件服务访问共享文件，通过打印服务发送文件到打印机，通过电子支付服务购买商品。对服务的访问仅仅是通过服务输出的操作。例如，一个文件服务提供对文件的读、写和删除操作。

服务将资源访问限制到一组定义良好的操作是属于标准的软件工程实践，但它也反映出分布式系统的物理组织结构。分布式系统的资源是物理封装在计算机内的，其他计算机仅通过通信才能访问它。为了有效地共享，每个资源必须由一个程序管理，通过这个程序提供的通信接口，资源可以获得可靠的、一致的访问和修改。

大多数读者很熟悉服务器这一术语，它指的是在联网的计算机上的一个运行程序（一个进程），这个程序接收来自其他计算机的请求，完成一个服务，并返回结果。发出请求的进程被称为客户。请求以消息的形式从客户端发送到服务器端，应答以消息的形式从服务器端发送到客户端。当客户发送一个要执行的操作请求，我们说，客户调用服务器上的操作。客户和服务器之间的完整交互，指从客户发送一个请求到它接收到服务器的应答，称为一个远程调用。

同一个进程可能既是客户又是服务器，因为服务器有时要调用其他服务器上的操作。术语“客户”和“服务器”仅仅应用于在一个请求中扮演的角色。就它们的不同点而言，客户是主动的，服务器是被动的；服务器是连续运行的，而客户只持续与其相关的那部分应用程序的时间。

注意，默认情况下，术语“客户”和“服务器”指的是进程而不是运行客户程序或服务器程序的计算机，虽然在日常用法中这些术语也指计算机自身。另一个不同（见第5章的讨论）是在用面向对象语言实现的分布式系统中，资源被封装成对象，由客户对象进行访问，这时，
8 称一个客户对象调用了一个服务器对象上的方法。

许多（但肯定不是所有的）分布式系统可以完全用客户和服务器的交互来构造。万维网、电子邮件和网络打印机都满足这种模式。第2章将讨论除客户－服务器系统之外的其他系统类型。

Web浏览器就是一个客户的例子。Web浏览器与Web服务器通信，从服务器上请求Web页面。下面将详细考察Web。

万维网

万维网[www.w3.org I, Berners-Lee 1991]是一个不断发展的系统，用于发布和访问因特网上的资源和服务。通过浏览器软件如Netscape和Internet Explorer，用户可使用Web获取和查看多种类型的文档，听音频流，看视频流或者与无限制的服务集进行交互。

Web是1989年在瑞士的欧洲原子能研究中心（CERN）出现的。作为物理学家之间在因特

网上交换文档的工具[Berners-Lee 1999]，Web的一个关键特征是它为所存储的文档提供了超文本结构，超文本结构反映了用户组织他们知识的要求，这意味着文档包含链接，链接指向其他也存储在Web上的文档和资源。

对Web用户来说，当他或她遇到文档中的一个图像或一段文字时，通常还可能伴随有到相关文档和其他资源的链接。链接的结构可以任意复杂，可加入的资源集是无限的，即链接的网确实是世界范围的。Bush[1945]在50年前就想像了超文本结构，因特网的发展使得这个想法能在世界范围内得以证实。

Web是一个开放的系统：它可以在不妨碍已有功能的前提下采用新的方法进行扩展和实现（见1.4.2节）。首先，它的操作是基于已公布的和广泛实现的通信标准和文档标准。例如，浏览器有多种类型，在多数情况下，每一种浏览器在几种平台上实现并且有多种Web服务器实现方式。一种构造的浏览器能从另一种构造的服务器中检索资源，所以，用户能访问大多数设备（从PDA到桌面计算机）上的浏览器。

第二，相对于能发布和共享的“资源”类型而言，Web是开放的。在Web上最简单的形式是一个Web页面或其他能保存在文件中并呈现给用户的内容，如程序文件、媒体文件、PS（PostScript）格式的文件和PDF（portable document format）格式的文档。如果有人发明了一种新的图像存储格式，那么这种格式的图像能马上在Web上发布。用户需要能够查看这种新格式图像的方法，而浏览器恰恰以“辅助程序”和“插件”的形式提供了新内容的显示功能。

Web已超越这些简单的数据资源而开始包含服务，如电子购物。虽然Web不断在发展，但它基本的体系结构没有改变。Web基于3个主要的标准技术组件：

9

- 超文本标记语言（HTML）是指定页面在Web浏览器上显示时内容和布局的说明语言。
- 统一资源定位符（URL）用于识别保存成Web某一部分的文档和其他资源。第9章讨论其他Web标识方式。
- 具有标准交互规则（超文本传送协议HTTP）的客户－服务器系统体系结构，其中交互规则用于浏览器和其他客户从Web服务器上获取文档和其他资源。图1-4显示了一些Web服务器和向它们发送请求的浏览器。用户可以定位并管理因特网上他们自己的Web服务器，这个特征很重要。

下面依次讨论这些组件，解释当用户选取Web页面并单击页面上的链接时，浏览器和Web服务器如何操作。

HTML 超文本标记语言[www.w3.org II]用于指定组成Web页面内容的文本和图像以及这些元素的布局和格式化，以便显示给用户。Web页面包含结构化的成分如标题、段落、表格和图像。HTML也用于指定链接和与链接相关联的资源。

用户可通过标准的文本编辑器手工生成HTML，或用能识别HTML的“所见即所得”型编辑器，根据用户给出的一个图示布局生成HTML。下面是一段典型的HTML正文：

```
<IMG SRC = "http://www.cdk3.net/WebExample/Images/earth.jpg">          1
<P>                                                                    2
Welcome to Earth! Visitors may also be interested in taking a look at the   3
<A HREF = "http://www.cdk3.net/WebExample/moon.html">Moon</A>.         4
<P>                                                                    5
(etcetera)                                                             6
```

这段HTML正文保存在一个文件中，例如*earth.html*，供Web服务器访问。浏览器从Web服务

器——本例中是一个名为*www.cdk3.net*的计算机上的服务器——获得这个文件的内容。浏览器读取从服务器返回的内容，把它变成格式化的文本和图像，以大家熟悉的方式放到Web页面上，仅由浏览器——不是服务器——来解释HTML文本。但是服务器确实把它返回的内容类型通知了浏览器，用于区分HTML文件和其他文件如PostScript文档。服务器能从文件的扩展名".html"中推断出内容类型。

HTML的指令即标记放在尖括号里，如*<P>*。例子中的第1行确定了一个包含显示图像的文件，它的URL是*http://www.cdk3.net/WebExample/Images/earth.jpg*。第2行和第5行都是一条开始新段落的指令，第3行和第6行包含一些要在Web页面上以标准的段落格式显示的文本。

第4行指定了Web页面上的一个链接，它包含的单词"Moon"位于两个相关的HTML标记*<A HREF…>*和*</A>*中间。这些标记之间的文本显示在Web页面上时出现在链接上。大多数浏览器可配置成在链接文本上加下划线，所以用户看到的上面的段落将是：

Welcome to Earth! Visitors may also be interested in taking a look at the Moon.

浏览器记录了链接的显示文本与包含在*<A HREF…>*标记中的URL之间的关联，在这个例子中是：

http://www.cdk3.net/WebExample/moon.html

当用户单击文本时，浏览器获取由相应URL标识的资源，并显示给用户。在这个例子中，资源是一个HTML文件，它指定了关于月球的一个Web页面。

URL 统一资源定位符[www.w3.org III]的目的是以浏览器能定位资源的方式标识资源。浏览器检查URL以便从Web服务器上取得相应的资源。有时用户在浏览器中键入一个URL，更常见的方式是用户单击一个链接或选择一个"书签"，或当浏览器获取一个嵌入Web页面的资源时（如一个图像），由浏览器查找相应的URL。

按绝对的完整的格式，每一个URL有两个顶层组件，即：

模式: 模式特定的位置

第一个成分"模式"声明了URL的类型。要求URL能指定多种资源的位置，也能指定多种通信协议。例如，*mailto:joe@anISP.net*标识出一个用户的电子邮件地址；*ftp://ftp.downloadIt.com/software/aProg.exe*标识一个用文件传输协议（FTP）获取而不是常用的HTTP协议获取的文件。模式的其他例子有"NNTP"（用于指定一个Usenet新闻组）和"Telnet"（用于登录到一台计算机）。

利用URL中的模式指示符，Web对于可访问的资源类型是开放的。如果有人发明了一种新的有用的"widget"资源——可能用它自己的地址模式定位widget，用它自己的协议访问widget——那么大家就能用"*widget:…*"格式的URL。当然，浏览器必须具备使用新的"widget"协议的能力，这一点可通过增加一个辅助程序或插件来完成。

最广泛使用的HTTP URL是利用标准的HTTP协议获取资源。一个HTTP URL有两个主要的工作：其一是识别出哪一个Web服务器维护着资源，其二是识别出服务器上的哪些资源是被请求的。图1-4显示了3个浏览器发出的资源请求，被请求的资源由3个Web服务器管理。最上面的浏览器在给一个搜索引擎发一个请求，第二个浏览器请求另一个Web站点的默认网页，最下面的浏览器请求一个指定了全名（包括相对于服务器的路径名）的Web页面。Web服务器的文件保存在服务器文件系统的一个或多个子树（目录）下，每一个资源用相对于服务器的文件路径名识别。

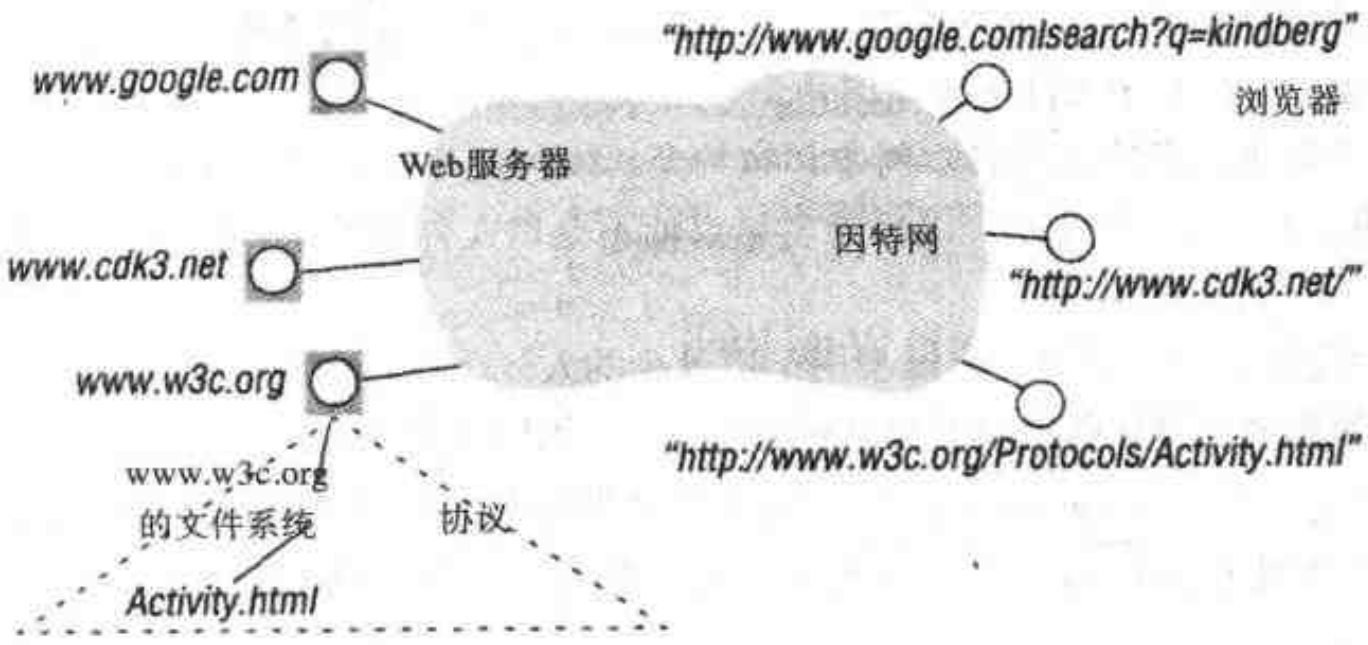


图1-4 Web服务器和Web浏览器

通常，HTTP URL的格式如下：

http://服务器名字[:端口][/服务器上的路径名][?参数]

其中方括号中的项可选。一个HTTP URL全名总是以“http://”开始的，后跟一个服务器名字，该服务器名表示成一个DNS（Domain Name Service）名（见9.2节）。服务器的DNS名后面可以加服务器监听请求的“端口”号（见第4章），接着是可选的服务器资源的路径名，如果缺少这一项，那么请求的是服务器的默认网页。最后，URL以一组可选的参数结束。例如，当一个用户提交诸如搜索引擎查询页的格式项时，URL就有参数项。

查看下列URL:

http://www.cdk3.net/

http://www.w3.org/Protocols/Activity.html

http://www.google.com/search?q=kindberg

上述URL可分解成如下部分：

服务器的DNS名字	服务器上的路径	参数
www.cdk3.net	（默认）	（无）
www.w3.org	Protocols/Activity.html	（无）
www.google.com	search	q=kindberg

第一个URL指定了由*www.cdk3.net*提供的默认网页。第二个URL指定了在*www.w3.org*服务器上的一个文件，该文件的路径名为*Protocols/Activity.html*。第三个URL指定一个对搜索引擎的查询。路径指定了一个叫“search”的程序，“？”字符后面的串是该程序的参数，本例中指定了查询串。在考虑更高级的特征时，我们将详细讨论表示程序的URL。

读者可能已经注意到了在URL的结尾出现有锚标志——带有“#”做前缀的名字，如“#references”，锚标志表示文件中的一个位置。锚标志不属于URL自身，但被定义成HTML规范的成分。仅由浏览器解释锚标志，显示从指定位置开始的Web页面。浏览器总是从服务器上检索整个Web页面，而不是检索由锚标志指示的部分页面。

发布资源　虽然Web有一个清晰的从URL检索资源的模型，但是在Web上发布资源的方法仍然是笨拙的，通常需要人工干预。为了在Web上发布资源，用户必须首先在Web服务器能访问的目录中放置相应的文件。用户知道了服务器*S*的名字和服务器能认识的文件*P*的路径名后，

10
)
12

才能构造像*http://S/P*这样的URL。用户可以把这个URL放在已有的文档中作为一个链接或将这个URL发给（例如，通过电子邮件）其他用户。

有一些服务器能识别约定的路径名。例如，按约定，以*~joe*开头的路径名是在用户joe主目录的子目录*public_html*下。类似地，以目录名结束而不是以文件名结束的路径名通常表示该目录中的*index.html*文件。

Huang等[2000]提供了一个模型，该模型用于以最少的人工干预将内容插入Web中。当用户需要从多种设备如照相机等抽取内容发布到Web页面时，该模型特别有用。

HTTP 超文本传输协议[www.w3.org IV]定义了浏览器和其他类型的客户与Web服务器的交互方式。第4章将详细考察HTTP，这里先给出它的主要特征（我们的讨论将限制在对文件资源的检索范围内）：

- *请求-应答交互* HTTP是一个“请求-应答”协议。客户发送一个请求消息到包含所请求资源的URL所在的服务器。（服务器仅需要遵循服务器自身DNS名字的部分URL。）服务器查找路径名，如果它存在，就在应答消息中将文件的内容返回给客户，否则，返回一个出错应答。
- *内容类型* 浏览器没有必要能够处理或利用每一种内容类型。当浏览器发出一个请求，其中包括它能够处理的内容类型的清单。例如，原则上它能够显示“GIF”格式的图像而不是“JPEG”格式的图像。服务器在将内容返回给浏览器时会考虑到这些。服务器在应答消息中包含了内容类型，所以浏览器能知道如何处理它。表示内容类型的串被称为MIME类型，这已在RFC 1521[Borenstein and Freed 1993]中被标准化。例如，如果内容是“text/html”类型，那么浏览器将把正文解释成HTML并显示；如果内容是“image/GIF”类型，那么浏览器将以GIF格式把该内容显示成图像；如果内容类型是“application/zip”，那么说明数据以“zip”格式压缩，浏览器将启动一个外部的辅助应用程序解压缩。浏览器可配置对指定的内容类型所采用的动作，读者可以检查自己浏览器的设置。
- *一次请求一个资源* 在HTTP 1.0版本中（写作本书时，HTTP 1.0是使用最广泛的版本），客户的一个HTTP请求只请求一个资源。如果Web页面包含了9个图像，那么浏览器要发出总共10个单独请求才能获得该页的整个内容。通常浏览器可同时发几个请求，以减少对用户的整个延迟。
- *简单的访问控制* 默认情况下，用户通过网络与Web服务器相连，能访问任何已发布的资源。如果用户希望限制对一个资源的访问，那么他可以配置服务器，对发请求的客户回发一个“质询”，相应的用户通过输入口令之类等操作证明他们有权限访问该资源。

13

更高级的特征——服务和动态网页 到目前为止，我们已经描述了用户如何在Web上发布Web页面和其他保存在文件中的内容。内容可能随时间而改变，但这对任何人都是一样的。然而大多数Web用户的经验是对用户能交互的服务的经验。例如，当在一个联机商店中购买一个物品时，用户经常填写一个Web表单，写明他们个人的细节或准确指定他们要买什么。Web表单是包含用户指令和诸如正文域、复选框等的窗口输入部件的Web页面。当用户提交表单（通常通过按下按钮或“返回”键）时，浏览器就发送一个HTTP请求到Web服务器，请求中包含了用户已经输入的值。

因为请求的结果取决于用户的输入，所以服务器必须处理用户的输入。因此，URL或它

的初始成分要指定服务器上的一个程序，而不是一个文件。如果用户的输入较短，那么它通常在“?”字符后作为URL的最后一个成分发送（否则它作为请求的额外数据发送）。例如，包含下列URL *http://www.google.com/search?q=kindberg*的请求调用www.google.com上的一个“search”程序并指定了一个查询串“kindberg”。

“search”程序产生HTML正文形式的输出，用户将看见包含“kindberg”单词的页的列表（读者可以在他们喜欢的搜索引擎中输入一个查询，注意查看在结果返回时浏览器显示的URL）。服务器返回程序生成的HTML正文与从文件中检索到的一样。换句话说，从一个文件中取的静态内容和动态生成的内容对浏览器都是透明的。

Web服务器运行为客户生成内容的程序通常称为公共网关接口（CGI）程序。只要CGI程序能分析客户提供给它的参数，并且能够产生所要求类型的内容（通常是HTML正文），一个CGI程序就可以具有任何特定的应用功能。程序在处理请求中会经常查询或修改数据库。

下载的代码　CGI程序在服务器上运行。有时Web服务的设计者要求在用户计算机端的浏览器内部运行一些与服务相关的代码。例如，用Javascript[www.netscape.com]写的代码经常被用于下载Web表单，以便提供比HTML标准部件质量更好的用户交互。Javascript改造过的页面对于非法项能马上反馈给用户（而不是在服务器端检查用户输入值，并且要花费较长的时间）。Javascript也能用于不取Web页面的全新版本而只更新或修改部分Web页面的内容。

Javascript的功能相对受限。相对而言，小程序是浏览器在取相应Web页面时能自动下载并运行的应用程序，小程序利用Java[java.sun.com, Flanagan 1997]语言功能访问网络，提供定制的用户界面。例如，有时用小程序实现“聊天”程序，此程序在用户的浏览器与服务器程序一起运行。小程序发送用户的正文给服务器，服务器依次把该正文分送给所有的小程序，用于对用户的显示。2.2.3节将详细讨论小程序。 14

对Web的讨论　Web巨大的成功在于通过它资源能很容易地发布，其超文本结构适合组织多种类型的信息，并且其系统结构具有开放性。Web体系结构所基于的标准很简单，这些标准在建立初期就被广泛地发布并且使得许多新的资源类型和服务集成在一起。

Web的成功掩盖了它的一些设计问题。首先，它的超文本模型在某些方面有缺陷。如果删除或移动了一个资源，那么仍然存在对资源的所谓的“悬空”链接问题，这会引起用户请求的落空。还存在用户“失落在超空间”中这个常见问题。用户经常发现跟随众多混乱互异的链接会到达完全不同的页面，在有些情况下其可靠性值得怀疑。搜索引擎作为在Web上查找信息的手段是对链接－跟随的补充，但它在满足用户某些特定的需求方面并不是十分完美的。按照资源描述框架中的说明[www.w3.org V]，对这个问题的解决方案是将Web资源的元数据格式标准化。元数据记录Web资源的属性，在诸如查找或编辑有关链接列表的时候，可由工具读取元数据，帮助那些一起处理Web资源的用户。

在Web上交换多种结构化数据的需求激增，但HTML是受限的，因为它对超出信息浏览的应用是不可伸缩的。HTML具有静态结构，如段落，它们与数据呈现给用户的方式是捆绑在一起的。最近，扩展标记语言（XML）[www.w3.org VI]已被设计成一种用标准的、结构化的、应用特定的格式表示数据的方式。例如，XML用于描述设备的能力以及用来描述用户的个人信息。XML是描述数据的元语言，它使得数据在应用之间可移植。扩展样式表语言[www.w3.org VII]描述了XML格式的数据如何呈现给用户。利用两种不同的样式表，在Web页面上表示的用户的同一信息能在某一种情况下图形化，在另一种情况下表示成简单的列表

形式。

作为系统体系结构，Web面临着规模扩展的问题。受欢迎的Web服务器在每一秒的间隔内都可能有多次“点击”，导致对用户的响应变慢。在第2章中描述了如何在浏览器和代理服务器中使用缓存的方法改善上述现象。但是，Web的客户－服务器结构意味着它还没有有效的手段让用户保留最新版的页面。用户不得不按浏览器的“重新载入”键来确保它们获得的是最新的信息，这时浏览器被强制与服务器通信，检查资源在本地的副本是否有效。

最后，一个Web页面不总是一个满意的用户界面。HTML定义的界面部件有限，设计者需要经常在Web页面上添加小程序或图像，使得页面外观和功能更具有可接收性。这会导致延
15 长下载的时间。

1.4 挑战

1.2节的例子试图说明分布式系统的范围，提出在设计中出现的问题。虽然分布式系统随处可见，但它们的设计相当简单，还存在很大的空间来开发更富有挑战性的服务和应用。本节讨论的许多要求已经被满足，但将来的设计者还需要了解它们，在设计分布式系统时还应仔细地考虑它们。

1.4.1 异构性

因特网使得用户能够在大量异构计算机和网络上访问服务和运行应用程序。下面这些系统均有异构性（即，相互有区别）：

- 网络
- 计算机硬件
- 操作系统
- 编程语言
- 由不同开发者完成的实现

虽然因特网由多种网络组成（如图1-1所示），但由于所有连接到因特网的计算机都使用因特网协议相互通信，因而屏蔽了它们的区别。例如，以太网中的计算机要在以太网上实现因特网协议，而在另一种网络上的计算机需要在自己的网络上实现因特网协议。我们将在第3章中解释如何在多种不同网络上实现因特网协议。

整数之类的数据类型在不同种类的硬件上可以有不同的表示方法。例如，整数的字节顺序就有两种方法表示。如果要在不同硬件上运行的两个程序之间进行消息交换，那么就要处理这些在表示上的不同。

因特网上所有计算机的操作系统都要实现因特网协议，但它们没必要提供相同的协议接口。例如，UNIX中消息交换的调用方法与Windows NT中的调用方法不同。

不同的编程语言对字符和数据结构采用不同的表示方法，如数组和记录。如果要求用不同语言编写的程序能够相互通信，就必须解决这些差异。

除非使用公共标准，否则不同开发者编写的程序不能相互通信，例如，网络通信和消息中原始数据项和数据结构的表示均要求使用公共标准。所以，要制定和采用像因特网协议那样的标准。

中间件 术语中间件是指一个软件层，它提供了一个编程抽象以及对底层网络、硬件、操

作系统和编程语言异构性的屏蔽。第4章、第5章和第17章描述的CORBA是一个中间件的例子。 16
有些中间件，如Java RMI（见第5章）仅支持一种编程语言。大多数中间件在因特网协议上实现，由这些协议屏蔽了底层网络的不同。所有的中间件都要解决操作系统和硬件的不同，如何做到这一点是第4章的主题。

除了解决异构性的问题，中间件还为服务器和分布式应用的编程人员提供了一致的计算模型。可能的模型包括远程对象调用、远程事件通知、远程SQL访问和分布式事务处理。例如，CORBA提供了远程对象调用，它允许运行在一台计算机上的程序中的对象调用运行在另一台计算机上的一个程序中的一个对象。它从实现上屏蔽了为了发送调用请求和应答，消息要通过网络传递的事实。

异构性和移动代码 术语移动代码用于指能从一台计算机发送到另一台计算机并在目标计算机上运行的代码，如Java小程序。因为计算机的指令集依赖于它的硬件，所以适合一种类型计算机硬件的机器码可能不适合在另一种计算机硬件上运行。例如，PC用户有时把可执行文件作为电子邮件的附件，以便于接收者能运行它，但接收者不能在Macintosh或Linux计算机上运行该程序。

虚拟机方法提供了一种使代码可在任何硬件上运行的方法：某种语言的编译器生成一台虚拟机的代码而不是某种硬件代码，例如，Java编译器生成Java虚拟机的代码，而为了使Java程序能运行，要在每种硬件上实现Java虚拟机。然而，Java这种解决方法并不能完全应用到其他语言编写的程序。

1.4.2 开放性

计算机系统的开放性决定系统是否能以各种不同的方式扩展和重现。分布式系统的开放性主要取决于新的资源共享服务能被增加和供多种客户程序使用的程度。

除非软件开发者能够获得系统组件的主要软件接口的规范和文档，否则无法达到开放性。因此，要发布主要接口。这个过程类似接口的标准化，由于官方的标准化程序通常麻烦且进度缓慢，因此，它经常绕过官方的标准化程序。

然而，发布接口仅是分布式系统增加和扩展服务的起点。对设计者的挑战是如何解决由不同人构造的且有多个组件组成的分布式系统的复杂性。

因特网协议的设计者引入了一系列所谓的“征求意见稿”，即RFC的文档，每个文档有一个数字标识。20世纪80年代早期出版了这一序列的因特网通信协议的规范；20世纪80年代中 17
期出版了在因特网上运行的应用规范，如文件传输规范、电子邮件规范和Telnet规范。这种方式一直在继续，从而形成了因特网技术文档的基础。除了协议规范，该序列还包含了讨论。可从[www.ietf.org]获得这些资料。最初发布的因特网通信协议构造了庞杂的因特网系统和应用，例如Web就是因特网最近新增的服务。RFC不是发布的惟一手段。例如，CORBA是通过一系列技术文档发布的，其中包括CORBA服务接口的完整规约，请参见[www.omg.org]。

按这种方式支持资源共享的系统称为开放的分布式系统，以强调它们是可伸缩的。通过增加计算机到网络中，可以实现系统在硬件级扩展，通过引入新的服务与重新实现旧的服务可实现在软件级上的扩展，从而使得应用程序能共享资源。开放系统常被提及的另一个优点是它们与独立经销商无关。

总结如下：

- 开放系统的特征是它们的主要接口是对外发布。
- 开放的分布式系统是基于提供一致的通信机制和已发布访问资源共享的接口。
- 开放的分布式系统能用不同经销商的异构硬件和软件构造，其每个组件与发布的标准之间的一致性要仔细测试和验证，以确保系统正常工作。

1.4.3 安全性

分布式系统维护的众多信息源对用户具有很高的内在价值，因此它们的安全性相当重要。信息源的安全有3个部分：机密性（防止泄漏给未授权的个人）、完整性（防止改变或讹误）、可用性（防止对访问资源的手段的干扰）。

1.1节指出虽然因特网允许一台计算机中的程序与另一台计算机上的程序通信，而不管它们的位置，但安全风险与允许自由访问企业内部网的所有资源相关。虽然防火墙能被用于形成围绕企业内部网的屏障，限制进入和外出的流量，但这不能确保用户恰当地使用企业内部网资源以及因特网的资源，而后一种资源不能被防火墙保护。

18

在分布式系统中，客户需要在网络上通过消息发送信息请求去访问由服务器管理的数据。例如：

1. 医生可能请求访问医院病人的数据或发送新增的病人数据。
2. 在电子商务和银行中，用户在因特网上发送信用卡号码。

上面两个例子所面临的挑战是以安全的方式在网络上通过消息发送敏感信息，但安全不仅是为消息的内容保密，它还涉及确切知道消息所代表的用户或其他代理的身份。在第一个例子中，服务器需要知道用户确实是一个医生。在第二个例子中，用户需要确保他们正在交易的商店或银行的身份。这里的第二个挑战如何是正确地识别远程用户或其他代理。利用加密技术可满足这两个挑战。因特网上广泛使用加密技术，我们将在第7章中讨论这些问题。

然而，下面两个安全挑战还没有完全满足：

- *拒绝服务攻击* 另一个安全问题是，用户可能因为某些原因希望中断服务。他们可用下面的方法实现这种情况：用大量无意义的请求攻击服务，使得重要的用户不能使用它，这称为拒绝服务攻击。在编写本书的时候（2000年初），就发生了几次对几个众所周知的Web服务进行的拒绝服务攻击事例。现在通过在事件后抓获和惩罚犯罪者来反击这种攻击，但这并不是这类问题的通用解决方法。正在开发基于改善的网络管理的反击手段，第3章将涉及这些问题。
- *移动代码的安全* 移动代码需要小心处理。设想从电子邮件附件中接收到一个可执行程序，运行该程序带来的后果是不可预测的。例如，它可能看似显示一幅有趣的画，但实际上它可能会访问本地资源，也可能是拒绝服务攻击的一部分。确保移动代码安全的一些手段将在第7章中概述。

1.4.4 可伸缩性

从小型企业内部网到因特网，分布式系统可在不同的规模的网络中有效地运转。如果资源数量和用户数量激增，系统仍能保持有效，那么该系统可被描述成可伸缩的。因特网就是一个计算机数量和服务动态增长的分布式系统的例子，图1-5给出了到1999年为止的20年间因特网上计算机数量的增加情况。图1-6给出了到1999年为止的Web发展的6年中计算机和Web服

务器的增加数量，见[info.isoc.org]。

日　期	计算机	Web服务器
1979年12月	188	0
1989年7月	130 000	0
1999年7月	56 218 000	5 560 866

图1-5　因特网上的计算机

可伸缩分布式系统的设计面临下列挑战：

日　期	计算机	Web服务器	百分比
1993年7月	1 776 000	130	0.008
1995年7月	6 642 000	23 500	0.4
1997年7月	19 540 000	1 203 096	6
1999年7月	56 218 000	6 598 697	12

图1-6　因特网上的计算机和Web服务器

- 控制物理资源的开销　当对资源的需求增加时，应该能以合理的开销扩展系统，从而满足该要求。例如，在企业内部网上文件被访问的频度可能随用户和计算机数量的增加而增加，如果一台文件服务器不能处理所有的访问请求，应该增加服务器数量，以避免可能发生的性能瓶颈现象。通常，要使有n个用户的系统成为可伸缩的系统，那么所需的物理资源数量应该至多为$O(n)$，即正比于n。例如，如果一个文件服务器能支持20个用户，那么两台这样的服务器应该支持40个用户。虽然这听起来很显然，但实际上并不容易达到，见第8章。
- 控制性能丢失　如果数据量的大小与系统中的用户或资源数量成正比，想一下如何管理这些数据，例如，记录计算机的域名和对应的域名系统持有因特网地址的对应表，这种表主要用于查找如*www.amazon.com*这样的DNS名字。采用层次结构的算法其伸缩性要好于采用线性结构的算法。但即便使用层次结构算法，数量的增加仍将导致一些性能的丢失：用于访问层次化的结构数据的时间是$O(log\ n)$，其中n是数据集的大小。对一个可伸缩的系统，最大的性能丢失应该不比这个差。
- 防止软件资源耗尽　用作因特网地址（因特网的计算机地址）的数字是一个缺乏伸缩性的例子。在20世纪70年代后期，决定用32位做因特网地址，但像第3章解释的那样，可用的因特网地址将在21世纪初被用尽。因此，新版的协议将使用128位的因特网地址。然而，公正地说，因特网的早期设计者对这个问题没有正确的答案。很难预测一个系统若干年后的需求。而且，过分地考虑将来增长的需求可能还不如到不得不变时再变，大的因特网地址会占据了额外的消息空间和计算机存储空间。
- 避免性能瓶颈　通常算法应该分散，避免性能瓶颈。我们用域名系统的前身来说明这一点。在以前的域名系统中，名字表被保留在一个主文件中，可被任何需要它的计算机下载。当因特网只有几百个用户时这是可行的，但这种方式不久便成了一个严重的性能和管理瓶颈。现在，域名系统将名字表分区，分散到因特网上的不同的服务器中，并采用本地管理的方式解决了这个瓶颈——见第3章和第9章的描述。

19～20

有些共享资源被非常频繁地访问；例如，许多用户访问同一Web页面，会引起性能

下降。第2章将介绍应用缓存和复制方法提高频繁使用的资源的性能的方法。

理想上，系统规模增加时系统和应用程序应该不需要改变，但这一点很难做到。伸缩问题是分布式系统开发的主要问题。本书将深入地讨论已经成功的技术，包括复制数据的使用（第8章和第14章）、缓存的相关技术（第2章和第8章）、部署多服务器处理常见的任务以使得几个类似的任务能并发地完成。

1.4.5 故障处理

计算机系统有时会发生故障。当硬件或软件发生故障时，程序可能产生不正确的结果或者在它们完成应该进行的计算之前就停止了。第2章将讨论并对在分布式系统的进程和网络中可能发生的故障类型进行分类。

分布式系统的故障是部分的——也就是说，有些组件正常运行时，有些组件却会发生故障。因此，故障的处理相当困难。本书将讨论下列处理故障的技术：

- 检测故障　有些故障能被检测。例如，校验和用于检测出错的消息数据或文件数据。第2章将解释该方法很难或甚至可能检测不到某些故障，如因特网上一台远程服务器的崩溃。面临的挑战是如何管理不能被检测但可以被怀疑的故障。
- 故障覆盖　有些被检测到的故障能被隐藏或降低危害性。下面是隐藏故障的两个例子：
  1. 在消息不能到达时进行重传。
  2. 将文件数据写入两个磁盘，这样如果一个磁盘坏了，另一个磁盘的数据仍是正确的。

  降低故障危害的例子是丢掉被损坏的消息——它应该被重传。读者可能意识到隐藏故障的技术在最坏情况下不能保证系统正常工作。例如，第二个磁盘上的数据可能也坏了，或消息无论怎样重传都不能在合理的时间内到达。
- 容错　因特网上的大多数服务确实会有故障——在这么大的网络上，这么多组件中试图检测和隐藏所有可能发生的故障是不太现实的。这些服务的客户端能设计成容错的，这通常也涉及到让用户参与到容错的动作中。例如，当Web浏览器不能与Web服务器相连，它不能让用户一直等待它与服务器建立连接，它会通知用户出现了这个问题，让用户自由选择是否以后再试。容错服务见下面关于冗余问题的讨论。

21

- 故障恢复　恢复涉及到软件的设计，以便在服务器崩溃后，永久数据的状态能被恢复或“回滚”。通常，在出现错误时，程序完成的计算是不完整的，被修改的永久数据（文件和其他保存在永久存储中的资料）可能处在一个不一致的状态。恢复的描述见第13章。
- 冗余　服务可以利用冗余的组件实现容错。考虑下面的例子：
  1. 在因特网的任意两个路由器之间，至少应该存在两个不同的路由。
  2. 在域名系统中，每个名字表至少被复制到两个不同的服务器上。
  3. 数据库可以复制到几个服务器上，保证在任何单个服务器出现故障后数据仍是可访问的；服务器被设计成能检测对方的错误，当检测到一个服务器上的错误时，客户就会被重新定向到剩下的服务器。

如何设计有效的技术以便保持快速变化的数据的副本最新而又不过度地丧失性能是一个挑战。具体方法见第14章的讨论。

对硬件故障，分布式系统提供了高可用性。系统的可用性是系统可用时间比例的度量。当分布式系统中的一个组件出现故障时，仅使用受损组件的那部分工作受到影响。如果用户

正在使用的计算机出现故障，用户可以移到另一台计算机上；服务器进程能在另一台计算机上启动。

1.4.6 并发

分布式系统中服务和应用两者均提供可被客户共享的资源。因此，可能几个客户同时试图访问一个共享的资源。例如，在接近拍卖最终期限时记录拍卖竞价的数据结构可能被非常频繁地访问。

管理共享资源的进程在一个时刻接受一个客户请求。但这种方法限制了吞吐量。因此服务和应用通常允许并发地处理多个客户的请求。为了详细说明此问题，假设每个资源被封装成一个对象，调用在并发线程中执行。在这种情况下，几个线程可能在一个对象内并发地执行，它们对于对象的操作可能相互冲突，产生不一致的结果。例如，如果拍卖中两个并发的 22
竞标是“Smith:122美元”和“Jones:111美元”，那么相应的操作在没有任何控制下可能是交叉进行的，它们可能被保存为“Smith:111美元”和“Jones:122美元”。

这个例子说明在分布式系统中代表共享资源的任何一个对象必须负责确保它在并发环境中操作正确，这不仅适用于服务器也适用于应用中的对象。因此，负责实现那些不用于分布式系统的对象的程序员必须确保对象在并发环境中能安全使用。

为了使对象在并发环境中能安全使用，对象的同步操作必须保持数据的一致性。这可通过标准的技术，如大多数操作系统使用的信号量技术来实现。这个话题以及对分布式共享对象集合的扩展见第6章和第12章的讨论。

1.4.7 透明性

透明性被定义成对用户和应用程序员屏蔽分布式系统的组件的分散性，系统被认为是一个整体，而不是独立组件的集合。透明性的含义对系统软件的设计有重大的影响。

ANSA参考手册[ANSA 1989]和国际标准化组织的开放分布式处理的参考模型（RM-ODP）[ISO 1992]识别了8种透明性。下面将解释原始的ANSA定义，并用移动透明性替换迁移透明性，前者的范围更广：

- 访问透明性　用相同的操作访问本地和远程的资源。
- 位置透明性　不需要知道资源的位置就能够访问它们。
- 并发透明性　几个进程能并发地对共享资源进行操作而互不干扰。
- 复制透明性　使用资源的多个实例增加可靠性和性能，而用户和应用程序员无需了解副本的存在。
- 故障透明性　屏蔽错误，不论是硬件故障还是软件组件故障，允许用户和应用程序完成它们的任务。
- 移动透明性　在不影响用户或程序的操作的前提下允许资源和客户在系统内移动。
- 性能透明性　当负载变化时，允许系统重构以提高性能。 23
- 伸缩透明性　在不改变系统结构或应用算法的前提下，允许系统和应用扩展。

两个最重要的透明性是访问透明性和位置透明性，它们的有无对分布式资源的利用有很大影响。有时它们一起被称为网络透明性。

作为访问透明性的说明，考虑具有文件夹的图形用户界面，无论文件夹中的文件在本地

还是在远程，图形用户界面是一样的。另一个例子是文件API，它用相同的操作访问本地和远程文件（见第8章）。作为没有访问透明性的例子，我们考虑这样一个分布式系统，除非用户利用FTP程序，否则系统不允许用户访问远程计算机上的文件。

Web资源名或URL是位置透明的，因为URL中识别Web服务器域名的部分指的是域中的计算机名字，而不是因特网地址。然而，URL不是移动透明的，因为某人的个人Web网页不能移动到另一个域中新的工作位置，这是因为其他页面上的所有链接仍将指向原来的页面。

通常，类似于URL等包括计算机域名的标识符妨碍了复制透明性。虽然DNS允许域名指向几台计算机，但它在查找名字时只选其中的一个。因为复制模式通常需要能够访问所有参加的计算机，它需要根据名字访问DNS条目中的每台计算机。

为了说明网络的透明性，考虑电子邮件地址，如*Fred.Flintstone@stoneit.com*的使用。该地址由用户名和域名组成。值得注意的是，虽然邮件程序将用户名接收为本地用户，但还是附加了本地域名。发送邮件给这样的用户不需知道他们的物理位置或网络位置。发送邮件消息的过程也不依赖接收者的位置。这样因特网中的电子邮件可提供位置透时性和访问透明性（即网络透明性）。

故障透明性也可以在电子邮件中说明，即使服务器或通信链接出了故障，邮件最终还是被投递了。通过一直重发直到邮件被成功传递这种方式实现屏蔽故障，即使这个过程花费了几天时间。中间件通常将网络和进程的故障转换成程序级的例外（见第5章的解释）。

为了说明移动透明性，考虑移动电话的情况。假设打电话的人和接电话的人都在一个国家的不同地方乘火车旅行，即从一个环境（蜂窝）移到另一个环境。我们将打电话人的电话作为客户，接电话人的电话作为一个资源，两个电话用户并不清楚电话（客户和资源）在蜂窝之间的移动。

透明性对用户和应用程序员隐藏了与手头任务无直接关系的资源，并匿名使用。例如，通常相似的硬件资源应该被交替分配以完成任务，用于执行一个进程的处理器身份通常对用户隐藏并一直处于匿名状态。但并不是总要求这样，例如，旅行者将便携计算机连到所到访的办公室的局域网，他应该能利用本地服务如发送邮件服务，以及利用本地的服务器等服务。甚至，如果在一个建筑物内，将要打印的文件送到一台特定的被命名的打印机打印也是很正常的，当然，通常是靠近用户的那台打印机。对开发并行程序的程序员而言，并不是所有的处理器都是匿名的。他或她可能对用于执行并行程序的处理器感兴趣，至少应关心它们的数量和拓扑结构。

24

1.5 小结

分布式系统随处可见。因特网使得全世界的用户无论身在何地都能访问因特网的服务。每个组织管理一个企业内部网，为本地用户提供本地服务和因特网服务，同时也为因特网上其他用户提供服务。小型的分布式系统可用移动计算机和其他小型的可连接到无线网的计算设备构造。

资源共享是构造分布式系统的主要动力。诸如打印机、文件、Web页面或数据库记录等资源均由相应类型的服务器管理。例如，Web服务器管理Web页面和其他Web资源。资源由客户访问，Web服务器的客户通常称为浏览器。

分布式系统的构造产生了许多挑战：

- 异构性　分布式系统必须基于多种不同的网络、操作系统、计算机硬件和编程语言构造。

因特网通信协议屏蔽了网络的不同，中间件能处理其他的不同。

- 开放性　分布式系统应该可伸缩，第一步是发布组件的接口，但由不同程序员编写的组件的集成是一个真正的挑战。
- 安全性　加密用于给共享资源提供适当的保护，在网络上用消息传送敏感信息时，加密用于给敏感信息保密。拒绝服务攻击仍然是一个有待解决的问题。
- 可伸缩性　就必须要增加的资源而言，如果分布式系统增加一个用户的开销是一个常量，那么这个分布式系统是可伸缩的。用于访问共享数据的算法应该避免性能瓶颈，数据应该表示成层次结构以获得最好的访问时间。频繁访问的数据能被复制。
- 故障处理　任一进程、计算机或网络都可能独立地发生故障。因此每个组件需要清楚所依赖的组件可能出现故障的方式，组件要设计成能适当地处理每个故障。
- 并发性　分布式系统中多个用户的出现是对资源发生并发请求的根源。每个资源必须被设计成在并发环境中是安全的。 25
- 透明性　目的是使得分布的某些特性对应用程序员具有不可见性，这样应用程序员只需要关心特定应用的设计问题。例如，他们不需要关心它的位置或它的操作如何被其他组件访问的细节，或它是否被复制或迁移。甚至网络和进程故障也可以以例外的方式呈现给应用程序员，当然他们必须处理这些例外。

练习

1.1　给出能被共享的5种类型的硬件资源和5种类型的数据或软件资源。给出它们在实际的分布式系统中发生共享的例子。

1.2　在不参考外部时间源的情况下，由局域网连接的两台计算机的时钟如何实现同步？什么因素限制了你所描述的过程的精确性？由因特网连接的大量的计算机时钟是如何实现同步的？讨论那个过程的精确性。

1.3　一个随身携带能无线联网PDA的用户到达一个她以前从没有到过的火车站。请给出建议：在用户不输入火车站的名字或属性的情况下，如何提供关于本地服务和火车站环境的情况？要克服哪些技术挑战？

1.4　作为信息浏览的核心技术，HTML、URL和HTTP各自的优势和不足是什么？通常，这些技术中的哪一个适合作为客户－服务器计算的基础？

1.5　用万维网举例说明资源共享、客户和服务器的概念。

万维网和其他服务的资源用URL命名，首字母缩略语URL是指什么？给出能用URL命名的3种不同的Web资源例子。

1.6　给出一个URL的例子。

列出URL的3个主要成分，叙述如何表示它们的边界，并用例子说明每个成分。

在什么程度上URL是位置透明的？

1.7　用某种语言（例如C++）编写的一个服务器程序提供了一个BLOB对象的实现，该对象用于由不同语言编写（例如Java）的客户访问。客户和服务器计算机可能是不同的硬件，但它们都连在因特网上。描述由于存在异构性的5个方面，要使客户对象能够调用服务器对象上的方法，需要解决哪些问题？ 26

1.8　一个开放的分布式系统允许新的资源共享服务，诸如练习1.7中的BLOB对象能被加

入和被多种客户程序访问。讨论在这个例子中，在什么范围内开放性的需求与异构性的需求不同。

1.9 假设BLOB对象的操作分成两类，对所有用户开放的公共操作和只对某些命名用户开放的保护操作。叙述为确保只有命名用户能使用保护操作必须考虑的所有问题。假设访问一个保护操作提供了不能对所有的用户公开的信息，那么会引起什么问题？

1.10 INFO服务管理了一组可能非常大的资源集，每一个资源都能通过因特网利用关键字（一个字符串名字）被用户访问。讨论资源名字的设计方法，使得在服务中的资源数量增加时性能的损失最小。对INFO服务的实现提出建议，以避免在用户数量变得很大时出现性能瓶颈。

1.11 列出在客户进程调用服务器对象的方法时可能出现故障的3个主要的软件组件，每一种情况给出一个故障例子。对组件的设计给出建议，使得它能容忍彼此的故障。

1.12 一个服务器进程维护一个共享的信息对象，诸如练习1.7中的BLOB对象。请你就允许客户请求在服务器上并发执行给出赞成和反对的意见。在它们并发执行的情况下，给出可能在不同客户操作之间发生“干扰”的例子，建议如何避免这种干扰。

27

1.13 对于一个由几个服务器实现的服务，解释为什么资源要在它们之间传输。采用将所有的请求组播到服务器组以获得客户的移动透明性的方法，是否能获得满意的效果？

第2章 系统模型

2.1 简介
2.2 体系结构模型
2.3 基础模型
2.4 小结

分布式系统的体系结构模型涉及系统各部分的位置和它们之间的关系。客户－服务器模型和对等进程模型均为体系结构的例子。客户－服务器模型可以有以下方面的修改：

- 在合作的服务器上进行数据分区或复制
- 由代理服务器和客户进行数据缓存
- 使用移动代码和移动代理
- 以便利的方式增加和删除移动设备

基础模型涉及对所有体系结构模型中公共属性的一种较形式化的描述。

分布式系统没有全局时间，所以，不同计算机上的时钟不必给出相同的时间。进程间的所有通信通过消息实现。计算机网络上的消息通信会受延迟的影响，也会遇到多种故障，并且容易受到安全性攻击。这些问题可以由下面3个模型描述：

- 交互模型处理消息发送等的性能问题，解决在分布式系统中设置时间限制的难题。
- 故障模型试图给出进程和信道故障的一个精确的规约。它定义了什么是可靠通信和正确的进程。
- 安全模型讨论了对进程和信道的各种可能的威胁。它引入了安全通道的概念，这可以保证在上述威胁下通信的安全。

28
~
29

2.1 简介

在实际环境中使用的系统无论在何种可能的环境下，无论面对何种可能的困难和威胁（本节文本框中的文字给出了一些例子），都应该能保证其功能的正确性。第1章的讨论和举例表明不同类型的分布式系统相互共享重要的基本特性，并且引发公共的设计问题。本章以描述型模型的形式给出分布式系统的公共特性和设计问题，每个模型试图对分布式系统设计的一个相关方面给出一个抽象、简化但是一致的描述。

体系结构模型定义了系统中组件相互交互方式以及它们映射到计算机基础网络的方式。2.2节描述了分布式系统软件的分层结构和若干决定组件位置和交互方式的体系结构模型。我们首先讨论了客户－服务器模型的变种，包括使用移动代码的系统；接着考虑便于增加、删除移动设备的分布式系统特征；最后查看分布式系统的一般设计需求。

2.3节介绍了3个基础模型，帮助分布式系统设计者揭示某些关键问题。目的是指出开发正确、可靠、安全的分布式系统必须要解决的设计问题、困难和挑战。这些基础模型对影响分布式系统的可依赖性（包括正确性，可靠性和安全性）特征提供了抽象的描述。

分布式系统的困难和挑战 下面是分布式系统设计者要面对的一些问题：

使用模式多样性 系统的组件会承受不同的工作负载，例如，有些Web页面每天会有几百万次的访问量。系统的有些部分可能断连或连接不稳定，例如当系统中包括移动计算机时。一些应用对通信带宽和延迟有特殊的需求，例如，多媒体应用。

系统环境的多样性 分布式系统必须适应异构的硬件、操作系统和网络。网络可能在性能上有很大不同——无线网的速度只有局域网的一小部分。系统必须能支持不同的规模——从几十台计算机到上百万台计算机。

内部问题 包括非同步的时钟、冲突的数据修改、多种涉及系统单个组件的软硬件故障模式。

30

外部问题 包括对数据完整性、私密性的攻击以及拒绝服务。

2.2 体系结构模型

一个系统的体系结构是用指定组件表示的结构，其整体目标是确保结构能满足现在和将来可能的需求，主要关心的是如何使系统可靠、可管理、可适应和低成本高效益。这与一个建筑物的体系结构设计有类似的方面——它不仅仅决定了建筑物的外观，而且决定了建筑物的总体结构，体系结构风格（哥特式、新古典式、现代式）为设计提供了一个一致的参考框架。

本节将描述分布式系统采用的几种主要的体系结构模型——分布式系统的体系结构风格。我们将根据第1章中介绍的对象和进程的概念建立体系结构模型。一个分布式系统的体系结构模型将简化和抽象分布式系统单个组件的功能，并考虑：

- 在计算机网络上的组件的放置，即为数据和负载分布寻找有用的模式定义。
- 组件之间的关系，就是说，组件的功能角色和组件之间通信的模式。

最初的简化是将进程分成服务器进程、客户进程和对等进程。对等进程是以对称方式进行协作和通信以完成任务的进程。进程分类用于识别每类进程的责任，因此，有助于评估它们的负载并决定每个进程出现故障时可能带来的影响。该分析结果可用于指定进程的放置，使进程的分布能满足目标系统的性能和可靠性目标。

一些动态性更强的系统能作为客户－服务器模型的变种而创建：

- 从一个进程到另一个进程的代码移动允许一个进程代理另一个进程的任务。例如，客户可以从服务器下载代码，在本地运行。对象和访问对象的代码可以移动以减少访问延迟和最小化通信流量。
- 一些分布式系统的设计使得计算机和其他移动设备可以无缝地增加或删除，允许它们发现可用的服务并为其他服务提供它们自己的服务。

已经有一些广泛使用的模式用于分布式系统的工作分配，它们对目标系统的性能和有效性有重要的影响。在计算机网络中组成分布式系统的进程的实际分配也受到许多具体的性能问题、可靠性问题、安全问题和代价问题的影响。此处描述的体系结构模型仅提供重要的分布模式的一个简化的观点。

2.2.1 软件层

术语软件体系结构原指在一台计算机中软件的分层或模块结构，近来更多地指同一计算

机或不同计算机上进程之间所请求和提供的服务，这个面向进程和面向服务的观点能用服务层表达。图2-1表达了这个观点，详细的阐述见第3章～第6章。服务器是一个接收其他进程请求的进程。一个分布式服务可由一个或多个服务器进程提供，这些进程相互交互，并与客户进程交互，维护该服务在系统范围内关于资源的一致视图。例如，在因特网上基于网络时间协议（NTP）可实现一个网络时间服务，其中运行在主机上的服务器进程将当前的时间提供给任何发出请求的客户，作为与服务器交互的结果，客户调整它们的当前时间。 31

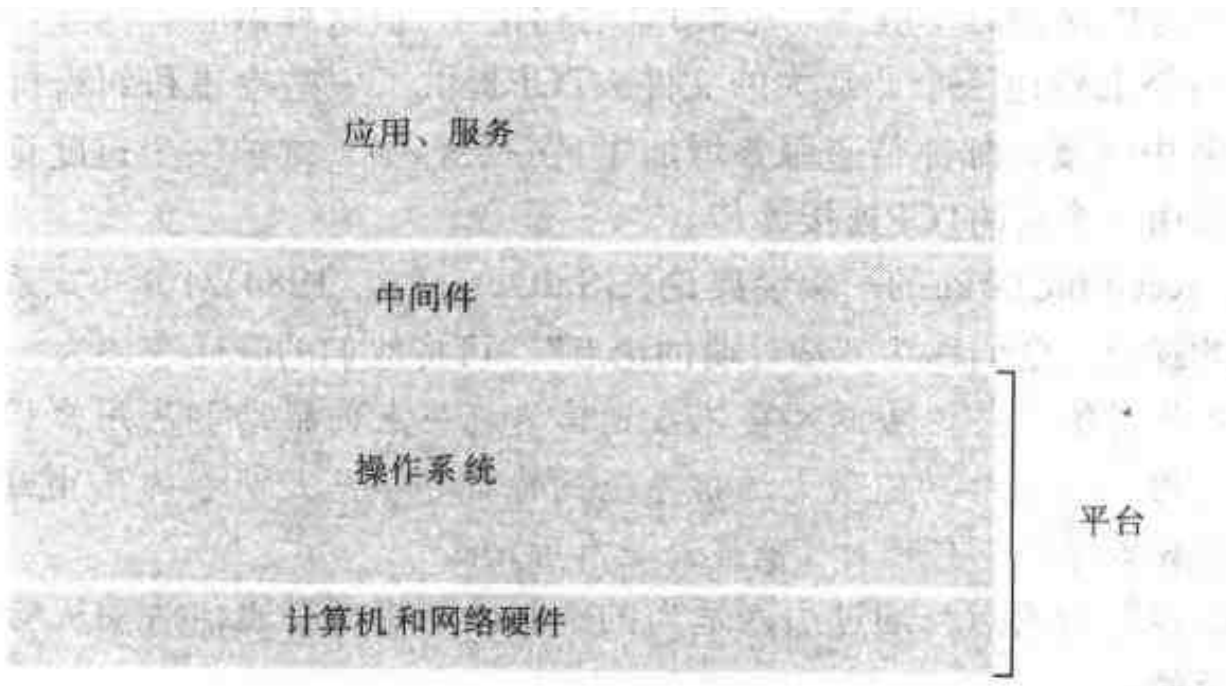


图2-1 分布式系统中的软件和硬件服务层

图2-1引入了重要的术语平台和中间件，具体定义见如下描述：

平台 最底层的硬件和软件层通常被称为分布式系统和应用的平台。这些底层向上层提供服务，它们在每个计算机中都是独立实现的，它们提供系统的编程接口，方便进程之间的通信和协调。主要的例子有：Intel x86/Windows、Sun SPARC/SunOS、Intel x86/Solaris、PowerPC/MacOS和Intel x86/Linux。

中间件 1.4.1节把中间件定义成一个软件层，它的目的是屏蔽异构性，为应用程序员提供方便的编程模型。中间件表示成一组计算机上的进程或对象，它们相互交互，实现分布式应用的通信和资源共享支持。中间件可以提供有用的构造模块，构造在分布式系统中一起工作的软件组件。特别地，它通过对抽象的支持，如远程方法调用、进程组之间的通信、事件的通知、共享数据的复制、多媒体数据的实时传送，提高了应用程序通信活动的层次。第4章介绍组通信，第11章和第13章还有详细的讨论，第5章描述事件通知。第14章讨论数据复制，第15章讨论多媒体系统。 32

早期和当前最广泛开发的中间件实例有远程过程调用包（如Sun RPC（见第5章描述））和组通信系统（如Isis（第14章））。面向对象中间件产品和标准包括OMG的公共对象请求代理体系结构（CORBA）、Java远程方法调用（RMI）、Microsoft的分布式公共对象模型（DCOM）和ISO/ITU-T的开放分布式处理的参考模型（RM-ODP）。CORBA和Java RMI的描述见第5章和第17章；DCOM和RM-ODP的细节可参见Redmond [1997]以及Blair和Stefani [1997]。

中间件还能提供应用程序使用的服务。这些服务是基础服务，与中间件提供的分布式编程模型紧密绑定。例如，CORBA提供了许多给应用提供方便的服务，如命名、安全、事务、永久存储和事件通知。第17章讨论了一些CORBA服务。图2-1顶层的服务是特定领域的服务，它利用了中间件的通信和中间件自己的服务。

中间件的限制　许多分布式应用完全依赖中间件提供的服务，用以支持它们对通信和数据共享的要求。例如，适合客户－服务器模型的应用（如名字和地址的数据库）能依赖提供远程方法调用的中间件。

通过利用中间件功能，在简化分布式系统编程方面已经获得很多成效，但系统可依赖性的某些方面需要应用层的支持。

考虑大的电子邮件消息从发送方的邮件主机传递到接收方的邮件主机。乍一看，这是一个TCP数据传送协议的简单应用（见第3章的讨论）。但这样做会有问题：用户试图在一个可能不可靠的网络上传递一个非常大的文件。TCP提供了一些检错和纠错机制，但它不能从严重的网络中断中恢复。邮件传递服务增加了另一层容错，维护一个进度记录，如果原来的连接断了，就利用一个新的TCP连接续传。

Saltzer、Reed和Clarke的一篇经典论文[Saltzer *et al.* 1984]对分布式系统的设计给出了类似的有价值的观点，他们称为“端对端的论点”。可将他们的陈述表述为：

> 一些与通信相关的功能只需依靠通信系统终点的帮助和应用知识就能完全可靠地实现。因此将这些功能做为通信系统的特征并不总是明智的（由通信系统提供一个不完全版本的功能有时对性能提高是有用的）。

33 可以看出他们的论点与通过引入适当的中间件层将所有通信活动从应用编程中抽象出来的观点是相反的。

论点的关键是分布式程序正确的行为在很多层上依赖检查、纠错机制和安全手段，其中有些要访问应用地址空间的数据。任何企图在通信系统中单独完成的检查将只能保证部分所要求的正确性。因此，同样的任务可能在应用程序中重复，这不仅浪费了编程，更为重要的是，增加了不必要的复杂性，增加了冗余的计算。

此处限于篇幅，不再进一步详细讨论他们的论点；强烈推荐读者阅读所引证的文章，那里有许多例子说明。原文作者之一最近指出：为满足当前应用需求而转向网络服务专门化的趋向可能替代该论点给因特网设计带来的实质性好处[www.reed.com]。

2.2.2 系统体系结构

系统组件（应用，服务器和其他进程）之间责任的划分和网络上计算机组件的放置可能是分布式系统设计最明显的方面。这些对最终系统的性能、可靠性和安全性有较大的影响。本节给出主要的体系结构模型，责任的分布将基于这些模型。

在分布式系统中，具有明确责任的进程相互交互完成有用的活动。本节我们的注意力将聚焦在进程的放置上，并按图2-2的方式给出计算机（灰方框）中进程（椭圆）的部署。我们使用术语“调用”和“结果”来标注消息，它们也能标注成“请求”和“应答”。

体系结构模型的主要类型见图2-2~图2-5的图示和以下的描述。

客户－服务器模型　这是讨论分布式系统时最常引用的体系结构，也是历史上最重要的体系结构，现在仍被最广泛地使用。图2-2给出了客户访问由服务器管理的共享资源时，客户进程与在单独主机上的服务器进程交互的简单结构。

服务器也可以是其他服务器的客户，如图中所示。例如，Web服务器通常是管理存储Web页面文件的本地文件服务器的客户。Web服务器和大多数其他因特网服务是DNS服务的客户，DNS服务用于将因特网域名翻译成网络地址。另一个与Web相关的例子是搜索引擎，搜索引

擎能让用户通过因特网查找Web页面上可用的信息总汇。这些信息通过称为“网络爬虫”的程序形成，此程序在搜索引擎站点以后台方式运行，通过因特网利用HTTP请求访问Web服务器。这里，搜索引擎既是服务器又是客户：它回答来自浏览器客户的查询，它又运行作为其他Web服务器客户的网络爬虫程序。在这个例子中，服务器任务（对用户查询的回答）和网络爬虫的任务（向其他Web服务器发请求）是完全独立的，很少需要它们同步，它们可以并行运行。事实上，一个典型的搜索引擎正常情况下包含许多并发执行的线程，一些线程为它的客户服务，另一些线程运行网络爬虫。练习2.4将要求读者考虑这种类型的并发搜索引擎可能出现的同步问题。

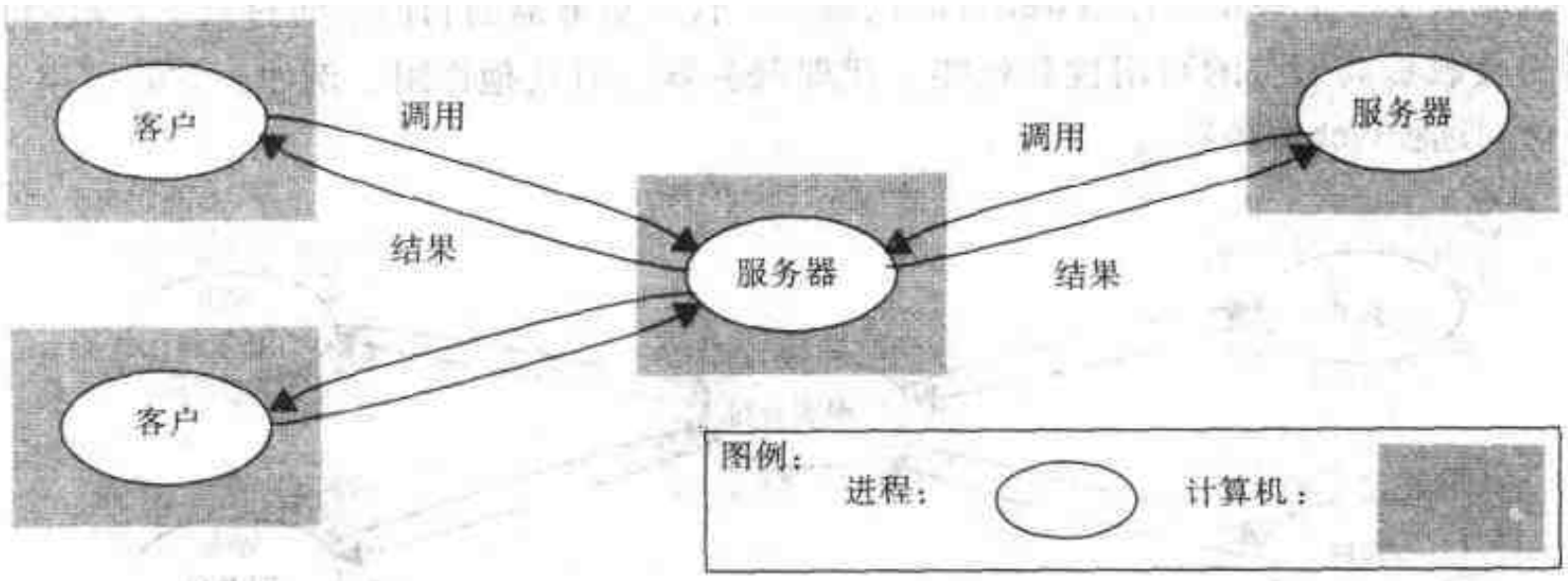


图2-2 客户－服务器模型

由多个服务器提供的服务 与客户进行必要交互的几个服务器进程可在一个单独主机上实现，以便给客户进程提供服务（如图2-3所示）。服务器可以将服务所基于的对象集分区，在它们之间分布对象，或者服务器可以在几个主机上维护复制的服务。这两种选择可用下列例子说明。

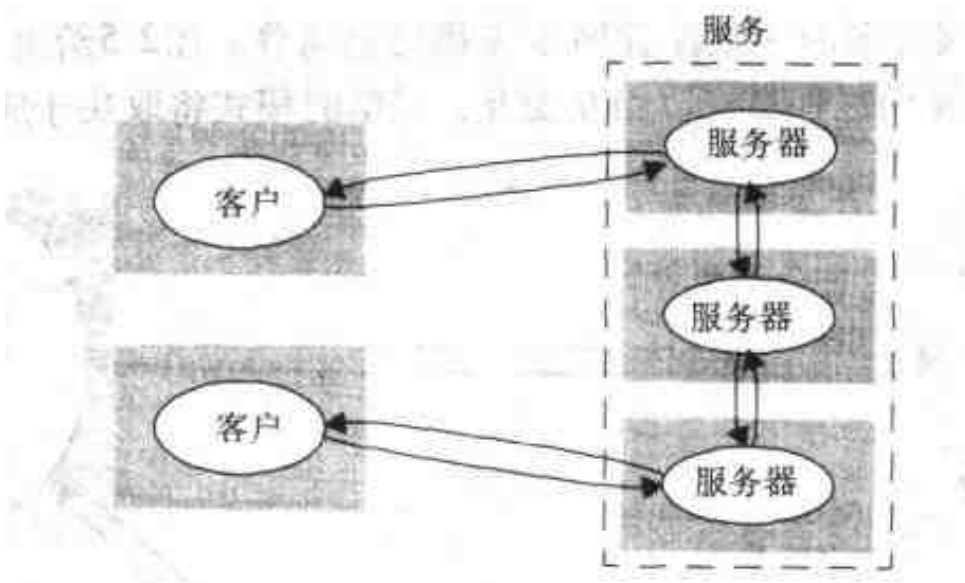


图2-3 由多个服务器提供的服务

Web就是一个常见的将数据分区的例子，这里，每个Web服务器管理自己的资源集。用户利用浏览器访问任一个服务器上的资源。

复制可用于提高性能、可用性与容错能力。它提供了多个一致的数据副本，这些数据分布在不同计算机的进程中。例如，*altavista.digital.com*提供的Web服务被映射到几个在内存复制了数据库的服务器上。

34
|
35

另一个基于复制数据的服务是Sun NIS（网络信息服务），在用户登录时，LAN中的计算机使用该服务。每个NIS服务器有它自己的口令文件副本，该副本包含了用户登录名和加密的

口令列表。在第14章详细讨论复制技术。

代理服务器和缓存 *缓存*存储最近使用的数据对象。当计算机接收到一个新的对象时，它就被加了缓存，必要的时候要替换一些已存在的对象。当客户进程需要一个对象时，缓存服务首先检查缓存。如果缓存中有最新的副本，则可直接提供，如果没有可用的，才去取一个最新的副本。每个客户都可以配置缓存或者由几个客户共享的代理服务器完成该功能。

缓存在实践中被广泛使用。Web浏览器维护一个缓存，它在客户本地的文件系统中存放最近访问的Web页面和其他Web资源，并在显示之前用一个特殊的HTTP请求到原来的服务器上检查被缓存的页面是否是最新的。Web代理服务器（如图2-4所示）为一个站点或多个站点的客户机提供了一个共享的存放Web资源的缓存。代理服务器的目的是通过减少广域网和Web服务器的负载提高服务的可用性和性能。代理服务器还有其他作用，例如，它们可用于通过防火墙访问远程Web服务器。

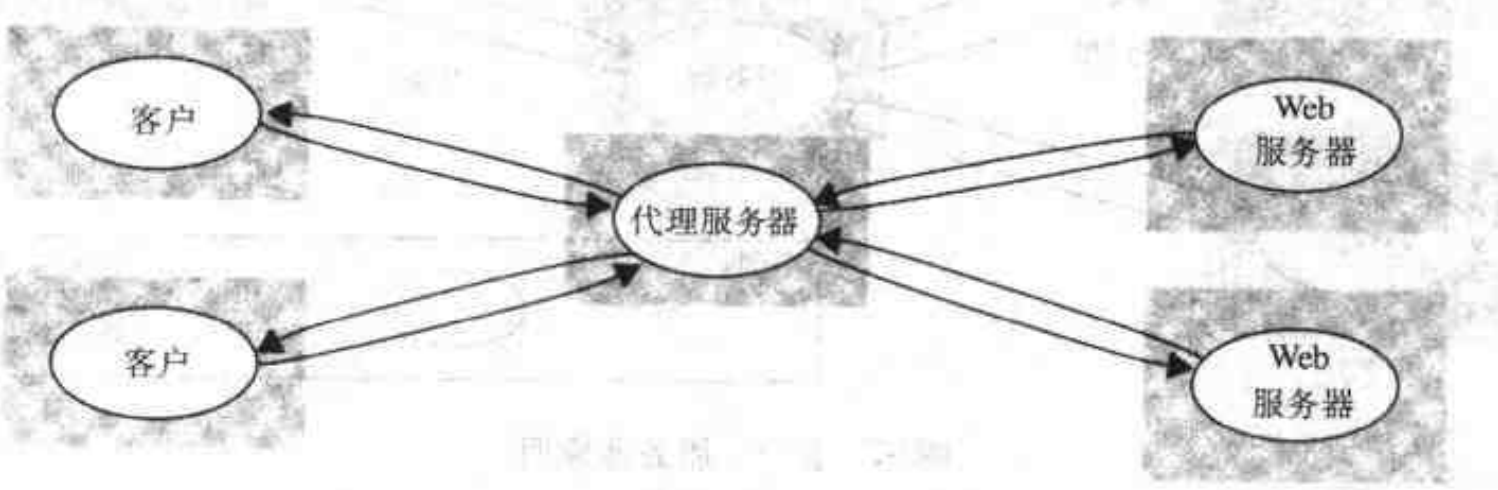


图2-4 Web代理服务器

对等的进程 在这种体系结构中，为了完成一项分布式活动或计算，所有的进程扮演相同的角色，作为对等方进行协作交互，不区分客户和服务器。在这种模型中，对等进程中的代码在必要时维护应用层资源的一致性并同步应用层的动作。图2-5给出了这种结构中3个结点的例子，通常，*n*个对等的进程都可以相互交互，通信的模式将取决于应用需求。

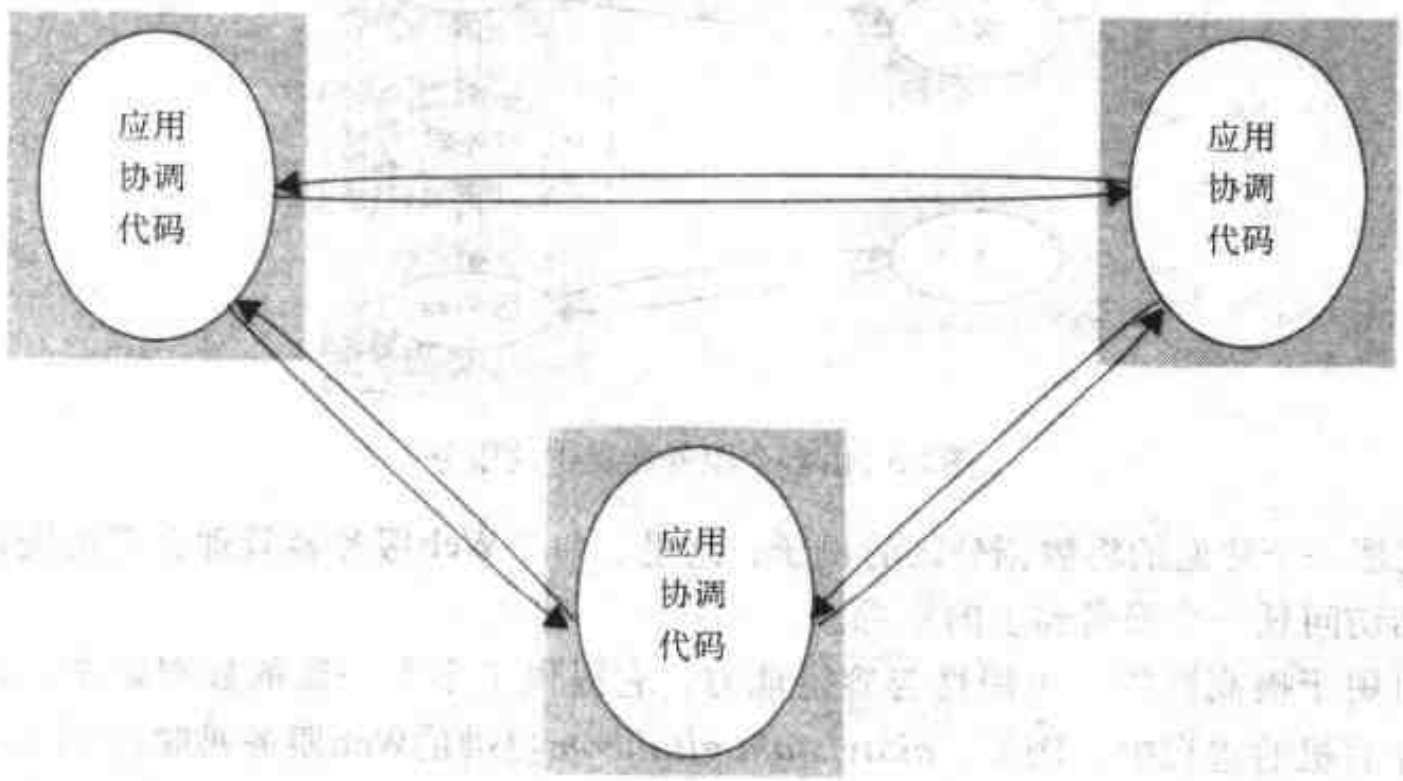


图2-5 基于对等进程的分布式应用

消除服务器进程可以减少为访问本地对象带来的进程间通信延迟。考虑一个分布式的

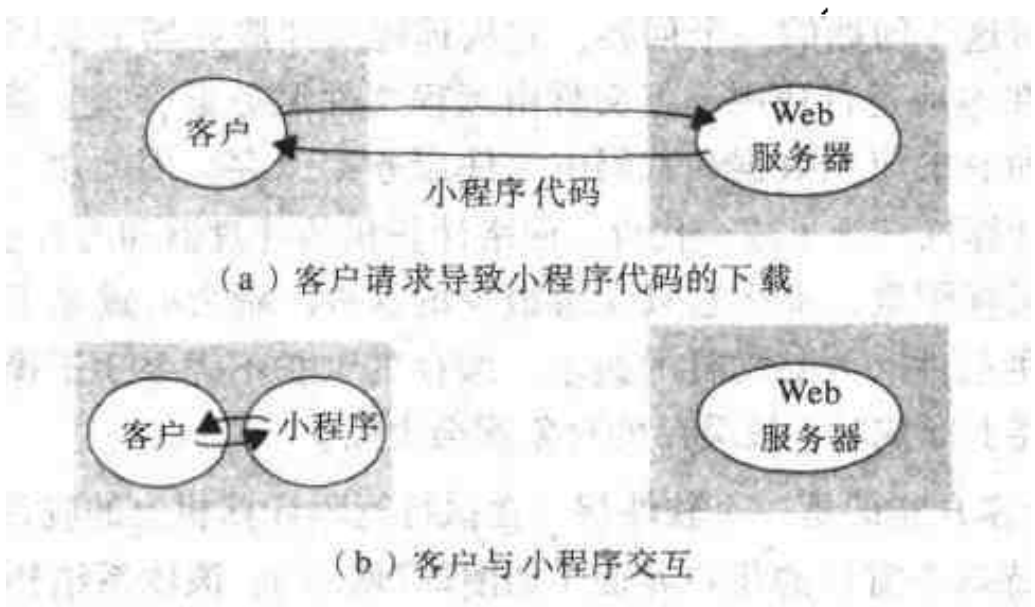
"白板"应用，该应用允许几个计算机上的用户查看和交互修改共享的一幅图画（Floyd *et al.* [1997]就是一个例子）。这能实现成每个站点有一个应用进程，由该进程依赖中间件层完成事件通知和组通信，从而通知所有应用进程该图画所做的修改。这给共享分布对象的用户提供了比通常基于服务器体系结构更好的交互响应。

2.2.3 客户－服务器模型的变种

在考虑了下列因素后，客户－服务器模型能派生出几个变种：

- 使用移动代码和移动代理。
- 用户需要硬件资源有限的、便于管理的低价格计算机。
- 能方便地增加和删除移动设备。

移动代码 第1章介绍了移动代码。Java小程序是一个众所周知的并被广泛使用的移动代码例子，运行浏览器的用户选择了到一个小程序的链接，小程序的代码存储在Web服务器上，可以将它的代码下载到浏览器上并在浏览器端运行，如图2-6所示。在本地运行下载的代码，因为不受与网络通信相关的延迟或带宽变化的影响，所以它具有较好的交互响应。



（a）客户请求导致小程序代码的下载

（b）客户与小程序交互

图2-6 Web小程序

访问服务意味着运行能调用它们操作的代码。一些服务可能进行了标准化，所以可以通过一个已存在的众所周知的应用访问，Web就是一个最熟悉的例子，但即使是这样，有些Web站点仍然使用了在标准浏览器中找不到的功能，而要求下载额外的代码，额外的代码可以与服务器通信。在考虑一个应用时，该应用要求用户应该与发生在服务器信息源端的变化保持一致，这一点不能通过与Web服务器的正常交互获得，因为那种交互总是由客户发起。解决方案是使用额外的按推模式操作的软件——由服务器而不是客户发起交互。

例如，股票经纪人可能提供一个定制的服务通知顾客股票价格的变动。为了使用这个服务，每个顾客要下载一个特殊的小程序接收来自经纪人服务器的修改，该小程序可给用户显示修改，还可能自动地完成买卖操作，这些操作根据顾客设置的、存储在顾客本地计算机上的条件而触发。

第1章提到的移动代码对目的地计算机资源是一个潜在的安全威胁。因此，如果浏览器用7.1.1节讨论的模式，那么只允许小程序对本地资源进行有限的访问。

移动代理 移动代理是一个运行的程序（包括代码和数据两者），它从一台计算机移动到网络上的另一台计算机，代表某人完成诸如信息搜集之类的任务，最后返回结果。一个移动代理可能多次调用所访问场地的本地资源，例如，访问单个数据库条目。如果将这种体系结

构与对资源做远程调用的静态客户相比，那么后者可能会传输大量的数据，前者通过用本地调用替换远程调用在通信开销和时间上有所减少。

移动代理可用于安装和维护一个组织内部的计算机软件，或通过访问每个销售商的站点并执行一系列数据库操作比较多个销售商的产品价格。一个类似想法的早期例子是在Xerox PARC开发的[Shoch and Hupp 1982]所谓的蠕虫程序，该程序利用空闲的计算机完成密集型计算。

移动代理（类似于移动代码）对所访问的计算机上的资源是一个潜在的安全威胁。接收一个移动代理的环境应该根据代理当前代表的用户的身份决定哪些本地资源被允许使用，它们的身份必须以安全的方式包括在移动代理的代码和数据中。另外，移动代理自身是脆弱的，如果它们被拒绝访问所需的信息，就可能完不成任务。由移动代理完成的任务也能通过其他手段完成。例如，需要通过因特网访问Web服务器上资源的网络爬虫程序可通过远程调用服务器处理，而运行得相当成功。因为这些理由，移动代理的适用性是有限的。

36
?
38

网络计算机 在图1-2所示的体系结构中，应用运行在用户本地的桌面计算机上。桌面计算机的操作系统和应用软件通常要求大多数活动代码和数据位于本地磁盘上。但是管理应用文件和维护本地软件库要求提供技术努力，而用户大多没有这个能力。

网络计算机是对这个问题的一个回答。它从远程文件服务器下载用户所需的操作系统和任一应用软件，可在本地运行应用，但文件由远程文件服务器管理。像Web服务器这样的网络应用也能运行。所有的应用数据和代码由文件服务器保存，因此用户就可以从一台网络计算机迁移到另一台计算机。为了减少代价，网络计算机的处理器和内存能力受到限制。

如果网络计算机有硬盘，那么它只保留最少的软件。剩余的磁盘用作缓存存储，存储最近从服务器下载来的软件和数据文件的副本。缓存的维护不需要手工操作：当文件的一个新版本在相关的服务器上形成时，被缓存的对象就会失效。

瘦客户 术语瘦客户指的是一个软件层。在执行远程计算机上的应用程序时，由它在用户本地的计算机上支持基于窗口的用户界面（如图2-7所示）。该体系结构与网络计算机模式具有相同的低管理和硬件开销，但是它在计算服务器——一台具有同时运行大量应用能力的计算机——上运行应用，而不是下载应用代码到用户的计算机。通常计算服务器是一个多处理器或集群计算机（见第6章），运行UNIX或Windows NT等多处理器版本的操作系统。

瘦客户体系结构的主要缺点在高度交互的图形活动如CAD和图像处理中，用户感受到的延迟，包括网络和操作系统的延迟，会因为在瘦客户和应用进程之间传输图像和向量信息的需要而增加。

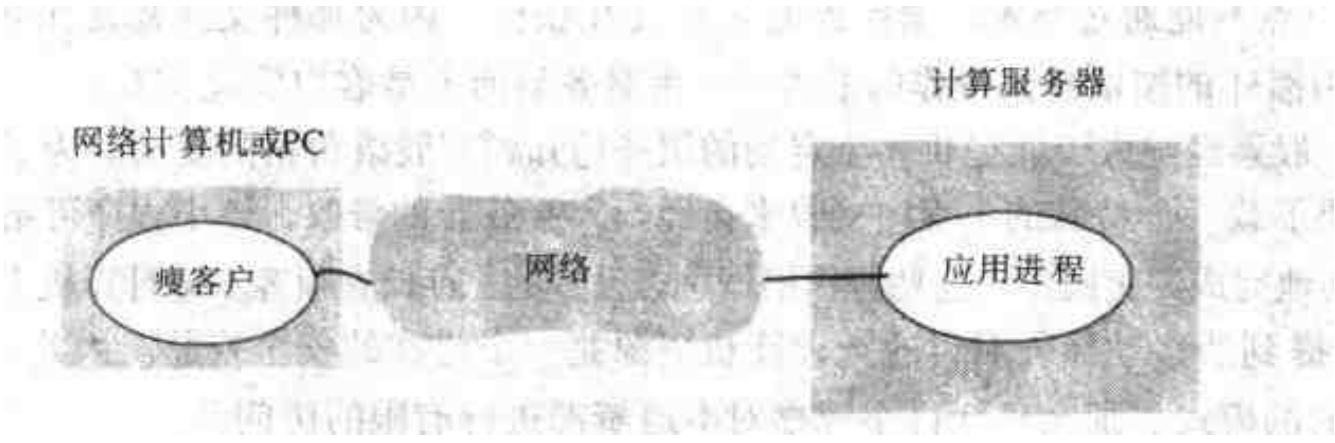


图2-7 瘦客户和计算服务器

瘦客户实现 瘦客户系统在概念上是简单的，在某些环境中它们是直接的实现。例如，大多数UNIX的变种，包含X-11窗口系统，见文本框中关于X-11窗口系统的讨论。

39

为了支持客户和服务器之间的边界能被画出一个图形用户界面，在图形操作流水线中需要给出若干要素。X-11在画线、画形状和窗口处理的图形原语层给出了这个边界。Citrix公司的WinFrame产品[www.citrix.com]是一个使用广泛的实现了瘦客户概念的商业产品，它按类似的方式操作。该产品提供一个瘦客户进程，可运行在众多平台上，支持桌面系统对Windows NT主机上运行的应用提供交互访问。其他实现包括由英国剑桥大学的AT&T实验室开发的远距离传输和虚拟网络计算机（VNC）系统[Richardson *et al.* 1998]。VNC在屏幕像素层次画出了边界，当用户在计算机间移动时为他们维护图形上下文。

X-11窗口系统 X-11窗口系统[Scheifler and Gettys 1986]是一个进程，它管理计算机上的显示和交互输入设备（键盘、鼠标）。X-11提供内容丰富的过程库（X-11协议），用于显示和修改窗口中的图形对象，以及创建和操纵窗口自身。

X-11系统被称为窗口服务器进程。X-11服务器的客户是用户当前交互的应用程序。客户程序通过调用X-11协议的操作与服务器交互，包括在窗口中写文字和画图形对象等。客户不需要与服务器在同一台计算机上，因为服务器过程总是通过RPC机制调用，因此X-11窗口服务器具有瘦客户的关键特性。（X-11颠倒了客户－服务器术语，X-11服务器是由于它提供了给应用程序的图形显示服务而得名，而称它为瘦客户软件是出于它使用运行在计算服务器上的应用程序的角度考虑。）

移动设备和自发网络 像第1章解释的那样，小型便携的计算机设备数量在不断增长，这些设备包括膝上计算机、个人数字助理（PDA）之类的手持设备、移动电话和数字相机、智能手表之类的可穿戴计算机、嵌入到像洗衣机之类的日常家电设备。这些设备多数具有无线联网功能，有的在大都市或更大的范围（GSM、CDPD），有的范围有几百米（WaveLAN）或几米（蓝牙、红外线和HomeRF）。较小范围的网络带宽可达到10Mbps数量级，GSM可以保证有几百Kbps的数量级。

这些设备与分布式系统适当集成，能支持移动计算（见1.2.3节），由此用户可在网络环境之间携带移动设备，利用本地和远程的服务。将移动设备和其他设备集成到一个给定的网络，这种形式的分布用术语*自发网络*描述最适合。以迄今可能的非正规方式将移动和非移动设备连接到网络的应用均可由这个术语描述。像嵌入在家电中的设备给用户提供服务，给附近其他设备提供服务，像PDA之类的便携设备使得用户能访问当前场地提供的服务，以及像Web之类的全局服务。

40

第1章给出了用户访问一个组织的例子。图2-8给出了自发网络的另一个图示，覆盖一个酒店套房的无线网络。套房的高保真系统提供音乐服务；用户能将任意组合的音乐选择送到卧室、浴室的扬声器和无线连接的耳机上。闹钟系统通过音乐服务提供叫醒服务。TV/PC（带“机顶盒”的电视）既可用作电视也可用作计算机，它（通过酒店的因特网网关）提供对Web的访问，包括提供到所有酒店设施的Web接口。它可以为用户提供查看服务，使用户可以浏览存储在相机或便携式摄像机中的图像。在租借的基础上，它还能提供用户喜欢的应用。

图2-8给出了用户带到酒店套间里的设备：膝上计算机、数字相机和PDA。PDA的红外线能力使得它能作为一个“通用控制器”——类似电视遥控器的设备，用于控制酒店房间里的多种设备，包括照明和音乐服务。

然而酒店里的有些服务，如打印和自动售货机通常不是便携的，于是存在用最少的人工

干预即可使用它们的需求。它们与网络连接和访问网络服务应该不需要事先通知或事先进行管理操作才能进行。

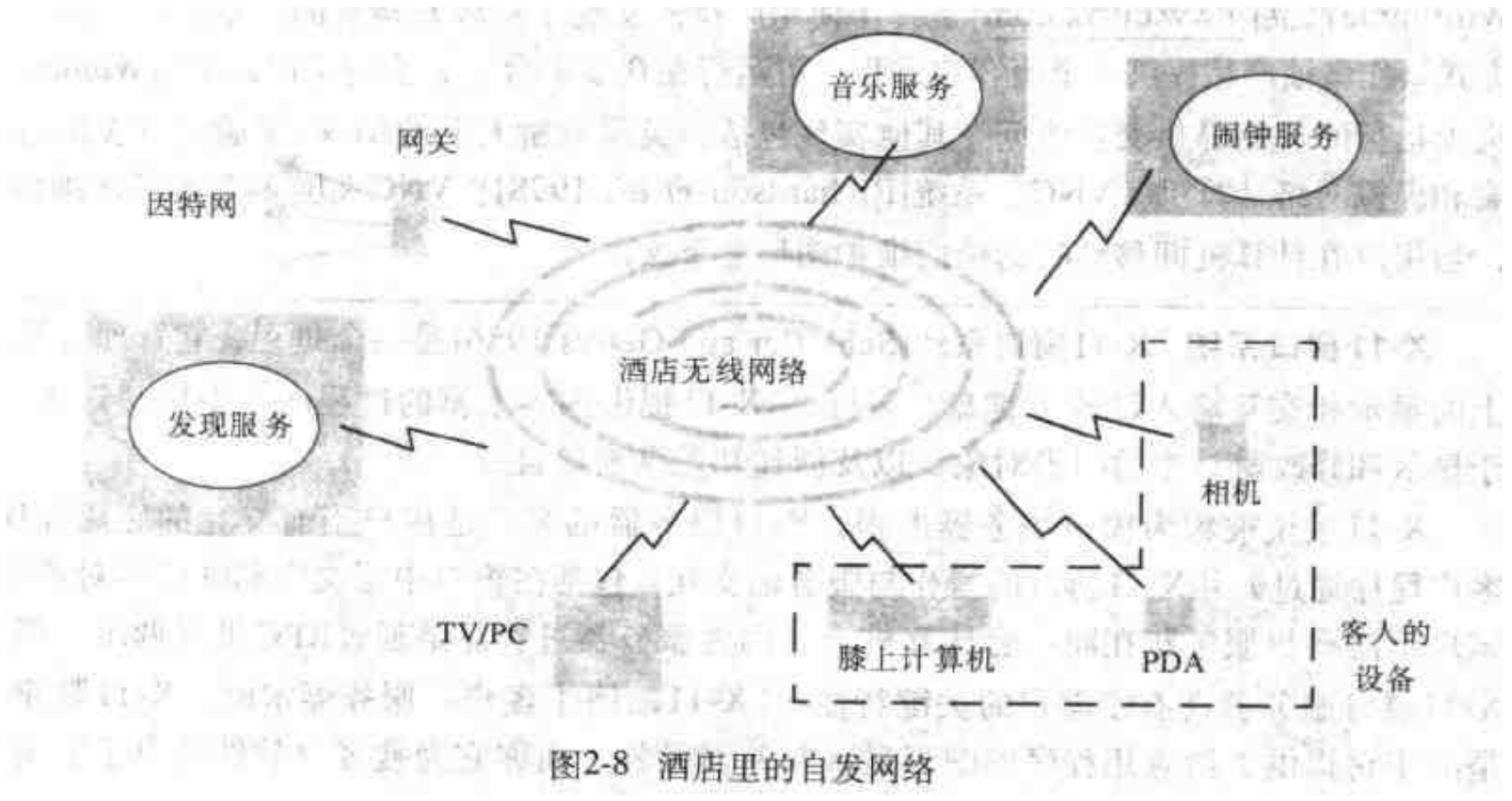


图2-8 酒店里的自发网络

自发网络的关键特征是：

- 容易连接到本地网 无线连接避免了预安装电缆的需要，避免了插头、插座带来的不方便和可靠性问题。它们是移动时（在火车上、飞机上、汽车上、自行车上或船上）惟一能维护的通信链路。带到一个新的网络环境中的设备能透明地重配置以获得那里的连接：用户不必输入本地服务的名字或地址即可获得本地服务。

41

- 容易与本地服务集成 设备插入到（和移动到）已有的设备网络，将自动发现那里提供什么服务，不需要用户进行特殊的配置动作。在酒店的例子中，客人的数码相机将自动发现诸如TV/PC提供的查看服务，这样，客人能发送相机中存储的图像用于显示。

自发网络引发了一些重要的设计问题。首先，支持方便的连接和集成是一个挑战。例如，因特网寻址和路由算法均假定计算机是定位在一个特定的子网。如果一台计算机移动到另一个子网，那么它就不能再用相同的因特网地址访问。第3章将讨论该问题的一种解决方法。

其他问题源于移动用户。这些用户在旅行时只有有限的连接，同时这些连接的自发性会引起安全问题。

- 有限的连接 用户在移动时并不总是保持连接。例如，他们可能乘火车旅游，在通过隧道时，与无线网的连接时断时续。他们也可能在无线连接完全中断的区域或在维持连接代价太大的时候长时间完全断连。这里的问题是系统如何支持用户在断连时继续工作。第14章将讨论这个主题。
- 安全性和私密性 许多安全和个人私密性问题会出现在诸如酒店客人所处的环境中。如果他们以无人监管的方式进行无线连接，酒店和酒店客人应对来自客人或酒店雇员的安全性攻击的能力是脆弱的。一些系统在用户移动时跟踪用户的物理位置（如基于“活动徽章”的系统[Want *et al.* 1992]），但这可能威胁到用户的隐私。最后，一些使得用户在移动时能访问企业内部网的设施可能会暴露应该在企业内部网防火墙后的数据，也可能打开企业内部网而使其受到外部的攻击。

发现服务　自发网络要求运行在便携设备和其他家电上的客户进程访问它们所连接到的网络上的服务。但在实现不同功能的设备中，为了完成任务，客户如何连接到他们所需要的服务？让我们在一个特定例子的环境中说明这一点：酒店客人回到酒店，希望查看他们用数码相机拍摄的相片，并在酒店打印其中的一些，然后将选中的照片以电子邮件方式发送回家。客人的房间有TV/PC，酒店可能有彩色数码打印服务，如果没有，可能提供传真机服务。

我们不能期望每个打印机制造商正好实现同样的协议，以便用于客户访问它的打印服务。即便是这样，客户也经常要求服务是可适应的，像我们的例子所表明的那样：数码相机客户可在本地TV上显示照片或将照片发送到酒店的传真机上。 42

我们需要的是客户发现所连接的网络上有什么服务可用以及查明它们属性的方法。发现服务的目的是接受并存储网上可用的服务细节，对客户关于这些的查询给出回应。更准确地说，发现服务提供两个接口：

- 注册服务接受来自服务器的注册请求，在发现服务的数据库中记录它们所包含的细节。
- 查找服务接受关于可用服务的查询，在数据库中查找与查询匹配的注册服务。返回的结果包括足够的细节使得客户能基于它们的属性在几个类似的服务之间选择，并与它们中的一个或几个建立连接。

回到我们的例子，酒店的发现服务包括在酒店中可用的打印和查看服务细节，像显示设备或打印机所在的房间号、设备的质量（分辨率）、设备接受的图像模型、色彩能力等等。客人的数字相机查询发现服务有关打印和查看服务事宜，处理结果清单中服务所在房间的邻近程度以及与它的图像模型的兼容性，最后将合适的设备清单提供给用户。用户选择相应房间中的TV/PC或打印机，并将一些图像分发到设备上，其他图像可能作为电子邮件附件发送给朋友和家人。

第9章包含对发现服务的近距离观察。

2.2.4 接口和对象

在一个进程（不论它是一个服务器进程或一个对等的进程）中可调用的函数集合由一个或多个接口定义指定。完全地接口定义描述见第5章，该概念与那些熟悉的语言如Modula、C++或Java类似。在客户－服务器体系结构的基本形式中，每个服务器进程可看成是一个具有固定接口的实体，其中的接口定义了能被调用的功能。

用面向对象语言如C++和Java再加适当额外的支持，分布式处理能以更面向对象的方式构造。许多对象能封装在服务器或对等的进程中，对它们的引用能传递到其他进程，这样它们的方法能由远程调用访问，CORBA和带远程方法调用（RMI）机制的Java采用了这种方法。在这个模型中，每个进程具有的对象的数量和类型（在支持移动代码的语言如Java中）可能随系统活动要求而变化，在一些实现中对象的位置也可能改变。

不论我们采用静态的客户－服务器体系结构还是前一段概述的更动态的面向对象模型，在进程之间和在计算机之间责任的分布仍然是设计的一个重要方面。在传统的体系结构模型中，责任是静态分配的（例如，一个文件服务器只负责文件，不负责Web页面或它们的代理或其他对象类型）。但在面向对象模型中新的服务以及在某些情况下新的对象类型，能被实例化并马上可供调用。 43

2.2.5 分布式体系结构的设计需求

在分布式系统中，促发对象和进程分布的因素有许多，它们的重要性是不可忽视的。20世纪60年代通过使用共享文件，共享第一次在分时系统中实现，大家很快认识到共享的好处，并在多用户操作系统如UNIX和多用户数据库系统如Oracle中得到了开发，使得进程能共享系统资源和设备（文件存储能力、打印机、音频和视频流），使得应用进程能共享应用对象。

以多处理器芯片形式出现的低廉的计算能力消除了对中央处理器共享的需要，中等性能的计算机网络的可用性和对相对昂贵的硬件资源如打印机和磁盘存储共享的要求导致了20世纪70年代和80年代分布式系统的开发。今天，资源共享被认为是想当然的，但大规模的有效的数据共享仍然是巨大的挑战，随着数据变化（大多数数据随时间变化），可能出现并发和有冲突的修改。如何控制共享数据的并发修改是第12章和第13章的主题。

性能问题 资源分布引起的挑战远远超出了并发修改的管理。由计算机有限的处理能力和通信能力引起的性能问题在下列子标题下予以考虑：

反应能力 交互应用的用户要求交互反应快速并且一致，但客户程序需要经常访问共享资源。当调用一个远程服务时，反应速度不仅由服务器、网络的负载和性能决定，还由所有涉及到的软件组件的延迟决定——客户和服务器操作系统的通信和中间件服务（例如，远程调用支持）以及实现服务的进程的代码。

另外，即使进程在同一台计算机上，进程之间的数据传递和相关的控制转换还是相对较慢的。为了获得较好的交互反应时间，系统必须由相当少的软件层组成，在客户和服务器之间传递的数据量必须小。

这些问题可通过客户浏览Web的性能加以说明，在那种情况下，访问本地缓存的页面和图像的反应速度最快，因为它们由客户应用保存。访问远程正文页面相对较快，因为它们的数据量小，而图像因为涉及的数据量大会有较长的延迟。

吞吐量 计算机系统性能的一个传统的度量是吞吐量——计算工作进行的比例。我们对分布式系统为所有它的用户完成工作的能力感兴趣。它受客户和服务器的处理速度、数据传输
44 率的影响。在远程服务器上的数据必须从服务器进程传递到客户进程，经过两个计算机上的若干个软件层。软件层的吞吐量与网络的吞吐量一样重要。

平衡计算负载 分布式系统的一个目的是使得应用和服务进程能不竞争同一资源，并发地运行，能开发可用的计算资源（处理器、内存和网络能力）。例如，在客户计算机上运行小程序的能力就减少了Web服务器的负载，使得服务器能提供更好的服务。更有意义的例子是用几个计算机组建一个服务，一些负载很重的Web服务器（搜索引擎、大的商业站点）需要这样做。它利用DNS域名查询服务的能力，为一个域名返回几个机器地址中的一个（见9.2.3节）。

在某些情况下，机器上的负载变化时，负载平衡可能要移动部分完成的工作。这要求系统能支持运行进程在计算机之间移动。

服务质量 一旦用户被提供了所要求的服务功能，如分布式系统中的文件服务，那么用户就能继续询问所提供的服务质量。系统影响客户和用户感知服务质量的主要非功能性特征是可靠性、安全性和性能。最近满足变化的系统配置和资源可用性的适应性也被认为是服务质量的一个重要的方面。Birman一直在列举理由证明这些质量方面的重要性，他的书[Birman 1996]提供了质量方面对系统设计影响的若干有意义的观点。

在大多数计算机系统设计中，可靠性和安全性是关键问题。它们与我们的两个基础模型有极大的关系：故障模型和安全模型，见2.3.2节和2.3.3节的介绍。

直到最近，服务质量的性能才按响应和计算吞吐量给出定义。但最近又按下面段落中讨论的满足适时保证的能力重新定义了服务质量。不论用哪一种定义，分布式系统的性能方面与2.3.1节定义的交互模型有很大的关系。所有这三个基础模型是贯穿全书的重要的参考点。

一些应用用于处理时间至关重要的数据——指要按确定的速度处理的数据流或按确定的速度从一个进程传输到另一个进程的数据流。例如，电影服务由一个客户程序组成，该客户程序从一个视频服务器获取电影，然后将它投射到用户的屏幕上。为了获得满意的结果，连续的视频画面要在指定的时间限制内显示给用户。

事实上，缩写QoS用于特指系统满足这些限制的能力。QoS的获得依赖于在所需时段内必要的计算和网络资源的可用性，这意味着要求系统要提供足够的计算和通信资源使得应用能按时完成每个任务（例如，显示视频画面的任务）。

当前通用的网络，例如可浏览Web的网络，可能有非常好的性能特征，但当它们的负载很重时，其性能会极大地恶化，从而决不能说它们提供了服务质量。QoS同样适用于操作系统。必须为要求QoS的应用保留每个关键资源，而且必须有提供保证的资源管理器。不能被满足的资源保留请求将被拒绝。这些问题将在第15章解决。 45

缓存和复制的使用 上面讨论的性能问题经常是分布式系统成功实施的主要障碍，但在系统设计中，可以通过数据复制和缓存的使用来克服性能问题，这方面已有很大进展。2.2.2节介绍了缓存和Web代理服务器，但没有讨论在服务器上的资源被修改后，该资源被缓存的副本如何保持最新的问题。为适应不同的应用，可使用多种不同的缓存一致性协议。例如，第8章给出了由两个不同的文件服务器设计所使用的协议。现在，我们要概述作为HTTP协议的一部分的Web缓存协议，而HTTP协议见4.4节的描述。

Web缓存协议 Web浏览器和代理服务器缓存都能响应对Web服务器的客户请求。因此，对一个客户请求的响应可能来自浏览器缓存，也可能来自客户和Web服务器之间的代理服务器缓存。通过配置缓存一致性协议可以将Web服务器持有的最新的资源副本提供给浏览器。但是，虑及性能、可用性和断连操作，可放宽数据新鲜度条件。

浏览器或代理通过检查原来的Web服务器能了解缓存的响应是否是最新的，从而验证缓存的响应。如果缓存的响应不能通过测试，那么Web服务器将返回一个最新的响应，用它替换缓存中过时的响应。当客户请求相应的资源时，浏览器和代理验证缓存的响应。如果缓存的响应还足够新，它们就不进行验证。即使Web服务器知道资源被修改的时间，它也不通知带缓存的浏览器和代理。但如果要这样做，Web服务器要记录对每个资源感兴趣的浏览器和代理。为了能让浏览器和代理确定它们存储的响应是否过时了，Web服务器给它们的资源赋予一个大致过期的时间，例如，从资源上一次被修改的时间估算。一个过期时间可能会有误导作用，因为Web资源可能在任何时刻被修改。每次Web服务器应答一个请求，都会把资源的过期时间和服务器上的当前时间附到响应上。

浏览器和代理把过期时间和服务器时间与缓存的响应存储在一起。这使得浏览器或代理在接收请求后能计算出缓存的响应是否可能过期。缓存的响应是否过期是通过与过期时间比较它们的年龄验证的。响应的年龄是响应被缓存的时间和服务器时间之和。注意这种计算并不依赖Web服务器上的计算机时钟以及浏览器或代理时钟的相互一致性。

可依赖性问题 可依赖性是大多数应用领域中的需求。它不仅在命令和控制活动中是关键的（这里生命是利害攸关的），而且在许多商业应用中也是关键的，包括快速开发的因特网商业领域，这里参与者的金融安全和完备性取决于他们操作的系统的可依赖性。在2.1节中，我们把计算机系统的可依赖性定义为正确性、安全性和容错。这里我们讨论安全和容错的需求对体系结构的影响，而把实现这些需求的相关技术的描述放在本书的后面。开发用于检查或
46 确保分布并发程序正确性的技术是许多当前正在进行的研究中的课题。虽然已获得一些有前途的结论，但还很少有能在实际应用中实施的成熟结果。

容错 在出现硬件、软件和网络故障的时候，可靠的应用应该能继续正确工作。可靠性通过冗余获得——提供多个资源使得在出现故障时系统和应用软件能重新配置并继续执行任务。因为冗余的代价高，所以其应用的范围有限，获得的容错的程度也受限制。

在体系结构层，冗余要求使用多个计算机和多个通信路径，其中可以在多个计算机上运行系统的每一个组件进程，并在多个通信路径上传递消息。数据和进程可在需要提供容错的地方进行复制。冗余的一个常见形式是在不同计算机上提供数据项的几个副本，只要这些计算机中的一个还在运行，数据就可以被访问到。一些关键的应用，如空中交通控制系统，对数据的容错有很高的要求，因此为保持多个副本的更新就必须付出很高的代价。第14章将进一步讨论这个问题。

其他形式的冗余用于使通信协议可靠。例如，消息被重传直到接收到一个确认消息。RMI底层的协议的可靠性见第4章和第5章。

安全性 安全需求对体系结构的影响涉及仅在能有效抵御攻击的计算机上定位敏感数据和其他资源的需求。例如，医院数据库包含病人的记录，其中有的部分是敏感的，应该仅仅给某些医生看，而其他部分是可以大家共享的。构造这样一个系统并不合适：在访问这个系统时，系统将整个病人的记录装载到用户的桌面计算机，因为典型的桌面计算机并不能构成一个安全的环境——用户能运行程序访问或修改存储在他们个人计算机中的任何一部分数据。2.3.3节介绍解决更广泛的安全需求的安全模型，第7章描述可用于实现上述目标的技术。

2.3 基础模型

上面所有的，甚至差别较大的系统模型都有一些相同基本特性。特别是，所有的模型由若干进程组成，这些进程通过在计算机网络上发送消息而相互通信。所有的模型共享上一节给出的设计需求，主要涉及到进程和网络的性能和可靠性特征，以及系统中资源的安全性。本节给
47 出基于基本特性的模型，它允许我们能更详细描述它们要展示的特性、故障和安全风险。

通常，一个模型仅包含我们要考虑的实质性成分，以便理解和推理系统行为的某些方面。一个系统模型必须解决下列问题：

- 什么是系统中的主要实体？
- 它们如何交互？
- 影响它们个体行为和集体行为的特征是什么？

模型的目的是：

- 显示有关我们正在建模的系统的假设。
- 给定这些假设，就什么是可能的，什么是不可能的给出结论。以有保证的通用算法或期望的性质的形式给出结论。这些保证依赖于逻辑分析和适当的数学证明。

知道我们的设计做什么和不做什么，就能从中获得很多。它允许我们决定在一个特定系统中实现的设计能否运作，我们只需询问在那个系统中我们的假设是否成立。通过清晰显示给出我们的假设，我们希望能利用数学技巧证明系统的性质，这些性质对任何满足我们假设的系统都成立。最后通过从硬件之类的细节中抽象系统的基本实体，我们能阐明对系统的理解。

我们希望在我们的基本模型中捕获的分布式系统情况可以帮助我们讨论和推理：

- 交互　计算发生在进程中，进程通过传递消息交互，导致在进程之间的通信（例如，信息流）和协调（活动的同步和排序）。在分布式系统的分析和设计中，我们特别关注这些交互。交互模型必须反映通信发生带来的延迟，这些延迟经常有较长的持续时间，交互模型必须反映出独立进程相互配合的准确性受限于这些延迟的现象，受限于在分布式系统中跨所有计算机维护同一时间概念的困难。
- 故障　每次在分布式系统运行的任一计算机（包括软件故障）或连接它们的网络出现故障时，系统正确的操作就会受到威胁。我们的模型将对这些故障定义和分类。这提供了分析它们潜在后果的基础，以及进行能容忍任何类型故障的系统设计的基础。
- 安全　分布式系统的模块特性和它们的开放性将它们暴露在外部和内部代理的攻击下。我们的安全模型对发生这种攻击的形式给出了定义并进行了分类，提供了对系统的威胁进行分析的基础，以及设计能抵抗这些威胁的系统的基础。

为了帮助讨论和推理，对本章介绍的模型进行了简化，省略了许多实际系统中的细节。它们与实际系统的关系，以及在模型帮助下所揭示的问题环境中的解决方案是本书的主题。 48

2.3.1 交互模型

2.2节对系统体系结构的讨论表明分布式系统由多个进行复杂交互的进程组成。例如：

- 多个服务器进程能相互协作提供服务，上面提到的例子有域名服务（它将数据分区并复制到因特网中的服务器上）和Sun的网络信息服务（它在局域网的几个服务器上保存口令文件的复制版本）。
- 对等进程能相互协作获得一个公共的目标。例如，一个语音会议系统，它以类似的方式分布音频数据流，但它有严格的实时限制。

大多数程序员非常熟悉算法的概念——完成所需计算的一系列步骤。简单的程序由算法控制，算法中每一步都是有严格顺序的。由算法决定程序的行为和程序变量的状态，这样的程序作为一个进程执行。由多个如上所述的进程组成的分布式系统是很复杂的，它们的行为和状态可以用分布式算法描述，分布式算法是组成系统的每个进程所采取的步骤的定义，包括进程之间消息的传递。在进程之间传递的消息用于在它们之间转移信息和协调活动。

每个进程进展的速率和进程之间消息传递的时限通常是不能预测的，要描述分布式算法的所有状态也是困难的，因为它必须处理所涉及的一个或多个进程的故障或消息传递的故障。

交互的进程完成分布式系统中所有的活动。每个进程有它自己的状态，该状态由进程能访问和修改的数据集组成，包括程序中的变量。属于每个进程的状态完全是私有的，就是说，它不能被任何其他进程访问或修改。

本节讨论分布式系统中影响进程交互的两个重要因素：

- 通信性能经常受限
- 不可能维护单个全局时间概念

信道的性能 我们模型中的信道可用分布式系统中的许多方法实现。例如，通过计算机网络上的流或简单消息传递来实现。计算机网络上的通信有下列有关等待时间、带宽和抖动的性能特征：

49

- 从一个进程发送消息到另一个进程接收消息之间的延迟称为等待时间。等待时间包括：
  - 第一个比特从网络传递到目的地所花的时间。例如，通过卫星连接传递消息的等待时间是无线电信号到达卫星并返回的时间。
  - 访问网络的延迟。当网络负载很重时，该延迟增长很快。例如，对以太网传送而言，发送站点要等待网络空闲。
  - 操作系统通信服务花在发送进程和接收进程上的时间。这一时间会根据操作系统当前的负载而变化。
- 计算机网络的带宽是指在给定时间内网络上能传递的信息总量。当大量信道使用同一网络，它们就不得不共享可用的带宽。
- 抖动是传递一系列消息所用时间的变化值。抖动与多媒体数据有关，例如，如果音频数据的连续采样在不同的时间间隔完成，那么声音将严重失真。

计算机时钟和时序事件 分布式系统中的每个计算机有它自己的内部时钟，供本地进程使用以获得当前时间值，因此在不同计算机上运行的两个进程能将时间戳关联到它们的事件上。但是，即使两个进程在同一时间读它们的时钟，它们的本地时钟也会提供不同的时间值，这是因为计算机时钟与绝对时间有偏差，而且它们的漂移率互不相同。术语时钟漂移率指的是计算机时钟不同于绝对参考时钟的相对量。即使分布式系统中所有计算机的时钟在初始的时候设置相同，但最终它们会相差很大，除非使用更正。

有几种更正计算机时钟的方法。例如，计算机可使用无线电接收器从GPS按约1μs的精度接收时间读数，但GPS接收器不能在建筑物内工作，而且花在每一台计算机上的开销也不能被接受。相反，具有像GPS之类精确时间源的计算机可发送时序消息给网络中的其他计算机。本地时钟时间之间的协议结果当然会受消息延迟影响。对时钟漂移和时钟同步的更详细的讨论见第11章。

交互模型的两个变种 在分布式系统中很难在进程执行、消息传递或时钟漂移的时间上设置时间限制。两种极端对立的观点提供了一对简单模型：第一个对时间有严格的假设，第二个对时间没有假设。

- 同步分布式系统 Hadzilacos和Toueg[1994]定义了一个同步分布式系统，给出了下列约束：
  - 已知进程执行每一步的时间有一个上限和下限。
  - 通过通道传递的每个消息在一个已知的时间范围内接收到。

50

  - 每个进程有一个本地时钟，它与实际时间的漂移率在一个已知的范围内。

建议分布式系统合适的关于进程执行时间、消息延迟和时钟漂移率的上下界是可能的，但是达到实际值和提供对所选值的保证是比较困难的，除非能保证边界值，否则任何基于所选值的设计都不会可靠。然而按同步系统构造算法有利于对实际分布式系统的行为提出一些想法，例如，在同步系统中，可能使用超时检测进程的故障，见本节故障

模型部分。

同步分布式系统确定可以构建，所要求的是进程用已知的资源条件完成任务，这些资源条件保证有足够的处理器周期和网络能力以及要为进程提供漂移率在一定范围内的时钟。

- *异步分布式系统* 许多分布式系统，例如因特网，不作为同步系统时非常有用。因此，我们需要另一个模型——异步分布式系统是对下列元素没有限制的系统：
  - 进程执行速度。例如，一个进程步可能只花亿万分之一秒，而另一个进程步要花一个世纪。就是说，每一步能花任意长的时间。
  - 消息传递延迟。例如，从进程A到进程B传递一个消息可能很快，也可能花几年时间。换句话说，消息可在任意长时间后才接收到。
  - 时钟漂移率。时钟漂移率可以是任意的。

异步模型对执行的时间间隔没有任何限制。这正好与因特网一致，在因特网中服务器或网络负载没有固有的限制，对像用FTP传输文件要花多长时间也没有限制。有时电子邮件消息要花几天时间才能到达。下一页的文本框部分说明在异步分布式系统中达成协定的困难。

即使有这些假设，有些设计问题仍然能解决。例如，虽然Web并不总在一个合理的时间限制内提供特定的响应，但浏览器已被设计成在用户等待时允许用户做其他事情。对异步分布式系统有效的任何解决方案对同步系统也有效。

实际的分布式系统经常是异步的，因为需要共享处理器的进程和共享网络的信道。如果有太多未知特性的进程共享一个处理器，那么它们任何一个的性能都不能保证。有许多不能在异步系统中解决的设计问题，在使用时间的某些特征后就能解决。在最终期限之前传递多媒体数据流的每个元素就是这样一个问题。对这样的问题，用户可使用同步模型。下页文本框部分说明在异步系统中同步时钟的不可能性。 51

事件排序 在许多情况下，我们对一个进程中的一个事件（发送或接收一个消息）是发生在另一个进程中的另一个事件之前、之后还是同时发生感兴趣。尽管缺乏精确的时钟，系统的执行仍能用事件和它们的顺序来描述。

例如，考虑下列在邮件列表中的一组电子邮件用户X、Y、Z、A之间的邮件交换：

1. 用户X发送主题为会议的消息。
2. 用户Y和Z回发一个主题为*Re*：会议的消息。

在实际环境中，X的消息最早发送，Y读取它并回复；Z读取X的消息和Y的回复并发出另一个回答，该回答参考了X和Y的消息。但是由于在消息发送中各自独立的延迟，消息的传递可能像图2-9一样，而一些用户可能以错误的顺序查看这两个消息，例如，用户A可能看见：

收件箱：		
序号	发件人	主题
23	Z	*Re*：会议
24	X	会议
25	Y	*Re*：会议

Pepperland协定 Pepperland军队的两个师“红师”和“蓝师”驻扎在邻近两个山上。沿着山谷下面是入侵的敌军。只要Pepperland的两个师留在驻地，它们就是安全的。它们通过派出通信兵穿越山谷进行通信。Pepperland的两个师需要协商它们中的哪一方带领发起对敌军的冲锋以及冲锋在何时发生。即使是在异步的Pepperland中，由谁带领冲锋还是可能达成一致的。例如，每个师给出它剩余人员的数量，人数多的一方带头冲锋（如果一样多，红师胜出）。但何时冲锋呢？在异步Pepperland中，通信兵的速度是变化的，如果红师派出一个通信兵，带着“冲锋”消息，蓝师可能3个小时也收不到，也可能5分钟就收到了。在同步Pepperland中，仍有协调问题，但是两师知道一些有用的限制：每个消息花至少*min*分钟至多*max*分钟到达。如果带头冲锋的师发送“冲锋”消息，那么它等待*min*分钟就可以冲锋。另一个师在收到消息后等待1分钟，然后冲锋。在带头冲锋的师之后不超过（*max* - *min* +1）分钟保证发起另一个师的冲锋。

52

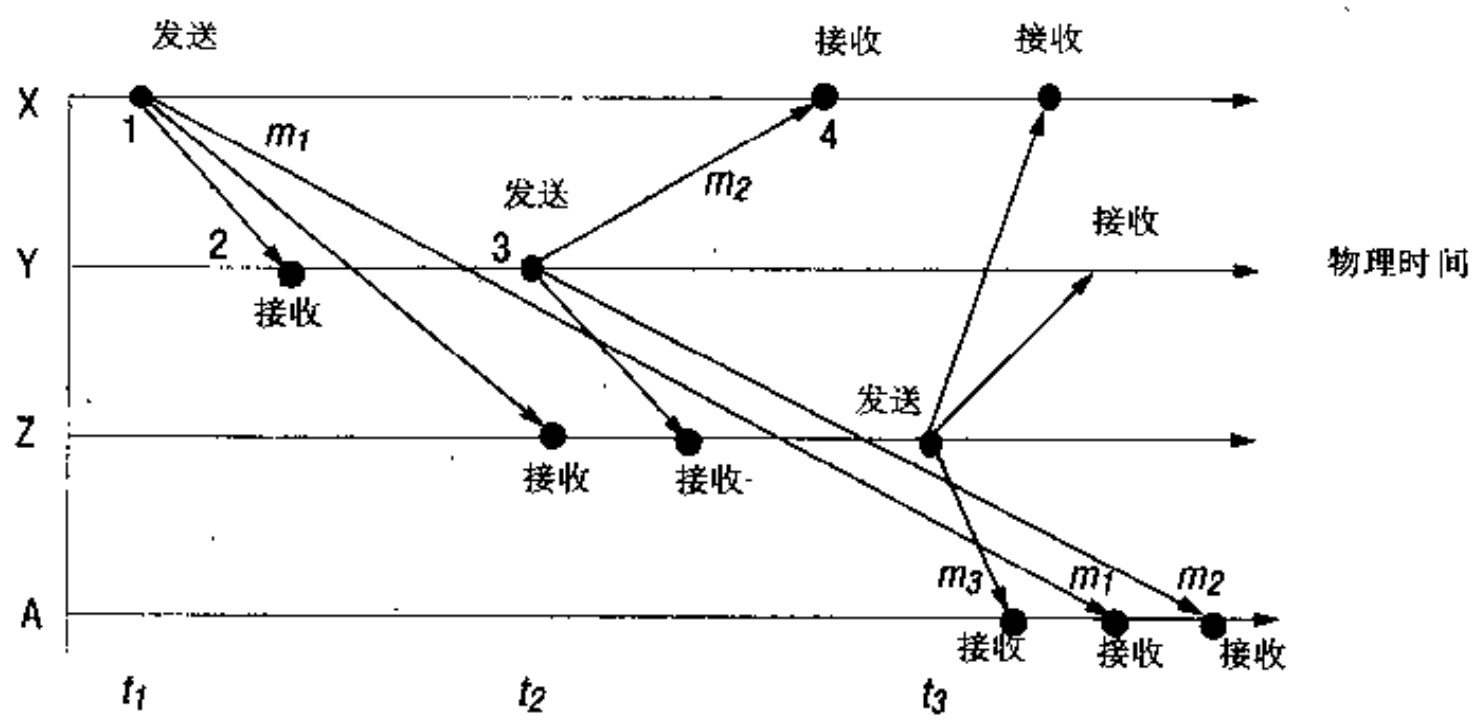


图2-9 事件的实时排序

如果X、Y、Z的计算机上的时钟能同步，那么每个消息在发送时能带上本地计算机的时钟。例如，消息m_1、m_2和m_3能携带时间t_1、t_2、t_3，其中$t_1<t_2<t_3$。接收到的消息将根据它们的时间排序显示给用户。如果时钟基本上同步，那么这些时间戳可以按正确的顺序排列。

因为在一个分布式系统中时钟不能精确同步，Lamport[1978]提出了*逻辑时间*的模型，用于在分布式系统中给运行在不同计算机上的进程的事件提供顺序。逻辑时间允许消息的顺序不需要求助于时钟就可以推断出来。具体细节见第10章，我们在这里说明逻辑排序的某些方面能应用到邮件排序问题。

逻辑上，我们知道消息在它发送之后才被接收，因此，我们能给图2-9所示的成对的事件一个逻辑排序。例如，考虑仅涉及X和Y的事件：

X在Y接收到m_1之前发送m_1；Y在X接收到m_2之前发送m_2。

我们也知道应答在接收到消息后发出，因此对Y，我们有下列逻辑排序：

Y在发送m_2之前接收m_1。

逻辑时间通过给每个事件赋予一个相当于它逻辑排序的数字，进一步拓展了这个想法，

这样，后发生事件的数字比先发生事件的数字大。例如，图2-9对X和Y上的事件给出了数字1到4。

2.3.2 故障模型

在分布式系统中，进程和信道都可能发生故障，即，它们可能不是正确或想要的行为。故障模型定义了故障可能发生的方式，以便理解故障的后果。Hadzilacos和Toueg[1994]提供了用于区别进程故障和信道故障的分类标准。这些故障分类标准将在下面的遗漏故障、随机故障和时序故障部分讲述。 53

本书将始终使用故障模型。例如：

- 在第4章中给出了数据报和流通信的Java接口，分别提供不同程度的可靠性。
- 在第4章中给出了支持RMI的请求－应答协议。它的故障特征依赖于进程和信道两者的故障特征。协议能用数据报或流通信创建。选择可根据对实现简单性、性能和可靠性的考虑作出。
- 在第13章中给出了事务的两阶段提交协议。用于解决进程和信道的良定义故障。

遗漏故障 被划分成遗漏故障的错误指的是进程或信道不能完成所要做的动作。

进程遗漏故障 进程主要的遗漏故障是崩溃。我们说进程崩溃了，是指进程停止了，将不再执行任何程序动作。能在故障面前存活的服务，如果假设它所依赖的服务能单纯地崩溃，即进程或者运行正确或者停止运行，那么就能简化它的设计。一个进程p是通过不能获得对q调用消息的应答这种方法检测到另一个进程q崩溃。然而，这种崩溃检测的方法依赖超时的使用，即进程用一固定时间等待某些事件的发生。在异步系统中，超时只能表明进程没有响应——它可能是崩溃了，也可能是慢，或者消息还没有到达。

如果其他进程能确切检测到进程已经崩溃，那么这个进程崩溃称为故障－停止。在同步系统中，如果确保消息已被发送，而其他进程又没有响应，那么进程可使用超时来检测同步系统中的故障-停止行为产生。例如，如果进程p和q的流程是q要回答来自p的消息，而且进程p在按p本地时钟度量的一个最大时间范围内没有收到进程q的应答，那么进程p可以作出结论：进程q出故障了。下面的阴影部分说明在异步系统中检测故障的困难以及在故障面前达成协定的困难。

通信遗漏故障 考虑通信原语*send*和*receive*。进程p通过将消息m插入到它的外发消息缓冲区来执行*send*。信道将m传递到q的接收消息缓冲区。进程q通过将m从它的接收消息缓冲区取走并完成交付来执行*receive*（如图2-10所示）。通常由操作系统提供外发消息缓冲区和接收消息缓冲区。

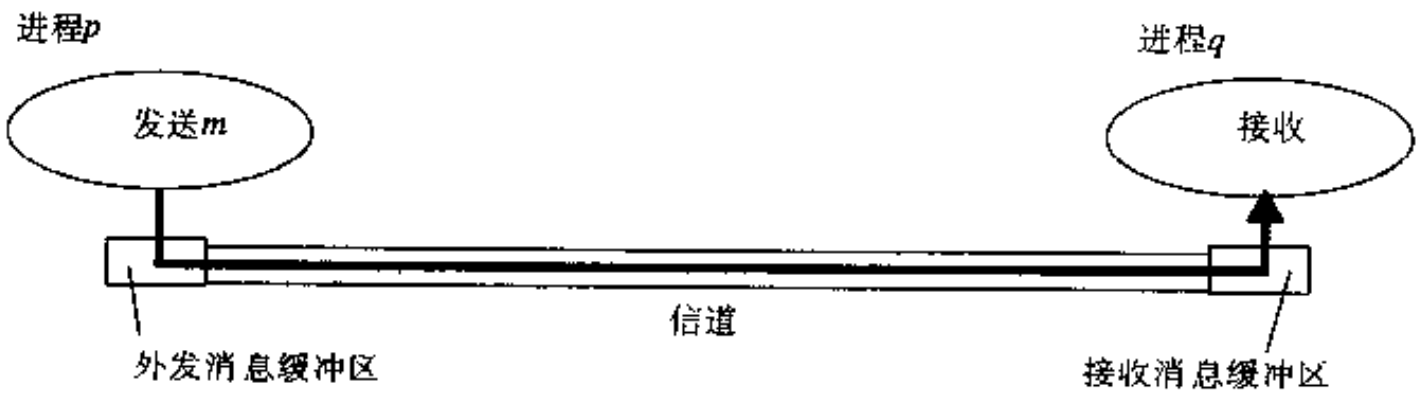


图2-10 进程和通道

故障检测 在Pepperland师驻扎在山顶的情况下，假设敌军聚集足够的力量攻击任何一个扎营的师，任一方都可能失败。进一步假设，在没有被打败的时候，各师定时地派出通信兵报告对方自己的状态。在异步系统中，没有一个师能判断是对方被打败了还是通信兵穿越中间山谷的时间太长。在同步的Pepperland，一个师通过应该定期出现的通信兵的缺席能区别另一个师是否被打败了，但是，另一个师可能在派出最后一个通信兵后就被打败了。

在故障面前达成协定的不可能性 我们一直假设Pepperland通信兵最终总能通过山谷，现在要假设敌军会抓住通信兵，阻止他到达（我们还假设敌人不可能给通信兵“洗脑”，从而让他通知错误的消息）。红师和蓝师能发送消息使得他们能一致决定对敌军冲锋或投降？但是，像Pepperland的理论家Ringo证明的一样，在这样的环境中，这些部队不能一致地决定做什么。为了了解这一点，假设两支部队执行了达成一致的Pepperland协定。每一方提出“冲锋！”或“投降！”，协定使得双方同意一方或另一方的动作。现在考虑在任一轮协定中发送的最后一个消息。携带消息的通信兵可能被敌军捕获。不论消息到达与否，最终结果必须一致，所以我们去掉这个消息。现在，我们对剩下消息中的最后一个应用同一论点。这个论点可再应用到那个消息，然后继续应用这个论点，最后我们将以没有要发送的消息结束！这表明如果通信兵被抓住，就不能保证存在Pepperland师之间一致的协定。

如果信道不能从p的外发消息缓冲区将消息传递到q的接收消息缓冲区，那么它就产生了遗漏故障，这就是所谓的“丢失的消息”，引起的原因通常是由于在接收端或中间的网关上缺乏缓冲区空间，或是由于网络传输错误（可由消息数据携带的校验和检测到）。Hadzilacos和Toueg[1994]将在发送进程和外发消息缓冲区之间的消息丢失称为*发送遗漏故障*；在接收消息缓冲区和接收进程之间的消息丢失称为*接收遗漏故障*；在两者之间的消息丢失称为*通道遗漏故障*。遗漏故障和随机故障的分类如图2-11所示。

54 ~ 55

故障类型	影　响	描　述
故障－停止	进程	进程停止并一直停止，其他进程可检测这个状态
崩溃	进程	进程停止并一直停止，其他进程可能不能检测到这个状态
遗漏	通道	插入外发消息缓冲区的消息不能到达另一端的接收消息缓冲区
发送遗漏	进程	进程完成了*send*，但消息没有放到它的外发消息缓冲区
接收遗漏	进程	一个消息已放在进程的接收消息缓冲区，但那个进程没有接收它
随机（拜占庭式）	进程或通道	进程/通道显示出随机行为：它可能在随机时间里发送/传递随机的消息，会有遗漏发生；一个进程可能停止或者采取不正确的步骤

图2-11　遗漏故障和随机故障

按照故障的严重性对故障进行分类。到现在为止我们描述的所有故障都是良性故障。在分布式系统中大多数故障是良性的。良性故障包括遗漏故障、时序故障和性能故障。

随机故障 术语随机或拜占庭故障用于描述可能出现的最坏的故障语义。这里，可能发生任一类型的错误。例如，一个进程可能在数据项中设置了错误的值，也可能为响应一个调用返回一个错误的值。

进程的随机故障是指进程随机地省略要做的处理步骤或执行一些不想要的处理步骤。因此进程的随机故障不能通过查看进程是否对调用有响应而被检测，因为它可能随机地遗漏应答。

信道也有随机故障。例如，消息内容可能损坏或者可能发送不存在的消息或者实际的消息可能被传递多次。信道的随机故障很少，因为通信软件能识别它们并拒绝出错的消息。例如，校验和可用于检测损坏的消息，消息序列号能用于检测不存在和重复的消息。

时序故障 时序故障适用于同步分布式系统。在同步系统中，进程的执行时间、消息传递时间和时钟漂移率均有时间限制。时序故障见图2-12的列表。这些故障中的任一个均可导致 56
在指定时间间隔内对客户没有响应。

故障类型	影　响	描　述
时钟	进程	进程的本地时钟超过了与实际时间的偏移率的范围
性能	进程	进程超过了两个进程步之间的间隔的范围
性能	通道	消息传递花了比规定的范围更长的时间

图2-12 时序故障

在异步分布式系统中，一个负载太重的服务器可能反应很慢，但我们不能说它有时序故障，因为它不提供任何保证。

实时操作系统是以提供时序保证为目的而设计的，但它们在设计上很复杂，可能要求冗余的硬件。大多数通用的操作系统如UNIX不能满足实时限制。

时序与有音频和视频通道的多媒体计算机密切相关。视频信息要求传输大量的数据。要做到没有时序故障地传递这样的信息对操作系统和通信系统两者都有特殊的要求。

故障屏蔽 分布式系统中的每个组件通常基于其他组件构造。可以从存在故障的组件构造可靠服务。例如，持有复制数据的多个服务器在其中一个服务器崩溃时能继续提供服务。了解组件的故障特性能使在设计新服务时屏蔽它所依赖的组件的故障。或者通过隐藏故障，或者通过将某一故障转换成一个更能接收的故障类型，一个服务可以屏蔽一个故障。作为后者的一个例子，校验和用于屏蔽损坏的消息，它有效地将随机故障转化为遗漏故障。我们将在第3章和第4章看到，可以使用将不能到达目的地的消息重传的协议隐藏遗漏故障。第14章给出了如何利用复制实现故障屏蔽。甚至也可以通过替换进程并根据原进程存储在磁盘上的信息恢复内存来实现屏蔽进程崩溃。

一对一通信的可靠性 虽然基本的信道可能有上面描述的遗漏故障，但用它来构造一个能屏蔽某些故障的通信服务是可能的。

术语可靠通信用下列有效性和完整性术语定义：

- 有效性 在外发消息缓冲区的任何消息最终被发送到接收消息缓冲区。
- 完整性 接收到的消息与发送的一致，没有消息被发送两次。 57

对完整性的威胁有两个独立的来源：

- 任何重发消息但不拒绝到达两次的消息的协议。协议能给消息附加上顺序号以便检测这些消息是否到达了两次。
- 可能插入伪造消息、重放旧的消息或篡改消息的恶意用户。在面对这种攻击时，为维护完整性要采取安全措施。

2.3.3 安全模型

在2.2节中，我们知道资源共享是分布式系统的促动因素，我们用封装对象的进程和通过与其他进程交互提供访问来描述它们的系统体系结构。那个体系结构模型为我们的安全模型提供了基础：通过使进程和用于进程交互的通道获得安全以及保护所封装的对象不被未经授权的用户访问而获得分布式系统的安全。

虽然这些概念可平等地应用到所有类型的资源，但是可以以对象进行描述。

保护对象 图2-13给出了代表一些用户管理一组对象的一个服务器。用户运行客户程序，由客户程序发送调用给服务器以完成在对象上的操作。服务器完成每个调用指定的操作并将结果发给客户。

对象可按不同的方式出不同的用户使用。例如，有些对象持有用户的私有数据，如他们的邮箱，而其他对象可能持有共享数据，如Web页面。为了对此进行支持，访问权限指定了谁被允许执行一个对象的操作。例如，谁被允许读或写它的状态。

这样我们必须在我们的模型中将用户包含在访问权限的受益人当中。我们将每个调用和每个结果均与对应的许可相联系。这样的一个许可称为一个委托（方）。一个委托方可以是一个用户或进程。在我们的图示中，调用来自用户，结果来自服务器。

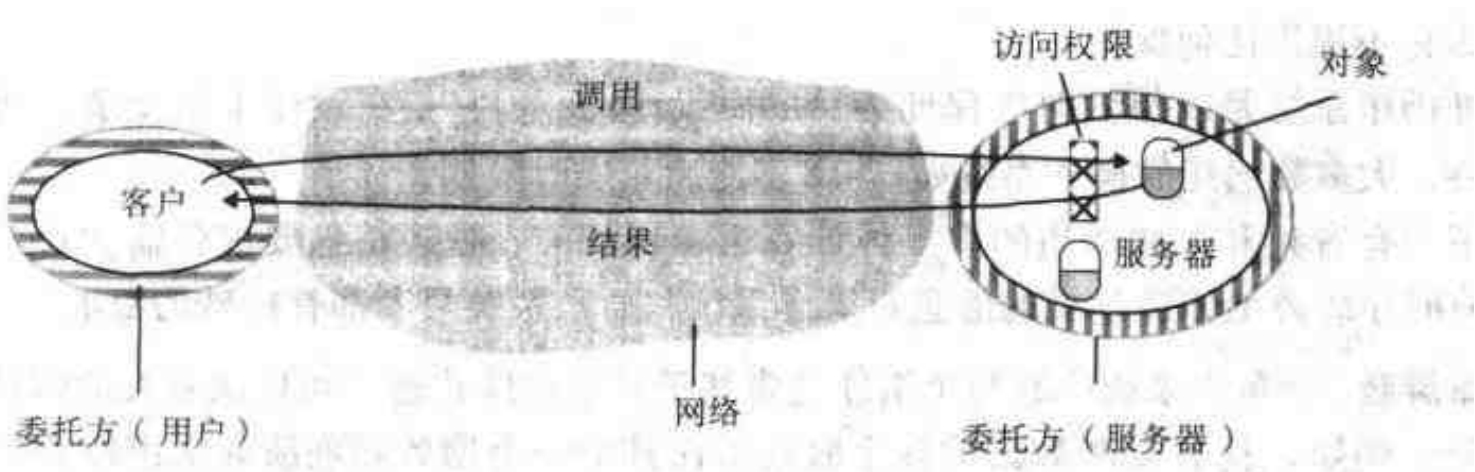


图2-13 对象和委托方

服务器负责验证在每个调用后面的委托方身份，检查它们是否有足够的访问权限在所调用的特定对象上完成所请求的操作，对没有权限的就拒绝它们的请求。客户可以检查服务器后面的委托方身份以确保结果来自所请求的服务器。

保护进程和它们的交互 进程通过发送消息进行交互。消息之所以容易受到攻击是因为它们所使用的网络和通信访问是开放的，这是为了使得任一对进程可以进行交互。服务器和对等进程对外提供它们的接口，使得任何其他进程能发送调用给它们。

分布式系统经常在可能受到敌对用户外部攻击的任务中使用和部署。对关键的信息，尤其是金融交易、机要、秘密信息或任何其他具有保密性或完整性的应用而言，这一点是千真万确的。完整性会由于安全违例以及通信故障而受威胁。所以，我们知道有可能存在对组成这样的应用的进程的威胁和来自在进程之间传送的消息的威胁。为了识别和解除这些威胁，

我们如何分析它们呢？下面的讨论介绍了一个分析安全威胁的模型。

敌人 为了给安全威胁建模，我们假定敌人（有时也称为对手）能发送任何消息给任何进程，读取或复制在一对进程之间的任何消息，如图2-14所示。这种攻击实现很简单，可利用连在网络上的计算机运行一个程序，读取发给网络上其他计算机的网络消息，或是运行一个程序生成假的服务请求消息并声称来自授权的用户。攻击可能来自合法连到网络上的计算机，也可以来自以非授权方式连接到网络上的计算机。

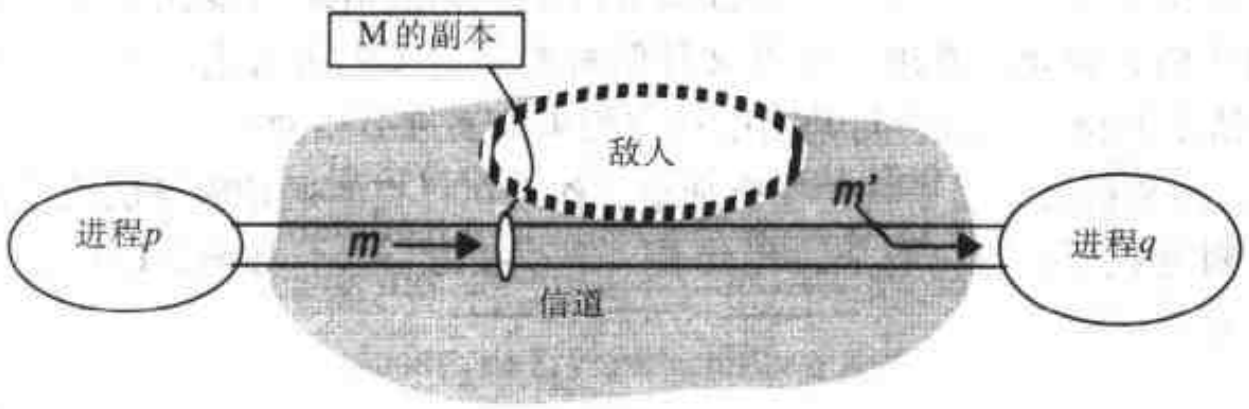


图2-14 敌人

在下面讲述对进程的威胁、对信道的威胁和拒绝服务时，将讨论来自一个潜在敌人的威胁。

对进程的威胁 在分布式系统中，一个用于处理到达的请求的进程可能会接收到来自其他进程的消息，它没有必要决定发送方的身份。通信协议，如IP，确实在每个消息中包括了源计算机的地址，但对一个敌人而言用一个假的源地址生成一个消息并不困难。对消息源缺乏可靠的知识是对服务器和客户能够正确工作的一个威胁，解释如下：

59

- 服务器 因为服务器能接收来自许多不同客户的调用，所以它未必能决定进行调用的委托方身份。即使服务器要求在每个调用中加入委托方的身份，敌人也可能用假的身份生成一个调用。在没有可靠的关于发送方身份情况的时候，服务器不能断定是执行操作还是拒绝操作。例如，邮件服务器不知道用户是否允许从指定邮箱中请求一个邮件项目，也不知道这个调用是否来自一个敌人。
- 客户 当客户接收到服务器调用的结果，它未必能区分结果消息来源，是来自预期的服务器还是来自一个“哄骗”邮件服务器的敌人。这样客户可能接收到一个与原始调用无关的结果，如一封假的邮件（不在用户邮箱中的邮件）。

对信道的威胁 一个敌人在网络和网关上行进时能拷贝、改变或插入消息。当信息在网络上传递时，这种攻击对信息的私密性和完整性构成威胁，对系统的完整性也构成威胁。例如，包含用户邮件的结果消息可能会泄漏给另一个用户，也可能被改成完全不同的东西。

另一种形式的攻击是试图保存消息的副本并在以后重放，这可能导致反复重用同一消息。例如，有些人通过重发从一个银行账户转账到另一个银行账户的请求调用消息而受益。

利用安全通道可解除所有这些威胁，安全通道技术基于密码学和认证，见下面的描述。

解除安全威胁 这里介绍安全系统的主要技术基础。第7章详细讨论安全分布式系统的设计和实现。

密码学和共享秘密 假设一对进程（例如，一个特定的客户和一个特定的服务器）共享一个秘密；它们两个知道秘密，但分布式系统中没有其他进程知道。如果由一对进程交互的消息包括证明发送方共享秘密的信息，那么接收方确实知道发送方是一对进程中的另一个进程。

当然，必须小心确保共享的秘密不泄露给敌人。

密码学是保持消息安全的科学，加密是将消息编码以隐藏其内容的过程。现代加密学基于使用密钥的加密算法——利用很难猜测的大数——来传输数据，这些数据只能用相应的解钥恢复。

认证　共享秘密和加密给消息的认证提供了基础，即证明由发送方提供的身份。基本的认证技术是在消息中包含加密部分，由它包含足够的消息内容以保证它的真实性。对于文件服务器的一个读取部分文件的请求，其认证部分可能包括所请求的委托方身份的表示、文件的
60 身份、请求的日期和时间，都用一个在文件服务器和请求的进程之间共享的密钥加密。服务器能解密这个请求并检查它是否与请求中指定的未加密细节相对应。

安全通道　加密和认证用于构造安全通道，作为在已有的通信服务层之上的服务层。安全通道是连接一对进程的信道，每个进程代表一个委托方行事，如图2-15所示。一个安全通道有下列特性：

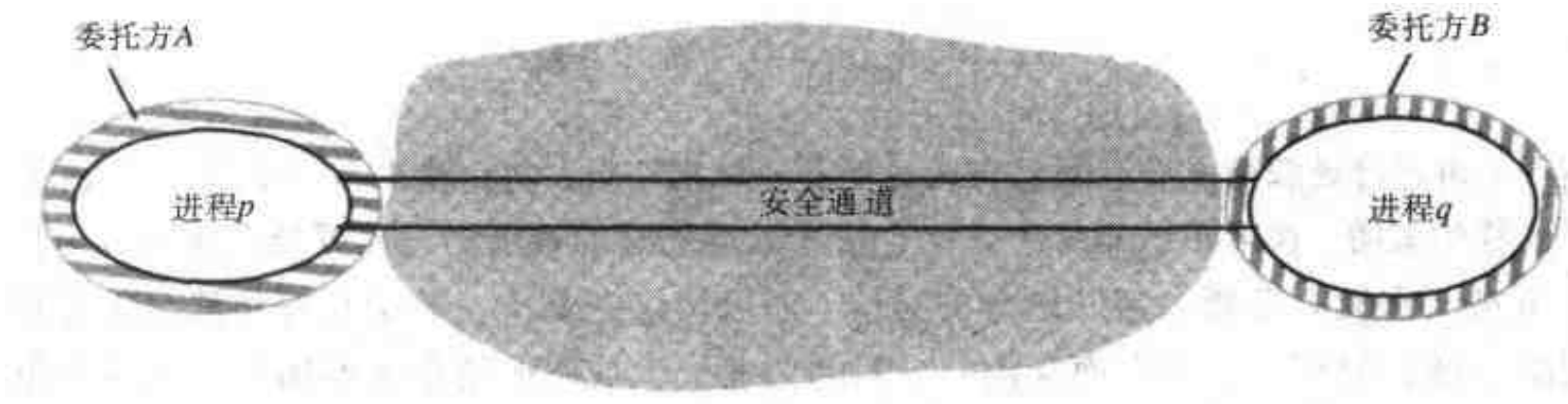


图2-15　安全通道

- 每个进程知道其他正在执行的进程所代表的委托方身份。因此，如果客户和服务器通过安全通道通信，那么服务器要知道发起调用的委托方身份，并能在执行操作之前检查它们的访问权限。这样服务器就能正确地保护它的对象，使客户相信它是从真实的服务器上接收到结果。
- 安全通道确保在其上传递的数据的私密性和完整性（防止篡改）。
- 每个消息包括一个物理的或逻辑的时间戳防止消息被重放或重排序。

构造安全通道的详细讨论见第7章。安全通道已成为保护电子商务和通信安全的一个重要的实用工具。虚拟私网VPN，（见第3章的讨论）和安全套接字层（SSL）协议（见第7章的讨论）就是实例。

其他可能的来自敌人的威胁　1.4.3节简单地介绍了两个安全威胁，服务拒绝攻击和移动代码的部署。作为敌人破坏进程活动的可能的机会，我们要再说一下这两个安全威胁。

- 拒绝服务　这种攻击形式是敌人通过超量地、无意义地调用服务或在网络上进行消息传递，干扰授权用户的活动，导致物理资源（网络带宽、服务器处理能力）的过载。这种攻击通常目的是延迟或阻碍其他用户的动作。例如，建筑物中控制电子门锁的计算机可
61 能由于受到非法请求这类攻击而导致电子锁失效。
- 移动代码　对于接收和执行来自其他地方的程序代码的进程，如1.4.3节提到的邮件附件，移动代码带来了新的有趣的安全问题。这样的代码可以很容易地扮演特洛伊木马的角色，声称完成的是无害的事情，但事实上可能包含了能够访问或修改资源的代码，这些资源对宿主进程是合法可用的，但对代码的编写者是不合法的。实现这种攻击有多种不同的

方法，主机环境必须非常小心地构造以避免这些攻击。这些问题中的大多数已在Java和其他移动代码系统中解决了，但从最近的一段历史看，移动代码问题暴露了一些让人窘迫的弱点。这一点很好地表明了所有安全系统的设计需要进行严格的分析。

安全模型的使用 有人可能会认为，在分布式系统中可以直接获得安全，它涉及根据预定义的访问权限控制对象的访问以及通信安全通道的使用。不幸的是，通常并不是这样。安全技术，如加密和访问控制的使用会带来实质性的处理和管理开销。上面概述的安全模型提供了分析和设计安全系统的基础，以保持这些开销最小，但对分布式系统的威胁会在许多地方发生，需要对系统网络环境、物理环境和人际环境中所有可能引发的威胁进行仔细的分析。这个分析涉及构造威胁模型，由它列出系统会遭遇的所有形式的攻击、风险评估和每个威胁的后果。要权衡为对抗威胁所需的安全技术的有效性和开销。

2.4 小结

大多数分布式系统根据一种体系结构模型设计。客户－服务器模型是流行的——Web和其他因特网服务，如FTP、新闻和邮件以及DNS归档和其他本地服务均是基于这个模型。像DNS有大量的用户并管理大量信息的服务是基于多个服务器的，并使用数据分区和复制提高可用性和容错，如DNS等。客户和代理服务器上的缓存被广泛地使用，以提高服务的性能。在对等进程模型中，分布式应用进程直接相互通信，不用服务器做中介。

从一个进程到另一个进程移动代码的能力导致了客户－服务器模型的一些变种。最常见的例子是小程序，它的代码由Web服务器提供，在客户方运行，给客户提供其没有的功能，并由于代码靠近用户而提高了性能。

便携式计算机、PDA和其他数字设备的存在以及与分布式系统的集成允许用户，在远离桌面计算机的时候，访问本地和因特网服务。在日常物品中具有无线通信能力的计算设备，如洗衣机、防盗警报器的出现，允许它们提供在家庭无线网络中可访问的服务。分布式系统 62
中移动设备的特征是不能预测它们何时连接和断连，从而导致对发现服务的需求。通过发现服务，移动服务器能提供服务，移动客户也能查找它们。

我们给出了交互模型、故障模型和安全模型。它们识别出构造分布式系统的基本组件的公共特征。交互模型牵涉到进程和信道的性能以及缺乏全局时钟，它将同步系统等同于在进程执行时间、消息传递时间和时钟偏移上有已知范围的系统，将异步系统等同于在进程执行时间、消息传递时间和时钟偏移上没有限制的系统，后者是对因特网行为的描述。

故障模型将分布式系统中的进程故障和基本的信道故障进行了分类。屏蔽是一项技术，依靠它，可将不太可靠的服务中的故障加以屏蔽，并基于此构造出较可靠的服务。特别是，通过屏蔽基本信道的故障可从基本的信道构造出可靠的通信服务。例如，遗漏故障可通过重传丢失的消息加以屏蔽。完整性是可靠通信的一个性质，它要求所接收到的消息与发送的消息一致，并且没有消息被发送两次。有效性是另一个属性，它要求任何放在外发缓冲区中的消息最终被传递到接收消息缓冲区。

安全模型识别出在一个开放的分布式系统中对进程和信道可能的威胁。有些威胁与完整性有关：恶意用户可能篡改消息或重播消息。其他的威胁涉及到私密性。另一个安全问题是如何认证消息发送所代表的委托方（用户或服务器）。安全通道使用密码技术，确保了消息的完整性和私密性，并使得相互通信的委托方可以进行验证。

练习

2.1 描述一个或多个主要的因特网应用（如Web，电子邮件或网络新闻）的客户－服务器体系结构并给出图解。

2.2 对于练习2.1中描述的应用，叙述服务器如何协作来提供服务。

2.3 练习2.1中讨论的应用是如何进行服务器之间的数据分区和复制（或缓存）？

2.4 搜索引擎是一个Web服务器，它响应客户的请求，在它存储的索引中查找，并（同时）运行几个Web蜘蛛任务创建和修改索引。如何使这些同时运行的活动之间同步？

2.5 举出对等进程模型的一些应用例子，区分所有对等状态需要一致的情况和需要较少一致性的情况。

2.6 列出易受不可靠程序攻击的本地资源的类型。不可靠程序是指从远地下载并在本地
63 运行的程序。

2.7 给出受益于移动代码的应用例子。

2.8 哪些因素会影响访问由服务器管理的共享数据的应用的反应能力？描述可用的补救方法并讨论它们的有效性。

2.9 区分缓冲和缓存。

2.10 给出在分布式系统中能/不能通过使用冗余来容错的软硬件故障的例子。在适当的情况中冗余使用到什么程度能使得系统容错？

2.11 考虑一个简单的服务器，它不用访问其他服务器就可完成客户请求。解释它通常为什么不可能设置服务器响应客户请求的时间限制。如何才能使服务器在一定时间范围内执行请求？这是一个实用的选择吗？

2.12 针对在信道上的两个进程之间传递消息所花费的时间，对影响该时间的每个因素，陈述需要什么手段才能对它占用的总时间设置一个限制。为什么这些手段在当前通用的分布式系统中没有提供？

2.13 网络时间协议服务能用于同步计算机时钟。解释为什么使用了该服务，对两个时钟之间的不同也还是不能给出保证的范围。

2.14 考虑在异步分布式系统中使用的两个通信服务。在服务A中，消息可能丢失、被复制或延迟，校验和仅应用到消息头。在服务B中，消息可能丢失、延迟或发送地太快以致接收方无法处理它，但到达目的地的消息内容一定正确。

描述每个服务可能出现的故障类型。根据对有效性和完整性的影响对故障分类。服务B能被描述成一个可靠的通信服务吗？

2.15 考虑一对进程X和Y，它们使用练习2.14的通信服务B相互通信。假设X是客户而Y是服务器，调用由从X到Y的请求消息开始，然后Y执行该请求，最后以从Y到X的应答消息结束。描述一个调用可能遇到的故障类型。

2.16 假设一个基本的磁盘读操作有时读取的值与写入的值不同。叙述基本的磁盘读操作可能发生的故障类型。建议如何能屏蔽故障以产生一种不同的良性故障，并建议如何屏蔽良性故障。

2.17 给出可靠通信的完整性的定义，列出所有可能来自用户和来自系统组件的对完整性的威胁。面对每种威胁，需要采取什么手段确保完整性。

2.18 描述可能发生在因特网上的几类主要的安全威胁（对进程的威胁、对信道的威胁、
64 拒绝服务）可能的发生情况。

第3章　网络和网络互联

3.1　简介
3.2　网络类型
3.3　网络原理
3.4　因特网协议
3.5　网络实例研究：以太网、无线LAN和ATM
3.6　小结

分布式系统使用局域网、广域网和互联网进行通信。底层网络的性能、可靠性、伸缩性、移动性以及服务质量都影响着分布式系统的行为，从而也影响这些系统的设计。用户需求的改变已经导致无线网络的出现和有服务质量保障的高性能网络的出现。

计算机网络原理包括协议的分层、包交换、路由以及数据流等。网络互联技术使得异构网络可以集成在一起。因特网就是一个重要的例子；它的协议广泛地应用于分布式系统。因特网中使用的寻址以及路由的模式已经经受了因特网快速成长所带来的考验。它们也在不断地被修改，以适应未来的发展，满足对移动性、安全性以及服务质量等的新需要。

在实例研究中给出特定网络的技术设计，包括以太网、异步传输模式（ATM）网络和IEEE 802.11无线网络标准。

3.1　简介

构造分布式系统所使用的网络首先需要众多的传输介质，包括电线、电缆、光纤以及无线频道；然后需要一些硬件设备，包括路由器、交换机、网桥、网络集线器、转发器和网络接口；最后还需要软件部分，包括协议栈、通信处理器和驱动器。以上这些都影响分布式系统和应用程序所能达到的最终功能和性能。我们把为分布式系统提供通信设施的软硬件成分称为通信子系统。计算机和其他使用网络进行通信的设备称为主机。而术语结点则用来称呼任何在网络上的计算机或者转发设备。

因特网是一个单一的通信子系统，它为所有接入的主机提供通信服务。因特网连接了大量采用不同网络技术的子网。一个子网是一组互联的结点，它们之间采用同样的技术进行通信。因特网的底层组织包括一个体系结构和软硬件成分，它们有效地将不同的子网集成为一个单一的数据通信服务。

通信子系统的设计受分布式系统的计算机所使用的操作系统的特征的影响，也受互联计算机的网络的影响。本章考虑了网络技术对通信子系统的影响，操作系统问题将在第6章讨论。

本章将从分布式系统的通信需求角度，对计算机网络作一个介绍性的概述。对计算机网络还不熟悉的读者应该将本章作为本书后面部分的基础，而对网络比较熟悉的读者也会发现本章是对计算机网络的诸多方面，尤其是和分布式系统有关的方面的一个扩展性总结。

计算机发明后不久就有了计算机网络的构思。1961年Leonard Kleinrock[1961]第一次在一篇文章中提出了包交换的基础理论。1962年，两位在20世纪60年代初期在MIT参加研制了第

一个分时系统的科学家J.C.R. Licklider和W. Clark在一篇论文中讨论了交互计算和广域网络的巨大潜能，这在某些方面也预示了因特网的将来[DEC 1990]。1964年，Paul Baran描绘出了一个可靠有效的广域网的实用设计轮廓[Baran 1964]。更多的有关计算机网络和因特网历史的资料和链接可以在下列资源中找到[www.isoc.org, Comer 1995, Kurose and Ross 2000]。

本节剩余部分将讨论分布式系统的通信需求。3.2节概括网络类型，3.3节将介绍计算机网
66 络原理。3.4节特别讨论因特网的相关方面。本章将在3.5节给出以太网、ATM和IEEE 802.11（WaveLAN）网络技术的实例研究。

分布式系统的网络问题

早期的计算机网络只是用来满足少量的、相对简单的应用要求，支持像文件传输、远程登陆、电子邮件、新闻组等的网络应用。随着分布式系统的不断发展，分布式应用程序需要能访问共享的文件或其他资源，为完成这样的交互应用，必须提出更高的性能标准。

近来，随着因特网的发展和商业化以及多种新使用模式的出现，对于可靠性、伸缩性、移动性、安全性和服务质量等的更高要求也开始出现。本节将对这些需求的本质给出详细的定义和描述。

性能 我们主要感兴趣的网络性能参数是那些影响两个相连计算机间消息传输速率的。它们分别是：等待时间和点到点间的传输率。

- 等待时间是指在执行发送操作之后到数据在目标地可用这一段时间。它可以用传输一个空消息的时间来度量。
- 数据传输率是指一旦输出过程开始，数据在网络上两台计算机间的传输速度，通常用bps（比特/s）作为单位。

根据这些定义，在两个计算机间传输length长度的消息网络所需要的时间为：

$$\text{消息传输时间} = \text{等待时间} + \text{length}/\text{数据传输率}$$

以上等式还须满足的条件是：消息长度不能超过网络所允许的最大值。长消息会被分成多个段，而总的传输时间也相应地变成这多个段传输时间的总和。

网络传输率主要是由网络的物理特性决定。而等待时间则主要由软件开销、路由延迟、源于争夺传输信道的依赖负载的统计因素这三方面组成。分布式系统在进程之间传送的消息有许多是很小的，因此等待时间在决定性能上与数据传输率有相同或更重要的意义。

网络的系统总带宽是吞吐量的度量——在给定的时间内网络可以传输的数据总量。在许多局域网技术中，如以太网，每一次的数据传输都用到了整个网络的传输能力，这时系统的带宽也就是数据传输率。但在大部分广域网中，消息可以同时在几个不同的信道中传输，这时系统总带宽和传输率没有直接的关系。网络性能在网络超载时会恶化——超载是指太多消息同时在网络中传输。超载给网络的等待时间、数据传输率以及系统总带宽所带来的精确影响
67 极大地依赖于网络技术。

现在考虑客户－服务器间通信的性能。在由标准PC或Unix系统构造的负载较轻的本地网环境下，传输一个短请求消息加上收到一个短应答的总时间，通常在一个毫秒文内。而调用一个已经在本地内存中的应用层对象操作，所需的时间是微秒以下。这也就是说，即使网络性能发展得再快，在本地网中访问共享资源的时间依然要比访问已经在本地内存中的资源慢1000倍或更多。另一方面，对于通过高速网络访问一个本地的Web服务器或者文件服务器，

同时对经常使用的文件放入一个大的缓存，那么其性能通常可以达到或超过直接访问本地硬盘文件的性能。因为网络的等待时间和带宽经常超过硬盘的性能。

因特网上往返的延迟平均在300ms~600ms之间，比快速的本地网要慢500倍。这块时间来自路由器的交换延迟和网络电路的竞争。

6.5.1节将详细地讨论和比较本地操作和远程操作的性能问题。

可伸缩性 计算机网络是现代社会不可缺少的基础设施。表1.4显示了20年来连入因特网中计算机主机数量的增长情况。未来因特网的大小将可能和地球上人口数量相当。到时将有数十亿的结点和近亿的主机连入网络。

这些数字表明因特网必须处理在数量和负载上的巨大变化。因特网所基于的网络技术设计时并未考虑网络现在的规模；但它们仍表现得相当地好。为了适应因特网下一阶段的发展，寻址和路由机制正在计划做出一些实质性的改变，这些将在3.4节加以讨论。

因特网没有可用的流量数字，不过可以从通信等待时间感受流量对性能的影响。可以在[www.mids.org]网址看到一组过去的等待时间数字和现在的等待时间数字。虽然大家都说"www是the world wide wait"，但这些数字表明网络的等待时间近年已减少到平均往返时间为100ms~150ms。偶尔也有较大的变化，等待时间的峰值会达到400ms。对于Web用户来说，这些很可能还不是他们经历的延迟的主要部分。简单的客户－服务器应用，比如Web，未来的数据流量将会和上网用户数量成正比的发展。因特网基础设施的能力若想支持这样的增长，必须依赖经济合理的使用这些设施，特别是对用户的处置和实际发生的通信模式——比如他们的位置范围。

可靠性 我们在2.3.2中对于故障模型的讨论中描述了通信错误的影响。许多应用可以从通信故障中恢复，因此并不要求保证无错通信。端对端间的争论（2.2.1节）也进一步支持了"通信子系统无须提供完全无错的通信"的观点；通信错误的检测和更正经常由应用层软件完成。大多数物理传输介质的可靠性很高。出现的错误通常是由于发送方或接收方中的软件故障或者是缓冲溢出，而不是网络错误。 68

安全性 第7章列出了分布式系统获得安全所需的需求和技术。大多组织采用的第一层防御是为他们的网络和计算机设立一个防火墙。防火墙在组织机构的局域网和因特网之间创建了一个保护的边界，其目的是保护组织中所有计算机的资源不受外面用户和进程的访问，并能控制组织中用户使用防火墙外的资源。

防火墙在网关上运行——所谓网关是在因特网到组织企业内部网入口点处的计算机。防火墙接收并且过滤所有进出这个组织的消息。防火墙通常按照组织的安全策略进行配置，允许某些进入或外出的消息通过，扔掉所有其他的。这一部分我们将到3.4.8节再谈。

为了让分布式程序在防火墙的限制下依然可以执行，我们需要建立一个安全的网络环境，可以部署大部分分布式应用，且具有端对端的认证、私密性和安全性。使用密码技术可达到这种细粒度的更灵活的安全形式。它通常应用于通信子系统以上的层次，因此不在本章讨论这一点，而是放在第7章。但也有一些例外的要求，包括保护网络成分如路由器免于对操作的未授权的干涉，对移动设备和其他外部结点建立安全链接以便使得它们能参加到一个安全的企业内部网——即虚拟私有网络（VPN）的概念，VPN将在3.4.8节讨论。

移动性 第2章介绍了分布式系统支持便携式计算机和手持数字设备的需求，以及为了支持与这些设备进行连续通信所要求的无线网络。移动的重要性已不仅限于对无线网络的需要。

移动设备常常改换所在的位置，在方便的网络连接处重新连入。因特网和其他网络的寻址和路由机制都是在移动设备出现之前开发的。虽然现有的机制已经进行了改进和扩展来解决这些问题，但随着未来使用移动设备的增长，还必须进行更进一步的扩展。

服务质量 在第2章中，我们把服务质量定义为：在传输和处理实时多媒体数据时满足期限要求的能力。这也给计算机网络提出了新的要求。传输多媒体数据的应用要求所使用的信道有足够的带宽和受限的等待时间。一些应用能动态地改变它们的要求，并同时指定最小可接受的服务质量和期望的最佳值。第15章将讨论如何提供这些保证和相关的维护。

组播 分布式系统中大部分的通信是在一对进程之间的，但也经常有一对多通信的需求。
69 显然这可以用向多个地址发送来模拟，但它要花的代价比需要的大，而且也不具有应用所需的容错性。因此，许多网络技术都支持同时向多个接收方传递消息。

3.2 网络类型

本节介绍用于支持分布式系统主要的网络类型：局域网、广域网、城域网、无线网和互联网。

一些网络类型的名字经常会被搞混，因为它们看上去指的是物理区域（局域，广域），其实它们也区分了物理传输技术和底层的协议。对于局域网和广域网来说，这些方面是不一样的。但最近开发的网络技术如ATM（异步传输模式）就可以同时适用于局域网和广域网。无线网络也同时支持局域网和城域网。

我们把由很多互联的网络组成，并且集成起来提供单一数据通信介质的网络称为互联网。因特网就是典型的互连网；如今它包括了成百上千的局域网、城域网和广域网。我们将在3.4节详细描述它的实现。

局域网（LAN） LAN在由单一通信介质连接的计算机之间以相对较高的速度传输消息。这里的介质包括双绞线、同轴电缆和光纤。段是指为一部门或者一楼层很多电脑服务的那部分电缆。在段中，消息不需要路由，因为段中的计算机都有直接的连接。整个系统的带宽由连接在段中的计算机共享。在大一些的局域网，比如校园网或者为办公楼服务的网络，由许多段组成，段之间通过交换机或集线器连接（详见3.3.7节）。对于局域网来说，除了消息流量很大的时候，系统总带宽较高，而等待时间较短。

20世纪70年代人们开发了几种局域网技术——以太网、令牌网和有槽环形网。每一个都提供了有效和高性能的解决方案，但最终以太网成为有线局域网中权威性的技术。它最初产生在20世纪70年代的早期，当时的带宽是10Mbps，最近扩展了100Mbps和1000Mbps的新版本。以太网操作的原理将在3.5.1节中加以描述。

局域网的适用性很强，它几乎可以工作在所有的工作环境中，只须有一两台以上的个人计算机或者工作站。它们的性能对实现分布性系统和应用来说已经足够了。以太网技术缺乏许多多媒体应用所需的等待时间和带宽保证。ATM网络的开发填补了这个空白，但它们昂贵的开销限制了它们在局域网应用中的使用。而高速以太网采用了交换模式，在很大程度上克
70 服了上述缺点，虽然还不如ATM有效。

广域网（WAN） WAN在不同组织甚至被远距离分隔开的结点之间以低速传递消息。这些结点可能在不同的城市、国家甚至不同的洲际。其通信介质是连接专用计算机（称为路由器）的通信电路。它们管理整个通信网络，并将消息或数据包路由到指定的地点。大多数的

网络中，路由操作在每个路由点都引进了一定的延迟，因此消息传送总的等待时间取决于消息路由和消息经过的网络段的流量负载。在如今的网络中，这些等待时间可能达到0.1s~0.5s这么长。最新的数字能从下面的网址[www.mids.org]中找到。大多数介质的电信号速度接近光速，这就给长距离网络的传输等待时间设置了一个下界。举例来说，一个电信号从欧洲到澳大利亚的时间大约是0.13s，而所有从地球同步卫星路由的传输信号会有大约0.2s的延迟。

因特网上各连接处的带宽也很不一样。有些国家，部分因特网上可以达到1Mbps~2Mbps的速度，但通常情况下，还是10Kbps~100Kbps的速度。

城域网（MAN） 这种类型的网络基于高带宽的铜线和光纤电缆，用来在距离50km以内的城镇传输视频、音频或者其他数据。这种电缆可以加以开发，以提供分布式系统所需的数据传输率。从以太网到ATM，人们已经使用了多种技术来实现在MAN中的数据路由。IEEE已经颁布了相应的标准802.6[IEEE 1994]，以满足MAN中的各种需要，该标准的实现目前正在进行。MAN仍处于它的幼年阶段，但它可能满足局域网能满足的需要并且能跨越很大的距离。

以目前在某些城市可用的DSL（数字用户线）和电缆调制解调器为例，通常DSL使用电话交换机中的ATM交换将数字信号路由到订购者的家里或办公室的双绞线上（与用于电话连接的电线一样），速度大约在0.25Mbps~6.0Mbps范围。因为使用的是双绞线，所以限制了订购者和交换机的距离在1.5km之内。电缆调制解调器这种连接方式是在同轴电缆架构的有线电视网络上进行传输，速度可以达到1.5Mbps，其范围大大地超过了DSL。

无线网 如在第2章所提到的一样，便携和手持设备的入网，需要无线通信的支持。最近已出现了很多数字无线通信技术。有一些是*无线局域网*（WLAN），用于代替有线LAN，比如IEEE 802.11标准（WaveLAN）在150m的范围提供2Mbps~11Mbps的数据传输。其他还有一些技术，可以用来连接附近的移动设备或者固定的设备。例如，可以将它们连入本地的打印机或者掌上电脑或者桌面计算机。有时，这些被称为*无线个域网*（WPAN）；常见的例子有红外连接（这都包括在掌上电脑和膝上计算机中）和蓝牙低耗无线电网络技术（在它上面可以在10m以内进行1Mbps~2Mbps速度的数据传输）[www.bluetooth.com]。很多移动电话网络基于数字无线网络技术，其中包括欧洲的GSM（全球移动通信系统）标准，这已经在世界上很多国家得以应用。在美国，现在大部分的移动电话是基于模拟AMPS蜂窝式无线电网络，CDPD（蜂窝式数字包数据）是位于其上的数字通信方法。通过使用蜂窝式无线连接，移动通信网络可以在非常广阔的区域操作（通常是整个国家或整个大洲）；它们的数据传输设施也为便携式设备提供了到因特网的广域移动连接。以上说的蜂窝网络提供相对低的数据率——9.6Kbps~19.2Kbps；而正在计划之中的下一个网络，其中在蜂房半径为几公里的情况下，数据传输率可达到128Kbps~384Kbps，若蜂房半径更小一些的话，速度可以提升到2Mbps。

71

由于便携式设备可用的带宽受限和其他一些限制，如它们的屏幕很小，所以开发了一个专为无线便携设备使用的协议WAP（无线应用协议）[www.wapforum.org]。

互联网 互联网是一个通信子系统，它将多个网络连接起来提供公共数据通信设施，但它隐藏了单个网络中的技术、协议以及用于互联的方法。

开发可伸缩、开放的分布式系统，需要类似互联网这样的网络。分布式系统的开放性特征暗示了，分布式系统所使用的网络应该是一个可扩展到大量计算机的网络，而单个网络的地址空间有限，有一些网络有性能限制，不宜于大规模的使用。在互联网中，众多的局域网和广域网技术——它们很可能是由不同的销售商提供的——为一组用户提供联网能力。这样，

互联网给分布式系统的通信提供了很多开放系统的好处。

互联网是由多种网络组建而成的。它们的互联是靠特定的称为路由器的计算机和通用的称为网关的计算机，集成通信子系统由软件层实现，它为整个网络的计算机提供寻址以及数据传输功能。可以把它想像为将互联网络层覆盖在由底层网络、路由器、网关等组成的一个通信介质上所构造的一个“虚拟网络”。因特网是网络互联的一个主要的例子，它所使用的TCP/IP协议是上面提到的集成层的一个很好的说明。

网络比较 图3-1给出了上面我们所讨论的各种网络类型的范围以及性能特点。而和分布式系统有关的额外需要比较的是不同网络中可能发生的故障频率和类型。除了在无线网络中数据包经常因为外部干扰而丢失之外，其他各种网络的底层数据传输机制的可靠性都很高。在所有网络中，都可能因为处理延迟或者目的地缓冲区溢出而引起数据包丢失。

	范围	带宽（Mbps）	等待时间（ms）
LAN	1km~2km	10~1000	1~10
WAN	世界范围	0.010~600	100~500
MAN	2km~50km	1~150	10
无线LAN	0.15km~1.5km	2~11	5~20
无线WAN	世界范围	0.010~2	100~500
因特网	世界范围	0.010~2	100~500

图3-1 网络的种类

数据包到达的顺序可以与发送的顺序不一样。这只出现在对分离的数据包可以单独路由的网络——主要是广域网。数据包的复制拷贝也可以被发送，但这一般是发送方假设以前发送的该数据包丢失了。数据包被重发后，接收方会同时收到原数据包和重发的数据包。

上面描述的所有错误都在TCP和其他所谓的可靠协议中被屏蔽了，这使得应用程序认为发送的消息都能被接收进程收到。而在分布式系统中，为了某些目的，可以使用一些相对不是很可靠的协议如UDP，在那样的环境下必须考虑出故障的可能性。

3.3 网络原理

计算机网络的基础是20世纪60年代发展起来的包交换技术。包交换这一步意义深远，它超越了使用电话电报通信的电信交换网，开发了计算机在数据传送时先存储的能力。这使得向多个地址发送的消息可以共享同一条通信链接。链接可用时，数据包按序排列在缓冲区中，然后发送。通信是异步的——消息经过一个延迟到达目的地，该延迟取决于数据包在网络中传递所花的时间。

3.3.1 数据包的传输

72
|
73

计算机网络的大多数应用需求是发送信息的逻辑单元或消息——任意长度的数据串。在消息传递前，它被分割成数据包。最简单的数据包格式是一定长度的二进制数据序列（比特或字节数组）以及识别源和目的地计算机的寻址信息。使用一定长度的数据包是为了：

- 网络中的每个计算机能为可能到来的最大的数据包分配足够的缓冲空间。
- 避免因长消息不加分割地传递而为等待信道空闲而引起的延迟。

3.3.2 数据流

以消息为基础的通信方法可以满足大多数应用的需要，但也有一些例外。我们在第2章中提到，多媒体应用中视频音频流的传输需要保证其速度和一定范围的等待时间。这样的流和数据包传输所针对的基于消息的流量类型有实质上的不同。视频音频流比分布式系统中其他大部分通信形式所需要的带宽都要高。

为了达到实时显示的目的，如果传输的是压缩的视频流数据，需要1.5Mbps的带宽，如果传输的视频流数据是未压缩的，则需要大约120Mbps。另外，和典型的客户－服务器交互程序断断续续的数据传递相对，这种流是连续的。多媒体元素的播放时间（对视频元素来说）是必须被显示的时间或（对声音采样而言）是必须转成音频的时间。举例来说，视频帧的流速是每秒包括24个帧。第*N*帧的播放时间是从流开始传输后的（*N*/24）秒。元素如果迟于它的播放时间到达目的地，它就不再有用，将被接收进程扔掉。

及时传输这种数据流依赖于具有一定服务质量的网络连接——带宽、等待时间和可靠性都必须有保证。现在所需要的是建立起多媒体流从源到目的地的通道，其中路由是预定义好的，每个结点为其经过和缓冲保留好了需要的资源以便为数据流中任何不规则的地方进行适当的修补。通过这个通道，数据可在要求的速率下从发送方传到接收方。

ATM网络（3.5.3节）特殊的设计提供了高带宽和低等待时间，并通过保留网络资源保证服务质量。IPv6，作为因特网新的网络协议，在未来10年中将得到广泛的应用（见3.4.4节的描述）。IPv6协议的一个特色就是每一个组成实时流的IP数据包都能在网络层被单独识别和处理。

通信子系统，若要提供服务质量保证，就要有能预分配网络资源并强行执行这些分配的设施。资源保留协议（RSVP）[Zhang *et al.*1993]使得应用能协商实时数据流的带宽预分配。实时传输协议（RTP）[Schulzrinne *et al.* 1996]是一个应用层的数据传输协议，它在每个数据包中包含了播放时间和其他定时要求。要在因特网中有效实现这些协议，传输层和网络层都必须做出实质性的改变。第15章将详细讨论分布式多媒体应用的需求。 74

3.3.3 交换模式

网络是一组由电路连接起来的结点组成的。为了能在任意两个结点间传输信息，交换系统是必不可少的。这里我们定义4种在计算机网络中使用的交换。

广播 广播是一种传输技术，不涉及交换。无论什么信息都将传给每一个结点。由接收方判断接收还是不接收。一些LAN技术，包括以太网，是基于广播的。无线网络也有必要基于广播，但是由于缺少固定电路，广播只能到达蜂窝结点。

电路交换 电话网曾经是惟一的电信网。它们的操作非常容易理解：当有人打电话时，本地电话交换总台的自动交换机就会自动将打出的电话线连入交换台，并连往所要接的电话线。长途电话的过程也是类似的，只不过要经过多个交换台而已。这种系统有时被称为老式电话系统或POTS。它是典型的电路交换网络。

包交换 计算机和数字技术的诞生为电信带来了新的契机。最根本的，它使得人们可以用机器处理和储存数据，这为通信网络的构造提供了一个全新的方法。这种新的通信网络类型叫作*存储转发网络*。存储转发网络并不是通过建立或取消连接来构造电路，而只是将数据包从它的源地址转发到目标地址。在每个交换结点上有一台计算机（也就是几个电路需要互连

的交汇处）。数据包到达一个结点后先存储在这个结点计算机的内存中，再由一个程序选择数据包的外出电路，将它们转向下一个离它们目的地更近的结点。

邮政系统就是一个信件的储存转发网络，由人和机器在信件分拣室完成这一处理。而在计算机网络中，数据包的存储和处理很快，以致人们得出瞬间传输的假象。至少在人们（包括电话工程师）考虑用计算机网络传输视频音频信息之前，一直以为是这样的。这种集成的优势是巨大的——只需要一个网络就可以处理计算机通信、电话服务、电视广播、无线电广播以及很多新的应用如电话会议。既然每样事物都要有它的数字形式，操纵和储存这些东西也都成为了可能。毋庸置疑，在这样一个集成的电信系统中，会诞生很多新的应用。

帧中继 数据包的存储转发型传输并不是瞬间完成的。一般情况下，每个结点转发一个数
75 据包需要的时间从几十微秒到几毫秒不等，取决于数据包的大小、硬件的速度和当时的流量情况。数据包在到达目的地址前，可能要通过很多的结点。众所周知因特网中大多基于存储转发交换，即使是很小的因特网数据包通常也需要200ms左右到达目的地。

这个量级的延迟对于电话这样的应用就太长了，要维持电话交谈，延迟不得超过50ms。因为延迟是累加的——数据包通过的结点越多，总的延迟时间也越长——并且每个结点上大部分的延迟来自包交换技术自身，所以对集成而言，这无疑是一个很严重的难题。

但电话和计算机网络的工程师是不会这么容易就缴械投降的。他们提出了另一种交换方法——帧中继——为包交换带来了一些电路交换的好处。结果出现了ATM网络。我们将在3.5.3节描述它们的操作。帧中继用快速地交换小的数据包（被称为帧）来解决延迟的问题。交换结点（通常是专用的并行数字处理器）通过检测帧的前几位信息来路由帧。帧并不作为一个整体存储在结点中，而是通过位流的形式通过结点。结果ATM技术在由很多结点组成的网络中传递数据包只需要几十微秒。

3.3.4 协议

术语协议是指为了完成一给定任务，进程间通信所要用到的一组众所周知的规则和格式。协议的定义包括两个很重要的部分：

- 必须被交换的消息的顺序约定。
- 消息中数据格式的约定。

协议的存在使得能独立地开发分布式系统的软件组件，并且能在代码次序不一样、数据表达不一样的计算机上用不同的程序语言实现分布式系统的软件组件。

一个协议是由分别位于发送方和接收方上的一对软件模块实现的。例如，一个传输协议将任意长的消息从一个发送进程传递给一个接收进程。想向另一个进程传输消息的进程给传输协议模块发出一个调用，并按指定的格式传递消息。传输软件接着负责将消息传递到目的地，它将消息分割成指定大小的数据包和格式，利用网络协议（另一个低层的协议）传输到目的地。接收方计算机中相应的传输协议模块通过网络层协议模块接收这些数据包，并在传递给接收进程之前，进行逆向的转换，重新生成消息。

协议层 网络软件是按层次排列的。每一层都为上面的层提供了相应的接口，并扩展了底层通信系统的性质。层表示成与网络相连的每一个计算机上的一个模块。图3-2说明了这个结构和通过分层协议传递消息时的数据流。每一个模块看起来都是和另一个计算机中相同层次的模块直接通信，但事实上数据并没有在两个同层次的模块上传输。网络软件的每一层都只

通过本地过程调用与它的上一层和下一层通信。

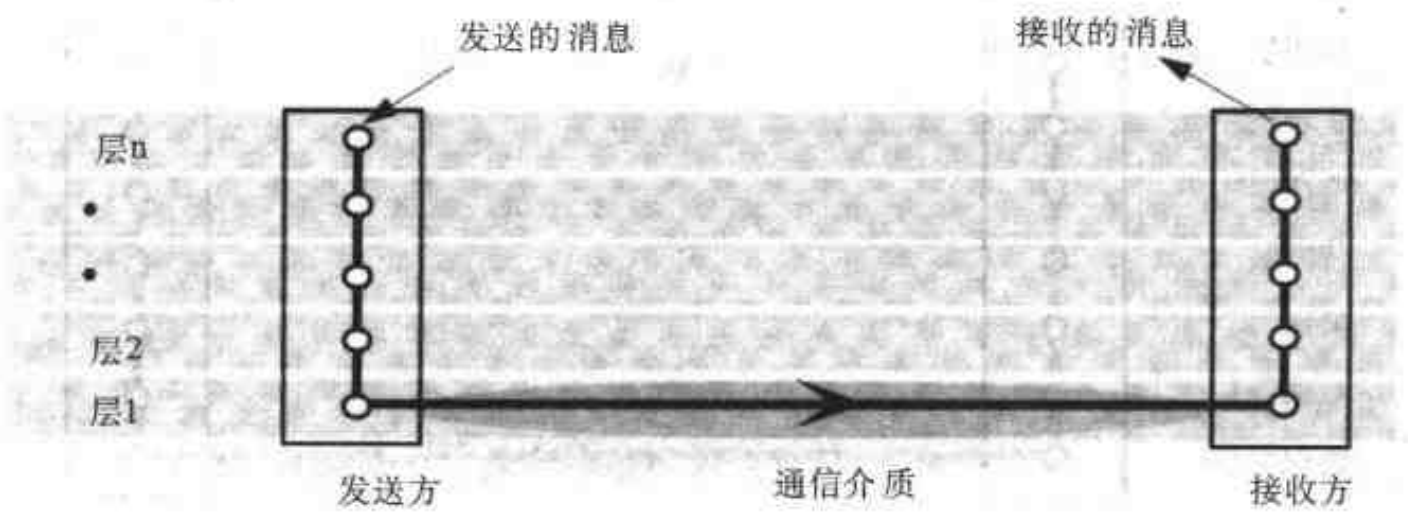
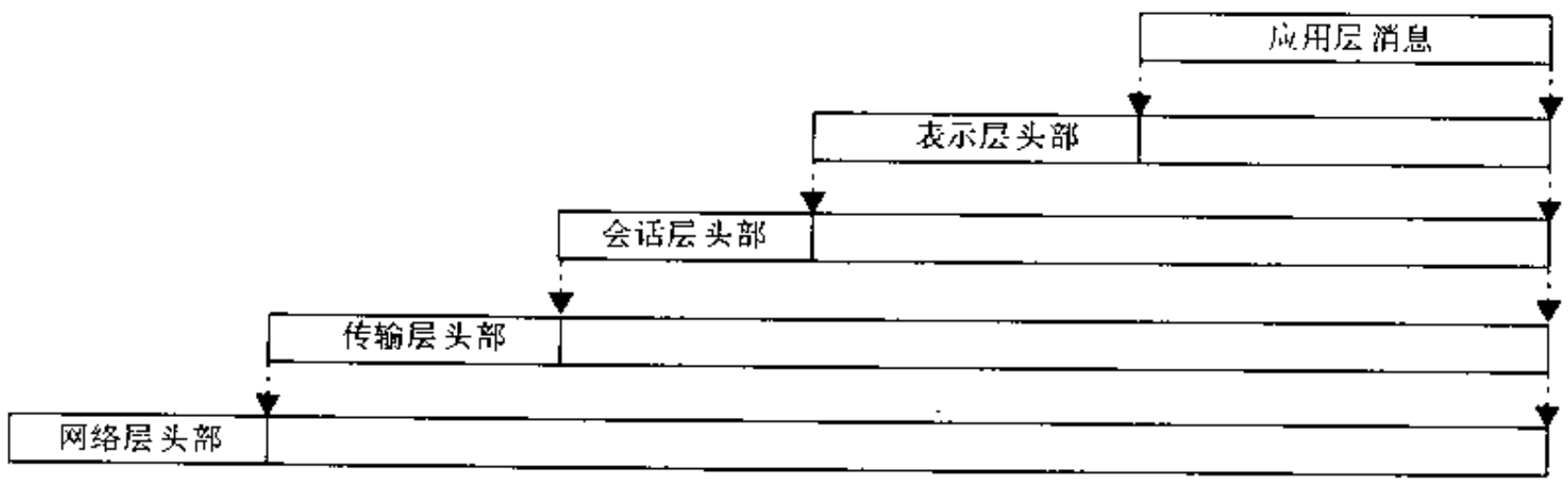


图3-2 协议软件的概念化分层

在发送方，每一层（除了最顶层，即应用层以外）从上一层按照指定的格式接收数据项，并在传送到下一层进行进一步处理之前，进行数据转换，按下一层的格式封装数据。图3-3说明了这一过程，图中该过程被应用于OSI协议组的头四个层。图中可以看出，数据包的头部包含了大部分网络相关的数据项，为了简洁起见，省去了在一些数据包类型中出现的附加部分；同时该图也假设应用层要传递的消息长度小于底层网络数据包的最大长度。否则的话，消息就要被封装成网络层的几个数据包。在接收方，下层接收到的数据项要做一次相反的转换，再传递到上一层。上层协议的种类已经包含在了每层的头部，这使得接收方的协议栈能选择正确的软件组件来拆分数据包。



76
|
77

图3-3 封装在分层协议中的应用

这样，每一层为上一层提供服务，并扩展下一层提供的服务。最下面的是物理层。它是由通信介质（铜线、光纤电缆、卫星通信信道或无线电传输）和在发送端放置信号以及接收端感应信号的模拟信号电路实现的。在接收端，接收到数据项后再一层层通过软件模块层，每一层都重新转换直到变成可传递给接收进程的格式。

协议组 一套完整的协议层次被称为协议组或者协议栈，这也反映了分层结构。图3-4显示了与国际标准组织（ISO）采用的开放系统互连（OSI）7层参考模型[ISO 1992]相一致的协议栈。采用OSI参考模型，是为了促进满足开放系统需求的协议标准的开发。

图3-5总结了OSI参考模型每一层的目的。正如名字所暗示的，这只是一个协议定义的框架，而不是很明确的定义。与OSI模型一致的协议组必须在模型定义的7层的每一层包括至少一个特定的协议。

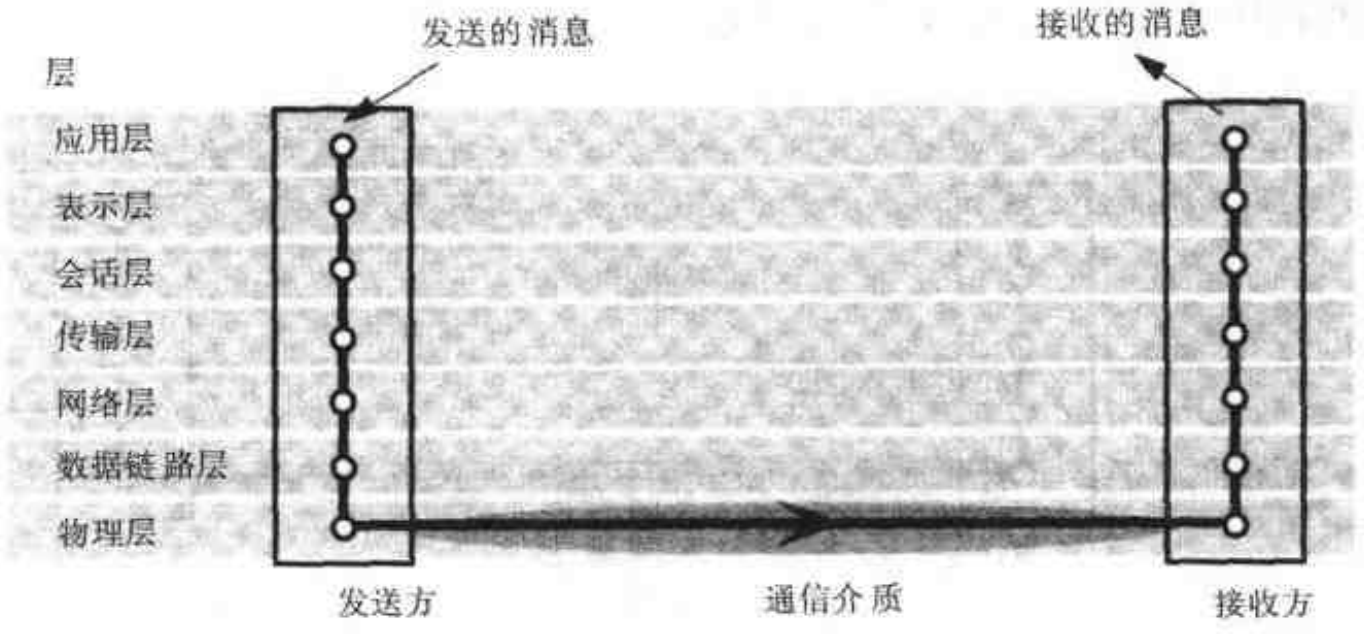


图3-4 ISO开放系统互连（OSI）协议模型中的协议层

协议分层给简化和概括访问网络通信服务的软件接口带来了实质性的好处，同时也带来了较大的性能开销。通过*N*层协议栈传输一个应用层的消息，通常在协议组中要进行*N*次控制传递，才能到达相关的软件层，协议组中至少有一个是操作系统的入口，数据的*N*份拷贝，也作为了封装机制的一部分。所有这些开销导致应用进程间的数据传输率远远低于可用的网络带宽。

层	描　述	例　子
应用层	这层协议是为满足特定应用的通信要求而设计的，通常定义一个服务的接口	HTTP、FTP、SMTP、CORBA IIOP
表示层	这层协议将以一种网络表示传输数据，这种表示与计算机使用的表示无关的。如果需要，可以在这一层对数据进行加密	安全套接字（SSL）、CORBA数据表示
会话层	该层要实现可靠性和适应性，比如：故障检测和自动恢复	
传输层	这是处理消息（大于数据包）的最低层。消息被定位到进程上的通信端口。这层的协议可以是面向连接的，也可以是无连接的	TCP、UDP
网络层	在指定网络中，在计算机间传输数据数据包，在一个WAN或是一个互联网，这一层负责生成一个通过路由器的路径。在单一的LAN中不需要路由	IP、ATM虚电路
数据链路层	负责在有直接物理连接的结点间传输数据包。在WAN中，传输是在路由器间或路由器和主机间进行的。在LAN中，传输是在任意一对的主机间进行	Ethernet MAC、ATM信元传送、PPP
物理层	指驱动网络的电路和硬件。它用模拟信号传输二进制数据序列：用电信号振幅或频率的调制信号（在电缆电路上），用光信号（在光纤电路上），或其他电磁信号（在无线电和微波电路上）	Ethernet基带信号、ISDN

图3-5 OSI协议小结

图3-5包括了在因特网中使用的协议的例子，但因特网的实现有两方面和OSI模型不一样。第一，因特网协议栈中，并没有清楚地区分应用层、表示层、会话层。在因特网协议栈中应用层和表示层或实现成单独的中间件层或在每个应用内部分开实现。这样CORBA就可以在每个应用进程包括的中间件库中实现对象间调用和数据表示（CORBA的进一步讨论见第17章）。

Web浏览器和其他的一些需要安全信道的程序也是用相似的方法，把安全套接字层（见第7章）用作它的程序库。

第二、会话层与传输层集成在一起。互联网络协议组包括应用层、传输层和互联网络层。互联网络层是一个“虚拟的”网络层，负责将互联网络的数据包传输到目的地址。互联网络数据包是在互联网络上传递的数据单元。

互联网络协议覆盖在底层的网络上，如图3-6所示。网络接口层接收互联网络层的数据包，并将其转换成适合每个底层网络的网络层传输的数据包。

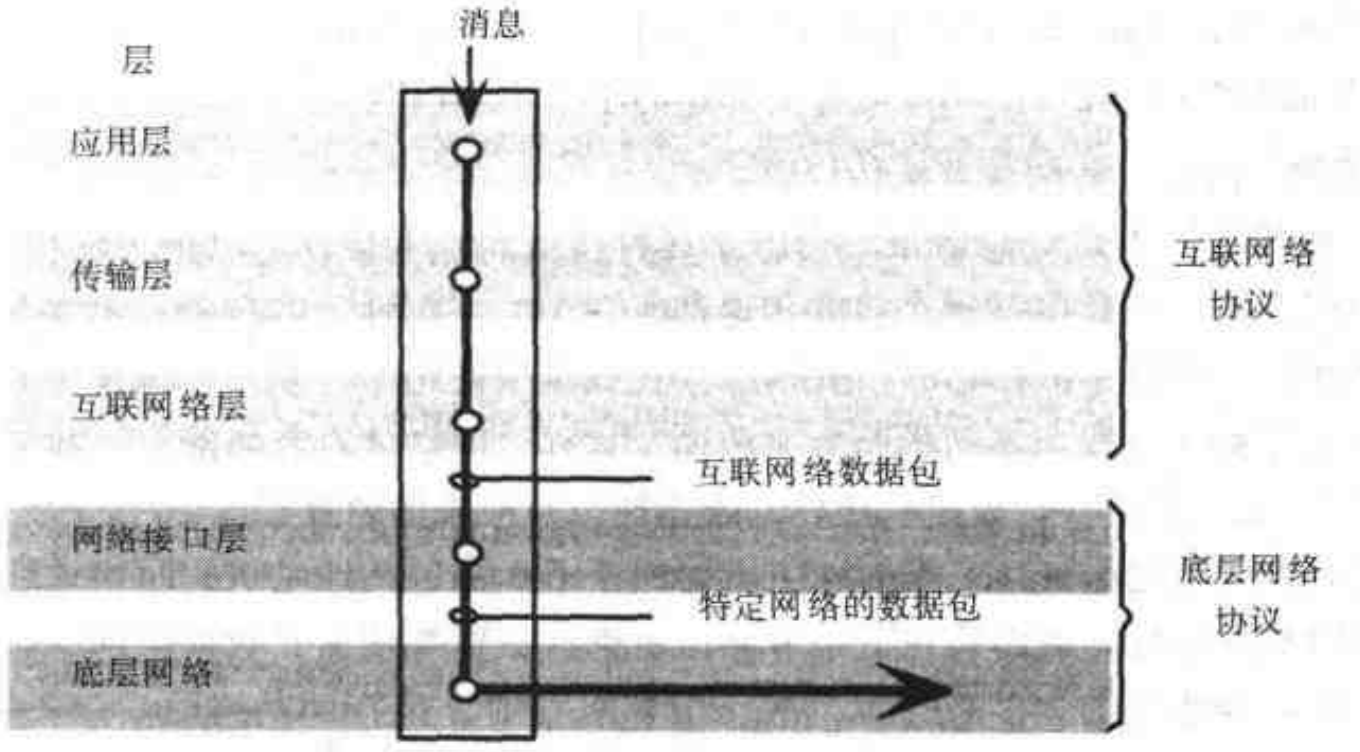


图3-6 互联网络层

数据包装配 传输前将消息分割成多个数据包以及在接收端重新组装各个数据包，通常都是由传输层完成。

网络层协议的数据包包括头部和数据域。在大部分网络技术中，数据域是变长的，但在一个称为最大传输单元（MTU）的范围内。如果消息的长度超过了底层网络层的MTU，就将其裁成多个适当大小的块，并标上顺序号以便其重新组装，再用多个数据包进行传输。例如，以太网的MTU是1500字节，不超过这个数据量，就能在一个以太网数据包中被传输。

尽管在因特网协议组中IP协议处于网络层协议的位置，但它的MTU却很大，有64KB（实际经常使用8KB，因为有些结点无法处理这么大的数据包）。无论IP数据包采用哪一个MTU值，比以太网MTU值大的数据包必须分割后，才能在以太网上传输。

端口 传输层的任务是在一对网络端口间提供网络独立的消息传送服务。端口是主机中可由软件定义的通信终点。它隶属于进程，可以让进程成对地通信。可以改变端口抽象的特定细节以便提供额外有用的性质。这里将详细讲述端口在因特网和大部分其他网络中的寻址过程。第4章讨论端口的使用。

寻址 传输层负责将消息传递到目的地址，其使用的传输地址由主机的网络地址和一个端口号组成。网络地址是惟一能标识主机的一个数字标识，可以让负责将数据路由到该主机的结点准确地定位它。在因特网中，每台主机都分配了一个IP地址，用于标识该主机和相应它连入的子网，使得从任何其他结点，如下面的节将描述的一样，都能路由到该主机。以太网中没有路由结点，每台主机都有责任辨识数据包的地址，并接收发给自己的数据包。

因特网中，每台计算机通常有几个众所周知的端口号，用作指定的因特网服务，如HTTP或FTP。一些常见的端口号以及服务定义都在权威机构登记了[www.iana.org]。如要访问指定主机的某个服务，只要将请求发给该主机相关的端口号就可以了。有些服务，例如FTP，会分配一个新的端口号（私有号码），并将这新的端口号发给客户端。客户端使用新的端口号建立一个TCP连接，用于完成交易或会话的剩余部分。其他服务，如HTTP，是通过共知的端口号处理所有的业务活动。

78 ~ 80

大于1023的端口号可供给新的服务和客户端进程，如客户端进程为了从服务器接收消息，需要分配新的端口。分布式系统通常有多个服务器，因时间和所属组织的不同而不同。确切的说，对这些服务，分配固定的主机和固定的端口是不太可行的。在分布式系统中如何为客户端定位服务器的解决办法将在第5章有关绑定的小节中加以讨论。

数据包传递 网络层有两种方法传递数据包：

- *数据报包传递* 术语“数据报”指出了这种传输模式和信件、电报传输模式的相似性。数据报网络的本质特征是每个包的传递都是一个“一次性”的过程；不需要计划，一旦包被传递后，网络不保存它的相关信息。在数据报网络中，从一个源地址到一个目的地的数据包序列可以按照不同的路由来传递（这样，网络就有容错能力，或缓解局部拥塞的影响），如果是这样，数据包序列可能不能按照原来的顺序到达。

  每个数据报包都包括完整的源主机的网络地址和目的地主机的网络地址，后者是路由过程的基本参数，我们将在下一节加以讨论。数据报传递是数据包网络根本的基础性概念，在当今使用的大多数计算机网络中都能找到。因特网网络层——IP——以太网以及大部分有线或无线的局域网技术都是基于数据报传递。

- *虚电路包传递* 一些网络层的服务，在实现包传输时，使用的是类似电话网络的方法。虚电路必须在能经源主机A到目的主机B传递包之前建立。建立虚电路，需要先识别出从源地址到目标地址的路由，这可能会经过一些中介结点。在沿着路径的每个结点上，都会有一表格项，指示路由的下一步该使用哪个连接。

  虚电路一旦建立起来，就可以传输任意数量的数据包了。每个网络层的数据包只包括一个虚电路号，而不是源地址和目的地地址。地址信息不需要了，因为在中介结点，数据包靠虚电路号就可以被路由了。数据包到了目的地后，从虚电路号就可以决定其源地址。

  与电话网络的类比并不能这样逐字逐句地看。在POTS中，一个电话呼叫就要建立从呼叫者到接话者的物理电路，而这一音频连接也将作为专用连接而被保留。在虚电路的数据包传递中，电路只是由一些在路由结点上的表格项来表示，而数据包所路经的连接也只在传递一个数据包的时候使用，在其余时间这些连接是空闲的，可供它用。因此一个单一的连接可以被多个独立的虚电路使用。如今使用的最重要的虚电路网络技术是ATM，我们已经提到过（在3.3.3节）它传送单个数据包的等待时间较短，这是因为它直接使用虚电路。但无论怎么说，数据包传送到一个新地址前要求有一个准备阶段确实造成了一个短的延迟。

81

数据报传递和虚电路传递之间的区别，请不要和传输层中两个相似的机制混淆——无连接传输和面向连接传输。我们将在3.4.6节有关因特网传输协议——UDP（无连接的）和TCP（面向连接的）——的内容中描述这些内容。这里我们只是让大家注意，在任何一种类型的网络层上都可以实现这些传输模式。

3.3.5 路由

路由是在除了那些局域网，比如以太网（局域网在所有相接的主机间两两都有直接连接）以外，其他所有的网络中都需要的功能。在大型网络中，采用的是自适应路由：网络两点间的最佳路由会被周期性地重新评估，评估时会考虑到当时的网络流量以及故障情况（如路由器或网络断连）。

在图3-7所示的网络中将数据包传递到目的地址是处于连接点的众多路由器的责任。除非源主机和目的主机都在同一个局域网中，不然数据包都必须经过一个或多个的路由结点，辗转多次才能到达。而决定数据包传输到目的地的路由是由路由算法负责的——它由每个结点中的一个网络层程序实现。

路由算法包括两个部分：

1. 它必须决定每个数据包在穿梭于网络时所应经过的路径。在电路交换网络层（如X.25）和帧中继网络（如ATM）中，虚电路或连接建立以后，整个路径也已经决定了。在包交换网络层（如IP），数据包的路径是单独决定的。如果不降低网络性能，算法必须特别简单有效。

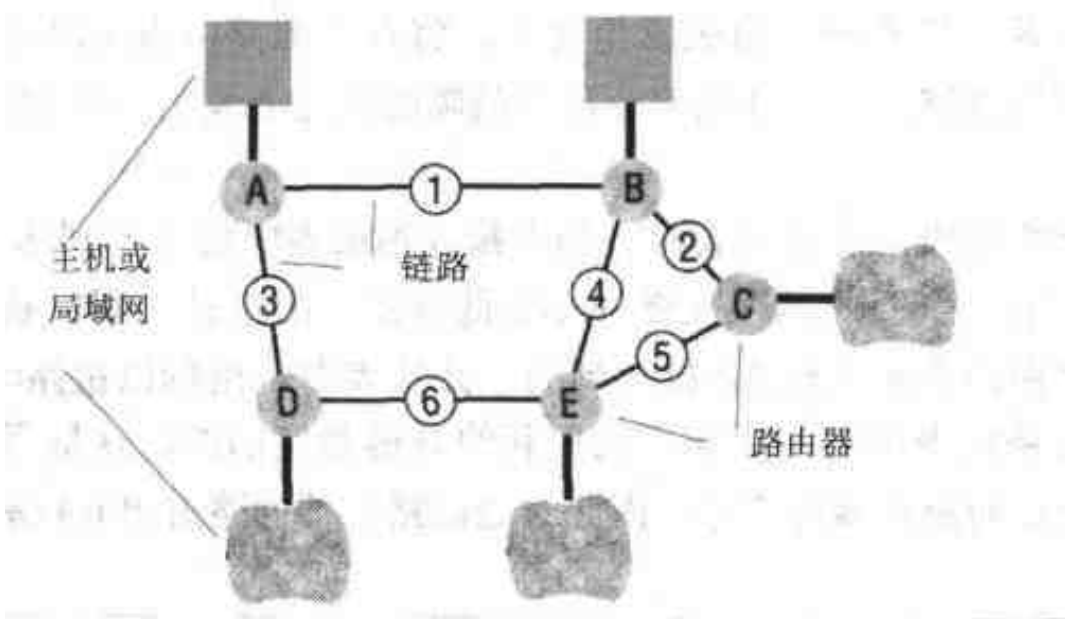


图3-7 广域网中的路由

2. 它必须能够根据监控流量和检测配置变化或故障动态地评估整个网络。在这种活动中时间并不是至关重要的：可以使用慢一些、计算量较大的技术。

这两个活动分布在整个网络中。路由是一段一段决定的，它用本地拥有的信息决定每个数据包下一步的方向。本地拥有的信息由一个分发链路状态信息（它们的负载和故障状态）的算法定期地更新。

现在开始讨论一种相对简单的路由算法。这将为3.4.3节中对链路-状态算法的讨论提供一个基础，从1979以来链路-状态算法就成为了因特网上的主要路由算法。我们在此描述的是一种“距离向量”算法。网络中的路由是在图中寻找路径问题的一个实例。Bellman的最短路径算法[Bellman 1957]早在计算机网络得到发展之前就发表了，为距离向量法提供了基础。Bellman的方法已被Ford和Fulkerson[1962]改写成一个适合大型网络实现的分布式算法，而基于他们这些工作成果的协议常常被称为“Bellman-Ford”协议。

图3-8给出了图3-7的网络中每个路由器中保存的路由表，其中假设网络中没有出故障的链路和路由器。每行给到特定地址的数据包提供了路由信息。链路域给到指定目的地的数据包指明了下一个段。开销域计算了向量距离，或是到达目的地的转折次数。对于各段带宽相似

的存储转发网络，这张表给出了一个数据包在网络中传输时间的合理估计。存储在路由表中的开销域的信息并不是在数据包路由，即算法的第1部分时使用的，而是在算法的第2部分用来建立和维护路由表的。

路由表中为每个可能的目的地都准备了一项，给出了数据包为了到达目的地而要走的下一跳。当数据包到达一个路由器时，目的地地址就会被抽取并在路由表中查找。路由表中的表项给出了指引数据包走向它的目的地的下一个链路。

例如，一个目的地为C的数据包发送到了路由器A，路由器从路由表中检查有关C的项。路由表表明数据包应该从A路由到标号为1的链路。数据包到达B后，按照和前面一样的过程，在B的路由表中查询，发现往C的路径需要经过标号为2的链路。到数据包到达C时，路由表中的相关项是“本地”，而不再是一个链路号。这表明应该数据包发到本地主机上去。

现在让我们来考虑一下路由表是怎样建立以及在网络上发生错误时又是怎样维护的。即上面所说的路由算法的第2部分是怎样完成的。因为每个路由表只指定下面一步，所以可以以分布的方式进行路由信息的建立或修正。每个路由器通过路由器信息协议（RIP）和每个邻接结点发送自己路由表信息的概要，相互交换网络信息。下面概括地描述一下路由器所完成的RIP动作：

1. 周期性地以及一旦本地路由表发生改变，将自己的路由表（以概要的方式）发给邻接的所有可访问的路由器。即在每个没有故障的链路上发出一个包含路由表副本的RIP数据包。
2. 当从邻接路由器收到这样的表时，如果接收到的表中给出了到达一个新目的地的路由，或对于已有的一个目的地存在更好（更低开销）的路径，则用新的路由更新本地的路由表。如果路由表是从链路n接收到的，并且表中给出的以链路n开头的到达某地点的开销和本地路由表中的不相同，则用新的开销替换旧的。这是因为，新表是从和相关的目的地更近的路由器传来的，因此对经该路由器的路由更有权威性。

路由：从A

到	链路	开销
A	本地	0
B	1	1
C	1	2
D	3	1
E	1	2

路由：从B

到	链路	开销
A	1	1
B	本地	0
C	2	1
D	1	2
E	4	1

路由：从C

到	链路	开销
A	2	2
B	2	1
C	本地	0
D	5	2
E	5	1

路由：从D

到	链路	开销
A	3	1
B	3	2
C	6	2
D	本地	0
E	6	1

路由：从E

到	链路	开销
A	4	2
B	4	1
C	5	1
D	6	1
E	本地	0

图3-8 图3-7所示网络中的路由表

图3-9给出的伪码程序可以更加准确地描述这个算法，其中*Tr*是从另一个路由器收到的表，*Tl*是本地路由表。Ford和Fulkerson[1962]已经证明，无论何时网络发生变化，上面描述的步骤能充分确保路由表都汇聚了到每个地址的最佳路径。即使网络没有发生变化，也会以频率*t*来传播路由表，这样做是为了确保其稳定性，例如，要在丢失RIP数据包的情况下保证稳定性。因特网采用的*t*值是30s。

82
t
84

为了处理故障，每个路由器都检测自己的链路并做以下的工作：

当检测到一条有故障的链路时，将本地路由表中指向故障链路的所有项的开销都设为∞，接着执行*Send*动作。

这样，一个链路断开的信息被表示成通往相关地址的开销值是无穷大。当这一信息传递到邻接路由器时，它们的路由表也将通过*Receive*动作进行更新（注意：∞+1=∞）。这一信息还会进一步传播，直到到达了有可通行路径到达该相关地址的结点（如果存在这样的结点）。最终，该结点会传播它的路由表，它的可通行的路由也将代替所有结点中的出故障的路由。

```
Send: 每隔t秒或Tl发生了变化，在每个没有故障的链路上发送Tl。

Receive: 每当从链路n接收到路由表Tr：
 for all rows Rr in Tr {
    if (Rr.link ≠ n) {
       Rr.cost = Rr.cost + 1;
       Rr.link = n;
       if ( Rr.destination 不在 Tl中 ) 将Rr加入到Tl; //向Tl中加入新的目的地
       else for Tl中的所有行Rl {
          if (Rr.destination = Rl.destination and
             (Rr.cost < Rl.cost or Rl.link = n)) Rl = Rr;
             // Rr.cost < Rl.cost : 远程结点有更好的路由
             // Rl.link = n : 远程结点更加权威
       }
    }
 }
```

图3-9 RIP路由算法

距离－向量算法可以用多种方法进行改进：开销（也被称为度量）可以根据实际链路的带宽来计算；算法可以加以修改，以增加信息收敛的速度，避免那些不需要的中间状态，比如循环，它一般是在信息汇聚完成前发生的。具有这些改进的路由信息协议是第一个在因特网中使用的路由协议，也就是众所周知的RIP-1，其具体描述见RFC 1058[Hedrick 1988]。但并没有很好地解决收敛速度过慢所带来的问题，这导致了当网络处于中间状态时路由低效和数据包丢失的问题。

后来路由算法的发展趋于在每个网络结点中增加对于网络的信息容量。这一类算法中最重要的一族是被人们称为链路－状态算法。它们的基本思想是分布并更新在每个结点中一个表示网络所有部分或重要部分的数据库。每个结点负责计算在自己的数据库中，到达目的地所要走的最佳路径。这种计算可由多种算法完成，有些算法就避免了在Bellman-Fords算法中存在的问题，如收敛的时间慢和不希望出现的中间状态。路由算法的设计是一个相当重要的话题，我们这里的讨论是非常有限的。对于因特网中更深入的路由问题，参阅Huitema[1995]，

85

如想综合地了解有关路由算法的话，参见Tanenbaum[1996]。

3.3.6 拥塞控制

网络的能力受限于通信链路和交换结点两者性能。当任何链路或结点的负载接近其负载能力时，主机中就会建立等待发送的数据包队列，中间结点也会堆满被其他数据传输所阻塞了的数据包。如果负载继续维持在这样的高水平，那么等待发送的队列就会不断增长，直到到达可用的缓冲区空间的限制。

一旦结点到达这样的状态，结点只能将未来到达的数据包都丢弃掉。前面我们已经提到，在网络层偶尔的数据包丢失现象是允许的，这可以通过从更高层重传丢失的数据包来弥补。而当数据包丢失率和重传到达一个很高的状态时，这对网络的吞吐量来说，其后果是灾难性的。这个道理很简单：数据包在中间结点丢失的话，已经被占用的网络资源就被浪费掉了，而重传还要再消耗同样多的资源。根据经验，当网络的负载超过其能力的80%以后，系统整个吞吐量会因为数据包丢失而下降，除非控制高负载链路的使用。

为了避免数据包在网络传递时经过拥塞结点而被丢掉的情况，最好将数据包存在未发生拥塞的结点中直到拥塞减少。这将增加数据包的延迟，但不会严重降低整个网络的吞吐量。用于实现该目的的技术称为拥塞控制。

通常，拥塞控制是通过通知发生拥塞的路径上的结点而实现的。因此数据包传输率会有所减少。对中间结点来说，这就意味着进入的数据包将会缓冲很长时间。而作为发出数据包的源主机，结果就是把要发送的数据包在主机中排队，或者阻止产生这些数据包的应用程序，直到网络能妥善地处理它们为止。

所有基于数据报的网络层，包括IP和以太网，都依靠端－端的流量控制。也就是，发送结点必须基于收到的接收方信息减少其发出数据包的速率。为发送结点提供拥塞信息，可以通过显式地传输一个请求减少传输率的特殊消息（被称为阻塞数据包）来实现；或通过一个特别的传输控制协议（TCP的名字也由此而来——3.4.6节将解释TCP中的机制）实现；或通过观察数据包丢失的发生情况（假设协议是要确认每一个数据包的）来实现。

在一些基于虚电路的网络中，拥塞信息可以在每个结点接收到并作用于每个结点。尽管ATM用的是虚电路传递，它仍要依靠服务质量管理（见3.5.3节和第15章）来保证每个电路都能完成所要求的流量。

86

3.3.7 网络互联

不同的网络、链路和物理层协议形成不同的网络技术。本地网络是基于以太网和ATM的技术建立起来的，而广域网络是建立在各种数字和模拟电话网络、卫星连接和广域ATM网络上的。个人的计算机和本地网络则通过调制解调器、ISDN链路、DSL连接接入因特网或企业内部网。

建立一个集成的网络（*互联网*），必须集成许多子网，而各子网基于上述网络技术。为了实现集成，需要如下规定：

1. 统一的互联网寻址机制，使得数据包可以找到连在任一子网中的任一主机。
2. 定义互联网中的数据包格式并给出相应处理规则的协议。
3. 互联组件，用于按照互联网地址将数据包路由到目的地，可用多种网络技术通过子网传递数据包。

对于因特网而言，IP地址提供了（1），（2）是IP协议，（3）由称为因特网路由器的组件实现。IP协议和IP寻址将在3.4节做详细地描述。这里我们将讨论有关因特网路由器和其他用来连接各网络的组件的功能。

图3-10展示了伦敦大学玛丽女王与韦斯特菲尔德学院（QMW）企业内部网的一小部分。部分细节将在后续章节中加以解释。这里我们要注意的是图中包含由路由器互连的多个子网那一部分。子网被标成灰色；一共有5个，其中3个共享了IP网络138.37.95（使用了无等级的域间路由机制，见3.4.3节）。数字表示的地址是IP地址，下面将加以解释。路由器在多个子网的内部，它们在每个子网都有一个IP地址（地址就写在链路上）。

路由器（主机名：hammer和sickle）实际上是一个通用的计算机，也能完成其他任务。其中一个路由器作为防火墙；防火墙的角色是和路由功能紧密相关的，我们将在下面讨论这点。138.37.95.232/29子网在IP层并没有和网络中的其他部分相连。只有文件服务器custard可以访问它，该服务器通过一个检测和控制相连打印机使用的服务器进程提供打印服务。

图3-10中所有的链路都是以太网。大部分的带宽是100Mbps，但有一个的带宽是1000Mbps，这是因为它支持着大量学生用的计算机和custard间的巨大数据流量，文件服务器包含了他们所有的文件。

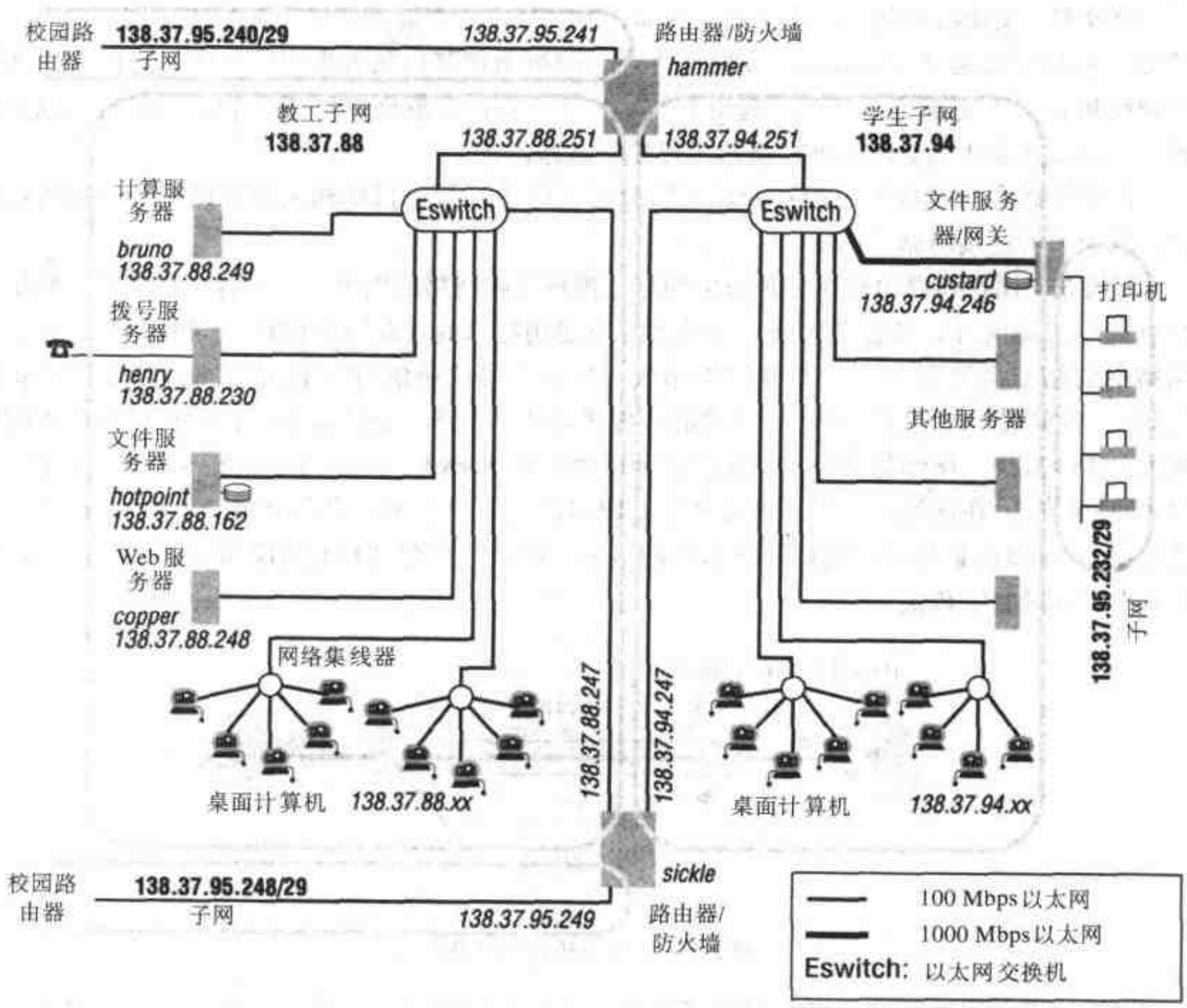


图3-10 QMW计算机科学网的简图

在图示的这部分网络中有两个以太网交换机和几个以太网网络集线器。两者对IP数据包来说都是透明的。以太网网络集线器简单地将多个以太网电缆的段连接在一起，在网络协议层，这些段形成了一个以太网。所有收到的以太网数据包将转播到所有的段。以太网交换机连接了几个以太网，用于将进入的数据包路由到目的主机所在的以太网中。

路由器 我们已经提到，除以太网和无线网络这样的网络以外——这些网络中的主机由一个传输介质所连接——其他所有网络都需要路由。图3-7显示了一个由6个链路连接5个路由器组成的网络。在一个互联网中，可直接连接到路由器，如图3-7所示，也可以通过子网连接路由器，如图3-10中的custard。在这两个例子中，每个路由器都负责将从任一连接来的互联网络数据包准确地发到下一条连接，为此路由器维护着路由表。

87
to
88

网桥 网桥连接不同种类的网络。一些网桥连接几个网络，它们也被称为网桥/路由器，这是因为它们也表现出了路由的功能。例如，QMW学院的校园网包括一个光纤分布式数据接口FDDI主干（没有在图3-10中显示），它就是由网桥/路由器连接到图中的以太网子网中。

网络集线器 网络集线器是将主机和以太网以及其他广播本地网技术的扩展段连接起来的一种方便的手段。它有多个插槽（通常有4~64个），每一个插槽都可以连接一台计算机。它们也被用于克服单个段带来的距离上的限制，提供添加额外主机的途径。

交换机 交换机的功能有点类似路由器，但这只限于本地网络（一般是以太网）之内。也就是，它们互连起多个分离的以太网，将到达的数据包路由到适当的网络中。它们在以太网的网络协议层上完成这一任务。起初它们对整个广阔的互联网络一无所知，逐渐通过观察数据流量以及在缺少信息时采取广播请求等建立其路由表。

在集线器上交换数据包的好处是它们分离了到达流量，仅在相关的外出网络上传输数据包，减少了所连接网络的拥塞。

隧道法 网桥和路由器在多种底层网络上传输互联网络数据包，不过有一种情形，底层网络协议在传输时可以被隐藏起来，并且不需要使用特殊的互联网络协议。当两个连在分离的网络中的结点需要用另一种类型的网络或“异型”协议通信时，它们之间可通过构造协议“隧道”来实现。图3-11显示的是隧道的一种建议使用方法，它使得因特网能迁移到最近刚被通过的IPv6协议。IPv6将会取代现在使用的IP协议版本IPv4，但它们并不兼容（IPv4，IPv6的描述见3.4节）。在逐渐转成IPv6的过程中，IPv4的海洋中会不断出现IPv6“岛屿”。在我们的图例中，A和B就是这样的岛屿。在岛屿的边界，IPv6数据包被封装成IPv4的格式，并以这种方式在IPv4网络中传输。

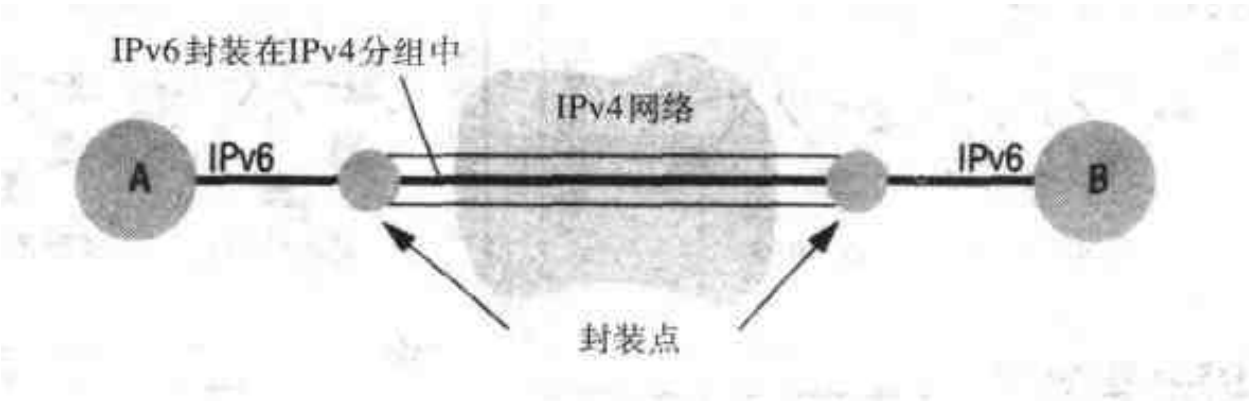


图3-11 向IPv6转移的隧道法

协议隧道其实就是在异型网络环境中传输数据包的软件层。看一个例子，移动IP协议通过从它们的本地基站到任一网络位置建立起一条隧道，来传递IP数据包。中间的网络结点不

需要修改以适应移动IP协议。IP组播协议在处理方式上很相似，依靠一些支持IP组播的路由器来决定路由，但用使用标准IP地址的路由器来传输IP数据包。另一个例子是在串行连接上传输IP数据包的PPP协议。 89

下面类比解释了选择术语隧道一词的原因，同时也提供了另一种方法来思考隧道的含义。穿山隧道使得车辆的经过成为了可能，如果没有隧道这是不可能的。路是连续的——隧道对于应用（车辆）来说是透明的。路是运输机制，而隧道使得它能在相异的环境中工作。

3.4 因特网协议

本节介绍TCP/IP协议组的主要特点，并讨论其在分布式系统中使用的好处及局限性。

因特网的研究始于20世纪70年代早期的ARPANET——第一个大规模计算机网络的开发[Leiner *et al.* 1997]，随着近20年的研究和开发，因特网渐渐成形。这项研究的一个重要部分是开发TCP/IP协议组，TCP是指传输控制协议，IP是指网际协议。TCP/IP和因特网应用协议在美国研究网络和最近许多国家中越来越多的商业网络中的广泛使用，使得全国的网络可以集成成一个互联网，它已经迅速发展到目前超过6000万的主机数量。许多应用服务和应用层的协议（列在下面的各个括号内）现在都是基于TCP/IP的，包括Web（HTTP）、电子邮件（SMTP，POP）、网络新闻（NNTP）、文件传输（FTP）、远程登录（Telnet）。TCP是一个传输协议，它可以直接支持应用程序，也可以将附加的协议加在它上面，以完成额外的特色。例如，通常HTTP传输时直接使用TCP，但当需要端-端安全性时，安全套接字层（SSL）协议（在7.6.3节讨论）在TCP的上层建立安全信道，HTTP消息通过这一安全信道传输。

因特网协议原本的目的主要是用来支持一些简单的广域应用，如文件传输和电子邮件，包括在地理上相隔很远的会有较高延迟的通信。但这些协议已被证明足以有效支持许多分布式应用的需求，不论这些应用是在广域网上还是在本地网上，它们现在几乎广泛地在分布式系统中使用。通信协议的标准化带来了巨大的好处。在广泛采用TCP/IP通信的同时，也有一些重要的例外情况：

- 便携式设备上无线应用使用的WAP协议。
- 支持多媒体流应用的特殊协议。

图3-6的互联网协议层被认成图3-12中因特网的特例。有两个传输协议——TCP（传输控制协议）和UDP（用户数据报协议）。TCP是一个面向连接的可靠协议，而UDP是一个不能保证可靠传输的数据报协议。因特网协议（IP）是因特网虚拟网络的底层“网络”协议——IP数据报为因特网和其他的TCP/IP网络提供了基本的传输机制。我们在前面的句子中给“网络”一词加上引号，因为它并不是惟一的实现因特网通信中的那个网络层。这是因为因特网协议通常是在另一个网络技术之上，比如说以太网，它已经提供了一个网络层，使得连接在同一网络中的计算机可以交换数据报。图3-13说明了通过TCP在底层以太网上传输消息的时候，数据包的封装过程。头部的标签是上层协议的类型，是为了能让接收协议栈正确地解开这个数据包。在TCP层，接收方的端口号起到类似的用途，使得接收主机的TCP软件组件可以将该消息送到特定的应用层进程中去。

TCP/IP的规范[Postel 1981a；1981b]没有详细描述因特网数据报层以下的层——因特网层的IP数据包会被转换成可以在几乎任何底层网络或数据链路上传输的包。

举例来说，IP起初运行在ARPANET上，这个网络包括主机和一些早期版本的由长距离数

据链路连接的路由器（称为PSE）。如今IP实际上几乎已经在每一个网络技术上使用了，包括ATM、本地网络如以太网和令牌网，在串行线和电话电路上通过PPP协议[Parker 1992]实现了IP，这使得IP可用调制解调器和其他串行链路来通信。

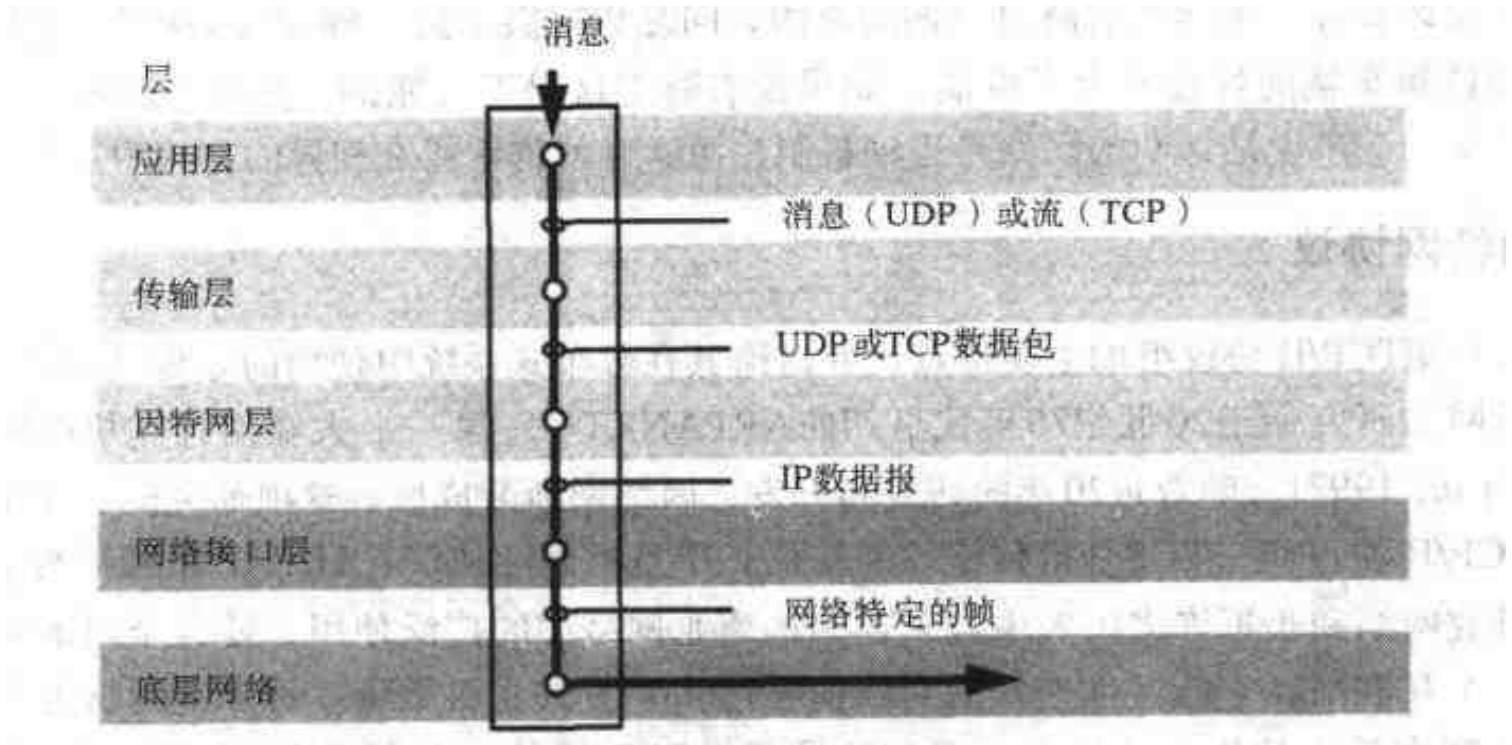


图3-12 TCP/IP层

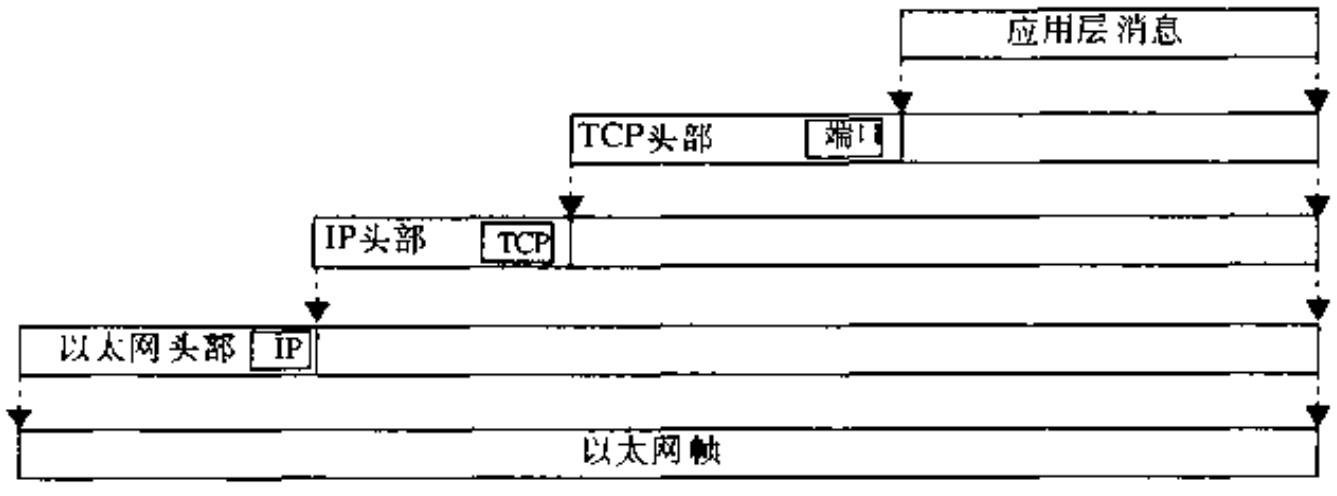


图3-13 通过TCP在以太网上传输消息时发生的封装

TCP/IP的成功在于它独立于底层传输技术，这使得互联网可以由许多异构的网络或数据链路建立起来。用户和应用程序感知到的是一个支持TCP和UDP的虚拟网络，TCP和UDP的实现者看到一个虚拟IP网络，它隐藏了底层传输媒介的多样性。图3-14说明了这一点。

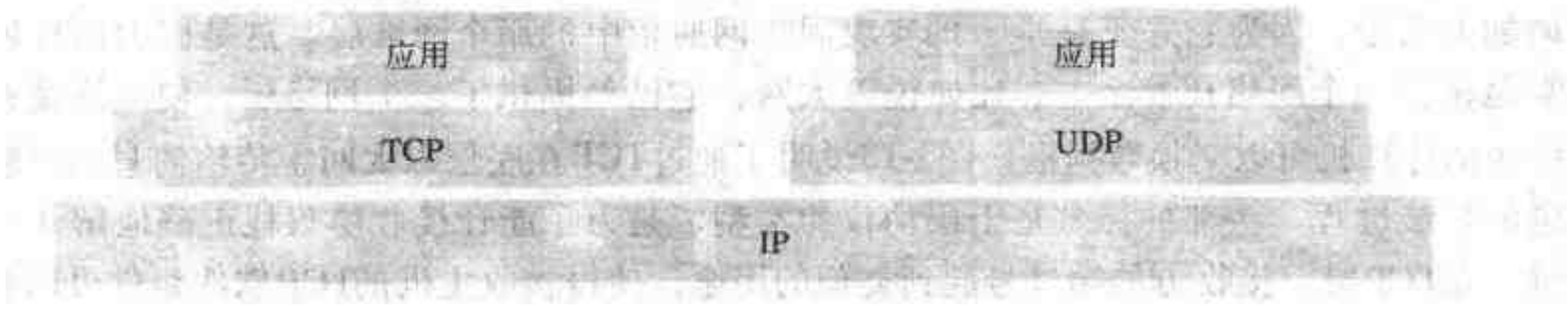


图3-14 编程者眼中TCP/IP因特网的概念

下面两节将详细描述IP寻址机制和IP协议。域名系统，用于将因特网用户很熟悉的*www.amazon.com*、*hpl.hp.com*、*stanford.edu*、*qmw.ac.uk*这些域名转化成IP地址，将在3.4.7节中介绍，第9章将有更全面的叙述。

现在整个因特网使用的IP协议的版本是IPv4（从1984年1月开始），这也是我们将在下面两个小节里讨论的版本。但由于因特网的飞速发展，人们也不得不出版新的IP版本（IPv6），以克服IPv4中地址数量的限制并为之增添一些新的功能需求。我们将在3.4.4节描述IPv6。由于大量的软件将受此影响，逐渐转成IPv6的过程计划在10年或更长的时间里来完成。

3.4.1 IP寻址

或许设计因特网协议最有挑战的方面是构造主机的命名、寻址机制以及将IP数据包路由到目的地的机制。分配主机网络地址的机制和计算机连接到它们的机制需要满足以下几点：

- 这必须是通用的——任何主机必须都可以发送数据包给因特网中的任何其他主机。
- 在使用地址空间方面，必须是有效的——预知因特网的最终规模、网络数量和所要求的主机地址数量是不可能的。地址空间必须仔细地分割以确保地址不会用完。1978～1982年，开发TCP/IP协议时，提供232或约40亿（大致等于当时世界的人口总数）的可寻址的主机被认为是足够了。这种判断已经被证明是目光短浅的，因为下列两个因素：
  - 因特网的增长率远远超过了任何预测；
  - 地址空间的分配和使用比预期的效率要低得多。
- 寻址机制必须有助于开发灵活有效的路由机制，但地址本身并不能包括太多将数据包路由到目标地的信息。

90～92

所选的方案为因特网中的每个主机都分配一个IP地址——一个32位的数字标识，其中包括了一个网络标识（惟一标识了因特网中的某个子网）、一个主机标识（惟一标识了连在该网络中的主机连接）。就是这些地址放在IP数据包中并被用于将其路由到目的地。

因特网地址空间所采用的设计如图3-15所示。一共有4类因特网地址——A类、B类、C类、D类。D类地址保留给因特网组播通信，组播通信仅在某些因特网路由器中实现，进一步地讨论见4.5.1节。E类地址包括一些未分配的地址，留待未来的需求。

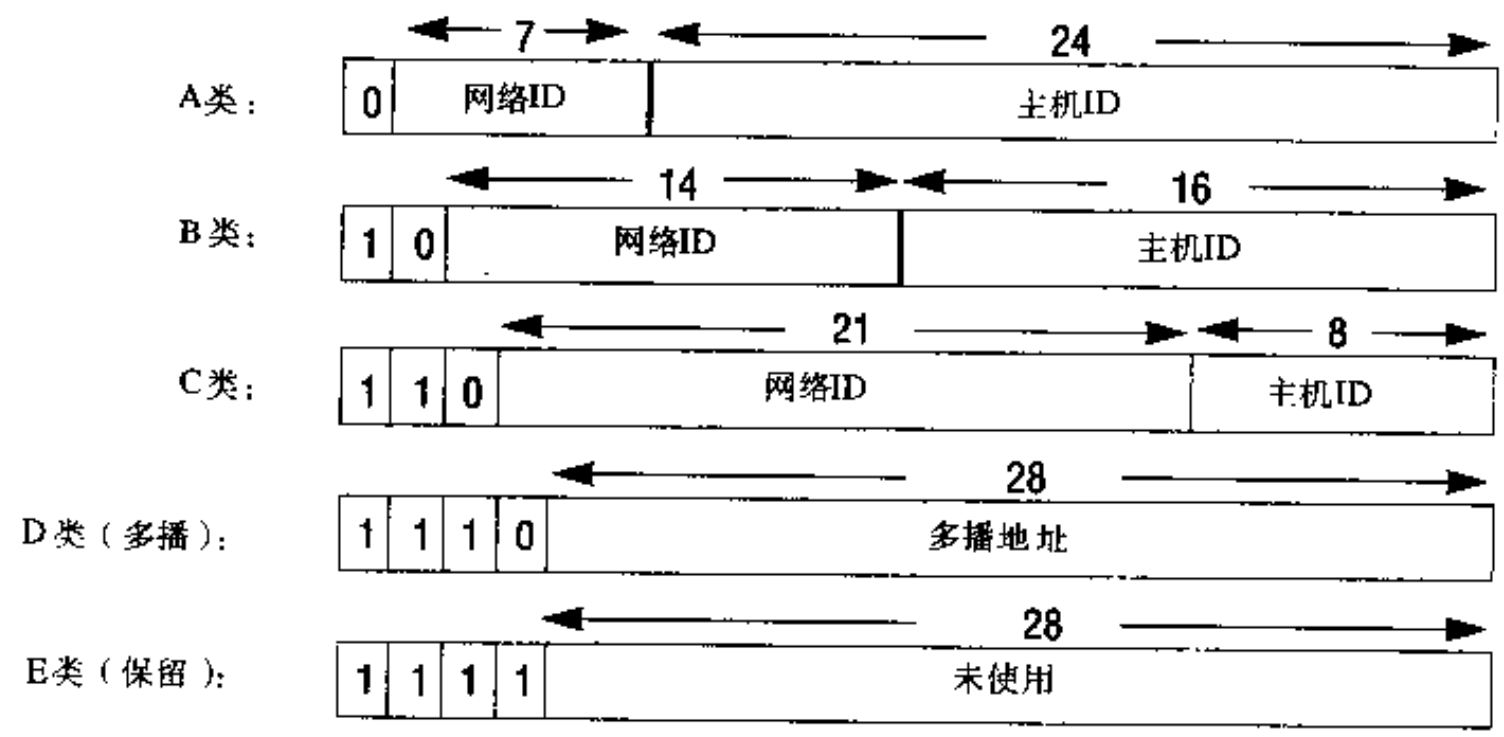


图3-15 因特网地址结构，域大小的单位是位

这些包含网络标识和主机标识的32位因特网地址通常写成由点分开的4个数字序列。每个数字表示一个字节或IP地址的8位位组。每一类网络地址的允许值如图3-16所示。

图3-16 十进制的因特网地址

三类地址的设计都是为了能满足不同类型组织的需要。A类地址，在每个子网中有2^{24}的主机容量，是为非常大的网络准备的，比如US NSFNet和其他全国性的广域网络。B类地址是分配给很可能超过255台计算机的网络，而C类地址则是分配给所有其他的网络。

因特网地址中主机标识为0和其他都是1（二进制）的地址留作特殊用途。地址中主机标识设为0的用来代表“本机”，若一个主机标识都是1，则表示这是一个广播消息，按照地址的网络标识部分所指定的网络，发到每个和该网络连接的主机上。

网络标识是由因特网网络信息中心（NIC），分配给连上因特网的组织。连入因特网的计算机的主机标识是由相关网络的管理员来分配的。

既然主机的地址包括一个网络标识，那么连在不止一个网络中的计算机必须在每个网络中都有一个独立的地址。每次计算机移到一个新的网络，它的因特网地址也必须改变。这些需求导致了实质性的管理开销，在便携计算机的情况下就会产生这种开销。

IP地址分配机制在实际中并不是很有效。主要的困难是，用户组织中的网络管理员很难预测出他们对主机地址的增长需求，一般都会过高地估计，犹豫之中就会选择B类地址。大约1990年，事情发展到，按照当时的IP地址分配速度，到1996年NIC就将用完所有的地址。当时采取了两个步骤。第一步是启动开发新的IP协议和寻址机制，结果也就是现在的IPv6；第二步是从根本上修改IP地址分配的方案。一个新的旨在更加有效地利用IP地址空间的地址分配和路由方案诞生了，称为无等级域间路由（CIDR），我们将在3.4.3节中讨论CIDR。

93
|
94

图3-10给出了伦敦大学玛丽女王与韦斯特菲尔德学院（域名：*dcs.qmw.ac.uk*）计算机科学系的部分网络。它拥有多个C类地址规模的子网（利用CIDR将一个B类地址空间细分掉，见3.4.3节的介绍），从138.37.88到138.37.95，由路由器连接。路由器负责将IP数据包传达到所有的子网，同时也负责处理子网间和子网到因特网其他部分的流量。

3.4.2 IP协议

IP协议将数据报从一个主机传到另一个主机，如果需要的话还会经过中间的路由器。完整的IP数据包格式是相当复杂的，图3-17给出了主要组成。有一些头部域没有显示在图中，它们是用于传输和路由算法的。

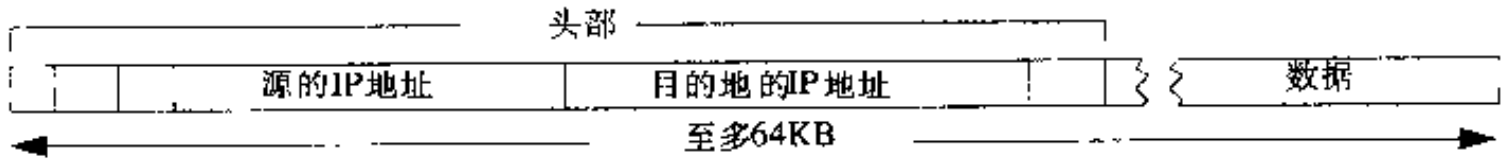


图3-17 IP数据包的设计

IP提供的传输服务被描述成有不可靠或最努力这样的传输语义，因为没有传输上的保证。数据包可能会丢失、重复、延迟或乱序，但这些错误只在底层网络失败或目的地缓冲区满的时候才会发生。IP中惟一的校验和是对于头部的校验和，这在计算上花费不多，还能确保任何寻址和数据包管理数据中发生的错误都会被检测到。没有对数据的校验和，这避免了经过路由器时的开销，而是让再高层的协议（TCP和UDP）来提供它们自己的校验——这是端对端争论中一个实际的例子（见2.2.1节）。

IP层将IP数据报放入适合底层网络（例如以太网）传输的网络数据包中。当IP数据报的长度大于底层网络的MTU时，就在发送端将它分割成多个小的数据包，然后在目的地重新组装。数据包还可能进一步分割以适用从起始地址到目的地址的旅程中所经过的网络（每个数据包都有一个片断标识，用于使得乱序的各个段重新整合起来）

IP层还必须在底层网络中插入消息目的地的“物理”网络地址。该地址可以从因特网网络接口层的地址解析模块获得（见下一小节的介绍）。

地址解析 地址解析模块负责将因特网地址转为特定底层网络所使用的网络地址（有时称为物理地址）。例如，如果底层网络是以太网，那么地址解析模块将把32位的因特网地址翻译成48位的以太网地址。

这种翻译是与网络技术相关的：

- 有一些主机直接与因特网包交换机相连；IP数据包可以不需要地址翻译就路由到它们。
- 一些局域网允许动态地将网络地址分配给主机，这样可以方便地选择地址以匹配因特网地址中的主机标识部分——翻译就是从IP地址中抽取主机标识。
- 对以太网和其他一些本地网络，每个计算机的网络地址都是和它的网络硬件接口固定的，和因特网地址没有直接的关系——翻译取决于主机的IP地址和以太网地址间的对应关系。

现在我们概述一下以太网中IP地址的解析方法。为了能让IP数据包在以太网上传输，每个连在以太网上的计算机都必须实现地址解析协议（ARP）。先考虑同一个以太网中一个主机向另一个主机用IP传送消息的情况。模块在发送前，发送方的IP软件必须将IP数据包中找到的接收方的因特网地址翻译成以太网地址。通过调用发送方的ARP模块来完成这一任务。

每个主机上的ARP模块都维护有一个高速缓冲，保存它以前获得的（IP地址，以太网地址）对。如果需要的IP地址在这个高速缓冲中，请求就会立刻被应答。如果没有，ARP模块会在本地的以太网上发出一个以太网广播数据包（ARP请求数据包），数据包中包括了想要的这个IP地址。本地以太网中的每个计算机都收到这个ARP请求数据包，并用自己的地址和数据包中的IP地址作匹配。如果匹配，就给ARP请求的原发出方发出一个ARP应答，应答中包括自己的以太网地址；如果不匹配，就忽略该数据包。原发出方的ARP模块在自己的本地缓冲中加入新的IP地址→以太网地址的映射，以后遇到相似的请求，它就不需要广播ARP请求了。过了一段时间，每个计算机上都包含了所有计算机的（IP地址，以太网地址）对。这时只有在有新计算机加入到本地以太网时才需要ARP广播。

IP伪冒 我们已经看到，IP数据包中包括一个源地址——发送方计算机的IP地址。它与封

95

装在数据域中的端口地址（对于TCP和UDP数据包）一起，经常被服务器用来生成一个返回地址。不幸的是，并不能保证源地址就是真正发送方的地址。心怀叵测的发送者可以轻易地使用别的地址来代替自己的。这个漏洞已成为多起著名攻击的源头，包括第1章1.4.3节提到的2000年2月的分布式服务拒绝攻击[Farrow 2000]。所使用的方法就是在几个站点向大量的计算机发出*ping*请求（*ping*是一个简单的服务，用于检查主机的可用性）。这些恶意的*ping*请求在它们的发送者地址域中都填上了目标计算机的IP地址，因此*ping*的应答就导向到了目标计算机，使得它们的输入缓冲溢出，导致合法的IP数据包无法通过。这种攻击将在第7章中进一步
96 讨论。

3.4.3 IP 路由

IP层将数据包从源路由到目的地。因特网上的每个路由器实现了IP层的软件，用于提供一个路由算法。

主干 因特网的拓扑逻辑在概念上先被分割成*自治系统*（AS），再被细分为区域。大多数大型机构如大学和大公司的企业内部网被作为AS，通常它们包含几个区域。图3-10中，校园网是一个AS，图中显示的部分是区域。拓扑逻辑图上每个AS有一个主干区域。将非主干区域连接到主干区域的路由器集合，以及将这些路由器互连的链路，两者构成了网络的主干。主干中的链路通常带宽很宽，并且为保证可靠性，链路都被复制。这样的层次结构仅存于概念中，用于管理资源与维护部件。它并不影响IP数据包的路由。

路由协议 作为因特网上使用的第一个路由算法，RIP-1是3.3.5节中距离－向量算法的一个版本。RIP-2（见RFC 1388 [Malkin 1993]）由它发展而来，但包含了其他附加需求，如无类别域间路由、更好的多点路由以及验证RIP数据包以避免路由器受到攻击等。

因特网规模在扩大，路由器的处理能力也在增加，带来的一个趋势是不再使用距离－向量型的算法，因为它收敛速度慢，并且具有潜在的不稳定性。现在趋向于使用3.3.5节中提到的链路－状态型算法，这个算法被称为*开放最短路径优先*（OSPF）。该协议基于Dijkstra[1959]的路径寻找算法，它表现出比RIP算法更快的收敛性。

我们应当注意到在IP路由器中可以渐进地采纳新路由算法。路由算法的改进将导致新版本RIP协议的诞生，而每个RIP数据包会携带一个版本号。当引入一个新的RIP协议时，IP协议并不改变。无论使用哪个版本的RIP协议，IP路由器都会基于一个合理的（未必是最优的）路线，将到达的数据包转发出去。但是对于那些在修改路由表过程中需要合作的路由器，它们必须使用相同的算法。为此，需要使用上面定义的拓扑区域。每个区域内部使用单个路由协议，区域中的路由器在维护路由表时相互合作。仅支持RIP-1的路由器依然很常见，利用新版本协议具有的向后兼容特性，与支持RIP-2与OSPF的路由器共存。

1993年的经验[Floyd and Johnson 1993]表明，RIP路由器信息交换频率为30s，这会导致IP传输性能的周期性。IP数据包传输的平均延迟每隔30s就会有一个尖峰。这可以追述到执行RIP协议的路由器的行为——当接收到一个RIP数据包时，路由器会延迟IP数据包的向前传送，一直到路由表对当前收到的所有RIP数据包的更新过程结束。这会引起路由器一批一批地执行
97 RIP动作。因此建议是路由器采纳15s～45s范围内的随机值作为RIP的更新周期。

默认路由 目前为止，我们对路由算法的讨论隐含要求每个路由器维护了一个完整的路由表，该表显示了到达因特网上每个目的地的（子网或直接连接的主机）路线。就因特网当前

的规模而言，这显然不可行（目的地的数目可能已经超过了1百万，而且仍在快速地增长）。

该问题有两个可能的解决方案，为缓解因特网的增长带来的后果，这两个方案同时被采纳。第一个方案是采纳某种形式的IP地址拓扑数据包。1993年以前从IP地址无法推理到有关其位置的任何信息。1993年为简化与节约IP地址的分配（这在下文的CIDR中讨论），对未来地址的分配决定使用下面的地区位置：

在欧洲，地址194.0.0.0到195.255.255.255

在北美，地址198.0.0.0到199.255.255.255

在中南美，地址200.0.0.0到201.255.255.255

在亚太，地址202.0.0.0到203.255.255.255

因为这些地理区域同时也对应于因特网上确切定义的拓扑区域，并且仅有部分网关路由器提供了对每个区域的访问，所以极大地简化了这些地址范围的路由表。例如，欧洲以外的路由器对于从194.0.0.0到195.255.255.255的地址，可以具有单个表项。路由器使用相同的路由将所有目的地在这个范围内的IP数据包发送到最近的欧洲网关路由器上。注意到，在这个决策之前，IP地址的分配通常与拓扑或地理位置无关，其中的大部分目前仍在使用，1993年的决策无法降低这些地址对应路由表项的规模。

解决路由表规模爆炸的第二个解决方案更简单而且非常有效。它基于这样的观察结果，如果离主干链路最近的关键路由器具有比较完整的路由表，那么大多数路由器中路由表信息的精确性可以放宽。放宽的表现形式为路由表中具有默认的目的地项，此默认项指定了所有目的地地址不在路由表中的IP数据包所使用的路线。为说明该情况，考虑图3-7与图3-8，假想结点C的路由表改为：

路 由：从C		
到	链路	开销
B	2	1
C	本地	0
E	5	1
默认	5	–

结点C不知道结点A与D。它将所有到达A与D的数据包都通过链路5路由到E。结果呢？目的地为D的数据包在路由过程中，不会损失有效性，但目的地为A的数据包会增加一个跳转，需要通过E和B。总之，使用默认路由是在表格大小与路由有效性之间做了一个折衷。但在有些情况下，特别是路由器在中继点位置时，所有向外发送的消息必须穿过某一个点，此时就不会损失有效性。默认的路由机制在因特网中使用很广泛，因特网上也没有一个路由器包含到达所有目的地的路线。 98

在本地子网上的路由 当数据包的目的地主机与发送者在同一网络上时，利用地址的主机标识部分可获得底层网络的目的地主机地址，只需一个跳转就能到达目的地。IP层仅仅使用ARP来获得目的地的网络地址，然后使用底层网络来传输数据包。

如果发送方计算机的IP层发现目的地在另一个网络上，那么它必须将消息发送到一个本地路由器。IP层使用ARP获得网关或路由器的网络地址，再使用底层网络将数据包传送给它们。网关和路由器被连接到两个或更多的网络上，它们具有几个因特网地址，每个连接到的网络对应一个地址。

无类别域间路由（CIDR） 3.4.1节指出IP地址短缺的问题，导致了1996年引入本机制，本机制用于分配地址以及管理路由表中的条目。主要问题在于B类地址不足——那些具有255个以上主机的子网。同时C类地址有足够多可用。CIDR对这个问题的解决方案是给那些需要255个地址以上的子网分配一批连续的C类地址。CIDR机制也允许将B类地址分割，以便把它分配到多个子网里。

将C类地址分批似乎是一个直接的步骤，但除非同时改变路由表的格式，才会对路由表的大小带来显著的影响，进而影响管理路由表的算法的性能。改变路由表的方法是给路由表加一个掩码域。掩码是一个位模式，用于选择与路由表项比较的IP地址部分。这有效地使得主机/子网地址是IP地址的任意部分，比A类、B类与C类地址提供了更多的灵活性，因此有了无类别域间路由这样的名称。同样，路由器的这些改变是增量式的，所以有些路由器执行CIDR，而其他使用旧的基于类别的算法。

该机制可以工作的原因是新分配的C类地址的范围是256的模，因此每个范围表示了一个整数值的C类大小的子网。另一方面，有些子网也使用CIDR分割单个网络中的地址，这个网络可以是A类、B类或C类。如果一组子网完全由CIDR路由器连接到外部，那么该组子网的IP地址范围可以成组分配到每个子网中，其中由任意长的二进制掩码决定子网。

例如，一个C类地址空间可以划分为32组8地址空间。图3-10显示的是，使用CIDR机制将138.37.95这样C类大小的子网划分为多个组，每组包含8个主机地址，用不同的方法路由。不同的组由138.37.95.232/29以及138.37.95.248/29等表示。这些地址中的/29表示附加一个32的
99 掩码，前29个是1，后三个是0。

3.4.4 IPv6

人们在寻找有关IPv4地址局限问题的更永久的解决方案，这导致了开发与使用具有更大地址空间的新版本的IP协议。早在1990年，IETF就注意到IPv4的32位地址会带来的潜在问题，于是启动了开发新版本IP协议的项目。1994年IETF采纳了IPv6，并且给出了版本迁移方法的建议。

图3-18显示了IPv6头的格式。在此我们并不详细给出它们的构造方法。要获得有关IPv6的入门资料，读者可以参考Tanenbaum[1996]或Stallings[1998a]。要获得IPv6设计过程与实现计划详尽的叙述，可以参阅Huitema[1998]。此处将概述IPv6所包含的主要进度。

- 地址空间 IPv6地址有128位（16字节）长。这提供了海量的可寻址实体数：2^{128}，或大约3×10^{38}。据Tanenbaum计算，整个地球表面的每平方米空间可以有7×10^{23}个地址。Huitema则给了稍保守的估计，他假设IP地址的分配像电话号码一样不经济，则整个地球表面的每平方米空间（陆地与水面）可以有1000个IP地址。

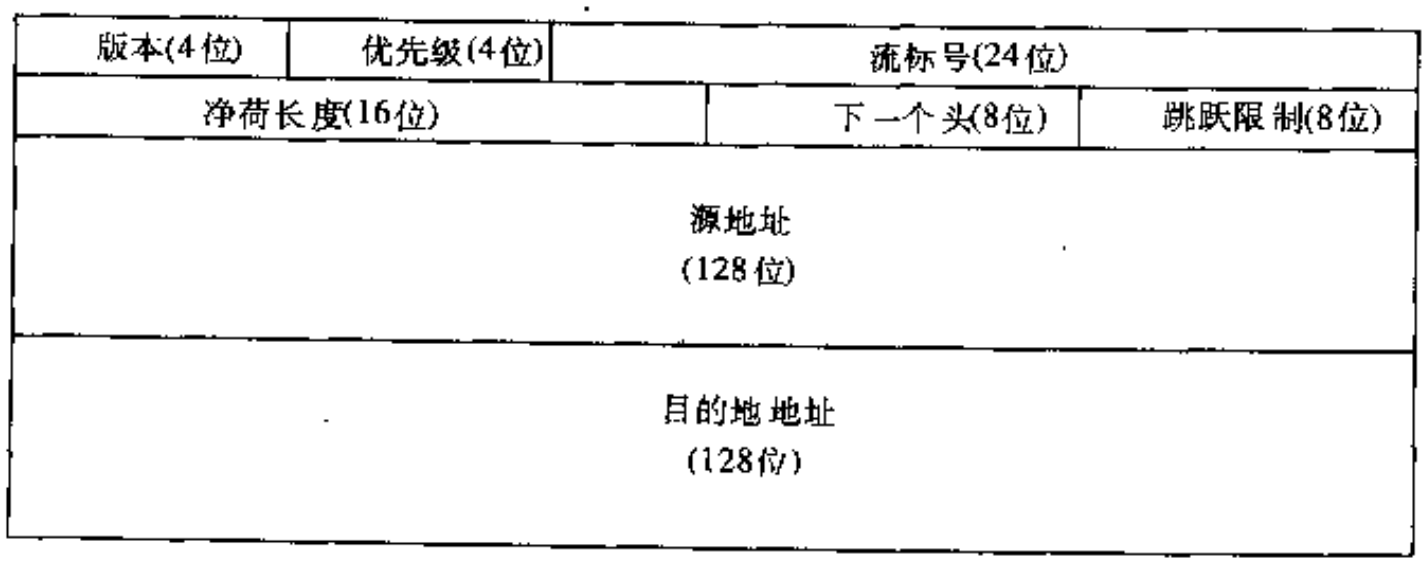


图3-18 IPv6头部格式

IPv6地址空间是被分区的。在此我们不能叙述分区的详情，但即使是最小的分区（其中的一个会包含整个IPv4地址范围，这里地址的映射是一对一的）也比整个IPv4地址空间大。很多分区（整个的72%）被保留下来，目前为止未被定义。两个大的分区（每个包含了1/8的IP地址空间）作为通用用途，将被分配给普通的网络结点。其中的一个根据地址结点的地理位置，而另一个根据机构位置。对于路由目的，这提供了两种不同的策略用于聚类地址——而哪种将更有效或更流行还有待观察。

- 路由速度　基本IPv6头部的复杂度以及在每个结点上的处理时间都被降低。数据包的内容（净荷）不使用任何校验和，一旦一个数据包开始它的旅程，就不可以再分片。前者被认为是可接受的，因为前者可在更高层检测错误（TCP确实包含了一个内容检验和），而后者通过支持在数据包发送前确定最小的MTU而达到目的。 100
- 实时以及其他特别服务　优先级与流标号域与此有关，多媒体数据以及其他实时数据元素序列可作为被标识的流的一部分传输。优先级可与流标号域同时使用，也可以独立使用，以使特定数据包比其他数据包处理速度更快或是更可靠。优先级0到8用于那些即使延迟也不会对应用造成灾难性后果的传输。值8到15保留给传输依赖于时间的数据包，这些数据包或者被迅速地发送，或者被丢弃——迟到的数据包毫无意义。

流标号使得资源被保留，以便满足特定实时数据流。例如，活动音频与视频传输的时间需求。第15章讨论了它们资源分配的需求与方法。当然，因特网上的路由器与传输链路具有有限的资源，以前未曾考虑。为特定用户预留资源的概念和应用。使用IPv6的这些设施将依赖于基础设施的增强，以及使用合适的方法对资源的分配进行收费与仲裁。

- 未来的演进　有关未来演进的关键规定是下一个头域。若为非0，则它定义了在数据包中包含的扩展头的类型。目前的扩展头类型提供了下列类型特定服务的附加数据：路由器信息、路线定义、分片处理、认证、加密信息以及目的地处理信息。每个扩展头类型具有明确大小以及预定义的格式。当出现新的服务需求时，可以定义进一步的扩展头类型。扩展头，如果存在的话，会紧接着基本头，而在净荷之前，它会包含下一个头域，使数据包可以使用多个扩展头。
- 多点与任意点　IPv4与IPv6支持将IP数据包通过单个地址（属于专为组播保留的地址范围）传送到多个主机的传输机制。IP路由器负责将数据包路由到所有订阅了该组（这个组由相关的地址标识）的主机。IP组播通信的详细描述可在4.5.1节找到。另外，IPv6支持一种称为任意点的新的传送模式。该服务将数据包发给至少一个订阅了相关地址的主机。
- 安全　到目前为止，需要认证或保密数据传输的因特网应用极大地依赖于应用层的加密技术。端对端的争论支持应该在应用层实现安全协议的论点，如果在IP层实现安全，那么用户与应用程序开发者依赖于沿线每个路由器都正确地实现了加密算法，为处理密钥，他们还必须信任路由器以及其他中间结点。

