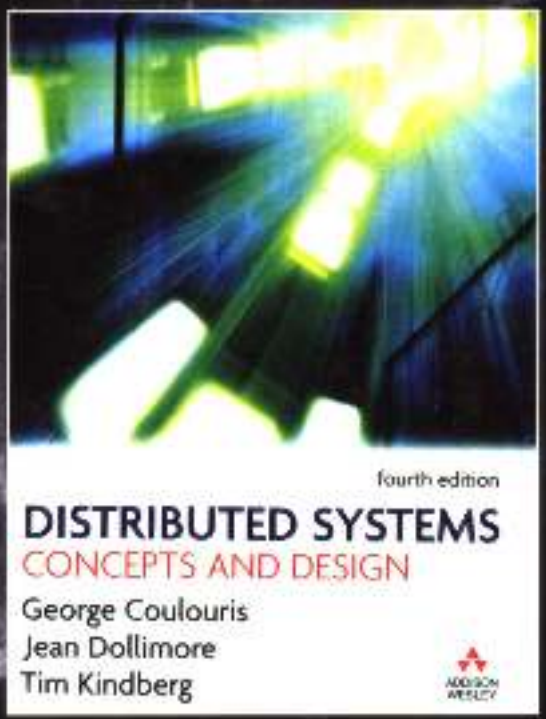
计 算 机 科 学 丛 书

原书第4版

分布式系统

概念与设计

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



Distributed Systems
Concepts and Design
Fourth Edition

机械工业出版社
China Machine Press

分布式系统概念与设计（原书第4版）

本书是衡量所有其他分布式系统教材的标准。

——Amazon.com.uk评

从移动电话到因特网，我们的生活越来越依赖于以无缝和透明的方式将计算机和其他设备链接在一起的分布式系统。本书全面介绍分布式系统的设计原理和实践及其新进展，并使用大量最新的实例研究来阐明分布式系统的设计与开发方法。

本书的前几版已经被爱丁堡大学、伊利诺伊大学、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、罗切斯特理工学院、北京大学等名校采纳为高级操作系统、计算机网络、分布式系统课程的教材。

本书网站www.cdk4.net和www.pearsoned.co.uk/coulouris为学生和教师提供了丰富的学习及教学资源（源代码、参考文献、教学幻灯片、勘误等）。

第4版的新内容：

- 全新的三章内容，分别介绍对等系统、Web服务、移动与无处不在计算系统。
- 超过25个常见系统的实例研究，其中8个是全新的，包括对网格、Coolltown、蓝牙以及WiFi WEP协议的研究。
- 全面更新XML及其安全扩展的内容以及用于无处不在系统的AES加密标准与安全设计。

作者简介

George Coulouris 伦敦大学皇后玛利学院荣誉教授，剑桥大学计算机实验室的资深委任研究员



Jean Dollimore 伦敦大学皇后玛利学院的高级研究员



Tim Kindberg 英国布里斯托尔惠普实验室高级研究员



限中国大陆地区销售

上架指导：计算机/分布式系统

ISBN 978-7-111-22438-9

9 787111 224389

ISBN 978-7-111-22438-9

定价：69.00元

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

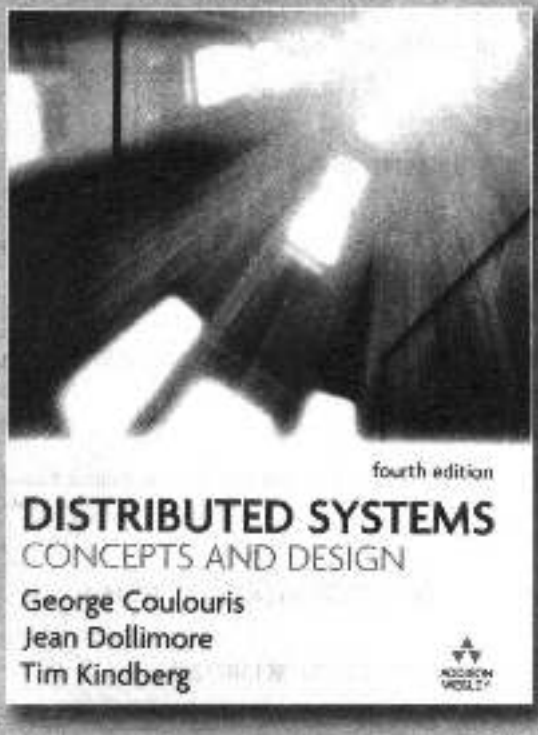
网上购书：www.china-pub.com

原书第4版
TP316.4/5=2
2008

计 算 机 科 学 丛 书

分布式系统
概念与设计

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



Distributed Systems
Concepts and Design
Fourth Edition

机械工业出版社
China Machine Press

本书旨在全面介绍因特网及其他常用分布式系统的原理、体系结构、算法和设计，内容涵盖分布式系统的相关概念、安全、数据复制、组通信、分布式文件系统、分布式事务等，以及相关的前沿主题，包括Web服务、网格、移动系统和无处不在系统等。

本书素材丰富、内容充实、深入浅出，每章后都有相关的习题，并有配套网站提供本书的学习和教学资源。本书可作为相关专业本科生及研究生的分布式系统课程的教材，也可供广大技术人员参考。

George Coulouris, Jean Dollimore, Tim Kindberg: Distributed Systems: Concepts and Design, Fourth Edition(ISBN: 0-321-26354-5).

Copyright © Addison Wesley Publishers Limited 1988, 1994, © Pearson Education Limited 2001, 2005.

This translation of Distributed Systems: Concepts and Design, Fourth Edition(ISBN: 0-321-26354-5) is published by arrangement with Pearson Education Limited.

All rights reserved.

本书中文简体字版由英国Pearson Education培生教育出版集团授权出版。

本书版权登记号：图字：01-2005-4697

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

图书在版编目（CIP）数据

分布式系统：概念与设计（原书第4版）/（英）库劳里斯（Coulouris, G.）等著；金蓓弘等译. －北京：机械工业出版社，2008.1

（计算机科学丛书）

书名原文：Distributed Systems: Concepts and Design, Fourth Edition

ISBN 978-7-111-22438-9

Ⅰ. 分…　Ⅱ. ①库…　②金…　Ⅲ. 分布式系统　Ⅳ. TP316.4

中国版本图书馆CIP数据核字（2007）第150732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：朱劼

北京京北制版厂印刷 · 新华书店北京发行所发行

2008年1月第1版第1次印刷

184mm × 260mm · 36.25印张

定价：69.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

译 者 序

随着网络技术的发展和计算机应用的深入，分布式系统已成为目前主流的软件系统。

本书介绍了分布式系统的概念、基本原理和核心技术，内容涉及通信、中间件、系统基础设施、分布式数据处理以及分布式算法等。通过阅读本书，读者既可以从系统层面上了解分布式系统构造的基本原理，又可以从算法层面上获知分布式系统构造的核心技术。

本书素材广泛、内容充实、叙述深入浅出、条理清楚，每章后都配有练习题，并有配套网站提供大量学习和教学资料。因此，本书可以作为高等院校高年级本科生和研究生“分布式计算”及相关课程的教材或参考书，也可供分布式计算领域的科研人员阅读、参考。

本书第3版和第4版都是由中国科学院软件研究所金蓓弘研究员组织和主持翻译的。

本书第3版由若干同仁通力合作、共同翻译而成。前言、第1、2、4、10、11章由金蓓弘翻译，第6、8、15、16、18章由李剑博士翻译，第12、13、14章由丁柯博士翻译，第5、17章由刘绍华博士翻译，第3、9章由阮彤博士翻译，第3章由王仲玉翻译，第7章由刘志军翻译，由金蓓弘通校了第3版全部译文。

本书第4版新增了三章。其中，第10章由张发恩翻译，第16章由张英翻译，第19章由臧志翻译。由金蓓弘、曹冬磊博士通校了第4版全部译文。

由于时间和水平所限，翻译中不当之处在所难免，欢迎广大读者提出批评和指正。

感谢机械工业出版社华章分社在引进、编辑、出版本书中所做的努力。

译 者

2007年8月于北京

前　言

在因特网和Web走向成熟、能够支持多种分布式系统之际，本书的第4版问世了。如今，分布式系统的规模已远远超过了本书第3版出版时的预期。

本书旨在介绍因特网和其他分布式系统所蕴涵的原理、体系结构、算法和设计。前两章是概念上的简介，概括分布式系统的特征和必须在设计中解决的挑战：可伸缩性、异构性、安全性和故障处理。这两章也给出了理解进程交互、故障和安全的抽象模型。后续几章关注连网、进程间通信、远程调用和中间件、操作系统和命名。

接着，我们论及一些比较成熟的主题，包括安全、数据复制、组通信、分布式文件系统、分布式事务、CORBA、分布式共享内存和多媒体系统。此外，还会讨论一些新的主题：Web服务、XML、网格、对等、移动和无处不在系统。与这些主题相关的算法将在相关主题中讨论。我们还将另辟几章讨论时序、协调和协定。

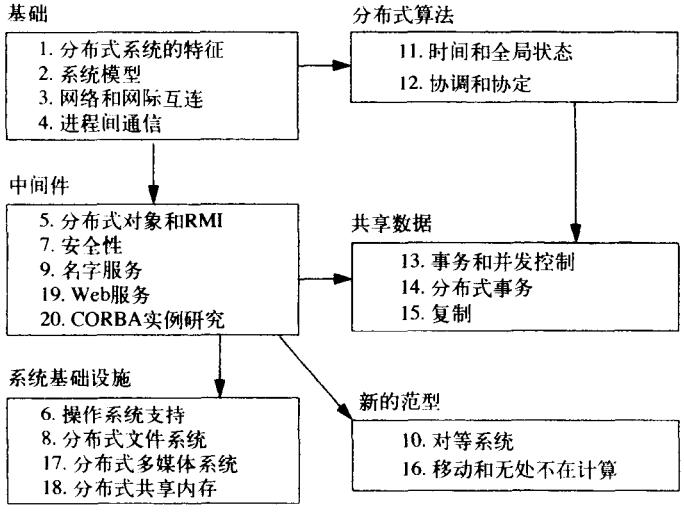
目的和读者群

本书可作为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，描述成功开发系统的方法。本书覆盖的领域有足够的宽度和广度，使得读者在读完本书后能继续研究分布式系统文献中的大多数研究论文。

本书希望读者具有面向对象编程、操作系统以及基本的计算机体系结构知识。本书涵盖与分布式系统有关的计算机网络的知识，包括因特网、广域网、局域网和无线网的基本技术。本书中的大部分算法和接口用Java描述，有一小部分用ANSI C描述。为了表述上的简洁明了，还将使用一种从Java/C中派生出来的伪代码。

本书的组织

下图表明本书的各章可划分为六个主题。它说明了本书的结构，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。



目　录

译者序
前言

第1章　分布式系统的特征 ……1
1.1　简介 ……1
1.2　分布式系统的实例 ……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计算 ……3
1.3　资源共享和Web ……5
1.4　挑战 ……10
1.4.1　异构性 ……11
1.4.2　开放性 ……11
1.4.3　安全性 ……12
1.4.4　可伸缩性 ……13
1.4.5　故障处理 ……14
1.4.6　并发性 ……15
1.4.7　透明性 ……15
1.5　小结 ……16
练习 ……17
第2章　系统模型 ……19
2.1　简介 ……19
2.2　体系结构模型 ……20
2.2.1　软件层 ……20
2.2.2　系统体系结构 ……22
2.2.3　变体 ……23
2.2.4　接口和对象 ……27
2.2.5　分布式体系结构的设计需求 ……27
2.3　基础模型 ……30
2.3.1　交互模型 ……31
2.3.2　故障模型 ……34
2.3.3　安全模型 ……36
2.4　小结 ……39
练习 ……40
第3章　网络和网际互连 ……42
3.1　简介 ……42
3.2　网络类型 ……44
3.3　网络原理 ……46
3.3.1　数据包的传输 ……47
3.3.2　数据流 ……47
3.3.3　交换模式 ……47
3.3.4　协议 ……48
3.3.5　路由 ……52
3.3.6　拥塞控制 ……54
3.3.7　网际互连 ……55
3.4　因特网协议 ……57
3.4.1　IP寻址 ……59
3.4.2　IP协议 ……60
3.4.3　IP路由 ……62
3.4.4　IPv6 ……65
3.4.5　移动IP ……67
3.4.6　TCP和UDP ……68
3.4.7　域名 ……69
3.4.8　防火墙 ……70
3.5　实例研究：以太网、WiFi、蓝牙和ATM ……72
3.5.1　以太网 ……73
3.5.2　IEEE 802.11无线LAN ……76
3.5.3　IEEE 802.15.1蓝牙无线PAN ……78
3.5.4　异步传输模式网络 ……80
3.6　小结 ……82
练习 ……82
第4章　进程间通信 ……84
4.1　简介 ……84
4.2　因特网协议的API ……85
4.2.1　进程间通信的特征 ……85
4.2.2　套接字 ……86
4.2.3　UDP数据报通信 ……87
4.2.4　TCP流通信 ……90
4.3　外部数据表示和编码 ……93
4.3.1　CORBA的公共数据表示 ……94
4.3.2　Java对象序列化 ……95
4.3.3　可扩展标记语言 ……97
4.3.4　远程对象引用 ……99
4.4　客户－服务器通信 ……100
4.5　组通信 ……105

4.5.1 IP组播——组通信的实现 ……106
4.5.2 组播的可靠性和排序 ……108
4.6 实例研究：UNIX中的进程间通信 ……108
4.6.1 数据报通信 ……109
4.6.2 流通信 ……110
4.7 小结 ……110
练习 ……111
第5章 分布式对象和远程调用 ……114
5.1 简介 ……114
5.2 分布式对象间的通信 ……116
5.2.1 对象模型 ……117
5.2.2 分布式对象 ……117
5.2.3 分布式对象模型 ……118
5.2.4 RMI的设计问题 ……120
5.2.5 RMI的实现 ……122
5.2.6 分布式无用单元收集 ……125
5.3 远程过程调用 ……126
5.4 事件和通知 ……129
5.4.1 分布式事件通知的参与者 ……131
5.4.2 实例研究：Jini分布式事件规约 …132
5.5 实例研究：Java RMI ……133
5.5.1 创建客户和服务器程序 ……136
5.5.2 Java RMI的设计和实现 ……138
5.6 小结 ……139
练习 ……139
第6章 操作系统支持 ……142
6.1 简介 ……142
6.2 操作系统层 ……143
6.3 保护 ……144
6.4 进程和线程 ……145
6.4.1 地址空间 ……146
6.4.2 新进程的生成 ……147
6.4.3 线程 ……149
6.5 通信和调用 ……157
6.5.1 调用性能 ……158
6.5.2 异步操作 ……162
6.6 操作系统的体系结构 ……164
6.7 小结 ……167
练习 ……167
第7章 安全性 ……169
7.1 简介 ……169
7.1.1 威胁和攻击 ……170
7.1.2 保护电子事务 ……172
7.1.3 设计安全系统 ……173
7.2 安全技术概述 ……174
7.2.1 密码学 ……175
7.2.2 密码学的应用 ……175
7.2.3 证书 ……177
7.2.4 访问控制 ……178
7.2.5 凭证 ……180
7.2.6 防火墙 ……181
7.3 密码算法 ……181
7.3.1 密钥（对称）算法 ……184
7.3.2 公钥（不对称）算法 ……186
7.3.3 混合密码协议 ……188
7.4 数字签名 ……188
7.4.1 公钥数字签名 ……189
7.4.2 密钥数字签名——MAC……189
7.4.3 安全摘要函数 ……190
7.4.4 证书标准和证书权威机构 ……191
7.5 密码实用学 ……192
7.5.1 密码算法的性能 ……192
7.5.2 密码学的应用和政治障碍 ……193
7.6 案例研究：Needham-Schroeder、Kerberos、TLS和802.11 WiFi……194
7.6.1 Needham-Schroeder认证协议 ……194
7.6.2 Kerberos ……195
7.6.3 使用安全套接字确保电子交易安全 ……199
7.6.4 IEEE 802.11 WiFi 安全设计中的缺陷 ……201
7.7 小结 ……203
练习 ……204
第8章 分布式文件系统 ……205
8.1 简介 ……205
8.1.1 文件系统的特点 ……207
8.1.2 分布式文件系统的需求 ……208
8.1.3 实例研究 ……209
8.2 文件服务体系结构 ……210
8.3 实例研究：SUN网络文件系统 ……214
8.4 实例研究：Andrew文件系统 ……222
8.4.1 实现 ……223
8.4.2 缓存的一致性 ……225
8.4.3 其他方面 ……227
8.5 最新进展 ……228
8.6 小结 ……232
练习 ……232
第9章 名字服务 ……234

9.1 简介 ……234
9.2 名字服务和域名系统 ……236
9.2.1 名字空间 ……237
9.2.2 名字解析 ……239
9.2.3 域名系统 ……241
9.3 目录服务 ……246
9.4 实例研究：全局名字服务 ……246
9.5 实例研究：X.500目录服务 ……248
9.6 小结 ……251
练习 ……252
第10章 对等系统 ……253
10.1 简介 ……253
10.2 Napster及其遗留系统 ……256
10.3 对等中间件 ……257
10.4 路由覆盖 ……259
10.5 路由覆盖实例研究：Pastry和Tapestry ……261
10.5.1 Pastry ……261
10.5.2 Tapestry ……266
10.6 应用实例研究：Squirrel、OceanStore和Ivy ……267
10.6.1 Squirrel Web缓存 ……267
10.6.2 OceanStore文件存储 ……269
10.6.3 Ivy文件系统 ……272
10.7 小结 ……274
练习 ……275
第11章 时间和全局状态 ……277
11.1 简介 ……277
11.2 时钟、事件和进程状态 ……278
11.3 同步物理时钟 ……279
11.3.1 同步系统中的同步 ……280
11.3.2 同步时钟的Cristian方法 ……281
11.3.3 Berkeley算法 ……281
11.3.4 网络时间协议 ……282
11.4 逻辑时间和逻辑时钟 ……284
11.5 全局状态 ……286
11.5.1 全局状态和一致割集 ……287
11.5.2 全局状态谓词、稳定性、安全性和活性 ……288
11.5.3 Chandy和Lamport的“快照”算法 ……289
11.6 分布式调试 ……291
11.6.1 观察一致的全局状态 ……293
11.6.2 判定可能的ϕ ……294
11.6.3 判定明确的ϕ ……294
11.6.4 在同步系统中判定可能的ϕ和明确的ϕ ……295
11.7 小结 ……296
练习 ……296
第12章 协调和协定 ……298
12.1 简介 ……298
12.2 分布式互斥 ……300
12.3 选举 ……305
12.4 组播通信 ……308
12.4.1 基本组播 ……309
12.4.2 可靠组播 ……310
12.4.3 有序组播 ……312
12.5 共识和相关问题 ……317
12.5.1 系统模型和问题定义 ……317
12.5.2 同步系统中的共识问题 ……320
12.5.3 同步系统中的拜占庭将军问题 ……320
12.5.4 异步系统的不可能性 ……323
12.6 小结 ……324
练习 ……325
第13章 事务和并发控制 ……327
13.1 简介 ……327
13.1.1 简单的同步机制（无事务） ……328
13.1.2 事务的故障模型 ……329
13.2 事务 ……329
13.2.1 并发控制 ……332
13.2.2 事务放弃时的恢复 ……334
13.3 嵌套事务 ……336
13.4 锁 ……337
13.4.1 死锁 ……342
13.4.2 在加锁机制中增加并发度 ……345
13.5 乐观并发控制 ……346
13.6 时间戳排序 ……349
13.7 并发控制方法的比较 ……353
13.8 小结 ……354
练习 ……355
第14章 分布式事务 ……359
14.1 简介 ……359
14.2 平面分布式事务和嵌套分布式事务 ……359
14.3 原子提交协议 ……361
14.3.1 两阶段提交协议 ……362
14.3.2 嵌套事务的两阶段提交协议 ……364
14.4 分布式事务的并发控制 ……367
14.4.1 加锁 ……367
14.4.2 时间戳并发控制 ……368

14.4.3 乐观并发控制 ……368
14.5 分布式死锁 ……369
14.6 事务恢复 ……374
14.6.1 日志 ……375
14.6.2 影子版本 ……377
14.6.3 为何恢复文件需要事务状态和意图列表 ……378
14.6.4 两阶段提交协议的恢复 ……378
14.7 小结 ……380
练习 ……381
第15章 复制 ……383
15.1 简介 ……383
15.2 系统模型和组通信 ……385
15.2.1 系统模型 ……385
15.2.2 组通信 ……386
15.3 容错服务 ……390
15.3.1 被动（主备份）复制 ……392
15.3.2 主动复制 ……393
15.4 高可用服务的实例研究：gossip体系结构、Bayou和Coda ……394
15.4.1 gossip体系结构 ……395
15.4.2 Bayou系统和操作变换方法 ……401
15.4.3 Coda文件系统 ……402
15.5 复制数据上的事务 ……407
15.5.1 复制事务的体系结构 ……407
15.5.2 可用拷贝复制 ……409
15.5.3 网络分区 ……410
15.5.4 带验证的可用拷贝 ……411
15.5.5 法定数共识方法 ……411
15.5.6 虚拟分区算法 ……413
15.6 小结 ……415
练习 ……415
第16章 移动计算和无处不在计算 ……417
16.1 简介 ……417
16.2 关联 ……423
16.2.1 发现服务 ……424
16.2.2 物理关联 ……427
16.2.3 小结和前景 ……428
16.3 互操作 ……428
16.3.1 易变系统的面向数据编程 ……429
16.3.2 间接关联和软状态 ……432
16.3.3 小结和前景 ……433
16.4 感知和上下文敏感 ……433
16.4.1 传感器 ……434
16.4.2 感知体系结构 ……434
16.4.3 位置感知 ……438
16.4.4 小结和前景 ……441
16.5 安全和私密性 ……442
16.5.1 背景 ……442
16.5.2 一些解决办法 ……443
16.5.3 小结和前景 ……447
16.6 自适应 ……447
16.6.1 内容的上下文敏感自适应 ……448
16.6.2 适应变化的系统资源 ……449
16.6.3 小结和前景 ……450
16.7 Cooltown实例研究 ……450
16.7.1 Web存在 ……451
16.7.2 物理超链接 ……452
16.7.3 互操作和eSquirt协议 ……454
16.7.4 小结和前景 ……455
16.8 小结 ……455
练习 ……456
第17章 分布式多媒体系统 ……458
17.1 简介 ……458
17.2 多媒体数据的特征 ……461
17.3 服务质量管理 ……462
17.3.1 服务质量协商 ……464
17.3.2 许可控制 ……467
17.4 资源管理 ……468
17.5 流适应 ……469
17.5.1 调整 ……470
17.5.2 过滤 ……471
17.6 实例研究：Tiger视频文件服务器 ……471
17.7 小结 ……474
练习 ……474
第18章 分布式共享内存 ……476
18.1 简介 ……476
18.1.1 消息传递机制和DSM ……477
18.1.2 DSM的实现方法 ……478
18.2 设计和实现问题 ……479
18.2.1 结构 ……479
18.2.2 同步模型 ……480
18.2.3 一致性模型 ……481
18.2.4 更新选项 ……483
18.2.5 粒度 ……485
18.2.6 系统颠簸 ……485
18.3 顺序一致性和Ivy实例研究 ……485
18.3.1 系统模型 ……486

18.3.2 写失效 ……487
18.3.3 失效协议 ……488
18.3.4 一个动态分布式管理器算法 ……489
18.3.5 系统颠簸 ……490
18.4 释放一致性和Munin实例研究 ……491
18.4.1 内存访问 ……491
18.4.2 释放一致性 ……492
18.4.3 Munin ……493
18.5 其他一致性模型 ……494
18.6 小结 ……495
练习 ……496
第19章 Web服务 ……498
19.1 简介 ……498
19.2 Web服务 ……499
19.2.1 SOAP ……501
19.2.2 Web服务与分布式对象模型的比较 ……504
19.2.3 在Java中使用SOAP ……505
19.2.4 Web服务和CORBA的比较 ……508
19.3 服务描述和Web服务接口定义语言 …509
19.4 Web服务使用的目录服务 ……512
19.5 XML安全性 ……513
19.6 Web服务的协作 ……516
19.7 实例研究：网格 ……517
19.7.1 World-Wide Telescope——一种网格应用 ……518
19.7.2 数据密集型科学应用的特征 ……518
19.7.3 开放的网格服务体系结构 ……519
19.7.4 一些网格应用的例子 ……521
19.7.5 Globus工具包 ……522
19.8 小结 ……523
练习 ……524
第20章 CORBA实例研究 ……526
20.1 简介 ……526
20.2 CORBA RMI ……527
20.2.1 CORBA客户和服务器实例 ……529
20.2.2 CORBA体系结构 ……532
20.2.3 CORBA接口定义语言 ……534
20.2.4 CORBA远程对象引用 ……537
20.2.5 CORBA语言映射 ……538
20.2.6 CORBA与Web的集成 ……538
20.3 CORBA服务 ……539
20.3.1 CORBA名字服务 ……540
20.3.2 CORBA事件服务 ……542
20.3.3 CORBA通知服务 ……543
20.3.4 CORBA安全服务 ……544
20.4 小结 ……544
练习 ……545
索引 ……548
参考文献[⊖]

⊖ 参考文献可从华章网站（www.hzbook.com）下载。

第1章 分布式系统的特征

分布式系统是其组件分布在连网的计算机上，组件之间通过传递消息进行通信和动作协调的系统。该定义导出了分布式系统的下列特征：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三个例子：

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

资源共享是构造分布式系统的主要动力。资源可以由服务器管理，由客户访问，或它们被封装成对象，由其他客户对象访问。作为一个资源共享的例子，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处理其组件的异构性、开放性（允许增加或替换组件）、安全性、可伸缩性（用户数量增加时能正常运行的能力）、故障处理、组件的并发性和透明性问题。

1.1 简介

计算机网络无处不在。因特网也是其中的一个，因为它是由许多种网络组成的。移动电话网、协作网、企业网、校园网、家庭网、车内网，所有这些，既可单独使用，又可相互结合，它们具有相同的本质特征，这些特征使得它们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连网的计算机的特征，给出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。

我们把分布式系统定义成一个其硬件或软件组件分布在连网的计算机上，组件之间通过传递消息进行通信和动作协调的系统。这个简单的定义覆盖了所有可部署连网计算机的系统。

由一个网络连接的计算机可能在空间上的距离不等。它们可能分布在地球上不同的洲，也可能在同一栋楼或同一个房间里。分布式系统有如下显著特征：

并发：在一个计算机网络中，执行并发程序是常见的行为。用户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会随着网络资源（例如，计算机）的增加而提高。在本书的许多地方将描述有效实施这种额外能力的方法。对共享资源的并发执行的程序的协调也是一个重要和重复提及的主题。

缺乏全局时钟：在程序需要协作时，通过交换消息来协调它们的动作。密切的协作通常取决于对程序动作发生的时间的共识。但是，事实证明，网络上的计算机与时钟同步所达到的准确性是有限的，即没有一个正确时间的全局概念。这是由于通信仅仅是通过网络发送消息这个事实带来的直接结果。定时问题和它们的解决方案将在第11章描述。

故障独立性：所有的计算机系统都可能出故障，一般由系统设计者负责为可能的故障设计结果。分布式系统可能以新的方式出现故障。网络故障导致网上互连的计算机的隔离，但这并不意味着它们停止运行，事实上，计算机上的程序不能够检测到网络是出现故障还是网络运行得比通常慢。类似的，计算机的故障或系统中程序的异常终止（崩溃），并不能马上被与它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后果将是本书的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共享资源的期望。“资源”一词是相当抽象的，但它很

2 好地描述了能在连网的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、数据库和所有的数据对象）。它包括来自数字摄像机的视频流和移动电话呼叫所表示的音频连接。

本章主要论述分布式系统的本质，以及成功部署分布式系统所面临的挑战，1.2节展示了分布式系统的一些重要的例子，以及构造系统所需的组件和这些组件的作用。1.3节描述了在万维网环境中资源共享系统的设计，1.4节则阐述了分布式系统设计者所要面对的重要挑战：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 分布式系统的实例

本节选用的实例基于大家熟悉和广泛使用的计算机网络，包括因特网、企业内部网和新兴的基于移动设备的网络技术。我们举这些例子主要是说明由计算机网络支持的服务和应用具有很广的范围，然后从这些系统开始讨论支持系统实现的技术问题。

1.2.1 因特网

因特网是一个巨大的由多种类型计算机网络互连的集合。图1-1摘取了因特网的部分典型组成。因特网上的计算机程序通过传递消息进行交互，采用了一种公共的通信手段。因特网通信机制（因特网协议）的设计和构造是一项重大的技术成果，它使得一个在某处运行的程序能给另一个地方的程序发送消息。

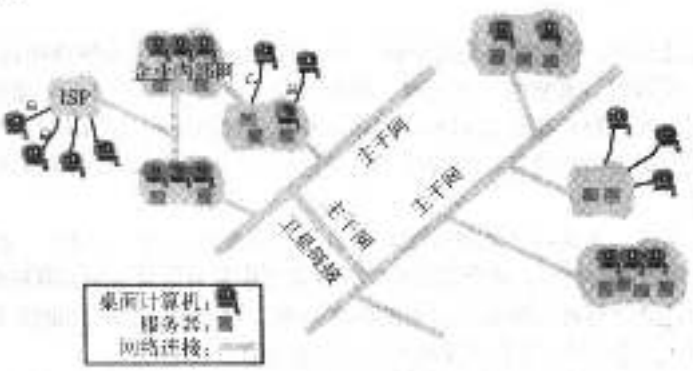


图1-1 因特网的典型部分

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、电子邮件和文件传送等服务。（有时，不确切地说，Web等同于因特网。）服务集是开放的，它能够通过服务器计算机和新的服务的增加而被扩展。图1-1还展示了许多企业内部网——由公司和其他组织操作的子网。因特网服务提供商（ISP）是给个体用户和小型组织提供调制解调器链接和其他类型连接的公司，使他们能获得因特网上的服务，同时提供诸如电子邮件和Web主机等本地服务。企业内部网通过主干网实现互相链接，主干网是具有高传送能力的网络链接，通常采用卫星连接、光缆和其他高宽带线路。

因特网能够提供多媒体服务，使用户能获得包括音乐、广播和TV频道在内的音频和视频数据，并召开电话和视频会议。由于目前因特网没有提供为单个数据流预留网络容量所必需的设施，因此它处理多媒体数据这类特殊通信需求的能力还很有限。第17章将讨论分布式多媒体系统的需求。

因特网的实现和它支持的服务已经解决了分布式系统的许多问题（包括在1.4节中定义的大多数问题）。本书将着重阐述这些解决方案，并在适当的时候说明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，它是独立管理的，具有一个可被配置来执行本地安全策略的边界。图1-2给出了一个典型的企业内部网。它由几个通过主干网连接的局域网（LAN）组成。每个企业内部网的网络配置都由管理企业内部网的组织负责，这种管理的范围差异很广，可以从单个场地的LAN到（可能分布在不同的国家）属于同一个公司或组织的若干部门的若干LAN。

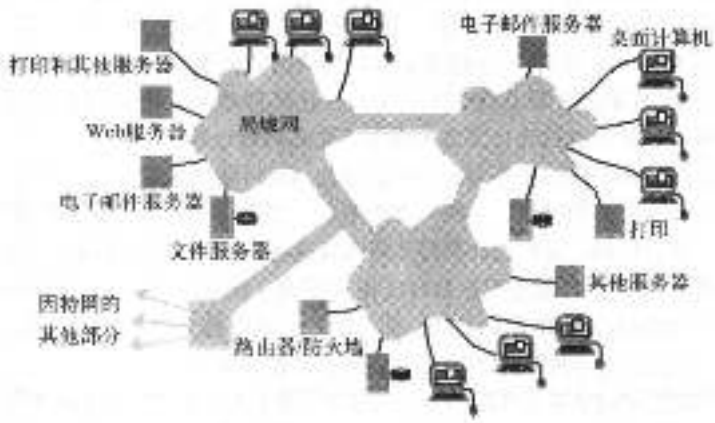


图1-2 典型的企业内部网

企业内部网通过路由器连接到因特网，因此企业内部网内的用户能使用因特网上的服务（如Web或电子邮件）。企业内部网也允许其他企业内部网的用户访问它提供的服务。许多组织需要保护他们自己的服务以免其他地方可能有恶意的用户未经授权便使用。例如，公司不希望保密的信息被竞争对手获取；医院不希望病人病历被曝光。公司也希望自己免受病毒入侵这样的有害程序影响，以免他人对企业内部网内的计算机的攻击并摧毁有用的数据。

防火墙的作用是通过防止未授权消息进出网络来保护企业内部网。防火墙是通过过滤进出的消息来实现的，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所保护的企业内部网。

一些组织根本不希望将他们的内部网连接到因特网上。例如，警察机关和其他安全法律执行机构可能至少有几个内部网和外部世界隔离；一些军事组织在战争时期会将它们的内部网与因特网断连。但即使是这样的组织，也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。这些组织通常采用的解决方案是像上面描述的那样操作企业内部网，但不与因特网相连。这样一个企业内部网可以没有防火墙，或者，从另一个角度来看，这有可能是最有效的防火墙——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，会出现下列主要问题：

- 需要文件服务以使用户能共享数据，文件服务的设计将在第8章讨论。
- 防火墙试图阻止对服务的合法访问——当需要在企业内部网和外部用户之间共享资源时，防火墙必须增加细粒度的安全机制。这部分内容将在第7章讨论。
- 用于软件安装和支持的花销是一个重要的问题。通过使用诸如网络计算机和瘦客户这样的系统体系结构，能减少这些开销。这方面的内容参见第2章。

1.2.3 移动计算和无处不在计算

设备小型化和无线网络方面的技术进步已经逐步使得小型和便携式计算设备集成到分布式系统中。这些设备包括：

- 笔记本电脑。
- 手持设备，包括个人数字助理（PDA）、移动电话、传呼机、摄像机和数码相机。
- 可穿戴设备，如具有类似PDA功能的智能手表。
- 嵌入在家电（如洗衣机、高保真音响系统、汽车和冰箱）中的设备。

这些设备大多数具有可携带性，再加上它们可以在不同地方方便地连接到网络的能力，使得移动计算成为可能。移动计算，也叫游牧计算[Kleinrock 1997]，是指用户在移动或参观某处（而不是在通常环境下）执行计算任务的性能。在移动计算中，远离其本地的企业内部网（指工作环境或其住处的企业内部网）的用户也能通过他们携带的设备访问资源。他们能继续访问因特网，继续访问在他们本地内部企业网上的资源。为用户在其移动时方便地利用周围资源（如打印机）的设备也在不断增加。后者也称为位置感知或上下文感知的计算。

无处不在计算[Weiser 1993]是指对在用户的物理环境（包括家庭、办公室和其他地方）中存在的多个小型便宜的计算设备的利用。术语"无处不在"意指小型计算设备最终将在日常不会引人注意的物品中普及。也就是说，它们的计算行为将无痕迹地紧密捆绑到这些日常物品的物理功能上。

各处的计算机只有在它们能相互通信时才变得有用。例如，如果用户能通过一个"通用远程控制"设备控制家里的洗衣机和音响系统，那么用户会觉得很方便。而洗衣机在完成洗衣后能通过一个智能报警器或智能手表呼叫用户，也会让人觉得很方便。

无处不在计算和移动计算有交叉的地方，因为从原理上说，移动用户能利用遍布各处的计算机。但一般而言，它们是不同的。无处不在计算能让呆在家里或医院这样单一的环境中的用户受益。类似地，即使移动计算只涉及常见的分立的计算机和设备（如笔记本电脑和打印机），它还是有优势的。

图1-3显示了一个正在访问一个组织的用户，该图显示出用户本地的内部网和用户正在访问的内部网。两个企业内部网通过因特网相连。

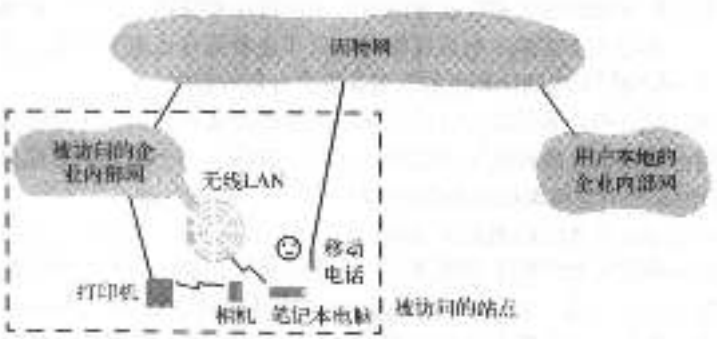


图1-3 分布式系统中的便携式设备和手持设备

用户可以使用三种无线连接。笔记本电脑可以连接到被访问组织的无线LAN。无线LAN覆盖方圆几百米的范围（即建筑物的一层）。它通过网关连接到被访问组织企业内部网。用户还有一部连到因特网的移动电话。在电话中这些信息按页显示在电话的显示屏上。最后，用户携带一台数码相机，它能通过一个个人局域无线网络（其覆盖范围大约为10m）与打印机这样的设备通信。

利用适当的系统基础设施，用户能用他们携带的设备完成一些简单的任务。当用户连接到被访问的站点时，他能通过移动电话从Web服务器上取得最新的股票价格。在与访问企业开会时，通过把数码相机的照片直接发送到会议室的一台可用的打印机上，用户就能展示最近的照片。这仅仅要求相机和打印机之间具有无线连接，原则上，用户可以利用无线LAN或是有线的以太网链接从笔记本电脑上把文件发送到同一台打印机。

移动计算和无处不在计算是一个热门的研究领域。第16章将继续讨论这两个主题。

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。大家通常共享硬件资源（如打印机）、数据资源（如文件）和具有特定功能的资源（如搜索引擎）。

从硬件资源的观点看，大家共享设备（如打印机和磁盘）可以减少花费，但共享与用户应用、日常工作和社会活动有关的更高层的资源对用户的意义更大。例如，用户关心以共享数据库或Web页面方式出现的共享数据，而不是实现上述服务的硬盘和处理器。类似地，用户关心诸如搜索引擎或货币转换器的共享资源，而不关心提供这些服务的服务器。 7

实际上，资源共享的模式随其工作范围和与用户工作的密切程度的不同而不同。一种极端是，Web上的搜索引擎给全世界的用户提供工具，而用户之间并不需要直接接触；另一种极端是，在*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小组中共享诸如文档之类的资源。用户在地理上的分布以及用户之间进行共享的模式决定了系统必须提供协调用户动作的机制。

我们使用术语*服务*表示计算机系统中管理相关资源并提供功能给用户和应用的一个单独的部分。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。仅仅通过服务提供的操作可以实现对服务的访问。例如，一个文件服务提供对文件的read、write和delete操作。

服务将资源访问限制为一组定义良好的操作，这属于标准的软件工程实践，同时它也反映出分布式系统的物理组织。分布式系统的资源是物理地封装在计算机内的，其他计算机只能通过通信才能访问。为了实现有效的共享，每个资源必须由一个程序管理，这个程序提供通信接口使得对资源进行可靠和一致的访问和更新。

大多数读者很熟悉术语*服务器*，它指的是在连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上正在运行的程序请求，执行一个服务并适当地响应。发出请求的进程称为*客户*。请求以消息的形式从客户发送到服务器，应答以消息的形式从服务器发送到客户。当客户发送一个要执行的操作请求，就称客户*调用那个服务器上的操作*。客户和服务器之间的完整交互，即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*。

一个进程可能既是客户又是服务器，因为服务器有时调用其他服务器上的操作。术语“客户”和“服务器”仅仅是针对在一个请求中扮演的角色而言。就它们扮演的角色不同这点而言，客户是主动的，服务器是被动的；服务器是连续运行的，而客户所持续的时间只是客户所属的那部分应用程序持续的时间。

注意，默认情况下，术语“客户”和“服务器”指的是*进程*而不是运行客户或服务器的计算机，虽然在日常用法中这些术语也指计算机。另一个不同（见第5章）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，并由客户对象访问，这时，称一个*客户对象*调用了一个*服务器对象*上的方法。

许多（但不是所有的）分布式系统可以完全用客户和服务器交互的形式来构造，万维网、电子邮件和连网的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。

一个正在执行的Web浏览器是一个客户的例子。Web浏览器与Web服务器通信，从服务器上请求Web页面。下面将详细讨论Web。 8

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。通过常用的Web浏览器，用户可以检索和查看多种类型的文档、收听音频文件、观看视频文件、与无数服务进行交互。

Web是1989年在瑞士的欧洲原子能研究中心（CERN）诞生的，作为通过因特网连接的物理学家之间交换文档用的工具[Berners-Lee 1999]。Web的一个关键特征是它在所存储的文档中提供了超文本结构，超文本结构反映了用户对知识组织的要求。这意味着文档包含链接（或超链接），链接指向其他存储在Web上的文档和资源。

对Web用户来说，当他遇到文档中的一幅图像或一段文字时，它很可能伴有到相关文档和其他资源的链接，链接的结构可以简单，也可以复杂，可加入的资源集是无限的，即链接的Web确实是世界范围的。Bush[1945]在五十年前就设想出了超文本结构，因特网的发展使得这个想法能在世界范围内得到证实。

Web是一个开放的系统，它可以被扩展，并且在不妨碍已有功能的前提下用新的方法实现扩展（见1.4.2节）。

首先，它的操作是基于被自由发布和广泛实现的通信标准和文档标准的。例如，浏览器的类型是多种多样的。在多数情况下，每种浏览器可以在多个平台上实现；有多种Web服务器实现，一种构造的浏览器能从不同构造的服务器中检索资源。所以，用户能访问大多数设备（从移动电话到桌面计算机）上的浏览器。

其次，相对于能在其上发布和共享的“资源”的类型而言，Web是开放的。在Web上，最简单的资源是一个Web页面或其他能保存在文件中并提交给用户的内容，如程序文件、介质文件和PostScript和PDF格式的文件。如果有人新发明了一种图像存储格式，那么这种格式的图像能马上在Web上发布，用户需要一种查看这种新格式图像的工具，而浏览器以“帮助者”应用和“插件程序”的形式来支持新的内容显示功能。

Web的发展已超越这些简单的数据资源而开始包含服务，如电子化的商品购买。Web一直在发展，但其基本的体系结构没有改变。Web基于以下三个主要的标准技术组件：

- 超文本标记语言（HTML）是页面在Web浏览器上显示时指定其内容和布局的语言。
- 统一资源定位器（URL）用于识别保存成Web一部分的文档和其他资源。第9章将讨论有关的Web标识符的其他术语。
- 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。浏览器和其他客户可利用标准交互规则从Web服务器上获取文档和其他资源。图1-4给出了一些Web服务器和向它们发送请求的浏览器。用户可以定位和管理位于因特网上任何地方的他们自己的Web服务器，这是一个很重要的特征。

9

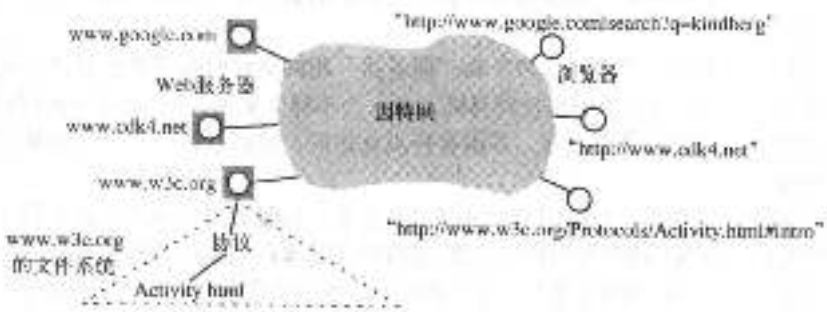


图1-4 Web服务器和Web浏览器举例

接下来我们依次讨论这些组件，并解释用户获取Web页面并单击页面上的链接时，浏览器和Web服务器的操作。

HTML 超文本标记语言将这些内容[www.w3.org II]用于指定组成Web页面内容的文本和图像，指定它们以何种布局方式和组织方式将这些内容显示给用户。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。

用户可使用标准的文本编辑器手写生成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文本：

```
<IMG SRC = "http://www.cdk4.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.cdk4.net/WebExample/moon.html "> Moon </A>.                  4
<P>                                                                                5
```

这段HTML文本保存在一个Web服务器可以访问的文件（例如earth.html文件）中。浏览器从Web服务器（本例中是一个位于名为www.cdk4.net的计算机上的服务器）中检索这个文件的内容，浏览器读取从服务器返回的内容后，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。只能由浏览器（不是服务器）解释HTML文本，但是服务器确实通知了浏览器它所返回的内容的类型，用于区分html文件和其他文件（如PostScript文件）。服务器能从文件的扩展名“.html”中推断出内容类型。

注意，HTML的指令（即标记）放在尖括号里，如〈P〉。例子中的第一行确定了一个包含图片显示的文件，图片的URL是http://www.cdk4.net/WebExample/Images/earth.jpg。第二行的指令表示开始新段落，第三行和第四行包含要在Web页面上以标准的段落格式显示的文本，第五行的指令表示该段落结束。 10

其中，第四行指定了Web页面上的一个链接。它包含词“Moon”，该词位于两个匹配的HTML标记〈A HREF…〉和〈/A〉中间。这些标记之间的文本在Web页面上显示时是以链接的形式出现的。大多数浏览器在默认情况下给链接的文本加下划线，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和包含在<A HREF…>标记中的URL之间的关联，在这个例子中是：

http://www.cdk4.net/WebExample/moon.html

当用户单击文本时，浏览器获取由相应URL识别的资源，并将它显示给用户。在这个例子中，资源是一个HTML文件，它指定了关于月亮的一个Web页面。

URL 统一资源定位器[www.w3.org III]的作用是识别资源。在Web体系结构文档中使用的术语是统一资源标识符（URI），在不引起混淆的前提下，本书使用更为人们所熟悉的术语URL。浏览器检查URL以便从Web服务器上访问相应的资源。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，由浏览器查找相应的URL；或当浏览器去取一个Web页面里的内嵌资源（如一个图像）时，由浏览器查找相应的URL。

按绝对完整的格式，每一个URL有两个不可或缺的组成部分：

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能识别各种资源。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标识一个用文件传送协议（FTP）获取而不是用更常用的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“mid”（用于标识一个邮件消息）。

从Web可访问（利用URL中的模式指示器）的资源类型的角度来说，它是开放的。如果有人发明了一种新的有用的“widget”资源（可能用它专有的寻址方案定位widget，用它专有的协议访问widget）那么大家就能使用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插件实现。

HTTP URL是使用最广泛的定位器，它利用标准的HTTP协议访问资源。HTTP URL有两项主要的工作：识别出哪一个Web服务器维护资源；识别出该服务器上的哪些资源是被请求的。图1-4显示了三个浏览器发出请求，而被请求的资源由三个Web服务器管理。最上面的浏览器向一个搜索引擎发出查询，第二个浏览器请求另一个Web站点的默认页。最下面的浏览器请求一个指定了
11 全名（包括了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路径名识别。

通常，HTTP URL具有下列格式：

http://服务器名[:端口][/路径名][?查询][#片段]

其中方括号中的项是可选的。一个HTTP URL全名总是以“http://”开始，后跟一个服务器名，该服务器名表示成一个DNS名（参见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（参见第4章）——默认值是80。端口号后面是可选的服务器资源的路径名，如果没有这一项，那么请求的是服务器的默认页。最后，URL以一个可选的查询成分——（例如，当一个用户提交诸如搜索引擎查询页这样的表单中的项）或片段标识符（它标识资源的成分）结束。

下面分析图1-4的URL:

http://www.cdk4.net

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

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

上述URL可分解成如下部分：

服务器DNS名	路径名	查询	片段
www.cdk4.net	（默认）	（无）	（无）
www.w3.org	Protocols/Activity.html	（无）	intro
www.google.com	search	q=kindberg	（无）

第一个URL指定了由www.cdk4.net提供的默认页，第二个URL指定了与www.w3.org服务器相关的路径名为Protocols/Activity.html的HTML文件的片段。片段标识符（由URL中#后面的字符指定）是intro，浏览器在下载整个文件后将在HTML文本中查找该片段标识符。第三个URL给搜索引擎指定一个查询。路径指定了一个名为“search”的程序，“？”字符后面的串是作为该程序的参数查询字符串。在考虑更高级的特征时，我们将详细讨论识别程序资源的URL。

发布资源：虽然Web有一个用于从URL检索资源的清晰定义的模型，但是在Web上发布资源的方法仍然依赖于Web服务器的实现。为了在Web上发布资源，最简单的方法是在Web服务器能访问的目录下放上相应的文件。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，才能构造像http://S/P这样的URL。用户可以把这个URL放在已有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的路径名约定。例如，按约定以~joe开头的路径名是在用户joe主目录的
12 子目录public_html下。类似地，通常不是以文件名结束而是以目录名结束的路径名指的是该目录中的index.html文件。

Huang等人[2000]提供了一个模型，该模型可用于以最少的人工干预将内容插入Web。它在用户需要从多种设备（如照相机）提取内容发布到Web页面时特别有用。

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章将详细讨论HTTP，这里先概述它的主要特征（我们的讨论将限制在对文件资源的检索上）：

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给被请求资源的URL所在的服务器。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，它返回一个出错应答，例如大家熟悉的“404 Not Found”消息。

内容类型：浏览器没必要能够处理每一种内容类型。当浏览器发出一个请求时，其中包括浏览器擅长处理的内容类型的清单——例如，原则上浏览器能够以“GIF”格式而不是“JPEG”格式显示图像。服务器在将内容返回给浏览器时应当能考虑到这些方面。服务器在应答消息中包含内容类型，以便浏览器能知道如何处理服务器返回的内容。表示内容类型的串称为MIME类型，RFC 1521[Freed and Borenstein 1996]已对其做了标准化。例如，如果内容是“text/html”类型，那么浏览器将把文本解释成HTML并加以显示；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帮助者应用程序来将数据解压缩。浏览器对指定的内容类型所采取的动作是可配置的，读者可以检查自己浏览器的设置。

一次请求一个资源：客户的每个HTTP请求只指定一个资源。如果Web页面包含9个图像，那么浏览器总共要发出10个单独请求才能获得该页完整的内容。通常浏览器可同时发出几个请求，以减少对用户的整体延迟。

简单的访问控制：默认情况下，通过网络与Web服务器相连的用户能访问所有已发布的资源。如果用户希望限制对一个资源的访问，那么他要配置服务器，给发出请求的客户回发一个“质询”。对应的用户要证明他们有权限访问该资源（如通过输入口令）。

动态页面 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，然而大多数用户对Web的经验来自与服务交互而不是检索数据。例如，当用户在一个网上商店中购买一个物品时，用户经常要填写一个Web*表单*，写明他们的个人信息或详细说明他们要购买什么商品。Web表单是包含用户指令和诸如文本字段、复选框等窗口输入部件的Web页面。当用户提交表单（通常通过按下按钮或“回车”键）后，浏览器就发送一个HTTP请求到Web服务器，请求中包含用户已经输入的值。 13

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或它的第一个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=kinderg”的请求就表示调用“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文本作为输出，用户将看见若干包含单词“kindberg”的页面。（读者可以在自己常用的搜索引擎上输入一个查询，注意查看在返回结果时浏览器显示的URL。）服务器返回“search”程序生成的HTML文本，就好像是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间的差别对浏览器是透明的。

在Web服务器上运行的为客户生成内容的程序通常称为公共网关接口（CGI）程序。一个CGI程序可以具有任何应用特定的功能，只要它能分析客户提供给它的参数，产生所要求类型的内容（通常是HTML文本），程序在处理请求时会经常查询或更新数据库。

下载的代码：CGI程序在服务器上运行。有时Web服务的设计者需要一些与服务相关的代码，以便在用户计算机的浏览器内部运行。例如，用Javascript[www.netscape.com]编写的代码下载时通常带有Web表单，以便提供比HTML标准窗口部件质量更好的用户交互。用Javascript增强的页面对无效项能给用户及时的反馈（而不是在服务器端检查用户输入值，这种方法要花费较长的时间）。Javascript也能用于更新Web页面的部分内容而不必取得该页面的全新版本并重新显示。

Javascript在功能上有一定的限制。相比之下，applet是浏览器在取得相应Web页面时能自动下载并运行的应用程序。applet能利用Java[java.sun.com, Flanagan 2002]语言设施来访问网络，提供定制的用户界面，例如，“聊天”应用程序有时用applet实现，它在用户的浏览器上运行，同时还

要有一个服务器程序。applet把用户的文本发送给服务器，服务器再给所有的applet分发该文本，用于给用户显示。2.2.3节将详细讨论applet。

Web服务 到目前为止，我们主要从用户操作浏览器的角度讨论Web，但除浏览器之外的程序也会是Web的客户；通过程序访问Web资源确实也是常事。但HTML和HTTP标准在一定程度上缺乏与程序的互操作性。

首先，交换Web上的多种类型的结构化数据的需求在上升，但HTML的功能是有限制的，因为它不能扩展为信息浏览之外的应用。HTML具有一套静态结构（例如段落），并且其数据显示给用
14 户的方式也是受限的。可扩展标记语言（XML）（见4.3.3节）是一种以标准的、结构化的、特定于应用的格式表示数据的方式。例如，用XML描述设备的功能和用户的个人信息。XML是描述数据的元语言，它使得数据在应用之间移植成为可能。

其次，HTTP没有提供结构，用于指定服务特定的可调用Web资源的操作、操作的参数和出错应答。例如，在amazon.com网上商店，Web服务操作包括订购图书和检查订单当前状态。灵活性的另一面是在软件操作方式上缺乏健壮性。第19章将深入研究Web服务框架，在Web服务框架下，Web服务的设计者能准确地告诉程序员客户应当如何访问它们。

对Web的讨论 Web之所以取得巨大成功，是因为许多个人和机构能比较容易地发布资源，它的超文本结构适合组织多种类型的信息，而且Web系统体系结构具有开放性。Web的体系结构所基于的标准很简单，而且它们早已被广泛地发布。它们使得许多新的资源类型和服务可以集成在一起。

Web成功的背后也存在一些设计问题。首先，它的超文本模型在某些方面有所欠缺。如果删除或移动了一个资源，那么就会存在对资源的所谓的“悬空”链接，会使用户请求落空。此外，还存在用户“在超空间迷失”这个常见的问题。用户经常发现自己处于混乱状态下，跟随许多无关的链接打开完全不同的页面，在有些情况下其可靠性值得怀疑。

在Web上查找信息的另一种方法是使用搜索引擎，但这种方法在满足用户真正需求方面是相当不完美的。要解决这个问题，Resource Description Frame work [www.w3.org V]中介绍过，一种方法是生成标准的表达事物元数据的词汇、语法和语义，并将元数据封装在相应的Web资源中供程序访问。除了查找Web页面中出现的单词外，从原理上，程序可以完成针对元数据的搜索，然后，根据语义匹配编译相关的链接的列表。总而言之，由互连的元数据资源组成的Web就是语义Web。

作为系统体系结构，Web面临规模的问题。受欢迎的Web服务器会在一秒中有很多点击量，结果导致对用户的回答变慢。第2章将描述在浏览器和代理服务器中使用缓存来加快应答，以及将服务器负载分配到集群计算机上。但是，Web的客户－服务器结构意味着它没有有效的手段让用户保留最新版的页面。用户不得不点击浏览器的“刷新”按钮确保它们获得的是最新的信息，浏览器被强制与服务器通信，检查资源的本地拷贝是否还是有效的。

最后，一个Web页面不总是一个让用户十分满意的界面。HTML定义的界面部件是有限的，设
15 计者经常在Web页面上加入applet或图像使得页面外观和功能更能被用户接受，但这也导致下载时间增加了。

1.4 挑战

1.2节的例子试图说明分布式系统的范围，并提出在设计中出现的问题。虽然分布式系统已经无处不在，但它们的设计是相当简单的，还存在很大的空间来开发功能更加强大的服务和应用。虽然本节讨论的许多挑战已经得到解决，但将来的设计者还需要了解它们，并仔细地思考它们。

1.4.1 异构性

因特网使得用户能在大量异构计算机和网络上访问服务和运行应用程序。下面这些均存在异构性（即存在多样性和差别）：

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

虽然因特网由多种网络组成（见图1-1），但因为所有连接到因特网的计算机都使用因特网协议来相互通信，所以这些不同网络的区别被屏蔽了。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该网络上实现因特网协议。第3章将解释因特网协议如何在多种不同网络上实现。

整型等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表示方法。如果要在不同硬件上运行的两个程序之间交换消息，那么就要考虑它们在表示上的不同。

虽然因特网上所有计算机的操作系统均要包含因特网协议的实现，但不同的操作系统对协议的应用编程接口可以是不同的。例如，UNIX中消息交换的调用与Windows NT中的调用是不一样的。

不同的编程语言用不同的方式表示字符和数据结构（如数组和记录）。如果想让用不同语言编写的程序能够相互通信，那么必须解决这些差异。

不同开发者只有使用公共标准，他们编写的程序才能相互通信。例如，网络通信和消息中的基本数据项和数据结构的表示均要使用公共标准。所以，要制订和采用像因特网协议一样的公共标准。

16

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，同时屏蔽了底层网络、硬件、操作系统和编程语言的异构性。第4、5和20章描述的公共对象请求代理（CORBA）就是一个中间件。有些中间件，如Java 远程方法调用（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差异，但所有的中间件要解决操作系统和硬件的不同，如何做到这一点将是第4章讨论的主题。

除了解决异构性的问题外，中间件为服务器和分布式应用的程序员提供了一致的计算模型。这些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，它允许在一台计算机上运行的程序中的对象调用在另一台计算机上运行的某个程序中的一个对象的方法。它从实现上屏蔽了为了发送调用请求和应答，消息通过网络传递的事实。

异构性和移动代码 *术语移动代码*指能从一台计算机发送到另一台计算机，并在目的计算机上运行的代码，Java applet是一个例子。适合在一种计算机上运行的代码未必适合在另一种计算机上运行，因为可执行程序通常依赖于计算机的指令集和操作系统。例如，Windows/x86用户能够把可执行文件作为电子邮件的附件发送，但接收者不能在运行MacOS X 的Macintosh计算机或运行Linux的x86计算机上执行该文件。

*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一次Java虚拟机，然而，Java解决方案并不能应用到用其他语言编写的程序。

1.4.2 开放性

计算机系统的开放性是决定系统能否以不同的方式被扩展和重新实现的特征。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。

除非软件开放者能获得系统组件的关键软件接口的规约和文档，否则无法达到开放性。一句话，发布关键接口。这个过程类似接口的标准化，但它经常避开官方的标准化过程，官方的标准化过程通常繁琐且进度缓慢。

然而，发布接口仅是分布式系统增加和扩展服务的起点。设计者所面临的挑战是解决由不同
17 人构造的由许多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称为“征求意见文件”（即RFC）的文档，每个文档有一个编号。20世纪80年代早期发布了因特网通信协议的规约，并放入RFC中，中期发布了在因特网上运行的应用的规约，如文件传输规约、电子邮件规约和telnet规约。这种活动一直在继续，形成了因特网技术文档的基础。除了协议规约，该序列还包含讨论。读者可从[www.ietf.org]获得这些资料。最初的因特网通信协议的发布使得各种因特网系统和应用（包括Web）应运而生。RFC不是唯一的发布方式，例如，CORBA是通过一系列技术文档发布的，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之所以称为*开放的分布式系统*，主要是强调它们是可扩展的。它们可通过在网络中增加计算机实现在硬件级的扩展，通过引入新的服务、重新实现旧的服务在软件级的扩展，最终使得应用程序能共享资源。开放系统常被提到的另一个好处是它们与销售商无关。

开放的分布式系统的特征总结如下：

- 发布系统的关键接口是开放系统的特征。
- 开放的分布式系统是基于一致的通信机制和发布接口访问共享资源的。
- 开放的分布式系统能用不同销售商提供的异构硬件和软件构造，但如果想让系统正确工作，就要仔细测试和验证每个组件与发布的标准之间的一致性。

1.4.3 安全性

分布式系统维护和使用的众多信息资源对用户具有很高的内在价值，因此它们的安全相当重要。信息资源的安全性包括三个部分：机密性（防止泄漏给未授权的个人）、完整性（防止被改变或被破坏）、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，而且可以不考虑它们的位置，但安全风险与允许自由访问企业内部网内的所有资源相关。虽然防火墙能形成保护企业内部网的屏障，限制进出企业内部网的流量，但这不能确保企业内部网的用户恰当地使用资源，或恰当地使用因特网的资源，后一种资源不被防火墙保护。

在分布式系统中，客户发送访问由服务器管理的数据的请求，这涉及在网络上通过消息发送
18 信息。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）在电子商务和电子银行中，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。但安全性不只是涉及对消息的内容保密，它还涉及确切知道用户或代表用户发送消息的其他代理的身份。在第一个例子中，服务器要知道用户确实是一个医生；在第二个例子中，用户要确保他们正在交易的商店或银行的身份正确。这里，所面临的第二个挑战是正确地识别远程用户或其他代理的身份。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，第7章将讨论这个问题。

然而，下列两个安全方面所面临的挑战目前还没有圆满解决：

拒绝服务攻击：另一个安全问题是出于某些原因用户可能希望中断服务。可用下面的方法实现这个目的：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。

时不时地就会发生对几个众所周知的Web服务进行的拒绝服务攻击。现在通过在事件发生后抓获和惩罚犯罪者来解决这种攻击，但这不是解决这种问题的通用方法。以改善网络管理为根本的反击手段正在开发过程中，第3章会讲解这些问题。

移动代码的安全：移动代码需要小心处理。设想用户接收到一个作为电子邮件附件发送的可执行程序，那么运行该程序会带来的后果是不可预测的。例如，它可能看似显示了一幅有趣的画，但实际上它可能在访问本地资源，或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会在第7章中谈到。

1.4.4 可伸缩性

分布式系统可在不同的规模（从小型企业内部网到因特网）下有效且高效地运转。如果资源数量和用户数量激增，系统仍能保持其有效性，那么该系统就称为*可伸缩的*。因特网就是一个计算机数量和服务数量不断增长的分布式系统的例子。图1-5列出了到2003为止的24年间因特网上计算机数量的增加情况，图1-6则列出到2004年为止Web出现的10年间计算机和Web服务器数量的增长情况，见[zakon.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
2003年1月	171 638 297	35 424 956

图1-5 因特网上的计算机数量（带注册的IP地址）

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
2001年7月	125 888 197	31 299 592	25
2003年7月		42 298 371	

一个Web服务器可能部署在多个场地。

图1-6 因特网上的计算机和Web服务器数量

可伸缩分布式系统的设计面临下列挑战：

控制物理资源的开销：当对资源的需求增加时，应该可以花费合理的开销扩展系统以满足要求。例如，在企业内部网上文件被访问的频率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文件访问请求，那么必须能增加服务器数量以避免可能出现的性能瓶颈。通常，要使有n个用户的系统成为可伸缩的，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能支持40个用户。虽然这听起来好像是理所当然，但实际上并不容易达到，具体内容请参见第8章。

控制性能损失：如果数据集的大小与系统中的用户或资源数量成正比，设想一下这些数据的管理，例如，记录计算机的域名和对应的由域名系统持有的因特网地址的表，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使用线性结构的算法。但即使使用层次结构，数量的增加仍将导致一些性能上的损失，即访问有层次的结构化数据的时间是$O(\log n)$，其中n是数据集的大小。对一个可伸缩的系统，最大的性能损失莫过于此。

防止软件资源用尽：用做因特网地址（IP地址）的数字是缺乏伸缩性的一个例子。在20世纪70年代晚期，决定用32位作为因特网地址，但第3章将会提到，可用的因特网地址将会用尽。由于这个原因，新版的协议将使用128位的因特网地址，这就要求对许多软件组件进行修改。要对因特网的早期设计者公平一些，这个问题是没有正确答案的，因为很难预测一个系统若干年后的需求。而且，过度考虑将来的增长可能比面临问题（过长的因特网地址将占据额外的消息空间和计算机存储空间）时再做改变的效果更糟。

避免性能瓶颈：通常，算法应该是分散型，以避免性能瓶颈。我们用域名系统的前身来说明这一点，那时名字表被保存在一个主文件中，可被任何需要它的计算机下载。当因特网中只有几百个计算机时这是可以的，但随着用户数量的增加，这就变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，分散到因特网中的服务器上并采用本地管理的方式，从而解决了这个瓶颈（参见第3章和第9章）。

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页面会引起网络性能下降。我们将在第2章了解到缓存和复制可以用于提高频繁使用的资源的性能。

理想状态下，系统规模增加时系统和应用程序应该不需要随之改变，但这一点很难达到。规模问题是分布式系统开发中面临的主要问题，本书将深入地讨论已经成功应用的技术。它们包括复制数据的使用（见第15章）、缓存的相关技术（见第2、8章）、部署多服务器以处理常见的任务从而使几个类似的任务能并发地完成。

1.4.5 故障处理

计算机系统有时会出现故障。当硬件或软件发生故障时，程序可能会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章将讨论可能在分布式系统的进程和网络中发生的故障并对其分类。

分布式系统的故障是部分的，也就是说，有些组件运行正常而有些组件出了故障。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，校验和可用于检测消息或文件中出错的数据。第2章将解释该方法很难甚至不可能检测到其他一些故障（如因特网上一台远程服务器的崩溃）。这种方法面临的挑战是如何处理不能被检测但被怀疑的故障。

21

掩盖故障：有些被检测到的故障能被隐藏起来或降低它的严重程度。下面是隐藏故障的两个例子：

1）消息在不能到达时进行重传。

2）将文件数据写入两个磁盘，如果一个磁盘损坏，另一个磁盘的数据仍是正确的。

降低故障严重程度的例子是丢掉被损坏的消息（可以对它进行重传）。读者可能意识到，在最坏情况下，隐藏故障的技术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有可能发生故障，要在容纳了众多组件的大规模的网络上检测可能发生的故障，并将故障隐藏是不太实际的。服务器的客户能设计成容错的，这通常也涉及容忍错误的用户。例如，当Web浏览器不能与Web服务器联系时，它不会让用户一直等待它与服务器建立连接，而是通知用户这个问题，让用户自由选择是否尝试稍后再连接。容错的服务见下面关于冗余的讨论。

故障恢复：恢复涉及软件的设计，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，被更新的永久数据（文件和其他保存在永久存储介质中的资料）可能处在不一致的状态。恢复的讲解见第14章。

冗余：利用冗余组件，服务可以实现容错。考虑下面的例子：

1）在因特网的任何两个路由器之间，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，以保证在任何一个服务器出现故障后数据仍是可访问的。服务器应该能检测其他服务的错误，当检测到一个服务器上有错误时，客户就被重定向到剩下的服务器上。

设计有效的技术来保证服务器上数据的副本是最新的，而且不过度地损失网络的性能，这是一个挑战。具体方法将在第15章讨论。

面对硬件故障，分布式系统具有高可用性。系统的可用性是系统可用时间的比例的度量。当分布式系统中的一个组件出现故障时，仅仅是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，用户可以将工作转移到另一台计算机上，并且服务器进程能在另一台计算机上启动。 22

1.4.6 并发性

在分布式系统中，服务和应用均提供可被客户共享的资源。因此，可能有几个客户同时试图访问一个共享的资源的情况。例如，在接近拍卖最终期限时，记录拍卖竞价数据结构可能被非常频繁地访问。

管理共享资源的进程一次接受一个客户请求，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请求。为了详细说明这一点，假设每个资源被封装成一个对象，在并发线程中执行调用。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竞标是“Smith:$122”和“Jones:$111”，那么相应的操作在没有任何控制时可能是交错进行的，它们可能保存成“Smith:$111”和“Jones:$122”。

这个例子是为了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，持有不希望用于分布式系统的对象实现的程序员必须做一些事情，使得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，它的操作必须在数据保持一致的基础上同步。这可通过标准的技术（如大多数操作系统使用的信号量）来实现。这个主题及其在分布式共享对象方面的扩展见第6章和第13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离性，使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]确定了八种透明性。下面将解释最初的ANSA定义，并用范围更广的移动透明性替换迁移透明性：

访问透明性：用相同的操作访问本地资源和远程资源。

位置透明性：不需要知道资源的物理或网络位置（例如，哪个建筑物或IP地址）就能够访问它们。

并发透明性：几个进程能并发地使用共享资源进行操作且互不干扰。 23

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知道副本的相关信息。

故障透明性：屏蔽错误，不论是硬件组件故障还是软件组件故障，用户和应用程序都能够完成它们的任务。

移动透明性：资源和客户能够在系统内移动而不会影响用户或程序的操作。

性能透明性：当负载变化时，系统能被重新配置以提高性能。

伸缩透明性：系统和应用能够进行扩展而不改变系统结构或应用算法。

最重要的两个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。有时它们统一称为网络透明性。

为了说明访问透明性，我们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异地，图形用户界面应该是一样的。另一个例子是API文件，它使用相同的操作访问本地和远程文件（见第8章）。如果一个分布式系统不允许访问远程计算机上的文件，除非用户利用ftp程序进行访问，否则认为这个系统就缺乏访问透明性。

Web资源名或URL具有位置透明性，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Web页面不能移动到另一个域中新的工作位置，因为其他页面上的所有链接仍将指向原来的页面。

像URL（包括计算机域名）这样的标识符妨碍了复制透明性。虽然DNS允许一个域名指向几台计算机，但它在查找名字时只选其中的一台计算机。因为复制方案通常要能够访问网络中所有参与其中的计算机，所以应该根据名字访问DNS条目中的每台计算机。

为了说明网络透明性，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。注意，虽然邮件程序将用户名接收为本地用户，但它们还是附加了本地域名。给这样的用户发送邮件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于接收者的位置。因此，因特网中的电子邮件支持位置透明性和访问透明性（即网络透明性）。

故障透明性也可以用电子邮件的例子来说明，即使服务器或通信链接出现故障，最终邮件还是能被发送。此时故障被屏蔽，因为邮件被一直重发直到它被成功传递到目的地址，即使这个过
24 程花费几天时间。中间件通常将网络和进程的故障转换成程序级的异常（详细解释参见第5章）。

为了说明移动透明性，再举一个移动电话的例子。假设打电话者和接电话者都在一个国家的不同地方乘火车旅行，他们从一个环境（蜂窝）移到另一个环境。我们将打电话者的电话作为客户，接电话者的电话作为一个资源。两个使用电话的用户并没有意识到电话（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，并使得这些资源能被匿名使用。例如，通常为完成任务，对相似的硬件资源的分配是可互换的——用于执行一个进程的处理器通常对用户隐藏身份并一直处于匿名状态。但正如1.2.3节指出的，情况并不总是如此。例如，旅行者每到一处便将他的笔记本电脑连接到本地网络，他应该能通过每处的不同的服务器使用本地服务（如发送邮件服务）。即使在一栋建筑物内，将要打印的文件发送到靠近用户的某台打印机上打印也是很常见的。

1.5 小结

分布式系统无处不在。因特网的出现使全世界用户无论走到哪里都能访问因特网上的服务。每个组织都管理一个企业内部网，并通过该企业内部网为本地用户提供本地服务和因特网服务，也为因特网上的其他用户提供服务。小型的分布式系统可由移动计算机和其他可连接到无线网络的小型计算设备构造。

资源共享是促进分布式系统的构建的主要因素。打印机、文件、Web页面或数据库记录这样的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。

分布式系统的构造面临着许多挑战：

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差异，中间件能处理其他的差异。

开放性：分布式系统应该是可扩展的——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：加密用于为共享资源提供充分的保护，在网络上用消息传送敏感信息时，通过加密的手段来保护敏感信息。服务拒绝攻击仍然是一个问题。

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，数据应该组织成层次化的结构以获得最好的访问时间。频繁访问的数据应能被复制。 25

故障处理：任一进程、计算机或网络都可能出现故障。因此每个组件需要清楚其所依赖的组件可能出现故障的方式，组件应当能适当地处理每个故障。

并发性：分布式系统中多个用户的存在是对资源产生并发请求的根源。每个资源必须被设计成在并发环境中是安全的。

透明性：此特性能够保证分布的某些方面对应用程序员不可见，这样应用程序员只需要关心特定应用的设计问题。例如，程序员不需要关心特定应用的位置或操作如何被其他组件访问等细节问题，或它是否被复制或迁移。甚至网络和进程故障也可以以异常的形式（但异常必须被处理）呈现给应用程序员。

练习

1.1 列出能被共享的五种类型的硬件资源和五种类型的数据或软件资源，并举出它们在实际的分布式系统中发生共享的例子。（第2页，第7～9页）[⊖]

1.2 在不参考外部时间源的情况下，通过本地网络连接的两台计算机的时钟如何同步？什么因素限制了你描述的过程的准确性？由因特网连接的大量的计算机的时钟是如何同步的？讨论该过程的准确性。（第2页）

1.3 一个用户随身携带能无线连网的PDA，来到一个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，如何得到关于本地服务和火车站环境的情况？要克服哪些技术问题？（第6页）

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？这些技术是否适合作为客户－服务器计算的基础？（第9页）

1.5 用万维网作例子说明资源共享、客户和服务器的概念。

万维网的资源和其他服务用URL命名，缩略语URL是指什么？给出能用URL命名的三种不同的Web资源例子。（第7页）

1.6 给出一个HTTP URL的例子。

列出HTTP URL的主要成分，叙述各个成分是如何表示的，并用例子说明每个成分。在什么程度上HTTP URL是位置透明的？（第7页） 26

1.7 用一种程序设计语言（例如C++）编写的一个服务器程序提供了一个BLOB对象的实现，该对象用于被不同语言编写（例如Java）的客户访问。客户计算机和服务器计算机可以有不同的硬件，但它们都连到企业内部网上。要使得一个客户对象调用服务器对象上的方法，请描述由于异构性的五个方面所带来的需要解决的问题。（第16页）

1.8 一个开放的分布式系统允许添加新的资源共享服务（如练习1.7中的BLOB对象）并被多种客户程序访问。讨论在这个例子中，开放性的需求与异构性的需求在什么范围内有不同。（第17页）

⊖ 本书习题中的页码为英文原书页码，即书中边栏标注的页码。

1.9 假设BLOB对象的操作分成两类——用于所有用户的公共操作和只用于某些命名用户的受保护操作。叙述为确保只有命名用户能使用保护操作所涉及的所有问题。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？ （第18页）

1.10 INFO服务管理一个可能非常大的资源集，用户能通过因特网利用关键字（一个字符串名字）访问这些资源。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性能出现瓶颈。 （第19页）

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三个主要的软件组件，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。 （第21页）

1.12 一个服务器进程维护一个共享的信息对象（如练习1.7中的BLOB对象）。讨论是否允许客户请求在服务器上并发执行。在它们并发执行时，给出可能在不同客户操作之间发生“干扰”的例子，说明如何避免这种干扰。 （第23页）

1.13 一个服务由几个服务器实现，试解释为什么资源能在它们之间传输。要实现客户的移动透明性，采用客户组播所有的请求到服务器组是否能获得满意的效果？ （第23页）

27

第2章 系统模型

分布式系统的体系结构模型涉及系统各个部分的位置和它们之间的关系。体系结构模型的例子包括客户－服务器模型和对等进程模型。客户－服务器模型可以通过以下方式进行修改：

- 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。
- 以便捷的方式增加或去除移动设备的需求。

基础模型涉及对所有体系结构模型中公共属性的一种更为形式化的描述。

在分布式系统中没有全局时间，所以不同计算机上的时钟不必给出相同的时间。进程间的所有通信是通过消息完成的。计算机网络上的消息通信会受延迟的影响，会遇到多种故障，对安全方面的攻击很脆弱。这些问题可通过下面三个模型解决：

- 交互模型处理分布式系统的性能问题并解决在分布式系统中设置时间限制的难题，例如解决消息传送问题。
- 故障模型试图给出进程和通信通道故障的一个精确的规约。它定义了可靠的通信和正确的进程。
- 安全模型讨论了对进程和通信通道的各种可能的威胁。它引入了安全通道的概念，安全通道能保证在上述威胁下通信的安全。

2.1 简介

打算要在实际环境中使用的系统应该在各种可能的环境下，面对各种困难和潜在的威胁（后面的“分布式系统的困难和威胁”部分将给出一些例子）时，保证其功能的正确性。第1章的讨论和例子表明不同类型的分布式系统共享重要的基本特性，也出现了公共的设计问题。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抽象、简化但一致的描述。

体系结构模型定义了系统中组件之间的交互方式和它们映射到计算机基础网络的方式。2.2节将介绍分布式系统软件的分层结构和决定组件位置和交互的主要体系结构模型。我们讨论客户－服务器模型的变体（包括它们是因为使用了移动代码）；接着考虑便于增加、去除移动设备的分布式系统特征；最后我们看一下分布式系统的一般设计需求。

2.3节介绍三个基础模型，用于为分布式系统设计者揭示若干关键问题。其目标是指出开发正确、可靠、安全地执行任务的分布式系统必须要解决的设计问题、困难和威胁。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威胁 下面是分布式系统设计者要面对的一些问题：

使用模式的多样性：系统的组件会承受各种工作负载，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线或连接不稳定，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：分布式系统必须能容纳异构的硬件、操作系统和网络。网络可能在性

能上有很大不同，如无线网的速度只达到局域网的几分之一，必须支持不同规模的系统，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、冲突的数据更新、多种涉及系统单个组件的软硬件故障模式。

30 外部威胁：包括对数据完整性、保密性的攻击以及服务拒绝。

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，整体目标是确保结构能满足现在和将来可能的需求。主要关心的是系统可靠性、可管理性、适应性和性价比。建筑物的体系结构设计有类似的方面。不仅要决定它的外观，而且它的总体结构和体系结构风格（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，即分布式系统的体系结构风格。我们将围绕第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首先将简化和抽象分布式系统单个组件的功能，然后考虑：

- 组件在计算机网络上的放置——为数据和负载分布定义有用的模式。
- 组件之间的关系——就是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来完成的。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。进程分类用于确定每个进程的责任，因此，有助于我们评估它们的负载，并决定每个进程出现故障可能造成的影响。分析的结果可用于指定进程的放置方式，使得进程的分布能满足目标系统的性能和可靠性要求。

一些更为动态的系统可作为客户-服务器模型的变体而创建：

- 将代码从一个进程移动到另一个进程的可能性使得一个进程将任务委托另一个进程也成为可能。例如，客户可以从服务器下载代码，并在本地运行，可以移动对象和访问对象的代码以减少访问延迟并将通信流量降至最低。
- 一些分布式系统具有无缝地增加或去除计算机和其他移动设备，允许它们发现可用的服务，为其他服务提供它们自己的服务的功能。

已有几个广泛使用的用于分布式系统的工作分配的模式，它们对目标系统的性能和有效性有巨大的影响。在计算机网络中组成分布式系统的进程的实际分配也受性能、可靠性、安全性和性价比方面的许多问题的影响。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个计算机里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请求和提供的服务。这个面向进程和面向服务的观点可以用服务层表达。图2-1表达了这个观点，详细的解释见第3章到第6章。服务层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，维护该服务中的资源在系统范围内的一致视图。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给任一发出请求的客户提供当前的时间。



图2-1 分布式系统中的软件和硬件服务层

作为与服务器交互的结果，客户调整它们的当前时间。

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称为分布式系统和应用的平台。这些底层给上层提供服务，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、Intel x86/Solaris、PowerPC/MacOS、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，给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，这些进程或对象相互交互，实现分布式应用的通信和资源共享支持。中间件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、共享数据对象在多个协作的计算机上的分布、放置和检索、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章将介绍组通信、第12章和15章还有详细的讨论、第5章描述事件通知。第10章描述在多个计算机上共享大量数据对象的方法。第15章讨论数据复制，多媒体系统见第17章。 31 ~ 32

早期的中间件实例有远程过程调用包（如Sun RPC，见第5章）和组通信系统（如Isis，见第15章）。目前，广泛使用面向对象中间件产品和标准，它们包括：

- CORBA
- Java RMI
- Web 服务
- Microsoft的分布式组件对象模型（DCOM）
- ISO/ITU-T的开放分布式处理参考模型（RM-ODP）

Java RMI、Web服务和CORBA的详细内容见第5章、第19章和第20章；DCOM和RM-ODP的介绍可参见Redmond[1997]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具有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20章将讨论一些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于可用的中间件提供的服务，以支持它们在通信和数据共享方面的要求。例如，适合客户－服务器模型的应用（如一个名字和地址的数据库）可以依赖只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成效，但在系统可依赖性的一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的问题。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）。但考虑下面这个问题：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提供了一定程度的错误检测和错误纠正，但它不能从严重的网络中断中恢复。另外，邮件传递服务增加了另一层容错，用来维护进度记录，如果原来的连接断开，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“端对端的争论”。可将他们的陈述表述为： 33

只依靠通信系统终点的应用的帮助和应用系统提供的知识，就能完整、可靠地实现一些与通信相关的功能。因此将这些功能做为通信系统的特征不总是明智的。（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。）

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。

争论的关键是分布式程序正确的行为在很多层面上依赖检查、错误校正机制和安全手段，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正确性。因此，可能在应用程序中重复同样的任务，降低了编程效率，更重要的是增加了不必要的复杂性并要执行冗余的计算。

本书没有进一步介绍“争论”的细节，强烈推荐读者阅读前面提到的那篇论文——那里有许多说明的实例。原文作者之一最近指出争论给因特网设计带来的实质性好处有被最近为满足当前应用需求而向网络服务专门化趋向所替代的危险[www.reed.com]。

2.2.2 系统体系结构

谈到分布式系统设计，人们可能最先想到系统组件（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。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，并将基于这些模型进行责任的分配。

在分布式系统中，具有明确责任的进程彼此交互，以执行有用的活动。本节我们的注意力将集中在进程的放置上，并按图2-2的方式给出计算机（黑框）中进程（椭圆）的部署。我们使用术语“调用”和“结果”来标注消息（或者用“请求”和“应答”标注）。

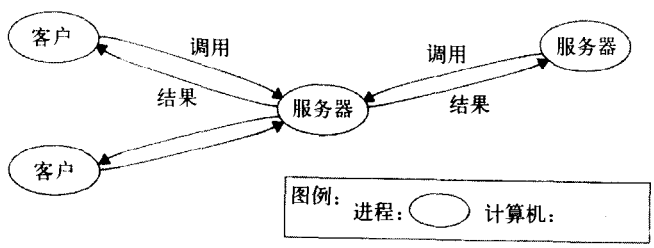


图2-2 客户调用单个服务器

图2-2和图2-3给出了两种主要的体系结构模型，下面将分别介绍这两种模型。

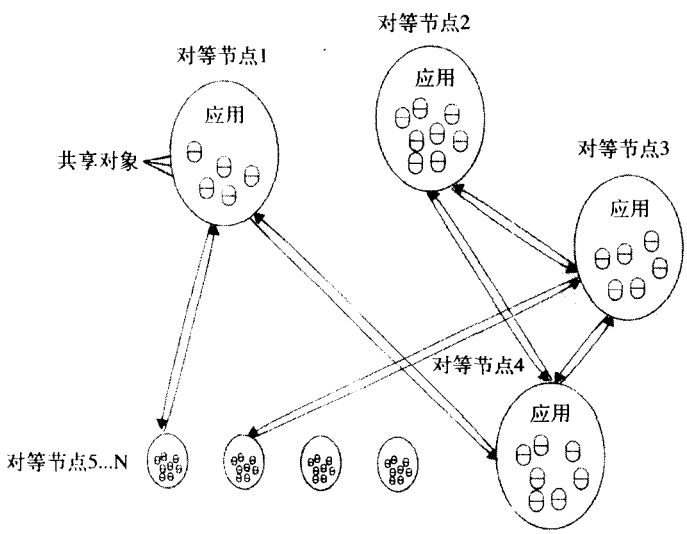


图2-3 基于对等体系结构的分布式应用

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。过去它是最重要的体系结构，现在仍被广泛使用。图2-2给出为了访问服务器管理的共享资源，客户进程与在单独主机上的一个服务器进程交互的简单结构。

如图2-2所示，一台服务器也可以是其他服务器的客户。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。这些信息汇总通过称为“Web蜘蛛”的程序形成，该程序在搜索引擎站点以后台方式运行，利用HTTP请求访问因特网上的Web服务器。因此，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Web蜘蛛程序。在这个例子中，服务器任务（对用户查询的回答）和Web蜘蛛的任务（向其他Web服务器发请求）是完全独立的，很少需要同步它们，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，另一些进程运行Web蜘蛛程序。练习2.4将请读者考虑这种类型的并发搜索引擎会出现的同步问题。

对等体系结构 在这种体系结构中，一项任务或活动涉及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或运行它们的计算机。虽然客户－服务器模型为数据和其他资源的共享提供了一个直接和相对简单的方法，但客户－服务器模型的伸缩性比较差。将一个服务放在单个进程中意味着集中化地提供服务和管理，它的伸缩性不会超过提供服务的计算机的能力和该计算机所在网络连接的带宽。

下一节将描述客户－服务器体系结构的几个变体，它们的出现就是为了应对上述问题，但它们都没有解决基本问题——如何将共享资源进行更广泛地分布，以便将访问资源带来的计算和通信负载分散到大量的计算机和网络链接中。

今天计算机具有的硬件容量和操作系统功能已经超过了昨天的服务器，而且大多数计算机配备有随时可用的宽带网络连接。对等体系结构的目的是挖掘大量参与计算机中的资源（数据和硬件）来完成某个给定的任务或活动。对等应用和对等系统已经被成功地构造出来，使得无数计算机能访问它们共同存储和管理的数据及其他资源。最早和最广为人知的系统是共享数字音乐文件的Napster应用程序。虽然它由于其他非体系结构的原因而变得声名狼藉，但它验证了对等系统的可行性，并使体系结构模型向多个有价值的方向发展。

图2-3说明了对等应用的形式。应用由大量运行在独立计算机上的对等进程组成，进程之间的通信模式完全依赖于应用的需求。大量数据对象被共享，单个计算机只保存一部分的应用数据库，访问对象的存储、处理和通信负载被分布到多个计算机和网络连接中。每个对象在几个计算机中被复制，以便以后分散负载并在某个计算机断链时仍能正常工作（这在对等系统针对的大型异构网络中是不可避免的）。在众多计算机上放置对象并检索它们，同时维护这些对象的副本，这种应用需求使得对等体系结构本质上比客户服务器体系结构要复杂得多。

34 ~ 36

对等应用的开发和中间件对对等应用的支持将在第10章中深入介绍。

2.2.3 变体

在考虑了下列因素后，上述模型能派生出几个变体：

- 使用多个服务器和缓存，增加性能和灵活性。
- 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的需求。

由多个服务器提供的服务 服务可实现成在一个单独主机上的几个服务器进程，在必要时进行交互以便给客户进程提供服务（参见图2-4）。服务器可以将服务所基于的对象集分区，然后将这些

分区分布到各个服务器上；或者服务器可以在几个主机上维护复制的对象集。这两种选择可用下列例子说明。

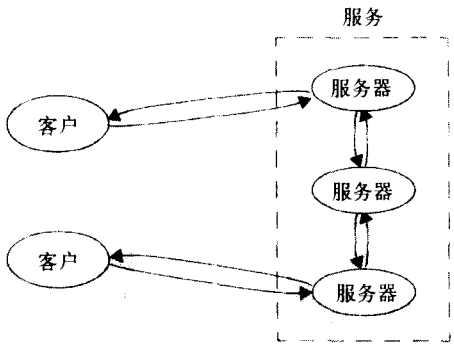


图2-4 由多个服务器提供的服务

Web就是一个常见的将数据分区的例子，其中的每个Web服务器管理它自己的资源集。用户可以利用浏览器访问任一个服务器上的资源。

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，该副本记录了用户登录名和加密的口令清单。第15章将详细讨论复制技术。

多服务器体系结构中紧耦合程度更高的是集群，它用于扩展性更高的Web服务，如搜索引擎和在线商店。一个集群最多可用数以千计的商用处理主板构成（见6.4.2节）。可在这些主板上对服务处理进行分区或复制。

37

代理服务器和缓存 缓存用于存储最近使用的数据对象。当计算机接收到一个新的对象，它就被存入缓存，必要的时候会替换缓存中已存在的对象。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拷贝可用就提供缓存中的对象；如果缓存没有可用的对象，才去取一个最新的拷贝。每个客户都可以配置缓存或者将缓存放置在由几个客户共享的代理服务器上。

缓存在实际工作中被广泛使用。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（见图2-5）为一个地点或多个地点的客户机提供共享的存放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。代理服务器能承担其他角色，例如它们可以用于通过防火墙访问远程Web服务器。

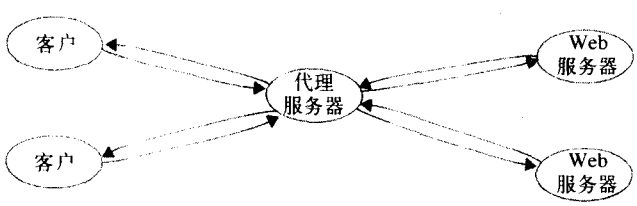
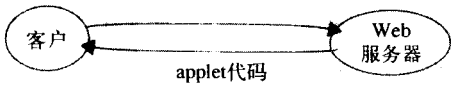


图2-5 Web代理服务器

移动代码 第1章介绍了移动代码。applet是一个众所周知的并被广泛使用的移动代码例子，即运行浏览器的用户选择了一个到一个applet的链接，applet的代码存储在Web服务器上，applet的代码下载到浏览器并在浏览器端运行，见图2-6。在本地运行下载的代码的好处是能够提供良好的交互响应，因为它不受与网络通信相关的延迟或带宽变化的影响。



a) 客户请求导致applet代码的下载



b) 客户与applet交互

图2-6 Web applets

访问服务意味着运行能调用服务所提供的操作的代码。一些服务可能进行了标准化，所以能用一个已有的且众所周知的应用访问它们——Web就是一个大家很熟悉的例子，但有些Web站点使用了在标准浏览器中找不到的功能，还要求下载额外的代码。例如，用额外的代码与服务器通信。考虑一个应用，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。这一功能不能通过与Web服务器的正常交互获得，因为那种交互总是由客户发起。解决方案是使用另外一种称为按推模式操作的软件，在这种方式下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能接收来自经纪人服务器的更新的applet，该applet可向用户显示更新，还可能自动地完成买卖操作，这些操作是根据顾客设置的、存储在顾客本地计算机上的条件而触发的。

第1章提到，移动代码对目的计算机中的本地而言资源是一个潜在的安全威胁。因此，浏览器采用7.1.1节讨论的方案对applet访问本地资源进行了限制。

移动代理 移动代理是一个运行的程序（包括代码和数据），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地点的本地资源——例如，访问一个数据库条目。如果将这种体系结构与对某些资源进行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而降低了通信开销和时间。

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，来比较多个销售商的产品价格。类似想法的一个早期例子是在Xerox PARC开发的所谓的蠕虫程序[Shoch and Hupp 1982]，该程序利用空闲的计算机完成密集型计算。

移动代理（和移动代码一样）对所访问的计算机上的资源而言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所代表的用户的身份决定允许使用哪些本地资源——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——如果它们访问所需的信息的要求被拒绝，那么它们可能完不成任务。由移动代理完成的任务可以通过其他手段完成。例如，需要通过因特网访问Web服务器上资源的Web蜘蛛程序可通过远程调用服务器进程而运行得相当成功。基于上述理由，移动代理的适用性是有限的。

网络计算机 在图1-2所示的体系结构中，应用在用户本地的计算机上运行。计算机的操作系统和应用软件通常要求大多数可运行的代码和数据位于本地磁盘上，但是应用文件的管理和本地软件库的维护所要求的技术大多数用户都不具备。

38
39

网络计算机提供了解决这个问题的途径。它能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，应用可在本地运行，但文件由远程文件服务器管理。像Web浏览器这样的网络应用也能运行。因为所有应用数据和代码由文件服务器保存，所以用户就可以把任务从一台网络计

算机迁移到另一台网络计算机。为了减少开销，对网络计算机的处理器和内存容量进行了限制。

如果网络计算机有硬盘，那它可以只保留最少的软件，剩余的磁盘用作缓存存储，存储最近从服务器下载的软件和数据文件的拷贝。缓存的维护不需要手工操作：当文件的一个新版本写入相关的服务器上后，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，由该软件层在用户本地的计算机上支持基于窗口的用户界面（参见图2-7）。该体系结构与网络计算机方案一样，具有管理开销和硬件开销低的特点，但是它在计算服务器——台能并行运行大量应用的计算机——上运行应用，而不是下载应用代码到用户计算机上。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等的多处理器版本的操作系统。

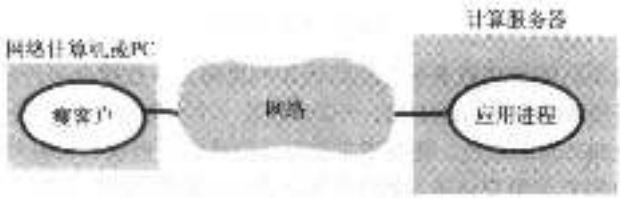


图2-7 瘦客户和计算服务器

瘦客户体系结构的主要缺点是：在交互频繁的图形活动（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。

瘦客户实现：在概念上瘦客户系统是简单的，并且它们的实现在某些环境中也很直观。例如，大多数UNIX的变体包含X-11窗口系统，见后文“X-11窗口系统”部分的讨论。

为了说明客户和服务器之间的边界，需要在图形操作流水线中列出支持一个图形用户界面的
40 若干要素。X-11在画线、画形状和窗口处理的图形原语层实现了这个边界。Citrix公司的WinFrame产品[www.citrix.com]是一个被广泛使用的实现了瘦客户概念的商业产品，它以类似的方式操作，该产品提供一个瘦客户进程，该进程可在众多平台上运行，支持对Windows NT主机上运行的应用提供交互访问的桌面系统。其他实现包括由英国Cambridge的AT&T实验室开发的虚拟网络计算机（VNC）系统[Richardson et al. 1998]。VNC在屏幕像素操作的层次给出了边界，当用户在不同的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，管理运行该进程的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通信，其中包括在窗口中写文本和绘制图形对象等。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服务器之名源于它给应用程序提供的图形显示服务，而称它为瘦客户软件是从使用运行在计算服务器上的应用程序的角度来说的。）

移动设备和自发互操作 我们已经讨论了移动代理形式的软件移动性。移动代理是在物理计算机之间迁移。与之相反，移动设备是在物理位置之间移动的计算组件的硬件，也就是设备带着软件组件在网络上移动。像第1章解释的那样，移动设备的数量不断增长，包括笔记本电脑、个人数字助理（PDA）之类的手持设备、移动电话和数码相机、智能手表之类的可穿戴计算机等。这些

设备多数具有无线联网功能，其范围为一个国家或更广的范围（如GSM和3G电信网络），范围有几百米，如WiFi（IEEE 802.11）或几米（如蓝牙）。

设备移动性可推广至客户－服务器系统。客户和服务器都可在移动设备上——移动客户是目前为止最常见的情况。考虑一辆环游世界的汽车，在WiFi可用的时候可用WiFi连接，在其他地方可用低带宽广域电信连接。乘客和他们的家人想更新阅读一个共享的Web网站，该网站包含这次旅游的图片和记录。实现该目标最通用的方式是大家都访问一个固定的服务器，汽车乘客上传数据并从移动客户端读取数据。另一种方法是在汽车上运行Web服务器，这样服务器自身是移动的。 41
服务器不仅提供用户的图片和记录，而且用GPS（全球定位系统）接收器，服务器能提供显示汽车当前位置的地图[Kindberg etal 2002a]。

移动透明性（见1.4.7节）是移动设备经常面临的一个问题，同时也是移动代理面临的问题。例如移动服务器的客户（如汽车上的移动客户）不必清楚汽车在网络之间移动（即使家人可能对汽车的地理位置感兴趣，而汽车的位置可以显示在最新的GPS地图上）。第3章讲述的移动IP将给出一个解决方案，使得汽车在不同网络之间移动时可为汽车上的服务器保留因特网（IP）地址，客户能使用永久URL访问汽车上的服务器。

变化的连接性是另一个重要的问题，它可能影响汽车作为服务器的能力和乘客访问外部服务的能力。汽车在穿越无线屏障（如隧道）时可能间歇地断开与无线网络的连接，或者在无线连接不通的地区长时间地断开连接。汽车在WiFi和电信连接之间移动时，会经历非常大的带宽变化。第14章将讨论处理断链的问题。第16章将讨论适应变化带宽的问题。

最后，移动性会导致*自发互操作*，它是客户－服务器模型的一个变体，其中设备之间的关联经常会被创建和删除。在1.2.3节中，我们描述了一个用户访问一个组织，用她的移动设备连接东道主的设备（如打印机）。类似地，可以向汽车乘客提供与汽车所经过的物理环境集成在一起的服务，例如关于当地名胜的信息。这里所面临的挑战是要使得互操作方便快捷（即自发），即使用户处于她以前从来没有到过的环境也是如此。这需要让访问者的设备与东道主网络通信，将设备与适合的本地服务相关联——这个过程叫*服务发现*。我们将在第16章深入探讨自发互操作和移动性的其他方面。第16章也涵盖了无处不在计算的相关领域，无处不在计算强调嵌入物理世界中的设备（例如传感器），而*上下文感知*服务是依赖于物理世界状态的，例如显示汽车当前位置的地图。

2.2.4 接口和对象

在一个进程（不论它是一个服务器进程还是一个对等的进程）中可调用的功能集合由一个或多个*接口定义*指定。接口定义的详细描述见第5章，但是这个概念对于精通Modula、C++或Java等语言的高手而言是很熟悉的。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，再加上相应的支持，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，这样它们的方法能被远程调用访问。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中，如Java）可能随系统活动要求而变化，在 42
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在前面介绍的更为动态的面向对象模型，进程之间和计算机之间责任的分布仍然是设计中一个重要的方面。在传统的体系结构模型中，责任是静态分配的（例如，文件服务器只负责文件，不负责Web页面、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（在某些情况下新的对象类型）能被实例化并马上可供调用。

2.2.5 分布式体系结构的设计需求

在分布式系统中触发对象和进程分布的因素有许多，它们的重要性不可小觑。20世纪60年代

通过使用共享文件，在分时系统中第一次实现了共享。共享的好处马上得到大家认可，并在多用户操作系统（如UNIX）和多用户数据库系统（如Oracle）中采用，以启用共享系统资源和设备（文件存储能力、打印机、音频和视频流）的进程和共享应用对象的应用进程。

以多处理器芯片形式出现的廉价的计算消除了共享中央处理器的需要，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）的共享要求导致了20世纪70年代和80年代分布式系统的开发。今天，资源共享被认为是理所当然的。但大规模的有效的数据共享对我们而言仍然是一个巨大的挑战——随着数据变化（大多数数据随时间变化），出现并发和冲突性更新的概率不断上升。关于控制共享数据的并发更新是第13、14章的主题。

性能问题 资源分布所带来的挑战远远不止是需要对并发更新进行管理。下面列出了由计算机和网络有限的处理能力和通信能力所带来的性能问题：

响应能力：交互应用的用户要求交互响应快速并一致，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，响应速度不仅由服务器和网络的负载及性能决定，还由所有涉及的软件组件——包括客户和服务器操作系统的通信，中间件服务（例如，远程调用支持）以及实现服务的进程代码——的延迟决定。

另外，即使进程在同一台计算机上，它们之间的数据传递和相关的控制转换相对还是较慢的。为了获得满意的交互响应时间，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，此时，访问本地缓存的页面和图像时响应速
43 度最快，因为它们由客户应用保存；访问远程文本页面的速度也比较快，因为它们的数据量小；而访问图像会有较长的延迟，因为涉及的数据量大。

吞吐量：计算机系统性能的一个传统的度量是吞吐量——计算工作完成的比例。我们关心分布式系统为它的所有用户完成工作的能力。这种能力受客户和服务器的处理速度以及数据传输率的影响。远程服务器上的数据必须经过两个计算机上的若干个软件层，从服务器进程传递到客户进程。与网络的吞吐量一样，其间的软件层的吞吐量也很重要。

平衡计算负载：分布式系统的一个目的是使应用和服务进程并发地运行而不必竞争同一资源，并能利用可用的计算资源（处理器、内存和网络能力）。例如，在客户计算机上运行applet的能力就减轻了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实现一个服务，一些负载很重的Web服务器（如搜索引擎、大型商业网站）有这样的需求。它利用了DNS域名查找服务提供的设施，为一个域名返回几个主机地址中的一个（见9.2.3节）。

在某些情况下，主机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 一旦提供给用户所要求的服务功能（如分布式系统中的文件服务），那么用户就会继续关心所提供服务的质量。影响客户和用户所感知的服务质量的系统的主要非功能性特征是*可靠性*、*安全性*和*性能*。满足变化的系统配置和资源可用性的*适应性*已被认为是服务质量的一个重要的方面。Birman一直在探讨质量的这些方面的重要性，他的书[Birman 1996]提供了质量的这些方面对系统设计影响的一些观点。

在大多数计算机系统设计中，可靠性和安全问题是非常关键的，它们与我们的两个基础模型：故障模型和安全模型密切相关，这两种模型分别在2.3.2节和2.3.3节介绍。

服务质量的性能方面最初按响应能力和计算吞吐量定义，但最近又按下面讨论的满足时序保证的能力重新定义了。不论用哪一种定义，分布式系统的性能方面都与2.3.1节定义的交互模型有很大的关系，这三个基础模型是贯穿全书的要点。

一些应用处理实时数据——指要按固定的速度处理的数据流或按固定的速度从一个进程传输到另一个进程的数据流。例如，电影播放服务包括一个客户程序，该客户程序从一个视频服务器检

索电影，然后将它展示在用户的屏幕上。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特指系统满足这些限制的能力，QoS的获得取决于在适当的时间必要的计算和网络资源的可用性。这意味着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）。 44

今天所广泛使用的网络，例如浏览Web，可能有非常好的性能特征，但当它们的负载很重时，它们的性能会迅速地恶化——凭借这一点就不能说它们提供了服务质量。QoS应用于操作系统和网络。每个关键资源必须为要求QoS的应用保留，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17章解决。

缓存和复制的使用 前面概述的性能问题经常是分布式系统成功部署的主要障碍，但在系统设计中，通过使用数据复制和缓存克服性能问题已有很大进展。2.2.2节介绍了缓存和Web代理服务器，但没有讨论服务器上的资源被更新后，被缓存的资源拷贝如何保持最新。为适应不同的应用，可使用多种缓存一致性协议。例如，第8章给出了两个不同的文件服务器设计所使用的协议。现在我们简单介绍作为HTTP协议的一部分的Web缓存协议，HTTP协议将在4.4节中介绍。

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。可以配置缓存一致性协议以便将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、可用性和断链操作，可放宽数据新鲜度条件。

浏览器或代理通过检查原来的Web服务器来了解缓存的响应内容是否还是最新的，从而验证缓存的响应内容。如果缓存的响应内容不能通过测试，那么Web服务器将返回一个最新的响应内容，用它替换缓存中过时的响应内容。当客户请求相应的资源时，浏览器和代理验证缓存的响应内容。但如果缓存的响应内容已经足够新，那么它们并不进行验证。即使Web服务器知道资源被更新的时间，它也不通知带缓存的浏览器和代理——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。为了让浏览器和代理确定它们存储的响应内容是否过时，Web服务器给它们的资源赋予一个大致过期的时间，这个时间可以从资源上一次被更新的时间估算出来。过期时间可能有误导作用，因为Web资源可能在任何时刻被更新。只要Web服务器响应一个请求，都会把资源的过期时间和服务器上的当前时间附到响应内容上。

浏览器和代理把过期时间和服务器时间与缓存的响应内容存储在一起。这使得浏览器或代理在接收请求后，能计算出缓存的响应内容是否可能过期。缓存的响应内容是否过期是通过与过期时间比较而得到的。响应内容的*年龄*（age）是响应内容被缓存的时间和服务器时间之和。注意，这种计算并不要求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。 45

可依赖性问题 可依赖性是大多数应用领域的需求。它不仅对于命令和控制活动非常关键——这里生命是利害攸关的，而且对于许多商业应用也很关键，包括快速发展的因特网商业领域，这里参与者的金融安全性和完备性取决于它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对体系结构的影响，而把实现安全和容错的相关技术放在后面介绍。开发检查或确保分布并发程序正确性的技术是正在进行的研究课题。虽然已获得一些有前途的结果，但还很少有成熟到能在实际应用中实施的。

容错：在出现硬件、软件和网络故障的时候可依赖的应用应该能继续正确工作。可靠性通过冗余获得——提供多个资源以便在出现故障时系统和应用软件能重新配置并继续执行它的任务。冗余很昂贵，它能应用的范围有限，因此能获得的容错的程度也有限。

在体系结构层，冗余要求使用多个计算机和多个通信路径，在多个计算机上运行系统的每个组件进程，在多个通信路径上传递消息。数据和进程可在需要时复制，以提供所需的容错级别。一种常见的冗余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台还在运行，

数据就可以被访问。有些关键的应用（如空中交通控制系统）对数据容错有很高的要求，这涉及为保持多个副本为最新要付出很高的代价。第15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更为可靠。例如，消息被重传直到接收到一个确认消息为止。RMI下层的协议的可靠性见第4、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找到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，只能被某些医生看到，而其他部分是大家都可以看的。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计算机中，因为典型的计算机并不构成一个安全的环境——用户能运行程序访问或更新存储在他们个人计算机中的任何一部分数据。2.3.3节将介绍解决更广泛的安全需
46 求的安全模型，第7章讲解可用于达到上述目标的技术。

2.3 基础模型

上面的各种系统模型具有一些基本特性。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，所有的模型都能满足前一节提出的设计需求，包括进程及网络的性能和可靠性特征，以及系统中资源的安全性。本节给出基于基本特性的模型，利用这些模型，我们能更详细地描述系统可能展示的特征、故障和安全风险。

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

- 系统中的主要实体是什么？
- 它们如何交互？
- 影响它们单个和集体行为的特征是什么？

模型的目的是：

- 显式地表示有关我们正在建模的系统的假设。
- 给定这些假设，就什么是可行的什么是不可行的给出结论。结论以通用算法或保证成立的特性的形式给出。特性成立的保证依赖逻辑分析和（适当时候的）数学证明。

了解设计依赖什么、不依赖什么我们就能从中获益。如果在一个特定系统中实现一个设计，只需验证假设是否成立，就能够推断出这个设计能否运作。通过清晰、显式地给出我们的假设，就能利用数学方法证明系统的特征，这些特征对任何满足假设的系统都成立。最后，通过从细节（如硬件）中抽象系统的基本实体，我们就能阐明对系统的理解。

我们希望在我们的基本模型中提取的分布式系统情况能解决下列问题：

交互：计算在进程中发生，进程通过传递消息，并引发进程之间的通信（例如信息流）和协调（活动的同步和排序）进行交互。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带来的延迟，这些延迟的持续时间会比较长，必须反映独立进程相互配合的准确性受限于这些延迟，受限于在分布式系统中很难跨所有计算机维护同一时间概念。

47 故障：只要分布式系统运行的任一计算机上出现故障（包括软件故障）或连接它们的网络中出现故障，分布式系统的正确操作就受到威胁，我们的模型将对这些故障进行定义和分类。这为分析它们潜在后果以及设计能容忍任何类型故障的系统奠定了基础。

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代理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，为分析系统的威胁以及设计能抵御这些威胁的系统奠定了基础。

为了帮助讨论和推理，我们对本章介绍的模型进行了必要的简化，省略了许多真实系统中的细节。它们与真实系统的关系，以及在模型帮助下揭示的问题环境中的解决方案是本书讨论的主题。

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以复杂方式进行交互的进程组成。例如：

- 多个服务器进程能相互协作提供服务。前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共同的目标。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——采取一系列步骤以执行期望的计算。简单的程序由算法控制，算法中的每一步都有严格的顺序。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。由多个上面所说的进程组成的分布式系统是很复杂的。它们的行为和状态能用*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，*包括它们之间消息的传递*。消息在进程之间传递以便在它们之间传递信息并协调它们的活动。

每个进程执行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非常困难，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。

交互的进程执行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更新的数据集组成，包括程序中的变量。属于每个进程的状态完全是私有的——也就是说，它不能被其他进程访问或更新。

本节讨论分布式系统中影响进程交互的两个重要因素：

48

- 通信性能。
- 不可能维护一个全局时间概念。

通信通道的性能 在我们的模型中，通信通道在分布式系统中可用许多方法实现，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与延迟、带宽和抖动有关的性能特征：

- 从一个进程开始发送消息到另一个进程开始接收消息之间的间隔时间称为*延迟*。延迟包括：
  - 第一串比特从网络传递到目的地所花费的时间。例如，通过卫星链接传递消息的延迟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，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有空。
  - 操作系统通信服务在发送进程和接收进程上所花费的时间，这个时间会随操作系统当前的负载的变化而变化。
- 计算机网络的*带宽*是指在给定时间内网络能传递的信息总量。当大量通信通道使用同一个网络时，它们就不得不共享可用的带宽。
- *抖动*是传递一系列消息所花费的时间的变化值。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 分布式系统中的每台计算机有自己的内部时钟，本地进程用这个时钟获得当前时间值。因此，在不同计算机上运行的两个进程能将时间戳与它们的事件关联起来。但是，即使两个进程在同时读它们的时钟，它们各自的本地时钟也会提供不同的时间值。这是因为计算机时钟和绝对时间之间有偏移，更重要的是，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情况下都设置成相同的时间，它们的时钟最后也会相差巨大，除非进行校正。

有几种校正计算机时钟的时间的方法。例如，计算机可使用无线电接收器从全球定位系统（GPS）以大约1μs的精度接收时间读数。但GPS接收器不能在建筑物内工作，同时，为每一台计算

机增加GPS在费用上也不合理。相反，具有精确时间源（如GPS）的计算机可发送时序消息给网络中的其他计算机。在两个本地时钟时间之间进行协商当然会受消息延迟的影响。有关时钟漂移和
49 时钟同步的更详细的讨论见第12章。

交互模型的两个变体 在分布式系统中，很难对进程执行、消息传递或时钟漂移的时间设置时间限制。两种截然相反的观点提供了一对简单模型：第一个模型对时间有严格的假设，第二个模型对时间没有假设。

同步分布式系统：Hadzilacos和Toueg[1994]定义了一个同步分布式系统，它满足下列约束：

- 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。
- 每个进程有一个本地时钟，它与实际时间的偏移率在一个已知的范围内。

对于分布式系统，建议给出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。但是达到实际值并对所选值提供保证是比较困难的。除非能保证上下界的值，否则任何基于所选值的设计都不会可靠。但是，按同步系统构造算法，可以对算法在实际分布式系统的行为提供一些想法。例如，在同步系统中，可以使用超时来检测进程的故障，见2.2.3节。

同步分布式系统是能够被构造出来的。所要求的是进程用已知的资源需求完成任务，这些资源需求保证有足够的处理器周期和网络能力；还有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，是非常有用的，但它们不具备同步系统的资格。因此我们需要另一个模型：异步分布式系统，它是对下列因素没有限制的系统：

- 进程执行速度——例如，进程的一步可能只花费亿万分之一秒，而进程的另一步要花费一个世纪的时间，也就是说，每一步能花费任意长的时间。
- 消息传递延迟——例如，从进程A到进程B传递一个消息的时间可能快得可以忽略，也可能要花费几年时间。换句话说，消息可在任意长时间后接收到。
- 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假设。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范围，对像用ftp传输文件要花费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。下面的“Pepperland协定”部分说明在异步分布式系统中达成协定的困难性。

即使有假设，有些设计问题也能得到解决。例如，虽然Web并不总能在一个合理的时间限制
50 内提供特定的响应，但浏览器能让用户在等待时做其他事情。对异步分布式系统有效的任何解决方案对同步系统同样有效。

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通信通道。例如，如果有太多未知特性的进程共享一个处理器，那么任何一个进程的性能都不能保证。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，可使用同步模型。下面的“Pepperland协定”部分将说明在异步系统中同步时钟的不可能性。

事件排序 在许多情况下，我们有兴趣知道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、之后或同时。尽管缺乏精确的时钟，但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Meeting的消息。

2）用户Y和Z发送一个主题为Re：Meeting的消息进行回复。

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复，该回复引用了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2-8所示的一样，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：Meeting
24	X	Meeting
25	Y	Re：Meeting

Pepperland协定 Pepperland军队的两个师“红师”和“蓝师”驻扎在邻近两座山的山顶上。山谷下面是入侵的敌军。只要Pepperland的两个师留在驻地，他们就是安全的，他们通过派出通信兵通过山谷进行通信。Pepperland的两个师需要协商它们中的哪一方率先发起对敌军的冲锋以及冲锋何时进行。即使是在异步的Pepperland，由谁率先冲锋是可能达成一致的。例如，每个师报告剩余人员的数量，人数多的一方率先冲锋（如果人数一样多，则由红师率先冲锋）。但何时冲锋呢？非常遗憾，在异步Pepperland，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，蓝师可能三个小时也收不到这个消息，也可能五分钟就收到这个消息了。在同步Pepperland中，仍然有协调问题，但是两个师知道一些有用的约束：每个消息至少花费min分钟至多花费max分钟到达。如果率先冲锋的师发送“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。在率先冲锋的师之后不超过（$max-min+1$）分钟另一个师保证发起冲锋。

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可以附带本地计算机时钟的时间。例如，消息m_1、m_2和m_3能附带时间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，那么这些时间戳通常会以正确的顺序排列。

因为在一个分布式系统中时钟不能精确同步，所以Lamport[1978]提出了*逻辑时间*的模型，用于为在分布式系统中运行于不同计算机上的进程的事件提供顺序。使用逻辑时间不需要求助于时钟就可以推断出消息的顺序。详细内容可参见第11章，我们在这里只介绍逻辑排序的某些方面如何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，我们为图2-8所示的成对事件给出一个逻辑排序。例如，仅考虑涉及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，因此对于Y，我们有下列逻辑排序：

Y在发送m_2之前接收m_1

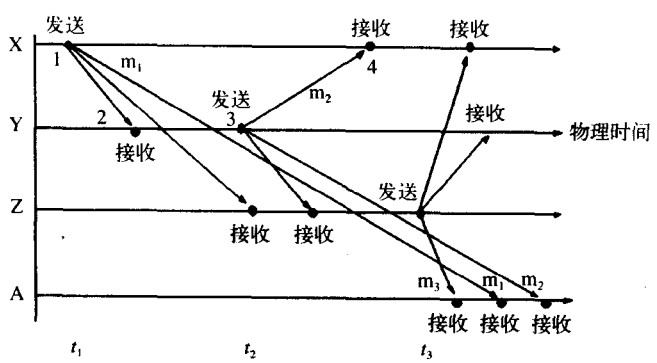


图2-8 事件的实时排序

逻辑时间通过给每个事件赋予一个与它的逻辑顺序相对应的数字而进一步拓展了这个思想。这

样，后发生的事件的数字比早发生的事件的数字大。例如，图2-8为X和Y上的事件分配了数字1～4。

2.3.2 故障模型

在分布式系统中，进程和通信通道都有可能出故障，即它们可能偏离被认为是正确或所期望的行为。故障模型定义了故障可能发生的方式，以便理解故障所产生的影响。Hadzilacos和Toueg[1994]提供了一种分类法，用于区分进程故障和通信通道故障。这些故障将分别在下面的“遗漏故障”、“随机故障”和“时序故障”部分介绍。

本书将始终使用故障模型。例如：

- 第4章将给出数据报和流通信的Java接口，它们分别提供不同程度的可靠性。
- 第4章将给出支持RMI的请求－应答协议。它的故障特征取决于进程和通信通道两者的故障特征。请求－应答协议能用数据报或流通信实现。可根据实现的简单性、性能和可靠性作出决定。
- 第14章将给出事务的两阶段提交协议。它用于解决进程和通信通道的确定性故障。

遗漏故障 遗漏故障类错误指的是进程或通信通道不能完成要求它做的动作。

进程遗漏故障： 进程主要的遗漏故障是崩溃。当我们说进程崩溃了，意思是进程停止了，将不再执行程序的任何步骤。能在故障面前存活的服务，如果假设该服务所依赖的服务能干净利落地崩溃——即进程或者正确运行或者停止运行，那么它的设计能简化。一个进程p是通过不能获得对q的调用消息的应答而检测到进程q崩溃的。然而，这种崩溃检测的方法依赖超时的使用，即进程用一段固定时间等待某个事件的发生。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，也可能是执行速度慢，即消息还没有到达。

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，如果当确保消息已被发送，而其他进程又没有响应时，进程使用超时，那么就会产生故障－停止行为。例如，对于进程p和q，如果设计q应答来自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可以得出结论：进程q出故障了。下面的“故障检测”和“在故障前达成协定的不可能性”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。

51～53

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。通信通道将m传输到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传递来执行receive（见图2-9）。通常由操作系统提供外发消息缓冲区和接收消息缓冲区。

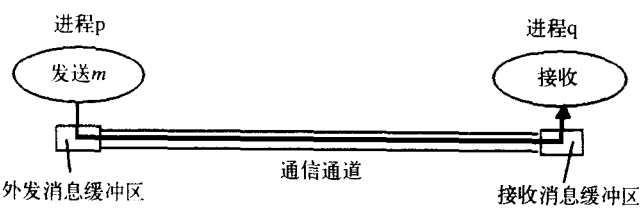


图2-9 进程和通道

如果通信通道不能将消息从p的外发消息缓冲区传递到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“丢失消息”，造成消息丢失通常是因为在接收端或中间的网关上缺乏缓冲区空间，或因为网络传输错误（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见图2-10。

故障检测 在Pepperland师驻扎在山顶的情况下（见“Pepperland协定”部分），假设敌军聚集足够的力量攻击任意一个扎营的师，那么任意一个师都可能失败。进一步假设，在没有被攻击的时候，各师定时地派出通信兵向对方报告自己的状态。在异步系统中，没有一个师能区别是对方被打败了还是通信兵跨越中间山谷的时间太长。在同步的Pepperland中，一个师通过应该定期出现的通信兵的缺席就能判断出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。

存在故障时达成协定的不可能性 我们一直假设Pepperland通信兵最终总能设法通过山谷，但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”，从而让他传达错误的消息）。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吗？非常遗憾，正如Pepperland理论家Ringo大师证明的一样，在这样的环境中，两个师不能一致地决定做什么。为了了解这一点，假设两个师执行达成一致的Pepperland协议。某一方提出“冲锋！”或“投降！”，协议使得双方同意这一方或另一方的动作。现在考虑在某一轮协议中发送的最后一个消息。携带消息的通信兵可能被敌军俘虏。不论消息到达与否，最后的结果必须一致。所以我们去掉它。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俘虏，就没有保证Pepperland师之间一致的协议存在。

故障可以按照它们的严重性分类。到现在为止，我们描述的所有故障是良性故障。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以及时序故障和性能故障。

随机故障 术语随机故障或拜占庭故障用于描述可能出现的最坏的故障，此时可能发生任何类型的错误。例如，一个进程可能在数据项中设置了错误的值，或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需要的处理步骤。因此，进程的随机故障不能通过查看进程是否应答调用来检测，因为它可能随机地遗漏应答。

通信通道也会出现随机故障。例如，消息内容可能被损坏或者发送不存在的消息，也可能多次发送实际的消息。通信通道的随机故障很少，因为通信软件能识别这类故障并拒绝出错的消息。例如，可用校验和检测损坏的消息，消息序号可用于检测不存在和重复的消息。

故障类型	影响对象	描　述
故障－停止	进程	进程停止并一直停止。其他进程可检测到这个状态
崩溃	进程	进程停止并一直停止。其他进程可能无法检测到这个状态
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，但消息没有放入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2-10 遗漏故障和随机故障

时序故障 时序故障适用于同步分布式系统。在这样的系统中，进程执行时间、消息传递时间和时钟漂移率均有限制。时序故障见图2-11的列表。这些故障中的任何一个均可导致在指定时间间隔内对客户没有响应。

在异步分布式系统中，一个负载过重的服务器的响应时间可能很长，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，但这种系统在设计上很复杂，会要求冗余

的硬件。大多数通用的操作系统（如UNIX）不能满足实时限制。

故障类型	影响对象	描　述
时钟	进程	进程的本地时钟超过了与实际时间的漂移率的范围
性能	进程	进程超过了两个进程步之间的间隔范围
性能	通道	消息传递花费了比规定的范围更长的时间

图2-11 时序故障

时序与有音频和视频通道的多媒体计算机关系尤为密切。视频信息要求传输海量的数据。若要在传递视频信息时不出现时序故障，那么就要对操作系统和通信系统提出特殊的要求。

故障屏蔽 分布式系统中的每个组件通常是基于其他一组组件构造的。利用存在故障的组件构造可靠的服务是可能的。例如，保存有数据副本的多个服务器在其中一个服务器崩溃时能继续提供服务。了解组件的故障特征有利于在设计新服务时屏蔽它所依赖的组件的故障。一个服务通过隐藏故障或者通过将它转换成一个更能接收的故障类型来屏蔽故障。对于后者，我们给出一个例子，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章看到通过使用将不能到达目的地的消息重传的协议可以隐藏遗漏故障。第15章将介绍利用复制进行故障屏蔽的方法。甚至进程崩溃也可以屏蔽——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。

一对一通信的可靠性 虽然基本的通信通道可能出现前面描述的遗漏故障，但用它来构造一个能屏蔽某些故障的通信服务是可能的。

术语可靠通信可从下列有效性和完整性的角度来定义：

有效性：外发消息缓冲区中的任何消息最终能发送到接收消息缓冲区。

完整性：接收到的消息与发送的消息一致，没有消息被发送两次。

对完整性的威胁来自两个方面：

- 任何重发消息但不拒绝到达两次的消息的协议。要检测消息是否到达了两次，可以在协议中给消息附加序号。
- 心怀恶意的用户，他们可能插入伪造的消息、重放旧的消息或篡改消息。在面对这种攻击时为维护完整性要采取相应的安全措施。

2.3.3 安全模型

在2.2节中，我们认为构建分布式系统源于对资源共享的需求，我们用封装了对象和通过与其他进程的交互来访问它们的进程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：

通过保证进程和用于进程交互的通道的安全以及保护所封装的对象不会在未经授权时访问可达到分布式系统的安全。

这里，采用了保护对象这种说法，尽管这些概念可平等地应用到所有类型的资源上。

保护对象 图2-12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，由客户程序向服务器发送调用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由不同的用户使用。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。为了解决这样的问题，通过访问权限指定允许谁执行一个对象的操作——例如，允许谁读或写它的状态。

这样，我们必须在我们的模型中包括作为访问权限受益人的用户。我们将每个调用和每个结果均与对应的授权方相关联。这样的一个授权方称为一个主体。一个主体可以是一个用户或进程。

54 ~ 57

在我们的图示中，调用来自用户，结果来自服务器。

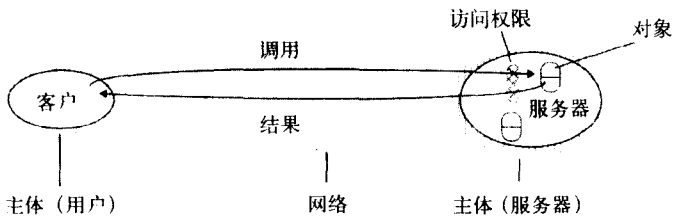


图2-12 对象和主体

服务器负责验证每个调用的主体的身份，检查它们是否有足够的访问权限在所调用的某个对象上完成所请求的操作，如果没有权限就拒绝它们的请求。客户可以检查服务器的主体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。消息易于受到攻击，因为它们所使用的网络和通信服务是开放的，以使得任一对进程进行交互。服务器和对等进程暴露它们的接口，使得任何其他进程能给它们发送调用。

分布式系统经常在可能受到来自敌对用户的外部攻击的任务中使用和部署。对处理金融交易、机要或秘密信息以及任何注重保密性或完整性的信息的应用而言，这一点是千真万确的。完整性会由于违反安全规则以及通信故障而受到威胁。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对在进程之间传送的消息的威胁。但为了识别和抵御这些威胁，我们如何分析它们呢？下面的讨论将介绍分析安全威胁的一个模型。

敌人 为了给安全威胁建模，我们假定敌人（有时也称为对手）能给任何进程发送任何消息，并读取或复制一对进程之间的任何消息，如图2-13所示。这种攻击能很简单地实现，它利用连接在网上的计算机运行一个程序读取发送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。攻击可能来自合法地连接到网络的计算机或以非授权方式连接到网络的计算机。