在IP层实现安全的好处在于，它可用于不考虑应用程序安全的场合。例如，系统管理员可以将它实现到防火墙中，一致地应用到所有对外的通信中，而内部通信可以不用加密而省却了相应的开销。路由器也可利用IP级的安全机制，保证它们之间交换的路由表更新信息的安全。 101

在IPv6中使用认证与加密的安全净荷扩展头类型实现安全性。这些实现特点与2.3.3节介绍的安全通道概念类似。根据需要，可给净荷加密或者（并且）给净荷进行数字签

名。类似的安全特征也可在IPv4中获得，这时使用了实现IPSec规范（见RFC2411 [Thayer 1998]）的IP隧道。

从IPv4迁移 因特网基础设施的基本协议层改变带来的后果是深刻的。每台主机的TCP/IP协议栈和路由器软件都需要处理IP。很多应用与实用工具中都需要处理IP地址。为了支持新版本的IP协议，上述软件都需要升级。但这个改变是不可避免的，因为IPv4提供的地址空间即将耗尽。负责IPv6协议的IETF工作组定义了一个迁移策略——它主要包括了下列问题的实现：使用隧道技术，将IPv6的路由器和主机“岛屿”与其他IPv6的路由器与主机“岛屿”通信，然后逐渐地形成一个大的岛屿。正如前面所指出的，IPv6路由器和主机在处理混合通信时应该没有任何困难，因为IPv4空间被嵌在IPv6内。

该策略在技术上是完备的，但实现过程非常地慢，这也许是由于经济原因。我们希望，即将来临的对因特网增长的限制，可以充分触发因特网共同体对升级路由器与主机协议软件、修改应用与其他后续工作进行必要的投资。

3.4.5 移动IP

像膝上型和掌上型移动计算机在移动时，它们会在不同地点连接到因特网上。当用户在自己办公室时，移动计算机可以连接到本地的以太网，再通过路由器连接到因特网上；在乘轿车或火车旅行中，可以通过移动电话连接；然后，在另一个结点连接到以太网上。用户可以在任何一个地方访问诸如邮件和Web这样的服务。

对服务的简单访问并不需要给移动计算机保留一个单独的地址，它可在任意结点获得一个新的IP地址；这是动态主机配置协议（DHCP）的目的，它允许新连上的计算机获得一个临时的IP地址以及从本地DHCP服务器上获得诸如DNS服务器这样的本地资源地址，它也需要发现它所访问的每个站点有那些本地服务（如打印、邮件传送等）。发现服务是一种命名服务，将在第9章中介绍。

在移动计算机上会有其他人员需要访问的文件或其他资源，或者它运行了一些分布式应
102 用如共享监控服务，它接收用户拥有的股票超过了一定阈值这样的特定监控事件。当移动计算机在局域网和无线网络之间移动时，如果要让用户和资源共享应用访问移动计算机，它必须保持单个IP号，但IP路由是基于子网的。子网在固定的地点，因而将数据包正确地路由到子网依赖于子网在网络上的位置。

移动IP是后一个问题的解决方案，该方案被透明地实现，因此当移动主机在不同位置的子网中移动时，IP通信正常地继续。它给“家”（home）域的子网中的每个移动主机，永久地分配了一个固定IP地址。

当移动主机连接到它的家时，数据包以正常方式路由到它。当它在其他地方连上因特网时，有两个代理进程负责重新路由。它们是家代理（HA）与外地代理（FA）。这些进程运行在家站点以及移动主机当前所在位置附近的固定计算机上。

HA负责保存移动主机当前位置（移动计算机可以通过该IP地址到达）的最新情况。它在移动主机本身的帮助下完成该功能。当一个移动主机离开家站点时，它会告知HA，HA注意到它的缺席。当主机缺席时，HA的行为就像一个代理服务器。为实现代理功能，它会通知本地路由器取消与移动主机IP地址有关的任何缓存记录。当HA作为一个代理服务器时，HA回应有关移动主机IP地址的ARP请求，将自己的局域网地址作为移动主机的网络地址。

当移动主机到达一个新站点时，它会通知在此站点上的FA。FA给它分配一个“转交”地址——一个本地子网上的新的临时IP地址。然后FA与HA联系，将移动主机的家IP地址以及分配给它的转交地址给HA。

图3-19说明了移动IP的路由机制。当地址为移动主机的家地址的一个IP数据包被传送到家网络上时，它先被路由到HA。然后HA将IP数据包封装到一个移动IP数据包，发送给FA。FA拆解出原来的IP数据包，并通过局域网发送到移动主机。注意，HA与FA将原始数据包重新路由到预期接收者的方法，是在3.3.7节描述的隧道技术的实例。

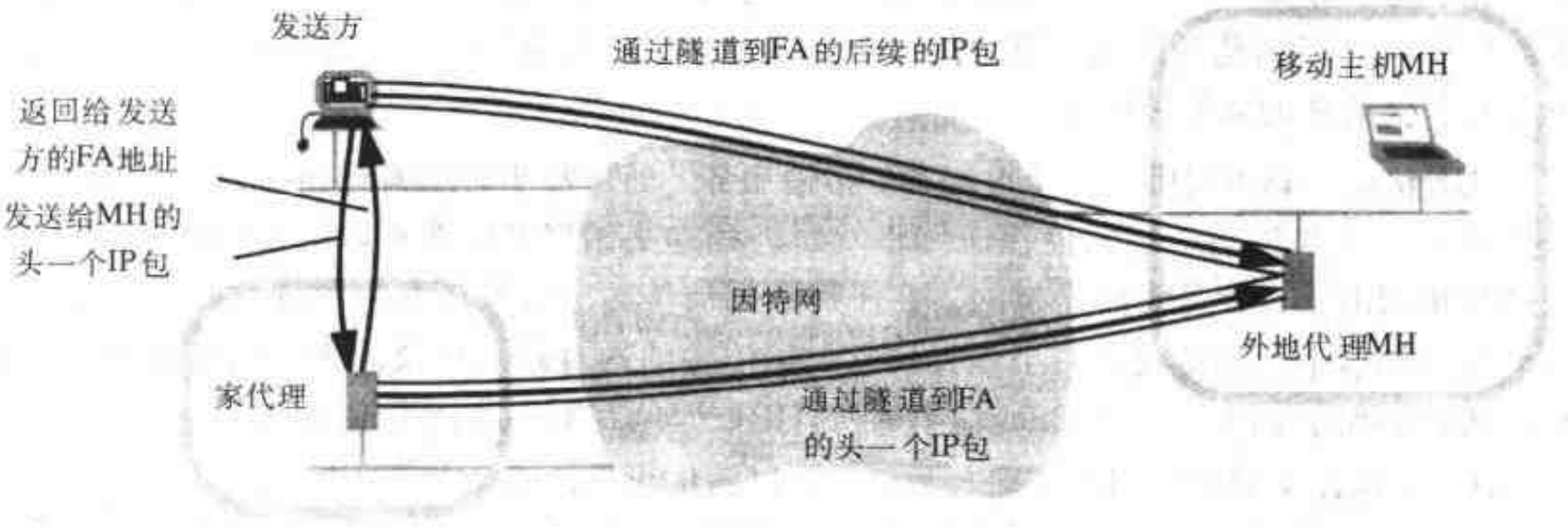


图3-19 移动IP路由机制 103

HA也将移动主机的转交地址送到原来的发送者。如果发送者能处理移动IP，它将注意到新的地址，并且使用它与移动主机接着通信，避免了通过HA重新路由的开销。如果发送者不能处理移动IP，它将忽视地址的改变，而后续的通信依然通过HA重新路由。

移动IP方案是可行的，但效率低。将移动主机作为一等公民的方法会更好一些，这样可以允许主机漫游时无需预先给出通知，并且不使用隧道技术就可将数据包路由到主机。我们应注意到，这个明显很难的技术已在移动电话网中实现——当移动电话在蜂窝乃至国家之间移动时，并不需要改变电话号码。相反地，它们只需时常通知本地移动电话网基站它们的存在。

3.4.6 TCP和UDP

TCP和UDP以一种对应用程序有用的形式，提供了因特网的通信能力。应用开发者可能需要其他类型如提供实时保证或安全的传输服务，但这些服务需要比IPv4更多的网络层支持。TCP和UDP反映了IPv4提供的应用编程级的通信设施。IPv6是另一种方式，它必然会继续支持TCP和UDP，但同时也包含了一些TCP和UDP无法方便访问的功能。当IPv6的部署已足够广时，可引入其他类型的传输服务来挖掘这些功能。

第4章从分布式程序开发者的角度描述了TCP和UDP的特点。此处我们要非常简洁，仅描述它们给IP添加的功能。

端口的使用 第一个要注意的特点是，尽管IP协议提供了一对机器（由IP地址标识）之间的通信，但TCP和UDP作为传输层的协议，必须提供进程到进程的通信。这通过端口的使用完成。端口号用于将消息寻址到特定计算机上的进程，它仅在此计算机上有效。一个端口号是一个16位整数。一旦一个IP数据包被发送到目标主机，TCP或UDP层的软件就通过该主机的特定端口将它分派到一个进程中。

UDP的特点 UDP基本上是IP在传输层的一个复制。UDP数据报被封装在一个IP数据包中，它具有一个包含了源和目的地端口号的短的头部（相应的主机地址在IP头部），一个长度域和一个校验和。UDP不提供传输保证。我们已经认识到IP数据包可能会由于拥塞或网络错误被丢弃。UDP除了可选的校验和外，未增加任何额外的可靠性机制。如果校验和域非零，则接收主机根据数据包内容计算出一个校验值，与接收到的校验和相比，若两者不匹配则数据包被丢弃。

因此，UDP提供了一种在IP上附加最小开销或传输延迟、在进程对（或者在数据报地址是IP组播地址情况下，从一个进程发送到多个进程）之间传送最长达64KB的消息的方法。它不需要任何连接创建开销以及管理用的确认消息。但它的使用只适应于那些不需要可靠传送单个或多个消息的服务和应用。

104

TCP特点 TCP提供了一个更复杂的传输服务。通过基于流的编程抽象，TCP提供了任意长字节串的可靠传输。可靠性保证使得发送进程递交给TCP软件的数据传送到接收进程时，顺序是相同的。TCP是面向连接的。在任何数据被传送前，发送和接收进程必须合作，建立一个双向的信道。连接仅是一个执行可靠数据传输的端对端的协议；中间结点如路由器并无有关TCP连接的知识，一个TCP传输中的所有IP数据包并不一定使用相同的路由。

TCP层包含了额外机制（在IP之上实现）以满足可靠性保证。这些机制包括：

- *次序* TCP发送进程将流分割成数据片序列，然后将之作为IP数据包传送。每个TCP片均具有一个序列号。它对该数据片的第一个字节给出了流中的字节号。接收程序在将数据放入接收进程的输入流前，使用序列号排序收到的数据片。除非所有小号的数据片都已收到并且放入流中，大号的数据片才能被放入，因此，乱序的数据片必须保留在一个缓冲区中，直到它前面的数据片到达。
- *流控制* 发送方管理不能将接收方或是中间结点“淹没”等事宜，这通过数据片确认机制完成。每次接收方成功地接收了一个数据片，它会记录它的序列号。接收方会不时地向发送方发送确认信息，给出输入流中数据片最大的那个号以及一个窗口大小。如果有反向的数据流，则确认信息被包含在正常的数据片中，否则被放在确认数据片中。在确认数据片中的窗口大小域指定了在下一个确认之前发送方被允许传送的数据量。

当一个TCP连接用于与一个远程交互程序通信时，数据猝发产生，但可能产生的数据量很小。例如，键盘输入可能导致每秒仅几个字符，但字符必须能足够快地显示出来，以便用户看到自己的打字结果。这通过在本地缓冲区中设置一个超时值T来实现——一般是0.5s。使用这个简单的机制，一旦数据片已在输出缓冲区中停留T秒，或是缓冲区的内容到达MTU限制，数据片就被发出。该缓冲区机制对交互式延迟不会增加T秒以上。Nagle描述了另一个产生较少流量的算法，它对一些交互式应用更有效[Nagle 1984]。Nagle的算法被用在许多TCP实现中。大多数TCP实现是可以配置的，允许应用程序修改T值，或是在几个缓冲区算法中选择其一。

105

由于无线网络的不可靠性，导致数据包丢失频繁发生，上述的流控制机制对于无线网不是特别适用。这是广域移动通信的WAP协议族采纳另一个传输层机制的原因。但在无线网上实现TCP也是很重要的，为此提出了TCP机制的修改提议[Balakrishnan *et al.* 1995, 1996]。其想法是在无线基站上（有线和无线网络之间的网关）实现一个TCP支持成分。该成分探听来自/到达无线网络的TCP片，重传任何未被移动接收方快速确认的外发数据，以及当注意到序列号有间隔时，请求重传接收数据。

- 重发　发送方记录它发送的数据片的序列号。当它接收到一个确认时，它知道数据片被成功接收，从而将之从外发缓冲区中清除。如果在一个指定超时时间内，数据片并没有得到确认，发送方就重发它。
- 缓冲　接收方的接收缓冲区用于平衡发送方和接收方之间的流量。如果接收进程发出*receive*操作的速度比发送进程发出*send*操作的速度慢很多，那么缓冲区的数据量会增加。通常情况下，数据在缓冲区满之前被取出，但最终缓冲区会溢出，此时到来的数据片不被记录就简单地被丢弃了。因此，接收方不会给出相应的确认，而发送方被迫重新发送。
- 校验和　每个数据片包含了一个对头部和数据的校验和，如果接收到的数据片和校验和不匹配，则数据片会被丢弃。

3.4.7 域名

第9章将详细描述域名系统（DNS）的设计与实现，在此我们给出一个简单的介绍，以完成本章有关因特网协议的讨论。因特网支持一种使用符号名标识主机和网络的机制，例如*binkley.cs.mcgill.ca*或*essex.ac.uk*。名字实体被组织成一个层次结构。名字实体被称为域，而符号名被称为域名。域被组织成一个层次，以便反映它们的组织结构。命名层次与构成因特网的网络物理布局完全独立。域名对于用户很方便，但它们在被用作通信标识符之前，必须被翻译成因特网地址（IP地址），这是DNS服务的责任。应用程序将请求发送给DNS，以便将用户指定的域名转化成因特网地址。

DNS实现为一个服务器进程，可在因特网的任意主机上运行。每个域至少有两台DNS服务器，一般情况下会更多。每个域的服务器持有域名树的部分视图。它们至少必须存储在自 106
己域中的所有域名和主机名，但它们经常包含树的更大部分。若DNS服务器接收到的请求中，需要翻译的域名在自己那部分树以外，则DNS服务器通过向相关域的服务器发送请求，递归地自右向左解析名字的各个成分。翻译结果被缓存在处理原始请求的服务器上，以便未来处理同一域名请求时，无需查阅其他服务器就可以解析该名字。若不广泛地使用缓存技术，DNS将没法工作，因为基本上在每种情况下“根”名字服务器都会被查询，从而形成一个服务访问瓶颈。

3.4.8 防火墙

几乎所有的组织机构都需要因特网连接，以便给顾客或其他外部用户提供服务，同时使内部用户可以访问信息和服务。大多数机构中的计算机是不同的，运行多种操作系统和应用软件。软件的安全性相差更大，有些提供了先进的安全保证，但大多数软件没有能力或有很少能力保证进入的通信是可靠的，或是向外的通信是秘密的。总之，在一个有很多计算机和很多软件的企业内部网中，不可避免地会出现系统的有些部分在安全性攻击下会非常地脆弱。攻击的形式将在第7章中详细讨论。

防火墙的目的在于监视和控制进出企业内部网的所有通信。一个防火墙本身由一组进程实现，作为通向企业内部网的网关（图3-20(a)），它应用了机构规定的一个安全策略。

防火墙安全策略可能包括下面任一或所有的目标：

- 服务控制　用于确定内部主机上的哪些服务可以接受外部访问，以及需要拒绝哪些服务请求。外发的服务请求和响应也被控制。这些过滤行为可以基于IP数据包的内容以及其

中包含的TCP和UDP请求。例如，到达的HTTP请求目标应该是公开的Web服务器主机，否则可能会被拒绝。

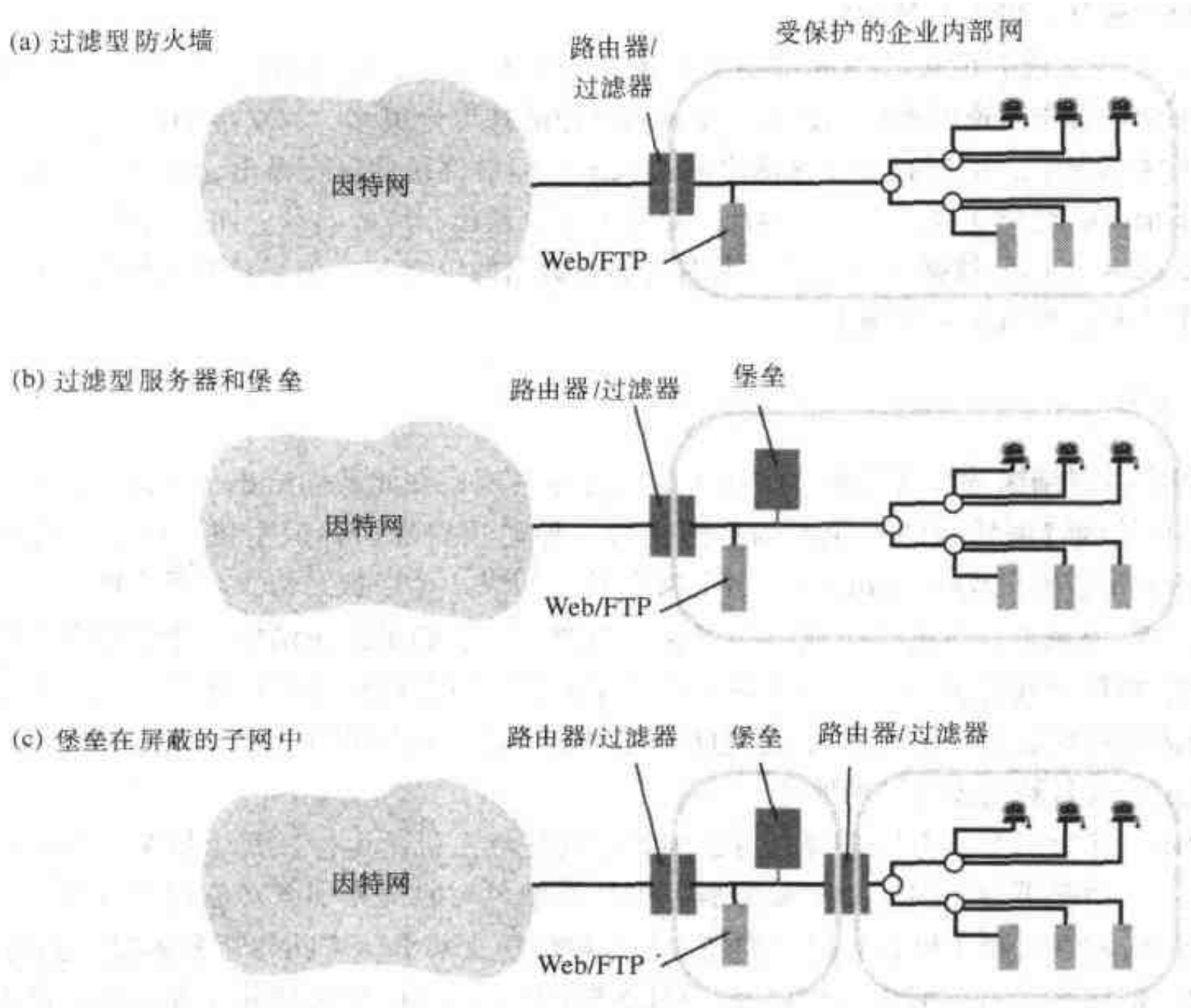


图3-20 防火墙配置

- 行为控制 行为控制用于防止破坏公司策略的、反社会的、找不到可辨认的合法目的的行为，这些行为被怀疑为攻击的一部分。其中的某些过滤行为可在IP或TCP层进行，但其他可能会需要在更高层对消息进行解释。例如，过滤邮件垃圾攻击，可能会需要检查消息头中发送方的邮件地址甚至是消息内容。
- 用户控制 机构可能会希望分辨用户，允许其中某些人访问外部服务，而其他人被禁止。另一个可能大家更易接收的用户控制例子是，避免接收系统管理员组成员以外的其他用户的软件，以防止病毒感染或是维护软件标准。上面的例子中如果不禁止普通用户使用Web，事实上是很难实现。

用户控制的另一个实例是拨号以及其他不在站点上的用户连接的管理。如果防火墙同时也是调制解调器连接的主机，它可以在连接时认证用户，并且对所有通信使用一个安全通道（以防止外来的窃听、伪装和其他攻击）。这是下一节描述的虚拟私网（VPN）技术的目的。

这些策略必须由过滤操作表达，而这些操作由在不同层的过滤进程执行：

- IP数据包过滤 这是一个检查单个IP数据包的过滤进程，它可能会根据目的地和源地址决策。它也有可能检查IP数据包的服务类型域，并根据类型解释数据包的内容。例如，它可以根据目的端口号过滤TCP数据包，因为服务通常位于众所周知的端口上，这使得

可以根据请求的服务而过滤数据包。例如，很多站点禁止外部客户使用NFS服务器。 107～108

为性能目的，IP过滤通常由路由器操作系统内核中的进程执行。如果使用多个防火墙，第一个防火墙可能标识某些数据包以便后来的路由器做更彻底的检查，同时让“干净”的数据包继续。也有可能基于IP数据包序列进行过滤，例如，在执行登录命令前，禁止对FTP服务器的访问。

- TCP网关　TCP网关进程检查所有的TCP连接请求以及数据片的传输。在安装了TCP网关进程后，可控制TCP连接的创建，检查TCP片的正确性（一些服务拒绝攻击用残缺的TCP片来破坏客户的操作系统），在需要时，它们可以被路由到应用层网关进行内容检查。
- 应用层网关　应用层网关进程作为应用进程的代理。例如，希望有这样的策略：允许特定内部用户向特定外部主机创建Telnet连接。当一个用户在本地运行Telnet程序时，程序试图和远程主机建立一个TCP连接，请求被TCP网关截获。TCP网关启动一个Telnet代理程序，原有的TCP连接被路由到它。如果代理通过了Telnet操作（用户被授权使用远程主机），那么它会建立另一个通向远程主机的连接，并由它中转所有来往的TCP数据包。一个类似的代理进程将代表每个Telnet客户而运行，而类似的代理可能被FTP和其他服务所采纳。

一个防火墙通常由在不同协议层的多个进程组成。为性能和容错起见，通常在防火墙中使用的计算机超过一台。在下面描述的并由图3-20说明的所有配置中，我们显示了一个不受保护的Web服务器和FTP服务器：它包含了一些发布信息，并且对公共访问不加防范，而服务器软件确保只有授权的内部用户可以修改它。

IP数据包过滤通常由路由器执行——路由器是一台至少有两个在不同IP网络上的网络地址的计算机。路由器运行一个RIP进程，一个IP数据包过滤进程以及其他数个可能的进程。路由器/过滤器仅运行可信的程序，以保证过滤策略的执行。这保证了不能运行特洛伊木马进程，以及路由器和过滤器软件未被修改或破坏。图3-20(a)显示了仅依赖于IP过滤并只使用了一个路由器的简单的防火墙配置，图3-10中的网络配置包含了两个作为此类防火墙的路由器/过滤器。为性能和可靠性起见，该配置有两个路由器/过滤器，它们遵循同样的过滤策略，而第二个没有增加系统的安全性。

当需要TCP和应用层网关进程时，这些进程通常会运行在单独的计算机上，称为堡垒（该术语源于城堡的构筑，城堡会有一个突出的了望塔用于保护城堡）。堡垒计算机是一台在企业内部网中的由IP路由器/过滤器保护的主机，它运行TCP和应用层网关（图3-20(b)）。与路由器/过滤器一样，这个堡垒只运行可信的软件。在一个足够安全的企业内部网内，代理必须处理所有对外部服务的访问。读者可能对Web访问代理已经很熟悉了。它们都是防火墙代理的应用实例，通常和Web缓存服务器（见第2章的描述）以某种方式集成构建。这些代理以及其他代理可能需要很大的处理和存储资源。 109

串联地部署两台路由器/过滤器可以提高安全性能，此时位于单独子网内的堡垒以及公共服务器和路由器/过滤器相链接（图3-20(c)），这种配置在安全方面有以下好处：

- 如果堡垒策略严格的话，企业内部网内主机的IP地址根本不需要对外界公开，企业内部网计算机也无需知道外部地址，因为所有外部通信都要通过堡垒内的代理进程，而代理进程可以访问两边的计算机。
- 如果第一个路由器/过滤器被攻破，那么由于原本外部不可见而不易受攻击，第二个路

由器/过滤器会继续承担挑选和拒绝不可接收的IP数据包的责任。

虚拟私网 通过使用IP层的密码保护安全通道，虚拟私网（VPN）将防火墙保护的界限延伸到本地企业内部网之外。在3.4.4节中，我们概述了如何使用IPSec隧道技术对IPv6和IPv4进行IP安全扩展，这些是实现VPN的基础。VPN可用于外部个人用户或者在使用公共因特网链接、位于不同站点的企业内部网之间实现安全连接。

例如，一个工作人员需要通过因特网服务连接到企业企业内部网，一旦连接成功，就需要拥有防火墙内部用户同样的权利。若本地主机实现了IP安全，则上面要求可以完成。本地主机保存了与防火墙共享的一个或多个密钥，这些密钥用来建立安全通道。安全通道机制将在第7章中详细介绍。

3.5 网络实例研究：以太网、无线LAN和ATM

到目前为止，我们已经讨论了有关构造计算机网络的原理，描述了因特网的“虚拟网络层”IP。在结束此章前，本节描述3种实际网络的原理与实现。

在20世纪80年代初，美国电子与电气工程师协会（IEEE）建立了一个委员会来制订局域网的一系列标准（802委员会[IEEE 1990]），它的分委员会产生了一系列LAN的关键标准。大多数情况下，这些标准基于从20世纪70年代出研究而来的已有工业标准。相关的分委员会以及迄今出版的标准如图3-21所示。

IEEE 标准编号	标　题	参考文献
802.3	CSMA/CD网络（以太网）	[IEEE 1985a]
802.4	令牌总线网	[IEEE 1985b]
802.5	令牌环型网	[IEEE 1985c]
802.6	城域网	[IEEE 1994]
802.11	无线局域网	[IEEE 1999]

图3-21 IEEE 802网络标准

这些标准的差别在于性能、有效性、可靠性和成本，但它们都提供了在中短距离相对而言比较高的网络带宽。IEEE 802.3以太网标准极大地赢得了有线LAN市场的战斗。作为有线LAN的代表技术，我们将在3.5.1节中描述它。尽管以太网可实现几种带宽，但它们的操作原理是相同的。

IEEE 802.5令牌环型网标准在90年代是一个重要的竞争者，它的优势是比以太网更有效并支持带宽保证。但它现在已经从市场上消失了。如果读者对这种有趣的LAN技术感兴趣，可以在www.cdk3.net/networking找到它的简要描述。

IEEE 802.4令牌总线是为具有实时需求的工业应用而开发的，并仍在为该领域服务。IEEE 802.6城域网标准覆盖高达50km的距离，成为跨越城镇的网络。在Tanenbaum[1996]的书中，可找到4种IEEE 标准（802.3~802.6）的详细描述以及它们之间的比较。

IEEE 802.11无线LAN标准的出现有一点晚，但通过Lucent（WaveLAN）和其他制造商的产品，目前已经在市场上占据了重要的位置，并且可能会随移动计算设备或是无处不在计算设备的到来而变得更为重要。IEEE 802.11标准支持简单无线传输/接收器设备之间的距离在150m之内高达11Mbps速度的通信。我们将在3.5.2节描述它的操作原理。IEEE 802.11网络的

详情可以在Crow *et al.* [1997]、Kurose以及Ross[2000]中找到。

ATM技术在20世纪80年代末到90年代初之间从电信和计算机界的研究和标准化工作中产生[CCITT 1990]。它的目的是提供适合于电话、数据以及多媒体（高品质语音与视频）应用的、高带宽的广域数字网络技术。尽管它的采纳比预期的缓慢，但ATM现在是超高速广域网的主宰技术。它在某些地方的LAN应用中替代了以太网，但在LAN市场上不是太成功，这是因为100Mbps和1000Mbps以太网可以通过低得多的价格与之竞争。我们将在3.5.3节中概述ATM的操作原理。ATM的详细情况以及其他高速网络技术可以在Tanenbaum[1996]和Stallings[1998a]中找到。

3.5.1 以太网

以太网是1973年[Metcalfe and Boggs1976；Shoch等 1982; 1985]在Xerox Palo Alto研究中心作为个人工作站和分布式系统研究计划的一部分开发的。该实验以太网是第一个高速局域网，展示了将单个场地上的计算机通过高速局域网连接以低错误率、无交换延迟的高速速率互相通信的可行性和用途。最初的以太网原型以3Mbps运行，现在以太网系统的带宽已经扩展到10Mbps到1000Mbps。许多专用网用同一种的操作方法实现，同时具有适合于不同应用的开销/性能。使用这种原理的操作，在最低开销水平，可用100Kbps~200Kbps的传输速率将低成本的微机连接起来。

我们将描述在IEEE 802.3标准[IEEE 1985a]中定义的10Mbps以太网的操作原理。它是第一个广泛部署的，可能目前仍是最流行的局域网技术。尽管新的安装更多的是选择100Mbps的变种。我们将继续列出目前以太网传输技术更重要的变种以及可用的带宽。所有以太网变种的综合描述，请参见Spurgeon[2000]。

单个以太网是一个简单的或分支总线，它使用的传输介质由通过网络集线器或中继器连接的一个或多个连续的电缆段组成。集线器和中继器是连接线路的设备，它使得同样的信号能穿过所有线路。几个以太网可在以太网网络协议层次上通过以太网交换机或网桥连接。交换机和网桥在以太网帧的层次上操作，将地址为邻接以太网的帧转发过去。对于诸如IP这样的高层协议，连接起来的以太网可看出是一个单个网络（如在图3-10中，IP子网138.37.88和138.37.94都由几个以太网组成，它们由标记为Eswitch的部件连接）。特别是ARP协议（3.4.2节），它可以跨越连接好的以太网组来解析IP地址；每个ARP请求都广播到子网中所有连接的网络上。

以太网的操作方法定义为“具有检测冲突的载波侦听多路访问”（简称为CSMA/CD），属于竞争总线类网络。竞争总线网络使用单种传输介质连接所有的主机。管理介质访问的协议称为介质访问控制（MAC）协议。由于单一链路连接所有主机，所以MAC协议将数据链路层（负责在通信链路上传输数据包）协议和网络层（负责将数据包传输到主机）协议的功能合并在一个协议层。

数据包广播 CSMA/CD 网络中的通信方法是在传输介质上广播数据数据包。所有工作站都在不断地“监听”介质上传输的数据包的地址是否是自己。任何想发送消息的工作站会广播一个或多个数据包（在以太网中称为帧）到介质上。每个帧包含目的地工作站地址、发送工作站地址和表示传输消息的变长比特序列。数据传输以10Mbps（或为100/1000Mbps以太网制定的更高速度）速度进行，数据包长度为64字节到1518字节。因此在10Mbps以太网上传输一个数据包

的时间是50μs-1200μs，具体时间取决于数据包的长度。在IEEE标准中MTU被定义为1518字节，尽管除了需要限制冲突产生的延迟外，没有任何其他技术原因需要制订固定的极限。

目的地工作站的地址通常是指单个网络接口。每个工作站的控制器硬件接收数据包的一个副本。通过比较每个数据包的目的地址和本地的硬编码地址，忽略地址为其他工作站的数据包，将地址匹配的数据包接收到本地主机。目的地址也可定义为一个广播或者组播地址。普通地址通过最高位与广播地址和组播地址相区分（前者为0，后者为1）。全为1的地址被保留为广播地址，用于一条消息被网络上所有工作站接收的情况。例如，可用于实现ARP IP地址解析协议。任何收到具有广播地址的数据包的工作站将把数据包传送到本地主机。一个组播地址指定了一种受限的广播方式，一个数据包由一组其网络接口被配置为可接收组播地址的工作站接收。并不是所有的以太网接口都可以识别多点发送地址。

以太网网络协议（在一对主机之间传输以太网数据包）由以太网硬件接口实现，而传输层以及传输层之上的协议需要协议软件。

以太网数据包格式 以太网上传输的数据包（或称之为帧更精确）具有如下的格式：

字节：7	1	6	6	2	46≤长度≤1500	4
前同步符	S	目的地址	源地址	数据长度	要传输的数据	校验和

除了已提到目的地址和源地址，帧还包括一个8字节的固定前缀、一个长度域、一个数据域和一个校验和。前缀用于硬件定时目的，包含了7个字节的前同步符，每个前同步符由位模式10101010组成，前同步符后面紧跟着单字节的开始帧分界符（在图中是S），分界符的模式为10101011。

尽管标准不允许单个以太网多于1024个工作站，但以太网的地址占了6个字节，提供了2^{48}个地址空间。这使得每个以太网硬件接口制造商可以给硬件接口一个惟一的地址，保证所有互连的以太网中的工作站都有惟一的地址。美国IEEE作为以太网地址分配的权威，将48位地址的不同范围分配给以太网硬件接口制造商。

数据域包含要传输的消息的全部或部分（如果数据包长度超过1500字节）。46字节的下限保证数据包最小长度为64字节，设置下限是必要的，这用于保证网络上所有工作站的冲突都可被检测，下文对此做了解释。

帧校验序列是一个校验和，由发送者产生并插入，用于接收者验证数据包。那些带有不正确的校验和的数据包由接收工作站的数据链路层丢弃。这是端对端争论的另一个例子：为了保证消息的传输，必须使用像TCP这样的传输层协议，确认每个数据包的接收并重传未被确认的数据包。在局域网中数据出错的情况非常少，所以当需要保证传输时，使用这种错误恢复方法完全令人满意。并且当不需要保证数据传输时，它可以采用像UDP这样的，开销比较小的协议。

113

数据包冲突 即使数据包的传输时间相当短，网络上两个工作站之间同时传输消息的情况也有可能发生。如果一个工作站试图传输一个数据包，而没有检查介质是否被另一个工作站使用，就会产生冲突。

以太网有3种机制来处理这种可能性。第一种称为载波侦听。每个工作站的接口硬件监听在介质上出现的信号（称为载波，类似于无线电广播）。当一个工作站想传输一个数据包，它会一直等到介质上没有信号出现才开始传输。

不幸的是载波侦听不能阻止所有的冲突。冲突存在的原因是，一个在介质的某个点插入的信号到达所有的点需要有穷时间τ（以电波速度传播：大约2×10^8m/s）。考虑两个工作站A和B几乎同时准备传输。如果A首先开始传输，在A开始传输之后的$t<\tau$时间内，B检查介质，未发现有信号，于是B开始传输但它干扰了A的传输，最后A和B的数据包都会被干扰破坏。

从这种干扰中恢复的技术称为冲突检测。当一个工作站通过硬件输出端口传送一个数据包时，它也监听它的输入端口，并比较两个信号。如果两者不同，则说明发生了冲突。此时工作站停止传输并产生阻塞信号，通知所有工作站产生了一个冲突。我们已经指出，最小数据包的长度确保冲突通常可以被检测到。如果两个工作站几乎同时从网络的相反端传输，它们在2τ之内不会意识到冲突（因为当第一个发送者接收到第二个信号时，必须仍然继续发送）。如果它们发送的数据包所花的广播时间小于τ，就注意不到冲突，因为每个发送工作站直到它传输完自己的数据包，才会看别的数据包，而中间的工作站将会因为同时接收两个数据包而产生数据崩溃。

阻塞信号发出之后，所有传输和监听的工作站取消当前的数据包。传输工作站不得不试图重新传输它们的数据包。这产生了进一步的困难。如果发生冲突的工作站都试图在阻塞信号之后立即重传它们的数据包，可能发生另一个冲突。为避免这种情况，使用了称为后退的技术。发生冲突的每个工作站选择在传输之前等一段时间$n\tau$。n是一个随机整数，由每个工作站分别选取，并局限于在网络软件中定义的一个常数L。如果冲突进一步产生，L的值被加倍，必要时整个过程可以重复10次。 114

最后，接收工作站的接口硬件计算接收序列，并将之与数据包中传送的校验和相比。使用所有这些技术，连接到以太网的工作站在无任何集中控制或同步的情况下，可以管理介质的使用。

以太网的效率 以太网的效率是成功传送的数据包数与无冲突下能传输的理论最大值的比率。它受τ值影响，因为数据包传送后的2τ秒间隔是冲突的“可能窗口”——在数据包开始传输2τ后不会有冲突发生。网络上工作站的数目以及它们的活动性也会影响效率。

对于1km的电缆，τ的值比5μs小，从而冲突概率会足够小，以确保高的有效性。尽管当通道利用率大于50%时，争夺通道所造成的延迟变得相当明显，以太网仍可以获得80%到95%的通道利用率。负载是变化的，因此，不可能在一个固定的时间内保证给定信息的传输。因为网络可能在消息准备传输时变得满负荷运行。在一个给定延迟内消息被传递的概率等同或好于其他网络技术。

Shoch与Hupp[1980]在施乐PARC报告的以太网性能的实际度量中确认了上述分析。在实际中，在分布式系统中使用的以太网负载变化很大。很多网络主要用于异步客户－服务器交互，在大多数情况下，网络在无工作站等待传输、通道利用率接近1的状况下运行；支持大量用户进行批量数据访问的网络会承受更多的负载；那些携带多媒体流的网络，如果有几个流并行传输的话，则有可能被淹没。

物理实现 上面的叙述定义了所有以太网的MAC层协议。市场对以太网的广泛采纳，使得我们可以获得实现了以太网算法的低成本的控制器硬件，它成为很多桌面计算机与消费类计算机的标准部件。

为提供不同的性能/成本的平衡以及利用增长的硬件性能，有很多不同的以太网物理实现。不同点来源于使用不同的传输介质——同轴电缆，双绞线（与电话线相似）以及光纤——它们

具有不同的传输范围、而使用更高的信号速度，会带来更高的系统带宽与更短的传输范围。IEEE采纳了不同的物理层实现标准以及用于区分它们的命名机制。诸如使用10Base5与100BaseT这样的名字，它们具有如下形式：

<R><B><L>　　此处：*R* = 以Mbps计的数据率

115　　*B* = 媒体信号类型（基带或宽带）

L = 以米/100为单位的最大数据片长度或者T(双绞线)

我们将当前可用的标准配置以及电缆类型的带宽与最大传输范围列在下表里：

	10Base5	10BaseT	100BaseT	1000BaseT
数据率	10Mbps	10Mbps	100Mbps	1000Mbps
最大数据片长度：				
双绞线（UTP）	100m	100m	100m	25m
同轴电缆（STP）	500m	500m	500m	25m
多模式光纤	2000m	2000m	500m	500m
单模式光纤	25000m	25000m	20000m	2000m

以T结尾的配置由UTP电缆（非屏蔽双绞线即电话线）实现，它被组织为集线器层次结构，计算机作为树的叶子。在这种情况下，表中给出的数据片长度是计算机到集线器的最大允许长度的两倍。

3.5.2 IEEE 802.11无线LAN

本节将总结无线LAN技术中必须处理的无线网络的特殊性质，同时解释IEEE 802.11是如何处理这些特征的。IEEE 802.11标准扩展了以太网（IEEE 802.3）技术中的载波侦听多路复用（CSMA）原理以适应无线通信特征。802.11标准旨在支持距离在150m之内的速度达到11Mbps的计算机间通信。

图3-22是一个包含无线LAN的一部分企业内部网，移动无线设备间通过基站和企业内部网其他设备通信，这里基站称为有线LAN的访问点。通过访问点与传统LAN连接的无线网络称为基础设施网络。

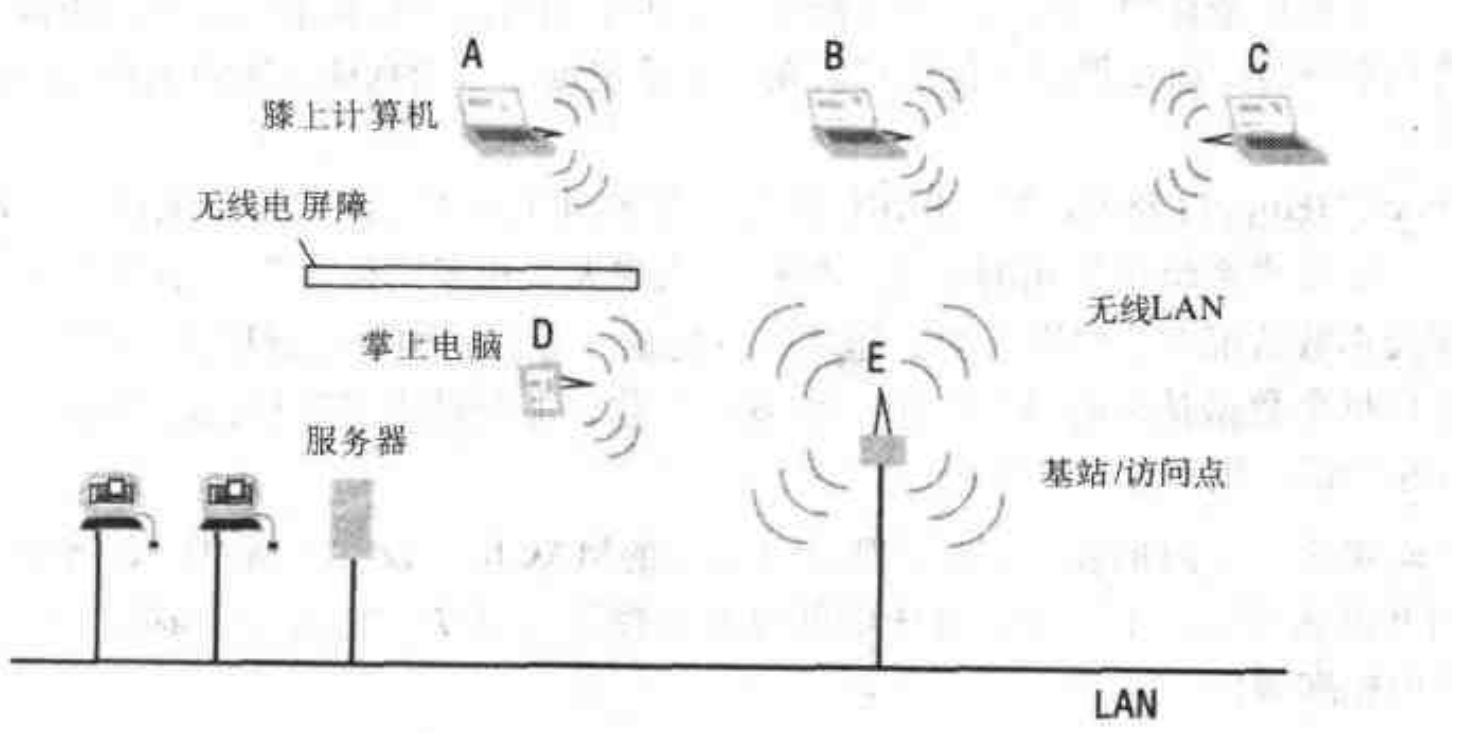


图3-22　无线LAN配置

无线网的另一种配置方式称为临时网络，临时网络不包括访问点或基站。它们通过同一邻域的无线接口检测到彼此的存在，然后在运行中建立起网络。当同一房间内的两个或者多个膝上计算机用户发起与任何可用站点的连接时，就会形成一个临时网络。它们可以通过在某台机器启动文件服务器进程来共享文件。

IEEE 802.11网络中站点采用无线电频率信号（2.4GHz波段）或者红外线作为传输介质。标准中的无线电版本在商业上广受注意，下面我们将介绍它。它采用了不同的频率选择或者跳频技术，用以避免外部干扰以及独立的无线LAN之间相互干扰（后者我们不准备详细讨论）。我们集中讨论对CSMA/CD机制所做的改变，这些改变使得广播传输可以用到无线电传输中。

和以太网一样，802.11MAC协议为所有的站点提供相同的机会使用传输通道，任意站点之间可以直接传输，MAC协议控制不同站点对通道的使用。对以太网而言，MAC层起到了数据链路层和网络层的作用，它负责将数据包发送到网络的主机上。

使用无线电波而非电线作为传输介质会产生一些问题。这些问题都来源于以太网使用的载波侦听和冲突检测机制仅在整个网络的信号强度大致相同时才有效。

我们回忆一下，载波侦听的目的是确定在发送和接收站点间的所有结点上传输介质是否空闲，冲突检测的目的是确定在接收者邻域内的介质是否在传输时能不受传输干扰。由于无线LAN空间内信号强度不均匀，所以载波侦听和冲突检测可能出现如下几种错误：

- 站点隐藏　载波侦听没能检测到网络上另一个站点正在传输。下面用图3-22加以说明，掌上电脑D正在向基站E传输，由于图中所示的无线电屏障，膝上计算机A可能发现不了D的信号。于是A开始传输，若不采取措施防止A传输，在E点将造成冲突。
- 信号衰减　电磁波传输按照平方速度衰减，因此随着和传输者距离的增加，无线电信号强度迅速衰减。一个无线LAN内的某个站点可能在其他站点的范围之外。如图3-22所示，虽然膝上计算机A或C可以成功地向B或E传输信号，但A却可能检测不到C的传输。信号衰减使得载波侦听和冲突检测都失效。
- 冲突屏蔽　不幸的是，以太网中用来检测冲突的侦听技术在无线电网络中并不是十分有效。因为上面提到的平方衰减规律，本地产生的信号总是比别处产生的信号强很多，极大地覆盖了远程传输。因此，膝上计算机A和C可能同时向E传送，两者都没有检测到冲突，但E却收到了乱码。

尽管如此，IEEE 802.11网络中并没有废弃载波侦听，而是通过在MAC协议中加入时隙保留机制，对载波侦听机制进行了加强。这种方案称为具有避免冲突的载波侦听多路访问（CSMA/CA）。 116 ~ 117

在站点准备发送时，首先检测介质状态。如果没有检测到载波信号，它假设以下条件之一为真：

1. 介质可用。
2. 范围之外的站点正在请求获得一个时隙。
3. 范围之外的站点正在使用它保留的时隙。

时隙保留协议包括发送者和接收者之间交换一组短消息（帧）。首先是发送者给接收者发一个请求发送（RTS）帧，RTS帧指定了需要的时隙长度。接收者回复清除发送（CTS）信号，并重复时隙的长度。这种交换的效果如下：

- 发送者所及范围内的站点将获得RTS帧，并记录时隙长度。
- 接收者所及范围内的站点将获得CTS帧，并记录时隙长度。

结果是，发送者和接收者所及范围内的所有站点在规定的时隙内都不传输；留出空闲通道给发送者传输一定长度的数据帧。最后，接收者对数据帧的成功传输发出确认信息，以帮助处理通道的外部干扰问题。MAC协议的时隙保留特征从以下几个方面有助于避免冲突：

- CTS帧有助于避免站点隐藏和信号衰减问题。
- RTS和CTS帧很短，所以冲突的风险也小。如果检测到冲突或者RTS没有得到CTS回复，则像以太网那样，使用一个随机后退周期。
- 如果正确地交换了RTS和CTS帧，随后的数据和确认帧应当没有冲突，除非间歇性信号衰减导致第三方没有接收到RTS帧或者CTS帧。

安全性 通信的隐私性和完整性显然是无线网络中必须要关注的。范围内有发送/接收器的任何一个站点都可能加入这个网络，如果失败，它可能窃听其他站点之间的传输。IEEE 802.11标准考虑了这个问题，它要求加入网络的站点要交换认证信息，证明它拥有共享密钥。这和第7章描述的类似，基于共享密钥认证机制，可以有效地防止没有密钥的站点加入网络。

预防窃听要通过简单加密办法来实现。通过将传输数据内容与一组随机数序列进行位异或操作而实现内容屏蔽。此随机数序列从共享密钥中产生，任何拥有此密钥的站点可以重新
118 生成此序列以及重现原始数据。这是7.3节描述的流加密技术的一个应用实例。

3.5.3 异步传输模式网络

设计ATM是用来传输多种不同数据的，包括像音频和视频这样的多媒体数据。它是一种快速包交换网，基于一种称为信元中继的数据包路由方式，信元中继比传统包交换方式快很多。它的高速来源于在传输中间站点避免流控制和检错，因此，传输链路和结点的数据出错的可能性必须很低。性能提高的另一个原因是数据传输采用简短、定长的单元，这可以减少中间站点缓冲区大小、复杂性和队列延迟。ATM以面向连接的模式运行，但只有在有足够资源时才能建立连接，一旦建立连接，就可以保证质量（即带宽和延迟特征）。

ATM是一种数据交换技术，它可在已有的数字电话网上实现，后者一直是同步的。当叠加在像SONET同步光纤网[Omidyar and Aldridge 1993]那样的同步高速数据链路网上时，ATM构成了一个更灵活的带有许多虚连接的高速数据包网络。每个ATM虚连接提供带宽和延迟保证，因此其虚电路可支持多种不同速度的服务，包括语音（32Kbps）、传真、分布式系统服务、视频和高清晰度电视（100Mbps~150Mbps）。ATM[CCITT 1990]标准推荐虚电路提供的数据传输率要高达155Mbps或622Mbps。

ATM可在光纤、铜线和其他传输介质上使用本地模式直接实现，使用现有的光纤技术，它们的带宽可以达到每秒10亿位。该模式在局域网和城域网中使用。

ATM服务的结构分为3层，图3-23中用深黑条表示，ATM适配层为端到端层，只在发送/接收主机上实现。它的目标是支持在ATM层上叠加现有的TCP/IP与X.25协议等高层协议。为适应不同的高层协议的要求，适配层的不同版本提供不同的适配功能。为了支持高层协议，它包括了一些通用的功能，如数据包装配和拆卸。

ATM层提供面向连接的服务，传输称为信元的定长数据包。一条连接由虚拟路径上的虚拟通道序列组成。一个虚拟通道（VC）是在源到目的地的物理路径中的一条链路的两个端点之间的一个逻辑上的单向连接。一条虚拟路径（VP）是交换结点间物理路径上的一组虚拟通
119 道。虚拟路径的目标是支持一对端点间的半永久连接。连接建立时动态分配虚拟通道。

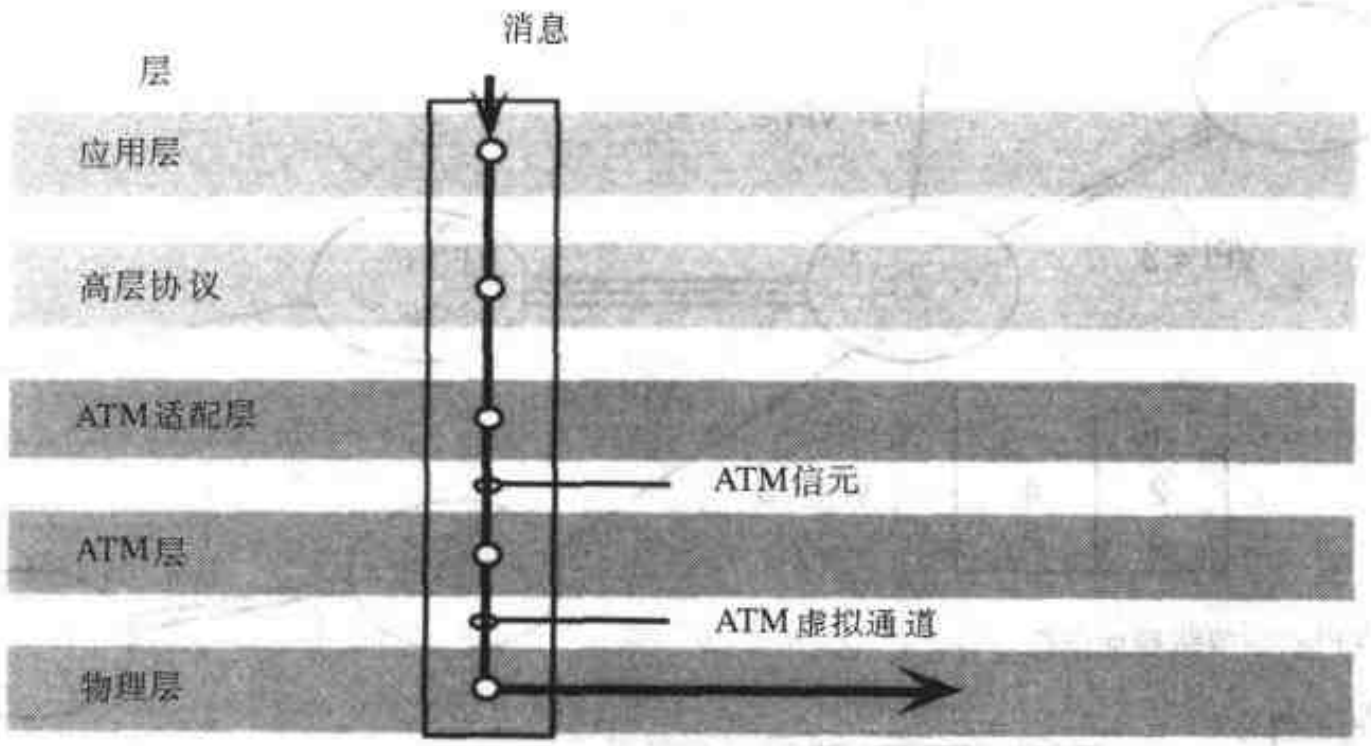


图3-23 ATM协议层

ATM网络的结点扮演下列3种不同角色：

- 主机，发送和接收信息。
- VP交换机，保存显示了输入虚拟路径和输出虚拟路径之间关系的表格。
- VP/VC交换机，为虚拟路径与虚拟通道保存类似的表格。

一个ATM信元有5字节的信元头和48字节的数据域（如图3-24所示），即使只有部分数据，也发送完整的数据域。信元头分别包括一个虚拟通道标识符和一个虚拟路径标识符，两者为信元在网络上路由提供信息。虚拟路径标识符指定了信元传输的物理链路上的一条特定虚拟路径，虚拟通道标识符指定虚拟路径内一条特定的虚拟通道。其他信元头域用于表示信元类型、信元损失优先权和信元边界。

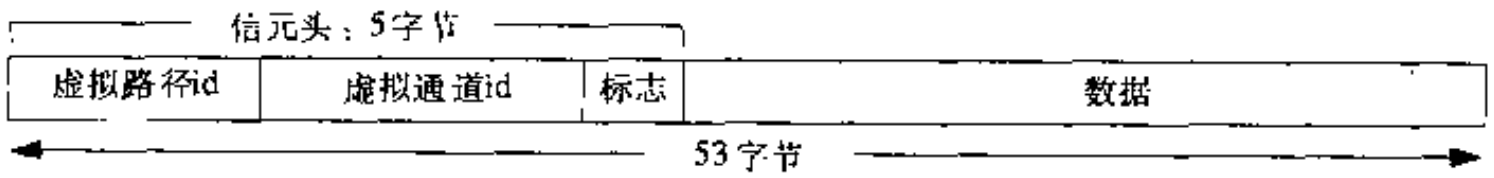


图3-24 ATM信元结构

当信元到达一个VP交换机时，在路由表中查找信元头中的虚拟路径标识符，以找出与输出物理路径对应的虚拟路径标识符，如图3-25所示。交换机将新的虚拟路径标识符置入信元头，然后将该信元传输到输出物理路径上。VP/VC交换机基于虚拟路径标识符和虚拟通道标识符可完成类似的路由功能。

注意，VP和VC标识符是局部定义的，该机制的优点是无需定义整个网络范围内的全局标识符，如果定义的话会是一个很大的数字。全局寻址机制还会带来管理开销，需要在交换机上有信元头和表格等多种信息。 120

ATM提供了低延迟的服务——交换延迟大约是每个交换机25μs。例如，一条消息通过10台交换机需要250μs延迟。这和分布式系统的性能要求（见3.2节）很相符，表明ATM网络可以支持进程间通信和客户－服务器交互，性能与现有局域网的性能持平或更优。ATM还可能以高达600Mbps的速度，提供保证业务质量的、适合于多媒体数据流传输的超宽带通道。纯ATM网络甚至可以达到10Gbps的速度。

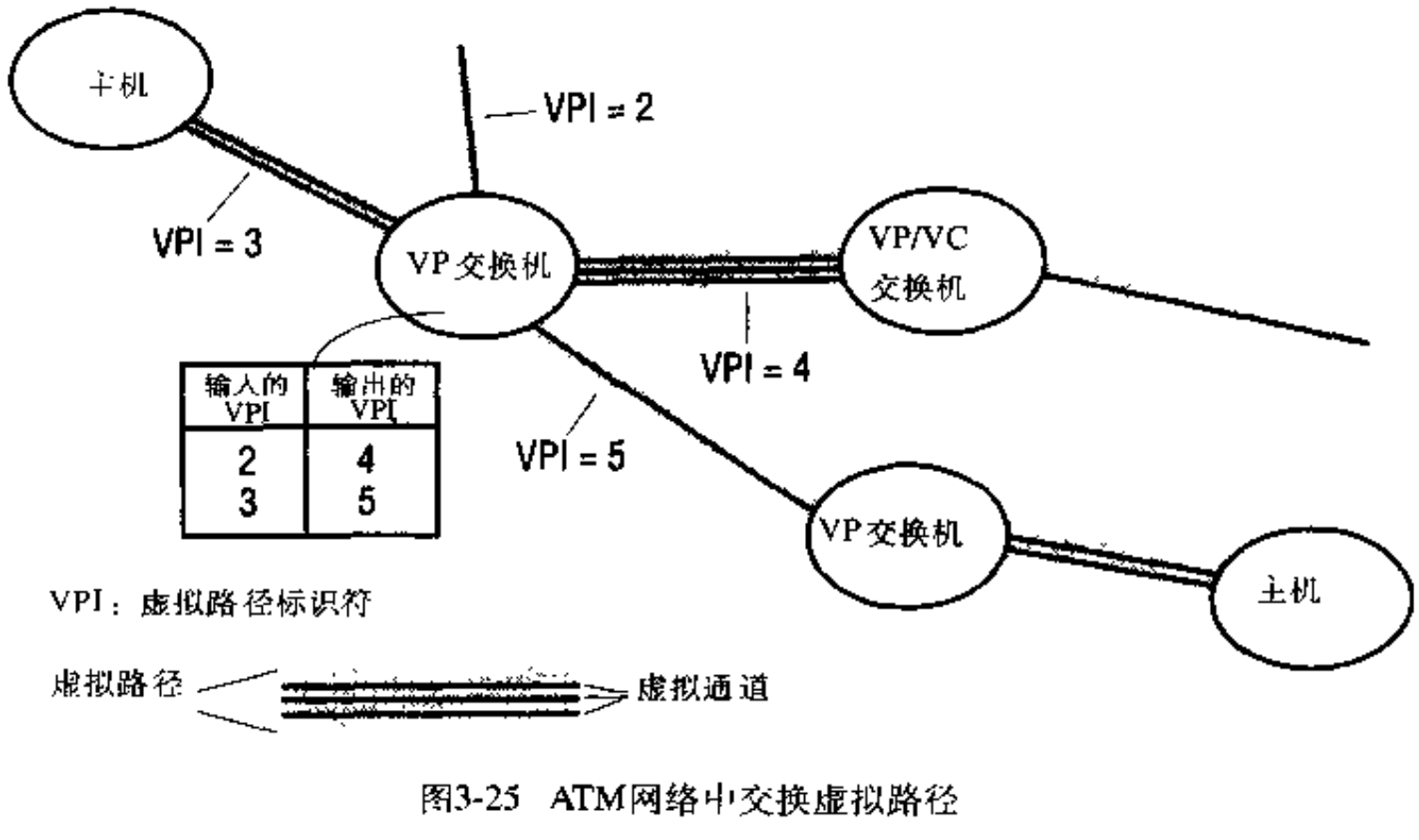


图3-25 ATM网络中交换虚拟路径

3.6 小结

我们集中讨论了作为分布式系统基础的网络概念和技术，并从一个分布式系统设计者的角度对此做了探讨。数据包网络和分层协议是分布式系统的通信基础。局域网基于共享介质上的数据包广播技术，其中以太网是主流技术。广域网则通过包交换将数据包路由到目的地。路由选择是关键问题，有不少的路由算法，其中距离－向量算法是最基本而且非常有效的一种。必须要有拥塞控制来防止接收方和中间结点的缓冲区溢出。

通过在路由器连接的一组网络上叠加“虚拟”互联网络协议构造互联网络，因特网的
121 TCP/IP协议使得互联网上的计算机可以以统一的方式通信，无论它们是在同一个局域网上，还是在不同的国家。因特网标准包括许多适合广域分布式应用的应用层协议。IPv6为将来因特网的发展预留了一个极大的地址空间，并对服务质量和安全性等新的应用需求做了规定。

移动IP支持了移动用户广域漫游，基于IEEE 802.11无线LAN支持移动用户的本地连接。基于有服务质量保证的虚电路，ATM提供了超宽带的异步通信。

练习

3.1 一个客户将200字节的请求消息发送到一个服务，服务产生了一个5000字节的应答。估算在下列情况下，完成请求的时间。

（i）使用无连接（数据报）通信（例如，UDP）。

（ii）使用面向连接的通信（例如，TCP）。

（iii）服务器进程与客户进程在同一机器上。

[在发送或接收时每个数据包的延迟（本地或远程）：5ms

建立连接的时间（仅对TCP）：5ms

数据传输速率：10Mbps

MTU:1000字节

服务器请求处理时间：2ms

假设网络处于轻负载状态。]

3.2 因特网非常大，任一路由器均无法容纳所有目的地的路由信息，那么因特网路由机制如何处理这个问题呢？

3.3 以太网交换机的任务是什么？它维护了哪些表？

3.4 构造一个类似于图3-5的表格，叙述当因特网应用与TCP/IP协议组在以太网上实现时，每个协议层软件所做的工作。 122

3.5 端到端论点[Saltzer *et al.* 1984]是如何用于因特网的设计的？考虑虚拟电路网协议替代IP会如何影响万维网的可行性。

3.6 我们能确认因特网的两台计算机不会有同一个IP地址吗？

3.7 对于下面应用层和表示层协议，比较各自使用无连接（UDP）与面向连接（TCP）协议的实现方案。

（i）虚拟终端访问（例如，Telnet）。

（ii）文件传输（例如，FTP）。

（iii）用户位置（例如，rwho, finger）。

（iv）信息浏览（例如，HTTP）。

（v）远程过程调用

3.8 解释为什么在广域网络中会发生数据包序列到达目的地的顺序与出发时不同的现象。为什么这在局域网中不可能出现？它可能在ATM网中出现吗？

3.9 在诸如Telnet这样的远程终端访问协议中需要解决一个特定的问题，即在传送数据之前将“kill信号”之类的异常事件从终端传输到主机的需求。kill 信号需要在任何其他正在进行的传输之前到达目的地。讨论该问题在无连接与面向连接协议下的解决方案。

3.10 使用网络层广播定位本地资源的缺点有哪些？

（i）在单个以太网中？

（ii）在企业内部网中，以太网组播在何种程度上改善了广播？

3.11 提出一个改善移动IP的机制，以便一个移动设备可以访问某Web服务器，该移动设备有时通过移动电话连接到因特网上，而更多的是通过有线网连接到因特网上。

3.12 显示在图3-7中，标号为3的链路被断开后，图3-8中路由表的改变序列（根据图3-19中给出的RIP算法）。

3.13 将图3-13作为基础，描述到服务器的一个HTTP请求的分割与封装过程以及相应的应答。假设请求是一个短的HTTP消息，而应答包括了至少2000字节的HTML。

3.14 考虑在Telnet远程终端客户中使用TCP。客户是如何缓冲键盘输入的？在（a）一个Web服务器（b）一个Telnet应用（c）一个具有连续鼠标输入的远程图形应用使用TCP时，研究Nagle与Clark的流控制算法[Nagle 1984, Clark 1982]与3.4.5节描述的简单算法，比较这两个算法。

3.15 类似于图3-10，构造你的学校或工作单位的网络图。 123

3.16 描述如何配置防火墙，以保护你的学校或工作单位的局域网。需要阻截哪些进出的请求？

3.17 一个连接到以太网的新安装的个人计算机是如何发现本地服务器的？它是如何将IP地址翻译成以太网地址的？

3.18 防火墙是否可以防止如3.4.2节“IP伪冒”一段描述的拒绝服务攻击？对于这样的攻击，可以使用哪些其他方法？

124

第4章　进程间通信

4.1　简介
4.2　因特网协议的API
4.3　外部数据表示和编码
4.4　客户 - 服务器通信
4.5　组通信
4.6　实例研究：UNIX系统的进程间通信
4.7　小结

本章关注分布式系统进程间通信协议的特征，包括它自身的固有特征和支持对象之间通信两方面。

用于因特网中进程间通信的Java API提供了数据报和流通信两种方式。本章给出这两者的描述，同时讨论它们的故障模型。它们提供了可互换的通信协议构造成分。

本章讨论消息中数据对象集合的表示协议和引用远程对象的协议。

本章讨论支持分布式程序中两种常用通信模式的协议的构造，这两种通信模式是：

- 客户 - 服务器通信——在该通信模式下，请求和应答消息是远程方法调用或远程过程调用的基础。
- 组通信—— 在该通信模式下，同一消息被发送到几个进程。

本章将用UNIX系统的进程间通信作为实例研究。

4.1　简介

本章和下一章关注中间件。本章关注图4-1中的深色层，该层的上层在第5章中讨论，它涉及将通信集成到编程语言模型中，例如，通过提供远程方法调用（RMI）或远程过程调用（RPC）。远程方法调用允许一个对象调用一个远程进程中对象的方法。远程方法调用的系统实例有CORBA和Java RMI。类似地，远程过程调用允许客户调用在远程服务器上的一个过程。

图4-1　中间件层

第3章讨论了因特网传输层协议UDP和TCP，但没有叙述中间件和应用程序如何使用这些协议。下一节将介绍进程间通信的特征，并从编程人员的角度讨论UDP和TCP，给出这两个

协议的Java接口，然后讨论它们的故障模型。本章的最后一节作为实例研究，给出了UDP和TCP的UNIX套接字接口。

UDP的应用程序接口提供了消息传递抽象——进程间通信的最简单形式。这使得一个发送进程可给一个接收进程传递一个消息。包含这些消息的独立的数据包被称为数据报。在Java和UNIX API中，发送方用套接字指定目的地。套接字是由目的地计算机上的目标进程使用的一个特定端口的一个非直接引用。

TCP的应用程序接口提供了一对进程之间的双向流抽象。相互通信的信息由没有消息边界的一连串数据项组成。流成为生产者－消费者通信[Bacon 1998]的组成成分。生产者和消费者形成一对进程，前者的作用是产生数据项，后者的作用是消费数据项。由生产者发送给消费者的数据项按到达顺序排在队列中，直到消费者准备好接收它们。在没有数据项时，消费者必须等待。如果存放入队数据项的存储空间耗尽的话，生产者必须等待。

考虑到不同的计算机可能对简单数据项使用不同的表示，本章的第3节讲述应用程序使用
126 的对象和数据结构如何翻译成适合在网络上发送消息的格式。第3节还讨论了分布式系统对象引用的合适的表示方法。

本章的第4节和第5节讨论支持客户－服务器和组通信的协议设计。请求－应答协议用于支持RMI或RPC方式的客户－服务器通信。组播协议用于支持组通信。组播是进程间通信的一种，在组播中，一组进程中的一个进程将同一消息传送给小组的所有成员进程。

消息传递操作被用于构造协议，支持特定的进程角色的和通信模式，例如远程方法调用。通过检查角色和通信模式，设计基于实际交换的通信协议和避免冗余是可能的。特别是这些专门的协议不应该包括冗余的确认。例如，在一个请求－应答通信中，确认应答通常被认为是冗余的，因为应答本身就是一个确认。如果一个更专用的协议要求发送方确认或其他特定的特征，那么它们要提供专门的操作。着眼于用最少量的消息交换来实现协议，那么可行的想法就是仅在需要时才增加专门的功能。

4.2 因特网协议的API

本节讨论进程间通信的普遍特征，然后作为一个例子讨论因特网协议，解释程序员如何通过UDP消息或TCP流使用这些协议。

4.2.1节回顾了2.3.2节引入的消息通信操作*send*和*receive*，并讨论它们如何相互同步以及如何在分布式系统中指定消息目的地。4.2.2节介绍套接字，它用在UDP和TCP的应用编程接口中。4.2.3节讨论UDP和它的Java API。4.2.4节讨论TCP和它的Java API。Java的API是面向对象的，但它很类似原来在Berkeley BSD4.x UNIX操作系统中设计的API，相关的讨论见4.6节。研究本节程序例子的读者应该查阅Java联机帮助或Flanagan[1997]的书，以便从中得到所讨论的类（在包*java.net*中）的完整说明。

4.2.1 进程间通信的特征

在一对进程间进行消息传递需要两个消息通信操作支持：*send*和*receive*，它们均用目的地和消息定义。为了使一个进程能够与另一个进程通信，一个进程要发送一个消息（字节序列）到目的地，而另一个在目的地的进程接收消息。该活动涉及数据从发送进程到接收进程
127 的通信，可能包含两个进程的同步。4.2.3节给出了因特网协议在Java API中*send*和*receive*操作的定义。

同步通信和异步通信 每个消息目的地与一个队列相关。发送进程将消息加到远程队列中，接收进程从本地队列中移走消息。发送进程和接收进程之间的通信可以是同步的也可以是异步的。在同步通信中，发送进程和接收进程在每个消息上同步，这时，*send*和*receive*都是阻塞操作。每次发送一个*send*后，发送进程将一直阻塞直到发送了相应的*receive*。每次发送一个*receive*，进程将一直阻塞直到消息到达。

在异步通信中，*send*操作是不阻塞的，因为只要消息被复制到一个本地缓冲区，发送进程就被允许继续进行其他处理，消息的传递处理与发送进程是并行工作的。*receive*操作可有阻塞和不阻塞的两种。在不阻塞的*receive*操作中，接收进程在发出*receive*操作后可继续执行它的程序，这时*receive*操作在后台提供一个缓冲区，但它必须通过轮询或中断独立地接收缓冲区满的通知。

在支持多线程进程的系统环境如Java中，阻塞型*receive*的缺点较少，因为*receive*操作可由一个线程发出，而该进程中的其他线程仍然是活动的。阻塞型*receive*的一个显著优点是根据到达的消息同步接收线程在实现上很简单。非阻塞型的通信效率似乎更高，但接收进程需要从它的控制流之外获取到达的消息，这包含有额外的复杂工作。因此，当前的系统通常不提供非阻塞型的*receive*。

消息目的地 第3章说明了在因特网协议中消息被发送到（因特网地址，本地端口）对。本地端口是计算机内部的消息目的地，用一个整数指定。一个端口只有一个接收者（组播端口除外，参见4.5.1节）但有多个发送者。进程可以使用多个端口接收消息。任何知道端口号的进程都能给端口发消息。服务器通常公布它们的端口号供客户使用。

如果客户使用一个固定的因特网地址访问一个服务，那么这个服务就必须总在这个地址的计算机上运行，以保持该服务的有效性。下列任一方法可避免这种情况，提供位置透明性：

- 客户程序通过名字使用服务，在运行时用一个名字服务器或绑定程序（见5.2.5节）把服务的名字翻译成服务器位置，这样就使得服务能被重定位但不能迁移（迁移指在系统运行时移动服务所在的位置）。
- 操作系统如Mach（见第18章）给消息目的地提供了一个位置独立的标识符，并将它们映射到一个底层地址以便于将消息传递到端口，这种方法允许服务迁移和重定位。

替代端口的另一种方法是将消息发给进程，系统V就是这种做法[Cheriton 1984]。然而，端口有它的优点，它为一个接收进程提供了几个可选的入口点。在一些应用中，将同一个消息分发给一组进程是非常有用的。因此，一些IPC系统提供了将消息发给目的地组的能力（目的地或者是进程或者是端口）。例如，Chorus[Rozier 等 1990]提供了端口组。 128

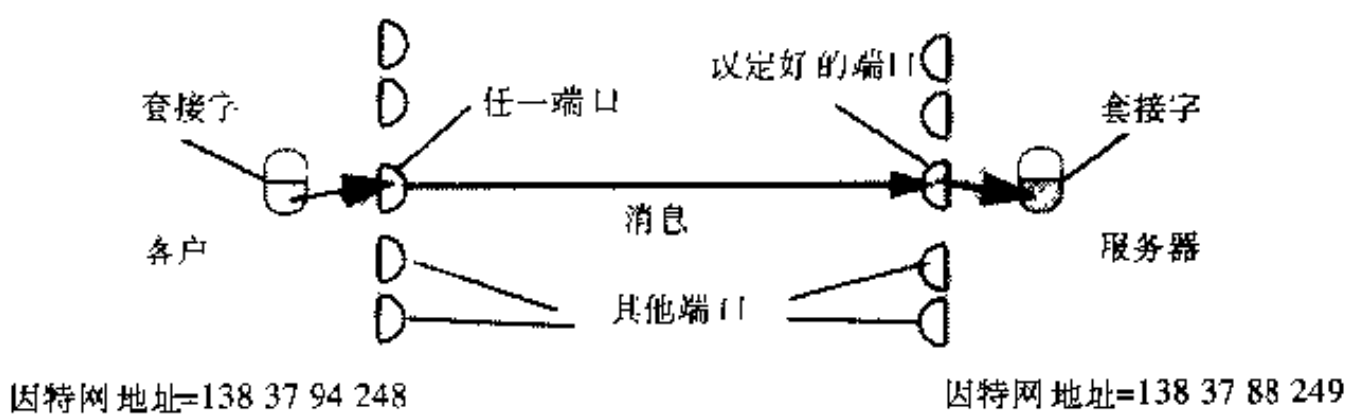
可靠性 第2章以有效性和完整性定义了可靠通信。就有效性而言，如果一个点对点消息服务无论丢失了多少“合理”数量的数据包，总能保证发送消息，那么它就被称为可靠的。相反的，如果在丢失一个数据包的情况下，消息就不能保证发送，那么这个点对点消息服务就是不可靠的。从完整性而言，到达的消息必须完好无损，且没有重复。

排序 有些应用要求消息按发送方顺序发送，就是说，按照发送方发送消息的顺序。以与发送方顺序不一致的顺序发送消息被这些应用认为是一个失败的发送。

4.2.2 套接字

两种形式的通信（UDP和TCP）都使用套接字抽象，套接字提供进程间通信的一个端点。

套接字源于BSD UNIX但也在UNIX的大多数版本中出现，包括Linux以及Windows NT和Macintosh OS。进程间通信是在一个进程的一个套接字与另一个进程的一个套接字之间传送一个消息，如图4-2所示。对接收消息的进程，它的套接字必须绑定到本计算机的一个因特网地址和一个本地端口。发送到特定因特网地址和端口号的消息只能被套接字与这个因特网地址和端口号相连的进程接收。进程可以使用同一套接字发送和接收消息。每个计算机有大量（2^{16}）可用的端口号用于本地进程接收消息。任一进程可利用多个端口来接收消息，但一个进程不能与同一台计算机上的其他进程共享端口。使用IP组播的进程是一个例外，因为它们共享端口（参见4.5.1节）。然而任何数量的进程都可以发送消息到同一个端口。每个套接字与特定协议相连——或是UDP或是TCP。



129

图4-2 套接字和端口

Java API中的因特网地址 因为UDP和TCP底层的IP数据包被发送到因特网地址，所以Java提供了一个类，*InetAddress*，用以表示因特网地址。用户用域名服务（DNS）的主机名表示计算机（见3.4.7节）。例如，包含因特网地址的*InetAddress*实例通过调用*InetAddress*的静态方法（以DNS主机名做参数）创建。该方法使用DNS获得相应的因特网地址。例如，对于DNS名为*bruno.dcs.qmw.ac.uk*的主机，为了得到表示其因特网地址的对象，使用下列语句：

```
InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmw.ac.uk");
```

该方法会抛出*UnknownHostException*。注意，用户不需要给出显式的因特网地址值。事实上，*InetAddress*类封装了因特网地址表示的细节，这样，该类的接口就不依赖于表示因特网地址的字节数（在IPv4中是4字节、而在IPv6中是16字节）。

4.2.3 UDP数据报通信

由UDP发送的数据报从发送进程传输到接收进程，它不需要确认或重发。如果发生故障，消息可能到达不了。数据报在进程之间传送，一个进程发送它，另一个进程接收它。需要发送或接收消息的进程必须首先创建与一个因特网地址和本地端口绑定的套接字。服务器将把它的套接字绑定到一个服务器端口，该端口能让客户知道以便客户可以向该端口发送消息。客户将它的套接字绑定到任何一个空闲的本地端口。*receive*方法除了获得消息还获得发送方的因特网地址和端口，这些信息允许接收方发送应答。

下面讨论与数据报通信有关的一些问题：

- 消息大小 接收进程需要指定大小固定的用于接收消息的字节数组。如果消息大于数组大小，那么它在到达时就被截断了。底层的IP协议允许数据包的长度最大为2^{16}字节，包括消息头和消息本身。然而，大多数环境将其大小限制在8KB。如果应用程序有大于最大值的消息，那么需要将该消息分割成若干段。通常，由应用决定消息大小，不需要选

用很大的值，只要适用即可。

- 阻塞　UDP数据报通信使用非阻塞型的*send*和阻塞型的*receive*。当*send*操作将消息传递给底层的UDP和IP协议后就返回，由UDP和IP协议负责将消息传递给目的地。消息到达时被放在与目的端口绑定的套接字队列中。通过该套接字的下一个*receive*调用可以从队列中获取该消息。如果没有一个进程具有绑定到目的端口的套接字，那么消息就在目的地被丢弃了。

  除非在套接字上设置了超时，否则*receive*方法将一直阻塞直到接收到一个数据报。如果调用*receive*方法的进程在等待消息时还有其他工作要做，那么应该单独安排一个线程。线程的讨论见第6章。例如，当服务器从客户端接收到一个消息，消息可能指定要做 130
  的工作，这时，服务器将使用不同的线程或做要做的工作或等待其他客户发送的消息。
- 超时　一直阻塞的*receive*适用于正在等待接收客户请求的服务器。但在有些程序中，发送进程可能崩溃或期待的消息已经丢失，而使用*receive*操作的进程不适合无限制地等待。为了适应这些需求，套接字上要设置超时。选择适当的超时间隔很困难，但与传输消息所要求的时间相比，它应该相对大些。
- 任意接收　*receive*方法不指定消息的源。相反，调用*receive*可获得从任何地方发到它的套接字上的消息。*receive*消息返回发送方的因特网地址和本地端口，允许接收方检查消息的来源。可以将数据报套接字连接到特定的一个远程端口和因特网地址，这时，套接字仅从那个地址接收和发送消息。
- 故障模型　第2章给出了信道的故障模型，用两个特性，即完整性和有效性定义了可靠通信。完整性要求消息不能被损坏或重复。利用校验和可保证接收到的消息被损坏的可能性微乎其微。第2章的故障模型可用于UDP数据报的故障模型，UDP数据报存在下列故障：
- 遗漏故障　消息偶尔会丢失，或是因为校验和错误或是因为在发送端或目的地端没有可用的缓冲区空间。为简化讨论，我们把发送遗漏故障和接收遗漏故障（如图2-11所示）视为信道中的遗漏故障。
- 排序　消息有时没有按发送方顺序发送。

为了获得所要求的可靠通信的质量，使用UDP数据报的应用要自己提供检查手段。可以在一个有遗漏故障的服务的基础上通过使用确认来构造可靠传送服务。4.4节讨论如何在UDP上构造可靠的用于客户－服务器通信的请求－应答协议。

UDP的使用　对一些应用，使用偶尔有遗漏故障的服务是可接受的。例如，域名服务（负责查找在因特网上的DNS名）就是在UDP上实现的。有时UDP数据报是一个很有吸引力的选择，因为它们不包含与保证消息传递相关的开销。三大主要的开销是：

1. 需要在源和目的地存储状态信息
2. 传输额外的消息
3. 发送方的时间延迟

这些开销的原因见4.2.4节的讨论。 131

UDP数据报的Java API　Java API通过两个类：*DatagramPacket*和*DatagramSocket*，提供数据报通信。

- *DatagramPacket*　该类提供构造函数，用一个包含消息的字节数组、消息长度和目的地

套接字的因特网地址和本地端口号生成一个实例，见下图：

数据报的数据包

包含消息的字节数组	消息长度	因特网地址	端口号

*DatagramPacket*实例可以在进程之间传送，其中一个进程发送，另一个进程接收。

该类还提供了另一个构造函数在接收消息时使用，它的参数是一个用于接收消息的字节数组和数组长度。*DatagramPacket*存放一个接收到的消息以及消息长度和发送套接字的因特网地址和端口。用*getData*方法从*DatagramPacket*获取消息。用*getPort*和*getAddress*方法获取端口和因特网地址。

- *DatagramSocket* 该类支持套接字发送和接收UDP数据报。它提供一个以端口号为参数的构造函数，用于需要使用特定端口的进程。它也提供一个无参数的构造函数，允许系统选择一个空闲的本地端口。如果端口已经被使用或在UNIX下指定了一个保留端口（小于1024的数字），那么这些构造函数会抛出*SocketException*。

  类*DatagramSocket*提供的方法包括：

  - *send*和*receive* 这些方法用于在一对套接字之间传送数据报。*send*的参数是包含消息及其目的地的*DatagramPacket*实例。*receive*的参数是一个空的*DatagramPacket*，用于存放消息、消息长度和来源。方法*send*和*receive*能抛出*IOException*。
  - *setSoTimeout* 该方法可以设置超时。设置超时后，*receive*方法将阻塞指定的时间，然后抛出一个*InterruptedIOException*。
  - *connect* 该方法用于连接到一个指定的因特网地址和远程端口。这时，套接字仅能从那个地址发送和接收消息。

在图4-3的客户程序中，客户先创建一个套接字，然后给服务器6789端口发送一个消息，最后等待接收应答。*main*方法的参数是消息和服务器的DNS主机名。消息被转换成一个字节数组，DNS主机名被转换成一个因特网地址。图4-4给出了相应的服务器程序，服务器创建绑定到服务器6789端口的套接字，然后一直等待客户的请求消息，然后回发相应的消息做应答。

4.2.4 TCP流通信

TCP协议的API源于BSD 4.x UNIX，它提供了抽象的可读写的字节流。流抽象可隐藏网络的下列特征：

- 消息大小 应用程序可以选择写到流中和从流中读取的数据量的大小。它可处理非常小或非常大的数据集。TCP流的底层实现决定了在作为一个或多个IP数据包传送前要收集的数据量。数据到达后，实现根据应用的请求将数据传递给应用。如果必要，应用程序能强制数据马上发送。
- 丢失的消息 TCP协议使用确认机制。用一个简单的模式做例子（在TCP中并没有这样使用），发送端记录每个发送的IP数据包，接收端确认所有消息的到达。如果在一个超时时间段内，发送方没有接收到一个确认消息，发送方就重传该消息。更复杂的滑动窗口模式[Comer 1991]减少了所需的确认消息。
- 流控制 TCP协议试图匹配读写流的进程的速度。对读方来说，如果写方太快，那么它会被阻塞，直到消耗掉足够的数据。

```
import java.net.*;
import java.io.*;
public class UDPClient{
  public static void main(String args[]){
      // args give message contents and server hostname
     DatagramSocket aSocket = null;
      try {
          aSocket = new DatagramSocket();
          byte [] m = args[0].getBytes();
          InetAddress aHost = InetAddress.getByName(args[1]);
          int serverPort = 6789;
          DatagramPacket request =
                  new DatagramPacket(m,  args[0].length(), aHost, serverPort);
          aSocket.send(request);
          byte[] buffer = new byte[1000];
          DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
          aSocket.receive(reply);
          System.out.println("Reply: " + new String(reply.getData()));
       } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       } catch (IOException e){System.out.println("IO: " + e.getMessage());
       } finally { if(aSocket != null) aSocket.close();}
    }
}
```

图4-3 UDP客户发送一个消息到服务器并获得一个应答

```
import java.net.*;
import java.io.*;
public class UDPServer{
     public static void main(String args[]){
       DatagramSocket aSocket = null;
       try{
          aSocket = new DatagramSocket(6789);
          byte[] buffer = new byte[1000];
          while(true){
            DatagramPacket request = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(request);
            DatagramPacket reply = new DatagramPacket(request.getData(),
               request.getLength(), request.getAddress(), request.getPort());
            aSocket.send(reply);
          }
       } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       } catch (IOException e) {System.out.println("IO: " + e.getMessage());
       } finally {if(aSocket != null) aSocket.close();}
    }
}
```

图4-4 UDP服务器重复地接收请求并将它发回到客户

- *消息重复和排序* 每个IP数据包与消息标识符相关联，这使得接收方能检测和丢弃重复的消息，或重排没有以发送方顺序到达的消息。
- *消息目的地* 一对通信进程在它们能通信之前要先建立连接。一旦建立了连接，进程就可以不需要使用因特网地址和端口读写流。在发生通信前建立连接涉及客户向服务器发送一个*connect*请求，然后服务器给客户发送一个*accept*请求。对单个客户－服务器请求和应答而言，这是相当大的开销。

流通信的API假设，当一对进程建立连接时，其中的一个扮演客户角色，另一个扮演服务器角色，但过后它们又是平等的。客户角色涉及创建绑定到端口的流套接字，然后发出连接请求，在它的服务器端口上请求与服务器连接。服务器角色涉及创建绑定到服务器端口的监听套接字，然后等待客户的连接请求。监听套接字维护到达的连接请求队列。在套接字模型中，当服务器接受一个连接，就创建一个新的流套接字用于与客户的通信，同时保持在服务器端口上的套接字用于监听其他客户的*connect*请求。*connect*和*accept*操作的更多细节在本章最后的UNIX实例研究中描述。

客户和服务器的一对套接字由一对流相连，每个方向一个流。这样每个套接字有一个输入流和一个输出流。一对进程中的任一个进程可以通过将信息写入自己的输出流将信息发送给另一个进程，而另一个进程通过读取自己的输入流来获得信息。

132 ~ 134

一个应用关闭一个套接字意味着它不再写任何数据到它的输出流，在输出缓冲区中的任何数据被送到流的另一端，放在目的进程套接字的队列中，并指明流已经中断了。目的进程能读取队列中的数据，但在队列为空之后的任何读操作都会返回流结束标志。当进程退出或失败，它的所有套接字最终被关闭，任何试图与它通信的进程将发现连接已中断。

下面说明一些与流通信相关的未解决的问题。

- *数据项的匹配* 两个通信进程需要对在流上传送的数据内容有一致约定。例如，如果一个进程在流中先写入一个整型值，后跟写一个双精度值，那么另一端必须先读取一个整型值，后读取一个双精度值。当一对进程不能在流使用上正确协作时，在解释数据时读进程可能会出错，或者可能由于流中数据不足而阻塞在那里。
- *阻塞* 写入流的数据被保留在目的地套接字的队列中。当进程试图从输入通道读数据时，它将从队列中获得数据或一直阻塞直到有数据。如果在另一端的套接字队列中的数据与协议允许的数据一样多，那么将数据写到流的进程可能被TCP流控制机制阻塞。
- *线程* 当服务器允许连接，它通常创建一个新线程用于与新客户通信。为每个客户使用单独的线程的好处是服务器在等待输入时能阻塞而不延误其他客户。在不提供线程的环境中，另一种方法是在试图读取从一个流输入的数据之前测试该输入数据是否可用，例如，UNIX环境中的*select*系统调用就是用于这个目的。

故障模型 为了满足可靠通信的完整性，TCP流使用校验和检测并丢弃损坏的数据包，使用顺序号检测并丢弃重复的数据包。为保证有效性，TCP流使用超时和重传来处理丢失的数据包。因此，即使底层有些数据包丢失了，消息还是可以保证被传到目的地。

但是，如果连接上丢失的数据包超过了限制或连接一对通信进程的网络不稳定或严重拥塞时，那么负责发送消息的TCP软件将收不到确认，这种情况持续一段时间之后，TCP就会声明该连接已中断。这时TCP不能提供可靠通信，因为它不能保证在所有可能出现问题的情况都能正确地传送消息。

网络连接中断后，如果使用它的进程还试图读或写，就会接到有关的通知。连接中断会

产生以下影响：

- 使用连接的进程不能区分是网络故障还是连接另一端的进程出现了故障。 135
- 通信进程不能确定最近的消息是否被接收。

TCP的使用 许多常用的服务在TCP连接上运行，它们使用保留的端口号。这些服务包括：

- *HTTP* 超文本传送协议用于在Web浏览器和Web服务器之间通信；这部分见本章后面的讨论。
- *FTP* 文件传送协议允许浏览远程计算机上的目录，以及将文件通过连接从一台计算机传输到另一台计算机上。
- *Telnet* Telnet利用终端会话访问远程计算机。
- *SMTP* 简单邮件传送协议用于在计算机之间发送邮件。

TCP流的Java API TCP流的Java API由类*ServerSocket*和*Socket*给出。

- *ServerSocket* 该类用于在服务器端口上创建一个套接字以便监听客户的*connect*请求。它的*accept*方法从队列中获得一个*connect*请求，或者如果队列为空，它就阻塞直到有消息到达。执行*accept*的结果是一个*Socket*实例——可用于访问与客户通信的流套接字。
- *Socket* 该类用于连接的一对进程。客户使用构造函数（需指定服务器的DNS主机名和端口）创建套接字，该构造函数不仅创建与本地端口相关联的套接字，而且连接到指定的远程计算机和端口号。如果主机名出错它会抛出*UnknownHostException*，如果发生IO错误，它会抛出*IOException*。

*Socket*类提供了*getInputStream*和*getOutputStream*方法用于访问与套接字相关联的两个流。这些方法的返回类型分别是*InputStream*和*OutputStream*——定义了读写字节的抽象类。返回值用于对应输入输出流的构造函数参数。我们的例子使用了*DataInputStream*和*DataOutputStream*，它们允许以机器独立的方式读写简单数据类型的二进制表示。

图4-5给出了一个客户程序，其中*main*方法的参数提供了一个消息以及一个服务器的DNS

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
    // arguments supply message and hostname of destination
      Socket s = null;
      try{
        int serverPort = 7896;
        s = new Socket(args[1], serverPort);
        DataInputStream in = new DataInputStream( s.getInputStream());
        DataOutputStream out =
              new DataOutputStream( s.getOutputStream());
        out.writeUTF(args[0]);      // UTF is a string encoding see Sn 4.3
        String data = in.readUTF();
        System.out.println("Received: "+ data) ;
      }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
      } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
      } catch (IOException e){System.out.println("IO:"+e.getMessage());
      } finally {if(s!=null) try {s.close();}catch (IOException e){/*close failed*/}}
    }
}
```

图4-5 TCP客户与服务器建立连接，发送请求并接收应答

主机名。客户创建了一个绑定到主机名和服务器端口7896的套接字。*main*方法从套接字的输入和输出流生成*DataInputStream*和*DataOutputStream*，然后将消息写入它的输出流，并等待从它的输入流中读取应答。图4-6的服务器程序打开服务器端口（7896）的服务器套接字，监听*connect*请求。当有请求到达，就生成新线程用于与客户通信。新线程从它套接字的输入和输出流中创建*DataInputStream*和*DataOutputStream*，然后等待读取消息并回写。

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
      try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                  Socket clientSocket = listenSocket.accept();
                  Connection c = new Connection(clientSocket);
            }
      } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}
class Connection extends Thread {
      DataInputStream in;
      DataOutputStream out;
      Socket clientSocket;
      public Connection (Socket aClientSocket) {
         try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            this.start();
         } catch(IOException e)
               {System.out.println("Connection:"+e.getMessage());}
      }
      public void run(){
         try {                        // an echo server
            String data = in.readUTF();
            out.writeUTF(data);
         } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
         } catch(IOException e) {System.out.println("IO:"+e.getMessage());
         } finally { try {clientSocket.close();}catch (IOException e){/*close failed*/}}
      }
}
```

图4-6 TCP服务器为每个客户建立连接，然后回应客户的请求

因为我们的消息由串组成，客户和服务器进程使用*DataOutputStream*的*writeUTF*方法将它写入输出流，使用*DataInputStream*的*readUTF*方法从输入流中读取。UTF是表示串的特定格

式编码，见4.3节的描述。

若一个进程关闭了套接字，它将不能再使用输入流和输出流。接收数据的进程能从它的队列中读取数据，但在队列为空后的读操作会产生*EOFException*异常。试图使用一个关闭的套接字或向一个损坏的流中写数据，都会产生*IOException*异常。

4.3 外部数据表示和编码

在运行的程序中存储的信息都表示成数据结构——如相互关联的对象集合——但是消息中的信息由字节序列组成。不论使用何种通信形式，数据结构在传输前必须平整化（转换成字节序列），到达目的地后再进行重构。在消息中传送的单个基本数据项可以是多种不同类型的数据值，并不是所有的计算机都以同样的顺序存储诸如整数之类的基本类型值。浮点数的表示在不同的体系结构中也是不同的。另一个问题是用于表示字符的代码集：例如，UNIX系统使用ASCII字符编码，一个字符占一个字节，但是Unicode标准允许表示许多不同语言的文字，每个字符占两个字节。有两种不同的方法表示整数编码顺序：所谓的大序法排序，即最高有效字节排在最前面，和小序法排序，即最高有效字节排在最后面。

136 ~ 138

下列任一方法可用于两台计算机交换数据值：

- 值在传送前先转换成一致的外部格式，然后在接收端转换成本地格式；如果两台计算机是同一类型，可以不必转换成外部格式。
- 值以发送端的格式传送，同时传送所使用格式的标志，接收方根据需要转换该值。

注意，字节本身在传送过程中并不改变。为了支持RMI或RPC，任何能用作参数或返回值的数据类型必须能够平整化，单个基本数据值能以一致的格式表示。对数据结构和基本值的一致的表示标准称为*外部数据表示*。

*编码*是将多个数据项组装成适合消息传送的格式。*解码*是在消息到达后拆装消息，在目的地生成等价的数据项集合。这样，编码是将结构化数据项和基本值转换成一个外部数据表示。类似地，解码是从外部数据表示生成基本值，并重构数据结构。

我们将讨论两种外部数据表示和编码的方式：

- CORBA的公共数据表示，它涉及到在CORBA的远程方法调用中能作为参数和结果的结构化类型和基本类型的外部表示，它可用于多种编程语言中（见第17章）。
- Java对象的序列化，它涉及需要在消息中传送或存储到磁盘上的单个对象或对象树的平整化和外部数据表示。它仅用于Java。

在这两个例子中，编码和解码活动均由中间件层完成，不涉及任一方的应用程序员。因为编码过程要求考虑组成组合对象的基本组件的表示细节，所以手工完成该过程很容易出错。效率是设计自动生成型编码程序要考虑的另一个问题。

本节讨论的两种方法将基本数据类型编码成二进制形式。另一种方法是将所有要传送的对象编码成ASCII文本，相对来说这种方式易于实现，但编码的形式通常较长。4.4节描述的HTTP协议是后一种方法的例子。

虽然我们对使用RMI、RPC的参数和结果的编码感兴趣，但将数据结构或对象转换成一个适合消息传送或文件排序的格式更具通用意义。

139

4.3.1 CORBA的公共数据表示（CDR）

CORBA CDR是CORBA 2.0[OMG 1998a]定义的外部数据表示。CDR能表示所有在CORBA远程调用中用作参数和返回值的数据类型。它们由15个基本类型组成，其中包括*short*（16位）、*long*（32位）、*unsigned short*、*unsigned long*、*float*（32位）、*double*（64位）、*char*、*boolean*（TRUE，FALSE）、*octet*（8位）和*any*（它可表示任何基本的或构造类型），还有一套组合类型，如图4-7的描述。远程调用中每个参数或结果表示成调用消息或结果消息中的字节序列。

类　型	表　示
sequence	长度（unsigned long），后面是有序元素
string	长度（unsigned long），后面是有序字符（可以有宽字符）
array	有序的数组元素（不用指定长度，因为它是定长的）
struct	按成员声明的顺序表示
enumerated	无符号长整数（值由声明的顺序指定）
union	类型标志，后随选中的成员

图4-7　构造类型的CORBA CDR

- 基本类型　CDR定义了大序法排序和小序法排序的表示。值按发送端消息中排定的顺序传送，接收端如果要求不同的排序就要进行转换。例如，16位*short*占消息中的两个字节，若用大序法排序，最高有效位占用第一个字节，最低有效位占用第二个字节。每个基本类型值以它的大小为索引放在字节序列中。假设字节序列的索引从零开始，那么n字节大小（其中n=1，2，4或8）的基本类型值将追加到字节流序列中索引为n的倍数的位置。浮点值遵循IEEE标准——其中符号、指数和小数部分放在按大序法排序的0～n字节，按小序法排序是反过来放。字符用客户和服务器均认可的代码集表示。
- 构造类型　组成每个构造类型的基本类型值按特定的顺序（如图4-7所示）加到字节序列中。

图4-8给出了CORBA CDR表示的一个*struct*消息，该结构包含3个域，3个域各自的类型分别是*string*、*string*和*unsigned long*。图给出了每行4字节的字节序列。每个串用一个*unsigned long*表示长度，后面紧跟串中的字符。为简单起见，我们假设每个字符仅占一个字节。变长数据用零填充，以便形成标准格式，用以比较编码数据或它的校验和。注意每个*unsigned long*占4个字节，其

140 开始位置在一个4的倍数的索引处。图4-8没有区分大序法排序和小序法排序。虽然图4-8中是一个简单的例子，但CORBA CDR能表示任何不使用指针的由基本类型和构造类型组成的数据结构。

字节序列的索引 ← 4字节 →	表示法说明
0–3	5
4–7	"Smit"
8–11	"h___"
12–15	6
16–19	"Lond"
20–23	"on__"
24–27	1934

串 '*Smith*' 的长度

串 '*London*' 的长度

unsigned long

平整化的格式表示*Person* struct具有值 {'Smith'，'London'，1934}

图4-8　CORBA CDR 消息

外部数据表示的另一个例子是Sun XDR标准，该标准在RFC 1832中说明[Srinivasan 1995b]，其描述见www.cdk3.net/ipc。它由Sun开发，用于Sun NFS中客户和服务器之间的消息交换（见第8章）。

CORBA CDR或Sun XDR标准均没有在消息的数据表示中给出数据项类型。这是因为它假定发送方和接收方对消息中数据项的类型和顺序有公共的约定。特别是对RMI或RPC，每个方法调用传递特定类型的参数，而结果也是特定类型的值。

CORBA中的编码 根据消息中要传送的数据项的类型规范能自动生成编码操作。数据结构的类型和基本数据项类型用CORBA IDL描述（见17.2.3节），IDL提供了描述RMI方法的参数类型和结构类型的符号表示法。例如，可用CORBA IDL描述图4-8消息的数据结构：

```
struct Person{
    string name;
    string place;
    long year;
};
```

CORBA接口编译器（见第5章）根据远程方法的参数类型和结果类型的定义为远程方法的参数和结果生成适当的编码操作和解码操作。 141

4.3.2 Java 对象序列化

在Java RMI中，对象和基本数据值都可以作为方法调用的参数和结果传递。一个对象是一个Java 类的实例。例如，等价于CORBA IDL定义的*Person struct*的Java类是：

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
    // followed by methods for accessing the instance variables
}
```

上面的类申明它实现了*Serializable*接口，该接口没有方法。申明一个类实现了*Serializable*接口（该接口在*java.io*包中提供）意味着它的实例能序列化。

在Java中，术语序列化指的是将一个对象或一组有关联的对象平整化成适合于磁盘存储或消息传送的串行格式，例如像RMI中的参数或结果。解序列化是指从串行格式中恢复对象或一组对象。假设进行解序列化的过程预先不知道序列化格式中对象的类型，因此，关于每个对象类的一些信息要包含在序列化格式中。这些信息使得接收方在解序列化对象时能装载恰当的类。

类信息由类名和一个版本号组成。当类有大的改动时要改动版本号。它可由程序员设置

或根据类、类的实例变量、方法和接口的名字的散列值自动计算。解序列化对象的过程能检查对象的类的版本是否正确。

Java对象能包含对其他对象的引用。当对象序列化时，所有它引用的对象也随它一起序列化，以确保对象在目的地重构时能恢复所有它引用的对象。引用被序列化成句柄——在这种情况下，句柄是在序列化格式内对一个对象的引用。例如句柄可以是在正整数序列中的下一个数字。序列化过程必须确保在对象引用和句柄之间是一一对应的。它也必须确保每个对象只被写一次——在对象的第二次或后继出现时，写句柄而不是写对象。

为了序列化一个对象，要写出它的类信息，后面跟实例变量的类型和名字。如果实例变量属于新的类，那么一定要先写出它们所属的类信息，然后再写那些实例变量的类型和名字。这个递归过程一直进行到所有必须的类的类信息和实例变量的类型和名字都被写出。每个类
142 有一个句柄，没有一个类会多次写入字节流——在需要的地方写入句柄。

整形、字符、布尔、字节和长型等基本类型的实例变量的内容用*ObjectOutputStream*类的方法写成一个可移植的二进制格式。对字符串和字符用*writeUTF*方法，该方法使用通用传输格式（UTF），UTF用一个字节表示ASCII字符，而用多个字节表示Unicode字符。字符串前面是串占据的字节数。

作为一个例子，考虑下列对象的序列化：

Person p=new Person("Smith","London",1934);

序列化格式如图4-9所示，图上省略了句柄的值和表示对象、类、串和其他对象的类型标记。第一个实例变量（1934）是有固定长度的整数；第二个和第三个实例变量是串，在串的前面是它们的长度。

序列化值				解释
Person	8字节的版本号		h0	类名、版本号
3	int year	java.lang.String name:	java.lang.String place:	实例变量的个数、类型和名字
1934	5 Smith	6 London	h1	实例变量的值

真正的序列化格式包含额外的类型标记；h0和h1是句柄

图4-9 Java的序列化格式表示

为了利用Java序列化对*Person*对象序列化，要创建类*ObjectOutputStream*的实例，并以*Person*对象为参数调用它的*writeObject*方法。为了从数据流中解序列化一个对象，要在流上打开一个*ObjectInputStream*，用它的*readObject*方法重构原来的对象。这一对类的使用类似图4-5和图4-6中说明的*DataOutputStream*和*DataInputStream*。

远程调用的参数和结果的序列化和解序列化通常由中间件自动完成，不需要应用程序员的参与。如果有特殊需求，程序员可以编写自己的读写对象的方法。如何编写自己的读写对象方法以及关于Java序列化的更多信息请阅读有关对象序列化的手册[java.sun.com II]。另一种程序员修改序列化效果的方法是将不应该被序列化的变量声明为*transient*。对本地资源如文件、套接字的引用就是不应该被序列化的变量的例子。

反射的使用 Java语言支持反射——查询类属性（如类实例变量和方法的名字和类型）的能力。反射使能根据类名创建类，以及为一个给定的类创建具有给定参数类型的构造函数。反射使以完全通用方式进行序列化和解序列化成为可能。这意味着没有必要像CORBA中一样 143
为每种对象类型生成特殊的编码函数。更多的关于反射的信息请参见Flanagan[1997]。

Java对象序列化使用反射找到要序列化的对象的类名以及类实例变量的名字、类型和值。这是序列化格式所需的全部信息。

对解序列化而言，序列化格式中的类名用于创建类。这个类又用于创建一个新的构造函数，它具有与序列化格式相对应的类型。最后，新的构造函数用于创建新的对象，其实例变量的值是从序列化格式中读取的。

4.3.3 远程对象引用

客户调用远程对象中的一个方法，就会向存放远程对象的服务器进程发送一个调用消息。这个消息需要指定哪一个特定的对象具有要调用的方法。*远程对象引用*是远程对象的标识，它在整个分布式系统中有效。远程对象引用用于在调用消息中指定调用哪一个对象。第5章说明了远程对象引用也被用在远程方法调用的参数和结果中，第5章还说明了每个远程对象有一个远程对象引用，以及能通过比较远程对象引用来确定它们是否指向同一个远程对象。现在讨论远程对象引用的外部表示。

远程对象引用必须按确保空间和时间惟一性的方法生成。通常，在远程对象上有许多进程，所以远程对象引用必须在分布式系统的所有进程中惟一。即使在与给定远程对象引用相关的远程对象删除后，该远程对象引用也不能被重用，因为可能有潜在的调用者还保留着过期的远程对象引用。对于任何调用已删除对象的企图应该产生一个出错信息而不应该允许其访问另一个对象。

可用几种方法确保远程对象引用是惟一的。一种方法是通过拼接计算机的因特网地址、创建远程对象引用的进程的端口号、创建时间和本地对象编号来构造远程对象引用。每次进程创建一个对象，本地对象编号就相应加一。

端口号加时间在计算机上产生惟一的进程标识。用这种方法，远程对象引用可用图4-10中的格式表示。在RMI的最简单实现中，远程对象仅在创建它们的进程中存在，并只在该进程运行时存活。在这种情况下，远程对象引用能被用做远程对象的地址。换句话说，调用的消息被发送到远程引用中的因特网地址，并传递给该计算机上使用给定端口号的进程。

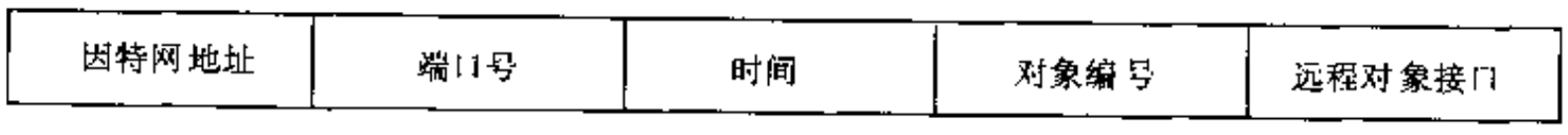


图4-10 远程对象引用的表示

为了允许远程对象在不同计算机的不同进程中被重定位，远程对象引用不应该被用做远程对象的地址。17.2.4节讨论了远程对象引用的一种格式，它允许对象在生存期中在不同的服务器上被激活。

图4-10中远程对象引用的最后一个字段包含了关于远程对象接口的信息，例如，接口名。该信息与将远程对象引用接收为远程调用的参数或结果的进程有关，因为它需要知道由远程

对象提供的方法。这一点将在5.2.5节解释。

4.4 客户－服务器通信

设计这种形式的通信是用于支持在典型客户－服务器交互中角色和消息的交换。在正常的情况下，请求－应答通信是同步的，因为客户进程将一直阻塞到服务器发出的应答消息到达本地。它也是可靠的，因为服务器发出的应答是对客户的一个有效确认。异步请求－应答通信是另一种方法，适用于客户可能在稍后取回应答的情形——参见6.5.2节。

虽然当前许多实现使用了TCP流，但下面用Java的UDP数据报API中的*send*和*receive*操作描述客户－服务器信息交换。在数据报上构造的协议避免了与TCP流协议相关的不必要的开销，特别是：

- 确认消息是冗余的，因为应答紧跟着请求。
- 除了进行请求和应答的一对连接之外，需要建立包含两对额外消息的连接。
- 对大多数仅传递少量参数和结果的调用来说，流控制是冗余的。

请求－应答协议 下列协议基于3个通信原语：*doOperation*、*getRequest*、*sendReply*，如图4-11所示。大多数的RMI和RPC系统期望由类似的协议支持。此处描述的协议经过裁减可支持RMI，因为针对请求消息中调用的方法，该协议可为该方法所属的对象传递一个远程对象引用。

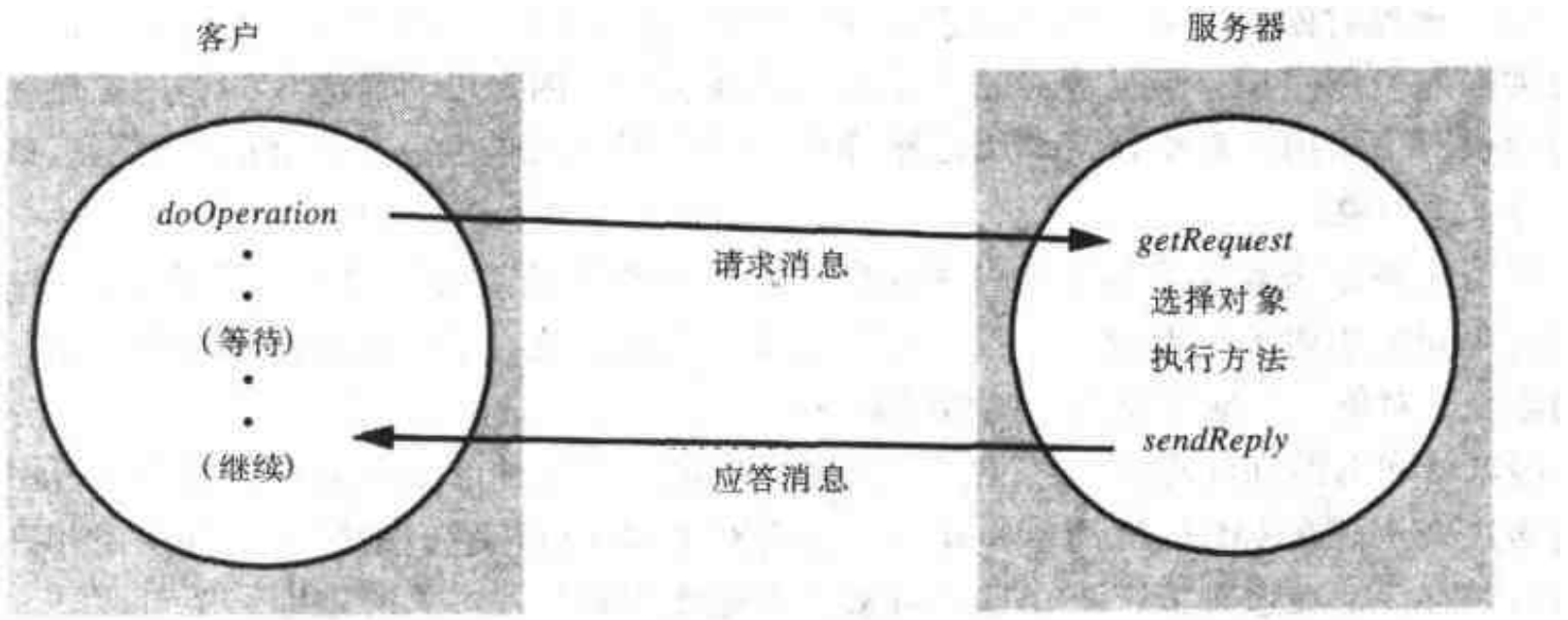


图4-11 请求－应答通信

这种特殊设计的请求－应答协议能够匹配请求和应答。它可设计成提供某种传递保证。如果使用UDP数据报，传递保证必须由请求－应答协议提供，即可以使用服务器应答消息作为客户请求消息的确认。图4-12概述了这3个通信原语。

客户使用*doOperation*操作调用远程操作。它的参数指定了要调用的远程对象和方法、以及该方法要求的额外的信息（参数）。其结果是一个RMI应答。它假设调用*doOperation*的客户将参数编码成一个字节数组，然后从返回的字节数组中将结果进行解码。*doOperation*的第一个参数是类*RemoteObjectRef*的实例，它可以用图4-10所示的格式表示远程对象引用。该类提供的方法可获得远程对象所在的服务器的因特网地址和端口。*doOperation*方法发送一个请求消息到服务器，服务器的因特网地址和端口由参数中的远程对象引用指定，在发送请求消息之后，*doOperation*调用*receive*获得一个应答消息，并从中抽取结果返回给调用者。*doOperation*

的调用者将一直阻塞直到服务器上的远程对象完成所请求的操作，然后将应答消息传递给客户进程。

*getRequest*用于服务器进程获取服务请求，如图4-11所示。当服务器调用某指定对象的方法时，它使用*sendReply*发送应答消息给客户。当客户接收到应答消息时，原来的*doOperation*解除阻塞、继续执行客户程序。

public byte[] doOperation(RemoteObjectRef o, int methodId, byte[] arguments)
发送请求消息到远程对象并返回应答。参数指定了远程对象、要调用的方法和该方法的参数。
public byte[] getRequest();
通过服务器端口获得客户请求。
public void sendReply(byte[] reply, InetAddress clientHost, int clientPort);
发送应答消息*reply*到该因特网地址和端口上的客户。

144～146

图4-12 请求－应答协议的操作

在请求消息或应答消息中被传送的信息如图4-13所示。第一个域指示消息是一个请求消息还是一个应答消息。第二个域请求Id包含一个消息标识。客户的*doOperation*为每个请求消息生成一个请求Id，服务器将它们拷贝到相应的应答消息。这使得*doOperation*能检查应答消息是当前请求的结果还是一个延迟到达的以前的请求结果。第三个域是编码成图4-10所示的远程对象引用。第四个域是要调用的方法的标识，例如，在接口中的方法可以编码成1、2、3…。如果客户和服务器均使用支持反射的公共语言，那么该方法的表示可以放在这个域中——在Java中方法的实例可以放在该域。

消息类型	*int*（0=请求，1=应答）
请求Id	*int*
对象引用	*RemoteObjectRef*
方法Id	*int* 或*Method*
参数	//字节数组

图4-13 请求－应答消息的结构

消息标识 任何涉及消息管理、用以提供额外的诸如可靠消息传递或请求－应答通信特性的机制均要求每个消息具有惟一的消息标识，供消息被引用。消息标识由两部分组成：

1. 请求Id，由发送进程根据一个增长的整数序列设置。
2. 发送进程的一个标识，例如它的因特网地址和端口。

第一部分使得标识在发送方惟一，第二部分使得标识在分布式系统中惟一。（第二部分能独立获得。例如，如果使用UDP，就可以从接收到的消息中获得）。

当请求Id的值到达无符号整数的最大值（例如，$2^{32}-1$）时，就重置为0。这里仅有的限制是消息标识的生存期应该远远小于用尽整数序列值的时间。

请求–应答协议的故障模型 如果3个原语*doOperation*、*getRequest*和*sendReply*在UDP数据报上实现，那么它们会出现同样的通信故障，即：

- 存在遗漏故障
- 不能保证消息按发送方顺序到达

147

此外，协议还要承受进程故障（见2.3.2节）。我们假设进程有崩溃故障，就是说，一旦进程挂起，它们就一直挂起——它们不产生拜占庭行为。

考虑到服务器故障、丢失请求消息或丢失应答消息的情况，*doOperation*在等待服务器应答消息的时候使用超时。超时发生时采取的行为取决于要提供的传递保证。

超时　超时后*doOperation*的行为有不同的选项。最简单的是马上从*doOperation*返回，并给客户一个标志表示*doOperation*操作失败。这不是通常使用的方法——超时可能是由于请求或应答消息丢失——在后一种情况，操作已经完成。为了补偿可能的丢失消息，*doOperation*重复地发送请求消息直到它获得一个应答或者它能确保延迟是由于服务器没有反应而不是消息丢失为止。最后，当*doOperation*返回时，它通知客户出现了一个异常“没有接收到结果”。

丢弃重复的请求消息　万一请求消息被重传了，服务器就可能多次收到该消息。例如，服务器可能收到第一个请求消息，但它执行命令并将结果返回所花的时间超过了客户的超时时限，这就会导致服务器对同一请求执行多次操作。为了避免这种情况，协议的设计要能识别具有相同请求标识（来自同一客户的）的后续消息，过滤掉重复的消息。如果服务器还没有发送应答，也不需要采取特殊的动作——服务器在完成操作的执行后会传递应答。

丢失应答消息　如果服务器在收到一个重复的请求时它已发送了应答，那么它将需要再次执行操作以获得结果，除非它将原来执行的结果保存起来了。有些服务器能多次执行操作，每次都能获得相同的结果。幂等操作是指能重复执行的操作，每次执行的结果都一样。例如，将一个元素加到一个集合的操作是一个幂等操作，因为每次执行对集合的效果都是一样的。但是，将一数据项追加到一个序列就不是一个幂等操作，因为每次它将扩展序列。若服务器上的操作都是幂等的，那么服务器就不用采取特殊的手段避免多次执行它的操作。

历史　对那些要求不重新执行操作而重传应答的服务器，可以使用历史。术语“历史”用于指包含已经传送的（应答）消息记录的结构。历史中的项包含一个请求标识、消息以及消息发送的目的客户标识。它的作用是允许服务器在客户进程发请求时重传应答消息。与历史使用相关的问题是它的内存开销。除非服务器能够看出什么时候消息不再需要重传，否则历史将变得很大。

因为客户每次只能发送一个请求，服务器便将每个请求解释成客户对前一个应答的确认。
148 因此，历史只需要包含发送给每个客户的前一个应答消息。然而服务器历史中的应答消息量在服务器具有大量客户时会成为一个问题，特别是，当一个客户进程终止，它没有确认它已接收到上一个应答时——因此历史中的消息在过了一段时间后将被正常丢弃。

RPC交换协议　三种具有不同通信故障语义的协议用于实现不同类型的RPC。它们最初由Spector[1982]确定：

- 请求（R）协议。
- 请求-应答（RR）协议。
- 请求-应答-确认应答（RRA）协议。

图4-14总结了在这些协议中传递的消息。R协议用于没有数据从过程中返回而且客户不要求对过程的执行进行确认的情况。客户可以在发送请求消息之后马上继续进行其他动作，因为它不需要等待应答消息。RR协议用于大多数客户-服务器交互，因为它基于请求-应答协议。这一协议不需要特殊的确认消息，因为服务器的应答消息被认为是对客户请求消息的确认。类似地，客户的后续调用可以被认为是对服务器应答消息的确认。

名字	消息发送方		
	客户	服务器	客户
R	请求		
RR	请求	应答	
RRA	请求	应答	确认应答

图4-14 RPC交换协议

RRA协议基于3种消息的交换：请求－应答－确认应答。确认应答消息包含被确认应答消息的请求Id。这使得服务器能丢弃历史中的数据项。确认消息中的请求Id的到达将被解释成对所有小于该请求Id的应答消息的接收进行确认，所以确认消息的丢失是无害的。虽然数据交换包含了一个额外的消息，但它不需要阻塞客户，因为确认在应答到达客户之后传递。但是这一协议确实耗费处理器资源和网络资源。练习4.22建议了一种对RRA协议的优化。

使用TCP流实现请求–应答协议 数据报一节提到通常决定接收数据报用的缓冲区的合适大小是很困难的。在请求－应答协议中，服务器接收请求的缓冲区和客户接收应答的缓冲区 149
也涉及到类似的问题。数据报的有效长度（通常是8KB）并不被认为是使用透明RMI系统的合适长度，因为过程的参数或结果可以是任何大小的。

避免实现多包协议是选择在TCP流上实现请求－应答协议的理由之一，因为TCP流允许传输任意大小的参数和结果。特别地，Java对象序列化是一个流协议，它允许参数和结果通过流在客户和服务器之间传送，并且可能可靠地传递任意大小的对象集合。如果使用TCP协议，它确保请求消息和应答消息能可靠地传递，所以没有必要让请求－应答协议处理消息重传和重复消息的过滤或使用历史。另外，流控制机制允许大的参数和结果不必为避免接收方的崩溃采用特殊的手段进行传输。因此选择TCP协议实现请求－应答协议，是因为它能简化协议的实现。如果同一对客户－服务器之间的后继请求和应答在同一个流上发送，那么不需要在每个远程调用上都有连接开销。当一个应答消息跟随在请求消息之后时，确认消息的开销也将减少。

有时，应用并不需要由TCP提供的所有功能，一个更有效的经过特别裁减的协议可在UDP上实现。例如，前面章节提到的，Sun NFS并不需要大小不限的消息，因为它在客户和服务器之间传输固定长度的块。除此之外，它的操作是幂等的，这样，为了重传丢失的应答消息，即使多次执行操作也没有关系，同时这也使得它没有必要维护历史。

HTTP：请求－应答协议举例 第1章介绍了超文本传输协议（HTTP），Web浏览器客户使用这一协议向Web服务器发送请求并从服务器接收应答。简要地说，Web服务器管理资源有以下几种实现方式：

- 以数据实现的资源，例如，HTML页面的正文，一个图像或小程序的类。
- 以程序实现的资源，例如，能在Web服务器上运行的cgi程序和servlet（见[java.sun.com III]）。

客户请求指定了一个URL，它包含了Web服务器的DNS主机名和Web服务器上的一个可选的端口号以及该服务器上一个资源的标识。

HTTP协议指定了请求－应答交互所需的消息、方法、参数和结果以及它们在消息中的表示（编码）规则。它还支持一个固定的可用于所有资源的方法集（*GET*、*PUT*、*POST*等等）。HTTP协议不像前面提到的协议，在那些协议中每个对象有自己的方法。除了在Web资源上调 150
用方法之外，HTTP协议允许内容协商和口令形式的认证。

- 内容协商　客户请求中包含了可以接收何种数据表示之类的信息（例如语言或介质类型），它使得服务器能选择最适合用户的数据表示。
- 认证　证书和询问用于支持口令形式的认证。在第一次试图访问受口令保护的区域时，服务器的应答包含了该资源所需的质询。第7章解释了如何进行质询。当客户接收到一个质询，它让用户输入名字和口令，并在后继的请求中提交相关的证书。HTTP是在TCP上实现的，在该协议的原始版本中，每个客户－服务器交互由下列步骤组成：
  - 客户请求一个连接，服务器在默认的服务器端口或在URL指定的端口上建立连接。
  - 客户发送请求消息到服务器。
  - 服务器发送应答消息到客户。
  - 关闭连接。

然而，为每个请求－应答交互建立和关闭连接开销太大，它不仅会使服务器过载而且也在网络上发送了太多的消息。考虑到浏览器通常会对同一个服务器发多个请求，HTTP协议的一个较新版本（HTTP 1.1；见RFC 2616[Fielding *et al.* 1999]）使用了*永久连接*——一个在客户和服务器之间的一系列请求－应答交互上一直打开的连接。可由客户或服务器在任何时候通过发送一个标志给对方来关闭永久连接。服务器会关闭空闲了一段时间的永久连接。客户在发送请求中间，有可能会从服务器收到一个消息说连接关闭了，这时，假设相关的操作是幂等的，浏览器将不需要用户介入就会重发请求。例如，下面描述的*GET*方法是幂等的。当涉及到非幂等操作时，浏览器将质询用户下一步做什么。

请求和应答编码成ASCII文本串放入消息，但资源可以表示成字节序列，还可能经过了压缩。在外部数据表示中使用文本简化了直接与协议打交道的应用程序员对HTTP的使用。在此处，文本表示对消息的长度增加不多。

以数据形式实现的资源在参数和结果中具有MIME式的结构。多用途因特网邮件扩展（MIME）是在电子邮件中发送包含文本、图像、声音等多部分数据的标准。数据用*Mime*类型做前缀，这样接收方将知道如何处理它。一个Mime类型指定了一个类型和一个子类型。例如，*text/plain*、*text/html*、*image/gif*、*image/jpeg*。客户也能指定它们愿意接收的Mime类型。

HTTP方法　每个客户请求指定了一个要应用到服务器上的资源的方法和该资源的URL。
151 应答报告了请求的状态。请求和应答可能还包含资源数据、一个表单的内容或在Web服务器上运行的一个程序资源的输出。有下列方法：

- *GET*　该方法请求一个资源，该资源的URL以参数方式给出。如果URL指向数据，那么Web服务器返回由URL指向的数据。如果URL指向程序，那么Web服务器运行该程序，将程序结果返回给客户。参数可加入到URL，例如，*GET*能将一个表单的内容作为参数给一个*cgi*程序。可以根据资源上一次被修改的日期选择是否进行*GET*操作。*GET*操作也能配置成获取部分数据。
- *HEAD*　该请求与*GET*等同，但它不返回任何数据。它返回所有与数据有关的信息，如上次修改时间、数据类型或大小。
- *POST*　该方法指定一个资源（例如一个程序）的URL，该资源能处理请求中提供的数据。对数据进行的处理依赖于在URL中指定的程序的功能。该方法用于：
  - 为诸如servlet或cgi程序的数据处理过程提供数据块（例如一个表单）
  - 将消息放到公告牌、邮件列表或新闻组

• 用追加操作扩展数据库

• *PUT* 该方法将请求中的数据作为资源（或是对已有资源的修改，或是一个新资源）存储起来并用给定的URL作为其标识。

• *DELETE* 服务器删除由给定URL标识的资源。服务器可能永远都不允许这个操作，在这种情况下，将返回请求失败的应答。

• *OPTIONS* 服务器提供给客户可在给定URL上应用的方法列表（例如，*GET*、*HEAD*、*PUT*）以及服务器的特殊需求。

• *TRACE* 服务器发回请求消息，用于诊断。

上述请求可由一个代理服务器截获（见2.2.2节）。对*GET*和*HEAD*的响应可由代理服务器缓存。

消息内容 请求消息指定了一个方法名、一个资源的URL、协议版本、若干消息头和一个可选的消息体。图4-15给出了一个HTTP 请求消息（方法为*GET*）的内容，当URL指定一个数据资源时，*GET*方法没有消息体。代理服务器要求图中所示的完整的URL，但对于一个初始服务器可以只发送路径名。

方法	URL或路径名	HTTP版本	头	消息体
GET	http://www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

图4-15 HTTP 请求消息

152

头域包含请求的修改者和客户信息，例如最近一次修改资源的条件或可接受的内容类型（例如，HTML文本、音频或JPEG）。授权域用于提供客户的证书，以表明它们访问资源的权力。

应答消息指定了协议版本、状态码、“理由”、若干消息头和一个可选的消息体，如图4-16所示。状态码和理由在请求成功时提供一个报告，否则前者是一个由程序解释的3位整数，后者是一个可以理解的文本词组。头域用于传递有关服务器或有关资源访问的额外信息。例如，如果请求要求认证，那么应答的状态码将给出相应指示，并且在头域包含了一个质询。一些返回的状态有十分复杂的结果。特别是，303状态告诉浏览器查看另一个URL，该URL在应答的一个头域中。当程序需要将浏览器重定向到一个选中的资源时，303状态可用于由*POST*请求激活的这个程序的应答中。

HTTP版本	状态码	理由	头	消息体
HTTP/1.1	200	OK		资源数据

图4-16 HTTP 应答消息

请求或应答消息中的消息体包含了与请求中指定的URL相关的数据。消息体有自己的指定了有关数据信息的头部，如消息体的长度、Mime类型、字符集、内容编码和上一次修改时间。Mime类型域指定了数据的类型，例如*image/jpeg*或*text/plain*。内容编码域指定了所使用的压缩算法。

4.5 组通信

如果一个服务或是为了提供容错能力或是为了提高可用性而实现成在多个不同计算机上的多个不同的进程，那么就会有一个进程到一组进程的通信。消息成对交换不是一个进程到

一组进程通信的最佳模式。组播是更合适的方式——这是一种将单个消息从一个进程发送到一组进程的每个成员的操作，通常，组的成员对发送方是透明的。组播的行为有很多可能情
153 况。最简单的组播不提供消息传递保证和排序保证。

组播的消息提供了构造具有下列特征的分布式系统的有用的基础结构：

1. 基于复制服务的容错：一个复制的服务由一组服务器组成。客户请求被组播到组的所有成员，每一个都完成相同的操作。即使一些成员出了故障，仍能为客户提供服务。
2. 在自发网络中找到发现服务器：2.2.3节介绍了自发网络中的发现服务。客户和服务器能使用组播消息定位可用的发现服务以便在分布式系统中注册服务接口或查找其他服务的接口。
3. 通过复制的数据获得更好的性能：数据复制能增加服务的性能——在某些情况下，数据的副本放在用户的计算机上。每次数据改变，新的值被组播到管理副本数据的各个进程。
4. 事件通知的传播：组播到一个组可用于在发生某些事情时通知有关进程。例如，当一个新闻消息被贴到某个特定的新闻组时，新闻系统可通知感兴趣的用户。Jini系统在新的查询服务发布其存在时用组播通知感兴趣的客户。

我们先介绍IP组播，然后回顾上述使用组通信的要求，看IP组播能满足它们中的哪些要求。对那些不能满足的要求，我们在组通信协议中提出了更进一步的IP组播之外的特征。

4.5.1 IP组播——组通信的实现

本节讨论IP组播，通过*MulticastSocket*类给出组播的Java API。

IP组播 IP组播建立在因特网协议（IP）的上层。注意IP数据包是面向计算机的——端口属于TCP和UDP层。IP的组播允许发送方将单个IP数据包传送给组成一个组播组的计算机集合。发送方不清楚单个接收方身份和组的大小。组播组由D类因特网地址（如图3-15所示）指定——就是说，在IPv4中，头4位是1110的地址。

一个组播组的成员允许计算机接收发送给组的IP数据包。组播组的成员是动态的，即允许计算机在任何时间加入或离开，也允许计算机加入任意数量的组。无需成为组成员就可以将数据报发送到一个组播组。

在应用程序开发层，IP组播仅通过UDP可用。应用程序通过组播地址和普通的端口号发送UDP数据报完成组播，通过将其套接字加入到组来加入一个组播组以便能从组接收信息。在IP层，当一台计算机的一个或多个进程具有属于一个组播组的套接字时，该计算机属于这
154 个组播组。当一个组播消息到达一个计算机，消息副本被传递到所有已经加入到指定组播地址和指定端口号的本地套接字上。下列细节是IPv4特有的：

- 组播路由器 IP数据包既能在局域网上也能在因特网上组播。本地的组播使用了局域网的组播能力，例如以太网的组播能力。因特网上的组播利用了组播路由器，由它将单个数据报转发给其他成员所在网络的路由器上，在那里又组播到本地成员。为了限制组播数据报传播的距离，发送方能指定允许通过的路由器数量——称为存活时间，简称为TTL。要理解路由器如何知道其他哪一个路由器具有一个组播组的成员，请参见Comer[1995]。
- 组播地址分配 组播地址可以是永久的也可以是临时的。永久组甚至可以在没有组员的

情况下存在——因特网管理当局将它们的地址设成224.0.0.1到224.0.0.255。例如，第一个地址指向所有组播主机。

剩下的组播地址可用于临时组，这些组必须在使用前创建，在所有成员离开的时候停止存在。一个临时组创建时，它要求一个空闲的组播地址以避免加入到一个已有组中，IP组播协议没有解决这个问题。但当它的用户仅需要本地通信时，可以把TTL设成一个小的值，使得它不可能与其他组选择同一个地址。然而，在因特网上使用IP组播的程序需要解决这个问题。会话目录（sd）程序用于启动或加入一个组播会话[mice.ed.ac.uk, session directory]。它提供了一个具有交互界面的工具，允许人们浏览已发布的组播会话，或发布它们自己的会话，可指定时间和持续时段——它为每个新的会话选择一个组播地址。

组播数据报的故障模型 在IP组播上的数据报与UDP数据报有相同的故障特征——就是说，它们也会遭遇遗漏故障。如果组播中遇到该故障，那么即使只有一个遗漏故障，消息也不能保证传递到一个特定组的所有成员。就是说，有一些但不是所有组成员能接收到消息，这被称为不可靠的组播，因为它不能保证消息传递到组的任何一个成员。可靠的组播见第11章的讨论。

IP组播的Java API Java API通过类*MulticastSocket*提供IP组播的数据报接口，类*MulticastSocket*是*DatagramSocket*的子类，具有加入组播组的能力。类*MulticastSocket*提供了两个构造函数，允许用一个指定的本地端口（例如，如图4-17所示的6789）或任何空闲的本地端口创建套接字。一个进程可调用它的组播套接字的*joinGroup*方法用一个给定的组播地址加入到一个组播组。实际上，套接字在给定端口加入到一个组播组，它将接收其他计算机上的进程发送到这个端口上的给这个组的数据报。进程通过调用它的组播套接字的*leaveGroup*方法离开指定的组。

155

在图4-17的例子中，*main*方法的参数指定了要组播的消息和组的组播地址（例如，"*228.5.6.7*"）。在加入到组播组后，进程生成包含消息的*DatagramPacket*实例，并通过组播套接字将该消息发送到端口6789上的组播组地址。这以后，它试图通过它的套接字从其他同属同一端口上的组成员处接收3个组播消息。当该程序的几个实例同时在不同的计算机上运行时，它们都加入到同一个组，它们中的每一个接收自己的消息和来自同一组的消息。

Java API允许通过*setTimeToLive*方法对组播套接字设置TTL。默认值是1，这一设置允许组播仅在局域网中传播。

在IP组播上实现的应用可以使用多个端口。例如，MultiTalk[mbone]应用允许用户组保持基于文本的对话，它用一个端口发送和接收数据，用另一个端口交换控制数据。

4.5.2 组播的可靠性和排序

上一节叙述了IP组播的故障模型。也就是说，它会遭遇遗漏故障。对局域网的组播而言，它利用网络的组播能力让单个数据报到达多个接收者，但任何一个接收者都可能因为它的缓冲区满而丢弃消息。从一个组播路由器发送到另一个路由器的数据报也可能丢失，这妨碍了通过路由器到达的接收者接收消息。

另一个因素是任何进程可能失败。如果组播路由器出了故障，那么通过路由器到达的组成员将不能接收到组播消息，虽然本地成员还可以接收。

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
     // args give message contents & destination multicast group (e.g. "228.5.6.7")
    MulticastSocket s =null;
      try {
        InetAddress group = InetAddress.getByName(args[1]);
        s = new MulticastSocket(6789);
        s.joinGroup(group);
        byte [] m = args[0].getBytes();
        DatagramPacket messageOut =
            new DatagramPacket(m, m.length, group, 6789);
        s.send(messageOut);
        byte[] buffer = new byte[1000];
        for(int i=0; i< 3; i++) {   // get messages from others in group
            DatagramPacket messageIn =
                  new DatagramPacket(buffer, buffer.length);
            s.receive(messageIn);
            System.out.println("Received:" + new String(messageIn.getData()));
        }
        s.leaveGroup(group);
      } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
      } catch (IOException e){System.out.println("IO: " + e.getMessage());
      } finally { if(s != null) s.close();}
    }
}
```

图4-17 组播成员加入一个组，然后发送和接收数据报

排序是另一个问题。在互联网上发送的IP数据包不必按发送顺序到达，这样，同属一组的一些成员从同一个发送者接收的数据报的顺序可能与其他一些成员不一样。另外，由两个不同的进程发送的消息不必以相同的顺序到达组的所有成员。

可靠性和排序的作用举例 我们现在就4.5节介绍的4个使用复制的例子来考虑IP组播故障语义的作用。

1. 基于复制服务的容错：考虑这样一个复制的服务，它由一组服务器组成，这些服务器以相同的初始状态启动，总是按相同的顺序完成相同的操作，以便维持相互之间的一致性。这个组播应用要求，或者所有的副本或者没有副本接收到一个操作请求——如果有一个错过了该请求，那么它将变得与其他的不一致。这就要求在大多数情况下所有成员按相同的顺序接收请求消息。
2. 在自发网络中找到发现服务器：假设想定位发现服务器的进程在它启动后周期性地发送组播请求，那么在定位发现服务器时偶尔地丢失请求不是一个问题。事实上，Jini在它的组播请求协议上使用IP组播寻找发现服务器。Arnold *et al.*[1999]等人描述了这个问题。
3. 通过复制的数据获得更好的性能：考虑利用组播消息分布复制数据本身而不是数据上操作，消息丢失和不一致排序的影响取决于复制的方法和所有最新副本的重要性。例

如，新闻组的复制不必在任何时候都相互一致——消息甚至可以以不同的顺序出现，用户可以处理这种情况。

4. 事件通知的传播：特定应用决定了组播所要求的质量。例如，Jini的通告服务使用IP组播按一定的间隔发通告，将新近可用的服务通知感兴趣的各方。 156 ~ 157

这些例子说明一些应用程序要求比IP组播更可靠的组播协议。特别是，有可靠组播的需求——传输的任何消息或者被一个小组的所有成员都收到或者都收不到。这些例子也说明有些应用有较强的排序需求，最严格的称为全排序组播，这时，所有传输到一个组的消息按相同的顺序到达所有成员。

第11章将定义和介绍如何实现可靠组播和各种有用的排序保证，包括全排序组播。

4.6 实例研究：UNIX系统的进程间通信

UNIX BSD 4.x版本中的IPC原语是以系统调用方式提供的，它们作为在因特网TCP和UDP协议上实现的一层。消息目的地指定为套接字地址——一个套接字地址由一个因特网地址和一个本地端口号组成。

进程间通信操作基于4.2.2节描述的套接字抽象。像那里描述的一样，消息在发送套接字上排队等待直到网络协议传输它们，或者如果协议要求有确认消息，那么就直到确认到达时再传输。当消息到达时，它们在接收方的套接字上排队直到接收进程用一个适当的系统调用接收它们。

任何进程可以创建一个套接字用于与其他进程通信。这需要调用*socket*系统调用，它的参数指定了通信域（通常是因特网）、类型（数据报或流）和特定协议（有时需要）。通常由系统根据是数据报通信还是流通信选择协议（例如TCP或UDP）。

*socket*调用返回一个描述符，用以在后续的系统调用中引用该套接字。该套接字将一直存在直到它被关闭或直到使用该描述符的每个进程都退出。可为相同或不同计算机上进程之间的双向或单向通信使用一对套接字。

在一对进程通信前，接收方必须将它的套接字描述符绑定到一个套接字地址。如果发送方要求有应答，它也必须将它的套接字描述符绑定到一个套接字地址。*bind*系统调用用于该目的，它的参数是一个套接字描述符和一个结构引用，引用的结构包含了该套接字被绑定到的套接字地址。一旦绑定了一个套接字，它的地址就不能改变了。

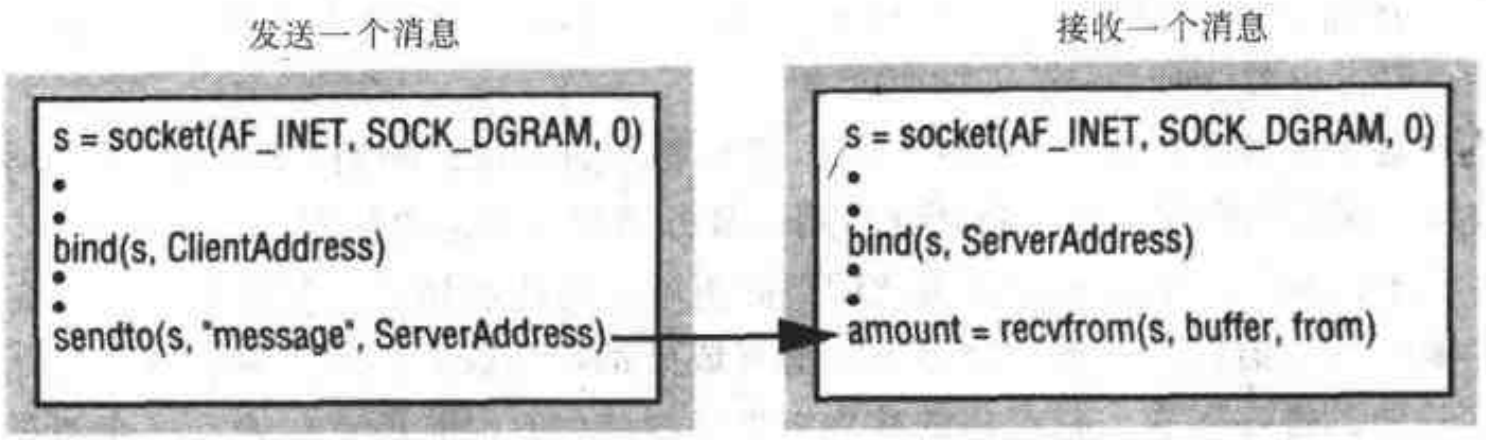
用一个系统调用同时完成创建套接字和将名字绑定到一个套接字似乎更合理，就像在Java API中一样。用两个独立的系统调用的好处是套接字在没有套接字地址时也有用。

就套接字地址用作进程目的地而言，套接字地址是公开的。在进程将它的套接字绑定到一个套接字地址之后，另一个指向相应套接字地址的进程就可以找到该套接字。任何进程，例如计划通过它的套接字接收信息的服务器，必须首先将那个套接字绑定到一个套接字地址，并使得该套接字地址被潜在的客户知晓。 158

4.6.1 数据报通信

为了发送数据报，每次通信时要识别一对套接字。这需要发送进程在每次发送消息时使用它本地的套接字描述符和接收套接字的套接字地址。

图4-18给出了说明，图上简化了参数描述。



*ServerAddress*和*ClientAddress*是套接字地址

图4-18 用于数据报的套接字

- 两个进程均使用*socket*调用创建套接字并获得该套接字的描述符。*socket*的第一个参数将通信域指定为因特网域，第二个参数表明使用数据报通信。*socket*调用的最后一个参数可用于指定特定的协议，将它设置为0表示将由系统选择一个合适的协议——在本例中是UDP。
- 两个进程接着使用*bind*调用将它们的套接字绑定到套接字地址上。发送进程将它的套接字绑定到一个指向任何可用的本地端口号的套接字地址。接收进程将它的套接字绑定到包含服务器端口的套接字地址，而发送方必须知道该地址。
- 发送进程使用*sendto*调用，其参数指定了消息要发送到的套接字、消息自身和目的地的套接字地址（对该地址结构的一个引用）。*sendto*调用将消息传递到底层UDP和IP协议，返回发送的实际字节数。当我们请求数据报服务时，消息传递到目的地并不需要确认。如果消息太长而不能发送，会返回一个错误（同时消息不被传输）。
- 接收进程使用*recvfrom*调用，其参数指定了接收消息的本地套接字和存储消息的内存位置和发送套接字的套接字地址（对该地址结构的一个引用）。*recvfrom*调用收取套接字队列上的第一个消息，如果队列为空，该调用将等待直到消息到达。

159

仅当一个进程中的*sendto*将它的消息导向到另一个进程中*recvfrom*使用的套接字时，才发生通信。在客户－服务器通信中，服务器不必预先知道客户套接字地址，因为*recvfrom*操作为它传递的每个消息提供发送方的地址。UNIX数据报通信的特性与4.2.3节描述的一样。

4.6.2 流通信

为了使用流协议，两个进程必须首先建立一对套接字之间的连接。双方的流程是不对称的，因为一个套接字将监听连接请求，而另一个套接字用于请求一个连接，见第4.2.4节的描述。一旦一对套接字建立了连接，它们可用于双向或单向传输数据。就是说，它们像流一样是因为任何可用的数据按照它们写入的顺序马上被读出，没有消息边界的说明。然而，接收套接字的队列是有界的，如果它空了，接收方会阻塞，如果它满了，发送方会阻塞。

对客户和服务器之间的通信，客户请求连接，监听服务器接收连接。当连接被接收时，UNIX自动创建一个新的套接字，与客户端套接字成为一对，这样，服务器可以通过原来的套接字继续监听其他客户的连接请求。在后续的流通信中能一直使用这对互连的流套接字，直到连接关闭。

流通信的过程如图4-19所示，图上简化了参数描述。该图没有给出服务器关闭监听的套接字。正常情况下，服务器应该首先监听并接收一个连接，然后派生一个新的进程与客户通信。同时，它将继续在原来的进程中监听连接请求。

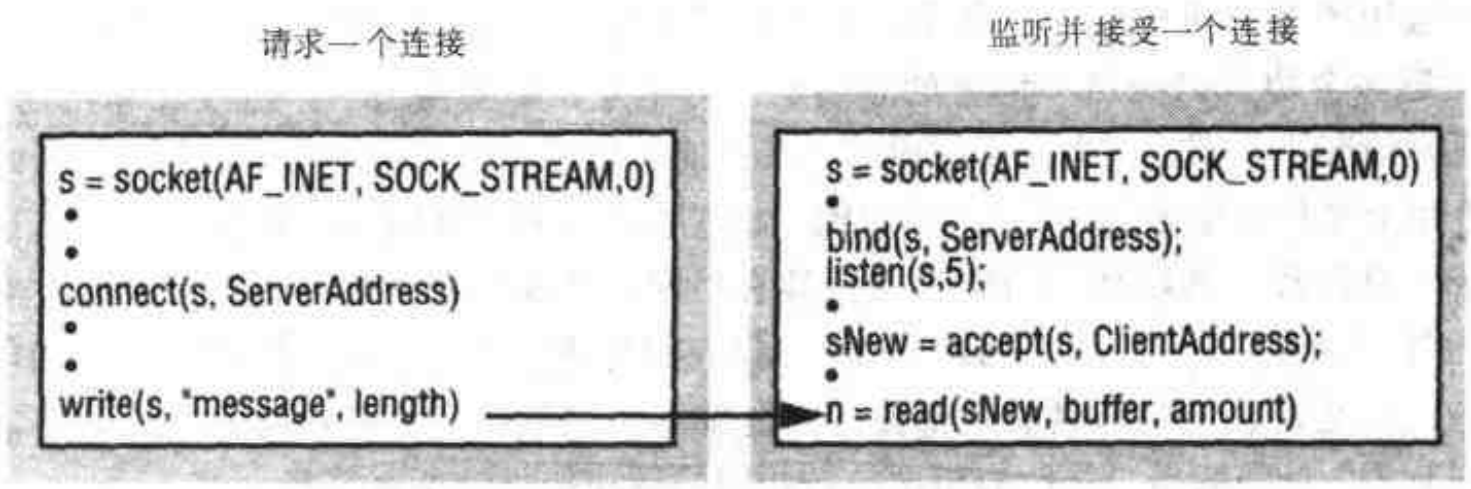


图4-19 使用流的套接字

- 服务器或监听进程首先使用*socket*操作创建一个流套接字，再用*bind*操作将它的套接字绑定到服务器的套接字地址。*socket*系统调用的第二个参数值是SOCK_STREAM，表明使用的是流通信。如果第三个参数设成0，将自动选择TCP/IP协议。它使用*listen*操作监听它的套接字上客户对建立连接的请求。*listen*系统调用的第二个参数指定能在该套接字上排队的连接请求的最大值。 160
- 服务器使用*accept*系统调用接收客户请求的连接，并获得一个新的套接字用于与该客户的通信。原来的套接字仍可以用于接收其他客户后续的连接请求。
- 客户进程使用*socket*操作创建一个流套接字，然后使用*connect*系统调用通过监听进程的套接字地址请求一个连接。因为*connect*调用自动将一个套接字名字与调用方的套接字绑定，所以事先的绑定是不必要的。
- 在连接建立之后，双方进程可以在各自的套接字上通过连接使用*write*和*read*操作发送和接收字节序列。*write*操作类似对文件的写操作，它指定要发送到套接字的消息，它将消息传递到底层的TCP/IP协议，然后返回实际发送的字符数。*read*操作在它的缓冲区中接收字符，并返回接收到的字符数。

UNIX流通信的特性与4.2.4节描述的一样。

4.7 小结

本章第1节说明了因特网协议提供了两个可替换的协议构造成分。在这两个协议之间存在一种有趣的权衡：UDP提供了一个简单的消息传递功能，它存在遗漏故障但没有内在的性能损失。另一方面，TCP保证了消息传递但以额外的消息、长延迟和存储开销为代价。

第2节给出了两种可互换的编码风格。CORBA和它的前任采用的编码数据的方式要求接收者具有各个成分的类型知识。相反，当Java序列化数据时，它包括了关于它内容类型的所有信息，允许接收方根据内容重构。另一个大的区别是CORBA需要编码（用IDL）数据项类型的规范以便生成编码和解码方法，而Java使用反射进行序列化对象和解序列化串行格式的对象。

请求－应答协议一节给出了一个有效的有特殊目的的分布式系统协议，它是基于UDP数据报的。应答消息形成了对请求消息的确认，这样避免了确认消息的额外开销。如果有必要，协议还能做到更可靠。按照该协议，不能保证“请求消息的发送将导致方法的执行”——对于一些应用这一协议已经可以满足要求了。但还可以获得额外的可靠性，例如通过利用消息 161
标识和消息重传确保一个方法最终确实被执行了。对具有幂等操作的服务，这是充分的。然

而，其他应用要求：重传应答消息不会重执行请求中的方法，借助于历史可实现这一点。这说明了构造多个协议以满足不同类的应用是个好主意，不要构造一个过度可靠的通用协议，因为在正常情况下（即，较少发生错误时）其性能比较差。

组播用于进程组成员之间的通信。IP组播提供了一个既可用于局域网又可用于因特网的组播服务。这种形式的组播与UDP数据报具有相同的故障语义，但除了会有遗漏故障外，它对许多组播应用是一个有用的工具。其他一些应用有更强的需求——特别是，组播传递应该是原子的，就是说，它应该具有全部传递或全部不传递的性质。另外的关于组播的需求与消息的排序有关，最强的需求是所有组成员按相同的顺序接收到所有消息。

练习

4.1 一个端口有多个接收者，这是否有用？

4.2 服务器创建了一个端口，用于从客户接收请求。讨论端口名字和客户使用的名字之间的关系的设计问题。

4.3 图4-3和图4-4中的程序可从cdk3.net/ipc获得。用它们做一个测试包判断数据报丢失的条件。提示：客户程序应该能改变发送消息的个数和它们的大小；来自某个特定客户的消息如果丢失了，服务器应该能检测到。

4.4 利用图4-3中的程序做一个客户程序，让它反复地从用户处读入输入，将它用UDP数据报消息发送到服务器，接着从服务器接收一条消息。客户在它的套接字上设置超时，以便在服务器不回答时客户能通知用户。用图4-4中的服务器测试该客户程序。

4.5 图4-5和图4-6中的程序可从cdk3.net/ipc获得。修改它们以便客户能反复获取用户输入行并将它写到流中；服务器反复地从流中读取，并将每次读取的结果打印出来。在用UDP数据报发送数据和流上发送数据之间做一个比较。

4.6 用练习4.5开发的程序测试接收方崩溃时发送方的结果以及反之的情况。

4.7 Sun XDR在传输前将数据编码成标准的大序法格式。与CORBA的CDR比较，讨论
162 这种方法的好处和不足。

4.8 Sun XDR在每个基本类型值上进行4字节边界对齐，而CORBA CDR对一个大小为n字节的基本类型在n字节边界对齐。讨论在选择基本类型值占用空间上的折衷。

4.9 为什么在CORBA CDR中没有显式的数据类型？

4.10 用伪代码编写一个算法描述4.3.2节中描述的序列化程序。算法应该给出类和实例句柄被定义或替换的时间。对下列类*Couple*的实例进行序列化，根据你的算法，描述要生成的序列化格式。

```
class Couple implements Serializable{
    private Person one;
    private Person two;
    public Couple(Person a, Person b) {
        one = a;
        two = b;
    }
}
```

4.11 用伪代码编写一个算法描述由练习4.10定义的算法产生的序列化格式的解序列化过程。提示：使用反射，根据它的名字创建一个类，根据它的参数类型创建构造函数，根据构

造函数和参数值创建对象的新实例。

4.12 定义一个其实例表示远程对象引用的类。它应该包含类似图4-10的信息，应该提供请求－应答协议的访问方法。解释每个访问方法如何被协议使用。对包含远程对象接口信息的实例变量，给出其类型选择的理由。

4.13 定义一个类，该类的实例表示如图4-13所示的请求和应答消息。该类应该提供一对构造函数，一个构造函数用于请求消息，另一个用于应答消息，这些构造函数给出了请求标识是如何赋予的。它还应该提供一个方法将自身编码成字节数组，另一个方法将字节数组解码成一个实例。

4.14 利用UDP通信，但不增加任何容错手段，为图4-12请求－应答协议的3个操作编程。应该使用练习4.12和练习4.13定义的类。

4.15 给出服务器实现的概要，说明创建新线程执行每个客户请求的服务器如何使用操作*getRequest*和*sendReply*。说明服务器如何从请求消息中复制请求Id到应答消息以及它如何获得客户IP地址和端口。

4.16 定义*doOperation*方法的一个新版本，它在等待应答消息时可设置一个超时。过了超时，它将重传请求消息*n*次。如果仍没有应答，它将通知调用者。

4.17 描述如下的情形：客户接收到一个应答，但该应答来自早先发出的请求调用。 163

4.18 描述请求－应答协议屏蔽操作系统和计算机网络异构性的方式。

4.19 讨论下列操作是否是幂等的：

- 按电梯的请求按钮。
- 写数据到文件。
- 将数据追加到文件。

操作不应该与任何状态相关是不是幂等的必要条件？

4.20 解释与减少服务器端应答数据量有关的设计选择。比较使用RR和RRA协议时，各自的存储需求。

4.21 假设使用RRA协议。服务器应该对未确认的应答数据保留多长时间？为了接收到一个确认，服务器应该反复地发送应答吗？

4.22 为什么在协议中交换的消息数比发送的数据总量对性能而言更重要？设计RRA协议的一个变体，采用捎带确认法，即如果有合适的下一个请求，就将确认附加在该消息中传输，否则用单独的消息发送确认。（提示：在客户端使用额外的定时器。）

4.23 IP组播提供的服务存在遗漏故障。做一个测试包，可基于图4-17中的程序，用于发现组播消息被组播组成员丢弃的条件。该测试包应该设计成能允许多个发送进程。

4.24 设计一个消息重传的机制，用于在IP组播中克服消息丢失问题。应该考虑下列几点：

- 可以有多个发送方。
- 通常只有一小部分消息丢失。
- 与请求－应答协议不同，接收方没必要在特定时间限制内发送消息。

假设没有丢失的消息按发送方的顺序到达。

4.25 练习4.24的解决方案应该克服IP组播的消息丢失问题。这一解决方案在什么意义上与可靠组播的定义不同？

4.26 设计一个方案，由不同客户发送的组播按不同的顺序传递到两个组成员。假设使用了一些消息重传，但未丢失的消息按发送方的顺序到达。接收方如何补救这种情况。

4.27 利用IP组播，定义组形式的请求－应答交互的语义并为之设计一个协议。 164

第5章　分布式对象和远程调用

5.1　简介
5.2　分布式对象间的通信
5.3　远程过程调用
5.4　事件和通知
5.5　Java RMi实例研究
5.6　小结

本章将通过远程方法调用（RMI）介绍分布式对象之间的通信。能够接收远程方法调用的对象称为远程对象，远程对象实现了一个远程接口。因为调用者与被调用对象存在分别失败的可能性，所以RMI与本地调用有着不同的语义。虽然可以使RMI看起来与本地调用非常相似，但是完全透明性未必是人们所希望的。根据远程接口定义，接口编译器会自动生成用于参数编码、参数解码、发送请求消息和应答消息的代码。

远程过程调用（RPC）之于RMI就像过程调用之于对象调用，本章将通过Sun RPC实例研究描述和阐明远程过程调用。

基于事件的分布式系统允许对象预订发生在远程兴趣对象上的事件并且在这一事件发生时接收到通知。事件和通知提供了一种在异构对象间进行异步通信的方式。本章将把Jini分布式事件规范作为一个实例研究。

在Java RMI实例研究中阐明了对RMI的应用。

第17章是关于CORBA的实例研究，包括了CORBA RMI和CORBA事件服务两方面的内容。

5.1　简介

本章涉及到分布式应用的编程模型。分布式应用是指由运行在不同进程中相互协作的程序组成的应用。这类程序要能够调用另一个进程中的操作，而另一个进程通常运行在不同的计算机中。为了做到这一点，人们扩展了一些熟悉的编程模型，然后将它们应用到分布式程序中：

- 最早的而且或许也是最为熟知的扩展就是从传统的过程调用模型到远程过程调用模型的扩展，它允许客户程序调用运行在另一个进程而且通常运行在不同于该客户的另一台计算机中的服务器程序。
- 最近，基于对象的编程模型已经扩展到允许不同进程里的对象通过远程方法调用（RMI）的方式彼此通信。RMI是本地方法调用的扩展，它使得生存在某一进程中的对象可以调用生存在另一个进程中的对象的方法。
- 基于事件的编程模型允许对象接收它感兴趣的对象上发生的事件的通知。这一模型已经扩展到允许编写基于事件的分布式程序。

注意一点，此处用术语“RMI”指普通情形中的远程方法调用，它不应该与远程方法调

用的某些特例如Java RMI等混淆。当前大多数的分布式系统软件都使用面向对象的语言编写，RPC可以被理解为与RMI相关。因此本章集中阐述了RMI和事件模式，它们都将应用于分布式对象。5.2节将介绍分布式对象之间的通信，接着讨论RMI的设计和实现。5.5节给出了Java RMI实例研究。RPC将在5.3节Sun RPC实例研究中讨论。5.4节讨论了事件和分布式通知，而更深入的关于CORBA的实例研究将在17章中给出。

中间件 在进程和消息传递等基本构造模块之上提供编程模型的软件称为中间件。中间件层使用基于进程间消息的协议来提供更高级的抽象，如远程调用和事件等，如图5-1所示。例如，远程方法调用抽象就是基于4.4节讨论的请求－应答协议。

中间件的重要作用是提供位置透明性以及与具体通信协议、操作系统和计算机硬件的无关性。某些形式的中间件还允许各个组件以不同的编程语言编写。

图5-1 中间件层

位置透明性 在RPC中，调用某一过程的客户不能判断该过程是运行在同一个进程中还是运行在不同的进程中，甚至可能是运行在另外一台不同的计算机中。客户也不必知道服务器的位置。类似地，在RMI中，发起调用的对象也不能判断它调用的对象是否在本地，同时也不必知道它的位置。而且在基于事件的分布式程序中，产生事件的对象和接收那些事件通知的对象都无需知道对方的位置。

通信协议 支持中间件抽象的协议与其下层的传输协议无关。例如，请求－应答协议既可以在UDP上实现，也可以在TCP上实现。

计算机硬件 用于外部数据表示的两种公认标准在4.3节已经描述过了。在消息编码和解码的时候需要应用它们。它们隐藏了由于硬件体系结构差异而导致的不同，例如字节次序。

操作系统 中间件层提供的更高级的抽象与基本的操作系统无关。

几种编程语言的使用 一些中间件允许分布式应用使用一种以上的编程语言。特别是CORBA（见第17章）允许以一种语言编写的客户程序调用以另一种语言编写的服务器程序中的对象的方法。它的实现方法是通过使用接口定义语言或IDL来定义接口。IDL将在下面讨论。

接口

大多数现代编程语言提供了有关方法，可以把一个程序组织成一系列能彼此通信的模块。模块之间的通信可以采用模块间的过程调用，或者直接访问另外一个模块中的变量。为了控制模块之间可能的交互，必须为每一个模块定义显式的接口，模块接口指定了可供其他模块访问的过程和变量。这样，模块实现后就隐藏了除通过其接口可获得的信息之外的所有信息。只要模块的接口保持相同，它的实现的改变就不会影响模块使用者。

分布式系统中的接口 在分布式程序中，模块可能运行在彼此独立的进程中。让运行在一个进程中的模块访问另一个进程中的变量是不可能的。因此，用于RPC或RMI的模块的接口不能指定对变量的直接访问。注意，CORBA IDL接口可以指定属性，这看起来像是打破了上述的规则。然而，也不允许直接访问这些属性，而必须通过一些为接口自动添加的获取或设置过程来访问。

166
?
167

当过程及其调用者分布在不同进程中的时候，用于本地过程调用的值调用和引用调用等参数传递机制就不合适了。在分布式程序里，模块接口中的过程或方法的规约把参数描述为输入（input）型或者输出（output）型，或者有时候兼而有之。输入型参数被传递给远程模块，它先通过请求消息发送参数的值，然后将这些值提供给服务器作为操作执行的参数。输出型参数在应答消息中返回，可以作为调用的结果或者替换调用环境中相应变量的值。当一个参数既是输入型又是输出型时，它的值必须既在请求消息又在应答消息中被传输。

本地模块和远程模块之间的另一点差异在于，一个进程中的指针在另一远程进程里是无效的。因此，指针不能作为参数传递，也不能作为远程模块调用的结果返回。

下面两段讨论原始客户－服务器模型中的RPC使用的接口和分布式对象模型中RMI使用的接口：

服务接口 在客户－服务器模式下，每个服务器提供一系列供客户使用的过程。例如，文件服务器会提供读写文件的过程。术语服务接口指的是由服务器提供的过程规约，它定义了每个过程中的输入、输出参数的类型。

远程接口 在分布式对象模型中，远程接口指定了可供其他进程中对象进行调用的对象的方法，它定义了每个方法的输入输出参数的类型。然而，很大的差异在于远程接口中的方法可以把对象作为参数传递或者作为方法的结果传递。另外，也可以传递远程对象引用，引用的概念不能与指针混淆，指针描述的是特定的内存地址（4.3.3节描述了远程对象引用的内容）。

服务接口和远程接口都不能指定对变量的直接访问。后者还禁止直接访问对象的实例变量。

接口定义语言 RMI机制可以集成到某个编程语言中，如果该语言包含了适当的定义接口的表示法，就允许将输入和输出参数映射成该语言中正常使用的参数。Java RMI就是将RMI机制加到面向对象编程语言上的一个例子。当一个分布式应用的所有部分都是用同一种语言编写时，这种方法非常有效。因为它允许编程员用同一种语言实现本地和远程调用，所以这种方法也很方便。

然而，许多现存有用的服务是用C++和其他语言编写的。为了远程访问的需要，允许程序采用包括Java在内的各种语言编写是有益的。接口定义语言（IDL）使得以不同语言实现的对象能相互调用。IDL提供了一种定义接口的表示法，接口中方法的每个参数在类型说明之外还可以描述为输入型参数或输出型参数。

168

图5-2给出了CORBA IDL的一个简单例子。*Person*结构与4.3.1节中用于说明编码的结构相同。名为*PersonList*的接口指定了在实现这个接口的远程对象中对RMI可用的方法。例如，方法*addPerson*指定它的参数是*in*型，意味着它是一个输入参数；而方法*getPerson*是通过名字获取一个*Person*实例，它将其第2个参数指定为*out*型，意味着这是一个输出参数。我们的实例研究包括了CORBA IDL和Sun XDR，其中CORBA IDL是RMI的一种IDL例子（见第17章），而

Sun XDR是RPC的一种IDL例子。

```
//在文件Person.idl中
struct Person {
        string name;
        string place;
        long year;
};
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

图5-2 CORBA IDL的例子

其他例子包括用于OSF的分布式计算环境（DCE）中RPC系统的接口定义语言[OSF 1997]，它使用了C语言的语法，也称为IDL；还有DCOM IDL，它是建立在DCE IDL基础上的[Box 1998]，用于微软的分布式组件对象模型（DCOM）中。

5.2 分布式对象间的通信

第1章介绍的用于分布式系统的基于对象的模型扩展了面向对象编程语言支持的模型，使得它能应用到分布式对象。本节讨论分布式对象间通过RMI方式的通信。相关的材料被安排在下列标题下：

- 对象模型　对象模型相关方面的简单回顾，适合于具有一种面向对象编程语言（如Java或C++）基础知识的读者。

169

- 分布式对象　介绍基于对象的分布式系统，表明对象模型非常适合于分布式系统。
- 分布式对象模型　讨论将对象模型进行必要的扩展以支持分布式对象。
- 设计问题　一系列关于设计方案的争论。
  1. 本地调用恰好只执行一次，但是什么是适合远程调用的语义呢？
  2. RMI的语义如何才能与本地方法调用相似，有什么差别不能被消除？
- 实现　阐述了在请求－应答协议之上，如何设计中间件层，使其支持应用层分布式对象间的RMI。
- 分布式无用单元回收　介绍适用于RMI实现的分布式无用单元回收的一种算法。

5.2.1 对象模型

用Java或C++等编写的面向对象程序由一个交互对象集合组成，其中每个对象包含一组数据和方法。一个对象与其他对象之间的通信就是调用对方的方法，通常要传递参数和接收结果。对象可以封装它们的数据和方法的代码。有些语言，例如Java和C++，允许程序员定义其实例变量可以被直接访问的对象。但是在分布式对象系统应用中，一个对象的数据应该只能通过其方法来存取。

对象引用 可以通过对象引用来访问对象。例如，在Java中，一个变量看上去拥有一个对象，但实际上只拥有一个对象引用。为了调用对象的一个方法，需要给出对象引用和方法名，并带有必要的参数。其方法被调用的对象有时被称为目标，有时被称为接收者。对象引用作为一类值可以赋给变量，也可作为参数传递或者作为方法的结果返回。

接口 接口在无需规定其实现的情况下提供了一系列方法标记名称的定义（即参数的类型、返回值和异常）。如果对象的类包含实现接口方法的代码，那么对象将提供特定的接口。在Java中，一个类可以实现几个接口，也可以由任意类实现一个接口的方法。接口还定义了用于声明参数类型、变量类型及方法返回值的类型。注意，接口没有构造函数。

动作 在面向对象程序中，动作由一个对象启动，这个对象调用另一个对象中的方法。调用可能包含执行方法所需要的额外信息（参数）。接收者执行适当的方法，然后将控制返回给调用对象，有时候会提供一个结果。方法的调用将产生两重结果：

1. 可能会改变接收者的状态。
2. 可能会在其他对象中发生其他方法调用。

170

因为调用可能导致其他对象的方法调用，所以动作就是一连串相关方法调用，它们中的每一个最终将返回。这种解释没有考虑异常的发生。

异常 程序可能会遇到各种错误和无法预计的严重状况。在方法执行期间，可能会发现许多不同的问题：例如，对象变量的值不一致，或者因读写文件或网络套接字而产生的失败。程序员需要在他们的代码中插入测试语句以处理所有可能并不经常出现的情况或出错情况，这降低了正常情况下的代码的清晰性。异常提供了一种在不使代码复杂化的前提下处理错误条件的好方法。另外，每个方法标题都清楚地列出了发生异常的错误条件，以便调用方法的用户去处理它们。或许会定义一块代码，以便在某种不期望发生的条件或错误出现的时候抛出异常。这意味着程序控制转给了另一块用于捕获异常的代码，控制不会再返回到抛出异常的地方。

无用单元回收 有必要提供一种手段在不再需要对象时释放其占用的空间。有的语言，例如Java，可以自动检测出什么时候应该收回一个已经不再访问的对象占用的空间，并将此空间分配给其他对象使用。这种处理被称为无用单元回收。有的语言（例如C++）不支持无用单元回收，那么程序员必须自己处理如何释放分配给对象的空间。这是一个主要的出错源。

5.2.2 分布式对象

对象的状态由它的实例变量值组成。在基于对象的模型中，程序的状态被划分为几个独立的部分，每个部分都与一个对象关联。因为基于对象的程序在逻辑上是分区的，所以在分布式系统中很自然地将对象物理地分布在不同的进程或计算机中。

分布式对象系统可以采用客户－服务器体系结构。在此情形下，对象由服务器管理，它们的客户通过远程方法调用来调用对象的方法。在RMI中，客户调用一个对象方法的请求以消息的形式传递到管理该对象的服务器，通过在服务器端执行对象的方法来完成该调用，并将处理的结果在另一个消息中返回给客户。考虑到会有一连串的相关调用，因此在服务器中的对象也允许成为其他服务器中对象的客户。

分布式对象也可采用在第2章中描述的其他体系结构模型。例如，为了获得良好的容错性和增强的性能，可能会复制对象。又如，为了增强性能和可用性，可能会迁移对象。

将客户和服务器对象分布在不同的进程中，可提高封装性。也就是说，一个对象的状态只能被该对象的方法访问，这意味着不可能让未经授权的方法作用于该对象的状态。例如，不同机器上的对象可能会并发RMI，这意味着可能会并发地访问一个对象。因此，访问冲突
171 的可能性就出现了。然而，事实上，那些只能通过对象自己的方法访问的数据使对象可以提供保护自身免遭不正确访问的方法。例如，它们会用条件变量等同步原语来保护对它们实例变量的访问。

将分布式程序的共享状态视为一个对象集的另一个好处是，可以通过RMI来访问对象。如果类的实现在本地可用，则可以将对象拷贝到一个本地缓存并进行直接访问。

对象只能通过自己的方法访问的事实有益于异构系统。因为异构系统通常具有不同场合使用的不同数据格式，而使用RMI访问对象方法的客户将不会注意到数据格式的不同。

5.2.3 分布式对象模型

本节讨论对象模型的扩展以使它适用于分布式对象。每个进程包含若干对象，其中有些对象既可以接收远程调用又可以接收本地调用，而另一些对象只能接收本地调用，如图5-3所示。不管是否在同一台计算机内，不同进程中的对象之间的方法调用都被认为是远程方法调用。在同一进程中的对象间的方法调用被称为本地方法调用。

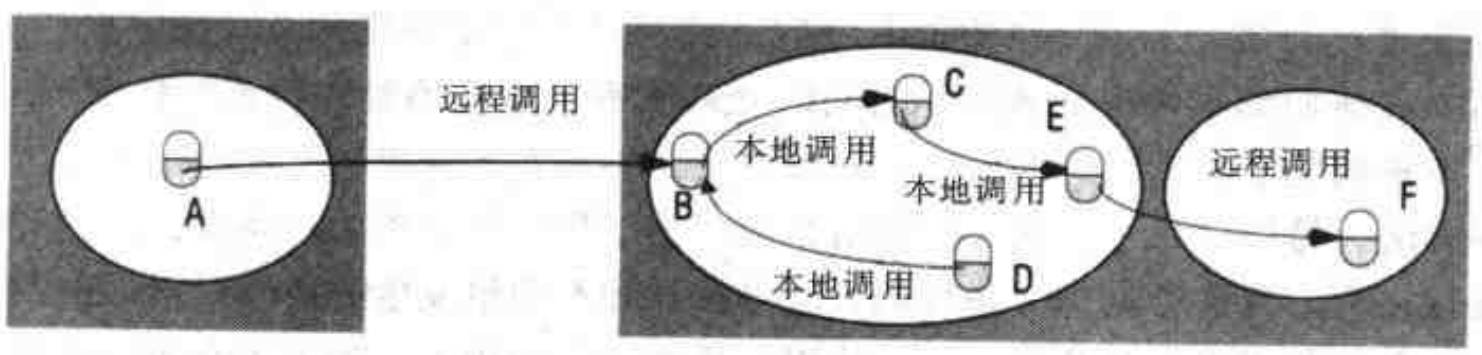


图5-3 远程方法调用和本地方法调用

我们将那些能够接收远程调用的对象称为远程对象。在图5-3中，对象B和F是远程对象。所有对象都能接收本地调用，虽然它们只能接收引用了它们的其他对象发出的本地调用。例如，对象C必须有一个对象E的引用，只有这样它才可以调用E的方法。下面两个基本概念是分布式对象模型的核心：

- 远程对象引用　如果其他对象能访问某个远程对象的远程对象引用，那么它们就可以调用这个远程对象上的方法。例如在图5-3中，B的远程对象引用对A必须是可用的。
- 远程接口　每个远程对象都有一个远程接口，由该接口描述哪些方法是可以远程调用的。
172 例如，对象B和F必须具有远程接口。

后续的段落将讨论远程对象引用、远程接口和分布式对象模型的其他方面。

远程对象引用　对象引用的概念要加以扩展，即，允许那些接收RMI的对象具有远程对象引用。远程对象引用是一个可以用于整个分布式系统的标识符，用于指向某个惟一的远程对象。它的表示法通常与本地对象引用不同，我们已经在4.3.3节中讨论过了。远程对象引用与本地对象引用有以下两点类似：

1. 以远程对象引用的形式指定接收远程方法调用的远程对象。
2. 远程对象引用可以作为远程方法调用的参数和结果传递。

远程接口　远程对象类实现其远程接口中的方法，例如，在Java中作为公有实例方法实现。

在其他进程中的对象只能调用属于其远程接口的方法，如图5-4所示。本地对象可以调用远程接口中的方法和由远程对象实现的其他方法。注意，远程接口像所有接口一样，没有构造函数。

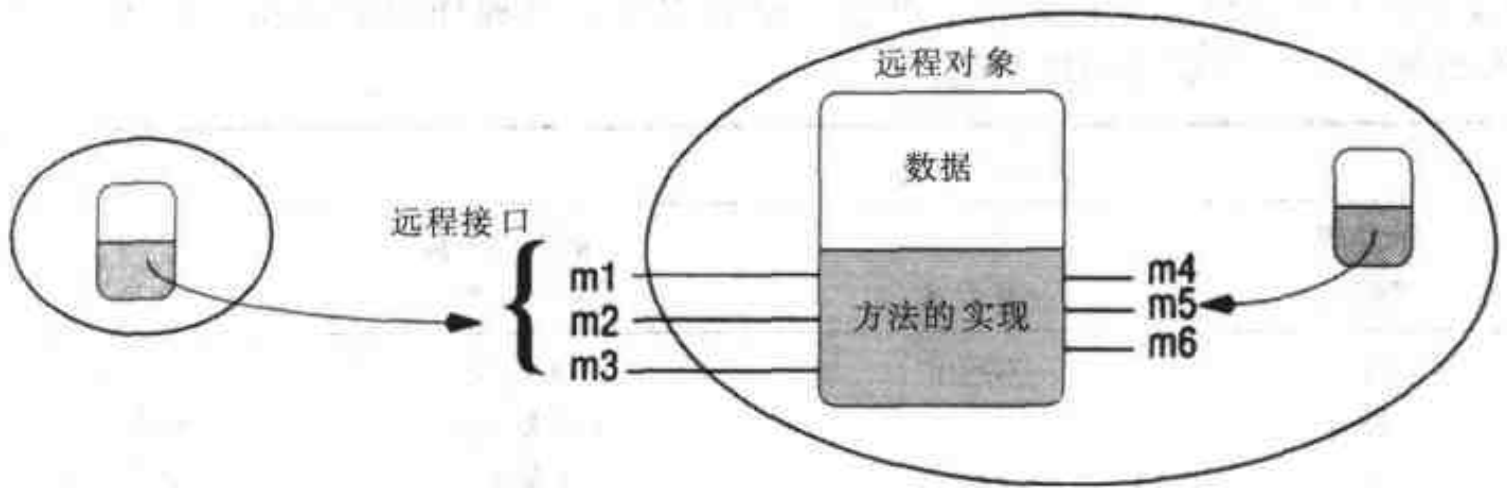


图5-4 远程对象及其远程接口

CORBA系统提供一个接口定义语言（IDL），用于定义远程接口。图5-2是一个用CORBA IDL定义的远程接口的例子。远程对象的类和客户程序可以用任何IDL编译器适用的语言实现，例如C++或Java。CORBA客户不需要为了能够远程调用其方法而使用与远程对象相同的语言。

在Java RMI中，以与任何其他Java接口相同的方式定义远程接口。它们通过扩展一个名为*Remote*的接口而获得远程接口的能力。CORBA IDL（见17.2.3节）和Java都支持接口的多重继承，即一个接口可以扩展一个或多个其他接口。

分布式对象系统中的动作 类似于非分布式系统中的情形，一个动作是由方法调用启动的，这可能会导致其他对象上的方法调用。但是在分布式系统中，涉及相关调用链的对象可 173
能处于不同的进程或不同的计算机中。当调用跨越了进程或计算机边界的时候，就要使用RMI，而为了使RMI成为可能，对象的远程引用必须是可用的。图5-3中，对象A需要有对象B的远程对象引用。远程对象引用可以作为远程方法调用的结果返回。例如，图5-3中的对象A可能会从对象B得到一个对象F的远程引用。

在分布式对象系统中的无用单元回收 如果语言（例如Java）支持无用单元回收，那么任何与之相关的RMI系统也应该允许远程对象的无用单元回收。分布式无用单元回收通常通过已有的本地无用单元回收器和一个完成分布式无用单元回收的附加模块（一般基于引用计数）的协作来实现。5.2.6节就此做了详尽的描述。

异常 任一远程调用都可能会因为被调用对象的种种原因而失败，而被调用的对象与调用对象在不同的进程或计算机中。例如，包含远程对象的进程可能崩溃了，或者由于太忙而无法应答，或者是调用或结果消息丢失了。因此，远程方法调用应该能够产生异常，如因分布引起的超时异常，以及方法调用执行期间导致的各种异常。后者的例子是一个超越文件末尾的读操作尝试，或者是未经正确授权的文件访问。

CORBA IDL提供了指定应用级异常的表示法。当因为分布而引起错误时，底层系统生成标准异常。CORBA客户程序要能处理异常，例如，一个C++客户程序会使用C++中的异常机制。

5.2.4 RMI的设计问题

前面的章节已经指出，RMI是本地方法调用的自然扩展。本节将讨论在进行这种扩展时出现的两个设计问题：

- 虽然本地调用恰好只执行一次，但这并不总是适用于远程方法调用的场合。此处讨论其

他语义。

- 对RMI最合适的透明性级别。

在5.2节余下的部分，我们将把那些包含远程对象的进程作为服务器，把那些包含调用者的进程作为客户。服务器也可能是客户。

容错措施			调用语义
重发请求消息	过滤重复请求	重新执行过程或重传应答	
否	不适用	不适用	或许
是	否	重新执行过程	至少一次
是	是	重传应答	至多一次

图5-5 调用语义

RMI调用语义 4.4节讨论了请求－应答协议，说明了*doOperation*可以通过不同的方式实现以提供不同的传输保证。主要的选择有：

- *重发请求消息* 在接收到应答或者假定服务器已经出现故障之前是否要重发请求消息。
- *过滤重复请求* 当启用重传请求的时候，是否要在服务器过滤掉重复的请求。
- *重传结果* 是否要在服务器保存结果消息的历史记录，以便无需重新执行服务器上的操作就能重传丢失的结果。

这些选择的组合导致了调用者所见到的远程调用可靠性的各种可能语义。图5-5给出了有关选择及其产生的调用语义名。注意，对于本地方法调用，语义是恰好一次，意味着每个方法都恰好执行一次。调用语义定义如下：

或许调用语义 采用或许调用语义，调用者不能判断一个远程方法是已经执行过一次还是从来没有执行过。当没有采取任何容错措施时，或许语义就启用了。它可能会遇到如下类型的故障：

- 遗漏故障，如果调用或结果消息丢失。
- 系统崩溃，由于包含远程对象的服务器出现故障。

如果在超时后没有接收到结果消息，并且也不再重发请求消息，那么就不能确定是否执行过该方法。如果调用消息丢失了，那么不会执行该方法。但也可能执行过方法了，只是结果消息丢失了。系统崩溃可能发生在执行方法之前或之后。此外，在异步系统中，执行方法返回的结果可能会在超时后才到达。或许语义仅对那些可以接受偶然调用失败的应用是有用的。

至少一次调用语义 采用至少一次调用语义，调用者或者可以接收到返回的结果，在这种情况下，调用者知道至少执行过一次该方法；或者会接收到一个异常，通知它没有接收到执行结果。可以通过重发请求消息来实现至少一次调用语义，它克服了调用或结果消息的遗漏故障。至少一次调用语义可能会遇到下列类型的故障：

- 由于包含远程对象的服务器发生故障而引起的系统崩溃。

174
~
175

- 随机错误。重发调用消息时，远程对象可能会多次接收到这一消息并执行某一方法，这可能导致存储或返回错误的值。

第4章定义了一个幂等操作，这种操作反复执行后达到的结果看起来与恰好执行过一次一样。非幂等操作在多次执行之后可能会出现错误的结果。例如，一个向银行账户增加10美元的操作只应该执行一次，如果重复执行它的话，存款余额就可能不断增加！如果对服务器中

的对象进行设计，使其远程接口中所有的方法都是幂等操作的话，那么至少一次调用语义是可以接受的。

至多一次调用语义　采用至多一次调用语义，调用者或者可以接收到返回的结果，在这种情况下，调用者知道该方法恰好执行过一次；或者接收到一个异常，通知它执行结果没有接收到，在这种情形下，调用者知道该方法要么执行过一次，要么从未执行过。能通过使用所有的容错措施来实现至多一次调用语义。正如前面的情形，重发请求消息可以克服所有调用或结果消息的遗漏故障，容错措施通过确保执行每个RMI方法永远不超过一次来避免随机错误。在Java RMI和CORBA中，调用语义都是至多一次，但CORBA也允许采用或许语义，用于那些不返回结果的方法。Sun RPC提供至少一次调用语义。

透明性　RPC的创始人Birrell和Nelson[1984]致力于使远程过程调用与本地过程调用尽可能相似，使得本地和远程过程调用的语法没有差别。所有对编码和消息传递过程必要的操作都对编写调用的程序员隐藏起来。虽然在超时后重新发送请求消息，但是这对调用者是透明的，这使得远程过程调用的语义与本地过程调用相似。这种透明性概念延伸应用到了分布式对象中，但是它不仅隐藏编码和消息传递过程，而且还隐藏了定位和连接远程对象的任务。举例来说，Java RMI通过允许使用相同的语法使得远程方法调用和本地调用非常相似。

然而，远程调用比本地调用更易出现故障，因为它们涉及网络、另一台计算机和另一个进程。不论选择上面哪种调用语义，总有可能接收不到结果，而且出现故障时，不能判别其来源是由于网络的失效还是由于远程服务器进程的故障。这就要求发出远程调用的对象能够从这些故障中恢复。

远程调用的延迟要比本地调用大好几个数量级。这表明利用远程调用的程序要把这个因素考虑进去，例如尽可能减少远程交互等。Argus的设计者[Liskov and Scheifler 1982]建议调用者应该能够中止那种花费了很长时间但是对服务器却毫无效果的远程过程调用。为了做到这一点，服务器要能恢复到过程调用之前的状态。这些问题将在第12章讨论。

Waldo等[1994]认为，应该在远程接口上表现本地对象和远程对象之间的不同，让对象对可能出现的部分失败以一致的方式做出反应。比起这种关于远程调用的语法是否应该与本地调用不同的争论来，有些系统要做得更多：以Argus为例，它扩展了编程语言，使得程序员要显式地进行远程操作。

关于远程调用是否应该透明的选择也同样适用于IDL的设计者。例如，在CORBA中，当客户不能与远程对象通信时，远程调用就会抛出一个异常。此时要求客户程序能处理这一异常，并解决此类故障。IDL也可能提供一种指定方法调用语义的功能，这会有助于远程对象的设计者。例如，倘若为了避免至多一次调用语义造成的系统开销而选择了至少一次调用语义，那么对象的操作应该被设计成幂等的。

当前似乎形成了一致意见：从远程调用的语法与本地调用一致的角度看，远程调用应该是透明的，但本地和远程对象间的不同应该表现在它们的接口上。在Java RMI中，能通过实现*Remote*接口和抛出*RemoteExceptions*的事实来区分远程对象。用IDL指定其接口的远程对象的实现者显然了解其不同。通过远程调用访问对象，这对对象的设计者还有另一层含义：对象应该能在多个客户并发访问的情况下保证它的状态是一致的。

5.2.5 RMI的实现

完成远程方法调用包括几个独立的对象和模块。如图5-6所示，一个应用级对象A拥有一

个远程应用级对象B的远程对象引用并调用B的一个方法。本节讨论图中每一组成部分的作用，首先讨论通信和远程引用模块，然后讨论运行在它们上面的RMI软件。

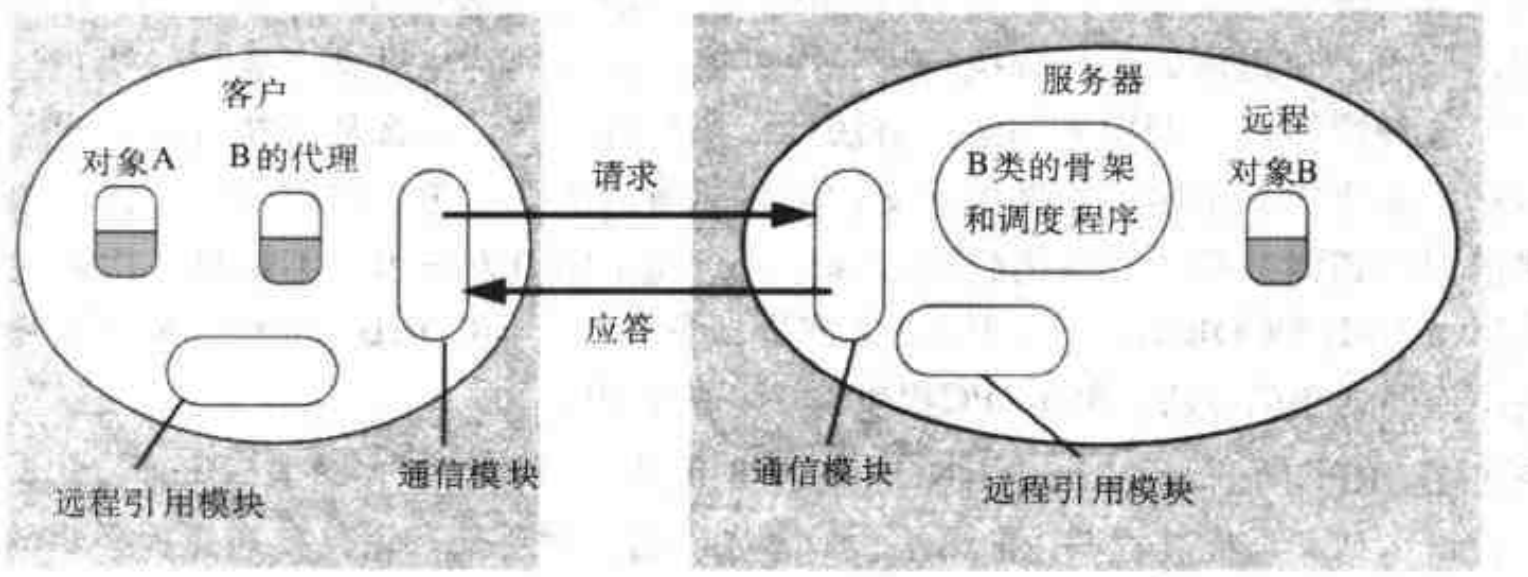


图5-6 在远程方法调用中的代理和骨架角色

本节剩余部分阐述以下的话题：代理的创建，将名字绑定到它们的远程对象引用，对象的激活和钝化以及根据远程对象引用进行对象定位。

通信模块 两个相互协作的通信模块实现请求－应答协议，它们在客户和服务器之间传递
177 请求和应答消息。请求和应答消息的内容如图4-13所示。通信模块只使用前3项，即消息类型、它的请求ID和被调用对象的远程引用。方法ID和所有的编码与解码都由下面讨论的RMI软件考虑。两个通信模块一起负责提供一个指定的调用语义，例如至多一次调用语义。

服务器端通信模块为被调用对象类选择调度程序，传输其本地引用，该本地引用取自远程引用模块，用来替换请求消息中的远程对象标识符。调度程序的角色在下面介绍RMI软件时讨论。

远程引用模块 远程引用模块负责翻译本地和远程对象引用以及创建远程对象引用。为完成其职责，每个进程中的远程引用模块都有一个远程对象表，记录着该进程的本地对象引用和远程对象引用（整个系统的）的对应关系。这张表包括：

- 该进程拥有的所有远程对象的表项。例如，在图5-6中，远程对象B会记录在服务器端的表中。
- 每个本地代理的表项。例如，在图5-6中，在客户端的表中会记录B的代理。

代理的作用在下面讨论。远程引用模块的动作如下：

- 当远程对象第一次作为参数或结果传递时，要求远程引用模块创建一个远程对象引用，并把它添加到表中。
- 当远程对象引用随请求或应答消息到达时，远程引用模块提供对应的本地对象引用，它可能指向一个代理，也可能指向一个远程对象。若远程对象引用不在表中，那么RMI软件就创建一个新的代理并要求远程对象引用模块把它添加到表中。

在为远程对象引用进行编码和解码时，由RMI软件的组件调用这个模块。例如，当请求消息到达的时候，此表用于找出是哪个本地对象被调用。

RMI软件 它由应用层对象和通信模块、远程引用模块之间的软件层组成。在图5-6中，中间件对象的作用包括如下几种：

- 代理 代理的作用是通过在调用者面前表现得像本地对象一样，使远程方法调用对客户

透明，它不执行调用，而是将调用放在消息里传递给远程对象。它隐藏了远程对象引用的细节、参数的编码、结果的解码以及客户消息的发送和接收。对于具有远程对象引用的进程，其每个远程对象都有一个代理。代理类实现它所代表的远程对象的远程接口定义的方法，这可以保证远程方法调用与远程对象的类型相匹配。然而，代理实现它们的方式非常与众不同。代理中的每个方法会把一个目标对象的引用、它自身的方法ID和它 178
的参数编码进一个请求消息并发送到目标，等待应答消息，然后解码并将结果返回给调用者。

- 调度程序 服务器对表示远程对象的每个类都有一个调度程序和骨架。在我们的例子中，服务器有远程对象B这个类的调度程序和骨架。调度程序接收来自通信模块的请求消息。它传递请求消息，并使用方法ID选择骨架中恰当的方法。调度程序和代理对远程接口中的方法使用相同的方法ID。
- 骨架 远程对象类有一个骨架，用于实现远程接口中的方法。它们的实现与远程对象中的方法具有很大的不同。骨架方法解码请求消息中的参数，并调用远程对象中的相应方法，它等待调用的完成，然后将结果和任何异常信息编码进应答消息，发送给代理的方法。

远程对象引用以图4-10中的形式编码，包括关于远程对象的远程接口的信息，例如远程接口名或远程对象类。这条信息能确定代理类，以便在需要的时候创建一个新的代理。例如，可以通过把“*_proxy*”添加到远程接口名中创建代理类名。

创建代理类、调度程序类和骨架类 由接口编译器自动创建在RMI中使用的代理类、调度程序类和骨架类。例如，在CORBA的Orbix实现中，远程对象的接口以CORBA IDL定义，而接口编译器能用C++语言创建代理类、调度程序类和骨架类。对于Java RMI，由远程对象提供的方法集合被定义为一个Java接口，它是在远程对象类中实现的。Java RMI编译器根据远程对象类创建代理类、调度程序类和骨架类。

服务器程序和客户程序 服务器程序包含调度程序类和骨架类，以及它支持的所有远程对象类的实现（有时也称为伺服类）。另外，服务器程序包含一个初始化部分（例如，在Java或C++中，这一部分在main方法中）。初始化部分负责创建和初始化至少一个包含在服务器上的远程对象，另外的远程对象可能应客户发出的请求而创建。初始化部分也可以用一个绑定程序（见“绑定程序”一段）注册它的一些远程对象。通常情况下，它会只注册一个远程对象，该对象可以用来访问其他所有对象。

客户程序包含它将调用的所有远程对象的代理类，它用一个绑定程序查找远程对象引用。

厂方法 我们早就注意到远程对象接口不能包括构造函数。这意味着远程对象不能通过对构造函数的远程调用来创建。远程对象或在初始化部分创建或在为该用途而设计的远程接口 179
方法中创建。术语厂方法有时用以指创建远程对象的方法，厂对象指具有厂方法的对象。任何远程对象要想能根据客户的需求创建新的远程对象，就必须在它的远程接口中提供用于此用途的方法。这样的方法称为厂方法，当然它们实际上也是普通的方法。

绑定程序 客户程序通常要有一种手段，用以获得服务器端至少一个远程对象的远程对象引用。例如，在图5-3中，对象A需要对象B的一个远程对象引用。分布式系统中的绑定程序是一种独立的服务，它维护着一张表，表里包含从文本名字到远程对象引用的映射。服务器根据这张表按照名字注册它们的远程对象，客户根据这张表查找这些远程对象。第17章包含对CORBA命名服务的讨论。Java绑定程序（即RMIregistry）将在5.5节Java RMI实例研究中简

单讨论。

服务器线程 一旦对象执行远程调用，该调用可能会导致其他远程对象的方法调用，因此可能需要一段时间才会返回。为了避免一个远程调用的执行延误另一个远程调用的执行，服务器一般为每个远程调用的执行分配一个独立的线程。这时，远程对象实现的设计者必须考虑并发执行状态产生的影响。

远程对象的激活 有些应用要求信息能长时间地保留。然而，让表示这一信息的对象无限期地运行在进程中是不切实际的，特别是因为并不是在所有的时间都要使用它们。为了避免在全部时间里运行管理这些远程对象的服务器造成的潜在的资源浪费，服务器应该在客户需要它们的任何时候启动，就像标准的TCP服务（如FTP）做得那样：*Inetd*服务会根据需要启动FTP。启动包含远程对象的服务器的进程由于以下几个原因被称为*激活器*。

当一个远程对象在一个运行的进程中可供调用时，就认为它是主动的；然而，如果它现在不是主动的但是可以被激活为主动的，就认为它是被动的。一个被动对象包括两个部分：

1. 方法的实现
2. 编码格式的状态

*激活*是指根据相应的被动对象创建一个主动对象，方法是创建被动对象类的一个新实例并根据存储的状态初始化它的实例变量。被动对象可以根据要求被激活，例如当它们需要被其他对象调用的时候。

*激活器*负责：

- 注册可以被激活的被动对象，包括记录服务器名称，而不是相应被动对象的URL或者文件名。

180

- 启动命名的服务器进程并激活进程中的远程对象。
- 记录已经激活的远程对象所在的服务器位置。

CORBA实例研究描述了它的激活器，即实现仓库。Java RMI在每个服务器计算机上使用一个激活器，它负责激活那台计算机上的对象。

持久对象存储 那些在进程两次激活之间仍然保证存活着的对象称为*持久对象*。持久对象一般由持久对象存储来管理，它在磁盘上以编码格式存储持久对象的状态，例如CORBA持久对象服务（见17.3节）和Persistent Java [Jordan 1996，java.sun.com IV]。

一般来说，持久对象存储会管理大量的持久对象，它们都存储在磁盘中，直至需要它们的时候。当它们的方法被其他对象调用的时候，它们就会被激活。激活一般设计为透明的，也就是说，调用者不能判断一个对象是已经在主存中，还是在其方法被调用之前必须被激活。在主存中不再需要的持久对象可以被钝化的。在大多情况下，为了容错，对象只要达到一个一致的状态，就保存在持久对象存储中。持久对象存储需要一个决定何时钝化对象的策略。例如，它可能会在激活对象的程序中为响应某个请求（如事务结束或程序退出时）而钝化对象。持久对象存储一般要对钝化进行优化，即只保存那些自上次保存后修改过的对象。

持久对象存储一般允许相关持久对象集具有可读的名字，例如路径名或者URL等。实际上，每个可读的名字都与持久对象集的根有关。

有两种方法可以判断一个对象是否是持久的：

- 持久对象存储维护一些持久根，任何通过持久根访问到的对象都被定义为持久的。这种方法被Persistent Java和PerDis[Ferreira *et al.* 2000]采用。采用这种方法的持久对象存储

使用无用单元回收器剔除不再从持久根可达的对象。

- 持久对象存储提供一些持久类，持久对象属于这些持久类的子类。例如，在Arjuna [Parrington *et al.* 1995]中，持久对象基于提供交易和恢复的C++类。不需要的对象必须被显式地删除。

有些持久对象存储，例如PerDis和Khazana [Carter *et al.* 1998]允许在多个用户的本地缓存中激活对象，而不是在服务器中。在这种情况下，就要求有缓存一致性协议。第16章将讨论多种一致性模型。

对象定位 4.3.3节描述了一种远程对象引用格式，它包含了创建远程对象的进程的因特网地址和端口号，用以作为保证惟一性的一种方式。这种远程对象引用格式也能作为远程对象的地址，只要该对象在余下的生命周期中保持在相同的进程里。但是有些远程对象在其整个生命周期里会存在于一系列不同的进程中，可能还会存在于不同的计算机中，在这种情况 181
下，远程对象引用不能用作地址。进行调用的客户同时需要一个远程对象引用和一个调用发送目的地址。

定位服务帮助客户根据远程对象引用定位远程对象，它使用了一个数据库，该数据库用于将远程对象引用映射成它们当前的大概位置。这个位置是大概的，因为对象可能已经从上次知道的位置迁移了。例如，Clouds系统 [Dasgupta *et al.* 1991] 和Emerald系统 [Jul *et al.* 1988] 使用了一个缓存/广播机制，其中每台计算机上定位服务的一个成员拥有一个小缓存，存放远程对象引用到位置的映射。如果远程对象引用在缓存里，就尝试用那个地址调用，但是如果对象已经移动了调用就会失败。为了定位一个已经移动的对象或位置不在缓存中的对象，系统要广播一条请求。改善该机制可以通过使用转发定位指针，定位指针含有关于对象的新位置的提示线索。

5.2.6 分布式无用单元回收

分布式无用单元回收器的目的是确保：如果一个本地或者远程对象引用还在分布式对象集合中的任何地方，那么该对象本身将继续存在，但是一旦没有任何对象引用它，该对象将被回收，并且它使用的内存也将被回收。

我们描述一下Java分布式无用单元回收算法，它与Birrell等 [1995]描述过的算法很相似。它基于引用计数。一旦一个远程对象引用进入一个进程，进程就会创建一个代理，只要还有对这个代理的需求，它就一直待在那里。对象生存的进程（它的服务器）会被通知关于客户上的新代理的信息。此后当客户不再有代理时，也应告知服务器。分布式无用单元回收器与本地无用单元回收器按如下的方式协作：

- 每个服务器进程为它的每个远程对象维护一组拥有远程对象引用的进程，例如，*B.holders*是具有对象*B*的代理的客户进程（虚拟机）的集合（在图5-6中，这个集合包括图示的客户进程）。这个集合可能放在远程对象表的一个附加列里。
- 当客户*C*第一次接收到远程对象*B*的远程引用时，它发出一个*addRef*(*B*)调用到远程对象的服务器并创建一个代理，服务器将*C*添加到*B.holders*。
- 当客户*C*的无用单元回收器注意到远程对象*B*的一个代理不再可达时，它发送一个*removeRef*(*B*)调用到相应的服务器，然后删除该代理，服务器从*B.holders*中删除*C*。
- 当*B.holders*为空时，服务器的本地无用单元回收器将回收被*B*占有的空间，除非还存在

*B*的一些本地持有者。

通过在远程引用模块之间采用至多一次调用语义的请求－应答通信可实现该算法，它不
182 要求任何全局同步。也要注意为无用单元回收算法所发送的额外调用不能影响到每个正常的RMI，它们只发生在创建和删除代理的时候。

有一种可能，在一个客户发出一个*removeRef*(*B*)调用的同时，另一客户端恰好发出*addRef*(*B*)调用。如果*removeRef*调用先到而此时*B.holders*为空，那么远程对象*B*可能会在*addRef*调用到来之前被删除。为避免这种情况，当传递远程对象引用的时候，如果*B.holders*集合是空的，就添加一个临时的条目直至*addRef*调用到达。

Java分布式无用单元回收算法通过使用下面的方法可以容忍通信故障。*addRef*和*removeRef*操作是幂等的。当*addRef*(*B*)调用返回一个异常（意味着执行过一次该方法或者根本不执行）时，客户不创建代理而是发出一个*removeRef*(*B*)调用。*addRef*不管成功与否，*removeRef*的结果都是正确的。*removeRef*失败的情况通过租借来处理。

Java分布式无用单元回收算法可以容忍客户进程的故障。为做到这点，服务器将它们的对象租借给客户一段有限的时间。借期起始于客户向服务器发出*addRef*调用，截止于时间到期或者客户向服务器发出一个*removeRef*调用。存储在服务器端的关于每个租借的信息包括客户虚拟机的标识符和租期。客户负责在租期期满之前向服务器请求更新租借。

Jini中的租借 Jini分布式系统包括一个租借规范[Arnold *et al.* 1999]，它可以用于一个对象为另一对象提供资源的各种情形，例如远程对象提供引用给其他对象。提供这种资源的对象可能会在用户不再对资源感兴趣或用户程序可能已经退出的情况下维护该资源。为了避免用复杂的协议判断使用资源的用户是否还有兴趣，资源只在一段有限时间内被提供。允许在一段时间内使用资源的授权称为租借。提供资源的对象会负责维护它直到租期结束。资源的用户负责在过期的时候请求更新它们的租约。

租期可以在授权者与租借者之间进行磋商，但是这不会发生在Java RMI所使用的租借中。表示租借的对象实现*Lease*接口，该接口包含关于租期的信息和能令租借更新或取消的方法。授权者在提供一种资源给另一对象的时候返回一个*Lease*的实例。

5.3 远程过程调用

远程过程调用与远程方法调用很类似，都是客户程序调用另一运行在服务器进程中的过程。服务器也可能是其他服务器的客户，从而形成RPC链。正如在本章简介中提到的，服务器进程在它的服务接口中定义可远程调用的过程。RPC，就像RMI，可被实现为5.2.4节讨论的任一调
183 用语义——通常会选择至少一次或者至多一次。RPC一般会在请求－应答协议之上实现，就像在4.4节中讨论的那样，4.4节做了一些简化，在请求消息中省略了对象引用。除了*ObjectReference*域被省略之外，请求和应答消息的内容与图4-13中为RMI描述的那些内容相同。

支持RPC的软件如图5-7所示。因为远程过程调用不关心对象和对象引用，所以除了不需要远程引用模块之外，它与图5-6非常类似。访问服务的客户为服务接口中的每个过程包含一个存根过程。存根过程的作用与代理的作用类似。它表现得就像客户端的本地过程，但它并不是执行调用，而是将过程标识和参数编码到请求消息中，通过它的通信模块发送给服务器。当应答消息到达的时候，它解码该结果。服务器进程包含一个调度程序和一个服务器存根过程，以及与服务接口中每个过程相对应的一个服务过程。调度程序根据请求消息中的

过程标识选择一个服务器存根过程。服务器存根过程就像一个骨架方法，由它来解码请求消息中的参数、调用相应的服务过程和编码应答消息中的返回值。服务过程实现服务接口中的过程。

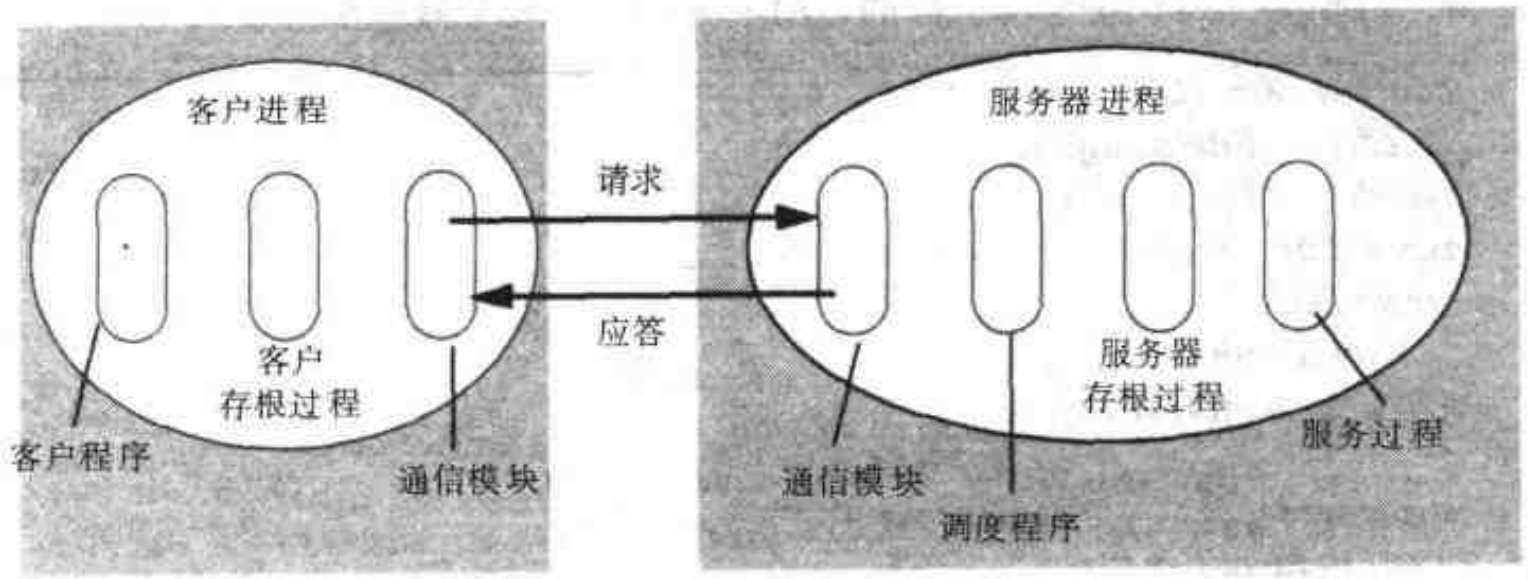


图5-7 RPC中客户和服务器存根过程的作用

客户存根过程和服务器存根过程以及调度程序能由接口编译器根据服务的接口定义生成。

Sun RPC实例研究

RFC 1831 [Srinivasan 1995a]描述了Sun RPC，它是为Sun NFS网络文件系统中客户－服务器通信设计的。Sun RPC有时也称为ONC（开放网络计算）RPC。它作为各种Sun和其他UNIX操作系统的一部分提供，并且也可以安装在其他NFS中。远程过程调用可以基于UDP协议实现，也可以基于TCP协议实现。当Sun RPC采用UDP协议时，请求和应答消息的长度被限制在一定长度内，理论上可以到64KB，但在实际中更常见的是8KB或9KB。它使用至少一次 184
调用语义。广播型RPC是一个选项。

Sun RPC系统提供了一种称为XDR的接口语言和一个可以用于C编程语言的接口编译器*rpcgen*。

接口定义语言 Sun XDR语言，原本设计用于说明外部数据表示，现在扩展成为一种接口定义语言。通过指定过程定义和支持类型定义，XDR可用于定义Sun RPC的服务接口。与CORBA IDL或Java相比，它的表示方法相当简单。特别是在以下几个方面：

- 大多数语言允许指定接口名，但Sun RPC不是这样，它提供一个程序号和一个版本号。程序号可以从授权中心获得，以保证每个程序都有惟一的号码。当一个过程的标记名称改变时，版本号也跟着改变。程序号和版本号都在请求消息里传递，以便客户和服务器能验证它们使用的是相同的版本。
- 一个过程定义指定了一个过程标记名称和一个过程号。过程号用作请求消息中的过程标识符。（接口编译器可能会生成过程标识符。）
- 只允许单个输入参数。因此，需要多个参数的过程必须把参数作为一个结构的组成部分。
- 通过单个结果返回过程的输出参数。
- 过程标记名称由结果类型、过程名和输入参数的类型组成。返回结果和输入参数的类型可以指定为单个值，也可以指定为包含几个值的一个结构。

例如，图5-8中的XDR定义定义了用于读文件和写文件的两个过程的接口。它的程序号是

9999，版本号是2。*READ*过程（第2个标记行）将一个带有3个域的结构（包括文件标识符、文件中的位置和要求的字节数）作为输入参数。它的结果也是一个结构，包括返回的字节数和文件数据。*WRITE*过程（第1个标记行）没有结果。*WRITE*和*READ*过程被赋予了号码1和2。号码0保留给空过程，空过程是自动生成的，用于测试一个服务器是否可用。

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
 version VERSION {
    void WRITE(writeargs)=1;                    1
    Data READ(readargs)=2;                      2
    }=2;
} = 9999;
```

图5-8 用Sun XDR描述的文件接口

接口定义语言提供了用于定义常量、类型预定义（typedef）、结构、枚举类型、联合和程序的表示法。类型预定义、结构、枚举类型使用C语言的语法。接口编译器*rpcgen*能根据接口定义生成以下成分：

- 客户存根过程
- 服务器*main*过程、调度程序和服务器存根过程
- 用于调度程序、客户与服务器存根过程的XDR编码和解码过程

绑定 Sun RPC在每台计算机一个熟知的端口号上运行一个称为端口映射器的本地绑定服务。端口映射器的每个实例记录着正在本地运行的每项服务所使用的程序号、版本号和端口号。当服务器启动时，它在本地端口映射器中注册它的程序号、版本号和端口号。当客户启动时，它通过发送指定程序号和版本号的远程请求给服务器主机上的端口映射器，从而找到服务器上的端口。

当一个服务有多个运行在不同计算机上的实例时，各个实例可以使用不同的端口号接收客户的请求。如果一个客户需要组播一个请求给所有使用不同端口号的服务实例，那么它不

能使用直接广播达到目的。解决办法是客户广播指定程序和版本号的请求到所有的端口映射器以实现对远程过程调用的组播。每个端口映射器判断是否有合适的本地服务程序，如果有，就向它转发所有这样的调用。

认证 Sun RPC请求和应答消息提供了一些附加域，使得能够在客户和服务器之间传输认证信息。请求消息中包含了正在运行客户程序的用户的证书，例如，按UNIX的认证风格，证书包括用户的*uid*和*gid*。访问控制机制能构建在认证信息之上，该认证信息可以通过第二个参数用于服务器过程。服务器程序负责实施访问控制，根据认证信息决定是否执行每个过程调用。例如，如果服务器是一个NFS文件服务器，那么它要验证用户是否具有足够的权限执行所请求的文件操作。

185 ~ 186

Sun RPC可以支持几种不同的认证协议，它们包括：

- 不认证
- 如上文描述的UNIX的认证风格
- 为标记RPC消息创建共享关键字的认证风格
- Kerberos认证风格（见第7章）

RPC头部的一个域指明它使用的是何种认证风格。

保障安全性的一种更通用的方法见RFC 2203中的描述[Eisler *et al.* 1997]。它提供RPC消息和认证的保密性和完整性。它允许客户和服务器协商安全上下文，在该上下文中或者不提供安全性或者在有安全性要求时提供消息完整性或者消息隐私性或者两者都提供。

客户程序和服务器程序 关于Sun RPC的深入材料可以在www.cdk3.net/RMI中找到。它包括与图5-8中所定义的接口相对应的客户和服务器程序示例。

5.4 事件和通知

使用事件这一概念主要是希望描述一个对象能对发生在另一对象上的改变做出反应。事件通知在本质上是异步的，并取决于它们的接收者。特别是，在交互式应用中，用户在对象上执行的动作可以看作是事件，例如通过鼠标操作一个按钮或者用键盘在文本框中输入文本，事件改变了维护应用状态的对象。每次状态改变，就要通知负责显示当前状态视图的对象。

基于事件的分布式系统扩展了本地事件模型，它允许多个不同位置的对象接到通知，从而被告知某个对象上的事件正在发生。它们使用发布-订阅模型，生成事件的对象发布事件的类型，其他对象可以观察到它。对象如果想从一个已经发布了事件的对象上接收通知，就要订阅它们感兴趣的事件类型。不同的事件类型指的是由兴趣对象执行的不同方法。表示事件的对象称为通知。可以存储通知，也可以在消息中发送通知，可以在各种不同的事物中查询和应用通知。当一个发布者经历一个事件时，曾经表示过对该类事件有兴趣的对象就会接收到通知。订阅某一类型事件也称为对那类事件注册兴趣。

187

事件和通知能广泛地用于各种不同的应用中，例如传送一个图形并添加到绘图程序中，对文档的变更，或者是一些事实，例如某个人已经进入或离开了一个房间，或者是一件设备或一本有电子标签的书放在了一个新的位置上。活动标记和嵌入式设备（见2.2.3节）的使用使得后面两个例子成为可能。

基于事件的分布式系统具有两个主要特征：

- 异构性 当事件通知作为分布式对象间一种通信方式时，分布式系统中的组件虽然原本不

是设计用于互操作的，现在也可以让它们一起工作。所有要做的事情就是生成事件的对象发布它们所提供的事件类型，而其他对象订阅事件并提供接收通知的接口。例如，Bates等[1996]描述了基于事件的系统能怎样用于连接因特网中的异构组件。他们描述了一个系统，其中应用可以知道用户的位置和活动，例如使用计算机、打印机或者具有电子标签的书。他们想像到未来在家庭网络的环境中，会有这样的命令："如果孩子们回家，就打开中央暖气"。

- 异步性 生成事件的对象异步地将通知传输给所有向其订阅过该事件的对象，避免了发布者需要与订阅者同步，也就是说发布者和订阅者需要去耦合。Mushroom [Kindberg *et al.* 1996]是一个基于事件的分布式系统，它被设计用于支持协同工作，该系统中的用户接口显示表示用户的对象和信息对象（例如在被称为网络空间的共享工作区内的文档与记事本）。在当前该处用户的计算机上复制每处空间的状态。事件用于描述对象和用户兴趣焦点的改变。例如，一个事件可能说明某个用户已经进入或者离开了一个地方，或者已经执行了一个对象上的某个动作。与某类事件相关的对象的每个副本订阅这类事件，并接收这类事件发生时的通知。但是订阅者与经历事件的对象是去耦合的，因为不同的用户在不同的时间活动。

事件的作用将在下面交易所的例子中阐述：

简单的交易所系统 考虑一个简单的交易所系统，它的任务是允许交易者使用计算机查看他们当前正在交易的股票的最新市场行情。单只已命名的股票的市场价格通过一个具有几个实例变量的对象来表示。信息以更新代表股票的对象的一部分或者全部实例变量的形式从几个不同的外部源到达交易所，然后由我们称为信息提供者的进程进行汇集。交易者一般只对他们关注的股票感兴趣。可采用两个完成不同任务的进程为交易所系统建模：

- 一个信息提供者进程持续不断地从单个外部源接收新的交易信息，并应用到适当的股票对象上。对股票对象的每次更新被认为是一个事件。经历这一事件的股票对象通知所有订阅了该股票的交易者。每个外部源都将会有一个单独的信息提供者进程。
- 一个交易者进程创建一个对象，用来表示用户请求显示的每只股票。这个本地对象订阅代表由相关信息提供者提供的那只股票的对象。然后它接收发给它的通知中的所有信息，并将这些信息显示给用户。

188

通知的通信方式如图5-9所示。

事件类型 一个事件源能创建一个或多个不同类型的事件。每个事件都有指定该事件信息的属性，例如生成该事件的对象的名称或标识符、操作、事件的参数和时间（或者一个序列号）。类型和属性均用于事件的订阅和通知。当订阅一个事件的时候，要指定事件的类型，有时候还要加上关于属性值的标准。一旦与属性匹配的事件发生，就会通知对该事件感兴趣的一方。在交易所系统例子中，存在一类事件（更新股票的信息到达），属性可以指定股票的名字、它的现价、最新的涨幅或落幅。例如，交易者可能会指明他们对具有某个名字的股票相关的所有事件都感兴趣。

5.4.1 分布式事件通知的参与者

图5-10显示了一个体系结构，它说明了参与到基于事件的分布式系统中的对象的作用。我们的描述来源于因特网上Rosenblum和Wolf [1997]关于事件和通知的论文。设计该体系结构用于发布者和订阅者之间的去耦合，以便允许发布者独立于它们的订阅者进行开发，并且尽可能

地限制订阅者施加于发布者的工作。其主要部分是事件服务，它维护一个已发布事件与订阅者兴趣的数据库。在一个兴趣对象上的事件发布在事件服务上。订阅者通知事件服务它们感兴趣的事件类型。当一个事件在一个兴趣对象上发生的时候，就发送一个通知到该类事件的订阅者。

参与对象的作用如图5-10所示：

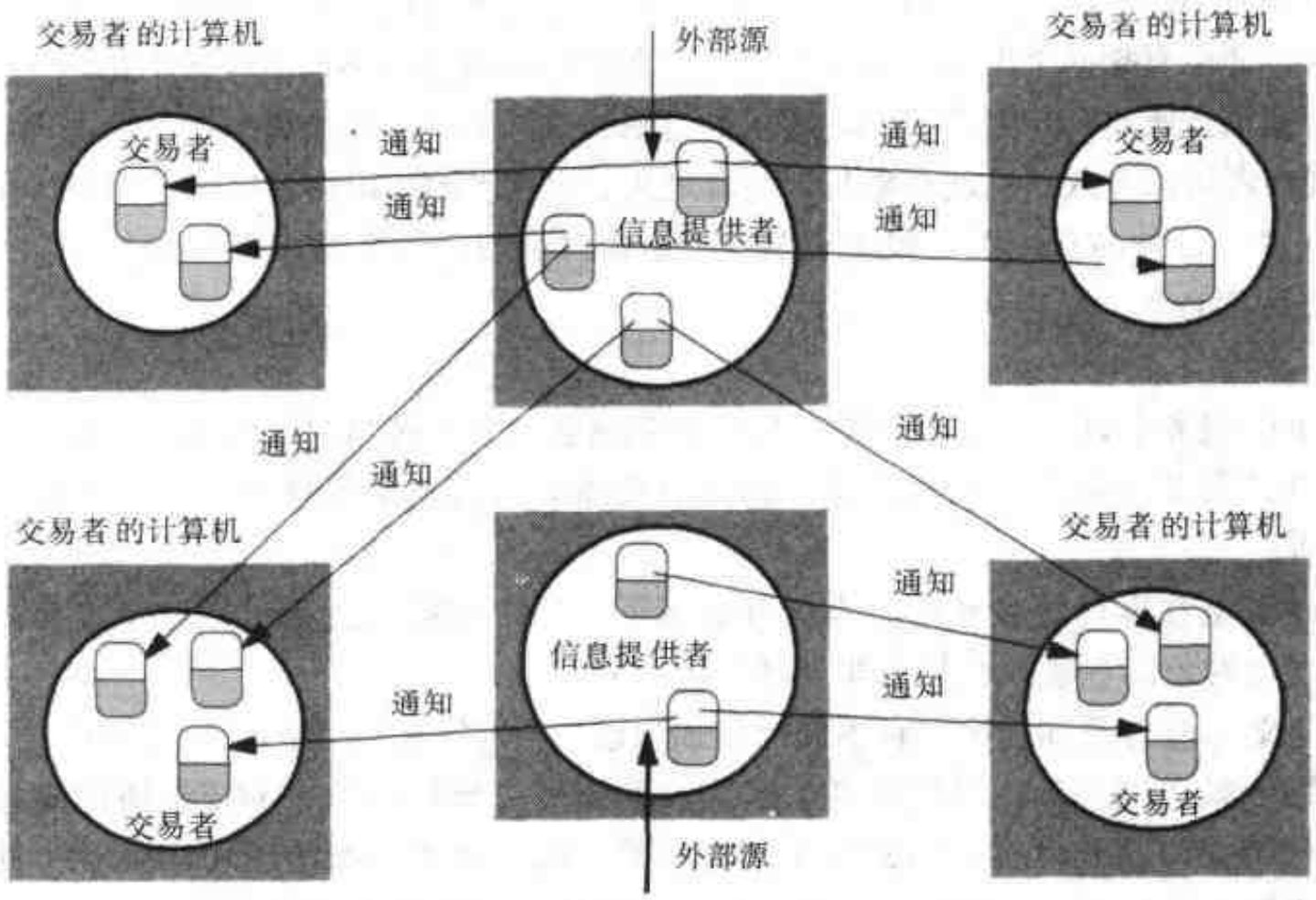


图5-9 交易所系统

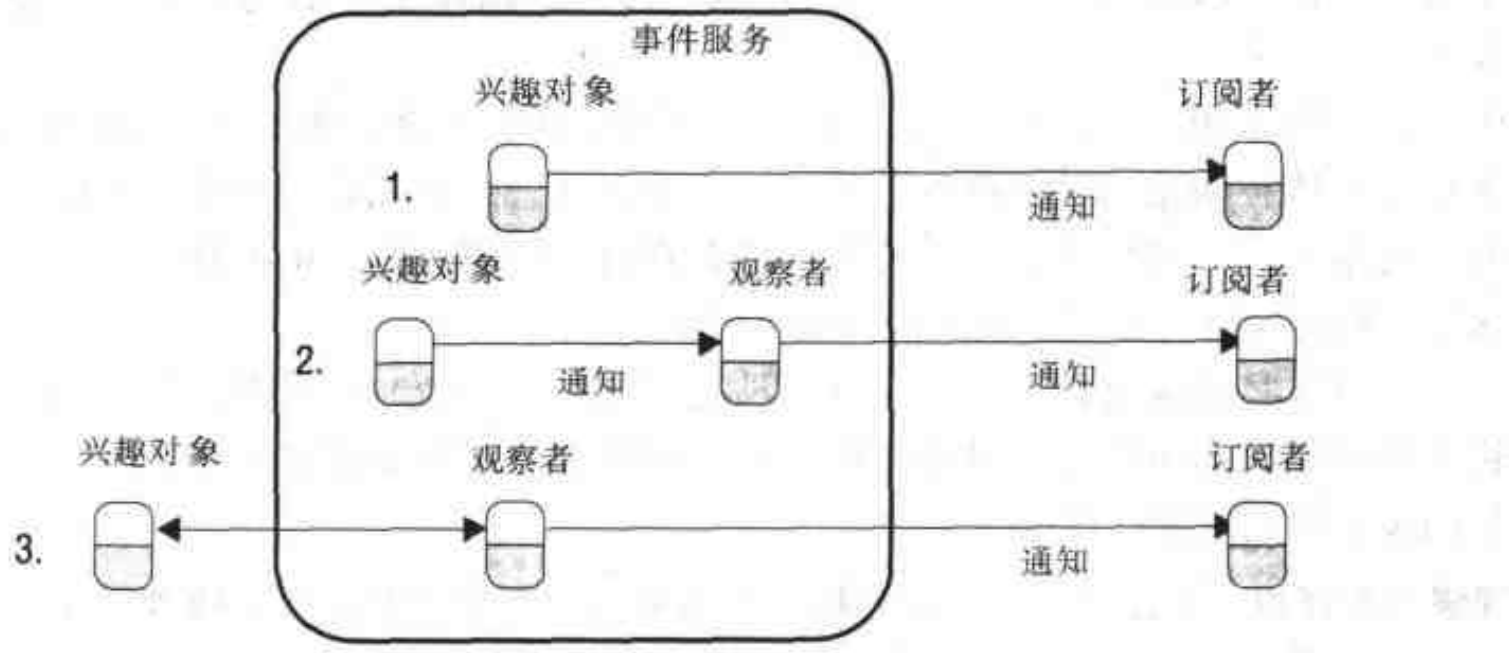


图5-10 分布式事件通知的体系结构

- 兴趣对象 这是一个由于其操作被调用而经历状态改变的对象。它的状态改变可能令其他对象感兴趣。这一描述考虑到了诸如戴着活动标记的人进入房间这样的事件，在这种情形下，房间是兴趣对象，而操作是将新来的人的信息添加到进入房间的人员的记录中。兴趣对象如果传递通知，就被认为是事件服务的一部分。
- 事件 事件发生在兴趣对象上，作为方法执行完成的结果。
- 通知 通知是一个对象，它包含关于事件的信息。一般来说，它包含事件类型及其属性，

通常包括兴趣对象的标识符、调用的方法、发生时间或者一个序列号。

189 ~ 190

- 订阅者　订阅者是一个对象，它订阅另一对象上的某些类型事件，并接收关于这些事件的通知。
- 观察者对象　观察者的主要目的是将兴趣对象和其订阅者去耦合。一个兴趣对象可能会有许多具有不同兴趣的不同订阅者。例如，各个订阅者可能在它们感兴趣的事件类型方面不相同，那些有相同类型需求的订阅者可能在它们感兴趣的属性值方面不相同。如果兴趣对象必须执行所有用于判断订阅者需求的逻辑，那么兴趣对象就可能过于复杂化。一个或多个观察者可以介入到兴趣对象与订阅者之间。观察者的作用将在后续段落里详细讨论。
- 发布者　这种对象声明它将生成某种类型事件通知。一个发布者可能是一个兴趣对象或一个观察者。

图5-10给出了3种情形：

1. 在事件服务中有一个兴趣对象，但没有观察者。它直接将通知发送到订阅者。
2. 在事件服务中有一个兴趣对象，而且有观察者。兴趣对象通过观察者将通知发送到订阅者。
3. 兴趣对象在事件服务外边。在这种情形下，一个观察者会为了知道事件何时发生而查询兴趣对象。观察者会将通知发送给订阅者。

传递语义　能为通知提供多种不同的传递保证——选择哪一个取决于应用需求。例如，如果IP组播用于发送通知给一组接收者，那么故障模型将与4.5.1节中对IP组播的描述有关，它不保证某些特定的接收者会接收到某个通知消息。因为可能下次更新可能会继续处理，所以这对某些应用已经足够了，例如在因特网游戏中传输玩家的最新状态。

然而，其他应用会有更强的需求。考虑交易所应用：为了公平对待对某一只股票感兴趣的交易者，我们要求相同股票的所有交易者接收到相同的信息。这意味着应该使用可靠的组播协议。

在上面提到的Mushroom系统中，能将关于对象状态改变的通知可靠地传输到服务器上，由服务器负责维护对象的最新副本。然而，通知也可以以不可靠的组播方式传送到用户计算机的对象副本上。如果后者丢失了通知，那么它们能从服务器上获得对象状态。当应用需要时，通知会被排序后再被可靠地传输到对象副本上。

一些应用有实时需求，这些应用包括核电厂系统、医院病人监护系统。对于这些应用，可能需要设计提供实时保证和可靠性以及系统中具有排序功能的组播协议，其排序功能是为了满足同步分布式系统特性。

观察者的作用　尽管可以直接从兴趣对象传输通知到接收者，但是处理通知的任务还是可能被分解到不同作用的观察者进程中。我们描述一些例子：

191

- 转发　转发观察者可以代表一个或多个兴趣对象完成所有将通知发送给订阅者的工作。一个兴趣对象需要做的所有事情就是发送一个通知给转发观察者，然后由它继续自己正常的工作。为了使用转发观察者，兴趣对象要将关于其订阅者兴趣的信息传输给那个转发观察者。
- 通知过滤　观察者可以使用过滤，以便根据对每条通知内容的判断减少接收到的通知数量。例如，一个事件可能与从银行账户上取钱有关，但是接收者只对那些大于100美元的交易感兴趣。

- 事件模式　当对象订阅一个兴趣对象上的事件的时候，它们可以说明自己感兴趣的事件的模式。一种模式指明了几个事件之间的一种关系。例如，一个订阅者可能对“没有存过一次款却从银行账户中取过三次钱”感兴趣。一种类似的需求是关联各种兴趣对象上的事件，例如，只有在一定数量的对象生成了这类事件的时候才通知订阅者。
- 通知邮箱　在有些情形下，通知需要延迟到潜在的订阅者准备接收它们时。例如，订阅者联系不上，或者对象已经钝化而需要再次激活的时候。一个观察者可以扮演通知信箱的角色，由它代表订阅者接收通知，只有等到订阅者准备好接收的时候才传递它们（通过单独的批次传递）。订阅者应该可以根据自身的需要打开或关闭这种传输选项。订阅者可以在向某兴趣对象注册的时候设置一个通知信箱，同时要将通知信箱指定为通知发送到的地方。

5.4.2 Jini分布式事件规范

由Arnold等[1999]描述的Jini分布式事件规范，允许一个Java虚拟机（JVM）中的潜在订阅者订阅和接收另一JVM（通常在另一计算机上）中的兴趣对象的事件通知，可以将若干观察者插入到兴趣对象与订阅者之间。Jini分布式事件规范里的主要对象包括：

- 事件生成者　事件生成者是一个允许其他对象向它订阅事件并生成通知的对象。
- 远程事件监听者　远程事件监听者是一个能接收通知的对象。
- 远程事件　远程事件是一个按值方式传递给远程事件监听者的对象。远程事件等价于我们所说的通知。
- 第三方代理　第三方代理可以插入到兴趣对象和订阅者之间。它们等价于前面介绍的观察者。

一个对象订阅事件，它首先通知事件生成者事件的类型，然后将远程事件监听者指定为发送通知的目标。 192

Java RMI用于从事件生成者向订阅者发送通知，它可能会借助于一个或多个第三方代理。设计者规定事件监听者应尽快应答通知调用，以免延误事件生成者。它们在返回之后处理通知。Java RMI也用于订阅事件。通过下列接口和类提供Jini事件：

- *RemoteEventListener*　这个接口提供一个称为*notify*的方法。订阅者和第三方代理要实现*RemoteEventListener*接口，以便它们可以在调用*notify*方法的时候接收通知。*RemoteEvent*类的一个实例表示一个通知，并且作为参数传递到*notify*方法。
- *RemoteEvent*　这个类的实例变量具有：
  - 对一个发生事件的生成者的引用。
  - 一个事件标识符，它指定事件生成者上事件的类型。
  - 一个序列号，它应用于指定类型的事件。序列号应该随着事件发生而递增。它使接收者可以根据一个给定的来源对某些类型的事件进行排序或避免两次应用同一个事件。
  - 一个编码对象，它在接收者订阅指定类型事件时提供，供接收者使用。它通常拥有一些接收者需要的信息，以标识这个事件并对事件的发生做出反应。例如，在通知它时，它会包含将要运行的事件的闭包。
- *EventGenerator*　该接口提供一个称为*register*的方法。事件生成者实现*EventGenerator*接口，其*register*方法用于订阅事件生成者上的事件。*register*的参数指定了：
  - 一个事件标识符，用于指定事件的类型。

- 一个编码对象，随着每个通知退还。
- 一个事件监听者对象的远程引用，即通知发送到的地方。
- 一个请求的租期。租期指定了订阅者要求租借的时间段，但是实际许可的租期由*register*的结果返回。对订阅的时间限制避免了事件生成者保留无效事件订阅的问题。租期过期了，可以续订。

根据Jini规范，*EventGenerator*接口就是这样一种接口的例子，订阅者可以用它在兴趣对象上注册兴趣事件。有些应用可能要求一个不同的接口。

第三方代理 插入到事件生成者和订阅者之间的第三方代理的作用很多，包括上述的那些。

193

在最简单的情形下，一个订阅者向事件生成者注册对某类事件的兴趣，并将自己指定为远程事件监听者。这对应于图5-10中的第一种情况。

第三方代理可以由事件生成者或者订阅者设置。

事件生成者可能会在它自己与订阅者之间插入一个或多个第三方代理。例如，每台计算机上的事件生成者可以使用一个共享的第三方代理来负责可靠的通知传递。

为了达到所要求的传递策略，订阅者可以创建一个第三方代理链，然后向事件生成者注册兴趣，将第三方代理链的第一个第三方代理指定为通知发送到的地方。例如，在订阅者做好接收通知的准备之前，它可以安排由第三方代理存储发送给它的通知，第三方代理可能还会负责更新租借。

CORBA事件服务将在17.3.2节中讨论。

5.5 Java RMI实例研究

Java RMI扩展了Java的对象模型，它在Java语言中提供对分布式对象的支持。特别是，它允许对象使用与本地调用相同的语法调用远程对象上的方法。而且，类型检查也等效地应用到本地调用和远程调用。然而，发送远程调用的对象知道它的目标对象是远程的，因为它必须处理*RemoteException*；并且远程对象的实现者也知道它是远程的，因为它必须实现*Remote*接口。尽管分布式对象模型以一种自然的方式集成到了Java中，但是，由于调用者和目标对象彼此是远程的，所以其参数传递语义是不相同的。

用Java RMI进行分布式应用的编程相对来说较为容易，因为它是一个单语言系统，其远程接口用Java语言定义。如果使用一个多语言系统如CORBA，程序员就需要学习IDL，并理解它如何映射到实现语言中。然而，即使在一个单语言系统中，远程对象的程序员仍然必须考虑这些对象在并发环境下的行为。

下面我们给出一个远程接口的例子，然后讨论与该例子有关的参数传递语义，最后，我们讨论类的下载和绑定程序。实例研究的第2小节讨论如何为该例子接口创建客户程序和服务器程序。第3小节着重于Java RMI的设计和实现。对于Java RMI的全部细节，参见关于远程调用的教程[java.sun.com I]。

在这个实例研究以及第17章中的CORBA实例研究中，我们均使用一个共享白板作为例子。这是一个允许一组用户共享一个绘图区域的分布式程序，该绘图区域包含图形对象，例如矩形、线和圆等，每个图形由一个用户绘制。服务器为了维护绘图当前的状态，为客户提供一个操作，可以把用户最新绘制的图形告诉服务器，并保留一份它接收到的所有图形的记录。服务器也允

许客户通过轮询服务器的方式获取由其他用户绘制的最新图形。服务器具有一个版本号（一个整数），每当新图形出现时，就递增版本号，并将新版本号赋予新图形。服务器还提供有关操作，允许客户查询它的版本号和每个图形的版本号，以避免客户取到它们已经有的图形。 194

在Java RMI中的远程接口 远程接口通过扩展一个在*java.rmi*包中提供的称为*Remote*的接口来定义。方法必须抛出*RemoteException*，但是也可以抛出应用特有的异常。图5-11给出两个远程接口*Shape*和*ShapeList*。在这个例子中，*GraphicalObject*是一个拥有图形对象状态（例如它的类型、位置、外接矩形、线条颜色和填充颜色）的类，该类提供存取和更新其状态的操作。*GraphicalObject*必须实现*Serializable*接口。先来考虑*Shape*接口：*getVersion*方法返回一个整数，而*getAllState*方法返回*GraphicalObject*类的一个实例。现在考虑*ShapeList*接口：它的*newShape*方法将*GraphicalObject*类的一个实例作为参数传递，并将一个带有远程接口的对象（即一个远程对象）作为结果返回。要注意重要的一点，普通对象和远程对象都能作为远程接口中的参数和结果。后者总是以它们的远程接口名来表示。在下一个段落中，我们讨论普通对象和远程对象作为参数和结果是如何传递的。

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;                1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;             2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

图5-11 Java 远程接口*Shape*和*ShapeList*

传递参数和结果 在Java RMI中，方法的参数被假定为输入型的参数，而方法的结果是一个输出型参数。4.3.2节描述了Java序列化，它用于编码Java RMI中的参数和结果。任何可序列化的对象，即实现了*Serializable*接口的对象，都能作为Java RMI中的参数或结果来传递。所有的简单类型和远程对象都是可序列化的。在必要的时候，作为参数和结果值的类可以由RMI系统下载给接收者。

- **传递远程对象** 当将参数类型或者结果值类型定义为远程接口的时候，相应的参数或结果总是作为远程对象引用传递。例如在图5-11中第2个标记行，*newShape*方法的返回值就定义为远程接口*Shape*。当接收到一个远程对象引用时，它就在它所指的远程对象上进行RMI调用。 195
- **传递非远程对象** 按值方式拷贝和传递所有可序列化的非远程对象。例如，在图5-11（第2个标记行和第1个标记行）中，*newShape*的参数和*getAllState*的返回值都是*GraphicalObject*类型，它是可序列化的并且按值方式传递。当按值方式传递一个对象时，在接收者的进程中就要创建一个新对象。这个新对象的方法可以在本地调用，这可能导致它的状态与发送者进程中原来的对象状态不同。

这样，在我们的例子中，客户使用*newShape*方法传递给服务器一个*GraphicalObject*实例；服

务器创建一个包含*GraphicalObject*状态的*Shape*类型的远程对象，并返回一个它的远程对象引用。在远程调用中的参数和返回值用4.3.2节描述的方法被序列化到一个流中，并具有下列改变：

1. 一旦一个实现*Remote*接口的对象被序列化了，它就被它的远程对象引用代替，该远程对象引用包含了它（远程对象）的类名。
2. 任何一个对象被序列化时，它的类信息就加注了该类的地址（作为一个URL），使该类能由接收者下载。

类的下载 Java允许类从一个虚拟机下载到另一个虚拟机。这与按远程调用方式通信的分布式对象密切相关。我们已经看到，非远程对象按值方式传递，而远程对象按引用方式传递RMI的参数和结果。如果接收者还没拥有按值方式传递的对象类，那么它就会自动下载它的代码。类似地，如果远程对象引用的接收者还没拥有代理类，那么也会自动下载代理类的代码。这样做有以下两点好处：

1. 不必为每个用户在他们的工作环境中保留相同类的集合。
2. 一旦添加了新类，客户程序和服务器程序都能透明地使用它们的实例。

例如，考虑白板程序并假设它的*GraphicalObject*初始实现没有考虑到文本。那么一个带有文本对象的客户就会实现*GraphicalObject*的一个子类，用以处理文本，并将其实例作为*newShape*方法的参数传递到服务器上。在那以后，其他客户能够使用*getAllState*方法获取这个实例。新类的代码将自动地从第一个客户下载到服务器，然后再下载到其他需要的客户。

RMIregistry RMIregistry是Java RMI的绑定程序。RMIregistry的实例必须运行在每个包含远程对象的服务器计算机上。它维护着一张表，将文本格式的、URL风格的名字映射到包含在该计算机上的远程对象引用。它通过*Naming*类的方法来存取，*Naming*类的方法以一个
196 URL格式的字符串作为参数：

//computerName : port/objectName

其中，*computerName*和*port*指向RMIregistry的地址。如果它们被省略，就认为是本地计算机和默认端口。RMIregistry的接口提供如图5-12所示的方法，这些方法中没有列出异常，实际上所有的方法都可以抛出*RemoteException*。该项服务不是系统范围的绑定服务，客户必须将它们的*lookup*查询定向到特定的主机。

void rebind (String name , Remote obj)

此方法由服务器用于以名字注册一个远程对象的标识符，如图5-13第3标记行所示。

void bind (String name , Remote obj)

服务器可以选择使用此方法以名字注册一个远程对象，但是如果该名字已经绑定到一个远程对象引用上，它会抛出异常。

void unbind (String name , Remote obj)

此方法删除一个绑定。

Remote lookup (String name)

客户可以使用此方法通过名字查找一个远程对象，如图5-15第1标记行所示。它返回一个远程对象引用。

String [] list ( )

此方法返回一个*String*数组，该数组包含绑定到注册表中的名字。

图5-12 Java RMIregistry的*Naming*类

5.5.1 创建客户程序和服务器程序

本节以图5-11中所示的远程接口*Shape*和*ShapeList*为例，概述创建使用*Remote*接口的客户程序和服务器程序的必需步骤。服务器程序是实现*Shape*和*ShapeList*这两个接口的白板服务器的简化版本。我们描述一个简单的轮询客户程序，然后介绍回调技术，这一技术可以避免使用轮询服务器。本节阐述的这几个类的完整版本可以参见cdk3.net/rmi。

服务器程序 服务器是一个白板服务器：它把每种图形表示成一个实现*Shape*接口的远程对象，它拥有图形对象的状态和它的版本号；它以一个实现*ShapeList*接口的远程对象代表它的图形集，并把图形集存放在向量中。

为实现它的每个远程接口，服务器包括一个*main*方法和一个伺服类。服务器的*main*方法创建*ShapeListServant*的一个实例，并将其绑定到RMIregistry中的一个名字上，如图5-13所示（第1标记行和第2标记行）。注意，与名字绑定的值是一个远程对象引用，其类型是其远程接口*ShapeList*的类型。两个伺服类是*ShapeListServant*（它实现*ShapeList*接口）和*ShapeServant* 197
（它实现*Shape*接口）。图5-14给出了*ShapeListServant*类的轮廓。注意*ShapeListServant*（第1标记行），与许多伺服类一样，扩展了一个名为*UnicastRemoteObject*的类，该类提供远程对象，这些远程对象的生存时间与创建它们的进程一样长。

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();                    1
            Naming.rebind("Shape List", aShapeList );                         2
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

图5-13 带有*main*方法的Java类 *ShapeListServer*

伺服类中远程接口方法的实现是直截了当的，因为它们可以在不考虑任何通信细节的情况下完成。考虑图5-14中的*newShape*方法（第2标记行），因为它允许客户请求创建一个远程对象，因此被称为厂方法。它使用*ShapeServant*的构造函数，创建一个新的远程对象，该对象包含作为参数传递的*GraphicalObject*和版本号。*newShape*的返回值的类型是*Shape*（由新的远程对象实现的接口）。在返回之前，*newShape*方法把新的图形添加到包含图形列表的向量中（第3标记行）。

服务器的*main*方法需要创建一个安全性管理者，以使Java安全性能为RMI服务器提供合适的保护。有一个默认的安全性管理者，称为*RMISecurityManager*，它保护本地的资源，以确保从远程站点载入的类不会对像文件这样的资源造成影响，但是它允许程序提供它自己的类装载程序和使用反射。如果一个RMI服务器没有设置安全性管理者，代理和类就只能从本地的类路径装载，以保护程序不受下载的代码（作为远程方法调用的一个结果）的干扰。

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
     private Vector theList;          // contains the list of Shapes                1
     private int version;
     public ShapeListServant()throws RemoteException{...}
     public Shape newShape(GraphicalObject g) throws RemoteException {          2
          version++;
          Shape s = new ShapeServant( g, version);                              3
          theList.addElement(s);
          return s;
     }
     public  Vector allShapes()throws RemoteException{...}
     public int getVersion() throws RemoteException { ... }
}
```

图5-14 Java类*ShapeListServant*实现*ShapeList*接口

客户程序 *ShapeList*服务器的一个简化的客户程序如图5-15所示。任何客户程序的开始都需要使用绑定程序查找远程对象引用。这个客户程序设置了一个安全性管理者，然后使用RMIregistry的*lookup*操作（第1标记行）为远程对象查找一个远程对象引用。在获取了一个初始的远程对象引用后，客户继续发送RMI给那个远程对象，或者根据应用的需要发送给在执行期间发现的其他对象。在这个例子中，客户调用远程对象的*allShapes*方法（第2标记行），并接收一个存储在当前服务器中的所有图形的远程对象引用的向量。如果客户正在实现一个白板显示，那么它将使用服务器上*Shape*接口中的*getAllState*方法获取向量中的每个图形对象，并将它们显示在窗口里。每次用户绘制完图形对象，它就调用服务器中的*newShape*方法，将新的图形对象作为参数传递。客户会记录服务器上的最新版本号，它还不时调用服务器上的*getVersion*方法查找是否存在某些新的图形已经由别的用户添加进去了。如果有这样的新图形，它将检索出来并加以显示。

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
   public static void main(String args[]){
      System.setSecurityManager(new RMISecurityManager());
      ShapeList aShapeList = null;
      try{
           aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");        1
           Vector sList = aShapeList.allShapes();                               2
      } catch(RemoteException e) {System.out.println(e.getMessage());
      }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

198
~
199

图5-15 *ShapeList*的Java客户程序

回调 回调所蕴含的想法是，客户端不必为查明某个事件是否已经发生而轮询服务器，而是当那一事件发生时，由服务器通知其客户。术语回调用于指服务器为某一事件通知客户的动作。在RMI中按如下方式实现回调：

- 客户创建一个远程对象，该对象实现一个接口，接口中包含一个供服务器调用的方法。我们称该对象为回调对象。
- 服务器提供一个操作，让感兴趣的客户通知服务器它们各自的回调对象的远程对象引用，服务器将这些记录在一张列表中。
- 一旦兴趣事件发生了，服务器就调用感兴趣的客户。例如，白板服务器会在添加一个图形对象时调用其客户。

回调的使用避免了客户轮询服务器上的兴趣对象，但它有如下缺点：

- 服务器的性能会因为经常性的轮询而降低。
- 客户不能及时通知用户已经做了更新。

回调也有其自身的问题：首先，服务器需要有客户回调对象的最新列表，但是客户并不总能在它退出之前通知服务器，这会导致服务器中的列表不正确。5.2.6节介绍的租借技术可用来解决此问题。与回调相关的第二个问题是服务器需要发送一系列同步的RMI给列表中的回调对象。想了解第二个问题的解决之道，请参见5.4.1节和练习5.18。

我们阐述了白板应用中回调的使用。*WhiteboardCallback*接口可以定义为：

```
public interface WhiteboardCallback implements Remote {
    void callback(int version) throws RemoteException;
};
```

由客户作为远程对象实现该接口，它使服务器能在添加新对象时将版本号发送给客户。但是在服务器这样做之前，客户要通知服务器它的回调对象。为使之成为可能，*ShapeList*接口还需要一些附加方法，例如*register*和*deregister*，其定义如下：

```
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

在客户取得了*ShapeList*接口的远程对象引用（例如图5-15中，第1标记行）并创建了一个回调对象实例之后，它就使用*ShapeList*的*register*方法通知服务器它对接收回调感兴趣。*register*方法返回一个整数（*callbackId*）指向注册项。当客户完成的时候，它会调用*deregister*通知服务器它不再请求回调了。服务器负责维持一张兴趣客户的列表，并在每次版本号增加时通知这些客户全体。

200

5.5.2 Java RMI的设计和实现

初始的Java RMI系统使用了图5-6中所示的所有组件。但是在Java 1.2中，使用反射功能可以创建通用的调度程序并且避免使用骨架。客户代理由一个称为*rmic*的编译器根据已经编译好的服务器类而不是根据远程接口定义来创建。

反射的使用 反射用于传递请求消息中关于调用方法的信息。这是借助于反射包中的*Method*类完成的。*Method*的每个实例代表一个特定方法的特征，包括它的类、参数类型、返回值和异常。这个类最有意义的功能是其实例能通过*invoke*方法被一个合适的类的对象调用。

调用方法需要两个参数，第一个参数指定接收调用的对象，第二个是一个包含参数的*Object*数组。调用的返回结果是*Object*类型。

回到RMI中*Method*类的使用：代理必须把方法及其参数的信息编码到请求消息中。它把方法编码成*Method*类的一个对象，把参数放入一个*Object*数组并编码该数组。调度程序从请求消息中解码*Method*对象和它在*Object*数组中的参数。通常，目标对象的远程引用已经被解码了，对应的本地对象引用已从远程引用模块中获得。然后调度程序调用*Method*对象的*invoke*方法、获得目标对象和参数值数组。当已执行方法后，调度程序将结果或者任何出现的异常编码入应答消息。

这样，调度程序是通用的，也就是说，相同的调度程序能用于所有远程对象类，而且不需要骨架了。

支持RMI的Java类 图5-16给出了支持Java RMI服务器的类的继承结构。程序员只需要知道*UnicastRemoteObject*类，每个简单的伺服类都要扩展它。*UnicastRemoteObject*类扩展了一个称为*RemoteServer*的抽象类，后者提供远程服务器所请求的方法的抽象版本。对*RemoteServer*，第一个要提供的是*UnicastRemoteObject*。另一个称为*Activatable*，现在用于提供主动对象。其他选择可能提供复制对象。*RemoteServer*类是*RemoteObject*类的一个子类，*RemoteObject*的实例变量持有一个远程对象引用并提供如下方法：

- *equals* 这一方法用于比较远程对象引用。
- *toString* 这一方法按*String*类型给出远程对象引用的内容。
- *readObject*、*writeObject* 这些方法用于解序列化/序列化远程对象。

另外，*instanceOf*操作可用于测试远程对象。

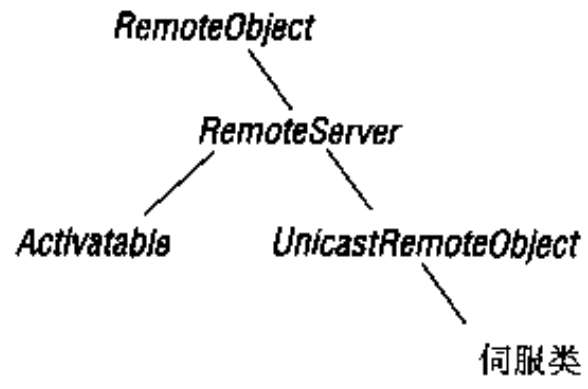


图5-16 支持Java RMI的类

5.6 小结

本章讨论了两种分布式编程的模型——远程方法调用和基于事件的系统。这两种模型都把对象看成是能接收来自远程对象的调用的独立实体。在第一种模型中，某个对象的远程接口中的一个方法被同步地调用（调用者等待应答）。在第二种模型中，每当兴趣对象上发生了一个已发布的事件，就把通知异步地传送给多个订阅者。

分布式对象模型是面向对象编程语言所使用的本地对象模型的一种扩展。封装的对象构成了分布式系统中有用的组件，因为封装性使它们完全负责管理其自身状态，而且可以将方法的本地调用扩展到远程调用。分布式系统中的每个对象都有一个远程对象引用（一个全局惟一的标识符）和一个指定可以远程调用其哪个操作的远程接口。

本地方法调用提供恰好一次语义，而远程方法调用不能保证也这样，因为两个参与对象在不同的计算机上，它们可能分别发生故障，而且用于连接的网络也可能出故障。最好管理

的是最多一次调用语义。由于它们不同的故障和性能特征，以及对远程对象并发访问的可能性，让远程调用与本地调用表现得完全相同未必是一个好的想法。

RMI中间件实现提供了代理、骨架和调度程序等组件，这些组件为客户和服务器的编程人员隐藏了编码、消息传递和定位远程对象的细节。接口编译器可以生成这些组件。Java RMI使用相同的语法将本地调用扩展到远程调用，但是远程接口必须通过扩展一个名为*Remote*的接口来说明，并让每个方法抛出一个*RemoteException*异常。这可以确保让编程员知道什么时候发送远程调用或者实现远程对象，使他们能处理错误，或者为并发访问设计合适的对象。

基于事件的分布式系统可用于分布式异构对象集合的相互通信。与RMI不同，对象不必具有接收消息的远程接口，它们所需要做的全部事情就是实现一个接收通知和订阅事件的接口。生成事件的对象需要发送异步通知。接口的简单性使得可以非常直接地给已经存在的对象增加事件。处理事件的其他工作（如过滤和查找模式）可以由观察者完成。观察者是为此类目的添加到系统的第三方对象。

201
~
202

练习

5.1 *Election*接口提供两个远程方法：

vote 带有两个参数，客户端通过这两个参数提供一个候选者名字（一个字符串）和“投票者编号”（一个用于确保每个用户只投票一次的整数）。投票者编号可以在整数的范围内随机选择，以使它们很难被猜测出来。

result 带有两个参数，服务器通过这两个参数向客户提供候选者的名字和候选者的投票编号。

这两个过程中的哪个参数是输入型的，哪个是输出型的？

5.2 讨论在TCP/IP连接之上实现请求－应答协议时可以获得的调用语义，该调用语义要确保数据按发送顺序到达，既不丢失也不重复。考虑导致连接中断的所有条件。

5.3 以CORBA IDL和Java RMI定义*Election*服务的接口。注意CORBA IDL提供32位的整数类型*long*。比较这两种语言中指定输入和输出参数的方法。

5.4 *Election*服务必须确保每次用户想投票的时候，其选票就被记录下来。

讨论在*Election*服务上或许调用语义的效果。

至少一次调用语义会被*Election*服务接受吗？你认为应该推荐使用至多一次调用语义吗？

5.5 将请求－应答协议在一种带有遗漏故障的通信服务上实现，以提供至少一次RMI调用语义。在第一种情形中，实现者假设一个异构的分布式系统。在第二种情形中，实现者假设通信和远程方法执行的最大时间是*T*。后者以哪种方法能简化实现？

5.6 在*Election*服务中，要确保在多个客户并发访问时，其选举记录能保持一致。概述*Election*服务的实现。

5.7 即使服务器进程崩溃了，*Election*服务必须确保安全地存储所有的选票。参考你在练习5.6中实现概述的回答，解释如何实现这一点。

203

5.8 解释如何采用Java反射构造*Election*接口的客户代理类。给出该类中一个方法的实现细节，它应该用以下标记名称调用*doOperation*方法：

byte [] doOperation (RemoteObjectRef o , Method m , byte [] arguments) ;

提示：代理类的一个实例变量应该持有一个远程对象引用（见练习4.12）。

5.9 解释如何从CORBA接口定义（练习5.3的解答）出发，使用如C++这种不支持反射的语言，生成一个客户代理类。给出该类中一个方法实现的细节，它应该调用图4.12中定义的*doOperation*方法。

5.10 解释如何使用Java反射构造一个通用的调度程序。给出具有下列标记名称的调度程序的Java代码：

public void dispatch (Object target, Method aMethod, byte [] args);

参数包括目标对象、被调用的方法和以字节数组表示的方法所需的参数。

5.11 练习5.8要求客户在调用*doOperation*之前将*Object*参数转化成一个字节数组，练习5.10要求调度程序在调用方法之前将字节数组转化成一个*Object*数组。讨论具有下列标记名称的*doOperation*的新版本的实现：

Object [] doOperation(RemoteObjectRef o, Method m, Object [] arguments) ;

它使用*ObjectOutputStream*和*ObjectInputStream*类在客户和服务器之间基于TCP连接传递请求和应答消息。这些改变将如何影响调度程序的设计？

5.12 客户向服务器发出远程过程调用。客户花5ms时间计算每个请求的参数，服务器花10ms时间处理每个请求。本地操作系统处理每次发送和接收操作的时间是0.5ms，网络传递每个请求或者应答消息的时间是3ms。编码或者解码每个消息花0.5ms时间。

计算下列两种条件下客户创建和返回消息所花费的时间：

（i）如果它是单线程的

（ii）如果它有两个线程，这两个线程能在一个处理器上并发地处理请求

你可以忽略上下文切换时间。如果客户和服务器处理器是线程化的，需要异步RPC吗？

5.13 设计一个能支持分布式无用单元回收和在本地与远程对象引用之间转化的远程对象
204 表。给出一个例子，包括在不同地址上的几个远程对象和代理，以阐述该表的使用。说明当一个调用导致创建新代理时这个表的变化。说明当一个代理不能用时这个表的变化。

5.14 5.2.6节中描述了分布式无用单元回收算法的一个简单版本，该算法在每次创建一个新的代理时就调用远程对象所在地的*addRef*，每次删除一个代理时就调用*removeRef*。概述算法中通信和进程故障可能造成的所有影响。提出克服每种影响的建议，但是不能使用租借。

5.15 讨论在共享白板应用程序中，如何使用Jini分布式事件规范中描述的事件和通知。*RemoteEvent*类由Arnold等人[1999]定义，如下所示：

```
public class RemoteEvent extends java.util.EventObject {
    public RemoteEvent(Object source, long eventID,
        long seqNum, MarshalledObject handback)
    public Object getSource () {...}
    public long getID() {...}
    public long getSequenceNumber() {...}
    public MarshalledObject getRegistrationObject() {...}
}
```

构造函数的第一个参数是远程对象。要求用通知告诉监听者事件已经发生了，但是由监听者负责获取更进一步的细节。

5.16 提出一种通知邮箱服务的设计，该服务用于代表多个订阅者存储通知，允许订阅者指定什么时候它们要求传递通知。说明那些并不总是主动的订阅者如何利用你描述的这种服务。该服务怎样处理订阅者在打开了消息传输后进程崩溃的情况?

5.17 说明如何使用一个转发观察者以增强事件服务中兴趣对象的可靠性和性能。

5.18 提出一种观察者能使用的方法，用于增强练习5.15中所提出的解决方案的可靠性或性能。

205

第6章　操作系统支持

6.1　简介
6.2　操作系统层
6.3　保护
6.4　进程和线程
6.5　通信和调用
6.6　操作系统体系结构
6.7　小结

本章介绍在分布式系统的结点中操作系统设施是如何支持中间件的。操作系统实现了在服务器端的资源封装和保护，同时它还支持用于访问资源的调用机制，这其中包括通信和调度。

系统内核的角色是本章的一个重要主题。本章旨在使读者了解在保护域中划分功能的优点和缺点，特别是划分内核级代码和用户级代码的功能。本章还介绍了内核级设施与用户级设施的平衡，包括效率和健壮性之间的关系。

本章还探讨了多线程进程和通信设施的设计和实现问题，然后介绍已经设计实现的主要的内核体系结构。

6.1　简介

第2章介绍了分布式系统中的主要软件层次。我们知道，资源共享是分布式软件系统的一个重要方面。客户端的应用程序所调用的资源很可能在另一结点上，或至少在另一进程上。应用程序（以客户的形式出现）和服务器（以资源管理者的形式出现）使用中间件层进行交互。中间件提供了分布式系统的各结点中对象或进程间的远程调用。第5章介绍了中间件中远程调用的主要类型，例如Java RMI和CORBA。本章将继续介绍没有实时保证的远程调用（第15章将介绍对实时和面向数据流的多媒体通信的支持）。

在中间件层下面是操作系统（OS）层，它是本章的主题。本章将介绍这两层之间的关系，特别要介绍操作系统是如何满足中间件需求的。其中，中间件需求包括访问物理资源的效率和健壮性，以及实现多种资源管理策略的灵活性。

任何一个操作系统的目标都是提供一个在物理层（处理器、内存、通信设备和存储介质）之上面向问题的抽象。例如，UNIX或Windows NT[Custer 1998]操作系统给程序员提供的是文件和套接字的形式，而不是磁盘块和原始网络访问。操作系统管理某结点的物理资源并通过系统调用接口抽象表示这些资源。

在详细描述操作系统对中间件的支持之前，首先回顾一下在分布式系统发展过程中的两个概念：网络操作系统和分布式操作系统。虽然它们的定义不同，但它们的概念却有些相似。下面将对它们作一些介绍。

UNIX和Windows NT都是网络操作系统的例子。它们都具有网络的功能，因此人们可

以用它们来访问远程资源。它们能网络透明地（network-transparent）访问一些资源，但不是所有资源，例如，通过分布式文件系统如NFS，用户能网络透明地访问文件，也就是说，用户访问的许多文件存储在远端或服务器上，而对于应用程序来说这些文件是位置透明的。

网络操作系统的独特特点是运行于其上的结点能独立地管理自己的进程资源。也就是说，在网络上的每一个结点上都有一个系统映像。通过网络操作系统，用户能使用rlogin或Telnet远程登录到其他的计算机上并在那个计算机上运行进程。然而与操作系统管理本结点计算机的进程的方式不同，网络操作系统并不在结点间调度进程，用户必须参与进程的调度。

相反，可以想象存在一种用户不用关心程序的运行地点或资源位置的操作系统，其中只有一个单一系统映像。这种操作系统必须控制系统中所有的结点，并且它能透明地将进程定位在符合其调度策略的结点上。例如，它能在最小负载的结点上生成一个新的进程用来防止
208 单个结点的过载。

如果一个操作系统像这样对分布式系统中的所有资源只生成单一的系统映像，那么这个系统就是一个分布式操作系统[Tanenbaum and van Renesse 1985]。

中间件和网络操作系统 事实上除了UNIX、MacOS和各种版本的Windows这些网络操作系统外，几乎没有其他普遍应用的分布式操作系统。有两个主要原因导致这种情况，其一是用户过于注重于满足他们当前需要的应用软件，尽管新的操作系统能提高效率，但如果它不能运行当前的应用软件，用户也不会使用它。有人尝试在新的系统核心上模拟UNIX和其他操作系统，但模拟执行的性能却不能令人满意。不管怎样，模拟所有的主流操作系统仍是一项繁重的工作。

第二个原因是用户即使在一个小单位里也更愿意独立地管理自己的机器。其中一个重要的因素是性能[Douglis and Ousterhout 1991]。例如，当琼斯写一个文档时，她需要很好的交互性，而系统由于运行史密斯的程序而变慢，她必定会不高兴。

中间件和网络操作系统的结合提供了在独立性和网络资源透明访问之间的平衡。网络操作系统使用户能运行其字处理程序和其他独立应用程序。中间件使他们能享受到分布式系统提供的服务。

下一节将介绍操作系统层的功能。6.2节介绍资源保护的低层机制，以便我们能了解进程、线程和内核之间的关系。6.4节介绍进程、地址空间和线程抽象，其中主要介绍并发、局部资源管理和保护以及调度。6.5节介绍调用机制的一部分——通信。6.6节介绍操作系统体系结构的不同类型，其中包括整体内核和微内核设计。第18章提供了关于Mach内核的实例研究。读者可以在www.cdk3.net/oss上找到Amoeba、Chorus和Clouds操作系统的实例研究。

6.2 操作系统层

只有当中间件和操作系统的联合系统具有良好的运行性能时，用户才能满意。中间件运行在分布式系统上每个结点的操作系统和硬件设备结合的平台上。运行在结点上的操作系统都有其内核和相关的用户级服务，如一些库，它还为进程、存储和通信提供本地硬件抽象支
209 持。中间件将这些局部资源联合起来以实现在结点的对象或进程之间提供远程调用机制。

图6-1给出了两个结点上的操作系统层是如何支持一个公共的中间件层的，从而为应用和服务提供一个分布式基础设施。

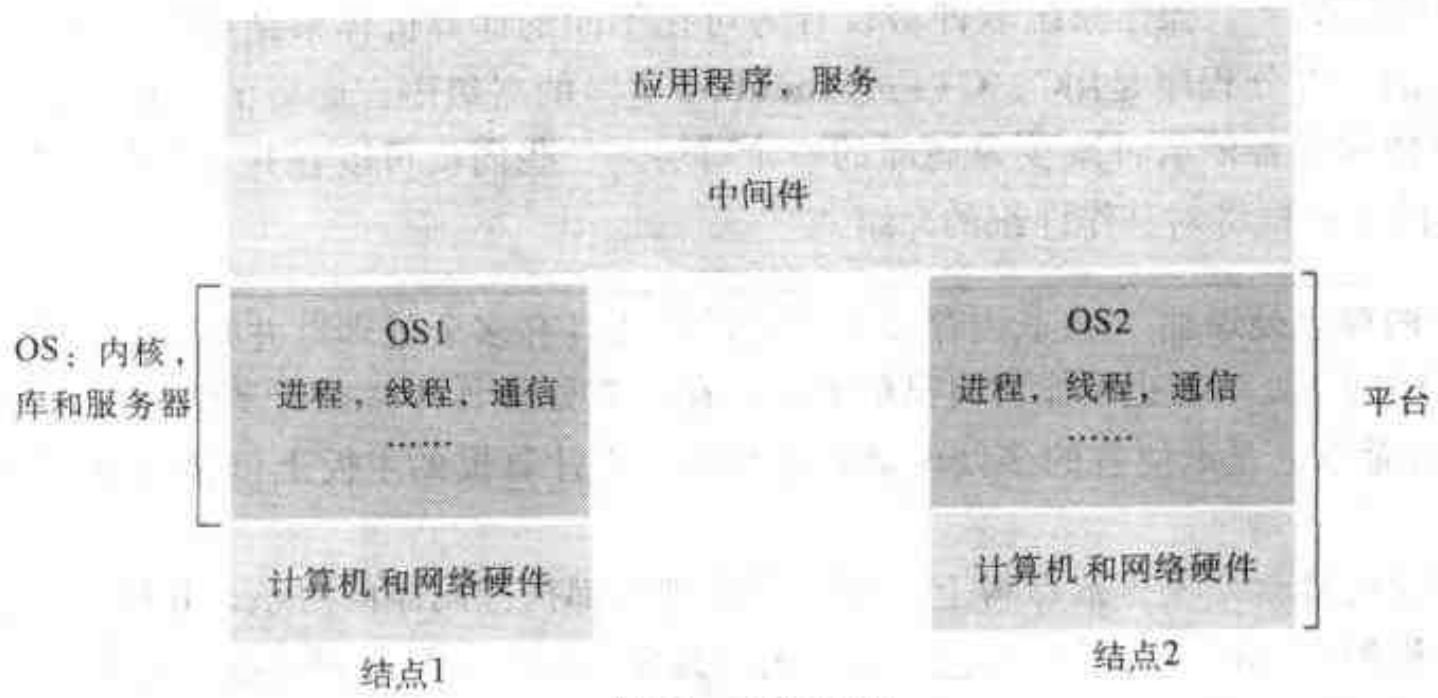


图6-1 系统层次

本章旨在讨论特定的操作系统机制对中间件提供共享分布资源能力的影响。内核和运行于其上的客户进程和服务器进程是我们关心的主要体系结构组件。内核和服务器进程是用于管理资源和为客户提供资源接口的组件，因此，它们至少应该具备以下特点：

- *封装* 它们必须提供一个有用的服务接口以访问物理资源，也就是说，提供的操作集必须满足客户的需要，而像如何管理内存和管理用于实现资源的设备这样的细节必须对客户隐藏。
- *保护* 资源需要被保护以防止非法访问。例如，没有文件读权限的用户不能访问文件，并且设备注册信息也不能被应用程序进程访问。
- *并发处理* 客户应该可以共享和并发地访问资源。资源管理器负责实现并发透明性。

客户访问资源可以通过远程方法调用访问服务器对象，或用系统调用调用内核。我们称访问一个已封装资源的方式为*调用机制*，而不管其是如何实现的。库、内核和服务器的联合系统可以实现如下与调用相关的任务：

- *通信* 资源管理器在网络上或计算机内接收操作参数并返回结果。
- *调度* 当调用一个操作时，在内核或服务器上调度其操作。

210

图6-2给出了我们所关心的操作系统的几个内核部分：进程和线程管理、内存管理以及本机进程间的通信（图上水平的分割线表示依赖关系）。内核提供其中的大部分功能，在某些操作系统中，内核完成上述全部功能。

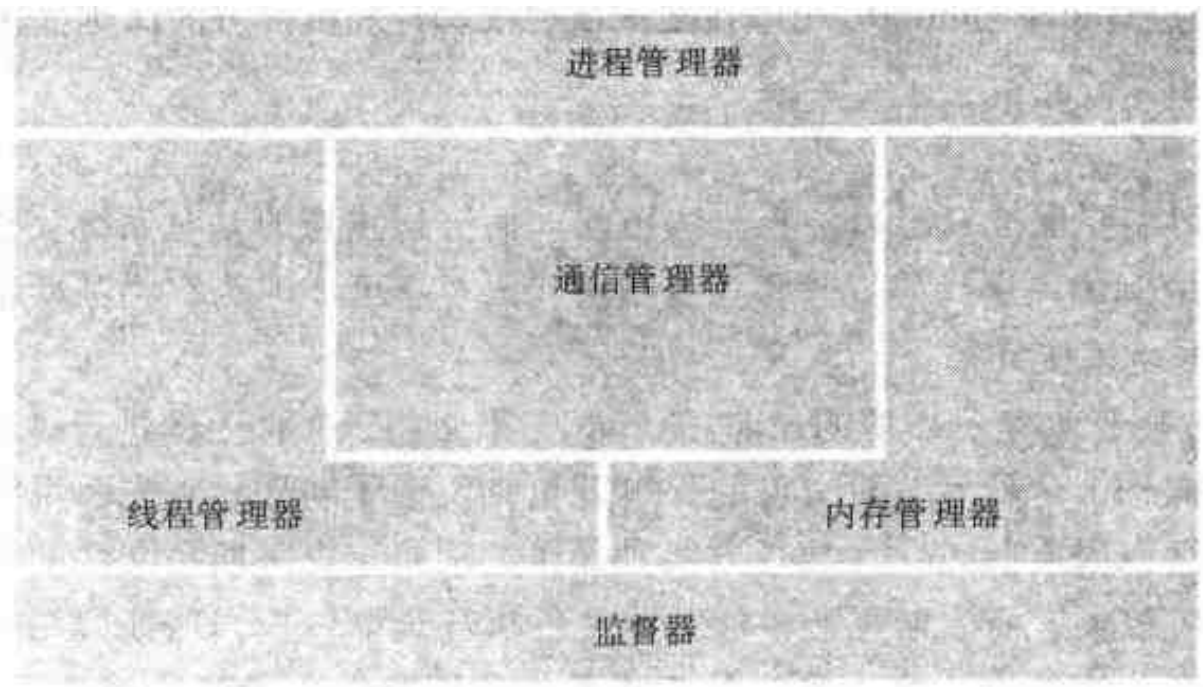


图6-2 OS核心功能

在可能的情况下，操作系统软件被设计为可在不同的计算机体系结构间移植。这就意味着操作系统的大部分程序是用C、C++或Modula-3这样的高级语言编写的，而且它是分层的，这样可以将依赖机器的组件减少为底部的一个薄层。一些内核可以在共享内存的多处理器上执行，下面的文本框将对其作详细的介绍。

共享内存多处理器 共享内存多处理器计算机具有多个处理器并共享一个或多个内存块（RAM）。处理器也可能有自己的私有内存。多处理器计算机有多种构造方式[Stone 1993]。最简单也是最便宜的多处理器系统是在个人计算机的主板上包含若干（2~8）个处理器。

在普通的对称处理体系结构上，每一个处理器都执行同样的内核，并且这些内核在管理硬件资源时都扮演同样的角色。这些内核共享可运行线程队列这样的主要数据结构，但它们也拥有一些私有的工作数据。处理器能同时执行各自的线程，同时也可以访问在共享内存中的私有（由硬件保护的）或公有数据。

许多高性能计算任务可以由多处理器来实现。在分布式系统中，因为多处理器的服务器可以运行一个具有多线程的程序来同时处理多个客户的请求，故它特别适合于实现高性能服务器，例如提供访问共享数据库的服务（见6.4节）。

211

操作系统包括以下核心组件：

- 进程管理器 负责进程的创建和操作。一个进程是一个资源管理单位，其中包括一个地址空间以及一个或多个线程。
- 线程管理器 负责线程创建、同步和调度。线程调度活动与进程相关，这将在6.4节作详细描述。
- 通信管理器 负责同一计算机上不同进程中的线程之间的通信。一些内核可以支持在远程进程中的线程之间的通信。另外一些内核不能访问其他计算机，它们需要附加的服务来进行外部通信。6.5节将讨论通信设计。
- 内存管理器 负责管理物理内存和虚拟内存。6.4和6.5节将介绍利用内存管理技术来实现高效的数据复制和数据共享。
- 监督器 负责处理中断、系统调用陷阱和其他异常，同时控制内存管理单元和硬件缓存以及处理器和浮点寄存器操作。在Windows NT中，它便是硬件抽象层。读者可以在Bacon[1998]和Tanenbaum[1992]中找到内核中依赖计算机部分的详细描述。

6.3 保护

上文曾经提到需要保护资源以防止非法访问。而值得注意的是对系统完整性的威胁并不仅仅来源于恶意编制的程序代码。非恶意编制的代码也有可能因为存在某些错误或具有未预料的行为而导致系统工作异常。

为了使读者了解什么是对资源的“非法访问”，下面将以文件为例进行讨论。假设对打开的文件只有两种操作，读和写，对文件的保护就包括两个子问题。首先，系统需要保证客户必须有相应的执行权限才能对文件执行这两种操作。例如，史密斯是文件的拥有者，她有对文件的读权限和写权限，而琼斯可能只能对此文件执行读操作。当琼斯试图对文件进行写操作时，便产生一个非法访问。完全解决分布式系统的资源保护子问题需要运用密码学技术，

本书将在第7章中对此进行介绍。

另外一种类型的非法访问是客户错误地执行了资源不能提供的操作。例如，在上面的例子中，如果史密斯或琼斯试图执行一个既不是读也不是写的操作，就属于这种类型的非法访问。假设史密斯试图直接访问文件指针变量，她能够构造一个*setFilePointerRandomly*的操作，这一操作将文件指针设置为一个随机值。当然，这是一个没有实际意义的操作，并且它可能扰乱对文件的正常使用，因此，系统不应该设计这样一个文件操作。 212

我们应该保护资源以防止如*setFilePointerRandomly*这样的非法调用。一种方法是使用类型安全的编程语言，例如Java或Modula-3。在类型安全的语言中，模块只能访问有引用的目标模块，而不能像C或C++那样通过指针来调用，并且它只能用其引用来执行目标模块提供的可用调用（方法调用或过程调用）。换句话说，它不能任意改变目标模块的变量。相反，C++程序员可以把指针转换成任意类型，从而执行非类型安全的调用。

我们也可以使用硬件来保护模块以防止其他模块的非法调用，这样可以不考虑调用模块是用什么样的语言写成的。我们需要一个具有这种机制的操作系统内核使这样的操作机制在通用计算机上可行。

内核和保护 内核是一个计算机程序，它的特点是时刻保持运行并且对主机的物理资源有完全的访问权限，特别是它可以控制内存管理单元并设置处理器的寄存器。这样，其他程序代码除非通过某种可接受的方式，否则不能访问机器的物理资源。

大多数处理器都有硬件模式的寄存器，用来决定能否执行特权指令。例如有些寄存器决定内存管理单元当前采用哪一个保护表。当内核进程执行时，处理器处于管理（特权）模式，而内核安排其他进程在用户（非特权）模式下运行。

内核也建立地址空间来保护自己和其他进程以防止其他异常进程的访问，同时也给正常进程提供它们所需要的虚拟内存。一个地址空间是若干虚拟内存区域的集合，其中每一个区域都被赋予特定的内存访问权限，例如只读或读写权限。进程不能访问其地址空间以外的内存。术语*用户进程*和*用户级进程*是用来描述在用户模式下执行并拥有用户级地址空间的进程（相对于内核对地址空间的访问，这些进程有受限的内存访问权力）。

当一个进程执行应用程序代码时，它在用户级地址空间中执行；而当这一进程执行内核代码时，它在内核地址空间内执行。通过中断或*系统调用陷阱*（这种资源调用机制由内核管理），这一进程可以安全地在用户级地址空间和内核地址空间中切换。一次系统调用陷阱由一个机器级的TRAP指令实现，它将处理器切换为管理模式并将地址空间切换为内核地址空间。当执行TRAP指令时，计算机硬件强制处理器执行内核提供的处理函数以保证没有其他进程获得对硬件的控制。

因为保护机制，系统在执行程序时会导致额外的开销。在地址空间之间切换会占用许多处理器的处理周期，并且系统调用陷阱也比简单的过程或方法调用更耗费处理器资源。下面我们将了解这些不利因素是如何影响调用开销的。 213

6.4 进程和线程

传统的操作系统概念是一个进程执行一个单独的活动，但在20世纪80年代，人们发现它不能胜任分布式系统，同时也不能胜任那些需要内部并发的更复杂的单机应用。主要问题是传统的进程使相关活动的共享变得困难且开销大。

对此问题的解决方法是增强进程的概念使它能与多个活动联系起来。现在，一个进程包括一个执行环境和一个或多个线程。一个线程是一个活动的操作系统抽象（这一术语来源于“执行线”）。一个执行环境是一个资源管理单元，也就是进程的线程所能访问的由内核管理的本地资源集。一个执行环境主要包括：

- 一个地址空间。
- 线程同步和通信资源，如信号量和通信接口（例如套接字）。
- 高级资源，如打开的文件和窗口。

创建和管理执行环境在通常情况下都会耗费系统资源，但是多个线程可以共享一个执行环境，也就是说，它们可以共享执行环境中的所有可用资源。换句话说，一个执行环境表示允许线程在其中执行的保护域。

线程可以按需要自动创建和终结。多线程执行的主要目的是最大化操作间的并发程度，这样可以将计算和输入输出同时执行，同时也可以支持在多处理器上的并发执行。当多个并发的客户请求可能降低服务器的速度使其成为瓶颈时，这种方式的作用尤其明显。例如，当为客户服务的某个线程正在等待磁盘访问完成时，另一个线程可以处理一个客户的请求。

一个执行环境提供一种资源保护以防止外部线程访问，这样执行环境内的数据和其他资源在默认情况下是不能被其他执行环境中的线程访问的。但是，某些内核允许可控制地共享资源，例如它们可以共享同一计算机上不同执行环境间的物理内存。

因为许多早期的操作系统在一个进程上只允许运行一个线程，所以有时我们强调使用术语多线程进程以示区别。容易混淆的是在一些编程模型和操作系统中的术语“进程”实际是指我们所说的线程。读者也可能在文献中遇到术语重量级进程，其包含了它的执行环境，而
214 轻量级进程则不包含执行环境。下面将用一个比喻说明线程及其执行环境。

线程和进程的比喻 下面是一个在*comp.os.mach* USENET组上由Chris Lloyd描述的关于线程和执行环境有趣的比喻。一个执行环境是一个内部装有空气和食物的封了口的瓶子。开始，瓶子中只有一个苍蝇——线程。这个苍蝇可以生出其他苍蝇，也可以杀死它们。它的后代也能这样做。苍蝇会消耗瓶子内的资源（空气和食物）。假设苍蝇们排队消耗资源。如果它们不遵守这一原则，它们会在瓶子中撞在一起——也就是说，当它们以一种没有约束的方式去试图消耗同一资源时会产生冲突从而产生无法预料的结果。苍蝇能与瓶子中的其他苍蝇通信，但是它们都不能飞出这个瓶子，外面的苍蝇也不能飞进来。按这种观点，一个标准的UNIX进程是一个瓶子，并只包括一个不能生出其他苍蝇的苍蝇。

6.4.1 地址空间

前面章节已经介绍过，一个地址空间是一个进程的虚拟内存的管理单元。它可以很大（通常最大可达到2^{32}字节，有时可达到2^{64}字节），可以拥有一个或多个区域，这些区域被不可访问的虚拟内存区分隔开。一个区域（如图6-3所示）是一个可以被本进程的线程访问的连续的虚拟内存区，区域间不能重叠。注意，我们区分区域及其内容。每一个区域包括如下性质：

- 范围（最低的虚拟内存地址和区域大小）

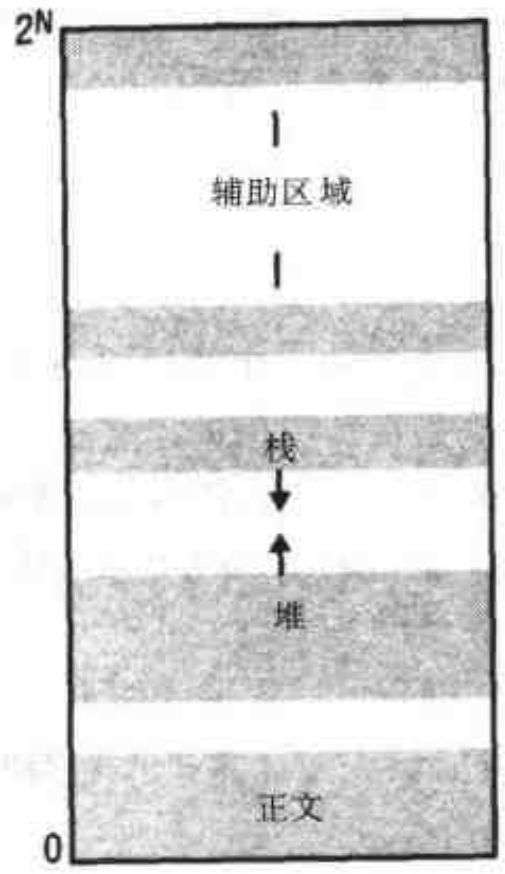


图6-3 地址空间 215

- 对本进程的线程的读/写/执行权限
- 是否能够向上或向下扩展

注意，此模型是基于页的而不是基于段的。区域与段不同，当区域扩展大小时，它们最终可能会重叠。区域之间留有空隙，用于区域增长。这种由若干不相交区域组成的地址空间表示是对UNIX地址空间的一种概括，UNIX地址空间包含了3个区域：固定的不可变的正文区包含程序代码；一个堆，其中一部分由存储在程序的二进制文件中的值初始化，并且这个区域可以向更高的虚拟地址空间扩展；一个栈，它能向更低的虚拟地址空间扩展。

有几个因素影响了应提供的区域的数目。其中之一是系统需要为每一个线程提供一个独立的栈。为每一个线程分配一个栈使系统能检测栈的溢出并控制栈的增长。未被分配的虚拟内存在这些栈之外，访问这些内存会引起异常（页失配）。另一种方法是将栈放在堆的上方，但是这样会使系统难于检测线程超过其栈的界限这一错误。

采用以上机制的另一个原因是它能将所有的文件——而不仅仅是二进制文件正文和数据区——映射到地址空间。一个映射文件是一个在内存中可访问的字节数组。虚拟内存系统确保对内存的访问映射到底层的文件存储系统上。18.6节描述了Mach内核是如何扩展虚拟内存抽象，以及它是如何使区域对应于任意的“内存对象”，而不仅仅对应于文件。

在进程之间或在进程与内核之间共享内存的需要是导致产生地址空间中额外区域的另一个因素。一片共享内存区（或简称为共享区）和其他地址空间的一片或多片内存区可能由同一片物理内存区支持。因此，进程可以通过访问这些内存区来共享内存，而它们拥有的没有被共享的区域是受到保护的。共享区的应用包括如下几个方面：

- *库* 如果将库的代码分别装载在每个使用它的进程的内存区中，它可能会占用相当大的内存。相反，可以将一个库代码的拷贝映射到需要它的多个进程的地址空间的区域中，以达到共享的目的。
- *内核* 通常将内核代码和数据映射到同一位置的每一个地址空间中，当进程进行系统调用或进行异常处理时，系统不需要进行地址空间映射的切换。

- *数据共享和通信* 两个进程或进程和内核可能需要共享数据以达到协同工作的目的。相对于消息之间的传递而言，将共享数据映射到同在两个地址空间中的区域可以提高效率。6.5节将介绍将区域共享用于通信。

216

6.4.2 新进程的创建

创建新的进程传统上是操作系统提供的一个不可再分的操作。例如，UNIX的*fork*系统调用创建一个新的进程，它的执行环境是从其调用进程复制得来的（除了*fork*的返回值）。UNIX的*exec*系统调用将调用进程转换为执行一个指定名字的程序的代码。

在分布式系统中，进程创建机制的设计必须考虑到如何利用多台计算机；因此，支持进程的基础设施被划分为几个分离的系统服务。

在分布式系统中，新进程的创建过程可以划分为两个独立的方面：

- 选择目标主机。例如，系统可能从充当计算机服务器的集群计算机中选择一个结点作为进程的主机（见下面的阴影部分）。
- 创建可执行环境（和一个初始线程）。

> **集群** 集群是由像100Mbps以太网这样的高速通信网连接起来的计算机集合（一般为几十个，有时达到上百个）。每一个计算机可能是标准的PC机或工作站，也有可能由插有多个处理器的主板组成，这些计算机可以是单处理器也可以是多处理器。集群的一个应用是为瘦客户提供计算能力，另一个应用功能是提供高可用性和可伸缩性服务（例如在因特网上给用户提供搜索引擎），通过在集群的处理器之间复制或分区服务器状态可以实现以上应用功能[Fox *et al.* 1997]。集群也可以用于运行并行程序[Anderson *et al.* 1995, now.cs.berkeley.edu, TFCC]。

进程主机的选择 选择新进程驻留的结点（进程分配决定）是一个策略问题。一般情况下，进程分配策略包括从总是在原工作站上运行的新进程到在多个计算机上共享处理负载等策略。Eager等人[1986]区分了下列负载共享的策略分类。

*转移策略*决定是将新进程运行在本机还是运行在其他机器上，而这取决于本地结点是轻负载还是重负载。

*定位策略*决定选择哪一个结点作为新进程的主机，这取决于结点的相对负载情况、机器的体系结构和它是否拥有某些特殊资源。V系统[Cheriton 1984]和Sprite系统[Douglis and Ousterhout 1991]都为用户提供命令，用于在操作系统选择的当前空闲的工作站上执行一个程序（在给定的时间限制内可能有一些这样的机器）。在Amoeba系统[Tanenbaum *et al.* 1990]中，运行服务器为每个进程从一个共享处理器池中选择一个处理器作主机。在上面的例子中，如何选择主机对程序员和用户来说是透明的。然而，那些并行或容错程序需要有指定进程位置的方法。

217

进程定位策略有两类，*静态的*或*适应性的*。前者不考虑当前的系统状态，它是根据系统长期的特点设计的。它们是基于数学分析的，其目的是优化处理器吞吐量这样的参数。它们可能是决定性的（“结点A总是将进程转移给结点B”）或者可能是可能性的（“结点A应该将进程随机转移给结点B到结点E之中的任何一个”）。适应性策略根据每个结点的负载这样的当前运行因素采取启发式方法作出进程分配决定。

负载共享系统可能是集中化的、层次化的或分散化的。集中化的系统有一个负载管理器组件，而层次化的系统有多个这样的组件并组织成树形结构。负载管理器负责收集结点的信息并根据这些信息将新进程分配到结点上。在层次化系统中，负载管理器尽可能将进程分配到自己的底层结点上，但是管理器在某些负载情况下可以通过与其他管理器的公共祖先结点将进程转移到其他结点上。在分散化的负载共享系统中，结点之间为完成进程分配决策可直接传递信息。例如，Spawn系统[Waldspurger *et al.* 1992]将结点看做对计算机资源的“购买者”和“销售者”并且用（分散化的）“市场经济”规则来管理它们。

在发送方启动的负载共享算法中，需要创建一个新进程的结点负责启动转移决策。如果它自己的负载超过了某一界限，它会启动一个转移过程。相反，在接收方启动的算法中，结点在自己的负载低于某一界限时建议其他结点将工作转移给自己。

可迁移的负载共享系统不仅可以在创建新进程时转移负载，任意时刻它已都可以转移负载。将一个正在执行的进程从一个结点转移到另一个结点的机制称为进程迁移。Milojicic等[1999] 详细描述了进程迁移和其他类型的移动性。尽管现在有一些进程迁移机制，但它们没有被广泛应用，其中一个主要的原因是它们的代价太高。为了将进程转移到其他结点上，系统需要从内核中提取进程运行的当前状态，而这种操作是相当困难的。

Eager等人[1988]考察了3种负载共享的方法从而总结出：在任何负载共享机制中，简单是一个很重要的性质。这是因为相对高的开销（例如状态收集开销）带来的弊端可能超过较为复杂机制所带来的好处。

创建新的执行环境 一旦选择了主机，新进程需要一个包含地址空间和初始化信息（可能还包含其他资源，如默认打开的文件）的执行环境。

有两种定义和初始化新进程地址空间的方法。当地址空间是一个静态定义的格式时采用第一种方法，例如，地址空间可能只包含一个程序正文区、一个堆和一个栈。在这种情况下，地址空间区域根据指定了地址空间区域内容的列表来创建，然后地址空间区域由一个可执行文件进行初始化或用零填满。

另外一种方法是根据一个已存在的执行环境来定义地址空间。例如，在UNIX的*fork*操作的语义中，新创建的子进程共享其父进程的正文区，同时，它的堆和栈是父进程的复制（长 218
度和初始化值）。此机制可推广成父进程的每一个区域都可能被其子进程继承（或忽略）。区域继承可以通过共享父进程的区域实现，也可以通过复制父进程的区域实现。当父进程和子进程共享一片区域时，属于父进程区域的页面帧（对应于虚拟内存页面的物理内存单元）同时被映射到相应子进程的区域中。

Mach[Accetta *et al.* 1986]和Chorus[Rozier *et al.*1988，1990]在将父进程的区域复制到子进程区域的过程中采用了写时复制的优化机制。默认情况下，被复制的区域没有在物理上拷贝到目标区域中，组成被继承区域的页面帧被两个地址空间共享。当其中的一个进程试图修改页面时，系统才进行物理上的页面复制。

写时复制是一种通用的技术，例如，它也用于复制大量信息。下面将介绍其操作机制。在图6-4 中，进程A和进程B分别拥有内存区RA和RB，这两个区域是用写时复制机制共享的。假设进程A将区域RA设置成允许其子进程B复制继承，这样在进程B的地址空间中就创建了区域RB。

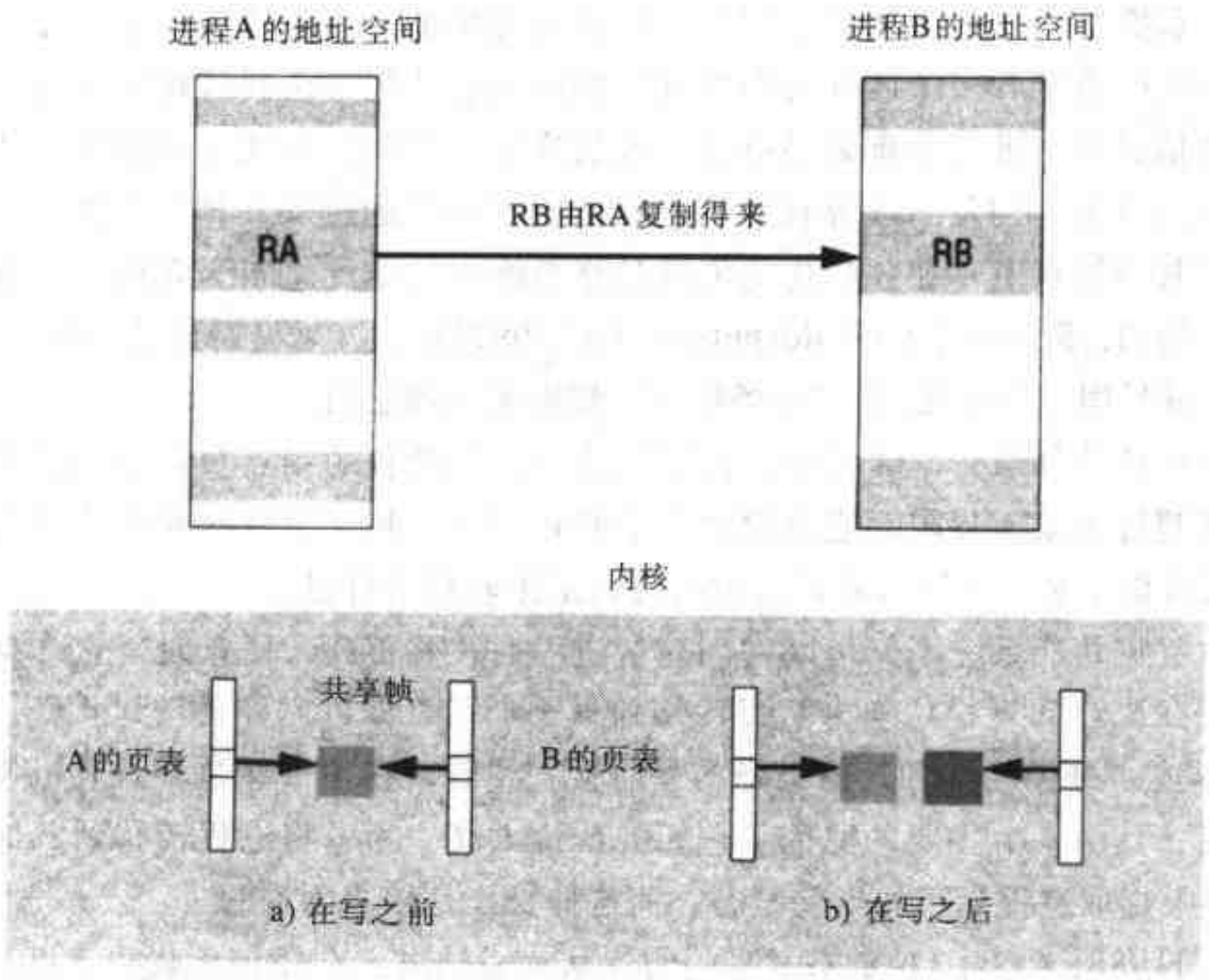


图6-4 写时复制

为简单起见，假设区域RA中的页面都在内存中。开始，区域中的所有页面帧被两个进程的页表共享。即使这些页面所在的区域是逻辑上可写的，但初始时这些页面在硬件级是被写保护的。如果进程中的某一个线程试图修改数据，系统会产生一个页失配。假设进程B试图写内存，页失配处理程序会为进程B分配一个新页面帧，并将原页面帧中的数据拷贝到新页面帧中。同时，在进程B的页表中的那个旧的页面帧号被新的页面帧号所代替，而在进程A的页表中原有的页面帧号不变。此后，进程A和进程B的对应页都在硬件级被设为可写。在完成以上操作后，进程B的修改指令就可以运行了。

219

6.4.3 线程

进程的另一个重要的内容是线程。本小节主要介绍客户进程和服务器进程拥有多线程带来的好处，其后会用Java线程作例子讨论使用线程编程，最后介绍实现线程的各种方式。

如图6-5所示，服务器拥有一个包含一个或多个线程的线程池，其中每一个线程重复地从队列中取出已收到的请求并对其进行处理。本小节暂不讨论如何接收请求和排队请求以等待线程处理。并且，为了简单起见，假设每一线程都采用同样的程序来处理请求。假设每一个请求平均占用2ms的处理时间和8ms的输入输出延迟，其中输入输出延迟是由于服务器从磁盘上读取信息造成的（此系统没有缓存）。同时还假设服务器在一个单处理器的计算机上执行。

下面将讨论以处理器每秒处理客户请求数为度量，考虑不同数目的线程运行时的服务器的最大吞吐量。如果只有一个线程执行所有的操作，因为处理一个请求的周转时间平均需要2 + 8=10ms，那么服务器在一秒内能处理100个客户请求。当服务器在处理请求时，任何新到达的请求将在服务器端口上排队。

现在考虑服务器的线程池中包含两个线程的情况。假设每一个线程都是独立调度的——也就是说，当一个线程由于输入输出被阻塞的时候仍可调度另一线程。这样当第一个线程被阻

塞时，第二个线程能处理第二个请求，反之亦然，这样可以提高服务器的吞吐量。但是，在这个例子中，线程会被单一的磁盘存取阻塞。如果对于所有的磁盘存取请求都要串行执行，且每一请求占用8ms，那么，服务器最大的吞吐量为每秒处理1000/8 = 125个请求。

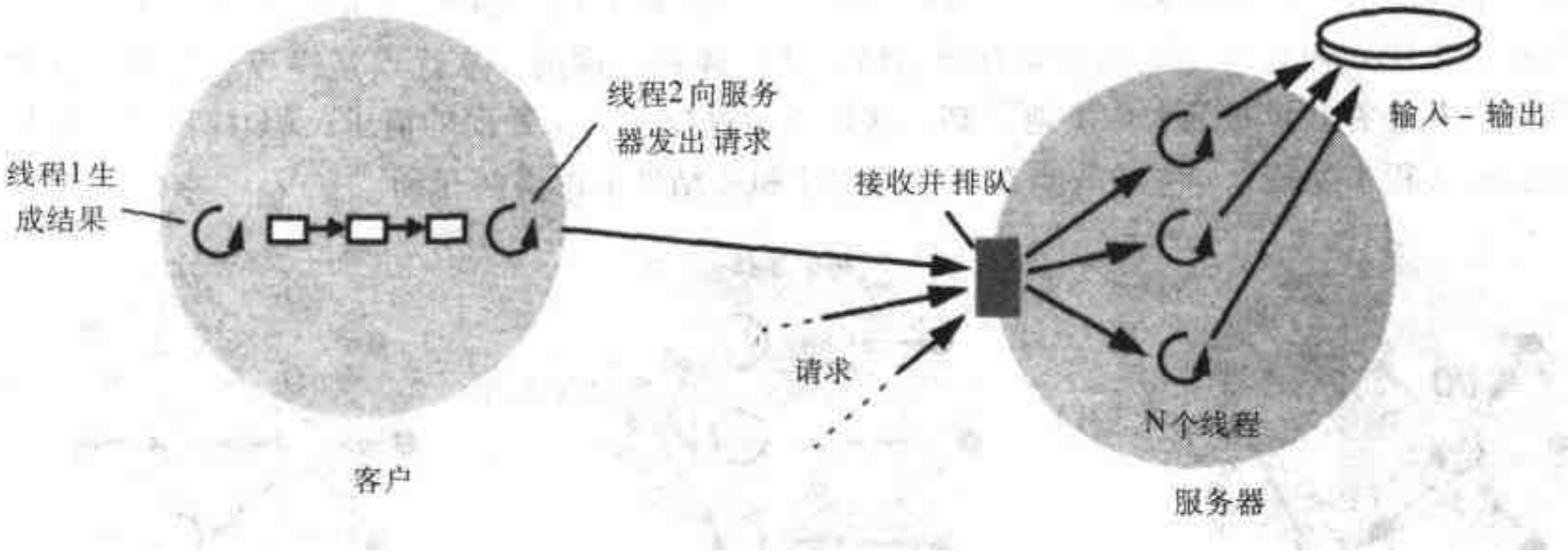


图6-5 拥有线程的客户和服务器

现在假设系统中引入磁盘缓存，服务器将它读的数据放在缓冲区内；当服务器线程检索数据时，它会首先在缓存中查找数据，如果数据在缓存中，它就从缓存中读取数据，而不去访问磁盘。如果在缓存中找到数据的平均命中率为75%，这就意味着平均输入输出时间减少为（0.75 ×0 + 0.25 × 8）=2ms，理论上最大的吞吐量增加到每秒处理500个请求。但如果由于缓存的原因（在每次操作中寻找缓存中的数据需要耗费额外的时间），平均处理时间增加到每次请求耗费2.5ms，那么系统将无法达到上面理想情况下的吞吐量。这时，由于处理器的限制，服务器只能每秒处理1000/2.5 =400个请求。

如果采用共享内存的多处理器来缓解处理器的瓶颈，系统的吞吐量会提高。多线程的进程可以自然地映射到共享内存的多处理器上。可以在共享内存中实现其共享的执行环境，并且可以在多个处理器上运行多线程。假设服务器在有两个处理器的多处理机上执行，线程可以在不同的处理器上独立地运行，那么最多有两个线程可以并行地处理请求。读者可以自己计算并得出结果：两个线程每秒可以处理444个请求，而使用3个或更多的线程，由于受输入输出时间的限制，每秒可以处理500个请求。

多线程服务器的体系结构 上文已经介绍了多线程体系结构如何使服务器增加其吞吐量，其中，吞吐量是用每秒处理的请求数度量的。为了描述在服务器内将请求分配给线程的不同方式，我们引用了Schmidt[1998]的总结结果，他描述了多线程体系结构在CORBA的对象请求代理（ORB）上的多种实现。ORB处理一组套接字上到达的请求。不管系统是否使用CORBA，其线程体系结构与多种类型的服务器相关。

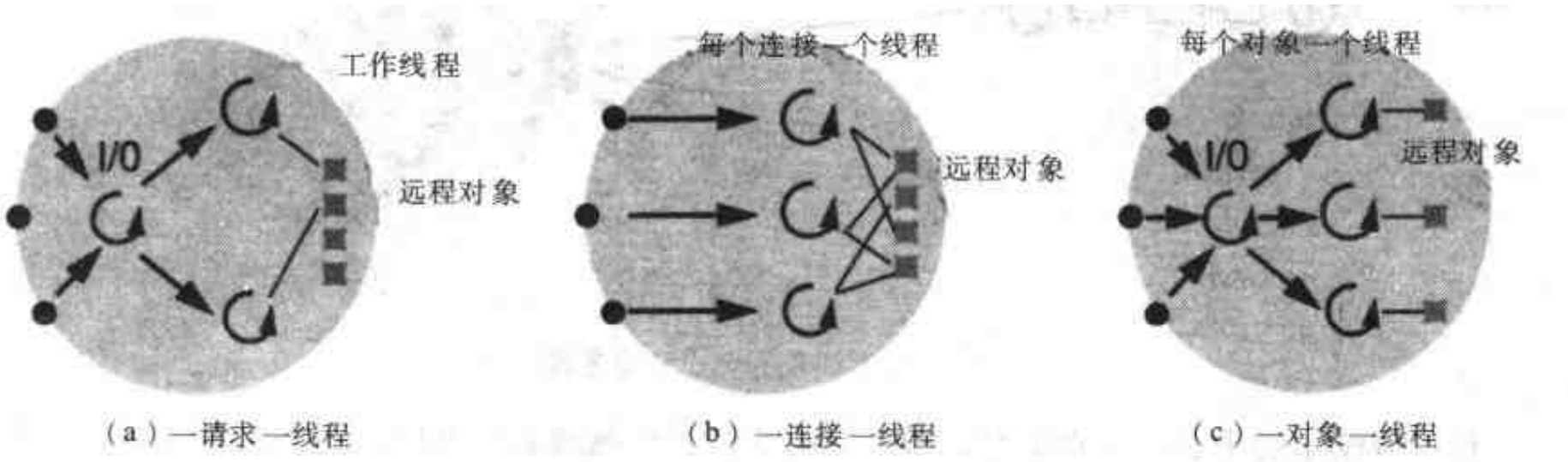
图6-5给出了一种可能的线程体系结构，即工作池体系结构。它的最简单的形式是由服务器创建一个固定的“工作”线程池以便处理请求。在图6-5中出“接收并排队”标记的模块通常由一个“I/O”线程实现，这一线程从一组套接字或端口中接收请求，并将它们放在共享的请求队列中以便工作线程进行检索。

有些时候，系统需要按不同的优先级处理请求。例如，一个公司的Web服务器必须根据产生请求的用户类别优先处理某些请求[Bhatti and Friedrich 1999]。我们可以在工作池体系结构中引入多个请求队列处理多个请求优先级，这样，工作线程能按优先级降序扫描这些队列。

这种体系结构的一个缺点是它缺乏灵活性：在上文提出的例子中，工作线程的数量可能太少，因此不能恰好处理到达的用户请求。它的另一个缺点是当其操纵共享请求队列时，系统可能频繁地在I/O和工作线程中切换。

220
~
221

在一请求一线程体系结构（如图6-6a所示）中，I/O线程为每一个请求派生一个新的工作线程，工作线程处理完此请求会自动终结。这种体系结构的一个优点是线程不会竞争共享的队列，并且吞吐量被最大限度地提高，这是因为对应多个未处理的请求，I/O线程会产生同样数目的线程来处理它。它的缺点在于线程创建和终结带来的高额开销。



（a）一请求一线程　（b）一连接一线程　（c）一对象一线程

图6-6 几种服务器线程体系结构（参见图6-5）

一连接一线程体系结构（如图6-6b所示）为每个连接分配一个线程。服务器在每个客户建立连接时创建--个新的工作线程，并当此连接关闭时终结该工作线程。在这一过程中，客户在此连接上可发送多个请求，并可以访问一个或多个远程对象。一对象一线程体系结构（如图6-6c所示）将每个远程对象分别与一个线程相连。有一个I/O线程接收请求并将其放入工作线程队列排队，此处是每一个对象有一个请求队列。

相对于一请求一线程的体系结构而言，在后两种体系结构中，服务器降低了线程管理的开销。而其缺点在于当一个工作线程有多个请求等待处理时，客户的请求会被延迟，而同时可能有其他线程处于空闲状态。

Schmidt[1998]描述了这些体系结构的变种以及它们的--些混合类型。他详细讨论了这些体系结构的优点和缺点。6.5节将描述在单机调用环境中的不同的线程模型，在该模型中客户线程将访问服务器的地址空间。

客户线程 线程也可以应用于客户，正像它应用于服务器一样。图6-5也描述了一个包含两个线程的客户进程。第一个线程生成的结果将对服务器进行远程方法调用，但线程不需要获得远程方法调用的返回结果。即使调用者不需要等待，远程方法调用也会阻塞调用者。客户进程包含的第二个线程可以执行远程方法调用并且被阻塞，而此时第一个线程可以继续进行计算工作。第一个线程将结果放在缓冲区内，而第二个线程从缓冲区内取出结果。仅当缓冲区满了以后，第一个线程才被阻塞。

Web浏览器是采用多线程的客户结构的明证。用户在获取网页时经常会经历延迟，因此浏览器必须能处理获得多个网页的并发请求。

线程对多进程 从上面的例子中可以了解到，线程的使用可以允许计算与输入输出并行，在多处理器情况下，还可以允许多个计算任务并行。读者可能注意到使用多个单线程的进程
222 也可能达到同样的并行执行。那么，为什么要使用多线程的执行模型呢？其原因包括两方面：

创建和管理线程的开销比进程少；同时，因为线程共享一个执行环境，线程之间比进程之间更容易共享资源。

图6-7给出了执行环境和线程要维护的几种主要的状态组件。一个执行环境拥有一个地址空间、套接字这样的通信接口、打开的文件这样的高级资源以及信号量这样的线程同步对象。图6-7也列出了与之相联系的线程。线程拥有一个调度优先级、一个执行状态（例如阻塞或运行）、线程被阻塞时保存的处理器的寄存器值以及与软件中断处理有关的状态。一个软件中断是一个导致线程中断（类似于硬件中断）的事件。如果线程被加载一个处理程序，它就获得了系统控制权限。UNIX的信号便是软件中断的一个例子。

执行环境	线　程
地址空间表	被保存的处理器寄存器
通信接口，打开的文件	优先级和执行状态（例如阻塞状态）
信号量，其他同步对象	软件中断处理信息
线程标识符列表	执行环境标识符
（驻留在内存的地址空间页面，硬件缓存入口）	

图6-7　与执行环境和线程相关联的状态

上图显示了执行环境和其内的线程都与驻留在内存中的地址空间的页面以及在硬件缓存中的数据和指令相联系。

下面，我们将对进程和线程的比较作一个总结：

- 在一个已有进程内创建一个线程比创建一个进程开销小。
- 更重要的是，在一个进程的不同线程之间切换比在不同进程的线程之间切换开销小。
- 一个进程内的线程共享数据和其他资源比不同进程共享资源效率高。
- 线程不能防止同一进程内其他线程的非法访问。

考虑在一个已有的执行环境中创建新线程的开销。创建新线程的主要任务是为进程栈分配一个区域并为处理器寄存器和线程执行状态（初始值可以是挂起或运行）以及优先级提供一个初始值。因为执行环境已经存在，所以系统只需要在线程的描述符记录（其中包含管理线程执行的必要数据）中放置此执行环境的标识符即可。 223

创建进程的开销一般要大大高于创建一个新的线程。创建进程时，必须首先建立一个新的执行环境，其中包括一个地址空间表。Anderson等人[1991]引用了下列数字：花费11ms用于创建一个新的UNIX进程，而在同一CVAX处理器体系结构上运行Topaz内核创建一个线程只用1ms；此处，度量的时间包括用一个新的实体调用一个空的过程然后退出。这些数字只是给出一个大致的估计。

当这个新的实体执行一些有用的工作，而不是只调用一个空的过程时，它会产生一个长期的开销，但创建新进程产生的这一开销仍比创建新线程多。在操作系统的内核支持虚拟内存的情况下，新创建的进程第一次引用数据和结构时会发生一个页失配，最初，硬件缓存不包含新进程的数值，它必须在进程执行中获得缓存输入。在新线程创建的过程中，这样的长期开销也可能存在，但它相对要小一些。当新线程所要访问的程序代码和数据已经被进程中的其他线程访问时，它便自动地利用了硬件或主存缓存带来的好处。

线程的第二个性能上的优点在于线程间的切换。所谓线程切换是指在给定处理器上运行一个新的线程以代替原来运行的线程。切换开销是非常重要的，因为这在线程的生存期中会

经常发生。共享同一执行环境的线程之间切换的开销要比不同进程的线程之间切换的开销低很多。线程切换的开销主要来源于调度（选择下一个将要运行的线程）和上下文切换。

处理器的上下文包括如程序计数器这样的处理器寄存器的值，它还包括当前硬件的保护域：地址空间和处理器保护模式（管理模式或用户模式）。上下文切换是在线程切换时或一个线程进行系统调用或处理其他类型异常时发生的上下文转换。它包括以下两方面：

- 保存处理器寄存器中原有的状态，并装载新的状态。
- 在某些情况下，转换到新的保护域，即域转换。

共享同一执行环境的线程只有完全在用户层切换才不会引起域转换，并且其系统开销也比较低。切换到内核或通过内核切换到属于同一执行环境的其他线程会引起域转换，其开销会相对高一些，然而，如果内核被映射到进程的地址空间，其系统开销还是比较低的。如果在不同执行环境的线程间切换，其开销就比较大。下面的文本框部分描述了为实现域转换而采用硬件缓存的代价。当这类域转换发生时，更多地会用到由访问硬件缓存入口地址和主存页面地址引起的长期开销。Anderson等人[1991]引用的数据说明，在同一执行环境下，在
224 UNIX的进程上需花费1.8ms进行线程切换，而在Topaz内核上只需花费0.4ms。如果线程在用户级上切换，则花费的时间更少（0.04ms）。这些数据只是一个大致的估计，没有考虑长期的缓存开销。

> **别名问题** 内存管理单元通常包括一个硬件缓存，用于加速在虚拟地址和物理地址间的转换，被称为翻译后援缓冲区（TLB）。TLB和存放虚拟地址中的数据和指令的缓存都会遇到别名问题。同一虚拟地址可以在两个不同的地址空间中都有效，但实际上它们在两个地址空间中引用的是不同的物理数据。除非它们的入口被标记了上下文的标记，否则TLB和虚拟地址的缓存都不知道这一点，从而会包含不正确的数据。因此，TLB和缓存内容必须有选择地进入不同的地址空间。物理地址的缓存不会遇到别名问题，但是通常是采用虚拟地址来实现缓存查找，这是因为这种查找操作可以和地址转换并行进行。

在上面包含两个线程的客户进程的例子中，第一个线程产生数据，并将其传递给第二个线程，由第二个线程进行远程方法调用或远程过程调用。因为这两个线程共享地址空间，因此不需要在它们之间用消息传递的方式传递数据。两个线程都能通过一个公共变量访问数据。此处存在着使用多线程操作的优点和危险。优点在于它们可以方便高效地访问共享数据，在服务器方这也是优点。然而，如果共享同一地址空间的线程不是用类型安全的语言编写的，那么它们受不到保护。一个异常的线程可以随意地改变其他线程使用的数据，这会造成错误。如果必须保护执行的线程，那么必须用安全类型语言编写线程，或者改用多进程取代多线程。

线程编程 线程编程是一种并发程序编程，正如在操作系统领域中传统研究一样。本节将介绍并发编程的概念。Bacon[1998]透彻地解释了下列概念：竞争条件、临界区（Bacon把它叫做临界区域）、监视器、条件变量、信号量。

许多线程程序是用诸如C这样的常规语言编写的，这些语言一般有扩充线程库。为Mach操作系统开发的C线程包[Cooper 1988]便是其中的一个例子。最近，IEEE POSIX 1003.4a线程标准，也就是所说的*pthreads*，已经被广泛采用了。Boykin等[1993]基于Mach系统描述了C线程和*pthreads*。

一些语言提供了对线程的直接支持，其中包括Ada95[Burns and Wellings 1998]、Modula-3[Harbison 1992]和最近的Java[Oaks and Wong 1999]。下面将简单介绍一下Java线程。

如许多线程实现一样，Java提供了创建、终结和同步线程的方法。Java的*Thread*类包括图6-8中的构造函数和管理方法。*Thread*和*Object*的同步方法在图6-9中列出。 225

Thread(ThreadGroup group, Runnable target, String name)
创建一个初始为挂起状态的新线程，它将属于*group*组，其标识名为*name*；这一线程会执行*target*的*run<>*方法

setPriority(int newPriority), getPriority()
设置并返回线程的优先级

run()
线程执行其目标对象的*run()*方法，如果其目标对象有*run()*方法。否则它执行自己的*run()*方法（*Thread*实现可运行）

start()
将线程的挂起状态转换为运行状态

sleep(int millisecs)
将线程转换为挂起状态，并持续指定的时间

yield()
进入就绪状态并唤醒调度管理

destroy()
终结线程

图6-8 Java 线程的构造函数和管理方法

thread.join(int millisecs)
调用线程被阻塞指定的时间直到*thread*终结

thread.interrupt()
中断*thread*：使其从导致它阻塞的方法（如*sleep*()）返回

object.wait(long millisecs, int nanosecs)
阻塞调用线程直到*Object*的*notify*()或*notifyAll*()方法唤醒了线程，或者线程被中断或过去了指定的时间

object.notify(), object.notifyAll()
分别唤醒一个或多个在*Object*上调用*wait*()方法的线程

图6-9 Java线程同步调用

线程生存期 新线程和它的创建者在同一台Java虚拟机（JVM）上，一开始处于挂起状态。在执行了*start*()方法以后新线程处于运行状态，此后，它执行在其构造函数中指定的一个对象的*run*()方法。JVM和在其上的线程都是在操作系统的一个进程内执行的。线程可以被赋予一个优先级，因此，支持优先级的Java实现会在低优先级线程之前运行高优先级线程。当线程从*run*()方法返回或其*destroy*()方法被调用时，线程的生存期便结束了。

程序可以按组管理线程。在线程创建时，它可以被指定属于一个组。当有许多应用程序

共存于同一JVM时，线程组是非常有用的。一个使用组的例子是利用它的安全性：在默认情况下，一个组内的线程不能执行其他组中的线程的管理操作。例如，一个应用程序线程不能恶意打断系统窗口（AWT）线程。

线程组也方便了对线程相关优先级（在支持优先级的Java实现中）的控制，这对浏览器运行Java小程序和Web服务器运行创建动态Web页面的*servlet*程序[Hunter and Crawford 1998]都是有益的。在小程序或*servlet*内部的无授权的线程只能创建属于自己线程组的线程，或者将新线程加入到其内部生成的子孙线程组中（其详细的限制由它所在的安全管理器决定）。浏览器和服务器可以将属于不同小程序线程或*servlet*的线程加入到不同的组中并将这些组（包括子孙线程组）作为一个整体设置一个最大的优先级。小程序线程和*servlet*线程不能超越其管理器线程设置的线程组的优先级，因为它们不能再用*setPriority*()设置优先级。

线程同步 编程者必须很小心地编写多线程的进程程序。其中主要的困难是共享对象和用于线程协调与合作机制的技术。每一个线程的方法中局部变量是其私有的（线程有一个私有栈）。然而，线程没有静态（类）变量或对象实例变量的私有拷贝。

例如，考虑本节前面介绍的共享队列的例子，I/O线程和工作线程在一些服务器线程体系结构中传输请求。线程并发处理诸如队列这样的数据结构时必然会产生竞争条件。除非仔细协调线程的指针操作，否则必然会引起队列中请求的丢失或重复处理。

Java提供了*synchronized*关键字以便程序员为线程的协调指定监视器。程序员可以指定完整的方法，也可以指定任意代码块属于某个对象的监视器。监视器可以保证在监视器内任一时刻最多只有一个线程在执行。通过将*Queue*类的*addTo*()和*removeFrom*()方法指定为*synchronized*方法，我们可以将例子中I/O线程和工作线程的操作串行化。在这些方法中，所有访问变量的操作都是互斥完成的。

通过任何用作条件变量的对象，Java允许线程被阻塞或唤醒。需要阻塞等待某一条件的线程调用该对象的*wait*()方法。所有的Java对象都实现了这一方法，因为它属于Java的根*Object*类。另外一个线程调用*notify*()方法为至多一个等待该对象的线程解除阻塞状态，也可以调用*notifyAll*()方法为所有等待该对象的线程解除阻塞状态。这两个唤醒方法也属于*Object*类。

作为一个例子，当一个工作线程发现没有可处理的请求时，它会调用*Queue*类对象的*wait*()方法。当I/O线程在队列中加入一个请求时，它会调用队列的*notify*()方法唤醒工作线程。

图6-9给出了Java的同步方法。除了已经提到的同步原语外，*join*()方法将阻塞其调用者，直到目的线程终结为止。*interrupt*()方法用于提前唤醒等待进程。Java实现了所有标准的同步原语，例如信号量。但程序员必须注意，Java的监视器只应用于对象的同步代码。一个类可能会同时包括同步和非同步的方法。还要注意，Java对象实现的监视器只包括一个隐式条件变量，而通常，一个监视器可以包含多个条件变量。

线程调度 抢占性和非抢占性线程调度策略之间的区别非常明显。在抢占性调度中，线程可以在执行中的任一时刻因被其他线程抢占而挂起，甚至当已经抢占处理器的线程正准备运行时也是如此。在非抢占性调度中（有时也叫协同调度），当系统准备让某一线程退出运行并让其他线程运行时，此线程并不退出，它要运行直到进行一次线程系统的调用（例如系统调用）为止。

非抢占性调度的好处在于每一个不包含线程系统调用的代码区都自动地成为临界区。这样就很方便地避免了竞争条件。另一方面，因为非抢占性调度的线程是独占式运行的，所以

它们不能利用多处理器。要小心对待长期运行的不含线程系统调用的代码区。程序员可能需要在程序中加入一个*yield*()调用，其惟一的作用在于使其他线程能被调度执行。非抢占性调度的线程也同样不适合于实时应用系统，在实时应用系统中，事件必须在规定的时间内被处理。

尽管有在Java上实现的实时系统[www.rtj.org]，但在默认情况下，Java不支持实时处理。例如，处理音频和视频数据的多媒体应用程序对通信和处理（例如过滤和压缩）有实时要求[Govindan and Anderson 1991]。第15章将讨论实时系统线程调度的要求。进程控制是实时领域中的另一个例子。一般来说，每一实时领域都有其本身的线程调度要求。因此，有时需要应用程序实现其本身的调度策略。考虑到这一点，下面我们将介绍线程的实现。

线程实现 许多操作系统内核，包括Windows NT、Solaris、Mach和Chorus，提供对多线程进程的支持。这些内核提供了用于线程创建和管理的系统调用，同时它们还负责线程调度。其他一些内核只提供了单线程进程的抽象，必须由一个与应用程序相联系的过程库实现多线程的进程。在这种情况下，内核不知道这些用户级的线程，因此就不能调度它们。这时，由线程的运行库组织线程调度。通过一个阻塞系统调用，一个线程可以阻塞一个进程和进程中所有线程，这样可以开发内核的异步（非阻塞的）输入输出功能。类似地，也可以利用内核提供的定时器和软件中断来实现线程的时间片机制。

226 ~ 228

当内核不提供对多线程的进程的支持时，实现用户级的线程会遇到以下问题：

- 一个进程内的线程不能利用多处理器。
- 一个线程在遇到页失配时会阻塞整个进程和进程内的所有线程。
- 不能按统一的优先级方案调度不同进程中的线程。

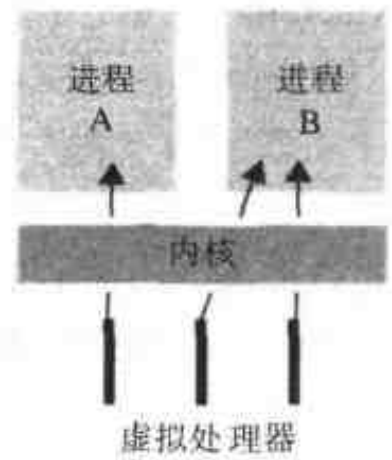
另一方面，相对于内核级的线程实现，用户级的线程实现有如下优点：

- 某些线程操作的系统开销小。例如，在同一进程内的线程切换不必进行系统调用。系统调用需要陷入内核，其开销是比较大的。
- 因为线程调度模块是在内核外部实现的，用户可以定制或改变其功能以满足某些特殊应用的需要。诸如多媒体处理的实时性质这样的有特殊性的应用系统对线程调度的要求各不相同。
- 能提供的用户级线程比内核默认支持更多。

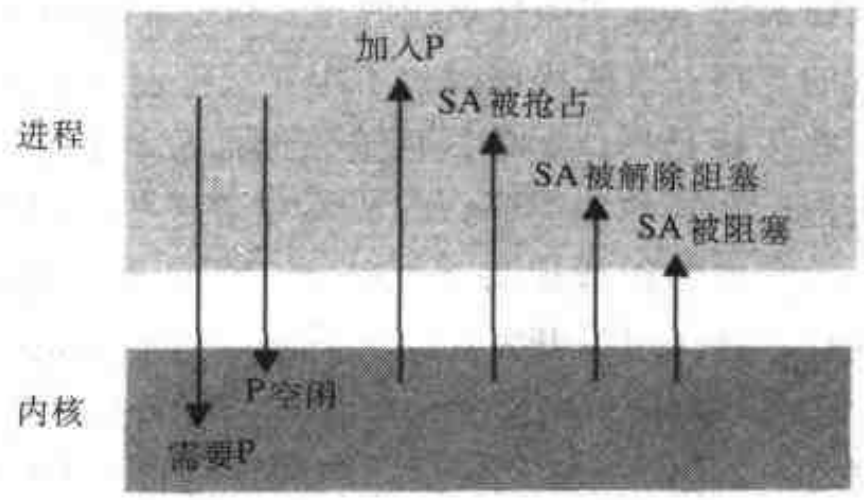
将用户级线程实现和内核级线程实现的优点组合起来是可能的。一种已经被应用的方法是用户级代码为内核线程调度提供调度提示，例如Mach内核[Black 1990]。另外一种方法是采用层次化调度，已被Solaris 2操作系统采用。每一个进程可以创建一个或多个内核级线程，在Solaris中叫“轻量级进程”，其同时也支持用户级线程。用户级调度器将每一个用户级线程指定到一个内核级线程上。这一模式可以充分利用多处理器，同时也可以获得在用户级上进行线程创建和线程切换的好处。这一模式的缺点在于它缺少灵活性：如果一个线程在内核被阻塞了，那么所有被指定在其上的用户级线程，不管是否具备运行条件，都将不能运行。

为了进一步提供更大的有效性和灵活性，一些研究项目开发了层次化调度。这其中包括所谓的调度器激活[Anderson *et al.* 1991]、Govindan和Anderson[1991]的多媒体工作、Psyche多处理器操作系统[Marsh *et al.* 1991]、Nemesis内核[Leslie *et al.* 1996]和SPIN内核[Bershad *et al.* 1995]。设计这些系统的原因是：用户级调度对内核的需要并不仅仅是映射了用户级线程的一组由内核支持的线程。用户级线程调度还需要内核将与用户级线程调度决定相关的事件通知它。下面将介绍调度器激活的设计，以使读者能详细了解这一点。

Anderson等人[1991]提出的快速线程包是一个层次化、基于事件调度系统的实现。他们考虑到主要的系统组件是一个运行在一个或多个处理器上的内核和一组在其上运行的应用程序。
229 每一个应用进程包括一个用户级的调度器，它负责管理进程内的线程。内核负责给进程分配虚拟处理器。为一个进程指定的虚拟处理器的数量取决于下列因素：应用程序的需求、它们相对的优先级以及对处理器总的需求量。图6-10a描述了一个包含3个处理器的计算机的例子，其中，内核将一个虚拟处理器分配给进程A，用于执行一个相对低优先级的任务，同时将两个虚拟处理器分配给进程B。因为内核可能将不同的物理处理器分配给进程，以保证达到其指定的处理器数量，所以称为虚拟处理器。



a. 将虚拟处理器分配给进程



b. 在用户级调度器和内核之间的事件
关键词：P=处理器；SA=调度器激活

图6-10 调度器激活

分配给进程的虚拟处理器的数量也可以变化。进程可以让出一个它不再需要的虚拟处理器，它也可以请求一个额外的虚拟处理器。例如，如果进程A请求获得一个额外的虚拟处理器而进程B终结了，那么内核可以将一个处理器分配给进程A。

图6-10b描述了当一个虚拟处理器“空闲”并不再需要时，或者当进程请求获得额外的虚拟处理器时，进程将通知内核。

图6-10b还描述了当4种类型的事件发生时，内核会通知进程。*调度器激活*（SA）是一个从内核到进程的调用，它通知进程的调度器有一个事件发生。从低层（内核）进入代码体通常被称为上调。内核通过从物理处理器的寄存器中载入上下文来创建一个SA，其上下文使进程的代码在用户级调度器指定的过程地址处开始执行。这样，一个SA也是虚拟处理器上一个时间片的分配单元。用户级调度器把处于就绪状态的线程指定给当前正在执行的SA集合。SA的数量最多不能超过内核分配给进程的虚拟处理器的数量。

下面是内核通知用户级调度器（下面将简称为调度器）的4种事件：

- *虚拟处理器已分配* 内核已经将一个新的虚拟处理器分配给进程，并且这正是其上的第一个时间片。调度器可以用就绪状态的线程的上下文载入SA，线程重新开始执行。
230
- **SA被阻塞** SA在内核中被阻塞。内核准备使用一个新的SA通知调度器，调度器将相应线程的状态设置为阻塞，并分配一个就绪线程给用于通知消息的SA。
- **SA被解除阻塞** 在内核中阻塞的SA解除阻塞，可以再次在用户级执行。调度器现在可以将相应的线程排到就绪线程队列中。为了创建用于通知消息的SA，内核或者为进程分配一个新的虚拟处理器，或者抢占在同一进程上的另一个SA。在后一种情况中，内

核同时还将抢占事件发送给调度器，调度器可以对SA重新评估线程的分配情况。

- SA被抢占　内核从进程处夺走一个SA（虽然可以通过将一个处理器分配给在同一进程上新的SA的方式完成以上工作），调度器将被抢占的线程放到就绪队列中，并重新评估线程分配情况。

因为进程的用户级调度器可以采用多种策略在低级事件的基础上将线程分配给SA，所以层次化调度方式更具有灵活性。内核总按同一方式运行，它不会影响用户级调度器的行为，但是它通过事件通知以及提供阻塞和抢占线程的寄存器状态来协助调度器工作。这一方式可能是有效的，因为如果有一个虚拟处理器空闲且可以运行线程，那么用户级线程就无需等待在就绪状态。

6.5 通信和调用

下面我们将通信作为调用机制实现的一部分进行讨论。调用包括远程方法调用、远过程调用或事件通知，调用的作用是在不同的地址空间执行对资源的操作。

通过考虑下面关于操作系统的问题，我们可以概括出操作系统设计中涉及的问题及概念：

- 操作系统提供什么样的操作原语？
- 操作系统支持什么样的协议以及通信实现的开放性有多大？
- 采取哪些步骤以使通信尽可能地有效？
- 为高延迟和断连操作提供了哪些支持？

通信原语　一些为分布式系统设计的内核提供的通信原语与第5章描述的调用类型相适应。例如，Amoeba[Tanenbaum *et al.* 1990] 提供了*doOperation*、*getRequest*和*sendReply*这样的通信原语。Amoeba、V系统和Chorus系统都提供了组通信原语。在内核中加入相对高层的通信功能可以提高效率。例如，如果中间件在UNIX连接（TCP）套接字上提供RMI，那么客户就必须为每次远程调用进行两次通信系统调用（套接字的读和写）。而在Amoeba上，它只需要调用一次*doOperation*。用组通信在系统调用上的开销可能节省更大。 231

实际上，是中间件而不是内核提供了现在系统中所能找到的大多数高级通信方式，包括RPC/RMI、事件通知和组通信。在用户级上开发这样复杂软件系统的代码比在内核上开发要容易得多。开发者通常在提供对因特网标准协议访问的套接字上实现中间件——通常使用有连接的TCP套接字，有时也使用无连接的UDP套接字。使用套接字的主要原因是可移值性和互操作性：中间件需要尽可能地在多种操作系统之上操作，并且诸如UNIX和Windows系列这样的操作系统通常都提供了类似的套接字API以便对TCP和UDP协议进行访问。

尽管广泛使用的是由公共内核提供的TCP和UDP套接字，但研究人员还是在一些试验性的操作系统内核中进行了一些低开销的通信原语研究。6.5.1节将进一步讨论其性能问题。

协议和开放性　操作系统提供标准的协议，这些协议能帮助不同平台的中间件实现完成互连，这是对操作系统提出的主要要求之一。在20世纪80年代，一些研究性的操作系统内核将自己的网络协议与RPC交互结合起来，其中有著名的Amoeba RPC[van Renesse *et al.*1989]、VMTP[Cheriton 1986]和Sprite RPC[Ousterhout *et al.* 1988]。然而，除了自身的研究环境之外，这些协议并没有被广泛应用。相反，Mach 3.0和Chorus内核（也包括20世纪90年代的内核，如L4[Härtig *et al.*1997]）的设计者们决定采用完全开放的网络协议。这些内核只提供在本地进程之间的消息传递机制，并将网络协议处理留给在内核上运行的一个服务器。

如果每日需要访问因特网，对于操作系统来说需要对最小的网络设备达到在TCP和UDP层的兼容性。操作系统也需要提供支持使中间件能利用新的低层协议。例如，用户希望在不更新其应用程序的情况下利用诸如红外和射频(RF)传输这样的无线技术。这就需要相应的协议，例如可以将支持红外网络和蓝牙技术的IrDA协议和支持RF网络的HomeRF协议集成。

协议通常被安排在一个层次的栈中（见第3章）。许多操作系统允许新的层次被静态地集成，这种集成是靠加入诸如IrDA这样的永久安装的协议“驱动器”作为其新的一层来完成的。相反，动态协议合成是一项能灵活合成协议栈的技术，其合成可以满足特定应用的需要，并能够根据平台当前的连通性利用可用的物理层。例如，当用户在路上时运行在笔记本电脑上的Web浏览器可以利用广域无线连接，当用户返回办公室时，则可以利用快速以太网连接。

动态协议合成的另一个例子是用户可以在无线网络层上使用用户定制的请求－应答协议
232 来减少往返延迟。标准的TCP协议实现已经被证实不能在无线网络介质上很好地工作[Balakrishnan *et al.*1996]，因为相对于有线介质，在无线介质上可能会有更高的丢包率。原则上，一个像HTTP这样的请求－应答协议只有通过直接使用无线传输层，而不是使用中间的TCP层，才能使无线连接的结点有效地工作。

在UNIX流设施[Ritchie 1984]的设计中支持协议合成。最近，Horus[van Renesse *et al.* 1995]和x-kernel[Hutchinson and Peterson 1991]都提供了动态协议合成。

6.5.1 调用性能

在分布式系统的设计中，调用性能是一个非常关键的因素。如果设计者在地址空间之间分离的功能越多，就必须越多地使用远程调用。客户和服务器在其生存期内可能会执行上百万条与调用有关的操作，这样就应该有小部分时间计入调用开销。网络技术一直在发展，但调用时间并没有因网络带宽的增加而成比例地减少。本节将解释为什么在调用时间上软件的开销会比网络的开销大得多，至少在局域网或企业内部网上是如此。这与在因特网上的远程调用（例如获得一个Web资源）形成鲜明对比，在因特网上，网络等待时间通常变化很大，平均值也很高，带宽经常很小，服务器的负载主要花费在对每一个请求的处理上。

RPC和RMI的实现问题已经成为研究的主题，这是因为通用的客户－服务器处理过程广泛采用了这些机制。许多研究已经涉及到在网络上的调用，并且特别研究了如何更好地利用高性能网络实现调用机制[Hutchinson *et al.* 1989、van Renesse *et al.* 1989、Schroeder and Burrows 1990、Johnson and Zwaenepoel 1993、von Eicken *et al.* 1995、Gokhale and Schmidt 1996]。当然，我们还要介绍一个重要的、特殊的在同一计算机不同进程之间进行RPC的例子[Bershad *et al.*1990、Bershad *et al.* 1991]。

调用开销 调用一个传统过程或方法，进行一次系统调用，发送一条消息，远程过程调用和远程方法调用，这些都是调用机制的例子。每种机制都导致在调用程序和对象作用域之外的代码被执行。一般每种机制都涉及到将参数传递给调用代码的通信，以及将结果返回给调用者的通信。调用机制可以是同步的，例如传统调用和远程过程调用，也可以是异步的。

除了是否同步，调用机制间与性能相关的重要的区别还包括是否引起域转换（即是否跨越了一个地址空间）、是否涉及网络通信以及是否涉及线程调度和切换。图6-11显示了一个系统调用、在同一计算机上不同进程之间的远程调用和在分布式系统上不同结点的计算机进程
233 之间的远程调用这3个例子。

在网络上的调用 一个空的RPC（类似于一个空的RMI）是执行一个空的过程并且无返回值的不带参数的RPC。它负责交换不包含任何用户数据、只包含很少的系统数据的消息。现在，在两台500MHz的PC机和100Mbps的LAN上进行一次空的RPC，其时间开销在几十ms这个数量级内，而一个传统的空过程调用只需小于1μs的时间。假设这一空的RPC调用总共有100字节的数据需要传输，在带宽为100Mbps的网络上，这些数据总共的数据传输时间大约是0.01ms。显然，大部分的延迟（客户调用RPC总共花费的时间）来源于操作系统内核代码和用户级RPC执行代码的执行时间开销。

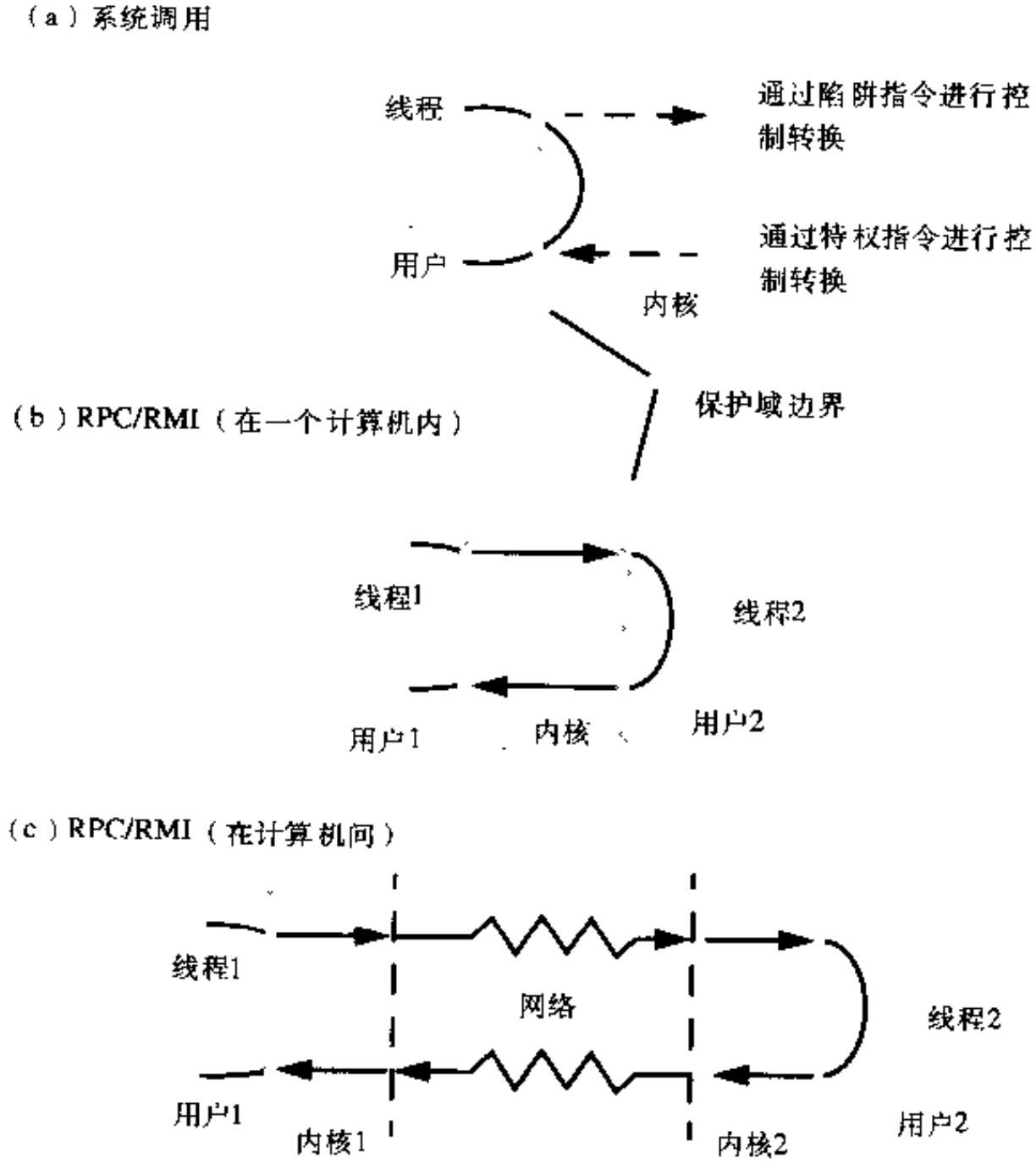


图6-11 在地址空间之间的调用

空调用（RPC、RMI）开销是非常重要的，因其度量了一个固定的开销，也就是等待时间。随着参数和结果数据量的增加，调用开销也会增加，但在许多情况中，空调用的等待时间比其他类型的延迟要大得多。 234

考虑一个从服务器上获得定量数据的RPC。它包含一个整型请求参数，用于指明所需数据量的大小。在返回结果时，它包括两个应答参数，一个整型参数表示调用是成功还是失败（客户可能提供一个非法的数据量大小），并且，在调用成功的情况下，另一参数为从服务器返回的一个字节数组。

图6-12示意性地表示了请求数据量与客户延迟的关系。在数据量的大小达到一个同网络数据包大小相近的阈值之前，延迟和数据量的大小基本上成正比。超过这一阈值之后，为了

传输额外的数据，系统至少要多传一个额外的包。根据协议，为了确认这个额外的包，可能还需要传一个数据包。每次传输包的数量增加时，在图上便会出现一个跳跃。

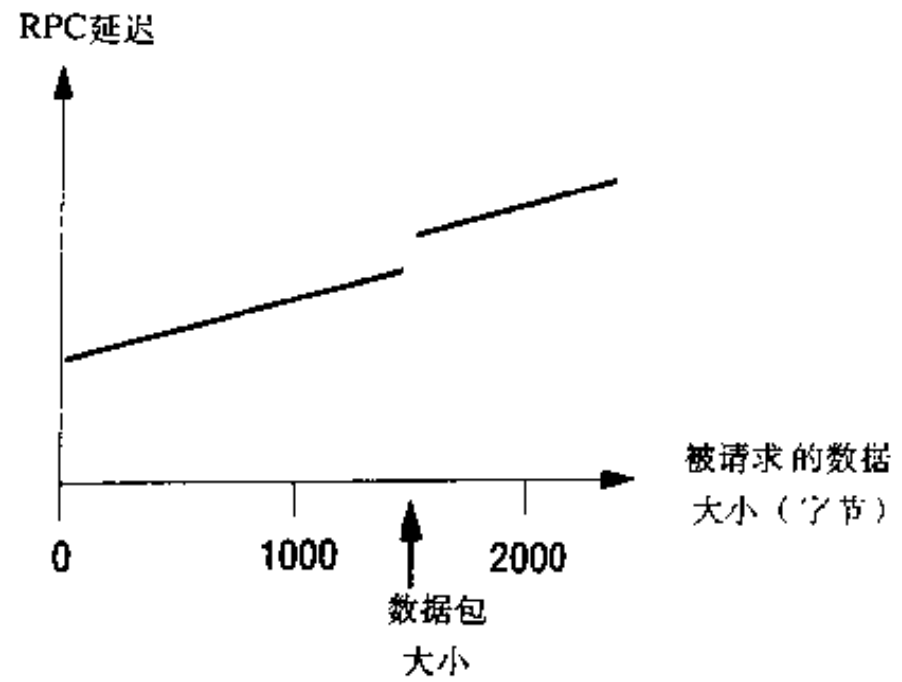


图6-12 RPC延迟与参数数据量大小的关系

在RPC的实现中，延迟并不是被关注的惟一数字：当数据传输量比较大时，RPC带宽（或吞吐量）也会被关注。它表示在一个RPC内不同计算机间的数据传输率。从图6-12中我们可以知道：当固定处理的时间开销占总开销的绝大部分时，对于少量的数据，RPC的带宽相对较低。随着数据量的增加，RPC带宽会增加，这是因为固定处理的时间开销变得相对较小。Gokhale和Schmidt[1996]引用了一个例子：在具有155Mbps带宽的ATM网络上，当一个在工作站之间的RPC传输64KB数据时，其RPC的吞吐量为80Mbps。即花费0.8ms传输64KB的数据，它与上面提到的在100Mbps以太网上进行一个空的RPC属于同一数量级。

回想一下一个RPC包括如下步骤（RMI也包含类似的步骤）：

- 一个客户存根程序将调用参数编码为消息，并将其发送出去，然后接收应答消息，并将其解码。
- 在服务器端，一个工作线程接收到达的请求，或者由一个I/O线程负责接收请求，并将其传递给工作线程；不论哪种情况，都要调用合适的服务器存根程序。

235

- 服务器存根程序将请求消息解码，调用指定的程序，然后将程序返回值编码并发送出去。

下面是除网络传输时间之外计入远程调用延迟的主要的因素：

- *编码* 编码和解码涉及复制和转换数据，当数据量增加时，这成为一个重要的时间开销。
- *数据复制* 即使在编码之后，在一个RPC过程中，消息可能会被复制数次：
  1. 跨越用户－内核边界，在客户或服务器地址空间与内核缓冲区之间复制。
  2. 在每个协议层之间（例如，RPC/UDP/IP/以太网）复制。
  3. 在网络接口和内核缓冲区之间复制。

网络接口和主存之间的传输通常是靠直接内存访问（DMA）来完成的，其他复制是由处理器处理的。

- *包初始化* 包括初始化协议头部和协议尾部，其中包括校验位。它的开销部分地与需传输数据量的大小成正比。

- 线程调度和上下文切换 包括如下部分：
  1. 在一个RPC过程中会产生多次系统调用（即上下文切换），正如存根程序调用内核的通信操作。
  2. 调度一个或多个服务器线程。
  3. 如果操作系统采用一个单独的网络管理进程，那么每次发送信息会涉及它们之间线程的切换。
- 确认等待 RPC协议的选项可能会影响到延迟，特别是当有大量数据需要传输的时候。

小心设计操作系统可以帮助减少这些开销。在www.cdk3.net/oss上可以找到Firefly RPC的实例研究，其中还包括在中间件实现中可用的技术。我们已经介绍过了合适的操作系统是怎样支持线程以减少多线程开销的。操作系统通过内存共享机制可以减少内存拷贝开销。

内存共享 共享区域（在6.4节介绍过）可以被用来在用户进程和内核之间或者在用户进程之间快速通信。通过在共享区域中写数据和读数据可以实现数据通信。这样可以高效地传输数据，而不需要从内核地址空间或向内核地址空间拷贝数据。但系统调用和软件中断可能需要同步——例如，当用户进程写完数据后应立刻将其传输，或者当内核写完数据后，用户进程应立刻获得。当然，只有在共享区域带来的优点大于建立它所带来的开销时，才会使用共享区域。 236

即使使用共享区域，内核仍然需要从其缓冲区向网络接口拷贝数据。U-Net体系结构[von Eicken *et al.* 1995]甚至允许用户级的代码直接访问网络接口，因此，用户级的代码可以不用任何拷贝就能把数据传输到网络。

协议的选择 在TCP协议上进行请求－应答交互时客户所经历的延迟并不一定比运行在UDP上的长，有些时候甚至还要短一些，特别是在传输大量信息的时候。然而，在诸如TCP这样的协议之上实现请求－应答交互必须要小心，因为这些协议并非为此目的而设计的。特别是TCP的缓冲区机制会妨害其性能，其连接开销与UDP相比也处于劣势，除非在一个连接上的数据量相当大，这样才可以忽略单个请求的开销。

在Web请求中，TCP连接上的开销就特别明显，这是因为HTTP 1.0为每一个请求建立一个独立的TCP连接。当建立连接时，客户的浏览器被延迟。而且，TCP的慢启动算法具有延迟HTTP数据传输的作用，而这在许多情况下是不必要的。在面对可能的网络阻塞时，TCP慢启动算法采用一种悲观操作，即在接收到确认前，先只向网络传输一个小数据窗口。Nielson等[1997]讨论了HTTP1.1怎样使用持久连接，持久连接能在几个调用过程内持续存在。只要对同一Web服务器有多个调用请求，初始的连接开销就被分摊在几个调用过程中。这是很有可能的，因为用户经常从同一网址获得多个页面，而每个页面可能包含多个图像。

Nielson等也发现取代操作系统的默认缓冲行为对调用延迟有显著影响。较好的机制是收集多个小信息，然后将它们一起发送出去，而不是分别用独立包将其发出去，因为这样会产生上面所描述的每一个包的等待时间。为此，操作系统并不需要在每一次套接字的*write*()调用后立即将数据分发到网络上。默认情况下，操作系统要等待缓冲区满或者将超时作为将数据分发到网络上的标准。

Nielson等发现在HTTP 1.1中默认的操作系统缓冲行为会因为超时而产生明显的不必要的延迟。为了避免这些延迟，他们改变了内核的TCP设置，并且在HTTP请求边界上强制进行数据分发。这是一个很好的例子，说明了操作系统的实现策略是如何帮助或阻碍中间件的。

在一个计算机内的调用 Bershad等[1990]进行的一项研究表明：在客户－服务器环境中，大多数跨地址空间的调用并不像期望的那样发生在计算机间，而是发生在计算机内部。将服务功能放置在用户级服务器中的趋势意味着有越来越多的调用是针对本地进程的。特别在客户所需要的数据可能在本地服务器上这种情况。在一个计算机内的RPC开销作为系统性能的
237 一个参数变得越来越重要。这些考虑表明，这种本地类型的调用应该被优化。

图6-11说明除了底层的消息传递在本地进行之外，在一个计算机内的跨地址空间的调用和在计算机间的几乎完全一样。实际上，这是经常实现的模型。Bershad等[1990]为在同一机器上两个进程之间的调用开发了一种更有效的机制叫做轻量级RPC(LRPC)。LRPC设计注重基于数据拷贝和线程调度的优化。

首先，他们提出利用共享区域提供客户－服务器通信比较有效，这里在服务器和每个本地客户之间使用不同（私有）的区域。这样一个区域包含一个或多个A栈（如图6-13所示）。在这种设计中，客户和服务器可以通过一个A栈传递参数和返回结果，而不必涉及到在内核和用户地址空间之间的数据复制。客户和服务器存根程序也采用同一栈。在LRPC中，参数只被复制一次，即当它被编码后进入A栈时。在一个等价的RPC中，数据被复制了4次：从客户存根程序的栈中取出复制到消息中；从消息复制到内核缓冲区；从内核缓冲区复制到服务器消息中；从服务器消息复制到服务器存根程序的栈中。在共享区中可能有数个A栈，因为可能同一时间在同一客户上有多个线程调用服务器。

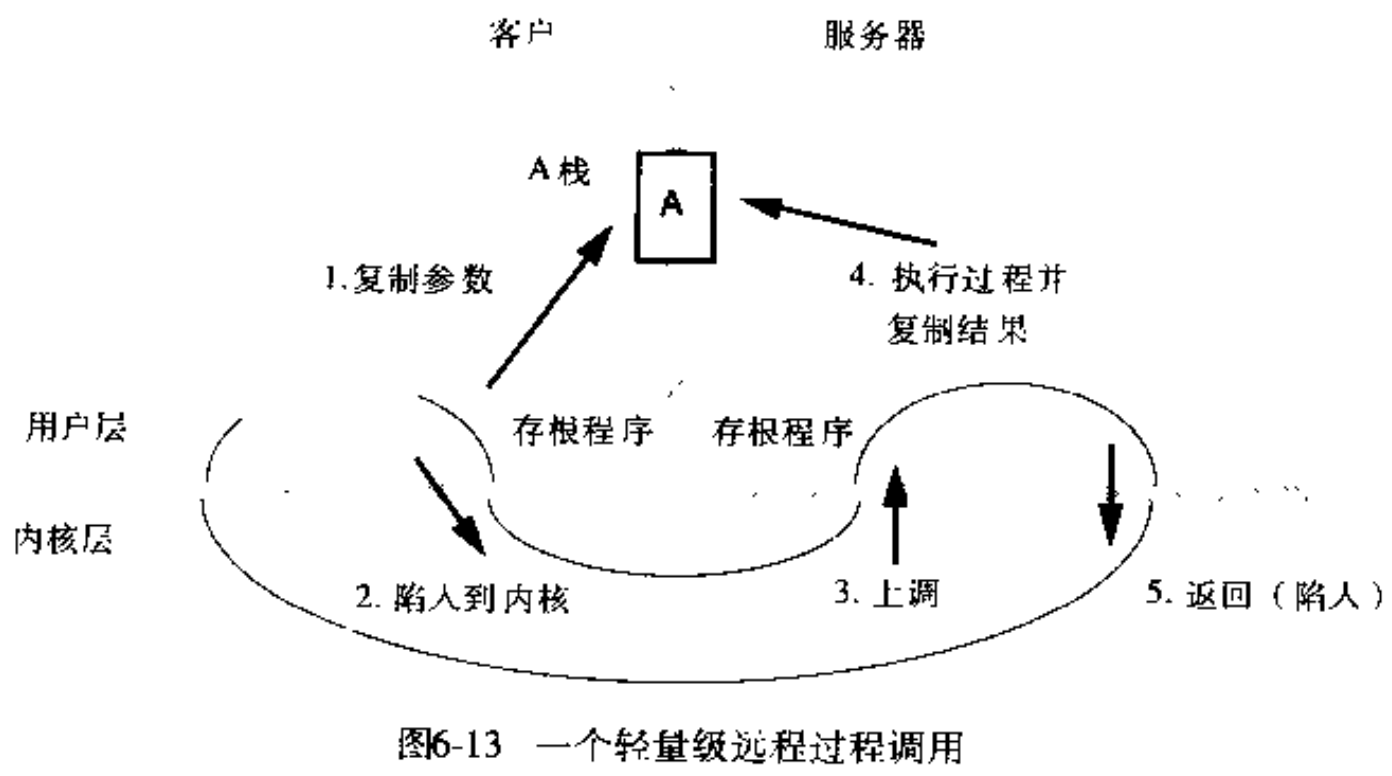


图6-13 一个轻量级远程过程调用

Bershad等也考虑到线程调度的开销。比较在图6-11中的系统调用和远程过程调用可以发现一些不同点：当一个系统调用发生时，大多数内核并不调度一个新的线程处理调用，而是在调用线程中进行一次上下文切换，这样它便可以处理系统调用。在一个RPC中，一个远程过程可能与客户的线程在不同的计算机上，这样服务器上的一个线程必须被调度来执行被调用的过程。然而，若服务器和客户在同一台机器上，客户线程（它可能被阻塞）调用在服务器地址空间内的过程，其执行效率可能更高。

在这种情况下，服务器的设计与以前描述过的服务器有所不同。服务器输出一系列过程
238 以备调用，而不是建立一个或多个线程来监听端口是否有调用请求。当本地进程中的线程开始调用服务器输出的过程时，本地进程中的线程就可进入服务器的执行环境。需要调用服务

器操作的客户必须首先绑定服务器的接口（没有在图中显示）。上述过程是通过内核通知服务器的，当服务器响应内核并提供一系列允许访问的过程地址时，内核便允许客户调用服务器的操作。

图6-13表示了一个调用。客户线程首先通过陷入内核来进入服务器的执行空间。内核检查其合法性并只允许上下文切换到合法的服务器处理过程上。如果它是合法的，内核便将线程的上下文切换到服务器执行环境中被调用的过程上。当此服务器中的过程运行结束并返回后，线程返回到内核，内核将线程切换回客户执行环境。其中需要注意的是：客户和服务器采用存根程序对应用程序员隐藏其细节。

对LRPC的讨论 只要有足够的调用来抵消内存管理的开销，在本机上LRPC比RPC有更高的效率，这一点是无可置疑的。Bershad等统计得到LRPC的延迟小于本地RPC的1/3。

Bershad的LRPC实现并没有牺牲位置透明性。一个客户存根程序在绑定时检查其记录，判断服务器是在本地还是在远端，与之对应地选择采用LRPC还是RPC。应用程序并不知道使用的是LRPC还是RPC。然而，当一个资源从本地服务器转移到远程服务器上或反之的情况，将很难实现迁移透明性，这是因为需要改变调用机制。

在其后的工作中，Bershad等[1991]描述了几种性能改进方法，主要适用于多处理器操作。其改进主要注重于避免陷入内核以及在调度进程中避免不必要的域转换。例如，当一个客户线程试图调用服务器过程时，如果在服务器的内存管理上下文中有一个处理器是空闲的，线程应该被转移到此处理器上。这种方式避免了域的转换，同时，客户的处理器可以被客户的其他线程复用。这些改进还包括两层（用户和内核）线程调度的实现（见6.4节的介绍）。

6.5.2 异步操作

我们已经讨论了操作系统是如何帮助中间件层提供有效的远程调用机制。但是，我们也观察到在因特网环境中长延迟、低带宽和高服务器负载的影响可能抵消操作系统提供的好处。我们还应该考虑网络的断连和重新连接，网络的断连和重新连接被认为是造成超长延迟通信的原因。用户的移动计算机并不是在所有时间都连接在网络上。即使使用广域无线访问技术（例如，使用GSM），他们也可能随时断连，例如，当用户乘坐的火车进入了屏蔽信号的隧道。

异步操作是为了应付长延迟的一种常用技术。它在两种编程模型中出现：并发调用和异步调用。这些模型主要出现在中间件领域，而不是出现在操作系统内核设计。但当我们讨论到调用性能时，还是应该考虑其作用。 239

使调用并发化 在第一个模型中，中间件只提供阻塞型调用，但应用程序产生多个线程来并发执行阻塞型调用。

Web浏览器是其中的一个很好的例子。一个Web页面通常包含多个图像。浏览器必须为每个图像执行独立的HTTP *GET*请求（因为标准的HTTP1.0 Web服务器只支持对单个资源的请求）。浏览器不需要按特定的顺序来获得这些图像，因此它可以发出并发请求，通常在同一时间内最多可达到4个并发请求。这种方式获得所有图像的时间通常比串行请求的时间延迟短。通常情况下，不仅仅是通信延迟减少了，浏览器也可以将通信和图像绘制并行执行。

图6-14表示了这种在一个客户和一个单处理器的服务器间交错调用的好处。在串行的情况下，客户将参数编码并调用*Send*操作，然后等待服务器的应答到达，而服务器执行*Receive*操作，解码并处理结果。在此之后它才能执行第二个调用。

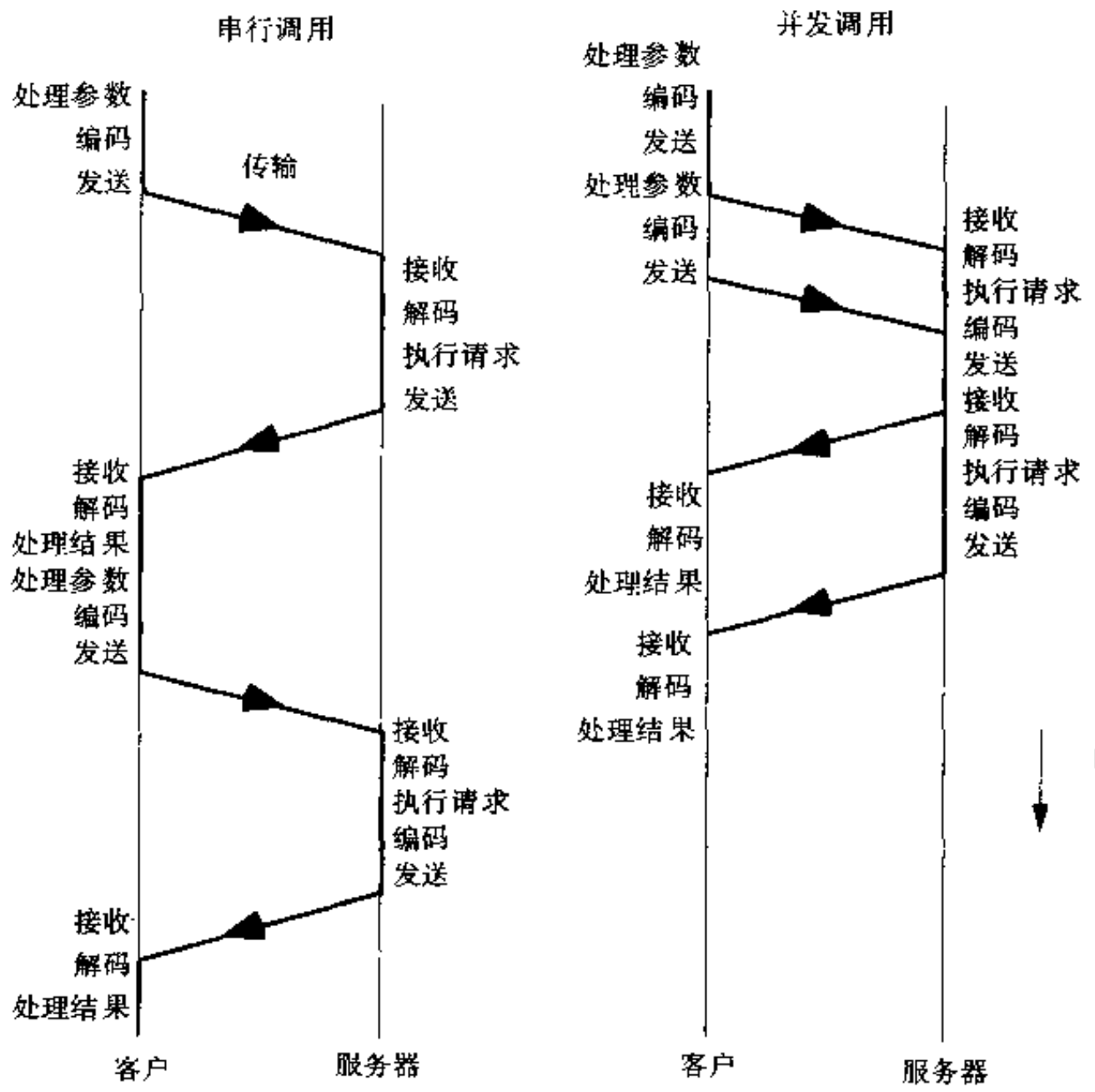


图6-14 串行调用和并发调用的时序

在并发情况下，第一个客户线程将参数编码并调用*Send*操作。然后，第二个线程立即执行第二个调用。每一个线程等待接收各自的调用结果。如图所示，总时间一般低于串行调用。类似地，当客户线程对多个服务器执行并发调用时也能获得减少总调用时间的效果；如果客户在多处理器上执行，则可能获得更大的吞吐量，这是因为两个线程的操作可以并行进行。

回到HTTP的例子，前面介绍的Nielson等[1997]的研究还衡量了在持久连接上并发执行HTTP 1.1调用（他们称为管道）的结果。他们发现：只要操作系统为缓冲区提供了合适的接口以覆盖默认的TCP行为，管道就可以减少网络流量并能为客户提高性能。

异步调用 异步调用是对调用者调用的一次异步执行。也就是说，调用者进行非阻塞调用，只要创建了调用请求信息并准备分发，调用便结束了。

有些时候，客户不需要任何回复（除非目标主机连接不上时需要故障信息），例如，CORBA的单向调用包含或许语义。否则，客户使用单独的调用来收集调用结果。例如，Mercury通信系统[Liskov and Shrira 1988]支持异步调用。一个异步操作返回一个叫*promise*的对象。最后，当调用成功或注定要失败时，Mercury系统将系统状态和返回值放在*promise*中。调用者使用*claim*操作来从*promise*中获得结果。*claim*操作一直被阻塞直到*promise*准备好，由*promise*返回调用的结果或异常信息。*ready*操作可以不阻塞地测试*promise*，它对应*promise*的就绪或阻塞状态分别返回true或false。

持久异步调用 诸如Mercury调用和CORBA单向调用这类传统异步调用机制是在TCP流上实现的。当TCP流中断时（例如网络连接中断或目标主机崩溃）调用就会失败。

断连操作使得持久异步调用的改进型异步调用模型的实际意义更大。就所提供的操作而言，这一模型与Mercury相似，它们的错误语义不同。一个传统的调用机制（同步或异步）被设计成在超过给定的超时时间后调用失败。但是这些短期的超时经常不适应连接中断或长延迟发生。

持久异步调用系统试图无限地执行调用，直到它知道调用成功或失败，或者直到应用程序取消调用。其中一个例子是用于移动信息访问的Rover 工具包[Joseph *et al.* 1997]中的QRPC（排队的RPC）。 240 ↓ 241

正如其名字所描述的，当没有网络连接时，QRPC将调用请求在固定的日志中排队，当网络连接建立时，它调度并发送请求给服务器。类似地，它将服务器的返回结果排队并放置在我们所认为的客户调用“邮箱”中，直到客户重新连接服务器并收集结果。请求和返回结果在排队时可能被压缩，这样它们可以在低带宽的网络上传输。

QRPC可以利用不同的通信连接发送调用请求和接收应答。例如，当用户在移动过程中时，其调用请求可以在GSM连接上发送，但当用户将他的设备连接在企业内部网上时，其调用应答可能通过以太网连接发送。原则上，调用系统可以在靠近用户的下一个可能的接入点位置存储调用结果。

客户网络调度器可以根据不同的规范操作，而没有必要一定依照FIFO的顺序发送请求。应用程序可以为单个调用赋予优先级。当有可利用的连接时，QRPC会估计其带宽以及使用这个连接的开销。它首先发送高优先级的调用，但在连接速度很慢并且开销大（例如广域无线连接）的情况下，它不会发送所有的调用，因为它假设在不久的将来有像以太网这样更快更廉价的网络连接可被利用。类似地，当QRPC从低带宽连接的邮箱中获取调用结果时，它也会考虑优先级。

在异步调用系统（持久或其他）的编程中有如下问题：当调用结果未知的情况下，用户如何在客户设备上继续使用其应用程序。例如，用户可能想知道是成功地更新了共享文档的一个段落，还是另一个用户同时进行了一次有冲突的更新，例如删除了这一段落？第14章将讨论这一问题。

6.6 操作系统体系结构

本节讨论适用于分布式系统的内核体系结构。我们采用第一原则方法，首先从开放性的需求出发讨论已有的主要的内核体系结构。

一个开放的分布式系统应该达到以下方面的要求：

- 仅在每台计算机上运行那些为在系统体系结构中承担特定作用的系统软件，不同的计算机对系统软件的需求可能会不同，例如个人数字助理和专职服务器对系统软件的需求就不同。载入多余的模块会浪费内存资源。
- 允许实现特定服务功能的软件（和计算机）能独立于其他部分而被更换。
- 当需要适应不同用户或应用时，允许提供同一服务的其他实现。
- 在不破坏已存在系统的一致性的情况下加入新的服务。 242

从资源管理策略中分离固定资源管理机制这一方法随着应用程序和服务的不同而不同，它已在一个很长的时间内成为操作系统设计的指导原则[Wulf *et al.* 1974]。例如，我们说一个理想的调度系统应提供如下机制：系统既能满足一个像视频会议这样多媒体应用程序的实时

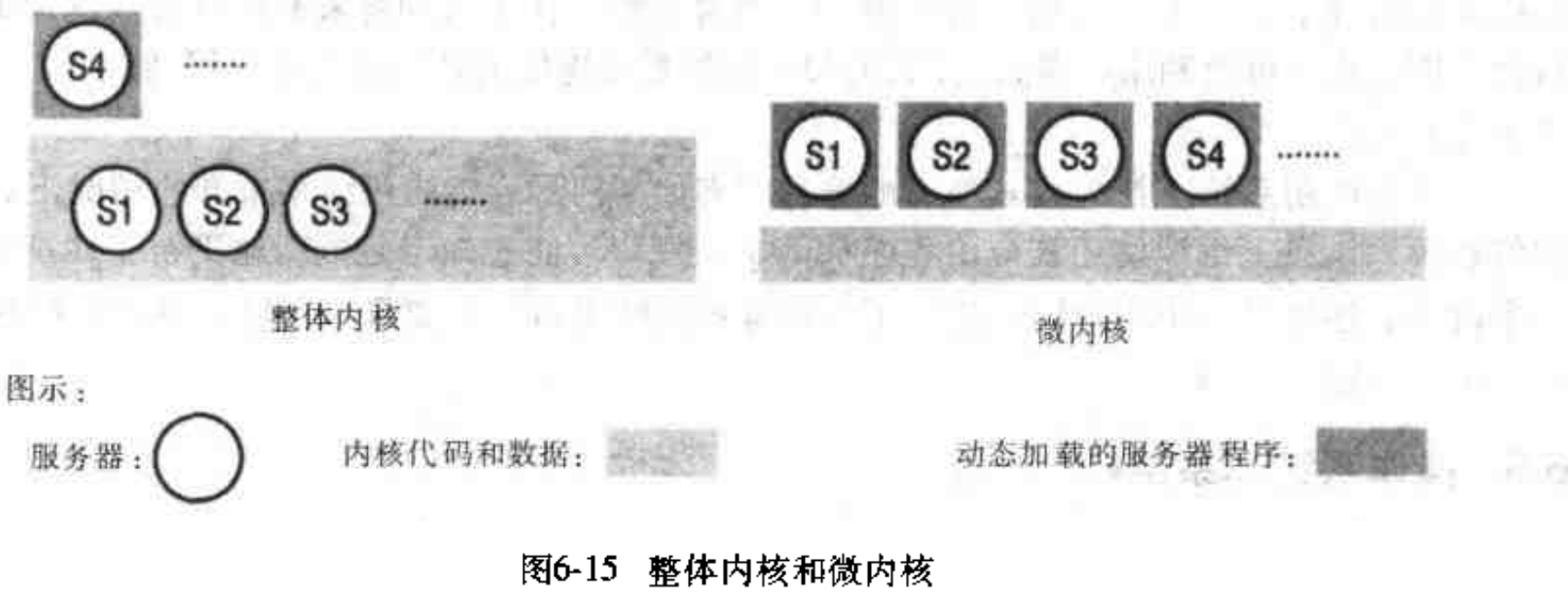
需求，也能满足一个像网页浏览这样的非实时应用程序的要求。

按理想的想法，内核应该只提供在一个结点上实现通用资源管理任务的最基本机制。应按需动态装载服务器模块，以便为当前运行的应用实现所需的资源管理策略。

整体内核和微内核 内核设计有两个主要例子：即整体内核和微内核方法。这两个设计间的主要区别在于：如何决策哪些功能属于内核、哪些功能属于服务器进程以便在运行时动态载入这些功能。尽管微内核没有被广泛应用，但理解它们与当今一般内核相比的优点和缺点仍然是有益的。

UNIX操作系统内核被称作为整体内核（见下面文本框部分的定义）。这一名称说明这种内核的巨大：它完成所有的基本操作系统功能，其代码和数据量达到上兆字节。这种内核是未分化的：它以非模块方式编码。这在很大程度上导致它是难于管理的：为变化的需求改变单个软件模块将会很困难。Sprite网络操作系统[Ousterhout *et al.* 1988] 是另一个整体内核的例子。一个整体内核可以容纳在其地址空间内执行的若干服务器进程，其中包括文件服务器和一些网络进程。这些进程执行的代码是标准内核配置的一部分（如图6-15所示）。

> **整体（monolith）** 《钱伯斯20世纪字典》为*monolith*和*monolithic*给出了如下定义：
>
> **monolith**，名词，由一块石头构成的柱子或圆柱：任何像整体的事物都是一致的、大块的或难管理的。形容词**monolithic**属于或像一个整体：一个国家或一个组织机构等，大块的，并且全体一致，因此难于管理。



243 图6-15 整体内核和微内核

相反，在微内核的设计中，内核只提供最基本的抽象，主要是地址空间、线程和本地进程间通信；所有其他系统服务都由服务器提供，这些服务在分布式系统需要它们的时候才动态加载到计算机上（如图6-15所示）。客户使用内核基于消息的调用机制来访问这些服务。

上面我们说过用户不会接受不能运行它们应用程序的操作系统。但除了扩展性之外，微内核设计者还有其他目标：二进制模拟像UNIX这样的标准操作系统[Armand *et al.* 1989, Golub *et al.* 1990, Härtig *et al.* 1997]。

图6-16给出了最通用形式的微内核在整个分布式系统中的位置。其中，内核表现为在硬件层与包含主要系统模块被称为子系统层之间的一层。如果主要设计目标是性能而不是可移植性，那么中间件可以直接使用微内核的设施。否则，它使用语言支持于系统或由操作系统模拟子系统提供的高层操作系统接口。这些都是由可连接在应用程序上的过程库和运行在微内核上的服务器实现的。

微内核通过子系统支持中间件

图6-16 微内核的作用

可以在同一底层平台之上给程序员提供多个系统调用接口（多种操作系统）。这种情形是IBM 370体系结构的复活，其VM操作系统能够为运行在同一（单处理器）计算机上的不同的程序提供不同的虚拟机。这种情形在分布式系统中的一个例子是：在Mach分布式操作系统内核上实现UNIX和OS/2系统。

比较 基于微内核的操作系统的主要优点是它的可伸缩性和其在内存保护边界的基础上增强模块化的能力。另外，一个相对较小的内核的缺陷数量可能比大而复杂的内核要少。 244

整体内核设计在调用操作方面效率相对高一些。但系统调用可能比常规的过程操作开销大，甚至在使用了上一节介绍的技术后也是如此。在同一结点上的一个用户级地址空间上的调用开销仍然比较大。

通过使用像分层（在MULTICS[Organick 1972]中使用）或像在Choices[Campbell *et al.* 1993]中使用的面向对象设计这样的软件工程技术可以避免整体内核设计中的无结构性。Windows NT采用了以上两种方法的组合[Custer 1998]，但是Windows NT仍然是“巨大”的，并且大多数功能没有被设计为可替换的。模块化的大内核也难于维护，同时它只为开放的分布式系统提供有限的支持。只要模块在同一地址空间内执行，并且用C或C++语言编写并编译成高效代码，而且允许随意的数据访问，就可能破坏严格的模块性，因为程序员可能试图使用一种更高效的实现方法，这样一个模块中的缺陷可能会破坏另一个模块的数据。

一些混合的方法 两种原始的微内核Mach[Acetta *et al.* 1986]和Chorus[Rozier *et al.* 1990]在其开发的过程中只使用用户进程作为运行服务器。它们通过硬件支持的地址空间增加模块性。在服务器进程需要直接访问硬件时，系统为这些特权进程提供了特殊的系统调用，用于将设备寄存器和缓冲区映射到它们的地址空间内。内核将中断转换为消息，这样用户级服务器就可以处理中断。

由于性能问题，Chorus和Mach微内核设计最终允许将服务器进程动态地加载到内核地址空间内或用户级地址空间内。在这两种情况中，客户进程可以用相同的进程间通信调用与服务器进程交互。因此开发者可以在用户级调试服务器进程，同时，在开发完成时，为了优化系统性能，系统允许服务器进程在内核地址空间内运行。如果服务器进程包含某种缺陷，它就会影响系统的一致性。

SPIN操作系统[Bershad *et al.* 1995]设计采用语言保护机制权衡效率和安全性。其中，内核和所有动态载入到内核的模块在一个地址空间内执行，但是它们都是由类型安全的语言（Modula-3）编写的，所以它们相互间是受保护的。内核地址空间内的保护域是使用受保护命

名空间建立的。除非具有访问引用，否则进入内核的模块是不能访问资源的，同时Modula-3限制了引用只能用于执行程序员允许的操作。

为了减少系统模块之间的依赖，SPIN系统的设计者选择了一个基于事件的模型作为进入内核地址空间模块的交互机制（见5.4节对基于事件的程序并发的讨论）。系统定义了一系列核心事件，例如网络包到达、定时器中断、发生页失配和线程状态改变。系统组件将自己注册为影响它们的事件的处理程序。例如，一个调度程序可以将自己注册为一个处理程序，用
245 于处理与6.4节讨论的调度器激活相似的事件。

诸如Nemesis[Leslie *et al.* 1996]这样的操作系统发现了这样一个事实：即使在硬件级，一个地址空间也不必是单一的保护域。内核和所有动态加载的系统模块以及所有的应用程序都可以共存在单一地址空间内。当地址空间载入应用程序时，内核将应用程序的代码和数据放置在当前可用的空间内。64位寻址处理器的出现使单地址空间的操作系统变得更加诱人，这是因为它们支持很大的地址空间，可以容纳许多应用程序。

单地址空间操作系统的内核在其地址空间内的独立区域上设置保护来限制用户级代码的访问。用户级代码在处理器的特定的保护上下文中运行（由处理器和内存管理单元中的设置决定），它给了代码访问其自身区域的完全权限和特定的共享其他区域的权限。相对于多地址空间设计，单地址空间设计的开销节省在：当域转换时，内核不需要复制并清除任何缓存。

最近一些内核设计，例如L4[Härtig *et al.* 1997]和Exokernel[Kaashoek *et al.* 1997]采用了我们所描述的“微内核”方法，但也包含许多与此机制相反的策略。L4是“第二代”微内核设计，它要求动态加载的模块在用户级地址空间内执行，但它优化了进程间通信以减少上述策略带来的开销。通过将地址空间的管理授权给用户级服务器，它减少了内核的复杂性。Exokernel系统采用了一种完全不同的方法，它采用用户级库代替用户级服务器来提供功能扩展，这为磁盘块等低级资源提供了保护性分配，并且它希望所有其他资源管理功能（甚至是文件系统）都用库的方式连接到应用程序上。

用一个微内核设计者[Liedtke 1996]的话说：“微内核的发展过程充满了困难绝境，也处处体现出奇思妙想”。至今仍然没有一个全面评估，用于评定如何设计一个具有充分扩展性并且相对于整体设计具有较好性能的操作系统体系结构。

6.7 小结

本章介绍了操作系统通过提供对共享资源的调用来支持中间件层。操作系统提供了若干机制，用于实现满足本地需要的多种资源管理策略。它允许服务器封装和保护资源，同时允许客户并发地共享资源。它提供了客户调用资源操作的必要机制。

进程由执行环境和线程组成。执行环境包括地址空间、通信接口和其他像信号量这样的本地资源；线程是执行环境中的活动抽象。地址空间必须较大且彼此间稀疏，这样可以支持共享访问和映射访问像文件这样的对象。新的地址空间可能由继承其父进程的区域内容来创
246 建。写时复制是重要的区域拷贝技术。

进程可以拥有多个线程，这些线程共享进程的执行环境。多线程进程可以利用多处理器的优势使并发操作的开销相对较少，这对客户和服务器都有益。当前有些线程实现允许两层调度：用户级代码处理调度策略的细节，而内核提供对多处理器的访问。

操作系统为通过共享内存进行的通信提供了基本的消息传递原语和机制。大多数内核将

网络通信作为一个基本的设施包含在其内部，其他内核只提供本地通信并将网络通信功能交给服务器程序，这样可以实现一系列的通信协议。这是在性能和灵活性之间的一种折中。

我们讨论了远程调用并且说明了直接来源于网络硬件的开销和来源于操作系统代码执行的开销之间的区别。对于一个空调用而言，花费在软件上的时间相对较大，但当调用参数的数据量增大时其时间占总时间的比例会减小。调用中可以被优化的主要开销来源于编码、数据拷贝、包初始化、线程调度和上下文切换以及流控制协议的应用。在同一计算机内地址空间之间的调用是一个重要的特殊例子，我们描述了在轻量级RPC中使用的线程管理和参数传递技术。

实现内核体系结构有两种主要方法：整体内核和微内核。它们之间的主要区别在于是由内核管理资源还是由动态载入（通常是用户级）的服务器来管理资源。微内核至少必须支持进程和进程间通信。它支持操作系统模拟子系统、语言支持子系统和其他子系统（如实时处理子系统）。

练习

6.1 在UNIX文件服务的例子中讨论封装、并发操作、保护、名字解析、参数和返回结果的通信以及调度等操作的任务。

6.2 为什么一些系统接口由专门的系统调用（对内核）实现，而其他一些系统接口由基于消息的系统调用实现？

6.3 史密斯认为进程中的每个线程都拥有自己的保护栈，而进程的其他区域应该被完全共享。这样做有意义吗？

6.4 信号（软件中断）处理器应属于进程还是线程？

6.5 讨论共享内存区域的命名问题。

6.6 假设要设计一个平衡各计算机负载的方案，必须考虑如下问题。 247

（i）这一方案能满足用户或系统的哪些需求。

（ii）它能适应哪种类型的应用程序。

（iii）如何度量负载以及在何种精确程度上度量负载。

（iv）假设进程不能迁移，怎样监控负载并为新的进程选择位置。

如果能在计算机之间迁移进程，你的设计将受到哪些影响？你认为进程迁移的开销很大吗？

6.7 解释在UNIX中区域拷贝用写时复制的好处，其中在一个*exec*调用后通常是一个*fork*调用。在使用写时复制的区域是自我复制的情况下会发生什么？

6.8 一个文件服务器使用缓存，其命中率为80%。当服务器的缓存中有被请求的块时，服务器的文件操作要花费5ms的CPU时间，否则它还要花另外的15ms用于磁盘I/O。对于下面假设的各种情况，估计服务器的吞吐量（平均请求次数/s）。

（i）单线程。

（ii）在一个处理器上运行的两个线程。

（iii）在双处理器计算机上运行的两个线程。

6.9 比较工作池多线程体系结构和一请求一线程体系结构。

6.10 什么样的线程操作开销最显著？

6.11 spin锁（见Bacon[1998]）是一个通过测试和设置（test-and-set）原子指令访问的布尔变量，用于实现互斥。你能使用spin锁在单进程的计算机上实现线程间的互斥吗？

6.12 说明内核应为用户级线程的实现提供哪些支持，例如在UNIX中的Java。

6.13 页失配是用户级线程实现中的问题吗？

6.14 说明有哪些因素使得在“调度器激活”设计中选择使用混合调度方法（而不是纯粹的用户级或内核级调度）。

6.15 为什么我们必须注意线程包的线程阻塞或解除阻塞事件？为什么当虚拟处理器即将被抢占时，我们也要注意它（提示：可以继续分配其他虚拟处理器）？

6.16 网络传输时间占一个空RPC总耗时的20%，而传输1KB数据（小于一个网络包的大小）时，网络传输时间占RPC总耗时的80%。如果网络由原来的10Mbps升级到100Mbps，这两次操作的网络传输时间将改进多少？

6.17 一个“空”的RMI不包含参数，它调用一个空过程并不返回结果，其延迟为2ms。
248 请解释导致延迟的原因。

在同一个RMI系统中，每1KB的用户数据会增加额外的1.5ms延迟。一个客户希望从文件服务器获取32KB的数据，它应该使用一个32KB的RMI还是应该使用32个1KB的RMI？

6.18 影响远程调用的哪些因素会影响消息传递？

6.19 解释共享区域是如何应用于进程读取内核写的数据的。解释应包括实现同步的必要操作。

6.20 （i）轻量级过程调用的服务器能控制其中的并发度吗？

（ii）请解释在轻量级RPC中为什么客户不允许调用服务器内的任何代码，并说明如何实现。

（iii）LRPC是不是比传统的RPC（假设是共享内存的）承担更多的交互干扰危险？

6.21 一个客户对一个服务器进行RMI调用。客户需要对每一个请求进行5ms的参数计算，服务器要花费10ms处理每一个请求。每一个*send*和*receive*操作的本地OS处理时间是0.5ms，同时传输每一个请求或应答消息的时间是3ms。每个消息的编码或解码时间是0.5ms。

在如下情况下，估计客户产生两个请求并返回结果的时间：（i）单线程；（ii）在单处理器上有两个线程，它们并发地发出请求。

如果进程是多线程的，系统需要使用异步RMI吗？

6.22 解释什么是安全性策略，在诸如UNIX这样的多用户操作系统中，与之相对应的是什么机制？

6.23 解释当服务器进程动态载入内核地址空间内时，程序必须满足的连接要求，并说明这种情形与在用户级执行服务器进程的区别。

6.24 中断是怎样与用户级服务器通信的？

6.25 在某个计算机上，我们预计：不管运行哪种OS，线程调度花费50μs，一个空过程调用花费1ms，上下文切换到内核花费20μs，一个域转换花费40μs。在使用Mach和SPIN操作
249 系统这两种情况下，估计客户调用动态载入的空过程的开销。

第7章 安 全 性

7.1 简介
7.2 安全技术概述
7.3 加密算法
7.4 数字签名
7.5 密码实用学
7.6 实例研究：Needham-Schroeder、Kerberos、SSL 和 Millicent
7.7 小结

在分布式系统中，对资源的私密性、完整性以及可用性都需要采取相应的手段加以保证。安全性攻击的形式多种多样，有窃听、伪装、篡改和拒绝服务等。可靠的分布式系统的设计者们必须解决暴露的服务接口和不安全网络的问题，而攻击者可能了解其中所使用的算法并部署计算资源。

密码学为消息的私密性和完整性以及认证消息提供了理论基础，而要使密码学付诸应用还需要精心设计的安全性协议。加密算法的选择和密钥的管理是效率、性能和安全机制可用性的关键。公开密钥加密算法使得密钥分派比较容易，但其性能不适于对大批量数据的加密。相比之下，保密密钥加密算法更适合大量数据加密任务。混合型协议，例如**SSL**（安全套接字层）用公开密钥加密算法先建立一个安全通道，然后使用通道交换保密密钥，并将此保密密钥用于后继的数据交换。

数字化信息可以用于签名，这就是数字证书。这个证书在用户和组织间建立起相互间的信任。

7.1 简介

一旦计算机系统遇到恶意的或是恶作剧性质的攻击，就必须将安全性手段结合到计算机系统中去。对于处理全融交易、高度机密的信息或者其他对私密性和完整性要求严格的信息系统来说，更应如此。图7-1总结了自20世纪60年代和70年代以来，多用户分时系统中因共享数据的出现而产生的安全性需求变化。而最近出现的广域开放的分布式系统提出了更多的安全问题。

保护信息和资源的私密性和完整性的需求在数字世界和物理世界中都是广泛存在的，无论该资源是属于个人还是属于组织。它起源于资源的共享。在物理世界中，公司采用安全策略，在指定范围内保证资源的共享。例如，某公司只允许公司的职员以及可以信赖的访问者进人大楼。而文档的安全策略可以规定某些组的成员只能访问某些类的文件，这样的规定也可以针对单个文件和使用者。

安全策略是通过安全机制执行的。例如，进人大楼由接待员控制，他给可以信赖的访问者发放徽章，再由门卫或者电子门锁检验。对于纸质文档的访问，通常采用保密和限制性发送来控制。

在电子应用中，安全策略和安全机制的区别仍然非常重要，没有它，就很难判别一个系统是否安全。安全策略和所使用的技术是相对独立的，就像在门上装锁并不能确保大楼的安全，除非对于它的使用制定一些策略（例如，在没有人守门时，门就会被锁上）。我们所描述的安全机制本身并不能保证系统的安全。在7.1.2节中，我们将概述各种简单的电子商务场景中的安全需求，并说明在那个环境中的策略需求。作为初始例子，考虑一个联网的文件服务器的安全性，它的接口对于客户是可访问的。为了维护文件的访问控制，就需要有一个相应的策略，即所有的请求必须包含一个已认证的用户身份。

本章的重点是提供对数据、计算机资源以及联网事务的保护机制。我们将讨论这些机制，它们使得安全策略可在分布式系统中得以执行。这些机制也足以对付大部分确定性的攻击。

	1965~1975年	1975~1989年	1990~1999年	现在
平台	多用户分时计算机	基于本地网络的分布式系统	因特网、广域服务	因特网+移动设备
共享资源	内存、文件	本地服务（如NFS）、本地网络	电子邮件、Web站点、因特网商业	分布式对象、移动代码
安全需求	用户识别和身份认证	服务保护	商业事务的强安全性	对单个对象的访问控制、安全的移动代码
安全管理环境	单个的授权、单个的授权数据库（如/etc/passwd）	单个的授权、委托、复制的授权数据库（如NIS）	多个授权、没有网络范围的授权	对每次行动的授权、共享责任的组

图7-1 安全需求演变过程

了解安全策略和安全机制间的区别，对于设计安全系统是有帮助的，但我们又常常不能确定一套给定的安全机制是否可以完全实现所需的安全策略。在2.3.3节中，我们介绍了一个安全模型，它可以用来分析分布式系统中潜在的安全隐患。我们将第2章的安全模型总结如下：

- 进程封装了资源（如程序语言层的对象和其他系统定义的资源），而且允许客户通过它们的接口访问这些资源。可以显式地授予主体（用户或者其他进程）操作资源的权限。资源被保护以防止未授权的访问。
- 进程通过多用户共享的网络进行交互。敌人（攻击者）也可以访问网络。他们能拷贝或试图读取所有通过该网络传输的消息，也可以向网络中插入任意的消息，指向任何目的地址，而声称是来源于某一位置。

安全模型识别出分布式系统中可能使系统受到攻击的漏洞。本章将详细说明这些攻击和应对攻击的安全技术。

密码学进入公众领域 密码学为大多数计算机安全机制提供了基础。密码学有一段悠久而有趣的历史。军方对安全通信的需要以及相应的截获和解密敌人信息的要求使得当时一些杰出的数学人才为之倾注了大量的精力。读者如果对这段历史感兴趣的话，可以参阅由David Kahn[1967,1983,1991]和Simon Singh[1999]写的书。Whitfield Diffie，公开密钥加密算法发明人之一，以第一手信息记录了近几年密码学的历史和政治[Diffie 1988, Diffie and Landau 1998]，并写入了Schneier一书[1996]的序言。

以前由于政治军事组织控制密码学的发展和使用，直到最近密码学才真正被解放出来。现在，它成为了一个十分活跃的开放性研究课题。研究结果也出版在许多书籍、杂志和会议论文集中。Schneier应用密码学[1996]的出版是该领域知识展开的一个里程碑。这是第一本包括了很多重要算法且带源码的书，这也是勇敢的一步。因为当第1版1994年面世的时候，这样的出版物是否合法也还没有定论。在现代密码学的大部分领域，Schneier具有了相当的权威性。Menezes等[1997]也出版了一本有很强理论基础的实用性手册。 252 ι 253

密码学进一步开放是非军事应用和分布式计算机系统安全需求巨大发展的结果。这也导致出现了第一批在军事领域以外、自主的密码学研究群体。

具有讽刺意味的是，密码学对公众的开放和应用使密码技术得到了突飞猛进的发展。这些发展不仅表现在对抗敌人的攻击能力方面，也表现在密码技术部署的方便性上。公开密钥算法就是在开放以后收获的成果之一。另外，DES标准加密算法起先还是一个军事秘密，它最终的公布和人们成功的破解，反而促进了密码学的发展，创造出更多更强有力的保密密钥加密算法。

另外一个有益的副产品是公共术语学和方法的发展。作为后者的一个例子是，为一个要受保护的事务中的角色（主体）选取一组习惯用名。为主体和攻击者都起一个习惯用名，有利于阐明和看清安全协议和潜在攻击的描述，是识别其弱点的重要一步。图7-2中显示的名字在安全文献中广为使用，我们在此也将沿袭使用这些人名。我们不知道这些名字的确切由来，但据我们所知，最早它们出现于最初的RSA公开密钥算法论文[Rivest *et al.* 1978]中，而且关于它们的使用有一个有趣注释，请参阅Gordon[1984]。

Alice	第一参加者
Bob	第二参加者
Carol	三方或四方协议的参加者
Dave	四方协议的参加者
Eve	窃听者
Mallory	恶意的攻击者
Sara	服务器

图7-2 为安全协议中的角色起的名字

7.1.1 威胁和攻击

有一些威胁是显然的——例如，在大多数类型的本地网络中，很容易就可以构造并运行一个程序，以获得其他计算机间传递的消息的副本。其他威胁就更加狡猾——当客户不能通过服务器认证时，该程序就安装其自身，并取代真实的文件服务器，从而拿到客户发过来机密信息。

除了直接破坏而导致丢失或者损坏信息和资源，攻击者还可能针对系统拥有者做出系统是不安全的欺骗性声明。为了避免这些声明，拥有者必须证明系统在受到攻击后仍然是安全的，或者为这个产生疑问时期中的所有事务生成一个日志文件以反驳这些声明。一个常见的例子就是自动取款机上的“假象提款”问题。最好的解决方法就是银行提供一个由账户持有者进行了数字签名的事务记录，使得第三方不能伪造。 254

安全旨在将对信息和资源的访问限制到已被授权的主体中。信息威胁一般可分为三大类：

- 泄露　未经授权的接收方获得了信息。
- 篡改　未经授权就对信息进行改动。
- 恶意破坏　干扰系统的正确操作，对破坏者本身无益。

对分布式系统的攻击是通过获得对现有信道的访问或者建立伪装成授权连接的新通道的方式进行的（我们这里用术语通道来代指任何进程间的通信机制）。还可以按照通道被误用的方式，进一步对攻击的方法进行分类：

- 窃听　获取未经授权的消息副本。
- 伪装　未经授权而使用其他主体的身份收发消息。
- 消息篡改　在消息传往接收者之前，截获并修改其中的内容。中间人攻击这种消息篡改方式是指攻击者截获了密钥交换的第一个消息。攻击者替换掉他们达成的密钥，使得自己可以对他们的后继消息进行解密，并再将消息按正确的密钥加密后，传递出去。
- 重发　存储截获的信息，并在以后发送它们。这种攻击甚至对已认证的消息和加密消息可能都有效。
- 拒绝服务　用大量的消息淹没通道或者其他资源，使得其他访问被拒绝。

这些是理论上的危险，但实际上这些攻击是怎样实现的呢？成功的攻击取决于是否发现安全系统中的漏洞。遗憾的是，这些威胁在现在的系统中都过于普通。Cheswick和Bellovin[1994]指出了42种弱点，他们认为这些在广泛使用的因特网系统及其组件中具有很大的风险。这些弱点从口令猜测到对完成网络时间协议或处理邮件传输的程序进行攻击。其中有些已经成为成功的并广为宣扬的攻击的入口点[Stoll 1989，Spafford 1989]，还有许多已被用于恶作剧或者用于犯罪。

设计因特网和与之相连的系统时，安全性并不是被优先考虑的。设计者或许没有想到因特网会发展成如此规模，而且，诸如UNIX之类的系统的基本设计也先于计算机网络出现。我
255 们可以看到的，安全手段需要在基本设计阶段就仔细进行考虑。本章内容就是为此思想提供基础。

我们已经讨论过由于暴露信道及其接口而产生的种种对分布式系统的威胁。对许多系统而言，这些是惟一需考虑的威胁（人为错误引起的威胁不在考虑之列——安全机制并不能防卫用户使用非常简单好猜的口令或者用户粗心泄露口令的情况）。但对于包含移动程序的系统和对信息泄露特别敏感的系统，还有其他威胁。

对移动代码的威胁　最近开发的一些程序设计语言允许程序从远程服务器中下载到一个进程，并在本地执行。这样的话，执行进程中的内部接口和对象都暴露在移动代码的攻击范围内了。

Java是这种类型中最广为使用的语言。为了限制这种暴露，设计者也深思熟虑过语言的设计和构造，以及远程下载机制（沙盒模型即用于对付移动代码）。

Java虚拟机（JVM）在设计的时候就考虑了移动代码。它为每个程序提供其自身运行的环境。每个环境都有一个安全管理器，用于确定哪些资源对于该程序是可用的。例如，安全管理器会停止程序读写文件或对其网络访问加以限制。一旦设置了安全管理器，它就不能被替换。当用户运行一个程序如浏览器下载移动代码用于本地运行时，他确实没有更好的理由来相信这些移动代码会可靠地执行。实际上，下载并运行恶意代码删除文件或访问私人信息的危险性是存在的。为了使用户免受这些不可信代码的损坏，大部分浏览器都限定了Java小

程序不能访问本地文件、打印机和网络套接字。一些移动代码的程序能在下载的代码中采用多种信任级别。这样，安全管理器通过配置就可提供更多地对本地资源的访问。

为保护本地环境，JVM还提供了下面两个手段：

1. 下载的类和本地的类分开保存，防止它们用假冒的版本来替换本地的类。
2. 检验字节码以验证其有效性。有效的Java字节码由一组来自指定集合的Java虚拟机指令组成。这些指令也会被检验，以保证它执行的时候不会发生某些错误，例如访问非法的内存地址。

在最初采用的机制不能避免漏洞[McGraw and Felden 1999]这一问题逐渐被认识的过程中，Java的安全性成为了许多后继研究的主题。被识别的漏洞得到了修补，Java保护系统也只有在有授权时，才允许移动代码访问本地资源[java.sun.com V]。

除了包括类型检查和代码有效性机制，合并到移动代码系统的安全机制仍然达不到像保护信道和接口所能达到的信任级别。这是因为执行程序的环境为错误提供了很多的机会，而且并不能肯定避免所有的错误。Volpano和Smith[1999]已经指出另外一个方法，它基于证明移动代码的行为是完备的，这或许是一个较好的解决办法。 256

信息泄露 如果可以观测到两个进程间的消息传递，那么就可以收集到一些信息——例如，大单的某种股票的消息会显示这支股票有一个比较高的交易率。还有许多微妙的信息泄露形式。有些是恶意的，而有些是源于疏忽。一旦观测到计算的结果，其潜在的泄露危险就增大了。在20世纪70年代，人们就开始进行防止这类安全隐患的工作[Denning and Denning 1977]。所采取的方法是为信息和通道赋予安全等级，并分析进入通道中的信息流，以保证高层信息不会流入低层通道。Bell和LaPadula[1975]第一次描述了信息流的安全控制方法。最近一些研究的主题[Myers and Liskov 1997]是用组件之间的互不信任将此方法扩展到分布式系统。

7.1.2 保护电子事务

许多使用因特网的工业、商业和其他地方都包括一些对安全性要求很高的事务。例如：

- 电子邮件 虽然电子邮件系统原本不包括安全性，但许多用户的信件内容都必须保持机密（例如，当发送一个信用卡号时），或者内容和消息的发出者必须经过认证（例如，用电子邮件提交一个拍卖的竞价）。基于本章所述技术的密码安全性目前已经应用到了许多邮件客户中了。
- 购物和服务 这样的事务现在已经随处可见。购买者在Web上选定商品并为之付账，所购的商品就会通过适合的配送机制送到他们的手中。软件或者其他的数字产品（例如唱片和录像）可以通过从因特网上下载来完成传递。其他有形的商品，例如图书、CD和其他几乎所有种类的商品也由因特网提供商售卖，然后通过配送服务传递给购买者。
- 银行事务 电子银行为用户提供了常规银行所能提供的所有的服务。它们可以检查余额状态、转账、定期缴纳各种款项等等。
- 微事务 因特网参与提供少量信息和其他服务，而面对的是众多的用户。例如，大部分的Web页面还没有收费，但要将Web开发成为一个高质量的发布媒介，信息提供商肯定需要信息的消费者为他们所使用的信息付费。因特网上音频和视频会议的使用正是提供了这样的例子，也就是只有最终用户付费之后，才可以提供相应的服务。这些服务的价格或许不到一分钱，这些支付开销必须相对地低。通常来说，为每件包括银行或信用卡 257

服务器事务的方案还不能达到这点。

这样的事务想要安全执行，必须有合适的安全策略和安全机制的保护。需要保护购物者以防止在传输中泄露他们的信用卡代码（卡号），以及防止那些欺骗性的供货商在收到付款后不准备发货。供货商必须在发货前得到付款，对于下载的产品，他们还必须确认只有顾客得到了可用的数据。用于所需保护的开销，与事务的价值相比必须是合理的。

为因特网供货商和购买者制订的安全策略导致了以下Web交易的安全需求：

1. 为购买者认证供货商，这样购买者就可以确信他们是在和准备进行交易的供货商的服务器联系。
2. 不能让购买者的信用卡号和其他支付信息落入第三方手中，同时保证这些资料不加改变地在购买者和供货商之间传输。
3. 如果商品的形式适合下载，那么保证它们的内容不加改变地传给了购买者，同时避免泄露给第三方。

通常供货商并不对购买者加以认证（除非是传递不可下载的商品）。供货商会希望能检测购买者的付款能力，但这通常是在发送商品前，向购买者的银行要求支付款项时完成的。

把银行账户持有者比作购买者，银行比作供货商，那么使用开放网络的银行事务的安全需要类似购买事务，但显然还有下列需要：

4. 在给予银行的账户持有者访问账户的权限之前，要对其身份加以认证。

注意在这种情况下，银行必须保证账户持有者不能抵赖他们参加了事务，即不可抵赖。

除了上述由安全策略规定的需求，还有一些系统需求，它们与因特网巨大的规模有关，因为它使得购买者和供货商在实践中难以形成某种关系（通过注册密钥，以供以后使用）。购买者应该可以在没有第三方或以前与供货商没有联系过的情况下完成一个安全的事务。一些技术，例如使用“cookies”——用于记录以前的交易，并储存在客户主机上——有明显的安全缺陷，而且台式和移动主机都经常处于不安全的物理环境中。

258 由于因特网商业安全的重要性以及其飞速的发展，我们将讲述一些密码安全性技术的应用，如7.6节将描述实际上的标准安全协议，也是在大部分电子商务中使用的——安全套接字层（SSL）——和Millicent，一个专为微事务所设计的协议。

因特网商业是安全技术的一个很重要的应用，但并不是惟一的一个，任何个人或是组织在存储和交流重要信息时都会用到它。个人间使用加密的电子邮件进行交流已经成为大家关心的话题，我们将在7.5.2节中就此展开讨论。

7.1.3 设计安全系统

近年来，密码技术及其应用都得到了极大的发展，但安全系统的设计依然是一项棘手的工作。陷入这种困境的主要原因是设计者们总是想能排除掉所有可能的攻击和漏洞。这就像程序员的目标就是去掉程序中所有的错误。无论哪种情况都没有具体的方法能保证在设计阶段达到此目的。按已知的最好的标准去设计，再进行非正式的分析和检测。设计完成后可以选择是否进行形式化的验证。对安全协议进行形式化验证的工作已产生了许多重要的结果[Lampson *et al.* 1992, Schneider 1996, Abadi and Gordon 1999]。可以在www.cdk3.net/security网址找到介绍这个方向迈出的第一步——BAN认证逻辑[Burrows *et al.* 1990]及其应用。

安全就是有关避免大灾难和最小化一般的灾难。进行安全性设计时，必须考虑到最坏的情况。下面的阴影部分给出了一些有用的假设和设计指南，这些假设成为本章讨论的技术思想的基础。

为了说明一个系统中使用的安全机制的有效性，系统设计者必须首先列出一张隐患单子——会破坏安全策略的方法——并给出防止相应隐患的机制。这种说明可以采取非正式讨论的形式，或更好的，采用逻辑证明的形式。

没有哪张隐患清单会穷尽所有的问题，因此在安全敏感的程序中还必须使用审计的方法，以查出违规操作。如果安全敏感系统中的安全日志文件总是详细记录用户的操作和他们的授权信息，那么审计是很容易实现的。

个安全日志会对用户的操作打上时间戳并按序记下。日志中的记录至少要包括主体的身份、所完成的操作（如删文件、更新账户记录）、被操作对象的标识和一个时间戳。在怀疑有违规操作的地方，记录还会包含使用物理资源（网络带宽和外围设备）的使用，或者进行日志记录的进程会记录一些对特殊对象的操作。后继的分析可以是基于统计或是基于搜索。即使没有可疑之处，随着时间的流逝，这些统计数字也会帮助发现任何不同寻常的趋势或事件。

安全系统的设计是平衡开销与隐患的实践。用来保护进程和为进程间通信提供安全保护所采用的技术范围相当的广阔且强大，足以对付几乎任何攻击，但它们的应用也招致了一些开销和不便： 259

- 在使用安全系统时，产生了额外的开销（用于计算和网络的应用），这种开销必须和隐患相平衡。
- 指定了不合适的安全手段，会使合法的用户也无法执行必要操作。

不与安全折中，很难达成这样的平衡，这似乎与本小节第一段中的建议相冲突。但安全技术的力量可以根据预估的攻击开销来量化和选择。例如，在为小型商业事务服务的Millicent协议中采用了相对低开销的技术，我们将在7.6.4节加以讨论。

最坏假定和设计原则

- *接口是暴露的* 分布式系统由提供服务或共享信息的进程组成，它们的通信接口必须是开放的（为了让新的客户访问它们）——攻击者可以给任一接口发送消息。
- *网络是不安全的* 例如，消息源是伪造的——消息看似来自Alice，而其实是来自Mallory。主机地址可能是“哄骗”的——Mallory用Alice的地址连上网络，并接收发给Alice的消息。
- *限制每个秘密的范围和生存时间* 当密钥第一次产生的时候，我们应该对这个密钥不会被损害有信心。我们用的时间越长，知道它的范围也就越广，危险系数也就越大。使用秘密如口令或者共享的密钥应该是有时间限制的，而且这种共享也应该是有限制的。
- *算法和程序代码可能被攻击者得到* 一个秘密的分布越大越广，它被泄露的风险也就越大。秘密的加密算法已经完全不能适应现在大规模的网络环境。最好的方法是公布用来进行加密和认证的算法，仅依靠密钥的秘密性。这样可以保证在第三方仔细研究的情况下，算法依然是坚固的。
- *攻击者可能访问大量计算资源* 计算能力的开销在迅速地下降。我们应该假设攻击者能访问的计算机是该系统一生的时间中最大最强劲的，并增加若干数量级以备不可预计的发展。

- 最小化信任基础　系统中负责实现安全的那部分和它们所依赖的所有软硬件，是必须信任的——这也常被称为可信任计算基础。在这个可信任基础上的任何缺陷或程序错误都会产生安全漏洞，所以我们应该紧缩其大小。例如，应用程序就不足以被用户信任能保护其数据。

260

7.2 安全技术概述

本节旨在向读者介绍一些保护分布式系统和应用的重要技术和机制。这里我们将简单地对其进行说明，比较严格的描述将留在7.3节和7.4节。我们将使用图7-2中为主体所起的名字，并将图7-3所示的符号用做相应的标记。

K_A	Alice的保密密钥
K_B	Bob的保密密钥
K_{AB}	Alice和Bob共享的保密密钥
K_{Apriv}	Alice的私钥（只有Alice知道）
K_{Apub}	Alice的公开密钥（由Alice公布的，所有人都可读）
$\|M\|_K$	由密钥K加密的消息M
$[M]_K$	由密钥K签发的消息M

图7-3　密码符号

7.2.1 密码学

加密就是将消息编码以隐藏原有内容的一个过程。现代密码学包括一些加密和解密消息的安全算法，它们都基于秘密（也称为密钥）的使用。密钥是加密算法中用到的一个参数，也就是说，如果不知道密钥，就不可能进行加密的逆运算。

通常使用的加密有两类算法。第一类使用的是共享的保密密钥——发送者和接收者必须知道这个密钥，但不能让其他人知道。第二类加密算法使用的是公钥/私钥对——消息发送者用一个公共密钥——这个密钥已经被接收者公布了——来加密消息。接收者用一个相应的私钥对消息解密。尽管许多主体都会检测公开密钥，但只有接收者可以解密消息，因为他有私钥。

两种加密算法都非常有用，且都在建立安全的分布式系统中得到了广泛的使用。公开密钥的加密算法所需的处理能力一般要比保密密钥算法高100到1000倍，但它的便利性大大弥补了这一缺陷。

7.2.2 密码学的应用

密码学在安全系统的实现中扮演了三大角色，我们在此通过一些简单的场景概括地进行介绍。在本章后面几节中，我们会详细讨论这些和其他一些协议，着重解决此处提到的几个未解决的问题。

261

在下面的场景中，我们假设Alice、Bob和其他参与者已经对所用的加密算法达成了一致意见，同时也实现了这些算法。我们还假设任何保密密钥或私钥都会得到妥善的保存，不会被攻击者获得。

秘密性和完整性　密码学用于维持信息的秘密性和完整性，即使信息暴露于潜在的攻击下

也应如此，例如网络传输期间，信息很容易被窃听或者篡改。密码学的这种使用相当于它在军事和情报活动中的传统作用。它利用这一事实，也就是由特定加密密钥加密的消息只能由知道相应解密密钥的接收者解开。这样只要解密密钥没有被损害（泄露给未参加通信方）并且假设加密算法足以击败任何破解它的尝试，那么将能维持住加密消息的秘密性。如果包括像校验和这样的冗余信息并对之加以检查，那么加密过程也可维护加密信息的完整性。

场景1 用共享的保密密钥进行秘密通信：Alice想要秘密地发一些信息给Bob。Alice和Bob共享着一个保密密钥K_{AB}。

1. Alice使用K_{AB}和两人达成协议的加密函数$E(K_{AB}, M)$加密后，发出任意数量的消息$\{M_i\}K_{AB}$给Bob。（只要K_{AB}没有被损害，Alice就可以继续使用K_{AB}）。
2. Bob利用相应的解密函数$D(K_{AB}, M)$对加密消息解密后就可以读到原来的消息了。

Bob现在可以读到原有的消息M。如果当Bob解开消息的时候，消息是有意义的，或者更好的情况是，如果它包括Alice和Bob之间达成一致的值，例如消息的校验和，那么当时Bob就可以知道这个消息是来自Alice，而且没有被篡改过。但仍然存在一些问题：

- 问题1 Alice怎样将共享的密钥K_{AB}安全地发送给Bob?
- 问题2 Bob怎样知道一个$\{M_i\}$不是Alice以前发过，后来由Mallory截获并重发的加密消息？为了完成这样的攻击，Mallory并不需要有密钥K_{AB}——他可以简单地复制代表消息的位模式，然后送给Bob。例如，如果消息是表示一个付钱给某人的请求，那么Mallory就会让Bob多付了一次。

我们将在本章后面给出如何解决这些问题的答案。

认证 密码学可以用来支持主体间通信的认证机制。主体用特定的密钥成功解开消息后，如果它包括校验和或者（如果用了加密的块链接模式，见7.3节）其他期望出现的值，则可以假设消息是可信的。它们可以推断出具有相应加密密钥的消息发送者，如果这个密钥只为两方知道，那么还可以推断出发送者的身份。如果密钥为私人所有的，则成功的解密也就认证了来自一个特定的发送方的已解密的消息。

场景2 与服务器间的认证通信：Alice想访问Bob拥有的文件，也就是她工作单位的本地局域网中的一个文件服务器，Sara是一个认证服务器，它向用户发送口令，并且保存着系统中它服务的所有主体的当前保密密钥（通过在用户口令上进行一些转换）。例如，它知道Alice的密钥K_A和Bob的密钥K_B。在这个场景中，我们涉及票证，票证是由认证服务器发出的一个加密项，包括发送票证的主体的身份和一个用于当前通信会话的共享密钥。 262

1. Alice向Sara发送了一条（未加密的）消息，声明了她的身份，并向Sara要求一张访问Bob的票证。
2. Sara用K_A加密应答消息，并发给Alice，应答消息包括一个由K_B加密的票证（是发送给Bob的每次访问文件请求）和一个新的保密密钥K_{AB}，K_{AB}用于和Bob通信。于是Alice收到类似应答：$\{\{Ticket\}_{K_B}, K_{AB}\}_{K_A}$。
3. Alice用K_A解开应答（K_A是根据Alice的口令由同样的转换过程生成的，口令没有通过网络传输。一旦被使用后，就从本地存储中删除它，以免其被泄露）。如果Alice有正确的从口令中生成的密钥K_A，那么她就可以得到一个合法访问Bob服务的票证和一个新的和Bob通信的加密密钥。Alice并不能解开或篡改票证，因为它是由K_B加密的。如果接收者不是Alice，那么他不知道Alice的口令，从而也无法解开消息。

4. Alice将票证、自己的身份和一个访问文件的请求R一起发给Bob：$|Ticket|_{K_B}$，$Alice$，R。
5. 那个最先由Sara产生的票证，实际上是$|K_{AB}, Alice|_{K_B}$。Bob用自己的密钥K_B解开票证。Bob得到了Alice的身份认证（基于只有Alice和Sara知道Alice的口令这一点）和一个新的共享保密密钥K_{AB}，可以用来和Alice交互了。（这也被称为会话密钥，因为Alice和Bob可以安全地用它进行一系列交互）。

以上所说的是认证协议的一个简化版本，该认证协议最初是由Roger Needham和Michael Schroeder [1978]开发的，后来又在MIT [Steiner *et al.* 1988] Kerberos系统上得到了进一步的发展和使用，详见7.6.2节。在上面的简化版协议描述中，没有措施防止对旧认证信息的重发。这和其他一些弱点将在完全版的Needham-Schroeder协议（见7.6.1节）中解决。

我们描述的认证协议需要认证服务器Sara事先知道Alice和Bob的密钥K_A和K_B。这在一个单一的组织中是可行的。这时，Sara运行在一个物理安全的计算机上，并由可信的主体管理它，主体产生这些密钥的初始值，并通过单独的安全通道传输给相应的用户。但这在电子商务或其他广域应用上是不适合的，那里使用单独安全通道非常不方便，并且要求一个可信的第三方是不切实际的，公开密钥算法让我们摆脱了这种两难境地。

质询的有效性 Needham和Schroeder在1978年的一个重要突破是认识到用户的口令并不
263 需要在每次认证时都发送到一个认证服务（从而会暴露在网络中）。相反，他们引入了加密质询的概念。见上面那个场景的第2步，服务器Sara把用Alice的保密密钥K_A加密的票证发送给Alice。这里包括一个质询，因为Alice除非能解开这个票证，不然就不能使用它。而且只有在她知道K_A的时候才可以解开它，而K_A来自于Alice的口令。一个冒名顶替Alice的人在这点上就会被击败。

场景3 用公开密钥的认证通信：假设Bob已经生成了一个公钥/私钥对，下面的对话可以使Bob和Alice建立一个共享保密密钥K_{AB}。

1. Alice访问一个密钥分发服务得到公开密钥证书，公开密钥证书给出了Bob的公开密钥。它之所以称为证书，是因为它是由一个可信的权威机构签发的——一个广为人知的可靠的人或组织。在检验过签名后，Alice从证书中读出Bob的公开密钥K_{Bpub}。（我们在7.2.3节讨论公开密钥证书的构造和使用。）
2. Alice创建一个新的与Bob共享的密钥K_{AB}，并用一个公开密钥的加密算法和K_{Bpub}对新密钥加密。她将结果和一个能惟一标识一个公钥/私钥对的名字一起发给Bob（因为Bob可能有多个这样的对）。于是Alice发给Bob的是：密钥名字、$\{K_{AB}\}K_{Bpub}$。
3. Bob从他的私钥库中选出相应的私钥K_{Bpub}，并用它解开K_{AB}。注意Alice给Bob发的消息在传输过程中会被破坏和篡改，结果也就是Alice和Bob不能共享这个密钥K_{AB}。如果这是个问题的话，可以比较巧妙地绕过，即在消息中加入协商好的值或字符串，例如Alice和Bob的名字或电子邮件地址，这样Bob就可以在解密后检查一下。

上面的场景说明了如何使用公开密钥密码学发送一个共享的保密密钥。这项技术被称为混合密码协议并被广泛使用，因为它结合了公开密钥算法和保密密钥算法二者的优点。

问题 这种密钥交换很容易受到中间人攻击。Mallory可能截取Alice最初向密钥分发服务索要Bob公开密钥证书的请求，并回复一个包括自己公开密钥的消息。然后他就可以截取所有后续的消息。按我们上面所说的，为了防止这种攻击，我们要求Bob的证书应该由一个众所周知的权威机构来发送。同时，Alice必须确保她收到的Bob的公开密钥证书是由

一个公开密钥（下面将会讲到）签发的，此公开密钥是以安全方式收到的。

数字签名 我们将使用密码学实现一种称为数字签名的机制。它的任务就是模拟通常意义的签名，用于向第三方核实消息或文档在签名人完成后未被改变过。

数字签名技术基于一个只有签名人知道的秘密不可逆转的对消息或文档的绑定。这可以通过对消息加密来实现——或更好的方法是用只有签名人知道的密钥将消息压缩成摘要。摘要是由一个安全摘要函数计算而成的固定长度的值。安全摘要函数类似于校验和函数，但它 264
不会为两个不同的消息产生一个相似的摘要值。加密的摘要附在消息上作为签名。公开密钥密码学通常这样使用：最先的签名人用他们的私钥产生一个签名，签名可以用相应的公开密钥被任何接收者解开。还有一个额外的要求：验证人必须能确信此公开密钥是属于被称为签名人的主体的——这由公开密钥证书的使用来解决，见7.2.3节的描述。

场景4 使用安全摘要函数的数字签名：Alice要签名一个文件M，使得任何接收者能核实她是这个文件的签发人。这样，当Bob后来通过任何途径或从任何资源（例如来自消息或者一个数据库）访问这个签了名的文件时，他就可以验证Alice是它的签发人。

1. Alice对文件计算出一个固定长度的摘要 $Digest(M)$。
2. Alice用她的私钥为这个摘要加密，并附在M上，再将M，$\{Digest(M)\}_{K_{Apriv}}$公布给需要的用户。
3. Bob得到这个签了名的文件，抽取出M并且计算$Digest(M)$。
4. Bob用Alice的公开密钥K_{Apub}解开 $\{Digest(M)\}_{K_{Apriv}}$，将结果和自己计算的$Digest(M)$做比较，如果相匹配的话，签名就是有效的。

7.2.3 证书

数字证书是由主体签发的、包含一个声明（通常较短）的文档。我们用一个场景来说明这个概念。

场景5 使用证书：Bob是一个银行。当他的顾客和他建立联系时，即使他们以前从来没有和Bob接触过，他们需要能确认他们是在和银行Bob交互。Bob则在给他们权限访问他们的账户前，要对其身份加以验证。

例如，Alice觉得从她的银行获得一张证明她有银行账号的证书（如图7-4所示）很有用。Alice可以在购物时用到这个证书，以证明自己在Bob银行开了户。证书由Bob银行的私钥K_{Bpriv}签发。供货商Carol如果能验证第5个域中的签名，她就可以接受这个证书，用Alice的账号为商品付款。为此，Carol需要有Bob的公开密钥，而且还要进行验证，防止Alice签发了一个假的将自己名字关联到别人账号的证书。而为了能造此类型的假，Alice只要产生一个新的$K_{B'pub}$、$K_{B'priv}$密钥对，并用它们产生一个假的证书，且声称它来自于Bob银行就可以了。

1.证书种类：	账号
2.姓名：	Alice
3.账号：	6262626
4.证明方：	Bob的银行
5.签名：	$\{Digest(field2 + field3)\}_{K_{Bpriv}}$

图7-4 Alice的银行账号证书

Carol现在需要的是由有名的可信权威机构签发的、含有声明了Bob公开密钥的证书。我

们假设Fred代表银行家联盟，其职责之一是证明银行公开密钥。Fred为Bob发行了一个公开密钥证书（如图7-5所示）。

1. 证书种类：	公开密钥
2. 姓名：	Bob的银行
3. 公开密钥：	K_{Bpub}
4. 证明方：	Fred——银行家联盟
5. 签名：	$\{Digest\ (field2\ +\ field3\ )\}_{K_{Fpriv}}$

图7-5 Bob银行公开密钥的证书

当然，这个证书还需要依靠Fred公开密钥K_{Fpub}的真实性，这样我们就面临一个真实性的递归问题——如果Carol能确信她知道Fred真实的公开密钥K_{Fpub}，她只能依靠这个证书。我们可以让Carol用某种可信的方式得到K_{Fpub}，从而打破这一递归——证书可能是由Fred的一个代表亲手交给她或者她从自己认识且信任的人处收到一个签名的证书，而此人说这个证书直接来自Fred。我们的例子说明了一个证书链，当前情况就是一个有两个环节的链。

我们已经间接提到证书引发的一个问题——如何选择一个可信的权威机构，使得认证链得以开始。信任很少是绝对意义上的，因此选择权威机构就必须取决于证书打算是给谁的。其他问题随私钥被损害（泄露）的危险和一个证书链可容许的长度而引发——证书链长度越长，一个脆弱的链环就要冒更大的风险。

如果仔细地解决了这些问题，证书链就成为了电子商务和其他真实世界事务的重要基础。它们帮助解决了规模问题：这个世界有60亿的人口，我们怎样才能在任意人之间建立起信任关系？

证书可用于建立多种声明的真实性。例如，一个小组或协会的成员可能要维护一份电子邮件列表，并只对组内成员公开。解决这一问题的一个办法，是让成员资格管理者（Bob）给每个成员发送一个成员资格证书（S, Bob, $\{Digest(S)\}_{K_{Bpriv}}$），这里$S$表示“Alice是友好社的一个成员，$K_{Bpriv}$是Bob的私钥”这类的声明。想要加入友好社电子邮件列表的成员必须向列表管理系统提供这个证书的一个副本，而它会在检查证书后，才允许Alice加入此列表。

265
|
266

为了使证书有用，需要做两件事情：

- 证书需要一个标准的格式和表现形式，这样证书签发者和证书用户就可以成功地构造并解释证书。
- 证书链的构造方式必须达成一致，特别是权威机构的概念。

我们将会在7.4.4节讨论这些需求。

有时需要撤销一个证书——例如Alice不想继续成为友好社的成员，但她或其他人还会保留她的成员证书的副本。跟踪并删除所有这类证书，开销昂贵或不可能。而且使一个证书无效也是不容易的——它要通知所有可能接收这个被撤销的证书的接收者。通常解决此问题的办法是在证书中包括一个过期日期。收到过期证书的人应该将它抛掉。证书的主题也必须请求更新自己。如果需要更加迅速地撤销，就要借助于以上这些麻烦的机制了。

7.2.4 访问控制

这里我们将概述分布式系统中对资源访问的控制所基于的概念以及实现的技术。在Lampson[1971]的一篇经典论文中非常清晰地给出了保护和访问控制的概念基础。而非分布式

的实现细节可以在许多操作系统[Stallings 1998b]的书中看到。

从历史看，分布式系统中的资源保护大部分是面向特定服务的。服务器收到此种格式的请求消息：*<op, principal, resource>*，其中*op*是所请求的操作，*principal*是发请求的主体的一个标识或者一组证书，resource用于识别操作将应用的资源。服务器必须先认证请求消息和主体的证书，然后应用访问控制，从而拒绝主体没有在特定的资源上完成某类操作的访问权限的任何请求。

在面向对象的分布式系统中，可能会有很多种对象必须应用访问控制。而具体的决定又经常是应用特定的。例如，每天只允许Alice从银行取一次现金，而允许Bob取三次。访问控制的决定通常是留给应用层的代码来处理，但也提供一些通用的支持，包括主体的认证、请求的签名和认证、证书和访问权限数据的管理。

保护域 保护域是一组进程共享的一个执行环境：它包括一组*<resource, right>*对，列出了在域内执行的所有进程能访问到的资源以及每个资源所能进行的操作。保护域通常是和给定的主体相关联的——当一个用户登录时，认证她的身份，并为她要运行的进程建立一个保护域。概念上，这个域包括了主体所具有的所有访问权限，包括她以多个小组成员身份得到的权限。例如，在UNIX中，进程的保护域是由在登录时附在该进程上的用户或组的标识决定的。权限是按照允许的操作来指定的。例如，一个文件对于这个进程来说既可读，也可写，而对另一个进程来说可能只可读。 267

保护域只是一个抽象。有两种实现在分布式系统中普遍使用，即权能和访问控制列表。

权能 每个进程在它所在的域中都具有一组权能。权能是一个二进制值，作为允许所有者对特定资源进行某种访问的权限。在分布式系统中，权能必须是不可伪造的，采用形式如下所示：

资源标识	对目标资源的惟一标识
操作	允许对资源进行的操作
认证代码	使权能不可伪造的数字签名

当服务认证了客户是属于自己的保护域时，它就给客户提供权能。权能中的操作是为目标资源定义的操作的一个子集，通常被编码成一个位图形式。可以用不同的权能表示对同一资源不同的访问权限。

使用权能时，客户请求的形式是*<op, userid , capability>*。请求包括要访问的资源的权能，而不是一个简单的标识，这可以使得服务器立刻就能知道客户有访问该资源的权限。对附有权能的请求的访问控制检查包括检查权能的有效性以及检查请求的操作是否在权能允许的集合中。这是权能机制的主要优点——它们组成一个自包含的访问钥匙，就像物理门锁的钥匙是进入受保护的大楼的关键。

权能同样也存在物理门锁的钥匙的两个缺点：

- *钥匙被偷窃* 任何有钥匙的人都可以用它进入大楼，无论他是否是这把钥匙合法的拥有者——他们可以用偷盗或其他欺诈手段来得到钥匙。
- *回收问题* 保管钥匙的资格会随时间变更。例如钥匙拥有者不再是大楼主人的雇员，但他还保管或者复制了一把钥匙，他就有可能以不合法的方式使用它。

对于物理钥匙的这些问题，可行的解决办法是（1）将违法的钥匙拥有者投进监狱——这并不能永远防止那些违法事情的发生；（2）换锁并重发钥匙给所有的钥匙保管者——一个比

较笨而且昂贵的办法。

在权能上类似的问题有：

- 由于不小心或者窃听攻击，权能会落入不是应该被发送到的主体手中。一旦这样，服务器很难阻止它们被非法使用。
- 取消权能是很困难的。持有者的状态可能会改变，因此其访问的权限也该相应地改变，但他们依然能够使用权能。

268

解决这两个问题的方案已被提出并开发，一是包括对持有者身份验证的信息，二是设置超时并附带回收权能的列表[Gong 1989，Hayton *et al.* 1998]。尽管他们为原本简单的概念增加了复杂性，但权能依然是一个重要的技术，例如，它们可以和访问控制列表一起使用来优化对同一资源的重复访问，它们为实现委托提供了最简洁的机制[见7.2.5节]。

非常有趣的事是权能和证书的相似性。回想一下7.2.3节介绍的证明Alice有其银行账号的证书。它与权能的区别在于没有允许操作的列表，也不识别发送者。在某些环境下，权能和证书是可以互换的概念。只要请求者能被证明是Alice本人，Alice的证书就可以被看成是向Alice的银行账号做一切账号持有者允许的操作的访问权限凭证。

访问控制列表　每个资源都有这个列表，格式为<*domain*，*operations*>，它指出了对该资源有访问权限的域和域所允许的操作。一个域可以由一个主体的标识指定，也可以是一个用于确定主体所在域的表达式。例如，文件的所有者是一个表达式，它的值可以用存在文件中的所有者的标识和主体的标识作比较而求得。

这是大多数文件系统采用的方案，包括UNIX和Windows NT，即每个文件都附有一组表示访问权限的比特，同时根据存在每个文件中的所有者信息定义权限被授予的域。

发向服务器的请求具有<*op*，*principal*，*resource*>的形式。对每一个请求，服务器会验证主体，并会检验所请求的操作是否包含在相关资源的访问控制列表中。

实现　数字签名、证书和公开密钥证书提供了安全访问控制的密码学基础。安全通道提供了性能优势，利用它可以使得在处理多条请求时不需要重复地检查主体和证书[Wobber *et al.* 1994]。

CORBA和Java都提供了关于安全性的API。支持访问控制是它们的一个主要目的。Java为分布式对象提供了支持，包括用Principal类、Signer类、ACL类和默认的认证方法进行访问控制，还有对证书、签名有效性验证及访问控制检查的支持。Farley [1998]为Java的这些特色作了一个很好的介绍。对于Java程序包括移动代码的保护是基于保护域的概念——为本地代码和下载的代码提供不同的保护域并在不同的保护域内执行。每个下载的资源都可以有一个保护域，对不同的本地资源的访问权限取决于下载代码中设置的信任级别。

269

CORBA向ORB提供了一个安全服务规范[Blakley 1999，OMG 1998b]及其模型，提供安全通信、认证、基于证书的访问控制、ACL和审查，这将在17.3.4节做进一步的描述。

7.2.5 凭证

凭证是主体在请求访问某个资源的时候提供的一组证据。在最简单的情况下，一个从相关权威机构发出的用于证明主体身份的证书就足够了，它可以被用来在一个访问控制列表中检查主体所允许的操作（见7.2.4节）。这经常就是所有要提供的，但这些概念还可以再推广一下，以处理更加微妙的需求。

对于用户来说，在每次需要访问受保护的资源时都让他们的权威机构去验证是很不方便

的。替代的方法是引入“凭证证明主体”的概念。这样，用户的公开密钥证书可以证明用户——任一进程收到由用户的私钥认证的请求，都可以假设请求就是由该用户所发出的。

证明的想法还可以进一步的延伸。例如，在一个合作任务中，可能要求一些敏感的操作只能由某个组的两名成员授权来完成。在这种情况下，请求这个操作的主体就会提交自己的凭证和该组另外一个成员的凭证，并表明在检查凭证时它们是一起的。

相似地，选举投票时每个选举请求都会附有选举人的证书和一张身份证书。委托证书允许主体可以代表另外一个人来操作等等。通常，访问控制检查包括对一个与证书相关的逻辑公式的求值。Lampson等[1992]提出了一个认证逻辑，用于评估由一组凭证形成的证明授权。Wobber等[1994]描述了一个系统，用于支持这种非常通用的方法。在真实世界合作化任务中使用到的、有关凭证的有用形式的进一步工作，可以在[Rowley 1998]找到。

在设计实际的访问控制方案[Sandhu *et al.* 1996]时，基于角色的凭证显得尤为有用。为组织机构或合作性任务，可以定义成组的基于角色的凭证，应用层的访问权限也可通过这些凭证建立起来。在特定的任务或组织机构中，角色可以用生成一个角色证书（它将主体与一个指定的角色相关联）的途径，分配给特定的主体[Coulouris *et al.* 1998]。

委托 凭证的一个特别有用的形式是某个主体或代理某个主体的进程，在另一个主体授权的情况下，执行某个操作。下列情况需要有委托：服务需要访问一个受保护的资源，目的是代表其客户完成一个动作。考虑打印服务器接收打印文件的请求。复制整个文件将是对资源的浪费。所以用户只将文件的名字送到打印服务器，而由打印服务器代表用户来访问文件。如果这个文件是读保护的，那么打印服务器只有得到临时的读权限，才能进一步工作。委托 270
就是为了解决此类问题而设计的一种机制。

委托可以用委托证书或者权能来实现。证书由请求的主体签发，它授权另外一个主体（在我们的例子中指打印服务器）访问一个指定的资源（要打印的文件）。在支持权能的系统中，它也可以不需要识别主体而达到同样的效果——访问某资源的权能放在请求中，一起送到服务器。权能是一个不可伪造的、资源访问权限的编码集。

委托权限后，一般会限制受委托方使用的权限为委托人权限的子集。这样受委托方就不会错用这些权限。在我们的例子中，证书应该是有时间限制的，以防止因打印服务器的代码受损害，而使得文件被泄露给第三方。CORBA安全服务包括一个基于证书的权利委托机制，支持对权限的限制。

7.2.6 防火墙

3.4.8节就对防火墙进行过介绍。防火墙保护内部互联网，对流入和流出的通信进行过滤。这里我们讨论它作为安全机制的优点和缺点。

在理想的情况下，通信总是在相互信任的进程中进行，且总是使用安全通道。有许多理由可以解释为什么不能达到这种理想的情况，有些是分布式系统开放的本质所固有的或者是由大多数软件中仍存在的错误导致的。由于请求消息可以轻松地被送到任何地方的任何服务器，以及大多数服务器在设计时就没有考虑到防范黑客的恶意攻击和突发性错误，这使得机密信息很容易从组织的服务器里泄露出去。一些不想要的东西也会渗透进组织的网络，例如蠕虫程序或病毒进入计算机。对防火墙的评论参见[web.mit.edu II]。

防火墙创造了一个本地通信环境，使得所有的外部通信都被截取。只有通信被显式授权，

消息才会发往本地的接收者。

访问内部网络可能要受到防火墙的控制，但访问因特网上的公共服务是不受限制的，因为其目的是为广大用户提供服务。使用防火墙并不能保护组织免受内部的攻击，而它对外来访问的控制也是粗糙的。人们需要细粒度的安全机制，使得个人用户在不损害私密性和完整性的前提下能够与选定的其他人分享信息。Abadi等[1998]提供了一个基于Web隧道机制供外部用户访问私人Web数据的方法，该机制可以被集成到防火墙中去。该机制提供了一个基于HTTPS协议（SSL上的HTTP）的安全代理，而这些可信和认证过的用户将通过此代理访问内部的Web服务器。

防火墙对于拒绝服务攻击（如我们在3.4.2节提到的基于IP伪冒的那个攻击）不是很有效。问题在于这种攻击生成的洪水般的消息淹没了任何一个像防火墙之类的单点防御。所以必须在目标的上游对洪水般的消息加以处理。使用服务质量机制限制网络中消息流，将它控制在
271 目标所能处理的级别内，这种处理方式看起来是最有发展前途的。

7.3 加密算法

发送方用某种规则将明文消息（正常顺序的比特流）转换成密文消息（改变了顺序的比特流）再发送出去，这就是加密消息的过程。接收方必须知道这个转换规则，才能够将密文转为原来的明文。其他主体无法解密该密文，除非他们知道这个规则。加密的转换过程由两部分定义：函数E、密钥K。加密后的消息写成：$\{M\}_K$。即：

$$E(K, M) = \{M\}_K$$

加密函数E定义了一个算法，用于将明文中的数据项，通过与密钥结合和调换位置，转化成加密的数据项，对于它们的变换很大程度上依靠的是密钥的值。我们可以认为一个加密算法是一簇函数的规约，通过给定的密钥可以从中选出一个。解密算法是由一个逆函数D来执行，它也以一个密钥作为参数。对保密密钥加密而言，解密使用的密钥和加密使用的是一样的：

$$D(K, E(K, M)) = M$$

因为对称地使用密钥，所以保密密钥密码学也经常被称为对称密码学，而公开密钥密码学就被称为非对称的，因为用于加密和解密的密钥是不一样的。下面我们将描述这两种类型的几个广泛使用的加密函数。

对称算法 如果不考虑密钥参数即定义$F_K([M]) = E(K, M)$，那么我们就得到强加密函数的一个性质，即$F_K([M])$相对容易计算，而其逆$F_K^{-1}([M])$难于计算，而这使得此种想法不太可行。这些函数被称为单向函数。加密信息的有效性取决于具有单向性质的加密函数F_K，也是由F_K在给出$\{M\}_K$的情况下保护M免于被解开。

对于像下一小节所介绍的设计巧妙的对称算法，K的大小决定了从明文M及相应密文$\{M\}_K$的情况下求出K的运算量。通常最有效的也是最拙劣的攻击形式是一种被称为的强行攻击的形式。强行攻击的方法是：运行所有可能的K值，求出$E(K, M)$，和已知的$\{M\}_K$比较，直到相互匹配为止。如果K有N位，那么强行攻击平均要进行2^{N-1}次迭代，最多要进行2^N次迭代才
272 能找到K。因此破解K的时间是K的位数的指数级时间。

非对称算法 当使用公钥/私钥对的时候，单向函数就以另外一种形式得到了利用。第一个可行的公开密钥方案是由Diffie和Hellman[1976]提出的，它作为一种密码学方法，消除了通信

双方间互相信任的需要。所有公开密钥方案的基础是陷门函数。陷门函数是一个有秘密出口的单向函数——它在一个方向是容易计算的，而求其逆，如果不知道密钥，几乎是不可能的。寻找这样的函数并将其应用到实际的密码学方案中的可能性是Diffie和Hellman第一次提出的。此后，一些实际的公开密钥方案被提出并得到开发，它们都依靠使用大数函数作为陷门函数。

非对称算法中所要使用的密钥对是由一个公共根导出的。7.3.2节描述的RSA算法所使用的根是任意选择的非常大的素数对，再由一个单向函数从根导出密钥对。在RSA算法中，将两个大素数相乘——即使是非常大的素数，该计算也只需几秒钟，最后的乘积N当然又比被乘数大得多。在某种意义上来说，这种使用乘法的过程就是单向函数，因为想从乘积得到原来的被乘数——即乘积的分解——在计算上是不可行的。

密钥对的一个用来加密。在RSA中，加密函数隐藏明文的方式是将每个比特块作为二进制数字，用密钥为指数对其作求幂运算，再将结果对N取模，结果值是相应的密文块。

*N*的大小和密钥对中的至少一个比对称密钥所需的安全密钥大得多，以保证*N*是不可分解的。因为这个原因，对于RSA强行攻击的可能就很小了，它对攻击的抵抗力主要依赖于分解*N*的不可行性。我们将在7.3.2节讨论*N*的安全大小。

块密码 大多数加密算法是在固定大小的数据块上操作，64位是块通常的大小。消息被分割成多个块，如果必要的话，最后一块会补足到标准长度。每个块被独立地加密，第一个块一旦加密好了，就可以用于传输了。

对于简单的块加密，每个块的密文值都不依赖于前面的块。这形成了一个弱点，因为攻击者可以识别重复的模式并推算出它们和明文间的关系。另外消息的完整性也得不到保证，除非使用校验和或安全摘要机制。大多数块加密算法使用密文块链（CBC）来克服这些弱点。

密文块链 在密文块链模式中，每个明文块在加密前，先和前面块的密文进行异或操作（XOR），因为它加密过（如图7-6所示）。解密时，块先被解密，再和前面块的密文（应该将它保存起来）进行XOR操作，从而得到原先的明文块。这种方法能成功是因为XOR操作是自反的——两次应用它会产生原来的值。

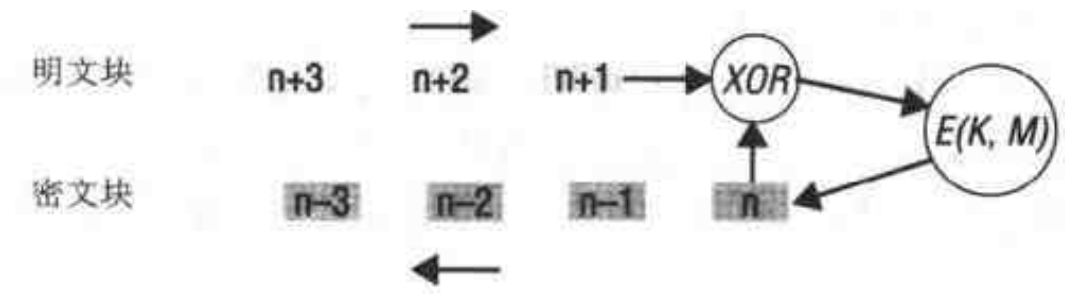


图7-6 密文块链

CBC会防止明文中的相同部分加密后在密文中还是相同的。但在每个块序列的起始处都存在着一个弱点——如果我们要与两个地址建立加密的连接，并向其发送同样的消息，那么加密的块序列就是一样的。这样窃听者就可以从中得到有用的信息。为了防止这样的漏洞，我们需要在每个消息的前面加一段不同的明文。这样的明文叫做*初始向量*。时间戳是一个很好的初始向量，它强制每个消息都以不同的明文块开头。这和CBC操作结合在一起，产生的结果就是即使相同的明文，也转化成了不同的密文。

使用CBC模式必须保证加密数据在可靠连接上传输。任何密文块的丢失都会导致解密的失败，因为解密过程不能解开后续的块。因此它不适合用于第15章所描述的程序，那里能容忍一些数据丢失。流密码将用于这样的环境。

流密码 对于一些应用，例如对电话交谈的加密，块的加密方法就不太恰当了。因为数据流是多个实时产生的小块。数据采样可以小到8位，甚至到1个位。这样将它们补足到64位再加密传输就显得特别浪费。流密码是一种增量式加密的加密算法，它每次1位地将明文变为密文。

听起来很难实现，但实际上将一个块密码算法转换成流密码算法来使用是很容易的。技巧在于构造一个密钥序列发生器。密钥序列是任意长度的一个比特序列，通过将其和数据流的内容进行XOR操作，从而完成加密的过程（如图7-7所示）。如果这个密钥序列是安全的，那么得到的加密数据流也是安全的。

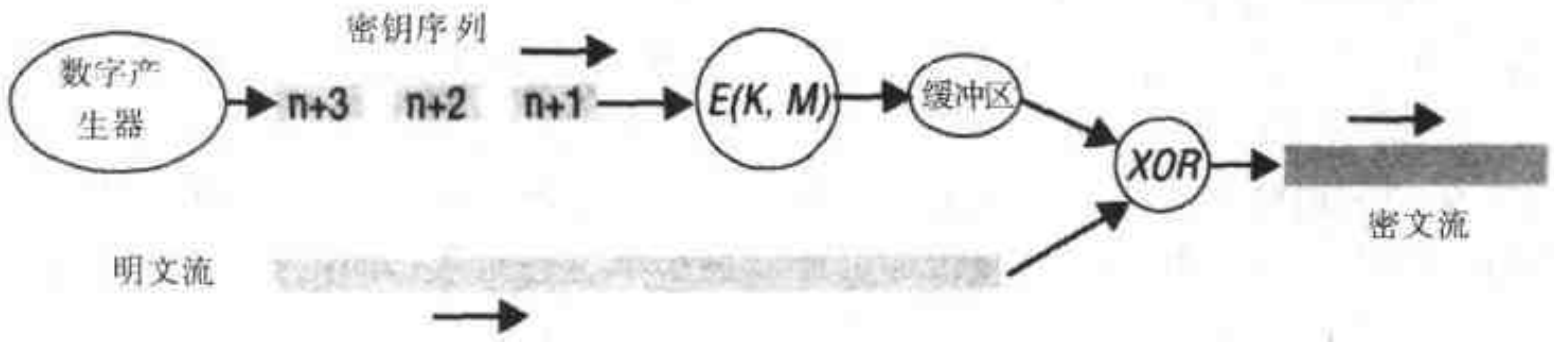


图7-7 流密码

这种想法和在智能社区避免窃听而用到的“白噪声”的方法是类似的。白噪声就是在对室内的交谈录音时，加入白噪音，以掩盖谈话内容。如果房间的嘈杂声和白噪声是分开录制的话，那么就可以从嘈杂的谈话录音中去掉白噪音的录音，从而得到没有噪音的原谈话内容。

273 ~ 274

密钥序列发生器是对一个范围的输入值重复地应用一个数学函数，得到一个连续的输出值序列，然后将这个输出值连接起来组成明文块，再将这些块以收发双方共享的密钥加密。密钥序列还可以进一步利用CBC来伪装，得到的加密块就用来作为密钥序列。几乎所有函数的迭代都可以产生一组不相同的非整数值，因此都可以作为整个过程的原材料，但通常我们使用的是一个随机数发生器，其初始值是由收发双方协商决定的。为了保证数据流服务的质量，密钥序列块应该比用到它们的时间略微提早一会儿产生、同时产生它们的进程也不应有太多的处理工作以免数据流被延迟。

这样，原则上，在可以提供充足的处理能力进行实时加密密钥序列的情况下，实时数据的加密可以像批量的数据一样安全。当然，有些设备，例如移动电话，可以从实时加密的过程中得到好处，但它没有强大的处理器，这种情况下有必要相应地降低它的密钥序列算法的安全性。

加密算法的设计 有很多设计得很好的加密算法，例如$E(K, M) = |M|_K$，它隐藏了M的值，从而使得想找到K的值基本上不可能比强行攻击快。所有的加密算法都依赖于使用基于信息论[Shannon 1949]的原则，对M进行了信息保留操作。Schneier[1996]将Shannon的两个基本原理，混乱和扩散，用于隐藏密文块M的内容，将内容和一个足够大小的密钥K相组合，可以对付强行攻击。

混乱 非破坏性的操作如XOR和循环移位用于将每个明文块和密钥相组合，产生一个新的位模式，从而隐藏M和$|M|_K$中各个块之间的关系。如果一个块有多个特征，那么这样就可以抵抗住基于特征频率分析的攻击。（WWII德国Enigma机器使用的是链式单字母块，它有可能被统计分析攻击。）

扩散 在明文中通常会有重复和冗余。扩散是通过对每个明文块调换位置来消除规律性模

式。如果使用CBC，稍长一点的正文依然会产生冗余。流密码不能使用扩散，因为不存在块。

在下面两小节，我们将讨论几个重要的实用算法的设计。所有这些算法都是在上述这些基本原理的指导下设计的，它们也经过了严格的分析，可以抵挡所有已知的攻击，并有相当的安全富余。除了TEA算法只是用于说明性的目的，其他讨论的算法都广泛应用在一些需要强大安全性的程序里。它们中有些还有一些小的漏洞或需要考虑的地方，由于篇幅所限，我们不能在这里讨论所有要考虑的问题，读者可以自己参阅Schneier [1996]来获取进一步的信息。我们在7.5.1节中总结和比较这些算法的安全性和性能。

不需要理解加密算法操作的读者，可以跳过7.3.1和7.3.2。 275

7.3.1 保密密钥（对称）算法

近年来开发和发布了许多加密算法。Schneier[1996]描述的对称算法多达25种以上，其中很多都被认为对于已知的攻击是安全的。我们在此只描述其中的3种。第一个是TEA，因其设计和实现上的简单性，我们用它来具体说明这一类算法的本质。然后讨论DES和IDEA算法，不过篇幅会少一些。多年来DES一直是美国的国家标准，但现在这变得充满了历史性的趣味，因为56位的密钥太短了，对于有现代硬件条件下的强行攻击是无法抵挡的。IDEA采用128位的密钥，它可能是最有效的对称块加密算法，并且对于大量数据的加密，它也具有多方面的优点。

1997年，美国国家标准和技术研究所（NIST）颁布了一项提议，建议采用一种新的算法作为新的高级加密标准（AES）。下面我们将介绍此项活动的一些进展。

TEA 上面概述的对称算法的设计原则都在剑桥大学开发的微加密算法[Wheeler and Needham 1994]中得到了很好的说明。C语言形式的加密函数如图7-8所示。

```
void encrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];                                  1
    unsigned long delta = 0x9e3779b9, sum = 0; int n;                        2
    for (n= 0; n < 32; n++) {                                                3
        sum += delta;                                                        4
        y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);                5
        z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);                6
    }
    text[0] = y;  text[1] = z;                                               7
}
```

图7-8 TEA加密函数

TEA算法利用多轮整数加法、XOR（运算符^）和逻辑移位（<<和>>）来完成对明文中位模式的混乱和扩散。每个明文块是64位的，所以就以两个32位整数的形式保存在向量*text[]*中。密钥是128位长的，表示成4个32位的整数。

在32轮的每一轮中，正文的两半分别与正文逻辑移动后的部分及密钥逻辑移动后的部分相组合，见程序的第5标号行和第6标号行。XOR的使用和正文的移位完成了混乱，正文两部分的移位和交换则完成了对明文的扩散。在每个循环中，常数*delta*与正文的每个部分相组合，以免密钥因正文中一部分没有变化而泄露。解密函数是加密的逆函数，如图7-9所示。

这段程序提供了一个安全和合理快速的密钥加密算法。它的性能大约是DES算法的3倍，而它的简明性也有助于优化和硬件实现。128位的密钥足以对付强行攻击。它的作者和其他人

只发现了它两个很小的漏洞，参见[Wheeler and Needham 1997]。

```
void decrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = delta << 5;  int n;

    for (n= 0; n < 32; n++) {
        z -= ((y << 4) + k[2]) ^ (y + sum) ^ ((y >> 5) + k[3]);
        y -= ((z << 4) + k[0]) ^ (z + sum) ^ ((z >> 5) + k[1]);
        sum -= delta;
    }
    text[0] = y; text[1] = z;
}
```

图7-9 TEA解密函数

为了说明它的使用，图7-10给出了一个简单的使用TEA的程序，可以对以前打开的文件进行加密或者解密（使用了C语言*stdio*库）。

```
void tea(char mode, FILE *infile, FILE *outfile, unsigned long k[]) {
/* mode is 'e' for encrypt, 'd' for decrypt, k[] is the key.*/
    char ch, Text[8]; int i;

    while(!feof(infile)) {
        i = fread(Text, 1, 8, infile);          /* read 8 bytes from infile into Text */
        if (i <= 0) break;
        while (i < 8) { Text[i++] = ' ';}       /* pad last block with spaces */
        switch (mode) {
        case 'e':
            encrypt(k, (unsigned long*) Text); break;
        case 'd':
            decrypt(k, (unsigned long*) Text); break;
        }
        fwrite(Text, 1, 8, outfile);            /* write 8 bytes from Text to outfile */
    }
}
```

图7-10 TEA的使用

DES 数据加密标准（DES）[National Bureau of Standards 1977]由IBM开发，随后被采用作为美国的国家标准，用于政府和商业应用上。在这个标准中，加密函数用56位的密钥将64位的明文映射成64位的加密输出。算法中有16个依赖密钥的阶段，被称为轮。每个轮中，要加密的数据都会根据由密钥决定的一组二进制位和3个不依赖密钥的移位值进行转换进行位轮换。该算法对于20世纪70和80年代的计算机上的软件是非常耗时的，但它在VLSI硬件中实现，并且可以轻松地结合到网络接口和其他的通信芯片上。

276 ~ 277

在1997年6月，它被一个广为宣扬的强行攻击成功地破解了。此次攻击是在一次竞赛中为了演示低于128位的密钥缺乏安全性而进行的[www.rsasecurity.com I]。一个因特网用户社团召集了1000到14000台计算机（PC或者工作站），在上面运行相应的客户端程序[Curtin and Dolske 1998]。

客户端程序旨在破解出在已知的明文/密文采样中使用的特定的密钥，并用它解开加密的消息。客户端与一个服务器交互信息，该服务器负责协调它们的工作，向它们发送要检查的一段密钥值，并负责从它们那里接收相应的进展报告。一般客户计算机将客户端程序作为一个后台活动运行，这类计算机的性能大约相当于200MHz的Pentium处理器。密钥可在12周内被破解，大约检查了总可能值（2^{56}或6×10^{16}）的25%。1998年由Electronic Frontier Foundation[EFF 1998]建造的一台机器，可以在3天左右的时间成功地破解DES密钥。

尽管在很多商业和其他应用中依然使用着DES算法，但基本形式的DES被认为是过时了，除了一些低价值的信息保护。经常使用的一种算法被称为三重DES加密算法（或3DES）[ANSI 1985，Schneier 1996]。它包括用两个密钥K_1和K_2使用3次DES。

$$E_{3DES}(K_1, K_2, M) = E_{DES}(K_1, D_{DES}(K_2, E_{DES}(K_1, M)))$$

这相对于给出了一个112位的密钥，也就有了充足的对付强行攻击的力量。但其缺点是性能差，这是由于对一个按现代标准已经较慢的算法应用了3次。

IDEA 国际数据加密算法（IDEA）是在20世纪90年代初作为DES的接任者被开发出来的[Lai and Massey 1990，Lai 1992]。像TEA一样，它使用了128位的密钥来加密64位的块，它主要基于群代数的运算，有8轮的XOR、模2^{16}的加法和乘法。对DES和IDEA而言，同样的函数既可以用于加密，也可以用于解密；这个性质有助于算法在硬件上的实现。

对IDEA也进行了广泛的分析，还没有发现重大的漏洞。它加密和解密时间约为DES的3倍。

AES 1997年，美国国家标准和技术研究所（NIST）颁布了一个提议，采用一个安全有效的算法作为新的高级加密标准（AES）[NIST 1999]。

作为对最初AES提议的回应，密码研究团体一共递交了15种算法。经过一段激烈的学术上的讨论，其中的5个被选出进入下一阶段的评估。所有的候选算法都支持128、192和256位 278
的密钥大小，而最后的5个表现了很高的性能。一旦颁布，AES可能会成为最广泛使用的对称加密算法。

7.3.2 公开密钥（非对称）算法

至今只有少数实用的公开密钥方案被实现。它们都使用大数的陷门函数产生密钥。密钥K_e和K_d是一对很大的数字，而加密函数用它们其中之一进行运算，例如对M作求幂运算。解密是使用另外一个密钥的一个类似函数。如果求幂使用了模的运算，可以证明结果和M的原值是相同的，即：

$$D(K_d, E(K_e, M)) = M$$

想要和别人进行安全通信的主体产生一对密钥K_e和K_d，并对解密密钥K_d加以保密。而加密密钥K_e可以公开，以供任何想要通信的人使用。加密密钥K_e可以看成单向加密函数E的一部分。而K_d是使得主体p能够转换出加密内容的一部分秘密信息。所有K_e的拥有者都可以加密消息$\{M\}_{K_e}$，而只有拥有密钥K_d的主体才可以操作这个陷门。

大数函数的使用造成了在计算函数E和D时有很大的运算开销。我们后面可以看到，这个问题的解决是仅在安全通信会话的初始阶段使用公开密钥。RSA算法显然是最为广泛使用的公开密钥算法，我们将在此详细介绍它。另一类算法是基于平而椭圆曲线行为派生的函数，

这些算法为同样级别的安全提供了在加密解密函数上低开销的可能，但它们的实际应用还不是很先进，我们仅简要地说明一下。

RSA Rivest、Shamir和Adelman（RSA）对于公开密钥密码的设计[Rivest *et al.* 1978]是基于两个大素数（大于10^{100}）的乘积，其安全性依赖于这样大的数的素因于判定在计算上非常困难，也就是说在计算上是不可行的。

尽管做了广泛的研究，还没有发现RSA有什么漏洞，它现在仍被广泛使用。下面简要叙述RSA算法。为了找到密钥对e，d：

1. 选择两个大素数，P和Q（每个都大于10^{100}），并且形成：

$$N = P \times Q$$

$$Z = (P-1) \times (Q-1)$$

2. 对于d，选择任意和Z互质的数（也就是，数d和Z除1外没有公因于）。

我们用小的素数P和Q来说明计算的过程：

279

$$P = 13,\ Q = 17 \rightarrow N = 221,\ Z = 192$$

$$d = 5$$

3. 为找出e，求出等式：

$$e \times d = 1 \bmod Z$$

也就是说，$e \times d$是在Z+1，2Z+1，3Z+1…序列中，能被d整除的最小数。

$$e \times d = 1 \bmod 192 = 1,\ 193,\ 385,\ \cdots$$

385可被d整除

$$e = 385/5 = 77$$

为了用RSA方法加密正文，明文被分成每个k位长的块，其中$2^k < N$（也就是，一个块数字上的值总是小于N的；在实际应用中，k通常在512到1024之间）。

$k = 7$，因为$2^7 = 128$

加密明文M中一个块的函数是：

$$E'(e, N, M) = M^e \bmod N$$

对于消息M，密文就是$M^m \bmod 221$

将加密正文c的一个块，解密成原明文块的过程是：

$$D'(d, N, c) = c^d \bmod N$$

Rivest、Shamir和Adelman证明了对于满足$0 \leqslant P \leqslant N$的所有$P$的值，$E'$和$D'$是互逆的（即，$E'(D'(x)) = D'(E'(x)) = x$）。

两个参数e、N可以被看做加密函数的密钥，类似地，d和N可以表示解密函数的密钥。于是我们可以写出$K_e = <e, N>$和$K_d = <d, N>$，加密函数是$E(K_e, M) = \{M\}_K$（注意这里指出了加密的消息只能由私钥K_d的所有者来解开），解密函数是$D(K_d, \{M\}_K) = M$。

值得注意的是所有的公开密钥算法都有一个潜在的弱点——因为公开密钥对于攻击者也是公开的，他们可以很容易地产生加密消息。这样他们就可以穷举任意的比特序列，将它加密后，与未知的加密消息比较，直到获得匹配为止。这种攻击，也被称为明文选择攻击。这种

攻击在确保消息都比密钥长的情况下，就失败了。所以这种类型的强制攻击其实还不如对密钥的直接攻击。

一个秘密信息的准接收者必须公布或发送<*e*, *N*>对，而保持*d*私密。对<*e*, *N*>对的公布并不损害*d*的秘密，因为想要知道*d*必须知道最初的两个素数*P*和*Q*，对此，只有对*N*进行因式分解。对于大数的因式分解（我们提到*P*和*Q*都是大于10^{100}的，于是$N>10^{200}$）即使是在高性能很高的计算机上，也是非常耗时的。1978年，Rivest等人得出结论，按照已知的最好的算法，分解一个和10^{200}差不多大的数，所花费的时间，在每秒执行100万条指令的计算机上，将超过40亿年。 280

自Rivest等人得出1978年的结论以来，诞生了很多更快的计算机和更好的分解方法。RSA社团发起了一系列对100位以上十进制数的分解挑战[www.rsasecurity.com II]。在一个分布式因特网用户联盟的努力下，已成功地分解了至多155位十进制数字，其使用的方法和前面提到的对DES的解法是相似的。写这本书的时候，155位十进制数字（约是500位的二进制数字）已被成功分解了，这使我们对512位密钥的安全性产生了怀疑。对155位十进制数字分解的过程还没有公布，但140位十进制数字在1999年2月花了2000MIPS年解出了（一个MIPS年就是由一个可以每秒完成100万个指令的处理器工作一年的运算能力）。今天一个单处理器芯片速度就可以达到200MIPS数量级，使其可以用10年的时间完成这项任务，但分布式的分解程序和并行的处理器可以将这一任务所花的时间减少到天的量级。RSA社团（拥有RSA算法的专利权）建议采取至少768位或者230位十进制数字长的密钥，才能保证长期（约为20年）的安全。一些程序中已用到了2048位的密钥。

以上这些计算都假设了现在知道的分解算法是可用的最佳算法。RSA和其他使用素数乘法作为它们的单向函数的非对称算法，在更快的分解算法发现后，必将会很脆弱。

椭圆曲线加密 已经开发和测试了一个基于椭圆曲线的性质生成公钥/私钥对的方法。详细内容可以参见Menezes为该主题写的书[Menezes 1993]。密钥来源于一个不同的数学分支，与RSA不同，椭圆曲线加密安全性不需要依靠分解大数的困难度。更短一些的密钥也可以是安全的，加密和解密所需的运算需要也小于RSA。椭圆曲线加密算法可能在将来得到更为广泛的应用，尤其是那些包含了处理资源有限的移动设备的系统。相关的数学知识包括了一些椭圆曲线非常复杂的性质，超出了本书要讨论的范围。

7.3.3 混合密码协议

公钥密码学对电子商务是很便利的，因为它不需要安全的密钥分发机制（当然还需要对公开密钥的认证，不过这并不麻烦，只需和密钥一起捆绑一个公开密钥证书就可以了）。但公钥密码的运算开销即使是对中等大小的消息加密也是巨大的，而在电子商务中一般都会遇上中等大小的消息。在大多数大规模分布式系统中，所采取的解决办法是，使用混合加密方案，其中公钥密码用来认证通信的双方并对保密密钥交互进行加密，这个密钥将用于随后所有的通信中。我们将在7.6.3节的SSL实例研究中讨论混合协议的实现。 281

7.4 数字签名

强大的数字签名是安全系统的一个重要部分。在证明某些信息的场合需要数字签名，例如为了提供可信赖的声明，这些声明将用户的身份绑定到他们的公开密钥上，或者将一些访

问权利和角色绑定到用户的身份上。

在各种各样的商业和个人交易中，数字签名的必要性勿庸置疑。文档出现伊始，手写签名就作为一种证明文件，被用来为收件人证明文档的：

- 可信性 它使收件人确信签名者特意对该文挡进行了签名，并且文档没有被其他人篡改。
- 不可遗忘性 它证明了是签名者本人而不是他人特意签名了文档。该签名不能被复制和置于其他文档上。
- 不可抵赖性 签名者不能否认他们对该文档进行了签名。

事实上，使用传统的签名不能完全获得上述想要的签名性质——由于难以检测签名伪造和拷贝，因此文档在签名后可以被篡改，有时候签名者无意中或不知情地被欺骗签名了文档——但是鉴于欺骗的难度和被查获的风险，我们乐于接受这种非理想状况。与手写签名类似，数字签名基于将一个惟一的且秘密的签名者属性绑定到文档中。在手写签名的例子中，该秘密属性即为签名者的手写体模式。

保存在存储文件或消息中的数字文档性质和那些纸质文档的性质完全不同，一般很容易生成、复制和改变数字文档。简单地将创作者的身份认证信息附加在文档后头，无论是一个文本字符串、一张照片还是一幅手写体图像，都没有任何的验证价值。

因此需要这样一种方法，它将签名者的身份认证信息绑定到代表文档的整个比特序列上，并且该操作不可撤销。这应该满足了上述的第一个需求，可信性。和手写签名一样，签名不能保证文档的日期。签名文档的接收方只知道文档在接收前已经被签名了。

至于不可抵赖性，还存在这样一个问题，该问题并非起因于手写签名。如果签名者故意泄露了他们的私钥并且随后否认了已签名的文档，他们声称由于私钥的非私有性，签名可能是其他人做的，那又当如何？在不可抵赖数字签名[Schneier 1996]的题目下，一些协议已经被设计出来解决此问题，但是它们增加了相当大的复杂性。

一个带有数字签名的文档比手写签名对伪造更具有抵抗力。但是“原始”对于数字文档意义并不大。正如我们将从我们对电子商务需求的讨论中所看到的那样，数字签名本身并不
282 能防止电子货币的两次支付——它还需要其他的措施来防范。我们现在将描述用于数字签名文档的两种技术，它们都依赖密码技术的使用，将主体的身份绑定到文档中。

数字签名 主体A可以通过使用密钥K_A加密M的副本并且将加密的信息附加到M的明文副本和A的标识上来对电子文档或消息M进行签名。因此加密的文档包括：M、A、$[M]_{K_A}$。签名可以被随后的接收文档的主体验证以确定文档是由A发起的，并且包含的内容M未被篡改。

如果使用一个保密密钥来加密文档，则只有共享该秘密的主体可以验证这个签名。但是如果使用了公开密钥密码，那么签名者用他自己的私钥加密，任何人只要拥有相应的公开密钥就可以验证该签名。这一个更好的对传统签名的模拟，它满足了更为广泛的用户需要。签名的验证过程根据用以产生签名的是保密密钥密码还是公开密钥密码而有所不同。这两种情形分别在7.4.1节和7.4.2节给予阐述。

摘要函数 摘要函数也被称为安全散列函数，用$H(M)$表示。必须仔细设计这些函数以确保对所有可能的消息对M和M'，函数值$H(M)$和$H(M')$一定不同。如果存在不同消息M和M'且$H(M)=H(M')$，那么会出现这种情况：一个不诚实的主体发送了消息M，但是当面临问题时，他可以声称他初始发送的是消息M'，并且消息一定是在传送途中被篡改了。我们将在7.4.3节讨论这些安全散列函数。

7.4.1 公开密钥数字签名

公开密钥密码特别适合于数字签名的生成，因为它相对简单且不需要文档接收者、文档签名者或任何第三方之间的通信。

*A*给消息*M*签名，*B*进行验证的方法如下（如图7-11所示）：

1. *A*产生一个密钥对K_{pub}和K_{priv}并且把公开密钥K_{pub}发布出去，放在一个大家都知道的地方。
2. *A*使用一个大家认可的安全散列函数*H*计算消息*M*的摘要*H*(*M*)，并用私钥K_{priv}加密来产生签名$S = \{H(M)\}_{K_{priv}}$。
3. *A*把已签名的消息$[M]_K$=*M*、*S*发送给*B*。
4. *B*用公开密钥K_{pub}解密*S*并且计算*M*的摘要*H*(*M*)。如果结果和解密所得的摘要相一致就说明签名是有效的。

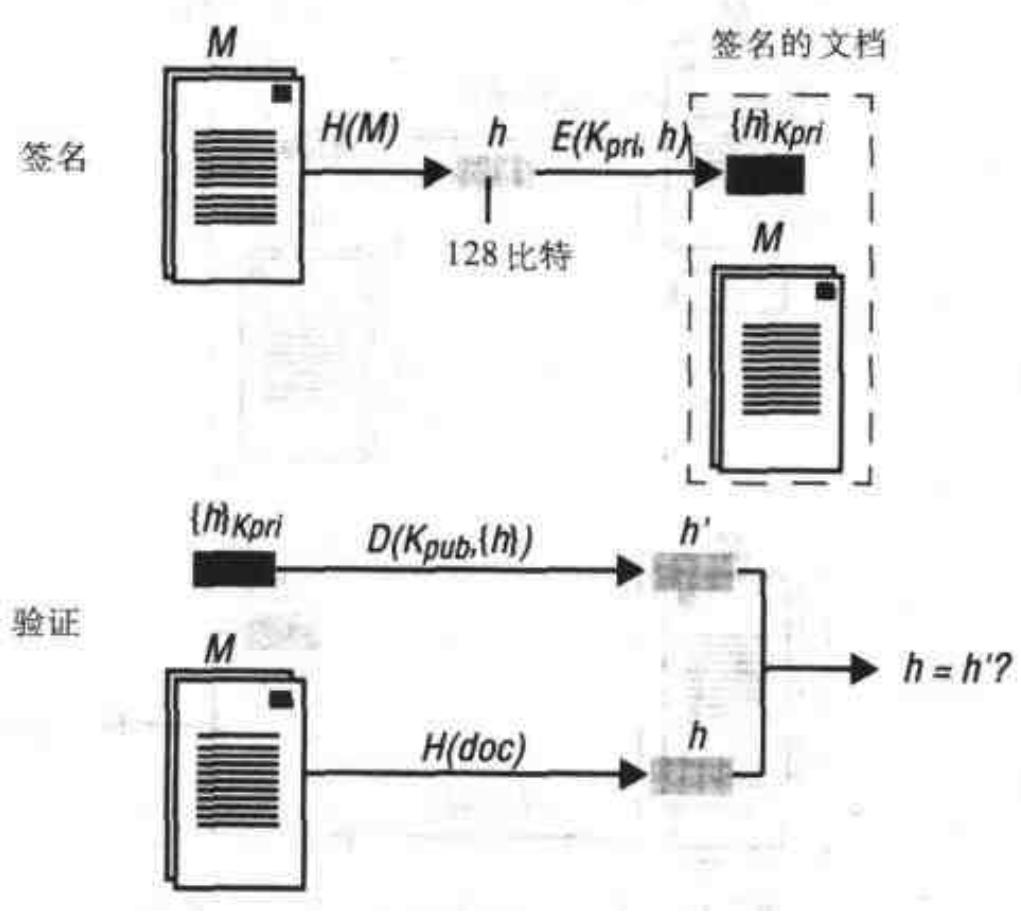


图7-11 公开密钥数字签名

RSA算法非常适合用来构造数字签名。注意到这里签名者的私钥是用来加密签名的，与当目的是秘密传输数据时接收者用公开密钥来加密数据的情形相反。对于这种差异的解释是一目了然的——一个签名必须用只有签名者知道的密钥来建立，但它可以被所有其他人验证。

7.4.2 保密密钥数字签名——MAC

没有什么技术原因可用于说明为什么一个保密密钥加密算法不应该被用来加密数字签名，但是为了验证这样的签名，密钥必须被透露出去，这会导致一些问题：

- 签名者必须安排验证方安全接收用于签名的保密密钥。
- 在若干上下文环境中以及不同的时刻，验证签名是必要的。在签名时刻，签名者可能不知道验证者的身份。为了解决这个问题，验证可以被委托给一个可信赖的第三方机构，该机构持有所有签名者的保密密钥，但是这增加了安全模型的复杂度并且要求与第三方可信机构进行安全通信。
- 透露用于签名的保密密钥是令人不快的，因为它削弱了签名的安全性——一个用它生成

的签名可以被一个密钥持有人伪造，而他并不是密钥所有者。

鉴于所有这些原因，用公开密钥法来产生和验证签名在大多数情况下提供了最便利的解决方案。

但是下面的情况是一个例外：一个安全通道被用来传输未加密的消息但是需要证实消息的真实性。因为一个安全通道在一对进程之间提供了安全通信，可以使用7.3.3节的混和方法建立共享的保密密钥并用它生成低开销的签名。这些签名被称为消息认证码（MAC）以反映它们有限的目的——它们基于一个共享的秘密，在一对主体之间认证通信。

283
l
284

一个基于共享保密密钥的低开销签名技术（如图7-12所示）可以为许多不同的目的提供充足的安全保障，我们将在下面进行阐述。这种方法依赖于安全通道，通过该通道，共享的密钥可以被分发出去：

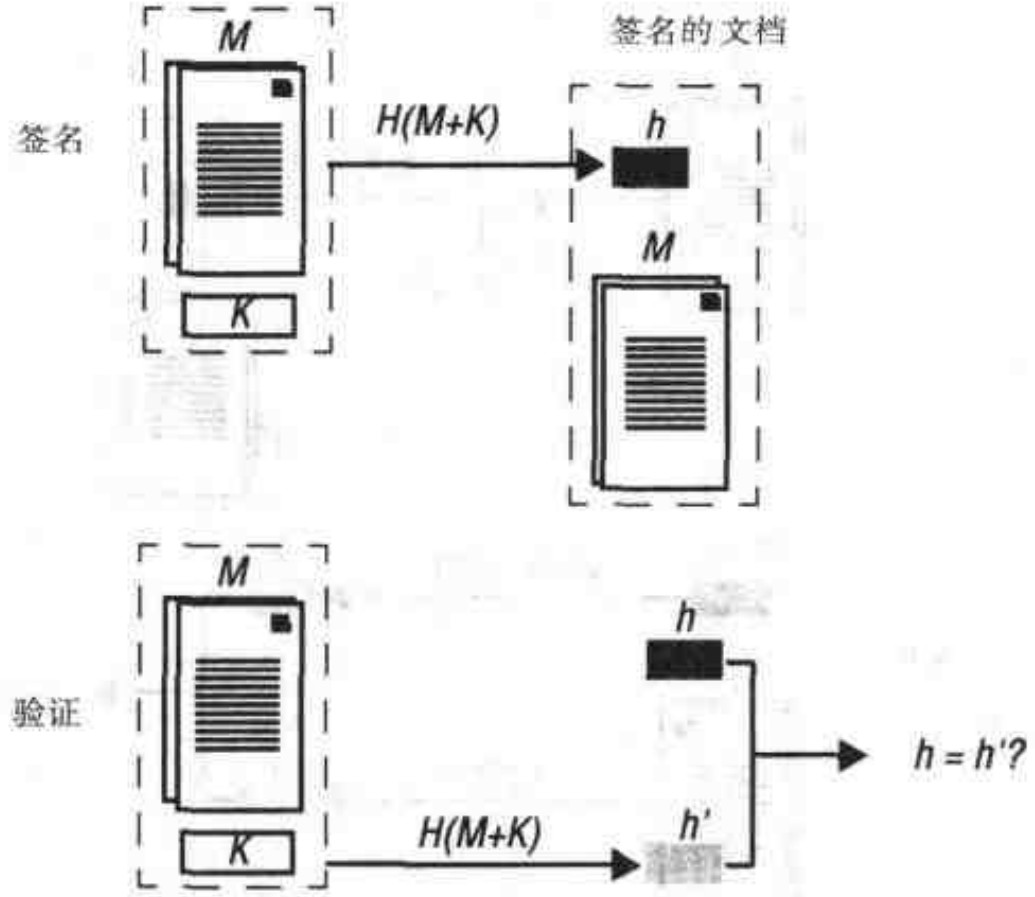


图7-12 用共享保密密钥的低开销签名

1. A产生一个随机的密钥K用以签名，并且通过安全通道将K分发给一个或多个需要鉴定接收到A的消息的主体。这些主体被信任不会泄露共享密钥。
2. 对于A希望签名的任何文档M，A将M和K连接起来，计算连接结果的摘要：$h = H(M+K)$，然后将签名好的文档$[M]_K=M$，h发给任何希望验证签名的人（摘要h是一个MAC）。由于散列函数完全模糊了K的值，因此K不会因为h的泄露面受到损害。
3. 接收者B将保密密钥K和接收到的文档M连接起来，计算摘要$h' = H(M+K)$。如果$h=h'$，那么签名即被证实有效。

虽然这种方法有上述的不足，但是由于它不需要加密，因而有着性能上的优势。（通常安全散列比对称加密快3～10倍，见7.5.1节）。7.6.3节描述的SSL安全通道协议支持MAC的广泛运用，包括这里叙述的方案。该方法也被用于7.6.4节描述的Millicent电子货币协议，那里为小金额交易保持低处理开销尤为重要。

285

7.4.3 安全摘要函数

有许多种方法可产生固定长度的位模式，这些位模式刻画一个任意长度的消息或文档。

也许最简单的方法是反复用XOR操作来组合源文档的固定长度片断。这样的一个函数经常被用在通信协议中进行错误检测，主要是用它生成一个能刻画消息的短的、定长的散列值，但是它并不适于作为数字签名方案的基础。一个安全摘要函数$h=H(M)$应该有以下性质：

1. 给定M，很容易计算h。
2. 给定h，很难算出M。
3. 给定M，很难找到其他消息M'，使得$H(M)=H(M')$。

这样的函数也称为单向散列函数，这样命名不言而喻，是基于前两个性质的。性质3要求额外的特性：即使我们知道散列函数的结果不能保证惟一（因为摘要是一个信息减损的转化过程），我们需要确定，即对给定的消息M及其产生的散列值h，攻击者也不能找到其他的具有相同散列值h的消息M'。如果攻击者可以做到这点，那么他们可以不需要知道签名密钥，就从签名文档M中拷贝签名并附加在M'上，从而伪造签名文档M'。

必须承认，经过散列后具有相同散列值的消息的集合是有限的，攻击者产生一个有意义的伪造会十分困难，但是如果有耐心他还是能办到的，所以必须进行防范。在生日攻击情况下这种可能性显著增加：

1. Alice给Bob准备两个合同版本M和M'，对Bob而言M是有利的而M'不是。
2. Alice制作了M和M'的只有几个细微差别的不同版本，例如在行尾增加空格等，两个版本的差别在视觉上难以分辨。她比较所有的M和M'的散列值，如果她发现两个是相同的，她可以进行下一步；如果未能发现，她继续产生两个文档的细微差别版本，直到两个文档产生匹配的散列值。
3. 当她获得一对有着相同散列值的文档M和M'时，她把有利的文档M给了Bob，让Bob用他的私钥进行数字签名。当Bob把签好的文档发回给Alice，她可以用和M匹配的不利的版本M'替换掉M，并且保留着从M来的签名。

如果我们的散列值有64位长，我们平均只要2^{32}个M和M'的版本就可以进行攻击。这个值太小了难以让人放心，因此我们需要使散列值至少达到128位长才足以防范这类攻击。

这种攻击根据统计学悖论的生日悖论——在给定的一个集合中找到一个匹配对的概率远远大于在其中寻找与给定的个体匹配的概率。Stallings[1999]为这种在一个n个人的集合中存在两个具有相同生日的人的概率给出了统计学的推导。结果是对于一个只有23人的集合来说机会是均等的，而我们需要一个253人的集合才有某人的生日在一个指定的日子这种相等的机会。 286

为了满足上述性质，必须小心设计安全摘要函数。使用的位层次操作和它们的先后顺序与对称密码学中的相似，但是在这种情况下操作不必保存信息，因为函数不需要是可逆的。所以安全摘要函数可以利用算术的各种方法和基于位的逻辑操作。原始文本的长度通常包含在摘要的数据里。

在实际应用中得到广泛使用的两个摘要函数是MD5算法（之所以这样命名是因为它是由Ron Rivest开发的消息摘要算法系列中的第5个）和被美国国家标准和技术研究所（NIST）用作标准的SHA（安全散列算法）。这两个算法都经过仔细测试和分析，可以充分满足可预见将来的安全需要，同时它们的实现相当高效。我们在这里只给出了简要的描述。Schneier [1996]和Mitchell等[1992]对数字签名技术和消息摘要函数给出了详细的综述。

MD5 MD5算法[Rivest 1992]共有4轮操作，输入文本以512位为一块，每一块又划分为16个32位的段，每轮对一个段应用4个非线性函数中的一个，经过4轮操作，将结果级联产生

一个128位的摘要。MD5是当前可用的最高效的算法之一。

SHA SHA[NIST 1995]是一个产生160位摘要的算法。它基于Rivest的MD4算法（与MD5算法类似），附加了一些额外操作。运行速度比MD5慢的多，但是160位的摘要值可以提供更大的安全保障以防止强行攻击和生日类型的攻击。

使用加密算法生成摘要 可以使用如7.3.1节所述的那些对称加密算法来生成一个安全摘要。在这种情况下，应该将密钥发布出去，让任何希望证实数字签名的人可以运用摘要算法进行有关的验证。加密算法用于CBC模式，其摘要是倒数第二个CBC值和最终加密块的组合结果。

7.4.4 证书标准和证书权威机构

X.509是运用最为广泛的证书标准格式[CCITT 1988b]。虽然X.509证书格式是X.500标准的一部分，用于进行全球命名和属性目录的构建[CCITT 1988a]，但是它在加密处理中通常作为一种独立证书的格式定义。我们将在第9章描述X.500的命名标准。

X.509证书的结构和内容如图7-13所示，它将一个公开密钥绑定在一个被称为主题的命名实体上。该绑定存在于签名中，这个签名被另一个称为发布者的实体发布。证书有一个有效时限，由一对日期所定义。<标志名>项指的是一个人、组织或其他有着足够上下文信息用以保证惟一性的实体的名字。在一个完整的X.500的实现中，这种上下文关系可以从命名实体所
287 在的目录层次中抽取出来，但是如果没有全局的X.500实现，这种关系只能是一个描述性的字符串。

主题	标志名、公开密钥
发布者	标志名、签名
有效时限	不早于某日期且不晚于某日期
管理信息	版本、序列号
扩展信息	

图7-13 X.509证书格式

这种格式被包含在SSL协议中运用于电子商务，它在实际的服务和客户端的公开密钥认证中得到广泛运用。某些众所周知的公司和组织已经建立并担当了证书权威机构（例如，Verisign[www.verisign.com]、CREN[www.cren.net]），其他公司和个人通过向这些组织提交符合要求的身份证明，可获得X.509公开密钥证书。这导致对任何X.509证书的一个两阶段的验证过程：

1. 从一个可信之处获得发布者（证书权威机构）的公开密钥证书。
2. 验证签名。

SPKI方法 X.509是基于标志名的全局惟一性的。但是它被认为是一个不实际的目标，它不能很好地反映当前法律和商业实践的现实[Ellison 1996]，因为个体的身份不能被看作是惟一的，而在参考其他个人和组织时是惟一的。这在使用驾驶执照或银行证明信认证一个人的名字和地址（单独的一个名字在世界范围内不可能惟一）中相当常见。这就导致了更长的验证链，因为存在许多可能的公开密钥证书的发布者，他们的签名必须通过一串验证链验证，最后验证被传给执行验证的主体所知的、且信任的某人。得到的验证结果可能更让人信服，验证链中的许多步骤可以缓存，从而在未来某些场合中缩短处理过程。

上述讨论是最近开发的简单公开密钥基础设施（SPKI）的依据（参看RFC 2693[Ellison *et al.* 1999]）。这是一个建立和管理公共证书集合的方案，它使得用逻辑推理来处理的证书链能生成派生的证书。例如，“Bob相信Alice的公开密钥是K_{Apub}”并且“Carol在Alice的密钥上信任Bob”，这就意味着“Carol相信Alice的公开密钥是K_{Apub}”。 288

7.5 密码实用学

在7.5.1节中，我们比较了前面叙述的加密算法和安全散列算法的性能。我们把加密算法和安全散列函数放在一起考虑是因为加密算法有时候被用来进行数字签名。

在7.5.2节中，我们讨论了一些围绕密码使用上的非技术问题。自从密码算法开始出现在公共领域，还没有地方对发生在此学科上的大量的政治讨论进行过公正的评判，而且对该学科的争论也没有达成明确的结论。我们的目的只是让读者了解一些正在进行的争论。

7.5.1 加密算法的性能

图7-14给出我们在本章讨论的加密算法和安全摘要函数的密钥长度（或者安全摘要函数的散列值长度）和速度。在适用的地方，我们给出两个速度度量，标有“外推速度”栏里的值是基于Schneier的数据（TEA除外，它是基于[Wheeler and Needham 1994]的数据）。在“PRB优化”栏里，我们给出的数据是基于[Preneel *et al.* 1998]发表的数据。在这两种情况下，我们已经考虑了从那时到1999年底处理器性能的提高，并对数据进行了调整。这些数据可以看作运行在PII 330MHz机器上算法性能的一个粗略估计。密钥的长度粗略地指示出了算法的强度——实际上它只能粗略指示出对密钥进行强行攻击的计算开销。密码算法的真实强度更加难以估计，依赖于算法在隐藏明文时的变换。在两个性能栏中的巨大差异可以用这样一个事实来说明，即Schneier [Schneier 1996]的数据是基于他发表的C程序源码的，没有尝试进行代码优化，而PRB的数据是为形成专利而做巨大的研究努力的结果，其中使用了汇编语言对算法的实现进行了优化。Preneel 等[1998]对主要的对称算法的强度和性能展开了讨论。

	密钥长度/散列长度（比特）	外推速度（KBps）	PRB优化（KBps）
TEA	128	700	-
DES	56	350	7746
Triple-DES	112	120	2842
IDEA	128	700	4469
RSA	512	7	-
RSA	2048	1	-
MD5	128	1740	62425
SHA	160	750	25162

图7-14 加密和安全摘要算法的性能

我们这里只给出了软件的性能数据，目前正在进行大量的工作以产生高性能、低开销的硬件上（单芯片）实现，有望产生更高的性能。

7.5.2 密码学的应用和政治障碍

上述算法均在1980年到1990年之间出现，期间计算机网络开始用于商业用途，而计算机

网络缺乏安全越来越明显地成为主要问题。正如我们在本章简介中所述，美国政府强硬抵制密码软件的出现。有两个原因，其一，认为美国国家安全局（NSA）有这样一个政策，该政策限制其他国家可用的密码长度只能达到这样一个水平，即国家安全局可以基于军事情报的目的破解任何秘密通信；其二，美国联邦调查局（FBI）为执行法律的目的，要确保它的机构拥有访问所有在美的私有组织和个人的密钥的特权。

在美国，密码软件被分类为军需品，严格限制出口。其他国家，特别是美国的盟国，也采用类似的做法，在某些情况下甚至更为严厉。而政治家以及一般公众就什么是密码软件和它潜在的非军事化应用一无所知，这使得问题更加复杂化。来自美国软件公司的抗议认为这种限制抑制了诸如浏览器软件的出口，该出口限制最终确定为允许使用不超过40位密钥（不太强的密码）的软件代码出口。

出口限制可能已经阻碍了电子商务的发展，但是它们在防范密码技术的散布和保持密码软件不被其他国家所控制上并不是特别有效，因为在美国国内外有许多程序员热衷于能够实现和分发密码代码。当前的情形是，实现了绝大多数主流密码算法的软件已在全世界流行多年了，包括出版物[Schneier 1996]和在线资料、商业和免费软件版本[www.rsasecurity.com, cryptography.org, privacy.nb.ca, www.openssl.org]。

一个例子是称为PGP（Pretty Good Privacy）的程序[Garfinkel 1994, Zimmermann 1995]，最早是由Philip Zimmermann开发的，并由他和其他人分发出去。这是一个技术和政治运动的一部分，它要求确保密码方法不被美国政府所控制。PGP已经被开发出来并被分发出去，目的是使所有计算机用户都可以在他们的通信中使用公开密钥密码算法，从而享受由此带来的私密性和完整性。PGP代表用户生成并管理公开密钥和私钥，它使用RSA公开密钥加密算法进行认证并把保密密钥传送给通信伙伴，并使用IDEA或者3DES保密密钥加密算法来加密邮件消息和其他文档（PGP刚开始开发时，DES算法被美国政府控制）。PGP有免费的和商业版本。它经由不同的分发站点发布给北美用户[www.pgp.com]和世界上其他地区的用户[International PGP]以（完全合法地）挫败美国的出口规则。

289～290

美国政府最终认识到NSA的观点是毫无作用的，认识到这带给美国计算机工业的危害（无法在全球范围内出售网络浏览器、分布式操作系统和其他许多产品的安全版本）。2000年1月，美国政府引入了一个新的政策法规[www.bxa.doc.gov]，目的是允许美国软件供货商出口结合有很强加密功能的软件产品，希望以此简化国际密码软件市场。新的法规允许出口结合着最多达64位加密密钥，以及最多达1024位的用于签名和交换密钥的公开密钥的软件产品。法规要求政府“审查”出口的软件，但是同时也允许密码源代码的非限制出口，它明显地取消了对密钥长度的限制。

其他的政治动机，目的是通过引入立法来维持对密码的使用进行控制，立法坚持软件必须包含只对政府执法部门和安全机构有效的入口或者后门。这些建议源自这样的设想，因为秘密的信道可以被各种各样犯罪分子使用。历史上，美国政府曾经截取、分析公众的通信信息，密码从根本上改变了这种状况。但是这些立法提案妨碍了密码学的使用，它们遭到那些关心他们自身的隐私权被冲击的公民和公民自由团体的强烈反对。迄今为止，这些立法提案没有一个被采纳，但是政治上的努力也是持久的，最终将不可避免地导致引入一个合法的使用密码的框架。

7.6 实例研究：Needham-Schroeder、Kerberos、SSL和Millicent

最初由Needham和Schroeder[1978]发表的认证协议是许多安全技术的核心，我们将在7.6.1节详细说明。其中最为重要的保密密钥认证协议的应用是Kerberos系统[Neumann and T'so 1994]，这是我们第二个案例的主题（7.6.2节）。Kerberos被设计用于为网络上客户和服务器间提供认证服务，从而形成一个单一的管理域（企业内部网）。

我们还有两个描述应用级安全协议的案例，它们对于电子商务的运用十分重要。第一个应用级案例涉及安全套接字（SSL）协议，它被专门设计用于满足安全电子交易的需要的，该协议目前被大多数Web浏览器和服务器所支持，并被大多数Web电子商务交易采用。我们的第二个案例描述了Millicent协议，它被专门设计用于满足微型交易的支付需要。 291

7.6.1 Needham-Schroeder认证协议

这里描述的协议是为了满足在网络上安全管理密钥（和口令）的需要而开发的。在发表这项工作[Needham and Schroeder 1978]的时候，网络文件服务刚刚出现，在本地网中存在着对更好的安全管理方法的迫切需要。

在管理型网络中，需要能由安全保密密钥服务以质询（见7.2.2节）的形式发布会话密钥来满足上述需求，这就是Needham和Schroeder开发保密密钥协议的目的。在同一篇论文中，Needham和Schroeder也陈述了一种基于使用公开密钥认证和密钥分发的协议，该协议不依赖已有的保密密钥服务器，因此更适合于在因特网这样有着许多独立管理域的网络中使用。在这里我们不准备描述公开密钥版本，但是在7.6.3节叙述的SSL协议是它的一个变种。

Needham和Schroeder提出了一个认证和密钥分发的解决方案，它基于一个给客户提供保密密钥的认证服务器。认证服务器的工作是为一对进程提供一个安全的方式以获得共享密钥。为了做到这点，它必须使用已经被加密的消息与客户通信。

包含保密密钥的Needham-Schroeder认证协议 在他们的模型中，一个进程代表一个主体A，A希望启动与代表主体B的其他进程的安全通信，A进程可以为这个目的获得一个密钥。这个协议是对任意两个进程A和B来说的，但是在客户－服务器系统中，A可能是一个对某个服务器B发起一系列请求的客户。提供给A的密钥有两种形式，一种是让A用来加密传递给B的消息的，另一种可以安全地传递给B（后一种用一个B可知而A不知道的密钥加密，因此B可以进行解密并且该密钥在传输过程中未受到损害）。

认证服务器S维护一张表，为系统所知的每个主体保存一个名字和一个保密密钥。保密密钥只用于认证客户到认证服务器的进程，并用于在客户进程和认证服务器之间进行安全消息传输。该密钥从不泄露给第三方，在生成密钥后它在网络上最多传送一次（在理想情况下，一个密钥应该总是通过其他途径传送，例如书面形式或口头形式，避免密钥在网络上暴露）。一个保密密钥与在集中式系统中用于认证用户的口令等价。对于主体人，认证服务持有的名字是他们的“用户名”，保密密钥是他们的口令，用户名和口令都是由用户在向代表他们的客户进程发出请求时由用户提供的。

这个协议是基于认证服务器产生和传送票证的。一个票证是一个加密了的消息，它包含用于在A和B之间通信的保密密钥。我们将Needham和Schroeder的保密密钥协议制成表格如图7-15所示。其中S是认证服务器。

消息头	消　息	注　释
1. A →S:	A, B, N_A	A请求S提供一个用于与B通信的密钥
2. S→A:	$\{N_A, B, K_{AB}, \{K_{AB}, A\}_{K_B}\}_{K_A}$	S返回用A的保密密钥加密的消息，消息含有用B的保密密钥加密的新生成的密钥K_{AB}和一个“票证”。当前时刻N_A说明该消息是响应前一个消息的。由于只有S知道A的保密密钥，所以A相信是S发送了消息
3. A→B:	$\{K_{AB}, A\}_{K_B}$	A将“票证”发送给B
4. B →A:	$\{N_B\}_{K_{AB}}$	B解密票证，并使用新的密钥K_{AB}来加密另一个当前时间N_B
5. A→B:	$\{N_B - 1\}_{K_{AB}}$	A通过返回一致的当前时间N_B的某种形式给B，证明它是前一个消息的发送者

图7-15　Needham-Schroeder保密密钥认证协议

N_A和N_B是填充值。填充值是一个整数值，它被加到消息里以说明该消息是新近产生的。填充值只被使用一次，在需要的时候生成。例如，填充值可以是一个序列号或者是读取的发送方时间。

如果成功完成该协议，那么A和B都可以确定任何所收到的用K_{AB}加密的消息是来自对方的，任何用K_{AB}加密的发送给对方的消息只能被对方或S（S被认为是可信任的）所理解，这是因为被传送的带有K_{AB}的消息是用A或B的保密密钥加密的。

该协议存在一个不足之处，因为B没有理由相信消息3是新近产生的。入侵者如果获得密钥K_{AB}并且复制了票证和认证者消息$\{K_{AB}, A\}_{K_B}$（这些信息可能由于疏忽或以A的权限运行客户程序运行错误而被暴露），他就可以假扮A，并使用它们发起和B的信息交换。在现今的术语中，Needham和Schroeder的威胁列表中没有包括这种可能性，但是由于这种攻击会引起一个旧的密钥K_{AB}值受到损害，因此多数观点认为应该包括这种可能性。通过给消息3增加填充值或时间戳可以弥补这个不足，所以消息变成为：$\{K_{AB}, A, t\}_{K_{Bpub}}$。B解密这个消息并检查时间戳$t$是否是新近的，这就是Kerberos采取的解决方案。

7.6.2 Kerberos

Kerberos是MIT大学[Steiner *et al.* 1988]在20世纪80年代开发出来的，目的是为MIT校园网和其他企业内部网提供一系列认证和安全设施。根据用户的经验和反馈，Kerberos协议经历了几个版本。下面要描述的是第5版，它沿用因特网的标准途径（参见RFC1510[Kohl and Neuman 1993]），现今被许多公司和大学使用。Kerberos的实现源码可以从MIT[web.mit.edu]获得。OSF的分布式计算环境（DCE）[OSF 1997]和微软[www.microsoft.com II]的Windows 2000操作系统都包含Kerberos的实现，并把它作为默认的身份验证服务。扩展Kerberos的提议认为可以利用公开密钥证书来进行主体的初始认证。（见图7-16步骤A）[Neuman *et al.* 1999]。

292
~
293

图7-16给出了Kerberos系统的体系结构，Kerberos处理3类安全对象：

- 票证　Kerberos票证授予服务给每个客户发一张标记，该标记送给一个特殊的服务器，证实Kerberos最近已经认证了发送者，票证包括过期时间和新生成的会话密钥供客户和服务器使用。
- 认证　由客户构造的一个标记，将它送给服务器，证明用户身份以及当前与服务器的通

信。一个认证器仅可以使用一次，它包含客户的名字和时间戳，并用恰当的会话密钥加密。

- 会话密钥　会话密钥是由Kerberos随机产生的，在与某个服务器通信时发给客户使用。对于与服务器进行的所有通信，并非都要强制加密；会话密钥就是用来对要求加密的、服务器之间的通信进行加密，也用来对所有认证器加密。

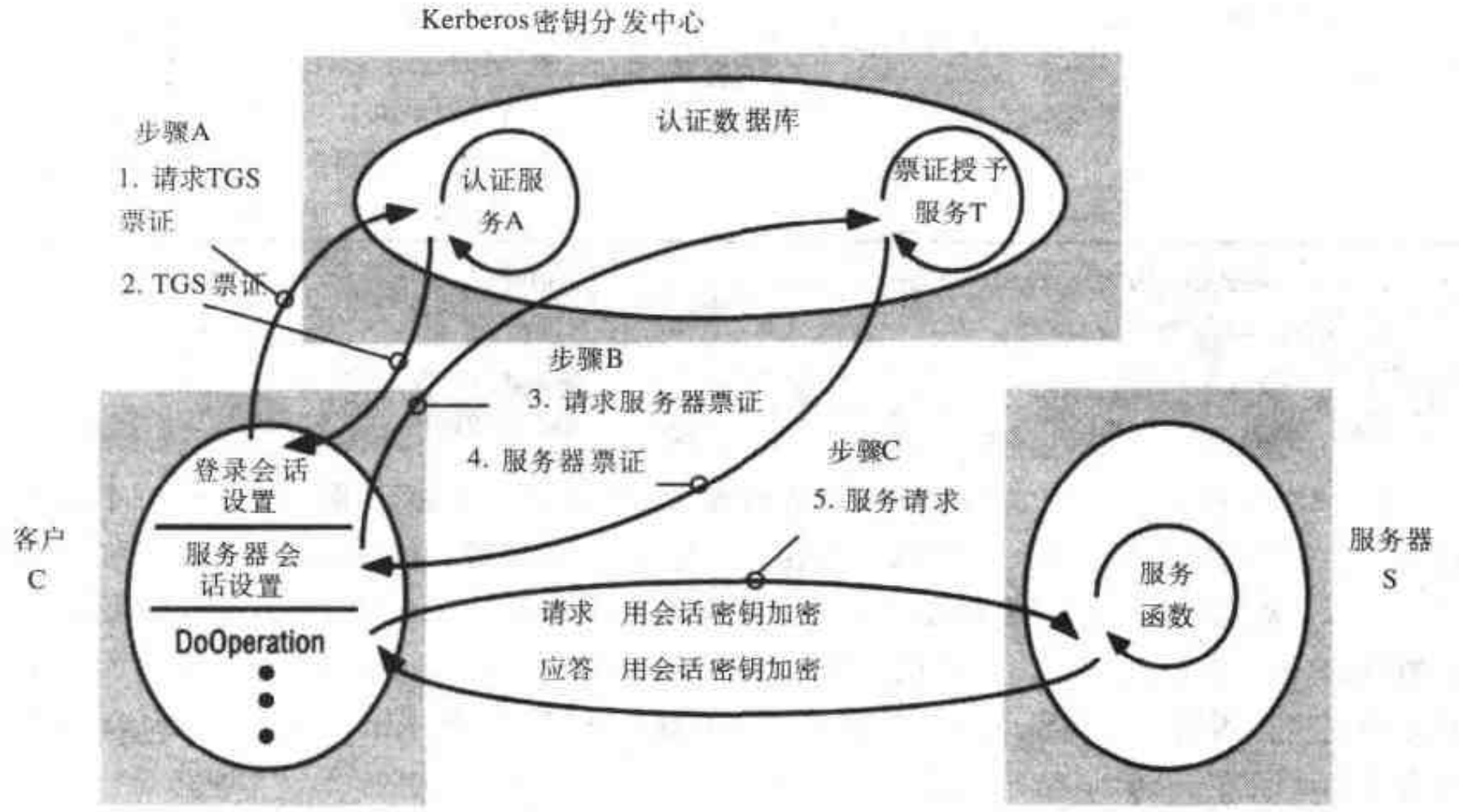


图7-16　Kerberos系统体系结构

客户进程对它们所使用的每个服务器都必须有票证和会话密钥。对客户－服务器系统的每次交互都提供新票证和密钥是不切实际的，因此大多数票证允许让客户在几小时内使用以同特定服务器进行交互，直至到期为止。

一个Kerberos服务器是一个密钥分发中心（KDC）。每个KDC提供认证服务（AS）和票证授予服务（TGS）。用户登录时，AS用网络安全的口令认证用户，然后给代表用户的客户进程提供一张能授予票证的票证和用来与TGS通信的会话密钥。这样，一个客户进程及其子进程可以用授予票证的票证从TGS中获取用于指定服务的票证和会话密钥。 294

Needham-Schroeder协议与Kerberos协议很接近，用时间值（表示日期和时间的整数）表示当前时间。这有两个目的：

- 防止重播从网络中截取的旧消息，或复用在授权用户已退出登录的机器内存中发现的旧票证（在Needham-Schroeder协议中用当前时间达到此目的）。
- 应用票证的生命期，使系统能够在用户不再是系统的授权用户时收回他们的权利。

下面我们详细描述Kerberos协议，所用记号见下面的阴影部分。首先，我们描述客户为访问TGS而获得票证和会话密钥的协议。

Kerberos票证有固定的有效期限：从t_1开始，到t_2结束。客户C访问服务器S的票证，其形式为：$\{C, S, t_1, t_2, K_{CS}\}_{K_S}$，记做$\{ticket(C, S)\}_{K_S}$，客户名包含在票证中，以免被冒充者使用。图7-16中的步骤和消息号对应于下面描述栏中的内容，注意消息1没有被加密，也不含

C的口令。它包含当前时间值，用于检查应答的有效性。

A 每次登录时，将获得Kerberos会话密钥和TGS票证		
消息头	消　息	注　释
1.C→A: 请求TGS票证	C, T, n	客户C请求Kerberos认证服务器A提供与票证授予服务T通信的票证
2 A→C: TGS会话密钥和票证	$\{K_{CT}, n\}_{K_C}$ $\{ticket(C,T)\}_{K_T}$ 包含 C, T, t_1, t_2, K_{CT}	A返回一条消息，它包含A的保密密钥加密的票证和C要用的会话密钥（与T一起使用）。当前时间n是用K_c加密的，它表明：消息来自消息1的接收者，他必须知道K_c

A：Kerberos认证服务名　　n：当前时间
T：Kerberos票证授予服务名　　t：时间戳
C：客户名　　t_1：票证有效起始时间
295　　t_2：票证有效终止时间

消息2有时称为“质询”，因为它告诉请求者的信息是只有知道C的保密密钥K_c后才有用的信息。冒充者企图靠发消息1来模仿C，但由于无法对消息2解密，他没法继续下去。对于那些用户主体，K_c是用用户的口令拼凑出来的。客户进程会提示用户键入口令，并试图用该口令对消息2解密。如果用户给出正确的口令，客户进程就能获得会话密钥K_{CT}和票证授予服务的有效票证了；否则，它获得无意义的信息。服务器有它们自己的保密密钥，只有有关的服务器进程和认证服务器知道这些密钥。

从认证服务获得有效票证后，客户C可以用它与票证授予服务T通信，以多次获得其他服务器票证，直至票证超期。因此，为了获得任一服务器S的票证，C构造一个用K_{CT}加密的认证器，其形式为：$\{C, t\}_{K_{CT}}$，记作：$\{auth(C)\}_{K_{CT}}$，然后向T发请求：

B. 每次客户服务器会话时，将为服务器S获得票证		
3.C→T: 请求服务S的票证	$\{auth(C)\}_{K_{CT}}$, $\{ticket(C,T)\}_{K_T}, S, n$	C请求T提供与另一服务器S通信的票证
4.T→C: 服务票证	$\{K_{CS}, n\}_{K_{CT}}, \{ticket(C,S)\}_{K_S}$	T检查票证。若票证有效，T就生成新的随机会话密钥K_{cs}，与用服务器保密密钥K_s中加密的S的票证一起返回

然后C开始向服务器S发请求消息：

C. 发送一个带有票证的服务器请求		
5.C→S:服务请求	$\{auth(C)\}_{K_{CS}}, \{ticket(C,S)\}_{K_S}$, $request, n$	C向S发票证，附上为C新生成的认证器及请求。若要求数据保密，则用K_{cs}加密该请求

为了让客户确信服务器的真实性，S应向C返回一个当前时间n（为减少请求的消息数，可以把它包含在含有服务器“应答”的消息中）：

D. 认证服务（可选）		
6. S→C:服务器认证	$\{n\}_{K_{CS}}$	（可选）：S向C发当前时间n，n用K_{cs}加密

Kerberos的应用 Kerberos是为MIT的Athena项目开发的，是面向大学生教育的校园网计算设施，该校园网拥有许多工作站和服务器，为5000多位用户提供服务。运行环境中，客户、网络和提供网络服务的机器的安全性都不可信赖——例如，未对工作站进行保护以防止安装用户开发的系统软件，服务器（除了Kerberos服务器）缺乏必要的安全保障用以防止用软件配置进行物理干扰。 296

Kerberos在Athena系统中提供了所有的安全保护。它用于认证用户和其他主体。大多数运行在网络上的服务器进行了扩展，在每个客户－服务器交互开始时要求客户提供票证，包括文件存储（NFS和Andrew文件系统）、电子邮件、远程登录和打印。用户的密码只有用户本身和Kerberos认证服务器知道。服务拥有的保密密钥只为Kerberos和提供服务的服务器所知。

我们将描述用Kerberos来进行用户登录认证的方式。如何使用Kerberos来保护NFS文件服务将在第8章进行描述。

用Kerberos登录 当用户登录到工作站时，登录程序将用户名发送给Kerberos认证服务，如果认证服务认可用户名，则它返回用该用户的口令加密的会话密钥、当前时间和用于票证授予服务的票证。登录程序尝试用用户键入的口令解密会话密钥和当前时间。如果密码正确，登录程序即可获得会话密钥和当前时间，它检查当前时间，并保存好会话密钥和票证以备随后与票证授予服务通信时使用。这时，登录程序可以从内存中删除用户的密码，因为票证现在可以用以认证该用户。在这台工作站上用户的登录会话即开始了。注意，用户的口令从来不暴露在可能被监听的网络上——它只被保存在工作站上，一旦登录立刻从内存中删除。

通过Kerberos访问服务器 运行在工作站上的程序一旦需要访问一个新的服务，它就从票证授予服务请求该服务的票证。例如，当一个UNIX用户希望登录到一个远程计算机上，在用户的工作站上的*rlogin*命令程序从Kerberos票证授予服务获得票证用来访问远程计算机的*rlogind*网络服务。在用户希望登录时，*rlogin*命令程序响应远程机器处的*rlogind*进程的要求，发送票证和一个新的认证器。*rlogind*程序使用*rlogin*服务的保密密钥解密票证，并检查票证的有效性（即票证未过期）。服务器必须小心地把它们的保密密钥存储在入侵者难以达到的地方。

然后*rlogind*程序使用包含在票证中的会话密钥解密认证器并检查认证器是否是新近产生的（认证器只能被使用一次）。一旦*rlogind*程序确信票证和认证器都是有效的，它就不再需要检查用户的名字和口令，因为*rlogind*程序已经知道用户的身份，这时一个远程用户的登录会话就被建立起来了。

Kerberos实现 Kerberos可以作为一个在安全机器上运行的服务器来实现。提供一些库供客户应用程序和服务程序使用。也可以采用DES加密算法，不过这是作为独立模块实现的，可以容易地被替换掉。 297

Kerberos服务是可伸缩的——它将世界分成不同的认证区域，称为域，每个域有自己的Kerberos服务器。多数主体仅在一个域中登记，但Kerberos的票证授予服务器（TGS）可在所有域中登记。通过本地TGS，主体可以在其他域中的服务器上认证自己。

在一个域中，可以有几个认证服务器，它们全都有同一个认证数据库的副本。认证数据

库的复制采用一种简单的主从技术。由Kerberos数据库管理服务（KDBM）负责更新主副本，KDBM只在主机上运行。KDBM处理用户改变口令的请求，以及系统管理员增删主体和改变口令的请求。

为了使这种机制对用户透明，TGS票证的生命期应像可能最长的登录会话一样长，因为使用过期的票证会导致拒绝服务请求，惟一的补救方法就是让用户重新认证登录会话，然后为所有使用中的服务请求新的服务器票证。在实际应用中，一般用12小时作为服务器的票证生命期。

对Kerberos的评价 上面描述的Kerberos第5版包含了针对早期版本[Bellovin and Merritt 1990, Burrows *et al.* 1990]批评的一些改进。对Kerberos第4版中最重要的批评是认证器中的当前时间是用时间戳来实现的，对认证器的保护以防止重播取决于客户和服务器时钟之间的松散同步。而且，若使用同步协议使客户和服务器时钟松散同步，那么同步协议本身也必须安全，能防范安全攻击。有关时钟同步协议的内容请参考第10章。

Kerberos第5版定义的协议允许用时间戳或者序列号实现认证器中的当前时间，无论用哪种方法，都要求它们是惟一的，并且服务器应该保留最近收到的每个客户的当前时间，以便检查有没有重播它们。这种要求实现起来很不方便，并且在服务器出错时难以得到保证。Kehne等[1992]已经公布了一个不依赖同步时钟的Kerberos协议的改进建议。

Kerberos的安全性依赖于有限的会话生命期的——TGS票证的有效期通常只有几个小时。这个期限必须选得足够长，以避免服务中断造成的不便，同时又必须足够短，以确保撤销登记的用户或降级的用户不会继续长期使用资源。这可能会给商业应用带来困难，因为要求用户在交互过程中的任一点提供新的认证细节可能会妨碍应用。

7.6.3 使用安全套接字确保电子交易安全

安全套接字层协议（SSL）最初是由Netscape公司[Netscape 1996]开发的，它提出了一种
298 标准专门用于满足上述需要。SSL的扩展版本传输层安全协议（TLS）已经被采用作为因特网标准，具体描述参见RFC 2246 [Dierk and Allen 1999]。大多数浏览器都支持SSL协议，它被广泛应用于因特网电子商务。其主要特性如下：

协商加密和认证算法 在一个开放的网络中，我们不应该认为所有的人都使用相同的客户软件，也不能认为所有的客户和服务器软件包含了特定的加密算法。实际上，一些国家的法律试图限制仅在这些国家使用某些加密算法。SSL的设计，可以在连接的两端初始化握手通信时，在进程间协商加密和认证的算法。因此可能出现通信的双方没有足够的公共算法导致连接尝试失败的情况。

自举安全通信 为了满足安全通信的要求而不需通过早先协商或第三方的帮助，可以用类似于前面提过的混合机制协议建立安全通道。使用未加密的通信进行初始化交换，然后使用公开密钥密码，一旦建立共享保密密钥，就可以转换到保密密钥密码学上来。每个转换都是可选的，都通过协商进行。

安全通道是完全可配置的，它允许对每个方向上的通信进行加密和认证（但是不要求这么做），这使得计算资源不必执行不必要的加密操作而被消耗掉。

SSL协议的细节已经被公布并标准化了，有一些软件库和工具包能够支持它[Hirsch 1997, www.openssl.org]，其中一些是在公众领域里。SSL已被结合到许多应用软件中，其安全性也

经过独立审核得到证实。

SSL由两层组成（如图7-17所示）。一层是SSL记录协议层，该层实现了一个安全通道，用来加密和认证通过任何面向连接的协议传输的消息；另一层是握手层，包含SSL握手协议和两个其他相关协议，它在客户和服务器之间建立并维护一个SSL会话（即一个安全通道）。这两层通常都是用客户和服务器应用层的软件库实现的。SSL记录协议是一个会话层协议，可以用来在有安全性、完整性和真实性保证的进程之间透明地传送应用层数据。这些即是我们在安全模型（参见2.3.3节）中为安全通道详细说明的性质，但是在SSL中是可选的，通信伙伴可以选择是否在每个方向上部署消息的解密和认证。每个安全会话被赋予一个标识符，每个通信伙伴可以在缓存中存储会话标识符以备后续使用，当要求与相同伙伴进行其他安全会话时，它避免了建立新会话的系统开销。

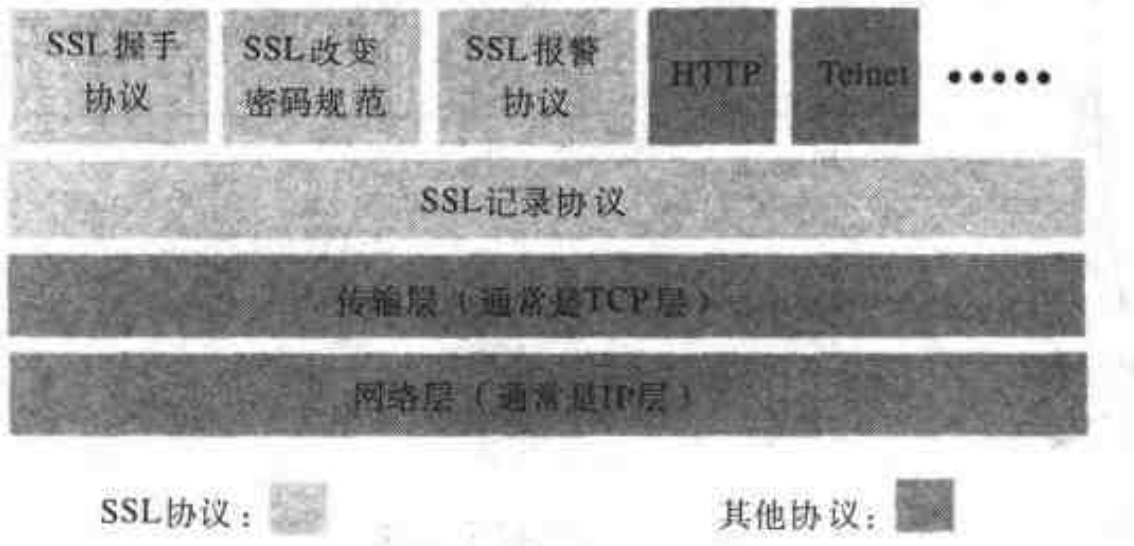


图7-17 SSL协议栈[⊖]

SSL被广泛使用在现有应用层协议之下增加一个安全通信层。该协议最早发布于1994年，经修订整合后形成了两个后续版本。SSL 3.0是标准化提案的主体[Netscape 1996]。它可能最广泛地应用于因特网商务和其他安全敏感的应用中，以保护HTTP交互。几乎所有Web浏览器和Web服务器都实现了SSL：它通过在URL中使用协议前缀https，在浏览器和Web服务器间建立起一个SSL安全通道。它也被广泛地部署以提供Telnet、FTP和许多其他应用协议的安全实现。对于那些要求安全通道的应用，SSL是事实上的标准，它通过提供CORBA和Java的API，为商业和公众领域提供了多种可用的实现选择。

SSL握手协议如图7-18所示。握手操作是在一个已建立的连接上进行的。它通过交换一致的选项和参数来建立SSL会话，这些选项和参数是执行加密和认证所需要的。握手序列根据是否需要客户和服务器的认证而有所不同。握手协议也可以在迟些时候为改变一个安全通道的规范时调用，例如，在通信开始时可能只用消息认证码来认证消息。在迟些时候，可以增加加密。它的实现是通过利用现有的通道，再次执行握手协议进行协商从而获得一个新的密码规范。

用SSL初始化握手容易受到7.2.2节场景3中所述的“中间人”攻击。为了预防这种情况，用来验证第一个接收到证书的公开密钥可以通过一个单独的通道传送——例如，经由CD-ROM发行的浏览器和其他因特网软件可以包括一些著名的证书权威机构的公开密钥。另一个著名

⊖ 本小节的图表基于Hirsch[1997]中的图表，其使用得到Frederick Hirsch的许可。

服务的客户防范措施，是根据在它的公开密钥证书中包含了服务的域名——客户只能用来自该域名对应IP地址的服务。

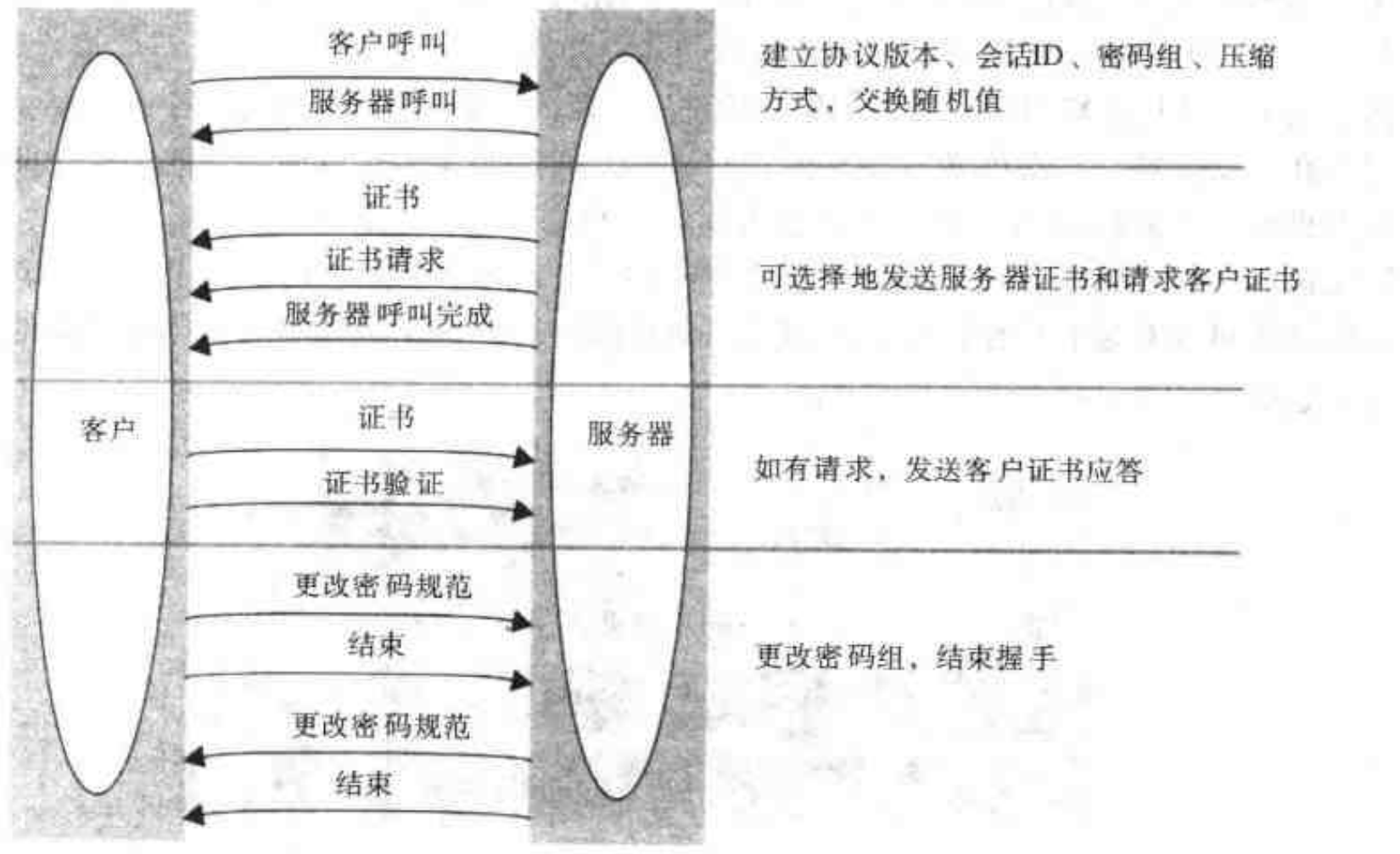


图7-18 SSL握手协议

对将使用的密码函数，SSL支持多种选项。它们全体被称作密码组。一个密码组为图7-19所示的每个特性提供单一的选项。

在客户和服务器上预先装了许多带有标准标识符的流行的密码组。在握手时，服务器为客户提供了可用的密码组标识符清单，客户选择其中的一个（如果没有匹配选项，则给出错误指示）。在这个阶段，它们也就压缩方法（可选的）和CBC块加密函数（见7.3节）的随机起始值达成一致。

下一步，通信双方按照X.509格式交换签名的公开密钥证书进行互相认证（可选）。这些证书可能是从一个公开密钥权威机构获得的，或者可以为此目的简单地暂时生成。在任何情况下，至少有一个公开密钥必须是在握手的下一个阶段可用的。

随后通信一方生成一个前导的密文，用公开密钥加密后发送给另一方。前导密文是一个大随机数，通信双方都使用这个数生成用来加密传送数据的两个会话密钥（称为写密钥）和用以消息认证的消息认证密文。当所有这些完成后，一个安全会话就开始了。这是由通信双方交换的更改密码规范（ChangeCipherSpec）消息触发的。随后是结束（Finished）消息。一旦交换了结束消息，所有进一步的通信就可以根据所选的密码组连同议定的密钥进行加密和签名。

组　件	描　述	例　子
密钥交换方式	用来交换一个会话密钥的方法	带公开密钥证书的RSA
数据传输密码	用于数据加密的块密码或流密码	IDEA
消息摘要函数	用于创建消息认证码（MAC）	SHA

图7-19 SSL握手配置选项

图7-20 显示出记录协议的操作。一个要传输的消息最初被切片为大小便于处理的块，然后可选择地压缩这些块。严格来说，压缩并不是安全通信的一个特性，但是由于一个压缩算法可以有效地共享加密和数字签名算法处理大量数据的工作，因此在这里提供了压缩选项。换句话说，数据转换管道可以在SSL记录层中建立，由SSL记录层执行所有转换，这种转换比独立转换更为有效。

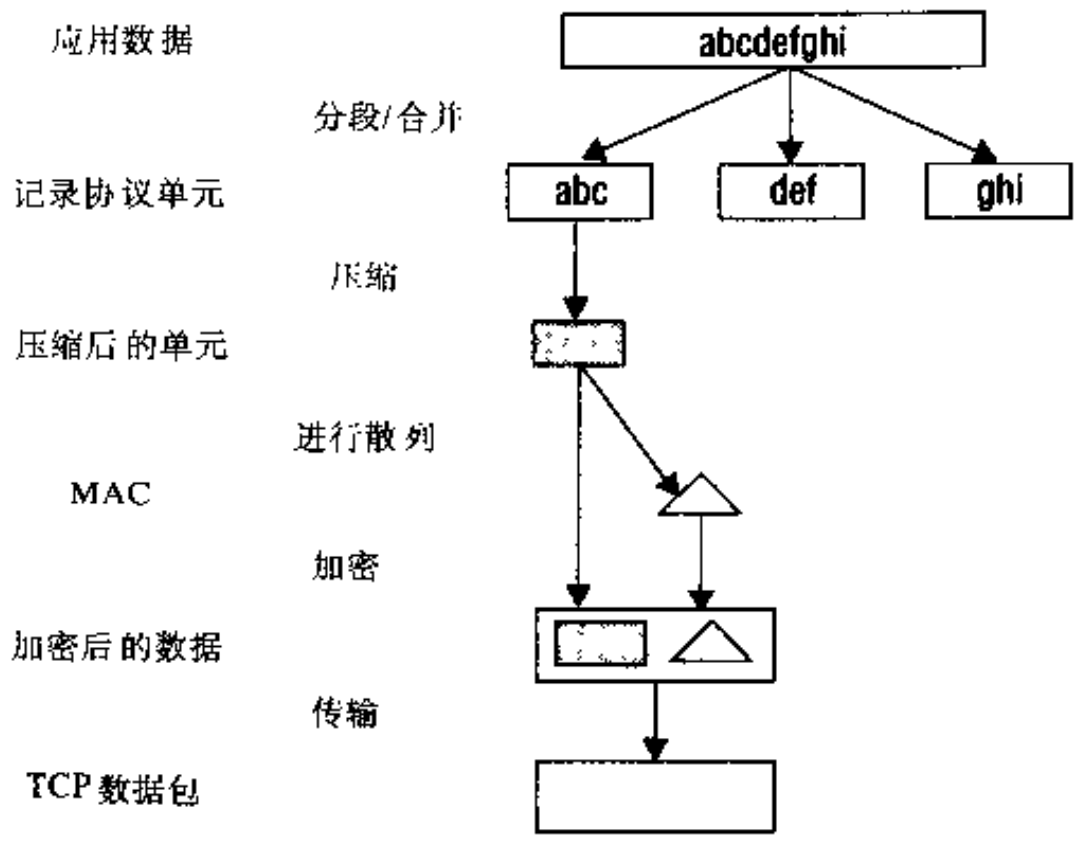


图7-20 SSL记录协议

加密和消息认证（MAC）转换使用了经协商后的密码组中指定的算法，正如7.3.1节和7.4.2节所述的那样。最后通过相关的TCP连接，将签名和加密后的数据块传送给另一方，接收方执行逆向转换，生成原始数据块。

小结 SSL提供了一个实用的混合加密方案的实现，它能进行认证和基于公开密钥进行密钥交换。因为在握手中协商密码，所以它不依赖于任何专门的算法，也不依赖于在会话建立时的任何安全服务。惟一需要的是权威机构发布的通信双方认可的公开密钥证书。

出于SSL协议及其参考实现的公布[Netscape 1996]，它成为评论和争辩的主题。对早期的设计已经进行了一些修正，作为一种有价值的标准，它得到了广泛的认可。现在SSL已经被集成到大多数的Web浏览器和Web服务器中，也被应用于诸如安全Telnet、FTP等的其他应用中。在商业和公众领域[www.rsasecurity.com，Hirsch 1997，www.openssl.org]中，有众多的程序库和浏览器插件可供使用。

299 ~ 302

7.6.4 小额电子交易：Millicent协议

为了支持小额服务的购买，访问价格在0.1美分到100美分之间的电子资源，例如，为访问网页、传输电子邮件或因特网电话而向用户收费，需要一个高效、便利的因特网安全支付系统。大多数现有的支付方案如信用卡交易代价过高。

这里我们描述Millicent方案[Glassman *et al.* 1995]提供的解决方案。该方案使用了基于保密密钥的数字签名的简单形式来降低计算开销。只有涉及隐私时才使用加密。

现有支付系统及其缺陷 Millicent协议的创造者总结了下列当前可用的支付方式的缺点。

所有这些方式都要求密码安全保护，它们的计算和通信开销差别很大，但是在所有情形下都相当昂贵。

信用卡　使用信用卡进行小交易的问题是和颁卡公司的中央系统必须进行的交互的交易代价十分高昂，当引入安全电子交易和给用户的声明等其他特点时，代价剧增。

顾客和供货商保持账号　顾客需要在开始交易前同供货商建立一个账号。虽然交易代价低廉，但是初始开销阻碍了临时交易。一个供货商必须在相当长的时间内维护顾客在账号数据库中的数据。

交易组合　供货商可以记录顾客做了几次购买并且在最后开出账单。这和维护账号相似，但是可能节约了一些初始开销。即使顾客只进行了一次购买，供货商也必须为顾客维护记录，因此开销可能超过收入。

数字现金　和传统的现金一样，数字现金——例如由银行或代理发行的数值代币——应该提供一个有效的小交易支付办法，但是数字现金的开发者必须解决两次支付问题。数字代币持有者可以对数字代币进行任意次无法觉察的复制，结果导致了两次支付。因此代币必须是惟一标识的，它们必须在使用时被证明为有效的、未花费的。困难在于设计一个在所有环境下都可伸缩的、可靠的并且经济有效的验证方案。

Millicent方案　Millicent项目已经开发出安全分发和使用临时凭证的方案——临时凭证是
303 一种专门设计用来进行小额交易的数字现金形式。Millicent方案相当有趣，因为它虽然也使用了本章叙述的若干种安全技术，却产生了与SSL及其他安全系统非常迥异的效果。

临时凭证是一种数字现金形式，它只对专门的供货商有效。它有以下特点：

- 只对特定的供货商有价值
- 只能被花费一次
- 能抵制篡改，难以伪造
- 只能被正当的所有者花费
- 能高效地产生和认证

Millicent方案的设计是可以伸缩的，因为每个供货商的服务器只负责验证他发行的临时凭证。顾客可以通过传统的交易从供货商处直接获得临时凭证，也可以从持有许多供货商临时凭证的代理商手中获得临时凭证。

临时凭证——由Millicent引入的供货商专有的货币，可由数字符号表示，格式如下所示：

供货商	价值	临时凭证ID	顾客ID	过期日期	属性	证书

属性字段由供货商决定使用——例如它可能包含顾客居住的国家或省市，由此可以应用相应的税率。证书是一个数字签名，它保护临时凭证中的所有域不被改变。签名由7.4.2节叙述的MAC方法产生。其他域的目的将在后面进行阐述。

临时凭证由代理生成、发布——代理是处理大量临时凭证的服务器，它减轻了顾客和供货商的部分负担。代理把临时凭证换成真实货币，按一定的折扣从供货商购得临时凭证（或生成临时凭证的权利），并且将临时凭证通过信用卡或其他支付方式卖给顾客。顾客可以从一个代理那里买到多个供货商的临时凭证，集中购买费用并在最后阶段支付。