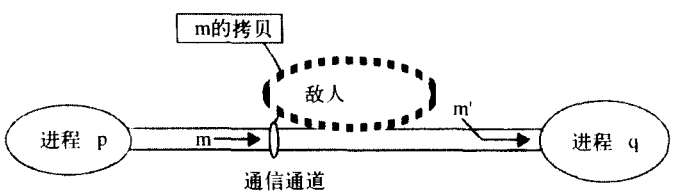


图2-13 敌人

来自一个潜在敌人的威胁将在下面的“对进程的威胁”，“对通信通道的威胁”和“服务拒绝”部分讨论。

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以接收来自其他进程的消息，它没有必要确定发送方的身份。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。缺乏消息源的可靠的知识对服务器和客户的正确工作而言是一个威胁，具体解释如下：

- **服务器**：因为服务器能接收来自许多不同客户的调用，所以它未必能确定进行调用的主体的身份。即使服务器要求在每个调用中加入主体的身份，敌人也可能用假的身份生成一个调用。在没有关于发送方身份的可靠知识时，服务器不能断定应执行操作还是拒绝操作。例如，

邮件服务器不知道从指定邮箱中请求一个邮件的用户是否有权限这样做，或者它是否为来自一个敌人的请求。

- **客户**：当客户接收到服务器的调用结果时，它未必能区分结果消息来自预期的服务器还是来自一个“哄骗”邮件服务器的敌人。因此，客户可能接收到一个与原始调用无关的结果，如一个假的邮件（不在用户邮箱中的邮件）。

对通信通道的威胁：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，这种攻击会对信息的私密性和完整性构成威胁，对系统的完整性也会构成威胁。例如，包含用户邮件的结果消息可能泄漏给另一个用户或者它可能被改变成完全不同的东西。

另一种形式的攻击是试图保存消息的拷贝并在以后重放这个消息，这使得反复重用同一消息成为可能。例如，有些人通过重发请求从一个银行账户转账到另一个银行账户的调用消息而受益。

利用安全通道可解除这些威胁，安全通道是基于密码学和认证的，详细内容见下面的描述。

解除安全威胁 下面将介绍安全系统所基于的主要技术。第7章将详细讨论安全的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（例如某个客户和某个服务器）共享一个秘密，即它们两个知道秘密但分布式系统中的其他进程不知道这个秘密。如果由一对进程交换的消息包括证明发送方共享秘密的信息，那么接收方就能确认发送方是一对进程中的另一个进程。当然，必须小心以确保共享的秘密不泄露给敌人。

58
59

*密码学*是保证消息安全的科学，*加密*是将消息编码以隐藏其内容的过程。现代密码学基于使用密钥（很难猜测的大数）的加密算法来传输数据，这些数据只能用相应的解密密钥恢复。

认证：使用共享秘密和加密为消息的认证——证明由发送方提供的身份——奠定了基础。基本的认证技术是在消息中包含加密部分，该部分中包含足够的消息内容以保证它的真实性。对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请求的主体身份的表示、文件的标识、请求的日期和时间，所有内容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，安全通道作为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通信通道，每个进程代表一个主体行事，如图2-14所示。一个安全通道有下列特性：

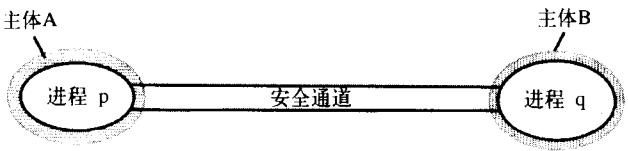


图2-14 安全通道

- 每个进程确切知道其他正在执行的进程所代表的主体身份。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主体身份，并能在执行操作之前检查它们的访问权限。这使得服务器能正确地保护它的对象，以便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以防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（SSL）协议（见第7章的讨论）就是安全通道的实例。

60

其他可能的来自敌人的威胁 1.4.3节简要介绍了两个安全威胁——拒绝服务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，我们要再介绍一下这两个安全威胁。

拒绝服务：在这种攻击形式下，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。这种攻击通常意在延迟或阻碍其他用户的动作。例如，建筑物中的电子门锁可能由于受到对计算机控制的电子锁的过多非法请求而失效。

移动代码：如果进程接收和执行来自其他地方（如1.4.3节提到的邮件附件）的程序代码，那么这些移动代码就会带来新的、有趣的安全问题。这样的代码很容易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包括了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，因此必须非常小心地构造宿主环境以避免攻击。其中的大多数问题已在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。这一点也很好地说明了所有安全系统的设计都需要严格的分析。

安全模型的使用 分布式系统的安全涉及根据预定义的访问权限控制对象的访问以及通信的安全通道的使用。有人认为，在分布式系统中获得安全是件简单的事。但是通常却不是这样。安全技术（如加密）和访问控制的使用会产生实质性的处理和管理开销。前面概述的安全模型提供了分析和设计安全系统的基础，其中这些开销保持最少，但对分布式系统的威胁会在许多地方出现，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。这种分析涉及构造威胁模型，由它列出系统会遭遇的各种形式的攻击、风险评估和每个威胁所造成的后果。要在抵御威胁所需的安全技术的有效性和开销之间做出权衡。

2.4 小结

大多数分布式系统都根据某种体系结构模型安排。客户－服务器模型是一种流行的体系结构模型——Web和其他因特网服务（如ftp、新闻和邮件以及Web服务和DNS）均基于这个模型，文件归档和其他本地服务也是如此。像DNS这种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能力。客户和代理服务器上的缓存得到广泛使用以提高服务的性能。 61

在对等模型中，所有进程扮演类似的角色，利用大量参与工作的大量计算机上的资源完成共同的任务。

将代码从一个进程移动到另一个进程的能力导致了客户－服务器模型的一些变体的出现。最常见的例子是applet，它的代码由Web服务器提供，由客户运行，为客户提供它所不具备的功能，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使用户能够在不方便使用桌面计算机的时候访问本地和因特网服务。分布式系统中移动设备的特征是它们不能预测连接和断链。这导致了客户－服务器模型的变体——自发互操作的出现，在自发互操作中，设备之间的关联经常被创建或销毁。

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共同特征。交互模型关注进程和通信通道的性能以及全局时钟的缺乏。它将同步系统看成在进程执行时间、消息传递时间和时钟漂移上有已知范围的系统，将异步系统看成在进程执行时间、消息传递时间和时钟漂移上没有限制的系统——时钟漂移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通信通道故障进行了分类。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

屏蔽基本通信通道的故障，可从基本的通信通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。有效性是可靠通信的另一个性质——它要求发送消息缓冲区中的任何消息最终都能传递到接收消息缓冲区。

安全模型可识别出一个开放的分布式系统中对进程和通信通道可能的威胁。有些威胁与完整性有关：恶意用户可能篡改消息或重放消息。其他的威胁则会损害私密性。另一个安全问题是发送消息所代表的主体（用户或服务器）的认证。安全通道使用密码技术来确保消息的完整性和私密性，并使得相互通信的主体可以进行验证。

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。 （第35页）

62 2.2 对于练习2.1中描述的应用，叙述服务器如何协作以提供服务。 （第37页）

2.3 对于练习2.1中讨论的应用，如何进行服务器之间的数据分区和复制（或缓存）？ （第37页）

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蜘蛛任务创建和更新索引。在这些并发的活动之间进行同步的需求是什么？ （第35页）

2.5 在对等系统中使用的主机通常是用户办公室或家里的计算机。对共享数据对象的可用性和安全性而言，这意味着什么？通过使用复制能多大程度上克服弱点？ （第35页，第37页）

2.6 列出易受不可靠程序（从远程站点下载并在本地运行的程序）攻击的本地资源的类型。 （第37页）

2.7 通过实例说明使用移动代码的好处。 （第37页）

2.8 对于访问由服务器管理的共享数据的应用，什么因素影响应用的响应能力？描述补救的方法并讨论其有效性。 （第43页）

2.9 区分缓冲和缓存。 （第45页）

2.10 给出在分布式系统中能/不能通过使用冗余来容忍的软硬件故障的例子。在适当的情况下，使用何种程度的冗余能使系统容错？ （第46页）

2.11 设计一个简单的服务器，它不用访问其他服务器就可完成客户请求。解释为什么它通常不可能对服务器响应客户请求的时间设置限制。需要怎样做才能使服务器可以在一定时间范围内执行请求？这是一个实用的选择吗？ （第48页）

2.12 针对影响通信通道上的两个进程之间传递消息所花的时间的每个因素，说明需要对哪些影响总时间的度量设置限制。为什么在当前通用的分布式系统中不提供这些度量？ （第49页）

2.13 网络时间协议服务能用于同步计算机时钟，解释为什么即使使用该服务，也不能对两个时钟之间的不同也不能给出确定的范围。 （第49页）

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、延迟或发送得太快以致接收方无法处理它，但到达目的地的消息的内容一定正确。

描述上面两个服务会出现的故障类型，根据它们对有效性和完整性的影响为故障分类。

63 服务B能被描述成可靠的通信服务吗？ （第53页，第56页）

2.15 有一对进程X和Y，它们使用练习2.14中的通信服务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，最后从Y到X发送应答。思考这样一个调用会出现的故障类型。 （第53页）

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会出现的故障类型。阐述如何屏蔽故障以产生另一种良性故障，并对如何屏蔽良性故障提出建议。

（第56页）

2.17 定义可靠通信的完整性，列出所有来自用户和系统组件的对完整性的可能的威胁。面对每种威胁，要采取什么手段确保完整性？（第56页，第59页）

2.18 简述可能出现在因特网上的几类主要的安全威胁（对进程的威胁、对通信通道的威胁、服务拒绝）。（第58页） 64

第3章　网络和网际互连

分布式系统使用局域网、广域网和互连网络进行通信。底层网络的性能、可靠性、可伸缩性、移动性以及服务质量都影响着分布式系统的行为，因而也影响这些系统的设计。为满足用户需求的改变，无线网络和有服务质量保障的高性能网络应运而生。

计算机网络所基于的原理包括协议分层、包交换、路由以及数据流等，网际互连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中。因特网中使用的寻址以及路由方案经受了因特网快速成长所带来的影响。它们也被不断地修正，以适应未来的发展并满足新的对移动性、安全性以及服务质量的需求。

在实例研究中将给出特定网络的技术设计，包括以太网、IEEE 802.11（WiFi）和蓝牙无线网
65 络以及异步传输模式（ATM）网络。

3.1　简介

要构建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、集线器、转发器和网络接口；最后还需要*软件组件*，包括协议栈、通信处理器和驱动器。上述因素都会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组件称为*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*。术语*节点*指的是在网络上的所有计算机或者交换设备。

因特网是一个通信子系统，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*。一个子网是一个路由单位（负责在因特网不同部分之间传递数据），它包含一组互连的节点，它们之间采用相同的技术进行通信。因特网的基础设施包括体系结构和软硬件组件，它们将不同的子网有效地集成为一个数据通信服务。

通信子系统的设计在很大程度上受组成分布式系统的计算机所使用的操作系统的特征的影响，也受与之互连的网络的影响。本章将探讨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，对计算机网络加以概述。不熟悉计算机网络的读者应该将本章作为本书后续内容的基础，而熟悉网络的读者也会发现本章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进行了总结。

计算机发明后不久，人们就有了计算机网络的构想。1961年，Leonard Kleinrock[1961]第一次在一篇文章中提出了包交换的理论基础。1962年J.C.R. Licklider和W. Clark，这两位于20世纪60年代初期在MIT参加第一个分时系统的开发的科学家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的轮廓[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 2006, Kurose and Ross 2000]。

本节后面的部分将讨论分布式系统的通信需求。3.2节将对网络类型进行概括，3.3节将介绍计算机网络原理，3.4节将专门讨论因特网。3.5节将给出有关以太网、IEEE 802.11（WiFi）、蓝牙和
66 ATM网络技术的实例研究。

分布式系统的连网问题

早期的计算机网络只能满足少量的、相对简单的应用需求，支持像文件传输、远程登录、电子邮件、新闻组这样的网络应用。随着分布式系统的不断发展，分布式应用程序能访问共享的文件或其他资源，为满足交互应用的需求，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的使用模式的出现，对于网络可靠性、可伸缩性、移动性、安全性和服务质量提出了更高要求。本节将详细介绍这些需求的本质。

性能 我们感兴趣的网络性能参数是影响两个互连计算机间消息传输速度的参数。它们是延迟和点到点的数据传输率。

*延迟*是指执行发送操作之后数据到达目的地之前这一段时间。它可以用传输一个空消息的时间来度量。这里我们只考虑网络延迟，它是2.3.1节定义的进程－进程延迟的一部分。

*数据传输率*是指一旦传输过程开始，数据在网络上两台计算机间传输的速度，通常用bps（比特/秒）作为单位。

根据上述定义，要在两个计算机间传输长度为length比特的消息，网络所需的时间为：

消息传输时间=延迟+length/数据传输率

上式还需满足以下条件：消息长度不能超过网络所允许的最大值。长消息会被分割成多个段，传输时间是多个段传输时间的总和。

网络的传输率主要是由它的物理特征决定的，而延迟则主要由软件开销、路由延迟和与负载有关的统计因素（源于访问传输信道的冲突性命令）决定。分布式系统的许多在进程之间传送的许多规模消息很小，因此延迟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*是度量吞吐量的指标，它表示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），每一次数据传输都使用了整个网络的传输容量，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。但是，在网络过载时网络性能会恶化，过载是指同时在网络中传输的消息过多。过载给网络的延迟、数据传输率以及系统总带宽所带来的影响与网络技术紧密相关。

67

现在考虑客户－服务器通信的性能。在负载较轻的本地网环境（包括系统开销）下，节点之间传输一个短的请求消息和收到一个短的应答的总时间通常在半毫秒左右；而调用一个本地内存中的应用层对象的操作，所需的时间在微秒以内。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倍以上。但是网络的延迟和带宽经常超越硬盘的性能；对于访问一个本地的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。

信息在因特网上往返的延迟在50～750ms之间，平均值为200ms左右，所以在因特网上传送请求比在快速本地网络上传送大约慢100倍。这个时间差缘于路由器的交换延迟和网络电路的竞争。

6.5.1节将详细讨论和比较本地操作和远程操作的性能问题。

可伸缩性 计算机网络已成为现代社会不可缺少的基础设施。图1-4显示了近25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，到那时网络上将有数十亿的节点和上亿可用的主机。

这些数字表明了因特网必须能够处理未来在数量和负载上的变化，目前的网络技术甚至不能很好地应付现在的网络规模；但它们已经表现得相当不错了。为了适应因特网下一阶段的发展，技术人员正在对寻址和路由机制进行一些实质性的改变，这方面内容将在3.4节加以讨论。对于简单的客户－服务器应用（比如Web），未来的数据流量的增长至少将和上网用户数量成正比。因特网基础设施的能力是否能适应这样的增长，必须依赖经济学的使用，特别是在对用户的收费和实际发生的通信模式方面——比如应根据用户的位置范围做某种处理。

可靠性 2.3.2节关于故障模型的讨论描述了通信错误所带来的影响。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（参见2.2.1节）也进一步支持了“通信子系统无需提供完全无错的通信”这一观点，通信错误的检测和校正通常由应用级软件完成。大多数物理传输介质的可靠性很高。错误通常是由于发送方或接收方的软件故障（例如，接收方计算机未能接收到一个包）或者缓冲溢出造成的，而不是网络错误造成的。

安全性 第7章将列出分布式系统获得安全性所需的需求和技术。大多数组织采用的第一层防
68 御是为他们的网络和计算机设置一个防火墙。防火墙在组织的内部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上的资源不被外部用户或进程访问，并控制组织中的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，允许某些进入或流出的消息通过并拦截其他消息。这个内容我们将在3.4.8节中继续讨论。

为了让分布式应用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，使得大部分分布式应用可以被部署，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且更灵活的安全形式。它通常应用于通信子系统以上的层次，因此不在这里讨论，而是在第7章进行讨论。但也有一些例外的要求，包括保护网络组件（如路由器）的操作不会受到未授权的干涉，对移动设备和其他外部节点建立安全链接以便它们能参与到一个安全的企业内部网——虚拟私网（VPN）的概念，VPN将在3.4.8节讨论。

移动性 移动设备（如笔记本电脑、PDA和可连网的移动电话）常常改变所处的位置，可以在方便的网络连接处重新连入，或者甚至在移动的时候使用。虽然无线网络提供了对这些设备的连接，但因特网的寻址和路由机制都是在移动设备出现之前开发的，并不太适合与不同子网进行间歇连接的需求。虽然因特网机制已经有所改进并被扩展来支持移动性，但随着未来移动设备使用数量的增长，还必须进行更进一步地开发。

服务质量 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流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通信通道有足够的带宽和并对延迟有所限制。一些应用能动态地改变它们的要求，并指定可接受的最低服务质量和期望的最佳值。第17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进行的，但也经常有一对多通信的需求。显然这可以用向多个地址发送来模拟，但这种方式所花的代价比真正需要花费的代价要大，而且也不具备应用所需的容错性。因为这些原因，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

69 本节介绍主要用于支持分布式系统的网络类型：个域网、局域网、广域网、城域网以及它们的无线变体。互连网络（如因特网）是基于这些类型的网络构造出来的。图3-1给出了下面讨论的各种网络的性能特征。

一些网络类型的名字经常会被混淆，因为它们看上去指的是物理范畴（局域、广域），其实它们也确定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，尽管一些网络技术，如ATM（异步传输模式）既适用于局域网又适用于广域网，一些无线网络也同时支持局域网和城域网传输。

我们把由很多互连的网络组成，并且集成起来提供单一数据通信介质的网络称为互连网络。因特网就是典型的互连网络，它由数百万的局域网、城域网和广域网组成。我们将在3.4节详细描述它的实现。

	实　例	范　围	带宽（Mbps）	延迟（ms）
有线：				
LAN	以太网	1～2kms	10～1000	1～10
WAN	IP路由	世界范围	0.010～600	100～500
MAN	ATM	2～50kms	1～150	10
互连网络	因特网	世界范围	0.5～600	100～500
无线：				
WPAN	蓝牙（IEEE 802.15.1）	10～30m	0.5～2	5～20
WLAN	WiFi（IEEE 802.11）	0.15～1.5km	2～54	5～20
WMAN	WiMAX（IEEE802.16）	5～50km	1.5～20	5～20
WWAN	GSM、3G电话网	世界范围	0.010～2	100～500

图3-1　网络性能

个域网　个域网（Personal Area Network, PAN）是本地网的子类，其中一个用户携带的各种数字设备由一个廉价、低能量网络连接起来。有线PAN不是太重要，因为很少有用户希望自己身上有有线网络，但由于移动电话、PDA、数码相机、音乐播放器等个人设备数量的增加，无线个域网（WPAN）的重要性也随之增加。我们将在3.5.3节描述蓝牙WPAN。

局域网　局域网（Local Area Network, LAN）在由单一通信介质连接的计算机之间以相对高的速度传输消息，这里的通信介质包括双绞线、同轴电缆和光纤。网段是指为某个部门或者一个楼层中很多计算机服务的那部分电缆。在段中，消息不需要路由，因为网段中的计算机都有直接的连接。整个系统的带宽由连接在网段范围内的计算机共享。大一些的局域网，比如校园网或者办公楼中的网络，由许多网段组成，段之间通过交换机或集线器互连（详见3.3.7节）。对于局域网来说，除了消息流量很大的时候，系统总带宽很高，而延迟时间很短。 70

20世纪70年代，人们开发了多种局域网技术——以太网、令牌环和有槽环形网，这些技术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的主导技术。它产生于20世纪70年代的早期，当时的带宽是10Mbps（每秒100万比特），最近扩展为100Mbps和1000Mbps（每秒1G比特）。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，它可以在几乎所有的工作环境中工作，只需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式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延迟和带宽保证，但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。而高速以太网采用交换模式加以部署，在很大程度上克服了上述缺点，虽然它的有效性不如ATM网络。

广域网　广域网（Wide Area Network, WAN）在属于不同组织以及可能被远距离分隔开的节点之间以较低速度传递消息。这些节点可能分布在不同的城市、国家甚至不同的洲。其通信介质是连接专用计算机（称为路由器）的通信电路。路由器管理整个通信网络，并将消息或数据包路由到指定的地点。在大多数的网络中，路由操作在每个路由点都引进了一定的延迟，因此消息传送总的延迟取决于消息经过的路由和消息经过的网络段的流量负载。在如今的网络中，这些延迟可能达到0.1～0.5s。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延迟设置了一个下限。举例来说，一个信号从欧洲到澳大利亚通过陆路连接的传播时间大约是0.13s，而地球表面上任意两个点之间经过地球同步卫星传输的信号有大约0.20s的延迟。

因特网上可用的带宽也变化很大。在部分因特网上速度可以达到600Mbps，但通常情况下，传输大量数据的速度还是1～10Mbps。

城域网　城域网（Metropolitan Area Network, MAN）基于城镇或城市里高带宽的铜线和光纤

电缆，在50km的范围内传输视频、音频或者其他数据。人们已经使用了多种技术来实现在MAN中的数据的路由，例如，从以太网到ATM。

以目前在许多城市可用的DSL（数字用户线）和电缆调制解调器连接为例。DSL通常使用电话交换系统中的ATM交换机（参见3.5.4节）（在已有的用于电话连接的电线上用高频信号）将双绞线上的数字信号以大约0.25～8.0Mbps的速度路由到用户的家或办公室中。因为DSL用户连接使用的
71 是双绞线，所以限制用户和交换机的距离要在5.5km之内。电缆调制解调器连接是在同轴电缆架构的有线电视网络上使用模拟信号传输，速度可以达到1.5Mbps，其范围大大地超过了DSL。

无线局域网 无线局域网（Wireless Local Area Network, WLAN）用于替代有线LAN，为移动设备提供连接，或者说，使得家里和办公楼内的计算机不需要有线的基础设施就能相互连接并连到因特网上。它们都是广泛使用的IEEE 802.11标准（WiFi）的变体，在1.5km范围内提供10～100Mbps的带宽。3.5.2节将给出这些方法的详细介绍。

无线城域网 IEEE 802.16 WiMAX标准针对这类网络。无线城域网（Wireless Metropolitan Area Network, WMAN）旨在替换家庭和办公楼中的有线连接，并在某些应用中超越802.11 WiFi网络。

无线广域网 无线广域网（Wireless Wide Area Network, WWAN）大部分移动电话网络基于数字无线网络技术，如世界上大部分国家采用的GSM（全球移动通信系统）标准。移动电话网络通过使用蜂窝无线连接可在广阔的地域（通常是整个国家或整个大洲）上运行，它们的数据传输设施为便携设备提供了到因特网的广域移动连接。上述蜂窝网络提供的数据传输率相对较低，只有9.6～33kbps，而“第三代”移动电话网络在几公里的蜂窝半径内提供128～384kbps的数据传输率，在更小的蜂窝半径内提供至多2Mbps的数据传输率。对移动和无线网络领域快速发展的技术感兴趣的读者可参考Stojmenovic的手册[2002]。

互连网络 互连网络是一个通信子系统，它将多个网络连接起来提供公共数据通信设施，这些数据通信设施覆盖了单个网络中的技术和协议以及用于互连的方法。

开发可扩展、开放的分布式系统，需要用到互连网络。分布式系统的开放性特征意指分布式系统所使用的网络应该是一个可扩展到含有大量计算机的网络，而单个网络的地址空间有限，且一些网络有性能限制，都不宜于大规模地使用。在互连网络中，可将众多的局域网和广域网技术集成起来为各类用户提供连网能力。这样，互连网络给分布式系统的通信提供了很多开放系统所具有的好处。

互连网络是由多种网络组建而成的。它们的互连依靠称为*路由器*的专用计算机和称为*网关*的通用计算机，集成通信子系统由软件层实现，它为互连网络的计算机提供寻址以及数据传输功能。可以把互连网络想象成一个“虚拟网络”，它是在由底层网络、路由器、网关组成的通信介质上覆盖一个互连网络层而构造出来的。因特网是网际互连的一个主要的例子，它所使用的TCP/IP协议
72 是上面提到的集成层的一个例子。

网络错误 图3-1的比较没有提到的一点是不同网络中会发生的故障频率和类型。除了在无线网络中数据包经常会因为外部干扰而丢失之外，其他各种网络的底层数据传输介质的可靠性都很高。但在所有网络中，都会由于处理延迟、交换机缓冲区溢出或者目的节点缓冲区溢出而引起数据包丢失，而这也是迄今为止数据包丢失最常见的原因。

数据包到达的顺序可以与发送的顺序不一样，这种情况只出现在对分离的数据包可以单独路由的网络——主要是广域网中。如果发送方假设以前发送的数据包丢失了，那么可以发送数据包的拷贝。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。

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。它使得发送到多个地址的消息可以共享同一条通信链接，这不同于常规电话所采用的电路交换技术。当链接可用时，数据包按顺

序排列在缓冲区中，然后发送。通信是异步的——消息经过一段延迟到达目的地，该延迟取决于数据包在网络中传递所花费的时间。

3.3.1 数据包的传输

计算机网络的大多数应用需求是按逻辑单元发送信息或消息——任意长度的数据串。在消息传递前，它被分割成数据包。形式最简单的数据包是长度有限的二进制数据序列（比特或字节数组）以及识别源和目的地计算机的寻址信息。使用长度有限的数据包是为了：

- 网络中的每台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。
- 避免长消息不加分割地传递所引起的为等待通信通道空闲而出现的过度延迟。

3.3.2 数据流

我们在第2章中曾提到，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内的延迟。这样的流和数据包传输所针对的基于消息的流量类型有本质上的不同。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。 73

为了达到实时显示的目的，如果传输的是压缩的数据，则视频流的传输需要1.5Mbps的带宽；如果传输的是未压缩的数据，则需要大约120Mbps的带宽。另外，和典型的客户－服务器交互程序所产生的断断续续的数据流量相反，这种流是连续的。多媒体元素的播放时间是必须被显示的时间（对视频元素来说）或必须转成音频的时间（对声音采样而言）。举例来说，视频帧的流速是每秒24个帧，那么第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，将被接收进程丢弃。

及时传输这种数据流依赖于具有一定服务质量（带宽、延迟和可靠性必须都有保证）的网络连接。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，在经过的节点上保留需要的资源，在通道中对任何不规则的数据流进行适当的缓冲。通过这个通道，数据可在要求的速率下从发送方传送到接收方。

ATM网络（参见3.5.4节）专门设计为提供高带宽和低延迟，并通过保留网络资源保证服务质量。IPv6（因特网新的网络协议，其描述见3.4.4节）的一个特色是实时流中的每一个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（Resource Reservation Protocol, 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（Real Time Transport Protocol, RTP）[Schulzrinne et al. 1996]是一个应用级数据传输协议，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作出实质性的改变。第17章将详细讨论分布式多媒体应用的需求。

3.3.3 交换模式

网络是一组由电路连接起来的节点组成的。为了能在任意两个节点间传输信息，交换系统是必不可少的。这里我们定义在计算机网络中使用的四种交换。

广播 广播是一种不涉及交换的传输技术。任何信息都将被传给每一个节点，由接收方判断是否接收。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，广播只能到达蜂窝内的节点。

电路交换 电话网曾经是唯一的电信网。它们的操作非常容易理解：当主叫方拨号时，主叫方电话到本地电话交换台的线路会通过自动交换机连接到被叫方的电话线。长途电话的拨叫过程也 74
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（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领域带来了新的契机。从根本上说，它使得人们可以处理和存储数据，这使得以完全不同的方式构造通信网络成为可能。这种新的通信网络叫做*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。在每个交换节点上（也就是几个电路需要互连的交汇处）有一台计算机。数据包到达一个节点后先存储在这个节点的内存中，再由一个程序选择数据包的外出电路，将它们转发到下一个离它们目的地更近的节点。

这里没有什么全新的内容，邮政系统就是一个信件的存储转发网络，其处理由人或机器在信件分拣室完成。而在计算机网络中，数据包的存储和处理很快，即使数据包路由了许多节点，也能给人们瞬间传输的假象。

帧中继 现实中，存储转发网络中每个节点转发一个数据包需要的时间从几十微秒到几ms不等，这个交换延迟取决于数据包的大小、硬件的速度和当时的流量情况，但它的下限由网络带宽决定，因为整个数据包必须在它转发给另一个节点之前先收到。数据包在到达目的地址前，可能要通过很多的节点。因特网中大多数据包基于存储转发交换，正如我们已经知道的，即使是很小的因特网数据包通常也需要200ms左右的时间到达目的地。这个量级的延迟对于电话会议、视频会议这样的实时应用而言就太长了，要维持高质量的会谈，延迟不得超过50ms。

*帧中继*交换方法给包交换网络引入了电路交换的一些优势。它们通过很快地交换小的数据包（称为帧）来解决延迟的问题。交换节点（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节点中，而是以位流的形式通过节点。ATM网络是一个最好的例子，我们将在3.5.4节描述它们的操作。高速ATM网络在由很多节点组成的网络中传递数据包只需要几十微秒。

3.3.4 协议

*协议*是指为了完成给定任务，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重要的部分：

- 必须交换的消息的顺序的规约。
- 消息中数据格式的规约。

75

众所周知的协议的存在使得分布式系统的软件组件能独立地开发，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。

一个协议是由分别位于发送方计算机和接收方计算机上的一对软件模块实现的。例如，一个*传输协议*将任意长度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。接着传输软件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）将消息传输到目的地。接收方计算机中相应的传输协议模块通过网络级协议模块接收这些数据包，并在传递给接收进程之前，进行逆向转换，重新生成消息。

协议层 网络软件是按层的层次结构排列的。每一层都为上面的层提供了相应的接口，并扩展了下层通信系统的性质。层由与网络相连的每一个计算机上的一个模块表示。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网络中另一个计算机上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协议模块之间直接传输。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。

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，并在将其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，在图中，该过程被应用于OSI协议组的前四层。从图中可以看出，数据包的头部包含大部分与网络相关的数据项，但为了简洁起见，它省略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应用层消息的长度小于底层网络数据包的最大长度。否则，消息就要被封装成几个网络层的数据包。在接收方，下层接收到的数据项要进行一次相反的转换，再传递到上一层。上层协议的类型已经包括在了每层的头部，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。

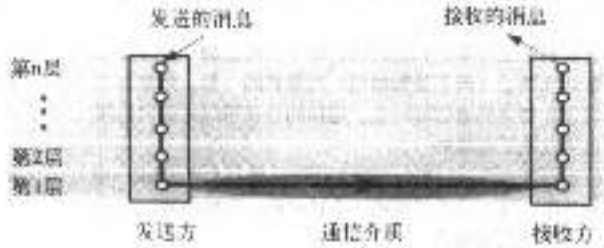


图3-2 协议软件中层的概念



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、光缆、卫星通信信道或无线电传输）和在发送节点将信号放置在通信介质上，在接收节点感应该信号的模拟信号电路实现的。在接收节点，接收到的数据项通过软件模块的层次结构向上传送，在每一层都重新转换直到变成可传递给接收进程的格式为止。

协议组 一套完整的协议层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（Open System Interconnection, OSI）的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。

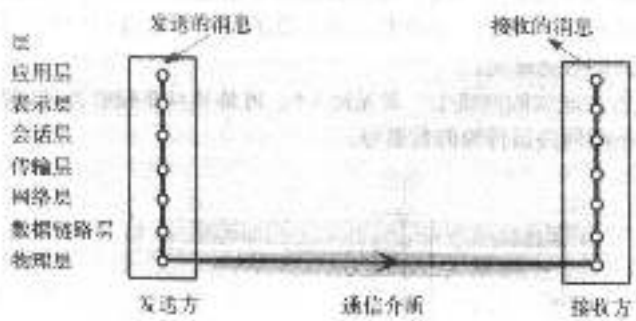


图3-4 ISO开放系统互连（OSI）协议模型中的协议层

图3-5总结了OSI参考模型的每一层的目标。顾名思义，这只是一个用于协议定义的框架，而不是特定协议组的定义。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。

层	描述	例子
应用层	这层协议是为满足特定应用的通信需求而设计的，通常定义了一个服务接口	HTTP、FTP、STMP、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，两种表示可能完全不同。如果需要，可以在这一层对数据进行加密	TLS安全、CORBA数据表示
会话层	在这层被实现可靠性和适应性，比如故障检测和自动恢复	SIP
传输层	这是处理消息（而不是数据包）的最低的一层。消息被定位到与进程相连的通信端口上。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	在特定网络中的计算机间传输数据包。在一个WAN或一个互连网络中，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节点间传输数据包。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的	Ethernet MAC、ATM信元传送、PPP
物理层	指驱动网络的电路和硬件。它通过发送模拟信号传输二进制数据序列。同电信号的振幅或频率调制信号（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

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，同时也带来了极大的性能开销。通过N层协议栈传输一个应用级的消息，通常在协议组中要进行N次控制传输，才能到达相关的软件层，其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低于可用的网络带宽。

图3-5包括了在因特网中使用的协议的例子，但因特网的实现在两方面没有遵循OSI模型。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。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单独实现。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20章）。Web浏览器和其他的一些需要安全信道的程序也以相似的方法，即采用过程库方式的安全套接字层（见第7章）。

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

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

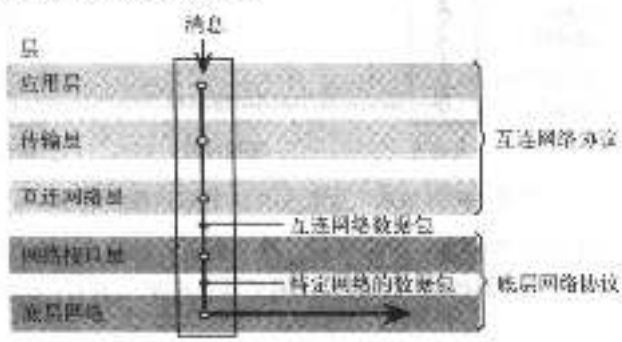


图3-6 互连网络层

数据包组装 在传输前将消息分割成多个数据包并在接收端重新组装各个数据包的任务通常是由传输层完成。

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，数据域是长度可变的，其最大长度称为*最大传输单元*（Maximum Transfer Unit, MTU）。如果消息的长度超过底层网络层的MTU，就将其分割为多个大小适当的块，并标上序列号以便其重新装配，再用多个数据包进行传输。例如，以太网的MTU是1500字节，如果消息不超过这个数据量，就能在一个以太网数据包中进行传输。

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（实际中通常使用8KB，因为一些节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值大的数据包必须经过分割才能在以太网上传输。

76 ~ 79

端口 传输层的任务是在一对网络*端口*间提供与网络无关的消息传送服务。端口是主机中可由软件定义的目的点。它隶属于进程，使得数据能传输到位于目的节点的指定进程。这里我们将详细讲述端口在因特网和大部分其他网络中实现的端口寻址过程。第4章将讨论端口的编程。

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。网络地址是能唯一标识主机的一个数字标识符，可以让负责将数据路由到该主机的节点准确地定位它。在因特网中，为每台主机都分配了一个IP地址，用于标识该主机和它连入的子网，使得从其他节点都能路由到该主机（下一节将介绍这一内容）。以太网中没有路由节点，由每台主机负责辨识数据包的地址，并接收发给自己的数据包。

众所周知的因特网服务（如HTTP或FTP）已经被分配了*关联的端口号*。它们都在权威机构（即因特网编号管理局，简称IANA）进行了登记[www.iana.org]。要访问指定主机上的某个服务，只要将请求发给该主机上相关的端口就可以了。有些服务，如FTP（关联端口为21），会被分配一个新的端口号（私有号码），并将新的端口号发送到客户端。客户端使用新的端口号完成交易或会话的剩余部分。其他服务，如HTTP（关联端口为80），通过关联端口处理所有的业务活动。

编号小于1023的端口被定义为公共端口。在大多数操作系统中，它们的使用被限制在特权进程中。1024~49151之间的端口是IANA拥有的服务描述的已注册端口，其他直到65535的端口可用于个人目的。实际上，大于1023的所有端口都可用于个人目的，只是为个人目的使用这些端口的计算机不能同时访问相应的已注册服务。

在开发经常包括许多动态分配的服务器的分布式系统中，分配固定端口号并不恰当。这个问题的解决方案涉及动态为服务分配端口以及提供绑定机制，使得客户能用符号化名字定位服务和相应的端口。这些将在第5章做进一步讨论。

数据包传递 网络层采用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、电报的传输模式的相似性。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，网络就不再保存它的相关信息。在数据报网络中，从一个源地址到一个目的地址的数据包序列可以按照不同的路由来传递（这样，网络就有能力处理故障，或缓解局部拥塞带来的影响），在这种情况下，数据包序列可能不按照原来的顺序到达。

每个数据报包都包括完整的源主机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最初所基于的概念，可以在目前使用的大多数计算机网络中找到它。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基于数据报传递的。

80

虚电路包传递：一些网络级的服务利用类似于电话网络中传递的方式实现包传输。必须在经源主机A到目的主机B传递包之前建立虚电路。要建立虚电路，涉及确定从源地址到目的地址的路

由，这可能会经过一些中间节点。在路由中的每个节点上都会有一个表格项，指示路由下一步该使用哪一个链接。

一旦建立起虚电路，就可以用它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，而不是源和目的地址。此时已不需要地址信息，因为在中间节点，通过引用虚电路号来路由数据包。数据包到达目的地址后，根据虚电路号就可以决定其源地址。

虚电路与电话网络的类比不能这样从表面上看，在POTS中，进行一次电话呼叫就要建立从主叫者到被叫者的物理电路，而这一音频链接也将作为专用连接而被保留。在虚电路的包传递中，电路只是由一些在路由节点上的表格项来表示，而数据包所路经的链接也只在传递一个数据包时使用，在其余时间这些链接是空闲的，可供它用。因此，一个链接可以被多个独立的虚电路使用。目前使用的最重要的虚电路网络技术是ATM。我们已经提到过（见3.3.3节），它传送单个数据包的延迟较短，这是使用虚电路的直接结果。但无论怎么说，数据包传送到一个新目的地址前要求有一个准备阶段确实造成了短时间的延迟。

不要将网络层的数据报传递和虚电路包传递之间的区别和传输层中名字相似的机制（即无连接传输和面向连接传输）混淆。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）——的内容中描述这些技术。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

路由是除了局域网以外，比如以太网（局域网在所有相连的主机间两两都有直接连接），其他网络都需要的功能。在大型网络中，采用的是自适应路由，即网络两点间通信的最佳路由会周期性地重新评估，评估时会考虑到当时的网络流量以及故障情况（如路由器故障或网络断链）。

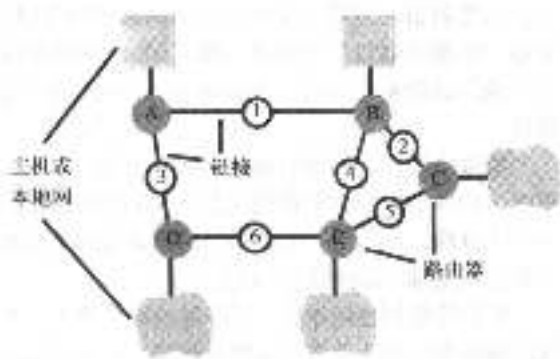


图3-7 广域网中的路由

如图3-7所示，在网络中将数据包传递到目的地址是处于连接点的路由器的共同责任，除非源主机和目的主机都在同一个局域网中，否则数据包都必须经过一个或多个的路由节点，辗转多次才能到达。而决定数据包传输到目的地址的路由是由路由算法负责的——它由每个节点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穿越下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，一旦建立虚电路或连接，路由也就确定了。在包交换网络层（如IP）中，数据包的路由是单独决定的，如果希望不降低网络性能，算法必须特别简单有效。

2）它必须通过监控流量和检测配置变化或故障来动态地更新网络的知识。在这种活动中，时间并不是至关重要的，可以使用速度较慢但计算量较大的技术。

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进入的数据包下一步的方向。本地拥有的路由信息依靠一个分发链路状态信息（它们的负载和故障状态）的算法定期更新。

一个简单的路由算法 我们在这里描述的是“距离向量”算法。这将为3.4.3节中讨论链路－状态算法提供基础，而链路－状态算法从1979以来就成为因特网上主要的路由算法。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]早在计算机网络出

现之前就发表了，它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，而基于他们的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。路由表的每行为发送给定目的地址的数据包提供了路由信息。*链路*域为发送到指定目的地的数据包指明了下一段链路。*开销*域计算向量距离，或到达目的地的跳数。对于具有相似带宽的链路的存储转发网络，这张表对一个数据包传输到目的地所需的时间给出了合理估计。存储在路由表中的开销信息并不是路由算法的第1部分所采取的包路由动作中使用的，而是在算法的第2部分建立和维护路由表时使用。 81 ~ 82

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所示网络的路由表

路由表中为每个可能的目的地单独设置一项，给出了数据包到达目的地而要采取的下一跳(hop)。当数据包到达一个路由器时，就会抽取目的地址并在本地路由表中查找该地址。路由表中的表项给出了指引数据包发送到目的要经过的下一个链路。

例如，一个目的地为*C*的数据包从路由器*A*开始发送，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*沿标号为1的链路路由。数据包到达*B*后，按照前述的过程，在*B*的路由表中查询，发现需要经过标号为2的链路路由到*C*。当数据包到达*C*时，路由表中的相关项显示“本地”，而不是一个链路号。这表明应该将数据包发送到本地主机上去。

现在让我们来考虑一下怎样建立路由表，以及在网络发生故障时怎样维护路由表，即上面所说的路由算法的第2部分是怎样完成的。因为每个路由表只为每个路由指定一跳，所以路由信息的构建或修正就可以按分布的方式进行。每个路由器使用*路由器信息协议*（Router Information Protocol, RIP）通过发送自己路由表信息的概要和邻接节点相互交换网络信息。下面简要描述一下路由器所完成的RIP动作： 83

1）*周期性地并且只要本地路由表发生改变*，就将自己的路由表（以概要的方式）发给邻接的所有可访问的路由器。也就是说，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更好（开销更低）的路由，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，并且表中给出的从链路*n*开始到达某地的开销和本地路由表中的不相同，则用新的开销替换本地表中已有的开销。这样做的原因是，新表是从和相关的目的地更近的路由器传来的，因此对经过该路由器的路由而言更加有权威性。

图3-9给出的伪代码程序将更准确地描述这个算法，其中*Tr*是从另一个路由器接收到的表，*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，上面描述的步骤都能充分确保路由表收敛到到达每个目的地的最佳路由。即使网络没有发生变化，也会以频率*t*来传播路由表，以确保其稳定性，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。

```
Send：每隔t秒或在Tl发生变化时，在每个没有故障的链路上发送Tl。
Receive：当在链路n上接收到路由表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：远程节点有更好的路由
       // Rl.link = n：远程节点更加权威
   }
  }
}
```

图3-9 RIP路由算法

为了处理故障，每个路由器都监控着自己的链路并做以下的工作：

当检测到一条有故障的链路n时，将本地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。

84

这样，一个断开的链路信息被表示成通往相关目的地的开销值是无穷大。当这一信息传播到邻接路由器时，它们的路由表也将根据Receive动作进行更新（注意：∞ + 1 = ∞），然后继续传播，直到到达了有路由到相关目的地的节点（如果存在这样的节点）。最终，具有可用路由的节点会传播它的路由表，它的可用路由也将代替所有节点中的故障路由。

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*，可以根据链路的实际带宽来计算；可以修改算法，以增加信息收敛的速度，并避免那些在达到收敛前可能出现的不希望出现的中间状态，比如循环。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]。但收敛速度过慢所带来的问题并没有得到很好的解决，当网络处于中间状态时就会出现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节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*链路－状态算法*。它们的基本思想是分布并更新在每个节点中一个表示网络所有部分或重要部分的数据库。每个节点负责计算在自己的数据库中所显示的到达目的地的最佳路由。这种计算可利用多种算法完成，有些算法避免了Bellman-Ford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主题，我们这里的讨论是非常有限的。我们将在3.4.3节重新讨论这个主题，在那里将描述RIP-1算法的操作，RIP-1算法是最早用于IP路由的算法之一，目前在因特网的许多地方还在使用它。对于因特网中更深入的路由问题，请参阅Huitema [2000]，如想全面地了解路由算法，请参阅Tanenbaum[2003]。

3.3.6 拥塞控制

网络的能力受到通信链路性能和交换节点性能的限制。当任何链路或节点的负载接近其负载能力时，试图发送数据包的主机中就会建立队列，传输数据的中间节点因为被其他数据传输所阻

塞也会建立队列。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，直到达到可用的缓冲区空间的上限为止。

一旦节点达到这样的状态，节点只能将以后到达的数据包丢弃。前面已经提到过，在网络层偶尔出现数据包丢失是允许的，这种损失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率达到一个很高的水平，那么会给网络的吞吐量带来灾难性的后果。道理很简单：如果数据包在中间节点被丢弃，那么已经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。经验表明，当网络的负载超过其能力的80%，系统的总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。 85

为了避免数据包在网络中传递时经过拥塞节点而被丢弃的情况，最好将数据包保存在发生拥塞之前的节点中直到拥塞减少。这固然会增加数据包的延迟，但不会极大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由上的节点而实现的，因此它们的数据包传输率会有所减少。对中间节点来说，这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，或者阻塞产生这些数据包的应用程序，直到网络能妥善地处理数据包为止。

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。也就是说，发送节点必须基于收到的接收方的信息降低其发出数据包的速率。要为发送节点提供拥塞信息，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*），也可以通过实现一个专门的传输控制协议（TCP的名字也由此而来，3.4.6节将解释TCP中的机制），或通过观察丢弃数据包发生的情况（假设协议要确认每一个数据包）来实现。

在一些基于虚电路的网络中，每个节点可以接收到拥塞信息，拥塞信息也可以作用于每个节点。尽管ATM使用虚电路传递，但它仍要依靠服务质量管理（见3.5.3节和第17章）来保证每个电路都能完成所要求的流量。

3.3.7 网际互连

不同的网络、链路和物理层协议形成了不同的网络技术。局域网络是基于以太网和ATM技术建立起来的，而广域网是基于各种数字和模拟电话网络、卫星连接和广域ATM网络建立的。单个的计算机和局域网则是通过调制解调器、无线连接和DSL连接接入因特网或企业内部网的。

为了建立一个集成的网络（*互连网络*），我们必须集成许多子网，而它们各自基于上述某种网络技术。为了实现集成，需要实现以下几方面：

1）统一的互连网络寻址方案，使得数据包可以找到接入任一子网的任一主机。

2）定义互连网络中的数据包格式并给出相应处理规则的协议。

3）互连组件，用于按照互连网络地址将数据包路由到目的地，可用具有多种网络技术的子网传递数据包。

对于因特网而言，IP地址可实现上面第1个要求，第2个要求是IP协议，第3个要求由称为*因特网路由器*的组件实现。IP协议和IP寻址将在3.4节详细描述。这里我们将讨论因特网路由器和其他用来连接各网络的组件的功能。 86

图3-10展示了2000年年中伦敦大学Queen Mary学院（QMUL）的企业内部网的一小部分，更多细节将在后面的小节中加以解释。这里我们要注意的是图中包含通过路由器互连的多个子网那一部分。该部分有5个子网，其中3个子网共享IP网络138.37.95（使用了无等级的域间路由方案，见3.4.3节）。图上的数字是IP地址，它们的结构将在3.4.1节中解释。图上的路由器是多个子网的成员，它们在每个子网中都有一个IP地址（地址就写在连接的链路上）。

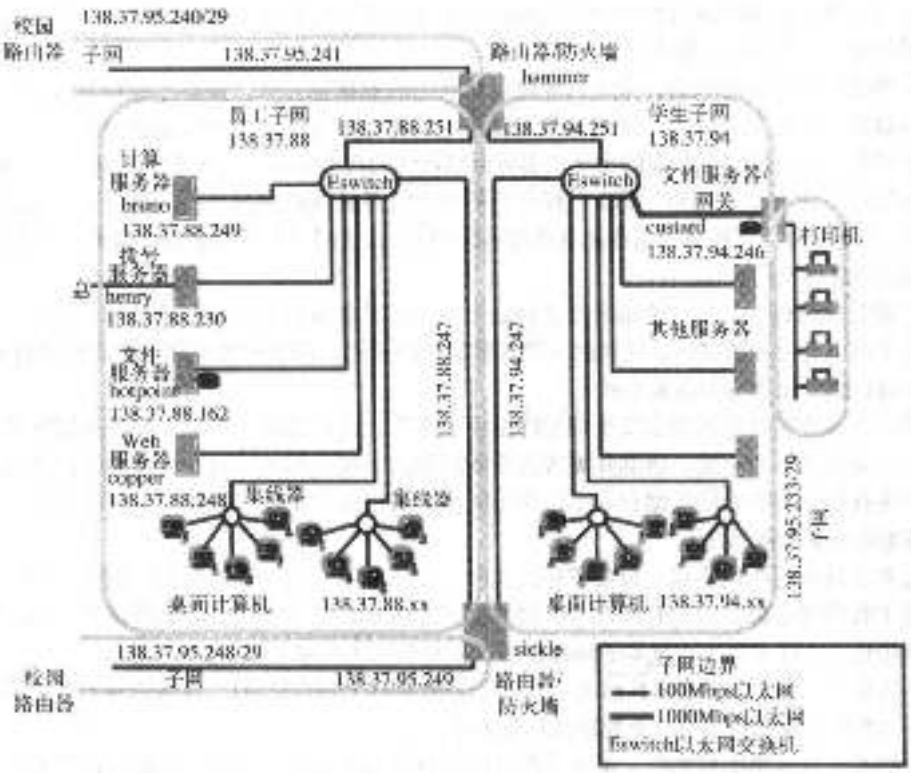


图3-10 QMUL计算机科学网的简图（2000年年中）

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，其中一个任务是作为防火墙使用。防火墙的作用和路由功能紧密相关的，我们将在下面讨论这一点。

87 138.37.95.232/29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，该服务器在相连的打印机上通过一个监控和控制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。大部分链路的带宽是100Mbps，但有一个链路的带宽是1000Mbps。因为它支持着大量学生使用的计算机和包含所有文件的文件服务器custard间的巨大数据流量。

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集线器。两者对IP数据包来说都是透明的，以太网集线器只是一种将以太网电缆的多个段连接在一起的手段。在网络协议层，这些段形成一个以太网。主机收到的所有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，除了像以太网和无线网络（这些网络中的主机由一种传输介质连接），其他所有网络都需要路由。图3-7显示了一个由6条链路连接5个路由器组成的网络。在一个互连网络中，可由直接连接将路由器链接起来，如图3-7所示。也可以通过子网将路由器互连，如图3-10中的custard。在这两种情况下，路由器都负责将从任一连接来的互连网络数据包准确地发送到下一条连接。路由器也因为这个目的而维护路由表。

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，因为它们也表现出了路由的功能。例如，QMW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的。

集线器 集线器是将主机、以太网和其他广播型局域网技术的扩展网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。它们也用于克服单

个网段带来的距离上的限制，提供添加额外主机的途径。

交换机 交换机的功能与路由器相似，但路由器只用于局域网（一般是以太网）。也就是说，它们将多个分离的以太网互连，将到达的数据包路由到适当的外出网络中。它们在以太网的网络协议层上完成这一任务。起初它们对互连网络有多大范围一无所知，通过观察数据流量以及在缺少信息时采取广播请求的方式建立其路由表。

与集线器相比，交换机的好处是它分离了到达的流量，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 网桥和路由器通过网络层协议和一个互连网络协议的转换，实现在各种底层网络上传输互连网络数据包，不过在一种情形下，底层网络协议可以被隐藏起来不被其上的层看到，不需要使用互连网络协议。当一对连接到同一类型的两个网络中的节点需要通过另一种类型的网络进行 88
通信时，它们之间通过构造协议"隧道"来达到这一目标。协议隧道其实就是在相异网络环境中传输数据包的软件层。

下面类比解释了选择"隧道"这一术语的原因，同时也提供了另一种方式来思考隧道的含义。穿山隧道使得车辆通过成为可能，如果没有隧道这是不可能实现的。公路是连续的，隧道对于应用（车辆）来说是透明的。公路是传输机制，而隧道使得它能在相异的环境中工作。

图3-11显示的是隧道的一种建议使用方法，它支持从因特网迁移到IPv6协议。IPv6将会取代现在使用的IP协议版本IPv4，但它们不兼容（IPv4和IPv6的描述见3.4节）。在向IPv6过渡的过程中，IPv4的海洋中会不断出现IPv6"岛屿"。在我们的图中，A和B就是这样的岛屿。在岛屿的边界处，IPv6数据包被封装成IPv4的格式，并以那种方式在IPv4网络中传输。

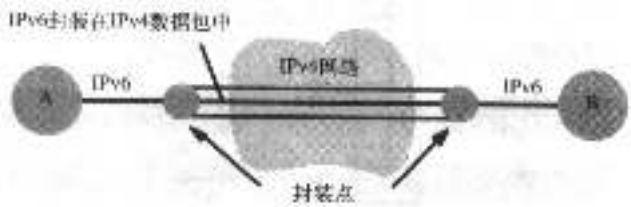


图3-11 IPv6迁移使用的隧道

看另一个例子，移动IP协议（其描述见3.4.5节）通过建立从本地基站到任一网络位置的隧道，来将IP数据包传输到因特网上的任何移动主机。中间的网络节点不需要为处理移动IP协议而加以修改。IP组播协议在处理方式上也与此相似，依靠一些支持IP组播路由的路由器来决定路由，但通过使用标准IP地址的路由器来传输IP数据包。另一个例子是在串行链路上传输IP数据包的PPP协议。

3.4 因特网协议

本节将介绍TCP/IP协议组的主要特点，并讨论在分布式系统中使用它们的好处及局限性。

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中的广泛采用以及最近在许多国家的商业网络中的广泛使用，使得全国的网络可以集成为一个互连网络， 89
这一网络已经迅速发展到目前数量超过6千万主机的规模。现在许多应用服务和应用层的协议（列在下面的各个括号内）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）和远程登录（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，以提供额外的特点。例如，通常HTTP传输时直接使用TCP，但当需要端－端的安全性时，传输层安全（TLS）协议（在7.6.3节讨论）就会放在TCP的上

层，以建立安全信道，HTTP消息通过这一安全信道传输。

最初，开发因特网协议是用来支持一些简单的广域应用，如文件传输和电子邮件，这涉及在地理上相隔很远的有较长延迟的通信。但这些协议已被证明足以有效支持很多分布式应用的需求，不论这些应用是在广域网上还是在局域网上，它们现在广泛使用于分布式系统中。通信协议的标准化带来了巨大的好处。

图3-6所示的互连网络协议层的一般性说明被翻译成图3-12所示的因特网的特例，其中有两个传输协议——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。网际协议（IP）是因特网虚拟网络的底层"网络"协议，也就是说，IP数据报为因特网和其他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，因为它并不是唯一的因特网通信实现所涉及的网络层。这是因为网际协议通常是在另一个网络技术之上，比如以太网，它已经提供了一个网络层，该层使得连接到同一网络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。头部的标签给出了上层协议的类型，以便接收协议栈正确地解开这个数据包。在TCP层，接收方的端口号有类似的作用，使得接收主机的TCP软件组件可以将消息送到特定的应用层进程中去。

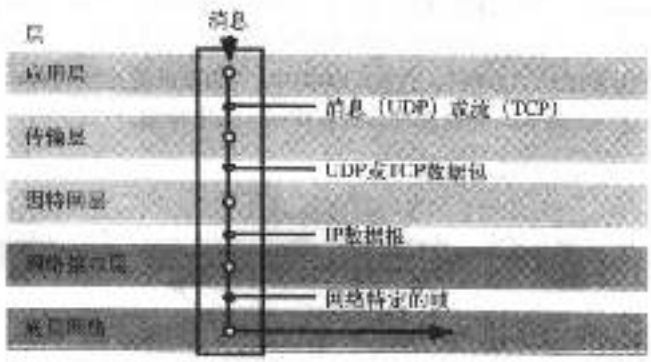


图3-12 TCP/IP层

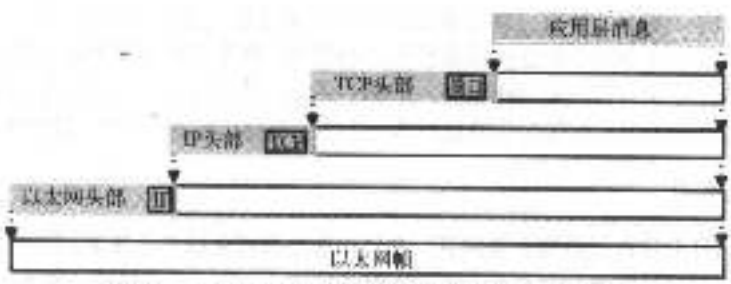


图3-13 通过TCP在以太网上传输消息时发生的封装

TCP/IP规约[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APPANET上，这个网络包括主机和一些由长距离数据链路连接的早期版本的路由器（称为PSE）。如今，IP实际上已经用于各种网络技术了，包括ATM、局域网（如以太网）和令牌环网。在串行线路和电话电路上通过PPP协议[Parker 1992]实现IP，使得IP可用于与调制解调器连接和其他串行链路的通信。

TCP/IP的成功源于它独立于底层传输技术，这使得互连网络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，它隐藏了底层传输介质的多样性。图3-14说明了这个观点。

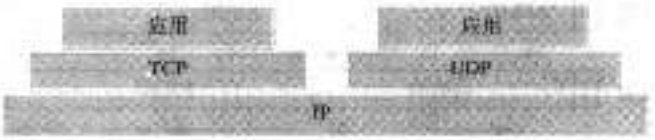


图3-14 编程者眼中TCP/IP因特网的概念

下面两节将详细描述IP寻址方案和IP协议。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地址的域名系统将在3.4.7节中介绍，第9章将给出更全面的叙述。

90
-
91

现在，因特网上使用的主要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小节里讨论的版本。但由于因特网使用的飞速发展，人们也不得不发布新的IP版本IPv6，以克服IPv4中地址数量的限制并为之增添功能以满足新的需求。我们将在3.4.4节描述IPv6。由于大量的软件将受此影响，所以逐渐过渡到IPv6的计划将在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富有挑战之处是构造主机的命名和寻址方案以及将IP数据包路由到目的地的方案。分配主机网络地址的方案和计算机连接到它们的方案需要满足以下一些需求：

- 这必须是通用的——任何主机必须可以发送数据包给因特网中的任何其他主机。
- 地址空间的使用，必须是有效的——预知因特网的最终规模、网络数量和所需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，当开发TCP/IP协议时，认为提供2^{32}（即约40亿，大致等于当时全世界的人口总数）的可寻址的主机就足够了。但这种判断已经被证明是目光短浅的，原因如下：

—因特网的增长速度远远超过了当初的预测。

—地址空间的分配和使用比预期的要低效得多。

- 寻址方案必须有助于开发灵活有效的路由方案，但地址本身并不能包括太多的用于将数据包路由到目的地的信息。

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比特的数字标识符，其中包括一个网络标识符（唯一标识了因特网中的某个子网）、一个主机标识符（唯一标识了到该网络的主机连接）。这些地址将放在IP数据包中并被路由到目的地。

因特网地址空间所采用的设计如图3-15所示。一共有4类已分配的因特网地址——A、B、C、D。D类地址为因特网组播通信保留，组播通信仅在一些因特网路由器中实现，其进一步的讨论见4.5.1节。E类地址包括一些未分配的地址，为满足未来的需求而保留。

这些包含网络标识符和主机标识符的32比特因特网地址通常写成由点分开的4个十进制数字序列。每个十进制数字表示一个字节或IP地址的8比特组。每一类网络地址的允许值如图3-16所示。

三类地址用于满足不同类型组织的需要。A类地址（在每个子网中能容纳2^{24}台主机）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。B类地址可分配给网络中的计算机超过255台的组织，而C类地址则是分配给所有其他的网络。

主机标识符为0和全1（二进制）的因特网地址将留作特殊用途。主机标识符为0的地址代表"本机"，若主机标识符为全1，则表示这是一个广播消息，并将消息发送到与地址的网络标识符部分指定的网络连接的主机上。

网络标识符是由因特网编号管理局（IANA）分配给其网络与因特网相连的组织。连接到因特

网的计算机的主机标识符是由相关网络的管理员来分配的。



图3-15 因特网地址结构（域大小的单位是比特）



图3-16 十进制的因特网地址

既然主机的地址包括一个网络标识符，那么连接到多个网络的计算机必须在每个网络中都有独立的地址。每次计算机移到一个新的网络，它的因特网地址必须改变。这些需求导致了实质性的管理开销，在使用便携计算机的情况下就会有这种开销。

IP地址分配方案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不能很容易地预测出未来他们对主机地址需求的增长，一般都会过高地估计，从而选择B类地址，到了1990年前后，按照当时的IP地址分配速度，到1996年前后就可能用完所有的地址。当时采取了三个步骤。第一步是开始开发新的IP协议和寻址方案，结果也就是现在的IPv6。

第二步是从根本上修改IP地址的分配方案。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。该方案称为无等级域间路由（CIDR），我们将在3.4.3节中讨论CIDR。图3-10中的局域网拥有多个C类地址规模的子网，从138.37.88到138.37.95，这些子网通过路由器连接。路由器负责将IP数据包传送到所有的子网，同时也负责处理子网间和子网到因特网其他部分的流量。该图也说明了使用CIDR划分一个B类地址空间，形成若干C类地址规模的子网。

第三步是使未注册的计算机能通过实现了网络地址翻译（NAT）方案的路由器间接地访问因特网。我们在3.4.3节讨论该方案。

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路由器。完整的IP数据包格式是相当复杂的，图3-17给出了其主要组成部分，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。

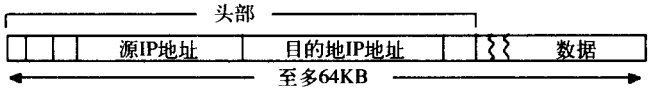


图3-17 IP数据包的布局

IP提供的传输服务被描述成有不可靠或尽力而为这样的传输语义，因为没有传输上的保证。数据包可能会丢失、重复、延迟或顺序错误，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中唯一的校验和是头部的校验和，其计算代价不高，还能确保检测到任何寻址和数据包管理数据中发生的错误。它没有提供数据的校验和，这避免了经过路由器时的开销，而是让更高层的协议（TCP和UDP）来提供它们自己的校验和——这是端对端争论中的一个实际例子（参见2.2.1节）。 92~94

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IP数据报分割成多个小的数据包，然后在目的地重新组装。数据包还可以进一步分割以适合从源地址到目的地址的路径中所经过的网络（每个数据包都有一个片断标识符，使得打乱顺序的各个段能够重新组合起来）

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比特的因特网地址转换成48比特的以太网地址。

这种转换是与网络技术相关的：

- 有一些主机直接与因特网数据包交换机相连，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，这样就可以方便地选择地址以匹配因特网地址中的主机标识符部分——翻译就是从IP地址中抽取主机标识符。
- 对于以太网和其他局域网，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，其具体实现是通过地址解析协议（ARP）完成的。

现在我们概述一下以太网中ARP的实现。为了能在计算机加入局域网时让IP数据包在以太网上传输，使用了动态询问并利用缓存来减少询问消息。先考虑同一个以太网中一个主机用IP向另一个主机传送消息的情况。发送方的IP软件模块在发送数据包前，必须将IP数据包中的接收方的因特网地址翻译成以太网地址。它调用发送方的ARP模块来完成这一任务。

每个主机上的ARP模块都维护一个缓存，保存它以前获得的（IP地址，以太网地址）对。如果需要的IP地址位于这个缓存中，请求就会立刻被应答。如果没有需要的IP地址，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），数据包中包括了所需的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IP地址和数据包中的IP地址进行匹配。如果匹配，就给ARP请求的发出方发送一个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。发出方的ARP模块在自己的本地（IP地址，以太网地址）缓存中加入新的IP地址→以太网地址映射，以后遇到相似的请求，它就不需要广播ARP请求了。一段时间之后，每个计算机上的ARP缓存中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。 95

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。遗憾

的是，并不能保证给定的源地址就是真正的发送方的地址。心怀叵测的发送者可以轻易地使用别的地址来代替它。这个漏洞已成为多起著名攻击的源头，包括1.4.3节提到的2000年2月出现的分布式拒绝服务攻击[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请求在它们的发送方地址域中都填上了目标计算机的IP地址，因此ping的应答就指向目标计算机，造成它们的输入缓冲溢出，造成合法的IP数据包无法通过。这种攻击将在第7章中进一步讨论。

3.4.3 IP 路由

IP层将数据包从源地址路由到目的地址。因特网上的每个路由器都实现了IP层的软件，用以提供一个路由算法。

主干 因特网的拓扑图在概念上被分割成*自治系统*（Autonomous System, AS），再被细分为*区域*。大多数大型机构（如大学和大公司）的企业内部网可看作AS，通常它们包含几个区域。在图3-10中，校园网是一个AS，图中显示的部分是一个区域。拓扑图上的每个AS都有一个*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构成了网络的主干。主干中的链路通常带宽很高，并且为保证可靠性，链路都被复制。这样的层次结构仅存在于概念中，主要用于管理资源与维护组件。它并不影响IP数据包的路由。

路由协议 RIP-1作为因特网上使用的第一个路由算法，是3.3.5节中描述的距离－向量算法的一个版本。RIP-2（见RFC 1388 [Malkin 1993]）由它发展而来，但包含了其他需求，如无类别域间路由、更好的组播路由以及认证RIP数据包以避免路由器受到攻击。

96 随着因特网规模的扩大，路由器的处理能力也不断增加，不再使用距离－向量算法已成为一个趋势，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－状态算法，这个算法被称为*开放最短路径优先*（Open Shortest Path First, OSPF）。该协议基于Dijkstra[1959]的路径寻找算法，它比RIP算法收敛得更快。

应当注意，在IP路由器中可以渐进地采纳新路由算法。路由算法的变化将导致新版本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。但是对于那些在更新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。在每个区域中使用一个路由协议，区域中的路由器在维护路由表时相互合作。只支持RIP-1的路由器依然很常见，它们利用新版本协议具有的向后兼容特性，与支持RIP-2和OSPF的路由器共存。

1993年，实际观测获得的数据[Floyd and Jacobson 1993]表明，RIP路由器的信息交换频率为30s，这会使IP传输性能产生周期性。IP数据包传输的平均延迟每隔30s就会出现一个尖峰。这可以追溯到执行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，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RIP动作。建议路由器采用15～45s范围内的随机值作为RIP的更新周期以进行纠正。

默认路由 到目前为止，我们对路由算法的讨论说明，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（子网或直接连接的主机）的路线。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100万，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所带来的后果，这两个方案同时被采纳。第一个方案是采用某种形式的IP地址拓扑分组。1993年以前无法从IP地址推断出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：

地址194.0.0.0到195.255.255.255在欧洲

地址198.0.0.0到199.255.255.255在北美地区

地址200.0.0.0到201.255.255.255在中南美地区

地址202.0.0.0到203.255.255.255在亚太地区

因为这些地理区域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范围在194.0.0.0到195.255.255.255的地址，可以只有一个表项。路由器将所有目的地在这个范围内的IP数据包使用相同的路由发送到最近的欧洲网关路由器上。注意，在做出这个决策之前，IP地址的 97
分配通常与拓扑或地理位置无关，目前这些地址的大部分仍在使用，1993年的决策无法减少这些地址对应路由表项的规模。

解决路由表大小爆炸性增长的第二个解决方案更简单而且非常有效。它基于下述观察结果，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的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址不在路由表中的IP数据包所使用的路由。为了说明这种情况，考虑图3-7与图3-8，假设节点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	—

节点C忽略了节点A与D。它将所有到达节点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的数据包可以到达其目的地，在路由过程中不会损失有效性，但目的地为A的数据包会增加一跳，需要通过E和B进行传输。总之，默认路由的使用在表格大小与路由有效性之间作出了折衷。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通过某一个点，此时不会损失有效性。默认路由方案在因特网路由中使用很广泛，因特网上没有一个路由器包含到达所有目的地的路由。

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符部分可获得底层网络上的目的主机的地址，只需一跳就能将数据包传送到目的地。IP层使用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，它必须将消息发送到一个本地路由器。它使用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，它们具有多个因特网地址，每个地址对应一个所连接的网络。

无类别域间路由（CIDR） 3.4.1节指出，IP地址的短缺导致1996年引入CIDR方案，该方案用于分配地址以及管理路由表中的项。主要问题在于B类地址不足，B类地址用于那些具有255个以上主机的子网，同时又有大量的C类地址可用。CIDR对这个问题的解决方案是给那些需要255个以上 98
地址的子网分配一批连续的C类地址。CIDR方案也允许将B类地址空间分割，以便把它分配给多个子网。

将C类地址分批似乎是一个简单的方法，但除非同时改变路由表的格式，它才会对路由表的大小产生显著的影响，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增加一个*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。这有效地使主机/子网地址成为IP地址的任意部分，比A类、B类与C类地址提供了更大的灵活性，*无类别域间路由*也因此得名。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，而其他路由器仍然使用旧

的基于类别的算法。

该方案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C类大小的子网地址对应的一个整数值范围。另一方面，有些子网也使用CIDR划分单个网络中的地址范围，这个网络可以是A类、B类或C类网络。如果一组子网完全由CIDR路由器与外部世界相连接，那么该组子网的IP地址范围可以成批分配到每个子网中，其中由任意大小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组的8地址空间。图3-10包含一个使用CIDR机制将138.37.95这样的C类地址规模的子网划分为多个组，每组包含8个主机地址，每个地址的路由不同。不同的组用138.37.95.232/29以及138.37.95.248/29等符号表示。这些地址中的/29表示附加一个32的掩码，前29位是1，后3位是0。

未注册的地址和网络地址翻译（NAT） 不是所有访问因特网的计算机和设备都需要分配全局唯一的IP地址。局域网中的计算机通过具有NAT功能的路由器访问因特网，它依靠路由器将到达的UDP和TCP包重定向。图3-18给出了一个典型的家庭网络，其中的计算机和其他网络设备通过一个具有NAT功能的路由器与因特网相连。网络包括能访问因特网的计算机，它们通过有线以太网连接到路由器，还包括通过WiFi接入点连接的设备。为了保证完整性，图中给出了一些具有蓝牙功能的设备，但它们不是与路由器连接，因此不能直接访问因特网。家庭网络具有由因特网服务提供商分配的一个已注册的IP地址（83.215.152.95）。这里描述的方法适合任何希望其没有注册IP地址的计算机连接到因特网的组织。

家庭网络上所有能访问因特网的设备都被分配了192.168.1.x C类子网上的一个未注册的IP地址。大多数的内部计算机和设备由路由器上运行的动态主机配置协议（Dynamic Host Configuration Protocol, DHCP）动态分配一个IP地址。在图中，192.168.1.100以上的数字由DHCP服务使用，数字较小的节点（例如PC 1）已经以手工方式分配了数字，这样做的理由将在后面解
99 释。虽然NAT路由器使得这些地址对因特网的其他部分完全隐藏，但通常使用IANA为私有互连网保留的三块地址（10.z.y.x, 172.16.y.x或192.168.y.x）之一中的一段。

NAT的介绍见RFC 1631[Egevang and Francis 1994]，它的扩展见RFC 2663[Srisuresh and Holdrege 1999]。具有NAT功能的路由器维护一张地址翻译表，使用UDP和TCP包中源和目的地端口号域，将每个到达的应答消息发给发送该请求消息的内部计算机。注意，请求消息中给定的源端口总是被用做相应应答消息中的目的地端口。

最常用的NAT寻址算法的工作流程如下：

- 当内部网络上的计算机发送一个UDP或TCP包给网络外的计算机时，路由器接收到数据包并将源IP地址和端口号保存为地址翻译表中一个可用的项。
- 路由器用路由器的IP地址替换包中的源地址，用虚拟端口号替换源端口，虚拟端口号指向包含发送计算机的地址信息的地址翻译表项。
- 已修改源地址和端口地址的数据包经路由器向它的目的地转发。现在，地址翻译表包含最新的从内部网上计算机发出的包的端口号和从虚拟端口号到实际内部IP地址的映射。
- 当路由器从外部计算机处接收到一个UDP或TCP包时，它使用包中的目的地端口号访问地址翻译表中的项。它用存储在表项中的值替换已接收包中的目的地址和端口号，然后将修改后的包转发到由目的地地址标识的内部计算机。

只要该端口还在使用，路由器就将保留端口映射并重用它。每次路由器访问表中的一项，就重设计时器。如果在计时器过期之前没有访问该表项，那么就从表中删除该表项。

上述方案很好地解决了未注册计算机的通信模式，在这种模式下，未注册计算机可以作为外部服务（例如Web服务器）的客户。但未注册计算机不能作为处理到达请求的服务器。为了处理这种情况，可以手工配置NAT路由器，将某个指定端口上所有到达的请求转发到一台指定的内部计算机上。作为服务器的计算机必须保留同样的内部IP地址，这一点可通过手工分配它们的地址

（类似对PC1所做的操作）来达到。只要不要求多于一台内部计算机在指定端口提供服务，这种提供对服务的外部访问的解决方法是令人满意的。

NAT是一种解决个人和家庭计算机分配IP地址的短期解决方案。它使得因特网使用的扩张得比预期的更大，但它也有一些限制，例如上例中的最后一点。IPv6被看成是下一步骤，它将使得所有计算机和便携设备能全方位地参与因特网。

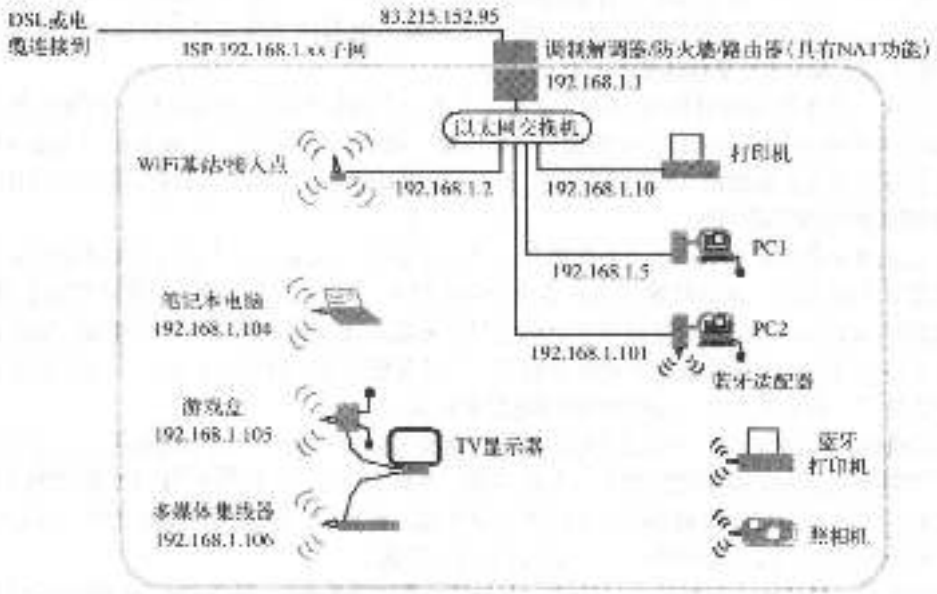


图3-18 一个典型的基于NAT的家庭网络

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，这导致了具有更大地址空间的新版本的IP协议的开发与使用。早在1990年，IETF就注意到IPv4的32比特地址所带来的潜在问题，并启动了开发新版本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3-19显示了IPv6头的格式。在此，我们不详细介绍它们的构造方法。要获得有关IPv6的相关内容，读者可以参考Tanenbaum[2003]或Stallings[1998a]。要获得IPv6设计过程与实现计划详尽的介绍，可以参阅Huitema[1998]。这里将概述IPv6的主要改进。



图3-19 IPv6头部格式

地址空间：IPv6的地址有128比特（16字节）。这提供了海量的可寻址实体数：2^{128}，即大约3×10^{38}。据Tanenbaum计算，在整个地球表面，每平方米空间可以有7×10^{23}个IP地址。Huitema的

估计比较保守，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。

IPv6地址空间进行了分区。在此我们不详细介绍分区，但即使是最小的分区（其中的一个分区会包含整个IPv4地址范围，这里地址的映射是一对一的）也远远大于整个IPv4地址空间。很多分区（占总数的72%左右）被保留，目前为止向未被定义。两个大的分区（每个分区包含1/8的IP地址空间）作为日常使用，将被分配给普通的网络节点。其中的一个分区根据地址节点的地理位置组织，而另一个分区根据机构位置组织。这提供了两种不同的用于聚类地址的策略以便进行路由——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节点上的处理时间都被降低。数据包的内容（有效负载）不使用任何校验和，一旦数据包开始传输，就不能再将它分段。前者被认为是可接受的，因为可在更高层检测错误（TCP确实包含了一个内容校验和），而后者通过支持在发送数据包前确定最小的MTU而达到目的。

实时以及其他特殊服务：*流量类别*与*流标号*域与此有关，多媒体流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。*流量类别*的前六位可与*流标号*同时使用，也可以独立使用，以使指定数据包比其他数据包的处理速度更快或是更可靠。流量类别值0～8用于即使有延迟也不会对应用造成灾难性后果的传输。其他值被保留，用于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（例如有播的音频与视频传输）的时间需求。第17章将讨论它们的资源分配需求与分配方法。当然，因特网上的路由器与传输链路的资源有限，为特定用户预留资源的概念和应用以前未曾考虑。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。

100 ~ 102

未来的发展：提供未来发展的关键是下一个*头域*。若该域为非0，则它定义了数据包中的扩展头的类型。目前的扩展头类型为下列类型的特殊服务提供附加数据：路由器信息、路由定义、片断处理、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的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）放在基本头之后有效负荷之前，它会包含下一个头域，使数据包可以使用多个扩展头。

组播与任意播：IPv4与IPv6支持将IP数据包通过一个地址（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参见4.5.1节。另外，IPv6支持一种称为*任意播*的新的传输模式。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，需要认证的因特网应用或私密性数据传输依赖于应用层的加密技术。端到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层实现安全性，那么用户与应用程序开发者依赖于传输路径上的每个路由器都正确地实现了加密算法，为处理密钥，他们必须信任路由器以及其他中间节点。

在IP层实现安全性的好处在于，它可用于应用程序不清楚具体安全实现的场合。例如，系统管理员可以将它实现到防火墙中，将它统一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，从而保证它们之间交换的路由表更新消息的安全。

在IPv6中使用*认证与加密的安全性有效负载*扩展头类型实现安全性。这些实现特征与2.3.3节介绍的安全通道的概念类似。根据需要，可给有效负载加密或者（并且）应用数字签名。类似的安全特征也可在IPv4中获得，这时使用了实现IPSec规约（见RFC 2411 [Thayer 1998]）的IP隧道。

从IPv4迁移 改变因特网基础设施的基本协议层带来的后果是深远的。每台主机的TCP/IP协议栈和路由器软件都需要处理IP，很多应用与实用程序都需要处理IP地址。为了支持新版本的IP协

议，上述应用都需要升级。进行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，它主要涉及下列问题的实现：使用隧道技术，将IPv6的路由器和主机“岛屿”与其他IPv6“岛屿”通信，然后逐渐地形成一个大的“岛屿”。

我们在前面提到过，IPv6路由器和主机在处理混合流量时应该没有任何困难，因为IPv4地址空间被嵌入到IPv6空间内。所有主要的操作系统（Windows XP、MacOS X、Linux和其他UNIX变体）已经包括了在IPv6上UDP和TCP套接字（见第4章）的实现，这使得应用能通过简单的升级完成迁移。 103

该策略的理论从技术上说是可行的，但实现过程非常缓慢，这也许是由于CIDR和NAT已经减轻了所期望的更大范围使用因特网的压力，但这在移动电话和便携设备市场已经发生了改变。所有这些设备在不久的将来就可能具备访问因特网的功能，同时它们不能容易地隐藏在NAT路由器后面。例如，预计到2014年，印度和中国将部署超过10亿台IP设备。只有IPv6能解决这样的需求。

3.4.5 移动IP

像笔记本电脑和掌上电脑这样的移动计算机可以在移动时从不同的位置连接到因特网。当用户在自己办公室时，笔记本电脑可以先连接到本地以太网，然后通过路由器连接到因特网；在乘车旅行途中，可以通过移动电话连接到因特网，然后，在另一个地点连接到以太网上。用户希望在任何一个地方查看电子邮件和访问Web。

对服务的简单访问并不需要移动计算机保留一个地址，它可在任意地方获得一个新的IP地址。动态主机配置协议（DHCP）正是用于这一目的的，它使新接入网络的计算机动态获得一个在本地子网地址范围内的IP地址，并从从本地DHCP服务器上找到诸如DNS服务器这样的本地资源地址，它也需要找到它所访问的每个站点上有哪些本地服务（如打印、邮件传送等）。发现服务是有助于完成此工作的一种命名服务，其具体内容将在第16章（16.2节）中介绍。

笔记本电脑上可能有其他人员需要访问的文件或其他资源，或者它正在运行分布式应用（如共享监控服务），它接收用户拥有的股票超过一定阈值这样的特定事件的通知。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，移动计算机必须保持单个IP号，但IP路由是基于子网的。子网位于固定的地点，将数据包正确地路由到子网取决于子网在网络上的位置。

移动IP是后一个问题的解决方案，该方案的实现对用户是透明的，因此当移动主机在不同位置的子网中移动时，IP通信会继续正常地进行。这是因为“主”（home）域的子网中的每台移动主机拥有永久固定的IP地址。

当移动主机在“主站点”中连接到因特网时，数据包会以正常方式路由到主机上；当移动主机在其他地方连入因特网时，有两个代理进程负责重新路由。它们是*主代理*（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处的固定计算机上。

HA负责保存移动主机当前位置（即可以到达该移动计算机的IP地址）的最新情况，它在移动主机自身的帮助下完成该功能。当一个移动主机离开主站点时，它会告知HA，HA会注意到该移动主机离开。当主机离开时，HA就充当一个代理服务器。为实现代理功能，HA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会响应有关移动主机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发送给该请求。

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，将移动主机的主IP地址以及分配给它的转交地址告知HA。

图3-20说明了移动IP的路由机制。当一个以移动主机的主地址为地址的IP数据包被传送到主网络上时，它将被路由到HA。然后，HA将该IP数据包封装到一个移动IP数据包中，并发送给FA。

FA拆解出原来的IP数据包，并通过它当前连接的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3.3.7节描述的隧道传输技术的实例。

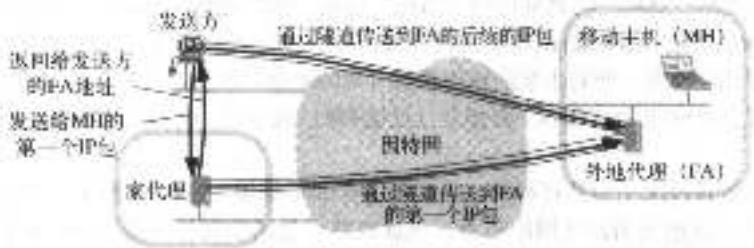


图3.20 移动IP路由机制

HA也将移动主机的转交地址发送到原来的发送者。如果发送者支持移动IP，它将注意到新的地址，并且使用新的地址与移动主机接着通信，避免了通过HA重新路由的开销。如果发送者不支持移动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，但还不是十分有效。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，并且不必使用隧道技术就可将数据包路由到主机。应该注意，这个看上去很难的技术已在移动电话网中实现——当移动电话在不同蜂窝乃至国家之间移动时，并不需要改变电话号码，它们只需时常通知本地移动电话网基站它们的存在即可。

104 / 105

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的传输服务，如提供实时保证或安全性，但这些服务需要比IPv4更多的网络层支持。TCP和UDP忠实地反映了IPv4提供的应用编程级的通信设施。IPv6是另一件事。它必然会继续支持TCP和UDP，但它包含了通过TCP和UDP无法方便访问的功能。当IPv6的部署已足够广使得那些功能被证明应该被开发的时候，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的特征。这里我们仅描述它们给IP加入的功能。

端口的使用 第一个要注意的特征是，尽管IP协议支持两台计算机（由IP地址标识）之间的通信，但作为传输层的协议，TCP和UDP必须提供进程到进程的通信。这通过使用端口来完成。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，端口号是一个16位整数。一旦一个IP数据包被发送到目的主机，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端口号的短的头部（相应的主机地址位于IP头部）、一个长度域和一个校验和。UDP不提供传输保证。我们已经注意到，IP数据包可能会由于拥塞或网络错误被丢弃。除了可选的校验和外，UDP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，依赖IP传输，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。它不需要任何创建开销以及管理用的确认消息，但它只适应于不需要可靠传送单个或多个消息的服务和应用。

TCP的特点 TCP提供了一个更复杂的传输服务。它通过基于流的编程抽象，提供了任意长度字节串的可靠传输。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，其顺序是相同的。TCP是面向连接的，在数据被传送前，发送进程和接收进程必须合作，建立一个双向的通

信通道。连接只是一个执行可靠数据传输的端到端的协议，中间节点（如路由器）并没有关于TCP连接的知识，一个TCP传输中传输数据的所有IP数据包并不一定使用相同的路由。

TCP层包含额外机制（在IP之上实现）以保证可靠性。这些机制包括： 106

排序：TCP发送进程将流分割成数据片断序列，然后将之作为IP数据包传送。每个TCP片断均有一个序号。它在该片断的第一个字节给出流中的字节数。接收程序在将数据放入接收进程的输入流前，使用序号对收到的片断排序。只有所有编号较小的片断都已收到并且放入流中后，编号大的片断才能被放入流中，因此，未按顺序到达的片断必须保留在一个缓冲区中，直到它前面的片断到达为止。

流控制：发送方管理不能使接收方或是中间节点过载，这通过片断确认机制完成。当接收方成功地接收了一个片断后，它会记录该片断的序号。接收方会不时地向发送方发送确认信息，给出输入流中片断的最大序号以及*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片断中，否则被放在确认数据片中。确认片断中的窗口大小域指定了在下一个确认之前允许发送方传送的数据量。

当一个TCP连接用于与一个远程交互程序通信时，会猝发产生数据，但产生的数据量可能很小。例如，利用键盘输入可能每秒仅输入几个字符，但字符的显示必须足够快，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（一般是0.5s）来实现。使用这个简单的方案，只要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，就将片断发送到接收方。该缓冲区方案不会使交互式延迟再增加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的算法已用于许多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。

由于无线网络的不可靠性，会导致数据包丢失频繁发生，上面的流控制机制对于无线通信不是特别适用。这是广域移动通信使用的WAP协议族采纳另一种传输机制的原因。但对无线网络而言，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]。其思想是在无线基站（有线网络和无线网络之间的网关）实现一个TCP支持组件。该组件探听进出无线网络的TCP片断，重传任何未被移动接收方快速确认的外发片断，并且在注意到序列号有间隔时，请求重传接收数据。

重传：发送方记录它发送的片断的序号。当它接收到一个确认消息时，它知道片断被成功接收，并将之从外发缓冲区中清除。如果在一个指定超时时间内，片断并没有得到确认，则发送方重发该片断。 107

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，那么缓冲区中的数据量就会增加。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，此时到达的片断不被记录就直接被丢弃了。因此，接收方不会给出相应的确认，而发送方将被迫重新发送片断。

校验和：每个片断包含一个对头部和片断中数据的校验和，如果接收到的片断和校验和不匹配，则片断被丢弃。

3.4.7 域名

第9章将详细介绍域名系统（DNS）的设计与实现，在此我们只做简单的介绍，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方案，如binkley.cs.mcgill.ca或essex.ac.uk。已命名的实体被组织成一个命名层次结构。已命名的实体称为*域*，而符号名称为*域名*。域被组织成一个层次结构，以便反映它们的组织结构。命名层次结构与构成因特网的网络物理布局完全无关。域名对于用户很方便，但它们在被用作通信标识符之前，必须翻译成因特网地址（IP地址），这是DNS服务的职责。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实现为一个可在因特网的任意主机上运行的服务器进程。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该域之下的域名树的部分视图。它们至少必须存储自己域中的所有域名和主机名，但通常包含树的更大的部分。若DNS服务器接收到的请求中，需要翻译的域名在自己所保存的那部分树以外，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部分。翻译结果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将无法工作，因为基本上在每种情况下都会查询“根”名字服务器，从而形成一个服务访问瓶颈。

3.4.8 防火墙

几乎所有的组织都需要连接因特网，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。大多数组织中的计算机是完全不同的，它们运行不同的操作系统和应用软
108 件。软件的安全性差别更大，有些软件提供了先进的安全措施，但大多数软件没有能力或有很少的能力保证进入的通信是可靠的，向外的通信是私密的。总之，在一个有很多计算机和多种软件的企业内部网中，系统的有些部分在安全攻击下会非常地脆弱是不可避免的。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。防火墙由一组进程实现，它作为通向企业内部网的网关（参见图3-21a），应用了组织规定的安全策略。

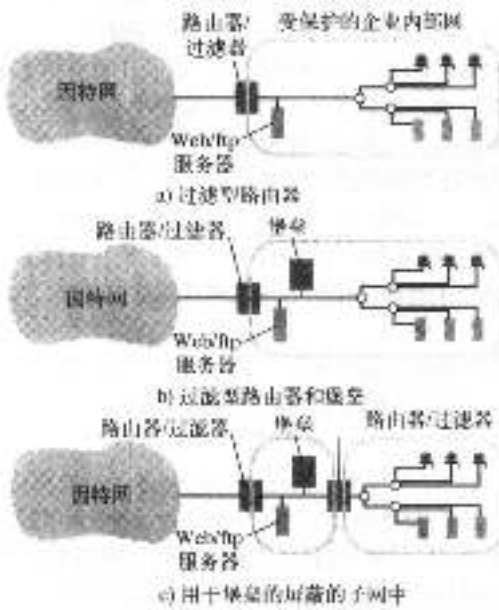


图3-21 防火墙配置

防火墙安全策略的目标可能包括下面的某些或所有内容：

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，并拒绝其他的服务请求。外发服务请求和应答也受到控制。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请求来完成。例如，到达的HTTP请求的目的地应该是官方的Web服务器主机，否则该请求会被拒绝。

行为控制：行为控制用于防止破坏公司策略的、反社会的、找不到可辩认的合法目的的行为，
109 这些行为被怀疑为构成了攻击的一部分。其中的某些过滤行为可在IP或TCP层进行，但其他行为可能需要在更高层对消息进行解释。例如，过滤垃圾邮件攻击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：组织可能希望在用户之间加以区分，允许某些用户访问外部服务，而其他用户则禁止访问外部服务。另一个大家更易接收的用户控制例子是，避免接收系统管理组成员以外的其他用户的软件，以免感染病毒或是维护软件标准。这是个特殊的例子，如果不禁止普通用户使用Web，要想实现上述目的是很困难的。

用户控制的另一个实例是拨号以及其他为不在站点的用户提供连接的管理。如果防火墙同时也是通过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（防止外来的窃听、伪装和其他攻击）。这是下一节将要描述的虚拟私网（VPN）技术的目的。

这些策略必须以过滤操作的方式表达，而这些操作由在不同层操作的过滤进程执行：

IP数据包过滤：这是一个检查单个IP数据包的过滤进程，它会根据源地址和目的地址进行决策。它也会检查IP数据包的服务类型域，并根据服务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，因为服务通常位于大众熟知的端口上，从而可以根据请求的服务过滤数据包。例如，很多站点禁止外部客户使用NFS服务器。

从性能方面考虑IP过滤通常由路由器的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面的防火墙做更彻底的检查，同时让“干净”的数据包继续发送，也有可能基于IP数据包的顺序进行过滤，例如，在执行登录命令前，禁止对FTP服务器进行访问。

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。安装了TCP网关进程后，可控制TCP连接的创建，检查TCP数据片的正确性（一些服务拒绝攻击用残缺的TCP数据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，用户希望有这样的策略：允许特定内部用户的Telnet连接创建到特定外部主机。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，该请求被TCP网关截获。TCP网关启动一个Telnet代理进程，原有的TCP连接被路由到该进程。如果代理通过了Telent操作（用户被授权使用所请求的主机），那么它会建立另一个通向所请求的主机的连接，并由它中转所有来往的TCP数据包。一个类似的代理进程 110
将代表每个Telnet客户而运行，而类似的代理可能被FTP和其他服务所采用。

一个防火墙通常由工作在不同协议层的多个进程组成。考虑到性能和容错，通常在防火墙中使用一台以上的计算机。在下面描述的并由图3-21说明的所有配置中，我们给出了一个不受保护的Web服务器和FTP服务器。它只包含一些已发布的信息，这些信息对公共访问不加防范，而服务器软件必须确保只能由授权的内部用户修改。

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位于不同IP网络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个数尽可能少的其他进程。路由器/过滤器仅运行可信的软件，其运行方式要保证过滤策略的执行。这涉及不能运行特洛伊木马进程，以及路由器和过滤器软件不被修改或破坏。图3-21a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中的网络配置包含两个作为此类防火墙的路由器/过滤器。该配置中有两个路由器/过滤器，以确保性能和可靠性。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，该计算机称为堡垒（这个术语源于城堡的构筑，城堡有一个突出的瞭望塔用来保护城堡）。堡垒计算机是一台位于企业内部网中由IP路由器/过滤器保护的主机，它运行TCP和应用层网关（参见图3-21b）。与路由器/过滤器类似，堡垒只运行可信的软件。在一个足够安全的企业内部网内，代理必须用于访问所有的外部服务。读者可能已经对用于Web访问的代理很熟悉了，它们都是防火墙代理的应用实例，并且通常和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

需要大量的存储和处理资源。

应用以串联方式部署的两台路由器/过滤器以及堡垒和位于与路由器/过滤器相链接的单独子网内的公共服务器可以提高安全性能（见图3-21c），这种配置在安全方面有以下优势：

- 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，因为所有的外部通信都要通过堡垒内的代理进程完成，而代理进程可以访问两端的计算机。
- 如果第一个路由器/过滤器被攻破，那么第二个路由器/过滤器（由于原本外部不可见而不易受攻击）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，我们概述了使用IPSec隧道技术对IPv6和IPv4进行的IP安全扩
111 展[Thayer 1998]，这些都是实现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的位于不同站点的企业内部网之间实现安全连接。

例如，一个员工需要通过ISP连接到组织的企业内部网。一旦连接成功，他就应该拥有和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，这些密钥用来在连接时建立安全通道。安全通道机制将在第7章中详细介绍。

3.5 实例研究：以太网、WiFi、蓝牙和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。在结束本章前，我们将描述三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）成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]），它的分会制订了一系列已成为LAN关键标准的规约。在大多数情况下，这些标准基于20世纪70年代由研究而来的已有工业标准。相关的分会以及迄今发布的标准如图3-22所示。

IEEE No. 名字	标 题	参 考
802.3 以太网	CSMA/CD网络（以太网）	[IEEE 1985a]
802.4	令牌总线网	[IEEE 1985b]
802.5	令牌环网	[IEEE 1985c]
802.6	城域网	[IEEE 1994]
802.11 WiFi	无线局域网	[IEEE 1999]
802.15.1 蓝牙	无线个域网	[IEEE 2002]
802.15.4 ZigBee	无线传感器网络	[IEEE 2003]
802.16 WiMAX	无线城域网	[IEEE 2004a]

图3-22 IEEE 802网络标准

这些标准在性能、有效性、可靠性和成本上有所不同，但它们都提供了在中短距离上相对较高的网络带宽。IEEE 802.3以太网标准极大地赢得了有线LAN市场。作为有线LAN的代表技术，我们将在3.5.1节中描述它。尽管以太网实现有多种可用带宽，但它们的操作原理是相同的。

IEEE 802.5令牌环网标准在20世纪90年代是以太网的一个重要竞争者，它比以太网更有效并能保证带宽。但它现在已经从市场上消失了。如果读者对这种LAN技术感兴趣，可以在www.cdk4.net/
112 networking找到它的简要描述。以太网交换机的广泛使用（与集线器相对）使得以太网能以提供带宽和延迟保证（进一步的讨论见3.5.1节中“用于实时应用和质量保证至关重要的应用的以太网”）的方式被配置，这是它取代令牌环网技术的一个理由。

IEEE 802.4令牌总线标准是为具有实时需求的工业应用而开发的，并应用于该领域。IEEE 802.6城域网标准覆盖高达50公里的距离，并用于跨城镇的网络。

IEEE 802.11无线LAN标准的出现稍晚一些，但由于许多制造商生产的WiFi产品以及它被安装到大量移动设备和手持计算设备上，它目前已经在市场上占据了主要的位置。IEEE 802.11标准支持具有简单的无线发送器/接收器设备之间的通信，设备间的距离在150米之内，速度可高达54Mbps。我们在3.5.2节描述它的操作原理。IEEE 802.11网络的详情可以在Crow等[1997]以及Kurose和Ross[2000]中找到。

IEEE 802.15.1无线PAN标准（蓝牙）基于1999年由爱立信公司开发的技术，该技术可在不同设备（例如PDA、移动电话和耳机）之间传输低带宽的数字声音和数据，并在2002年标准化成IEEE 802.15.1。3.5.3节将详细介绍蓝牙。

IEEE 802.15.4（ZigBee）是另一个WPAN标准，它用于为家中极低带宽低能量设备（例如远程控制、防盗报警和加热系统传感器）和无处不在设备（例如Active badge、标签读取器）提供数据通信。这样的网络称为*无线传感器网络*，它们的应用和通信特征见第16章。

IEEE 802.16无线MAN标准（商用名称为WiMAX）在2004～2005年被批准。IEEE 802.16标准作为家庭和办公室的“最后一公里”连接的电缆和DSL连接的替代品。该标准的一个变体意在替代802.11 WiFi网络成为室内外公共区域中笔记本电脑和移动设备之间的主要连接技术。

20世纪80年代末到90年代初，ATM技术从电信和计算机界的研究和标准化工作中产生[CCITT1990]。它的目标是提供适合电话、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。尽管它被接受的过程比预期缓慢，但ATM现在是超高速广域网的主导技术。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，因为100Mbps和1000Mbps以太网通过低得多的价格与之竞争。我们将在3.5.4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参阅Tanenbaum[2003]和Stallings[1998a]的著作。

3.5.1 以太网

以太网是1973年[Metcalfe and Boggs1976; Shoch et al. 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出来的。该实验以太网是第一个高速局域网， 113
展示了在某个地点链接计算机，并使其以低错误率、无交换延迟的高速传输速率互相通信的高速局域网的可行性和可用性。最初的以太网原型以3Mbps的速度运行，现在以太网系统的可用带宽已经扩展到10Mbps～1000Mbps。

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局域网技术。100Mbps的变体是现在广泛使用的一种以太网，它的操作原理与10Mbps类似。本节最后将总结目前以太网传输技术更重要的变体以及可用的带宽。所有以太网变体的综合描述，请参见Spurgeon[2000]。

单个以太网是一个简单的或有分支的类似总线的连接，它使用的传输介质由通过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通过以太网交换机或网桥连接。交换机和网桥在以太网帧层操作，将地址为邻接以太网的帧转发过去。对于IP这样的高层协议，连接起来的以太网可看作一个网络（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的交换机连接）。特别是ARP协议（参见3.4.2节），它可以跨越相互连接的一组以太网来解析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“具有冲突检测的载波侦听多路访问”（简称CSMA/CD），它们属于*竞争总线类网络*。竞争总线使用一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*（Medium Access Control, MAC）协议。由于单一链路连接所有主机，所以MAC协议将数

据链路层协议（负责在通信链路上传输数据包）和网络层协议（负责将数据包传输到主机）的功能合并到一个协议层中。

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包。所有工作站不断地“监听”介质上传输的数据包的目的地是否是自己。任何想发送消息的工作站会广播一个或多个数据包（在以太网规约中称为帧）到介质上。每个数据包包含目的工作站地址、发送工作站地址和表示要传输消息的变长比特序列。数据传输以10Mbps的速度（在100Mbps和1000Mbps以太网上以更高速度）进行，数据包长度为64B到1518B。因此，在10Mbps以太网上传输一个数据包的时间是50～1200μs，具体时间取决于数据包的长度。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界限，但在IEEE标准中，MTU还是被定义为1518B。

114 目的工作站的地址通常指一个网络接口。每个工作站的控制器硬件接收每个数据包的一个副本。它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，并将地址匹配的数据包接收到本地主机。目的地址也可以指定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区分（前者为0，后者为1）。全为1的地址被保留为广播地址，在一条消息被网络上所有工作站接收时使用。这可用于实现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。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具有组播地址的数据包的工作站接收，但不是所有的以太网接口实现都可以识别组播地址。

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 以太网工作站上传输的数据包（或更准确地说是帧）具有以下格式：

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外，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，包含7字节的前同步符，每个前同步符包括位模式10101010组成，其后是一字节的开始帧分界符（在图中是S），分界符的模式为10101011。

尽管标准规定单个以太网中的工作站不能超过1024个，但以太网的地址占了6字节，可提供2^{48}个不同的地址。这使得每个以太网硬件接口制造商可以给硬件接口分配一个唯一的地址，以保证所有互连的以太网中的工作站都有唯一的地址。美国电气和电子工程师协会（IEEE）作为以太网地址分配的负责方，将48比特地址的不同范围分配给以太网硬件接口制造商。这些地址被称为MAC地址，因为它们用于介质访问控制层。事实上，以这种方式分配的MAC地址已经被IEEE 802家族中其他网络类型（例如802.11（WiFi）和802.15.1（蓝牙））采用为唯一地址。

数据域包含要传输的消息的全部或一部分（如果消息长度超过1500字节）。数据域的下限为46字节，这可以保证数据包最小长度为64字节，设置下限是必要的，这可以保证网络上所有工作站的冲突都能检测到，下文对此做了解释。

帧校验序列是一个校验和，它由发送者产生并插入数据包中，由接收者用于验证数据包。校验和不正确的数据包由接收工作站的数据链路层丢弃。这是端到端争论的应用的另一个例子，即
115 为了保证消息的传输，必须使用像TCP这样的传输层协议，它会对每个接收到的数据包发出确认信息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能获得令人满意的效果，并且当不需要保证数据传输时，可以采用像UDP这样开销比较小的协议。

数据包冲突 即使数据包的传输时间相当短，也有可能出现网络上两个工作站同时传输消息的

情况。如果一个工作站试图传输一个数据包，而没有检查传输介质是否被另一个工作站使用，就会产生冲突。

以太网有三种机制来处理这种可能性。第一种机制称为载波侦听。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）。当一个工作站欲传输一个数据包时，它会等到介质上没有信号出现时才开始传输。

遗憾的是，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限时间τ（信号以电波速度传播，大约每秒2×10^8m）。假设两个工作站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。当一个工作站通过其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。我们已经注意到，最小数据包长度可以确保检测到冲突。如果两个工作站几乎同时从网络的另一端传输，它们在2τ秒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传输完自己的数据包才会看到别的数据包，而中间的工作站将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，所有传输工作站和监听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。这会产生更大的困难。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，就可能发生另一个冲突。为避免这种情况，可以使用称为后退的技术。发生冲突的每个工作站选择在传输之前等待一段时间$n\tau$。n是一个随机整数，由每个工作站分别选取，并小于在网络软件中定义的常数L。如果产生进一步的冲突，将L的值加倍，必要的话可将整个过程重复10次。

最后，接收工作站的接口硬件计算校验序列，并将之与数据包中传送的校验和相比。使用这 116
些技术，连接到以太网的工作站便可以在无任何集中控制或同步的情况下管理介质的使用。

以太网的效率 以太网的效率定义为成功传送的数据包的个数与无冲突情况下理论上能传输的数据包的最大值之间的比率。它受τ值的影响，因为数据包传送后的2τ秒间隔是冲突的“机会窗口”，即在数据包开始传输2τ秒后不会有冲突发生。网络上工作站的数目以及它们的活动性也会影响效率。

对于长度为1km的电缆，τ的值小于5ms，因此冲突概率很小，足以确保传输的高效性。尽管当通道利用率大于50%时，争夺通道造成的延迟足以令人重视，但以太网仍可以获得80%～95%的通道利用率。因为负载是变化的，所以不可能保证在一段固定的时间内传递给定信息，原因是网络可能在准备传输消息时变成满负荷运行。但在给定的延迟内传递消息的概率等同或好于其他网络技术。

Xerox PARC的Shoch与Hupp[1980]报告的关于以太网性能的实际测量数据确认了上述分析。在实际中，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，对于那些携带多媒体流的网络，如果有几个流同时传输的话，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应用，使得我们可以获得执行以太网算法的低成本的控制器硬件，它已成为很多桌面计算机与消费类计算机的标准部件。

有很多不同的以太网物理实现，它们是基于不同的性能/成本权衡提出的，也利用了不断增长的硬件性能。不同的实现源于使用了不同的传输介质，包括同轴电缆、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，并有一个区分它们的命名方案。可使用10Base5与100BaseT这样的名字，它们具有如下形式：

<*R*><*B*><*L*>

其中：*R* = 以Mbps计的数据率

B = 媒体信号类型 （基带或宽带）

L = 以米/100计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范围列在图3-23中。以*T*结尾的配置由
117 UTP电缆（非屏蔽双绞线，即电话线）实现，它被组织成集线器层次结构，而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。

	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
多模光纤	2000m	2000m	500m	500m
单模光纤	25000m	25000m	20000m	2000m

图3-23 以太网范围和速度

用于实时应用和质量保证至关重要的应用的以太网 以太网MAC协议因为它缺乏传递延迟的保障，所以不适合实时应用或需要质量保证的应用，这一点经常被讨论。但应该注意到，现在大多数以太网的安装都基于MAC层交换机的使用（如图3-10所示，有关的描述见3.3.7节），而不是以前的集线器或带有堵头的电缆。交换机的使用使得每个主机对应一个单独的网段，除了到达这个主机的包之外没有其他包传递给它。因此，如果到该主机的流量来自一个源，那么就没有介质冲突——有效性是100%，延迟是常量。竞争的可能性仅出现在交换机上，这些能够并且经常用于并发地处理包。因此，一个轻负载的基于交换机的以太网安装几乎100%有效，能延迟通常是一个小常量，所以它们经常被成功地用于关键性应用。

对以太网风格的MAC协议的实时支持可见[Rether, Pradhan and Chiueh 1998]的描述，类似的方案在开源的Linux扩展[RTnet]中实现。这些软件方法通过实现一个应用层协作协议为介质的使用保留了时间槽，从而解决了竞争问题，它依赖连接到一个网段的所有主机的协作。

3.5.2 IEEE 802.11无线LAN

本节将总结无线LAN技术中必须解决的无线网络的特殊特征，同时解释IEEE 802.11是如何处理这些特征的。IEEE 802.11（WiFi）标准扩展了以太网（IEEE 802.3）技术采用的载波侦听多路复用（CSMA）原理以适应无线通信的特征。802.11标准旨在支持距离在150m之内以最高54Mbps的速度进行的计算机间通信。

图3-24是包含无线LAN的企业内部网的一部分。几个移动无线设备通过基站和企业内部网的
118 其他设备通信，这里基站是有线LAN的*接入点*。通过接入点与传统LAN连接的无线网络称为基于*基础设施的无线网络*。

无线网络的另一种配置方式称为*自组织网络*。自组织网络不包括接入点或基站。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笔记本电脑用户发起与任何可用站点的连接时，就会形成一个自组织网络。它们可以通过在某台机器上启动文件服务器进程来共享文件。

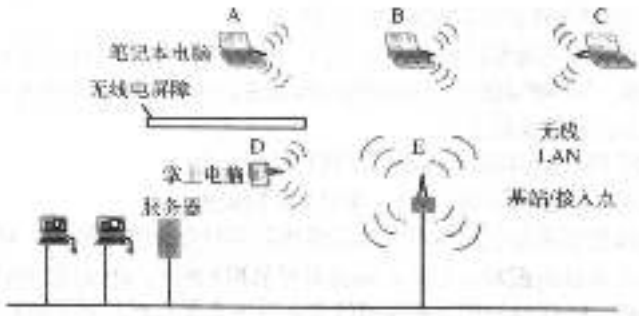


图3-24 无线LAN配置

IEEE 802.11网络在物理层采用无线电频率信号（利用免牌照使用的2.4GHz和5GHz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。IEEE 802.11b标准是第一个广泛使用的派生标准。它在2.4GHz波段运行，支持高达11Mbps的数据通信。它从1999年起在许多办公室、家庭和公共场所与基站一起被安装，使笔记本电脑和PDA能访问局域网设备或因特网。IEEE 802.11g是对802.11b最近的更新，它仍使用2.4GHz波段但使用不同的信号技术从而获得高达54Mbps的速度。最后，802.11a派生标准工作在5GHz波段，在更短范围内带宽可达54Mbps。所有的派生标准采用不同的频率选择或者跳频技术，以避免外部干扰以及独立的无线LAN之间的相互干扰（后者我们不准备详细讨论）。我们重点讨论对CSMA/CD机制做的修改，这些修改是802.11的所有版本的MAC层所需要的，并使得广播数据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，站点之间可以直接传输。但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这一事实。

我们回忆一下，载波侦听的目的是确定发送工作站和接收工作站间的所有节点上的介质是否空闲，冲突检测的目的是确定在接收者邻域内的介质是否空闲，以免在传输时受到干扰。由于无线LAN操作的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：

工作站隐藏：载波侦听没能检测到网络上另一个工作站正在传输。图3-24可以说明这一点，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，笔记本电脑A可能发现不了D的信号，于是A开始传输，若不采取手段防止A传输，将在E点造成冲突。

信号衰减：由于电磁波传输遵循反平方规则衰减，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内的某个工作站可能在其他工作站的范围之外。如图3-24所示，虽然笔记本电脑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：遗憾的是，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其他地方产生的信号强很多，极大地掩盖了远程传输。因此，笔记本电脑A和C可能同时向E传送，它们都没有检测到冲突，但E却只收到了乱码。

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加强。这种方案称为具有冲突避免的载波侦听多路复用（CSMA/CA）。

在工作站准备发送消息时，它侦听介质。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）范围之外的工作站正在请求获得一个时隙。

119

3）范围之外的工作站正在使用以前保留的时隙。

时隙保留协议包括在发送者和接收者之间交换一对短消息（帧）。首先是发送者给接收者发一个*请求发送*（RTS）帧，RTS消息指定了需要的时隙长度。接收者回复*清除发送*（CTS）帧，并重复时隙的长度。这种交换的效果如下：

120
- 发送者范围内的工作站将获得RTS帧，并记录时隙长度。
- 接收者范围内的工作站将获得CTS帧，并记录时隙长度。

结果，发送者和接收者范围内的所有工作站在规定的时隙内都不传输，留出空闲通道给发送者，使之能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在以下几个方面有助于避免冲突：

- CTS帧有助于避免工作站隐藏和信号衰减问题。
- RTS和CTS帧很短，所以冲突的风险也很小。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，那么随后的数据和确认帧应当没有冲突，除非间歇性的信号衰减导致第三方没有接收到RTS帧或者CTS帧。

安全性 通信的私密性和完整性显然是无线网络中必须关注的问题。处于范围内且配有发送器/接收器的任何一个工作站都可能加入这个网络，如果失败，它也可能窃听其他工作站之间的传输。第一个试图为IEEE 802.11解决安全问题的是WEP（有线等价私密性）。遗憾的是，WEP并没有达到它名字所隐含的目标。它的安全设计在几个方面都有漏洞，使得它很容易被破坏。我们将在7.6.4节描述它的弱点，并总结当前的改进情况。

3.5.3 IEEE 802.15.1 蓝牙无线PAN

蓝牙是一种无线个域网技术，源于通过无线连接移动电话与PDA、笔记本电脑以及其他个人设备的需求。由L. M. Ericsson领导的移动电话和计算机制造商的一个特别的兴趣小组（SIG）为无线个域网（WPAN）开发了一个规约，用于传输数字声音流和数据 [Haartsen et al. 1998]。1.0版的蓝牙标准于1999年发布，蓝牙这个名字出自一个海盗王。然后，IEEE 802.15工作组采用它为802.15.1标准并发布了用于物理层和数据链路层的规约[IEEE 2002]。

蓝牙网络与另一个广泛采用的无线网络标准IEEE 802.11（WiFi）有本质区别，它们在反映WPAN的不同应用需求、不同开销和能量消耗目标上有所不同。蓝牙主要针对非常小的低开销的设备，（例如佩无线耳机），它从移动电话接收数字声频流，同时也支持计算机、电话、PDA和其他移动设备之间的互连。开销目标是在手持设备的开销上增加5美元，能量目标是仅使用电话或
121 PDA总电量的一小部分，甚至能用可穿戴设备（如耳机）的少量电池操作数小时。

目标应用一般要求的带宽比典型无线LAN应用更少，传输范围更短。蓝牙很幸运地与WiFi网络、无绳电话和许多紧急服务通信系统都在2.4GHz免牌照频率带宽上操作。传输以低能量方式进行，在所允许频带的79个1MHz的子带宽上以每秒1600次的比率跳跃，以减少干扰。正常蓝牙设备的输出功率是1mW，覆盖范围仅为10m；100mW设备的覆盖范围约为100m，适用于家庭网络类的应用。通过加入*自适应范围*的设施，可以进一步提高能量的有效性。自适应范围的设施能在协作的设备在附近（由最初接收的信号强度决定）时，将传输功率调整到一个较低的层次。

蓝牙节点动态结对，事先不需要有关知识。下面将给出节点关联协议。在成功关联后，发起节点成为*主节点*角色，其他节点是*从节点*。*微微网*是由一个主节点和至多7个活动的从节点组成的动态关联网络。主节点控制通信通道的使用，给每个从节点分配时间片。一个参与多个微微网的节点可以作为沟通主节点的桥梁——按这种方式链接的多个微微网叫*散射网*。大多数设备具备作为主节点或从节点的能力。

虽然只有主节点的MAC地址用于协议中，但所有的蓝牙节点也都配备一个全局唯一的48位

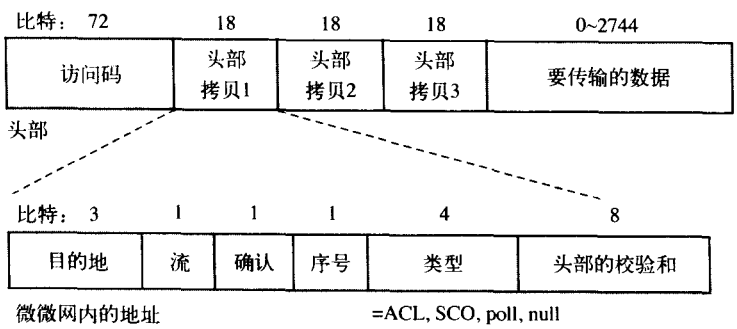
MAC地址（见3.5.1节）。当一个从节点在微微网中被激活，那么就给它赋予一个范围为1~7的临时的本地地址，以减少包头部的长度。除了7个激活的从节点外，一个微微网可以包含至多255个停放节点，以低功率的模式等待来自主节点的激活信号。

关联协议 为了节省能源，在关联前或最近没有发生通信时，设备将保持睡眠或备用模式。在备用模式下，设备每隔0.64～2.56s监听一次激活消息。为了与附近已知的节点（停放节点）相关联，发起节点以16频率子波段，发送16页的包序列，这个过程能重复多次。为了与范围内未知节点相关联，发起者必须首先广播查询消息序列。在最坏情况下，这些传输序列最多占用5s，从而使最大的关联时间约为7～10s。

关联之后，是一个可选的认证交换，该交换基于用户提供的或以前接收到的认证令牌完成，以确保与想要关联的节点关联，而不是与一个欺骗节点关联。接着，通过观察从主节点定时发送的包（即使这些包不是发送给从节点的），从节点与主节点保持同步。未激活的从节点将被主节点置为停放模式，将它在微微网中占用的槽释放供其他节点使用。

如果网络需要支持同步通信通道，并要求有足够的服务质量以进行双向实时音频的传输（如在电话和用户的无线耳机间的传输），同时，需要对数据交换异步通信提供支持，那么网络的体系结构与以太网和WiFi网的尽力而为多路访问的设计不同。同步通信是通过同步面向连接（SCO）的链路实现的，SCO是在主节点和一个从节点之间的一个简单的双向通信协议，主节点和从节点必须轮流地发送同步包。异步通信是通过异步无连接（ACL）链路实现的，这时，主节点周期性地向从节点发送异步轮询包，从节点仅在接收到轮询后发送包。

蓝牙协议的所有变体都使用结构如图3-25所示的帧。一旦建立了微微网，那么访问码由一个固定的导言组成，以使发送者和接收者同步，并识别槽的起点，然后是从主节点的MAC地址中导出的唯一识别微微网的代码。后者确保帧在有多个重叠的微微网的情况下也能正确地路由。因为介质可能有噪声，并且实时通信不能依赖重传，所以头部总是传输三次，头部的每一个拷贝也携带一个校验和，接收者检查校验和并使用第一个有效的头部。



SCO包（例如传递声音数据的）有240比特有效负载，包含80比特数据的3个拷贝，正好占一个时间槽。

图3-25 蓝牙的帧结构

地址域只有3比特，以便寻址到7个当前激活的从节点。发自主节点的0地址表示是一个广播。流控制、确认和序号均用1比特的域表示。流控制比特是供从节点使用的，用于告知主节点它的缓冲区已满。主节点应该等待来自从节点的确认比特非0的帧。每次新的帧从同一节点发出，序列号位就翻转一下。这用于检测重复（即重传的）帧。

SCO链路被用于时间关键性应用，例如双向语音交谈的传输。为了保持低延迟，数据包必须短，在这种应用中报告或重传损坏的数据包，没有太大的意义，因为重传的数据到达得太晚就没有用了。所以，SCO协议使用了一个简单的高度冗余的协议，其中80比特的声音数据按3倍量传输，

即产生240比特的有效负载。任何两两匹配的80比特的副本被认为是有效的。

122～123 另一方面，ACL链路可用于数据传输应用，例如在一台计算机和一部电话之间的地址簿同步，此时的负载比上述的应用更大。这里不复制负载，但可能包含一个内部的校验和，用于应用层的检查，如果出现故障，可以要求重传。

数据以包为单位传递，由主节点分配和控制数据包传递所需的时间，数据包传递占据625μs的时间槽。每个数据包按不同的频率沿一个由主节点指定的跳跃顺序传输。因为这些槽没有大到足以允许实际的负载，所以帧可以被扩展至占据1、3或5个槽。这些特征和底层的物理传输方法使微微网的最大吞吐量达到1Mbps，可在主节点和从节点之间提供3个64Kbps的同步双工通道，或一个用于异步数据传递的速率最大为723Kbps的通道。这些吞吐量是根据上述最冗余的SCO协议版本计算出来的。其他协议变体则是为获得更大吞吐量权衡了3倍数据复制的健壮性和简单性（因此计算开销降低）而定义的。

与大多数网络标准不同，蓝牙包含了几个应用层协议的规约（叫设置文件），有些协议是专用于某一类应用的。设置文件的目的是增加互连不同厂商制造的设备的可能性。13个应用设置文件包括：通用访问、服务发现、串行端口、通用对象交换、LAN访问、拨号网络、传真、无绳电话、对讲机、耳麦、对象推、文件传输和同步。其他的设置文件还在准备中，包括通过蓝牙传输高质量的音频甚至视频。

蓝牙在无线局域网中占据特殊的地位。它达到了支持具有令人满意的服务质量的同步实时音频通信（参见第17章有关服务质量问题的进一步讨论）以及用非常低的开销、小型便携式硬件、低能耗和有限带宽进行异步数据传输的目标。

蓝牙主要的不足在于与新设备关联所花的时间（最多可达10s）。这妨碍了它在某些应用中的使用，特别是在设备之间相对移动的情况下的使用，例如在道路收费或在移动电话用户经过一个商店时传递提示信息给他。关于蓝牙连网的详细内容可参考Bray和Sturman[2002]的书。

蓝牙标准2.0版（其数据吞吐量可高达3Mbps，足够承载CD音质的音频数据）在本书编写的时候正在批准的过程中，包括更快的关联机制和更大的微微网地址等的改进还在开发中。

3.5.4 异步传输模式网络

ATM是用来传输各种数据的，包括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，其操作速度比传统包交换方式快很多。它的高速源于在传输中间站点上避免了流控制和检错，因此，传输链路和节点的数据出错的可能性必须很低。影响性能的另一个因素是数据传输采用简短、定长的单元，这可以减少中间节点缓冲区的大小、
124 复杂性和队列延迟。ATM以连接模式运行，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ATM叠加在像SONET同步光纤网[Omidyar and Aldridge 1993]这样的同步高速数据链路网络上时，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速度不同的多种服务，其中包括语音（32Kbps）、传真、分布式系统服务、视频和高清电视（100～150Mbps）。ATM[CCITTT 1990]标准推荐数据传输率高达155Mbps或622Mbps的虚电路。

ATM网络可在光纤、铜线和其他传输介质上使用固有模式直接实现，使用现有的光纤技术，它们的带宽可以达到每秒十亿位。该模式在局域网和城域网中使用。

ATM服务的结构分为三层，如图3-26中用深黑条表示的部分，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层上实现现有的TCP/IP与X.25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。为了构建某个高层协议，还可

以包括一些通用的功能，如数据包的组装和拆卸。

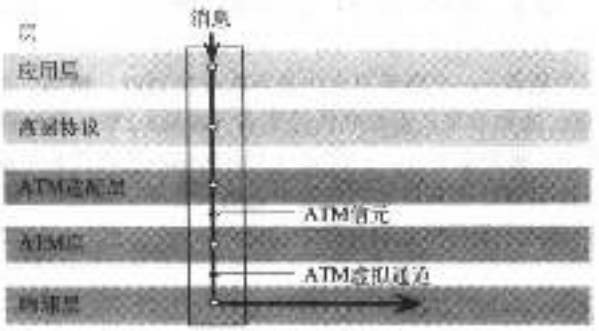


图3-26 ATM协议层

ATM层提供面向连接的服务，传输称为信元的定长数据包。一个连接由虚拟路径上的虚拟通道序列组成。虚拟通道（VC）是在源地址到目的地址的物理路径中的一条链路的两个端点之间的一个逻辑单向连接。虚拟路径（VP）是与交换节点间物理路径相关的一组虚拟通道。虚拟路径的目标是支持一对端点间的半永久连接。在连接建立时，会动态分配虚拟通道。 125

ATM网络的节点扮演下列三种角色：

- 主机：发送和接收信息。
- VP交换机：保存用于显示了输入虚拟路径和输出虚拟路径之间关系的表。
- 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（参见图3-27），即使只有部分数据域中有数据，也发送整个数据域。信元头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传输信元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、信元丢失优先级和信元边界。

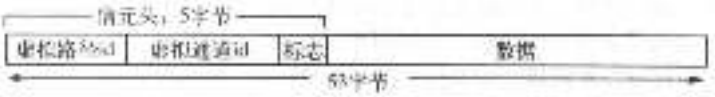


图3-27 ATM信元结构

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3-28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。

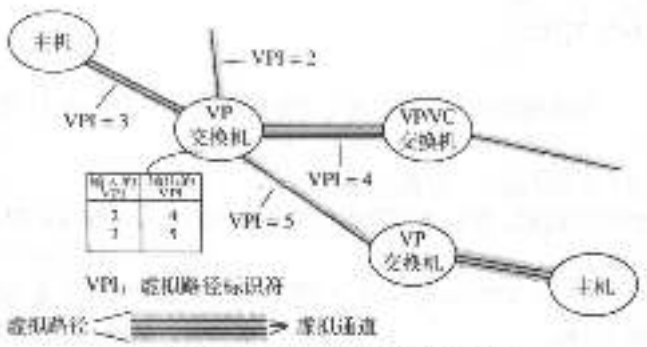


图3-28 在ATM网络中交换虚拟路径

注意，VP和VC标识符是局部定义的。这种方案的优点是无需定义整个网络范围内的全局标识符，那样会是一个很大的数字。全局寻址方案还会带来管理开销，并需要在交换机上的信元头和表保存更多的信息。

ATM提供了低延迟的服务——交换延迟大约是每次交换25μs。例如，当一条消息通过10台交换机时，需要250μs延迟。这和分布式系统的性能要求（参见3.2节）很相符，这表明ATM网络可以支持进程间通信和客户－服务器交互，其性能与现有局域网的性能相当或优于现有局域网的性能。ATM还可以使用，提供保证业务质量的超宽带通道（适合以高达600Mbps的速度传输多媒体数据流）。纯ATM网络的速度甚至可以达到每秒十亿位。

3.6 小结

我们重点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。广域网则基于包交换将数据包通过连接的网络路由到目的地。路由选择是关键问题，目前有不少路由算法，其中距离－向量算法是最基本但却非常有效的一种算法。必须要进行拥塞控制来防止接收方和中间节点的缓冲区溢出。

通过在路由器连接的一组网络上叠加“虚拟”互连网络协议，可以构造互连网络，因特网的TCP/IP协议使因特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相当大的地址空间，并对服务质量和安全性等新的应用需求做了规定。

126
~
127

移动IP支持移动用户进行广域漫游，基于IEEE 802标准的无线LAN支持移动用户进行本地连接。ATM基于有服务质量保证的虚电路提供超宽带的异步通信。

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5000字节的应答。估算在下列情况下，完成请求的时间（其性能假设在后面列出）。