场景　这是一个使用临时凭证的电子购买交易处理：顾客Alice对从供货商Venetia那里购买小产品或服务感兴趣（例如电话呼叫或网页）。Alice可能已经持有适合与Venetia进行交易

的临时凭证，这些临时凭证是前次交易所剩余的；如果没有，Venetia把代理Bob的URL给Alice，Bob处理Venetia的临时凭证（我们称之为V-临时凭证），Alice可从Bob处购买一些V-临时凭证。

然后Alice向Venetia发送一个购买请求，并且附上V-临时凭证，当然V-临时凭证应能支付得起她想购买的商品。Venetia验证临时凭证——检查其临时凭证ID在不在她的已经使用过的临时凭证列表内，以及临时凭证内的顾客ID是不是Alice的ID。如果临时凭证金额大于所需要金额，Venetia为差额产生一个新的临时凭证，把它作为零钱找给Alice。

代理可以通过几种方式获得供货商临时凭证。在最简单的情形中，Venetia制造临时凭证，按一定的折扣把它卖给Bob。另一种选择是，Venetia授权给Bob让他代表自己制造临时凭证。在这个模型中，在Venetia从顾客那里接收到付款之前，她对临时凭证一无所知。Venetia不时 304
地与Bob联系，把她从顾客那里接收到的临时凭证的标识符发送给Bob，Bob则支付给她这些临时凭证的折扣价。

目标 Millicent是为电子现金系统设计的，该系统避免了与中央服务器的通信开销和延迟，同时提供充分的安全保障防止欺诈使用。临时凭证的安全性依赖于数字签名，在必要的时候使用加密来保密。之所以使用低成本密码，是因为没必要使破坏系统安全的代价远远大于用它进行交易的价值。

Millicent方案是独立自主的。它不需要其他服务如安全的名字服务支持就可以实现，因为它不依赖外部的对顾客和供货商的认证。与之相反，Millicent方案为通信双方生成和发布惟一的保密密钥以进行临时凭证交易。

实现 图7-21给出了Millicent体系结构的概要。我们已经提到过，通过对临时凭证进行签名以防止篡改和伪造。当生成临时凭证时，附在每个临时凭证上的证书所包含的签名即构造出来了，它使用主临时凭证密文作为签名密钥。签名的方法和7.4.2节描述的MAC签名方法完全一样。密钥被附加在临时凭证的其他域上，使用一个MD5或SHA之类的安全摘要函数产生一个128位的散列值，即成为证书。

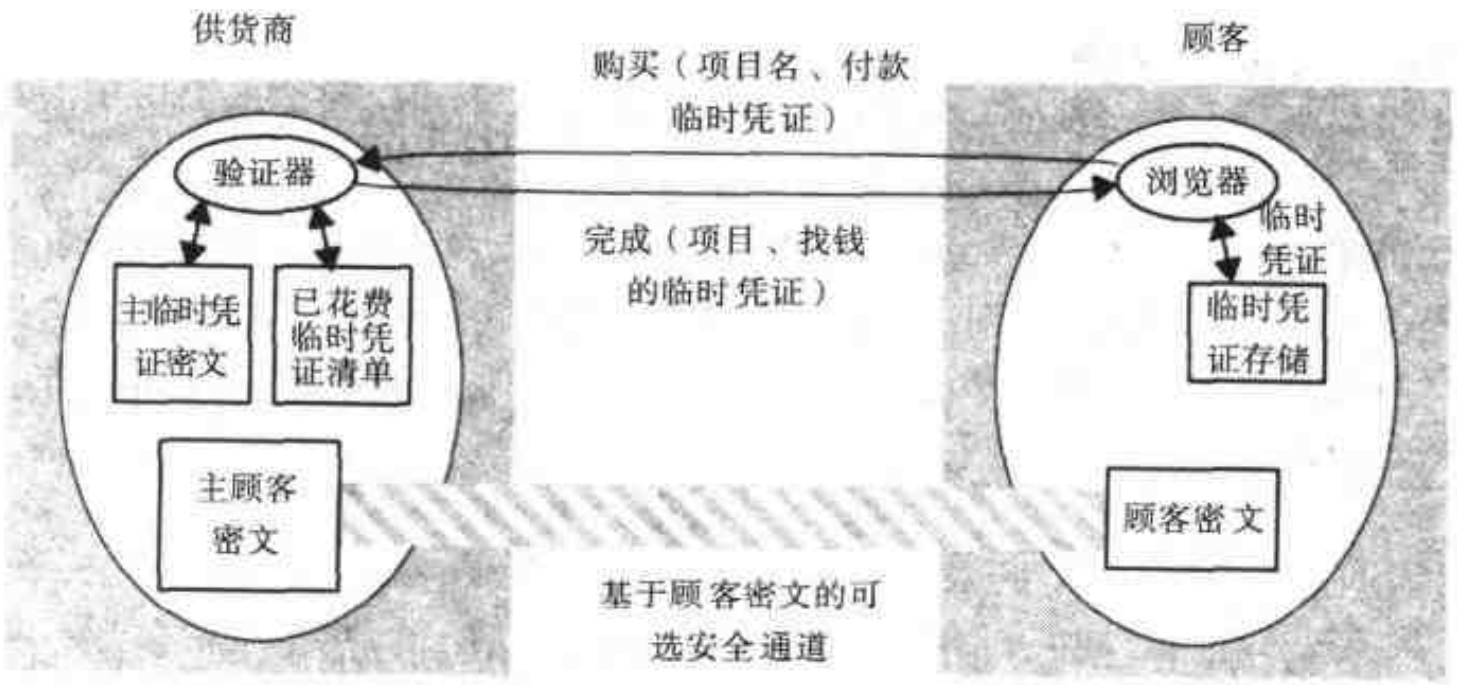


图7-21 Millicent体系结构

在产生任何临时凭证以前，供货商（或者被供货商授权的代理商）将产生一个64位的主临时凭证密文向量。这样的一个密文是足够的，但是有若干个可用的密文使得可以时常选择

一个新的密文，以防范密文的偶然泄露或某次成功的攻击。

每个临时凭证都有一个惟一的标识符，其值包括一个8位的主密文索引，用于从供货商的主密文向量中选择一个主密文。主密文由供货商保持，用于为产生的每个临时凭证生成一个
305 证书。

Millicent协议的一些变种已经被提了出来，以提供不同级别的安全。在所有这些协议中，供货商必须验证顾客提交的每个临时凭证项，我们先描述这个步骤。

为验证一个临时凭证：

1. 供货商通过使用相关的主密文（使用从临时凭证ID的主密文索引以从主密文向量中选择主密文）产生一个检查签名，验证该临时凭证没有被伪造或篡改，并且把它和证书中的签名进行比较。
2. 供货商验证临时凭证还没有被使用过。为做到这点，她必须维护一个ID的清单和所有她发布的临时凭证的过期日期，带有临时凭证是否被花费过的指示。过期日期域使供货商可以从清单中删除那些过期的临时凭证，以防止清单无止境地增长。顾客必须在过期日期前将老的临时凭证换成新的。

在上面场景中描述的交易可以仅仅基于验证就可以安全地执行。这保护了供货商，防止了伪造和两次支付，但是它不对顾客的临时凭证被盗进行保护，也不能为顾客和供货商提供任何保密性。

对盗窃的防范需要销售临时凭证的供货商或代理维护一个顾客主密文的向量（用顾客ID的一部分选择），并连同顾客密文发布给顾客。顾客密文的构造是通过把顾客主密文附加在顾客的ID上，并对结果应用如MD5的安全散列函数。顾客密文必须通过一个安全通道从供货商传给顾客。建议为起始的临时凭证的购买使用SSL。

一旦顾客密文被传给了顾客，顾客密文可以作为顾客和供货商之间的共享保密密钥。顾客可以使用它来签名交易，当需要保密时，顾客和供货商都可以把它作为加密密钥使用。

为了防止被盗，交易由顾客使用顾客密文签名，供货商检查临时凭证中的顾客ID是否与顾客密文相对应的顾客ID相匹配。如果不匹配，则表示该临时凭证在被该顾客以外的其他人花费——换言之，它已经被盗。

为了提供保密性，顾客把他的顾客ID发送给供货商，他们把顾客密文作为加密密钥，用保密密钥加密算法建立一个安全通道。这个安全通道即可以用于完成保密交易。

我们已经详细描述了Millicent协议，因为它提供了一个现实世界的例子以说明本章描述的许多安全技术的应用。Millicent系统是若干个已开发的应用于电子商务的电子现金方案之一。它最初是由在加利福尼亚Palo Alto的数字系统研究中心开发出来并由Compaq公司推向市场的
306 [www.millicent.com]。

7.7 小结

对于分布式系统的安全威胁是一个普遍现象。保护信道和可能成为攻击目标的信息处理系统的接口是非常重要的。个人电子邮件、电子商务和其他金融交易都是这样的信息例子。要小心地设计安全协议以防止出现漏洞。安全系统的设计起始于一系列威胁和一组最坏情况的假设。

安全机制是基于公开密钥密码学和保密密钥密码学的。密码算法以某种方式对消息进行

交换和混乱，使之在不知道解密密钥的情况下不可逆。保密密钥密码学是对称的——相同的密钥服务于加密和解密。如果通信双方共享一个保密密钥，他们可以交换加密了的信息，而不必冒着被窃听和篡改的风险，且能保证真实性。

公开密钥密码学是非对称的——加密和解密使用不同的密钥，只知道其中一个密钥不会泄露另一个。一个密钥是公开的，任何人可以发送安全消息给相应的私钥持有者，允许私钥持有者对消息和证书进行签名。证书可以作为使用被保护的资源的凭证。

资源通过访问控制机制得到保护。访问控制机制把权限分派给实体（即凭证持有者），使之能对分布式对象和对象集合执行操作。权限可能保存在与对象集合相关联的访问控制列表里（ACL），或者由主体按权能持有——权能为访问资源集合的不可伪造的密钥。权能对于授予访问权利来说十分便利，但是难于撤销。对ACL的改变能即刻生效，能回收以前的访问权利，但是对于ACL的管理比对权能的管理复杂得多，也昂贵得多。

直到最近，DES加密算法才成为最为广泛使用的对称加密方案，但是56位的密钥长度不够大到以足以防止强行攻击。DES的第3版给出了112位密钥长度，该长度是安全的，但其他的现代算法例如IDEA（128位密钥）运行速度更快而且提供了等同的或者更大的保护强度。

RSA是使用最为广泛的非对称加密机制。为了防范因数分解攻击，它应该使用768位或更大的密钥。保密密钥（对称）算法比公开密钥（非对称）算法性能优越多个数量级，因此公开密钥算法一般只用于混合协议如SSL中，例如在SSL中建立安全通道，该安全通道使用共享密钥进行后续的交换。

Needham-Schroeder认证协议是第一个通用的、实用的安全协议，它也为许多实际的系统奠定了基础。Kerberos是一个设计优良的单个组织中进行用户认证和服务保护的方案。Kerberos是基于Needham-Schroeder协议和对称密码学的。安全协议SSL被设计并广泛地运用于电子商务中。它是个灵活的协议，用于建立和使用基于对称密码学和非对称密码学的安全通道。Millicent实现了一个为小额交易设计的一种专门的“电子现金”形式。 307

练习

7.1 描述一些你们机构中的物理安全策略。按照可以在一个计算机化的门禁系统中实现的方式来表达。

7.2 描述一些情形，在这些情形中，传统的电子邮件易受到窃听、伪装、篡改、重播以及拒绝服务攻击。对电子邮件如何针对每种攻击形式采取相应的保护措施提出建议。

7.3 公开密钥的初始交换易受到中间人攻击。尽可能多地描述相应的防范措施。

7.4 PGP被广泛应用于安全电子邮件通信。为了有私密性和真实性保障，描述两个用户交换电子邮件前，使用PGP的步骤。在哪些范围内要使初始密钥协商对用户不可见？（PGP协商是混合方案的一个实例）。

7.5 如何使用PGP或其他相似的方案把电子邮件发送给一个大的接收者列表。当这个列表被频繁使用时，试提出一个更为简单快速的方案。

7.6 在图7-8～图7-10中给出的TEA对称加密算法的实现不可在所有的机器体系结构间移植，试解释原因。如何使一个由TEA算法实现加密的消息被传送，并正确地在所有其他的体系结构中进行解密？

7.7 修改图7-10中的TEA应用程序以使用密码块链接（CBC）。

7.8 根据图7-10中程序，构建一个流密码的应用程序。

7.9 已知强行攻击程序的内循环对于每个密钥值需要10个指令，再加上加密一个8字节明文的时间（见图7-14）。试估计使用一个500MIPS（每秒兆指令）的工作站，通过强行攻击破解一个56位DES密钥需要的时间，对于一个128位IDEA密钥进行同样的计算。推测如果使用一个50 000MIPS的并行处理器（或是一个具有相同处理能力的因特网社团）所需的破解时间。

7.10 在带有保密密钥的Needham-Shroeder认证协议中，试解释为什么消息5的下面这个
308 版本是不安全的：

$A \rightarrow B: \{N_B\}_{K_{AB}}$

第8章　分布式文件系统

8.1　简介
8.2　文件服务系统结构
8.3　Sun网络文件系统
8.4　Andrew文件系统
8.5　最新进展
8.6　小结

分布式文件系统支持在企业内部网上以文件的形式共享信息。一个设计良好的文件服务系统使用户访问存储在服务器上的文件时能获得与访问本地磁盘文件类似（在某些情况下更好）的性能和可靠性。一个分布式文件系统使程序可以像存储和访问本地文件那样的对远程文件进行操作，允许用户访问在企业内部网中任一计算机上的文件。

我们将介绍两种已被广泛使用十多年的分布式文件系统：

- Sun网络文件系统，NFS
- Andrew文件系统，AFS

这两个实例描述了模拟UNIX文件系统接口的多种设计方案，这些设计方案具有不同的可伸缩性和容错能力，但每一种设计并没有完全严格地模拟UNIX中的单一副本文件更新语义。

为了获得高性能、高容错和高伸缩性的文件系统，最近的分布式文件系统的设计利用了交换局域网的高带宽连接和新的磁盘数据组织模式。

8.1　简介

在第1章和第2章中，我们已经说明了共享资源是分布式系统的主要目标。共享存储信息可能是分布资源共享的最重要的一个方面。在因特网上，通过使用Web服务器的方式可在很大范围内共享资源，而在局域网和企业内部网上的共享信息时，客户需要另外一种类型的服务——它必须支持数据和各种类型程序的持久性存储和最新数据的一致性分布。本章旨在讨论基本分布式文件系统的结构和实现。我们在这里使用的“基本”一词表示分布式文件系统的主要目的是在多个远程计算机系统上为客户模拟非分布式文件系统的功能。它并不维持一个文件的多个持久副本，也不提供对多媒体数据流的带宽和实时保证——这些需求会在后面的章节中讨论。基本分布式文件系统对企业内部网上的有组织计算提供了必要支持。

我们首先简单介绍一下分布式和非分布式存储系统。文件系统最初是作为一项操作系统功能为集中式计算机系统或台式机设计开发的，它提供了方便的磁盘存储的程序接口。后来又加入了访问控制和文件锁机制以实现数据和程序的共享。在企业内部网上，分布式文件系统以文件和硬件资源等持久存储的形式来支持共享信息。一个设计良好的文件服务提供了与访问本地文件的性能和可靠性相似或更好的对服务器文件的访问方式。它们的设计能适应局域网的性能和可靠性特点，因此它们能提供在企业内部网使用中更有效的共享持久存储。在20世纪70年代研究者开发出第一个文件服务器[Birrell and Needham 1980，Mitchell and Dion

1982，Leach *et al.* 1983]，在20世纪80年代早期Sun的网络文件系统[Sandberg *et al.* 1985，Callaghan 1999]也投入了使用。

分布式文件系统的文件服务允许用户在企业内部网上的任一计算机上访问自己的文件，其程序可以像对待本地文件一样存储和访问远程文件。在几个服务器上集中存储文件可以减少本地磁盘存储，同时（更为重要的是）可以使对组织机构持有的持久数据的管理和存储更有效率。名字服务、用户认证服务和打印服务可以要求文件服务满足它们对持久存储的需求，因而可以更容易实现这些服务。Web服务器依赖于文件系统来存储其网页。在机构内，用户可以从外部或通过企业内部网访问Web服务器，而Web服务器经常从内部分布式文件系统中获取和存储数据。

随着分布式面向对象编程的出现，用户需要系统提供对共享对象的持久存储和分布。一种实现方法是序列化对象（按4.3.2节描述的方式），并使用文件存储并检索序列化对象。但对于快速变化的对象来说，这种获得持久性和分布性的方法是不可行的，因此研究者开发出一些更直接的方法。Java的远程对象调用和CORBA的ORB提供了访问远程共享对象的方式，但
310 它们都没有保证对象的持久性，也没有保证分布式对象的复制。

存储信息分布最近的发展包括分布式共享内存（DSM）系统和持久对象存储。第16章将详细介绍DSM。DSM通过在每一个主机上复制内存页或内存段，提供了对共享内存的一种模拟。它没有必要提供自动的持久性。持久对象存储已在第5章中介绍了，旨在为分布式共享对象提供持久性。此类例子有CORBA的持久对象服务（见第17章）和Java的持久性扩充[Jordan 1996，java.sun.com IV]。最近的研究开发出支持对象自动复制和持久存储的平台（例如，PerDiS[Ferreira *et al.*2000]和Khazana[Carter *et al.*1998]）。

图8-1给出了我们所提到的不同类型存储系统的一些性质。其中，一致性这一列表示当数据发生更新时是否有一种机制来维护其多个副本之间的一致性。实际上，所有存储系统都使用缓存来优化程序的性能，缓存首先应用到主存和非分布式文件系统，对它们而言，一致性是严格的（在图8-1中用“1”表示一个副本的一致性）——在更新后，程序不能发现存储数据与其缓存副本之间的任何区别。使用分布式副本时，很难达到严格的一致性。诸如Sun NFS和Andrew文件系统这样的分布式文件系统会将客户机的一部分文件复制到缓存中，并且采用了一种特殊的机制来近似地维持一致性。它在图8-1的一致性列中用钩（√）来表示——我们将在8.3节和8.4节讨论这些机制和它们与严格一致性的偏离程度。

	共　享	持久性	分布式缓存/副本	维　护一致性	例　子
主存	×	×	×	1	RAM
文件系统	×	√	×	1	UNIX文件系统
分布式文件系统	√	√	√	√	Sun NFS
Web	√	√	√	×	Web 服务器
分布式共享内存	√	×	√	√	Ivy (第16章)
远程对象（RMI/ORB）	√	×	×	1	CORBA
持久对象存储	√	√	×	1	CORBA 持久对象服务
持久分布式对象存储	√	√	√	√	PerDiS, Khazana

图8-1　存储系统及其性质

Web会使用客户机上的缓存和代理服务器上的缓存。在Web代理和客户机缓存上的副本和

原服务器中数据的一致性只能由用户来维持。当原服务器中的网页更新时客户并不知道，他 311
们必须执行检查操作来保持他们的本地副本为最新。在网页浏览中，这就能够满足要求了，但它不支持像共享分布式白板这样的协作式应用程序。第16章将详细介绍DSM系统使用的一致性机制。不同的持久对象系统使用缓存和一致性的差别相当大。CORBA和持久Java模式都只保持持久对象的单一副本，访问这些对象需要使用远程调用，所以它们的一致性问题仅仅是保持在内存中的对象和其在磁盘上的副本的一致性，这对远程用户是不可见的。前面介绍过的PerDiS和Khazana项目在缓存中维持对象的副本，它们采用了相当完备的一致性机制来保证与在DSM系统中相似的一致性形式。

在讨论了与持久和非持久数据的存储和分布相关的问题之后，我们现在返回到本章的主题——基本分布式文件系统的设计。我们将在8.1.1节介绍（非分布式的）文件系统的一些相关特性，在8.1.2节介绍分布式文件系统的需求，在8.1.3节介绍贯穿本章的实例研究。在8.2节中，我们将定义基本分布式文件服务的抽象模型，其中包括程序的接口集。8.3节介绍Sun NFS系统，它具有抽象模型的许多特征。在8.4节中，我们将描述Andrew文件系统——它是一个被广泛使用的系统，采用了完全不同的缓存和一致性机制。8.5节将回顾在文件服务设计领域的一些最新进展。

本章所描述的系统并没有包括分布式文件和数据管理系统的所有情况。本书后面的章节将会介绍几个具有更先进特性的系统。第14章介绍Coda系统，它是一个分布式文件系统，为了维持其可靠性、可用性和断连工作，它维护文件的多个持久副本。第15章介绍Tiger视频文件服务器，它的设计目的是为大量的用户提供实时的数据流传输。

8.1.1 文件系统的特点

文件系统负责文件的组织、存储、检索、命名、共享和保护（如图8-2所示）。它提供了描述文件抽象的程序接口，这样程序员就不必关心存储分配和存储布局的细节。文件存储在磁盘或其他不易失的存储介质上。

目录模块：	将文件名与文件ID关联
文件模块：	将文件ID和特定文件关联
访问控制模块：	检查操作请求的许可性
文件访问模块：	读或写文件数据或属性
磁盘块模块：	访问和分配磁盘块
设备模块：	磁盘I/O和缓冲

图8-2 文件系统模块

文件包括数据和属性。其数据部分包括一系列的数据项（通常是8比特的字节），读和写操作可访问任一部分数据。属性部分用一个记录表示，其中包括如文件长度、时间戳、文件类型、所有者身份和访问控制列表。图8-3描述了一个典型的属性记录结构。其中带阴影的属性是由文件系统管理的，用户程序不能更新它。

文件系统用与存储和管理大量文件，它具有创建、命名和删除文件的功能。目录系统支持文件命名。目录通常是一种特殊类型的文件，它提供从文件名到内部文件标识的映射。目录可以包括其他目录的名字，这样可以构建一种层次化的文件命名方案以及UNIX和其他一些

操作系统中使用的多部分组成的路径名。文件系统还负责控制对文件的访问，并根据用户授权及请求的访问类型（读、更新、执行及其他操作）限制对文件的访问。

文件长度
创建时间戳
读时间戳
写时间戳
属性时间戳
引用计数
所有者
文件类型
访问控制列表

图8-3 文件属性记录结构

元数据这一术语是指用于管理文件所需的存储在文件系统中的所有特殊信息。它包括文件属性，目录和其他文件系统使用的持久信息。

图8-2给出了传统操作系统中非分布式文件系统的实现具有的一个典型的层次模块结构。每一层只依赖其下面一层。分布式文件服务的实现需要图中所示的所有部件，可能需要附加模块来实现客户－服务器通信、分布命名以及文件定位。

312
~
313

文件系统操作 图8-4总结了在UNIX系统中应用程序可用的主要的文件操作。这些是由内核实现的系统调用，应用程序员通常通过调用诸如C标准输入输出库或Java文件类来访问这些操作。这里我们给出的原语是希望文件服务支持的操作，并用于与下面介绍的文件服务接口相比较。

filedes = open(name, mode)	打开一个名字为*name*的已存在文件
filedes = creat(name, mode)	用名字*name*创建一个新文件
	以上两个操作都给出打开文件的文件描述符，其中，*mode*可以取值*read*、*write*或*read*、*write*两者
status = close(filedes)	关闭已被打开的*filedes*文件
count = read(filedes, buffer, n)	从被*filedes*引用的文件中传输*n*字节给*buffer*缓冲区
count = write(filedes, buffer, n)	从*buffer*缓冲区传输*n*字节给被*filedes*引用的文件
	以上两个操作都会返回实际的传输字节数并移动读写指针
pos = lseek(filedes, offset, whence)	将读写指针移动到指定的位移处（根据*whence*决定是相对位移还是绝对位移）
status = unlink(name)	从目录结构中删除文件名*name*，如果此文件没有其他名字，它就被删除
status = link(name1, name2)	为文件（*name1*）添加新的名字（*name2*）
status = stat(name , buffer)	获得文件*name*的文件属性，并将其放入缓冲区*buffer*

图8-4 UNIX文件系统操作

UNIX操作基于一个程序模型，在这个程序模型中，对每个运行的程序，其文件状态信息被存储在文件系统中。它包含一系列当前打开的文件，在每个文件上有一个读-写指针，它用于为下一次读或写指示文件位置。

文件系统还负责文件的访问控制。在诸如UNIX这样的本地文件系统中，当文件被打开时，它在访问控制表中检查用户的权限，并将其与在*open*系统调用中请求访问的模式比较。如果权限符合模式，文件就被打开，同时该模式被记录在打开文件的状态信息中。

8.1.2 分布式文件系统的需求

在分布式文件系统的早期开发中发现了许多分布式服务设计的需求和潜在的困难。最初，分布式文件系统提供访问透明性和位置透明性，在开发的过程中，出现了性能、可伸缩性、并发控制、容错和安全需求，这些需求在随后的开发阶段中都得到了满足。我们将在后面的小节中讨论这些需求以及其他相关的需求。 314

透明性 在企业内部网上，文件服务通常都是负载最重的服务，因此它的功能和性能非常关键。文件服务的设计应该支持1.4.7节描述的分布式系统的许多透明性需求。其设计还必须平衡灵活性和可伸缩性与软件的复杂性和性能。下列透明性是当前文件服务部分解决或完全解决的透明性。

- 访问透明性 客户程序不必了解文件的分布性。用户通过一个统一的文件操作集来访问本地或远程文件。访问本地文件的程序在不做修改的情况下也应该能访问远程文件。
- 位置透明性 客户程序应该使用单一的文件名空间。在不改变路径名的情况下，多个文件或文件组应该可以被重定位，同时用户程序在任一时刻执行时都使用同样的命名空间。
- 移动透明性 当文件被移动时，客户程序和在客户结点上的系统管理表都不必改变。它们允许文件的移动性——多个文件或文件卷可以被系统管理员移动或自动移动。
- 性能透明性 当服务负载在一个特定范围内变化时，客户程序应可以保持满意的性能。
- 伸缩透明性 文件服务可以不断地扩充，以满足负载和网络规模的增长需要。

并发文件更新 一个客户改变文件的操作不应该影响其他客户访问或改变同一文件的操作。这就是为大家所熟知的并发控制问题，第12章会对此进行详细的讨论。许多应用程序都需要对共享信息的访问进行并发控制，一些实现技术也为大家所熟知，但开销比较大。当前大多数文件服务都遵循现代UNIX标准，提供建议性的和强制性的文件级或记录级加锁。

文件复制 在支持文件复制的文件服务中，一个文件可以表现为在不同位置文件内容的多个副本。这带来了两个好处——它允许多个服务器共享文件服务的负载，增强了服务的伸缩性，同时改进容错性能，因为当一个文件损坏时，客户可以访问另一具有此文件副本的服务器。少数文件服务完全地支持复制，但大部分都支持文件缓存或本地部分文件复制，这只是一种有限的复制形式。数据复制的讨论详见第14章，其中包括Coda复制文件服务的描述。

硬件和操作系统异构性 文件服务的接口必须有明确的定义，这样在不同的操作系统和计算机上可以实现客户和服务器软件。这一需求是开放性的一个重要方面。

容错 文件服务在分布式系统中的中心角色决定了它必须在客户和服务器出现故障时能继续使用。幸运的是，有一种专门适用于简单服务器的合适的容错设计。为了应对短暂的通信错误，容错设计可以基于至多一次的调用语义（见5.4.2节）。而在按幂等操作设计的服务器

315 协议中容错设计可以使用更简单的至少一次语义，以保证重复的请求不会导致对文件的不合法更新。服务器可以是无状态的，这样它可以重新启动，同时在发生错误后恢复时，它不需要恢复以前的状态。文件复制可以实现对连接中断或服务器故障的容错，这一点很难达到，我们会在第14章讨论这一问题。

一致性 像UNIX文件系统这样的传统的文件系统提供的是单一副本更新语义。它提供了一个对文件并发访问的模型，即当多个进程并发访问文件时，它们只看到仅有一个文件副本存在。当文件在不同的地点被复制或被缓存时，对一个副本所做的修改要被传播到所有副本，这中间有一个不可避免的延迟，这种情况可能会导致偏离单一副本语义。

安全性 几乎所有的文件系统都提供基于访问控制列表的访问控制机制。在分布式文件系统中，客户的请求需要被认证，这样在服务器上的访问控制需要基于正确的用户身份，同时还需要使用数字签名和私密数据加密（可选）机制来保护请求和应答消息的内容。

效率 分布式文件系统应该提供比传统的文件系统更多、更强的功能，并且在性能方面也应该能与传统文件系统相比。Birrell和Needham[1980]用如下语句介绍了他们的Cambridge文件服务器（CFS）的设计目标：

> 为了共享一个昂贵的资源，也就是磁盘，我们希望拥有一个简单、初级的文件服务器，这样我们就可以自由地设计适合于特定客户的文件系统，但同时我们也希望有客户可共享的高级的系统。

磁盘存储费用的节省降低了第一个目标的重要性，但满足不同客户的不同需求的目标仍然存在，它能用前面描述的模块化体系结构解决。

实现文件服务的技术是分布式系统设计中的一个重要部分。一个分布式文件系统应提供在性能和可靠性方面能与本地文件系统相比拟的、甚至更好的服务。它必须提供相应的操作和工具，使得系统管理员能方便地安装和管理系统。

8.1.3 实例研究

我们为文件服务构造了一个抽象模型，这个模型与实现机制分离并且比较简单，在此将其作为介绍性的例子。我们略微详细地描述了Sun网络文件系统，这些细节添加到简单的抽象模型上，可以使它的体系结构更加清晰。然后，我们介绍Andrew文件系统，它采用不同的方

316 法获得可伸缩性和一致性维护。

文件服务系统结构 这一抽象体系结构模型同时支持NFS和AFS。它基于在3个模块间的职责划分。这3个模块中包括一个为应用程序模拟传统文件系统接口的客户模块和两个分别为客户提供目录和文件操作的服务器模块。这种体系结构设计使得服务器可以实现成无状态方式。

Sun NFS Sun Microsystem的网络文件系统（NFS）自从1985年面世以来，被广泛应用于产业和教学科研。1984年，Sun Microsystem的工作人员承担了NFS的设计和开发[Sandberg *et al.*1985；Sandberg 1987，Callaghan 1999]。尽管已经开发出来了一些分布式文件服务，并且成功应用于学校和研究机构，但NFS是第一个作为产品设计的文件服务。NFS的设计和实现在技术上和商业上获得了相当大的成功。

为了将NFS推广为一个标准，Sun公开了NFS主要的接口定义[Sun 1989]，以允许其他提供商并发实现，同时通过授权的方式其他厂商还可获得参考实现的源代码。现在，许多提供商支持NFS，同时定义在RFC 1813[Callaghan *et al.*1995]的NFS协议（第3版）也已经成为一个

因特网标准。Callaghan关于NFS的书[Callaghan 1999]是关于NFS的设计和实现以及相关主题的一本极好的参考书。

NFS为运行在UNIX和其他系统上的客户程序提供了访问远程文件的透明性。通常，系统中（至少在UNIX系统中）每台计算机的系统内核中都安装有NFS客户模块和服务器模块。客户－服务器的关系是对称的：每一个在NFS网络上的计算机可以同时扮演客户和服务器两种角色，同时每一台机器上的文件都可以被其他机器远程访问。当对外提供自己的文件时，计算机扮演的是服务器的角色；当访问其他机器的文件时，它扮演的是客户的角色。但在实际环境中，通常会将某些配置较高的机器作为服务器，而将其他机器作为工作站。

NFS的一个重要目标是实现对硬件和操作系统异构性的高层支持。NFS的设计是独立于操作系统的：客户和服务器几乎可以在当前所有的操作系统平台上实现，包括Windows 95、Windows NT、MacOS、VMS以及Linux和几乎所有其他版本的UNIX。有一些提供商在高性能多处理器主机上开发了NFS实现，它们被广泛使用以满足企业内部网并发用户的存储需要。

Andrew文件系统 Andrew系统是卡内基·梅隆大学（CMU）为校园计算和信息系统[Morris *et al.* 1986]开发的一个分布计算环境。Andrew文件系统（以后简称为AFS）的设计反映了通过减少客户－服务器通信量来支持大范围内共享信息这一目的。通过在客户和服务器之间传输整个文件（对大文件，传输64KB的文件块），并在客户机中缓存文件直到服务器收到一个更新的版本，以上设计可以达到前面提出的目的。使用Andrew系统的校园计算网络预计在系统生存期中会包括5000到10000台工作站。在Satyanarayanan[1989a；1989b]的描述之 317
后，我们会介绍AFS-2，第一个“产品”级实现。对AFS的近期介绍可以在Campbell[1997]和[Linux AFS]中找到。

AFS首先在CMU运行BSD UNIX和Mach操作系统的网络工作站和服务器中实现，随后，实现了它在商业和公用领域版本。最近，在Linux操作系统[Linux AFS]上也实现了AFS的公用领域版本。1991年在CMU，AFS大约支持由40台服务器提供服务的800台工作站，同时还支持其他的由互联网连接的远程客户和服务器。AFS被用作开放软件基金会（OSF）的分布式计算环境（DCE）[www.opengroup.org]中的DCE/DFS文件系统的基础。DCE/DFS的设计在一些重要方面超越了AFS，我们会在8.5节介绍。

8.2 文件服务系统结构

如果文件服务的结构包含3个模块——平面文件服务、目录服务和客户端模块，那么它的开放性和可配置性的范围便扩大了。图8-5显示了相关的模块以及它们之间的关系。平面文件服务和目录服务分别为客户程序提供一个接口，它们与RPC接口一起提供了一组完整的访问文件的操作。客户模块提供了同传统文件系统相似的关于文件操作的单一程序接口。设计的开放性在于可以用不同的客户模块实现不同的程序接口，模拟不同操作系统的文件操作，优化不同的客户和服务器硬件配置的性能。

模块之间的职责划分如下：

平面文件服务 平面文件服务注重于在文件内容上的实现操作。文件惟一标识符（UFID）在所有平面文件服务操作的请求中用于指明文件。文件服务和目录服务的职责划分是基于UFID的使用。UFID是一长串位序列，每个文件的UFID在分布式系统的所有文件中是惟一的。当平面文件服务接收到一个创建文件的请求，它生成一个新的UFID并将此UFID返回给请求者。

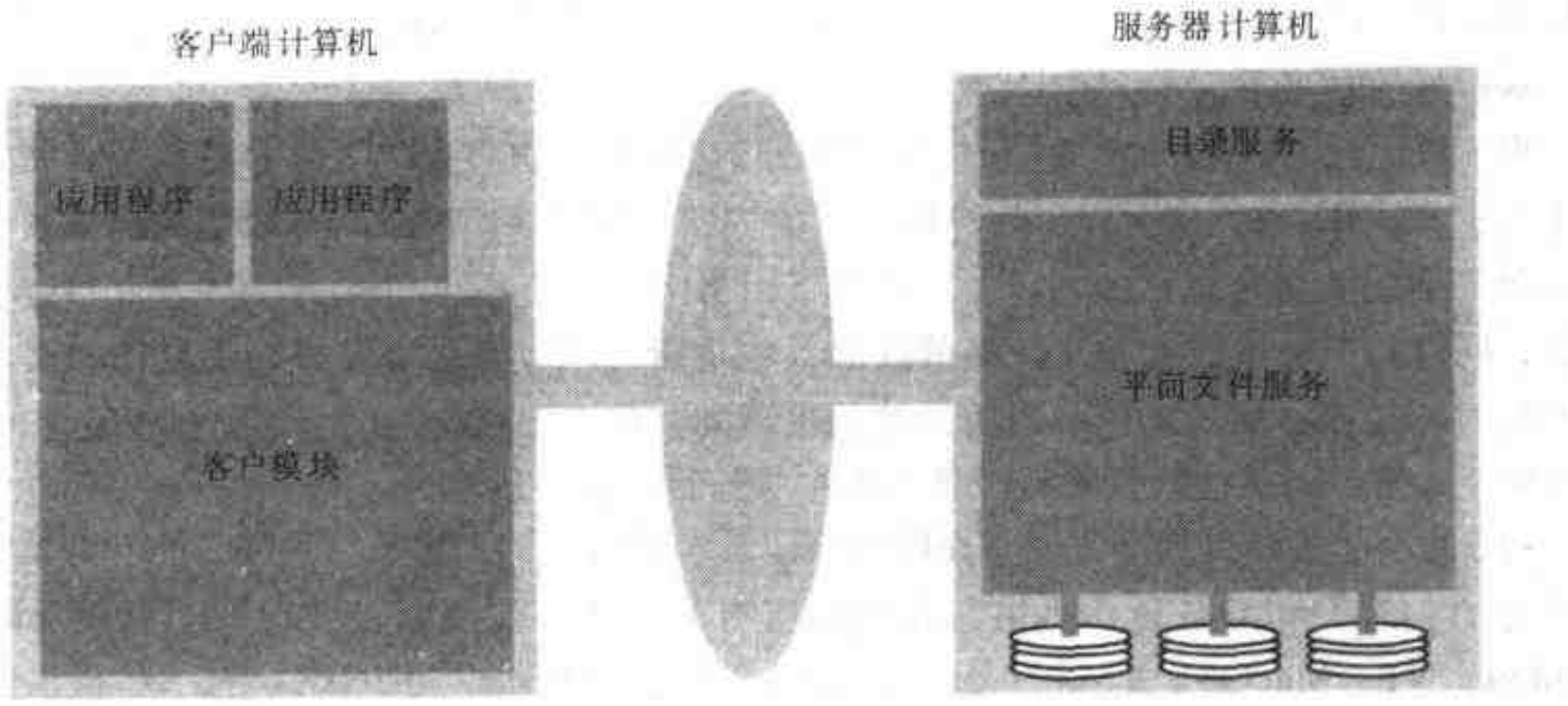


图8-5 文件服务系统结构

目录服务 目录服务提供文件名到UFID的映射。客户通过向目录服务提供文件名获得文件的UFID。目录服务提供生成目录、在目录中加入新的文件名以及从目录中获得UFID所必需的功能，它是平面文件服务的客户，其目录文件存储在平面文件服务提供的文件中。当采用像UNIX的层次化文件命名方案时，目录包含对其他一些目录的引用。

客户模块 客户模块运行在客户计算机上，它集成和扩展了平面文件服务和目录文件服务的操作，并为客户计算机上的用户级程序提供单一应用程序接口。例如，在UNIX主机上，一个客户模块通过向目录服务发出重复的请求来解释UNIX的多部分文件名，从而模拟完整的UNIX文件操作集。客户模块同时也拥有平面文件服务器和目录服务器进程的网络位置信息。最后，客户模块还可以扮演的重要角色是通过在客户端缓存最近使用的文件块方式来获得满意的性能。

平面文件服务接口 图8-6包含了对平面文件服务的接口定义。这是客户模块使用的RPC接口，它并不是直接被用户级程序使用。当*FileId*所指的文件不在服务器进程中，或访问权限不允许对此文件进行操作时，*FileId*是非法的。如果*FileId*参数包含非法的UFID或用户没有足够的访问权限，那么，除了*Create*之外的所有接口上的过程都会抛出异常。为清晰起见，定义中省略了这些异常。

Read(FileId, i, n) → Data – 抛出*BadPosition*	如果$1 \leq i \leq Length(File)$：从文件中读取从*i*项位置开始到*n*项，并在*Data*中返回结果
Write(FileId, i, Data) – 抛出*BadPosition*	如果$1 \leq i \leq Length(File)+1$：从文件的*i*项位置开始写入*Data*序列，在需要时扩展文件
Create()→FileId	生成一个长度为0的新文件，并为其指定一个UFID
Delete(FileId)	从文件存储中删除一个文件
GetAttributes(FileId) →Attr	返回指定文件的文件属性
SetAttributes(FileId, Attr)	设置文件属性（图8-3中没有阴影的那些属性）

图8-6 平面文件服务操作

318 ~ 319

读和写是最重要的文件操作，*Read*和*Write*操作都需要一个参数*i*来指定文件的读写位置。*Read*操作从指定文件的第*i*项开始顺序复制*n*个数据项到*Data*中，然后将*Data*返回给客户。*Write*操作复制*Data*中的一系列数据项到指定文件的第*i*项位置，它会替换原有文件相应位置的

内容，并在需要时扩展文件。

*Create*操作创建一个新的空文件并返回生成的UFID。*Delete*操作删除指定的文件。

*GetAttributes*和*SetAttributes*操作使客户能访问属性记录。*GetAttributes*操作通常对每个可以读文件的客户都可用。对*SetAttributes*操作的访问通常被限制在提供访问文件的目录服务。属性记录的长度和时间戳的值不会被*SetAttributes*操作改变，它们由平面文件服务自身管理。

与UNIX比较 这一接口和UNIX的文件系统原语在功能上等价。用下一节介绍的平面文件服务和目录服务可以很容易地构建模拟UNIX系统调用的客户模块。

相对于UNIX的接口，平面文件服务没有*open*和*close*操作——只要是提供恰当的UFID即可访问文件。在平面文件服务的接口中，*Read*和*Write*请求包括指明文件中起始读写点的参数，而在与之等价的UNIX操作中则没有。在UNIX中，每一个*read*或*write*操作在读－写指针的当前位置开始操作，并且读－写指针在*read*或*write*操作传输完数据后会自动改变当前位置。*seek*操作用于显式地使读写指针重新定位。

平面文件服务的接口与UNIX文件系统接口的差别主要因为容错的缘故：

- **可重复的操作** 除了*Create*操作之外，其他操作是幂等的，它允许使用至少一次的RPC语义——客户可能在收不到应答的情况下重复调用。重复执行*Create*操作会每次生成一个新的文件。
- **无状态服务器** 接口适合于用无状态服务器实现。无状态服务器可以在失败后重启，它可以在不需要客户或服务器存储任何状态的情况下继续操作。

UNIX 文件操作既不是幂等的，也与无状态实现的需求不一致。当文件被打开时，读－写指针由UNIX文件系统生成，并且与访问控制检查的结果共同保持到文件关闭为止。UNIX 的*read*或*write*操作不是幂等的，如果意外重复一个操作，读－写指针的自动变化会导致在重复的操作中访问文件的不同位置。读－写指针是一个隐藏的、与客户相关的状态变量。为了在文件服务中模仿它，系统需要提供*open*和*close*操作，并且必须在相关文件打开后就一直维持它的值。通过消除读－写指针，我们消除了大多数文件服务中需要的代表客户的保留状态信息。 320

访问控制 在UNIX文件系统中，系统会将用户的访问权限和在*open*中的请求访问（读或写）模式做比较（图8-4给出了UNIX文件系统的API），并且只有在用户拥有必要的权限时，才能打开文件。在访问权限检查中使用的用户标识（UID）是用户早期认证登录的结果，并且在非分布式的实现中，它是不能被修改的。访问权限会保持到文件关闭，并且在同一文件上进行随后操作时，系统不需要进行第二次检查。

在分布式的实现中，访问权限检查必须在服务器上进行，这是因为服务器RPC接口是一个访问文件的无保护的点。用户标识必须在请求中传输，并且服务器容易被伪造的标识欺骗。更为严重的是，如果访问权限检查的结果被保留在服务器上并在下一次访问中使用时，服务器就不再是无状态的。有两种方法可以解决后一个问题：

- 当文件名被转化为UFID时，系统进行一次访问检查，同时其结果按权能的形式编码（见7.2.4节），它作为以后一系列请求的访问许可将被返回给客户。
- 在每一次客户请求时，客户都发送用户标识，并且在每一次文件操作时，服务器都进行访问检查。

这两种方法都支持无状态的服务器实现，并且它们都已经用在分布式系统中了。第二种方法的应用更广泛一些；NFS和AFS都使用这种方法。两种方法都没有解决关于伪造用户标识

的安全问题。此问题可以由第7章介绍的数字签名解决。Kerberos是一种有效的认证方案，它已经用于NFS和AFS中。

在我们的抽象模型中，我们没有说明采用哪种访问控制实现方法。用户标识可以作为一个隐式参数传递，并且在需要的时候使用它。

目录服务接口 图8-7包含了目录服务的RPC接口定义。目录服务的主要目的是提供从文件名字到UFID的翻译服务。为了做到这一点，它保留了一个包含文件名到UFID映射的目录文件。每一个目录像普通文件一样存储，也有其UFID。因此，目录服务是文件服务的一个客户。

Lookup(Dir, Name) →*FileId*	在目录中获得文件名的位置，并返回相应的UFID
—抛出*NotFound*	如果在目录中没有找到*Name*，便抛出异常
AddName(Dir, Name, File)	如果目录中没有*Name*，将（*Name*, *File*）加入到目录中，并更新其文件属性记录
—抛出*NameDuplicate*	如果在目录中已经有*Name*，便抛出异常
UnName(Dir, Name)	如果在目录中已经有*Name*包含*Name*的条目被删除
—抛出*NotFound*	如果在目录中没有找到*Name*，便抛出异常
GetNames(Dir,Pattern) →*NameSeq*	返回在目录中所有符合正则表达式*Pattern*的文件名

图8-7 目录服务操作

我们只定义了在单个目录上的操作。在每一个操作中，需要包含在目录中的文件的UFID（在*Dir*参数中）。基本目录服务中的*Lookup*操作执行一个文件名→*UFID*转换。它是一个构造块，供其他服务或客户模块使用以完成更复杂的翻译，如在UNIX中的层次文件名翻译。像以前一样，定义中省略了由于访问权限不够而引起的异常。

改变目录有两种操作：*AddName*和*UnName*。*AddName*向目录增加一个条目，并且在文件的属性记录中增加引用计数。

*UnName*从目录中删除一个条目并将引用计数减一。当引用计数减少到零的时候，文件被删除。*GetNames*使客户可以检查目录内容，还可以实现类似于UNIX shell中的对文件名的模式匹配操作。它返回存储在给定目录中的全部文件名或文件名的子集。在此操作中，系统通过对客户提供的正则表达式进行模式匹配来寻找文件名。

*GetNames*操作提供的模式匹配使用户能够通过给出一个不完全的文件名来查找一个或多个文件。一个正则表达式是一种由子字符串和标识可变字符或重复出现的字符子串的符号组成的字符串表达式。

层次文件系统 像UNIX提供的由树型结构组织目录的文件系统是一种层次文件系统。每一个目录包含文件名和其他可以从此目录访问的目录名。可以使用路径名来访问任一文件或目录——路径名是代表树中一条路径的多部分名字。其根有一个特定的名字，并且每一个在目录中的文件或目录都有其名字。UNIX的文件命名机制不是严格的层次型结构——一个文件可能有多个名字，它们可以在相同或不同的目录中。这是用*link*操作实现的，该操作可以为指定目录中的文件加入新的名字。

像UNIX这样的文件命名系统可以通过使用了平面文件服务和目录服务的客户模块实现。在目录的树型结构中，文件在叶结点，而目录在树的其他结点。树的根是一个具有固定UFID的目录。可以通过使用*AddName*操作和在属性记录中使用引用计数来实现文件的多重命名。

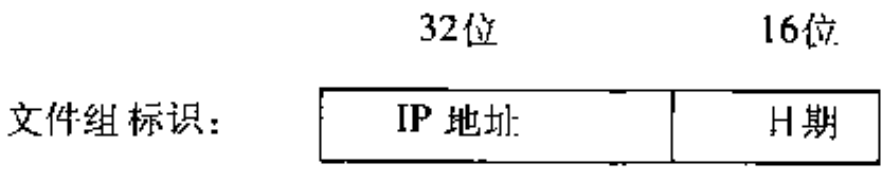
客户模块提供一个函数，用于获得给定路径的文件的UFID。该函数从根开始解释路径名，

通过使用*Lookup*操作可获得路径上每一个目录的UFID。

在层次化目录服务中，与文件相联系的文件属性应该包括一个区别普通文件和目录的特定域。可以用它来区分路径名各部分是否指向目录。 321~322

文件组 文件组是一个给定服务器上的文件集合。一个服务器可能包含多个文件组，组不能在服务器之间移动，并且文件不能改变它隶属的组。在UNIX和大多数其他操作系统中用到了一个相似的构造（叫做文件集系统）。文件组最早用于支持在计算机间移动那些存储在可移动磁盘上的文件集合。在分布式文件服务中，文件组支持将文件以更大的逻辑单位分配到服务器上，同时它还支持用存储在几个服务器上的文件实现文件服务。在支持文件组的分布式文件系统中，UFID包括一个文件组标识组件，它能使每个客户计算机上的客户模块负责向包含相应文件组的服务器发送请求。

在分布式系统中，文件组标识必须惟一。因为文件组可以被移动，同时也因为开始分离的分布式系统可以合并成一个单一的系统，所以保证文件组标识在给定的系统中惟一的方法只能是：用一个确保全局惟一性的算法生成文件组标识。例如，创建新的文件组时，可由创建新文件组的主机的32位IP地址和一个根据日期生成的16位整数拼接而成的48位整数来形成惟一标识。



要注意的是，IP地址不能用来定位文件组，这是因为它可以被移动到其他服务器上。文件服务应该维护一个在组标识和服务器之间的映射。

8.3 Sun网络文件系统

图8-8给出了Sun NFS的体系结构。它遵循前面介绍的抽象模型。所有的NFS实现都支持NFS协议——为客户提供操作远程存储的文件的远程过程调用集合。NFS协议是与操作系统无关的，但是它最初是在UNIX系统网络环境中开发出来的，我们将描述NFS协议（第3版）的UNIX实现。

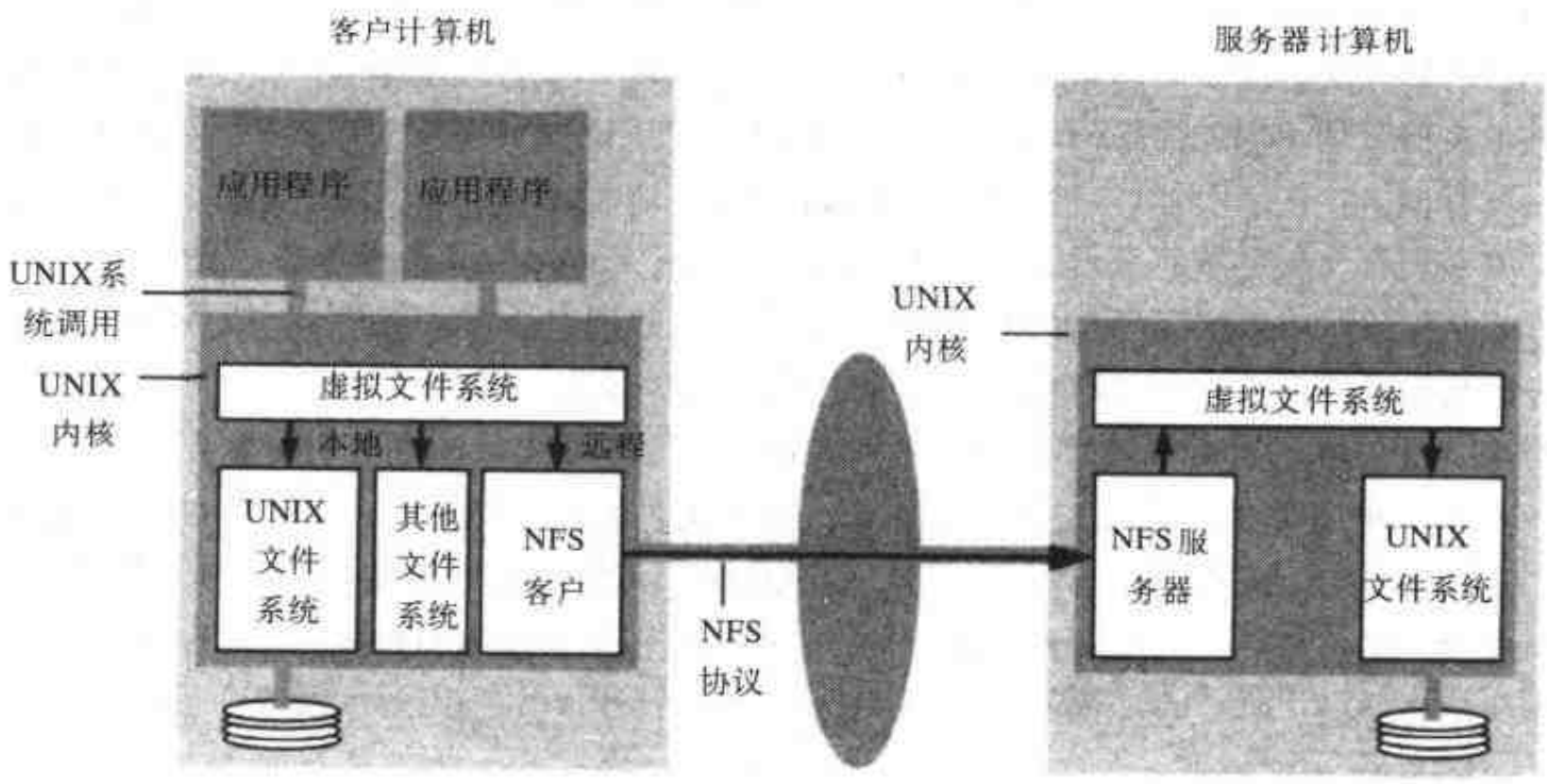


图8-8 NFS体系结构

NFS服务器模块放置在每一个作为NFS服务器的计算机内核上。客户模块将对在远程文件系统中的文件的引用转换为NFS协议操作，并将它传输给保存相关文件系统的计算机的NFS服务器模块。

NFS客户和服务器模块使用远程过程调用进行通信。5.3.1节描述的Sun RPC系统是为NFS开发的。它可以配置为基于UDP或基于TCP，而NFS协议可以适应这两种配置情况。其中包括一个端口映射服务，它能使客户将服务绑定在指定名字的主机上。RPC为NFS服务器提供的接口是开放的：任何进程都能向NFS服务器发请求，如果请求是有效的并且包含合法的用户凭证，那么系统会进行相应的操作。提交有用户签名的凭证可以作为一个可选的安全机制，它就像数据加密一样能提供私密性和完整性。

虚拟文件系统 图8-8表明NFS提供访问透明性：用户程序可以保证用户对本地和远程文件的访问没有什么区别。其他分布式文件系统也可能支持UNIX系统调用，如果是这样，那么它们可以用同样的方法集成。

虚拟文件系统（VFS）模块可以实现以上集成，该模块已经被加入到UNIX内核中，用于区别本地和远程文件，它还用于在NFS使用的独立于UNIX的文件标识与UNIX及其他文件系统中使用的内部文件标识之间的转换。另外，VFS保持对当前可用的本地和远程文件集系统的跟踪，并且它将每一个请求发送到合适的本地系统模块上（UNIX文件系统、NFS客户端模块或其他文件系统中的服务模块）。

在NFS中使用的文件标识被称为*文件句柄*，文件句柄对客户是透明的，它包含服务器区分单个文件所需要的信息。在NFS的UNIX实现中，文件句柄是从文件的i-结点号得来的，它在其中加入如下的两个附加域（UNIX文件的i-结点号用于在存储文件的文件系统中标识和定位文件的数值）：

文件句柄：	文件集系统标识	文件的i-结点号	i-结点产生号

NFS采用UNIX的可安装文件集系统作为前面定义的文件组单元（注意术语上的区别：*文件集系统*指的是*存储*在一个*存储*设备或分区上的文件集合，而*文件系统*指的是提供文件访问的软件组件）。*文件集系统*标识字段是在生成每一文件集系统后为其分配的一个惟一的数值（在UNIX实现中，它被*存储*在文件系统的超级块中）。因为在传统的UNIX文件系统中，i-结点号在文件被删除后被其他文件复用，因此需要i-结点产生号。在VFS对UNIX文件系统的扩展中，i-结点产生号和文件一样存储，并在每次i-结点被复用时（例如，在UNIX的creat系统调用中）加一。文件句柄是在客户安装远程文件系统时获得的。文件句柄以*lookup*、*create*和*mkdir*等操作（见图8-9）的调用结果形式从服务器传送给客户，以所有服务器操作的参数列表形式从客户传到服务器。

323 ~ 324

在虚拟文件系统层中，对应于每一个已安装的文件系统有一个VFS结构，并且对每一个打开的文件有一个*v*-结点。VFS结构将一个远程文件系统与VFS安装的本地目录联系起来。*v*-结点包含一个指示此文件是本地还是远程的标识。如果文件是在本地，*v*-结点包含对本地文件索引的引用（在UNIX实现中，是一个*i*-结点）。如果是远程文件，它包含远程文件的文件句柄。

客户集成 NFS客户模块扮演的是文件系统结构模型中客户模块的角色，它提供适用于传统应用程序使用的接口。但与模型中的客户模块不同的是：它精确模拟标准UNIX文件系统原语的语义，并且它与UNIX内核集成到一起，而不是以客户进程运行时动态加载的库的形式提供，这样：

- 用户程序可以通过UNIX系统调用访问文件无需重新编译或重新加载库。

- 一个客户端模块，通过使用一个最近使用的文件块（将在下面介绍）的共享缓存为所有的用户级进程服务。
- 传输给服务器用于认证用户ID的密钥可以由内核保存，这样可以防止用户级客户假冒用户身份。

在每一台客户机器上，NFS客户模块与虚拟文件系统协同工作。它以一种与传统UNIX文件系统相似的方式来操作，它在服务器和客户之间传输文件块，并在可能的情况下将文件块缓存到本地的内存中。它共享本地输入输出系统使用的缓冲区缓存，但由于可能会有在不同主机上的多个客户同时访问同一远程文件，所以出现了新的缓存一致性问题。

访问控制和认证 与传统UNIX文件系统不同的是：NFS服务器是无状态的，并且不为客户持续打开文件。因此在用户发出每一个新的文件请求时，服务器必须比较用户标识和文件访问许可属性来检查是否允许用户进行相应的访问。Sun RPC协议需要用户在每一次请求时发送用户认证信息（例如，传统UNIX的16位用户ID和组ID），同时将它与文件属性中的访问许可进行对比。图8-9是对NFS协议的简单介绍，其中没有给出这些附加参数，它们由RPC系统自动提供。

lookup(dirfh, name)→fh, attr	返回目录*dirfh*中的文件*name*的文件句柄和属性
create(dirfh, name, attr)→newfh, attr	在目录*dirfh*中创建具有*attr*属性的新文件*name*，返回新文件的句柄和属性
remove(dirfh, name)→status	从目录*dirfh*中删除文件*name*
getattr(fh)→attr	返回文件*fh*的文件属性（类似于UNIX的*stat*系统调用）
setattr(fh, attr)→attr	设置属性（模式、用户ID、组ID、文件大小、访问时间和文件的修改时间），将文件大小设为0意味着截断文件
read(fh, offset, count) →attr, data	从文件*offset*位置开始读*count*个字节的数据，并返回文件的最新属性
write(fh, offset, count, data) →attr	从文件*offset*位置开始写*count*个字节的数据，并返回写完后文件的属性
rename(dirfh, name, todirfh, toname) →status	将在*dirfh*目录中的文件*name*的名字改为在*todirfh*目录中的*toname*文件名
link(newdirfh, newname, dirfh, name) →status	在目录*newdirfh*中创建一个*newname*条目，该条目指向目录*dirfh*中的*name*文件
symlink(newdirfh, newname, string) →status	在目录*newdirfh*中创建一个类型为*symbolic link*、值为*string*的新条目*newname*，服务器并不解释*string*而是建立一个符号连接文件保存该*string*
readlink(fh) →string	返回由*fh*标识的符号连接文件所保存的字符
mkdir(dirfh, name, attr) →newfh, attr	创建一个具有*attr*属性的新目录*name*，并且返回新的文件句柄和属性
rmdir(dirfh, name) →status	从父目录*dirfh*中删除空目录*name*，如果此目录不空，操作失败
readdir(dirfh, cookie, count) →entries	从目录*dirfh*中返回*count*字节的目录条目，每一个条目包含一个文件名、一个文件句柄和一个指向下一个目录条目的指针，该指针被称为*cookie*。*cookie*用于在下一个*readdir*操作中从下一个目录条目中开始读。如果*cookie*的值是0，读目录中第一个条目
statfs(fh) →fsstats	为包含文件*fh*的文件系统返回文件系统信息（例如块大小，空块的数目等等）

图8-9 NFS服务器操作（简化表示）

这里将以最简单的形式介绍访问控制机制的安全漏洞。在每个主机的已知端口上，NFS服务器提供了一个传统的RPC接口，并且作为客户，每一进程可以向服务器发请求来访问和更新文件。客户可以修改RPC调用中的用户ID以假冒该用户。这一安全漏洞可以通过在RPC协议中使用用户认证信息的DES加密方法来弥补。最近，Kerberos已经与Sun NFS集成起来，以对用户认证和安全性问题提供功能性更强、更全面的解决方法，我们将在下面介绍它。

325 ? 326

NFS服务器接口 图8-9给出了由NFS服务器（在RFC 1813[Callaghan *et al.* 1995]中定义的）提供的RPC接口的一个简化表示。NFS的文件访问操作*read*、*write*、*getattr*和*setattr*几乎等同于我们的平面文件服务模型（图8-6）中定义的*Read*、*Write*、*GetAttributes*和*SetAttributes*操作。在图8-9中定义的*lookup*操作和其他大多数目录操作和我们在目录服务模型（图8-7）中定义的操作相类似。

文件和目录操作被集成在一个服务中，单一的create操作就能在目录中创建和插入文件名，该操作将新文件的文件名和目标目录的文件句柄作为参数。其他在目录上的NFS操作包括*create*、*remove*、*rename*、*link*、*symlink*、*readlink*、*mkdir*、*rmdir*、*readdir*和*statfs*。除了*readdir*（提供了一个读目录内容的方法）和*statfs*（给出远程文件系统状态信息）之外，它们都在UNIX中有对应的操作。

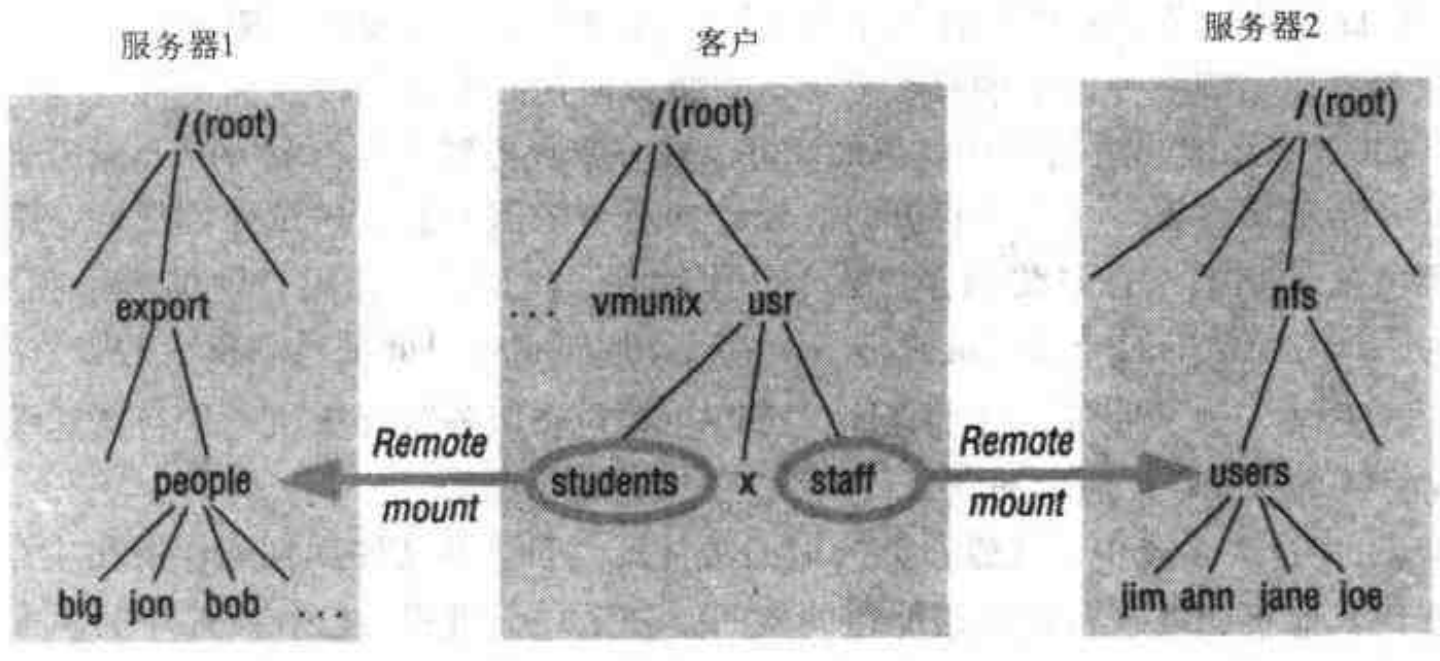
安装服务 运行在每一个NFS服务器计算机上的安装服务进程支持客户安装远程文件集系统的子树。每一个服务器有一个已知名字的文件（*/etc/exports*），它包含可被用来远程安装的本地文件集系统的名字。每一个文件集系统的名字与一个访问列表相联系，该表用来指明哪些主机被允许安装这个文件集系统。

客户使用一个修改过的UNIX *mount*命令，其中指定远程主机名字、远程文件集系统的目录路径和将要安装的本地名字，来请求安装一个远程文件集系统。远程目录可以是所请求的远程文件系统的任意子树，这使得客户能安装任一部分的远程文件集系统。修改过的*mount*命令使用安装协议与远程主机上的安装服务进行通信。安装协议是一种RPC协议，并且包含以目录路径名为参数，返回指定目录的文件句柄的操作，其前提是用户拥有相关文件集系统的访问许可。服务器的位置（IP地址和端口号）和远程目录的文件句柄被发送到VFS层和NFS客户。

图8-10描述了一个具有两个远程安装的文件存储的客户。在服务器1和服务器2上的文件集系统中的*people*和*users*结点被安装到客户本地文件存储的*students*和*staff*结点上。这意味着运行在客户的程序可以通过使用像/usr/students/jon和/usr/staff/ann这样的路径来访问服务器1和服务器2上的文件。

远程文件集系统可以以硬安装和软安装两种安装方式安装到客户计算机上。当一个用户级进程访问硬安装的文件集系统中的一个文件时，进程被挂起直到完成请求，如果远程主机由于某种原因无法使用时，NFS客户模块会继续重复其请求直到其要求被满足。这样，在服务器失效的情况下，用户级进程会一直挂起直到服务器重启，然后继续执行其工作，就好像没有出现过故障。但如果相应的文件集系统是以软安装方式安装的，NFS客户模块会在几次重新请求后返回一个故障消息。构建恰当的程序可以检测到故障，并能执行合适的恢复或报告操作。但许多UNIX应用不检测文件访问操作的故障，当软安装文件集系统失效时，它们可能以一种非预期的方式执行。基于此原因，许多安装只使用硬安装，结果造成NFS服务器在长时段不可用时，程序不能很好地恢复。

327



注意：安装在客户*/usr/students*上的文件系统实际上是位于服务器1上的*/export/people*下的一个子树；安装在客户*/usr/staff*上的文件系统实际上是位于服务器2上的*/nfs/users*下的一个子树

图8-10 在NFS客户端可访问的本地和远程文件系统

路径名转换 每次使用*open*、*creat*或*stat*系统调用时，UNIX文件系统会一步步地将多部分文件路径名转换为*i*-结点引用。在NFS中，路径名不能在服务器上转换，这是因为一个名字可能涉及到客户端的一个“安装点”——拥有多部分名字的不同目录可能驻留在不同服务器上的文件集系统中。所以要分析路径名，由客户以交互方式完成路径名的转换，系统使用远程服务器的独立的*lookup*请求将指向远程安装目录的名字的每一部分转换为文件句柄。

*lookup*操作在给定的目录中查找路径名的一个部分，返回相应的文件句柄和文件属性。前一步返回的文件句柄被当作下一步*lookup*的参数，系统首先将文件句柄中的文件系统标识与客户拥有的远程安装表中的条目进行比较，这样可以知道是否应访问另一个远程安装的文件存储系统。路径转换的每一步结果可以被存储在缓存中，这样可以利用对文件和目录引用的本地特性提高进程执行的效率；用户和程序通常仅访问一个或少量目录中的文件。

自动装载器 为了在客户访问到一个“空的”安装点时动态地安装一个远程目录，人们在NFS的UNIX实现中加入了自动装载器。最初自动装载器是通过在每一个客户计算机上运行一个用户级的UNIX进程来实现的。此后的版本（称为*autofs*）实现于Solaris和Linux的内核中。在此，我们介绍最初的版本。

自动装载器维持一张记录装载点（路径名）和对应的一个或多个NFS服务器列表。在客户机上，它像一个本地的NFS服务器一样工作。当NFS客户模块试图解析包含一个安装点的路径名时，它向本地自动装载器发出一个*lookup*()请求，由自动装载器在它的列表中定位所需的文件集系统，并且向表中所列的服务器发出“试探性”的请求。然后第一个响应的服务器上文件集系统通过正常的安装服务安装到客户端上。被安装的文件集系统通过符号连接连接在安装点上，这样客户在下一次访问中就不需要再向自动装载器发出请求了。除非在数分钟内系统没有对符号连接的引用（这种情况下，自动装载器卸载会远程文件集系统），否则以后都可以按正常的方式来进行文件访问。

新近的内核实现方式以真实的安装取代了符号连接方式，它避免了因为缓存用户级自动装载器使用的临时路径名而引起的一些问题[Callaghan 1999]。

如果在自动装载器列表中存在几个包含同一文件集系统或文件子树副本的服务器，那么自动装载器可以实现一种简单的只读复制。对频繁使用而不经常改变的文件系统，该机制十分有效，如UNIX系统二进制文件。例如，可以在多个服务器上存储*/usr/lib*目录及其子树的副本。当*/usr/lib*的文件被一个客户打开时，系统向所有的服务器发送试探性信息，第一个响应的服务器的文件集系统被安装到客户端上。这种方式提供了一定限度内的容错和负载平衡，这是因为第一个响应的服务器将是正常工作着的，同时它也可能是负载较轻的服务器。

服务器缓存 为了获得良好的性能，可以在客户和服务器计算机上进行高速缓存，它是NFS实现的一个不可缺少的功能。

在传统的UNIX系统中，从磁盘上读取的文件页、目录和文件属性被保留在主存缓冲区缓存中，直到其他页面要求占用该缓冲区的空间。如果一个进程对已在缓存中的页面发送一个读或写的请求，系统不需要再访问磁盘就可以完成此操作。预先读用子预测读访问，并将那些最近使用最多的页面取入内存，而延迟写用于优化写的性能：当一个页面已经被改变时（因为一个写操作），仅当该缓冲区页将被其他页占用时才写到磁盘中。为了防止因系统崩溃引起的数据丢失，UNIX的*sync*操作每隔30s将改变的页面写到硬盘中。这些缓存技术在传统的UNIX环境中都可行，这是因为在传统的UNIX中出用户级进程发出的所有的读和写请求都被发送到实现在UNIX内核中的一个缓存中。这一缓存保持的内容是最新的，同时文件操作不能绕过这一缓存。

仅当NFS服务器被用于其他文件访问时，它才使用服务器上的缓存。使用服务器的缓存保存最近读取的磁盘块不会引起任何一致性问题；但当服务器执行写操作时，系统需要特殊的方法来保证客户可以信任写操作的结果是持久性的，甚至当服务器崩溃时也是如此。在NFS协议第3版中，写操作提供了两种选项（没有在图8-9中表示）：

1. 客户发出的写操作中的数据被存储在服务器的内存缓存中，在给客户发送应答前先将应答写入磁盘。这被称为写透缓存。客户可以相信，当他收到应答时数据已经被持久存储了。
2. 写操作中的数据仅被存储在内存缓存中。当系统接收相关文件的*commit*操作时它被写到磁盘中。仅当客户接收到相关文件的*commit*操作的应答时，客户才能确定数据被持久存储了。标准的NFS客户使用这种操作方式，每当为写操作而打开的文件被关闭时，它发送一个*commit*。

329

*commit*是NFS协议第3版提供的一个附加操作，它用来解决在具有大量写操作的服务器中因写透操作模式引起的性能瓶颈问题。

在分布式文件系统中对写透的需求是第1章讨论的独立故障模式的一个实例——服务器出故障时客户可继续工作，同时应用程序在以前写操作的结果被磁盘存储的假设下，继续执行。这种情况不可能发生在本地文件更新上，因为本地文件系统的错误注定会导致运行在相同计算机上的应用程序进程发生错误。

客户缓存 为了减少传输给服务器的请求数量，NFS客户模块将*read*、*write*、*getattr*、*lookup*和*readdir*操作的结果放在缓存中。客户缓存可能导致在不同的客户结点上存在不同版本的文件或文件部分，这是因为在一个客户上的写操作可能不会引起在其他客户上的同一文件缓存副本的同时更新。因此，要由客户负责用轮询的方式来检查它们所拥有的缓存数据是否是最新的。

一种基于时间戳的方法被用来在使用缓存块之前对缓存块进行验证。每个在缓存中的数据或元数据项被标记上两种时间戳：

- *Tc*是缓存条目上次被验证的时间。
- *Tm*是在服务器上次修改文件块的时间。

当前时间为*T*，如果*T* – *Tc*小于更新的时间间隔*t*，或者当记录在客户端的*Tm*值和在服务器上的*Tm*值相等时（也就是说，在这个缓存条目更新后，服务器上的数据就没有被更新过），那么该缓存条目是有效的。以下是用形式化方法表示的有效性条件：

$$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$$

*t*值的选择是一致性和效率的折衷。一个非常短的更新间隔会导致近似于单一副本的一致性，但因为服务器要频繁地检查Tm_{server}，开销比较大。作为Sun Solaris客户，根据每个文件更新的频度，*t*可在3s~30s之间取值。而对于目录，*t*可在30s~60s之间取值，这说明了目录并发更新的风险比较低。

因为NFS客户不知道文件是否被共享，所以验证程序要施加到所有的文件访问。每次使用缓存条目，系统就执行有效性检查。前半个有效性条件的检查可以不访问服务器。如果其检查结果为真，那么系统不需要检查第二个条件；如果它为假，那么就要从服务器上获得当前的Tm_{server}值（对服务器应用*getattr*调用），并将它与本地的Tm_{client}值进行比较。如果它们相同，那么此缓存条目便被认为有效，并且其T_c值将被更新为当前时间。如果它们不相同，那么缓存保存的数据已在服务器上被更新过，此条目不合法，这会导致产生一个获得服务器上相关数据的请求。 330

有几种方法用来减小对服务器进行*getattr*调用的数量：

- 当客户收到一个新的Tm_{server}值时，将该值应用于所有从这个相关文件派生的缓存条目。
- 在每一个文件操作的结果中捎带上当前文件属性，如果Tm_{server}值改变了，客户便用它来更新缓存中与文件相关的缓存条目。
- 采用适应性算法来设置更新间隔值*t*，对大多数文件而言，可以大量减少调用数量。

验证过程不能保证提供像传统UNIX系统提供的同级别的一致性，这是因为共享一个文件的客户并不总是能够及时知道数据的更新。存在两种来源的时间延迟：写数据后更新在客户内核缓存中的相应数据之前的延迟，以及用于缓存验证的3s的“窗口”。幸运的是，大多数UNIX应用程序并不严格依赖于文件的同步更新，而且这样做几乎没有什么困难。

写操作的处理方式不同。当一个缓存的页被修改后，它被标记为脏的，并通过调度被异步地更新到服务器中。当客户关闭文件或发生*sync*操作时，修改的页被更新到服务器中，如果使用bio-daemon（见下面的介绍），它的更新频度更高。客户缓存并不能提供像服务器缓存一样的持久性保证，但它模拟本地写操作的行为。

为了实现预先读和延迟写，NFS客户需要异步地执行读和写操作。在NFS的UNIX实现中，通过在每一个客户使用一个或多个bio-daemon进程可以实现这一点（*bio*代表块输入输出，*daemon*经常指执行系统任务的用户级进程）。bio-daemon负责执行预先读和延迟写操作。每当发生读请求，就通知bio-daemon，由它请求将这些文件块从服务器传输给客户缓存。在写的情况下，当一个块被客户操作填满时，bio-daemon会将此块发给服务器。当目录发生改变时，相应的目录块会立即发送。

bio-daemon进程改进了性能，以保证客户模块不会因等待服务器端对读返回或者写的确认而阻塞。这些并不是逻辑上的需要，因为在没有预先读的情况下，用户进程的一个读操作会触发对相关服务器的同步请求，当相关的文件关闭时或当客户端的虚拟文件系统执行一个*sync*操作时，用户进程的写操作的结果将被传输给服务器。

其他优化 Sun文件系统是UNIX BSD快速文件系统，它使用8KB磁盘块，相对于以前的UNIX 系统，可以减少对顺序文件的文件系统调用数量。实现Sun RPC的UDP数据包已经扩充到9KB，这使得一个数据包可以容纳一个完整块的RPC调用信息，当顺序读取文件时，这还
331 可以减小网络延迟。NFS第3版没有限制读和写操作处理的最大文件块的大小，当文件块的大小超过8KB并且客户端和服务器都可以处理这类文件块时，它们将进行协商处理。

正如上面提到的，对于活动的文件，客户应该至少每隔3s更新缓存中此文件的状态信息。为了减少由*getattr*请求引起的服务器负载，关于文件或目录的所有操作都隐含*getattr*请求，并且可以在其他操作的结果中捎带上当前的属性值。

用Kerberos实现NFS的安全性 在7.6.2节中，我们介绍了MIT开发的Kerberos认证系统。它已经成为保护企业内部网免受非授权访问和攻击的工业标准。使用Kerberos方案认证客户增强了NFS实现的安全性。本小节将介绍NFS的“Kerberos化”实现。

在NFS最初的标准实现中，用户标识以非加密的数字标识符形式放置在每一个请求中（在以后的NFS版本中，这些标识符可以被加密）。NFS并没有采取其他措施检查客户提供的标识符的真实性。这就意味着必须高度信任客户计算机及其NFS软件的诚实性，而Kerberos和其他基于认证的安全系统的目的就是减少需要信任假设的组件范围。实质上，当在“Kerberos化”环境中使用NFS时，它只能接收那些通过Kerberos身份认证的客户发出的请求。

Kerberos开发者考虑过的一种直接的解决方案，将NFS所需要的凭证的本质转变为成熟的Kerberos票证和认证器。但因为NFS是作为无状态服务器的形式实现的，所以每一个文件的访问请求都是按请求内容处理的，并且每一个请求中必须包含认证数据。这种设计是难以接受的，因为执行必要的加密所需的时间相当长，同时在每个工作站内核中都必须加入Kerberos客户库。

实际中采用了一种混合的方法，即安装用户的内部文件集系统和根文件集系统时，向NFS安装服务器提供用户所有的Kerberos认证数据。认证结果包含用户常规的数字标识符和客户计算机的地址，它们被保存在服务器每个文件集系统的安装信息中（尽管NFS服务器并没有保存与单个客户进程相关的状态，但它还是保存了每一个客户计算机的当前安装信息）。

对每一个文件访问请求，NFS服务器检查用户标识和发送者的地址，仅当它们与存储在服务器中的相关客户的安装信息相符时，NFS服务器才允许访问。这种混合的方法仅涉及到很少的附加开销，而且，因为在同一时刻每一台客户计算机只允许一个用户登录，所以它对于大多数形式的攻击是安全的。MIT采用这种方法设计其系统。最近的NFS实现将Kerberos认证作为几种认证选项之一，并且建议在NFS服务器之外还单独运行Kerberos服务器的场地选择
332 此选项。

性能 最初由Sandberg[1987]给出的性能数字说明：相对于访问本地磁盘文件，使用NFS通常不会导致性能降低。他提出了两个问题：

- 为了从服务器获得时间戳以进行缓存验证，系统要频繁地使用*getattr*调用。
- 因为写透是在服务器端进行的，这导致了写操作性能相对较差。

他注意到：在典型的UNIX工作负载中，写操作相对不多（大约占对服务器调用的5%），因此，除了将大文件写入服务器这种情况外，写透操作的开销是可以容忍的。他所测试的NFS的版本并不包含前面所提到的*commit*机制，而当前NFS版本中的这一机制将明显提高写性能。他给出的结果还表明*lookup*操作大约占服务器调用的50%，这是使用UNIX文件名语义所需的逐步的路径名转换方法导致的后果。

Sun和其他NFS实现者通常使用像LADDIS[Keith and Wittle 1993]这样的基准程序集的改进版本进行测试。当前和过去的一些测试结果可以在[www.spec.org]上找到。那里总结了不同厂家的NFS实现在不同硬件配置上的性能。最近的结果是：吞吐量为每秒5011个服务器操作，平均延迟为3.71ms，最大延迟为8ms（1×450 MHz P Ⅲ CPU，在一个控制器上有34个磁盘，1Gbps的以太网，实时操作系统）。另外一个结果是：吞吐量为每秒29083个服务器操作，平均延迟为4.25ms，最大延迟为7.8ms（24×450MHz IBM RS64-Ⅲ CPU，4个控制器控制的289个磁盘，5个1Gbps的网络，操作系统为AIX UNIX）。这些数字说明：NFS可以为大多数企业内部网的分布式存储需求提供良好的服务，不管负载是支持数百软件工程师开发的传统UNIX，还是从NFS服务器获得数据的Web服务器组。

NFS小结 Sun NFS与我们的抽象模型十分相似。如果NFS的安装服务为每个客户都提供类似的命名空间，那么这种设计便能提供良好的位置透明性和访问透明性。NFS支持异构的硬件和操作系统。NFS服务器的实现是无状态的，它使得客户和服务器在出现故障后不需要进行任何恢复过程就可以继续执行操作。NFS不支持文件或文件集系统的迁移，除非在将一个文件集系统移动到另一新位置后，由客户手工干预，重新配置安装指令。

在每个客户计算机上缓存文件块可以明显改善NFS的性能。为了达到满意的性能，这一点很重要，但是它导致系统偏离了UNIX严格的单一副本文件更新语义。

下面是其他NFS的设计目标以及落实的程度：

- 访问透明性 NFS的客户模块为应用程序提供的对本地进程的接口与它为本地操作系统提供的接口相同。这样UNIX的客户可以使用正常的UNIX系统调用来访问远程文件。用户无需修改现有的程序就能使这些应用程序访问远程文件。 333
- 位置透明性 每个客户通过将一个已安装的远程文件集系统的目录加入自己的本地命名空间，建立一个文件名空间。如果运行在客户端的进程要访问一个远程文件系统，那么包含远程文件系统的计算机结点必须输出该文件系统，并且客户在使用前必须远程安装该文件系统（如图8-10所示）。远程安装的文件系统安装在客户名分级系统中的安装点由客户决定，因此NFS并没有强制实现单一网络范围的文件命名空间——每个客户看到的远程文件集系统都是自己定义的，同一远程文件集系统对不同的客户可能有不同的路径名，为了实现位置透明性，客户可以根据恰当的配置表来建立统一的命名空间。
- 移动透明性 文件集系统（在UNIX中，它是文件树的子树）可以在服务器之间移动，但为了使客户能访问新位置上的文件集系统，要分别更新每一个客户上的远程安装表，所以NFS不能完全达到迁移透明性。
- 可伸缩性 已经公布的性能数据表明NFS服务器可以以一种比较有效的方式处理现实工作环境中的大量负载。通过增加处理器、磁盘和控制器可以提高单个服务器的性能。但达到处理极限时，必须加入新的服务器，同时需要在服务器间重新分配文件集系统。这种策略提高效率的程度受“热点”文件限制——“热点”文件是指被频繁访问从而导致

服务器达到性能极限的文件。若负载超过了这种策略可提供的最大性能，还可采用其他更好的策略：可以使用支持复制可更新文件的分布式文件系统（例如Coda，见第14章），或者如AFS通过缓存整个文件减少协议通信量。我们将在8.5节介绍实现可伸缩性的其他方法。

- 文件复制　只读文件可以复制到多个NFS服务器上，但NFS不支持复制文件的更新。Sun网络信息服务（NIS）是一个单独的服务，可与NFS一起使用，它支持结构为“关键字-值”对的简单数据库的复制（例如，UNIX的系统文件*/etc/passwd*和*/etc/hosts*）。它根据一个简单的主-从复制模型（或者叫主副本模型，将在第14章讨论，该模型在每个场地上提供数据库的部分副本或完全副本）来管理分布式更新和访问复制的文件。NIS为不经常变化的系统信息提供了一个共享库，并且它不要求更新在所有场地同步进行。
- 硬件和操作系统异构性　几乎在所有已知的操作系统和硬件平台上都实现了NFS，有许多文件系统支持NFS。
- 容错　NFS文件访问协议的无状态和幂等特性使客户看到的访问远程文件的故障模式与访问本地文件的故障模式类似。当服务器失效后，它提供的服务会一直挂起直到服务器重启，但一旦重启了服务器，用户级进程就可以从服务被打断的那一点继续执行，它不需要了解服务器出了什么故障（除了访问软安装的远程文件系统外）。实际上，在大多数情况下系统使用的是硬安装，这样将阻止让应用程序处理服务器故障。

  客户计算机或客户用户级进程的故障不会影响它使用的服务器，因为服务器不存储代表客户状态的任何信息。

334

- 一致性　我们已经较详细地描述了更新行为。它提供的语义近似于单一副本语义，它能满足大多数应用程序的要求，但我们不推荐将NFS提供的文件共享用于通信或在不同计算机进程之间的紧密协作。
- 安全性　当将企业内部网连接到因特网时，NFS提出了安全性要求。NFS与Kerberos的结合是一个巨大的进步。最近还有一些进展，例如提供安全RPC实现（RPCSEC_GSS，见RFC 2203[Eisler *et al.* 1997]），用于认证和读写操作中传输的数据的私密性和安全性。许多安装程序还没有部署这些安全性机制，因此它们是不安全的。
- 效率　几种NFS实现的性能测试结果和NFS在产生大量负载的环境中被广泛使用的事实，都说明了NFS协议实现所具有的效率。

8.4 Andrew文件系统

与NFS一样，AFS为运行在工作站上的UNIX程序提供了对远程共享文件的透明访问。AFS用普通的UNIX文件原语访问AFS文件，这使现有的UNIX程序可以不经过修改或重编译就可以访问AFS文件。AFS和NFS是兼容的。AFS服务器拥有“本地”UNIX文件，但在服务器上的文件系统是基于NFS的，因此它使用NFS风格的文件句柄来引用文件，而不是使用*i*-结点号，并且可通过NFS远程访问文件。

AFS主要在设计和实现方面与NFS有区别。区别主要在于可伸缩性这一重要的设计目标。相对于其他分布式文件系统而言，AFS被设计用来满足更多用户使用的需要。AFS实现可伸缩性的主要策略是在客户结点上缓存整个文件。AFS有两个设计特点：

- 整体文件服务　AFS服务器将整个文件和目录的内容都传输到客户计算机上（在AFS-3

中，大于64KB的文件以64KB文件块的形式传输）。

- *整体文件缓存* 当一个文件或文件块的副本被传输到客户计算机上时，它被存储到本地磁盘的缓存中。缓存包含了该计算机最近使用的数百个文件。该缓存是持久的，不会随客户计算机的重启而丢失缓存内容。文件的本地副本用于尽可能满足客户访问远程文件副本的*open*请求。 335

场景 下面是一个简单的场景，用于说明AFS操作：

- 当一个客户计算机上的用户进程向在共享文件空间内的一个文件发出*open*系统调用，并且这一文件不在本地的缓存上时，AFS查找文件所在的服务器的位置，并向其请求传输此文件的一个副本。
- 收到的文件副本被存储在客户计算机的本地UNIX文件系统中，这一文件副本被打开，相应的UNIX文件描述符被返回给客户。
- 客户计算机上的进程在本地文件副本上进行一系列*read*、*write*和其他操作。
- 当客户进程发出一个*close*系统调用时，如果本地文件副本的内容已经被改变，该文件就被传回服务器。服务器更新此文件的内容和时间戳。客户本地磁盘上的副本一直被保留，以供在同一工作站上的用户级进程下一次使用。

下面，我们将讨论AFS的性能，但我们只能根据上面提到的AFS的设计特点来粗略地观察和预测其性能：

- 对于那些不常更新的共享文件（例如那些包含UNIX命令和库的代码的文件）和那些通常只有一个用户访问的文件（例如在用户的主目录及其子目录中的文件），本地缓存的副本可以在相当长的时间内维持其文件副本的正确性——在第一种情况中，是因为文件不被更新，在第二种情况中，是因为如果文件被更新，更新的文件副本会被保存在用户自己的工作站缓存中。这两种类型的文件占被访问文件的总数的绝大部分。
- 本地缓存建立在每个工作站的磁盘上，它可以获得相当大的空间，比如说100MB。通常，对于一个用户使用的工作文件集来说，这一空间是足够大的。为文件工作集提供足够的缓存存储空间，能使在给定工作站上经常使用的文件存储在缓存中以便下次使用。
- 设计策略基于一些假设，这些假设包括UNIX系统中文件的平均大小、最大文件大小以及文件引用的区域性。这些假设是通过对学术和其他环境中一些典型的UNIX负载的研究得到的[Satyanarayanan 1981；Ousterhout *et al.* 1985；Floyd 1986]。其中最重要的结果包括：
  - 文件比较小；大多数小于10KB。
  - 文件的读操作比写操作更常见（通常是6倍以上）。
  - 顺序访问更常见，随机访问不常见。
  - 大多数文件只由一个用户读写。当文件被共享时，通常只有一个用户修改它。 336
  - 文件引用是爆发性的，如果一个文件最近被引用，那么很有可能在不久的将来被再次引用。

  上述研究结果被用于指导AFS的设计和优化，而不是限制用户可用的功能。
- 对于上面第一点所提到的文件类型，AFS能很好地运行。还有一类重要的文件类型，不属于上述文件类型——数据库通常由多个用户共享，并被频繁地更新。AFS的设计者已经明确地从设计目标中排除了数据库的存储功能，他们认为由于不同命名结构具有的约束（即，基于内容的访问）以及对细粒度数据访问、并发控制、更新原子性的需要，很

难设计一个分布式数据库，使它同时也成为分布式文件系统。他们认为应该单独考虑分布式数据库的功能[Satyanarayanan 1989a]。

8.4.1 实现

上面的场景介绍了AFS操作，但留下许多有关其实现的尚未解决的问题，其中比较重要的问题包括：

- 当客户对共享文件空间中的文件发出*open*或*close*系统调用时，AFS怎样获得控制？
- 如何定位包含所需文件的服务器？
- 在工作站上为缓存文件分配什么空间？
- 当文件可能被多个客户更新时，AFS如何保证缓存中的文件副本是最新的？

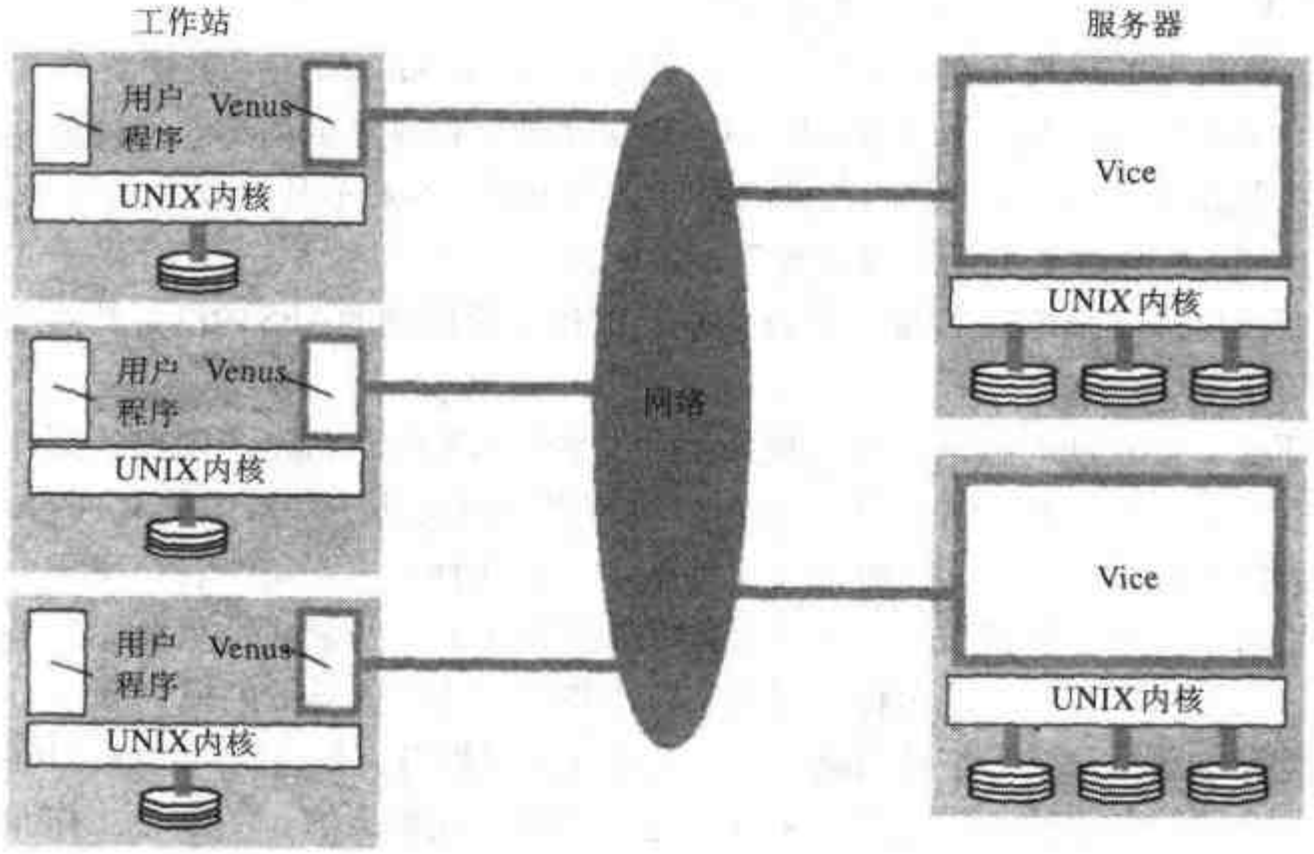


图8-11 在Andrew文件系统中的进程分布

下文将回答这些问题。

AFS由两个软件组件实现，这两个软件组件作为两个UNIX进程*Vice*和*Venus*存在。图8-11给出了Vice和Venus进程的分布，Vice是服务器软件的名字，它作为用户级的UNIX进程运行在每个服务器计算机上，Venus是运行在客户计算机上的用户级进程，相当于我们给出的抽象模型中的客户模块。

运行在工作站上的用户进程可访问的文件或是本地文件或是共享文件。本地文件被当做普通的UNIX文件来处理。它们被存储在工作站磁盘上，并且只有本地用户进程可以访问它。共享文件被存储在服务器上，工作站在本地磁盘上缓存它们的副本。图8-12给出了用户进程所看到的命名空间。它是一个传统的UNIX目录结构，其中有一个包含所有共享文件的子树（称为*cmu*）。将文件名空间划分为本地文件和共享文件两部分会丧失一部分位置透明性，但除了系统管理员以外，一般的用户很难注意到这一点。本地文件仅用于作为临时文件(*/tmp*)以及工作站启动使用的进程。其他标准的UNIX文件（例如通常在*/bin*、*/lib*下的文件）实际上是通过将本地文件目录中的文件符号连接到共享文件空间实现的。用户目录被放在共享空间中，

这使得用户可以从任意一个工作站访问其文件。

工作站和服务器上的UNIX内核是BSD UNIX的修改版本。修改的部分主要是截获*open*、*close*和其他一些文件系统调用，当它们指向在共享名字空间中的文件时，便将它们传递给客户计算机上的Venus进程处理（如图8-13所示）。另外一个对内核的修改是基于性能的考虑，将在后面介绍。

每个工作站的本地磁盘上都有一个文件分区被用作文件的缓存，保存从共享空间内拷贝来的文件缓存副本。Venus进程管理这一缓存，当文件分区满，并且有一个新的文件需要从服务器复制到工作站时，它将最近最少使用的文件从缓存中删除。通常，这些文件分区都足够大，可以容纳数百个平均文件大小的文件，这样，当客户缓存已经包含了当前用户文件和经常被用到的系统文件时，工作站可以基本独立于Vice服务器。

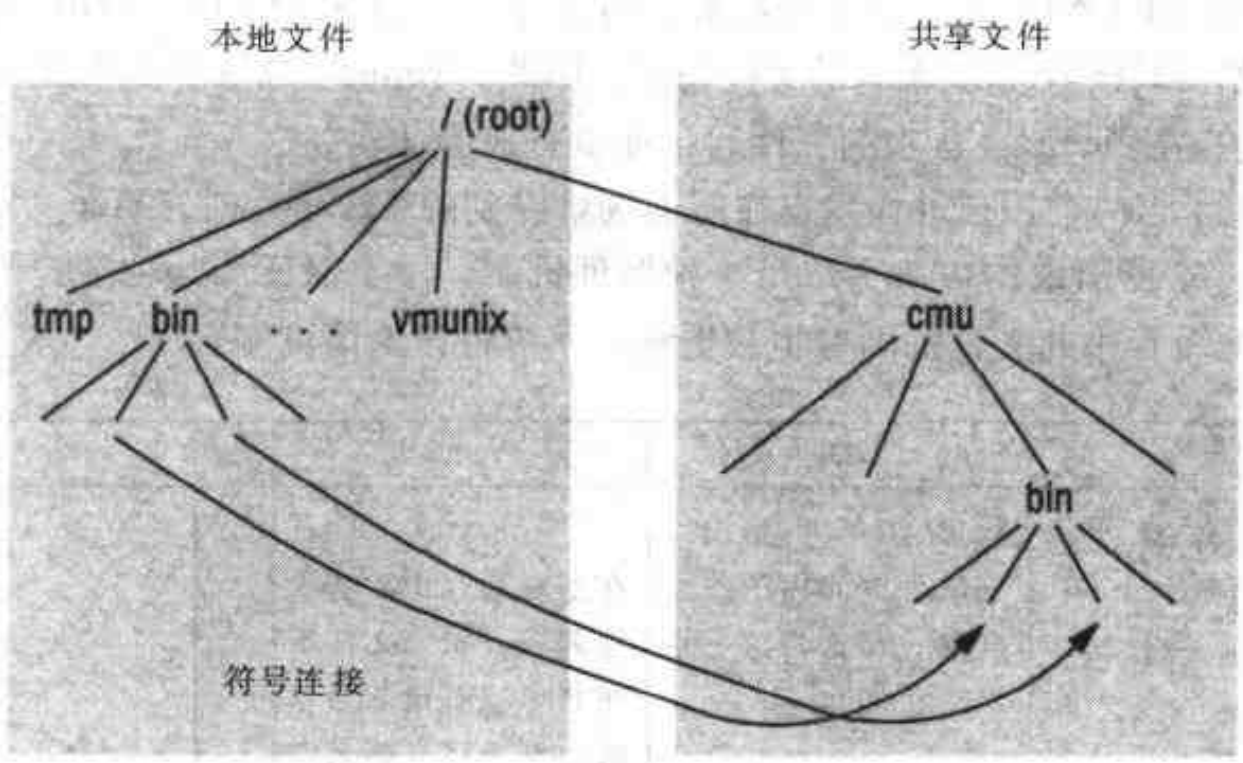


图8-12 AFS客户看到的文件名空间

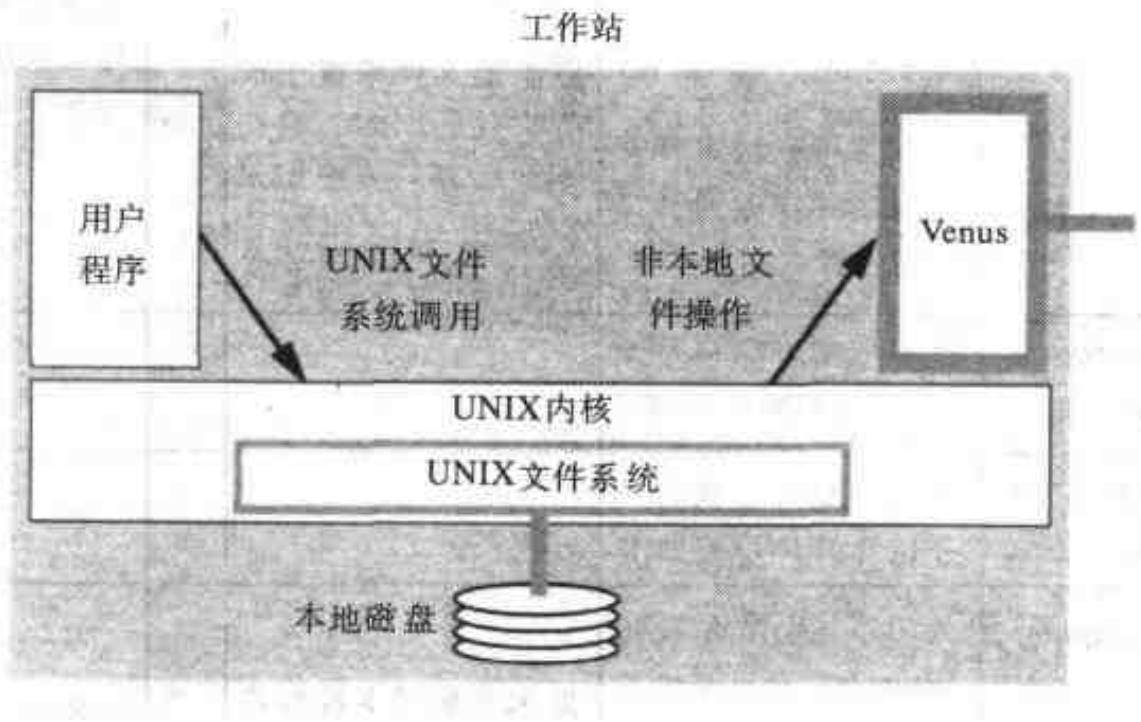


图8-13 AFS中系统调用拦截

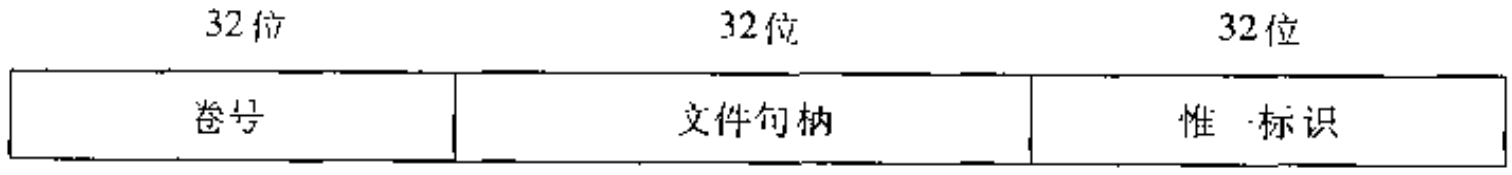
AFS和8.2节描述的抽象文件服务模型在下列方面很相似：

- Vice服务器实现了平面文件服务，工作站上的Venus进程实现了UNIX用户程序所需的层次型目录结构。

- 共享文件空间中的每一个文件和目录由类似UFID的、惟一的、96位的文件标识符（*fid*）标识。Venus进程将客户使用的文件路径名转换为*fid*。

文件被聚集成卷以方便定位和移动。卷通常比UNIX的文件集系统小一些，文件集系统是NFS中的文件分组单位。例如，每个用户的个人文件通常在独立的卷中。其他卷被用来存储系统二进制文件、文档和库代码。

*fid*的表示包括文件所在卷的卷号（参照UFID中的文件组标识）、标识卷中文件的NFS文件句柄（参照UFID中的文件号）以及保证此文件标识不被复用的惟一标识。



337～339

用户使用传统的UNIX的文件路径名来引用文件，但AFS使用*fid*在Venus与Vice进程之间通信。Vice服务器只接收用*fid*表示的文件请求。因此，Venus要将客户提供的文件路径名转换为*fid*，Venus通过在Vice服务器中的文件目录逐步查找信息实现这一点。

图8-14描述了当一个用户进程发出在前面场景提到的系统调用时，Vice、Venus和UNIX内核采取的动作。这里所说的回调承诺是一种保证机制，用于保证当其他客户关闭更新后的同一文件时，本地缓存中此文件的副本也被更新。下节将讨论该机制。

用户进程	UNIX内核	Venus	网络	Vice
open(FileName, mode)	如果*FileName*指向共享文件空间内的一个文件，那么将这一请求传给Venus	在本地缓存中检查文件列表，如果文件不在其中或者没有合法的回调承诺，那么向包含此文件的Vice服务器发送一个请求	→	向工作站传输一个文件副本以及一个回调承诺，记录该回调承诺
	打开这一本地文件并向应用程序返回其文件描述符	在本地文件系统中放置文件的副本，并在本地缓存列表中记录本地名字，同时向UNIX返回其本地名字	←	
read(FileDescriptor, Buffer,length)	在本地副本上执行一个正常的UNIX读操作			
write(FileDescriptor, Buffer,length)	在本地副本上执行一个正常的UNIX写操作			
close(FileDescriptor)	关闭本地副本并通知Venus此文件已经被关闭了	如果本地副本被修改了，向管理此文件的Vice服务器发送此副本	→	替换此文件的内容并向拥有此文件回调承诺的其他客户端发送回调

图8-14 AFS中文件系统调用的实现

8.4.2 缓存的一致性

当Vice为Venus进程提供文件副本时，它同时也提供了一个回调承诺——由管理该文件的Vice服务器发布的一种标识，用于保证当其他客户修改此文件时通知Venus进程。回调承诺和被缓存的文件一起存储在工作站磁盘上，它有两种状态：有效或取消。当服务器执行一个更新文件请求时，它会通知所有收到过它发送的回调承诺的Venus进程，其方式是向每一个进程发送一个回调——从服务器到Venus进程的一种远程过程调用。当Venus进程接收到回调时， 340
它将相关文件的回调承诺标识设置为取消状态。

当Venus处理客户的*open*请求时，它首先检查其缓存。如果所需的文件在缓存中，它便检查其标识。如果其值为取消，那么必须从Vice服务器取得文件的最新副本。如果其值为有效，那么Venus不需要引用Vice就可以打开并使用缓存中的文件副本。

当工作站因为故障或关机而重启时，Venus要在本地磁盘上保留尽可能多的缓存文件，但它不能肯定回调承诺标识是正确的，因为一些回调可能已经丢失了。因此，在重启后初次使用缓存文件或目录之前，Venus要生成一个缓存有效性请求发送给管理该文件的服务器，该请求包含文件修改时间戳。如果其时间戳是当前的，服务器就应答一个有效信息，其标识值被恢复。如果时间戳显示该文件是过期的，那么服务器便应答一个取消信息，其标识就被设置为取消状态。在打开文件之前，如果从文件被缓存开始已经有*T*时间（通常为几分钟）没有与服务器通信，那么必须更新回调。这样做用于处理可能的通信故障，因为通信故障可能导致回调信息丢失。

相对于采用了与NFS相似的基于时间戳的原型（AFS-1）方法而言，这种基于回调的维持缓存一致性的机制可以提供更大的可伸缩性。在AFS-1中，拥有缓存文件副本的Venus进程进行*open*操作时会询问Vice进程，以便判定在本地副本上的时间戳是否与服务器上的时间戳相符合。基于回调的方法具有更大的可伸缩性，因为它只在文件更新时才导致客户和服务器的通信以及服务器上的活动，而时间戳方法会在每一次*open*操作时都导致客户和服务器的交互，即使本地有有效的副本。因为绝大多数文件都不会被并发访问，同时在大多数应用中，读操作比写操作多得多，回调机制使客户和服务器的交互量大量减少。

与AFS-1、NFS以及我们的文件服务模型不同，AFS-2和以后的AFS版本使用的回调机制要求Vice服务器维护一些Venus客户的状态信息。这些与客户有关的状态信息由接收回调承诺的Venus进程列表组成。这一回调列表应在服务器发生故障时还能保持——它们被保存在服务器磁盘上，同时系统使用原子操作更新它们。

图8-15显示了AFS服务器提供的用于文件操作的RPC调用（也就是AFS服务器为Venus进程提供的接口）。

更新语义 缓存一致性机制的目标是，在不明显降低系统性能的前提下，近似实现单一副本文件语义。UNIX文件访问原语的单一副本语义的严格实现要求每一个文件的写操作的结果必须在发生进一步访问操作之前发布到所有在缓存中包含此文件的场地上。在规模较大的系统中，这是不可行的；而回调承诺机制则保持了一种对单一副本语义的较好的近似实现。 341

对AFS-1来说，可以用很简单的方法形式化表示它的更新语义。若客户*C*操作服务器*S*管理的文件*F*，*F*副本的传播要保证满足：

Fetch(fid)→attr,data	返回用*fid*标识的文件属性（状态）和可选的文件内容，同时记录一个回调承诺
Store(fid,attr,data)	更新指定文件的文件属性和文件内容（后者可选）
Create()→fid	创建一个新文件并记录一个回调承诺
Remove(fid)	删除指定的文件
SetLock(fid,mode)	将指定的文件或目录加锁（锁的模式可以是共享的或排它的），30min后，没有删除的锁将过期失效
ReleaseLock(fid)	为指定的文件或目录解锁
RemoveCallback(fid)	通知服务器一个Venus进程已经将文件从其缓存中清除
BreakCallback(fid)	Vice服务器对Venus进程使用这一调用。它取消相关文件上的回调承诺

注意：图中没有显示目录和管理操作（*Rename*、*Link*、*Makedir*、*Removedir*、*GetTime*、*CheckToken*等）。

图8-15 Vice服务接口的主要组件

在成功的*open*操作后： $latest(F, S)$

在失败的*open*操作后： $failure(S)$

在成功的*close*操作后： $updated(F, S)$

在失败的*close*操作后： $failure(S)$

其中$latest(F,S)$表示文件F在客户C的当前值与其在服务器S的值相同，$failure(S)$表示*open*和*close*操作并没有在S上执行（故障可以被客户C检测到），同时$updated(F,S)$表示客户C的文件F的值已经被更新到服务器S上。

对AFS-2来说，对*open*操作的传播保证相对要弱一些，同时相对应的形式化表示要复杂一些。这是因为客户可能会打开一个旧的副本，而该文件已被其他客户更新过了。当因为网络故障等原因，回调信息丢失时，这种情况可能发生。但系统设置了一个客户不知道文件具有较新版本的最大时间T。因此，我们有下列保证：

在成功的*open*操作后: $latest(F,S,0)$ *or* ($lostCallback(S,T)$ *and* $inCache(F)$ *and* $latest(F,S,T)$)，
342 其中$latest(F,S,T)$表示客户所见到的F的文件副本的过期时间不会超过时间T，$lostCallback(S,T)$表示在最近的时间T内从S传递到C的回调信息已经丢失了，$inCache(F)$表示在*open*操作前客户C的缓存中就包含文件F。以上这些形式化表示说明：或者在*open*操作后客户C的文件F的副本是系统中最新的版本，或者回调信息已丢失（因为通信故障）而不得不使用已在缓存中的文件版本；被缓存的文件F的副本过期的时间不会超过T（T是一个系统常量，它表示回调承诺必须被更新的时间间隔。在大多数的系统安装中，T的值被设置为10min）。

为了符合这一目标——提供大范围的、与UNIX兼容的分布式文件服务——AFS并没有提供进一步控制并发更新的机制。上述缓存一致性算法只在*open*操作和*close*操作中起作用。一旦文件被打开，客户可以在不知道其他工作站进程的情况下任意访问和更新本地副本。当文件被关闭后，文件副本被返回给服务器，替代服务器上的当前版本。

如果在不同工作站上的客户对同一文件并发执行*open*、*write*和*close*操作，除了最后*close*的更新结果外，其他更新结果通常会丢失（没有报错）。如果客户需要实现并发，客户必须独立实现并发控制。另一方面，当同一工作站上的两个客户进程打开一个文件时，它们共享同一缓存文件副本，并且依照平常的UNIX方式逐块更新文件。

尽管更新语义依赖于并发进程访问文件的位置，它与标准的UNIX文件系统提供的语义并不完全相同，但它已经足够使大部分已有的UNIX程序正确运行了。

8.4.3 其他方面

UNIX内核修改 我们注意到Vice服务器是运行在服务器计算机上的用户级进程，并且服务器主机被用来提供AFS服务。AFS主机中的UNIX内核被修改过了，这样Vice可以用文件句柄而不是UNIX文件描述符处理文件操作。这是AFS所需要的惟一的内核修改，如果Vice不维护任何客户状态（如文件描述符），它就必须采用这种方式。

位置数据库 每一个服务器包含一个位置数据库的副本，用于将卷名映射到服务器上。当一个卷被移动后，该数据库会出现暂时的不精确，但这是无害的，因为新的信息存储在此卷被移动前所在的服务器上。

线程 Vice和Venus使用非抢占性线程，使客户（其中数个用户进程可能同时访问文件）和服务器能并行地处理请求。在客户端，描述缓存内容和文件卷数据库的表被放在内存中，供Venus线程共享。

343

只读复制 经常被读但很少被修改的文件卷，例如UNIX包含系统命令的*/bin*和*/usr/bin*目录以及包含手册信息的*/man*目录，可以被作为只读的卷副本到多个服务器上。这样操作以后，系统中只存在一个读写副本，所有的更新都指向此副本。在写操作后，更新信息由一个显式的操作过程传播到每个只读副本上。在位置数据库中，对应于被复制的卷，其条目是一对多的形式，并且可根据服务器负载和访问能力为每一个客户请求选择服务器。

批量传输 AFS以64KB的文件块形式在客户和服务器之间传输文件。使用大的数据包有助于减少网络延迟，提高性能。这样，AFS的设计可以优化对网络的使用。

部分文件缓存 如果应用程序只需要读文件的一小部分，却必须将整个文件传输到客户端，这种方式显然是低效的。AFS第3版删除了这种文件操作方式，在保留了AFS协议的一致性语义和其他特征的同时，允许文件数据以64KB块的形式传输以及缓存。

性能 AFS的主要目标是可伸缩性，因此，它在批大量用户环境中的性能具有特别的意义。Howard等[1988]详细介绍了性能测试，测试过程使用了专为AFS开发的AFS基准测试软件，该基准测试软件后来广泛应用于分布式文件系统的测试过程。意料之中的是，缓存整个文件和回调协议可以显著地减少服务器的负载。Satyanarayanan[1989a]说在运行标准基准测试软件的具有18个客户结点的AFS系统中，服务器的负载是40%，而在运行同样基准测试软件的NFS系统，它的服务器的负载是100%。Satyanarayanan将这些性能的提高归功于AFS使用回调来通知客户文件的更新情况而使服务器端负载减少，而在NFS中，系统却是采用超时机制检查缓存在客户的页面的有效性。

广域支持 AFS第3版支持多个管理单元，每一单元具有自己的服务器、客户、系统管理者和用户。每一个单元是一个完全自治的环境，但这些协作的单元可以共同为用户提供一个统一的、无缝的文件名空间。Transarc公司开发了此类系统，并且公布了详细的性能统计结果[Spasojevic and Satyanarayanan 1996]。这一系统被安装在超过150个场地的超过1000台服务器上。统计结果表明，访问一个具有32 000个文件卷200GB数据的示例系统时，缓存命中率为96%～98%。

8.5 最新进展

在NFS和AFS出现以后，分布式文件系统的设计又有了一些进展。在这一节中，我们将介

绍在增强传统分布式文件系统的性能、可用性和可伸缩性方面的一些进展。我们将在本书的
344 其他章节介绍一些更有影响的进展，包括在Bayou和Coda系统中，通过维持读写文件集系统副本的一致性来支持断连和高可用性（14.4.2节和14.4.3节），以及在Tiger视频文件服务器系统中保证实时传输数据质量的高伸缩性体系结构（15.6节）。

NFS的改进 一些研究项目已经解决了单一副本更新语义问题，它们扩展了NFS协议使其包括*open*和*close*操作并加入回调机制使包含失效缓存条目的客户能即时得到服务器的通知。下面将介绍其中的两方面工作，它们的结果说明，在不引起过度复杂性和额外的通信开销下，可以采用这些改进。

Sun和一些NFS开发者致力于在广域网络上提供更易获得、更具使用价值的NFS服务器。尽管Web服务器支持的HTTP协议为整个因特网内的客户提供了有效的和高伸缩性的文件访问方式，但它不适用于应用程序访问大文件或更新部分文件。WebNFS的开发（将在下面介绍）使应用程序可以在因特网内的任一地点成为NFS服务器的客户（通过直接使用NFS协议而不是非直接地通过内核模块的方式）。这种方式连同合适的支持Java和其他网络编程语言的库一起，提供了实现直接共享数据的因特网应用的可能性，例如多用户的游戏或者大规模动态数据库的客户端。

达到单一副本更新语义 NFS的无状态服务器体系结构提高了NFS的健壮性，并使NFS容易实现，但它偏离了精确的单一副本更新语义（多个客户并发对同一文件写的结果不保证与在一个UNIX系统中多个进程并发写本地文件的结果完全相同）。同时它也未使用回调方式，通知客户文件的改变，这样就导致客户为了检查文件是否改变而频繁地调用*getattr*操作。

有两个已开发的研究系统解决了这些缺陷。Spritely NFS[Srinivasan and Mogul 1989, Mogul 1994]是为Berkeley Sprite分布式操作系统[Nelson *et al.* 1988]开发的文件系统。Spritely NFS在其NFS协议的实现中加入了*open*和*close*调用。当本地用户级进程执行对服务器上的文件的打开操作时，客户模块必须发送一个*open*操作。Sprite的*open*操作的参数指定了访问模式（读、写或两者都有）以及当前分别为读操作和写操作而打开文件的本地进程的个数。类似地，当本地进程关闭远程文件时，它必须给服务器发送一个包含读写计数的*close*操作。服务器将这些计数和客户的IP地址和端口号一起记录在一个打开文件表中。

当服务器接收到一个*open*调用时，它在打开文件表中找出其他打开同一文件的客户，并且将回调信息发送给这些客户，告知它们修改其缓存策略。如果这一*open*操作是写模式的，并且如果其他客户正在以写模式打开此文件，这一操作便会失败。在一个客户写文件时，系统会通知其他以读模式打开文件的客户，告知它们该文件的所有本地缓存副本都已经失效。

对于指定以读模式进行的*open*操作，服务器会将回调信息发送给正在对此文件写的客户，
345 告知它停止进行缓存（使用严格的写透模式），并且它会告知所有读此文件的客户停止将此文件缓存（这样所有的本地读调用导致对服务器的一个请求）。

这种方法使得文件服务能维持UNIX的单一副本更新语义，但它需要在服务器上记录一些客户相关状态。缓存文件的写操作效率也有所提高。如果服务器在易失存储器中保存客户相关状态，这些状态信息就易受服务器崩溃的影响。Spritely NFS实现了一个恢复协议，通过查询最近在服务器上打开文件的客户列表来恢复整个打开文件表。客户列表存储在磁盘上，并且很少被更新，这是一种“悲观”策略——客户列表中包含的客户比在系统崩溃时打开文件的客户多。出故障的客户可能也在打开文件表占据一个条目，但当这一客户重启时，该条目

会被删除。

当将Spritely NFS与NFS第2版进行比较时，前者的性能有一定程度的改进。这来源于对写文件缓存的改进。NFS第3版至少达到了同样程度的改进，但Spritely NFS项目的结果表明，在不明显损失性能的情况下实现单一副本更新语义是可能的，虽然这样做客户和服务器模块会比较复杂，并且需要一个恢复机制以便在服务器崩溃后能恢复原有状态。

NQNFS NQNFS（Not Quite NFS）项目[Macklem 1994]的目标和Spritely NFS类似——在NFS协议中加入更精确的缓存一致性并且通过更好地使用缓存来改进性能。NQNFS服务器维持与Spritely NFS相似的关于客户打开文件的相关状态，但它使用租借（5.2.6节）作为服务器崩溃后的恢复。服务器为客户持有打开文件设置一个租期上限。如果客户需要在超出此时间后继续持有该打开文件，它必须续租。当写请求发生时，系统使用与Spritely NFS相似的回调机制通知客户刷新其缓存，但如果客户不应答，服务器在响应新的写请求之前会一直等待，直到租期满为止。

WebNFS Web和Java小程序的出现使NFS开发小组和一些机构及相关人员认识到：一些因特网应用可以直接访问NFS服务器，这样做不会产生与模拟标准的NFS客户中包含的UNIX文件操作相关的开销。

WebNFS（在RFC2055和2056中描述[Callaghan 1996a, 1996b]）旨在使Web浏览器、Java程序和其他程序与NFS服务器直接进行交互，访问使用公共文件句柄“公布”的文件，访问在公共根目录下的文件。这种使用模式避免了安装服务和端口映射服务（见第5章的描述）。WebNFS客户通过一个约定的端口号（2049）与服务器交互。为了根据路径名访问文件，它使用一个公共文件句柄发出一个*lookup*请求。这一公共文件句柄有一个约定的值，由服务器上的虚拟文件系统专门解释这个值。由于广域网的高延迟性，系统使用多组件的*lookup*操作来查找单个请求中的多部分路径名。

这样在较少的安装开销下，WebNFS使客户能够访问远程场地NFS服务器上的文件。它也提供了访问控制和认证，但在许多情况下，客户只需要对公共文件进行读访问，此时，认证选项可以被关掉。在支持WebNFS的NFS服务器上，为了读一个文件，系统需要建立一个TCP连接和两个RPC调用——一个多组件的*lookup*操作和一个*read*操作。NFS协议不限制所读数据块的大小。 346

例如，一个气象服务可能在它的NFS服务器上公布一个文件，该文件包含一个经常变化的气象数据的大数据库，它的URL为：

nfs://data.weather.gov/weatherdata/global.data

可以用Java或其他任何支持WebNFS过程库的语言构建一个显示气象图的交互式WeatherMap客户程序。客户仅需要读取/weatherdata/global.data文件中构建用户请求的气象图必需的信息，而使用HTTP的类似应用程序需要将整个数据库文件传输到客户，或者需要使用特殊的服务器程序，才可以访问气象图服务器。

NFS第4版 在本书出版时，新版本的NFS协议正在开发中。RFC 2624[Shepler 1999]和Brent Callaghan的书[Callaghan 1999]中都描述了NFS第4版的目标。与WebNFS相似，它的目标是使NFS适用于广域网应用和因特网应用。它将包含WebNFS的特征，但新的协议的引入也可能对其进行更彻底的改进（WebNFS在更改那些没有在协议中加入新操作的服务器时会受到限制）。

开发NFS第4版的工作组希望利用过去十几年中文件服务器设计领域的一些研究成果，例如使用回调或租借机制来维持一致性。NFS第4版将通过允许文件系统透明地从一个服务器转移到另一个服务器来支持服务器故障后的即时恢复。以类似于在Web上的使用方式使用代理服务器可以改善系统的可伸缩性。

AFS的改进 我们已经提到过DCE/DFS，它是包括在开放软件基金会的分布式计算环境中的[www.opengroup.org]，基于Andrew文件系统的分布式文件系统。DCE/DFS的设计超越了AFS，特别是它达到缓存一致性的方法。在AFS中，仅当服务器接收到对已经被更新的文件的*close*操作请求时，系统才生成回调。DFS使用了一种与Spritely NFS和NQNFS相似的策略，一旦文件被更新就生成回调。为了更新一个文件，客户必须从服务器获得一个*write*标志，用于在文件中指定允许客户更新的一系列字节。在请求*write*标志后，具有同一文件副本（用于读）的客户会收到取消操作的回调。可使用其他类型的标志获得缓存文件属性和其他元数据的一致性。所有标志都有一个生存期，在期满了以后，客户必须续延标志的生存期。

存储组织结构的改进 存储在磁盘上的文件数据的组织结构有很大的改进。分布式文件
347 系统需要支持更多的负载，具有更高的可靠性，这种需求推动了这方面的研究工作，也导致了文件系统性能的大幅度提高。这些研究工作的主要成果如下：

- *廉价磁盘冗余阵列（RAID）* 这种存储模式[Patterson *et al.* 1988，Chen *et al.*1994]将数据分解成多个大小固定的块，这些数据块与冗余的错误更正代码一起存储在跨越多个磁盘的“条带”上，这样在磁盘失效后，数据块可以被完全重建，系统可以继续操作数据。RAID的性能比单个磁盘的性能好，这是因为组成块的条带可以被并发地读写。
- *日志结构的文件存储（LFS）* 与Spritely NFS一样，该项技术起源于Berkeley Sprite分布式操作系统项目[Rosenblum and Ousterhout 1992]。他们注意到在文件服务器中，当使用更多内存作为文件缓存时，缓存命中率的提高导致了相应良好的读文件性能，但是写文件性能仍然没有提高，这是由于将单个数据块写入磁盘以及更新元数据块（包含文件属性和文件指针集的数据块，即*i*-结点）时存在延迟。

  LFS的解决方案是在内存积累若干写操作，然后将它们提交给划分了大的、连续的、定长的段的磁盘。这些段被称为日志段，因为它严格按修改顺序来存储数据和元数据块。一个日志段是1MB或更大，存储在一个磁道上，这样就省掉了与写单个块相关的磁盘头延迟。已修改数据和元数据块的最新副本总是要进行写操作，因此要求维护一个指向*i*-结点的动态映射（在具有持久备份的内存中）。还需对旧的数据块进行无用单元回收，其方法是将“活动”的块放置在一起以便为日志段的存储留出连续的空间。无用单元回收操作是比较复杂的，它由一个叫无用单元回收器的组件以后台活动的方式执行。根据仿真的结果，现在已开发了一些比较复杂的无用单元回收算法。

  尽管有这些额外的开销，整体性能改进还是比较显著的：Rosenblum和Ousterhout测试得到的写吞吐量高达磁盘可用带宽的70%，而在传统的UNIX文件系统中，这一数字小于10%。日志结构也简化了服务器崩溃后的恢复。Zebra文件系统[Hartman and Ousterhout 1995]作为初始LFS工作的后继者，将日志结构的写操作和分布式RAID方法结合起来——日志段被划分为包含错误更正代码的节并且被写到不同网络结点的磁盘上。在写大文件时，能获得4~5倍于NFS系统的性能，而对于小文件，性能提高要小一些。

新的设计方法 高性能交换网络的开发（例如ATM和高速交换以太网）促使研究人员投

人一部分精力研究如何在有许多结点的企业内部网上按高伸缩性和容错方式提供分布式文件数据的持久性存储系统，把管理元数据和客户请求服务的职责与读和写数据的职责相分离。下面列出两个在此方面的进展。 348

这些方法比我们在前面章节中介绍的集中式服务器方法有更好的伸缩性。它们通常需要合作提供服务的计算机之间有高级别的信任度，因为它们通常使用低级别的协议在持有数据的结点间通信（有些类似于一个“虚拟磁盘”API）。因此它们的访问常常只限于单个本地网络。

xFS 加州大学伯克利分校的一个小组设计了一个无服务器网络文件系统体系结构并开发了一个原型实现即xFS[Anderson *et al.*1996]系统。有3个因素促成了这一方法：

1. 快速交换局域网使局域网上的多个文件服务器可以并发地向客户传输大量的数据。
2. 不断增加的访问共享数据的需求。
3. 基于集中式文件服务器的一些限制。

关于第3点，他们提出这样一个事实：构建高性能的NFS服务器需要相对昂贵的多个CPU、多个磁盘和多个网络控制器，并且存在划分文件空间的限制——需要将不同文件集系统的共享文件安装到不同的服务器上。他们还指出一个中央的服务器代表一个单点故障。

xFS是“无服务器”的，它在单个文件的粒度上将文件服务器处理职责分散到局域网上一组可用的计算机上。存储的分布职责独立于管理和其他服务职责：xFS实现了一个软件的RAID存储系统，它将文件数据分散存储到多个计算机硬盘上（在这种意义上它是第15章中描述的Tiger视频文件系统的先驱），并使用了与Zebra文件系统相似的日志结构技术。

管理每个文件的职责可以被分配到任意一个支持xFS服务的计算机上。通过被复制到每一个客户和服务器上的叫做管理映射表的一个元数据结构可以实现这一策略。文件标识包含一个作为此管理器映射表索引的字段，并且此映射表中的每一个条目都标识了当前负责管理相应文件的计算机。其他一些元数据结构用于管理日志结构文件存储和条带化磁盘存储，它们与在日志结构和RAID存储系统中的元数据结构相似。

已经构造了一个xFS的初步原型，并且进行了性能评估。进行性能评估时，这一原型还是不完全的——崩溃恢复机制还没有完全实现并且日志结构的存储机制也缺少一个无用单元回收模块恢复被陈旧的日志和压缩文件占据的空间。

对这一初步原型进行性能评测使用的是连接在高速网络上的32台单处理器的Sun SPARC工作站和一台双处理器的Sun SPARC工作站。运行在32台工作站上的xFS和运行在一个双处理器工作站上的NFS和AFS进行了比较。32台服务器的xFS的读写带宽超过运行在一个双处理器上的NFS和AFS的读写带宽近10倍。当使用标准的AFS基准测试软件时，xFS和NFS、AFS的性能差距并不明显。总之，结果表明：xFS的高度分布处理和存储体系结构为分布式文件系统达到更好的可伸缩性提供了一个有前景的方向。 349

Frangipani Frangipani是在DEC系统研究中心（现在是Compaq系统研究中心）开发和部署的高伸缩性的分布式系统[Thekkath *et al.* 1997]。它的目标与xFS十分相似，并且和xFS一样，其设计也是将持久存储职责与其他文件服务活动相分离。但Frangipani的服务被划分为完全独立的两个层次。其较底层由Petal分布式虚拟磁盘系统[Lee and Thekkath 1996]提供。

Petal在交换局域网上的多个服务器的磁盘间提供了一个分布式的虚拟磁盘抽象。这一虚拟磁盘抽象通过存储数据的多个副本来应对大多数的硬件和软件错误，它还通过为数据重定

位来自动平衡服务器上的负载。UNIX磁盘驱动器通过标准的块输入输出操作访问Petal虚拟磁盘，所以Petal虚拟磁盘可用于支持大多数文件系统。Petal增加了10%～100%的磁盘访问延迟，但缓存策略可以使其读写吞吐量至少与底层的磁盘驱动相同。

Frangipani服务器模块运行在操作系统内核中。与xFS中一样，管理文件和相关任务的职责（包括对客户端提供的文件锁服务）被动态地分配到主机上，并且所有的机器看到的是一个统一的文件名空间，它们可以一致地（使用近似的单一副本语义）访问共享的可被更新的文件。数据以日志结构和条带格式存储在Petal虚拟磁盘中。Petal的使用使Frangipani减少了管理物理磁盘空间的需要，从而可以实现一个较简单的分布式文件系统。Frangipani可以模拟几种已有的文件服务的服务接口，包括NFS和DCE/DFS。Frangipani的性能至少与UNIX文件系统的Digital实现一样好。

8.6 小结

分布式文件系统的主要设计问题包括：

- 有效地使用客户缓存以便获得与本地文件系统相同甚至更好的性能。
- 当更新文件时，维持此文件的多个客户副本的一致性。
- 在客户和服务器出错后的恢复。
- 读写具有不同数据量的文件的高吞吐量。
- 可伸缩性。

分布式文件系统在有组织的计算中被广泛使用，其性能已成为许多优化的主题。NFS包
350 含一个简单的无状态协议，借助于对协议的细小改进、优化的实现和高性能的硬件支持，NFS一直保持它在分布式文件系统的技术统治地位。

AFS显示了一种相对简单的体系结构的灵活性，它使用服务器状态减少维持客户缓存一致性的耗费。AFS在许多情况下的性能好于NFS。最近，AFS使用了跨越数个磁盘的数据条带和日志结构写操作，这些研究进展进一步改进了AFS的性能和可伸缩性。

当前最先进的分布式文件系统具有高伸缩性，可提供跨越局域网和广域网的优良性能，维护单一副本文件更新语义，并且能容错和进行故障恢复。未来的需求包括支持经常有断连操作的移动用户，支持自动重集成以及能满足持久存储和传输多媒体数据流以及其他实时数据需要的服务质量保障。在第14章和第15章我们将介绍对这些需求的解决办法。

练习

8.1 为什么在我们的平面文件服务或目录服务的中没有*open*操作或*close*操作。我们的目录服务模型中的*Lookup*操作和UNIX的*open*操作有哪些区别？

8.2 列出客户模块用于模拟UNIX文件系统接口的方法，请使用我们的文件服务模型。

8.3 写出一个*PathLookup*(*Pathname*, *Dir*) →*UFID*程序实现对UNIX式路径名的*Lookup*操作，这一程序基于我们的目录服务模型。

8.4 为什么UFID必须在多个可能的文件系统上保持惟一？UFID的惟一性是怎样保证的？

8.5 Sun NFS在何种程度上偏离了单一副本文件更新语义？请构造这样一个场景：共享同一文件的两个用户级进程可以在一个UNIX主机上正常操作，但当它们运行在不同主机上时便会观察到不一致性。

8.6 Sun NFS的目标是通过提供一个独立于操作系统的文件服务来支持异构的分布式系统。一个非UNIX的操作系统的NFS服务器的实现者所必须采取的关键决策是什么？为了实现NFS服务器，其底层的文件系统要遵守什么限制？

8.7 NFS客户模块必须拥有哪些用户级进程的数据？

8.8 使用图8-9中的NFS RPC调用，（i）不使用客户缓存；（ii）使用客户缓存；给出在这两种情况下，UNIX的*open*()和*read*()系统调用的客户模块实现。

8.9 请解释为什么NFS的最初实现中的RPC接口可能是不安全的。NFS第3版通过使用加密弥补了这个安全漏洞。密钥是如何保密的？密钥有足够的安全性吗？ 351

8.10 在一个RPC调用访问一个硬安装文件系统超时后，客户模块并没有将控制返回到最初发出调用的用户级进程，为什么？

8.11 NFS的自动安装器是如何改进NFS的性能和可伸缩性的？

8.12 处理存储在NFS服务器上的包含5部分的路径名（例如，/usr/users/jim/code/xyz.c），需要多少个查询调用？执行逐步转换的原因是什么？

8.13 为了在基于NFS的文件系统上获得访问透明性，客户计算机上的安装表的配置应该满足哪些条件？

8.14 当客户发送一个对共享文件空间内的一个文件的打开和关闭系统调用时，AFS如何获得控制？

8.15 将访问本地文件时的UNIX更新语义和NFS及AFS的更新语义进行比较。什么情况下客户可以意识到其差异？

8.16 AFS是如何处理可能丢失回调信息这一问题的？

8.17 AFS设计有哪些特点使得它比NFS有更大的可伸缩性？假设需要加入服务器，那么其伸缩性有什么限制？有哪些最近的研究成果提供了更好的可伸缩性？ 352

第9章 命名服务

9.1 简介
9.2 命名服务和域名系统
9.3 目录服务和发现服务
9.4 实例研究：全局命名服务
9.5 实例研究：X.500目录服务
9.6 小结

本章将命名服务作为一个独特的服务来介绍，通过使用命名服务，客户进程可以根据名字获取资源或对象的地址等属性。被命名的实体可以是多种类型，并且可由不同的服务管理。例如，命名服务经常用于保存用户、计算机、网络域、服务以及远程对象的地址以及其他细节。除命名服务外，我们还将介绍目录服务与发现服务，这些服务可以根据属性寻找特定服务。

我们以因特网域名服务为例，介绍了命名服务的基本设计要点，例如服务可以识别的名字空间的结构与管理、命名服务支持的操作等。

我们也考察了命名服务的实现，涉及一系列问题，如名字解析过程中的服务器导航、为提高性能与可用性对名字数据进行缓存与复制等。

本章包含了两个实例研究：全局命名服务（GNS）、X.500目录服务以及LDAP。

9.1 简介

在分布式系统中，名字用于指称计算机、服务、远程对象、文件以及用户等广泛的资源。命名在分布式系统设计中其实是一个非常基础的问题，尽管它极易被忽略。名字方便了通信与资源共享。在计算机系统需对数个资源中的某个资源进行操作的情况下，就必须需要一个名字。例如，访问特定Web页面需要一个以URL形式表示的名字。只有在所有进程一致地命名了计算机系统管理的特定资源的情况下，进程才能共享这些资源。同样，在分布式系统中，用户之间必须能够给出对方名字，才能互相通信，例如，利用电子地邮件址通信。

名字并不是识别对象的惟一方法，描述性的属性也可用于识别对象。有时客户并不知道它们寻找的实体的名字，却知道一些用于描述这些实体的信息。也有可能客户需要一个服务（而不是实现它的一个特定实体），却只知道该服务具有的一些特征。

这引出了命名服务以及相关的目录服务与发现服务。命名服务在分布式系统中，可向客户提供被命名的对象的数据。而目录服务和发现服务提供的是满足某个描述的对象的数据。我们将域名服务（DNS）、GNS以及X.500作为实例，描述了设计与实现这些服务的方法。首先我们将考察名字、属性等一系列基本概念。

名字、地址及其他属性

任何请求一个资源的进程必须拥有该资源的名字或标识，例如文件名*/etc/passwd*，URL *http://www.cdk3.net/*以及因特网域名*dcs.qmw.ac.uk*等，都是我们可以阅读的名字。而标识这个

术语有时指只有程序才能够解释的名字。远程对象引用以及NFS文件句柄都是标识的例子。选择标识的一个重要指标是软件存储与查询标识的效率。

Needham[1993]将纯粹的名字与其他名字区分开来。纯粹的名字仅仅是未解释的比特模式。而非纯粹的名字包含了被命名的对象的信息，特别地，它们可能会包含对象的地址信息。在使用纯粹的名字前必须对其进行查询。与纯粹的名字完全相反的是对象的地址，该值识别了对象的位置而不是对象本身。地址通常可用于访问对象，但对象有时会被重定位，因此，地址并不足以作为标识方法。例如，用户的电子邮件地址通常会因用户在不同的组织或因特网服务提供商之间移动而改变，因此，邮件地址本身并不能长时间地指称特定用户。

当一个名字被翻译成被命名的资源或对象的数据时，我们称一个名字被解析，解析对象名的目的通常是为了在对象上调用一个动作。名称与对象之间的关系称为绑定。通常，名字
354 被绑定到被命名对象的属性，而不是对象本身的实现。属性是对象特性的值，与分布式系统相关的一个重要属性就是对象的地址。例如：

- DNS将域名映射到主机的属性上，包括主机的IP地址、条目的类型（例如，是邮件服务器还是其他主机）以及主机条目的有效时间。
- X.500目录服务用于将人名映射到邮件地址、电话号码等一系列属性上。
- CORBA命名服务与交易服务在第17章介绍。该命名服务将一个远程对象名映射到它的远程对象引用上，而交易服务在将一个远程对象名映射到它的远程对象引用上的同时，也给出了用户可以理解的对象的一些属性。

注意，“地址”常被看作另一个可用于查找的名字，或者它包含了一个可查找的名字。IP地址被用于查找以太网地址等网络地址。类似地，Web浏览器以及邮件客户使用DNS来解释URL中的域名与邮件地址。图9-1给出了一个URL的域名部分，它首先通过DNS被解析成IP地址，然后通过ARP被解析成一个Web服务器的以太网地址，而URL的最后一部分被Web服务器上的文件系统解析，用于寻找相关文件。

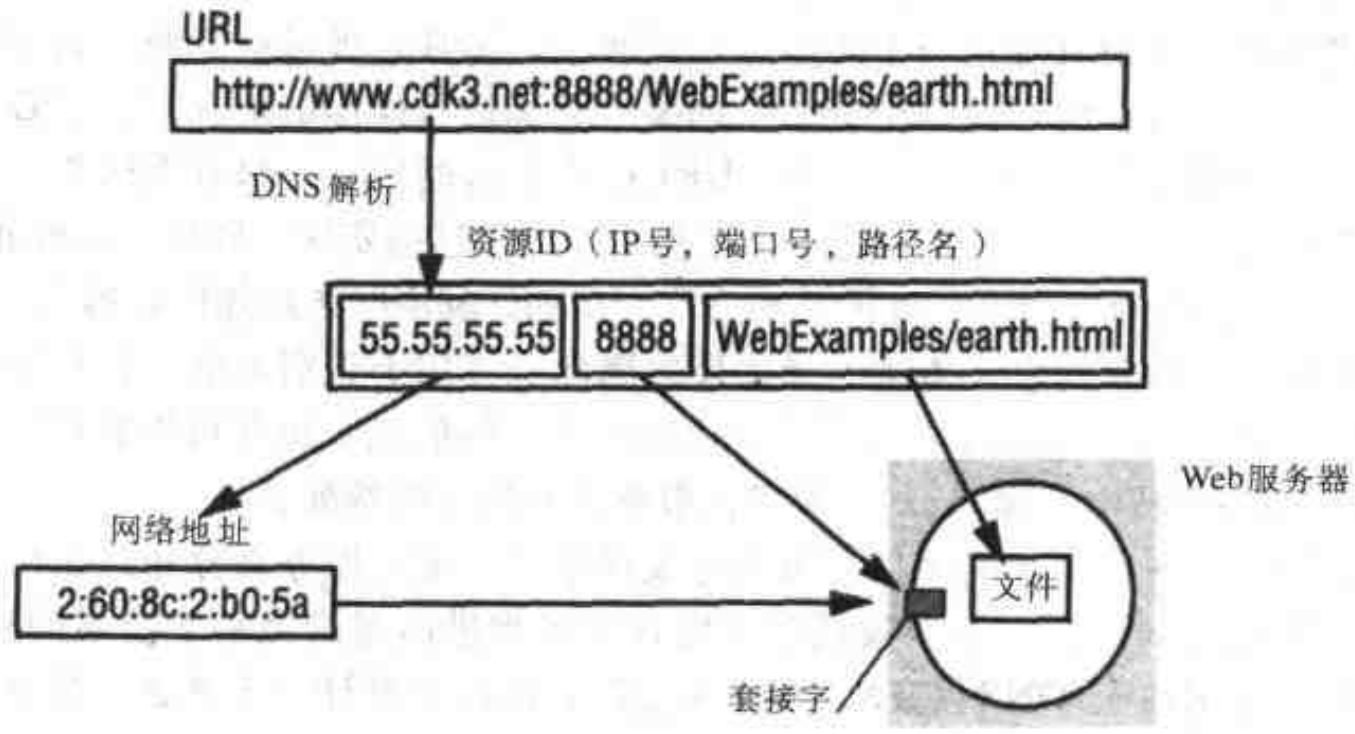


图9-1 使用组合命名域从URL访问资源

名字与服务 分布式系统使用的许多名字专用于特定的服务。客户程序使用名字请求特定服务在被命名的对象或资源上执行某个操作。例如，当删除一个文件时，需要将文件名传送给文件服务；如果需要向特定进程发送信号，该进程的标识会被传送到进程管理服务。除客

户基于共享对象通信的情况外，上述名字仅用于管理命名对象的服务的上下文中。 355

在分布式系统中，名字也需要能指称超出单个服务范畴的实体。这些实体的主要例子包括用户（有专有名、登录名、用户标识以及电子邮件地址）、计算机（有主机名，如bruno、bronwyn）以及服务本身（如文件服务、打印服务）。在基于对象的中间件中，名字指向提供了服务或应用的远程对象。注意上述名字必须是人们能阅读并理解的，因为用户与系统管理员需要使用分布式系统中的主要组件与配置；程序员需要使用程序中的服务；而用户需要通过分布式系统相互通信并讨论系统不同部分有哪些可用的服务。考虑到因特网的连接无处不在，这些命名需求范围上可能是全球性的。

统一资源标识符 我们在第1.3节介绍了URL，它是识别Web资源的主要方法。事实上，URL是统一资源标识符（URI）的一个特殊类型[URI]。

URL具有一些重要的特性，如它对无限的Web资源具有可伸缩性，它是资源访问的高效句柄。使用资源的URL（包含了一个DNS机器名与该机器上的路径名）信息，可以很方便地访问到资源。因为URL实质上是Web资源的地址，它具有这样一个不足：如果一个Web资源被删除或移动，例如，从一个站点移动到另一个站点，通常对于包含旧URL的资源，会发生“悬挂链接”的情况。若用户单击一个悬挂链接，则Web服务器或者会响应“资源未找到”，或者提供当前占据了该URL的另一个资源。

URI的另一个主要类型是*统一资源名称*（URN），URN试图解决悬挂链接问题，提供访问Web资源更丰富的模式。URN的思想是：每个Web资源具有一个永久的URN，即使该资源可能被移动。资源的拥有者将资源名以及资源当前的URL注册到一个URN查询服务，此服务可通过给定的URN找到资源的URL。当资源移动时，拥有者会注册一个新的URL。URN的形式为*urn:nameSpace:nameSpace-specificName*。例如，*urn:ISBN:0-201-62433-8*可用于指标准ISBN命名机制下的名为0-201-62433-8的书。*urn:doi:10.555/music-pop-1234*可用于指在数据对象标识[www.doi.org]机制下的，出版者为10.555，出版物名为*music-pop-1234*的对象。类似地，诸如*urn:dcs.gormenghast.ac.uk:TR2000-56*的URN，在理论上可被用于获取技术报告*TR2000-56*的最新URL，在此之前，*TR2000-56*被注册到Gormenghast大学计算机科学系的*dcs.gormenghast.ac.uk*上的查询服务。

统一资源特征（URC，也称统一资源引用）是URN的子集。URC是Web资源的一个描述，它包含了资源的一组属性，例如：‘author=Leslie Lamport’，‘keywords = time,…’。URC用于描述Web资源，也可用于寻找匹配其属性规范的Web资源。下面两节我们将讨论根据名字和描述查询资源的服务。 356

9.2 命名服务和域名系统

*一个命名服务*存储了一个或多个命名上下文——有关用户、计算机、服务以及远程对象等对象的文本名字与属性之间的绑定的集合。命名服务支持的主要操作是名字解析——即根据一个给定的名字查询相应的属性，我们将在9.2.2节中描述名字解析的实现。其他操作，如生成新的绑定、删除绑定、列表绑定的名称以及增删上下文，也都是命名服务必须支持的操作。

名字管理从其他服务中分离出来的主要原因在于分布式系统的开放性，开放性带来了下列动机：

- *一致性* 不同服务管理的资源使用相同的命名机制会带来方便，URL是一个很好的例子。

- 集成性　在分布式系统中并不总能预测共享的范围。在某些情况下，必须共享（从而命名）在不同管理域中创建的资源。如果没有一个公共的命名服务，管理域会使用完全不同的命名约定。

通用命名服务的需求　命名服务最初非常简单，因为它仅被设计用来满足名字与单个管理域中的地址绑定的需求，而单个管理域对应于单个LAN或WAN。然而，网际互联以及分布式系统的不断扩展带来了一个更大的名字映射问题。

Grapevine[Birrell *et al.* 1982]是最早的可伸缩的多域命名服务之一。它的设计，从名字的数量以及可处理的负载来看，至少可在两个数量级范围内伸缩。

数字设备公司的系统研究中心 [Lampson 1986] 开发的全局命名服务是Grapevine的改进，全局命名服务具有更宏大的目标，包括：

- 处理任意数量的名字，为任意数量的管理组织提供服务　例如，系统尤其需要能处理全世界计算机用户的电子邮件地址。
- 长生命周期　在生命周期中，名字集的组织、实现服务的组件都会发生变化。
- 高可用性　很多系统依赖命名服务，命名服务一旦崩溃，则系统无法工作。
- 故障隔离　局部故障不会带来整个服务的崩溃。
- 不信任容忍性　一个大的开放系统很难拥有被系统中所有客户都信任的组件。

Globe 命名服务[van Steen *et al.* 1998]与handle系统[www.handle.net] 都将目标集中于命名服务在大规模对象情况下的可伸缩性。而第3章中介绍的因特网域名系统（DNS）的设计，在
357 对象的数量方面，没有上面的系统理想，然而它仍然被广泛地使用。DNS命名了因特网上的对象（实际中的计算机）。为提供理想的服务，它极大地依赖于名字数据的复制与缓存。我们假设在DNS以及其他命名服务的设计中，无需像缓存的文件副本那样严格保证缓存一致性，因为名字修改并不是那么频繁，同时使用一个过期的名字翻译副本通常可被客户程序探测到。

本节用DNS为例子，讨论命名服务的主要设计问题，然后对DNS进行详细实例研究。

9.2.1 名字空间

一个名字空间是一个特定服务所能够识别的所有有效名字的集合，所谓有效就是服务将试图查询之，即使该名字并不对应于任何对象——未被绑定。名字空间需要语法定义，例如，名称‘Two’不应是一个UNIX进程的名字，然而数字“2”却可能是；同样，名字“…”作为计算机的DNS名是不可接受的。

在UNIX文件系统中，或在组织机构的层次中，如因特网域名服务，名字有一个内部结构，表示它们在层次型名字空间中的位置，或者名字可从一组平面的数字标识或符号标识集合中选出。层次型名字空间最重要的好处在于名字的每个部分总是相对于一个独立的上下文进行解析，而相同的名字在不同的上下文中有不同的含义。对于文件系统，每个目录代表了一个上下文。因此*/etc/passwd*是一个具有两个成分的层次型的名字。首先“etc”相对于“/”或根目录上下文被解析，而第二个部分“passwd”，相对于上下文“/etc”被解析。名字“*/oldetc/passwd*”可能有不同的解析结果，因为该名字的第二个部分在不同的上下文中被解析。类似地，同样的名字“*/etc/passwd*”，在两台不同的机器上、基于两个不同的上下文，会解析到不同的文件上。

层次型名字空间可以是无限的，所以系统可以无限增长。平面名字空间通常是有限的，

它们的大小由名字所允许的最大的长度决定。如果在平面名字空间中名字的长度没有限制，那么名字空间也可以是无限的。层次型名字空间另一个潜在的好处是可由不同的人管理不同的上下文。

第1章介绍了URL的结构，URL名字空间包括诸如*../images/figure1.jpg*的相对名。在*../images/figure1.jpg*中，浏览器将该路径所在的主机名与服务器目录名作为该URL的主机名与服务器目录名。

DNS名通常被称为域名——它们是与UNIX绝对路径名相似的字符串。DNS名的例子有：*bruno.dcs.qmw.ac.uk*（计算机名）、*dcs.qmw.ac.uk*、*com*与*purdue.edu*（后两个例子是域）。

DNS名字空间具有层次结构：一个域名包括了一个或多个字符串，这些串通常称为名字成分或标签，并且由分隔符“.”分隔。尽管为管理方便，DNS名字空间的根结点有时用“.”指代，但事实上，域名的开头与结尾并没有分隔符。名字成分是不包含“.”符号的非空的可打印字符串。一般来说，名字的前缀指的是从名字开头开始包含了零个或多个完整名字成分的部分。例如，在DNS中，*dcs*与*dcs.qmw*均是*dcs.qmw.ac.uk*的前缀。DNS名对大小写不敏感，因此，*ac.uk*与*AC.UK*的含义相同。

DNS服务器不识别相对名，所有的名字都基于全局的根结点。然而，在实际实现中，客户软件会维持一个域名表，在解析单成分域名时，会将该列表自动附加到单成分域名之后。例如，域*dcs.qmw.ac.uk*中的名字*bruno*有可能指的是*bruno.dcs.qmw.ac.uk*。在试图解析*bruno*时，客户软件将默认域名*dcs.qmw.ac.uk*附加在*bruno*后。如果解析失败，则附加其他默认域名。最后，*bruno*作为绝对名传递到根结点被解析。另外，具有多于一个成分的名字通常不做预处理，而是作为绝对名被送到DNS。

别名 遗憾的是，带一个或两个成分以上的名字在输入与记忆上都不太方便。通常，别名与UNIX风格的符号连接相似，允许用一个方便的名字替代复杂的名字。DNS允许的别名使用方法是，定义一个域名来表示另一个。提供别名的原因是为了提供透明性。例如，别名通常用于指定一个运行了Web服务器与FTP服务器的机器名。名字*www.dcs.qmw.ac.uk*是*copper.dcs.qmw.ac.uk*的别名。这样的好处在于客户可以通过一个不涉及特定机器的通用的名字指定一个Web服务器，如果Web服务器被移动到另一台计算机，修改DNS数据库中的别名是惟一要做的工作。

命名域 命名域是仅通过一个总的管理权威管理有关该域中的名字指派问题的名字空间。该权威机构完全控制哪些名字可以被绑定到域中，它也可以将这个任务委托出去。

DNS的域是域名的集合。语法上，一个域的名字是该域中所有域名的公共后缀，除了公共后缀这个特点，域名很难与其他名字如计算机名区分开来。例如，*qmw.ac.uk*是一个包含了*dcs.qmw.ac.uk*的域。注意“域名”这个术语可能会令人迷惑，因为仅有一部分域名标识了域。甚至计算机名可与域名相似，例如，*yahoo.com*是域*yahoo.com*中的一台Web服务器的名字。

域的管理可以被移交到子域中。域*dcs.qmw.ac.uk*——英国玛利女王与韦斯特菲尔德学院的计算机系——可以包含任何该部门想要的名字。但*dcs.qmw.ac.uk*域名本身需要得到学院权威机构的认同。类似地，*qmw.ac.uk*必须得到已注册的权威机构*ac.uk*的认同。

管理命名域，以及管理由命名服务使用的权威数据库以使该数据库保持最新状态，这两个职责是密切相关的。通常，属于不同命名域的命名数据被存储在不同的命名服务器上，这些命名服务器由不同权威机构管理。

组合与定制名字空间 DNS提供了一个全局的、同构的名字空间，在DNS中，无论是哪台计算机上的哪个进程进行查询，同一个名字总是指向同一个实体。与之相反，某些命名服务允许不同的名字空间——甚至是异构的名字空间——嵌入其中。而有些命名服务允许定制名字空间，以满足个别组织、用户甚至进程的需要。

359

合并 在UNIX与NFS中安装文件系统的实践（见8.3节）提供了一个名字空间的一部分被方便地嵌入到另一个名字空间的实例。此刻我们考虑如何合并两个（或更多）完整的UNIX文件系统，这两个系统在两台计算机名分别为*red*与*blue*的计算机上。每台计算机有自己的根，具有重叠的文件名。例如，*/etc/passwd*在*red*上指的是一个文件，在*blue*上指的是另一个文件。合并两个文件系统的最明显的方法是：用一个“超级根”替代原来每台计算机的根，然后将每台计算机的文件系统安装到该超级根目录下，称为*/red*与*/blue*。这样用户与程序可以将上文中的文件分别称为*/red/etc/passwd*与*/blue/etc/passwd*。然而，这个新的命名规范本身就会导致在两台计算机上仍然使用旧名字*/etc/passwd*的程序发生故障。一个解决方法是，将每台计算机旧根下的内容仍然保留，而将两台计算机上已装载的文件系统*/red*与*/blue*嵌入（假设这样做不会带来旧根下的名字冲突）。

结论是我们总是可以通过构造更高一级的根上下文而合并名字空间，但这样做会带来向后兼容问题。而修补兼容性问题又会给我们带来混合名字空间问题，而且不得不麻烦地在两台计算机的用户之间翻译旧的名字。

异构性 分布式计算环境（DCE）的名字空间[OSF 1997]允许嵌入异构名字空间。DCE名可以包含接合点，接合点的概念与NFS和UNIX（见8.3节）中的安装点相似。只不过它允许安装异构的名字空间。例如，考虑完整的DCE名*/.../dcs.qmw.ac.uk/principals/Jean.Dollimore*。名字的第一部分*/.../dcs.qmw.ac.uk/*标识了一个称为单元的上下文。下一个成分是一个接合点。例如，接合点*principals*是一个包含了安全主体的上下文，在该上下文中，查询名字的最后一个成分*Jean.Dollimore*。类似地，在*/.../dcs.qmw.ac.uk/files/pub/reports/TR2000-99*中，接合点*files*是一个对应于文件系统目录的上下文，在该上下文中，查询名字的最后成分*pub/reports/TR2000-99*。接合点*principals*与*files*是异构名字空间的根，它们由异构命名服务实现。

定制 从上面嵌入NFS文件系统的例子中可看出，在有些情况下，用户偏爱于构造自己的名字空间，而不是共享单个的名字空间。文件系统的安装使用户可以引入存储在服务器上的共享文件，而其他名字依然指向本地未共享的文件，并且可以被自动管理。然而，即使是同一个文件，如果从不同的计算机访问，会被安装到不同的安装点上，从而有不同的名字。在不共享整个名字空间的情况下，用户必须在不同的计算机间翻译名字。

定制的另一个动机是同一个名字在不同的计算机上可以指向不同的文件。例如，名字*/bin/netscape*，在理论上可以绑定到奔腾计算机的x86二进制格式程序上，也可以绑定Sun计算机的SPARC二进制格式程序上。这种将相同名字映射到不同文件的方式，使得包括这些名字的脚本程序可在不同的机器配置下正常工作。

Spring命名服务[Radia *et al.* 1993]提供了动态构造名字空间以及选择共享个人命名上下文

360

的能力。与上面例子不同的是，同一台计算机上的两个不同的进程也可以有不同的命名上下文。Spring命名上下文是在分布式系统中可以共享的头等对象。例如，一个在计算机red上的用户试图运行计算机*blue*上的一个程序，该程序寻找诸如*/etc/passwd*的文件路径，但是，该路径应该解析到*red*文件系统上的文件，而不是*blue*上的。在Spring中，可以通过将*red*的本地命

名上下文的引用传递给*blue*并将其作为程序的命名上下文，达到此目的。Plan 9[Pike *et al.* 1993]也允许进程具有自己的文件系统名字空间。Plan 9的一个新颖特色（该特色也可在Spring中实现）是它的物理目录可以被排序和被合并为单个逻辑目录。其效果为，当返回属性时在单个逻辑目录中被查询的名字会在后续的物理目录中被查询，直到查询得到匹配。这样做的好处是，在搜寻程序或库文件的过程中，用户无需提供一组路径。

9.2.2 名字解析

名字解析通常是一个迭代的过程，通过该过程，名字被反复地送到命名上下文中。一个命名上下文或者直接地将一个给定的名字映射到一组简单属性中（例如，一个用户的属性），或者将之映射到一个更深的命名上下文中，同时将一个派生名送到该上下文。在解析一个名字时，该名字首先被送到某个初始命名上下文中；随着更深的命名上下文以及派生名的输出，解析过程不断进行。在9.2.1节的开始，我们以*/etc/passwd*为例阐述了该过程，在此例中，首先“etc”被送到上下文/，然后“passwd”被送到上下文*/etc*。

解析过程的另一个迭代特性是别名的使用。例如，当请求DNS服务器解析诸如*www.dcs.qmw.ac.uk*之类的别名时，服务器首先将该别名解析到另一个域名（在该例中，为*copper.dcs.qmw.ac.uk*），然后，这个域名进一步地被解析到一个IP地址上。

通常，别名的使用可能会导致名字空间带有循环，这种情况下，解析过程将永不终止。有两种解决方法，一是一旦到达解析的阈值，就放弃解析；第二个是让管理员禁止任何会带来循环的别名。

命名服务器与导航 诸如DNS这样的命名服务，存储了一个巨大的数据库，并由众多用户访问，通常不会将所有的名字信息放在单个服务器上。如果只使用一台服务器存放名字信息，那么这台服务器会成为一个瓶颈以及故障的临界点。任何重负载的命名服务都应该使用复制以提高可用性。我们看到，DNS规定数据库的任何一个子集都必须被复制到至少两个不会同时失效的服务器上。

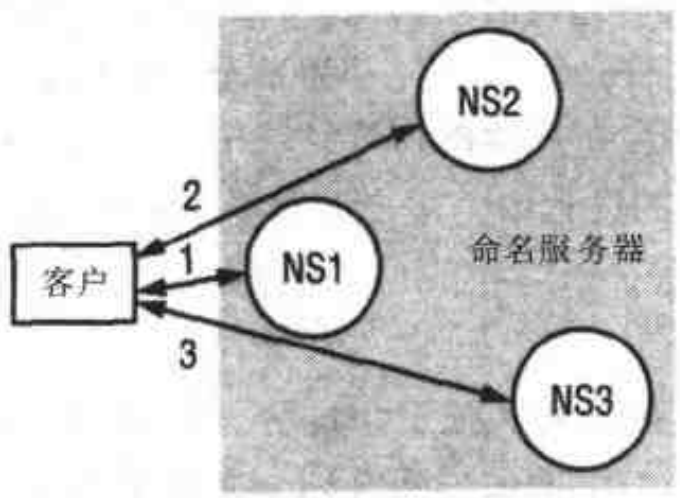
我们曾提到，属于一个命名域的数据通常被存储在该域的权威管理机构管理的本地命名服务器上。尽管在某些情况下，一个命名服务器会存储多个域的数据，但通常情况下，还是会根据域的不同将数据分区到不同服务器上。我们看到，在DNS中，大多数条目是有关本地计算机的。当然，也有些命名服务器存储更高的域（例如*yahoo.com*、*ac.uk*）和根信息。

数据的分区意味着本地命名服务器若没有其他命名服务器的帮助，将无法回答所有的询问。例如，在*dcs.qmw.ac.uk*域中的命名服务器将不能提供域*cs.purdue.edu*中的计算机的IP地址，除非该计算机的域名被缓存——此时，该域名必然不是第一次被访问。 361

为解析一个名字，从一个或多个命名服务器上定位命名数据的过程被称为导航。客户端的名字解析软件代表客户进行导航。它们在解析名字的过程中，会与命名服务器进行必要的通信。它们可能作为库代码链接到客户端，例如DNS的BIND实现（见9.2.3节）或是Grapevine[Birrell *et al.* 1982]。在X.500中使用不同的方法，在另一个独立的进程中提供命名解析，而该进程可被该计算机上的所有客户进程共享。

DNS支持迭代导航模型（见图9-2）。在解析一个名字时，客户将该名字送到本地命名服务器，该服务器试图解析之。若本地命名服务器有这个名字，则立刻返回结果。如果没有，则它会建议另一个能提供帮助的服务器。解析在另一个服务器上继续，进一步的导航过程会

一直继续，直到该名字被定位或是被发现并未与任何名字绑定。



（为解析一个名字，客户反复地与命名服务器NS1~NS3联系）

图9-2 迭代导航

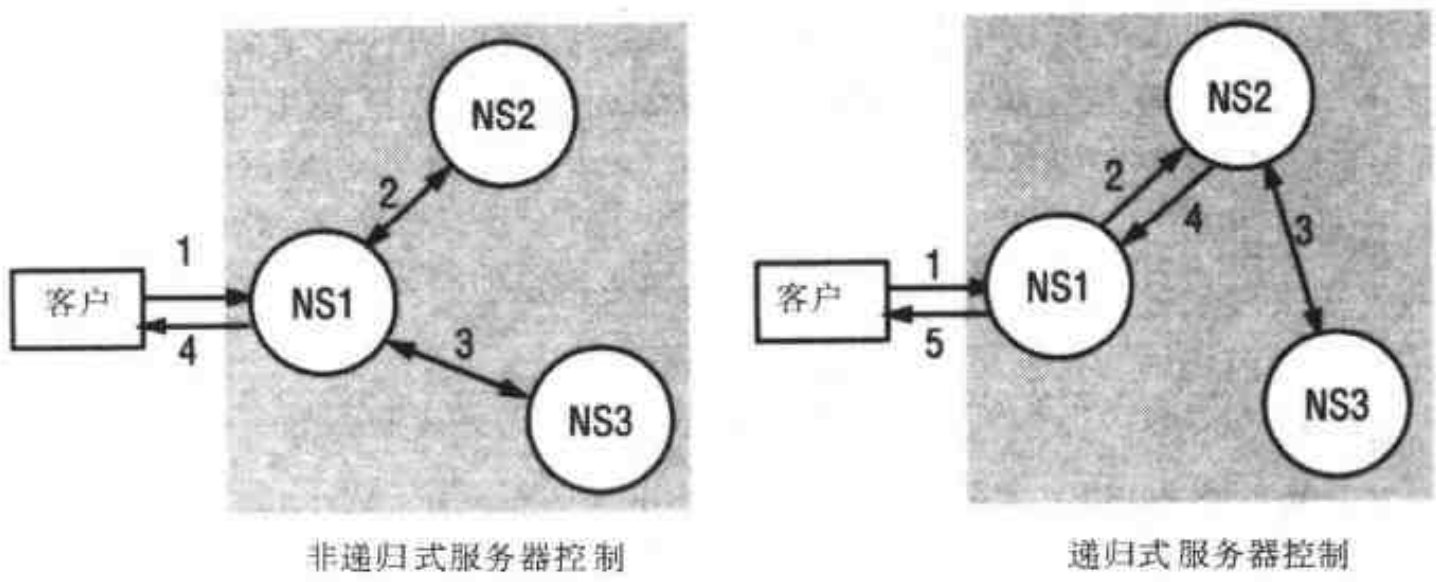
由于DNS设计用于容纳数百万个域的条目，可被大量的客户访问，因此，所有的查询都从根服务器开始是不可行的，即使在根服务器被大量复制的情况下。将DNS数据库划分到不同服务器上的策略是：大多数查询可在本地被满足，其余的查询无需单独解析名字的每一个部分。9.2.3节将详细描述DNS解析名字的机制。

NFS在解析文件名时，以成分为基础，也使用了迭代导航（见第8章）。这是因为在解析文件名时，文件服务可能会遇到符号连接。必须在客户的文件系统中解释符号连接，因为符号连接可能指向另一个服务器目录中的文件。客户必须确定是哪个服务器，因为仅有客户知道安装点。

在组播导航中，客户向命名服务器组组播需解析的名字以及需要的对象类型。只有包含
362 了命名属性的服务器会响应该请求。遗憾的是，如果名字确实并未绑定到任何对象，则该请求不会得到任何响应。Cheriton与Mann[1989]描述了一个基于组播的导航机制，在该机制中，服务器组中包含一个独立的服务器来处理名字未被绑定的情况。组播导航也被用到发现服务中（见9.3节）。

迭代导航模型的替代方案是，由命名服务器协调名字的解析过程，并将结果返回给用户代理。Ma[1992]区分了非递归式服务器控制的导航以及递归式服务器控制的导航（如图9-3所示）。在非递归式服务器控制的导航中，任何命名服务器都可被客户选中。该服务器使用上文描述的方式，通过组播或迭代与其他服务器通信，如同它是一个客户一样。在递归式的服务器控制的导航中，客户依然只与一个服务器打交道。如果服务器未存储该名字，则服务器与另一个存储了该名字（更长）前缀的服务器联系，此服务器接着试图解析该名字，该过程递归地继续下去，直到名字被解析。

若一个命名服务跨越了不同的管理域，则在一个管理域中执行的客户可能会被禁止访问另一个管理域上的命名服务器。此外，命名服务器甚至会被禁止探测另一个管理域中的命名服务器上的命名数据的部署。这样，客户控制的导航与非递归式服务器控制的导航两种方式均不能适用，此时必须使用递归式服务器控制的导航方式。经授权的命名服务器向指定的命名服务器请求命名服务数据，该服务器由另外的管理者部门管理，它返回相应的属性，而并不暴露命名数据库的不同部分在何处存储。



命名服务器NS1代表客户与其他命名服务器通信

图9-3 非递归式和递归式服务器控制的导航

缓存 在DNS以及其他命名服务中，客户端的名字解析软件以及服务器维护了一个以往名字解析结果的缓存。当客户发出一个名字查询请求时，客户端的名字解析软件就查询它的缓存，如果该缓存包含了上次查询该名字得到的一个最近的结果，那么将该结果返回给客户；否则，客户软件将着手在某个服务器上寻找结果，而服务器又有可能返回缓存在其他服务器中的数据。

缓存是命名服务性能的关键，即使在命名服务器崩溃的情况下，缓存也可以帮助维护命名服务器以及其他服务器的可用性。其作用非常清晰，即通过节约与命名服务器的通信时间 363
提高响应速度。缓存可用于导航路径上消除高层的命名服务器——特别是根服务器，同时在一些服务器失效的情况下，缓存允许解析过程继续进行。

因为名字数据的改变比较少，从而客户端的名字解析程序的缓存在命名服务中得到广泛使用，并且特别成功。例如，计算机或服务地址的信息很可能会在几个月或几年内不变。然而，一个命名服务也可能返回过时的属性信息，例如，在解析过程中得到过时的地址。

9.2.3 域名系统

域名系统是一个命名服务设计，主要在因特网上使用它的命名数据库。它主要由Mockapetris[1987]设计，用于替换原有的因特网命名机制。在原有的机制中，所有的主机名和地址都保存在一个中央主文件中，需要这些信息的计算机以FTP方式下载[Harrenstien *et al.* 1985]，此机制有以下3个缺点：

- 计算机数量众多时，缺乏可伸缩性。
- 本地组织需要管理自己的命名系统。
- 需要通用的命名服务——而不是仅查询计算机地址的命名服务。

DNS命名的对象主要是计算机，IP地址作为其属性存储。本章中涉及的命名域在DNS中简称为域。然而原则上，所有的对象都能被命名，DNS的体系结构可以支持各种各样的实现，在DNS实现中，组织和部门可以管理自己的命名数据。因特网DNS绑定了上千万个名字，而全世界的机器都基于该DNS查询。任何名字均可被任何客户解析，这是由名字数据库的层次划分、命名数据的复制以及缓存来实现的。

域名 DNS可供多种实现方式使用，每种都可以拥有自己的名字空间，但是实际上只有

一种方法运用最广，即在因特网上使用的命名方式。因特网DNS名字空间既有组织性划分也有地域性划分，名字中最高级的域位于右端。现今因特网上广泛使用的顶级组织域名（也称作通用域）包括：

com——商业化组织
edu——大学以及其他教育机构
gov——美国政府机构
mil——美国军事组织
net——主要的网络支持中心
org——上文未提及的组织
364 *int*——国际组织

此外，每个国家拥有自己的域名：

cn——中国
us——美国
uk——英国
fr——法国
…——……

每个国家，尤其是美国之外的其他国家，使用自己的域来区分国家内各个组织。以英国为例，有*co.uk*和*ac.uk*域，分别对应于*com*和*edu*（*ac*代表“academic community”）。但值得注意的是，尽管有相似的后缀，例如*uk*，一个诸如*doit.co.uk*的域，也能在Doit Ltd（一家英国公司）的西班牙办事处拥有数据，换句话说，地域式的域名仅仅是一种习惯用法，事实上，域名完全独立于其物理位置。

DNS 查询 因特网DNS主要用于简单的主机名解析与电子邮件主机查询，具体如下：

- *主机名解析* 通常，应用程序使用DNS将主机名解析为IP地址。例如，当一个Web浏览器获得一个包含了*www.dcs.qmw.ac.uk*的URL之后，它将发出DNS查询，并获得相应的IP地址。正如第4章指出的，浏览器使用HTTP协议与占据了特定IP地址保留端口号的Web服务器通信。FTP以及SMTP的工作方式类似，例如，当FTP客户程序获得域名*ftp.dcs.qmw.ac.uk*后，它会发出一个DNS查询以获得该域名的IP地址，然后使用TCP/IP协议在一个保留端口上与服务器联系。*www*、*ftp*以及*smtp*等名可能是运行服务的计算机的实际域名的别名。也存在不使用别名的例子，考虑这样的情况，一个用户使用*telnet*程序与域名为*jeans-pc.dcs.qmw.ac.uk*的主机联系，*telnet*查询获得相应的IP地址后，与服务器的默认端口联系。
- *邮件主机定位* 电子邮件软件使用DNS将域名解析为邮件主机的IP地址——邮件主机用于接收相应域的邮件。例如，当需要解析*tom@dcs.rnx.ac.uk*时，使用地址*dcs.rnx.ac.uk*查询DNS，类型指定为“mail”。如果存在对应的邮件服务器，那么DNS会返回可接收*dcs.rnx.ac.uk*的邮件的主机的域名列表（有时可选择返回IP地址）。DNS可能会返回多于一个的域名，这样，当主邮件服务器因某种原因不可用时，邮件软件可以尝试其他服务器。DNS对每个邮件主机均会返回一个整型的优先级，表示尝试邮件主机的顺序。

有些安装版本也包括了其他类型的查询，但远没有上面用的多，它们是：

- *反向解析* 一些软件需要通过IP地址获得域名。这与正常的主机名查询恰恰相反。对于
365 接收查询的命名服务器，仅当IP地址在自己的域中，才会应答。

- 主机信息　DNS可以存储主机域名相应计算机的体系结构类型和操作系统信息。有人建议不应实现该选项，因为它为那些试图在未授权情况下访问计算机的潜在黑客提供了有用的信息。
- 已知服务　有些命名服务器提供已知服务，给定计算机域名，命名服务返回该计算机运行的一组服务（例如telnet、FTP）以及用于获得服务的协议（即因特网的UDP或TCP协议）。

原则上DNS可以存储任意属性。一个查询通过域名、类别与类型三者而定义。因特网上域名的类别是IN。查询的类型定义了是否需要一个IP地址、一个邮件主机、一个命名服务器或其他信息。特殊域*in-addr.arpa*存储了IP地址，可以用于反向查询。类别属性用于分辨不同类型的命名服务，例如，区分因特网命名数据库与其他实验阶段的DNS命名数据库。对一个给定的数据库会有一组类型定义，因特网数据库的类型定义见图9-5。

DNS命名服务器　通过分区、复制以及在需要地点的最近处缓存命名数据库这些方法的综合解决可伸缩性问题。DNS数据库分布在一个逻辑服务器网络上。每个服务器存储了命名数据库的一部分——主要是本地域的数据。大多数查询涉及到本地域的计算机，并且由该域中的服务器给出回答。然而每个服务器记录了其他命名服务器的域名与地址，这样可满足对本域以外对象查询的需要。

DNS命名数据被划分为区域，一个区域包含了下列数据：

- 除那些由较低层的权威机构管理的子域内的数据外，一个域里名字的所有属性数据。例如，一个区域可能包含了属于玛丽女王与韦斯特菲尔德学院——*qmw.ac.uk*——的数据，而包含各系，如计算机科学系*dcs.qmw.ac.uk*的数据较少。
- 至少两台命名服务器的名称与地址，这些服务器提供了该区域的权威数据，而且数据的版本被认为是最新的。
- 一些命名服务器的名字，这些服务器存储了被委托的子域的权威数据，以及给出了服务器IP地址后的一些“粘合”数据。
- 区域管理参数，例如，某些参数用于管理区域数据的缓存与复制。

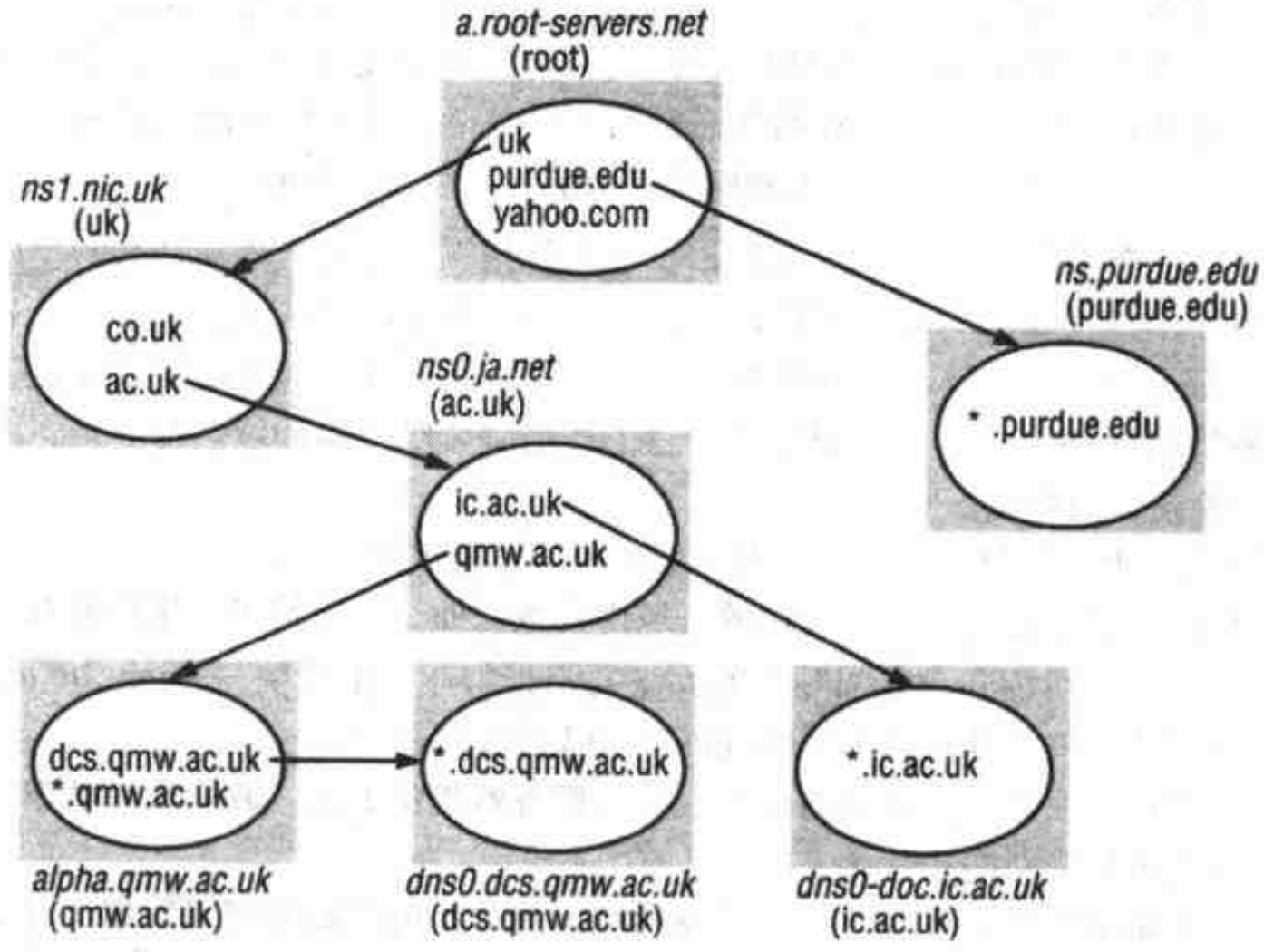
一个服务器可以拥有零个或多个区域的权威数据。为了在单个服务器失效的情况下，名字数据依然可用，DNS体系结构规定每个区域必须至少在两台服务器上复制。

系统管理员将一个区域的数据送入一个主控文件中，此文件是该区域权威数据的源头。有两种服务器被认为可提供权威数据：主服务器直接从本地主控文件中读取区域数据。从服务器从主服务器下载区域数据。它们周期性地通信，以检验从服务器上的版本是否与主服务器上的一致。若从服务器上的版本过期，则主服务器将最近的版本发送给它。管理员将从服务器检查过期的频率作为一个区域参数来设定，通常它的值是一天一次或两天一次。 366

任何服务器均可缓存其他服务器的数据，以避免在解析名字再次需要同一数据时，还要与那些服务器联系。这样做的附带条件是当客户收到缓存数据时，需被告知数据是不可信的。区域中的每个条目有一个存活期值。当一个非权威服务器缓存来自权威服务器的数据时，它会记录存活期然后会做到仅在存活期范围内向客户提供缓存数据；对于过了存活期后的查询，服务器需要重新与权威服务器联系，核对它的数据。这是一个有用的特性，它减少了网络流量，同时保留了系统管理的灵活性。在预料到属性改变会较少时，它们可以给定相当长的存活期。如果管理员知道属性可能很快就会改变，那么他/她将相应地减少属性的存活期。

图9-4给出了DNS数据库的部分安排，注意到在实际中，诸如*a.root-servers.net*这样的根服务器除了保存一级域名外，也会保存多个级别的域条目。这样在域名被解析时，可以降低导

航的次数。根命名服务器拥有顶级命名服务器的权威条目，也同时是*com*、*edu*等常用顶级域的权威命名服务器，然而根命名服务器不是国家域的命名服务器。例如，*uk*域目前有7个命名服务器，其中一个被称为*ns1.nic.net*。这些命名服务器知道英国二级域如*ac.uk*与*co.uk*的命名服务器。域*ac.uk*的命名服务器（目前有5个）知道本国所有大学域的命名服务器，例如*qmw.ac.uk*或*ic.ac.uk*。在某些情况下，一个大学域将管理权委派给一个子域，如*dcs.qmw.ac.uk*。



注意：命名服务器名用斜体，而相应的域在括号中。箭头指示了命名服务器条目

图9-4 DNS命名服务器

如上所述，根域信息由主服务器复制到约一打从服务器上。尽管如此，据Liu与Albitz[1998]的调查，一些根服务器依然需每秒接收约1000个查询。所有DNS服务器都会存储一个或多个根命名服务器的地址，这些服务器的地址通常不发生改变。DNS服务器通常也会存储父域的一个权威服务器的地址。查询诸如*www.berkeley.edu*这样的具有3个成分的域名，最坏情况下需要两步导航：第一步是向存储了合适的命名服务器条目的根服务器发出请求，第二步则向第一次查询得到的服务器发出请求。

参见图9-4，域名*jeans-pc.dcs.qmw.ac.uk*可以使用本地服务器*dns0.dcs.qmw.ac.uk*从域*dcs.qmw.ac.uk*查询到。该服务器未存储Web服务器*www.ic.ac.uk*的任何条目，但它缓存了*ic.ac.uk*的条目（从权威服务器*ns0.ja.net*中获得）。服务器*dns0-doc.ic.ac.uk*服务器可用于解析全名。

导航与查询处理 DNS客户被称为解析器，通常它被实现为库代码。它接收查询后，将查询格式化为符合DNS协议的格式，再与一个或多个命名服务器通信。通信中一般使用的是简单的请求-应答协议，通常情况下使用因特网的UDP包（DNS服务器使用的是一个已知的端口号）。解析器在需要时会检测超时，并重发其查询。解析器可被配置成与一组带优先级的初

始命名服务器联系，以应对某个或某几个服务器不可用的情况。

DNS体系结构允许迭代导航，也允许递归导航。当与命名服务器联系时，解析器指定需要何种类型的导航。然而，命名服务器并不一定实现递归导航。正如上面所指出的，递归导航会占用服务器线程，这意味着其他请求会被延迟。

为解决网络通信问题，DNS协议允许多个查询被打包到同一个请求消息中，相应地，命名服务器可以在应答消息中发送多个回答。

资源记录 区域数据以多种固定类型的资源记录形式存储到命名服务器的文件中。对于因特网数据库，包含了图9-5所示的类型。每条记录指的是一个域名，在图中未表示出来。除了*TXT*条目，表中的条目大多在上文已提及。*TXT*条目主要是为了使域名的任意信息都可被存储。

记录类型	含　义	主要内容
A	计算机地址	IP号
NS	权威命名服务器	服务器的域名
CNAME	别名的标准名	别名的域名
SOA	标识了一个区域数据的开始	管理该区域的参数
WKS	已知服务的描述	服务名与协议的列表
PTR	域名指针（反向解析）	域名
HINFO	主机信息	机器体系结构与操作系统
MX	邮件交换	<优先级，主机>列表
TXT	正文串	任意文本

图9-5 DNS 资源记录

一个区域的数据从一个*SOA*类型的记录开始，该记录包含了区域参数，如版本号以及从服务器刷新副本的频率。*SOA*类型的记录后紧跟着类型为*NS*的记录集合，用于指定域的命名服务器，然后跟着类型为*MX*的记录集合，用于给出邮件主机的优先级和域名。例如，域*dcs.qmw.ac.uk*的数据库有下列记录，记录中的*1D*表示存活期为1天：

域名	存活期	类别	类型	值
	1D	IN	NS	dns0
	1D	IN	NS	dns1
	1D	IN	NS	cancer.ucs.ed.ac.uk
	1D	IN	MX	1 mail1.qmw.ac.uk
	1D	IN	MX	2 mail2.qmw.ac.uk

数据库中后面的类型为A的记录会给出两个命名服务器——*dns0*与*dns1*的IP地址。邮件主机以及第三个命名服务器的IP地址在相应域的数据库中给出。

对于诸如*dcs.qmw.ac.uk*这样较低层的区域，数据库剩下的主要记录会是*A*类型，它将计算机的域名映射到其IP地址。对于一些著名的服务，数据库可能会包含一些别名，例如：

域名	存活期	类别	类型	值
www	1D	IN	CNAME	copper
copper	1D	IN	A	138.37.88.248

如果该域还有子域，那么将会有更多的*NS*类型记录，指定了子域的命名服务器，这些服

务器也会有自己的A类型的条目。例如，数据库*qmw.ac.uk*对于子域*dcs.qmw.ac.uk*的命名服务器会有下列记录：

域名	存活期	类别	类型	值
dcs	1D	IN	NS	dns0.dcs
dns0.dcs	1D	IN	A	138.37.88.249
dcs	1D	IN	NS	dns1.dcs
dns1.dcs	1D	IN	A	138.37.94.248
dcs	1D	IN	NS	cancer.ucs.ed.ac.uk

367 ~ 369

命名服务器的负载共享 对于某些站点，诸如Web、FTP等重负荷的服务由同一网络上的一组计算机同时支持。在这种情况下，该组的每个成员使用的是同一个域名。当一个域名由多台计算机共享时，命名服务器对该组的每台计算机都有一条记录，并给出其IP地址。对于名字会涉及到多条记录的查询，命名服务器根据循环调度方法返回结果。这样，后续的客户访问被分发到不同的服务器，以便服务器之间能均衡负载。而缓存可能会破坏这种机制，因为一旦一个非权威的命名服务器或客户在它的缓存中包含了某个服务器的IP地址，那么它会持续地使用该地址。为消除该后果，资源记录一般给定较短的存活期。

DNS的BIND实现 伯克利因特网域名系统（Berkeley Internet Name Domain，BIND）是UNIX上的DNS实现。客户程序作为解析器连接BIND的软件库。DNS命名服务器所在的计算机运行BIND的守护进程。

BIND允许3类命名服务器：主服务器、从服务器以及仅提供缓存功能的服务器。指定的程序根据配置文件内容，仅实现3类中的一类服务。前两类如上文所述。缓存服务器从一个配置文件中读取足够多的权威服务器的名字与地址用于解析。因此，缓存服务器仅存储这些数据以及在为客户解析名字中所学到的数据。

一个组织通常具有一个主服务器以及在站点的不同局域网段提供命名服务的一个或多个从服务器。另外，各个计算机常常运行自己的缓存服务器，以进一步降低网络开销，减少响应时间。

关于DNS的讨论 考虑到因特网命名数据的海量数量以及网络的全球性规模，可以说DNS的因特网实现获得了较短的平均查询响应时间。我们看到，这一效果是通过对命名数据的分区、复制以及缓存而获得的。命名的对象主要是计算机、命名服务器以及邮件主机。计算机（主机）名到IP地址的映射以及命名服务器与邮件主机的标识等信息改变不太频繁，因此，缓存与复制在一个相对比较宽松的环境中发生。

DNS允许命名数据不完全一致，即当命名数据修改时，其他服务器在几天时间内仍会向客户提供过期的信息，在第14章中讨论的复制技术未在此处被使用。然而，只有在客户试图使用过期信息的情况下，不一致性才会产生影响。DNS自己未解决如何探测过期数据的问题。

除计算机外，DNS也命名了一种特殊类型的服务：基于每个域的邮件服务。DNS假设在每个指定的域中，仅有一个邮件服务器，因此，用户无须显式地用名字指出服务的类型。电子邮件应用在与DNS服务器联系时使用合适的查询类型，透明地选择该服务。

总之，DNS存储的不同类型的名字数据是非常有限的，但这已足够使诸如电子邮件这样的应用将它们自己的名字机制加到域名之上。DNS数据库作为对大量因特网用户有用的、最低层的公共命名者的地位可能是值得质疑的。DNS并不是为因特网设计的惟一命名服务，它

与本地名字与目录服务共存，后者存储了与本地需求有关的数据（如Sun的网络信息服务，该服务存储了加密的口令，或者微软的活动目录服务[www.microsoft.com]，该服务存储了有关一个域的所有资源的详细信息）。 370

DNS设计的一个潜在的问题是，它的设计过于严格，很难改变它的名字空间的结构，同时，缺乏定制名字空间以满足本地需求的能力。在9.4节的全局命名服务实例研究中，考虑了命名设计的这些方面。在此之前，我们考察目录服务与发现服务。

9.3 目录服务和发现服务

我们已描述了命名服务存储<名字，属性>集合的方法以及如何通过名字查询属性。很自然，我们会考虑上述情况的相对面，即将属性作为查询的值。在这些服务中，文本名仅仅被看作是另一个属性。有时，用户希望找到一个特殊的个人或资源，他不知道对方的名字，仅知道对方的一些属性。例如，一个用户可能会问："电话号码为020-555 9980的用户名是什么？"有时，用户需要一个服务，但只要服务可以被方便地被访问即可，并不关注系统中的哪个实体提供了该服务。例如，用户会问："本大厦的哪台计算机是运行了MacOS 8.6 操作系统的Macintosh机？"或者，"我在哪儿可以打印一个高分辨率的彩色图像？"

具有下列功能的服务称为*目录服务*：存储了一组名字和属性的绑定，条目的查询基于属性规范。目录服务的例子有：微软的活动目录服务、X.500以及LDAP（在9.5节描述）、Univers[Bowman *et al.* 1990]和Profile[Peterson 1988]。目录服务有时也称为*黄页服务*，而传统的命名服务被称为*白页服务*，与不同类型的电话簿目录的功能相似。目录服务有时也被称为*基于属性的命名服务*。

目录服务返回满足特定属性的所有对象的属性。例如，'TelephoneNumber = 020-555 9980'这样的请求可能会返回{'Name = John Smith'，'TelephoneNumber = 020-555 9980'，'emailAddress = john@dcs.gormenghast.ac.uk'，…}。客户可能会指定感兴趣的属性子集——例如，仅返回匹配对象的邮件地址。X.500以及其他目录服务也允许通过传统的层次型文本名查找对象。

属性用于指定对象显然比名字更强，在不知道名字的情况下，可以通过编写程序，根据精确的属性规范选择对象。属性的另一个优点是不会将组织机构内部的结构暴露给外界，而根据组织机构划分的名字会发生这种情况。然而，使用文本名相对简单，这使得在很多应用中，命名服务不可能被基于属性的命名方法替代。

发现服务 发现服务是一种目录服务，它注册了自发网络环境下提供的服务。正如在2.2.3节所解释的，在自发网络中，设备倾向于在不发出警告、未做管理预备工作的情况下接入网络。自发网络的目标是客户与服务的集合可以动态地改变，并且无需用户干预即可集成。为满足这些特定的需求，发现服务提供了可自动注册与注销服务的接口以及客户接口，客户接口用于从目前已有的服务中找到他们需要的服务。 371

例如，考虑一个招待所或是宾馆（见1.2.3节与2.2.3节）的某个偶然的房客，他需要打印一个文档。用户不太可能在便携机上配置本地特定打印机的名字，也不可能猜出它们的名字（'\\myrtle\titus'以及'\\lionel\frederick'）。比起强迫用户在旅行时重新配置他们的机器，更好的方法是便携机使用发现服务的查询接口，寻找满足用户需求的、可用的网络打印机。可以通过与用户交互或是参考一个关于用户偏好的记录来选择特定的打印机。Macintosh操作系

统的用户会对该功能非常熟悉。

打印服务需要很多属性，例如，指定是“激光打印机”还是“喷墨打印机”，是否提供彩色打印，相对于用户的物理位置（例如，打印机的房间号）等。

类似地，服务需要通过它的注册接口，通知发现服务它的存在。例如，一个打印机（或者是管理它的一个服务）会用如下字符串向发现服务注册其属性：

'resourceClass=priner, type=laser, colour=yes, resolution=600dpi,

location=room101, url = http://www.hotelDuLac.com/services/printer57'

上面的URL指定了打印机的网络位置。

注意，使用发现服务进行查询服务时不一定牵涉到用户。例如，家用冰箱在不能通过自检时，为了通过家用PC通知主人，可能会去发现一个错误日志服务。

在发现服务中，发现的上下文有时称为范围。有些服务，例如简单服务发现协议，适用于由本地网络可达性确定的范围中，例如家里由无线网络连接的所有资源构成的范围。这在很多情况下是合适的，例如，当一个用户请求一台打印机时，他们可能需要一台物理上邻近的机器，这常意味着在局域网上可达。与此相反，诸如X.500的目录服务，为了反映地理以及组织的范围空间，在结构上是层次型的。例如，X.500可能用于在本地组织机构或是在一个国家内部查询一个人。

发现服务的最新发展包括Jini发现服务（见下文）、服务位置协议[Guttman 1999]、国际命名系统[Adjie-Winoto *et al.* 1999]、简单服务发现协议（该协议是全球即插即用创新技术的核心[www.upnp.com]）以及安全服务发现协议[Czerwinski *et al.* 1999]。

Jini Jini系统 [Waldo 1999, Arnold *et al.* 1999]设计用于自发网络。它完全基于Java——它假设JVM在所有计算机上运行，允许它们通过RMI协议（见第5章）的方式与其他人通信，并且在必要的时候下载代码。Jini提供了有关服务发现、事务处理、称为JavaSpaces的共享数据空间（与Tuple空间类似）以及事件服务的功能。这里我们仅描述发现系统。

372

在Jini系统中，与发现服务相关的组件是查询服务、Jini服务以及Jini客户（见图9-6）。查询服务实现了发现服务，尽管Jini仅在发现查询服务本身时才使用“发现”这个术语。查询服务允许Jini服务注册所提供的服务，允许Jini客户向查询服务请求与具体需求相匹配的服务。一个Jini服务（例如打印服务）可以向一个或多个查询服务注册。Jini服务提供的同时也是查询服务存储的是实现了该服务的对象以及服务的属性。Jini客户询问查询服务，以获得满足需求的Jini服务，如果找到一个匹配，则它们从查询服务下载提供了该服务的对象。对于客户请求的匹配可以基于属性或Java类型，例如，允许客户请求一台具有相应Java接口的彩色打印机。

与其他任何支持自发网络的系统相似，Jini面临自举时的连接问题。当一个用户或服务进入网络，它必须使用查询服务，但如何定位查询服务本身呢？在有些情况下，客户与服务不经观测即可得到特定查询服务的地址——它由用户键入。但更有趣的情形是，客户与服务是新的，它们没有按任何方式配置，因此必须定位查询服务。Jini使用的方法与其他发现服务解决该问题的方法相同：通过向一个已知的IP组播地址使用组播，Jini软件的所有实例均知道该地址。

当Jini客户或服务启动时，它们对该组播地址发送一个请求。在组播请求中，使用了“存活期”值，以限制携带数据报的请求到达其邻近网络。查询服务监听绑定到同一地址上的套接字，以接收上面的请求。任何收到请求并可响应的查询服务（见下文）使用服务请求的单

点发送地址回答，请求者则执行一个远程调用，在该发现服务中查询或注册一个服务（在Jini中，注册被称为加入）。

查询服务也使用相同的组播地址发送数据包以表明本服务可用。Jini客户与服务也同时监听该组播地址，以便获知新的查询服务。

一个给定的Jini客户或服务通过多点通信可获得多个查询服务的实例。每个这样的服务实例通过一个或更多的组名配置，例如："admin"、"finance"以及"sales"，此时组名作为范围标签。当客户或服务请求一个查询服务时，它们会指定感兴趣的组，只有属于相应组的查询服务会响应。例如，一个公司被分成不同的部门，请求"admin"打印机的设备会找到属于"admin"组的打印机，而请求"finance"打印机的设备会找到与"finance"组绑定的打印机。

图9-6显示了一个发现并使用打印服务的Jini客户。客户需要一个属于"finance"组的查询服务，因此，它组播出一个具有该组名的请求。仅有一个查询服务绑定到"finance"组（该服务也绑定到"admin"组），该服务做出应答。查询服务的应答包括了客户的地址，客户用该地址通过RMI直接定位所有类型为"printing"的服务。仅有一个打印服务在查询服务中注册在"finance"组下，因此，访问该服务的一个对象被返回。然后客户利用返回的对象，直接使用打印服务。图9-6也显示了另一个打印服务，即在"admin"组中的那个。该图还有一个不绑定到任何组的公司信息服务（该服务向所有的查询服务注册）。

Jini利用了第5章讨论的租借。当Jini服务向查询服务注册时，这些服务获得一个租借，它保证了注册条目会存活一个有限的时间周期。如果服务未能在租借过期之前与查询服务联系以更新租借，则服务被认为失效，而查询服务则可以删除相应条目。

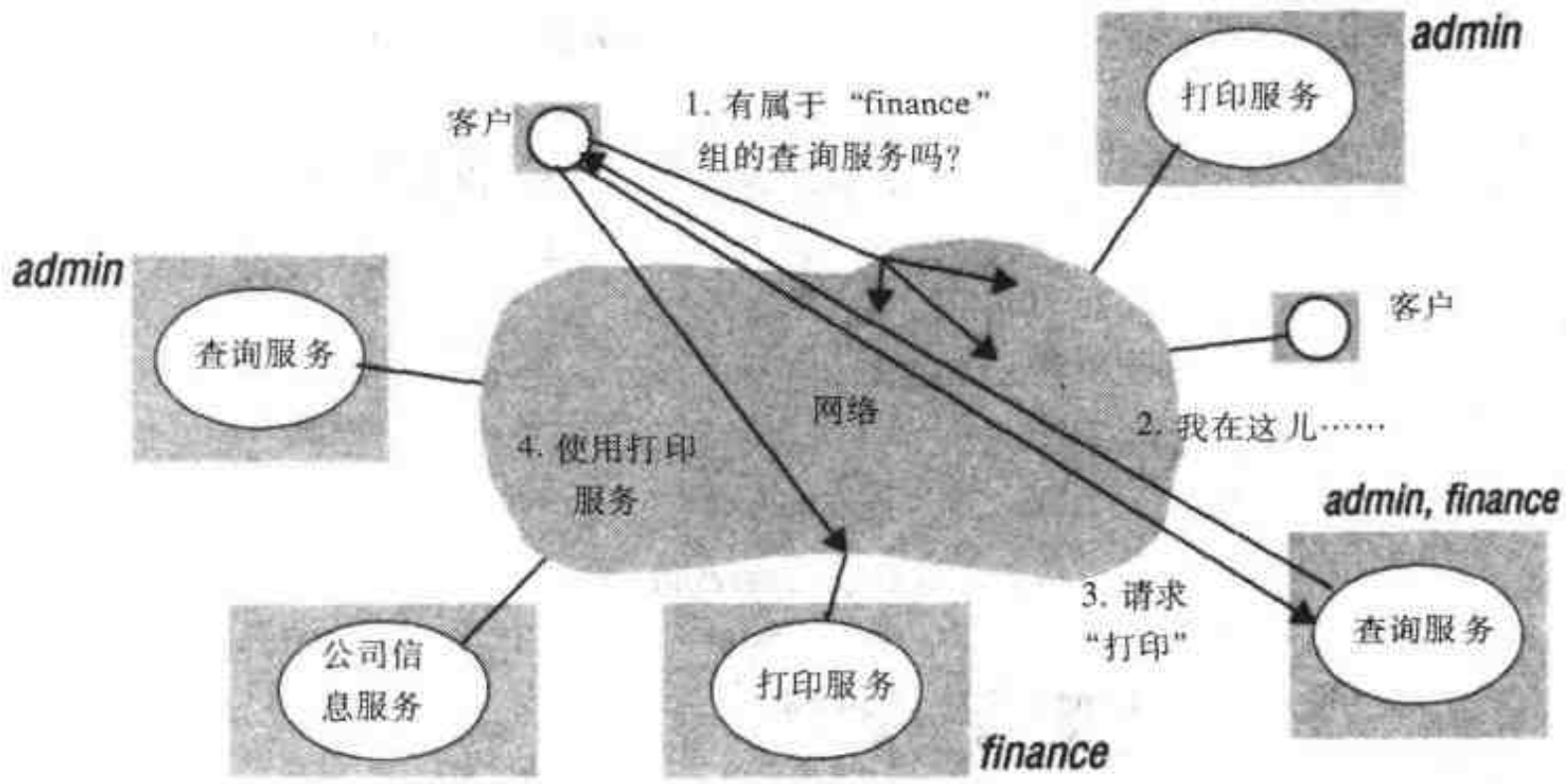


图9-6 Jini中的服务发现

9.4 实例研究：全局命名服务

全局命名服务（GNS）是由Lampson与DEC系统研究中心[Lampson 1986]的同事设计与实现的，用于提供资源定位、邮件寻址以及认证等功能。GNS的设计目标已在9.1节的最后列出，这些目标反映的事实是：互联网使用的命名服务必须支持一个命名数据库，该数据库可以扩展到包含数百万台计算机的名字以及几十亿的用户邮件地址。GNS的设计者也意识到名字数

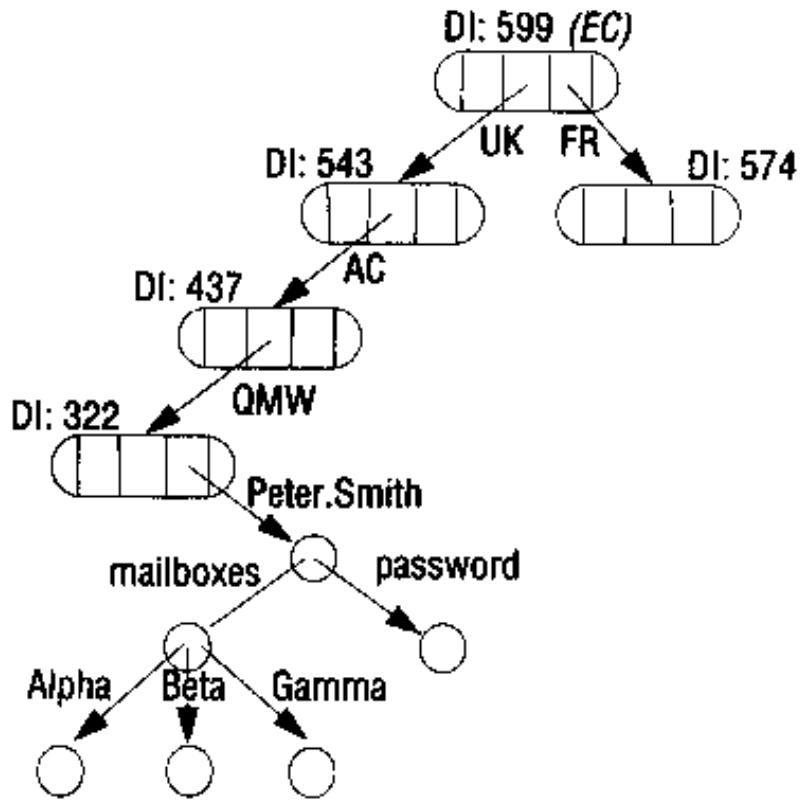
据库可能会有很长的生存期，它必须能在规模由小变大以及底层网络发展等情况下有效地工作。在此过程中，名字空间的结构可能会改动以反映组织机构的变化。命名服务必须允许个人、组织、小组的名字的变化，除此以外，也允许一个公司被另一个公司接管时名字结构发生变化。此处我们集中描述提供这些变化的设计要点。

373
~
374

由于GNS可能在大规模分布式环境中运行，具有海量命名数据库，从而缓存的使用成为设计要点，而有了缓存，维护数据库条目的所有副本的完全一致性就变得困难。所采取的缓存一致性策略基于下面的假设：数据库不会被频繁修改，而因为客户可以探测和修复对旧命名数据的使用，所以慢速发送数据修改是可以接受的。

GNS管理的名字数据库由一个包括了名字与值的目录树构成。目录命名方式可以是相对于根或相对于某个工作目录的多部分路径名，与UNIX文件系统很相似。每个目录被分配一个整数作为惟一的目录标识（DI）。本节中，我们使用斜体字表示目录的DI，如*EC*是EC目录的标识。目录包含了一组名字与引用。目录树的叶子存储的值被组织成值树，这样与名字相关的属性可以是结构化的值。

GNS中的名字有两个部分：<目录名，值名>。第一部分标识了一个目录，而第二部分指的是值树或是值树的一部分，如图9-7所示。在图中为说明方便，DI都是小整数，尽管在实际中为保证惟一性，DI会从很广的整数范围中选择。目录QMM下的用户Peter.Smith的属性会存储在一个名为<*EC*/UK/AC/QMW, Peter.Smith>的值树中。该值树包含了一个口令和多个邮件地址，口令可以通过<*EC*/UK/AC/QMW, Peter.Smith/password>方式来引用，而每个邮件地址都作为值树的单独结点，以<*EC*/UK/AC/QMW, Peter.Smith/mailboxes>作为结点名列出。



375

图9-7 GNS中用户Peter.Smith的目录树和值树

目录树分区后被存储在多个服务器中，每个分区又由多台服务器复制。首先要维护在两个或多个并行修改的情况下树的一致性——例如，两个用户试图同时用同一名字构造一个条目，应该仅有一人能成功。复制目录带来了另一个一致性问题，可以通过一个能够保证最终一致性的、基于异步修改的分布式算法解决该一致性问题，但不能保证所有的副本都是最新的。在这一问题上达到这个级别的一致性是可以被接受的。

适应改变 现在我们转到与命名数据库的增长与改变有关的设计方面。在客户与管理员层，通过正常的目录树的扩展适应增长的需要。但我们可能会集成两个原来分离的GNS服务命名树。例如，我们如何将图9-7中的*EC*目录下的数据库与*NORTH AMERICA*数据库进行集成？图9-8显示了在需要合并的树的根之上，引入了新根*WORLD*。这个技术非常直接，但对继续使用合并前的“根”作为名字的客户有什么影响呢？例如，</UK/AC/QMW, Peter.Smith>是在合并之前客户使用的名字。它是一个绝对名（因为它以“/”作为起始），但根指的是*EC*，而不是*WORLD*。*EC*与*NORTH AMERICA*是工作根——工作根是一个初始环境，这时必须查询以根“/”开始的名字。

惟一目录标识的存在可以解决这个问题，每个程序的工作根必须作为执行环境的一部分被识别（与一个程序的工作目录相同）。当一个在欧盟的客户使用形式为</UK/AC/QMW, Peter.Smith>的名字时，由于它的本地代理知道它的工作路径，因此会在名字前加了目录标识EC(#599)前缀，这样构造出名字<#599/UK/AC/QMW, Peter.Smith>。用户代理在一个对GNS服务器的查询请求中发送出该派生名。用户代理对指向工作目录的相对名使用相似的方法。了解新的配置的客户也会向GNS服务器提供绝对名，它指向包含了所有目录标识的概念上的超根目录，例如，<*WORLD*/EC/UK/AC/QMW, Peter.Smith>，但设计无法假设考虑到该变化所有的客户会被更新。

上述技术解决了逻辑上的问题，它甚至在新插入一个真实的根的情况下，依然允许用户以及客户程序继续使用已定义的、相对于旧根的名字。但这样做遗留了一个实现问题：在包含了上千万目录的分布式命名数据库中，仅给定诸如#599这样的目录标识，GNS服务如何定位一个目录？GNS采纳的解决方案是在命名数据库当前真实的根下包含一个表，称为“已知目录”表，该表列出了所有作为工作根使用的目录，如EC。一旦命名数据库真实的根发生改变，如图9-8所示，所有GNS服务器被通知真根的新位置。然后它们可使用常用方式解释WORLD/EC/UK/AC/QMW（相对于真实根）形式的名字，也可使用“已知目录”将“#599/UK/AC/QMW”格式的名字翻译成以真实根开始的完全路径名。

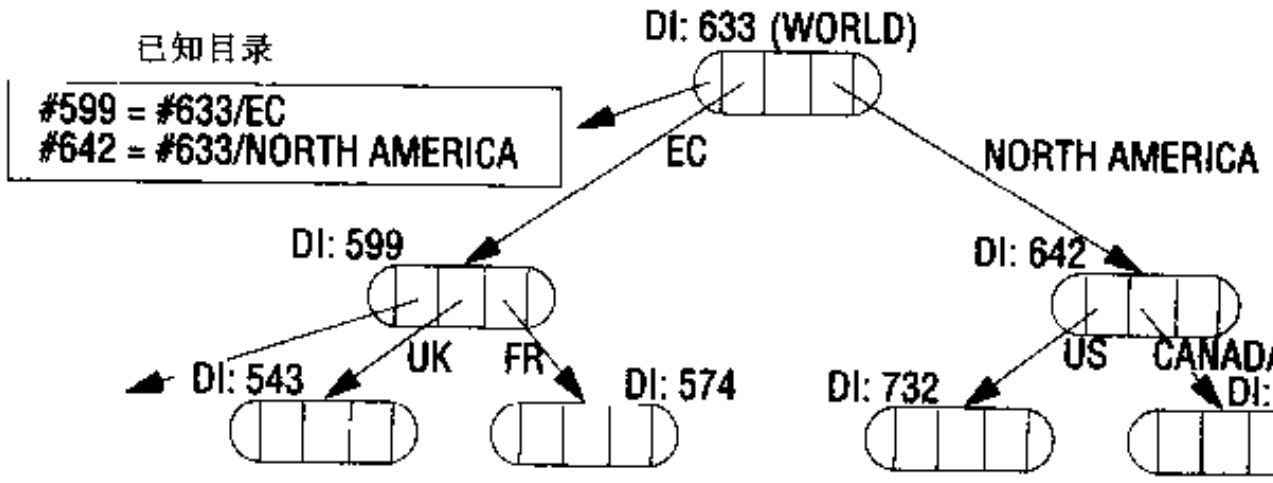
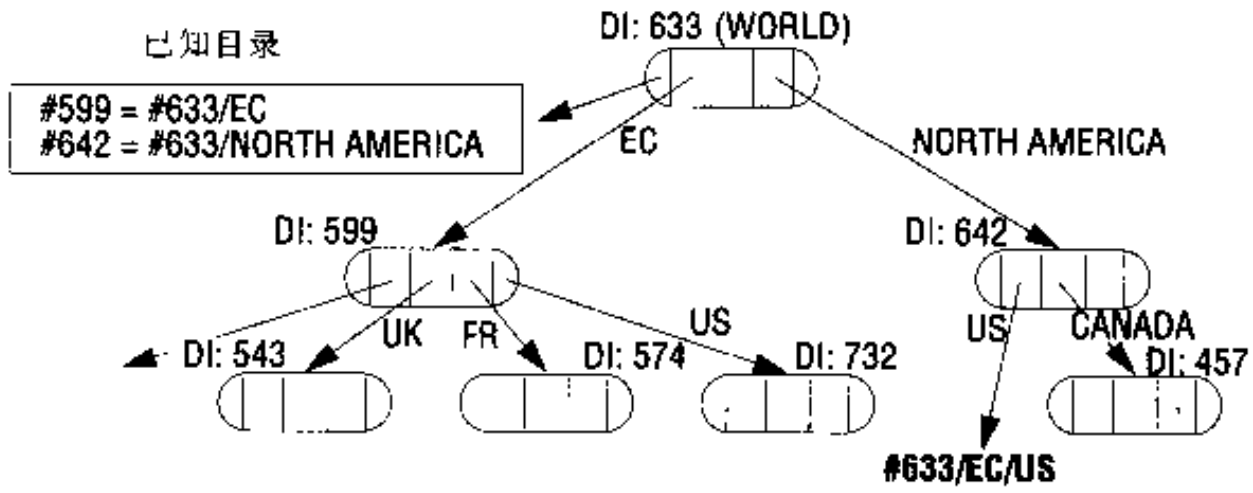


图9-8 在一个新根下合并树

GNS也支持数据库的重构，以适应组织变化。试想美国成为欧盟的一部分（！）。图9-9给出了新的目录树。但如果US子树仅被简单地移到EC目录下，*WORLD*/NORTH AMERICA/US将无法继续工作。GNS采取的方法是增加一个“符号连接”替代原有的US条目（见图9-9中加黑的一部分）。GNS目录查找过程将连接重定向到新位置上的US目录。

GNS的讨论 GNS由Grapevine[Birrell *et al.* 1982]以及Clearinghouse[Oppen and Dala1

1983]发展而来，这两个系统是施乐公司成功开发的，主要面向邮件发送系统的命名服务。GNS成功地解决了可伸缩性与可配置性问题，但合并与移动目录树采用的方法导致一个数据库（“已知目录”表）在每个结点被复制。在大规模的网络中，重配置可以在每个层次上发生，而该表可能会增长到很大．这与可伸缩性的目标相冲突。



376
~
377

图9-9 重构目录

9.5 实例研究：X.500目录服务

X.500是9.3节定义的目录服务。它可以按传统的命名服务的使用方式使用，但它通常被用于满足描述性的查询，被设计用来发现其他用户或系统资源的名字与属性。在网络用户、组织机构以及系统资源目录下，用户可能会有各种搜索与浏览需求，以获取该目录包含的实体的信息。服务的使用有可能非常分散。查询的范围很广，可以使用电话簿相似的方法，例如简单的“白页”访问，以获得一个用户的电子邮件地址；或是一个“黄页”查询，例如获得一个专修某种类型汽车的修车厂的名字与电话号码，再如，使用目录访问个人的工作角色、饮食习惯甚至照片等信息。

这些查询可以由用户发出，如上文中修车厂例子所代表的“黄页”查询；或是从进程发出，例如，在用于识别满足某个功能的服务时。

在网络中，个人与组织可以使用目录服务使大量有关自己的信息以及提供的资源被他人访问。用户可在仅有部分名字、结构或内容的信息的情况下，搜索目录寻找特定信息。

ITU与ISO标准组织已经将X.500目录服务[ITU/ISO 1997]定义为一个满足上述需求的网络服务。该标准称X.500为一个访问有关“现实世界实体”信息的服务，但它也可用于访问有关软硬件服务与设备的信息。X.500被定义为开放系统互连（OSI）标准中的一个应用级的服务，但其设计在很大程度上并不依赖于其他OSI标准，它可以被看做是一个通用的目录服务。我们将在这里概述X.500目录服务的设计与实现。对X.500更详细的描述以及对实现方法感兴趣的读者可以参看Rose[Rose 1992]有关该主题的书。X.500也是LDAP的基础（将在下面讨论），用于DCE的目录服务[OSF 1997]。

在X.500服务器中存储的数据被组织成一个由名字结点构成的树状结构，正如在本章中提到的其他命名服务一样，但在X.500中，树的每个结点存储了大量的属性，访问不仅可以根据名称，也可以根据属性的组合进行搜索。

X.500名字树也被称为目录信息树（DIT），而整个目录结构以及存储在内的数据被称为目录信息库（DIB）。一般倾向于全世界范围内的机构提供的信息被存储在单个集成的DIB中，

而DIB的一部分存储在单独的X.500服务器中。一个中等规模或大规模的组织通常会提供至少一个服务器。客户通过向服务器建立一个连接以及发出访问请求来访问目录。客户可以通过一个查询与服务器联系。如果请求的数据并不在连接的服务器的DIB中，该服务器或者会调用 378
其他服务器以解析查询，或者将客户重定向到另一个服务器。

在X.500标准术语中，服务器被称为目录服务代理（DSA），客户被称为目录用户代理（DUA），图9-10给出了软件体系结构以及可能的几种导航模型中的一种，图上，每个DUA客户与单个DSA服务器交互，而DSA服务器在需要的时候访问其他DSA。

DIB的每个条目由一个名字和一组属性集组成。与其他命名服务器相似，一个条目完整的名字对应于DIT的一个从树根到条目的路径。除完整名或绝对名外，DUA可以建立一个上下文环境，它包括一个基础结点，然后DUA即可使用较短的相对名，该名字给出了从基础结点到被命名条目的路径。

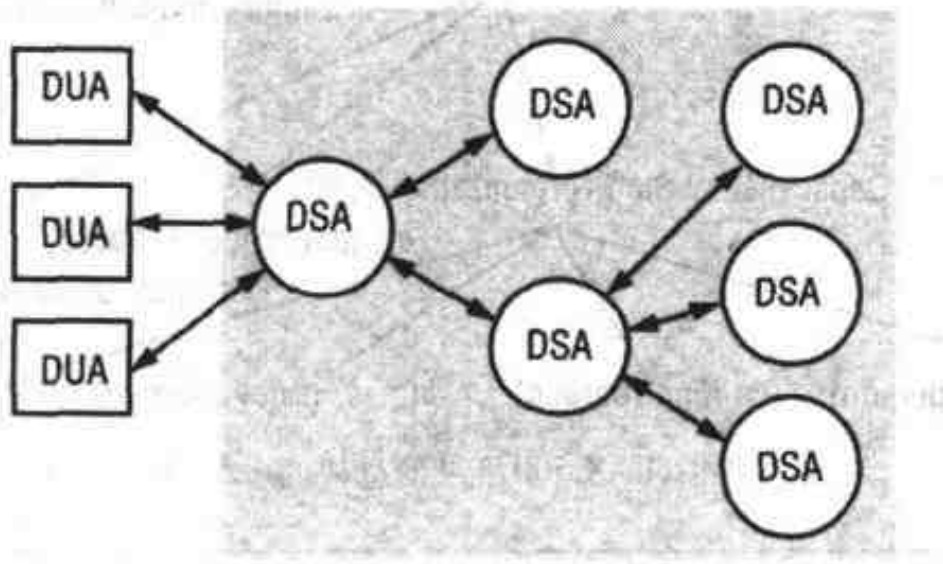


图9-10 X.500服务的体系结构

图9-11显示了包括了英国Gormenghast大学的那部分目录信息树，图9-12是一个相关的DIB条目。DIB与DIT中的条目非常灵活。一个DIB条目包括一组属性，而每个属性由一个类型和一个或多个值组成。属性的类型由类型的名字表示（如*countryName*、*organizationName*、*commonName*、*telephoneNumber*、*mailbox*、*objectClass*）。新的属性类型在需要的时候可以被定义，该定义包括一个类型描述以及一个使用ASN.1符号表示法（一个语法定义的标准符号表示法）的语法定义，语法定义确定了该类型的值域。

DIB条目的分类方式与面向对象语言中的对象类结构相似。每个条目包括了一个*objectClass*属性，它定义了一个条目指向的对象的类，*organization*、*organizational Person*以及*document*都是*objectClass*值的例子。在需要的时候可以进一步地定义类。类定义确定了给定类的条目中哪些属性是必需的，哪些是可选的。类的定义组织成一个继承层次，其中，除了类*topClass*，所有类都必须有*objectClass*属性，*objectClass*属性的值必须是一个或多个类的名字。如果有多个*objectClass*值，对象继承每个类的必需的和可选的属性。 379

确定DIB条目的名字（确定了它在DIT中位置的名字）是通过选择一个或多个属性作为辨别属性。基于该目的而被选中的属性被称为条目的辨别名（DN）。

现在我们可以考虑访问目录的方法。有两种类型的访问请求：

- 读　给定某个条目的绝对的或相对名字（按X.500术语称为域名）以及需要被读的属性列表（或者需要所有属性的指示）即可。DSA通过在DIT中导航，定位被命名的条目，当它没有相关条目时，会向其他DSA服务器发出请求。DSA检索需要的属性，并将它们返回给客户。

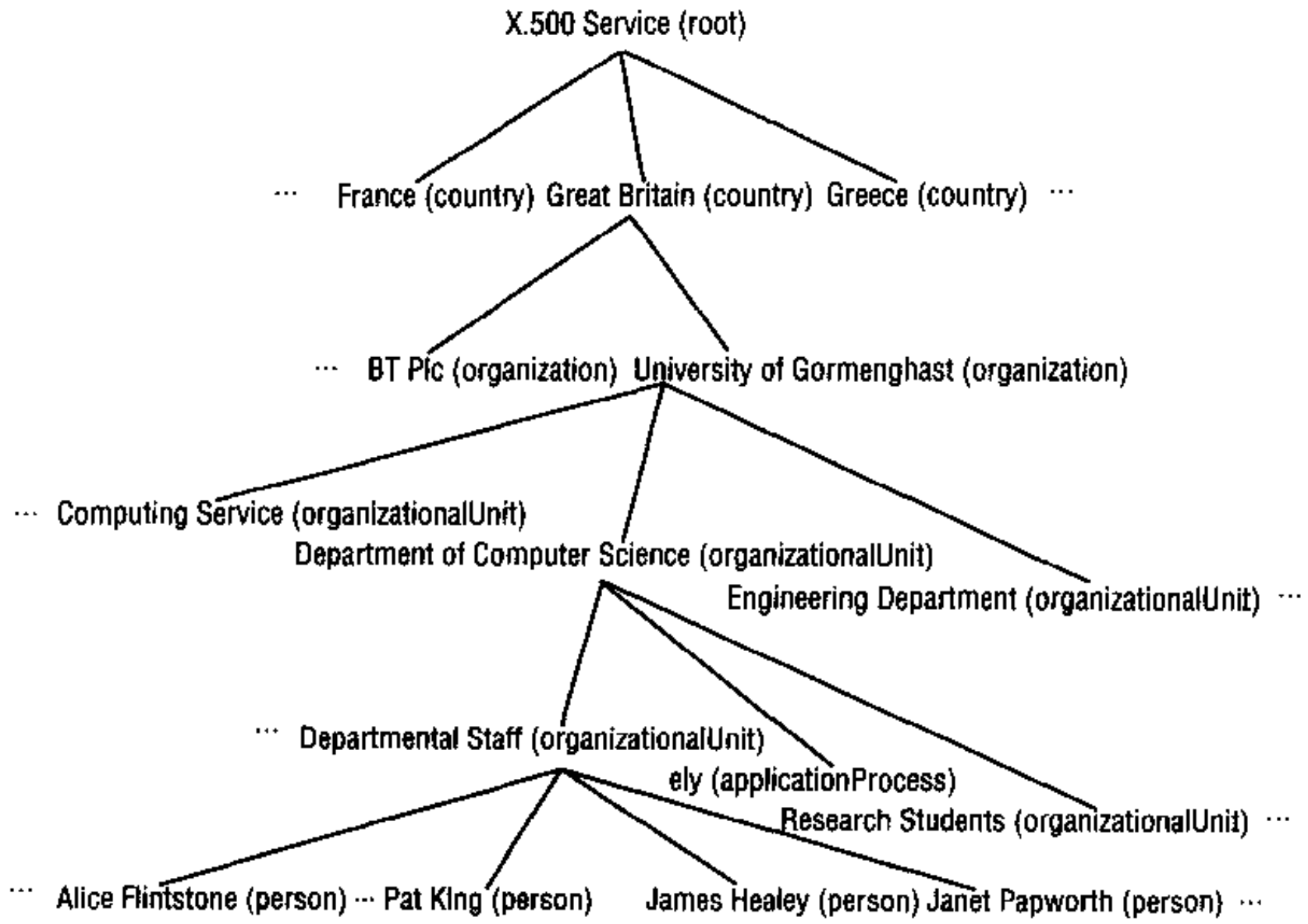


图9-11 X.500目录信息树的一部分

info

Alice Flintstone, Departmental Staff, Department of Computer Science,
University of Gormenghast, GB

commonName
Alice.L.Flintstone
Alice.Flintstone
Alice Flintstone
A. Flintstone

surname
Flintstone

telephoneNumber
+44 986 33 4604

uid
alf

mail
alf@dcs.gormenghast.ac.uk
Alice.Flintstone@dcs.gormenghast.ac.uk

roomNumber
Z42

userClass
Research Fellow

图9-12 一个X.500 DIB条目

- *搜索* 这是一个基于属性的访问请求。需提供一个基本名以及一个过滤器表达式作为参数。基本名指定了在DIT中开始搜索的结点，过滤表达式是一个布尔表达式，用于对基本结点以下的每个结点进行求值。过滤表达式指定了一个搜索策略：对条目属性值进行各种逻辑组合测试。*search*命令会返回一组名字（域名），这些名字是基本结点之下的条目名并且这些条目的过滤表达式计算值为真。

例如，可以构造一个过滤表达式，用于寻找在Gormenghast大学计算机科学系占据了

房间Z42的员工的*commonName*（如图9-12所示）。然后使用一个读请求获得这些DIB条目的任意属性。

搜索目录树的大子树（可能会存储在多台服务器中）需要很大开销。可以提供更多的参数以限制搜索的范围，例如继续搜索的时间以及返回条目的数量。

DIB的管理与更新 DSA接口包括了增加、删除以及修改条目的操作，查询与更新操作都提供了访问控制，因此对部分DIT的访问必须限定到特定用户或是一组用户。

一般，DIB是被分区的，期望每个组织将提供至少一个服务器用于容纳该组织中实体的细节。DIB的各个部分可以被复制到多个服务器上。

作为一个标准（或按CCITT术语称为“建议”），X.500未谈及实现。然而，很清楚的是，在广域互联网中，任何包含了多个服务器的实现必须广泛地使用复制与缓存技术，以避免过多地查询重定向。

Rose[1992]描述了X.500的一个实现——QUIPU[Kille 1991]，它是伦敦大学专科学院开发的。在该实现中，缓存与复制的级别是单个DIB条目，或是在同一结点下的条目集。假设系统在修改后值变得不一致，而一致性被恢复的时间间隔可能会有几分钟。对于目录服务应用，这种更新分发的形式被认为是可接收的。 380 ~ 381

轻量级目录访问协议 X.500的标准接口使用了涉及ISO协议栈较上层的协议。密执安大学的一个研究小组提出了一个更轻量级的方法，称为*轻量级目录访问协议*（LDAP），在该协议中，DUA直接通过TCP/IP访问X.500目录服务，参见RFC2251[Wahl *et al.* 1997]。LDAP还用其他方法简化了X.500接口。它提供了一个相对简单的API，并使用文本编码替代了ASN.1编码。

尽管LDAP规范基于X.500，但LDAP并不需要X.500规范。任何实现都可以使用符合了更简单的LDAP规范的目录服务器——与X.500规范相反。例如，微软的活动目录服务提供了一个LDAP的接口。LDAP已被广泛采用，特别是在企业内部网中。它通过认证提供了安全的目录访问。

X.500的讨论 X.500为不同范围（单个组织或全球）的目录服务指定了一个详细的模型。它的主要影响在企业内部网级别上。在企业内部网中，它的影响通过LDAP软件间接传播出去。X.500将来是否能作为全球（因特网）目录标准还不明朗。首先，是否需要一个全球目录系统（基于X.500或其他方法）并不清楚——比如目录服务会导致个人隐私的泄露。其次，这样一个全球服务系统应与现有因特网命名标准集成，包括DNS名与邮件地址。最后，目录中提供的信息范围需要在国家或国际范围内协调，以保证DIB中存储的对象类的一致性。

9.6 小结

本章描述了分布式系统中命名服务的设计与实现。命名服务存储了分布式系统中的对象的属性——特别是它们的地址——并在用一个文本名查询时，返回这些属性。

命名服务的主要需求是处理任意数量名字的能力，服务应具有长期性、高可用性、故障隔离性与不信任容忍性。

主要设计问题有：首先，名字空间的结构——名字管理的语法规则。相关问题是解析模型，即多成分的名字被解析为一组属性的规则。另外，绑定名的集合必须被管理。大多数设计将名字空间分割为域——名字空间的离散区域，每个域具有一个独立的权威机构，该机构控制

域内名字的绑定。

382 命名服务的实现可能会跨越不同的组织机构与用户群。换句话说，名字与属性绑定构成的集合被存储在多个命名服务器上，每个至少存储一个命名域的部分名字集。从而出现了导航问题，即当需要的信息被存储在多个站点上时名字的解析方式。支持的导航类型有迭代、组播、递归式服务器控制的导航以及非递归式服务器控制的导航。

另一个有关命名服务器实现的重要方面是复制与缓存的使用。两者均对提高服务可用性以及减少名字解析时间有帮助。

本章考察了两个主要的命名服务的设计与实现。域名系统广泛地用于在因特网上命名计算机与寻址电子邮件，它通过复制与缓存取得了理想的响应时间。全局命名服务的设计解决了组织机构变化时名字空间的重新配置问题。

本章也考察了目录服务与发现服务。当客户提供基于属性的描述时，它们返回与对象与服务相匹配的数据。X.500目录服务是一个由CCITT和ISO定义的标准，它可以被企业内部网范围与因特网范围的目录使用。我们也描述了Jini发现服务，它的设计面向自发网络。Jini的查询服务为服务提供了注册自己的接口，也为客户发现满足它们需求的服务提供了接口。

练习

9.1 描述在分布式文件服务如NFS中（见第8章）所使用的名字（包括标识）与属性。

9.2 讨论在命名服务中使用别名带来的问题，并且指出如何解决这些问题。

9.3 解释为什么在不同名字空间可以局部集成的命名服务中，例如由NFS提供的文件命名机制中，需要迭代导航。

9.4 描述组播的导航中出现的名字未绑定问题。通过安装一个服务器，解决查询过程中名字的未绑定问题，意味着什么？

9.5 缓存如何提高了命名服务的可用性？

9.6 讨论DNS的绝对名与相对名在语法上缺乏差别（如最后的“.”）的情况。

9.7 考察DNS域与服务器的本地配置。你可以寻找一个在UNIX系统上安装的程序，如*nslookup*，它可以执行单个命名服务器的查询。

9.8 为什么DNS根服务器包含了两层名字的实体，如*yahoo.com*与*purdue.edu*，而不是一
383 层名字如*edu*与*com*？

9.9 默认情况下，DNS命名服务器包含了哪些命名服务器的地址，为什么？

9.10 为什么DNS客户选择递归导航而不是迭代导航？递归导航选项与命名服务器的并发性如何相关？

9.11 何时一个DNS服务器给一个名字查找返回多个回答，为什么？

9.12 Jini查询服务对客户请求的匹配基于属性或Java类型。使用例子解释两种匹配的不同。允许两种匹配的好处在哪里？

9.13 解释Jini查询服务在服务可能崩溃或不可访问的情况下如何使用租借以确保查询服务器注册的服务最新？

9.14 Jini“发现”服务使客户与服务器可以定位到查询服务器，请描述在该服务中IP组播和组名的使用。

9.15 GNS未保证命名数据库的所有副本是最新的，GNS的客户如何会意识到自己接收了一个过期的条目？在何种情况下，这是有害的？

9.16 讨论用X.500目录服务替代DNS与因特网邮件传送程序的可能的好处与不足。勾画一个互联网中的邮件传送程序的设计框架，其中每个邮件用户与邮件主机都注册到一个X.500数据库。

9.17 哪些安全问题可能会与目录服务相关，例如，在一个大学里运行的X.500目录服务？

384

第10章　时间和全局状态

10.1　简介
10.2　时钟、事件和进程状态
10.3　同步物理时钟
10.4　逻辑时间和逻辑时钟
10.5　全局状态
10.6　分布式调试
10.7　小结

本章介绍分布式系统中与时间有关的若干问题。时间是一个重要的实际问题。例如，我们要求全世界的计算机为电子商务事务给出一致的时间戳。时间也是理解分布运行是如何展开的一个重要的理论概念。但在分布式系统中，时间是不确定的。每个计算机可以有它自己的物理时钟，但时钟通常会偏离，又不能完美地同步它们。本章分析了使物理时钟大致同步的算法，然后解释逻辑时钟，包括时钟向量。时钟向量是给事件排序的一种工具，它不需要精确地知道事件是何时发生的。

全局物理时间的缺乏使我们很难查明分布式程序在执行时的状态。我们经常需要知道当进程**B**处在某种状态时进程**A**处在什么状态，但我们不能依靠物理时钟来了解在同一个时刻什么才是真实的情况。本章的后半部分研究在缺乏全局时间的情况下确定分布式计算中全局状态的算法。

10.1　简介

本章介绍一些基本的概念和算法，它们与分布式系统运行时的监控有关，与发生在分布式系统运行中的事件定序有关。

有几个理由表明，在分布式系统中，时间是一个重要而有趣的问题。首先，时间是我们想要精确度量的数量。为了知道一台特定计算机上的一个特定事件在什么时间发生，与一个权威的外部时间源同步它的时钟是必要的。例如，一个“电子商务”事务涉及的事件是在商人的计算机和银行的计算机上发生的。为了便于审核，这些事件必须要精确标记时间戳。

其次，为了解决分布的几个问题，已经开发了若干依赖时钟同步的算法[Liskov 1993]。这些算法包括维护分布数据一致性的算法（12.6节讨论用时间戳来串行化事务）；检查发送给服务器的请求的真实性的算法（Kerberos认证协议的一个版本依赖松散同步的时钟，具体讨论见第7章）；消除重复更新的算法（参见Ladin *et al.* [1992]）。

爱因斯坦在他的相对论中论证了从观察中得出的结论：不管观察者的相对速度如何，光速对所有的观察者是一个常量。他从这个假设证明了，两个事件在一个参照系下是同时的，而对其他与这个参照系相对运动的参照系中的观察者而言，它们不一定是同时的。例如，在地球上的观察者和在宇宙飞船中飞向太空的观察者对事件之间的时间间隔会有不同的意见，当他们的相对速度增加时，他们的看法就差距更大。

此外，对于两个不同的观察者，两个事件的相对顺序甚至会是相反的。但如果一个事件能引起另一个事件发生，那么上述情况就不可能出现。在这种情况下，对所有的观察者而言，虽然观察到的在原因和结果之间的时间间隔不同，但物理结果必然跟随在物理原因之后。这样就证明了，物理事件的时序对观察者是相对的，牛顿的绝对物理时间概念是不足以信的。在要度量时间间隔时，在宇宙中没有一个特别的能引起我们兴趣的物理时钟。

在分布式系统中物理时间的概念也是不确定的。这不是由于相对性的影响，相对性在常规计算机中是可忽略或不存在的（除非在太空旅行中用计算机计数！）。问题是我们的能力有限，不能对不同结点上的事件记下足够精确的时间，以便知道事件发生的顺序或事件是否同时发生。没有绝对的全局时间。可是，我们有时需要观察分布式系统，确定事件的某些状态是否发生在同一时间。例如，在面向对象系统中，我们需要能够确定对某一特定对象的引用是否不再存在——对象是否已经变成了无用单元（这时我们能释放它的内存）。判断这些需要观察进程的状态（找出它们是否包含引用）和进程之间的信道（万一包含引用的消息正在传
386 送过程中）。

在本章的前半部分我们分析了用消息传递使计算机时钟能大致同步的方法，接着介绍逻辑时钟，包括时钟向量，时钟向量用于定义事件的顺序，它不需要度量事件发生时的物理时间。

本章的后半部分描述了一些算法，这些算法用于捕获分布式系统在运行中的全局状态。

10.2 时钟、事件和进程状态

第2章介绍了分布式系统中进程之间的交互模型。我们将细化该模型，以帮助大家理解如何随系统的执行刻画系统的演化，如何给系统执行中用户感兴趣的事件标记时间戳。我们从如何给发生在一个进程中的事件排序和标记时间戳开始。

设一个分布式系统由N个进程(p_i, i=1,2,…,N)组成，记为$\mathcal{P}$。每个进程在一个处理器上执行，处理器之间不共享内存（第16章考虑了共享内存的进程的情况）。在$\mathcal{P}$中，进程p_i的状态是s_i，通常，在进程执行时进行状态变换。进程的状态包括进程中所有变量的值，可能还包括在它本地操作系统环境中任一受它影响的对象（如文件）的值，此处假设除了通过网络发送消息外，进程之间不能相互通信。例如，如果进程操纵机器人手臂，这些手臂连接到系统中各自独立的结点，那么不允许机器人通过握手来通信。

当每个进程p_i执行时，它有一系列的动作，每个动作或是一个消息发送/接收操作，或是一个状态转换操作，即改变在s_i中的一个或多个值。实际上，我们可以根据应用选择使用动作的高层描述，例如，如果$\mathcal{P}$中的进程用于一个电子商务应用，那么动作可能是“客户发出订单消息”或“交易服务器在日志中记录事务情况”。

我们把事件定义成发生了一个动作（通信动作或状态转换动作），该动作由一个进程完成。在进程p_i中的事件序列可以有一个全序，我们用事件之间的关系$\rightarrow_i$表示，就是说，$e \rightarrow_i e'$当且仅当在p_i中事件e在e'前发生。这个排序是良定义的，不论进程是不是多线程的，因为，我们假设进程在单个处理器上执行。

现在我们把进程p_i的历史定义成在该进程中发生的事件的序列，而且按关系$\rightarrow_i$排序：

$$history(p_i) = h_i = \langle e^0_i, e^1_i, e^2_i, \ldots \rangle$$

时钟 我们已经知道如何在一个进程中给事件排序，但还不知道如何给事件标记时间戳，
387 即给事件赋予一个日期和时间。每个计算机有它们自己的物理时钟。这些时钟是电子设备，

计算有固定频率的晶体的振荡发生次数，把计量值分割一下，保存在计数器寄存器中。时钟设备可以被编程以按一定间隔产生中断用于实现时间片之类的功能；我们可以不关心这个方面的时钟操作。

操作系统读取结点的硬件时钟值$H_i(t)$，按一定比例放大，再加上一个偏移量，从而产生软件时钟$C_i(t) = \alpha H_i(t) + \beta$ 用于近似度量进程p_i的实际物理时间t。换句话说，当在一个绝对参照系中的实际时间为t时，$C_i(t)$是软件时钟的读数，例如，$C_i(t)$可以是从一个方便的参考时间开始的已流逝的以纳秒数累计的64位值，通常，时钟不完全准确，所以$C_i(t)$与t不一样。然而，如果$C_i(t)$表现得相当好（我们将马上研究时钟正确的概念），我们能用它的值给p_i的事件打时间戳。注意，只要时钟分辨率（时钟值修改的周期）比连续事件之间的时间间隔小，连续的事件就能相对于不同的时间戳。事件发生的速率依赖于处理器指令周期长度这样的因素。

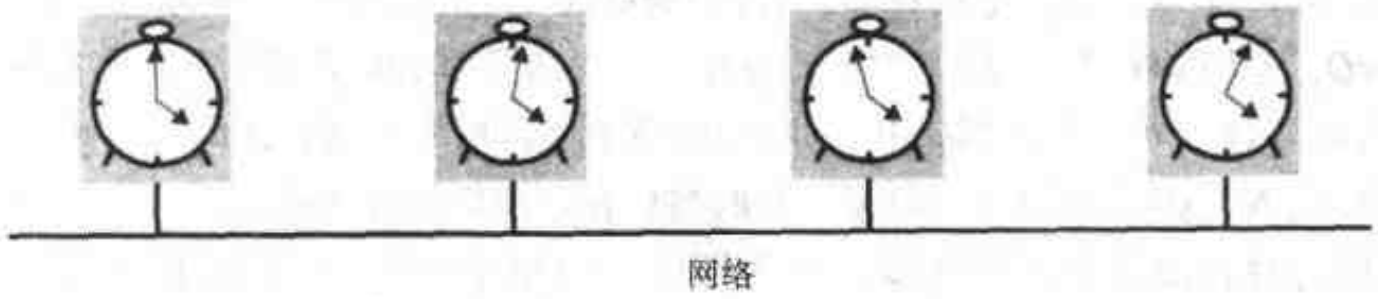


图10-1 分布式系统中计算机时钟之间的偏移

时钟偏移和时钟飘移 计算机时钟与其他时钟一样，并没有完美的一致性（图10-1）。两个时钟的读数之间的瞬间不同称为时钟偏移。在计算机中使用的基于晶体的时钟和其他时钟一样有时钟飘移问题，即它们以不同的频率给事件计数，所以会产生差异。时钟的振荡器在物理上会有不同，结果是振荡器的频率会有不同。有些设计试图弥补这种不同的时钟，但这些设计不能消除这个问题。两个时钟之间的振荡周期的不同可能相对很小，经过许多次的累加仍会形成在时钟计数器中可观察到的差异，不论这两个时钟它们的初始值是多么的一致。时钟的飘移率是指对由参考时钟度量的每个单位时间，在时钟和名义上完美的参考时钟之间的偏移量。对于普通的基于石英晶体的时钟，这大约在10^{-6}s/s，即每1 000 000s或11.6天有1s的差别。"高精度"的石英钟的飘移率大约为10^{-7}或10^{-8}。

通用协调时间 计算机时钟能与外部的高精度时间源同步。最准确的物理时钟使用原子振荡器，它的飘移率大约为10^{-13}。这些原子时钟的输出被用作实际时间的标准，称为国际原子时间（international atomic time）。从1967年起，标准的秒被定义为铯133（Cs^{133}）在两个层次之间的跃迁周期的9192631770倍。 388

秒、年和其他我们使用的时间单位来源于天文时间。它们原来按地球的自转和公转定义。然而，地球自转周期在慢慢变长，主要因为潮汐的反作用；大气的影响和地球内核的对流也导致周期短期的增加和减少。所以天文时间和原子时间步调不一致。

通用协调时间（coordinated universal time，UTC，这一缩写由法语得来）是国际计时标准。它基于原子时间，但不时要增加闰秒或在极偶尔的情况下要删除闰秒，以便同天文时间保持一致。UTC信号由覆盖世界大部分地方的广播电台和卫星进行同步和广播，例如，在美国，广播电台WWV用几个短波频率广播时间信号，卫星设备包括全球定位系统（global positioning system，GPS）。

接收器可从商家获得。与"完美的"UTC相比，从陆地站接收的信号具有0.1ms~10ms级

的精度，这取决于所使用的广播站。从GPS接收的信号能精确到1ms。与接收器相连的计算机能用这些时序信号同步它们的时钟。计算机也能通过电话线从诸如美国国家标准和技术研究所这样的组织接收时间，其精度大约几个毫秒。

10.3 同步物理时钟

为了知道在分布式系统$\mathcal{P}$的进程中发生的事件是在哪个时间——例如，为了进行会计工作——有必要用权威的外部时间源同步进程的时钟C_i。这是外部同步。如果时钟C_i相互同步到一个已知的精度，那么我们能通过本地时钟度量发生在不同计算机上的两个事件的间隔——即使它们没有必要与时间的外部源同步。这是内部同步。我们在实际时间I的一个间隔上定义如下的两个同步模式：

- 外部同步　对一个同步范围$D>0$，在I中对UTC时间源S的所有实际时间t，满足$|S(t) - C_i(t)|<D$，其中$i = 1, 2, \cdots, N$，该定义的另一种说法是时钟C_i在范围D中是准确的。
- 内部同步　对一个同步范围$D>0$，对I中的所有实际时间t，满足$|C_i(t) - C_j(t)|<D$，其中$i, j = 1, 2, \cdots, N$，该定义的另一种说法是时钟C_i在范围D中是一致的。

389 内部同步的时钟没必要外部同步，因为即使它们相互一致，它们整体上与时间的外部源仍有偏差。然而，根据定义，如果系统$\mathcal{P}$在范围D内是外部同步的，那么同一系统在范围$2D$内是内部同步的。

时钟正确性概念有不同的提法。一般一个硬件时钟H如果它的偏移率在一个已知的范围$\rho>0$内（该值从制造商处获得，例如10^{-6}s/s），那么它就被定义成是正确的。这表明度量实际时间t和t'（$t'>t$）的时间间隔的误差是有界的：

$$(1-\rho)(t'-t) \leqslant H(t') - H(t) \leqslant (1+\rho)(t'-t)$$

该条件禁止了硬件时钟值的跳跃（在正常操作中）。有时我们也要求软件时钟遵循该条件。但还有一个较弱的单调性条件可以满足，单调性是指一个时钟C前进的条件：

$$t' > t \Rightarrow C(t') > C(t)$$

例如，UNIX的*make*是一个工具，用于编译那些自上一次编译以来被修改的源文件。*make*将源文件和相应的目标文件的修改日期进行比较，以便决定是否编译。如果一台计算机时钟运行得快了，在编译源文件后修改源文件前把该时钟调整正确，那么源文件可能显得在编译前被修改了，*make*就会不编译该源文件，这是错误的。

尽管时钟被发现运行快了，我们还是能获得单调性的。我们仅需要改变比率，使得对时间的更新与应用一样。可不改变硬件时钟滴答的比率而用软件获得这一点——回忆等式$C_i(t) = \alpha H_i(t) + \beta$，这里我们可自由选择$\alpha$和$\beta$的值。

有时使用的一个混合正确性的条件是要求时钟遵循单调性条件，同时它的漂移率在两个同步点之间是有界的，但是在同步点允许时钟值可跳跃前进。

不满足任一正确性条件的时钟被定义成是有错误的。时钟完全停止滴答称为时钟的崩溃故障。其他时钟故障是随机故障。有千年虫的时钟故障就是此类故障的例子，它破坏了单调性条件，在1999年12月31日后把日期登记成1900年1月1日，而不是2000年1月1日。另一个例子是时钟的电池不足，它的漂移率会突然变得很大。

注意，根据定义，时钟不必非常准确才是正确的。因为目标可以是内部同步而不是外部

同步，正确的标准仅仅与时钟“机制”的正常运行有关，而不是它的绝对设置。

现在描述外部同步和内部同步的算法。

390

10.3.1 同步系统中的同步

考虑最简单的情况：在一个同步分布式系统中，两个进程之间的内部同步。在同步系统中，已知时钟漂移率的范围、最大的消息传输延迟和进程每一步的执行时间（见2.3.1节）。

一个进程在消息m中将本地时钟的时间t发送到另一个进程。原则上，接收进程能将它的时钟设成$t+T_{trans}$，其中T_{trans}是传输m所花的时间。两个时钟应该能一致（因为目标是内部同步，它不管发送进程的时钟是否精确）。

但T_{trans}是常常变化和未知的。通常，其他进程与要同步的进程在各自的结点上竞争资源，其他消息与m竞争网络。如果没有其他进程执行，没有其他网络通信，总有一个最小的传输时间min可以获得，min可以被度量或适当地估计。

根据定义，在一个同步系统中，用于传输消息的时间也有一个上界max。设消息传输时间的不确定性为u，那么$u=(max-min)$。如果接收方将它的时钟设成$t+min$，那么时钟偏移至多为u，因为事实上消息可能花了max时间才到达。类似的，如果将时钟设成$t+max$，那么时钟偏移可能与u一样大，然而，如果将时钟设成中间点$t+(max+min)/2$，那么时钟偏移至多为$u/2$。通常，对一个同步系统，同步N个时钟时，可获得的时钟偏移最优范围是$u(1-1/N)$[Lundelius and Lynch 1984]。

大多数实际的分布式系统是异步的：导致消息延迟的因素有很多，消息传输延迟没有上界max，在因特网上尤其如此。对一个异步系统，我们只能说$T_{trans}=min+x$，其中$x\geq 0$。x的值是不知道的，虽然对特定的环境，其值的分布是可以度量的。

10.3.2 同步时钟的Cristian方法

Cristian[1989]建议使用一个时间服务器，它连接到一个接收UTC信号的设备上，用于实现外部同步。在接收到请求后，服务器进程S根据它的时钟提供时间，如图10-2所示。Cristian观察到虽然在异步系统中消息传输延迟没有上界，但在一对进程之间进行消息交换的往返时间通常相当短——1s的一小部分时间。他把算法描述成有条件的：只有在客户和服务器之间的往返时间与所要求的精确性相比足够短，该方法才能达到同步。

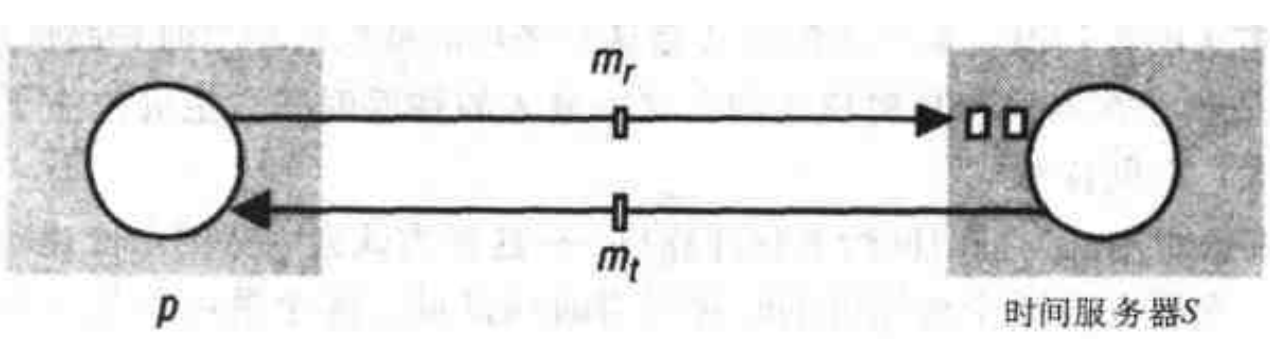


图10-2 用时间服务器的时钟同步

进程p在消息m_r中请求时间，在消息m_t中接收时间值t（t在从S的计算机传送之前的最后可能时刻插入到m_t）。进程p记录了发送请求m_r和接收应答m_t的整个往返时间T_{round}。如果时钟偏移率小，该值可相当精确地度量这段时间。例如，往返时间在LAN上应该是1ms~10ms数量级，偏移率为10^{-6}s/s的时钟在这段时间里变化至多10^{-5}ms。

在m_t中假设往返时间在S放置t之前和之后进行平分，那么对进程p应该设置它的时钟的一

个简单估计时间是$t + T_{round}/2$。正常情况下，这是一个相当精确的假设，除非两个消息在不同的网络上传递。如果最小传输时间min的值是已知的或者能保守地估计，那么我们能像下面一样决定结果的精确性。

S能在m_i中放置时间的最早点是在p发出m_i之后的min。它能做此工作的最近时间点是在m_i到达p之前的min。因此，应答消息到达时S的时钟的时间在$[t+min, t+T_{round} - min]$范围内。这个范围的宽度是$T_{round} - 2min$，所以精确度是$\pm (T_{round}/2 - min)$。

通过给S发几个请求（间隔发送请求以便清除暂时的拥堵）并用T_{round}的最小值给出最精确的估计，可在一定程度上处理可变性。精确性要求越高，达到它的可能性越小。这是因为最精确的结果要求两个消息在接近min的时间中传输——在繁忙的网络中，这是不太可能的。

关于Cristian算法的讨论 正如已描述的一样，Cristian方法存在的问题与所有由单个服务器实现的服务相关，单个时间服务器可能出现故障，以致于暂时不能同步。因为这个理由，Cristian建议，时间应该由一组同步时间服务器提供，每一个都带一个UTC时间信号接收器。例如，一个客户能将它的请求组播到所有服务器并仅使用获得的第一个应答。

用假的时间值做应答的有毛病的时间服务器或故意用不正确的时间做应答的假冒的时间服务器会引起计算机系统中的灾难。这些问题超出了Cristian[1989]所描述的工作的范围，Cristian假设外部时间信号源是自检测的。Cristian和Fetzer[1994]描述了内部时钟同步的条件协议族，其中每一个都能容忍某类故障。Srikanth和Toueg[1987]首先描述了一个算法，它在容忍一些故障的同时，在同步时钟的精确性上是最优的。Dolev等[1986]认为，如果f是所有N个时钟中出错时钟的个数，那么如果其他正确的时钟仍能达成一致，必须满足$N > 3f$。处理出错时钟的问题部分可由下面描述的Berkeley算法解决。恶意干扰时间同步的问题由认证技术来处理。

391 ~ 392

10.3.3 Berkeley算法

Gusella和Zatti[1989]描述了一个内部同步的算法，用于运行Berkeley UNIX的计算机群。在该算法中，选择一台协调者计算机用作主机。与Cristian协议不同，这个计算机周期性地调查其他时钟要被同步的计算机（称为从属机）。从属机将它们的时钟值返回给主机。主机通过观察往返时间（类似Cristian的技术），估计它们的本地时钟时间，并计算所获得值（包括它自己时钟的读数）的平均值。概率上的均衡指这个平均值能抵偿单个时钟跑快或跑慢的趋势。协议的精确性依赖于在主机和从属机之间名义上最大的往返时间。主机排除了任何偶尔的比这个最大值更大的时间读数。

主机不是发送更新的当前时间给其他计算机——这种方式会因为消息传递时间而引入更多的不确定性——而是发送每个从属机的时钟所需的调整量。这个量可以是一个正数或是一个负数。

算法排除了有错误的时钟读数。如果用一个普通平均值的话，这种有错的时钟会产生重大的负面影响。主机采用了容错平均值。就是说，在时钟中选择差值不多于一个指定量的子集，平均值仅根据这些时钟的读数计算。

Gusella和Zatti描述了涉及到15台计算机的实验，用他们的协议这些计算机的时钟可同步在20ms ~ 25ms之内。本地的时钟漂移率要小于2×10^{-5}，最大的往返时间为10ms。

如果主机出了故障，要能选举另一个接管，并像它的前任一样工作。11.3节讨论了一些通

用的选举算法。注意，它们并不保证在有限时间内选出一个新的主机，所以如果使用它们，在两个时钟之间的差异会是不受约束。

10.3.4 网络时间协议

Cristian的方法和Berkeley算法主要在企业内部网中使用。网络时间协议（NTP）[Mills 1995]定义了时间服务的体系结构和在因特网上发布时间信息的协议。

下面是NTP主要的设计目标和特色：

- 提供一个服务，使得跨因特网的用户能精确地同UTC同步　尽管在因特网通信中会遇到大的可变的消息延迟，NTP还是采用了过滤时序数据的统计技术，它用于辨别不同服务器的时序数据。
- 提供一个能在长久的连接损耗中生存的可靠服务　提供冗余的服务器以及服务器之间冗余的路径。如果其中一个不可达，服务器能重新配置以便继续提供服务。
- 使得客户能经常有效地重新同步以抵消在大多数计算机中存在的偏移率　服务被设计成能扩展到大量客户和服务器的情况。
- 提供保护，防止对时间服务的干扰，无论是恶意的还是偶然的　时间服务使用认证技术，用于检查来自声称是可信源的时序数据。它也验证发送给它的消息的返回地址。

通过因特网上的服务器网提供NTP服务。主服务器直接连到像收音机时钟这样的接收UTC的时间源；二级服务器最终与主服务器同步。服务器位于称为同步子网的逻辑层次结构中（见图10-3），其中的分层叫层次。主服务器占据层次1；它们在根上。层次2的服务器是与主服务器直接同步的二级服务器；层次3的服务器与层次2的服务器同步，等等。最低层（叶子）服务器在用户的工作站上执行。

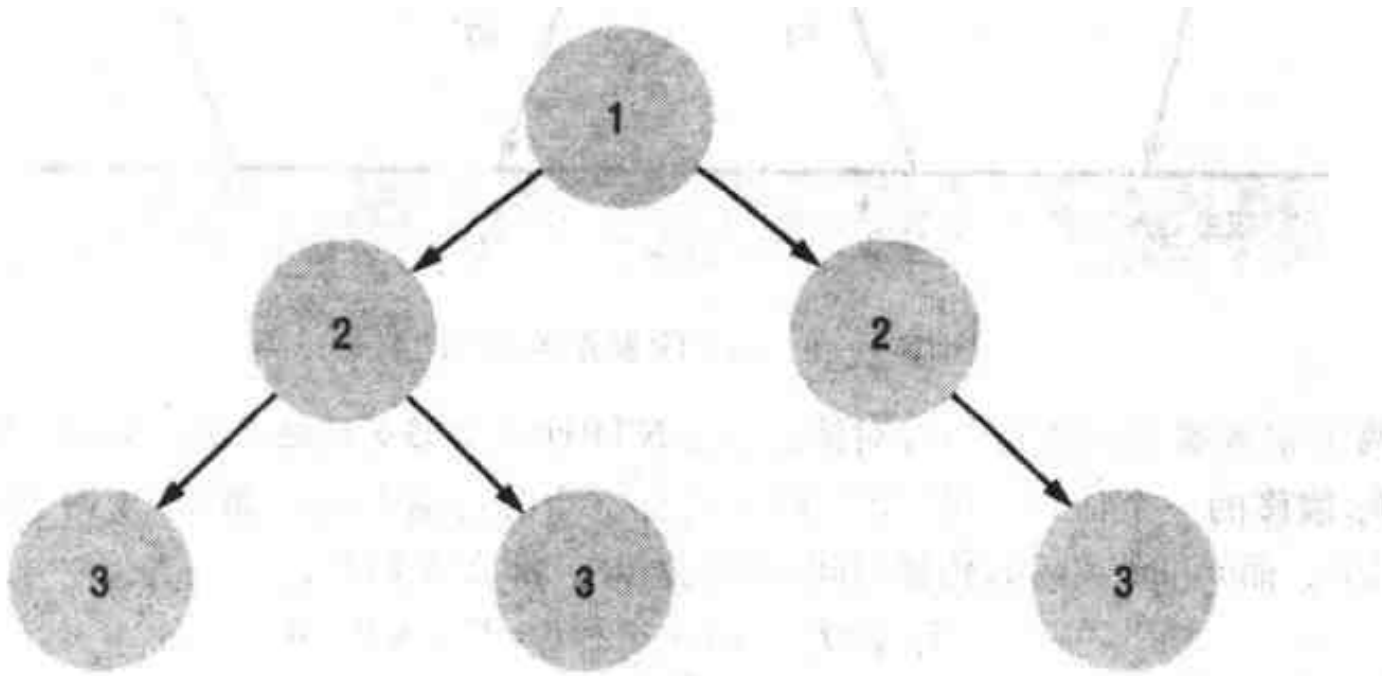


图10-3　用NTP实现的同步子网的例子

属于层次数大的服务器上的时钟比层次数小的更容易不精确，因为在任何同步层都会引入错误。NTP在评估出一个指定服务器拥有的计时数据的质量时，也考虑了整个消息到根的往返时间延迟。

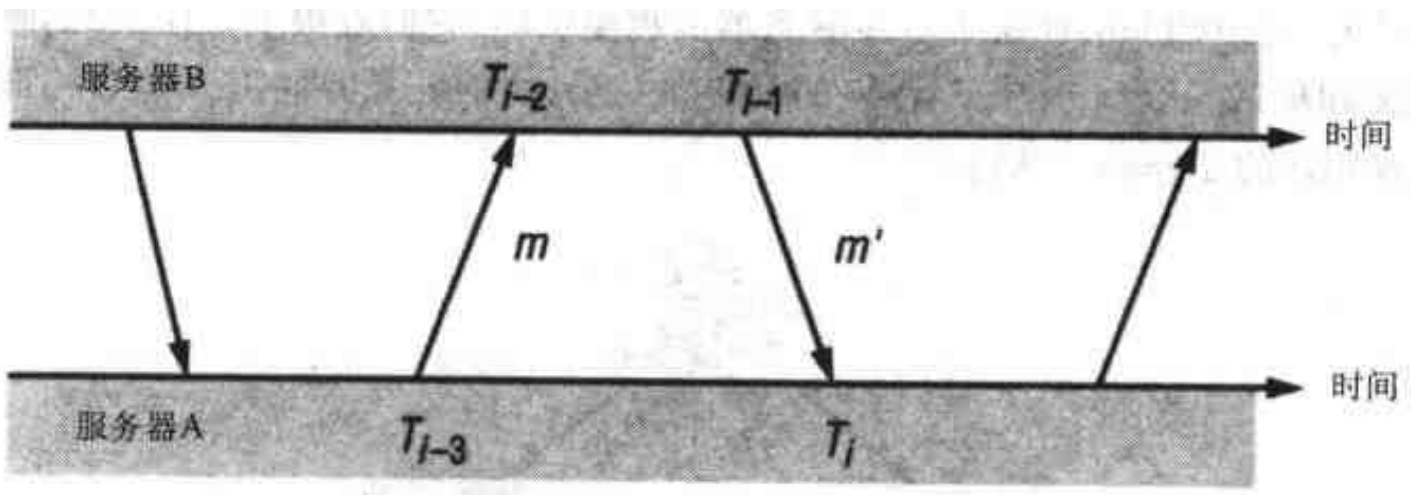
在服务器不可达或出现故障时，同步子网能重新配置。例如，如果主服务器的UTC源出现故障，那么它能变成层次2的二级服务器。如果二级服务器的常规同步源出现故障或变得不可达，那么它可以与另一个服务器同步。

NTP服务器按3种模式中的一种相互同步：组播，过程调用，对称模式。组播模式用于高速LAN。一个或多个服务器周期性地将时间组播到由LAN连接的其他计算机上的服务器，并设置它们的时钟（假设延迟很小）。这个模式仅能获得相对较低的精确性，但对许多目的而言，它已足够的。

过程调用模式类似上述Cristian算法的操作。在这个模式下，一个服务器从其他计算机接收请求，并用时间戳（当前的时钟读数）应答。这个模式适合精确性要求比组播更高的地方——或不能用硬件支持组播的地方。例如，在同一LAN或邻近LAN中的文件服务器，它们需要为文件访问保留精确的时序信息，可以以过程调用模式与本地服务器打交道。

最后，对称模式用于在LAN中提供时间信息的服务器和同步子网的较高层（较小的层次号），即要获得最高精确性的地方。按对称模式操作的一对服务器交换有时序信息的消息。时序数据作为服务器之间的关联的一部分被保留，由服务器维护时序数据以便提高时间同步的精确性。

在所有的模式中，消息传递用标准UDP因特网传输协议，是不可靠的。在过程调用模式和对称模式中，进程交换消息对。每个消息有最近消息事件的时间戳：发送和接收前一个NTP消息的本地时间，发送当前消息的本地时间。NTP消息的接收者记录它接收消息的本地时间。图10-4给出了在服务器A和B之间发送的消息m和m'的4个时间T_{i-3}、T_{i-2}、T_{i-1}、T_i。注意在对称模式中，与上面描述的Cristian算法不一样，在一个消息的到达和另一个消息的发送之间可以存在不可忽视的延迟，消息也可能丢失，但是由每个消息携带的3个时间戳仍是有效的。



395

图10-4 在一对NTP服务器之间的消息交换

对两个服务器之间发送的每对消息，由NTP计算偏移o_i和延迟d_i，偏移o_i是对在两个时钟之间实际偏移的一个估计，延迟d_i是两个消息整个的传输时间。如果B上时钟相对于A的真正偏移量是o，而m和m'实际的传输时间分别是t和t'，那么我们有：

$$T_{i-2}=T_{i-3}+t+o \text{ 和} T_i=T_{i-1}+t'-o$$

由它推出：

$$d_i=t+t'=T_{i-2}-T_{i-3}+T_i-T_{i-1}$$

以及

$$o=o_i+(t'-t)/2 \text{，其中} o_i=(T_{i-2}-T_{i-3}+T_{i-1}-T_i)/2$$

利用t，$t'\geq 0$的事实，有$o_i-d_i/2\leq o\leq o_i+d_i/2$。这样$o_i$是偏移的估计，$d_i$是该估计的精确性的一个度量。

NTP服务器在连续的$<o_i, d_i>$对上应用数据过滤算法，用于估计偏移o并计算这个估计的质量（采用称为过滤离中趋势的统计值形式）。一个相对高的过滤离中趋势表示相对不可靠的

数据。保留8个最近的< o_i , d_i>对。与Cristian的算法一样，选择对应于最小值d_i 的o_j的值用于估计o。

与单个源通信得到的偏移值未必用于控制本地时钟，但是通常，一个NTP服务器参加与几个对等方的消息交换。除了应用到每个对等方交换的数据过滤，NTP还使用对等方选择算法。它检查从与几个对等方互换中获得的值，查找相对不可靠的值。这个算法的输出使服务器可以改变它主要用于同步的对等方。

具有较小层次号的对等方比在较大层次的服务器上的对等方更受欢迎，因为它们靠近主时间源。具有最低同步离中趋势的也相对地受欢迎。这是服务器和同步子网的根之间度量的过滤离中趋势之综合。(对等方在消息中交换同步离中趋势，这样就可以计算该总计值。)

NTP采用了一个阶段锁循环模型[Mills 1995]，它按照对偏移率的观察修改本地时钟的更新频率。举一个简单的例子，如果发现一个时钟总是以一定比例走快，如每小时快4s，那么为了弥补这一点，它的频率可稍微降低(用软件或硬件)。这样，时钟在两次同步间隔中的偏移会减少。 396

Mills提到同步精确性在因特网路径上是10ms数量级，在LAN上是1ms数量级。

10.4 逻辑时间和逻辑时钟

从任何单个进程的角度，事件可惟一地按照本地时钟显示的时间进行排序。但像Lamport[1978]指出的，由于我们不能通过分布式系统完美地同步时钟，因此通常我们不能使用物理时间找出在分布式系统中发生的任何一对事件的顺序。

通常，我们使用类似于物理因果关系的模式，但把它应用到分布式系统是为了给发生在不同进程里的事件排序。这个排序是基于下面既简单又显然的两点：

- 如果两个事件是发生在同一个进程p_i (i =1,2,… ,N)，那么它们发生的顺序是p_i观察到的顺序——这是我们上面定义的顺序$\rightarrow_i$。
- 每次消息在不同进程之间发送，发送消息的事件在接收消息的事件之前发生。

Lamport将推广这两种关系得到的偏序称为发生在先关系。有时它也称为因果序或潜在的因果序。

我们如下所示定义发生在先关系，用→表示：

HB1：如果∃进程p_i：$e \rightarrow_i e'$，那么$e \rightarrow e'$。

HB2：对任一消息m，$send(m) \rightarrow receive(m)$，

其中$send(m)$是发送消息的事件，$receive(m)$是接收消息的事件。

HB3：如果e，e'和e''是事件，且有$e \rightarrow e'$和$e' \rightarrow e''$，那么$e \rightarrow e''$。

这样，如果e和e'是事件，且$e \rightarrow e'$，那么我们能找到在一个或多个进程中发生的事件e_1，e_2，…，e_n有$e=e_1$，$e'=e_n$，并且，对i =1，2，…，$N-1$，在e_i和e_{i+1}之间或可以应用HB1或HB2。就是说，或者它们在同一个进程中连续发生，或存在一个消息m使得$e_i=send(m)$，$e_{i+1}=receive(m)$。事件e_1，e_2，…，e_n的顺序不必是惟一的。

图10-5中的3个进程p_1，p_2和p_3可用于说明关系→。可以看到$a \rightarrow b$，因为在进程p_1中事件按这个顺序发生($a \rightarrow_1 b$)，类似的有$c \rightarrow d$。进一步有$b \rightarrow c$，因为这些事件是发送和接收消息m_1，类似的有$d \rightarrow f$。结合这些关系，我们可以说，$a \rightarrow f$。

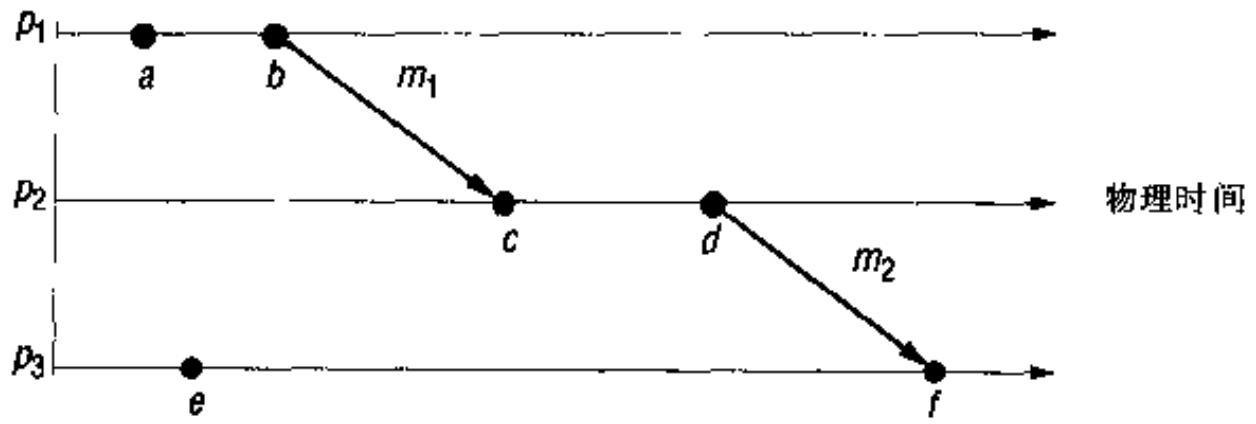


图10-5 发生在3个进程中的事件

从图10-5可以看出不是所有的事件与关系→相关。例如，$a \not\rightarrow e$和$e \not\rightarrow a$，因为他们发生在不同的进程中，且它们之间没有消息链。我们说像a和e这样不能由→排序的事件是并发的，写成$a \| e$。

关系→捕获了两个事件之间的数据流。但是要注意，原则上数据可以按非消息传递的方式流动。例如，如果Smith输入一条命令让进程发送一条消息，然后给Jones打电话，Jones让自己的进程发另一条消息，那么头一条消息的发送非常清楚地在第二条消息之前发生。但是，因为在进程之间没有发送网络消息，我们不能在我们的系统中为这种类型的关系建模。

要注意的另一点是，如果发生在先关系在两个事件之间成立，那么第一个事件可能或不可能实际地引起了第二个事件。例如，如果服务器接收一个请求消息，后来发送了一个应答，那么非常清楚，应答的传递是由请求的传递引起的。但是，关系→捕获可能的因果关系，两个事件即使没有真正的联系，也可以有→关系。例如，一个进程可能收到一个消息，后来又发了另一个消息，但这个消息是每五分钟一次的，与头一个消息没有特别的关系。这里，并没有实际的因果关系，但这些事件可以用关系→来排序。

逻辑时钟 Lamport发明了一种简单的机制，称为逻辑时钟，它可数字化地捕获发生在先排序。Lamport逻辑时钟是一个单调增长的软件计数器，它的值与任何物理时钟无关。每个进程p_i维护它自己的逻辑时钟，L_i，进程用它给事件加上所谓的Lamport时间戳。我们用$L_i(e)$表示p_i的事件e的时间戳，用$L(e)$表示发生在任一进程中的事件e的时间戳。

为了捕获发生在先关系→，进程按下列规则修改它们的逻辑时钟，并在消息中传递它们的逻辑时钟值：

LC1：在进程p_i发出每个事件之前L_i加1：

$L_i := L_i + 1$。

LC2：（a）当进程p_i发送消息m时，在m上捎带上值$t = L_i$。

（b）在接收（m，t）时，进程p_j计算$L_j := max(L_j, t)$，然后在给$receive(m)$事件打时间戳前应用$LC1$。

虽然上面给时钟的增量是1，但我们能选用任何正数。通过在与事件e和e'有关的事件序列上进行长度归纳，能很容易地看到：$e \rightarrow e' \Rightarrow L(e) < L(e')$。

注意相反的情况是不成立的。如果$L(e) < L(e')$，我们不能推出$e \rightarrow e'$。图10-6给出了对图10-5中给出的例子如何使用逻辑时钟。进程p_1，p_2和p_3都有各自的逻辑时钟，初始值为0。时钟值紧邻着事件给出。注意，例如，$L(b) > L(e)$但$b \| e$。

397
~
398

全序逻辑时钟 一些由不同进程生成的不同的事件对Lamport时间戳在数字值上可能相同。然而，我们可以创建事件的全序——就是说，所有的事件对都能排序——通过考虑发生事

件的进程的标识。如果e是在p_i发生的事件，本地时间戳为T_i，e'是在p_j发生的事件，本地时间戳为T_j，我们为这些事件分别定义全局逻辑时间戳（T_i,i）和（T_j,j）。当且仅当$T_i < T_j$或$T_i = T_j$以及$i<j$时定义（T_i, i）< （T_j, j）。这个排序没有通常的物理意义（因为进程标识是随机的），但它有时有用。例如，Lamport用它在一个临界区给进程排序。

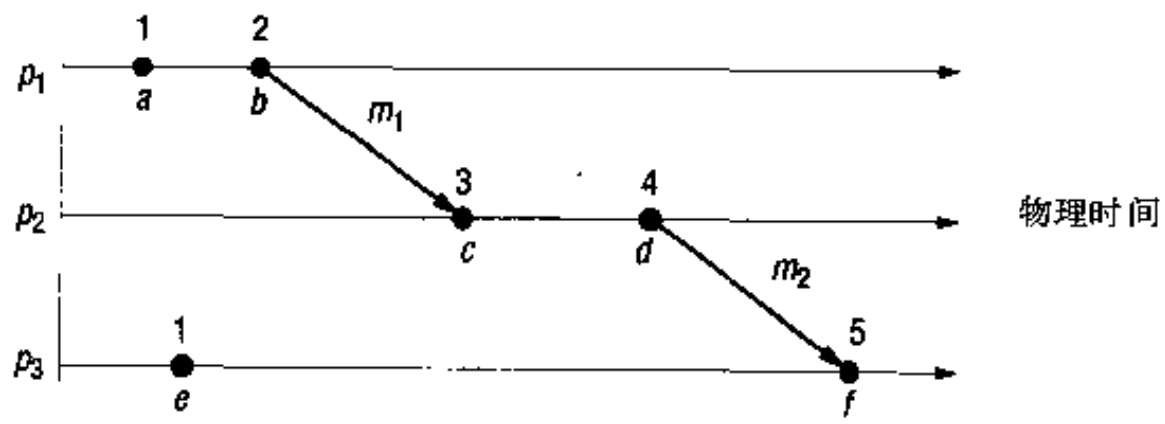


图10-6 图10-5给出的事件的Lamport时间戳

时钟向量 Mattern[1989]和Fidge[1991]开发了时钟向量用以克服Lamport时钟的缺点：从$L(e) < L(e')$，我们不能推出$e \rightarrow e'$。N个进程的系统的时钟向量是整数N的一个数组。每个进程维护它自己的时钟向量V_i，用于给本地事件加时间戳。类似Lamport时间戳，进程在发送给对方的消息上捎带时间戳向量，更新时钟的规则如下：

VC1：初始，$V_i[j] = 0$，对$i, j=1, 2, \cdots, N$。

VC2：在p_i给事件加时间戳之前，设置$V_i[i] := V_i[i] + 1$。

VC3：p_i在它发送的每个消息中包括值$t = V_i$。

VC4：当p_i接收到消息中的时间戳t时，设置$V_i[j] := max(V_i[j], t[j])$，$j=1, 2, \ldots, N$。像这样的取两个时间戳向量的最大值称为合并操作。

对时钟向量V_i，$V_i[i]$是p_i已经加了时间戳的事件的个数，$V_i[j]$（$j \neq i$）是在p_j中发生的可能会影响p_i的事件的个数（在这一时刻，进程可能给多个事件加时间戳，但至今没有信息流向p_i）。 399

我们用下列方法比较时间戳向量：

$$V = V' \ iff \ V[j] = V'[j] \text{ for } j = 1, 2, \cdots, N$$

$$V \leqslant V' \ iff \ V[j] \leqslant V'[j] \text{ for } j = 1, 2, \cdots, N$$

$$V < V' \ iff \ V \leqslant V' \wedge V \neq V'$$

设$V(e)$是发生e的进程所应用的时间戳向量。通过在与事件e和e'相关的事件序列的长度上进行归纳，可以看到有$e \rightarrow e' \Rightarrow V(e) < V(e')$。练习10.13要读者证明：如果$V(e) < V(e')$，那么$e \rightarrow e'$。

图10-7给出了图10-5中的事件的时间戳向量。从图上可以看到，$V(a) < V(f)$反映了$a \rightarrow f$的事实。类似地，通过比较时间戳，我们能区分何时两个事件是并发的。例如，从$V(c) \leqslant V(e)$和$V(e) \leqslant V(c)$均不成立的事实可推出$c \| e$。

与Lamport时间戳相比，时间戳向量的不足在于占用存储以及消息有效负载量与进程数N成正比。Charron-Bost[1991]证明了，如果我们能通过观察时间戳区分两个事件是否并发，那么维数N是不能避免的。但是，想要存储和传递小量数据，然后通过计算，重构完整的向量，这种技术是存在的。Raynal和Singhal[1996]对这些技术中的一部分给出了报道。他们还描述了矩阵钟的概念，进程凭借它保持自己和其他进程的向量时间。

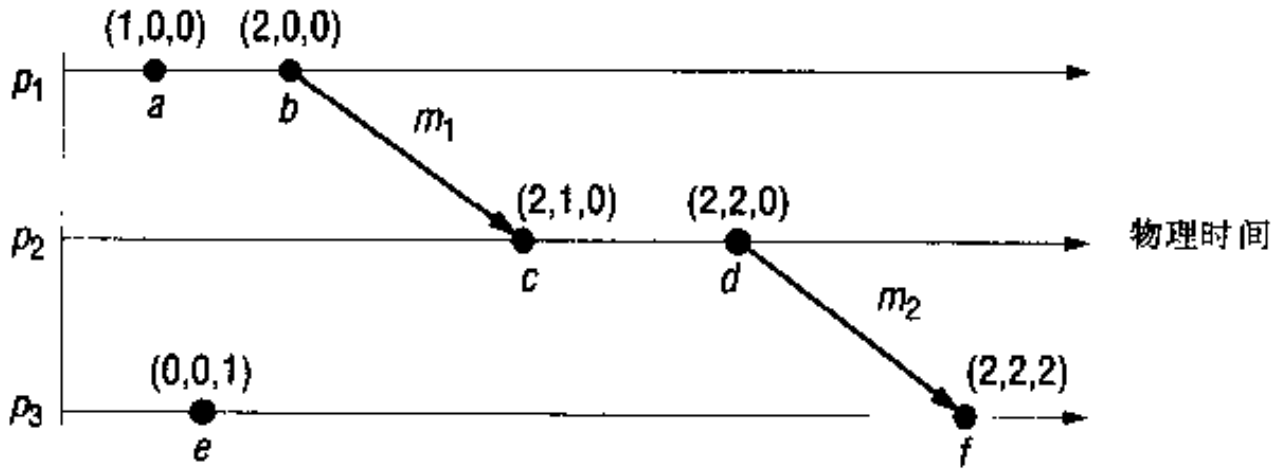


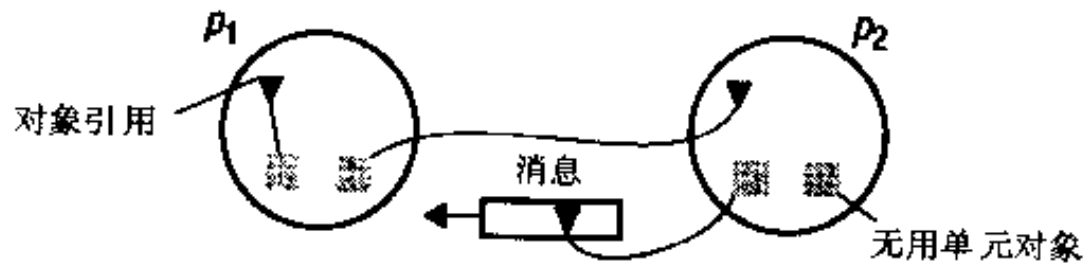
图10-7 图10-5给出的事件的时间戳向量

400

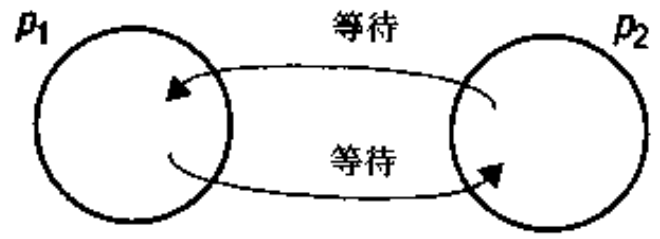
10.5 全局状态

本节和下节将研究查找分布式系统中的一个性质在系统执行时是否成立的问题。我们从分布式无用单元收集、死锁检测、终止检测和调试的例子开始。

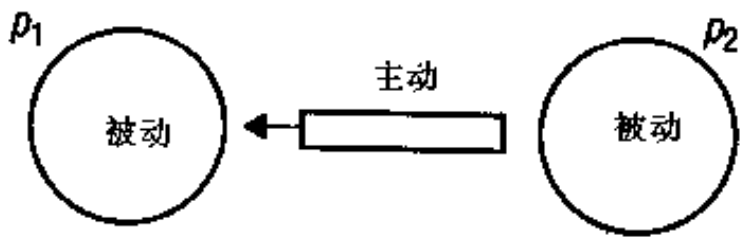
- *分布式无用单元收集* 如果在分布式系统中一个对象不再有任何对它的引用，那么它被认为是无用的。一旦知道对象是无用的，那么它所占据的内存就可以回收。为了检查一个对象是否是无用的，我们必须验证系统中没有任何地方引用它。在图10-8a中，进程p_1有两个对象，它们都有引用——一个引用在进程p_1内部，而进程p_2引用了另一个对象。进程p_2有一个无用对象，在系统中没有对它的引用。p_2还有一个对象，p_1和p_2都没有引用它，但在进程之间的暂态消息中有一个对它的引用。这说明了当我们考虑系统的性质时，我们必须包括信道的状态和进程的状态。



a. 无用单元回收



b. 死锁



c. 终止检测

图10-8 检测全局性质

- *分布式死锁检测* 当一组进程中的每一个都在等待另一个进程给它发送消息，并且在这种“等待”关系图中存在循环时，就会发生分布式死锁。图10-8b中进程p_1和p_2都在等待对方的消息，所以这个系统不会有任何进展。

- 分布式终止检测　这里的问题是检测一个分布式算法是否终止。检测终止是一个听起来很容易解决的问题：它看起来仅要测试是否每个进程都已经停止。为了说明并不是这样简单，考虑由两个进程p_1和p_2执行的一个分布式算法，每个进程可能请求另一个进程的值。我们能确定在一个瞬间一个进程是主动的或是被动的——一个被动的进程没有进行它自己的任何活动但准备回应另一个进程请求的值。假设我们发现p_1是被动的，p_2也是被动的（图10-8c）。

401

为了领会我们不能推断算法已经终止，考虑下列情形：当我们测试p_1的被动性时，一个消息正在从p_2到p_1的路上，p_2在发出该消息后马上又变成被动的。p_1接收消息，在我们发现它是被动之后又变成主动的。因此算法还没有终止。

终止和死锁的现象在一些方面有类似的地方，但它们是不同的问题。首先，死锁仅影响系统中的进程子集，而所有进程必须终止。其次，进程被动性与死锁循环中的等待不一样：死锁进程试图完成进一步的动作，该动作是另一个进程等待的；一个被动进程不参与任何活动。

- 分布式调试　分布式系统的调试非常复杂[Bonnaire *et al.* 1995]。要非常仔细才能确定系统执行中发生了什么。例如，Smith写的应用中，每个进程p_i包含了一个变量x_i(i =1, 2, ..., N)。变量随程序执行而改变，但它们被要求相互在一个δ值范围内。但是，程序中有一个缺陷，Smith怀疑在某种情况下对某些i和j有$|x_i - x_j| > \delta$，从而破坏了一致性限制。这里的问题是在变量值变化的同时要计算这种关系。

上述的每个问题有适合它的特定的解决方案，但它们都说明了有观察全局状态的需要，所以迫切需要一个通用的方案。

10.5.1　全局状态和一致割集

从原理上观察单个进程的连续状态是可能的，但如何查明系统的全局状态问题——进程集的状态——是非常难解决的。

本质的问题是缺乏全局时间。如果所有进程有完全同步的时钟，那么我们可以在同一时间让每个进程记录下它的状态的时间——结果将是系统实际的全局状态。从进程状态集，我们能区分进程是否死锁等。但我们不能获得完美的时钟同步，所以这个方法不适用。

我们可能问：从在不同时间记录的本地状态能否汇总出一个有意义的全局状态。回答是有条件的肯定，为了说明这一点，我们先引入一些定义。

回到有N个进程p_i(i = 1, 2, ⋯, N)的一般系统$\wp$中，我们将研究它的执行。我们在上面说过在每个进程中发生了一系列的事件，我们可以通过每个进程的历史来描述每个进程的执行：

$$history(p_i) = h_i = <e^0_i, e^1_i, e^2_i, \cdots>$$

类似地，我们可以考虑进程历史的任何一个有限前缀：

$$h^k_i = <e^0_i, e^1_i, \cdots, e^k_i>$$

402

每个事件或是进程的内部动作（例如，修改一个变量）或是在与进程相连的信道上发送或接收一个消息。

原理上，我们能记录在$\wp$执行中发生的一切。每个进程能记录本进程发生的事件，以及它经过的连续状态。我们用s^k_i表示进程p_i在第k个事件发生之前的状态，所以s^0_i是p_i的初始

状态。我们注意到在上面的例子中信道的状态有时是相关的。我们不引入新的状态类型，作为进程状态的一部分，我们让进程记录所有消息的发送或接收。如果我们发现进程p_i已经记录它发送了消息m到进程p_j $(i \neq j)$，那么通过检查p_j是否接收到该消息，我们能推断出m是否是p_i和p_j之间通道状态的一部分。

通过取单个进程历史的并集，我们能形成$\mathcal{H}$的*全局历史*：

$$H = h_0 \cup h_1 \cup \cdots \cup h_{N-1}$$

数学上，我们能取单个进程状态的任一集合来形成一个全局状态$S = (s_1, s_2, \cdots, s_N)$。但是哪个全局状态是有意义的——就是说，哪些进程状态能在同一个时间发生？一个全局状态相应与单个进程历史的初始前缀。系统执行的割集是系统全局历史的子集，是进程历史前缀的并集：

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \cdots \cup h_N^{c_N}$$

在（相对于割集C的）全局状态S中的状态s_i是在由p_i处理的最后一个事件即$e_i^{c_i}$ $(i = 1, 2, \cdots, N)$之后的p_i的状态。事件集$\{e_i^{c_i} : i = 1, 2, \cdots, N\}$称为割集的*边界*。

考虑图10-9中给出的在进程p_1和p_2发生的事件。该图给出了两个割集，一个割集边界是$< e_1^0, e_2^0 >$，另一个割集边界是$< e_1^2, e_2^0 >$。最左的割集是*不一致的*。这是因为在p_2中它包含了对消息m_1的接收，但在p_1中它不包含对该消息的发送。这是一个没有“原因”的“结果”。实际的执行不会处于该割集边界所对应的全局状态。从原理上，我们通过检查事件之间的$\rightarrow$关系可获得这一点。相反，最右割集是*一致的*，它包括消息的m_1的发送和接收，它也包括m_2的发送但不包括m_2的接收。这与实际执行相一致——毕竟，消息要花一些时间才能到达。

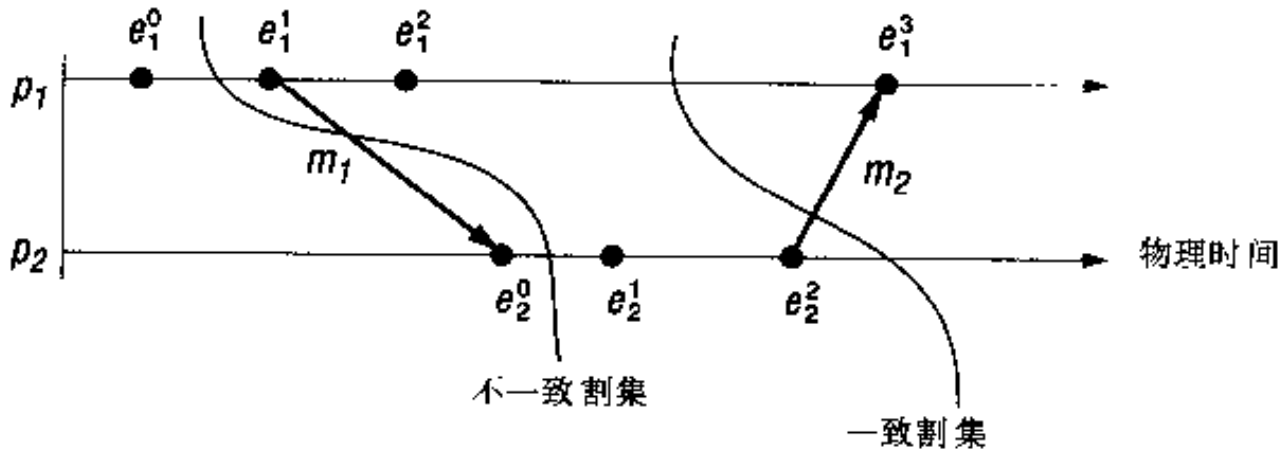


图10-9 割集

割集C如果是一致的话，对于它包含的每个事件，它也包含了所有在该事件之前发生的所有事件：

对所有事件 $e \in C, f \rightarrow e \Rightarrow f \in C$

*一致的全局状态*是指相对于一致割集的状态。我们可以把一个分布式系统的执行描述成在系统全局状态之间的一系列变迁：

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots$

在每个变迁中，系统保证一个事件只在单个进程中发生。这个事件或是发送消息，或是接收消息，或是一个外部事件。如果两个事件同时发生，我们可以认为它们按一定的顺序发生——按照进程标识排序（同时发生的事件必须是并发的；而不是一个在另一个之前发生。）系统通过一致全局状态这种方式逐步发展。

*走向*是全局历史中所有事件的全序，它与每个本地历史排序$\rightarrow_i$ $(i = 1, 2, \cdots, N)$一致的。

线性化走向或一致的走向是全局历史中所有事件的全序，与H上的发生在先关系→是一致的。

不是所有的走向都经历一致的全局状态，但所有线性化走向只经历一致的全局状态。我们说如果有一个经过S然后又经过S'的线性化走向，状态S'是从状态S可达的。

有时我们可以在一个线性化走向中变换并发事件的排序，得到的走向仍是经历一致全局状态的走向。例如，如果线性化走向中两个连续的事件是由两个进程接收消息，那么我们可以交换这两个事件的顺序。

10.5.2 全局状态谓词、稳定性、安全性和活性

检测一个像死锁和终止之类的条件实际上是求一个全局状态谓词的值。全局状态谓词是一个从系统$\mathcal{P}$的进程全局状态集映射到{*True*, *False*}的函数。与对象成为无用单元、系统死锁、系统终止状态相关的一个谓词特征是稳定的：一旦系统进入谓词为True的状态，它将在所有可从该状态可达的状态中一直保持*True*。相反，当我们监控或调试一个应用程序，我们经常对不稳定谓词感兴趣，如在前面变量的例子中，变量的差别是受限的，即使应用程序到达了受限范围内的一个状态，它也不必停留在这个状态。

我们还注意到与全局状态谓词有关的两个进一步的概念：安全性和活性。假设有一个不希望有的性质α，该性质是一个系统全局状态的谓词——例如，α是成为死锁的性质。设S_0是系统的原始状态。关于α的安全性是一个断言，即对所有可从S_0到达的所有状态S，α的值为*False*。相反，设β是系统全局状态希望有的性质——例如，到达终止的性质。关于β的活性是对任一从状态S_0开始的线性化走向L，对可从S_0到达的状态S_L，β的值为*True*。

403 ~ 404

10.5.3 Chandy和Lamport的“快照”算法

Chandy和Lamport[1985]描述了决定分布式系统全局状态的“快照”算法。算法的目的是记录进程集$p_i(i=1,2,\cdots, N)$的进程状态和通道状态集（“快照”），这样，即使所记录的状态组合可能从来没有在同一时间发生，但所记录的全局状态还是一致的。

我们将看到快照算法记录的状态能很方便地求稳定的全局谓词的值。

算法在进程本地记录状态，它没有给出在一个场地收集全局状态的方法。收集状态的一个显然的方法是让所有进程把它们记录的状态发送到一个指定的收集进程，但我们这里不对这个问题做进一步地讨论。

算法假设：

- 不论是通道还是进程都不出故障；通信是可靠的，因此每个发送的消息最终被完整、准确地接收到一次。
- 通道是单向的，提供FIFO顺序的消息传递。
- 描述进程和通道的图是强连接的（在任意两个进程之间都有一条路径）。
- 任一进程可在任一时间开始一个全局快照。
- 在拍快照时，进程可以继续它们的执行、发送和接收正常的消息。

对每个进程p_i，设进入通道是其他进程向p_i发送消息的通道；类似地，p_i的外出通道是p_i向其他进程发送消息的通道。算法的基本思想如下：每个进程记录它的状态，对每个接入通道还记录发送给它的消息。对每个通道，进程记录在它自己记录下状态之后和在发送方记录下

它自己状态之前到达的任何消息。这种安排可以记录不同时间的进程状态并且能用已传送但还没有接收到的消息说明进程状态之间的差别。如果进程p_i已经给进程p_j发送了一个消息m，但p_j还没有接收到，那么m属于它们之间通道的状态。

算法的处理是使用了特殊的标记消息，它与进程发送的其他消息不一样，可在正常执行中发送和接收。标记消息有双重作用：如果接收者还没有保存自己的状态，则提示其保存；决定哪些消息包括在通道状态中。

算法定义了两个规则：标记消息接收规则和标记消息发送规则（图10-10）。标记消息接收规则强制进程在记录下自己的状态但还没有发送其他消息之前发送一个标记。

标记消息接收规则强制没有记录状态的进程去记录状态。在这种情况下，这是它接收到的第一个标记消息。它记录在其他进入通道上后来到达了哪些消息。当一个已保存状态的进程接收到一个（在另一个通道上的）标记消息，它就把那个通道的状态记在从它保留它的状态以来所接收到的消息集中。

```
进程p_i 的标记消息接收规则
  p_i接收通道c上的标记消息；
  if (p_i还没有记录它的状态),
        p_i记录它的进程状态；
        将c的状态记为空集；
        开始记录从其他进入通道上到达的消息；
else
        p_i把c的状态记录到从保留它的状态以来它在c上接收到的消息集合中
end if
进程p_i 的标记消息发送规则
        在p_i记录了它的状态之后，对每个外出通道c：
        (在p_i从c上发送任何其他消息之前)
        p_i在c上发送一个标记消息
```

图10-10 Chandy和Lamport的“快照”算法

任何进程可以在任何时候开始这个算法，进程可假设它好像已接收到一个（在一个不存在的通道上的）标记消息，并遵循标记消息接收规则。这样，进程记录自己的状态并开始记录在所有进入通道上到达的消息。几个进程可以以这种方式并发地开始记录（只要能区别它们使用的标记消息）。

我们用一个系统来说明这个算法，这个系统有两个进程p_1和p_2，它们由两个单向通道c_1和c_2相连。两个进程进行“小部件”交易。进程p_1通过c_2向p_2发送小部件的订单，并以每个小部件10美元附上付款。一些时间以后，进程p_2沿通道c_1向p_1发送小部件。进程的初始状态如图10-11所示。进程p_2已经接收到5个窗口小部件的订单，它将马上分发给p_1。

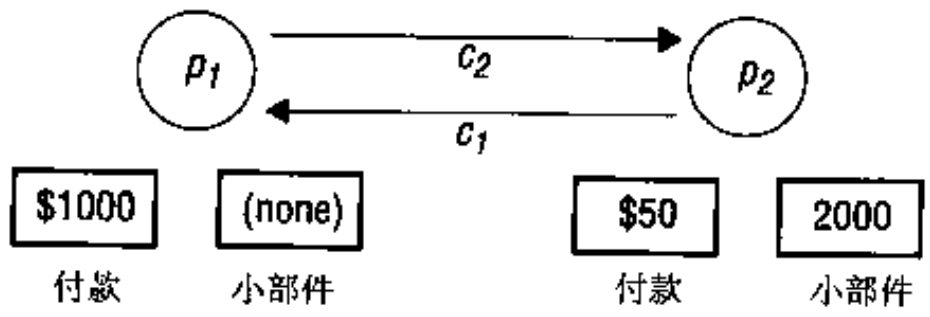


图10-11 两个进程和它们的初始状态

图10-12给出了系统的执行以及记录的状态。进程p_1在实际的全局状态S_0中记录它的状态，当时p_1的状态是<$1000, 0>。根据标记消息发送规则，进程$p_1$在它通过通道$c_2$发送下一个应用层消息(Order 10, $100)之前，在它的外出通道上发送一个标记消息。系统进入实际的全局状态S_1。

在p_2接收到标记消息之前，它通过c_1发送一个应用消息(five widgets)以响应p_1以前的订单，产生新的全局状态S_2。

现在进程p_1接收到p_2的消息(five widgets)，p_2接收到标记消息。根据标记消息接收规则，p_2将它的状态记录成<$50, 1995>，将通道$c_2$的状态记录成空序列。根据标记消息发送规则，它通过$c_1$发送标记消息。

当进程p_1接收到p_2的标记消息时，它将通道的状态记录成在它第一次记录它的状态之后接收到的那个消息(five widgets)。最后实际的全局状态是S_3。

最后记录的状态是p_1：<$1000, 0>；$p_2$：<$50,1995>；c_1：<(five widgets)>；c_2：<>。注意这个状态与系统实际经过的所有全局状态不同。

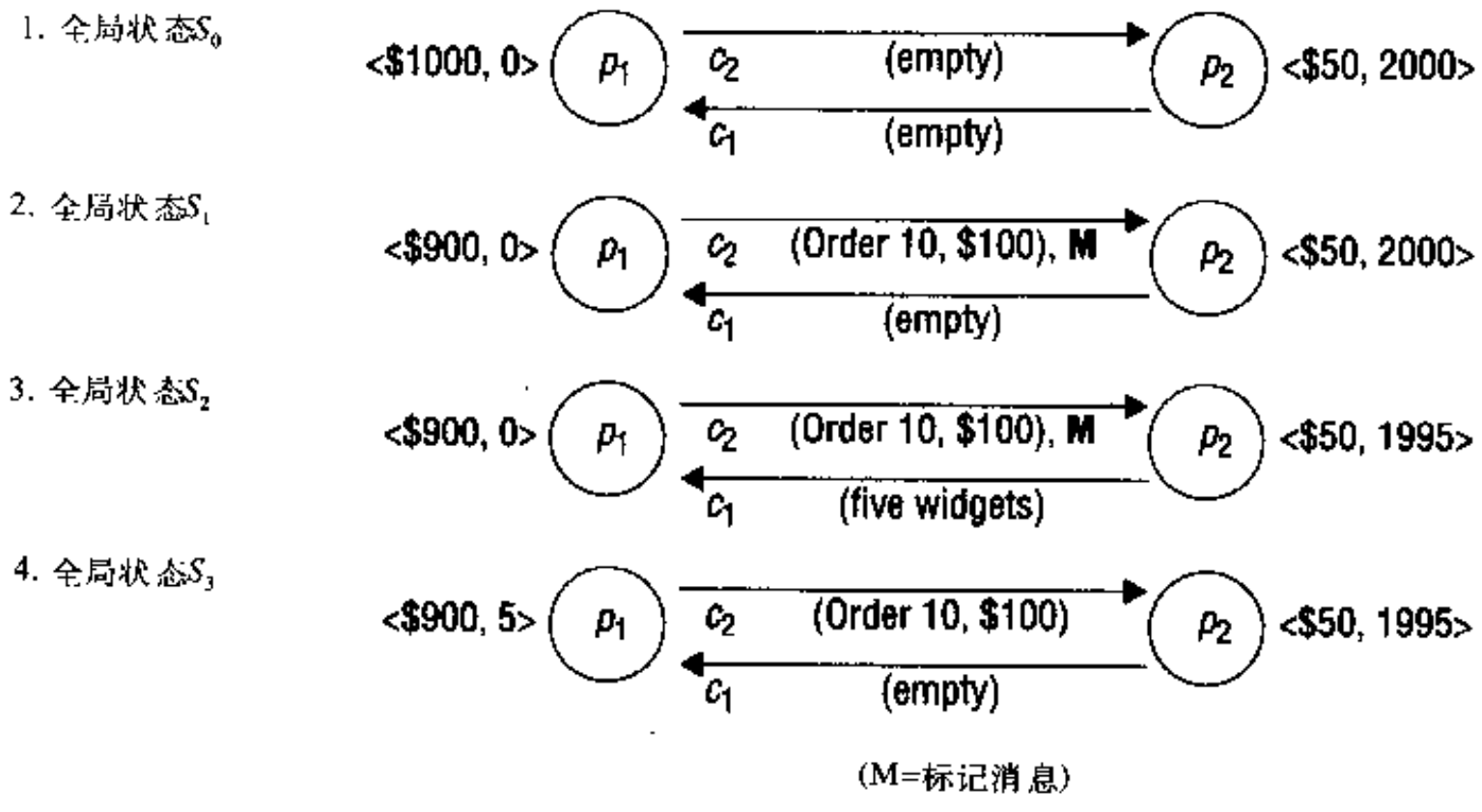


图10-12 图10-11中进程的执行

快照算法的终止 我们假设一个已经接收了一个标记消息的进程在有限的时间里记录了它的状态，并在有限的时间里通过每个外出通道发送了标记消息（甚至当它不再需要在这些通道上发送应用消息时）。如果有一条从进程p_i到进程p_j（$j \neq i$）的信道和进程的路径，那么可清楚地假设，在p_i记录它的状态之后的有限时间里p_j将记录它的状态。因为我们假设进程和通道图是强连通的，所以在一些进程记录它的初始状态之后的有限时间内，所有的进程将记录它们的状态和接入通道的状态。

405 ? 407

刻画所观察到的状态 快照算法从执行的历史中选择了一个割集，因此，割集与该算法记录的状态是一致的。为了看清这一点，设e_i和e_j分别是在p_i和p_j中发生的事件，且有$e_i \rightarrow e_j$。我们断言如果e_j在割集中，那么e_i也在割集中。就是说，如果e_j在p_j记录它的状态之前发生，那么e_i必须在p_i记录它的状态之前已经发生。如果两个进程是相同的，这一点非常明显，所以我们假设$j \neq i$。现在假设，我们要证明的反面命题成立：在e_i发生之前p_i记录了它的状态。考虑H个消息序列m_1、m_2、…、m_H $(H \geq 1)$，导致关系$e_i \rightarrow e_j$。通过在传递这些消息的通道上进行

FIFO排序，以及标记消息发送和接收规则，一个标记消息将在每个m_1、m_2、…、m_H之前到达p_j。根据标记消息接收规则，p_j将在事件e_j之前记录它的状态。这与我们的假设e_j在割集中相矛盾，所以得证。

我们将在根据算法运行时所观察到的全局状态与初始和最终的全局状态之间建立可达关系。设$Sys=e_0$，e_1，…是系统执行时的线性化走向（若两个事件在同一个时间发生，我们将按照进程标识给它们排序）。设S_{init}是在第一个进程记录它的状态之前的全局状态；S_{final}是在快照算法终止，最后一个状态记录动作之后的全局状态，S_{snap}是所记录的全局状态。

我们将找到Sys的一个排列，$Sys' = e'_0$，e'_1，e'_2，…使得所有3个状态S_{init}，S_{final}，S_{snap}都在Sys'中、S_{snap}可从Sys'中的S_{init}处到达，S_{final}可从Sys'中的S_{snap}处到达，图10-13给出了这种情况，上面的线性化走向是Sys，下面的线性化走向是Sys'。

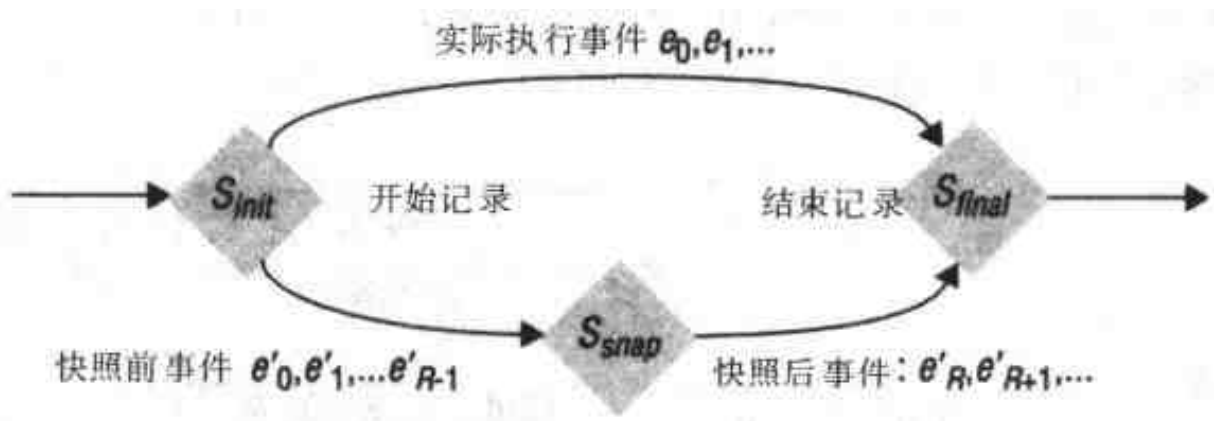


图10-13 在快照算法中状态之间的可达性

我们首先通过把Sys中的所有事件分成快照前事件或快照后事件，从Sys得到Sys'。进程p_i的快照前事件是在进程p_i记录它的状态之前发生的事件；所有其他事件是快照后事件。如果事件在不同的进程中发生，那么在Sys中快照后事件可以在快照前事件之前发生，理解这一点很重要（当然在同一进程中，快照前事件之前不可能发生快照后事件）。

我们将给出在快照后事件之前给快照前事件排序的方法以便获得Sys'。假设e_j是一个进程
408 的快照后事件，而e_{j+1}是另一个进程的快照前事件。它不能是$e_j \rightarrow e_{j+1}$。这两个事件可能分别是一个消息的发送和接收。标记消息必须领先于消息，使得消息的接收是一个快照后事件，但根据假设e_{j+1}是一个快照前事件。因此我们可以不违反发生在先关系而交换两个事件（就是说，事件的结果序列仍然是一个线性化走向）。交换并不引入新的进程状态，因为我们没有改变任何单个进程发生的事件排序。

我们继续按这种方法交换相邻事件对，直到在Sys'执行结果中，所有快照前事件e'_0，e'_1，e'_2，…，e'_{R-1}排序在所有快照后事件e'_R，e'_{R+1}，e'_{R+2}，…之前。对每个进程，在e'_0，e'_1，e'_2，…，e'_{R-1}中的该进程发生的事件集正好是它在记录它的状态之前经历的事件集。因此，在那一时刻每个进程的状态和信道的状态就是算法记录的全局状态S_{snap}。我们不干扰线性化走向开始和结束的状态S_{init}和S_{final}。这样我们就建立了可达关系。

所观察到的状态的稳定性和可达关系 快照算法的可达性质对检测稳定谓词是有用的。通常，在状态S_{snap}中成为$True$的任何不稳定谓词在记录全局状态的实际执行中可以是$True$，也可以不是$True$。但是，如果在S_{snap}状态中稳定谓词为$True$，那么我们可以肯定在S_{final}状态中谓词也是$True$，因为由定义，一个状态S为$True$的稳定谓词在从S可达的任一状态都是$True$。类似地，如果在S_{snap}状态谓词为$False$，那么在S_{init}状态，该谓词也一定是$False$。

10.6 分布式调试

我们现在研究记录系统全局状态的问题，以便我们能对实际执行中暂态状态——与稳定状态相对——做出有用的判断。这是调试分布式系统时通常所要求的。上面我们给出了一个例子，即进程集合中的每一个进程p_i都有一个变量x_i。在这个例子中所要求的安全条件是$|x_i - x_j| \leqslant \delta$（$i$，$j = 1, 2, \cdots, N$）；即使进程可能在任何时候改变它的变量值，也要满足这个限制。另一个例子是一个控制工厂管道系统的分布式系统，这里我们感兴趣的是是否所有的值（由不同的进程控制）在某一个时间都是开放的。在这些例子里，通常我们不能同时观察变量的值或阀门的状态。这里的挑战是随时监控系统的执行——即捕获“跟踪”信息而不是单个快照——以便我们能在此之后了解所要求的安全条件是否成立或已被破坏。

Chandy和Lamport的快照算法以分布式的方式收集状态，我们指出了系统中的进程如何把它们收集的状态发送到一个监控进程中。下面描述的算法（归功于Marzullo和Neiger[1991]）是集中式的。被观察的进程将它们的状态发送到一个称为监控器的进程，它根据接受到的信息汇总成全局一致状态。我们认为监控器在系统之外观察系统的执行。 409

我们的目的是在我们所观察的系统执行点判定一个给定的全局状态谓词ϕ明确为*True*，以及它可能为*True*的情况。“可能”作为一个自然的概念出现，因为我们可以从一个执行系统中抽取一个一致的全局状态S并发现$\phi(S)$为*True*。仅仅观察一个一致的全局状态不是以使我们得出是否一个非稳定谓词在实际的执行中曾为*True*。不过，我们有兴趣了解是否它们可能发生，直到我们通过观察系统的执行来明确这一点。

“明确”这一概念应用于实际执行，而不是应用于我们推断的运作。考虑在实际的执行中发生了什么听起来有点荒谬，但是，通过考虑所观察事件的所有线性化走向情况求解得出ϕ是否是明确的*True*是可能的。

现在我们按照线性化走向定义概念可能的ϕ和明确的ϕ

- 可能的ϕ　可能的ϕ意味着存在一个一致的全局状态S，经历了H的一个线性化走向，而且该S使得$\phi(S)$为*True*。
- 明确的ϕ　明确的ϕ意味着对H的所有线性化走向L，存在L经历的一个一致的全局状态S，使得$\phi(S)$为*True*。

当我们使用Chandy和Lamport快照算法获得全局状态S_{snap}时，如果$\phi(S_{snap})$正好是*True*，我们就可以断言可能的ϕ。但通常，求解可能的ϕ需要对从所观察到的执行中得出的所有一致的全局状态进行搜索。仅对所有一致的全局状态S有$\phi(S)$为*False*，它还不是可能的ϕ。还要注意到虽然我们从¬可能的ϕ能得出明确的$(\neg\phi)$，但我们不能从明确的$(-\phi)$得出¬可能的ϕ。后者说明：在每个线性化走向中，在部分状态$\neg\phi$成立时，在另一部分状态ϕ成立。

我们现在描述：

- 进程状态如何收集。
- 监控器如何抽取一致的全局状态。
- 监控器如何在异步和同步系统中求解可能的ϕ和明确的ϕ。

收集状态　所观察的进程p_i（$i = 1, 2, \cdots, N$）在开始的时候用状态消息向监控器进程发送它们的初始状态，这以后是间隔地发送状态消息。监控器进程在单独的队列Q_i（$i = 1, 2, \cdots, N$）中记录来自进程p_i的状态消息。

准备和发送状态消息的活动可能会延迟所观察进程的正常执行，但其他方面没有受干扰。

除了初始时和状态改变时，其他时候没有必要发送状态。有两种优化技术可减少到监控器的状态消息流量。第一，全局状态谓词可以只依赖进程状态的某一部分。例如，它可以仅依赖特定变量的状态。所以所观察的进程仅需要向监控器进程发送相关状态。第二，进程仅在谓
410 词ϕ变成*True*或不再为*True*时发送它们的状态。发送不影响谓词值状态的变化值是没有意义的。

例如，在进程p_i应该遵循$|x_i - x_j| \leqslant \delta$ $(i, j = 1, 2, ..., N)$限制的系统例子中，进程仅需要在它们自己的变量x_i值改变时通知监控器。当它们发送状态时，仅需提供x_i的值而不需要发送任何其他变量。

10.6.1 观察一致的全局状态

为了计算ϕ，监控器必须汇总一致的全局状态。先回忆一下：一个割集C是一致的当且仅当对割集C中所有的事件e，$f \rightarrow e \Rightarrow f \in C$。

例如，图10-14给出了两个进程p_1和$p2$，分别有变量x_1和x_2。在时间线上的事件（用时间戳向量）是对两个变量的值做调整。初始时，$x_1 = x_2 = 0$。要求是$|x_1 - x_2| \leqslant 50$。进程对变量做调整，但“大的”调整将引起新值被发送到其他进程。当一个进程从另一个进程接收到一个调整消息时，它会把它的变量设成消息中所含的值。

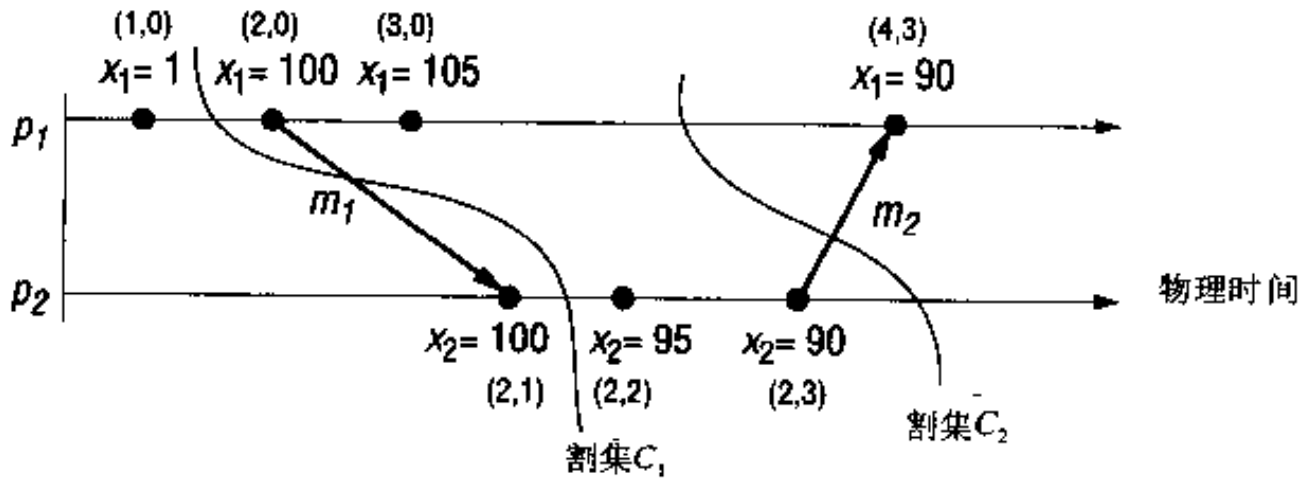


图10-14 执行图10-9产生的时间戳向量和变量值

每次进程p_1或p_2中的一个调整了它的变量值（不论是“小的”调整还是“大的”调整），它就通过状态消息给监控器进程发送一个值。监控器进程在为p_1、p_2而设置的队列中保存该消息用于分析。如果监控器进程使用图10-14中不一致割集C_1中的值，那么它将发现x_1=1，x_2=100，违反了限制$|x_1 - x_2| \leqslant 50$，但这个事件状态是不会发生的。另一方面，来自一致割集C_2的值显示x_1=105，x_2=90。

为了监控器能区分不一致的全局状态和一致的全局状态，被观察的进程在它们的状态消息中附上了时钟向量值。每个队列Q_i都以发送顺序排序，这是通过检查时间戳向量的第i个成分实现的。监控器程序可能因为变量消息有延迟而从到达次序上推断不出不同进程发送的状态的顺序。所以它必须检查状态消息的时间戳向量。

设$S = (s_1, s_2, \cdots, s_N)$是从监控器进程接收到的状态消息中得出的全局状态。设$V(s_i)$是从
411 p_i接收到的状态s_i的时间戳向量。那么S是一致的全局状态当且仅当：

$$V(s_i)[i] \geqslant V(s_j)[i] \quad (i, j = 1, 2, \cdots, N) \quad \text{——(CGS条件)}$$

这是说，当p_i发送s_j时，p_j知道的p_i的事件个数不多于在p_i发送s_i时在p_i发生的事件的个数。换句话说，如果一个进程的状态依赖于另一个进程的状态（根据发生在先排序），那么全局状态

也包含了它所依赖的状态。

总之，我们的方法是使用由被观察进程保持的时间戳向量和在被观察进程发送给监控器的状态消息上捎带信息。用该方法监控器进程可以得出一个给定的全局状态是否一致。

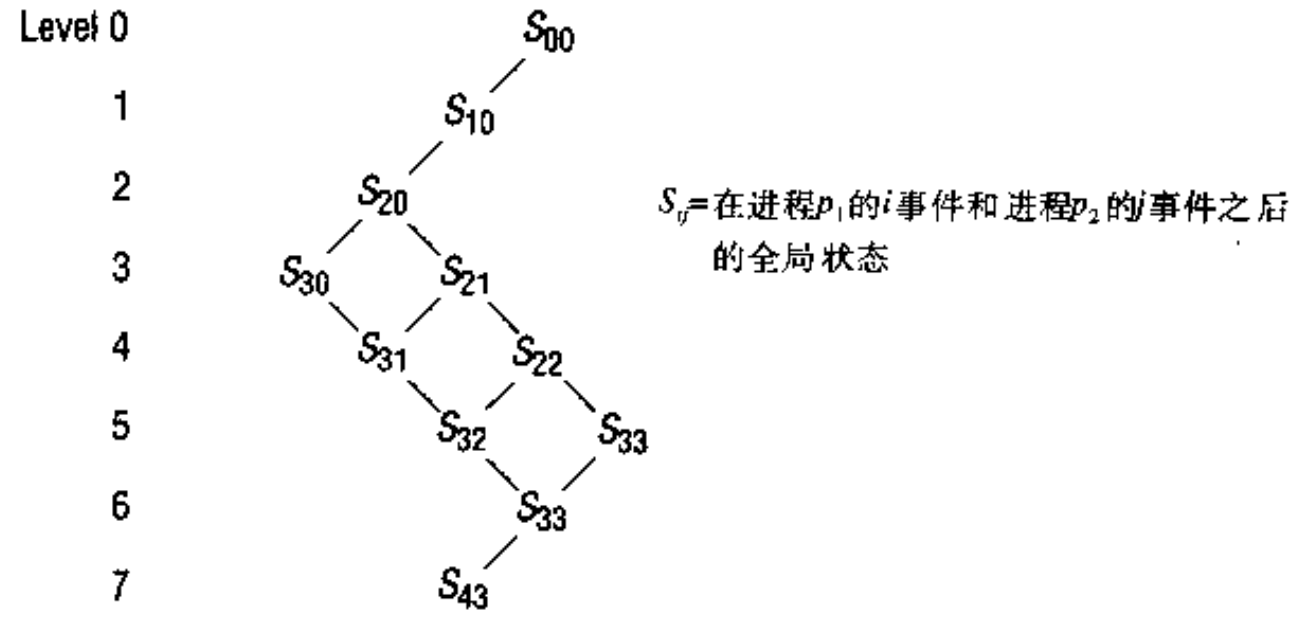


图10-15 执行图10-14产生的全局状态网格

图10-15给出了相对于图10-14的两个进程执行的一致的全局状态的网格。这个结构捕获了一致全局状态之间的可达性关系。结点表示全局状态，边表示状态之间可能的变迁。全局状态S_{00}表示在它们的初始状态中有两个进程；S_{10}中p_2仍在它的初始状态，p_1处在它的本地历史中的下一个状态。状态S_{01}不是一致的，因为消息m_1从p_1发送到p_2，所以它没有出现在网格中。

网格按层次排列，例如，S_{00}在层次0，S_{10}在层次1。通常，S_{ij}在（$i+j$）层次。线性化走向从任一全局状态开始遍历网格到达下一层的全局状态——就是说，在每一步，一些进程经历了一个事件。例如，S_{22}可从S_{20}到达，但S_{22}不能从S_{30}到达。

网格给出了相对于一个历史的所有线性化走向。现在从原理上能清楚地知道一个监控器进程如何求解可能的φ和明确的φ。为了求解可能的φ，监控器进程从初始状态开始，经过从这点开始可达的所有的一致状态，在每一步都求解φ。在φ的值为*True*时停止计算。为了求解明确的φ，监控器进程必须试图找到所有线性化走向必须经过的、φ求值为*True*的状态集。例如，如果图10-15中的$\phi(S_{30})$和$\phi(S_{21})$都是*True*，那么因为所有的线性化走向经过这些状态，所以明确的φ成立。

412

10.6.2 求解可能的φ

为了求解可能的φ，监控器进程必须从初始状态（$s_1^0, s_2^0, \cdots, s_N^0$）开始遍历可达状态的网格。算法见图10-16所示。算法假设执行是无限的，可以很容易改写成有限执行。

根据下列方法，监控器进程可以发现在$L+1$层中的可从L层一个给定的一致状态可达的一致状态集。设$S=(s_1, s_2, \cdots, s_N)$是一个一致的状态。那么从$S$可达的下一层的一致状态具有$S'=(s_1, s_2, \cdots, s'_i, \cdots, s_N)$的形式，它与$S$的不同仅仅在于包含了一些进程$p_i$的（在一个事件之后的）下一个状态。通过遍历状态消息Q_i（i = 1，2，…，N）的队列，监控器能找到所有这样的状态。状态S'从S可达当且仅当：

对$j=1, 2, \cdots, N, j \neq i$ 有：$V(s_j)[j] \geqslant V(s'_i)[j]$

该条件来自上面的CGS条件以及S已经是一个一致的全局状态这个事实。通常一个给定的状态

可从前一个层次的几个状态到达，所以监控器进程应该仅对每个状态求解一致性一次。

```
1. 对N个进程的全局历史H求解可能的φ
L := 0;
States := {(s⁰₁, s⁰₂,⋯,s⁰_N)};
while(对所有的S ∈ States, φ(S) = False)
    L:=L + 1;
    Reachable := {S': 从一些S ∈ States 可达H中的S'且level(S') = L};
    States:=Reachable
end while
输出 "可能的φ";
2. 对N个进程的全局历史H求解明确的φ
L := 0;
if (φ(s⁰₁, s⁰₂,⋯, s⁰_N)) then States := {} else States := {(s⁰₁, s⁰₂, ⋯, s⁰_N)};
while(States ≠ {})
    L:=L + 1;
    Reachable := {S': 从一些S ∈States 可达H中的S'且 level(S') = L};
    States:= { S ∈Reachable: φ(S) = False}
end while
输出 "明确的φ";
```

413

图10-16 求解可能的φ和明确的φ

10.6.3 求解明确的φ

为了求解明确的φ，监控器进程再次从初始状态（$s_1^0, s_2^0, \cdots, s_N^0$）开始，每次一层地遍历可达状态的网格。算法（见图10-16所示）又一次假设执行是无限的，但它可很容易地改编成有限的执行。它维护*States*集合，该集合包含了当前层的通过遍历φ为*False*的状态可从初始状态可达的状态。只要这样的线性化走向存在，我们就不可以对明确的φ做出断言：执行能采用这个线性化走向，φ在每个阶段将会是*False*。如果我们到达了一个不存在这样的线性化走向的层次，我们就能断定明确的φ。

在图10-17中，在第3层状态*States*仅由一个状态组成，它只要通过一个所有状态都是*False*（用粗线标记）的线性化走向就是可达的。在第4层所考虑的状态仅有一个标记为"F"的（右边的状态没有被考虑，因为它仅能通过φ值为*True*的状态到达）。如果φ在第5层的状态为*True*，那么我们可以断定明确的φ。否则，算法必须在这个层次上继续。

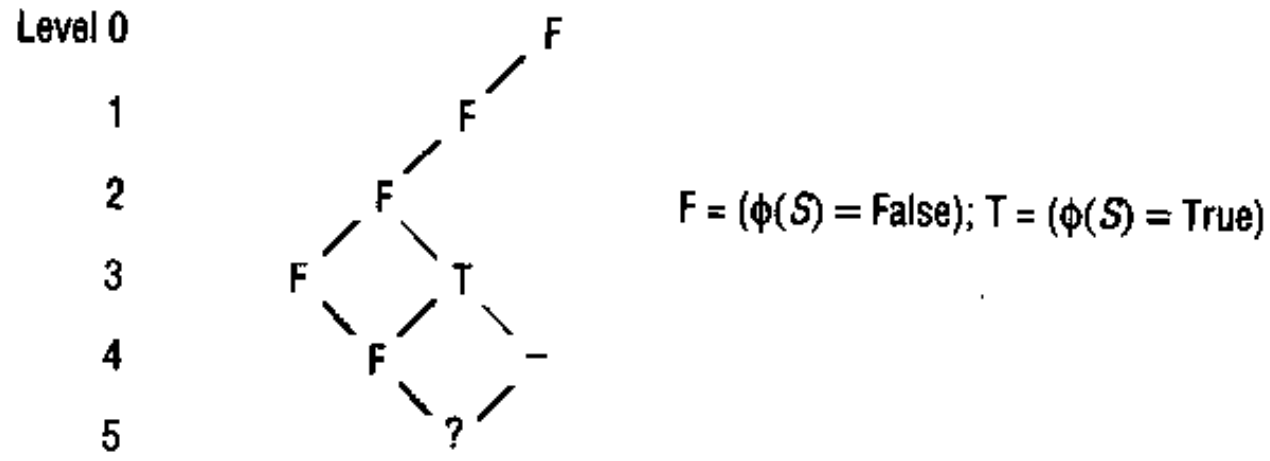


图10-17 求解明确的φ

开销 刚才描述的算法是组合爆炸的。假设k是在单个进程中的事件的最大个数。那么我

们描述的算法需要$O(k^N)$次比较（监控器进程相互比较N个所观察的进程的状态）。

这些算法的空间开销是$O(k^N)$。但是，我们观察到，当从另外进程到达的其他状态项不可能与包含s_i的一个一致的全局状态相关时，就是说，在下列条件成立时：

$$V(s_j^{last})[i] > V(s_i)[i] \text{ 对于} j=1,\ 2,\ \cdots,\ N,\ j \neq i$$

其中s_j^{last}是监控器进程从进程p_j接收到的最后的状态，那么，监控器进程可以从队列Q_i删除包含状态s_i的消息。

414

10.6.4 在同步系统中求解可能的φ和明确的φ

到目前为止我们所给出的算法是在一个异步系统中工作：我们没有设置时序的假设。但为此付出的代价是监控器所检查的一个一致的全局状态$S=(s_1,s_2,\cdots,s_N)$，在系统实际执行中，其中任意两个本地状态s_i和s_j可能发生间隔任意长的时间。而现在，我们的需求是仅考虑这些实际执行在原则上能遍历的全局状态。

在同步系统中，假设进程均保持它们的物理时钟内部同步在一个已知的范围，并假设所观察的进程在它们的状态消息中提供物理时间戳和时间戳向量。接着给定时钟的近似同步值，监控器进程仅需要考虑那些本地状态可能已经同时存在的一致全局状态。在足够精确的时钟同步条件下，这些状态将比所有的全局一致状态少。

我们现在给出一个算法按这种方式开发同步时钟。假设每个要观察的进程p_i（$i=1,\ 2,\ \ldots,\ N$）和监控器进程（我们称为p_0）保持一个物理时钟C_i（$i=0,\ 1,\ 2,\ \cdots,\ N$）。它们在一个已知的范围$D>0$内同步；就是说，在同一实际时间：

$$|C_i(t) - C_j(t)| < D \quad (i,\ j=0,\ 1,\ \cdots,\ N)$$

所观察的进程将带有时间向量和物理时间的状态消息发送给监控器进程。监控器进程应用一个条件，该条件不仅用于测试全局状态S的一致性，而且在给定物理时钟值时用于测试是否在同一实际时间能发生每对状态，换句话说，对$i,\ j=1,\ 2,\ \cdots,\ N$:

$V(s_i)[i] \geq V(s_j)[i]$，而且s_i和s_j能在同一实际时间发生。

条件的第一个部分是我们以前使用的条件。对第二个部分，注意p_i从它第一次通知监控器进程的时间$C_i(s_i)$到稍后的本地时间$L_i(s_i)$即在p_i发生下一个状态变迁的时候均处在状态s_i。考虑到时钟同步的边界问题，对在同一实际时间上获得的s_i和s_j，有：

$$C_i(s_i) - D \leq C_j(s_j) \leq L_i(s_i) + D，\text{反之亦然（交换}i\text{和}j\text{）}$$

监控器进程必须计算$L_i(s_i)$的值，用它与p_i的时钟相比。如果监控器进程已经接收到一个p_i的下一个状态$s'i$的状态消息，那么$L_i(s_i)$就是$C_i(s'_i)$。否则，监控器进程把$L_i(s_i)$估计为$C_0 - max + D$，其中C_0是监控器当前的本地时钟值，max是状态消息的最大传递时间。

415

10.7 小结

本章的开始描述了分布式系统精确计时的重要性，接着描述了同步时钟的算法，尽管存在时钟漂移和计算机之间消息延迟的可变性。

实际可获得的同步精确度可满足许多需求，但对确定发生在不同计算机上的任意事件对的排序还是不够的。发生在先关系是事件上的偏序关系，它反映了事件之间的信息流——或

在一个进程中反映，或在进程之间通过消息反映。一些算法要求事件按发生在先顺序排序，例如，后续的更新在数据的一个单独的备份里进行。Lamport时钟是一个计数器，它们依照事件之间的发生在先关系进行更新。时钟向量是Lamport时钟的改进，因为通过检查它们的时间戳向量，可以决定两个事件是否是按发生在先关系排序或是并发的。

我们引入了下列概念：事件、本地和全局历史、割集、本地和全局状态、走向、一致状态、线性化走向（一致走向）和可达性。一致状态或走向是与发生在先关系一致的状态。

接着，我们考虑通过观察系统执行记录一致全局状态的问题。我们的目的是为在这个状态上的谓词求解。一类重要的谓词是稳定谓词。我们描述了Chandy和Lamport的快照算法，它捕获一致全局状态，并可以就一个稳定谓词是否在实际执行中成立做出判断。接着我们给出了Marzullo和Neiger的算法，用于判断一个谓词是否在实际的运行中成立或可能成立。算法采用了一个监控器进程收集状态。监控器检查时间戳向量来抽取一致的全局状态，它构造并检查所有一致全局状态的网格。这个算法的计算复杂性很高，但对理解很有价值，它在只有相对少的事件改变全局谓词值的实际系统中运行得很好。这个算法在时钟可以同步的同步系统中有一个更有效的变种。

练习

10.1 为什么计算机时钟同步是必要的？描述用于同步分布式系统中的时钟的系统的设计需求。

10.2 当发现一个时钟快4s时，它的读数是10:27:54.0（小时:分钟:秒）。解释为什么在这时不愿将时钟设成正确的时间，并（用数字表示）给出它应该如何调整以便在8s后变成正确的时间。

10.3 一种实现至多一次的可靠消息传递的模式是使用同步时钟拒收重复的消息。进程在
416 它们发送的消息中放上它们本地的时钟值（一个“时间戳”）。每个接收者为每个发送进程维护一张表，在其中给出了它已看到的最大的消息时间戳。假设时钟被同步在100ms范围以内，消息在传递后至多50ms能到达。

（1）如果一个进程已经记录了从另一个进程接收到的最后的消息有时间戳T，那么这个进程何时能忽略具有时间戳T的消息？

（2）何时接收方能从它的表格中删除时间戳175 000ms？（提示：使用接收者本地的时钟值。）

（3）时钟能内部同步或外部同步吗？

10.4 一个客户试图与一个时间服务器同步。它在下表中记录了由服务器返回的往返时间和时间戳。

下面哪个时间可以用于设置它的时钟？它应该设成什么时间？与服务器时钟相比，估计设置的精确性。如果已知系统发送消息和接收消息之间的时间是至少8ms，那么你的答案应该如何改变？

往返时间（ms）	时间（h：min：s）
22	10:54:23.674
25	10:54:25.450
20	10:54:28.342

10.5 在练习10.4的系统中，要求与文件服务器时钟同步在±1ms的范围内。讨论它与

Cristian算法的关系。

10.6 在NTP同步子网中，你希望发生怎样的重配置？

10.7 一个NTP服务器B在16:34:23.480接收来自服务器A的带有时间戳16:34:13.430的消息，并对消息给出应答。A在16:34:15.725接收到带有B的时间戳16:34:25.7的消息。估计B和A之间的偏差和估计的精确性。

10.8 当决定一个客户应该与哪一个NTP服务器同步它的时间时，需要考虑的因素。

10.9 通过观察时间上的漂移率，讨论补偿同步点之间的时钟偏移的可能的方法。讨论该方法的局限性。

10.10 通过考虑连接事件e和e'的零或多个消息链，使用归纳方法，证明$e \rightarrow e' \Rightarrow L(e) < L(e')$。

10.11 证明 $V_j[i] \leqslant V_i[i]$。

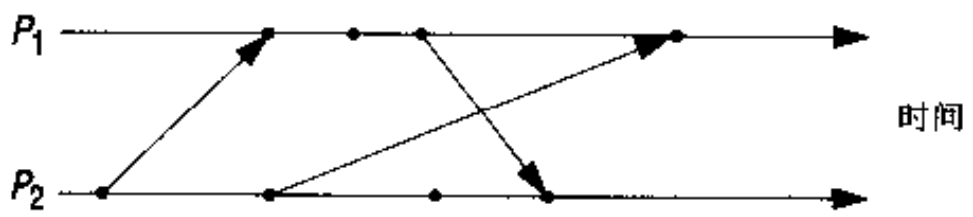
10.12 按练习10.10类似的方式，证明$e \rightarrow e' \Rightarrow V(e) < V(e')$。

10.13 利用练习10.11的结果证明，如果事件e和e'是并发的，那么$V(e) \leqslant V(e')$ 和$V(e') \leqslant V(e)$均不成立。因此证明：如果$V(e) < V(e')$，那么有$e \rightarrow e'$。

10.14 两个进程P和Q用两个通道连成一个环，它们不断地轮转消息m。在任何一个时刻，系统中仅有一份m的拷贝。每个进程状态由它接收到m的次数组成，P首先发送m。在某一点，P得到消息且它的状态是101。在发送m之后，P启动快照算法。给定由快照算法报告的可能的全局状态，试解释该情况下算法的操作。

10.15 下图给出了在两个进程p_1和p_2中发生的事件。进程之间的箭头表示消息传递。

从初始状态（0,0）开始，画出并标注一致状态（p_1的状态，p_2的状态）的网格。



10.16 Jones正在运行一组进程p_1，p_2，…，p_N。每个进程p_i包含一个变量v_i。她希望判定所有变量v_1，v_2，…，v_N在执行中是否相等。

（i）Jones的进程在同步系统中运行。她使用一个监控器进程判定变量是否相等。应用进程何时应该与监控器进程通信，它们的消息应该包含什么？

（ii）解释语句：可能的$(v_1 = v_2 = \cdots = v_N)$。Jones如何能判定该语句在她的执行中成立。

第11章　协调和协定

11.1　简介
11.2　分布式互斥
11.3　选举
11.4　组播通信
11.5　共识和相关问题
11.6　小结

本章介绍的一些话题和算法与如下问题有关：分布式系统中的进程如何协调它们的动作和对共享值达成协议（不考虑故障）。本章的开始介绍实现一组进程互斥的算法，可用于协调这些进程对共享资源的访问。接下来研究在分布式系统中如何实现选举，即，在前一协调者出现故障后，一组进程如何能就新协调者达成一致。

本章后半部分研究与组播通信、共识、拜占庭协定和交互一致性有关的问题。在组播中，问题是对于像消息发送顺序这样的事情如何达成协议。共识和其他的问题是由如下问题归纳而来：任何一组进程如何对一些值达成协议（不管这些值的值域是什么）。我们遇到了分布式系统理论中的一个基本结果：在某些条件下（甚至包括良性故障条件），不可能保证进程会达成共识。

11.1　简介

本章介绍一组算法，这些算法目标不同，但却都具有分布式系统的一个基本目的：供一组进程来协调它们的动作或对一个或多个值达成协议。例如，在像太空船这样的复杂设备情况下，一个基本要求是控制它的各个计算机对太空船的任务是在继续还是已经终止等诸如此类的条件达成协议。此外，各个计算机必须正确地协调它们关于共享资源（太空船的传感器和传动装置）的动作。计算机必须能做这些，即使在各个成分之间没有固定的主－从关系（主－从关系使协调变得简单）的地方。避免固定的主－从关系的原因是，我们经常需要系统即使在故障出现时也能保持正确工作，因此就需要避免单个结点例如固定的主控器的故障。

正如在第10章中那样，对于我们来说一个重要的差别是所研究的分布式系统是异步的还是同步的。在异步系统中不做时序上的假设。在同步系统中，我们假设消息传送的最大延迟、进程的每步运行时间以及时钟漂移率都有约束。这些同步假设允许我们用超时来检测进程崩溃。

除了讨论算法，本章的另一个重要目的是考虑故障以及在设计算法时如何处理故障。2.3.2节介绍的一个故障模型将用于本章。处理故障是一个精细的工作，因此我们先考虑一些不容许故障的算法，然后是针对良性故障的进展，直到考虑怎样容许随机故障。我们遇到了分布式系统理论中的一个基本结果：即使在良性故障条件下，在异步系统中也不可能保证一组进程能对一个共享值达成协议——例如太空船的所有控制进程对“继续任务”或“放弃任务”达成协议。

11.2节研究分布式互斥问题。这是大家熟悉的在内核和多线程应用中避免竞争条件的问题

在分布式系统的扩展。由于在分布式系统中大多都是资源共享，因此这是一个重要的要解决的问题。随后，11.3节介绍一个相关但更一般的问题，即如何“选举”一组进程中的一个来完成特定任务。例如，在第10章中，我们看到进程如何把时钟同步到一个指定的时间服务器。如果这个服务器出了故障，而多个生存的服务器可以完成这一任务，那么为了一致性起见，必须只选择一个服务器来接管。

组播通信是11.4节的主题。正如在4.5.1节解释的，组播是一个非常有用的通信范型，从定位资源到协调复制数据的更新都有相应的应用。11.4节研究组播可靠性和排序语义，给出了多种算法。组播传递本质上是进程间的协定问题：接收者对接收哪些消息和按什么顺序接
420 收达成一致。11.5节更一般性地讨论协定问题，主要形式是共识和拜占庭协定。

本章后面的论述包括陈述假设和要达到的目标，以及以非形式化方式解释所列的算法为何是正确的。此处没有足够的空间来提供更严格的论述。读者可参考给出分布式算法详尽说明的参考书，如Attiya和Welch [1998]和 Lynch [1996]。

在给出问题和算法之前，我们先讨论在分布式系统中的故障假设和实际的检测故障问题。

故障假设和故障检测器

为简单起见，本章假设每对进程之间都由可靠的通道连接。即，尽管底层网络组件可能有故障，但进程使用可靠通信协议能屏蔽故障——例如通过重传丢失或损坏的消息。为简单起见，我们还假设进程故障不隐含对其他进程的通信能力的威胁。这意味着没有进程依赖于其他进程来转发消息。

注意一个可靠的通道最终将消息传递到接收者的输入缓冲区。在同步系统中，我们假设在需要的地方有硬件冗余，这使得尽管有底层故障，可靠通道不仅能最终传递每个消息，而且能在指定时间限制内完成。

在任何特定时间间隔内，一些进程之间的通信可能成功，而另一些进程之间的通信则被延迟。例如，两个网络之间的路由器故障可能意味着，4个进程被分为两对，每个网络内的进程对通信是可能的，但两对进程间通信在路由器故障时是不可能的。这被称为网络分区（图11-1）。在一个点对点的网络上，如因特网，复杂的拓扑结构和独立的路由选择意味着连接可能是非对称的：从进程p到进程q的通信是可能的，但反过来不行。连接还可能是非传递的：从进程p到进程q和从进程q到进程r的通信都是可能的，但p不能直接与r通信。因此我们的可靠性假设要假设任何有故障的连接或路由器最终会被修复或避开。然而，所有进程不能够同时进行通信。

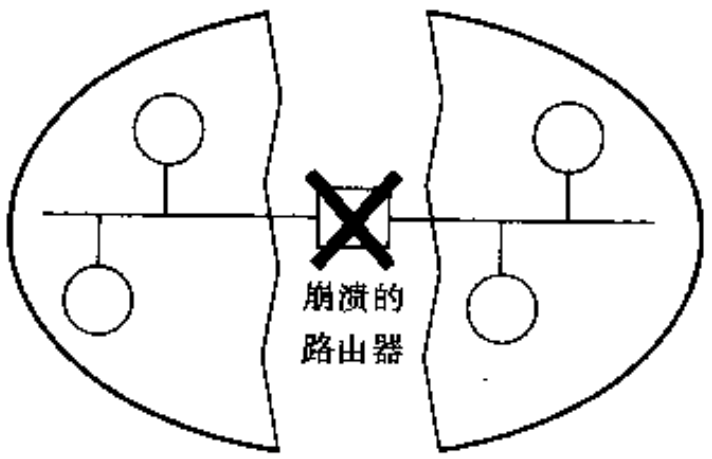


图11-1 网络分区

421

本章假定，除非特别说明，进程只在崩溃时出故障——这个假定对许多系统来说都足够了。在11.5节，我们将考虑如何对待进程有随机（拜占庭）故障的情况。不论何种故障，一个正确的进程是在所考虑的运行中的任何点都没有故障的进程。注意正确性是应用于整个运行，而非仅仅应用于运行的一部分。因此一个出现崩溃故障的进程在这一点之前是“无故障”的，但不是“正确”的。

设计克服进程崩溃的算法中遇到的问题之一是判断进程何时已经崩溃。一个故障检测器[Chandra and Toueg 1996, Stelling *et al.* 1998]是一个服务，进程利用该服务可以询问某个特定进程是否已经出现故障。故障检测器通常是由（同一计算机上的）每个进程中的一个对象实现的，此对象与其他进程的对应部分一起执行一个故障检测算法。每个进程中的这个对象叫做本地故障检测器。我们稍后将介绍如何实现故障检测器，但首先我们将专注于故障检测器的一些性质。

一个故障检测器没有必要精确。它们大多属于不可靠故障检测器的范畴。当给出一个进程标识时，一个不可靠故障检测器可以产生下列两个值之一：*Unsuspected*和*Suspected*。这两种结果都只是提示，这种提示可能精确地也可能不精确地反映进程是否确实出故障了。结果*Unsuspected*表示检测器最近已收到表明进程没有故障的证据，例如，最近从某一进程收到一个消息。但是那个进程可能从那时起就有故障了。结果*Suspected*表示故障检测器有迹象表明进程可能已经出故障了。例如，迹象可能是在多于最长沉默时间里没有收到来自进程的消息（即使在异步系统里，实际使用的上限也可被用作提示信息）。这样的怀疑可能是错的：例如，进程可能运行正确，但在网络分区的另一边；或者进程可能运行得比预期的慢得多。

可靠的故障检测器是能精确检测进程故障的检测器。对于进程的询问，它回答*Unsuspected*——与前面一样，这只是一个提示——或*Failed*。结果*Failed*表示检测器检测到进程已崩溃。如前所述，已崩溃进程会一直这样下去，因为根据定义，进程一旦崩溃就不会再采取其他步骤。

意识到以下一点是重要的，即尽管我们说一个故障检测器是作用于一组进程的，但是故障检测器对一个进程的应答实际上只是相当于该进程可用的信息。故障检测器有时会对不同的进程给出不同的应答，因为不同进程的通信条件不同。

我们可以用下述算法实现不可靠的故障检测器。每个进程p向所有其他进程发送消息“p is here”，并且每T秒发送一次。故障检测器用最大消息传输时间D秒作为评估值。如果进程q的本地故障检测器在最后一次$T+D$秒内没有收到“p is here”的消息，则向q报告p是*Suspected*。然而，如果后来收到“p is here”的消息，则向q报告p是*OK*。

在实际的分布式系统中，存在消息传送时间的实际限制。即使电子信箱系统也会在几天后放弃，因此很可能通信连接和路由器在此时间里已被修复。如果我们为T和D选择很小的值（比如它们总共为0.1s），那么故障检测器很可能怀疑非崩溃进程许多次，并且很多带宽会被“p is here”消息占据。如果我们选择一个大的总超时值（比如一星期），那么崩溃的进程会经常被报告为*Unsuspected*。 422

对此问题的一个实用解决方案是使用的超时值是反映所观察网络延迟条件的。如果本地故障检测器在20s而不是预期的10s内收到“p is here”，那么它会依据此值为p重置超时值。这个故障检测器仍然是不可靠的，它对询问的应答仍只是提示，但检测精确的概率增加了。

在同步系统中，可以使我们的故障检测器变得可靠。我们可以选择D，使得它不是一个评

估值，而是消息传输时间的绝对界限。在$T+D$秒内没收到消息“p is here”，本地故障检测器就可以得出p已经崩溃的结论。

读者可能想知道故障检测器是否实用。不可靠故障检测器可能怀疑一个无故障的进程（即它们可能是不精确的）；它们也可能不怀疑一个实际已经出故障的进程（即它们可能是不完全的）。另一方面，可靠的故障检测器要求系统是同步的（而实际系统很少是这样的）。

我们介绍故障检测器是因为它们帮助我们思考分布式系统中故障的本质，而任何被设计对付故障的实际系统必须检测故障——不管多么不完美。但是看来即使不可靠的故障检测器，只要具有某些良构特性，也能帮助我们提供有故障时进程协调问题的实际解决方案。我们在11.5节再讨论这个问题。

11.2 分布式互斥

分布式进程常常需要协调它们的动作。如果一组进程共享一个或一组资源，那么访问资源时，常需要互斥来防止干扰和保证一致性。这就是在操作系统领域中熟悉的临界区问题。然而，在分布式系统中，一般来说共享变量或者单个本地内核提供的设施都不能被用来解决这个问题。我们需要一个分布式互斥的解决方案：一个仅基于消息传送的解决方案。

在某些情况下，管理共享资源的服务器也提供互斥机制。第12章描述了服务器如何同步客户对资源的访问。但在某些实际情况下，需要一个单独的互斥机制。

考虑多个用户更新一个文本文件。保证他们更新一致的一个简单方法是，通过要求编辑器在更新之前锁住文件，一次只允许一个用户访问文件。第8章描述的NFS文件服务器在设计上是无状态的，因此不支持文件加锁。为此，UNIX系统提供由守护进程*lockd*实现的一个单
423 独的文件加锁服务，用于处理客户的加锁请求。

一个特别有趣的例子是一组对等进程在没有服务器的环境下必须协调它们对共享资源的访问。这种情况经常出现在以太网、‘自组织’模式的IEEE 802.11无线网等网络，其中网络接口作为对等成分进行协作，使得在共享介质上一次只有一个结点进行传输。再考虑一个监控一个停车场车位的系统，在每个入口和出口有一个进程来跟踪车辆进出的数目。每个进程保持一个停车场内车辆总数的计数，并且显示是否计数已满。这些进程必须一致地更新车辆数的计数。有几个方法能实现这点，比较方便的方法是这些进程只要通过相互通信就获得互斥，这样可以不需要单独的服务器。

可随意使用的分布式互斥的一般机制是有用的——这种机制独立于特定的资源管理方案。我们现在就来研究可以达到这一目的算法。

互斥算法

考虑无共享变量的N个进程p_i，i=1, 2 ,…, N的系统。这些进程只在临界区访问公共资源。为简单起见，我们假设只有一个临界区。可以很容易地把我们将要介绍的算法扩展到多个临界区。

我们假设系统是异步的，进程不出故障，并且消息传递是可靠的，这样任何传递的消息最终都被完整地发送恰好一次。

执行临界区的应用层协议如下：

```
enter()                 //进入临界区——如必要则阻塞
resourceAccesses()      //在临界区访问共享资源
```

exit() //离开临界区——其他进程现在可以进入

我们对互斥的基本要求如下：

ME1:（安全性） 在临界区（CS）一次最多有一个进程可以执行。

ME2:（活性） 进入和离开临界区的请求最终成功执行。

条件ME2隐含着既无死锁也无饥饿。死锁涉及两个或多个进程，它们由于相互依赖而在试图进入或离开临界区时被无限期地锁住。但是，即使没有死锁，一个差的算法也可能导致饥饿问题：进程的进入请求被无限推迟。

没有饥饿问题是一个公平性条件。另一个公平性问题是进程进入临界区的顺序。按进程请求时间决定进入临界区的顺序是不可能的，因为没有全局时钟。但有时使用的一个有用的公平性条件利用了请求进入临界区的消息之间的发生在先顺序（10.4节）： 424

ME3:（→顺序） 如果一个进入CS的请求发生在先，那么进入CS仍按此顺序。

如果一种解决方案用发生在先顺序来赋予临界区的进入，并且如果所有请求都按发生在先建立联系，那么在有其他进程等待时，一个进程就不可能进入临界区多于一次。这种顺序也允许进程协调它们对临界区的访问。一个多线程的进程可以在一个线程等待被准予进入临界区时，继续进行其他处理。在此期间，它可能给另一进程发消息，该进程因此也试图进入临界区。ME3指定第一个进程在第二个进程之前被准许进入临界区。

我们按下列标准评价互斥算法：

- 消耗的带宽，与在每个进入和退出操作中发送的消息数成比例；
- 每一次进入和退出操作由进程导致的客户延迟；
- 算法对系统吞吐量的影响，这是在假定后续进程间的通信是必要的条件下，进程整体访问临界区的比率。我们用一个进程离开临界区和下一个进程进入临界区之间的同步延迟来衡量着这个影响，当同步延迟较短时，吞吐量较大。

在我们的描述中，不考虑资源访问的具体实现。但是，我们假设客户进程行为正常，并且在临界区中花费有限的时间访问资源。

中央服务器算法 实现互斥的最简单的方法是使用一个服务器来授予进入临界区的许可。图11-2给出了该服务器的使用。为进入一个临界区，一个进程向服务器发送一个请求消息并等待服务器的应答。概念上，该应答构成一个表示允许进入临界区的令牌。如果在请求时没有其他进程拥有这个令牌，服务器就立刻应答来授予令牌。如果此时另一进程持有该令牌，服务器就不应答而是把请求放入队列。在离开临界区时，一个消息发往服务器，交回这个令牌。

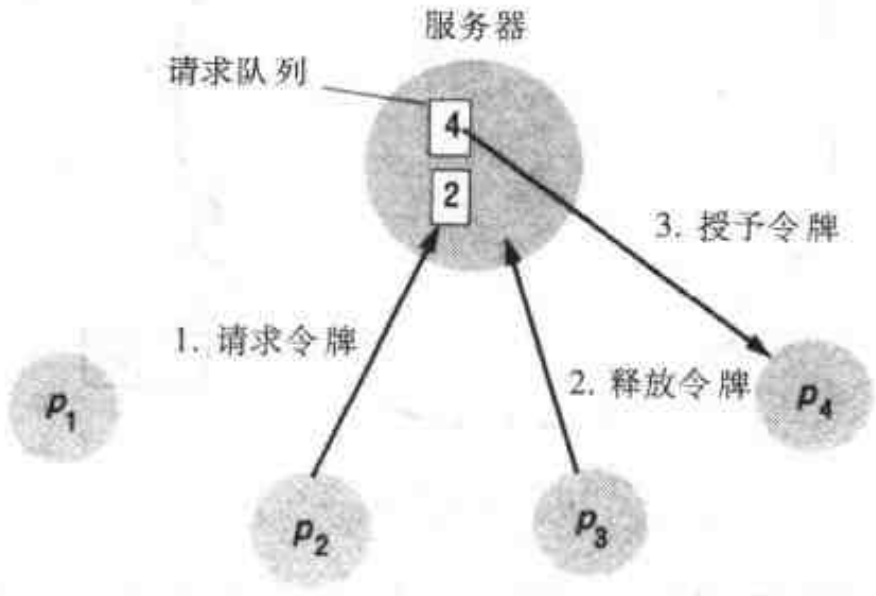


图11-2 为一组进程管理互斥令牌的服务器

如果等待进程的队列不空，服务器会选择队列中时间最早的项，把它从队列中删除并应答对应的进程。这样，这个进程持有权标。图上给出了p_2的请求被加入了已经包含p_4请求的队列的情况。p_3离开临界区，服务器删除p_4的项并通过应答p_4来授予p_4进入临界区的许可。进程p_1目前不需要进入临界区。

在没有故障的假设下，容易看到此算法满足安全性和活性条件。然而，读者可以检查此算法不满足性质ME3。

我们现在来评估此算法的性能。进入临界区——即使在当前没有进程占有它时——需要两个消息（请求和随后的授权），这样，请求进程被延迟了这一往返时间。离开临界区需要一个释放消息。假设采用异步消息传递，这不会对要离开临界区的进程造成延迟。

服务器可能会成为整个系统的一个性能瓶颈。同步延迟是下面两个消息的一次往返要花费的时间：发到服务器的释放消息，和随后让下一进程进入临界区的授权消息。

基于环的算法 在N个进程间安排互斥而不需另外进程的最简单的方法之一，是把这些进程安排在一个逻辑环中。这只需每个进程p_i与环中下一个进程$p_{(i+1) \bmod N}$有一个信道。该方法的思想是通过获得在进程间沿着环单向——如顺时针——传递的消息为形式的令牌来赋予互斥。环的拓扑结构可以与计算机之间的物理互连无关。

如果一个进程在收到令牌时不需要进入临界区，那么它立即把令牌传给它的邻居。需要权标的进程将一直等待，直到接收到令牌，它会保留令牌。为了离开临界区，进程把令牌发送到它的邻居。

进程的布置如图11-3所示。验证该算法满足条件ME1和ME2是很容易的，但令牌不必按发生在先顺序获得。（记住，进程可以独立于环的旋转交换消息。）

该算法连续地消耗网络带宽（除了当一个进程在临界区时）：进程沿着环发送消息，即使在没有进程需要进入临界区时，也是这样。请求进入临界区的进程会延迟0个（这时，它正好收到令牌）~N个（这时，它刚传递了令牌）消息。离开临界区只需要一个消息。在一个进程离开和下一个进程进入临界区之间的同步延迟可以是1~N个消息传输。

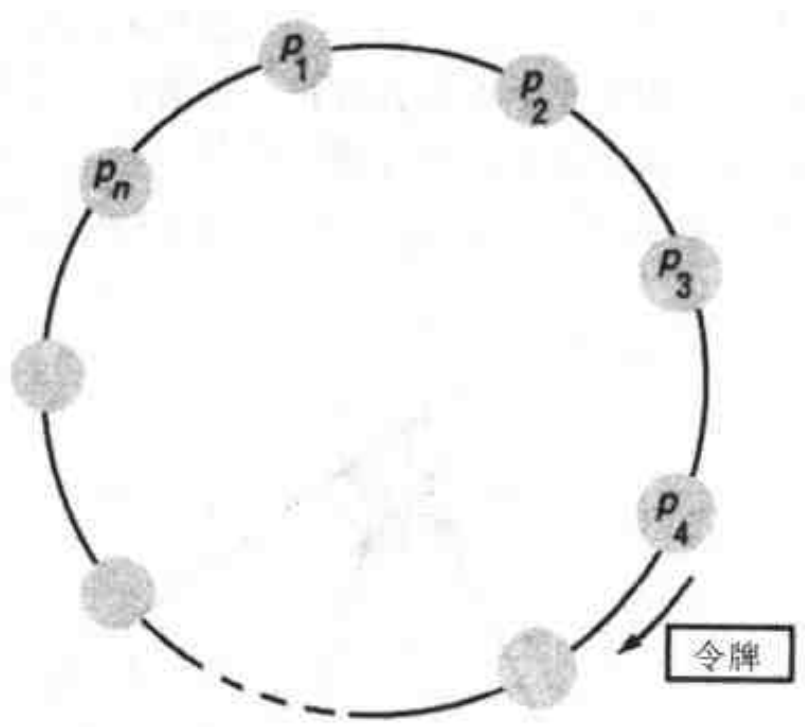


图11-3 传输互斥令牌的进程环

使用组播和逻辑时钟的算法 Ricart和Agrawala[1981]开发了一个基于组播的实现N个进程

间互斥的算法。基本思想是需要进入临界区的进程组播一个请求消息，并且只有在所有其他进程都回答了这个消息时才能进入。进程回答请求的条件被设计为满足条件ME1~ME3。

进程p_1, p_2, …, p_N具有不同的数字标识符。假设进程互相都有信道，且每个进程p_i保持一个根据10.4节规则LC1和LC2更新的Lamport时钟。请求进入的消息形如<T, p_i>，其中T是发送者的时间戳，p_i是发送者的标识。

每个进程在变量*state*中记录它的状态，包括在临界区外（RELEASED），希望进入（WANTED），或在临界区中（HELD）。图11-4给出了协议。

```
初始化：
        state := RELEASED;
为了进入临界区：
        state := WANTED;                          ┐
        组播请求给所有进程；                       ├ 请求处理在此被延迟
        T :=请求的时间戳;                          ┘
        Wait until (接收到的应答数= (N – 1));
        state := HELD;
在p_j (i ≠ j)接收一个请求<T_i, p_i>:

        if (state = HELD or (state = WANTED and (T, p_j) < (T_i, p_i)))
        then
                将请求放入p_j队列，不给应答；
        else
                马上给p_j应答；
        end if
为了退出临界区：
        state := RELEASED;
        对已入队列的请求给出应答；
```

图11-4 Ricart和Agrawala算法

如果一个进程请求进入，而所有其他进程的状态都是RELEASED，那么，所有进程会立即应答请求，请求者会得以进入。如果某进程在状态HELD，那么该进程在结束对临界区的访问前不会应答请求，因此在这期间请求者不能得以进入。如果两个或多个进程同时请求进入，则发最早时间戳的请求的进程将是第一个收集N-1个应答的进程，它将被准许下一个进入。如果请求具有相等的Lamport时间戳，那么请求将根据进程的标识排序。注意，当一个进程请求进入时，它推迟处理来自其他进程的请求，直到发送了它自己的请求，并且记录了该请求的时间戳T。这样做的目的是为了进程在处理请求时做出一致的判定。

该算法达到安全性特性ME1。如果两个进程p_i和p_j（$i \neq j$）能同时进入临界区，那么这两个进程必须已经互相应答了对方。但是，因为<T_i, p_i>对是全排序的，所以这是不可能的。验证算法满足需求ME2和ME3留给读者思考。

为了说明上述算法，考虑图11-5所示的3个进程p_1, p_2和p_3的情况。假设p_3对进入临界区不感兴趣，而p_1和p_2并发地请求进入。p_1的请求的时间戳是41，p_2的是34。当p_3接到它们的请求时，立即回答。当p_2接到p_1的请求时，它发现自己的请求有更早的时间戳，因此不予应答，延缓p_1。然而，p_1发现p_2的请求比自己的请求有更早的时间戳，因此立即应答。p_2一收到第二个

应答，便能进入临界区。当p_2离开临界区时，它将应答p_1的请求，因此授予p_1进入。

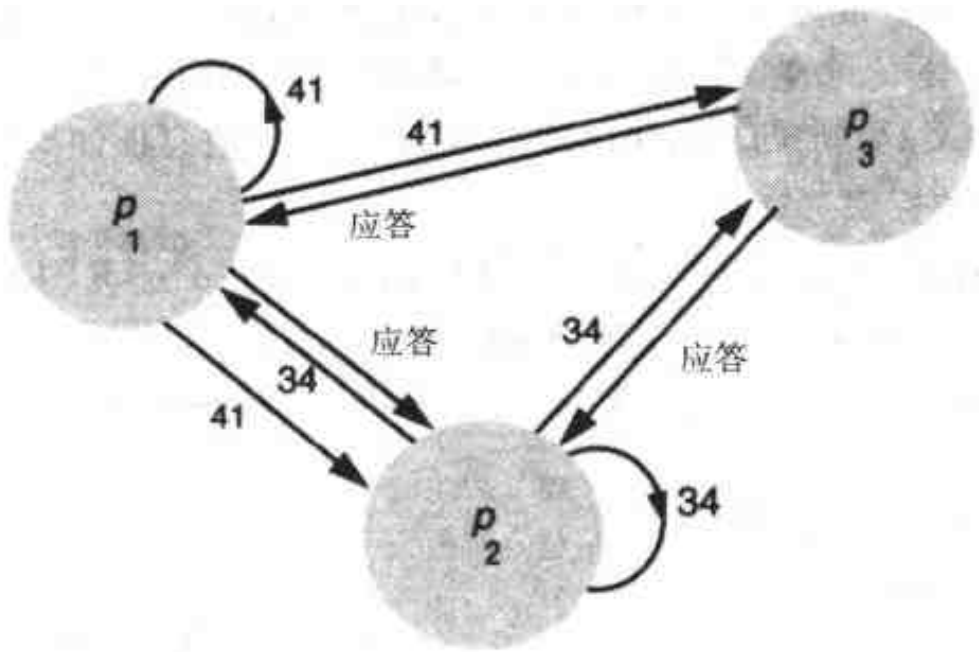


图11-5 组播同步

在该算法中，获得进入需要$2(N-1)$个消息：$N-1$个消息用于组播请求，随后是$N-1$个应答。或者，如果硬件支持组播，请求只需要一个消息，总数是N个消息。因此，在带宽消耗方面，该算法比前述算法更昂贵。然而，请求进入的客户延迟仍是一个往返时间（忽略组播请求消息带来的延迟）。

该算法的优点是它的同步延迟仅是一个消息传输时间。前两种算法都有一个往返的同步延迟。

该算法的性能可以改进。首先，我们注意到最近一次进入过临界区、且没有接到其他的进入请求的进程，仍需如描述的那样遍历协议，即使它可以简单地在本地把令牌重新分配给自己。其次，Ricart和Agrawala改进了协议，使它在没有硬件支持组播时，在最坏（也是通常的）情况下需要N个消息来获得进入。对此的描述是Raynal [1988]。

Maekawa投票算法 Maekawa [1985]观察到为了让一个进程进入临界区，不必要求所有对等进程都同意它的访问。进程只需要从其对等进程的子集获得进入许可，只要任两进程使用的子集有重叠。我们可以想象进程是互相选举，进入临界区。一个“候选”进程为进入必须收集到足够的选票。在两个投票集合的交集中的进程，通过把选票只投给一个候选者，保证了安全性ME1，即最多只有一个进程可以进入临界区。

Maekawa把每个进程p_i（$i=1, 2, \cdots, N$）关联到一个选举集V_i，其中$V_i \subseteq \{p_1, p_2, \cdots, p_N\}$。集合$V_i$的选择使得对所有$i, j = 1, 2, \cdots, N$，有：

- $p_i \in V_i$
- $V_i \cap V_j \neq \phi$ ——任两个选举集至少有一个公共成员
- $|V_i| = K$ —— 为公平起见，每个进程有同样大小的选举集
- 每个进程p_j被包括在选举集V_i中的M个集合中。

Maekawa指明了，最优解，即，使K最小而允许进程达到互斥的情况，具有$K \sim \sqrt{N}$且$M = K$（因此每个进程所在的选举集数与这些集合中的每一个的元素数相同）。计算最优集合R_i并不简单。作为一种近似，得到使$|R_i| \sim 2\sqrt{N}$的集合R_i的一个简单方法是把进程放在一个$\sqrt{N} \times \sqrt{N}$矩阵中，并令V_i是包含p_i的行和列的并集。

Maekawa算法如图11-6所示。为获得进入临界区，进程p_i发送请求消息到V_i的所有其他$K-1$

个成员。在收到所有K－1个应答消息前，p_i不能进入临界区。当V_i中的进程p_j收到p_i的请求消息时，它立即发送一个应答消息，除非它的状态是HELD，或者它自从它上次收到一个释放消息以来已经给了应答（“已投票”）。否则，它把请求消息加入队列（按到达时间顺序），但现在不应答。当一个进程收到一个释放消息时，它从请求队列中删除队头（如果队列不空），并发送一个应答消息（一个“投票”）响应它。为了离开临界区，p_i发送释放消息给V_i中的所有其他K－1个成员。 425~429

```
初始化：
    state := RELEASED;
    voted := FALSE;
p_i为了进入临界区：
    state := WANTED;
    将请求组播给V_i – {p_i}中的所有进程；
    Wait until (接收到的应答数 = (K – 1));
    state := HELD;
在p_j (i ≠ j)接收来自p_i的请求：
    if (state = HELD or voted = TRUE)
    then
            将来自p_i的请求放入队列，不给应答；
    else
            将应答发给p_i;
            voted := TRUE;
    end if
p_i为了退出临界区：
    state := RELEASED;
    将释放组播给V_i - {p_i}中的所有进程；
在p_j (i ≠ j)接收到来自p_i的释放：
    if (请求队列非空)
    then
            删除队列头——例如，p_k；
            将应答发给p_k;
            voted := TRUE;
    else
            voted := FALSE;
    end if
```

图11-6 Maekawa算法

该算法达到安全性ME1。如果两个进程p_i和p_j能同时进入临界区，那么在$V_i \cap V_j \neq \phi$中的进程必须已经对它们两个投票。但该算法允许一个进程在连续收到的释放消息之间最多投一个选票——所以上述情况是不可能的。

但是，该算法易于死锁。考虑3个进程p_1, p_2, p_3和$V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$, $V_3 = \{p_3, p_1\}$。如果3个进程并发地请求进入临界区，那么可能p_1应答p_2但延缓p_3，p_2应答p_3但延缓p_1，p_3应答p_1但延缓p_2。每个进程收到两个应答中的一个，故都不能继续。 430

可以修改算法 [Saunders 1987] 使其成为无死锁的。在修改的协议中，进程按发生在先顺

序对有待应答的请求排队，因此也满足需求ME3。

该算法的带宽使用是每次进入临界区需$2\sqrt{N}$个消息和每次退出需$\sqrt{N}$个消息（假设没有硬件组播故障）。如果$N>4$，$3\sqrt{N}$要优于Ricart和Agrawala算法的$2(N-1)$。客户延迟与Ricart和Agrawala算法一样，但同步延迟更差一些：是一个往返时间，而不是单个消息的传输时间。

容错 在容错方面，评估以上算法的要点是：

- 当消息丢失时会发生什么？
- 当进程崩溃时会发生什么？

如果通道不可靠，我们已介绍的算法都不能容忍消息丢失。基于环的算法不能容忍任何单个进程的崩溃故障。Maekawa算法可以容忍一些进程的崩溃故障：如果一个崩溃进程不在所需的一个投票集中，那么它的故障不会影响其他进程。中央服务器算法可以容忍一个既不持有也不请求令牌的客户进程的崩溃故障。修改我们已描述的Ricart和Agrawala算法，即，使进程隐式地授权所有请求，可以容忍这样的进程的崩溃故障。

我们请读者考虑，假设存在可靠的故障检测器，如何修改算法来容错。即使有一个可靠的故障检测器，也需要注意允许在任何点出故障（包括在恢复过程期间）和在故障被检测到以后重构进程的状态。例如，在中央服务器算法中，如果服务器发生故障，那么它必须被建立起来，无论它持有令牌或客户进程中的一个持有令牌。

在11.5节我们研究在有故障时进程如何协调它们的动作这一一般性的问题。

11.3 选举

选择一个惟一的进程来扮演特定角色的算法称为选举算法。例如，在我们的互斥“中央服务器”算法的一个变种中，“服务器”是从需要使用临界区的进程$p_i, i = 1, 2, ..., N$中选择。需要一个选举算法来选择哪一个进程将扮演服务器的角色。一个基本要求是所有进程都同意
431 这个选择。然后，如果担任服务器角色的进程想退休，那么需要另一次选举来选择替代者。

我们称一个进程*召集选举*，如果该进程采取行动启动了选举算法的一次运行。一个进程每次最多召集一次选举，但原则上N个进程可以并发召集N个选举。在任何时间点，进程p_i或者是一个*参加者*——意指它参加选举算法的某次运行，或者是*非参加者*——意指它当前没有参加任何选举。

一个重要要求是对当选进程的选择必须惟一，即使若干个进程并发地召集选举。例如，两个进程可以独立判定一个协调进程已经失败，并且都召集选举。

不失一般性，我们要求选择具有最大标识的进程为当选进程。“标识”可以是任何有用的值，只要标识惟一且可按全序排序。例如，通过用$\langle 1/load, i\rangle$作为进程的标识（其中$load>0$且进程索引i用于对负载相同的标识排序），我们可以选举具有最低计算负载的进程。

每个进程p_i（$i=1, 2, \cdots, N$）有一个变量$elected_i$，用于包含当选进程的标识。当进程第一次成为一次选举的参加者时，它把变量值置为特殊值‘$\perp$’、表示该值还没有定义。

我们的要求是，在算法的任何一次运行期间：

E1:（安全性） 参与的进程p_i有$elected_i = \perp$，或$elected_i = P$，其中P是在运行结束时具有最大标识的非崩溃进程。

E2:（活性） 所有进程p_i都参加并且最终置$elected_i \neq \perp$，或者崩溃。

注意，可能有还不是参加者的进程p_j，它在$elected_j$中记录着上次当选进程的标识。

我们通过总的网络带宽使用（与发送消息的总数成比例）和算法的回转时间（从启动算法到终止算法之间的串行消息传输的次数）来衡量一个选举算法的性能。

基于环的选举算法 我们给出Chang和Roberts [1979] 的算法，它适用于按逻辑环排列的一组进程。每个进程p_i有一个到下一进程$p_{(i+1) \bmod N}$的信道，所有消息沿着环顺时针发送。我们假设没有故障发生，并且系统是异步的。该算法的目标是选举一个叫做协调者的进程，它是具有最大标识的进程。

最初，每个进程被标记为选举中的一个非参加者。任何进程可以开始一次选举。它把自己标记为一个参加者，然后，把自己的标识放到一个选举消息里，并把消息发送到它的顺时针邻居。

当一个进程收到一个选举消息时，它比较消息里的标识和它自己的标识。如果到达的标识较大，它把消息转发到它的邻居。如果到达的标识较小，且接收进程不是一个参加者，它把消息里的标识替换为自己的，并转发消息；如果它已经是一个参加者，它就不转发消息。任何情况下，当转发一个选举消息时，进程把自己标记为一个参加者。 432

然而，如果收到的标识是接收者自己的，这个进程的标识一定最大，该进程就成为协调者。协调者再次把自己标记为非参加者并向它的邻居发送一个当选消息，宣布它的当选并将它的身份放入消息中。

当进程p_i收到一个当选消息时，它把自己标记为非参加者，置变量$elected_i$为消息里的标识，并且把消息转发到它的邻居，除非它是新的协调者。

容易看到该算法满足条件E1。所有标识都被比较了，因为一个进程在发送当选消息前必须将自己的标识接收回来。对任意两个进程，标识较大的进程不会传递另一进程的标识。因此不可能两者都收到它们自己的标识。

从保证环的遍历（没有故障）立即得到条件E2。注意非参加者和参加者状态的使用方式，这种使用方式使在另一进程同时开始进行的一次选举所引发的消息被尽可能地压制，并且总在“获胜的”选举结果宣布之前进行。

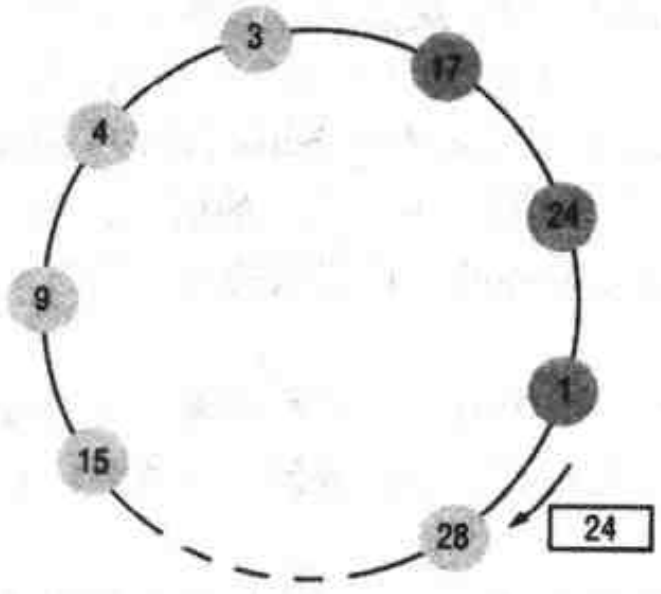
如果只有一个进程启动一次选举，最坏的执行情况是它的逆时针邻居具有最大的标识。这时到达该邻居需要$N-1$个消息，并且还需要N个消息再完成一个回路，才能宣布它的当选。接着当选消息被发送N次，共计$3N-1$个消息。回转时间也是$3N-1$，因为这些消息都是顺序发送的。

进行中的一次基于环的选举的例子如图11-7所示。选举消息当前包含24，但进程28会在消息到达时，把它替换为自己的标识。

虽然基于环的算法有助于理解一般选举算法的性质，但是它不容错的事实使得它只有有限的实用价值。然而，通过利用可靠的故障检测器，当一个进程崩溃时重构环原则上是可能的。

霸道算法 霸道算法 [Garcia-Molina 1982] 虽然假定进程间消息发送是可靠的，但它允许在选举期间进程崩溃。与基于环的算法不同，该算法假定系统是同步的；它使用超时来检测进程故障。另一个不同是，基于环的算法假定进程相互之间具有最小的先验知识：每个进程只知道如何与邻居通信，且没有进程知道其他进程的标识；而霸道算法假定每个进程知道哪些进程有较高的标识，并且可以和所有这些进程通信。

在该算法中有3种类型的消息。选举消息用于宣布选举；回答消息用于回复选举消息；协调者消息用于宣布当选进程的身份——新的“协调者”。一个进程通过超时发现协调者已经出现故障时开始一次选举。几个进程可能并发地观察到此现象。



注：选举从进程17开始。到目前为止，所遇到的最大的进程标识是24。参与的进程用深色显示

图11-7 进行中的一次基于环的选举

因为系统是同步的，我们可以构造一个可靠的故障检测器。有一个最大消息传输延迟T_{trans}和一个最大消息处理延迟$T_{process}$。因此，我们可以计算时间$T = 2T_{trans} + T_{process}$，是从发送一个消息给另一进程到收到回复的总时间的上界。如果在T时间内没有收到回答，本地故障检测器可以报告说请求的预期接收者已经出现故障。

知道自己有最大标识的进程可以简单地通过发送协调者消息给所有有较低标识的进程，来选举自己为协调者。另一方面，有较低标识的进程开始一次选举，要通过发送选举消息给那些有较大标识的进程，并等待回复的回答消息。如果在时间T内没有消息到达，该进程认为自己是协调者，并发送协调者消息给所有有较低标识的进程来宣布这一结果。否则，该进程再等待时间T'用于接收从新的协调者发来的消息。如果没有消息到达，它开始另一次选举。

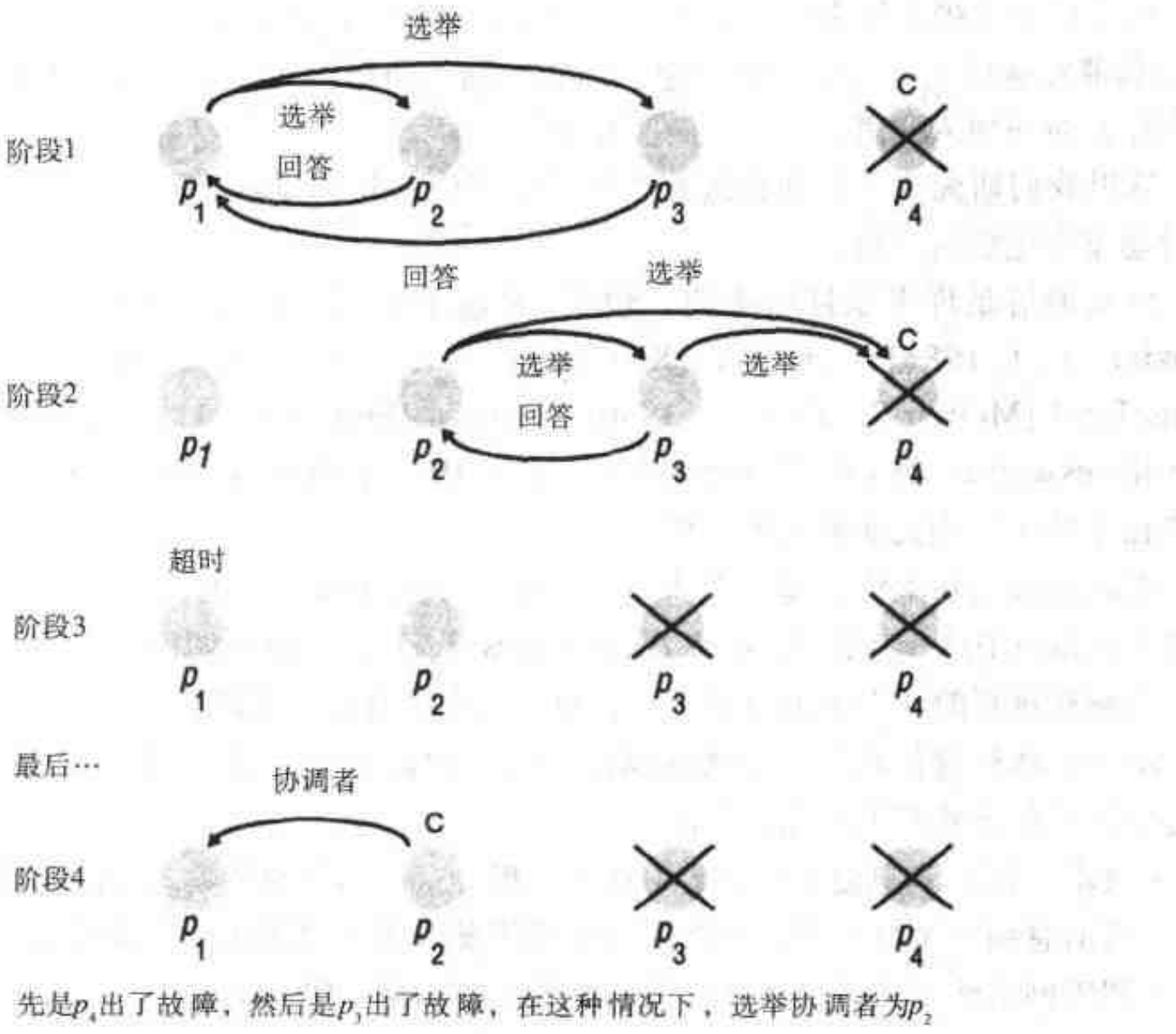
如果进程p_i收到一个协调者消息，它把它的变量$elected_i$置为消息中包含的协调者的标识，并把这个进程作为协调者。

如果一个进程收到一个选举消息，它回送一个回答消息并开始另一次选举——除非它已经开始了一次选举。

当一个进程被启动来替换一个崩溃进程时，它开始一次选举。如果它有最大的进程标识，它会决定自己是协调者，并向其他进程宣布。因此即使当前协调者正在起作用，它也会成为协调者。正是因为这个原因，该算法被称为“霸道”算法。

算法的运转如图11-8所示。有4个进程p_1~p_4。进程p_1检测到协调者p_4的故障，并宣布进行选举（图中阶段1）。当收到p_1发来的选举消息时，进程p_2和p_3发回答消息给p_1，并开始它们自己的选举；p_3发一个回答消息给p_2，但p_3没有从出现故障的进程p_4收到回答消息（阶段2）。因此它决定自己是协调者。但在它发出协调者消息之前，它也出现故障（阶段3）。当p_1的超时周期T'过去后（我们假设这发生在p_2的超时周期过去之前），它得出没有协调者消息的结论并开始另一选举。最终，p_2被选为协调者（阶段4）。

依据可靠消息传输的假定，该算法显然满足活性条件E2。而且如果没有进程被替换，算法满足条件E1。两个进程不可能都决定它们是协调者，因为有较低标识的进程会发现另一进程的存在并服从于它。



先是p_4出了故障，然后是p_3出了故障，在这种情况下，选举协调者为p_2

图11-8 霸道算法

但是如果崩溃的进程被替换为具有相同标识的进程，该算法不能保证满足安全性条件E1。正在另一进程（它已经检测到进程p的崩溃）已经决定它有最大的标识时，替换p的进程可能决定它有最大的标识。两个进程可能并发地宣布它们自己为协调者。但是，在消息的传输顺序上没有保证，这些消息的接收者可能得出谁是协调者的不同的结论。

此外，如果假定的超时值证明是不准确的——即如果进程的故障检测器是不可靠的，那么可能违反条件E1。

433 ~ 435

考虑上面给出的例子，假设p_3或者没有崩溃但运行异乎寻常地慢（即系统同步的假定是不正确的）、或者p_3已经崩溃但被替换。正在p_2发送它的协调者消息时，p_3（或替换者）也做同样的事情。p_2在发送自己的协调者消息后收到p_3的消息，因此置$elected_2 = p_3$。由于消息传输延迟，p_1在收到p_3的协调者消息后收到p_2的消息，因此最终$elected_1 = p_2$。条件E1被违反。

关于算法的性能，最好情况是具有次大标识的进程发现了协调者的故障。于是它可以立即选举自己并发送$N-2$个协调者消息。回转时间是一个消息。在最坏情况下霸道算法需要$O(N^2)$个消息——即具有最小标识的进程首先检测到协调者的故障。然后$N-1$个进程一起开始选举，每个进程都发送消息到有较大标识的进程。

11.4 组播通信

4.5.1节描述了IP组播，它是组通信的一个实现。组或组播通信需要协调和协定。目的是使一组进程中的每一个都收到发到组中的、往往带有发送保证的消息的副本。此保证包括对组中每个进程应当收到的消息集合以及在组成员间的发送顺序达成一致。

组通信系统极其复杂。即使是提供最小发送保证的IP组播，也需要很大的工程上的努力。时间和带宽利用效率是主要的考虑，即使对静态的进程组，这也是具有挑战性的。当进程可以在任意时间加入或离开组时，问题会成倍增加。

这里我们研究成员已知的进程组的组播通信。第14章将把研究扩展到成熟的组通信，包括对动态变化组的管理。

组播通信是许多项目的主题，包括V系统 [Cheriton and Zwaenepoel 1985]，Chorus [Rozier *et al.* 1988]，Amoeba [Kaashoek *et al.* 1989, Kaashoek and Tanenbaum 1991]，Trans/Total [Melliar-Smith *et al.* 1990]，Delta-4 [Powell 1991]，Isis [Birman 1993]，Horus [van Renesse *et al.* 1996]，Totem [Moser *et al.* 1996]和Transis [Dolev and Malki 1996]——我们还将在本节中引用其他著名的产品。

组播通信的基本特点是一个进程只调用一个组播操作来发送一个消息到进程组中的每个进程（在Java中这个操作是*aSocket.send(aMessage)*），而不是调用多个针对单个进程的发送操作。与所有进程的一个子组通信相对，与系统中所有进程通信称为广播。

436 用一个组播操作代替多个发送操作所得到的远不止是程序员的方便。它使实现有效，并且提供比其他方式都强的发送保证。

- 效率　同一消息要发送到组中所有进程，这一信息允许实现可以有效使用带宽。通过发送消息到一个分布树，可以采取步骤使发送消息到任何通信连接不超过一次；而且在能利用网络硬件支持的地方可以用硬件支持组播。实现不采用独立、串行地传输消息，故还能使发送消息到所有目的地的总时间最少。

  为了了解这些优点，比较如下情况的带宽使用和总传输时间，即从伦敦的一台计算机，(a) 通过两个独立的UDP发送；(b) 通过一个IP组播操作，发送同一消息到Palo Alto的在同一以太网上的两台计算机。在前一种情况下，消息的两个副本被独立发送，且第二个还被第一个延迟。在后一种情况下，消息不是发送两次，一组允许组播的路由器把消息的一个副本从伦敦转发到目的地LAN的一个路由器上。然后，这个路由器利用硬件组播（由以太网提供）将消息传递到目的地。

- 传递保证　如果一个进程发多个独立的发送操作到独立的进程，那么实现就无法提供能影响整个进程组的发送保证。如果发送者在发送中途出现故障，组中的一些成员可能收到消息，而其他成员则没收到。而且发送到组中任两成员的两个消息的相对顺序是没有定义的。在IP组播的情况下，实际上，顺序或可靠性保证都没有提供。但是可以做更强的组播保证，我们随后会定义一些。

系统模型　系统包含一组进程，它们可以通过一对一的通道可靠地进行通信。如前所述，进程在崩溃时才出现故障。

进程是组的成员，它们是使用组播操作发送的消息的目的地。通常，允许进程同时是几个组的成员是有用的——例如，进程通过加入几个组，能接收几个来源的信息。但是为了使顺序性质的讨论简单化，我们有时限制进程一次最多是一个组的成员。

操作 *multicast*(*g*,*m*) 发送消息*m*到进程组*g*的所有成员。相应地，有一个操作 *deliver*(*m*) 传递由组播发送的消息到调用进程。我们使用术语 *deliver* 而不是 *receive*，以阐明组播消息并不总是一被进程结点收到就被提交到进程内部的应用层。在随后讨论组播传递语义时对此会给出解释。

每个消息*m*携带发送它的进程 *sender*(*m*) 的惟一标识和惟一目的地组标识 *group*(*m*)。我们假定进程不会谎报消息的源和目的地。

一个组称为是封闭的，如果只有组的成员可以组播到它（图11-9）。封闭组中的一个进程将任何它组播到组的消息传递给它自己。一个组是开放的，如果组外的进程可以发送消息给它（“开放的”和“封闭的”分类也应用于邮件列表，具有相似的含义）。协作的服务器相互发送只有它们自己才应当接收的消息，这样的例子适用于封闭组。传递事件到感兴趣的进程的组，这样的例子适用于开放组。 437

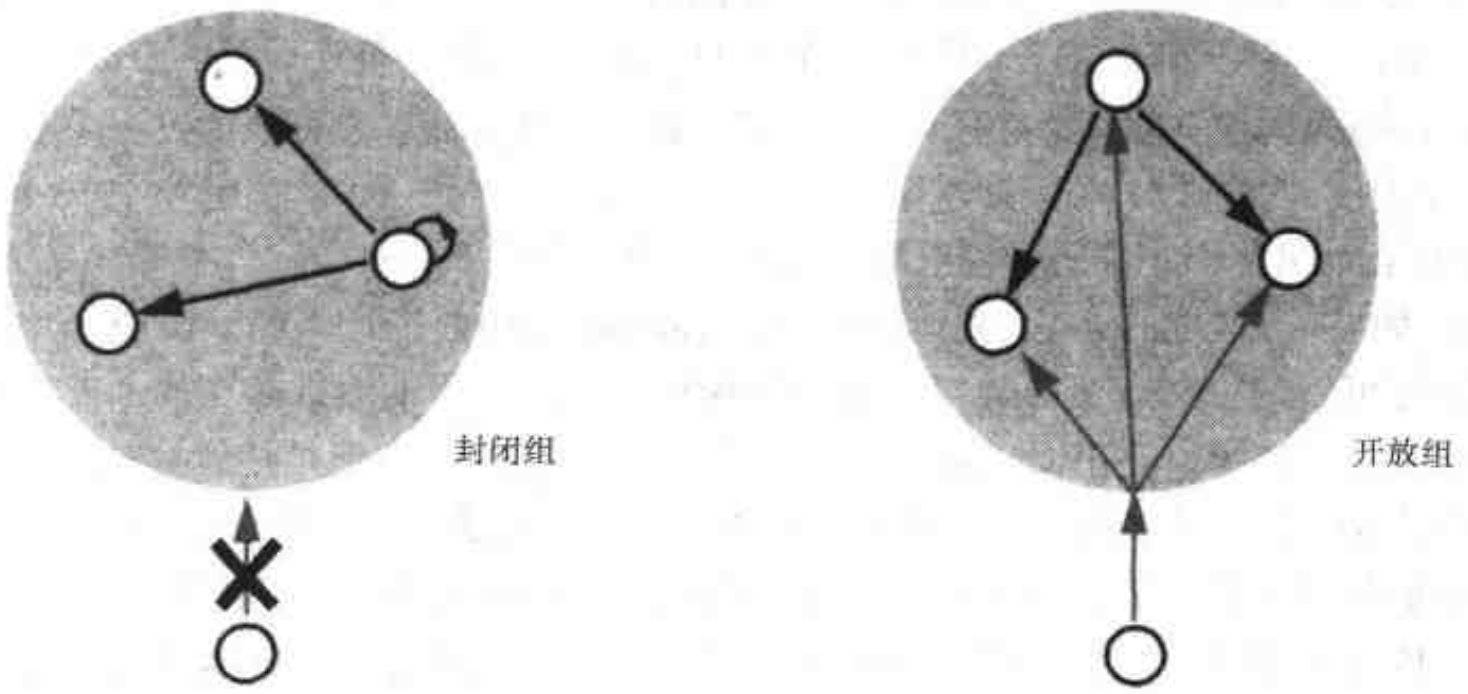


图11-9 开放组和封闭组

一些算法假定组是封闭的。通过挑选组中一成员并发送消息给它（一对一），由它组播到组里，可以在封闭组中达到开放组的相同效果。Rodrigues 等人[1998] 讨论了开放组的组播问题。

11.4.1 基本组播

可自由使用的基本组播原语是有用的，与IP组播不同，该原语保证，只要组播进程不崩溃，一个正确的进程最终会传递消息。我们把这个原语称为*B-multicast*，而与它对应的基本传递原语是*B-deliver*。我们允许进程属于几个组，而每个消息发往某个特定组。

实现*B-multicast*的一个简单的方法是使用一个可靠的一对一*send*操作，如下：

- *B-multicast*(*g*, *m*)：对每个进程$p \in g$, *send*(*p*,*m*);
- 进程*p* *receive*(*m*)时：*p*执行*B-deliver*(*m*)。

为了减少传递消息的总时间，实现上可以利用线程来并发执行*send*操作。但是，如果进程数很大，这样的实现很可能经受一种叫做确认爆炸的情况。作为可靠*send*操作的一部分发送的确认很可能从许多进程同时到达。进行组播的进程的缓冲区会很快充满，因此很可能丢掉确认消息。于是进程会重新发送消息，导致更多的确认和浪费更多的网络带宽。更为实用的基本组播服务可以使用IP组播建立，我们请读者完成这一服务。 438

11.4.2 可靠组播

2.3.2节定义了一对进程之间可靠的一对一信道。所要求的安全性被称为完整性——即任何传递的消息与发送的消息相同，且没有消息被传递两次。所要求的活性被称为有效性：即

任何消息最终会被传递到目的地，如果它是正确的。

按照 Hadzilacos 和Toueg [1994]、Chandra和Toueg [1996]的成果，我们现在定义可靠组播以及相应的操作*R-multicast*和*R-deliver*。 在可靠组播中，显然非常需要类似完整性和有效性的性质。但我们还增加另外一个性质：要求如果组中任何一个进程收到一个消息，那么组中所有正确的进程都必须收到这个消息。这不是基于可靠的一对一*send*操作的*B-multicast*算法的性质，认识到这一点是重要的。在*B-multicast*进行时，发送进程可能在任何点出故障，因此一些进程可能传递消息而另一些则不传递。

一个可靠组播是满足以下性质的组播，我们在叙述这些性质后再对这些性质进行解释。

- 完整性： 一个正确的进程p传递一个消息m至多一次。而且，$p \in group(m)$且m由$sender(m)$提供给一个组播操作。（与一对一通信一样，消息总可以通过一个与发送者相关的顺序号来区别。）
- 有效性：如果一个正确的进程组播消息m，那么它终将传递m。
- 协定：如果一个正确的进程传递消息m，那么在$group(m)$中的其他所有正确的进程终将传递m。

完整性与可靠的一对一通信类似。有效性保证了发送进程的活性。这看上去可能是一个与众不同的性质，因为它是不对称的（它只提到一个特定进程）。但是注意有效性和协定一起得到一个全面的活性要求：如果一个进程（发送者）最终传递了一个消息m，那么，因为正确的进程在它们传递的消息上是一致的，可知m终将被传递到组中所有正确的成员。

按照自传递来表达有效性条件的优点是简单性。我们需要的是消息最终被组中的某个正确的成员传递。

协定条件与原子性相关（即“都有或都没有”的性质），是将原子性应用于对组的消息传递。如果一个组播消息的进程在传递消息之前就崩溃了，则可能这个消息将不被传递到组中的任何进程；但是如果消息被传递到某个正确的进程，则其他所有正确的进程会传递它。文
439 献中的许多文章用术语“原子的”来包括一个全排序条件；我们稍后给出定义。

用B-multicast实现可靠组播算法 图11-10给出了一个使用原语*R-multicast*和*R-deliver*的可靠组播算法，它允许进程同时属于几个封闭的组。为了*R-multicast*一个消息，一个进程*B-multicast*消息到目的组中的进程（包括它自己）。当消息被*B-deliver*时，接收者依次*B-multicast*消息到组中（如果它不是最初的发送进程），然后*R-deliver*消息。因为消息到达任何结点可能多于一次，消息的副本被检测且不被传递。

```
初始化：
    Receive:={};
进程p将R-multicast消息m发给组g：
    B-multicast(g, m);        //p ∈ g作为目的地被包括在内
在进程q B-deliver(m)时，其中g=group(m)
if(m ∉ Received)
then
        Received:=Received ∪ {m};
        if(q ≠p)then B-multicas{g, m};end if
        R-deliver m;
        end if
```

图11-10 可靠组播算法

这个算法显然满足有效性，因为一个正确的消息终将*B-deliver*消息到它自己。根据用在*B-multicast*中的信道的完整性，算法也满足完整性。

每一个正确的进程在*B-deliver*消息后都*B-deliver*该消息到其他进程这一事实可以说明该算法遵循协定。如果一个正确的进程没有*R-deliver*某一消息，这只能是因为它从来没有*B-deliver*此消息，更进一步，这又只能是因为也没有其他正确的进程*B-deliver*此消息，因此，没有进程会*R-deliver*此消息。

我们描述的这个可靠组播算法在异步系统中是正确的，因为我们没做时间假设。但是该算法从实用角度来说是低效的。每个消息被发送每个进程|g|次。

用IP组播实现可靠组播 *R-multicast*的另一种实现是组合使用IP组播、捎带确认法（即确认附加在其他消息上）和否定确认，这个*R-multicast*协议是基于下述观察，即IP组播通信常常是成功的。在该协议中，进程不发送单独的确认消息；作为替代，它们在发送到组中的消息上捎带确认。只有当进程检测到它们漏过一个消息时，它们才发送一个单独的应答消息。指出一个预期的消息没有到达的应答被叫作否定确认。

该协议假定组是封闭的。每个进程p对于它属于的组g维持一个顺序数S_g^p。顺序数最初为零。每个进程还记录R_g^q，即它最近传递的来自进程q并且发送到组g的消息的顺序数。 440

p要*R-multicast*一个消息到组g时，它在消息上捎带值S_g^p。它还在消息上捎带确认，形如$<q, R_g^q>$。这个确认给出了一个顺序数，即自从该进程上一次组播消息后，它最近传递的来自进程q并且发往该组的消息的顺序数。然后，组播进程p把消息连同它捎带的顺序数和确认一起IP组播到g，并且把S_g^p加一。

当且仅当$S = R_g^p + 1$，一个进程*R-deliver*一个来自p并发往g、具有顺序数S的消息，并且它在传递后立即把R_g^p加一。它把任何它还没能传递的消息保留在一个保留队列中（图11-11）——这样的队列常需要满足消息传递保证。另一方面，如果一个到达的消息有$S \leqslant R_g^p$，那么r已经传递了它，所以丢弃该消息。如果$S > R_g^p + 1$，或对任意附上的确认$<q, R>$有$R > R_g^q$，那么r已经漏了一个或多个消息。它通过发送否定确认来请求这些消息。它可能发送请求到那个使得它了解到消息遗漏的进程或到最初的发送进程，如果两者是不同的。在该协议的一些变种中，它把请求组播到它的网络邻居，如果它们已经收到了请求。

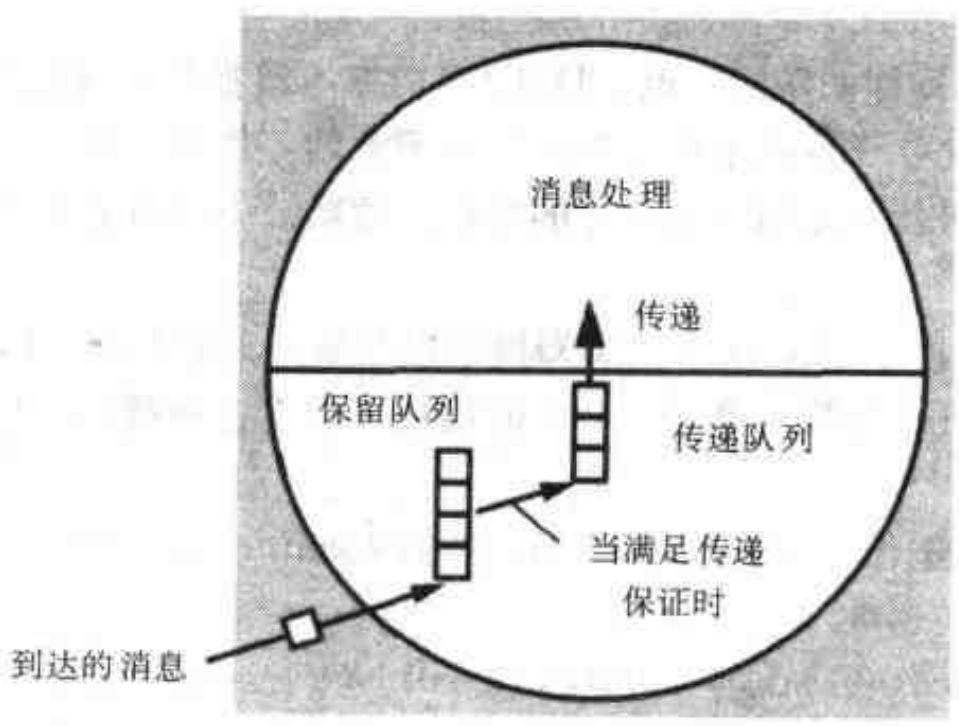


图11-11 用于到达的组播消息的保留队列

从对副本和IP组播性质的检测（使用校验和来除去损坏的消息），可以得到完整性。有效性仅当进程总可以检测漏掉的消息时成立，这又意味着进程总会收到又一个消息，使它能够检测到这个遗漏。因此，只在假定每个进程都无限组播消息的情况下，这个协定具有有效性。对任何消息，只要保证一个没有收到该消息而又需要它的进程能够得到它的一个副本，就能
441 保证协定成立，因此我们假定进程无限地保留它们已传递消息的副本。

我们为保证有效性和协定所做的假设都是不实用的。但是在我们的协定所源自的协议中，对有效性和协定进行了实用的研究：Psync协议 [Peterson *et al.* 1989]、Trans 协议 [Melliar-Smith *et al.* 1990] 和可伸缩的可靠组播协议 [Floyd *et al.* 1997]。Psync和Trans协议还提供传递顺序保证。

统一的性质 上面给出的协定定义只提到正确进程的行为——即永无故障的进程。请考虑图11-10的算法，如果一个进程不是正确的，并且在*R-deliver*一个消息后崩溃，会发生什么。由于任何*R-deliver*消息的进程必先*B-multicast*它，可知所有正确的进程最终仍会传递此消息。

无论进程是否正确都成立的性质称为统一的性质。我们定义统一的协定如下：

- 统一的协定 如果一个进程传递消息m，不论该进程是正确的或出故障，在$group(m)$中的所有正确的进程终将传递m。

统一的协定允许一个进程在传递了一个消息后崩溃，同时仍然保证所有正确的进程将传递此消息。我们已经论证了图11-10的算法满足这一性质，该性质比前面定义的非统一的协定更强。

对于一些应用，其中进程在崩溃前可以采取行动产生一个可观察的不一致，在这种应用中，统一的协定是有用的。例如，考虑进程是管理银行账户副本的服务器，且账户的更新是使用可靠组播发送到服务器组的情况。如果组播不满足统一的协定，那么恰在一个服务器崩溃前，访问该服务器的客户可以观察到一个其他服务器都不会处理的更新。

注意，有趣的是，在图11-10中，如果颠倒“*R-deliver m*”和“*if* $(q \neq p)$ *then B-multicast*(g, m); *end if*”这两行的顺序，那么算法不满足统一的协定。

正如协定有一个统一的版本，任何组播性质都有统一的版本，包括有效性、完整性和我们将要定义的排序性质。

11.4.3 有序组播

由于底层的一对一发送操作的随机延迟，11.4.1节的基本组播算法按任意顺序传递消息到进程。这种顺序保证的缺少对许多应用都是不能令人满意的。例如，在一个核电站里，表示对安全条件的威胁的事件和表示控制单元动作的事件，被系统中的所有进程以同样的顺序观察到可能是重要的。

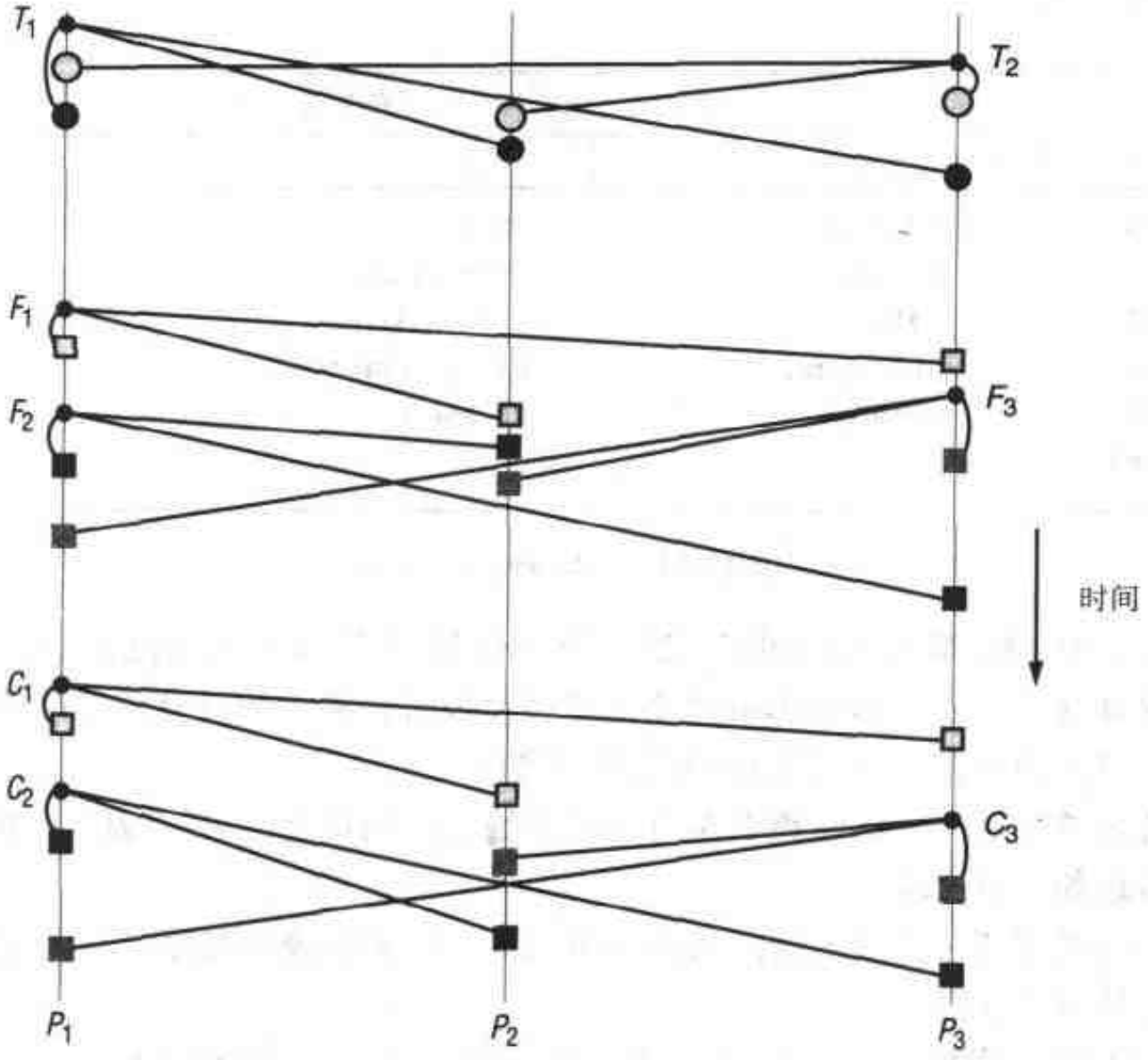
通常的排序需求有全排序、因果排序、FIFO排序以及全－因果和全－FIFO的混合。为了简化讨论，我们在假定任何进程至多属于一个组的前提下定义这些排序。后面我们还讨论允许组之间有重叠的本质。

- FIFO排序 如果一个进程发$multicast(g, m)$，然后发$multicast(g, m')$，那么每个传递m'的
442 正确的进程将在m'前传递m。
- 因果排序 如果 $multicast(g, m) \rightarrow multicast(g, m')$，其中$\rightarrow$是发生在先关系且只由$g$的成员之间发送的消息引起，那么任何传递$m'$的正确的进程将在$m'$前传递$m$。

- 全排序　如果一个正确的进程在传递m'前传递消息m，那么任何其他传递m'的正确的进程将在m'前传递m。

因果排序隐含FIFO排序，因为同一进程的任何两个组播都被发生在先关系联系起来。注意FIFO排序和因果排序都只是偏序：一般地，不是所有的消息都由同一进程发送；同样地，一些组播是并发的（不是按发生在先关系排序）。

图11-12说明了3个进程情况下的排序。对图的仔细检查表明，全排序消息的传递与它们被发送的物理时间的顺序相反。事实上，全排序的定义允许消息的传递可以随机排序，条件是该顺序在不同进程中是一样的。因为全排序不必同时也是FIFO或因果排序，我们把FIFO-全的混合排序定义为消息传递既遵守FIFO也遵守全排序的排序；同样地，在因果-全排序下，消息传递既遵守因果也遵守全排序。



注意　全排序的消息T_1和T_2，FIFO关系的消息F_1和F_2和因果关系的消息C_1和C_3之间一致的排序，以及消息的其他随机传递顺序

图11-12　组播消息的全排序，FIFO排序和因果排序

有序组播的定义并不假定或隐含可靠性。例如，读者可以证明，在全排序下，如果正确的进程p传递消息m然后传递m'，那么正确的进程q可以传递m而不传递m'或按序排在m后的任何消息。

我们也可以构造有序的和可靠的混合协议。一个可靠的全排序的组播在文献中常被称为原子的组播。同样地，我们可以构造可靠的FIFO组播、可靠的因果组播和混合排序组播的可靠版本。

正如我们看到的那样，对组播消息的传递排序在传递延迟和带宽消耗方面是昂贵的。我

们已描述的排序语义可能不必要地延迟消息的传递，即，在应用层，一个消息可能因为另一个它事实上不依赖的消息而被延迟。因为这个原因，一些人提出了只用应用特定的消息语义来确定消息传递的顺序[Cheriton and Skeen 1993, Pedone and Schiper 1999]。

公告牌的例子 为使组播传递语义更具体，考虑用户张贴消息到公告牌的应用。每个用户运行一个公告牌应用进程。每个讨论的主题有自己的进程组。当一个用户张贴一个消息到一个公告牌时，应用进程把用户的张贴组播到相应的组。每个用户的进程是他或她感兴趣的主题的组的成员，所以用户只收到关于这个主题的张贴。

如果每个用户最终要收到每个张贴，就需要可靠的组播。图11-13给出了出现在一个特定用户面前的张贴。在最低程度上，需要FIFO排序，因为这样才能使用户可以按同样的顺序收到来自一个给定用户（比如说“A.Hanlon”）的每一个张贴，这样，用户才可以一致地讨论A.Hanlon的第二个张贴。

公告牌：对操作系统感兴趣的		
编号	张贴人	主题
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re:Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

443
l
444

图11-13 公告牌程序的显示

注意，主题为“Re:Microkernels”（25）和“Re:Mach”（27）的消息出现在它们谈及的消息之后。为保证这个关系，需要因果排序的组播。否则，随机的消息延迟可能意味着，消息“Re:Mach”可能出现在最初的关于Mach的消息之前。

如果组播传递是全排序的，那么左边一栏的编号在用户之间是一致的。用户可以无二义地谈及某某消息如“消息24”。

实际上，USENET公告牌系统既未实现因果排序也未实现全排序。在大范围实现这些排序的通信代价超过了它们的好处。

实现FIFO排序 FIFO排序的组播（带有操作*FO-multicast*和*FO-deliver*）可以用顺序数实现，很像我们在一对一通信中实现的那样。我们只考虑非重叠组。读者可以验证，11.4.2节中我们在IP组播之上定义的可靠组播也保证了FIFO排序，但我们将展示如何在给定的任何基本组播之上构造FIFO排序的组播。我们使用11.4.2节可靠组播协议中进程p保存的变量S_g^p和R_g^q：S_g^p是进程p已发送到g的消息计数，R_g^q是p已传递的来自进程q并且发往组g的最近的消息的顺序数。

p要*FO-multicast*一个消息到组g时，它在消息上捎带值S_g^p，接着*B-multicast*消息到g，然后把S_g^p加1。当收到来自q的顺序数为S的消息时，p检查是否$S = R_g^q + 1$。如果是，这个消息是预期的来自发送进程q的下一个消息，p *FO-deliver* 消息，并且置$R_g^q := S$。如果$S > R_g^q + 1$，它把消息放到保留队列中，直到介于其间的消息已被传递且$S = R_g^q + 1$。

因为来自一个给定发送进程的所有消息是以同样的次序传递，并且消息的传递被延迟直到到达该顺序数，显然FIFO排序的条件是满足的。但是这仅在组不重叠的假定下成立。

注意，在这个协议中，我们可以使用*B-multicast*的任何实现。而且，如果用可靠的*R-multicast*代替*B-multicast*，则可以获得可靠的FIFO组播。

实现全排序组播 实现全排序的基本途径是为组播消息指派全排序标识，使得每个进程可以基于这些标识做出相同的排序决定。传递算法与我们为FIFO排序描述的算法很相似；区别是进程保持组特定的顺序数，而不是进程特定的顺序数。我们只考虑如何全排序发送到非重叠组的消息。我们把组播操作称为*TO-multicast*和*TO-deliver*。

我们讨论为消息指派标识的两种主要方法。第一种方法是由一个叫做顺序者的进程来指派标识（图11-14）。一个要*TO-multicast*消息m到组g的进程把一个惟一的标识$id(m)$附加到消息上。发往g的消息在被发送到g的成员的同时，也被发送到g的顺序者$sequencer(g)$。（顺序者可以选择为g的一个成员。）进程$sequencer(g)$维护一个组特定的顺序数s_g，用来给它*B-deliver*的消息指派连续的且不断增大的顺序数。它通过给g发送*B-multicast*顺序消息来宣布顺序数（详见图11-14）。 445

1. 组成员p的算法

初始化：r_g:=0;

为了给组g发*TO-multicast*消息：

B-multicas($g \cup \{sequencer(g)\}$,<m,i>);

在*B-deliver*(<m,i>)时，其中g=*group*(m)

　　将<m,i>放在保留队列中；

在*B-deliver*(m_{order}=< "order"，i, S>)时，其中g=*group*(m_{order})

　　wait until<m,i>在保留队列中并且$S=r_g$;

　　TO-deliver m; //在从保留队列中删除它之后

　　$r_g=S+1$;

2. 顺序者g的算法

初始化：s_g=0;

在*B-deliver*(<m,i>)时，其中g=*group*(m)

B-multicast(g,< "order" ,i,s_g>);

s_g:=s_g+1;

图11-14 使用顺序者的全排序

一个消息将无限地保留在保留队列中，直到它依照相应的顺序数可以被*TO-deliver*。因为顺序数是（被顺序者）明确定义的，所以满足全排序的标准。而且，如果进程使用*B-multicast*的一个FIFO排序的变种，则全排序的组播也是因果序的。我们把这个证明留给读者。

基于顺序者的方案有一个明显的问题是，顺序者可能成为瓶颈，并且是故障的一个关键点。有一些解决故障问题的实用算法。Chang和Maxemchuk [1984] 首先提出了一个使用一个顺序者（它们称为标记场地）的组播协议。Kaashoek等人 [1989] 为Amoeba系统发展了一个基于顺序者的协议。这些协议保证一个消息被传递前在$f+1$个结点的保留队列中，因此多达f个故障可以被容忍。像Chang和Maxemchuk一样，Birman等人[1991]也使用一个作为顺序者的

标记保留场地。标记可以在进程之间传递，使得如果只有一个进程发送全排序的组播，那么这个进程可以作为顺序者，从而节省通信。

Kaashoek等人的协议使用基于硬件的组播——可在以太网上使用的——而不是可靠的点对点通信。在他们的协议的最简单的变种里，进程把要组播的消息一对一发送到顺序者。顺序
446 者把消息本身连同标识和顺序数一起组播。这样的优点是组中其他成员每次组播只接收一个消息；缺点是带宽使用增加。完整的协议描述见www.cdk3.net/coordination。

我们研究的实现全排序组播的第二种方法，是一种进程以分布式方式集体地对分配给消息的顺序数达成一致的方法。一个简单的算法——与最初为ISIS工具箱开发的实现全排序的组播传递的算法 [Birman and Joseph 1987a] 类似——如图11-15所示。又是一个进程把消息*B-multicast*到组成员。组可以是开放或封闭的。当消息到达时，接收进程提出消息的顺序数，并把它们返回给发送者，后者用这些顺序数来产生协定的顺序数。

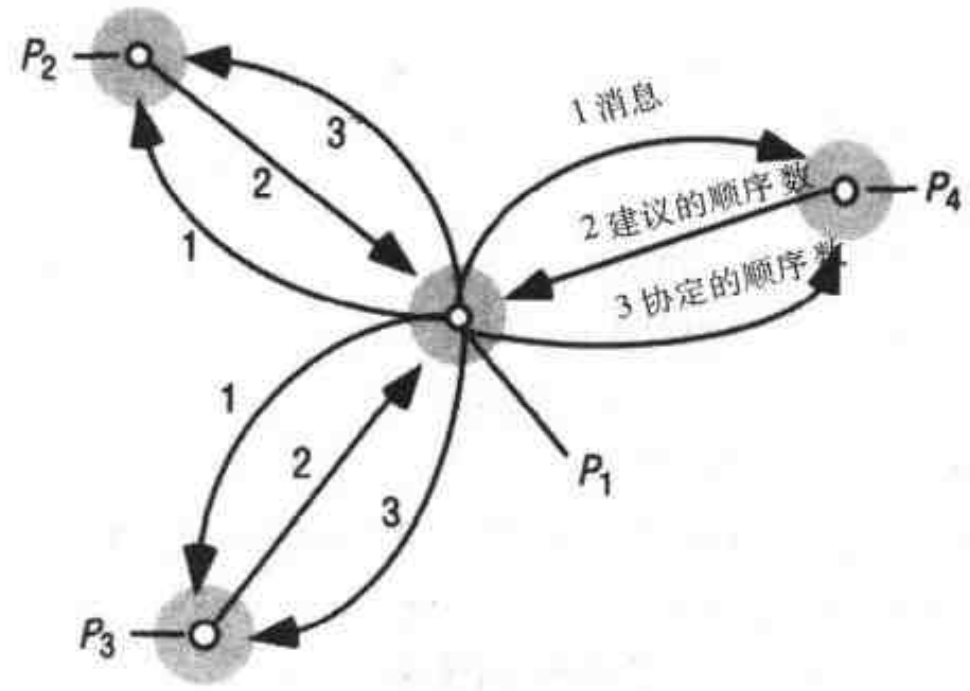


图11-15 全排序的ISIS算法

组g中的每个进程q保存A_g^q，即它迄今为止从组g观察到的最大的协定顺序数，和P_g^q，即它自己提出的最大顺序数。进程p组播消息m到组g的算法如下：

1. p *B-multicast* $<m, i>$到g，其中i是m的一个惟一的标识。

2. 每个进程q应答发送者p，提议$P_g^q := Max(A_g^q, P_g^q)+1$为此消息的协定顺序数。实际上，在提议的$P_g^q$里必须包括进程标识以保证全排序，否则不同的进程可能提议相同的整数值；但为简单起见我们在这里不显式地这样做。每个进程临时把提议的顺序数分配给消息，并把消息放入它的保留队列中，保留队列的顺序是最小的顺序数在队首。

3. p收集所有提议的顺序数，并选择最大的数a作为下一个协定顺序数。然后它*B-multicast* $<i, a>$到g。g中每个进程q置$A_g^q := Max(A_g^q, a)$，并把a附加到消息（标识为i）上。如果协定顺序数与提议的不一样，它把保留队列中的消息重新排序。当在保留队列队首的
447 消息被赋予协定顺序数时，它被转移到传递队列的队尾。但是，已被赋予协定顺序数、但不在保留队列队首的消息不被转移。

如果每个进程同意同一组顺序数，并按相应的顺序传递它们，那么全排序是满足的。显然，正确的进程最终会对同一组顺序数达成一致。但我们必须指出顺序数是单调递增的，并且正确的进程不能过早地传递消息。

假定消息m_i被指派了一个协定顺序数，并已到达保留队列的队首。根据构造规则，在这

阶段以后收到的消息将在、也应在m_1后传递：它将有一个比m_1'大的提议顺序数，因此也有一个比m_1大的协定顺序数。这样，令m_2是尚未指定协定顺序数、但在同一队列中的任何其他消息。根据刚给出的算法，我们有：

$$agreedSequence(m_2) \geq proposedSequence(m_2)$$

因为m_1在队首：

$$proposedSequence(m_2) > agreedSequence(m_1)$$

所以：

$$agreedSequence(m_2) > agreedSequence(m_1)$$

这样，全排序得到了保证。

这个算法比基于顺序者的组播有更大的延迟：在一个消息可以被传递前，在发送者和组之间要串行发送3个消息。

注意这个算法选择的全排序并不保证因果或FIFO序：受通信延迟的影响，任意两个消息被按着本质上随机的全排序来传递。

实现全排序的其他方法见Melliar-Smith等人[1990]，Garcia-Molina、Spauster [1991]和Hadzilacos、Toueg [1994]的文章。

实现因果排序 图11-16给出了一个非重叠封闭组的算法，该算法基于Birman等人[1991]开发的算法，其中因果序组播操作是*CO-multicast*和*CO-deliver*。该算法只考虑由组播消息建立的发生在先关系。如果进程互相发送一对一消息，那么这些进程将不会被考虑。

```
对组成员p_i（i=1,2,...,N）的算法
初始化：
    V_i^g[j]:=0(j=1,2,...,N);
        为了给组g发CO-multicast消息：
    V_i^g[i]:= V_i^g[i]+1；
    B-multicast(g,< V_i^g[j],m>);
在B-deliver(< V_j^g[j],m>)来自p_j(j≠0)的一个消息时,其中g=group(m)
将< V_j^g[j], m>放入保留队列;
    直到 V_i^g[j]=V_i^g[j]+且 V_j^g[k]  V_i^g[k] (k≠
    CO-deliver m; //在把它从保留队列中删除后
    V_i^g[j]=V_i^g[j]+1；
```

图11-16 使用时间戳向量的因果排序

每个进程p_i ($i = 1, 2, \cdots, N$)维护自己的时间戳向量（见10.4节）。时间戳的分量记录来自每个进程的、发生在下一个要组播的消息之前的组播消息数。

为了*CO-multicast*一个消息到组g，进程在时间戳的相应分量上加1，并且把消息和时间戳*B-multicast*到g。

当进程p_i *B-deliver*来自p_j的一个消息时，它必须在它能*CO-deliver*该消息前把消息放入保留队列中：直到可以保证它已经传递了按因果关系在该消息前的任何消息。为实现这个目的，p_i等待直到（a）它已传递了由p_j发送的任何较早的消息；（b）它已传递了p_j在组播该消息时已传递的任何消息。这些条件都可以通过检查时间戳来检测，参见图11-16。注意一个进程可以把它*CO-multicast*的任何消息立即*CO-deliver*到它自己，虽然这在图11-16中没有被描述。

每个进程在传递消息时，要更新它的时间戳向量以维护按因果关系在先的消息计数。它是通过把时间戳的第j个分量加一来做到这一点的。这是对10.4节在更新时钟向量的规则里出现的合并操作的一种优化。考虑到图11-16的算法中传递条件保证只有第j个分量会增加，我们可以做到这种优化。

我们概述此算法的正确性如下。假设$multicast(g, m) \rightarrow multicast(g, m')$。令$V$和$V'$分别是$m$和$m'$的向量时间戳。从算法可以简单地归纳证明$V < V'$。特别地，如果进程$p_k$组播$m$，那么$V[k] \leqslant V'[k]$。

考虑当某个正确的进程p_i *B-deliver* m'（与*CO-deliver*相反）但没有先*CO-deliver* m时，会发生什么。根据算法，仅当p_i组播一个来自p_k的消息时，$V_i[k]$可以加1。但p_i还没有收到m，因此$V_i[k]$的增长不可能超过$V[k] - 1$。于是p_i不可能*CO-deliver* m'，因为这需要$V_i[k] \geqslant V'[k]$，从而$V_i[k] \geqslant V[k]$。

读者应该证明，如果用可靠的*R-multicast*原语替换*B-multicast*，能得到既可靠又是因果序的组播。

此外，如果把因果组播协议和基于顺序者的全排序传递协议结合起来，那么我们就得到既是全排序又是因果序的消息传递。顺序者根据因果序传递消息，并按收到消息的次序组播消息的顺序数。目的组中进程，直到收到了来自顺序者的排序消息，并且一个消息是传递队列中的下一个时，才发送此消息。

448 ~ 449

因为顺序者按因果序传递消息，并且所有其他进程按与顺序者相同的顺序传递消息，因此确实既是全排序又是因果序。

组重叠 在FIFO、全排序和因果排序语义的定义和相关算法中，我们只考虑非重叠的组。这样简化了问题，但这并不令人满意，因为进程一般需要成为多个重叠组的成员。例如，一个进程可能对来自多个地方的事件感兴趣，并因此要加入事件分发组的相应集合。

我们可以把排序定义扩展为全局排序 [Hadzilacos and Toueg 1994]，其中我们必须考虑如果消息m被组播到g，且消息m'被组播到g'，则两个消息被发到$g \cap g'$的成员。

- *全局*FIFO排序 如果一个正确的进程发出$multicast(g, m)$，然后是$multicast(g', m')$，则$g \cap g'$中的每一个传递m'的正确的进程将在m'前传递m。
- *全局的因果排序*：如果$multicast(g, m) \rightarrow multicast(g, m')$，其中→是任何组播消息链都包含的发生在先关系，则$g \cap g'$中的任何传递$m'$的正确的进程将在$m'$前传递$m$。
- *进程对的全排序* 如果一个正确的进程在传递发送到g'的消息m'前传递了发送到g的消息m，则$g \cap g'$中的任何传递m'的其他正确的进程将在m'前传递m。
- *全局的全排序* 令“<”是传递事件之间的排序关系，我们要求“<”遵守进程对的全排序，并且无环——在进程对的全排序下，默认情况下“<”不是无环的。

实现这些排序的一种方法可能是组播每个消息m到系统中所有进程的组。每个进程根据消息是否属于$group(m)$来放弃或传递消息。这会是一个低效的和不令人满意的实现：除了目的组的成员以外，组播应该涉及尽可能少的进程。在 Birman等人[1991]、Garcia-Molina和Spauster [1991]、Hadzilacos和Toueg [1994]、Kindberg [1995]以及Rodrigues等人 [1998] 的文章中研究了其他的方法。

在同步和异步系统中的组播 本节描述了可靠的无序组播、（可靠的）FIFO排序的组播、（可靠的）因果序组播和全排序组播的算法。我们还指出如何实现既是全排序又是因果序的组

播。我们把既保证FIFO序又保证全排序的组播原语的算法的设计留给读者。我们描述的所有算法在异步系统中工作正确。

然而，我们没有给出一个算法，能既保证可靠的传递又保证全排序的传递。虽然看起来令人惊奇，但，具有这些保证的协议在同步系统中是可能的同时，这些协议在异步的分布式系统中是不可能的——即使是一个在最坏情况下忍受单个进程崩溃故障的协议。我们在下节再回到这一点。 450

11.5 共识和相关问题

本节介绍共识问题[Pease *et al.* 1980, Lamport *et al.* 1982]、相关的拜占庭将军问题和交互一致性问题。我们把这些问题统称为协定。粗略地说，该问题是在一个或多个进程提议了一个值应当是什么后，使进程对这个值达成协定。

例如，第2章描述了一种两个部队要一致地决定进攻或撤退的情形。相似地，我们要求，在每一个计算机提议了一个动作后，控制飞船引擎的所有正确的计算机要决定“继续”或“放弃”。在把一笔资金从一个账户转到另一账户的事务里，涉及的计算机必须对相应的借和贷达成一致。在互斥中，进程对哪个进程可以进入临界区达成协定。在选举中，进程对当选进程达成协定。在全排序组播里，进程对消息传递顺序达成协定。

适合这些个别类型协定的协议是存在的。我们在上面描述了它们中的一些，在第12章和第13章还会研究事务。但是考虑协定的更一般形式，探索共同的特点和解决方案，对我们是有用的。

本节更精确地定义共识，并把它与3个相关的协定问题相联系：拜占庭将军问题、交互一致性问题和全排序组播问题。接下来我们研究在什么情况下这些问题可解，并概述一些解决方案。特别地，我们将讨论众所周知的 Fischer等人[1985] 的不可能性结果，它声明在异步系统中，只含有一个有错进程的进程组不能保证达成共识。最后，我们考虑尽管有不可能性结果，实用算法是如何存在的。

11.5.1 系统模型和问题定义

我们的系统模型包括一组通过消息传递进行通信的进程p_i ($i = 1, 2, \cdots, N$)。在许多实际情况下，一个重要的要求是，即使有故障也要能达成共识。如前所述，我们假设通信是可靠的，但是进程可能出现故障。本节将考虑拜占庭（随机）进程故障以及崩溃故障。我们有时假设N个进程中至多有f个是有错的——即它们表现出某些特定类型的错误，其余的进程是正确的。

如果可以出现随机故障，那么刻画系统的另一因素是进程是否对它们发送的消息进行数字签名（参见7.4节）。如果进程对它们的消息签名，那么一个故障进程可能造成的伤害就受到限制。特别地，在一个协定算法过程中，它对一个正确的进程发送给它的值不能做出错误的断言。当我们讨论拜占庭将军问题的解时，消息签名的相关性将变得更为清楚。默认情况下，我们假设不进行签名。 451

共识问题的定义 为达到共识，每个进程p_i开始于一个未决状态，并且提议集合D中的一个值v_i ($i = 1, 2, \cdots, N$)。进程之间互相通信，交换值。然后，每个进程设置一个决定变量d_i ($i = 1, 2, \cdots, N$)的值。图11-17给出了参加一个共识算法的3个进程。两个进程提议“继续”，第三个进程提议“放弃”但随后崩溃。保持正确的两个进程每个都决定“继续”。

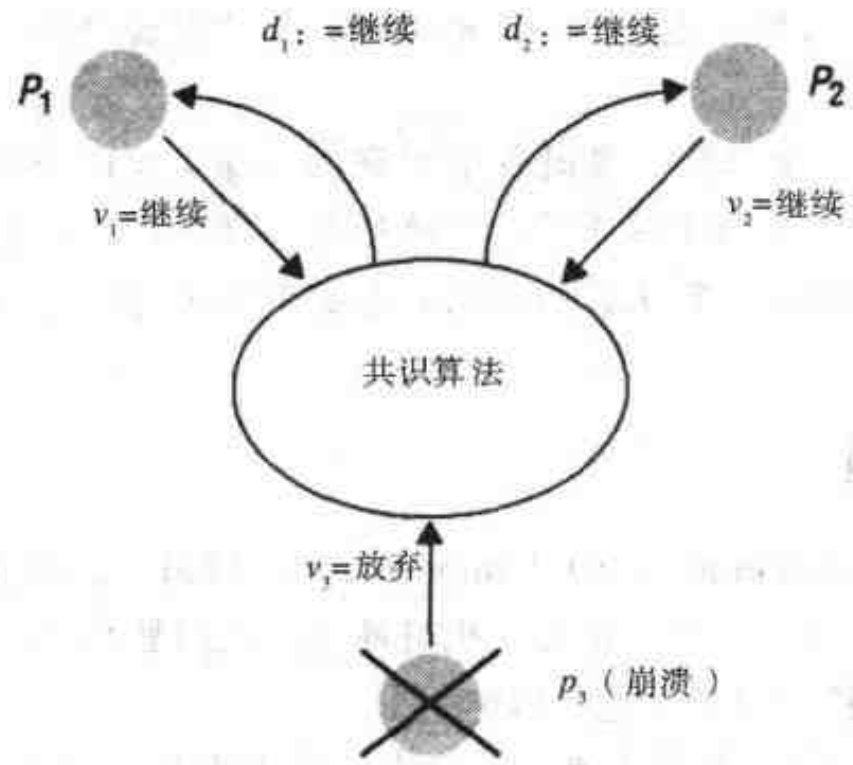


图11-17 3个进程的共识

共识算法的要求是在每次运行中满足以下条件：

- 终止性 每个正确进程最终设置它的决定变量。
- 协定性 所有正确进程的决定值都相同：如果p_i和p_j是正确的并且已进入决定状态，那么$d_i = d_j\ (i, j = 1, 2, \cdots, N)$。
- 完整性 如果正确的进程都提议相同值，那么在决定状态的任何正确进程已选择了该值。

根据应用不同，完整性定义可以有变化。例如，一种较弱的完整是决定值等于某些正确进程提议的值，而不必是所有进程。我们将使用上面的定义。

为帮助理解问题的表达是如何翻译为算法的，考虑进程不出故障的一个系统。这时，解决共识是简单的。例如，我们可以把进程集中为一组，并让每个进程可靠地组播它提议的值到组中的成员。每个进程等待，直到它收集到N个值（包括它自己的）。然后它计算函数$majority(v_1, v_2, \cdots, v_N)$，该函数返回它的参数中出现最多的值，如果没有，返回特殊值$\perp \notin D$。

452

终止性由组播操作的可靠性保证。协定性和完整性由*majority*的定义和可靠组播的完整性保证。每个进程收到相同的提议值集合，并且每个进程计算这些值上的同一函数。因此它们一定都一致，并且如果每个进程提议相同的值，那么它们都决定这个值。

值得注意的是，这些进程为了从候选值中选出一个共同认可的值可以采用的函数有很多，*majority*只是其中之一。例如，如果那些值是有序的，那么函数*minimum*、*maximum*也是合适的函数。

如果进程可能崩溃，那么，检测错误本身就很复杂，共识算法的执行是否能够终止并不是马上就能得出的。事实上，如果系统是异步的，它可能不会终止，后面将讨论这个问题。

如果进程以随机（拜占庭）方式出现故障，那么出错的进程原则上可以向别的进程发送任何数据。虽然在实际中看起来是不可能的，但是一个有漏洞的进程确实可能会出现这样的错误。而且，这样的错误可能不是偶然的，而是一些恶意的操作的结果。某些人可能故意让一个进程给一组进程中不同进程发送不同的值，借此来阻止这组进程达到一致性。如果遇到这种不一致的情况，正确的进程必须用他们自己接收的值和别的进程声明的所接收到的值进行比较。

拜占庭将军问题 拜占庭将军问题[Lamport *et al.*1982]可以非正式地表述成：3个或者更多的将军协商是进攻还是撤退。一个将军，司令，发布命令；其他的，作为司令手下的中尉，决定进攻还是撤退。但是一个或者多个将军可能会叛变，也就是说，出错。如果司令叛变，他可能会让一个中尉进攻，而让另一个中尉撤退。如果一个中尉叛变，他可能告诉其他某个中尉说司令让他进攻，而告诉另一个中尉说司令让他撤退。

拜占庭将军问题和共识问题的区别在于：前者有一个独立的进程提供一个值，其他的进程来决定是否采取这个值；而后者是每个进程都提供一个值。拜占庭将军问题的要求如下：

- 终止性 每个正确进程最终设置它的决定变量。
- 协定性 所有正确进程的决定值都相同：如果p_i和p_j是正确的并且已进入决定状态，那么$d_i = d_j$ $(i, j = 1,2, \cdots, N)$。
- 完整性 如果司令进程是正确的，那么所有正确的进程都采取司令提议的那个值。

值得注意的是，在拜占庭将军问题中，当司令正确的时候，完整性隐含着协定性，但是司令并不需要一定是正确的。

交互一致性 交互一致性问题是共识问题的另一个变种，这个问题中每个进程都提供一个值。算法的目的是正确的进程最终就一个向量达成一致，向量中的分量与一个进程的值对应。我们称这个向量为“决定向量”。例如，这样可以让一组进程中的每一个进程获得相同的关于该组中每一个进程的状态信息。

交互一致性的要求如下：

- 终止性 每个正确进程最终设置它的决定变量。
- 协定性 所有正确进程的决定向量都相同。 453
- 完整性 如果进程p_j是正确的，那么所有正确的进程都把v_j作为它们决定向量中的第i个分量。

共识问题与其他问题的关联 虽然人们通常用随机进程故障考虑拜占庭将军问题，但是实际上3个问题——共识、拜占庭将军、交互一致性——中每一个在随机故障和崩溃故障的环境中都是有意义的。同样，它们中的每一个都可以用于同步或者异步的系统。

有时候可以用解决另一个问题的方法获得解决这个问题的方法。这是一个很有用的性质，不仅是因为增加了我们对问题的理解，也是因为通过重用已有的解决方案，我们能潜在地节约实现的付出以及复杂性。

假设存在如下方法能够解决共识（C）、拜占庭将军(BG)和交互一致性(IC)：

- 在一个对共识问题的解决方案中，$C_i(v_1, v_2, \cdots, v_N)$返回进程$p_i$的决定值，其中$v_1, v_2, \cdots, v_N$代表进程所提议的值。
- 在一个对拜占庭将军的解决方案中，$BG_i(j, v)$返回进程p_i的决定值，其中p_j是司令，他建议的值是v。
- 在一个对交互一致性问题的解决方案中，$IC_i(v_1, v_2, \cdots, v_N)[j]$返回进程$p_i$的决定向量的第$j$个分量，其中$v_1, v_2, \cdots, v_N$是各个进程提议的值。

在对C_i、BG_i、IC_i的定义中，我们假设一个有错的进程提议一个概念值，也就是说虽然它可能对不同的进程提供不同的值，我们只用一个概念值。这只是为了方便：我们的解决方案不会依赖于这个概念值的具体内容。

可以从对另外问题的解决方案中构造出对一个问题的解决方案。我们给出如下的3个例子：

- 从*BG*构造*IC* 通过将BG算法运行*N*次，每次都以不同的进程$p_i(i, j=1, 2, \cdots, N)$作为司令，我们可以构造对*IC*的解决方法：

$$IC_i(v_1, v_2, \cdots, v_N)[j] = BG_i(j, v_j)\ (i, j=1,2,\cdots,N)$$

- 从*IC*构造*C* 通过运行*IC*算法能够在每个进程产生一个值向量，然后，在该向量值上使用一个适当的函数可以获得一个单一的值：

$C_i(v_1, v_2, \cdots, v_N) = majority(IC_i(v_1, v_2, \cdots, v_N)[1], \ldots, IC_i(v_1, v_2, \cdots, v_N)[N])$
$(i = 1, 2, \ldots, N)$，其中*majority*如前定义。

- 从*C*构造*BG* 我们采用如下的方式从*C*中构造*BG*的解决方案：
  - 司令进程p_j把它提议的值*v*发送给它自己以及其余的进程。
  - 所有的进程都用它们收到的那组值$v_1, v_2, \cdots, v_N$作为参数运行C算法（其中p_j可能是错误的）。
  - 最后得到 $BG_i(j, v) = C_i(v_1, v_2, \cdots, v_N)(i=1, 2, \ldots, N)$。

454

读者可以证明在每一个例子中都有终止性、协定性和完整性。Fischer[1983]提供了关于这3个问题的更多细节。

解决共识问题等同于解决可靠且全排序组播：给定其中的一个解决方案，就可以解决另一个。使用一个可靠且全排序组播操作*RTO-multicast*实现共识问题是显而易见的。我们将所有的进程组成一个组，设为*g*。为了达到共识，每个进程p_i运行*RTO-multicast*（g, v_i）。然后每个进程选择$d_i = m_i$，其中m_i是p_i *RTO-deliver*的第一个值。终止性是利用组播的可靠性得到的。协定性和完整性是利用组播的可靠性和全排序得到的。Chandra和Toueg[1996]证明了如何从共识问题中得出可靠且全排序组播。

11.5.2 同步系统中的共识问题

本节描述解决同步系统中共识问题的算法，该算法仅使用了一个基本的组播协议。算法假设*N*个进程中最多有*f*个进程会出现崩溃故障。

为了达到共识，每个正确的进程从别的进程那里收集提议值。算法进行*f*+1个回合，在每个回合中，正确的进程*B-multicast*值。根据假设，最多*f*个进程可能崩溃。最坏的情况下，所有*f*个进程都崩溃了，但是算法还是能够保证在这些回合结束后，所有活下来的正确的进程处于一个一致的状态。

该算法参见图11-18所示，是基于Dolev和Strong[1983]的算法，以及Attiya和Welch[1998]的表达方式。在第*r*个回合开始的时候，进程p_i将自己知道的那组提议值存放在变量$Values_i^r$中。每个进程都将自己前一个回合没有发出的那个值集合组播出去。然后它接收从别的进程组播来的相似的消息，并且记录新的值。虽然图11-18中没有提到最大时限，但是每个回合持续的时间是基于每个正确的进程组播消息所需要的最长时间来确定的。经过*f*+1个回合以后，每个进程选择它所收到的最小值作为它的决定值。

既然系统是同步的，终止性是显然的。为了检查算法的正确性，我们必须能够证明每个进程在最后一个回合结束的时候，每个进程达到一个相同的值集合。同时因为进程选择了*minimum*函数作用于这个集合，所以能够保证协定性和完整性。

反之，假设两个进程的最终值不同。不失一般性，某个正确的进程p_i所得到的值是*v*，另一个正确的进程 p_j（$i \neq j$）得到的值不是*v*。出现这种情况惟一的解释是另外还有一个进程，

假设是p_k,在把v传送给p_i后，还没有来得及传送给p_j，就崩溃掉了。同样道理，在前一个回合里p_k得到值v而p_j没有得到值的惟一解释是在前一个回合中发送v的进程崩溃掉了。以此类推，每个回合至少一个进程崩溃掉了。但是我们假设最多只有f个进程崩溃掉了，而我们进行了$f+1$个回合。这样就得出了矛盾。

```
对属于g的进程p_i的算法：算法进行到f+1 轮
初始化
    Values_i^1 := {v_i}; Values_i^0 = {};
在第r轮(1 ≤ r ≤ f+1)
    B-multicast(g, Values_i^r − Values_i^(r−1)); //仅发送还没有发送的值

    Valuse_i^(r+1) = Values_i^r;
    while(在第r轮)
    {
            在B-deliver(V) 来自p_j的消息时
            Valuse_i^(r+1) = Values_i^r ∪ V_j;
    }
在(f+1)轮之后
    将d_i赋成minimum( Valuse_i^(f+1) );
```

图11-18 同步系统中的共识

事实上，不管如何构造，如果要在至多f个进程崩溃的情况下仍然能够达到共识，必须要进行$f+1$轮的信息交换[Dolev and Strong 1983]。这个下限同样适用于拜占庭故障。[Fischer and Lynch 1980]。

11.5.3 同步系统中的拜占庭将军问题

现在我们讨论同步系统中的拜占庭将军问题。与前一节描述的共识问题不同的是，现在我们假设进程可能出现随机故障错误。也就是说一个出错的进程可能在任何时刻发送任何消息，也可能漏发消息。假设N个进程中最多有f个有错误。正确的进程通过超时能发现丢失了信息，但是由于发送这个消息的进程可以沉默一段时间又发送消息，所以，这个正确的进程并不能断定发送者已经崩溃。

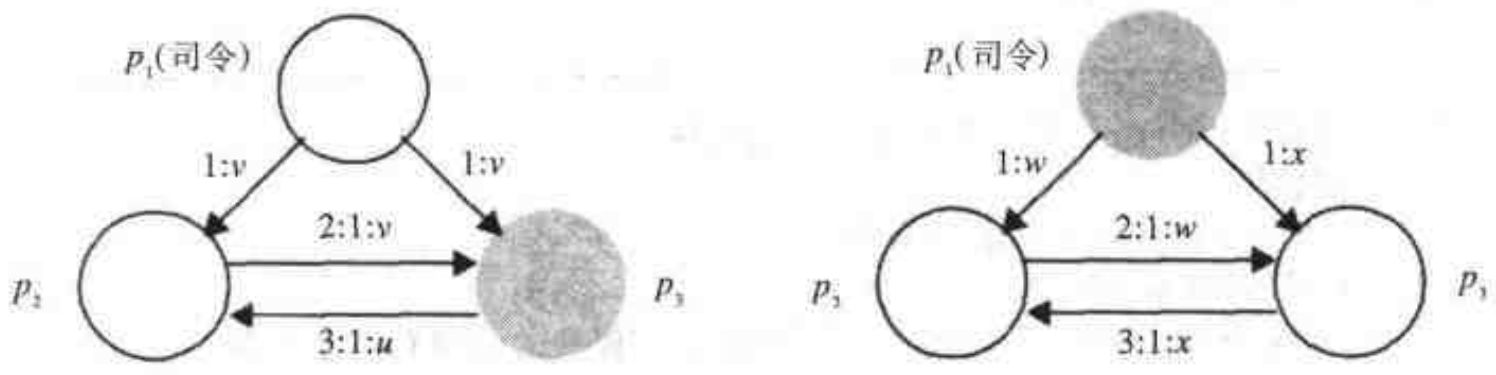
我们假设在每对进程之间的信道是私有的。如果一个进程可以检查其他进程发送的所有消息，那么它就可以发现一个错误的进程给不同进程发送的消息是不一致的。我们一般认为信息通道是可靠的，也就是说一个错误的进程不能把消息插入到正确进程之间的消息通道中。

Lamport等人[1982]讨论了3个进程相互之间发送未签名消息的情景。他们证明如果允许一个进程出现故障，那么将无法能够保证满足拜占庭将军问题的条件。他们还将这一结果推广到$N \leqslant 3f$，此时也没有解决方法。稍后我们将会简短说明这个结论。他们还给出一个算法，解决在同步系统中$N \geqslant 3f+1$的情况下未签名消息（他们将这些消息称为“口头的”）的拜占庭将军问题。

455 ~ 456

三个进程的不可能性 图11-19给出了3个进程中有一个进程出现错误的两种情况。在左边

的情况中，一个中尉p_3有错；对于右边的情况，司令p_1有错。图11-19 中给出了两个回合的消息交换：司令发送的值、两个中尉相互发送的值。数字前缀表明消息的来源，并且给出了不同的回合数。我们可以把消息中的“:”读成“说”，例如“3:1:u”读成“3说1说u”。



有故障的进程用阴影显示

图11-19 3个拜占庭将军

在左边的情况中，司令正确地将同一个值v发送给其他的两个进程，p_2正确地将这个消息发送给p_3。然而，p_3将$u \neq v$发送给p_2。p_2知道的只是它收到了两个不同的值，它并不能判断哪个值是司令传过来的。

在右边的情况中，司令有错误，它发给两个中尉的值是不同的。p_3发送了它收到的值x后，p_2处于和前一种情况（p_3有错时）相同的状态：它也收到两个不同的值。

如果存在一个解决办法，那么当司令是正确的时候，进程p_2必须决定值v，这是完整性条件所约束的。如果我们接受没有算法能够区分这两种情况，则p_2还是必须选择右边情况中司令发送的值。

如果对p_3做完全相同的推理，假设p_3是正确的，由于对称性，我们必须得出结论：p_3也选择司令发来的值作为它的决定值。但这就违反了协定性条件（司令出错的时候对不同的进程发出了不同的值）。所以，不存在可能的解决办法。

注意，上面的讨论基于我们的直觉，那就是在第一阶段我们不能分辨哪个进程是错误的，而在以后我们也无法增加一个正确进程的知识。我们可以证明这一直觉的正确性[Pease *et al.*1980]。如果将军们能够数字签名它们发出的消息，那么，3个将军中有一个有错，也能实现拜占庭协定。

对于N≤3f的不可能性 Pease等人推广了3个将军的不可能结论，证明只要$N \leqslant 3f$，就不可能有解决方法。下面给出证明的轮廓。假设在$N \leqslant 3f$时，有一个解决方案。我们假设3个进程p_1，p_2，p_3分别模拟n_1，n_2，n_3个将军，其中$n_1 + n_2 + n_3 = N$并且n_1，n_2，$n_3 \leqslant N/3$。我们进一步假设3个进程中有一个有错误。p_1，p_2，p_3中正确的进程模拟正确的将军：进程在内部模拟
457 内部将军之间的交互，并且自己的将军还会向被其他进程模拟的将军发送信息。错误的进程模拟出错的将军：它发送给其他两个进程的信息可能是伪造的。既然$N \leqslant 3f$并且n_1，n_2，$n_3 \leqslant N/3$，所以最多f个将军可能出错。

由于假设进程运行的算法是正确的，所以模拟能够终止。那些正确的将军（在那两个正确的进程中）就会达到结束点并且满足完整性。但是，这就是说3个进程中的两个达到了共识：每个进程对由所有将军选择的值做出决定。这就与前面的3个将军中有一个是有错将军的不可能性结论相矛盾。

对一个有错进程的解决方案 Pease等人提出了一个算法，解决同步系统中$N \geqslant 3f + 1$的拜

占庭将军问题。在这里没有足够的空间来讨论这个算法，但是我们将给出$N \geq 4$，$f = 1$的算法操作，并以$N = 4$，$f = 1$来说明。

正确的将军通过两轮消息取得一致：

- 第一轮，司令给每个中尉发送一个值。
- 第二轮，每个中尉将收到的值发送给自己的同等官衔的人。

每个中尉收到从司令来的一个值，以及从其他中尉来的$N-2$个值。如果司令有错，而所有中尉都是正确的，那么每个中尉都会收到司令发出的那些值。否则，一个中尉有错，它的其他同事收到司令发来的值的$N-2$份副本，再加上有错的中尉发来的一个值。

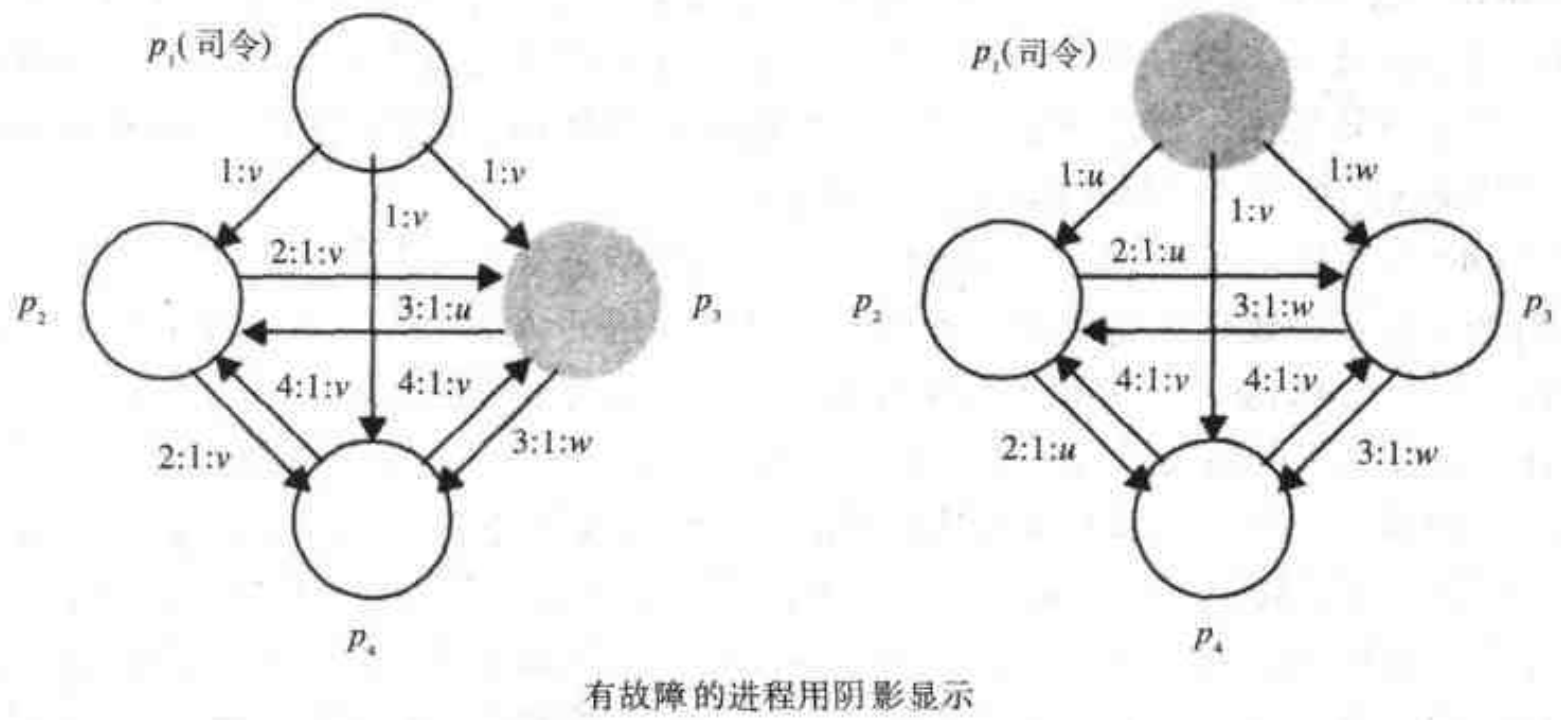
不管在那种情况下，每个正确中尉只需要对它们收到的值集合应用一个简单的*majority* 函数。既然 $N \geq 4$则$(N-2) \geq 2$。因此，*majority*函数会忽略掉出错中尉发来的值，并且当司令是正确的时候，该函数能产生司令发来的值。

我们用4个将军的情况说明上述算法。图11-20给出了与图11-19相近的两个情况，但是现在是4个进程，其中一个是有错的。像在图11-19中一样，在左边的图中，中尉p_3是有错的；在右边的图中，司令p_1是有错的。

当出现左边情况时，两个正确的中尉进程在决定司令的值时达成一致：

p_2决定 $majority(v, u, v) = v$

p_4决定 $majority(v, v, w) = v$



有故障的进程用阴影显示

图11-20 4个拜占庭将军

在右边的情况中，司令是有错的，但是正确的3个中尉进程能达成一致：

p_2、p_3 和p_4决定$majority(u, v, w) = \perp$（特殊值 $\perp$ 代表没有占多数的值存在）。

这个算法考虑了一个错误进程可能漏发消息。如果一个正确的进程在一个适当的时间范围内（系统是同步的）没有收到一个消息，它就当成错误进程向它发送了特殊值 $\perp$，然后继续处理。

讨论 对于一个解决拜占庭将军——或者其他协定问题——的算法，我们通过以下两个问题来度量其效率： 458

- 进行了多少轮消息传递？（这个因素影响算法终止需要的时间）
- 发送了多少消息，消息的长度是多少？（这个因素度量带宽的利用，并且会影响执行的时间）

一般情况下($f \geq 1$)，Lamport 等人的算法用于不签名的消息传送时，需要操作$f + 1$轮。在每轮中，每个进程发送它在前一轮中收到的其他进程发来的值的一个子集。算法代价很高，它需要发送$O(N^{f+1})$条信息。

Fischer 和 Lynch[1982]证明了如果允许出现拜占庭故障，那么任何确定性的解决共识问题的算法至少需要$f + 1$ 轮消息。所以在这个方面，没有算法能比 Lamport等人的算法更快。但是可以改善消息的复杂度，例如 Garay和Moses [1993]做的改进。

几个算法，例如 Dolev和Strong[1983]的算法，利用了对消息进行签名。他们的算法也需要进行$f + 1$轮，但是发送消息的数量仅是$O(N^2)$。

算法的复杂性和代价说明了只在安全威胁很严重的情况下才用这些算法。如果威胁来自硬件错误，那么出现随机行为错误的可能性是很小的。如果解决方案获得的错误模型的知识越详细，那么可以得到的解法就越有效[Barborak等1993]。如果威胁来自于恶意的用户，那么受到威胁的系统更可能使用数字签名，一个不使用签名的解决方案是不合实际的。

11.5.4 异步系统的不可能性

现在我们已经提供了同步系统中共识和拜占庭将军问题的解决方案（因此，由推导可得对交互一致性的解决方案）。然而这些算法都依赖于系统是同步的。算法假定消息交换是按轮
459 进行，进程有超时机制，可以因为超过最大延迟而认为出错的进程在那轮没有发送消息。

Fischer等人[1985]证明在一个异步系统中，即使是只有一个进程出现崩溃故障，也没有算法能够保证达到共识。因为在一个异步系统中，进程可以随机发出响应的消息，所以没有办法分辨一个进程是速度很慢还是已经崩溃。他们的证明显示了进程的执行总是有延续来阻止进程达到共识。详细的证明已经超出本书的范围。

从Fischer等人的结论中我们立刻可以得到：在异步系统中，我们没有可以确保解决拜占庭将军问题、交互一致性问题或者全排序可靠组播问题的方法。如果有这样的解决办法，根据11.5.1的结论，我们就会有共识问题的解决办法——这与不可能性结论是相矛盾的。

注意，我们在不可能性结论中使用了“确保”。这并不是说在分布式系统中，如果有一个进程出现了错误，进程就永远不可能达到共识。它允许我们达到共识的概率大于0，这与实际相符合。例如，尽管我们的系统通常是异步的，但是事务系统多年来一直能达到共识。

绕过不可能性结论的办法是考虑部分同步系统。部分同步系统比同步系统要弱，足够成为实际应用的系统；但又比异步系统要强，使得共识问题能够被解决[Dwork *et al* .1988]。这个方法同样超出了本书的范围。我们将简要介绍绕过不可能性结论的3种方法：故障屏蔽，利用故障检测器达到共识，随机化进程各方面的行为。

故障屏蔽 第一种完全避免不可能性结论的技术是屏蔽所有发生的进程故障(2.3.2节有故障屏蔽的介绍)。例如，事务系统使用持久储存，能够在发生崩溃的时候保留信息下来。如果一个进程崩溃掉了，它会被重启（自动的或者由管理者重启）。进程在程序的关键点在持久存储中保留了足够多的信息，以至于它在崩溃和重启时能够利用这些数据正确地继续被中断的工作。换句话说，它能够像正确的进程那样工作，只是有时候它需要很长时间来执行一个处理。

当然，故障屏蔽一般应用到系统设计中。第13章讨论了事务系统如何利用持久存储。第14章描述了如何利用软件组件的复制来屏蔽进程故障。

使用故障检测器达到共识 另一个绕过不可能性结论的方法是使用故障检测器。一些实际的系统使用“完美设计”的故障检测器来达到共识。实际上一个异步系统中如果检测器仅仅依靠消息传递是不可能真正达到完美的。然而进程可以协商后认为一个超过指定时间没有反应的进程已经出错了。一个没有响应的进程未必已经出错了，但是其余的进程把它看作已经出错了。它们把这个故障变成“失败－沉默”，将它们接下来收到的所有从出错的进程发来的消息全部抛弃。换句话说，我们已经有效地将一个异步系统转化为一个同步系统。这项 460
技术被应用在ISIS系统中 [Birman 1993]。

该方法需要故障检测器通常是精确的。如果不精确的话，系统在工作中可能放弃一个成员，而实际上这个成员能够为系统的效果做出贡献。但是，让故障检测器保证合理的精确性需要设定很长的超时值，这就需要进程等待一个相对较长的时间（并且不能进行有用的工作）才能得出一个进程已经出错的结论。这个方法还引起了另一个问题是网络分区，我们将在第14章讨论这个问题。

一个完全不同的方法是使用不精确的故障检测器，在达到共识的时候允许被怀疑的进程正确行动而不是排除出去。Chandra和Toueg[1996] 为了解决在异步系统中的共识，分析了一个故障检测器必须拥有的属性。他们证明了，即使是使用不可靠的故障检测器，只要通信是可靠的，崩溃的进程不超过$N / 2$，那么异步系统中的共识是可以解决的。我们称能够实现这个目标的最弱的故障检测器为*最终弱故障检测器*。该检测器具有如下性质：

- *最终弱完全* 每一个错误进程最终常常被一些正确进程怀疑。
- *最终弱精确* 经过某个时刻后，至少一个正确的进程从来不被其他正确的进程所怀疑。

Chandra和Toueg证明了在异步系统中，我们不能只依靠消息传递来实现一个最终弱故障检测器。但是，我们在11.1节中描述了一个基于消息的故障检测器，它能够根据观察到的响应时间调节它的超时值。如果一个进程或者一个到检测器的连接很慢，那么超时值就会增加以至于错误地怀疑一个进程的情况变得很少。在很多实际系统中，从实用目的看，这个算法与最终弱故障检测器非常相似。

Chandra和Toueg的共识算法允许被错误怀疑的进程继续它们正常的操作，并且允许怀疑它们的进程正常地接受它们发出的消息并处理。虽然这使得应用程序员的工作很复杂，但是这样做有个好处：正确的进程不会被错误地排斥出去而浪费。而且，与ISIS方法相比，故障检测的超时值可以不必那么保守。

使用随机化达到共识 Fischer等人的结论依赖于我们考虑的“敌人”是什么。这是一个“人物”（实际上是一个随机事件的集合），能够利用异步系统的现象来阻止进程达到共识。敌人操纵网络来延迟消息以便使它们在错误的时刻到达，或减缓或加速进程，使得当进程收到一个消息的时候处于错误的状态。

第3种解决不可能性结论的技术是引入一个关于进程行为的可能性元素，使得敌人不能有效地实施它们的阻碍战术。共识在有的情况下还是不能达到，但是这个方法使得进程能够在一个有限的被期望的时间内达到共识。Canetti和Rabin[1993]提出了一个概率算法可以解决共 461
识甚至拜占庭故障问题。

11.6 小结

本章开始讨论了进程在互斥条件下访问共享资源的必要性。锁并不总是由管理共享资源

的服务器实现的，所以一个单独的分布式互斥服务是必要的。我们考虑了3种实现互斥的算法：一种使用了中央服务器的算法，一种基于环的算法以及一种基于组播的使用逻辑时钟的算法。像我们描述的那样，它们中没有一个能够经受住故障，虽然经过修改它们能够容忍一些错误。

接下来本章考虑一个基于环的算法和霸道算法，它们共同的目的是从一个给定的集合中选出惟一的一个进程——即使几个选举同时发生。例如，在主时钟服务器或者锁服务器出故障时，霸道算法可用于选取一个新的服务器。

本章还描述了组播通信。讨论了可靠组播——正确的进程对要传递的消息集合达成一致——以及具有FIFO、因果、全排序的组播。我们给出了可靠组播的算法，还给出了所有3种传递顺序的算法。

最后我们描述了共识问题、拜占庭将军问题以及交互一致性问题。我们定义了它们的解决方案的条件，并且证明了这些问题之间的关系——包括共识和可靠全排序组播之间的关系。

在同步系统中可以解决上述问题，我们描述了一些算法。实际上，即使可能出现随机故障，解决的方法也是存在的。我们大致描述了Lamport 等的关于拜占庭将军问题的解法的一部分内容。最近的算法有更低的复杂度，但是原理上，没有一个算法能比该算法采用的$f + 1$轮更好，除非消息采用数字签名。

本章结尾描述了Fischer等人的基本结论，即关于异步系统中保证共识的不可能性。虽然有这样的结论，但我们仍然讨论了通常使异步系统达成一致的方法。

练习

11.1 使用一个不可靠的信道有没有可能实现一个可靠的或者不可靠（进程）的故障检测器?

11.2 如果所有的客户进程都是单线程的，那么用来按发生在先顺序指定位置的互斥条件ME3是否有用?

462

11.3 根据同步时延给出计算互斥系统最大吞吐量的公式。

11.4 在互斥用的中央服务器算法中，描述使得两个请求不是按照发生在先顺序处理的情景。

11.5 修改用于互斥的中央服务器算法，使之能够处理任何客户（在任何状态）的崩溃故障，假设服务器是正确的，并且有一个可靠的故障检测器。讨论这个系统是否能够容错。如果拥有令牌的客户被错误地怀疑为出了故障，会发生什么样的情况?

11.6 给出一个基于环的算法的执行的例子，用以表明进程不必以发生在先顺序授权进入临界区。

11.7 在某个系统中，每个进程常常多次使用一个临界区后另一个进程才需要访问。解释为什么Ricart和Agrawala的基于组播的互斥算法在这种情况下效率很低，描述如何提高它的效率。你的修改是否满足活性条件ME2?

11.8 在霸道算法中，恢复进程开始一个选举，并且如果它比当前的协调者进程有更大的标识，那么它就成为新的协调者。这是算法所必需的吗?

11.9 如何修改霸道算法以处理暂时的网络分区（通信变慢）以及处理过程速度变慢。

11.10 设计一个在IP组播上进行基本组播的协议。

11.11 对开放组的情况，怎样修改可靠组播的完整性、协定性、有效性定义。

11.12 在图11-10中，如果颠倒这两个语句的顺序："*R-deliver m*" 和 "*if*($q \neq p$) *then B-multicast*(g,m);*end if*"，那么算法将不再满足统一的协定。基于IP组播的可靠组播算法是否满足统一的协定?

11.13 解释为什么基于IP组播的可靠组播算法不适用于开放组。给定任何一个用于封闭组的算法，我们如何从它构造一个用于开放组的算法?

11.14 在基于IP组播的可靠组播协议中，为了达到有效性和协定性做了一些不合实际的假设，如何解决这些假设。提示：当一个消息被传递后，增加一个删除保留消息的规则；考虑增加一个哑"心跳"信息，这个信息永远不会发送给应用，而是当应用没有信息要发送的时候由协议发送。

11.15 在基于FIFO顺序的组播中，考虑同一个信息源发送两个信息给两个有重叠的组，以及一个处于两个组的交集中的进程，证明这个算法不适用于有重叠组。修改该算法使之能用于重叠组。提示：进程应该在它们的消息中包括发给所有组的消息的最新顺序号。

11.16 证明：如果我们在图11-14所示的基本组播算法中是FIFO排序的，那么组合成的全排序组播也是因果排序的。任何一个FIFO排序并且是全排序的组播是不是也是因果排序的? 463

11.17 考虑如何修改因果排序的组播协议来处理重叠组。

11.18 在讨论Maekawa的互斥算法的时候，我们给出了3个进程的3个子集可能导致死锁的例子。使用这些子集作为组播的组，证明为什么进程对的全排序不一定是无环的。

11.19 使用一个可靠组播和一个解决共识问题的方法，在同步系统中建立一个可靠的、全排序组播。

11.20 从可靠全排序组播（需要包括选择第一个可以传递的值）可以得到共识的解决方法。从基本原理解释，为什么在一个异步系统中，我们不能从可靠的但不是全排序的组播服务以及"majority"函数得到共识的解决方案。（注意如果我们能够做到，这就会与Fischer等的不可能性结论相矛盾！）提示：考虑速度慢的或者出故障的进程。

11.21 在3个将军的拜占庭将军问题中，证明如果将军对消息进行签名，那么一个将军有问题，也可以达成协定。 464

第12章　事务和并发控制

12.1　简介
12.2　事务
12.3　嵌套事务
12.4　锁
12.5　乐观并发控制
12.6　时间戳排序
12.7　并发控制方法的比较
12.8　小结

本章讨论事务和并发控制在服务器管理共享对象时的应用。

事务是服务器上的一个操作序列，由服务器保证这些操作序列在多个客户并发访问和服务器出现故障情况下的原子性。嵌套事务定义了若干事务之间的嵌套结构，在分布系统中可以具有更高的并发度。

所有的并发控制协议都是基于串行相等的标准，起源于用于解决操作间冲突的规则。本章描述了以下3种方法：

- 锁用于在多个事务访问同一个对象时，根据这些操作访问同一对象的先后次序给事务排序。
- 乐观并发控制不会阻塞事务运行，只是在提交时通过检查来确定已执行的操作是否存在冲突。
- 时间戳排序利用时间戳将访问同一对象的事务根据其起始时间进行排序。

12.1　简介

事务的目标是在多个事务访问对象以及服务器面临崩溃的情况下，保证所有由服务器管理的对象始终维持在一个一致的状态上。第2章介绍了分布式系统的故障模型。事务能够处理进程的崩溃故障和通信的遗漏故障，但不能处理任何随机（或拜占庭式的）行为。12.1.2节将给出事务的故障模型。

能够在服务器崩溃后恢复的对象被称为可恢复对象。通常这些对象存储在挥发性存储（例如RAM）或持久存储（例如硬盘）中。即使对象存放在挥发性存储中，服务器仍然可以利用持久存储来保存足够多的状态信息，以便在服务器进程崩溃后能够恢复这些对象。这使得服务器能保证对象是可恢复的。事务是由客户定义的针对服务器对象的一组操作，它们组成一个不可分割的单元，由服务器执行。服务器必须保证或者整个事务被执行并将执行结果记录到持久存储中，或者在出现故障时，能消除这些操作的所有影响。下一章将讨论涉及几个服务器的事务，特别是如何决定一个分布式事务的结果。本章重点研究单服务器上的事务。从其他客户事务的角度而言，一个客户的事务也被认为是不可分割的，因为一个事务中的操作不能观察到另一个事务中的操作的部分结果。12.1.1节介绍对象访问中的简单同步控制方法。12.2节介绍事务，事务需要防止客户之间冲突的更高级的并发控制技术。12.3节讨论嵌套事务。12.4节至12.6节分别讨论单服务器上的事务的3种并发控制方法，即锁、乐观并发控制和时间

戳排序。第13章进一步讨论如何将这些方法加以扩展，运用到多个服务器上的事务中。

为了方便本章讨论，我们使用了一个银行的例子，如图12-1所示。每个银行账户由一个远程对象表示，它支持一个*Account*接口，该接口提供存款、取款、查询和设置账面余额等操作。银行分行用一个远程对象表示，该对象实现*Branch*接口，该接口提供通过名字创建新账户、查找账户和计算分行总余额等操作。

*Account*接口的操作
deposit (*amount*) 向账户存*amount*数量的钱 *withdraw*(*amount*) 从账户中取*amount*数量的钱 *getBalance*()→*amount* 返回账户余额 *setBalance*(*amount*) 将账户余额设置成*amount*
Branch接口的操作
create(*name*)→*account* 用给定用户名创建一个新账户 *lookUp*(*name*) →*account* 根据给定用户名查找账户，并返回该账户的一个引用 *branchTotal*()→(*amount*) 返回支行中所有账户余额的总和

图12-1 Account和Branch接口的操作

12.1.1 简单的同步机制（无事务）

本章的一个主要问题是如果不仔细设计服务器，不同客户执行的操作有时会相互冲突。这种冲突会导致对象产生不正确的值。本节讨论没有事务时客户操作如何同步。

服务器上的原子操作 通过本书前面的章节，我们已经看到多线程的使用可以提高服务器的性能。我们也注意到使用多线程能够让不同的客户并发执行并且访问同一个对象。因此，对象必须被设计成支持多线程的运行环境。以银行为例，如果*deposit*方法和*withdraw*方法没有根据多线程的特殊性而设计，那么当多个线程并发执行这些方法时，可能会导致这些方法的交织执行，从而产生奇怪的账户对象数据。

第6章引入的*synchronized*关键字是应用在Java方法中用以保证一次只能有一个线程访问对象。在我们的例子中，实现*Account*接口的类可以将方法声明成同步的。例如：

```
public synchronized void deposit(int amount)throws RemoteException(
    //将amount数量的钱加入账户余额
}
```

当一个线程调用某个对象的同步方法时，该对象在调用期间将被一直锁住，这时如果另

一个线程也调用该同步方法，那么该线程将被阻塞，直到相应的锁被释放。这种形式的同步将线程的执行分隔在不同的时间中，从而保证对一个对象的实例变量的访问一致性。如果没有同步机制，那么两个不同的*deposit*方法调用可能在对方未更新前读取账户余额值——导致不正确的数据。因此，应该同步所有访问会发生变化的实例变量的方法。

免受其他线程中并发操作干扰的操作被称为原子操作。Java语言中的同步方法是实现原子操作的途径之一。在其他多线程服务器的编程环境中，为了保证对象的一致性，对象上的操作仍然需要原子操作，通过互斥机制例如mutex变量可实现这一点。 466 ~ 467

通过服务器操作的同步加强客户协同 客户可以将服务器作为一种共享资源的设施来使用。即一些客户调用服务器上对象的方法更新对象，而另一些客户调用方法访问对象。上述同步访问对象的机制提供了大多数应用中所需要的东西——避免了线程相互干扰。但是，某些应用需要线程间相互通信的机制。

例如，会出现这种情况：某个客户的操作要到另一个客户操作结束才能完成。一个典型的例子是某些客户是生产者而另一些客户是消费者——这些消费者在生产者提供更多的所需商品前必须等待。这种情况在客户共享某种资源时也会出现——请求资源的客户必须等待其他客户释放资源。在本章的后面部分，我们还会看到：在用锁或时间戳进行事务并发控制时，也会有类似的情况。

第6章介绍的Java *wait*和*notify*方法允许线程相互通信，它们能够解决上述问题。这两个方法必须用于同步方法中。当一个线程调用某个对象的*wait*方法后，该线程被挂起并允许其他线程执行该对象的方法。线程通过调用*notify*方法通知等待该对象的线程它已改变了该对象的一些数据。在线程等待时，对对象的访问仍是原子的，调用*wait*的线程把放弃锁和挂起自身作为单个原子动作。当线程被通知重新开始时，它需要重新获得对象上的锁，继续*wait*之后的执行。而调用*notify*的线程（从一个同步方法内）在它执行完当前方法后才会释放对象锁。

现在考虑共享对象*Queue*的实现，*Queue*有两个方法：*first*方法用于删除并返回队列中的第一个对象，*append*方法用于将一个给定对象放到队列尾部。*first*方法首先将检查队列是否为空，如果队列为空则调用该队列对象的*wait*。因此在队列为空时，某个客户调用*first*方法将不会返回，必须等待其他客户向队列发送对象——*append*方法在将对象加入队列时会调用*notify*，这使得等待队列对象的线程能继续执行，并将队列中的头一个对象返回给客户。在线程通过*wait*和*notify*同步对象操作时，对子不能立即满足的请求，服务器将暂时挂起它们，客户只有在另一个客户产生它们所需的数据后才能得到应答。

在后面关于事务锁的章节中，我们将讨论利用对象的同步操作来实现一个事务锁。当某个客户试图获取一个锁时，它首先必须等到其他客户释放该锁。

如果没有这种线程同步机制，那么请求不能马上满足的客户，例如客户在一个空队列上调用*first*方法，被告之以后重试。这种调用方式是不能令人满意的，因为它导致客户不断轮询服务器，加大服务器负载。另外，服务器在处理这些轮询时，其他客户必须等待，这也是不公平的。 468

12.1.2 事务的故障模型

Lampson[1981a]提出过一个分布事务的故障模型，包括了磁盘故障、服务器故障以及通信故障。该故障模型声称：可以保证算法在可预见故障下正确工作，但是对于不可预见的灾难性故障则不能保证正常处理。尽管可能会出现错误，但是可以在发生不正确行为之前发现并处理这些错误。Lampson的故障模型包括以下故障：

- 对持久存储的写操作可能发生故障——或因为写操作无效或因为写入错误的值——例如，将数据写到错误的磁盘块被认为是一个灾难性故障。文件存储有可能损坏。从持久存储中读数据时可根据校验和来判断数据块是否损坏。
- 服务器可以偶尔崩溃。当一个崩溃的服务器由一个新进程替代后，其内存被重置。此后新进程根据持久存储中的信息以及从其他进程获得的信息设置对象的值，包括与两阶段提交协议有关的对象的值（见第13.6节）。当一个处理器有故障时，服务器也会崩溃，这样它就不会发送错误的消息或将错误的值写入永久存储，即，它不会产生随机故障。服务器崩溃可能出现在任何时候，即使在服务器重启恢复时也可能出现。
- 消息传递可能有任意长的延迟。可能丢失、重复或者损坏消息。接收方通过校验和能够检测到受损消息。未发现的受损消息和伪造的消息会导致灾难性故障。

利用这个关于永久存储、处理器和通信的故障模型能够设计出一个可靠系统，该系统可对付任何单一故障。特别是，可设计出一个可靠存储，该存储提供一个原子写操作，即使出现一个写操作故障或者进程崩溃故障。它的实现是通过将每一个数据块复制到两个磁盘块上，此时一个写操作作用于两个磁盘块上，在出现单一故障的情况下，另一个好的数据块能提供正确数据。可靠处理器使用可靠存储，用于在崩溃后恢复对象。可通过可靠的远程过程调用机制来屏蔽通信错误。

12.2 事务

在某些情况下，客户要求给服务器的一组请求按下列意义是原子的：

1. 它们不受其他并发客户操作的干扰。

2. 所有操作或者全部成功完成，或者在服务器出现故障时，对所有操作没有任何影响。

让我们回到银行的例子来说明事务概念。当一个客户对特定账户操作时，它首先利用
469 *lookUp*——根据用户名查询到相应的银行账户，然后直接在相关账户上进行*deposit*、*withdraw*和*getBalance*操作。我们的例子使用了账户名为*A*、*B*和*C*的3个账户。客户查找这些名字并将它们的引用存储在*Account*类型的变量*a*、*b*和*c*中。为简化起见，我们略去了由名字查找账户和变量声明等细节。

图12-2给出了一个简单客户事务的例子，该事务指定了若干涉及账户*A*、*B*和*C*的相关动作。头两个动作是从账户*A*转账100元至账户*B*，后两个操作从账户*C*转账200元至账户*B*。客户是通过一个取款操作和一个存款操作完成转账的。

```
Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);
```

图12-2 一个客户的银行事务

事务起源于数据库管理系统。数据库管理系统中的事务是访问数据库的一个程序的执行。事务后来通过事务文件服务器，例如XDFS[Mitchell and Dion 1982]，被引入到分布式系统中。在事务文件服务器中，事务是指客户执行一组文件操作请求。在若干研究项目（如Argus[Liskov 1988]和Arjuna[Shrivastava *et al.* 1991]）中，事务又被引入分布式对象系统。这

时的事务是指一组客户请求的执行，如图12-2的例子所示。从客户角度来看，事务是组成一个步骤的一组操作，它将服务器的数据由一个一致性状态转换到另一个一致性状态。

事务可以作为中间件的一部分提供。例如，CORBA提供了对象事务服务规范[OMG 1997e]，它的IDL接口允许客户事务访问多个服务器上的多个对象。客户可利用有关操作来指定事务的开始和结束。客户ORB为每个事务维持一个事务上下文，事务上下文随着操作调用而传递。CORBA对象如果在它们的接口中扩展了*TransactionalObject*接口，那么它就能够支持事务。

在以上的讨论中，事务总是应用到可恢复对象上并具有原子性。事务常常被称作原子事务（见下面的阴影部分）。这里的原子性包含两方面的含义：

全有或全无　一个事务或者成功完成，使其操作的所有效果都记录到相关对象中；或者由于故障或有意取消等原因而不留下任何效果。这种全有或全无本身又包含两层含义：

- 故障原子性　即服务器崩溃时事务的效果是原子的。
- 持久性　一旦事务成功完成，它的所有效果将都被保存到持久存储中。这里的“持久存储”指的是磁盘或其他永久介质中的文件。文件中存放的数据不受服务器崩溃影响。 470
- 隔离性　每个事务的执行不受其他事务的影响。换言之，事务在执行过程中的中间效果对其他事务是不可见的。

ACID特性　Härder和Reuter[1983]建议用“ACID”记住事务的属性。

原子性（Atomicity）：事务必须是全有或全无。

一致性（Consistency）：事务将系统从一个一致状态转换到另一个一致状态。

隔离性（Isolation）。

持久性（Durability）。

在我们的事务属性列表中没有包括“一致性”，因为它通常是服务器和客户端程序员的责任，由它们确保事务使得数据库是一致的。

作为一致性的一个例子，假设在银行的例子中，一个对象拥有所有账户余额的总计，该值被作为*branchTotal*的结果。客户或者通过使用*branchTotal*或者在每个账户上调用*getBalance*来得到所有账户余额的总计。从一致性的角度看，这两种方法应该得到相同的结果。为了维护这个一致性，*deposit*和*withdraw*操作必须更新拥有所有账户余额总计的对象。

为了支持故障原子性和持久性，对象必须是可恢复的。当服务器进程由于硬件故障或软件错误而崩溃时，所有已完成事务的更新必须保留在持久存储中。这样，当服务器被新的进程替代后，它可以利用这些更新信息来恢复对象状态。当服务器确认完成了一个客户事务时，事务中所有对对象的改变必须已经记录在永久存储中了。

支持事务的服务器必须有效地对操作进行同步以保证事务之间的隔离性。最简单的方法是串行执行事务——可以按任意次序一次一个地执行。遗憾的是，这种解决方案对有多个交互用户共享其资源的服务器而言是不可接受的。在我们的银行例子中，就需要同时允许多个银行业务员进行联机银行事务。

任何支持事务的服务器的目标都是最大化并发度。因此，如果事务的并发执行与串行执行具有相同的效果——即它们是串行等价的或可串行化的——那么可允许事务并发执行。

事务能力能加到有可恢复对象的服务器上。每个事务都由协调者创建和管理，协调者实
471 现了如图12-3中所示的*Coordinator*接口。协调者为每个事务赋予一个事务标识或TID。客户调用协调者的*openTransaction*方法来创建一个新事务——分配并返回一个事务标识或TID。当事务结束时，客户调用*closeTransaction*方法表示事务结束——该事务访问的所有可恢复对象都应该被保存。如果由于某种原因，客户需要放弃事务，那么它调用*abortTransaction*方法——事务的所有效果将被取消。

openTransaction()→ *trans*; 开始一个新事务，并返回该事务的惟一标识TID，该标识将用于事务的其他操作中 *closeTransaction*(*trans*) → (*commit*, *abort*); 结束事务：如果返回值为*commit*，表示该事务被成功提交；否则返回 *abort*，表示该事务被放弃 *abortTransaction*(*trans*); 放弃事务

图12-3 Coordinator接口的操作

事务的完成需通过一个客户程序、若干可恢复对象和一个协调者之间的合作。客户指定了组成事务的一系列针对可恢复对象的操作。为了实现这一点，客户在每次调用中需要发送由*openTransaction*返回的事务标识。一种简单的实现方式是将TID作为可恢复对象的每个调用的一个额外参数。例如，在银行服务中，*deposit*操作可能按如下定义：

deposit(*trans*, *amount*)

在TID为*trans*的事务中给账户存款*amount*

如果事务由中间件提供，那么所有介于*openTransaction*和*closeTransaction*或*abortTransaction*之间的远程调用都隐式地传递TID。这正是CORBA事务服务的做法。因此，在我们的例子中不再列出TID。

通常，事务在客户调用*closeTransaction*后结束。如果事务正常进展，那么*closeTransaction*的返回值表明事务被提交——它给客户一个承诺：所有事务所请求的更新都被永久记录。此后的其他事务访问同一数据时将见到这些更新的结果。

另一种情况是，事务由于某些原因，比如事务自身的特性、与其他事务发生冲突或者计算机或进程崩溃，而不得不放弃。一旦事务被放弃，参与方（可恢复对象和协调者）必须保证在对象和永久存储中清除所有效果，使该事务的影响对其他事务不可见。

事务或者成功执行，或者以两种方式之一被放弃——客户放弃事务（使用
472 *abortTransaction*调用）或服务器放弃事务。图12-4分别列出了事务的3个执行历史。在这几种情况中，我们都称事务执行失败。

进程崩溃时的服务器动作 如果服务器进程意外崩溃，它最终会被新服务器进程替代。新的服务器进程将放弃所有未提交事务，并使用一个恢复过程将对象的值恢复成最近提交的事务所产生的值。为了处理事务过程中客户进程的意外崩溃，服务器给每个事务都设定一个过期时间，服务器将放弃在过期时间还未完成的事务。

服务器进程崩溃时的客户动作 如果服务器在执行事务期间崩溃，那么客户在超时后会接收到一个异常。如果在执行事务期间，服务器崩溃了且被新服务器进程替代，那么未完成的事务将不再有效，当客户发起新操作时它会收到异常。在任何一种情况下，客户需要建立一个计划，通过人工干预等方式来完成或放弃事务所在的任务。

成功执行	被客户放弃	被服务器放弃	
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•		•
•	•	服务器放弃事务 →	•
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

图12-4 事务执行历史

12.2.1 并发控制

本节将以银行为例，说明并发事务中的两个著名问题——“更新丢失”和“不一致检索”。然后本节给出如何利用事务的串行等价执行来避免这些问题。我们假设*deposit*、*withdraw*、*getBalance*和*setBalance*都是同步操作——即，它对记录账户余额的实例变量的效果是原子的。

更新丢失问题 更新丢失问题可用银行账户*A*、*B*和*C*上的两个事务来说明。这3个账户的初始余额分别是100美元、200美元和300美元。事务*T*将资金由账户*A*转到账户*B*，事务*U*将资金由账户*C*转到账户*B*。两次转账的金额都是当前B账户余额的10%。因此，两次转账的最终效果是增加账户*B*的余额10%两次，*B*的最终值为242美元。 473

下面来看看事务*T*和事务*U*并发执行的效果，见图12-5。两个事务获得账户*B*的余额200美元，然后存入20美元。结果是将账户*B*的余额提高了20美元，而不是42美元，这是不正确的。这就是所谓的“更新丢失”问题。事务*U*的更新被丢失是因为事务*T*覆盖了它的更新。两个事务在写入新数据前读出的都是旧数据。

在图12-5的后半部分，我们列出了对相应账户余额有影响的操作，其中，在某行之上的行之后执行该行上的操作。

事务*T*：		事务*U*：	
balance = b.getBalance();		*balance = b.getBalance();*	
*b.setBalance(balance*1.1);*		*b.setBalance(balance*1.1);*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance();	200美元		
		balance = b.getBalance();	200美元
		*b.setBalance(balance*1.1);*	220美元
*b.setBalance(balance*1.1);*	220美元		
a.withdraw(balance/10)	80美元		
		c.withdraw(balance/10)	280美元

图12-5 更新丢失问题

不一致检索 图12-6列出了另一个与银行账户有关的例子：事务*V*将资金由账户*A*转到账

户*B*，事务*W*调用*branchTotal*方法获得银行所有账户的总余额。账户*A*和*B*的最初余额都是200美元，但是*branchTotal*计算*A*和*B*的总和，结果却是300美元，这是错误的数据值。这就是“不一致检索”问题。事务*W*的检索是不一致的，因为在*W*计算总和的时候，*V*已经完成了转账操作中的取款部分。

事务*V*：		事务*W*：	
a.withdraw(100)		*aBranch.branchTotal()*	
b.deposit(100)			
a.withdraw(100);	100美元		
		total = a.getBalance()	100美元
		total = total + b.getBalance()	300美元
		total = total + c.getBalance()	
b.deposit(100)	300美元	•	
		•	

图12-6 不一致检索问题

串行等价性 如果每个事务知道它单独执行的正确效果，那么我们可以推断出这些事务按某种次序一次执行一个事务的组合效果也是正确的。如果并发事务交错执行操作的效果等同于按某种次序一次执行一个事务的效果，那么这种交错执行是一种串行等价的交错执行。我们说两个事务具有相同效果，是指读操作返回相同的值，并且事务结束时，所有对象的实例变量也具有相同的值。

使用串行等价性作为标准来判断正确的并发执行，可以防止更新丢失和不一致检索问题的出现。

在两个事务都读取了一个变量的旧数据，并用它来计算新数据时，会出现更新丢失问题。如果两个事务一前一后执行，就不会发生这个问题，因为后执行的事务将读取到前面执行的事务更新后的数据。由于两个事务进行串行等价的交错执行能够产生与串行执行同样的效果，所以通过串行等价，我们能够解决丢失更新问题。图12-7列出了这样的一种交错执行，这时影响共享账户*B*的操作实际上是串行的，因为事务*T*在事务*U*之前完成了所有对*B*的操作。另一种具有该性质的*T*和*U*的交错执行是事务*U*在事务*T*之前完成它对账户*B*的操作。

现在我们在事务*V*将资金从账户*A*转账到*B*而事务*W*正在获取所有余额总和（见图12-6）的情况下，考虑与不一致检索有关的串行等价性的效果。不一致检索出现在某个检索事务与一个更新事务并发运行的时候。如果检索事务在更新事务之前或之后执行，问题就不会发生。一个检索事务和一个更新事务进行串行等价地交错执行（如图12-8中所示的例子），可以防止不一致检索的发生。

冲突的操作 如果两个操作的执行效果和它们的执行次序相关，我们称这两个操作相互冲突。为简化讨论，我们考虑一对操作*read*和*write*。*Read*读取对象值，而*write*更新对象值。一个操作的效果，是指由*write*操作设置的对象值和由*read*操作返回的结果。图12-9给出了*read*和*write*操作的冲突规则。

对任意的两个事务，可以确定它们之间冲突操作的访问次序。那么，串行等价性可以用如下冲突操作来定义：

事务T：		事务U：	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	200美元		
*b.setBalance(balance*1.1)*	220美元		
		balance = b.getBalance()	220美元
		*b.setBalance(balance*1.1)*	242美元
a.withdraw(balance/10)	80美元		
		c.withdraw(balance/10)	278美元

图12-7 串行等价地交错执行事务*T*和*U*

事务V：		事务W：	
a.withdraw(100);		*aBranch.branchTotal( )*	
b.deposit(100)			
a.withdraw(100);	100美元		
b.deposit(100)	300美元		
		total = a.getBalance( )	100美元
		total = total + b.getBalance()	400美元
		total = total + c.getBalance()	
		...	