（1）使用无连接（数据报）通信（例如UDP）

（2）使用面向连接（数据报）的通信（例如TCP）

（3）服务器进程与客户进程在同一台计算机上。

其中：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。

（第66页，第105页）

3.2 因特网非常大，任何路由器均无法容纳所有目的地的路由信息，那么因特网路由方案如何处理这个问题呢？（第81页，第97页）

3.3 以太网交换机的任务是什么？它要维护哪些表？（第88页，第113页）

3.4 构造一个类似于图3-5的表，描述当因特网应用与TCP/IP协议组在以太网上实现时，每个协议层中的软件所做的工作。（第77页，第105页，第113页）

3.5 端到端争论[Saltzer et al. 1984]是如何用于因特网的设计的？考虑用虚电路网协议代替IP会如何影响万维网的可行性。（第39页，第79页，第89页）[www.reed.com]

3.6 我们能确保因特网中不会有两台计算机使用同一个IP地址吗？ （第92页）

3.7 对于下面应用层和表示层协议的实现，比较无连接（UDP）与面向连接（TCP）通信。

(1) 虚拟终端访问（例如Telnet）

(2) 文件传输（例如FTP）

(3) 用户位置（例如rwho、finger）

(4) 信息浏览（例如HTTP）

(5) 远程过程调用

（第105页）

3.8 解释在广域网络中，为什么会发生数据包序列到达目的时的顺序与出发时的顺序不同的现象。为什么这种现象在局域网中不可能出现？它可能在ATM网络中出现吗？

（第80页，第114页，第124页） 128

3.9 在Telnet这样的远程终端访问协议中需要解决一个问题，即“Kill 信号”这样的异常事件需要在前面传输的数据之前到达主机。Kill 信号应该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。 （第105页）

3.10 使用网络层广播在以下网络中定位资源有哪些缺点？

(1) 在单个以太网中

(2) 在企业内部网中

以太网组播在何种程度上改善了广播？ （第113页）

3.11 提出一个改善移动IP的方案，以便一个移动设备可以访问Web服务器，该移动设备有时通过移动电话连接到因特网上，而在其他时候通过有线网连接到因特网上。 （第104页）

3.12 说明在图3-7中标号为3的链路断开后，图3-8中路由表的改变序列（根据图3-19中给出的RIP算法）。 （第81页~第85页）

3.13 以图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，而应答包括至少2000字节的HTML。 （第76页，第91页）

3.14 考虑在Telnet远程终端客户中使用TCP。应该如何在客户端缓冲键盘输入？在（1）一个Web服务器；（2）一个Telnet应用；（3）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 1084, Clark 1982]与103页描述的简单算法，比较这两个算法。 （第85页，第107页）

3.15 参照图3-10，构造你工作单位的局域网的网络图。 （第87页）

3.16 描述如何配置防火墙，以保护你的工作单位的局域网。应该拦截哪些进出的请求？

（第108页）

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IP地址的？它是如何将IP地址翻译成以太网地址的？ （第94页）

3.18 防火墙是否可以防止96页描述的服务拒绝攻击？可以使用哪些其他方法处理这样的攻击？

（第96页，第108页） 129

第4章　进程间通信

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

用于因特网中进程间通信的Java API提供数据报和流通信。本章将介绍这两方面的内容，同时讨论它们的故障模型。它们为通信协议提供了可互换的构造成分。

本章将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。

本章还将讨论支持分布式程序中常用的两种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的进程间通信作为实例研究。

130
,
131

4.1　简介

本章和下一章将关注中间件。本章关注图4-1中深色部分标出的组件设计，该层的上层将在第5章中讨论，它涉及将通信集成到编程语言范型中。例如，通过提供远程方法调用（RMI）或远程过程调用（RPC）完成集成。远程方法调用使一个对象能够调用一个远程进程中的对象的方法。远程调用的系统实例有CORBA和Java RMI。类似地，远程过程调用使客户能够调用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，但没有介绍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，给出这两个协议各自的Java接口，同时讨论它们的故障模型。本章的最后一节是实例研究，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能够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，套接字是对目的计算机上的目标进程使用的一个端口的间接引用。

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。流为制造者－消费者通信提供了构造成分[Bacon 2002]。制造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。由制造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为止。在没有可用的数据项时，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，制造者也必须等待。

考虑到不同的计算机可能对简单数据项使用不同的表示方法，本章的第3节将介绍如何将应用程序使用的对象和数据结构翻译成适合的形式，以便在网络上发送消息。第3节还将讨论分布式系 132
统中适合表示对象引用的一种方法。

本章的第4节和第5节将讨论支持客户－服务器和组通信的协议的设计。请求－应答协议用于支持以RMI或RPC方式进行的客户－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，在这种形式下，一组进程中的一个进程将同一消息传送给组中的所有成员进程。

消息传递操作用于构造协议，来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，确认请求消息通常被认为是冗余的，因为应答消息本身就是一个确认。如果一个更专门的协议需要发送方的确认或其他特定的特征，那么它们要提供专门的操作。考虑到用最少量的消息交换来实现协议，那么可行的想法是仅在需要时才增加专门的功能。

4.2 因特网协议的API

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

4.2.1节将回顾2.3.2节提到的消息通信操作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的目的地。4.2.2节将介绍*套接字*，它用于UDP和TCP的应用编程接口中，4.2.3节会讨论UDP和它的Java API，4.2.4节讨论TCP和它的Java API。Java API是面向对象的，但它与最初在Berkeley BSD4.x UNIX操作系统中设计的API很相似，相关的讨论参见4.6节。研究本节程序例子的读者应该参阅Java联机文档或Flanagan[2002]的书，以便得到所讨论的类（在java.net包中）的完整规约。

4.2.1 进程间通信的特征

由send和receive这两个消息通信操作来支持一对进程间进行的消息传递，它们均用目的地和消息定义。为了使一个进程与另一个进程通信，一个进程发送一个消息（字节序列）到目的地，在目的地的另一个进程接收消息。该活动涉及发送进程到接收进程间的数据通信，会涉及两个进 133
程的同步。4.2.3节将给出因特网协议的Java API中的send和receive操作的定义。

同步和异步通信 每个消息目的地与一个队列相关。发送进程将消息添加到远程队列中，接收进程从本地队列中删除消息。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。每次发出一个send操作后，发送进程（或线程）将一直阻塞，直到发送了相应的receive操作为止。每次发送一个receive后，进程（或线程）将一直阻塞，直到消息到达为止。

在异步通信中，send操作是非阻塞型的，只要消息被复制到本地缓冲区，发送进程就可以继续进行其他处理，消息的传递与发送进程并行进行。receive操作有阻塞型和非阻塞型两种形式。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，但它必须通过轮循或中断独立接收缓冲区已满的通知。

在支持多线程的系统环境（如Java）中，阻塞型receive操作的缺点较少，因为在一个线程发出receive操作时，该进程中的其他线程仍然是活动的，到达的消息与接收线程同步的实现很简单是一个优势。非阻塞型的通信看上去更有效，但接收进程需要从它的控制流之外获取到达的消息，这涉及额外的复杂工作。鉴于此，当前的系统通常不提供非阻塞型的receive操作。

消息目的地 第3章解释了在因特网协议中，消息如何被发送到（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。一个端口只能有一个接收者（组播端口是一个例外，见4.5.1节）但可以有多个发送者。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向端口发送消息。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，那么这个服务必须总在该地址所代表的计算机上运行，以保持该服务的有效性。使用下列任何一种方法可避免这种情况，以提供位置透明性：

- 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器（参见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，参见www.cdk4.net/oss）给消息目的地提供了一个与位置无关的标识符，并将它们映射到一个底层地址以便于将消息分发到端口，这种方法使服务能够迁移和重定位。

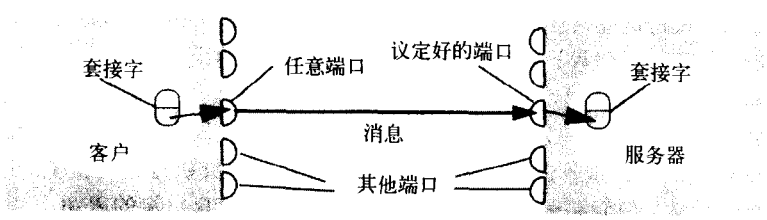
134 替代端口的另一种方法是将消息发给进程，V系统就采用了这种做法[Cheriton 1984]。然而，端口有它的优点，它给一个接收进程提供了多个可选的入口点。在一些应用中，将同一个消息分发给一组进程是非常有用的。因此，一些IPC系统提供了将消息发给目的地组的能力，这里的目的地可以是进程也可以是端口。例如，Chorus[Rozier et al. 1990]提供了端口组。

可靠性 第2章从有效性和完整性角度定义了可靠通信。就有效性而言，如果一个点对点消息服务尽管丢失了“合理”数量的数据包，但仍能保证发送消息，那么该服务就被称为可靠的。相反，哪怕只丢失一个数据包，但消息不能保证发送，那么这个点对点消息服务仍是不可靠的。从完整性而言，到达的消息必须没有损坏，且没有重复。

排序 有些应用要求消息要按发送方的顺序发送，也就是，按发送方发送消息的顺序。与发送方顺序不一致的消息发送会被这样的应用认为是失败的发送。

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 和Macintosh OS。进程间通信是在两个进程各自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它在其上运行的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一个其套接字与该因特网地址和端口号相关的进程接收。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}）可用的端口号供本地进程用于接收消息。任意一个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。每个套接字与某个协议（UDP或TCP）相关。



135 图4-2 套接字和端口

用于因特网地址的Java API 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。该类的用户用域名服务（DNS）的主机名表示计算机（参见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主机名作为参数）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.qmul.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.qmul.ac.uk");

该方法会抛出UnknownHostException异常。注意，类的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，该类的接口与表示因特网地址的字节数无关——在IPv4中是4字节，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，消息可能无法到达目的地。当一个进程发送（send）数据报，另一个进程接收（receive）该数据报时，数据报就会在进程之间传送。要发送或接收消息，进程必须首先创建与一个本地主机的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，该端口应让客户知道，以便客户给该端口发送消息。客户将它的套接字绑定到任何一个空闲的本地端口。Receive方法除了获得消息外，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：接收进程要指定固定大小的用于接收消息的字节数组。如果消息大于数组大小，那么消息在到达时会被截断。底层的IP协议允许数据包的长度最大为2^{16}字节，其中包括消息头和消息本身。然而，在大多数环境下，消息的大小被限制为8KB左右。如果应用程序有大于最大值的消息，那么必须将该消息分割成若干段。通常，由应用（如DNS）决定消息大小——不需要选用很大的值仅只要适用即可。

阻塞：套接字通常提供非阻塞型的send操作和阻塞型的receive操作以进行数据报通信（在某些实现中也会使用非阻塞型receive操作）。当send操作将消息传递给底层的UDP和IP协议后就返回，UDP和IP协议负责将消息传递到目的地。消息到达时被放在与目的端口绑定的套接字队列中。通过该套接字上的下一个receive调用，就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，那么消息就会在目的地被丢弃。

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为止。如果调用receive方法的进程在等待消息时还有其他工作要做，那么应该安排它单独使用一个线程，有关线程的讨论请参见第6章。例如，当服务器从客户端接收到一个消息时，消息会指定要做的工作，这时，服务器将使用单独的线程完成工作或等待其他客户发送的消息。 136

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，使用receive操作的进程不适合无限制地等待下去。为了解决这样的问题，要在套接字上设置超时。选择适当的超时间隔不太容易，但与传输消息所要求的时间相比，它应该更长一些。

任意接收：receive方法不指定消息的来源，而调用receive可获得从任何来源发到它的套接字上的消息。receive方法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某个远程端口和因特网地址，这时，套接字只从那个地址接收消息，并向该地址发送消息。

故障模型 第2章给出了通信通道的故障模型，并从完整性和有效性的角度定义了可靠通信。完整性要求消息不能损坏或重复，利用校验和可保证接收到的消息几乎不会损坏。第2章的故障模型可用于提供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，这可能是因为校验和错误或是在发送端或目的端没有可用的缓

冲区空间造成的。为简化讨论，我们把发送遗漏故障和接收遗漏故障（见图2-11）视为通信通道中的遗漏故障。

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。可以利用确认将一个有遗漏故障的服务构造为可靠传送服务。4.4节将讨论如何在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 对某些应用而言，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。VOIP（Voice Over IP）也运行在UDP上。有时UDP数据报是一个很有吸引力的选择，因为它们没有与保证消息传递相关的开销。开销主要源自以下三个方面：

1）需要在源和目的地存储状态信息。

2）传输额外的消息。

3）发送方的延迟。

137 产生这些开销的原因请参见4.2.4节的讨论。

UDP数据报的Java API Java API通过DatagramPacket和DatagramSocket这两个类提供数据报通信。

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，如下图所示：

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，此时其中一个进程发送，另一个进程接收。

该类还提供了另一个在接收消息时使用的构造函数，它的参数是一个用于接收消息的字节数组和数组长度。DatagramPacket存放接收到的消息、消息长度以及发送套接字的因特网地址和端口。可以从DatagramPacket通过getData方法检索消息。getPort和getAddress方法访问端口和因特网地址。

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，以便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异常。

类DatagramSocket提供以下方法：

- send和receive：这些方法用于在一对套接字之间传送数据报。send的参数是包含消息和它目的地的DatagramPacket实例。receive的参数是一个空的DatagramPacket，用于存放消息、消息的长度和来源。send和receive方法会抛出IOExceptions异常。
- setSoTimeout：该方法用于设置超时。设置超时后，receive方法将在指定的时间内阻塞，然后抛出一个InterruptedIOException异常。
- connect：该方法用于连接到某个因特网地址和远程端口，这时套接字仅能从该地址接收消息并向该地址发送消息。

在图4-3所示的客户程序中，客户先创建一个套接字，然后给位于端口6789的服务器发送消息，并等待接收应答。main方法的参数是消息和服务器的DNS主机名。消息被转换为一个字节数组，DNS主机名被转换为一个因特网地址。图4-4还给出了相应的服务器程序，服务器创建绑定到服务器端口6789的套接字，然后一直等待接收来自客户的请求消息，然后发回同样的消息作为应答。

```
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服务器不断接收请求并将它发回给客户

138
139

4.2.4 TCP流通信

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

消息大小：应用能选择它写到流中和从流中读取的数据量。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将数据作为一个或多个IP数据包传送前，要搜集多少数据。数据到达后按需求传递给应用，如果有必要，应用可以强制数据马上发送。

丢失的消息：TCP协议使用确认方案。以一个简单的方案作为例子（注意，在TCP中没有使用这种方案），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，发送方没有接收到确认信息，则发送方重传该消息。更复杂的滑动窗口方案[Comer2000a]减少了所需的确认消息的个数。

流控制：TCP协议试图匹配读写流的进程的速度。如果对读取流的进程来说写入流的进程太快，那么它会被阻塞直到读取流的进程消化掉足够的数据为止。

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在流上通信之前要先建立连接。一旦建立了连接，进程不需要使用因特网地址和端口就可以读写流。在通信发生前，建立连接涉及客户给服务器发送一个connect请求，然后服务器向客户发送一个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，当一对进程在建立连接时，其中一个进程作为客户，另一个进程作为服务器，但之后它们又是平等的。客户角色涉及创建绑定到端口的流套接字，然后，发出connect请求，在服务器的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，然后等待客户请求连接。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accept（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操作的更多细节将在本章最后的UNIX实例研究中加以介绍。

客户和服务器的套接字对由一对流相连接，每个方向一个流。这样，每个套接字有一个输入
140 流和一个输出流。进程对中的任何一个进程都可以通过将信息写入它的输出流来发送信息给另一个进程，而另一个进程通过读取它的输入流来获得信息。

当一个应用close一个套接字时，表示它不再写任何数据到它的输出流。输出缓冲区中的任何数据被送到流的另一端，放在目的地套接字的队列中，并指明流已断开了。目的地进程能读取队列中的数据，但在队列为空之后进行任何读操作都会返回流结束的标志。当进程退出或失败时，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重要问题。

数据项的匹配：两个通信进程需要对在流上传送的数据的内容达成一致。例如，如果一个进程在流中先写入一个整型数据，后面跟一个双精度型数据，那么另一端的进程必须先读取整型数据，后读取双精度型数据。当一对进程不能在流的使用上正确协作时，读进程在解释数据时可能会出错，或者可能由于流中数据不足而产生阻塞。

阻塞：写入流的数据保存在目的地套接字的队列中。当进程试图从输入通道读取数据时，它将直接从队列中获得数据或一直阻塞直到队列中有可用的数据为止。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，那么将数据写入流的进程可能被TCP流控制机制阻塞。

线程：当服务器接受连接时，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会延误其他客户。在不提供线程的环境中，另一种方法是在试图读取数据前测试来自流的输入是否可用。例如，在UNIX环境中select系统调用便

用于这个目的。

故障模型 为了满足可靠通信的完整性，TCP流使用校验和检查并丢弃损坏的数据包，使用序列号检测和丢弃重复的数据包。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，还是可以保证消息的传输。

但是，如果连接上的数据包丢失超过了限制以及连接一对通信进程的网络不稳定或严重拥塞，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，因为它不能面临各种可能的困难时保证消息的传输。

当连接中断后，使用它的进程如果还试图进行读或写操作，就会接到有关的通知。这会产生下列后果：

- 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故障。
- 通信进程不能区分最近它们发送的消息是否已被接收。

141

TCP的使用 许多经常使用的服务在TCP连接上运行，使用保留的端口号。这些服务包括：

- HTTP： 超文本传送协议用于Web浏览器和Web服务器之间的通信。这部分内容见本章后面的讨论。
- FTP：文件传输协议允许浏览远程计算机上的目录，以及通过连接将文件从一台计算机传输到另一台计算机。
- telnet：telnet利用终端会话访问远程计算机。
- SMTP：简单邮件传输协议用于在计算机之间发送邮件。

TCP 流的Java API TCP 流的Java接口由类ServerSocket和Socket给出。

1）ServerSocket：服务器使用该类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，如果队列为空，它就会阻塞，直到有消息到达队列为止。执行accept的结果是一个Socket实例——该套接字可用于访问与客户通信的流。

2）Socket：该类可供连接的一对进程使用。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的套接字，而且将套接字连接到指定的远程计算机和端口号。如果主机名错误它会抛出UnknownHostException异常，如果发生IO错误，它会抛出IOException异常。

Socket类提供了getInputStream和getOutputStream方法用于访问与套接字相关的两个流。这些方法的返回类型分别是InputStream和OutputStream，即定义了读写字节的方法的抽象类。返回值可作为合适的输入输出流的构造函数的参数。我们的例子使用DataInputStream和DataOutputStream，它们允许简单数据类型的二进制表示能以与机器无关的方式读写。

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和服务器的DNS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。它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中的服务器程序打开其服务器端口（7896）的服务器套接字，监听connect请求。当有请求到达时，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将其写回。

因为消息由串组成，客户进程和服务器进程使用DataOutputStream的writeUTF方法将消息写入输出流，使用DataInputStream的readUTF方法从输入流中读取消息。UTF-8是表示串的特定格式编码，参见4.3节的描述。

当一个进程关闭它的套接字后，它将不再能够使用它的输入和输出流。数据的目的进程能从它的队列中读取数据，但在队列为空后进行读操作会产生EOFException异常。试图使用一个关闭的套接字或向一个中断的流中写信息都会产生IOException异常。

```
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

```
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

4.3 外部数据表示和编码

存储在运行的程序中的信息都表示成数据结构，如相互关联的对象集合，而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“打平”（转换成字节序列），到达目的地后重构。在消息中传送的单个简单数据项可以是不同类型的数据值，不是所有的计算机以同样的顺序存储整数这样的简单值。浮点数的表示也随体系结构的不同而不同。表示整数的顺序有两种方法：所谓的*大序法*排序，即最高有效字节排在前面；和*小序法*排序，即最高有效字节排在后面。另一个问题是用于表示字符的代码集，例如，系统（如UNIX）上的大多数应用使用ASCII字符编码，每个字符占一字节，但是Unicode标准可以表示许多不同语言的文字，每个字符占两字节。

下列方法可用于使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。
- 值按照发送端的格式传送，同时传送所使用格式的标志，如果需要，接收方会转换该值。

142 ~ 144

注意，字节本身在传送过程中不改变。为了支持RMI或RPC，任何能作为参数传递或作为结果返回的数据类型必须被“打平”，单个的简单数据值以一致的格式表示。表示数据结构和简单值的一致的标准称为*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的过程。*解码*是在消息到达后分解消息，在目的地生成相等的数据项的过程。因此，编码是将结构化数据项和简单值翻译成一个外部数据表示。类似地，解码是从外部数据表示生成简单值，并重建数据结构。

我们将讨论三种外部数据表示和编码的方法：

- CORBA的公共数据表示，它涉及在CORBA的远程方法调用中能作为参数和结果传送的结构化类型和简单类型的外部表示。它可用于多种编程语言（参见第20章）。
- Java的对象序列化，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“打平”和外部数据表示。它仅用于Java。
- XML（即扩展标记语言），它定义了表示结构化数据的文本格式。它原本用于包含文本自描述型的结构化数据的文档，例如可从Web访问的文档。但它现在也用于在Web服务中被客户和服务器交换的消息中的数据（参见第19章）。

在前两种情况下，编码和解码活动均由中间件层完成，不涉及任何一方的应用程序员。即使在XML的情况下（XML是文本的，因此更容易处理编码），编码和解码软件也对所有平台和编程环境可用。因为编码要求考虑组成组合对象的简单组件的表示细节，所以如果手工完成该过程，那么整个过程很容易出错。简洁性是设计自动生成型编码程序要考虑的另一个问题。

在前两个方法中，简单数据类型被编码成二进制形式。在第三个方法（XML）中，简单数据类型表示成文本。通常，数据值的文本表示将比等价的二进制表示更长。4.4节描述的HTTP协议是文本方法的另一个例子。

与编码方法设计有关的另一个问题是被编码数据是否应该包括与其内容的类型有关的信息。例如，CORBA的表示只包括所传送的对象的值，不包含它们的类型。另一方面，Java序列化和XML都包括了类型信息，但表示方式不同。Java把所有需要的类型信息放到序列化后的格式中。但XML文档可以指向名字（和类型）的外部定义集合，即*名字空间*。

145

虽然我们对RMI、RPC的参数和结果的外部数据表示感兴趣，但将数据结构、对象或结构化文档转换成适合消息传送或文件存储的格式更为常用。

4.3.1 CORBA的公共数据表示

CORBA的公共数据表示（Common Data Representation, CDR）是CORBA 2.0[OMG 2004a]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。其中有15个简单类型，包括short（16比特）、long（32比特）、unsigned short、unsigned long、float（32比特）、double（64比特）、char、boolean（TRUE、FALSE）、octet（8比特）和any（它可表示任何基本类型或构造类型），此外还有一套复合类型，参见图4-7。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（无符号长整型），后面依次是元素
string	长度（无符号长整型），后面依次是字符（可以有宽字符）
array	依次是数组元素（不用指定长度，因为它是固定的）
struct	按组成部分声明的顺序表示
enumerated	无符号长整型（值按照声明的顺序指定）
union	类型标签，后面是所选中的成员

图4-7 结构化类型的CORBA CDR

简单类型：CDR定义了大序法排序和小序法排序的表示。值按发送端消息中指定的顺序传送，接收端如果要求不同的顺序就要进行翻译。例如，16比特short类型数据在消息中占两个字节，若用大序法排序，最高有效位占第一个字节，最低有效位占第二个字节。每个简单类型值根据它的大小顺序放在字节序列中。假设字节序列的下标最小为零，那么n字节大小（其中$n=1, 2, 4, 8$）的简单类型值将附加到字节流序列中为n的倍数的下标处，浮点值遵循IEEE标准，其中符号、指数和小数部分按大序法依次放在字节0～n处；按小序法排序则要反过来放。字符用客户和服务器均同意的代码集表示。

146

结构化类型：组成每个结构化类型的简单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，它包含三个域，三个域的类型分别是string、string和unsigned long。图中给出了每行有4个字节的字节序列。每个串的表示由一个表示长度的unsigned long，后跟串中的字符组成。为简单起见，我们假设每个字符只占一个字节。变长数据用零填充，以便形成标准格式，从而比较编码数据或它的校验和。注意，每个unsigned long占四个字节，其开始位置在一个4的倍数的下标处。图4-8没有区分大序法排序和小序法排序。虽然图4-8中的例子比较简单，但CORBA CDR能表示任何不使用指针的由简单类型和结构化类型组成的数据结构。

字节序列中的下标	← 4字节 →	注释
0～3	5	串的长度
4～7	"Smit"	'Smith'
8～11	"h___"	
12～15	6	串的长度
16～19	"Lond"	'London'
20～23	"on__"	
24～27	1934	unsigned long

打平的格式表示值为{'Smith','London',1934}的Person结构。

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中指定[Srinivasan 1995b]，其描述见www.cdk4.net/ipc。它由Sun公司开发，用于Sun NFS中客户和服务器之间的消息交换（参见第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共识。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 根据在消息中传送的数据项类型的规约，可以自动生成编码操作。数据结构的类型和基本数据项类型用CORBA IDL描述（见20.2.3节），IDL提供了描述RMI方法的参数类型和结构类型的表示法。例如，我们可以用CORBA IDL描述图4-8中消息的数据结构： 147

```
struct Person{
    string name;
    string place;
    unsigned long year;
};
```

CORBA接口编译器（参见第5章）根据远程方法的参数类型和结果类型定义为参数和结果生成适当的编码和解码操作。

4.3.2 Java 对象序列化

在Java RMI中，对象和简单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，与CORBA IDL中定义的Person struct作用相当的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
}
```

上面的类表明它实现了Serializable接口，该接口没有方法。表明一个类实现了Serializable接口（该接口在java.io包中提供）意味着它的实例能被序列化。

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打平成适合于磁盘存储或消息传送的串行格式，例如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的状态。它假设进行解序列化的进程事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。

类的信息由类名和版本号组成。当类有大的改动时要修改版本号。它可由程序员设置或自动根据类名、它的实例变量、方法和接口的名字的散列值计算，解序列化对象的进程能检查它的类版本是否正确。

Java对象可以包含对其他对象的引用。当对象序列化时，它引用的所有对象也随它一起序列化，以确保对象在目的地重构时它引用的对象也能恢复。引用被序列化成*句柄*——在这种情况下，句柄 148

是在序列化格式内对一个对象的引用，例如句柄可以是正整数序列中的下一个数字。序列化过程必须确保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能写一次——在对象第二次出现及之后再出现时，写入句柄而不是对象。

为了序列化一个对象，要写出它的类信息，随后是实例变量的类型和名字；如果实例变量属于新的类，那么要写出它们所属的新类的类信息，随后是新类的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为止。每个类都有一个句柄，没有一个类会多次写入字节流——在需要的地方会写入句柄。

整型、字符型、布尔、字节和长整型这样的简单类型的实例变量的内容可用ObjectOutputStream类的方法写成一个可移植的二进制格式。字符串和字符使用writeUTF方法写入，该方法使用通用传输格式（UTF-8），UTF依旧用一个字节表示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);

序列化后的格式见图4-9，图中省略了完整序列化格式中的句柄的值和表示对象、类、串和其他对象的类型标识符的值。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，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

真正的序列化格式还包含类型标识符；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。要从数据流中解序列化一个对象，应在流上打开一个ObjectInputStream，用它的readObject方法重构原来的对象。这一对类的使用与图4-5和图4-6中说明的DataOutputStream和DataInputStream类似。

远程调用的参数和结果的序列化及解序列化通常由中间件自动完成，不需要应用程序员参与。如果有特殊需求，程序员可以自己编写读写对象的方法。详细内容请参阅有关对象序列化的教程[java.sun.com II]了解如何自己编写方法并获取Java序列化的更多信息。程序员修改序列化效果的另一种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）的引用就不应该被序列化。

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及类型）的能力。反射实现了根据类名创建类，以及为给定的类创建具有给定参数类型的构造函数。反射使得以完全通用的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那样为每种对象类型生成特殊的编码函数。关于反射的更多信息请参见Flanagan[2002]。

Java对象序列化使用反射找到要序列化的对象的类名，以及该类的实例变量的名字、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。然后用类名创建一个新的构造函数，它具有与指定在序列化格式中的类型相应的类型。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 可扩展标记语言

可扩展标记语言（Extensible Markup Language, XML）是万维网联盟（W3C）定义的可在Web上通用的标记语言。通常，术语标记语言指的是一种文本编码，用于表示正文和关于正文结构或外观的细节。XML和HTML都是从一种非常复杂的标记语言SGML（标准化的通用标记语言）[ISO 8879]派生出来的。HTML（见1.3.1节）用于定义Web页面的外观，而XML用于编写Web上的结构化文档。

XML数据项以“标记”串做标签，标记用于描述数据的逻辑结构，并将属性－值对与逻辑结构关联起来。也就是说，在XML中，标记与它们围起来的正文结构相关，而在HTML中，标记指定浏览器如何显示正文。关于XML规约，请参见W3C提供的关于XML的网页[www.w3.org VI]。

XML用于实现客户与Web服务的通信以及定义Web服务的接口和其他属性。不过，XML也可用于其他方面。它可用于存档和检索系统——尽管一个XML存档文件比一个二进制文件要大，但它的优势在于可在任意一台计算机上阅读。其他使用XML的例子包括用户界面的规约和操作系统中对配置文件的编码。

XML是可扩展的，这意味着用户能定义自己的标记，这点与HTML不同，HTML只能使用固定的标记集合。如果打算将一个XML文档用于多个应用，那么标记的名字必须在这些应用中达成一致。例如，客户通常使用SOAP消息与Web服务通信。SOAP（参见19.2.1节）具有XML格式，其中的标签专门用于Web服务和它的客户。 149 ~ 150

一些外部数据表示（如CORBA CDR）不一定是自描述的，因为它假设客户和服务器对要交换的消息具有先验的知识，知道消息所包含的信息的顺序和类型。不过，XML原本希望供多个应用使用，并可用于不同的目的。提供标记以及定义标记含义的名字空间就是为了使上述目的成为可能。另外，标记的使用使得应用可选择它需要处理的部分：增加与其他应用相关的信息，并不影响原有应用。

因为XML文档是文本形式的，所以人人可读。通常，大多数XML文档由XML处理软件生成并读取，但是读XML的能力在出错的时候更有用。另外，文本的使用使得XML独立于某个平台。使用文本（而不是二进制表示）和标记会使消息变得更大，因而需要更长的时间处理和传输，也需要更大空间进行存储。19.2.4节比较SOAP XML格式的消息和CORBA CDR格式消息的效率。不过，文件和消息能被压缩——HTTP 1.1允许对数据进行压缩，从而节省传输的带宽。

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```

图4-10 用XML定义的Person结构

XML元素和属性 图4-10给出了Person结构的XML定义，这个结构用于说明CORBA CDR和Java中的编码功能。它说明XML由标记和字符数据组成。字符数据（例如，Smith或1934）是实际的数据。类似HTML，XML文档的结构由包含在一对尖括号内的标记定义。在上面的例子中，<name>和<place>都是标记。与HTML一样，良好的布局通常可以提高可读性。XML中注释的表示方法和HTML一样。

元素：XML中的元素由匹配的开始标记和结束标记包围的字符数据组成。例如，图4-10中的一个元素由包含在<name>…</name>标记对中的数据Smith组成。注意，具有<name>标记的元素包含在具有<person id=“123456789”>…</person>标记对的元素中。一个元素包含其他元素的能力使得XML具有表示层次数据的能力——这是XML一个非常重要的方面。一个空标记没有内容，用/>表示结束（而不是用>）。例如，<person>…</person>标签可以包括一个空标记<european/>。

属性：一个开始标记可以选择性地包含关联的属性名和属性值对，例如上述的id="123456789"。属性的语法与HTML的语法一样，其中属性名后面跟着一个等号和用引号括起来的属性值。多个属 151

性值用空格分开。

把哪些项表示成元素哪些项表示成属性要进行选择。元素通常是一个数据容器，而属性用于标记数据。在我们的例子中，123456789可以是应用程序使用的标识符，而name、place或year是需要显示的。如果数据包含子结构或多行信息，那么它必须被定义成元素，简单值定义成属性。

名字：XML中的标记名和属性名通常以字母开始，也可以以下划线或冒号开始。名字首字符后可以是字母、数字、连字符、下划线、冒号或句号。名字中的字母是区分大小的，以xml开始的名字是保留字。

二进制数据：XML元素中所有的信息必须被表示成字符数据。但问题是，我们如何表示加密的元素或安全的散列值？这两者将用于19.5节介绍的XML安全性中。答案是它们可以用base64表示法表示[Fred and Borenstein 1996]，这种方法仅用字母数字和具有特殊意义的+、/、=表示。

解析和良构的文档 XML文档必须是良构的，即它的结构必须符合规则。一个基本的规则是每个开始标记都要有一个匹配的结束标记。另一个基本的规则是所有标记要正确嵌套，例如，<x>…<y>…</y>…</x>是正确的，而<x>…<y>…</x>…</y>是不正确的。最后，每个XML文档必须有一个包围其他元素的根元素。这些规则对实现XML文档的解析器而言是非常简单的。当解析器读到一个非良构的XML文档时，它将报告一个致命的错误。

CDATA：XML解析器通常分析元素的内容，因为它们可能包含嵌套的结构。但如果文本需要包含一个尖括号或引号，那么它必须以特殊的方式表示，例如，&lt表示左尖括号。但如果因为某种原因，不需要解析某个部分，例如，它包含了特殊的字符，那么它可以表示成CDATA。例如，如果一个场地的名字中包含了一个撇号，那么可以用下面两种方式之一指定：

<place> King &apos Cross </place>

<place> <![CDATA[King's Cross]]></place>

XML*序言*：每个XML文档必须在它的第一行包含一个*序言*。序言必须至少指定使用的XML版本（当前是1.0）。例如：

<? XML version = "1.0" encoding="UTF-8" standalone = "yes"?>

第三个属性用于说明文档是独立的还是依赖于外部定义的。

编码方式：序言也必须指定编码方式（默认编码是UTF-8，参见4.3.2节的解释）。术语*编码方式*指的是用于表示字符的代码集——ASCII是我们最熟知的例子。注意，在XML序言中，ASCII被指定成us-ascii。其他可能的编码方式包括ISO-8859-1（或Latin-1），它也是一种8位编码方式，前128个值是ASCII字符，其他字符用于表示西方欧洲语言中的字符。其他8位编码用于表示其他字母表（如希腊语或斯拉夫语）。

152

XML名字空间 通常情况下，名字空间为设定名字的作用域提供了一个手段。一个XML名字空间是一个名字集合，通过URL引用，用于一组元素类型和属性。其他XML文档可通过引用名字空间的URL使用该名字空间。

利用XML名字空间的元素将名字空间指定成名为xmlns的属性，该属性的值是一个URL，指向包含名字空间定义的文件。例如：

xmlns:pers = "http://www.cdk4.net/person"

xmlns后面的名字（这里是pers）可以作为一个前缀，指向某个名字空间中的元素，如图4-11所示。pers前缀在person元素中被绑定到http://www.cdk4.net/person。一个名字空间的应用范围为开标记和闭标记所确定的范围内，除非被一个内含的名字空间定义取代。一个XML文档可能定义有多个不同的名字空间，每个名字空间用其唯一的前缀引用。

名字空间的约定允许一个应用程序利用不同名字空间中的多个外部定义，而不存在名字冲突的风险。

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1934 </pers:year>
</person>
```

图4-11 在Person结构中使用名字空间

XML模式 一个XML模式[www.w3.org VIII]定义了在文档中出现的元素和属性、元素如何嵌套、元素的顺序及个数、元素是否为空或能否包含文本等。对于每个元素，它定义了类型和默认值。图4-12给出了一个模式的例子，它定义了图4-10所示的person结构的XML定义的数据类型和结构。

目的是单个模式定义可被多个不同的文档共享。一个XML文档遵循某个模式定义，那么可通过这个模式进行验证。例如，SOAP消息的发送者可以使用XML模式编码消息，接收者将用相同的XML模式进行验证并解码消息。

文档类型定义：文档类型定义（即DTD）[www.w3.org VI] 是作为XML 1.0规约的一部分提供的，用于定义XML文档的结构，目前仍被广泛使用。DTD的语法与XML的其他部分不一样，其描述能力比较有限。例如，它不能描述数据类型，它的定义是全局的，因此元素名不能重复。DTD不用于定义Web服务，尽管它们仍可以用于定义由Web服务传输的文档。

访问XML的API 大多数常用的编程语言有可用的XML解析器和生成器。例如，将Java对象输出成XML（即编码）的Java软件和从类似结构中创建Java对象（即解码）的软件。Python编程语言有类似的软件用于Python数据类型和对象。

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
    <xsd:element name= "person" type ="personType" />
    <xsd:complexType name="personType">
        <xsd:sequence>
            <xsd:element name = "name" type="xs:string"/>
            <xsd:element name = "place" type="xs:string"/>
            <xsd:element name = "year" type="xs:positiveInteger"/>
        </xsd:sequence>
        <xsd:attribute name= "id" type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

图4-12 用于Person结构的XML模式

4.3.4 远程对象引用

本节的内容仅适用于诸如Java和CORBA这样的支持分布对象模型的语言，与XML无关。

客户调用远程对象中的一个方法时，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对象具有要调用的方法。远程对象引用是远程对象的标识符，它在整个分布式系统中有效。远程对象引用在调用消息中传递，以指定调用哪一个对象。第5章将介绍远程对象引用也作为远程方法调用的参数传递并作为远程方法调用的结果返回，第5章还将说明每个远程对象有一个远程对象引用，并通过比较远程对象引用确定它们是否指向同一个远程对象。现在我们讨论远程对象引用的外部表示。

远程对象引用必须以确保空间和时间唯一性的方法生成。通常，在远程对象上有许多进程，所以远程对象引用在分布式系统的所有进程中必须是唯一的。即使在删除与给定远程对象引用相

关的远程对象后，该远程对象引用也不能被重用，因为潜在的调用者还可能保留着过期的远程对象引用，记住这一点非常重要。试图调用已删除对象应该产生一个出错信息而不应该允许访问另一个对象。

有几个方法可以确保远程对象引用是唯一的。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，本地对象编号就增加一。

端口号与时间一起在计算机上产生唯一的进程标识符。利用这种方法，远程对象引用可用图4-13所示的格式表示。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，远程对象引用可以作为远程对象的地址。换句话说，调用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。

32比特	32比特	32比特	32比特	
因特网地址	端口号	时间	对象编号	远程对象接口

图4-13 远程对象引用的表示

为了使远程对象在不同计算机的不同进程中重定位，远程对象引用不应该作为远程对象的地址使用。20.2.4节讨论了远程对象引用的一种格式，它允许对象在它的生命周期中在不同的服务器上被激活。

第10章将要描述的对等覆盖网络使用完全与位置无关的远程对象引用。消息通过一个分布式路由算法路由到资源所在地。

图4-13中远程对象引用的最后一个域包含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节中解释。

4.4 客户－服务器通信

这种形式的通信用于支持典型客户－服务器交互中的角色和消息的交换。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，直到收到来自服务器的应答为止。请求－应答通信也是可靠的，因为来自服务器的应答是对客户的有效确认。异步请求－应答通信是另一种方法，这种方法当客户能在稍后取回应答时很有用——参见6.5.2节。

虽然当前许多实现使用TCP流，但下面根据用于UDP数据报的Java 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：

153～155

- 确认是冗余的，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，需要建立与两对额外消息有关的连接。
- 对大多数仅传递少量参数和结果的调用，流控制是冗余的。

请求－应答协议 下列协议基于三个通信原语：doOperation、getRequest和sendReply，如图4-14所示。大多数的RMI和RPC系统由类似的协议支持。此处描述的协议经过裁剪可支持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。

这种特殊设计的请求－应答协议将请求和应答匹配起来。它可提供一定的传递保证。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4-15概述了这三个通信原语。

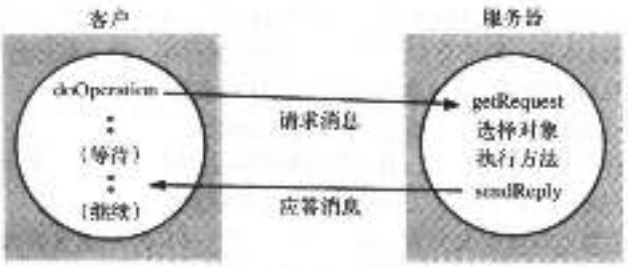


图4-14 请求－应答通信

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。

图4-15 请求－应答协议的操作

客户使用doOperation方法调用远程操作。它的参数指定了远程对象和要调用的方法以及该方法要求的额外的信息（参数）。它的结果是一个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4-13所示的格式表示远程对象引用。该类可提供方法来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作为参数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为止。

GetRequest被服务器进程用于获取服务请求。如图4-14所示，当服务器调用指定对象的方法时，它使用sendReply发送应答消息给客户。当客户接收到应答消息，原来的doOperation不再阻塞，继续执行客户程序。

在请求消息或应答消息中传送的信息如图4-16所示。第一个域指出消息是一个请求消息还是一个应答消息。第二个域RequestId包含一个消息标识符。客户的doOperation为每个请求消息生成一个RequestId，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延迟到达的请求结果。第三个域是按图4-10所示格式编码的远程对象引用。第四个域是要调用的方法的标识符。例如，可以对接口中的方法用1，2，3，……编号。如果客户和服务器均使用支持反射的语言，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。

messageType	int（0＝请求，1＝应答）
requestId	int
objectReference	远程对象引用
methodId	整数或方法
arguments	字节数组

图4-16 请求－应答消息的结构

消息标识符 任何涉及消息管理以提供额外的诸如可靠消息传递或请求－应答通信这样的性质的机制均要求每个消息具有唯一的消息标识符，以便消息被引用。消息标识符由两部分组成：

156～157

1）requestId，由发送进程从一个不断增加的整数序列中取出。

2）发送进程的一个标识符，例如它的因特网地址和端口。

第一部分确保标识符在发送方唯一，第二部分确保标识符在分布式系统中唯一。（第二部分可以单独获得，例如如果使用UDP，可以从接收到的消息中获得。）

当requestId的值达到无符号整数的最大值（例如，$2^{32}-1$）时，就重置为0。这里唯一的限制是消息标识符的生命周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 如果三个原语doOperation、getRequest和sendReply都在UDP数据报上实现，那么它们都可能出现同样的通信故障，即

- 存在遗漏故障。
- 消息不能保证按发送方顺序到达。

此外，协议还有可能遇到进程故障（参见2.3.2节），我们假设进程会崩溃。也就是说，当它们停止时，它们会一直停止下去——它们不产生拜占庭行为。

考虑到有服务器故障、丢失请求消息或应答消息的情况，doOperation在等待服务器应答消息的时候使用超时。超时后采取的动作取决于要提供的传递保证。

超时 超时后doOperation能发生不同的行为。最简单的是马上从doOperation返回，并给客户一个标志提示doOperation操作失败。但这不是常用的方法，因为超时可能是由于请求或应答消息丢失造成的。此时，操作已经完成。为了弥补丢失消息可能带来的不便，doOperation重复发送请求消息，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造成的为止。最后，当doOperation返回时，它通知客户出现了一个“没有接收到结果”的异常。

丢弃重复的请求消息 一旦请求消息被重传，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，协议要能识别具有相同请求标识符（来自同一客户的）的后续消息，并过滤掉重复的消息。如果服务器还没有发送应答，也不需要采取特别的动作——服务器在执行完操作后会传递应答的。

丢失应答消息 如果服务器在收到一个重复的请求时已经发送了应答，那么它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。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，每次执行的效果和执行一次的效果一样。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，

158

将一个数据项追加到一个序列不是一个幂等操作，因为每次执行它都会扩展这个序列。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同一个操作。

历史 对那些要求不重新执行操作只重传应答的服务器，可以使用历史。“历史”这个词指包含已经传送的（应答）消息的记录的结构。历史中的项包含一个请求标识符、一条消息和要发送到的客户标识符。它的目的是使服务器在客户进程发出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。除非服务器能够说明什么时候消息不再需要重传，否则历史会变得很大。

虽然客户每次只能发出一个请求，但服务器可以将每个请求解释成它对前一个应答的确认。因此，历史只需要包含发送给每个客户的上一个应答消息。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，它不能确认它已接收到上一个应答时——因此历史中的消息在经过一段时间后会被丢弃。

RPC交换协议 三个有不同通信故障语义的协议可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4-17总结了在这些协议中传递的消息。在R协议中，单个Request消息由客户发送到服务器。当远程方法没有返回值而且客户不要求对执行的方法进行确认时可使用R协议。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。该协议在UDP数据报之上实现，因此会出现与UDP数据报相同的通信故障。

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4-17 RPC交换协议

因为RR协议基于请求－应答协议，所以它用于大多数客户－服务器交互。它不需要专门的确认消息，因为服务器的应答消息被认为是对客户请求消息的确认。同样，客户的后续调用可以被认为是对服务器应答消息的确认。正如我们已经看到的，源于UDP数据报的通信故障可以通过重传请求、重复过滤和在重传记录中保存应答来屏蔽。 159

RRA协议基于三种消息的交换：请求－应答－确认应答。acknowledge reply消息包含被确认应答消息的requestId，这会造成服务器丢弃历史中的数据项。确认消息中的requestId的到达将被解释成对所有接收到的小于requestId的应答消息进行确认，所以确认消息的丢失是无害的。虽然数据交换涉及一个额外的消息，但它不需要阻塞客户，因为确认在应答到达客户之后传递，但它确实要进行处理并使用网络资源。本章的习题4.23将给出一种对RRA协议进行优化的方法。

使用TCP流实现请求－应答协议 介绍数据报时曾提到，决定接收数据报使用的缓冲区的大小是很困难的。在请求－应答协议中，决定服务器接收请求消息的缓冲区和客户接收应答的缓冲区也很困难。数据报的有效长度（通常是8KB）对于透明RMI系统而言并不是足够的，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，并且能够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处理历史。另外，流控制机制可以在不用采取特殊手段来避免接收方的崩溃的情况下传输大的参数和结果。因此，为请求－应答协议选择TCP协议，是因为它能简化请求－应答协议的实现。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。当一个应答消息紧随在请求消息之后时，TCP确认消息的开销也将减少。

有时，应用并不需要由TCP提供的所有设施，一个更有效的经过特别裁剪的协议可在UDP上实现。例如，我们提到过，Sun NFS并不需要允许消息的大小不限，因为它在客户和服务器之间传输固定长度的文件块；另外，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，同时也没有必要维护历史。

HTTP：请求－应答协议的例子 第1章介绍了超文本传输协议（HTTP），Web浏览器客户使用它给Web服务器发送请求并从服务器接收应答。扼要地说，Web服务器管理以不同方式实现的资源： 160

• 作为数据实现的资源，例如，HTML页面的文本、一个图像或applet的类。

• 作为程序实现的资源，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符。

HTTP是一个协议，它指定了请求－应答所需的消息、方法、参数和结果以及在消息中表示（编码）它们的规则。它支持一个固定的可用于所有资源的方法集（包括GET、PUT、POST等）；上面的那些协议的每个对象有它自己的方法，在这一点上HTTP与之不同。除了在Web资源上调用

方法之外，HTTP协议能使用内容协商和口令形式的认证。

内容协商：客户请求包含什么数据表示能接收（例如语言或介质类型）之类的信息，这使得服务器能选择最适合用户的数据表示。

认证：证书和质询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，服务器的应答包含应用于该资源的质询。第7章将解释质询。当客户接收到一个质询时，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。在该协议的最初版本中，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。
- 连接关闭。

然而，为每个请求－应答交互建立和关闭连接的开销太大，它不仅会使服务器过载，而且在网络上发送的消息也太多。考虑到浏览器通常会向同一个服务器发多个请求，所以HTTP协议在之后的版本（HTTP 1.1，见RFC 2616[Fielding et al. 1999]）中使用了*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。客户或服务器在任何时候通过发送一个标志给对方便可关闭永久连接。服务器会关闭空闲了一段时间的永久连接。客户在发送请求期间，有可能会从服务器收到消息说连接关闭了。这时，如果涉及的操作是幂等的，无需用户介入，浏览器就会重发请求。例如，下面描述的GET方法是幂等的。当涉及非幂等操作时，浏览器应该与用
161 户协商下一步做什么。

请求和应答可以被编码成ASCII文本串放入消息中，但资源能表示成字节序列，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。在这种情况下，文本表示不会过度增加消息的长度。

数据实现的资源在参数和结果中具有类似MIME的结构。在RFC 2045 [Freed and Borenstein 1996]中给出的多用途因特网邮件扩展（Multipurpose Internet Mail Extension, 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，以便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服务器要应用的一个资源的方法的名称和该资源的URL。应答报告请求的状态。请求和应答也会包含资源数据、表单的内容或在Web服务器上运行的一个程序资源的输出。有以下几种方法：

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指定的数据；如果URL指向程序，那么Web服务器运行该程序，为客户返回程序结果。可向URL添加参数，例如，GET可以将一个表单的内容作为参数发送给一个cgi程序。可以根据上次资源被修改的日期选择是否进行GET操作，也可以配置GET操作获得部分数据。

HEAD：该请求与GET相似，只是它不返回任何数据。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。在数据上进行的处理和URL中指定的程序的功能有关。该方法用于：

- 为servlet或cgi程序这样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存储起来并用给定的URL作为其资源标识符，请求中的数据或者作为已有资源的修改或者作为一个新的资源。

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用在给定URL上的方法（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，该请求消息用于诊断。

上述请求可被一个代理服务器截获（参见2.2.2节），对GET和HEAD的响应可由代理服务器缓存。

消息内容 请求消息指定了方法名、资源的URL、协议版本、若干消息头和可选的消息体。图4-18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。

方法	URL或路径名	HTTP版本	头部	消息体
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/1.1		

图4-18 HTTP请求消息

对代理的请求需要给出绝对URL，如图4-18所示。对源服务器（源服务器是资源所在地）的请求指定了一个路径名，并在头部的host域给出源服务器的DNS名字。例如：

GET /index.html HTTP/1.1

Host: www.dcs.qmul.ac.uk

通常，头部域包含请求修改者和客户信息，例如最近一次修改资源或可接受的内容类型（例如，HTML文本、音频或JPEG）的日期。授权域用于以证书形式提供客户的证明，以表明他们访问资源的权力。

应答消息指定了协议版本、状态码、“理由”、若干消息头和可选的消息体，如图4-19所示。状态码和理由在成功时提供一个报告，否则在执行请求时，状态码是一个由程序解释的三位整数，理由是一个可被用户理解的文本短语。头部域用于传递有关服务器或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，并且在头部域包含一个质询。一些返回的状态有十分复杂的效果。特别是，如果状态响应为303则浏览器要查看另一个URL，该URL位于应答的头部域中。它用于由POST请求激活程序的应答中，这时，程序要将浏览器重新引导到选中的资源。

HTTP版本	状态码	理由	头部	消息体
HTTP/1.1	200	OK		资源数据

图4-19 HTTP应答消息

162 ~ 163

请求消息或应答消息中的消息体包含与请求中指定的URL相关的数据。消息体有它自己的指定了有关数据信息的头部，这些信息有消息体的长度、Mime类型、字符集、内容编码和上一次修改时间。Mime类型域指定数据的类型，例如image/jpeg或text/plain，内容编码域指定所使用的压缩算法。

4.5 组通信

如果为了提供容错能力或为了提高可用性而将一个服务实现为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。组播操作是更合适的方式，这是一个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，组的成员对发送方通常是透明的。组播的行为有很多种可能情况。最简单的组播不提供消息传递保证和排序保证。

组播消息为构造具有下列特征的分布式系统提供了基础设施：

1）*基于服务复制的容错*：一个复制服务由一组服务器组成。客户请求被组播到组的所有成员，每一个成员都执行相同的操作。即使一些成员出现故障，仍能为客户提供服务。

2）*在自发网络中找到发现服务器*：16.2.1节将讨论自发网络中的发现服务。客户和服务器能使用组播消息找到可用的发现服务，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*：复制数据能提高服务的性能。在某些情况下，数据的复本放在用户的计算机上。每次数据改变，新的值便被组播到管理复本数据的各个进程。

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，当一个新消息被贴到某个新闻组时，新闻系统可通知感兴趣的用户。

我们先介绍IP组播，然后回顾上述使用组通信的要求，看看IP组播能满足其中的哪些要求。对
164 于不能满足的要求，我们在组通信协议中提出了IP组播已有特征之外的更多特征。

4.5.1 IP组播——组通信的实现

本节讨论IP组播，并通过MulticastSocket类给出组播的Java API。

IP组播 *IP组播*在网际协议IP的上层实现。注意，IP数据包是面向计算机的——端口属于TCP和UDP层。IP组播使发送方能够将单个IP数据包传送给组成组播组的一组计算机。发送方不清楚每个接收者身份和组的大小。*组播组*由D类因特网地址（参见图3-15）指定，即在IPv4中，前4位是1110的地址。

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，计算机可以在任何时间加入或离开，计算机也可以加入任意数量的组。可以无需成为成员就向一个组播组发送数据报。

在应用编程级，IP组播只能通过UDP可用。应用程序通过发送具有组播地址和普通的端口号的UDP数据报完成组播。应用通过将套接字加入到组来加入一个组播组，使得它能从组接收消息。在IP层，当一个或多个进程具有属于一个组播组的套接字时，该计算机属于这个组播组。当一个组播消息到达计算机时，消息复本被转发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。下列特点是IPv4特有的：

组播路由器：IP数据包既能在局域网上组播也能在因特网上组播。本地的组播使用了局域网（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到其他成员所在网络的路由器上，再通过路由器组播到本地成员。为了限制组播数据报传播的距离，发送方可以指定允许通过的路由器数量，这称为存活时间，简称TTL。要理解路由器如何知道其他哪一个路由器具有组播组的成员，请参见Comer[2000b]。

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成员的情况下存在——因特网管理局将它们的地址设成224.0.0.1到224.0.0.255。其中，第一个地址指的是所有组播主机本身。

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消失。创建一个临时组时，要有一个空闲的组播地址以避免意外地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，协议将把TTL设置为一个小的值，使得它不可能与其他组选择同一个地址。然而，用IP组播在因特网上编程需要解决这个问题。会话目录（sd）程序用于启动或加入一个组播会话[Handley 1998，session directory]。它提供了一个具有交互界面的
165 工具，允许人们浏览已公布的组播会话，或公布自己的会话，可指定时间和持续时间——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 IP组播上的数据报组播与UDP数据报有相同的故障特征，也就是说，它们也会遭遇遗漏故障。如果在组播上出现遗漏故障，那么哪怕遇到一个遗漏故障，消息也不能

保证传递到一个特定组的所有成员。也就是说，有部分组成员能接收到消息。这称为不可靠的组播，因为它不能保证消息传递到组的每一个成员。可靠的组播见第12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，图4-20中所示的6789）或任何空闲的本地端口创建套接字。一个进程可通过调用它的组播套接字的joinGroup方法以一个给定的组播地址为参数加入到一个组播组。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给位于这个端口的组的数据报。进程通过调用它的组播套接字的leaveGroup方法离开指定的组。

```
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4-20 组播成员加入一个组，然后发送和接收数据报

在图4-20所示的例子中，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的实例，并通过组播套接字发送该消息到端口6789上的组播组地址。之后，它试图通过它的套接字从同属同一端口上的其他组成员处接收三个组播消息。当该程序的几个实例同时在不同的计算机上运行时，它们都加入同一个组，每一个实例应该接收自己的消息和来自同一组的消息。

Java API允许通过setTimeToTive方法为组播套接字设置TTL。默认值是1，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文本格式的会话，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

上一节介绍了IP组播的故障模型。也就是说，IP组播会遭遇遗漏故障。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已满而丢弃消息。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。如果组播路由器出现故障，那么通过路由器到达的组成员将不能接收到组播消息，尽管本地成员还可以接收组播消息。

还有一个问题是排序。在互连网络上发送的IP数据包不一定按发送顺序到达，这样，组中的一些成员从同一个发送者处接收的数据报的顺序可能与其他组成员接收的顺序不一样。另外，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。

可靠性和排序的效果举例 我们现在用4.5节开始部分介绍的四个使用复制的例子考虑IP组播故障语义的作用。

1）*基于服务复制的容错*：考虑这样一个复制服务，它由一组服务器组成，这些服务器以相同的初始状态启动，总是以相同的顺序执行相同的操作，以便维持彼此之间的一致性。这个组播应
166 用要求要么所有的复本接收到同一个操作请求，要么所有的复本都没有接收到操作请求——如果有
167 一个复本错过了该请求，那么它就无法与其他复本保持一致。在大多数情况下，该服务将要求所有成员以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定期发送组播请求，那么在定位发现服务器时偶尔丢失请求不会产生太大的问题。事实上，Jini在它的协议中使用了IP组播寻找发现服务器。16.2.1节将讲述这个问题。

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的操作的情况，此时消息丢失和顺序不一致所产生的后果取决于复制的方法和所有最新复本的重要性。例如，新闻组的复本不必在任何时候都一致——消息甚至可以以不同的顺序出现，用户能处理这种情况。

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的查找服务使用IP组播通告服务的存在（参见16.2.1节）。

这些例子说明，一些应用需要比IP组播更可靠的组播协议。特别是，有*可靠组播*的需求即传输的任何消息要么被一个组的所有成员都收到，要么所有成员都收不到。这些例子也说明，有些应用对顺序有严格的需求，需求最严格的称为*全排序组播*，这时，传输到一个组的所有消息要以相同的顺序到达所有成员。

第12章将定义和说明如何实现*可靠组播*以及各种有用的排序保证，包括全排序组播。

4.6 实例研究：UNIX中的进程间通信

UNIX BSD 4.x中的IPC原语是以系统调用方式提供的，它们在因特网TCP和UDP协议上作为一层实现。消息目的地被指定为*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。4.2.2节介绍过，消息在发送套接字上排队，直到网络协议传输它们为止，如果协议要求的话则要等到确认到达才能传输。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为止。

任何进程都可以创建一个套接字来与其他进程通信。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
168 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，以便在后续的系统调用中引用该套接字。该套接字将一直存在，直到它被关闭或使用该描述符的所有进程都退出为止。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。

在一对进程能通信前，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系统调用就是用于上述目的的。它的参数是一个套接字描述符和一个结构引用，该结构包含了套接字所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。

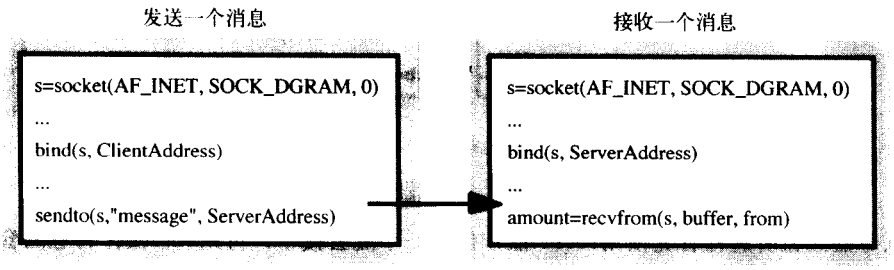
用一个系统调用完成套接字创建和将名字绑定到一个套接字似乎更合理，就像在Java API中一样。用两个独立的系统调用的好处是套接字在没有套接字地址时也可使用。

当套接字地址用作进程目的地时，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，其他指向相应套接字地址的进程就可以间接找到该套接字。任何进程（例如计划通过它的套接字接收消息的服务器）必须首先将套接字绑定到一个套接字地址，并将该套接字地址告知所有潜在的客户。

4.6.1 数据报通信

为了发送数据报，每次通信时要确定一对套接字。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来达到。

图4-21对此给出了说明，图中简化了对参数的描述。



ServerAddress和ClientAddress是套接字地址。

图4-21 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用于指定某个协议，将它设成0表示由系统选择一个合适的协议——在本例中是UDP。 169
- 两个进程接下来使用bind调用将它们的套接字绑定到套接字地址。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，并且让发送方知道该地址。
- 发送进程使用sendto调用，其参数指定消息通过哪个套接字发送、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，并返回所发送的实际字节数。当我们请求数据报服务时，消息传递到目的地并且不需要确认。如果消息太长而不能发送，那么会返回一个错误（同时消息不被传输）。
- 接收进程使用recvfrom调用，其参数指定接收消息的本地套接字、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调用收集套接字队列上的第一个消息，如果队列为空，该调用将等待，直到消息到达为止。

只有当一个进程中的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操作给它传递

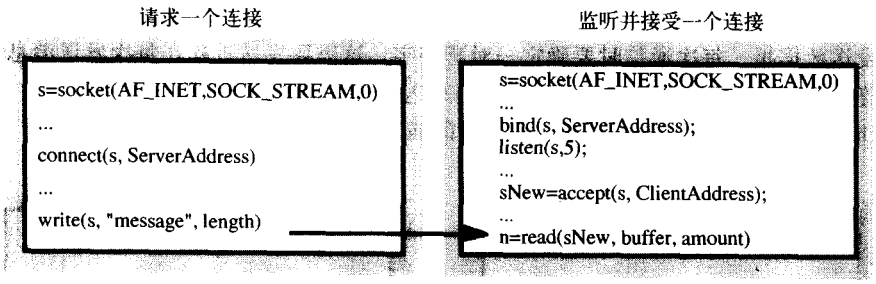
的每个消息提供发送方的地址。UNIX数据报通信的性质与4.2.3节描述的一样。

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连接，参见图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。也就是说，它们的行为和流一样，因为任何可用的数据可以按照写入它们的顺序马上被读出，而且没有消息边界的标志。然而，接收套接字的队列是有限的，如果队列为空，接收方会阻塞；如果队列满了，发送方会阻塞。

对于客户和服务器之间的通信，客户请求连接，监听服务器接受连接。当接受连接时，UNIX自动创建一个新的套接字，并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连接请求。在后续的流通信中可以一直使用一对互连的流套接字，直到连接关闭为止。

流通信如图4-22所示，该图简化了参数描述。图中没有给出服务器关闭监听的套接字。正常
170 情况下，服务器应该首先监听并接受一个连接，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。



ServerAddress和ClientAddress是套接字地址。

图4-22 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。如果将第三个参数设成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。
- 服务器使用accept系统调用接受客户请求的连接，并获得一个新的套接字来与该客户的通信。原来的套接字仍可以用于接收其他客户后续的连接。
- 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者的套接字绑定，所以以前的绑定是不必要的。
- 在连接建立之后，双方进程都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操作与文件的写操作类似。它指定要发送给套接字的消息。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操作从它的缓冲区中接收字符，返回接收到的字符数。

171 UNIX流通信的性质与4.2.4节描述的性质一样。

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设施，它存在遗漏故障但没有内在的性能障碍。另一

方面，TCP可以保证消息传递但需要以额外的消息、高延迟和存储开销作为代价。

第2节给出了三种编码风格。CORBA和它的前身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，它包括了关于数据内容类型的所有信息，允许接收方根据内容重构数据。XML类似Java，包含了所有类型信息。另一个大的区别是CORBA要求为数据项类型的规约编码（用IDL），以便生成编码和解码方法，而Java使用反射将对象序列化并解序列化串行格式的对象。生成XML可采用许多不同的手段，这取决于上下文。例如，许多编程语言（包括Java）提供了在XML和语言级对象之间进行转换的处理器。

请求－应答协议部分给出了一个有特殊目的的有效的分布式系统协议，它基于UDP数据报。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，协议还能做得更可靠。按照该协议，不能保证“请求消息的发送将导致方法的执行”——这已经可以满足一些应用了。通过利用消息标识符和消息重传确保一个方法最终确实被执行可以增加可靠性。对具有幂等操作的服务，这足够了。然而，其他应用要求重传应答消息但不再次执行请求中的方法。借助于历史可实现这一点。这说明，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，系统很少发生错误，这时过度可靠的通用协议性能比较差。

组播消息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，尽管会有遗漏故障，它对许多组播应用而言仍是一个有用的工具。其他一些应用有更高的需求——特别是，组播传递应该是原子的，即它应该具有全部传递或全部不传递的性质。关于组播的其他需求与消息的顺序有关，最严格的需求是组的所有成员都要按相同的顺序接收所有消息。 172

练习

4.1 请设想一个端口有几个接收者时，对消息的处理情况？ （第134页）

4.2 服务器创建了一个端口，用于从客户端接收请求。讨论有关端口名字和客户使用的名字之间关系的设计问题。 （第134页）

4.3 图4-3和图4-4中的程序可从www.cdk4.net/ipc下载，用它们制作一个测试包来确定数据报被丢弃的条件。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，服务器应该能检测到。 （第136页）

4.4 利用图4-3中的程序制作一个客户程序，让它反复地从用户处读取输入，并用UDP数据报消息把这些内容发送到服务器，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，以便在服务器没有应答时能通过客户通知用户。用图4-4中的服务器测试该客户程序。 （第136页）

4.5 图4-5和图4-6中的程序可从www.cdk4.net/ipc获得。修改程序以便客户能反复读取用户输入并将它写到流中；服务器反复地从流中读取内容，并将每次读取的结果打印出来。比较用UDP数据报发送数据和在流上发送数据。 （第140页）

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，以及发送方崩溃时接收方的情况。 （第140页）

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，讨论这种方法的好处和不足。 （第146页）

4.8 Sun XDR在每个简单类型值上进行4字节边界对齐，而CORBA CDR对一个大小为n字节的简单类型在n字节边界对齐。讨论在选择简单类型值占据的大小应做出的权衡。 （第146页）

4.9 为什么在CORBA CDR中没有显式的数据类型？ （第146页）

4.10 用伪代码写一个算法描述4.3.2节中描述的序列化程序。算法应该给出类和实例的句柄被定义或替换的时间。描述在对类Couple的实例进行序列化时，你的算法应该生成的序列化格式。 （第148页）

```
class Couple implements Serializable{
        Private Person one;
        Private Person two;
        Public Couple(Person a, Person b) {
            one=a;
            two=b;
        }
}
```

173

4.11 用伪代码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。（第148页）

4.12 为什么在XML中不能直接表示二进制数据，例如，表示成Unicode字节值？XML元素能携带表示成base64的串。讨论用这种方法表示二进制数据的好处或不足。（第150页）

4.13 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4-13所示的信息，应该提供请求－应答协议所需的访问方法。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，解释其类型选择。（第154页）

4.14 定义一个类，该类的实例表示如图4-16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符是如何赋予的。它还应该提供将自身编码成字节数组的方法和将字节数组解码成一个实例的方法。（第154页）

4.15 利用UDP通信，但不增加任何容错手段，为图4-15所示的请求－应答协议的三个操作编程。应该使用练习4.13和练习4.14定义的类。（第156页）

4.16 概要写出服务器的实现，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将requestId拷贝到应答消息以及它如何获得客户IP地址和端口。（第156页）

4.17 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。超过超时时间后，它将重传请求消息*n*次。如果仍没有应答，它将通知调用者。（第158页）

4.18 描述如下的情形：客户接收到一个对早先发出的调用的应答。（第156页）

4.19 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。（第156页）

4.20 讨论下列操作是否是幂等的：

- 按电梯的请求按钮。
- 将数据写入文件。
- 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？（第158页）

4.21 解释与将服务器端的应答数据量减至最小的设计选择。比较使用RR和RRA协议时，它们各自的存储需求。（第158页）

174

4.22 假设使用的是RRA协议。服务器应该把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？（第158页）

4.23 为什么对性能而言在协议中交换的消息数比发送的数据总量更重要？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：在客户端附加使用一个定时器。（第158页）

4.24 IP组播提供的服务存在遗漏故障。可基于图4-20中的程序制作一个测试包，用于发现组播消息被组播组成员丢弃的条件。该测试包应该能允许多个发送进程。（第158页）

4.25 设计一个消息重传机制，用于在IP组播中克服消息丢失的问题。你应该考虑以下几点：

- 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。 （第167页）

4.26 练习4.25的解决方案应该克服IP组播中的消息丢失问题。你的解决方案在什么意义上与可靠组播的定义不同？ （第167页）

4.27 设计一个场景，由不同客户发送的组播按不同的顺序传递到两个组成员。假设使用了某种形式的消息重传，但未丢失的消息按发送方的顺序到达。接收方如何对这种情况加以补救？ （第167页）

4.28 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。 （第156页，第165页）

第5章 分布式对象和远程调用

本章将介绍依靠远程方法调用（RMI）的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故障的可能性，所以RMI与本地调用有着不同的语义。虽然也可以使RMI看起来与本地调用非常相似，但是这种完全的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实例研究简要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在感兴趣的远程对象上发生的事件，并且在这样的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。

176
~
177

在Java RMI实例研究中将对RMI的使用进行介绍。

第20章是关于CORBA的实例研究，其中包括CORBA RMI和CORBA事件服务两方面的内容。

5.1 简介

本章涉及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的相互协作的程序组成的应用。这样的程序应该能够调用其他进程中的操作，而这些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：

- 最早的也是最为人们所熟知的扩展就是将传统的过程调用模型扩展为*远程过程调用模型*，远程过程调用模型允许客户程序调用运行在许多独立的进程中的服务器程序中的过程，而这些服务器程序通常运行在不同于该客户的计算机进程中。
- 20世纪90年代，基于对象的编程模型被扩展为允许不同进程里的对象依靠*远程方法调用*（Remote Method Invocation, RMI）彼此通信。RMI是本地方法调用的扩展，它允许生存在一个进程中的对象调用生存在另外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。这一模型已经扩展为允许编写基于事件的分布式程序。

注意，我们用术语“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相混淆。当前，大多数分布式系统软件都使用面向对象的语言来编写，RPC可以被理解为与RMI相关。因此，本章重点阐述RMI和事件范型，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，接下来讨论RMI的设计和实现。5.5节将给出Java RMI实例研究。RPC将在5.3节的Sun RPC实例研究中讨论。在第19章中，我们还将看到RPC在Web service中的应用。5.4节将讨论事件和分布式通知，而更深入的关于CORBA的实例研究将在20章中给出。

中间件 在进程和消息传递的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，参见图5-1。例如，远程方法调用抽象就是基于4.4节讨论的请求－应答协议的。

中间件的一个重要方面是提供位置透明性和与具体通信协议、操作系统、计算机硬件无关的独立性。某些形式的中间件还允许以不同的编程语言编写组件。

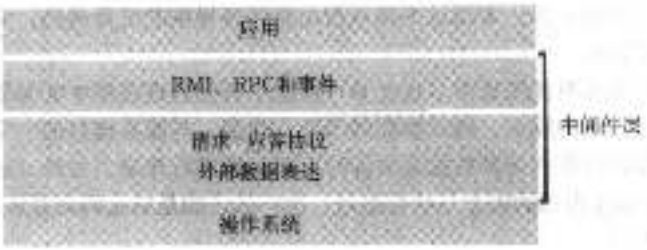


图5-1 中间件层

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，甚至可能是在另外一台计算机上。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是无关的。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：4.3节已经描述过用于表示外部数据的二种公认的标准。它们用于消息编码和解码。它们隐藏了硬件体系结构所导致的不同，如字节顺序等。

操作系统：中间件层提供的更高级的抽象是与其下层的操作系统无关。

多种编程语言的使用：一些中间件允许分布式应用使用多种编程语言。特别是CORBA（参见第20章），它允许客户调用以不同的编程语言编写的服务器程序中的对象的方法。而这是通过使用接口定义语言（Interface Definition Language, IDL）来定义接口而实现的。IDL将在下节中讨论。

接口

大多数现代编程语言提供了把一个程序组织成一系列能彼此通信的模块的方法。模块之间的通信可以依靠模块间的过程调用，或者直接访问另外一个模块中的变量来实现。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，模块接口指定可供其他模块访问的过程和变量。实现后的模块就隐藏了除接口以外的所有信息，只要模块的接口保持相同，模块的实现就可以随意改变而不影响模块的使用者。

分布式系统中的接口 在分布式程序中，模块可以在彼此独立的进程中运行。让运行在一个进程中的模块访问另一个进程中模块的变量是不可能的。因此，用于RPC或者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，这看起来好像打破了上述规则，然而，这些属性也不可以直接访问，而必须通过一些为接口自动添加的getter或setter过程来访问。

178 ~ 179

在本地过程调用中，参数传递可以采用传递值或者传递引用的方式，但这种参数传递机制在调用过程与被调用过程位于不同进程的时候就不合适了。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型、输出型或者两者兼而有之。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参数又是输出型参数时，它的值必须既在请求消息又在应答消息中传递。

本地模块和远程模块之间的另一个区别是，一个进程中的指针在其他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。

下面将分别讨论最初的客户-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-服务器模型下，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的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，与服务接口的一个很大的不同在于远程接口中的方法可以把对象作为参数或者方法的返回结果进行传递。另外，远程对象引用也可以传递——不能把远程对象引用的概念与指针混淆，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机制可以集成到某种编程语言中，只要该语言包含适当的定义接口的表示法，并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机制添加到面向对象编程语言的一个例子。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。因为它允许程序员用一种语言实现本地调用和远程调用，所以这种方法也很方便。

然而，许多现有的有用的服务是用C++和其他语言编写的。为了满足远程访问的需要，允许程序采用包括Java在内的各种语言编写是非常有益的。接口定义语言（IDL）允许以不同语言实现对象以便相互调用。IDL提供了一种定义接口的表示法，接口中方法的每个参数可以在类型声明之外附加输入或输出类型说明。

```
//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

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指定它的参数是输入型，意味着它是一个输入型参数。而方法getPerson是通过名字检索一个Person实例，它将其第二个参数指定为out型，意味着这是一个输出型参数。

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（参见第20章），而Sun XDR是RPC的一种IDL（参见5.3节）。而Web服务描述语言（WSDL）则是一种因特网范围内的RPC（参见19.3节）。

其他例子包括用于OSF的分布式计算环境（DCE）中的RPC系统的接口定义语言[OSF 1997]，它使用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）。

5.2 分布式对象间的通信

分布式系统中基于对象的模型扩展了面向对象编程语言支持的模型，使得它能适用于分布式对象。本节将介绍分布式对象间通过RMI方式的通信。本节将讨论以下几个方面：

对象模型：对象模型相关内容的简单回顾，适合具有某种面向对象编程语言（如Java或C++）的基础知识的读者阅读。

180 ~ 181

分布式对象：介绍基于对象的分布式系统，论证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讨论：

1）只执行一次本地调用，但有没有更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，有哪些差别不能被消除？

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。

分布式无用单元收集：介绍适用于RMI实现的分布式无用单元收集的一种算法。

5.2.1 对象模型

用Java或C++等语言编写的面向对象程序由一个交互对象集合组成，其中每个对象包含若干数据和方法。一个对象与其他对象通过调用对方的方法进行通信，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（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访问。

对象引用 对象可以通过对象引用来访问。例如，在Java中，一个变量看上去拥有一个对象，但实际上只拥有对该对象引用。为了调用对象的一个方法，需要给出对象引用和方法名，以及必要的参数。其方法被调用的对象有时候称为*目标*，有时候称为*接收者*。对象引用具有极高的价值，因为它们可以赋给变量，也可作为参数传递或者作为方法的结果返回。

接口 接口在无需指定其实现的情况下提供了一系列方法基调的定义（即参数的类型、返回值和异常）。如果类包含实现接口中方法的代码，那么就称对象提供该接口。在Java中，一个类可以实现几个接口，而一个接口的方法也可以由任意类实现。接口还可以定义用于声明参数类型、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，动作由调用另一个对象的方法的对象启动。调用可以包含执行方法所需的附加信息（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。方法的调用会产生三个结果。 182

1）接收者的状态会发生改变。

2）可以实例化一个新的对象，例如使用Java或C++中的构造函数进行实例化。

3）可能会在其他对象中发生其他方法调用。

因为调用可能导致其他对象对方法的调用，所以动作就是一连串相关的方法调用，每个调用最终都会返回。这里的解释没有考虑异常。

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，会发现许多不同的问题，例如，对象变量的值不一致，无法读写文件或网络套接字。为此，程序员需要在他们的代码中插入测试语句以处理所有不常出现的情况或出错情况，但这会降低正常情况下的代码的清晰性。利用异常，便可以在不使代码复杂化的情况下清晰处理错误条件。另外，每个方法的标题都清楚地列出了产生异常的错误条件，以便方法的用户去处理它们。可以定义一块代码，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。这意味着要将控制传递给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。

无用单元收集 当不再需要对象时有必要提供一种手段释放其占用的空间。有的语言（如Java）可以自动检测出什么时候该收回一个已经不再访问的对象占据的空间，并将此空间分配给其他对象使用。这个过程称为*无用单元收集*。有的语言（例如C++）不支持无用单元收集，那么程序员必须自己处理释放分配给对象的空间的问题。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。在基于对象的范型中，程序的状态被划分为几个单独的部分，每个部分都与一个对象关联。因为基于对象的程序是从逻辑上划分的，所以在分布式系统中可以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它们的方法。在RMI中，客户调用一个对象方法的请求以消息的形

式传送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，并将处理的结果通过另一个消息返回给客户。考虑到会有一连串的相关调用，因此服务器中的对象也可以成为其他服务器中对象的客户。

183 分布式对象也可以采用其他体系结构模型。例如，为了获得良好的容错性并提高性能，可以复制对象。又如，为了改善性能和可用性，可以迁移对象。

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状态。例如，不同计算机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，也就可能出现访问冲突。然而，对象的数据只能由其自己的方法访问这一事实允许对象提供保护自身免遭不正确访问的方法。例如，它们会使用条件变量这样的同步原语来保护对它们的实例变量的访问。

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，对象可以通过RMI来访问，若类是本地实现的话，可将对象拷贝到一个本地缓存并进行直接访问。

对异构系统而言，对象只能由其方法访问这个事实还有一个好处，即在不同场合使用的不同数据格式——使用RMI访问对象方法的客户不会注意到数据格式的不同。

5.2.3 分布式对象模型

本节将讨论对象模型的扩展以便使它可以用于分布式对象。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，而其他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称为本地方法调用。

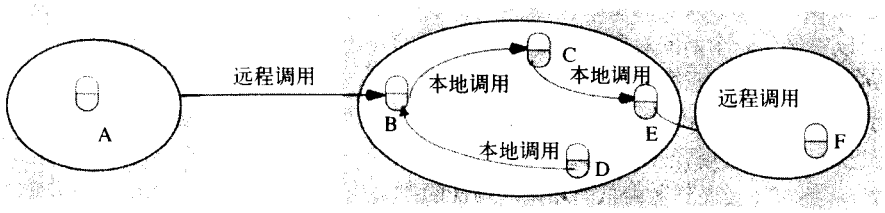


图5-3 远程和本地方法调用

我们将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。所有对象都能够接收本地调用，当然它们只能接收来自拥有该对象引用的其他对象发出的本地调用。例如，对象C必须具有对对象E的引用，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

184 型的核心：

远程对象引用：如果对象能访问远程对象的*远程对象引用*，那么它们就可以调用该远程对象上的方法。例如，在图5-3中，B的远程对象引用必须对A是可用的。

远程接口：每个远程对象都有一个*远程接口*，由该接口指定哪些方法可以被远程调用。例如，对象B和F必须具有远程接口。

下面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，使那些能接收RMI的对象都具有远程对象引用。远程对象引用是一个可以用于整个分布式系统的标识符，它指向某个唯一的远程对象。它的表示通常与本地对象引用不同，我们已经在4.3.4节中讨论过了。远程对象引用与本地对象引用主要在以下两方面类似：

1）调用者通过远程对象引用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 远程对象的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。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远程对象实现的其他方法。注意，和所有的接口一样，远程接口没有构造函数。

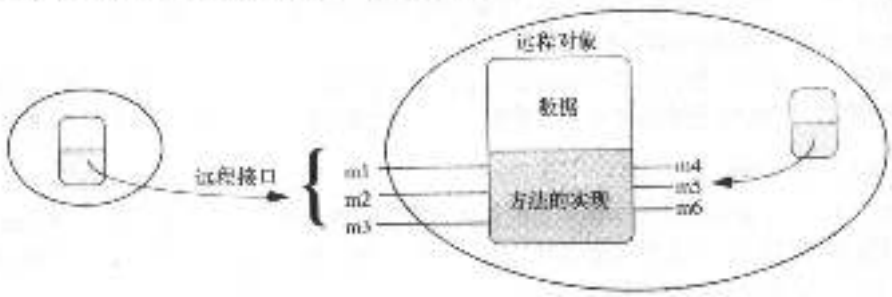


图5-4 远程对象及其远程接口

CORBA系统提供了一种接口定义语言（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。如C++，Java或Python。CORBA客户不需要为了远程调用其方法而使用与远程对象相同的语言。

在Java RMI中，远程接口以和任何其他Java接口相同的方式定义，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（参见20.2.3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。 185

分布式对象系统中的动作 和非分布式的情形类似，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。但是在分布式情形下，涉及一连串相关调用的对象可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。此时，对象的远程引用必须对调用者是可用的。在图5-3中，对象A需要有到对象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可以从对象B得到一个对对象F的远程引用。

当一个动作导致一个新的对象被实例化时，这个对象的生命周期通常就是实例化该对象的进程的生命周期，例如，使用构造函数时。如果这个新实例化的对象有远程接口，那么它就是一个拥有远程对象引用的远程对象。

分布式应用可以提供一些远程对象，通过这些对象提供的方法，可以实例化另一些对象，而这些新实例化的对象可以通过RMI来访问。这种方式提供了一种有效的实例化远程对象的方式。例如，在图5-5中，假设对象L包含能生成远程对象的方法，则来自对象C和对象K的远程调用将分别导致对象M和对象N的实例化。

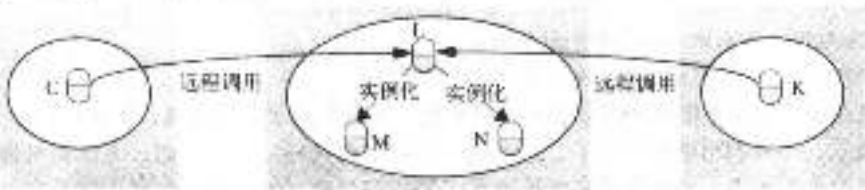


图5-5 远程对象的实例化

分布式对象系统中的无用单元收集 如果语言（例如Java）支持无用单元收集，那么任何与之相关的RMI系统也应该支持远程对象的无用单元收集。分布式无用单元收集通常通过已有的本地无用单元收集器和一个执行分布式无用单元收集的附加模块（一般基于引用计数）的协作来实现。5.2.6节对此做了详尽的描述。如果语言不支持无用单元收集，那么无用的远程对象应当被删除。

异常 任何远程调用都可能会因为被调用对象的种种原因而失败（这里被调用的对象处于与调用对象不同的进程或计算机中）。例如，包含远程对象的进程可能已经崩溃，或者由于太忙而无法应答，又或者调用消息或结果消息丢失了。因此，远程方法调用应该能够引起异常，比如因分布引起的超时异常，以及被调用的方法执行期间导致的各种异常。后者的例子有超过文件末尾的读
186 操作，或者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。本节将讨论在进行扩展时出现的设计问题：

- 调用语义的选择——虽然本地调用只执行一次，但它并不一定适用于远程方法调用的场合。
- 对RMI而言最合适的透明性级别。

在5.2节的余下部分，我们将把驻留了远程对象的进程作为服务器，把驻留调用者的进程作为客户。服务器也可能是客户。

RMI调用语义 4.4节讨论了请求－应答协议，在该节中说明了可以通过不同的方式实现doOperation以提供不同的传输保证。主要的选择有：

- *重发请求消息*：是否要重发请求消息，直到接收到应答或者认定服务器已经出现故障为止。
- *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*：是否要在服务器上保存结果消息的历史，以便无需重新执行服务器上的操作就能重传丢失的结果。

将这些选择组合使用便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5-6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。RMI调用语义定义如下：

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5-6 调用语义

或许调用语义：采用或许调用语义，远程方法可能执行一次或者根本不执行。当没有使用任何容错措施的时候，就启用了或许语义。它可能会遇到以下的故障类型：

- 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的话，那么该方法是否执行过就不能确定。如果调用消息丢失了，那么该方法就不会执行。另一方面，方法也可能执行过了，只是结果消息丢失了。系统崩溃可能发生在方法执行之前，也可能发生在方法执行之后。此外，在异构系统中，方法执行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，调用者可能收到返回的结果，也可能收到一个异常。在收到返回结果的情况下，调用者知道该方法至少执行过一次，而异常信息则通知它没有接收到执行结果。至少一次调用语义可以通过重发请求消息来达到，它屏蔽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下列类型的故障：

- 由于包含远程对象的服务器故障而引起的系统崩溃。
- 随机故障。重发调用消息时，远程对象可能会接收到这一消息并多次执行某一方法，结果导致存储或返回了错误的值。

第4章定义了幂等操作，这种操作反复执行后的结果与只执行一次的结果一样。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，如果重复执行的话，存款余额就可能不断增加！如果能设计服务器中的对象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，调用者可以接收返回的结果，也可以接收一个异常。在接收返回结果的情况下，调用者知道该方法恰好执行过一次。而异常信息则通知调用者没有收到执行结果。在这种情形下，方法要么执行过一次，要么根本没有执行。至多一次调用语义可以通过使用所有的容错措施来达到。正如前面的情形，重发请求消息可以屏蔽所有调用或结果消息的遗漏故障。另外的容错措施通过确保每个RMI方法永远执行不超过一次来避免随机故障。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。 187 ~ 188

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，使得本地过程调用和远程过程调用在语法上没有差别。所有对编码和消息传递过程的必要调用都对编写调用的程序员面隐藏起来。尽管请求消息在超时后重新发送，但这对调用者而言也是透明的——使远程过程调用的语义与本地过程调用的语义相似。透明性概念可以被扩展以应用到分布式对象中，但是它不仅涉及隐藏编码和消息传递过程，而且还涉及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令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容易失败，因为它们涉及网络、另一台计算机和另一个进程。不论选择上述哪种调用语义，总有可能接收不到结果，而且在出现故障的情况下，不可能判别故障是源于网络的失效还是源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样的情形中恢复。

远程调用的延迟要比本地调用的延迟大好几个数量级。这表明，利用远程调用的程序要把延迟因素考虑进去，比如尽可能减少远程交互等。Argus[Liskov and Scheifler 1982]的设计者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13章讨论。

Waldo等[1994]认为，本地对象和远程对象之间的不同应该表现在远程接口上，让对象对可能出现的部分故障以一致的方式做出反应。有些系统，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，有些系统则做出了实际性改进。以Argus为例，它已被扩展到远程操作对于程序员而言是显式的程度。

IDL的设计者也会面临远程调用是否应该透明的抉择。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。客户程序应该能处理这一异常，并解决此类故障。IDL也可以提供一种指定方法的调用语义的机制，这对于远程对象的设计者是有帮助的——例如，倘若为了避免至多一次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。

当前比较一致的意见是，从远程调用的语法与本地调用的语法一致的角度看，远程调用应该是透明的，但本地对象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异常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。通过远程调用来访问对象的知识对对象的设计者而言还有另一层含义，即它应该能在多个客户并发访问的情况下保证状态的一致性。 189

5.2.5 RMI的实现

完成远程方法调用涉及几个独立的对象和模块。如图5-7所示，一个应用级对象A拥有一个对B的远程对象引用，可以调用远程应用级对象B的一个方法。本节将讨论图中每一个组件扮演的角色，首先讨论通信和远程引用模块，然后讨论运行在模块上面的RMI软件。

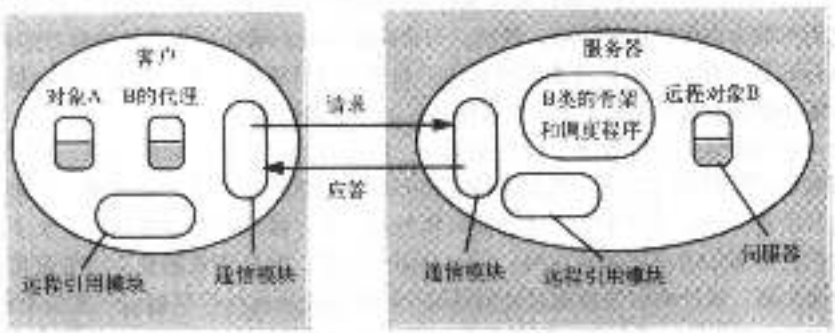


图5-7 在远程方法调用中的代理和骨架角色

除此之外，本节将讨论以下几个主题：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执行请求—应答协议，它们在客户和服务器之间传递请求和应答消息。请求和应答消息的内容见图4-16。通信模块只使用前三项，即消息类型、requestId和被调用对象的远程引用，methodId和所有的编码与解码都与下面讨论的RMI软件有关。两个通信模块一起负责提供一个指定的调用语义，如至多一次。

服务器端通信模块为被调用的对象类选择分发器，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。分发器的作用将在下面的RMI软件中讨论。

远程引用模块 远程引用模块负责在本地对象引用和远程对象引用之间进行翻译，并负责创建远程对象引用。为履行其职责，每个进程中的远程引用模块都有一个远程对象表，该表记录着该进程的本地对象引用和远程对象引用（整个系统范围内）的对应关系。这张表包括：

190

- 该进程拥有的所有远程对象。例如，在图5-7中，远程对象B会记录在服务器端的表中。
- 每个本地代理。例如，在图5-7中，B的代理会记录在客户端的表中。

代理的作用将在下面的RMI软件中讨论。远程引用模块的动作如下：

- 当远程对象第一次作为参数或者结果传递时，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，远程引用模块要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的时候，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，可使用这张表找出调用了哪个本地对象。

伺服器 伺服器是一个提供了远程对象主体的类的实例。由相应的骨架传递的远程请求最终是由伺服器来处理的。伺服器存活于服务器端的进程中。当远程对象被实例化时，就会生成一个伺服器，而且这些伺服器可以一直使用到不再需要远程对象为止。最终，伺服器也将作为无用单元被回收或删除。

RMI软件 它由位于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5-7中，中间件对象的角色有如下几种角色：

代理：代理的作用是通过对调用者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，其中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则有很大区别。代理中的每个方法会把一个目标对象的引用、它自身的methodId和它的参数编码进一个*请求*消息并发送到目标，然后等待*应答*消息，解码并将结果返回给调用者。

分发器：服务器对表示远程对象的每个类都有一个分发器和骨架。在我们的例子中，服务器有远程对象B的类的分发器和骨架。分发器接收来自通信模块的*请求*消息，并传递*请求*消息，并使用methodId选择骨架中恰当的方法。分发器和代理对远程接口中的方法使用相同的methodId。

骨架：远程对象类有一个*骨架*，用于实现远程接口中的方法。这些方法与作为远程对象的主 191
体的伺服器中的方法极为不同。一个骨架方法将*请求*消息中的参数解码，并调用伺服器中的相应方法。它等待调用完成，然后将结果和异常信息编码进*应答*消息，传送给发送方代理的方法。

远程对象引用以图4-13中的形式编码，其中包括远程对象的远程接口的信息，例如远程接口的名字或者远程对象类。这条信息能确定代理类，以便在需要的时候可以创建一个新的代理。例如，可以通过把“_proxy”添加到远程接口名中来创建代理类名。

创建代理类、分发器类和骨架类 在RMI使用的代理类、分发器类和骨架类由接口编译器自动创建。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++或Java语言创建代理类、分发器类和骨架类[www.iona.com]。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、分发器类和骨架类。

动态调用：可替换代理的选择 上面提到的代理是静态的，即代理类是通过接口定义生成的，并且被编译到客户端的代码中。但在有些情况下，这是不实际的。例如，如果一个远程引用指向了客户端程序中的对象，而这个对象的远程接口在编译时是不能确定的。在这种情况下，需要采用其他的方法调用该远程对象，这就称*动态调用*。客户应用程序可以通过动态调用获得远程调用的一般性表示，例如练习5.8中的DoOperation方法，这个方法是RMI的基础体系结构的一部分（参见第4.4节）。客户端会提供远程对象引用、方法名和DoOperation方法的参数，然后等待接收结果。

需要注意的是，尽管远程对象引用包含远程对象接口的信息，例如远程对象接口的名字。但是这些信息是不够的，因为动态调用还需要知道远程对象接口的方法名和参数的类型。在20.2.2节，我们将会看到，CORBA使用一个称为Interface Repository的组件来提供所需的信息。

将动态调用接口作为代理并不方便，但如果应用程序中的某些远程对象的接口不能在设计时确定，那么动态调用接口就会非常有用。例如，这种应用的一个例子是我们在描述Java RMI（参见第5.5节）、CORBA（参见第20.2节）和Web服务（参见第19.2.3节）时给出的共享白板。共享白板这个应用程序能够显示各种图形，例如圆、矩形和直线，但是它也应当能够显示那些客户端编译时没有预先定义的图形。客户端可以通过动态调用解决这个难题。在第5.5节中，我们还可以看到，在Java RMI中客户端可以通过动态的下载类的方法来代替动态调用。 192

动态骨架：从上面的例子中，我们可以清楚地看到，服务器有时需要驻留那些接口在编译时尚不能确定的远程对象。例如，运行共享白板程序的服务器需要保存客户提供的新的图形。使用动态骨架的服务器便能够解决这种问题。我们将在第20.2.2节中描述动态骨架。在第5.2节我们会看到，利用一个普通的分发器将类动态地下载到服务器，Java RMI便可以解决这个问题。

服务器和客户程序 服务器程序包含分发器类和骨架类，以及它支持的所有伺服类的实现。另外，服务器程序包含*初始化*部分（例如，在Java或C++中的main方法里）。*初始化*部分负责创建并

初始化至少一个驻留在服务器上的伺服器，其余的伺服器可以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（参见后文）注册它的一些伺服器。通常情况下，它只注册一个伺服器，该伺服器可以用来访问其他对象。

客户程序会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。

工厂方法：我们早就注意到远程对象接口不能包括构造子。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。伺服器可以在*初始化*部分创建，也可以在为该用途而设计的远程接口中创建。术语*工厂方法*有时指创建伺服器的方法，*工厂对象*指具有工厂方法的对象。任何远程对象，它要想能应客户的需求而创建新的远程对象，就必须在它的远程接口中提供用于此用途的方法。这样的方法称为工厂方法，尽管实际上它们也是普通的方法。

绑定程序 客户程序通常要有一种手段，以便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要求对象B的一个远程对象引用。分布式系统中的*绑定程序*就是一个单独的服务，它维护着一张表，表中包含从文本名字到远程对象引用的映射。服务器用该表来按名字注册远程对象，客户用它来查找这些远程对象。第20章将讨论CORBA命名服务。Java绑定程序，即RMIregistry，将在第5.5节的Java RMI实例研究中简要讨论。

服务器线程 一旦对象执行远程调用，该调用可能会涉及调用其他远程对象的方法，因此可能需要过一段时间才会返回。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到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保留在运行的进程中是不切实际的，因为并不是在所有的时间都要使用它们。为了避免因为在全
193 部时间里运行管理这些远程对象的服务器造成潜在的资源浪费，服务器应该在客户需要它们的任何时候启动，就像TCP服务的标准集（如FTP）那样，Inetd服务会根据需要才启动FTP。启动用于驻留远程对象的服务器的进程被称为*激活器*，原因如下。

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*；如果它现在不是主动的但是可以激活为主动的，就认为它是*被动的*。一个被动对象包括两个部分：

1）它的方法的实现。

2）它的编码格式的状态。

*激活*是指根据相应的被动对象创建一个主动对象，具体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，这涉及记录服务器名字，而不是相应被动对象的URL或者文件名。
- 启动已命名的服务器进程并激活进程中的远程对象。
- 跟踪已经激活的远程对象所在的服务器位置。

Java RMI具有将一些远程对象变为可*激活*[java.sun.com IX]的能力。当一个可激活对象被调用时，如果这个对象的当前状态不是激活状态，那么这个对象将从它的编码状态转化为激活状态，然后执行调用。它在每一个服务器机器中都使用一个激活器。

CORBA实例研究中描述了它的实现仓库——一种弱形式的激活器，它在初始状态下激活含有对象的服务。

持久对象存储 那些在进程两次激活之间仍然保证存活的对象称为*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，如CORBA持久状态服务（见第20.3节）、Java Data Objects[java.sun.com VIII]和Persistent Java [Jordan 1996，java.sun.com IV]。

一般来说，持久对象存储将管理海量的持久对象，这些持久对象都存储在磁盘或数据库中，

直到需要它们的时候才被调用。当这些持久对象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，调用者应该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已经被激活。主存中不再需要的持久对象要变成被动的。在大多情况下，为了容错起见，对象只要达到一个一致的状态，就能够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（如在事务结束或者程序退出的时候）而这样做。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以来修改过的对象。 194

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。实际上，每个可读的名字都与相关的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，任何可以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、Java Data Objects和PerDis[Ferreira et al. 2000]采用。它们使用无用单元收集器剔除从持久根不再可达的对象。
- 持久对象存储提供一些持久类——持久对象属于它们的子类。例如在Arjuna [Parrington et al. 1995]中，持久对象基于提供事务和恢复的C++类。不想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]允许对象在用户的多个本地缓存中激活，而不是在服务器中激活。在这种情况下，就要求有缓存一致性协议。第18章将讨论多种一致性模型。

对象定位 4.3.4节描述了一种远程对象引用，它包含创建远程对象的进程的因特网地址和端口号，用以作为保证唯一性的一种方式。这种形式的远程对象引用也能用作远程对象的地址，只要该对象在余下的生命周期中存在于相同的进程里。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，可能这些进程存在于不同的计算机中。在这种情况下，远程对象引用不能当作地址用。发出调用的客户同时需要一个远程对象引用和一个调用发送到的地址。

*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到它们当前的大概位置——位置是大概的，因为对象可能已经从已知的前一次位置迁移了。例如Clouds系统 [Dasgupta et al. 1991] 和Emerald系统 [Jul et al. 1988] 使用缓存/广播方案，其中每个计算机上的定位服务的一个成员拥有一个小缓存，存放远程对象引用－位置的映射。如果远程对象引用位于缓存中，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者位置不在缓存中的对象，系统要广播一条请求。要改善该方案，可以使用转发定位指针，转发定位指针含有关于对象的新位置的提示。在第9.1节中，我们将给出另一个例子，即将一个资源的URN转换为它当前的URL的解析服务。

5.2.6 分布式无用单元收集

分布式无用单元收集器的目的是提供以下保证：如果一个本地对象引用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时，该对象将被收集，并且它使用的内存将被回收。 195

我们将描述Java的分布式无用单元收集算法，它与Birrel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，只要需要这个代理，它就一直存在。对象生存的进程（它的服务器）应该告知给客户上的新代理。随后当客户不再有代理时，也应告知服务器。分布式无用单元收集器与本地无用单元收集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拥有远程对象引用的一组进程名，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5-7中，这个集合包括图示的客户进程。）这个集合可以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的无用单元收集器注意到远程对象B的一个代理不再可达时，它发出一个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。
- 如果不存在B的一些本地持有者，当B.holders为空时，服务器的本地无用单元收集器将回收被B占有的空间。

通过在进程中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——它不要求任何全局同步。但要注意，为无用单元收集算法所发送的额外调用不能影响到每个正常的RMI，它们只在代理创建和删除的时候发生。

有一种可能，在一个客户发出一个removeRef（B）调用的同时，另一个客户恰好发出addRef（B）调用。若removeRef调用先到达而此时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，就添加一个临时的入口直至addRef调用到达为止。

Java分布式无用单元收集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（意味着该方法要么执行过一次，要么根本没有执行）时，客户不创建代理而是发出一个removeRef（B）调用。removeRef的效果是否正确取决于addRef是否成功。removeRef失败的情况通过租借来处理，如下文所述。

Java分布式无用单元收集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。借期从客户给服务器发出addRef调用开始，到达过期时间后终止或者客户给服务器发出一个removeRef调用后终止。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。客户负责在租期过期之前向服务器请求续借。 196

Jini中的租借 Jini分布式系统包括一个租借规约[Arnold et al. 1999]，它可以用于一个对象给另一对象提供一种资源的各种情形，例如远程对象提供引用给其他对象。提供这种资源的对象要冒一些风险，即在用户不再对之感兴趣或者它们的程序可能已经退出的情况下，对象将不得不维护该资源。为了避免用复杂的协议判断资源用户是否还有兴趣，资源只提供一段有限长的时间。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延续它们的租约。

授权者与租借者可以就租期进行磋商，当然这不会发生在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延续或取消的方法。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类似，都是客户程序调用另一个运行在服务器进程的程序中的过程。服务器可以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。实际上，这类服务很像一个有状态和方法的远程对象，但它缺少创建对象的新实例能力，因此也不支持远程对象引用。

可以实现RPC（就像RMI）使之具有5.2.4节讨论的任一调用语义——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4.4节讨论的那样，但该节做了一些简化，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4-16中为RMI描述的那些内容相同。

支持RPC的软件如图5-8所示，它可以是一种不支持类和对象的过程式语言，例如C。这与我们的Sun RPC实例研究是相关的，这其中便使用了C语言，而且由于历史原因，大多数早期的RPC系统都是基于C语言的。

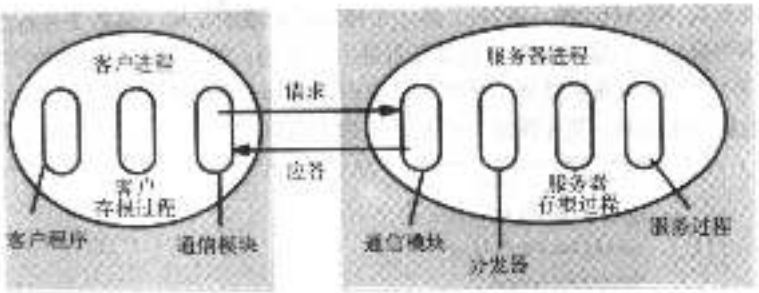


图5-8 使用过程式语言的RPC中客户和服务器存根过程的角色

因为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，该软件与图5-7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方法的作用类似，它表现得就像客户端的本地过程，但它并不执行调用，而是将过程标识符和参数编码进请求消息、通过它的通信模块将消息发送给服务器，当收到应答消息的时候，它解码该结果。服务器进程包含一个分发器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。分发器根据请求消息中的过程标识符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。

客户和服务器存根过程以及分发器可以由接口编译器根据服务的接口定义生成。如果客户或服务器使用面向对象语言，如C++或Java，那么一个服务的客户存根过程可以实现为一个代理，一组服务器存根过程可以实现为一个骨架。

实例研究：Sun RPC

RFC 1831 [Srinivasan 1995a]中描述了Sun RPC，它是为Sun NFS网络文件系统中的客户－服务器通信而设计的。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时，请求消息和应答消息的长度被限制在一定范围内——理论上可以到64KB，但在实际中通常为8KB或9KB。它使用至少一次调用语义，广播型RPC是可选的。

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语言最初用于指明外部数据表达，现在扩展成为一种接口定义语言。通过指定一组过程定义并支持类型定义，XDR可用于定义Sun RPC的服务接口。与CORBA IDL或Java使用的接口定义语言相比，它的表示方法相当简单。特别是体现在以下几个方面：

197
|
198

- 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其唯一的编号。当一个过程签名改变时，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是否正在使用相同的版本。
- 一个过程定义指定一个过程签名和一个过程号。过程号用作请求消息中的过程标识符。（它可能会为接口编译器生成过程标识符。）
- 只允许使用单个输入参数。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。
- 过程的输出参数以单个结果返回。
- 过程签名由结果类型、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的值，也可以指定为包含几个值的一个结构。

例如，见图5-9中的XDR定义，它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行）将一个带有三个域的结构（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，其结果也是一个结构，其中包括返回的字节数和文件数据。WRITE过程（第1行）没有返回结果。WRITE和READ过程分别被赋予编号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5-9 Sun XDR中的文件接口

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。可以使用接口编译器rpcgen根据接口定义生成以下部分：

- 客户存根过程。
- 服务器main过程、分发器和服务器存根过程。
- 用于分发器、客户与服务器存根过程的XDR编码和解码过程。

绑定 Sun RPC在每台计算机上的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正在本地运行的每个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，从而找到服务器的端口。

当一个服务有多个实例运行在不同计算机上的时候，每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消息来达到目的。解决办法是客户发出组播远程过程调用将指定程序和版本号的请求广播到所有的端口映射器。每个端口映射器判断如果有一个合适的本地服务程序的话，就给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，以便在客户和服务器之间传输认证信息。请求消息中包含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有足够的权限来执行所请求的文件操作。

Sun RPC支持几种不同的认证协议，它们包括：

- 没有认证。
- 上文描述的UNIX风格。
- 为标记RPC消息创建共享密钥的风格。
- Kerberos认证风格（见第7章）。

199 ~ 200

RPC头部的一个域指明它使用的风格。

关于安全的一种更通用的方法可参见RFC 2203中的描述[Eisler et al. 1997]。它对RPC消息和消息认证的安全性和完整性提供保障。它允许客户和服务器就安全上下文进行协商，在该上下文中或者不应用任何安全机制或者要求有安全性保障，可能会提供消息完整性、消息私密性保障，或者两者兼而有之。

客户和服务器程序 关于Sun RPC的详细介绍可以在www.cdk4.net/rmi中找到。它包括与图5-9中定义的接口所对应的客户和服务器程序例子。

5.4 事件和通知

事件所基于的思想是，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，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，可以看作是*事件*，事件改变了维护应用状态的对象。只要状态改变，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，它允许将某个对象上发生的事件通知给几个不同位置的对象。它们使用*发布－订阅模式*，生成事件的对象*发布*事件的类型，使其他对象可以发现它。对象如果想从一个已经发布了它的事件的对象处接收通知，就要*订阅*它们感兴趣的事件类型。不同的事件*类型*指的是由感兴趣的对象执行的不同方法。表示事件的对象称为*通知*。通知可以被存储，也可以在消息中发送，可以查询和应用在各种不同的事物中。当一个发布者经历一个事件时，曾经对该类事件表示有兴趣的对象就会接收到通知。订阅某一类型的事件也称为对那类事件*注册兴趣*。

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把它添加到绘图程序中、变更文档、某个事实（比如某个人已经进入或离开了一个房间，一件设备或者是一本有电子标签的书放在了一个新的位置上）的变更。主动标记和嵌入式设备（参见16.1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：当事件通知用作分布式对象间一种通信方式的时候，虽然分布式系统中的组件原本不能用于互操作，但现在也可以让它们一起工作。所要做的事情就是让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如何用于连接因特网中的异构组件。他们描述了这样一个系统，其中的应用可以知道用户的位置和活动，例如使用计算机、打印机或者阅读具有电子标签的书。他们设想未来在家庭网络的环境中，能实现这样的需求："如果孩子们回家，就自动打开中央暖气"。

201

异步性：生成事件的对象给所有向其订阅过该事件的对象异步地传输通知，从而避免了发布者需要与订阅者同步的问题——发布者和订阅者需要解耦合。Mushroom [Kindberg et al. 1996]是一个基于事件的分布式系统，用于支持协同工作，该系统中的用户界面显示表示用户的对象和信

息对象，如被称为网络空间的共享工作区内的文档与记事本。每处空间的状态被复制到当前该处用户的计算机上，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，或者已经在一个对象上执行了某个动作。与某类事件相关的对象的每个复本订阅这类事件，并在这类事件发生时接收通知。但是订阅者与经历事件的对象是解耦合的，因为不同的用户在不同的时间活动。

我们将在下面的证券交易所的例子中阐述在某些情况下事件的重要作用。

简单的证券交易所系统 考虑一个简单的证券交易所系统，它的任务是让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价格。单个上市股票的市场价格通过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，从几个不同的外部来源到达交易所，然后由称为信息提供者的进程收集这些信息。交易者一般只对他们持有的几种特定的股票感兴趣。可采用两个完成不同任务的进程为证券交易所系统建模：

- 信息提供者进程不断地从一个外部源接收新的交易信息，并应用到相应的股票对象上。对股票对象的每次更新都是一个事件，经历这一事件的股票对象通知所有持有该股票的交易者。每个外部源都将会有一个单独的信息提供者进程。
- 交易者进程创建一个对象，用于表示用户请求显示的每支上市股票。这个本地对象订阅由相关信息提供者提供的代表那个股票的对象。然后它接收发给它的所有通知中的信息，并将其显示给用户。

通知的通信如图5-10所示。

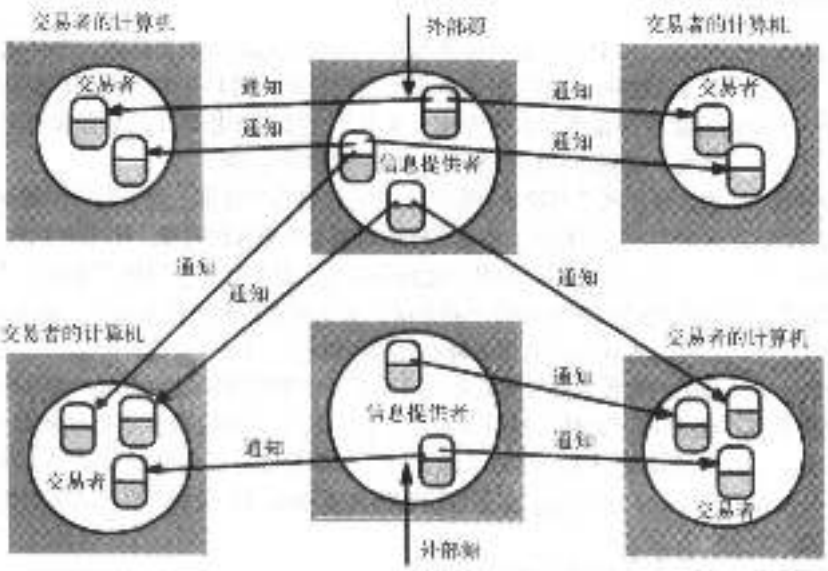


图5-10 证券交易所系统的结构

事件类型 一个事件源能生成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字或者标识符、操作、它的参数和时间（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。只要与属性匹配的事件发生，就会通知感兴趣的一方。在证券交易所系统例子中，存在一类事件（股票的更新信息到达），属性可以是股票的名字、它的现价、最新的涨幅或跌幅。交易者可以指定他们对与具有某个名字的股票相关的所有事件都感兴趣。

在5.4.1节中，我们将介绍一种体系结构，用以描述在分布式事件通知中参与者的作用。这种体系结构解决了事件通知的异构性与异步性。在5.4.2节中，Jini事件服务的实例研究便采用了这种

结构。16.3.1节将讨论在不稳定系统中，用于组件间数据通信的事件服务。在20.3.2节中，我们将在CORBA实例研究中给出有关CORBA事件服务的描述。 202~203

5.4.1 分布式事件通知的参与者

图5-11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所扮演的角色。本段内容摘自因特网上Rosenblum和Wolf [1997]关于事件和通知的论文。该体系结构用于发布者和订阅者之间的解耦合，以便发布者可以在独立于它们的订阅者的情况下进行开发，并且尽可能地限制订阅者施加于发布者的工作。其主要部件是事件服务，它维护一个已发布事件与订阅者兴趣的数据库。一个感兴趣的对象上的事件在事件服务上发布。订阅者向事件服务通告它们感兴趣的事件类型。当一个事件在一个感兴趣的对象上发生的时候，就发送一个通知到该类事件的订阅者。

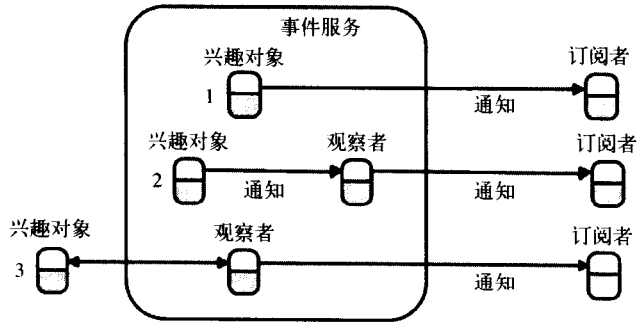


图5-11 分布式事件通知的体系结构

参与对象的角色如下：

兴趣对象：这是一个经历了由于调用操作而导致的状态改变的对象。它的状态改变可能令其他对象感兴趣。这一描述考虑到了像戴着主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谁在房间的记录中。兴趣对象如果传递通知的话，就被认为是事件服务的一部分。

事件：事件在兴趣对象上发生，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，它包含事件类型和属性，而属性通常包括兴趣对象的标识、调用的方法、发生时间或者一个序号。

订阅者：订阅者是一个订阅了另一对象上某些类型事件的对象。它接收关于该事件的通知。 204

观察者对象：观察者的主要作用是将兴趣对象和其订阅者解耦合。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，订阅者感兴趣的事件类型不相同，而类型需求相同的订阅者所感兴趣的属性值可能不相同。如果要求兴趣对象必须执行所有用于区分订阅者需求的逻辑，那么它就可能过于复杂。一个或多个观察者可以介入到兴趣对象与订阅者之间。观察者的角色，将在后续段落里详细讨论。

发布者：这是一个声明自身将生成某种类型事件通知的对象。发布者可以是一个兴趣对象或一个观察者。

图5-11给出了三种情形：

1）在事件服务中有一个兴趣对象，但没有观察者。兴趣对象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之外。在这种情形下，一个观察者会向兴趣对象查询事件何时发生，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——选择哪一个应该取决于应用需求。例如，如

果IP组播用于给一组接收者发送通知，那么故障模型将与4.5.1节中用于IP组播的故障模型有关，它不能保证所有接收者会接收到某个通知消息。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，如果最新状态在下一次更新时到达，对玩家也不会有太大的影响。

然而，其他应用会有更强的需求。考虑证券交易所的例子：为了公平对待对同一股票感兴趣所有的交易者，我们要求持有该股票的所有交易者都接收到相同的信息。这意味着应该使用可靠的组播协议。

在前面提到的Mushroom系统中，关于对象状态改变的通知能可靠地传输到服务器上，由服务器负责维护对象的最新复本。然而，通知也可以以不可靠的组播方式发送到用户计算机的对象复本上。如果在后一种情况下丢失了通知，那么用户还能从服务器上获得对象状态。当应用需要的时候，通知会被排序然后再可靠地传输到对象复本上。

某些应用有实时需求，其中包括核电站或医院病人监视系统中的事件。可能需要设计提供实时保证和可靠性且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。

观察者的角色 尽管通知可以直接从兴趣对象传输到接收者，但是处理通知的任务还是可能分
205 配到扮演不同角色的观察者进程中。我们给出一些例子：

转发：转发观察者可以代表一个或多个兴趣对象完成所有给订阅者发送通知的工作。一个兴趣对象需要做的所有事情就是给转发观察者发送一个通知，由转发观察者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预测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款有关，但是接收者只对金额超过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，它们可以指定自己感兴趣的事件的模式。模式指明了几个事件之间的关系。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：在有些情况下，通知需要延迟到潜在的订阅者准备接收它们才能发送。例如，与订阅者失去联系，或者对象需要再次激活的时候。这时，观察者可以扮演通知邮箱的角色，由它代表订阅者接收通知，只有等到订阅者准备好接收通知的时候才传递它们（在单独的一批中）。订阅者应该可以根据需要打开或者关闭这种传输功能。订阅者可以在向某兴趣对象注册的时候设置一个通知邮箱，将通知邮箱指定为通知发送的目的地。

5.4.2 实例研究：Jini分布式事件规约

由Arnold等人[1999]描述的Jini分布式事件规约允许一个Java虚拟机（JVM）中的潜在订阅者订阅和接收另一个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兴趣对象与订阅者之间。Jini分布式事件规约里的主要对象包括：

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。

对象订阅事件的方式如下：首先向事件生成者通知事件的类型，然后将远程事件监听者指定
206 为通知发送的目标。

Java RMI用于从事件生成者向订阅者发送通知，这可能要借助一个或多个第三方代理。设计者规定事件监听者应该尽快回答通知调用，以免延误事件生成者。它们在返回之后再处理通知。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notify方法。订阅者和第三方代理实现RemoteEvent-Listener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：

- 对一个已经发生了事件的事件生成者的引用。
- 一个事件标识符，它指定事件生成者上的事件类型。
- 一个序号，它应用在那种类型的事件上。序号应该随着事件的发生而递增。它可以让接收者从一个给定的来源对某种类型的事件进行排序，或者避免两次应用同一个事件。
- 一个编码对象。它在接收者订阅那类事件时提供，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，这个编码的对象中可以包含一个终止操作，这个操作可以在接到相应通知时被执行。

EventGenerator：这个接口提供一个称为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，随着每个通知返回订阅者。
- 对一个事件监听者对象的远程引用——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订阅的问题。租期过期后可以续订。

根据Jini规约，EventGenerator接口就是订阅者可以用来在兴趣对象上注册兴趣事件的接口的例子。有些应用可能需要不同的接口。

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可以扮演多种有用的角色，包括上述的那些角色。

在最简单的情形下，一个订阅者在事件生成者上注册感兴趣的事件类型，并将自己指定为远程事件监听者。这对应于图5-11中的情形1。 207

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进行可靠的通知传递。

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对某类事件的兴趣，将第三方代理链的第一个代理指定为通知要发送到的地方。例如，订阅者可以安排由第三方代理存储发送给它的通知，一直到订阅者准备好之后才接收这些通知。第三方代理可能还会负责续订租借。

5.5 实例研究：Java RMI

Java RMI扩展了Java的对象模型，以便为Java语言中的对分布式对象提供支持。特别是，它允许对象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，发出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，因此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——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。不过，即使在一个单语言系统中，远程对象的程序员也必须考虑它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。接下来讨论如何为该例子接口创建客户和服务器程序。然后研究Java RMI的设计和实现。关于Java RMI的详细介绍，请参见关于远程调用的教程 [Java.sun.com I]。

在这个实例研究和第20章的CORBA实例研究，以及第19章中关于Web服务的研究中，我们均使用*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，该绘图区域可以包含图形对象，例如矩形、线、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，用于把用户最新绘制的图形告诉服务器，并记录它接收到的所有图形。服务器也提供相应操作，让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
208 个版本号（一个整数），每当新图形到达的时候，版本号就递增，并赋予给该新图形。服务器还提供有关操作，让客户询问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。

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异常，但是也可以抛出特定于应用的异常。图5-12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（如类型、位置、外接矩形、线条颜色和填充颜色）的类，该类提供访问和更新图形对象状态的操作。GraphicalObject必须实现Serializable接口。首先考虑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。需要注意的重要一点是，普通对象和远程对象都可以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。在后文中，我们讨论普通对象和远程对象怎样作为参数和结果传递。

```
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5-12 Java远程接口Shape和ShapeList

传递参数和结果 在Java RMI中，假定方法的参数为输入型的参数，而方法的结果是一个输出型参数。4.3.2节讲述了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（即它实现了Serializable接口）都能作为Java RMI中的参数或结果传递。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，相应的参数或者结果总是作为远程对象引用传递。例如，在图5-12的第2行中，newShape方法的返回值就定义为Shape——一个远程接口。当接收到一个远程对象引用时，它就可用于在它所指的远程对象上进行
209 RMI调用了。

传递非远程对象：所有可序列化的非远程对象是在复制之后以值方式传递。例如，在图5-12（第2行和第1行）中，newShape的参数和getAllState的返回值都是GraphicalObject类型，它是可序

列化的并且以值方式传递。当一个对象以值方式传递时，就要在接收者的进程中创建一个新对象。这个新对象的方法可以在本地调用，但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方法给服务器传递一个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。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，并有以下改变：

1）一旦一个实现Remote接口的对象被序列化，它就被它的远程对象引用代替，该远程对象引用包含它（远程对象）的类的名字。

2）任何一个对象被序列化的时候，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，这与以远程调用方式通信的分布式对象尤其相关。我们已经看到，非远程对象以值方式传递，而远程对象以引用方式传递RMI的参数和结果。如果接收者还没拥有以值方式传递的对象类，那么它就会自动下载类的代码。类似地，如果远程对象引用的接收者还没拥有代理类，那么代理类的代码也会自动下载。这样做有两个好处：

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，客户和服务器程序能透明地使用它们的实例。

例如，考虑白板程序并假设GraphicalObject的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。之后，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的一个实例必须运行在每个驻留了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风格的名字映射到驻留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URL格式的字符串作为参数：

//computerName：port/objectName

210

其中，computerName和port指向RMIregistry的地址。如果它们被省略的话，就被认为是本地计算机和默认端口。RMIregistry的接口提供如图5-13所示的方法，该方法中没有列出异常——所有的方法都可以抛出RemoteException异常。这项服务不是系统范围的绑定服务。客户必须将它们的lookup查询导向到特定的主机。

void rebind (String name , Remote obj)

服务器用这个方法以名字注册一个远程对象的标识符，如图5-14的第3行所示。

void bind (String name , Remote obj)

服务器可以选择使用这个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了一个远程对象引用上，那么它会抛出一个异常。

void unbind (String name , Remote obj)

这个方法删除一个绑定。

Remote lookup (String name)

客户可以使用这个方法以名字查找一个远程对象，如图5-16的第1行所示。

它返回一个远程对象引用。

String [] list ( )

这个方法返回一个Strings数组，该数组包含绑定到注册表里的名字。

图5-13 Java RMIregistry的Naming类

5.5.1 创建客户和服务器程序

本节将以图5-12所示的远程接口Shape和ShapeList为例，概述创建使用远程接口的客户和服务器程序的步骤。服务器程序是实现Shape和ShapeList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，它可用于避免轮询服务器。本节阐述的这几个类的完整版本可以参见www.cdk4 .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接口的伺服器，它拥有图形对象的状态和它的版本号；它以另一个实现ShapeList接口的伺服器代表它的图形集，并把图形集存放在Vector中。

为实现它的每个远程接口，服务器包括一个main方法和一个伺服器类。服务器的main方法创建ShapeListServant的一个实例，并将它绑定到RMIregistry中的一个名字上，如图5-14所示（第1行和第2行）。注意，绑定到名字的值是一个远程对象引用，它的类型是它的远程接口ShapeList的类型。两个伺服器类是ShapeListServant（它实现ShapeList接口）和ShapeServant（它实现Shape接口）。
211 图5-15给出了ShapeListServant类的轮廓。注意，ShapeListServant（第1行）与许多伺服器类一样，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。（一个可被激活的对象可以继承Activetable类。）

```
import java.rmi.*;
public class ShapeListServer {
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();          1
            Naming.rebind("ShapeList", aShapeList);                 2
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

图5-14 带有main方法的Java类 ShapeListServer

伺服器类中远程接口方法的实现非常简单，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5-15中的newShape方法（第2行），它可以称为一个工厂方法，因为它允许客户请求创建一个伺服器。它使用ShapeServant的构造函数，该构造函数创建一个新的伺服器，其中包含作为参数传递的GraphicalObject和版本号。NewShape的返回值的类型是Shape——由新的伺服器实现的接口。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行）。

```
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5-15 Java类ShapeListServant实现ShapeList接口

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，但是它不同于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类路径（classpath）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。

客户程序 ShapeList服务器的一个简化的客户见图5-16所示。任何客户程序都需要从使用绑定程序查找远程对象引用开始。我们的客户设置了一个安全性管理者，然后使用RMIregistry的lookup操作（第1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。在我们的例子中，客户调用远程对象的allShapes方法（第2行），并接收一个当前存储在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，以获取向量中的每个图形对象，并将它们显示在窗口中。每次用户绘制完图形对象后，它就调用服务器里的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，以查找别的用户是否已经添加了一些新的图形。如果有新图形，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());}
   }
}
```

图5-16 ShapeList的Java客户

212
213

回调 回调的基本思想是，客户不用为找出某个事件是否已经发生而轮询服务器，而是当事件发生时，由服务器通知它的客户。回调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客户的回调对象的远程对象引用，服务器将这些引用记录在一张列表中。
- 一旦感兴趣事件发生，服务器就调用感兴趣的客户。例如，白板服务器会在添加了一个图形对象的时候调用它的客户。

使用回调可以避免客户轮询服务器上的兴趣对象，但它有以下缺点：

- 服务器的性能会因为时常的轮询而降低。
- 客户不能及时通知用户已经做了更新。

然而，回调也有它自身的问题。首先，服务器需要有客户回调对象的最新列表，但是客户并

不总是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。利用5.2.6节介绍的租借技术可以解决这个问题。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。WriteboardCallback接口可以定义为：

```
public interface WhiteboardCallback implements Remote {
    void callback ( int version ) throws RemoteException ;
} ;
```

由客户将该接口作为远程对象实现，使服务器能在添加了新对象的时候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接口还要求一些附加的方法，例如register和deregister，其定义如下：