图12-8 串行等价地交错执行事务*V*和*W*

不同事务的操作		是否冲突	原 因
read	*read*	否	因为一对*read*操作的结果不受它们的执行顺序的影响
read	*write*	是	因为*read*和*write*操作的结果与它们的执行顺序相关
write	*write*	是	因为一对*write*操作的结果与它们的执行顺序相关

图12-9 Read和Write操作的冲突规则

两个事务串行等价的充分必要条件是，两个事务中所有的冲突操作都按相同的次序执行。 474 ~ 476

考虑下面的例子，事务*T*和事务*U*定义如下：

T: x=read(i); write(i, 10); write(j, 20);

U: y=read(j); write(j, 30); z=read(i);

图12-10列出了它们的一种交错执行过程。注意，每个事务相对于另一个事务对对象*i*和*j*的访问是串行的，因为事务*T*对变量*i*访问都在事务*U*对*i*访问之前，而*U*对变量*j*的访问都在事务*T*对*j*访问之前。但是这个执行次序不是串行等价的，因为对两个对象的冲突操作并未按照

相同次序执行。串行等价的执行次序要求下面两个条件之一：

1. 事务*T*在事务*U*之前访问*i*，并且事务*T*在事务*U*之前访问*j*。
2. 事务*U*在事务*T*之前访问*i*，并且事务*U*在事务*T*之前访问*j*。

事务*T*：	事务*U*：
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

图12-10 非串行等价地执行事务*T*和*U*的操作

串行等价性可作为一个标准用于形成并发控制协议。并发控制协议用于将访问对象的并发事务串行化。有3种常用的并发控制方法：锁、乐观并发控制和时间戳排序。然而，大多数实际系统利用锁方法（参见12.4节的讨论）。使用锁方法时，每当对象被访问之前，服务器就为该对象设置一个锁，并在该锁上标记上事务标记，当事务完成后服务器再删除这些锁。某个对象被锁住后，只有锁住该对象的事务可以访问它；而其他的事务必须等到对象被解锁，或者某些情况下共享该锁。使用锁可能会导致死锁，此时，事务相互等待其他事务释放锁。例如，有两个事务各自锁住了一个对象，而又要访问被对方锁住的对象。关于死锁及其补救方法，我们将在12.4.1节讨论。

12.5节将描述乐观并发控制。在乐观并发控制中，事务能够一直运行而不被锁住，当它请求提交事务时，服务器检测该事务是否执行了与其他并发事务相冲突的操作，一旦检测出冲突，服务器就放弃该事务并重新启动该事务。检测的目的是为了保证所有对象都是正确的。

时间戳排序将在12.6节描述。在时间戳排序中，服务器记录每个对象最近一次读写访问的时间。事务访问对象时，需要比较事务的时间戳和对象的时间戳，来决定是否允许立即访问、
477 延迟访问或拒绝访问该对象。如果决定延迟访问，那么该事务就要等待；如果决定拒绝访问，那么将放弃该事务。

在检测到操作冲突之后，主要通过让一个客户事务等待另一个客户事务或是重新运行事务或是两者的结合来实现并发控制。

12.2.2 事务放弃时的恢复

服务器必须记录所有已提交事务的效果，而不保存被放弃事务的效果。因此，服务器必须保证事务被放弃后，它的更新作用完全取消，而不影响其他并发事务。

本节以银行为例，阐述与事务放弃相关的两个问题。这两个问题是“读取脏数据”和“过早写入”，这两个问题在事务的串行等价执行中仍然出现。这两个问题与对象上的操作效果有关，如影响银行账户的余额。为简化讨论，我们将所有的操作分为读操作和写操作，在我们的例子中，*getBalance*是读操作而*setBalance*是写操作。

读取脏数据 事务的隔离特性要求未提交事务的状态对其他事务是不可见的。如果某个事务读取了另一个未提交事务写入的数据，那么这种交互会引起“读取脏数据”问题。考虑图

12-11中的事务执行情况，事务*T*读取账户*A*的余额并将其增加10美元，事务*U*也读取*A*的余额并将其增加20美元，这两个事务的执行是串行等价的。现在假设提交事务*U*之后事务*T*被放弃，由于必须将账户*A*的余额恢复到它的初始值，所以事务*U*所读取的数据是一个从不存在的值。我们称事务*U*进行了一次读取脏数据。因为它已经被提交，所以它不能被取消。

事务*T*：		事务*U*：	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	100美元		
a.setBalance(balance + 10)	110美元		
		balance = a.getBalance()	110美元
		a.setBalance(balance + 20)	130美元
		commit transaction	
abort transaction			

图12-11 事务*T*放弃时的读取脏数据

478

事务可恢复性 如果某个事务（例如*U*）访问了被放弃事务的更新结果，并且已提交，那么该事务是不可恢复的。为了确保不出现这种情况，所有进行了脏数据读取的事务必须推迟提交。可恢复的策略是推迟事务提交，直到它读取更新结果的其他事务都已提交。在我们的例子中，事务*U*必须在事务*T*提交后才能提交，如果事务*T*放弃了，那么事务*U*也必须放弃。

连锁放弃 在图12-11中，假设事务*U*直推迟提交直到事务*T*放弃，那么此时事务*U*也要放弃。遗憾的是，观察到*U*结果的其他事务同样也要放弃。这些事务的放弃可能导致更多的事务放弃。这种情况被称为连锁放弃。防止这种情况出现的方法是，只允许事务读取已提交事务写入的数据。为了保证这一点，读某对象的操作必须一直推迟直到写该对象数据的事务提交或放弃。防止连锁放弃是一个比保证事务可恢复性更强的条件。

过早写入 考虑事务放弃的另一种可能性。它牵涉两个事务针对同一个对象进行写操作的交互。如图12-12所示，账户*A*上的事务*T*和事务*U*都调用*setBalance*。事务开始前，账户*A*的余额是100美元，图中的事务执行是串行等价的，事务*T*将余额更改为105美元，事务*U*将余额更改为110美元。如果事务*U*放弃而事务*T*提交，那么余额将恢复为105美元。

事务*T*：		事务*U*：	
a.setBalance(105)		*a.setBalance(110)*	
a.setBalance(105)	100美元 105美元		
		a.setBalance(110)	110美元

图12-12 重写未提交数据

一些数据库系统在放弃事务时，将变量的值恢复到该事务所有写操作的“前映像”。在我们的例子中，*A*的初始值是100美元，它是事务*T*的写“前映像”，类似地，事务*U*的写前映像是105美元。所以，如果事务*U*放弃了，那么正确的账户余额为105美元。

现在考虑事务*U*提交而事务*T*放弃的情况。此时，余额应该是110美元，但事务*T*的写“前

映像”是100美元，所以我们最终获得了100美元的错误值。类似地，如果事务T先放弃接着U也放弃，由于U的写“前映像”是105美元，所以我们得到错误的账户余额为105美元，但是正确的数值应该是100美元。

为了保证使用前映像进行事务恢复，获得正确的结果，写操作必须等到前面修改同一对象的其他事务提交或放弃后才能进行。

479

事务的严格执行 为了避免“读取脏数据”和“过早写入”，通常要求事务推迟读操作和写操作。如果读操作和写操作都推迟到写同一对象的其他事务提交或放弃后才进行，那么这种执行被称为是严格的。事务的严格执行可以真正保证事务的隔离特性。

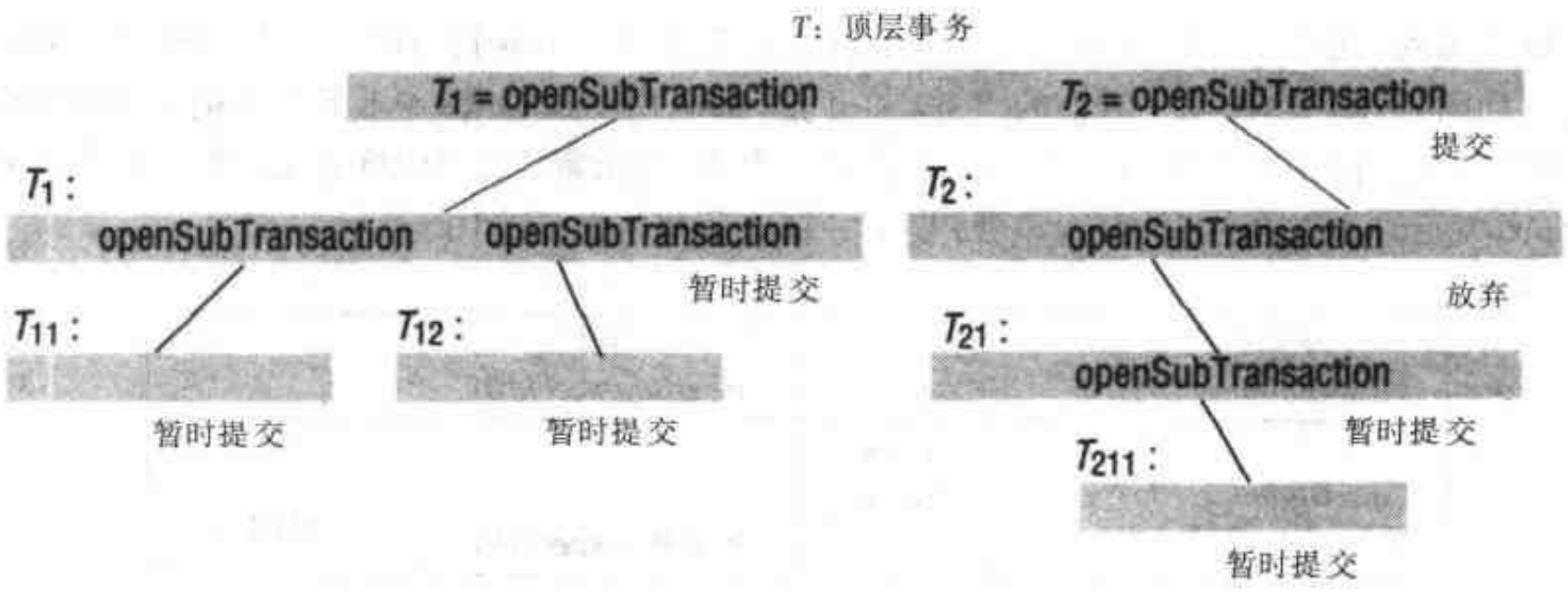
临时版本 对于参与事务的可恢复对象服务器，它必须保证事务放弃后，能清除所有对象的更新。为了达到此目的，事务中所有的更新操作都是针对对象在挥发性存储中的临时版本。每个事务都有本事务已更改的对象的临时版本集。事务的所有更新操作将值存储在自己的临时版本中，如果可能，事务的访问操作就从事务的临时版本中取值，如果取值失败，才从对象取值。

只有当事务提交时，临时版本的数据才会用来更新对象，与此同时，它们也被记录到持久存储中。这个过程是一个原子步骤，其间将暂时不让其他事务访问相关对象。如果事务放弃，系统将删除它的临时版本。

12.3 嵌套事务

嵌套事务扩展了前面介绍的事务模型，它允许事务由其他事务构成。这样，从一个事务内可以发起几个事务，从而能够将事务看成按需组成的模块。

嵌套事务的最外层事务被称为顶层事务。除顶层事务之外的其他事务被称为子事务。例如在图12-13中，事务T是一个顶层事务，它启动两个子事务T_1和T_2。子事务T_1启动它的子事务T_{11}和T_{12}，子事务T_2启动它的子事务T_{21}，T_{21}又启动子事务T_{211}。



480

图12-13 嵌套事务

就事务的并发访问和故障处理而言，子事务对它的父事务是原子的。在同一个层次的子事务（例如T_1和T_2）可以并发运行，但它们对公共对象的访问是串行化的，例如通过12.4节描述的锁机制。每一个子事务可能独立于父事务和其他子事务而出现故障。当某个子事务放弃时，其父事务有时可能选择另一个子事务来完成它的工作。例如，某个事务需要将一个邮件

消息发送给一个列表中的所有接收者，该事务可以由一系列子事务组成，每个子事务负责将消息发送给一个接收者。如果某些子事务执行失败，父事务可记录这些信息，然后提交整个事务，结果是提交所有成功的子事务。然后，可以启动另一个事务来重新发送第一次未发出的那些消息。

为了以示区别，我们称前文介绍的事务为平面事务。平面事务的所有工作都在*openTransaction*和*commit/abort*之间的同一个层次里完成，它不可能提交或放弃部分事务。嵌套事务有下列主要的优势：

1. 在同一个层次的子事务可以并发运行，这能提高事务内的并发度。如果这些子事务运行在不同的服务器上，那么它们将能够并行执行。例如，考虑银行例子中的*branchTotal*操作，它的实现可以通过在分行的每一个账户上调用*getBalance*完成。现在每次*getBalance*调用可以用一个子事务实现，这些子事务可并发执行。由于这些操作应用于不同的账户，所以子事务之间不存在冲突的操作。
2. 子事务可以独立提交和放弃。与单个事务相比，若干嵌套的子事务可能更强壮。前面的发送邮件的例子可以表明这一点——如果利用平面事务，一个事务失败会导致整个事务重启。事实上，父事务可以根据子事务是否放弃来决定采用不同的动作。

嵌套事务的提交规则相当细致：

- 事务在它的子事务完成以后，才能提交或放弃。
- 当一个子事务执行完毕后，它可以独立决定是暂时提交还是放弃。如果决定放弃，那么这个决定是最终的。
- 父事务放弃时，所有的子事务都被放弃。例如，如果T_2放弃了，那么子事务T_{21}和T_{211}也必须放弃，即使它们可能已经暂时提交了。
- 如果某个子事务放弃了，那么父事务可以决定是否放弃。在我们的例子中，虽然T_2放弃了，但T可决定提交。 481
- 如果顶层事务提交，那么所有暂时提交的子事务将最终提交（这里假设它们的祖先没有放弃）。在我们的例子中，事务T的提交将允许事务T_{11}和T_{12}提交，但T_{21}和T_{211}不能提交，因为它们的父事务T_2已放弃了。需要注意的是，只有当顶层事务提交后，子事务的作用才能持久化。

某些情况下，由于一个或多个子事务被放弃，顶层事务最终选择放弃。例如，考虑下面的事务*Transfer*：

从*B*转账100美元到*A*

a.deposit(100)

b.withdraw(100)

事务*Transfer*可以组织成两个子事务：一个执行*withdraw*操作，另一个执行*deposit*操作。如果两个子事务都成功提交，那么*Transfer*事务也能够提交。假设遇到账户透支，*withdraw*子事务将放弃。现在考虑*withdraw*子事务放弃，而*deposit*子事务提交——回想一下子事务的提交将视父事务提交而定，我们假设顶层（*Transfer*）事务选择放弃，父事务的放弃将导致子事务放弃，所以*deposit*事务放弃，其效果被消除。

CORBA的对象事务服务同时支持平面事务和嵌套事务。在分布式系统中，由于子事务可以在不同服务器上并发执行，所以嵌套事务显得尤其重要。我们将在第14章讨论这个问题。

嵌套事务的这种形式是由Moss提出的[Moss 1985]。嵌套事务有很多变种，这些变种具有不同的串行特性，可参见Weikum[1991]。

12.4 锁

事务必须通过调度使它们对共享数据的执行效果是串行等价的。服务器可以通过串行化对象访问来达到事务的串行等价。图12-7的例子表明如何在并发的情况下达到串行等价——事务*T*和事务*U*都访问账户*B*，但事务*T*在*U*开始访问前就完成了它的访问。

一种简单的串行化机制是使用互斥锁。在这种锁机制下，服务器试图给客户事务操作所访问的对象上锁。如果客户请求要访问的一个对象已被其他事务锁住，那么服务器将暂时挂起这个请求，直到对象被解锁。

图12-14说明了互斥锁的使用。它给出的事务与图12-7中的相同，但多出一列用于为每个事务列出加锁、等待和解锁等动作。这个例子假设在事务*T*和*U*运行前，账户*A*、*B*和*C*均未加
482 锁。当事务*T*准备访问账户*B*时，账户*B*被事务*T*锁住。此后，当事务*U*准备访问*B*时，由于*B*被*T*锁住，所以*U*必须等待。事务*T*提交时，*B*被解锁，此时事务*U*继续执行。在*B*上使用锁有效地串行化了对*B*的访问。需要注意的是，如果事务*T*在*getBalance*和*setBalance*之间释放*B*的锁，那么事务*U*对*B*的*getBalance*操作就能穿插在*T*的操作之间。

事务 T:		事务 U:	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(bal*1.1)*		*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)		*c.withdraw(bal/10)*	
操作	锁	操作	锁
openTransaction			
bal = b.getBalance()	锁住*B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	锁住*A*	*bal = b.getBalance()*	等待事务*T*在*B*上的锁释放
closeTransaction	解锁*A*, *B*	•••	
			锁住*B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	锁住*C*
		closeTransaction	对*B*, *C*解锁

图12-14 事务*T*和*U*使用互斥锁

串行等价性要求一个事务对一个对象的所有访问相对于其他事务进行的访问是串行化的。两个事务的所有的冲突操作对必须以相同的次序执行。为了保证这一点，事务在释放任何一个锁之后，都不允许再申请新的锁。每个事务的头一个阶段是一个“增长”阶段，在这个阶段中，事务不断地获取新的锁；在第二个阶段中，事务释放它的锁（一个“收缩阶段”）。这被称为两阶段加锁。

12.2.2节介绍了事务的放弃可能引起读取脏数据和过早写入问题，需要用严格执行来防止这些问题。在事务的严格执行中，事务对某个对象的读写必须等到其他写同一对象的事务提

交或放弃之后。为了保证这一点，所有在事务执行过程中获取的锁必须在事务提交或放弃后才能释放。这被称为*严格的两阶段加锁*。锁可以阻止其他事务读或写对象。在事务提交时，为了保证可恢复性，锁必须在所有被更新的对象写入持久存储之后才能释放。

服务器通常包含大量的对象，而一个事务只访问很有限的一些对象，不太可能与其他事务冲突。并发控制使用的粒度是一个重要问题，因为如果并发控制（例如，锁）只能同时应用到所有对象上，那么服务器中对象的并发访问范围将会严重受限。在我们的银行例子中，如果一次将分行中的所有账户都锁住，那么在任何时候，只有一个业务员能够进行联机事务——这是不可接受的限制。 483

对其访问必须被串行化的那部分对象应尽可能少；也就是说，尽量限制为仅仅是事务请求的每个操作所涉及的对象。在银行例子中，分行包含了众多账户，每个账户都有一个余额。每次银行业务操作会影响一个或多个账户余额——*deposit*操作和*withdraw*操作影响一个账户余额，而*branchTotal*影响所有账户余额。

下面介绍的并发控制机制不假定任何特定的粒度。我们讨论可应用于对象的并发控制协议，其中对象的操作可以抽象成对象上的*read*和*write*操作。为了保证协议能够正常工作，每个*read*和*write*操作在对象上的效果必须是原子性的。

并发控制协议用于解决不同事务中的操作访问同一个对象时的冲突。本章使用操作之间的冲突来解释协议。图12-9给出了*read*操作和*write*操作的冲突规则，其中不同事务对同一个对象的*read*操作是不冲突的。因此，对*read*和*write*操作都使用简单的互斥锁会过多地降低并发度。

可以采用这样一种锁机制，它能够支持多个并发事务同时读取某个对象，或者允许一个事务写对象，但它不允许两者同时存在。这通常被称为“多个读者/一个写者”机制。该机制使用两种锁：读锁和写锁。在事务进行读操作之前，在对象上加读锁。在事务进行写操作之前，在对象上加写锁。如果不能设置相应的锁，那么事务（和客户）必须等待，直到可以设置相应的锁——从不拒绝客户的请求。

由于不同事务的读操作不冲突，所以在已有读锁的对象上设置读锁总是成功的。所有访问同一对象的事务共享其读锁——正是这个原因，读锁有时也被称为*共享锁*。

操作冲突规则表明：

1. 如果事务*T*已经针对某个对象进行了读操作，那么并发事务*U*在事务*T*提交或放弃前不能写该对象。
2. 如果事务*T*已经针对某个对象进行了写操作，那么并发事务*U*在事务*T*提交或放弃前不能写或读该对象。

为了保证规则1，如果一个对象上有另一个事务的读锁，那么对该对象的写锁请求将被延迟。为了保证规则2，如果一个对象上有另一个事务的写锁，那么对该对象的读锁或写锁请求将被延迟。 484

图12-15给出了任一对象上读锁和写锁的相容性。表中的第一列是对象上已设置的锁类型，第一行是请求的锁类型。每个单元项分别指明，当对象在另一个事务中被左边类型的锁锁住时，一个事务请求读锁或写锁的结果。

不一致检索和更新丢失是在没有并发控制机制（如锁）保护的情况下，由于一个事务的读操作和另一个事务的写操作之间的冲突而引起的。通过在更新事务之前或之后运行检索事

务，可以避免不一致检索问题。如果检索事务先执行，那么其读锁将推迟更新事务的执行；如果检索事务后执行，那么其读锁请求将被挂起，这会延迟检索事务的执行，直到更新事务完成为止。

对某一对象		被请求的锁	
		读	写
已设置的锁	无	OK	OK
	读	OK	等待
	写	等待	等待

图12-15 锁的相容性

更新丢失出现在两个事务同时读取了对象，然后利用读取的数据来计算新值。通过让后面的事务推迟它们的读操作直到前面的事务完成为止，可以避免更新丢失问题。它的实现方式是：每个事务在读对象时都设置一个读锁，然后在写同一对象时将读锁提升为写锁——此时，当后继事务要求一个读锁时，该请求将被延迟直到当前事务完成工作为止。

如果一个事务的读锁被多个事务共享，那么该事务不能将读锁提升为写锁，因为它可能会与其他事务拥有的读锁相冲突。这样，该事务必须请求一个写锁并等待其他读锁被释放。

锁的提升是指将某个锁转化为更强的锁——即更具有互斥性的锁。锁的相容性列表给出了锁的互斥性强弱。读锁允许其他读锁，但是写锁不允许其他读锁。两者都不允许其他写锁。因此写锁比读锁更具有互斥性。锁可以被提升，因为结果是一个更加互斥的锁。但是在事务提交前降低一个事务的锁却是不安全的，因为结果是一个更宽容的锁，它可能允许执行与串行等价不一致的其他事务。

图12-16总结了在严格的两阶段加锁实现中锁的使用规则。为了保证规则，客户不允许直接调用加锁和解锁操作。在*read*和*write*操作的请求将被应用到可恢复对象上时，执行加锁，
485 而解锁是由事务协调者的*commit*或*abort*操作完成。

1. 在某个事务中有一个操作访问某个对象时：
   (a) 如果该对象未被加锁，那么它被加上锁并且操作继续执行。
   (b) 如果该对象已被其他事务设置了一个冲突的锁，那么该事务必须等待直到对象被解锁。
   (c) 如果该对象被其他事务设置了一个不冲突的锁，那么这个锁被共享并且操作继续执行。
   (d) 如果该对象已被同一事务锁住，那么必要时提升该锁，并且操作继续执行(当一个冲突的锁阻止了锁的提升，那么使用规则(b))。
2. 当事务被提交或被放弃时，服务器将释放该事务在对象上所加的所有锁。

图12-16 在严格的两阶段加锁中使用锁

例如，CORBA的并发控制服务[OMG 1997a]既可以用于事务的并发控制，也可以在不使用事务时直接用来保护对象。该服务为资源（例如可恢复对象）提供一个锁集合（*lockset*）。这个锁集合支持获取和释放锁。锁集合的*lock*方法用来获取锁，如果这个锁暂时不能获取时，调用者将被阻塞。锁集合提供的其他方法还可用来提升和释放锁。事务性的锁集合所支持的方法与锁集合一致，但要求将事务标识作为参数。我们在前面提到CORBA的事务服务将所有在同一个事务中的客户请求都标记上同一个事务标识。这就允许可恢复对象在访问之前必须加上合适的锁。当事务提交或放弃时，事务协调者负责释放所有的锁。

由于锁一旦被获取就一直保持到事务提交或放弃，所以图12-16中的规则保证了事务执行的严格性。然而，没必要为确保严格性而保持读锁，读锁只需保持到提交请求或放弃请求到达为止。

锁的实现 锁的授予通常由服务器上的一个对象实现，我们称该对象为锁管理器。锁管理器把所拥有的锁存放在诸如散列表之类的数据结构中。每个锁都是*Lock*类的一个实例，并与特定对象相关联。图12-17给出了*Lock*类。*Lock*类的每个实例在它的实例变量中维护以下信息：

- 被锁住对象的标识。
- 当前拥有该锁的事务的事务标识（共享锁能有若干拥有者）。
- 锁的类型。

486

*Lock*的方法都是同步方法，这样试图获得或释放锁的线程将不会相互干扰。另外，当试图获取正被使用的锁时，线程将调用*wait*方法等待该锁释放。

```
public class Lock {
    private Object object;        // the object being protected by the lock
    private Vector holders;       // the TIDs of current holders
    private LockType lockType;  // the current type

    public synchronized void acquire(TransID trans,   LockType aLockType ){
        while(/*another transaction holds the lock in conflicing mode*/) {
            try {
                wait();
            }catch ( InterruptedException e){/*... */ }
        }
        if(holders.isEmpty()) { // no TIDs  hold lock
            holders.addElement(trans);
            lockType  = aLockType;
        } else if(/*another transaction holds the lock, share it*/ ) ){
            if(/* this transaction not a holder*/) holders.addElement(trans);
        } else if (/* this transaction is a holder but needs a more exclusive lock*/)
                lockType.promote();
        }
    }

    public synchronized void release(TransID trans ){
        holders.removeElement(trans);    // remove this holder
        // set locktype to none
        notifyAll();
    }
}
```

图12-17 *Lock*类

*acquire*方法实现了图12-15和图12-16给出的规则。它的两个参数分别是事务标识和该事务请求的锁类型。它首先测试能否满足该请求，如果另一个事务以相冲突的模式拥有锁，那么它调用*wait*，将调用者线程挂起直到接收到相应的*notify*。注意，*wait*调用被放在一个*while*循环中，这是因为当多个等待线程被通知时，并非所有的线程都可以继续执行。当条件最终

被满足后，该方法的剩余部分将进行锁的设置：

- 如果没有其他事务拥有该锁，将当前事务设为锁的拥有者并设置相应的锁类型。
- 否则如果有其他的事务拥有该锁，那么将当前事务设为该锁的共享拥有者（除非它已是一个拥有者）。
- 否则如果该事务本身就是锁的拥有者，而它正在请求更互斥的锁，那么提升当前锁。

487

*release*方法的参数是需要释放锁的事务的标识。该方法从锁的拥有者中删除该事务标识，将锁的类型设置为*none*并且调用*notifyAll*。倘若有多个事务正在等待获得读锁，那么该方法就会通知所有等待的线程，使得它们能够继续执行。

图12-18给出了*LockManager*类。所有的事务要求加锁和解锁的请求都被送往类*LockManager*的某个实例。

```
public class LockManager {
   private Hashtable theLocks;

   public  void setLock(Object object, TransID trans,  LockType lockType){
     Lock foundLock;
     synchronized(this){
          // find the lock associated with object
          // if there isn't one, create it and add to the hashtable
       }
     foundLock.acquire(trans, lockType);
   }

  // synchronize this one because we want to remove all entries
   public synchronized void unLock(TransID trans) {
     Enumeration e = theLocks.elements();
     while(e.hasMoreElements()){
        Lock aLock = (Lock)(e.nextElement());
        if(/* trans is a holder of this lock*/ ) aLock.release(trans);
     }
   }
}
```

图12-18 *LockManager*类

- *setLock*方法的参数指定了给定事务要锁住的对象和锁类型。它在散列表中寻找该对象相应的锁，如果没有则创建一个新锁，然后调用该锁的*acquire*方法。
- *unLock*方法的参数指定了释放锁的事务，它在散列表中找出该事务拥有的所有锁，分别调用锁的*release*方法。

488

一些策略问题　我们注意到，当若干线程等待同一个被锁住项时，*wait*方法的语义将保证每个事务都会被处理。在上面的程序中，冲突规则允许锁的拥有者可以是多个读者或一个写者。因此除非拥有者拥有写锁，请求读锁总能成功。请读者考虑下面的问题：

- 如果不断面临连续的读锁请求，那么写事务的结果会如何？有没有其他的实现方法？

当某个拥有者拥有一个写锁时，那么可能有多个读者和写者在等待。请本书读者考虑*notifyAll*的执行效果以及其他实现方法。如果读锁的拥有者试图提升被共享的锁，那么它将被阻塞。是否可以解决这个问题？

嵌套事务的加锁规则 嵌套事务的锁机制用于串行化访问对象，以便：

1. 每个嵌套事务集是一个单独的实体，它不能观察到其他嵌套事务集的部分效果。

2. 一个嵌套事务集中的每个事务不能观察到同一事务集中其他事务的部分更新效果。

实施第一个规则是要求成功子事务执行完成后，由其父事务继承子事务所获得的所有锁，随后，这些继承的锁继续被更高层的事务继承。注意这里的继承是从底层向高层传递。因此，顶层事务最终将继承嵌套事务中任何层次的成功子事务所获得的所有锁。这种方式确保了这些锁能一直保存到顶层事务提交或放弃，从而防止了不同嵌套事务集的成员观察到对方的部分效果。

以下机制用于实施第二个规则：

- 父事务不允许和子事务并发运行。如果父事务拥有某个对象上的一个锁，那么它将在子事务执行时保留该锁。这意味着，子事务在执行过程中需要临时从父事务处获取该锁。
- 同层次的子事务允许并发执行，这样，在它们访问同一个对象时，锁机制必须串行化它们的访问。

下列规则描述了锁的获取和释放：

- 如果子事务获取了某个对象的读锁，那么其他活动事务不能获取该对象的写锁，只有该子事务的父事务们可以持有该写锁。
- 如果子事务获取了某个对象的写锁，那么其他活动事务不能获取该对象的写锁或读锁，只有子事务的父事务们可以持有该写锁或读锁。
- 在子事务提交时，它的所有锁由它的父事务继承，即允许父事务保留与子事务相同模式的锁。

489

- 在子事务放弃时，它的所有锁都被丢弃。如果父事务已经保留了这些锁，那么它可以继续保持。

当同层次的子事务访问同一个对象时，子事务将轮流从父事务处获取锁，这保证了它们对公共对象访问的串行性。

例如，假设图12-13中的子事务T_1、T_2和T_{11}访问同一个对象，而顶层事务T不访问该对象。如果子事务T_1最先访问该对象并成功获取了一个锁，那么在T_{11}执行时T_1将该锁传给T_{11}，并在T_{11}结束时收回该锁。当T_1运行结束时，顶层事务T将继承该锁，并保留到整个嵌套事务结束。子事务T_2在执行时可以从T获取该锁。

12.4.1 死锁

使用锁会引起死锁。考虑图12-19中锁的使用。因为*deposit*和*withdraw*方法是原子的，所以我们在图上显示它们需要获得写锁——虽然实际上这两个方法是先读取账户余额，然后写入新余额。图12-19表示两个事务分别获取了一个账户的写锁，但在访问另一个锁住的账户时被阻塞。这就是死锁的情景——两个事务都在等待并且相互依赖对方，只有对方释放锁才能继续执行。

在客户涉及交互程序的情况下，死锁是一个常见的情形，由于交互程序中的事务通常运行时间较长，造成很多对象被锁住，从而阻止了其他客户使用这些对象。

我们注意到，在结构化对象的子项上加锁有助于避免操作冲突和可能的死锁情形。例如，日记中的某一天可以被组织成很多时间段，每个时间段可以为了更新而独立加锁。如果应用

需要给不同操作加不同粒度的锁时，层次化的加锁机制是非常有用的，参见12.4.2节。

死锁定义 死锁是一种系统状态，在该状态下一组事务中的每一个事务都在等待其他事务释放某个锁。等待图可用于表示当前事务之间的等待关系。在等待图中，结点表示事务，边表示事务之间的等待关系——如果事务*T*在等待事务*U*时释放某个锁，那么在等待图中有一条从结点*T*指向结点*U*的边。图12-20的等待图表示了图12-19中的死锁。回想一下图中的死锁是由于事务*T*和*U*都试图获取另一个事务拥有的锁造成的，因此事务*T*等待事务*U*并且事务*U*等待事务*T*。事务之间的依赖关系是间接的——通过对象上的依赖。图12-20的右图表示了事务*T*和*U*分别拥有和等待的对象。由于每个事务只能等待一个对象，因此可以把对象从等待图中删去——简化成图12-20的左图。

490

事务 *T*		事务 *U*	
操作	锁	操作	锁
a.deposit(100);	*A* 上的写锁		
		b.deposit(200)	*B* 上的写锁
b.withdraw(100)			
•••	等待事务*U*在*B*上的锁	*a.withdraw(200);*	等待事务*T*在*A*上的锁
•••		•••	
•••		•••	
		•••	

图12-19 写锁造成的死锁

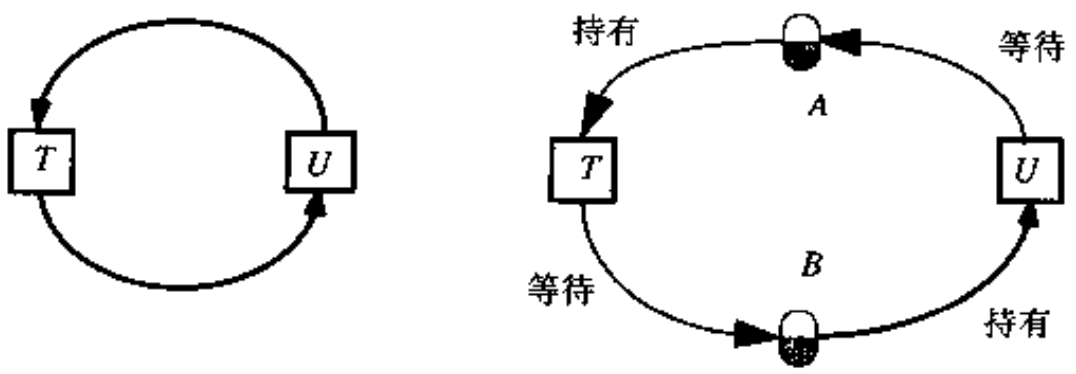


图12-20 图12-19的等待图

假设如图12-21中一样，等待图中包含环路$T \rightarrow U \rightarrow \cdots \rightarrow V \rightarrow T$，那么环路中的每一个事务都在等待下一个事务。所有的事务都被阻塞着等待锁。由于没有一个锁会释放，所以这些事务被死锁住了。如果环路中的某一个事务被放弃，那么它的锁就被释放，从而打破环路。例如，如果图12-21中事务*T*被放弃，那么它将释放事务*V*正在等待的锁——事务*V*将不再等待*T*。

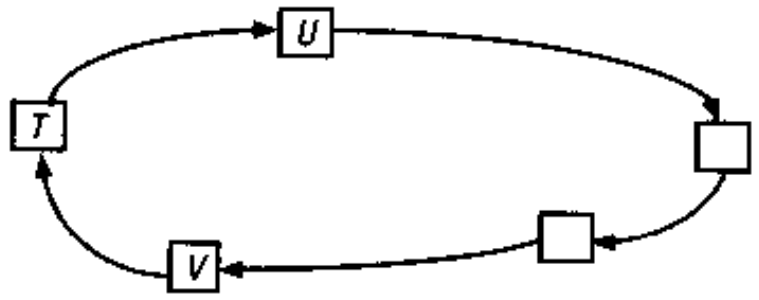


图12-21 等待图中的环路

如图12-22中的右图所示，事务*T*、*U*和*V*共享对象*C*上的读锁，事务*W*拥有对象*B*上的写锁，

而事务V正在等待对象B的锁。接着事务T和W请求对象C上的写锁，那么系统将进入死锁状态：事务T等待U和V，V等待W，而W又在等待T、U和V，如图12-22中的左图所示。这表明尽管事务一次只能等待一个对象，但是它却可能处于多个等待环路中。例如，事务V在环路$V\rightarrow W\rightarrow T\rightarrow V$和$V\rightarrow W\rightarrow V$中。 491

在这个例子中，假设事务V被放弃。这将释放对象C上的V所加的锁，V所在的两个环路均被打破。

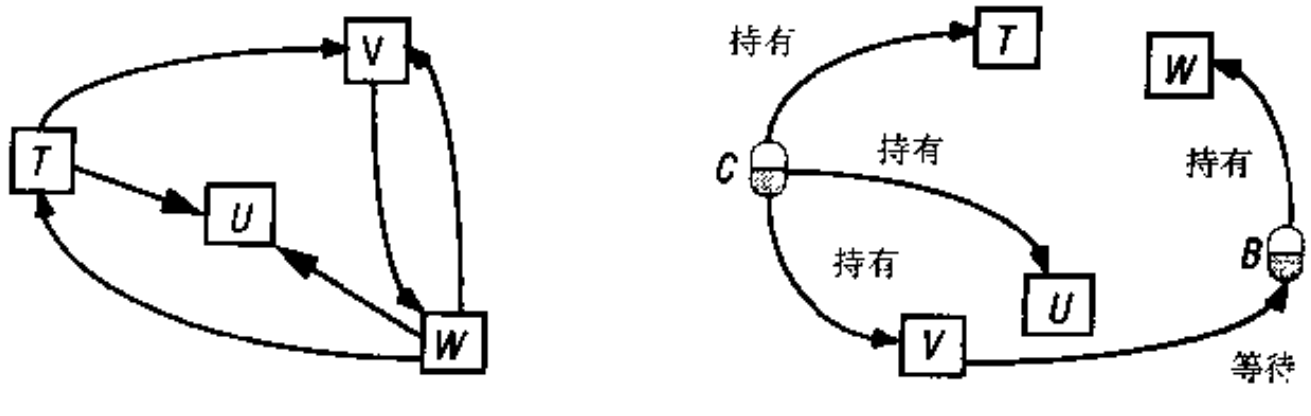


图12-22 另一个等待图

死锁预防 死锁的一个解决方案是预防发生死锁。一个简单但不是很好的克服死锁的方案是让每个事务在开始运行时就锁住它要访问的所有对象。为了避免在这一步出现死锁，这个过程必须是原子性的。该方案防止了死锁，但是却带来了不必要的资源访问限制。而且，很多情况下在事务开始时无法预计事务将访问哪些对象。在交互应用中这是常见的情形，否则用户必须事先说明准备使用哪些对象，这在浏览型应用（允许用户查找他们事先不知道的对象）中是不可想象的。死锁还可以通过以固定次序加锁来预防，但是这会造成过早加锁和减少并发度。

死锁检测 通过寻找等待图中的环路可以检测死锁。一旦检测出死锁，必须选择出一个事务，将其放弃来打破环路。

负责死锁检测的软件通常是锁管理器的一部分。它必须维护一个等待图，以便不时检测死锁。锁管理器的*setLock*和*unLock*操作将增加或删除等待图中的边。死锁检测软件在图12-22左图表示的时刻有如下信息：

事务	等待
T	U, V
V	W
W	T, U, V

492

当锁管理器遇到事务T请求事务U已锁住对象上的锁时，在等待图中增加边$T\rightarrow U$。注意如果锁被共享时，那么可能增加多条边。一旦事务U释放了T等待的锁并允许事务T继续执行时，边$T\rightarrow U$将从等待图中删去。练习12.14包含了对死锁检测实现的详细讨论。如果事务共享一个锁，那么该锁不被释放，但通向特定事务的边被删除了。

每次有新边加入等待图时，可以检测是否存在环路。为了避免不必要的开销，也可以降低检测频率。一旦检测出死锁，必须选择出环路中的一个事务并将其放弃。此时，等待图中与该事务有关的结点和边也被删除。这发生在被放弃的事务删除其锁时。

选择一个要放弃的事务不是个简单的问题。要考虑的因素有事务的运行时间以及它处于

多少环路中。

超时 锁超时是解除死锁最常用的方法。每个锁都有一个时间期限。一旦超过期限，锁将成为可剥夺的。如果没有其他事务竞争被锁住的对象，那么具有可剥夺锁的这个对象可以继续被锁住。但是，一旦有一个事务正在等待由可剥夺锁保护的对象时，这个锁将被等待事务剥夺（即对象被解锁），等待事务将继续执行。被剥夺锁的事务通常被放弃。

使用超时作为死锁的补救方法有很多问题：最坏的情况是系统中本没有死锁，但是某些事务由于它们的锁变成可剥夺的，正好其他事务在等待它们的锁，因此这些事务被放弃。在一个负载很大的系统中，超时事务的数量将增加，长时间运行的事务会被经常放弃。另外，适当的超时时间长度很难确定。相反，如果使用死锁检测，事务被放弃是因为已经出现死锁并且死锁检测已决定放弃哪一个事务。

利用锁超时，我们可以解除图12-19中的死锁，如图12-23所示。事务*T*在对象*A*上的锁在锁超时后变为可剥夺的。事务*U*正在等待获取*A*上的写锁，因此事务*T*被放弃并释放*A*上的锁，从而允许事务*U*继续执行并完成该事务。

事务*T*		事务*U*	
操作	锁	操作	锁
a.deposit(100);	给*A*加写锁		
		b.deposit(200)	给*B*加写锁
b.withdraw(100)			
•••	等待事务*U*在*B*上的锁	*a.withdraw(200);*	等待事务*T*在*A*上的锁
	（超时）	•••	
	*T*在*A*上的锁变成可剥夺的，释放*A*上的锁，放弃*T*	•••	
		a.withdraw(200);	给*A*加写锁，释放*A*、*B*上的锁

图12-23 图12-19中死锁的解除

当事务访问的对象分布在多个不同的服务器上时，可能会出现分布式死锁。在分布式死锁中，等待图可能涉及多个服务器上的对象。关于分布式死锁将在13.5节讨论。

12.4.2 在加锁机制中增加并发度

即使加锁规则建立在读操作和写操作之间的冲突上，并且所应用的锁的粒度也尽可能小，仍然有增加并发度的空间。我们将讨论两种已被使用的方法。在第一种方法（双版本加锁）中，互斥锁的设置推迟到事务提交时才进行。在第二种方法（层次锁）中，使用混合粒度的锁。

双版本锁 这是一种乐观策略：允许一个事务针对对象的临时版本进行写操作，而其他的事务读取同一对象提交后的版本。读操作只在其他事务正在提交同一个对象时才等待。这种机制比读－写锁具有更高的并发度，但是写事务在试图提交时要冒等待甚至被拒绝的风险。一个事务在有其他未完成事务正在读取对象时，不能立即提交它们对同一对象的写操作。因此在这种情况下，请求提交的事务必须等待读事务完成，在事务等待提交的时候可能发生死

锁。因此，在事务等待提交时，为了解除死锁，可能需要放弃这些事务。

这种策略用在严格的两阶段加锁上时，使用了3种锁：读锁、写锁和提交锁。在进行事务的读操作之前，必须在对象上设置读锁——除非对象上有一个提交锁，否则读锁总能成功设置，当对象上有提交锁时，事务必须等待。在进行事务的写操作之前，必须在对象上设置写锁——除非对象上有一个提交锁或写锁，否则写锁总能成功设置，当对象上有提交锁或读锁时，事务必须等待。

当事务协调者收到提交事务的请求后，它试图将事务的所有写锁转换为提交锁。如果其中某些对象上还有读锁，那么要提交的事务必须等待设置这些锁的事务完成并释放读锁。读锁、写锁和提交锁之间的相容性关系如图12-24所示。

对某个对象		要设置的锁		
		read	*write*	*commit*
已设置的锁	*none*	OK	OK	OK
	read	OK	OK	等待
	write	OK	等待	–
	commit	等待	等待	–

图12-24 锁的相容性（读锁、写锁和提交锁）

在性能方面，双版本锁和普通的读-写锁机制有两个主要不同。一方面，在双版本加锁机制中，读操作只在其他事务提交时（而不是事务的整个执行过程）才会延迟——在大多数情况下，提交协议只占整个事务的执行时间中很少的一部分时间。另一方面，某个事务的读操作可能推迟其他事务的提交。

层次锁 对某些应用，适合一个操作的锁粒度不一定适合另一个操作。在我们的银行例子中，大多数操作要求在账户粒度上加锁。但是*branchTotal*操作不同——它读取所有账户的余额值，因此似乎要在这些账户上都加上读锁。为了减少加锁开销，应当允许混合粒度的锁同时存在。

Gray[1978]提出使用具有不同粒度的层次锁。在每一层，设置父辈锁与设置等价的子辈锁具有相同的效果。这样可以有效减少需要设置的锁数量。在我们的银行例子中，支行是父辈结点，而账户是子辈结点（如图12-25所示）。

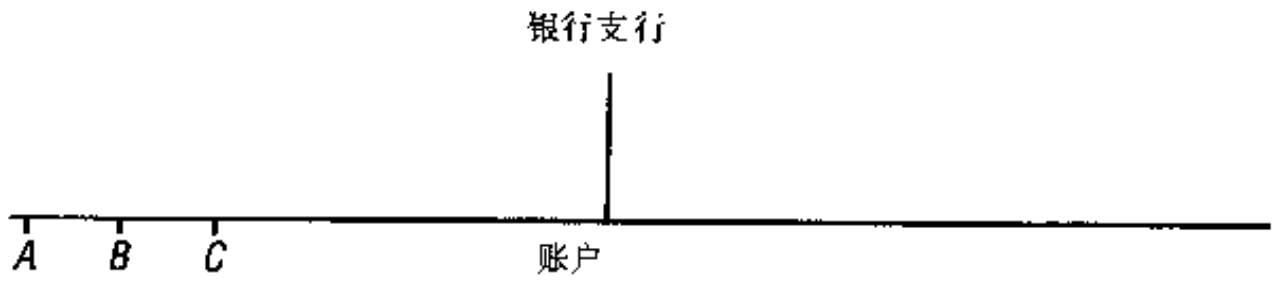


图12-25 银行例子的锁层次

混合粒度的锁在日记系统中很有用，这里的数据按照一周7天分成7部分，而每天的数据又可以继续按时间段进行细分，如图12-26所示。查看一周情况的操作需要在整个层次的最顶层加锁，而输入约会的操作只需要在某个时间段上加写锁。加在星期上的读锁会阻塞任一子结构（例如那星期每一天的所有时间段上）的写操作。

图12-26 日记的锁层次

在Gray的机制中，层次中的每个结点都可被加锁——此后，锁的拥有者能显式访问该结点和隐式访问其子结点。在图12-25的例子中，对支行的读/写锁也隐含地对所有账户加上了读/写锁。在子结点加上读/写锁时，需要在它的父结点和祖先结点（如果有）设置一个读/写试图锁。按照常规，这个试图锁与其他类型的试图锁是相容的，但是与读/写锁冲突。图12-27给出了层次锁的相容性表。Gray还提出了第三种类型的试图锁——该锁组合了读锁和写试图锁的性质。，

对某个对象		要设置的锁			
		read	*write*	*I-read*	*I-write*
已设置的锁	*none*	OK	OK	OK	OK
	read	OK	等待	OK	等待
	write	等待	等待	等待	等待
	I-read	OK	等待	OK	OK
	I-write	等待	等待	OK	OK

图12-27 层次锁的锁相容性表

在我们的银行例子中，*branchTotal*操作请求在支行上加上读锁，即隐含地对所有账户加上了读锁。*deposit*操作需要在余额上设置写锁，但是它首先试图在支行上加上写试图锁。图12-27中的规则可以防止这两个操作并发运行。

当需要混合粒度的锁时，层次锁具有减少锁数量的优势。但是它的相容性表和锁提升规则更加复杂。

493 ~ 496

混合粒度的锁支持每个事务按其需要锁住部分数据。一个访问大量对象的长事务可能需要锁住整个系统，但是一个短事务只需锁住细粒度的数据。

12.5 乐观并发控制

Kung和Robinson[1981]指出了锁机制存在的许多固有的不足，提出了另一种串行化事务的乐观方法来避免锁机制的缺点。我们将加锁的缺点总结如下：

- 锁的维护带来了新的开销，这在不支持共享数据并发访问的系统中是没有的。即使是只读事务（查询），本身不可能改变数据的完整性，通常必须利用锁来保证数据在读取时不会被其他事务修改。但是锁只在最坏的情况下起作用。

  例如，有两个并发执行的客户进程将n个对象的值增加。如果这两个客户程序同时开始执行并运行相同的时间，但它们访问对象的次序不相关，使用独立的事务来访问并

增加对象的值，那么这两个程序同时访问同一个对象的概率只有1/*n*，因此每*n*个事务中只有1个真正需要加锁。

- 使用锁会引起死锁。预防死锁会严重降低并发度，因此必须利用超时或者死锁检测来解除死锁，但这两种死锁解除方法对交互程序来说都不理想。
- 为了避免连锁放弃，锁必须保留到事务结束才能释放。这会显著地降低潜在并发度。

Kung和Robinson提出的方法是一个“乐观”策略，这是因为他们发现这样一个现象，即在大多数应用中，两个客户事务访问同一个对象的可能性是很低的。事务总是允许执行，就好像事务之间不存在冲突。当事务完成其任务并发出*closeTransaction*请求时，再检测是否有冲突。如果确实存在冲突，那么一些事务将被放弃，并需要客户重新启动该事务。每个事务分成下面几个阶段：

- 工作阶段　在事务的工作阶段，每个事务拥有所有它修改的对象的临时版本。这个临时版本是对象最新提交版本的副本。利用临时版本可以允许事务或者在工作阶段或者由于与其他事务冲突不能通过验证而放弃（不产生副作用）。可以立即执行读操作——如果事务的临时版本已经存在，那么读操作访问这个临时版本；否则，访问对象的最新提交的值。写操作将对象的新值记录成临时版本（这个版本对其他事务是不可见的）。当系统中存在多个并发事务时，一个对象有可能存在多个临时版本。另外，每个事务还维护对象访问的两个记录：读集合包含了事务读的所有对象，写集合包含了事务写的对象。注意是在对象的提交版本上执行所有的读操作，因此不会出现读取脏数据。 497
- 验证阶段　在接收到*closeTransaction*请求时，验证事务，判断它在对象上的操作是否与其他事务对同一对象的操作相冲突。如果验证成功，那么该事务就允许提交，否则，必须使用某种冲突解除机制，或者放弃当前事务，或者放弃其他与当前事务冲突的事务。
- 更新阶段　当事务通过验证以后，记录在所有临时版本中的更新将持久化。只读事务可在通过验证后立即提交。写事务在对象的临时版本记录到持久存储后即可提交。

事务的验证　验证过程使用读-写冲突规则来确保某个事务的执行对其他重叠事务是串行等价的，重叠事务是指在该事务启动时还没有提交的任何事务。为了帮助完成验证过程，每个事务在进入验证阶段之前（即，在客户发出*closeTransaction*时）被赋予一个事务号。如果事务通过验证并且成功完成，那么它就保留这个事务号；如果事务未通过验证并被放弃，或者它是只读事务，那么这个事务号被释放以便重用。事务号是整数，并按照升序分配，因此事务号定义了该事务所处的时间位置——一个事务总是在小数字的事务之后完成它的工作阶段。也就是说，如果$i<j$，事务号为T_i的事务总是在事务号为T_j的事务之前。（如果在工作阶段的开始赋予事务号，那么一个事务若比另一个小数字事务之前到达工作阶段的结尾，就要在验证前一直等待前者完成。）

对事务T_v的验证测试是基于事务T_i和T_v之间的操作冲突。事务T_v对重叠事务T_i是可串行化的，那么它们的操作必须符合以下规则：

T_v	T_i	规则
write	*read*	1. T_i不能读取T_v写的对象
read	*write*	2. T_v不能读取T_i写的对象
write	*write*	3. T_i不能写T_v写的对象，并且T_v不能写T_i写的对象

与事务的工作阶段相比，验证过程和更新过程通常只需要很短的时间，因此可以采用一个简单的方法：每次只允许一个事务进行验证和更新阶段。当任何两个事务都不会在更新阶段重叠时，就满足了规则3。注意，在写操作上的这个限制和不发生读取脏数据这个事实，将产生严格执行。为了防止重叠，整个验证和更新阶段可被实现成一个临界区，使得每次只能
498 有一个客户执行。为了增加并发度，验证和更新的部分操作可以在临界区的外部实现，但是必须串行执行事务号的分配。我们注意到，在任何时候，当前的事务号就像一个伪时钟，每当事务成功结束时这个时钟就加一。

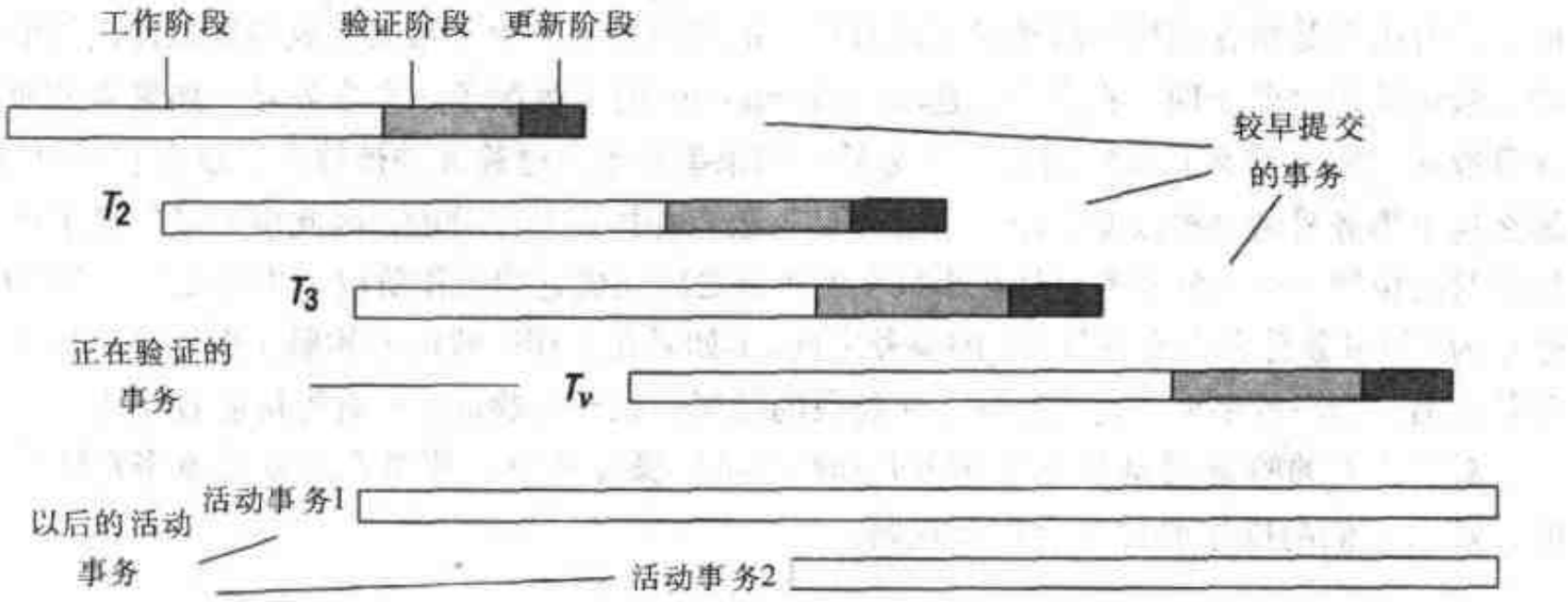
事务的验证必须保证事务T_v和T_i的对象之间的重叠遵守规则1和规则2。有两种形式的验证——向前验证和向后验证 [Härder 1984]。向后验证检查当前事务和其他较早重叠事务之间的冲突，向前验证检查当前事务和其他较晚的事务之间的冲突。

向后验证 由于较早的重叠事务的读操作在T_v验证之前进行，所以它们不会受当前事务写操作的影响（满足规则1）。T_v的验证过程将检查它的读集（受T_v的读操作影响的对象）是否和其他较早的重叠事务T_i的写集重叠，如果存在重叠，验证失败。

设*startTn*是事务T_v进入其工作阶段时系统分配（给其他一些已提交事务）的最大事务号，*finishTn*是T_v进入验证阶段时系统分配的最大事务号。下面的程序描述了T_v的验证算法：

```
boolean valid = true;
for （int Ti = startTn + 1; Ti <= finishTn; Ti++ ） {
        if （Tv 的读集与Ti的写集相交） valid = false;
}
```

图12-28给出了T_v验证过程中需要考虑的重叠事务。时间从左至右增长。T_1、T_2和T_3是较早时间提交的事务。T_1在T_v开始之前提交。T_2和T_3在T_v完成其工作阶段前提交。并且有*startTn*+1=T_2和*finishTn*=T_3。向后验证过程必须比较T_v的读集和T_2和T_3的写集。



499 图12-28 事务的验证过程

向后验证比较被验证事务的读集和已提交事务的写集，因此一旦验证失败，惟一解决冲突的方法就是放弃当前进行验证的事务。

在向后验证中，没有读操作（只有写操作）的事务无需进行验证。

向后验证的乐观并发控制要求最近提交事务中对象的已提交版本的写集合必须保留，直到没有可能发生重叠的未验证事务。每当一个事务成功通过验证，它的事务号、*startTn*和写

集合被记录在前述的事务列表中，该列表由事务服务维护。注意该列表按事务号排序。如果有长事务存在时，较早事务的写集合的保留将是一个问题。例如在图12-28中，T_1、T_2、T_3和T_v的写集合必须保留到活动事务$active_1$结束之后。值得注意的是，尽管这个活动事务有事务标识，但它还没有事务号。

向前验证 在事务T_v的向前验证中，T_v的写集合与所有重叠的活动事务的读集合进行比较——活动事务是那些处在工作阶段中的事务（规则1）。规则2自动满足，因为活动事务在T_v完成之前不会进行写操作。设活动事务具有（连续的）事务标识，从$active_1$到$active_N$，那么下面程序描述了T_v的向前验证算法：

```
boolean valid = true;
for （int T_id = active_1; T_id <= active_N; T_id ++ ） {
        if （T_v 的写集与 T_id 的读集相交） valid = false;
}
```

在图12-28中，T_v的写集合必须和事务$active_1$和$active_2$的读集合进行比较。（向前验证应该允许活动事务的读集合在验证过程和写入过程中可能改变。）由于被验证事务的读集合不包括在验证过程中，因此只读事务总能通过验证。因为与被验证事务进行比较的事务仍是活动的，所以冲突发生时，可以选择或者放弃被验证事务或者用其他方法解决冲突。Härder[1984]提出了下面几个策略：

- 推迟验证，直到冲突事务结束。但是不能保证被验证的事务在将来一定能够通过验证，在验证完成前，总是存在机会启动会产生冲突的活动事务。
- 放弃所有有冲突的活动事务，提交被验证事务。
- 放弃被验证事务。这是最简单的策略，但是由于冲突的活动事务可能在将来放弃，因此这种策略会造成被验证事务的不必要放弃。

500

向前验证和向后验证的比较 我们看到向前验证在处理冲突时有较强的灵活性，而向后验证只有一种选择——即放弃被验证的事务。通常，事务的读集合比写集合大得多。因此向后验证将较大的读集合和较早事务的写集合进行比较，而向前验证将较小的写集合和其他活动事务的读集合比较。我们注意到向后验证具有已提交事务写集合的存储开销。另一方面，向前验证不得不允许新事务在验证过程中开始。

饥饿 在一个事务被放弃后，它通常由客户程序重新启动。但是这种依赖放弃和重新运行的机制不能保证事务最终能够通过验证检查，这是因为每次重新运行后它都有可能与其他事务访问相同的对象而产生冲突。这种阻止事务最终提交的现象被称为饥饿。

可能很少出现饥饿，但是使用了乐观并发控制的服务器必须保证客户的事务不能反复放弃。Kung和Robinson认为，服务器在检测到事务已被放弃若干次后，能保证该事务不再被放弃，他们建议一旦服务器检测到这样的事务，服务器应该让该事务利用由信号量保护的临界区对服务器上的资源进行排它性访问。

12.6 时间戳排序

在基于时间戳排序的并发控制机制中，事务中的每一个操作在执行之前首先进行验证。如果该操作不能通过验证，那么将立即放弃该事务，然后客户可重新启动该事务。每个事务在启动时被赋予惟一的一个时间戳。这个时间戳定义了该事务在事务时间顺序中的次序，来

自不同事务的操作请求可以根据它们的时间戳进行全排序。基本的时间戳排序规则基于操作之间的冲突，也是非常简单的：

- 只有在对象最后一次读访问或写访问是由一个较早的事务执行的情况下，事务的对该对象的写请求是有效的。只有在对象的最后一次写访问是由一个较早的事务执行的情况下，事务的对该对象的读请求是有效的。

此规则假设系统中的每个对象只有一个版本，并且每个对象一次只能由一个事务访问。如果每个事务都有其所访问对象的临时版本，那么多个并发事务可同时访问一个对象。通过细化时间戳排序规则可以保证每个事务访问的对象版本是一致的，同时它也必须保证每个对象的临时版本按事务的时间戳所决定的顺序提交。实现这一点是通过在必要时让事务等待，以便使较早的事务完成它们的写操作。这些写操作可在*closeTransaction*操作返回之后执行，这样，客户就不用等待了。但是当读操作需要等待较早的事务完成时，客户必须等待。由于
501 事务总是等待较早的事务（在等待图中不可能形成环），因此不会引起死锁。

可以根据服务器的时钟来赋值时间戳，或者利用前面介绍的“伪时间”赋值时间戳，伪时间基于一个计数器，每次获取时间戳的请求都将计数器加一。关于在事务服务是分布的、一个事务涉及几个服务器的环境中如何生成时间戳的问题，我们将延迟到第13章讨论。

下面我们描述SDD-1[Bernstein *et al.* 1980]系统中采用的并由Ceri和Pelagatti[1985]的描述的基于时间戳的并发控制方法。

和其他方法一样，写操作被记录在对象的临时版本中并对其他事务是不可见的，直到调用了*closeTransaction*请求并提交了事务。每个对象都有一个写时间戳、若干临时版本和一个读时间戳集合，其中每个临时版本都有一个写时间戳。（已提交）对象的写时间戳比其所有临时版本都要早，并且它的所有读时间戳可以用其中的最大值来代表。每当服务器接受一个事务对某个对象的写操作时，服务器就创建该对象的一个新的临时版本，并将该临时版本的写时间戳设置为这个事务的时间戳。事务的读操作作用于时间戳为小于该事务时间戳的最大写时间戳的对象版本上。一旦事务对某个对象的读操作被接受，该事务的时间戳就被加到其读时间戳集合中。当事务被提交时，临时版本的值变成对象的值，临时版本的时间戳变成该对象的时间戳。

在时间戳排序中，需要检查事务对对象的每个读写操作请求，以查看它是否与操作冲突规则一致。当前事务T_c的请求会与其他事务T_i操作相冲突，T_i的时间戳表明它们应该比T_c晚。图12-29给出了这些规则，其中$T_i > T_c$表示T_i晚于T_c，$T_i < T_c$表示T_i早于T_c。

规则	T_c	T_i	
1.	*write*	*read*	如果$T_i > T_c$，那么T_c不能写被T_i读过的对象，这要求$T_c \geq$该对象的最大读时间戳
2.	*write*	*write*	如果$T_i > T_c$，那么T_c不能写被T_i写过的对象，这要求$T_c >$已提交对象的写时间戳
3.	*read*	*write*	如果$T_i > T_c$，那么T_c不能读被T_i写过的对象，这要求$T_c >$已提交对象的写时间戳

502 图12-29 时间戳排序中的操作冲突

时间戳排序的写规则　通过组合规则1和规则2，我们可以得到下列规则，用于决定是否可接受由事务T_c对对象D执行写操作：

if （$T_c \geqslant D$的最大读时间戳 && $T_c > D$的提交版本上的写时间戳）

　在D的临时版本上执行写操作，写时间戳置为T_c

else /* 写操作太晚了 */

　放弃事务 T_c

如果写时间戳为T_c的对象临时版本已经存在，那么写操作直接作用于此版本，否则服务器创建一个新的临时版本并且将其标记上写时间戳T_c。值得注意的是，“到来太晚的”写操作将引起事务放弃——太晚是指后来的事务已经读或写了这个对象。

图12-30说明了T_3事务的写操作的执行情况，其中T_3 ≥对象的最大读时间戳（图上没有给出读时间戳）。在情况（a）、（b）和（c）中，T_3>对象提交版本的写时间戳，因此服务器创建一个写时间戳为T_3的对象临时版本，并将其插入到按事务时间戳排序的临时版本列表中。在情况（d）中，T_3<对象提交版本的写时间戳，因此事务T_3被放弃。

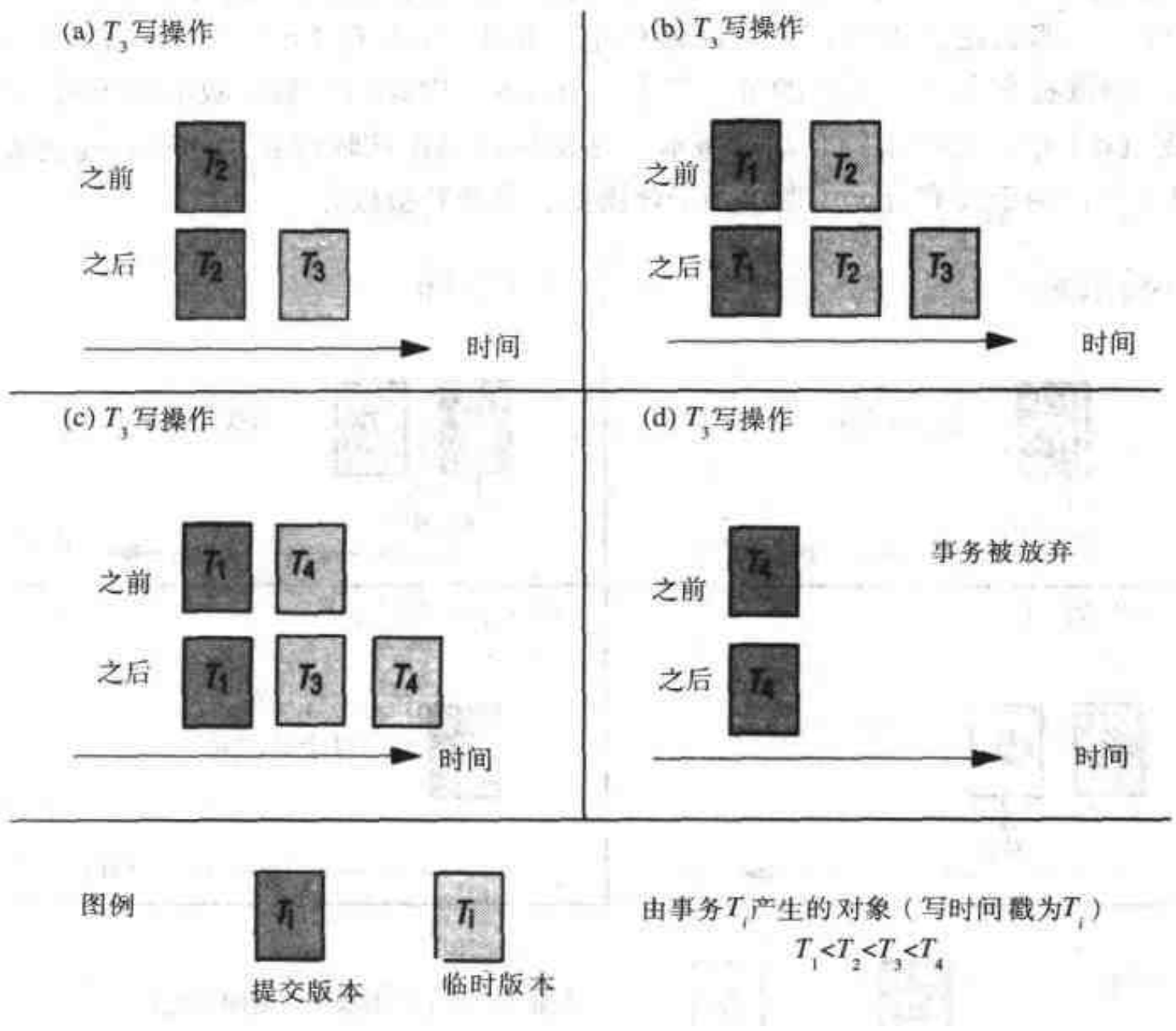


图12-30　写操作和时间戳　503

时间戳排序的读规则　应用规则3，我们可以得到下面的规则，用于决定马上接受、等待或拒绝事务T_c对对象D执行读操作：

if （$T_c > D$提交版本的写时间戳）{

　设 $D_{selected}$ 是D的具有最大写时间戳的版本 ≤ T_c

　if （$D_{selected}$已提交）

在$D_{selected}$版本上完成读操作

else

等待直到形成$D_{selected}$版本的事务提交或放弃

再应用读规则

} else

放弃事务T_c

注意以下几点：

- 如果事务T_c已经写了对象D的临时版本，那么读操作将针对这个临时版本。
- 如果读操作来得太早，那么它要等待前面的事务完毕。如果前面较早的事务提交，T_c的读操作将针对对象的提交版本。如果这个较早的事务被放弃，那么T_c将重复读规则（选择以前的版本）。此规则可防止读取脏数据。
- “到达太晚的”读操作将被放弃——太晚是指后来的事务已经写了相应的对象。

图12-31说明了时间戳排序的读规则，图中共有4种情况，标记为（a）~（d），均用于说明事务T_3的读操作动作。对每种情况，服务器选出一个写时间戳小于或等于T_3的版本。如果存在这样的版本，那么在图中用一个短线做标记。情况（a）和（b）中，读操作针对提交版本——（a）中该提交版本是对象的惟一版本，而（b）中有一个临时版本属于另一个较晚的事务。情况（c）中，读操作针对临时版本，必须等待制作该临时版本的事务提交或者放弃。在情况（d）中，由于没有合适的版本用于读操作，事务T_3被放弃。

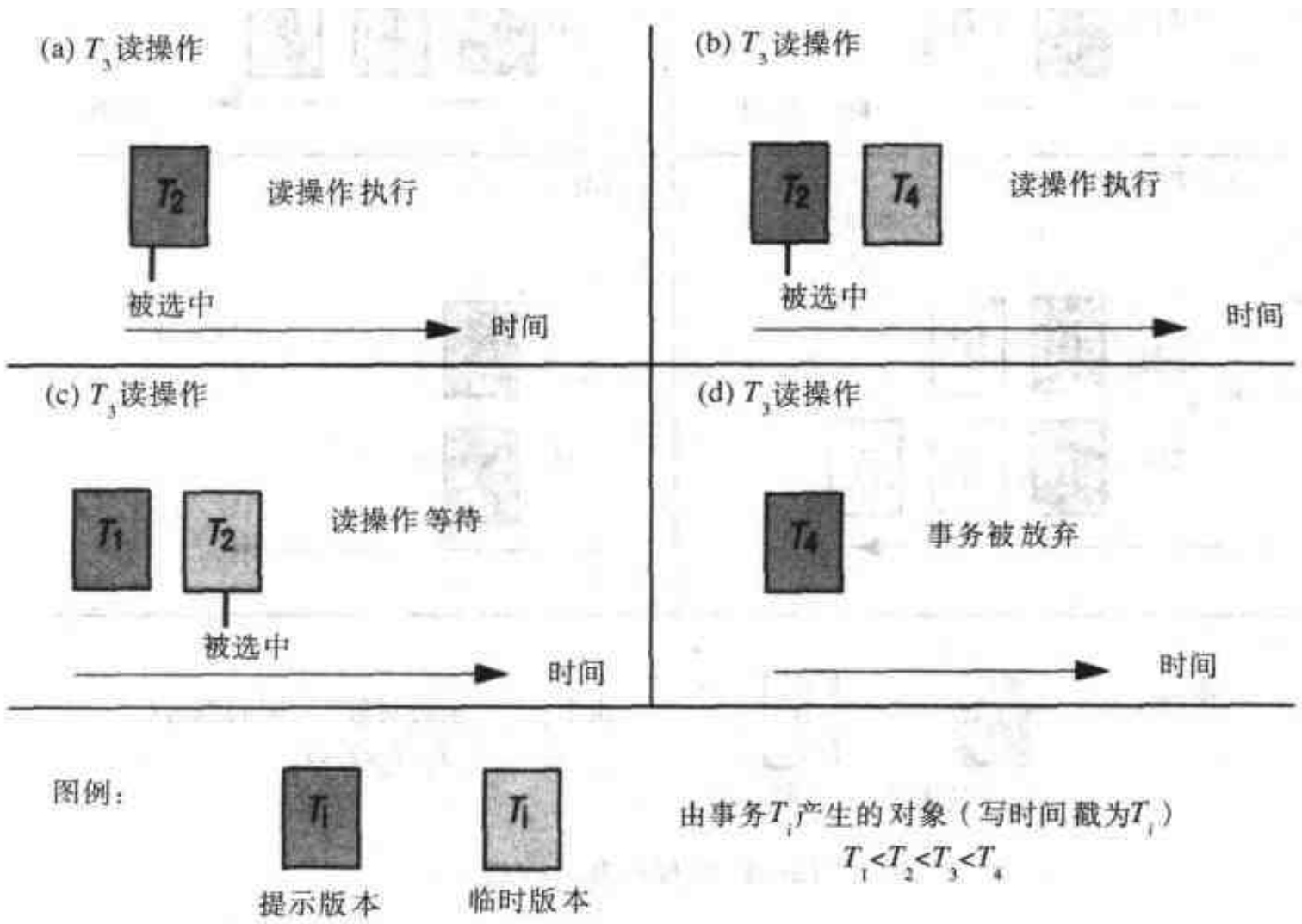


图12-31 读操作和时间戳

当一个协调者收到事务提交请求后，由于事务的所有操作在执行之前都进行了一致性检查，因此它总能提交。必须按照时间戳排序创建每个对象的提交版本。因此，协调者在写入某个事务所访问的对象的提交版本之前可能需要等待较早的事务结束，不过客户并不需要等

待。为了保证在服务器崩溃后事务是可恢复的，在确认客户的提交事务的请求之前，必须将对象的临时版本和提交信息记录到持久存储中。

需要指出的是，这里的时间戳排序算法是严格的——它保证了事务的严格执行（见12.2节）。时间戳排序的读规则要求事务对对象的读操作等待，直到所有写对象的较早事务提交或者放弃。对象的提交版本也按时间戳序排列，保证了事务对对象的写操作必须等待，直到所有写对象的较早事务提交或者放弃。

图12-32说明了利用时间戳排序方法来控制图12-7中的并发银行事务*T*和*U*。其中列*A*、*B*和*C*各表示不同的银行账户。每个账户有一项RTS，用于记录最大的读时间戳，还有一项WTS，用于记录每个版本的写时间戳——其中提交版本的时间戳用粗体表示。最初，每个账户拥有由事务*S*写入的提交版本，读时间戳集合为空。假设$S < T < U$。图中例子表示当事务*U*准备获取账户*B*的余额时，它必须等待事务*T*结束，这样它才能读取由*T*设置的值（假设*T*提交）。

T	*U*	对象的不同版本及其时间戳					
		A		*B*		*C*	
		RTS	*WTS*	*RTS*	*WTS*	*RTS*	*WTS*
		{}	***S***	{}	***S***	{}	***S***
openTransaction							
bal = b.getBalance()				{*T*}			
	openTransaction						
*b.setBalance(bal*1.1)*					***S, T***		
	bal = b.getBalance()						
	wait for T						
a.withdraw(bal/10)	•••		***S, T***				
commit	•••		***T***		***T***		
	bal = b.getBalance()			{*U*}			
	*b.setBalance(bal*1.1)*				***T, U***		
	c.withdraw(bal/10)						***S, U***

图12-32 事务*T*和*U*中的时间戳

这里介绍的时间戳方法能够避免死锁，但是它容易造成很多事务重启动。一个被称为“忽略过时写”规则的修改提供了一种改进方法。以下是它对时间戳排序的写规则做的一些改动：

- 如果写操作来得太晚，那么直接忽略该操作，而不是放弃该事务，这是因为即使它来得早一些时，它的更新效果也会被覆盖。然而，如果有其他事务已经读了该对象，那么该事务的写操作会因为读时间戳而失败。

多版本时间戳排序 本节将介绍如何通过允许每个事务写自己的对象临时版本提高基本时间戳排序的并发度。在由Reed[1983]引入的多版本时间戳排序中，每个对象除了有若干临时版本外，还有一个已提交版本列表。此列表记录了对象值的历史。利用多版本的好处在于，过迟到达的读操作不需要被拒绝。

对象的每个版本除了有一个写时间戳外，还有一个读时间戳，记录了读该版本的事务的

最大时间戳。和从前一样，每当一个写操作被接受后，它将针对事务写时间戳相应的临时版本。每当读操作执行时，它将针对小于该事务时间戳的最大写时间戳的版本，如果事务时间戳大于所使用版本的读时间戳，那么该版本的读时间戳被设置成该事务的时间戳。

当读操作到达太迟时，它可以读取一个较早的已提交版本，这样就没必要放弃读操作了。在多版本时间戳排序中，读操作总是被允许的，尽管它们有可能要等待较早的事务结束（或提交或放弃）来保证事务执行是可恢复的。练习12.22讨论了连锁放弃的可能性问题。它用于处理时间戳排序的冲突规则3。

不同事务的写操作之间不存在冲突，因为每个事务进行写操作都针对所访问对象在本事务中的已提交版本。这样时间戳排序的冲突规则2就不需要了，仅留下下面的规则：

规则1. T_c不能写事务T_i读过的对象，其中$T_i > T_c$

504～506

如果对象的某个版本的读时间戳大于T_c，那么这条规则就被破坏了，但只有在该版本有一个小于或等于T_c的写时间戳时才这样。（此写操作不能影响以后的版本。）

多版本时间戳排序的写规则　由于每个可能冲突的读操作被引导到对象最近的一个版本上，所以服务器查看小于或等于T_c的最大写时间戳的对象版本$D_{maxEarlier}$。以下规则用于事务T_c请求在对象D上执行写操作：

if （$D_{maxEarlier}$的读时间戳 $\leq T_c$）

　　在D的临时版本上完成写操作，并标记上写时间戳T_c

else

　　放弃事务T_c

图12-33说明了一个写操作被拒绝的例子。图中对象有两个写时间戳为T_1和T_2的提交版本。该对象收到以下对对象进行操作的请求序列：

T_3读；T_3写；T_5读；T_4写

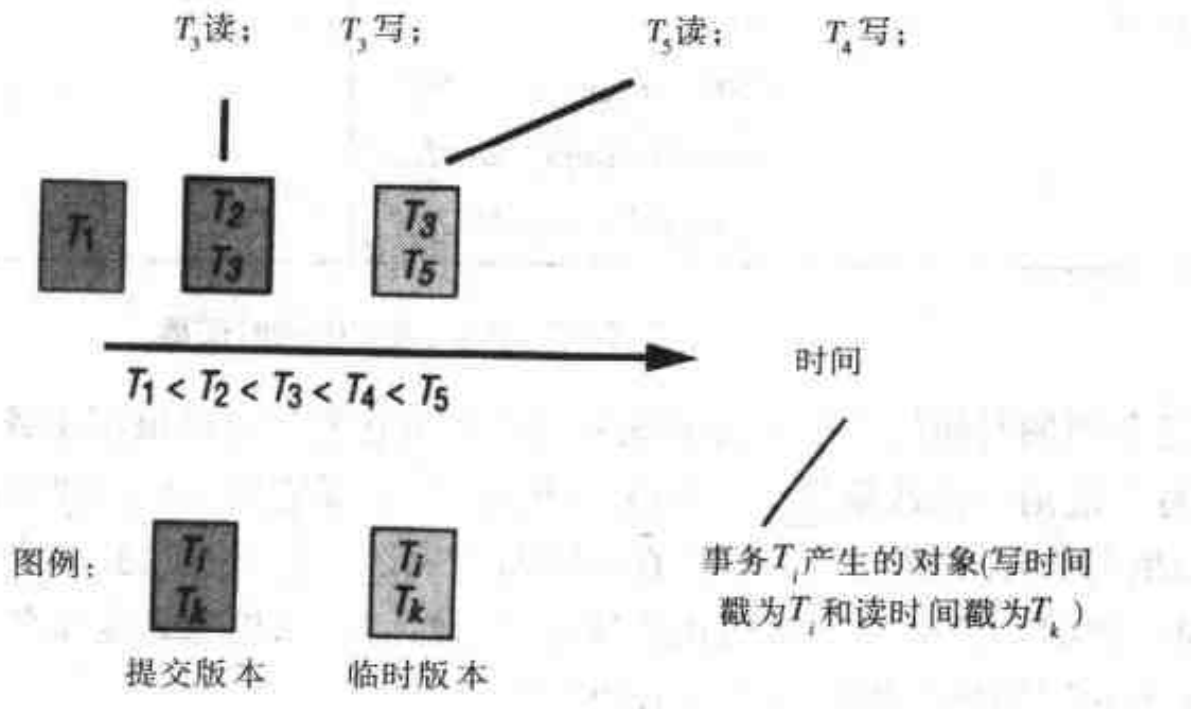


图12-33　过迟的写操作将使读操作失效

1. T_3请求一个读操作，它在T_2版本上设置读时间戳T_3。
2. T_3请求一个写操作，生成一个写时间戳为T_3的新临时版本。
3. T_5请求一个读操作，它访问写时间戳为T_3的版本（小于T_5的最高的时间戳）。

4. T_4请求一个写操作，由于写时间戳为T_3的版本的读时间戳T_5大于T_4，该写操作被拒绝。（如果该操作不被拒绝，那么新版本的写时间戳将是T_4。如果允许这个版本，那么这会和T_5的读操作相冲突，T_5的读操作应该使用时间戳为T_4的版本。）

507

当一个事务被放弃时，它创建的所有版本都被删除。当事务提交时，它创建的所有版本都被保留。但是为了控制存储空间的使用，将周期性地删除旧版本。尽管多版本时间戳排序方法会引入存储空间的开销，但既可以带来可观的并发度的提高，又不会带来死锁，而且读操作总能进行。有关多版本时间戳排序的进一步讨论，参见Bernstein等的文章[1987]。

12.7 并发控制方法的比较

我们已经描述了3种不同的控制并发访问共享数据的方法：严格的两阶段加锁、乐观方法和时间戳排序。所有方法都需要时间和空间的开销，并且它们在某种程度上都限制了并发操作的可能性。

时间戳排序方法类似于两阶段加锁，是因为它们都使用了悲观方法，即在访问每个对象时都检测事务之间是否会产生冲突。一方面，时间戳排序静态地决定事务之间的串行顺序——在事务开始时就决定它们的顺序。另一方面，两阶段加锁动态地决定事务之间的串行顺序——根据对象被访问的顺序。对只读事务来说，时间戳排序特别是多版本时间戳排序优于严格的两阶段加锁。如果事务的绝大多数操作是更新操作，那么两阶段加锁具有更好的性能。

一些研究人员根据时间戳排序对以读操作为主的事务有益，而加锁对写操作多于读操作的事务有益的这种现象，提出一种混合方案，即某些事务利用时间戳排序进行并发控制，而另一些事务利用两阶段加锁进行并发控制。对混合方法的使用有兴趣的读者可阅读Bernstein等[1987]的文献。

悲观方法在检测到对对象访问冲突时有不同的解决策略。时间戳排序将事务立即放弃，而加锁机制让事务等待——但是有可能在稍后为避免死锁而放弃该事务。

在使用乐观并发控制时，所有的事务被允许执行，但是其中一些事务在试图提交时被放弃，或在较早的向前验证中被放弃。如果并发事务之间的冲突较小时，乐观并发控制具有较好的性能，但当事务被放弃时，乐观并发控制需要重复非常多的工作。

加锁在数据库系统中已被使用多年，时间戳排序也用于SDD-1数据库系统中。两种方法都用于文件服务器中。

一些研究性的分布式系统，例如Argus[Liskov 1988]和Arjuna[Shrivastava *et al.* 1991]，研究了语义锁的使用、时间戳排序和长事务的新方法。

研究人员发现上述并发控制机制对两个应用领域的支持还不够充分。其中的一个领域是多用户应用，所有的用户都希望看到对象不断被其他用户更新的公共版本。这种应用需要数据在并发更新和服务器崩溃时能够保证原子性，事务技术看上去提供了一个解决方案。但是这些应用有两个与并发控制有关的新要求：（1）用户要求在其他用户更新数据时马上得到通知——这与隔离性是相违背的；（2）用户需要在其他用户完成其事务前就能访问对象，这需要开发新型锁，这种锁用于在对象被访问时触发动作。在这个领域的研究工作提出了不少放宽隔离性和提供更新通知的方案，这些工作的总结参见Ellis等[1991]的文献。另一个应用领域是所谓的高级数据库应用——诸如协同CAD/CAM和软件开发系统。在这些应用中，事务通常持续很长时间，用户针对对象的独立版本进行工作，这些对象版本是从一个公共数据库中检

508

出的，在工作结束时再检入，对象版本之间的合并需要用户之间的协同。有关这方面的工作总结可参见Barghouti和Kaiser [1991]的介绍。

12.8 小结

面对并发执行和服务器崩溃，事务提供了一种由客户指定原子操作序列的手段。实现原子性第一方面含义是通过运行事务使得它们的执行效果是串行等价的。已提交事务的效果被记录在持久存储中，这样，事务服务能从进程崩溃中恢复。为了实现事务放弃后消除所有的效果，事务的执行必须是严格的——也就是说，某个事务的读写操作必须推迟到另一个写同一对象的事务提交或放弃之后。为了保证事务可自己选择是提交还是放弃，其操作针对于其他事务不可访问的临时版本。当事务提交时，对象的临时版本被拷贝到实际对象以及持久存储中。

嵌套事务由若干子事务组合形成。在分布式系统中，嵌套事务是非常有用的，因为它允许在不同服务器上并发执行子事务。嵌套事务还有一个好处是允许独立恢复部分事务。

操作冲突是形成各种并发控制协议的基础。并发控制协议不仅要确保串行化，并且要用严格执行来保证恢复处理，以避免与事务放弃，例如连锁放弃，有关的问题。

在调度事务中的某个操作时有3种策略：（1）立即执行；（2）推迟执行；（3）放弃事务。

严格的两阶段加锁使用了前两种策略，只有在死锁时才求助于放弃事务。它根据事务访问公共对象的时间对事务进行排序来保证事务的串行化。它的主要缺点是会造成死锁。

时间戳排序利用了所有的3种策略，它根据事务开始时的时间来排列事务对对象的访问顺序。这种方法不会引起死锁，并且对只读事务很有利。但是，到来较晚的事务必须被放弃。多版本时间戳排序是一种特别有效的方法。

509

乐观并发控制在事务的执行过程中不进行任何形式的检测，直到事务完成。事务在提交之前必须通过验证。向后验证需要维护已提交事务的多个写集合，而向前验证必须验证活动事务，它的好处是允许多种策略解决冲突。在乐观并发控制甚至在时间戳排序中，由于事务不能通过验证，不断地放弃会引起饥饿。

练习

12.1 TaskBag是一个提供“任务描述”仓库的服务。它支持在几个计算机上运行的客户并行执行部分计算。一个主进程放置TaskBag中一个计算的子任务描述，工作者进程从TaskBag中选择任务并实现它们，然后将结果的描述返回给TaskBag。主进程收集结果并将它们组合起来，产生最后的结果。

TaskBag服务提供下列操作：

setTask 允许客户向TaskBag中增加任务描述。

takeTask 允许客户从TaskBag中取出任务描述。

当一个任务当前是不可用的，但可能不久就是可用的时候，客户发出*takeTask*请求。讨论下列方法的优缺点：

（i）服务器马上回答，告诉客户以后重试。

（ii）让服务器操作（和客户）等待，直到任务变成可用。

（iii）使用回调。

12.2 一个服务器管理对象$a_1, a_2, \ldots, a_n$，它为客户提供下面两种操作：

read(*i*)返回对象a_i的值。

write(*i*, *Value*)将对象a_i设置为值*Value*。

事务*T*和*U*定义如下：

T：*x*=*read*(*j*); *y*=*read*(*i*); *write*(*j*, 44); *write*(*i*, 33)

U：*x*=*read*(*k*); *write*(*i*, 55); *y*=*read*(*j*); *write*(*k*, 66)

请给出事务*T*和*U*的3个串行化等价的交错执行。

12.3 针对练习12.2的事务*T*和*U*的串行化等价交错执行，给出满足下面特性的执行：（1）严格执行；（2）虽然不是严格执行，但是不会造成连锁放弃；(3) 会引起连锁放弃。 510

12.4 操作*create*在银行分行中创建一个新的银行账户。事务*T*和*U*分别定义如下：

T：*aBranch.create*("Z")

U：*z.deposit*(10); *z.deposit*(20)

假设账户Z不存在，并假设*deposit*操作在账户不存在时不做任何操作。考虑下面的事务*T*和*U*的交错执行：

T	U
aBranch.create(Z);	*z.deposit(10);* *z.deposit(20);*

按这个执行顺序，给出账户Z在执行后的余额。这种执行是否与*T*和*U*的串行等价执行一致？

12.5 练习12.4中被创建的对象如账户Z有时被称为假像。就事务*U*看来，账户Z一开始不存在，然后就像幻影一样出现。请用一个例子来说明删除账户时也会出现假像。

12.6 “转账”事务*T*和*U*分别定义如下：

T：*a.withdraw*(4); *b.deposit*(4);

U：*c.withdraw*(3); *b.deposit*(3);

假设它们组织成一对嵌套事务：

T_1：*a.withdraw*(4); T_2：*b.deposit*(4);

U_1：*c.withdraw*(3); U_2：*b.deposit*(3);

请比较T_1、T_2、U_1和U_2之间的串行等价性执行的数目和*T*和*U*的串行等价性执行的数目。试解释为什么嵌套事务比平面事务有更多的串行等价性执行？

12.7 考虑练习12.6中嵌套事务的恢复问题。假设*withdraw*事务在账户透支时将放弃，从而父事务也将放弃。请给出T_1、T_2、U_1和U_2的串行等价执行，分别满足条件：（1）严格执行；（2）非严格执行。并考虑严格的执行在多大程度上减少嵌套事务的并发度？

12.8 请解释为什么串行等价性要求一旦事务释放了对象上的某个锁，它就不允许再获得其他锁？

一个服务器管理对象$a_1, a_2, \ldots, a_n$。该服务器为客户提供两种操作： 511

read(*i*)返回对象a_i的值

write(*i*, *Value*)将对象a_i设置为值*Value*

事务*T*和*U*定义如下：

T：*x*=*read*(*i*); *write*(*j*, 44);

U：*write*(*i*, 55); *write*(*j*, 66);

请给出事务T和U的一个交错执行，在这个执行中由于锁过早释放而导致执行不是串行等价的。

12.9 练习12.8中的事务T和U在服务器上分别定义如下：

T：$x=read(i)$; $write(j, 44)$;

U：$write(i, 55)$; $write(j, 66)$;

对象a_i和a_j的初值分别是10和20，下面的执行哪些是串行等价的？哪些可能出现在两阶段加锁中？

(a)

T	U
x= read (i);	
	write(i, 55);
write(j, 44);	
	write(j, 66);

(b)

T	U
x= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(c)

T	U
	write(i, 55);
	write(j, 66);
x= read (i);	
write(j, 44);	

(d)

T	U
	write(i, 55);
x= read (i);	
	write(j, 66);
write(j, 44);	

12.10 考虑将两阶段锁的限制适当放宽，只读事务可以较早地释放读锁。那么一个只读事务是否能达到一致检索？对象是否会变得不一致？请用练习12.8中的事务T和U来说明你的结论：

T：$x=read(i)$; $y=read(j)$;

U：$write(i, 55)$; $write(j, 66)$;

其中对象a_i和a_j的初值分别是10和20。

12.11 事务的严格执行要求某个事务的读写操作必须推迟到写这个对象的所有其他事务提交或放弃之后。请解释图12-16中的加锁规则是如何保证严格执行的。

12.12 当事务在它完成所有操作后但在提交前就释放写锁时，请描述此时如何引起不可恢复的状态。

12.13 如果事务在它完成所有操作后但在提交前就释放读锁，请解释为什么此时事务的执行仍是严格的。根据这一点来改进图12-16中规则2。

512

12.14 考虑单服务器上的死锁检测机制，精确地描述何时将边加入等待图和从等待图中删除边。

利用练习12.8中的服务器上运行的事务T、U和V来说明你的答案：

T	U	V
	write(i, 66)	
write(i, 55)		
		write(i, 77)
	commit	

当事务U释放它在a_i的写锁时，T和V都在试图等待获取这个写锁。如果T(首先到达)在V之前获得锁，你的方案能否正确工作？如果不能，请修改你的描述。

12.15 考虑图12-26中的层次锁。如果某次会见被安排在w周的d天的时刻t，那么需要设置哪些锁？这些锁应该按照什么次序设置？是释放这些锁的次序吗？

当查看w周的每天的时间段时需要设置哪些锁？在已有锁的时候，能做吗？

12.16 考虑将乐观并发控制应用于练习12.9中的事务T和U的情况。如果事务T和U同时处于活动状态，试描述以下几种情况的结果如何：

（1）服务器首先处理T的提交请求，使用向后验证方式。

（2）服务器首先处理U的提交请求，使用向后验证方式。

（3）服务器首先处理T的提交请求，使用向前验证方式。

（4）服务器首先处理U的提交请求，使用向前验证方式。

对于上面的每种情况，描述事务T和U的操作顺序，注意写操作在验证通过之后才真正起作用。

12.17 考虑事务T和U的交错执行：

T	U
openTransaction	*openTransaction*
y = read(k);	
	write(i, 55);
	write(j, 66);
	commit
x = read(i);	
write(j, 44);	

513

在使用乐观并发控制的向后验证时，由于事务T的针对a_i的读操作与事务U的写操作冲突，事务T将被放弃，尽管这个执行是串行等价的。请改进算法来处理这种情况。

12.18 试比较练习12.8中事务T和U分别在两阶段加锁(练习12.9)和乐观并发控制(练习12.16)中的操作执行顺序。

12.19 考虑将时间戳排序用于练习12.9中事务T和U的各种交错执行情况。对象a_i和a_j的初值分别是10和20，初始的读写时间戳都是t_0。假设每个事务在开始第一个操作之前就获得时间戳，例如在情况(a)中，T和U获得的时间戳分别是t_1和t_2，并且$t_0 < t_1 < t_2$。请根据时间顺序描述T和U的各个操作的效果。对每个操作需陈述：

（1）根据读规则或写规则，这个操作是否允许执行。

（2）赋给事务或者对象的时间戳。

（3）临时对象的创建和它们的值。

对象的最终值和时间戳分别是什么？

12.20 根据下面的事务T和U的交错执行，重新考虑练习12.19。

T	U
openTransaction	
	openTransaction
	write(i, 55);
	write(j, 66);
x = read (i);	
write(j, 44);	
	commit

T	U
openTransaction	
	openTransaction
	write(i, 55);
	write(j, 66);
	commit
x = read (i);	
write(j, 44);	

12.21 利用多版本时间戳排序，重新考虑练习12.20。

12.22 在多版本时间戳方式中，读操作可以访问对象的临时版本。请举例说明，如果所有的读操作都允许立即执行，则有可能造成连锁放弃。

12.23 与普通的时间戳排序相比，多版本时间戳排序有哪些优点和缺点？

12.24 试比较练习12.8中事务T和U分别在两阶段加锁(练习12.9)和乐观并发控制(练习12.16)中的操作执行次序。

514

第13章　分布式事务

13.1　简介
13.2　平面分布式事务和嵌套分布式事务
13.3　原子提交协议
13.4　分布式事务的并发控制
13.5　分布式死锁
13.6　事务恢复
13.7　小结

本章介绍分布式事务，即涉及多个服务器的事务。分布式事务可以是平面事务，也可以是嵌套事务。

原子提交协议是参与分布式事务的服务器使用的一个协作过程，它使多个服务器能够共同决策是提交事务还是放弃事务。本章描述了两阶段提交协议，它是最常用的原子提交协议。

分布式事务的并发控制一节讨论为支持分布式事务将如何扩展加锁、时间戳排序和乐观并发控制。

使用锁机制可能会造成分布式死锁，本章将讨论分布式死锁的检测算法。

每个提供事务的服务器都包含一个恢复管理器，在出现故障之后服务器被替换时，用它来恢复服务器所管理的对象上的事务。恢复管理器将对象、意图列表和每个事务的状态信息记录在持久存储中。

13.1　简介

第12章讨论了只访问一个服务器中对象的平面事务和嵌套事务。通常情况下，不管是平面事务还是嵌套事务，它们都需要访问不同计算机上的对象。访问由多个服务器管理的对象的平面事务或嵌套事务被称为分布式事务。

当一个分布式事务结束时，事务的原子特性要求所有参与该事务的服务器必须全部提交或全部放弃。为了实现这一点，其中一个服务器承担了协调者的角色，由它来保证在所有的服务器上获得同一结果。协调者的动作取决于它选用的协议。“两阶段提交协议”是最常用的协议，该协议允许服务器之间相互通信，可以就提交或放弃共同做出决定。

分布式事务的并发控制基于第12章中讨论的方法。每个服务器对自己的对象应用本地的并发控制，以保证事务在局部是串行化的。分布式事务还需要保证全局串行化，如何实现这一点与是否使用锁、时间戳排序或乐观并发控制有关。在某些情况下，事务在单个服务器上是串行化的，但在同一时间，由于在不同服务器之间存在相互依赖循环，所以可能导致分布式死锁。

事务恢复用于保证事务所涉及的所有对象都是可恢复的。除此之外，它还保证对象只反映已提交事务所做的更新，不反映被放弃事务所做的更新。

13.2 平面分布式事务和嵌套分布式事务

如果客户事务调用了若干不同服务器上的操作，那么它就成为一个分布式事务。有两种构造分布式事务的方式：按平面事务构造和按嵌套事务构造。

在平面事务中，客户给多个服务器发请求。例如，在图13-1（a）中，事务T是一个平面事务，它调用了服务器X、Y和Z上的对象操作。一个平面客户事务完成一个请求之后才发起下一个请求。因此，每个事务顺序访问服务器上的对象。当服务器使用加锁机制时，事务一次只能等待一个对象。

在嵌套事务中，顶层事务可以创建子事务，每个子事务可以进一步地嵌套任意深度的子事务。图13-1（b）给出了一个客户事务T，它创建了两个子事务T_1和T_2，它们分别访问服务器X和Y上的对象。子事务T_1和T_2又进一步创建了子事务T_{11}，T_{12}，T_{21}和T_{22}，它们分别访问服务器M、N和P上的对象。在嵌套事务中，同一层次的子事务可并发执行，所以T_1和T_2是并发执行的，又由于它们访问不同服务器上的对象，所以它们能并行执行。同样，T_{11}，T_{12}，T_{21}和T_{22}也是并发执行的。

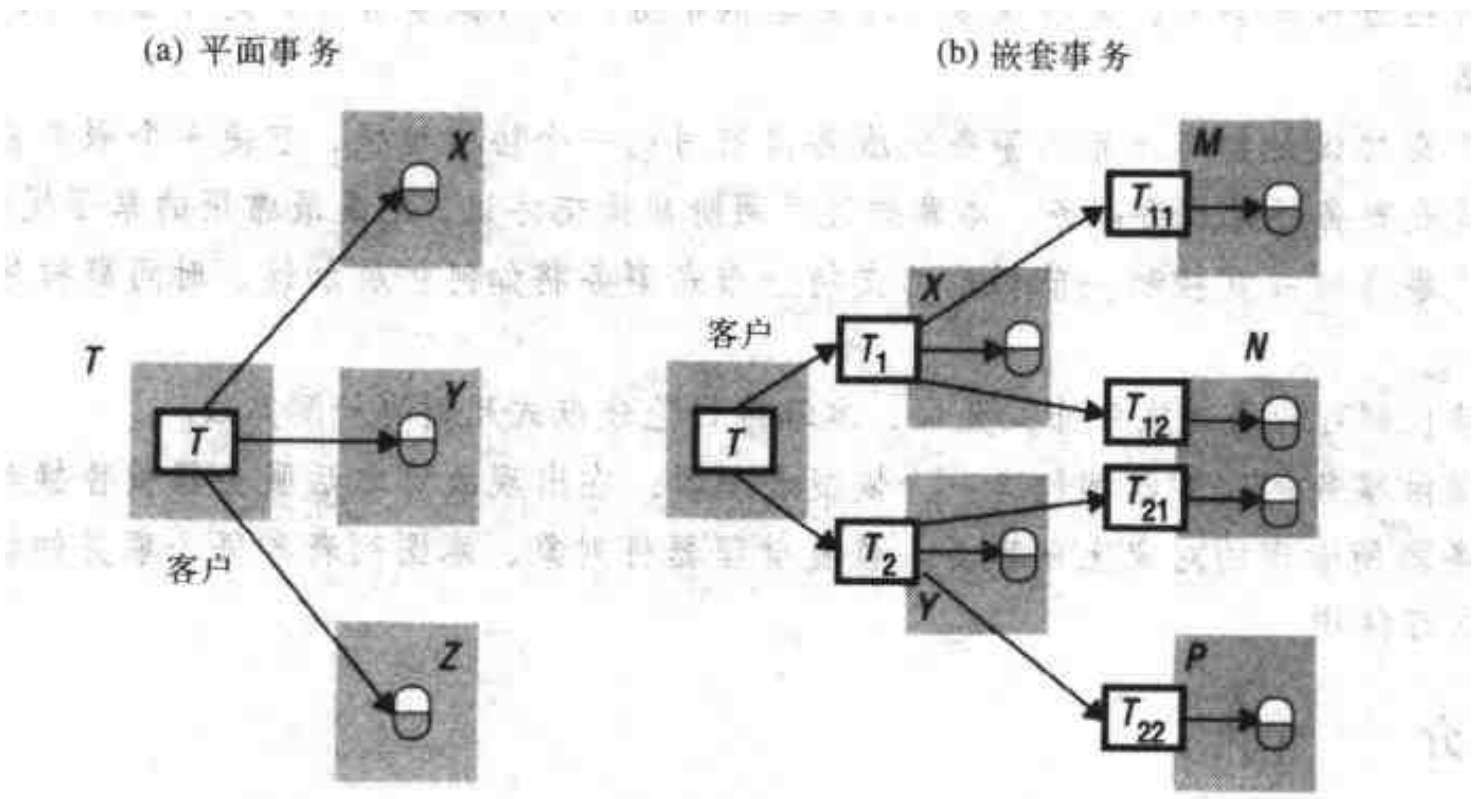


图13-1 分布式事务

现在考虑这样一个分布式事务：客户从A账户转账10美元到C账户，然后从B账户转账20美元到D账户。账户A和B分别在服务器X和Y上，而账户C和D在服务器Z上。如果将该事务组织成4个嵌套事务（如图13-2所示），那么4个请求（两个*deposit*请求和两个*withdraw*请求）可

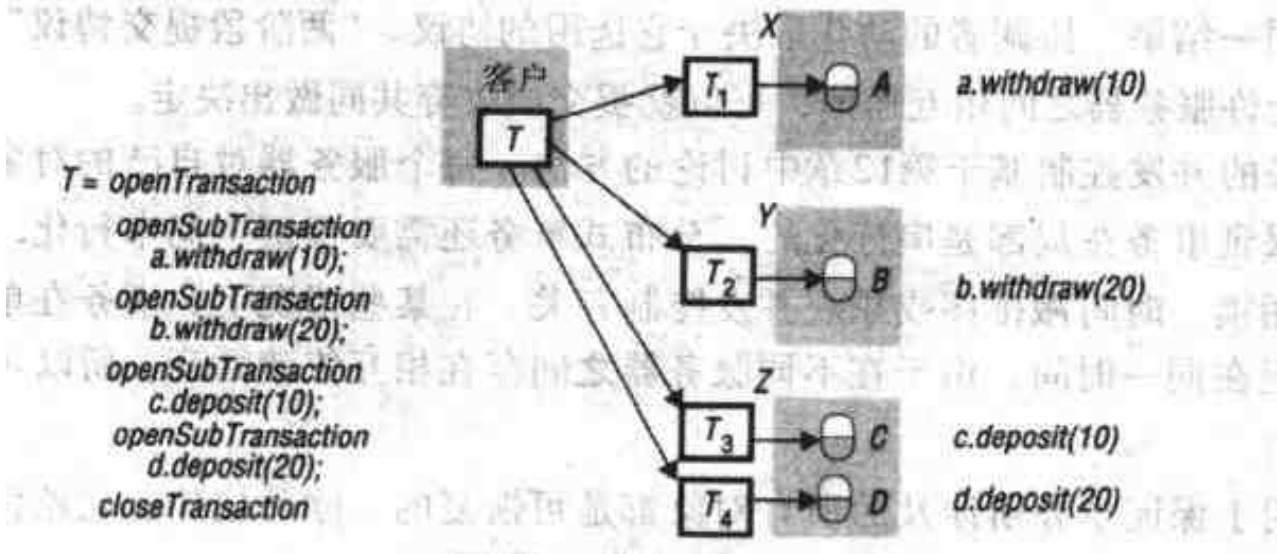


图13-2 嵌套的银行事务

以并行执行，从而整体执行性能优于4个操作被顺序调用的简单事务。 516 ~ 517

分布式事务的协调者

执行分布式事务请求的服务器需要相互通信，以确保在事务提交时能够协调它们之间的动作。客户在启动一个事务时，向任一服务器上的协调者发出一个*openTransaction*请求，参见12.2节的描述。协调者处理完*openTransaction*请求后，将事务标识（TID）返回给客户。分布式事务的事务标识必须在整个分布式系统中是惟一的。一种构造TID的简单的方法是将TID分成两部分：创建该事务的服务器的标识（例如IP地址）以及对服务器来讲是惟一的数字。

创建某一分布式事务的协调者成为该分布事务的协调者，它在分布式事务结束时负责提交或放弃事务。分布式事务访问的每个服务器都是该事务的参与方，每个服务器提供一个我们称为参与者的对象。每个事务参与者负责跟踪所有参与分布式事务的可恢复对象。这些参与者配合协调者共同执行提交协议。

在事务的执行过程中，协调者在列表中记录所有对参与者的引用，每一个参与者都记录一个对协调者的引用。

图12-3给出的*Coordinator*接口提供了方法*join*，它用于将一个新的参与者加入当前事务：

join(Trans, reference to participant) 通知协调者一个新的参与者已加入到事务*Trans*中

协调者将新的参与者记录到参与者列表中。事实上，协调者知道所有的参与者，而每个参与者也知道协调者，这样在事务提交时，协调者和参与者都能收集到必要的信息。

图13-3显示了一个客户，它的（平面）银行事务涉及到服务器*BranchX*、*BranchY*和*BranchZ*上的账户*A*、*B*、*C*和*D*。该客户事务*T*从账户*A*转账4美元到账户*C*，然后从账户*B*转账3美元到账户*D*。将图中左边描述的事务*T*展开，我们可以看到事务*T*的*openTransaction*和*closeTransaction*操作被送往协调者，协调者可以位于任何一个参与事务的服务器上。每个服

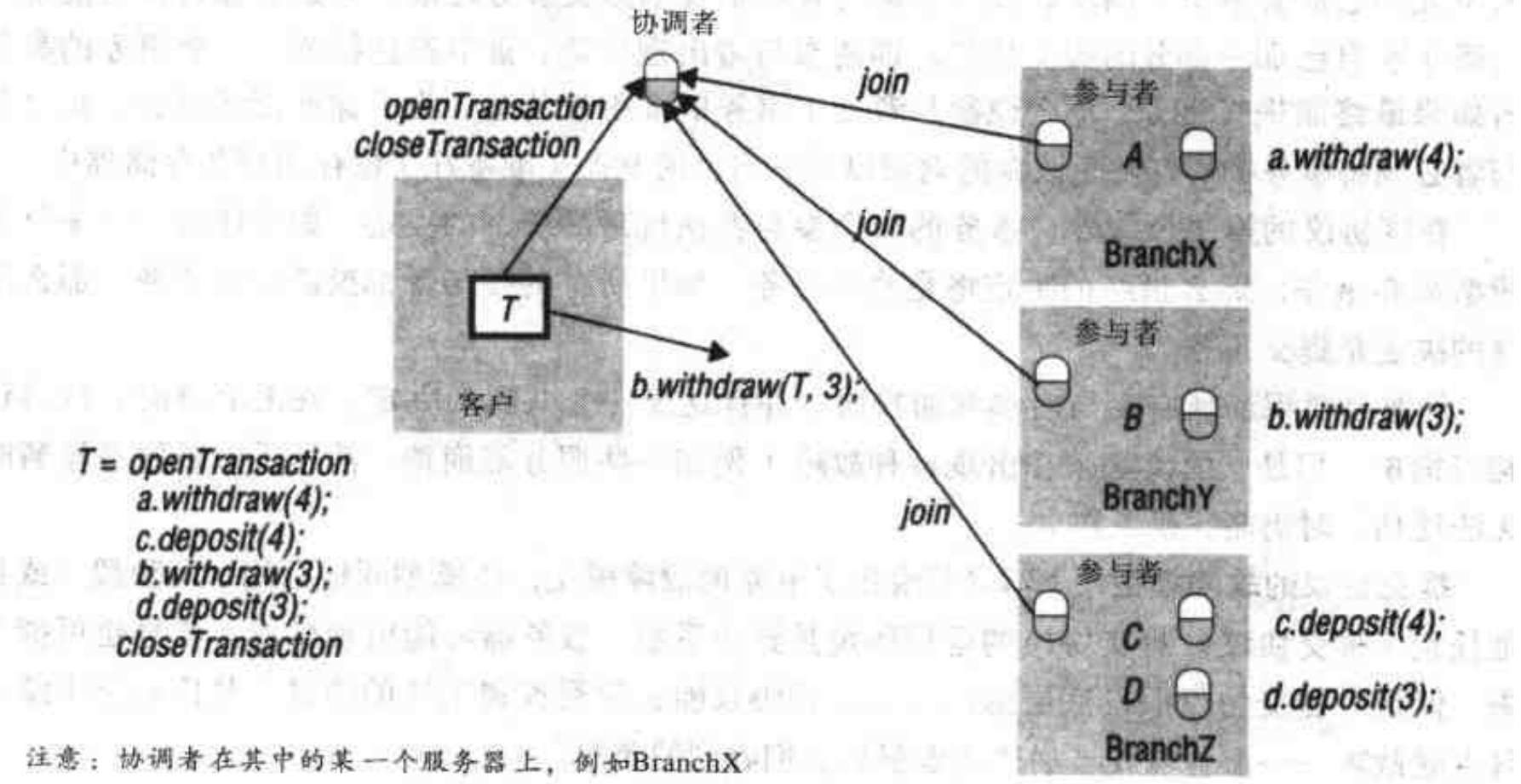


图13-3 一个分布式银行事务

务器上都有一个参与者，它们通过调用协调者的join方法加入该事务。当客户调用事务中的一个方法时，例如*b.withdraw*(T, 3)接收该调用的对象（服务器*BranchY*的*B*对象）将通知它的参与者对象说自己属于事务*T*。如果在这之前没有通知过协调者，则参与者对象将调用*join*方法来通知协调者。在这个例子中，我们看到事务标识也被作为参数传递，这样，接收者就能将它传递给协调者。在客户调用*closeTransaction*时，协调者就拥有了所有参与者的引用。

值得注意的是，任何一个参与者可能由于某些原因无法继续事务而调用协调者的*abortTransaction*方法。

13.3 原子提交协议

事务的提交协议最早发明于20世纪70年代，而两阶段提交协议是由Gray[1978]提出的。事务的原子特性要求分布式事务结束时，它的所有操作要么全部执行，要么全部不执行。就分布式事务而言，客户请求了多个服务器上的操作。在客户请求提交或放弃事务时，事务结束。以原子方式完成事务的一个简单的方法是让协调者不断地向所有参与者发送提交或放弃请求，直到所有参与者确认已执行完相应操作。这是一个单阶段原子提交协议的例子。

但是这种简单的单阶段原子提交协议是不够用的，在客户发出提交请求后，该协议不允许任何服务器单方面放弃事务。阻止服务器提交事务的原因通常与并发控制问题有关。例如，如果使用加锁，为了解除死锁问题需要将事务放弃，客户在发起新的请求之前并不知道事务已被放弃。如果使用乐观并发控制，某个服务器的验证失败将导致放弃事务。在分布式事务的进行过程中，协调者可能并不知道某个服务器已经崩溃并且已被替换——这样的服务器也需要放弃事务。

两阶段提交协议的设计出发点是允许任何一个参与者自行放弃它那部分事务。由于事务原子性的要求，如果一个事务部分被放弃，那么整个分布式事务也要被放弃。在该协议的第一个阶段，每个参与者投票表决事务是放弃还是提交。一旦参与者投票要求提交事务，那么就不允许它放弃事务。因此，在一个参与者投票要求提交事务之前，它必须保证最终能够执行属于它自己那一部分的提交协议，即使参与者出现故障，被中途替换掉。一个事务的参与者如果最终能提交事务，那么说参与者处于事务的准备好状态。为了保证能够提交，每个参与者必须将事务中所有发生改变的对象以及它自己的状态（准备好）保存到持久存储器中。

518 ~ 519

在该协议的第二个阶段，事务的每个参与者执行最终统一的决定。如果任何一个参与者投票放弃事务，那么最终的决定将是放弃事务。如果所有的参与者都投票提交事务，那么最终的决定是提交事务。

问题是要保证所有参与者都参加投票，并且达成一个共同的决定。在无故障时，该协议相当简单。但是，协议必须在出现各种故障（例如一些服务器崩溃、消息丢失或服务器暂时无法通信）时仍能正确工作。

提交协议的故障模型 12.1.2节给出了事务的故障模型，该模型同样适用于两阶段（或其他任何）提交协议。提交协议的运行环境是异步系统，服务器可能出现崩溃，消息也可能丢失。但是，提交协议假设底层的请求－应答协议能去除受损和重复的消息，并且系统中没有拜占庭故障——服务器或者崩溃或者服从它们收到的消息。

两阶段提交协议是一种达到共识的协议。第11章断言，在异步系统中，如果进程可能崩溃，那么是不可能达到共识的。但是，两阶段提交协议确实在这些条件下达成了共识，这是

由于进程的崩溃故障被屏蔽了，崩溃的进程被一个新进程替代，新进程的状态由持久存储器中的信息和保存在其他进程中的信息设定。

13.3.1 两阶段提交协议

在事务的进行过程中，除了参与者在加入分布式事务时通知协调者之外，协调者和参与者之间没有其他通信。客户的事务提交（或放弃）请求被送往协调者。如果客户请求*abortTransaction*，或者事务已被某个参与者放弃，那么协调者可以立即通知所有参与者放弃事务。只有当客户请求协调者提交事务时，两阶段提交协议才开始被使用。

在两阶段提交协议的第一个阶段，协调者询问所有的参与者是否准备好提交；在第二个阶段，协调者通知它们提交（或放弃）事务。如果某个参与者可以提交它那部分事务，那么当它将所做的更新和它的状态记录到持久存储器以后，它就同意提交事务，并准备好提交。为实现两阶段提交协议，分布式事务中的协调者和参与者利用图13-4总结出的操作进行通信。其中，*canCommit*、*doCommit*和*doAbort*方法是参与者接口中的方法，而方法*haveCommitted*和*getDecision*在协调者接口中。

520

两阶段提交协议由投票阶段和完成阶段组成，如图13-5所示。在步骤2结束时，协调者和所有投*Yes*票的参与者都准备提交。在步骤3结束时，事务实际已经结束。在步骤3（a）时，协调者和参与者提交事务，因此协调者将提交事务的决定通知客户；在步骤3（b）时，协调者将放弃事务的决定通知客户。

canCommit?(trans)→ Yes / No
协调者询问参与者是否能够提交事务，参与者将应答自己的投票结果
doCommit(trans)
协调者告诉参与者提交它那部分事务
doAbort(trans)
协调者告诉参与者放弃它那部分事务
haveCommitted(trans, participant)
参与者向协调者确认它已经提交了事务
getDecision(trans)→ Yes / No
当参与者投Yes票后一段时间内未收到应答时，参与者向协调者询问事务的投票表决结果。该操作用于从服务器崩溃或消息延迟中恢复

图13-4 两阶段提交协议中的操作

阶段1（投票阶段）：
1. 协调者向事务的所有参与者发送*canCommit*? 请求。
2. 当参与者收到*canCommit*? 请求后，它将向协调者应答它的投票（*Yes*或者*No*）。在投*Yes*票之前，它在持久存储中保存所有对象，准备提交。如果投*No*票，参与者立即放弃。

阶段2（根据投票结果完成事务）：
3. 协调者收集所有的投票（包括它自己的投票）。
（a）如果不存在故障并且所有的投票均是*Yes*时，那么协调者决定提交事务并向所有参与者发送*doCommit*请求。
（b）否则，协调者决定放弃事务，并向所有投*Yes*票的参与者发送*doAbort*请求。
4. 投*Yes*票的参与者等待协调者发的*doCommit*或者*doAbort*请求。一旦参与者接收到任何一种请求消息，它根据该请求放弃或者提交事务。如果请求是提交事务，那么它还要向协调者发送一个*haveCommitted*来确认事务已经提交。

图13-5 两阶段提交协议

在步骤4，所有的参与者确认它们已提交，此时，协调者就知道它所记录的事务信息将不再需要。

显然，由于一个或多个服务器崩溃或服务器之间的通信中断，协议可能出错。为处理可能的崩溃，每个服务器需要将与两阶段提交协议相关的信息保存到持久存储器中。这些信息可由替代崩溃服务器的新进程获取。分布式事务的恢复处理将在13.6节讨论。

协调者和参与者之间的信息交换会由于服务器崩溃或消息丢失而失败。采用超时可防止进程无限等待。当进程监测到超时后，它必须采取适当的措施。考虑到这一点，协议在进程可能阻塞的每一步都包括了一个超时动作。这些动作的设计虑及下列事实：在异步系统中，超时并不一定意味着服务器出现故障。

两阶段提交协议的超时动作 在两阶段协议的不同时期，协调者或参与者在接收到其他进程的请求或应答之后才能进行它那部分的协议处理。

首先来考虑这样的情形：某个参与者投*Yes*票并等待协调者发回投票结果，即告诉它是提交事务还是放弃事务。参见图13-6的步骤2，这样一个参与者的结果是不确定的，它在从协调者处得到投票结果之前不能进行进一步处理。参与者不能单方面决定下一步做什么，同时该事务使用的对象也不能被释放用于其他事务。参与者向协调者发出*getDecision*请求来获取事务的结果。当它收到应答时，才能进行图13-5中协议的步骤4。如果协调者发生故障，那么参与者将不能获得决定，直到协调者被替代，这可能导致处在不确定状态的参与者长时间地延迟。

另一种不依靠协调者获取最终决定的策略是通过参与者协作来获得决定。这种策略的优点是可以在协调者出故障时使用。有关详细情况请参考练习13.5和文献[Bernstein等1987]。但是，即使使用协作协议，如果所有的参与者都处于不确定状态，那么它们仍然不能得到决定，直到协调者或一个参与者得知最终结果为止。

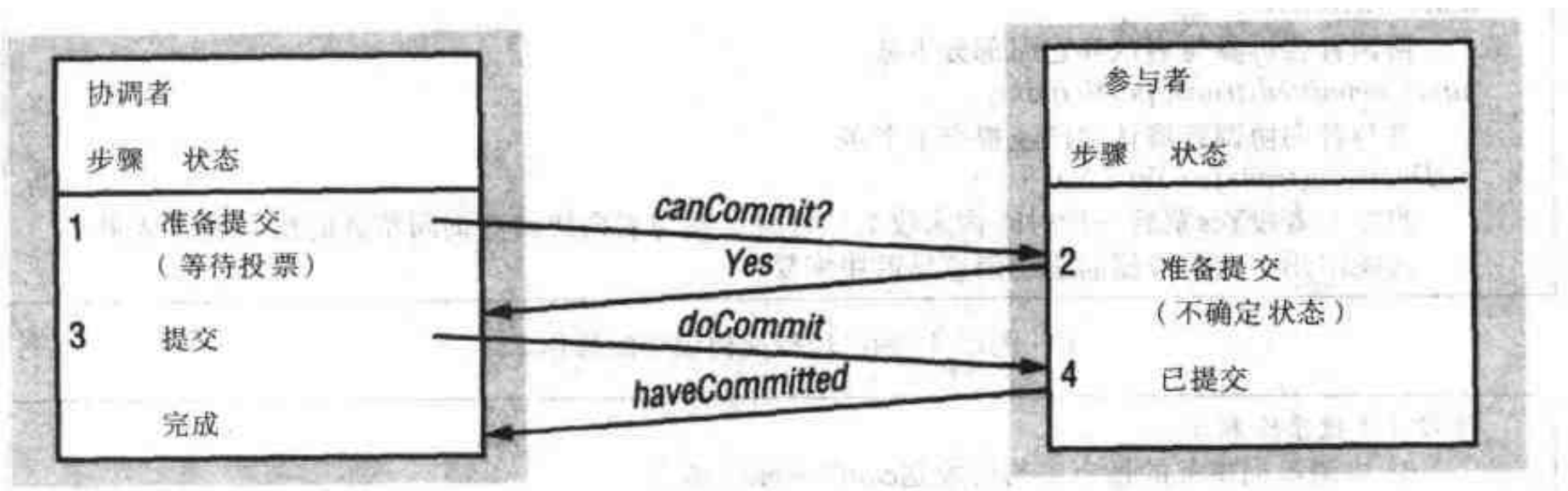


图13-6 两阶段提交协议中的通信

另一种可能导致参与者延迟的情况是，参与者已经完成了事务中所有的客户请求，但还没有收到协调者发来的*canCommit?*消息。当客户向协调者发送*closeTransaction*时，如果参与者长时间（例如，在一个锁操作的超时时段中）未收到任何有关该事务的操作请求，那么它只能检测这种情况。因为在这个阶段还没有做任何决定，所以参与者能在一段时间后决定单方面放弃该事务。

协调者在等待参与者投票时可能会被延迟。由于它还未决定事务的最终命运，在等待一段时间后它可以决定放弃该事务。但是它必须给所有参加了投票的参与者发送*doAbort*。一些反

应较慢的参与者此后仍然可能投*Yes*票，但这些投票将被忽略，它们将进入前述的不确定状态。

两阶段提交协议的性能 假设一切运行正常——即协调者和参与者不出现崩溃，通信也正常时，有N个参与者的两阶段提交协议需要传递N个*canCommit?*消息和应答，然后再有N个*doCommit*消息。这样，消息开销与$3N$成正比，时间开销是3次消息往返。由于协议在没有*haveCommitted*消息时仍能正确运行——它们的作用只在于通知服务器删除过时的协调者信息，因此在估计协议开销上，不将*haveCommitted*消息计算在内。

在最坏的情况下，两阶段提交协议执行过程中可能出现任意多次服务器和通信故障。尽管协议不可能指定完成的时间限制，但它能够处理连续故障（服务器崩溃或消息丢失），并保证最终完成。

像前面提到的超时问题，两阶段提交协议可能造成参与者很长时间停留在不确定状态上。这些延迟主要由于协调者故障或者不能从参与者得到*getDecision*请求的应答。即使协作协议允许参与者可以向其他参与者发送*getDecision*请求，但是当所有参与者都处于不确定状态时，延迟仍然不可避免。

三阶段提交协议用来减少这种延迟。但是这种协议代价很大，在正常的情况（无故障情况）下需要更多的消息和更多的消息往返延时。关于三阶段提交协议的详细情况，参见练习13.2和Bernstein等[1987]。

521 ~ 523

13.3.2 嵌套事务的两阶段提交协议

一组嵌套事务的最外层事务称为*顶层事务*，除顶层事务之外的其他事务被称为*子事务*。在图13-1（b）中，T是顶层事务，T_1，T_2，T_{11}，T_{12}，T_{21}和T_{22}是子事务。T_1和T_2是事务T的孩子事务，即T是它们的父事务。类似地，T_{11}和T_{12}是事务T_1的孩子事务，T_{21}和T_{22}是事务T_2的孩子事务。每个子事务在父事务开始后才能执行，并在父事务结束前结束。例如，T_{11}和T_{12}在T_1开始后执行，在T_1结束前结束。

当子事务执行完毕时，它独立决定是临时性提交还是放弃。临时提交和准备好状态是不同的：它只是一个局部决定，也不用备份到持久存储器中。如果服务器此后崩溃，那么该服务器的替代者不能执行以前的临时提交。正是由于这个原因，嵌套事务的两阶段提交协议需要服务器放弃出故障的临时提交的事务。

如图13-7所示，子事务的协调者提供创建下一层子事务的操作，并提供操作使子事务的协调者可以查询父事务是已经提交了还是放弃了。

openSubTransaction(trans)→ subTrans 创建一个新的子事务，它的父事务是*trans*，该操作返回一个惟一的子事务标识 *getStatus(trans)→ committed, aborted, provisional* 向协调者询问事务*trans*的当前状态。返回值表示下列情况：已提交、已放弃、临时提交

图13-7 嵌套事务中协调者的操作

客户使用*openTransaction*操作创建一个顶层事务，从而启动一组嵌套事务。*openTransaction*操作返回顶层事务的事务标识。客户调用*openSubTransaction*操作创建子事务，*openSubTransaction*操作的参数要求指定它的父事务。新创建的子事务自动加入到父事务中并返回一个新创建子事务的标识。

子事务的标识必须是其父事务标识的扩展，子事务标识的构造方法应使得能根据这一标

识确定父事务或顶层事务的标识。另外，所有子事务的标识必须是全局惟一的。客户通过在顶层事务的协调者上调用*closeTransaction*或*abortTransaction*来结束整个嵌套事务。

与此同时，每个嵌套事务执行自己的操作。当它们结束时，管理这些子事务的服务器记
524 录下事务临时提交或放弃的信息。注意，如果父事务放弃，那么它的子事务将被强制放弃。

第12章提到，尽管子事务可能被放弃，但是它的父事务——包括顶层事务——仍然可以提交。在这种情况下，父事务将根据子事务提交或是放弃的不同情况采取不同的行动。例如，银行需要在特定的某一天在某一支行上完成“未结算订单”事务。这个事务包含若干个嵌套的*Transfer*子事务，每个*Transfer*事务由嵌套的*deposit*子事务和*withdraw*子事务组成。我们假设当某个账户透支时，*withdraw*事务将被放弃，从而*Transfer*事务也被放弃。但是某个*Transfer*子事务的放弃并不要求放弃所有的长期订单事务。相反，顶层事务将发现*Transfer*事务执行失败并采取相应的行动。

考虑图13-8所示的顶层事务*T*和它的子事务（图13-8基于图13-1（b））。每个子事务或者临时提交或者放弃。例如，T_{12}临时提交而T_{11}被放弃，但是T_{12}的命运由它的父事务T_1决定，且最终依赖顶层事务*T*。尽管T_{21}和T_{22}都临时提交了，但T_2被放弃了，这意味着T_{21}和T_{22}必须也被放弃。假设顶层事务*T*不考虑T_2被放弃的事实，最终决定提交，同时T_1也不考虑T_{11}被放弃的事实，仍决定提交。

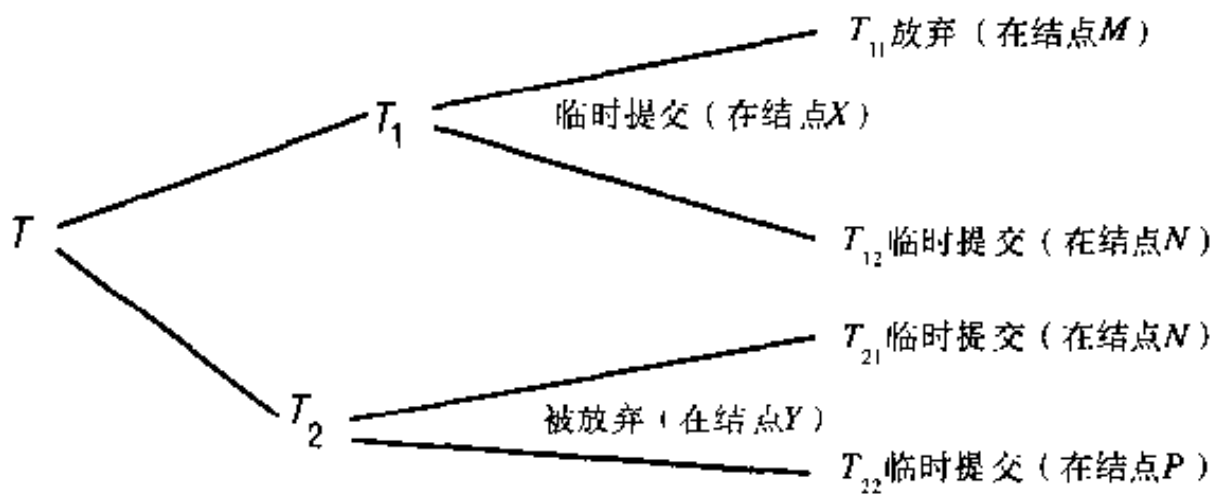


图13-8 事务*T*决定是否提交

当顶层事务完成后，它的协调者将执行两阶段提交协议。参与者子事务不能完成的惟一原因是在它临时提交后服务器出现崩溃。回想一下，当每个子事务被创建时，它就加入到父事务中。因此，每个父事务的协调者都有一个它的孩子子事务列表。当一个嵌套事务临时提交时，它将自己的状态和所有子事务的状态报告给它的父事务。当一个嵌套事务放弃时，它只需将自己的放弃报告给父事务，而不用报告子事务的任何信息。最终，顶层事务将获得嵌套事务树中所有子事务及其状态列表，而被放弃的子事务不在这个列表中。

图13-8给出的例子中各协调者持有的信息在图13-9中列出。注意T_{12}和T_{21}在同一个服务器*N*上运行，因此它们共用一个协调者。当子事务T_2放弃时，它将把该事实报告它的父事务*T*，但不传递它的子事务T_{21}和T_{22}的任何信息。如果事务的某个祖先事务或是被显式放弃或是由于其
525 协调者崩溃而被放弃，那么，我们把这个子事务称为孤儿。在我们的例子中，子事务T_{21}和T_{22}都是孤儿，因为它的父事务被放弃，从而没有将它们的信息传递给顶层事务。但是，它们的协调者可以使用*getStatus*操作来获取父事务的状态。如果某个事务被放弃，那么它的临时提交的子事务也必须放弃，而不管顶层事务最终是否提交。

事务的协调者	子事务	是否为参与者	临时提交列表	放弃列表
T	T_1, T_2	是	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	是	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	否（被放弃）		T_2
T_{11}		否（被放弃）		T_{11}
T_{12}, T_{21}		T_{12}是（不包含T_{21}）	T_{21}, T_{12}	
T_{22}		否（父事务被放弃）	T_{22}	

图13-9 嵌套事务各协调者持有的信息

顶层事务在两阶段提交协议中扮演协调者角色，参与者列表由所有临时提交子事务的协调者组成，注意这些子事务必须是没有被放弃的祖先事务，到了这个阶段，程序的逻辑已经决定了顶层事务将试图提交整个事务，而不管是否有一些被放弃的子事务。在图13-8中，事务T以及事务T_1和T_{12}的协调者是参与者，它们将投票表决是否提交。如果它们投票提交事务，那么它们必须将对象保存到持久存储器中准备提交。这个状态被记录在顶层事务中，此后，两阶段提交协议可以使用层次的或平面的方式来完成。

两阶段提交协议的第二阶段与非嵌套的情况是一致的。协调者收集所有的投票，然后将最终决定通知所有的参与者。协议结束时，协调者和参与者将一致地提交或一致地放弃整个事务。

层次两阶段提交协议 在这个方法中，两阶段提交协议变成一个多层的嵌套协议。顶层事务的协调者和直接的子事务的协调者进行通信。它向每一个子事务的协调者发送*canCommit?*消息，这些子事务协调者收到消息后，又向各自的子事务协调者发送该消息（直至整个嵌套事务树）。每个参与者首先收集所有子事务的应答，然后再应答自己的父事务。在我们的例子中，T向事务T_1的协调者发送*canCommit?*消息，然后T_1向T_{12}发送*canCommit?*消息询问T_1子事务的应答。由子事务T_2被放弃，协议就没有向它的协调者发送消息。图13-10给出了*canCommit?*需要的参数。其中，第一个参数是顶层事务的TID，而第二个参数是发起*canCommit?*调用的事务的TID。每当参与者接收到调用后，将在它的事务列表中查看已临时提交的事务或子事务是否与第二个参数中的TID相匹配。例如，由于T_{12}的协调者和T_{21}的协调者运行在同一个服务器上，所以两者是一样的，但是如果服务器收到的*canCommit?*调用的第二个参数是T_1时，只需处理T_{12}即可。

526

canCommit?(*trans*, *subTrans*) → *Yes* / *No*

用来向某个子事务的协调者询问是否能够提交某个子事务*subTrans*。第一个参数*trans*是顶层事务的标识。参与者将应答自己的投票*Yes*或者*No*

图13-10 层次两阶段提交协议中的*canCommit?*调用

如果参与者找到能够匹配第二个参数的任何子事务，那么它就准备提交对象并且应答*Yes*。如果不能找到匹配的子事务，那么它在执行子事务之后系统必定出现过崩溃，因此它将应答*No*。

平面两阶段提交协议 在这个方法中，顶层事务的协调者向临时提交列表中的所有子事务发送*canCommit?*消息。在我们的例子中，顶层事务向T_1和T_{12}的协调者发送消息。此时，每个参与者都用顶层事务的TID来引用事务。每个参与者都查找自己的事务列表，来寻找能够匹

配顶层事务TID的事务及子事务。例如，T_{12}的协调者也是T_{21}的协调者，因为它们运行在同一个服务器上（服务器N）。

但是，当服务器N上的协调者存在临时提交和放弃状态并存的子事务时，这种方法就不能为正确处理提供足够的信息。如果服务器N的协调者正准备提交T，根据本地信息，T_{12}和T_{21}都是临时提交状态，那么两者均会提交，但是对于T_{21}来说，由于其父事务T_2被放弃，提交T_{21}是不正确的。为了处理这种情况，平面两阶段提交协议中的*canCommit?*操作的第二个参数提供了放弃子事务的列表，如图13-11所示。参与者提交顶层事务的后代，除非这些后代有被放弃的祖先。当参与者收到*canCommit?*请求后，它进行下面操作：

- 如果参与者有临时提交的子事务，并且它们是顶层事务*trans*的后代事务时：
  - 确保这些子事务的祖先不在*abortList*中，然后准备提交（在持久存储器中记录事务状态和它的对象）；
  - 如果子事务的祖先在*abortList*中，则放弃这些子事务；
  - 向协调者发送*Yes*投票。
- 如果参与者没有任何临时提交的顶层事务的后代事务，那么在执行子事务后系统一定曾经崩溃过，故向协调者发送*No*投票。

canCommit?(*trans*, *abortList*)→*Yes* / *No* 由协调者向参与者调用该操作，用来询问它是否能够提交某个事务。参与者将应答自己的投票*Yes*或者*No*

图13-11 平面两阶段提交协议中的*canCommit?*调用

两种方法的比较 层次协议的优点在于，在任何阶段，参与者只需查找直接的子事务，而平面协议要求提供一个放弃列表来去除那些父辈已放弃的子事务。Moss[1985]更喜欢平面算法，因为平面协议允许顶层事务的协调者可以直接和所有的参与者进行通信，而层次事务需要按嵌套关系来传递一系列消息。

超时动作 与非嵌套事务一样，嵌套事务的两阶段提交协议也会在同样3个地方造成协调者和参与者延迟。除此之外，还有第四个地方会延迟子事务。考虑被放弃子事务的临时提交孩子事务：它们没必要获取事务提交或放弃的信息。在我们的例子中，T_{22}就是这样一个子事务——它临时提交，但是它的父事务T_2却被放弃，所以T_2没有成为参与者。为了解决这个问题，任何未收到*canCommit?*消息的子事务在经过一个超时时段后将进行查询。图13-7中的*getStatus*操作可支持子事务查询它的父事务是否提交或放弃。为了使这些查询成为可能，放弃子事务的协调者需要存活一段时间。如果一个孤儿子事务不能联系上其父事务，那么它将最终被放弃。

13.4 分布式事务的并发控制

每个服务器管理一组对象，它必须保证并发事务访问这些对象时，对象保持一致性。因此，每个服务器需要对自己的对象应用并发控制机制。分布式事务中所有服务器共同保证事务以串行等价方式执行。

这意味着，如果事务T对某一个服务器上对象的冲突访问在事务U之前，那么在所有服务器上都按这一顺序访问被T和U访问的对象。

13.4.1 锁

在分布式事务中，某个对象的锁总是在同一个服务器中，授权这些锁是本地锁管理器决定的。本地锁管理器决定是满足客户对锁的请求，还是让发请求的事务等待。但是事务在所有服务器上被提交或放弃之前，本地锁管理器不能释放任何锁。在使用锁机制的并发控制中，原子提交协议进行过程中对象始终被锁住，其他事务不能访问这些对象。如果事务在第一阶段就被放弃时，锁可以提早释放。

527 ≀ 528

由于不同服务器上的锁管理器各自独立地设置对象锁，因此，对不同的事务，它们之间的加锁次序可能不一致。见下图中的事务*T*和事务*U*在服务器*X*和服务器*Y*之间的交替执行：

T			*U*		
Write(A)	服务器*X*	对*A*加锁			
			Write(B)	服务器*Y*	对*B*加锁
Read(B)	服务器*Y*	等待*U*			
			Read(A)	服务器X	等待*T*

事务*T*锁住了服务器*X*上的对象*A*，而事务*U*锁住服务器*Y*上的对象*B*。此后，当*T*试图访问服务器*Y*上的对象*B*时，要等待*U*在*B*上的锁。同样，事务*U*在访问服务器*X*的对象*A*时也需要等待*T*在*A*上的锁。从而，在服务器*X*上，事务*T*在事务*U*之前，而在服务器*Y*上，事务*U*在事务*T*之前。这种不同的事务次序导致事务之间的循环依赖，从而引起分布式死锁。有关分布式死锁的检测和解除问题在本章下一节讨论。一旦检测出死锁，必须放弃其中的某个事务来解除死锁。这时，协调者将被通知，并且它将放弃该事务涉及的所有参与者的事务。

13.4.2 时间戳排序并发控制

对于单服务器事务，协调者在事务开始运行时附上一个惟一的时间戳。串行等价性通过访问对象的事务按它们的时间戳次序提交对象的版本来完成。在分布式事务中，协调者必须保证每个事务附上全局惟一的时间戳。全局惟一的时间戳由事务访问的第一个协调者发给客户。若服务器上的对象执行了事务中的一个操作，那么事务时间戳被传给该服务器上的协调者。

分布式事务中的所有服务器共同保证事务执行的串行等价性。例如，如果在某个服务器上，由事务*U*访问的对象版本在事务*T*访问后提交；而在另一个服务器上，事务*T*和事务*U*又访问了另外的同一个对象，那么它们也必须按相同次序提交对象。为了保证所有服务器上的相同排序，协调者必须就时间戳顺序达成一致。时间戳是一个二元组<本地时间戳，服务器id>。时间戳排序的一致性基于比较。在时间戳的比较中，首先比较本地时间戳，然后比较服务器id。

即使各服务器之间的局部时钟不同步，也能保证事务之间的相同排序。但是为了提高效率，各协调者之间的时钟还是要求大致同步。如果是这样的话，事务之间的排序通常与它们实际开始的时间次序相一致。利用第10章中的局部物理时钟同步方法可以保证时间戳的基本同步。

529

当利用时间戳排序进行并发控制时，冲突在执行每个操作的时候解除。如果为了解决冲突需要放弃某个事务时，相应的协调者将被通知并且它将在所有参与者上放弃该事务。这样，如果事务能够坚持到客户发起提交请求命令时，这个事务总能提交。因此在两阶段提交协议

中，参与者通常发回同意提交的投票。参与者不同意提交的惟一情形是参与者在事务执行过程中崩溃过。

13.4.3 乐观并发控制

在乐观并发控制中，每个事务在提交之前必须首先进行验证。事务在验证开始时被附上一个事务序号，事务的串行化是根据这些事务序号的顺序实现的。分布式事务的验证由一组独立的服务器共同完成，每个服务器验证访问自己对象的事务。这些验证在两阶段提交协议的第一个阶段进行。

考虑两个事务T和U的交替执行，它们分别访问服务器X和Y上的对象A和B：

T		U	
Read(*A*)	服务器X上	*Read*(*B*)	服务器Y上
Write(*A*)		*Write*(*B*)	
Read(*B*)	服务器Y上	*Read*(*A*)	服务器X上
Write(*B*)		*Write*(*A*)	

在服务器X上，事务T在事务U之前访问对象；在服务器Y上，事务U在事务T之前访问对象。如果现在事务T和U同时开始验证过程，但服务器X首先验证T，而服务器Y首先验证U。在12.5节中介绍的简化的验证协议要求一次只能有一个事务执行验证和更新阶段，这样，服务器在一个事务完成验证前不能验证其他事务，从而造成提交死锁。

12.5节中介绍的验证协议假设验证过程很快，这在单服务器事务时是成立的。但在分布式事务中，由于两阶段提交协议需要一定的时间，在获得一致提交决定之前，可能延迟其他事务进入验证过程的时间。在分布式乐观并发控制中，每个服务器使用并行的验证协议。这是对向前及向后验证的扩展，允许多个事务同时进入验证阶段。在这种扩展验证中，向后验证除了检查规则2，还必须检查规则3。也就是说，必须检查验证事务的写集合和较早的重叠事务的写集
530 合，看二者是否重叠。Kung和Robinson[1981]在他们的论文中叙述了并行验证过程。

如果使用了并行验证，事务就不会在提交过程中出现死锁。然而，如果服务器只是简单地进行独立验证，同一个分布事务的不同服务器可能按不同的次序来串行化同一组事务，例如，在服务器X上先执行T再执行U，在服务器Y上先执行U再执行T。

分布式事务的服务器必须防止这种情况的发生。一个解决方案是在每个服务器完成局部验证后，再执行一个全局验证[Ceri and Owicki 1982]。全局验证用来检查每个服务器上的事务执行次序是否可全局串行化，换言之，这些事务不会形成验证环路。

另一种方案是让分布式事务的所有服务器在验证开始时使用一个全局惟一的事务序号[Schlageter 1982]。两阶段提交协议的协调者负责生成全局惟一的事务序号，并将此事务序号通过*canCommit?*消息传给参与者。由于不同的服务器会协调不同的事务，这些服务器必须像在分布式时间戳排序协议中一样按一致的顺序生成事务序号。

Agrawal等人[1987]提出了Kung和Robinson算法的一个变种，这个变种对只读事务进行了优化，并且还结合了一种MVGV（多版本通用验证）算法。MVGV是一种并行验证，它确保事务序号反映了串行化次序，但是它要求延迟某些事务在提交之后的可见性。同时，MVGV还允许事务序号的改变，这样可以使更多的即将失败的事务执行验证。Agrawal等人的论文还提出了一种用于提交分布式事务的算法。与Schlageter的方案类似，同样需要全局惟一的事务

序号。在读阶段结束时，协调者发布一个全局事务序号，然后每个参与者就试图用这个事务序号来验证它们的局部事务。但是，如果发布的全局事务序号太小，某些参与者不能验证自己的事务，那么它会通知协调者要求增大事务序号。如果不能找到合适的事务序号，那么参与者只能放弃事务。最终如果所有的参与者都能够验证它们的事务，协调者将收到每个参与者发来的事务序号，如果这些事务序号相同，那么事务就能提交。

13.5 分布式死锁

在12.4节中有关死锁问题的讨论表明，单服务器在使用锁机制的并发控制时可能出现死锁。服务器要么防止死锁发生，要么检测并解除死锁。采用超时的方法来解除死锁是一种麻烦的方法——主要是设定合适的超时间隔很困难，它会导致事务不必要地放弃。通过死锁检测方法，只有死锁中的事务才被放弃。大多数死锁检测方法都是通过在事务等待图中寻找环路实现的。在包含多事务访问多服务器的分布式系统中，全局等待图在理论上可以通过局部等待图构造出来。在全局等待图中可能会出现在局部等待图中并不存在的环路，也就是说，可能出现分布式死锁。等待图是有向图，其结点由事务和对象组成，边表示事务拥有某个对象或者事务正在等待对象。死锁出现的充要条件是等待图中存在一个环路。

图13-12表示了3个事务*U*、*V*和*W*之间的交替执行，它们分别访问服务器*X*上对象*A*和服务器*Y*上对象*B*，以及服务器*Z*上对象*C*和*D*。

图13-13（a）的等待图显示了一个死锁环路由不同的边组成，分别代表某个事务等待某一对象以及某一对象被某个事务持有。由于任何事务一次只能等待一个对象，因此可以在死锁环路中移走对象结点，从而将等待图简化为13-13（b）。

U		*V*		*W*	
d.deposit(10)	锁住*D*				
		b.deposit(10)	在服务器*Y*锁住*B*		
a.deposit(20)	在服务器*X*锁住*A*				
				c.deposit(30)	在服务器*Z*锁住*C*
b.withdraw(30)	在服务器*Y*等待				
		c.withdraw(20)	在服务器*Z*等待		
				a.withdraw(20)	在服务器*X*等待

图13-12 事务*U*、*V*和*W*的交替执行

分布式死锁的检测要求在分布于多个服务器上的全局等待图中寻找环路。第12章提到局部等待图可以在每一服务器上由锁管理器构造。在上面的例子中，各服务器的局部等待图为：

服务器*Y*：*U*→*V*（在*U*请求*b.withdraw*(30)时出现）

服务器*Z*：*V*→*W*（在*V*请求*c.withdraw*(20)时出现）

服务器*X*：*W*→*U*（在*W*请求*a.withdraw*(20)时出现）

由于每个服务器都构造出全局等待图的一部分，因此各服务器之间通过通信才能发现图中的

环路。

一种简单的方法是使用集中式死锁检测，由其中的一个服务器担任全局死锁检测器。全局死锁检测器通过收集合并各服务器发送的最新的局部等待图的拷贝来构造全局等待图。全局死锁检测器在全局等待图中检查环路。一旦它发现了环路，就要决定如何解除死锁，并通知各服务器通过放弃相应事务来解除死锁。

532

集中式的死锁检测并不是一个好的方法，最主要的问题是它依赖单一的服务器执行检测。因此它和分布式系统中其他集中式解决方案一样，可用性较差，缺乏容错，没有可伸缩性。并且，频繁地传输局部等待图代价很大。如果不频繁地收集全局等待图，那么可能需要更长的时间才能检测出死锁。

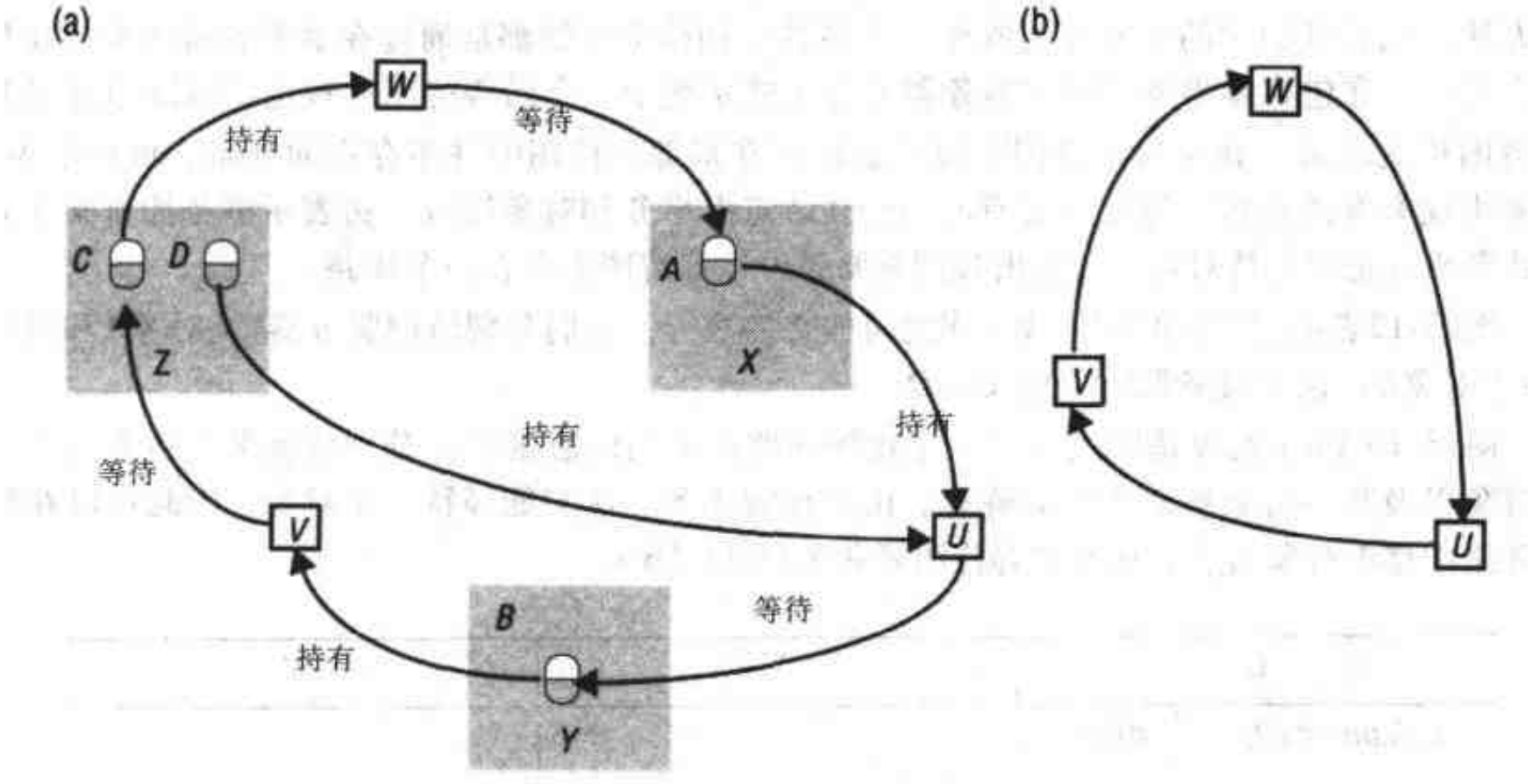


图13-13 分布式死锁

假死锁 如果“检测”算法检测出的死锁并非真正的死锁，那么这个死锁被称为“假死锁”。在分布式死锁检测中，等待图的信息在服务器之间传递。如果确实存在死锁，那么最终有一个结点有足够的信息来发现环路。但是由于收集过程需要一定的时间完成，在这段时间内，可能有的事务已经放弃了某些锁，这种情况下死锁就不存在了。

考虑图13-14中的情景，一个全局死锁检测器收到来自服务器X和Y的局部等待图。假设此时事务U释放了服务器X上的对象，并且请求服务器Y上被事务V拥有的对象。假设全局检测器先收到服务器Y的等待图，再收到服务器X的等待图。此时，尽管$T \rightarrow U$并不存在，它仍然检测出环路$T \rightarrow U \rightarrow V \rightarrow T$，这就是一个假死锁。

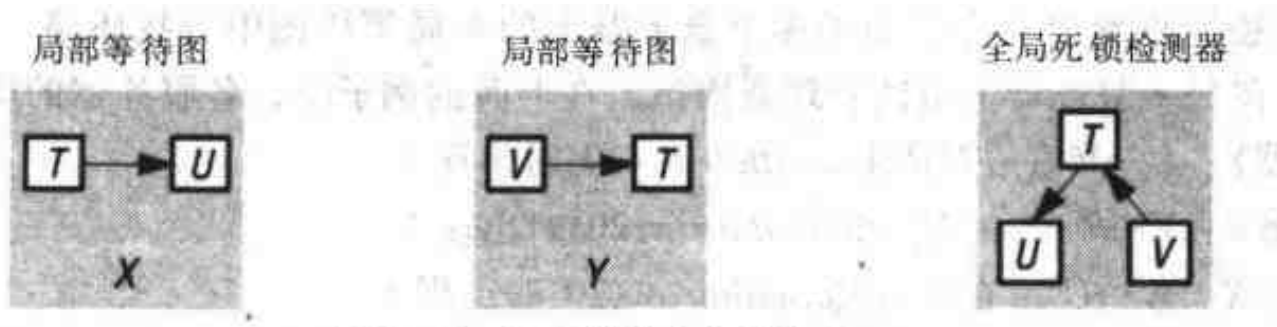


图13-14 局部等待和全局等待图

细心的读者可能意识到在采用两阶段加锁的情况下，事务不能释放对象也就不能获取新的对象，因此假死锁也就不会出现。现在来考虑检测到环路$T\rightarrow U\rightarrow V\rightarrow T$之后，要么表明这是一个死锁，要么表明$T$、$U$和$W$最终都要提交。但实际上，它们中的任何一个都不能提交，因为它们彼此相互永远等待不会释放的对象。

如果等待环路中的某个事务被放弃，那么也有可能检测出假死锁。例如，如果检测出死锁环路$T\rightarrow U\rightarrow V\rightarrow T$后，事务$U$被放弃，由于环路被打断，也就不存在死锁。

边追逐方法 另一种分布式死锁检测方法称为边追逐方法或路径推方法。在这种方法中，不需要构造全局等待图，但是每个服务器都有很多边的信息。服务器通过转发probe（探查）消息来发现环路，整个分布式系统的图的边都带有探查消息。一个探查消息包含了全局等待图中表示路径的一个事务等待关系。

问题是：探查消息将在何时被发出？考虑图13-13中服务器X的情形。此时该服务器刚刚在它的局部等待图中加入边$W\rightarrow U$，并且与此同时，事务U正在等待访问对象B，而服务器Y上的对象B正被事务V使用。这条边可能是环路$V\rightarrow T_1\rightarrow T_2\rightarrow\ldots\rightarrow W\rightarrow U\rightarrow V$的一部分。这意味着可能存在分布式死锁循环，可以通过向服务器Y发送探查消息找到这个分布式死锁循环。

现在来考虑当服务器Z将边$V\rightarrow W$加入它的局部等待图之前的情景：此时，W并没有等待。因此不需要发送探查消息。

每个分布式事务在某个服务器（被称为事务的协调者）上启动，并在若干个服务器（事务的参与者）之间移动，每个参与者和协调者进行通信。任何时刻，事务或者是活动的，或者在某个服务器上正在等待。协调者负责记录事务是活动的还是正在等待某个对象，并且参与者可以从它们的协调者处获取这些信息。锁管理器在事务开始等待对象时通知协调者，同样在事务获取对象而又成为活动事务时也通知协调者。当事务被放弃而打破死锁时，它的协调者将通知所有的参与者，所有的相关锁将被释放，与该事务有关的所有边也从局部等待图中删除。

533 ~ 534

边追逐算法由下面3步组成——开始阶段、死锁检测和死锁解除。

- 开始阶段 当服务器发现某个事务T开始等待事务U，而U正在等待另一个服务器上的对象时，该服务器将发送一个包含边$<T\rightarrow U>$的探查消息，启动一次检测过程，这个消息将发送到阻塞U的服务器。有时U和其他事务共享锁，那么探查消息将被转发到这些锁的持有者。有时有些事务可能会在稍后共享该锁，这时探查消息也将发送给这些事务。
- 死锁检测 死锁检测过程包含接收探查消息并确定是否有死锁产生，以及是否需要转发探查消息。

例如，当服务器对象接收到探查消息$<T\rightarrow U>$（表示T正在等待持有本地对象的事务U）时，那么它检查U是否也在等待。如果U也在等待另一个事务（以V为例），那么V就加到探查消息中（使它成为$<T\rightarrow U\rightarrow V>$），如果$V$在等待另外的对象，那么转发探查消息。

就这样，全局等待图上的路径被逐一构造出来。在转发探查消息之前，服务器将检测当事务（以T为例）加入到等待序列后是否造成探查消息产生环路（例如$<T\rightarrow U\rightarrow V\rightarrow T>$）。如果图中产生环路，那么就检测出死锁。

- 死锁解除 当环路被检测出来后，环路中的某个事务将被放弃以用来打破死锁。

在我们的例子中，下面的步骤描述了在对应的检测阶段，如何开始死锁的检测过程，以及转发的探查消息：

- 服务器X首先发起死锁检测过程，向对象B的服务器Y发送$< W \rightarrow U >$。
- 服务器Y收到探查消息$< W \rightarrow U >$后，发现对象B被事务V持有，因此将V附加在探查消息上，产生$< W \rightarrow U \rightarrow V >$。由于$V$在服务器$Z$上等待对象$C$，因此该探查消息被转发到服务器$Z$上。
- 服务器Z收到探查消息$< W \rightarrow U \rightarrow V >$，并且发现$C$被事务$W$持有，那么将$W$附加在探查消息后形成$< W \rightarrow U \rightarrow V \rightarrow W >$。

这个路径上包含一个环路，服务器就检测到一个死锁。必须放弃环路中的某个事务来解除死锁。可根据事务的优先级来选择被放弃的事务。

图13-15表示了对象A的服务器发起检测过程并最终在对象C的服务器上检测出死锁的过程。其中探查消息用带箭头的粗线表示，对象用圆圈表示，事务协调者用矩形表示。每个探查消息直接连接两个对象。在实现中，在服务器发送探查消息到另一个服务器之前，它首先将询问等待路径上最后一个事务的协调者，来确定该事务是否在等待其他对象。例如，在对象B的服务器发送探查消息$< W \rightarrow U \rightarrow V >$之前，它询问$V$的协调者来确定$V$正在等待对象$C$。在绝大多数边追逐算法中，对象所在的服务器通常向事务协调者发送探查消息，然后事务协调者再将消息转发到事务等待对象所在的服务器。在我们的例子中，对象B的服务器发送探查消息$< W \rightarrow U \rightarrow V >$到$V$的协调者，然后$V$的协调者再将其转发到$C$的服务器。这表明，转发一个探查消息需要发送两个消息。

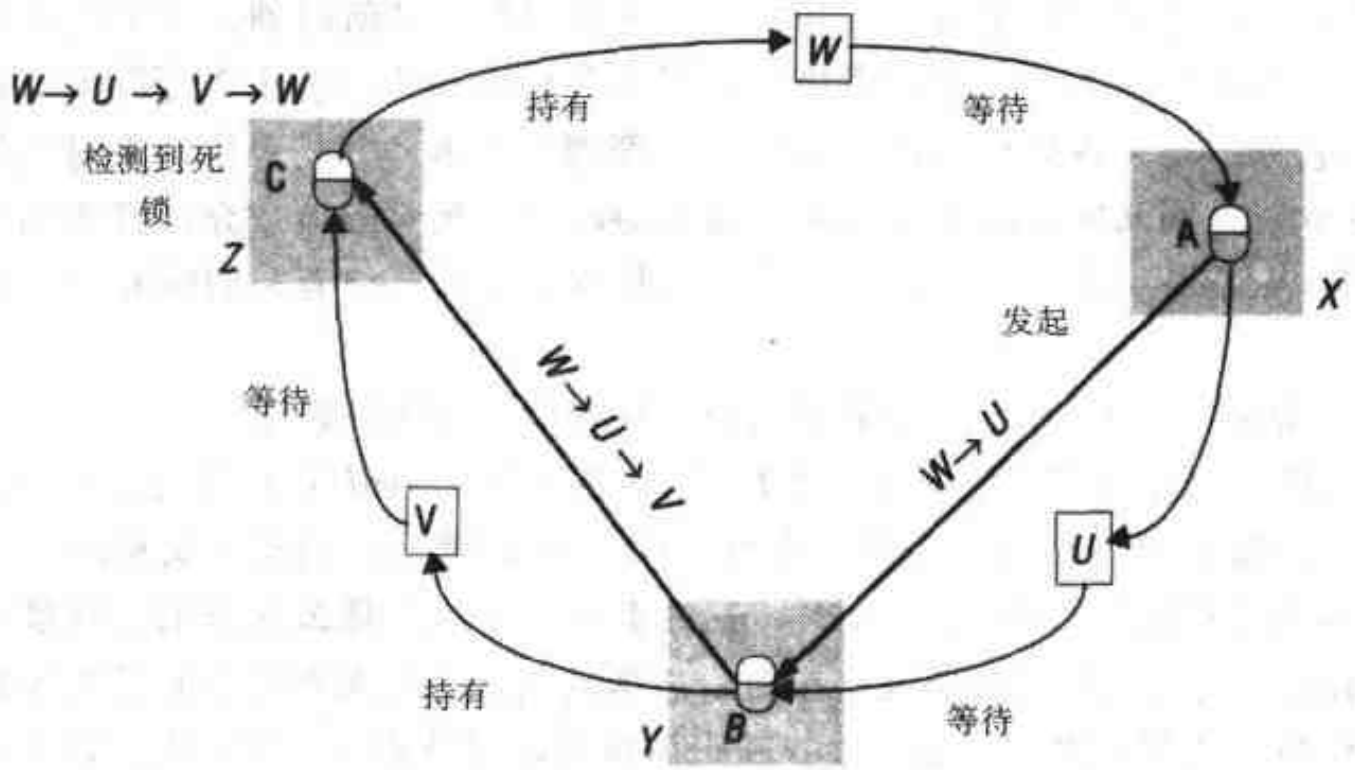


图13-15 探查消息的传递和死锁检测

假设等待的事务没有放弃，并且不会丢失消息，服务器也不会崩溃，那么上面的算法能够找到任何出现的死锁。为了理解这一点，考虑一个死锁环路，其中最后的事务W开始等待并且闭合该环路。当W开始等待某个对象时，服务器开始发出一个探查消息传给持有W正在等待对象的服务器。探查消息的接收者扩展这个消息并转发给它们发现的所有等待事务请求的对象所在的服务器。因此所有W直接或者间接等待的事务将最终加到探查消息中，除非检测出死锁。当死锁出现后，W就间接地等待自己。这样，探查消息将返回到W持有的对象处。

为了检测死锁，需要发送大量的消息。在上面的例子中，我们看到为了检测出3个事务的

死锁需要发送两个探查消息，而每个探查消息通常需要两个消息（从对象发送到协调者，再由协调者发送到对象）。

如果一个检测环路的探查消息涉及N个事务，那么这个探查消息需要被$(N-1)$个事务协调者转发，并且经过$(N-1)$个对象的服务器，因此最终需要$2(N-1)$个消息。幸运的是，绝大多数的死锁只涉及两个事务，因此不用考虑过量的消息。这一结论来源于数据库领域的研究，它也可扩展到计算对象的冲突访问的概率问题上，参见Bernstein等[1987]。

事务优先级 在上面的算法中，死锁包含的每个的事务都可能发起死锁检测。这样多个事务同时发起死锁检测会造成死锁在多个服务器上被检测出来，当解除死锁时会使多个事务被放弃。

535 ~ 536

在图13-16（a）中，考虑事务T、U、V和W，事务U正在等待事务W，事务V正在等待事务T。几乎在同一时刻，T请求U持有的对象并且W请求V持有的对象。两个独立的探查消息$<T \rightarrow U>$和$<W \rightarrow V>$由这些对象的服务器同时发起和转发，最终有两个不同的服务器检测出死锁。在图13-16（b）中，等待环路是$<T \rightarrow U \rightarrow W \rightarrow V \rightarrow T>$，在图13-16（c）中，等待环路是$<W \rightarrow V \rightarrow T \rightarrow U \rightarrow W>$。

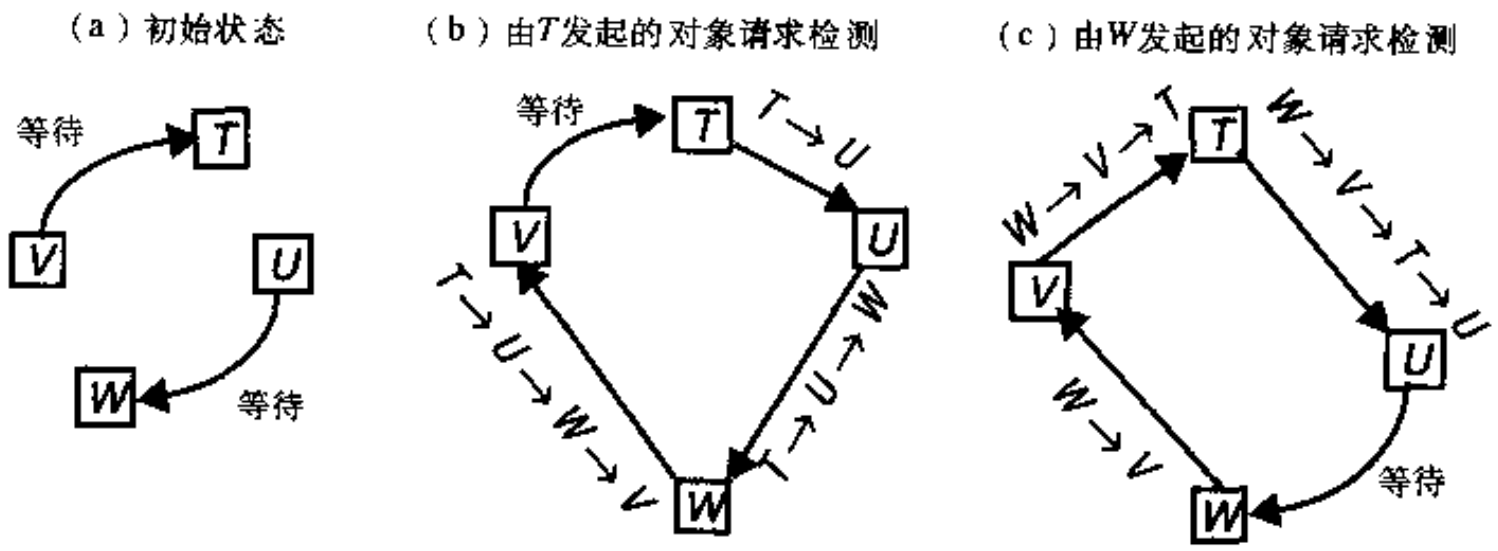


图13-16 同时发起两个探查消息

为了保证环路中只有一个事务被放弃，每个事务都被附上一个优先级，这样事务之间就建立了一个全序关系。例如时间戳就可以作为事务的优先级。当检测出死锁环路后，具有最低优先级的事务被放弃。这样尽管若干个不同的服务器同时检测出死锁环路，但对于“放弃哪一个事务”的问题它们仍然可以做出一致的决定。我们用$T>U$表示T的优先级高于U。在上面的例子中，假设$T>U>V>W$，那么不管是检测到环路$<T \rightarrow U \rightarrow W \rightarrow V \rightarrow T>$还是环路$<W \rightarrow V \rightarrow T \rightarrow U \rightarrow W>$，事务$W$都将被放弃。

如果要求死锁检测只有在高优先级的事务等待低优先级事务时才能发起，那么事务优先级也可以用来减少发起死锁检测的次数。在图13-16的例子中，由于$T>U$，所以探查消息$<T \rightarrow U>$被发出，而由于$W<V$，因此探查消息$<W \rightarrow V>$不能发出。如果我们假设事务开始等待另一个事务时，高优先级的等待事务和低优先级的等待事务具有一半的概率，那么利用上面的死锁检测发起规则，可以减少一半探查消息。

事务优先级还可以用来减少探查消息转发的次数。一般的想法是探查消息只能“向下”传递——即从高优先级的事务到低优先级的事务。这样，服务器不会将探查消息转发到比发起者优先级还高的对象持有者事务。这是由于如果某一对象持有者正在等待另一个事务，那

537 么它在开始等待时一定已经发起了死锁检测。

然而，这种明显的改进存在一个缺陷。在图13-15的例子中，当事务W开始等待U时，事务U正在等待事务V，事务V正在等待W。如果不使用优先级的规则，在W开始等待时通过发送一个探查消息$< W \rightarrow U >$来发起死锁检测；如果使用优先级规则，因为$W<U$，所以不发出这个探查消息，因此死锁就不能检测出来。

这里的问题是事务开始等待的次序会决定死锁是否被检测出来。为避免以上的缺陷，协调者可以将所有收到的代表每一事务的探查消息保存到探查队列中。当事务开始等待一个对象时，它将探查消息转发到对象所在的服务器，由它将探查消息向下传递。

在图13-15的例子中，当U开始等待V时，V的协调者将保存探查$< U \rightarrow V >$，如图13-17（a）。接着当V开始等待W时，W的协调者将保存$< V \rightarrow W >$，并且V将它的探查队列$< U \rightarrow V >$发给W。图13-17（b）中，W的探查队列包含$< U \rightarrow V >$和$< V \rightarrow W >$。当W开始等待A时，它就转发它的探查队列$< U \rightarrow V \rightarrow W >$到$A$的服务器，该服务器发现新的依赖$W \rightarrow U$，并将它和已收到的信息合并，发现$U \rightarrow V \rightarrow W \rightarrow U$。这样死锁可以被检测出来。

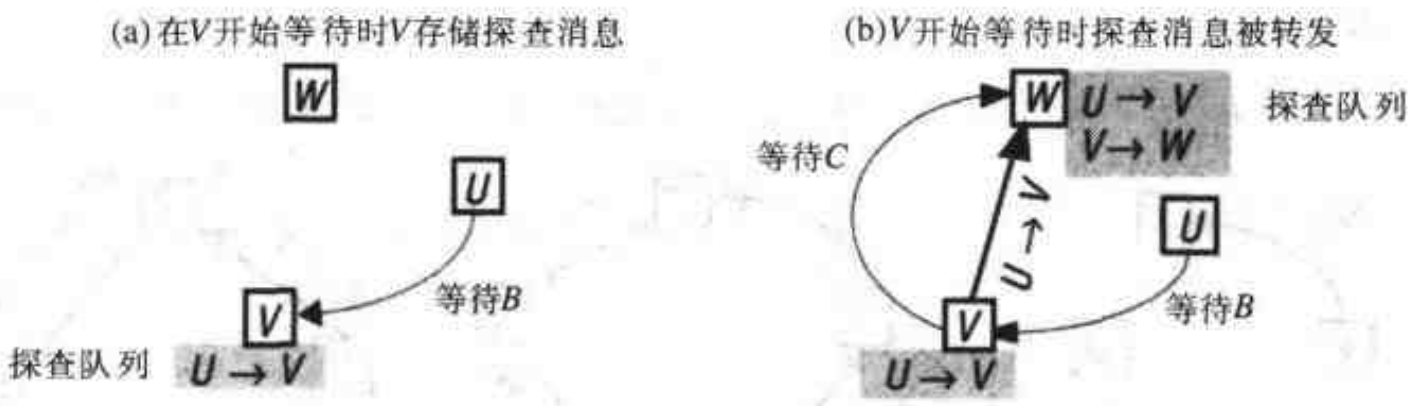


图13-17 探查消息向下传递

如果一个算法要求将探查消息存储在探查队列中，同样它也要求将探查消息传递到新的服务器并且当事务提交或放弃时，丢弃相关的探查消息。如果相关的探查消息被丢弃，某些死锁就有可能检测不到；另一方面，如果过期的探查消息仍然保留，就可能检测出假死锁。这样边追逐算法会变得很复杂。有兴趣的读者可参阅[Sinha and Natarajan 1985]和[Choudhary *et al.* 1989]了解这些算法的细节，其中介绍了使用排它锁的算法。Choudhary等人指出Sinha和Natarajan的算法是不正确的，该算法不能检测出所有的死锁，并且还会发现一些假死锁。Choudhary等人的算法仍然存在这些问题。Kshemkalyani和Singhal[1991]更正了Choudhary等

538 人的算法，并且提供了一个关于更正后的算法的正确性证明。在随后的文献中，Kshemkalyani和Singhal [1994]指出由于分布系统中不存在全局状态或时间，因此理解分布式死锁有一定的困难。事实上，任何一个收集到的环路所包含的信息记录自不同的时间。另外，在死锁发生时结点可以立刻获得死锁的信息，但是这些结点的解除死锁的信息却会被延迟任意时间。该文献利用在不同场地的事件之间的因果关系，描述了一种由分布式共享内存引起的分布式死锁。

13.6 事务恢复

事务的原子特性要求事务访问的对象能够反映出所有已提交事务的作用，而取消所有未完成或放弃的事务的作用。这个特性包含两方面的含义：持久性和故障原子性。持久性要求对象被永久地保存在持久存储器中并且一直可用。因此，如果客户提交请求一旦被确认，那

么事务的所有更新就被记录到持久存储器中或者服务器（易失性）对象中。故障原子性要求即使在服务器崩溃时，事务的作用也是原子性的。事务恢复的功能就是保证服务器上对象的持久性并提供故障原子性服务。

虽然文件服务器和数据库服务器将数据保存在持久存储器中，但其他类型的服务器上的可恢复对象不必如此保存，除非是为了恢复。本章假设当服务器运行时，它的所有对象都存放在易失性存储器中，而提交后的对象保存在一个或多个恢复文件中。这样，事务恢复过程实际上就是根据持久存储器中最后提交的对象版本来恢复服务器中对象的值。由于数据库需要处理大量数据，它们通常将对象保存在磁盘的稳定的存储器中，而在易失性内存中维护一个缓存。

持久性要求和故障原子性要求两者并非完全独立，它们可以利用统一的机制来解决，即利用恢复管理器。恢复管理器的任务是：

- 对已提交事务，将它们的对象保存在持久存储器（在一个恢复文件）中；
- 服务器崩溃后恢复服务器上的对象；
- 重新组织恢复文件以提高恢复操作的性能；
- 回收（恢复文件中的）存储空间。

在某些情况下，恢复管理器要求能够应对介质故障。所谓介质故障是指，由于系统崩溃时恢复文件受损或任意的延时或持久存储故障，造成恢复文件故障，以至于磁盘数据丢失。此时我们就需要恢复文件的副本。当然，这也可以在稳定存储器中实现，如利用位于不同场地的镜像磁盘或异地备份可以降低持久存储器出现故障的几率。 539

意图列表 每一个支持事务的服务器都需要记录被客户事务访问的对象。第12章提到，当客户创建一个事务时，首先与之联系的服务器将提供一个新的事务标识，并返回给客户。此后的每个客户操作，包括最后的提交和放弃操作，都要将这个事务标识作为一个参数传递。在事务的进行过程中，所有的更新操作都是针对该事务私有的临时版本对象集进行。

在每个服务器上，意图列表用来记录该服务器上的所有活动事务，而对于每个特定事务，意图列表都记录了该事务修改对象的值和引用的列表。当事务提交时，它的意图列表被用来确定所有受影响的对象，然后将该事务的临时版本替换对象的正式版本，并将对象的新值写入到恢复文件中。当事务放弃时，服务器利用意图列表来删除该事务所有临时对象。

前面介绍过分布式事务在提交和放弃时必须执行一个原子提交协议。我们论及的恢复基于两阶段提交协议：首先所有的参与者投票表决是否准备提交；如果都准备好，那么它们统一执行真正的提交动作。如果有参与者不同意提交，那么该事务必须放弃。

一旦某个参与者表示已准备好提交，那么它的恢复管理器必须将意图列表和对象的临对版本都保存到恢复文件中，此后不管中途是否出现崩溃故障，它总能完成提交动作。

如果分布式事务中的所有参与者一致同意提交，那么协调者将向所有的参与者发送提交命令并通知客户。当客户得知事务已被提交时，参与者的恢复文件必须保存足够的信息。这样不管服务器在准备好提交和提交之间出现崩溃故障，都能保证事务最终完成提交。

恢复文件中的内容 为了处理分布式事务的恢复问题，除了保存对象值外，还需在恢复文件中保存其他信息。这些信息和事务状态相关，即事务是否处于“已提交”、“已放弃”还是“准备好”状态。另外，恢复文件中的每一个对象都通过意图列表和某个特定的事务联系在一起。图13-18示意性地列出了恢复文件中的记录类型。

类　型	描　述
对象	某个对象的值
事务状态	事务标识，事务的状态（prepared, committed, aborted）以及其他用于两阶段提交的状态值
意图列表	事务标识和一系列意图记录，每个意图记录包含对象标识、对象值在恢复文件中的位置

图13-18　恢复文件中的记录类型

两阶段提交协议中的事务状态对提交协议恢复的作用将在13.6.4小节讨论。下面描述恢复文件的两种常用方法：日志方法和影子版本方法。

13.6.1　日志

在日志技术中，恢复文件包含了该服务器上执行的所有操作的历史。操作历史由对象值、事务状态和意图列表组成。日志中的次序反映了服务器上事务准备好、已提交或放弃的次序。

在服务器的正常操作过程中，每当事务准备提交、提交或放弃时，恢复管理器就被调用。当服务器准备提交某个事务时，恢复管理器将所有意图列表中的对象添加到恢复文件，另外事务的当前状态（准备好）和意图列表也被添加到文件。当该事务最终提交或放弃后，恢复管理器将事务相应的状态添加到恢复文件。

我们假定恢复文件的添加写操作是原子的，即它总是写入完整的内容。如果服务器崩溃，那么只有最后一次写操作可能不完整。为了提高磁盘的利用率，几次连续的写操作可能暂时缓冲在一起，然后通过一次操作写入恢复文件。日志技术的另一个优点就是顺序写盘操作要比随即写盘操作快。

所有未提交的事务在崩溃后将根据日志内容被全部放弃。因此，当事务提交时，它的“提交”状态应强制写入日志文件，连同其他缓冲的内容一并写入。

恢复管理器为每个对象附上惟一的标识符，这样在恢复文件中，对象的不同版本可以与服务器上的对象联系起来。例如，远程对象引用的持久形式（例如CORBA的持久引用）就可以作为对象标识符。

图13-19表示了图12-7银行业务中事务*T*和*U*的日志机制。日志文件被重新组织后，双线左边的内容表示事务*T*和*U*开始前对象*A*、*B*和*C*的值。本图直接利用*A*、*B*和*C*作为对象标识符。双线右边的内容表示了事务*T*已提交而事务*U*准备好但未提交的状态。在事务*T*准备提交时，对象*A*和*B*的值分别写入日志位置P_1和P_2处，紧接着事务状态和意图列表（$<A, P_1>$, $<B, P_2>$）也被写入日志。当*T*提交时，它的提交状态被写入位置P_4处。然后，在事务*U*准备提交时，对象*C*和*B*的值被分别写入位置P_5和P_6处，事务状态和意图列表（$<C, P_5>$, $<B, P_6>$）也被写入日志。

540～541

在日志文件中，每个记录都包含一个的指针，指向同一个事务的前一个记录位置。这样，恢复管理器可根据这个指针逆向读取某个事务的所有事务状态值。事务状态记录序列的最后一个指针指向检查点。

对象的恢复　当服务器进程因崩溃而被替换后，它首先将对象置为默认的初始值，然后将控制转给恢复管理器。恢复管理器的任务是恢复所有对象的值，使这些值反映按正确次序的所有已提交事务的效果，而不包含任何未完成或放弃事务的效果。

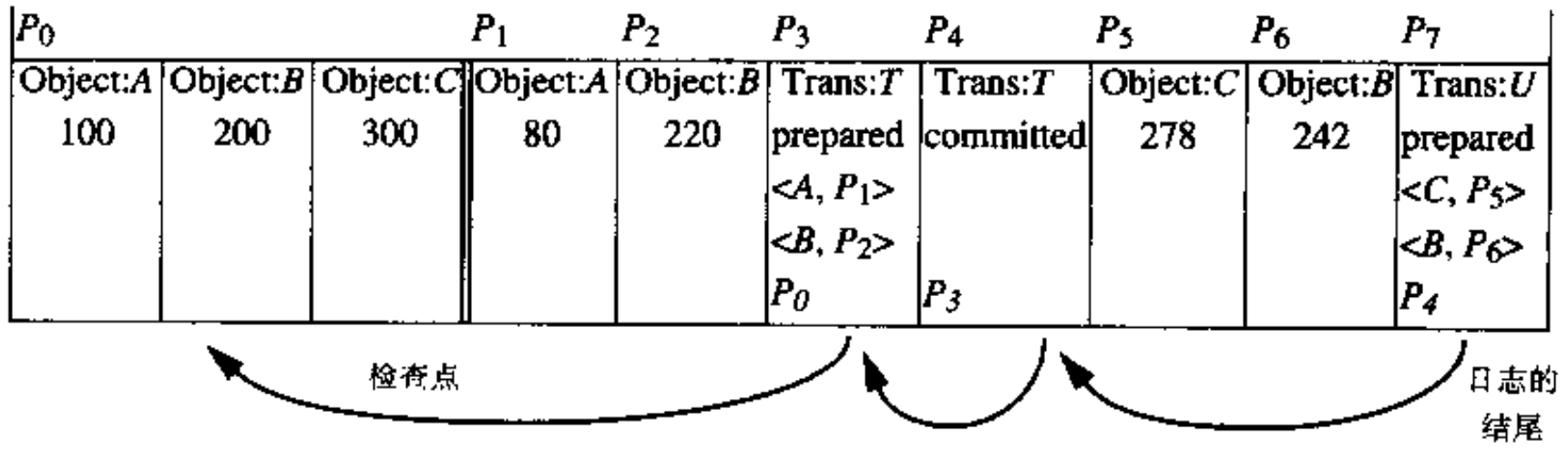


图13-19 银行服务例子的日志

有关事务最新的信息在日志的尾部。根据恢复文件来恢复数据有两种方法。一种方法是，恢复管理器将对象的值恢复到最近一次检查点时的值，同时对所有已提交事务更新对象值。这种方法按事务的执行次序来更新对象值，由于检查点离日志尾部可能很远，因此需要读取大量的日志记录。第二种方法是恢复管理器通过逆向读取日志文件来恢复对象值。根据日志文件中的向前指针，可以恢复所有提交事务更改的对象。这种方式的优点是每个对象只用恢复一次。

为了恢复已提交事务的效果，恢复管理器从日志文件中读取相应的意图列表。列表包含了所有更新对象的标识符和更新后对象值的日志位置。

以图13-19为例，如果服务器崩溃后日志文件的内容如图所示时，它的恢复管理器将按如下步骤进行恢复处理。首先它读取日志文件的最后一个记录（在P_7处），从而得知事务U尚未提交，它的更新应全部撤销。接着，它读取下一个事务状态记录（在P_4处），得知事务T已提交。为了恢复事务T的更新，恢复管理器读取在P_3处的前一个事务状态记录，获取T的意图列表（<A, P_1>, <B, P_2>），从而读取位置P_1和P_2处的记录恢复A和B的值。此时，还未恢复对象C的值，它移回到检查点P_0处，恢复C的值。 542

为了有助于后继的恢复文件重组，恢复管理器在以上过程中记录了所有准备提交的事务。对每个准备提交的事务，它在日志文件中追加一个放弃的状态记录。这样可以保证每个事务总是处于提交或放弃状态。

服务器在恢复过程中仍然可能发生故障，因此有必要保证恢复过程是幂等，即可以重复进行多次而保证执行效果不变。由于我们假设所有的对象都存储在内存中，因此恢复过程自然是可重复的。但在数据库系统中，由于数据保存在持久存储中，内存只有一个缓存，在服务器崩溃时，有些在持久存储中的对象可能已经过期。这样，它的恢复管理器必须恢复这些在持久存储中的对象。这样，它在恢复过程中又崩溃时，某些对象可能仍处于过期状态，这使达到幂等效果的难度稍微大一些。

恢复文件的重组 恢复管理器为了使恢复过程执行得更快或为了节省存储空间，有时需要重组恢复文件。如果恢复文件一直不重组，那么恢复过程可能需要逆向搜索整个恢复文件，直到找到对象的所有值为止。实际上，恢复过程需要的信息只需包含所有对象的提交后版本即可，这些值可用非常紧凑的方式存储。检查点过程就是用于这个目的。检查点过程是将当前所有已提交的对象版本写入一个新恢复文件的过程，同时还需写入的信息包括事务的状态记录和尚未完全提交事务的意图列表。检查点指该过程写入的信息。检查点的目的是减少恢复过程中需要处理的事务数目和回收文件空间。

检查点过程可以在恢复过程结束后新事务开始之前进行。但是，恢复过程并不经常发生。在服务器正常处理过程中需要不时进行检查点过程。检查点被写入另一个新的恢复文件，在检查点写入完毕后，当前恢复文件将不再使用。在检查点过程开始时，首先在恢复文件中做一个标记，然后将服务器的对象写入新的恢复文件，接着，复制标记点前与未完成事务有关的内容，以及标记点后的所有内容到这个新的文件中。当检查点完成时，这个新文件可被用于以后的操作。

恢复管理器通过丢弃旧的恢复文件来减少磁盘空间。当恢复管理器执行恢复过程时，可
543 能遇到检查点。一旦遇到检查点，它就立即根据检查点中的对象值来恢复对象。

13.6.2 影子版本

日志技术将事务状态信息、意图列表和对象记录在同一个日志文件中。影子版本是另一种恢复文件的组织方式。它利用一个映射表来定位在版本存储文件中的某个特定对象版本。这个映射表将对象标识符和对象当前版本在版本存储中的位置对应起来。每个事务写入的对象均是对象的影子版本。当使用影子版本方式的恢复处理时，事务状态和意图列表被分别对待。下面首先介绍对象的影子版本。

当事务准备提交时，该事务更新的所有对象被追加到版本存储中，并保留对象的原始版本不变。对象的这个新的临时版本被称为影子版本。当事务提交时，系统从旧映射表复制一个新的映射表并根据影子版本来更新对象的位置。接着，将这个新映射表替换旧映射表即完成提交过程。

当服务器崩溃后恢复对象时，恢复管理器读取映射表，并使用映射表中的信息来定位版本存储中的对象。

图13-20表示了事务T和U使用影子版本时的情况。表的第一列表示事务T和U运行之前的映射表，这时账户A、B和C的余额分别是100美元、200美元和300美元。表的第二列表示了事务T提交后的映射表。

事务开始时的映射	事务提交后的映射
$A \rightarrow P_0$	$A \rightarrow P_1$
$B \rightarrow P_0'$	$B \rightarrow P_2$
$C \rightarrow P_0''$	$C \rightarrow P_0''$

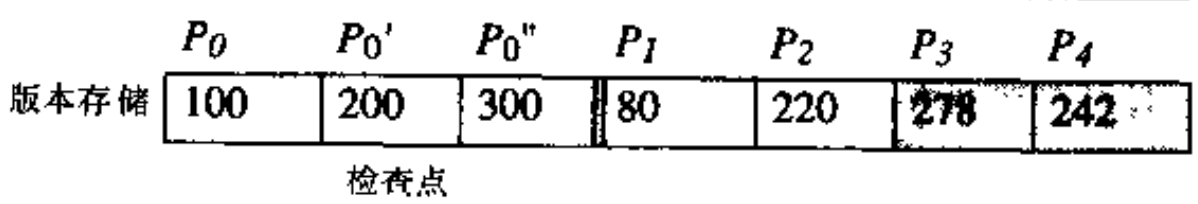


图13-20 影子版本

版本存储文件包含一个检查点，接着是事务T更新的对象A和B的版本，分别位于P_1和P_2处。它也包含事务U更新的对象版本，分别在P_3和P_4处。

映射表必须保存在一个已知的位置，例如版本存储的开始处或者一个独立的文件，这样当系统需要进行恢复时总能找到它。

事务提交时从旧映射表到新映射表的切换必须用一个原子步骤完成。为了保证这一点，必须将映射表放在持久存储中，这样即使写文件操作失败后仍然能保留有效的映射表。在恢

复过程中，由于影子版本方式记录了所有对象的最新提交版本，因此它比日志具有更好的性能。但在正常操作过程中，日志显示出更高的性能，这是因为日志只需向同一个文件追加日志记录，而影子版本需要写不相关的磁盘块。 544

影子版本本身并不适用处理分布事务的服务器。事务状态和意图列表被记录在事务状态文件中。每个意图列表代表了某个事务提交后会改变的部分映射。事务状态文件可能组织成日志。

下图给出了银行例子所使用的映射表和事务状态文件，这时，事务T已经提交，而事务U正准备提交。

映射表
$A \rightarrow P_1$
$B \rightarrow P_2$
$C \rightarrow P_0$''

持久存储

事务状态文件

T	T	U
准备好	已提交	准备好
$A \rightarrow P_1$		$B \rightarrow P_3$
$B \rightarrow P_2$		$C \rightarrow P_4$

在提交状态写入事务状态文件和映射表更新之间的时间段内，服务器有可能崩溃——此时，客户不会得到通知。恢复管理器必须处理这种情况，例如可以检查映射表是否包含最后提交事务的效果，如果没有，那么这个事务就应该标记成已放弃。

13.6.3 为何恢复文件需要事务状态和意图列表

使用不包含事务状态信息和意图列表的恢复文件进行恢复也是可能的，这种恢复文件用于单服务器上的事务比较合适。但对于参与分布事务处理的服务器来说，事务状态信息和意图列表是非常必要的。并且，对于非分布的事务，这种方法也是有益的，其原因有以下几点：

- 一些恢复管理器会较早地将对象写入恢复文件——假设事务会正常提交。
- 如果事务使用了很多大对象时，将这些对象连续地写入恢复文件会造成服务器设计的复杂化。如果对象在意图列表中的引用可以使用的话，那么对象可以存在恢复文件的任何地方。
- 在时间戳并发控制方法中，有时候服务器能够知道事务将最终提交，此时对象必须写入恢复文件（参见第12章）来保证持久性。但是，事务可能需要等待其他较早的事务提交。在这种情况下，恢复文件中相应的事务状态将是“等待提交”，然后是“提交”，以确保恢复文件中已提交事务的时间戳排序。在进行恢复时，任何一个等待提交的事务将被允许提交，这是因为它等待的事务或者已经提交，或者由于故障原因已被服务器放弃。 545

13.6.4 两阶段提交协议的恢复

在分布式事务中，每个服务器维护自己的恢复文件。前面介绍的恢复管理必须加以扩展，来处理两阶段提交过程中出现的服务器故障。这时恢复管理器会用到两个新的事务状态：“完成”和“不确定”。图13-6表示了这两个状态的含义。协调者用“已提交”状态来标记投票的结果是Yes，用“完成”状态表示两阶段提交协议已经完成。参与者用“不确定”状态表示它的投票是Yes但尚未收到事务的提交决议。另外，协调者需要记录所有的参与者，每个参与者需要记录协调者。

类型	描述
协调者	事务标识，参与者列表
参与者	事务标识，协调者

在协议的第一阶段，当协调者准备提交时（并且已经在恢复文件中添加了一个“准备好”状态记录），它的恢复管理器在恢复文件中添加一个“协调者”记录。每个参与者在它投Yes票之前，必须进入“准备好”状态，即在恢复文件中添加一个“准备好”记录。当它投Yes票时，它的恢复管理器在恢复文件中增加一个参与者记录，并写入“不确定”状态。当它投No票时，则在恢复文件中添加“已放弃”状态。

在协议的第二阶段，协调者的恢复管理器根据提交决议，在恢复文件中添加“已提交”或“已放弃”状态。这次添加必须是一次强制写入。参与者的恢复管理器根据收到的消息，在恢复文件中分别添加“已提交”或“已放弃”状态。当协调者收到所有参与者的确认消息之后，它的恢复管理器写入“完成”状态，这次写入不需要刷新。状态“完成”本身不是提交协议的一部分，但是使用它有利于组织恢复文件。图13-21表示了两阶段提交协议中日志文件的内容，其中服务器在事务*T*中扮演协调者角色，在事务*U*中扮演参与者角色。这两个事务的最初状态都是“准备好”。在事务*T*中，“准备好”状态记录之后跟着一个协调者记录和一个“已提交”状态记录（图中没有显示“完成”记录）。在事务*U*中，“准备好”状态记录之后跟着一个状态为“不确定”的参与者记录，接着是一个“已提交”或“已放弃”状态记录。

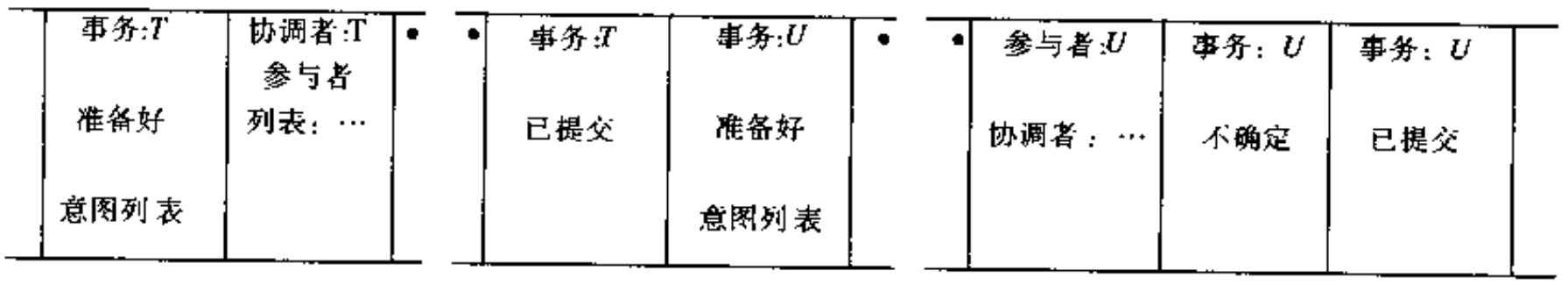


图13-21 与两阶段提交协议相关的日志记录

当服务器因崩溃而重启之后，恢复管理器除了需要恢复对象之外，还要处理两阶段提交协议。对任何一个服务器扮演协调者角色的事务，恢复管理器寻找协调者记录和事务状态信息。对任何一个服务器扮演参与者角色的事务，恢复管理器寻找参与者记录和事务状态信息。在这两种情况下，最新的事务状态信息——在日志的最后部分——反映了故障时的事务状态。此时，恢复管理器需要根据服务器是协调者或参与者以及状态不同，按图13-22进行恢复处理。

恢复文件的重组 在进行检查点的过程中，需特别注意不能将未完成的协调者状态从恢复文件中删除，这些信息必须在所有参与者确认之前一直保留。事务状态是“完成”的记录可以被丢弃。状态是“不确定”的参与者事务记录同样也必须保留。

嵌套事务的恢复 在最简单的情况下，嵌套事务的每个子事务访问不同的对象集。在两阶段提交过程中，每个参与者将它修改的对象和意图列表写入恢复文件，并且在这些记录上附上顶层事务的标识。尽管嵌套事务使用两阶段提交协议的变种，恢复管理器使用的事务状态和平面事务的情况是一样的。

但是如果不同层次上的子事务访问了相同的对象，那么事务放弃过程将变得复杂一些。12.4节讨论的锁机制支持父事务继承子事务的锁以及子事务从父事务处获取锁。这种锁方法要

求父事务和子事务在不同的时刻访问公共对象，并确保并发子事务对同一对象的访问必须是串行化的。

角　色	状　态	恢复管理器的动作
协调者	准备好	由于在服务器故障发生时尚未作出任何决定，因此它根据参与者列表向它们发送*abortTransaction*命令，并在恢复文件中记录一个已放弃记录。在已放弃状态下的操作也是这样，如果目前还没有参与者列表，那么参与者将由于超时最终放弃事务
协调者	已提交	在服务器故障发生时协调者已经作出决定要提交事务。因此它根据参与者列表向它们发送*doCommit*命令，继续执行两阶段提交协议的第4步骤
参与者	已提交	参与者向协调者发送*haveCommitted*消息。这允许协调者在下一个检查点处丢弃该事务的信息
参与者	不确定	参与者在获得事务提交决议之前发生故障，那么它在协调者通知它前不能确定事务的状态。因此参与者将向协调者发送*getDecision*请求来询问事务状态。当它获得应答后再提交或放弃事务
参与者	准备好	参与者尚未投票，它可以单方面放弃事务
协调者	完成	不需要任何操作

图13-22　两阶段提交协议的恢复

嵌套事务访问的对象由各子事务提供临时版本来保证其可恢复性。子事务使用的对象的不同临时版本之间的关系和锁之间的关系类似。为了支持事务放弃时的恢复，多个层次事务共享的对象按堆栈方式组织临时版本——每个嵌套事务使用一个堆栈。

每当嵌套事务的某个子事务第一次访问对象时，它提供当前提交版本对象的一个临时版本，并且这个临时版本被放置在栈顶。但是除非有其他的子事务访问同一个对象，否则这个堆栈实际上不需要产生。

当某个子事务访问同一个对象时，它将复制栈顶的版本并且重新入栈。所有的子事务更新都作用于栈顶的临时版本。当子事务临时提交后，它的父事务将继承新版本。为了实现这一点，子事务的版本和父事务的版本都从堆栈中丢弃，而将子事务的新版本重新放入堆栈（实际上替换了父事务的版本）。当子事务放弃后，它在栈顶的版本被丢弃。最终，当顶层事务提交时，栈顶版本成为新的提交版本。

例如在图13-23中，假设事务T_1，T_{11}，T_{12}和T_2按照T_1；T_{11}；T_{12}；T_2的顺序访问同一个对象A。设它们的临时版本分别是A_1，A_{11}，A_{12}和A_2。当T_1开始执行时，基于A提交版本的A_1被推入堆栈；当T_{11}开始执行时，基于A_1版本的A_{11}被推入堆栈，当它提交时，它替换父事务的对象版本（A_1）。事务T_{12}和T_2按相同方式执行，最终T_2的版本被留在栈顶。

546
|
548

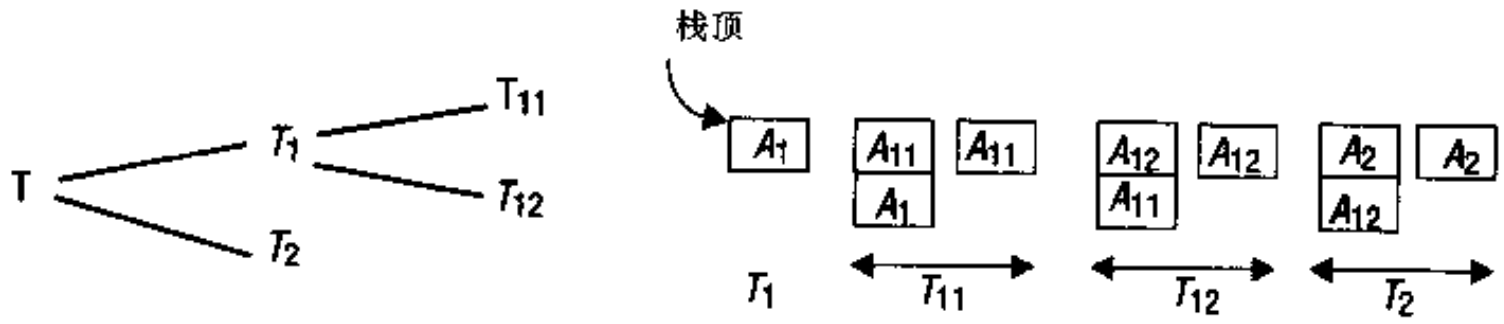


图13-23　嵌套事务

13.7 小结

通常情况下，一个客户发起的事务会操作多个不同服务器上的对象。一个分布式事务是指涉及多个不同服务器的事务。分布式嵌套事务在结构上支持更高的并发度，并允许独立提交。

事务的原子性要求分布式事务的所有参与者全部提交或者全部放弃。原子提交协议被用来保证这一点，即使在执行过程中出现服务器崩溃情况。两阶段提交协议允许任何一个服务器单方面放弃事务，它包含了一些超时操作来处理由于服务器崩溃造成的延时。两阶段提交协议不能保证在有限的时间期限内完成，但是它能确保最终完成。

分布式事务中的并发控制是模块化的——每个服务器负责访问它上面对象的事务的串行化。但是在保证事务全局串行化上需要额外的协议。分布式事务的时间戳方式需要一种在多个服务器之间保持惟一的时间戳生成方法。而在使用乐观并发控制时则需要一种全局验证方式。

利用两阶段加锁方式的分布式事务会导致分布式死锁。分布式死锁检测的目的是在全局等待图中寻找环路。一旦某个环路被发现，一个或者多个事务必须被放弃来解除死锁。边追逐方法是分布式死锁检测的一个非集中方法。

基于事务的应用通常对于长生命周期和存储信息的完整性方面有很强需求，但很多情况下它们不需要很快的响应时间。原子提交协议是分布式事务的关键，但它们不能保证在特定的时间限制内完成。事务的持久性是通过记录检查点和在恢复文件中记录日志完成，当服务器崩溃后被新的进程取代后，检查点和恢复文件被用来进行恢复处理。进行恢复处理过程中，用户会
549 感受到可观的延迟。尽管分布式事务运行在异步系统中，且服务器可能随时出现崩溃，由于新启动的进程可以从持久存储中获取必要的信息，因此这些服务器仍然能够达到一致决议。

练习

13.1 两阶段提交协议的一个非集中方式的变种是让各个参与者直接通信，而不是利用协调者进行间接通信。在第一阶段，协调者将它的投票发给所有的参与者。在第二阶段，如果协调者的投票是*No*，那么参与者只是简单地放弃事务；如果投票是*Yes*，那么每个参与者将它的投票发送给协调者和其他参与者，它们各自根据收到的投票来决定是否提交。请计算这种协议需要几轮消息发送以及消息总数，并将它和标准的两阶段提交协议进行比较，列出优缺点。

13.2 三阶段提交协议由以下步骤组成：

第一阶段：与两阶段提交协议相同。

第二阶段：协调者收集到所有投票后做出提交决定。如果决定是*No*，它放弃事务并通知所有投票为*Yes*的参与者；如果决定是*Yes*，它向所有的参与者发送*preCommit*请求。每个投票为*Yes*的参与者都等待*preCommit*或*doAbort*请求。接收到*preCommit*请求后它们会加以确认，收到*doAbort*请求后将放弃事务。

第三阶段：协调者等待收集确认消息。一旦收集到所有的确认，它就提交事务并且向参与者发送*doCommit*请求。每个参与者等待*doCommit*请求，该请求到来后就提交事务。

假设通信没有故障，请阐述上面的协议是如何避免参与者在不确定状态下的延时（由于协调者或参与者出现故障）？

13.3 试解释使用两阶段提交协议，一旦嵌套事务的顶层事务成功提交，如何保证所有的子事务都能提交？

13.4 请举例说明，两个分布式事务在每个服务器上都是串行的，但是在全局上不是串行的？

13.5 图13-4中定义的*getDecision*函数仅由协调者提供。请定义一个新的*getDecision*函数，该函数在协调者不可用时，由参与者提供并且被其他参与者使用？

假设每个参与者都可以调用其他参与者的*getDecision*函数，那么这样是否能解决不确定状态引起的延时问题？请解释你的答案。为了支持上面的通信，协调者在两阶段提交协议的什么时刻将所有参与者的标识通知给每个参与者？ 550

13.6 扩充两阶段加锁方法来支持分布式事务，并解释为什么局部使用严格的两阶段加锁就能够保证分布式事务的串行化？

13.7 假设系统使用严格的两阶段加锁方法，试描述两阶段提交协议和每个服务器的并发控制之间的关系。分布式死锁检测如何实现？

13.8 一个服务器用时间戳方式进行局部并发控制，在参与分布式事务处理时需要做哪些变化？在什么条件下，可以不必再用两阶段提交协议。

13.9 请考虑分布式乐观并发控制，其中每个服务器在局部顺序使用向后验证，请将答案与图13-6相对应。试描述两个并发事务试图同时提交时的所有可能情况。服务器采用并行验证时处理有何不同？

13.10 一个集中式全局死锁检测器收集所有的局部等待图。请给出一个例子说明：当一个死锁环路中的等待事务放弃后，这个死锁检测器可能发现假象死锁。

13.11 考虑无优先级的边追逐算法，请用例子说明它可能检测出假象死锁。

13.12 一个服务器管理对象$a_1, a_2, \ldots, a_n$，它向客户提供下面两个操作：

Read(*i*)返回对象a_i的值；

Write(*i*, *Value*)将值*Value*赋给对象a_i。

事务*T*、*U*和*V*定义如下：

T：*x* = *Read*(*i*); *Write*(*j*, 44);

U：*Write*(*i*, 55); *Write*(*j*, 66);

V：*Write*(*k*, 77); *Write*(*k*, 88);

试描述这3个事务写日志文件的情况，其中系统使用严格两阶段加锁，并且*U*在*T*之前访问a_i和a_j。请描述在服务器崩溃后，恢复管理器如何利用日志文件中的内容来恢复*T*、*U*和*V*的执行效果。阐述日志文件中提交记录次序的重要性。

13.13 向日志文件追加记录是原子操作，但是来自不同事务的追加记录操作可能相互交错。请阐述这种交错对练习13.12的影响。 551

13.14 练习13.12中的事务*T*、*U*和*V*用两阶段加锁，所进行的操作如下：

T	*U*	*V*
x = *Read*(*i*)		
		Write(*k*, 77)
	Write(*i*, 55)	
Write(*j*, 44)		
		Write(*k*, 88)
	Write(*j*, 66)	

假设恢复管理器在每次写操作后就立即将记录写到日志文件中，试描述日志文件中有关T、U和V的日志记录排列情况。这种立即写日志的方法是否会影响恢复过程的正确性？这种方法的优缺点如何？

13.15 事务T和U的并发控制利用时间戳方法，事务U的时间戳晚于事务T，因此必须等待T提交。试描述日志文件中有关事务T和U的日志记录排列情况，解释为什么日志文件中的提交记录必须按时间戳顺序排列？考虑下面两种情况下服务器如何恢复：（1）在两个事务提交之间服务器崩溃；（2）服务器在两个事务提交后崩溃。

T	V
$x = Read(i)$	
	$Write(i, 55)$
	$Write(j, 66)$
$Write(j, 44)$	
	$Commit$
$Commit$	

请解释在使用时间戳方法时提前写日志的优缺点。

13.16 练习13.15中的事务T和U采用乐观并发控制，并使用向后验证，验证失败时重新运行事务。请描述日志文件中这两个事务的日志记录排列情况，解释为什么日志文件中的提交记录必须按事务序号排列？在日志文件中的已提交事务的写集合是怎样的？

13.17 假设在两阶段提交协议运行过程中，协调者在意图列表记录到日志文件之后，但是尚未记录参与者列表或尚未发送*canCommit*请求之前，协调者崩溃。请描述参与者如何发现并
552 解决这种情况，协调者如何进行恢复？试问在记录意图列表之前先记录参与者列表是否更好？

第14章 复 制

14.1 简介
14.2 系统模型和组通信
14.3 容错服务
14.4 高可用服务
14.5 复制数据上的事务
14.6 小结

在分布式系统中，复制是提供高可用性和容错的关键技术。由于受移动计算和与此相关的断接操作的影响，高可用性正越来越引起人们广泛的兴趣；而容错在安全是关键要素的系统中通常作为一项必备的服务被提供。

本章的第一部分讨论了这样一个系统，它的每个操作都作用于复制对象的集合。本章开始描述了一个提供复制服务的体系结构组件和系统模型。我们还描述了对容错服务至关重要的组成员关系管理，它是组通信的一部分。

本章接着描述了实现容错的各种方法。首先介绍了线性化和顺序一致性的正确性标准。接着介绍并讨论了如下两种方法：被动（主备份）复制（客户与单个副本进行通信），主动复制（客户通过组播与所有副本进行通信）。

本章介绍了关于高可用性服务的3种系统。在gossip和Bayou体系结构中，共享数据各个副本之间的更新操作是延时传播的。在Bayou中，为了增强一致性，使用了操作变换技术。Coda是高可用文件系统的一个例子。

本章的结尾部分涉及了复制对象上的事务，详细阐述了复制事务系统的体系结构以及这些系统是如何处理服务器故障和网络分区的。

14.1 简介

本章研究数据的复制，即如何在多个计算机中进行数据副本的维护。由于复制能够增强性能，提供高可用性和容错能力，因此它是保证分布式系统有效性的一个关键技术。复制技术的使用非常广泛。例如，网络服务器的资源在浏览器上的缓存和在网络代理服务器上的缓存都属于复制形式，这是由于缓存中的数据是服务器中的数据的复制。第9章叙述的DNS名字服务维护关于计算机的名字－属性映射的副本，它是依赖于在因特网上的每日服务访问实现的。

复制是一种增强服务的技术。进行复制的动机包括改善服务性能，提高可用性，或者增强容错能力。

- 增强性能　迄今为止，客户和服务器的数据缓存是增强性能的常用手段。例如，第2章曾经提到，浏览器和代理服务器都对网络资源进行缓存以避免从原始服务器上读取数据。进而，数据有时还在同一个域中的多个原始服务器之间隐式复制。通过将所有服务器IP地址绑定到站点的DNS名字，负载在服务器之间能够得以平衡。以域名*www. aWeb Site. org*为例，当解析*www.aWebSite.org*域名时，将以循环方式返回几个服务器IP地址中的一

个（参见9.2.3）。不可变数据的复制是很简单的：它仅需花费极小的代价即可提高性能。变化数据(如Web数据）的复制需要额外的开销，来确保客户接受最新数据（参见2.2.5）。因此，作为性能增强的一项技术，复制有效性有一些限制。

- 提高可用性　用户要求服务是高度可用的，这就是说，在合理的响应时间内获得服务所占用的时间比例应该接近100%。除了由于悲观并发控制冲突（数据加锁）等原因造成的延迟外，同高可用性有关的有如下因素：
  - 服务器故障；
  - 网络分区和断接操作：通信断连通常是不可预计的，也可能是用户移动性的副作用。

对前一个问题，复制是一项避免服务器故障的技术，它能够自动维护数据的可用性。如果数据被复制到两个或者多个故障独立的服务器上，那么，客户软件就可能在默认服务器错误或者不可访问的情况下，通过其他备用服务器上获取数据。这就是说，通过复制服务器数据，服务器可用时间的比率就能够增加。如果n个服务器中的每一个都有独立的发生故障或者不可访问概率p，那么在所有服务器上都被保存的对象的可用性概率就是：

554

$$1 - \text{概率(所有服务器出故障或不可用)} = 1 - p^n$$

例如，有两个服务器，在给定的时间段内任何一个服务器出故障的概率是5%，那么，其可用性概率就是$1 - 0.05^2 = 1 - 0.0025 = 99.75\%$。缓冲系统和服务器复制的一个重要区别就是缓冲并不一定保存全部对象（如文件）集合。因而缓冲在应用层次上不一定能够增强可用性——因为用户需要的文件可能没有被缓存。

网络分区（参见第11.1节）和断连操作是影响高可用性的第二个因素。移动用户在移动过程时，可能有意或无意地将计算机从无线网络中断开。例如，一个乘坐火车的用户，他的笔记本电脑可能无法上网（无线网络可能会中断，或者可能没有这种性能）。为了能够在这种环境下工作——这被称为断连工作或者断连操作，用户经常将高使用率的数据（如共享日记的目录）从他们平时的应用环境中复制到笔记本电脑内。但是在断连过程中，总是存在一个关于可用性的权衡：当用户查阅或更新日记时，这些数据可能正在被其他人阅读或修改。例如，他们可能在一个已占用的时间段安排面谈。断连工作仅在用户能够处理这种过期数据并且此后能够解决由此导致的所有冲突时才有效。

- 容错性　高可用性数据并不一定是绝对准确的数据。例如，它们可能已经过时，或者两个在网络分区不同地方的用户作出冲突的更新操作。相反地，一个容错服务在一定数量和类型的故障范围内总能确保严格正确的行为。这里的正确性关注提供给客户的数据是否最新以及客户对数据做的操作的结果。正确性有时也要考虑服务的响应时间，例如在航空控制系统中，正确数据必须在短时间内获得。

在计算机之间复制数据和功能这一用于高可用性的基本技术同样可用来实现容错。如果$f+1$个服务器中有至多f个崩溃了，那么从理论上讲至少还有一个服务器能够提供服务。如果至多f个服务器会发生拜占庭错误，那么理论上$2f+1$个服务器能够通过正确的服务器进行投票找到故障服务器（其可能提供混乱值），从而提供正确的服务。但是容错性要比这种简单的描述复杂。系统必须处理其成员之间的协调来精确地处理任何时间都可能发生的故障。

复制透明性是数据复制的公共需求。这就是说，客户在一般情况下并不需要知道存在多个物理拷贝。客户关心的是，数据组织成独立的逻辑对象，当需要执行一个操作时，他们仅

对其中的一项进行操作。进而，客户期望操作仅仅返回一个值的集合，而不管事实上操作可能针对一个以上的物理拷贝。

数据复制的另一个公共需求是一致性，一致性强度在不同应用中会有所不同，它主要关注针对一个复制对象集合的操作必须满足这些对象的正确性要求。 555

我们看到在日记的例子中，断连数据操作可能造成数据（至少是暂时的）不一致。但是当客户保持连接时，如果不同的客户获取了不一致的数据，这通常是不可接受的。换言之，对应用正确性的破坏是不可接受的。

以下我们将考虑在利用复制数据保证高可用性和容错性服务时的更多细节问题。我们还要研究一些标准的解决方案和技术。首先，14.2节至14.4节描述了基于共享数据的客户操作。14.2节给出了一个管理复制数据的通用体系结构并介绍了作为重要工具的组通信。组通信对于实现容错极为有效，它是14.3节的主题。14.4节阐述各种高可用技术，包括断连操作。14.4节还包括了对gossip体系结构、Bayou和Coda文件系统的实例研究。14.5节介绍了如何在复制数据上进行事务处理。正如第12章和第13章所解释的，事务处理是由一系列操作组成的，而不是单个的操作。

14.2　系统模型和组通信

我们系统中的数据是由对象集合组成的。一个“对象”可以是一个文件，或者是一个Java对象。每一个逻辑上的对象是由若干称作副本的物理拷贝组成的集合实现的。副本是物理对象，每一个都存储在某台计算机上，这些副本上的操作遵循某种程度的一致性。给定对象的副本并不一定要完全相同，至少不必在任何的时间点上都要求一样。一些副本可能已经接受了更新的数据而另一些副本还没有更新。

本节先提供一个用来管理副本的通用系统模型，然后描述组通信系统。在通过复制达到容错的技术中，组通信是非常有用的。

14.2.1　系统模型

我们假定一个异步系统中的进程发生故障的惟一原因是崩溃。我们的默认假设是不会发生网络分区，但是我们也考虑出现网络分区时会出现的情况。我们使用故障检测器来获得可靠和全序的组播，但是网络分区使得建立故障检测器变得更困难。

从一般性方面考虑，体系结构组件是通过其功能来描述的，但不意味着每项功能必须用不同的进程或者硬件来实现。这个系统中的副本由不同的副本管理器（RM）来管理（见图14-1）。副本管理器是包含了特定计算机上的副本，并且直接操作这些副本的组件。该模型可以用在客户－服务器的环境中，此时，一个副本管理器就是一个服务器。我们有时简单的称副本管理器为服务器。同样的，该模型也可以应用到一个应用程序，在这种情况下应用进程既是客户又是副本管理器。例如，乘火车用户的笔记本电脑包含有一个应用，它的作用相当于用户日记的副本管理器。

我们应该始终要求一个副本管理器对于它副本的操作是可恢复的。因此我们可以假定，如果操作中途失败了，副本管理器也不会留下不一致的结果。我们有时要求每个副本管理器是一个状态机[Lamport 1978，Schneider 1990]。这样的一个副本管理器对其副本数据实行原子性操作（不可分操作），其执行等价于某种严格排序的执行。此外，副本数据的状态是其初

始状态的一个确定性函数，由在副本数据上的操作次序决定。其他外部因素，如时钟读取或传感器读取，不会对其状态值产生影响。没有这个假定，在独立接受更新操作的副本管理器之间建立一致性是不可能做到的。系统只能够决定在所有副本管理器上应用什么样的操作和它们的次序——它不会再次产生非确定的影响。这个假设意味着服务不可能是多线程的。

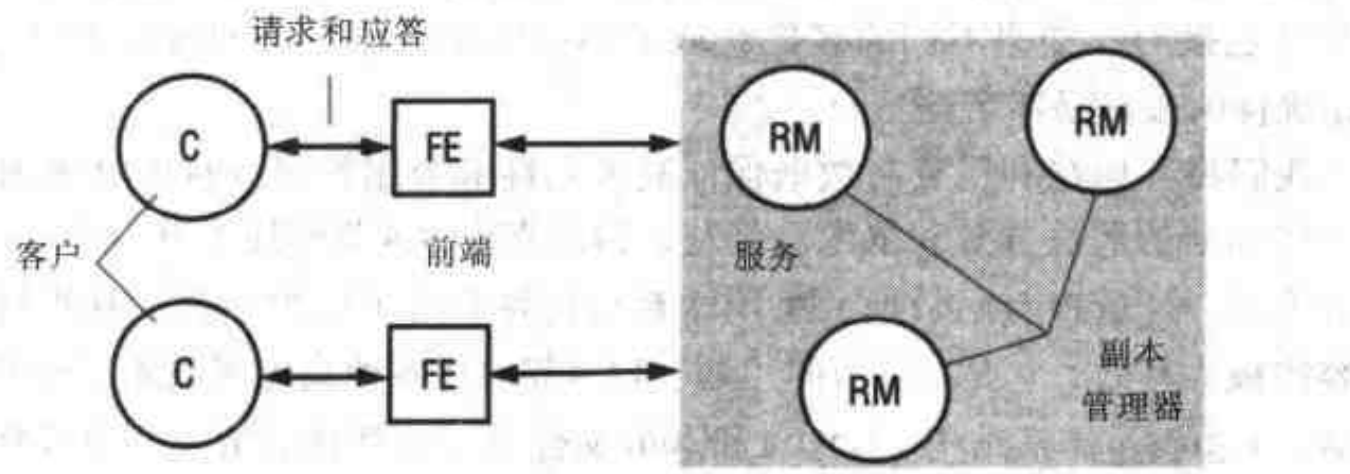


图14-1 管理复制数据的系统体系结构的基本模型

在不指明的情况下，每个副本管理器为每一个对象都维护一个副本。但一般情况下，不同对象的副本由不同的副本管理器来维护。例如，某个对象对一个网络上的客户可能非常需要，而另一个网络上的客户却需要另一个对象。相互复制这些对象到其他的网络管理器上就没什么好处。

副本管理器的集合可以是静态的或动态的。在动态系统中，新的副本管理器可能不断出现（例如，另一个秘书拷贝一份日记到他的笔记本电脑）；而在静态系统中这是不允许的。在动态系统中，副本管理器可能崩溃，那么它们被认为离开了这个系统（尽管它们可被替换）。
556
~
557
在静态系统中，副本管理器不会崩溃（崩溃意味着将永远不会执行下一步），但他们可能停止操作任意长一段时间。我们将在14.2.2节讨论故障问题。

图14-1表示了副本管理器的一般模型。副本管理器集合提供给客户某种服务。客户得到一个允许它们访问对象的服务（例如，日记或银行账户），而这个对象其实复制到了管理器上。客户每次请求一系列的操作——调用一个或多个对象。不包含更新操作的请求称作只读请求，包含更新操作的请求称作更新请求（更新请求也可能包含读操作）。

每个客户的请求先由一个被称为前端（FE）的组件处理。前端的作用是通过消息传递与多个副本管理器进行通信，而不是直接让客户进行通信。这是保证复制透明性的手段。前端可以实现在客户进程的地址空间内，也可以实现成一个独立的进程。

副本对象上的一个操作通常涉及5个阶段 [Wiesmann *et al.* 2000]。对于不同的系统，每一阶段的行为都不一样，下面的两节会做进一步的说明。例如，一个支持断连操作的服务行为与支持容错的服务行为是不同的。这些阶段分别是：

前端将请求传给一个或多个副本管理器。一种可能是前端只和某个副本管理器通信，然后这个管理器再与其他的管理器通信。另一种可能就是前端将请求组播到各个副本管理器。

- 协调　副本管理器首先进行协调以保证执行一致性。如果需要的话，在这一阶段，它们将就是否执行请求而达成一致（如果这一阶段出现故障，请求将不会被执行）。副本管理器同时决定该请求相对于其他请求的次序。11.4.3节中为组播定义的所有排序类型同样适用于请求处理，这里我们再次说明这些排序类型：

- FIFO序　如果前端发请求r，然后发请求r'，那么任何的副本管理器在处理r'之前，先处理r。
- 因果序　如果请求r要在请求r'之前发生，那么任何的副本管理器在处理r'之前，先处理r。
- 全序　如果一个正确的副本管理器在处理请求r'之前处理请求r，那么任何的副本管理器在处理r'之前，先处理r。

大多数的应用需要FIFO序。我们在下两节中讨论对因果序、全序、混合序的需求，混合序是指既有FIFO又有因果序或是既有因果序又有全序。

- 执行　副本管理器执行请求，包括尝试性的请求，请求执行的效果是可以去除的。
- 协定　副本管理器对于要提交的请求的影响达成一致。例如，在这个阶段，在一个事务系统中副本管理器可共同决定是放弃还是提交事务。 558
- 响应　一个或多个副本管理器响应前端。在某些系统中，只有一个副本管理器响应前端。在另一些系统中，前端接收一组副本管理器的应答，然后它选择或合成一个单独的应答返回给客户。例如，如果目标是保证高可用性，那么它可以返回第一个到达的应答给用户。如果目标是屏蔽拜占庭故障，那么它需要将大多数副本管理器提供的应答传给客户。

不同的系统可以选择对各个阶段进行不同的排序，也可以选择它们的内容。例如，在支持断连操作的系统中，尽可能早地将应答反馈给客户是非常重要的（比如用户笔记本电脑的应用）。用户并不希望一直等到笔记本电脑的副本管理器与办公室里的副本管理器能够协调。相反，在一个容错系统中，结果的正确性得到保证以前将不会把应答给客户。

14.2.2　组通信

11.4节讨论了组播通信，因为进程组是组播消息的目标，因此也被称作组通信。对管理复制数据，组非常重要的。在其他一些系统中，如果进程通过接收和处理相同的组播消息来合作完成共同目标，那么组也是非常重要的。当组成员独立地接收一个或多个共同消息流，特别是消息包含进程要独立做出反应的事件信息时，组同样非常有用。

11.4节将组成员定义成静态的，尽管组成员可能崩溃。然而实际系统经常需要动态的成员关系：在系统执行过程中，进程可以加入和离开组。例如，在管理复制数据的服务中，用户可以加入或删除副本管理器，或者一个副本管理器由于崩溃需要从组中删除。组通信的完整实现除了组播通信以外，还需要包含了一个组成员关系服务来管理动态组成员。

组播和组成员关系管理是非常紧密联系的。图14-2表示了一个开放的组，一个组外的进程向该组发送消息，不需要知道组的成员情况。当组播并发执行时，组通信服务必须管理组成员的变化。

组成员关系服务的作用　组成员关系服务有如下4个任务：

- 为组成员变更提供接口　组成员关系服务提供操作来创建或撤销进程组、在组中加入或删除某个进程。在大多数系统中，一个进程可以同时属于若干组。IP组播正是如此。
- 实现故障检测器　该服务有一个故障检测器（见11.1节）。该服务检测组中各成员，不仅考虑进程崩溃的情况，还考虑由于通信故障而不可达的情况。检测器将进程标记为“可疑”或“不可疑”的。服务利用故障检测器来决定组的成员关系：如果某个进程被怀疑发生了故障或是不可达时，就将其从成员列表中删去。 559
- 通告组成员关系变更　当一个进程被加入或删除时（由于故障或进程主动脱离组时），

该服务将通知组成员这个变更情况。

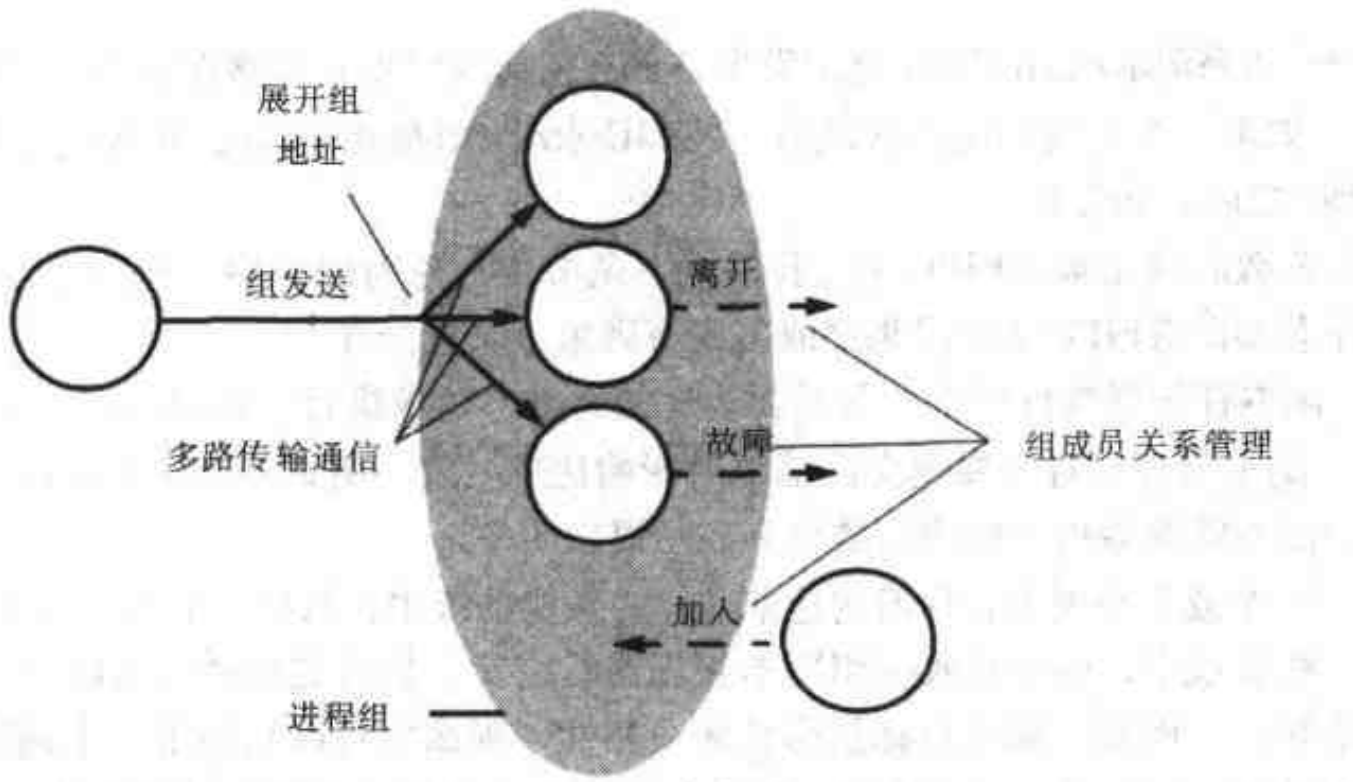


图14-2 进程组提供的服务

- *扩展组地址* 当一个进程组播消息时，它提供组标识而不是组中的进程列表。为了传送，组成员关系服务将标识扩展为当前组成员标识。通过控制地址扩展，服务能够协调好成员变化时的组播，即它能够一致地决定将给定的消息传送到哪，尽管成员在传送时会发生变化。我们将在下面讨论视图同步通信。

注意，IP组播是一种弱化的组成员关系服务，它具有组成员关系服务的某些但不是全部的性质。它允许进程动态地进入或离开组，它有地址扩展，这样在组播消息时，发送者只需要单个的IP组播地址作为目的地址。但是IP组播本身并不为组成员提供当前成员的信息，并且组播也不会随着成员的变化而调整。

如果系统要求自身能适应进程加入、离开和崩溃的情况，那么它（特别是容错系统）通常会要求有更高级的特性，例如故障检测和组成员变更通知等。一个完整的组成员关系服务维护组视图，即由进程标识符标识的当前组成员的列表。该列表是有序的，例如可按照成员加入组的顺序来排序。在进程加入组或从组中删除时，一个新的组视图就产生了。

560

非常重要的一点是，组成员关系服务可能因为某个进程处于“怀疑”状态而将其删除，尽管这个进程可能还未崩溃。通信故障可以使进程变得不可达，尽管它仍在正常执行。组成员关系服务总是删除这样的进程。删除的结果是，此后消息将不再发送给这个进程。另外，在一个封闭组中，如果进程又连接上了，任何它试图发送的消息将不会发给组成员，这个进程必须重新加入这个组（作为自己的一个“新生”，将获取有新的标识），或放弃它的操作。

如果错误地怀疑进程并进而将其从组中删除，这将降低组的有效性。除了要将故障检测器设计得尽可能准确，设计挑战还在于，如果当一个进程被错误地怀疑时，组通信也不会异常工作。

如何对待网络分区是组管理服务的一个重要考虑。诸如路由器等网络组件的断开或故障都会使一个进程组分割成若干个子组，这些子组之间不能通信。组管理服务分为两类：主分区或者可分区。在第一种情况下，组管理服务最多允许一个子组（一个较大的子组）能够从网络分区中存活下来，其他进程被告知挂起。这种安排非常适合于进程管理重要的数据以及

各子组之间的不一致的代价大于断连操作优点的情景。

另一方面，在某些情况下两个或更多的子组继续操作是可接受的，一个可分区的组服务就是这样。例如在一个应用中，用户使用音频或视频会议来讨论某些问题时，当分区发生时两个或更多的子组成员之间进行独立讨论是允许的。当分区修复或子组又连接上时，它们可以再融合它们的讨论。

视图传送　考虑某个程序员的任务是写一个应用，这个应用运行在一组进程中的每一个，它必须处理组成员的加入和删除。程序员需要知道当组成员变更时，系统用某种一致性的方法来对待每一个组成员。每当组成员变化时，如果程序员不能就如何响应变更根据局部信息做出决定，而不得不查询其他的组成员的状态才获得决定，那么这种方法是非常笨拙的。程序员工作的难易取决于系统能够确保何时将视图传送给组成员。

对于每个组g，组管理服务将一系列的视图$v_0(g)$, $v_1(g)$, $v_2(g)$,…传送给组的每个进程$p \in g$。例如，一个视图的序列可以是$v_0(g) = (p)$, $v_1(g) = (p, p')$, $v_2(g) = (p)$——首先p加入一个空组，然后p'加入这个组，然后p'离开这个组。尽管组成员变更可能会同时发生，例如当某个进程进入组时另一个进程离开组，但是，系统可以对视图强加一个次序。

如果当组成员发生变更时，一个成员将新的成员关系告知给应用（和传送组播消息类似），那么我们称这个成员传送了视图。对组播传送，传送视图和接收视图是截然不同的。组成员 561
协议将提出的视图放到一个缓存队列上，直到所有现存的组成员同意这个传送。

我们称一个事件发生在进程p的视图$v(p)$中，如果在事件发生时，p已经传送了视图$v(p)$但是还没有传送下一个试图$v'(g)$。

视图传送有如下一些基本要求：

- 顺序　如果一个进程p传送视图$v(g)$，然后传视图$v'(g)$，那么不存在这样的进程$q \neq p$：在$v(g)$之前传送$v'(g)$。
- 完整性　如果进程p传送视图$v(g)$，那么$p \in v(g)$。
- 非平凡性　如果进程q加入到一个组中，对于进程$p \neq q$来说变为可达的话，那么最终q总是在p发送的视图中。同样的，如果组被分割并形成分区并且分区仍然存在，那么最终任何一个分区所传送的视图将排除其他分区中的进程。

通过保证在不同的进程中视图变化总是以同样的次序发生，上面的第一个要求给予程序员一致性的保证。第二个要求是一个完整性检查。第三个是为了防止平凡的解决方案。例如，一个组成员关系服务不管进程的连接性，告诉每一个进程它自己在这个组中并没多大意思。非平凡性表明如果两个已经加入了同一个组的进程能进行无限期的通信，那么它们每一个将被认为是同一个组的成员。同样的，当一个分区发生了，组成员关系服务应该最终反映分区。条件并没有表明在有疑问的间歇性连接时组成员关系服务应如何处理。

视图同步的组通信　在组播消息传送方面，视图同步的组通信系统除了以上视图传递的要求外，还能达到额外的保证。视图同步的组播通信扩展了第11章讨论的可靠组播语义，因为考虑到了组视图的动态变化。为了简化讨论，我们只考虑不发生分区的情况。视图同步的组通信提供的保证如下：

- 协定　正确进程在任何给定的视图中传送同样的消息集合。换句话说，如果一个进程在视图$v(g)$中传送消息m然后传送另一个视图$v'(g)$，那么所有传送下一个视图$v'(g)$的进程，即在$v(g) \cap v'(g)$中的成员，也传送在视图$v(g)$中传送m。

- 完整性　如果一个进程p传送消息m，那么它不会再传送m，而且，$p \in group(m)$，并且m由$sender(m)$进行组播操作（对于可靠的组播也是如此）。
- 有效性（封闭组）　正确进程总是传送它们发送的消息。如果系统在给一个进程q传送消息时发生了故障，那么它将会传送一个将q删除了的视图给余下的进程。如果p是一个正确的进程，它在视图$v(g)$中传送消息m。如果某个进程$q \in v(g)$不在视图$v(g)$中传送m，那么在p传送的下一个视图$v'(g)$中将有$q \notin v'(g)$。

562

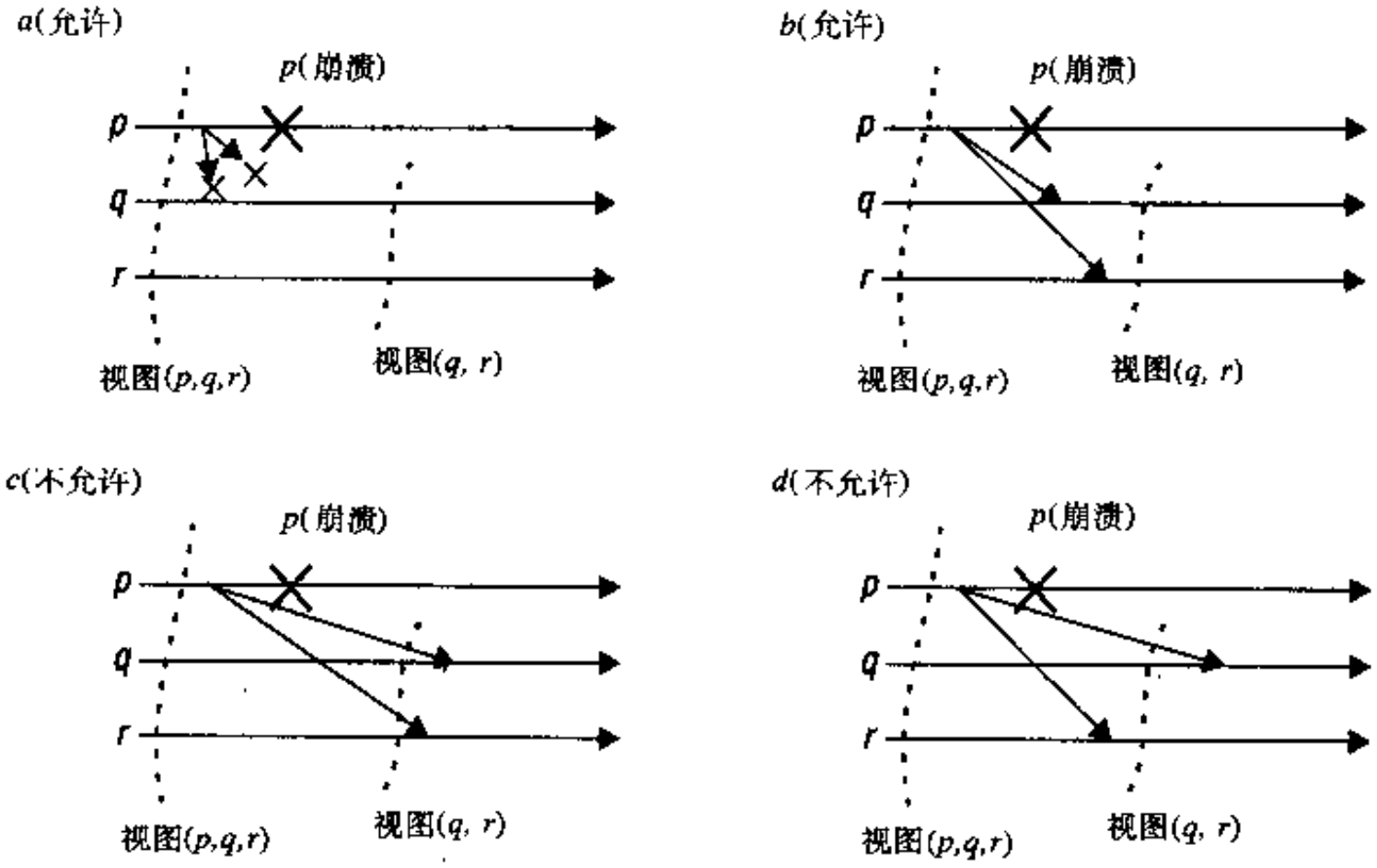


图14-3　组通信中的视图同步

考虑一个具有3个进程p，q，r的组（如图14-3）。假设p在视图(p, q, r)中发送一个消息m，p发送完消息m后就崩溃了，而q和r仍然正确运行。一种可能性是当m到达任何其他的进程之前p就崩溃了。在这种情况下，q和r每一个都发送新的视图(q, r)，但都不会传送消息m（如图14-3a）。另一种可能性是当p崩溃时，消息m至少到达了余下的两个进程中的一个。那么q和r都先传送消息m然后是视图(q, r)（图14-3b）。让q和r先传送视图(q, r)然后传送m是不允许的（图14-3c），因为那样它们将传送消息的进程已经被告知出现了故障；同样两个也不能以相反的次序传送消息和新视图（图14-3d）。

在一个视图同步系统中，新视图的传送在概念上绘制出一条横穿系统的线，每一个被传送的消息都被一致地传送到线的一边或另一边。这样，当程序员传送一个新的视图时，他能仅仅基于局部的消息传送和视图传送事件的次序，推断出其他正确进程传送的消息的集合。

用视图同步通信来获得状态转换——工作状态从一个进程组的当前成员转到组的另一个成员——可以说明视图同步通信的用处。例如，如果进程是保存有日记状态的副本管理器，然后在一个需要日记的进程加入组时，它需要获得日记的当前状态。然而当日记状态被捕获的同时，日记可能被更新了。而副本管理器不应遗漏任何在它获得的状态中没有反映的更新消息，也不应重新再使用已经反映在状态中的更新信息（除非这些信息是幂等的）。

563

为了获得这种状态转换，我们可以在如下的一个简单的方案中应用视图同步通信。当第

一个包含新进程的视图被传送时，一个现存的进程中的特殊的进程——比如最早的一个——截获了它的状态，将其一对一地发送给新的成员并且挂起它的执行。所有其他的进程也都暂停它们的执行。注意根据视图同步通信的定义，在这个状态中的更新集合要非常明确地告知其他所有的成员。当新进程收到状态后，新进程会综合它，然后组播一个“开始”消息给组，这时所有的进程再次执行。

讨论　我们已经讨论的视图同步的组通信概念是“虚拟同步”通信范型的一种形式。这个范型最早开发于ISIS系统中[Birman 1993, Birman *et al.* 1991, Birman and Joseph 1987b]。Schiper和Sandoz[1993]描述了一个获得视图同步（他们称做原子视图）通信的协议。注意组成员关系服务获得共识，它这样做并不违反Fischer等人[1985]的关于不可能性的理论结果。就像我们将在11.5.4节中描述的那样，系统可通过使用一个适当的故障检测器来巧妙地解决这个问题。

Schiper和Sandoz还提供了一个统一的视图同步通信版本。在这个版本中协议的情况包括进程崩溃的情况。对于组播通信的统一协议也是相似的，这已在11.4.2节中描述了。在视图同步通信的统一版本中，即使一个进程在它传送完消息后崩溃了，所有正确的进程被强迫在同样的视图中传送这个消息。这个更强的保证有时在容错应用中需要，因为一个已经传送了消息的进程在崩溃以前可能对外部的世界有一定的影响。出于同样的原因，Hadzilacos和Toueg[1994]考虑了在第11章中描述的可靠的和有序的组播协议的统一版本。

V系统[Cheriton and Zwaenepoel 1985]是第一个支持进程组的系统。在ISIS后，具有部分组成员关系服务的进程组开始在其他的系统中发展了，这包括Horus[van Renesse *et al.* 1996]和Totem[Moser *et al.* 1996]。

视图同步也有针对可分区的组成员关系服务的变种，包括支持网络分区处理的应用[Babaoglu *et al.* 1998]和扩充的虚拟同步[Moser *et al.* 1994]。

最后，Cristian[1991b]讨论了对于同步分布式系统的组成员关系服务。

对象组　对象组提供了针对组计算的一个面向对象方法。一个对象组是一个对象的集合（通常是同一类的实例）。这些对象处理相同的调用集合，并且每一个都返回应答。客户对象不需要知道复制。它们调用一个局部的对象，这个对象相当于组的代理，这个代理使用组通信系统给对象组的每一个成员发调用。

Electra[Maffeis 1995]是一个CORBA兼容的系统，它支持对象组。一个Electra组可以衔接到任何CORBA兼容的系统上。Electra最早建立在Horus组通信系统之上，用Horus来管理组的成员和组播调用。在“透明”模式下，局部代理返回第一个应答给客户对象。在“不透明”模式下，客户对象可以访问所有组成员返回的应答。Electra使用一个扩展的CORBA ORB接口，该接口具有创建和撤销对象组、管理它们的成员的功能。

564

Eternal[Moser *et al.* 1998] 和对象组服务[Guerraoui *et al.* 1998]也为组对象提供了CORBA兼容的支持。

14.3　容错服务

本节考察如何通过在副本管理器上复制数据和功能提供容错服务，即即使有至多f个进程出故障还能提供正确的服务。为了简单起见，我们仍然假定通信是可靠的，不发生分区。

每个副本管理器在没有崩溃时按照它管理的对象的语义规约来执行。例如，银行账户的

规约将包括如下保证：在银行账户间转账的资金不会消失，并且对于任何账户，只有存款和取款会影响余额。

直观上，如果在出现故障情况下，服务还能保持响应或客户不能区别服务是实现在副本数据上还是由一个正确的副本管理器提供的，那么基于复制的服务是正确的。我们必须非常仔细地对待这个准则。否则，在有许多副本管理器时，可能会发生异常——即便我们考虑的是一个单独的操作而不是一个事务。

考虑一个简单的复制系统，其中两个副本管理器分别在计算机A和B上，每一个都维护两个银行账户x和y的副本。客户在他们的局部的副本管理器上读取和更新账户，在那个管理器出现了故障才尝试另一个管理器。当响应完客户后，副本管理器会在后台相互传播更新。两个账户初始余额为0美元。

客户1在它局部的副本管理器B上更新x的余额为1美元，然后试图更新y的余额为2美元但是发现B出故障了。客户1因此将更新应用在A上。现在客户2在它的局部副本管理器A上读取余额，发现y有2美元然后发现x是0美元——由于B出现故障，B的x银行账户的更新没有传过来。情况显示如下，每个操作被标记上计算机名，另外操作按发生次序排列：

客户 1:	客户 2:
$setBalance_B(x, 1)$	
$setBalance_A(y, 2)$	
	$getBalance_A(y) \rightarrow 2$
565	$getBalance_A(x) \rightarrow 0$

这个行为不符合银行账户行为的规则：客户2如果读到了y的余额为2美元，而y的余额是x余额更新之后更新的话，那它应该读到x的余额为1美元。如果银行账目是由一台服务器实现的话，那么这种复制情况下的异常将不会发生。我们可以构建一个管理副本对象的系统，它不会因为我们在例子中采用了简单协议而发生异常行为。为此，我们必须首先理解复制系统的正确行为是什么样的。

线性化能力和顺序一致性 对于复制对象有不同的正确性准则。最严格正确的系统是可线性化的，该性质称为线性化能力。为了理解线性化能力，考虑一个有两个客户的复制服务实现。设客户i的某一执行中读和写操作的序列为$o_{i0}, o_{i1}, o_{i2}, \cdots$其中每一个$o_{ij}$在运行时包含操作类型、参数和返回值。我们假定每一个操作都是同步的，即客户在执行下一个操作前必须等待前一个操作的完成。

管理单副本对象的单个服务器总是串行化客户的操作。在只有客户1和客户2的执行情况下，这种执行的交织可能是$o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, o_{12}, \cdots$。我们通常参考一个虚拟的客户操作序列来定义复制对象的正确性准则。这种序列不一定在每台副本管理器上发生，但它建立了执行的正确性。

一个复制的共享对象服务被认为是可线性化的，如果对于任何执行，存在某一个全体客户操作的序列，满足以下两个准则：

- 操作的交叉执行序列符合对象的（单个）正确副本所遵循的规约；
- 操作之间的交叉执行序列和实际运行中的次序一致；

该定义符合这样的观点：对于任何客户操作，依靠共享对象的虚拟映像，有一个虚拟的规范执行——所谓规范执行是指由定义确定的交叉执行操作。每个用户看到的共享对象的视

图和那个映像一致：即当用户的操作发生在交叉执行序列中时，这些操作结果才有意义。

在所举的例子中，引起银行账户客户执行的服务不是线性化的。即使忽略了操作发生的真实时间，上述的例子也没有两个操作的任何序列满足银行账目规范的序列：为了进行审计，如果一个账目更新发生在另一个之后，那么如果已经看到第二个更新，这第一个更新也应该要保存。

注意到可线性化仅仅考虑个体操作的次序，并不打算是事务化的。如果没有并发控制的话，一个线性化操作可能破坏应用指定的一致性概念。

可线性化中的实时要求是现实世界所需的，这是因为它符合我们的观念：客户应能收到最新的数据。不过，实时性要求会引起了可线性化的一些现实问题，例如我们不能将时钟同步到要求的正确程度。一个较弱的正确性条件是顺序一致性。在不要求实时的情况下，这个条件抓住了处理请求的实质。顺序一致性保留了可线性化的第一个准则，但对第二个准则做了修改：

566

一个被复制的共享对象服务被称为是顺序一致的，如果对所有客户发出的操作，它们之间的交叉满足下面条件：

- 操作的交叉序列符合对象的（单个）正确副本所遵循的规范；
- 交叉执行中的操作次序和每个客户程序中执行的次序一致。

注意到绝对时间并没有出现在上述定义中，同样在操作上也没有要求任何全序。与次序相关的惟一概念是每个客户上的事件次序——程序的次序。操作的交叉执行会以任意方式来重新排列一个客户集合上的操作。这就像把几堆牌以某种方式混在一起，但要求保持每堆牌的原有次序。

每一个可线性化服务都是顺序一致的，这是因为实时次序反映了程序次序。但是反过来不成立。下面的例子是顺序一致性的，但不是可线性化的：

客户 1:	客户 2:
$setBalance_B(x, 1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y, 2)$	

这个执行在简单复制策略下是有可能出现的，即使计算机*A*和*B*都没出故障，但客户1在B上做的x的更新在客户2读它时没有到达的话，就会出现这个执行。该例中，由于$getBalance_A(x) \rightarrow 0$发生在$setBalance_B(x, 1)$之后；线性化的实时准则没有满足。但是下面的顺序却满足顺序一致性的两个准则：$getBalance_A(y) \rightarrow 0$, $getBalance_A(x) \rightarrow 0$, $setBalance_B(x, 1)$, $setBalance_A(y, 2)$。

Lamport考虑了共享内存寄存器的顺序一致性[1979]和线性化问题[1986]（尽管他使用的术语是“原子性”而不是“可线性化”）。Herlihy和Wing[1990]提出了任意共享对象的一般性方法。第16章将研究分布式共享内存，定义并讨论了一些弱一致性性质。

567

14.3.1 被动（主备份）复制

在用于容错的被动或主备份复制模型中（见图14-4），任何时候都有一个主副本管理器和一个或多个备份管理器。该模型的实质是，前端只和主管理器通信以获得服务。主副本管理

器执行操作并将更新操作的拷贝发送备份副本管理器。如果主管理器出故障了，那么某个备份管理器将被提升为主管理器。

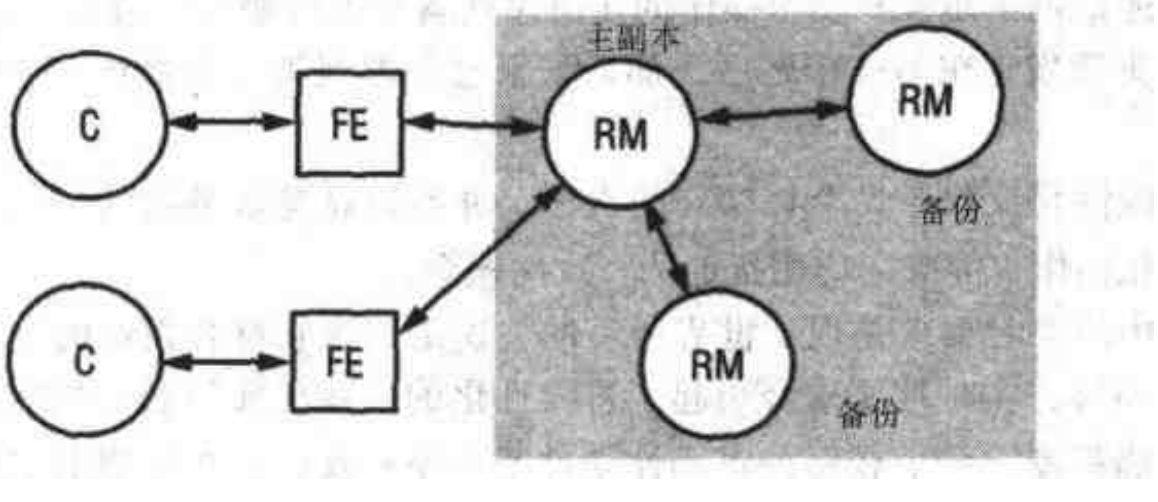


图14-4 用于容错的被动（主备份）模型

当用户需要执行一个操作时，系统执行的次序如下：

- 请求　前端将请求发送给主管理器，请求中包括了一个惟一标识。
- 协调　主管理器按收到请求的次序原子地执行每一个请求。它检查请求的惟一标识，如果请求已经执行了，那就简单地再次发送应答。
- 执行　主管理器执行请求并存储应答。
- 协定　如果请求是更新操作，那么主管理器向每个备份管理器发送更新后的状态、应答和惟一标识，备份管理器返回一个确认。
- 响应　主管理器将应答发给前端，前端再将应答发送给客户。

在主管理器正确运行的情况下，由于主管理器在共享对象上将所有操作顺序化，因此该系统显然是可线性化的。当主管理器出故障时，如果某个备份变为新的主管理器并且新的系统配置从故障点正确接管的话，那么系统仍是可线性化的：

- 主管理器被惟一的某个备份代替（如果两个客户使用两个备份，那系统将不会正确执行）；
- 当接管主管理器时，剩余的备份管理器在哪些操作已被执行上达成一致。

如果副本管理器（主管理器和备份管理器）组织为一个组，并且该组使用视图同步通信发送更新到备份，那么这两个要求都能达到。上述两个要求的第一个很容易满足。当主管理器崩溃时，通信系统最终传送一个新的视图给备份，该视图不包含原来的主管理器。替代主管理器的备份可以用任何函数选择，比如可选择视图中的第一个成员作为替代，选为替代的那个备份使用命名服务将自己登记为主管理器。

568

通过使用视图同步的次序性质，以及通过储存标识来检测重复的请求，可以满足第二个要求。视图同步的语义保证了在传送新视图以前，所有备份或者没有备份传送任何更新。新的主管理器和备份都对客户的更新是否已执行了能够达成一致。

现在来考虑前端没有收到应答的情况。前端将请求重传到接管后的主管理器。主管理器在执行操作的任何点都可能崩溃。如果它在协定阶段4之前崩溃，那么存活的副本管理器将不再处理这个请求。如果它在协定阶段中崩溃，那么它们可能执行了那个请求。但新的主管理器并不需要知道原来的管理器在崩溃时是在什么阶段，当它收到一个请求，它从第二阶段开始执行。通过视图同步，主管理器并不需要查询备份，因为它们处理了同样的消息集合。

有关被动复制的讨论　当主副本管理器以非确定性方式运行时（例如以多线程方式操作时），可以使用主备份模式。由于主副本管理器是将操作的更新状态发送给备份管理器，所以

备份只是被动地记录这些状态。

为了能够在至多f个进程崩溃时还能工作，被动的复制系统需要$f+1$个副本管理器（但该系统不能忍受拜占庭故障）。前端不需要任何容错功能，不过当主副本管理器不响应时，前端需要查找新的主副本管理器。

被动复制的缺点是需要相对大的开销。视图同步通信在每次组播时需要几个回合的通信，而且当主副本管理器发生故障时，组通信系统需要进行协商并传送新视图，这会导致更多的延时。

在该模型的一个变种中，用户可以将读请求提交到备份副本管理器，这样可减轻主副本管理器的负载。该系统不能保证线性化，但仍能提供顺序一致的服务。

被动复制系统在Harp复制文件系统[Liskov *et al.* 1991]中使用。Sun网络信息服务（NIS，以前的黄页）尽管采用了比顺序一致性要弱的保证，但通过被动复制获得了高可用性和高性能。在某些情况下，更弱的一致保证仍然可以满足需求，就像某些用于系统管理的记录存储。复制的数据在一个主服务器上被更新并从那里传播给备份服务器，使用一对一的（而不是组）通信。用户通过和主或备份服务器通信以获得信息。 569

14.3.2 主动复制

在用于容错的主动复制模型中（图14-5），副本管理器是一个状态机。每个副本管理器充当同等的角色并被组织成一个组。前端组播它们的消息到副本管理器组，并且所有的副本管理器按独立但相同的方式来处理请求。任何一个副本管理器的崩溃都不会影响服务的性能，剩下的副本管理器能继续正常地响应。我们将看到主动复制能容忍拜占庭故障，因为前端可以收集并比较收到的应答。

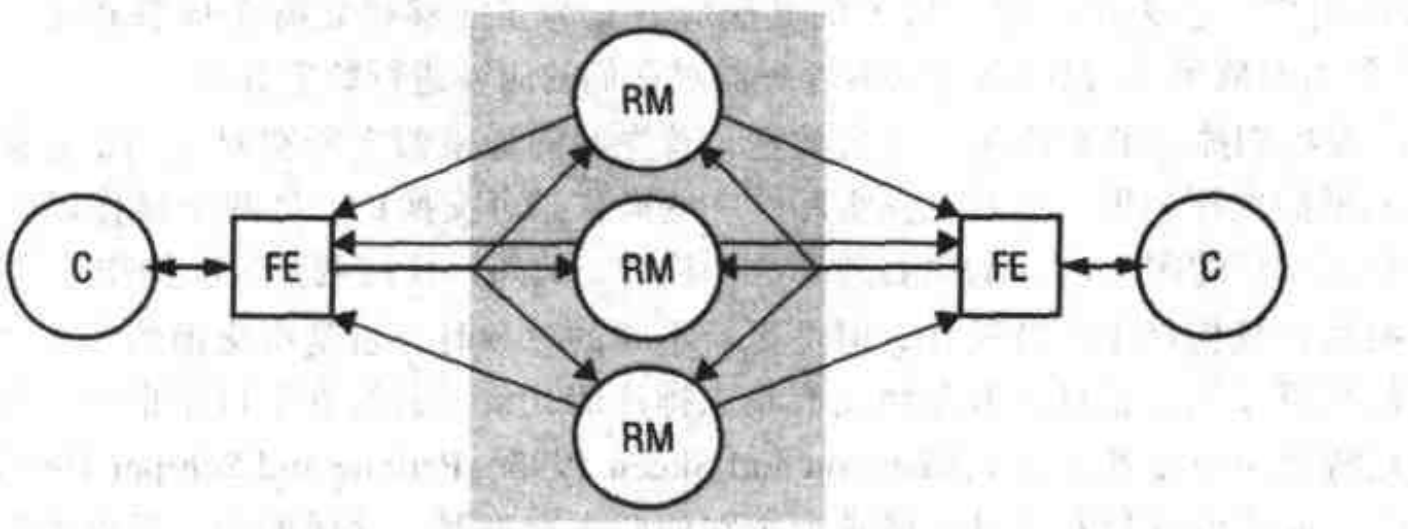


图14-5　主动复制

对主动复制，当客户请求一个操作时，系统按如下步骤操作：

- 请求　前端给请求加上一个惟一标识并将其组播到副本管理器组，这里使用一个全序的、可靠的组播原语。假设在最坏的情况下，前端会由于崩溃而出现故障。前端在收到应答之前不会发送新的请求。
- 协调　组通信系统以同样的次序（全序）将请求传送到每个正确的副本管理器。
- 执行　每个副本管理器执行请求。由于它们是状态机，并且请求到来的次序相同，因此正确的副本管理器以相同的方式处理请求，请求的应答包括客户的惟一标识。
- 协定　由于组播的传递语义，实际上不需要该阶段。

570

- 响应　每个副本管理器将它的应答送往前端，前端收到应答的数量取决于故障模型的假设和组播算法。例如，如果目标是容忍崩溃故障并且组播满足一致的协定和排序性质，那么前端可将第一个应答返回给客户，并丢弃其他应答（通过使用惟一标识，它能将这些应答从其他的应答中区分出来）。

这个系统具有顺序一致性。所有正确的副本管理器处理同样次序的请求。组播的可靠性保证了每一个正确的副本管理器处理同样的集合，全序保证了以同样的顺序处理它们。因为它们是状态机，在每一个请求后，它们都会到达同一个状态。每个前端的请求被组织按FIFO排序（因为前端在组织下一个请求前在等待应答），这和程序的顺序一样，保证了顺序一致性。

如果客户在等待它们请求的应答时并不和其他客户通信，那么它们的请求按发生在先次序处理。如果客户是多线程的，并且在等待应答时可以和其他的客户通信，那么为了确保请求以发生在先次序处理，我们必须将组播替换为既是因果序又是全序的传播方法。

出于副本管理器处理请求的全序并不一定和客户产生这些请求的实时次序相同，主动复制系统并不具有线性化能力。Schneider[1990]阐述了有大致同步时钟的同步系统中，副本管理器处理请求的全序能够根据前端为请求提供的物理时间戳顺序来实现。因为时间戳是不精确的，这并不能保证线性化，但能够保证大致上一致。

有关主动复制的讨论　我们假设存在保证全序和可靠组播的解决方法，就像第11章指出的，解决了可靠性和全序的组播等价于解决了共识。解决共识要求系统是同步的，或者使用了一个具有像故障检测器那样的技术，来绕过Fischer等人[1985]获得的理论上的不可能性。

对共识的解决方法，像Canetti和Rabin的方法[1993]，可以在有拜占庭故障下工作。如果某个特定的解决方案能够提供全序和可靠的组播，主动复制系统就能够屏蔽f个拜占庭故障，只要服务包含至少$2f+1$个副本管理器。每个前端一直等待它收集到$f+1$个相同的应答才将应答返回给用户。它丢弃对同一请求的其他应答。为了能够确定哪个应答和哪个请求相联系（假定有拜占庭故障），我们要求副本管理器对它们的应答进行数字签名。

可以对我们描述的系统进行适当放宽。首先我们原来假定所有对于共享复制对象的更新必须以同样的次序发生。然而，在实际中一些操作是可交换的：即两个操作以$o_1;o_2$次序的执行效果和以相反的次序$o_2;o_1$的执行效果是一样的。例如，任何两个只读操作（从不同的用户）是可交换的；任何两个非读操作，但是更新不同的对象时，也是可交换的。主动复制系统可

571 能需要使用可交换的信息来避免将所有请求排序的代价。我们在第11章指出一些系统已经采用了应用特定的组播排序语义[Cheriton and Skeen 1993，Pedone and Schiper 1999]。

最后，前端可以只发送只读请求到个别的副本管理器。这样的话，系统丧失了由组播请求而具有的容错，但是服务仍然是顺序一致的。此外，在这种情况下前端可非常容易地屏蔽副本管理器的故障，仅仅需要将这个只读请求发送到另一个副本管理器。

14.4 高可用服务

本章的余下部分讨论如何利用复制技术来获得服务的高可用性。我们现在的重点是使客户在合理的响应时间内访问服务——即使某些结果没有遵守顺序一致性。例如，本章开头所说的火车上的用户如果在离线时能继续工作，那么他们会愿意以后处理暂时不一致的数据（比如日记）。

在14.3节中，我们看到容错系统用一种“及时”的方式将更新传播到副本管理器：只要可

能，所有的副本管理器都收到更新，并在传送控制返回客户以前达成一致。这种方式并不适合高可用操作。另一种方式是，系统通过使用与客户连接的最小副本管理器集合，提供一个可接受的服务。当副本管理器协调它们的动作时应该尽量减少客户的等待时间。较弱程度的一致性通常要求较少的协定，使得共享数据的可用性提高。

下面研究支持高可用服务的一些系统：gossip、Bayou和Coda。

14.4.1 gossip系统

Ladin等[1992]开发了称为gossip的体系结构，用它作为框架实现了高可用性服务，具体途径是复制数据到需要这些数据的客户组的邻近点。它的名字就反映了这样的一个事实：副本管理器周期地通过gossip消息来传送客户的更新（见图14-6）。这种体系结构是基于Fischer和Michael[1982]，Wuu和Bernstein[1984]早期在数据库上的工作，可以用来创造一个高可用性的电子公告板或日记服务。

gossip服务提供两种基本操作：查询是只读操作，更新用来变更状态但却不读取状态（第二种操作比我们已经使用的更新具有更严格的定义）。一个关键的特征是前端发送查询和更新给任何它们选择的副本管理器——任何可利用和能提供合理响应时间的副本管理器。尽管副本管理器可能暂时不能和另一个通信，系统仍然做出以下两个保证：

- *随着时间推移，每个用户最终获得一致服务*　为了回答某个查询，副本管理器提供给一个客户的数据只要能反映迄今为止客户已经观测到的更新即可。用户可以在不同的时间 572
  和不同的副本管理器通信。
- *副本之间的松弛一致性*　所有的副本管理器最终将收到所有的更新。它们根据排序保证来应用这些更新，由排序规则使副本之间的一致性满足应用的要求。值得注意的是，尽管gossip体系结构可以用来获得顺序一致性，但它主要是用来保证较弱的一致性。尽管副本包含同样的更新集，两个用户仍可观察到不同的副本；用户也可能观察到过时的数据。

为了支持松弛一致性，gossip体系结构支持更新的因果序，因果序的定义见14.2.1节。它同样支持更强的排序保证：强制序（全序和因果序）和即时序。即时序的更新是在所有副本管理器上按一致的次序执行任何更新，不管这些更新的次序是因果的、强制的还是即时的。如果一个强制序的更新和一个因果序的更新之间不存在发生在先关系时，它们在不同副本管理器上执行次序可能不同，因此除了提供强制序外，还需要提供即时序。

各种次序的选择权被留给了应用的设计者，它反映了在一致性和操作代价之间的一种取舍，因果更新的代价大大低于其他排序的更新，只要可能时一般都使用它。注意由任一个副本管理器都能满足的查询，此类查询永远是和其他操作以因果序执行。

考虑一个电子公告板的应用，其中一个客户程序（它并入了一个前端）在用户机上执行，并和一个本地的副本管理器通信。客户程序将用户的投稿发送给本地的副本管理器，这个副本管理器将gossip消息的新投稿发送给其他的副本管理器。电子公告板的读者看到的是略微过时的投稿列表，但是如果延时是以分和小时计而不是天的话，一般不碍事。因果序可被用来投稿中。这意味着一般投稿在不同的副本管理器将以不同的次序出现。但是，一个标题为“Re：桔子”的投稿总是会比一个“桔子”的消息晚发送。强制序能够用来给电子公告板加入一个新的读者，这样用户加入记录是无二义的。即时序可以用来从电子公告板中的订阅列表

中删除一个用户，这样一旦删除操作返回，那个用户就不能从一个迟缓的副本管理器获得消息了。

一个gossip服务的前端通过使用应用特定的API来处理用户操作，并将其转为gossip操作。通常，用户操作可以是查询复制的状态、更新复制的状态或两者都有。由于在gossip中，更新操作只是修改状态，所以前端会把一个读取和修改都有的操作变为分离的查询操作和更新操作。

在我们的基本复制模型中，gossip服务处理查询和更新操作的大致流程如下：

- 请求　前端通常只发送请求到一个副本管理器。然而当它使用的副本管理器出故障或不可达时，前端将和另一个副本管理器通信。当正常的那个副本管理器负担过重时，前端也将尝试其他的副本管理器。因此前端和客户可能阻塞在一个查询操作上。与此相反，在默认情况下，更新操作一旦被传送给了前端，就可立即返回给用户；前端再给后台传送这个操作。为了提高可靠性，用户可以被阻塞直到更新已经传给了$f+1$个副本管理器，才继续执行，这样就算f个副本管理器出了故障，操作也将传送到任何位置。
- 对更新操作的响应　如果请求是一个更新，那么副本管理器只要一收到更新，它就立即回复。
- 协定　收到请求的副本管理器并不处理操作，直到它能根据所要求的次序约束处理请求，包括接收其他的副本管理器以gossip消息形式发送的更新。各副本管理器之间不存在其他方式的协调。
- 执行　副本管理器执行请求。
- 对查询操作的响应　如果请求是一个查询操作，副本管理器将在此给出应答。
- 协定　通过交换gossip消息，副本管理器包含了大量最近收到的更新。它们相互之间以惰性方式更新系统，这是因为gossip消息的交换是偶尔的。在收集到若干更新之后，或者当某个副本管理器发现它丢失了一个发送到其他副本管理器的更新而该管理器在处理新请求时又需要该更新时，系统才会交换gossip消息。

下面将更详细地描述gossip系统。我们先考虑前端与副本管理器为了维持更新排序保证而需要维护的时间戳和数据结构，然后解释副本管理器如何处理查询和更新。维持因果序更新的时间戳向量处理和11.4.3节的因果组播算法相似。

前端的版本时间戳　为了控制操作处理的次序，每个前端维持了一个时间戳向量，用来反映前端访问的（因而客户访问的）最新数据值。在该时间戳中，每个副本管理器有一个对应的记录（即是图14-6中的*prev*）。前端将其放入每一个请求消息中，与更新或查询操作的描述放在一起，发送给副本管理器。当副本管理器返回一个值作为查询操作的结果时，副本管理器提供一个新的时间戳向量（图中的*new*），因为处理最后一个操作后副本可能已经更新了。类似地，更新操作也返回一个惟一的时间戳向量（图中的*update*）。每一个返回的时间戳和前端先前的时间戳合并，用于记录已经被用户观察到的复制数据的版本（参见10.4节的时间戳向量合并定义）。

客户通过访问相同的gossip服务和相互直接通信来交换数据。由于客户到客户的通信也能导致服务操作之间的因果关系，因此交换数据同样也要通过前端。这样，前端可以顺便带回它们的时间戳向量给其他的客户。接收者将它们和自己的时间戳向量合并，这样可正确地保证因果次序。这种状态显示在了图14-7中。

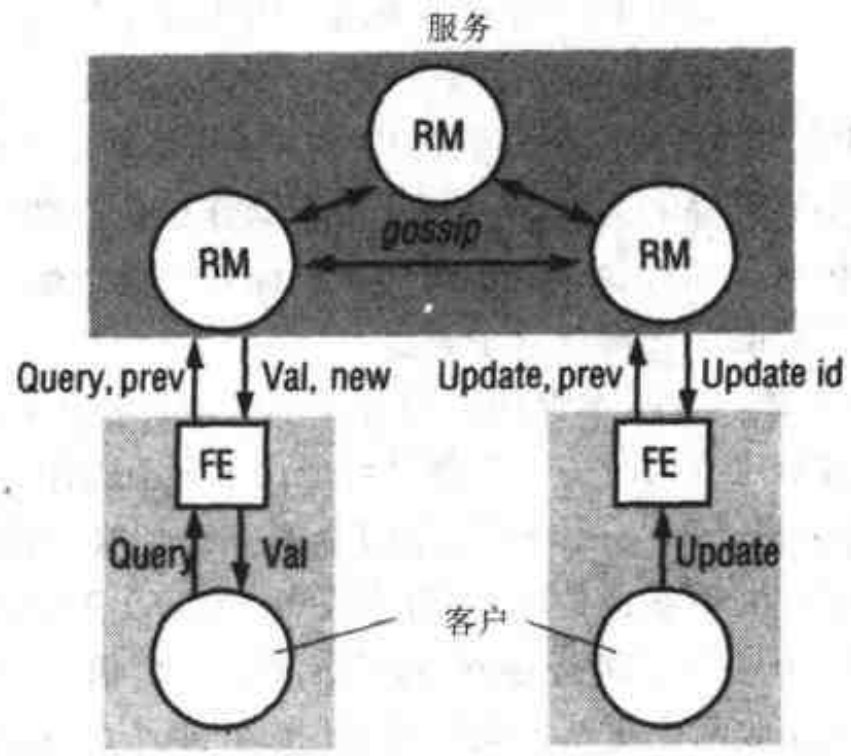


图14-6　gossip服务中的查询和更新操作

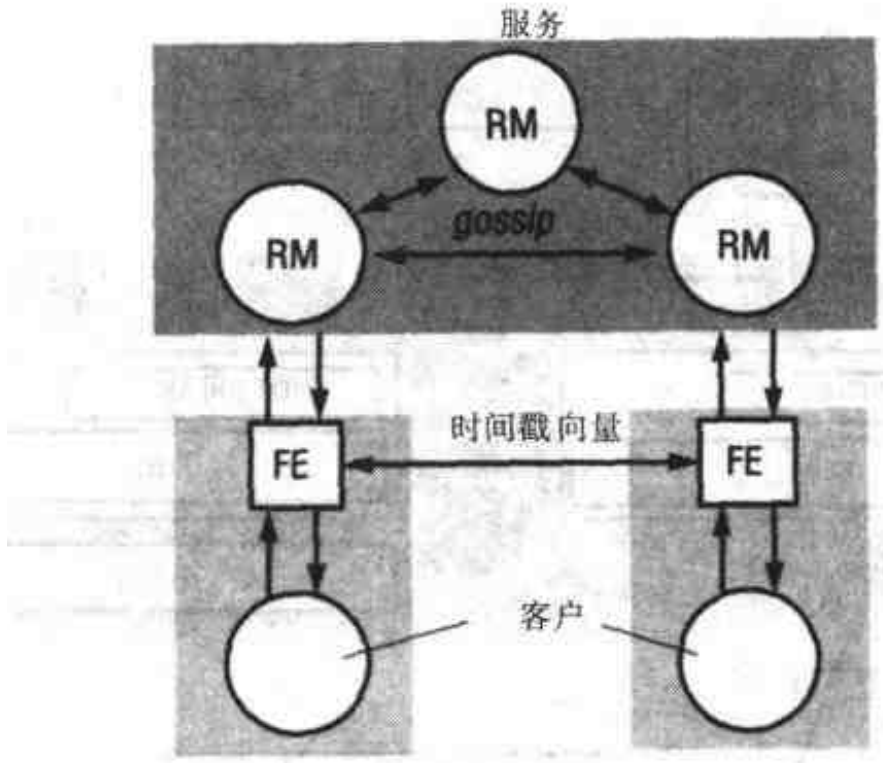


图14-7　当客户直接通信时前端传播它们的时间戳

副本管理器状态　在不考虑应用时，一个副本管理器包含的主要状态信息如下（图14-8）：

- 值　这是由副本管理器维持的应用状态的值。每个副本管理器是一个状态机，它起始于一个特定的初始值，此后，它的状态完全由更新操作来决定。
- 值的时间戳　这是代表更新的时间戳向量（更新反映在值中）。在该时间戳中，每个副本管理器有一个对应的记录，当在值上执行更新操作时，它就被更新。
- 更新日志　所有的更新操作只要它们被收到了，就将记录在这个日志中。一个副本管理器在日志中记录更新有两个理由。第一个理由是因为操作不稳定，副本管理器不能进行更新操作。一个稳定的更新操作是可以在它的排序保证（因果、强制和即时）下一致地执行。不稳定的更新必须阻止。第二个理由是，即使更新是稳定的并且已经在值上执行，副本管理器并没有收到已被其他所有副本管理器收到的确认，与此同时，它以gossip消息形式传播更新。
- 复制时间戳　这个时间戳向量代表了那些已经被副本管理器接收到的更新——即在管理

器日志中的。一般情况下，它和值时间戳不同，因为并不是所有在日志中的更新都是稳定的。

- *已执行操作表* 同样的更新可以从前端，也可以从其他的副本管理器通过gossip消息发送到一个给定的副本管理器。为了防止一个更新操作被执行两次，系统将维护一个“已执行操作”表，它包含了已经执行的更新的惟一标识，这个惟一标识由前端提供。副本管理器将更新加入日志前，先检查这个表。
- *时间戳表* 这个表为每一个副本管理器包含了一个填充在gossip消息中的时间戳向量。副本管理器使用此表来建立何时一个更新已经应用于所有的副本管理器。

副本管理器被编号为0，1，2，…，并且由第i个副本管理器掌握的时间戳向量中的的i个元素对应着通过i从前端收到的更新的数量；第j个成分（$i \neq j$）等于通过j收到的并传播给i的更新的数量。例如，在有3个副本管理器的gossip系统中，一个值时间戳(2，4，5)代表着这样的事实：从管理器0的前端接收两个更新，从管理器1接收到4个，从管理器2接收到5个。下面将会更详细地描述如何使用时间戳来保证次序。

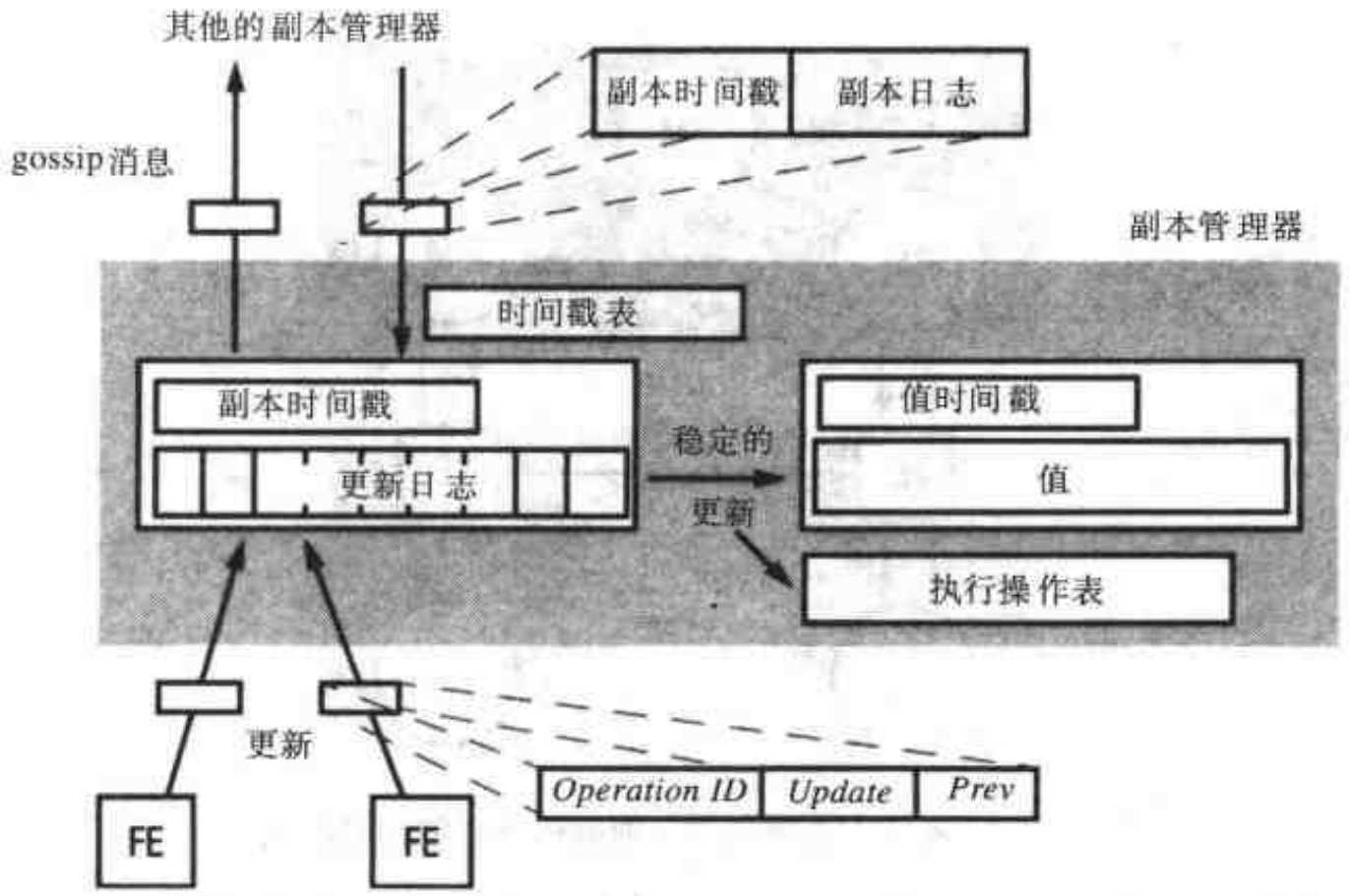


图14-8 gossip的副本管理器及其主要状态组件

查询操作 最简单的操作是查询操作。一个查询请求q包含了一个操作描述和一个由前端发送的时间戳$q.prev$，后者反映了已读到或作为更新已提交的值的最新版本。因此副本管理器的任务是返回一个最近的值。如果$valueTS$是副本的值时间戳，且下面条件满足，那么q能够在副本上执行：

$$q.prev \leqslant valueTS$$

副本管理器将q放在将执行的操作表中（即一个保持队列）直到这个条件满足。它能等待丢失的更新（丢失的更新最终将通过gossip消息到达）；也能从相关的副本管理器获取更新。例如，如果$valueTS$是(2, 5, 5)并且$q.prev$是(2, 4, 6)，可以看出来只有一个更新丢失了——从副本管理器2来的更新丢失了一个（提交q的前端必须与另一个副本管理器联系，这个管理器先前看见了这个更新而原来的管理器却没有看见）。

一旦执行了查询，副本管理器返回*valueTS*给前端，作为在图14-6中显示的时间戳*new*。前端将其和其他的时间戳合并：*frontEndTS* := *merge*(*frontEndTS*, *new*)。在所举的例子中，查询以前前端没有看到的副本管理器1上的更新（*q.prev*是4而副本管理器是5）将反映在*frontEndTS*的更新中（也可以反映在返回的值中，这取决于查询）。

按因果次序处理更新　前端提交一个更新请求给一个或更多的副本管理器。每一个更新请求*u*包含了一个更新的规约（它的类型和参数）*u.op*，前端的时间戳*u.prev*和一个前端产生的惟一的标识*u.id*。如果前端发送同样的请求*u*给若干副本管理器，每次在*u*中使用相同的标识——这样*u*就不会被处理成几个不同的请求而是相同的请求了。

576 ↓ 577

当副本管理器收到前端的更新请求时，它通过在已存在的操作表和它的日志中的记录查找这个操作的标识以确定这个请求是否已被处理。如果查找到了，它将丢弃这个请求；否则它将在复制时间戳的第*i*个元素加1，以记录它从前端直接收到的更新记数。然后副本管理器分配给更新请求一个惟一的时间戳向量（下面将给出这个向量的来源），并且一个更新记录被放置到副本管理器的日志中。如果*ts*是副本管理器分配给更新的惟一时间戳，那么更新记录按如下元组被构建并存在日志中：

$$logRecord := \langle i, ts, u.op, u.prev, u.id \rangle$$

副本管理器*i*将*u.prev*的第*i*个元素替换为它的复制时间戳的第*i*个元素（这个元素刚刚加一），完成从*u.prev*中生成*ts*时间戳，这样能使*ts*是惟一的，结果保证了所有的系统成分将正确地记录而不管它们是否观察到了更新。时间戳*ts*中剩下的元素从*u.prev*中获取，因为正是这些从前端送来的值被用来决定何时更新是稳定的。副本管理器立刻将*ts*返回给前端，前端将其与它的时间戳合并。注意前端可以提交它的更新给许多副本管理器并且收到许多不同的时间戳，所有这些时间戳都被合并入它的时间戳。

更新请求*u*的稳定性条件类似于对子查询请求：

$$u.prev \leq valueTS$$

这个条件说明了这个更新依靠的所有的更新——即所有由发起更新的前端观察到的更新——已经执行了。如果在更新提交时这个条件不满足，它将在gossip消息到达时重新检查。对于一个更新记录*r*，稳定条件已经达到，副本管理器将执行更新并更新值时间戳和已执行操作表：

value := *apply*(*value*, *r.u.op*)

valueTS := *merge*(*valueTS*, *r.ts*)

executed := *executed* ∪ {*r.u.id*}

这三个语句中，第一个代表更新值，第二个语句将更新的时间戳和那个值合并，第三个是将更新操作符加入已执行操作的标识符集合中——这用来检查重复的操作请求。

强制的和立即的更新操作　强制的更新和立即的更新需要特殊处理。更新的强制序和因果序是完全一样的。保证更新的强制次序的基本方法是将相联系的时间戳后加入一个惟一的顺序号，并以这个顺序号的次序来处理它们。像第11章所解释的，产生顺序号的一般方法是使用单一的顺序器进程。但是在一个高可用性环境中，依赖某个进程的可靠性是不够的。解决方法是在任何时候都指派一个主副本管理器作为顺序器，但当主副本管理器出故障时，另一个副本管理器能替代成为顺序器。所需要的是对于大多数的副本管理器（包括主管理器）在其操作被执行前，记录下哪个更新是下一个操作。那么，只要大多数副本管理器幸免于故

578

障，从存活的副本管理器中选出的新的主管理器可以实现这个排序决定。

gossip消息 副本管理器可以发送包含一个或多个更新信息的gossip消息，以便其他的副本管理器更新它们的状态。副本管理器使用它的时间戳表里的记录来估计其他的副本管理器更新是否收到（由于那个副本管理器可能收到了很多的更新，所以这只是个估计）。

一个gossip消息*m*包含两项：日志*m.log*和副本时间戳*m.ts*（见图14-8）。收到gossip消息的副本管理器有下面3个主要任务：

- 将到达的日志和它自己的日志合并（*m*可能包含接收者先前没看到的更新）。
- 执行任何以前没有执行并已经稳定了的更新（在gossip消息日志中的稳定的更新可能将许多未执行的更新变得稳定）。
- 当它知道更新已执行并且已经没有被重复的危险时，删除日志和已执行操作表中的记录。从日志和已更新表中删除冗余条目非常重要，不然的话，它们将无限制地增长。

将包含在gossip消息中的日志和接收者的日志进行合并是非常直接了当的。设*replicasTS*表示接收者的副本时间戳。在*m.log*中的记录*r*被加到接收者的日志中除非$r.ts \leq replicaTS$——此时，它已存在于日志中或已经被执行且被丢弃了。

副本管理器合并到来的消息中的时间戳和它自己的复制时间戳*replicaTS*，以便符合日志的增加：

$$replicaTS := merge(replicaTS, m.ts)$$

当新的更新记录被并入日志，副本管理器将确定所有已稳定的更新集合*S*。这些更新可以执行了，但必须仔细考虑它们执行的次序来维持发生在先关系。参照时间戳向量间的偏序“$\leq$”，
579 副本管理器对集合中的更新进行排序，然后它用这种次序来执行更新，即对于每一个$r \in S$被执行当且仅当没有$s \in S$且$s.prev < r.prev$。

副本管理器然后在日志中查找可丢弃的条目。如果gossip消息由副本管理器*j*发送并且如果*tableTS*是这个副本管理器的副本时间戳表，那么副本管理器设置：

$$tableTS[j] := m.ts$$

对于任何一个副本管理器都已收到的更新*r*，该副本管理器现在能够丢弃它。如果*c*是创建这个记录的副本管理器，那么我们要求所有的副本管理器*i*：

$$tableTS[i][c] \geq r.ts[c]$$

gossip体系结构同样定义了副本管理器如何删除已执行操作表中的条目。值得指出的是，操作不能过早删除，否则一个延迟过长的操作将被错误地执行两次。Ladin等人[1992]提供了该方案的细节。实质上，对于它们更新的应答，前端发出了确认，所以副本管理器知道前端何时会停止发送那个更新。它们假定了最大的更新传播延时。

更新传播 gossip体系结构并不指定何时副本管理器相互交换gossip消息，也不指定某个副本管理器如何选择其他的副本管理器来发送gossip消息。如果所有的副本管理器要在一个可接收的时间内收到所有的更新，必须要有一个健壮的更新传播策略。

所有副本管理器收到某个给定更新所花费的时间取决于3个因素：

- 网络分区的频率和持续期间
- 副本管理器发送gossip消息的频率
- 选择一个副本管理器并发送gossip的策略

第一个因素不在系统控制中，尽管用户可以在某些程度上决定他们离线工作的频率。

合适的gossip交换频率由应用决定。考虑一个由许多站点共享的电子公告板系统，每个条目看来并不需要立刻分派到所有的站点。但是如果gossip仅仅在一段很长的时间才交换，比如一天，那么会如何呢？如果只使用因果更新，非常可能的情况是，每一个站上的客户在同一个电子公告板上有它们自己的一致的讨论，而不考虑其他站点上的讨论。然后比如在深夜，所有的讨论将被合并。但是当他们要考虑其他人的讨论时，同一话题的讨论很容易不一致。在这个例子中，gossip交换的周期按小时或分钟计将更合适。

人们还提出一些由合作者选择的策略。Golding和Long[1993]在他们的“基于时间戳的反熵协议”中使用了一个类似gossip的更新传播机制，考虑了随机、确定和拓扑策略。

随机策略以随机的方式选择一个合作者，但是使用了加权概率来选择一些更合适的合作者——例如，邻近的合作者优于远的合作者。Golding和Long[1993]发现这种策略在模拟环境中 580
工作得非常好。确定性策略使用副本管理器的状态的一个简单函数来选择合作者。例如，一个副本管理器可以检查它的时间戳表，选择看上去在它收到的更新中最后的那个副本管理器。

拓扑策略将副本管理器安排为一个固定图。一种可能性是网格（mesh）：副本管理器将gossip消息发送到它连接到的4个副本管理器。另一种是将副本管理器组织为一个环，每个管理器只将gossip传给它的邻结点（比如，以顺时针方向传递），这样任何一个副本管理器的更新将遍历整个环。还有一些别的拓扑结构，如树。

像这些不同的合作者选择策略必须权衡通信量和高传播延时，以及单一的故障对其他副本管理器影响的可能性。实际中的选择和这些因素的重要性相关联。例如，环结构将产生较小的通信量，但可能造成高延时，这是因为gossip消息通常要遍历若干副本管理器。而且，如果某个副本管理器出现故障，那么整个环都不能正常工作，需要重新配置。比较而言，随机选择策略不易于出故障，但它可能产生变化的更新传播次数。

有关gossip体系结构的讨论　gossip体系结构的目标是为了保证服务的高可用性。在这种情况下，即使用户落到一个网络分区中，只要至少有一个副本管理器在这个分区中能工作，该用户可继续获得服务。但是这种可用性的代价是必须遵守松弛一致性。对像银行账户这样的对象，顺序一致性是必须的，gossip系统不会比14.3节中的容错系统工作得更好，gossip系统仅在一个主分区中提供服务。

更新传播的情性方法使一个基于gossip的系统不适应接近实时的更新复制，例如用户参加一个“实时”会议并更新一个共享文档，这种情况更合适用一个基于组播的系统。

gossip系统的规模是另一个问题。随着副本管理器数量的增长，需要传送的gossip消息的数量和使用的时间戳的大小也在增长。在一个客户进行查询时，通常需要两个消息（前端和副本管理器之间）。如果一个客户进行一个因果序的更新操作，并且R个副本管理器中的每一个都在gossip消息中收集G个更新，那么交换的消息数量为$2+(R-1)/G$。式中第一项是代表前端和副本管理器之间的通信次数，第二项是发送到其他的副本管理器的gossip消息的更新消息。提升G有助于减少消息数量，但它会使传递延时变长，因为副本管理器在传播消息前等待更多的更新到达。

为了增强基于gossip的服务的可伸缩性，一个方法是设置大多副本管理器是只读的。换言之，这些副本管理器只通过gossip消息进行更新但并不直接从前端接收更新。当*更新/查询*比例很小时，这是非常有用的。只读副本管理器可以靠近客户组，更新可由相对少的中央副本 581
管理器提供服务。因为只读副本管理器没有gossip消息传播，并且时间戳向量只需要包含那些

更新副本的条目，所以gossip流量会降低。

14.4.2 Bayou系统和操作变换方法

Bayou系统[Terry *et al.* 1995, Petersen *et al.* 1997]通过数据复制获得高可用性，类似于gossip体系结构和基于时间戳的反熵协议。Bayou系统提供的一致性保证弱于顺序一致性。在这些系统中，Bayou的副本管理器通过成对的交换更新来处理变化的网络连接；设计者也将这种交换方式称为反熵协议。但Bayou采用了一个显著不同的方法，它能够进行领域特定的冲突检测和冲突解决。

考虑一个在离线工作时需要更新日记的用户。如果需要严格一致性，在gossip体系结构中，更新将必须用强制（全序）操作执行。但那样的话，只有在主分区中的用户可以更新日记。用户对日记的访问将受限——不考虑实际上他们是否需要做会破坏日记完整性的更新。

相反，在Bayou中，火车上的和办公室中的用户都可以进行他们想要的任何更新。所有更新将被记录，并且记录它们到达了哪些副本管理器。当到达任何两个副本管理器的更新在一个反熵期间合并时，副本管理器检测并解决冲突，这时可以使用解决操作间冲突的领域规则。例如，如果一个行政主管和他的秘书都在同一个时间段加入了预约，那么Bayou需要在行政主管重新连接上他的笔记本电脑后检测到这个冲突。此外，它利用领域特定的策略解决这个冲突。在这种情况下，它能够批准行政主管的预约而取消秘书的预约。一个或多个相冲突的操作集合被取消或改变以解决冲突，这被称为操作变换。

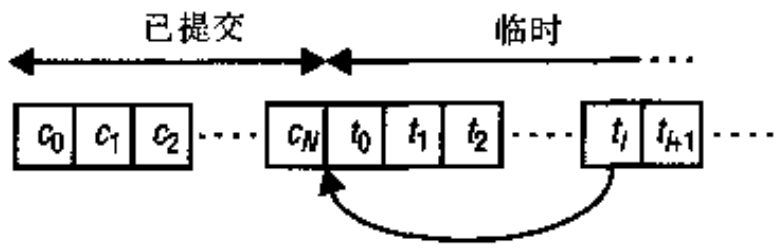
Bayou复制的状态以数据库的形式保存，它支持查询和更新（可以在数据库中插入、修改和删除条目）。尽管我们不将注意力集中在这一方面，但Bayou更新是事务的一种特殊情况。它由单个操作组成，是调用了一个“存储过程”，它影响着每个副本管理器的一些对象，但它以ACID作为保证。在执行过程中，Bayou可以取消和重做对数据库的更新。

Bayou保证最终每个副本管理器将收到相同的更新集合，副本管理器最终将以同一种方式执行这些更新，这种方式使副本管理器的数据库是相同的。实际上，可能有一个连续的更新流，数据库也永远不会相同。但如果一旦停止更新，数据库将变得相同。

提交更新和临时更新 当更新首次应用于数据库时，它们被标记为临时的。Bayou将临时的更新最终以规范次序放置并标记为提交的。在更新为临时的情况下，系统可取消和重复更新，因为系统会产生一个一致的状态。一旦提交，它们将按规定的顺序保留其效果。实际中，可以通过设计某些副本管理器为主副本管理器来获得提交的次序。通常，提交的次序是这样决定的：它收到临时的更新并且传播排序信息给其他的副本管理器。例如，对于主副本管理器，用户可以选择一个通常可用的快速机器，同样，如果用户更新占有优先权的话，主副本管理器可以是行政主管的笔记本电脑上的副本管理器。

在任何一个时间，一个数据库副本的状态来自一个（可能空的）临时的更新序列，后跟着一个（可能空的）提交的更新序列。如果来了第二个更新，或如果临时更新中的一个已经被执行变为了下一个提交的更新，那么必须对更新进行重排序。在图14-9中，t_i已经变为提交的。所有c_N后的更新都必须取消；然后，t_i是插在c_N后并且t_0到t_{i-1}和t_{i+1}等等在t_i后被重新执行。

依赖检查和合并过程 一个更新可能和已经执行的其他操作相冲突。由于这种可能性，除了操作说明（操作类型和参数），每一个Bayou更新包含一个依赖检查和一个合并过程。所有这些更新的成分都是领域特定的。



临时更新t_i成为下一个提交更新，并被插入到最新提交更新c_N之后

图14-9　Bayou中的提交更新和临时更新

一个副本管理器在执行操作前调用依赖检查过程。它用来检查一个更新执行时是否会产生冲突，为检查冲突，它可以检查数据库的任何部分。例如，考虑在日记中登记一个预约的情况。最简单的，依赖检查可以检查写-写冲突：是否另外一个用户已经占据了需要的时间段。依赖检查还能检查读-写冲突。例如，它能检查所需的时间段是空的并且那天的预约少于6个。

如果依赖检查检查出了一个冲突，那么，Bayou将调用操作的合并过程。该过程将改变那个已执行的操作以使得可以获得相似效果，但避免了冲突。例如，就日记来说，合并过程可以选择相近的另一个时间段代替，就像我们上面提到的，它可以使用一个简单的优先级方案以决定哪个预约更重要，然后采用那一个。合并过程可能在发现一个操作的合适变换时失败，这种情况下系统将报错。然而合并过程的影响是确定的——Bayou副本管理器是状态机。 583

讨论　Bayou和其他复制方案的不同之处在于它使得复制对于应用是不透明的。它利用应用语义提高数据的可用性，同时维持一个复制状态，那是我们所称为的*最终顺序一致性*。

这种方法的不足之处首先在于增加了应用程序员的复杂度，他必须提供依赖检查和合并过程。当大量的可能冲突需要检查并解决时，生成这两者都非常复杂。第二个不足是增加了用户的复杂度。不仅由用户处理所读的尝试性数据，而且用户指定的操作可能被改变。例如，用户在日记中登记了一个时间段，后来却发现登记已经“跳”到了邻近的一个时间段。给出用户一个清晰的指示，哪些数据是尝试性的，哪些数据是提交的，这一点非常重要。

Bayou使用的操作变换方法被用在支持CSCW（计算机支持的协同工作）的计算机系统中，那儿地理上分离的用户可能发生操作冲突 [Kindberg *et al.* 1996], [Sun and Ellis 1998]。该方法的实际应用限于冲突较少的应用、潜在的数据语义较简单的应用、用户可以处理尝试性信息的应用。

14.4.3　Coda文件系统

Coda文件系统来源于AFS系统（参见8.4节），其目标是完成一些AFS没能达到的需求，特别是除断连操作外的高可用性的要求。它是CMU的Satyanarayanan及其合作者承担的一个研究项目[Satyanarayanan *et al.* 1990; Kistler and Satyanarayanan 1992]。Coda的设计需求来源于CMU AFS项目和其他一些对局域网、广域网上的大型分布式系统的使用经验。

尽管在CMU的使用经验中发现，AFS系统的性能和易管理性令人满意，但是由于AFS只能提供非常有限的复制，使它的伸缩度受限，不适用于大规模地访问共享文件，如电子公告板系统和全球范围的数据库。

AFS提供服务仍然有提升可用性的空间。但AFS用户所经历的最常见困难来自服务器和网络组件的故障（或调度中断）。在CMU的系统规模下是每天发生一些服务故障，这些故障在几分钟到数小时内给用户造成了极为严重的不方便。

最后，AFS没有注意到一种计算机使用的新趋势——便携式计算机的移动使用。这就导致了下列需求：在计算机断连时不借助于工方式管理文件的位置，而能继续使用。

Coda就是瞄准这3个目标开发的，提供稳定的数据可用性。目标是提供一个共享文件存储，
584 并且在该存储全部或部分不可访问时可完全依赖本地资源继续操作计算机。Coda保留了AFS原来的目标，包括可伸缩性和仿真UNIX文件语义。

AFS的读写卷存储在一个服务器上，与之相比，Coda通过文件卷复制技术来提高文件访问操作的吞吐率和系统容错性。另外，Coda扩展了AFS的文件缓存机制，使客户未与网络连接时仍然能够继续操作。

我们在下文中将看到，Coda类似于Bayou系统（参见14.4.2节），也采用了乐观策略：它允许客户在网络分区情况下更新数据，只要冲突可能性较小并且冲突可随后修正。与Bayou类似，Coda使用了冲突检测；但与Bayou不同的是，它在进行检测时不考虑数据语义，并且它为解决副本之间的冲突只提供了非常有限的系统支持。

Coda体系结构 按照AFS的术语，Coda在客户计算机上运行的进程被称为Venus进程，在文件服务器上运行的进程被称为Vice进程。Vice进程就是我们称作的副本管理器，Venus进程是前端和副本管理器的混合体。它们扮演前端的角色，将服务的实现隐藏在本地客户进程中。由于它们管理文件的一个本地缓存，尽管和Vice进程的类型不同，它们仍是副本管理器。

掌握着一个文件卷的副本管理器集合被称为卷存储组（VSG）。在任何时候希望在这样的卷中打开一个文件的客户能访问的VSG某个子集，被称为可用的卷存储集（AVSG）。随着网络或服务故障使服务器变得可访问或不可访问，AVSG的成员也在变化。

正常情况下，Coda文件访问过程是和AFS的相似的。当前AVSG中的任何一个服务器提供文件的缓存拷贝给客户计算机。在AFS中，经由一个回调承诺机制，客户被告知文件的变化，而Coda依靠一个对每个副本管理器进行更新分布的附加机制。当文件关闭时，修改过的文件广播到AVSG中的所有服务器。

在Coda中，断连操作被认为发生于AVSG为空时。这可能是由于网络或服务器故障，或可能是客户计算机有意离线的结果，比如一个笔记本电脑的情况。断连操作的有效性依靠于客户计算机缓存中的文件是否能供用户继续工作。为了保证这一点，用户必须和Coda系统合作以产生应该缓存的文件列表。Coda提供了一个工具，用它来记录连接时使用的文件历史表，并用这个作为预测离线时文件使用情况的基础。

Coda的一个设计原则是服务器上的文件拷贝比客户计算机缓存中的更可靠。尽管逻辑上有可能构造一个文件系统，使其完全依靠客户计算机上的缓存拷贝，但这样的系统不大可能提供满意的服务质量。Coda服务器的存在是为了提供必要的服务质量。客户计算机缓存中的
585 文件拷贝被认为是有效的，只要它们的当前数据能周期性地与服务器上的拷贝进行验证。在断连操作的情况下，重新验证发生在断连操作停止并且将缓存文件和服务器上的文件合并时。最坏情况下，需要一些手工干预来解决不一致或冲突。

复制策略 Coda的复制策略是乐观的——在网络分区和断连操作期间，仍然允许进行文件修改。它依靠每个文件上附加的Coda版本向量（CVV）。CVV是一个时间戳向量，其中每一个元素对应着每个在相关VSG中的服务器。CVV中的每个元素是一个估计值，是服务器上文件的修改次数的估计。CVV的目的是提供足够的关于文件副本的更新历史，使得能够检测出潜在的冲突、提交手工干预和提交对过时复制的自动更新。

如果一个站点的CVV大于或等于所有其他站点相对应的CVV（10.4节定义了时间戳向量v_1和v_2，$v_1 \geqslant v_2$的定义），那么不会发生冲突。旧的复制（有严格小的时间戳）包括一个较新的副本中的所有更新，于是它们可以自动地将数据更新。

不是这种情况时，对于两个CVV，即当$v_1 \geqslant v_2$和$v_2 \geqslant v_1$均不成立时，那么存在一个冲突。每个副本至少反映了其他副本没反映的一个更新。一般情况下，Coda不自动解决冲突。文件被称为“不可操作”并且文件所有者被告知有冲突。

当一个修改的文件关闭后，由客户上的Venus进程发送一个更新消息（包括当前的CVV和文件的新内容）到当前的AVSG中的每一个站点。每个站点的Vice进程检查CVV，如果CVV比当前它掌握的要大，则存储文件新内容并返回一个肯定的确认。然后Venus进程计算一个新CVV：对更新消息进行肯定应答的服务器，增加它的修改记数，并且发放新的CVV给AVSG中的成员。

由于消息仅仅发送给AVSG中的成员而不是VSG，不在当前AVSG中的服务器收不到新的CVV。因此，对本地服务器，CVV经常包含一个正确的修改记数，但对于非本地的记数一般要更小，因为仅当服务器收到一个更新消息时它们才更新。

下面的例子说明在3个站点上使用CVV管理文件副本的更新。可以在[Satyanarayanan *et al.* 1990]中发现使用CVV管理更新的更多细节。CVV是基于Locus使用的复制技巧 [Popek and Walker 1985]。

例　考虑对卷中的一个文件F的一个修改序列，它在3个服务器S_1，S_2和S_3上有副本。对于F的VSG是$\{S_1, S_2, S_3\}$。F在同一时间被两个客户C_1和C_2修改。由于一个网络故障，C_1仅能访问S_1和S_2（C_1的AVSG是$\{S_1, S_2\}$），C_2仅能访问S_3（C_2的AVSG是$\{S_3\}$）。

1. 开始，F的CVV在所有的3个服务器上是相同的，比如[1，1，1]。
2. C_1运行一个进程，它打开F，修改F，然后关闭。C_1的Venus进程广播一个更新消息给它的AVSG即$\{S_1, S_2\}$，最后导致了F的一个新的版本和S_1、S_2上的一个CVV[2, 2, 1]，但没有在S_3上做任何改变。
3. 同时，C_2运行两个进程，每一个打开F，修改F，然后关闭。在每一次修改后，C_2的Venus进程广播一个更新消息到它的AVSG即$\{S_3\}$，最后导致了F的一个新的版本和在S_3上的一个CVV[1，1，3]。
4. 在以后的某个时间，网络故障修复了，C_2进行检查看以前VSG的不可访问的成员是否变成可达的了（进行这个检查的进程在下面描述），发现S_1和S_2现在是可达了。故包含F的卷修改它的AVSG为$\{S_1, S_2, S_3\}$，并且从新的AVSG的所有成员请求CVV。当它们到达时，C_2发现S_1和S_2每一个都有CVV[2,2,1]然而S_3有[1,1,3]。这是一个冲突，需要手工干预以使F能以最小地信息丢失方式进行更新。

另一方面，考虑一个相似但是更简单的情况，有上面提到的事件顺序，但删去了条目3，所以F没被C_2修改。S_3上的CVV因此没有变化，还是[1,1,1]，当网络故障修复后，C_2发现在S_1和S_2（[2,2,1]）的CVV控制了S_3。在S_1或S_2的文件的版本应该替代在S_3上的。

在一般的操作中，Coda的行为表现和AFS相似。一个缓存访问不中，对于用户是透明的并且仅仅是性能上的问题。从多个服务器上某些或全部文件卷的副本中获得的好处有：

- 对于可以访问至少一个副本的客户，可访问在一个复制卷上的文件。

- 系统中的性能可以通过分担客户请求的服务负荷得到提高。这个请求现作用于所有具有副本的服务器之间的一个复制卷上。

在断连操作（对于一个卷，没有一个服务器能被客户访问）中，一个缓存访问不中阻止了进一步地进行操作，计算被挂起直到重新连接上或用户放弃了进程。因此，在断连操作开始前加载缓存非常重要，这样可以避免缓存访问不中。

简而言之，和AFS比较，Coda通过文件在多个服务器上的复制和用户能在缓存范围之外操作，改善了可用性。两种方法都依靠使用一个乐观策略，用于在有网络分区的情况下检测更新冲突。这两种机制是相互补充的，又是相互独立的。例如，一个用户可以使用断连操作的好处，即使需要的文件卷被存在单个服务器上。

586
~
587

更新语义 当客户打开一个文件时，Coda提供的当前保证比AFS要弱，这反映了乐观更新策略。在AFS的当前保证中，单个服务器S被服务器集合$\bar{S}$（文件的VSG）代替。客户C可以访问$\bar{S}$的一个子集（C看到的文件的AVSG）。

非正式地，在Coda中一个成功的*open*提供的保证是这样的：它从当前的AVSG提供了F的最近拷贝，并且如果没有服务器是可访问的，那么如果有一个本地的缓存拷贝是可用的话，它将被使用。一个成功的*close*保证文件已经传播给当前可访问的服务器集合，或如果没有服务器可用，这个文件便被标识以便在最早的时机传播出去。

考虑到丢失回调的影响，将在AFS中应用的标记进行扩展，可以产生这些保证的一个更精确的定义，这些定义除了最后一个，都有两种情况：首先，$\bar{S} \neq \phi$，代表所有情形中AVSG不为空；然后处理断连操作：

在一个成功的 *open*之后:
$$(\bar{S} \neq \phi \ and \ (latest(F, \bar{S}, 0)$$
$$or \ (latest(F, \bar{S}, T) \ and \ lostCallback(\bar{S}, T) \ and \ inCache(F)))$$
$$or \ (\bar{S}=\phi \ and \ inCache(F))$$

在一个失败的*open*之后:
$$(\bar{S} \neq \phi \ and \ conflict(F, \bar{S}))$$
$$or \ (\bar{S}=\phi \ and \ \neg \ inCache(F))$$

在一个成功的*close*之后:
$$(\bar{S} \neq \phi \ and \ updated(F, \bar{s}))$$
$$or \ (\bar{S}=\phi)$$

在一个失败的 *close*之后:
$$\bar{S} \neq \phi \ and \ conflict(F, \bar{S})$$

上述模型假定是一个同步系统。T是客户不知道别处有一个对它缓存中的文件做了更新的最长时间，$latest(F, \bar{S}, T)$指客户C的文件F的当前值是最近T秒$\bar{S}$所有服务器中的最新值，与F的拷贝没有冲突。$lostCallback(\bar{S}, T)$指在最近T秒由$\bar{S}$的一些成员发送了一个回调，但在C端没有收到。$conflict(F, \bar{S})$指当前s'中的一些服务器上的F值有冲突。

访问副本 为了访问一个文件的副本，在*open*和*close*上使用的策略是读一个/写所有方法的一个变种。对于*open*，如果一个文件的拷贝并不在本地缓存中，客户确定AVSG中的一个服务器作为首选服务器。首选服务器可以随机选择或基于性能标准，比如物理上接近或根据服务器负荷。客户从一个首选服务器上请求一个文件属性和内容的拷贝，并且在接收时，检查AVSG中其他的所有成员以证实这个拷贝是最新可用版本。如果不是，AVSG中有最新版本的成员变为首选站点，文件内容将被重新获取，并且AVSG成员被告知一些成员有过时的副本。

当完成读取时，一个回调承诺被建立在那个首选服务器上。

当一个文件在客户修改后关闭时，将使用一个组播远程过程调用协议，将它的内容和属性传送到AVSG的所有成员。这将使一个文件在每个复制站点有当前版本的可能性最大。它并 588
不确保每个站点都有当前的版本，因为AVSG并不包括所有VSG成员。通过让客户传播文件的修改到各个复制场地，可以减轻服务器负载（在*open*操作发现一个过时的副本时，才需要服务器）。

因为在所有的AVSG成员中维持回调状态是昂贵的，回调承诺仅维持在首选服务器上。但这引入了一个新的问题：一个客户的首选服务器并不在另一个客户的AVSG中。如果是这种情况，第二个客户的一个更新将不会导致一个回调给第一个客户。下一小节将讨论对这个问题的解决方法。

Cache一致性　Coda的当前保证意味着每个客户的Venus进程必须在下面事件发生的*T*秒内检测到它们：

- 扩大一个AVSG（由于需要访问一个先前不可访问的服务器）；
- 收缩一个AVSG（由于一个服务器变得不可访问）；
- 回调事件丢失。

为了实现这点，Venus每*T*秒发送一个探询消息到文件的VSG中的所有服务器，表示它已经在它的缓存中。仅从可访问的服务器收到应答。如果Venus从一个先前不可访问的服务器收到应答，它扩大对应的AVSG并且丢弃相关卷的文件的回调承诺，这样做是因为缓存中的副本可能不再是新的AVSG中的最新可用版本了。

如果它不能从一个先前可访问的服务器*f*接收到应答，Venus就收缩对应的AVSG。并不需要对回调进行修改，除非收缩由丢失一个首选服务器导致，在这种情况下，那个服务器的所有回调承诺必须丢弃。如果一个应答显示一个回调消息发送了但没有收到，那么回调承诺将在对应的文件上被丢弃。

剩下的问题是，一个服务器没有收到更新，因为它没有在一个执行这个更新的另一个客户的AVSG中。为了处理这种情况，Venus发送一个卷版本向量响应每个探询消息。卷版本向量包含一个所有在卷中的文件的CVV的摘要。如果Venus检测到在卷CVV中任何的不匹配，那么一些AVSG成员肯定有一些文件版本是过时的。尽管过时的文件可能不是在本地缓存的，Venus使用悲观的假设，丢弃所有它具有的相关文件上的回调承诺。

值得注意的是，Venus只探询持有缓存副本的文件的VSG中的所有服务器，一个探询消息用于更新AVSG并检查某一文件卷中的所有文件的回调。这一点再加上相对大的T值（在实验性实现中这个值是在10分钟的量级上），意味着探询消息并不是具有大量服务器和广域网的 589
Coda的可伸缩性的障碍。

断连操作　在短暂的离线期间，诸如由于不可预料的服务中断而导致的离线，Venus采用最近最少使用的缓存替代策略，最大限度地避免离线的文件卷上的缓存失配。但除非采取另外的策略，否则，一个客户在断连模式下不访问不在缓存的文件或目录是不可能的。

因此Coda允许用户指定一个文件的优先级表和Venus应该力争保留在缓存中的目录。最高层的对象被认为是不变的，它们必须时时保持在缓存中。如果本地硬盘足够大以容纳所有的高层对象的话，用户可被保证它们将一直可以被访问。由于要精确地知道用户动作的任一次序将产生什么样的文件访问是非常困难的，因此Coda提供了一个工具使得用户能够将动作序

列加以分组；Venus 记录由访问序列生成的文件引用并且将它们标上一个给定的优先级。

在断连操作结束时，开始整合过程。对于每个在断连操作期间进行了修改、创建或删除的缓存文件或目录，Venus执行一系列的更新操作以使得AVSG副本和缓存副本相同。整合从每个缓冲文件卷的根起从顶向下处理。

在整合期间，由于其他客户更新AVSG复制，因此可能会检测到冲突，一旦发生了这样的情况，缓存的副本被存储在服务器上的一个临时位置，并且通知发起整合的用户。这种方法基于Coda采用的设计理念：Coda分配给基于服务器的副本的优先级要高于缓存中的副本的优先级。临时副本存储在一个合作卷中，它和一个服务器上每一个卷相连。合作卷很像传统UNIX系统中的*lost+found*目录。合作卷仅镜像部分文件目录结构，用于存放临时数据，并不怎么需要额外的存储，因为合作卷几乎总是空的。

性能 Satyanarayanan等[1990]用仿真AFS用户（从5~50个）的基准负载比较了Coda和AFS的性能。

没有复制，AFS和Coda的性能没有大的差别。Coda有三倍的复制。在5个典型用户的基准负载下，它完成负载的时间超过不进行复制的AFS的5%，但是，同样是三倍的复制，在50个典型用户的基准负载下，它完成负载的时间增加了70%，而在同样的负载下，对于不进行复制的AFS，完成负载的时间增加了16%，主要原因是与复制相关的开销，实现的不同也是性能不同的部分原因。

讨论 上面我们指出Coda和Bayou相似之处在于Coda也使用了乐观方法以获得高可用性（尽管它们在其他一些方面不同，比如一个管理文件，而另一个管理数据库）。我们也描述了Coda如何使用CVV检查冲突，并不考虑存储在文件中的数据的语义。这个方法可以检测潜在的写-写冲突但不能检查读-写冲突。这些只是“潜在”的写-写冲突，因为在应用语义的
590 层次上来说，可能并不存在冲突：客户可能适当地更新了文件中的不同的对象，因此一个简单的自动合并将是可能的。

Coda所用的非语义的冲突检测和手工解决的方法在一些情况下是可行的，尤其在需要人为判断的应用或系统中没有数据语义的知识的情况下。

目录是Coda的一个特殊情况。在冲突解决中自动地维持这些关键对象的完整性是有可能的，因为它们的语义相对简单：能使目录发生变化的只有目录项的插入和删除。Coda用它自己的方法解决目录问题，与Bayou的操作变换方法有相同的效果，但是Coda直接合并相互冲突的目录的状态，因为它没有记录客户完成的操作。

14.5 复制数据上的事务

到目前为止，在我们考虑的系统中，客户只在对象的复制集合上请求一个单独的操作。第12章和第13章解释了事务是一个或多个操作的序列，并具有ACID性质。对14.4节中的系统，事务系统中的对象可以通过复制来提高可用性和性能。

对客户而言，复制对象上的事务看上去应该和没有复制对象的事务一样。在无复制的系统中，事务以某种次序执行，来确保执行等价于一个串行化执行序列。作用于复制对象的事务应该和它们执行在单一对象上的事务具有一样的效果。这种性质叫做*单拷贝串行化*。该性质与*顺序一致性*非常相似，但不能混淆。顺序一致性考虑执行的有效性，并不考虑将客户的操作组合后放入到一个事务中。

每一个副本管理器为自己的对象提供并发控制和可恢复性。本节假定并发控制是通过两阶段加锁来实现。

一个副本管理器出了故障，不能再提供服务，但是其他成员在它不可用的时候，继续提供服务，这使恢复问题变得复杂。当副本管理器恢复故障后，考虑到在不可用期间发生的所有变化，它需要从别的副本管理器获取信息以恢复对象的当前值。

本节首先介绍处理复制数据上事务的系统体系结构。体系结构上的问题包括：一个客户请求能否寻址到某个副本管理器；为了成功完成一个操作需要多少副本管理器；是否某个接受客户请求的副本管理器能够推迟转发请求，直到事务完成；以及如何实现两阶段提交协议。 591

单副本串行化的实现可以通过读一个/写所有来说明：这是一个简单的复制方案，其中读操作由一个副本管理器完成，写操作由所有的副本管理器执行。

本节然后讨论服务器崩溃和恢复时如何实现复制方案，并介绍了读一个/写所有复制方案的一个变种，即可用拷贝复制方法——读操作由任何一个副本管理器完成，写操作由所有当前可用的副本管理器执行。

最后，本节提出了3种复制方案，这3种方案在出现网络分区，副本管理器集合被分为子组时，均可正确工作。

- *带验证的可用拷贝*　每一个分区中应用可用的拷贝复制，当修复分区后，通过一个确认过程来处理任何不一致情况。
- *法定数共识*　每个子组必须是一个法定组（意味着它有足够的成员），当出现分区时，能够允许它继续提供服务。当分区修复后（和当一个副本管理器在故障后重新启动时），副本管理器通过恢复过程获得它们的最新对象。
- *虚拟分区*　法定数共识和可用拷贝的结合。如果一个虚拟分区有一个法定组，它就能使用可用的拷贝复制。

14.5.1　用于复制事务的体系结构

在前面几节已考虑的系统范围中，一个前端可以将客户请求组播到副本管理器组或发送请求到某个副本管理器，这个副本管理器负责处理请求并响应客户。Wiesmann等人[2000]、Schiper和Raynal[1996]考虑了组播请求的情况，我们在此不再赘述。从现在开始，我们假定前端发送请求到逻辑对象的副本管理器组中的某一个副本管理器。在主拷贝方法中，所有的前端和一个“主”副本管理器通信来执行某个操作，由这个副本管理器负责更新备份。另外，前端可以和任一副本管理器通信来执行某个操作，但是这种情况下副本管理器之间的协调问题更加复杂。

收到请求的副本管理器针对特定对象执行操作，它负责协调组中具有那个对象拷贝的其他副本管理器。为了成功地完成一个操作，不同的复制方案有不同的规则，需要不同数量的副本管理器。例如，在读一个/写所有方案中，“读”请求可以由单一的副本管理器来执行，然而写请求需要由组中所有副本管理器来执行，如图14-10所示（不同对象可以有不同数目的副本）。法定数共识方案被用来降低执行一个更新操作所必须的副本管理器的数目，但它的代价是增加了执行只读操作的副本管理器的数目。 592

另一个问题是和前端相连的副本管理器是否应该延迟转发更新请求到别的管理器直到一个事务提交以后——即所谓的更新传播的惰性方法；或者相反，是否副本管理器应该在它提交事

务以前转发每一个更新请求到所有的管理器——及时方法。惰性方法具有很好的特性：它降低了副本管理器之间的通信量。但是在该方法中，需要仔细考虑并发控制。惰性方法有时用在主拷贝复制中（见下文），主副本管理器可将事务串行化。但如果几个不同的事务试图访问某对象在一个组中不同管理器上的副本时，为了确保事务能在所有的副本管理器上正确执行，每一个副本管理器必须知道其他管理器的执行情况。此时，及时方法是惟一可用的方案。

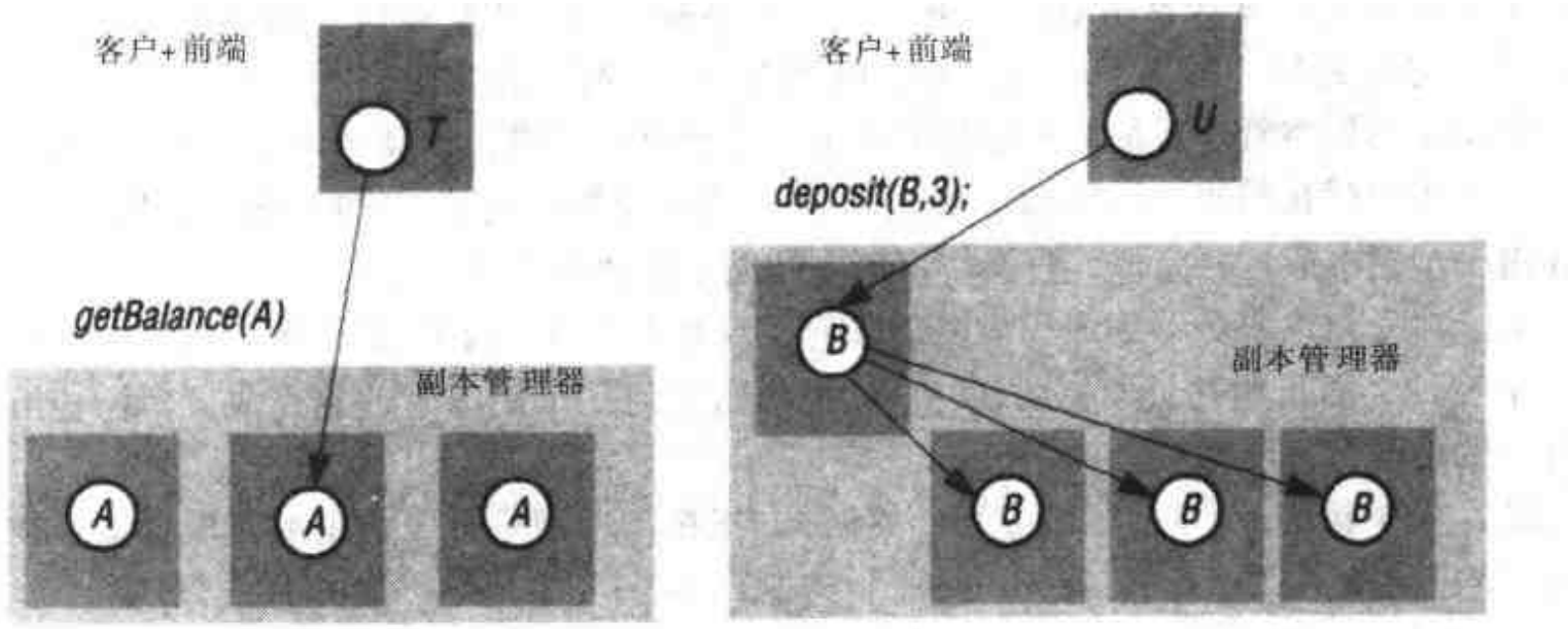


图14-10 复制数据上的事务

两阶段提交协议 在有复制数据的情况下，两阶段提交协议变为两层嵌套的两阶段提交协议。以前，一个事务的协调者和其他参与者进行通信。但是如果协调者或参与者是一个副本管理器时，那么它将和其他的副本管理器通信，它将在事务期间发送请求给这些副本管理器。

简而言之，在第一阶段，协调者发送*canCommit?*给参与者，参与者再将它传送给其他的副本管理器，并在回答协调者之前收集它们的应答。在第二阶段，协调者发送*doCommit*或*doAbort*请求，这个请求将传送给副本管理器组成员。

主拷贝复制 主拷贝复制可用在事务环境。在这个方案中，所有的客户请求（不管是否只读）直接送到一个主副本管理器（见图14-4）。对于主拷贝复制，并发控制被应用于主副本管理器上，当提交一个事务时，主副本管理器和备份副本管理器间相互通信，然后用及时方法应答用户。这种形式的复制允许当主副本管理器出故障时，一个备份副本管理器能一致地接管它。在惰性方法中，主副本管理器在它更新备份前就响应前端。在那种情况下，一个替代了故障前端的备份可能不会有数据库的最新状态。

593

读一个/写所有 我们使用这个简单方案来说明如何通过每个副本管理器上的两阶段锁，来获得单拷贝串行化，这里，前端可以和任何的副本管理器通信。每一个写操作必须在任何副本管理器上执行，在操作影响到的每个对象上加一个写锁。每个读操作由副本管理器执行，它在受此操作影响的对象上加一个读锁。

考虑在同一对象上的不同事务的两个操作：任何两个写操作在所有副本管理器上请求相冲突的锁；在单一的副本管理器上，一个读操作和一个写操作会请求相冲突的锁。结果，获得了单拷贝串行化。

14.5.2 可用拷贝复制

简单的读一个/写所有复制并不是一个现实的方案。因为当副本管理器崩溃或发生通信故

障时，由于一些副本管理器不可用，这种方案就不可能实现。可用拷贝复制方案允许某些副本管理器暂时不可用。这个方案是客户对一个逻辑对象的读请求可以被任何可用的副本管理器执行。但是一个客户的更新请求必须被组中具有那个对象拷贝的所有可用副本管理器执行。副本管理器中可用成员的概念和14.4.3节描述的Coda中的可用卷存储组非常相似。

在正常情况下，一个工作的副本管理器接收并执行客户的请求。读请求可由收到请求的副本管理器执行。写操作由收到请求的副本管理器和所有组中的可用副本管理器执行。例如，在图14-11中，事务*T*的*getBalance*操作由*X*执行，然而它的*deposit*操作由*M*，*N*和*P*执行。每个副本管理器上的并发控制影响本地的执行。再例如，在*X*上，事务*T*已经读了*A*因此事务*U*并不允许用*deposit*操作来更新*A*，直到事务*T*完成。只要可用的副本管理器集没有变化，本地的并发控制将和读一个/写所有一样可获得单副本串行化。但是，如果相冲突的事务在进行过程中，副本管理器出故障或在恢复，就不是这种情况了。

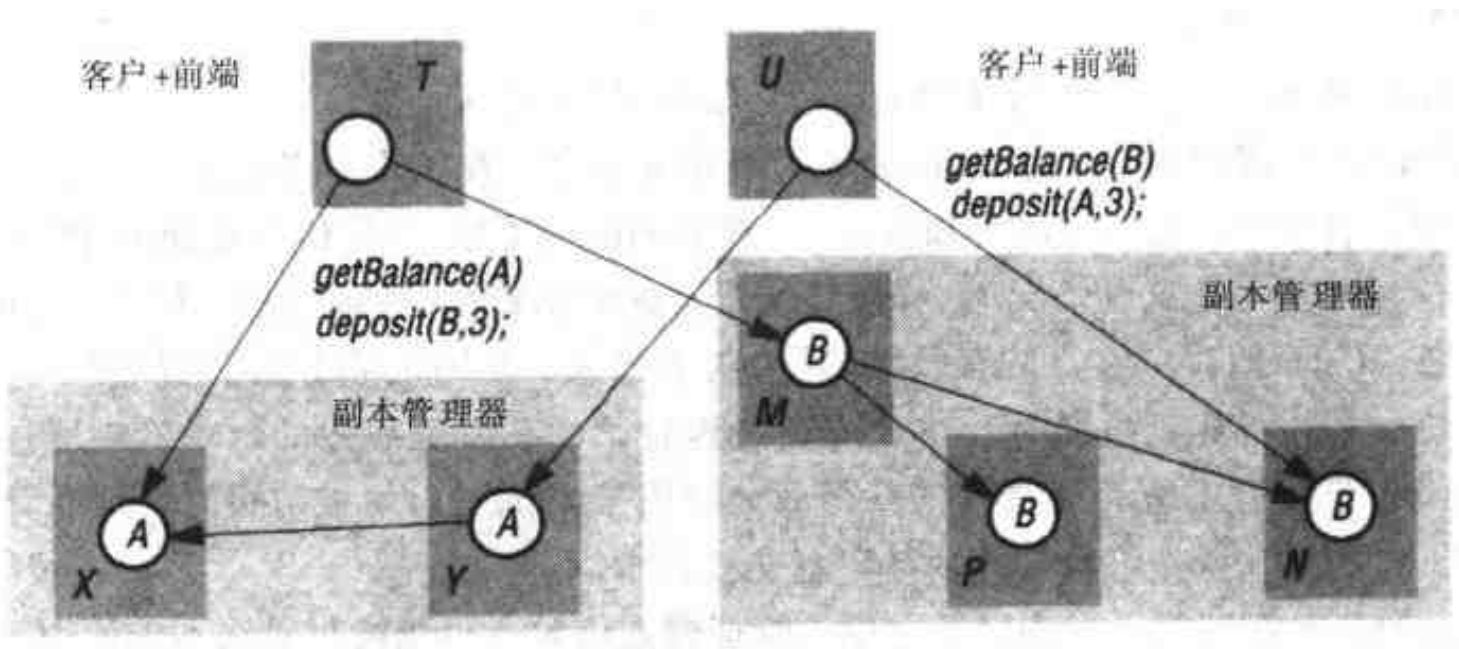


图14-11　可用的拷贝

副本管理器故障　我们假定副本管理器故障只是由于崩溃而引发的。当崩溃的副本管理器由一个新的进程取代时，它从恢复文件来还原对象的提交状态。前端使用超时检查来判断某个副本管理器当前是否可用。当客户发送一个请求到崩溃的副本管理器后，前端将会超时，它重新尝试并将请求发送到组中的另一个副本管理器。如果请求被某个副本管理器接收后，由于副本管理器尚未完全从故障中恢复而导致对象数据过时，副本管理器将拒绝请求，这时前端将发送请求到组中的另一个副本管理器。

就事务而言，单副本串行化要求崩溃和恢复都是串行化的。根据是否能够访问某个对象，一个事务在完成之后或在启动之前能够判断是否存在故障。当不同的事务观察到相互冲突的故障情况时，单副本串行化将不能得到满足。

考虑图14-11中的情况，副本管理器*X*在*T*已经执行了*getBalance*之后出故障，副本管理器*N*在*U*完成*getBalance*后出故障。假定是在*T*和*U*进行*deposit*操作以前副本管理器*X*和*N*出了故障。这暗示着*T*的*deposit*将在副本管理器*M*和*P*上执行，*U*的*deposit*将在副本管理器*Y*上执行，但是，副本管理器*X*的对于*A*的并发控制并不会阻止事务*U*在副本管理器*Y*上更新*A*。同样副本管理器*N*上对*B*的并发控制不会阻止*T*在副本管理器*M*和*P*上更新*B*。

这种现象与单副本串行化需求是相违背的。如果这些操作在对象的单一副本上执行，那么它们是可串行化的。要么事务*T*在*U*之前，要么*U*在事务*T*之后。这能保证一个事务读取另一

个事务设置的值。对象拷贝的本地并发控制并不能在可用的复制方案中保证单拷贝串行化。

由于写操作直接作用于所有可用的拷贝上，本地并发控制确实能保证在一个对象上的冲突写是可串行化。相反，一个事务的读操作和另一个事务的写操作并不一定影响对象的同一个拷贝。因此，该方案需要额外的并发控制方法以防止一个事务的读操作和另一个的写操作相互依赖而形成一个环。如果对事务而言，故障和对象复制的恢复是串行化的，那么这样的依赖性将不会产生。

本地验证 我们把额外并发控制过程称为本地验证。本地验证用来确保任何故障或恢复事件不会发生在事务的执行过程中。在我们的例子中，当T已经从X上的一个对象进行了读操作，X的故障一定出现在T完成以后。同样，当T试图更新对象时发现N出了故障，那N的故障一定在T之前，即：

594
~
595

N出故障→T在X上读对象A；T在M和P上写对象B→T提交→X出故障

同样对事务U而言，有：

X出故障→U在N上读对象B；U在Y上写对象A→U提交→N出故障

本地验证过程确保两个不相容的次序不会同时发生。在一个事务提交以前，它检查事务已访问的副本管理器的任何故障（和恢复）。在上面的例子中，T将检查发现N仍然不可用，而X、M和P仍然可用。在这种情况下，T能够提交。这暗示着在T验证之后、U验证之前X出现故障。换言之，U的有效性是在T的有效性之后的。U失效了是因为N已经出现故障了。

每当某个事务观察到故障时，本地验证过程将试图和发生故障的副本管理器通信来确信它们仍然没有恢复。本地验证过程的其他部分被用来测试当对象被访问时，副本管理器没有发生故障，这些部分操作可以并入两阶段提交协议中。

当正常的副本管理器不能和另外的副本管理器通信时，可用拷贝算法不能使用。

14.5.3 网络分区

复制方案需要考虑网络分区的可能性。网络分区将一个副本管理器组分为两个或更多的子组，分区使一个子组中的成员可相互通信，但不同子组中的成员不能通信。例如，在图14-12中，收到*deposit*的副本管理器不能将其发送到*withdraw*请求的副本管理器。

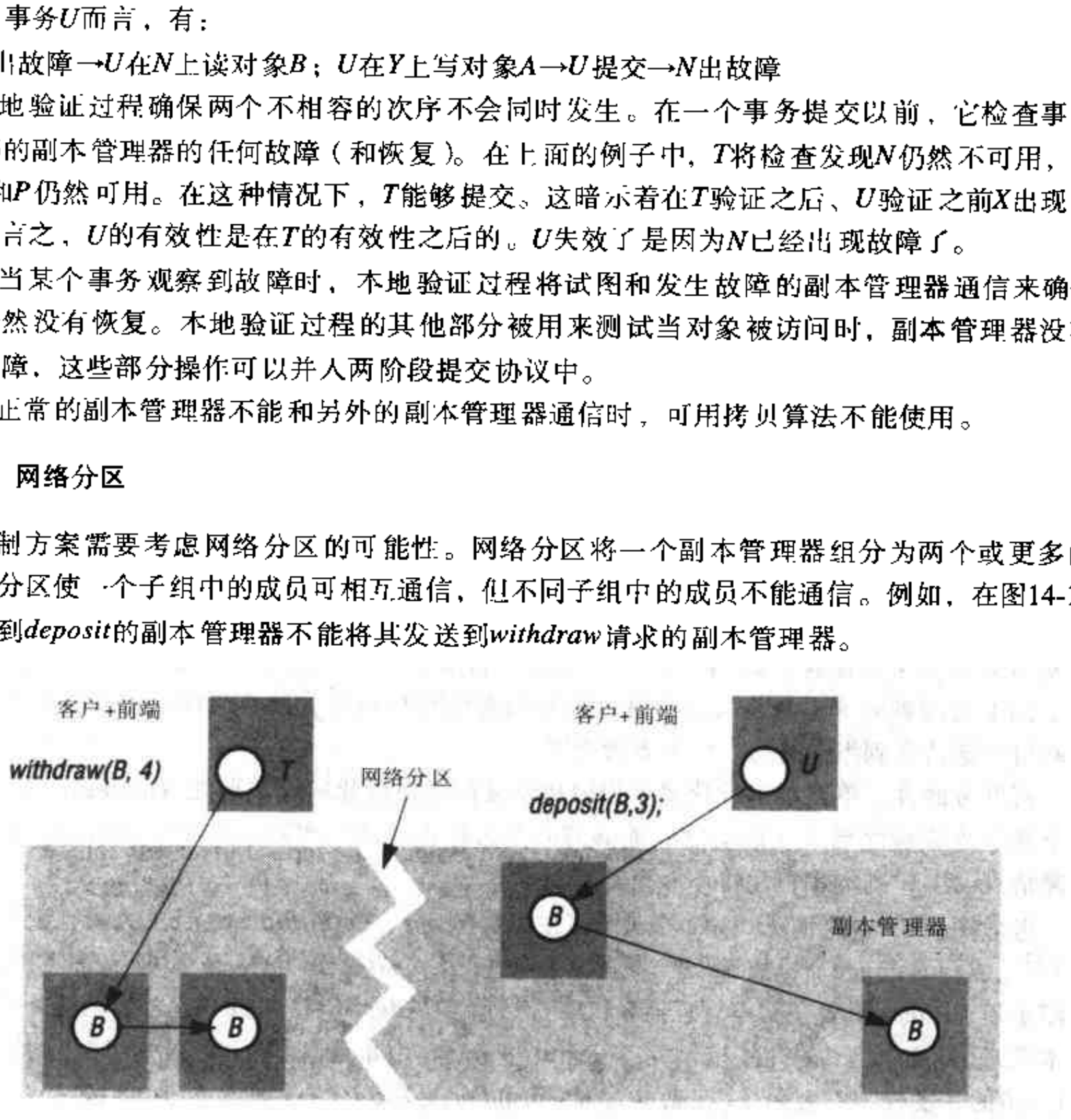


图14-12 网络分区

596

复制方案的设计基于这样的假定：网络分区最终将被修复。因此单个分区中的副本管理

器必须保证在分区期间它们执行的任何请求在分区修复后不会造成不一致。

Davidson等人[1985]讨论了多种不同的方法。按照不一致是否容易发生，这些方法可分类为乐观方法和悲观方法。乐观方法在分区期间不限制可用性，然而悲观方法却有所限制。

乐观方法允许在所有的分区中进行更新——这可能会导致分区的不一致，这必须在分区修复后解决。这种方法的一个例子是可用拷贝算法的一个变种，即在分区中允许进行更新，并且当分区恢复时，更新是有效的——任何违背单拷贝串行化准则的更新将被丢弃。

即使没有分区，悲观算法也限制可用性，但它阻止了在分区时任何不一致的产生。当一个分区恢复时，所要做的是更新对象的拷贝。法定数共识方法是悲观方法，它允许在主分区中进行更新并当分区修复时将更新传给其他的副本管理器。

14.5.4 带验证的可用拷贝

带验证的可用拷贝算法可在每一个分区内使用。这种乐观方法即使在分区期间也维持读操作的正常层次的可用性。当一个分区恢复后，需要对发生在不同分区中的可能相互冲突的事务进行验证。如果验证失败，必须采取某些步骤来克服这种不一致。如果没有分区发生的话，相互冲突的两个事务之一将被延迟或放弃。但是，当分区存在时，冲突的事务可以在不同的分区中提交。这种情况发生后的惟一选择就是放弃其中的一个。这需要在对象中进行某些变化，甚至在某些情况下要补偿现实世界中的影响，例如银行的账户透支。当应用可以进行补偿行为时，乐观方法才是可行的。

可利用版本向量来确认相互冲突的写操作。这些方法在14.4.3节中已经描述过，已被用于Coda文件系统中。这种方法并不能检测到读-写冲突，但在事务多是访问单个文件并且读-写冲突不重要的系统中，这种方法能够很好地工作。它并不适合类似银行例子之类的应用，因为这种应用的读-写冲突很重要。

Davidson[1984]使用前驱图来检测分区间的不一致。每一个分区维持着一个被事务读写操作影响的对象的日志。这个日志用来构建一个前驱图，图的结点是事务，它的边代表了事务读写操作之间的冲突。这样一个图应该不包含任何环，因为并发控制已经应用于分区中。验证过程取出分区的前驱图并在不同分区中的事务之间加上代表冲突的边。如果最终的图包含了环，那么验证失败。 597

14.5.5 法定数共识方法

一种阻止分区产生不一致的方法是制定一个规则，使操作只能在某一个分区中进行。由于不同分区中的副本管理器不能相互通信，因此每一个子组中的副本管理器必须独立地决定它们是否能进行操作。法定组是若干副本管理器组成的子组，它的大小使它能够执行这个操作。例如，如果拥有大多数成员是一个标准的话，那么含大多数成员的子组可形成一个法定组，因为其他的子组不会拥有大多数成员。

在一个法定数共识的复制方案中，一个逻辑对象上的更新操作可以成功地被副本管理器组中的一个子组完成。其他的组成员则有对象的过时的拷贝。版本号或时间戳可以用来决定拷贝是否已经更新。如果使用版本号，对象的初始状态是第一个版本，并且每一次变化后，我们有一个新的版本。每个对象的拷贝有一个版本号，而过时的拷贝有一个较早的版本号，操作只能应用于具有最新版本号的拷贝。

Gifford[1979a]开发了一个文件复制方案，其中一定数量的“选票”被分配给一个逻辑文件的副本管理器上的每个物理拷贝。选票可以看成是一个对使用特定拷贝的需求度权重。每个读操作必须在它从任何最新拷贝中进行读之前，先获得一个有R个选票的读法定组，每个写操作必须在它从任何最新拷贝中进行更新之前，获得一个有W个选票的写法定组。其中，R和W是副本管理器组，满足下面条件：

$$W > \text{总选票的一半}$$

$$R + W > \text{组选票的总数}$$

这就确保了任何一对（例如一个读法定组和一个写法定组或两个写法定组）必须包含相同的副本。因此，在分区出现时，不可能在不同的分区中进行同一拷贝上的冲突操作。

为了进行读操作，首先必须通过足够多的查询来收集一个读法定组，选票的数量不得少于R。并不要求所有这些拷贝都是最新的。由于每个读法定组和每个写法定组存在重叠，每个读法定组必定包括至少一个当前拷贝。读操作可在任何最新的拷贝上执行。

对于进行一个写操作，首先必须通过足够多的查询来收集一个写法定组，法定组中的成员必须是最新的，并且选票的数量不得少于W。如果没有足够的更新拷贝，那么一个不是当
598 前的文件被一个当前文件的拷贝替代，以使得法定组可以建立。由写法定组中的每个副本管理器进行写操作中的更新，提升所有对象副本的版本号，写操作的完成要报告客户。

然后，在余下副本管理器中的文件由写操作以后台任务方式进行更新。任何副本管理器，如果它的文件拷贝和写法定组相比有一个旧的版本号时，整个文件由来自法定组的一个文件替换。

在Gifford的复制方案中，两阶段锁可以用来进行并发控制。开始，用版本号查询来构造读法定组R时，每个副本管理器都被设置了一个读锁。当在写法定组W中执行写操作时，组中每个副本管理器上被设置了一个写锁。由于一个读法定组和一个写法定组重叠，并且两个写法定组也重叠，因此这些锁保证了单拷贝串行化。

副本管理器组的配置能力 加权选票算法的一个重要性质就是副本管理器组能够通过配置来提供不同的性能或可靠性。一旦通过它的选票配置建立了一个副本管理器组的可靠性和性能，写操作的可靠性和性能的提高可以通过减少W完成，同样可以通过减少R来提高读操作的可靠性和性能。

算法同样允许使用客户机上本地的文件拷贝，这就和使用文件服务器上的一样。客户机上的文件拷贝被认为是弱代表，并且经常分配0个选票。这就确保它们不会包含在任何法定组中。一旦某个读法定组构造成功，一个读操作可以在任何更新了的拷贝上执行。因此，如果一个文件的本地拷贝是最新的，则读操作可以在该拷贝上执行。弱代表可用来加速读操作。

Gifford的例子 Gifford给出了3个例子，这3个例子通过给一个组上的不同副本管理器分配权值和分配适当的R和W，显示了不同的特性。现在基于下面的表格，再现Gifford的例子。阻塞概率显示了在进行一个读或写操作时，不能获得法定组的概率。假设在发请求时，任何副本管理器都有0.01的概率不可用。

例1用来在一个有弱代表和单个副本管理器的应用中配置一个具有高读写率的文件。复制用来提高系统的性能，而不是可靠性。局域网上的一个副本管理器可以在75ms内被访问。两个客户已经选择在它们的本地磁盘上做弱代表，它们能在65ms内访问，结果导致了低延时和更少的网络交通。

例2用来配置一个有中读写率的文件，该文件主要通过一个局域网被访问。局域网上的副

本管理器分配两个选票，远程网络上的副本管理器每个被分配一个选票。读可以由本地的副本管理器来执行，但写操作必须访问本地副本管理器和一个远程副本管理器。客户为了更低的读延时，可以创建本地的弱代表。 599

例3用来配置一个具有高读写率的文件，比如在一个具有三副本管理器环境下的系统目录。客户能从任何副本管理器上读，文件不可用的概率很低。更新必须作用于所有的拷贝。同样，为了读操作延时低，可以在本地机上创建它们的弱代表。

		例1	例2	例3
延迟	副本1	75	75	75
(ms)	副本2	65	100	750
	副本3	65	750	750
选票配置	副本1	1	2	1
	副本2	0	1	1
	副本3	0	1	1
法定组大小	*R*	1	2	1
	W	1	3	3
文件包得到的性能:				
读	延迟	65	75	75
	阻塞概率	0.01	0.0002	0.000001
写	延迟	75	100	750
	阻塞概率	0.01	0.0101	0.03

法定数共识方法的缺点是由于需要从*R*个副本管理器中收集一个读法定组，读操作的性能被降低了。

Herlihy用抽象数据类型扩展了法定数共识方法 [Herlihy 1986]。这种方法允许考虑操作的语义，因此提高了对象的可用性。Herlihy的方法使用时间戳代替版本向量，这样的好处是不需要为了在执行一个写操作前得到一个新版本号而进行查询。Herlihy介绍说此方法的主要好处是使用语义知识可以提高法定组选择的数量。

14.5.6 虚拟分区算法

该算法由El Abbadi等人[1985]提出的，结合了法定数共识和可用拷贝两种方法。当出现分区时，法定数共识能够正确地工作而可用拷贝对于读操作的代价更低。虚拟分区是真实分区的一个抽象，包含了一个副本管理器的集合。注意术语“网络分区”意思是指将副本管理器分成许多部分，而术语“虚拟分区”是指这些部分本身。尽管它们并不通过组播通信相连接，但虚拟分区还是很像14.2.2节介绍的组视图。如果虚拟分区包含了充足的副本管理器而具有访问对象的读法定组和写法定组，那么一个事务能在该虚拟分区中操作。在这种情况下，这个事务使用可用拷贝算法，这样的好处是读操作只要访问某个对象的单一副本，因此可以通过选择“最近”的拷贝以提高性能。如果一个副本管理器发生故障并且在事务执行期间内虚拟分区变化了，那么这个事务将被放弃。由于所有存活的事务以同样的次序发现副本管理器的故障和恢复，这样能够确保事务的单副本可串行化。 600

每当虚拟分区的一个成员检测到它不能访问其他成员时——例如，当一个写操作没被确认时——它试图创建一个具有读、写法定组的新虚拟分区。

例如，设想我们有4个副本管理器*V*、*X*、*Y*和*Z*，每一个都有一个选票，并且写和读法定

组是$R=2$和$W=3$。开始，所有的副本管理器可相互连接。只要它们相连，它们就能使用可用拷贝算法。再例如，一个事务由读后紧跟着写的操作组成，它将在一个副本管理器（比如，V）上执行读，但在所有的4个副本管理器上进行写操作。

假设事务T开始在V上执行读操作时，V仍和X、Y、Z相连。发生网络分区如图14-13所示，其中V、X在一部分，Y和Z各在不同的分区。那么当事务T试图执行写操作时，V将注意到它已不能连接到Y和Z。

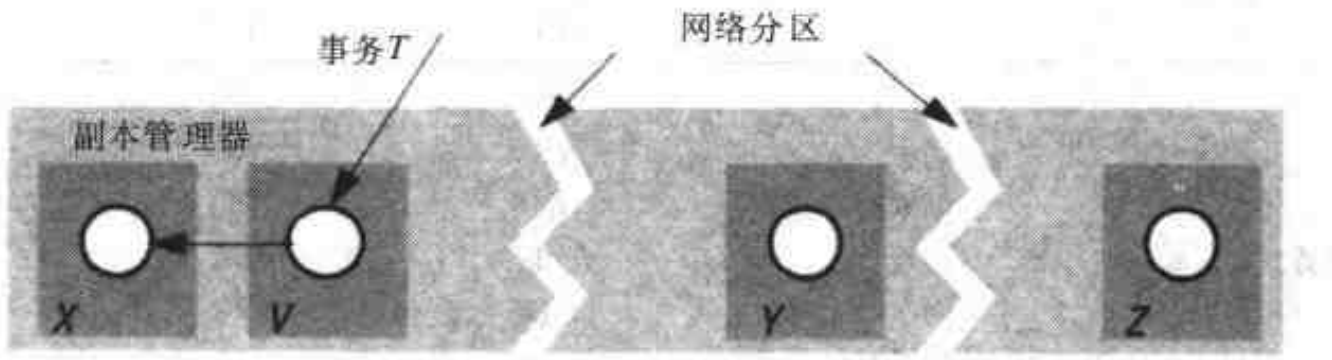


图14-13 两个网络分区

当一个副本管理器不能和它先前连着的副本管理器相连时，它不断地尝试直到它可以创建一个新的虚拟分区。例如，V将不断试图连接Y和Z直到它们中的一个或两个回应它，像图14-14所示，Y能被访问那样。副本管理器V、X和Y组成了一个虚拟分区，因为它们足以形成读法定组和写法定组。

在一个事务（比如事务T）已经在一个副本管理器上执行了操作期间，创建了一个新的虚拟分区，那么事务必须放弃。此外，一个新的虚拟分区内的副本必须通过拷贝其他的副本来进行更新。在Gifford的算法中可以使用版本号来决定哪个拷贝是最新的，所有的副本必须是最新的，因为读操作可在任何的副本上执行。

虚拟分区的实现 每个虚拟分区都有一个创建时间、一个潜在成员的集合和一个实际成员的集合。创建时间是逻辑时间戳。虚拟分区的实际成员具有相同的创建时间和成员关系。例如，在图14-14中，潜在的成员是V、X、Y和Z，而实际的成员是V、X和Y。

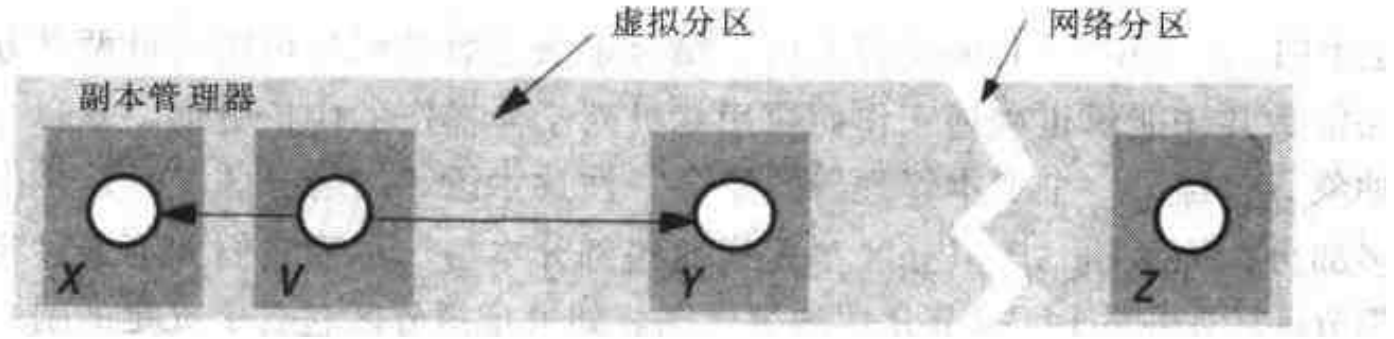


图14-14 虚拟分区

一个新虚拟分区的创建可以由一个合作协议来实现。这个协议由那些副本管理器可访问的潜在成员来实现。几个副本管理器可能同时试图创建一个新的虚拟分区。例如，设想在图14-13中的副本管理器Y和Z不断地试图连接其他的副本管理器，一段时间以后，网络分区部分获得恢复，以至于Y不能和Z通信但是两个组V、X、Y和V、X、Z却能相互通信。那么存在的一个危险是两个相互重叠的虚拟分区，如图14-15中的V_1和V_2可能都被创建了。

考虑在两个虚拟分区中执行不同事务的影响。在V、X、Y中的事务的读操作可能被应用于副本管理器Y上，这种情况下，它的读锁将不会和另外一个虚拟分区中事务的写操作设置的写

锁相冲突。重叠虚拟分区和单拷贝串行化相违背。

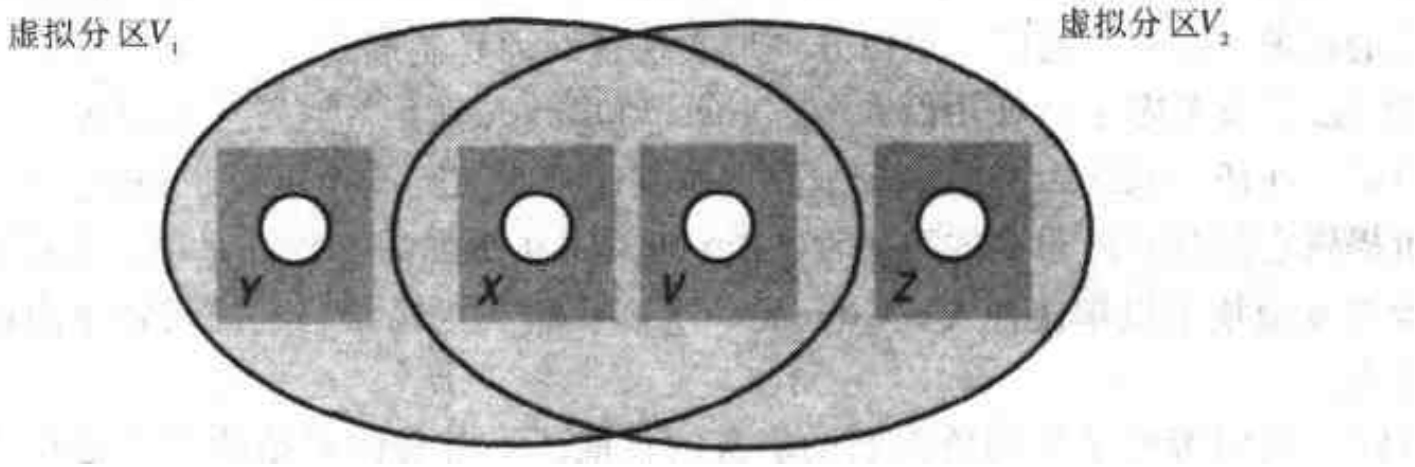


图14-15　两个重叠的虚拟分区

协议的目标是创建一个一致的新虚拟分区，即使在协议期间发生了真正的分区。创建一个新虚拟分区的协议有两个阶段，参见图14-16。

一个在阶段1回复*Yes*的副本管理器并不属于一个虚拟分区，直到它在阶段2中收到相应的确认消息。

601 ~ 602

阶段1：

- 发起者发送一个*Join*请求到每个潜在的成员。*Join*的参数是一个用于新虚拟分区的逻辑时间戳。
- 当某个副本管理器收到*Join*请求后，它比较请求的逻辑时间戳和自己当前虚拟分区的时间戳：
  - 如果请求中的逻辑时间戳大，那么它同意加入并应答*Yes*；
  - 如果请求中的逻辑时间戳小，那么它拒绝加入并应答*No*。

阶段2：

- 如果发起者收到了足够的*Yes*应答，从而具有读和写法定组，那么它通过发送一个*Confirmation*消息给同意加入的站点来创建一个新的虚拟分区。该虚拟分区的创建时间和实际成员列表以参数形式发送。
- 收到*Confirmation*消息的副本管理器加入新虚拟分区，并记录它的创建时间戳和实际成员列表。

图14-16　创建一个虚拟分区

在以上的例子中，在图14-13中显示的副本管理器*Y*和*Z*每一个都试图创建一个虚拟分区，具有较高逻辑时间戳的那个最终创建一个虚拟分区。

当分区并不经常发生时，这是一个有效的方法。在一个虚拟分区内，事务使用可用拷贝算法。

14.6　小结

复制对象是在分布式系统中获得高性能、高可用性和容错的重要手段。我们讨论了复制服务的体系结构，其中副本管理器掌管着对象的复制，前端使得复制对客户透明。客户、前端和副本管理器既可以是分开的进程，也可以在同一个地址空间中。

本章首先阐述了系统模型，其中每个逻辑对象都由物理副本的一个集合来实现。可以通过组通信来非常方便地更新这些复制。我们扩展了组通信内容，包括组成员关系服务和视图同步通信。

我们定义线性化和顺序一致性作为容错服务的正确性准则。这些准则表达了即使这些对象是复制的，服务器也必须保证它们与逻辑对象集合的单个映像等价。实践中最有意义的准则是顺序一致性。

603

在被动（主备份）复制中，通过直接将所有请求发送到一个选出的副本管理器，并在其出故障时选出一个备份代替它，可以获得容错。在主动复制中，所有的副本管理器独立地处理所有的请求。通过组通信，可以方便地实现这两种复制形式。

我们接下来考虑了高可用性服务。gossip和Bayou都允许当发生网络分区时在本地副本上进行更新。在任一系统中，副本管理器在恢复连接时相互交换更新。gossip以松弛因果一致性的代价提供它所具有的最高的可用性。Bayou提供了更强的一致性保证，采用了自动冲突检测以及操作变换技术以解决冲突。Coda是一个高可用文件系统，它使用版本向量以检测潜在的更新冲突。

最后，我们考虑了复制数据上的事务的性能。主备份体系结构存在这种情况，在前端可以与任何副本管理器通信的体系结构中也存在这种情况。我们讨论了事务系统如何允许副本管理器出现故障和网络分区。即使在某些环境下，即并不是所有的副本管理器都可达时，可用拷贝、法定数共识和虚拟分区的技巧仍能使事务中的操作继续进行。

练习

14.1 3个计算机一起提供一个复制服务。制造商声称每一台计算机平均5天出一次故障；一次故障一般需要4小时才能修复。那么这项复制服务的可用性是多少？

14.2 试解释为什么一个多线程的服务器不能看成是一个状态机。

14.3 在一个多用户的游戏中，玩家在一个公用屏幕上移动动画小人。游戏的状态被复制到玩家的工作站和一台服务器上，这个服务器包含控制游戏的所有服务。更新被组播给所有的复制。

（i）这些动画小人之间可能相互射弹，并且一段时间后可能被击中。那么这里需要什么样的更新次序？（提示：请考虑“投掷”，“碰撞”和“复活”等事件。）

（ii）玩家可能使用一个pick-up设备来玩这个游戏，那么对这个pick-up设备的操作需要什么样的次序？

604

14.4 一个路由器将进程p与另外两个进程q和r分开。p开始组播消息m后立刻就出现故障。如果组通信系统是视图同步的，接下来进程p将会怎样？

14.5 给你一个具有全序组播操作的组通信系统和一个故障检测器。是否能够只利用这些成分组成一个视图同步组通信系统？

14.6 同步有序的组播操作的传送语义和视图同步组通信中的传送视图的语义相同。在某个服务中，操作之间是因果排序的。该服务支持一个列表中的多个用户在特定服务上执行操作。试解释为什么从列表中删除用户是同步有序操作？

14.7 由状态迁移引起的一致性问题是什么？

14.8 对象o上的一个操作X引起o调用另一个对象o'上的操作。现在打算复制o但是不复制o'。试解释在调用o'上操作时可能出现的问题并给出一个解决方案。

14.9 试解释线性化和顺序一致性之间的不同。一般情况下，为什么在实现中后者更实际些？

14.10 在被动复制系统中，试解释为什么允许备份继续处理读操作的结果是顺序一致性，而不是线形化执行？

14.11 gossip体系结构能够应用于练习14.3描述的分布式游戏中吗？

14.12　在gossip体系结构中，为什么一个副本管理器需要保持一个“复制”时间戳和一个“值”时间戳？

14.13　在gossip系统中，前端有一个时间戳向量(3,5,7)，代表着它从一个有3个副本管理器的组中的成员接受到的数据。相应的，这3个副本管理器有时间戳向量(5,2,8)，(4,5,6)和(4,5,8)。哪一个或哪一些副本管理器能立刻满足前端的一个查询，前端最后的时间戳是什么？哪一个副本管理器能立刻从前端合成一个更新？

14.14　试解释为什么让某些副本管理器只读就可以提高gossip系统的性能？

14.15　对于一个简单的房间预订应用，写出（如Bayou中使用的）依赖性检查和合并过程的伪代码。

14.16　在Coda文件系统中，为什么在更新多个服务器上一个文件拷贝时，经常需要用户手工干预？

14.17　请设计一种方案来集成文件系统目录的两个副本，能够在断连操作下执行分离的更新。试使用Bayou的操作变换方法或者Coda方法。 605

14.18　在数据项A和B上应用可用拷贝复制，因此具有复制A_x、A_y和B_m、B_n。事务T和U定义如下：

T：*Read*(A)；*Write*(B,44)；∪：*Read*(B)；*Write*(A, 55)。

假定在副本上应用两阶段锁，设计一个T和U的交叉序列。利用这个例子来解释本地验证是如何确保单拷贝串行化的。

14.19　Gifford的法定数共识复制在服务器X、Y和Z上使用，这些服务器都有数据项A和B的副本。A和B副本的起始值是100并且在每个X、Y和Z上A和B的选票是1。同样对于A和B，$R = W = 2$。一个客户读A的值然后将它写到B上。

（i）当客户执行这些操作时，出现了一个分区，将服务器X和Y从服务器Z中分离了。描述当客户能访问X和Y服务器时，获得的法定组和发生的操作。

（ii）描述当客户仅能访问服务器Z时，获得的法定组和发生的操作。

（iii）分区修复了，然后另一个分区发生了，结果X和Z从Y中分离了。描述当客户能访问服务器X和Z时，获得的法定组和发生的操作。 606

第15章　分布式多媒体系统

15.1　简介
15.2　多媒体数据的特征
15.3　服务质量管理
15.4　资源管理
15.5　流适应
15.6　实例研究：Tiger视频文件服务器
15.7　小结

多媒体应用程序实时地生成和消耗连续的数据流。它们包含大量的音频、视频和其他基于时间的数据元素，并且其实质是实时地处理和发送数据元素（音频采样、视频帧）。迟到的数据元素是没有价值的，通常将其丢弃。

一个多媒体数据流的流规范通常包含如下部分：可以接受的从源到目的地传输数据的速率（带宽），每个数据元素的传输延迟（等待时间）以及数据元素的丢失率或丢弃率。在交互式应用程序中，等待时间是特别重要的。如果应用程序在某些数据丢失后可以重新调整到同步接收数据，那么一些程度较轻的多媒体数据的丢失是可以接受的。

为了满足多媒体和其他应用程序的需要而进行的资源计划分配和资源调度被称为服务质量管理。分配处理器处理能力、网络带宽和内存容量（用来缓冲那些被提前送到的数据元素）都很重要。系统根据服务质量请求的需要来分配它们。一个成功的QoS请求向应用程序发送一个QoS保证，并且将导致被请求的资源被预留和调度。[⊖]

15.1　简介

现代计算机可以处理像数字音频和数字视频数据这样的连续的、基于时间的数据流。其处理能力导致了分布式多媒体应用程序的发展，例如，网络视频库、因特网电话和视频会议。这些应用程序可以在当前用于普通目的的网络和系统上运行，但是它们的音频和视频质量经常难以令人满足。许多像大范围的视频会议、数字电视产品、交互式的电视以及视频监视系统这样对实时数据要求很高的应用程序不能被当前的网络和分布式系统技术所实现。

多媒体应用程序需要在有限时间内将多媒体数据流传输到用户端。音频和视频数据流被实时地生成和消耗，同时应用程序完整性的实质是实时地传输数据元素（音频采样，视频帧）。简单说，多媒体系统是实时系统：它必须按照外部决定的调度方案执行任务和传输结果。底层系统达到这些要求的程度便是应用程序拥有的服务质量（QoS）。

尽管在多媒体系统出现前就有人研究过实时系统的设计问题，并且人们已经开发出许多成功的实时系统（例如，Kopetz和Verissimo[1993]），但是它们没有被集成为一个被广泛使用的多媒体操作系统和网络。航空电子设备、航空控制、制造过程控制和电话交换这些现存的

⊖　本章大量地引用了Ralf Herrtwich的指导论文[1995]，我们非常感谢他允许我们使用他的资料。

实时系统所执行任务的特征和多媒体应用程序的特征不同。前者通常处理相对小的数据量和相对少的硬时间限制，但是如果超过了时间限制，就会导致严重的甚至是灾难性的结果。在这种情况下，解决办法是充分估计所需要的资源并为其指定固定的调度计划，这样可以保证在最坏的情况下满足其需要。

为了满足多媒体和其他应用程序的需要而进行的有计划的资源分配和资源调度，这被称为服务质量管理。大多数当前的操作系统和网络并没有包含支持多媒体应用程序所需要的QoS管理设施。

在多媒体应用程序中，例如在像视频点播服务、商业会议应用和远程医疗服务这样的商业环境中，超出时间限制的错误会带来很严重的后果，这些多媒体应用与其他一些实时应用程序的需求有很大差别：

- 多媒体应用程序通常是高度分布的，并且在通用的分布式计算环境中使用。因此在用户工作站和服务器上，它们都和其他分布式应用程序竞争网络带宽和计算资源。
- 多媒体应用程序对资源的需求是动态的。在一个视频会议系统中，随着参与人数的增加和减少，其所需的带宽也会增加和减少。该应用在每个用户的工作站上使用的计算资源也会变化，这是因为，例如，显示所需的视频数据流的数目会发生变化。多媒体应用程序可能有波动的负载和间歇性的负载。例如，一个多媒体讲座可能包括处理器密集型的
608 仿真活动。
- 用户总希望平衡多媒体应用程序的资源开销和其他活动的资源开销。例如，为了在参加会议时，进行一个独立的音频会话程序，或者用户希望在他们参加会议时能同时运行一个字处理程序用户希望减少会议应用程序对视频带宽的需求。

通过根据变化的需求和用户优先级来动态地管理和分配可用的资源，QoS管理系统希望能满足所有这些需求。一个QoS管理系统必须管理用于获得、处理和传输多媒体数据流的所有计算和通信资源，特别是那些由多个应用程序共享的资源。

图15-1给出了一个典型的分布式多媒体系统，它能支持多种应用程序，例如桌面会议、广播数字电视和广播。其中，QoS管理的资源包括网络带宽、处理器周期以及内存。在使用视频服务器时，还包括磁盘带宽资源。我们将采用资源带宽这一术语来表示用于传输和处理多媒体数据的所有硬件资源可提供的能力（网络，中央处理器，磁盘子系统）。

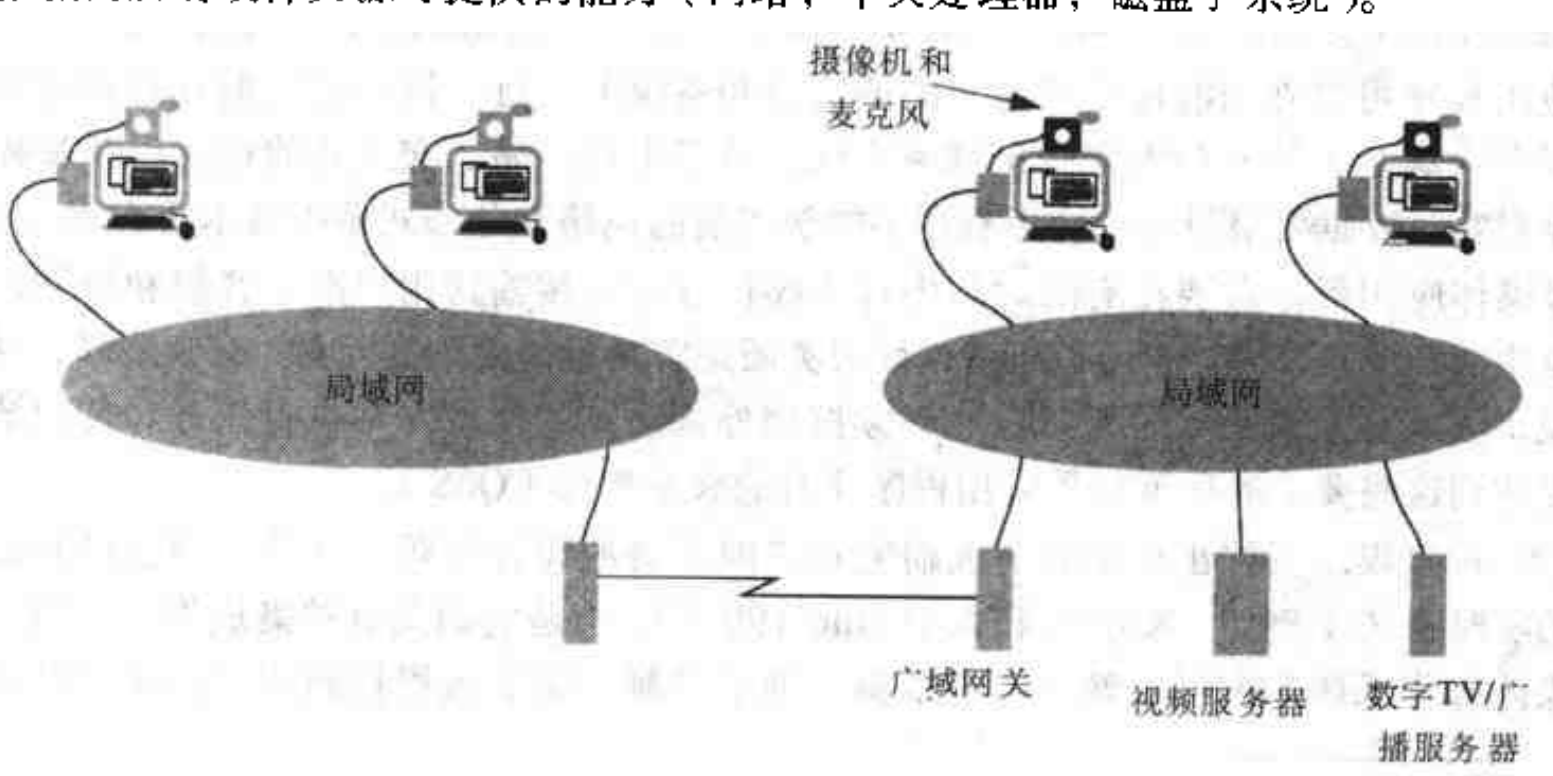


图15-1 一个分布式多媒体系统

在开放式的分布式系统中，系统可以在不预先安排的情况下，启动和使用多媒体应用程序。多个应用程序可以同时存在于一个网络中，甚至可能同时存在于一个工作站上。不管系统中的资源带宽和内存容量的总体质量如何，系统总是需要QoS管理。为了保证应用程序能在所需的时间限制内获得必要质量的资源，甚至是当其他应用程序竞争资源时，也是如此，系统必须有QoS管理。 609

一些多媒体应用程序已经应用在当今缺乏QoS管理但具有良好的计算能力和网络环境计算机系统上。它们包括：

- 基于Web的多媒体　这些应用程序最大限度地访问Web上公开的音频和视频数据流。它们曾经成功过，这是因为那时人们不需要或很少需要同步在不同的地点上的数据流。当前网络上的有限的带宽和变化的延迟以及当前操作系统对实时资源调度的不适应性都限制了它的执行性能。在音频和低质量的视频应用中，系统使用数据接收目的地的缓冲区来减缓带宽和延迟的变化，这样可以获得较平滑的视频显示，但是资源到达目的端的延迟可能会达到数秒。
- 网络电话和音频会议　这一应用程序对带宽的要求相对较低，特别是当其使用有效的压缩技术时。但是它的交互特性需要来回延迟比较小，并且这一点并不总是能达到。
- 视频点播服务　它们以数字的形式提供视频信息，它们从在线的存储系统中查询数据并将它们传输到终端用户的显示屏上。当有充分网络带宽可被使用时，这些应用可成功执行。它们也在目的端采用相当大的缓存。

高交互性的应用程序会遇到更多的问题。许多多媒体应用程序是合作性的（涉及多个用户），并且是同步的（需要紧密地协调用户的活动）。它们的应用面很广，例如：

- 一个简单的包括两个或多个用户的视频会议，每一个用户使用装备有数字摄像机、麦克风、声音输出设备和视频显示设备的工作站。有很多支持简单视频会议的应用软件（例如，CUSeeMe[Dorcey 1995, www.cuseeme.com]，NetMeeting[www.microsoft.com III]），但它的执行性能受到当今计算环境和网络环境的限制。 610
- 一个音乐排练程序使音乐家可以在不同的地点进行合练[Konstantas *et al.* 1997]。这是一个有特殊要求的多媒体应用程序，因为它的同步限制很严格。

这样的应用程序需要：

- 低延迟的通信　来回延迟<100ms，这样在用户之间的交互才会看起来是同步的。
- 同步的分布状态　如果一个用户将视频流停在给定的帧上，那么其他用户也应该看到这一点。
- 媒体同步　音乐合奏的所有参与者都应该在近似的同一时间（Konstantas等[1997]指出它的同步需要限制在50ms内）内听到其演奏。独立的声道和视频数据应该维持“音唇同步”，例如，当视频回放时，用户的评论也应与音乐同步。
- 外部同步　在会议系统和其他合作性的应用程序中，系统中可能会存在其他形式的活动数据，例如，计算机生成的动画，CAD数据，电子白板以及共享的文档。对这些数据的更新必须是分布式的，并且还必须近似地和基于时间的多媒体数据流同步。

这些应用程序只能在包含严格的QoS管理方案的系统上才能成功地运行。

匮乏区　许多当今的计算机系统提供了处理多媒体数据的一些能力，但是所提供的资源有很大的限制。特别是处理大量的音频和视频数据流时，许多系统受其能支持的流的数量和质量的限制。这种情况被描述为匮乏区[Anderson *et al.* 1990b]。当某一类的应用程序位于这一

区域时，为了提供所需的服务，系统需要小心地分配和调度它的资源（见图15-2）。在没有进入匮乏区时，系统所有的用于执行相关应用程序的资源并不充足。在20世纪80年代前，多媒体应用程序的处境便是如此。一旦应用程序离开了匮乏区，系统就可以有充分的资源来提供此服务，甚至在最坏的情况下和没有特殊的管理机制下也是如此。

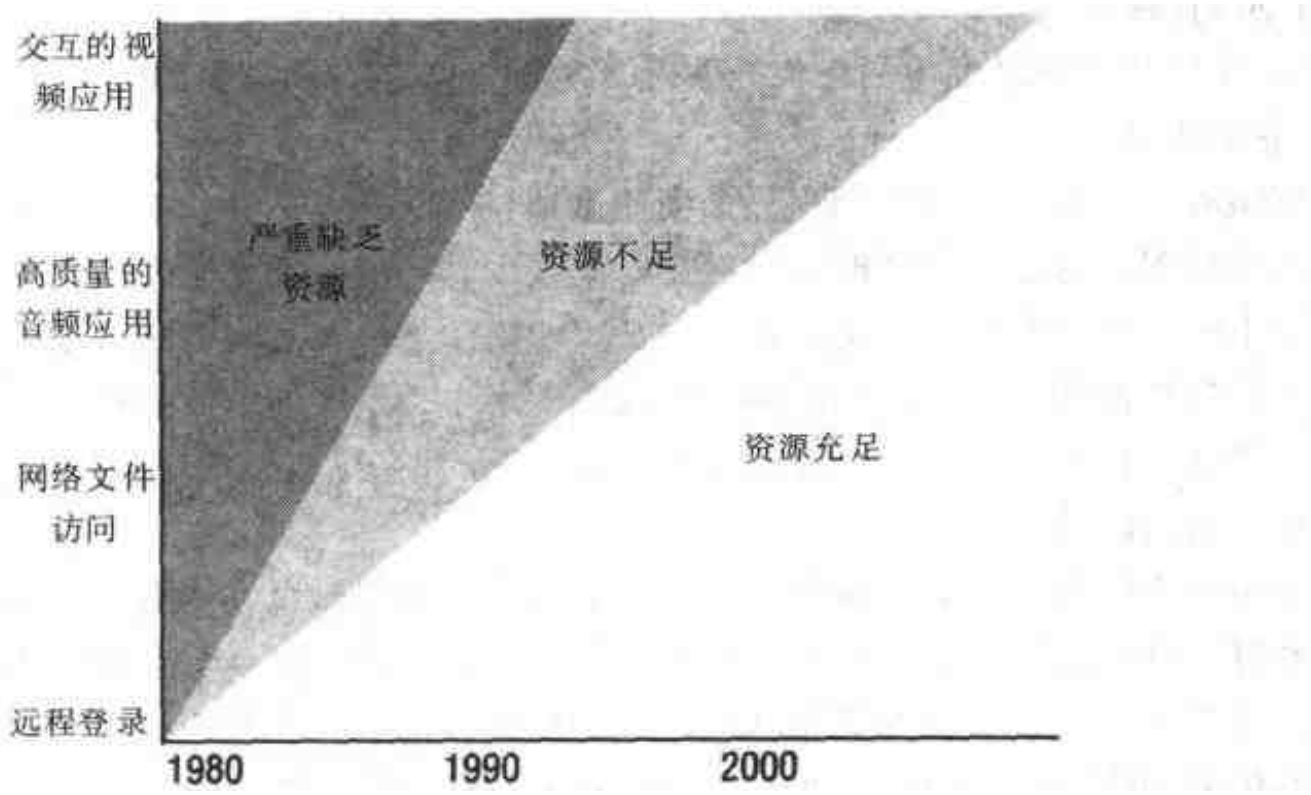


图15-2 计算和通信资源的匮乏区

系统性能的最新进展主要在于提高多媒体数据的质量，其中包括更高的帧传输率和更好的视频数据流的分辨率，或者是在像视频会议系统这样的应用中支持并发的多数据流。但是仍有一些需求很高的应用程序处于匮乏区内，其中包括虚拟现实和实时流处理（“特殊效果”）。

15.2节我们将回顾多媒体数据的特征，15.3节将介绍为了实现QoS管理而采取的匮乏资源的分配方法，15.4节讨论调度资源的方法，15.5节介绍在多媒体系统中优化数据流的方法。15.6节介绍Tiger视频文件系统，这是一个用于将存储的视频流并发地传输到大量客户的系统，该系统具有低开销并是可伸缩的。

15.2 多媒体数据的特征

我们已经提过视频和音频数据是连续的和基于时间的。我们怎样更精确地定义其特征呢？“连续性”一词表示的是从用户观点看的数据特征。实际上，连续的媒体是由连续的离散值组成的，后到达的值会替换先到达的值。例如，为了给出一个电视画面质量的动画，其图像阵列值每秒种要更新25次；为了传播电话质量的语音信息，其声音振幅值每秒要替换8000次。

因为音频和视频数据流中的实时数据元素定义了数据流的语义或“内容”，所以多媒体数据流被称为是基于时间（或等时）的。由于数据值被处理和记录的时间影响数据的正确性，因此，当支持多媒体应用程序的系统处理连续的数据时，它需要保持数据的时序。

多媒体数据流的数据量通常很大，因此，支持多媒体应用程序的系统必须比传统的系统有更大的移动数据的吞吐量。图15-3给出了一些经典的数据速率和帧/采样频率。我们注意到其中有些系统的资源带宽需求比较大。特别是在视频系统中，为了获得较好的质量，必须是这样的。例如，一个标准的TV视频数据流需要120Mbps的带宽，它超过了以太网能提供的

100Mbps的带宽。它对CPU处理能力的需求也随之扩大；一个用于标准TV视频数据流拷贝和数据传输的程序至少消耗一个400MHz CPU处理能力的10%。在高性能电视系统中，这些数据会更高，并且许多像视频会议这样的应用程序需要同时处理视频和音频信息。因此必须使用数据压缩技术，尽管压缩可能会增加处理的困难。

	数据速率（近似值）	采样或帧	
		大小	频率
电话交谈	64 Kbps	8 位	8000/秒
CD质量的声音	1.4 Mbps	16 位	44 000/秒
标准TV视频（未压缩）	120Mbps	最高640 × 480像素 × 16 位	24/秒
标准TV视频（用MPEG-1压缩）	1.5Mbps	可变的	24/秒
HDTV 视频（未压缩）	1000Mbps~3000Mbps	最高1920 × 1080像素 × 24 位	24~60/秒
HDTV视频（用MPEG-2压缩）	10Mbps~30 Mbps	可变的	24~60/秒

图15-3 典型多媒体数据流的特征

压缩可以以10到100的数据压缩比来减小对带宽的需求，但它不会影响连续数据的时序需求。为了设计出高效率、通用性的多媒体数据表示和压缩方法，人们进行了深入的研究，并定义了许多标准。这些工作的成果包括一些数据压缩格式，例如为图像数据设计的GIF、TIFF和JPEG标准以及为视频数据流设计的MPEG-1、MPEG-2和MPEG-4标准。在这里，我们不准备详细地介绍它们；其他的一些文献介绍了媒体类型、数据表示以及标准等内容，例如Buford[1994]以及Gibbs和Tsichritzis[1994]，并且Gibbs和Szentivanyi[multimedia index]的网页给出了当前多媒体标准的文档。

尽管使用压缩的视频和音频数据减少了对通信网络的带宽需求，但它增加了在源端和目的端对处理器资源的负载。系统也需要使用特殊的硬件来处理和发送视频和音频信息——为个人计算机设计的视频卡上包含视频和音频的编码/解码器。但是随着个人计算机和多处理器体系结构的计算能力的增加，系统可以用软件来执行编码和解码过滤功能。这种解决方法对特定应用的数据格式、特殊目的的应用逻辑以及同时处理多个媒体流的处理能力，提供了更好的支持，所以具有更好的灵活性。

使用MPEG视频格式的压缩方法是非对称的，包括一个复杂的压缩算法和一个相对简单的解压算法。这一点在桌面会议中是有用的，因为在桌面会议中，通常是由硬件来执行压缩，而由软件来执行对到达每个用户计算机的多个数据流的解压，这样可以不必考虑每个用户计算机上的解码器的个数，而运行会议的参与者数目可以动态地变化。

611 ~ 613

15.3 服务质量管理

当多媒体应用程序运行在个人计算机网络上时，它与运行着应用程序的工作站（处理器周期、主线周期、缓冲区容量）和网络（物理传输连接、开关、网关）竞争资源。工作站和网络可能必须同时支持多个多媒体程序和传统应用程序。在多媒体和传统应用程序间就有竞

争，在不同的多媒体应用程序之间甚至在单个应用程序的数据流之间都可能有竞争。

在多任务操作系统和共享网络中，物理资源都是可以被并发使用的。在多任务的操作系统中，中央处理器在每一时刻只处理一个任务（或进程），一个轮转或其他调度方法的调度程序负责在当前竞争处理器资源的任务中选出一个，并调度它到处理器上运行。

网络是被设计用来使不同来源的信息进行交流的，它允许多个虚拟通道存在于同一个物理通道上。以太网这一主要的局域网技术以最优的方式来管理共享的传输介质。当通道上是平静时，任何结点都可以使用这一通道。但是这样可能会发生信息包冲突，当发生冲突时，结点会等待随机的一段时间，然而重发包，这样可以防止冲突。当网络负载很重时，很容易发生包冲突，但是这一发送方案在这种情况发生时，不能提供关于带宽和延迟的任何保证。

这些资源分配方案的主要特点是：当对资源的需求增加时，它们将资源更稀疏地分配给每个竞争资源的任务。共享处理器周期和网络带宽的轮转和其他最优方法都不能满足多媒体应用程序的需要。显而易见，它们不能实时地处理和传输多媒体数据流。迟到的传输数据是没有价值的。为了实现实时传输，应用程序需要保证在需要的时候能得到必要的资源。

为了提供这一保障而进行的资源管理和分配被称为服务质量管理。图15-4显示了运行在两个个人计算机上一个简单的多媒体会议应用程序的底层组件，使用软件方式进行数据压缩和格式转换。白盒表示那些对资源的需求会影响应用程序的服务质量的软件组件。

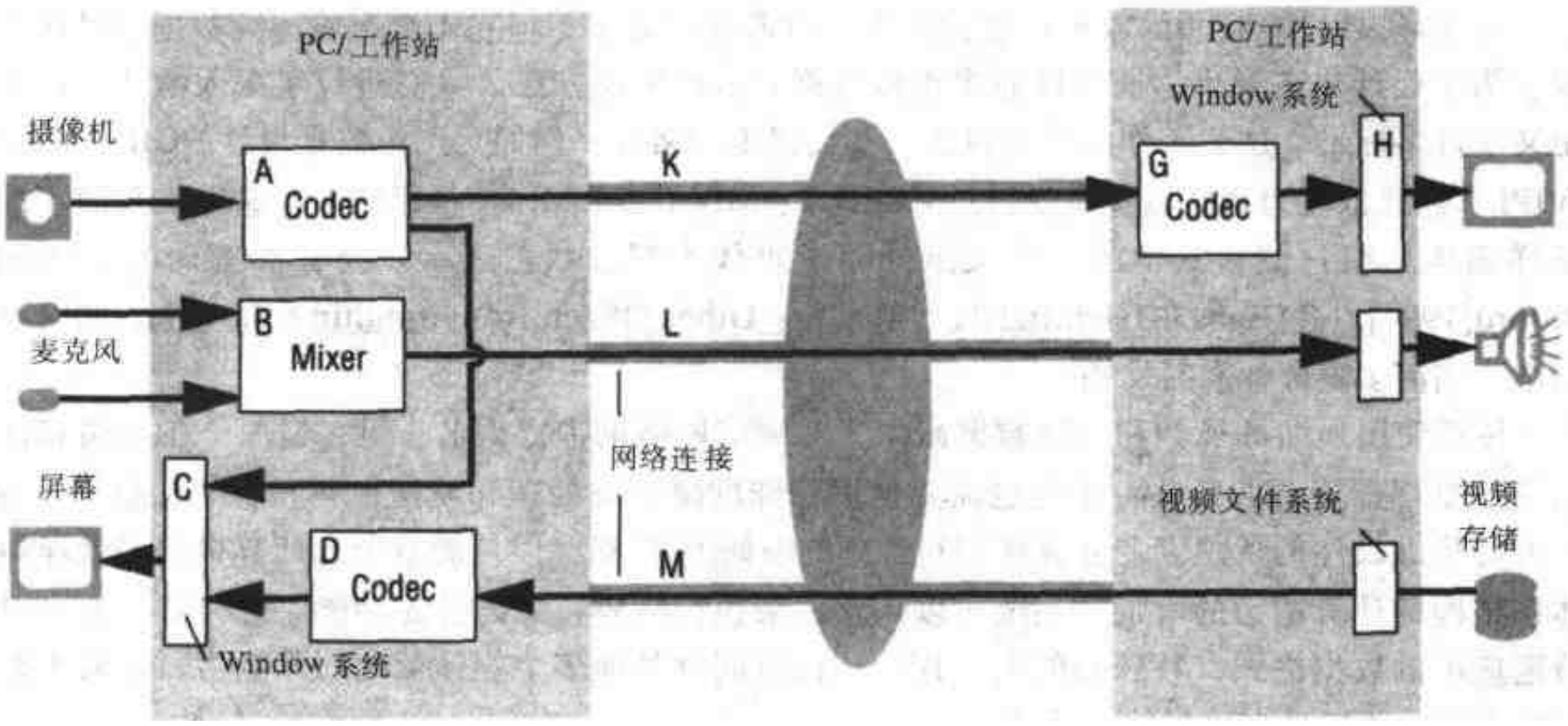


图15-4 多媒体应用程序典型的底层组件

这个图给出了多媒体软件常用的抽象体系结构，其中连续性的媒体数据流（视频帧，音频采样）被一系列进程处理，并通过进程间的连接在进程间传输。这些进程产生、传输和消耗连续的多媒体数据流。这些进程连接多媒体元素的源端和目标端，在这些端点上，多媒体数据被输出或者被消耗。这些连接可以由网络连接实现，当源和目标端的进程都位于同一计算机上时，这些连接也可以由内存内部传输实现。当多媒体数据元素及时地到达目标端时，

系统必须划分出充分的CPU时间、内存容量和网络带宽给处理这项任务的进程，同时，系统应调度这些进程，使它们能充分地使用资源以便能及时向下一个处理进程传输数据元素。

在图15-5中，我们列出了图15-4中主要软件组件和网络连接所需要的资源（图15-5中的字母对应于图15-4中的组件）。显然，系统中需要有一个软件组件负责分配和调度这些资源，这样才能保证能使用图中所示的资源。我们把这一软件组件称为服务质量管理。

组　件		带　宽	延　迟	丢失率	所需要的资源
摄像机	输出：	10帧/秒，原始视频数据 640×480×16位		零	-
A Codec	输入： 输出：	10帧/秒，原始视频数据 MPEG-1数据流	交互的	低	每100ms需CPU10ms；10MB RAM
B Mixer	输入： 输出：	2×44Kbps 音频数据 1×44Kbps 音频数据	交互的	很低	每100ms需CPU1ms；1MB RAM
H 窗口系统	输入： 输出：	可变 50帧/秒帧缓冲区	交互的	低	每100ms需CPU5ms 5MB RAM
K 网络连接	输入/输出：	MPEG-1数据流，大约1.5Mbps	交互的	低	1.5Mbps，低丢失率的数据流协议
L 网络连接	输入/输出：	44Kbps音频数据	交互的	很低	44Kbps，非常低的丢失率的数据流协议

图15-5　图15-4的应用程序组件的QoS规范

图15-6以流程图的形式说明了QoS管理的职责。在下面的两个小节中，我们将介绍QoS管理任务的两个主要的子任务：

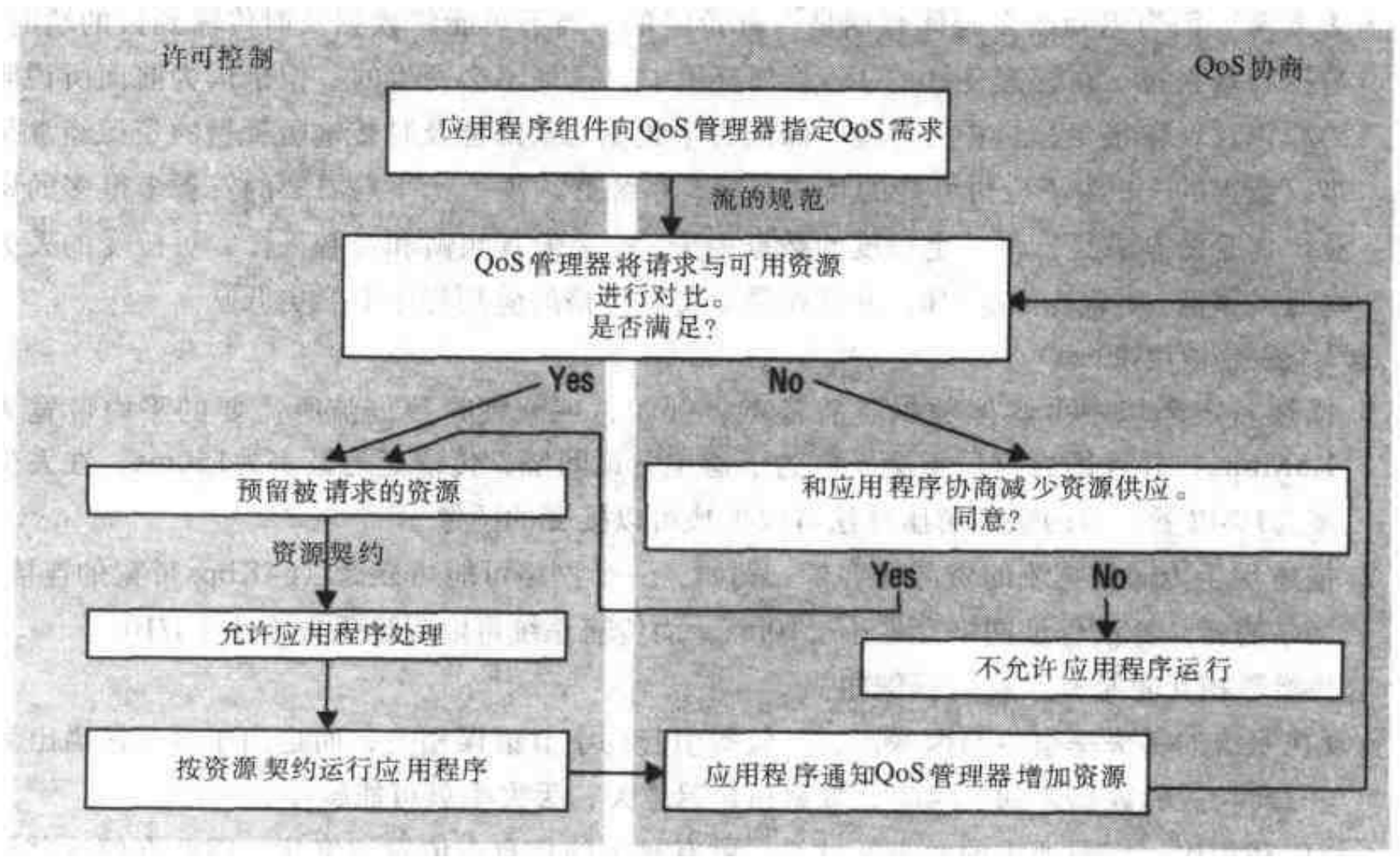


图15-6　QoS管理的任务

• 服务质量协商　应用程序向QoS管理提出自己的对资源的需求。QoS管理根据包含可用

资源和当前被使用资源信息的数据库来评估满足这些需求的可能性，然后它给应用程序一个肯定或否定的答复。如果其答复是否定的，这一应用程序可能会重新配置，以便减小对资源的需求，然后系统再重复以上过程。

- 许可控制　如果资源评估的结果是肯定的，被请求的资源就被预留了，同时应用程序获得一个资源契约，用于指定被预留使用的资源。这一契约包含一个时间限制。然后，应用程序就可以运行了。如果应用程序的资源需求发生了变化，它必须通知QoS管理。如果资源需求减小了，被释放的资源被加入到数据库中可用资源中，如果资源需求增加了，系统便需要进行新一轮的协商和许可控制。

在这一节的最后，我们将进一步描述执行这些于任务所使用的技术，当然，当一个应用程序正在运行时，它需要粒度适中的资源调度，以保证实时进程能及时接收到已分配的资源。15.4节将介绍这些技术。

15.3.1 服务质量协商

为了在应用程序和它底层的系统之间进行QoS协商，应用程序必须向QoS管理指定自己的QoS需求，它是通过传递一个参数集实现的。当处理和传输多媒体数据时，有3个参数非常重要，它们是：带宽、延迟和丢失率。

- 带宽　多媒体数据流或组件的带宽是数据流过的速度。
- 延迟　延迟是指单个数据元素从源端传输到目的端的时间。当系统中的数据量大小以及系统负载中的其他数值变化时，延迟也会发生变化。这种变化被称为抖动——抖动是延迟派生出来的。
- 丢失率　因为迟到的多媒体数据是没有价值的，当不可能将数据及时传输到目的端时，数据将被丢掉。在管理良好的QoS管理环境中，这是不会发生的，但是因为前面所说明的原因，现在很少有这样的环境。保证每个数据元素都能及时传输所耗费的资源经常是难以获得的——为了应付可能的传输高峰，系统必须预留比平均需要的资源多得多的资源。变通的方法是容忍一定程度的数据丢失——丢失视频帧和音频采样。可接受的丢失率通常很低——很少高于1%，并且在质量要求严格的应用程序中会更低。

这3个参数被用来：

1. 描述特定环境中多媒体数据流的特点。例如，一个视频数据流所需要的平均带宽为1.5Mbps，并且因为应用于会议，为了避免会话间隔，传输延迟最多为150ms。在丢失率为1%以下，目的端的解压算法可以生成可以接受的图像。
2. 描述用于传输数据流的资源的容量。例如，一个网络可能可以提供64Kbps带宽的连接，网络的排队算法保证网络延迟小于10ms，而传输系统可以保证丢失率小于$1/10^6$。

这些参数相互间是有联系的，例如：

- 现在系统的丢失率很少与因噪声、失效等引起的字节错误相关，而是由于缓冲区溢出和与时间相关的数据迟到。因此，带宽和延迟越大，丢失率就可能越小。
- 与负载相比，资源所占的总带宽越小，就有越多的信息在传输端聚集，因此存储这些信息的缓冲区就越大。缓冲区越大，就可能有更多的信息等待被服务，因此，系统也就有可能有更大的延迟。

614
~
617

为数据流设定QoS参数 QoS参数的值可以显式地给出（例如，图15-4中摄像输出流需要带宽50Mbps，延迟150ms，丢失率在10^3帧中少于1帧），也可以隐式地给出（例如，面对网络连接K，输入数据流的带宽等于对摄像输出流采用了MPEG-1压缩的数值）。

但是更常用的情况是我们需要指定一个值和一个允许变化的范围。这里我们将讨论一下对每个参数的需求：

带宽 大多数视频压缩技术根据原始视频数据的内容不同，生成的帧数据流的大小也不同。在MPEG中，平均压缩比为1:50到1:100之间，但是根据数据内容的不同，压缩比会动态的变化；例如，在内容变化很快时，需要的带宽会很高。因为以上原因，通常以最大值、平均值和最小值3种类型的值来表示QoS参数，选择哪种值依赖于当前使用哪种QoS管理制度。

与带宽说明相关的另一个问题是网络的数据爆发。假设有3条1Mbps的数据流。其中一条数据流每秒传输一个1M位的帧，第二条数据流是一个传输计算机生成的动画元素的异步数据流，平均带宽为1Mbps，第三条数据流每μs发送100位的声音采样信号。尽管这3条数据流都需要同样的带宽，它们的传输模式差别很大。

一种解决不规则数据爆发的方法是在传输率和数据帧大小之外定义爆发参数。这一爆发参数说明可能提前到达——也就是说，它们先于平常的速率到达时间到达了——的媒体元素的最大数目。Anderson[1993]使用的线性限制的到达处理（LBAP）模型将任意时间间隔t内的数据流最大数据量定义为$Rt+B$，其中R是传输速率，B为数据爆发的最大数目。这种模型的优点是它能很好地反映多媒体数据源的特点：从磁盘上读出的多媒体数据通常是成块的，而且从网络接受的数据通常是以小数据包序列的形式到达。在这种情况下，爆发参数提供了为避免丢失而需要的缓冲区空间大小。

延迟 多媒体中的一些时间性的需求来源于数据流本身：如果不能以数据流传输同样的速度在数据帧的到达点处理数据，等待处理的数据会越来越多，就会超出缓冲区的容量。为了避免这一点，数据帧滞留在缓冲区的时间不能高于$1/R$，其中R为数据流中帧的传输率，否则，缓冲区就可能发生积压。如果发生了积压，除了处理和传播时间之外，积压的数目和大小也会影响数据流端对端延迟的最大值。

另一种延迟需求来源于应用程序环境。在会议应用程序中，为了使参与者之间的交互看起来连续，数据流端对端的延迟应不超过150ms，这样用户在会话过程中不会感觉到有停顿，而在播放存储视频数据的系统中，为了保证像*Pause*和*Stop*这样的命令发出后能及时得到响应，最大的延迟应在500ms左右。 618

关系到多媒体数据传输延迟的第三种情况是抖动——传输两个相邻帧间隔时间的变化。尽管大多数多媒体设备都假设数据传输是没有抖动的，但软件（例如，处理视频帧的软件解码器）必须采取相应的方法来避免抖动。本质上，使用缓存可以解决抖动问题，但是抖动的时间范围必须有一个限制，这是因为端到端的整体延迟是受限于上面提到的种种考虑，所以媒体数据元素必须在固定的时间限制内到达。

丢失率 丢失率是最难指定的QoS参数。通常丢失率来源于对缓冲区溢出和延迟信息的概率统计。这种计算要么基于最坏情形的假设，要么基于标准的分配。这两种方法都不能很好地满足实际情况的需要。然而，系统必须使用丢失率说明来限定带宽和延迟参数：两个应用程序可能拥有同样的带宽和延迟特征；但一个应用程序丢失率为20%，而另一个应用程序丢失率为百万分之一时，它们的差别可能很大。

在带宽参数说明中，不仅仅是在一段时间内参数的数据总量很重要，在这段时间内每个时间间隔内传输的数据量也很重要，和带宽一样，系统需要考虑每个时间间隔内的数据丢失率。特别的，在无穷长时间间隔内的丢失率是没有用的，这是因为在一个短期内的丢失率可能会明显超过长期的丢失率。

流量调整 流量调整是用来描述使用输出缓冲来使数据元素流平滑这一方法的术语。多媒体数据流的带宽参数通常给出的是：发生在数据流传输时对实际传输模式的理想化近似。实际的传输模式越接近这一描述，系统就能越好地处理传输流量，特别是在系统使用为周期性请求设计的调度方法时，这一特点就会越显著。

带宽的LBAP模型要求对多媒体数据流的爆发进行管理。该模型通过在源端加入一个缓冲区和通过定义数据元素离开缓冲区的方法，管理任一数据流。漏桶的图像（图15-7(a)）形象化地说明了这种方法的特点：可以向这个桶中注水，直到它满了为止；因为在桶底有一个漏洞，水可以连续地流出。漏桶算法可以保证数据流的传输不超过R的速率。缓冲区B的容量被定为在没有丢失元素的情况下数据流爆发的最大值。数据元素停留在桶中的时间和B的大小有关系。

漏桶算法完全消除了数据爆发。只要带宽在任意时间间隔内都是有限制的，以上的消除就不是必须的。令牌桶算法（图15-7(b)）通过在数据流空闲时允许大一些的数据爆发短时间发生来做到这一点。这是漏桶算法的变种，其中发送数据的令牌以固定的速率R生成。只有当桶里至少有S个令牌时，S大小的数据才能被传输。然后，发送程序删除这S个令牌。令牌桶算法保证了在任意时间间隔t内数据的传输量不超过$Rt+B$。因此，它是LBAP模型的一个实现。

619

在令牌桶系统中，仅当数据流空闲了一段时间后，B大小的传输量高峰才会出现。为了避免这一数据爆发，可以在令牌桶后面放置一个简单的漏桶。为了使这种配置方案起作用，这个桶的流动速率F必须大大大于R。它的惟一目的是分解大的数据爆发。

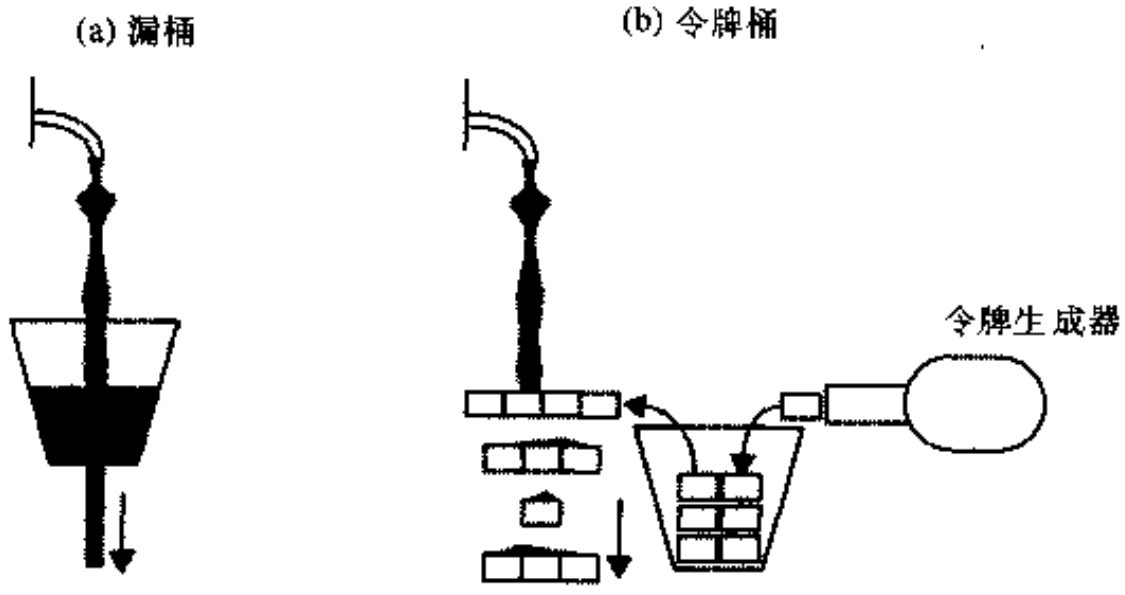


图15-7 流量调整算法

流的规范 QoS参数的集合通常被称为流的规范。现存的几个流规范的样本都比较相似。在因特网RFC 1363[Partridge 1992]中，流规范被定义为16位的数字值（图15-8），它以以下方式描述上面介绍的QoS参数：

- 最大传输单元和最大传输率决定了数据流所需要的最大带宽。
- 令牌桶大小和速率决定了数据的爆发量。
- 通过应用程序可以发现的最小延迟（因为我们希望避免对短延迟的过度优化）和其可以

接受的最大抖动来描述延迟特性。

• 通过在给定时间间隔内可接受的总丢失数和可接收的最大连续丢失数目来定义丢失特征。

还有许多表示每个参数组的方法。在SRP[Anderson *et al.* 1990a]中，通过一个最大预工作参数来给出数据流的爆发量，该参数定义了在任意时间点上提前到达的数据流信息的数量。Ferrari和Verma[1990]给出了一个最坏延迟量：如果系统不能保证在这一时间间隔内传输数据，那么对应用程序来说这一数据传输是没有用的。在RFC 1190的ST-II协议规范[Topolcic 1990]中，丢失率表示为每一个包被丢失的概率。

上面的例子给出了QoS参数值的多种表示。如果系统支持有限的应用程序和数据流，定义一个离散的QoS类集合就可能足够了：例如，电话质量和高保真声音，实况和回放视频等。所有系统组件必须显式地知道这些类的需求；当一些这样的通信类型混合时，系统也可以被配置。

620

协商过程 对分布式多媒体应用程序而言，一个数据流的组件可能位于多个结点上。在每个结点上有一个QoS管理器。直接的QoS协商办法是从源端到目的端一直跟随着数据流。源端组件通过向本地QoS管理器发送一个流规范来起动协议过程。这个QoS管理器可以检查数据库中记录的可用资源并决定所请求的QoS是否能满足。如果这一应用程序涉及到其他系统，流规范被传送到下一需要资源的结点。这一流规范传输过所有的结点，直到它最终到达目的端。然后，系统可以得出此QoS请求是否能满足的结论，并将该信息传输回源端。这种简单的协商方法可以满足多种目的，但它没有考虑到在不同结点上的并发QoS协商之间可能会发生冲突。为了彻底解决此问题，我们需要一个分布事务式的QoS协商过程。

应用程序很少是拥有固定的QoS需求。相对于返回一个关于QoS请求是否能被满足的布尔值这种方式，另一种方法可能更好，这种方法是由系统决定可以提供什么样的资源，并让应用程序来决定是否可接收。为了避免过度优化的QoS，或者为了在所需的服务质量明显不能达到的情况下放弃这一协商，通常应用程序会指定每个QoS参数的预期值和最坏值。一个应用程序可能会指定它需要1.5Mbps的带宽，但在1 Mbps带宽的时候它也能处理，只是延迟为200ms，但300ms是它可接收的最坏情况。因为在一个时间内只能优化一个参数，所以像HeiRAT[Vogt *et al.*1993]这样的系统希望用户只定义两个参数的值，并由系统来优化第三个参数。

	协议版本
带宽：	最大传输单元
	令牌桶速率
	令牌桶大小
	最大传输速率
延迟：	可被发现的最小延迟
	延迟变化的最大值
丢失率：	可以被发觉的丢失率
	可以被发觉的爆发丢失
	丢失间隔
	质量保证

图15-8 RFC 1363的流规范

如果一个数据流包含多个槽，系统将根据数据流分支协商路径。作为以上方案的直接扩
621 展，中间结点可以聚集目的端的QoS反馈并生成QoS参数的最坏情况值。这时，可用的带宽为各目的端可用带宽的最小值，延迟为各目的端延迟的最大值，丢失率为各目的端丢失率的最大值。像SRP、ST-II和RCAP[Banerjea and Mah 1991]这样的由发送端发起的协商协议使用了以上操作过程。

在目的端是异构的情况下，通常不适于在所有目的端上使用一个公共的最坏情况的QoS。相反，每一个目的端应接收最好的QoS可能值。这就要求接收端发起协商，而不是由发送方发起协商过程。RSVP[Zhang *et al.* 1993]属于另一类的QoS协商协议，其中由目的端连接数据流。源端将现存的数据流和它们的内在特征通知给各个目的端。目的端便可以连接到数据流经过的最近结点，并从那里获得数据。为了使它们能获得合适QoS的数据，它们可能需要像过滤（将在15.5节中介绍）这样的技术。

15.3.2 许可控制

许可控制管理对资源的访问，以避免资源过载，并防止资源接收不可能实现的请求。它涉及关掉那些与当前的QoS保证冲突的资源请求。

一个许可控制方案是基于整个系统容量和每个应用程序产生的负载这两方面的知识的。一个应用程序的带宽需求规范可能是应用程序需要的最大带宽、保证其运行的最小带宽，或者是它们之间的平均值。相应地，许可控制方案可以基于这些值之一进行资源分配。

如果所有的资源只由一个分配器控制，那么许可控制是直接的。如果资源分布在各个结点上，例如许多局域网环境，其可以使用一个集中式的访问控制，也可以使用一个分布式的许可控制算法，由它避免并发许可控制的冲突。工作站的总线仲裁算法属于这一类；然而，执行带宽分配的多媒体系统并不控制总线许可，因为总线带宽并不在匮乏区内。

带宽预留 保证多媒体数据流某一QoS级别的一个普通方法是预留一部分的资源带宽以便由它独占使用。为了在任一时刻实现数据流的需求，需要为它预留最大带宽。这是提供给应用程序有保障的QoS惟一可能的方法——至少需要没有发生灾难性的系统故障。在应用程序不能适应不同程度的QoS或者当其质量下降时程序就没有作用的环境下，程序可以使用这种方法。相应的例子包括一些医疗应用系统（在X光视频图像中，某症状图像可能正好在丢失的帧中）和视频记录系统（视频记录系统需要记录每一时刻的图像，丢掉的帧可能会导致缺陷）。

基于最大需求的预留方法可以是直接的：当控制一个带宽为B的网络时，仅当$\sum b_s <= B$时，带宽b_s的多媒体数据流s才能被允许访问。这样，一个16Mbps的令牌环网可能支持10个
622 1.5Mbps的数字视频流。

但是，容量计算并不总是和在网络中一样简单。为了以同一方法分配CPU带宽，系统需要知道每个应用程序进程的执行概况。然而，执行时间取决于所使用的处理器，并且不容易精确估计。虽然有一些执行时间的自动计算方法[Mok 1985，Kopetz *et al.* 1989]，但它们都没有被广泛应用。通常通过度量来决定执行时间，但这通常会产生很大的错误率并且可行性很有限。

对于像MPEG这样的典型的编码应用而言，应用程序实际使用的带宽可能比最大带宽小很多。预留最大带宽方法可能会导致资源带宽的浪费：尽管新的资源申请可以被满足，即，使用已被保留而未被已有程序实际使用的带宽，但是系统仍然拒绝其申请。

统计的多路技术 因为系统中可能存在潜在的未被利用的资源，这在超额预留资源的情况下经常发生。而一些保证技术可以提供使用这些资源的一些（通常很高）可能性，这些保证通常被称为统计保证或软保证，它与前面介绍的硬保证技术不同。因为不考虑最坏的情况，统计性保证技术可以提供更高的资源利用率。但是如果仅仅只依据最小或平均需求来分配资源，那么短期的负载高峰可能会导致服务质量的下降；应用程序必须能应付这样的服务质量降低。

统计的多路技术是基于这样一个假设：对大量数据流来说，虽然单个的数据流可能会发生变化，但这些数据流需要的总带宽相对稳定。它假设当一个数据流发送大量的数据时，就有可能有另一个数据流发送的数据量较小，这样总带宽需求保存平衡。当然这些数据流之间应该是没有联系的。

正像实验所显示的那样[Leland *et al.* 1993]，在典型环境下，多媒体数据流量并不符合这一假设。假设有大量的数据爆发，那么总流量仍然是爆发的。术语自相似被应用于这种现象，它表示总流量和组成它的单个流量具有相似性。

15.4 资源管理

为了向应用程序提供一定等级的QoS服务，系统不仅需要充分的资源（执行），它还需要在应用程序需要时有能力将这些资源提供给程序使用（调度）。

资源调度

系统需要根据进程的优先级来为其分配资源。资源调度器根据特定的标准来决定进程的优先级。在传统的分时系统中，CPU调度进程基于程序的响应时间以及公平原则来指定优先级：I/O量大的进程会获得高优先级，这样可以保证对用户做出快速响应，与CPU联系紧密的 623
任务获得低优先级，并且系统平等对待同一优先级的进程。

多媒体系统也可以使用这一标准，但是传输单个多媒体数据元素的时间限制改变了调度问题的特性。为了解决这一问题，系统可以使用下面介绍的实时调度算法。因为多媒体系统必须处理离散的和连续的媒体，因此在不引起离散媒体访问和其他交互应用程序饥饿的情况下，如何为实时性的数据流提供充分的服务，这是一个挑战。

调度算法必须管理（或协同）影响多媒体应用程序的所有资源。在通常的情况下，系统从磁盘上读取多媒体数据流，并将其通过网络传输到目的机器，在目的机器上，该数据流和其他来源的数据流同步合成起来，并最终显示。在这个例子中，系统需要的资源包括磁盘、网络、CPU以及内存和总线。

公平调度 如果有多个数据流竞争同一资源，系统必须考虑到公平性，防止不正常的数据流占用过多的带宽。保证公平性的一个简单方法是对同一优先级的数据流使用轮转调度方法。在Nagle[1987]中，这一方法是基于包的，在Demers等[1989]中，这种方法是基于二进制位的，因为包的大小和包到达时间会发生变化，所以后一种方法提供的公平性较好。这些方法被称为公平排队。

数据包实际上不能按位发送，但是在给定帧速率并且要求必须完全发送的情况下，系统可以计算每个包的时间。如果包传输是基于这一计算结果排序的，那么它可以获得接近于基于位的轮转传输方法所产生的传输结果，除非有个非常大的数据包需要传输，这是，它会阻

塞小数据包的传输，这就违背了基于位的调度方案。然而，任何包的延迟都不会高于最大包的传输时间。

所有的基本轮转算法都为每一个数据流分配同样的带宽。考虑到单个数据流的带宽，可以将基于位的方法进行扩展，一些数据流可以在每个周期传输更多的位信息。这种方法被称为基于权值的公平排队。

实时调度 人们已经开发出来一些实时调度算法来满足一些应用程序如航空工业过程控制的CPU调度需要。假设CPU资源并没有被过度分配（这是QoS管理器的任务），调度算法将CPU时间片以某种方式分配给多个进程，而这种方式必须使进程能及时地完成任务。

传统的实时调度算法十分适合规则的连续多媒体数据流模型。最早时间限制优先（EDF）调度算法几乎是这些方法的同义词。一个EDF调度器根据每个工作项的时间限制来决定下一个要处理的工作项：具有最早时间限制的工作项优先处理。在多媒体应用程序中，我们将到达进程的多媒体元素称为工作项。EDF调度策略被证明在基于时序标准分配单个资源方面是最优的：
624 如果系统中有一个调度满足了所有的时序需求，那么它就与EDF调度相似[Dertouzos 1974]。

EDF调度策略需要在每个信息（每个多媒体元素）上进行调度决策。如果它用于调度生存期更长的元素，会更有效。单一速率（RM）调度策略是一个适用于实时调度周期性进程的著名技术，它可以实现以上目标。系统根据数据流的工作速率指定其优先级：数据流的工作项速率越高，数据流的优先级越高。在多媒体程序使用的带宽低于69%时，RM调度策略会显示其优化性[Liu and Layland 1973]。使用这样的调度方案，剩余的带宽可以用于非实时应用程序。

为了应付爆发的实时通信量，基本的实时调度方法应该被调整为能识别有严格时间要求和无严格时间要求的连续媒体工作项。Govindan和Anderson[1991]介绍了时间限制/预工作调度方法。它允许在数据爆发时连续数据流中的消息可以提前到达，但是只根据其正常到达时间来对这一信息使用EDF调度。

15.5 流适应

当系统不能保证特定的QoS，或者当系统只能以某种概率保证QoS时，应用程序需要相应的调整自身执行，以便适应变化的QoS级别。对于连续媒体数据流而言，这种调整将转化为媒体表示质量的不同级别。

丢掉部分信息是最简单的调整。在音频数据流中，因为它的采样是相互独立的，因此这种方法比较容易实现，但是收听者可以发现这种丢失。在Motion JPEG编码技术中，因为它的视频帧是独立的，所以数据丢失还是可以容忍的。在MPEG等编码机制中，一个视频帧可能会依赖于数个相邻的帧，所以容忍数据丢失的能力比较弱：系统可能要花费一段时间来恢复，并且这种编码机制会放大错误。

如果系统中没有足够的带宽并且数据也没有被丢失，那么会发生延迟。对于非交互式的程序而言，可能是可以接收的，但是它可能会最终导致在源端或目的端的缓冲区溢出。对于会议系统和其他交互式的应用程序而言，增加的延迟是不可接收的，或者延迟只能持续一小段时间。如果一个数据流的播放时间延迟了，它的播放速率应该被加快，直到符合正确的播放时间表为止：当数据流被延迟时，数据帧在它可用时就应该立即被播出。

15.5.1 调整

如果在数据流的目的端执行流适应操作，系统中任一瓶颈的负载都没有被减少，并且系统仍然会过载。为了解决这一问题，系统需要在数据流经过瓶颈前就将数据流调整到可用带宽可以承受的范围，这被称为流调整。

当实时数据流被采样时，系统可以采用流调整。对于已存储的数据流来说，流调整取决于编码方法能以何种程度来压缩数据减少数据流量。如果数据流已经被压缩过，为了实现流调整，必须对它重新压缩，那么流调整会变得很麻烦。尽管所有的流调整方法都是相类似的：对给定信号重采样，但流调整算法是依赖于媒体的。对音频信息来说，可以通过减少音频采样的频率来实现这一重采样过程。也可以通过减少立体声传输中的一个声道来实现它。正如这个例子所表示的，不同的流调整方法在不同的粒度上工作。 625

下列的流调整算法适合于视频数据流：

- 时态调整　通过减少传输一个时间间隔的视频帧的数目，系统可以减少时间域中的视频数据流的分辨率。时态调整最适合于那些每个视频帧是自包含的并且对单个视频帧访问相对独立的视频数据流。它很难处理Delta压缩技术生成的数据流，这是因为不是所有的帧都可以轻易丢弃。因此，时态调整技术更适合于Motion JPEG数据流，而不太适合MPEG数据流。
- 空间调整　在视频数据流中减少每一帧的像素数。对空间调整技术而言，系统适合采用层次型管理方法，这样可以立即获得各种分辨率的压缩数据流。因此，视频数据可以在最终传输前不需要对每个图像再次编码，系统可以使用不同的分辨率来传输数据流。JPEG和MPEG-2支持不同的图像空间分辨率，因此它们适合于这种调整方法。
- 频度调整　改变应用于每个图像的压缩算法。这样会导致损失一部分图像质量，不过在通常的图像中，在可以察觉到图像质量降低前，可以大大提高数据的压缩率。
- 振幅调整　减少每个像素的颜色深度。这种调整方法实际上被H.261使用。
- 颜色空间调整　减少颜色空间的数目。实现颜色空间调整的一种方法是将彩色图像转变为黑白图像。

在需要时，还可以将这些调整方法组合起来。

执行调整的系统由一个在目的端的监视进程和一个在源端的调整进程组成。监视进程监视数据流中信息的到达时间。如果发生延迟，就说明系统中存在瓶颈。监视进程会向源端发送一个向下调整信息，源端就减少数据流的带宽。在一段时间以后，源端将数据流向上调整回原来的水平。如果瓶颈仍然存在，监视进程会再次检测到延迟，源端会再次调整数据流[Delgrossi *et al.* 1993]。流调整的问题包括：如何避免不必要的向上调整操作和如何避免系统进入抖动状态。

15.5.2 过滤

流调整在源端改变了数据流，但它并不适合于包含数个接收者的应用程序：当瓶颈发生在源端到其中一个接收者的路径上时，向源端发送一个向下调整信息会使所有的目的端接收到的图像质量下降，尽管其中可能有些接收者在处理原数据流时没有任何问题。

过滤技术可以为每个目的端提供可能达到的最好的QoS，它是通过在从源端到目的端的路

径上的相关结点上采用流调整技术（图15-9）来实现的。RSVP[Zhang *et al.* 1993]是支持过滤的QoS协商协议的一个例子。过滤需要一个数据流被分解为一个层次性的子流集合，其中每一个增加更高级别的质量。路径结点的容量决定了目的端接收到的子数据流数。其他的子数据流在靠近源的地方（甚至可能就在源端）就被过滤掉，这样可以避免传输后来被丢弃的数据。如果一个中间结点存在一条可以传输整个子数据流的向下传输的路径，那么子数据流在这个结点上就不会被过滤。

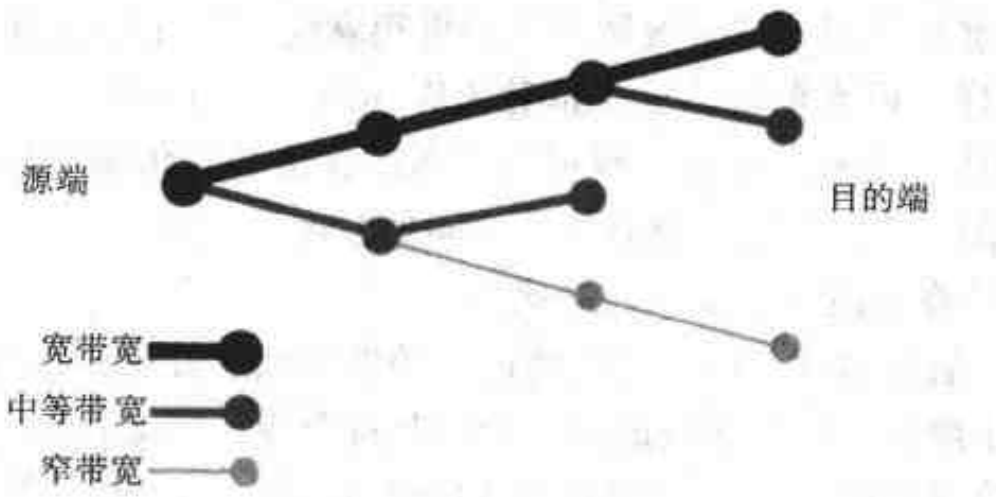


图15-9 过滤

15.6 实例研究：Tiger视频文件服务器

提供多个并发实时视频数据流的视频存储系统被看作为支持面向消费者的多媒体应用程序的一个重要的系统组件。人们已经开发了多个这种类型的程序原型，并且其中的一些已经形成了产品（见[Cheng 1998]）。其中一个进展最大的系统是Tiger视频文件服务器，它是在Microsoft研究院开发的[Bolosky *et al.* 1996]。

设计目标 这个系统的主要设计目标如下：

- *适用于大量用户的视频点播* 这一应用程序是向点播的用户提供电影的服务器。系统从大容量的数据电影库中选择电影。客户应在发送点播请求的数秒钟内就能获得电影图像的第一个帧，并且他还应该能随心所欲地执行暂停、回退和快进操作。尽管库中电影的数目很大，但是可能有一些电影是很受欢迎的，它们可能同时被多个客户不同步的访问，这就导致可能同时播放它们，但是播放的时间进度不同。
- *服务质量* 视频数据流的传输速率应保持稳定，其中客户端可用的缓冲区大小决定了系统能处理的最大的抖动，并且视频数据流还应保持低丢失率。

626
~
627

- *可伸缩性和分布性* 目的是以一种可伸缩的体系结构来设计系统，使它（通过增加计算机）可以同时支持10 000个客户。
- *低成本硬件* 这个系统是由低价的硬件构建的（“大众用的”PC机和普通的磁盘驱动器）。
- *容错性* 在单个服务器计算机或者是磁盘驱动器发送故障时，系统可以继续运行并且执行性能不会明显下降。

总而言之，这些需求需要一个存储和检索视频数据的基本方法和一个平衡多个相似服务器之间负载的有效调度算法。主要任务是将从磁盘上获取的高带宽的视频数据流传输到网络上、并且这应该是多个服务器共同承担的任务。

体系结构 在图15-10中显示了这一系统的硬件结构图。所有的部件都是可以买到的普通

产品。图中所示的cub计算机是包含同样数目的基本硬盘驱动器（通常是2到4个）的PC机。它们还安装了以太网和ATM网网卡。*controller*是另一台PC机，它并不处理多媒体数据，其职责只是处理客户请求和管理cub计算机的工作调度。

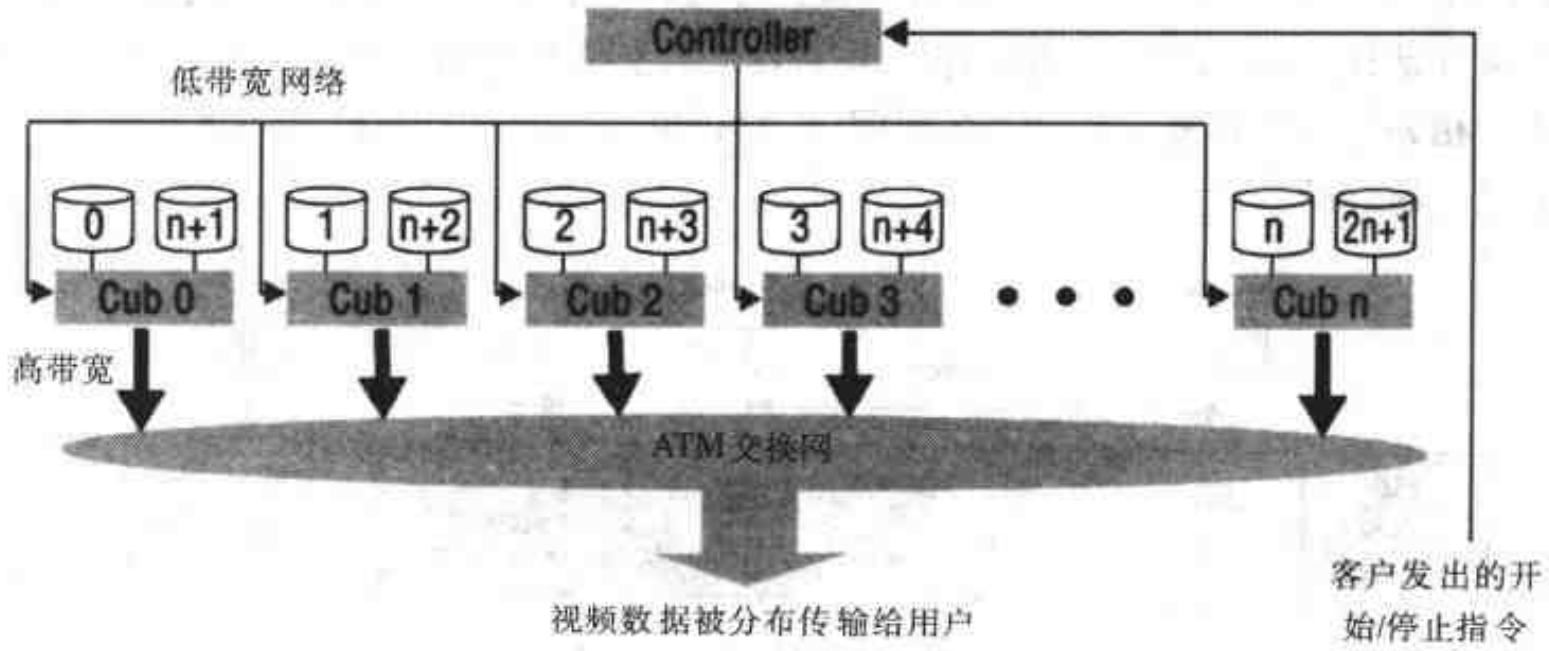


图15-10 Tiger视频文件系统的硬件配置

存储组织 其主要的设计问题是：如何在cub计算机的磁盘上分布存储视频数据，以实现计算机共享负载的目的。因为系统负载包括提供同一电影的多个数据流以及提供多个电影的多个数据流，因此使用单个磁盘来存储每个电影的方法是不能达到以上目标的。相反，电影被存储在跨越所有磁盘的条带上。它通过镜像方法来复制数据并提供容错机制，下面将介绍这一内容：

- *条带存储* 电影数据被划分为块（每一块数据的播放时间相等，通常是1s左右，大概占据0.5MB），并且组成一部电影的数据块集合（通常时间为两小时的电影大约包含7000个数据块）被存储在属于不同cub计算机的磁盘上，图15-10显示了其存储的顺序。一部电影的起始数据块可以存储在任意一台计算机上。当当前数据块存储在最大号的磁盘上后，下一数据块会被存储在第0号磁盘上，然后按顺序存储。 628
- *镜像* 镜像方案将每个数据块的数据备份划分为几个部分，称为二级备份，当一个cub计算机失效后，它保证本来有此cub计算机提供块数据的任务被分配到其他一些剩余的cub计算机上。二级备份的数目取决于散列因子d，它的值通常是在4到8之间。存储在磁盘i上的数据块的二级备份被分布存储在磁盘$i+1$到$i+d$上。假设系统中的cub计算机的数目多于d个，这些磁盘中没有一个和磁盘i属于同一个cub计算机。如果散列因子的值为8，那么不发生故障的任务可以使用cub计算机接近7/8的处理能力和磁盘带宽。其余1/8的资源应该足够用来处理出现故障的任务。

分布式调度 Tiger系统设计的核心是调度cub计算机的工作负载。系统的调度表实际上是一个槽的列表，每个槽表示播放一个数据块的电影这一工作——也就是说，它需要从相关的磁盘上读取数据块，并将其传输到ATM网络上。每一个可能的接收电影客户（被称为收看者）只能收到一个槽的信息，并且每个用户占据一个槽，表示收看者正在接收实时视频数据流。在调度表中，收看者的状态被表示为：

- 客户计算机的地址
- 被播放文件的标识

- 文件中收看者播放的位置（数据流中下一个要传送的数据块）
- 收看者播放的序列数（从中可以计算出下一个数据块的传送时间）
- 其他一些记录信息

图15-11给出了Tiger的调度。数据块播放时间T是指客户端上收看者播放一个数据块的时间，其通常是1s，并且假设对所有存储电影来说，都是相等的。因此，Tiger系统必须在每个数据流的相邻数据块传输中维持一个时间间隔T，通过在客户计算机上的缓冲机制，系统可以允许小范围的抖动。

图15-11 Tiger系统的调度

每一个cub计算机维持一个指针，指向每个磁盘的调度表。在数据块播放时，cub计算机必须处理它控制的磁盘所对应的槽，并且在当前块的播放时间内，完成数据的传送。cub计算机进行实时处理调度的步骤如下：

1. 将下一块从磁盘上读出，放在此cub计算机的缓冲区内。
2. 将此数据块打包并记入客户端计算机的地址，并将其传输到cub计算机的ATM网络控制器上。
3. 更新在调度表中的收看者的状态，这样可以给出下一个数据块和播放序列号。然后将更新的槽传送给下一个cub计算机处理。

假设这些活动所开销的最大时间为t（称为块服务时间）。正如图15-11中显示的那样，t实际上小于磁盘块的播放时间。t的值取决于磁盘带宽和网络带宽两者之间的较小值。（在cub计算机中其他资源足够完成对其磁盘访问的调度工作）。当一个cub计算机处理完当前磁盘块的播放这一调度任务，它就可以执行其他未调度的工作，直到下一个播放时间到来为止。实际上，磁盘并没有为数据块提供一个固定的延迟时间，为了适应这种不均匀的读盘时间，系统必须至少在数据块被打包和发送的前一个磁盘服务时间就启动读盘过程。

一个磁盘可以为T/t个数据流服务，并且T和t的值通常导致这一比率大于4。这一比率以及整个系统中的磁盘数目决定了Tiger系统可以服务的收看者的数目。例如，一个拥有5个cub计算机并且每个计算机上有3个磁盘的Tiger系统可以同时传送约70个视频数据流。

容错性 因为在Tiger系统中，电影文件的条带覆盖在所有的磁盘上，所以任意组件的错误（磁盘驱动器或cub计算机）都可能导致系统不能对所有的客户提供服务。Tiger系统为了应付这种情况所采取的方法是：当因为一个磁盘驱动器或一台cub计算机发生错误而引起一个主数据块不可用时，系统从它镜像的二级备份上读取数据。前面曾经提过，二级备份数据块比主数据块小，比例为散列因子d的值，一个数据块的二级备份被分布在数个不同cub的磁盘上。

当一个cub计算机或者磁盘失效后，一个临近的cub计算机会修改调度表并在其中加入数个镜像收看者状态，它表示存储这些二级备份的d个磁盘的工作任务。镜像收看者状态类似于普通的收看者状态，但是它拥有不同的磁盘号和时序需求。因为这一附加工作任务被d个磁盘和d

个计算机共享，如果在调度表中有一点空余空间，这项任务最好就利用这一空余，而不要打扰其他槽的工作任务。一个cub计算机的失效与它所属的磁盘的失效相似，其处理方式也相似。

网络支持 cub计算机只是将每部电影的数据块以及相关客户的地址信息传输到ATM网络上。能否顺序和实时地向客户计算机传输数据块，这依赖于ATM网络协议的QoS保证。客户需要大到能存储两个数据块的缓冲区，其中一个数据块是客户正在播放的，而另一个是正在从网络上接收的。当客户处理主数据块时，它只需要检查每个到达数据块的序列号，并将它们传送到处理程序上。当客户处理数据块的二级备份时，负责存储散列数据块的d个cub计算机负责将二级备份顺序地传输到网络上，客户负责收集这些二级备份并在缓冲区中将其组装起来。

629 ~ 630

其他功能 我们已经介绍过了Tiger服务器的有严格时间要求的活动。在它的设计中，所提供的服务还包括快进和回退。这些功能要求系统能将电影数据块中的一部分数据传输给客户，这样就由视频播放器提供一个对用户要求的视频反馈。cub计算机在非调度的时间内努力做到这一点。

其他一些功能还包括管理和分布调度表以及管理电影数据库，其中包括在磁盘上删除旧的电影、写入新的电影以及维持电影目录。

在Tiger系统的最初实现中，controller计算机处理调度表的管理和分布。因为这会导致单个结点的致命错误以及潜在的执行性能瓶颈，后来，调度表管理被设计为一个分布式的算法[Bolosky *et al.* 1997]。根据controller计算机发出的管理命令，cub计算机负责在非调度的时间内执行管理电影数据库的工作。

执行性能和可伸缩性 这一系统的原型系统在1994年被开发出来，使用了5台133MHz的奔腾PC，每台PC上配置有48MB的RAM和2GB的SCSI磁盘驱动器以及一个ATM网络控制器，运行的操作系统是Windows NT。使用模拟的客户负载来度量此系统的配置。在无错运行并且服务的客户数目达到68时，Tiger的数据传输性能相当不错——没有数据包被丢失或被延迟。当有一个cub计算机失效时（因此有3个磁盘失效），服务的数据丢失率仅为0.02%，这在设计允许的范围内。

另外一个度量是检查启动延迟，该延迟表示系统接收到客户请求到传输出第一个包之间的时间间隔。它与调度表中空闲槽的位置和数目密切相关。实现这项工作的算法开始会将客户请求放置在调度表中的一个槽中，这个槽是和所需电影的第一个磁盘块所在的磁盘相关的最近的一个空闲槽。这导致其启动延迟度量值在2s~12s范围以内。最近已经研究出一个新的槽分配算法[Douceur and Bolosky 1999]，它能改变调度表中被占据的槽的聚集性，使空闲的槽更均匀地分布在调度表中，这样可以减少平均启动延迟时间。

尽管最初试验系统的规模很小，但后来在14个cub计算机、56个磁盘以及使用了Bolosky等[1997]的分布式调度方案的系统上又进行了新的度量。当所有的cub计算机都工作时，系统可以同时发送602个2Mbps的的数据流，数据丢失率小于1/180 000。当有一个cub计算机失效时，数据丢失率小于1/40 000。这些结果可以证明在Tiger系统使用的cub计算机数达到1000时，它可以同时支持30 000 ~ 40 000个收看者。Microsoft NetShow Theater 服务器[www.microsoft.com]就包含了Tiger系统。

631

15.7 小结

多媒体应用程序需要新的系统机制以保证它们能处理大量的实时数据。这些机制中最重

要的一点是服务质量管理。它们必须合理地分配带宽和其他资源以保证系统可以满足应用程序的资源需求，同时它们必须调度对这些资源的使用过程，这样，多媒体程序可以获得比较好的执行效果。

服务质量管理负责处理应用程序提出的QoS请求，这一请求指定了多媒体数据流可接受的带宽、延迟和丢失率，服务质量管理同时也执行许可控制，它负责决定是否有足够的非预留资源来满足新请求的需要并在必要时与应用程序进行协商。一旦系统接受应用程序的QoS请求，其资源就被预留，同时系统发送一个保证信息给应用程序。

系统必须调度分配给应用程序的处理器处理能力和网络带宽以满足应用程序的需要。系统需要一个像最早时间限制优先和速率单调这样的实时处理调度算法来保证能及时地处理每个数据元素。

流量调整算法是用来缓冲实时数据的，它的目的是使数据流的数据元素之间的间隔时间更均匀。通过减少源端的输出带宽（调整）或减少中间结点的带宽（过滤），数据流可以被调整以适应较少的资源。

Tiger视频文件服务器系统是一个很好的可伸缩系统，它能为大范围的用户提供较好服务质量的数据流。它的资源调度方法比较特殊，它为这种类型的系统提供一个很好的例子。

练习

15.1 简要描述一个支持分布式音乐演奏的排练系统。请对这一系统使用的QoS需求以及硬件和软件配置提出建议。

15.2 当前因特网没有提供资源预留和服务质量管理服务。现存的基于因特网的音频和视频数据流应用程序如何获得可接受的服务质量？如果采用在多媒体应用程序中所采用的解决方法，有哪些局限性？

15.3 请说明分布式多媒体应用程序所可能需要的3种形式的同步（分布状态同步、媒体同步和外部同步）之间的区别。对于一个视频会议应用程序的例子，请对其实现同步的机制提出你的建议。

15.4 假设有由一个ATM网络连接多个桌面计算机并支持多个并发的多媒体应用程序的
632 系统，简要描述其QoS管理。为你描述的QoS管理定义API接口，同时给出其操作功能以及它们的参数和可能返回的结果。

15.5 为了给出处理多媒体数据的软件组件的资源需求，我们需要估计其处理负载。如何在合理的开销下获得这种信息？

15.6 当有大量的用户同时访问同一部电影时，Tiger系统如何处理？

15.7 Tiger系统的调度表包含了大量的结构化数据，并且被频繁地更新，但是每一个cub计算机需要获得它当前处理部分在调度表中的最新信息。请设计一个为cub计算机分发调度表的机制。

15.8 当Tiger系统要对一个失效的磁盘或cub计算机操作时，系统使用其数据块的二级备份来代替主数据块。二级备份数据块比主数据块小n倍（n是散列因子），数据块的大小改变了，
633 系统如何适应？

第16章　分布式共享内存

16.1　简介
16.2　设计问题和实现问题
16.3　顺序一致性和Ivy
16.4　释放一致性和Munin
16.5　其他一致性模型
16.6　小结

本章将介绍分布式共享内存（DSM），它是在不共享物理内存的不同计算机进程之间共享数据的一个抽象。DSM的目的是允许系统使用一个共享内存的编程模型，这一模型比基于消息的模型拥有更多的优点。例如，使用DSM，程序员无需对数据项进行编码。

实现DSM的一个核心问题是：在系统中包含大量计算机时，如何获得良好的性能。对DSM的访问可能会涉及到网络的通信。处理对同一数据项或相邻数据项的竞争可能会导致大量的通信。通信量和DSM的一致性模型紧密相关，当一个进程读取DSM数据时，一致性模型决定在多个写入数据版本中选择返回哪一个值。

本章将讨论DSM中像一致性模型这样的设计问题以及实现问题，例如，在对一个数据副本进行了写操作后，同一数据的其他副本哪些将失效，哪些将被更新。接着本章还详细讨论了失效协议。最后还描述了释放一致性——一个相对较弱的一致性模型，在许多情况下，达到这种一致性就足够了，同时它也较容易被实现。

16.1　简介

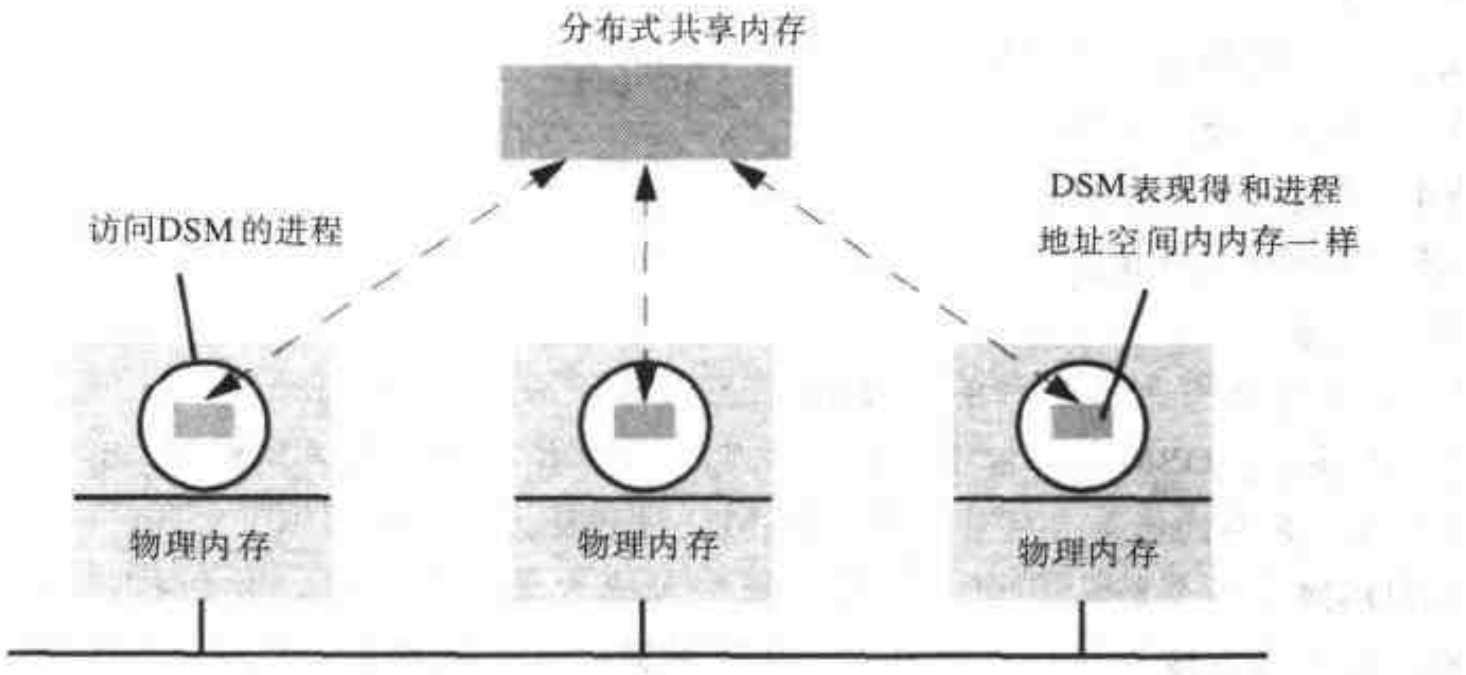
分布式共享内存（DSM）是在不共享物理内存的计算机之间实现共享数据的一个抽象。进程可以使用像访问自己地址空间内的普通内存一样的读写操作来访问DSM。然而，有一个在底层运行的系统来保证其透明性，这一透明性使得进程可以观察到其他不同计算机上进程所进行的数据更新。看上去，进程都访问单个共享内存，但实际上物理内存是分布式的（见图16-1）。

DSM的最关键一点是，它使得程序员不必关心数据通信，而使用传统方式时，就不可避免地要涉及到消息传递。并行应用程序或任意分布式应用程序以及群件应用程序都可以使用DSM这一工具，它可以使这些程序直接访问共享数据。通常DSM不太适合于客户－服务器系统，因为客户通常将服务器拥有的数据作为抽象数据并且通过发请求来访问这些数据（基于模块性和安全性的原因）。然而，服务器可以提供客户共享的DSM。例如，系统可以共享内存映射的文件，并且因此系统可以维持一定程度的一致性，这是DSM的一种形式。（在MULTICS操作系统［Organick 1972］中就引入了映射文件。）

在分布式系统中，消息传递是不可避免的：如果没有物理共享内存，那么DSM的运行时支持不得不在计算机之间传递数据更新。DSM系统管理复制的数据：每个计算机都拥有最近访问的数据项的本地副本，原始数据项存储在DSM中，这样可以加快访问速度。DSM的实现

问题和第14章讨论的问题相关，同时它也和第8章讨论的缓存共享文件问题相关。

Apollo域文件系统［Leach *et al.* 1983］是一个DSM的实现例子，不同工作站上的进程通过将共享文件映射到各自的地址空间内来同时访问这些文件。这个例子说明了分布式共享内存可以是持久性的。也就是说，它的生存期比访问它的任意进程或进程组的生存期长，并且不同的进程组可以长期地共享它。



636

图16-1 分布式共享内存抽象

随着共享内存的多处理器结构（参见6.3节）的发展，DSM的重要性也在不断提高。当前进行的许多研究是关于适用于这些多处理器结构上的并行计算的算法。在硬件体系结构层，研究进展包括缓存策略和处理器－内存快速互连方法，它的目标是在获得较快的内存访问和较大的吞吐量的同时，增加处理器的数目[Dubois *et al.* 1988]。在处理器和内存模块通过普通总线连接的系统中，因为处理器需要竞争总线的缘故，故处理器数目限制在10~20之间，否则这一系统的执行性能将会明显下降。共享内存的处理器通常是4个一组，它们通过一个电路板上的总线共享内存。在非一致内存访问（NUMA）结构的主板上，系统可以支持多达64个处理器。它是一种层次性的结构，其中系统使用高性能的开关或高一级的总线将多个拥有4个处理器的主板连接起来。但是其中处理器访问本主板上内存的延迟时间比访问其他主板上内存的延迟时间少——这就是这一体系结构名称的来源。

在分布式内存的多处理器结构中，处理器不共享内存，但是它们通过高速网络连接起来。和通用的分布式系统一样，这种体系结构的系统中，处理器和计算机的数目可能很大，可能比共享内存的多处理器结构的64个处理器的上限数大很多。DSM和多处理器结构所面临的一个核心问题是：共享内存算法和其相关软件的知识能否直接应用于更大规模的分布式共享内存系统。

16.1.1 消息传递机制和DSM的比较

作为一种通信机制，DSM和消息传递相似，但和基于请求－应答的通信方式不同，这是因为它的应用程序是并行处理的，使用的是异步通信。我们将从以下方面对应用于编程的DSM和消息传递方法进行比较：

- 编程模型　使用消息传递模型时，当一个进程向另一个进程发送数据时，发送进程必须对变量进行编码，而接收进程必须对变量进行解码。相反，使用共享内存的进程可以直

接共享变量，这样它就不需要进行编码和解码——甚至对指向共享数据的指针变量也是如此，因此不需要单独的通信操作。大多数DSM的实现允许系统可以以访问非共享变量的方式来对存储在DSM中的变量进行命名和访问。而在另一方面，使用消息传递的好处是：通过进程的私有地址区，使通信进程的数据受到保护。而这些进程在使用DSM的共享数据时，可能会因为一个进程不适当地改变了共享数据而导致其他进程崩溃。而且，在异构的计算机系统之间进行通信时，通信系统可以根据数据在不同系统之间的表示方式采用不同的编码方式；然而共享内存机制无法实现这一点，例如，不同整数表示方式的计算机之间如何共享内存呢?

通信模型使用消息传递原语来实现进程之间的同步，使用的是第11章介绍的像锁服务器实现这类技术。在DSM中，进程的同步是通过通常共享内存程序使用的同步机制实现的，其中包括锁和信号量（在分布式内存环境中，它们可能需要不同的实现方法）。第6章在介绍线程编程时简单讨论了这些同步对象。 637

最后，因为DSM可以是持久的，通过DSM通信的进程的生命期可能并不重叠。一个进程可以在合适的内存地点上留下它的数据，而另一个进程可以在它运行时读取该数据。相反，通过消息传递机制通信的进程必须同时运行。

- *效率* 一些试验证明：一些为DSM开发的并行程序可以完成在同一硬件上——系统使用的计算机数量比较小（大约10个左右）——使用消息传递机制的应用程序可以完成的功能[Carter *et al.* 1991]。然而，这一结果不适用于所有的系统。运行在DSM上程序的执行性能取决于多个因素，特别是数据共享的方式这一因素（例如一个数据项是否能被多个进程更新），在下面我们将对其进行讨论。

这两种类型的编程的开销明显不同。在消息传递机制中，远程数据访问是显式的，并且程序员可以了解某一操作是进程内的还是会引起通信开销的。而在使用DSM时，一个读或写操作可能会涉及到底层动态支持的通信，也可能不涉及。是否涉及到通信取决于若干因素，包括像访问的数据是否以前被属于不同计算机的进程访问过这样的因素。

DSM是否比消息传递机制更适合于特定的应用程序，这一问题没有确定的答案。DSM是一个很有希望的工具，它的最终应用情况取决于其实现效率。

16.1.2 DSM的实现方法

通过使用特殊的硬件组合、传统的分页虚拟内存或者中间件，我们可以实现分布式共享内存：

- *硬件* 基于NUMA体系结构（例如，Dash[Lenoski *et al.* 1992] 和PLUS[Bisiani and Ravishankar 1990]）的共享内存多处理器体系结构依赖于特殊的硬件为多个进程提供一致共享内存。它们通过与远程内存和缓存模块通信来处理LOAD和STORE指令，这些远程内存和缓存存储有所需的数据。这种通信建立在一种与网络类似的高速连接之上。Dash多处理器的原型系统拥有用NUMA体系结构连接方式连接的64个结点。
- *分页虚拟内存* 许多系统将DSM实现为一个虚拟内存区，这一虚拟内存区在每个参与的进程内占据同样的地址范围。这些系统包括：Ivy[Li and Hudak 1989]、Munin[Carter *et al.* 1991]、Mirage[Fleisch and Popek 1989]、Clouds[Dasgupta *et al.* 1991]（见www.cdk3.net/oss）、Choices[Sane *et al.* 1990]、COOL[Lea *et al.* 1993]和Mether[Minnich

and Farber 1989]。这种类型的实现适用于具有公共数据表示和分页结构的同构计算机。

638

- 中间件 —些像Orca[Bal *et al.* 1990]的语言和一些像Linda[Carriero and Gelernter 1989]以及其后的JavaSpaces[java.sun.com-VI]和Tspaces[Wyckoff *et al.* 1998]的系统可以以平台独立的方式支持DSM，不需要额外的硬件和分页机制支持。在这种类型的实现中，通过在客户和服务器上的用户级支持层之间的通信来实现共享内存机制。当进程访问DSM中的数据时，它对这一支持层发出调用。不同计算机上的这一层程序访问本地数据，并且在必要时通过相互间的通信来维持数据一致性。

本章主要讨论在普通计算机上使用软件来实现DSM。即使是在使用硬件支持的DSM系统中，系统也需要使用高层软件技术的支持来减少组件之间的通信量。

基于分页的实现方法的优点在于不在DSM上强加特别的结构，DSM表现为一系列的字节。在原则上，它可以使共享内存多处理器程序运行在没有共享内存的计算机系统上。像Mach和Chorus这样的微内核系统提供了对DSM的支持（以及对其他内存抽象的支持——第18章将介绍Mach的虚拟内存设施）。现在，基于分页的DSM系统主要是在用户级实现的，这样系统可以提供更多的灵活性。这种类型的实现使用内核支持来处理用户级的页失配。UNIX和一些版本的Windows系统可以提供这一功能。64位地址空间的微处理器使基于分页的DSM的应用范围更广，它放松了对地址空间的限制[Bartoli *et al.* 1993]。

```
#include "world.h"
struct shared { int a,b; };

Program Writer:
    main()
    {
        struct shared *p;
        methersetup();                          /*初始化Mether运行时 系统*/
        p = (struct shared *)METHERBASE;
                                    /*覆盖在METHER段上的结构*/
        p->a = p->b = 0;                        /*将域的初值设为0*/
        while(TRUE){                            /*连续更新结构中的域*/
            p ->a = p ->a + 1;
            p ->b = p ->b -1;
        }
    }

Program Reader:
    main()
    {
        struct shared *p;
        methersetup();
        p = (struct shared *)METHERBASE;
        while(TRUE){                            /*每一秒钟读一次域的值*/
            printf("a = %d, b = %d\n", p ->a, p ->b);
            sleep(1);
        }
    }
```

图16-2 Mether系统程序

图16-2给出的例子包含两个C语言程序：*Reader*和*Writer*，它们通过Mether系统[Minnich and Farber 1989]支持的分页式DSM通信。*Writer*程序更新在Mether DSM段开始处（从地址*METHERBASE*处开始）一个记录中两个域的值，并且*Reader*程序周期性地从这两个域中读出数据值并将它打印出来。

这两个程序都不包含特殊的操作；它们被编译成机器指令，这些指令可以访问每个公共的虚拟内存区（从*METHERBASE*处开始）。Mether系统运行在普通的Sun工作站和网络上。

使用中间件实现DSM的方法和使用特殊硬件及分页的方法完全不同，它并不使用现存的共享内存代码。它能使我们开发出共享对象的高层抽象，而不是共享内存地址位置。

16.2 设计问题和实现问题

本节将讨论关于DSM系统主要特征的设计问题和实现问题，包括DSM中数据的结构问题；在应用层一致访问DSM的同步模型；DSM的一致性模型，一致性模型管理不同计算机访问数据的一致性；在计算机之间交换所写信息的更新选项；在DSM实现中的共享粒度以及颠簸问题。

16.2.1 结构

第14章讨论了像日记和文件这样的对象的复制系统。这些系统允许客户程序以操纵单一对象副本的方式来操纵对象，但实际上它们访问的是多个不同的物理对象副本。系统对对象副本所允许的差异的程度做出保证。

DSM系统是上述这样的一个数据复制系统。每一个应用程序进程都拥有一些抽象的对象集合，但在这种情形下，“集合”有点像内存。也就是说，它们可以被不同的访问方式访问。不同的访问DSM的方式的区别在于：它们被看成何种“对象”以及如何访问对象。我们将介绍3种不同的访问方法，它们分别将DSM看作是由连续的字节、语言级对象或不变数据项组成的。

639
~
640

面向字节的 像访问通常的虚拟内存一样访问这种类型的DSM——DSM是一组连续的字节。Mether系统就采用了这种访问方法。其他一些DSM系统也采用了这一访问方法，包括Ivy系统，我们将在16.3节介绍它。这种方式允许应用程序（以及语言实现）在共享内存上实现所需要的任意数据结构。这些共享对象直接是可寻址的内存位置（实际上，这些共享位置可以是多个字节的字，而不是单个的字节）。对这些对象的操作只有两种：*read*（或LOAD）操作和*write*（或STORE）操作。如果*x*和*y*是两个内存地址位置，下面是使用这两种操作的例子：

R(*x*)*a*——从*x*位置读出值*a*的*read*操作。

W(*x*)*b*——将值*b*写入*x*位置的*write*操作。

一个执行序列是：*W*(*x*)1，*R*(*x*)2 。该进程在*x*位置写入值1，它又从这一位置读出值2。这是因为可能有其他进程同时在该地址写入值2。

面向对象的 这类共享内存由一系列语言级的对象如栈、字典等组成，这些对象比简单的*read*/*write*变量具有更高级别的语义。应用程序只能通过调用这些对象才能改变共享内存的值，它不能通过直接访问对象成员变量的方式来改变内存的值。这种管理内存的方式优点在于可以利用对象的语义来加强一致性。Orca系统将DSM看作共享对象的集合，同时它可以自动将给定的对象操作进行串行化处理。

不变数据 这类系统将DSM看作一组不变数据项的集合，进程可以读这些数据项并可添加和删除数据项。这类系统包括Agora系统[Bisiani and Forin 1988]和更有名的Linda系统以及由它派生的TSpaces和JavaSpaces系统。

Linda这类系统向程序员提供了元组的集合，其被称为元组空间。一个元组包含一个或多个有类型的数据域，例如，<“fred”，1958>、<“sid”，1964>和<4，9.8，“Yes”>。在同一元组空间内可能会存在由元组联合组成的元组。进程通过访问同一元组空间来共享数据：它们通过使用*take*操作来向元组空间内加入元组，它们通过*read*或*take*操作来读出或获取元组。*write*操作可以在不影响元组空间内已存在元组的前提下加入一个元组。*read*操作可以在不影响元组空间内容的前提下返回一个元组值。*take*操作也返回一个元组值，但它同时从元组空间内删除此元组。

641

当从元组空间内读出或获取出一个元组时，进程提供一个元组规约，系统从元组空间内返回一个符合这一规范的元组，这种规范是一种联合寻址。为了使进程间的活动能同步，*read*操作和*write*操作会被阻塞到在元组空间内找到一个符合规范的元组为止。一个元组规范包括元组的域的数目和在特定域中所需的类型或值。例如，*take*(<String，integer>)操作可能会获取<“fred”，1958>或<“sid”，1964>；*take*(<String，1958>)操作可能只会获取以上两元组中的<“fred”，1958>。

在Linda系统中，系统不允许进程直接访问元组空间中的元组，进程为了修改元组，必须用一个新的元组替换原有的元组。例如，假设在元组空间内，有多个进程共享一个计数器。元组<“counter”，64>表示计数器的值（为64）。为了在元组空间*myTS*中增加计数器的值，进程必须执行以下形式的代码：

```
<s, count>:= myTS.take (< "counter", integer>);
myTS.write (< "counter", count +1 >);
```

因为*take*操作从元组空间内获取元组，所以，读者应该检查系统中是否会发生竞争。

16.2.2 同步模型

许多应用程序会在共享内存存储的值上加上某些限制。正像为共享内存的多处理器系统（或者是共享数据的并发程序，例如操作系统内核和多线程服务器）编写的程序一样，基于DSM的应用程序也采用这一机制。例如，如果*a*和*b*是存储在DSM中的两个变量，其中的限制可以是*a*永远等于*b*。如果有两个或更多的进程同时执行以下代码：

```
a:= a +1;
b:= b +1;
```

那么可能发生不一致。假设*a*和*b*的初始值为0，进程1已经将*a*的值设为1。在它将*b*的值设为1之前，进程2将*a*的值设为2并将*b*的值设为1。这就违背了以上限制。其解决方法是将这一代码段设置为临界区：这样可以保证在某一时刻只有一个进程可以执行它。

为了使用DSM，系统需要提供一个分布式的同步机制，它应包括与锁或信号量相似的构造。甚至当DSM是由一组对象组成时，实现对象时也必须考虑到同步机制。通过使用消息传递，系统可以实现同步机制（见第11章，其中描述了分布式的锁服务器）。在共享内存的多处理器系统中使用的像*testAndSet*这样的特殊指令也可以应用于基于分页的DSM，但这些操作的效率很低。采用应用程序级同步机制的DSM可以减少更新传递量，因此DSM系统将同步机制

看作是它的一个集成的组件。

16.2.3 一致性模型

DSM通过将共享内存的内容缓存在不同的计算机上来复制这些共享内容，正如我们在第14章中所描述的那样，像DSM这样的系统会产生一致性问题，按第14章的术语，每个进程拥有一个本地副本管理器，它保存存储在缓存中的对象副本。在大多数实现中，为了提高效率，系统从本地副本中读取数据，但是对数据的更新必须告知其他副本管理器。

系统通过组合中间件（在每个进程中的DSM运行层）和内核的功能实现本地副本管理器。通常中间件完成大多数的DSM处理。甚至在基于分页的DSM实现中，内核通常只提供基本的页映射、页失配处理和通信机制，而中间件负责实现页共享机制。如果DSM段是持久的，那么会有一个或多个存储服务器（例如，文件服务器）扮演副本管理器的角色。 642

除了缓存之外，DSM实现可以将更新放入缓冲区，然后通过一次传送多个更新操作来减少通信开销。第14章介绍的gossip体系结构也采用了类似的方法来减少通信开销。

一个内存一致性模型[Mosberger 1993]说明了DSM系统所采用的一致性保证，该模型假设进程会访问每一个对象的副本，并且有多个进程可能对对象进行更新，该模型能保证进程读取数据的一致性。请注意：这种一致性和在前面应用程序同步中所讨论的高级别的、应用程序特定的一致性有所不同。

Cheriton[1985]描述了不同形式的DSM可以接受的不同程度的不一致性。例如，系统可以使用DSM来存储网络中计算机的负载信息，这样客户可以选择最小负载的计算机来运行应用程序。因为这些信息在相当短的时间片内本身就是不精确的，因此，系统不需要在所有时刻都保持在所有计算机上此信息的一致性。

然而大多数应用程序需要更强的一致性限制。系统必须给程序员提供一个一致性模型，由模型规定内存预期的表现形式。在更详细的介绍内存的一致性需求之前，让我们先来看一个例子，它将对我们理解一致性模型有所帮助。

假设一个应用程序中有两个进程（图16-3），它们访问两个变量：a和b，它们的初始值都为0。进程2按顺序增加a和b的值。进程1按顺序将b和a的值分别读入到局部变量br和ar中。这里没有应用程序级的同步。直觉上，进程1可以读到的是下列值的组合：ar=0，br=0；ar=1，br=0；ar=1，br=1。读到哪一个值的组合取决于进程1在进程2执行的哪一点上读a和b的值（隐含在语句ar=a，br=b中）。换句话说，$ar \geq br$总能满足，进程1总能打印“OK”。然而DSM的实现系统可能不按照进程1的副本管理器更新数据的顺序来更新a和b的值，所以在这种情况下，ar=0，br=1这样的值的组合也可能发生。

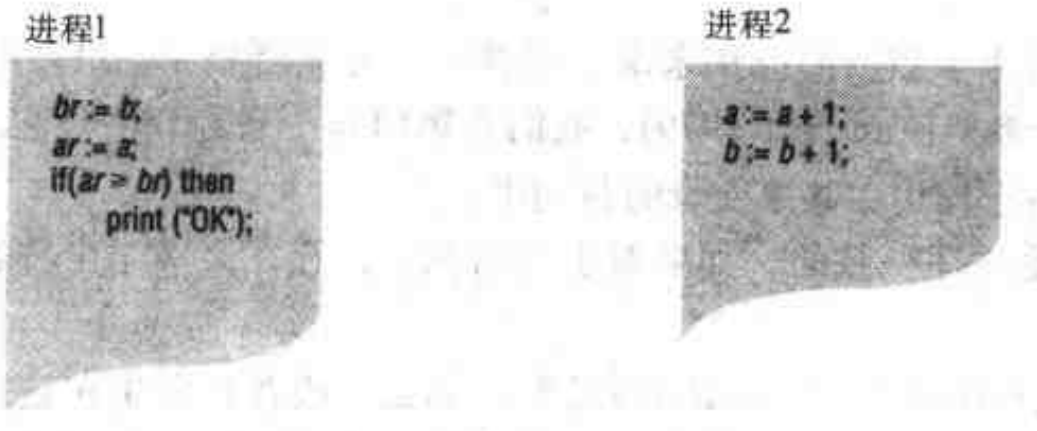


图16-3 访问共享变量的两个进程

643

在上面的DSM实现中，两个更新的顺序被颠倒了，用户对这个例子的直接反应是DSM实现可能是不正确的。如果进程1和进程2同时运行在一个单处理器的机器上，我们可以认为内存子系统出错了。然而，在分布式环境中，这可以是一个比我们预计的一致性模型更弱一些的一致性模型的正确表现，虽然该一致性模型比较弱，但它很有用并且效率比较高。

Mosberger[1993]列出了用于共享内存的多处理器系统和软件DSM系统的一系列的一致性模型。在DSM中实现的主要是顺序一致性模型和基于弱一致性的一些模型。

描述一个特定的内存一致性模型的主要问题是当进程对内存的某一地址进行读操作时，可能对同一地址进行过多次写操作，那么采用哪一个值返回给读操作？在最弱的一致性模型中，返回的是在读操作前任意一个写操作的值。如果副本管理器不确定地更新其数据，系统就只能达到这种一致性。这种一致性太弱了，以至于其没有实际用处。

而在最强的一致性模型中，所有更新的值在更新的同时对所有进程都有效：读操作返回的是在它读数据时的最新的更新值。这一定义存在两方面的问题。首先，读操作和写操作都需要持续一段时间，它不是一个简单的时间点，因此，“最新的”这一术语的意思并不明确。每一种类型的访问都被定义为在一个时间点上发生，但是它们需要花费一定的时间（例如，在消息传递之后完成）。其次，第11章说明了在分布式系统中时钟同步的局限性。因此，很难精确地断定一个事件是在另一个事件前发生。

不管怎样，人们还是给出并研究这个一致性模型。读者可能已经了解它了：在第14章中，它被称为线性化能力。在DSM的文献中，线性化能力通常被称为原子一致性。在这里，我们将再次介绍在第14章中的线性化能力的定义。

一个复制的共享对象服务被认为是可线性化的，如果对于任何执行，存在某一个全体客户操作的序列，满足以下两个准则：

L1：操作的交叉执行序列符合对象的（单个）正确副本所遵循的规范。

L2：操作之间的交叉执行序列和实际运行中的次序一致。

这一定义具有普遍性，它能应用于任意包含共享复制对象的系统。因为我们将把它应用于共享内存，所以我们对它进行详细说明。举一个简单的例子，假设一个共享内存的组织结构是一组可读可写的变量集。所有的操作都是读或写操作，在这里我们使用16.2.1节中曾经使用过的表示方法：$R(x)a$表示从变量x中读出值a；$W(x)b$表示向变量x写入值b。我们可以用变量（共享对象）的形式解释第一个准则：

L1'：如果交叉执行序列中有一个$R(x)a$操作，那么，或者在发生这一操作之前的最后一个写操作是$W(x)\ a$，或者在此之间没有发生过写操作并且x的初始值为a。

644

这一准则隐含了一个前提条件：变量只能被写操作改变。线性化能力的第二个准则L2则保持不变。

顺序一致性 对大多数实际应用来说，线性化能力太严格了。最强的应用于DSM的内存一致性模型是顺序一致性[Lamport 1979]，我们在第14章中曾经介绍过它。这里，我们将采用第14章的定义，并将它应用在共享变量的特例中。

一个DSM系统是顺序一致的，如果对于任何执行，存在某一个全体客户操作的序列，满足以下两个准则：

SC1：当交叉执行序列中有一个$R(x)a$操作，那么，或者在发生在这一操作之前的最后一个写操作是$W(x)\ a$，或者在此之间没有发生过写操作并且x的初始值为a。

SC2：每个客户执行程序的顺序和交叉执行中的操作被执行的顺序一致。

条件SC1和L1'完全相同。条件SC2注重的是操作在程序中的顺序，而不是时序，这一点使顺序一致性的实现成为可能。

这些准则可以用另一种说法来描述：进程读写操作序列可以对应到在一个虚拟内存映像上的一个虚拟的交叉执行顺序。该交叉执行顺序保持了每个操作在程序中的顺序，而且，该顺序中每个读操作读取的都是最后被写入的值。

在实际执行中，内存操作可能会重叠，并且在不同的进程中更新的顺序会不同，但是它可能并不违反以上定义所要求的约束。请注意，为了满足顺序一致性的条件，要考虑在整个DSM上的操作——而不仅仅是单个位置上的操作。

前述的$ar=0$，$br=1$的组合不可能发生在顺序一致性的系统中，这是因为进程1读取值的顺序与进程2的程序顺序相冲突。图16-4给出了按顺序一致性执行的操作执行序列。另外，尽管这个例子显示的是读和写操作实际的交叉执行顺序，但一致性定义规定了操作的执行应该像严格的交叉执行一样。

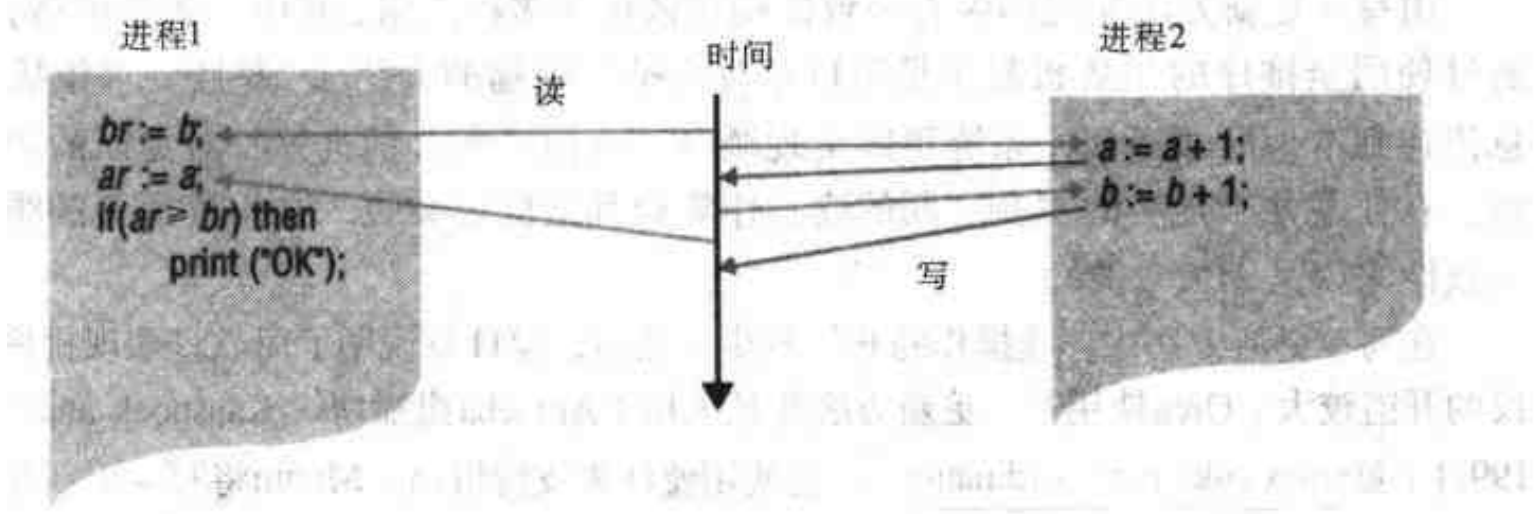


图16-4 顺序一致性下的交叉执行顺序

使用一个管理所有共享数据的服务器，并且使所有的进程通过向服务器发送请求来执行读和写操作，这样可以实现顺序一致的DSM。这种体系结构的DSM实现效率很低，下面将介绍实际使用的实现顺序一致性的方法。不过，顺序一致性一直是一个实现开销比较高的一致性模型。

连贯性 针对顺序一致性模型的高开销，人们设计出了一个弱一些的、具有良好特性的一致性模型。连贯性是一种弱一些的一致性。按照连贯性要求，每个进程就同一地点上的写操作的执行顺序达成一致即可，不同地点上的写操作的执行顺序没必要达成一致。我们可以将连贯性看作在每个位置上实现的顺序一致性。通过使用实现顺序一致性的协议并将该协议应用在每个复制数据单元上——例如每一页，我们可以实现连贯的DSM。连贯DSM的优点在于：因为协议是独立应用在每个页上，所以对不同的页的访问是相互独立的，一个访问操作不会延迟对其他页的访问操作。

弱一致性 Dubois等[1988]设计了一个弱一致性模型，该模型在保留顺序一致性效果的同时，试图避免顺序一致性在多处理器上的开销。为了放宽内存一致性，这一模型利用了同步操作所隐含的知识，这样对程序员来说，系统好像实现了顺序一致性（最少，在某些特定的环境下是如此，这超出了本书的介绍范围）。例如，如果程序员使用锁机制来实现一个临界区，那么，DSM系统可以假设没有其他进程可以访问加上互斥锁的数据项。因此，在进程离开这

一临界区之前，DSM系统传播这些数据项的更新信息是冗余的。尽管这一数据项在这段时间内存储的数据是“不一致的”，但是因为在这些时间内没有其他进程可以访问它；因此，这些执行看起来都是顺序一致的。Adve和Hill[1990]推广了弱一致性概念，称为弱顺序性：“（一个DSM系统）相对于一个同步模型是弱顺序的，当且仅当它对所有遵守同步模型的软件都表现为顺序一致”。释放一致性是弱一致性的一种改进模型，16.4节将介绍释放一致性。

16.2.4 更新选项

一个进程向其他进程传递更新数据有两种方法：写－更新和写－失效。它们可以应用到各种DSM一致性模型（包括顺序一致性模型）上。下面简要介绍这两种更新选项：

645
~
646

• 写－更新　进程进行的更新将改变本地数据并会组播到其他具有此数据项拷贝的副本管理器上，副本管理器在收到更新消息后会立即修改本地进程所读的数据（图16-5）。进程可以读数据项的本地副本，它不需要再进行通信。除了允许多个读数据进程之外，还允许多个进程在同一时刻写同一数据项；这就是多个读进程/多个写进程共享。

用写－更新方法实现的内存一致性模型依赖于多种因素，其中主要是组播排序属性。通过使用全排序的组播机制（见第11章对全排序组播的介绍），并且要求组播在更新消息传递到本地后才返回，系统可以实现顺序一致性。所有的进程都对更新的顺序达成一致。在任意两个连续的更新之间的读操作集合都是良定义的，这些读操作的顺序对顺序一致性来说是无关紧要的。

在写－更新方法中，读操作的开销较少。然而，第11章说明了用软件实现排序的组播协议的开销较大。Orca使用写－更新方法并且采用了Amoeba组播协议[Kaashoek and Tanenbaum 1991]（见www.cdk3.net/coordination），它使用硬件来支持组播。Munin将写－更新作为一个选项。在PLUS多处理器体系结构中，使用特殊硬件支持的系统可以使用写－更新协议。

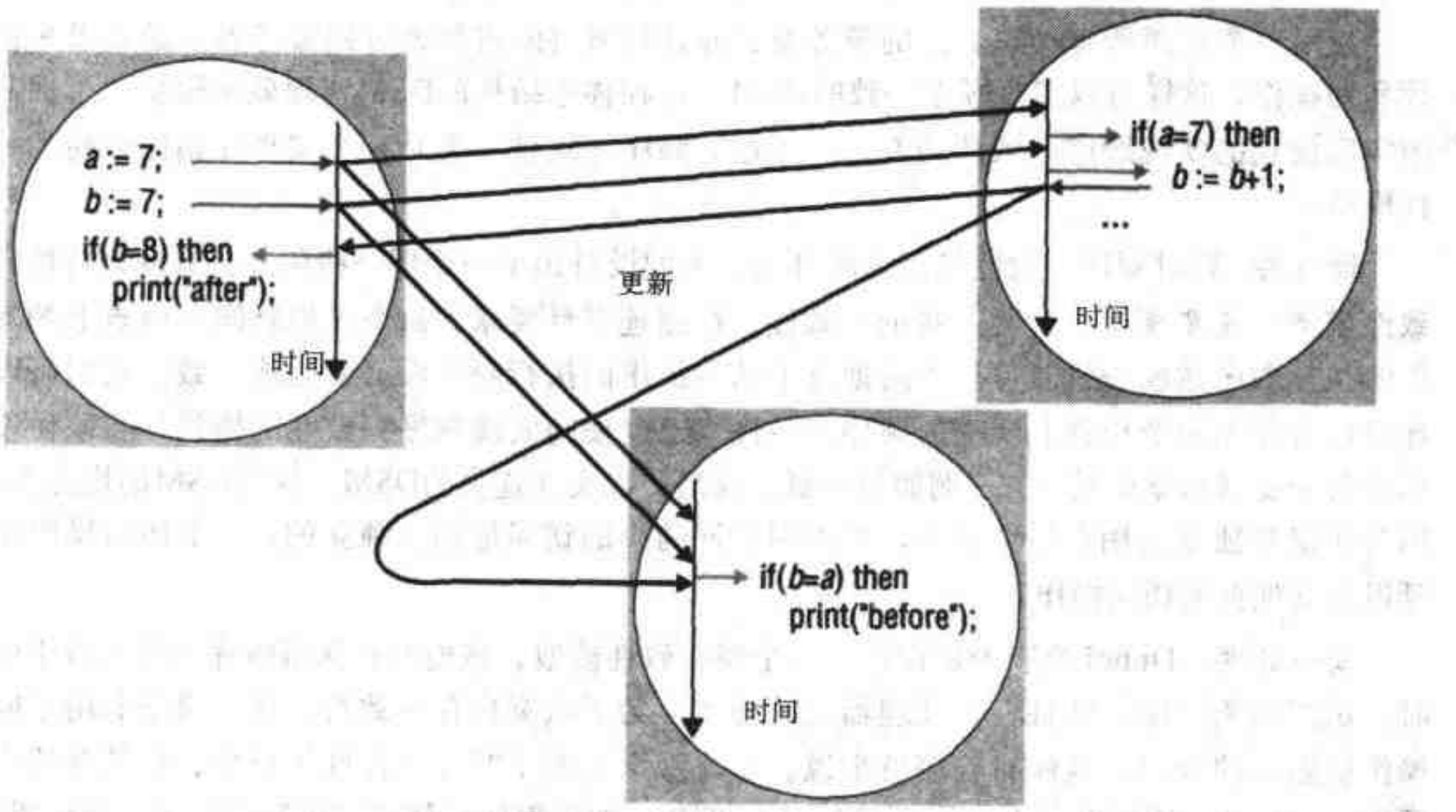


图16-5　使用写－更新的DSM

- 写 - 失效　它通常用于多个读进程/单个写进程的共享环境中。在任意时刻，通常有一个或多个进程以只读的方式来访问一个数据项，或者这一数据项由惟一一个进程进行读写。当前被只读方式访问的数据项可以有多个数据副本。当一个进程试图写这一数据时，系统首先向这一数据的所有副本组播一个失效消息，并且在写操作执行之前得到收到失效消息的确认；这时，系统阻止其他进程读这一数据项的过时数据（也就是说，这一数据项不是最新的）。如果有一个写进程正在写数据，其他试图读这一数据项的进程都会被阻塞。最后，写进程交出控制权，其他进程在更新的数据被送到后就可以访问此数据了。 647
该方案是在先来先服务的基础上处理数据访问的。Lamport[1979]给出的证据说明这种方案可以获得顺序一致性。16.4节中，我们可以看到在释放一致性的情况中，失效操作可以被延迟。

在这种失效方案中，仅当数据被读时，更新信息才传递到被读的数据副本上，并且在这种更新通信前可能会进行多个更新操作。与这些优点相对的缺点是：在写操作发生前，要花费对只读副本的失效开销。在所描述的多个读进程/单个写进程方式中，这种开销可能很大。但是，如果读/写的频率很高，那么允许多个读进程并行访问所得到的并行性可以弥补这一开销。当读/写频率很小时，系统可能更适合采用单个读进程/单个写进程方案，即，在某一时刻，最多只允许一个进程进行只读访问。

16.2.5 粒度

与DSM结构相关的一个问题是共享的粒度。在概念上，所有的进程共享DSM的所有内容。然而，共享DSM的程序执行时，实际上只有在特定的时间内特定的一部分数据被共享。如果DSM实现中进程访问和更新数据总是涉及到整个DSM的内容，那么就会产生浪费。在DSM实现中应采用什么样粒度的共享单元？也就是说，当一个进程对DSM进行写操作后，为了保持其他地点的一致性，DSM运行时应发送什么样的数据？

在这里，我们主要针对基于分页的实现，尽管粒度问题在其他实现中也同样存在（见练习16.11）。在基于分页的DSM中，硬件能有效地进行以页为单位的地址变换——主要是在页表中更换新的页面指针（可参见Bacon[1998]对分页的描述）。页面的大小通常可以达到8KB，所以在更新发生时，网络需要传输相当大数量的数据。在默认情况下，不管是整页更新，还是只更新页中的一个字节，系统都要进行整页的传输。

使用更小的页面尺寸——512字节或1KB——并不会使整个系统的执行性能得到明显改善。首先，当进程更新大量连续数据时，传输一个大一些的页比传输数个小一些的页效率高，这是因为每个网络包有固定的软件开销。其次，如果在DSM实现中使用小页面，系统就必须管理更多的数据单元。

使这一问题变得更复杂的是，当页面很大时，进程间很容易产生竞争页的情况，这是因为随着页的增大，访问落在同一页内的数据的可能性也将增大。例如，有两个进程，一个进程只访问数据项*A*，而另一个进程只访问数据项*B*，它们落在同一页内（图16-6）。具体来说，假设一个进程读数据*A*，另一个写数据*B*。在应用程序层，这是没有冲突的。然而，因为在默认情况下，DSM并不知道这一页的哪一个位置被改变了，所以系统必须在进程之间传输整个页。这种现象被称为*错误共享*：有两个或多个进程共享包括多个部分的页面，但实际上每个部分只被一个进程访问。在写 - 失效协议中，错误共享可能会导致不必要的失效。在写 - 更 648

新协议中，当有多个写进程错误地共享数据项时，可能会导致它们对一个老版本数据项的重复更新。

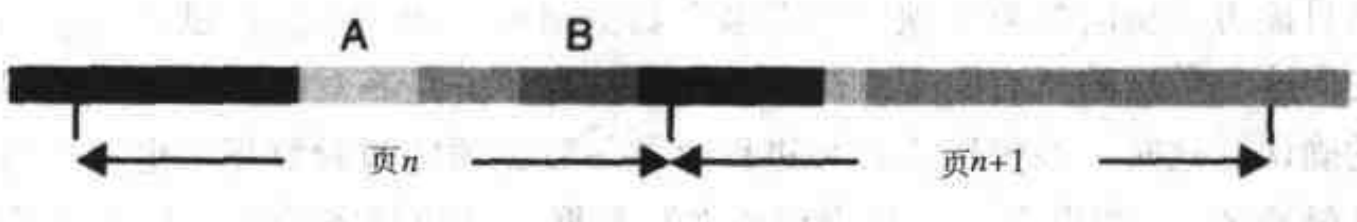


图16-6 在页上分布的数据项

实际上，尽管当页面很小时，可以用几个连续的页面作为数据单元，但选择共享单元的大小时仍然是基于可用物理页的大小。相对于页边界的数据布局是在程序执行时确定传输页数目的一个重要因素。

16.2.6 系统颠簸

写－失效协议的一个潜在问题是系统颠簸。相对于DSM花费在正常工作上的时间而言，DSM在失效操作和传输共享数据上花费了大量的时间，那么系统就发生了颠簸。当多个进程竞争同一数据项或错误共享的数据项时，系统会产生颠簸。例如，如果有一个进程反复地读取一个数据项，而另一个进程有规律地更新它，那么写数据端将经常地输出数据，而读数据端的数据经常失效。在这一个例子中，写－更新方式比写－失效方式更有效。下一节将介绍消除颠簸的Mirage方法，该方法中计算机只在一小段时间内“拥有”页面；16.4节介绍了Munin系统是如何允许程序员为DSM系统确定访问方式，以便为每一数据项选择合适的更新方式，避免系统颠簸。

16.3 顺序一致性和Ivy

本节将介绍实现顺序一致的、基于分页的DSM的方法，我们以Ivy系统[Li and Hudak 1989]作为一个实例进行研究。

16.3.1 系统模型

要考虑的基本模型是有多个进程共享DSM的一个段（图16-7）。这个段被映射到每个进程的相同的地址区上，所以在这个段中可以存储有使用意义的指针值。我们可以假设在每个计算机上只有一个进程访问DSM段。实际上在一个计算机中可能有多个进程。然而，这些在同一计算机上的进程可以直接共享DSM页（不同的进程可以使用在计算机系统页表中的同一页面）。惟一复杂的是在两个或多个本地进程访问页面时，如何协调获取和传播页面更新。下面的描述将省略这一细节。

649

分页机制对于在进程内的应用组件是透明的；它们可以在逻辑上读写DSM中的任意数据。然而，为了进程读写时维持系统的顺序一致性，DSM运行系统将限制页的访问许可。内存页管理将对数据页的访问许可设置为*none*、*read-only*或*read-write*模式。如果一个进程试图超越当前的访问许可，那么系统会根据它的访问方式产生一个读失配或写失配。内核将页失配重定向到由每个进程中的DSM运行层指定的一个处理器。页失配处理器——对应用程序来说是透明的——会在把控制权交还给应用程序前以下面介绍的这种特殊方式处理这个故障。在像Ivy这样的DSM原型系统中，内核本身就执行了我们在上面所介绍的许多操作。我们将介绍执

行页失配处理和通信处理的进程。实际上，进程中的DSM运行层和内核共同提供了这些处理功能。通常，进程中的DSM运行层包含了大多数这样的功能，并且不会因为改变内核而需要重实现和优化。

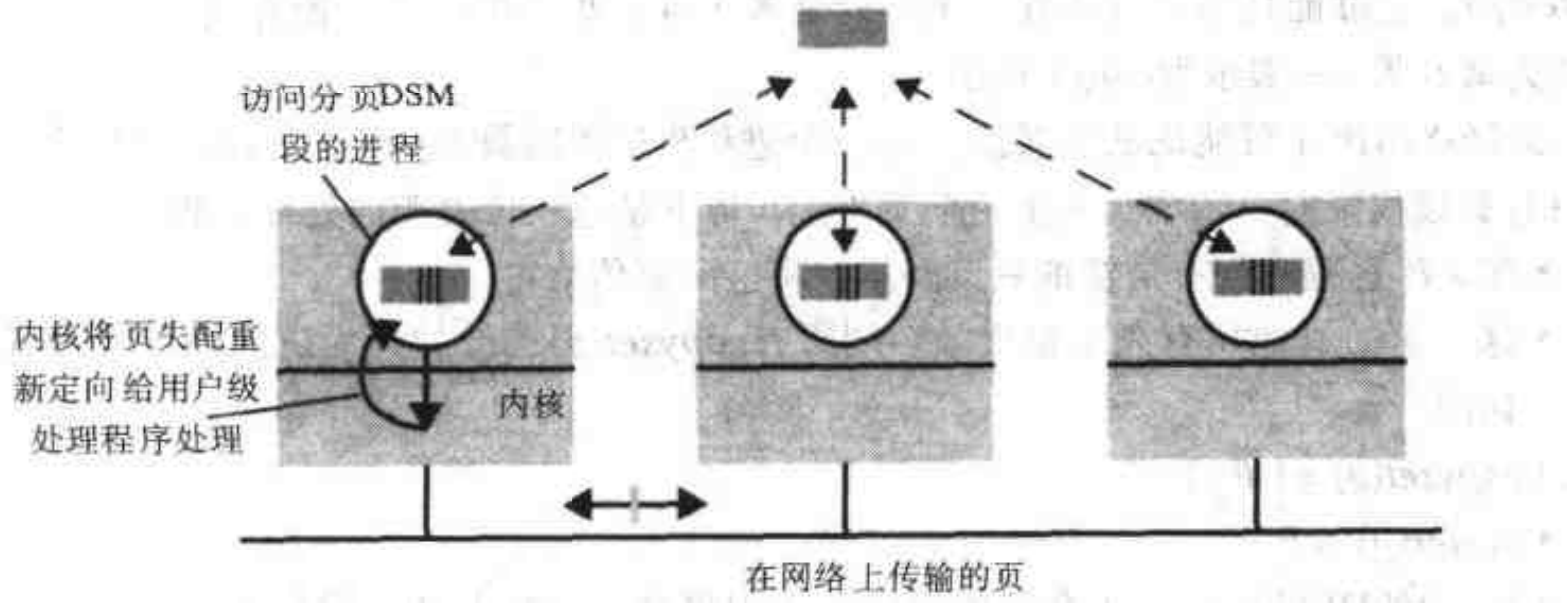


图16-7 基于分页DSM的系统模型

这一描述省略了常规虚拟内存中的页失配处理。除了DSM段可能会和其他段竞争页面这种情况之外，这种实现是相对独立的。

写－更新的问题 前一节简要介绍了写－更新和写－失效这两种实现方法。实际上，如果DSM是基于分页的，那么仅当写操作的更新数据可以被缓存时，系统才可以使用写－更新方法。这是因为基本的页失配处理不适用于处理对一个页面进行多个更新的任务。

为了说明这一点，假设每一次更新都要组播到其余所有的数据副本上。假设一个页已经被写保护了，当一个进程试图对这个页进行写操作时，系统会产生一个页失配并且调用一个处理程序。在原则上，这个处理程序会检查产生故障的指令以决定被写入的值和地址，在恢复写访问和返回到导致故障的指令前，它会组播所做的更新。 650

但是如果恢复写访问，后面对此页的更新就不会引起页失配。为了使每个对此页的写操作都产生页失配，页失配处理程序需要将进程设置为TRACE模式，这样系统在每执行一条指令后都产生一个TRACE异常。TRACE异常处理程序将关掉写访问许可，然后再次关掉TRACE模式。当系统遇到下一个页失配时，它会重复整个操作。显而易见，这种操作方法的开销相当大。在进程的执行过程中，系统可能会产生许多异常处理。

实际上，在基于分页的实现中，使用写－更新仅仅发生下列情况中：在第一个页失配发生后，页仍然保持写许可，并且在传播更新的页之前，系统允许多个写操作发生。Munin使用了这一写－缓冲技术。作为一种极有效的方法，Munin系统试图避免传输整个页——可能这个页只有一小部分被更新。当一个进程试图写这个页时，Munin系统处理故障的方法是：在允许写操作之前获得这个页面的副本，并将其放在一边。然后，当Munin系统准备传播这一更新时，它将更新的页和保存的页副本进行比较，并将两个页的差异进行编码。这些差异的数据量通常比一个整页的数据量小。其他进程可以根据这些差异和更新前的页副本生成更新后的页副本。

16.3.2 写失效

基于失效的算法使用页保护来实现一致数据共享。当有一个进程更新一个页时，该进程

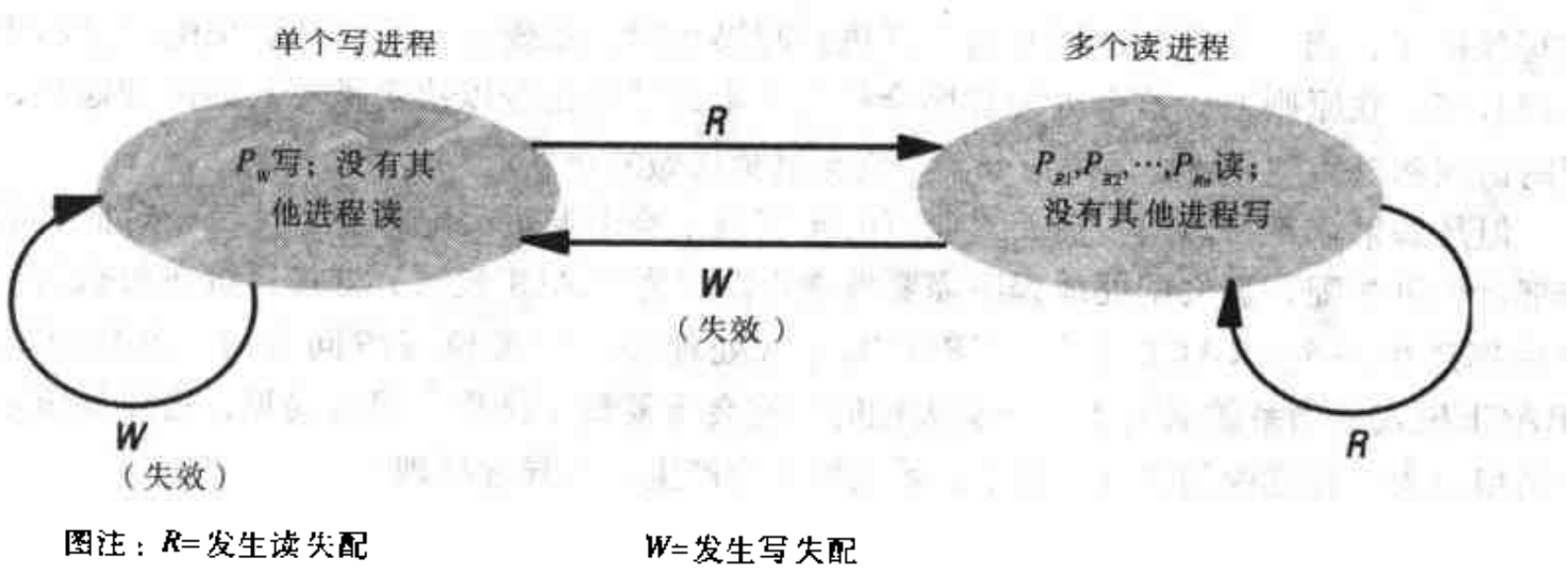
持有对本地数据的读和写许可，其他进程没有对这一页的访问许可。当有一个或多个进程读一个页时，这些进程具有只读许可，所有其他进程没有访问许可（尽管它们可以获得读许可）。除此之外，没有第三种情况。拥有页面p最新版本的进程被指定为这一页的*拥有者*——表示为*owner*(p)。它可能是单个写进程，也可能是多个读进程中的一个。拥有页面p副本的进程集合被称为*副本集*——表示为*copyset*(p)。

图16-8给出了可能的状态转换。当一个进程P_w试图写页面p并且它对这一页没有访问权限或只有只读权限时，系统会发生一个页失配。以下是这一页失配的处理过程：

- 如果P_w进程不拥有最新的只读副本，将这一页传给它
- 这一页的其他所有副本都失效，则所有*copyset*(p)成员的页面访问许可被设置为不允许访问
- *copyset*(p):={ P_w }
- *owner*(p):= P_w
- P_w 中的DSM运行层在合适的地址空间内将这一页加上读-写许可，并且从发生失配的指令处重新执行

651

请注意，两个以上拥有只读副本的进程可能在同一时间发生写失配。当系统最终授予页面所有权时，这个页的一个只读副本可能已经不是最新的了。为了检查当前页的只读副本是否是最新的，系统可以记录每个页的序列号，当页的所有权转移时，它的序列号会增加。当一个进程需要进行写访问时，它会获得这一只读副本的序列号。那么当前的拥有者就可以知道是否这个页已经被更新，是否需要发送出去。Kessler和Livny[1989]将这一方案描述为“准确算法”。



图注：R=发生读失配　　W=发生写失配

图16-8　在写-失效方法中的系统状态转换

当进程P_R试图读页p，而它没有对这一页的访问许可，系统就会发生一个读失配。以下是这一页失配的处理过程：

- 系统将页从*owner*(p)拷贝到P_R中。
- 如果当前页拥有者是单个写进程，那么仍然保留它对p的所有权，并且它对p的访问许可被设置为只读访问。系统最好保留它的读访问许可，因为这一进程随后可能试图读这一页——它已经保留了这一页的一个最新副本。然而，如果这样，甚至在这一进程不再访问这一页的情况下，因为它是这一页的拥有者，它必须处理随后的对这一页的请求。这说明系统最好将这一进程的访问许可改为不允许访问，并且将页的拥有权转交给P_R。

- *copyset*(*p*):= *copyset*(*p*) ∪ { P_k }。
- P_k中的DSM运行层在合适的地址空间内将这一页加上读－写许可，并且从发生失配的指令处重新执行。

系统可能在上面所介绍的转换算法执行中发生第二个页失配。为了使这一转换能一致地执行，系统不处理新的页请求，知道这一转换完成为止。 652

以上仅介绍了必须要做什么。下面将介绍如何高效实现页失配处理问题。

16.3.3 失效协议

实现失效方案的协议要解决两个重要的问题：

1）如何为给定的页 *p*定位*owner*(*p*)。

2）在什么地方存储*copyset*(*p*)。

在Ivy系统中，Li和Hudak[1989]描述了多种体系结构和协议，采用了多种方法来解决这一问题。我们要介绍的一个最简单的算法是他们改进的中央管理器算法。在这个算法中，系统使用一个被称为管理器的单个服务器来为每个页*p*存储*owner*(*p*)的位置（传输地址）。这个管理器可能是运行应用程序的进程之一，也可能是其他进程。在这个算法中，系统将集合*copyset*(*p*)存储在*owner*(*p*)处。也就是说，系统存储*copyset*(*p*)成员的进程标识和传输地址。

正如图16-9所示，当一个页失配发生时，本地进程（我们称它为客户）向管理器发送一个包含页号和关于访问类型（读和读－写）的信息。然后客户等待应答。管理器通过查找*owner*(*p*)的地址并将这一请求转发给页拥有者这种方式来处理这一消息。在发生写失配的情况下，管理器将新的所有权交给写数据的客户，后续的请求会在客户方进行排队，直到客户完成了所有权转换为止。

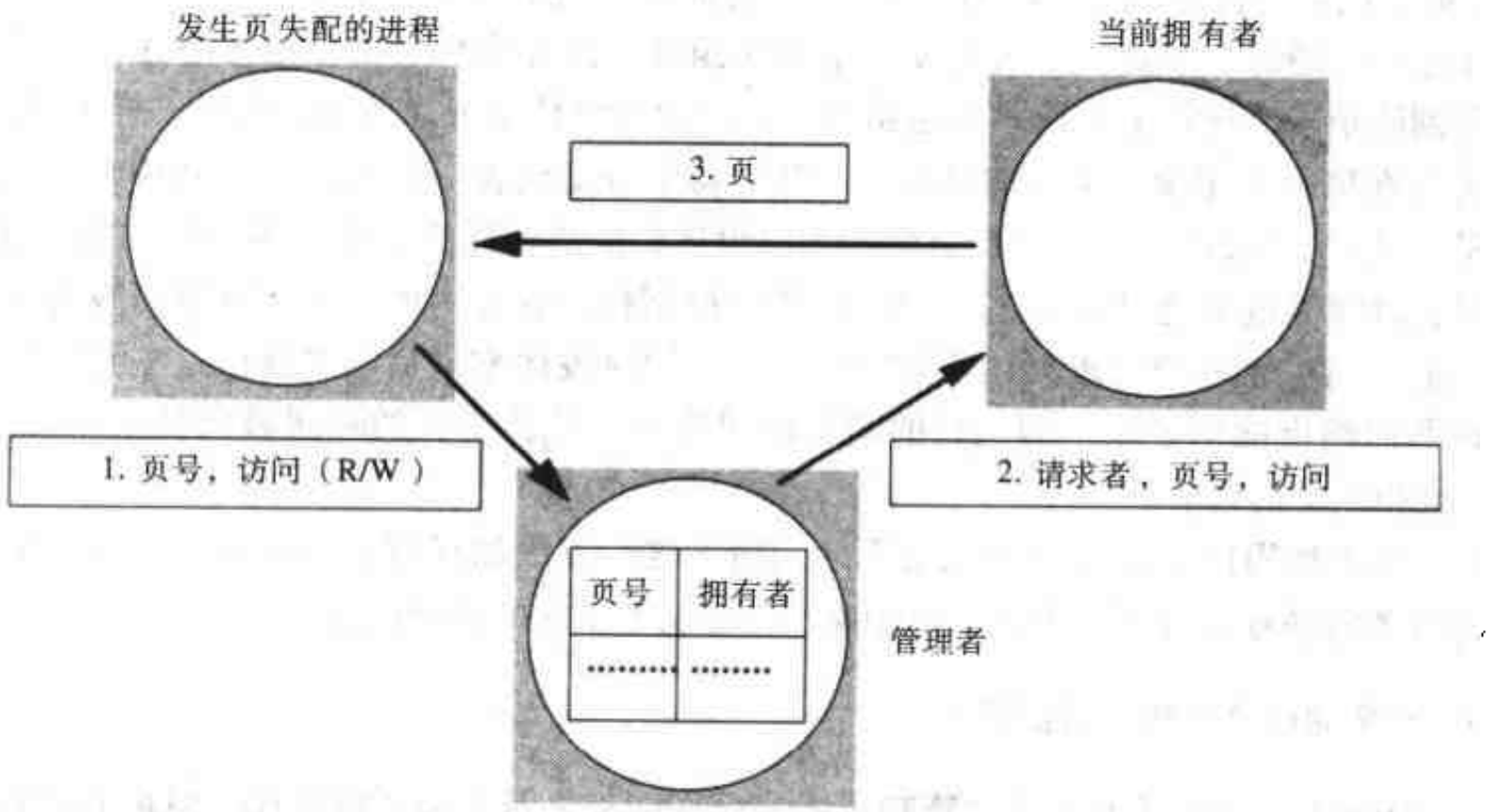


图16-9 中央管理器和与其相关的信息

前一个页拥有者将这一页传给客户。在发生写失配的情况下，它同时也发送页的副本集*copyset*(*p*)。当客户得到这一副本集时，它就执行失效过程。它向副本集的成员进程组播一个请求，并等待这些进程发回“失效已经被执行”的确认信息。这一组播不需要对消息发送进

行排序。客户不需要向前一个页拥有者发送失效消息，这是因为它在所有权转移时已经将自己失效了。至于副本集的管理目前留给读者自己解决，读者可以参考上面所描述的通用失效
653 算法来解决该问题。

这个管理器是系统性能的瓶颈，并且它的失效会导致系统崩溃。Li和Hudak介绍了3种改进的方法来使多个计算机共同承担页管理的负载：固定的分布式页管理、基于组播的分布式管理和动态分布式管理。在第一种方法中，系统使用多个管理器，每一个都在功能上与上面所介绍的中央管理器等价，页面被静态地划分给这些中央管理器。例如，每一个管理器只管理那些页号被散列函数处理为特定值的页。客户计算所需页的散列数，然后在一个预先决定的配置表中找到相应管理器的地址。

这种方案通常可以缓解负载集中这一问题，但是缺点在于系统可能不适合使用固定的从页到管理器的映射。当进程不是均匀地访问页时，一些管理器的负载可能比其他管理器的负载大很多。下面将介绍基于组播的分布式管理和动态分布式管理。

使用组播来定位拥有者 使用组播机制可以完全不使用管理器。当一个进程发生页失配时，它将自己的页请求组播给所有其他进程，只有那个拥有此页的进程给出应答。但必须注意，系统可能会发生有两个客户在同一时刻发出对同一页的请求：甚至在一个进程正在进行所有权转换时，另一个进程发出对这一页的请求，这都可能导致两个客户同时拥有这一页。

假设有两个客户C_1和C_2，它们使用组播来定位一个进程O拥有的页。假设O首先接收到了C_1的请求，并将所有权传给它。在传输的页面到达C_1之前，C_2对这一页的请求到达O和C_1，因为O不再拥有这一页，所以它会拒绝这一请求。Li和Hudak指出C_1会延迟处理C_2的请求，直到它获得这一页为止——否则，如果它因为还不是拥有者的原因而拒绝了这一请求，那么请求会被丢失。然而，这样仍然有问题。因为C_2的等待队列中也会有C_1的请求。当C_1最终将页面传给C_2时，C_2就会接收和处理C_1的请求——而此时这一请求已经过期了。

对这个问题的一个解决方法是使用全排序组播，这样客户可以拒绝那些在自己的请求到达前收到的请求（进程会将请求发送给自己，就像它将请求发送给其他进程一样）。另一种解决方法是在每页上记录一个时间戳向量，其中每个分量代表一个进程（见第10章介绍的时间戳向量），这一方法可以使用开销更少的无序组播，但是消耗的带宽要多一些。当页的所有权转移时，时间戳也随之转移。当一个进程获得控制权，它会增加自己在页时间戳向量中的时间戳的值。当一个进程请求页的所有权时，它会同时发送它最后一次获得这一页面时的时间戳。在我们给出的例子中，因为C_1的请求时间戳低于C_2获得页的时间戳向量中C_1的时间戳，所以C_2会拒绝C_1的请求。

不管系统使用的是有序组播还是无序组播，这一方法拥有所有组播方法的共同缺点：不是页拥有者的进程会被不相关的信息中断，这样浪费了它的处理时间。

16.3.4 一个动态分布式管理器算法

Li和Hudak建议的动态分布式管理器算法允许进程之间传递页所有权，但是它没有使用组
654 播机制，而是使用了一种定位页所有者的方法。该方法是在访问页的计算机之间分担定位页操作的负载。对每个页p，每个进程都记录该页当前拥有者的提示信息——p的可能拥有者被记为$probOwner(p)$。开始时，每个进程都记录了页面位置的精确信息。然而，在一般情况下，这些值被称为提示信息，这是因为页面可以在任意时间内转换它的所有权。而在以前的算法

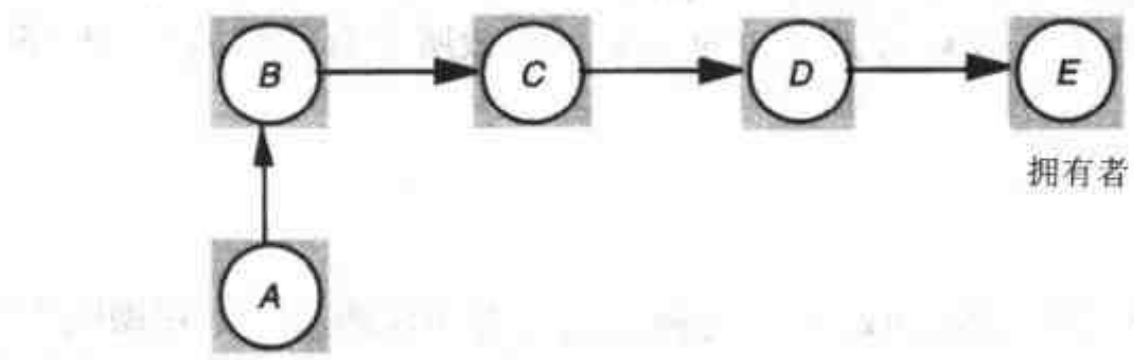
中，系统只有在写失配发生时才转换页所有权。

系统可以沿着由多个进程的提示信息组成的提示链找到页拥有者。这一链的长度——也就是为了找到拥有者需要传递信息的次数——可能会无限增长。通过采用在有更新的值可用时才更新提示信息这一方法，算法解决了链长无限增长的问题。以下就是更新提示信息以及传递请求信息的方式：

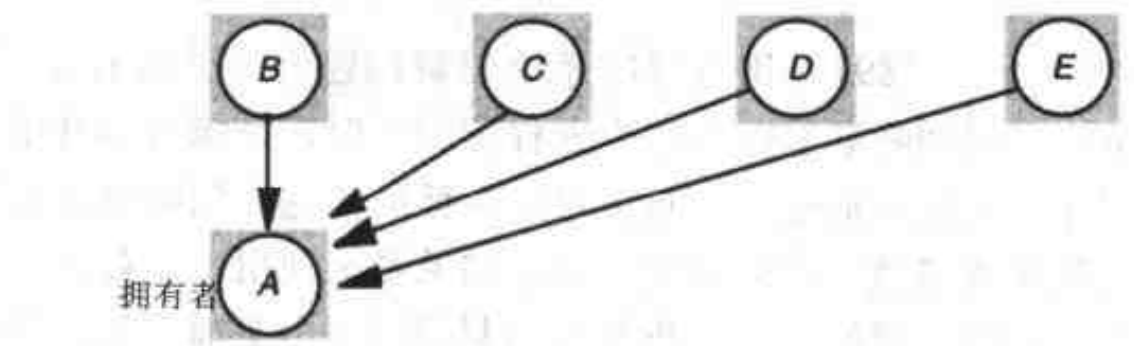
- 当一个进程将页*p*传输给其他进程时，它将*probOwner*(*p*)设置为接收进程。
- 当一个进程处理页*p*的失效请求时，它将*probOwner*(*p*)设置为请求失效的进程。
- 当一个请求读访问的进程获得页*p*时，它将*probOwner*(*p*)设置为这一页的提供进程。
- 当一个进程接收到对一个不属于它的页*p*的请求时，它将这一请求转发给*probOwner*(*p*)进程，并且将*probOwner*(*p*)设置为请求页的进程。

前面的3种更新只是在页所有权转移和提供只读副本时发生。而转发请求时更新所有者的合理性在于：对于写请求，请求者立即成为拥有者。实际上，在Li和Hudak的算法中，不管进程接收到是读请求还是写请求，它立即进行*probOwner*更新。后面，我们将再对其进行讨论。

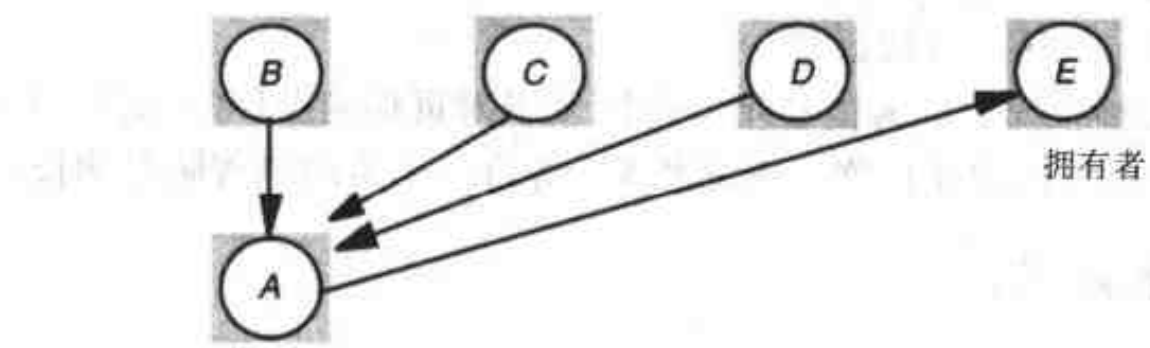
图16-10（(a)和(b)）表示了在进程*A*发生一个写失配之前和之后的*probOwner*指针的情况。进程*A*的*probOwner*指针最初是指向进程*B*的，进程*B*、*C*和*D*的*probOwner*指针组成的指针链最终将请求发送到进程*E*；然后，根据刚才所介绍的更新规则，这些指针都被修改为指向进程*A*。在处理页失配后的指针安排显然要优于处理前的指针安排：指针链消失了。



(a) 在进程*A*在*E*拥有的页上发生一个页失配前的*probOwner*指针情况



(b) 写失配：在*A*的写请求被传输后的*probOwner*指针情况



(c) 读失配：在*A*的读请求被传输后的*probOwner*指针情况

图16-10 更新*probOwner*指针

然而，如果*A*发生了读失配，那么进程*B*对这一页的访问步骤变少了（原来访问*E*需要3步，

现在只需要两步），进程*C*对这一页的访问和以前一样（都是两步），但是进程*D*的情况变糟了，原来只需要一步，现在需要两步（图16-10（c））。为了观察这一策略的性能，需要进行模拟估计。

定期向所有进程广播页的当前拥有者地址可以减少指针链的平均长度。在一次广播后，所有的指针链的长度为1。

为了观察指针更新算法的效率，Li和Hudak实现了一个仿真系统，并给出了仿真结果。如果随机选择故障进程，对1024个处理器，如果系统平均每256次页失配就广播一次拥有者地址，消息到达页拥有者的平均传递步骤是2.34，而如果平均每1024次页失配广播一次，那么，平均传递步骤是3.64。这些数字只作为一个参考，Li和Hudak[1989]给出了完整的结果。请注意，使用中央管理器的DSM系统为了获得页所有权也需要两条消息才能到达页拥有者。

最后，Li和Hudak描述了一种优化方法，该方法可能可以使失效操作效率提高，并可以使处理一个读失配所需的消息传递步骤减少。在此算法中，进程不需要向页拥有者获取页副本，客户可以从任意拥有正确副本的进程中获得副本。当客户试图定位页拥有者时，它可能沿指针链到达拥有者之前就遇到拥有正确副本的进程。

为了实现这一点，进程必须保留一个从该进程获得页面副本的客户记录。拥有页的只读副本的进程被组织成一个树，它的根就是页拥有者，每个结点指向子结点，这些子结点是从父结点获得页副本的进程。一个页的失效操作从页拥有者开始，并沿着树向下传递。当一个结点收到失效消息后，它将此消息传递到子结点上，并使子结点上的页副本失效。整体效果是使一些失效操作并行进行。这样可以减少使一个页失效所花费的时间——特别是在硬件不支持组播的系统环境中。

655
~
656

16.3.5 系统颠簸

有人说避免系统颠簸是程序员的职责，这种说法是有争议的。为了帮助DSM运行层减少页副本和所有权转换，程序员可以为数据项加上注解。下一节介绍Munin DSM系统时会介绍这一方法。

Mirage[Fleisch and Popek 1989]采用的解决系统颠簸问题的方法对程序员是透明的。Mirage将每一个页与一个小时间间隔联系起来。当进程访问页时，系统可以在给定的时间间隔（就像某种类型的时间片）内允许延迟该访问。对这一页的其他访问同时也被延迟。这一方案的一个明显的缺点是很难确定这一时间片的长度。如果系统使用一个静态选择的时间片长度，在很多情况下是不合适的。例如，一个进程可能只访问一个页面一次，然后再也不访问它了；然而，其他进程的访问因此被延迟。考虑到公平性，系统可以在这一进程结束使用页之前就赋予其他进程访问这一页的权利。

DSM系统可以动态地选择时间片的长度。可以基于对页访问操作的观察（使用内存管理单元的引用记录位）来选择时间片长度，还应当考虑等待一个页的进程队列的长度。

16.4 释放一致性和Munin

前一节所介绍的算法是用来实现顺序一致的DSM。实现顺序一致性的优点是DSM能以程序员所希望的方式来共享内存。它的缺点是它的实现开销比较大。在DSM中，不论使用写-更新或写-失效算法，它的实现通常都需要组播机制——虽然失效操作用无序组播就足够了。

定位页拥有者的开销也比较大：管理所有页拥有者定位的中央管理器很可能会成为系统的瓶颈；如果使用进程指针，则一般要传递更多的信息。另外，基于失效的算法可能会引起系统颠簸。

释放一致性是由Dash多处理器系统引入的，这一系统是用硬件实现DSM的，主要使用的是写－失效协议[Lenoski *et al.* 1992]。Munin和Treadmarks[Keleher *et al.* 1992]使用软件实现了这一系统。释放一致性比顺序一致性弱一些，并且它的实现开销也小一些。不过，程序员可以比较容易使用它的语义。

释放一致性的思想是通过利用程序员使用的同步对象——例如信号量、锁和屏蔽等——来减少DSM的开销。DSM的实现系统可以利用这样的知识，在访问这些对象时，允许在某些时刻系统内存存在不一致性，然而因为使用同步对象，系统在应用程序层仍然可以保持一致性。 657

16.4.1 内存访问

为了理解释放一致性——或者其他考虑同步的内存模型——我们首先根据在同步中内存访问的角色将内存访问进行分类。此外，我们还将讨论为了获得较好的性能如何异步进行内存访问，我们还给出了一个简单的操作模型，说明内存访问是如何生效的。

正如我们在前面提到的，在通用分布式系统上实现的DSM可能使用消息传递来实现同步，而不是使用共享变量，原因是基于效率方面的考虑。但是，了解基于共享变量的同步将有助于理解后面所介绍的内容。下面的伪码用变量上的*testAndSet*操作实现了锁。*testAndSet*函数将lock值置为1，并且在lock值为0时返回0；lock值是1时返回1。它以原子操作方式来执行这一操作。

```
acquireLock(var int lock):        // 锁通过引用传递
    while(testAndSet(lock)=1)
            skip;
releaseLock(var int lock):        //锁通过引用传递
    lock: =0;
```

内存访问的类型 内存访问主要分为两类，竞争性访问和非竞争性（常规）访问。当如下情况发生时，两个访问是竞争性的：

- 它们是同时发生的（它们之间没有强制的顺序）
- 至少有一个访问是写访问。

所以两个读访问操作是不会竞争的。两个进程对同一内存地点进行的一个读访问和一个写访问，如果它们之间有同步机制（因此会对它们进行排序），那么它们也不是竞争的。

我们又可以将竞争性访问分为同步和非同步访问两种类型：

- 同步访问是指那些与同步有关的读或写操作。
- 非同步性的访问是指那些并发但与同步无关的读或写操作。

在（上面的）*releaseLock*中，由*lock* := 0隐含的写操作是同步访问。在*testAndSet*中隐含的读操作也是同步访问。

同步访问是竞争的，这是因为同步进程将会以并发的方式来访问这些同步变量，并且它们会更新这些变量：只使用读操作不可能实现同步。但是并不是所有的竞争性访问都是同步访问——使用某些并行算法的进程，它们只是更新和读取另一个进程记录的结果，它们不是

同步的，但对共享变量进行竞争性的访问。

658 同步访问又可以进一步根据访问的作用是阻塞进行访问的进程还是让一些其他进程进行访问，被分为获得访问和释放访问。

执行异步操作 我们可以发现，在顺序一致DSM的实现中，内存操作可能会发生明显的延迟。尽管有这些延迟，系统仍可采用多种形式的异步操作来增加进程执行的效率。首先，写操作可以异步实现。在写操作的结果被传播，并且其他进程可以得知写操作的效果之前，它的值可以被放置在缓冲区内。其次，DSM系统可以预计可能到达的访问，事先就获得数据，这样可以在进程需要获得数据时不产生延迟。最后，处理器可以不按照规定的顺序执行指令。它们可以在等待当前内存访问完成的时候就执行下一条指令，当然下一条指令不能依赖于当前指令。

就我们所列出的异步操作而言，我们将区分读或写操作发生的时间点——当进程开始执行操作的时刻——和指令被执行完或完成的时间点。

我们假设我们的DSM系统至少是连贯的。正如16.2.3节介绍的，这说明在同一地点上每个进程的顺序和写操作的顺序一致。在这一假设下，我们可以在给定的地点上明确地给出写操作的顺序。

在分步式共享内存系统中，我们可以为进程P执行的内存操作o画一条时间线（见图16-11）。

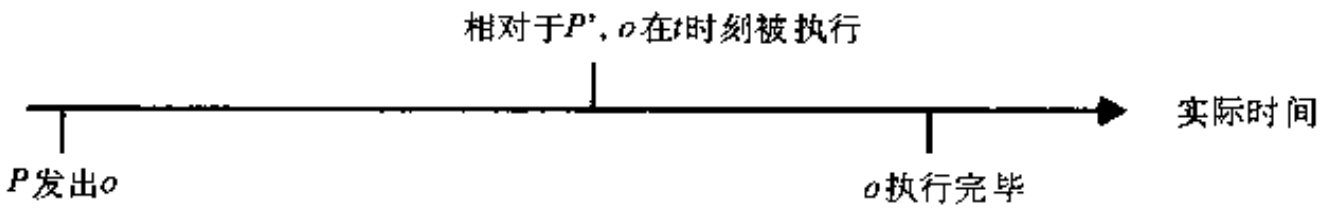


图16-11 执行DSM 读或写操作的时间线

我们说，一个进程P执行了一个写操作$W(x)v$，如果在该操作后，进程P的读操作会返回写操作写入的值v，或者由后续的操作写入的值x（其他操作可能也会写入同样的值v）。

相似地，我们说进程P执行一个读操作$R(x)v$，如果在同一地点上没有发生后续的写操作来提供进程P读出的值v。例如，P可能事先就获取它所需要读出的值。

最后，如果对所有的进程而言都完成了这个操作o，我们就说操作o完成了。

16.4.2 释放一致性

我们希望满足的系统需求包括：

659
- 保持对象（如锁和屏蔽等）的同步语义；
- 为了获得较好的性能，我们允许内存操作一定程度的异步性；
- 为了保证执行能提供与顺序一致性等价的一致性，限制在内存访问之间的重叠。

人们设计出释放一致性模型来满足这些需求。Gharachorloo等[1990]是这样定义释放一致性的：

RC1：在允许其他进程执行普通的读或写操作前，必须执行完所有以前的获得访问。

RC2：在允许其他进程执行释放操作前，必须执行完所有以前的普通读和写操作。

RC3：就获得和释放操作而言，两者必须是顺序一致性的。

RC1和RC2保证了当释放操作发生时，没有其他获得锁的进程可以读那些由已执行释放的进程所更新的旧版本数据。这种方式和程序员所希望的释放锁的方式一致，例如，标明进程

已经结束对临界区中的数据进行修改的操作。

如果DSM系统知晓有哪些同步访问，那么它能增强释放一致性保证。例如，在Munin系统中，程序员只能使用Munin系统自己的*acquireLock*、*releaseLock*和*waitAtBarrier*原语。（屏蔽是一种同步对象，它阻塞住一个进程集合中的所有进程，直到所有的进程都等待它；然后，所有的进程继续执行。）程序必须使用同步来保证其他进程都可及时获得更新的结果。如果DSM的实现只使用上面所给出的单一保证，那么共享DSM但不使用同步对象的两个进程可能永远不能获得对方的更新结果。

请注意，释放一致性模型是允许采用某些异步操作来实现的。例如，当一个进程在一个临界区内执行更新时，系统没有必要阻塞它。同时，在它用释放锁的方式离开临界区以前，也没有必要立即传播它的更新。另外，多个更新可以被收集在一起，通过一个消息来传递该更新结果。这时，只需要传播最后一个对数据项的更新。

考虑图16-12中的进程，它们为了访问变量a和b这一对数据（a和b的初始值都为0）获得和释放锁。进程1在互斥的情况下更新a和b，所以进程2不能在进程1进行写操作时同时读取a和b的值，所以会发现$a=b=0$或$a=b=1$。临界区保证了应用程序层的一致性——a等于b。系统没有必要立即传播在临界区内变量的更新。如果进程2在临界区外试图访问a，它会获得一个过期的值。对应用程序的编写者来说，这是一个问题。

假设进程1首先获得锁，那么进程2会被阻塞，并且它不会引起相关于DSM的任意活动，直到它获得锁并试图访问a和b为止。如果两个进程对具有顺序一致性的内存系统操作，那么当进程1更新a和b时，进程1将会被阻塞。在写-更新协议中，直到所有版本的数据被更新后，系统才能解除对它的阻塞；在写-失效协议中，它将被阻塞到所有数据副本被失效后为止。 660

```
进程 1:
        acquireLock();          //进入临界区
        a := a + 1;
        b := b + 1;
        releaseLock();          //离开临界区
进程 2:
        acquireLock();          //进入临界区
        print("The values of a and b are:",a, b);
        releaseLock();          //离开临界区
```

图16-12 进程在释放一致的DSM上的执行

在释放一致性中，当进程1访问a和b时，进程1不会被阻塞。DSM运行系统知道这些数据被更新了，但是它没有必要此时就采取进一步的行动。仅当进程1释放了这一锁后，系统才需要通信。在写-更新协议中，系统需要传播对a和b的更新值；在写-失效协议中，系统需要发送失效消息。

程序员（或编译器）负责将读和写操作标记为释放、获得和非同步访问——其他指令被认为是普通操作。这些访问操作的标记将指导DSM系统执行释放一致性条件。

Gharachorloo等[1990]描述了适当标记程序的概念。他们证明了这样的一个程序在释放一致性DSM上和在顺序一致性DSM上没有区别。

16.4.3 Munin

Munin DSM系统[Carter *et al.* 1991]的设计试图通过实现释放一致性模型来改进DSM的效

率。此外，Munin系统允许程序员以数据项被共享的方式来标记数据项，这样，系统可以对维护一致性的更新选项进行一些优化。Munin实现在V内核[Cheriton and Zwaenepoel 1985]上，这个系统是首先允许用户级进程来处理页失配和操作页表的内核之一。

下面是Munin系统实现释放一致性所采用几个机制：

- 当锁被释放时，Munin系统系统立即发送更新或失效信息。
- 程序员可以在与锁相关的数据项上加上注解。在这种情况下，DSM运行系统可以利用将锁传输给等待进程的这一信息来传播相关的更新——假设锁的接收进程在访问数据前就拥有这一数据的副本。

相对于Munin在锁释放时就及时发送更新和失效消息的及时方法，Keleher等[1992]描述了另一种方法。这种惰性方法仅当这一锁下一次被获得时才进行消息传播。此外，它只将这些信息发送给那些需要获得锁的进程，这一信息是搭载在授予锁的信息中一起传送的。在其他进程获得这一锁以前，没有必要让这些进程知道这一更新结果。

661

共享注解 Munin实现了多种一致性协议，它们可以被应用在不同的数据项粒度上。这些协议的参数根据如下选项确定：

- 是使用写－更新协议还是使用写－失效协议；
- 是否允许多个可修改的数据项副本同时存在；
- 是否延迟更新或失效操作（例如，在释放一致性模型中）；
- 是否允许数据项有固定的拥有者（所有的更新操作都被送到这一拥有者上）；
- 是否允许多个写进程并发修改同一数据项；
- 是否允许数据项被固定的进程集共享；
- 是否允许数据项被修改。

系统管理者可以根据数据项的特性和进程共享数据项的模式来选择这些选项。程序员也可以对每个数据项进行这样的参数选择。然而，Munin系统为程序员提供了一个标准的小注解集，这些注解可以应用在数据项上，每一个注解对应一组以上选项的参数选择，这一注解集适合于不同的应用程序和数据项。这个注解集包括如下注解：

- 只读 数据项在初始化之后就不能被更新，它可以被自由复制。
- 迁移 进程通常轮流地访问数据项，其中至少有一个访问是更新操作。例如，进程在临界区内访问数据项。Munin系统对这样的对象同时进行读和写访问，甚至当进程发生一个读失配时也是如此。这样可以节省后面的写失配处理。
- 写－共享 多个进程对同一数据项进行并发更新（例如，一个数组），但是这一注解表示程序员已说明进程不会同时对这一数据项的同一部分进行更新。这就意味着Munin系统可以避免错误共享，而只传播那些实际被每个进程更新的部分。为了做到这一点，Munin系统（而不是写失配处理程序）在执行更新前复制该页。在更新后，系统只传递两个版本的差异信息。
- 生产者－消费者 数据对象被固定的进程集共享，其中只有一个进程对数据对象进行更新。正如前面我们在介绍系统颠簸时所提到的，写－更新协议最适合这种情况。此外，在释放一致性模型下，假设进程使用锁来同步数据访问，那么更新可能被延迟。

662

- 缩影 数据项通过加锁、读、更新和解锁面被修改。其中一个例子是：如果并行计算的一个全局缩影比局部缩影大，它必须以原子方式获得和修改数据。这些数据项被存储在

一个固定的拥有者上。数据更新被传送到拥有者上，它负责向其他进程传播此更新。

- 结果　系统中有多个进程更新一个数据项的不同部分；一个进程读整个数据项。例如，不同的“工作者”进程可能负责填充数组的各个元素，而一个“管理者”进程处理这一数组。其中要指出的一点是，更新只要发送到管理者进程上，而没有必要发送到工作者进程上（它可能在上面“写-共享”注解的情况中发生）。
- 常规型　数据项由和前面所描述的失效协议相似的协议管理。因此，进程不会读到数据项的过期版本。

Carter等[1991]详细介绍对应于以上注解的选项参数设定。这一注解集并不是固定的。当需要与之不同的选项参数设置时，其他人也可以生成新的注解。

16.5 其他一致性模型

内存一致性模型可以被划分为两类：统一型模型和混合型模型。统一型模型不区分内存访问的类别，而混合型模型区分普通访问和同步访问（以及其他类型的访问）。

一些统一型模型比顺序一致性弱一些。我们在16.2.3节介绍了连贯性，连贯性是在每一个场地都是顺序一致的。进程执行写操作的顺序和在给定地点上的操作被执行的顺序一致，但是在不同进程中和不同地点上的写操作被执行的顺序可能不一致[Goodman 1989, Gharachorloo *et al.* 1990]。

其他统一型一致性模型包括：

- 因果一致性　在读和写操作之间可能存在发生在先关系（见第10章）。当内存操作是以下几种情况之一时，它们之间就存在这种关系。这几种情况是：（a）它们是同一进程执行的操作；（b）一个进程读另一个进程写入的数值；（c）存在着用于连接两个操作的操作序列。这个模型的限制在于，一个读操作的返回值必须与发生在先关系一致。Hutto和Ahamad[1990]描述了这一模型。
- 处理器一致性　内存是连贯的，并且符合管道RAM模型（见下面）。简单地说，处理器一致性就是内存是连贯的，并且操作执行的顺序和同一进程执行的任意两个写访问的顺序一致——也就是说，它们符合程序的顺序。Goodman[1989]首先非形式化地定义了这一模型，后来Gharachorloo等[1990]和Ahamad等[1992]形式化地定义了这一模型。
- 管道RAM　所有处理器处理操作的顺序和任意给定的一个处理器发出写操作的顺序一致[Lipton and Sandberg 1988]。

663

除了释放一致性之外，混合型模型还包括：

- 变量项一致性　变量项一致性是为Midway DSM系统设计的[Bershad *et al.* 1993]。在这一模型中，每一个共享变量都和一个像锁这样的同步对象联系起来，这些同步对象管理对变量的访问。系统可以保证首先获得锁的进程读到数据最新的值。写变量的进程必须首先以“互斥”的方式获得相应的锁——这样使其成为惟一能对变量操作的进程。通过以非互斥的形式来共同拥有锁，多个进程可以并发地读变量。Midway系统避免了在释放一致性中的错误共享，但是它增加了编程的复杂性。
- 作用域一致性　这种内存一致性模型[Iftode *et al.* 1996]试图简化变量项一致性的编程模型。在作用域一致性中，变量可以较自动地与同步对象关联起来，而不是需要程序员来将锁和变量联系起来。例如，系统可以管理在临界区内更新的变量。

- 弱一致性 弱一致性[Dubois *et al.* 1988]不区分获得和释放同步访问。它的保证之一是所有以前的普通操作在任意类型的同步操作之前完成。

讨论 释放一致性和一些其他的一致性模型比顺序一致性模型弱，它们似乎更适合于DSM。在释放一致性模型中DSM运行系统必须知道同步操作，但是它并没有成为明显的缺点——只要系统能提供足够的选项满足程序员的需要。

在混合型模型中，只要程序员适当地同步他们的数据访问，他们就不必考虑特别的内存一致性语义，认识到这一点很重要。但是，在DSM的设计中，为了使程序高效地执行，就要求程序员对他的程序加入很多注解，这样做是很危险的。这些注解包括使用同步对象标明数据项的注解和那些像在Munin系统中共享的注解。基于消息传递的共享内存编程的优点之一是它的开销相对小一些。

16.6 小结

本章介绍了分布式共享内存的概念，我们将其看作为在分布式系统中共享内存的一种抽象，在分布式系统中，它是与基于消息通信的共享内存相对的另一种方法。DSM主要应用于并行处理和数据共享。在某些应用程序中，它的执行效果和消息传递机制一样好，但是高效实现DSM是很难的，并且它的性能随应用程序的不同变化很大。

本章主要介绍DSM的软件实现——特别是那些使用虚拟内存子系统的软件实现，不过，
664 目前DSM已经在硬件支持的基础上实现了。

DSM最主要的设计问题和实现问题包括：DSM结构、应用程序实现同步的方法、内存一致性模型、使用写－更新协议还是写－失效协议、共享的粒度以及系统颠簸。

DSM的结构包括如下几种形式：字节序列、共享对象的集合或者是一些像元组这样的不变数据的集合。

为了满足与应用程序相关的一致性约束，使用DSM的应用程序需要同步。它们使用像锁这样的对象来满足这一要求，为了获得较好的效率，它们使用消息传递机制来实现。

在DSM系统中最常见的严格内存一致性是顺序一致性。但因为它的开销比较大，人们设计出了一些弱一些的一致性模型，例如连贯性和释放一致性。释放一致性使DSM的实现可以在不破坏应用程序层的一致性的情况下来使用同步对象以获得更好的效率。本章还列出了其他几种一致性模型，其中包括变量项、范围和弱一致性，它们都利用同步来实现应用层一致性。

写－更新协议在数据项被修改时将更新信息传递到它所有的数据副本上。尽管它也有用全排序组播实现的，但通常还是使用硬件实现的。写－失效协议在数据项被更新时，通过将数据副本失效来阻止进程读到过期的数据。它非常适合于基于分页的DSM系统，因为在这种系统中使用写－更新协议可能会使系统开销比较大。

DSM的粒度会影响进程竞争的可能性，因为这些进程访问的内容可能被包含在同一共享单元中（例如页），因而，可能错误地共享了数据，从而引起进程竞争。粒度也会影响在计算机之间传输一个字节的更新所需的开销。

当系统使用写－失效协议时，它可能会发生系统颠簸。系统颠簸是指在竞争进程之间重复地传递数据，妨碍程序的进展。用程序层的同步，或者允许计算机在一小段时间内保留页，或者通过标记数据项来使系统同时批准读写访问，这些方法都可以减少系统颠簸。

本章还描述了Ivy系统的应用于分页的DSM的3个主要写－失效协议，主要介绍了管理副

本集和定位页拥有者问题。它们包括：中央管理器协议（使用单个进程来存储当前每一页的拥有者信息）；使用组播来定位页的当前拥有者协议以及动态分布式管理器协议（使用向前的指针来定位页拥有者）。

Munin系统是实现释放一致性的一个实例。在锁被释放时，它就发送更新信息或失效信息，这样它实现了及时的释放一致性。另外，也存在惰性释放一致性的实现，惰性方法仅在需要的时候才传播这些信息。Munin系统允许程序员为他们的数据项加上注解，这样可以选择那些最适合于这些数据项的协议。 665

练习

16.1 请说明在哪些方面DSM适合于客户－服务器系统，在哪些方面不适合。

16.2 请讨论，消息传递机制和DSM二者中哪一个适合于容错应用程序。

16.3 在异构的计算机系统上可以使用中间件来实现DSM，那么，如何解决不同的数据表示这一问题？如果是在基于分页的DSM实现上，又将如何解决这一问题？你的解决方法是否涉及到指针？

16.4 为什么我们需要在用户级实现基于分页的DSM系统，实现它需要些什么？

16.5 你将怎样使用元组空间来实现一个信号量？

16.6 下列两个进程执行后，内存是否保持顺序一致性（假设初始时所有的变量值为0）？

P_1: $R(x)1$; $R(x)2$; $W(y)1$

P_2: $W(x)1$; $R(y)1$; $W(x)2$

16.7 使用$R()$和$W()$的标记方法，设计出一个是连贯的但不是顺序一致的内存执行例子。可能出现内存是顺序一致的但不是连贯的这种情况吗？

16.8 在写－更新中，如果每个更新都在异步传播到达其他副本管理器前就在本地副本上执行了，甚至组播是全排序的，系统也可能不符合顺序一致性。请讨论是否异步组播可以被用来实现顺序一致性。（提示：考虑是否阻塞后续的操作。）

16.9 使用写－更新协议并采用一个异步的、全排序的组播可以实现顺序一致性内存。请讨论实现连贯的内存对组播的排序需求有哪些。

16.10 请解释为什么在写－更新协议中只需要传播那些被本地更新的数据项。

设计一个算法，用于表示一个页在更新前和更新后的差异。讨论这一算法的性能。

16.11 请解释为什么在DSM系统中，粒度是一个重要的问题。请比较在面向对象和面向字节的DSM系统中的粒度问题，注意考虑它们的实现问题。

为什么粒度和包含不变数据的元组空间相关？

什么是错误共享？它能导致不正确的执行吗？

16.12 DSM在页替换策略上的实质是什么（也就是，为了换入一个新页，选择哪一页被淘汰）？ 666

16.13 请证明Ivy系统的写－失效协议保证了顺序一致性。

16.14 在Ivy系统的动态分布式管理算法中，为了减少找到一个页所需的查找次数，采取了哪些步骤？

16.15 为什么在DSM系统中，系统颠簸是一个重要的问题，有哪些方法可以解决这一问题？

16.16 请讨论释放一致性的条件RC2是如何放宽的。然后区分及时释放一致性和惰性释放一致性。

16.17 一个传感器进程将当前的温度值写入存储在释放一致DSM系统的变量t中。一个监控器进程周期性地读取t。说明传播更新给t的同步需求，包括那些在应用程序层不需要的需求。这些进程中，需要谁来执行同步操作？

16.18 请说明下列的操作不符合因果一致性：

P_1: $W(a)0$; $W(a)1$

P_2: $R(a)1$; $W(b)2$

P_3: $R(b)2$; $R(a)0$

16.19 如果DSM实现系统知道数据项和同步对象之间的关联，它如何根据这些情况提高效率？将这种关联显式表示的缺点是什么？

667

第17章 CORBA实例研究

17.1 简介
17.2 CORBA RMI
17.3 CORBA服务
17.4 小结

CORBA是一种中间件设计，它允许不同应用程序间进行相互通信，而不用考虑它们的编程语言、软硬件平台、通信的网络以及它们的实现者。

许多应用是由CORBA对象创建的，CORBA对象实现了以CORBA接口定义语言（IDL）定义的接口。客户通过RMI的方式访问CORBA对象IDL接口中的方法。支持RMI的中间件组件称为对象请求代理（ORB）。

CORBA规范已经由对象管理组织（OMG）的成员共同提出。根据规范实现了许多不同的并支持各种各样编程语言的ORB。

CORBA服务提供能在广大应用领域采用的通用机制。它们包括命名服务、事件服务和通知服务、安全服务、事务服务和并发服务以及交易服务。

17.1 简介

为了使面向对象程序设计能够推动软件的发展，也为了应用日益流行的分布式系统，一个旨在推广采用分布式对象系统的组织OMG (对象管理组织)于1989年正式成立。为了实现其宗旨，OMG倡导使用基于标准面向对象接口的开放式系统。开放式系统可以由异构的硬件、异构的计算机网络、异构的操作系统和编程语言来实现。

一个重要的动机是：允许分布式对象以任意编程语言实现，并且能够进行相互通信。因此，OMG设计了一种独立于任何特定实现语言的接口语言。

他们引入了一种比喻，提出了对象请求代理（ORB）的概念，它的任务是帮助客户调用对象上的方法，包括定位对象、在需要的时候激活对象、把客户的请求传递给执行及应答这些请求的对象。

在1991年，对象请求代理体系结构的规范，也就是熟知的CORBA（公共对象请求代理体系结构），为众多公司所认同和接受。紧接着，1996年又推出了CORBA 2.0规范[OMG 1998a]，其中定义了一些标准，目的是使由不同开发者开发的实现之间能相互通信。这些标准就称为通用ORB间协议（GIOP）。根据设计，GIOP能够在任何具有连接的传输层之上实现。在因特网上实现的GIOP使用的是TCP/IP 协议，故称为因特网ORB 间协议（称IIOP）。

CORBA的语言无关的RMI框架包含以下几个主要组成部分：

- 接口定义语言，即熟知的IDL，17.2节的开头部分将对其做简要介绍，而更完整的描述将体现在17.2.3节。
- 体系结构，详见17.2.2节。
- GIOP定义了一种外部数据表达，称为CDR，详见4.3节的介绍。它还定义了请求－应答

协议中消息的特定格式。除此之外，它还规定了询问对象位置的消息，以便取消请求和报告错误。

- IIOP定义了远程对象引用的标准格式，详见17.2.4节的描述。

CORBA体系结构还考虑了CORBA服务——即一系列可能对分布式应用十分有用的通用服务，这些都将在17.3节做详细介绍，17.3节还更加详细地阐述了命名服务、事件服务、通知服务和安全服务。与CORBA有关的论文的汇总详见CACM特刊[Seetharaman1998]。

在讨论CORBA的以上组成部分之前，我们先从一个程序员的角度介绍一下CORBA RMI
670 的概念。

17.2 CORBA RMI

与单语言RMI系统，如Java RMI的编程相比，像CORBA RMI这样的多语言RMI系统的编程，需要更多的程序员。他们必须首先学习以下几个新概念：

- CORBA提供的对象模型。
- 接口定义语言及其在实现语言上的映射。

CORBA编程的其他方面与第5章的讨论类似。需要特别指出的是，程序员定义了远程对象的远程接口后，用一个接口编译器产生相应的代理和骨架。但在CORBA中，代理是用客户语言创建的，而骨架是用服务器语言创建的。我们将以5.5节中介绍过的简单白板程序为例，介绍如何编写一个IDL说明，以及如何创建服务器程序和客户程序。

CORBA的对象模型 CORBA对象模型与5.2节中介绍过的相似，但是客户不一定必须是对象，任何能向远程对象发送请求消息并能接受应答的程序都可以作为客户。我们所说的CORBA对象是指远程对象。这样，一个CORBA对象实现一个IDL接口，拥有一个远程对象引用，并能应答对其IDL接口中方法的调用。CORBA对象可以由非面向对象语言（例如没有类这种概念的语言）实现。既然不同的实现语言中类的概念不同，有些甚至没有，所以在CORBA中不存在类的概念。因此，类不能在CORBA IDL中定义，也就是说，类的实例不能作为参数传递。但是，各种不同类型和任意复杂性的数据结构都可以作为参数传递。

CORBA IDL 一个CORBA IDL接口描述一个客户能够请求的名字和一系列方法。图17-1显示了两个分别称为*Shape*（第3标号行）和*ShapeList*（第5标号行）的接口，它们是图5-11中定义的接口的IDL版本。在它们的前面是两个*struct*定义，它们在方法定义中用作参数的类型。特别要注意的是，*GraphicalObject*也被定义成一个*struct*，而它在Java RMI例子中是一个类。一个*struct*类型的组件有一系列字段，可包含各种类型的值，比如像一个对象的实例变量，但是它不包含方法。关于IDL更多的介绍见17.2.3节。

CORBA IDL中的参数和结果 通过使用关键字*in*、*out*或者*inout*可将每个参数标记为输入类型或输出类型或者二者皆是。图5-2描述了一个使用这些关键字的简单例子。在图17-1的第7标号行中，*newShape*的参数是一个*in*类型参数，说明该参数应该通过请求消息从客户传递到服务器。返回值提供了一个附加的*out*类型参数——如果没有*out*类型参数，可以说明为*void*类型。

参数可以是基本类型中的任何一种，例如*long*或*boolean*类型，或者是一种构造类型，例如*struct*或者*array*。基本类型和构造类型在17.2.3节有更详细的描述。我们的例子在第1标号行和第2标号行给出了两个*struct*类型数据的定义。序列和数组用*typedef*定义，如第4标号行所示，
671 该行表示一个长度为100的*Shape*类型的元素序列。参数传递的语义如下：

```
struct Rectangle{                                                    1
        long width;
        long height;
        long x;
        long y;
};
struct GraphicalObject {                                             2
        string type;
        Rectangle enclosing;
        boolean isFilled;
};
interface Shape {                                                    3
        long getVersion();
        GraphicalObject getAllState();    // 返回GraphicalObject的状态
};
typedef sequence <Shape, 100> All;                                   4
interface ShapeList {                                                5
    exception FullException{ };                                      6
    Shape newShape(in GraphicalObject g) raises (FullException);     7
    All allShapes();              // 返回远程对象引用的序列          8
    long getVersion();
};
```

图17-1 *Shape*和*ShapeList*的IDL接口

- 传递CORBA对象 任何参数，如果它的类型是一个IDL接口的名字，例如第7标号行中的返回值*Shape*，那么它就是一个到CORBA对象的引用，并且传递的是远程对象引用的值。
- 传递CORBA基本类型和构造类型 基本类型和构造类型的参数是拷贝之后按值传递的。参数到达的时候，由接收者的进程创建一个新的值。例如，作为参数传递的*GraphicalObject*结构（第7标号行）就在服务器上创建了这种类型的一个新拷贝。

在*allShapes*方法（第8标号行）中组合了这两种形式的参数传递，它的返回类型是一个*Shape*类型的数组——即一个远程对象引用的数组。它的返回值是该数组的拷贝，其中每个元素都是一个远程对象引用。

Object类型 *Object*是一种值为远程对象引用的类型的名字。它实际上是所有IDL接口类型例如*Shape*和*ShapeList*等的公共父类型。 672

CORBA IDL中的异常 CORBA IDL允许在接口中定义异常并由它们的方法抛出这种异常。为了说明这一点，我们在服务器中将*Shape*列表定义为一个定长序列（第4标号行），还定义了*FullException*（第6标号行）异常，如果客户试图在序列全满的情况下添加一个图形的话，就由*newShape*方法（第7标号行）抛出该异常。

调用语义 CORBA中的远程调用在默认情况下采用至多一次调用语义。然而，IDL可以通过使用*oneway*关键字指定某个方法的调用为或许语义。客户不会因*oneway*请求阻塞，它只能用于没有返回结果的方法。对于*oneway*请求的一个例子，参见17.2.1节末尾关于回调的例子。

CORBA命名服务 CORBA命名服务将在17.3.1节讨论。它是一个绑定程序，提供多种操作，包括为服务器提供*rebind*操作，用于让服务器能按名字注册CORBA对象的远程对象引

用；还包括为客户提供*resolve*操作，用于让客户能按名字查找这些引用。名字以层次方式构造，并且一条路径中的每个名字都在一个称为*NameComponent*的结构之中。这使得这个简单例子中的访问看起来有些复杂。

CORBA伪对象 CORBA的实现提供了程序员需要使用的一些ORB的功能性接口。由于它们不能像CORBA对象那样使用，例如，它们不能在RMI中作为参数传递，所以称它们为伪对象。它们具有IDL接口，并实现成程序库。与我们的简单例子相关的一点是：

- ORB是代表程序员需要访问的ORB功能性接口的名字。它包括：
  - *init*方法，为了初始化ORB必须调用它。
  - *connect*方法，用于向ORB注册CORBA对象。
  - 其他方法，用于在远程对象引用和字符串之间进行转换。

17.2.1 CORBA客户和服务器举例

本节将概述利用图17-1中IDL定义的*Shape*和*ShapeList*接口创建客户和服务器程序的必要步骤，随后讨论CORBA中的回调。我们使用Java作为客户和服务器编程语言，但是此处介绍的方法对于其他语言来说也是相似的。接口编译器*idltojava*可以应用到CORBA接口中，用于创建如下项目：

- 等价的Java接口。例如，图17-2所示的*ShapeList*的Java接口。
- 每个*idl*接口的服务器骨架。骨架类的名字以*ImplBase*结束，例如*_ShapeListImplBase*。
- 代理类或者客户存根，每一个IDL接口对应一个这样的类。这些类的名字以*Stub*结尾，例如*_ShapeListStub*。

673

- 对应于IDL接口中定义的每个*struct*的Java类。在我们的例子中，创建了*Rectangle*和*GraphicalObject*类。这些类中的每一个都包含了对应于*struct*中每个字段的一个实例变量和一对构造函数的声明，但是没有其他方法。
- 被称为*helper*和*holder*的类，每种类型在IDL接口中都有一个定义。*helper*类包括*narrow*方法，用于将一个给定的对象引用从它所属的类转换到更低层次的类。例如，*ShapeHelper*中的*narrow*方法就将对象类型转换到类*Shape*。*holder*类可以处理*out*和*inout*参数，它不能直接映射到Java。练习17.9是使用*holder*的一个例子。

服务器程序 服务器程序应该包含一个或多个IDL接口的实现。对一个以面向对象语言如Java或C++编写的服务器来说，这些实现是作为伺服类实现的。CORBA对象是伺服类的实例。

```
public interface ShapeList extends org.omg.CORBA.Object {
  Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
  Shape[] allShapes();
  int getVersion();
}
```

图17-2 根据CORBA接口*ShapeList*由*idltojava*生成的Java接口*ShapeList*

当一个服务器创建一个伺服类的实例时，它必须向ORB注册它，ORB将该实例放入CORBA对象中，并给它一个远程对象引用。除非做完这一步，否则它就不能接收远程调用。仔细研究过第5章的读者可能会认识到，注册一个对象会导致它被记录在远程对象表中的相应位置。

在我们的例子中，服务器包括*Shape*和*ShapeList*接口的伺服类形式的实现以及一个在*main*

方法中包含初始化部分的服务器类。

- 伺服类　每个伺服类扩展了相应的骨架类，并使用在等价的Java接口中定义的方法实现了IDL接口中的方法。尽管也可以选择别的名字，但我们还是将实现*ShapeList*接口的伺服类命名为*ShapeListServant*。该类的要点显示在图17-3中。考虑第1标号行中的方法*newShape*，因为它创建了*Shape*对象，所以是一个工厂方法。为了使*Shape*对象成为一个CORBA对象，*newshape*通过它的*connect*方法向ORB注册，如第2标号行所示。关于这个例子的IDL接口和客户类、服务器类的完整版本参见cdk3.net/corba。

```
import org.omg.CORBA.*;
class ShapeListServant extends _ShapeListImplBase {
    ORB theOrb;
    private Shape theList[];
    private int version;
    private static int n=0;
    public ShapeListServant(ORB orb){
        theOrb = orb;
        // 初始化其他实例变量
    }
    public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException {1
        version++;
        Shape s = new ShapeServant( g, version);
        if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s;
        theOrb.connect(s);                                          2
        return s;
    }
    public  Shape[] allShapes(){ ... }
    public int getVersion() { ... }
}
```

图17-3　CORBA接口*ShapeList*的Java服务器程序中的*ShapeListServant*类

- 服务器　服务器类*ShapeListServer*中的*main*方法如图17-4中所示。它首先创建并初始化ORB（第1标号行），然后创建*ShapeListServant*伺服类的一个实例，即一个Java对象（第2标号行），随后通过在ORB中注册这个伺服类的实例使之成为一个CORBA对象（第3标号行）。此后，该对象获得命名服务的一个引用（第4标号行），并在命名服务中使用*rebind*操作注册服务器。最后它等待客户请求的到来。

服务器使用命名服务先获得一个根名字上下文对象（第4标号行），然后创建一个名字组件*NameComponent*（第5标号行），定义一条路径（第6标号行），并且最后使用*rebind*方法（第7标号行）注册名字和远程对象引用。客户执行相同的步骤，但是使用图17-5第2标号行中的*resolve*方法。

客户程序　图17-5给出了一个客户程序的例子。它创建和初始化ORB（第1标号行），然后联系命名服务，利用它的*resolve*方法获得一个远程*ShapeList*对象的引用（第2标号行）。在它调用*allShapes*方法获得服务器上当前拥有的所有*Shape*对应的一系列远程对象引用之后，它调用返回序列中第一个远程对象引用的*getAllState*方法（第4标号行）；结果作为*GraphicalObject*类的一个实例提供。

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListServer {
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                                    1
            ShapeListServant shapeRef = new ShapeListServant(orb);             2
            orb.connect(shapeRef);                                             3
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");                 4
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");             5
            NameComponent path[] = {nc};                                       6
            ncRef.rebind(path, shapeRef);                                      7
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {   sync.wait();}
        } catch (Exception e) { ... }
    }
}
```

图17-4 Java类*ShapeListServer*

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                                    1
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
                ShapeListHelper.narrow(ncRef.resolve(path));                   2
            Shape[] sList = shapeListRef.allShapes();                          3
            GraphicalObject g = sList[0].getAllState();                        4
        } catch(org.omg.CORBA.SystemException e) {...}
}
```

图17-5 CORBA接口*Shape*和*ShapeList*的Java客户程序

*getAllState*方法看起来与我们原来的声明，即在CORBA中对象不能按值传递似乎是矛盾的，因为客户和服务器都处理了*GraphicalObject*类的实例。然而，实际上二者并没有矛盾：CORBA对象返回了一个结构，使用不同的语言的客户会以不同的方式看待它。例如，在C++语言中客户会将它看作一种结构，即使在Java中，创建的*GraphicalObject*类也更像一种结构，因为它不包含任何方法。

客户程序应该总能捕获CORBA的*SystemExceptions*异常，这一异常报告因分布性而引起

的各种错误（见第5标号行）。客户程序也应该能捕获IDL接口中定义的异常，例如由*newShape*方法抛出的*FullException*异常。 674 ~ 676

这个例子阐述了*narrow*操作的使用：命名服务的*resolve*操作返回一个*Object*类型的值，这个类型被狭义化以适合所要求的特定类型——*ShapeList*。

回调 回调可以在CORBA中以一种类似于5.5.1节Java RMI中描述的方式来实现。例如，*WhiteboardCallback*接口可以如下定义：

```
interface WhiteboardCallback{
    oneway void callback (in int version)
};
```

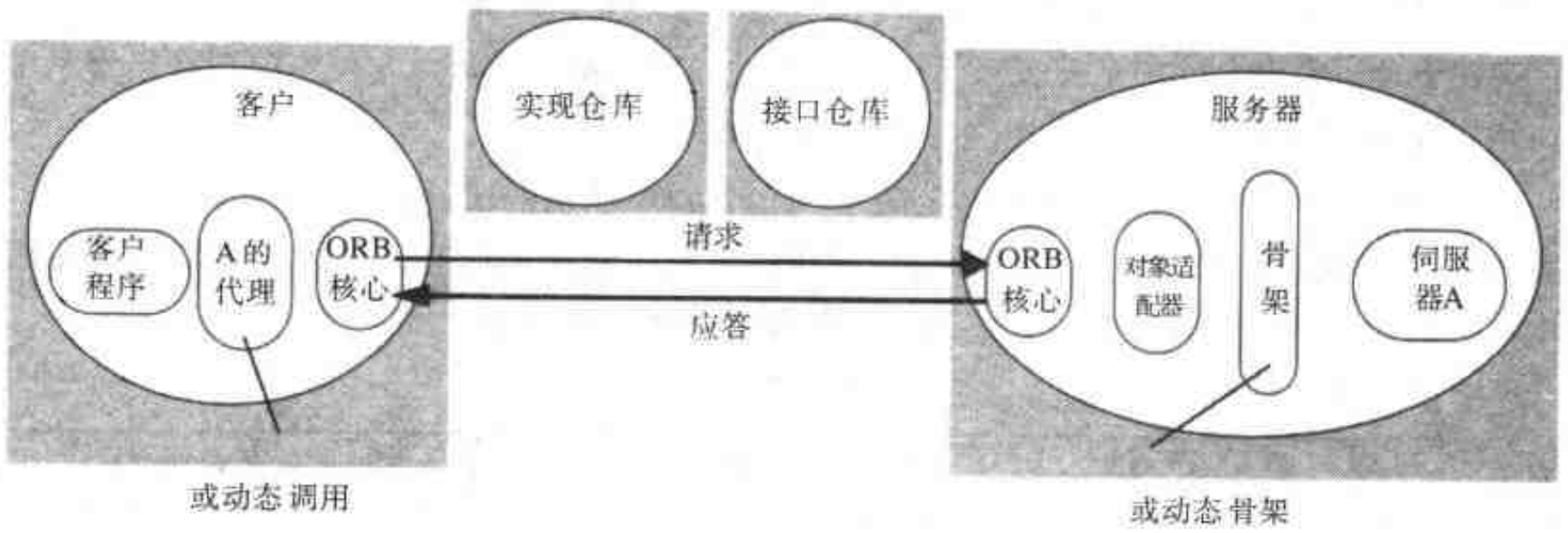
这个接口作为CORBA对象由客户实现，使服务器能在添加新对象的时候发送一个版本号给客户。但是在服务器能这样做之前，客户需要将它的对象的远程对象引用告之服务器。为使这一接口操作成为可能，*ShapeList*接口要求增加额外的方法，例如下面的*register*和*deregister*：

```
int register (in WhiteboardCallback callback);
void deregister (in int callbackId );
```

在客户获得了*ShapeList*对象的一个引用并创建了一个*WhiteboardCallback*的实例之后，它使用*ShapeList*的*register*方法告诉服务器它对接收回调感兴趣。服务器中的*ShapeList*对象负责维护一个感兴趣的客户的列表，并在每次添加新对象而增加版本号时通知这些感兴趣的客户。*callback*方法被声明为*oneway*类型，用以让服务器能使用异步调用，避免因为通知每个客户而带来延迟。