```
int register ( WhiteboardCallback callback ) throws RemoteException ;
void deregister ( int callbackId ) throws RemoteException ;
```

在客户取得了对具有ShapeList接口的远程对象引用（例如图5-16中的第1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）代表注册。当客户完成的时候，它会调用deregister通知服务器它不再
214 请求回调了。服务器负责维持一张感兴趣客户的列表，并在每次版本号增加的时候通知所有客户。

5.5.2 Java RMI的设计和实现

最初的Java RMI系统使用了图5-7中的所有组件。但是在Java 1.2中，使用了反射机制来创建通用的分发器并避免骨架的使用。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来创建——不再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被调用方法的信息。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方法的特征，包括它的类、它的参数类型、返回值和异常。这个类的最大特点是Method的实例能通过它的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，第二个参数是一个包含参数的Object数组。结果作为Object类型返回。

再回到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。对于方法，代理将它编码成Method类的一个对象。它把参数放入一个Objects数组中，然后编码该数组。分发器从请求消息中解码Method对象和它在Objects数组中的参数。通常，目标对象的远程引用已经被解码，对应的本地对象引用已从远程引用模块中获得。然后，分发器用目标对象和参数值数组调用Method对象的invoke方法。执行方法后，分发器将结果或者出现的异常编码到reply消息中。

这样，分发器是通用的。也就是说，相同的分发器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5-17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，Remoteserver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，另一个要提供的是Activatable，现在用于提供主动对象。其他选择可能就是提供复制对象了。RemoteServer类是RemoteObject类的一个子类，它的实例变量有一个远程对象引用，并提供如下的方法：

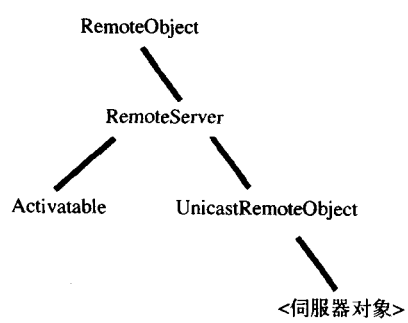


图5-17 支持Java RMI的类

equals：这一方法用于比较远程对象引用。

toString：这一方法用于以String类型给出远程对象引用的内容。

readObject、writeObject：这些方法用于序列化/解序列化远程对象。

另外，instanceOf操作符能用于测试远程对象。

5.6 小结

本章讨论了两种分布式编程的范型——远程方法调用和基于事件的系统。这两种范型都把分布式对象看成是能够互相通信的独立实体。在第一种情形下，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；在第二种情形下，每次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到多个订阅者。

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自己的状态，而且方法的本地调用可以扩展到远程调用。分布式系统中的每个对象都有一个远程对象引用（一个全局唯一的标识符）和一个指定它的哪个操作可以被远程调用的远程接口。

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提供恰好一次语义，因为两个参与的对象在不同的计算机上，它们可能分别出故障，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。由于它们的故障和性能特征不同，加之对远程对象并发访问的可能性，让远程调用与本地调用的表现完全相同未必是个好想法。

RMI中间件实现提供了代理、骨架和分发器等组件，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。这些组件可以由接口编译器生成。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称为Remote的接口来指定，并让每个方法抛出一个RemoteException异常。这可以确保让程序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的相互通信。不同于RMI，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。接口的简单性使得给已经存在的对象增加事件变得很容易。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而添加到系统的第三方对象。 215～216

练习

5.1 Election接口提供两个远程方法：

vote：带有两个参数，客户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刚好只投票一次的整数）。投票者编号在整数的范围内随机地选择，以便不会被他人轻易地猜中。

result：带有两个参数，服务器通过这两个参数给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？ （第179页）

5.2 讨论在TCP/IP连接之上实现请求－应答协议的时候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，既不丢失数据也不复制数据。考虑导致连接中断的所有条件。

（4.2.4节，第187页）

5.3 以CORBA IDL和Java IDL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型和输出型参数的方法。 （图5-2，图5-12）

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服务上使用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？你认为应该使用至多一次调用语义吗？（第188页）

5.5 在一种带有遗漏故障的通信服务上实现请求－应答协议，以提供至少一次的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。用哪种方法能简化后者的实现？（第187页）

5.6 在Election服务中，要确保在多个客户并发访问时，选举记录能保持一致。简述其如何实现。（第198页）

5.7 Election服务必须确保安全地存储所有的选票，即使服务器进程崩溃也是如此。参考练习5.6，解释如何实现这一点。（第193页～第194页）

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，它应该用以下基调调用doOperation方法：

byte [] doOperation (RemoteObjectRef o , Method m , byte [] arguments)

217 提示：代理类的一个实例变量应该具有一个远程对象引用（见练习4.12）。（图4-15，第215页）

5.9 解释如何根据CORBA接口定义（练习5.3中给出的）使用像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4-15中定义的doOperation方法。（第192页）

5.10 解释如何使用Java 反射构造一个通用的分发器。给出具有下列基调的分发器的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。（第215页）

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要求分发器在调用方法之前将字节数组转化成一个Objects数组。讨论具有下列基调的doOperation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。这些改变会如何影响分发器的设计？（4.3.2节，第215页）

5.12 一个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。本地操作系统操作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。

计算下列情况下客户创建和返回消息所花费的时间：

（1）如果它是单线程的。

（2）如果它有两个线程，这两个线程能在一个处理器上并发地发出请求。

你可以忽略其他上下文转换的时间。如果客户和服务器处理器是线程化的，就需要异步RPC吗？（第193页）

5.13 设计一个支持分布式无用单元收集和在本地对象引用与远程对象引用之间转化的远程对象表。给出一个例子，其中涉及在不同地址上的几个远程对象和代理，以阐述该表的使用。给出当调用导致创建新代理时表的变化。然后给出当一个代理不可用时表的变化。（第195页）

5.14 5.2.6节描述了分布式无用单元收集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
218 信故障和进程故障可能造成的影响。提出解决每种影响的建议，但是不能使用租借。（第195页）

5.15 讨论在共享白板应用程序中，如何使用Jini分布式事件规约中描述的事件和通知。RemoteEvent类由Arnold等人[1999]定义如下：

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。用通知告诉监听者事件已经发生，由监听者负责获取更进一步的细节。 （第206页，第208页）

5.16 设计一种通知邮箱服务，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要求传递通知。解释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掉的情形？ （第206页）

5.17 解释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。 （第205页）

5.18 提出一种观察者能使用的方法，以增强为练习5.15所提供的解决方案的可靠性或性能。 （第205页） 219

第6章　操作系统支持

本章介绍在分布式系统的节点上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，其中包括通信和调度。

系统内核的作用是本章的一个重要内容。本章的目标是使读者了解在保护域中划分功能的优点和缺点，特别是划分内核级和用户级代码的功能。本章还将介绍内核级设施与用户级设施间的平衡，其中包括效率和健壮性之间的关系。

220 ~ 221

本章还将探讨多线程处理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结构。

6.1 简介

第2章已经介绍了分布式系统中的主要软件层次。我们已经知道，资源共享是分布式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节点上或另一进程上。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）使用中间件来进行交互。中间件提供了分布式系统的各节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17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。本章会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。这些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）之上的面向问题的抽象。例如，UNIX（及其衍生版本，如Linux）或Windows（及其各个版本，如XP）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。操作系统管理某节点的物理资源并通过系统调用接口抽象地表示这些资源。

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。它们的定义虽然不同，但它们的概念却有些相似。下面将对它们加以介绍。

UNIX和Windows 都是*网络操作系统*。它们都具有网络连接功能，因此可以用它们来访问远程资源。它们能网络透明地访问一些资源（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，许多用户所需的文件是存储在远端或服务器上的文件，而对于应用程序来说这些文件是位置透明的。

网络操作系统的特点是运行于其上的节点能独立地管理自己的进程资源。也就是说，在网络的每一个节点上都有多个系统映像。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其上运行进程。然而，与操作系统管理本地节点计算机上的进程不同，网络操作系统并不在节点间调度进程。

相反，可以设想存在这样一种操作系统，通过它用户不必关心程序的运行地点和资源位置，也就是说，网络上只有一个*单一系统映像*。这种操作系统必须能够控制系统中所有的节点，并且

222

它能透明地将新的进程定位在符合调度策略的节点上。例如，它能在负载最小的节点上生成新的进程以防止单个节点过载。

如果一个操作系统能如上所述在分布式系统中，对所有资源只生成单一系统映像，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Windows这些网络操作系统外，几乎没有普遍应用的分布式操作系统。有两个主要原因造成这种情况，其一是用户已经为能够满足他当前需要的应用软件进行了大量的投资，所以无论新的操作系统有多么优越的特性，如果它不能运行这些应用软件，用户也不会使用它。有人尝试在新的系统核心上模拟UNIX和其他操作系统的内核，但模拟的性能却不能令人满意。而且，模拟所有的主流操作系统本身就是一项繁重的工作。

第二个原因是，即使在一个小单位里，用户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]的原因，例如，当Jones写一个文档时，她需要很好的交互性，而如果系统由于运行了Smith的程序而使交互变慢，她必定会不高兴。

中间件和网络操作系统的结合为独立性和网络资源透明访问之间提供了平衡。网络操作系统使用户能运行他的字处理程序和其他独立运行的程序。中间件使他们能享受到分布式系统所提供的服务。

下一节将介绍操作系统层的功能。6.3节介绍资源保护的低层机制，以便我们能了解进程和线程之间的关系，以及内核的作用。6.4节介绍进程、地址空间和线程，其中主要介绍并发、局部资源的管理保护和调度。6.5节介绍调用机制的一部分——通信。6.6节介绍不同类型的操作系统的体系结构，其中包括整体内核和微内核设计。读者可以在www.cdk4.net/oss上找到有关Mach内核以及Amoeba，Chorus和Clouds操作系统的实例分析。

6.2 操作系统层

只有当中间件和操作系统的联合具有良好的性能时，用户才会满意。由于分布式系统的每个节点都有各自的操作系统和硬件设备（平台），因此中间件需要运行在不同的操作系统和硬件组合上。运行在节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，这些操作系统能为远程、存储和通信提供本地硬件资源的抽象。中间件将这些本地资源联合起来以实现在不同节点的对象和进程之间提供远程调用的机制。 223

图6-1给出了两个节点上的操作系统层如何支持一个公共的中间件层，从而为应用和服务提供一个分布式基础设施。

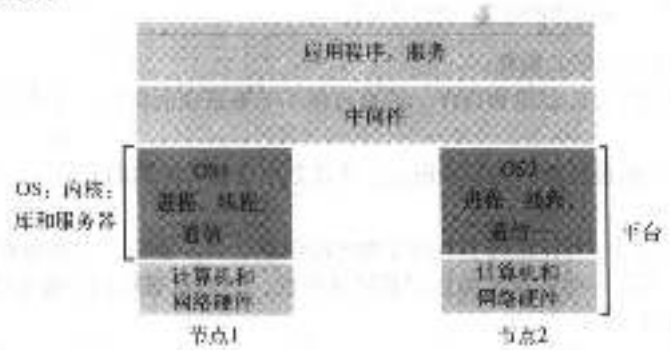


图6-1 系统层次

本章的目标是讨论某种操作系统机制对中间件提供共享分布式资源能力的影响。内核和运行于其上的客户和服务器进程是我们所关心的主要组件。内核和服务器进程用于管理资源和为客户提供资源接口。因此，它们至少应该具备以下特点：

共享：它们应该提供有用的能够访问资源的服务接口，也就是说，它所提供的操作集必须满足客户的需要，而像内存管理和设备管理这样由于实现资源的服务的细节应该对客户隐藏。

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，而且

应用程序进程也不能访问设备寄存器。

并发处理：客户可以共享资源并能并发地访问它们。并发透明性由资源管理器负责实现。

客户可以通过远程方法调用访问一个服务器对象，或者通过系统调用访问内核。我们将访问一个已封装资源的方式称为调用机制，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：资源管理器接收来自于网络上或计算机内部的操作参数并返回结果。

调度：当一个操作被调用时，必须在内核或服务器上调度相应操作。

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，内核实现上述全部功能。

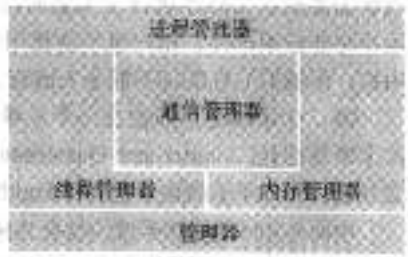


图6-2 核心OS功能

在可能的情况下，操作系统软件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代码是用C、C++或Modula-3这样的高级语言编写的，而且操作系统的各个部分是分层实现的，从而可以将依赖于机器的组件作为底层实现。一些内核可以在共享内存的多处理器上执行，下面将对其做详细的介绍。

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模块（RAM）。处理器也可以有自己的内存。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方式是在单台个人计算机的主板上包含若干（2—8）处理器来实现多处理器系统。

在常见的对称处理体系结构上，每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。这些内核共享关键的数据结构（例如可运行线程队列），但它们也拥有一些私有的数据。处理器能同时执行各自的线程，也可以同时访问共享内存中的私有（受硬件保护的）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，故它特别适合于高性能服务器的实现，例如提供访问共享数据库的服务（见6.4节）。

224
|
225

操作系统包括以下几个核心组件：

进程管理器：负责进程的创建和操作。进程包括一个地址空间以及一个或多个线程，是资源管理的基本单位。

线程管理器：负责线程创建、同步和调度。线程是与进程相关的调度活动，这将在6.4节详细描述。

通信管理器：负责同一台计算机上不同进程中的线程之间的通信。一些内核也支持远程进程的线程之间的通信。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节将讨论通信的设计。

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拷贝和数据共享。

管理器：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。这就是Windows中的硬件抽象层。读者可以在Bacon[2002]和Tanenbaum[2001]中找到内核中依赖计算机的那一部分的详细描述。

6.3 保护

上文曾经提到需要保护资源以防止非法访问，值得注意的是，对系统完整性的威胁不仅仅来

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曾预料的行为而导致系统工作异常。

为了使读者了解什么叫对资源的“非法访问”，下面将以文件为例子进行讨论。假设对打开的文件只有两种操作——read和write，那么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，他就有对文件的read权限和write权限，而琼斯只能对此文件执行read操作。当琼斯试图对文件进行write操作时，这便是一个非法访问。完全解决分布式系统的资源保护子问题需要运用密码技术，本书的第7章将对此进行介绍。

另外一种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，当史密斯或琼斯试图执行一个既不是read也不是write的操作就会产生这种非法访问。假设当史密斯设法直接访问文件指针变量时，他可以构造一个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。 226

我们应该能保护资源来防止像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，一个模块只能访问它所引用的目标模块，而不能像C或C++那样通过指针来访问一个模块，并且它只能用其对目标模块的引用来执行提供的可用调用（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，而不用考虑调用模块是用什么样的语言写成的。如果要在通用的计算机上实现这种保护机制，就需要有相应的系统内核支持。

内核和保护 内核不同于其他计算机程序，它的特点是一直保持运行并且对其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，这就使得其他程序必须通过内核允许的方式来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，它们的设置决定了特权指令能否被执行。例如有些寄存器的值可以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。

内核也通过建立地址空间来保护自己和其他进程以防止异常进程的访问，同时也为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访问权限，例如只读或读写权限。进程不能访问自己地址空间以外的内存空间。术语*用户进程*或*用户级进程*表示在用户模式下执行并且拥有用户级地址空间的进程（相对于内核，这些进程有受限的内存访问权限）。

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。通过中断和*系统调用陷阱*（一种由内核管理的资源调用机制），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转换。系统调用陷阱由一个机器级的TRAP指令实现，它将处理器转换为管理模式并将地址空间切换到内核地址空间。当TRAP指令（具有某种异常）执行时，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。

保护机制使系统在执行程序时会产生额外的开销。在地址空间之间切换会占用处理器的许多处理周期，并且系统调用陷阱也要比简单的过程调用或方法调用耗费更多的处理器资源。我们将会了解到这些不利因素是如何影响调用开销的。 227

6.4 进程和线程

在传统的操作系统概念中，进程只能执行一个活动。到20世纪80年代，人们发现这一特性不

能满足分布式系统的要求，也不能胜任那些需要内部并发的复杂的单机应用。主要问题是在传统的进程中实现相关活动之间的共享是很困难的，而且代价也很大。

对此问题的解决方法是完善进程的概念使它能与多个活动联系起来。现在，一个进程是由一个执行环境和一个或多个线程组成的。一个线程是一个活动的操作系统抽象（这一术语来源于术语“执行线”）。执行环境是资源管理的基本单位，它是一个进程的线程所能访问的由本地内核管理的资源集。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代价很高，而多个线程可以共享执行环境。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，一个执行环境可以表示为运行于其中的线程的保护域。

线程可以动态地创建和销毁。多线程执行的主要目的是尽可能增加操作间并发执行程度，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。多线程执行对服务器端运行的程序特别有用，因为处理多个并发的客户请求会降低服务器的执行速度，使其成为瓶颈。例如，一个线程可以处理一个客户的请求，而同时为另一个客户请求服务的线程要等待磁盘访问的完成。

执行环境可以保护其内部的资源不被外部线程访问，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，某些内核允许有条件地共享资源，例如同一计算机上不同执行环境的物理内存。

因为许多老式的操作系统在一个进程上只允许运行一个线程，所以我们使用术语多线程进程来强调这一区别。容易引起混淆的是，在一些编程模型和操作系统中，术语“进程”实际是指我们这里所说的线程。读者也可能在其他文献中遇到过术语重量级进程。其中就包含了它的执行环境，而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和其执行环境。

228

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的有趣比喻。一个执行环境是一个装有空气和食物且封了口的瓶子。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，也可以杀死它生出的这些苍蝇；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），但它们必须有顺序的消耗资源。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（通过发送消息）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，且瓶子中只有一只不能生出其他苍蝇的苍蝇。

6.4.1 地址空间

前面已经介绍过，地址空间是一个进程的虚拟内存的管理单元。它可以很大（通常可达到2^{32}字节，有时可达到2^{64}字节），同时可以拥有多个区域，这些区域被不可访问的虚拟内存区隔开。一个区域（图6-3）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。我们要注意区分区域和它们的内容。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）。
- 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，这个模型是基于页面的而不是基于段的。与段不同的是，当区域



图6.3 地址空间

扩展时，它们最终会重叠。区域之间留有空隙，用于区域增长。可以将UNIX地址空间概括为由若干不相交区域组成的地址空间，它包含3个区域：一个固定的包含程序代码的不可更改的正文区域；一个堆，其中一部分可以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。通过给每一个栈分配一个区域，系统就能检测栈的溢出并控制栈的增长。试图访问在这些栈之外的没有被分配的虚拟内存将会引起异常（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栈的溢出。

另一个因素是它能将所有的文件——而不仅仅是二进制文件的正文和数据区——映射到地址空间。*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反映到底层的文件存储上。Section CDK3-18.6（www.cdk4.net/oss/mach）描述了Mach内核是如何扩展虚拟内存的，以便使区域对应于任意的“内存对象”而不仅仅是文件的。

在进程之间或在进程与内核之间共享内存的需求是产生地址空间中额外区域的另一个原因。*共享内存区域*（或简称为*共享区域*）是同一片物理内存区域，并可以作为其他地址空间的一片或多片内存区域。因此，进程通过访问这些共享内存区域可以获得一致的内存内容，而它们非共享的区域仍然是受到保护的。共享区的应用包括如下几个方面：

库：库的代码可以很大，因此如果每一个使用它的进程都需要独立地装载这个库，那么就会占用相当大的内存。相反，可以将库代码的一个拷贝映射到需要它的多个进程的共享内存区中，以达到共享的目的。

内核：通常，内核代码和数据会被映射到每一个地址空间中的相同位置。这样，当进程进行系统调用或出现异常时，系统不需要切换到新的地址空间映射。

数据共享和通信：两个进程之间或进程和内核之间可能需要共享数据以达到协同工作的目的。将共享数据映射到相应的两个地址空间中的特定区域比将共享数据放在消息中传递的效率更高。6.5节将介绍如何将区域共享用于通信。

229 ~ 230

6.4.2 新进程的生成

一般而言，新进程的创建是由操作系统提供的一个原子操作。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拷贝得来的（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程序代码的进程。

在分布式系统中，设计进程创建机制时必须考虑到多个计算机的使用。因此，支持进程的基础设施被划分为几个独立的系统服务。

在分布式系统中，新进程的创建过程可以被划分为两个独立的阶段：

- 选择目标主机。例如，系统可以在作为服务器的计算机集群中选择一个节点作为进程的主机（见下面的介绍）。
- 创建执行环境（和一个初始线程）。

进程主机的选择 选择新进程驻留的节点（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产生进程的主机上运行新进程到在多个计算机上共同分担处理负载等一系列策略。Eager等人[1986]给出了一个负载共享策略的分类。

> **集群** 集群是由速度可达Gbps的交换以太网这样的高速通信网连接起来的计算机集合（有时达到上千台）。其中的计算机可以是标准PC机或工作站，也可以是由插有多个PC 处理器的主板组成的计算机；它们可以是单处理器也可以是多处理器。集群可以提供高可用性和高可伸缩性的服务，例如在因特网上为用户提供的搜索引擎，而这是通过在集群的处理器之间复制或分解处理和服务器的状态来实现的[Fox et al. 1997]。集群也可以用来运行并行程序[Anderson et.al. 1995, now.cs.berkeley.edu, TFCC]。

转移策略决定是使新进程在本机运行还是在其他机器上运行，而这取决于本机节点的负载是轻还是重。

定位策略决定选择哪一个节点来驻留被转移的新进程，这取决于节点的相对负载情况、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]都为用户提供了相应的命令，可以在操作系统选择的当前空闲的工作站（在任意时刻，通常会有很多这样的机器）上执行一个程序。在Amoeba系统[Tanenbaum et al. 1990]中，运行服务器从一个共享处理器池中为每个进程选择一个处理器作为主机。在上面的例子中，如何选择目标主机对程序员和用户来说是透明的。然而，对那些并行程序或容错程序编程可能需要指定的进程定位。

进程定位策略有两类——静态的或适应性的。前者不考虑系统的当前状态，尽管它们是根据系
231 统的长期特点设计的。它们是基于数学分析的，其目的是优化像处理器吞吐量这样的系统参数。它们可能是确定性的（“节点A总是将进程转移给节点B”）也可能是非确定性的（“节点A应该将进程随机转移给节点B～E之间的任何节点”）。另一方面，适应性策略根据不确定的运行时因素（例如，每个节点的负载）采取启发式方法来做出进程分配决定。

负载共享系统可能是中心化的、层次化的或分散化的。中心化的系统有一个负载管理器组件，而层次化的系统中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节点的信息并根据这些信息将新进程分配到节点上。在层次化系统中，负载管理器尽可能将分配进程的决定权交给它的树形结构中的底层节点上，但是管理器在某些负载条件下也可以通过与其他管理器的公共祖先节点将进程转移到其他节点上。在分散化的负载共享系统中，节点之间为制定进程分配决策可直接交换信息。例如，Spawn系统[Waldspurger et al. 1992]将节点看作计算资源的“购买者”和“销售者”，并用（分散化的）“市场经济”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节点负责启动转移决策。如果它的负载超过了某一阈值，它就会启动一个转移过程。相反，在接收方启动的负载共享算法中，节点在自己的负载低于某一阈值时建议其他节点将工作转移给自己。

可迁移的负载共享系统可以在任一时间转移负载，而不仅限于在创建一个新进程时转移负载。它们使用一种称为“进程迁移”的机制，将一个正在执行的进程从一个节点转移到另一个节点。Milojicic等人[1999] 通过一系列论文详细描述了进程迁移和其他类型的移动。尽管现在已经构建出一些进程迁移机制，但它们并没有得到广泛应用，其中一个主要的原因是它们的代价高昂，而且为了将进程转移到其他的节点上，系统需要从内核中提取出进程运行的当前状态，而实现这种操作是相当困难的。

Eager 等人[1986]考察了三种负载共享的方法，从而总结出：在任何负载共享机制中，简单性是一个很重要的性质。这是因为高额的开销（例如状态收集开销）可能会抵消其带来的好处。

创建新的执行环境 一旦选定了主机，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。

有两种方法可以为新创建的进程定义和初始化地址空间。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域、一个堆和一个栈。在这种情况下，地址空间区域可根据指定了地址空间区域范围的列表来创建，然后地址空间区域由一个可执行文件进行初始化或者用零填满。

第二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 fork操作的语义中，新创建的子进程共享其父进程的正文区域，同时，它的堆和栈（以及初始内容）在某种程度上是
232 父进程的拷贝。这个机制可以加以推广使子进程继承（或忽略）父进程的地址空间中的每一个区域，被继承的区域可以共享父进程的区域，也可以逻辑地拷贝父进程的区域。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）同时被映射到相应的子区域中。

例如，Mach[Accetta et al. 1986]和Chorus[Rozier et al. 1988，1990]采用了一种称为写时复制的优化机制来对父进程的区域复制到子进程区域的过程进行优化。默认情况下，区域被复制，但是没有进行物理拷贝，组成继承区域的页面帧被两个地址空间共享。只有当其中的一个进程试图修改内存区的页面内容时，系统才进行物理上的页面拷贝。

写时复制是一种通用的技术，例如，它也用于拷贝大量信息。下面将介绍它的操作机制。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。更明确地说，由于进程B创建的区域RB是通过复制继承父进程A的区域RA得到的。

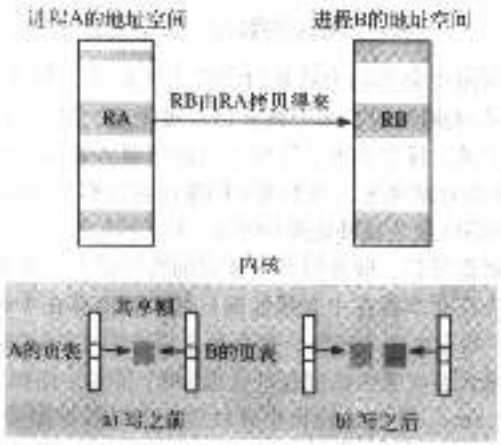


图6-4 写时复制

为简单起见，假设区域A中的页面都在内存中。初始情况下，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，但是页面最初是在硬件级被写保护的。如果某一个进程的线程试图修改数据，就会产生一个称为页失配的硬件异常。假设进程B试图写内存，页失配处理程序会为进程B分配一个新的帧，并将原帧中的数据以字节为单位拷贝到这个新的帧中。同时，在进程B的页表中的那个旧的帧号被新的帧号所代替，而在进程A的页面表中的页面帧号不变。此后，进程A和进程B的对应页都在硬件级被设为可写，在完成了以上操作后，进程B的修改指令就可以运行了。 233

6.4.3 线程

进程的另一个重要的方面是它的线程。本节主要介绍使客户和服务器进程拥有多个线程所带来的好处，然后会用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。

考虑图6-5所示的服务器（稍后将介绍客户）。服务器拥有一个包含一个或多个线程的线程池，其中每一个线程反复地从队列中取出已收到的请求并对其进行处理。本节暂不讨论是如何接收请求并将其排队以等待线程处理的。并且为了简单起见，我们假设每一个线程都采用同样的过程来处理请求。假设每一个请求平均占用2ms的处理时间和8ms的I/O延迟，其中I/O延迟是由于服务器从磁盘上读取信息造成的（假设此系统没有缓存）。同时假设服务器在一个单处理器的计算机上执行。

下面以处理器每秒处理的客户请求数为度量，讨论不同数目的线程运行时服务器的最大吞吐量。如果只有一个线程来执行所有的处理，因为执行一个请求平均需要2＋8＝10ms，那么服务器在1s内能处理100个客户请求。当服务器在处理一个请求时，任何新到达的请求将在服务器端口上排队。

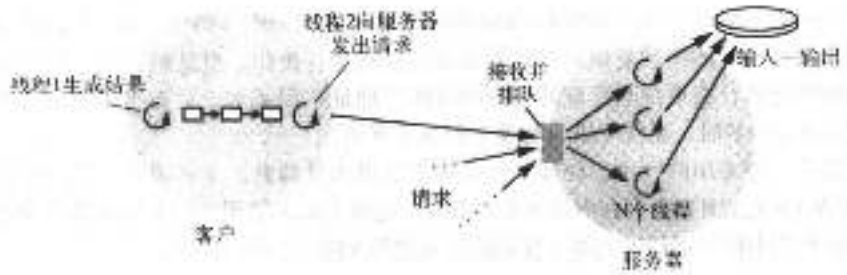


图6-5 拥有线程的客户和服务器

现在考虑服务器的线程池中包含两个线程时会发生的情况。假设每个线程都是独立调度的，也就是说，当一个线程因为I/O阻塞时，另一线程仍可被调度。这样当第一个线程被阻塞的时候，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。遗憾的是，在我们的例子中，线程会被单一的磁盘存取阻塞。如果所有的磁盘请求都被串行化，且每一个请求用8ms来
234 执行，那么，服务器最大的吞吐量为每秒处理1000/8=125个请求。

现在假设系统中加入磁盘缓存。服务器将它读到的数据放在其地址空间的缓冲区内。当服务器线程检索数据时，它首先在共享缓存中查找数据，如果数据存在于缓存中，就不需要再访问磁盘。如果在缓存中数据的平均命中率为75%，那么每个请求的平均I/O时间减少为（0.75×0+0.25×8）=2ms。于是理论上最大吞吐量将达到每秒处理500个请求。但如果由于缓存的原因，每个请求的平均处理时间增加到2.5ms（在每次操作中寻找缓存中的数据需要耗费额外的时间），那么系统将无法达到上面的理想情况下的吞吐量。这时，由于处理器的限制，服务器每秒只能处理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，则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。其共享的执行环境可以在共享内存中实现，并且多个线程可以在多个处理器上运行。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单独被调度，那么最多有两个线程可以并行地处理请求。由此可以计算出：两个线程每秒可以处理444个请求，而使用三个或更多的线程，因为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是如何增加服务器的吞吐量，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]总结的结果，他描述了CORBA的对象请求代理（ORB）的多种实现的线程体系结构。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”线程池以便在启动时处理请求。在图6-5中“接收并排队”模块通常由一个I/O线程实现，该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上以便工作线程检索。

有时候，系统需要按不同的优先级处理请求。例如，一个公司的Web服务器可以根据产生请求的用户类别来优先处理某些请求[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来处理不同的请求优先级，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缺乏灵活性。在上文提出的例子中，如果池中的工作线程数量太少，则不能及时处理每一个用户请求。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。

在一请求一线程体系结构（见图6-6a）中，I/O线程为每一个请求派生一个新的工作线程，并且当工作线程处理完此请求时，它会销毁自己。这种体系结构的一个优点是线程不会竞争共享队列，并且吞吐量被提高到最大限度，这是因为对每一个未处理的请求，I/O线程都会产生一个线程来处理它。它的缺点在于创建和销毁线程所带来的开销很大。 235

一连接一线程体系结构（见图6-6b）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一个新的工作线程，并在连接关闭时销毁该工作线程。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（见图6-6c）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队列等待工作线程的处理，此时每一个对象有一个请求队列。

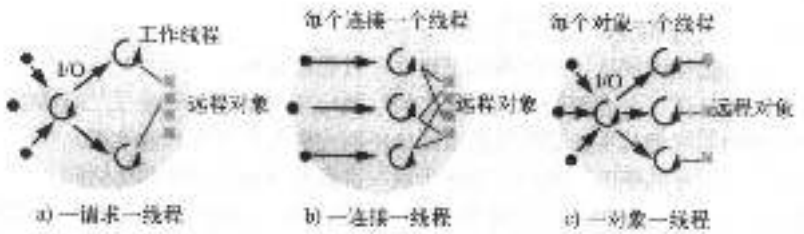


图6-6 几种服务器线程体系结构（参见图6-5）

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，服务器可以从降低的线程管理的开销中获益。这两种体系结构的缺点是当一个工作线程有多个请求等待处理时，其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衍生形式以及它们的一些混合类型，并详细讨论了它们各自的优点和缺点。6.5节将描述在单机调用环境中的线程模型，在该模型中客户线程可以访问服务器的地址空间。

客户线程 线程也可以应用于客户，就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通过远程方法调用传递到服务器，但线程不需要得到响应。然而，即使调用者不需要返回结果，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内，而第二个线程从缓冲区内取出结果。只有在所有缓冲区满了以后，第一个线程才被阻塞。

Web浏览器就是采用多线程的客户结构的例子。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多个同时到达的获取网页的请求。

线程与多进程 从上面的例子中可以了解到，线程的使用允许计算与I/O操作同时进行，在多处理器的情况下，还可以允许多个计算任务同时进行。读者可能注意到，使用多个单线程的进程也可能达到同样的并行执行的结果，那么，为什么要使用多线程进程模型呢？其原因包括两方面： 236
线程的创建和管理开销比进程少，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、像套接字这样的通信接口、像打开的文件这样的高级资源以及像信号量这样的线程同步对象。表中也列出了与之相关的线程，线程拥有一个调度优先级、一个执行状态（例如BLOCKED或RUNNABLE）、当线程阻塞时存储处理器的寄存器值和与线程的软中断处理有关的状态。一个软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。UNIX的信号便是软中断的一个例子。

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BLOCKED状态）
信号量及其他同步对象	处理消息的软件中断
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的状态

图6-7显示了执行环境和其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关。

下面我们将对进程和线程的比较总结如下：

- 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的开销小。
- 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的效率高。
- 然而，出于同样的原因，线程不能防止同一进程内其他线程的非法访问。

下面考虑在一个已有的执行环境中创建新线程的开销。创建新线程的主要工作是为进程栈分配一个区域并为处理器中的寄存器、线程执行状态（初始值可以是SUSPENDED或RUNNABLE），以及优先级提供一个初始值。因为执行环境已经存在，因此系统只需要在线程的描述符记录（其
237 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。

创建进程的开销一般远远高于创建一个新的线程的开销。创建进程时，必须首先创建一个新的执行环境，其中包括一个地址空间表。Anderson等人[1991]给出了下列数据：在运行Topaz内核的CVAX处理器体系结构上，系统花费11ms用于创建一个新的UNIX进程，而创建一个线程只用1ms。其中，时间的度量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，它会产生长期的开销，但创建新进程所产生的这一开销仍比在已有进程中创建新线程的开销多。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指令时会发生一个页失配。在初始情况下，硬件缓存不包含新进程的数据值，它必须在进程执行时获得缓存数据。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所访问时，它便自动地利用硬件或主存缓存。

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。它的开销非常重要的，因为这在线程的生命期中会经常发生。在共享同一执行环境的线程之间切换的开销要比在不同进程的线程之间切换的开销低得多。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程序计数器这样的处理器寄存器的值，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先的状态，并装载新的状态。
- 在某些情况下，转换到新的保护域，这就是所说的*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涉及域转换，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。下面会描述为实现域转换而采用硬件缓存。当域转换发生时，

人们更希望长期开销只是由访问硬件缓存和主存页面引起的。Anderson等人[1991]提供的数据说明，在同一执行环境下，在UNIX的进程间需花费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，此处没有考虑长期的缓存开销。 238

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，所以不需要在它们之间通过消息传递数据，而是通过一个公共变量来访问数据。这里存在着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，在服务器方这一优点体现得更为充分，如上面给出的缓存文件数据的例子。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，那么它们就得不到保护。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，或者改用多进程而不是多线程。

别名问题 内存管理单元通常包括一个硬件缓存，用于加速虚拟地址和物理地址间的翻译，该硬件缓存被称为翻译检索缓冲区（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指向的是不同的物理数据。仅当它们的入口被标记了上下文标识符，TLB和虚拟寻址的缓存才会知道这一点，否则缓存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。物理寻址的缓存不会遇到别名问题，但是通常采用虚拟寻址来实现缓存查找，这是因为这种查找操作可以和地址翻译同时进行。

线程编程 线程编程是一种并发编程，就像通常在操作系统中的研究一样。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。

许多线程是用C这样的常规语言编写的，这些语言一般有线程库。为Mach操作系统开发的C线程包便是一个例子。最近，POSIX线程标准IEEE 1003.1c-1995，也就是所说的pthreads，已经被广泛采用了。Boykin等人[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和Wong 1999]。下面将简单介绍一下Java线程。

像许多线程实现一样，Java提供了线程创建、销毁和同步的方法。Java 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。 239

Thread(ThreadGroup group, Runnable target, String name)
创建一个状态为*SUSPENDED*的新线程，它将属于*group*，其标识符为*name*；这一线程会执行*target*的*run*()方法。
setPriority(int newPriority), getPriority()
设置和返回线程的优先级。
run()
如果线程的目标对象有*run*（）方法，线程执行其目标对象的*run*（）方法，否则它执行自己的*run*()方法（*Thread*实现*Runnable*）。
start()
将线程的*SUSPENDED*状态转换为RUNNABLE状态。
sleep(int millisecs)
将线程转换为*SUSPENDED*状态，并持续指定的时间。
yield()
进入READY状态并调用调度程序。
destroy()
销毁线程。

图6-8 Java线程的构造函数和管理方法

thread.join(int millisecs)
调用进程阻塞指定的时间，直到thread终止为止。
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，直到调用object的notify()或notifyAll()方法唤醒线程或者线程被中断，又或者阻塞了指定的时间为止。
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

线程生命期 新线程和它的创建者在同一台Java虚拟机（JVM）上，一开始处于SUSPENDED状态。在它执行了start()方法以后处于RUNNABLE状态，这之后，它执行在其构造函数中指定的一个对象的run()方法。JVM和其上的线程都是在操作系统上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实现会在运行低优先级线程之前运行高优先级线程。当线程执行完run()方法或调用destroy()方法时，线程的生命期便结束了。

程序可以按组管理线程。在创建线程时，它可以被指定属于一个组。当有许多应用程序在同一个JVM上运行时，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。

线程组也使对线程相关优先级（在支持优先级的Java实现中）的控制更方便。这对运行applet的浏览器和运行servlet程序的Web服务器有好处[Hunter and Crawford 1998]，其中servlet能创建动态Web页面。在applet或servlet内部的无授权的线程只能生成属于自己线程组的线程，或者将新线程加入到在其内部生成的后代线程组中（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applet或servlet的线程加入到不同的组中并将这些组（包括后代线程组）作为一个整体设置一个最大的优先级。applet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()覆盖优先级。

线程同步 编程者必须很小心地编写具有多线程的进程。其中最困难的问题是共享对象和线程协调和合作所用的技术。每一个线程的方法中的局部变量是其私有的——线程有其私有栈。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，在上面的共享队列的例子中，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。线程并发处理像队列这样的数据结构时必然会引起竞争。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控器。监视器可以保证在同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。

Java允许通过任何作为条件变量的对象来阻塞或唤醒线程。需要阻塞以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 ()方法来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()方法以唤醒工作线程。

240
~
241

图6-9给出了Java的同步方法。除了上文提到的同步原语外，join()方法将阻塞其调用者，直到目的线程终结为止。interrupt()方法用于过早地唤醒等待进程。Java实现了所有标准的同步原语，

如信号量。但必须注意，因为Java的监视器保证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），当系统准备让一个线程退出运行并让其他线程运行时，此线程并不一定退出，它要运行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，每一个不包含对线程系统调用的代码区都自动地成为临界区。这样可以很方便地避免竞争。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。程序员需要在程序中加入一个yield()调用，其唯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，在实时应用中，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17章将讨论实时线程调度的需求。进程控制是实时领域中的另一个例子。一般来说，每个实时领域都有其自己的线程调度要求。因此，有时需要应用程序实现自己的调度策略。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 、Linux、Solaris、Mach和Chorus都支持多线程进程。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。多线程进程必须在一个与应用程序相联系的过程库中实现。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，由线程运行时库组织线程调度。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中断来实现线程的时间片机制。

当内核不支持多线程进程时，实现用户级的线程时会遇到下列问题：

- 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所有的线程。
- 不同进程中的线程不能按统一的优先级方案调度。

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：

- 某些线程操作的开销小。例如，在同一进程内的线程间切换不必进行系统调用，而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应用的需要。像多媒体处理系统（实时性）这样的有特殊性的应用系统对线程调度的要求各不相同。
- 能支持比内核默认支持更多的用户级线程。

可以将用户级线程实现和内核级线程实现的优点组合起来。一种已经被应用的方法是使用户级代码能为内核线程调度器提供调度提示，例如Mach内核[1990]。另外一种方法是采用层次化调度，如Solaris2操作系统采用的方式。每一个进程可以创建一个或多个内核级线程，在Solaris中叫做“轻量级进程”，它同时支持用户级线程。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。这一机制可以充分利用多处理器，同时也可以获得因为线程创建和线程切换都在用户级进行而得到好处。这一模式的缺点在于它缺少灵活性。如果一个线程在内核中被阻塞，那么所有指定在其上的用户级线程，不管其是否能够运行，都不能运行了。

为了进一步提供有效性和灵活性，一些研究项目开发出了层次化调度。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能完全由将内核级线程映射到用户级线程这种方式来满足。用户级调度器还需要内核向它通知与它调度决定相关的事件。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[1991]的“快速线程包”是一个层次化、基于事件的调度系统。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些在其上运行的应用程序。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、它们的相对优先级以及对
243 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，用于执行一个优先级相对低的任务；同时将两个虚拟处理器分配给进程B。因为内核随着时间的流逝可以为进程分配不同数量的处理器，然而在不同的时间可以将不同的物理处理器分配给该进程，所以将其称为虚拟处理器。



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

注：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，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终止了，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，进程通知内核的情况。

图6-10b也描述了四种类型的事件发生时，内核通知进程的情况。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。从一个低层（内核）进入一个上层代码区的方式被称为上调，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。这个上下文用以使进程的代码在用户级调度器指定的过程地址处得以开始运行。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READY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指定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）的四种事件：

虚拟处理器已分配：内核已经将一个新的虚拟处理器指定给进程，并且这正是其上的第一个时间片；调度器可以在READY状态线程的上下文中载入SA，线程重新开始执行。

SA被阻塞：SA在内核中被阻塞。内核准备使用一个新的SA来通知调度器，调度器将相应线
244 程的状态设置为BLOCKED，并分配一个READY线程给用于通知SA。

SA被解除阻塞：在内核中阻塞的SA被激活，可以再次在用户级执行。调度器现在可以将相应的线程插入到READY列表中。为了创建用于通知的SA，内核或者为进程分配一个新的虚拟处理器，或者它抢占同一进程上的另一个SA。在后一种情况中，它同时将抢占事件发送给调度器，调度器可以重新评估到SA的线程分配情况。

SA被抢占：内核从进程夺走一个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READY列表中，并重新评估线程分配情况。

因为进程的用户级调度器可以根据在低级事件的基础上建立的任何协议将线程分配给SA，所以层次化调度方式更具有灵活性。内核总以同一方式运行。它不会影响用户级调度器的行为，但是它通过事件通知和提供阻塞和抢占线程的寄存器状态来帮助调度。这一方式可能是有效的，因为它保证了当有一个虚拟处理器可以运行时，就不会有用户级线程仍需要处于READY状态。

6.5 通信和调用

下面我们将把通信作为调用机制的实现的一部分来进行讨论。调用的例子有远程方法调用、远过程调用和事件通知，其作用是在不同的地址空间上执行对资源的操作。

通过考虑下面的关于操作系统的问题，我们可以探讨一下操作系统的设计：

- 操作系统提供什么样的通信原语？
- 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？
- 应采取哪些步骤以使通信尽可能地有效？
- 为高延迟和断链操作提供了哪些支持？

通信原语 一些为分布式系统内核所提供的通信原语与第5章描述的调用类型是相适应的。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。用组通信更能节省系统调用上的开销。 245

实际上，是中间件而不是内核提供了现在系统中的大多数高级通信方式，包括RPC/RMI、事件通信和组通信。在用户级上开发这种复杂的软件系统的代码比在内核上开发要容易。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协议，有时也使用无连接的UDP协议。使用套接字的主要原因是考虑到可移植性和互操作性：中间件需要尽可能地在多种操作系统之上运行，并且像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以便通过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，但在一些试验性的操作系统内核上还是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，由这些协议实现在不同平台上的中间件之间的互操作，这是对操作系统主要需求之一。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，这些协议并没有被广泛应用。相反，Mach 3.0和Chorus 内核（也包括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完成。

如果需要频繁地访问（例如，每天都访问）因特网，那么（除了最小的网络设备的）操作系统需要实现在TCP和UDP层的兼容性。操作系统也要求中间件能利用新的低层协议。例如，用户希望在不升级它的应用程序的情况下利用像红外和射频(RF)传输这样的无线技术时，就需要集成相应的协议（例如红外网络使用的IrDA和RF网络使用的HomeRF或蓝牙技术）。

协议通常被安排在一个有层次的栈中（见第3章）。许多操作系统允许新的层次被静态地集成，而这是靠加入像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使协议栈能灵活合成新的层次的技术，其合成可以满足特定应用的需要，并能够利用可用的物理层。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，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-应答协议来减少往返延迟。已经证实，标准的TCP协议实现不能在无线网络介质上很好地工作[Balakrishnan et al. 1996]，
246 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，而不是用TCP层，才能使无线连接的节点有效地工作。

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，使用的远程调用也就越多。客户和服务器在其生命期内可能会执行上百万个与调用有关的操作，这样应该有一小部分时间应计入调用开销。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，最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）完全相反。在因特网上，网络延迟通常变化很大，平均值也很高，带宽通常很低，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，因为通用的客户－服务器处理广泛采用了这种机制。许多研究已经涉及在网络上的调用，并且特别研究了如何更好地利用高性能网络来实现调用机制[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，也有一些研究注重于在同一计算机不同进程之间的RPC[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，进行远程过程调用和远程方法调用，这些都是调用机制的例子。每一种调用机制都导致在调用过程和对象之外的代码被执行。一般每次调用都涉及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（例如传统调用和远过程调用），也可以是异步的。

除了是否异步，对调用机制的性能影响最大的因素为是否涉及域转换（也就是说，它是否跨越了一个地址空间）、它是否涉及网络上通信，以及它是否涉及线程调度和切换。图6-11表示了一个系统调用、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节点的计算机进程之间的远程调用三个例子。

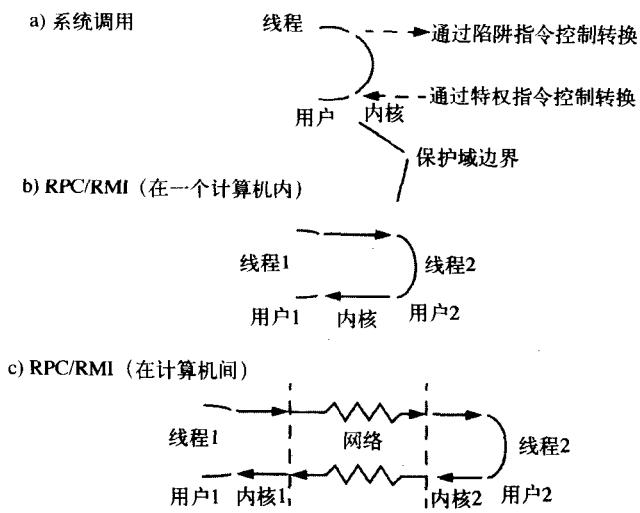


图6-11 在地址空间之间的调用

在网络上的调用 一个空的RPC（类似的，一个空的RMI）是一个没有参数、执行一个空的过程，也不返回值的RPC。它负责交换不包括任何用户数据、只包含很少系统数据的消息。现在，通过100Mbps的LAN在两台500MHz的PC机上的用户进程间进行一次空的RPC，其时间开销在几十毫秒这个数量级内，而一个传统的空过程调用只需小于1微秒的时间。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——是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。

空调用（RPC、RMI）的开销是非常重要的，因为它度量了一个固定的开销，也就是延迟。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，空调用的延迟比其他类型的延迟要大得多。

假设一个RPC从服务器上获得指定数量的数据。它包含一个整数型请求参数，用于指明所需的数据量的大小。在返回结果时，它包括两个返回参数，一个整数型参数表示调用是成功还是失败（客户可能提供的是一个非法的数据量大小）。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。 247 ~ 248

图6-12表示了所需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，系统至少要多传送一个额外的包。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。每一次需要多传送一个包时，图上便会出现一个跳跃。

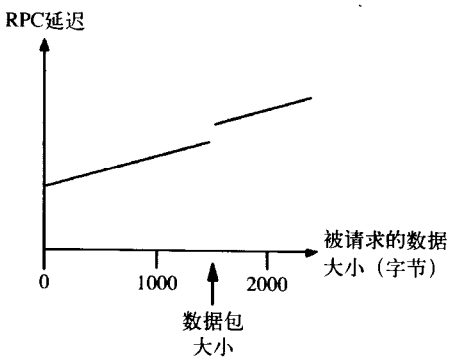


图6-12 RPC延迟与参数量大小的关系

在RPC的实现中，延迟并不是被关注的唯一数据：当数据成批传输时，RPC带宽（或吞吐量）也会受到关注。它表示在一个RPC内的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，带宽会增加，这是因为那些固定处理的开销变得相对比较小。Gokhale和Schmidt[1996]引用了一个例子：在具有155Mbps带宽的ATM网络上，当在工作站之间的RPC上传输64KB数据时，吞吐量大约为80Mbps，即花费0.8ms来传输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，并将请求消息发送出去，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。不论在哪种情况下，都要调用合适的服务器存根程序。
- 服务器存根程序将请求消息解码，调用指定的过程并将返回值编码并发送出去。

下面是除网络传输时间之外造成远程调用延迟的主要因素： 249

编码：编码和解码涉及拷贝和转换数据，当数据量增加时，它们会成为一个重要的时间开销。

数据拷贝：即使在编码之后，在一个RPC过程中，消息数据也可能会被多次拷贝：

1）跨越用户－内核边界，在客户或服务器地址空间和内核缓冲区之间。

2）在每个协议层之间（例如，RPC/UDP/IP/以太网）。

3）在网络接口和内核缓冲区之间。

网络接口和主存之间的传输通常是由直接内存访问（DMA）来处理的。其他拷贝则是由处理

器处理的。

包初始化：它涉及初始化协议头部和协议尾部，包括校验和。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：它包括如下部分：

1）当存根程序调用内核的通信操作时，在一个RPC过程中会产生几个系统调用（即上下文切换）。

2）调度一个或多个服务器线程。

3）如果操作系统采用单独的网络管理器进程，那么每一次Send操作会涉及线程之间的上下文切换。

确认等待：RPC协议的选择会影响延迟，特别是当有大量数据需要传输的时候。

小心设计操作系统有助于减少这些开销。在www.cdk3.net/oss上可以找到Firefly RP的实例研究，其中还包括在中间件实现中可用的技术。我们已经介绍过了一些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（见6.4节）可以用于用户进程和内核之间或者在用户进程之间的快速通信。通过在共享区域中写数据和读数据可以实现数据通信。这样可以实现高效数据传输，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它。当然，当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。

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]允许用户级的代码直接访问网络接口，因此使用户级的代码可以不经任何拷贝就能把数据传输到网络。

250 **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，有些时候甚至要短一些，特别是传输大量消息的时候。然而，在像TCP这样的协议之上实现请求－应答交互必须要小心，因为这些协议并不是专为此目的而设计的。特别是TCP的缓冲机制会妨害其性能，它的连接开销与UDP相比也处于劣势，除非在一个连接上需要传输的数据量相当大，这样才可以忽略连接单个请求的开销。

在Web调用中，TCP连接上的开销特别明显，这是因为HTTP 1.0为每一个调用建立一个独立的TCP连接。当建立连接时，客户的浏览器被延迟。而且，TCP的慢启动算法延迟了HTTP数据传输，而很多情况下这是不必要的。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，首先只向网络传输一个小窗口的数据。Nielson等[1997]讨论了HTTP 1.1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，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，修改操作系统的默认缓冲行为对调用延迟有显著的影响。比较好的机制是收集多个比较小的信息，然后将它们一起发送，而不是分别用独立的包将其发送出去，因为每一个包都会产生上面所描述的延迟。由于上面的原因，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。操作系统在默认情况下应该等待缓冲区满或者超时之后才将数据分发到网络上。

Nielson等发现，在HTTP1.1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。为了避免这种延迟，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想象的那样发生在计算机之间，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更多的调用是针对本地进程的。特别是在客户所需要的数据可能在本地服务器上的情况。将一个计算机内的RPC开销作为系统性能的一个参数已经变得越来越重要。这些都表明，计算机内的调用应该被优化。

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调用几乎完全一样。实际上，这是一种经常实现的模型。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的设计基于数据拷贝和线程调度的优化。

首先，他们提出利用共享内存区域为客户－服务器提供有效的通信，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（见图6-13）。在这种设计中，客户和服务器可以通过A栈传递参数和返回结果，而不必涉及在内核和用户地址空间之间的RPC参数的拷贝。客户和服务器存根程序也采用同样的栈。在LRPC中，参数只拷贝一次，即当它被编码后进入A栈时。而在一个等价的RPC中，数据被拷贝四次，分别为从客户存根程序的栈中拷贝到消息中；从消息拷贝到内核缓冲区；从内核缓冲区拷贝到服务器消息中；从服务器消息拷贝到服务器存根程序的栈中。在共享区域中可能有数个A栈，因为在同一时间同一客户上会有多个线程调用服务器。

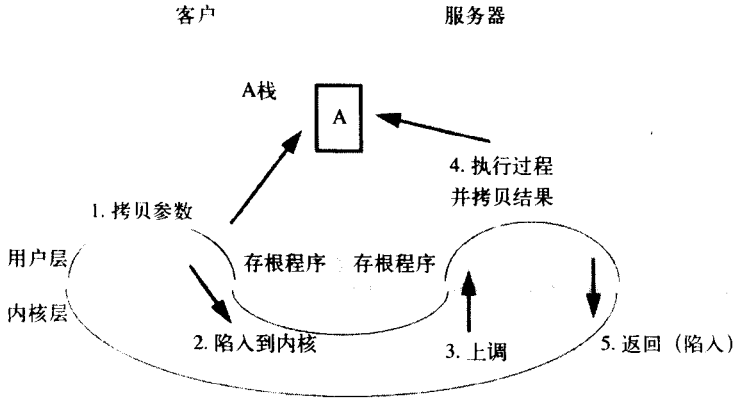


图6-13 一个轻量级远程过程调用

Bershad等也考虑到线程调度的开销。通过比较图6-11中的系统调用和远程过程调用，可以发现：当发生一个系统调用时，大多数内核并不调度一个新的线程来处理调用，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，服务器程序与以前描述过的服务器有所不同。服务器输出一系列过程以备调用，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，它们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表示一个调用。客户端线程通过首先陷入内核来进入服务器的执行环境。内核检查其合法性并只允许上下文切换到合法的服务器过程上。如果它是合法的，内核便将线程的上下文转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，线程便回到内

核，内核将线程转换回客户执行环境。需要注意的是：客户和服务器采用存根程序来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的效率更高这一点是无可置疑的。Bershad等统计得到LRPC的延迟比本地RPC的延迟小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，然后相应地选择采用LRPC或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，则迁移透明性将很难实现，这是因为需要改变调用机制。

在随后的工作中，Bershad等[1991]描述了几种改进性能的方法，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和在调度进程时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，那么线程应该被转移到此处理器上。这种方式避免了域转换，同时，客户的处理器可以被客户的其他线程重用。改进还包括两层（用户和内核）线程调度的实现（见6.4节）。

6.5.2 异步操作

我们已经讨论了操作系统如何帮助中间件层来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。我们还可以算上网络的断链和重新连接的开销，网络的断链和重新连接被认为是造成高延迟通信的原因。用户的移动计算机并不是一直都连接在网络上的。即使使用广域无线访问技术（例如，使用GSM），它们也可能随时断链，例如，当他们乘坐的火车进入了隧道。

异步操作是应付高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中。但当我们讨论调用性能时，还是应该考虑到异步操作的作用。

使调用并发执行 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
253 发执行阻塞型调用。

Web浏览器是这种应用一个很好的例子。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HTTP 1.0 Web浏览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发出4个并发请求。在这种方式下，获得所有图像的时间通常比用串行请求所花的时间少。通常情况下，不仅总通信延迟会减少，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在单处理器上的服务器间交错调用（如HTTP请求）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。服务器执行Receive操作，进行解码并处理结果。在此之后，客户才能执行第二个调用。

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它的调用结果。如图所示，并发调用的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发生并发请求时也能减少总调用时间。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处理可以并行进行。

回到HTTP的例子，前面所介绍的Nielson等[1997]也研究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，只要操作系统为刷新缓冲区提供合适的接口，管道可以减少网络流量并能提高客户性能。

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，调用者进行的是非阻塞调用，只要创建了调用请求信息并准备发送，调用便结束了。

有些时候，客户不需要任何回复（除了需要故障信息，如目标主机连接不上等），例如，

CORBA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做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里。调用者使用claim操作从promise中获得结果。claim操作一直被阻塞到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——它根据promise的状态为就绪或阻塞状态分别返回true 或false。

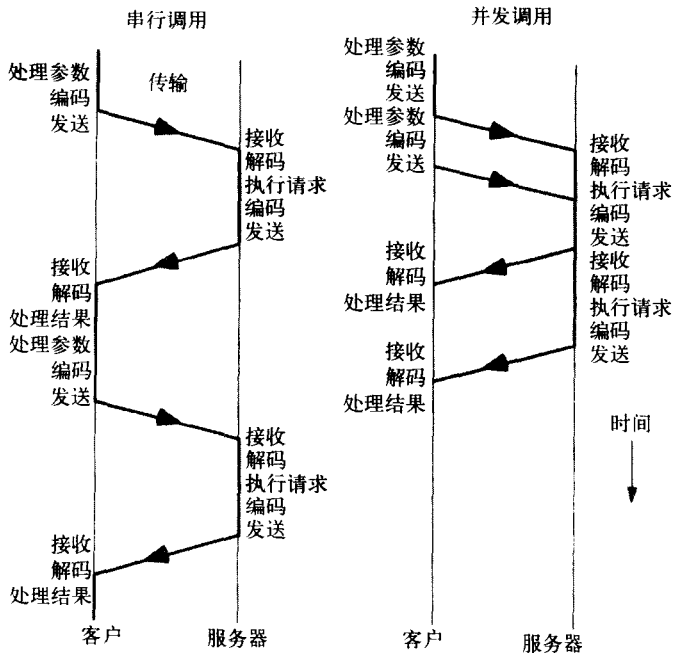


图6-14 串行调用和并发调用的时间

持久异步调用 像Mercury调用和CORBA单向调用这种传统异步调用机制是在TCP流上实现的。当TCP连接中断时，例如网络连接中断或目标主机崩溃，调用就会失败。

断链操作使得称为持久异步调用的改进异步调用模型变得更加有意义。就所提供的操作而言，该模型与Mercury相似，它们的不同在错误语义上。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就会失败。但是这些短期的超时经常不能适应连接中断或长延迟的情况。

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，或者应用程序取消调用为止。其中一个例子是用于移动信息访问的Rover 工具集中的QRPC（排队的RPC）[Joseph et al. 1997]。

顾名思义，当没有网络连接时，QRPC将调用请求排队在固定的日志中，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为止。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。

254 ~ 255

QRPC可以利用不同的通信链路发送调用请求和接收应答。例如，当用户在路上时，其调用请求可以通过GSM发送，但当用户将他的设备连接在企业内部因特网上时，其调用应答可能通过以太网发送。原则上，调用系统可以在靠近用户的下一个可能的接入点存储调用结果。

客户网络调度器可以依照不同的标准操作，而没有必要依照FIFO的顺序来发送调用。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评估其带宽和使用它的开销。它首先

发送高优先级的调用请求，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——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ORPC从低带宽连接的邮箱上获取调用结果时，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：在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，用户可能想知道是否成功地更新了共享文档中的一个段落，还是另一个用户同时进行了一次有冲突的更新，如删除了这一段落。第15章将讨论这一问题。

6.6 操作系统的体系结构

本节将讨论适用于分布式系统的内核结构。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要求：

- 在每台计算机上仅运行那些在系统体系结构中承担特定角色的系统软件，对系统软件的需求可能会不尽相同，例如个人数字助理和专门的服务器计算机对系统软件的需求就会不同。而载入多余的模块会浪费内存资源。
- 允许实现特定服务的软件（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不同实现。
- 在不破坏已有系统的一致性的情况下加入新的服务。

从资源管理策略中分离固定资源管理机制的方法（它随着应用程序和服务的不同而不同）已
256 经在很长一段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：系统既要使用一个像视频会议这样的多媒体应用程序来满足实时需求，也要支持像Web浏览这样的非实时应用程序。

理想情况下，内核应该只提供在一个节点上实现通用资源管理任务的最基本机制。服务器模块应按需动态装载，以便为当前运行的应用实现所需的资源管理。

整体内核和微内核 内核设计有两个主要例子：整体内核和微内核。这两种设计之间主要的区别在于：如何决定哪些功能属于内核和哪些功能属于服务器进程，其中服务器进程可以在运行时动态载入这些功能。尽管微内核没有被广泛应用，但理解它们与当前一般内核相比的优点和缺点仍然是有益的。

UNIX操作系统内核被称为整体内核（见下面的定义）。这一名称说明了其内核的巨大：它具有所有的基本操作系统功能，其代码和数据量达到上兆字节，并且它是未分化的，即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，因为为变化的需求改变单个软件组件将会很困难。Sprite网络操作系统是另一个整体内核的例子[Ousterhout et al. 1988]。一个整体内核可以包含在其地址空间内执行的若干服务进程，其中包括文件服务和一些网络进程。这些进程执行的代码是标准内核配置的一部分（见图6-15）。



图6-15 整体内核和微内核

整体 Chambers 20th Century Dictionary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，因此而难于管理。

相反，在微内核的设计中，内核只提供最基本的抽象，主要为地址空间，线程和本地进程间通信。所有其他系统服务由服务器提供。这些服务在分布式系统需要它们的时候才动态加载到计 257
算机上（参见图6-15）。客户使用内核提供的基于消息的调用机制来访问这些系统服务。

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：像UNIX这样的标准操作系统的二进制模拟[Armand et al. 1989, Golub et al. 1990, Härtig et al. 1997]。

图6-16给出了微内核（以最通用的形式）在整个分布式系统中的位置。其中，微内核为在硬件层与包含主要系统组件的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，它使用语言运行时支持子系统或由操作系统模拟子系统提供的高层操作系统接口。这些都是由可链接在应用程序上的库过程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（多个“操作系统”）。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。需要注意的是，操作系统模拟不同于机器虚拟化（稍后评述）。

比较 基于微内核的操作系统的主要好处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，一个相对小的内核的缺陷可能比大而复杂的内核少。

整体内核设计在操作调用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，甚至在使用了我们前面介绍过的技术时也是如此。在同一节点上的一个用户级地址空间上调用的开销仍然比较大。

通过使用像分层（在MULTICS[Organick1972]中使用）或像在Choices[Campbell等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采用了以 258
上两种方法的组合[Custer 1998]。但是Windows仍然是“巨大”的，并且其大多数功能没有被设计为可替换的。模块化的大内核也很难维护，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，并且该模块是用C或C++语言编译为高效代码，并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，这样在一个模块的缺陷可能会破坏另一个模块的数据。

虚拟化 虚拟化是指将多个虚拟机（虚拟硬件映像）分配给同一个机器硬件，每一个虚拟机运行独立的操作系统实例。虚拟化最早出现在IBM 370体系结构中，它的VM操作系统可以将同一台电脑的硬件虚拟为若干完整的虚拟机，而且不同的程序可以运行在不同的虚拟机上，而实际上这些程序运行在同一台电脑硬件上。最近，一些项目（例如，Xen[Barham et al.

2003]）为普通的PC 机开发出了虚拟机监控器。虚拟机监控器是一个位于机器硬件同常用的操作系统（如Linux 或Windows ）之间的软件层。它允许单台PC 机同时运行多个任意的操作系统，而且这些操作系统不需要任何改变，虚拟机监控器能够将这些操作系统隔离，并且保护它们以免互相干扰。PC机和集群中的节点已经有了商业的实现，例如VMWare ，这些实现可以更灵活地完成分配给它们的工作。

一些混合的方法 两种最初的微内核Mach[Acetta et al. 1986]和Chorus[Rozier et al. 1990]在其开发周期中将运行的服务器仅仅作为用户进程。它们通过硬件支持的地址空间来增加模块性。在服务器需要直接访问硬件的地方，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，这样使用户级服务器可以处理中断。

因为性能问题，Chorus和Mach微内核设计最终允许服务器可以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，客户可以用相同的进程间通信调用与服务器交互。因此开发者可以在用户级调试服务器，同时，在开发完成时，为了优化系统性能，系统允许服务器在内核地址空间内运行。但这样的服务器程序会影响系统的一致性，因为它可能包含某种错误。

SPIN操作系统[Bershad et al. 1995]设计采用语言保护机制来取得效率和安全性的折中。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。但是它们都是用类型安全的语言[Modula-3]编写的，所以它们相互保护。内核地址空间内的保护域是使用被保护的名字空间来建立的。除非具有对它的引用，否则进入内核的模块是不能访问这一资源的。同时，Modula-3限制了
259 引用只能用来执行程序员允许的操作。

为了尽可能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内模块的交互机制（见5.4节对基于事件的编程的讨论）。系统定义了一系列核心事件，如网络包到达、定时器中断、出现页失配和线程状态改变等。系统组件将自己注册为这些事件的处理程序。例如，一个调度器可以将自己注册为一个处理程序，用于处理与那些我们在6.4节讨论过的调度器激活相似的事件。

通过像Nemesis[Leslie et al. 1996]这样的操作系统，可以发现这样一个事实：即使在硬件级，一个地址空间也不一定必须是一个保护域。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个地址空间内。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运行时可用的空间内。64位地址的处理器的出现使单地址空间的操作系统变得更加吸引人，这是因为它们可以支持很大的地址空间，其中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一个区域上设置保护属性来限制用户级代码的访问。用户级代码仍然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），它给了代码访问本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，单地址空间设计节省了开销，因为当域转换时，内核不需要刷新任何缓存。

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来减少上述策略带来的开销。通过将地址空间的管理委托给用户级服务器，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。它提供了像磁盘块这样极低级资源的保护性分配，并且它希望其他资源管理功能——甚至是文件系统——都作为库连接到应用程序上。

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和绝境，但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是如何通过提供在共享资源上的调用来支持中间件层的。操作系统提供了若干机制，用于实现满足本地需求及技术改进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上的操作的必要机制。 260

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；线程是在执行环境中执行的活动抽象。地址空间必须比较大并且是稀疏的，以便支持对文件这样的对象的共享和映射访问。新创建的地址空间可能继承了其父进程的区域。写时复制是一个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。多线程进程允许我们利用多处理器并行的优势，以较低的代价实现并发。这对客户和服务器都很有用。最近的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。

操作系统为经由共享内存进行的通信提供了基本的消息传递原语和机制。大多数内核都包含一个实现网络通信的基本设施；其他内核只提供本地通信并将网络通信功能交给服务器完成，它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。我们发现，对于一个空调用而言，花费在软件上的时间相对比较大，但当调用参数的字节数增大时其时间占总时间的比例会减小。调用中要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例，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。微内核至少要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像实时处理这样的子系统。

练习

6.1 在UNIX文件服务的例子中（或其他你熟悉的例子）讨论封装、并发处理、保护、名字解析、参数和返回结果的通信以及调度。（第224页） 261

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基于消息的系统调用？（第224页）

6.3 史密斯认为在他的进程中，每个线程都应拥有其自己的保护栈，而线程的其他区域必须被完全共享。这样做有意义吗？（第228页）

6.4 信号（软件中断）处理器应属于进程还是线程？（第228页）

6.5 讨论共享内存区域的命名问题。（第230页）

6.6 假设要设计一个平衡各计算机负载的方案，你必须考虑如下问题：

1）这一方案能满足用户或系统的哪些需求？

2）它能适应哪一种类型的应用程序？

3）怎样度量和以何种精确程度度量负载？

4）假设进程不能迁移，怎样监控负载和为新的进程选择地点？

如果进程能在计算机之间迁移，你的设计将受到哪些影响？进程迁移的开销很大吗？（第231页）

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？（第233页）

6.8 一个文件服务器使用缓存，并且其命中率为80%。当服务器在缓存中查找被请求的块时，服

务器中的文件操作要花费5ms的CPU时间，否则它还要再花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/秒）：

（1）单线程。

（2）在一个处理器上运行的两个线程。

（3）在两个处理器计算机上运行的两个线程。 （第234页）

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。 （第235页）

6.10 什么样的线程操作开销最大？ （第237页）

6.11 spin锁（见Bacon[1998]）是一个用原子性的测试－设置指令访问的布尔变量，它用于实现互斥。你能使用spin锁在单进程的计算机上实现线程间的互斥吗？ （第241页）

6.12 解释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。 （第242页）

6.13 页失配是用户级线程实现中的问题吗？ （第242页）

6.14 解释在“调度器激活”设计中使用混合调度方法（而不是纯粹的用户级或内核级调度）的原因。 （第243页）

262

6.15 为什么线程包会对线程的阻塞或解除阻塞事件感兴趣？为什么会对即将被抢占的虚拟处理器感兴趣（提示：可以继续分配其他虚拟处理器）？ （第244页）

6.16 网络传输时间占一个空RPC的总耗时的20%，而它占一个传输1024用户字节（小于一个网络包的大小）的RPC的总耗时的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善百分之多少？ （第247页）

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。请解释导致延迟的原因。

在同一个RMI系统中，每1K的用户数据会增加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？ （第247页）

6.18 影响远程调用的哪些因素会影响消息传递？ （第249页）

6.19 请解释共享区域是如何应用于进程读取内核写的数据的。你的解释应包括实现同步的必要机制。 （第250页）

6.20 （1）轻量级过程调用的服务器能控制其中的并发度吗？

（2）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。

（3）LRPC是不是比传统的RPC（假设是共享内存的）具有交互干扰的风险更大？ （第251页）

6.21 一个客户对一个服务器进行RMI调用。客户需要5ms对每一个请求进行参数计算，并且服务器要花费10ms处理每一个请求。每一个send和receive操作的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，请估计客户产生两个请求并返回结果的时间：（1）单线程；（2）在单个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？ （第253页）

6.22 请解释什么是安全性策略，在像UNIX这样的多用户操作系统中，相对应的是什么机制？ （第256页）

6.23 请解释当服务器动态载入内核地址空间内时，程序必须要满足的的连接要求，并说明这种情形与在用户级执行服务器的区别。 （第257页）

263

6.24 中断是怎样与用户级服务器通信的？ （第259页）

6.25 在某个计算机上，我们预计：不管其运行哪种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操作系统的情况下，请估计客户调用动态载入的空过程的开销。 （第259页）

264

第7章 安 全 性

在分布式系统中，资源的私密性、完整性以及可用性都需要有相应的措施加以保证。安全性攻击会采取窃听、伪装、篡改和拒绝服务等形式。安全的分布式系统的设计者们必须在攻击者可能了解系统所使用的算法和部署计算资源的环境下解决暴露的服务接口和不安全的网络所引发的问题。

密码学为保证消息的私密性和完整性以及消息认证奠定了基础。为使密码学得以应用，需要有精心设计的安全性协议。加密算法的选择和密钥的管理是安全机制的效率、性能和可用性的关键。公钥加密算法使得分发密钥比较容易，但对大数据量数据的加密而言其性能不够理想。相比之下，密钥加密算法更适合大批的加密任务。混合型协议，例如TLS（传输层安全）用公钥加密先建立一个安全通道，然后使用通道交换密钥，并将此密钥用于后续的数据交换。

可为数字信息签名，生成数字证书。通过数字证书，可以使用户和组织建立起互相信任。 265

7.1 简介

在2.3.3节中，我们曾经给出过一个简单的模型用于解释分布式系统对于安全性的需求。我们总结出，分布式系统对安全机制的需求源自共享资源的需求。（对于不需要共享的资源，通常需要将它们同外部访问隔离开。）如果我们将共享资源也看作对象，那么任何封装了共享对象的进程都要受到保护，而且它们之间进行交互的通信信道也应当受到保护，以避免可预料的任何形式的攻击。2.3.3节中介绍的模型有利于理解安全需求，总结如下：

- 进程封装了资源（包括程序语言层的对象和系统定义的资源），并且允许客户通过接口访问这些资源。已授权的主体（用户或进程）可以操作这些资源，而资源必须被保护以避免未授权的访问（见图2-13）。
- 进程通过多用户共享的网络进行交互。敌人（攻击者）也可以访问这个网络，它们能够复制或者尝试读取任何在网络中传输的消息，也可以向网络中插入任何的消息，这些欺骗性的消息可以被发送到网络上的任何地方，并且谎称它们来自于其他地方（见图2-14）。

无论是在数字世界里还是在物理世界里，无论是个人还是组织，对信息和资源的私密性和完整性的需求是广泛存在的。这种需求源于对共享资源的期望。在物理世界中，组织采用*安全策略*，在指定范围内允许资源的共享。例如，某公司只允许公司的职员以及受信赖的访问者进入办公大楼；而文档的安全策略可以规定某些工作组的成员可以访问指定类的文件。也可以针对单个文件和用户制定安全策略。

安全策略通过安全机制执行。例如，某人是否允许进入办公大楼可以由接待员来决定，他给受信赖的访问者派发通行证，再由保安人员或者电子门锁来验证谁能进入大楼。对纸质文档的访问，通常可以采用加密和限制性地发送等手段来控制。在电子世界中，安全策略和安全机制的区别同样重要；没有它，就很难判断一个系统是否安全。安全策略和所使用的技术无关，就像在门上装锁，这并不能确保办公大楼的安全，除非为它的使用制订一些策略（例如，当没有人在入口处守卫的时候，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，我们将概述各种简单的电子商务场景中的安全需求，并说明每个环境中需要的安全策略。

本章的重点是分布式系统中数据和其他资源的保护机制，这种保护机制允许计算机在安全策略许可的范围内进行交互。这些机制用来实施安全策略以应付大部分已确定的攻击。 266

密码学的任务 数字密码学为大多数计算机安全机制奠定了基础，但是注意下面这一点很重要，即计算机安全同密码学是两个不同的主题。密码学是信息编码的艺术，它通过一些特定的格式，仅允许特定的接收者访问。类似于传统交易中的签名，密码学也可以用来验证信息的真实性。

密码学有一段悠久的耐人寻味的历史。军方对安全通信的需求以及截获并解密敌人信息的需求，使得当时一些杰出的数学家为此花费了大量的精力。读者如果对这段历史感兴趣，可以参阅由David Kahn[Kahn 1967,1983,1991]和Simon Singh[Singh 1999]所编写的著作。Whitfield Diffie，公钥加密的发明人之一，用第一手的信息记录了近代密码学的历史和加密策略[Diffie 1988, Diffie and Landau 1998]。

以前，政治军事组织控制密码学的发展和使用。直到最近，密码学才被真正地解放。现在，它成为一个大型、活跃的研究团体中一个开放的研究课题。研究结果也相继在各种书籍、杂志和会议上发表。Schneier的《应用密码学》[1996]的出版成为开放该领域知识的一个里程碑。这是第一本包括了很多重要算法且附带源码的著作，这也是勇敢的一步。在现代密码学的大多数领域，Schneier的书具有相当的权威性。Schneier与他人合著的新书[Ferguson and Schneier 2003]中对计算机密码学进行了精彩的介绍，其中讨论了目前使用的所有的重要的算法和技术，其中有些算法和技术在Schneier早年的著作中有所涉及。此外，Menezes等人[1997]也出版了一本有很强理论基础的实用性手册Network Security Library（网络安全库）[www.secinf.net]是有关实践知识和经验的一个很出色的在线资源。

Ross Anderson的“Security Engineering”[Anderson 2001]也是一本很出色的书，其中有丰富的从现实世界的情况和系统安全故障中得出的系统安全设计的实例。

密码学的非军事应用和分布式计算机系统对安全性的需求的巨大增长在很大程度上推进了密码学的开放性。于是，在军事领域以外的第一个自主的密码学研究团体应运而生。

密码学对公众开放并允许公众使用以来，密码技术得到了突飞猛进的发展，不仅体现在对抗敌人的攻击能力方面，而且体现在密码技术部署的方便性上。公钥密码学就是密码技术开放以后获得的成果之一。再举一个例子，DES标准加密算法最初是一个军事秘密，只能由美国军方和政府部门应用，但当它最终公布并被成功地破解之后，反而促进了密码学的发展，产生出更多更强有力的密钥加密算法。

另一个进步是使常见术语和方法得到发展。例如，为一个受保护的事务中的角色（主体）选取一组熟悉的名字。为主体和攻击者都起一个熟悉的名字，有利于阐明安全协议和对这些安全协议的潜在攻击，这是识别其弱点的重要一步。图7-1中显示的名字在安全文献中广为使用，我们在书中也将使用这些名字。我们还不知道这些名字的由来。据我们所知，它们最初出现于RSA公钥算法的论文[Rivest et al. 1978]中。对于它们的使用的注释，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	一个服务器

图7-1 为安全协议中的角色起的名字

7.1.1 威胁和攻击

有一些威胁是很明显的。例如，在大多数的本地网络中，可以很容易地在互连的计算机上构造并运行一个程序，用于获得在其他计算机间传递的消息的拷贝。另一些威胁则较为隐蔽，例如，当客户不能认证服务器时，这个程序就安装自己，并取代真实的文件服务器，从而获得客户发送

该书已由机械工业出版社华章分社引进并翻译出版，ISBN: 7-111-07588-9/TP.1216。——编辑注

的机密信息。

除了直接破坏而导致信息和资源的丢失或损坏外，攻击者还可能向系统拥有者做出系统是不安全的欺骗性声明。为了避免这些欺骗性声明，拥有者必须证明系统在受到攻击后仍然是安全的，或者为这个可疑时期中的每个事务都产生一个日志文件来反驳这些声明。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，而第三方无法伪造出这个签名。

安全的主要目的是只允许获得授权的主体访问信息和资源。安全威胁一般可以分为三大类：

泄漏——未经授权的接收方获得了信息。

篡改——未经授权对信息进行改动。

恶意破坏——干扰系统的正确操作，对破坏者本身无益。

267 ~ 268

对分布式系统的攻击依赖于对现有通信通道的访问或者伪装成授权的连接来建立新的通道。（我们用术语“通道”来指代任何进程间的通信机制。）可以按照恶意使用通道的方式，对攻击方法进行进一步的分类：

窃听——未经授权获得消息副本。

伪装——在未经授权的情况下，用其他主体的身份收发消息。

消息篡改——在将消息传递给接收者之前，截获并修改消息的内容。*中间人攻击*就是一种消息篡改方式，其中攻击者截获了密钥交换的第一个消息来建立安全通道。攻击者替换掉发送方与接收方达成的密钥，以便让自己可以对后继消息进行解密，然后再将消息用正确的密钥加密后，传递出去。

重发——存储截获的信息，并稍后发送它们。这种攻击甚至对已认证的消息和加密消息都可能有效。

拒绝服务——用大量的消息使通道或者其他资源瘫痪，使得其他访问被拒绝。

这些都是理论上存在的威胁，但实际上这些攻击是怎样实现的呢？成功的攻击取决于发现系统安全方面的漏洞。遗憾的是，在现有系统中普遍存在安全漏洞，有的甚至很明显。Cheswick和Bellovin[1994]指出了42种缺陷，他们认为这些在广泛使用的因特网系统及其组件中的存在的缺陷会带来很大的风险。这些弱点包括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的攻击。其中有些已经成为人们所熟知的攻击入口点[Stoll 1989，Spafford 1989]，攻击者会通过这些入口点进行恶作剧或者网络犯罪。

起初设计因特网和与其相连的系统时，安全性没有被充分考虑。设计者可能没有想到因特网会发展成如此规模，而且像UNIX之类的系统的基本设计也先于计算机网络出现。我们可以看到，安全手段需要在基本设计阶段就被仔细的考虑。本章的内容就是为此提供一些基础。

我们已经注意到因为暴露通信通道和接口而对分布式系统产生的种种威胁。对许多系统而言，只需要考虑这些威胁（人为错误引起的威胁不在考虑之列，因为安全机制并不能防止用户使用非常简单易猜的口令或者用户粗心泄漏口令而造成的威胁）。对于包含移动程序的系统和其安全性对信息泄漏特别敏感的系统，还存在其他威胁。

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中，并在本地执行。在这种情况下，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。

269

Java是这种类型的语言中使用最广泛的。为了限制这种暴露，设计者也仔细考虑过语言的设计和构造，以及远程下载机制（沙盒模型就是用于对付移动代码的）。

Java虚拟机（JVM）在设计的时候就考虑到了移动代码。它给每个应用分配各自独立的运行的环境。每个环境都有一个安全管理器，用于决定哪些资源对于该应用来说是可用的。例如，安全管理器会终止应用的读写文件操作或限制程序对网络的访问。一旦设置了网络管理器，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，确实不能保证这些移

动代码会可靠地执行。实际上，会存在下载并运行恶意代码的风险，这些恶意代码会删除文件或访问私人信息。为了保护用户免受这些不可信代码的攻击，大部分浏览器都限定applets不能访问本地文件、打印机和网络套接字。一些使用移动代码的应用能在下载的代码中设置多种信任级别。这样，安全管理器就允许移动代码访问更多的本地资源。

为保护本地环境，JVM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，以保证程序执行的时候不会发生某些错误，如访问非法的内存地址。

当人们逐渐意识到最初采用的安全机制不能避免漏洞的时候[McGraw and Felden 1999]，Java的安全性就成为许多后续研究的主题。这使得被发现的漏洞得到修补，Java保护系统也进行了修正，允许移动代码在获得授权时访问本地资源[java.sun.com V]。

尽管包括了类型检查和代码验证机制，整合到移动代码系统中的安全机制仍然达不到用于保护通信通道和接口的安全机制所能达到的安全级别。这时因为执行程序的环境为错误的发生提供了很多机会，而且也不能肯定所有的错误都能被避免。Volpano和Smith[1999]已经指出一个相对较好的解决办法，该方法基于移动代码的行为是完备的证明。

信息泄漏 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，如果某只股票有大量交易消息表明这支股票有较高的交易率。还有许多微妙的信息泄漏形式，有些是恶意的而有些则源于疏忽。一旦观测到计算结果，则潜在的泄漏危险就会增加。在20世纪70年
270 代，人们就开始了防止这类安全威胁的工作[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]率先描述了信息流的安全控制方法。最近一些研究主题是用组件之间的互不信任关系将这种方法扩展到分布式系统[Myers and Liskor 1997]。

7.1.2 保护电子事务

因特网在工业、商业和其他领域的许多应用中都包括一些对安全性要求较高的事务。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密（例如，当发送一个信用卡号的时候）或者内容和消息的发出者必须经过认证（如用电子邮件提交一个拍卖的竞价）。本章所述的密码安全技术现在已经应用到许多邮件客户中了。

购物和服务：这样的事务现在已经非常常见了。购买者在Web上选定商品并付账，所购的商品会通过相应的配送机制送到购买者的手中。软件或者其他的数字产品（如唱片和录像）可以通过从因特网上下载来交付给购买者。其他有形的商品，如图书、CD和其他各种商品也可以从因特网供应商处购买，商品通过配送服务交付到购买者手中。

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。用户可以检查余额状态、转账、定期缴纳各种款项等。

微事务：因特网参与向大量用户提供少量的信息和其他服务。例如，大部分的Web页面还没有收费，但作为一个高质量的发布媒介的网页的发展取决于信息提供者从信息的消费者处获得的费用。例如，因特网上音频和视频会议的使用，就是一种有偿服务。这些服务的价格或许不到一分钱，支付开销必须相应较低。通常来说，让每一个事务包括一个银行或信用卡服务器的方案，不能满足降低开销的要求。

要想安全地执行这样的事务，必须有相应的安全政策和安全机制。要避免在消息传输过程中
271 泄漏购物者的信用卡号码（卡号），以及防止那些无诚信的供货商在收到付款后却不发货。供货商必须在发货前收到付款，对于下载的产品，他们还必须确保只有顾客得到了可用的数据。保护所需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敏感的安全政策，会产生下列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自己准备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，保证这些资料不加改变地在购买者和供货商之间传输。

3）如果商品是可以下载的，那么要保证它们的内容不加改变地传递给了购买者，而且不会泄漏给第三方。

通常供货商并不需要对购买者的身份进行认证（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的时候完成的。

把购买者比作银行账户持有者，供货商比作银行，那么使用开放网络的银行事务的安全需要与购买事务类似，但显然还有下列需要：

4）在给予银行账户的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，银行必须保证参与了事务的账户持有者不能抵赖，这一点非常重要。这种安全需求称为*不可抵赖*。

除了上述由安全政策规定的需求外，还有一些系统需求。这些需求源于因特网巨大的规模过于庞大，所以购买者和供货商难于达成某种特定的关系（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从未与供货商联系过的情况下，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，并存储在客户主机上）有明显的安全缺陷，因为台式和移动主机都经常处于不安全的物理环境中。

考虑到因特网商业安全的重要性以及因特网商业的飞速发展，我们将讲述一些密码安全性技术的使用，如7.6节将描述在大部分电子商务中使用的实际上的标准安全协议——传输层安全（TLS），我们还会描述一个专门为微事务设计的协议Millicent，参见www.cdk4.net/security。

因特网商业是安全技术的一个很重要的应用，但它不是唯一的应用。任何个人或是组织在存储和交互重要信息的地方都会用到它。在个人通信间使用加密的电子邮件已经成为大家关心的主题。我们将在7.5.2节中提到这场辩论。 272

7.1.3 设计安全系统

近年来，密码技术及其应用都得到了巨大发展，但安全系统的设计依然是一个十分困难的任务。出现这种局面的原因是，设计者们总是想尽可能地应付所有可能的攻击和漏洞。这就像是让程序员消除程序中所有的错误。在这两种情况下，都没有具体的方法能保证实现目标。按已知的最好的标准去设计，并进行非形式化的分析和检测。一旦设计完成，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.cdk4.net/security找到介绍在这个方向迈出的第一步——BAN认证逻辑[Burrows et al. 1990]及其应用——的介绍。

安全就是要避免大灾难和最小化一般的灾难。进行安全设计的时候，必须假设处于最坏的情况下。下面的“最坏情况的假设和设计指导”部分给出了一些有用的假设和设计指南。这些假设是本章讨论的技术思想的基础。

最坏情况的假设和设计指导

暴露的接口：分布式系统由提供服务或共享信息的进程组成。进程之间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。

不安全的网络：例如，消息源可以是伪造的——有些消息看似来自Alice，而其实是来自

Mallory。主机地址也可能是伪造的——Mallory用与Alice相同的地址连接到网络，并可以接收发送给Alice的消息的副本。

限制保密的时间和范围：当密钥产生时，我们相信这个密钥是安全的。随着密钥的使用时间越来越长、范围越来越广泛，它的安全风险也随之增加。一些保密措施（如密码和共享密钥）的使用时间应当是有限的，而且共享范围应该被严格控制。

攻击者能够获得算法和程序代码：一个秘密分布的范围越广泛，它被泄漏的风险也就越大。在当今如此大规模的网络环境中，现有的密钥加密算法是不够的。最好的保密方法是公布用来加密和认证的算法，而仅仅依靠加密密钥的秘密性。这样可以通过第三方的检验，增强对算法可靠性的信心。

攻击者可能访问大量资源：计算开销在迅速地下降。我们应该假设攻击者能访问一个系统生命期中计算能力最强的计算机，并通过执行大量命令产生不可预计的结果。

使可信库最小化：系统的各个部分都应当对系统的安全负责，而且系统的所有软、硬件组件都应该是可信的——这也常被称为*可信的计算库*。这个可信库中的任何缺陷或程序错误都可能产生安全漏洞，所以我们应该使可信库的规模最小。例如，不能信任应用程序来保证用户数据的安全。

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所有可能的威胁，即会破坏安全政策的方法，并给出解决威胁的机制的说明。这种说明可以采取非正式讨论的形式，或采用逻辑证明的形式（这种方式更好）。

在这张威胁列表中不可能列出所有的问题，因此在安全敏感的程序中还必须使用审计的方法。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。

一个安全日志会对用户的操作打上时间戳并按序记录。日志中的记录至少要包括主体的身份、所完成的操作（如删除文件、更新账户记录）、被操作对象的标识和一个时间戳。在可疑的地方，记录中还会包含对物理资源（网络带宽和外围设备）使用的记录或是在日志中记录对一些特殊对象的操作。后续的分析可以是基于统计的或基于搜索的。随着时间的流逝，即使没有可疑之处，这些统计信息也有助于发现异常的趋势或事件。

安全系统的设计必须在试图解决各种威胁的机制和这种机制带来的开销之间加以权衡。用来保护进程和进程间通信的技术可以涉及相当广的范围而且强大到足以对付几乎任何攻击，但使用它们的也会导致一些开销和不便：

- 在使用安全系统时，产生了额外的开销（在计算机效率和网络的使用上）。必须在这种开销和所要解决的威胁间加以权衡。
- 使用了不合适安全策略，会导致合法的用户也要执行不必要的操作。

不与安全相折衷，这样的平衡就很难达成，也似乎与本小节第一段中的建议相冲突。但安全技术可以根据估计的攻击开销来量化和选择。www.cdk4.net/security中描述的小型商业事务使用的Millicent协议就采用开销较低的技术。

在7.6.4节中，我们将回顾在IEEE 802.11 WiFi网络标准的安全设计中遇到的问题，作为对安全系统的设计过程中可能遇到的困难的一个例子。

7.2 安全技术概述

273 / 274

本节的目的是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。这里我们将非形式化地描述它们，更为严格的描述将在7.3节和7.4节给出。我们将使用图7-1中为主体所起的名字，并将为加密和签发的项目应用图7-2中所示的符号。

7.2.1 密码学

加密就是将消息编码以隐藏原有内容的过程。现代密码学包括多种加密和解密消息的安全算法，它们都基于密钥的使用。密钥是加密算法中的一个参数，也就是说，如果不知道密钥，就不可能解密。

K_A	Alice的密钥
K_B	Bob的密钥
K_{AB}	Alice和Bob共享的密钥
K_{Apriv}	Alice的私钥（只有Alice知道）
K_{Apub}	Alice的公钥（由Alice公布的，所有人都可以获得）
$\{M\}_K$	用密钥K加密的消息M
$[M]_K$	用密钥K签发的消息M

图7-2 密码符号

通常使用的加密算法有两类。第一类使用的是*共享的密钥*，即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*，即消息发送者用一个公钥（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钥，但只有接收者可以解密消息，因为他有私钥。

这两类加密算法都非常有用，并且在建立安全的分布式系统中得到了广泛的使用。公钥加密算法的处理能力一般是密钥算法的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种角色。我们在此只通过一些简单的场景概要地介绍一下。在本章后面的小节里，我们会详细讨论它们以及其他一些协议，并着重解决此处提到的几个未解决的问题。

在下面的场景中，我们假设Alice、Bob和其他参与者都已经对所用的加密算法达成了一致，同时也实现了这些算法。我们还假设任何密钥或私钥都会得到妥善保存，不会被攻击者获得。

秘密性和完整性 密码学用于维护暴露于潜在的攻击下的信息的秘密性和完整性，例如在通过网络传输的时候，信息很容易被窃听或者篡改，这是密码学在军事和情报活动中的传统作用。它是根据这样一个事实，由某个加密密钥加密的消息，只能由知道相应解密密钥的接收者才能解密。只要解密密钥被妥善保存（未泄漏给第三方），就能保持加密消息的秘密性。当然，还要求加密算法足以应付任何破解它的尝试。如果在加密过程中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。 275

场景1：用共享的密钥进行秘密通信 Alice想要秘密地给Bob发送一些信息。Alice和Bob共享密钥K_{AB}。

1）Alice使用K_{AB}和两人达成一致的加密函数$E(K_{AB}, M)$加密消息，并将任意数量的消息$\{M_i\}_{K_{AB}}$发送给Bob。（只要K_{AB}是安全的，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得到原来的消息了。

Bob现在可以读取原始的消息M。如果当Bob解密消息的时候，消息是有意义的，或者更好的情况是，它包括Alice和Bob之间达成一致的值，例如消息的校验和，那么Bob就可以知道这个消息确实是来自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怎样知道任何$\{M_i\}$是Alice以前发送的加密消息，而不是后来由Mallory截获并重发的？在进行这样的攻击时，Mallory并不需要有密钥K_{AB}——他只需拷贝表示消息的比特流，然后发送给Bob即可。例如，如果消息是一个付钱给某人的请求，那么Mallory就会让Bob多付一次钱。

我们将在本章后面给出这些问题的解决方案。

认证 密码学可以用来实现主体间通信的认证机制。主体用特定的密钥成功解密消息后，如果它包括正确的校验和或者（使用了加密的块链接模式，见7.3节）其他期望出现的值，则认为消

息是可信的。如果这个密钥只为通信双方所知，就可以推断消息的发送者具有相应加密密钥，也就可以推断出发送者的身份。如果密钥是私人所有的，则成功地解密也就认证了已解密的消息是来自特定的发送方。

场景2：和服务器间的认证通信 Alice想访问Bob拥有的文件，也就是她的工作单位的本地网中的一个文件服务器。Sara是一个被安全管理着的认证服务器。Sara向用户发送口令，并且保存着系统中所有主体的当前密钥（通过在用户口令上进行一些转换而得到）。例如，它知道Alice的密钥K_A和Bob的密钥K_B。在这个场景中，我们将谈到*票证*。票证是由认证服务器发出的一个加密项，包括向其发送票证的主体的身份和一个用作当前通信会话的共享密钥。

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请求一张访问Bob
276 的票证。

2）Sara用K_A加密应答消息，并回发给Alice，应答消息包括一个用K_B加密的票证（同访问文件的请求一起发送给Bob）和一个新的密钥K_{AB}，K_{AB}用于和Bob通信。因此Alice收到的应答形式为：$\{\{Ticket\}K_B, K_{AB}\}K_A$。

3）Alice用K_A解密应答（K_A是根据Alice的口令用同样的转换过程生成的，该口令没有在网络上传输。一旦被使用后，就从本地存储中删除它，以防泄露）。如果Alice从口令中生成正确的K_A，那么她就可以得到一个访问Bob服务的有效票证和一个用于与Bob通信的新的加密密钥。Alice不能解密或篡改票证，因为它是用K_B加密的。如果接收者不是Alice，那么就不知道Alice的口令，也就无法解密消息。

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）最初由Sara产生的票证实际上是$\{K_{AB}, Alice\}K_B$。Bob用自己的密钥K_B解密票证。Bob便可以得到Alice的身份认证（基于只有Alice和Sara知道Alice的口令事实）和一个用来和Alice交互的新的共享密钥K_{AB}。（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）。

上面的场景是最初由Roger Needleham和Michael Schroeder [1978]开发的认证协议的一个简化的版本，后来又在MIT [Steiner et al. 1988] 开发并使用的Kerberos系统上得到了使用，详见7.6.2节。在上面的简化版协议中，没有措施防止对旧认证信息的重放。这个弱点和其他一些弱点将在完整的Needleham-Schroeder协议（见7.6.1节）的描述中解决。

我们描述的认证协议取决于认证服务器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，并用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，此时使用单独的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，而公钥加密的出现让我们摆脱了这种两难境地。

质询的有效性：Needham和Schroeder在1978年取得了一个重要的突破，他们认识到用户的口令并不需要在每次认证时都发送给一个认证服务（这样会暴露在网络中）。相反，他们引入了加密*质询*的概念。在上面场景的第2步中，服务器Sara把用Alice的密钥K_A加密的票证发送给Alice。这里包括一个质询，因为Alice除非能解密这个票证，否则就不能使用它，而且Alice只有在知道K_A的情
277 况下才可以解密票证，而K_A来自于Alice的口令。冒充Alice的人不可能通过这一步。

场景3：使用公钥的认证通信 假设Bob已经产生了一个公钥/私钥对，下面的对话可以使Bob和Alice建立一个共享的密钥K_{AB}。

1）Alice访问一个密钥分发服务得到*公钥证书*，它给出了Bob的公钥。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从证书中读取Bob的公钥K_{Bpub}。（我们将在7.2.3节讨论公钥证书的构造和使用。）

2）Alice创建一个与Bob共享的新密钥K_{AB}，并用公钥算法和K_{Bpub}对新密钥加密。她将结果和一个能唯一标识公钥/私钥对的名字发给Bob（因为Bob可能有多个公钥/私钥对）。于是Alice发送给

Bob的是“密钥名字，$\{K_{AB}\}K_{Bpub}$”。

3）Bob从他的众多私钥中选出相应的私钥K_{Bpriv}，并用它解密K_{AB}。注意，Alice给Bob发送的消息在传输过程中可能会被破坏和篡改。结果是Alice和Bob不能共享密钥K_{AB}。如果存在这个问题的话，可以比较巧妙地解决：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就可以在解密的时候检查一下。

上面的场景说明了使用公钥加密发送一个共享的密钥的方法。这项技术被称为*混合密码协议*并被广泛使用，因为它利用了公钥加密算法和密钥加密算法两者的特点。

问题：这种密钥交换很容易受到中间人攻击。Mallory可能截获Alice最初向密钥分发服务索要Bob公钥证书的请求，并回复一个包括自己公钥的消息。然后，他就可以截获所有后续的消息。前面介绍过，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签发。同时，Alice必须确保Bob的公钥证书是由一个她在完全安全方式下收到的公钥（下面将会讲到）签发的。

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。它的作用和通常意义的签名相似，用于向第三方核实消息或文档在签名人完成后没有被改变过。

数字签名技术是基于将一个只有签名人才知道的秘密不可逆地绑定在消息或文档上实现的。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才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，但它不会为两个不同的消息产生相似的摘要值。加密的摘要附在消息上作为签名。通常按以下方式使用公钥加密：首先，签名人用他们的私钥产生一个签名；签名可以由任何接收者用相应的公钥解密。另一个要求是，验证人必须能确保这个公钥就是签名人的公钥，这使用公钥证书来解决，见7.2.3节的描述。 278

场景4：使用安全摘要函数的数字签名 Alice要对一个文件M签名，使得任何接收者都能验证她是这个文件的签发人。这样，当Bob通过某种途径或资源（例如来自消息或者一个数据库）接收到文件后访问这个签了名的文件，他就可以验证Alice是文件的签发人。

1）Alice为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的公钥K_{Apub}解密$\{Digest(M)\}K_{Apriv}$，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

数字证书是由一个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是一家银行。每当他的顾客和他建立联系时，他们需要确认他们是在和银行Bob交互，即使他们以前从来没有和Bob接触过。Bob则在授予用户访问他们的账号的权限前，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的银行账号的证书（见图7-3）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。证书用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，她就可以接受用这个证书为Alice付账。为此，Carol需要有Bob的公钥，而且还要进行验证，防止Alice签发了一个将自己名字关联到别人账号的假的证书。要进行这样的攻击，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。

Carol现在需要的是由可信权威机构签发的含有声明了Bob公钥的证书。我们假设Fred代表银

行家联盟，他是能证明银行公钥的人之一。Fred为Bob发行了一个公钥证书（参见图7-4）。

1. 证书种类：	账户号码
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7-3 Alice的银行账号证书

1. 证书种类：	公钥
2. 姓名：	Bob的银行
3. 公钥：	K_{Bpub}
4. 证明方：	Fred——银行家联盟
5. 签名：	$\{Digest\ (field2+field3)\}K_{Fpriv}$

图7-4 Bob银行公钥的证书

当然，这个证书取决于Fred公钥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真实的公钥K_{Fpub}，她才能信任这个证书。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信任的人那里收到一个签名的证书，而这个证书直接来自Fred。我们的例子说明了一个证书链，当前情况下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。信任通常不是绝对的，因此对权威机构的选择就必须取决于证书是打算给谁的。由于私钥有被泄漏的危险以及证书链可容许的长度会引发其他问题，证书链越长，冒的风险就越大。

如果小心解决了这些问题，证书链就成为了电子商务和真实世界其他事务的重要基础。它们有助于解决了大规模认证的问题：世界上有60亿人口，我们怎样才能在任意人之间建立起信任关系？

证书可用于验证多种声明的真实性。例如，一个组织或协会的成员可能要维护一份电子邮件列表，并只对组织内成员公开。解决这一问题的办法是让具有管理成员资格的经理（Bob）给每个成员发送一个成员资格证书（$S, Bob, \{Digest(S)\}\ K_{Bpriv}$），这里$S$是形如“Alice是友好社的一个成员”的语句，$K_{Bpriv}$是Bob的私钥。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拷贝，而管理系统会在检查证书后允许Alice加入这个列表。

为了使用证书，需要做两件事情：

- 证书要有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，特别是对权威机构。

279 ~ 280

我们将会在7.4.4节讨论这些需求。

有时需要收回一个证书。例如，Alice不想继续成为友好社的成员，但她或其他人还可能保留她的成员证书的拷贝。跟踪并删除所有这类证书的开销巨大甚至根本就不可能实现，而且取消证书的有效性也是不容易的，因为要通知所有可能接收这个被撤销的证书的接收者。通常，解决这种问题的办法是在证书中包含一个过期日期，收到过期证书的人应该将证书抛弃。证书的主体也必须请求更新自己。如果需要更加迅速地撤销，就要借助于以上提到的这些麻烦的机制了。

7.2.4 访问控制

本节我们将概述分布式系统中对资源访问控制的概念以及实现技术，在Lampson[1971]的一篇经典论文中非常清晰地介绍了保护和访问控制的概念，而非分布式的实现细节可以在许多操作系统的书中看到[stallings 1998b]。

从历史上看，分布式系统中的资源保护大部分是面向特定服务的。服务器收到下列格式的请求消息：<*op*, *principal*, *resource*>，其中*op*是所请求操作的名称，*principal*是发送请求的主体的一个标识或者一组证书，*resource*是操作所应用的资源。服务器必须先认证请求消息和主体的证书，然后进行访问控制，拒绝没有访问权限的主体的在特定的资源上完成某类操作的请求。

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面向特定应用的。例如，每天只允许Alice从银行取一次现金，而允许Bob取三次现金。访问控

制的决定通常留给应用层的代码来处理，但同时也为支持访问控制决定的大部分机器提供一些通用的支持。这包括主体认证、请求的签名和认证、管理证书和访问权限数据。

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，rights>对，列出了在域内执行的所有进程允许访问的资源以及在每个资源上所能进行的操作。保护域通常和给定的主体相关——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。从概念上讲，这个域包括主体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符决定的。权限是按照允许的操作来指定。例如，一个文件对这个进程可以读/写，而对另一个只可读。

保护域只是一个抽象。在分布式系统中普遍使用的实现方式有两种，即*权能*和*访问控制列表*。 281

权能：每个进程根据它所在的域中都持有一组权能。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，形式如下：

资源标识符	对目标资源的唯一标识
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

当服务认证了客户属于它所声明的保护域时，它就给客户提供权能。权能中的操作是目标资源定义的操作的一个子集，通常被编码成一个比特标志。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符，这可以使服务器立刻就能知道客户有权能标识的访问该资源的权限，能够进行权能指定的操作。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访问门锁所保护的大楼的关键。

权能保留了物理锁的钥匙的两个缺点：

- *钥匙被盗*：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的合法拥有者——他们可以用偷盗或其他不合法的手段来得到钥匙。
- *回收问题*：保管钥匙的资格会随时间的流逝变更。例如，曾经的钥匙拥有者不再是大楼主人的雇员，但他如果仍然保管或者复制了一把钥匙，他就有可能以不合法的方式来使用它。

针对物理钥匙的这些问题，唯一可行的解决办法是（1）将违法的钥匙拥有者送进监狱，但这并不能永远防止那些违法事情的发生。（2）换锁并把新钥匙发给当前所有合法的钥匙保管者，这是一种代价高昂的办法。

对于权能而言，类似的问题有：

- 由于不小心或者被窃听，权能可能会落入非法主体手中。一旦这样，服务器很难阻止他们非法使用权能。
- 取消权能是很困难的。持有者的状态可能会改变，因此其访问的权利也应相应地改变，但他们依然拥有着权能。

现在，已经有解决这两个问题的途径，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]。尽管加入这些信息使得原本简单的概念复杂起来，但权能依然是一项重要的技术。例如，它们可以和访问控制列表一起使用来优化对同一 282
资源的重复访问，它们为实现委托提供了最简洁的实现机制[见7.2.5节]。

注意，权能和证书具有相似性。回想一下7.2.3节介绍的证明Alice有银行账号的证书。证书与权能的区别在于没有允许操作的列表，也不对发出权能者进行认证。在某些环境下，权能和证书

是可以互换的概念。Alice的证书可以被看成对Alice的银行账号作一切账号持有者允许的操作的访问钥匙凭证，只要请求者能被证明是Alice本人。

访问控制列表：每个资源都有这个列表，有格式为<domain，operations>的项，它指出了对该资源有访问权限的域和对该域所允许的操作。一个域可以是由一个主体的标识指定，也可以是一个用于决定主体在域中资格的表达式。例如，“*文件的所有者*”是一个表达式，它可以用保存在文件中的所有者的标识和主体的标识作比较而求得该表达式的值。

这是大多数文件系统采用的方案（包括UNIX和Windows NT），每个文件都附有一组表示访问权限的比特值，同时权限被授予的域则是由存在于每个文件中的所有者信息定义的。

发到服务器的请求具有<op，principal，resource>的形式。对每一个请求，服务器会认证主体，并检验所请求的操作是否包含在相关资源的访问控制列表的主体项中。

实现 数字签名、证书和公钥证书提供了安全访问控制的密码学基础。安全通道具有性能优势，利用它可以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、Signer、ACL类和默认的认证方法进行访问控制，还有对证书、签名有效性及访问控制检查的支持。同时支持密钥和公钥密码学。Farley [1998]对Java的这些特色作了很好的介绍。对于Java程序（包括移动代码）的保护则基于保护域的概念——本地代码和下载的代码分别在不同的保护域内执行。每个下载的代码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。

CORBA提供了一个安全服务规约[Blakley 1999，OMG 2002b]，并给出了一个ORB模型以便提供安全通信、认证、基于证书的访问控制、ACL和审计，这将在17.3.4节做进一步的描述。

7.2.5 凭证

283

凭证是主体在请求访问某个资源的时候提供的一组证据。最简单的情况下，具有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）。通常这就是所有要提供的，但这些概念还可以再推广一下，以处理更加细微的需求。

对于用户来说下面的操作是很不方便的，即在每次需要访问受保护的资源时都让他们同系统交互并给出自己的身份验证，有一种折衷的方法是引入“凭证证明主体”的概念。这样，用户的公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，就可以认为请求就是由该用户所发出的。

证明的想法还可以进一步延伸。例如，在一个合作任务中，可能要求一些敏感的操作只能由团队中具有权限的两名成员来完成。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，并要表明在检查凭证时它们是在一起的。

类似地，投票选举时每张选票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。通常，访问控制检查包括对一个结合了证书的逻辑公式的求值。Lampson等人[1992]提出了一个认证逻辑，用于评估由一组凭证形成的证明；Wobber等人[1994]描述了一个系统，用于支持这种非常通用的检查方法；还可以在[Rowley 1998]中找到真实世界的合作任务中使用的更为有用的形式。

在设计实际的访问控制方案时，基于角色的凭证显得尤为有用[Sandhu et al. 1996]。对于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，角色可以用产生一个角色证书（它将主体与一个命名的角色相关联）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让某个主体或代理某个主体的进程，在另一个主体的授

权下，执行某个操作。下列情况需要使用委托：服务需要访问一个受保护的资源，以代表其客户完成一个动作。考虑接收打印文件的请求的打印服务器。拷贝整个文件将是对资源的浪费，所以用户只需将文件的名字发送给打印服务器，而由打印服务器代表发出请求的用户来访问这个文件。如果这个文件是读保护的，那么只有打印服务器得到临时的读权限，才能进一步工作。委托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委托权能来实现。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）访问某个资源（要打印的文件）。在支持权能的系统中，也可以不需要标识主体便达到同样的效果——访问某资源的权能放在请求中，一起发送到服务器。权能是一个不可伪造的、有关资源访问权限的编码集。 284

委托权限后，一般会将受委托方使用的权限限制在委托人权限的子集内。这样受委托的主体就不会错用权限。在我们的例子中，证书应该是有时间限制的，以降低打印服务器的代码被损害，而使得文件泄漏给第三方的风险。CORBA安全服务包括一个基于证书的权限委托机制，支持对权限的限制。

7.2.6 防火墙

3.4.8节已对防火墙进行过介绍。它可以保护内部网，对流入和流出网络的信息进行过滤。这里我们将讨论它作为安全机制的优点和缺点。

在理想的世界里，通信总是在相互信任的进程中进行，也总是使用安全的通道。但实际上，有许多原因造成这种理想的情况不能达到，有些原因源于分布式系统开放性本质中所固有的限制，有些原因则源于大多数软件中存在的错误。由于请求消息可以被轻松地发送到任何地方的任何服务器，而且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，这使得机密信息会很容易地从组织的服务器里泄漏出去。一些意想不到的东西也会渗透进组织的网络，例如蠕虫程序或病毒。对防火墙的进一步讨论参见[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。只有获得授权的通信消息才会发往本地的接收者。

访问内部网络会受到防火墙的控制，但访问因特网上的公共服务是不受此限制的，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网络免受来自组织内部的攻击，而它对外来访问的控制也是粗略的。人们需要细粒度的安全机制，以便个人用户在私密性和完整性不被损害的前提下能够和选定的其他人分享信息。Abadi等人[1998]提供了一个供外部用户访问私人Web数据的方法，它基于Web隧道机制，该机制可以被集成到防火墙中。该机制提供了一个基于HTTPS协议（TLS 上的HTTP）的安全代理，而这些可信和认证过的用户将通过这个代理访问内部的Web服务器。

防火墙对于避免拒绝服务攻击（如我们在3.4.2节提到的基于IP伪冒的那种攻击）不是很有效。问题在于这种攻击生成的消息会像洪水一般淹没任何一个像防火墙之类的防御点。所以必须在目标的上游对洪水般的进入消息加以处理。使用服务质量机制限制网络中的消息流，将它控制在目标所能处理的水平上，似乎还有可能缓解这种攻击。 285

7.3 密码算法

发送方按照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），这就是消息加密的过程。接收方必须知道这一转换规则，才能够将密文正确转换为原来的明文。其他主体无法解密该密文，除非他们知道转换规则。加密的转换过程由两个部分定义：*函数E*和*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，通过和密钥结合并加以转换，将它们转

化成加密的数据项，对于明文的变换很大程度上依赖于是密钥的值。我们可以将一个加密算法看成一簇函数的规约，通过给定的密钥可以从中选出一个函数。解密是由一个逆函数D来执行的，它也以一个密钥作为参数。对密钥加密而言，解密使用的密钥和加密使用的密钥是相同的：

$$D(K, E(K, M)) = M$$

因为需要对称地使用密钥，所以密钥密码学通常被称为*对称密码学*，而公钥密码学则被称为*不对称*的，因为它使用的加密密钥和解密密钥是不一样的。下面我们将描述这两种密码学常用的一些加密算法。

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，这样的函数被称为单向函数。加密信息所使用方法的有效性取决于具有单向性质的加密函数F_K的使用，也就是说，通过使用F_K可以抵御下面的攻击，即通过破解$\{M\}_K$而得到M。

对于下一小节将要介绍的设计巧妙的对称算法而言，K的大小决定了从明文M及加密后的$\{M\}_K$求出K的运算量。通常，最有效的也是最拙劣的攻击是一种被称为*强行攻击*的攻击形式。强行攻击方法的原理是：运行所有可能的K值，求出$E(K, M)$，和已知的$\{M\}_K$比较，直到匹配为止。如果K有N比特，那么强行攻击找到K平均要进行2^{N-1}次迭代，最多要进行2^N次迭代。因此破解K的时间是K的比特数的指数级时间。

不对称算法 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应用。第一个可行的公钥方案是由Diffie和Hellman[1976]年提出的，它作为一种密码学方法，消除了通信双方必须
286 要互相信任的前提。所有公钥方案的基础是*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，而在不知道密钥的情况下几乎不可能求出其逆。似乎寻找这样的函数并将其应用到实际的密码学方案中是Diffie和Hellman第一次提出的。此后一些实际的公钥方案陆续被提出并不断发展，它们都依靠使用大数函数作为陷门函数。

不对称算法所要使用的密钥对是从一个公共根导出的。7.3.2节描述的RSA算法使用任意选择的非常大的素数对作为根。再由一个单向函数从根导出密钥对。在RSA算法中，需要将两个大素数相乘——即使使用非常大的素数，该计算也只需几秒钟就可完成。最后的乘积N当然比被乘数大的多。在某种意义上来说，乘法的使用就是单向函数，因为想从乘积得到原来的被乘数——即乘积的分解——从计算上看是不可行的。

密钥对中的一个被用来加密。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，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的大小和至少一个密钥对要比对称密钥所需的安全密钥尺寸大得多，以保证N是不可分解的。因为这个原因，强行攻击RSA的可能性就很小了；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的；通常，块的大小为64比特。消息被分割成多个块，必要时，如果最后一块达不到，标准长度会被补足。每个块被独立地加密。一旦第一个块加密好了，就可以用于传输了。

对于简单的块加密，每个密文块的值都与前面的块无关。这样存在一个弱点，攻击者可以识别重复的模式并推导出它们和明文间的关系。而且，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*密码块链接*（CBC）来克服这些弱点。

密码块链接：在密码块链接模式中，每个明文块在加密前先和前面的密文块进行异或操作（XOR）（参见图7-5）。解密时，块先被解密，再和前面的密文块（应该将它保存起来）作XOR操作，从而得到原先的明文。这种方法能成功是因为XOR操作是*幂等*的，即两次应用它会产生原来的值。

CBC意图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——如果我们要与两个目的地建立加密的连接，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化向量，时间戳是一个很好的初始化向量，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：即使用相同的明文，也可转化成不同的密文。

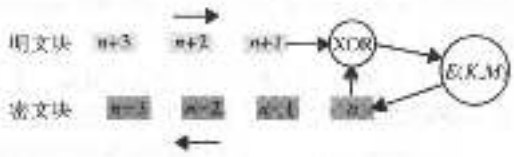


图7-5 密码块链

使用CBC模式必须保证加密数据在可靠的连接上传输，任意密文块的丢失都会导致解密失败，因为解密过程不能解密后续的密文块。因此它不太适合应用到第15章所描述的程序中，该程序要能容忍一些数据的丢失。为此，我们引入流密码的概念。

流密码 对于一些应用，例如对电话交谈加密，块加密的方法就不太合适了，因为数据流是实时产生的多个小块，数据采样可以小到8比特，甚至1比特。将它们补足到64比特再加密并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，它每次将明文中的1比特加密为密文。

这个提议听起来很难实现，但实际上很容易就可以将一个块密码算法转换成流密码算法，其技巧就在于构造一个密钥流产生器。密钥流是任意长度的比特序列，通过将其和数据流做XOR操作，即可完成加密的过程（参见图7-6）。如果这个密钥流是安全的，那么得到的加密数据流也是安全的。

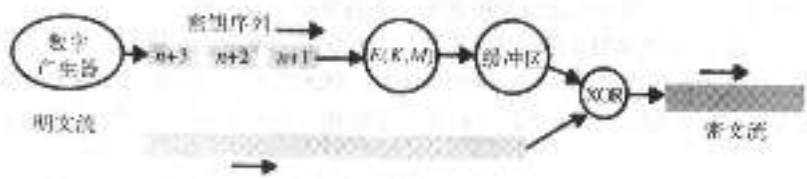


图7-6 流密码

这种想法和在智能社区避免窃听用到的“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，加入噪声，以掩盖谈话内容。如果嘈杂的房间谈话声和白噪声是单独录制的话，那么可以从嘈杂的谈话录音中去掉白噪声的录音，从而得到没有噪声的谈话内容。

密钥流产生器是通过对某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流而得到的。然后将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。密钥流还可以进一步利用CBC来伪装，得到的加密块就作为密钥流。任何可以产生一组不相同的非整数值的函数的迭代都可以作为密钥流产生器的候选函数，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。为了保证用于数据流的服务的质量，密钥流块应该在用到它们之前产生，同时产生它们的进程也不应执行太多的操作以免数据流被延迟。

287
-
288

因此，从原则上讲，在可以提供充足的处理能力实时加密密钥流的情况下，实时数据的加密可以像批处理数据一样安全。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，但它没有功能强大的处理器，这种情况下有必要降低它的密钥流算法的安全性。

密码算法的设计 有很多设计得很好的密码算法，例如$E(K,M)=\{M\}_K$隐藏了M的值，并且找到K的值的速度不可能比执行强行攻击快。所有的加密算法都是基于信息论[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：含混和扩散用于隐藏密文块M的内容，通过将内容和一个足够大的密钥K相组合，来对付强行攻击。

含混：使用非破坏性的操作（如XOR）和循环移位将每个明文块和密钥组合，产生一种新的

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，那么这种方法就可以抵抗基于特征频率知识的分析。（WWII German Enigma机器使用的是链式单字母块，它无法抵御统计分析。）

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。

在下面两小节中，我们将讨论几个重要的实用算法的设计。这些算法都是基于上述基本原理而设计的，它们也经过了严格的分析，可以抵挡所有已知的攻击，并有相当的安全性。除了TEA算法只是用于说明外，其他的算法都广泛应用在一些需要强大安全性支持的程序里。其中有些算法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需要考虑的问题，读者可以自己参阅Schneier [1996]来获取更多的信息。我们将在7.5.1节总结和比较这些算法的安全性和性能。

不需要理解密码算法的读者可以跳过7.3.1和7.3.2节。

7.3.1 密钥（对称）算法

近年来开发和发布了许多密码算法。Schneier[1996]中描述的对称算法多达25种以上，其中很
289 多算法都被认为对于已知的攻击是安全的。我们在此只讨论其中的三种。第一个是TEA，因为其在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。然后简单讨论DES和IDEA算法。多年来，DES一直是美国的国家标准，但现在它逐渐地带上了历史的色彩，因为56比特的密钥太短了，无法抵抗现代高性能硬件的强行攻击。IDEA采用128比特的密钥，它可能是最有效的对称块加密算法之一，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，建议采用一个新的算法来代替DES作为新的高级加密标准（AES）；2000年10月，从来自11个不同国家的密码学家提交的21种算法中，选出了Rijndael算法——这个算法因其健壮性和高效性而脱颖而出。下面我们将对此作详细的介绍。

TEA 上面概述的对称算法的设计原则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7-7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7-7 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含混和扩散。每个明文块是64比特的，所以就以两个32比特整数的形式保存在向量text[]中。密钥是128比特的，表示成4个32比特的整数。

在32轮的每一轮中，正文的两半分别与密钥逻辑移动后的部分以及彼此相组合，见程序的第5行和第6行。XOR的使用和正文的移位完成了含混，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，以免密钥因正文中某部分没有变化而泄

漏。解密函数是加密的逆函数，参见图7-8。

```
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7-8 TEA解密函数

这段程序提供了一个安全、合理、快速的密钥加密算法。它比DES算法速度快，而程序的简洁性也有助于优化和硬件实现。128比特的密钥足以对付强行攻击。它的作者和其他人只发现了两个很小的漏洞，在[Wheeler and Needham 1997]中有详细描述。

为了说明它的使用，图7-9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 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7-9 TEA的应用

DES 数据加密标准（DES）[National Bureau of Standards 1977]由IBM开发，随后被采用为美国的国家标准，在政府和商业中应用。在这个标准中，加密函数用56比特的密钥将64比特的明文映射成64比特的密文。算法中有16个依赖密钥的阶段，被称为轮（round）。每个轮中，要加密的数据都会根据由密钥决定的一组比特和三个不依赖密钥的移位值转换每个比特的位置和值。使用20世纪70~80年代计算机上的软件来实现该算法是非常耗时的，但它可以在高效的VLSI硬件中实现，并且可以轻松地集成到网络接口和其他的通信芯片上。

290 ~ 291

1997年6月，一次著名的强行攻击改写了DES未被攻破的历史。此次攻击是在一次竞赛中，为了演示低于128比特的密钥缺乏安全而进行的[www.rsasecurity.com I]。这次攻击是由一个因特网用户社团召集了1000～14 000台计算机（PC及工作站），并在上面运行相应的客户程序完成的[Curtin and Dolske 1998]。

客户程序的目的是破解出在已知的明文/密文采样中使用的密钥，并用它解密出原来加密的消息。客户与一个服务器交互，服务器负责协调客户的工作，向它们发送要检查的一段密钥值，并从客户处接收相应的进展报告。一般客户计算机将客户程序作为一个后台活动运行，其性能相当于200MHz的Pentium处理器。密钥在12周内被破解，检查的值占所有可能值（2^{56}或6×10^{16}）的约25%。1998年，由Electronic Frontier Foundation[EFF 1998]开发的机器可以用三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DES算法，但应该认为基本的DES已经过时了。目前常用的一种算法被称为*三重DES加密算法*（或3DES）[ANSI 1985, Schneier 1996]。它包括利用两个密钥K_1和K_2，并使用三次DES。

$$E_{3DES}(K_1, K_2, M) = E_{DES}(K_1, D_{DES}(K_2, E_{DES}(K_1, M)))$$

这相当于给出了一个112比特的密钥，也就有了充足力量对付强行攻击。但其缺点是效率低，因为它是将一个按现代标准来说比较慢的算法应用了3次。

IDEA 国际数据加密算法（IDEA）是在20世纪90年代初作为DES的替代者被开发出来的[Lai and Massey 1990, Lai 1992]。像TEA一样，它使用128比特的密钥来加密64比特的块，它主要基于群代数，有8轮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：这个性质对于能在硬件上实现的算法非常有用。

IDEA也被进行了广泛的分析，还没有发现重大的漏洞。它加密和解密时间约为DES的3倍。

RC4 RC4是一种由Ronald Rivest[Rivest 1992a]发明的流密码。密码长度不超过256字节。RC4很容易实现[Schneier 1996, pp.397-8]，而且加密与解密的效率约为DES算法的10倍。因此，RC4算法一度被大量的产品广泛使用，包括IEEE 802.11 WiFi网络；但不久之后，Fluhrer等人[2001]就发现了这个算法的缺陷，针对这个缺陷攻击者可以破解一些密钥，这也导致了802.11安全模块的重新设计（见7.6.4节的进一步讨论）。

AES 被美国NIST选做高级加密标准算法的Rijndael算法是由Joan Daemen和Vincent Rijmen[Daemen and Rijmen 2000, 2002]发明的。算法中密码块的大小和密钥的长度都是可变的，密钥长度可以为128、192或256比特，密码块的大小可以是128、192或256比特。密码块的大小和
292 密钥的长度都可以扩展为32比特的整数倍。算法根据密码块的大小和密钥的长度需要9～13轮完成。Rijndael算法可以被很多处理器实现，也可以通过硬件实现。

7.3.2 公钥（不对称）算法

至今只开发了少数几个实用的公钥方案。它们都使用大数的陷门函数来产生密钥。密钥K_e和K_d是一对很大的数，而加密函数用它们其中之一做运算，如对M作求幂运算。解密时使用另外一个密钥的一个类似的函数。如果求幂过程中使用了模运算，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，以供任何想要和他通信的人使用。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秘密信息。所有K_e的拥有者都可以将消息加密为$\{M\}_{Ke}$，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，这个问题的解决方法是仅在安全通信会话的初始阶段使用公钥。RSA算法显然是使用最为广泛的公钥算法，我们将详细介绍它。另一类算法是基于平面椭圆曲线行为派生的函数。这些算法具有同样级别的安全性，提供了用低开销加密/解密函数的可能，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）设计的公钥密码[Rivest et al. 1978]基于两个大素数

(大于10^{100}) 乘积的使用，其基本思想就是分解大整数的素因子的计算是非常困难的，不可能有效地计算出来。

尽管做了广泛的研究，但还没有发现RSA的漏洞，它现在被广泛使用。下面将给出RSA方法的概述。要找到密钥对e，d：

1）选择两个大素数，P和Q（每个数都大于10^{100}），并且计算：

$$N=P\times Q$$

$$Z=(P-1)\times(Q-1)$$

2）对于d，选择任意和Z互质的数（也就是，数d和Z没有公因子）。

我们用比较小的素数P和Q来说明计算的过程：

$$P=13,\ Q=17\rightarrow N=221,\ Z=192$$

$$d=5$$ 293

3）为找出e，求下列等式：

$$e\times d = 1 \bmod Z$$

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。

$$e\times d=1 \bmod 192=1,\ 193,\ 385,\ \cdots\cdots$$

385可被d整除

$$e=385/5=77$$

为了使用RSA方法加密正文，明文被分成长度为k比特的块，其中$2^k < N$（也就是说，一个块的数字值总是小于N；在实际应用中，k通常在512～1024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$M^{77} \bmod 221$

将加密正文c的一个块解密成原明文块的函数是：

$$D'(d, N, c)=c^d \bmod N$$

Rivest、Shamir和 Adelman证明，对于满足$0\leqslant P\leqslant N$的所有P，E'和D'是互逆的（即，$E'(D'(x))=D'(E'(x))=x$）。

参数e，N可以看成加密函数的密钥，类似地，d和N可以看成解密函数的密钥。于是我们可以写出$K_e=<e, N>$和$K_d=<d, N>$，并且得到加密函数是$E(K_e, M)=\{M\}_K$（注意这里指出了加密的消息只能由私钥K_d的所有者来解密），解密函数是$D(K_d, \{M\}_K)=M$。