17.2.2 CORBA体系结构

该体系结构设计用于支持对象请求代理，使客户能调用远程对象的方法，同时还允许客户和服务器都能以多种编程语言实现。CORBA体系结构的主要组件如图17-6所示。



677

图17-6 CORBA体系结构的主要组件

该图应该与图5-6对比，这样就会注意到CORBA体系结构包含了3种额外的组件：对象适配器、实现仓库和接口仓库。

CORBA区分了动态与静态调用。在编译时如果知道远程接口，就使用静态调用，以使客户存根和服务器骨架可用。如果在编译时还不知道远程接口，就必须使用动态调用。大多数程序员喜欢使用静态调用，因为它提供了更自然的编程模型。

我们现在讨论CORBA体系结构的组件，将动态调用留到最后考虑。

ORB核心 ORB核心的角色类似于图5-6中的通信模块。另外，ORB核心提供的接口包括如下操作：

- 启动和停止ORB的操作。
- 在远程对象引用和字符串之间进行转换的操作。
- 为采用动态调用的请求提供参数列表的操作。

对象适配器 对象适配器的任务是在具有IDL接口的CORBA对象和对应的伺服类的编程语言接口之间建立一座桥梁。它还包括图5-6中显示的远程引用模块和调度程序模块的功能。对象适配器具有如下的任务：

- 为CORBA对象创建远程对象引用。
- 通过适当的伺服器骨架调度每个RMI。
- 激活对象。

一个对象适配器给每个CORBA对象一个惟一的对象名，这也成为其远程对象引用的一部分。每次都使用相同的名字激活同一个对象。对象名可以由应用程序指定，或者由对象适配器创建。每个CORBA对象都在它的对象适配器中注册，对象适配器维护一个远程对象表，将CORBA对象的名字映射到它们的伺服器对象。

每个对象适配器有它自己的名字，这也成为它管理的所有CORBA对象的远程对象引用的一部分。这个名字既可以由应用程序指定，也可以自动地创建。CORBA 2.2标准中将相应的对象适配器称为可移植的对象适配器（POA）。它之所以被称为是可移植的，是因为它允许应用和伺服器对象运行在由不同开发者创建的ORB之上[Vinoski 1998]。

骨架 骨架类由IDL编译器用服务器方的语言创建。与以前类似，远程方法调用通过恰当的骨架调度给某个特定的伺服类，骨架还解码请求消息中的参数，并编码应答消息中的异常和结果。

客户存根/代理 这些类用客户方的语言编写。不同客户语言的IDL编译器在IDL接口上创建代理类（对于面向对象语言）或者创建一系列存根过程（对于过程化语言）。与以前类似，
678 客户存根/代理对调用请求中的参数进行编码，对应答消息中的结果和异常进行解码。

实现仓库 实现仓库根据需要激活已经注册过的服务器，并负责定位当前正在运行的服务器。用对象适配器名字指向那些要注册和激活的服务器。

一个实现仓库存储从对象适配器的名字到包含对象实现的文件路径名的映射。通常，当安装服务器程序的时候，对象实现和对象适配器名就注册在实现仓库中。当对象实现在服务器中被激活的时候，服务器的主机名和端口号便添加到该映射中。

实现仓库的数据项：

对象适配器名	对象实现的路径名	服务器的主机名和端口号

并不是所有的CORBA对象都需要应请求而激活。有些对象（例如由客户创建的回调对象）就只运行一次，并且在不再需要的时候便停止存在，它们不使用实现仓库。

通常，实现仓库还可以存储关于每个服务器的额外信息，例如关于允许谁激活它或者调用它的操作这类的访问控制信息。为了提供可用性或者容错性，可能需要复制实现仓库中的信息。

接口仓库 接口仓库的任务是将关于已经注册过的IDL接口的信息提供给需要它的客户和服务器。对于一个给定类型的接口，它可以提供方法的名字，并且对于每个方法，它可以提

供参数和异常的名字与类型。这样一来，接口仓库为CORBA添加了一种反射机制。假设一个客户程序接收到一个对新的CORBA对象的远程引用，而且客户还没有代理，那么客户会向接口仓库询问该对象的方法和它们要求的参数类型。

IDL编译器在编译接口时赋予每个IDL类型一个惟一的类型标识，包含在对相应类型的CORBA对象的远程对象引用中。该类型标识被称为仓库ID，因为它可以用作注册在接口仓库中的IDL接口的关键字。

那些使用带有客户存根和IDL骨架的静态（普通）调用的应用不需要接口仓库。并不是所有的ORB都提供接口仓库。

动态调用接口 在有些应用中，没有相应代理类的客户可能需要调用远程对象中的一个方法。例如，浏览器可能需要显示关于分布式系统中各种服务器上可用的CORBA对象的信息。让这样一个程序不得不连接所有这些对象的代理是不可行的，特别是随着时间流逝又会有新的对象添加到系统中。CORBA不允许像Java RMI中那样，在运行时下载代理类。CORBA的解决方法是使用动态调用接口。 679

动态调用接口允许客户动态地调用远程CORBA对象，它用于不便于使用代理的时候。客户可以从接口仓库中获取关于给定CORBA对象中可用方法的必要信息。客户可以利用这一信息构造一个带有合适参数的调用，并将它发送给服务器。

动态骨架接口 它允许CORBA对象接收没有骨架的接口上的调用，该接口没有骨架是因为其接口的类型在编译时还不知道。当动态骨架接收到调用的时候，它检查请求的内容以查找目标对象、待调用的方法以及参数等，然后它就调用这个目标对象。

遗留代码 术语*遗留代码*用来指那些已经存在的代码，但是它们并不是按照分布式对象的想法设计的。通过为遗留代码定义一个IDL接口并提供对象适配器和骨架的一个实现，可以将这些代码整合到CORBA对象中。

17.2.3 CORBA接口定义语言

CORBA接口定义语言IDL提供了定义模块、接口、类型、属性和方法的标记符号的机制。除了模块之外，我们已经在图5-2和图17-1中显示了与上述所有概念相关的例子。IDL具有与C++相同的词法规则，但是它还有一些附加的关键字用以支持分布性，例如*interface*、*any*、*attribute*、*in*、*out*、*inout*、*readonly*、*raises*等。它也允许标准的C++预处理机制。可参见图17-7中用*typedef*定义*All*类型。IDL的语法是ANSI C++的一个子集再加上支持方法标记符号的构造函数。我们在此只给出了IDL的一个简要描述。Baker [1997]、Henning和Vinoski [1999]给出了实用的综述和许多例子。全部完整的规范参见[OMG 1997d]。

IDL模块 模块构造允许接口和其他IDL类型定义按逻辑单元分组。一个模块定义了一个名字作用域，它可以避免定义在一个模块内的名字与定义在模块外的名字发生冲突。例如，接口*Shape*和*ShapeList*的定义可能属于一个称为*Whiteboard*的模块，参见图17-7。

IDL接口 正如我们已看到的，一个IDL接口描述了在实现该接口的CORBA对象中可用的所有方法。CORBA对象的客户程序可能就是根据对CORBA对象的IDL接口的了解而进行开发的。根据对我们例子的研究，读者将会看到一个接口定义了一系列操作和属性，并且通常依赖于一系列与这个接口同时定义的类型。例如，图5-2中的*PersonList*接口定义了一项属性、3

种方法并且依赖于类型*Person*。

```
module Whiteboard {
    struct Rectangle{
    ...} ;
    struct GraphicalObject {
    ...};
    interface Shape {
    ...};
    typedef sequence <Shape, 100> All;
    interface ShapeList {
    ...};
};
```

图17-7 IDL模块*Whiteboard*

IDL方法 方法标记符号的通常格式是：

[*oneway*] <返回类型> <方法名>(参数1，…，参数*L*)

[*raises*(异常1，…，异常*N*)] [*context*(名字1，…，名字*M*)]

其中，方括号中的表达式是可选的。下面是一个只包含必要部分的方法的标记符号：

void getPerson(*in string name, out Person p*) ;

正如17.2节解释的，参数被说明为*in*、*out*或者*inout*类型，其中*in*参数的值从客户传递到被调用的CORBA对象，而*out*参数从被调用的CORBA对象传递给客户，*inout*类型的参数极少使用，该类型表示参数的值可以双向传递。如果没有返回值，可以将返回类型说明为*void*。

可选的*oneway*表达式表示调用方法的客户将不会在目标对象执行该方法时阻塞。此外，*oneway*调用采用的是或许调用语义。我们在17.2.1节看到过下面这样的例子：

oneway void callback(*in int version*) ;

在该例子中，服务器在每次添加一种新图形的时候就调用客户，偶尔丢失一次请求也不会对客户产生影响，因为该调用只表示最新的版本号，而且后续的调用也未必会再丢失。

可选的*raises*表达式表示用户定义的异常，可能因为终止方法的执行而引发这些异常。例如，根据图17-1考虑如下的例子：

exception FullException {};

Shape newShape (*inGraphicalObject g*) *raises* (*FullException*);

*newShape*方法带有*raises*表达式，它可以引发一个称为*FullException*的异常，该异常定义在*ShapeList*接口中。在我们的例子中，异常不包含变量。但是，异常也可以定义为包含变量，例如：

exception FullException { GraphicalObject g} ;

680 ~ 681

当引发了包含变量的异常时，服务器可以使用这些变量将与异常相关的上下文返回给客户。

CORBA也能创建系统异常，包括与服务器相关的问题（例如服务器忙或无法激活），以及与通信和客户相关的问题。客户程序应该处理用户定义的异常和系统异常。可选的*context*表达式用于支持从字符串名到字符串值的映射。参见Baker[1997]对上下文的解释。

IDL类型 IDL支持15种基本类型，包括*short*（16位）、*long*（32位），*unsigned short*、*unsigned long*、*float*（32位）、*double*（64位）、*char*、*boolean*（TRUE，FALSE）、*octet*（8位）和*any*类型（它能表示任何一种基本类型或者构造类型）。大多数基本类型的常量形式和常量字符串都可以使用*const*关键字声明。IDL提供一种特殊的类型，称为*Object*，它的值是远程对象引用。如果一个参数或者结果是*Object*类型，那么对应的参数就可以指向任何CORBA对象。

IDL的构造类型在图17-8中描述，所有这些类型在作为参数或结果时都是按值传递的。所有用作参数的序列和数组必须以*typedef*定义。基本或者构造数据类型都不能包含引用。

属性 除了方法，IDL接口还可以有属性。属性就像Java中的公用类字段。属性可以定义为*readonly*。对CORBA对象来说，属性是私有的，但是对于每个声明过的属性，都会由IDL编译器自动创建一对存取方法，一个是获取属性的值，另一个是设置属性的值。对于*readonly*属性，只能提供获取属性值的方法。例如，定义在图5-1中的*PersonList*接口包括如下的属性定义：

```
readonly attribute string listname;
```

继承 IDL接口可以扩展。例如，如果接口*B*扩展了接口*A*，这意味着它可以在*A*的基础上添加新的类型、常量、异常、方法和属性。一个被扩展的接口可以重新定义类型、常量和异常，但是不允许重新定义方法。扩展类型的值作为父类型的参数或结果的值是合法的。例如类型*B*作为类型*A*的参数或结果的值是合法的。

```
interface A { };
interface B: A{ };
interface C {};
interface Z : B, C {};
```

另外，IDL接口可以扩展不止一个接口。例如，接口*Z*扩展了*B*和*C*。这意味着*Z*具有*B*和*C*的所有组成部分（不包括它重新定义的那些）和它扩展定义的部分。

当一个像*Z*这样的接口扩展了不止一个接口的时候，它就有可能会继承来自两个不同接口却带有相同名字的类型、常量或者异常。例如，假设*B*和*C*都定义了一种类型称为*Q*，那么在*Z*接口中*Q*的使用就是不明确的，除非给出其域名，比如*B*::*Q*或者*C*::*Q*。IDL不允许一个接口从 682
两个不同接口继承带有相同名字的属性或者方法。

所有的IDL接口都继承自*Object*类型，它隐含表示所有的IDL接口都与*Object*类型兼容。这使定义能将任意类型的远程对象引用作为参数处理或者作为结果返回的IDL操作成为可能。命名服务中的*bind*和*resolve*操作就是这种例子。

IDL类型名 17.2.2节提到由IDL编译器为IDL接口中的每种类型创建的惟一类型标识符。例如，*Shape*接口类型的IDL类型可能是：

```
IDL: Whiteboard / Shape:1.0
```

这个例子表明一个IDL类型名有3个部分——IDL前缀、类型名和版本号。开发者可以利用IDL前缀杂注（见Henning和Vinoski [1999]）把一个附加的字符串放在类型名的前面，以区分是他们自己的类型还是其他开发者的类型。接口的IDL类型名包含在远程对象引用中。

对CORBA的扩展 CORBA规范最近又添加了一些新的特征，包括以按值传递方式传递非CORBA对象的能力以及RMI异步变量。Vinoski[1998]在*CACM*论文中对这些新特性都做

了讨论。

类　型	举　例	使　用
序列	*typedef sequence<Shape,100> All;* *typedef sequence<Shape> All* 有界和无界*Shape*序列	定义变长序列类型，序列元素为某种IDL类型。可以指定长度的上界
字符串	*string name ;* *typedef string<8> SamllString ;* 有界和无界字符串序列	定义以空字符结束的字符串序列．可以指定长度的上界
数组	*typedef octet uniqueId*[12]; *typedef GraphicalObject GO*[10][8]	定义多维定长序列类型，序列元素是某种IDL类型
记录	*struct GraphicalObject* { *string type ;* *Rectangle enclosing ;* *boolean isFilled ;* };	定义包含一组相关实体的记录类型。*struct*在参数和结果中按值传递
枚举	*enum Rand* (*Exp, Number, Name*) ;	IDL中的枚举类型将一个类型名映射到一个小的整数集合中
联合	*union Exp switch (Rand)* { *case Exp: string vote;* *case Number: long n;* *case Name: string s;* };	IDL联合允许一个给定的类型集合可以作为一个参数传输。头部由一个*enum*参数化，该*enum*指定正在使用哪种类型

图17-8　IDL构造类型

按值传递的对象　正如我们在上面看到的，IDL参数和结果不论是构造类型还是原始类型，都是按值传递，而那些指向CORBA对象的参数和结果是通过引用传递的。对按值传递非CORBA对象的支持最近被添加到了CORBA规范中[OMG 1998e]。它是通过将一种用以表示非CORBA对象、被称为*valuetype*的类型附加到IDL中来实现的。*valuetype*是一种附加了方法标记符号的结构（就像接口中的那些）。*valuetype*类型的参数和结果是按值传递的，也就是说，状态被传输到远方，并用于在目的地创建一个新的对象。这一新对象的方法可以在本地调用，导致它的状态脱离了原来对象的状态。传递方法的实现不是这么直接，因为客户和服务器可能使用不同的语言。然而，如果客户和服务器都用Java实现的话，那么代码就可以下载。对于C++语言的实现，需要在客户和服务器上提供必要的代码。

异步RMI　CORBA的异步RMI非常适用于客户可能暂时无法联网的环境——例如，客户正在列车上使用膝上电脑。CORBA消息规范[OMG 1998d]提出了一种RMI的替代形式，用于那些容易暂时无法联网的客户。例如，它通过一个中间代理将请求发过去，中间代理会保证请求能抵达并且必要的时候会存储应答。另外，RMI的调用语义有了两个新的变种：回调和轮询。回调是指客户在调用中传输一个回调的引用，这样，服务器可以带有结果回调；轮询是指服务器返回一个能用于轮询或者等待应答的*valuetype*对象。

17.2.4 CORBA远程对象引用

CORBA 2.0规范说明了一种适于应用的远程对象引用的格式，不管该远程对象是否是可激活的。采用这种格式的引用称为互操作对象引用（IOR）。下图基于Henning[1998]的描述，它包括了IOR的详细说明：

IOR格式

IDL接口类型名	协议和地址细节			对象关键字	
接口仓库标识符	IIOP	主机域名	端口号	适配器名称	对象名

- IOR的第一个字段说明CORBA对象的IDL接口的类型名。注意如果ORB有一个接口仓库，这个类型名也就是IDL接口的接口仓库标识符，它允许一个接口的IDL定义在运行时获取。 683 ~ 684
- 第二个字段说明传输协议以及该协议用以识别服务器的具体要求。举例来说，因特网ORB互操作协议（IIOP）使用的是TCP/IP，因此，服务器地址由一个主机域名和一个端口号组成。
- 第三个字段由ORB使用，用于标识一个CORBA对象。它包括服务器中对象适配器的名称和由对象适配器指定的CORBA对象的对象名。

CORBA对象的*暂态IOR*持续的时间只和它们的宿主进程一样长，而*持久IOR*可以在CORBA对象激活期间持续。暂态IOR包含了CORBA对象的宿主服务器的具体地址，而持久IOR包含了实现仓库的具体地址。在这两种情形中，客户ORB都发送请求消息给在IOR中给出了具体地址的服务器。我们现在讨论两种情形下IOR如何定位代表CORBA对象的伺服器。

- 暂态IOR　服务器ORB的核心接收到包含对象适配器名和目标对象名的请求消息。它利用对象适配器名定位到对象适配器，然后再由对象适配器使用对象名定位到伺服器。
- 持久IOR　实现仓库接收到请求，然后，从请求的IOR中抽取出对象适配器名。假设IOR表中有对象适配器的名称，如果必要的话它会尝试激活指定的主机地址上的CORBA对象。一旦CORBA对象被激活，实现仓库就将它的地址细节返回给客户ORB，客户ORB将它们用作RMI请求消息的目标（请求消息包括对象适配器名和对象名）。这些使服务器ORB核心可以定位对象适配器，和前面一样，对象适配器再利用对象名定位伺服器。

在IOR格式中，考虑到对象或者实现仓库可能被复制到几个不同的地方，所以，指定主机域名和端口号的第二个域可以重复多次。

请求－应答协议中，应答消息包括了头信息，头信息用于使持久IOR的过程能被执行。特别是，它包括一个状态项，能标识出请求是否应该被转发到另外的服务器上，在此情形下，应答消息的消息体会包括一个IOR，该IOR中包含了最近被激活的对象的服务器地址。

17.2.5 CORBA语言映射

从例子我们已经看到，从IDL的类型映射到Java类型是非常直接的。IDL中的基本类型映射到Java中相应的基本类型。*struct*、*enum*和*union*映射为Java中的类，IDL中的序列和数组映射为Java中的数组。一个IDL异常映射为一个Java类，该类提供具有异常字段的实例变量和构造函数。IDL到C++的映射也同样地直接。 685

但是，我们也已经看到，在将IDL的参数传递语义映射到Java时出现了一些困难。尤其是，IDL允许方法通过输出参数返回几个独立的值，而Java只能返回一个结果。Java为此提供了*Holder*类来克服这一差异，但是这就要求程序员去使用它们，它们的使用并不是很直接。例如，图5-2中的*getPerson*方法用IDL定义成如下形式：

void getPerson (in string name , out Person p);

而在Java接口中与之等价的方法被定义为：

void getPerson (String name, PersonHolder p) ;

并且客户必须提供一个*PersonHolder*的实例作为它调用的参数。*Holder*类有一个实例变量，它保存参数的值，以便客户在调用返回时访问该值。*Holder*类也具有在客户和服务器之间传输参数的方法。

虽然CORBA的C++实现能非常自然地处理*out*和*inout*参数，但是，C++程序员却面临着由参数引起的与存储管理相关的一系列问题。这些困难出现在将对象引用和变长实体如字符串或者序列等作为参数传递的时候。

例如，在Orbix[Baker 1997]中，ORB负责远程对象的引用计数和代理，当不再需要对象时释放它们。它提供给程序员释放或者复制对象的方法。一旦一个服务器方法执行完毕，*out*参数和结果就被释放，如果还需要它们，程序员就必须复制它们。例如，一个实现*ShapeList*接口的C++伺服器需要复制由*allShapes*方法返回的引用。当不再需要对象时，传递给客户的对象引用必须释放。类似的规则也适用于变长的参数。

通常，使用IDL的程序员不仅必须学习IDL自身的表示法，还必须理解它的参数怎样映射到实现所用语言的参数上。

17.3 CORBA服务

CORBA包括分布式对象可能需要的服务的规范。特别地，命名服务是对任何ORB的基本补充，正如我们在本节的编程例子中所看到的。关于所有这些服务的索引参见OMG的Web站点[www.omg.org]。Orfali等[1996]对CORBA服务进行了描述，这些服务包括：

- 命名服务　CORBA命名服务具体见17.3.1节。
- 事件服务和通知服务　CORBA事件服务在17.3.2节中讨论，通知服务见17.3.3节。
- 安全服务　CORBA安全服务在17.3.4节中讨论。
- 交易服务　与命名服务允许CORBA对象按名字定位相反，交易服务允许CORBA对象按属性定位——它是一种目录服务。它的数据库包括一个从服务类型和它们相关的属性到CORBA对象的远程对象引用的映射。服务类型是一个名字，每个属性都是一个名-值对。客户通过说明所要求的服务类型对属性值的约束以及匹配项的优先顺序来进行查询。交易服务器可以形成联盟，以这种方式它们不仅可以使用自己的数据库，而且还可以代表另一个客户执行查询。交易服务更详细的描述参见Henning和Vinoski[1999]。
- 事务服务和并发控制服务　对象事务服务允许分布式CORBA对象参与到平面或嵌套的事务中。客户将事务描述成一系列RMI调用，它以*begin*开始，由*commit*或者*rollback*（*abort*）终止。ORB将事务标识符附加到每个远程调用上，并处理*begin*、*commit*和*rollback*（*abort*）请求。客户也可以挂起和恢复事务。事务服务执行两阶段提交协议。

686

并发控制服务把锁应用到访问CORBA对象的并发控制中。它可以在事务内部使用，或者独立地使用。

- 持久对象服务　5.2.5节解释了持久对象能在它们不用的时候在持久对象存储中以被动的形式存储，而在需要的时候激活。虽然ORB激活带有持久对象引用的CORBA对象，并根据实现仓库取得对象的实现，但是ORB并不负责保存和恢复CORBA对象的状态，而是由每个CORBA对象自己保存和恢复它自己的状态，例如通过文件的形式保存。CORBA持久对象服务（POS）就是设计用于CORBA对象的持久对象存储。该规范定义了它的组件接口。POS能以各种不同的方式实现，例如，通过文件或者通过一个关系数据库系统或者面向对象数据库系统的方式实现。

POS体系结构考虑到一系列可用的数据存储——每个持久对象都有一个持久标识符，该标识符包括它的数据存储标识符和在数据存储中的对象编号。通过发送一个请求到POS再由POS转发到合适的数据存储，客户或者CORBA对象可以决定什么时候保存它的状态。为了让CORBA对象的客户控制它的持久性，该CORBA对象要继承了一个包括保存、恢复和删除它的接口。

持久CORBA对象的实现必须选择带有数据存储的通信协议。例如，可以将它的数据通过流的方式传递。POS规范提议了完成这项任务的若干备选协议。例如，直接访问协议允许CORBA对象访问数据存储中的持久对象的属性。在此情形中，CORBA对象的实现者采用了类似IDL的数据定义语言来定义在它的持久状态中的属性。 687

17.3.1 CORBA命名服务

CORBA命名服务是第5章描述的绑定程序的一个复杂的例子。它在名字上下文范围内将名字绑定到CORBA对象的远程对象引用。

正如9.2节解释的，名字上下文是一系列名字应用的范围——在上下文中的每个名字必须是惟一的。名字可能与一个应用中的CORBA对象的对象引用相关，或者与命名服务中的另一个上下文相关。上下文也可能嵌套在一起，提供一种层次的名字空间，如图17-9所示，其中CORBA对象以通常的方式显示，而名字上下文以无色椭圆显示。左边的图形显示了17.2.1节编程实例中描述的*ShapeList*对象的入口。

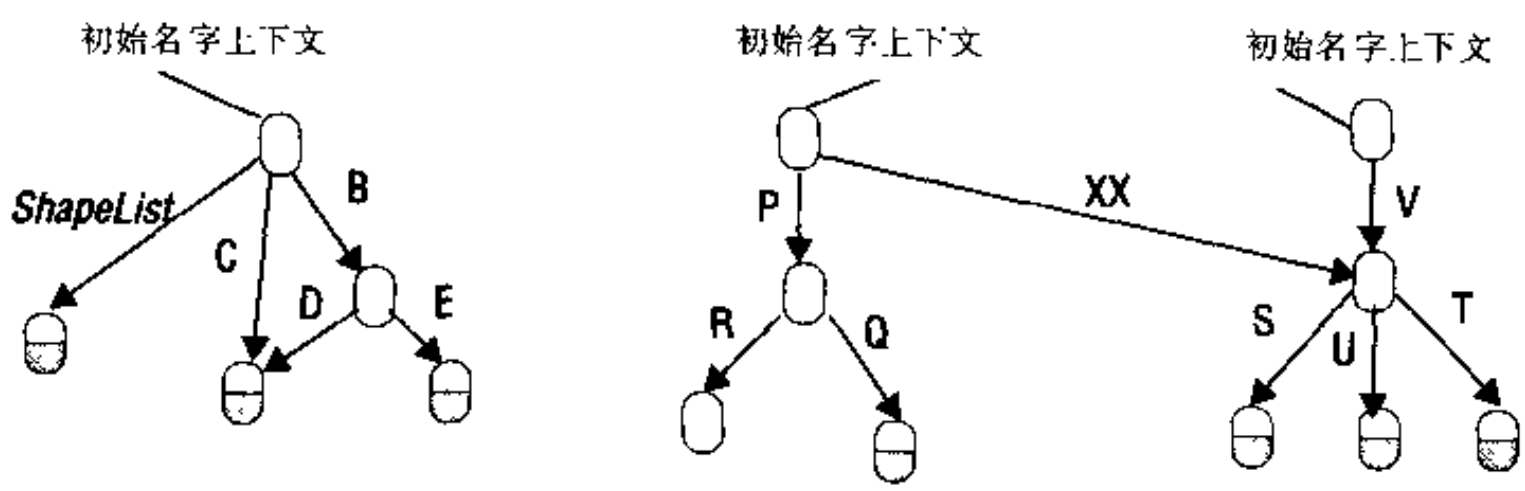


图17-9　CORBA命名服务中的名字图

初始名字上下文提供了用于一系列绑定的根。注意可能不止一个初始名字上下文会指向同一个名字图。实际上，ORB的每个实例都有一个初始名字上下文，但是与不同ORB相关的命名服务器可以形成联盟，这在本节的后面将会介绍到。客户和服务器程序需要从ORB获取

初始名字上下文，通过调用ORB的方法*resolve_initial_references*，并以“*NameService*”作为参数给出，参见图17-5。ORB返回一个*NamingContext*类型对象的引用（参见图17-10），它指向那个ORB的命名服务器的初始上下文。因为可能会有好几个初始上下文，所以对象没有绝对的名字——名字总是相对于一个初始上下文来解释的。

带有一个或者多个成分的名字可以在任意名字上下文中开始进行解析。为了解析一个具有几个成分的名字，命名服务在起始的上下文中查找与第一个成分相匹配的绑定。如果存在一个匹配的绑定，则它要么是一个远程对象引用，要么是到另一个名字上下文的引用。如果结果是一个名字上下文，那么名字的第二个成分就在那个上下文中解析。这个过程不断重复，直到名字的所有成分都已经被解析，并且最终取得了一个远程对象引用，除非中途匹配失败。

```
struct NameComponent { string id; string kind; };
typedef sequence <NameComponent> Name;
interface NamingContext {
    void bind (in Name n, in Object obj);
        在该上下文中，将给定的名字和远程对象引用绑定在一起
    void unbind (in Name n);
        按给定名字删除一个已存在的绑定
    void bind_new_context(in Name n);
        创建一个新的名字上下文，并将它与上下文中一个给定的名字绑定在一起
    Object resolve (in Name n);
        在上下文中查找该名字，并返回它的远程对象引用
    void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
        返回该上下文里绑定中的名字
};
```

图17-10 IDL中CORBA命名服务的*NamingContext*接口部分

CORBA命名服务使用的名字是由两部分组成的名字，称为名字成员，其中每个名字成员包括两个字符串，一个是名字，另一个是对象的种类。种类字段提供一个为应用所采用的属性，并且可能包含一些有用的描述信息；它不由命名服务解释。

尽管CORBA对象由命名服务给出了层次的名字，但是这些名字不能表示为像UNIX文件那样的路径名。所以，在图17-9中，我们不能用/V/T表示指向最右边的对象。这主要是因为名字可能包括任意字符，它排除了具有分隔符的可能性。

图17-10给出了CORBA命名服务的*NamingContext*类提供的主要操作，它们是用CORBA IDL语言定义的。完整的规范可以从OMG[1997b]处获得。为了简单起见，图17-10没有给出由方法抛出的异常。例如，*resolve*方法可能抛出*NotFound*异常，而*bind*方法可能抛出*AlreadyBound*异常。

客户使用*resolve*方法按名字查找对象引用，它的返回类型是*Object*，因此它可以返回属于应用的任意类型对象的引用。返回结果必须先进行类型缩小，然后才能用于调用一个应用程序的远程对象的方法，参见图17-5。*resolve*的参数是*Name*类型的，它定义为一个名字成员的序列。这意味着客户在调用之前必须构造一个名字成员的序列。图17-5给出了一个客户，它

创建了一个称为*path*的数组，里面包含了一个单独的名字成员，并将它用作*resolve*方法的参数。这似乎并不是一种很方便使用正常路径名的方法。

远程对象的服务器采用*bind*操作注册对象的名字，采用*unbind*操作删除它们。*bind*操作将一个给定的名字与远程对象引用绑定在一起，可以在进行绑定操作的上下文中调用它。参见图17-4，该图中，名字*ShapeList*被限制在初始名字上下文中。在该例子中，如果在调用*bind*操作的时候，所采用的名字已经有一个绑定的话，就会抛出一个异常，而*rebind*操作是允许绑定被替代的，所以这时可采用*rebind*方法。 688~689

*bind_new_context*操作用于创建一个新的上下文，并将它与原上下文中的一个给定的名字绑定。另一种称为*bind_context*的方法将一个给定的名字上下文与原上下文中的给定名字绑定。*unbind*方法可以删除名字，也可以删除上下文。

*list*操作用于浏览命名服务一个上下文中可用的信息。它从目标*NameContext*返回一张绑定的列表。每个绑定包括一个名字和类型——类型表示它是一个对象还是一个上下文。有时，一个名字上下文可能包含非常多的绑定，在此情形下，*list*操作可能不愿意将它们全体都作为一次单独调用的结果返回。因为这种原因，*list*方法将最多的绑定作为一次*list*调用的结果返回，如果还要求发送更多绑定的话，它就组织分批返回结果。这可以通过将一个迭代子作为第二次结果返回来实现。客户使用迭代子以一次一部分的方式读取剩下的结果。

图17-10给出了*list*方法，但是为了简单起见，省略了其中的参数类型定义。*BindingList*类型是一个绑定的序列，序列的每个元素包括一个名字和它的类型——表示它是一个上下文或是一个远程对象引用。*BindingIterator*类型提供了一个*next_n*方法，用于访问它的下一个绑定的集合；它的第一个参数说明需要多少绑定，第二个参数接收绑定的序列。客户调用*list*方法时，以马上要得到的最大数量的绑定数目作为第一个参数，获得的绑定序列放在第二个参数，第三个参数是一个迭代子，它能用于获取剩下的绑定（如果还有的话）。

CORBA名字空间考虑到了命名服务的联盟，采用的机制是每个服务器在这张名字图中提供一个子集。例如，在图17-9中，中间的图和右边图中的初始名字上下文由不同的服务器管理。中间的图有一个标记为“XX”的绑定，指向右边图中的上下文，通过该绑定，客户可以访问在远程图中命名的对象。为了完成该联盟，右边的图需要添加一个绑定到中间图上的结点。一个组织可以通过提供远程命名服务器和指向它们的远程引用来提供对其名字空间中一些或者全部上下文的访问。

CORBA命名服务的Java实现非常简单，并被称为是透明的，因为它将其所有绑定存储在易失存储器中。任何认真的实现都至少会把它的名字图的拷贝保留在文件中。正如我们在DNS研究中看到的，复制可以用于提供更好的可用性。

17.3.2 CORBA事件服务

CORBA事件服务规范定义了许多接口，用于允许兴趣对象（称为提供者）传递通知给订阅者（称为消费者）。通知可以作为一般同步CORBA远程方法调用的参数或者结果传输。通知的传播方式既可以是由提供者推给消费者的方式，也可以由消费者从提供者拉的方式。在第一种方式中，消费者实现*PushConsumer*接口，该接口包括一个*push*方法，它可以将任意CORBA数据类型作为参数。消费者在提供者上注册它们的远程对象引用。提供者调用*push*方法，将通知作为参数传递。在第二种情形中，提供者实现*PullSupplier*接口，该接口包括一个 690

*pull*方法，可以将任意的CORBA数据类型作为其返回值接收。提供者在消费者上注册它们的远程对象引用。消费者调用*pull*方法并接收到一个作为结果的通知。

通知本身作为*any*类型的参数或结果传输，这意味着对象交换通知必须有一个关于通知内容的协定。然而，应用程序员可以定义自己的IDL接口，可以带有任意类型的通知。

事件通道是可以用于允许多个提供者与多个消费者以异步方式进行通信的CORBA对象。事件通道充当提供者与消费者之间的缓冲区。它也能给多个消费者组播通知。通过事件通道的通信既可以用推方式，也可以用拉方式。这两种方式也可以混合；例如，提供者可以将通知推到通道，再由消费者从通道拉出它们。

当分布式应用需要使用异步通知的时候，它创建一个事件通道，该通道是一个CORBA对象，它的远程对象引用可以通过命名服务或者以RMI的方式提供给应用组件。应用程序中的提供者从事件通道获得代理消费者并传输远程对象引用给代理消费者，从而连接提供者与代理消费者，这样，提供者就可以被订阅了。应用程序中的消费者从事件通道获得代理提供者从而将消费者与它们相连，这样，消费者就可以订阅通知了。代理提供者和代理消费者既可以用于推方式，也可以用于拉方式中。当提供者采用交互的推方式创建一个通知的时候，它调用推代理消费者的*push*方法。通知通过通道传输并传送给代理提供者，然后传递给消费者，如图17-11所示。如果消费者采用了交互的拉方式、那么，消费者将调用拉代理消费者的*pull*方法。

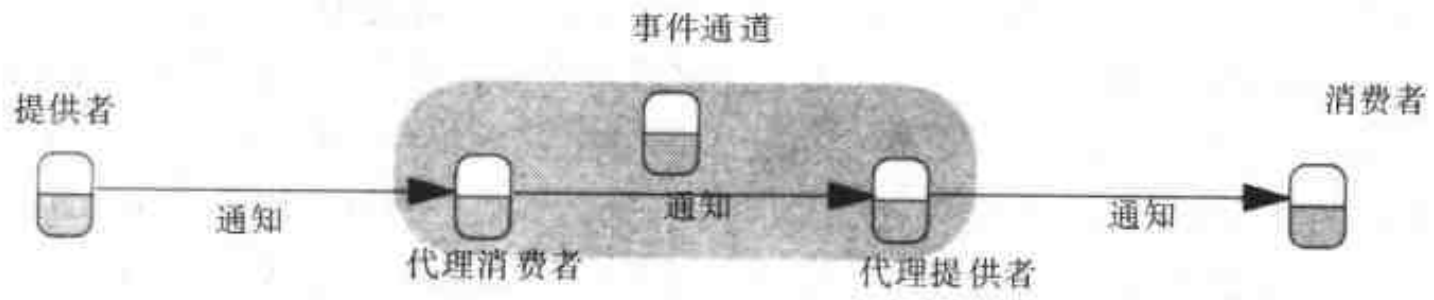


图17-11 CORBA事件通道

代理消费者和代理提供者的存在使构造事件通道链成为可能。在事件通道链中，每个通道提供将被后续通道消费的通知。CORBA模型中的事件通道与图5-10中定义的观察者很相似。用程序实现事件通道的话，可以实现一些5.4节中讨论的观察者的角色。然而，通知不带有任何形式的标识符，因此模式匹配或者通知过滤就需要基于出应用放在通知中的类型信息。

691

Farley[1998]给出了CORBA事件服务的较完整的解释和主要接口概述。CORBA事件服务的完整规范在[OMG 1997c]里。然而，规范没有声明如何创建一个事件通道和如何请求需要的可靠性。

17.3.3 CORBA通知服务

CORBA通知服务[OMG 1998c]保留了CORBA事件服务的所有特征，包括事件通道、事件消费者和事件提供者，同时扩展了该服务。事件服务不提供对过滤事件或者说明传输需求的支持。如果不使用过滤器的话，所有与通道相连的消费者都不得不接收相同的通知。另外如果没有说明传输需求的能力，则所有通过通道发送的通知的传输保证都是固定在实现中的。

通知服务增加了如下一些新的功能：

- 通知可以定义为数据结构。这增强了事件服务中通知的有限能力，事件服务中，通知的

类型只能是*any*或者是由应用程序员说明的一种类型。

- 事件消费者可以使用过滤器准确地说明它们对哪些事件感兴趣。过滤器可以连接到通道中的代理。代理将根据在过滤器中对每种通知内容说明的约束，把通知转发给事件消费者。
- 事件提供者具有一种发现消费者对哪些事件感兴趣的手段。这使它们可以只创建那些满足消费者需求的事件。
- 事件消费者可以发现在一个通道中由提供者供应的事件类型，这使它们可以及时订阅新出现的事件。
- 可以配置通道、代理或者某个事件的属性。这些属性包括事件传输的可靠性、事件的优先级、排序需求（例如FIFO或者以优先级排序）以及放弃存储事件的策略。
- 可以额外增加事件类型仓库。它将提供对事件结构的访问，使得定义过滤约束变得非常方便。

通知服务引入了结构化事件这种数据结构，用这种数据结构可以映射大量不同类型的通知。过滤器可以按结构化事件类型的组成成分进行定义。一个结构化事件包括一个事件头和一个事件体。事件头包括：

域类型	事件类型	事件名	需　求
“home”	“burglar alarm”	“21 Mar at 2pm”	“priority”，1000

域类型指定义域（例如，“finance”、“hotel”或者“home”）。事件类型对域内事件的类型进行惟一地分类（例如，“stock quote”、“breakfast time”、“burglar alarm”）。事件名惟一地标识正在传输的事件的特定实例。事件头的剩下部分包括一系列<名，值>对，可以设计用于说明关于事件传输的可靠性和其他需求。 692

结构化事件体包括如下信息：

可过滤部分			
名，值	名，值	名，值	剩下部分
“bell”，“ringing”	“door”，“open”	“cat”，“outside”	

事件体的第一部分包括一系列能用于过滤器的<名，值>对。不同的行业领域希望能定义相应标准应用于事件体的可过滤部分中的<名，值>对——即在定义过滤器的时候使用相同的名字和值。或许，当夜贼警报响起的时候，该事件会包括警铃的状态、前门是否是打开的和猫在哪儿。事件体的剩下部分设计用于传递与特定事件相关的数据；例如，当夜贼警报响起的时候，它可能包括一个屋内的数码照片。

过滤器对象由代理使用，用于决定是否转发每个通知。一个过滤器设计为一个约束的集合，每个约束是一个具有两个成员的数据结构：

- 一个数据结构的列表，列表中的每个结构由域名和事件类型组成，表示一种事件类型，例如，“home”，“burglar alarm”。该列表包括约束应用到的所有事件类型。
- 一个含有一个布尔表达式的字符串，该表达式涉及上述事件类型的值。例如：

```
( “domain type” == “home” && “event type” == “burglar alarm” )&&
        ( “bell” ! = “ringing” || “door” == “open” )
```

我们的例子使用了不规范的语法。通知服务规范包括约束语言的定义，它是用于交易服务的约束语言的扩展。

17.3.4 CORBA安全服务

CORBA安全服务[Blakly 1999，Baker 1997，OMG 1998b]包括如下内容：

- 主体（用户和服务器）的认证；为主体创建证书（即声明权限的证书）；支持7.2.5节中描述的证书的委托。
- 访问控制可以应用到CORBA对象接收远程方法调用的时候。访问权限可以在访问控制列表（ACL）中说明。
- 服务器审计远程方法调用。
- 不可抵赖的机制。当一个对象代表一个主体执行一次远程调用时，服务器创建并存储一些证书，证明该调用已经由服务器代表请求主体完成过。

693

为了保证安全性正确地应用到远程方法调用中，安全服务要求ORB的协作。为了完成一次安全的远程方法调用，要在请求消息中发送客户的证书。当服务器接收到一个请求消息时，它就要验证客户的证书，检查它们是否是新的、是否是由可接受的权威签署的。如果这些证书是有效的，它们就可以用于判断该主体是否有采用请求消息中的方法访问远程对象的权限。作出这一决定要查询一个对象，该对象包含了关于目标对象上的每个方法（可能是以ACL的形式）都允许由哪个主体访问的信息。如果客户有足够的权限，就执行调用，并将结果返回给客户，如果需要的话，也可以同时返回服务器的证书。目标对象还可以将调用的细节记录在审计日志中或者存储在不可抵赖的证书中。

CORBA允许根据需求说明各种不同类型的安全策略。消息保护策略声明客户或者服务器（或者二者皆有）是否必须被认证以及消息是否必须被保护以免泄露和/或修改。还可以指定审计和不可抵赖的策略；例如，一个策略可以声明它们应该应用哪些方法和参数。

访问控制考虑到，许多应用有大量的用户和更大量的对象，每个对象都有它自己的方法集。根据用户的不同角色，为他们提供了一类特别的称为特权的证书。将对象分组成若干域，每个域有一个的访问控制策略，规定具有某些特权的用户对该域中对象的访问权限。考虑到各种各样难以预见的方法，将每种方法都分类为4种通用方法（*get*、*set*、*use*和*manage*）中的一种。*get*方法只返回对象状态的一部分，*set*方法修改对象的状态，*use*方法导致对象做一些工作，而*manage*方法完成那些不允许一般性使用的特定功能。因为CORBA对象有各种不同的接口，所以必须根据上面的通用方法为每个新接口说明其访问控制权限。这牵涉到那些正致力于访问控制应用、设置适当的特权属性（例如，组或者角色）并帮助用户为完成他们的任务获取适当特权的应用设计者。

在最简单的形式中，可以采用对应用透明的方式使用安全性。它包括将要求的保护策略应用到远程方法调用和审计中。安全服务允许用户获得他们各自的证书和特权，以提供像口令之类的认证数据。

694

17.4 小结

CORBA的主要组件是对象请求代理或者称作ORB，它允许以某一种语言编写的客户可以调用以另一种语言编写的远程对象（称为CORBA对象）中的操作。CORBA还解决了下列异

构性问题：

- CORBA通用ORB间协议（GIOP）包括一种称为CDR的外部数据表示，它使客户和服务器实现与硬件无关的通信成为可能。它也规定了远程对象引用的标准格式。
- GIOP还包括一个与下层操作系统无关的请求－应答协议的操作规范。
- 因特网ORB间协议（IIOP）在TCP/IP协议之上实现请求－应答协议。IIOP远程对象引用包括服务器的域名和端口号。

CORBA对象实现IDL接口中的操作。关于如何访问一个CORBA对象，客户所需要知道的全部只不过是那些在CORBA对象的接口中可用的操作。客户程序通过代理或者存根访问CORBA对象，它们是根据客户方语言编写的IDL接口自动创建的。CORBA对象的服务器框架根据客户方语言编写的IDL接口自动创建的。对象适配器是CORBA服务器的重要组件，它的任务是创建远程对象引用，并将请求消息中的远程对象引用与CORBA对象的实现联系在一起。

CORBA体系结构允许CORBA对象在需要的时候被激活，要达到这一点需要借助一个称为实现仓库的组件，它维护一个实现的数据库，并以它们的对象适配器名作为索引。当一个客户调用一个CORBA对象时，它能在必要的时候被激活以执行该调用。

接口仓库是一个以仓库ID为索引的IDL接口定义数据库，仓库ID包括在IOR中。为了支持动态方法调用，可以通过接口仓库获得与CORBA对象接口中的方法有关的信息。

CORBA服务在RMI之上提供分布式应用需要的有关功能，以便应用可以使用额外的服务如命名服务和目录服务、事件通知、事务或者安全等。 695

练习

17.1 任务包是一个对象，存储（关键字和值）对。关键字是一个字符串，值是一个字节序列。它的接口提供如下的远程方法：

- *pairOut* 带有两个参数，客户通过这两个参数说明将被存储的关键字和值。
- *pairIn* 它的第一个参数让客户说明将从任务包中删除的（关键字和值）对中的关键字，任务对中的值则通过第二个参数提供。如果没有相匹配的对，就抛出一个异常。
- *readPair* 除了把（关键字和值）对留在任务包之外，其他与*pairIn*相同。

采用CORBA IDL定义任务包的接口。定义一个异常，一旦一个操作无法执行时，就抛出该异常。所定义的异常应该返回一个用来标识问题序号的整数和一个描述问题的字符串。任务包接口应该定义一个单独的属性，给出包中任务的数目。

17.2 定义方法*pairIn*和*readPair*的另一种符号标记，它的返回值没有相匹配的任务对可用。该返回值应该定义为一种枚举类型，它的值可以是*ok*和*wait*。讨论两种方法的优点。你会用哪种方法标识类似于关键字中包含了非法字符这样的错误？

17.3 任务包接口中的哪些方法可以被定义为*oneway*操作？给出一个关于*oneway*方法的参数和异常的通用规则。以什么方式可以使*oneway*关键字的意义与IDL其他部分的不同？

17.4 IDL的*union*类型可以用作一种参数，在少数几个类型中只传输一个。采用*union*定义一种参数的类型，该类型可以为空的，也可以为值类型。

17.5 在图17-1中，类型All定义为一个固定长度的序列。把它重新定义为一个相同长度的数组。给出一些关于在一个IDL接口中应该选择数组还是序列的建议。

17.6 任务包设计本意用于协作的客户，一些客户添加任务对（描述任务），而另一些客

户则把任务对删除（并执行所描述的任务）。当一个客户被通知说没有相匹配的任务对可用的时候，它就不能继续它的工作，直到出现一个可用的任务对。定义一个适用于这种情形的回调接口。

17.7 描述对任务包接口的必要修改，使得可以使用回调。

696

17.8 任务包接口中方法的哪些参数是按值传递的，哪个参数通过引用传递？

17.9 使用Java IDL编译器处理你在练习17.1中定义的接口。在所生成的IDL接口的Java等价类中检查*pairIn*和*readPair*方法的符号标记定义。再看看为用于方法*pairIn*和*readPair*的值参数的*Holder*方法而创建的定义。请给出一个例子，说明客户将如何调用*pairIn*方法，解释它将怎样通过第二个参数得到返回值。

17.10 给出一个例子，说明Java客户将如何访问用于给出任务包对象中任务个数的属性。一个属性在哪些方面与一个对象的实例变量不同？

17.11 解释为什么通常情况下远程对象的接口，特别是CORBA对象不提供构造函数。解释如何在没有构造函数的情况下创建CORBA对象。

17.12 根据练习17.1用IDL重新定义任务包接口，以使它可以使用*struct*表示一个任务对，该结构包括一个关键字和一个值。注意，不必使用*typedef*定义*struct*。

17.13 从伸缩性和容错的角度，讨论实现仓库的功能。

17.14 为使CORBA对象可以从一个服务器迁移到另一个服务器，需要进行哪些扩展？

17.15 讨论CORBA命名服务中的两部分名字或*NameComponent*的优点和缺点。

17.16 给出一个算法，描述在CORBA命名服务中如何解析一个多部分名字。一个客户程序需要解析一个带有“A”、“B”和“C”组成部分的多部分名字。在命名服务中，怎样说明resolve操作的参数？

17.17 一个虚拟企业包括一系列相互合作的公司，大家共同完成某个项目。每个公司都希望提供给其他公司只与该项目有关的CORBA对象的访问权限。说明一种合适的方法，可以让整个集团联合它们的CORBA命名服务。

17.18 在共享白板应用程序中，讨论如何使用CORBA事件服务中直接连接的提供者和消费者。用IDL定义*PushConsumer*和*PushSupplier*接口如下：

```
interface PushConsumer {
   void push( in any data ) raises (Disconnected) ;
   void disconnect_push_consumer ( ) ;
}
interface PushSupplier {
   void disconnect_push_supplier();
}
```

通过调用*disconnect_push_supplier()*或者*disconnect_push_consumer()*，提供者和消费者都可以

697 分别决定终止事件通信。

17.19 根据你在17.18中的解决方案，说明如何将一个事件通道插入到提供者和消费者之间，事件通道具有如下的IDL接口：

```
interface EventChannel {
    ConsumerAdmin for_consumers();
```

```
        SupplierAdmin for_suppliers();
    };
```

其中，接口*SupplierAdmin*和*ConsumerAdmin*可以使提供者和消费者得到通过如下IDL定义的代理：

```
interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ---
};
interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    ---
};
```

下面是用IDL定义的代理消费者和代理提供者的接口：

```
interface ProxyPushConsumer : PushConsumer{
    void connect_push_supplier (in PushSupplier supplier)
        raises (AlreadyConnected);
};
interface ProxyPushSupplier : PushSupplier{
    void connect_push_consumer (in PushConsumer consumer)
        raises (AlreadyConnected);
};
```

698

使用事件通道具有什么好处?

第18章　Mach实例研究

18.1　简介
18.2　端口、命名和保护
18.3　任务和线程
18.4　通信模型
18.5　通信实现
18.6　内存管理
18.7　小结

本章介绍Mach微内核系统的设计。Mach系统可以运行在多处理器系统上，也可以运行在通过网络连接的多个单处理器的计算机系统上。它的设计初衷是希望新的分布式系统在保持与UNIX系统兼容的同时，能不断发展。

Mach的设计具有创造性。特别值得一提的是，其中集成了精心设计的进程间通信机制和虚拟内存管理方案。我们将介绍Mach的体系结构及其主要的抽象机制：线程和任务（进程）、端口（通信的末端）以及虚拟内存，其中包括它在任务间进行高效通信所承担的角色和通过计算机间的分页来提供对对象共享的支持。

18.1　简介

Mach项目[Acetta *et al.* 1986，Loepere 1991，Boykin *et al.* 1993]从开始到1994年一直是在美国的卡内基－梅隆大学研究的。随后，在那儿被继续改进为具有实时内核的系统[Lee *et al.* 1996]、而与此同时，犹他大学和开放软件基金会的一些小组也在继续着Mach的研发工作。Mach项目有两个后续项目：RIG[Rashid 1986]和Accent[Rashid and Robertson 1981，Rashid 1985，Fitzgerald and Rashid 1986]。RIG是罗彻斯特大学在20世纪70年代开发的，Accent是由卡内基－梅隆大学在20世纪80年代前半期开发的。相对于RIG和Accent这两个后续系统，Mach项目从来就没有计划开发出一个完整的分布式操作系统。相反，Mach内核可以直接与BSD UNIX兼容。它为UNIX提供增强的内核功能，以便使UNIX的实现系统可以扩展到多处理器系统上或者多个单处理器计算机组成的网络上。从一开始，Mach设计者就希望大多数UNIX可以像用户进程一样实现。

尽管设计意图是这样的，然而作为Mach两个主要发行版本之一的Mach 2.5还是在它的内核中包含了所有的UNIX兼容代码。它可以在SUN-3s、IBM RT PC、多处理器和单处理器的VAX系统以及Encore Multimax和Sequent多处理器系统上运行。从1989年开始，Mach 2.5集成为OSF/1的基本部分，而开放软件基金会正是用OSF/1与UNIX的业界标准版本UNIX System V的第4版相抗衡。更老一些的一个Mach版本被用作NeXT工作站的操作系统的基础。

3.0版的Mach内核删除了UNIX代码，这才是我们要描述的Mach版本。最近，Mach 3.0成为了MkLinux系统实现的基础，而MkLinux系统是运行在Power Macintosh 计算机[Morin 1997]上的Linux操作系统之一。3.0版的Mach内核也可以运行在基于Intel x86的PC机上；也可

以运行在DEC station 3100和5000系列工作站、一些基于Motorola 88000的工作站和SUN SPARC Station工作站上；它还为IBM RS6000、HP的Precision Architecture以及DEC的Alpha服务器预留了端口。

Mach 3.0是构造操作系统的用户级仿真系统、数据库系统、语言运行时系统以及其他能被称为子系统的系统软件（图18-1）的基础。

对传统操作系统的仿真可以使为之开发的二进制应用程序能够执行。另外，程序员可以开发出适用于这些传统操作系统的新应用程序。同时，人们还可以开发中间件和应用程序，以充分利用分布式带来的好处；并且传统操作系统的实现也可以是分布式的。在操作系统仿真中有两个主要问题。首先，分布式的仿真不可能完全精确，这是因为在分布的情况下会产生新的错误模式。其次，仍然存在能否获得适用于广泛应用的可接受的性能的问题。

700

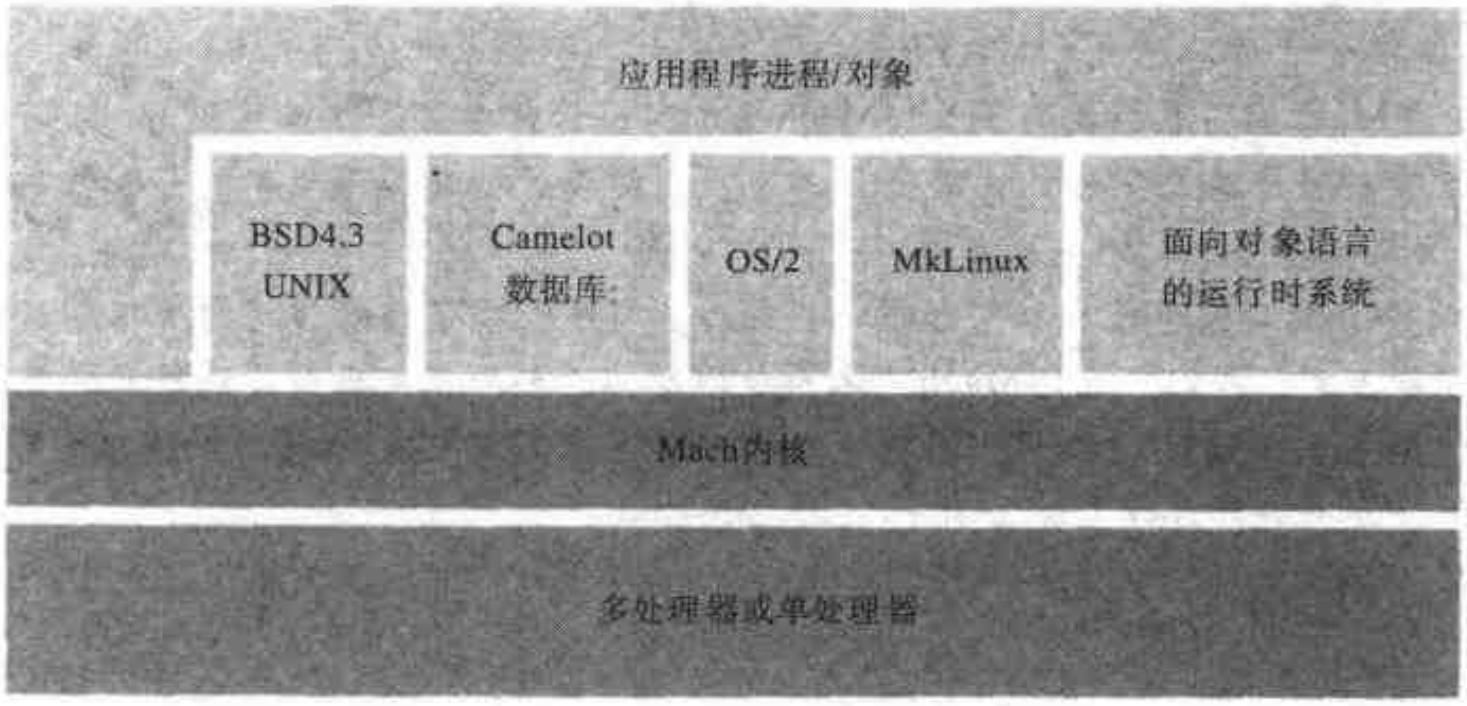


图18-1 Mach对操作系统、数据库和其他子系统的支持

18.1.1 设计目标和主要设计特点

Mach的主要设计目标和特点如下：

- 多处理器操作 Mach被设计用来在共享内存的多处理器系统中执行，这样，可以由任一处理器执行核心线程和用户态线程。Mach提供了用户进程的多线程模型，线程的执行环境被称为任务。不管线程是属于同一任务还是属于不同任务，它们都是抢占式调度，这样可以允许在共享内存的多处理器上并行执行。
- 对网络操作的透明扩展 为了能使分布式程序透明地应用到网络上的单处理器和多处理器系统上，Mach采用了一种位置无关的通信模型，该模型将端口作为通信目的地。然而，Mach内核被设计为100%与网络无关。Mach设计完全依赖于用户级网络服务器进程在网络上透明地传递消息（图18-2）。考虑到第6章我们介绍过的上下文切换开销，这还是一种有争议的设计选择。然而，它却能使系统完全灵活地控制网络通信策略。
- 用户级服务器 Mach实现了一个面向对象的模型，其中资源可以由内核管理，也可以由动态加载的服务器管理。最初系统只允许用户级服务器，但是后来Mach改造成在内核的地址空间内可以载入服务器。正如我们已经提过的，Mach系统的主要目标之一是将绝大多数的UNIX程序都在用户级实现，同时又能提供和已有UNIX的二进制兼容性。除

了一些内核管理的资源之外，系统统一使用消息传递来访问资源，但这些资源是被服务器管理的，每一个资源都有一个与之相对应的由服务器管理的端口。Mach接口生成器(MiG)可以产生RPC存根，这些RPC存根被用来在语言级隐藏基于消息的访问[Draves *et al*. 1989]。

701

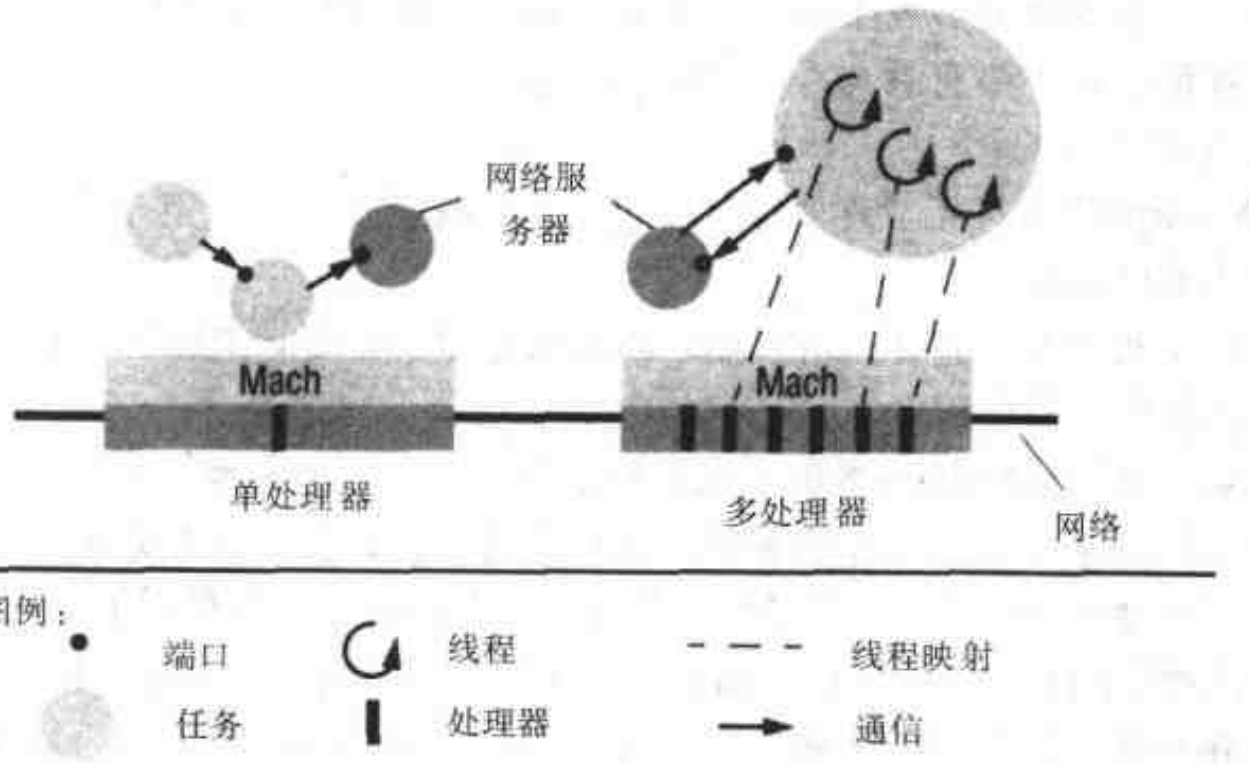


图18-2 Mach的任务、线程和通信

- *操作系统仿真* 为了支持对UNIX和其他操作系统二进制层次的仿真，Mach允许将操作系统调用透明地重定向到仿真库调用，然后调用用户级的操作系统服务器——这种技术叫做蹦床。它也能提供另一种功能，允许服务器处理像非法地址访问这类由应用程序引起的异常。读者可以在www.cdk3.net/oss中找到在Mach和Chorus中进行UNIX仿真的实例研究。
- *灵活的虚拟内存实现* Mach中做了许多工作，用于增强虚拟内存，使之可应用于UNIX系统仿真和支持其他子系统，其中包括采用一种灵活的方法分配进程的地址空间。Mach支持大的、分散的进程地址空间，这意味着它包含了许多区域。例如，消息和打开的文件都可以表示为一个虚拟存储区域。任务可以拥有私有的地址区域，任务之间也可以共享地址区域，而且任务可以从其他任务中复制地址区域。这种设计包括使用内存映射技术，特别是其中的写时复制技术，可以减少诸如在任务之间传递消息此类的数据复制。最后，Mach可以允许服务器而不是内核本身来实现虚拟分页的存储支持。区域可以映射到外部分页器管理的数据上。被映射的数据可以驻留在内存资源的任何一般化抽象中，例如驻留在分布式共享内存或文件中。
- *可移植性* Mach可以移植到多种硬件平台上。因此，Mach尽可能地分离与机器相关的程序代码。特别是，虚拟内存代码被划分为机器相关部分和机器无关部分[Rashid *et al*. 1988]。

702

18.1.2 Mach主要的抽象概述

我们可以将Mach内核提供的抽象机制总结如下（本章后面将对它们进行详细介绍）：

- *任务* 一个Mach任务是一个执行环境。它主要包括一个被保护的地址空间和一个内核管理的权能集合，这些权能主要用于访问端口。
- *线程* 任务可以包含多个线程。在共享内存的多处理器系统中，属于同一个任务的线程

可以在不同的处理器上并行执行。

- 端口　在Mach中，一个端口是一个具有相关消息队列的单点单方向通道。Mach程序员不能直接访问端口，端口也不是任务的一部分。但是，程序员可以获得端口权限的处理权柄，这里，端口权限是向端口发送消息或从端口接收消息的权能。
- 端口集　一个端口集是属于同一任务的接收权限的端口集，合它用于从端口集合中任一端口接收消息。不要将它与端口组混淆。端口组是组播的目的地集合，在Mach并中没有实现端口组。
- 消息　Mach中的消息除了数据外，还包含端口权限。内核采用存储管理技术在任务间高效地传输消息数据。
- 设备　像文件服务器这样运行在用户级的服务器一定要访问设备。基于此原因，内核为底层设备提供了底层接口。
- 内存对象　在一个Mach任务的虚拟地址空间中，每一个区域都对应于一个内存对象。这种内存对象通常是在内核外实现的，但是当内核执行虚拟内存分页操作时，内核就会访问它。一个内存对象是一个抽象数据类型的实例，这一抽象数据类型包含了获取和存储，当线程试图引用相应区域的地址而引发一个页失配时需要访问的数据的操作。
- 内存缓存对象　对于每一个被映射的内存对象，系统中存在一个内核管理的对象，这一对象包含对存在于主存中相应区域的存储页的缓存。它被称为内存缓存对象。它支持实现内存对象的外部分页器所需要的操作。

我们现在将考虑Mach的主要抽象。为了简明起见，我们省略了设备抽象。

18.2 端口、命名和保护

Mach用端口来区别每个资源。访问资源的请求会通过消息传递到相应的端口上。Mach认为服务器会管理许多端口：每个资源都对应一个端口。一个单服务器的UNIX系统大约使用
703 2000个端口[Draves 1990]。因此，端口的创建和管理开销必须降低。

如何保护资源不被非法访问的问题等同于如何保护相应的端口以防止非法的消息传送。在Mach中，在通过内核控制对端口权能的获取来保护端口的同时也通过网络服务器控制从网络到达的信息来保护端口。

端口权能有一个字段，用于表示任务对端口的访问权限。有3种类型的端口权限。发送权限允许拥有该权限的任务内的线程可以向对应的端口发送消息。比这一权限类型更严格的是发送一次权限，它最多只允许向相应端口发送一次消息，在这一次消息发送后，内核自动收回其权限。这种权限允许客户在知道服务器不会再向其发送信息的前提下获得服务器的应答（这样可以保护客户，防止它受到发生错误的服务器的影响）；另外，服务器可以不用回收从客户接收到的发送权限。最后一种权限是接收权限，它允许任务中的线程从端口消息队列中接收信息。在任意时刻，最多只能有一个任务拥有接收权限，而可以有任意多个任务同时拥有发送权限和发送一次权限。Mach支持*N*对1的消息通信：内核不直接支持组播。

系统在创建任务时赋予任务一个自举端口权限，这是一个发送权限，用来获得其他任务的服务。在创建后，属于任务的线程通过创建端口或者通过接收消息中的端口权限来获得更多的端口权限。

Mach的端口权限存储在内核中，并且受其保护（图18-3）。任务通过本地标识来引用端口

权限，这些本地标识仅在任务的本地端口名字空间内有效。这就允许了内核实现者可以选择这些权能更有效的表示方法（例如指向消息队列的指针），并且选择整数本地标识名以便于内核根据名字查找权能。实际上，和UNIX的文件描述符相似，本地标识是整数，它可以用来对包含任务权能信息的内核表建立索引。

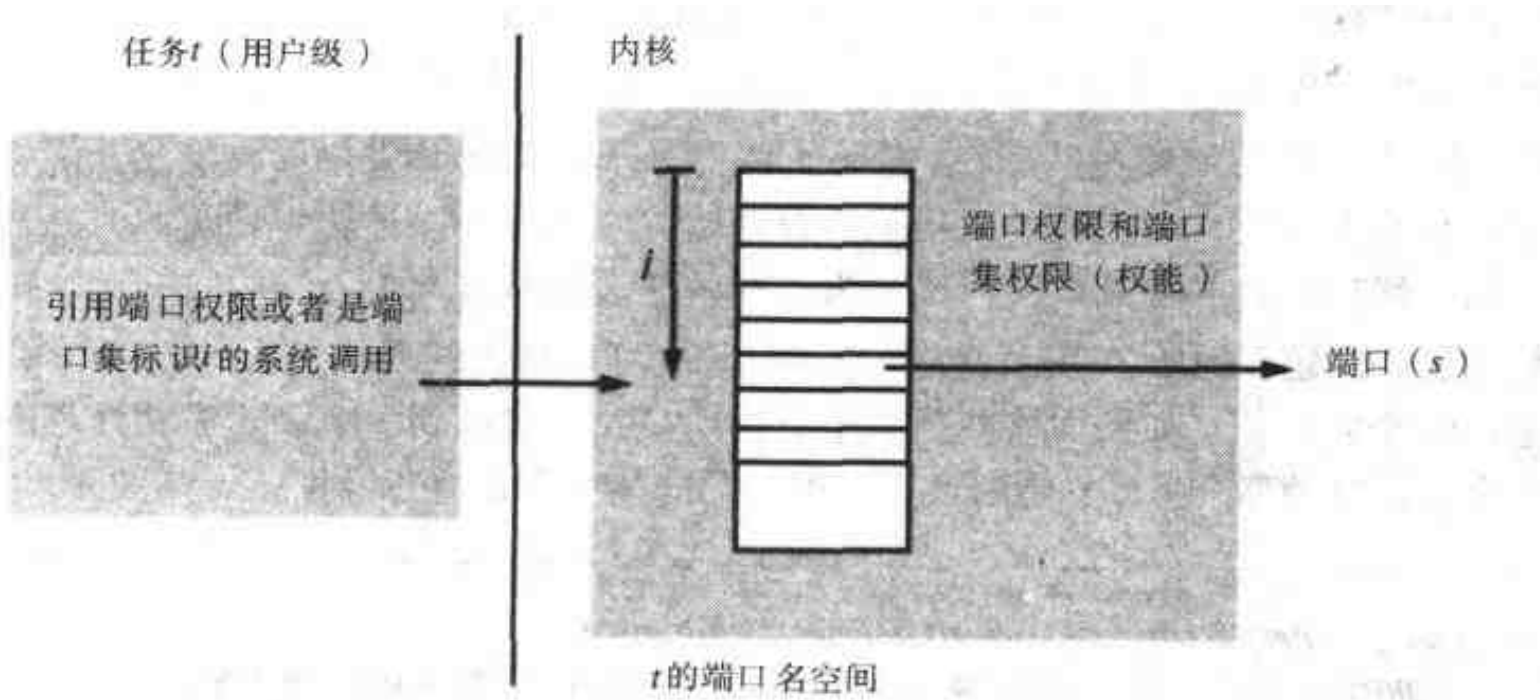


图18-3 一个任务的端口名字空间

Mach的命名和保护方案使系统可以通过给定的用户级标识快速地访问本地消息队列。与此优点相对应的缺点是，当任务之间以消息的形式传输权限时，内核处理的开销可能比较大。 704
至少，在接收方的任务命名空间及其核心表空间内，系统必须为发送权限分配一个本地名字。而且，从安全的角度考虑，网络服务器传输的端口权限信息必须被加密，这样才能防止窃听等方式的攻击[Sansom *et al.* 1986]。

18.3 任务和线程

一个任务是一个执行环境：任务自己不能执行任何动作；只有任务中的线程可以执行动作。然而，为了方便起见，当我们说任务执行某动作时，我们是指在这一任务中的线程执行动作。与任务直接相关的主要资源是任务的地址空间、线程、端口权限、端口集以及用于查找端口权限和端口集的本地名字空间。我们将讨论创建一个新任务的机制以及与任务管理和任务中线程执行相关的一些特点。

创建一个新的任务 UNIX的*fork*命令通过复制一个已存在的进程创建一个新的进程。Mach的进程创建模型是UNIX模型的推广。通过参考我们所说的蓝图任务（它不必是创建者）可以创建一个新任务。新任务和蓝图任务驻留在同一计算机上。因为Mach没有提供任务迁移功能，在远程计算机上建立一个任务的惟一方法是通过一个已经驻留在于那台远程计算机上的任务。新任务的自举端口权限是从它的蓝图任务继承的，新任务的地址空间要么是空的，要么是从它的蓝图任务继承的（在后面介绍Mach的虚拟内存时，会讨论地址空间的继承）。刚创建的任务不包含线程。任务的创建者会要求在该任务的子任务中创建一个线程。此后，通过在任务中已存在的线程创建新的线程。参见图18-4所描述的一些与任务和线程创建有关的Mach调用。

调用内核操作 当一个Mach任务或线程被创建后，系统就赋予它一个所谓的内核端口。

Mach的“系统调用”被划分为两类，一类直接用内核陷入实现，另一类通过将消息传递到内核的端口上来实现。后一种方法的优点在于：它允许远程任务和线程的操作和本地操作一样，是网络透明的。内核服务管理核心资源的方式与用户级服务器管理其他资源的方式相同。每一个任务在它的内核端口上都具有发送权限，这使得任务可以调用基于自身的操作（例如创建一个新的线程）。每个通过消息传递访问的内核服务都有一个接口定义。任务通过存根过程访问这些服务，其中，存根是由Mach接口生成器根据它的接口定义生成的。

异常处理 除了内核端口之外，任务和线程（可选的）还可以拥有一个异常端口。当某种类型的异常发生时，内核通过向相关的异常端口发送一个描述这一异常的消息来响应这一异常。如果线程没有异常端口，内核便为其寻找任务的异常端口。接收这一消息的线程可能试图解决这一问题（例如，当地址空间越界时，它可以增大线程的栈），然后，它会在应答消息中返回一个状态值。如果内核找到一个异常端口并且接收到一个表示成功处理的应答，那么它就会重新启动那个引发异常的线程。否则，内核将终止这一线程。

705

task_create(parent_task, inherit_memory, child_task)
parent_task is the task used as a blueprint in the creation of the new task, *inherit_memory* specifies whether the child should inherit the address space of its parent or be assigned an empty address space, *child_task* is the identifier of the new task.

thread_create(parent_task, child_thread)
parent_task is the task in which the new thread is to be created, *child_thread* is the identifier of the new thread. The new thread has no execution state and is suspended.

thread_set_state(thread, flavour, new_state, count)
thread is the thread to be supplied with execution state, flavour specifies the machine architecture, *new_state* specifies the state (such as the program counter and stack pointer), *count* is the size of the state.

thread_resume(thread)
This is used to resume the suspended thread identified by *thread*.

图18-4 创建任务和线程

例如，当一个任务发生非法地址空间访问或者除零时，内核会向异常端口发送一个消息。异常端口的拥有者可以是一个调试任务，依靠Mach的与位置无关的通信方式，这一调试任务可以在网络上的任意位置执行。页失配由外部分页器处理。18.4节将介绍Mach系统如何在一个仿真操作系统上处理系统调用。

任务和线程管理 在内核接口中大约有40个过程是关于任务和线程的创建和管理的。每一个过程的第一个参数是相应内核端口的发送权限，并且消息传递系统调用被用来请求目标内核的操作。图18-4给出了其中的一些任务和线程调用。总而言之，系统可以分别设置线程调度优先级；线程和任务可以被挂起、解挂和终止；并且系统可以在外部设置、读出或修改线程的执行状态。后一种功能对调试和设置软件中断很重要。但更多的内核接口调用是关于如何将任务的线程分配给特定处理器集合的。在多处理器系统中，处理器集合是处理器的子集。通过将线程分配到处理器集合上，当前可用的计算资源可以根据不同的活动进行粗略的划分。读者可以在Loepere[1991]中找到关于内核对任务和线程管理以及对处理器分配的详细

介绍。Tokuda等[1990]和Lee等[1996]描述了为实现实时线程调度和同步而对Mach所进行的扩充。

706

18.4 通信模型

Mach为消息传递提供了一个独立的系统调用：*mach_msg*。在描述它之前，我们再详细介绍一下Mach中的消息和端口。

18.4.1 消息

消息包含一个固定大小的消息头部，后面是一个变长的数据项列表（图18-5）。

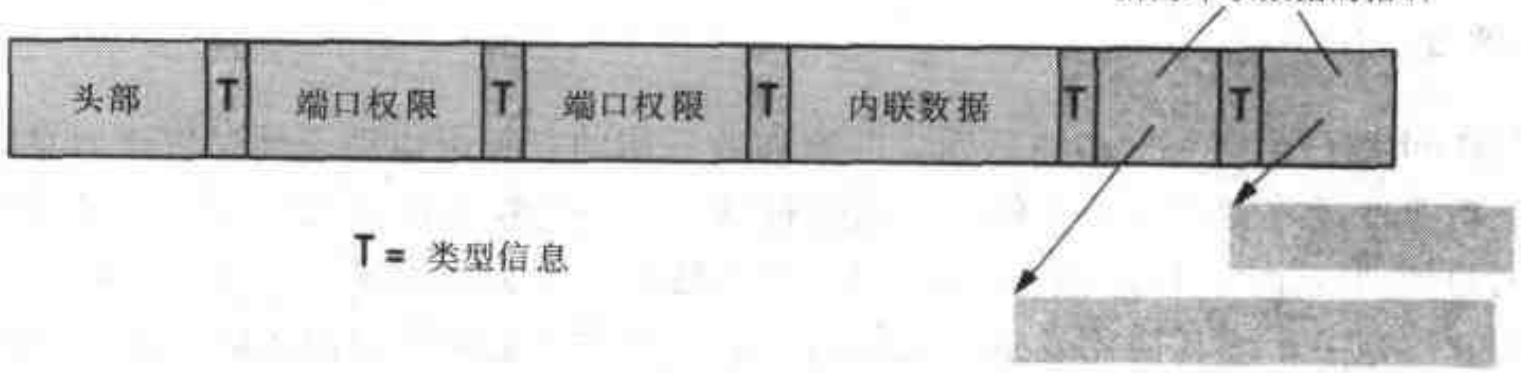


图18-5 包含端口权限和外联数据的Mach消息

固定大小的消息头包括：

- 目的端口 为简单起见，目的端口是消息的一部分，而不是*mach_msg*系统调用的一个独立的参数。它由适当的发送权限的本地标识来指定。
- 应答端口 如果需要应答，那么在消息中要包含本地端口（它是发送此消息的线程具有接收权限的一个端口）的发送权限。
- 操作标识 它标识在服务接口中的操作（过程），它只对应用程序有意义。
- 额外的数据长度 在消息头部之后（紧接着消息头部），一般是一个类型项的变长列表。除了对该字段有二进制位数的限制以及地址空间总长度的限制之外，额外数据没有长度的限制。

消息头部之后是一个列表，列表的每一个数据项是下面几种类型的数据之一（在消息中，它们可按任何次序出现）：

- 带类型的消息数据 独立的且具有内联类型标记的数据项；
- 端口权限 通过它们的本地标识给出；
- 外联数据的指针 此类指针所指向的数据被存储在独立的不连续的内存块中。

Mach消息由一个固定大小的消息头部和多个变长数据块组成，其中的一些数据是外联的（即数据块是非连续的）。然而，当外联数据被发送后，内核——不是接收信息的任务——在接收信息的任务的地址空间内选择存储外联数据的存储位置。这是使用写时复制技术来传输这种数据时所产生的副作用。接收任务在接收到包含附加的虚拟内存区的消息后，如果它不再需要这一数据时，它必须自己显式地释放这些区域。因为虚拟内存操作的开销要大于对小块数据复制的开销，因此建议只有在需要传输大量数据时才使用外联数据传输。

707

在一个消息中允许有多个数据部分的优点在于它可以允许程序员分别为数据和元数据分配内存。例如，一个文件服务器可能从自己的缓存中定位所请求的磁盘块。它不是将数据块

的内容复制到消息缓冲区内，并将其放置到消息头部之后，而是在应答消息中提供一个适当的指针，接收任务可以从这一指针指向的磁盘块所在地直接获得数据。这是一种分散-聚集I/O的形式，按这种方法，在一次系统调用中，数据写入调用者地址空间内的多个区域，或者从多个区域读出数据。UNIX的*readv*和*writev*系统调用也提供了这一功能[Leffler *et al.* 1989]。

Mach消息中的每个数据项都由发送者指定类型（例如，在ASN.1中）。当数据在网络上传输时，这使用户级网络服务器能以一个标准格式将这些数据编码。然而相对于用存根过程（由接口定义生成的）进行的编码和解码，这种编码方案的执行效率较低。存根过程中包含涉及的数据类型，所以，它不必将这些类型信息包含在消息中就可以直接将数据编码成消息（见4.2节）。然而，当一个网络服务器对发送者的带类型数据进行编码时，它必须将这些带类型的数据复制到另一个消息中。

18.4.2 端口

一个Mach端口包含一个消息队列，具有接收权限的任务可以动态地设置该消息队列的长度。这个功能使接收者能实现某种形式的流控制。当一个线程使用正常的发送权限向一个端口发送消息，而这个端口的消息队列已满，那么这个线程会被阻塞，直到消息队列中有空间可用时为止。当一个线程使用发送一次权限时，接收方总会将该消息入队，即使消息队列满了也是如此。因为使用的是发送一次权限，接收方知道源端不会再发送消息。在向客户发出应答消息时，服务器线程可以通过使用发送一次权限来避免被阻塞。

发送端口权限 当消息中包含端口发送权限时，消息接收者就获得了这一端口的发送权限。当接收权限被传输时，发送消息的任务自动放弃了这一端口的接收权限。这是因为同一端口接收权限在同一时刻的拥有者不能多于一个。新的端口接收权限拥有者可以接收在端口队列上排队的消息和后续到达这一端口的消息，在某种意义上说，这对于向此端口发送消息的任务是透明的。当在单个计算机内传输权限时，系统可以相对直接地实现接收权限的透明传输。其获得的权能仅是指向本地消息队列的一个指针。而在计算机间进行传输时，会导致一系列更复杂的设计问题。我们将在后面讨论它。

监控连接 当发送消息和接收消息无效时，内核需要通知发送者和接收者。为了这一目的，
708 内核记录了给定端口的发送权限数目和接收权限数目。如果没有任务拥有特定端口的接收权限（例如，因为拥有这一权限的任务出故障了），那么，在本地任务的端口名字空间中的所有发送权限将成为*死名*。当一个发送者试图使用指向一个端口的名字，而这一端口的接收权限又不存在时，内核将这一名字变为死名并返回一个错误提示。相似地，任务可以请求内核异步地通知给定端口的发送权限不存在了。内核使用请求线程给它的发送权限发送给请求线程一个消息来完成该通知。可以通过引用计数来得知是否具有发送权限：当创建一个发送权限时，引用计数加一，当删除一个发送权限时，它就减一。

需要强调的是在单个内核的范围内跟踪无接收者/无发送者的开销比较小。相反，在分布式系统中，这是一个复杂且开销很大的工作。如果权限可以由消息发送，那么给定端口的接收或发送权限就有可能处于下列几种情况之一：被某个任务所拥有；在消息中；在某一端口的等待队列中或在计算机之间传送。

端口集 端口集是在一个任务内创建的由本地管理的端口集合。当一个线程对一个端口集执行一个接收操作时，内核会返回一个消息，该消息被传送给这一集合的某些成员。它也返

回这一端口接收权限的标识，这样，线程就可以根据它来处理消息。

因为通常一个服务器需要始终接收所有端口的客户消息，因此端口集是有用的。线程从一个消息队列为空的端口接收消息会被阻塞，即使此时这一线程可以处理的消息已经到达了另一端口，此线程仍然被阻塞。为每一端口指定一个线程可以克服这一问题，但是对于有大量端口的服务器来说，这是不可行的，这是因为线程的开销比端口大。通过将这些端口集合成为一个端口集，那么系统可以使用单个线程来处理到达的信息，而不必要担心信息丢失。此外，如果在任意端口上都没有消息时，那么这一线程会被阻塞，所以这种方法可以避免"忙等"的情况，即线程不停轮询直至有消息到达某个端口。

18.4.3 mach_msg 系统调用

*mach_msg*系统调用用于异步消息传递和请求－应答方式交互，这使得该调用极其复杂。我们只对其语义做简单介绍。下面表示的是一个完整的调用：

---

mach_msg(*msg_header*, *option*, *snd_siz*, *rcv_siz*, *rcv_name*, *timeout*, *notify*)

*msg_header*指向被发送消息和所接收消息的公共的消息头部，*option*标明是发送、接收还是两者都有。*snd_siz*和*rcv_siz*给出发送和接收消息缓冲区的大小。*rcv_name*表示端口或端口集的接收权限（如果是接收一个消息），*timeout*设置发送和/或接收消息的总的时间限制，*notify*提供了端口权限，在发生异常时，内核可以用这一端口权限来发送通知信息。

709

*mach_msg*可以发送消息、接收消息或者同时发送和接收消息。客户发送请求消息和接收应答，服务器应答上一个客户和接收下一个请求消息都使用这个系统调用。另一个使用发送/接收组合的系统调用的好处是：若客户和服务器都在同一计算机上执行，那么，实现可以采用一种叫传递调度的方法来进行优化。该方法是指，当一个任务发送一个消息给另一个任务后，在它即将被阻塞时，它将自己剩余的时间片"赠予"给其他任务的线程。这比在可运行线程队列中查找下一个可运行的线程的开销要少。

同一线程发送的消息会按照发送顺序来发送，同时消息传递是可靠的。至少，公共内核上的两个任务间的消息是有保证的，即使在缺乏缓冲区的情况下也是如此。当消息通过网络传输到故障独立的计算机上时，系统可以提供至多一次的传递语义。

线程可能被无限期地阻塞，例如线程等待一个永远不会到达的消息，或者等待一个有故障的服务器端口上的队列空间，在这种情况下，超时机制是有用的。

18.5 通信实现

Mach通信实现的最有趣的一个方面是使用用户级网络服务器。在每一个计算机上有一个网络服务器（在Mach中被称为*netmsgservers*），它们相互合作，负责将本地通信的语义扩展到网络上。这包括尽可能保持传递保证以及如何使网络通信变得透明，还包括实现和监视端口权限的传输。特别是，网络服务器负责保护端口以防止非法访问，同时维护在网络上传输的消息数据的私密性。读者可以在Sansom等[1986]中找到关于Mach对保护问题的详细介绍。

18.5.1 透明消息传递

因为端口通常只局限于在Mach内核上，所以有必要从外部加入一个网络端口的抽象概念，

有了网络端口，消息可以在网络上传播。每个网络端口都有一个全局惟一的标识，它只由网络服务器处理，网络服务器在任意时刻都将网络端口与一个Mach端口联系起来。网络服务器拥有网络端口的发送和接收权限的方式和任务拥有Mach端口的发送和接收权限的方式相同。

图18-6表示了在不同计算机上的任务之间的消息通信方式。发送任务所拥有的权限是对于本地端口的，本地网络服务器所拥有的是这一端口的接收权限。图中网络服务器对于接收权限的本地标识是8。在发送端计算机上的网络服务器根据它具有发送权限的端口在网络端口表中查找该端口的权限标识项，从而产生网络端口和网络地址提示。它通过附加网络端口向表中提示的地址的网络服务器传输消息。接收到这一消息的网络服务器会从消息中抽取网络
710 端口，然后，它在网络端口表中查找这一端口项。如果它找到了一个合法项（网络端口可能已经被重分配给了其他内核），那么该项包含了对一个本地Mach端口的发送权限标识。这个网络服务器会使用这一权限来传送消息，这样消息就被传输到相应的端口上。网络服务器的整个处理过程对发送方和接收方来说都是透明的。

如何建立图18-6所示的表呢？新启动计算机上的网络服务器会进行一个初始化协议，利用该协议，可以获得整个网络内的发送权限。下面考虑当包含端口权限的消息在网络服务器之间传输后发生的情况。这些权限是具有类型的，因此网络服务器可以找到这些权限信息。如果一个任务发送一个本地端口的发送权限，本地网络服务器就创建一个网络端口标识，并为这一本地端口创建一个网络端口表项（如果它不存在），同时将这一标识附加在要发送的权限消息上。接收到权限消息的网络服务器在对应表项不存在的情况下也建立一个端口表项。

当接收权限被传输时，情况就更复杂了。系统需要实现迁移透明性，以下是一个例子：当端口迁移时，客户应该能将消息发送到该端口上。首先，发送者本地的网络服务器需要接收权限。以这一端口为目的的所有消息，不管是本地的还是远程的，都先发送到这一服务器上。接着，它用协议保证将接收权限一致传输到目的网络服务器上。

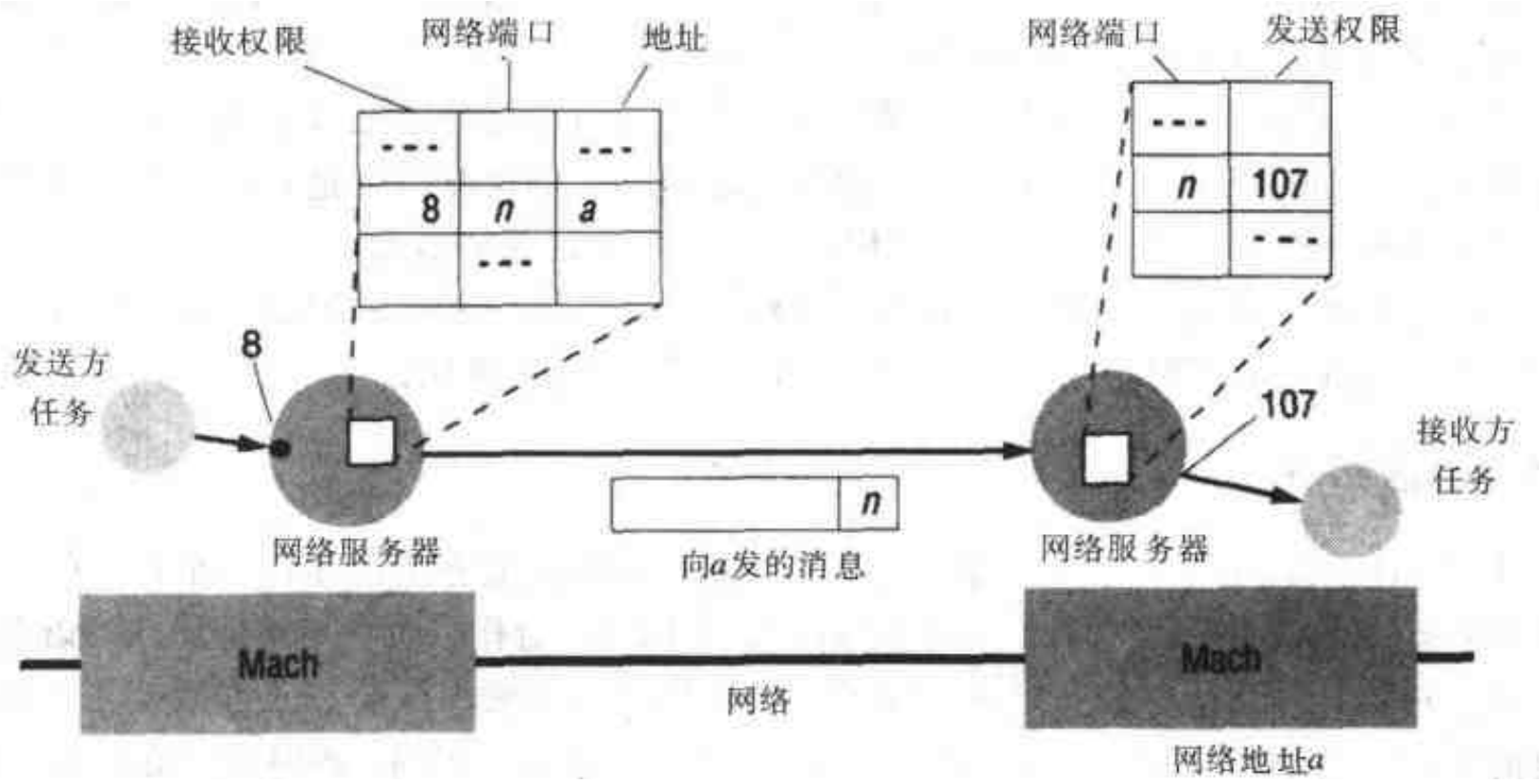


图18-6 在Mach中的网络通信

这一权限传输中的一个重要的问题是，如何使原来传输到这一端口上的消息现在传输到新获得接收权限的计算机上。一种可能的方法是，由Mach记录下具有针对给定的网络端口发送权限的网络服务器，并且当接收权限被传输时，直接通知这些服务器。因为这种方法的管

理开销太大，所以不被采纳。使用硬件组播方式可以向所有网络服务器广播权限位置的改变，这种方法的开销要少一些。然而，这种广播服务是不可靠的，并且并不是所有的网络上都有硬件广播。因此，Mach由拥有端口发送权限的网络服务器负责定位网络端口。 711

前面提到过，网络服务器使用地址提示信息来向另一个网络服务器发送消息。可能的回复信息包括：

- *port here* 这一目的端拥有接收权限
- *port dead* 这一端口已经被删除了
- *port not here*，*transferred* 接收权限被传输到一个特定的地址上
- *port not here*，*unknown* 没有网络端口的记录
- *no response* 目的计算机已经关闭了

如果系统返回一个发送地址，网络服务器就按这一地址发送消息；但是系统返回的地址只是一个提示，它可能不精确。如果发送者将这些地址都试过了，还没有找到合适的地址，那么它们就只好求助于广播了。如何管理这一系列由发送地址组成的地址链以及拥有发送地址的计算机崩溃时要做的操作是这种定位算法的主要设计问题，特别是在广域网中的主要设计问题。Black和Artsy[1990]描述了在广域网上如何使用这种发送地址链的问题。

获得迁移透明性的第二个问题是如何同步消息传递。Mach保证同一线程发送的两个消息以同样的顺序传输。当接收权限被传输时，如何实现这一保证呢？如果不小心的话，在旧的信息还在原来的计算机上排队时，新的网络服务器就有可能传递新的信息。网络服务器为了解决这一问题可以延迟原计算机上的消息传递，直到所有排队的消息都被发送到目的端计算机为止。此后，系统可以安全地对消息传输进行重新路由，同时发送地址被返回给发送者。

18.5.2 开放性：协议和驱动程序

传输协议 尽管Mach的设计初衷是运行在用户空间内的网络服务器应可以允许广泛的传输协议，然而，广泛使用的Mach网络服务器只采用TCP/IP作为它的传输协议。采用这一协议的原因之一是UNIX的兼容性，而卡内基－梅隆大学选择它的另一个原因在于：该校网络包含1700多台计算机，其中大约有500台运行Mach。在这样的网络系统上使用TCP/IP可以获得效率比较高的健壮性。然而，当LAN中请求－应答交互占主导地位时，从性能方面来考虑，以上选择并不是非常合适。我们将在下面18.6节介绍Mach通信的性能。

用户级网络驱动程序 一些网络服务器提供了它们自己的用户级的网络设备驱动程序，目的是加快网络访问。除了获得关于使用各种硬件的灵活性之外，将设备驱动程序放置在用户级可以补偿由于使用用户级的网络服务器而导致系统性能的降低。内核为每一设备提供一个抽象，包含一个将设备控制器的寄存器映射到用户空间的操作。在以太网的情况下，控制器使用的寄存器和包缓冲区可以被映射到网络服务器的地址空间内。此外，当中断发生时，在内核运行的特殊代码会唤起用户级线程（在这种情况下，它属于网络服务器）。这一线程可以处理中断，处理方法是在它与控制器使用的缓冲区之间传输数据以及为下一操作重新设置控制器。 712

18.6 内存管理

Mach系统的著名之处不仅是由于它使用了大而分散的地址空间，而且还在于它允许任务

间共享内存的虚拟内存技术。不仅是同一Mach内核上执行的任务之间可以在物理上共享内存，而且Mach对外部分页器（在Mach文献中被称为内存管理器）的支持还可以允许任务间共享虚拟内存，甚至当它们位于不同的计算机上也是如此。最后，Mach虚拟内存实现上的著名之处在于，划分了机器相关和机器无关两层，以便有助于移植内核[Rashid *et al.* 1988]。

18.6.1 地址空间结构

第6章介绍了包含多个区域地址空间的一个通用模型。每个区域是一段具有公共属性集的连续的逻辑地址空间。这些属性包括访问许可（读/写/执行），还包括可伸缩性。例如，系统允许栈区向低地址空间扩展；堆可以向高地址区扩展。Mach使用的模型和这个通用模型相似。

然而，Mach将它的地址空间看作一系列连续的页面组，它们用自己的地址命名，而不是将地址空间看作各自标记的区域。因此，Mach中的保护是应用在页上而不是应用在区域上。Mach系统调用访问地址和扩展，而不是区域标识。例如，Mach不会“增加栈区”。而是在栈当前使用的页面下面分配更多的页。然而，在大多数情况下，这种差异并不是很重要。

我们可以将拥有公共属性的连续页的集合看作区域。正如我们已经看到的，Mach支持大量的区域，它可以用于多种用途，例如消息数据或被映射的文件。

以下是4种创建区域的方法：

- 通过调用*vm_allocate*显式地分配区域。用*vm_allocate*新创建的区域默认用0填充。
- 使用*vm_map*可以创建与内存对象相关联的区域。
- 通过对一个蓝图任务的区域使用*vm_inherit*调用，系统将一些区域（或者区域和区域的内容）声明为是从蓝图任务中继承而来，并分派给新的任务。

713

- 作为消息传递的一种副作用，区域可以自动地被分配在任务的地址空间内。

通过使用*vm_protect*，所有的区域可以拥有读/写/执行许可集。通过使用*vm_copy*，区域可以在任务内被复制，其他任务可以通过使用*vm_read*和*vm_write*来读写区域的内容。通过使用*vm_deallocate*，可以释放所有以前分配的内存区。

我们将介绍Mach中实现UNIX *fork*操作的方式和消息传递的虚拟内存实现，为了便于利用内存共享，它们被集成在Mach中。

18.6.2 内存共享：继承和消息传递

Mach通过内存继承机制来允许UNIX *fork*语义的一般化。我们已经知道新的任务是从另一个任务创建的，而这个任务被作为一个蓝图。从蓝图任务继承而来的区域包含同样的地址范围，它的内存是以下两种类型之一：

- 共享　由同一内存支持；
- 拷贝　一个“拷贝-继承”区域是一个蓝图任务区域的副本，是在子区域创建时复制的。

在子任务不需要区域时，该区域可以被设置成非继承的。

在UNIX的*fork*操作中，子任务可以继承或共享其蓝图任务的程序正文区。对一个包含共享库代码的区域，也是这样。然而，作为蓝图区域的副本，应该继承程序的堆区和栈区。另外，如果需要蓝图任务和其子任务共享数据区域（正如UNIX System V允许的那样），那么可以将这一区域设置为共享的继承方式。

外联消息数据在任务间的传输方式和拷贝－继承方式比较相似。Mach在接收方的地址空间内创建新的区域，并且它的初始内容是从发送方作为外联数据传递来的区域副本。与继承不同的是，接收区域通常不必占据和发送区域相同的地址范围。发送区域的地址范围可能在接收方已经被另一个区域所占据。

在拷贝继承和消息传递时，Mach使用了写时复制技术。因此，Mach进行了以下乐观假设：拷贝－继承或作为外联消息数据传递的内存不会被任何任务所改写，甚至在写许可存在的情况下也是如此。

为了证实这一乐观假设，我们以UNIX的*fork*系统调用作为例子。通常在*fork*操作后会接着一个*exec*调用，它会重写地址空间的内容，包括可写的堆和栈。如果系统在*fork*操作时就物理地复制内存，那么大部分复制工作就被浪费掉了，因为在两个调用之间只有很少的页被修改。

下面以发送到本地网服务器的消息作为第二个重要的例子，这一消息可能很大。然而，如果发送任务在传输消息时并不修改这一消息，或者只修改部分消息，那么就可以节省大量的内存复制工作。网络服务器没有理由修改这一数据。那么，写时复制优化技术将帮助系统 714
节省在通过网络服务器传输中发生的上下文切换开销。

与写时复制技术相反，Chorus和DASH[Tzou and Anderson 1991]通过移动（不是复制）页支持地址空间之间的通信。Fbufs为复制和移动页开发了一些虚拟内存操作[Druschel and Peterson 1993]。

18.6.3 对写时复制的评价

虽然写时复制可以帮助在任务和网络服务器之间传递消息数据，但写时拷贝并没有减少跨网传输的困难。这是因为所涉及的计算机不共享物理内存。

只要消息传递涉及的数据量足够大，通常写时拷贝会比较有效。避免物理复制所带来的好处超过了操纵页表（当缓存被虚拟映射时——见6.3节，还包括操纵缓存）的开销。Abrossimov等[1989]给出了在Sun3/60上实现的Mach系统的一些数据，我们在图18-7中介绍了其中的部分数据。这里给出的数据只是为了说明，它并不代表大多数Mach系统的最新性能。

			被复制的数据量（在写时）		
区域长度	简单复制	创建区域	0KB（0页）	8KB（1页）	256KB（32页）
8KB	1.4	1.57	2.7	4.82	–
256KB	44.8	1.81	2.9	5.12	66.4

注：所有的时间都是以ms计。

图18-7 写时复制开销

图18-7给出了使用8KB（1页）和256KB（32页）这两种大小的区域时的时间。前两列的数据仅供参考。“简单复制”列所表示的时间是指在两个已经存在的区域之间进行所有数据的简单复制（也就是说，不使用写时复制）所耗费的时间。“创建区域”列给出的时间是创建一个用0填充的区域（还没有被访问的区域）所耗费的时间。其余列所给出的数据来源于实验，这些实验是使用写时复制将已存在的区域复制到其他区域中。这些数据包括创建复制区域、

复制修改数据以及删除复制区域的时间。对每一种区域长度都给出了在源区域中有不同数量数据被修改时的时间值。

我们比较以一页为区域长度的时间值，它的值是4.82 - 2.7=2.12ms，来源于对页的写操作，其中1.4ms来源于对页的复制，其余的0.72ms来源于页失配处理和对内部虚拟存储管理数据结
715 构的修改。对在两个本地任务之间发送8KB消息这种情况，当进行简单复制时，使用外联方式（也就是说，使用写时复制）来传输数据的好处不见得多于使用内联方式传输。内联传输会涉及到两次复制（用户到内核以及内核到用户），因此它总共会多耗费2.8ms，但是最坏的外联传输情况却相当差。另一方面，如果系统符合乐观假设，那么使用外联方式传输256KB消息所耗费的时间远小于内联方式；即使在最坏的情况下，外联方式所耗费的66.4ms要远远小于使用内联方式将256KB数据复制到内核和复制出内核所需要的2 × 44.8ms」。

在用于复制和共享数据的虚拟内存技术中，最后一点要说的是，我们必须注意指定相关的数据。尽管在前面我们没有提到过，但是，用户可以指定区域的地址范围不以页边界为起点和终点。然而，Mach强迫使用页粒度来共享内存。虽然如此，在页内没有被用户特别指定的其他数据也可以在任务间复制。这可以作为我们所说的错误共享的例子（参见第16章对共享数据项粒度的讨论）。

18.6.4 外部分页

为了保持微内核体系，Mach内核不直接支持文件或任何其他外部存储抽象。它假设这些资源是通过外部分页器实现的。与Multics系统[Organick 1972] 一样，Mach为内存对象选用了映射访问模型。程序员不需要采用显式的*read*和*write*操作来访问存储数据，他只需要直接访问相应的虚拟内存位置。映射访问的优点之一是它的一致性：系统对程序员呈现的是单一的访问数据模型，而不是两个访问数据模型。然而，关于映射访问是否比使用直接操作更优越这一问题，因为它涉及到许多方面，特别是涉及到性能，所以比较复杂，在这里我们不进行讨论。我们将集中考虑Mach虚拟内存实现的分布方面，主要包括内核之间的协议和一个管理
716 数据映射的外部分页器。

通过使用*vm_map*的调用，Mach允许一个区域与从某一内存对象的指定偏移位置开始的连续数据相关联。这一关联意味着对区域内地址的读操作可以获得由内存对象保存的数据，并且写操作对区域内数据的修改也能传播到内存对象上。尽管内核自身也提供了默认的分页器，但内存对象通常由一个外部分页器管理。内存对象由外部分页器使用的端口的发送权限表示，外部分页器处理内核关于内存对象方面的请求。

内核为每一个由它负责映射的内存对象都保留一个被称为内存缓存对象（图18-8）的本地资源。实际上，它是一个页列表，其中包含了由相应内存对象保存的数据。

外部分页器的任务包括：（a）存储那些已经被内核从缓存页中清除的数据，（b）提供内核所需要的页数据，（c）在内存资源被共享，并且同时有多个内核拥有同一内存对象的内存缓存对象的情况下，维持对底层内存资源抽象的一致性限制。

图18-9给出了内核(K)和外部分页器（EP）之间消息传递协议的主要成分。当*vm_map*被调用时，本地内核使用内存对象端口发送权限与外部分页器进行交互，该发送权限由*vm_map*调用提供，通过*memory_object_init*消息发送给外部分页器。内核在这一消息中提供发送权限，外部分页器使用这一权限来控制内存缓存对象。数据访问请求也要指定在内存对象中所需数

据的大小和地址偏移，以及所需的访问类型（读/写）。外部分页器通过*memory_object_set_attributes*消息响应请求，这一消息告诉内核是否分页器已经准备好并可以开始处理数据请求，以及是否需要提供与内存对象有关的分页器的进一步的需求信息。当一个外部分页器接收到了*memory_object_init*消息，它就能决定是否需要实现一个一致性协议，因为希望访问数据的多个内核都发送了该消息。

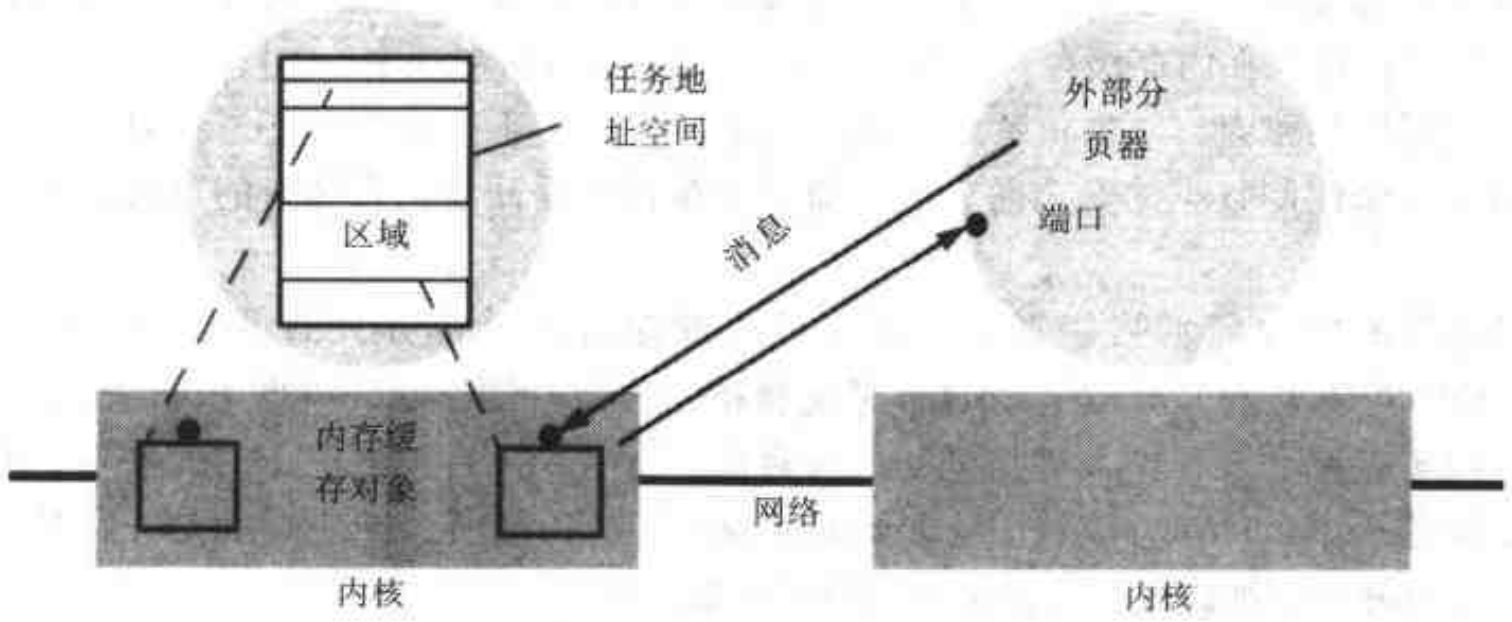


图18-8 外部分页

事　件	发送者	消　息
由任务调用的*vm_map*	K →EP	*memory-object_init*
	EP → K	*memory_object_set_attributes*, 或
	EP → K	*memory_object_data_error*
当数据帧不存在时，任务发生页失配	K →EP	*memory_object_data_request*
	EP → K	*memory_object_data_provided*, 或
	EP → K	*memory_object_data_unavailable*
内核将被修改的页写入持久存储中	K →EP	*memory_object_data_write*
外部分页器授予内核写页/设置访问许可	EP → K	*memory_object_lock_request*
	K →EP	*memory_object_lock_completed*
当页访问权限不够时发生的页失配	K →EP	*memory_object_data_unlock*
	EP → K	*memory_object_lock_request*
内存对象不再被映射	K →EP	*memory_object_terminate*
外部分页器撤销内存对象	EP → K	*memory_object_destroy*
	K →EP	*memory_object_terminate*

图18-9 外部分页器消息

18.6.5 对访问内存对象的支持

我们首先来考虑内存对象没有被共享的情况——只有一个计算机映射到内存对象上。为了具体说明，我们可以将内存对象作为文件。假设外部分页器没有预取文件数据，对应于这一文件映射区域中的所有页在开始时是由硬件保护的，不允许任何类型的访问，这是因为在这些页中还没有存储文件数据。当有一个线程试图读这一区域中的一个页时，系统便发生页失配。内核查找对应于被映射区域的内存对象端口的发送权限，并且向外部分页器（在我们的

例子中是一个文件服务器）发送一个*memory_object_data_request*消息。如果情况许可，外部分页器就发送一个*memory_object_data_provided*消息返回有关的页数据来响应请求。

如果文件数据被映射到其上的计算机修改了，那么系统有时需要写它的内存缓存对象中的页。为了做到这一点，它向外部分页器发送一个*memory_object_data_write*消息，消息中包含页数据。修改过的页被传输到外部分页器上，这是页面置换（当内核需要为另一页找出空间时）所带来的副作用。另外，为了满足持久性保证，内核可以决定将这一页写入到后备存储器中（但是把它留在内存缓存对象中）。例如，在UNIX的实现中，至少每30s，系统便将修改过的数据写回到磁盘，以防止系统崩溃。其他一些操作系统允许程序通过对一个打开文件发送*flush*命令来控制数据的安全性，这一命令会在调用返回之前将所有的已修改的文件页写回到磁盘中。

不同类型的内存资源可以拥有不同的持久性保证。外部分页器可以向内核发送一个*memory_object_lock_request*消息，这样系统将指定区域内的已修改数据送回到分页器上，分页器便可以实现符合某种持久性保证的一致存储。当内核执行完被请求的程序时，它会向外部分页器发送一个*memory_object_lock_completed*消息。（外部分页器需要这一消息，这是因为它并不知道哪些页被修改了并且需要被写回到磁盘中。）

需要注意的是，我们所描述的所有消息都是异步发送的，甚至当它们有时是在请求－应答这一组合中发生时也是如此。采用这种方式的原因如下：首先，线程在发送出请求后不必挂起，它可以继续进行其他工作。另外，当线程向一个已经崩溃的外部分页器或内核发出请求后（或者当内核向非正常运行的外部分页器发送请求而此时分页器不进行应答），它不会持续等待。最后，外部分页器可以应用基于异步消息的传输协议来实现页预取策略。它可以通过*memory_object_data_provided*消息在数据使用之前来向内存缓存对象发送页数据，而不需要等待页失配发生并且随后数据被请求。

717 ~ 718

支持对内存对象的共享访问 在我们给出的例子中，我们假设有多个在不同计算机上的任务映射到一个公共文件上。如果文件以只读方式被映射到每一个访问它的区域上，那么，就不存在一致性问题，并且对文件页的请求可以立即被满足。然而，如果至少有一个任务以可写的方式被映射到此文件，那么，外部分页器（也就是文件服务器）必须实现一个协议来保证任务不会读取同一页数据的不一致版本。

读者可以将用于获得顺序一致性的写失效协议（参见第16章的讨论）翻译成在Mach内核和外部分页器之间的传递的消息。

18.7 小结

Mach内核可以运行在多处理器计算机上，也可以运行在由网络连接的多个单处理器计算机上。它被设计成可以加入新的分布式系统特性，同时又可以保持对UNIX的兼容性。最近，Mach 3.0微内核成为了Linux操作系统中MkLinux实现的基础。

由于Mach被设计用于仿真UNIX，所以，内核本身十分复杂。内核接口包含几百个调用，尽管其中许多调用仅仅是实现*mach_msg*系统调用陷入的存根。不能仅仅靠消息传递仿真一个像UNIX这样的操作系统，还需要各种完备的虚拟内存功能，而Mach提供了这些功能。Mach的任务和线程模型以及虚拟内存管理和通信的集成在基本的UNIX功能之上进行了相当大的改进，特别是，它还试图实现UNIX服务器。它的任务内通信模型在功能上十分丰富，在语义上

非常复杂。然而，需要注意的是，只有少数系统程序员才使用这种初级的访问方式，例如，简单的UNIX管道和远过程调用都是在此之上提供的。

尽管Mach内核通常被认为是微内核，它仍包含500KB数量级的代码和初始数据（包括部分设备驱动代码）。在此之后的被称为第二代微内核系统的操作系统提供了更简单的存储管理和进程间通信功能。

第二代微内核系统优化了进程间的通信，并且超过了Mach对UNIX的仿真性能。例如，L4微内核系统[Härtig *et al.* 1997]的设计者公布他们的用户级Linux实现的开销不超过同一计算机上原有的Linux实现开销的5%~10%。与此相对照的，他们还公布了一个基准测试，在Mach 3.0上的MkLinux用户级Linux仿真的吞吐量比同一计算机上原有的Linux的吞吐量平均相差50%。

Mach将所有外部功能放在用户级这种理念最后被放弃了，系统允许服务器和内核共同定位[Condict *et al.* 1994]。但是，Härtig等公布的数据说明，甚至是内核级的MkLinux仿真也比Linux系统慢30%。 719

尽管Mach对UNIX的仿真有性能局限性，然而，Mach和Chorus系统仍然是十分重要的新型设计。它们都在被继续使用，并且它们对于内核体系结构的发展是无价的参考。读者可以在www.cdk3.net/oss上找到关于内核设计的更多的资料，其中包括关于Amoeba、Chorus以及在Mach和Chorus上仿真UNIX系统的内容。

练习

18.1 为多处理器操作设计的内核与为单处理器计算机操作设计的内核有哪些不同？

18.2 请定义（二进制层）操作系统仿真。假设用户需要它，那么，用户为什么需要它，为什么它没有被广泛采用。

18.3 请解释为什么Mach消息的内容是带类型的。

18.4 请讨论是否需要将Mach内核对特定端口发送权限数目的监控扩展到网络上。

18.5 为什么Mach要提供端口集，它什么时候同时提供线程？

18.6 为什么Mach只提供了单个通信系统调用*mach_msg*？客户和服务器怎样使用这一系统调用？

18.7 网络端口和（本地）端口的区别是什么？

18.8 Mach中的服务器管理许多名称不详的资源。

（i）请讨论以下关联的优点和缺点：

a) 所有资源与一个端口相关联

b) 一个资源与一个端口相关联

c) 一个客户与一个端口相关联

（ii）客户为服务器提供一个资源标识，服务器使用端口权限进行应答。服务器将什么类型的端口权限发送回客户？请解释为什么服务器对应于端口权限的标识和它对应于客户的标识不同。

（iii）客户所需要的资源可能位于不同的服务器计算机上。请详细解释客户如何获得使它能和服务器进行通信的端口权限，即使Mach内核只能在本地任务间传输端口权限时也是如此。

（iv）假设客户和服务器在不同的计算机上，当客户发出对某一资源操作的请求时，请说明Mach中发生的通信事件的顺序。

18.9 在机器*A*上的Mach任务向机器*B*上的任务发送一个消息。系统中发生了多少次域转
720 换，如果消息是页对齐的，那么，消息内容被复制了多少次？

18.10 设计一个协议，使其能够在端口被迁移时获得迁移透明性。

18.11 像网络驱动器这样的设备驱动如何能在用户级操作？

18.12 请解释当Mach仿真UNIX的*fork*()系统调用时使用的两种类型的区域共享，假设子任务在同一计算机上执行。子进程可以再次调用*fork*()。请解释它引起怎样的实现问题的，并对如何解决这些问题提出建议。

18.13 （i）当系统使用写时复制时，是不是必须由内核选择接收消息的地址范围？

（ii）在Mach中是不是使用写时复制来向远程目的地发送消息？

（iii）在10MIPS的32位计算机（页大小为8KB）上，一个任务异步发送给一个本地任务16KB的消息。请比较以下情况的开销：

a）简单复制消息数据（不使用写时复制）

b）使用写时复制的最好情况

c）使用写时复制的最坏情况

假设：

- 创建一个长度为16KB的空区域要用1000条指令；
- 处理页失配以及在区域内分配新页需要用100条指令。

18.14 请总结提供外部分页器的依据。

18.15 一个文件由没有共享物理内存机器上的两个任务同时打开和映射。请讨论由此引起的一致性问题。请设计一个使用Mach外部分页器消息的协议，以保证文件内容的顺序一致
721 性（参见第16章）。

参考文献

联机参考

在www.cdk3.net/refs上的联机参考列表给出了下列类型的资源链接：

- 已发表论文联机版本的链接。在参考文献列表中用ωωω符号标记，表明这些资料可通过联机参考列表给出的链接从Web获取。
- 仅在Web上存在的文档的链接。用下划线标记，例如，www.omg.org或Linux AFS，并且也带有ωωω标记。

通过链接，可直接访问到文档或者到达一个包含该文档链接的索引页。关于RFC的参考文献是被称为“征求意见稿”的一系列因特网标准和规范，我们可以从因特网工程任务组的网页www.ietf.org/rfc/和其他一些知名的站点获得相关的内容。

作者编写的一些联机资料可作为本书的补充，在书中用www.cdk3.net标记，但并未包含在本书的参考文献列表中。例如，www. cdk3.net/ipc指向我们的Web站点上关于进程间通信的一些补充资料。

Abadi and Gordon 1999 — Abadi, M. and Gordon, A. D. (1999). A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, Vol. 148, No. 1, pp. 1-70, January. ωωω

Abadi *et al.* 1998 — Abadi, M., Birrell, A.D., Stata, R. and Wobber, E.P (1998). Secure Web tunneling. In *Proceedings 7th International World Wide Web Conference*, pp. 531-9. Elsevier, In *Computer Networks and ISDN Systems*, Volume 30, Nos. 1-7. ωωω

Abrossimov *et al.* 1989 — Abrossimov, V., Rozier, M. and Shapiro, M. (1989). Generic virtual memory management for operating system kernels. *Proceedings of 12th ACM Symposium on Operating System Principles*, pp. 123-36. ωωω

Accetta *et al.* 1986 — Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M. (1986). Mach: A new kernel foundation for UNIX development. In *Proceedings Summer 1986 USENIX Conference* pp. 93-112.

Adjie-Winoto *et al.* 1999 — Adjie-Winoto, W., Schwartz, E., Balakrishnan, H. and Lilley, J. (1999). The design and implementation of an intentional naming system. In *Proceedings 17th ACM Symposium on Operating System Principles*, published as *Operating Systems Review* Vol. 34, No. 5, pp. 186-201.

Adve and Hill 1990 — Adve, S. and Hill, M. (1990). Weak ordering – a new definition. In *Proceedings 17th. Annual Symposium on Computer Architecture*, IEEE, pp. 2-14.

Agrawal *et al.* 1987 — Agrawal, D., Bernstein, A., Gupta, P. and Sengupta, S. (1987). Distributed optimistic concurrency control with reduced rollback. *Distributed Computing* Vol. 2: pp. 45-59. Springer-Verlag.

Ahamad *et al.* 1992 — Ahamad, M., Bazzi, R., John, R., Kohli, P. and Neiger, G. (1992). *The Power of Processor Consistency*. Technical report GIT-CC-92/34,

Georgia Institute of Technology, Atlanta.

Anderson 1993 — Anderson, D.P. (1993). Meta-scheduling for distributed continuous media. *ACM Transactions on Computer Systems*, Vol. 11, No. 3.

Anderson *et al.* 1990a — Anderson, D.P., Herrtwich, R.G. and Schaefer, C. (1990). *SRP – A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet*. Technical report 90-006, International Computer Science Institute, Berkeley, Calif.

Anderson *et al.* 1990b — Anderson, D.P., Tzou, S., Wahbe, R., Govindan, R. and Andrews, M. (1990). Support for continuous media in the DASH System. *Tenth International Conference on Distributed Computing Systems*, Paris.

Anderson *et al.* 1991 — Anderson, T., Bershad, B., Lazowska, E. and Levy, H. (1991). Scheduler activations: efficient kernel support for the user-level management of parallelism. In *Proceedings 13th ACM Symposium on Operating System Principles*, pp. 95–109.

Anderson *et al.* 1995 — Anderson, T., Culler, D., Patterson, D. and the NOW team. (1995). A case for NOW (Networks Of Workstations), *IEEE Micro*, Vol. 15, No. 1.

Anderson *et al.* 1996 — Anderson, T.E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S. and Wang, R. Y. (1996). Serverless Network File Systems. *ACM Trans. on Computer Systems* 14,1. pp. 41-79. February.

ANSA 1989 — ANSA (1989). *The Advanced Network Systems Architecture (ANSA) Reference Manual*. Castle Hill, Cambridge, England: Architecture Project Management.

ANSI 1985 — American National Standards Institute (1985). *American National Standard for Financial Institution Key Management*, Standard X9.17 (revised).

Arnold *et al.* 1999 — Arnold, K., O'Sullivan, B., Scheifler, R.W., Waldo, J. and Wollrath, A. (1999). *The Jini Specification*, Reading, Mass: Addison-Wesley.

Attiya and Welch 1998 — Attiya, H., and Welch, J. (1998). *Distributed Computing – Fundamentals, Simulations and Advanced Topics*, McGraw-Hill.

Babaoglu *et al.* 1998 — Babaoglu, O., Davoli, R., Montresor, A. and Segala, R. (1998). System support for partition-aware network applications. In *Proceedings 18th InternationalConference on Distributed Computing Systems* (ICDCS '98), pp. 184-191.

Bacon 1998 — Bacon, J. (1998). *Concurrent Systems*, second edition. Wokingham, England: Addison-Wesley.

Baker 1997 — Baker, S. (1997). *CORBA Distributed Objects Using Orbix*, Harlow, England: Addison-Wesley.

Bal *et al.* 1990 — Bal, H.E., Kaashoek, M.F. and Tanenbaum, A.S. (1990). Experience with distributed programming in Orca. In *Proceedings International Conference on Computer Languages '90*, IEEE, pp. 79-89.

Balakrishnan *et al.* 1995 — Balakrishnan, H., Seshan, S. and Katz, R.H. (1995). Improving reliable transport and hand-off performance in cellular wireless networks. In *Proceedings ACM Mobile Computing and Networking Conference*, ACM, pp. 2-11.

Balakrishnan *et al.* 1996 — Balakrishnan, H., Padmanabhan, V., Seshan, S. and Katz, R. (1996). A

Comparison of Mechanisms for Improving TCP Performance over Wireless Links. *Proceedings of the ACM SIGCOMM '96 Conference*, pp. 256-69.

Banerjea and Mah 1991 — Banerjea, A. and Mah, B.A. (1991). The real-time channel administration protocol. *Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg.

Baran 1964 — Baran, P. (1964). *On Distributed Communications.* Research Memorandum RM-3420-PR, Rand Corporation. ωωω

Barborak *et al.* 1993 — Barborak, M., Malek, M. and Dahbura, A. (1993). The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, Vol. 25, No. 2, pp. 171-220.

Barghouti and Kaiser 1991 — Barghouti, N.S. and Kaiser G.E. (1991). Concurrency control in advanced database applications. *Computing Surveys*, Vol. 23, No. 3, pp. 269-318.

Bartoli *et al.* 1993 — Bartoli, A., Mullender, S.J. and van der Valk, M. (1993). Wide-address spaces – exploring the design space. *ACM Operating Systems Review*, Vol. 27, No. 1, pp. 11-17.

Bates *et al.* 1996 — Bates, J., Bacon, J., Moody, K. and Spiteri, M. (1996). Using events for the scalable federation of heterogeneous components, *European SIGOPS Workshop.*

Bell and LaPadula 1975 — Bell, D.E. and LaPadula, L.J. (1975). *Computer Security Model: Unified Exposition and Multics Interpretation*, Mitre Corporation, 1975.

Bellman 1957 — Bellman, R.E. (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Bellovin and Merritt 1990 — Bellovin, S.M. and Merritt, M. (1990). Limitations of the Kerberos authentication system. *ACM Computer Communications Review*, Vol. 20, No. 5, pp. 119-32.

Berners-Lee 1991 — Berners-Lee, T. (1991). World Wide Web Seminar. ωωω

Berners-Lee 1999 — Berners-Lee, T. (1999). *Weaving The Web*. HarperCollins.

Bernstein *et al.* 1980 — Bernstein, P.A., Shipman, D.W. and Rothnie, J.B. (1980). Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions Database Systems*, Vol. 5, No. 1, pp. 18–51.

Bernstein *et al.* 1987 — Bernstein, P., Hadzilacos, V. and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems.* Reading, Mass.: Addison-Wesley.

Bershad *et al.* 1990 — Bershad, B., Anderson, T., Lazowska, E. and Levy, H. (1990). Lightweight remote procedure call. *ACM Transactions Computer Systems*, Vol. 8, No. 1, pp. 37-55.

Bershad *et al.* 1991 — Bershad, B., Anderson, T., Lazowska, E. and Levy, H. (1991). User-level interprocess communication for shared memory multiprocessors. *ACM Transactions Computer Systems*, Vol. 9, No. 2, pp. 175-198.

Bershad *et al.* 1993 — Bershad, B., Zekauskas, M. and Sawdon, W. (1993). The Midway distributed shared memory system. In *Proceedings IEEE COMPCON Conference, IEEE*, pp. 528-37.

Bershad *et al.* 1995 — Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C. and Eggers, S. (1995). Safety and performance in the

SPIN operating system. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267-84.

Bhatti and Friedrich 1999 — Bhatti, N. and Friedrich, R. (1999). *Web Server Support for Tiered Services*. Hewlett-Packard Corporation Technical Report HPL-1999-160.

Birman 1993 — Birman, K.P. (1993). The process group approach to reliable distributed computing. *Comms. ACM*, Vol. 36, No. 12, pp. 36–53.

Birman 1996 — Birman, K.P.(1996). *Building Secure and Reliable Network Applications*. Greenwich, CT: Manning.

Birman and Joseph 1987a — Birman, K.P. and Joseph, T.A. (1987). Reliable communication in the presence of failures. *ACM Transactions Computer Systems*, Vol. 5, No. 1, pp. 47–76.

Birman and Joseph 1987b — Birman, K., and Joseph, T. (1987). Exploiting virtual synchrony in distributed systems. In *Proceedings 11th ACM Symposium on Operating Systems Principles*, pp. 123-38.

Birman *et al*. 1991 — Birman, K.P., Schiper, A. and Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Transactions Computer Systems*, Vol. 9, No. 3, pp. 272–314.

Birrell and Needham 1980 — Birrell, A.D. and Needham, R.M. (1980). A universal file server. *IEEE Transactions Software Engineering*, Vol. SE-6, No. 5, pp. 450–3.

Birrell and Nelson 1984 — Birrell, A.D. and Nelson, B.J. (1984). Implementing remote procedure calls. *ACM Transactions Computer Systems*, Vol. 2, pp. 39–59.

Birrell *et al*. 1982 — Birrell, A.D., Levin, R., Needham, R.M. and Schroeder, M.D. (1982). Grapevine: an exercise in distributed computing. *Comms. ACM*, Vol. 25, No. 4, pp. 260–73.

Birrell *et al*. 1995 — Birrell, A., Nelson, G. and Owicki, S. (1993). Network objects. In *Proceedings 14th ACM Symposium on Operating Systems Principles*, pp. 217-30.

Bisiani and Forin 1988 — Bisiani, R. and Forin, A. (1988). Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions Computers*, Vol. 37, No. 8, pp. 930-45.

Bisiani and Ravishankar 1990 — Bisiani, R. and Ravishankar, M. (1990). Plus: a distributed shared memory system. In *Proceedings 17th International Symposium on Computer Architecture*, pp. 115-24.

Black 1990 — Black, D. (1990). Scheduling support for concurrency and parallelism in the Mach operating system, *IEEE Computer*, Vol. 23, No. 5, pp. 35–43.

Black and Artsy 1990 — Black, A. and Artsy, Y. (1990). Implementing location independent invocation, *IEEE Transactions Parallel and Distributed Systems*, Vol. 1, No. 1.

Blair and Stefani 1997 — Blair, G.S. and Stefani, J.-B. (1997). *Open Distributed Processing and Multimedia*. Harlow, England: Addison-Wesley.

Blakley 1999 — Blakley, R. (1999). *CORBA Security – An Introduction to Safe Computing with Objects*. Reading, Mass.: Addison-Wesley.

Bolosky *et al*. 1996 — Bolosky, W., Barrera, J., Draves, R., Fitzgerald, R., Gibson, G., Jones, M., Levi, S., Myhrvold, N. and Rashid, R. (1996). The Tiger video

fileserver, *6th NOSSDAV Conference*, Zushi, Japan, April.

Bolosky *et al.* 1997 — Bolosky, W., Fitzgerald, R. and Douceur, J. (1997). Distributed schedule management in the Tiger video fileserver, *16th ACM Symposium on Operating System Principles*, pp. 212-23, St. Malo, France, October.

Bonnaire *et al.* 1995 — Bonnaire, X., Baggio, A. and Prun, D. (1995). Intrusion free monitoring: an observation engine for message server based applications. In *Proceedings of the 10th International Symposium on Computer and Information Sciences* (ISCIS X), pp. 541-48.

Borenstein and Freed 1993 — Borenstein, N. and Freed, N., (1993). *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. September. Internet RFC 1521.

Bowman *et al.* 1990 — Bowman, M., Peterson, L. and Yeatts, A. (1990). Univers: an attribute-based name server. *Software–Practice and Experience*, Vol. 20, No. 4, pp. 403–24.

Box 1998 — Box, D. (1998). *Essential COM*. Reading, Mass: Addison-Wesley.

Boykin *et al.* 1993 — Boykin, J., Kirschen, D., Langerman, A. and LoVerso, S. (1993). *Programming under Mach*. Reading, Mass.: Addison-Wesley.

Buford 1994 — Buford, J.K. (1994). *Multimedia Systems*. Addison–Wesley.

Burns and Wellings 1998 — Burns, A. and Wellings, A. (1998). *Concurrency in Ada* , Cambridge University Press.

Burrows *et al.* 1989 — Burrows, M., Abadi, M. and Needham, R. (1989). *A logic of authentication*. Technical Report 39, Palo Alto, Calif.: Digital Equipment Corporation Systems Research Center.

Burrows *et al.* 1990 — Burrows, M., Abadi, M. and Needham, R. (1990). A logic of authentication. *ACM Transactions Computer Systems*, Vol. 8, pp. 18-36.

Bush 1945 — Bush, V. (1945). As we may think. *The Atlantic Monthly*, July.

Callaghan 1996a — Callaghan, B. (1996). *WebNFS Client Specification*, Internet RFC 2054, October.

Callaghan 1996b — Callaghan, B. (1996). *WebNFS Server Specification*, Internet RFC 2055, October.

Callaghan 1999 — Callaghan, B. (1999). *NFS Illustrated*, Reading, Mass.: Addison-Wesley.

Callaghan *et al.* 1995 — Callaghan, B., Pawlowski, B. and Staubach, P. (1995). *NFS Version 3 Protocol Specification*, Internet RFC 1813, June.

Campbell 1997 — Campbell, R. (1997). *Managing AFS: The Andrew File System*, Prentice-Hall.

Campbell *et al.* 1993 — Campbell, R., Islam, N., Raila, D. and Madany, P. (1993). Designing and implementing Choices: an object-oriented system in C++. *Comms. ACM*, Vol. 36, No. 9, pp. 117-26.

Canetti and Rabin 1993 — Canetti, R. and Rabin, T. (1993). Fast asynchronous byzantine agreement with optimal resilience.In *Proceedings 25th ACM Symposium on Theory of Computing*, pp. 42-51.

Carriero and Gelernter 1989 — Carriero, N. and Gelernter, D. (1989). Linda in context. *Comms. ACM*, Vol. 32, No. 4, pp. 444-58.

Carter *et al.* 1991 — Carter, J.B., Bennett, J.K. and Zwaenepoel, W. (1991). Implementation and performance of Munin. In *Proceedings 13th ACM Symposium on Operating System Principles*, pp. 152-64.

Carter *et al.* 1998 — Carter, J., Ranganathan, A. and Susarla, S. (1998). Khazana, An Infrastructure for Building Distributed Services, In *Proceedings of ICDCS '98*, Amsterdam, The Netherlands. ωωω

CCITT 1988a — CCITT (1988). *Recommendation X.500: The Directory – Overview of Concepts, Models and Service*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.

CCITT 1988b — CCITT (1988). *Recommendation X.509: The Directory - Authentication Framework*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.

CCITT 1990 — CCITT (1990). *Recommendation I.150: B-ISDN ATM Functional Characteristics*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.

Ceri and Owicki 1982 — Ceri, S. and Owicki, S. (1982). On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, Berkeley,Calif. pp. 117-30.

Ceri and Pelagatti 1985 — Ceri, S. and Pelagatti, G. (1985). *Distributed Databases – Principles and Systems*. McGraw-Hill.

Chandra and Toueg 1996 — Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, Apr., pp. 374-82.

Chandy and Lamport 1985 — Chandy, K. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 63-75.

Chang and Maxemchuk 1984 — Chang, J. and Maxemchuk, N. (1984). Reliable Broadcast Protocols, *ACM Transactions on Computer Systems*. Vol. 2. No. 3. pp. 251-75.

Chang and Roberts 1979 — Chang, E.G. and Roberts, R. (1979). An improved algorithm for decentralized extrema-finding in circular configurations of processors. *Comms. ACM*, Vol. 22, No. 5, pp. 281-3.

Charron-Bost 1991 — Charron-Bost, B. (1991). Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, Vol. 39, July, pp. 11-16.

Chen *et al.* 1994 — Chen, P., Lee, E., Gibson, G., Katz, R. and Patterson, D. (1994). RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, Vol. 26, No. 2, June, pp. 145-188.

Cheng 1998 — Cheng, C.K. (1998). *A survey of media servers*. Hong Kong University CSIS, November, ωωω

Cheriton 1984 — Cheriton, D.R. (1984). The V kernel: a software base for distributed systems. *IEEE Software*, Vol. 1 No. 2, pp. 19–42.

Cheriton 1985 — Cheriton, D.R. (1985). Preliminary thoughts on problem-oriented shared memory: a decentralized approach to distributed systems. *ACM Operating Systems Review*, Vol. 19, No. 4, pp. 26-33.

Cheriton 1986 — Cheriton, D.R. (1986). VMTP: A protocol for the next generation of communication systems.In *Proceedings SIGCOMM '86 Symposium on*

Communication Architectures and Protocols, ACM, pp. 406-15.

Cheriton and Mann 1989 — Cheriton, D. and Mann, T. (1989). Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions Computer Systems*, Vol. 7, No. 2, pp. 147–83.

Cheriton and Skeen 1993 — Cheriton, D. and Skeen, D. (1993). Understanding the limitations of causally and totally ordered communication. In *Proceedings 14th ACM Symposium on Operating System Principles*, Dec., pp. 44-57.

Cheriton and Zwaenepoel 1985 — Cheriton, D.R. and Zwaenepoel, W. (1985). Distributed process groups in the V kernel. *ACM Transactions Computer Systems*, Vol. 3, No. 2, pp. 77–107.

Cheswick and Bellovin 1994 — Cheswick, E.R. and Bellovin, S.M. (1994). *Firewalls and Internet Security*. Reading, Mass.: Addison-Wesley.

Choudhary *et al.* 1989 — Choudhary, A., Kohler, W., Stankovic, J. and Towsley, D. (1989). A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Transactions Software Engineering,* Vol. 15, No. 1.

Clark 1982 — Clark, D.D. (1982). *Window and Acknowledgement Strategy in TCP*, Internet RFC 813. 00000

Comer 1991 — Comer, D.E. (1991). *Internetworking with TCP/IP, Volume 1: Principles, Protocols and Architecture*. second edition. Englewood Cliffs, NJ: Prentice-Hall.

Comer 1995 — Comer, D.E. (1995). *The Internet Book*. Englewood Cliffs, NJ. Prentice-Hall.

Condict *et al.* 1994 — Condict, M., Bolinger, D., McManus, E., Mitchell, D. and Lewontin, S. (1994). *Microkernel modularity with integrated kernel performance.* Technical report, OSF Research Institute, Cambridge, Mass, April.

Coulouris *et al.* 1998 — Coulouris, G.F., Dollimore, J. B. and Roberts, M.(1998). Role and task-based access control in the PerDiS groupware platform. *Third ACM Workshop on Role-Based Access Control*, George Mason University, Washington DC, October 22-23. 00000

Cristian 1989 — Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing*, Vol. 3, pp. 146-58.

Cristian 1991a — Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Comms. ACM*, Vol. 34, No. 2.

Cristian 1991b — Cristian, F. (1991). Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, Springer Verlag, Vol. 4, pp 175-87.

Cristian and Fetzer 1994 — Cristian, F., and Fetzer, C. (1994). Probabilistic Internal Clock Synchronization. In *Proceedings 13th Symposium on Reliable Distributed Systems*, IEEE Computer Society Press, October 25-27, pp. 22-31.

Crow *et al.* 1997 — Crow, B., Widjaja, I. , Kim, J. and Sakai, P. (1997). IEEE 802.11 Wireless local area networks. *IEEE Communications Magazine*, Sept. 1997, pp. 116-26.

cryptography.org — *North American Cryptography Archives.* 00000

Curtin and Dolske 1998 — Kurtin, M. and Dolski, J. (1998). A brute force search of DES Keyspace.

;login: – the Newsletter of the USENIX Association, May.

Custer 1998 — Custer, H. (1998). *Inside Windows NT*, second edition. Microsoft Press.

Czerwinski *et al.* 1999 — Czerwinski, S., Zhao, B., Hodes, T., Joseph, A. and Katz, R. (1999). An architecture for a secure discovery service. In *Proceedings Fifth Annual International Conference on Mobile Computing and Networks.*

Dasgupta *et al.* 1991 — Dasgupta, P., LeBlanc Jr., R.J. Ahamad, M. and Ramachandran, U. (1991). The Clouds distributed operating system. *IEEE Computer*, Vol. 24, No. 11, pp. 34-44.

Davidson 1984 — Davidson, S.B. (1984). Optimism and consistency in partitioned database systems. *ACM Transactions Database Systems*, Vol. 9, No. 3, pp. 456-81.

Davidson *et al.* 1985 — Davidson, S.B., Garcia-Molina, H. and Skeen, D. (1985). Consistency in partitioned networks. *Computing Surveys*, Vol. 17, No.3, pp. 341-70.

DEC 1990 — Digital Equipment Corporation. (1990). *In Memoriam: J. C. R. Licklider 1915-1990*, Technical Report 61, DEC Systems Research Center.

Delgrossi *et al.* 1993 — Delgrossi, L., Halstrick, C., Hehmann, D., Herrtwich, R.G., Krone, O., Sandvoss, J. and Vogt, C. (1993). Media scaling for audiovisual communication with the Heidelberg transport system. *ACM Multimedia '93*, Anaheim, Calif.

Demers *et al.* 1989 — Demers, A., Keshav, S., Shenker, S. (1989). Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM '89.*

Denning and Denning 1977 — Denning, D. and Denning, P. (1977). Certification of programs for secure information flow. *Comms. ACM*, Vol. 20, No. 7, pp. 504-13.

Dertouzos 1974 — Dertouzos, M.L. (1974). Control robotics – the procedural control of physical processes. *IFIP Congress.*

Dierk and Allen 1999 — Dierk,T. and Allen, C. (1999). *The TLS Protocol Version 1.0*, Internet RFC 2246.

Diffie 1988 — Diffie, W. (1988). The first ten years of public-key cryptography. *Proceedings of the IEEE*, Vol. 76, No. 5, May 1988, pp. 560-77.

Diffie and Hellman 1976 — Diffie, W. and Hellman, M.E. (1976). New directions in cryptography. *IEEE Transactions Information Theory*, Vol. IT-22, pp. 644-54.

Diffie and Landau 1998 — Kahn, D. and Landau, S. (1998). *Privacy on the Line*. Cambridge, Mass: MIT Press.

Dijkstra 1959 — Dijkstra, E.W. (1959). A note on two problems in connection with graphs. *Numerische Mathematic*, Vol. 1, pp. 269-71.

Dolev and Malki 1996 — Dolev, D. and Malki, D. (1996). The Transis approach to high availability cluster communication. *Comms. ACM*, Vol. 39, No. 4, pp. 64-70.

Dolev and Strong 1983 — Dolev, D. and Strong, H. (1983). Authenticated algorithms for byzantine agreement. *SIAM Journal of Computing*, Vol. 12, No. 4, pp. 656-66.

Dolev *et al.* 1986 — Dolev, D., Halpern, J., and Strong, H. (1986). On the possibility and impossibility of achieving clock synchronization. *Journal of Computing Systems Science* 32, 2 (Apr.), pp. 230-50.

Dorcey 1995 — Dorcey, T. (1995). CU-SeeMe Desktop Video Conferencing Software, *Connexions*, vol. 9, no. 3 (March).

Douceur and Bolosky 1999 — Douceur, J.R. and Bolosky, W. (1999). Improving responsiveness of a stripe-scheduled media server. *SPIE Proceedings*, Vol. 3654. *Multimedia Computing and Networking*. pp. 192-203. ωωω

Douglis and Ousterhout 1991 — Douglis, F. and Ousterhout, J. (1991). Transparent process migration: design alternatives and the Sprite implementation, *Software – Practice and Experience*, Vol. 21, No. 8, pp. 757-89.

Draves 1990 — Draves, R. (1990). A revised IPC interface, In *Proceedings USENIX Mach Workshop*, pp. 101-21, October.

Draves *et al.* 1989 — Draves, R.P., Jones, M.B. and Thompson, M.R. (1989). *MIG - the Mach Interface Generator*. Technical Report, Dept. of Computer Science, Carnegie-Mellon University.

Druschel and Peterson 1993 — Druschel, P. and Peterson, L. (1993). Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings 14th ACM Symposium on Operating System Principles*, pp. 189-202.

Dubois *et al.* 1988 — Dubois, M., Scheurich, C. and Briggs, F.A. (1988). Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, Vol. 21, No. 2, pp. 9-21.

Dwork *et al.* 1988 — Dwork, C., Lynch, N. and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, Vol. 35, No. 2, pp. 288-323.

Eager *et al.* 1986 — Eager, D., Lazowska, E. and Zahorjan, J. (1986). Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, pp. 662-675.

EFF 1998 — Electronic Frontier Foundation (1998). *Cracking DES, Secrets of Encryption Research, Wiretap Politics & Chip Design*. Sebastapol Calif.: O'Reilly & Associates.

Eisler *et al.* 1997 — Eisler, M., Chiu, A, and L. Ling, L. (1997). *RPCSEC_GSS Protocol Specification*. Internet RFC 2203. September. ωωω

El Abbadi *et al.* 1985 — El Abbadi, A., Skeen, D. and Cristian, C. (1985). An efficient fault-tolerant protocol for replicated data management. In *4th Annual ACM SIGACT/SIGMOD Symposium on Principles of Data Base Systems*, Portland, Ore.

Ellis *et al.* 1991 — Ellis, C., Gibbs, S. and Rein, G. (1991). Groupware – some issues and experiences. *Comms. ACM*, Vol. 34, No. 1, pp. 38–58.

Ellison 1996 — Ellison, C. (1996). Establishing identity without certification authorities. In *6th USENIX Security Symposium*, San Jose, July 22-25. ωωω

Ellison *et al.* 1999 — Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B. and Ylonen, T. (1999). *SPKI Certificate Theory*. Internet RFC 2693, September. ωωω

Farley 1998 — Farley, J. (1998). *Java Distributed Computing*. Cambridge, Mass: O'Reilly.

Farrow 2000 — Farrow, R. (2000). Distributed denial of service attacks - how Amazon, Yahoo, eBay and others were brought down. *Network Magazine*, April. ωωω

Ferrari and Verma 1990 — Ferrari, D. and Verma, D. (1990). A scheme for real-time channel

establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 4.

Ferreira *et al.* 2000 — Ferreira, P., Shapiro; M., Blondel, X., Fambon, O., Garcia, J., Kloostermann, S., Richer, N., Roberts, M., Sandakly, F., Coulouris, G., Dollimore, J., Guedes, P., Hagimont, D. and Krakowiak, S. (2000). PerDiS: Design, Implementation, and Use of a PERsistent DIstributed Store. In *LNCS 1752: Advances in Distributed Systems*. Springer-Verlag Berlin, Heidelberg, New York. pp 427-53. ωωω

Fidge 1991 — Fidge, C. (1991). Logical Time in Distributed Computing Systems. *IEEE Computer*, Vol. 24, No. 8, pp. 28-33.

P. and Berners-Lee T. (1999). *Hypertext Transfer Protocol – HTTP/1.1*. Internet RFC 2616. ωωω

Fischer 1983 — Fischer, M. (1983). The Consensus Problem in Unreliable Distributed Systems (a Brief Survey). In M. Karpinsky, ed., *Foundations of Computation Theory*, Vol. 158 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 127-140. Yale University Technical Report YALEU/DCS/TR-273.

Fischer and Lynch 1982 — Fischer, M. and Lynch, N. (1982). A lower bound for the time to assure interactive consistency. *Inf. Process. Letters*, Vol. 14, No. 4, June, pp. 183-6.

Fischer and Michael 1982 — Fischer, M.J. and Michael, A. (1982). Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proceedings Symposium on Principles of Database Systems*, ACM, pp. 70–5.

Fischer *et al.* 1985 — Fischer, M., Lynch, N. and Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, Vol. 32, No. 2, Apr., pp. 374-82.

Fitzgerald and Rashid 1986 — Fitzgerald, R. and Rashid, R.F. (1986). The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions Computer Systems*, Vol. 4, No. 2, pp. 147–77.

Flanagan 1997 — Flanagan, D. (1997). *Java in a Nutshell*. Cambridge, England: O'Reilly.

Fleisch and Popek 1989 — Fleisch, B. and Popek, G. (1989). Mirage: a coherent distributed shared memory design. In *Proceedings 12th ACM Symposium on Operating System Principles*, December, pp. 211-23.

Floyd 1986 — Floyd, R. (1986). *Short term file reference patterns in a UNIX environment*. Technical Rep. TR 177, Rochester, NY: Dept. of Computer Science, University of Rochester.

Floyd and Johnson 1993 — Floyd, S. and Jacobson, V. (1993). The Synchronization of Periodic Routing Messages. *ACM Sigcomm '93 Symposium*.

Floyd *et al.* 1997 — Floyd, S., Jacobson, V., Liu, C., McCanne, S. and Zhang, L. (1997). A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, Vol. 5, No. 6, pp. 784-803.

Ford and Fulkerson 1962 — Ford, L.R. and Fulkerson, D.R. (1962). *Flows in Networks*. Princeton, NJ: Princeton University Press.

Fox *et al.* 1997 — Fox, A., Gribble, S., Chawathe, Y., Brewer, E. and Gauthier, P. (1997). Cluster-based scalable network services. *Proceedings of the 16th ACM*

Cluster-based scalable network services. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 78-91.

Garay and Moses 1993 — Garay, J., and Moses, Y. (1993). Fully polynomial Byzantine agreement in *t*+1 rounds. In *Proceedings 25th ACM symposium on theory of computing*, ACM Press, pp. 31-41, May.

Garcia-Molina 1982 — Garcia-Molina, H. (1982). Elections in Distributed Computer Systems. *IEEE Transactions on Computers*, Vol. C-31, No. 1, pp. 48-59.

Garcia-Molina and Spauster 1991 — Garcia-Molina, H. and Spauster, A. (1991). Ordered and Reliable Multicast Communication. *ACM Transactions Computer Systems*, Vol. 9, No. 3, pp. 242–71.

Garfinkel 1994 — Garfinkel, S. (1994). *PGP: Pretty Good Privacy*, O'Reilly.

Gharachorloo *et al.* 1990 — Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J. (1990). Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th. Annual International Symposium on Computer Architecture*, May, pp. 15–26.

Gibbs and Tsichritzis 1994 — Gibbs, S.J. and Tsichritzis, D.C. (1994). *Multimedia Programming*. Addison-Wesley.

Gifford 1979 — Gifford, D.K. (1979). Weighted voting for replicated data. In *Proceedings 7th Symposium on Operating Systems Principles*, ACM, pp. 150–62.

Glassman *et al.* 1995 — Glassman, S., Manasse, M., Abadi, M., Gauthier, P. and Sobalvarro, P. (1995). The Millicent Protocol for Inexpensive Electronic Commerce. *Fourth International WWW Conference*, December.

Gokhale and Schmidt 1996 — Gokhale, A. and Schmidt, D. (1996). Measuring the Performance of Communication Middleware on High-Speed Networks. *Proceedings of SIGCOMM '96*, ACM, pp. 306-17.

Golding and Long 1993 — Golding, R. and Long, D. (1993). *Modeling replica divergence in a weak-consistency protocol for global-scale distributed databases.* Technical report UCSC-CRL-93-09, Computer and Information Sciences Board, University of California, Santa Cruz.

Gong 1989 — Gong, L. (1989). A Secure Identity-Based Capability System. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, Calif., May, pp.56-63.

Goodman 1989 — Goodman, J. (1989). *Cache Consistency and Sequential Consistency.* Technical Report 61, SCI Committee.

Gordon 1984 — Gordon, J. (1984). *The Story of Alice and Bob.*

Govindan and Anderson 1991 — Govindan, R. and Anderson, D.P. (1991). Scheduling and IPC Mechanisms for Continuous Media. *ACM Operating Systems Review*, Vol. 25, No. 5, pp. 68-80.

Gray 1978 — Gray, J. (1978). Notes on database operating systems. In *Operating Systems: an Advanced Course. Lecture Notes in Computer Science*, Vol. 60, pp. 394–481, Springer-Verlag.

Guerraoui *et al.* 1998 — Guerraoui, R., Felber, P., Garbinato, B. and Mazouni, K. (1998). System support for object groups. *In Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications* (OOPSLA'98).

Gusella and Zatti 1989 — Gusella, R. and Zatti, S. (1989). The accuracy of clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions Software Engineering*, Vol. 15, pp. 847-53.

Guttman 1999 — Guttman, E. (1999). Service Loocation Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, Vol. 3, No. 4, pp. 71-80.

Hadzilacos and Toueg 1994 — Hadzilacos, V. and Toueg, S. (1994). *A Modular Approach to Fault-tolerant Broadcasts and Related Problems*, Technical report, Dept. of Computer Science, University of Toronto.

Härder 1984 — Härder, T. (1984). Observations on Optimistic Concurrency Control Schemes. *Information Systems*, Vol. 9, No. 2, pp. 111-20.

Härder and Reuter 1983 — Härder, T. and Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *Computing Surveys*, Vol. 15, No. 4.

Härtig *et al.* 1997 — Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J. (1997). The performance of kernel-based systems. In *Proceedings 16th ACM Symposium on Operating System Principles*, pp. 66-77.

Hartman and Ousterhout 1995 — Hartman, J. and Ousterhout, J. (1995). The Zebra Striped Network File System. *ACM Trans. on Computer Systems*, Vol. 13 , No. 3, August, pp. 274-310.

Hayton *et al.* 1998 — Hayton R., Bacon J. and Moody K. (1998). OASIS: Access Control in an Open, Distributed Environment, In *Proceedings IEEE Symposium on Security and Privacy*, Oakland, Calif., pp3-14, May. 00000

Hedrick 1988 — Hedrick, R. (1988). *Routing Information Protocol*, Internet RFC 1058. 00000

Henning 1998 — Henning, M. (1998). Binding, Migration and Scalability in CORBA, *Comms. ACM*, October, Vol. 41, No. 10. pp. 62-71.

Henning and Vinoski 1999 — Henning, M. and Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Reading, Mass.: Addison-Wesley.

Herlihy 1986 — Herlihy, M. (1986). A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions Computer Systems*, Vol. 4, No. 1, pp. 32-53.

Herlihy and Wing 1990 — Herlihy, M. and Wing, J. (1990). On Linearizability: a correctness condition for concurrent objects. *ACM Transactions on programming languages and systems*, Vol. 12, No. 3, (July), pp. 463-92.

Herrtwich 1995 — Herrtwich, R.G. (1995). Achieving Quality of Service for Multimedia Applications. *ERSADS '95, European Research Seminar on Advanced Distributed Systems*, l'Alpe d'Huez, France, April.

Hirsch 1997 — Hirsch, F.J. (1997). Introducing SSL and Certificates using SSLeay. *World Wide Web Journal*, Vol. 2, No. 3, Summer. 00000

Howard *et al.* 1988 — Howard, J.H., Kazar, M.L., Menees, S.G, Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N. and West, M.J. (1988). Scale and Performance in a Distributed File System. *ACM Transactions Computer Systems*, Vol. 6, No. 1, pp. 51-81.

Huang *et al.* 2000 — Huang, A., Ling, B., Barton, J. and Fox, A. (2000). Running the Web backwards: appliance data services. *Proceedings 9th international World Wide Web conference.* 00000

Huitema 1995 — Huitema, C. (1995). *Routing in the Internet*. Englewood Cliffs, NJ: Prentice-Hall.

Huitema 1998 — Huitema, C. (1998). *IPv6 – the New Internet Protocol*. Upper Saddle River, NJ: Prentice-Hall.

Hunter and Crawford 1998 — Hunter, J. and Crawford, W. (1998). *Java Servlet Programming*, O'Reilly.

Hutchinson and Peterson 1991 — Hutchinson, N. and Peterson, L. (1991). The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, Vol. 17, No. 1, pp. 64-76.

Hutto and Ahamad 1990 — Hutto, P. and Ahamad, M. (1990). Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings 10th International Conference on Distributed Computer Systems*, IEEE, pp. 302-11.

Hyman *et al.* 1991 — Hyman, J., Lazar, A.A. and Pacifici, G. (1991). MARS – The MAGNET-II Real-Time Scheduling Algorithm. *ACM SIGCOM '91*, Zurich.

IEEE 1985a — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – CSMA/CD Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.3, IEEE Computer Society.

IEEE 1985b — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – Token Bus Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.4, IEEE Computer Society.

IEEE 1985c — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – Token Ring Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.5, IEEE Computer Society.

IEEE 1990 — Institute of Electrical and Electronic Engineers (1990). *IEEE Standard 802: Overview and Architecture*. American National Standard ANSI/IEEE 802, IEEE Computer Society.

IEEE 1994 — Institute of Electrical and Electronic Engineers (1994). *Local and metropolitan area networks – Part 6: Distributed Queue Dual Bus (DQDB) access method and physical layer specifications*. American National Standard ANSI/IEEE 802.6, IEEE Computer Society.

IEEE 1999 — Institute of Electrical and Electronic Engineers (1999). *Local and metropolitan area networks – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, American National Standard ANSI/IEEE 802.11, IEEE Computer Society.

Iftode *et al.* 1996 — Iftode, L., Singh J. and Li, K. (1996). Scope consistency: a bridge between release consistency and entry consistency. In *Proceedings 8th annual ACM symposium on Parallel algorithms and architectures*. pp. 277-87.

info.isoc.org — Zakon, R.H. *Hobbes' Internet Timeline v5.0.* ωωω

international pgp — *The International PGP Home Page.* ωωω

ISO 1992 — International Standards Organization (1992). *Basic Reference Model of Open Distributed Processing, Part 1: Overview and guide to use.*

ISO/IEC JTC1/SC212/WG7 CD 10746-1, International Standards Organization, 1992.

ITU/ISO 1997 — ITU/ISO (1997). *Recommendation X.500 (08/97): Open Systems Interconnection - The Directory: Overview of concepts, models and services*. International Telecommunication Union.

java.sun.com I — Sun Microsystems. *Java Remote Method Invocation.*

java.sun.com II — Sun Microsystems. *Java Object Serialization Specification.*

java.sun.com III — Sun Microsystems. *Servlet Tutorial.*

java.sun.com IV — Jordan, M. and Atkinson, M. (1999). *Orthogonal Persistence for the Java Platform - Draft Specification*. Sun Microsystems Laboratories. Palo Alto, Calif.

java.sun.com V — Sun Microsystems, *Java Security API.*

java.sun.com VI — Sun Microsystems Inc. (1999). *JavaSpaces technology.*

Johnson and Zwaenepoel 1993 — Johnson, D. and Zwaenepoel, W. (1993). The Peregrine High-performance RPC System. *Software–Practice and Experience*, Vol. 23, No. 2, pp. 201-21.

Jordan 1996 — Jordan, M. (1996). Early Experiences with Persistent Java. In *Proceedings first international workshop on persistrence and Java.* Glasgow, Scotland.

Joseph *et al.* 1997 — Joseph, A., Tauber, J. and Kaashoek, M. (1997). Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, Vol. 46, No. 3, pp. 337-52.

Jul *et al.* 1988 — Jul, E., Levy, H., Hutchinson, N. and Black, A. (1988). Fine-grained Mobility in the Emerald System. *ACM Transactions Computer Systems*, Vol. 6, No. 1, pp. 109-33.

Kaashoek and Tanenbaum 1991 — Kaashoek, F. and Tanenbaum, A. (1991). Group Communication in the Amoeba Distributed Operating System. In *Proceedings 11th International Conference on Distributed Computer Systems*, pp. 222-30.

Kaashoek *et al.* 1989 — Kaashoek, F., Tanenbaum, A., Flynn Hummel, S. and Bal, H. (1989). An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, Vol. 23, No. 4, pp. 5-20.

Kaashoek *et al.* 1997 — Kaashoek, M., Engler, D., Ganzer, G., Briceño, H., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J. and Mackenzie, K. (1997). Application performance and flexibility on exokernel systems. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 52-65.

Kahn 1967 — Kahn, D. (1967). *The Codebreakers: The Story of Secret Writing.* New York: Macmillan.

Kahn 1983 — Kahn, D. (1983). *Kahn on Codes*. New York: Macmillan.

Kahn 1991 — Kahn, D, (1991). *Seizing the Enigma*. Boston: Houghton Mifflin.

Kehne *et al.* 1992 — Kehne, A., Schonwalder, J. and Langendorfer, H. (1992). A Nonce-based Protocol for Multiple Authentications. *ACM Operating Systems Review*, Vol. 26, No. 4, pp. 84-9.

Keith and Wittle 1993 — Keith, B.E. and Wittle, M. (1993). LADDIS: The Next Generation in NFS File Server Benchmarking, *Summer USENIX Conference*

Proceedings. USENIX Association, Berkeley, Calif, June.

Keleher *et al.* 1992 — Keleher, P., Cox, A. and Zwaenepoel, W. (1992). Lazy consistency for software distributed shared memory. In *Proceedings 19th Annual International Symposium on Computer Architecture*. pp. 13-21, May 1992.

Kessler and Livny 1989 — Kessler, R.E. and Livny, M. (1989). An Analysis of Distributed Shared Memory Algorithms, In *Proceedings 9th International Conference Distributed Computing Systems*. IEEE, pp. 98–104.

Kille 1992 — Kille, S. (1992). *Implementing X.400 and X.500: The PP and QUIPU Systems*. Artech House.

Kindberg 1995 — Kindberg, T. (1995). A Sequencing Service for Group Communication (abstract), In *Proceedings 14th annual ACM Symposium on Principles of Distributed Computing*, p. 260. Technical Report No. 698, Queen Mary and Westfield College Dept. of CS, 1995. ωωω

Kindberg *et al.* 1996 — Kindberg, T., Coulouris, G., Dollimore, J. and Heikkinen, J. (1996). Sharing objects over the Internet: the Mushroom approach. In *Proceedings IEEE Global Internet 1996*, London, Nov., pp. 67-71.

Kistler and Satyanarayanan 1992 — Kistler, J.J. and Satyanarayanan, M. (1992). Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 3–25.

Kleinrock 1961 — Kleinrock, L. (1961). *Information Flow in Large Communication Networks*, MIT, RLE Quarterly Progress Report, July.

Kleinrock 1997 — Kleinrock, L. (1997). Nomadicity: anytime, anywhere in a disconnected world. *Mobile Networks and Applications*, Vol. 1, No. 4, pp. 351-7.

Kohl and Neuman 1993 — Kohl, J. and Neuman, C. (1993). *The Kerberos Network Authentication Service (V5)*, Internet RFC 1510, September. ωωω

Konstantas *et al.* 1997 — Konstantas, D., Orlarey, Y., Gibbs, S. and Carbonel, O. (1997). Distributed Music Rehearsal. In *Proceedings International Computer Music Conference 97*. ωωω

Kopetz and Verissimo 1993 — Kopetz, H. and Verissimo, P. (1993). Real Time and Dependability Concepts. in Mullender ed, *Distributed Systems*, second edition , Addison-Wesley.

Kopetz *et al.* 1989 — Kopetz, H. Damm, A., Koza, C., Mulazzani, M., Schwabl, W. Senft, C and Zainlinger, R. (1989). Distributed Fault-Tolerant Real-Time Systems – The MARS Approach. *IEEE Micro*, Vol. 9, No. 1.

Kshemkalyani and Singhal 1991 — Kshemkalyani, A. and Singhal, M. (1991). Invariant-Based Verification of a Distributed Deadlock Detection Algorithm. *IEEE Transactions on Software Engineering*, Vol. 17, No. 8. August.

Kshemkalyani and Singhal 1994 — Kshemkalyani, A. and Singhal, M. (1994). On Characterisation and Correctness of Distributed Deadlock detection, *Journal of Parallel and Distributed Computing*, Vol. 22, pp. 44-59.

Kung and Robinson 1981 — Kung, H.T. and Robinson, J.T. (1981). Optimistic methods for concurrency control. *ACM Transactions on Database Systems*, Vol. 6, No. 2, pp. 213–26.

Kurose and Ross 2000 — Kurose, J.F. and Ross, K.W. (2000). *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison Wesley

Longman.

Ladin *et al.* 1992 — Ladin, R., Liskov, B., Shrira, L. and Ghemawat, S. (1992). Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, Vol. 10, No. 4, pp. 360–91.

Lai 1992 — Lai, X. (1992). On the Design and Security of Block Ciphers, *ETH Series in Information Processing*, Vol. 1, Konstanz: Hartung-Gorre Verlag.

Lai and Massey 1990 — Lai, X. and Massey, J. (1990). A proposal for a new Block Encryption Standard. *Advances in Cryptology–Eurocrypt '90* In *Proceedings*, Springer-Verlag, pp. 389–404.

Lamport 1978 — Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, Vol. 21, No. 7, pp. 558–65.

Lamport 1979 — Lamport, L. (1979). How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions Computers*, Vol. C-28, No. 9, pp. 690–1.

Lamport 1986 — Lamport, L. (1986). On interprocess communication, parts I and II. *Distributed Computing*, Vol. 1, No. 2, pp. 77-101.

Lamport *et al.* 1982 — Lamport, L., Shostak, R. and Pease, M. (1982). Byzantine Generals Problem. *ACM Transactions Programming Languages and Systems*, Vol. 4, No. 3, pp. 382–401.

Lampson 1971 — Lampson, B. (1971). Protection. In *Proceedings 5th Princeton Conference on Information Sciences and Systems*, Princeton, p. 437. Reprinted in *ACM Operating Systems Review*. Vol. 8, No. 1, January, p. 18.

Lampson 1981a — Lampson, B.W. (1981). Atomic Transactions. In *Distributed systems: Architecture and Implementation. Lecture Notes in Computer Science 105*, pp. 254–9. Berlin: Springer-Verlag.

Lampson 1986 — Lampson, B.W. (1986). Designing a Global Name Service.In *Proceedings 5th ACM Symposium Principles of Distributed Computing*, pp. 1–10, August.

Lampson *et al.* 1992 — Lampson, B.W., Abadi, M., Burrows, M. and Wobber, E. (1992). Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, Vol. 10, No. 4, pp. 265–310.

Lea *et al.* 1993 — Lea, R., Jacquemot, C. and Pillevesse, E. (1993). COOL: system support for distributed programming. *Comms. ACM*, Vol. 36, No. 9, pp. 37–46.

Leach *et al.* 1983 — Leach, P.J., Levine, P.H., Douros, B.P., Hamilton, J.A., Nelson, D.L. and Stumpf, B.L. (1983). The architecture of an integrated local network. *IEEE J. Selected Areas in Communications*, Vol. SAC-1, No. 5, pp. 842–56.

Lee and Thekkath 1996 — Lee, E.K. and Thekkath, C.A. (1996). Petal: Distributed Virtual Disks, In *Proc. 7th Intl. Conf. on Architectural Support for Prog. Langs. and Operating Systems*, October, pp. 84-96.

Lee *et al.* 1996 — Lee C., Rajkumar, R. and Mercer C. (1996). Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. In *Proceedings Multimedia Japan '96.*

Leffler *et al.* 1989 — Leffler, S., McKusick, M., Karels, M. and Quartermain J. (1989). *The Design and Implementation of the 4.3 BSD UNIX Operating System.*

Reading, Mass: Addison-Wesley.

Leiner 1997 — Leiner, B.M., Cerf, V.G., Clark, D.D., Kahn, R.E., Kleinrock, L., Lynch, D.C., Postel, J., Roberts, L.G. and Wolff, S. (1997). A Brief History of the Internet, *Comms. ACM*, Vol. 40, No. 1, Feb., pp. 102-108.

Leland *et al.* 1993 — Leland, W. E., Taqqu, M.S., Willinger, W. and Wilson, D.V. (1993). On the Self-Similar Nature of Ethernet Traffic. *ACM SIGCOMM '93*, San Francisco.

Lenoski *et al.* 1992 — Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.D., Gupta, A., Hennessy, J., Horowitz, M. and Lam, M.S. (1992). The Stanford Dash multiprocessor. *IEEE Computer*, Vol. 25, No. 3, pp. 63–79.

Leslie *et al.* 1996 — Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R. and Hyden, E. (1996). The design and implementation of an operating system to support distributed multimedia applications, *ACM Journal of Selected Areas in Communication*, Vol. 14, No. 7, pp. 1280-97.

Li and Hudak 1989 — Li, K. and Hudak, P. (1989). Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321–59.

Liedtke 1996 — Liedtke, J. (1996). Towards real microkernels, *Comms. ACM*, Vol. 39, No. 9, pp. 70-7.

Linux AFS — *The Linux AFS FAQ*.

Lipton and Sandberg 1988 — Lipton, R. and Sandberg, J. (1988). *PRAM: A scalable shared memory*. Technical Report CS-TR-180-88, Princeton University.

Liskov 1988 — Liskov, B. (1988). Distributed programming in Argus. *Comms. ACM*, Vol, 31, No. 3, pp. 300–12.

Liskov 1993 — Liskov, B. (1993). Practical uses of synchronized clocks in distributed systems, *Distributed Computing*, Vol. 6, No. 4, pp. 211-19.

Liskov and Scheifler 1982 — Liskov, B. and Scheifler, R.W. (1982). Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions Programming Languages and Systems*, Vol. 5, No. 3, pp. 381–404.

Liskov and Shrira 1988 — Liskov, B. and Shrira, L. (1988). Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings SIGPLAN '88 Conference Programming Language Design and Implementation*. Atlanta.

Liskov *et al.* 1991 — Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., Williams, M. (1991). Replication in the Harp File System. In *Proceedings 13th ACM Symposium on Operating System Principles*, pp. 226–38.

Liu and Albitz 1998 — Liu, C. and Albitz, P. (1998). *DNS and BIND*, third edition. O'Reilly.

Liu and Layland 1973 — Liu, C.L. and Layland, J.W. (1973). Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1.

Loepere 1991 — Loepere, K. (1991). *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie-Mellon University.

Lundelius and Lynch 1984 — Lundelius, J. and Lynch, N. (1984). An Upper and Lower Bound for Clock Synchronization. *Information and Control* 62, 2/3 (Aug./Sep.),

pp. 190-204.

Lynch 1996 — Lynch, N. (1996). *Distributed Algorithms*, Morgan Kaufmann.

Ma 1992 — Ma, C. (1992). *Designing a Universal Name Service*. Technical Report 270, University of Cambridge.

Macklem 1994 — Macklem, R. (1994). Not Quite NFS: Soft Cache Consistency for NFS. *Proceedings of the Winter '94 USENIX Conference*, San Francisco, Calif., January, pp. 261-278.

Maekawa 1985 — Maekawa, M. (1985). A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pp. 145-159.

Maffeis 1995 — Maffeis, S. (1995). Adding group communication and fault tolerance to CORBA. In *Proceedings of the 1995 USENIX conference on object-oriented technologies.*

Malkin 1993 — Malkin, G. (1993). *RIP Version 2 – Carrying Additional Information*, Internet RFC 1388.

Marsh *et al.* 1991 — Marsh, B., Scott, M., LeBlanc, T. and Markatos, E. (1991). First-class User-level Threads. In *Proceedings 13th ACM Symposium on Operating System Principles*, pp. 110–21.

Marzullo and Neiger 1991 — Marzullo, K., and Neiger, G. (1991). Detection of global state predicates, In *Proceedings 5th International Workshop on Distributed Algorithms*, Toug, S., Spirakis, P. and Kirousis, L., eds, Springer-Verlag, pp. 254-72.

Mattern 1989 — Mattern, F. (1989). Virtual Time and Global States in Distributed Systems, In *Proceedings Workshop on Parallel and Distributed Algorithms*. Cosnard, M. *et al.* (eds), Amsterdam: North-Holland, pp. 215-26.

mbone — *MBone Software Archives.*

McGraw and Felden 1999 — McGraw, G. and Felden, E. (1999). *Securing Java*. John Wiley & Sons.

Melliar-Smith *et al.* 1990 — Melliar-Smith, P., Moser, L. and Agrawala, V. (1990). Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 17-25.

Menezes 1993 — Menezes, A. (1993). *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers.

Menezes *et al.* 1997 — Menezes, A., van Oorschot, O. and Vanstone, S. (1997). *Handbook of Applied Cryptography*. CRC Press.

Metcalfe and Boggs 1976 — Metcalfe, R.M. and Boggs, D.R. (1976). Ethernet: distributed packet switching for local computer networks. *Comms. ACM*, Vol. 19, pp. 395–403.

mice.ed.ac.uk — Handley, M.. *The sdr Session Directory.*

Mills 1995 — Mills, D. (1995). Improved Algorithms for Synchronizing Computer Network Clocks, *IEEE Transactions Networks*, June, pp. 245-54.

Milojicic *et al.* 1999 — Milojicic, J., Douglis, F. and Wheeler, R. (1999). *Mobility, Processes, Computers and Agents*, Reading: Addison-Wesley.

Minnich and Farber 1989 — Minnich, R. and Farber, D. (1989). The Mether System: a Distributed Shared Memory for SunOS 4.0. In *Proceedings Summer 1989 Usenix*

Conference.

Mitchell and Dion 1982 — Mitchell, J.G. and Dion, J. (1982). A comparison of two network-based file servers. *Comms. ACM*, Vol. 25, No. 4, pp. 233–45.

Mitchell *et al.* 1992 — Mitchell, C.J., Piper, F. and Wild, P. (1992). Digital Signatures. In *Contemporary Cryptology*. Simmons, G.J. ed., New York: IEEE Press.

Mogul 1994 — Mogul, J.D. (1994). Recovery in Spritely NFS, *Computing Systems*, Vol.7, No. 2.

Mok 1985 — Mok, A.K. (1985). SARTOR – A Design Environment for Real-Time Systems. *Ninth IEEE COMP-SAC.*

Morin 1997 — Morin, R. (ed.) (1997). *MkLinux: Microkernel Linux for the Power Macintosh*. Prime Time Freeware.

Morris *et al.* 1986 — Morris, J., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S. and Smith, F.D. (1986). Andrew: a distributed personal computing environment. *Comms. ACM*, Vol. 29, No. 3, pp. 184–201.

Mosberger 1993 — Mosberger, D. (1993). *Memory Consistency Models*. Technical Report 93/11 , University of Arizona.

Moser *et al.* 1994 — Moser, L., Amir, Y., Melliar-Smith, P. and Agarwal, D. (1994). Extended Virtual Synchrony. In *Proceedings 14th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pp. 56-65.

Moser *et al.* 1996 — Moser, L., Melliar-Smith, P., Agarwal, D., Budhia, R., and Lingley-Papadopoulos, C. (1996). Totem: a Fault-Tolerant Multicast Group Communication System. *Comms. ACM*, Vol. 39, No. 4, pp. 54-63.

Moser *et al.* 1998 — Moser, L., Melliar-Smith, P. and Narasimhan, P. (1998). Consistent object replication in the Eternal system. *Theory and practice of object systems*, Vol. 4, No. 2.

Moss 1985 — Moss, E. (1985). *Nested Transactions, An Approach to Reliable Distributed Computing*. MIT Press.

multimedia index — Gibbs, S. and Szentivanyi, S., *Index to Multimedia Information Sources*.

Myers and Liskov 1997 — Myers, A.C. and Liskov, B. (1997). A Decentralized Model for Information Flow Control, *ACM Operating Systems Review*, Vol. 31, No. 5, pp. 129-42, December.

Nagle 1984 — Nagle, J. (1984). Congestion Control in TCP/IP Internetworks, *Computer Communications Review*, Vol. 14, pp. 11-17, October.

Nagle 1987 — Nagle, J. (1987). On Packet Switches with Infinite Storage. *IEEE Transactions on Communications*, Vol. 35, No. 4.

National Bureau of Standards 1977 — National Bureau of Standards (1977). *Data Encryption Standard (DES)*. Federal Information Processing Standards No. 46, Washington DC: US National Bureau of Standards.

Needham 1993 — Needham, R. (1993). Names. In *Distributed Systems, an Advanced Course*. (Mullender, S., ed.), second Edition. Wokingham, England: ACM Press/Addison-Wesley. pp. 315-26.

Needham and Schroeder 1978 — Needham, R.M. and Schroeder, M.D. (1978). Using encryption for authentication in large networks of computers. *Comms. ACM*, Vol. 21,

pp. 993–9.

Nelson *et al.* 1988 — Nelson, M.N., Welch, B.B. and Ousterhout, J.K. (1988). Caching in the Sprite Network File System, *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 134-154.

Netscape 1996 — Netscape Corporation (1996). *SSL 3.0 Specification*. ωωω

Neuman *et al.* 1999 — Neuman, B.C., Tung, B. and Wray, J. (1999). *Public Key Cryptography for Initial Authentication in Kerberos*, Internet Draft ietf-cat-kerberos-pk-init-09, July. ωωω

Neumann and Ts'o 1994 — Neuman, B.C. and Ts'o, T. (1994). Kerberos: An Authentication Service for Computer Networks, *IEEE Communications*, vol. 32, no. 9, pp. 33-38. Sept.. ωωω

Nielson *et al.* 1997 — Nielsen, H., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. and Lilley, C. (1997). Network Performance Effects of HTTP/1.1, CSS1, and PNG. *Proceedings SIGCOMM '97.*

NIST 1995 — National Institute for Standards and Technology (1995). *Secure Hash Standard*. NIST FIPS PUP 180-1, US Department of Commerce. ωωω

NIST 1999 — National Institute for Standards and Technology (1999). *AES – a Crypto Algorithm for the Twenty-first Century*, US Department of Commerce. ωωω

now.cs.berkeley.edu — *The Berkeley NOW project home page*. ωωω

Oaks and Wong 1999 — Oaks, S. and Wong, H. (1999). *Jave Threads* (second edition), O'Reilly.

OMG 1997a — Object Management Group. (1997). *Concurrency Control Service Specification*, Framingham, Mass: OMG. ωωω

OMG 1997b — Object Management Group (1997). *Naming Service Specification*. Framingham, Mass: OMG. ωωω

OMG 1997c — Object Management Group (1997). *Event Service Specification*. Framingham, Mass: OMG. ωωω

OMG 1997d — Object Management Group (1997). *The CORBA IDL Specification*. Framingham, Mass: OMG. ωωω

OMG 1997e — Object Management Group, (1997). *Object Transaction Service Specification*. Framingham, Mass: OMG. ωωω

OMG 1998a — Object Management Group (1998). *CORBA/IIOP 2.3.1 Specification*. Framingham, Mass: OMG. ωωω

OMG 1998b — Object Management Group (1998). *CORBA Security Service Specification*. Framingham, Mass: OMG. ωωω

OMG 1998c — Object Management Group (1998). *Notification Service Specification*. Framingham, Mass: OMG. Technical report telecom/98-06-15. ωωω

OMG 1998d — Object Management Group (1998). *CORBA Messaging*. Framingham, Mass: OMG. ωωω

OMG 1998e — Object Management Group(1998). *Objects by Value*. Framingham, Mass: OMG. ωωω

Omidyar and Aldridge 1993 — Omidyar, C.G. and Aldridge, A. (1993). Introduction to SDH/SONET. *IEEE Communications Magazine*, Vol. 31, pp. 30-3, Sept.

Oppen and Dalal 1983 — Oppen, D. C. and Dalal Y.K. (1983). The Clearinghouse: a decentralized agent for locating named objects in a distributed

environment. *ACM Trans. on Office Systems*, Vol. 1, pp. 230-53.

Orfali *et al.* 1996 — Orfali, R., Harkey, D. and Edwards, J. (1996). *The Essential Distributed Objects Survival Guide*. New York: Wiley.

Organick 1972 — Organick, E.I. (1972). *The MULTICS System: An Examination of its Structure*. Cambridge, Mass: MIT Press.

Orman *et al.* 1993 — Orman, H., Menze, E., O'Malley, S. and Peterson, L. (1993). A fast and general implementation of Mach IPC in a Network. In *Proceedings Third USENIX Mach Conference*, April.

OSF 1997 — *Introduction to OSF DCE*. The Open Group.

Ousterhout *et al.* 1985 — Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M. and Thompson, J. (1985). A Trace-driven analysis of the UNIX 4.2 BSD file system. In *10th ACM Symposium Operating System Principles*.

Ousterhout *et al.* 1988 — Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B. (1988). The Sprite Network Operating System. *IEEE Computer*, Vol. 21, No. 2, pp. 23–36.

Parker 1992 — Parker, B. (1992). *The PPP AppleTalk Control Protocol (ATCP)*. Internet RFC 1378.

Parrington *et al.* 1995 — G. D. Parrington, S. K. Shrivastava, Wheater, S.M. and Little, M. C. (1995). The Design and Implementation of Arjuna, *USENIX Computing Systems Journal*, Vol 8, No 3.

Partridge 1992 — Partridge, C. (1992). *A Proposed Flow Specification*. Internet RFC 1363.

Patterson *et al.* 1988 — Patterson, D., Gibson, G and Katz, R. (1988). A Case for Redundant Arrays of Interactive Disks, ACM *International Conf. on Management of Data (SIGMOD)*, pp. 109-116, May.

Pease *et al.* 1980 — Pease, M., Shostak, R. and Lamport, L.(1980). Reaching agreement in the presence of faults. *Journal of the ACM*, Vol. 27, No. 2, April, pp. 228-34.

Pedone and Schiper 1999 — Pedone, F. and Schiper, A. (1999). Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing* (DISC '99), September.

Petersen *et al.* 1997 — Petersen, K., Spreitzer, M., Terry, D., Theimer, M. and Demers, A. (1997). Flexible update propagation for weakly consistent replication. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 288-301.

Peterson 1988 — Peterson, L. (1988). The Profile Naming Service. *ACM Transactions Computer Systems*, Vol. 6, No. 4, pp. 341–64.

Peterson *et al.* 1989 — Peterson, L.L., Buchholz, N.C. and Schlichting, R.D. (1989). Preserving and Using Context Information in Interprocess Commun ication. *ACM Transactions on Computer Systems*, Vol. 7, No. 3, pp. 217–46.

Pike *et al.* 1993 — Pike, R., Presotto, D., Thompson, K., Trickey, H. and Winterbottom, P. (1993). The Use of Name Spaces in Plan 9. *Operating Systems Review*, Vol. 27, No. 2, April 1993, pp. 72-76.

Popek and Walker 1985 — Popek, G. and Walker, B. (eds.). (1985). *The LOCUS Distributed System Architecture*. Cambridge Mass: MIT Press.

Postel 1981a — Postel, J. (1981). *Internet Protocol*. Internet RFC 791.

Postel 1981b — Postel, J. (1981). *Transmission Control Protocol.* Internet RFC 793. ωωω

Powell 1991 — Powell, D. (ed.) (1991). *Delta-4: a Generic Architecture for Dependable Distributed Computing*. Berlin and New York: Springer-Verlag.

Preneel *et al.* 1998 — Preneel, B., Rijmen, V. and Bosselaers, A. (1998). Recent developments in the design of conventional cryptographic algorithms, In *Computer Security and Industrial Cryptography, State of the Art and Evolution*, Lecture Notes in Computer Science, No. 1528, Springer-Verlag, pp. 106-131. ωωω

privacy.nb.ca — *International Cryptography Freedom.* ωωω

Radia *et al.* 1993 — Radia, S., Nelson, M. and Powell, M. (1993). *The Spring Naming Service*. Technical Report 93-16, Sun Microsystems Laboratories, Inc.

Rashid 1985 — Rashid, R.F. (1985). Network operating systems. In *Local Area Networks: An Advanced Course, Lecture Notes in Computer Science*, 184, Springer-Verlag, pp. 314–40.

Rashid 1986 — Rashid, R.F. (1986). From RIG to Accent to Mach: the evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Conference*, ACM, November.

Rashid and Robertson 1981 — Rashid, R. and Robertson, G. (1981). Accent: a communications oriented network operating system kernel. *ACM Operating Systems Review*, Vol. 15, No. 5, pp. 64–75.

Rashid *et al.* 1988 — Rashid, R., Tevanian Jr, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W.J. and Chew, J. (1988). Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions Computers*, Vol. 37, No. 8, pp. 896–907.

Raynal 1988 — Raynal, M. (1988). *Distributed Algorithms and Protocols*. Wiley.

Raynal 1992 — Raynal, M. (1992). About Logical Clocks for Distributed Systems. *ACM Operating Systems Review*, Vol. 26, No. 1, pp. 41–8.

Raynal and Singhal 1996 — Raynal, M. and Singhal, M. (1996). Capturing Causality in Distributed Systems. *IEEE Computer*, February, pp. 49-56.

Redmond 1997 — Redmond, F.E. (1997). *DCOM: Microsoft Distributed Component Model.* IDG Books Worldwide.

Reed 1983 — Reed, D.P. (1983). Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, Vol. 1, No. 1, pp. 3–23.

Ricart and Agrawala 1981 — Ricart, G. and Agrawala, A.K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Comms. ACM*, Vol. 24, No. 1, pp. 9–17.

Richardson *et al.* 1998 — Richardson, T., Stafford-Fraser,Q., Wood, K.R. and Hopper, A. (1998). Virtual Network Computing, *IEEE Internet Computing*. Vol. 2, No.1, Jan/Feb, pp. 33-8. ωωω

Ritchie 1984 — Ritchie, D. (1984). A Stream Input Output System. *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, pt. 2, pp. 1897–910.

Rivest 1992 — Rivest, R. (1992). *The MD5 Message-Digest Algorithm.* Internet RFC 1321. ωωω

Rivest *et al.* 1978 — Rivest, R.L., Shamir, A. and Adelman, L. (1978). A method of obtaining digital signatures and public key cryptosystems. *Comms. ACM*, Vol. 21, No. 2, pp. 120–6.

Rodrigues *et al.* 1998 — Rodrigues, L., Guerraoui, R., and Schiper, A. (1998). Scalable Atomic Multicast. In *Proceedings IEEE IC3N '98*. Technical Report 98/257. École polytechnique fédérale de Lausanne. 00000

Rose 1992 — Rose, M. T. (1992). *The Little Black Book: Mail Bonding with OSI Directory Services*. Englewood Cliffs, NJ: Prentice-Hall.

Rosenblum and Ousterhout 1992 — Rosenblum, M. and Ousterhout, J. (1992). The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, February, pp. 26-52. 00000

Rosenblum and Wolf 1997 — Rosenblum, D.S. and Wolf, A.L. (1997). A Design Framework for Internet-Scale Event Observation and Notification. In *Proceedings sixth European Software Engineering Conference/ ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering*, Zurich, Switzerland.

Rowley 1998 — Rowley, A. (1998). *A Security Architecture for Groupware*, Doctoral Thesis, Queen Mary and Westfield College, University of London. 00000

Rozier *et al.* 1988 — Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. and Neuhauser, W. (1988). Chorus Distributed Operating Systems. *Computing Systems Journal*, Vol. 1, No. 4, pp. 305–70.

Rozier *et al.* 1990 — Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. and Neuhauser, W. (1990). *Overview of the Chorus Distributed Operating System*. Technical Report CS/TR-90-25.1, Chorus Systèmes, France.

Saltzer *et al.* 1984 — Saltzer, J.H., Reed, D.P. and Clarke, D. (1984). End-to-End Arguments in System Design, *ACM Transactions on Computer Systems* Vol.2, No.4, pp. 277-88. 00000

Sandberg 1987 — Sandberg, R. (1987). *The Sun Network File System: Design, Implementation and Experience*. Technical Report. Mountain View Calif.: Sun Microsystems.

Sandberg *et al.* 1985 — Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B. (1985). The Design and Implementation of the Sun Network File System. In *Proceedings Usenix Conference*, Portland, Ore.

Sandhu *et al.* 1996 — Sandhu, R., Coyne, E., Felstein, H. and Youman, C. (1996). Role-Based Access Control Models, *IEEE Computer*, Vol. 29, No. 2, February. 00000

Sane *et al.* 1990 — Sane, A., MacGregor, K. and Campbell, R. (1990). Distributed Virtual Memory Consistency Protocols: Design and Performance. *Second IEEE Workshop on Experimental Distributed Systems*, pp. 91–6, October.

Sansom *et al.* 1986 — Sansom, R.D., Julin, D.P. and Rashid, R.F. (1986). *Extending a capability based system into a network environment*. Technical Report CMU-CS-86-116, Carnegie-Mellon University.

Santifaller 1991 — Santifaller. M. (1991). *TCP/IP and NFS, Internetworking in a Unix Environment*. Reading, Mass: Addison-Wesley.

Satyanarayanan 1981 — Satyanarayanan, M. (1981). A study of file sizes and functional lifetimes. In *Proceedings 8th ACM Symposium on Operating System Principles*, Asilomar, Calif.

Satyanarayanan 1989a — Satyanarayanan, M. (1989). Distributed File Systems. In *Distributed Systems, an Advanced Course*. (Mullender, S. ed.), second edition, Wokingham: ACM Press/Addison-Wesley. pp 353–83.

Satyanarayanan 1989b — Satyanarayanan, M. (1989). Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, Vol. 7, No. 3, pp. 247–80.

Satyanarayanan *et al.* 1990 — Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H. and Steere, D.C. (1990). Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 447–59.

Saunders 1987 — Saunders, B. (1987). The Information Structure of Distributed Mutual Exclusion Algorithms. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pp. 145-59.

Scheifler and Gettys 1986 — Scheifler, R.W. and Gettys, J. (1986). The X window system. *ACM Transactions on Computer Graphics*, Vol. 5, No. 2, pp. 76–109.

Schiper and Raynal 1996 — Schiper, A. and Raynal, M. (1996). From Group Communication to Transactions in Distributed Systems. *Comms. ACM*, Vol. 39, No. 4, pp. 84-7.

Schiper and Sandoz 1993 — Schiper, A. and Sandoz, A. (1993). Uniform reliable multicast in a virtually synchronous environment. *Proceedings 13th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pp. 561-8.

Schlageter 1982 — Schlageter, G. (1982). Problems of Optimistic Concurrency Control in Distributed Database Systems. *SigMOD Record.* Vol. 13, No. 3, pp. 62–6.

Schmidt 1998 — Schmidt, D. (1998). Evaluating architectures for multithreaded object request brokers, *Comms. ACM*, Vol. 44, No. 10, pp. 54-60.

Schneider 1990 — Schneider, F.B. (1990). Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, Vol. 22, No. 4, pp. 300–19.

Schneider 1996 — Schneider, S. (1996). Security properties and CSP. In *IEEE Symposium, on Security and Privacy*, pp. 174-187.

Schneier 1996 — Schneier, B. (1996). *Applied Cryptography*, second edition. New York: John Wiley.

Schroeder and Burrows 1990 — Schroeder, M. and Burrows, M. (1990). The Performance of Firefly RPC. *ACM Transactions on Computer Systems*, Vol. 8, No. 1. pp. 1–17.

Schulzrinne *et al.* 1996 — Schulzrinne, H., Casner, S., Frederick, D. and Jacobson, V. (1996). *RTP:A Transport Protocol for Real-Time Applications*, Internet RFC 1889, January. ωωω

Seetharaman 1998 — Seetharaman, K. (ed.) (1998). Special Issue: The CORBA Connection, *Comms. ACM*, October, Vol. 41, No. 10.

session directory — User Guide to sd. (Session Directory). ωωω

Shannon 1949 — Shannon, C.E. (1949). Communication Theory of Secrecy Systems, *Bell*

System Technical Journal, Vol. 28, No. 4, pp. 656-715.

Shepler 1999 — Shepler, S. (1999). *NFS Version 4 Design Considerations*, Internet RFC 2624, Sun Microsystems, June.

Shoch and Hupp 1980 — Shoch, J.F. and Hupp, J.A. (1980). Measured performance of an Ethernet local network. *Comms. ACM*, Vol. 23, No. 12, pp. 711–21.

Shoch and Hupp 1982 — Shoch, J.F. and Hupp, J.A. (1982). The 'Worm' programs – early experience with a distributed computation. *Comms. ACM*, Vol. 25, No. 3, pp. 172–80.

Shoch *et al.* 1982 — Shoch, J.F., Dalal, Y.K. and Redell, D.D. (1982). The evolution of the Ethernet local area network. *IEEE Computer*, Vol. 15, No. 8, pp. 10–28.

Shoch *et al.* 1985 — Shoch, J.F., Dalal, Y.K., Redell, D.D. and Crane, R.C. (1985). The Ethernet. In *Local Area Networks: an Advanced Course, Lecture Notes in Computer Science*. No. 184, Springer-Verlag, pp. 1–33.

Shrivastava *et al.* 1991 — Shrivastava, S., Dixon, G.N. and Parrington, G.D. (1991). An Overview of the Arjuna Distributed Programming System. *IEEE Software*, January. pp. 66–73.

Singh 1999 — Singh, S. (1999). *The Code Book*. London: Fourth Estate.

Sinha and Natarajan 1985 — Sinha, M. and Natarajan, N. (1985). A Priority Based Distributed Deadlock Detection Algorithm. *IEEE Transactions on Software Engineering*. Vol. 11, No. 1, pp. 67-80.

Spafford 1989 — Spafford, E.H. (1989). The Internet Worm: Crisis and Aftermath. *Comms. ACM*, Vol. 32, No. 6, pp. 678–87.

Spasojevic and Satyanarayanan 1996 — Spasojevic, M. and Satyanarayanan, M. (1996). An Empirical Study of a Wide-Area Distributed File System, *ACM Transactions on Computer Systems*, Vol. 14, No. 2, May, pp. 200-222.

Spector 1982 — Spector, A.Z. (1982). Performing remote operations efficiently on a local computer network. *Comms. ACM*, Vol. 25, No. 4, pp. 246–60.

Spurgeon 2000 — Spurgeon, C.E. (2000). *Ethernet: The Definitive Guide*. O'Reilly.

Srikanth and Toueg 1987 — Srikanth, T. and Toueg, S. (1987). Optimal Clock Synchronization. *Journal ACM*. 34, 3 (July), pp. 626-45.

Srinivasan 1995a — Srinivasan, R. (1995). RPC: *Remote Procedure Call Protocol Specification Version 2*. Internet RFC 1831. August.

Srinivasan 1995b — Srinivasa, R. (1995). *XDR: External Data Representation Standard*. Sun Microsystems. Internet RFC 1832. August.

Srinivasan and Mogul 1989 — Srinivasan, V. and Mogul, J.D. (1989). Spritely NFS: Experiments with Cache-Consistency Protocols, 12th ACM Symposium on Operating System Principles, Litchfield Park, Az., December, pp. 45-57.

Stallings 1998a — Stallings, W. (1998). *High Speed Networks – TCP/IP and ATM Design Principles*. Upper Saddle River, NJ: Prentice-Hall.

Stallings 1998b — Stallings, W. (1998). *Operating Systems*, third edition. Prentice-Hall International.

Stallings 1999 — Stallings, W. (1999). *Cryptography and Network Security – Principles and Practice*, second edition, Upper Saddle River, NJ: Prentice–Hall.

Steiner *et al.* 1988 — Steiner, J., Neuman, C. and Schiller, J. (1988). Kerberos: an authentication service for open network systems. In *Proceedings Usenix Winter Conference*, Berkeley: Calif.

Stelling *et al.* 1998 — Stelling, P., Foster, I., Kesselman, C., Lee, C. and von Laszewski, G. (1998). A Fault Detection Service for Wide Area Distributed Computations, *Proceedings 7th IEEE Symposium on High Performance Distributed Computing*, pp. 268-78.

Stoll 1989 — Stoll, C. (1989). *The Cuckoo's Egg: Tracking a Spy Through a Maze of Computer Espionage*. New York: Doubleday.

Stone 1993 — Stone, H. (1993). *High-performance Computer Architecture*, third edition. Addison-Wesley.

Sun 1989 — Sun Microsystems Inc. (1989). *NFS: Network File System Protocol Specification*. Internet RFC 1094. ωωω

Sun 1990 — Sun Microsystems Inc. (1990). *Network Programming*. Sun Microsystems, Mountain View, Calif.

Sun 1999 — Sun Microsystems Inc. (1999). *JavaSpaces technology*. ωωω

Sun and Ellis 1998 — Sun, C. and Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. *Proceedings Conference on Computer Supported Cooperative Work Systems*, ACM Press, pp. 59-68.

Tanenbaum 1992 — Tanenbaum, A.S. (1992). *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Tanenbaum 1996 — Tanenbaum, A.S. (1996). *Computer Networks*, third edition. Prentice-Hall International.

Tanenbaum and van Renesse 1985 — Tanenbaum, A. and van Renesse, R. (1985). Distributed Operating Systems, *Computing Surveys, ACM*, Vol. 17, No. 4, pp. 419-70.

Tanenbaum *et al.* 1990 — Tanenbaum, A.S., van Renesse, R., van Staveren, H., Sharp, G., Mullender, S., Jansen, J. and van Rossum, G. (1990). Experiences with the Amoeba Distributed Operating System. *Comms. ACM*, Vol. 33, No. 12, pp. 46–63.

Terry *et al.* 1995 — Terry, D., Theimer, M., Petersen, K., Demers, A, Spreitzer, M. and Hauser, C. (1995). Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 172-183.

TFCC — *IEEE Task Force on Cluster Computing*. ωωω

Thayer 1998 — Thayer, R. (1998). *IP Security Document Roadmap*, Internet RFC 2411, November. ωωω

Thekkath *et al.* 1997 — Thekkath, C.A., Mann, T. and Lee, E.K. (1997). Frangipani: A Scalable Distributed File System, In *Proc. 16th ACM Symposium on Operating System Principles*, St. Malo, France, October, pp. 224-237. ωωω

Tokuda *et al.* 1990 — Tokuda, H., Nakajima, T. and Rao, P. (1990). Real-time Mach: towards a predictable real-time system. In *Proceedings USENIX Mach Workshop*, pp. 73–82, October.

Topolcic 1990 — Topolcic, C. (ed.) (1990). *Experimental Internet Stream Protocol*, Version 2. Internet RFC 1190. ωωω

Tzou and Anderson 1991 — Tzou, S.-Y. and Anderson, D. (1991). The performance of message-passing using restricted virtual memory remapping. *Software–Practice and Experience*, Vol. 21, pp. 251-67.

van Renesse *et al.* 1989 — van Renesse, R., van Staveran, H. and Tanenbaum, A. (1989). The

Performance of the Amoeba Distributed Operating System. *Software – Practice and Experience*, Vol. 19, No. 3, pp. 223–34.

van Renesse *et al.* 1995 — van Renesse, R., Birman, K., Friedman, R., Hayden, M. and Karr, D. (1995). A Framework for Protocol Composition in Horus. *Proceedings PODC 1995*, pp. 80-9.

van Renesse *et al.* 1996 — van Renesse, R., Birman, K. and Maffeis, S. (1996). Horus: a Flexible Group Communication System. *Comms. ACM*, Vol. 39, No. 4, pp. 54-63.

van Steen *et al.* 1998 — van Steen, M., Hauck, F., Homburg, P. and Tanenbaum, A. (1998). Locating objects in wide-area systems. *IEEE Communication*, Vol. 36, No. 1, pp. 104-109.

Vinoski 1998 — Vinoski, S. (1998). New Features for CORBA 3.0, *Comms. ACM*, October 1998, Vol. 41, No. 10. pp. 44-52.

Vogt *et al.* 1993 — Vogt, C., Herrtwich R.G. and Nagarajan, R. (1993). HeiRAT – The Heidelberg Resource Administration Technique: Design Philosophy and Goals. *Kommunikation in verteilten Systemen*, Munich, Informatik aktuell, Springer.

Volpano and Smith 1999 — Volpano, D. and Smith, G. (1999). Language Issues in Mobile Program Security. To appear in *Lecture Notes in Computer Science*, Springer.

von Eicken *et al.* 1995 — von Eicken, T., Basu, A., Buch, V. and Vogels, V. (1995). U-Net: a user-level network interface for parallel and distributed programming. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 40-53.

Wahl *et al.* 1997 — Wahl, M., Howes, T. and Kille, S. (1997). *The Lightweight Directory Access Protocol (v3)*. Internet RFC 2251.

Waldo 1999 — Waldo, J. (1999). The Jini Architecture for Network-centric Computing. *Comms. ACM*. Vol. 42, No. 7, pp. 76-82.

Waldo *et al.* 1994 — Waldo, J, Wyant, G., Wollrath, A. and Kendall, S. (1994). A Note on Distributed Computing. In Arnold *et al.* 1999, pp. 307-26.

Waldspurger *et al.* 1992 — Waldspurger, C., Hogg, T., Huberman, B., Kephart, J. and Stornetta, W. (1992). Spawn: A Distributed Computational Economy. *Transactions on Software Engineering*, Vol. 18, No. 2, pp. 103-17.

Want *et al.* 1992 — Want, R., Hopper, A., Falcao, V and Gibbons, V. (1992). The Active Badge Location System. *ACM Transactions on Information Systems*, Vol. 10, No.1, January, pp. 91-102.

web.mit.edu I — *Kerberos: The Network Authentication Protocol.*

web.mit.edu II — *The Three Myths of Firewalls.*

Weikum 1991 — Weikum, G. (1991). Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions Database Systems*, Vol. 16, No. 1, pp. 132–40.

Weiser 1993 — Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Comms. ACM*, Vol. 36, No. 7, pp. 74-84.

Wheeler and Needham 1994 — Wheeler, D.J. and Needham, R.M. (1994). TEA, a Tiny Encryption Algorithm. Technical Report 355, *Two Cryptographic Notes*, Computer Laboratory, University of Cambridge, December, pp. 1–3.

Wheeler and Needham 1997 — Wheeler, D.J. and Needham, R.M. (1997). *Tea Extensions*, October 1994, pp. 1–3.

Wiesmann *et al.* 2000 — Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. and Alonso, G. (2000). Understanding replication in databases and distributed systems. In *Proceedings 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, Taipei, Republic of China, IEEE.

Wobber *et al.* 1994 — Wobber, E., Abadi, M., Burrows, M. and Lampson, B. (1994). Authentication in the Taos operating system. *ACM Transactions Computer Systems*. 12, 1 (Feb.), pp. 3-32.

Wuu and Bernstein 1984 — Wuu, G.T. and Bernstein, A.J. (1984). Efficient Solutions to the Replicated Log and Dictionary Problems. *ACM Proceedings Third Annual Symposium Principles of Distributed Computing*, pp. 233–42.

www.bluetooth.com — *The Official Bluetooth SIG Website.*

www.bxa.doc.gov — Bureau of Export Administration, US Department of Commerce, *Commercial Encryption Export Controls.*

www.cdk3.net — Coulouris, G., Dollimore, J. and Kindberg, T. (Eds), *Distributed Systems, Concepts and Design: Supporting material.*

www.citrix.com — Citrix Corporation, *Server-based Computing White Paper.*

www.cooltown.hp.com — Hewlett-Packard Corporation, *CoolTown nomadic computing project pages.*

www.cren.net — Corporation for Research and Educational Networking, *CERN Certificate Authority.*

www.cuseeme.com — CU-SeeMe Networks Inc, *Home page.*

www.doi.org — International DOI Foundation, *Pages on digital object identifiers.*

www.handle.net — Handle system, *Home page.*

www.iana.org — Internet Assigned Numbers Authority, *IANA Home Page.*

www.ietf.org — Internet Engineering Task Force, *Internet RFC Index Page.*

www.isoc.org — Robert Hobbes Zakon, *Hobbes' Internet Timeline.*

www.microsoft.com — Microsoft Corporation, *NetShow Theater Server Web Page.*

www.microsoft.com I — Microsoft Corporation, *Active Directory Services.*

www.microsoft.com II — Microsoft Corporation, *Windows 2000 Kerberos Authentication*, White Paper.

www.microsoft.com III — Microsoft Corporation, *NetMeeting home page.*

www.mids.org — Matrix Information and Directory Services Inc. *Internet Performance.*

www.millicent.com — Compaq Corporation, *Millicent MicroCommerce system.*

www.omg.org — Object Management Group, *Index to CORBA services.* OMG: Framingham, Mass.

www.opengroup.org — Open Group, *Portal to the World of DCE.*

www.openssl.org — OpenSSL Project, *OpenSSL: The Open Source toolkit for SSL/TLS.*

www.pgp.com — *PGP home page.*

www.reed.com	Read, D.P. (2000). *The End of the End-to-End Argument.*	0000
www.rsasecurity.com	RSA Security Inc., *Home page.*	0000
www.rsasecurity.com I	RSA Corporation (1997). *DES Challenge.*	0000
www.rsasecurity.com II	RSA Corporation (1999). *RSA Factoring Challenge.*	0000
www.rtj.org	*Real-Time for Java TM Experts Group.*	0000
www.spec.org	*SPEC SFS97 Benchmark.*	0000
www.upnp.com	*Universal Plug and Play home page.*	0000
www.verisign.com	Verisign Inc., *Home page.*	0000
www.w3.org I	World Wide Web Consortium, *Home page.*	0000
www.w3.org II	World Wide Web Consortium, *Pages on the HyperText Markup Language.*	0000
www.w3.org III	World Wide Web Consortium, *Pages on Naming and Addressing.*	0000
www.w3.org IV	World Wide Web Consortium, *Pages on the HyperText Transfer Protocol.*	0000
www.w3.org V	World Wide Web Consortium, *Pages on the Resource Description Framework and other metadata schemes.*	0000
www.w3.org VI	World Wide Web Consortium, *Pages on the Extensible Markup Language.*	0000
www.w3.org VII	World Wide Web Consortium, *Pages on the Extensible Stylesheet Language.*	0000
www.wapforum.org	WAP Forum, *White Papers and Specifications.*	0000
www.wlana.com	*The IEEE 802.11 Wireless LAN Standard.*	0000
Wyckoff *et al.* 1998	Wyckoff, P., McLaughry, S., Lehman, T. and Ford, D. (1998). T Spaces. *IBM Systems Journal*, Vol. 37, No. 3.	
Zhang *et al.* 1993	Zhang, L., Deering, S.E., Estrin, D., Shenker, S. and Zappala, D. (1993). RSVP – A New Resource Reservation Protocol. *IEEE Network Magazine*, Vol. 9, No. 5.	
Zimmermann 1995	Zimmermann, P.R. (1995). *The Official PGP User's Guide*, Cambridge, Mass.: MIT Press.	

索　引

索引中的页码为英文原书页码，与书中边栏标注的页码一致。

A

abort（放弃），472
access control（访问控制），267~270
access control list（访问控制列表），268, 316
access rights（访问权限），58
access transparency（访问透明性），23, 315, 333
ACID properties（ACID特性），471
ack-implosion（确认爆炸），438
activation（激活），180
activator（激励器），180
active object（主动对象），180
active replication（主动复制），570~572
address resolution protocol（ARP）（地址解析协议），96
address space（地址空间），213, 215~216
  aliasing（别名），225
  inheritance（继承），219
  region（区域），215
  shared region（共享区域），216, 236
admission control（许可控制），615, 622~623
AES(Advanced Encryption Standard)（高级加密标准），278
agreement（协定），
  in consensus and related problems（在共识和相关问题中），452~453
  of multicast delivery（关于组播传递的），439
  problems of（的问题），451
  uniform（一致的），442
alias（别名），359
Amoeba（Amoeba系统），
  multicast protocol（组播协议），446
  run server（运行服务器），217
Andrew File System(AFS)（Andrew文件系统（AFS），），317, 335~344
  for Linux（Linux上的），318
  in DCE/DFS（DCE/DFS上的），347
  performance（性能），344
  wide-area support（广域支持），344
anti-entropy protocol（反熵协议），580.582
Apollo Domain（Apollo域），636
applet（小程序），14
  threads within（内部线程），226
application layer（应用层），77, 79
architectural models（体系结构模型），31~43
ARP（ARP，参见address resolution protocol），
asymmetric cryptography（非对称密码学），273,279~281
asynchronous communication（异步通信），128
asynchronous distributed system（异步分布式系统），51, 391, 450, 459
asynchronous invocation（异步调用），240
  persistent（持久），241
asynchronous operation（异步操作），239~242
asynchronous RMI（异步RMI），
  in CORBA（CORBA中的），684
at-least-once invocation semantics（至少一次调用语义），175
ATM(Asynchronous Transfer Mode)（异步传输模式），71, 74, 76, 79, 86, 111, 119~121
at-most-once invocation semantics（至多一次调用语义），176
atomic commit protocol（原子提交协议），516~528
  failure model（故障模型），520
  two-phase commit protocol（两阶段提交协议，参见two-phase commit protocol），
atomic consistency（原子一致性），644
atomic operation（原子操作），466
atomic transaction（原子事务）
audio conferening application（音频会议应用），610
authentication（认证），60,262~264
authentication server（认证服务器），292
authentication service（认证服务），294
availability（可用性），22, 554, 572~591

B

backbone（主干网），97
bandwidth（带宽），50
Bayou（Bayou），582~584
  dependency check（依赖检查），583
  merge procedure（合并程序），583

Bellman-Ford routing protocols（Bellman-Ford路由协议），83
best-efforts resource scheduling（最佳资源调度），614
big-endian order（大序法排序），138
binder（绑定器），180
  portmapper（端口映射器），186
  rmiregistry（rmi注册表），196
binding（绑定），
  socket（套接字），158
birthday attack（生日攻击），286
block cipher（分组密码），273~274
blocking operations（阻塞操作），128
BlueTooth wireless network（蓝牙无线网络），71
bridge（网桥），88
broadcast（广播），436
brute-force attack（强行攻击），272
byzantine failure（拜占庭式故障），56
byzantine generals（拜占庭将军问题），453, 456~459

C

cache（缓存，高速缓冲存储器），36, 46, 554
  coherence of cached files（缓存文件的一致性），589
caching（高速缓冲存储），
  file, write -through（文件，写透），329
  files at client（客户端文件），330
  files at server（服务器端文件），329
  of whole files（所有文件的），335
  validation procedure（验证程序），331, 341
callback（回调），
  in CORBA remote method invocation（在CORBA远程方法调用中），677
  in Java remote method invocation（在JAVA远程方法调用中），200
callback promise（回调承诺），340
capability（权限），268
cascading abort（连锁放弃），479
case studies（实例研究），
  Andrew File System(AFS)（Andrew文件系统(AFS)），335~344
  ATM(Asynchronous Transfer Mode)（异步传输模式），119~121
  Bayou（Bayou），582~584
  Coda file system（Coda文件系统），584~590
  CORBA（CORBA），670~695
  Domain Name System(DNS)（域名系统），364~371
  Ethernet（以太网），111~116
  Global Name Service（全局名字服务），374~377
  Gossip architecture（闲谈体系结构），572~582
  IEEE 802.11 wireless LAN（IEEE 802.11 无线局域网），116~119
  Interprocess communication in UNIX（UNIX系统的进程间通信），158~161
  Ivy（Ivy），653~657
  Java remote method invocation（Java远程方法调用），194~202
  Kerberos（Kerberos认证），293~298
  Mach（Mach操作系统），699~720
  Millicent protocol（Millicent协议），303~306
  Munin（Munin），661~663
  Needham-Schroeder protocol（Needham-Schroeder协议），292~293
  Network File System(NFS)（网络文件系统(NFS)），323~335
  Network Time Protocol（网络时间协议），393~396
  Secure Sockets Layer(SSL)（安全套接字层（SSL）），298~303
  Sun RPC（Sun远程过程调用），184~187
  X.500 directory service（X.500目录服务），378~382
catch exception（捕获异常），171
causal consistency（因果一致性），663
causal ordering（因果排序），397
  of multicast delivery（组播传递的），443
  of request handling（请求处理的），558
CDR（CDR，参见CORBA），
  Common Data Representation（公共数据表示），
certificate（证书），265~267
  X.509 standard format（X.509标准格式），287
certificate authority（证书权威机构），288
challenge, for authentication（认证质询），264
checkpointing（检查点），543
Chorus（Chorus系统），245
chosen plaintext attack（明文选择攻击），280
cipher block chaining(CBC)（密码组链接），274
cipher suite（密码组），300
ciphertext（密文），272
circuit switching network（电路交换网络，参见network, circuit switching），
classless interdomain routing(CIDR)（无等级域间路由(CIDR)），99
clients（客户），8
client-server communication（客户－服务器通信），145~153
client-server model（客户－服务器模型），34~36
  variations（变种），37~42
clock（时钟），
  accuracy（精确），389

agreement（协定），389
computer（计算机），46, 50,387
correctness（正确性），390
drift（漂移），50, 388
faulty（有错的），390
global（全局的），2
logical（逻辑的），398
matrix（矩阵），400
monotonicity（单调性），390
resolution（分辨率），388
skew（偏差），388
synchronization（同步，参见synchronization of clocks）
vector（向量），399
cluster, of computers（计算机集群），217
Coda file system（Coda文件系统），584~590
available volume storage group(AVSG)（可用的卷存储组），585
Coda version vector(CVV)（Coda版本向量（CVV）），586
volume storage group(VSG)（卷存储组（VSG）），585
codec（多媒体数字信号编/解码器），613
coherence（连贯性），
of distributed shared memory（分布式共享内存），645~646
collision detection（冲突检测），114
commit（提交），472
communication（通信），
asynchronous（异步），128
client-server（客户－服务器），145~153
group（组），153~158
operating system support for（操作系统支持），231~239
reliable（可靠的），57, 129
synchronous（同步），128
communication channels（通信通道），
performance（性能），49
threats to（威胁），60
commuting operations（可交换操作），571
competing memory accesses（竞争性内存访问），658
Compression（压缩），613
concurrency（并发），2, 22, 23
of file updates（文件修改的），315
concurrency control（并发控制），473~514
comparison of methods（方法的比较），508
conflicting operations（冲突的操作），476
in CORBA concurrency control service（在CORBA并发控制服务中），486, 687
in distributed transaction（在分布式事务中），528~531
inconsistent retrieval（不一致的检索），474
with locks（用锁，参见locks），
lost update（更新丢失），473
operation conflict rules（操作冲突规则），476, 484
optimistic（乐观，参见 optimistic concurrency control），
by timestamp ordering（时间戳排序，参见 timestamp ordering），
concurrency transparency（并发透明性），23
conflicting operations（冲突的操作），476
confusion(in cryptography)（混乱(在密码学中)），275
congestion control（拥塞控制，参见 network），
congestion control（拥塞控制），
connection（连接），
persistent（持久），237
consensus（共识），451~462
impossibility result for an asynchronous system（异步系统的不可能性结果），459
in a synchronous system（在同步系统中），455
related to other problems（与其他问题的关系），454
consistency（一致性），
of replicated data（复制数据的），555
of shared memory（共享内存的），643
context switch（上下文转换），224
cookie（Web服务器发给Web浏览器的一小段信息），326
Coordinated Universal Time (UTC)（通用协调时间UTC），388
copy-on-write（写时复制），219
in Mach（Mach系统的），714~716
CORBA（公共对象请求代理体系结构），
architecture（体系结构），677~680
object adapter（对象适配器），678
object request broker（对象请求代理），678
proxy（代理），678
skeleton（骨架），678
case study（实例研究），670~695
client and server example（客户和服务器举例），673~677
Common Data Representation（公共数据表示），140~141
dynamic invocation interface（动态调用接口），679
dynamic skeleton interface（动态骨架接口），680
extensions（扩展），684
asynchronous RMI（异步RMI），684
objects by value（传递值的对象），684
general Inter-ORB protocol(GIOP)（通用ORB间协议

(GIOP)), 670
implementation repository (实现仓库), 679
interface definition language (接口定义语言), 169, 671~673, 680~684
attribute (属性), 682
inheritance (继承), 682
interface (接口), 680
method (方法), 680
module (模块), 680
parameters and results (参数和结果), 671
pseudo object (伪对象), 673
type (类型), 682
interface repository (接口仓库), 679
internetInter-ORB protocol(IIOP) (因特网ORB间协议(IIOP)), 670
interoperable object reference(IOR) (互操作对象引用(IOR)), 684
persistent (持久), 685
transient (暂态), 685
language mapping (语言映射), 685
marshalling (编码), 141
object model (对象映射), 671
object request broker(ORB) (对象请求代理), 670, 678
remote method invocation (远程方法调用), 671~677
callback (回调), 677
remote object reference (远程对象引用, 参见 CORBA),
interoperable object reference (互操作对象引用),
services (服务), 686~695
concurrency control service (并发控制服务), 486, 687
event service (事件服务), 690~692
notification service (通知服务), 692~693
persistent object service (持久对象服务), 311, 687
security service (安全服务), 693~694
transaction service (事务服务), 687
crash failure (崩溃故障), 54, 422
credentials (证书), 270~271
critical section (临界区), 423
cryptography (密码学), 60, 261~265, 272~281
and politics (和政治学), 290
performance of algorithms (算法的性能), 289
CSMA/CA (具有避免冲突的载波侦听多路访问), 118
CSMA/CD (具有检测冲突的载波侦听多路访问), 112
cut (割集), 403
consistent (一致的), 403
frontier (边界), 403

D

data compression for multimedia (多媒体的数据压缩), 613
data link layer (数据链路层), 79
data streaming (数据流, 参见 network, data streaming),
datagram (数据报), 81, 95
deadlock (死锁), 490~494
definition (定义), 491
detection (检测), 401, 492
distributed (分布式的, 参见 distributed deadlock),
prevention (预防), 492, 493
timeouts (超时), 493
wait-for graph (等待图), 491
with read-write locks (读-写锁), 490
dealing room system (交易所系统), 188
debugging distributed programs (调试分布式程序), 402, 409~415
delegation (of rights) (权利的委托), 270
delivery failures (发送故障), 147
denial of service (拒绝服务), 61
denial of service attack (拒绝服务攻击), 19, 255
dependability (可依赖性), 46
DES(Data Encryption Standard) (数据加密标准), 277, 289
design requirements (设计需求), 44
detecting failure (检测故障), 21
diffusion (in cryptography) (扩散 (在密码学中)), 275
digest function (摘要函数), 283
digital signature (数字签名), 264~265, 282~288
directory service (目录服务), 321~322, 371
attribute (属性), 371
dirty read (读取脏数据), 478
disconnected operation (断连操作)), 555, 582, 590
discovery service (发现服务), 42~43, 371
Jini (Jini (Sun的一种分布式技术)), 372~374
locating (定位), 373
registration (注册), 372
scope (作用域), 372
simple service discovery protocol (简单服务发现协议), 372
dispatcher (调度程序), 179
distance vector routing algorithm (距离向量路由算法), 83
distributed deadlock (分布式死锁), 531~539
edge chasing (边追逐), 534~539
transaction priorities (事务优先级), 536
phantom deadlock (假象死锁), 533
distributed garbage collection (分布式无用单元回收), 174,182~183

in Java （在Java中）, 174, 182~183
use of leases（租借的使用）, 183
distributed object（分布式对象）, 171
communication（通信）, 169~183
distributed object model（分布式对象模型）, 172
distributed operating system（分布式操作系统）, 209
distributed shared memory（分布式共享内存）, 635~665
byte-oriented（面向字节的）, 641
centralized manager（中心管理器）, 653
consistency model（一致性模型）, 642~646, 663~664
design and implementation issues（设计和实现问题）, 640~649
distributed manager（分布式管理器）, 654~657
granularity（粒度）, 648
immutable data stored in（存储的不变数据）, 641
object-oriented（面向对象的）, 641
synchronization model（同步模型）, 642
thrashing（系统颠簸）, 649
update options（更新选项）, 646~648
write-invalidation（写失效）, 647, 651~657
write-update（更新写）, 646
distributed shared memory (DSM)（分布式共享内存(DSM)）, 311
distributed transaction（分布式事务）,
atomic commit protocol（原子提交协议）, 516~528
concurrency control（并发控制）, 528~531
locking（锁）, 528
optimistic（乐观的）, 530
timestamp ordering（时间戳排序）, 529
coordinator（协调者）, 518~519
flat（平面）, 516
nested（嵌套）, 516
one-phase commit protocol（单阶段提交协议）, 519
two-phase commit protocol（两阶段提交协议，参见two-phase commit protocol）,
DNS （DNS，参见Domain Name System）,
Domain Name System (DNS)（域名系统）, 106,364~371
BSD implementation（BSD实现）, 370
domain name （域名）, 364
name server（名字服务器）, 366~371
navigation（导航）, 368
query（查询）, 365
resource record（资源记录）, 369
zone（区域）, 366
domain transition（域转换）, 224
downloaded code（下载的代码）, 14
downloading of code（代码的下载）,
in Java remote method invocation （在Java远程方法调用中）, 196
DSL (digital subscriber line)（数字用户线）, 71, 87
DSM（DSM，参见distributed shared memory）

E

eager update propagation（及时更新传递）, 593, 661
eavesdropping attack（窃听攻击）, 255
election（选举）, 431~436
bully algorithm（霸道算法）, 434
for processes in a ring（对环中进程的）, 432
electronic commerce（电子商务）,
security needs（安全性需要）, 257
elliptic curve encryption （椭圆曲线加密）, 281
emulation （模拟）,
of operating system （操作系统的）, 244, 702
encapsulation （封装）, 77, 91
encryption（加密）, 60
enemy（敌方）, 59
enemy, security threats from an （来自敌方的安全威胁）, 61
entry consistency（变量项一致性）, 664
Ethernet（以太网）, 88,111~116
Ethernet hub（以太网网络集线器）, 87, 89
Ethernet switch（以太网交换机）, 87
event（事件）, 187~194
concurrency（并发）, 398
delivery semantics （传递语义）, 191
filtering（过滤）, 192
forwarding（前推）, 192
in CORBA event service（在CORBA事件服务中）, 690~692
in CORBA notification service （在CORBA通知服务中）, 692~693
in Jini （在Jini中）, 192~194
notification （通知）, 187~194
object of interest（兴趣对象）, 190
observer（观察者）, 191
ordering（排序）, 52
patterns（模式）, 192
publisher（发布者）, 191
subscriber（订阅者）, 190
type（类型）, 187, 189
exactly once（恰好一次）, 175
exception（异常）, 171
catch（捕获）, 171
in CORBA remote invocation（在CORBA远程调用中）, 177
in Java remote method invocation（在Java远程方法调用中）, 177

throw（抛出），171
execution environment （执行环境），214
Exokernel（Exokernel内核设计），246
external data representation （外部数据表示），138~143
external pager（外部换页），716~719

F

factory method （工厂方法），180, 674
factory object （工厂对象），180
fail-stop（故障停止），54
failure（故障），
  arbitrary（随机的），56
  byzantine（拜占庭式），56
  timing（时序），56
failure atomicity （故障原子性），470, 539
failure detector（故障检测器），422~423
  to solve consensus （为了解决共识），460
failure handling（故障处理），21, 22
failure model（故障模型），53~58
  atomic commit protocol （原子提交协议），520
  IP multicast （IP组播），155
  request-reply protocol（请求－应答协议），147
  TCP（TCP），135
  transaction（事务的），469
  UDP（UDP），131
failure transparency（故障透明性），23
fairness（公平性），424
false sharing（错误共享），648, 716
fault-tolerant average（容错平均值），393
fault-tolerant service （容错服务），555,565~572
FIFO ordering（FIFO排序），
  of multicast delivery （组播传递的），442
  of request handling（请求处理的），558
file（文件），
  mapped（映射的），216, 636
  replicated（复制的），585
file group identifier（文件组标识），323
file operations（文件操作），
  in directory service model（在目录服务模型中的），322
  in flat file service model（在平面文件服务模型中的），319
  in NFS server（在NFS服务器中的），326
  in UNIX（在UNIX中的），314
Firefly RPC（Firefly远程过程调用），236
firewall（防火墙），4,107~110,271~272
flat file service（平面文件服务），318~321
flow control（流控制），105
flow specification（流说明），620
frame relay （帧中继，参见network, frame relay），
Frangipani distributed file system（Frangipani 分布式文件系统），350
front end （前端），558
FTP（文件传输协议），79,80,90,109
fundamental models（基础模型），47~62

G

garbage collection（无用单元回收），171, 401
  distributed（分布式的），182~183
  in distributed object system（在分布式对象系统中的），174
  local（局部的），171
gateway （网关），72
global clock（全局时钟），2
Global Name Service（全局名字服务），374~377
  directory identifier（目录标识符），375
  working root（工作根），376
Global Positioning System (GPS)（全球定位系统），389
global state（全局状态），400~415
  consistent（一致性），404
  predicate（谓词），404
  snapshot（快照），405~409
  stable（稳定的），404
GNS （GNS，参见Global Name Service），374
gossip architecture（gossip型体系结构），572~582
  gossip message （gossip消息），574
    processing（处理），579~580
    propagating（传播），580~581
  query processing（查询处理），577
  update processing（更新处理），577~579
GPS（GPS，参见Global Positioning System），
group （组），
  closed （封闭的），437
  membership service（成员访问），559
  of objects（对象的），564
  open （开放的），438
  overlapping（重叠的），450
  view（视图），560,561~564
group communication（组通信），153~158,436,559~565
  view-synchronous（视图同步），562~564
GSM cellular phone network（GSM蜂窝式电话网），72

H

handle system（处理系统），357
handshake protocol, in SSL（握手协议，在SSL中），300

happened-before（发生在先），397
heterogeneity（异构性），16, 17, 315, 317
  name service（名字服务），360
historical notes（历史记录），
  distributed file systems（分布式文件系统），310
  emergence of modern cryptography（现代密码学的出现），253
  networking（网络），266
  security, evolution of needs（安全，需求的演变），252
history（历史），387
  global（全局的），403
  of server operations in request-reply（请求－应答中服务器操作的），148
hold-back queue（保留队列），441
hostname（主机名），356
HTML（超文本标记语言），10
HTTP（超文本传输协议），13, 79, 80, 81, 90, 91, 107, 150~153
  over persistent connection（基于持久连接的），240
  performance of（性能的），237
hub（网络集线器），89
hybrid cryptographic protocol（混合密码协议），264, 281
hybrid memory consistency model（混合内存一致性模型），663

I

IANA (Internet Assigned Numbers Authority)（因特网编号管理局），80
IDEA (International Data Encryption Algorithm)（国际数据加密算法），278, 289
idempotent operation（幂等操作），148, 176,320
identifier（标识符），354,356
IDL（IDL，参见interface definition language），
IEEE 802 standards（IEEE 802标准），111
IEEE 802.11 wireless LAN（IEEE 802.11无线局域网），116~119
implementation repository, in CORBA（实现仓库，在CORBA中），679
inconsistent retrieval（不一致检索），474, 485
independent failure（独立故障），2
initialization vector (for a cipher)（初始化向量（用于密码）），274
i-node number（i-结点号），324
input parameters（输入参数），168
integrity（完整性），
  in consensus and related problems（在共识和相关问题中），452~454
  of multicast delivery（在组播传递中），439
  of reliable communication（在可靠通信中），57
intentions list（意图列表），540,545
interaction model（交互模型），49~53
interactive consistency（交互一致性），453
interface（接口），43,167~169,170
  in distributed system（在分布式系统中），167
  remote（远程），168
  service（服务），168
interface definition language（接口定义语言），167, 168
  CORBA（CORBA），671~673,680~684
  CORBA IDL example（CORBA IDL例子），169
  Sun RPC example（Sun RPC例子），185
interface repository, in CORBA（接口仓库，在CORBA中），679
International Atomic Time（国际原子时钟），389
Internet（因特网），3, 4, 80, 90~110
  routing protocols（路由协议），97~99
  traffic statistics（流量统计），68
internet address, Java API（因特网地址，Java API），130
Internet protocol（因特网协议见，参见IP），
internetwork（互联网），72,79,87~90
interoperable Inter-ORB protocol (IIOP)（可互操作的ORB间协议），685
interprocess communication（进程间通信），
  characteristics（特征），127
  in UNIX（在UNIX中），158~161
intranet（企业内部网），4, 5
invocation mechanism（调用机制），210
  asynchronous（异步），240
  latency（延迟，等待时间），234
  operating system support for（操作系统支持），231~239
  performance of（性能），233~239
  scheduling and communication as part of（其中的调度和通信），210
  throughput（吞吐量），235
  within a computer（在一个计算机内），237~239
invocation semantics（调用语义），
  at-least-once（至少一次），175
  at-most-once（至多一次），176
  maybe（或许），175
invokes an operation（调用一个操作），8
IOR（IOR，参见CORBA），
  interoperable object reference（互操作对象引用），
IP（IP），79,95~104
  API（应用程序接口），127~138
IP addressing（IP寻址），92~95

IP multicast（IP组播），89, 154~156, 436, 440
address allocation（地址分配），155
failure model（故障模型），155
Java API（Java API），155
router（路由器），155
IP spoofing（IP电子欺骗），96
IPC （IPC，参见interprocess communication），
IPv4（第4版因特网协议），92
IPv6（第6版因特网协议），74,89,100~102
ISDN（综合业务数字网），79,87
ISIS（智能系统和信息服务），564
isochronous data streams（等时数据流），613
Ivy （Ivy），653~657

J

Java（Java）
object serialization（对象序列化），142~144
reflection （反射），143
thread （线程，参见thread, Java），
Java API（Java API），
DatagramPacket（DatagramPacket类），132
DatagramSocket（DatagramSocket类），132
InetAddress（InetAddress类），130
MulticastSocket（MulticastSocket类），155
ServerSocket（ServerSocket类），136
Socket（Socket类），137
Java remote method invocation （Java远程方法调用），194~202
callback （回调），200
client program（客户程序），199
design and implementation（设计和实现），201~202
downloading of classes（类下载），196
exception （异常），177
parameter and result passing （参数和结果传递），195
remote interface （远程接口），195
RMI registry（RMI注册），196
server program（服务器程序），197
use of reflection （反射的使用），201
Java security （Java安全性），256
Jini（Jini），
discovery service（发现服务），372~374
leases（租借），374
distributed event specification （分布式事件描述），192~194
leases（租借），183
jitter（抖动），50

K

Kerberos （Kerberos），293~298
Kerberos authentication （Kerberos 认证），
for NFS（NFS的），332
kernel（内核），213
monolithic（整体的），243
keystream generator（密钥序列发生器），274

L

L4 microkernel（L4 微内核），246
Lamport timestamp （Lamport时间戳），398
LAN （LAN，参见network, local area）
latency（延迟，等待时间），49
layers in protocols （协议中的层），77
lazy update propagation（惰性更新传播），593, 661
LDAP （LDAP，参见lightweight diretory access protocol），
leaky bucket algorithm（漏桶算法），619
leases（租借），183, 200
for callbacks（用于回调），200
in distributed garbage collection （在分布式无用单元回收中），183
in Jini （在Jini中），183
in Jini discovery service （在Jini发现服务中），374
in Jini event service（在Jini事件服务中），193
lightweight directory access protocol（轻量级目录访问协议），382
lightweight RPC（轻量级RPC），237~239
Linda（Linda系统），641
linear-bounded arrival processes (LBAP)（线性限制的到达处理），618
linearizability（线性能力），566
linearization（线性化），404
links（链接），9
link-state routing algorithms（连接态路由算法），85
little-endian order（小序法排序），138
liveness property（活性），405
load balancing（负载平衡），45
load sharing（负载共享），217~218
local area network （局域网，参见network, local area），
location service （定位服务），182
location transparency（位置透明性），23,315, 334
in middleware（在中间件中），166
location-aware computing（位置清楚的计算），6
lock manager（锁管理器），487
locking（加锁），
in distributed transaction（在分布式事务中），528
locks（锁），482~496
causing deadlock （引起死锁，参见deadlock）
exclusive（排它），482

granularity（粒度）, 483
hierarchic（层次）, 495, 496
implementation（实现）, 486
in nested transaction（在嵌套事务中）, 489
lock manager（锁管理器）, 486
operation conflict rules（操作冲突规则）, 484
promotion（提升）, 485
read-write（读－写）, 484, 485
read-write-commit（读－写－提交）, 494
shared（共享的）, 484
strict two-phase（严格两阶段）, 483
timeouts（超时）, 493
two-phase locking（两阶段加锁）, 483
two-version（两版本）, 494
logging（日志）, 540~543
logical clock（逻辑时钟）, 398
logical time（逻辑时间）, 53,397~400
log-structured file storage（结构化记录文件存储）, 348
lost reply message（丢失的应答消息）, 148
lost update（更新丢失）, 473, 485

M

Mach（Mach系统）, 245,699~720
communication（通信）, 707~713
external pager（外部换页）, 716~719
memory cache object（内存缓存对象）, 703, 717
memory management（内存管理）, 713~719
memory object（内存对象）, 703
message（消息）, 703,707~708
naming（命名）, 703
network communication（网络通信）, 710~713
network port（网络端口）, 710
network server（网络服务器）, 710
port（端口）, 703, 703~705
port right（端口权限）, 703, 704, 711
port set（端口集）, 703, 709
task（任务）, 703, 705~706
virtual memory（虚拟内存）, 702
MAN（MAN，参见network, metropolitan area）,
man-in-the-middle attack（中间人攻击）, 255
marshalling（编码）, 139
masking failure（屏蔽故障）, 21
masquerading attack（伪装攻击）, 255
maximum transfer unit (MTU)（最大传输单元）, 79, 95, 113
maybe invocation semantics（或许调用语义）, 175
MD5 message digest algorithm（MD5消息摘要算法）, 287, 289
media synchronization（媒体同步）, 611
memory object（内存对象）, 717~719
message（消息）,
destination（目的地）, 128~129
reply（应答）, 147
request（请求）, 147
message authentication code (MAC)（消息鉴别码(MAC)）, 284
message digest（消息摘要）, 264
message passing（消息传递）, 126
message tampering attack（消息篡改攻击）, 255
metadata（元数据）, 313
metropolitan area network（城域网，参见 network, metropolitan area）,
microkernel（微内核）, 243~246
comparison with monolithic kernel（与整体内核比较）, 244
middleware（中间件）, 16, 32, 166
and heterogeneity（和异构性）, 167
operating system support for（操作系统支持）, 209
Millicent protocol（Millicent协议）, 303~306
mime type（多用途因特网邮件扩展类型）, 151
mobile agent（移动代理）, 38
mobile code（移动代码）, 17, 19, 37,62
security threats（安全威胁）, 256
mobile computing（移动计算）, 6~7
mobile devices（移动设备）, 40
mobileIP（移动IP）, 102~104
mobility transparency（移动透明性）, 23,315
model（模型）,
failure（故障）, 53~58
fundamental（基础）, 47~62
interaction（交互）, 49~53
security（安全性）, 58~62
MPEG compression（MPEG 压缩）, 618
MPEG video compression（MPEG 视频压缩）, 612, 613
MTU（MTU，参见maximum transfer unit）,
multicast（组播）, 436~450
atomic（原子）, 444
basic（基本的）, 438
causally ordered delivery（因果排序的发送）, 443, 448
FIFO-ordered delivery（FIFO排序的发送）, 442, 445
for discovery services（发现服务的）, 154
for event notifications（事件通知的）, 154
for fault tolerance（容错的）, 154, 568, 570
for highly available data（高可用的数据的）, 154
for replicated data（复制数据的）, 157

group（组），436
ordered（排序的），442~450
reliable（可靠的），439~442
to overlapping groups（给重叠组的），450
totally ordered delivery（全排序的发送），443, 445
multicast group（组播组），154
multimedia applications（多媒体应用），119
multimedia stream（多媒体流），608~611,613
burstiness（分布式内存），618
typical bandwidths（典型带宽），612
multiprocessor（多处理器），
distributed memory（分布式内存），637
Mach support for（Mach 支持），701
shared memory（共享内存），211
Non-Uniform Memory Access (NUMA)（非一致内存访问(NUMA)），637
Munin（Munin系统），661~663
mutual exclusion（互斥），423~431
between processes in a ring（环中进程之间的），426
by central server（中央服务器），425
Maekawa's algorithm（Maekawa算法），429
token（令牌），426
using multicast（使用组播），427~429

N

Nagle's algorithm（Nagle算法），105
name（名字），354, 356
component（成分），358
prefix（前缀），358
pure（纯的），354
unbound（解开），358
name resolution（名字解析），354, 357, 361
name service（名字服务），354~383
caching（高速缓冲存储），363, 367
CORBA naming service（CORBA名字服务），688~690
heterogeneity（异构性），360
navigation（导航），361~363
replication（复制），366
name space（命名空间），358
naming domain（命名域），359
navigation（导航），
multicast（组播），362
server-controlled（服务器控制的），363
Needham-Schroeder protocol（Needham-Schroeder协议），292~293
negative acknowledgement（否定的确认），440
Nemesis（Nemesis），246
nested transaction（嵌套事务），480~482, 548
locking（加锁），489
recovery（恢复），548~549
two-phase commit protocol（两阶段提交协议），524~528
timeout actions（超时动作），528
netmsgserver（netmsgserver，参见 Mach, network server），
network（网络），
ATM(Asynchronous Transfer Mode)（异步传输模式），71, 74, 76, 79, 86, 111, 119~121
bridge（网桥），88
circuit switching（电路交换），75
congestion control（拥塞控制），86
CSMA/CA（具有避免冲突的载波侦听多路访问），118
CSMA/CD（具有检测冲突的载波侦听多路访问），112
data streaming（数据流），74
data transfer rate（数据传输率），67
Ethernet（以太网），88, 111~116
frame relay（帧中继），75
gateway（网关），72
IEEE 802 standards（IEEE 802标准），111
IEEE802.11 wireless LAN（IEEE 802.11无线局域网），116~119
Internet（因特网），80,90~110
IP multicast（IP组播），89
latency（延迟，等待时间），67
local area（局域），70
metropolitan area（城域），71
packet assembly（包装配），79
packet switching（包交换），75
packets（数据包），73
performance parameters（性能参数），67
port（端口），80
protocol（协议，参见 protocol），
reliability requirements（可靠性需求），68
requirements（需求），67~70
router（路由器），88
routing（路由），71, 82~86, 97, 97~99
scalability requirements（伸缩性需求），68
security requirements（安全性需求），69
TCP/IP（TCP/IP），90
total system bandwidth（系统总带宽），67
transport address（传输地址），80
tunnelling（隧道法），89
WaveLAN（WaveLAN），111

wide area （广域），71
wireless（无线），71
network computer（网络计算机），39
Network File System (NFS)（网络文件系统(NFS)），317, 323~335
Automounter（自动安装器），328
benchmarks（基准测试软件），333
enhancements（增强），345
hard and soft mounting （硬安装和软安装），327
Kerberos authentication （Kerberos认证），332
mount service （安装服务），327
performance（性能），333
Spritely NFS（Spritely NFS），345
virtual file system(VFS)（虚拟文件系统(VFS)），324
v-node（v-结点），325
WebNFS（WebNFS），346
Network Information Service (NIS)（网络信息服务），569
network layer（网络层），79
network operating system（网络操作系统），208
network partition （网络分区），421,596~597
primary（主的），561
virtual（虚拟的），600~603
network phone application （网络电话应用），610
Network Time Protocol（网络时间协议），393~396
network transparency（网络透明性），24
NIS （NIS，参见Network Information Service），
NNTP（网络新闻传输协议），90
nomadic computing （游动计算），6
non-blocking send （非阻塞发送），128
nonce（当前时刻值），292
non-preemptive scheduling （非抢占式调度），227
non-repudiation（不可抵赖），258, 282
notification（通知），187~194
NQNFS (Not Quite NFS), 346
NTP （NTP，参见Network Time Protocol）

O

object（对象），43
activation （激活），180
active（主动），180
distributed（分布式的），171
model （模型），172
factory（工厂），180
interface（接口），43
location （位置），182
model（模型），170
CORBA（CORBA），671
passive （被动的），180
persistent（持久的），181
protection（保护），58
reference（引用），170
remote （远程），172
remote reference（远程引用），172
object adapter（对象适配器），678
object request broker (ORB)（对象请求代理），670
object serialization, in Java（对象序列化，在Java中），142~144
OMG (Object Management Group)（对象管理工作组），670
omission failure（遗漏故障），54
communication （通信），54
process（进程），54
one-copy serializability（单副本串行化），591
one-way function（单向函数），272, 273
one-way hash function（单向散列函数），286
open distributed systems（开放的分布式系统），18
open shortest path first (OSPF)（开放最短路径优先算法），97
open system（开放系统），18, 242
open systems interconnection （开放系统互联，参见OSI Reference Model），
openness（开放性），17~18,315
operating system（操作系统），
architecture（体系结构），242~246
communication and invocation support （通信和调用支持），231~242
policy and mechanism（策略和机制），243
processes and threads support （进程和线程支持），214~231
operation（操作），
multicast（组播），153
operational transformation （操作变换），582
optimistic concurrency control（乐观并发控制），497~501
backward validation （后向有效性），499
comparison of forward and backwarkd validation （前向有效性与后向有效性的比较），501
forward validation（前向有效性），500
update phase （更新阶段），498
validation （有效性），498
working phase（工作阶段），497
Orca（Orca），639
OSI Reference Model（OSI参考模型），78
output parameters （输出参数），168

P

packet assembly （包装配）, 79
packet switching network （包交换网络，参见network, packet switching）,
packets（包，参见network, packets ）,
page fault（缺页故障）, 219
passive object（被动对象）, 180
passive replication （被动复制，参见primary-backup replication ）,
perform, of memory access（执行，内存访问的）, 659
performance issues （性能问题）, 44
performance transparency（性能透明性）, 23, 315
persistent connection （一致性连接）, 151,237
persistent object （持久对象）, 181
  in CORBA persistent object service（在CORBA持续对象服务中）, 687
persistent object store（持久对象存储）,
  Khazana system（Khazana 系统）, 311~312
  PerDiS system（PerDiS 系统）, 311~312
  persistent Java（持久Java）, 312
Petal distributed virtual disk system（Petal 分布式虚拟磁盘系统）, 350
phantom deadlock （假象死锁）, 533
physical layer （物理层）, 78, 79
pipelined RAM（管道RAM）, 663
plaintext（明文）, 272
Plan（计划，规划）, 9 361
platform（平台）, 32, 209
play time, for multimedia data elements（娱乐时间，适于多媒体数据元素）, 74
POP（从邮件服务器获得邮件的协议）, 90
port（端口）, 80
  in Mach （在Mach中）, 703~705
port mapper（端口映射器）, 186
POTS (plain old telephone system)（老式电话系统）, 75
PPP（点对点协议）, 79, 90, 92
precedence graph （优先级图）, 597
preemptive scheduling（抢占调度）, 227
premature write （过早写入）, 479
presentation layer（表示层）, 79
Pretty Good Privacy (PGP)（良好隐私（一种加密电子邮件的方案））, 290
primary copy（主副本）, 334
primary-backup replication （主备份复制）, 568~570
principal（主体）, 58
process（进程）, 214~231
  correct（正确的）, 422
  creation（生成）, 217~220
  creation cost（创建开销）, 224
  migration （迁移）, 218
  multi-threaded（多线程的）, 214
  threats to （威胁）, 59
  user-level（用户级）, 213
process migration（进程迁移）, 218
processor consistency （处理器一致性）, 663
promise（承诺）, 240
protection（保护）, 212~213
  and type-safe language（类型安全的语言）, 213
  by kernel （内核的）, 213
protection domain（保护域）, 267
protocol（协议）, 76~82
  application layer（应用层）, 77, 79
  ARP（地址解析协议）, 96
  data link layer（数据链路层）, 79
  dynamic composition （动态组成）, 232
  FTP （文件传输协议）, 79, 80, 90, 109
  HTTP（超文本传输协议）, 79, 80, 81, 90, 91, 107
  Internetwork layer（互联网络层）, 79
  IP（因特网协议）, 79, 80, 95~104
  IPv4（第4版因特网协议）, 92
  IPv6（第6版因特网协议）, 74, 89,100~102
  layers（层）, 77
  mobileIP（移动IP）, 102~104
  network layer（网络层）, 79
  NNTP（网络新闻传输协议）, 90
  operating system support for （操作系统支持）, 232
  physical layer（物理层）, 78, 79
  POP （从邮件服务器获得邮件的协议）, 90
  PPP（点对点协议）, 79, 90, 92
  presentation layer（表示层）, 79
  session layer （会话层）, 79
  SMTP（简单邮件传输协议）, 79, 90
  SSL（加密套接字协议层）, 79, 90
  stack（堆栈）, 78, 232
  suite （组）, 78
  TCP （传输控制协议）, 79, 104~106
  TCP/IP（TCP/IP协议）, 90
  transport（传输）, 77
  transport layer（传输层）, 79
  UDP（用户数据报协议）, 79, 91, 104
  WAP（无线应用协议）, 91
proxy（代理）, 178
  in CORBA（在CORBA中）, 678
proxy server （代理服务器）, 36
public key （公开密钥）, 261
public-key certificate （公开密钥证书）, 264,266
public-key cryptography （公钥密码学）, 279~281

publish-subscribe（出版－订阅），187

Q

QoS（QoS，参见quality of service）
quality of service（服务质量），45
quality of service management（服务质量管理），608,614~623
quality of service negotiation（服务质量协商），615,616~622
quality of service parameters（服务质量参数），616
quorum consensus（法定数共识），592, 598~600

R

randomization（随机），461
reachability（可达性），404
Real Time Transport Protocol (RTP)（实时传输协议），74
real-time scheduling（实时调度），624
receive-omission failures（接收遗漏故障），55
recovery（恢复），478~480, 539~549
  cascading abort（连锁放弃），479
  dirty read（读取脏数据），478
  from abort（从回滚中），478
  intentions list（意图列表），540, 545
  logging（日志），540~543
  nested transactions（嵌套事务），548~549
  of two-phase commit protocol（两阶段提交协议的），546~549
  premature write（过早写入），479
  shadow versions（影子版本），544~545
  strict executions（严格执行），480
  transaction status（事务状态），540, 545
recovery file（恢复文件），539~549
  reorganization（重组织），543,548
recovery from failure（故障恢复），22
recovery manager（恢复管理器），539
redundancy（冗余），22
redundant arrays of inexpensive disks (RAID)（廉价磁盘冗余阵列（RAID））, 348
reflection（反射），
  in Java remote method invocation（在Java远程方法调用中），201
  Java（Java），143
region（区域，参见address space），
registering interest（注册兴趣），187
release consistency（释放一致性），657~661
reliable channel（可靠的通道），421
reliable communication（可靠的通信），57
  integrity（完整性），57
  validity（有效性），57
reliable multicast（可靠的组播），158, 439~442
remote interface（远程接口），168, 172, 173
  in Java remote method invocation（在Java远程方法调用中），195
remote method invocation（远程方法调用），8,172
  communication module（通信模块），177
  CORBA（CORBA），671~677
  dispatcher（调度器），179
  duplicate filtering（重复过滤），174
  implementation（实现），177~182
  Java case study（Java实例研究），194~202
  null（空），234
  parameter and result passing in Java（在Java中的参数和结果传递），195
  performance of（性能，参见invocation mechanism, performance of），
  proxy（代理），178
  remote reference module（远程引用模块），178
  retransmission of replies（应答重传），174
  retry request message（重试请求消息），174
  semantics（语义），174
  skeleton（骨架），179
  transparency（透明度），176
remote object（远程对象），172
remote object reference（远程对象引用），144,172,173
  in CORBA（在CORBA中），684
remote object table（远程对象表），178
remote procedure call（远程过程调用），183~187
  exchange protocols（交换协议），148
  lightweight（轻量级），237~239
  null（空），234
  performance of（性能，参见invocation mechanism, performance of），
  queued（排队的），242
remote reference module（远程引用模块），178
replaying attack（重播攻击），255
replica manager（复制管理器），557
replication（复制），46,553~603
  active（主动的），570~572
  available copies（可用的副本），592, 594~596
    with validation（带验证的），597~598
  of files（文件的），315
  primary-backup（主备份），568~570
  quorum consensus（法定共识），592, 598~600
  transactional（事务性的），591~603

transparency （透明的）, 555
virtual partition （虚拟分区）, 592, 600~603
replication transparency （复制透明性）, 23
reply message （应答消息）, 147
request message （请求消息）, 147
request-reply protocol （请求 - 应答协议）, 146~153
doOperation （doOperation）, 146
duplicate request message （重复的请求消息）, 148
failure model（故障模型）, 147
getRequest （getRequest）, 146
history of server operations （服务器操作的历史）, 148
lost reply messages（丢失的应答消息）, 148
sendReply （sendReply）, 146
timeout（超时）, 148
use of TCP（TCP的使用）, 149
resource（资源）, 2
invocation upon （调用）, 210
resource bandwidth （资源带宽）, 609
resource management (for multimedia)（资源管理（多媒体））, 623~625
Resource Reservation Protocol (RSVP)（资源保留协议(RSVP)）, 74
resource sharing （资源共享）, 7~9
responsiveness（相应性）, 44
RFC（RFC）, 17
RIP-1（路由信息协议1）, 85, 97
RIP-2（路由信息协议2）, 97
RMI（RMI，参见remote method invocation ）,
RMIRegistry（RMIRegistry）, 196
router（路由器）, 87, 88
router information protocol (RIP)（路由信息协议）, 84
routing （路由，参见network，routing）,
RPC （RPC，参见remote procedure call）,
RPC exchange protocols（RPC交换协议）, 149
RR （请求 - 应答协议）, 149
RRA（请求 - 应答 - 确认协议）, 149
RSA public key encryption algorithm（RSA公开密钥加密算法）, 289
RSVP protocol for bandwidth reservation （带宽预留的RSVP 协议）, 622
run （运行）, 404

S

safety property （安全特性）, 405
sandbox model of protection （保护的沙盒模型）, 256
scalability （可伸缩性）, 19~21, 315
scaling transparency（伸缩透明性）, 24, 315
scheduler activation （调度活动）, 230
scope consistency （作用域一致性）, 664
Scrip 304
secret key（保密密钥）, 261
secure channel （安全通道）, 61
secure digest function （安全摘要函数）, 265
secure hash function （安全散列函数）, 283
Secure Sockets Layer (SSL)（安全套接字层(SSL)）, 298~303
security （安全性）, 18, 19
"Alice and Bob" names for protagonists（角色名字如"Alice 和 Bob"）, 254
denial of service attack（拒绝服务攻击）, 255
design guidelines（设计原则）, 259
eavesdropping attack （窃听攻击）, 255
in CORBA security service（在CORBA安全服务中）, 693~694
in Java （在Java中）, 256
information leakage models（信息泄露模型）, 257
man-in-the-middle attack（中间人攻击）, 255
masquerading attack（伪装攻击）, 255
message tampering attack （消息篡改攻击）, 255
replaying attack （重发攻击，重播攻击）, 255
threats from mobile code（来自移动代码的威胁）, 256
threats: leakage, tampering, vandalism（威胁：泄露，篡改，恶意破坏）, 255
Transmission Layer Security (TLS)（传输层安全，参见Secure Sockets Layer）, 299
trusted computing base（可信赖的计算基础）, 260
security mechanism （安全机制）, 252
security model（安全模型）, 58~62,253
security policy （安全策略）, 252
send operation, non-blocking （发送操作，非阻塞的）, 128
sender order（发送方顺序）, 129
send-omission failures（发送遗漏故障）, 55
sequencer（顺序者）, 445
sequential consistency（顺序一致性）, 567
of distributed shared memory（分布式共享内存的）, 645
serial equivalence （串行相等性）, 471, 475
serialization（串行化）, 142
server（服务器）, 8
multi-threaded（多线程的）, 220
multi-threading architecture（多线程体系结构）, 221
throughput（吞吐量）, 220
server port（服务器端口）, 130
server stub procedure （服务器存根程序）, 184

serverless file system (xFS)（无服务器文件系统(xFS)），349
service（服务），8
  fault-tolerant（容错），555, 565~572
  highly available（高可用的），572~591
service interface（服务接口），168, 183
servlet（servlet程序），226
session key（会话关键字），263
session layer（会话层），79
SHA secure hash algorithm（SHA安全散列算法），287, 289
shadow versions（影子版本），544~545
shared whiteboard（共享的白板），194
signature of method（方法签名），170
Simple Public-key Infrastructure (SPKI)（简单公开密钥基础设施），288
simple service discovery protocol（简单服务发现协议），372
skeleton（骨架），179
  in CORBA（在CORBA中），678
SMTP（简单邮件传输协议），79, 90
socket（套接字），129
  address（地址），158
  bind（绑定），158
  close（关闭），158
  connect（连接），137
  datagram communication（数据报通信），159
  pair（对），158
  stream communication（流通信），160
  system call（系统调用），158, 159, 160
socket address（套接字地址），158
sockets（套接字），129~130
software interrupt（软中断），223
Solaris（Solaris系统），
  lightweight process（轻量进程），229
speaks for relation (in security)（关系表达（在安全性中）），270
SPIN（SPIN），245
spontaneous networking（自发网络），40~42
  discovery service（发现服务），371
spring naming service（弹性命名服务），360
Spritely NFS（Spritely NFS），345
SSL（安全套接字层），79,90
SSL（SSL，参见Secure Sockets Layer），
starvation（饥饿问题），424, 501
state machine（状态机），557
state transfer（状态转换），563
stateless server（无状态服务器），320
stream adaptation（流适应），625
stream cipher（流密码），274~275
strict executions（严格执行），480
stub procedure（存根程序），184
  in CORBA（在CORBA中），678
subscribe（订阅），187
subsystem（子系统），244
Sun Network File System (NFS)（Sun网络文件系统，参见Network File System），
Sun RPC（Sun远程过程调用），184~187, 323
  external data representation（外部数据表示），186
  interface definition language（接口定义语言），185
  rpcgen（rpcgen），185
supervisor（管理器），212
supervisor mode（管理模式），213
symmetric cryptography（对称密码学），272
symmetric processing architecture（对称处理结构），211
synchronization memory accesses（同步内存访问），658
synchronization of clocks（时钟同步），389~396
  Berkeley algorithm（Berkeley算法），393
  Cristian's algorithm（Cristian算法），391~393
  external（外部的），389
  in a synchronous system（在同步系统中），391
  internal（内部的），389
  Network Time Protocol（网络时间协议），393~396
synchronization, of server operations（同步，服务器操作的），468
synchronous communication（同步通信），128
synchronous distributed system（同步分布式系统），391, 415, 420, 423, 450, 455
system call trap（系统调用陷阱），213

T

task（任务，参见Mach, task），701
TCP（传输控制协议），79, 104~106, 134~138
  and request-reply protocols（和请求-应答协议），149, 237
  API（API），134
  failure model（故障模型），135
  Java API（Java API），136~138
  UNIX API（UNIX API），160
TCP/IP（TCP/IP，参见protocol, TCP/IP），
TEA (Tiny Encryption Algorithm)（微加密算法），276~277,289
termination（终止性），
  of consensus and related problems（在共识和相关问题中），452~453
termination detection（终止检测），401

thin client（瘦客户），39
thrashing（系统颠簸），649
thread（线程），128, 214, 220~231
  blocking receive（阻塞型接收），128
  C（C），225
  in client（在客户端的），222
  comparison with process（与进程的比较），222
  creation cost（生成耗费），224
  implementation （实现），228~231
  in server（在服务器端的），220
  Java （Java），225~228
  kernel-level（内核级），229
  on multiprocessor（多处理器上的），222
  multi-threading architecture（多线程结构），221
  POSIX（POSIX），225
  programming（编程），225~228
  scheduling（调度），227
  switching（切换），224
  synchronization （同步），227
  worker（工作者），221
throughput（吞吐量），44
throw exception （抛出异常），171
Ticket Granting Service (TGS)（票证授予服务），294
ticket, of authentication （票证，认证的），263
Tiger video file server（Tiger视频文件服务器），627
time（时间），385~400
  logical（逻辑的），397~400
time-based data streams（基于时间的数据流），613
time-critical data（关键时间数据），45
timeouts（超时），54
timestamp （时间戳），
  Lamport（Lamport），398
  vector（向量），399
timestamp ordering（时间戳排序），501~508
  in distributed transaction（在分布式事务中），529
  multiversion （多版本），506~508
  operation conflicts（操作冲突），502
  read rule（读规则），504
  write rule（写规则），503
timing failure（时序故障），56
TLS（TLS，参见Secure Sockets Layer），
tolerating failure（容错），21
total ordering（全排序），387, 399
  of multicast delivery（组播传递的），443
  of request handling（请求处理的），558
totally ordered multicast（全排序组播），158
traffic shaping (for multimedia data)（流量调整（对多媒体数据）），619
trampolining（蹦床），702
transaction （事务），469~482
  abort（放弃），472
  abort Transaction （abort 事务），472
  ACID properties（ACID属性），471
  closeTransaction （closeTransaction），472
  commit（提交），472
  concurrency control（并发控制，参见 concurrency control），
  failure model（故障模型），469
  in CORBA transaction service （在CORBA的事务服务中），687
  openTransaction （openTransaction），472
  recovery（恢复，参见recovery），
  serial equivalence（串行相等性），475
  with replicated data（重复数据的），591~603
transaction status（事务状态），540, 545
Transmission Layer Security （传输层安全，参见Secure Sockets Layer），
transparency（透明性），23~25
  access（访问），23,315,333
  concurrency（并发），23
  failure（故障），23
  in remote method invocation （在远程方法调用中），176
  location （位置），23, 315, 334
  mobility（移动），23, 315
  network（网络），24
  performance（性能），23, 315
  replication （复制），23, 555
  scaling（升级），24, 315
transport address（传输地址），80
transport layer（传输层），79
transport protocol（传输协议），77
trap-door function （陷门函数），273
triple-DES encryption algorithm（三重DES加密算法），289
trusted computing base （可信赖的计算基础），260
tunnelling（隧道法），89
tuple space （元组空间），641~642
two-phase commit protocol （两阶段提交协议），520~523
  nested transaction （嵌套事务），524~528
    flat commit（平面提交），527
    hierarchic commit（层次提交），526
  performance（性能），523
  recovery （恢复），546~549
  timeout actions（超时动作），521

U

ubiquitous computing（无处不在计算），6
UDP（用户数据报协议），79, 91, 104, 130~133
  failure model（故障模型），131
  for request-reply communication（对请求～应答通信），150, 237
  Java API（Java API），132~133
  UNIX API（UNIX API），159
  use of（使用），131
UFID（UFID，参见unique file identifier），
uniform memory consistency model（统一内存一致性模型），663
uniform property（一致性特性），442
Uniform Resource Characteristic（统一资源特征），356
Uniform Resource Identifier（统一资源标识符），356
Uniform Resource Locator（统一资源定位器），11~13,356
Uniform Resource Name（统一资源名称），356
unique file identifier (UFID)（文件惟一标识符(UFID)），318, 339
universal plug and play（通用即插即用），372
Universal Transfer Format（通用传输格式），143
UNIX（UNIX），
  i-node（i-结点），325
  signal（信号），223
  system call（系统调用），
    accept（accept），161
    bind（bind），159
    connect（connect），161
    exec（exec），217
    fork（fork），217, 218, 714
    listen（listen），161
    read（read），161
    recvfrom（recvfrom），159
    sendto（sendto），159
    socket（socket），159,160
    write（write），161
unmarshalling（解码），139
unreliable multicast（不可靠的组播），155
upcall（上调），230
update semantics（更新语义），316, 341
URC（URC，参见Uniform Resource Characteristic），
URI（URI，参见Uniform Resource Identifier），
URL（URL，参见Uniform Resource Locator），
URN（URN，参见Uniform Resource Name），
user mode（用户模式），213
UTC（UTC，参见Coordinated Universal Time），
UTF（UTF，参见Universal Transfer Format），

V

V system（V系统）
  remote execution（远端执行），217
  support for groups（对组的支持），564
validity（有效性）
  of multicast delivery（组播传递的），439
  of reliable communication（可靠通信的），57
vector clock（时钟向量），399
vector timestamp（时间戳向量），399
  comparison（比较），400
  merging operation（合并操作），399
Verisign Corporation（Verisign公司），288
videoconferencing（视频会议），610, 613
  CU-SeeMe application（CU-SeeMe应用），610
  NetMeeting application（网络会议应用），610
video-on-demand service（视频点播服务），610
view, of group（视图，组的，参见group, view），
view-synchronous group communication（视图同步的组通信），562~564
virtual channel (in ATM networks)（虚通道（在ATM网络中）），119
virtual circuit（虚拟电路），81
virtual file system (VFS)（虚拟文件系统，参见Network File System (NFS), virtual file system），
virtual machine（虚拟机），17
virtual memory（虚拟内存），
  external pager（外部换页），716~719
  in Mach（Mach），702, 713~719
virtual partition（虚拟分区），600~603
virtual path (in ATM networks)（虚拟路径（在ATM网络中）），119
virtual private network (VPN)（虚拟私有网络），69, 110
virtual processor（虚拟处理器），230
virtually synchronous group communication（虚拟同步组通信），564
voting（投票），429

W

WAN（WAN，参见network, wide area），
WAP (Wireless Application Protocol)（无线应用协议），72, 91
WaveLAN（产品名（无线局域网的一种）），111
weak consistency（弱一致性），646, 664
Web（Web，参见World Wide Web），
web tunnel（Web隧道），271

web, caching （Web，缓存），46
web-based multimedia（基于网络的多媒体），610
WebNFS（WebNFS），346
wide area network （广域网，参见network, wide area），
window of scarcity (of resources for multimedia applications)（窗口限制（对于多媒体应用的资源）），611
wireless network （无线网，参见 network, wireless ），
WLAN (wireless local area network)（WLAN(无线局域网)，参见network, wireless ），
World Wide Web（万维网），9~15
WPAN(wireless personal area network) （WPAN(无线个域网)，参见network, wireless ）

X

X.500 directory service（X.500目录服务），378~382
  directory information tree（目录信息树），378
  LDAP（LDAP），382
X.509 certificate（X.509证书），287
XDR（XDR，参见Sun RPC），
  external data representation （外部数据表示），
xFS serverless file system（xFS 无服务器文件系统），349

[General Information]
书名=分布式系统概念与设计 （原书第3版）
作者=
页数=592
SS号=11180061
出版日期=

封面
书名
版权
前言
目录
正文