值得注意的是，所有的公钥算法都有一个潜在的弱点，因为公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。这种攻击也称为*明文选择攻击*。这种攻击可以通过确保消息比密钥长来破解，此时破解明文的复杂度就已经超过了破解了密钥的复杂度，所以这种类型的强制攻击其实还不如对密钥直接攻击。

一个秘密信息的准接收者必须公布或发布$<e, N>$对，而自己保留d。公布$<e, N>$对并不损害d的安全性，因为想要知道d必须知道最初的两个素数P和Q，而这，又只有对N进行分解才能做到。对于大数的因式分解（我们提到P和Q都是$>10^{100}$的，于是$N>10^{200}$）即使是在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，在每秒执行100万条指令的计算机上，分解一个规模为10^{200}的数，所花费的时间将超过40亿年；而类似的计算任务在现在的计算机上只需要100万年即可完成。 294

RSA组织公布了一系列对100比特以上十进制数的分解挑战[www.rsasecurity.com II]。编写这本书的时候，174比特十进制数字（约576比特的二进制数字）被成功分解了，这使我们对使用512

比特密钥的RSA的安全性产生了怀疑。RSA组织（拥有RSA算法的专利权）建议采取至少768比特（即230比特十进制数字）长的密钥才能保证长期（约为20年）的安全性。一些程序中已用到了2048比特的密钥。

以上这些计算都假设现在知道的分解算法是可用的最佳算法。对于RSA和其他使用大素数乘法作为它们单向函数的不对称算法，在发现更好的分解算法后必将会变得很脆弱。

椭圆曲线算法 目前已经开发并测试了一个基于椭圆曲线的性质生成公钥/私钥对的方法。详细的内容可以参见Menezes以这个主题写的书[Menezes 1993]。密钥来源于一个与RSA不同的数学分支，它们的安全性不是建立在分解大数的困难性的基础上的。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远小于RSA。椭圆曲线加密算法可能在将来得到更为广泛的应用，尤其是对于那些包含了移动设备的系统，因为它们的处理资源很有限。由于该算法相关的数学知识包括了椭圆曲线一些非常复杂的性质，所以本书不再详细讨论。

7.3.3 混合密码协议

公钥密码学对电子商务而言是很便利的，因为它不需要安全的密钥分发机制（当然还需要对公钥进行认证，不过这并不麻烦，只需和密钥一起发送成为一个公钥证书就可以了）。但公钥密码的运算开销巨大，甚至对电子商务中经常遇到的中等大小的消息加密也是这样。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和对密钥交互进行加密，这个密钥将用于随后所有的通信中。我们将在7.6.3节的TLS实例研究中讨论混合协议的实现。

7.4 数字签名

强大的数字签名功能是安全系统的一个基本需求。数字签名可用于证明某些信息的场合，例如为了提供可信赖的声明，可将用户的身份绑定到他们的公钥上，或者将一些访问权限和角色绑
295 定到用户的身份上。

在各种商业和个人交易中，数字签名的必要性毋庸置疑。从文档出现伊始，手写签名就作为一种文件证明，用来满足收件人在以下方面证明文档的需要：

- 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。
- 不可伪造性：它证明了是签名者本人而不是他人特意签名了文档。该签名不能被拷贝和置于其他文档上。
- 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我们所希望的签名性质，因为难以检测签名是否被伪造和拷贝，而且文档在签名后可以被篡改，有时候签名者在无意间或在不知情的情况下被骗对文档进行了签名，但是考虑到欺骗有一定的难度且被查获后要承担的责任，我们可以接受这种威胁。与手写签名类似，数字签名是将一个唯一的且秘密的签名者属性绑定到文档中。在手写签名中，该秘密即为签名者的手写体模式。

保存在存储的文件或消息中的数字文档的性质和纸质文档的性质完全不同。数字文档一般很容易生成、拷贝和改变。简单地将作者的身份信息附加在文档之后，无论是一个文本字符串、一张照片还是一副手写体图像，对于验证而言没有任何价值。

因此需要这样一种方法，它将签名者的身份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，文档的日期不能由签名所保证，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，该问题并非源于手写签名。如果签名者故意泄漏了他们的私钥并且随后否认了已签名的文档，他们声称由于私钥并非私有，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]的主题下，已经设计出一些协议来解决这个

问题，但是它们相当复杂。

一个带有数字签名的文档比手写签名更难于伪造，但是“原始文档”对于数字文档意义并不大。正如我们将从对电子商务需求的讨论中所看到的那样，数字签名本身并不能防止电子货币的两次支付——还需要其他的措施来防范这种问题。我们现在将描述用于以数字方式签署文档的两种技术，它们都依赖密码技术的使用，将主体的身份信息绑定到文档中。

数字签名 主体A可以通过使用密钥K_A加密电子文档或消息M并且将加密的信息附加到M的明文和A的标识上，从而完成对M的签名。因此签名后的文档包括M、A、$[M]_{K_A}$。签名可以被随后接收文档的主体验证以确定文档是由A发出的，并且包含的内容M未被篡改。 296

如果使用一个密钥来加密文档，则只有共享该密钥的主体可以验证这个签名。但是如果使用公钥密码，那么签名者使用他自己的私钥加密，任何人只要拥有相应的公钥就可以验证该签名。这是对传统签名更好的模拟，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密钥密码还是公钥密码而有所不同。这两种情形将分别在7.4.1节和7.4.2节给予阐述。

摘要函数 摘要函数也称为安全散列函数，用$H(M)$表示。必须仔细设计摘要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'使$H(M)=H(M')$，那么会出现下述情况：一个不诚实的主体发送了消息M，但是当面临问题时，他可以声称他发送的原始消息是M'，并且说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 公钥数字签名

公钥密码特别适合于生成数字签名，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进行认证的方法如下（见图7-10）：

1）A产生一个密钥对K_{pA}和K_{sA}，并且把公钥K_{pA}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_{sA}加密摘要来产生签名$S=\{H(M)\}_{K_{sA}}$。

3）A把已签名的消息$[M]_K=M$，S 发送给B。

4）B用公钥K_{pA}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。

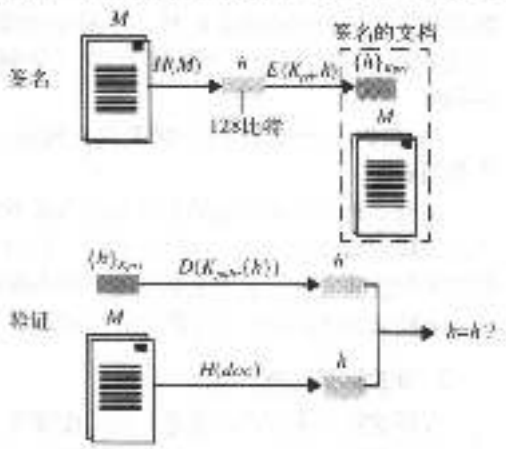


图7-10 公钥的数字签名

RSA算法非常适合用来构造数字签名。注意，这里签名者的私钥是用来加密签名的，这与秘密传输数据时，接收者用公钥来加密数据的情形相反。解释这种差别是非常简单的，一个签名只能用只有签名者知道的密钥来建立，但此签名可以被所有人认证。

7.4.2 密钥数字签名——MAC

一个密钥加密算法不应该用于加密数字签名并没有什么技术上的原因，但是为了验证这样的签名，密钥必须被透露出去，这会造成一些问题：

- 签名者必须安排验证者接收用来进行可靠签名的密钥。
- 在某些上下文和不同的时刻，有必要进行认证；在签名时，签名者可能不知道验证者的身份。为了解决这个问题，验证可以委托给一个可信赖的第三方机构完成，该机构持有所有签名者的密钥，但是这增加了安全模型的复杂度并且要求与可信的第三方进行安全通信。

- 我们不希望透露用于签名的密钥，因为这会削弱签名的安全性，一个签名可以被一个密钥持有人伪造，而该持有人未必是密钥的合法拥有者。

鉴于上述原因，用公钥方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。

当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时会出现例外。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节介绍的混合方法建立共享的密钥并用它生成低开销的签名。这些签名称为消息认证码（Message Authentication Code, MAC）。这个名字可以反映出它们有限的目的——它们基于一个共享的秘密，在一对主体之间认证通信。

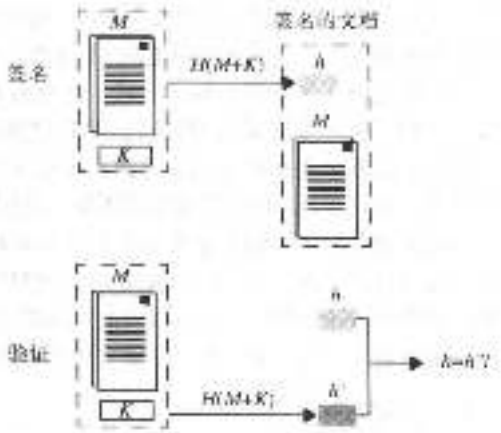


图7-11 使用共享密钥的低开销签名

一个基于共享密钥的低开销签名技术（如图7-11所示）可以为许多不同的目的提供足够的安全保障，我们将在下面进行阐述。这种方法基于安全通道的存在，通过该通道，共享的密钥可以被分发出去：

1）A产生一个随机密钥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认证A发出的消息的主体。这些主体是受信任的，不会泄漏共享密钥。

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的泄漏而受到损害。

3）接收者B将密钥K和接收到的文档M连接起来，计算摘要$h'=H(M+K)$。如果$h=h'$，那么签名即得到验证。

虽然这种方法有上述的不足，但是由于它不涉及加密，因此拥有性能上的优势（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TLS安全通道协议支持MAC的广泛运用，包括这里叙述的方案。该方法也可用于Millicent电子货币协议[www.cdk4.net/security]，在该协议中为小金额交易保持低处理开销是尤为重要的。

7.4.3 安全摘要函数

有许多种方法可产生固定长度的比特模式，这些比特模式可以刻画一个任意长度的消息和文档。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用于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较短的、定长的散列值，但是它作为数字签名方案的基础还不够。一个安全摘要函数$h=H(M)$应该有以下性质：

297～299

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。这个名字源于前两个性质。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唯一（因为摘要是一个信息减损的转化过程），我们需要保证，即使知道产生散列值h的消息M，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一点，那么他们可以不需要知道签名密钥，就从已签名文档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

造签名会十分困难，但是如果有耐心，他还是能办到的，所以必须对此进行防范。在生日攻击(birthday attack）情况下这种可能性显著增加：

1）Alice给Bob准备了两个合同版本M和M'，对Bob而言，M是有意义的，而M'对他没有意义。

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，如果她发现两个值是相同的，她可以进行下一步；如果不相同，她继续产生两个文档具有细微差别版本，直到两个文档产生匹配的散列值为止。

3）当她获得一对有着相同散列值的文档M和M'时，她把有意义的文档M给了Bob，让Bob用他的私钥对文档进行数字签名。当Bob把签好的文档发回给Alice，她可以用和M匹配的没有意义的版本M'替换M，并且保留着从M得来的签名。

如果我们的散列值有64比特长，那么平均只要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比特长才足以防范这类攻击。

这种攻击依赖于统计学悖论，即所谓的生日悖论（birthday paradox）——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]为这种在一个有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，从只有23个人的集合中寻找一对生日相同的人的概率，与从253人中寻找在某一个指定的日期过生日的人的概率是相同的。

为了满足上述性质，必须小心设计安全摘要函数。使用的比特级操作和它们的先后顺序与对称密码学相似，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任何算术方法和基于比特位的逻辑操作。源文本的长度通常包含在摘要数据里。 300

在实际应用中，广泛应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SHA-1（安全散列算法），这两个算法被美国国家标准和技术研究所（NIST）采纳为标准。这两个算法都经过仔细测试和分析，可以充分满足可预见的将来的安全需要，同时它们的实现相当高效。我们在这里只给出简短的描述。Schneier [1996]和Mitchell等人[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，源文本以512比特为一块，每一块又划分为16个32比特的段，每轮对一个段应用四个非线性函数中的一个，结果产生一个128比特的摘要。MD5是当前可用的最高效的算法之一。

SHA-1 SHA-1[NIST 2002]是一个产生160比特摘要的算法。它基于Rivest的MD4算法（与MD5算法类似），并附加了一些额外操作。运行速度比MD5慢得多，但是160比特的摘要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。SHA算法也被包含在标准[NIST 2002]中，SHA算法能产生更长的摘要（224、256和512比特）。当然，摘要越长，生成摘要的开销越大，需要的存储空间更大，数字签名和MAC通信的开销也越大。但是，根据公布的对SHA-1改进前的算法的攻击记录可以得知，SHA-1算法是易受攻击的[Randall and Szydlo 2004]。美国国家标准和技术研究所宣称将在2010年之前将美国政府软件用有更长摘要的SHA算法重新加密[NIST 2004]。

使用加密算法生成摘要 可以使用7.3.1节所述的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，让任何希望验证数字签名的人可以运用摘要算法进行有关的验证。加密算法被用于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是应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，用于进行全球性的名字和属性目录的构建[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7-12所示，它将一个公钥绑定到一个称为主题（subject）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（issuer）的实体发布。证书有一个有效期（period of validity），其中包含起止日期。<标志名>项指一个人、组织或其他有着足够上下文信息用以保证唯一性的实体的名字。在一个完整的X.500的实现中，这种上下文信息可以从命名实体所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、公钥
发布者	标志名、签名
有效期	不早于某日期且不晚于另一个日期
管理信息	版本，序列号
扩展信息	

图7-12 X.509证书格式

这种格式被包含在TLS协议中应用于电子商务，它在实际的服务和客户端的公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来获得X.509公钥证书。于是，对任何X.509证书都有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）的公钥证书。

2）验证签名。

SPKI方法 X.509方法基于标志名是全局唯一的，但这被认为是一个不实际的目标，它不能很好地反映当前的法律和商业实践[Ellison 1996]，因为个体的身份不能被看作是唯一的，而是相对于其他个人和组织时是唯一的。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（一个名字在世界范围内不可能唯一）中是相当常见。这就使验证链加长，因为存在许多可能的公钥证书的发布者，他们的签名必须通过一个验证链认证，最后认证被传给执行验证的主体知道的且信任的人。如果得到的认证足以让人信服，而且验证链中的许多步骤可以缓存起来，以便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钥基础设施（Simple Public-key Infrastructure, SPKI）的依据（参见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的公钥是K_{Apub}”。

7.5 密码实用学

301
~
302

在7.5.1节中，我们将比较前面介绍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，我们将讨论围绕密码技术的应用一些非技术问题。自从功能强大的密码算法出现在公共领域，还没有对发生在此学科上的大量的政治性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 密码算法的性能

图7-13给出了对称加密算法和本章讨论的安全摘要算法的性能比较，我们给出两个速度上的度量。在“PRB优化”这一列的数据是根据Preneel等[Preneel et al. 1998]提供的数据给出的。“Crypto++”这一列的数据则是最近刚刚从密码方案[www.cryptopp.com]的Crypto++开源库的作者处得到的。同时，在列标题中也注明了相应性能测试过程中使用的硬件速度。Preneel的实现是通过手工优化的汇编程序，而Crypto++的实现则是通过一个优化的编译器生成的C++程序。

密钥的长度决定了对其进行强制攻击所需的计算开销，加密算法的真实强度很难衡量，而且它依赖于算法能否成功地加密明文。Preneel等[1998]对主要的对称算法的强度和性能进行了有益的讨论。

	密钥长度/散列长度（比特）	PRB优化90MHz奔腾I（兆字节/秒）	Crypto++ 2.1GHz奔腾IV（Mbps）
TEA	128	—	23.801
DES	56	2.113	21.340
Triple-DES	112	0.775	9.848
IDEA	128	1.219	18.963
AES	128	—	61.010
AES	192	—	53.145
AES	256	—	48.229
MD5	128	17.025	216.674
SHA-1	160	—	67.977

图7-13 加密和安全摘要算法的性能

那么，上面的性能数字对于现实的密码应用程序（例如，为保证安全的网络交互（7.6.3节讨论的https协议）使用的TLS机制中的应用）有什么意见呢？网页的大小很少会超过100KB，所以任何一个网页的内容都可以采用任何一种对称算法由一个如今看起来很慢的单处理器在几毫秒之内完成加密。RSA算法主要用于数字签名，而且RSA算法也仅需几毫秒的执行时间。所以，算法性能对https程序的运行速度产生的影响是很小的。 303

不对称的算法（如RSA）很少用于数据加密，但是它们的性能对于签名服务还是很有吸引力的。Crypto++开源库的网页显示，利用图7.13中最后一列提到的硬件资源，使用带有1024比特密钥的RSA算法对一段待加密的信息（可以认为是由160比特的SHA-1算法生成的）进行签名，耗费时间为4.75ms，而验证这个签名则仅消耗了0.18ms。

7.5.2 密码学的应用和政治障碍

上述算法均在20世纪80年代到90年代之间出现，在这期间计算机网络开始用于商业用途，而同时计算机网络缺乏安全性也已成为制约其商业化的一大问题。正如我们在本章开始所述，美国政府非常抵制密码软件的出现。有两个原因，其一，美国国家安全局（NSA）有这样一个政策，它将其他国家可用的密码长度限制在一个较低的水平，使得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）以执法为目的，要确保它的机构拥有访问所有在美的私有组织和个人所使用的密钥的特权。

在美国，密码软件被列为军需品，有严格的出口限制。其他国家，特别是美国的盟国，也是采取类似的做法，在某些情况下甚至有更为严格的限制。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的一无所知，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如浏览器这样的软件的出口，因为该出口限制最终确定为只允许使用不超过40比特密钥（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扩散和保持密码软件不被其他国家所控制方面并不是特别有效，因为在美国国内外有许多程序员热衷于实现和分发密码代码。当前的情形是，实现了绝大多数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[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]，它最早是由Philip Zimmermann开发的，并由他和其他人分发出去。这种方式使得密码方法的使用不被美国政府所控制。PGP已经被开发出来并分发出去，目的是使所有计算机用户都可以在他们的通信中使用公钥密码算法，从而享受由此带来的私密性和完整性。PGP生成并管理公钥和私钥，它使用RSA公钥加密算法进行认证并把密钥传送给通信伙伴，并使用IDEA或者3DES密钥加密算法来加

304 密邮件消息和其他文档（PGP刚开发时，DES算法的使用被美国政府控制）。PGP有免费版本和商业版本，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]。

美国政府最终认识到NSA的观点是毫无作用的，并认识到这带给美国计算机业的危害（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包含有很强加密功能的软件产品。2004年，新的管理措施允许出口包含有高达64比特加密密钥，以及最高达1024比特的用于签名和交换密钥的公钥的软件产品。法规要求政府“审查”出口的软件，允许软件中采用更长的密钥。当然，美国并不持有密码软件生产或出版的专利，对所有知名的算法都已有了开源的实现[www.cryptopp.com]。这些法规仅会限制某些美国生产的商业软件的市场销售。

一些人提出通过立法来坚持软件必须包含只对政府法律执行和安全机构有效的入口或者后门，以便由国家来控制密码的使用。这些建议源自这样的设想：为了防止秘密的通信通道可以被各种各样的犯罪分子使用。在数字密码出现之前，美国政府一直依靠截取来分析公众的通信信息，而数字密码的出现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，同时也遭到关心自身隐私权的公民和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，但是政治上的努力依旧会持续下去，最终必将引入一个合法的使用密码的框架。

7.6 案例研究：Needham-Schroeder、Kerberos、TLS和802.11 WiFi

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。最为重要的密钥认证协议的应用是Kerberos系统 [Neuman and Ts'o 1994]，这是我们第二个案例的主题（参见7.6.2节）。Kerberos用于为网络上客户和服务器间提供认证服务，从而形成一个管理域（内部因特网）。

我们的第三个案例（参见7.6.3节）是关于传输层安全（TLS）协议的，这是专门用于满足电子交易安全的需要的，该协议目前被大多数Web浏览器和服务器支持，并被大多数Web商务交易采用。

最后一个案例（参见7.6.4节）将阐述工程安全系统的困难。1999年发布的IEEE 802.11 WiFi标
305 准就带有一个有关安全的规约。但是，随后的分析和攻击结果表明，这个规约有严重的不足，我们将揭示这种不足，并讨论其与本章提到密码学原理的关系。

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地管理密钥（和口令）的需要而开发的。在这项工作[Needham and Schroeder 1978]发布的时候，网络文件服务刚刚出现，在局域网中迫切需要更好的安全管理方法。

在管理型网络中，需要能由以质询（见7.2.2节）的形式发布会话密钥的密钥服务来满足，这就是Needham和Schroeder开发的密钥协议的目的。在同一篇论文中，Needham和Schroeder也陈述了一种基于使用公钥认证和密钥分发的协议，该协议不依赖已有的密钥服务器，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钥版本，但是将在7.6.3节描述的TLS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地获得共享密钥的方式。为了做到这一点，它必须使用加密消息与客户通信。

Needham-Schroeder密钥 在该模型中，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，为达到这个目的A进程可以获得一个密钥。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（后者在一个B可知而A不知道的密钥中被加密，因此B可以对其进行解密并且该密钥在传输过程中未被篡改）。

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密钥。密钥只用来认证连接到认证服务器的客户进程，并在客户进程和认证服务器之间安全地进行消息传输。该密钥从不泄露给第三方，在密钥生成后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，例如以书面形式或口头形式，以避免密钥暴露在网络上）。一个密钥与在集中式系统中用来认证用户的口令等价。对于主体是人的情况，认证服务持有的名字是他们的“用户名”，密钥是他们的口令，它们都是由用户在向代表他们的客户进程发出请求时提供的。

这个协议基于认证服务器产生和传送的票证。票证是一个加密了的消息，它包含用于在A和B之间通信的密钥。我们将Needham和Schroeder的密钥协议中的消息制成表格，如图7-14所示。其中S是认证服务器。

消 息 头	消 息	注 释
1. $A \rightarrow S$:	A, B, N_A	A请求S提供一个用于与B通信的密钥
2. $S \rightarrow A$:	$\{N_A, B, K_{AB}$ $\{K_{AB}, A\} K_B\} K_A$	S返回用A的密钥加密的消息，消息含有新生成的密钥K_{AB}和一个用B的密钥加密的“票证”。当前时间N_A说明该消息是响应前一个消息的。由于只有S知道A的密钥，所以A相信是S发送了消息
3. $A \rightarrow B$:	$\{K_{AB}, A\} K_B$	A将“票证”发送给B
4. $B \rightarrow A$:	$\{N_B\} K_{AB}$	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. $A \rightarrow B$:	$\{N_B - 1\} K_{AB}$	A通过返回一致的当前时间N_B的转换给B，证明它是前一个消息的发送者

图7-14 Needham-Schroeder密钥认证协议

N_A和N_B是当前时间。当前时间是一个整数值，它被加入到消息里以说明该消息是新近产生的。当前时间只被使用一次，并在需要的时候生成。例如，当前时间可以是一系列顺序的整数值或者可以通过读取发送机器的时钟值生成。

如果成功完成协议，那么A和B都可以确定任何从对方接收到的用K_{AB}加密的消息确实是来自对方的，任何发送给对方的用K_{AB}加密的消息只能被对方或S（S被认为是可信任的）解密，这是因为只有传送带有K_{AB}的消息才能用A或B的密钥加密。

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在A的授权下客户程序运行错误而被置于一个暴露的地方），他就可以假扮A，并使用上述内容发起和B的信息交换。这种攻击会损害密钥K_{AB}的旧值，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，但多数观点认为应该包括这种可能性。通过给消息3增加当前时间或时间戳可以弥补这个不足，所以消息变成：$\{K_{AB}, A, t\} K_{Bpub}$，$B$解密这个消息并检查$t$是否是新近的，这就是Kerberos采取的解决方案。

7.6.2 Kerberos

Kerberos是20世纪80年代在MIT大学[Steiner et al. 1988]开发出来的，目的是为MIT校园网和其他内部因特网提供一系列认证和安全设施。根据用户和组织的经验和反馈，Kerberos协议经历了多次修订和改进。下面要描述的是版本5[Neuman and Ts'o 1994]，它遵循因特网的标准轨迹（参见RFC1510[Kohl and Neuman 1993]），目前被许多公司和大学使用。Kerberos的实现源代码可以从

306
~
307

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，并且Windows 2000操作系统把它作为默认的认证服务。扩展Kerberos的提议认为可以利用公钥证书来进行主体的初始认证（参见图7-15步骤A）[Neuman et al. 1999]。

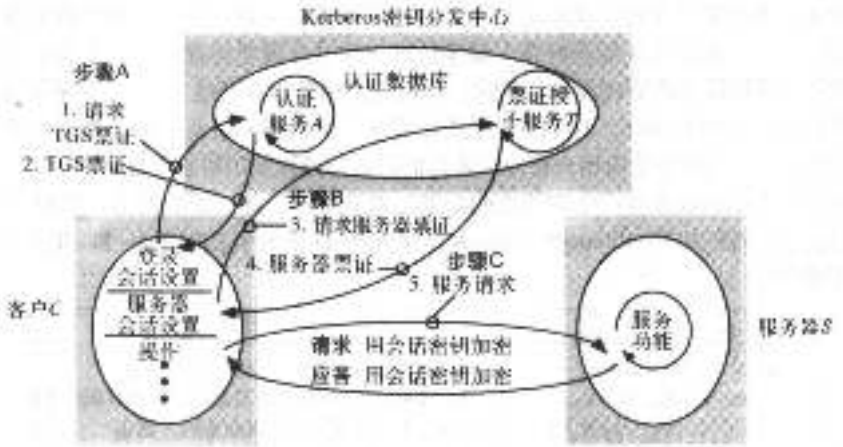


图7-15 Kerberos系统体系结构

图7-15显示了Kerberos的进程体系结构。Kerberos处理二类安全对象：

- 票证：Kerberos票证授予服务给每个客户发一张标记，该标记用于发送给某一个服务器。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以供客户和服务器使用。
- 认证：由客户构造的一个标志，并发送给服务器，用于证明用户身份以及任何与服务器的通信的currency。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，并非必须要加密；会话密钥就是用来对与要求加密的与服务器之间的通信运行加密，也用来对所有认证器加密（参见上面的描述）。

客户进程对它们所使用的每个服务器都必须提供票证和会话密钥。为客户-服务器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客户在几小时内使用以便与某一个服务器进行交互，直至到期为止。 308

一个Kerberos服务器也称为一个密钥分发中心（KDC），每个KDC提供认证服务（AS）和票证授予服务（TGS）。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（ticket granting ticket）和用来与TGS通信的会话密钥。因此，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。

Needham-Schroeder协议与Kerberos协议很接近，Kerberos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：

- 防止从网络中截取的旧消息的重播，或重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）。
- 应用票证生命期，使系统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，所用符号如下所示。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。

符号：

A	Kerberos认证服务的名字	n	当前时间
T	kerberos票证授予服务的名字	t	时间戳
C	客户的名字	t_1	票证有效期的开始时间
		t_2	票证有效期的截止时间

Kerberos票证有固定的有效期：从t_1开始，到t_2结束。客户C访问服务器S的票证的形式如下：$\{C, S, t_1, t_2, K_{C,S}\}_{K_S}$，记做$\{\text{ticket}(C, S)\}_{K_S}$，客户名包含在票证中，以免被冒充者使用（将在后文介绍）。图7-15中的步骤和消息号对应于描述栏A中的内容，注意消息1没有加密，也不含C的口令。它包含当前时间值，用来检查应答的有效性。

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通信的票证
2. A→C: TGS会话密钥和票证	$\{K_{C,T}, n\}_{K_C}, \{ticket\ (C, T)\}_{K_T}$，包含$C, T, t_1, t_2, K_{C,T}$	A返回一条消息，其中包含用A的密钥加密的票证和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

消息2有时称为“质询”，因为它发送给请求者的信息是只有知道C的密钥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$K_{C,T}$和用于票证授予服务的有效票证；否则，它获得无意义的信息。服务器有它们自己的密钥，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通信，以多次获得其他服务器票证，直至票证到期为止。因此，为了获得某一服务器S的票证，C构造一个用$K_{C,T}$加密的认证器，形式如下：$\{C, t\}_{K_{C,T}}$，记做$\{\text{auth}\ (C)\}_{K_{C,T}}$，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	$\{auth\ (C)\}_{K_{C,T}}$, $\{ticket\ (C, T)\}_{K_T}, S, n$	C请求票证授予服务器T提供与同一服务器S通信的票证
4. T→C: 服务票证	$\{K_{C,S}, n\}_{K_{C,T}}, \{ticket\ (C, S)\}_{K_S}$	T检查票证，若票证有效，T就生成新的随机会话密钥$K_{C,S}$，并用S的密钥K_S加密的S的票证一起返回

然后C开始向服务器S发出请求消息：

C. 发有一个带有票证的服务器请求		
5. C→S: 服务请求	$\{auth\ (C)\}_{K_{C,S}}$, $\{ticket\ (C, S)\}_{K_S}, request, n$	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$K_{C,S}$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（为减少需要的消息数，可以把它包含在含有服务器对请求的应答的消息中）：

D. 认证服务器（可选）		
6. S→C: 服务器认证	$\{n\}_{K_{C,S}}$	（可选）：S向C发送当前时间n，n用$K_{C,S}$加密

309 - 310

Kerberos的应用 Kerberos是MIT为在Athena项目中使用而开发的，是面向大学教育的校园网计算设施的，其中有许多工作站和服务器，为5000多比特用户提供服务。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，而对服务器（除了Kerberos服务器外）提供了多余的安全保障用以防止利用软件配置进行物理干扰。

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都进行了扩展，从而在每个客户-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。用户的口令只有用户自己和Kerberos认证服务知道。服务拥有的密钥只为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。如果用户名通过认证服务的认证，则返回用该用户的口令加密的会话密钥、当前时间和用于TGS的票证。登录程序在口令提示下尝试用用户键入的口令解密会话密钥和当前时间。如果口令正确，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TGS通信时使用。这时，登录程序可以从内存中删除用户口令，因为票证现在可以用于认证该用户。然后，这台工作站上的用户的登录会话开始。注意，用户的口令从来不暴露在可能被监听的网络上——它只保存在工作站上，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，它就向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时，用户的工作站上的rlogin命令程序从Kerberos票证授予服务处获得票证用来访问rlogind网络服务。在用户希望登录的计算机上，rlogin命令程序响应远程机器的rlogind进程的要求，发送票证和一个新的认证器。rlogind程序使用rlogin服务的密钥解密票证，并检查票证的有效性（即票证是否过期）。服务器必须小心地把它们的密钥存储到入侵者难以访问的地方。

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为新近产生的（认证器只能使用一次）。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
311
的名字和口令，因为rlogind程序已经知道用户的身份，并建立一个远程用户的登录会话。

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。可以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，可以很容易地被替换掉。

Kerberos服务是可扩展的——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。大多数主体仅在一个域中登记，但Kerberos的票证授予服务器（TGS）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。

在一个域中可以有多个认证服务器，它们都有同一个认证数据库的备份。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）负责更新主拷贝，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。

为了使这种方案对用户透明，TGS的生命期应该至少与可能最长的登录会话一样长，因为使用过期的票证会导致服务请求被拒绝，唯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，域的票证生命期一般为12小时。

对Kerberos的评价 上面描述的Kerberos 版本5针对早期版本的一些批评做了改进[Bellovin and Merritt1990，Burrows et al. 1990]。对Kerberos 版本4最主要的批评是认证器中的当前时间是用时间戳来实现的，且防止认证器重播至少需要客户和服务器时钟松散同步。如果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，并且能防范安全攻击。有关时钟同步协议的内容请参考第11章。

Kerberos5的协议定义允许认证器中的当前时间可以用时间戳或者序号实现，无论用哪种方法，

都要求它们是唯一的，并且服务器应该保留最近收到的每个客户的当前时间，以便检查它们有没有重播。这种要求实现起来很不方便，并且在服务器出现故障时难以得到保证。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——TGS票证的有效期通常只有几个小时。这个有效期必须选得足够长，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。这可能会给某些商业应用带来困难，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实际应用。 312

7.6.3 使用安全套接字确保电子交易安全

安全套接字层（SSL）协议最初是由Netscape公司[Netscape 1996]开发的，它提出了一种标准用于满足上述需求。SSL的扩展版本传输层安全（TLS）协议已经被采纳为因特网标准，具体描述参见RFC 2246 [Dierk and Allen 1999]。大多数浏览器都支持TLS协议，它广泛应用于因特网电子商务。它的主要特性如下：

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都包含特定的加密算法。实际上，一些国家的法律试图限制只能在这些国家使用某些加密算法。TLS可以在连接的两端进行初始化握手通信时，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因为没有足够的公共算法而导致连接尝试失败。

自举安全通信 为了满足安全通信的要求而不需要事先协商或第三方的帮助，可以用与前面提过的混合方案类似的协议建立安全通道。使用未加密的通信进行初始化交换，然后使用公钥密码，一旦建立共享密钥，就可以转换到密钥密码学上来。每个转换都是可选的，都通过协商进行。

因此，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因为执行不必要的加密操作而消耗掉。

TLS协议的细节已经被公布并标准化了，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。TLS已被整合到许多应用软件中，其安全性也经过独立审核得到验证。

TLS由两层组成（参见图7-16）。一层是TLS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TLS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TLS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。TLS记录协议是一个会话层协议，可以用来在保证安全性、完整性和真实性的前提下在进程之间透明地传送应用层数据。这些就是我们在安全模型（见2.3.3节）中为安全通道指定的性质，但是这些性质在TLS中是可选的。通信各方可以选择是否在每个方向上都部署消息的解密和认证。每个安全会话被赋予一个标识符，通信各方可以在缓存中存储会话标识符以备以后重用。当要求与相同的一方进行其他安全会话时，便可避免建立新会话的系统开销。 313



图7-16 TLS协议栈（图7-16～图7-19基于Hirsh[1997]中的图表，并得到Frederick Hirsch的出版许可）

TLS被广泛用于在现有应用层协议之下增加一个安全通信层。它最常用于因特网商务和其他安全性敏感的应用中，以保证安全的HTTP交互。几乎所有Web浏览器和Web服务器都实现了TLS：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TLS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，TLS是事实上的标准，它通过提供CORBA和Java的 API，为商业和公共领域提供了多种可用的实现选择。

TLS握手协议如图7-17所示。握手操作是在一个已建立的连接上进行的。它通过交换已认可的选项和参数来建立TLS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变化。握手协议也可以在之后改变一个安全通道的规约时调用，例如，在通信开始时可能只用消息认证码来认证消息。在这之后可以使用加密。这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而实现的。

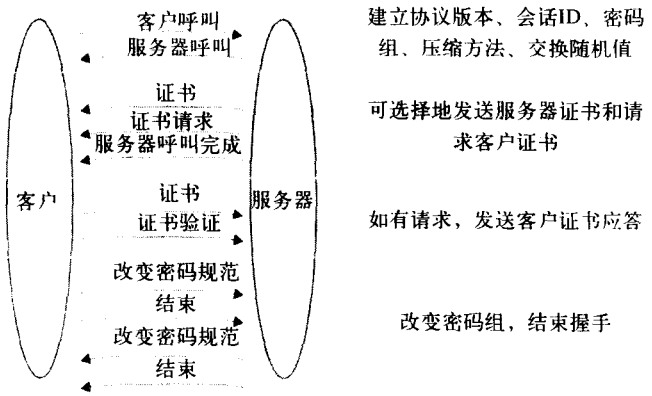


图7-17 TLS握手协议

TLS初始化握手易受到7.2.2节场景3所述的“中间人”攻击。为了防止这种情况，用来验证接收到的第一个证书的公钥可以通过一个单独的通道传送——例如，经由CD-ROM交付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钥。另一个众所周知的服务的客户防范措施，是基于在它的的公钥证书中包含了服务的域名——客户只能用和域名相一致的IP地址来处理服务。

TLS支持密码函数的多种选项。它们统称为*密码组*。一个密码组为图7-18所示的每个特性包含了一个选项。

组件	描述	例子
密钥交换方法	用来交换一个会话密钥的方法	带公钥证书的RSA
数据传输密码	用于数据的块密码或流密码	IDEA
消息摘要函数	用于创建消息认证码（MAC）	SHA-1

图7-18 TLS握手配置选项

客户和服务器上预装各种带有标准标识符的常用密码组。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。

接下来，通信双方按照X.509格式交换签名的公钥证书进行互相认证。这些证书可能是从一个公钥权威机构获得的，或者只是为此目的临时生成的。在任何情况下，至少有一个公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控制消息，用公钥加密后发送给另一方。控制消息是一个大随机数，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）和用来认证消息的消息认证密文。当这些工作完成后，一个安全会话就开始了。这是由通信双方交换的ChangeCipherSpec（改变密码规约）消息触发的。随后是Finished（结束）消息。一旦交换了Finished消息，所有后续的通信就可以根据所选的密码组连同设定的密钥进行加密和签名。

314 / 315

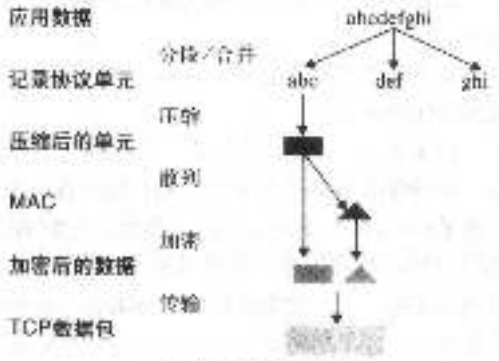


图7-19 TLS记录协议

图7-19显示出记录协议的操作。一个要传输的消息首先被分割成便于处理的块，然后有选择地压缩这些块。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参与加密和数字签名算法中对大量数据的处理工作，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TLS记录层中建立，由TLS记录层执行所有转换，这种转换比独立转换更有效。

加密和消息认证（MAC）转换部署了经协商后的密码组中指定的算法，如7.3.1节和7.4.2节所述。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　TLS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钥进行密钥交换。因为密码在握手中协商，所以它不依赖于任何专门的算法，也不依赖于会话建立时的任何安全服务。唯一需要的是权威机构发布的通信双方认可的公钥证书。

由于作为TLS基础的SSL协议及其参考实现的公布[Netscape 1996]，它逐渐成为争论的主题。早期的设计已经有了一些修改，作为一种有价值的标准，它得到了广泛的认可。现在TLS已经被集成到大多数Web浏览器和Web服务器中，也被应用于诸如安全Telnet、FTP等其他应用中。商业和公共领域[www.rsasecurity.com，Hirsch 1997，www.openssl.org]实现通常以程序库和浏览器插件的形式供用户使用。

316

7.6.4 IEEE 802.11 WiFi安全设计中的缺陷

3.5.2节描述的面向无线局域网的IEEE 802.11标准最初是于1999年发布的[IEEE 1999]。从发布之日起，它就被广泛应用在移动通信领域，有许多基站、笔记本电脑和便携设备都实现了这个规约。遗憾的是，人们不久就发现了这个标准的安全设计在某些方面有严重的缺陷。我们将简要介绍它的最初设计和安全缺陷，并以此作为7.1.3节提到过的安全设计困难的一个实例。

大家认为，无线网络比有线网络更容易遭受攻击，因为网络和传输数据很容易被装备有同频收发器的设备所窃听和篡改。最初的802.11协议用于为WiFi网络提供访问控制，并且依照称为有线等效加密（Wired Equivalent Privacy，WEP）的安全规约保证传输数据的私密性与完整性。WEP包含下面几项，网络管理员可以有选择地激活这些项。

访问控制：通过质询-应答协议进行访问控制（cf. Kerberos，7.6.2节），即当一个节点加入网络时，基站会质询该节点是否有正确的共享密钥。网络管理员会指定一个密钥K，并将K在基站和所有已认证的设备之间共享。

私密性与完整性：使用任一种基于RC4流密码的加密机制来保证私密性与完整性，加密过程中使用的密钥与访问控制中使用的密钥，都为K。密钥长度可以是40、64或128比特。每个分组通过包含加密校验和来保证其完整性。

在803.11标准公布不久，便被发现有下列缺陷和设计弱点：

网络用户共享单一的密钥是设计中的一个缺陷，因为在实践中会有下面的问题：

- 密钥可能在未受保护的信道上发送给一个新的用户。
- 一个粗心的或是恶意的用户（例如，心怀不满的前雇员）拥有访问密钥的权限，他们会破坏整个网络的安全性，而且这种破坏有可能完全不被发觉。

解决办法：像TLS/SSL（见7.6.3节）所采用的一种基于公钥的协议一样，通过协商获得私有密钥。

基站是不需要认证的，所以一个知道当前共享密钥的攻击者可以采取欺骗手段，窃听，添加或篡改任何消息。

317 解决办法：基站应当提供一个证书，它可以通过第三方提供的公钥被认证。

WEP不恰当地使用了流密码，而没有使用块密码（见7.3节对流密码和块密码的描述）。图7-20给出了802.11 WEP安全协议下的加密和解密流程。每个分组都通过与一个RC4算法产生的密钥流进行XOR操作来加密。接收站利用RC4算法产生一个相同的密钥流，并通过XOR操作对每一个分组进行解码。为了避免密钥流在分组丢失或被破坏时产生同步错误，RC4算法会用一个新的起始值重新开始，这个起始值是通过在全局共享密钥后面连接一个24比特的初始值得到的。这个初始值被更新并被包含在每一个传输分组中。共享密钥在大多数应用中不会轻易改变，所以起始值仅有$S = 2^{24}$（约10^7）个不同的状态。因此，在发送过10^7个分组后就会产生重复的起始值以及密钥流。在实际系统中，这种情况在几个小时内就会发生，而且当有分组丢失时，产生重复起始值的周期会更短。攻击者从截获的加密分组中总可以侦测出重复出现的起始值，因为这些起始值显式地包含于分组中。

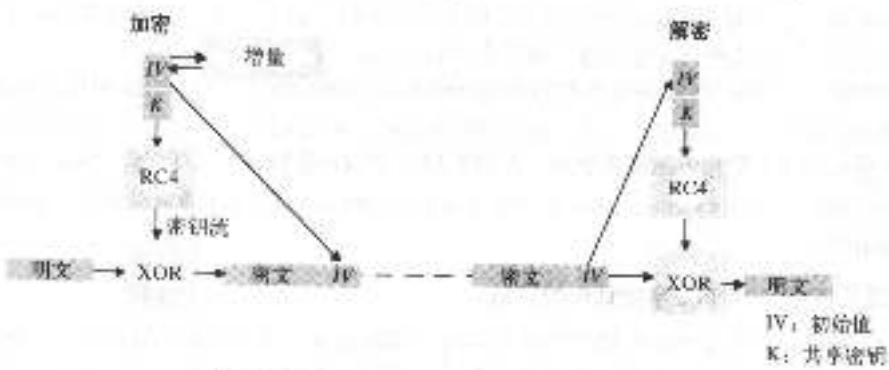


图7-20 IEEE 802.11 WEP中使用的RC4流加密

RC4规范中对密钥流的重复问题给出了明确的警告。因为如果攻击者截获了加密分组C_i，并且知道明文P_i（例如，通过猜测密文是一个标准的服务器问讯信息）就能计算出用于加密分组的密钥流K_i。同样的K_i值在S个分组之后又会重现，于是攻击者就可以通过已知的K_i来解密这个分组。通过正确地猜测明文分组，攻击者最终可以解密大部分的分组。Borisov等[2001]首先指出了这个安全缺陷，并领导了对WEP安全机制的重新评估以及在802.11的更新版本中使用的新的安全机制。

解决办法：根据最坏情况下密钥序列重复出现时间，选择一个小于它的时间段，每个时间段后通过协商获得新的密钥。和在TLS中一样，这个过程需要一个明确的中止代码。

318 正如7.5.2节中论及的，由于美国政府严格限制出口设备的密钥长度最多为40比特（后改为64比特），因此，802.11标准中同时引进了40比特和64比特两种密钥长度。但40比特密钥很容易被强行攻击破解，因此40比特密钥提供的安全保障有限。即使是64比特的密钥也会有因持续攻击而被破解的危险。

解决方法：使用128比特密钥。最近许多WiFi产品都使用了128比特密钥。

在802.11标准公布后，就发现RC4流密码即使在密钥流没有出现重复的情况下，通过观察真实的数据流质量也可能泄漏密钥[Fluhrer et al. 2001]。这个缺陷已经在实践中被证明，这个缺陷说明即使使用128比特的密钥，WEP方案也并不安全，有些公司也因此限制他们的雇员使用WiFi网络。

解决办法：采用类似TLS的办法，提供一种协商机制来决定选择何种密码规约以及加密算法。RC4算法是被紧紧绑定在WEP标准中的，它不提供选择其他加密算法的协商机制。

用户通常不会部署WEP提供的保护机制，其原因可能是用户没有意识到他们的数据是暴露的。这并不是标准设计时的缺陷，而是产品所基于的市场造成的。大多数产品的初始设置都会关闭安全功能，而且与安全风险相关的文档常常不足。

解决办法：提供更好的默认设置和文档。但是用户往往更注重获得更好的性能，而当硬件可用的情况下，开启加密功能会明显减慢通信的速度。用户避免使用WEP加密功能的需求导致基站需要增加新的特性，即基站不可以像平常那样将含有识别信息的分组广播出去，并且拒绝从未经认证的MAC地址（见3.5.1节）发来的分组。但上述措施也不足以提供足够的安全保障，因为网络中的分组容易被截获（“嗅探”），而且可以通过修改操作系统轻易地篡改MAC地址。

2004年发布的IEEE 802.11i 是针对802.11安全标准的升级版本，它修正了上述的所有缺陷，在新的版本中使用了互相认证，通过动态的协商获得成对的密钥，以及使用AES加密算法等措施[IEEE 2004b, Edney and Arbaugh 2003]。

7.7 小结

分布式系统常常面临安全威胁。保护通信通道和可能成为攻击目标的用于信息处理的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。要小心地设计安
319 全协议以防止出现漏洞。安全系统的设计从一系列威胁和一组最坏情况的假设开始。

安全机制基于公钥密码学和密钥密码学。密码算法以某种方式搅乱原有的消息，在不知道解密密钥的情况下不可能对密文解密。密钥密码学是对称的，即加密和解密使用相同的密钥。如果通信双方共享一个密钥，那么他们可以交换加密后的信息，而不存在被窃听和篡改的风险，并且能保证信息的真实性。

公钥密码学是非对称的，即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密钥。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。访问控制方案把权限分派给持有凭证的主体，使之能对分布式对象和对象集合执行操作。权限可以保存在与对象集合相关联的访问控制列表里（ACL），或者由主体以权能的形式持有，权能是不可伪造的访问资源集合的密钥。使用权能对于授予访问权限来说十分便利，但是很难收回。对ACL的改变能即刻生效，能收回以前的访问权限，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比特的密钥长度不足以防止强行攻击。DES的第3版实现了112比特密钥，该长度是安全的，但其他的算法（例如IDEA和AES）的运行速度更快而且提供了更高的安全性。

RSA是使用最为广泛的非对称加密方案。为了防范因数分解攻击，它应该使用768比特或更长的密钥。密钥（对称）算法比公钥（非对称）算法性能优越好几个数量级，因此公钥算法一般只用于混合协议（如TLS）中，例如在TLS中建立安全通道后，就可以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，它为许多实际的系统奠定了

基础。Kerberos是一个设计优良的用于在单个组织中进行用户认证和服务保护的方案。Kerberos基于Needham-Schroeder协议和对称密码学。TLS是广泛运用于电子商务中的安全协议。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。有关IEEE 802.11 WiFi安全协议的缺陷的讨论，使我们对安全设计中可能遇到的困难有了客观的认识。

练习

7.1 描述你所在机构的一些物理安全策略，参照一个计算机化的门锁系统中实现的方式来表达。（第266页）

7.2 举例说明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的情形。针对每种攻击形式电子邮件应如何采取相应的保护措施提出建议。（第268页） 320

7.3 公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。（第275页，第313页）

7.4 PGP被通常应用于安全电子邮件通信。在保证私密性和真实性的前提下，描述两个用户交换电子邮件消息前，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）。（第295页，第304页）

7.5 如何使用PGP或其他类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。（第304页，4.5节）

7.6 图7-7～图7-9中给出的TEA对称加密算法的实现不可在所有的机器间移植，试解释原因。如何使一个用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？（第290页）

7.7 修改图7-10中的TEA应用程序以使用密码块链接（CBC）。（第287页，第290页）

7.8 根据图7-9中程序，构建一个流密码的应用程序。（第288页，第290页）

7.9 试估计使用一个2000MIPS（每秒兆指令）的工作站，通过强行攻击破解一个56比特DES密钥需要的时间，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比特明文的时间（见图7-13）。对于一个128比特IDEA密钥进行同样的计算。推测如果使用一个200 000 MIPS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。（第291页）

7.10 在带有密钥的Needhan-Shroeder认证协议中，试解释为什么下面的消息5的版本是不安全的：

$$A \rightarrow B: \quad \{N_B\}_{K_{AB}}$$

（第306页）

7.11 回顾在讨论802.11 WEP协议设计时的解决办法，大致给出每种解决办法的实现方法，以及遇到的困难。（第317页） 321

第8章 分布式文件系统

分布式文件系统使程序可以像对本地文件那样对远程文件进行存储和访问，允许用户访问网络中的任一计算机上的文件。访问存储在服务器上的文件时应该能获得与访问本地磁盘文件类似的性能和可靠性。

在本章中，我们将给出文件系统的一个简单体系结构，并且介绍两种已被广泛使用20多年的分布式文件系统：

- Sun网络文件系统（NFS）
- Andrew文件系统（AFS）

这两个实例都模拟了UNIX文件系统接口，但它们具有不同的可扩展性和容错能力，以及同UNIX中的单拷贝文件修改语义的差异程度。

我们还将回顾一些相关的文件系统，它们采用了新的磁盘数据组织模式、高性能的多服务器访问、容错和可伸缩性的文件系统。书中的其他地方还将介绍其他类型的分布式存储系统，其中包括点对点存储系统（第10章）、复制文件系统（第15章）和多媒体数据服务器（第17章）。

322
~
323

8.1 简介

在第1章和第2章中，我们已经说明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式资源共享的一个最重要的方面。共享数据机制有许多种形式，我们将在本书中的相关部分分别介绍。Web服务器提供了一种严格的数据共享，其中客户可以通过因特网访问存储在服务器本地的文件。但是，通过Web服务器获得的数据是由服务器端或分布于本地网中的文件系统来管理和更新的。大规模广域可读写文件存储系统会产生负载平衡、可靠性、可用性和安全性问题，将在第10章介绍的对等网络文件存储系统的目标就是解决这些问题。第15章将重点讨论复制存储系统，它适合于需要对存储在系统上的数据进行可靠访问的应用，而系统中单独的主机不能保证可用性。在第17章中，我们将介绍一种媒体服务器，它用来满足大量用户实时的视频数据流传输。

局域网和企业内部网中的共享需求产生了一种不同类型的服务，它能够为客户端提供各种类型的程序和数据的存储持久性，以及更新数据的分布一致性。本章的主要目的是讨论基本分布式文件系统的体系结构和实现。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，也不提供多媒体数据流所需的宽带和实时保证——这些需求会在后面的章节中讨论。基本分布式文件系统为企业内部网上的有组织计算提供了必要支持。

文件系统最初是为集中式计算机系统和台式机开发的，它作为一种操作系统设施提供方便的磁盘存储的程序接口。后来，它们加入了访问控制和文件锁机制以实现数据和程序的共享。在企业内部网中，分布式文件系统以文件形式支持信息共享，并以持久存储的形式支持硬件资源共享等。一个设计良好的文件服务提供与访问局域文件性能和可靠性相似甚至更好的分布式文件访问。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中使用的更有效的共享永久存储。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]也开始被使用了。

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，程序可以像对待本地文件一样存储和访问远程文件。在几台服务器上集中存储文件可以减少本地磁盘存储，更重要的是可以使对组织机构拥有的持久数据的归档和管理更有效率。对于名字服务、用户
324 认证服务和打印服务等其他服务，当它们可以调用文件服务满足它们的持久存储需求时，它们可以更容易实现。Web服务器依赖于文件系统来存储其网页。在一个可以操纵Web服务器通过企业内部网进行外部和内部访问的机构中，而Web服务器经常从本地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永久存储和分布。一种实现方法是序列化对象（按4.3.2节描述的方式），并使用文件存储和检索序列化对象。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，但它们都不能保证对象的持久性，也不保证对分布式对象的复制。

图8-1概述了不同类型的存储系统。除了已经提到的存储系统，表中还包括了分布式共享内存（DSM）系统和持久对象存储，第18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，实现了对共享内存的模拟。它不一定要提供自动持久性。持久对象存储已在第5章介绍过了，其目标是为分布式共享状态提供持久性。此类例子有CORBA的持久状态服务（见第20章）和Java的持久性扩充[Jordan 1996 java.sun.com VIII]。一些研究项目已经开发出了支持自动复制和对象持久存储的平台（例如，PerDis[Ferreira et al. 2000]和Khazana[Carter et al. 1998]）。点对点存储系统提供了更大的伸缩性，以支持比本章介绍的系统大很多的客户负载。但是，它们为了提供安全访问控制和可更新复本间的一致性而付出了高额的性能代价。

	共享		分布式持久性	维护缓存/复本	例子一致性
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 (DSM，第18章)
远程对象（RMI/ORB）	✓	×	×	1	CORBA
持久对象存储	✓	✓	×	1	CORBA 持久状态服务
持久分布式对象存储	✓	✓	✓	2	OceanStore（第10章）

一致性类型：1：严格的单份复制，✓：弱保证，2：非常弱的保证。

325 图8-1 存储系统以及它们的性质

其中，一致性这一列表示当数据进行更新时是否有一种机制来维护其数据的多个拷贝之间的一致性。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”表示一个拷贝的一致性）——在更新后，程序不能发现存储数据与其缓存拷贝之间的任何区别。使用分布式副本时，很难达到严格的一致性。像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拷贝缓存起来，并且采用一种特定的一致性机制来维持一近似的致性。这在图8-1的“一致性”列中用“✓”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。

Web使用客户机上的缓存和由用户组织维护的代理服务器上的缓存。在Web代理和客户机缓存上的拷贝和原服务器中数据的一致性只能由用户行为来维持。当原服务器中的网页更新时并不通知客户；他们必须进行检查才能保持他们的本地拷贝为最新版本。在网页浏览中，这就能够满足要求了，但它不能支持像共享分布式白板这样的协作式应用程序的开发。第18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方法的差别相当大。CORBA和持久Java方案只维护持久对象的单一拷贝，访问这些对象需要使用远程调用，所以唯一的一致性问题是内存活动拷贝和磁盘上对象的持久拷贝之间的一致性，这对远程用户是不可见的。前面提到的PerDiS和Khazana项目维护缓存的对象副本，并采用了相当完备的一致性机制来产生与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及分布相关的问题之后，我们现在回到本章的主题——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。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节将介绍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——它是一种被广泛使用的系统，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最近的进展。

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形。本书后面的章将会介绍几个更先进的系统。第15章将介绍Coda系统，它是一种分布式文件系统，为了维持其可靠性、可用性和断链工作，它维护文件的多个持久拷贝。在第15章还将介绍一种分布式数据管理系统Bayou，它为了实现高可靠性，提供了副本的弱一致性形式。第17章将介绍Tiger视频文件服务器，它的目的是为大量的客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。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以及存储布局的细节。文件存储在磁盘或其他稳定的存储介质上。

文件包括*数据*和*属性*。其中，数据部分包括一系列的数据项（通常是8位），读和写操作可访问这些数据项的任何部分。属性部分用一个记录表示，其中包括文件长度、时间戳、文件类型、拥有者身份和访问控制列表等信息。图8-2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。

文件长度
创建时间戳
读时间戳
写时间戳
属性时间戳
引用计数
拥有者
文件类型
访问控制列表

图8-2 文件属性记录结构

文件系统用来存储和管理大量的文件，它具有创建、命名和删除文件的功能。应用目录系统可以为文件命名提供帮助。*目录*通常是一种特殊类型的文件，它提供从文本名字到内部文件标识符的映射。目录可以包括其他目录的名字，这样就形成了一种层次化的文件命名方案，UNIX和其他一些操作系统使用的是多部分组成的*路径名*。文件系统还负责控制对文件的访问，并根据用户授权和其请求的访问类型（读、更新、执行及其他操作）限制对文件的访问。

目录模块：	将文件名和文件ID关联
文件模块：	将文件ID和特定文件关联
访问控制模块：	检查操作请求是否许可
文件访问模块：	读或写文件数据或属性
块模块：	访问和分配磁盘块
设备模块：	磁盘I/O和缓冲

图8-3 文件系统模块

*元数据*这一术语是指文件系统用于管理文件而存储的所有额外信息。它包括文件属性，目录和其他文件系统使用的持久信息。

图8-3给出了传统操作系统中非分布式文件系统的实现所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，可能需要附加组件来处理客户-服务器通信、分布式命名以及文件定位。

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像C标准输入/输出库或Java文件类这样的库进程来访问这些

操作。这里我们给出的原语暗示了文件服务希望支持的操作，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)	用给定名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 包括read、write 或read、write两者
status = close(filedes)	关闭已打开的filedes文件
count = read(filedes, buffer, n)	从被filedes引用的文件中传输n字节给buffer
count = write(filedes, buffer, n)	从buffer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)	将读写指针移动指定的位移（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buffer中

图8-4 UNIX文件系统操作

326 ~ 328

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，文件状态信息是被存储在文件系统中的。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操作指示文件位置。

文件系统还负责文件的访问控制。在UNIX这样的本地文件系统中，当文件被打开时，系统就会进行访问控制。它在访问控制表中检查用户的权限，并将权限与在open系统调用中请求访问的模式做比较。如果权限与其模式匹配，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过程中，发现了许多分布式服务设计的需求和潜在的缺陷。最初，分布式文件系统只提供访问透明性和位置透明性，然而在后续的开发过程中，出现了性能、可伸缩性、并发控制、容错和安全需求，并且这些需求在开发中都得到了满足。我们将在后面的小节中讨论这些需求以及相关的需求。

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满足1.4.7节定义的分布式系统的透明性需求，其设计还必须平衡灵活性、可伸缩性、软件的复杂性和性能之间的关系。下列透明性是当前文件服务能够部分解决或完全解决的：

访问透明性：客户程序应该不了解文件的分布性。用户通过一组文件操作来访问本地或远程文件。操作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命名空间。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名字空间。

移动透明性：当文件被移动时，客户程序和客户节点上的系统管理表都不必进行修改。它们支持文件的移动性——多个文件或文件卷可以被系统管理者移动或自动移动。

性能透明性：当服务负载在一个特定范围内变化时，客户程序应该可以得到满意的性能。

伸缩透明性：文件服务可以不断扩充，以满足负载和网络规模增长的需要。

并发文件更新 客户改变文件的操作不应该影响其他客户同时进行的访问和改变同一文件的操作。这就是众所周知的并发控制问题，第13章会对此做详细讨论。许多应用程序都需要对共享

信息的访问进行并发控制，其实现技术也为大家所熟知，但其开销比较大。当前大多数文件服务都遵循现代UNIX标准，提供建议性的或强制性的文件级或记录级加锁。 329

文件复制 在支持文件复制的文件服务中，一个文件可以表示为其内容在不同位置的多个拷贝。这样做有两个好处——它允许当客户端访问相同的文件集合时多个服务器分担文件服务的负载，改善服务的伸缩性，同时改善容错性能，因为当一个文件损坏时，客户可以访问另一台具有此文件拷贝的服务器。少数文件服务支持完全的复制，但大部分文件服务支持文件缓存或本地部分文件复制（这是一种受限的复制形式）。关于数据复制的讨论详见第15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同样的客户和服务器软件。这一需求是开放性的一个重要方面。

容错 在分布式系统中，文件服务的中心角色决定了在客户和服务器出现故障时服务能继续使用是非常重要的。幸运的是，为一个简单的服务器设计一个中等的容错设计是比较容易的。为了应付暂时的通信故障，容错设计可以基于最多一次的调用语义（参见第5.2.4节）。而在按幂等操作设计的服务器协议中，容错设计可以使用更简单的最少一次语义，以保证重复的请求不会导致对文件的无效更新。服务器可以是无状态的，这样它可以重新启动，而且服务在发生故障后被恢复时，它不需要恢复以前的状态。文件复制可以实现对连接中断和服务器故障的容错，相比前面的情况而言这个目标很难达到，我们会在第15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个拷贝更新的语义。它提供了一个对文件进行并发访问的模型，即当多个进程并发访问或修改文件时，它们只看到仅有一个文件拷贝存在。当文件在不同的地点被复制或被缓存时，一个拷贝的被修改之处要传播到所有拷贝，这之间会有不可避免的延迟，这种情况可能会导致在一定程序上偏离单个拷贝语义。

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，客户的请求需要加以认证，于是服务器上的访问控制要基于正确的用户身份，同时还需要用数字签名和对机密数据加密（可选）机制来保护请求和应答消息。我们将在案例的描述中讨论这些需求的影响。

效率 分布式文件系统应该提供至少和传统的文件系统相同的能力，并且它还应满足一定的性能要求。Birrell和Needham[1980]对他们的Cambridge文件服务器（CFS）的设计目标的描述如下：

> 为了共享一个昂贵的资源（也就是硬盘），我们希望拥有一个简单、低级别的文件服务器。这样，我们就可以自由地设计适合特定客户的文件系统，但同时我们也希望拥有可以被客户共享的高级别的系统。

磁盘存储费用的降低减弱了效率的重要性，但不同客户仍然有不同需求，并且它能用上述的模块化体系结构加以解决。 330

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。分布式文件系统应提供在性能和可靠性方面能和本地文件系统比拟的、甚至更好的服务。它必须便于管理，能提供相应的操作和工具，使得系统管理员能方便地安装和操作系统。

8.1.3 实例研究

我们已经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，我们将它作为介绍性的例子。我们将详细地描述Sun网络文件系统，描述我们更为简单的抽象模型，以阐明它的体系结构。然后，我们将介绍Andrew文件系统，它采用不同的方法获得可伸缩性并保持一致性。

文件服务体系结构 这一抽象体系结构模型同时支持NFS和AFS。它基于三个模块间的责任划分——为应用程序模拟传统文件系统接口的客户模块、为客户提供目录和文件操作的服务器模块。这种体系结构设计启用了服务器模块的无状态实现。

SUN NFS Sun Microsystem的网络文件系统（NFS）自1985年面世以后，已广泛应用于工业界和学术界。1984年，Sun Microsystems的工作人员承担了NFS的设计和开发[Sandberg et al. 1985；Sandberg 1987，Callaghan 1999]。尽管当时已经开发出一些分布式文件服务，并且已应用于学校和研究性实验室，但NFS是第一个设计成产品的文件服务。NFS的设计和实现在技术上和商业上获得了巨大成功。

为了将NFS推广为一个标准，Sun公司公开了NFS主要的接口定义[Sun 1989]，允许其他供货商来产生实现，同时通过授权的方式允许其他计算机供货商获得参考实现的源代码。现在，NFS被许多供货商支持，同时定义在RFC 1813[Callaghan et al. 1995]的NFS协议（版本3）成为一个因特网标准。Callaghan关于NFS的书[Callaghan 1999]是关于NFS的设计和开发以及相关问题的一个极好的参考。

NFS为运行在UNIX 和其他系统上的客户程序提供对远程文件的透明访问。客户-服务器的关系是对称的：NFS网络上的每一台计算机既可以是客户，也可以是服务器，同时在每一台机器上的文件可以被其他机器远程访问。当输出自己的文件时，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专用服务器，而将其他机器作为工作站。

NFS的一个重要目标是对硬件和操作系统异构性实现高层支持。NFS的设计是独立于操作系统
331 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，包括各种版本的Windows、Mac OS、Linux和几乎所有其他版本的UNIX。有一些供货商在高性能多处理器主机上开发了NFS实现，它们被广泛用于满足具有许多并发用户的企业内部网的存储需要。

Andrew文件系统 Andrew文件系统是Carnegie Mellon大学（CMU）开发的一个分布式计算环境，它被作为校园计算和信息系统[Morris et al. 1986]。Andrew文件系统（以后简称为AFS）的设计反映了通过减少客户-服务器通信来支持大规模共享信息这一意图。它通过在客户和服务器之间传输整个文件，并在客户机中缓存文件直到服务器收到一个更新的版本的方式来实现这一意图。在介绍过Satyanarayanan[1989a；1989b]之后，我们会介绍AFS-2，这是AFS第一个“产品”级的实现。关于AFS更多最新的介绍可以在Campbell[1997]和[Linux AFS]中找到。

AFS最初在CMU运行BSD UNIX和Mach操作系统的工作站和服务器网络中实现，然后，它的商业和公用领域版本也相继实现。最近，在Linux操作系统[Linux AFS]上也可以使用AFS的公用领域实现。AFS已成为开放软件基金会（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，我们将在8.5节介绍这一点。

8.2 文件服务体系结构

为了清晰地划分文件访问问题的关注点，我们将文件系统的结构化分成三个组件——平面文件服务、目录服务和客户端模块。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将接口开放，供客户程序使用，它们同时和RPC接口一起提供了访问文件的操作。客户模块提供了同传统文件系统相似的关于文件操作的一个程序接口。设计的开放性体现在可以用不同的客户模块实现不同的程序接口，从而模拟不同操作系统的文件操作并根据不同的客户和服务器硬件配置优化性能。

模块之间的职责划分如下：

平面文件服务 平面文件服务注重实现在文件内容上的操作。唯一文件标识符（UFID）用于在所有平面文件服务操作的请求中标识文件。文件服务和目录服务的职责划分是基于UFID的使用。UFID是一长串比特，每个文件的UFID在分布式系统的所有文件中是唯一的。当平面文件服务接收到一个创建文件的请求时，它生成一个新的UFID并将此UFID返回给请求者。

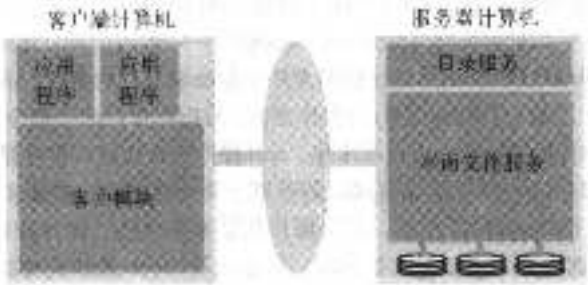


图8-5 文件服务体系结构

目录服务 目录服务提供文件的文本名字到UFID的映射。客户可以通过对目录服务引用文本名字来获得文件的UFID。目录服务提供生成目录、为目录增加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；它的目录文件存储在平面文件服务提供的文件中。当采用UNIX那样的层次化文件命名方案时，目录包含对其他目录的引用。

客户模块 客户模块运行在客户计算机上，它在一个应用程序接口下集成和扩展了平面文件服务和目录文件服务的操作。该程序接口可供客户计算机上的用户级程序使用。例如，在UNIX主机上，一个客户模块可以模拟UNIX所有文件操作的集合，并通过向目录服务适代地发出请求来解释UNIX的文件名的各个部分，从而模拟UNIX文件操作集。客户模块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，客户模块还可以通过在客户端缓存最近使用的文件块的方式来获得满意的性能。

平面文件服务接口 图8-6包含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所指的文件不在处理请求的服务器中，或访问权限不允许对此文件进行请求的操作时，FileId是无效的。如果FileId参数包含无效的UFID或用户没有足够的访问权限，那么除了Create接口之外的所有接口上的过程都会抛出异常。为清晰起见，这些异常从定义中省略了。

Read(FileId, i, n) → Data —改进BufPosition	如果1≤i≤Length(File)，则从文件中读取从i项开始的至多n n项的序列，并在Data中返回结果
Write(FileId, i, Data) —改进BufPosition	如果1≤i≤Length(File)+1，则从文件的i项开始 写入Data序列，必需要时扩展文件
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

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项开始顺序地拷贝n个数据项到Data中，然后将Data返回给客户。write操作拷贝Data中的数据序列到指定文件的第i项位置。它会替换原有文件在相应位置的内容，并在需要的时候扩展文件。

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操作使用户能访问属性记录。GetAttributes操作通常对每个能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中。属性记录的长度和时间戳的值不会受SetAttributes操作的影响。它们由平面文件服务单独管理。

与UNIX的比较：平面文件服务接口和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操作来构建模拟UNIX系统调用的客户模块是很容易的。

与UNIX接口相比，平面文件服务没有open和close操作——通过引用合适的UFID可以立刻访问文件。在我们的接口中，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-写指针指向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前移。seek操作用于使读写指针显式地重定位。

平面文件服务的接口与UNIX文件系统接口的差别主要对容错有一些影响：

可重复的操作：除了Create操作之外，其他操作都是*幂等级的*，即允许使用至少一次的RPC语义，客户可能在没有收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：接口适合用无状态服务器实现。无状态服务器可以发生故障后重启，它可以在不需要客户或服务器恢复任何状态的情况下继续操作。

332 ~ 334 UNIX文件操作既不是幂等级的，也与无状态实现的需求不一致。当文件被打开时，UNIX文件系统生成读-写指针，并且同访问控制检查的结果一起维持到文件关闭为止。UNIX的read或write操作不是幂等级的。如果一个操作意外重复时，读-写指针的自动前移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，系统应提供open和close操作，并且必须在相关文件打开后就一直维持读-写指针的值。通过消除读-写指针，我们消除了大多数文件服务中代表客户保留状态信息的需要。

访问控制 在UNIX文件系统中，系统会根据在open调用中请求的访问（读或写）模式来检查用户的访问权限（图8-4给出了UNIX文件系统的API），并且只有在用户拥有相应的权限时，才能打开文件。访问权限检查中使用的用户标识（UID）是用户认证登录的结果，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为止，并且在同一文件上进行后续操作时，系统不需要进行进一步检查。

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接口是访问文件的一个无保护的点。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今后的访问中使用时，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，系统执行一次访问检查，同时其结果以权能的形式编码（见第7.2.4节），它作为以后一系列请求的访问许可返回给客户。
- 在每一次客户请求时，都要提交用户标识，并且在每一次文件操作时，服务器都进行访问检查。

这两种方法都支持把服务器实现成无状态的，并且它们都已经应用在分布式系统中了。第二种方法更常用一些，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。这个问题可以利用第7章介绍的数字签名解决。Kerberos是一种有效的认证方案，它已经应用于NFS和AFS中。

在我们的抽象模型中，我们没有说明采用哪种方法实现访问控制。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包含目录服务的RPC接口的定义。目录服务的主要目的是提供将文本名字翻译为UFID的服务。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。每一个目录作为具有UFID的普通文件加以存储。因此，目录服务是文件服务的一个客户。

我们只定义了在单个目录上的操作。在每一个操作中，系统需要包含目录文件的UFID（在Dir参数中）。基本目录服务中的Lookup操作执行Name→UFID的转换。它可以供其他服务或客户模块
335 使用以完成更复杂的映射，如在UNIX中的层次化名字解释。像以前一样，定义中省略了访问权限不足可能引起的异常。

Lookup(Dir, Name) →FileId －抛出*NotFound*	在目录中找到文本名字，并返回相关的UFID。如果在目录中没有找到Name，便抛出异常
AddName(Dir, Name, FileId) －抛出*NameDuplicate*	如果目录中没有Name，则将（Name, File）加入到目录中，并更新其文件属性记录。如果在目录中已经有Name，便抛出异常
UnName(Dir, Name) －抛出NotFound	如果在目录中已经有Name，则包含Name的条目被删除。如果在目录中没有找到Name，便抛出异常
GetNames(Dir,Pattern) →NameSeq	返回在目录中所有与正则表达式Pattern匹配的文本名字

图8-7 目录服务操作

改变目录可采用两种操作：AddName和UnName。AddName给目录增加一个条目，并且在文件的属性记录中将引用计数字段增1。

UnName从目录中删除一个条目并将引用计数字段减一。当引用计数字段减少到零的时候，文件被删除。GetName使客户能够检查目录内容，同时它还实现像UNIX shell中的对文件名的模式匹配操作。它返回给定目录中存储的全部或部分名字。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。

GetName操作提供的模式匹配功能使得用户能够通过一个文件名中的部分字符的规约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，以及重复出现的字符/子串的符号组成的字符串表达式。

层次文件系统 类似UNIX提供的层次文件系统由组织成树型结构的目录组成。每一个目录包含文件和其他可以从此目录访问的目录的名字。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。树的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名字。UNIX的文件命名方案不完全是层次性的——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增加新的名字。

像UNIX这样的文件命名系统可以由使用了平面文件服务和目录服务的客户模块来实现。在目录的树型结构中，文件在叶节点上，而目录在树的其他节点上。树的根是一个具有“众所周知”的 336
UFID的目录。可以使用AddName操作和属性记录中的引用计数字段来为同一个文件取多个名字。

客户模块提供一个函数，用于实现对给定路径的文件查找其UFID的功能。该函数从根开始解析路径名，通过使用Lookup操作获得路径上每一个目录的UFID。

在层次化目录服务中，文件属性应该包括一个区别普通文件和目录的类型字段。可以根据它沿着路径确定名字的各个部分（除了最后一个部分）都是目录。

文件组 文件组是在一个位于给定服务器上的文件集合。一个服务器可能包含数个文件组，文件组可以在服务器之间移动，但文件不能改变它所属的组。在UNIX和大多数其他操作系统中使用的是一个相似的构造——文件系统。文件组最初被用来支持在计算机间移动存储在可移动介质上的文件集合。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在文件服务器上，同时它还支持用存储在几个服务器上的文件共同实现文件服务。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符，它能使每个客户计算机上的客户模块决定是否向包含相应文件组的服务器分发请求。

在分布式系统中，文件组标识符必须唯一。因为文件组可以被移动，同时最初分离的分布式系统也可以合并成一个系统，所以保证文件组在给定的系统中唯一的方法只能是：用一个确保全局唯一性的算法生成文件组标识符。例如，创建新的文件组时，可由创建新组的主机的32位的IP地址和一个根据日期生成的16位整数拼接而成的48位整数来形成唯一标识符。

文件组标识符：

32位	16位
IP 地址	日期

需要注意的是，IP地址不能用来定位文件组，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组标识和服务器之间的映射。

8.3 实例研究：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文件存储的远程过程调用集合。NFS协议与操作系统无关，但是它最初
337 是为在UNIX系统网络中使用而开发出的，我们将描述NFS协议（版本3）的UNIX实现。

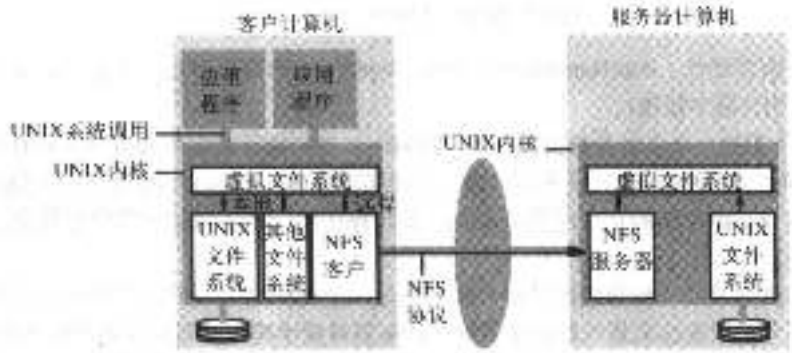


图8-8 NFS体系结构

NFS服务器模块驻留在每一个作为NFS服务器的计算机的内核上，客户模块将引用远程文件系统中的文件的请求翻译为NFS协议操作，并将它传输到保存相关文件系统的计算机的NFS服务器模块上。

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。它可以配置为使用UDP或使用TCP，NFS可以兼容这两种配置。该系统包括一个端口映射服务，它能使客户给定的主机名字绑定在服务上。RPC为NFS提供的接口是开放的，即任一进程都能向NFS服务器发送请求；如果请求是有效的并且包含有效的用户凭证，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能够提供访问透明性：用户程序可以对本地和远程文件发起访问而没有什么区别。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起来。

利用虚拟文件系统（VFS）模块可以实现上述集成，该模块已经加入到UNIX内核中，用于区分本地和远程文件，它还用于在NFS使用的独立于UNIX的文件标识符和在UNIX及其他文件系统中使用的内部文件标识符之间进行转换。另外，VFS保持对当前本地和远程均可用的文件系统的跟踪，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中使用的文件标识符称为文件句柄。文件句柄对客户是不透明的，它包含服务器区分
338 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节点数得来的。它在i节点中加入以下两个附加域（UNIX文件的i节点数是用来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：	文件系统标识符	文件的i节点数	i节点产生数

NFS采用UNIX的可安装文件集系统作为上一节定义的文件组单元（注意术语上的区别：文件集系统指的是在一个存储设备或分区上保存的文件集合，而文件系统指的是提供对文件访问的软件组件）。文件集系统标识符域是在创建每一个文件集系统后为其分配的一个唯一的数值（在

UNIX实现中，它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*节点数在文件被删除后就由其他文件重用，因此需要*i*节点产生数。在VFS对UNIX文件系统的扩展中，*i*节点产生数和文件一起存储，并在每次*i*节点被重用时（例如，在UNIX create系统调用中）加一。第一个文件句柄是在客户安装远程文件系统时获得的。文件句柄包含在*lookup*、*create*和*mkdir*等操作（见图8-9）的结果中，从服务器传送给客户，而在所有服务器操作的参数列表中，文件句柄都是从客户传到服务器端。

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，也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dirfh目录中的文件name的名字改为todirfh目录中的名字toname
link(newdirfh, newname, fh) →status	在目录newdirfh中创建一个条目newname，该条目指向文件或目录fh
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而是建立一个符号链接文件保存该string
readlink(fh) →string	返回与fh标识的符号链接文件关联的字符串
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，如果此目录非空，则操作失败
readdir(dirfh, cookie, count) →entries	从目录dirfh中返回目录条目的至多count字节。每一个条目包含一个文件名、文件句柄和一个指向下一个目录条目的不透明指针，该指针称为cookie。cookie用于在随后的readdir操作中从下一个目录条目中开始读。如果cookie的值是0，则从目录中第一个条目开始读
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，空闲块的数目等）

图8-9 NFS服务器操作（NFS v3协议，简化表示）

在虚拟文件系统层中，每一个已安装的文件系统有一个对应的VFS结构，并且每一个打开的文件有一个*v*节点。VFS结构将一个远程文件系统与安装VFS的本地目录联系起来。*v*节点包含一个指示此文件是本地文件还是远程文件的标识。如果文件是在本地，*v*节点包含对本地文件索引的引用（在UNIX实现中，是一个*i*节点）。如果是远程文件，它包含远程文件的文件句柄。

客户集成 在我们的体系结构模型中，NFS客户模块扮演的是客户模块的角色，提供适合传统应用程序使用的接口。但与我们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，并与UNIX内核集成在一起。它与内核集成到一起，而不是以客户进程运行时动态加载库的形式提供，这样会导致：

- 用户程序可以通过UNIX系统调用访问文件，而不需要重新编译或重新加载库。

- 一个客户端模块通过使用一个共享缓存存储最近使用的文件块（将在下面介绍），可以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。

在每一台客户机上，NFS客户模块与虚拟文件系统协同工作。它以一种和传统UNIX文件系统相似的方式操作，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在本地的内存中。它共享本地输入输出系统使用的缓冲区。但因为会有不同主机上的多个客户同时访问同一远程文件的情况，所以出现了新的且重要的缓存一致性问题。

访问控制和认证 与传统UNIX文件系统不同，NFS服务器是无状态的，并且不代表客户持续打开文件。因此在用户发出每一个新的文件请求时，服务器必须重新对比用户标识和文件访问许可属性来判断是否允许用户进行相应的访问。Sun RPC协议要求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协议的简介，其中没有给出这些附加参数，它们由RPC系统自动提供。

339 ~ 340

在最简单的形式下，访问控制机制有一个安全漏洞。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，而且每个进程可以作为一个客户向服务器发送访问和更新文件的请求。客户可以修改RPC调用以包括用户ID，从而防止该用户被假冒。这一安全漏洞可以通过在RPC协议中使用DES加密用户认证信息的方法来弥补。最近，Kerberos已经与Sun NFS集成起来，它为用户认证和安全性问题提供了功能更强、更全面的解决方案，我们将在下面介绍它。

NFS服务器接口 图8-9给出了由NFS服务器v3（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几乎等同于我们在平面文件服务模型（参见图8-6）中定义的Read、Write、GetAttributes和SetAttributes操作。在图8-9中定义的lookup操作和其他大部分目录操作与我们在目录服务模型（参见图8-7）中定义的操作类似。

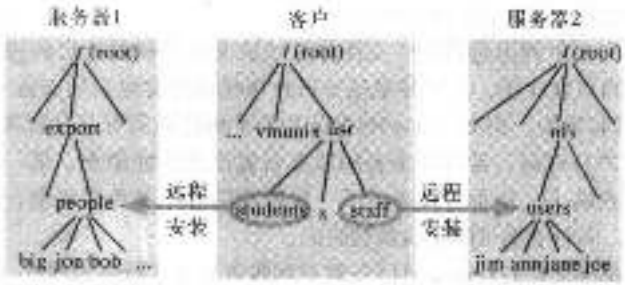
文件和目录操作集成在一个服务中。用一个create操作就能在目录中完成创建和插入文件名的操作，该操作取新文件的文件名和目标目录的文件句柄作为参数。目录上的其他NFS操作包括create、remove、rename、link、symlink、readlink、mkdir、rmdir、readdir和statfs。除了readdir（提供了一个读目录内容的独立于表示的方法）和statfs（给出远程文件系统的状态信息）之外，它们都在UNIX中有对应的操作。

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。每一个服务器上都有一个具有已知名字的文件（/etc/exports），它包含用于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关，该表用来指明哪些主机可以安装文件集系统。

客户使用一个修改过的UNIX mount命令，通过在其中指定远程主机名字、远程文件集系统的目录路径名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某个子树，使得客户可以安装任何一部分远程文件集系统。修改过的mount命令使用mount协议与远程主机上的安装服务进程进行通信。mount协议是一种RPC协议，它以一个给定的目录路径名作为参数，并返回指定目录的文件句柄，其前提是用户拥有访问相关文件集系统的权限。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。

341

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节点被安装到客户本地文件存储的students和staff节点上。这意味着运行在客户端的程序可以通过使用像/usr/students/jon和/usr/staff/ann这样的路径名来访问服务器1和服务器2上的文件。



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

远程文件集系统可通过硬安装和软安装两种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，进程被挂起直到请求完成。如果远程主机因为某种原因无法使用时，NFS客户模块会继续重复其请求直到该要求被满足为止。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为止，然后继续执行其工作，就好像没有出现故障一样。但是，如果相关文件集系统是采用软安装方式安装的，NFS客户模块会在数次重新请求失败后向用户级进程返回一个故障指示。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设施和应用并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，许多情况下只使用硬安装，结果造成NFS服务器如果在较长时段内不可用时，程序不能很好地恢复。

路径名翻译 每次使用open、create或stat系统调用时，UNIX文件系统一步步地将多部分文件路径名转换为i节点引用。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客户端的一个“安装点”——拥有多部分不同名字的目录可能驻留在不同服务器上的文件集系统中。所以要解析路径名，由客户以交互方式完成路径名的翻译。系统使用数个单独的对远程服务器的lookup请求将指向远程安装目录的名字的每一部分翻译为文件句柄。

lookup操作在给定的目录中查找路径名的一个部分，并返回相应的文件句柄和文件属性。在前一步返回的文件句柄作为下一个lookup的参数。由于文件句柄对NFS客户端代码是不透明的，所以虚拟文件系统负责将文件句柄解析为本地或远程目录，如果文件句柄引用了本地安装指针，虚拟文件系统还需要做一些间接转换。路径翻译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的本地引用来提高进程执行的效率。用户和程序通常仅访问一个或几个目录中的文件。 342

自动装载器 为了在客户引用一个“空”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，自动装载器的实现是在每一个客户计算机上作为一个用户级的UNIX进程来运行的。此后的版本（称为autofs）实现在Solaris和Linux的内核中。这里，我们介绍最初的版本。

自动装载器维护一张记录安装点（路径名）和对应的一个或多个NFS服务器的列表。在客户机上，它就像一个本地的NFS服务器一样。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，并且向表中对应的服务器发出“试探性”的请求。然后通过正常的安装服务，将第一个响应的服务器上的文件集系统安装到客户端上。被安装的文件集系统通过符号链接连接在安装点上，这样客户下一次访问时就不需要再向自动装载器发出请求。除非在数分钟内系统没有引用符号链接（这种情况下，自动装载器卸载了远程文件集系统），否则都可以正常的访问文件。

后来的内核实现方式以真实的安装取代了符号链接，这样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列出包含同一文件集系统或文件子树的拷贝的服务器，那么自动装载器可以实现一种简单的只读复制。对不经常改变但使用频繁的文件系统而言（如UNIX系统二进制文件），该机制是非常有用的。例如，/usr/lib目录及其子树的拷贝可以存储在多个服务器上。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消息，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程度的容错和负载平衡，因为第一个响应的服务器是正常工作着的，同时它也可能是负载较轻的。

服务器缓存 为了获得良好的性能，可以在客户和服务器上进行高速缓存，它是NFS实现的一个不可缺少的特征。

在传统的UNIX系统中，从磁盘上读取的文件页、目录和文件属性都保留在主存的*缓冲区缓存*上，直到其他页面要求占用该缓冲区的空间为止。如果一个进程对缓存中的页面发出一个读或写的请求，那么系统不需要再访问磁盘就可以完成此操作。*预先读*用于预测读访问，并将那些最近最常用的页面取入内存，而*延迟写*用于优化写操作的性能：当一个页面已经被改变时（因为一个写操作），只有在该缓冲区页将被其他页占用时才将该页面内容写到磁盘中。为了防止因系统崩溃引起的数据丢失，UNIX的sync操作每隔30s将改变的页面写到磁盘中。这些缓存技术在传统的
343 UNIX环境中都可行，因为由用户级进程发出的所有的读和写请求都被发送到在UNIX内核空间中实现的一个缓存上。该缓存保持的内容是最新的，同时文件操作不能绕过该缓存。

仅当NFS服务器被用于访问其他文件时，它才使用服务器上的缓存。使用服务器的缓存来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确信写操作的结果是持久性的，即使服务器崩溃时也是如此。在NFS协议版本3中，write操作为止提供了两种选项（没有在图8-9中标出）：

1）客户发出的write操作中的数据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，数据已经被持久地存储起来了。

2）write操作中的数据只存储在内存缓存中。当系统接收相关文件的commit操作时，它被写入磁盘中。仅当客户接收到相关文件的commit操作的应答时，客户才能肯定数据被持久地存储了。标准的NFS客户使用这种操作方式：在每次用于写而打开的文件关闭时，它发送一个commit。

commit是NFS协议版本3提供的一个附加操作，它用来解决在具有大量write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，对写透的需求是对第1章讨论的独立故障模式的一个实例——当服务器出现故障后，客户可以继续工作，同时应用程序在以前写操作的结果已经被提交到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故障一定会导致运行在相同计算机上的应用程序进程发生故障。

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操作的结果缓存起来。客户缓存可能导致在不同的客户节点上存在不同版本的文件或不同的文件内容，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拷贝的立即更新。要由客户用轮询服务器的方式来检查他们所拥有的缓存数据是否是最新的。

在使用缓存块之前，可以使用一种基于时间戳的方法对缓存块进行验证。缓存中的每个数据或元数据项被标记上两种时间戳：

- *Tc*是缓存条目上一次被验证的时间。
- *Tm*是服务器上一次修改文件块的时间。

设当前时间为*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，在缓存这个条目后，服务器上的数据就没有更新过），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：

$$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$$

选择t值时对一致性和效率进行了折衷。更新间隔过短会导致近似于单个拷贝的一致性，但因 344
为服务器要频繁地检查Tm_{server}，开销比较大。在Sun Solaris客户上，根据文件更新的频度，t可在3～30s之间取值。而对于目录，t可在30～60s之间取值，这说明发生目录并发更新的风险比较低。

每个文件的所有数据块都有一个Tm_{server}值，对于文件属性还有另一个值。因为NFS客户不知道文件是否被共享，所以验证过程要施加到所有被访问的文件。每次使用缓存项，系统就执行有效性检查：前一个有效性条件的判断可以不访问服务器就能进行。如果其判定结果为真，那么系统不需要检查第二个条件；如果结果为假，那么就要从服务器上获得当前的Tm_{server}值（对服务器应用getattr操作），并将它与本地的Tm_{client}进行比较。如果结果相同，那么此缓存项便被认为有效，并且其Tc值将被更新为当前时间。如果它们不相同，那么缓存的数据已在服务器上被更新过，此条目无效，这会产生一个获得服务器上相关数据的请求。

有几种方法可以减小对服务器进行getattr调用的数量：

- 当客户收到一个新的Tm_{server}值时，将该值应用于所有相关文件派生的缓存项。
- 将每一个文件操作的结果同当前文件属性一起发送，如果Tm_{server}值改变，客户便用它来更新缓存中与文件相关的条目。
- 采用自适应算法来设置更新间隔值t，对大多数文件而言，可以极大地减少调用数量。

验证过程不能保证提供和传统UNIX系统一样一致性，因为共享一个文件的所有客户并不是总能及时知道数据的更新，会存在两种时间延迟：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，由这个原因引发的麻烦已经引起人们的重视。

写操作以不同的方式被处理。当一个缓存的页面被修改后，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，修改的页面被更新到服务器中，如果使用bio-daemon（见下面的介绍），它的更新频率会更高。这并不能提供像服务器缓存一样的持久性保证，但它能够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，客户可以通过在每个客户端包含一个或多个bio-daemon进程实现这一点。（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。在执行写操作的情况下，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，相应的目录块会被立即发送。 345

bio-daemon进程改善了性能，确保客户模块不会因等待服务器端的read返回或者write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read操作会触发对相关服务器的同步请求，当相关的文件关闭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write操作的结果将被传输给服务器。

其他优化 Sun文件系统基于UNIX BSD快速文件系统，它使用8KB磁盘块，相对于以前的UNIX系统，它减少了用于顺序文件访问的文件系统调用。实现Sun RPC的UDP数据包扩充到9KB，这使得包含一个完整块的RPC调用可以作为一个参数在数据包中传送，当顺序读取文件时，还可以减小网络延迟的影响。NFSv3没有限制read或write操作处理的文件块的最大尺寸；当文件块的尺寸超过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。

正如上面所提到的，对于活动的文件，客户应该至少每隔3s更新在缓存的文件的状态信息。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服务器防止非授权访问和恶意攻击的工业标准。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（它由Kerberos的设计者完成）。

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来检查客户标识符的真实性。这意味着必须高度信任客户计算机及其NFS软件的真实性，而Kerberos和其他基于认证的安全系统的目的就是尽量减少需要信任的组件的范围。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是将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代价是相当大的，同时在每个工作站内核中都必须加入Kerberos客户库。

实际的系统采用了一种混合的方法，即安装用户的主文件集系统和根文件集系统时，给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
346 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）。

对于每个文件访问请求，NFS服务器检查用户标识符和发送者的地址，仅当这两者都与存储在服务器中的相关客户端的安装信息相符时，NFS服务器才允许访问。这种混合的方法仅需要很少的附加开销，而且如果在某一时刻每一个客户计算机上只有一个用户使用时，那么它对于大多数形式的攻击而言是安全的。MIT采用这种方法设计其系统，最近，NFS实现将Kerberos认证作为几种认证选项之一，并且建议在运行Kerberos服务器的机器上选择此选项。

性能 由Sandberg[1987]报告的早期性能图表说明：相对于访问存储在本地磁盘的文件而言，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，系统频繁地使用getattr调用。
- 因为写透是在服务器端使用的，这导致了write操作性能相对较差。

他同时指出，在典型的UNIX工作负载中，write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的版本并不包含上面所提到的commit机制，而当前NFS版本中的这一机制将明显提高写性能。他的结果也表明，lookup操作大约占服务器调用的50%。这是使用UNIX文件名语义所需的一步步的路径名翻译方法带来的结果。

Sun和其他NFS实现者使用LADDIS[Keith and Wittle 1993]这样的基准程序集进行正规的度量测试。现在和过去的一些测试结果可以在[www.spec.org]上找到，其中总结了不同厂商的NFS实现和不同硬件配置上的性能差别。基于PC硬件的单一CPU，以及专用的操作系统实现能够获得超过每秒12 000个服务器操作的吞吐量；而拥有多个磁盘和控制器的大规模多处理器配置能够获得每秒300 000个服务器操作的吞吐量。这些数字说明：不管是能够支持数以百计的软件工程师进行开发的传统UNIX，还是通过NFS服务器获取数据的Web服务器组，NFS可以为大多数企业内部网（无论它的规模和使用类型）的分布式存储需求提供有效的服务。

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名字空间，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一个新位
347 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大大提高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严格的单个拷贝文件更新语义。

下面是NFS其他的设计目标以及它们被实现的程度：

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。用户不需要修改现有的程序就能使这些应用程序正确地访问远程文件。

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名字空间来建立一个文件名空间。如果客户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节点必须导出该文件系统，并且客户在使用前必须远程安装该文件系统（参见图8-10）。远程安装的文件系统出现的客户名字层次上的地点由客户自己决定，因此NFS并没有强制实现一个网络范围的文件名字空间——每个客户看到的远程文件集系统都是本地定义的，同一远程文件在不同的客户上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名字空间。

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在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：已经发表的性能数据表明，NFS服务器可以以一种比较有效、高性价比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，单个服务器上的性能会提高。但达到处理极限时，必须加入新的服务器，同时在服务器间重新分配文件集系统。这种策略的效率受“热点”文件的限制，“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，分布式文件系统可提供更好的解决方案，例如可以使用支持复制可更新文件的分布式文件系统（如Coda，见第15章），或者像AFS这样通过缓存整个文件减少协议通信量的软件。我们将在8.5节介绍实现伸缩性的其他方法。

文件复制：只读文件可以复制到多个NFS服务器上，但NFS不支持具有更新的文件的复制。Sun网络信息服务（NIS）是一个可与NFS一起使用的服务，它支持以键-值对形式组织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（或者叫主拷贝模型，将在第15章讨论，该模型在每个场地上提供数据库的部分拷贝或全部拷贝）来管理分布式更新和对复制文件的访问。NIS为不经常变化的系统信息提供了一个共享库，并且它不要求所有的更新同步进行。 348

硬件和操作系统的异构性：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性本质确保在访问远程文件时客户发现的故障模式与访问本地文件时发生的故障模式类似。当服务器失效后，它提供的服务会挂起，直到服务器重启为止，一旦服务器重启，用户级客户进程就可以从服务中断的那一点继续执行，它不需要了解服务器出了什么故障（访问软安装的远程文件系统除外）。实际上，在大多数情况下系统使用的是硬安装，并且它阻止让应用程序处理服务器故障。

客户计算机或客户的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。

一致性：我们已经比较详细地描述了更新行为。它提供的语义近似于单个拷贝语义，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上时，对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]），用于认证和在读写数据时提供传输数据的私密性和安全性。许多安装还没有使用这些安全性机制，因此它们是不安全的。

效率：几个NFS实现的性能度量和NFS在大负载的环境的广泛使用，都说明NFS协议实现具有较高的效率。

8.4 实例研究：Andrew文件系统

和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。可以用正常的UNIX文件原语访问ASF文件，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，这样它使用NFS风格的文件句柄而不是i节点来引用文件，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
349 于其他分布式文件系统而言，AFS用来满足更多活动用户使用的需要。AFS实现可伸缩性的关键策略是在客户节点上缓存整个文件。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拷贝被传输到客户计算机上时，它被存储到本地磁盘的缓存中。该缓存包含该计算机最常用的数百个文件。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。文件的本地拷贝用于满足客户访问远程文件拷贝的open请求。

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，并且这一文件的当前拷贝不在本地缓存上时，AFS查找文件所在的服务器，并向其请求传输此文件的一个拷贝。
- 传输来的文件拷贝存储在客户计算机的本地UNIX文件系统中。该文件拷贝被打开，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此文件拷贝上进行一系列read、write和其他操作。
- 当客户进程发出一个close系统调用时，如果本地的文件拷贝的内容已经改变，则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。客户本地磁盘上的拷贝一直被保留，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），本地缓存的拷贝可以在相当长的时间内保持有效——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，更新的文件拷贝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。
- 本地缓存可以获得每个工作站的磁盘空间上相当大的空间，比如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空间，可以保证在给定工作站上常规使用的文件存储在缓存里以便下次使用。
- 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地域性。这些假设是通过观察学术和其他环境中的一些典型的UNIX负载得到的[Satyanarayanan 1981；Ousterhout et al. 1985；Floyd 1986]。其中最重要的结果包括：

350 - 通常文件比较小，大多数文件小于10KB。
- 文件的读操作比写操作更常用（通常是6倍以上）。
- 顺序访问更常用，随机访问不常用。

- 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。
- 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。

上述观察结论可用于指导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。还有一种重要的文件类型，它不属于上述文件类型——数据库通常许多用户共享，并且频繁地更新。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，他们认为由于不同的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，造成设计一个分布式数据库（它也是一个分布式文件系统）是比较困难的。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的操作，但留下许多有关其实现的问题。其中最重要的问题包括：

- 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？
- 在工作站上如何为缓存文件分配存储空间？
- 当文件可能被多个客户更新时，AFS怎样保证缓存中的文件拷贝是最新的？

下面将回答这些问题。

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。它是运行在每个服务器计算机上的用户级UNIX进程，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。

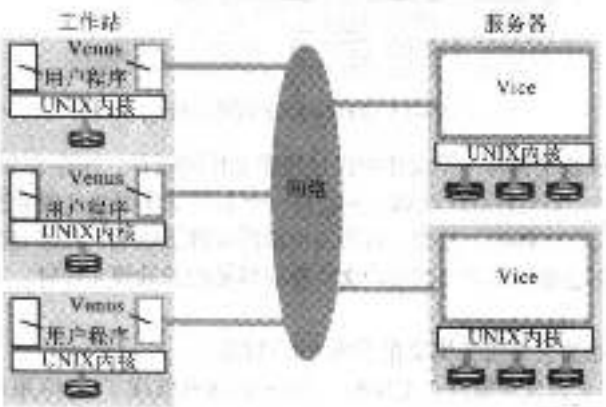


图8-11 在Andrew文件系统中的进程分布

可用于运行在工作站上的用户进程的文件是本地的或共享的。本地文件可作为普通的UNIX文件来处理，它们被存储在工作站磁盘上，只有本地用户进程可以访问它。共享文件存储在服务器上，工作站在本地磁盘上缓存它们的拷贝。图8-12显示了用户进程所看到的名字空间。它是一个传统的UNIX目录层次结构，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。本地文件仅作为临时文件(/tmp)，或者供工作站启动进程使用，其他标准的UNIX文件（例如那些通常在/bin，/lib目录下的文件）实际上是通过将本地文件目录中的文件符号链接到共享文件空间这种方式实现的。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他们的文件。

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那些指向共享名字空间中文件的调用，例如open，close和其他一些文件系统调用，并将它们传递给客户计算机上的Venus进程处理（参见图8-13）。对内核的另外一个修改是基于性能的考虑，将在后面介绍。

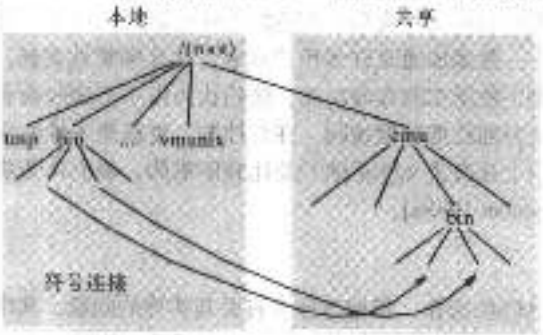


图8-12 AFS的客户所看到的文件名空间

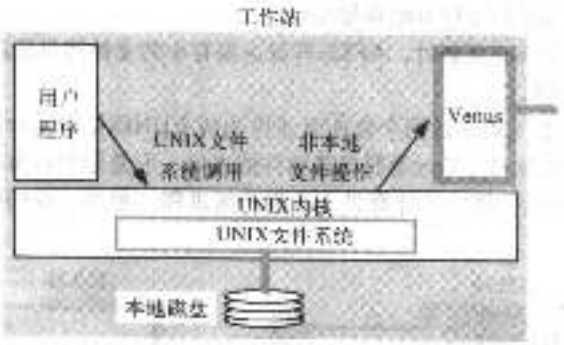


图8-13 AFS中系统调用拦截

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，保存共享空间中的文件拷贝。Venus进程管理这一缓存。当文件分区已满，并且有一个新的文件需要从服务器拷贝过来时，它将最近最少使用的文件从缓存中删除。通常，这些工作站缓存都足够大，可以容纳数百个一般大小的文件，这样，当客户缓存已经包含了当前用户文件和经常使用的系统文件时，工作站可以基本独立于Vice服务器工作。

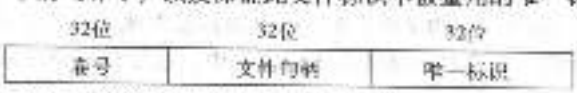
AFS和8.2节描述的抽象文件服务模型在下列方面很相似：

- 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目录结构。
- 共享文件空间中的每一个文件和目录是由类似于UFID的唯一的96位的文件标识符（fid）标识的。Venus进程将客户使用的文件路径名翻译为fid。

文件可以聚集成卷以便存储和移动。卷通常比UNIX的文件集系统小一些，文件集系统是NFS中的文件分组的单位。例如，每个用户的个人文件通常位于独立的卷中。其他卷用来存储系统二进制文件、文档和库代码。

351
,
353

fid的表示包括文件所在卷的卷号（类比：UFID中的文件组标识），用来标识卷中文件的NFS文件句柄（类比：UFID中的文件号）以及保证此文件标识不被重用的唯一标识：



用户程序使用传统的UNIX文件路径名来引用文件，但AFS在Venus和Vice进程之间通信中使

用fid。Vice服务器只接收用fid表示的文件请求。因此，Venus要将客户提供的路径名翻译为fid。这是由Venus通过一步步地在Vice服务器的文件目录中查找信息而实现的。

图8-14描述了当一个用户进程发出上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共享文件时，本地缓存中的此文件拷贝也被更新。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指共享文件空间内的一个文件，那么将这一请求传给Venus 打开本地文件并向应用程序返回其文件描述符	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，那么向管理包含此文件的卷的Vice服务器发送一个请求 在本地文件系统中放置文件的拷贝，并在本地缓存列表中输入本地名字，同时向UNIX返回其本地名字	→	向工作站传输一个文件拷贝以及一个回调承诺，记录该回调承诺
read(FileDescriptor, Buffer, Length)	在本地拷贝上执行一个正常的UNIX读操作			
write(FileDescriptor,Buffer,Length)	在本地拷贝上执行一个正常的UNIX写操作			
close(FileDescriptor)	关闭本地拷贝并通知Venus此文件已经被关闭	如果本地拷贝被修改，向管理此文件的Vice服务器发送此拷贝	→	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

354

8.4.2 缓存的一致性

当Vice为Venus进程提供文件拷贝时，它同时提供了一个回调承诺——由管理该文件的Vice服务器发送的一种标识，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，它会通知它发送过回调承诺的所有Venus进程，其方式是向每一个进程发送一个回调。回调是从服务器到Venus进程的一种远程过程调用。当Venus进程接收到回调时，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。如果标识的值是取消，那么必须从Vice服务器取得文件的最新拷贝；如果它的值是有效，那么Venus不需要引用Vice就可以打开和使用缓存中的文件拷贝。

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，在重启后第一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）没有和服务器通信了，那么回调必须被更新。这样可以处理可能的通信故障，因为通信故障可能导致回调信息的丢失。

相对于采用了和NFS相似的基于时间戳机制的原型（AFS-1）方法而言，这种维持缓存一致性的基于回调的机制可以提供更大的可伸缩性。在AFS-1中，拥有缓存文件拷贝的Venus进程进行open操作时会询问Vice进程，以便判定本地拷贝上的时间戳和服务器上的时间戳是否相符。基于回调的方法具有更大的可伸缩性，因为它只在文件被更新时才产生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个open操作时都产生客户和服务器的通信，即使本地有有效的拷贝。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，read操作比write操作多得多，回调机制使客户和服务器的交互量大大减少。

与AFS-1、NFS和我们的文件服务模型不同，AFS-2和其后的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包含发送过回调承诺的Venus进程列表。这一回调列表应在服务器故障时也被保留——它们被保存在服务器磁盘上，同时系统对它们使用原子性更新加以操作。

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）。

Fetch(fid)→attr,data	返回用fid标识的文件属性（状态）和文件内容（可选），同时记录一个回调承诺
Store(fid,attr,data)	更新指定文件的属性和文件内容（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)	为指定的文件或目录加锁，锁的模式可以是共享锁或排它锁。在30min后，没有解除的锁视为过期。
ReleaseLock(fid)	为指定的文件或目录解锁
RemoveCallback(fid)	通知服务器一个Venus进程已经将文件更新
BreakCallback(fid)	Vice服务器对Venus进程发出调用。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

更新语义 缓存一致性机制的目标是：在不对性能产生严重影响的情况下，近似实现单个拷贝文件语义。UNIX文件访问原语的单个拷贝语义的严格实现要求对每一个文件进行write操作时，其结果必须在发生进一步访问操作之前发送到所有在缓存中包含此文件的计算机上。在规模较大的系统中，这是不可行的，而回调承诺机制维护了一种对单个拷贝语义的较好的近似实现。

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拷贝的传播要保证满足以下条件：

- 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的当前值和在客户S上的值相同；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的值已经传播到服务器S上。

355
~
356

对AFS-2来说，对open操作的传播保证相对要弱一些，同时相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拷贝，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，这种情况就有可能发生。但系统设置了一个客户不知道文件最新版的最大时间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))

其中，latest(F,S,T)表示客户所见到的F的文件拷贝的过期时间不会超过Ts，lostCallback(S,T)表示在最近的Ts时间内从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缓存的文件F的拷贝是系统中的最新版本，或者回调信息被丢失（因为通信故障）而不得不使用已在缓存中的文件版本，二者必居其一；被缓存的文件F的拷贝的过期时间不会超过T秒。(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。)

为了实现这一目标，即提供大范围的与UNIX兼容的分布式文件服务，AFS并没有提供进一步的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以任意方式访问和更新本地拷贝。当文件被关闭后，文件拷贝返回到服务器，取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操作的更新结果外，其他更新结果通常会丢失（没有报错）。如果客户要实现并发，那么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，它们共享同一个缓存文件拷贝，并且依照UNIX方式（一块接一块）更新文件。

尽管更新语义随并发进程访问文件的位置不同而不同，并且和标准的UNIX文件系统提供的语义并不完全相同，但它已经足以使大部分已有的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上的用户级进程，并且服务器主机专用于提供AFS服务。AFS主机中的UNIX内核被修改过，这样Vice可以用文件句柄而不是UNIX文件描述符执行文件操作。这是AFS唯一需要的内核修改。如果Vice不维护任何客户状态（如文件描述符），这种修改是必须的。

位置数据库 每一个服务器包含一个位置数据库的拷贝，用于将卷名映射到服务器。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。 357

线程 Vice和Venus的实现使用非预先抢占性线程包，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存放在内存中，供Venus线程共享。

只读复制 经常执行读操作但很少被修改的文件卷，例如UNIX包含系统命令的/bin和/usr/bin目录和包含手册信息的/man目录，可以作为只读卷拷贝到多个服务器上。这样，系统中只存在一个读写拷贝，所有的更新都放在此拷贝上。在更新操作后，由一个显式的操作过程将改变传播到每个只读拷贝上。在位置数据库中，对于被复制的卷的位置数据库，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 当应用程序只需要读文件的一小部分时仍然将整个文件传输到客户端，这种方式显然是低效率的。AFS v3解决了这个问题，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的主要目标是实现可伸缩性，所以它特别关心在大量用户环境中的性能。Howard等人[1988]详细介绍了性能比较度量结果，它使用了专门的AFS基准测试，该基准测试后来被广泛应用于分布式文件系统的度量。不出所料，缓存整个文件和回调协议极大减少了服务器的负载。Satyanarayanan[1989a]解释说，在运行标准基准测试的具有8个客户节点的系统中，服务器的负载

是40%，而在运行同样基准测试的NFS系统中，服务器的负载是100%。Satyanarayanan将性能的提高归功于AFS使用回调来通知客户文件更新以减少服务器负载的方式，而在NFS中，系统采用超时机制检查缓存在客户端的页面的有效性。

广域支持 AFS v3支持多个管理单元，每一单元有自己的服务器、客户、系统管理员和用户。每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公司广泛部署了此类系统，并且发表了性能使用模式调查结果 [Spasojevic and Satyanarayanan 1996]。这一系统已安装在超过150个节点的超过1000台服务器上。调查结果表明，
358 在一个大约有200GB数据的32 000个文件卷的系统中，缓存命中率在96%～98%之间。

8.5 最新进展

NFS和AFS出现以后，分布式文件系统的设计又取得了一些进展。在这一节中，我们将介绍在改善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（参见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的质量的高伸缩性的体系结构（参见17.6节）。

NFS的改进 一些研究项目已经解决了单个拷贝的更新语义问题，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服务器能通知包含失效缓存条目的客户。下面将介绍其中的两方面工作。它们的结果说明：在可以容忍的复杂度和通信开销下，可以采用这些改进。

Sun公司和其他NFS开发者最近致力于使NFS服务器更易访问并使用在广域网上。尽管Web服务器支持的HTTP协议提供了有效的和高伸缩性的文件方法，以使整个文件可供因特网上每个用户使用，但它不适用于需要访问大文件或更新部分文件的应用程序。WebNFS的开发（将在下面介绍）使因特网内的任一地点的应用程序成为NFS服务器的客户成为可能（通过直接使用NFS协议而不是间接地通过内核模块的方式）。这种方式和合适的支持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具有大规模动态数据库的客户端。

达到单个拷贝更新语义：NFS的无状态服务器结构提高了NFS的健壮性，并使NFS更容易实现，但它偏离了精确的单个拷贝更新语义（不保证多个客户对同一文件并发写的结果，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通知客户文件发生了改变，这样就导致客户为了检查文件是否改变了而频繁地调用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的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。当本地用户级进程对服务器上的文件的执行打开操作时，客户模块必须发送一个open操作，open操作的参数指定了操作模式（读、写或两者都有）以及当前有打开的文件（进行读或写）的本地进程数。同样，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更新计数的close操作。服务器将这一数字和客户的IP地址和端口号一
359 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，它通过查找*打开文件表*找出所有打开同样文件的客户，并且将回调信息发送给这些客户，指导它们修改其缓存策略。如果该open操作指定的是写模式，并且有其他客户以写模式打开此文件时，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，使客户缓存中的文件拷贝失效。

对于一个以读模式的打开文件的客户，服务器会将回调消息发送给正在对此文件执行写操作的客户，通知它停止缓存（即使用严格的写透模式），并且它会通知所有读此文件的客户停止缓存

此文件（这样所有的本地读调用都会发出一个对服务器的请求）。

这种方法导致维持UNIX的单个拷贝更新语义的文件服务需要在服务器上记录一些与客户相关的信息。在处理对缓存文件的写操作方面，效率也有所提高。如果服务器在其非持久的存储器中保存与客户相关的状态，这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。这是基于一种“悲观”策略，即客户列表存储在磁盘上，并且很少被更新——可能它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中，但当该客户重启时，它会被删除。

当将Spritely NFS和NFS v2进行比较时，前者的性能有了一定程度的改进。这源于对写文件缓存的改进。NFS v3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个拷贝更新语义是可能的，虽然这样做客户和服务器模块比较复杂，并且需要一个恢复机制以便在服务器崩溃后能够恢复原有状态。

NQNFS：NQNFS（Not Quite NFS）系统[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状态，但它使用租借（参见5.2.6节）处理服务器崩溃后的恢复。服务器为客户持有打开文件的租借期设置一个上限。如果客户希望在超出此时间后继续持有该打开文件，它必须续租。当发生写请求时，系统使用与Spritely NFS相似的回调机制来通知客户刷新其缓存，但如果客户没有应答，服务器在响应新的写请求之前会一直等待，直到租借期满为止。

WebNFS：Web和Java applet的出现使NFS开发小组和其他人认识到：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RFCo2055和2056中描述[Callaghan 1996a, 1996b]）的目标是使Web浏览器、Java程序和其他应用程序与NFS服务器直接进行交互来访问文件，这些文件是使用公共文件句柄访问相对于公共根目录的文件而被“公开的”的。这种模式避免了安装服务和端口映射服务（见第5章）。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。该公共文件句柄有一个约定的值，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操作来查找请求中的多部分路径。 360

这样在较少的安装开销下，WebNFS使客户能够访问远程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，客户只需要读取公共文件，在这些情况下，认证选项可以关闭。在支持WebNFS的NFS服务器上，为了读一个文件的某一部分，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。

例如，一个天气服务可以在它的NFS服务器上公开一个文件，该文件包含一个需要经常更新的天气数据的数据库，它的URL为：

nfs://data.weather.gov/weatherdata/global.data

一个显示气象图的交互式WeatherMap客户可以用Java或其他支持WebNFS过程库的语言构建。客户只需要读取weatherdata/global.data文件中的部分信息就可以构建用户所需的气象图，而使用HTTP访问天气数据的类似应用程序需要将整个数据库传输给客户，或者需要使用专门服务器程序来获得所需的数据。

NFS第4版：在本书出版时，NFS协议的新版本正在开发中。RFC 2624[Shepler 1999]和Brent Callaghan的书[Callaghan 1999]中都描述了NFS第4版的目标。和WebNFS相似，它的目标是使NFS能适用于广域网和因特网的应用。它将包含WebNFS的特征，但它是个新的协议，有可能在WebNFS的基础上做更大的改进。（WebNFS对改变服务器有一定权限，它并没有在协议中加入新的操作。）

开发NFS v4的工作组希望利用在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 v4希望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。通过使用代理服务器（代理服务器的使用方式与在Web上的使用方式相似）来改进可伸缩性。

AFS的改进 我们已经提到过DCE/DFS，它是一种包含在开放软件基金会的分布式计算环境中的分布式文件系统[www.opengroup.org]，基于Andrew文件系统。DCE/DFS的设计超越了AFS，特别是在保证缓存一致性的方法方面。在AFS中，仅当服务器接收到对已经更新的文件的close操作请求时，系统才生成回调。DFS使用一种与Spritely NFS和NQNFS相似的策略在文件被更新时生成回调。为了更新一个文件，客户必须从服务器获得一个write标记，用于指定允许客户更新的文
361 件区域。在请求write标记后，具有同一文件拷贝（用于读）的客户会收到*撤回回调*。可使用其他类型的标识获得缓存文件属性和其他元数据的一致性。所有标记都有与之关联的生命期，在生命期满以后，客户必须续延其标识的生命期。

存储组织的改进 关于存储在磁盘上的文件数据的组织的研究有很大的进展。分布式文件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：

廉价磁盘的冗余阵列（RAID）：这是一种存储模式[Patterson et al. 1988, Chen et al. 1994]，其中数据被分解成固定大小的块，并存储在跨越多个磁盘的“条带”上，它们和冗余的错误更正代码存储在一起，更正代码用于在磁盘故障时完全重建数据块，系统可以继续操作数据。RAID也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）：和Spritely NFS一样，这项技术源于Berkeley Sprite分布式操作系统项目[Rosenblum and Ousterhout 1992]。他们注意到，在文件服务器中用于文件缓存的主存越多，相应缓存命中率越高，读文件操作的性能越好，但是写文件性能仍然没有提高。这源于将单个数据块写入磁盘以及更新元数据块（包含文件属性和指向文件中数据块的指针向量，例如*i*节点）操作的延迟。

LFS的解决方案是在内存积累若干写操作，然后将它们写到划分为大的、连续的、定长的段的磁盘上。这些段被称为*日志段*，因为数据和元数据块严格地按照被更新的顺序存储。一个日志段的大小为1MB或更大，存储在一个磁道上，它去掉了与写单个块相关的磁盘头延迟。被更新数据的最新拷贝和元数据块总是被写，因此要求维护一个指向*i*节点的动态映射。系统还要回收废弃的块空间，其方法是将“活”的块放置在一起以便为日志段的存储留出连续的空闲空间。后一种操作是比较复杂的，它由一个称为cleaner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cleaner算法。

尽管有这些额外的开销，但整个系统的性能改进还是比较显著的：Rosenblum和Ousterhout测量得到写的吞吐量高达可用带宽的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过程。Zebra文件系统[Hartman and Ousterhout]作为最初的LFS的后继成果，将结构化日志的写和分布式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节点的磁盘上。在写大文件时，其性能是NFS系统的4～5倍；在与小文件时，性能提高则不那么明显。

362 **新的设计方法** 高性能交换网络（例如ATM和高速交换以太网）的开发使研究人员注意研究如何在有许多节点的企业内部网上，以高伸缩性和高容错性的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写数据的职责相分离。下面将概述这方面的两项进展。

这些方法比我们在前面介绍的集中式服务器方法有更好的伸缩性。它们通常要求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节点间通信

(有些类似于一个“虚拟磁盘”的API)。因此它们的范围常常只限于单个本地网络。

xFS：美国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，即xFS[Anderson et al.1996]。有3个因素促成了该原型的实现：

1）快速交换局域网使本地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）不断增长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硬件，包括多个CPU、磁盘和网络控制器，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，一个中心式的服务器系统容易受单点故障的影响。

xFS是“无服务器”的，意味着它在单个文件的粒度上将文件服务器处理责任分散到本地网的可用的计算机上。存储责任独立于管理和其他服务责任进行分布：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磁盘上（从这个意义上说，它是第17章将描述的Tiger视频文件系统的先驱），它使用了和Zebra文件系统相似的结构化日志技术完成分散存储。

管理每个文件的责任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。文件标识符包含一个作为此管理器映射表索引的域，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来管理结构化日志文件存储和条带化磁盘存储，它们和其他结构化日志和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，这一原型还是不完善的——崩溃恢复还没有完全实现并且结构化日志的存储方案也缺少一个cleaner模块来恢复被废弃的日志和压缩文件所占据的空间。

对这一初步原型进行性能评估时，使用的是连接在高速网络上的32个单处理器和双处理器的Sun SPARC工作站。评测时对运行在32个工作站上的xFS和运行在双处理器Sun SPARC工作站上的NFS和AFS进行了比较。具有32个服务器的xFS的读写带宽是运行在一个双处理器上的NFS和AFS 363
的读写带宽的10倍左右。当使用标准的AFS基准测试时，xFS和NFS、AFS的性能差距并不明显。总之，结果表明：xFS的高度分布式处理和存储体系结构为分布式文件系统获得更好的可伸缩性提供了一个有希望的方向。

Frangipani：Frangipani是在数字系统研究中心（现在是Compaq系统研究中心）开发和部署的高可伸缩性的分布式系统[Thekkath et al. 1997]。它的目标和xFS十分相似，并且和xFS一样，其设计目的也是将持久存储责任和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。其底层由Petal分布式虚拟磁盘系统[Lee and Thekkath 1996]提供。

Petal为交换式局域网上的多个服务器磁盘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复本来应付大多数的硬件和软件错误，它还通过对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虚拟磁盘可以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和底层的磁盘驱动一样好。

Frangipani服务器模块运行在操作系统内核中。和xFS中一样，管理文件和相关任务的职责（包括对客户提供的文件锁服务）被动态地分配给主机，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（具有近似的单个拷贝语义）访问共享的可被更新的文件。数据以结构化日志和条带格式存储在Petal虚拟磁盘存储中。Petal减轻了Frangipani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的性能至少和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和本地文件系统相同甚至更好的性能。
- 当文件更新时，维护文件的多个客户拷贝的一致性。
- 在客户和服务器发生故障后进行恢复。
- 提高读写不同大小文件的吞吐量。
- 可伸缩性。

364

分布式文件系统在有组织的计算中被广泛使用，它们性能的提高是优化的目标。NFS包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技术领域的统治地位。

AFS显示了一种相对简单的体系结构的可行性，它使用服务器状态减小维护客户缓存一致性的开销。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结构化日志写操作，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较高的伸缩性，并且可提供跨越本地和广域网的优良性能，维护单个拷贝文件更新语义，并且能容错和从故障中恢复。未来的需求包括支持经常有断连操作的移动用户，支持自动重集成和服务质量保障，以便满足持久存储和传输多媒体数据流以及其他实时数据的需要。第15章和第17章将介绍这些需求的解决办法。

练习

8.1 为什么在平面文件服务或目录服务的接口中没有open操作或close操作。目录服务的lookup操作和UNIX的open操作有哪些区别？ (第334页～第336页)

8.2 请列出使用模型文件服务，客户模拟UNIX文件系统接口的方法。 (第334页～第336页)

8.3 写出一个PathLookup(Pathname, Dir)→UFID过程，该过程基于模型目录服务实现对UNIX式路径名的Lookup操作。 (第334页～第336页)

8.4 为什么UFID必须在多个可能的文件系统上保持唯一？这种唯一性是怎样保证的？ (第337页)

8.5 Sun NFS在何种程度上偏离了单个拷贝文件更新语义？请构造这样一个场景：两个共享一个文件的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的主机上时便会出现不一致性。 (第344页)

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？ (第338页)

8.7 NFS客户模块必须拥有哪些代表用户级进程的数据？ (第338页～第339页)

8.8 使用图8-9中的NFS RPC调用，分别给出在不使用和使用一个客户缓存情况下，UNIX的open()和read()系统调用的实现。 (第340页，第344页)

365

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 3通过使用加密弥补了这个安全漏洞。密钥是如何保密的？密钥的安全性足够吗？ (第340页，第346页)

8.10 在一个RPC调用访问一个硬安装的文件系统上的文件超时后，NFS客户模块并没有将控制返回到发出调用的用户级进程，为什么？ (第341页)

8.11 NFS的自动安装器是如何改进NFS的性能和可伸缩性的？ (第343页)

8.12 解析存储在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lookup调用？执行一步步翻译的原因是什么？ (第342页)

8.13 为了在基于NFS的文件系统上获得访问透明性，在客户计算机上的安装表的配置必须满足哪

些条件？ （第342页）

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？ （第351页）

8.15 将访问本地文件的UNIX更新语义和NFS及AFS的更新语义进行比较。在什么情况下，客户可以意识到其差异？ （第344页，第355页）

8.16 AFS是如何处理回调信息可能丢失这一风险的？ （第355页）

8.17 AFS设计的什么特点使得它比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？ （第347页，第358页，第362页）

366

第9章 名字服务

本章将名字服务作为一个独特的服务加以介绍。使用名字服务，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任何类型，并且可由不同的服务管理。例如，名字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。除名字服务外，我们还将介绍目录服务，它可以根据服务的属性寻找特定服务。

我们将以因特网域名服务为例来介绍名字服务的设计要点，如服务可以识别的名字空间的结构与管理、名字服务支持的操作等。

我们还将探讨名字服务的实现，其中涉及名字解析过程中的名字服务器导航、为提高性能与可用性对名字数据进行缓存与复制等。

本章包含两个实例研究：全局名字服务（GNS）与X.500目录服务（包括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，甚至用户等各种资源。命名在分布式系统设计中其实是一个非常基本的问题，尽管它极易被忽略。名字为通信与资源共享提供了便利。当要求计算机系统对某个资源进行操作时，就需要一个名字。例如，访问特定Web页面需要一个以URL形式表示的名字。只有在所有进程中一致地命名了计算机系统管理的特定资源，进程才能共享这些资源。同样，在分布式系统中，只有用户能够给出对方名字，双方才能互相通信，例如，电子地邮件址就是一种名字。

名字并不是识别对象的唯一方法，描述性的属性也可用于识别对象。有时候，一些客户并不知道它们寻找的实体的名称，却知道一些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实体），却只知道该服务具有的一些特征。

本章将介绍名字服务以及目录服务的相关概念。在分布式系统中，名字服务可向客户提供被命名对象的数据；而目录服务提供满足某个描述的对象的数据。我们将以域名服务（DNS）、GNS以及X.500作为研究实例，来描述设计与实现这些服务的方法。首先我们将讨论名字、属性等基本概念。

名字、地址及其他属性

任何请求访问一个资源的进程必须拥有该资源的名字或标识符。文件名（如/etc/passwd）、URL（如http://www.cdk4.net/）以及因特网域名（如dcs.qmw.ac.uk）都是我们可以阅读的名字。而标识符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文件句柄都是标识符的例子。选择标识符的一个重要指标是软件存储与查询标识的效率。

Needhan[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。而非纯粹的名字则包含被命名的对象的信息，特别地，它们会包含对象的位置信息。纯粹的名字在使用前必须被查找。与纯粹的名字完全相反的是对象的地址，该值标识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使用不同的因特网服务供应商而改变，因此邮件地址本身并不能永久地指代特定用户。

当一个名字被翻译成被其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

的目的是为了在对象上调用一个动作。名称与对象之间的关联通常称为绑定。一般而言，名字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 368
实体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：主机的IP地址、条目的类型（例如，引用的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名字服务与交易服务将在第20章介绍。名字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”常被看作另一种可用于查找的名字，或者它包含一个可查找的名字。必须查找IP地址来获得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解析成一个Web服务器的以太网地址，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。

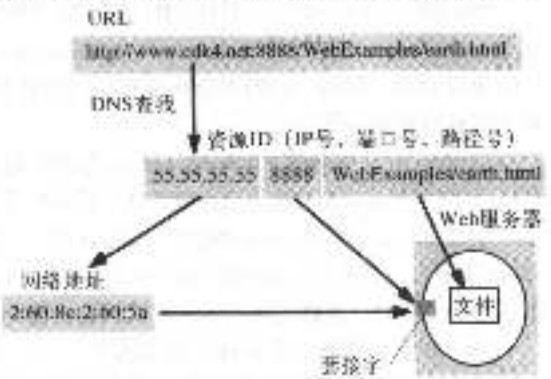


图9-1 使用组合命名域从URL访问资源

名字与服务 分布式系统使用的许多名字是特定的服务专用的。客户程序使用名字来请求特定服务以便在它管理的已命名对象或资源上执行某个操作。例如，当请求删除一个文件时，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，则要将该进程的标识符传送到进程管理服务。除客户基于共享对象通信的情况外，上述名字仅在管理命名对象的服务的上下文中使用。 369

在分布式系统中，名字也要能指称超出单个服务范畴的实体，这些实体主要包括用户（具有专有名、登录名、用户标识符以及电子邮件地址）、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，上述名字必须是人类能阅读与理解的，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。同时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，这些命名需求可能是全球性的。

统一资源标识符 统一资源标识符（Uniform Resource Identifiers，URI）[Berners Lee et al. 2005]是用来标识Web资源，以及其他因特网资源（如电子邮箱）的。它的重要目标是将资源以一致的方式标识出来，以便它们能被公共软件（如浏览器）处理。URI是“统一的”，它的语法结构整合了各种相对独立并且尚不明确的资源标识符类型（即URI方案）。还提供了处理全球名字空间方案的过程。统一的优点在于，它简化了引进新的标识符类型的过程，并且能够将已有的标识符类型应用到新的上下文中，而不会影响已有的应用。

例如，如果有人想发明一种“窗口部件”URI，则URI就可以成为一个窗口部件，必须要遵守全球URI语法规范，以及本地窗口部件标识符方案的规则。这样做，那些URI就能明确标识窗口部件资源，现有的并不会访问窗口部件的软件也可以正确处理窗口部件URI——例如，通过管理包含窗口部件的目录。下面是一个URI如何整合一个已有资源标识符的例子，即在已有的电话号码前增加机制名tel，并且将电话号码统一表示为如下的标准形式：tel: +1 -816 -555 -1212。这些tel URI

可以作为Web连接，而且当有人点击链接时，就会拨通相应的电话。

统一资源定位器：有些URI提供了资源的位置信息(例如一个DNS主机名和那台机器的路径名)，而另外一些URI则纯粹用作资源的名称。统一资源定位器（URL）作为资源位置的标识符，包括1.3节提到的“http”URL，例如http://www.cdk4.net/，它标识了主机www.cdk4.net上给定路径（“/”）的网页。另一个例子是“mailto”URL，如mailto:fred@flintstone.org标识了一个给定地址的邮箱。

URL是一种有效的用于访问资源的标识符，但是它也有下面的缺点：当一个资源被删除或者从一个网站移到另一个网站时，原来的URL就会成为一个指向该资源变动前所在网站的悬空链接。当用户点击这个悬空链接时，Web服务器或者会应答所访问资源不存在，或者更糟糕的是会返回给用户一个不同的资源，因为这个URL又被重新分配给了另一个资源。

统一资源名称：统一资源名称（URN）是URI的一种，它仅作为纯粹资源名，而不包含资源
370 定位符。例如，下面的URI：

mid:0E4FC272 -5C02 -11D9 -B115 -000A95B55BC8@hpl.hp.com

就是一个URN，它被标注在电子邮件的Message-Id域中，从而标识了该邮件消息。这个URI能将邮件消息区分开。但这个URI本身并没有提供任何有关该邮件地址的消息，如果需要邮件地址，则需要调用相应的查询操作。

以urn：开头的特殊的URI的子树是为URN而保留的，但并不是所有URN都必须以urn开头，例如上面例子中的URN就是以mid开头。以urn开头的URI都形如：urn:nameSpace:nameSpace-specificName。例如，urn:ISBN:0-201-62433-8标识以标准ISBN命名机制命名的编号为0-201-62433-8的书。又例如，urn:doi:10.555/music-pop-1234标识了依据数据对象标识方案[www.doi.org]，出版者为10.555，名称为music-pop-1234的出版物。

有一些*解析服务*（本章称为名字服务），例如用于处理URN的Handle系统可以通过数据对象标识[www.handle.net]获得资源属性，但是没有一种解析服务获得了广泛的应用。事实上，在Web和因特网搜索社区有关于独立分类的URN应该如何扩展的争论一直在继续。其中一派认为应该“保持URL不变”，换句话说，每一个人都应该保证URL所引用的资源持续有效。相反的观点则认为并不是每个人都要保证URL的有效性，因为这需要必要的资金来维持对域名和资源的管理。

9.2 名字服务和域名系统

一个名字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的绑定的集合。名字服务的主要操作是名字解析，即根据一个给定的名字查找相应的属性，我们将在9.2.2节中描述名字解析的实现。其他操作，如创建新的绑定、删除绑定、列出绑定的名称以及增删上下文都是名字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。开放性带来了下列需求：

- 一致性：不同服务管理的资源使用相同的命名方案会很方便，URL就是一个很好的例子。
- 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名字服务，管理域会使用完全不同的命名约定。

通用名字服务的需求 名字服务最初非常简单，因为它仅用来满足名字与单个管理域（如
371 LAN或WAN）中的地址绑定的需求。然而，网际互连以及分布式系统的不断扩展带来了更大规模的名字映射问题。

Grapevine[Birrell et al. 1982]是最早的可扩展多域名字服务之一。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]开发的全局名字服务是Grapevine的改进。GNS具有更宏大的目标，包括：

- 处理任意多的名字，为任意多的管理组织提供服务：例如，系统应该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：很多系统依赖名字服务，名字服务一旦崩溃，系统就无法工作。
- 故障隔离：局部故障不会带来整个服务的崩溃。
- 允许不信任：一个大型的开放系统很难使每一个组件都被系统中所有客户信任。

例如，全局名字服务 [van Steen et al. 1998]（包括电子邮件用户地址或文档）和Handle系统[www.handle.net]，它们都关注于提高系统在大规模对象条件下的可伸缩性。第3章中介绍的因特网域名系统（DNS）命名了因特网上的对象（实际上就是计算机）。为提供满意的服务，它极大地依赖于名字数据的复制与缓存。在DNS以及其他名字服务的设计中，对于缓存中的文件副本，假设无需严格保证缓存一致性。原因在于，修改并不是那么频繁，同时使用一个过期的名字翻译结果通常可被客户程序探测到。

本节将用DNS为例子，讨论名字服务设计的主要问题，然后给出有关DNS的实例研究。

9.2.1 名字空间

名字空间是一个服务所能识别的所有有效名字的集合，所谓有效意味着服务将试图查找之，即使该名字并不对应于任何对象，即未被绑定。名字空间需要语法定义。例如，名字Two不可能是一个UNIX进程的名字，然而数字2却可以是。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，表示它们在层次化名字空间中的位置，或者可以从一组平面的数字标识符或符号标识符集合中选择名字。层次化名字空间的最大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 372
相同的名字在不同的上下文中可以有不同的含义。对于文件系统，每个目录都代表一个上下文。因此，/etc/passwd是一个具有两个部分的层次化的名字。首先，etc相对于上下文“/”或根进行解析，而第二部分“/passwd”相对于上下文“/etc”被解析。名字“/oldetc/passwd”可以有不同的含义，因为该名字的第二个部分在不同的上下文中解析。类似地，同样的名字“/oldetc/passwd”在两台不同的机器上，基于不同的上下文会解析到不同的文件上。

层次化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，那么平面名字空间也可以是无限的。层次化名字空间另一个好处是可由不同的人管理不同的上下文。

第1章介绍了http URL的结构。URL名字空间包括../images/figure1.jpg这样的相对名。在这种URL方案中，浏览器将路径名所对应的主机名与服务器目录，作为它所嵌入的文档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绝对文件名相似的字符串。DNS名的例子有：www.cdk4.net（一台计算机）、net、com和ac.uk（后三个是域名）。

DNS名字空间具有层次结构，即一个域名包括一个或多个字符串，这些字符串通常称为名字成分或标签，并且用分隔符“.”分隔开来。尽管为管理方便，DNS名字空间的根节点有时用“.”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“.”符号的非空可打印字符串。一般来说，名字的前缀（prefix）指的是包含零个或多个完整名字成分的名字的最初一部分。例如，在DNS中，www和www.cdk4都是www.cdk4.net的前缀。DNS名不区分大小写，因此，www.cdk4.net与WWW.CDK4.NET的含义相同。

DNS服务器不能识别相对名，所有的名字都基于全局的根节点。然而，在实际实现中，客户软件会维护一个域名表，在解析单部分域名时，会将该列表自动附加到单部分域名之后。例如，域cdk4.net中的名字www有可能指的是www.cdk4.net。在试图解析www时，客户软件将默认域名

cdk4.net附加在www后。如果解析失败，则附加其他默认域名。最后，将www作为绝对名解析。在这种情况下，一个操作就失败了。另外，具有多个部分的名字通常不做预处理，作为绝对名被送到DNS。

别名 遗憾的是，带一个或两个部分以上的名字在键入与记忆上都很不方便。一般说来，*别名*（alias）与UNIX风格的符号链接相似，允许用一个方便的名字替代复杂的名字。在DNS服务中使用别名的方法是，通过定义一个域名来表示另一个域名。提供别名的原因是为了保持透明性。例如，别名通常用于指定一个运行Web服务器或FTP服务器的机器名。名字www.example.net是
373 fred.example.net的别名。这样做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引用一个Web服务器，如果Web服务器被移动到另一台计算机，唯一要做的是更新DNS数据库中的别名。

命名域 *命名域*是一个仅有一个总的管理权利来管理该域中的名字分派问题的名字空间。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，一个域的名字是在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，net是一个包含了cdk4.net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（而其他域名则标识了计算机）。

域的管理可以被移交到子域中。域dcs.qmul.ac.uk，即英国Queen Mary College（伦敦大学）的计算机系可以包含任何该系想要的名字。但dcs.qmul.ac.uk域名本身需要得到学院权威机构（它管理着域qmul.ac.uk）的认同。类似地，qmul.ac.uk必须得到已注册的权威机构ac.uk的认同。

管理命名域，以及管理由名字服务使用的存储在权威名字服务器上的权威数据库，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存储在不同的名字服务器上，这些名字服务器由不同的权威机构管理。

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，某些名字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。而且，有些名字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。

合并：在UNIX与NFS的安装文件系统的实践（见8.3节）中，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完整的UNIX文件系统，这两个系统在两台分别名为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，而在blue上指的是另一个文件。合并两个文件系统的最简单的方法是：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，并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来合并名字空间，但这样做会带来向后兼容问题。修正兼容性问题又会给我们带来混合名字空间问题，在两台计算机的用户之间翻译旧的
374 名字也非常不方便。

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名字可以包含*接合点*（junction），接合点的概念与NFS与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.qmul.ac.uk/principals/Jean.Dollimore。名字的第一部分/.../dcs.qmul.ac.uk/标识了一个称为*单元*的上下文。下一个部分是一个接合点。例如，接合点principals是一个包含安全主体的上下文，在该上下文中可以查询名字的最后一个部分

Jean.Dollimore。类似地，在/.../dcs.qmul.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，查询名字的最后一个部分pub/reports/TR2000-99。接合点principals与files是异构名字空间的根，它们由异构名字服务实现。

定制：从上面嵌入NFS文件系统的例子中可以看出，在有些情况下，用户愿意构造自己的名字空间，而不是共享某个名字空间。文件系统的安装使用户可以导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，并且可以进行自治管理。然而，即使是同一个文件，如果从不同的计算机访问，也会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原则上可以绑定到采用奔腾计算机的x86二进制格式的程序上，也可以绑定到Mac OS X计算机的PowerPC二进制格式的程序上。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名字服务[Radia et al. 1993]提供了动态构造名字空间以及有选择地共享个人命名上下文的能力。与上面例子不同的是，甚至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第一类对象。例如，假设计算机red上的用户试图运行blue上的一个程序，该程序寻找/etc/passwd这样的文件路径，但是该路径被解析到red文件系统上的文件，而不是blue上的文件。在Spring中，可以通过将对red的本地命名上下文的引用传递给blue并将其作为程序的名字上下文来达到这一目的。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的一个新颖的特色（该特色也可在Spring中实现）是它的物理目录可以被排序并合并为一个逻辑目录。这样做的效果是，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，直到查询得到匹配的结果，然后可以返回相应的属性。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将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 375
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；随着更深层次的命名上下文以及派生名的输出，解析过程不断迭代。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，etc首先被送到上下文/，然后passwd被送到上下文/etc。

解析过程迭代特性的另一个实例是别名的使用。例如，当请求DNS服务器解析诸如www.dcs.qmul.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apricot.dcs.qmul.ac.uk)，然后这个域名被进一步解析产生一个IP地址。

通常，别名的使用可能会导致名字空间带有循环，在这种情况下，解析过程将永不终止。有两个解决方案，一是一旦到达解析次数的阈值，就放弃解析；第二个是让管理员禁止任何会导致循环的别名。

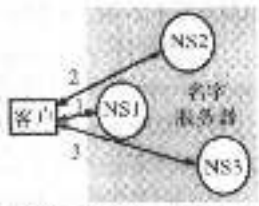
名字服务器与导航 诸如DNS这样的名字服务需要将数据存储在一个巨大的数据库中，并由一大群人访问，因此通常不会将所有的名字信息放在单个服务器上。这样的服务器会成为一个瓶颈以及故障的临界点。任何常用的名字服务都应该使用复制以提高可用性。我们将看到，DNS规定数据库的任何一个子集都必须复制到至少两个不会同时失效的服务器上。

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名字服务器上。尽管在某些情况下，一个名字服务器会存储多个域的数据，但数据根据域的不同被分区到不同服务器上却是事实。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

名字服务器存储更高的域（如yahoo.com、ac.uk）和根信息。

数据的分区意味着本地名字服务器若没有其他名字服务器的帮助，将无法回答所有的询问。例如，在dcs.qmul.ac.uk域中的名字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。

为解析一个名字，从超过一个名字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名字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Grapevine[birrell et al. 1982]。另一种方法（如在X.500中使用的）是在一个独立的进程中提供名字解析，而该进程可被该计算机上的所有客户进程共享。



为解析一个名字，客户反复求与名字服务器NS1-NS3联系

图9-2 迭代导航

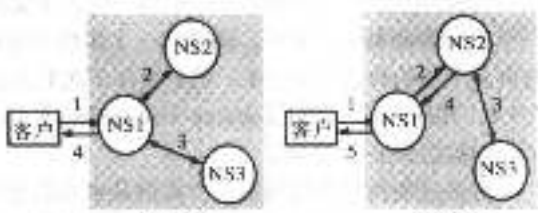
DNS支持迭代导航模型（参见图9-2）。在解析一个名字时，客户将该名字送到本地名字服务器，该服务器试图解析之。若本地名字服务器有这个名字，则立刻返回结果。如果没有这个名字，则它会建议另一个能提供帮助的服务器。在另一个服务器上继续解析，进行进一步的导航，直到该名字被定位或是发现并未与任何名字绑定为止。

由于DNS可以容纳数百万个域的条目，并且可以被大量的客户访问，因此，即使在根服务器被大量复制的情况下，所有的查询都从根服务器开始也是不可行的。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即可被满足。9.2.3节将详细描述DNS解析名字的方案。

NFS在解析文件名时，按每个部分进行解析，也使用了迭代导航（参见第8章）。这是因为在解析名字时，文件服务可能会遇到符号链接。符号链接必须在客户的文件系统名字空间中被解释，因为它可以指向另一个服务器目录中的文件，客户计算机必须确定该服务器是哪个，因为只有客户知道安装点。

在组播导航中，客户向名字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方案，在该方案中，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。

迭代导航模型的替代方案是，名字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]区分了非递归式以及递归式服务器控制的导航（参见图9-3）。在非递归式服务器控制的导航中，任何名字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对等服务器通信，如同它是一个客户一样。在递归式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。这个过程递归地继续下去，直到名字被解析为止。



名字服务器NS1代表客户与其他名字服务器通信

图9-3 非递归和递归的服务器控制的导航

若一个名字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名字服务器。此外，名字服务器甚至会被禁止探测在另一个管理域中的名字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均不适用，此时必须使用递归式的服务器控制的导航方式。获得授权的名字服务器向指定的名字服务器请求名字服务数据，该指定的名字服务器由其他的管理部门管理，它返回相应的属性，而并不暴露命名数据库

的不同部分是如何存储的。

缓存 在DNS以及其他名字服务中，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，如果该缓存包含通过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，客户软件将着手从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。

缓存是名字服务性能的关键，即使在名字服务器崩溃的情况下，缓存也可以帮助维护名字服务器以及其他服务的可用性。它的作用非常清晰，即通过节约与名字服务器的通信时间提高响应速度。缓存可用于在导航路径上消除高层的名字服务器——特别是根服务器，同时在一些服务器产生故障的情况下，允许解析过程继续进行。

因为名字数据不会经常改变，从而客户端的名字解析程序的缓存在名字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，一个名字服务也有可能在解析过程中返回过时的属性信息，例如过时的地址。

9.2.3 域名系统

域名系统是一个名字服务，它的主命名数据库主要在因特网上使用。它由Mockapetris[1987](RFC 1034)设计，用于替代原有的因特网命名方案。在原有的方案中，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通过FTP下载信息[Harrenstien et al.1985]。此方案有以下三个缺点：

- 计算机数量众多时，缺乏可伸缩性。
- 本地组织希望管理自己的命名系统。
- 需要通用的名字服务——而不是仅查找计算机地址的名字服务。

376 ~ 378

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中简单地称作域。然而原则上，所有的对象都能被命名，同时对象名字的结构可以支持各种各样的实现。组织和部门可以管理自己的命名数据。因特网DNS绑定了几百亿个名字，而全世界的计算机都基于该DNS查找。任何名字均可被任何客户解析，这是由名字数据库的层次化分区、命名数据的复制以及缓存来实现的。

域名 DNS可供多种实现使用，每种实现都可以拥有自己的名字空间，但实际上，只有一种方法应用最广，即在因特网上使用的命名方式。因特网DNS名字空间既按组织分区也按地域分区。名字中最高级的域位于右端。最初在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。
- int：国际组织。

在2000年的早些时候，又增加了一些新的顶级组织域名。现在，可以通过因特网编号管理局[www.iana.org]获得包含全部通用域名的列表。

此外，每个国家拥有自己的域名：

- 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的各个组织。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）。值得注意的是，尽管有相似的后缀uk，doit.co.uk这样的域也能在Doit Ltd（一家英国公司）的西班牙办事处拥有数据，
379 换句话说，地域式的域名仅仅是一种习惯用法，域名事实上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找。具体内容如下：

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.qmul.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协议与占据了保留端口号的特定IP地址的Web服务器通信。FTP服务与SMTP的工作方式类似。例如，当FTP客户程序获得域名ftp.dcs.qmul.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协议在一个保留端口上与服务器通信。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。也有不用别名的例子，考虑这样的情况，一个用户使用telnet客户程序与域名为jeans-pc.dcs.qmul.ac.uk的主机联系，telnet发出DNS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，邮件软件可以尝试使用其他服务器。DNS对每个邮件主机均返回一个整型的优先值，表示邮件主机应被尝试的顺序。

有些安装版本也包括了其他类型的查询，但远没有上面两种查询应用广泛。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。对于接收查询的名字服务器，仅当IP地址在自己的域中时才会应答。

主机信息：DNS可以存储主机域名相应的计算机的体系结构类型与操作系统信息。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黑客提供了有用的信息。

已知的服务：有些名字服务器支持这样的服务：给定计算机域名，名字服务返回该计算机运行的一组服务（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、类别与类型三者来定义。因特网上域名的类别是IN。查询的类型定义是否需要一个IP地址、一个邮件主机、一个名字服务器或其他类型信息。特殊域in-addr.arpa存储的是IP地址，可以用于反向查询。类别属性用于区分因特网命名数据库与实验阶段的DNS命名数据库。一个给定的数据库会有一组类型定义，因特网数据库的类
380 型定义参见图9-5。

DNS名字服务器 通过结合使用分区、复制以及在需要地点的最近处缓存命名数据库等方法，可以解决伸缩性问题。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。大多数查询涉及本地域的计算机，并且由该域中的服务器给出应答。然而，每个服务器记录了其他名字服务器的域名与地址，这样可满足对本地域以外对象查询的需要。

DNS命名数据被划分为区域，一个区域包含下列数据：

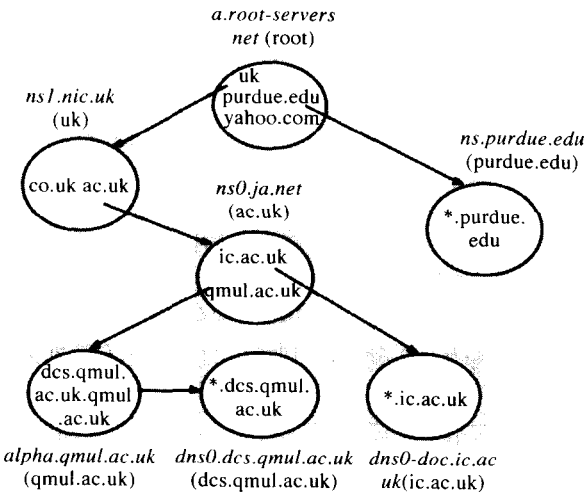
- 除那些由更低层的权威机构管理的子域内的数据外，一个域里名字的所有属性数据。例如，一个区域除了包括每个系（如计算机科学系dcs.qmul.ac.uk）持有的数据外，还包括属于伦敦大学Queen Mary College（qmul.ac.uk）的数据。
- 至少两台名字服务器的名称与地址，这些服务器提供该区域的权威数据，而且数据的版本被认为是最新的。
- 一些名字服务器的名字，这些名字服务器存储了被委托的子域的权威数据，以及给出了服务器IP地址后的一些“粘合”数据。
- 区域管理参数，例如管理区域数据缓存与复制的参数。

一个服务器可以拥有零个或多个区域的权威数据。为了在一个服务器发生故障的情况下，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来源。有两种服务器可提供权威数据：主服务器直接从本地主控文件中读取区域数据。次服务器从主服务器下载区域数据。它们周期性地与主服务器通信，以检验次服务器上的版本是否与主服务器上数据版本的一致。若次服务器上的版本过期，则主服务器将最新的版本发送给它。管理员将次服务器检查过期的频率作为一个区域参数设定，通常它的值是一天一次或两天一次。

任何服务器均可缓存其他服务器的数据，以避免在解析名字时再与那些服务器联系请求数据。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。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。它将仅在存活期内向客户提供缓存数据；对于超过存活期后的查询，服务器需要重新与权威服务器联系，核对它的数据。这是一个有用的特征，它减少了网络流量，同时保留了系统管理员的灵活性。在预料到属性不会经常改变时，可以给它们赋予相当长的存活期。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2001年时DNS数据库的部分安排。注意，在实际中，a.root-servers.net这样的根服务器除了保存第一级域名外，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。根名字服务器拥有顶级域名字服务器的权威条目。它们也同时是com、edu等常用顶级 381
域的权威名字服务器。然而，根名字服务器不是国家域的名字服务器。例如，uk域有若干名字服务器，其中一个被称为ns1.nic.net。这些名字服务器知道英国二级域（如ac.uk与co.uk）的名字服务器。域ac.uk的名字服务器知道本国所有大学域的名字服务器，例如qmul.ac.uk或ic.ac.uk。在某些情况下，一个大学域将某些管理权委派给一个子域，如dcs.qmul.ac.uk。



注意：名字服务器名用斜体，而相应的域在括号中。箭头指示了名字服务器入口。

图9-4 DNS名字服务器（在2001年时）

根域信息由主服务器复制到若干次服务器上。尽管如此，根据Liu与Albitz[1998]的调查，一些根服务器依然要做到每秒接收约1000个查询。所有DNS服务器都会存储一个或多个根名字服务器的地址，这些服务器的地址通常不会改变。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www.berkeley.edu这样的具有三个部分的域名，最坏情况下需要两步导航；第一步是向存储了合适的名字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.qmul.ac.uk可以使用本地服务器dns0.dcs.qmul.ac.uk，从域

382 dcs.qmul.ac.uk查找到。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协议格式的消息，再与一个或多个名字服务器通信。通信时一般使用简单的请求-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）。解析器有可能超时，必要时需要重发查询。解析器可被配置成与一组带优先级的初始名字服务器联系，以应付某个或某几个服务器不可用的情况。

DNS体系结构能够处理递归导航与迭代导航两种情况。当与名字服务器联系时，解析器指定需要何种类型的导航。然而，名字服务器并不一定实现递归导航。正如上面所指出的，迭代导航会占用服务器线程，这意味着其他请求会被延迟。

为节省网络通信，DNS协议允许将多个查询打包到一个请求消息中，相应地，名字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名字服务器的文件中。对于因特网数据库，包含图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条目主要是为了存储域名的任意信息。表中的数据是基于2001年时的情况。

记录类型	含 义	主要内容
A	计算机地址	IP号
NS	权威名字服务器	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（反向查找）	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	文本字符串	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，该记录包含区域参数，参数可指定版本号以及次服务器刷新副本的频率等。SOA类型的记录后紧跟着类型为NS的记录集合，用于指定域的名字服务器，接着是类型为MX的记录集合，用于指出邮件主机的优先级和域名。例如，域dcs.qmul.
383 ac.uk的数据库中记录的一部分如图9-6所示，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.qmul.ac.uk
	1D	IN	MX	2 mail2.qmul.ac.uk

图9-6 DNS区域数据记录

后面的类型为A的记录会给出两个名字服务器——dns0与dns1的IP地址。邮件主机以及第三个名字服务器的IP地址在域相应的数据库中给出。

对于dcs.qmul.ac.uk这样较低层的区域，数据库剩下的主要记录是A类型的，它将计算机的域名映射到一个IP地址。对于众所周知的服务，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	apricot
apricot	1D	IN	A	138.37.88.248

如果该域还有子域，那么将会有更多的NS类型的记录，这些记录指定了子域的名字服务器，而这些服务器也会有自己的A类型的条目。例如，qmul.ac.uk的数据库对于子域dcs.qmul.ac.uk的名字服务器，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

名字服务器的负载共享：对于某些站点，诸如Web、FTP等常用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，名字服务器对该组的每台计算机都有一条记录，记录其IP地址。对于名字会涉及多条记录的查询，名字服务器根据循环调度方法返回IP地址。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方案，因为一旦一个非权威的名字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。为消除这种后果，资源记录一般给定较短的存活期。

DNS的BIND实现 Berkeley因特网域名系统（Berkeley Internet Name Domain, BIND）是运行UNIX的计算机的DNS实现。客户程序通过链入BIND软件库作为名字服务的解析器。DNS名字服务器所在的计算机运行已命名的守护进程。

384

BIND允许使用三类名字服务器：主服务器、次服务器以及仅提供缓存功能的服务器。已命名的程序根据配置文件内容仅实现三类中的一类服务器。前两类服务器上文已介绍过。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时所积累的数据。

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名字服务的一个或多个次服务器。另外，每个计算机常常运行自己的缓存服务器，以降低网络开销，加速响应时间。

关于DNS的讨论 考虑到因特网命名数据的数量以及网络的规模，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，获得上述效果是通过对命名数据进行分区、复制以及缓存而达到的。命名的对象主要是计算机、名字服务器以及邮件主机。计算机（主机）名到IP地址的映射以及名字服务器与邮件主机的标识等信息的改变不太频繁，因此，缓存与复制可在一个相对宽松的环境中进行。

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给提供客户过期的信息，这里没有使用第15章中研究的复制技术。然而，只有在客户试图使用过期信息的情况下，不一致性才会产生不好的后果。DNS自己未指出如何探测过期数据。

除计算机外，DNS还命名了一种特殊的服务：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，用户无需显式地用名字指出服务。电子邮件应用在与DNS服务器联系时，通过使用合适的查询类型，透明地选择该服务。

总而言之，DNS存储的不同类型的名字数据是非常有限的，但这在诸如电子邮件这样的应用中已经足够了，因为在电子邮件这类应用中可以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应该是值得质疑的。DNS并不是为因特网设计的唯一名字服务，它与本地名字与目录服务共存，这两种服务都存储了与局部需求有关的数据（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]，该服务存储了一个域中所有资源的详细信息）。

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节的全局名字服务的实例研究中，考虑了命名设
385 计在这些方面的需求。在此之前，我们先来介绍目录服务。

9.3 目录服务

我们已描述了名字服务如何存储<名字，属性>对集合以及如何通过名字查找属性。很自然地，我们会考虑上述情况的另一面，即将*属性*作为查找的关键字。在这些服务中，文本名仅仅被看作是一个属性。有时，用户希望找到某一个人或资源，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访问，用户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 X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，条目的查找基于属性规范。目录服务的例子有：微软的活动目录服务、X.500以及它相关的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*（yellow pages service），而传统的名字服务被称为*白页服务*，这与不同类型的电话簿目录相似。目录服务有时也被称为*基于属性的名字服务*。

目录服务返回满足特定属性的所有对象的属性集合。例如，"TelephoneNumber = 020-555 9980"这样的请求会返回{'name = John Smith', 'TelephoneNumber = 020-555 9980', 'emailAddress = john@dcs.gormenghast.ac.uk',...}。客户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。将在19.4节描述的统一目录和发现服务（UDDI）提供了关于各个机构以及它们提供的服务的信息的白页和黄页服务。

除了UDDI，*发现服务*通常特指用于自发网络环境下设备提供的服务的目录服务。正如在2.2.3节中所述，自发网络中的设备连接与断连是不可预测的。发现服务与其他目录服务的本质区别在于一般目录服务的地址通常是众所周知的，而且在客户端是预先配置好的。但是，在自发网络环境下，一个设备会随时加入，这就会导致组播导航（至少这个设备第一次访问这个本地发现服务时会这样）。16.2节会详细讨论发现服务。

属性用于指定对象显然比名字更有效。在不知道名字的情况下，可以通过编写程序来，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，而使用组织机构划分的名字会有这种风险。然而，使用文本名相对简单，这使得在很多应用中，名字服
386 务不可能被基于属性的命名方法替代。

9.4 实例研究：全局名字服务

全局名字服务（GNS）是由Lampson与DEC系统研究中心的同事[Lampson 1986]设计与实现的，它用于提供资源定位、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：互联网使用的名字服务必须支持一个名字数据库，该数据库可以扩展到包含数百万台计算机的名字以及数十亿用户的邮件地址。GNS的设计者也意识到，名字数据库可能会有很长的生命周期，它必须在规模由小变大以及底层网络发展的情况下有效地工作。在此过程中，

名字空间的结构可以改变以反映组织结构的变化。名字服务必须允许其中的个人、组织、小组的名字发生变化。除此以外，也应允许一个公司被另一个公司接管时，名字结构发生变化。在本节中，我们重点描述允许这些变化的设计要点。

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，因此缓存的使用成为设计要点，在这种情况下维护数据库条目的所有副本的完全一致性就变得困难。决定采取的缓存一致性策略基于下面的假设：数据库的更新将是非频繁的，而因为客户可以探测和修复已过期命名数据的使用，所以慢速发送数据更新是可以接受的。

GNS管理的名字数据库由一个包括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，这与UNIX文件系统的文件名很相似。每个目录被赋予一个整数作为唯一的目录标识符（Directory Identifier, 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。目录树的叶子中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。

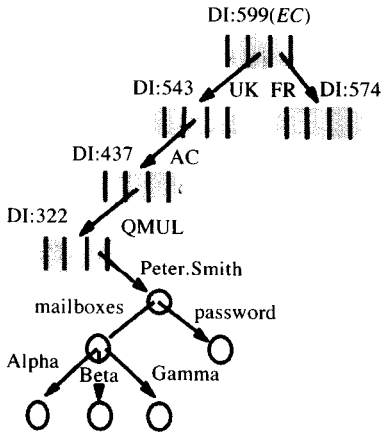


图9-7 GNS中用户Peter.Smith的目录树和值树

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。例如，参考图9-7，在图中为说明方便，DI都是小整数，尽管在实际中为保证唯一性，DI会从很广的整数范围中选择。目录QMUL下的用户Peter.Smith的属性会存储在一个名为<*EC*/UK/AC/QMUL, Peter.Smith>的值树中。该值树包含一个口令和多个邮件地址，口令可以通过<*EC*/UK/AC/QMUL, Peter.Smith/password>方式来引用，而每个邮件地址都作为值树的单独节点，以<*EC*/UK/AC/QMUL, Peter.Smith/mailboxes>作为节点名列出。

目录树被分区后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发生时树的一致性。例如，两个用户试图同时用同一名字创建一个条目，这时应该仅有一人能成功。复制目录带来了另一个一致性问题，通过一个能够保证最终一致性的异步更新分布式算法能解决该一致性问题，但不能保证所有的副本都是最新的。在该问题上达到这种级别的一致性被认为是令人满意的。

适应改变 现在我们探讨与适应名字数据库的增长和改变有关的设计方面。在客户与管理员层，可以通过正常的方式扩展目录树来适应增长。但我们可能会集成两个原来分离的GNS名字树。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显示了在要合并的树的根之上，引入了新根*WORLD*。这个技术非常简单直观，但对继续使用集成前的“根”作为名字的客户有什么影响呢？例如，</UK/AC/QMUL, Perter.Smith>是在集成之前客户使用的名字。它是一个绝对名（因为它以“/”开始），但根指的是*EC*，而不是*WORLD*。*EC*与*NORTH AMERICA*是工作根，工作根是一个初始上下文，这时必须查询以根“/”开始的名字。

唯一目录标识符的存在可用于解决这个问题。每个程序的工作根必须作为执行环境的一部分（与一个程序的工作目录相同）。当一个在欧盟的客户使用形如</UK/AC/QMUL, Perter.Smith>的名字时，它的本地用户代理由于知道它的工作根，会在名字前添加目录标识符EC(#599)，从而构造出名字<#599/UK/AC/QMUL, Perter.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符的概念上的超根目录，例如，<*WORLD*/EC/UK/AC/QMUL, Peter.Smith>，但设计无法假设所有的客户考虑到该变化而被更新。

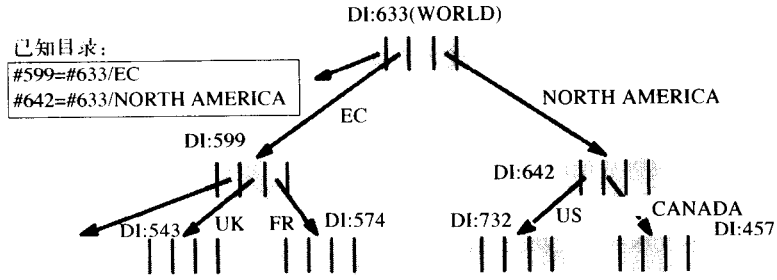


图9-8 在一个新根下合并树

上述技术解决了逻辑问题，它甚至在插入一个新的真实根的情况下，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。但这样做产生了一个实现问题：在包含上百万目录的分布式名字数据库中，若仅给定#599这样的目录标识符，GNS服务如何定位一个目录？GNS的解决方案是在名字数据库的当前真实根下包含一个表，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。一旦名字数据库真实的根发生改变，如图9-8所示，就会向所有GNS服务器通知真实根的新位置。然后它们可使用常用方式解释WORLD/EC/UK/AC/QMUL（指向真实根）形式的名字，也可使用“已知目录”表将#599/UK/AC/QMUL格式的名字翻译成以真实根开始的完全路径名。

GNS也支持数据库的重构，以适应组织变化。假设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子树仅被移到EC目录下，以*WORLD*/NORTH AMERICA/US开始的名字将无法继续工作。GNS采取的方法是增加一个“符号链接”替代原有的US条目（见图9-9中加黑的一部分）。GNS目录查找过程将链接重定向到新位置上的US目录。

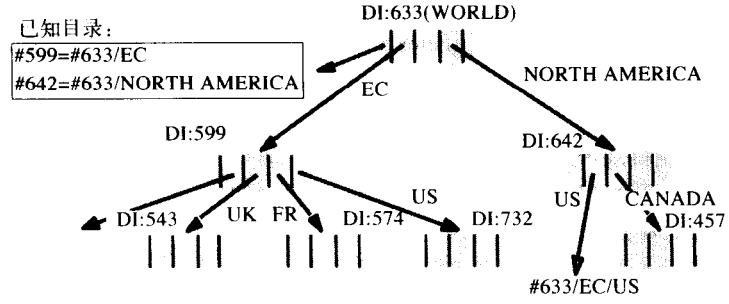


图9-9 重构目录

GNS的讨论 GNS由Grapevine[Birrell et al. 1982]以及Clearinghouse[Opeen and Dala1 1983]发展而来，这两个系统是Xerox Corporation成功开发的主要面向邮件发送名字系统。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）必须在每个节点被复制。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。

9.5 实例研究：X.500目录服务

X.500是9.3节所定义的目录服务。它可以按传统的名字服务的使用方式使用，但它通常用于满足描述性的查询，并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。这种服务的使用有可能非常的分散。查询的不同导致了目录服务的不同使用方法，可以使用与查询电话簿

相似的方法，例如通过简单的“白页”查询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职位、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。

个人与组织可以使用目录服务，在网络中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，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，但它的设计在很大程度上并不依赖于其他OSI标准，因此可以被看作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的详细信息以及实现方法感兴趣的读者可以参阅Rose[Rose 1992]有关该主题的书。X.500也是LDAP的基础（将在下面讨论），它被用于DCE的目录服务[OSF 1997]。 387 ~ 390

在X.500服务器中的数据被组织成一个由名字节点构成的树状结构，如在本章中提到的其他名字服务器一样。但在X.500中，树的每个节点存储了大量的属性，访问不仅可以根据名称进行，也可以根据属性的组合搜索条目。

X.500名字树也称为目录信息树（Directory Information Tree, DIT），而整个目录结构以及与节点有关的数据称为目录信息库（Directory Information Base, DIB）。一般倾向于将全世界范围内的机构提供的信息存储在一个集成的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至少会提供一个服务器。客户通过建立一个到服务器的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，该服务器会调用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。

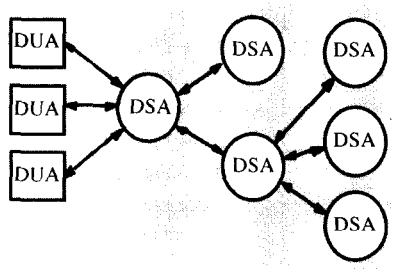


图9-10 X.500服务的体系结构

在X.500标准的术语中，服务器称为目录服务代理（Directory Service Agent, DSA），客户称为目录用户代理（Directory User Agent, DUA），图9-10给出了软件体系结构以及可能的导航模型中的一个。其中，每个DUA客户进程与单个DSA进程交互，而DSA进程在需要的时候访问其他DSA以满足请求。

DIB的每个条目由一个名字和一组属性集组成。与其他名字服务器相似，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，其中包括一个基本节点，然后DUA即可使用较短的相对名，该名字给出了从基本节点到已命名条目的路径。

图9-11显示了包括英国Gormenghast大学的部分目录信息树，图9-12是一个相关的DIB条目。DIB与DIT中的条目的数据结构非常灵活。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。属性的类型由类型名表示（如CountryName、organizationName、commonName、telephoneNumber、mailbox、objectClass）。在需要的时候可以定义新的属性类型。 391 对于每一个类型名、都有一个相应的类型定义，该定义包括一个类型描述以及一个使用ASN.1记号（一种语法定义的标准记号）表示的语法定义，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。每个条目包括一个objectClass属性，它定义了一个条目指向的对象的类。Organization、organizationPerson以及document都是objectClass的值的例子。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。类的定义可以组织成一个继承层次，其中，除了topClass类外，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。

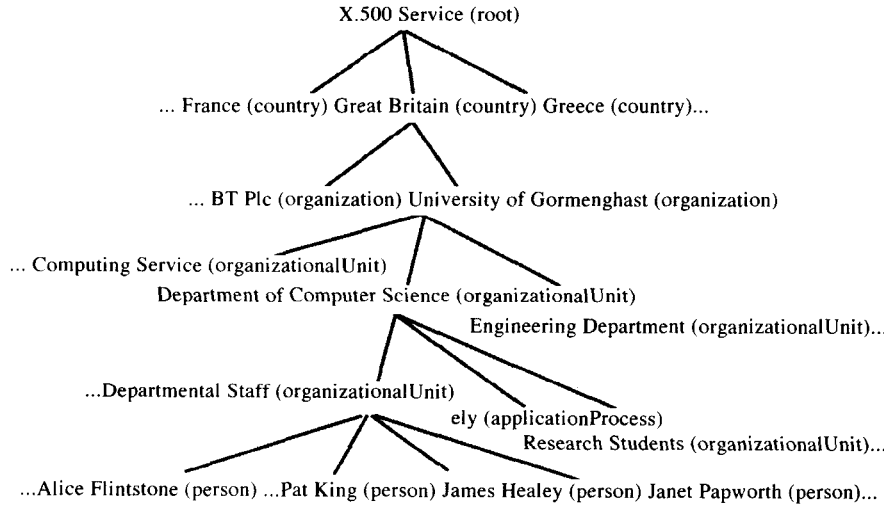


图9-11 X.500目录信息树的一部分

info	
Alice Flintstone, Departmental Staff, Department of Computer Science, University of Gormenghast, GB	
commonName	*uid*
Alice.L.Flintstone	alf
Alice.Flintstone	*mail*
Alice.Flintstone	alf@dcs.gormenghast.ac.uk
A.Flintatone	Alice.Flintstone@dcs.gormenghast.ac.uk
surname	*roomNumber*
Flintstone	Z42
telephoneNumber	*userClass*
+44 986 33 4604	Research Fellow

图9-12 一个X.500 DIB条目

要确定DIB条目的名字（确定它在DIT中位置的名字），可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*（Distinguished Name, DN）。

现在我们考虑访问目录的方法。一般有两种访问请求：

读：给定某个条目的绝对的或相对名字（在X.500中称为*域名*）以及可读的属性列表（或者需要的属性的指示）即可。DSA通过在DIT中导航，定位已命名的条目，当DIT中没有相关条目时，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。

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表达式作为参数。基本名指定在DIT中开始搜索的节点，过滤表达式是一个布尔表达式，用于对基本节点以下的每个节点进行判定。过滤表达式指定了一个搜索准则：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），这些名字是基本节点之下的条目名，并且这些条目的过滤表达式判定值为真。

例如，可以构造一个过滤表达式，用于寻找在Gormenghast大学的计算机科学系占据了Z42房

间的员工的commonName（参见图9-12）。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（可能会驻留在多台服务器中）需要很大开销。可以提供更多的参数以限制搜索的范围，如持续搜索的时间以及返回条目的数目。

DIB的管理与更新 DSA接口包括增加、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的访问必须限定只能由特定用户或是一类用户来进行。

一般来说，DIB是分区的，每个组织至少应提供一个服务器来容纳该组织中实体的细节。DIB的各个部分可以复制到多个服务器上。 392 ~ 393

作为一个标准（或按CCITT术语称为“建议”），X.500未涉及实现细节。然而，很清楚的是，任何在广域互连网络中的涉及多个服务器的实现必须广泛地使用复制与缓存技术，以避免过多的查询重定向。

Rose[1992]描述了X.500的一个实现——QUIPU[Kille 1991]，它是伦敦的University College开发的一个系统。在该实现中，缓存与复制的级别是单个DIB条目，或是同一节点下的条目集。系统假设在更新后值变得不一致，而恢复一致性的时间间隔可能会有几分种。这种更新分发的形式对于目录服务应用而言是可接收的。

轻量级目录访问协议 X.500的标准接口使用了涉及ISO协议栈中上层的协议。Michigan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*（Lightweight Directory Access Protocol, LDAP），在该协议中，DUA直接通过TCP/IP访问X.500目录服务，参见RFC 2251[Wahl et al. 1997]。LDAP还用其他方法简化了X.500接口。例如，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并不需要它。任何实现都可以使用符合更简单的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接口。LDAP已被广泛采用，特别适用于企业内部网目录服务。它通过认证提供了安全的目录访问。

9.6 小结

本章描述了分布式系统中名字服务的设计与实现。名字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。

名字服务的主要需求是处理任意数目名字的能力，服务应具有长期性、高可用性、故障隔离性与不信任容忍性。

首要的设计问题是名字空间的结构——管理名字的语法规则，相关问题是解析模型，即有多个部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。最后，大多数设计将名字空间分割为域，即名字空间的离散区域，每个域具有一个独立的相关权威机构，该机构控制域内名字的绑定。

名字服务的实现可以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名字服务器上，每个服务器至少存储一个名字域的部分名字集。因此，出现了导航问题，即当需要的信息存储在多个站点上时名字被解析的方式。支持的导航类型有迭代、组播、递归服务器控制的导航，以及非递归服务器控制的导航。 394

另一个有关名字服务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降低名字解析时间有帮助。

本章讨论了两个主要的名字服务的设计与实现。域名系统被广泛地用于因特网上的计算机命名与电子邮件寻址，它通过复制与缓存获得理想的响应时间。全局名字服务解决了组织机构变化时重新配置名字空间的问题。

本章还讨论了目录服务，当客户提供基于属性的描述时，该服务返回与对象和服务相匹配的

数据。X.500是目录服务的一个模型，它既可以用于个人组织的目录也可以用于全球目录。随着LDAP软件的使用，X.500也在企业内部网中被广泛使用。

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（包括标识符）与属性。（第368页）

9.2 讨论在名字服务中使用别名带来的问题，并且指出如何解决这些问题。（第373页）

9.3 解释为什么在不同名字空间可以局部集成的名字服务中（如由NFS提供的文件命名机制中）需要迭代导航。（第376页）

9.4 描述组播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，可以得到什么结论？（第377页）

9.5 缓存如何提高名字服务的可用性？（第378页）

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。（第379页）

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，它可以执行单个名字服务器的查询。（第381页）

9.8 为什么DNS根服务器包含两层名字（如yahoo.com与purdue.du）而不是一层名字（如edu与com？）（第381页）

9.9 默认情况下，DNS名字服务器包含哪些名字服务器的地址，为什么？（第381页）

9.10 为什么DNS客户选择递归导航而不是迭代导航？迭代导航选项会对名字服务器的并发性产生何种影响？（第383页）

395

9.11 什么情况下一个DNS服务器会给一个名字查询返回多个回答，为什么？（第383页）

9.12 GNS未保证命名数据库中的所有条目的副本是最新的，GNS的客户怎样才能意识到它们所持有的是一个过期的条目？在哪种情况下，这是有害的？（第387页）

9.13 讨论用X.500目录服务替代DNS与因特网邮件传送程序的好处与不足。为一个互联网粗略设计一下邮件传送程序，其中每个邮件用户与邮件主机都注册到一个X.500数据库。（第390页）

9.14 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？（第390页）

396

第10章 对等系统

对等系统代表构造分布式系统和应用的一种范型，在对等系统中，因特网上的众多主机以一种一致的服务方式提供它们的数据和计算资源。对等系统的出现源于因特网的快速发展，现在它们已经包含了数百万台电脑以及同等数量的要求访问共享资源的用户。

对等系统的一个关键问题是数据对象在多主机环境中的放置问题，以及考虑在负载平衡的前提下访问数据的方式，并且在不增加不必要开销的情况下保证系统的可用性。我们将描述几个最近开发出来的能够满足上述要求的对等系统和应用。

对等中间件系统也获得了越来越广泛的应用，这些中间件能够使全球“处于因特网边缘”的计算机共享计算资源、存储资源和数据资源。它们以一种新的方式利用现有的命名、路由、数据复制和安全技术，在一组不可靠、不可信的计算机和网络上建立一个可靠的资源共享层。

对等应用已用于提供文件共享、Web缓存、信息发布以及其他一些利用因特网上众多计算机资源的服务。对等应用在存储海量不变数据方面具有非常高的工作效率，但是这样的设计对于存储和更新可变数据对象则效率会有所降低。 397

10.1 简介

现在对因特网服务的需求越来越大，其规模的上界可能和世界人口数相当。对等系统的目标就是通过消除对单独管理的服务器以及相应的基础设施的需求，实现在一个巨大的范围内共享计算机数据和资源。

通过增加服务器来扩展服务的范围收效甚微，因为服务提供者要为提供服务而购置和管理大量的服务器。管理这些服务器和对其进行故障恢复也将耗费大量资源。另外，在可用物理链路上给一台服务器提供的网络带宽也是一个主要的限制。系统级的服务（如Sun NFS（见8.3节）、Andrew 文件系统（见8.4节）或视频服务器（见17.6节））和应用层服务（如Google、Amazon 或者eBay）都不同程度地表现出上述问题。

对等系统旨在利用存在于因特网以及其他不断发展的网络上的个人计算机和工作站上的数据和计算资源，提供有用的分布式服务和应用。随着桌面计算机和服务器之间的性能差异越来越小，以及宽带网络的不断增长，对等系统越来越引起人们的关注。

但是对等系统还有更为宏远的目标。一位作者[Shirky 2000]曾经将对等应用定义为“能够利用处于因特网边缘的计算机上的资源的应用，这些资源包括存储资源、CPU资源、内容资源和人本身”。上述的各种共享资源在现在个人计算机可用的分布式应用中都能够找到相应的代表。本章将描述一些通用的技术，这些技术可以简化对等系统的构造，增强系统的可伸缩性、可靠性和安全性。

传统的客户/服务器系统可以管理和访问文件、Web页面或者其他信息对象，这些对象位于一台服务器上或者一个紧耦合的小计算机集群上。采用集中式设计，需要对数据资源的放置以及对服务器硬件资源的管理做出决策，但是服务器提供服务的规模将会受到服务器硬件能力和网络连接的限制。对等系统可以对整个网络上的计算机上的资源进行访问（不论这个网络是因特网还是公司内的局域网）。设计这种系统的一个关键方面是有关信息对象的放置和检索的算法的设计，主要设计目标是提供高度分散和自治的服务，该服务允许计算机加入或者退出当前服务时，能够在参与的计算机中动态的平衡存储和处理负载。

对等系统具有以下特点：

- 系统设计确保每个用户都能向系统共享资源。
- 虽然各个参与的节点提供的资源不同，但在同一个对等系统中它们具有相同的功能和责任。
- 系统不需要一个中心管理系统就能正常运行。
- 398 系统的设计能够给资源的提供者和使用者提供一定限度的匿名性。
- 系统能够高效运行的一个关键点就是选择一个在大量主机中放置数据资源，以及访问这些资源的算法。这个算法能够自动平衡各个主机的负载，确保可用性，并且不会增加不必要的系统开销。

由不同用户和组织拥有和管理的计算机和网络连接往往都是易变的资源，因为拥有者们不能保证这些资源总是可访问的、连接在网络上的、不发生错误的，因此，参与到对等系统中的计算机和进程的可用性是不可预知的。所以，对等服务也不能确保对单个资源访问的万无一失，虽然可以通过访问相应资源的一个副本使得这种故障发生的概率足够小。值得注意的是，如果利用资源复制能在一定程度抵抗对恶意节点的干扰（例如，通过拜占庭容错技术，见第15章），那么可以把一个脆弱的对等系统变成一个健壮的系统。

几个早期的基于因特网的服务，包括DNS（见9.2.3节）和Netnews/Usenet[Kantor和Lapsley 1986]，都采用了多服务器的可伸缩的和可容错的体系结构。Xerox Grapevine名字注册和邮件传递服务[Birrell et al.1982，Schroeder et al.1984]也是早期的一个令人感兴趣的可伸缩并且具有容错机制的分布式服务。用于分布式共识的Lamport 兼职议会算法（Lamport Parliament Algorithm），Bayou的复制存储系统（见14.4.2节）和无类别域间IP路由算法（见3.4.3节）都是分布式算法的例子，这些算法描述了信息在分布式网络中如何放置或定位，它们也被认为是对等系统的前身。

但是对于利用因特网边缘资源的对等系统的潜力只有在大量用户能够获得宽带网络连接，并且保证经常在线，使他们的桌面计算机变成一个适合共享资源的平台时才能显现出来。最早的对等系统1999年在美国出现，到2004年年中的时候，全世界具有因特网宽带连接的数量已经超过了一亿[Internet World Stats 2004]。

对等系统和应用的发展到现在已经经历了三代。第一代是从提供音乐文件交换共享服务的Napster开始[OpenNap 2001]，我们将在下节描述它。第二代文件共享应用能够提供更大的伸缩性、可以匿名使用、具有容错机制，典型的软件包括Freenet[Clarke et al. 2000, Freenet 2004]、Gnutella、Kazaa[Leibowitz et al. 2003]和BitTorrent[Cohen 2003]。

对等中间件 第三代以中间件层的出现为特征，它能够在全球范围内管理与应用无关的分布式资源。一些研究团队已经完成了对等中间件平台的开发、评估和改良，并且把他们的研究成果部署到一系列的应用服务中。已经有完整版本的常用系统包括Pastry[Rowstron and Druschel 2001]、Tapestry[Zhao et al. 2004]、CAN[Ratnasamy et al. 2001]、Chord[Stoica et al.2001]和
399 Kademlia[Maymounkov and Mazieres 2002]。

这些平台可以把资源（数据对象、文件）放置到一组分布在因特网中的计算机上。它们可以代表客户进行消息路由，减轻客户放置资源的决策负担并记录要访问的资源的行踪。与第二代系统不同的是，它们可以保证使用不超过某个限制数量的网络跳数正确地传送请求。由于主机并不总是可用的、可信的，以及对负载平衡和信息存储、使用的本地化需求，这些系统以结构化的方式把资源的副本放到可用的主机上。

资源可以用一个全局唯一标识符（GUID）来标识，这些标识符通常是根据资源的全部或者部分状态计算一个安全散列码来获得的（见7.4.3节）。通过使用安全散列码，使得资源本身能够进行“自我验证”——客户接收资源时可以验证散列码。这样可以避免资源被不信任的节点（资源有可能存储在这个节点上）篡改。但是这个技术要求资源的状态不能改变，否则一旦资源的状态改变，它将会产生一个不同的安全散列值。因此，对等存储系统本质上适合不可变对象的存储（例如，

音乐或视频文件)。这些系统若用于存储可变对象，会遇到更大的挑战，不过这些挑战可以通过增加一些可信服务器来解决，这些服务器管理可变数据的一系列版本，也可以确定当前的版本（例如，10.6.2节和10.6.3节将介绍的OceanStore 和 Ivy)。

若将对等系统应用于要求资源有较高可用性的场合，那么需要进行精心的设计，以免所有对象副本同时不可用。如果对象被存储在一组相似的计算机中（所有者、地理位置、管理方式、网络连接、国家或者权限均相同)，那么具有一定的风险。使用随机分布的GUID可使对象的副本随机分布到底层的网络节点中，从而降低这种风险。如果底层网络跨越了全球的很多组织，那么这种资源及其副本同时不可用的风险将会大大降低。

覆盖路由与IP路由 乍一看，覆盖层上的路由和构成因特网通信机制基础的IP数据包路由（见3.4.3节）有很多相似点。因此，很自然会有“为什么在对等系统中还需要一个应用层路由机制？”的问题。这个问题的答案可以参考图10-1中列出的几个不同点。有人可能会认为这些不同点来自于作为因特网主协议的IP协议的遗留特性，但是遗留特性的影响太大，以至于我们在重新设计IP来更直接地支持对等应用的过程中无法克服它们。

	IP	应用层路由覆盖
规模	IPv4可寻址节点最多为2^{32}个。IPv6的名字空间是非常大的（可达2^{128})，但是在两个版本中，地址空间都是按层次结构构造的，由于管理上的需求，大量的地址已经被预先分配了	对等系统可以寻址到更多的对象。GUID的名字空间非常大且扁平（$>2^{128}$)，允许其中可使用的空间更大
负载平衡	路由器上的负载由网络拓扑以及相关的流量模型确定	对象的位置可以随机，因此流量模型与网络拓扑是不相关的
网络动态性（对象/节点的添加/删除）	IP路由表是基于常量时间（1小时）按尽力而为方式进行异步更新的	路由表可以同步或者异步更新（带有几秒的延迟）
容错	IP网络的管理者将冗余引进到IP网络中，以便当一台路由器或网络连接发生故障时，确保容错度。n倍复制的开销很高	路由和对象引用可以复制n次，从而保证在n个节点或网络链接故障时的容错能力
目标识别	每个IP地址唯一地映射到一个目标节点上	消息可以路由给离目标对象的最近副本
安全性和匿名性	只有当所有节点都是可信的时候，寻址才是安全的。地址的拥有者不能匿名	甚至可以在有限信任的环境中获得安全性。可以提供一定程度的匿名性

图10-1 IP路由和对等应用中的覆盖路由的不同点

分布式计算 利用终端用户空闲的计算资源一直是开发者颇有兴趣并付诸实验的主题。最早的在Xerox PARC[Shoch and Hupp 1982]个人计算机上开展的这项工作，说明在很多连入同一个网络的个人计算机上，通过运行后台进程来执行松耦合计算密集型任务是可行的。最近，更多的计算机用于进行科学计算中，这些科学计算都需要几乎无限数量的计算资源。

在这类应用中最著名的是SETI@home项目[Anderson et al. 2002]，它是外星智慧搜索工程的一部分。SETI@home项目把数字射电望远镜采集到的数据流分割成107个工作单元，其中每个单元的大小约为350KB，然后把它们分发给志愿提供计算资源的客户计算机。每个工作单元将冗余地分发给3～4个计算机，以免某些节点计算机发生错误或存在恶意的节点，这样还可以检查到重要的信号模式。由一台服务器负责与所有的客户通信，并且由该服务器分发工作单元，并且最后协调各个客户的结果。Anderson等[2002]指出，到2002年8月，已经有大约391万台个人计算机参与到SETI@home项目中来，它们处理完成了2.21亿个工作单元，这个工作量如果让具有每秒27.36万亿

次运算能力的巨型计算机来工作，它也需要12个月直到2002年7月才能完成。这个结果在当时创造了最大的单个计算量的记录。

400 2 401

SETI计算是不同寻常的：当客户计算机处理工作单元的时候，它们之间不涉及任何通信和相互协调，而当客户和服务器都可用时，计算结果就以一条短消息的方式从客户上传到中心服务器。其他一些类似的科学计算任务，如大素数的搜寻，暴力破解密码等，都可以利用类似的方式解决。但是如果想把因特网上的计算资源应用于更广泛的任务，那么就得依靠一个分布式平台，这个平台能够支持数据共享并协调大范围内计算机间的相互协调。这是网格系统的目标，我们将在19章讨论网格。

在本章中，我们只关注现有的在对等网络环境中用于数据共享的分布式系统和算法。在10.2节中，我们将总结Napster的设计并回顾了从中得到的经验。在10.3节中，我们将描述对等中间件层的基本需求。在之后的几节中，我们将介绍对等中间件平台的设计和应用，其中10.4节会给出一个抽象的规约，10.5节对两个已经完全开发好的例子加以详细描述，10.6节将给出这两个例子的一些应用。

10.2 Napster及其遗留系统

对等系统的第一个应用是数字音乐文件的下载，在这个应用中出现了在全球范围内存储信息和获取信息服务的需求。Napster文件共享系统[OpenNap 2001]为用户提供了共享文件的手段，它也第一次向人们展示了对等解决方案的必要性和可行性。Napster自1999年出现以后，很快在音乐文件交换领域得到广泛应用。高峰的时候，有几百万注册用户，有几千人同时交换音乐文件。

Napster的体系结构包括中心文件索引，但是文件由用户提供。这些文件存储在用户的个人计算机上，并且能够被访问。图10-2中的步骤说明了Napster操作的方法。注意在第5步，客户把他的计算机上可用的音乐文件以一个链接的方式传送到Napster文件索引服务器上，这样他就把自己的音乐文件添加到共享资源池中了。从中可以看出，Napster的动机和成功的关键就是通过因特网向用户提供一个巨大的、广泛分布的、对用户可用的文件集，通过提供对“处于因特网边缘上共享资源”的访问，它也实现了Shirky的宣言。



图10-2 Napster：利用一个集中式的、可复制的索引的对等网络文件共享

由于Napster在某些方面（例如，数字音乐）涉及相应的版权问题，它的服务提供者遭到了版权拥有者的起诉，最终由于法律原因，Napster被迫关闭（参见下面的“对等系统与版权归属问题”）。

对等系统与版权归属问题

Napster的开发者认为他们不应承担侵害所有者版权的责任，因为他们没有参与拷贝过程。拷贝过程是在用户的机器之间完成的。但是最终他们败诉了，因为索引服务器被认为是拷贝过

程的本质部分。既然索引服务器的地址是众所周知的，那么它们的运作者就不能保持匿名，结果他们就成了法律诉讼中的目标。

一个更完全的分布式文件共享系统也许可以更好地摆脱法律责任，把法律责任分散给所有的Napster用户，这将使得法律索赔变得异常困难，甚至是不可能的。无论怎样从共享版权保护角度看文件拷贝的合法性，但在一些应用上下文中，匿名的客户和服务器是有其社会、政治的合法性理由的。当要应付审查机构的审查和在一个社区或组织中保持个人言论自由时，匿名性就成为最好的手段。

众所周知，在社会政治危机时代，Email和Web站点在获得公众认知方面，扮演了重要角色；如果作者可以以匿名方式获得保护，那么他们扮演的角色还可以获得进一步的延伸。Whistle-blowing是一个相关的例子：Whistle-blower是一个雇员，他可以不暴露身份而向上级主管部门举报他们的雇主所做的坏事，从而不必害怕雇主们的制裁或者被他们解雇。在某些环境下，通过匿名性来保护这类行为是合理的。

如何使共享数据以及其他资源的访问者和提供者具有匿名性是对等系统设计者关心的一个方面。在多节点系统中，资源请求和结果返回的路由可以足够曲折，从而隐藏它们的来源；文件的内容也可以分布在多个节点上，使得资源的可用性进一步扩大。抵抗多数流量分析机制的匿名通信机制已经存在[Goldschlag et al. 1999]。如果在文件存储到服务器之前对它们加密，那么服务器的所有者看似可以拒绝任何内容。但是这些匿名技术增加了资源共享的开销，并且最近的研究表明：在应付网络攻击方面，匿名性非常脆弱[Wright et al. 2002]。

Freenet[Clarke et al. 2000]和FreeHaven[Dingledine et al. 2000]项目都强调提供因特网文件服务，该服务能为文件的提供者和用户提供匿名性。Ross Anderson推荐使用Eternity Service [Anderson 1996]，它是一项存储服务，通过避免各种灾难性的数据丢失和拒绝服务攻击，提供长期的数据可用性保障。对于可印刷的信息，出版图书是一种永久不变的模式（事实上，一旦图书出版并且已经被分发到世界上各个组织机构的图书馆中，我们将不可能对它进行删除操作了）；可是对于电子出版物来说，它不太可能像图书出版一样受到来自审查机构的各种管制审查。因此，Anderson认为这样的服务是必要的。为了确保存储的一致性，Anderson提出了技术上以及经济上的需求，并且还指出匿名性是必要的，因为它能够防备法律起诉，同样它还能避免非法操作，如贿赂或者攻击数据的创造者、拥有者或者持有者。

402 ~ 403

从Napster中得到的经验 Napster展示了构造一个有用的大规模服务的可行性，该服务依靠几乎整个因特网上普通用户的数据和计算机。为了避免单个用户（例如，第一个提供chart-topping歌曲的用户）的计算资源和网络连接的拥塞，当给一个查询歌曲的客户分配服务器时，Napster将网络地理位置也考虑进来（客户与服务器之间的跳数）。这种简单的负载分配机制使得服务可以伸缩，从而满足大量用户的需求。

局限性：Napster为所有可用的音乐文件建立一个（可复制的）统一索引。对这种应用来说，并不强烈要求保持副本之间的一致性，所以不会影响服务性能。但是对其他应用，这种方法还是有局限性的。除非数据对象的访问路径是分布的，否则对象的发现和定位将可能变成系统的瓶颈。

Napster利用文件共享应用的下列特征并针对这些特征进行设计：

- 音乐文件的内容从来不会被更新，避免了文件在更新后与其副本之间保持一致性的需求。
- 不需要保证单个文件的可用性——如果一个音乐文件暂时不可用，那么用户可以以后再下载。这就减少了对用户计算机和网络连接的依赖性。

10.3 对等中间件

在设计对等应用时，一个关键问题就是提供一个良好的机制，它能够保证客户无论处于因特

网的哪个位置都能快速、可靠地访问数据资源。为此，Napster通过维护可用文件的统一索引来提供文件所在的主机的网络地址。第二代对等文件存储系统（如Gnutella和Freenet），采用了分区和分布式索引算法，不过不同的系统其算法各有不同。

在出现对等范型之前，在某些服务中就存在定位的问题。例如，Sun NFS借助每个客户端的虚
404 拟文件系统抽象层来解决这个问题，它根据虚拟文件引用（例如v节点，见8.3节）来处理对文件的各种请求，这里被请求的文件可能位于多个服务器上。这种解决方案要在文件分布模式或者服务器改变时，在客户端做大量的预配置工作和人工干预。显然，这样的服务伸缩性差，仅局限于由单个组织管理的服务。AFS（见8.4节）也具有类似的特点。

对等中间件系统用于满足被对等系统和应用管理的分布式对象的自动放置及其定位需求。

功能性需求 对等中间件的功能是简化跨越多主机的服务的构建，这些主机可能位于大规模的分布式网络上。为了实现这个目标，它必须能够使客户可以定位单个资源（对相应的服务来说，该资源是可用的）的位置并和该资源通信，即使这些资源分布在多个主机上。其他重要的需求还包括：能够随意地添加新资源或者删除旧资源；能够添加主机或删除主机。与其他中间件一样，对等中间件应该能够向应用程序员提供一个简单的编程接口，该编程接口不应依赖于应用操纵的分布式资源的类型。

非功能性需求 为了使系统可以高效地运行，对等中间件必须解决以下非功能性需求[cf.Kubiatowcz 2003]：

全球可伸缩性：对等应用的一个目标就是利用因特网上大量主机的硬件资源。因此对等中间件应该支持能够访问存放于数万台主机上的数百万计的资源。

负载平衡：当所设计的系统使用了大量计算机时，它的性能将依赖于工作负载的均衡分布。对于我们正在考虑的系统，可以通过把资源放置在随机的节点中以及增加频繁使用的热门资源的副本数量来实现负载均衡。

优化相邻节点间的本地交互：节点之间的“网络距离”对于单个交互（如客户请求访问资源）的时间延迟有很大的影响，而且对于网络流量也会有影响。对等中间件应该能够将资源放在经常访问它们的节点的附近。

能够适应动态主机的可用性：大多数对等系统都允许主机在任何时候自由地加入或退出系统。对等系统中的主机和网段不可能专属于一个组织机构，因此它们的可靠性和能否持续参与提供服务也不能得到保证。构建对等系统的一个主要挑战是：尽管有上述的不利因素，系统仍能够提供可靠的服务。当主机加入系统的时候，这些主机必须集成到系统中，并且负载必须重新分布，从而利用新加入的主机的资源。当主机自愿（或非自愿地）退出系统时，系统必须能够检测到它们退出，并且能够重新分配负载和资源。

对对等应用和系统（如Gnutella和Overnet）的研究表明：参与到系统中的主机有时可用有时不
405 可用[Saroiu et al. 2002，Baghwan et al. 2003]。Overnet对等文件共享系统在因特网上拥有85 000台活动主机。Baghwan等在七天时间内从系统中随机抽取1468台主机，测得它们的平均会话时间为135分钟（中值为79分钟），其中有260～650台主机在任何时候都是可用的（会话表示一段时间，在这段时间内主机是可用的，没有自愿或者非自愿地退出系统）。

另一方面，微软的研究者们从连接到微软公司网络上的主机中随机地抽取了20 000台计算机，测得它们的平均会话时间为37.7小时，其中有14 700～15 600台主机在测试期间一直是可用的[Castro et al. 2003]。上述测试基于Farsite对等文件系统的可行性研究[Bolosky et al. 2000]。这些研究所获得的数字有巨大的出入，这主要是因为个人因特网用户和公司网络用户（例如，微软公司）在行为和网络环境方面有所不同。

能够在具有不同信任体系的环境下保持数据的安全性：在一个全球范围中的分布式系统中，参与其中的主机有着不同的归属，信任体系必须通过利用授权和加密机制来建立，从而确保信息

的完整性和保密性。

匿名性、可否认能力和对审查机构的抵抗：我们注意到（见上面关于版权的讨论），给予数据的持有者和接收者以匿名性时，在要求避免审查机构审查的场合会涉及合法性问题，因此，一个相关的技术需求就是能够保证数据的提供者或持有者能够合理地推卸责任。对等系统存在大量主机，这有助于获得上述性质。

综上所述，设计一个对等中间件层来支持全球规模的对等系统是一个难题。可伸缩性和可用性需求使得在系统中所有的客户节点上维持一个数据库来提供所有感兴趣的资源（对象）的位置是不可行的。

对象的位置信息必须进行分区并且分布于整个网络上。每个节点都负责维护名字空间中的一部分节点位置信息和对象位置信息，还应该对整个名字空间的拓扑结构有一个整体上的了解（参见图10-3）。在面对主机不稳定的可用性和时断时续的网络连接的时候，这些知识的大量复制对确保系统的可靠性是必要的。在下面我们将描述的系统中，其常用的重复因子高达16。

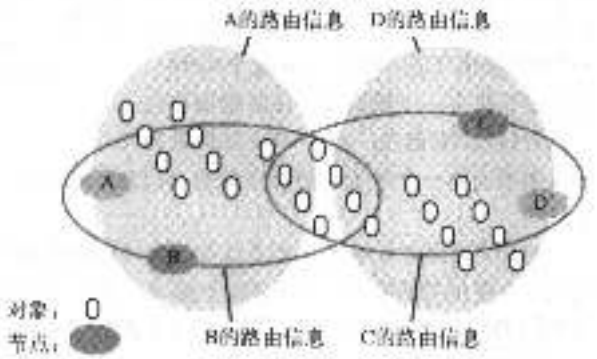


图10-3 在路由覆盖中的信息分布

10.4 路由覆盖

开发满足上述需求的对等中间件是一个非常活跃的研究主题，现在已经开发出几个重要的中间件系统。在本章中，我们将详细描述其中的两个。

*路由覆盖*是一个著名的分布式算法，它负责定位主机节点和对象。顾名思义，中间件层就是中间件形成一层，该层负责把来自客户的请求路由到持有相应数据资源的主机上，热门的对象可放到系统中，然后将它重新分布到其他节点中，这个过程不需要任何客户的参与。它之所以称为覆盖，是因为它在应用层实现了路由机制，但这个路由机制不同于部署在网络层的路由机制（如IP路由）。这种管理和定位复制对象的方法首先在Plaxton等[1997]的具有突破性的论文中被分析，并证明在含有大量节点的网络中该方法是高效的。

路由覆盖通过一系列的节点路由请求和利用每个节点关于目标对象的知识，确保系统的任意一个节点可以访问任意一个对象。对等系统通常会保存一个对象的多个副本以确保对象的可用性，在这种情况下，路由覆盖会维护所有可用副本的位置信息，并且将客户的请求发送到距离它最近的一个含有相关对象的拷贝的"活动"节点（即没有失效的节点）上去。

用于识别节点和对象的GUID是一个9.1.1节中提到过的"纯粹"名字的例子。因为GUID在引用对象的时候不会暴露对象的任何位置信息，所以它也称为不透明标识符。

路由覆盖的主要任务如下所述：

1）一个客户希望调用一个对象上的操作，那么他向路由覆盖提交一个请求，请求中包含相应对象的GUID，路由覆盖把这个请求路由到一个含有该对象副本的节点上。路由覆盖还必须能够完成其他一些任务：

2）如果一个节点想要向一个对等服务中添加一个新的对象，那么它可以通过计算得到该对象的GUID并通知路由覆盖，路由覆盖会确保系统中的其他客户能够访问到这个对象。

3）当客户请求从服务中移除对象时，路由覆盖必须使这些对象不再可用。

4）节点（即计算机）可以加入或者退出服务，当一个节点加入服务时，路由覆盖安排这个节点承担其他一些节点的责任。当一个节点退出时（可能是自愿的，也可能是因为系统或网络故障

406
-
407

造成的），它原来承担的责任被分布到其他节点上。

一个对象的GUID是根据该对象的全部或一部分状态通过一个函数计算出来的，这个值很有可能是唯一的。可以通过尝试使用同一GUID来搜索另外的对象来验证唯一性。通常，用一个散列函数（例如，SHA-1算法，见7.4节）来根据对象的值产生对象的GUID。因为使用这些随机产生的分布式标识符来定位和检索对象，所以路由覆盖系统有时也被描述为*分布式散列表*（Distributed Hash Table, DHT），图10-4中的简单形式的API就反映出了这个事实。在这个API中，put()操作提交一个数据项，该数据项和它对应的GUID存储在一起。DHT层负责为该数据项选择一个位置，然后存储它（以及它的副本，以确保可用性），并通过get()操作来对它进行访问。

put(GUID, data)
data被存储到根据该GUID确定的所有负责存储该对象的节点上。
remove(GUID)
删除所有对该GUID的引用和相关联的数据。
value = get(GUID)
从相关节点中返回和该GUID相关联的数据。

图10-4 由Pastry PAST API实现的分布式散列表（DHT）的基本编程接口

图10-5给出了*分布式对象定位与路由*（Distributed Object Location and Routing, DOLR）层提供的一个更加灵活的API。通过这个接口，对象可以存储到网络中的任何位置，DOLR层负责维护对象GUID和包含该对象副本的节点地址之间的映射。对象可能有多个副本，并且同一个GUID存放到不同的主机上，路由覆盖负责把对象的请求路由到最近的可用副本上。

publish(GUID)
GUID可以根据对象（或它的某一部分，比如它的名字）计算得出。
这个函数使得执行publish操作的节点成为与该GUID对应的对象的主机。
unpublish(GUID)
使与GUID对应的对象变成不可访问状态。
sendToObj(msg, GUID, [n])
遵从面向对象的规则，为了访问一个对象，发一个调用消息给它。这个消息可能是为了数据传输而要求打开一个TCP连接的请求，或者也可能是一条包含对象全部或部分状态的消息。最后一个可选的参数[*n*]（如果存在）要求该消息发送给相应对象的*n*个副本。

图10-5 Tapestry实现的分布式对象定位和路由（DOLR）的基本编程接口

在DHT模型中，一个GUID是X的数据项将被存储在一个节点上，这个节点的GUID数值最接近X；该数据项的副本还将存储到 r 个主机上，这些主机的GUID数值最接近X，其中 r 是复制因子，以确保具有较高的可用性。在DOLR模型中，数据对象副本的位置是在路由层外确定的，每个副本的主机地址是通过publish()操作来通知DOLR的。

图10-4和图10-5列出的编程接口都是基于一套抽象的表示，它最早由Dabek等[2003]提出，这也表明到今天为止，大多数对等路由覆盖实现提供的功能非常相似。

路由覆盖系统的设计工作开始于2000年，它一直是非常活跃的研究领域。目前已经成功开发出几个原型并加以评估。对这些原型的评估显示，它们的性能和可靠性足以使它们可以应用到多种生产环境中。在下一节，我们将针对其中的两个原型进行详细的描述：Pastry 实现了一个与图10-4类似的分布式散列表API；Tapestry 实现了与图10-5类似的API。Pastry 和Tapestry都利用了著名的*前缀路由机制*来确定基于GUID值的消息的传递路线。前缀路由在路由中选择每一跳时，通过应用一个二进制掩码，在目标GUID的十六进制形式中，选择具有递增数字的GUID，从而缩小搜索下一个节点的范围（这个技术也同样被应用于IP包的无等级域间路由，见3.4.3节）。

此外，还开发了其他几种路由方案，它们利用对节点之间距离的不同度量来缩小搜索下一跳的范围。Chord [Stoica et al. 2001]根据被选择节点和目标节点的GUID的不同做出选择。CAN

[Ratnasamy et al. 2001]将节点放入d维空间，并使用d维空间中节点之间的距离。Kademlia [Maymounkov and Mazieres 2002]对节点GUID进行异或操作得到的值来表示节点之间的距离。因为异或操作具有对称性而且参与者经常收到（包含在它们路由表中的）相同节点的请求，所以Kademlia可以非常容易地维护参与者的路由表。

GUID的可读性不佳，因此，客户应用必须通过搜索请求或一些以资源的可读名字为输入的索引服务来获得他们感兴趣资源的GUID。理想的情况下，这些索引也以对等方式存储，以此来克服Napster中的集中式索引的弱点。但是在简单情形下，例如音乐文件或者电子出版物的对等下载，可以简单地在Web页面上索引（cf. BitTorrent[Cohen 2003]）。在BitTorrent中，Web页面上的一个索引对应一个存根文件，这个存根文件包含所需资源的详细信息，包括该资源的GUID和跟踪器（tracker）的URL。这里，跟踪器是一个保存愿意提供该文件的计算机的最新网络地址列表的主机。

408 ~ 409

对上述的路由覆盖，读者可能质疑它的性能和可靠性。对于这些问题的解答，我们将在下一节通过描述一些实际的路由覆盖系统来给出。

10.5 路由覆盖实例研究：Pastry和Tapestry

Pastry和Tapestry均采用前缀路由方法。Pastry的设计直接而高效，因此它是一个可供我们研究的很好的例子。Pastry是消息路由的基础设施，它已经被部署到多个应用中，包括PAST[Druschel and Rowstron 2001]和Squirrel，Past是一个档案文件（不可变文件）存储系统，它以分布式散列表形式实现，具有图10-4所示的API。Squirrel是一个对等Web缓存服务，我们将在10.6.1节中描述它。

Tapestry是我们将在10.6.2节描述的OceanStore存储系统的基础。相对于Pastry来说，它具有更复杂的体系结构，因为它旨在提供更大范围内的定位方法。我们将在10.5.2节中通过和Pastry比较，对它加以详细描述。

10.5.1 Pastry

Pastry[Rowstron and Druschel 2001，Castro et al. 2002，FreePastry project 2004]是一个具有我们在10.4节中所列特点的路由覆盖系统。能够通过Pastry访问的所有节点主机和对象都被分配了一个128位的GUID值。每个节点都有一个公钥，将一个安全的散列函数（例如SHA-1，参见7.4.1节）应用到这个公钥上，通过计算便可以得到节点对应的GUID。而对于对象来说（例如文件），我们可以将安全的散列函数应用到对象名字或者它们的部分存储状态上，从而获得它们的GUID。最终的GUID具有安全散列值的常见特性，也就是说，它们随机分布到区间$[0, 2^{128}-1]$上。这些值不会提供任何有关生成该值的对象或节点的线索，同样不同的节点或对象之间的GUID发生冲突的可能性也是极低的。（如果真的发生了冲突，Pastry也能检测到并采取补救措施。）

在一个具有N个参与节点的网络中，Pastry路由算法能够在$O(\log N)$步内正确地将消息路由到任何GUID对应的地址上。如果GUID标识的节点当前处于活跃状态，那么消息将直接发送给这个节点；否则，消息将发送给最接近于该GUID且处于活动状态的节点。处于活动状态的节点负责处理发往它们邻近节点的请求。

在路由过程中，使用底层传输层协议（一般是UDP）将消息传输到一个最接近目的地的Pastry节点上去。但应该注意，这里提到的“接近”是指在这个逻辑空间——GUID构成的空间——中接近。跨越因特网在两个Pastry节点之间传输消息可能需要若干IP跳数。为了尽可能降低传输路径上不必要的风险，Pastry在为每个节点建立路由表时，使用了一个本地性指标，这个指标基于底层网络的网络距离（例如，跳数或者往返的传输延迟）来选择合适的相邻节点。

410

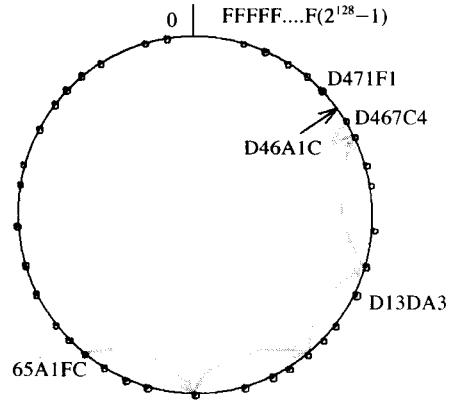
广泛分布的数以千计的主机均可以参与到Pastry的覆盖中，它是完全自治的。当新的节点加入到覆盖中时，它们可以从覆盖中已有成员的$O(\log N)$条消息中获得必要的数据来构建它们的路由表或者其他所需的状态，其中N表示参与到覆盖中的主机的数量。当一个节点失效或者退出时，其他

节点能够检测到它的消失，并且用相似数量的消息共同协作重新配置，以反映路由结构上的变动。

路由算法 完整的路由算法涉及每个节点上路由表的使用以便高效地路由消息，但是为了解释算法，我们用两个阶段来描述路由算法。第一阶段描述一个简化形式的算法，它能够在不用路由表的情况下正确地将消息路由到目的地，但是效率很低。在第二阶段，我们将描述完整的路由算法，它能够将一个请求以$O(\log N)$条消息的代价路由到任何节点上。

第一阶段：每个活跃节点都保存一个邻接集合（leaf set），邻接集合是一个大小为$2l$的向量L，L包含和当前节点GUID接近的$2l$个（l个大于，l个小于）其他节点的GUID和IP地址。当节点加入或者离开网络时候，Pastry负责维护节点的邻接集合。即使一个节点出现故障，仍可以在很短的时间内修正相应节点的邻接集合（故障恢复将在下面讨论）。因此，Pastry系统的不变式是：邻接集合反映了系统的当前状态，即使系统故障超过某个最大故障率时，它们仍能收敛于当前系统状态。

GUID空间是被当作一个环来处理的：比0小的GUID邻居是$2^{128}-1$。图10-6给出了分布在这个环形地址空间上的活跃节点。因为每个邻接集合包含的是与当前节点直接相邻的节点的GUID和IP地址，所以一个Pastry系统若具有正确的邻接集合并且邻接集合的大小至少为2，那么可按下述的方式把消息发送到任意GUID。



圆点代表活跃节点。空间可视为环形：节点0与节点$(2^{128}-1)$相邻。这个图描述了只使用邻接集合信息，从节点65A1FC路由一条消息到节点D46A1C的过程，此处假设邻接集合的大小是8（l=4）。这是一个退化的路由类型，它的伸缩性很差；因此并没有实际使用。

图10-6 单独的环路由虽然正确但低效（基于Rowstron 和Druschel [2001]）

对于任意节点A，当它收到一条目的地址是D的消息M时，它首先将自己的GUID和D的GUID比较，然后再将邻接集合中的GUID和D的GUID比较，最终消息M将发往和D的GUID最接近的节点。图10-6描述了这样的Pastry系统，它的l值是4（Pastry的典型安装中l的值为8）。基于邻接集合的定义，我们可以总结出：在每步中，消息M都将发往比当前节点更接近于D的节点，因此这个过程最终将消息M发送到距离D最接近的活跃节点上去。但是这样的路由方案的效率非常低，在具有N个节点的网络中，发送一条消息需要大约$N/2l$跳。

第二阶段：在算法解释的第二部分，我们将描述完整的Pastry算法，并且说明怎样在路由表的帮助下进行高效路由。

每个Pastry节点都维护一个树型结构的路由表，表中包含一系列节点的GUID和IP地址，这些GUID的值可能是2^{128}范围内任意一个值，其中当前节点的邻近节点在空间上密度更大。

图10-7给出了某个节点的路由表的结构，图10-8描述了路由算法的过程。路由表是按下述方式构造的：GUID值以十六进制表示，路由表依据GUID的十六进制数的前缀的不同对其进行分类。路由表的行数和GUID的十六进制表示的位数相同，因此对于我们正在描述的这个Pastry系统原型来说，路由表有128/4＝32行。对任意的行n，包含15项，每项对应于一个可能的第n个十六进制数位（不包括当前节点的GUID的第n个十六进制数位）。表中的每项指向具有相关GUID前缀的多个节点中的一个。

对任意节点A，路由过程都会使用该节点的路由表R中信息和邻接集合L中的信息，并根据图10-9中描述的算法，来处理来自于应用程序的请求和来自于其他节点的消息。我们可以确信这个算法总是能够成功地将信息M发送到它的目的地，因为程序中第1、2、7行执行的操作就是在上面第

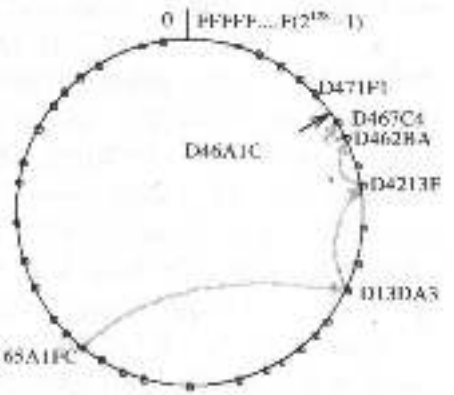
一阶段中描述的操作。我们已经说明，这些操作是一个完整但低效的算法。图10-9中的其他步骤是通过使用路由表信息来减少路由时需要的跳数，从而提高算法的性能。

p =	GUID prefixes and corresponding nodehandles n															
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	n	n	n	n	n	n		n	n	n	n	n	n	n	n	n
1	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6F	6E	6F
	n	n	n	n	n		n	n	n	n	n	n	n	n	n	n
2	650	651	652	653	654	655	656	657	658	659	65A	65B	65C	65D	65E	65F
	n	n	n	n	n	n	n	n	n	n		n	n	n	n	n
3	65A0	65A1	65A2	65A3	65A4	65A5	65A6	65A7	65A8	65A9	65AA	65AB	65AC	65AD	65AE	65AF
	n		n	n	n	n	n	n	n	n	n	n	n	n	n	n

这个路由表位于其GUID值以65A1开头的节点上。数字都是十六进制的。n代表[GUID,IP地址]对，将与消息目标地址对应的GUID具有的相同前缀的[GUID,IP地址]对作为该消息的下一跳。灰色实体中的GUID和当前GUID匹配的前缀长度最多为p，应该检查下一行或者邻接集合来确定路由。虽然在一个路由表中最多有128行，但是在一个具有N个活跃节点的网络中，平均只有$\log_{16}N$行会被填充。

图10-7 一个Pastry路由表的前四行

当D不处于当前节点的邻接集合的范围内时（$D>L_i$或者$D<L_{-i}$），并且路由表中相关项可用时，程序中的4－5行会被执行。在当前节点选择下一跳时，需要从左向右比较节点D和当前节点A的十六进制形式的GUID，然后确定出 p，p表示D和A最长公共前缀的长度。当要访问路由表中的元素时，p将作为路由表的行偏移量，D与A第一个不同的十六进制位（从左到右）作为列偏移量。根据路由表的构造方式，我们不难得出，该元素如果不为空，那么它包含了一个节点的IP地址，该节点的GUID与节点D的GUID有长为$p+1$的公共前缀。



从节点65A1FC路由消息给D46A1C。在结构良好的路由表的帮助下，最多通过$\log_{16}N$跳便可将一条消息成功传送到目的地。

图10-8 Pastry路由例子（基于Rowstron和Druschel [2001]）

如果D落在了邻接集合之外（$D>L_i$或者$D<L_{-i}$）并且相应的路由表单元为空时，执行程序的第7行。出现这种情况的概率是非常小的，只有当相应节点失效而路由表未来得及更新的时候，才可能出现这种情况。不过，当这种情况出现时，算法扫描邻接集合和路由表，然后选择一个节点作为发送消息的下一跳，该节点的GUID应该最接近目标节点D的GUID，并且具有长度为p的公共前缀。如果这个节点包含在L中，那么我们可以按照图10-6中描述的第一阶段的过程来操作。如果这个节点包含在路由表R中，那么该节点的GUID一定比L中的任何节点的GUID更接近于D的GUID，因此它是对第一阶段改进。

主机加入 新的节点加入时，使用了一种加入协议，以便获得它们的路由表和邻接集合，并且新节点向其他节点通知这一变化使它们更新自己的路由表。首先，要加入的新节点计算出一个GUID（通常对节点的公钥应用SHA-1散列函数而获得），然后和附近的一个Patry节点建立连接（这里我们使用的“附近”这个词是指网络距离，即较少的网络跳数或两点之间通信延迟很小。见下面关于最近邻居算法的介绍）。

要处理目标节点是D的消息M（其中$R[p, i]$是路由表中的第p行第i列的元素）：

```
1. If (L_-i < D < L_i) { // the destination is within the leaf set or is the current node.
2.     Forward M to the element L_i of the leaf set with GUID closest to D or the current
       node A.
3. } else { // use the routing table to despatch M to a node with a closer GUID
4.     find p, the length of the longest common prefix of D and A. and i, the (p+1)th
       hexadecimal digit of D.
5.     If (R[p,i] ≠ null) forward M to R[p,i] // route M to a node with a longer common
       prefix.
6.     else { // there is no entry in the routing table
7.         Forward M to any node in L or R with a common prefix of length i, but a
           GUID that is numerically closer.
       }
   }
```

图10-9 Pastry的路由算法

假设新节点（要加入的节点）的GUID是X，并且它所联系的附近节点的GUID为A。节点X发送一个专门的join请求消息给节点A，并且这个消息的目标地址被设为X。节点A按正常的方式通过Pastry分发join消息。Pastry将会把join消息发送到那些其GLID值与X接近的已有节点上去；我们不妨把这些节点称为Z。

节点A、Z以及所有在路由join消息路途上的节点（如B、C…），它们会在常规Pastry路由算法中加入一步，这将使它们路由表和邻接集合中的有关信息传递到节点X，然后节点X对这些信息进行检查，再利用这些信息构造自己的路由表和邻接集合。如果有必要的话，节点X在这个过程中还可以从其他节点请求获得一些额外的信息。

为了了解节点X构建它自己的路由表的方式，读者应该注意到路由表的第一行依赖于节点X的GUID值，而且为了使路由距离尽可能的小，构造出来的路由表应该做到尽可能通过邻居节点路由消息。A是X的一个邻居节点，因此节点A的路由表的第一行将是节点X路由表的第一行（X_0）的首选。另一方面，对于节点X路由表的第二行（X_1）来说，节点A的路由表可能与X_1是不相关的了，因为节点X的GUID和节点A的GUID的十六进制形式的第一位并不相同。不过，路由算法可以确保节点X的GUID和节点B的GUID的第一位相同，这也意味着节点B路由表的第二行（B_1）对于X_1（节点X路由表的第二行）来说是首选。相似的，节点C路由表的第三行（C_2）对于节点X路由表的第三行（X_2）来说是首选，其他的行依此类推。

此外，我们回想一下节点邻接集合的性质，注意到既然节点Z的GUID在数值上接近节点X的GUID，那么X的邻接集合应该和Z的邻接集合相似。事实上，理想情况下X的邻接集合与Z的邻接集合只有一个成员不同。因此Z的邻接集合对于X来说是一个足够好的近似，最终通过一系列与邻居节点的交互，这个近似集合得以优化。这将在下面的“容错”部分加以介绍。

最后，一旦节点X按照上面所说的方式建立起它的路由表和邻接集合，它就可以将路由表和邻接集合的信息发送给路由表和邻接集合中的所有节点，相关节点接收到X的信息，然后调整它们的路由表或邻接集合，从而真正将节点X并入当前系统。在一个新的节点加入到Pastry系统的整个过程中需要传送$O(\log N)$条消息。

主机失效或退出 处于Pastry基础设施中的节点可能失效或者没有任何预兆地退出Pastry。当Pastry中的一个节点的（在GUID空间意义上的）直接邻居节点不再与其通信时，便认为这个节点失效了。这时，含有这个失效节点GUID的邻接集合应该得到相应的修正。

当某个节点发现有节点失效时，为了修复自身的邻接集合L，它应该在L中寻找靠近失效节点的某个“活动”节点，然后从这个邻居节点中获得邻接集合L'的一份拷贝，L'中包含一部分与L重叠的GUID，其中有一个合适的代替失效节点的节点。其他的相邻节点也会收到有节点失效的通知，这些节点也会执行类似的操作，以修复它们的邻接集合。这个修复过程能够保证节点的邻接集合

可以得到修复，除非节点的l个相邻节点同时失效。

对路由表的修复基于“一旦发现”机制。当路由表中某些项所指向的节点失效的时候，消息仍可以通过同一行上其他项继续路由工作。

最近邻居算法

一个新节点要加入Pastry时，它至少应该知道Pastry中已有的一个节点的地址，不过这个已有的节点与新节点不必是相邻的。为了使新节点知道邻近节点的地址，Patry包含了一个“最近邻居”算法，它保存当前已知最近的节点，然后定期地给包含在当前最近节点的邻接集合中的节点发送探测消息，然后根据响应延迟来判断是否有比当前节点更近的节点，通过这个递归的过程，便可以使新节点找到它的邻近节点。

地域性 Pastry路由结构是高度冗余的，即在每对节点之间有许多条路由。构建路由表的目的就是利用大量冗余来减少消息转发的次数，它利用了低层传输网络（通常是因特网节点的一个子集）节点的地域属性。

我们回想一下，路由表中的每一行包含16项。第i行包含16个节点地址，将它们的GUID与当前节点的GUID相比较，它们前$i-1$个十六进制位与当前节点是相同的，而第i个十六进制位分别取可能的值。一个填充良好的Pastry路由覆盖包含的节点要比某个节点路由表中包含的节点多得多。当构建一个新的路由表的时候，都需要按照最近邻居选择算法 [Gummadi et al. 2003]在几个（从其他节点提供的路由信息中获得的）候选节点中做出选择。通常根据节点之间地域距离（IP跳数或通信延迟）来比较候选节点，最后选中最近的且可用的候选节点。因为可用的信息并不够全面，因此这种机制不能产生全局最优的路由。但是实验，显示这个路由平均只比最优路由多用时30%～50%。

容错 按照上面的描述，Pastry路由算法假设路由表中所有项和邻接集合对应的节点都是“活动”节点，且具有功能正常。所有节点都会发送“心跳消息”（即按固定时间间隔发送的消息，用来向其他节点表明发送消息的节点是“活动”节点）给自己邻接集合中的邻居节点。但是以这样的方式检测到的关于某个节点失效的信息并不能很快地发布给其他节点，从而消除路由错误。而且，这种方式也不能避免某些恶意节点试图干扰正确的路由。为了解决这些问题，依靠可靠消息传送的客户希望使用具有“至少一次”语义的传送机制（见5.2.4节），在没有收到响应时，重复发送消息。这样可以使得Pastry获得更长的时间窗口来检测和修复节点失效。

为了处理其他的故障或对付怀有恶意的节点，可在图10-9描述的路由选择算法基础上引入小范围的随机性。要点是对图10-9所示程序的第5行进行一下修改，随机地选择一小部分实例，它们具有公共的前缀，但是长度小于最大长度。这将导致可能使用路由表中靠前的行来路由，尽管这样的路由不够优化，但是它不同于前述的算法的标准版路由。通过在路由算法中使用这个随机变化，即使有少量的恶意节点存在，客户重传最终也可以获得成功。

可靠性 Pastry的作者已经开发出了一个更新的版本，叫做MSPastry[Castro et al. 2003]，它仍然使用同样的路由算法和相似的主机管理方法，但是它包含一些额外的保障可靠性的措施，并对主机管理算法的性能进行了优化。

411～416

保障可靠性的措施包括在路由算法中的每一跳都使用确认。如果发送消息的主机在指定的时间内没有收到相应的确认，那么它将选择另一个路由来重发这条消息。没有成功发送确认消息的节点，将被标记为可疑的失效节点。

为了探测到失效的节点，每个Pastry节点会定期发送心跳消息给处于邻接集合中左部的（即该节点的GUID比当前节点的GUID小）直接邻居节点。每个节点同时记录上一次从右部邻居节点（即该节点的GUID比当前节点的大）收到心跳信息的时间。如果从上次收到心跳消息到现在的时间间隔超出一个时间阈值，则探测节点将开始路由表的修复过程，它会联系邻接集合中的其他节

点，告知它们某个节点失效了，并且发出一个关于建议替代节点的请求。就算多个节点同时失效，当这个过程结束时，所有和失效节点有关的节点都将有一个新的邻接集合，其中包含和当前节点GUID最接近的l个“活动”节点。

我们已经看到，路由算法在只使用邻接集合时仍具有正确的功能，但是维护一个路由表对于提高性能来说是非常重要的。路由表中的可疑的失效节点可以被探测，其方式类似于可疑的失效节点处于邻接集合中的情况。探测时，如果可疑的失效节点没有响应，那么路由表中相关项包含的节点将被另一个合适的节点（从附近节点中获得）替代。另外，可以使用一个简单的*闲谈协议*（见14.4.1节）来定期在节点之间交换路由表信息，从而修复路由表失效的项，并避免地域特性的缓慢退化。闲谈协议每隔20分钟运行一次。

评估工作 Castro和他的同事对MSPastry进行了详尽的性能评估，他们的目的就是确定主机加入/离开率以及相关的可靠性机制对性能和可靠性的影响 [Castro et al. 2003]。

评估的方法是：有一个模拟器，它运行在一台计算机上，并且能够模拟大量的网络主机；在这个模拟器控制下，再运行MSPastry系统，消息传递由模拟的传输延迟替代。这个模拟试验实际上是根据现实应用中的参数而模拟的主机加入/离开行为和IP传输延迟。

MSPastry所有的可靠性机制都包含于该模拟中，并设置了实际的探测与心跳消息周期。通过将数据与实际负载（在52个节点的内部网络上运行了MSPastry）下的测量结果比较，模拟实验的有效性也得到了验证。

在这里我们只总结他们获得的关键结果。

可靠性：如果IP消息的丢失率是0%，那么在100 000个请求中，MSPastry只有1.5个请求没有成功传送到目标主机（可能是因为目标主机失效了），其他所有的请求都被正确传递到目标节点。

如果IP消息的丢失率为5%，那么MSPastry在100 000个请求中，大约有3.3个请求被丢失，另外大约有1.6个请求被传送到错误的节点。通过在MSPastry路由中的每一跳使用确认机制，可以保证所有丢失的或者被传错目的地的消息都能够被重发并最终到达正确的目标节点。

性能：评估MSPastry性能的指标被称做*相对延迟惩罚*（Relative Delay Penalty, RDP）[Chu et
417 al. 2000]或者*扩展度*（stretch）。RDP直接度量发生在覆盖路由层的额外开销。它是两个量之间的比值；第一个量是通过路由覆盖在两个节点发送一个请求的平均延迟，第二个量是在同样两个节点之间使用UDP/IP发送同一个消息的平均延迟。使用模拟负载，MSPastry观测到的RDP值在网络消息丢失率为0%情况下约为1.8，在网络消息丢失率为5%情况下约为2.2。

开销：对每个节点来说，每分钟内*控制流量*（指用来维护节点的邻接集合和路由表的一系列消息）产生的额外网络负载少于2条消息。由于存在初始安装开销，当会话时间少于60分钟时，RDP和控制流量都会显著增长。

总的来说，在有数以千计节点运行的真实环境中，具有高性能、高可靠性的路由覆盖层是可以被构建出来的。即使在平均会话时间少于60分钟和网络错误率很高的情况下，系统也只是有适度的退化，仍能够提供高效的服务。

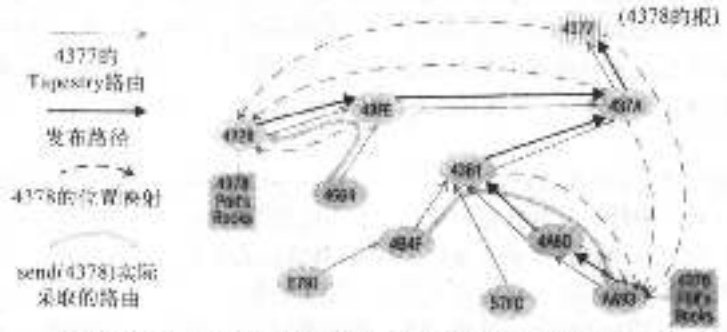
10.5.2 Tapestry

Tapestry实现了一个分布式散列表，并基于和资源相关的GUID，使用和Pastry类似的前缀路由方式将消息路由给节点。但是从应用层面上看，Tapestry的API把分布式散列表隐藏在了类似图10-5所示的DOLR接口后面。持有资源的节点使用publish(GUID)原语来告知Tapestry它们的存在，然后持有资源的节点仍负责存储这些资源。含有相同资源副本的节点使用相同的GUID来发布该资源，这使得Tapestry的路由结构中有多个路由项。

这给予了Tapestry应用一些额外的灵活性：它们可以把资源副本放到经常使用该资源的用户附近（按网络距离），从而降低延迟并最小化网络负载，还能够确保对网络和主机故障的容错。但是，

这并不是Pastry和Tapestry最本质的区别：让与GUID对应的对象作为更复杂的应用级对象的代理，Pastry应用也能够获得类似的灵活性；Tapestry也可以按DOLR API[Dabek et al. 2003]来实现一个分布式散列表。

在Tapestry中，使用160位的标识符来引用对象和执行路由操作的节点。标识符要么是标识实施路由操作的计算机的NodeId，要么是标识对象的GUID。对于任何GUID为G的资源来说，它们都具有唯一的根节点，并且这个根节点R_G具有的GUID是最接近于G的。持有资源G副本的主机H定期调用publish(G)，以确保新加入的主机能够获知G的存在。在每次调用publish(G)时，一个发布消息都会从调用者节点路由到节点R_G。一旦节点R_G收到该发布消息，它就为它的路由表添加一项(G，IP_H)，该项代表G和发送发布消息主机的IP地址之间的映射。路由该发布消息时，所经过的每个节点都会缓存这个映射。我们在图10-10中说明了这个过程。当节点存储了GUID为G的多个映射(G，IP)时，它会按照当前节点到这些IP地址的网络距离（往返时间）来对这些映射排序。随后发往该对象的消息，会在所有可用的对象副本中选择一个最近的作为消息的目标地址。



文件Phil's Book（G=4378）的副本存储在节点4228和节点AA93中。对于对象4378，节点4377是其根节点。显示的Tapestry路由是路由表中的一部分路由项。发布路径给出了发布消息后形成的路由，在这个路径上留下了对象4378的位置映射被存。位置映射被下次会用于路由发送给4378的消息。

图10-10 Tapestry的路由（来自[Zhao et al. 2004]）

Zhao等[2004]详细地描述了Tapestry路由算法，也给出了在遇到节点加入和退出的时候Tapestry路由表的管理。他们的论文还包括了详尽的性能评估数据，这些数据是基于对大规模Tapestry网络的模拟而得出的，这些数据表明Tapestry具有和Pastry相似的性能。10.6.2节描述的OceanStore文件存储就是在Tapestry上构造和部署的。

10.6 应用实例研究：Squirrel、OceanStore和Ivy

大规模对等系统现在还不是主流的技术。它们被广泛部署在为终端用户提供文件下载的应用中，这样的系统包括Napster、Freenet、Gnutella、Kazaa和BitTorrent，但是这些系统都没有单独的路由覆盖层，因此这些系统的性能评估结果很难用于推断其他的系统。

在前面描述的路由覆盖层已经被用到几个应用试验中，所形成的几个应用已经被广泛地评估。我们从其中选择三个做进一步的研究，这三个系统是基于Pastry的Squirrel Web缓存服务、OceanStore和Ivy文件存储系统。

10.6.1 Squirrel Web缓存

Pastry的作者已经开发出了用于个人计算机组成的局域网的对等Squirrel Web缓存服务[Iyer et al. 2002]。在中等规模和大型的局域网中，Web缓存服务通常由一台专门的服务器或者一个集群来

提供。Squirrel系统利用局域网中桌面计算机的存储和计算资源也能够完成同样的任务。我们首先概述一个Web缓存服务的运作原理，然后介绍Squirrel的设计，并回顾一下它的效率。

Web缓存 Web浏览器为因特网对象（HTML页面、图像等）产生HTTP GET请求。该请求获得服务的方式可能有多种：首先，客户端的浏览器缓存可以为该请求提供服务；再者，一个*代理Web缓存*——它是一个服务，运行在同一个局域网内的另一台计算机上或者在因特网上一个邻近的节点上——可以为该请求提供服务；再者，*源Web服务器*——这个服务器的域名包含在HTTP GET请求的参数中——可以为该请求提供服务。最终选择哪一种方式来提供服务，取决于哪种方式能够提供该对象的最新拷贝。每个本地缓存和代理缓存都包含一个最近检索的对象集合，这个集合按URL进行组织以提供快速的查询。另外，有些对象是不可缓存的，因为它们是由服务器根据请求动态产生的对象。

当浏览器缓存或代理Web缓存收到一个GET请求时，会有三种可能：被请求的对象是不可缓存的；被请求的对象不在缓存中；被请求的对象在缓存中。当出现前两种情况时，GET请求会被转发给源Web服务器。当在缓存中找到了被请求的对象时，还必须检测该对象是否是最新的。

存储在Web服务器和缓存服务器中的对象带有一些额外的元数据项，其中包括一个时间戳，这个时间戳给出该对象*最后被修改的日期*T。元数据项中可能还包含该对象的*生存时间*t或者一个eTag（从Web页面的内容计算出的一个散列值）。当一个对象返回给客户时，源服务器都会提供这些元数据项。

对于那些在元数据项中包含生存时间t的对象，只有当$T+t$表示的时间晚于当前时间时，该对象才被认为是最新的。对于那些在元数据项中不包含生存时间的对象，要检测该对象是否最新，将使用一个t的估计值（通常这个估计值是几秒）。如果检测结果是最新的，那么被缓存的对象会直接返回给客户，而不用再联系提供该对象的源服务器。如果结果不是最新的，那么会提交一个带条件的GET请求cGET给下一级进行验证。有两种基本的cGET请求：If-Modified-Since请求（它包含已知的最后一次被修改的时间戳）和If-None-Match请求（它包含一个代表对象内容的*eTag*）。这些cGET请求可能由另外一个Web缓存来提供服务，也可能由源服务器提供服务。收到cGET请求的Web缓存如果没有相应对象的最新拷贝，它就将该请求转发给源Web服务器。对GET请求的应答要么包含整个对象，要么是一条not-modified消息（表示缓存的对象还没有被改变过）。

当从源服务器接收到一个刚刚被修改的可以缓存的对象时，该对象都会被加入本地缓存的对象集合中（如果有必要，可以替换原来的仍然有效的老对象），同时如果存在该对象的时间戳、生存时间t和*eTag*，那么它们也会被同时保持。

集中式代理Web缓存服务已经被部署到大多数支持大量Web客户的局域网中，这种代理服务的
420 运行基础就是我们在上面描述的策略。代理Web缓存典型的实现是运行于一台专门主机上的多线程进程，或者是运行于计算机集群上的进程集合。它们都需要大量专用的计算资源。

Squirrel Squirrel Web缓存服务可以完成同样的功能，而它只需要使用网络中每台客户计算机的一小部分资源。对每个缓存对象的URL应用SHA-1安全散列函数，可以产生一个128位的Pastry GUID。像其他Pastry应用一样，既然GUID不是用来验证对象内容的，因此产生GUID不必依赖于整个对象的内容。Squirrel的作者依据端到端观点（见2.2.1节）给出自己的判断，他认为：一个Web页面从服务器传送到客户要经过很多中间节点，而在每个节点上页面的可靠性都可能遭到破坏，增加对缓存页面的认证对于全面的可靠性保证也只是杯水车薪。对于那些需要可靠性的交互，应该使用HTTPS协议（具有端到端传输层安全性，见7.6.3节）来获得更好的可靠性保证。

在Squirrel最简单的实现中（它已经被证明效率是最高的），如果一个节点的GUID和对象的GUID是最接近的，那么该节点就作为这个对象的主（home）节点，负责缓存和持有所有该对象的拷贝。

每个客户节点都包含一个本地Squirrel代理进程，该进程负责缓存所有本地和远程的Web对象。

如果在本地缓存中没有发现被请求对象的一个最新拷贝，Squirrel就会通过Pastry将一个GET请求或者cGet请求（当本地缓存包含被请求对象的过时版本时）路由到该对象的主节点。如果主节点有该对象的最新拷贝，它就直接给客户返回最新拷贝或者一条not-modified消息。如果主节点包含被请求对象的过时版本或者根本没有该对象的任何拷贝，那么它就分别发送cGet或Get请求给该对象的源服务器。源服务器的响应可能是一条not-modified消息，也可能是被请求对象的拷贝。在前一种情况下，主节点重新验证它的缓存项，并且向客户返回该对象的一个拷贝；在后一种情况下，它将新对象的拷贝转发给客户，并且如果该对象是可缓存的，它还在自己的本地缓存中保存该对象的一个拷贝。

对Squirrel的评估 Squirrel也是通过模拟的方式来评估的，使用的是根据对微软内部和其他真实环境中已有的集中式代理Web缓存的活动进行跟踪而获得的模型化负载。这里，微软的环境中有位于英国剑桥的105个活动客户，另外一个环境在美国华盛顿州的雷蒙德，包含超过36 000个活动客户。评估工作从三个方面比较了Squirrel Web缓存与传统集中式Web缓存的性能差别：

能够节省总的外部带宽：外部带宽的总使用量跟缓存命中率有关，只有当缓存中的对象不能命中时，才会发送请求给外部服务器。对于集中式Web缓存，观测到的缓存命中率是29%（雷蒙德）和38%（剑桥）。使用同一个活动日志，为Squirrel缓存产生一个相似的模拟负载，其中每个客户贡献100MB的磁盘存储，这样观测到的命中率和集中式Web缓存非常相似，分别是28%（雷蒙德）和37%（剑桥）。由此可得出结论：两者节省外部带宽的比例相似。

用户访问Web对象时感觉到的延迟：当一个客户发送请求给负责缓存相应对象的主机（主节点）时，如果使用路由覆盖，那么在局域网中将会有好几条消息被发送（路由跳数）。在雷蒙德的模拟环境中，传送一个GET请求的路由跳数的平均值是4.11。在剑桥的模拟环境中，传送一个请求的路 421
由跳数的平均值是1.8。然而对于访问集中式Web缓存服务来说，需要且仅需要传递一条消息。

虽然跨越因特网传送一条TCP消息大概需要10～100ms，但是在现代以太网硬件的支持下，在局域网中传输一条消息只需几毫秒，这其中还包含了TCP连接建立的时间。因此，Squirrel的作者们认为：一个要访问的对象在缓存中被找到所需要的延迟远远小于缓存中没有满足请求的对象时所需要的延迟。所以，相对于传统的集中式Web缓存，Squirrel缓存也能给用户以相似的体验。

用户节点需要承担的计算和存储负载：在整个评估过程里，每个节点为其他节点提供的缓存请求服务次数是极低的，仅为每分钟0.31次（在雷蒙德的模拟环境中），这说明消耗系统资源的整体比例是极低的。

基于上面描述的测量结果，Squirrel的作者们得出结论：Squirrel的性能与集中式缓存的性能相似。当集中式缓存服务器配备相似大小的专用缓存时，对于Web页面的访问，Squirrel的延迟比集中式Web缓存小一些。Squirrel给客户节点增加的额外负载很少，少到用户可能感觉不到。随后，Squirrel被部署到一个具有52台客户机器的局域网中，作为主Web缓存提供服务，最终的结果证明了他们的结论是正确的。

10.6.2 OceanStore文件存储

Tapestry的开发者已经为对等文件存储设计并建立了一个原型，与Pastry不同，它能够支持可变文件的存储。OceanStore [Kubiatowicz et al. 2000，Kubiatowicz 2003，Rhea et al. 2001，Rhea et al. 2003]的设计目标是提供一个大范围的、可增量伸缩的持久存储工具，以便存储可变数据对象，在网络和计算资源经常变化的环境中，仍提供对象存储的长期持久性和可靠性。OceanStore计划用于多种应用中，包括类似NFS文件服务的实现、电子邮件服务、数据库以及其他涉及大量数据对象共享和持久存储的应用。

OceanStore的设计还包括提供不变数据和可变数据的复制存储功能。受Bayou系统（见14.4.2节）中保持对象与其副本之间的一致性的机制的启发，把它适当的裁剪，就可以满足应用的需要。

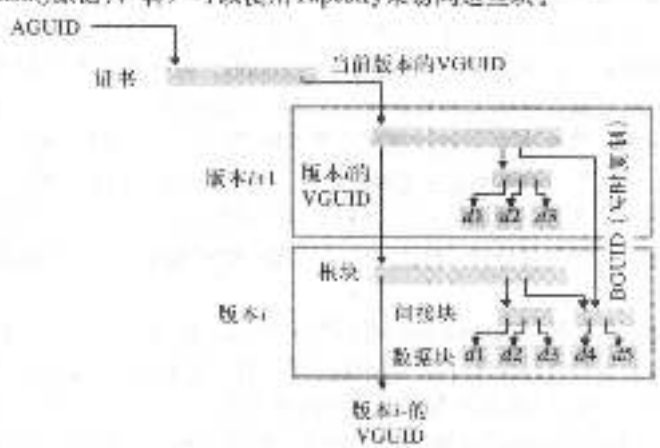
通过对数据进行加密和使用环占据式协定（见11.5.3节和11.5.4节）来更新复制对象，就可以保证数据的私密性和完整性。这样做是有必要的，因为我们不能对单个主机的可信任度做任何的假定。

目前，已经构建了一个OceanStore的原型Pond[Rhea et al. 2003]。它是以支持应用，并且为了验证OceanStore设计的正确性和比较OceanStore在性能上与传统方法的区别，对它的性能用各种基准测试进行了评估。在本节后面的部分，我们将概述OceanStore/Pond的设计，并总结它的评估结果。

Pond使用Tapestry路由覆盖机制把数据块放置在节点上，这些节点可能遍布整个因特网，然后
422 Pond再把请求分发给它们。

存储的组织 OceanStore/Pond的数据存储在一组块中，因此可以把它们比喻为文件。但是每个对象都被表示为若干不变的有序序列，并且原则上这些版本都是永久存储的。对对象的任何一个更新，都将导致该对象生成一个新版本。这些版本可以依据在6.4.2节描述的用于创建和更新对象的写时更新技术共享不变块。因此，如果某些版本之间只有很少的不同点，那么存储这些版本需要的额外空间也是很少的。

对象是按类似Unix文件系统的方式来构造的，它们的数据块的组织和访问都要通过一个元数据块，也叫根块，并且如果有必要还需要额外的间接数据块（cf. Unix inodes）。另外一个级别的间接块被用于将对象一系列版本和一个持久的文本名字或者其他外部可见的名字（例如一个文件的路径名）关联起来。图10-11阐明了对象的组织方式。GUID可以关联到数据对象（这样的GUID也称作AGUID）、数据对象不同版本的根块（这样的GUID也称作VGUID）、间接块和数据块（这样的GUID也称作BGUID）。若某个块有多个副本，它们将被存储到对等节点中，这些节点是根据本地性和存储可用性原则进行选择的，并且这些块的GUID会被每个包含这些副本的节点发布出来（使用图10-5中publish()原语），客户可以使用Tapestry来访问这些块。



注：版本n+1更新的块是d1、d2、d3。证书和根块包含的一些元数据没有显示。所有未标记的箭头都是BGUID。

图10-11 OceanStore对象的存储组织

有三种类型的GUID会被用到，如图10-12所示。前两种GUID通常用来赋给存储在Tapestry中的对象。它们都是根据相关块的内容使用一个安全的散列函数计算得出的，以后可以用它们来认证、验证内容的完整性。因此它们引用的块必须是不可变的，因为一个块的内容的任何改变都会使得用GUID作为内容的验证符变得毫无意义。

名 字	含 义	描 述
BGUID	块GUID	一个数据块的安全散列码
VGUID	版本GUID	某个版本根块的BGUID
AGUID	活动GUID	对象的所有版本的唯一标识符

图10-12 OceanStore中用到的标识符的类型

第三种标识符是AGUID。这些标识符用来（间接地）引用一个对象的所有版本构成的流，它使得客户可以访问对象的当前版本或者任何一个以前的版本。既然存储的对象是可变的，那么用来标识它们的GUID不能依赖它们的内容而产生，因为这样会导致对象内容改变时对象的GUID有关的所有索引信息失效。

解决的方式是，当一个新的存储对象被创建时，创建该对象的客户会为其提供一个应用特定的名字（例如，文件名），对这个名字以及一个代表对象拥有者的公钥（见7.2.5节）应用一个安全的散列函数可以产生出代表该对象的永久AGUID。在一个文件系统应用中，每个文件名都有一个相应的AGUID存储到目录里。

在对象所有版本构成的序列和标识它的AGUID之间存在一种关联，这种关联被记录在一个数字签名证书中。通过使用主拷贝复制方案，（也叫被动复制，见14.3.1节），数字签名证书可以被存储和复制。该证书中包含当前版本的VGUID，并且每个版本的根块都包含有其前一版本的VGUID，因此它们构成了一个引用链，使一个持有相应证书的客户可以遍历整个版本链（见图10-11）。为了确保关联是可信的并且是由授权主体做出的，必须要有数字签名证书。我们也期望客户能够对此进行检查。当创建一个对象的新版本时，就会有一个新的证书产生，它包含了新版本的VGUID、时间戳和一个版本序列号。

对等系统的信任模型要求构建每个新证书时都需要征得一小组主机的同意（将在下面描述），这一小组主机称作内部环。每当一个新的对象被存储到OceanStore中时，都会选择一小组主机作为该对象的内部环。它们使用Tapestry中的publish()原语来告知Tapestry某个新对象的AGUID。随后，当客户请求该对象的证书时，Tapestry就把该请求路由给被请求对象内部环中的一个节点。

新的数字签名证书会替代每个内部环节点上的旧证书，并且它还会被分发到更多的二级拷贝。由客户决定多久检查一次是否有新版本存在（例如，大多数NFS系统在安装时配置成客户与服务器一致以30s的时间窗口运行，见8.3节）。

通常在对等系统中，个人主机被认为是不可信任的。对主拷贝更新需要得到内部环中所有节点的一致同意。它们使用基于拜占庭协定算法（该算法由Castro和Liskov[2000]提出）的状态机来更新对象和对证书进行数字签名。拜占庭协定的使用可以确保证书能够被正确地维护，即使当内部环的某些节点失效或者有恶意行为时，仍能保持正确性。因为拜占庭协定的计算量和通信开销的增长速度与参与的节点主机的数量的平方成正比，因此内部环中的主机数量应保持一个很小值，并且通过使用刚才提到的主拷贝策略，数字签名证书可以被广泛复制。

423 ~ 424

实施一次更新还包括检查访问权限和用其他挂起的写操作序列化这个更新。一旦主拷贝的更新过程完成，通过使用由Tapestry管理的组播路由树，就可将新的结果将发布给二级副本，这些二级副本存储在内部环外的主机上。

由于数据块具有只读特性，因此它们的复制可以采用不同寻常的但具有更有效存储的机制。该机制将每个块分成m个大小相同的段，这些段使用*纠删码*（erasure code）[Weatherspoon and Kubiatowicz 2002]进行编码，最终得到n个段，其中$n>m$。纠删码的关键特性是：有可能从其中的m个段中重新构造出这个块。一个系统如果使用纠删码，当系统缺失的主机数量不多于$n-m$时，所有的数据对象仍是可用的。在Pond实现中，m的值是16，n的值是32，因此会耗费双倍的存储空间，但是该系统在不丢失数据的情况下，最多可容忍16台主机同时失效。可以使用Tapestry来存放和检索存储在网络中的段。

通过从利用纠删码形成的段中重新构造出块，我们可获得系统高容错性和数据高可用性，但是我们也要付出一点代价。为了使影响最小，所有的块都要使用Tapestry存储到网络中。既然块可以通过段重新构造，那么可以把这些块当作一个缓存——这些块不具备容错性，当需要存储空间的时候，可以丢弃它们。

性能 开发Pond的目的不是为了提供一个产品实现，而是为了提供一个原型，以证明大规模对等文件服务是可行的。Pond是用Java语言实现的，并且上面列出的设计也几乎全都实现了。对它

的评估是参考了几个专门的基准测试，并且让OceanStore对象模拟了NFS中的客户与服务器进行的。开发者们根据Andrew基准[Howard et al. 1998]（它能够模拟一个软件开发负载）测试了NFS模拟。测试的结果在图10-13中给出。这些结果是在运行Linux的使用奔腾III 1GHz的PC机上得到的。局域网的测试是在一个千兆以太网上完成的，广域网上的结果是用由因特网连接的两组节点得到的。

阶段	局域网		广域网		基准中的主要操作
	Linux NFS	Pond	Linux NFS	Pond	
1	0.0	1.9	0.9	2.8	读和写
2	0.3	11.0	9.4	16.8	读和写
3	1.1	1.8	8.3	1.8	读
4	0.5	1.5	6.9	1.5	读
5	2.6	21.0	21.5	32.0	读和写
总计	4.5	37.2	47.0	54.9	

数字表示运行Andrew基准测试不同阶段所需的时间（以秒计）。它有5个阶段：（1）递归地构建子目录；（2）拷贝源树；（3）检查树中所有文件的状态，但不检查它们的数据；（4）检查文件内数据的每个字节；（5）编译和链接文件。

图10-13 模拟NFS的Pond原型的性能评估

根据作者们得出的结论，我们可以看出：当OceanStore / Pond运行于一个广域网上（即因特网）时，它的性能在读文件方面明显地超越了NFS，而在更新文件和目录方面，其性能则大约是NFS的三倍。不过，在局域网内运行得到的性能结果比较差。总之，上述结果还是表明，基于OceanStore设计的应用于因特网范围内的对等文件服务，对于变化不是很频繁的分布式文件（例如网页的缓存副本）来说，将会是一个高效的解决方案。但是在广域网内作为NFS的替代品来使用，人们对它的潜力仍有怀疑；在纯局域网内使用时，很明显它就不具有任何竞争力了。

这些结果是在数据块的存储没有使用基于段和副本的纠删码的情况下获得的。公钥的使用对Pond操作的计算开销有实质性的贡献。上面图中的数字针对的是512位公钥的，它的安全性很好，但并不完美的。当使用1024位公钥时，对于涉及文件更新的基准测试阶段，测试结果很差。通过使用专门设计的基准测试，还获得了其他一些结果，包括测量拜占庭协定进程对文件更新延迟上的影响。这些结果都在100ms到10s之间。更新吞吐量的测试最大达到了每秒100次更新。

10.6.3 Ivy文件系统

跟OceanStore一样，Ivy[Muthitacharoen et al. 2002]也是一个支持多用户的读/写文件系统，它也是在一个路由覆盖层和分布式散列地址数据存储上实现的。与OceanStore不同的是，Ivy文件系统模仿了Sun NFS服务器。Ivy将由Ivy客户发出的文件更新请求作为文件的状态存储在日志中；在它的本地缓存不能满足某个访问请求时，它都会扫描这些日志，然后重构这个文件。这些日志记录被保存在DHash分布式散列地址存储服务中[Dabek et al. 2001]。（日志最早出现在8.5节描述的Sprite分布式操作系统[Rosenblum and Ooosterhout 1992]中，用来记录关于文件的更新；但那时它只是用来优化文件系统更新的性能。）

Ivy在设计上解决了一些以前未解决的问题，这些问题源于对部分可信或不可靠的机器中的主机文件的需求。Ivy解决的问题包括：

- 维护文件元数据（例如Unix/NFS文件系统中的节点的内容）的一致性，即使元数据有可能在不同节点并发地更新文件。它没有使用锁机制，因为节点失效或者网络的连通性可能会导致系统无限长时间的阻塞。
- 参与节点之间的部分信任问题和参与者主机易受攻击的问题。要从攻击导致的完整性失效中恢复，应基于文件系统中的视图概念。一个视图表示一个状态，该状态是从一组参与者所作的更新日志中构造出来的。参与者可能从系统中移除，那么视图被重新构造时，便不再包括

被移除的参与者做过的更新。因此一个共享的文件系统被看作是由一组（动态选择的）参与者实施的更新合并在一起得到的结果。 425 ↓ 426

- 允许网络分区期间继续进行操作，这会导致对共享文件更新的冲突，解决更新冲突使用的方法和Coda文件系统（见15.4.3节）使用的方法相似。

Ivy在每个客户节点实现了一个基于NFS服务器协议的API（和8.3节的图8.9列出的一组操作相似）。客户节点包括一个Ivy服务器进程，该进程通过基于键值（GUID，从记录的内容计算出来的散列值，见图10-14）的局域网络或者广域网，能够使用DHash来存储和访问节点上的日志记录。DHash实现了一个类似于图10-4所示的编程接口，并且在一些节点上复制所有内容，以获得更好的灵活性和可用性。Ivy的作者表示，从原理上说，DHash完全可以被其他分布式散列地址存储（如Pastry、Tapestry或者CAN）代替。

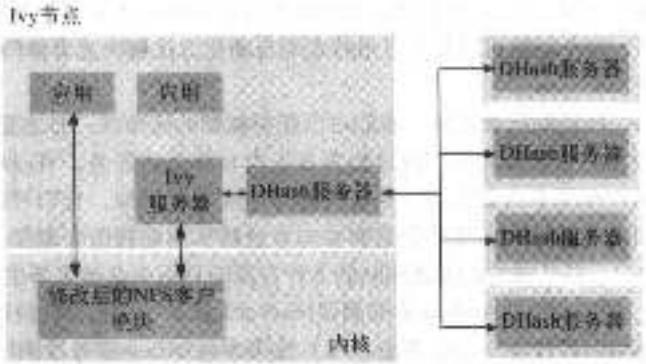


图10-14 Ivy系统体系结构

一个Ivy文件存储由一组更新日志组成，其中每个参与者拥有一个日志。每个Ivy参与者只能在自己的日志中添加内容，但是可以读取组成文件系统的所有日志。将更新被单独存储在各个参与者的日志中，这样当安全性遭到破坏或者出现一致性失效时，这些更新可以回滚。

一个Ivy日志是由一系列日志条目按照时间反序构成的链表。每个日志条目都是带有时间戳的记录，代表用户想要更改文件内容、文件元数据或更改目录的请求。DHash使用记录的160位SHA-1散列值作为键，用来放置和检索该记录。每个参与者也会维护一个可变的DHash块（也叫日志头部），它指向该参与者最近的日志记录。可变的块都由它们的拥有者赋予一个加密的公钥对。这些块的内容会使用一个私钥加以签名，此后就要根据相应的公钥对内容进行认证。当读取多个日志时，Ivy使用版本向量（即向量时间戳，参见10.4节）为所有的日志项进行一个全排序。 427

DHash使用日志记录的内容的SHA-1散列值作为键值来存储该日志记录。使用DHash键值作为链接，将日志记录按照时间戳顺序构成一个链表。日志头部持有最新日志条目的键。为了存储和检索日志头部，日志的拥有者会计算出一个公钥对。公钥的值将作为它的DHash键，而私钥被拥有者用来对日志头签名。任何持有公钥的参与者都可以取得相应的日志头，并且使用日志头来访问日志中所有记录。

假设一个文件系统当前只有一个日志，这时有一个需要从文件中读取一些字符的请求到来，那么对于这个请求，规范的执行方法需要首先对日志进行一系列扫描，以发现日志中有关文件更新部分的记录。日志并没有长度限制，但是当已扫描的日志条目能够满足请求，上述扫描就会中止。

如果要访问一个多用户、多日志的文件系统，规范的算法还包括比较日志记录中的向量时间戳，以确定更新的次序（因为我们不可能假设存在一个全局的时钟）。

就算完成一个简单的read请求，所需的时间也可能是很长的。通过结合使用本地缓存和快照，可以将时间降低到一个可以容忍、可以预测的范围内。快照是文件系统的一个表示，每个参与

者在使用日志时，都会将快照作为一个副产品计算出来，并存储在本地。如果一个参与者被当前系统中拒绝，这些快照可能不再有效，因此从这种意义上说，它们构成了文件系统的一个软表示。

更新一致性是*持续更新*（close-to-open）的；也就是说，一个应用程序对一个文件上实施更新的过程不会被其他进程看到，直到该文件被关闭为止。使用持续更新一致性模型可以确保当文件关闭时，所有在该文件上的write操作都可以保存到客户节点上；然后所有write操作构成的集合被写入一个日志记录，同时产生一个新的日志头部并记录下来（这是对NFS协议的一个扩展，这样可以确保当一个应用程序中的close操作发生时，该事件能够被通知给Ivy服务器进程）。

既然每个节点都有一个Ivy服务器，并且它们都独立地将更新分别存储在单独日志中，每个节点都不会与其他节点相互协作，那么为了将文件的内容构造出来，读取日志时，还必须将存储在这些日志中的所有更新序列化。写入日志记录中的版本向量可以用于大多数更新的排序，但是有可能会出现冲突的更新，这个问题可以通过应用特定的自动化方法解决或者像Coda那样（见15.4.3节）用手工的方法解决。

通过将已经提到的一些机制结合使用，我们可以获得数据的完整性：日志记录是不可改变的，它们的地址就是它们内容的安全散列值；通过检查日志内容的公钥签名，可以校验日志头部。但是信任模型可能会使某些恶意参与者获得文件系统的访问权限。例如，它们可能恶意地删除自己持有的文件。当检测到这类事件时，这些恶意的参与者会被从当前视图中删除；它们的日志不再用于计算文件系统的内容，并且被它们恶意删除的文件在新的视图中会被重新生成。

Ivy的作者们使用一种修改过的Andrew基准测试[Howard et al. 1988]来比较Ivy和标准NFS服务器在局域网和广域网环境中的性能。他们考虑：（1）使用本地DHash服务器的Ivy与单个本地NFS服务器相比，（2）使用位于远程因特网站点的DHash服务器的Ivy与单个远程NFS服务器相比。他
428 们还把性能特征作为一个函数，该函数有三个参数：视图中参与者的数量、同时进行写操作的参与者数量以及用来存储日志的DHash服务器数量。

他们发现，在大部分基准测试中，Ivy的执行时间是NFS的两倍，而对于所有的测试，Ivy的执行时间在NFS的三倍以内。为广域网部署的Ivy的执行时间是为局域网部署的Ivy的执行时间的10倍或更多，不过远程NFS服务器也有相似的比例。关于性能评估的细节可以在Ivy的论文中找到[Muthitacharoen et al. 2002]。但是我们应该注意到，NFS并不是用于广域网的，Andrew文件系统以及其他一些最近开发出来的基于服务器的系统（例如xFS[Anderson et al. 1996]）在广域网部署方面可以提供更高的性能。它们应该具有更好的与Ivy做比较的基础。Ivy最主要的贡献是：在一个部分可信的环境中（对于跨越多个组织和管辖区的大规模分布式系统来说，这样的环境是不可避免的）提供了一个新方法来管理安全性和完整性。

10.7 小结

最早出现的对等体系结构是为了支持大规模的数据共享，例如在因特网范围内使用的Napster以及专门用于数字音乐共享的Napster派生系统。它们的大量使用与版权法相冲突，但这并没有降低它们在技术上的重要性，尽管技术上的一些缺陷限制了它们只能部署在那些不用保证数据完整性和可用性的应用中。

后来的一些研究促进了对等中间件平台的发展，它们能够将请求发送给位于因特网任意位置上的数据对象。对象使用GUID寻址，GUID是一个纯粹的名字，不含有任何IP地址信息。根据每个中间件系统特有的映射函数，数据对象会被放置到相应节点上。数据传送需要使用中间件中的路由覆盖，它维护一个路由表，并且能够沿着路由线路不断转发请求，而路由线路是根据所选择的映射函数计算出相应的距离来确定的。

基于产生GUID的安全散列函数，中间件平台提供了数据完整性保证；基于在几个节点上复制对象以及具有容错能力的路由算法，中间件平台提供了数据可用性保证。

中间件平台已经被部署在几个大规模试验性应用中，对其也进行了改进和评估工作。最近的评估结果表明：该技术已经可以部署在那些包含大量共享数据对象和大量用户的应用中。对等系统的优点包括：

- 具有利用主机中未使用资源（存储资源、处理器资源）的能力。
- 具有很好的伸缩性，可以支持数量众多的客户和主机，在网络链接和主机计算资源方面能获得极好的负载均衡。 429
- 中间件平台的自组织特性使得系统开销很大程度上不依赖于所部署的客户和主机数量。

对等系统的缺点和当今的研究主题包括：

- 当它们应用于可变数据存储时，相对于可信的、集中式服务，它们的开销有点昂贵。
- 期望它们为客户和主机提供匿名性，但是对于匿名性还没有强有力的保证。

练习

10.1 早期的文件共享应用（如Napster）因为需要维护一个集中式的索引资源并且还要维护持有这些索引资源的主机，因此在可伸缩性方面受到了很大限制。关于索引引起的问题，你能想到其他解决方法吗？ （第402页～第404页，第409页，15.4节）

10.2 维护可用资源索引问题是和具体应用相关的。如果你已经给出了10.1题的一些答案，请考虑它们对于下面的应用是否适合：（1）音乐和多媒体文件的共享；（2）需要长期存储的归档材料，例如杂志或报纸内容；（3）通用的可读写文件的网络存储。

10.3 用户希望常规服务器（例如，Web服务器或文件服务器）能够给他们提供哪些方面的保障？ （1.4.5节）

10.4 常规服务器提供的保障有可能由于下述原因遭到破坏：

a）对主机的物理损害。

b）系统管理员或系统管理者的错误或不一致性。

c）对系统软件成功的攻击。

d）硬件或软件错误。

对于上述的每种破坏类型，各给出两个可能发生的事件例子。它们当中哪些违背了信任体系，哪些是犯罪行为？如果它们发生在一台个人电脑上，而该电脑为相关对等服务贡献了一些资源，那么它们算不算违背了信任体系？为什么它们都是和对等系统相关的？ （7.1.1节）

10.5 对等系统通常依赖不可靠的、易变的计算机系统维护它们的大多数资源。作为技术发展的结果，信用已经成为一种社会现象。易变性（即不可预测的可用性）也常常是因为人们自身的行为造成的。详细阐述你在10.4题中给出的答案，并根据下面给出的计算机的属性，讨论几种方式之间可能存在的区别。

- 所有权
- 地理位置
- 网络连通性
- 所属国家或管辖范围

在对等存储服务中，关于放置数据对象的策略，你有什么建议？ 430

10.6 评价你所在的环境中个人计算机的可用性和可信任性。你应该评估以下方面：

正常运行时间：计算机每天正常运行并且连接到因特网的时间。

软件一致性：软件是否由一个称职的技术人员管理？

安全性：计算机能否受到保护，免于被它的用户或其他人篡改？

基于你的评估，讨论在你评估的计算机集合中运行数据共享服务的可行性，并列出在对等数据共享服务中必须考虑的问题。（第405页～第406页）

10.7 解释怎样使用对象的安全散列码来识别对象并将消息路由给它，并且确保它不会受到损害的。这个散列函数应该具有什么样的性质？当很大一部分对等节点失效时怎样维护完整性？（第400页，第423页，7.4.3节）

10.8 经常有人认为，对等系统可以给访问资源的用户或提供资源的主机以匿名性支持。对这些主张进行讨论。提出一种方式，它也许能够改善对匿名性攻击的抵抗力。（第403页）

10.9 路由算法会根据某个地址空间内节点之间的估计距离选择下一跳。Pastry和Tapestry使用的都是环状的线性地址空间，在这个空间中使用一个函数，基于GUID的不同数值来确定节点之间的分离度。Kademlia对GUID进行异或运算。这对于维护路由表有什么帮助？异或运算是否能为距离指标提供合适特性？（第409页，[Maymounkov and Mazieres 2002]）

10.10 当模拟评估Squirrel对等Web缓存服务时，路由一个请求给一个缓存条目，在Redmond流量场景中，平均需要4.11跳；在Cambridge流量场景中，平均需要1.8跳。请解释这个现象，并且说明它能够支持Pastry声称的理论上的性能。（第410页，第421页）

第11章 时间和全局状态

本章将介绍分布式系统中与时间有关的若干问题。时间是一个重要的问题。例如，我们要求全世界的计算机为电子商务交易给出一致的时间戳。时间也是理解分布式运行是如何展开的一个重要的理论概念。但时间又是分布式系统中容易出现问题的方面。每个计算机可以有自己的物理时钟，但时钟通常会有偏离，我们无法使它们完全准确地同步。本章将分析使物理时钟大致同步的算法，然后解释逻辑时钟，其中包括向量时钟。向量时钟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得很难找到分布式程序在执行时的状态。我们通常需要知道在进程B处于某种状态时，进程A所处的状态，但我们不能依靠物理时钟了解在同一时刻什么是真的。本章的后半部分将研究在缺乏全局时间的情况下决定分布式计算中全局状态的算法。 433

11.1 简介

本章将介绍一些基本概念和算法，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时序有关。

在分布式系统中，时间是一个重要而有趣的问题，原因如下。第一，时间是我们想要精确度量的量。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，将计算机的时钟与一个权威的外部时间源同步是必要的。例如，一个“电子商务”事务涉及的事件是在贸易商的计算机和银行的计算机上发生的。为了便于审计，这些事件必须要精确地标记时间戳。

第二，为了解决分布方面的几个问题，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式数据一致性的算法（13.6节将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）以及消除重复更新的算法（参见Ladin等[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而言是一个常量。他从这个假设证明了，若两个事件在一个参照系下是同时的，但对于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，他们的看法就相差更大。

此外，对于两个不同的观察者，两个事件的相对顺序甚至是相反的。但如果一个事件能引起另一个事件的发生，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，但物理效果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而言是相对的，牛顿的绝对物理时间概念是不足信的。在度量时间间隔时，宇宙中没有一个专门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，不能准确记录不同节点上的事件的时间，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同时出现。例如，在面向对象系统中，我们要确定对某一对象的引用是否已不存在，即是否对象已经变成无用单元（这时我们能释放它的内存）。做出以上判断需要观察进程的状态（找出它们是否包含引用）和进 434

程之间的信道（万一包含引用的消息正在传送过程中）。

在本章的前半部分，我们将分析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，其中包括向量时钟。向量时钟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。

本章的后半部分将描述一些算法，这些算法用于捕获分布式系统在运行时的全局状态。

11.2 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。我们将精化该模型，以帮助大家理解如何随系统的执行描述系统的演化，如何给系统执行过程中用户感兴趣的事件打时间戳。我们将从如何给一个进程中发生的事件排序和打时间戳开始。

设一个分布式系统由N个进程 p_i ($i = 1, 2, \cdots, N$)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18章将考虑共享内存的进程）。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，还包括在它影响的本地操作系统环境中的对象（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节点），那么不允许通过机器人通过握手来通信。

当每个进程 p_i执行时，它会采取一系列动作，每个动作或是一个消息Send/Receive操作，或是一个转换状态p_i的操作，即改变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“交易服务器将事务情况记录到日志中”。

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。进程p_i中的事件序列可以用全序方式排列，我们用事件之间的关系$\rightarrow_i$表示。也就是说，当且仅当在p_i中事件e在e'前发生时，表示为$e \rightarrow_i e'$。不论进程是不是多线程的，这个排序都是定义良好的，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一系列事件，而且按关系$\rightarrow_i$排序：

$$history\ (p_i) = h_i = <e_i^0,\ e_i^1,\ e_i^2,\ \cdots>$$

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，
435 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，计算有固定频率晶体的振荡次数，把计数值分割一下，保存在计数器寄存器中。可以对时钟设备编程以便按照一定间隔产生中断，从而实现时间片之类的功能。不过，我们可以不关心这个方面的时钟操作。操作系统读取节点的硬件时钟值$H_i(t)$，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，$C_i(t)$则是软件时钟的读数，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为单位的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性的概念），那么我们能用它的值给p_i的事件打时间戳。注意，连续的事件将相对应于不同的时间戳，条件是*时钟分辨率*（时钟值更新的周期）比连续事件之间的时间间隔小。事件发生的速率取决于处理器指令周期长度这样的因素。

时钟偏移和时钟漂移 计算机时钟与其他时钟一样，并不是完全一致的（如图11-1所示）。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*时钟漂移问题*，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，因此振荡器的频率会有不同。而且，时钟频率有时会随温度不同而有所差别。有些设计试图弥补这种不同，但这些设计不能完全消除这种问题。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，不论这两个时钟的初始值是多么的一致。时钟的*漂移率*是指在由参考时钟度量的每个单位时间内，在时钟和名义上完美的参考时钟之间的偏移量。对普通的基于石英晶体的时钟，漂移率大约在10^{-6}s/s，即每1 000 000s或11.6天有1s的偏差。“高精度”的石英钟的漂移率大约为10^{-7}或10^{-8}。

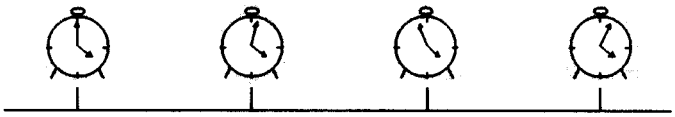


图11-1 分布式系统中计算机时钟之间的偏移

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，它的漂移率大约为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（Cs^{133}）在两个层次之间的跳跃周期的9 192 631 770倍。 436

秒、年和其他我们使用的时间单位来源于天文时间。它们最初按地球的自转和公转定义。然而，地球自转周期在慢慢变长，这主要因为潮汐的摩擦力；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并不一致。

通用协调时间（Coordinated Universal Time，UTC（该缩写是根据法语得来的））是国际计时标准。它基于原子时间，但偶尔需要增加闰秒或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）。

接收器可从商家购买。与“极为准确的”UTC相比，从陆地广播站接收的信号具有0.1～10ms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，其精度大约为几毫秒。

11.3 同步物理时钟

为了知道在分布式系统$\mathcal{P}$的进程中事件发生的具体时间（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。这标为*外部同步*。如果时钟C_i与其他时钟同步到一个已知的精度，那么我们能通过本地时钟度量在不同计算机上发生的两个事件的间隔——即使它们没有必要与外部时间源同步。这称为*内部同步*。我们在实际时间I的一个间隔上定义两个同步模式：

外部同步：设一个同步范围$D>0$，UTC时间源为S，I中的所有实际时间为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：设同步范围$D>0$，I中的所有实际时间为t，则有$|C_i(t) - C_j(t)|<D$，其中$i, j = 1, 2, \cdots, j, N$。该定义的另一种说法是时钟$C_i$在范围$D$中是一致的。

内部同步的时钟没必要进行外部同步，因为即使它们相互一致，它们与时间的外部源也有漂移。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。 437

*时钟正确性*概念有不同的提法。通常，如果一个硬件时钟H的漂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），那么该时钟就是正确的。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（在正常操作中）的跳跃。有时，我们也要求软件时钟遵循该条件。但用一个较弱的*单调性*条件就足够了。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，以决定是否进行编译。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，那么会出现源文件在编译前被修改的结果，此时make就会错误地不编译该源文件。

尽管发现时钟运行快了，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达到这一目标，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。

不满足正确性条件的时钟就被定义成是*有故障的*。当时钟完全停止滴答，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，因为将1999年12月31日后的日期登记成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，时钟不必非常正确。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。

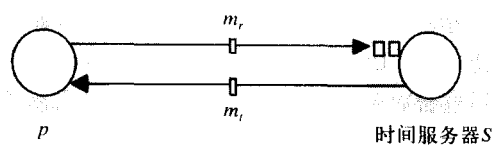


图11-2 用时间服务器进行时钟同步

11.3.1 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发送给另一个进程。原则上，接收进程可以将它的时钟设成$t + T_{trans}$，其中T_{trans}是在两个进程间传输m所花的时间。两个时钟应该能一致（因为是内部同步，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节点上竞争资源，其他消息与m竞争网络。如果没有其他进程要执行，也没有其他网络通信，那么总有一个最小的传输时间min，min可以被度量或适当地估计出来。

根据定义，在一个同步系统中，用于传输消息的时间有一个上界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。类似地，如果将时钟设成$t + max$，那么时钟偏移可能为u。然而，如果将时钟设成$t + (max + min)/2$，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 Lynch1984]。

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。对于一个异步系统，我们只能说$T_{trans} = min + x$，其中$x \geqslant 0$。x的值在某些情况下是不知道的，虽然对特定的环境，值的分布是可以度量的。

11.3.2 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11-2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，只有几分之一秒。他把算法描述成带条件的：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。

438 ~ 439

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}。如果时钟漂移率小，那么该值可以比较精确地度量这段时间。例如，往返时间在LAN上应该达到1～10ms数量级，漂移率为10^{-6}秒/秒的时钟在这段时间里变化至多10^{-5}ms。

假设S在m_t中放置t，往返时间在t时间点之前和之后平分，那么估计进程p应该设置它的时钟的时间为$t + T_{round}/2$。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，那么我们能用如下方法判断结果的精确性。

S能在m_t中放置时间的最早点是在p发出m_r之后的min。它能做此工作的最近时间点是在m_t到达p之前的min。因此，应答消息到达时S的时钟的时间位于范围$[t + min, t + T_{round} - min]$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发送几个请求（应该每隔一段时间发送一个请求以便造成拥堵）并用T_{round}的最小值给出最精确的估计，这样可在一定程度上应对可变性。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源于两个消息在接近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 如上所述，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，以至于暂时不能同步。因此，Cristian建议应该由一组同步时间服务器提供时间，每一个服务器都有一个UTC时间信号接收器。例如，一个客户可以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。

用假的时间值进行应答的故障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都会给计算机系统带来灾难。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，其中每一个协议都能容忍某类故障。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，那么要让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可由下面描述的Berkeley算法解决。恶意干扰时间同步的问题使用认证技术来应对。

440

11.3.3 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作为主机。与Cristian协议不同，这个计算机定期轮询其他要同步时钟的计算机（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）来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。概率的均衡是指这个平均值能抵偿单个时钟跑快或跑慢的趋势。协议的准确性依赖于主机和从属机之间名义上最大的往返时间。主机排除了某些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，也可以是一个负数。

算法避免了读取错误时钟的问题。如果用一个一般的平均值的话，这种有错的时钟会产生极大的负面影响。主机采用容错平均值。也就是说，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15台计算机的实验，使用他们的协议，这些计算机的时钟可同步在20～25ms之内。本地的时钟漂移率小于2×10^{-5}，最大的往返时间为10ms。

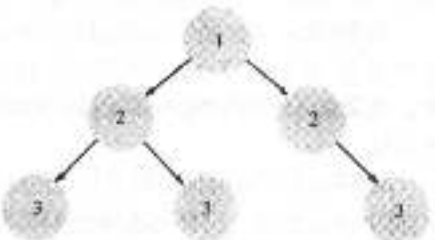
如果主机出现故障，要能选举另一个主机接管，并像它的前任一样工作。12.3节将讨论一些通用的选举算法。注意，它们并不保证在有限时间内选出一个新的主机，所以如果使用它们，在两个时钟之间的不同应不受约束。

11.3.4 网络时间协议

Cristian的方法和Berkeley算法主要应用于企业内部网。网络时间协议（Network Time Protocol，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。

NTP主要的设计目标和特色如下：

- 提供一个服务，使得跨因特网的用户能精确地与UTC同步；尽管在因特网通信中会遇到大的可变的消息延迟，但NTP采用了过滤时序数据的统计技术，以辨别不同服务器的时序数据。
- 提供一个能在漫长的连接丢失中生存的可靠服务；提供冗余的服务器并在服务器之间提供冗余的路径。如果其中一个服务器不可达，能重配置服务器以便继续提供服务。



注：箭头表示同步控制，数字表示层次。

图11-3 在NTP实现中同步子网的例子

- 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漂移率；服务能被扩展到处理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来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。

NTP服务由因特网上的服务器网提供。主服务器直接连接到像无线电时钟这样的接收UTC的时间源；二级服务器最终与主服务器同步。服务器在一个称为同步子网的逻辑层次中连接（见图11-3），其中的分层叫层次。主服务器占据层次1；它们是根。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，依此类推。最低层（叶子）服务器在用户的工作站上执行。

层次数大的服务器上的时钟比层次数小的服务器上的时钟更容易不准确，因为在同步的每一层都会引入误差。NTP在评估由某个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。

441
～
442

在服务器不可达或出现故障时，同步子网可以重配置。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服务器用以下三种模式中的一种相互同步：组播、过程调用和对称模式。组播模式用于高速LAN。一个或多个服务器定期将时间组播到由LAN连接的其他计算机上的服务器中，并设置它们的时钟（假设延迟很小）。这个模式能达到的准确性较低，但对许多目的而言，这已经足够了。

过程调用模式类似上述的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。这个模式适合准确性要求比组播更高的场合，或不能用硬件支持组播的场合。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持准确的时序信息。这时就可以以过程调用模式与本地服务器打交道。

最后，对称模式可用于在LAN中提供时间信息的服务器和同步子网的较高层（层次数较小）。即要获得最高准确性的地方。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。维护时序数据是为了提高时间同步的精确性。

在所有的模式中，使用标准UDP因特网传输协议进行消传递，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11-4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}和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会存在不可忽视的延迟。而且，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。

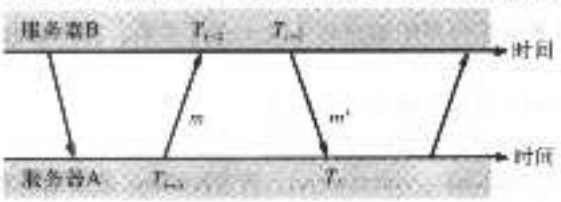


图11-4 一对NTP服务器之间的消息交换

443

对于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是对两个时钟之间实际偏移的一个估计，延迟d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和$t'\geq 0$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服务器对于连续的$<o_i, d_i>$对应用数据过滤算法，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），若过滤离中趋势较高，则表示数据相对而言不可靠。保留八个最近的$<o_i, d_i>$对。对于Cristian的算法，选择对应于最小值d_i的o_i的值用于估计o。

与某个源通信得到的偏移值未必用于控制本地时钟。通常，一个NTP服务器参与几个对等方的消息交换。除了应用到与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交换中获得的值，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。

层次较低的对等方比层次较大的对等方更受欢迎，因为它们“更接近”主时间源。具有最低同步离中趋势的对等方也比较受欢迎。这是服务器和同步子网的根之间度量的过滤离中趋势之和。（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采用一个阶段锁循环模型[Mills 1995]。它按照对漂移率的结果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固定比例走快，如每小时快4s，那么为了弥补这个问题，可稍微降低它的频率（用软件或硬件）。这样，时钟在两次同步间隔中的漂移会减少。

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。

444

11.4 逻辑时间和逻辑时钟

从单个进程的角度看，事件可唯一地按照本地时钟显示的时间进行排序。但Lamport[1978]指出，因为我们不能在一个分布式系统上完美地同步时钟，因此通常我们不能使用物理时间指出在分布式系统中发生的任何一对事件的顺序。

通常，我们使用类似物理因果关系的方案，但将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。这种排序是基于下面既简单又直观的两点：

- 如果两个事件发生在同一个进程p_i ($i=1,2,\cdots,N$) 中，那么它们发生的顺序是p_i观察到的顺序，即我们上面定义的顺序$\rightarrow_i$。
- 当消息在不同进程之间发送时，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*。

我们按如下所示定义发生在先关系（用$\rightarrow$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$e \rightarrow e''$。

由此，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$e=e_1$，$e'=e_n$，并且对于$i=1$，2，…，$N-1$，在e_i和e_{i+1}之间既可以应用HB1也可以应用HB2。也就是说，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的顺序不必是唯一的。

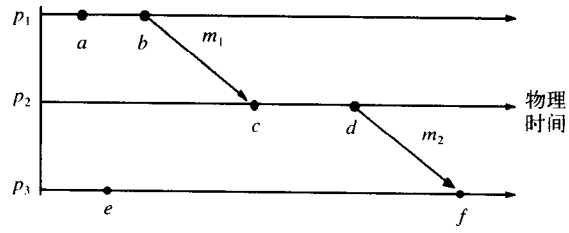


图11-5 发生在三个进程中的事件

图11-5中的3个进程p_1、p_2和p_3可用于说明关系$\rightarrow$。可以看到$a \rightarrow b$，因为在进程p_1中事件按这个顺序发生（$a \rightarrow_1 b$），类似地有$c \rightarrow d$。进一步有$b \rightarrow c$，因为这些事件是发送和接收消息m_1，类似地有$d \rightarrow f$。结合这些关系，我们可以得到$a \rightarrow f$。

445

从图11-5还可以看出，并不是所有的事件与关系$\rightarrow$相关。例如，$a \not\rightarrow e$和$e \not\rightarrow a$，因为它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$\rightarrow$排序的事件是*并发的*，写成$a \| e$。

关系$\rightarrow$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，那么第一条消息的发送显然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引起了第二个事件，也可能并未引起第二个事件。例如，如果服务器接收一个请求消息，后来发送了一个应答，那么很显然，应答的传送是由请求的传送引起的。但是，关系$\rightarrow$只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$\rightarrow$关系。例如，一个进程可能收到一个消息，后来又发送了另一个消息，但这个消息是每五分钟发送一次的，与第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$\rightarrow$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中附加值$t = L_i$。

(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)事件打时间戳时应用LC1。

尽管上面时钟的增量是1，但我们可以选用任何正数。通过在与事件e和e'有关的事件序列上进行长度归纳，可以很容易地看到：$e \rightarrow e' \Rightarrow L(e) < L(e')$。

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11-6给出了对图11-5中给出的例子使用逻辑时钟的结果。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$b \| e$。

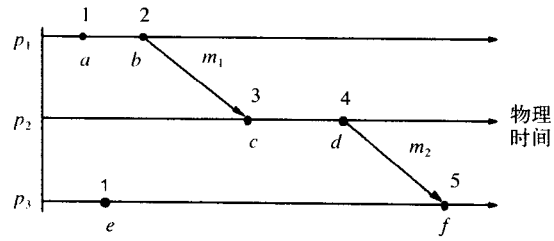


图11-6 图11-5中的事件的Lamport时间戳

全序逻辑时钟 一些由不同进程生成的不同的事件对会有用数字值表示的Lamport时间戳。然而，我们能通过考虑发生事件的进程的标识符来创建事件的全序，即对所有的事件对排序。如果e是在p_i中发生的事件，本地时间戳为T_i，而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 $(T_i, i) < (T_j, j)$。这种排序没有通常的物理意义（因为进程标识符是随机的），但它有时有用。例如，Lamport用它在一个临界区给进程排序。

向量时钟 Mattern[1989]和Fidge[1991]开发了向量时钟用以克服Lamport时钟的缺点：我们从$L(e) < L(e')$不能推出$e \rightarrow e'$。有N个进程的系统的向量时钟是N个整数的一个数组。每个进程维护它自己的向量时钟V_i，用于给本地事件加时间戳。与Lamport时间戳类似，进程在发送给对方的消息上附加向量时间戳，更新时钟的规则如下：

VC1：初始情况下，$V_i[j] = 0$，$i, j = 1, 2, \cdots, N$。

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$V_i[j] := \max(V_i[j], t[j])$，$j = 1, 2, \cdots, N$。这种取两个向量时间戳的最大值的操作称为合并操作。

对向量时钟V_i，$V_i[i]$是p_i已经附加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p_j可能给多个事件加时间戳，但至今没有信息流向p_i）。

446
↓
447

我们用下列方法比较向量时间戳：

$V = V'$ iff $V[j] = V'[j]$ $(j=1, 2, \cdots, N)$

$V \leqslant V'$ iff $V[j] \leqslant V'[j]$ $(j=1, 2, \cdots, N)$

$V < V'$ iff $V \leqslant V' \wedge V \neq V'$

设$V(e)$是发生e的进程所应用的向量时间戳。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$e \rightarrow e' \Rightarrow V(e) < V(e')$。练习10.13将要读者证明：如果$V(e) < V(e')$，那么$e \rightarrow e'$。

图11-7给出了图11-5中的事件的向量时间戳。从图上可以看到，$V(a) < V(f)$，这反映了$a \rightarrow f$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$c \| e$。

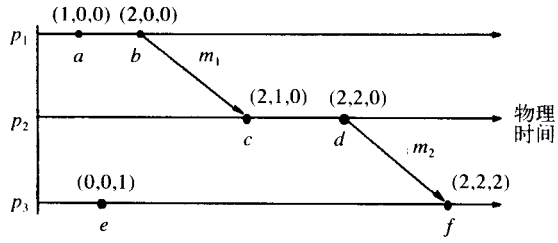


图11-7 图11-5中的事件的向量时间戳

与Lamport时间戳相比，向量时间戳的不足在于占用的存储以及消息的有效负载与进程数N成正比。Charron-Bost[1991]证明，如果我们能通过观察时间戳来区分两个事件是否并发，那么就不可避免地用到N维向量。但是，想以重构完整向量为代价来存储和传送更少量数据，这种技术是存在的。Raynal和Singhal[1996]对其中一些技术进行了介绍。他们还描述了*矩阵时钟*（matrix clock）的概念，进程凭借它保持自己和其他进程的向量时间。

11.5 全局状态

本节和下一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。

448

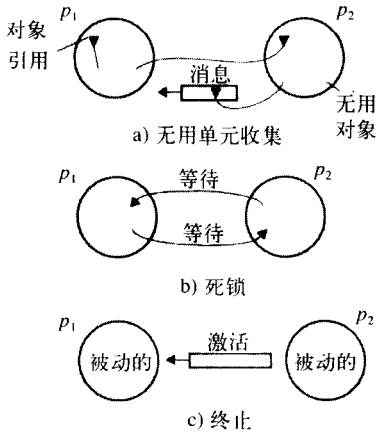


图11-8 检测全局性质

分布式无用单元收集：如果在分布式系统中不再对某个对象进行任何引用，那么该对象被认为是无用的。一旦认为对象是无用的，那么就要回收它所占据的内存。为了检查一个对象是否是无用的，我们必须验证系统中对它没有任何引用。在图11-8a中，进程$p1$有两个对象，它们都有引用——一个引用在进程$p1$内部，而进程$p2$引用了另一个对象。进程$p2$有一个无用对象，在系统中没有对它的引用。还有一个对象，$p1$和$p2$都没有引用它，但在进程之间的暂态消息中对它进行了引用。这说明，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。

分布式死锁检测：当一组进程中的每一个进程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。在图11-8b中，进程$p1$和$p2$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：看起来只要测试每个进程是否都已经停止而已。为了说明问题并不是这么简单，考虑由进程$p1$和$p2$执行的一个分布式算法，每个进程都会请求另一个进程的值。我们能确定在一个瞬间进程是主动的还是被动的——一个被动的进程没有参与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$p1$是被动的，$p2$是被动的（如图11-8c所示）。

449

为了说明我们不能推断算法已经终止，考虑下列情形：当我们测试$p1$的被动性时，一个消息在从$p2$向$p1$传送，$p2$在发出该消息后马上变成被动的。$p1$接收消息后，我们发现它又从被动变成主动。因此算法不能被终止。

终止和死锁的现象在某些方面比较类似，但它们是不同的问题。首先，死锁只影响系统中的

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：死锁进程试图执行进一步的动作，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过程中发生了什么。例如，在Smith写的应用中，每个进程p_i包含一个变量x_i (i =1, 2, ⋯, N)。变量随程序执行的进行而改变，但它们被要求相互之间的差值在一个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。

上述的每个问题都有适合的解决方案，但它们都说明了观察全局状态的必要，所以有必要开发一个通用的方案。

11.5.1 全局状态和一致割集

从原理上说，观察单个进程的连续状态是可能的，但查明系统的全局状态问题——进程集的状态——是非常困难的。

本质的问题是缺乏全局时间。如果所有进程都有完全同步的时钟，那么我们可以在同一时间让每个进程记录下它的状态——结果就是系统实际的全局状态。从进程状态集中我们可以判断进程是否发生死锁等。但我们不能获得完美的时钟同步，所以这个方法不适用。

我们可能会问：利用不同时间记录的本地状态能否得出一个有意义的全局状态？答案是在满足一定条件时可以，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$\mathcal{P}$中，我们将研究它的执行过程。在上面说过，在每个进程中发生了一系列事件，我们可以通过每个进程的历史来描述每个进程的执行过程：

$$history\ (p_i) = h_i = \langle e^0_i, e^1_i, e^2_i, \cdots \rangle$$

类似地，我们可以考虑进程历史的任何一个有限前缀：

$$h^k_i = \langle e^0_i, e^1_i, \cdots, e^k_i \rangle$$

450

每个事件或是进程的内部动作（例如，更新一个变量）或是在与进程相连的信道上发送或接收一个消息。

原则上，我们能记录在$\mathcal{P}$执行时发生的一切。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，而是让进程记录所有消息的发送或接收作为状态的一部分。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，我们就能推断出m是否是p_i和p_j之间信道状态的一部分。

通过取单个进程历史的并集，我们可以得到$\mathcal{P}$的全局历史：

$$H = h_0 \cup h_1 \cup \cdots \cup h_{N-1}$$

数学上，我们可以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，也就是说，哪些进程状态能同时发生？一个全局状态相当于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

$$C = h^{c_1}_1 \cup h^{c_2}_2 \cup \cdots \cup h^{c_N}_N$$

在对应于割集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$e^{c_i}_i$ (i = 1, 2, ⋯, N)之后的p_i的状态。事件集$\{e^{c_i}_i : i = 1, 2, \cdots, N\}$称为割集的边界。

考虑图11-9中给出的在进程p_1和p_2中发生的事件。该图给出了两个割集，一个割集的边界是$\langle e^0_1, e^0_2 \rangle$，另一个割集的边界是$\langle e^2_1, e^2_2 \rangle$。最左割集是不一致的。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。原则上，我们通过检查事件之间的→关系可获得这一点。

相反，最右割集是一致的。它包括消息的$m1$的发送和接收。它也包括$m2$的发送但不包括$m2$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。

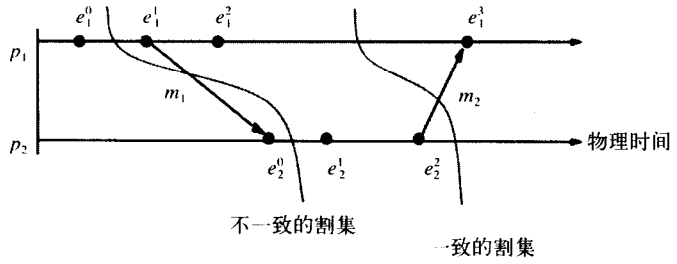


图11-9 割集

割集C是一致的，条件是对它包含的每个事件，它也包含了所有在该事件之前发生的所有事件，即

$$对于所有事件 e \in C, f \rightarrow e \Rightarrow f \in C$$

一致的全局状态是指对应于一致割集的状态。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转换：

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots$$

在每个转换中，正好一个事件在系统的一个进程中发生。这个事件或是发送消息，或是接收消息，也可以是一个外部事件。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——按照进程标识符排序（同时发生的事件必须是并发的，不是一个在另一个之前发生）。系统通过一致全局状态以这种方式逐步发展。

“走向”（run）是全局历史中所有事件的全序，并且它与每个本地历史排序$\rightarrow_i$ (i = 1, 2, …,N)是一致的。线性化走向或一致的走向是全局历史中所有事件的全序，并且与H上的发生在先关系是一致的。注意，线性化走向也是一个走向。

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。如果有一个经过S和S'的线性化走向，我们说状态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。

11.5.2 全局状态谓词、稳定性、安全性和活性

检测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、系统死锁、系统终止的状态相关的谓词的一个特征是这些谓词都是稳定的：一旦系统进入谓词值为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时，我们通常对不稳定谓词感兴趣，如在前面的例子中，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。

451
~
452

我们还注意到，与全局状态谓词有关的两个概念：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可以是成为死锁的性质。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的活性是对于任一从状态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。

11.5.3 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。该算法的目的是记录进程集p_i(i = 1, 2, …, 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用于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。收集状态的一个简单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讨论。

算法有如下假设：

- 不论是通道还是进程都不出现故障。通信是可靠的，因此每个发送的消息最终被完整地接收一次。
- 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（任意两个进程之间有一条路径）。
- 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，并发送和接收消息。

对每个进程p_i，设接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已经向进程p_j发送了消息m，但p_j还没有接收到，那么m属于它们之间通道的状态。

算法使用了特殊的标记消息，它与进程发送的其他消息不一样，它可在正常执行中发送和接收。标记有双重作用：如果接收者还没有保存自己的状态，那么标记作为提示；作为一种决定哪个消息包括在通道状态中的手段。

算法定义了两个规则：*标记接收规则*和*标记发送规则*（如图11-10所示）。标记接收规则强制进程在记录下自己的状态之后但在它们发送其他消息之前发送一个标记。

```
进程 p_i 的标记接收规则
  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 的状态记成空集；
        开始记录从其他接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11-10 Chandy和Lamport的“快照”算法

标记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，这是进程接收到的头一个标记。它记录在其他接入通道上后来收到了哪个消息。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，它就把那个通道的状态记录下来，作为从它保留它的状态以来所接收

到的消息集。

任何进程可以在任何时候开始这个算法。进程好像已接收到一个（在一个不存在的通道上的）标记，并遵循标记接收规则。这样，进程记录它的状态并开始记录在所有接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）。

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，它们通过两个单向通道c_1和c_2相连。两个进程进行“窗口部件”交易。进程p_1通过c_2向p_2发送窗口部件的订单，并以每个窗口部件10美元附上货款。一段时间以后，进程p_2沿通道c_1向p_1发送窗口部件。进程的初始状态如图11-11所示。进程p_2已经接收到5个窗口部件的订单，它将马上分发给p_1。

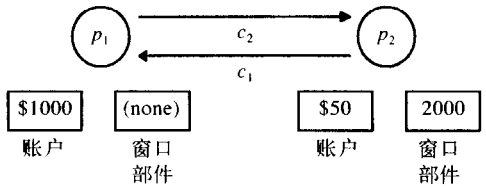


图11-11 两个进程和它们的初始状态

453
~
454

图11-12给出了系统的执行过程并记录系统的状态。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, $100）之前，在它的外出通道$c_2$上发送一个标记消息。系统进入实际的全局状态$S_1$。

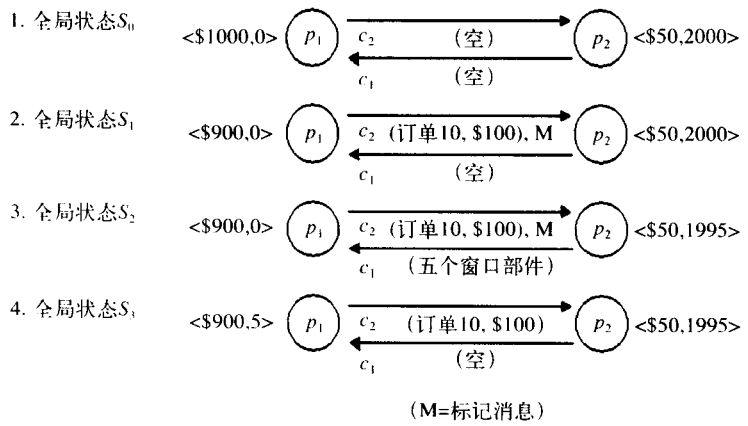


图11-12 图11-11中进程的执行

在p_2接收到标记之前，它通过c_1发出一个应用消息（5个窗口部件）以响应p_1以前的订单，产生新的实际全局状态S_2。

现在，进程p_1接收到p_2的消息（5个窗口部件），p_2接收到标记。根据标记接收规则，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c_1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（5个窗口部件）。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：<(5个窗口部件)>；c_2：<>。注意，这个状态与系统实际经过的所有全局状态不同。

快照算法的终止 我们假设一个已经接收到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（即使它不再需要在这些通道上发送应用消息）。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接的，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。

455

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。为了说明这一点，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。也就是说，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记录它的状态之前发生。如果两个进程是相同的，那么这一点非常明显，所以我们假设$j \neq i$。假设目前我们要证明的是：在e_j发生之前p_i记录了它的状态。考虑H个消息序列$m_1, m_2, \cdots, m_H (H \geqslant 1)$，有关系$e_i \rightarrow e_j$。通过在传递这些消息的通道上进行FIFO排序，以及标记发送和接收规则，一个标记消息将在每个$m_1, m_2, \cdots, m_H$之前到达p_j。根据标记接收规则，p_j将在事件e_j之前记录它的状态。这与我们e_j在割集中的假设相矛盾，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后的全局状态之间建立可达关系。设$Sys = e_0, e_1, \cdots$是系统执行时的线性化走向（若两个事件同时发生，我们将按照进程标识符给它们排序）。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_{rec}是所记录的全局状态。

我们将找到Sys的一个排列，$Sys' = e'_0, e'_1, e'_2, \cdots$，使得三个状态$S_{init}$，$S_{final}$，$S_{rec}$都在$Sys'$中发生，$S_{rec}$可从$Sys'$中的$S_{init}$处到达，$S_{final}$可从$Sys'$中的$S_{rec}$处到达。图11-13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'。

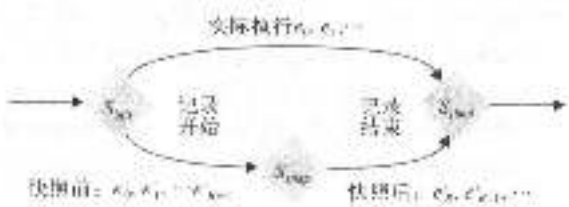


图11-13 在快照算法中状态之间的可达性

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，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，理解这一点是很重要的（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获得Sys'。假设e_j是一个进程的快照后事件，而e_{j-1}是另一个进程的快照前事件。不能得到$e_j \rightarrow e_{j-1}$。这两个事件可能分别是一个消息的发送和接收。标记消息必须在消息之前，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。因此我们可以在不违反发生在先关系的前提下交换两个事件（也就是说，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的顺序。 456

我们继续以这种方式交换相邻事件对，直到在Sys'执行结果中，所有快照前事件$e'_0, e'_1, e'_2, \cdots, e'_k$排列在所有快照后事件$e'_k, e'_{k+1}, e'_{k+2}, \cdots$之前。对每个进程，在$e'_0, e'_1, e'_2, \cdots, e'_k$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_{rec}。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非常有用。通常，在状态S_{rec}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_{rec}状态中稳定谓词为True，那么我们可以肯定在S_{final}状态中谓词是True。因为由定义可知，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，如果对于S_{init}状态谓词为False，那么在S_{rec}状态，该谓词也一定是False。

11.6 分布式调试

我们现在研究记录系统全局状态的问题，以使我们能对实际执行中的暂态状态（与稳定状态

相反）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阀门（由不同的进程控制）在某些时间都是开放的。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。这里我们面临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的快照算法按分布的方式收集状态，我们指出了系统中的进程如何把它们收集的状态发送给一个监控进程。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，监控器根据接受到的信息汇总成全局
457 一致状态。我们认为监控器在系统之外观察系统的执行。

我们的目的是在我们所观察的系统执行的某一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。出现“可能”这个概念是很自然的事，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ϕ（S）为True。仅仅观察一个一致的全局状态我们无法判断出一个非稳定谓词在实际的执行中是否曾为True。不过，我们有兴趣了解它们是否有可能发生，直到我们通过观察系统的执行来明确这一点。

概念“明确”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是有可能判断出ϕ是否明确为True的。

现在我们按照H的线性化走向为谓词ϕ定义*可能的*ϕ和*明确的*ϕ概念。

可能的ϕ　可能的ϕ意味着存在一个一致的全局状态S，H的一个线性化走向经历了这个全局状态S，而且该S使得ϕ（S）为True。

明确的ϕ　明确的ϕ意味着对于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ϕ（S）为True。

当我们使用Chandy和Lamport的快照算法，并获得全局状态S_{snap}时，如果ϕ（S_{snap}）正好是True，那么我们就可以认为可能的ϕ成立。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ϕ（S）为False，这还不是可能的ϕ的情况。还要注意，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（$\neg\phi$）得出¬可能的ϕ。后者是指如下断言：在每个线性化走向中，对于部分状态$\neg\phi$成立，而对于另一部分状态ϕ成立。

我们现在描述：

- 如何收集进程状态。
- 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, ⋯, N）最初用状态消息向监控器进程发送它们的初始状态，这以后也会不时发送状态消息。监控器进程在单独的队列Q_i（i =1, 2, ⋯, 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，但对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信息。有两种优化方法可减少发送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。这样，所观察的进程只需要向监控器进程发送相关状态。第二，进程仅在谓词
458 变成True或不再为True时发送它们的状态。发送不影响谓词值的状态的变化是没有意义的。

例如，在进程p_i应该遵循$|x_i - x_j| \leqslant \delta$ $(i, j = 1, 2, \cdots, N)$限制的系统例子中，进程只需要在它们自己的变量x_i的值改变时通知监控器。当它们发送状态时，它们只需提供x_i的值而不需要发送其他变量。

11.6.1 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有$f \rightarrow e \Rightarrow f \in C$时。

例如，图11-14给出了两个进程p_1和p_2，它们分别有变量x_1和x_2。在（具有向量时间戳的）时间线上的事件是对两个变量的值作调整。初始的时候，$x_1 = x_2 = 0$。要求是$|x_1 - x_2| \leqslant 50$。进程对变量作调整，但“大的”调整将使包含新值的消息被发送到其他进程。当一个进程从另一个进程接收到一个调整消息，它会把它的变量设成消息中所含的值。

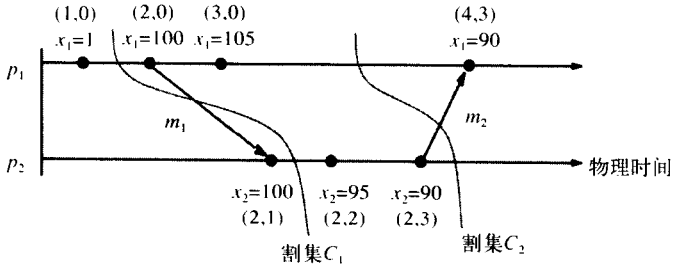


图11-14 执行图11-9产生的向量时间戳和变量值

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11-14中不一致割集C_1中的值，那么它将发现x_1=1，x_2=100，这违反了约束$|x_1 - x_2| \leqslant 50$。但这个状态是不会发生的。另一方面，来自一致割集C_2的值显示x_1=105，x_2=90。

为了让监控器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向量时钟值。每个队列Q_i都以发送顺序排序，这是通过检查向量时间戳的第i个部分实现的。监控器进程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。它必须检查状态消息的向量时间戳。

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p_i接收到的状态s_i的向量时间戳。那么S是一致的全局状态当且仅当：

$$V(s_i)[i] \geqslant V(s_j)[i] \quad (i, j = 1, 2, \cdots, N) \text{ —— (CGS条件)}$$

也就是说，当p_j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向量时间戳和在被观察进程发送给监控器的状态消息上附带信息，这样，监控器进程可以判断一个给定的全局状态是否一致。

图11-15给出了与图11-14的两个进程执行相对应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。节点表示全局状态，边表示状态之间可能的变换。全局状态S_{00}表示在初始状态中有两个进程；S_{10}表示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。

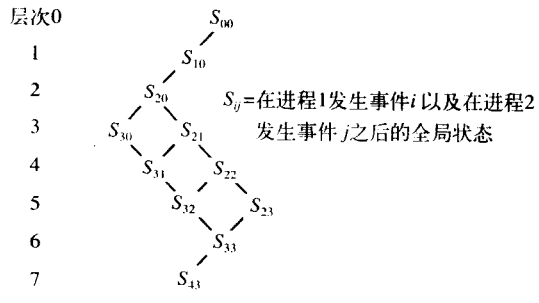


图11-15 执行图11-14产生的全局状态网格

459

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}位于层次（$i+j$）。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，也就是说，在每一步，都有一些进程经历了一个事件。例如，可从S_{20}到达S_{22}，但不能从S_{30}到达S_{22}。

网格给出了与一个历史相对应的所有线性化走向。现在从原理上能清楚地知道一个监控器进程应如何判定可能的ϕ和明确的ϕ。为了判定可能的ϕ，监控器进程从初始状态开始，经过从这点开始可到达的所有一致状态，在每一步判定ϕ。当ϕ判定为True时停止计算。为了判定明确的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判定为True的状态集。例如，如果图11-15中的
460 ϕ（S_{30}）和ϕ（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。

11.6.2 判定可能的ϕ

为了判定可能的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。算法如图11-16所示。算法假设执行是无限的。但可以很容易地将它改成有限的执行。

```
1. 对N个进程的全局历史H求解可能的φ
L := 0;
States := {(s1^0, s2^0, ⋯, sN^0)};
while(对所有的S ∈ States, φ(S) = False)
        L:=L + 1;
        Reachable := {S': H中从一些S ∈ States 可到达的状态 ∧ 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 (s1^0, s2^0, ⋯, sN^0)) 那么States := {} else States := {(s1^0, s2^0, ⋯, sN^0)};
while(States ≠ {})
        L:=L + 1;
        Reachable := {S': H中从 一些S ∈ States 可到达的状态 ∧ level(S') = L};
        States:= {S ∈ Reachable: φ (S) = False}
end while
输出 "明确的φ" ;
```

图11-16 求解可能的ϕ和明确的ϕ

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（$i = 1, 2, \cdots, N$）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：

$$V(s_j)[j] \geqslant V(s'_i)[j] \quad (j = 1, 2, \cdots, N, \text{且} j \neq i)$$

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
461 可从前一层的几个状态到达，所以监控器进程应该仅对每个状态判定一次一致性。

11.6.3 判定明确的ϕ

为了判定明确的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，每次一层地遍历可到达状态的网格。算法（如图11-16所示）又一次假设执行是无限的，但它可很容易地改成有限的执行。它维护States集合，该集合包含当前层的通过遍历ϕ为False的状态可从初始状态线性化可达的状态。

只要这样的线性化走向存在，我们就不可以断言明确的ϕ：执行可以采用这个线性化走向，ϕ在每个阶段可以是False。如果我们到达了一个不存在这样的线性化走向的层，我们就能断定明确的ϕ。

在图11-17中，第3层中的States集仅由一个状态组成，这个状态通过一个所有状态都是False（用粗线标记）的线性化走向可达。第4层只考虑一个标记为“F”的状态。（右边的状态没有被考虑，因为它仅能通过ϕ判定为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。

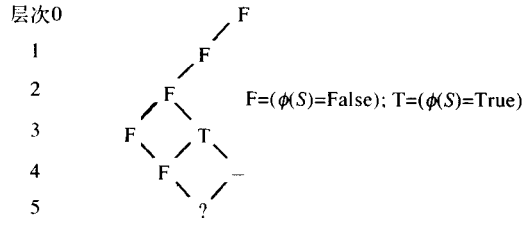


图11-17 判定明确的ϕ

开销 刚才描述的算法是组合爆炸的。假设k是一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：

$$V(s^{last}{}_j)[i] > V(s_i)[i] \quad j = 1, 2, \cdots, N, j \neq i$$

其中$s^{last}{}_j$是监控器进程从进程p_j接收到最后的状态，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。

462

11.6.4 在同步系统中判定可能的ϕ和明确的ϕ

到目前为止，我们所给出的算法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对于监控器所检查的一个一致的全局状态$S = (s_1, s_2, \cdots, s_N)$，在系统实际执行时，其中任意两个本地状态$s_i$和$s_j$可能间隔任意长的时间发生。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，假设进程均将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向量时间戳。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，这些状态的数量将比所有全局一致状态少。

我们现在按这种方式给出一个算法来利用同步时钟。假设每个要观察的进程p_i ($i = 1, 2, \cdots, N$)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。也就是说，在同一实际时间，有

$$|C_i(t) - C_j(t)| < D \ (i, j = 0, 1, \cdots, N)$$

所观察的进程将带有向量时间和物理时间的状态消息发送给监控器进程。监控器进程现在应用一个条件，该条件不仅用于测试全局状态$S = (s_1, s_2, \cdots, s_N)$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$i, j = 1, 2, \cdots, N$，有

$$V(s_i)[i] \geqslant V(s_j)[i]，且s_i和s_j能在同一实际时间发生$$

条件的第一个部分是我们以前使用的条件。对于第二个部分，我们注意到p_i是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发生下一个状态变换）的时候均处在状态s_i。考虑到时钟同步的边界，对在同一实际时间上获得的s_i和s_j，有：

$$C_i(s_i) - D \leqslant C_j(s_j) \leqslant L_i(s_i) + D，反之亦然（交换i和j）$$

监控器进程必须计算$L_i(s_i)$的值，这个值是用p_i的时钟来度量的。如果监控器进程已经接收到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输时间。

463

11.7 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，但对于判断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的偏序关系，它反映了事件之间的信息流——这些事件或在一个进程中，或是两个进程之间的消息。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。向量时钟是Lamport时钟的改进，因为通过检查它们的向量时间戳，可以判断两个事件是按发生在先关系排序还是并发的。

我们介绍了下列概念：事件、本地历史、全局历史、割集、本地状态、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来记录一致全局状态的问题。我们的目的是判定这个状态上的谓词。有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，并允许我们就一个稳定谓词是否在实际执行中成立给出断言。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走向中成立或可能成立。算法采用一个监控器进程收集状态。监控器检查向量时间戳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，它比较适合只有相对少的事件改变全局谓词值的实际系统。这个算法有一个适合于时钟可以同步的同步系统的变种。

练习

11.1 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。

（第434页）

11.2 当发现一个时钟快4秒时，它的读数是10:27:54.0（小时：分钟：秒）。解释为什么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8秒后变成正确的时间。

（第438页）

11.3 一种实现至多一次的可靠消息传递的方案是使用同步时钟来拒收重复的消息。进程在它们发送的消息中放上本地的时钟值（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，消息在传递后至多50ms能到达。

464

（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的时间戳为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中删除时间戳175 000ms？（提示：使用接收者本地的时钟值。）

（3）时钟应该进行内部同步还是外部同步？

（第439页）

11.4 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（小时：分钟：秒）
22	10:54:23.674
25	10:54:25.450
20	10:54:28.342

（第439页）

11.5 在练习11.4的系统中，要求将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。（第439页）

11.6 在NTP同步子网中，你希望发生怎样的重配置？（第442页）

11.7 一个NTP服务器B在16:34:23.480接收到来自服务器A的带有时间戳16:34:13.430的消息，并对消息给出了应答。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。（第443页）

11.8 讨论当决定一个客户应该与哪一个NTP服务器同步它的时间时，应考虑什么因素。（第444页）

11.9 通过观察时间的漂移率，讨论补偿同步点之间的时钟漂移的可能方法。讨论该方法的局限性。（第445页）

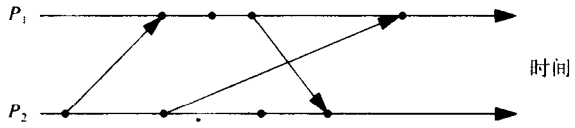
11.10 通过考虑连接事件e和e'的零或多个消息的链，并使用归纳方法证明$e \rightarrow e' \Rightarrow L(e) < L(e')$。（第446页）

11.11 证明 $V_j[i] \leqslant V_i[i]$（第447页）

11.12 按练习11.10的方式，证明$e \rightarrow e' \Rightarrow V(e) < V(e')$。（第448页） 465

11.13 利用练习11.11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$e \rightarrow e'$。（第448页）

11.14 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。在任何时刻，系统中只有一份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。（第453页）



11.15 上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。（第460页）

11.16 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。

1）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？

2）解释语句：possibly（$v_1 = v_2 = \cdots = v_N$）。Jones如何能判定该语句在她的执行中成立。（第461页） 466

第12章 协调和协定

本章介绍的主题和算法与如下问题有关：在发生故障时，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定。本章将首先介绍实现一组进程互斥的算法，该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，即在前一个协调者出现故障后，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，问题是对消息发送顺序这样的事情如何达成协定。共识和其他的问题是由如下问题归纳而来：一组进程如何对一些值达成协定，而不管这些值的值域是什么。我们会遇到分布式系统理论中的一个
467 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。

12.1 简介

本章将介绍一组算法，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定。例如，对于像太空船这样的复杂设备，一个基本要求是就控制它的各个计算机能对太空船的任务是继续还是已经终止这样的条件达成协定。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。计算机必须能做到这些，即使在各个部分之间没有固定的主-从关系（主-从关系会使协调变得简单）。避免固定的主-从关系的原因是，我们经常希望系统在出现故障时也能正确工作，因此就需要避免单节（例如固定的主控器）故障。

正如在第11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。本章将使用2.3.2节介绍的一个故障模型。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，然后考虑针对良性故障的算法，直到过渡到考虑怎样容许随机故障。我们会遇到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定——例如太空船的所有控制进程对“继续任务”或“放弃任务”达成协定。

12.2节将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中的扩展。由于在分布式系统中遇到的多是资源共享问题，因此这是一个重要的要解决的问题。随后，12.3节将介绍一个与之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11章中，我们看到进程如何把时钟与一个指定的时间服务器同步。如果这个服务器出现故障，而有多个正常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12.4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。12.4节将研究组播的可靠性和排序语义，并给出多种算法。组播传递本质上是进程间的协定问题，即接收者对接收哪些消息和按什么顺序接收消息达
468 成一致。12.5节将从更一般性的角度讨论协定问题，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给出的算法为何是正确的。由于篇幅所限，此处没有提供更严格的论述。读者可参考详细介绍分布式算法的教材，

如Attiya和Welch [1998]编写的教材以及Lynch [1996]编写的教材。

在给出问题和算法之前，我们先讨论分布式系统中的故障假设和检测故障的实际问题。

故障假设和故障检测器

为简单起见，本章假设每对进程都通过可靠的通道连接。也就是说，尽管底层网络组件可能出现故障，但进程使用能屏蔽故障的可靠通信协议，例如通过重传丢失或损坏的消息来屏蔽故障。为保持简洁性，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，我们假设在必要的地方有硬件冗余，以便在出现底层故障时，可靠通道不仅最终能传递每个消息，而且能在指定时间内完成传递工作。

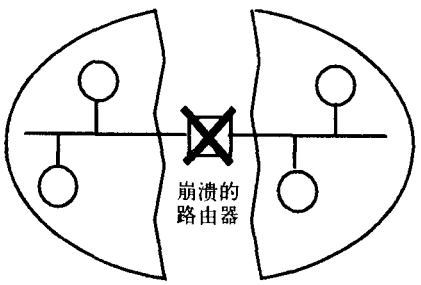


图12-1 网络分区

在某个时间间隔内，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可以通信，但两对进程间在路由器故障时是不可能进行通信的。这称为*网络分区*（参见图12-1）。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*，即从进程p到进程q可以通信，但反之不行。连接还可能是*非传递的*，也就是说，从进程p到进程q和从进程q到进程r都可以通信，但p不能直接与r通信。因此，我们的可靠性假设要包括任何有故障的链接或路由器最终会被修复或避开的内容。然而，所有进程不能够同时进行通信。 469

除非特别说明，本章假定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12.5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任何点都没有故障的进程。注意，正确性应用于整个运行，而非运行的一部分。因此，一个出现崩溃故障的进程在某一点之前是“无故障”的，但不是“正确”的。

设计克服进程崩溃的算法所遇到的问题之一是判断进程何时已经崩溃。*故障检测器*[Chandra and Toueg 1996, Stelling et al. 1998] 是一个服务，该服务用于处理有关某个进程是否已经出现故障的查询。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，但首先我们关注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。Unsuspected 表示检测器最近已收到表明进程没有故障的证据，例如，最近从该进程收到一个消息。但是那个进程可以自那以后出现故障。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，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作为提示）。这样的怀疑可能是错的：例如，进程可能正常运行，但在网络分区的另一边；或者进程可能运行得比预期慢得多。

*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。Failed表示检测器确定进程已崩溃。如前所述，已崩溃进程会保持原状，因为根据定义，进程一旦崩溃就不会再采取其他步骤。

要注意，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*向其他所有进程发送消息“*p* is here”，并且每隔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。但是，如果后来收到“*p* is here”消息，则向q报告p是OK。

在实际的分布式系统中，消息传送时间是有限制的。电子邮件系统也会在几天后放弃，即使
470 很可能通信链路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会多次怀疑非崩溃的进程，并且大部分带宽会被“*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。

对于此问题，一个实用解决方案是使用反映所观察网络延迟条件的超时值。如果本地故障检测器在20秒而不是预期的10秒内收到“*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，它对询问的回答仍只是提示，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。如果在*T+D*秒内没有收到消息“*p* is here”，那么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已经出现故障的进程（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（而实际系统很少是同步的）。

我们介绍故障检测器是因为它们有助于我们了解分布式系统中故障的本质，而任何用于应对故障的实际系统必须检测故障——不管多么不完美。但是即使是不可靠的故障检测器，只要它具有某些良构特性，也能为我们提供解决方案来处理发生故障时进程协调问题。我们在12.5节再讨论这个问题。

12.2 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，那么访问这些资源时，常需要互斥来防止干扰并保证一致性。这就是在操作系统领域中常见的*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核心提供的设施都不能用来解决这个问题。我们需要一个解决*分布式互斥*问题的解决方案：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13章将描述服务器如何同步客户对资源的访问。但在某些实际情况下，需要一个单独的用于互斥的机制。

考虑多个用户更新一个文本文件的情况。保证他们更新一致的一个简单方法是，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NSF文件服务器是无状态的，因此不支持文件加锁。为此，UNIX系统提供由守护进程locked实现的一个文件加锁服务，用于处
471 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无线网等网络中，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节点进行传输。再考虑一个监控一个停车场空位数的系统，在每个入口和出口有一个进程来跟踪进出车辆的数目。每个进程记录停车场内车辆总数，并且显示停车位是否已满。这些进程必须一致地更新车辆数的数目。有几个方法能实现这一点，比较方便的方法是这些进程只要通过相互通信就能互斥，这样可以不需要单独的服务器。

具有用于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, $i = 1, 2, \cdots, N$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。

假设系统是异步的。进程不出故障，并且消息传递是可靠的。这样传递的任何消息最终都被完整地恰好发送一次。

执行临界区的应用层协议如下：

```
enter()              //进入临界区——如果必要，可以阻塞进入
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问题。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。按进程请求的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。使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（参见11.4节）：

ME3: (一顺序)　如果一个进入CS的请求发生在先，那么进入CS时仍按此顺序

如果一种解决方案用发生在先顺序来决定进入临界区的先后，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予进入临界区。

我们按下列标准评价互斥算法的性能：

- 消耗的带宽，与在每个entry和exit操作中发送的消息数成比例。
- 每一次entry和exit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，一组进程作为一个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，没有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去访问资源。

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12.2给出了该服务器的使用。要进入一个临界区，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。从概念上说，该应答构成一个表示允许进入

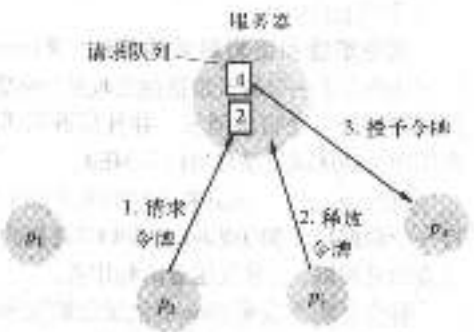


图12-2　为一组进程管理互斥令牌的服务器

422

473

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，那么服务器就不应答而是把请求放入队列。在离开临界区时，给服务器发送一个消息，交回这个令牌。

如果等待进程的队列不为空，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相应的进程。被选择的这个进程持有令牌。图中给出了p_2的请求被加入已经包含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来允许p_4进入临界区。进程p_1可能不需要进入临界区。

如果假设没有故障，很容易看到此算法满足安全性和活性条件。但是，读者会发现此算法不满足性质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，因为往返时间而使请求进程被延迟。离开临界区需要发送一个释放消息。假设采用异步消息传递，就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往返一次要花费的时间：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其他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。这样只要求每个进程p_i与环中下一个进程$p_{(i+1) \bmod N}$有一个通信通道。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实现互斥。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。需要令牌的进程将一直等待，直到接收到令牌为止，它会保留令牌。要离开临界区时，进程把令牌发送给它的邻居。

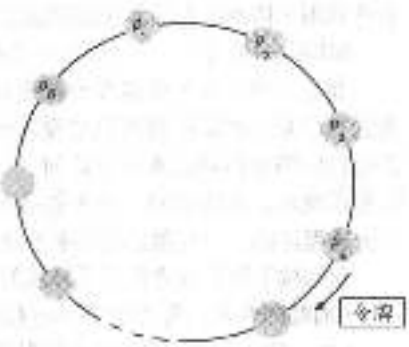


图12-3 传输互斥令牌的进程环

进程的布局如图12-3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，进程可以交换消息而不必理会令牌的轮转。）

该算法会不断消耗网络带宽（当一个进程在临界区中时除外）：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）到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。

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个对等进程间互斥的算法。该算法的基本思想是要进入临界区的进程组播一个请求消息，并且只有在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用于确保满足条件ME1～ME3。

进程p_1, p_2,…, p_N具有不同的数字标识符。假设进程互相之间都有通信通道。且每个进程p_i保持一个根据11.4节的规则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是发送者的标识符。

每个进程在变量state中记录它的状态，这些状态包括在临界区外（RELEASED）、希望进入（WANTED）以及在临界区内（HELD）。图12-4给出了协议。

如果一个进程请求进入，而其他进程的状态都是RELEASED，那么所有进程会立即回答请求，请求者将得以进入。如果某进程状态为HELD，那么该进程在结束对临界区的访问前不会回答请求，

因此在这期间请求者不能得以进入。如果有两个或多个进程同时请求进入临界区，那么时间戳最近的进程将是第一个收集到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符排序。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为止。这样做的目的是为了进程在处理请求时做出一致的决定。

```
初始化：
  state := RELEASED;
为了进入临界区：
  state := WANTED;
  组播请求给所有进程;        } 请求处理在此被延期
  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)找收一个请求<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_i队列，不给出应答;
  else
      马上给p_i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12-4 Ricart和Agrawala算法

该算法实现了安全性特性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回答了对方。但是，因为$<T_i, p_i>$对是全排序的，所以这是不可能的。请读者自行证明算法满足需求ME2和ME3。

为了说明上述算法，考虑图12-5所示的涉及三个进程p_1、p_2和p_3的情况。假设p_3不打算进入临界区，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的请求的时间戳是34。当p_3接到它们的请求时，将立即应答。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，将p_1搁置。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，因此允许p_1进入。

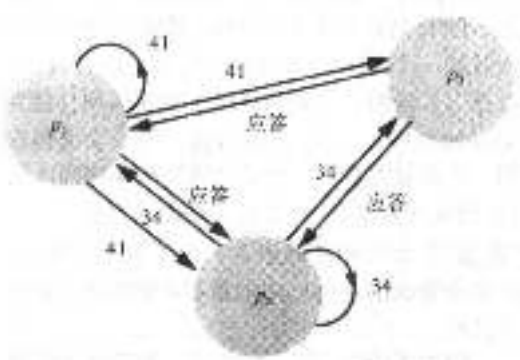


图12-5 组播同步

在该算法中，获得进入的许可需要$2(N-1)$个消息：$N-1$个消息用于组播请求，随后是$N-1$个应答。如果硬件支持组播，请求只需要一个消息，那么共需要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。前两个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，仍需如描述的那样执行协议，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个消息来获得进入许可。对此的描述见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。只要任意两进程使用的子集有重叠，进程只需要从其对等进程的子集获得进入许可即可，我们可以把进入临界区想象成进程互相选举。一个“候选”进程为进入必须收集到足够的选票。在两个投票者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：

- $p_i \in V_i$。
- $V_i \cap V_j \neq \phi$，即任意两个选举集至少有一个公共成员。
- $|V_i| = K$，即为公平起见，每个进程有同样大小的选举集。
- 每个进程 p_j 包括在选举集V_i中的M 个集合中。

Maekawa指明，最优解（即使K最小且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每个集合中的元素数相同）。计算最优集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12-6所示。为获得进入临界区的许可，进程p_i发送请求消息给V_i的所有K个成员（包括自己）。在收到所有K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”）。这时，它把请求消息加入队列（按到达时间顺序），但现在不回答。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”）响应该释放消息。为了离开临界区，p_i发送释放消息给V_i中的所有K个成员（包括自己）。

```
初始化：
  state := RELEASED;
  voted := FALSE;
p_i为了进入临界区：
  state := WANTED;
  将请求组播给Vi 中的所有进程；
  Wait until (接收到的应答数 = K );
  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，不予应答；
  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12-6 Maekawa算法

该算法实现了安全性ME1。如果两个进程p_i和p_j能同时进入临界区，那么$V_i \cap V_j \neq \phi$中的进程必须已经对它们两个投票。但该算法规定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。

遗憾的是，该算法易于死锁。考虑三个进程p_1、p_2、p_3，且$V_1 = \{p_1, p_2\}$，$V_2 = \{p_2, p_3\}$，$V_3 =$

$\{p_3, p_1\}$。如果三个进程并发地请求进入临界区，那么可能 p_1应答了自己但延缓 p_2，p_2应答了自己但延缓 p_3，p_3应答了自己但延缓 p_1。每个进程收到两个应答中的一个，故都不能继续。 473 ~ 478

可以修改算法 [Saunder 1987] 使其成为无死锁的。在修改后的协议中，进程按发生在先顺序对待应答的请求排队，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，每次退出需要$\sqrt{N}$个消息（假设没有硬件组播故障）。如果$N>4$，$3\sqrt{N}$的结果要优于Ricart和Agrawala算法的$2(N-1)$的结果。客户延迟与Ricart和Agrawala算法一样，但同步延迟更差一些，因为是一个往返时间，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。可以通过隐式地给所有请求授权来修改我们描述的Ricart和Agrawala算法，使得它容忍进程的崩溃故障。

请读者考虑，假设存在可靠的故障检测器，如何修改算法使之能够容错。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）并在检测到故障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，那么无论它持有令牌还是客户进程中的一个持有令牌，都必须恢复它们。

在12.5节我们将研究在有故障时进程如何协调它们的动作。

12.3 选举

选择一个唯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“中央服务器”互斥算法的一个变种中，“服务器”是从需要使用临界区的进程p_i, $(i = 1, 2, \cdots, N)$中选择的。这就需要一个选举算法来选择一个进程来扮演服务器的角色。基本要求是所有进程都同意这个选择。 479 然后，如果担任服务器角色的进程不想再担任此角色，那么需要再进行一次选举来选择替代者。

如果一个进程采取行动启动了选举算法的一次运行，则称该进程*召集选举*。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次选举。在任何时间点，进程p_i可以是一个*参加者*——意指它参加选举算法的某次运行，也可以是*非参加者*——意指它当前没有参加任何选举。

一个重要的要求是对当选进程的选择必须唯一，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符的进程为当选进程。“标识符”可以是任何有用的值，只要标识符唯一且可按全序排序即可。例如，通过用<1/*load*, *i*>作为进程的标识符（其中*load* >0且进程索引*i* 用于对负载相同的标识符排序），我们可以选举出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符。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，满足：

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符的非崩溃进程

E2：（活性） 所有进程p_i都参加并且最终置$elected_i \neq \perp$，或者进程p_i崩溃

注意，可能有还不是参加者的进程p_j，它在$elected_i$中记录着上次当选进程的标识符。

我们通过使用的总的网络带宽（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。该算法适合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 $p_{(i+1) \bmod N}$ 的通信通道，所有消息顺时针沿着环发送。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，它是具有最大标识符的进程。

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，然后把自己的标识符放到一个选举消息里，并把消息顺时针发送给它的邻居。

当一个进程收到一个选举消息时，它比较消息里的标识符和它自己的标识符。如果到达的标识符较大，它把消息转发给它的邻居。如果到达的标识符较小，且接收进程不是一个参加者，它就把消息里的标识符替换为自己的，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。

然而，如果收到的标识符是接收者自己的，这个进程的标识符一定最大，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*$_i$ 为消息里的标识符，并且把消息转发到它的邻居，除非它是新的协调者。

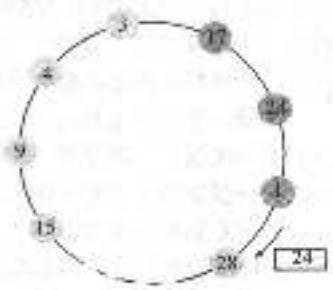
480

容易证明该算法满足条件E1。因为一个进程在发送当选消息前必须收到自己的标识符，所以所有标识符都被比较了。对任意两个进程，标识符较大的进程不会传递另一进程的标识符。因此不可能两者都收到它们自己的标识符。

根据算法保证环的遍历（没有故障）立即可证明条件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，最坏的情况是它的逆时针方向的邻居具有最大的标识符。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12-7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符。



注：选举从进程17开始，到目前为止，所发现的最大的选举标识符是24。参与的进程用深色显示。

图12-7 进行中的一次基于环的选举

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限制了它的实用价值。然而，通过利用可靠的故障检测器，在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。另一个区别是，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符。而霸道算法假定每个进程知道哪些进程有更大的标识符，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。

并开始一次选举。几个进程可能同时观察到此现象。

因为系统是同步的，所以我们可以构造一个可靠的故障检测器，最大消息传输延迟为T_{max}，最大消息处理延迟为$T_{process}$。因此，我们可以计算时间$T = 2T_{max} + T_{process}$，它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时间内没有收到应答，本地故障检测器可以报告请求的预期接收者已经出现故障。

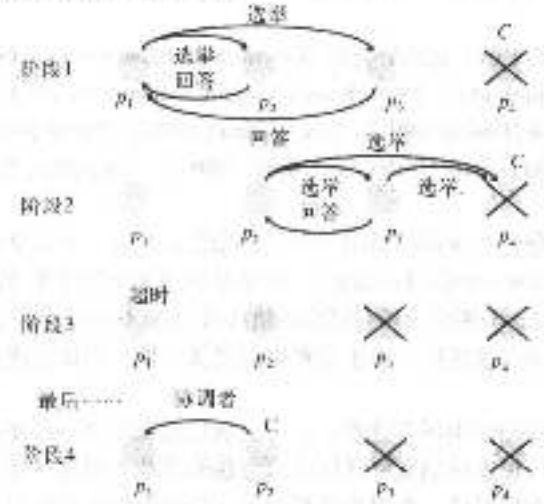
知道自己有最大标识符的进程可以通过发送协调者消息给所有具有较小标识符的进程，来选举自己为协调者。另一方面，有较小标识符的进程通过发送选举消息给那些有较大标识符的进程来开始一次选举，并等待回答消息。如果在时间T内没有消息到达，该进程便认为自己是协调者，并发送协调者消息给所有有较小标识符的进程来宣布这一结果。否则，该进程再等待时间T'以接收从新的协调者发来的消息。如果没有消息到达，它开始另一次选举。

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符，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。

当启动一个进程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符，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。

算法的运行过程如图12-8所示。有4个进程p_1~p_4。进程p_1检测到协调者p_4出现故障，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发送回答消息给p_1，并开始它们自己的选举：p_3发送一个回答消息给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次选举。最终，p_2被选为协调者（阶段4）。



最初是p_4出现故障，然后是p_3出现故障。在这种情况下，选举p_2为协调者。

图12-8 霸道算法

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，因为有较小标识符的进程会发现另一进程的存在并服从于它。

但是，如果崩溃的进程被替换为具有相同标识符的进程，那么该算法不能保证满足安全性条件E1。正在另一个进程（它已经检测到进程 p 崩溃）已经决定它有最大的标识符时，替换 p 的进程可能决定它有最大的标识符。两个进程可能同时宣布它们自己为协调者。遗憾的是，由于消息的传输顺序没有保证，这些消息的接收者对于谁是协调者可能得出不同的结论。

481 ~ 483

此外，如果假定的超时值被证明是不准确的，即如果进程的故障检测器是不可靠的，那么条件E1也可能会不成立。

考虑刚给出的例子，假设 p_3 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）也做着同样的事情。p_2 在发送自己的协调者消息后收到 p_3 的消息，因此置 $elected_2 = p_3$。由于消息传输延迟不同，p_1 在收到 p_3 的协调者消息后收到 p_2 的协调者消息，因此最终 $elected_1 = p_2$。于是，违反了条件E1。

关于算法的性能，最好情况是具有次大标识符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，即具有最小标识符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符的进程。

12.4 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进程都收到发到组中的、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的协定。

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应该重点关注的，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15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工作。

组播通信的基本特征是一个进程只调用一个组播操作来发送一个消息给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（broadcast）。

484

用一个组播操作代替多个发送操作不止为程序员提供方便。它使实现更为有效，并且提供的发送保证强于其他方式。

效率：同一消息要发送到组中所有进程，这一信息让实现可以有效使用带宽。通过发送消息到一个分布树，可以想办法使发送消息到任何通信链接的次数不超过一次，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、串行的方式传输消息，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，比较下列情况下带宽使用和总传输时间：从伦敦的一台计算机发送同一消息到位于Palo Alto的同一以太网上的两台计算机，（1）通过两个独立的UDP发送，（2）通过一个IP组播操作。在前一种情况下，消息的两个副本被独立发送，且第二个副本还被第一个延迟。在后一种情况下，消息不是发送两次，而是一组具有组播功能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：如果一个进程发送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消息。而且发送到组中任何两个成员的两个消息的相对顺序是未定义的。在IP组播的情况下，顺序或可靠性保证都没有提供。但是可以做出更强的组播保证，我们随后会定义。

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，进程可以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，为了简化顺序性质的讨论，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*给进程组*g*的所有成员。相应地，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，以阐明组播消息被进程节点收到后，并不总是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进行解释。

每个消息*m*携带发送它的进程 sender(*m*) 的唯一标识符和唯一目的组标识符 group(*m*)。我们假定进程不会谎报消息的源和目的地。 485

如果只有组的成员可以组播到该组，则该组称为*封闭的*（参见图12-9）。封闭组中的一个进程会将任何它组播到组的消息传递给它自己。如果组外的进程也可以向该组发送消息，则称该组是*开放的*。（"开放的"和"封闭的"分类也可应用于部件列表，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，这时可以用到进程的封闭组，传递事件到感兴趣的进程组，这时可以用到开放组。

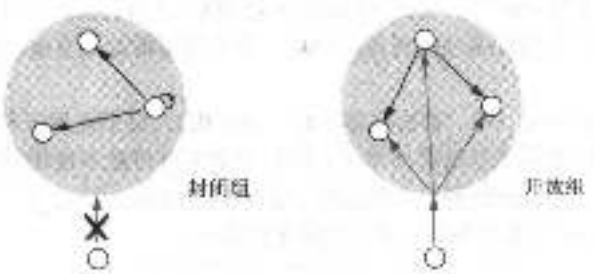


图12-9 开放组和封闭组

一些算法假定组是封闭的，通过挑选组中一员并发送消息给它（一对一），由它组播到组中，可以在封闭组中达到开放组的相同效果。Rodrigues 等[1998] 讨论了开放组的组播问题。

12.4.1 基本组播

拥有一个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，而每个消息发往某些特定组。

实现B-multicast的一个简单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，实现可以利用线程来并发执行send操作，遗憾的是，如果进程数很大，这样的实现会不可靠，很可能出现一种叫做确认爆炸的现象。作为可靠send操作的一部分发送的确认很可能从许多进程几乎同时到达。进行组播的进程的缓冲区会很快被填满，因此很可 486

能丢掉确认消息。于是进程会重新发送消息，导致更多的确认并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组播来构建，请读者自行完成这一服务。

12.4.2 可靠组播

2.3.2节定义了一对进程之间可靠的一对一通信通道。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（如果它是正确的话）。

按照 Hadzilacos和Toueg [1994]、Chandra和Toueg [1996]的研究成果，我们现在定义*可靠组播*以及相应的操作R-multicast和R-deliver。在可靠组播发送中，显然非常需要类似完整性和有效性的性质。但我们还要增加另一个性质：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发送操作的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一点出现故障，因此一些进程可能传递消息而另一些进程则不传递消息。

一个可靠组播是满足以下性质的组播，我们先给出这些性质，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$group(m)$中的其他正确的进程终将传递m。

完整性与可靠的一对一通信中的完整性类似。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（它只提到某个进程）。但是注意，有效性和协定一起得到一个完整的活性要求：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。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），原子性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，则这个消息有可能不被传递到组中的任何进程。但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都会传递它。文献中的许多文章用术
487 语“原子的”来包括一个全排序条件，我们稍后给出定义。

```
初始化：
  Received:={};
进程p为了将R-multicast消息m发给组g：
  B-multicast(g, m);                    // p∈g 被作为目的地被包括在内
在进程q On B-deliver(m)时，其中g=group(m)
  if (m ∉ Received)
  then
       Received := Received ∪ {m}
       if(q ≠ p)then B-multicast(g, m);end if
       R-deliver m;
  end if
```

图12-10 可靠组播算法

用B-multicast实现可靠组播 图12-10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，一个进程将消息B-multicast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。因为消息到达的次数可能多于一次，所以还要检测消息的副本且不传递它们。

这个算法显然满足有效性，因为一个正确的进程终将B-deliver消息到它自己。根据B-multicast中的通信通道的完整性，算法也满足完整性。

每一个正确的进程在B-deliver消息后都B-multicast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消息，这只能是因为它从来没有B-deliver此消息，而这又只能是因为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，因为我们没对时间进行假设。但是，该算法从实用角度来说是低效的，每个消息被发送到每个进程$|g|$次。

用IP组播实现可靠组播 R-multicast的另一种实现是将IP组播、捎带确认法（即确认附加在其他消息上）和否定确认结合使用。这个R-multicast协议基于下述观察，即IP组播通信通常是成功的，在该协议中，进程不发送单独的确认消息，而是在发送给组中的消息中捎带确认。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。

该描述假定组是封闭的。每个进程p为它属于的组g维持一个序号S_g^p。序号最初为零。每个进程还记录R_g^q，即来自进程q并且发送到组g的最近消息的序号。 488

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$<q, R_g^p>$。这个确认给出了一个序号，即自从发送进程q上一次组播消息后，p最近传递的来自进程q并且发往该组g的消息的序号。然后，组播进程p把消息连同它捎带的序号和确认一起IP组播到g，并且把S_g^p加一。

在组播消息中捎带的值使接收者了解到它们还没有接收到的消息。当且仅当$S = R_g^q + 1$，一个进程R-deliver一个来自p并发往g且序号为S的消息，在传递后立即把R_g^p加一。如果一个到达的消息有$S \leq R_g^p$，那么r已经传递了它，所以丢弃该消息。如果$S > R_g^q + 1$，或对任意封闭的确认$<q, R>$有$R > R_g^r$，说明r已经漏了一个或多个消息（在第一种情况下，很可能该消息已被丢弃）。它把满足$S > R_g^q + 1$的消息保留在一个保留队列中（参见图12-11）——这种队列常用于提供消息传递保证。它通过发送否定确认来请求丢失的消息。它或者发送请求到那个收到遗漏消息信息的进程q（这个进程收到一个确认$<q, R_g^q>$，R_g^q不小于所要求的序号）或发到最初的发送进程。

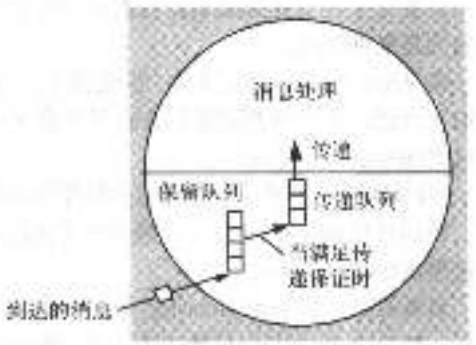


图12-11 用于到达的组播消息的保留队列

保留队列并不是可靠性必须的，但它简化了协议，使我们能使用序号来代表已传递的消息集。它也提供了传递顺序保证（见11.4.3节）。

通过检测副本和IP组播性质（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IP组播具有该性质时成立。当一致起见，我们首先要求进程总可以检测漏掉的消息。这又意味着进程会收到又一个消息，使它能够检测到遗漏。因此，只在假定每个进程都无限组播消息的情况下，这个协议具有有效性。其次，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，就能保证一致性成立，因此我们假定进程无限地保留它们已传递消息的副本。 489

我们为保证有效性和协定所作的假设都是不实用的（参见练习12.14）。但是，在我们所讲述的协议所派生出的协议中，协定已经被解决了：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。

统一性质 上面给出的协定定义只提到正确进程的行为，即进程从不崩溃。请考虑如果一个进程不是正确的，并且在R-deliver一个消息后崩溃，如图12-10的算法会发生什么。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性质。我们定义统一协定如下：

统一协定：如果一个进程传递消息m，不论该进程是正确的还是出故障，在$group(m)$中的所有正确的进程终将传递m。

统一协定允许一个进程在传递一个消息后崩溃，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12-10的算法满足这一性质，该性质比前面定义的非统一协定更强。

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现象，在这种应用中，统一协定是有用的。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协定，那么就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。

有趣的是，在图12-10中，如果颠倒“R-deliver m”和“if (q ≠ p) then B-multicast(g, m); end If”这两行的顺序，那么算法将不满足统一协定。

正如协定有一个统一的版本一样，任何组播性质也有统一的版本，包括有效性、完整性和我们将要定义的有序性。

12.4.3 有序组播

由于底层的一对一发送操作会发生随机延迟，因此12.4.1节的基本组播算法按任意顺序给进程传递消息。这种顺序保证的缺少对许多应用而言都是不能令人满意的。例如，在一个核电站里，表示对安全条件有威胁的事件和表示控制单元的动作的事件能被系统中的所有进程以同样的顺序观察到是很重要的。

常见的排序需求有全排序、因果排序、FIFO排序以及全-因果排序和全-FIFO排序的混合。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。后面我们还将讨论允许组之间有重叠的情况。

FIFO排序：如果一个正确的进程发出multicast(g, m)，然后发出multicast(g, m')，那么每个传递m'的正确的进程将在m'前传递m。

因果排序：如果 multicast(g, m)→multicast(g, m')，其中→是只由g的成员之间发送的消息引起的发生在先关系，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）。

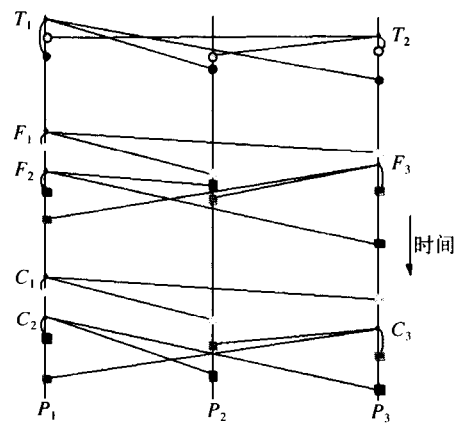


图12-12 组播消息的全排序、FIFO排序和因果排序

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12-12说明了3个进程的排序。仔细观察图可发现，全排序消息的传递顺序与它们被发送的

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，只要该顺序在不同进程中是一样的即可。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排序也遵守全排序。 490 ~ 491

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。

正如我们将看到的那样，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会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。因此，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的组播系统 [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。当一个用户将一个消息张贴到一个公告牌时，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。

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
结束		

图12-13 公告牌程序的显示

如果每个用户最终要收到每个张贴，就需要可靠的组播。用户也有排序的需求，图12-13给出了出现在某个用户面前的张贴。至少需要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“A.Hanlon”）的每一个张贴，用户才可以一致地讨论A.Hanlon的第二个张贴。 492

注意，主题为“Re:Microkernels”(25)和“Re:Mach”(27)的消息出现在它们引用的消息之后。为保证这种关系，需要因果排序的组播。否则，随机的消息延迟可能会造成消息“Re:Mach”出现在关于Mach最初的消息之前。

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个消息，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。在大范围内实现这些排序的通信代价超过了实现排序所带来的好处。

实现FIFO排序 FIFO排序的组播（具有FO-multicast和FO-deliver操作）可以用顺序号实现，就像我们在一对一通信中实现的那样。我们只考虑非重叠组的情况。读者可以验证，12.4.2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12.4.2节可靠组播协议中进程p的变量S^p_g和R^q_g，S^p_g是进程p已发送到g的消息个数，R^q_g是p已传递的来自进程q并且发往组g的最近的消息的序号。

p要FO-multicast一个消息到组g时，它在消息上捎带值S^p_g，接着B-multicast消息到g，然后把S^p_g加1。当收到来自q的序号为S的消息时，p检查是否 $S = R^q_g + 1$。如果满足该条件，说明这个消息是预期的来自发送进程q的下一个消息，p FO-deliver该消息，并且置 $R^q_g := S$。如果$S > R^q_g + 1$，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$S = R^q_g + 1$为止。

因为来自一个给定发送进程的所有消息以同样的次序传递，并且消息的传递被延迟直到到达该序号，显然FIFO排序的条件已满足。但是这仅在组不重叠的假设下成立。

注意，在这个协议中，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 实现全排序的基本途径是为组播消息指定全排序标识符，以便每个进程可以基于这些标识符做出相同的排序决定。传递算法与我们描述的用于FIFO排序的算法很相似，区别是进程保持组特定的序号，而不是进程特定的序号。我们只考虑如何对发送到非重叠组的消息进行全排序。我们把这类组播操作称为TO-multicast和TO-deliver。

我们讨论为消息指定标识符的两种主要方法。第一种方法是由一个叫做顺序者的进程来指定标识符（见图12-14）。一个要TO-multicast消息m到组g的进程把一个唯一的标识符$id(m)$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)。（顺序者可以是g的一个成员。）进程sequencer(g)维护一个组特定的序号s_g，用来给它B-deliver的消息指定连续的且不断增加的序号。它通过给g发送B-multicast顺序消息来宣布序号（详见图12-14）。

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-multicast(g ∪ {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12-14 使用顺序者的全排序

一个消息将一直保留在保留队列中，直到它依照相应的序列号可以被TO-deliver为止。因为序号是（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。证明的过程请读者自行完成。

基于顺序者的方案有一个明显的问题，即顺序者会成为瓶颈，并且是一个关键的故障点。有一些解决故障问题的实用算法。Chang and Maxemchuk [1984] 首先提出了一个使用一个顺序者（它们称为令牌场地）的组播协议。Kaashoek 等人 [1989] 为Amoeba系统开发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保留在$f+1$个节点的保留队列中，因此可以容忍多达f个故障。像Chang和Maxemchuk一样，Birman 等人[1991]也使用一个令牌保留场地作为顺序者。令牌可以在进程之间传递，这样，如果只有一个进程发送全排序组播，那么这个进程可以作为顺序者，从而减少通信。

Kaashoek等人的协议使用基于硬件的组播（如可在以太网上用的），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。顺序者把消息本身连同标识符和序号一起组播。这样做的优点是组中其他成员每次组播只接收一个消息，但缺点是带宽的使用增加。完整的协议描述见www.cdk4.net/coordination。

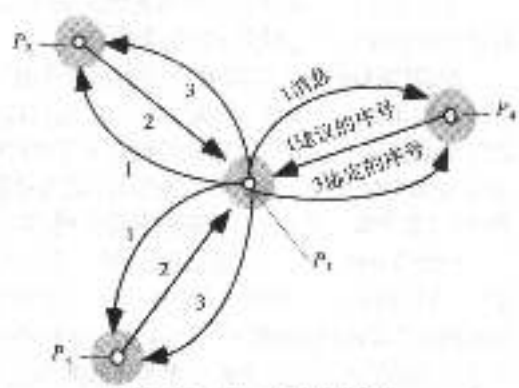


图12-15 全排序的ISIS算法

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序号达成一致的方法。一个简单的算法——与最初为ISIS工具包开发的实现全排序的组播传递的算法 [Birman and Joseph 1987a] 类似——如图12-15所示。它也是由一个进程把消息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序号，并把它们返回给发送者，后者用这些顺序数来产生协定的序号。

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序号）和P_g^q（即它自己提出的最大序号）。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的一个唯一的标识符。

2）每个进程q回答发送者p，提议$P_g^q := \text{Max}(A_g^q, P_g^q) + 1$为此消息的协定序号。实际上，在提议的$P_g^q$里必须包括进程标识符以保证全排序，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这样做。每个进程临时把提议的序号分配给消息，并把消息放入它的保留队列中，保留队列是按照最小的序号在队首的方式排序。

3）p收集所有提议的序号，并选择最大的数a作为下一个协定序号。然后，它B-multicast <i, a>到g。g中每个进程q置$A_g^q := \text{Max}(A_g^q, a)$，并把$a$附加到（标识符为$i$的）消息上。如果协定序号与提议的序号不一样，它把保留队列中的消息重新排序。当在保留队列队首的消息被赋予协定序号时，它被转移到传递队列的队尾。但是，已被赋予协定序号、但不在保留队列队首的消息不被转移。

493 ≀ 495

如果每个进程同意同一组序号，并按相应的顺序传递它们，那么满足全排序。显然，正确的进程最终会对同一组序号达成一致，但我们必须指出，序号是单调递增的，并且正确的进程不能过早地传递消息。

假定给消息m_1指派了一个协定序号，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大的提议序号，因此也有一个比m_1大的协定序号。这样，令m_2是尚未指定协定序号、但在同一队列中的其他消息。根据刚给出的算法，我们有：

$$agreedSequence(m_2) \geqslant proposedSequence(m_2)$$

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：在一个消息被传递前，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和Spauster [1991]和Hadzilacos、Toueg [1994]的文章。

实现因果排序 图12-16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。

每个进程p_i (i = 1, 2, …, N)维护自己的时间戳向量（见11.4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会一直等待，直到（1）它已传递了由p_j发送的任何较早的消息，（2）它已传递了p_j在组播该消息时

496

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12-16。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，虽然在图12-16中没有描述这一点。

```
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m：
  V_i^g[i]:= V_i^g[i] +1;
  B-multicast(g,< V_i^g,m>);
在B-deliver(<V_j^g,m>)来自p_j(j≠i)的一个消息时,其中g=group(m)：
  将<V_j^g,m>放入保留队列，直到V_i^g[j]= V_j^g[j]+ 1和 V_i^g[k]≤ V_j^g[k] (k≠j);
  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12-16 使用时间戳向量的因果排序

每个进程在传递消息时，要更新它的向量时间戳，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11.4节更新向量时钟的规则里出现的合并操作的一种优化。考虑到图12-16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证明如下。假设multicast(g, m)→multicast(g, m')。令V和V'分别是m和m'的向量时间戳。从算法可以直接地归纳证明$V < V'$。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传递一个来自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'，因为需要满足$V_i[k] \geqslant V'[k]$，这样的话，就会有$V_i[k] \geqslant V[k]$。

读者应该能证明，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
497 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，并按收消息的次序组播消息的序号。目的组中进程直到收到了来自顺序者的排序消息，并且消息是传递队列中的下一个消息时，才发送此消息。

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，但并不能令人满意，因为进程一般会成为多个重叠组的成员。例如，一个进程可能对来自多个来源的事件感兴趣，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)，然后发出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)→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。这是一个低效的并不令人满意的实现：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]、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自行完成。我们描述的所有算法在异步系统中都能正常工作。

然而，我们没有给出一个既保证可靠传递又保证全排序传递的算法。虽然看起来有点令人惊奇，但具有这些保证的协议在同步系统中是可能的同时，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。我们将在下节讨论这一问题。 498

12.5 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一致意见。

例如，第2章描述了一种两个部队要对进攻或撤退达成一致意见的情形。类似地，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”还是“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、贷动作达成一致。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。在全排序组播中，进程对消息传递顺序达成协定。

适合这几类协定的协议是存在的。我们描述了其中的一些协议，在第13章和第14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。

本节将更精确地定义共识以及与它相关的3个协定问题：拜占庭将军、交互一致性和全排序组播问题。接下来，我们研究在什么情况下这些问题可得到解决，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，即使进程组只含有一个有错进程也不能保证达成共识。最后，我们考虑在有不可能性结果情况下的实用算法。

12.5.1 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,…,N)。在许多实际情况下，一个重要的要求是，即使有故障也应能达成共识。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，即它们具有某种类型的错误，其余的进程是正确的。

如果出现随机故障，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会做出错误的断言。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。 499

共识问题的定义 为达到共识，每个进程p_i最初处于*未决*状态，并且提议集合D中的一个值v_i (i = 1, 2, …, 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, …, N)的值。在这种情况下，它进入*决定*状态。在此状态下，它不再改变d_i (i = 1, 2, …, N)。图12-17给出了参与一个共识算法的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都决定“继续”。

共识算法的要求是在每次执行中满足以下条件：

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同一个值，那么处于决定状态的任何正确进程已选择了该值。

根据应用的不同，完整性定义可以有变化。例如，一种较弱的完整性是决定值等于某些正确进程提议的值，而不必是所有进程提议的值。我们将使用上面的定义。

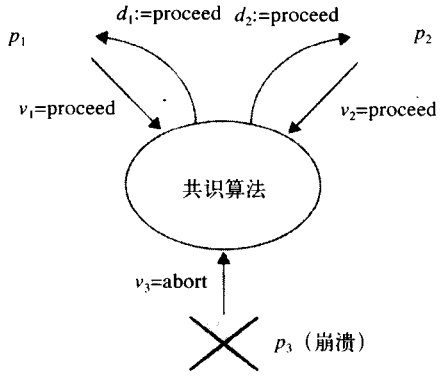


图12-17 3个进程的共识

为理解问题的表达是如何翻译为算法的，考虑进程不出现故障的一个系统。这时，解决共识是比较简单的。例如，我们可以把进程集中为一组，并让每个进程可靠地将它提议的值组播到组中的成员。每个进程等待，直到它收集到N个值（包括它自己的）为止。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 $\perp \notin D$。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相同函数。因此它们一定一致，并且如果每个进程提议相同的值，那么它们都决定这个值。

500

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很多函数，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，那么就会给检测故障带来复杂性，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。我们稍后再讨论这个问题。

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其他进程发送任何数据。虽然在实际中这看起来不太可能，但是一个有漏洞的进程确实可能出现这样的错误。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，以阻止这组进程达成一致。如果遇到这种不一致的情况，正确的进程必须用它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，其他的将军（作为司令手下的中尉）决定是进攻还是撤退。但是一个或者多个将军可能会叛变，也就是说会出错。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议一个值。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 $a_i = d_j$ $(i, j = 1, 2, \cdots, N)$。

完整性：如果司令是正确的，那么所有正确的进程都采取司令提议的值。

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值向量达成一致，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，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。

完整性：如果进程 p_i 是正确的，那么所有正确的进程都把 v_i 作为它们决定向量中的第 i 个分量。

501

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，共识、拜占庭将军、交互一致性问题在随机故障和崩溃故障的环境中都是有意义的。同样，它们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来解决这个问题。这是一个很有用的性质，不仅是因为加深了我们对问题的理解，也因为通过重用已有的解决方案，我们能降低实现的工作量以及复杂性。

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）问题：

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，它建议的值是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。

可以从其他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，我们可以从BG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：如果大部分进程是正确的，那么通过运行IC算法能够在每个进程中产生一个值向量，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$(IC_i(v_1, v_2,\cdots,v_N)[1],\cdots, IC_i(v_1, v_2,\cdots,v_N)[N])$

$(i = 1, 2\cdots N)$，其中*majority*如前定义。

从C构造BG 我们采用如下的方式从C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）。

502

- 最后得到 $BG_i(j, v) = C_i(v_1, v_2,\cdots,v_N)(i = 1, 2, \cdots, N)$。

读者可以证明在每一个例子都满足终止性、协定性和完整性。Fisher[1983]提供了关于这三个问题的更多细节。

在存在崩溃故障的系统中，解决共识问题等同于解决可靠且全排序组播，给定其中一个问题解决方案，就可以解决另一个问题。使用一个可靠且全排序组播操作RTO-multicast实现共识问题是比较简单的。我们将所有的进程组成一个组g。为了达成共识，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-delivers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]说明了如何从共识问题中得到可靠且全排序组播。

12.5.2 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。

为了达成共识，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，最多有f个进程可能崩溃。最坏的情况下，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。

如图12-18所示，该算法是基于Dolev和Strong[1983]的算法，其表示基于Attiya和Welch[1998]。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12-18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。

```
对p_i∈g的进程的算法；算法进行到f+1 轮
初始化：
  Values^1_i:={v_i}; Values^0_i={};
在第r轮(1≤r≤f+1)
  B-multicast(g, Values^r_i − Values^(r−1)_i); //仅发送还没有发送的值
  Values_i^(f+1) : = Values^r_i;
  While(在第r轮)
  {
      在B-deliver(V_j) 来自p_j的消息时：
        Values_i^(f+1) := Values^(r+1)_i ∪ V_j;
  }
在(f+1)轮之后
  将d_i赋成minimum(Values^(f+1)_i);
```

图12-18 同步系统中的共识

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。同时因为进程对这个集合应用了*minimum*函数，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集合不同。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。出现这种情况唯一的解释是另外还有一个进程，假设是p_k，它在把v传送给p_i后，还没有来得及传送给p_j，就崩溃了。同样道理，在前一个回合里p_k得到值v而p_j没有收到值的唯一解释是在该回合中发送v的进程崩溃了。以此类推，每个回合至少一个进程崩溃。但是我们假设最多只有f个进程崩溃，而我们进行了$f+1$个回合。这样我们就得出了矛盾。

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]。

12.5.3 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。也就是说，一个故障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个会发生故障。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再发送消息，因此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通信通道是私有的。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故障进程给不同进程发送的消息是不一致的。我们一般认为通道是可靠的，也就是说一个故障进程不能把消息插入到正确进程之间的通信通道中。

Lamport等人[1982]讨论了3个进程相互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。稍后我们将会简要说明这个结论。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。 503 - 504

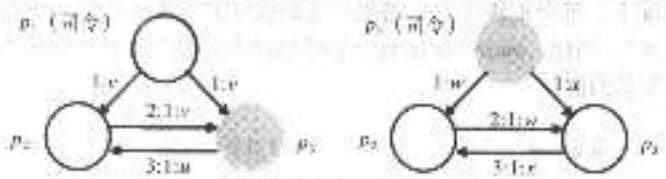


图12-19 3个拜占庭将军

3个进程的不可能性 图12-19给出了3个进程中只有一个进程出现故障的两个场景。在左边的场景中，中尉p_1有故障；对于右边的情况，司令p_1有故障。图12-19中给出了两个回合的消息交换：司令发送的值和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"。

在左边的场景下，司令正确地将同一个值v发送给其他两个进程，p_3正确地将这个消息发送给p_1。然而，p_3将$u \neq v$发送给p_2。在这个阶段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。

在右边的场景下，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_1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，这是完整性条件所要求的。如果没有算法能够区分这两种情况，那么p_2必须还是选择右边场景下司令发送的值。

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_2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（司令出现故障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有故障的，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]。如果将军们能够对他们发出的消息使用数字签名，那么3个将军中有一个出现故障，也能实现拜占庭协定。

对于$N \leqslant 3f$的不可能性 Pease等人推广了3个进程的不可能性结论，证明只要$N \leqslant 3f$，就不可能有解决方法。下面简要给出证明。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；进程在内部模拟内部将军之间的交互，并且自己的将军还会给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。 505

由于假设进程运行的算法是正确的，因此模拟能够终止。那些正确的将军（在两个正确的进程中）就会达成一致并且满足完整性。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。

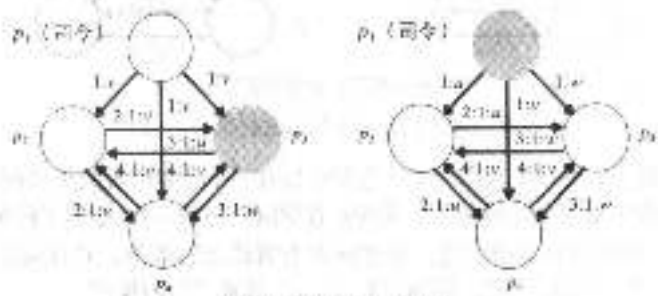
对一个有错进程的解决方案 Pease等人提出了一个算法来解决$N \geqslant 3f+1$的同步系统中的拜占庭将军问题。在这里没有足够的篇幅来讨论这个算法，但是我们将给出$N \geqslant 4$，$f=1$的算法操作，并以$N=4, f=1$来说明该算法。

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与自己同级的人。

每个中尉收到司令发来的一个值，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值，否则，一个中尉有错，他的其他同事收到司令发来的值的$N-2$份副本，以及有错的中尉发来的一个值。

不管在哪种情况下，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。由于$N \geqslant 4$，$(N-2) \geqslant 2$，因此，*majority*函数会忽略出错中尉发来的值，并且当司令是正确的时候，该函数能产生司令发来的值。



有故障的进程用灰色显示

图12-20 4个拜占庭将军

我们用有4个将军的情况说明上述算法。图12-20给出了与图12-19相似的两个场景，但是现在有4个进程，其中一个进程是有错的。像在图12-19中一样，左边图中的中尉p_2是有错的；在右边的图中，司令p_1是有错的。

当出现左图的场景时，两个正确的中尉进程在决定司令的值时达成一致：

p_3决定 $majority\ (v, u, v) = v$

p_4决定 $majority\ (v, v, w) = v$

在右图的场景中，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$majority\ (v, u, w) = \perp$（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偏发消息的情况，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，它就认为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：

- 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（$f \geqslant 1$），Lamport等人的算法用于未签名的消息传送时，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，如果允许出现拜占庭故障（因而也包括拜占庭将军问题），至

12.5.1节），那么任何确定性的解决共识问题的算法至少需要 $f+1$ 轮消息传递。所以在这个方面，没有算法能比 Lamport等人的算法执行更快。但是可以改善消息的复杂度，例如 Garay和Moses [1993]做的改进。

有些算法（例如 Dolev和Strong[1983]的算法）对消息进行签名。Dolev和Strong的算法也需要进行 $f+1$ 轮，但是发送消息的数量仅是 $O(N^2)$。

由于算法的复杂性和代价，因此建议只在安全威胁很严重的地方使用这些算法。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。如果解决方案所基于的错误模型的知识越详细，那么可以得到的解决方案更有效[Barborak et al. 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。

12.5.4 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（由此可推导出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。算法假定消息交换按轮进行，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。 506 ~ 507

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，进程可以随时发出响应的消息，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中断了再延续的情况，这阻止了进程达到共识。详细的证明已经超出本书的范围，这里不再细述。

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和可靠组播问题的方法。如果有这样的解决办法，根据12.5.1节的结论，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”这个词。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对同步性要弱，可以作为实际应用的系统的模型；但其同步性又比异步系统要强，使得共识问题能够被解决[Dwork et al .1988]。这个方法的介绍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三个方法：故障屏蔽、利用故障检测器达到共识、随机化进程的行为。

故障屏蔽 第一种完全避免不可能性结论的技术是屏蔽发生的所有进程故障（2.3.2节有故障屏蔽的介绍）。例如，事务系统使用持久存储，它能够从崩溃中恢复。如果一个进程崩溃，它会被重启（自动重启或者由管理员重启）。进程在程序的关键点的持久存储中保留了足够多的信息，以便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，故障屏蔽一般可应用到系统设计中。第14章讨论了事务系统如何利用持久存储。第15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 另一种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，在一个仅仅依靠消息传递工作的异步系统中，没有故障检测器是真正达到完美的。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，但是其余的进程认为它已经出错了。它们将接下来收到的所有从出错的进程发来的消息全部抛弃，从而把这个故障变成“失败-沉默”。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。这项技术被应用在ISIS系统中 [Birman 1993]。 508

该方法要求故障检测器是精确的。如果故障检测器不精确的话，系统在工作中可能放弃一个

成员，而实际上这个成员为系统的有效性做出贡献。遗憾的是，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（并且不能执行有用的工作）才能得出一个进程已经出错的结论。这个方法还会引起了另一个问题——网络分区，我们将在第15章讨论这个问题。

一个完全不同的方法是使用“不完美”的故障检测器，这种方法允许被怀疑的进程正确行动而不是排除它来达到共识。为了解决在异步系统中的共识问题，Chanadra和Toueg[1996]分析了一个故障检测器必须拥有的属性。他们证明，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问题是可以解决的。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：

最终弱完全性：每一个错误进程最终常常被一些正确进程怀疑。

最终弱精确性：经过某个时刻后，至少一个正确的进程从来没有被其他正确的进程所怀疑。

Chandra和Toueg证明，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12.1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，那么错误地怀疑一个进程的情况将变得很少。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相当相似。

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接受它们发出的消息并正常地处理。虽然这使得应用程序员的工作变得复杂，但是这样做的好处在于：正确的进程不会被错误地排斥出去而造成浪费。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），它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，或者减缓或加速进程，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。在有的情况下还是不能达到共识，但是这个方法使得进程能够在一个有限
509 的期望时间内达到共识。Canetti和Rabin[1993]提出了一个概率算法可以解决共识甚至拜占庭故障问题。

12.6 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，所以需要一个单独的分布式互斥服务。我们考虑了3种实现互斥的算法：一种使用中央服务器的算法，一个基于环的算法以及一个使用逻辑时钟的基于组播的算法。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，本章考虑了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唯一的一个进程——即使同时发生几个选举。例如，在主时间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意见）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，但是从原理上看，没有一个算法能比该算法采用的 $f+1$ 轮处理更好，除非消息采用数字签名。

本章最后描述了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。我们讨论了虽然有这样的结论，异步系统还能够达成一致的方法。

练习

12.1 使用一个不可靠的通信通道，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？ (第470页)

12.2 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？ (第473页)

12.3 根据同步延时给出计算互斥系统最大吞吐量的公式。 (第473页) 510

12.4 在用于互斥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。 (第474页)

12.5 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？ (第474页)

12.6 就基于环的算法，给出一个执行的例子，用以说明进程不必以发生在先顺序授权进入临界区。 (第475页)

12.7 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，描述如何提高它的性能。你的修改是否满足活性条件ME2？ (第477页)

12.8 在霸道算法中，恢复进程启动一次选举，并且如果它比当前的协调者进程有更高的标识符，那么它就成为新的协调者。这是算法所必需的吗？ (第482页)

12.9 如何修改霸道算法以处理两种情况：暂时的网络分区（通信变慢）以及处理变慢。 (第484页)

12.10 设计一个在IP组播上进行基本组播的协议。 (第486页)

12.11 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。 (第487页)

12.12 在图12-10中，如果颠倒以下两个语句的顺序："R-deliver m" 和 "if(q≠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？ (第488页)

12.13 解释为什么基于IP组播的可靠组播算法不适用于开放组也不适用于封闭组。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？ (第488页)

12.14 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，说明如何解决这些假设。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"消息，这个消息永远不会发给应用，而是当应用没有消息要发送的时候由协议发送。 (第488页)

12.15 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。 (第493页)

12.16 证明：如果我们在图12-14所示的基本组播算法中是FIFO序的，那么得到的全排序组播也是

511 因果排序的。任何一个为FIFO序并且是全排序的组播是不是也是因果序的？ (第494页)

12.17 考虑如何修改因果序的组播协议来处理重叠组。 (第497页)

12.18 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。 (第498页)

12.19 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。 (第498页)

12.20 从可靠全排序组播（涉及选择第一个可以传递的值）的解决方案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，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。 (第503页)

12.21 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，那么在一个将军有问题的情况下也可以达成协定。 (第505页)

12.22 解释如何修改IP组播上的可靠组播算法，从而消除保持队列，这样，收到的非重复的消息能马上被传递，但没有任何排序保证。提示：用集合而不是序号来表示到目前为止已经被

512 传递的消息。 (第489页)

第13章 事务和并发控制

本章将讨论事务和并发控制在服务器管理共享对象时的应用。

事务定义了一个服务器操作序列，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，它们因为具有更高的并发度，因而在分布系统中非常有用。

所有的并发控制协议都是基于串行相等的标准，它们都源于用来解决操作冲突的规则。本章描述了三种方法：

- 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。
- 乐观并发控制允许事务一直执行，直到它们准备提交为止，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它们的起始时间进行排序。

13.1 简介

事务的目标是在多个事务访问对象以及服务器面临故障的情况下，保证所有由服务器管理的对象始终保持一个一致的状态。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）行为。13.1.2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对象状态信息，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，能完全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的相关问题，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。13.1.1节将介绍对象的简单同步访问；13.2节将介绍事务，事务需要防止客户之间冲突的更高级的技术。13.3节讨论嵌套事务。13.4节～13.6节分别讨论单服务器上的事务的三种并发控制方法，即锁、乐观并发控制和时间戳排序。第14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13-1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，

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*
　返回账户中余额
setBalance(*amount*)
　将账户余额设置成amount

Branch接口中的操作
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
account接口和Branch接口的操作

图13-1 Account接口的操作

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，其接口为Branch，该接口提供创建新账户、通过名字查找账户和查询分行总余额等操作。

13.1.1 简单的同步机制（无事务）

本章涉及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。本节先讨论没有事务时客户操作如何同步。

服务器上的原子操作 从本书前面的章节，我们已经看到，使用多线程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，对象应该设计成支持多线程的上下文环境。以银行为例，如果deposit方法和withdraw方法在设计时没有考虑应用于多线程程序中，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为止。这种形式的同步将线程的执行分散到不同的时间中，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——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。

514
/
515

免受其他线程中执行的并发操作干扰的操作称为原子操作。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，对象上的操作仍然应该是原子操作。通过互斥机制，例如mutex，可实现这一点。

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。一些客户调用更新服务器上对象的操作，而另一些客户调用方法来访问对象便可实现上述目的。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：某个客户的操作要等到另一个客户操作结束后才能完成。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——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 notify和wait方法允许线程以一种能够解决上述问题的方式相互通信。这两个方法必须用于对象的同步方法中。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，因为调用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方法首先检查队列是否为空，如果队列为空则调用该队列的wait方法。因此在队列为空时，某个客户调用first方法将不会得到应答，必须等待其他客户向队列添加内容——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，并将队列中的第一个对象返回给客户。在线程通过wait和notify同步对

象操作时，对于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小节中，我们将讨论利用带同步操作的对象来实现一个事务锁。当某个客户试图获取一个锁时，它必须等待其他客户释放该锁。

如果没有这种线程同步机制，那么请求不能马上得到满足的客户，例如客户在一个空队列上调用first方法，会被告之以后重试。这种方式是不能令人满意的，因为它导致客户不断轮询服务器，服务器也要不断执行额外的请求。另外，服务器在处理这些轮询时，其他客户必须等待，这也造成了不公平。 516

13.1.2 事务的故障模型

Lampson[1981a]提出过一个分布事务的故障模型，包括了硬盘故障、服务器故障以及通信故障。该故障模型声称：可以保证算法在出现可预见故障时正确工作，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：

- 对持久性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。从持久性存储中读数据时可根据校验和来判断数据块是否损坏。
- 服务器可能偶尔崩溃。当一个崩溃的服务器由一个新进程替代后，它的可变内存被重置，崩溃之前的数据均已丢失。此后新进程执行一个恢复过程，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14.6节）。当一个处理器出现故障时，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持久存储，即它不会产生随机故障。服务器崩溃可能出现在任何时候，特别是在恢复时也可能出现。
- 消息传递可能有任意长的延迟。消息可能丢失、重复或者损坏。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。

利用这个关于持久存储、处理器和通信的故障模型能够设计出一个可靠系统，该系统的组件可对付任何单一故障，并提供一个简单的故障模型。特别是，*可靠存储*可以在出现一个write操作故障或者进程崩溃故障的情况下提供原子写操作。它是通过将每一个数据块复制到两个磁盘块上实现的。此时一个write操作作用于两个磁盘块上，在一个磁盘出现故障的情况下，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。

13.2 事务

在某些情况下，客户要求给服务器的一组请求是原子的，也就是说：

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，或者在服务器故障时不会产生任何影响。 517

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lookUp根据用户名查询到相应的银行账户，然后在相关账户上进行deposit、withdraw或者getBalance操作。我们的例子使用了账户名为A、B和C的三个账户。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。为了简化起见，我们略去了由名字查找账户和变量声明等细节。

```
Transaction T:
a.withdraw (100) ;
b.deposit (100) ;
c.withdraw (200) ;
b.deposit (200) ;
```

图13-2 一个客户的银行事务

图13-2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的动作。前两个动作是从账户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。

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 1998]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13-2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，它将服务器的数据从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 2003]，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上下文，该上下文随着操作调用而传递。在CORBA中，事务对象在事务作用域内被调用，通常有一些与它们相关的持久存储。

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。这样的事务常常被称作*原子事务*（见下面的讨论）。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终止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：

518

- 故障原子性：即使服务器崩溃，事务的效果也是原子的。
- 持久性：一旦事务成功完成，它的所有效果将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。

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”表示事务的下列属性：

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性状态转换到另一个一致性状态。

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，应由他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，一个对象持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要求，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，以达到全有或全无的要求。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持久存储中。

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事务。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柜员执行联机银行事务。

任何支持事务的服务器的目标是最大程度地实现并发。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。

事务功能可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，协调者实现了图13-3中的Coordinator接口。协调者为每个事务赋予一个事务标识符（TID）。客户调用协调者的openTransaciton方法来引入一个新事务——分配并返回一个事务标识符。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。 519

```
openTransaction( ) → trans;
  开始一个新事务，并返回该事务的唯一TID。该标识符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13-3 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，客户在每次调用中发送由openTransaction返回的事务标识符。一种可能的实现方式是将TID作为可恢复对象的每个方法的一个额外参数。例如，在银行服务中，deposit操作可能定义成：

```
deposit (trans, amount)
在TID为trans的事务中给账户存款amount
```

如果事务作为中间件提供，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。如果事务正常进行，那么closeTransaction的返回值表明事务被提交——它给客户一个承诺：事务所请求的所有更新都被永久记录。此后的其他事务访问同一数据时将看到这些更新的结果。

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）必须保证在持久存储中，在对象及其副本上清除所有效果，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abortTransaction调用）或服务器放弃事务。图13-4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。 520

成功执行	被客户放弃	被服务器放弃	
OpenTransaction	openTransaction		openTransaction
操作	操作		操作
操作	操作		操作
⋮	⋮	服务器放弃事务→	⋮
操作	操作		向客户报告ERROR
closeTransaction	abortTransaction		

图13-4 事务执行历史

进程崩溃时的服务器动作 如果服务器进程意外崩溃，它最终会被新的服务器进程替代。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意外崩溃的客户，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前还未完成的事务。

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，从而了解到服务器崩溃。如果在执行事务期间服务器崩溃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（可能通过人工干预等方式）来完成或放弃事务所在的任务。

13.2.1 并发控制

本节将用银行的例子说明并发事务中的两个著名问题——"更新丢失"问题和"不一致检索"问题。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
521 余额分别是$100、$200和$300。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两次以10%的幅度增加账户B的余额，B的最终值为$242。

下面来看看事务T和事务U并发执行的效果。如图13-5所示，两个事务获得账户B的余额$200，然后存入$20。结果是将账户B的余额提高了$20，而不是$42。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13-5的后半部分，我们列出了对相应账户余额有影响的操作（阴影部分）。我们假定某行上的操作在该行之前的行执行之后执行。

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		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10);	$80		
		c.withdraw(balance/10)	$280

图13-5 更新丢失问题

不一致检索 图13-6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，但是branchTotal计算A和B的总和，结果却是$300，这是错误的。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		
		total = a.getBalance();	$100
		total = total+b.getBalance();	$300
		total = total+c.getBalance();	
b.deposit(100)	$300		

图13-6 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结果也是正确的。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并发执行是否正确，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，我们能够解决更新丢失问题。图13-7列出了这样的一种交错执行，其中影响共享账户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交错执行是事务U在事务T开始之前完成它对账户B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		
*b.setBalance(balance*1.1);*	$220	*balance = b.getBalance();*	$220
		*b.setBalance(balance*1.1);*	$242
a.withdraw(balance/10);	$80		
		c.withdraw(balance/10);	$278

图13-7 串行等价地交错执行事务T和U

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13-6）的情况下，考虑与不一致检索有关的串行等价性的效果。不一致检索在某个检索事务与一个更新事务并发运行的时候出现。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的交错执行（如图13-8中的例子），可以防止不一致检索的发生。

522
523

事务 V:		事务 W:	
a.withdraw(100);		*aBranch.branchTotal();*	
b.deposit(100);			
a.withdraw(100);	$100		
b.deposit(100);	$300		
		total = a.getBalance();	$100
		total = total+b.getBalance();	$400
		total = total+c.getBalance();	
		...	

图13-8 串行等价地交错执行事务V和W

冲突操作 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13-9给出了read和write操作的冲突规则。

对任意两个事务，可以确定它们之间冲突操作的访问次序，那么，串行等价性可以从冲突操作角度定义如下：

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

在它们访问的对象上执行。

不同事务的操作		是否冲突	原 因
read	read	否	由于两个read操作的执行效果不依赖这两个操作的执行次序
read	write	是	由于一个read操作和一个write操作的执行效果依赖于它们的执行次序
write	write	是	由于两个write操作的执行效果依赖于这两个操作的执行次序

图13-9 Read和Write操作的冲突规则

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13-10列出了它们的一种交错执行过程。注意，每个事务相当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*的访问都在事务U对*i*访问之前进行，而U对变量*j*的访问都在事务T对*j*访问之前进行。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满足下面两个条件之一：

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13-10 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生成并发控制协议。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。大多数实际系统利用锁方法（参见13.4节的讨论）。使用锁方法时，对象在被访问之前，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该对象被解锁，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，就会产生死锁。关于死锁和它的补救方法，我们将在13.4.1节讨论。

13.5节将描述乐观并发控制。在乐观并发控制方案中，事务能够一直运行而不会被锁住。当它请求提交时，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是正确的。

524 - 525

时间戳排序将在13.6节描述。在时间戳排序中，服务器记录对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、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，一般通过让一个客户事务等待另一个事务或是重新运行事务或是两者的结合来实现并发控制。

13.2.2 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，但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。

本节以银行的例子阐述与事务放弃相关的两个问题。这两个问题是“脏数据读取”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read操作和write操作，在我们的例子中，getBalance是read操作而setBalance是write操作。

脏数据读取 事务的隔离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“脏数据读取”问题。考虑图13-11中的事务执行情况。事务T读取账户A的余额并为它增加$10，事务U也读取A的余额并给它增加$20。这两个事务的执行是串行等价的。现在假设事务U提交之后事务T被放弃，由于账户A的余额必须恢复到它的初始值，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进行了一次脏数据读取。因为它已经被提交，所以它不能被取消。 526

事务 T:		事务 U:
a.getBalance()		*a.getBalance()*
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*
balance = a.getBalance()	$100	
a.setBalance(balance + 10)	$110	
		balance = a.getBalance() $110
		a.setBalance(balance + 20) $130
		commit transaction
abort transaction		

图13-11 事务T放弃时的脏数据读取

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，那么服务器的状态就不可恢复。为了确保不出现这种情况，所有进行了脏数据读取的事务（例如U）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必须延迟到事务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13-11中，假设事务U推迟提交直到事务T被放弃，那么此时事务U也要放弃。遗憾的是，其他观察到U结果的事务同样也要放弃。这些事务的放弃可能导致后续更多的事务被放弃。这种情况称为连锁放弃，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对象。为了保证这一点，涉及对象的操作必须推迟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 考虑事务放弃隐含的另一种可能结果。它涉及两个事务针对同一个对象进行write操作。在图13-12中，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，图中的事务执行是串行等价的，事务T将余额更改为$105，事务U将余额更改为$110。如果事务U被放弃而事务T提交，那么余额将恢复为$105。

事务 T:		事务 U:
a.setBalance(105)		*a.setBalance(110)*
	$100	
a.setBalance(105)	$105	
		a.setBalance(110) $110

图13-12 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write操作的“前映像”。在我们的例子中，A的初始值是$100，它是事务T的write“前映像”，类似地，事务U的write前映像是$105。所以，如果事务U放弃了，我们可得到正确的账户余额$105。

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，但事务T的write“前映像”是$100，所以我们最终获得了$100的错误值。类似地，如果事务T先被放弃接着U也被放弃，由于U的write前映像是$105，所以我们得到的账户余额为$105，但是正确的数值应该是$100。

为了保证使用前映像进行事务恢复时获得正确的结果，write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。

事务的严格执行 为了避免"脏数据读取"和"过早写入"，通常要求事务推迟read操作和write操作。如果read操作和write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。为了达到这个目的，事务中所有的更新操作都是针对对象的挥发性存储中的临时版本完成。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，再从对象取值。

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。如果事务被放弃，系统将删除它的临时版本。

13.3 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称为顶层事务。除顶层事务之外的其他事务称为子事务。例如在图13-13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}。



图13-13 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。处于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13.4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其中一个接收者。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，结果将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。之所以称为平面的，是因为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（及其后代）可以并发运行，这提高了事务内的并发度。如果这些子事务运行在不同的服务器上，那么它们能够并行执行。例如，考虑银行例子中的branchTotal操作，可以通过在分行的每一个账户上调用getBalance来实现它。现在每次getBalance调用可以作为一个子事务实现，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复杂。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。如果决定是放弃，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决定提交。 527 ~ 529
- 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一个放弃）。在我们的例子中，事务T的提交将允许事务T_1、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到A
a.deposit(100)
b.withdraw(100)
```

事务Transfer包括两个子事务：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事务提交的情况。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，详情可参考Weikum[1991]。

13.4 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13-7的例子表明如何在某种程序的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。

一个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加锁。如果客户请求访问的一个对象已被其他客户的事务锁住，那么服务器将暂时挂起这个请求，直到对象被解锁。

图13-14说明了互斥锁的使用。它给出的事务与图13-7中的事务相同，但多出一列用于为每个事务列出加锁、等待和解锁的动作。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。 530

事务T： *balance = b.getBalance()* *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		事务U： *balance = b.getBalance()* *b.setBalance(b a l*1.1)* *c.withdraw(bal/10)*	
操作	锁	操作	锁
openTransaction			
bal = b.getBalance()	锁住B		
*b.setBalance(bal*1.1)*		*openTransaction*	等待事务T在B上的锁
a.withdraw(bal/10)	锁住A	*bal = b.getBalance()*	
closeTransaction	对A, B解锁	...	锁住B
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	锁住C
		closeTransaction	对B, C解锁

图13-14 事务T和U使用互斥锁

串行等价性要求一个事务对某个对象的所有访问相对于其他事务进行的访问而言是串性化的。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。每个事务的第一个阶段是一个“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”）。这称为两阶段加锁。

13.2.2节介绍了事务的放弃可能引起脏数据读取和过早写入问题，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才能进行。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。这称为严格的两阶段加锁。锁可以阻止其他事务读/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，而一个事务只访问其中少量的对象，不太可能与其他并发事务发生冲突。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客户账户都锁住，那么在任何时候，只有一个柜员能够进行联机事务——这是不可接受的限制。

531

对其访问必须被串行化的那部分对象的数量应尽可能少，即尽量限制与事务的每个操作相关的那部分对象。在银行例子中，分行包含众多账户，每个账户都有余额。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没有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13-9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，应给对象加上读锁。在事务进行写操作之前，给对象加上写锁。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为止——从不拒绝客户的请求。

由于不同事务的读操作不冲突，因此可以在已有读锁的对象上设置读锁。所有访问同一对象的事务共享它的读锁——正是这个原因，读锁有时也被称为共享锁。

操作冲突规则包括：

1）如果事务T已经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，那么给该对象加写锁的请求将被延迟。为了保证规则2，如果一个对象上有另一个事务的写锁，那么对该对象加读锁或写锁的请求将被延迟。 532

图13-15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。每个单元中的项分别指明，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。

对象一对象		被请求的锁	
		read	write
已设置的锁	none	OK	OK
	read	OK	等待
	write	等待	等待

图13-15 锁的相容性

不一致检索和更新丢失是在没有并发控制机制（如锁）的保护下，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。如果先执行检索事务，那么这个事务上的读锁将推迟更新事务的执行；如果后执行检索事务，那么检索事务对读锁的请求将推迟自身的执行，直到更新事务完成为止。

更新丢失在两个事务同时读取了对象的值，然后利用读取的数据来计算新值的时候出现。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，然后在写该对象时将读锁提升为写锁。这样，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。因此，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功能更强的锁，即互斥性更强的锁。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。因此写锁比读锁互斥性更强。锁可以被提升，因为结果是一个互斥性更强的锁。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13-16总结了在严格的两阶段加锁实现中锁的使用规则。为了保证遵守这些规则，客户不能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，而解锁则由事务协调者的commit或abort操作完成。 533

1. 当某个事务中有一个操作访问某个对象时，
   1）如果该对象未被加锁，那么它就加上锁并且操作继续执行。
   2）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，直到对象被解锁为止。
   3）如果该对象被其他事务设置了一个不冲突的锁，那么这个锁就共享并且操作继续执行。
   4）如果该对象已被同一事务锁住，那么在必要时提升该锁，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2）。
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施加的所有锁。

图13-16 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。该服务提供了一种将资源（例如可恢复对象）和一个锁的集合（称为锁集）相关联的方式。锁集支持获取和释放锁。锁集的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。事务性的锁集所支持的方法与锁集一致，但要求将事务标识符作为参数。我们在前面提到，CORBA的事务服务给所有在同一个事务中的客户请求都标上事务标识符。这就允许可恢复对象在被访问之前可以加上合适的锁。当事务提交或放弃时，事务协调者负责释放所有的锁。

由于锁一旦获取，就一直要保持到事务提交或放弃，所以图13-16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，并与某个对象相关联。图13-17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：

- 被锁住对象的标识符。
- 当前拥有该锁的事务的标识符（共享锁可以有若干拥有者）。
- 锁的类型。

534

类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13-17 Lock类

acquire方法实现了图13-15和图13-16给出的规则。它的两个参数分别是事务标识符和该事务请求的锁类型。它首先测试能否满足该请求。如果另一个事务以与之冲突的模式拥有锁，那么它调

用wait，将调用者线程挂起直到接收到相应的notify为止。注意，wait调用被放在一个while循环中，这是因为多个等待线程被通知但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设置适当的锁：

- 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。 535
- 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。

release方法的参数是需要释放锁的事务的标识符。该方法从锁的拥有者中删除该事务标识符，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，那么该方法通知所有等待的线程，使得它们能够继续执行。

图13-18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。

- setLock方法的参数指定了给定事务要锁住的对象和锁类型。它在散列表中查找该对象相应的锁，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，对每个锁分别调用release方法。

```
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13-18 LockManager类 536

一些策略问题：我们注意到，当若干线程等待同一个被锁住的项时，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，否则请求读锁总能成功。请读者考虑下面的问题：

- 如果不断面临读锁请求，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。这个问题有解决方法吗？

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，以便保证：

1）每个嵌套事务集是一个实体，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效果。

实施第一个规则要求子事务成功执行后，由它的父事务继承子事务所获得的所有锁，随后，这些被继承的锁继续由更高层的事务继承。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持到顶层事务提交或放弃，从而防止不同嵌套事务集的成员观察到其他事务集的部分效果。

下列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。
- 同层次的子事务可以并发执行，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。
- 当子事务提交时，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。

537

- 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这些锁。

注意，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13-13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。

13.4.1 死锁

使用锁有可能引起死锁。考虑图13-19中锁的使用。因为deposit和withdraw方法符合原子性，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13-19表示两个事务分别获取了一个账户的写锁，但在访问另一方锁定的账户时被阻塞。这就是死锁的情景——两个事务都在等待并且只有对方释放锁后才能继续执行。

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
…		…	
…		…	

图13-19 写锁造成的死锁

在客户涉及交互程序的情况下，死锁是一种常见的情形。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，层次化的加锁机制是非常有用的，参见13.4.2节。 538

死锁的定义 死锁是一种状态，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。等待图可用来表示当前事务之间的等待关系。在等待图中，节点表示事务，边表示事务之间的等待关系。例如，如果事务T在等待事务U释放某个锁，那么在等待图中有一条从节点T指向节点U的边。图13-20中的等待图表示了图13-19中的死锁的情形。回想一下，图中的死锁是由于事务T和U都试图获取对方拥有的锁造成的，因此事务T等待事务U，同时事务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13-20的右图表示事务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13-20所示的左图。

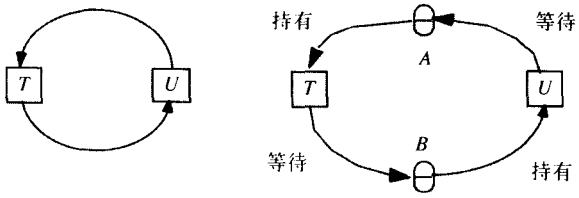


图13-20 图13-19的等待图

假设像图13-21一样，等待图中包含环路T→U→…→V→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以等待锁。由于没有一个锁会释放，因此这些事务均处于死锁状态。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13-21中的事务T被放弃，那么它将释放事务V正在等待的锁，即事务V将不再等待T。

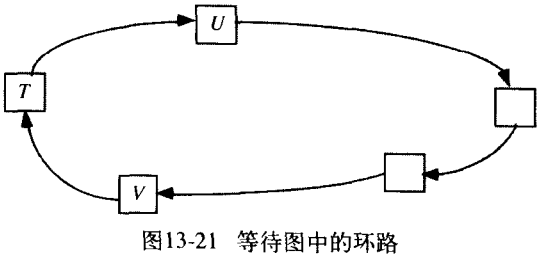


图13-21 等待图中的环路

如图13-22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正在等待获取对象B的锁。接着，事务T和W请求对象C上的写锁，那么会进入死锁状态：事务T等待U和V，V等待W，而W又在等待T、U和V，如图13-22的左图所示。这表明，尽管每个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→W→T→V和V→W→V中。

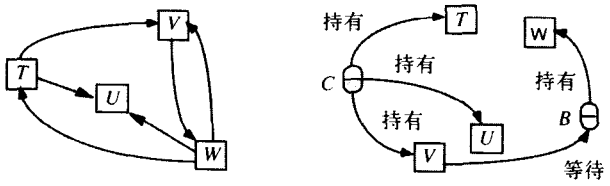


图13-22 另一个等待图

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。

预防死锁 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，有时在事务开始时无法预计事务将访问哪些对象。在交互应用中这种情形更为常见，因为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。死锁还可以通过以预定次序加锁来预防，但是这会造成过早加锁和减少并发度。

更新锁 CORBA的并发控制服务介绍了第三种类型的锁——更新锁，使用它是为了避免死锁。造成死锁的原因通常是：两个冲突的事务首先获得读锁，接着试图提升它们为写锁。一个在数据项上加更新锁的事务可以读该数据项，但该锁与其他事务加在同一数据项上的更新相冲突。这种类型的锁不能由读操作隐式地添加，而必须由客户添加。

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，必须选择放弃一个事务，从而打破环路。

539
~
540

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操作用于增加或删除等待图中的边。死锁检测软件在图13-22左图表示的时刻有下面信息：

事务	等待
T	U, V
V	W
W	T, U, V

当锁管理器因为事务T请求事务U已锁住对象上的锁而阻塞请求时，在等待图中增加边T→U。注意，如果锁被共享，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，将边T→U从等待图中删去。练习13.14包含了死锁检测实现的详细讨论。如果一个事务共享一个锁，那么该锁不被释放，但通向某个事务的边被删除了。

每次有新边加入等待图时，就检测一下是否存在环路。为了避免不必要的开销，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节点和边也被删除。这发生在被放弃的事务删除其锁的时候。

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所处的环路的数量。

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。一旦超过这个期限，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对象会被继续锁住。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会产生很多问题：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，长时间运行的事务经常被放弃。另外，很难确定适当的超时时间长度。相比之下，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能决定放弃哪一个事务。

利用锁超时，我们可以解除图13-19中的死锁，如图13-23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。

541

当事务访问的对象分布在不同的服务器上时，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14.5节讨论。

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13-23 图13-19中死锁的解除

13.4.2 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，但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。

双版本加锁 这是一种乐观策略，它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。一个事务在其他未完成事务正在读取对象时，不能立即提交它对同一对象的写操作。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。 542

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13-24所示。

在性能方面，双版本加锁和普通的读-写锁机制有两个主要区别。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中）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会推迟其他事务的提交。

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13-24 锁的相容性（读锁、写锁和提交锁）

层次锁 对于某些应用，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操作有所不同，它读取所有账户的余额值，因此应该在所有账户上加上读锁。为了减少加锁开销，应当允许有混合粒度的锁。

Gray[1978]提出使用具有不同粒度的层次锁。在每一层，设置父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，支行是父节点，而账户是子节点（如图13-25所示）。

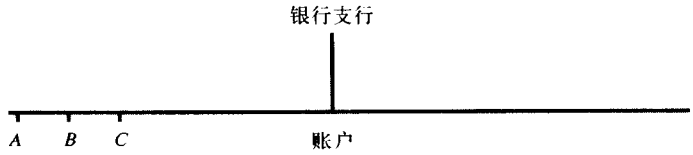


图13-25 银行例子的锁层次

混合粒度的锁在日记系统中很有用，这里的数据按照每周的天数分成不同部分，而每天的数据又可以继续按时间段进行细分，如图13-26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（例如该星期每一天的所有时间段上）的写操作。



图13-26 日记的锁层次

在Gray的机制中，层次中的每个节点都可以加锁——此后，锁的拥有者能显式访问该节点并隐式访问它的子节点。在图13-25所示的例子中，对支行的读/写锁隐含地对所有账户加上了读/写锁。在给子节点加上读/写锁时，需要在它的父节点和祖先节点（如果有）上设置一个读/写试图锁。这个试图锁和其他类型的试图锁是相容的，但是和读/写锁冲突。图13-27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——该锁结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13-27 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13-27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。

混合粒度的锁允许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，而一个短事务只需锁住细粒度的数据。

CORBA并发控制服务支持可变粒度的加锁，包括试图读和试图写锁类型。它们可按上述方式使用，从而利用在层次结构化数据中应用不同粒度锁的好处。

13.5 乐观并发控制

Kung和Robinson[1981]指出了锁机制的许多固有的不足，并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，这些开销在不支持对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），它不可能改变数据的完整性，通常仍然需要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1。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，并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到同一个对象的概率只有$1/n$，因此每n个事务只有1个才真正需要加锁。
- 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的并发度。

Kung和Robinson提出的另一个方法是一种“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。事务总是能够执行，就好像事务之间不存在冲突一样。当客户完成其任务并发出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：

543 ~ 545

- *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拷贝。使用临时版本，事务便可以在工作阶段放弃或者在与其他事务发生冲突不能通过验证时放弃（而不产生副作用）。读操作总是可以立即执行——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，访问对象最新提交的值。写操作将对象的新值记录成临时值（这个临时值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，每个事务还维护被访问对象的两个集合：*读集合*包含事务读的所有对象；*写集合*包含事务写的对象。注意，所有的读操作都是在对象的提交版本（或它们的副本）上执行，因此不会出现脏数据读取。
- *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而言是串行等价的，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——一个事务总是在序号比它小的事务之后完成它的工作阶段。也就是说，如果$i<j$，那么事务号为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分配事务号，那么一个事务若在另一个具有更小事务号的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之间的操作冲突完成的。事务T_v对重叠事务T_i而言是可串行化的，那么它们的操作必须符合下面的规则：

546

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处于验证和更新阶段，当任何两个事务都不会在更新阶段重叠时，规则3自动满足。注意，在写操作上的这个限制和不发生脏数据读取这个事实，将产生事务的严格执行。为了防止重叠，整个验证和更新阶段被实现成一个临界区，使得每次只能有一个客户执行。为了增加并发度，验证和更新的部分操作可以在临界区之外实现，但是必须串行地分配事务号。我们注意到，在任何时刻，当前的事务号就像一个伪时钟，每当事务成功结束，这个时钟就产生一次嘀嗒。

事务的验证必须保证事务T_i和T_j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_i验证之前进行，因此它们不会受当前事务写操作的影响（满足规则1）。T_i的验证过程将检查它的读集（受T_i的读操作影响的对象）是否和其他较早的重叠事务T_j的写集是否重叠（规则2）。如果存在重叠，验证失败。

设startTn是事务T_i进入其工作阶段时系统已分配（给其他已提交事务）的最大事务号，finishTn是T_i进入验证阶段时系统已分配的最大事务号。下面的程序描述了T_i的验证算法：

```
boolean valid = true;
for (int Tj = startTn + 1; Tj <= finishTn; Tj++) {
    if (Tj 的读集与Ti写集的交 valid = false;
}
```

图13-28给出了T_i验证过程中需要考虑的重叠事务。时间从左至右增加。T_1、T_2和T_3是较早提交的事务。T_1在T_i开始之前提交，T_2和T_3在T_i完成其工作阶段前提交，并且有$startTn+1=T_2$，$finishTn=T_i$。向后验证过程必须比较T_i的读集和T_2、T_3的写集。

547

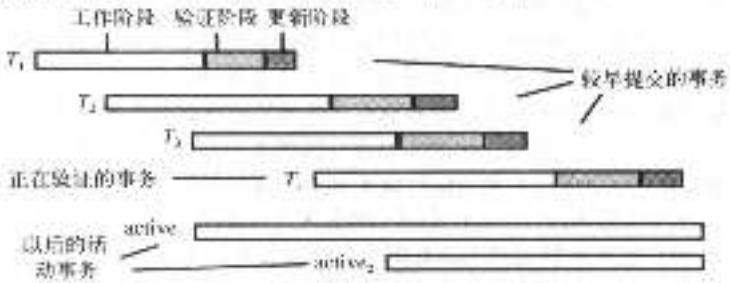


图13-28 事务的验证过程

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，解决冲突的唯一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，直到没有可能发生冲突的未验证重叠事务。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，这个列表由事务服务维护。注意，这个列表按事务号排序。如果有长事务存在，较早事务的写集合的保留将是一个问题。例如在图13-28中，T_1、T_2、T_3和T_i的写集合必须保留到活动事务active，结束之后。值得注意的是，尽管这个活动事务有事务标识符，但它还没有事务号。

向前验证 在事务T_i的向前验证中，T_i的写集合要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_i完成

之前不会进行写操作。设活动事务具有（连续的）事务标识符（从$active_1$～$active_N$），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13-28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改变。）由于被验证事务的读集合没有包括在验证过程 548
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，所以发生冲突时，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，直到冲突事务结束为止。但是这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，还是有可能启动会产生冲突的活动事务。
- 放弃所有有冲突的活动事务，提交已验证的事务。
- 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被放弃，因此这种策略会造成被验证事务的不必要放弃。

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活动事务的读集合比较。我们注意到，向后验证涉及存储已提交事务写集合（直到不再需要它们为止）的开销。另一方面，向前验证不得不允许在验证过程中开始新事务。

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启动事务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从而产生冲突。这种阻止事务最终提交的现象称为饥饿。

出现饥饿的情形很少，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被多次放弃后，能够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互斥访问。

13.6 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要先进行验证。如果该操作不能通过验证，那么事务将被立即放弃，然后由客户重新启动该事务。每个事务在启动时被赋予一个唯一的时间戳。这个时间戳定义了该事务在事务时间序列中的位置，来自不同事务的 549
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。

这个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，同时它也必须保证对象的临时版本按事务的时间戳所决定的顺序提交。这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来实现的。这些写操作可在closeTransaction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给时间戳赋值，或者利用前面介绍的“伪时间”来给时间戳赋值，伪时间基于一个计数器，每次获取时间戳的请求都会使计数器加一。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，我们将在第14章中进行讨论。

下面我们描述SDD-1[Bernstein et al. 1980]系统中采用的并由Ceri和Pelagatti[1985]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）对象的写时间戳比它的所有临时版本都要早，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入到读时间戳集合中。当事务被提交时，临时版本的值就变成对象的值，临时版本的时间戳变成相应对象的时间戳。

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，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之前的操作相冲突，T_i的时间戳表明它们应该比T_c晚。图13-29给
550 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$T_c \geqslant$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

图13-29 时间戳排序中的操作冲突

时间戳排序的写规则：通过结合规则1和规则2，我们可以得到下列规则，该规则用于决定是否接受事务T_c对对象D执行写操作：

```
if (Tc ≥ D的最大读时间戳 && Tc > 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Tc;
else /* 写操作太晚了 */
    放弃事务 Tc;
```

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这个版本，否则服务器创建一个新的临时版本并且为其标记上写时间戳T_c。值得注意的是，“到达太晚的”写操作将引起事务放弃，这里的“太晚”是指具有后来时间戳的事务已经读或写了这个对象。

图13-30说明了事务T_3的写操作的执行情况，其中$T_3 \geqslant$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，$T_3 >$对象的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。

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，该规则用于决定马上接受、等待或拒绝事务T_c对对象D执行读操作的请求：

```
if (Tc > D提交版本的写时间戳) {
    设Dselected是D的具有最大写时间戳的版本 ≤ Tc;
    if (Dselected已提交)
            在Dselected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，然后重新应用读规则；

} else

放弃事务T_i；

551

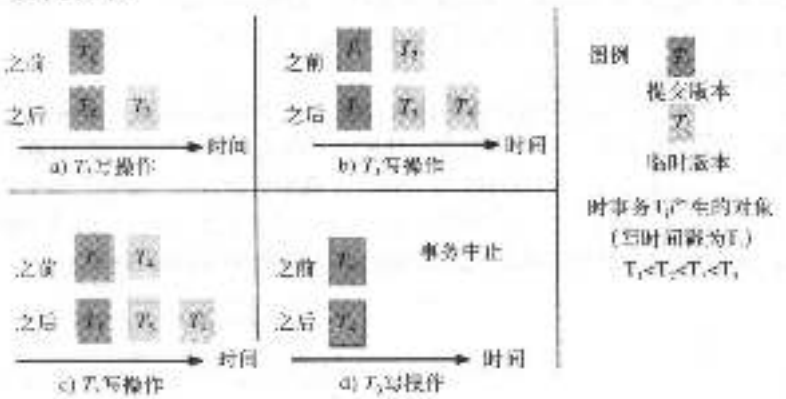


图13-30 写操作和时间戳

注意以下几点：

- 如果事务T_i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成。如果较早的事务已提交，则T_i的读操作将针对对象的已提交版本。如果较早的事务被放弃，那么T_i将重复读规则（选择以前的版本）。这个规则可防止脏数据读取。
- “到达太晚的”读操作将被放弃，太晚是指具有后来时间戳的事务已经写了相应的对象。

图13-31说明了时间戳排序的读规则，图中共有4种情况，分别标记为a～d，它们均用于说明事务T_3的读操作动作。在每种情况下，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。在情况a和b中，读操作针对提交版本——在a中，该提交版本是对象的唯一版本；而在b中，有一个临时版本属于另一个较晚的事务。在情况c中，读操作针对临时版本，并且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_2被放弃。

552

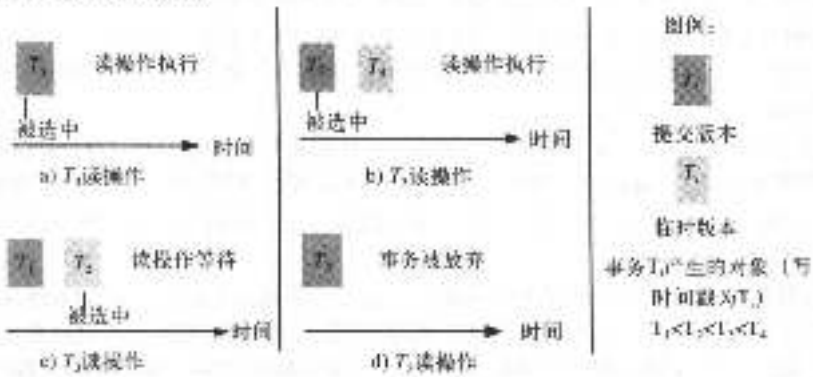


图13-31 读操作和时间戳

当一个协调者收到提交事务的请求后，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。必须按照时间戳顺序创建每个对象的提交版本。因此，协调者在写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（参见13.2节）。时间戳排序的读规则要求事务对对象的读操作等待，直到所有写该对象的较早事务提交或者放弃为止。对象的提交版本也按时间戳顺序排列，以保证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为止。

图13-32说明了如何利用时间戳排序方法来控制图13-7中的并发银行事务T和U。其中列A、B和C分别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，
553 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。图中的例子表示当事务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）。

		对象的不同版本及其时间戳					
T	*U*	*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13-32 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，但是它容易造成事务重启动。一个被称为“忽略过时写”规则的修改方案提供了一种改进方法。它对时间戳排序的写规则做了如下一些改动：

- 如果写操作来得太晚，那么直接忽略这个操作，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，如果其他事务读取了该对象，那么这个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来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会被拒绝。

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，用于记录读该版本的事务的最大时间戳。和以前一样，每当一个写操作被接受后，它将针对与事务写时间戳相应的临时版本进
554 行操作。每当执行读操作时，它将针对具有小于该事务时间戳的最大写时间戳的版本进行操作。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。

当读操作太迟到达时，允许它读取一个较早的已提交版本，这样就没必要放弃这个读操作了。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13.22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时都针对所访问对象的已提交版本。这样，就不需要时间戳排序的冲突规则2了，仅留下下面的规则：

规则1 T_i不能写事务T_j读过的对象，其中$T_i > T_j$。

如果对象的某个版本的读时间戳大于T_i，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_i的写时间戳时才会这样。（这个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作用于对象最近的一个版本上，所以服务器查看具有小于或等于T_i的最大写时间戳的对象版本$D_{selected}$。以下规则用于执行事务T_i在对象D上执行写操作的请求：

```
if (D_selected 的读时间戳 <T_i)
    在D的临时版本上完成写操作，并标记上写时间戳T_i
else
    放弃事务T_i
```

图13-33说明了一个写操作被拒绝的例子。图中的对象有两个写时间戳为T_1和T_2的提交版本，该对象收到下列对对象进行操作的请求序列：

T_3 read; T_4 write; T_5 read; T_6 write;

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_4请求一个写操作，生成一个写时间戳为T_4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_2的版本（小于T_3的最高的时间戳）。

4）T_6请求一个写操作，由于写时间戳为T_5的版本的读时间戳T_5大于T_6，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_6。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。）

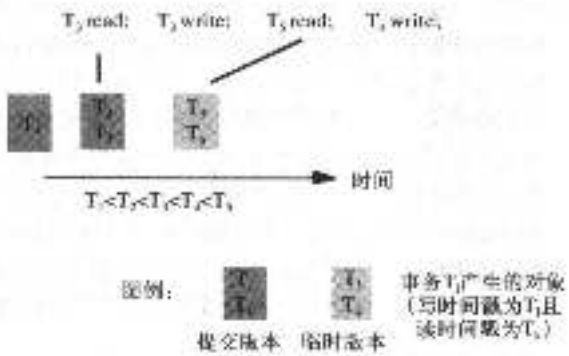


图13-33 过迟的写操作将使读操作失效

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，必须定期删除旧版本。尽管多版本时间戳排序方法会带来存储空间的开销，但这种方法既可以使并发度有极大提高，又不会造成死锁。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，请参见Bernstein等的文章[1987]。

13.7 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。所有的方法都会带来时间和空间的开销，并且它们都在一定程度上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，即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，即根据对象被访问的顺序确定串行顺序。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，那么两阶段加锁的性能更好。

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这个现象，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

两阶段加锁方法进行并发控制。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象访问冲突时有不同的解决策略。时间戳排序将立即放弃事务，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。

555
~
556

在使用乐观并发控制时，所有的事务被允许执行，但是其中的一些事务在试图提交时被放弃；如果采用向前验证，那么会在更早的时候放弃事务。如果并发事务之间的冲突较少时，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，而时间戳排序也已应用于SDD-1数据库系统中。这两种方法都用于文件服务器中。然而，在分布式系统中对数据访问进行并发控制的主流方法还是加锁，例如，前面提到的CORBA的并发控制服务就完全基于锁的使用。特别的，它提供层次加锁，这种方式允许对层次化结构数据进行混合粒度的加锁。

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、时间戳排序和针对长事务的新方法。

研究人员发现，上述并发控制机制在两个应用领域中是远远不够的。其中一个领域是多用户应用，其中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。这种应用要求数据在并发更新和服务器崩溃时能够保证原子性，事务技术似乎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需求：(1) 用户要求在其他用户更新数据时马上得到通知，这和隔离性是相违背的。(2) 用户要求在其他用户完成事务前就能访问对象，这引发了新型锁的开发，这种锁用于在对象被访问时触发动作。在这个领域的工作提出了不少放宽隔离性和提供更新通知的方案，这些工作的综述请参见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，这些对象版本从一个公共数据库中取出，在工作结束时再放回。对象版本之间的合并需要用户之间的协同。有关这方面工作的综述可参见文献[Barghouti and Kaiser 1991]。

13.8 小结

面对事务的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。实现原子性第一个方面的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性存储中，以便事务服务能从进程崩溃中恢复。为了在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。为了保证事务可自行选择是提交还是放弃，事务中的操作是针对其他事务不可
557
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性存储中。

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性，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，只有在死锁时才应用放弃事务。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。

时间戳排序利用了上述3种策略，它根据事务开始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。

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，直到事务完成为止。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，

它的好处是允许使用多种策略解决冲突。在乐观并发控制甚至在时间戳排序中，由于不能通过验证的事务不断地放弃，从而引起饥饿。

练习

13.1 TaskBag是一个提供“任务描述”仓库的服务。它支持在几台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。

当一个任务当前不可用，但可能不久就可用的时候，客户发出takeTask请求。讨论下列方法的优缺点：

（1）服务器马上回答，告诉客户以后重试。

（2）让服务器操作（和客户）等待，直到任务变成可用为止。

（3）使用回调。 （第516页） 558

13.2 一个服务器管理对象$a_1, a_2, \cdots a_n$，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的值设置为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。 （第523页）

13.3 针对练习13.2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。 （第527页）

13.4 操作create在银行分行中插入一个新的银行账户。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，请给出账户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？ （第523页）

13.5 练习13.4中新创建的账户Z有时被称为*假象*（phantom）。在事务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象。

13.6 “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之间的串行等价交错执行的数目和T和U的串行等价交错执行的数目。

559 试解释为什么嵌套事务比非嵌套事务串行等价交错执行的数目更多？ （第523页）

13.7 考虑练习13.6中嵌套事务的恢复问题。假设withdraw事务在账户透支时将放弃，因而父事务也被放弃。请给出满足下列条件的T_1、T_2、U_1和U_2的串行等价执行：（1）这是一个严格执行；（2）非严格执行。考虑严格的执行在多大程度上减少嵌套事务的并发度？ （第523页）

13.8 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：

- Read(i) 返回对象a_i的值。
- Write(i, *Value*) 将对象a_i的值设置为Value。

事务T和U定义如下：

T: *x=read(i); write(j, 44);*
U: *write(i, 55); write(j, 66);*

请给出一个事务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。 （第531页）

13.9 练习13.8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？

T	*U*
x=read(i);	
	write(i,55);
write(j,44);	
	write(j,66);

a)

T	*U*
x=read(i);	
write(j,44);	
	write(i,55);
	write(j,66);

b)

T	*U*
	write(i,55);
	write(j,66);
x=read(i);	
write(j,44);	

c)

T	*U*
	write(i,55);
x=read(i);	
	write(j,66);
write(j,44);	

d)

（第531页）

13.10 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13.8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);*

560 其中对象a_i和a_j的初值分别是10和20。 （第528页）

13.11 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才能进行。请解释图13-16中的加锁规则是如何保证严格执行的。 （第534页）

13.12 如果事务完成所有操作后但在提交前就释放写锁，请描述此时如何引起不可恢复的状态。 （第528页）

13.13 如果事务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13-16中规则2。 （第528页）

13.14 考虑单个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，何时从等待图中删除边。

利用练习13.8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上的写锁时，T和V都在等待获取这个写锁。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。 （第540页）

13.15 考虑图13.26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？应该按照什么次序设置这些锁？释放这些锁也按照上述次序吗？

当查看w周的每天的时间段时需要设置哪些锁？在已为某个约会设置了锁的时候，能这样做吗？ （第543页）

13.16 考虑将乐观并发控制应用于练习13.9中的事务T和U的情况。如果T和U同时处于活动状态，试描述以下几种情况的结果：

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。 （第545页） 561

13.17 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);	

在使用具有向后验证的乐观并发控制时，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。

（第545页）

13.18 试比较练习13.8中事务T和U分别在两阶段加锁（练习13.9）和乐观并发控制（练习13.16）中的操作执行顺序。

13.19 考虑将时间戳排序用于练习13.9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。对于每个操作需描述：

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？ （第549页）

13.20 对于下面的事务T和U的交错执行，重新考虑练习13.19中的问题。

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);	

（第549页）

562 13.21 利用多版本时间戳排序，重新考虑练习13.20。 （第554页）

13.22 在多版本时间戳排序中，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。 （第554页）

13.23 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？ （第554页）

13.24 试比较练习13.8中事务T和U分别在两阶段加锁（练习13.9）和乐观并发控制（练习13.16）

563 中的操作执行次序。 （第545页）

第14章 分布式事务

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所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。本章将描述两阶段提交协议，它是最常用的原子提交协议。

分布式事务的并发控制一节将讨论为支持分布式事务如何扩展加锁、时间戳排序和乐观并发控制。

使用加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，它的作用是在出现故障之后服务器被替代时，用它来恢复服务器所管理的对象上的事务的效果。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性存储中。 564 ~ 565

14.1 简介

第13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该事务。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样的结果。协调者的工作方式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的相互通信，以便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13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，以保证事务在本地是串行化的。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加锁、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，但同时，由于不同服务器之间存在相互依赖循环，因此可能出现分布式死锁。

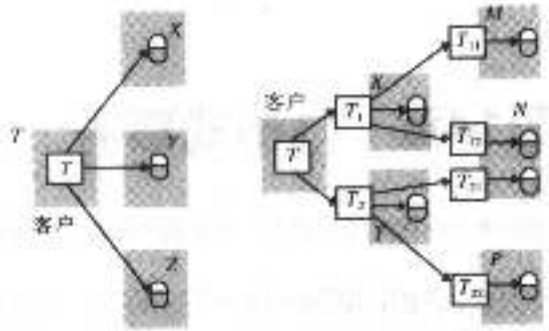
事务恢复用于保证事务所涉及的所有对象都是可恢复的。除此之外，它还保证对象只反映已提交事务所做的更新，不反映被放弃事务所做的更新。

14.2 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送请求。例如，在图14-1a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，子事务可以进一步地以任意深度嵌套子事务。图14-1b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又创建子事务T_{11}、T_{12}、T_{21}和T_{22}，这4个子事务分别访问服务器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，因此它们能并行运行。同样，T_{11}、T_{12}、T_{21}和T_{22}也可以并发执行。 566



a) 平面事务 b) 嵌套事务

图14-1 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到C账户，然后从B账户转账$20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14-2所示），那么4个请求（两个deposit操作和两个withdraw操作）可以并行运行，从而整体执行性能优于4个操作被顺序调用的简单事务。

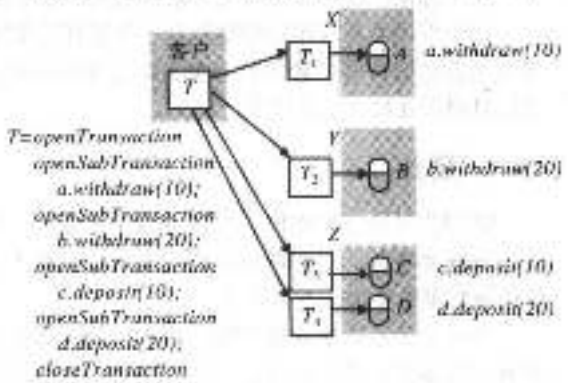


图14-2 嵌套的银行事务

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，向任意一台服务器上的协调者发出一个openTransaction请求，参见13.2节的描述。该协调者处理完openTransaction请求后，将事务标识符（TID）返回给客户。分布式事务的事务标识符在整个分布式系统中必须是唯一的。构造TID的一种简单方法是将TID分成两部分：创建该事务的服务器的标识符（例如IP地址）和一个对该服务器来说是唯一的数字。

创建某一分布式事务的协调者成为该分布式事务的协调者，它在分布式事务结束时负责提交或放弃事务。管理分布式事务访问的对象的每个服务器都是该事务的参与者。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，每一个参与者也记录一个对协调者的引用。

图13-3给出的Coordinator接口提供了一个额外的方法join，它用于将一个新的参与者加入当前事务：

567
~
568

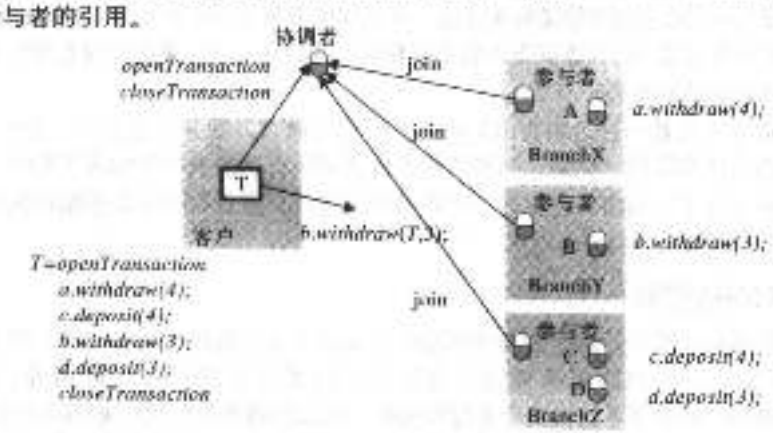
join (Trans, reference to participant)

通知协调者一个新的参与者已加入到事务Trans

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14-3显示了一个客户，它的（平面）银行事务涉及服务器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到账户C，然后从账户B转账$3到账户D。将图中左边的事务T展开，我们可以看到事务T的openTransaction和closeTransaction操作被发送给协调者。

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）将通知参与者对象自己属于事务T。如果在这之前没有通知过协调者，则参与者对象调用join操作来通知协调者。在这个例子中，我们看到事务标识符作为一个额外的参数传递，这样，接收者能将它传递给协调者。在客户调用closeTransaction时，协调者就拥有了对所有参与者的引用。



注意：协调者在其中的某一个服务器上，例如BranchX上。

图14-3 一个分布式银行事务

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。

14.3 原子提交协议

事务的提交协议最初于20世纪70年代提出。而两阶段提交协议是由Gray[1978]提出的。事务的原子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，在客户请求提交时，该协议不允许任何服务器单方面放弃事务，阻止服务器提交它那部分事务的原因通常与并发控制问题有关。例如，如果使用加锁，为了解除死锁需要将事务放弃，客户有可能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，协调者可能不知道某个服务器已经崩溃并且已被替换——而这个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自己的那部分事务。由于事务原子性的要求，如果部分事务被放弃，那么整个分布式事务也必须被放弃。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提交协议中它自己那一部分，即使参与者出现故障而被中途替换掉。一个事务的参与者如果最终能提交事务，那么可以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）保存到持久性存储中。

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。

问题是要保证每个参与者都投票，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不能通信）时能够正常工作。

提交协议的故障模型 13.1.2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，在该环境下服务器可能崩溃，消息也可能丢失。但是，提交协议假设底层请求-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所发送的消息。

两阶段提交协议是一种达到共识的协议。第12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所替代，新进程的状态根据持久性存储中保存的信息和其他进程所拥有的信息来设定。

14.3.1 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）请求被直接发送给协调者。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，那么它将把所有的更新和它的状态记录到持久存储中——也就是准备好提交。一完成这些工作，它就同意提交事务。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14-4总结的操作进行通信。其中，canCommit、doCommit和doAbort方法是参与者接口中的方法，而方法haveCommitted和getDecision位于协调者接口中。

569 ~ 570

canCommit?(trans)→ Yes / No
协调者用该操作询问参与者它是否能够提交事务，参与者将回复它的投票结果。
doCommit(trans)
协调者用该操作告诉参与者提交它那部分事务。
doAbort(trans)
协调者用该操作告诉参与者放弃它那部分事务。
haveCommitted(trans, participant)
参与者用该操作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，参与者用该操作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14-4 两阶段提交协议中的操作

两阶段提交协议由投票阶段和完成阶段组成，如图14-5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，事务实际上已经结束。在步骤3a处，协调者和参与者提交事务，因此协调者将事务提交的决定通知客户；在步骤3b发生时，协调者将放弃事务的决定通知给客户。

在步骤4，所有的参与者确认它们已提交，这样协调者能知道它所记录的事务信息何时将不再需要。

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。要处理可能的崩溃，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14.6节讨论。

阶段 1 （投票阶段）：
1）协调者向分布式事务的所有参与者发送canCommit? 请求。
2）当参与者收到canCommit? 请求后，它将向协调者回复它的投票（Yes或者No）。在投Yes票之前，它在持久性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送的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14-5 两阶段提交协议

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阻塞。当进程检测到超时后，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阶段，协调者或参与者都会遇到这种情景：不能处理它的那部分协议，直到接收到另一个请求或应答为止。

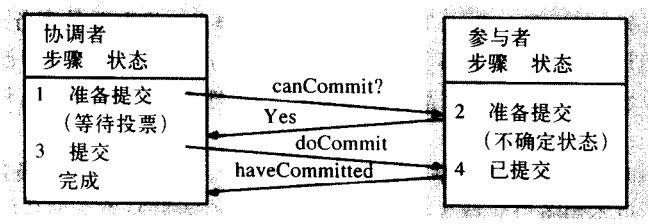


图14-6 两阶段提交协议中的通信

首先考虑这样的情形：某个参与者投Yes票并等待协调者发回最终决定，即告诉它是提交事务还是放弃事务。参见图14-6的步骤2。这样的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释放以用于其他事务。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，它才能进行图14-5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，直到协调者被替代为止，这可能导致处在不确定状态的参与者长时间地延迟。 571 ~ 572

不依靠协调者获取最终决定的方法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14.5和Bernstein et al.[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。

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，该参与者只能（通过锁超时）检测到它是否长时间未收到任何有关该事务的操作请求。因为在这个阶段还没有做出任何决定，所以参与者可以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，因此在等待一段

时间后它可以决定放弃该事务。但是它必须给所有发送了投票的参与者发送doAbort消息。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，它们将进入前面描述的不确定状态。

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haveCommited消息时仍能正确运行——它们的作用只是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haveCommited消息计算在内。

在最坏的情况下，两阶段提交协议在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协议完成的时间限制，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。

对于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。这些延迟主要源于协调者故障或者不能从参与者那里得到getDecision请求的回答。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次的消息往返。关于三阶段提交协议的详细情况，请参见练习14.2和
573 Bernstein等[1987]。

14.3.2 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*。在图14-1b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。每个子事务在其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，它独立决定是临时提交还是放弃。临时提交和准备好提交是不同的：它只是一个本地决定，也不用备份到持久存储中。如果服务器随后崩溃，那么该服务器的替代者不能提交。在所有子事务完成后，临时提交的事务参与到一个两阶段提交协议中，其中，临时提交子事务所在的服务器表示要提交的意图，而那些有放弃祖先的子事务将被放弃。准备好提交确保一个子事务能够提交，而临时提交仅意味着它正确完成了，如果随后被问及是否提交，它将可能同意提交。

如图14-7所示，子事务的协调者提供创建子事务的操作，并提供操作用来使子事务的协调者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，该操作返回一个唯一的子事务标识符。 *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14-7 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符。客户调用openSubTransaction操作创建子事务，该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父事务，并返回一个新创建子事务的标识符。

子事务的标识符必须是其父事务TID的扩展，子事务标识符的构造方法应使得能根据子事务的标识符确定父事务或顶层事务的标识符。另外，所有子事务的标识符必须是全局唯一的。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13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还是放弃采取不同的动作。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。我们假设当某个账号透支时，withdraw事务将被放弃，相应的Transfer事务也被放弃。但是放弃某个Transfer子事务并不要求放弃整个未结算订单事务。相反，顶层事务将发现Transfer子事务执行失败并执行相应的动作。

考虑图14-8所示的顶层事务T和它的子事务（图14-8基于图14-1b）。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}也必须被放弃。假设顶层事务T不管T_2被放弃的事实，最终决定提交，同时T_1不管T_{11}被放弃的事实，仍决定提交。

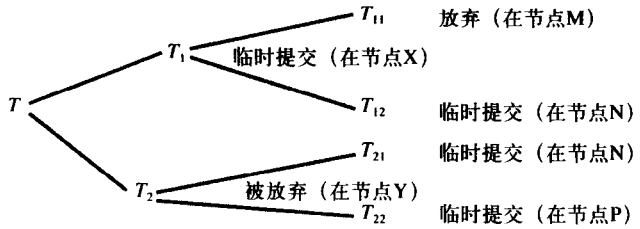


图14-8 事务T决定是否提交

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唯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后代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，而不用报告后代事务的任何信息。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，而被放弃的子事务则不在这个列表中。

图14-8给出的例子中的各协调者持有的信息在图14-9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，它把该事实报告给它的父事务T，但不传递它的子事务T_{21}和T_{22}的状态。如果事务的某个祖先事务被显式放弃了或者由于其协调者崩溃而被放弃，那么这个子事务被称为孤儿。在我们的例子中，子事务T_{21}和T_{22}都是孤儿，因为它们的父事务被放弃，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。

574 ~ 575

事务的协调者	孩子事务	参与者	临时提交列表	放弃列表
T	T_1, T_2	yes	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	yes	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	no(被放弃)		T_2
T_{11}		no(被放弃)		T_{11}
T_{12}, T_{21}		T_{12}(不包含T_{21})	T_{21}, T_{12}	
T_{22}		no(父事务被放弃)	T_{22}	

图14-9 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的角色，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14-8中，事务T的协调者、事务T_1和T_{12}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的状态保存到持久存储中来准备提交。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执行。

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。

层次化两阶段提交协议 在这种方法中，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作为子事务的直接父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其后代事务的应答，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。由于事务T_2被放弃，因此协议没有向它的协调者发送消息。图14-10给出了canCommit?需要的参数。其中，第
576 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与第二个参数中的TID相匹配的子事务。例如，由于T_{12}和T_{21}运行在同一个服务器上，因此T_{12}的协调者也是T_{21}的协调者，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。

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是顶层事务的标识符。参与者用Yes票或者No票来回复

图14-10 层次化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，那么它将准备对象并且回复Yes。如果没有找到匹配的子事务，那么它在执行子事务之后系统必定出现过崩溃，因此它将回复No。

平面两阶段提交协议 在这种方法中，顶层事务的协调者向临时提交列表中的所有子事务的协调者发送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，寻找能够匹配那个TID的事务或子事务。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（N）上。

但是，当服务器N上有临时提交子事务和放弃子事务并存时，这种方法不能为协调者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}都处于临时提交状态，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，因此提交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14-11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，它进行下面的操作：

- 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（将事务状态和它的对象记录到持久存储中）。
  - 如果子事务的祖先在abortList中，放弃这些子事务。
  - 向协调者发送Yes。
- 如果参与者没有任何顶层事务的临时提交子事务，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。

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。参与者用Yes票或者No票来回复

图14-11 平面两阶段提交协议中的canCommit?调用

两种方法的比较 层次化协议的优点在于，在任何阶段，参与者只需查找其直接父事务的子事务，而平面协议要求提供一个放弃列表来去除那些祖先已放弃的子事务。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直接和所有的参与者进行通信，而层次化事务需要按嵌套关系来传递一系列消息。

超时动作 与非嵌套事务的两阶段提交协议一样，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4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：它们没必要得到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——它被临时提交，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超时时间后将进行查询。图14-7中的getStatus操作可支持子事务查询它的父事务是否提交/放弃。为了保证这些查询是可能的，已放弃的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，那么它将最终放弃。

14.4 分布式事务的并发控制

每个服务器要管理很多对象，它必须保证在并发事务访问这些对象时，这些对象仍保持一致性。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对对象的冲突操作，事务T都在事务U之前。

14.4.1 加锁

在分布式事务中，某个对象的锁总是本地特有的（在同一个服务器中）。是否加锁是由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，还是让发出请求的事务等待。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。在使用加锁机制的并发控制中，原子提交协议进行时对象始终被锁住，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，锁可以提早释放。

577 ~ 578

由于不同服务器上的锁管理器独立设置对象锁，因此，对不同的事务，它们的加锁次序可能不一致。考虑下图中事务T和事务U在服务器X和服务器Y之间的交错执行：

T	U
Write(A) 在服务器X上对A加锁	
	Write(B) 在服务器Y上对B加锁
Read(B) 在服务器 Y 上等待U	
	Read(A) 在服务器A上等待T

事务T锁住了服务器X上的对象A，而事务U锁住了服务器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的锁。因此，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14.5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，协调者将得到通知，并且它将放弃该事务涉及的所有参与者上的事务。

14.4.2 时间戳并发控制

对于单服务器事务，协调者在它开始运行时分配一个唯一的时间戳。通过按访问对象的事务的时间戳次序提交对象的版本来保证串行等价性。在分布式事务中，协调者必须保证每个事务附上全局唯一的时间戳。全局唯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送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次序，协调者必须就时间戳排序达成一致。时间戳是一个二元组<本地时间戳，服务器id>对。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。

即使各服务器的本地时钟不同步，也能保证事务之间的相同顺序。但是为了效率的原因，各
579 协调者之间的时间戳还是要求大致同步。如果是这样的话，事务之间的顺序通常与它们实际开始的时间顺序相一致。利用第11章中的本地物理时钟同步方法可以保证时间戳的大致同步。

当利用时间戳机制进行并发控制时，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，相应的协调者将得到通知，并且它将在所有的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，正常情况下参与者同意提交。参与者不同意提交的唯一情形是参与者在事务执行过程中崩溃过。

14.4.3 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。事务在验证开始时首先要附加一个事务号，事务的串行化是根据这些事务号的顺序实现的。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的交错执行，它们分别访问服务器X和Y上的对象A和B：

T	U
Read(A) 在服务器X上	Read(B) 在服务器Y上
Write(A)	Write(B)
Read(B) 在服务器Y上	Read(A) 在服务器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13.5节介绍的简化的验证协议中，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。

13.5节中介绍的验证协议假设验证过程很快，这在单服务器事务的情况下是成立的。但在分布式事务中，由于两阶段提交协议需要一定的时间，因此在获得一致提交决定之前，可能推迟其他事务进入验证过程。在分布式乐观并发控制中，每个服务器使用并行验证协议。这是对向前及向后验
580 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，正在被验证事务的写集合必须和较早启动的与被验证事务重叠的事务的写集合进行检查，看两者是否重叠。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，如果服务器只是进行独立验证，同一个分布式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发生。一个解决方案是在每个服务器完成本地验证后，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满足串行化要求，换言之，这些事务不会形成环路。

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相同的全局唯一的事务号[Schlageter 1982]。两阶段提交协议的协调者负责生成全局唯一的事务号，并将此事务号通过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，对生成的事务号有个统一的排序。

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，并且结合了称为MVGV（多版本通用验证）的算法。MVGV是一种并行验证，它确保事务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。MVGV还允许事务号改变，以使更多的可能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事务的算法。它与Schlageter的方案类似，同样需要全局唯一的事务号。在读阶段结束时，协调者发布一个全局事务号，每个参与者试图用这个事务号来验证它们的本地事务。但是，如果发布的全局事务号太小，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号。如果没有找到合适的事务号，那么参与者只能放弃事务。最终，如果所有的参与者能够验证它们的事务，那么协调者将收到每个参与者发来的事务号；如果这些事务号相同，那么事务就能提交。

14.5 分布式死锁

在13.4节中有关死锁问题的讨论表明，单服务器在使用加锁机制进行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——因为设定合适的超时间隔很困难，它会导致事务不必要地放弃。利用死锁检测方法，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而实现的。在 581
包含多个事务访问多个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。全局等待图中的环路在局部等待图中可能不存在，也就是说，可能出现分布式死锁。等待图是有向图，其节点表示事务和对象，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14-12表示3个事务U、V和W的交错执行，它涉及服务器X上的对象A和服务器Y上的对象B，以及服务器Z上的对象C和D。

图14-13a的等待图表明一个死锁环路由不同的边组成，这些边分别代表某个事务等待某个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删除对象节点，从而将等待图简化为14-13b。

U		*V*		*W*	
d.deposit(10)	锁住*D*				
		b.deposit(10)	在节点*Y*		
a.deposit(20)	在节点*X*		锁住*B*		
	锁住*A*				
				c.deposit(30)	在节点*Z*
b.withdraw(30)	在节点*Y*				锁住*C*
	等待	*c.withdraw(20)*	在节点*Z*		
			等待		
				a.withdraw(20)	在*X*处等待

图14-12 事务U、V和W的交错执行

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13章提到局部等

待图可以由每一个服务器上的锁管理器构造。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）

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。

一种简单的解决方案是使用集中式死锁检测，其中的一个服务器担任全局死锁检测器。全局
582 死锁检测器通过收集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。一旦发现环路，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。

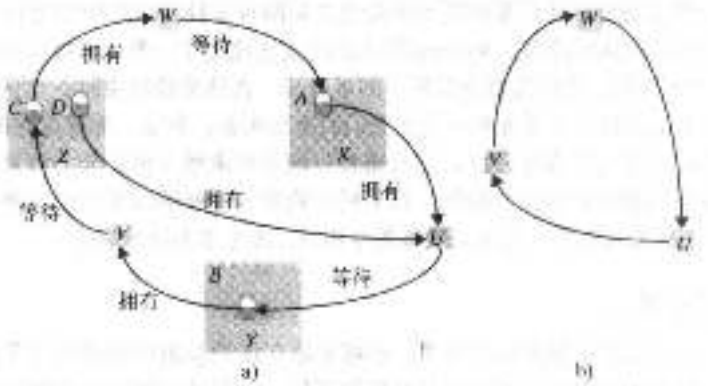


图14-13 分布式死锁

集中式死锁检测并不是一个非常好的方法，最主要的问题是它依赖单一的服务器来执行检测。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。而且，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。

假死锁 如果“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，等待关系的信息在服务器之间传递。如果确实存在死锁，那么最终有一个节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，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。

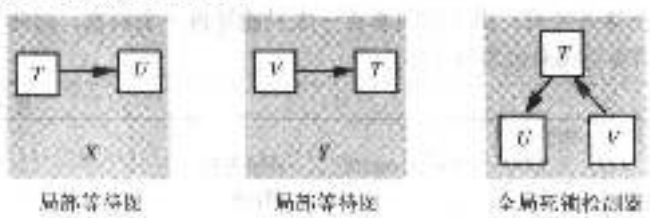


图14-14 局部等待图和全局等待图

考虑图14-14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。而且，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→U并不存在，但仍然检测出环路T→U→V→T，这就是一个假死锁。

细心的读者可能意识到在采用两阶段加锁的情况下，事务不能在释放对象后获取新的对象，因此假死锁也就不会出现。现在来考虑检测到环路T→U→V→T之后，要么表明这是一个死锁，要

么表明T、U和W最终都会提交。但实际上，它们中的任何一个都不能提交，因为它们彼此相互等待永远不会释放的对象。

如果在死锁检测过程中等待死锁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，如果有一个环路T→U→V→T，但在收集到U的信息后，事务U被放弃，由于环路被打断，也就没有死锁。

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关于边的信息。服务器通过转发probe（探询）消息来发现环路，这些探询消息沿着分布式系统的图的边发送。一个探询消息包含全局等待图中表示路径的一个事务等待关系。

问题是：服务器何时发送探询消息？考虑图14-13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上边W→U，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$V\rightarrow T_1\rightarrow T_2\rightarrow\cdots\rightarrow W\rightarrow U\rightarrow V$的一部分，它涉及使用其他服务器上对象的事务。这意味着可能存在分布式死锁环路，可以通过向服务器Y发送探询消息找到这个死锁环路。

现在来考虑当服务器Z将边V→W加入它的局部等待图之前的情景：此时，W并不在等待。因此不需要发送探询消息。

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通信。在任何时刻，事务或者是活动的，或者在某个服务器上等待。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那里获取这些信息。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，该事务的所有边也从局部等待图中删除。

583 ~ 584

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<T→U>的探询消息来启动一次检测过程，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，那么探询消息将被发送到这些锁的拥有者。有时，有些事务可能会在稍后共享该锁，这时，探询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询消息并确定是否有死锁产生，以及是否需要转发探询消息三个步骤。

例如，当对象所在的服务器接收到探询消息<T→U>（表示T正在等待拥有本地对象的事务U）时，它检查U是否也在等待。如果U也在等待另一个事务（例如V），那么V就添加到探询消息中（成为<T→U→V>），如果V也在等待另外的对象，那么继续转发探询消息。

就这样，全局等待图上的路径被逐一构造出来。在转发探询消息之前，服务器将检测当事务（以T为例）加入到等待序列后是否会使探询消息产生环路（例如<T→U→V→T>）。如果图中产生环路，那么就检测出死锁。

死锁解除：当检测出环路后，环路中的某个事务将被放弃以打破死锁。

在我们的例子中，下面的步骤描述了在相应的检测阶段，如何开始死锁的检测过程，以及探询消息的转发过程。

- 服务器X发起死锁检测过程，向对象B的服务器Y发送探询消息<W→U>。
- 服务器Y收到探询消息<W→U>后，发现对象B被事务V拥有，因此将V附加在探询消息上，产生<W→U→V>。由于V在服务器Z上等待对象C，因此该探询消息被转发到服务器Z。
- 服务器Z收到探询消息<W→U→V>，并且发现C被事务W拥有，那么将W附加在探询消息后形成<W→U→V→W>。

这个路径上包含一个环路，服务器会检测出一个死锁。必须放弃环路中的某个事务来解除死锁。可根据事务优先级来选择被放弃的事务。

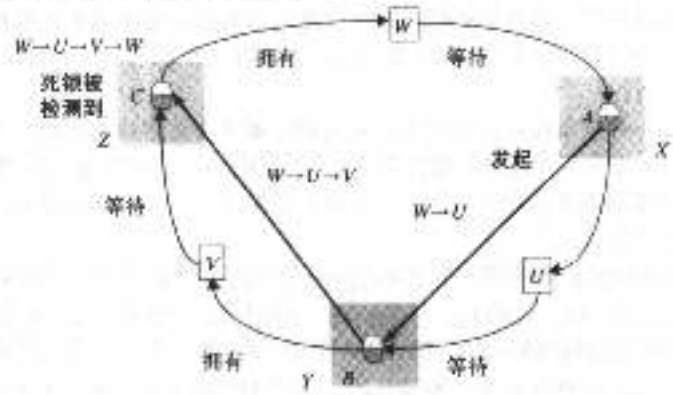


图14-15 传递探询消息来检测死锁

图14-15表示了从对象A的服务器发出探询消息并最终在对象C的服务器上检测出死锁的过程。其中探询消息用粗箭头表示，对象用圆圈表示，事务协调者用矩形表示。每个探询消息直接连接两个对象。在实现中，在服务器发送探询消息到另一个服务器之前，它首先将询问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的服务器发送探询消息<W→U→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询消息，事务协调者再将消息转发到事务等待的对象所在的服务器。在我们的例子中，对象B的服务器发送探询消息<W→U→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，转发一个探询消息需要发送两个消息。

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，服务器发出一个探询消息给W正在等待的对象的服务器。探询消息的接收者扩展这个消息并将这个消息转发到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询消息中，除非检测出死锁。当死锁出现后，W就间接地在等待自己。这样，探询消息将返回到W拥有的对象。

看起来，为了检测出死锁，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询消息，而每个探询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）。

如果一个死锁涉及N个事务，那么检测该死锁的探询消息需要被（$N-1$）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，绝大多数死锁只涉及两个事务，因此不用考虑过量的消息。这个结论来源于数据库领域的研究。它也可扩展到考虑对象冲突访问的概率问题上，参见[Bernstein et al. 1987]。

585 ~ 586

事务优先级 在上面的算法中，死锁涉及的每个事务都可能发起死锁检测。环路上的多个事务同时发起死锁检测会造成死锁检测在多个服务器上被执行，会使得多个事务被放弃。

在图14-16a中，考虑事务T、U、V和W，事务U正在等待事务W，V正在等待T。几乎在同一时刻，T请求U拥有的对象，W请求V拥有的对象。两个独立的探询消息<T→U>和<W→V>由这些对象的服务器同时发起和转发，最终由两个不同的服务器检测出死锁。在图14-16b中，等待环路是<T→U→W→V→T>。在图14-16c中，等待环路是<W→V→T→U→W>。

为了保证环路中只有一个事务被放弃，应给每个事务都附加一个优先级，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，它们仍然可以就放弃哪一个事务达成一致的决定。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，那么不管检测到环路<T→U→W→V→T>还是环路<W→V→T→U→W>，事务W都将被放弃。

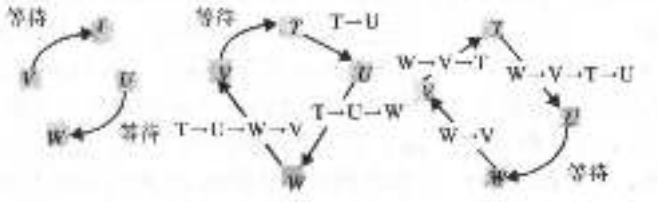


图14-16 同时发起两个探询消息

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14-16的例子中，由于T>U，因此发生探询消息<T→U>，而由于W<V，因此不能发出探询消息<W→V>。如果我们假设事务开始等待另一个事务时，等待事务的优先级比被等待事务的优先级高或低的概率相同，那么利用上面的死锁检测发起规则，可以减少一半探询消息。

事务优先级还可以用来减少探询消息转发的次数。一般的想法是探询消息只能“向下”传递，即从高优先级的事务到低优先级的事务。为了达到这一目的，服务器不会将探询消息转发到比发起者优先级还高的事务。这是由于如果目标事务正在等待另一个事务，那么它在开始等待时一定已经通过发送探询消息而发起了死锁检测。 587

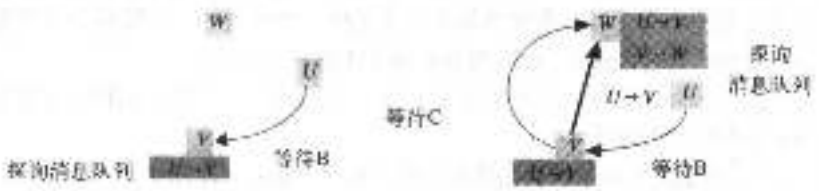


图14-17 探询消息向下传递

然而，这种明显的改进存在一个缺陷。在图14-15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。如果不使用优先级规则，在W开始等待时发起死锁检测，探询消息为<W→U>；如果使用优先级规则，因为W<U，所以不发出探询消息，死锁就不能检测出来。

这里的问题是由事务开始等待的次序决定死锁是否被检测出来。为避免上面的缺陷，协调者可以将所有收到的代表每一事务的探询消息存储在探询队列中。当事务开始等待一个对象时，它将探询消息转发到对象所在的服务器，由它将探询消息向下传递。

在图14-15的例子中，当U开始等待V时，V的协调者将保存探询消息<U→V>，见图14-17a。接着，当V开始等待W时，W的协调者将保存<V→W>，并且V将它的探询队列<U→V>发给W。在图14-17b中，W的探询队列包含<U→V>和<V→W>。当W开始等待A时，它就转发它的探询队列<U→V→W>到A的服务器，该服务器发现新的依赖W→U，并将它和已收到的信息合并，发现U→V→W→U，从而将死锁检测出来。

当一个算法要求将探询消息存储在探询队列中时，同时要求将探询消息传递到新的服务器并且丢弃已提交或已放弃时事务的探询消息。如果相关的探询消息被丢弃，某些死锁就有可能不被发现；另一方面，如果过期的探询仍然保留，就可能检测出假死锁。这样边追逐算法会变得很复杂。对算法的细节有兴趣的读者可参见[Sinha and Natarajan 1985]和[Choudhary et al. 1989]，其中给出了使用排他锁的算法。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，
588 [Kshemkalyani and Singhal 1991]又更正了Choudhary等人的算法（该算法不能检测出所有的死锁，而且可能报告假死锁），并且提供了一个更正后算法的正确性证明。在随后的一些文献中，[Kshemkalyani and Singhal 1994]指出，由于分布式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记录的信息来自不同的时间。另外，在死锁发生时，节点可能得到信息，但是这些节点得到死锁被解除的信息的时间却会被延迟。该文献利用在不同场地的事件之间的因果关系，描述了一种在分布式共享内存中的分布式死锁。

14.6 事务恢复

事务的原子性要求所有已提交事务的效果反映在事务所访问的对象中，而这些对象不呈现所有未提交或放弃事务的效果。这个特性可以从两方面加以描述：持久性和故障原子性。持久性要求对象被保存在持久性存储中并且一直可用。因此，如果客户提交请求得到确认，那么事务的所有影响就被记录到持久存储和服务器（的挥发性）对象中。故障原子性要求即使在服务器出现故障时，事务的更新作用也是原子的。事务恢复就是保证服务器上对象的持久性并保证服务提供故障原子性。

虽然文件服务器和数据库服务器将数据保持在持久性存储中，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。本章假设在服务器运行时，它的所有对象都存放在挥发性存储中，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存储中，而在挥发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（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，恢复管理器应能够应对介质故障。所谓介质故障是指，由于软件崩溃、持久存储老化或持久存储故障，造成恢复文件故障，以至于磁盘数据丢失。此时，我们需要恢复文件的另一个拷贝。当然，这也可以在稳定存储中实现，如利用位于不同场地的镜像磁盘或异地拷贝，
589 使持久存储不太可能出现故障。

意图列表 每一个提供事务支持的服务器都需要记录被客户事务访问的对象。第13章提到，当客户创建一个事务时，与之联系的服务器首先提供一个新的事务标识符，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符作为一个参数传递。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，每个事务的意图列表都记录了该事务修改的对象的值和引用列表。当事务提交时，它的意图列表用来确定所有受影响的对象，然后事务将用对象的临时版本替换成对象的提交版本，并将对象的新值写入到服务器的恢复

文件中。当事务放弃时，服务器利用意图列表来删除该事务形成的对象的所有临时版本。

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。我们讨论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；如果都准备好提交，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列表中的对象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，参与者服务器的恢复文件必须保存足够的信息。这样即使服务器在准备好提交和提交之间出现崩溃故障，也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所涉及的服务器的恢复问题，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事务联系在一起。图14-18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	事务标识符，事务的状态（准备好、已提交、已放弃）和其他用于两阶段提交协议的状态
意图列表	事务标识符和一系列意图记录，每个意图记录由<对象标识>、<对象值在恢复文件中的位置>组成

图14-18 恢复文件中的记录类型

两阶段提交协议中的事务状态值将在14.6.4节讨论。下面将介绍恢复文件的两种常用方法：日志方法和影子版本方法。

14.6.1 日志

在日志技术中，恢复文件包含该服务器执行的所有事务的历史。该历史由对象值、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、已提交或已放弃的顺序。实际上，恢复文件将包含服务器上所有对象的值的一个最近快照，随后存放该快照后的事务历史。

在服务器的正常操作过程中，当事务处于准备提交、提交或放弃状态时，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追加到恢复文件中，后面是事务的当前状态（准备好）和意图列表。当该事务最终提交或放弃时，恢复管理器将事务相应的状态追加到恢复文件。

我们假定恢复文件的追加操作是原子的，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。为了有效利用磁盘，可以将几次连续的写操作缓冲起来，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机写盘操作的速度快。

所有*未提交*的事务在崩溃后全部放弃。因此，当事务提交时，它的“提交”状态应强制写入日志文件，即连同其他缓冲的内容一并写入日志。

恢复管理器给每个对象附上唯一的标识符，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14-19表示了图13-7的银行业务中事务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的值分别写到日志位置P_1和P_2处，紧接着“已准备好”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，“已准备好”事务状态和意图列表（<C, P_5>, <B, P_6>）也被写入日志。

590 ~ 591

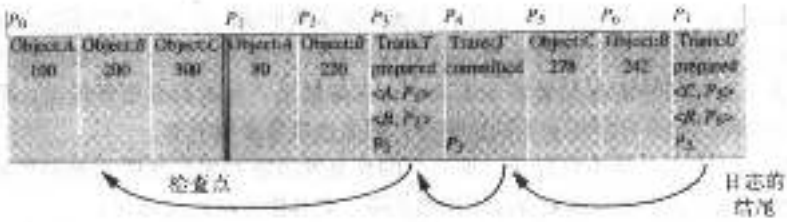


图14-19 银行服务例子的日志

在恢复文件中，每个事务状态记录都包含一个指针，指向恢复文件中前一个事务状态记录的位置。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执行的所有已提交事务的效果，而不包含任何未完成或放弃的事务的效果。

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，接着读取每一个对象的值，将它们与意图列表相关联，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恢复文件来恢复服务器的对象值，恢复文件中有一个向后指针从一个事务状态指向下一个事务状态。恢复管理器用具有已提交状态的事务来恢复还没有被恢复的对象，直到它恢复了所有服务器上的对象为止。这种方式的优点是每个对象只需恢复一次。

为了恢复事务的效果，恢复管理器从恢复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在恢复文件中的位置。

以图14-19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
592 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，它读取前一个事务状态记录（在P_5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_0>, <B, P_2>），从而读取P_1和P_2处的记录来恢复A和B的值。此时，还未恢复对象C的值，它移回到检查点P_6处，恢复C的值。

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。对于每个准备好提交的事务，它在恢复文件中追加一个放弃事务状态记录。这样可以保证每个事务总是处于已提交或已放弃状态。

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的，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可变内存中，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，可变内存中只有一个缓存，因此在服务器崩溃后被替换时，持久存储中的有些对象可能已经过期。这样，它的恢复管理器必须恢复这些持久存储中的对象，如果它在恢复过程中又崩溃，部分已恢复的对象可能还在那里，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，它有时需要重组恢复文件。如果恢复文件一直不重组，那么恢复过程必须逆向搜索整个恢复文件，直到找到所有对象的值为止。从概念上说，恢复过程需要的信息只需包含所有对象的提交后版本的拷贝，这是恢复文件最简洁的形式。检查点过程是将当前所有已提交的对象的值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的事务的意图列表（包括两阶段

提交协议相关的信息）。术语检查点指由该过程存储的信息。设置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，在检查点写入完毕之前，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，拷贝（1）标记点前与未完成事务有关的内容，（2）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，可能遇到恢复文件中的检查点。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。 593

14.6.2 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文件（即日志）中。影子版本是另一种恢复文件的组织方式。它利用一个映射来定位在版本存储文件中的某个对象版本。这个映射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。每个事务写入的版本均是前面提交版本的影子版本。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，并保留对象的相应的已提交版本不变。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在其中输入影子版本的位置。接着，用这个新映射表替换旧映射表即完成提交过程。

当服务器因崩溃而被替换时，要恢复对象，由恢复管理器读取映射，并使用映射中的信息来定位版本存储中的对象。

图14-20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，这时账户A、B和C的余额分别是$100、$200和$300。表的第二列表示事务T提交后的映射。

事务开始时的映射			事务提交后的映射			
$A \rightarrow P_0$			$A \rightarrow P_1$			
$B \rightarrow P_0'$			$B \rightarrow P_2$			
$C \rightarrow P_0''$			$C \rightarrow P_0''$			
	P_0	P_0'	P_0''	P_1	P_2	P_3	P_4
版本存储	100	200	300	80	220	278	242
	检查点						

图14-20 影子版本

版本存储包含一个检查点，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B和C的影子版本，分别在P_3和P_4处。

映射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中，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到新映射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。在恢复过程中，由于影子版本方式在映射中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。但在系统的正常操作过程中，日志操作应该更快，这是因为日志操作只需向同一个文件追加日志记录，而影子版本需要额外的相对稳定存储（涉及不相关磁盘块）的写操作。 594

影子版本对于处理分布式事务的服务器而言还是不够的。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
$A \rightarrow P_1$
$B \rightarrow P_2$
$C \rightarrow P_0''$

事务状态文件（持久存储）

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被更新之间的这段时间内，服务器有可能崩溃，此时，客户不会得到通知。在服务器崩溃后被替换的时候，恢复管理器必须允许出现这种可能性。遇到这种情况，可以检查映射是否包含了在事务状态文件中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。

14.6.3 为何恢复文件需要事务状态和意图列表

设计不包含事务状态信息和意图列表的简单恢复文件是可能的，这种恢复文件适用于单服务器上的事务。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。而且，对于非分布式事务的服务器，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使服务器设计更加复杂。如果对象可以从意图列表引用的话，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并告之客户，此时对象必须写入恢复文件（参见第13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允许提交，这是因为它等待的事务或者已经提交，或者由于服务器故障已被放弃。

595

14.6.4 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，以处理服务器故障时执行两阶段提交协议的事务。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14-6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。另外还使用了两种记录类型，以便让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。

记录类型	记录内容的描述
协调者	事务标识符，参与者列表
参与者	事务标识符，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追加了一个“准备好”状态记录），它的恢复管理器在恢复文件中追加一个“协调者”记录。每个参与者在它投Yes票之前，必须已经处于准备提交的状态，即在恢复文件中追加一个“准备好”状态记录。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”事务状态。当它投No票时，则在恢复文件中追加“已放弃”事务状态。

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。这必须是一次强制写入。参与者的恢复管理器根据从协调者收到的消息，在恢复文件中分别追加“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

恢复管理器向恢复文件中写入“完成”状态。这次写入不是强制要求的。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14-21表示了用于事务T的日志文件的内容，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”状态记录）。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。

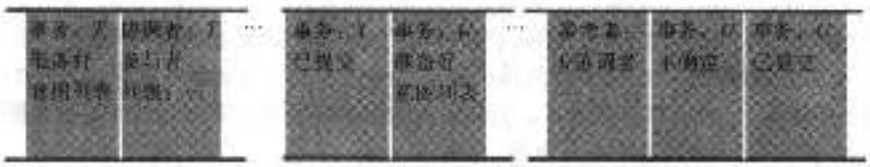


图14-21 与两阶段提交协议相关的日志记录

当服务器因崩溃而被替代之后，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而言，恢复管理器应当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，恢复管理器应当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故障时的状态采取动作，如图14-22所示。

角色	状态	恢复管理器的动作
协调者	准备好	由于在服务器发生故障时尚未做出任何决定，因此向参与者列表中的所有服务器发送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已经做出决定要提交事务。因此向参与者列表中的所有参与者发送doCommit命令，继续执行两阶段提交协议的第4步
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。当它获得回复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

596 - 597

图14-22 两阶段提交协议的恢复

恢复文件的重组 在执行检查点的过程中，需特别注意不能将未完成的协调者从恢复文件中删除。这些信息必须在所有参与者确认它们已完成事务之前一直保留。事务状态是“已完成”的记录可以被丢弃。状态是“不确定”的参与者也必须保留。

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。在两阶段提交协议中，当每个参与者准备提交时，它将它的对象和意图列表写入本地的恢复文件，并且在这些记录上附上顶层事务的标识符。尽管嵌套事务使用两阶段提交协议的变种，但恢复管理器使用的事务状态值和平面事务是一样的。

但是，如果相同或不同嵌套层次上的子事务访问了相同的对象，那么事务放弃和恢复过程将变得复杂一些。13.4节描述的加锁方案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。这种加锁方案要求父事务和子事务在不同的时刻访问公共数据对象，并确保并发子事务对同一对象的访问必须是串行化的。

根据嵌套事务规则访问的对象由各子事务提供临时版本来保证其可恢复性。嵌套事务的子事务所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的服务器按栈方式组织临时版本——每个嵌套事务使用一个栈。

每当嵌套事务中的第一个子事务访问对象时，该事务获得对象当前提交版本的一个临时版本，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，否则这个栈实际上不需要真正产生。

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把它重新入栈。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，它的父事务将继承这个新版本。为了实现这一点，子事务的版本和父事务的版本都从栈中丢弃，而将子事务的新版本重新放入栈中（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（如果有的话）将成为新的提交版本。

例如，在图14-23中，假设事务T_1、T_{11}、T_{12}和T_2以T_1，T_{11}，T_{12}，T_2的次序访问同一个对象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的提交版本的A_1被推入栈。当T_{11}开始执行时，它基于A_1上的A_{11}版本，并将A_{11}推入栈；当它完成时，它替换栈中父事务的版本。事务T_{12}和T_2按类似的方式执行，最终T_2的结果被留在栈顶。

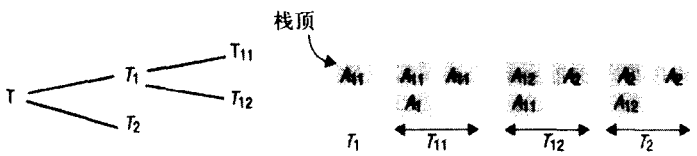


图14-23 嵌套事务

14.7 小结

在大多数情况下，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。在分布式事务中，可以使用嵌套事务结构，以便支持更高的并发度，并允许服务器独立提交。

事务的原子性要求参与分布式事务的所有服务器或者全部提交或者全部放弃。原子提交协议用于保证这一点，即使在执行过程中出现服务器崩溃的情况也可以保证原子性。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自己对象的事务的串行化。但是在保证事务全局串行化时需要额外的协议。使用时间戳排序的分布式事务需要一种时间戳排序生成方法，以便在多个服务器之间保持统一的时间戳排序。使用乐观并发控制的事务则需要一种全局验证或一种强制正在提交事务进行全局排序的手段。

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。一旦发现某个环路，必须放弃一个或者多个事务来解除死锁。边追逐方法是一种非集中式分布式死锁检测方法。

基于事务的应用通常在长生命周期和存储信息的完整性方面有很强的需求，但很多情况下它们对响应时间的要求不高。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执行检查点和在恢复文件中记录日志完成的，当服务器崩溃后被新的进程取代后，检查点和恢复文件可用来进行恢复处理。在进行恢复处理的过程中，事务服务的用户会感受到一些延迟。尽管运行分布式事务的服务器可能出现崩溃，并且处于异步系统中，

598 ~ 599

但由于替换崩溃的服务器的进程可以从持久存储或其他服务器中获取必要的信息，因此，这些服务器仍然能够就事务的结果达成共识。

练习

14.1 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送给所有的参与者。在第二阶段，如果协调者投的是No票，那么参与者只是放弃事务；如果投的是Yes票，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并进行执行。请计算这种协议需要进行几轮消息发送以及消息总数，并将它和集中式的两阶段提交协议进行比较，列出其优缺点。 （第570页）

14.2 三阶段提交协议由以下步骤组成：

第1阶段：和两阶段提交协议相同。

第2阶段：协调者收集所有投票并做出决定。如果决定是No，那么它放弃事务并通知所有投Yes票的参与者；如果决定是Yes，它向所有的参与者发送preCommit请求。每个投Yes票的参与者都等待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3阶段：协调者收集确认消息。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）？ （第573页）

14.3 试解释两阶段提交协议如何保证嵌套事务的顶层事务一旦成功提交，那么所有正确的后代事务都能提交或放弃。 （第574页）

14.4 请给出两个分布式事务的交错执行在每个服务器上都是串行的，但是在全局上不是串行的例子。 （第578页）

14.5 图14-4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需要获得决议的参与者使用。

假设每个参与者都可以对其他参与者调用getDecision函数，那么这样是否能够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符通知给每个参与者？ （第570页） 600

14.6 扩展两阶段加锁的定义来支持分布式事务，解释为什么本地使用严格的两阶段加锁就能够保证分布式事务的串行化？ （第578页，第13章）

14.7 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？ （第570页，第578页）

14.8 一个服务器用时间戳排序进行本地并发控制。在参与分布式事务处理时需要做哪些变化？在什么条件下，两阶段提交协议可以不必用时间戳排序。 （第570页，第579页）

14.9 请考虑分布式乐观并发控制，其中每个服务器在本地顺序（即一次仅有一个事务在验证和更新阶段）使用向后验证，请考虑练习14.4答案所涉及的情况。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？ （第13章，第580页）

14.10 一个集中式全局死锁检测器收集并合并所有的局部等待图。请给出一个例子，解释在死锁检测过程中，当一个死锁环路中的等待事务放弃后，怎样检测到假死锁。 （第583页）

14.11 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死锁。 （第584页）

14.12 一个服务器管理对象$a_1, a_2, \cdots, a_n$，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的定义如下：

T：x = Read(i); Write(j, 44);

U：Write(i, 55); Write(j, 66);

V：Write(k, 77); Write(k, 88);

试描述在使用严格两阶段加锁，并且U在T之前访问a_i和a_j的情况下，这3个事务写日志文件的情况。请描述服务器在崩溃后被替换时，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。 （第590～592页）

14.13 向日志文件追加记录是原子操作，但是追加来自不同事务的记录操作可能是相互交错的。请阐述这种交错对练习14.12的答案影响。 （第590～592页）

601

14.14 练习14.12中的事务T、U和V使用严格两阶段加锁，进行的交错操作如下：

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（而不是等到事务结束后再写入），试描述日志文件中有关T、U和V的日志记录的信息。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？ （第590～592页）

14.15 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。试描述写入日志文件中的有关事务T和U的信息。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。 （第595页）

14.16 练习14.15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。请描述写入日志文件中的这两个事务的信息，解释为什么日志文件中的提交记录必须按事务号排列？在日志文件中的已提交事务的写集合是怎样的？ （第590～592页）

14.17 假设事务的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请求之前崩溃。请描述参与者如何解决这种情况，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596页）

602

第15章 复 制

在分布式系统中，复制是提供高可用性和容错的关键技术。随着受移动计算的发展和与此相关的断链操作频繁发生，高可用性日益引起人们广泛的兴趣。容错在安全是关键要素的系统和其他重要系统中通常作为一项必备的服务被提供。

本章的第一部分将讨论这样一个系统，它的每次操作都作用于复制对象的集合。本章开始描述了一个应用复制的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。

本章接着将描述实现容错的各种方法。首先将介绍线性化和顺序一致性的正确性标准。接下来会介绍并讨论如下两种方法：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）。

本章对提供高可用性服务的三种系统进行了实例研究。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复制对象上的事务（操作的顺序），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。

15.1 简介

本章将研究数据的复制，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，Web服务器的资源在浏览器上的缓存和在Web代理服务器上的缓存都属于复制的一种形式，因为缓存中的数据和服务器中的数据彼此互为副本。第9章介绍的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每天对服务进行访问实现的。

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Web资源进行缓存以避免因为从原始服务器上读取数据而造成延迟。进而，数据有时还在同一个域中的多个原始服务器之间透明地复制。通过将所有服务器IP地址绑定到站点的DNS名字（如www.aWebSite.Org），负载便可以在服务器之间得以共享。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节）。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，例如设计有关，来确保客户接收最新数据（参见2.2.5节）。因此，作为性能增强的一项技术，复制在有效性方面有一些限制。

提高可用性：用户要求服务是高度可用的，也就是说，在合理的响应时间内获得服务的次数所占的比例应该接近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，与高可用性有关的因素有：

- 服务器故障。
- 网络分区和断链操作：通信断链通常是不可预计的，也可能是用户移动性带来的副作用。

对前一个问题，复制是一项在服务器故障的情况下能够自动维护数据的可用性的技术。如果数据被复制到两个或者多个不受对方故障干扰的服务器上，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获取数据。这就是说，通过复制服务器数据，服务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概率或者不可访问概率p，那么在每个服务器上保存的对象的可用性概率就是：

604 1－概率(所有管理器故障或不可用) = $1-p^n$

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12.1节）和断链操作是影响高可用性的第二个因素。移动用户在移动过程中，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，或者可能没有无线上网功能）。为了在这种环境下工作（这被称为断链工作或者断链操作），用户经常将使用率高的数据（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。但是在断链期间，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，他们可能把面谈安排在某个时间段，但这个时间段其实已经被占用了。断链工作仅在用户（或代表用户的应用程序）能够解决这种过期数据、以后能够解决由此导致的所有冲突时才有效。

容错性：高可用性数据不一定是绝对正确的数据。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进行了有冲突的更新操作，这里的冲突是需要解决的。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。这里的正确性关注的是提供给客户的数据是否最新以及客户对数据的操作的结果。正确性有时也考虑服务的响应时间，例如在航空控制系统中，必须在短时间内获得正确数据。

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个服务器崩溃，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故障，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。但是容错性要比这里给出的简单描述复杂。系统必须处理其组件之间的协调来精确地处理任何时间都可能发生的故障，以保证正确性。

*复制透明性*是数据复制的常见需求。也就是说，客户通常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（或对象），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，而不管事实上的操作可能是针对一个以上的物理拷贝进行的。

605 数据复制的另一个常见需求是一致性，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。

我们看到在日记的例子中，断链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（使用数据的不同物理副本）对同一逻辑对象发出请求，但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。我们还要研究一些处理这些问题的标准的解决方案和技术。首先，第15.2节至15.4节将描述基于共享数据的客户调用。第15.2节给出一个管理复制数据的通用体系结构并介绍作为重要工具的组通信。组通信对于实现容错极为有用，它是15.3节的主题。第15.4节阐述各种高可用技术，包括断链操作。15.4节还包括了对gossip体系结构、Bayou和Coda文件系统的实例研究。15.5节将介绍如何在复制数据上进行

事务处理，正如第13章和第14章所解释的，事务处理是由一系列操作，而不是单个的操作组成的。

15.2 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。每一个逻辑对象是由若干称为副本的物理拷贝组成的集合实现的。副本是物理对象，每一个副本存储在某台计算机上，这些副本上的数据和行为在系统操作下遵循某种程度的一致性。给定对象的副本并不一定完全相同，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收了更新的数据，而另一些副本还没有收到更新数据。

本节先给出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情况下，组通信是非常有用的。

15.2.1 系统模型

我们假定一个异步系统中的进程发生故障的唯一原因是崩溃。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发生的情况。我们使用故障检测器来获得可靠、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。模型中的副本由不同的副本管理器（RM）来管理（见图15-1）。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。该模型可以在客户-服务器的环境中应用，此时，一个副本管理器就是一个服务器。我们有时简单地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一个应用，它的作用相当于用户日记的副本管理器。

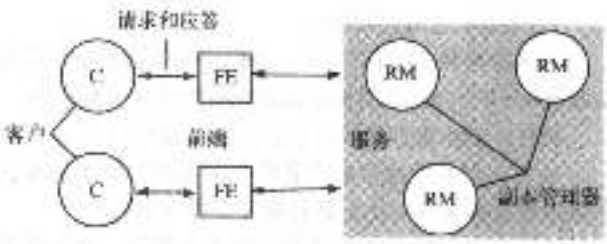


图15-1 管理复制数据的系统体系结构的基本模型

我们应该始终要求一个副本管理器对于它的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实行原子性操作（不可分操作），其执行等价于以某种严格顺序执行操作。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。如果没有这个假定，在独立接收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——它不会在再次产生非确定的影响。这个假设意味着服务器不可能是多线程的。

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。但在一般情况下，不同对象的副本由不同的副本管理器来维护。例如，一个网络上的客户可能经常需要某个对象，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，也可以是动态的。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）。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），但它们可能将停止操作任意长一段时间。我们将在15.4.2节讨论故障问题。

图15-1给出了副本管理的一般模型。副本管理器集合给客户提供某种服务。客户得到一个允许它们访问对象（例如，日记或银行账户）的服务，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。一个操作涉及读对象和更新对象。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一种手段。前端可以在客户进程的地址空间内实现，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五个阶段 [Wiesmann et al. 2000]。对于不同的系统，每一阶段的动作都不一样，下面的两节会做进一步的说明。例如，一个支持断链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。一种可能是前端和某个副本管理器通信，这个管理器再和其他副本管理器通信。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的一致性。如果需要的话，在这个阶段，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。12.4.3节中为组播定义的所有排序类型同样适用于请求处理，这里我们再次简述一下这些排序类型：

- **FIFO序**：如果前端发送请求r，然后发送请求r'，那么任何正确的副本管理器在处理r'之前先处理r。
- **因果序**：如果请求r在请求r'之前发生，那么任何正确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，那么任何正确的副本管理器在处理r'之前先处理r。

大多数应用需要FIFO序。我们将在下面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序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，包括临时请求，这种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
608 中副本管理器可共同决定是放弃还是提交事务。

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。在另外一些系统中，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，那么它将第一个到达的应答返回给用户。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送给客户。

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，在支持断链操作的系统中，尽早将应答反馈给客户（比如用户笔记本电脑的应用）是非常重要的。用户并不希望一直等到笔记本电脑的副本管理器和办公室里的副本管理器能够协调。相比之下，在一个容错系统中，在结果的正确性得到保证以前将不会把应答给客户。

15.2.2 组通信

12.4节讨论了组播通信，因为进程组是组播消息的目标，因此这种通信也称作组通信。对管理复制数据而言，组是非常重要的。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。

12.4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。一个副本管理器也可能由于崩溃而需要从组的操作中删除。组通信的完整实现除了组播通信以外，还需要包含一个组成员关系服务来管理动态组成员关系。

组播和组成员关系管理是紧密联系的。图15-2表示一个开放的组，其中组外的进程向该组发送消息时不需要知道组的成员关系。当组播并发执行时，组通信服务必须管理组成员关系的变化。

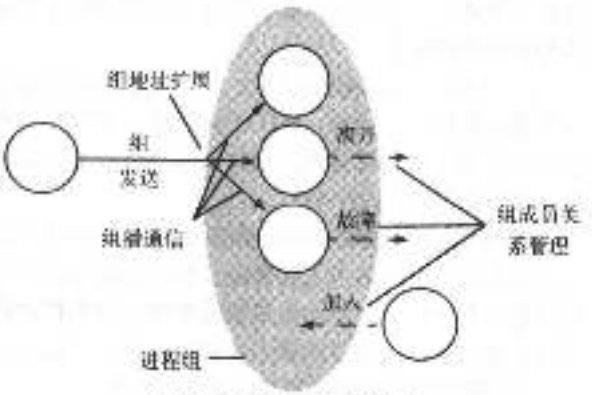


图15-2　进程组提供的服务

组成员关系服务的作用　组成员关系服务有如下四个任务：

1）为组成员关系变更提供接口：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。

2）实现故障检测器：该服务有一个故障检测器（见12.1节）。该服务监测组中各成员，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"。服务利用故障检测器来判断组成员的关系：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。

3）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），该服务将向组成员通知这个变化情况。

4）负责组地址的扩展：当一个进程组播消息时，它提供组标识符，而不是组中的进程列表。为了传递，组成员关系服务将标识符扩展为当前组成员关系。通过控制地址扩展，服务能够协调成员变化时的组播，即它能够一致地决定将给定的消息传递到哪，即使成员关系在传递时会发生变化。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，它执行地址扩展，这样在组播消息时，发送者只需要提供一个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关系的信息，并且组播传递也不会随着成员关系的变化而调整。

如果系统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关系变更通知等。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。当进程加入组或从组中删除时，一个新的组视图就产生了。

609 - 610

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。而且，在一个封闭组中，如果这个进程再次被连接，那么它试图发送的消息都不会发给组成员，这个进程必须重新加入这个组（作为自己的一个"新生"，将获取新的标识符），或放弃它的操作才能改善这种情况。

如果错误地怀疑进程并进而将其从组中删除，那么将降低组的有效性，组不得不负责退出的进程可能已经提供的可靠性和性能。除了要将故障检测器设计得尽可能准确，设计挑战还在于，要保证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需要重点考虑的。诸如路由器等网络组件的断链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网络分区中有一个子组（一个较大的子组），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链操作优点的情景。

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，一个可分区的组成员关系服务就是这样。例如，在一个应用中，用户召开音频或视频会议来讨论某些问题，当发生分区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，各子组又连接上时，它们可以再综合它们的讨论结果。

视图传递 考虑某个程序员的任务是写一个应用，这个应用在一组进程中的每一个进程上运行，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，如果程序员不能在本地就如何响应变更做出决定，而不得不查询其他的组成员的状态才能做出决定，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递给组成员。

对于每个组g，组管理服务将一系列的视图$v_0(g)$, $v_1(g)$, $v_2(g)$, …传递给组的每个进程$p \in g$。例如，一个视图的序列可以是$v_0(g) = (p)$, $v_1(g) = (p, p')$, $v_2(g) = (p)$，即首先p加入一个空组，然后p'加入这个组，接着p'离开这个组。尽管可能会同时发生多个组成员变更，例如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。

如果当组成员关系发生变更时，一个成员将新的成员关系告知给应用（和进程传递组播消息
611 类似），那么我们称这个成员传递了视图。对组播传递，传递视图和接收视图是截然不同的。组成员关系协议将提出的视图放到一个保留队列上，直到所有现有的组成员同意进行传递为止。

如果在事件发生时，p已经传递了视图v(p)但是还没有传递下一个视图v'(g)，我们称一个事件发生在进程p的视图v(p)中。

视图传递有如下一些基本要求：

- **顺序**：如果一个进程p传递视图v(g)，然后传视图v'(g)，那么不存在这样的进程$q \neq p$，它在v (g)之前传递v'(g)。
- **完整性**：如果进程p传递视图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递的视图将不包含其他分区中的进程。

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，上面的第一个要求向程序员提供一致性保证。第二个要求是一个完整性检查。第三个要求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如何，告诉每一个进程它自己在这个组中并没有任何作用。非平凡性条件表明，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，那么它们将被认为是该组的成员。同样的，当发生分区时，组成员关系服务应该最终反映分区。条件并没有说明在有问题的间歇性连接时应如何处理组成员关系服务。

视图同步的组通信 在组播消息传递方面，视图同步的组通信系统除了以上视图传递的要求外，还能做出额外的保证。视图同步的组播通信扩展了第12章讨论的可靠组播语义，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**：正确的进程传递相同序列的视图（从加入组的视图开始），并且在任何给定的视图中传递同样的消息集合。换句话说，如果一个正确的进程在视图v(g)中传递了消息m，那么所有其他传递m的正确的进程都在视图v(g)中传递m。
- **完整性**：如果一个正确的进程p传递消息m，那么它不会再传递m，而且，$p \in group(m)$，并且发送m的进程处于p传递m的视图中。

- **有效性（封闭组）**：正确进程总是传递它们发送的消息。如果系统在向进程q传递消息时发生了故障，那么它将传递一个删除了q的视图给余下的进程。也就是说，设p是一个正确的进程，612
  它在视图v(g)中传递消息m。如果某个进程q∈v(g)没有在视图v(g)中传递m，那么在p传递的下一个视图v'(g)中将有q ∉ v'(g)。

考虑一个具有三个进程p、q、r的组（如图15-3所示）。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都发送新的视图(q, r)，但都不会传递消息m（如图15-3a所示）。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未崩溃进程中的一个。此时q和r都先传递消息m，然后传递视图(q, r)（参见图15-3b）。让q和r先传递视图(q，r)然后传递m是不允许的（参见图15-3c），因为那样的话它们将传递来自已经出现故障的进程的消息。同样，两个进程也不能以相反的次序传递消息和新视图（见图15-3d）。

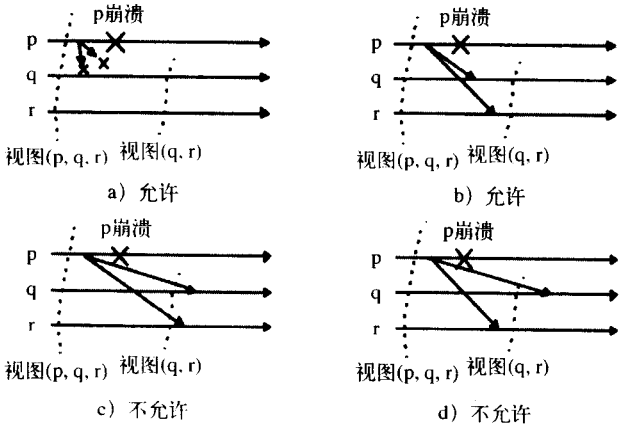


图15-3　组通信中的视图同步

在一个视图同步系统中，新视图的传递在概念上绘制出一条横穿系统的线，每一个被传递的消息都被一致地传递到线的一端或另一端。这样，当程序员传递一个新的视图时，他只需根据局部的消息传递和视图传递事件的次序，就能推断出其他正确进程传递的消息的集合。

用视图同步通信来进行状态转换——工作状态从一个进程组的当前成员转到组的一个新成员——可以说明视图同步通信的用处。例如，如果进程是保存日记状态的副本管理器，然后副本管理器在加入组时，需要获得日记的当前状态。然而当日记状态被捕获的同时，日记同时被更新了。副本管613
理器不应遗漏任何在它获得的状态中没有反映出来的更新消息，也不应重新使用已经反映在状态中的更新信息（除非这些更新是幂等的）。

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递时，现有成员中的一个不同的进程（比如最早的一个）截获了它的状态，将其以一对一方式发送给新的成员并且挂起它自身的执行。所有其他的已有进程也都暂停它们的执行。注意，根据视图同步通信的定义，反映到这个状态中的更新集合要非常明确地应用到其他所有的成员。当新进程收到状态后会集成它，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　我们已经提出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了共识，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12.5.4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，协定条件包括了进程崩溃的情况。对于组播通信的统一协定也是相似的，这已在12.4.2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递完消息后崩溃，所有正确的进程也会在同样的视图中传递这个消息。这个强大的保证有时在容错应用中很有用，因为一个已经传递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12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，具有某种组成员关系服务的进程组开始在其他的系统中开发出来，这包括Horus[van Renesse et al. 1996]和Totem[Moser et al. 1996]以及Transis[Dolev and Malki.1996]。

针对可分区的组成员关系服务，视图同步也有其相应的变种，包括支持分区处理的应用[Babaoglu et al. 1998]和扩展的虚拟同步[Moser et al. 1994]。

最后，Cristian[1991b]讨论了用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（通常是同一类实例）。这些对象处理相同的调用集合，并且每一个都返回响应。客户对象不需要知道复制。它们调用一个本地对象上的操作，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送调用。

Electra[Maffeis 1995]是一个兼容CORBA的系统，它支持对象组。一个Electra组可以衔接到任
614 何兼容CORBA的应用上。Electra最早建立在Horus组通信系统之上，用Horus来管理组的成员关系和组播调用。在“透明模式”下，本地代理返回第一个可用响应给客户对象。在“不透明”模式下，客户对象可以访问所有组成员返回的响应。Electra使用一个扩展的标准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。

Eternal[Moser et al. 1998] 和对象组服务[Guerraoui et al. 1998]也为组对象提供了兼容CORBA的支持。

15.3 容错服务

本节讨论如何通过在副本管理器上复制数据和功能来提供容错服务，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，并且不发生分区。

假设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，银行账户的规约包括如下保证：在银行账户间转账的资金不会消失，并且只有存款和取款会影响某个账户的余额。

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，如果不采取相应措施，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位于计算机A和B上，它们都维护两个银行账户x和y的副本。客户在本地的副本管理器上读取和更新账户，如果本地副本管理器出现故障，就尝试使用另一个管理器。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。

客户1在它的本地副本管理器B上更新x的余额为$1，然后试图更新y的余额为$2，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在它的本地副本管理器A上读取余额，发现y有$2，然后发现x是$0——由于B出现故障，B的银行账户x的更新没有传过来。这种情况如下所示，每个操作被标记上其首次发生时所处的计算机名。另外，操作按发生次序排列：

客户 1:	客户 2:
$setBalance_B(x, 1)$	
$setBalance_A(y, 2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

615

这个执行不符合银行账户行为的规约：客户2如果读到了y的余额为$2，而y的余额是在x余额更新之后更新的话，那它应该读到x的余额为$1。如果银行账户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。最严格的正确系统是*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的某一执行中读和更新操作的序列为$o_{i0}, o_{i1}, o_{i2}, \cdots$其中每一个操作$o_{ij}$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，这种操作的交错序列可能是$o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, o_{12}, \cdots$。我们通常参考一个*虚拟的*客户操作交错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。

一个被复制的共享对象服务，如果对于任何执行，存在某一个由全体客户操作的交错序列，并满足以下两个准则，则该服务被认为是可线性化的：

- 操作的交错执行序列符合对象的（单个）正确副本所遵循的规约。
- 操作的交错执行次序和实际运行中的次序实时一致。

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错执行操作。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错执行序列中时，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可线性化的。即使忽略了操作发生的真实时间，也没有两个客户操作的任何序列满足银行账目规范的序列。为了进行审计，如果一个账户的更新发生在另一个账户更新之后，那么如果已经看到第二个更新，第一个更新也应该保存起来。

注意，线性化能力仅仅考虑个体操作的交错次序，并不打算是事务化的。如果没有应用并发控制，一个可线性化操作可能破坏应用特定的一致性概念。

线性化能力中的实时要求是现实世界所需要的，这是因为它符合我们的观念：客户应该收到最新的数据。不过，定义中的实时性要求会引起线性化能力的一些现实问题，因为我们不能将时钟同步到要求的精确程度。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顺序的实质。顺序一致性保留了线性化能力的第一个准则，但对第二个准则做了修改：

616

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序列，存在某种交错执行，满足下面条件：

- 操作的交错序列符合对象的（单个）正确副本所遵循的规范。
- 操作在交错执行中的次序和在每个客户程序中执行的次序一致。

注意，在上述定义中并没有出现绝对时间，在操作上也没有要求任何*全序*。与次序相关的唯一概念是每个客户上的事件次序——程序的次序。操作的交错执行会以任意方式来重新排列一个客户集合上的操作，只要不违反每个客户的顺序，而且按照对象规范而言，每个操作的结果与重新排列

前的操作结果一致。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一样。

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，但是反之不成立。下面的例子满足顺序一致性，但不是可线性化的：

客户1:	客户2:
$setBalance_A(x, 1)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y, 2)$	

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上对x的更新在客户2读取它时没有到达的话，就会出现这种情况。在该例中，由于$getBalance_A(x) \rightarrow 0$发生在setBalanceB(x, 1)之后，因此线性化的实时准则没有满足。但是下面的交错执行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“线性化能力”）。Herlihy和Wing[1990]将这一思想加以推广，使之涵盖任意共享对象。第18章将研究分布式共享内存，其中会定义并讨论一些更弱的一致性的性质。

617

15.3.1 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15-4），任何时候都有一个主副本管理器和一个或多个次备份副本管理器，它们称为“备份”或“从管理器”。该模型的实质是，前端只和主副本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副本发送到备份副本管理器。如果主副本管理器出现故障，那么某个备份副本管理器将被提升为主副本管理器。

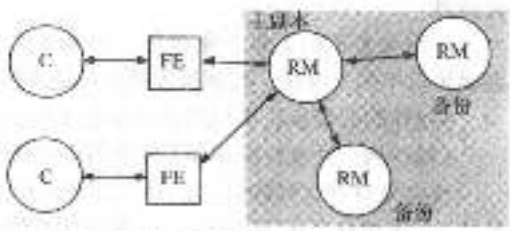


图15-4 用于容错的被动（主备份）模型

当用户需要执行一个操作时，事件的次序如下：

1）请求：前端将请求发送给主副本管理器，请求中包括了一个唯一标识符。

2）协调：主副本管理器按收到请求的次序原子地执行每一个请求。它检查请求的唯一标识符，如果请求已经执行了，那只需再次发送响应。

3）执行：主副本管理器执行请求并存储响应。

4）协定：如果请求是更新操作，那么主副本管理器向每个备份副本管理器发送更新后的状态，响应和唯一标识符，备份副本管理器返回一个确认。

5）响应：主副本管理器将响应前端，前端再将响应发送给客户。

在主副本管理器正确运行的情况下，由于主副本管理器在共享对象上将所有操作顺序化，因此该系统显然是具有线性化能力的。当主副本管理器出故障时，如果某个备份变为新的主副本管理器并且新的系统配置从故障点正确接管的话，那么系统仍具有线性化能力：

- 主副本管理器被唯一的备份副本管理器代替（如果两个客户使用两个备份，那么系统将不会正确执行）；并且
- 当接管主副本管理器时，剩余的备份副本管理器在哪些操作已被执行上达成一致。

618

如果副本管理器（主副本管理器和备份副本管理器）组织为一个组，并且主副本管理器使用视图同步组通信发送更新到备份，那么上述两个要求都能达到。上述两个要求中的第一个要求很容易满足。当主副本管理器崩溃时，通信系统最终传递一个新的视图给现有的备份，该视图不包含原来

的主副本管理器。替代主副本管理器的备份可以用针对该视图的任何函数选择，比如可选择视图中的第一个成员作为替代。作为替代的那个备份副本管理器使用名字服务将自己登记为主副本管理器，客户在怀疑主副本管理器出故障（或在第一次请求服务）时可以通过名字服务进行查询。

通过使用视图同步的排序性质，以及通过存储标识符来检测重复的请求，可以满足第二个要求。视图同步的语义保证了在传递新视图以前，或者所有备份或者没有备份传递任何更新。新的主副本管理器和存活下来的备份副本管理器能就客户的更新是否已执行达成一致。

现在来考虑前端没有收到响应的情况。前端将请求重传到作为主副本管理器接管的备份副本管理器。主副本管理器在操作执行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，那么它们可能已经执行了那个请求。如果它在协定阶段之后崩溃，那么它们肯定不会再处理这些请求。但新的主副本管理器并不需要知道原来的管理器在崩溃时处在什么阶段。当它收到一个请求，它从第二阶段开始执行。通过视图同步，主副本管理器并不需要查询备份副本管理器，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-备份模型。由于主副本管理器是将操作的更新状态发送给备份副本管理器而不是发送操作自身的规范，所以备份只是被动地记录这些由主副本管理器的行为独立决定的状态。

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，当当前的主副本管理器不响应时，前端需要查找新的主副本管理器。

被动复制的缺点是开销相对较大。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递新视图。这会导致更多的延时。

在该模型的一个变种中，客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，但仍能提供具有顺序一致性的服务。

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。例如存储某些用于系统管理的记录。复制的数据在一个主服务器上被更新并从主副本服务器以一对一的方式（而不是组）传播到备份服务器。客户通过和主或备份服务器通信以获得信息。但在NIS中，客户可能不请求更新，更新仅针对主服务器的文件进行。 619

15.3.2 主动复制

在用于容错的主动复制模型中（参见图15-5），副本管理器是一个状态机，其中每个副本管理器充当同等的角色并被组织成一个组。前端将它们的消息组播到副本管理器组，并且所有的副本管理器按独立但相同的方式来处理请求并给出应答。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。

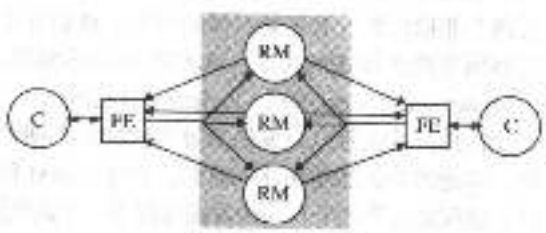


图15-5 主动复制

对于主动复制，当客户请求一个操作时，事件顺序如下：

1）请求：前端给请求加上一个唯一标识符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。前端在收到响应之前不会

发送新的请求。

2）协调：组通信系统以同样的次序（全序）将请求传递到每个正确的副本管理器。

3）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。请求的响应包括客户的唯一请求标识符。

4）协定：由于组播的传递语义，实际上不需要该阶段。

5）响应：每个副本管理器将它的响应送往前端，前端收到应答的数量取决于故障模型的假设
620 和组播算法。例如，如果目标是只容忍崩溃故障并且组播满足统一协定和排序性质，那么前端可将第一个响应返回给客户，并丢弃其他响应（通过使用唯一标识符，它能将这些响应与其他的响应中区分开来）。

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。组播的可靠性保证每一个正确的副本管理器处理同样的请求集合，全序保证以同样的顺序处理它们。因为它们是状态机，所以在每一个请求后，它们都会到达同一个状态。每个前端的请求以FIFO的顺序进行处理（因为前端在发出下一个请求前会等待响应），这和“程序的顺序”一样，从而保证了顺序一致性。

如果客户在等待它们请求的响应时并不和其他客户通信，那么它们的请求按发生在先顺序处理。如果客户是多线程的，并且在等待响应和其他的客户通信，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。

由于副本管理器处理请求的全序并不一定和客户发生这些请求的实时次序相同，因此主动复制系统并不具有线性化能力。Schneider[1990]阐述了在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，所以不能保证线性化，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案。就像第12章指出的，解决了可靠性和全序的组播等价于解决了共识。解决共识又要求系统是同步的，或者使用了一种技术（如应用故障检测器）来绕过Fischer等人[1985]获得的理论上的不可能性。

某些共识的解决方案，像Canetti和Rabin的方法[1993]，可以在有拜占庭故障的情况下工作。如果某个解决方案能够提供全序和可靠的组播，那么主动复制系统就能够屏蔽至多f个拜占庭故障，只要服务包含至少$2f + 1$个副本管理器。每个前端一直等待到它收集到$f + 1$个相同的响应才将响应返回给用户。它丢弃对同一请求的其他响应。为了确定哪个响应和哪个请求相联系（假定有拜占庭故障），我们要求副本管理器对它们的响应进行数字签名。

可以将我们描述的系统进行适当放宽。首先我们已经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$o_1;o_2$的顺序执行和以相反的顺序$o_2; o_1$的执行效果是一样的。例如，（来自不同客户的）任何两个只读操作是可交换的；任何两个非读操作，若更新不同的对象，也是可交换的。主动复制系统需要使用交换性的知识来避
621 免将所有请求排序的代价。我们在第12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某个副本管理器。这样，系统丧失了与组播请求有关的容错，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。

15.4 高可用服务的实例研究：gossip体系结构、Bayou和Coda

本节考虑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断链时能继续工作，那么他们会愿意以后接受数据副本（比如日记）间暂时的不一致，并在以后加以修正。

在15.3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，所有正确的副本管理器都收到更新，并在将控制传递回客户以前达成一致。这种方式并不适合高可用操作。反之，系统应该通过使用最小的与客户连接的副本管理器集合，提供一个可接受级别的服务。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，这使得共享数据的可用性提高。

下面研究三个提供高可用服务的系统的设计：gossip体系结构，Bayou和Coda。

15.4.1 gossip体系结构

Ladin等[1992]开发了gossip体系结构，用它作为框架实现了高可用性服务。具体实现方式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：副本管理器定期通过gossip消息来传递客户的更新（见图15-6）。这种体系结构是基于Fisher和Michael[1982]、Wuu和Bernstein[1984]早期在数据库方面的工作。它可以用来创建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：查询和更新。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。尽管某个副本管理器可能暂时不能和其他副本管理器通信，系统仍然做出以下两个保证：

- 随着时间推移，每个用户总能获得一致服务：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和不同的副本管理器通信，因此从原理上能与这个副本管理器通信，该副本管理器比以前使用的“稍微落后一些”。
- 副本之间松弛的一致性：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，排序保证使副本充分满足应用的要求。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，但两个客户仍会观察到不同的副本，客户也可能观察到过时的数据。

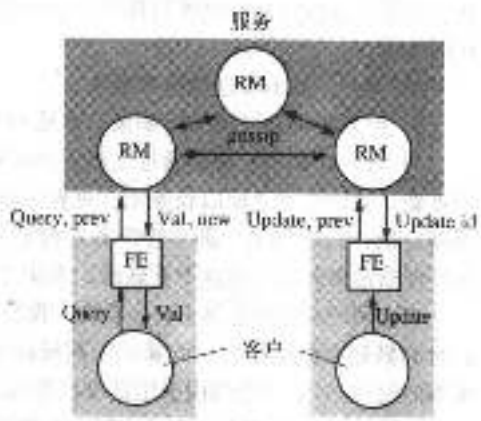


图15-6 gossip服务中的查询和更新操作

为了支持松弛的一致性，gossip体系结构支持更新的因果序，其定义见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的执行次序可能不同，因此除了提供强制序外，还需要提供即时序。

具体使用哪种排序由应用的设计者决定，它反映了在一致性和操作代价之间的一种取舍。因果更新的代价远远低于其他排序的更新，只要可能，一般都使用它。注意，任意一个副本管理器都能满足的查询相对于其他操作永远以因果序执行。

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，并和一个本地副本管理器通信。客户程序将用户的投稿发送给本地副本管理器，这个副本管理器在gossip消息中将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以天计的话，一般影响不大。因果序可用于投稿项。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，一个主题为“回复：桔子”的投稿总是比它引用的标题为“桔子”的消息晚发送。强制序能够用来在电子公告板中加入一个新的订阅者，这样用户加入记录的顺序是无二义的。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，那个用户就不会从一个响应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来处理客户操作，并将其转为gossip操作。通常，客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，因此前端会把一个读取和修改都有的操作转换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：

1）*请求*：前端一次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现故障或不可达时，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，前端也将尝试使用其他的副本管理器。因此，前端和客户可能阻塞在一个查询操作上。另一方面，在默认情况下，更新操作一旦被传递给前端，就可立即返回给客户；前端再在后台传播这个操作。为了提高可靠性，客户可以被阻塞到更新已经传给了$f+1$个副本管理器后才继续执行，这样就算f个副本管理器出现故障，操作也将传递到任何位置。

2）*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就立即应答。

3）*协调*：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为止。这涉及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。

4）*执行*：副本管理器执行请求。

5）*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。

6）*协定*：副本管理器通过交换gossip消息进行相互更新，这些gossip消息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
624 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这个更新时，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维护的时间戳和数据结构，在此基础上解释副本管理器如何处理查询和更新。维持因果更新的向量时间戳的处理和12.4.3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向量时间戳，它用来反映前端访问的（因而也是客户访问的）最新数据值。在该时间戳中（即图15-6中的prev），每个副本管理器有一条对应的记录。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向量时间戳（图中的new），因为完成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唯一的向量时间戳（图中的updateid）。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11.4节的向量时间戳合并的定义）。

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，前端可以顺便将它们的向量时间戳发送给其他的客户。接收者将它们和自己的向量时间戳合并，这样可正确地保证因果次序。这种情况如图15-7所示。

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（参见图15-8）：

1）*值*：这是由副本管理器维护的应用状态的值。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。

2）*值的时间戳*：这是代表更新的向量时间戳（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。

3）更新日志：所有的更新操作只要被收到了，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更新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。

4）副本时间戳：这个向量时间戳代表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更新。一般情况下，它和值的时间戳不同，因为并不是所有在日志中的更新都是稳定的。

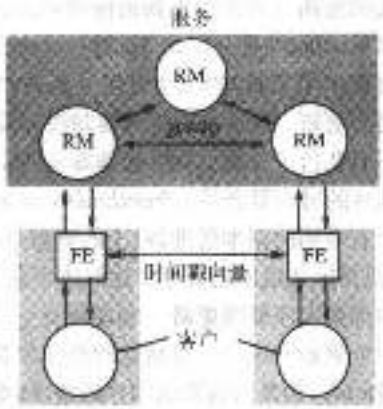


图15-7 当客户直接通信时，前端传播它们的时间戳

5）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，它包含已经执行的更新的唯一标识符。这个唯一标识符由前端提供。副本管理器将更新加入日志前，先检查这个表。

6）时间戳表：这个表为每个副本管理器包含一个向量时间戳，该时间戳来自gossip消息。副本管理器使用此表来确定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向量时间戳中的第j个元素对应于通过i从前端收到的更新的数量；第j个组件（$i \neq j$）等于通过j收到的并传播给i的更新的数量。例如，在有三个副本管理器的gossip系统中，管理器0上的一个值时间戳（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个更新，从管理器2接收到5个更新。下面将详细地描述如何使用时间戳来保证次序。

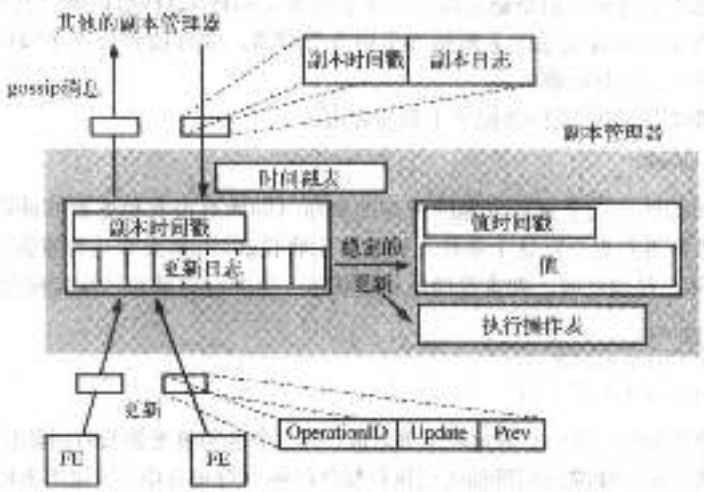


图15-8 gossip的副本管理器及其主要状态组件

查询操作 最简单的操作是查询操作。一个查询请求q包含操作的描述和一个由前端发送的时间戳$q.prev$，后者反映了前端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$valueTS$是副本的值的时间戳，且下面条件满足，那么q能够应用到副本的值上：

$q.pre \leqslant valutTS$

副本管理器将q放到将执行的操作表中（即一个保留队列），直到这个条件满足为止。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.pre*是（2, 4, 6），可以看出，只有一个更新丢失了，即从丢失了来自副本管理器2的一个更新（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这个更新）。

一旦执行了查询，副本管理器返回*valueTS*给前端，作为在图15-6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，查询执行前，前端没有看到的副本管理器1上的更新（*q.pre*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）。

625 ~ 627

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包含一个更新的规约（包括它的类型和参数）*u.op*、前端的时间戳*u.prev*和一个前端产生的唯一的标识符u.id。如果前端发送同样的请求u给若干副本管理器，那么每次在u中使用相同的标识符——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。

当副本管理器*i*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符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个数。然后，副本管理器给更新请求u分配一个唯一的向量时间戳（下面将给出这个向量的来源），并且将一个更新记录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唯一时间戳，那么更新记录按如下元组构建并保存在日志中：

$logRecord := \langle i, ts, u.op, u.prev, u.id \rangle$

副本管理器*i*将*u.prev*的第*i*个元素替换为它的副本时间戳的第*i*个元素（这个元素刚刚加一），完成从*u.prev*中生成*ts*时间戳的工作，这样能使*ts*是唯一的，从而保证所有的系统组件能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，因为正是要应用这些从前端送来的值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多个副本管理器，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下面的查询：

$u.prev \leqslant valueTS$

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，稳定条件已经满足时，副本管理器将更新值、值的时间戳和已执行操作表：

$value := apply(value, r.u.op)$
$valueTS := merge(valueTS, r.ts)$
$executed := executed \cup \{r.u.id\}$

在这三个语句中，第一个语句表示更新值，第二个语句将更新的时间戳和那个值的时间戳合并，第三个语句将更新操作的标识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。

强制的和即时的更新操作 强制更新和立即更新需要特殊处理。强制更新是全序加因果序。保证更新的强制次序的基本方法是在相联系的时间戳后加入一个唯一的序号，并以这个序号的次序来处理它们。像第12章所解释的，产生序号的一般方法是使用一个顺序者进程。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

628

器作为顺序者，当主副本管理器出故障时，可以选择另一个副本管理器能替代成为顺序者。这就要求对于大多数的副本管理器（包括主副本管理器）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未出现故障，就能从存活的副本管理器中选出新的主副本管理器，从而实现这个排序决定。

相对于强制更新，即时更新是通过使用主副本管理器来对更新序列进行排序。主副本管理器也决定了哪个因果更新被认为在一个即时更新之前。它通过与其他副本管理器的通信和同步来完成这工作，进一步的细节见Ladin等的文章[1992]。

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副本管理器还没有收到哪些更新（由于副本管理器可能收到了很多的更新，因此这只是个估计）。

源副本管理器发送的一个gossip消息m包含两项：日志m.log和副本时间戳*m.ts*（见图15-8）。收到gossip消息的副本管理器有下面三项主要任务：

- 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）。
- 当知道更新已执行并且已经没有被重复执行的危险时，删除日志和已执行操作表中的记录。从日志和已执行操作表中删除冗余条目非常重要，否则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简单的。设*replicasTS*表示接收者的副本时间戳。m.log中的记录r被加到接收者的日志中，除非$r.ts \leqslant replicsTS$——此时，它已存在于日志中或已经被执行且被丢弃了。

副本管理器将收到的消息中的时间戳和它自己的复制时间戳*replicaTS*合并，以便与日志的增加相一致：

$$replicaTS := merge(replicaTS, m.ts)$$

当新的更新记录被并入日志时，副本管理器将确定日志中所有已稳定的更新集合S。这些更新可以执行，但必须仔细考虑它们执行的次序，以维持发生在先关系。根据向量时间戳间的偏序“$\leqslant$”，副本管理器对集合中的更新进行排序，然后它以这种次序来执行更新，即当且仅当没有$s \in S$ 629
满足$s.prev < r.prev$时，才有$r \in S$。

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*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，该副本管理器现在能够丢弃日志中的记录r。也就是说，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：

$$tableTS[i][c] \geqslant r.ts[c]$$

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，前端会发出对更新的应答的确认，所以副本管理器知道前端何时会停止发送更新。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接收gossip消息的其他的副本管理器。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三个因素：

- 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与之交换gossip的策略。

第一个因素不由系统控制，尽管用户可以在一定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要经过很长的时间才交换一次，比如一天，那么会如何呢？如果只使用因果更新，那么很可能，每一个站点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。然后在深夜，所有的讨论将被合并。但是当要考虑其他人的讨论时，针对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。

人们还提出一些选择合作者的策略。Golding和Long[1993]在他们的著作“基于时间戳的反熵协议”（timestamped antientropy protocol）中使用了一个gossip风格的更新传播机制，考虑了随机、
630 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更合适的合作者。例如，邻近的合作者优于远距离的合作者。Golding和 Long[1993]发现，这种策略在模拟环境中工作得非常好。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位于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。一种可能性是安排为网格（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。另一种方案是将副本管理器组织为一个环，每个管理器只将gossip消息传给它的邻居（比如，以顺时针方向），这样任何一个副本管理器的更新将遍历整个环。还有其他一些可能的拓扑结构，如树。

这些合作者选择策略必须权衡通信量和高传播延时，以及某个故障对其他副本管理器产生影响的可能性。实际中的选择和这些因素密切相关。例如，环拓扑将产生较小的通信量，但可能造成高延时，因为gossip消息通常要遍历若干个副本管理器。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，而需要重新配置。比较而言，随机选择策略不易受故障影响，但它的更新传播时间可能会变化。

有关gossip体系结构的讨论 gossip体系结构的目标是保证服务的高可用性。在这种情况下，即使客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，该客户就能继续获得服务。但是这种可用性的代价是必须遵守松弛的一致性。对银行账户这样的对象，顺序一致性是必须的，gossip体系结构不会比15.3节研究的容错系统表现得更好，gossip系统仅在一个主分区中提供服务。

更新传播的惰性方法使一个基于gossip的系统不适合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。在这种情况下更合适使用一个基于组播的系统。

gossip系统的可伸缩性是另一个问题。随着副本管理器数量的增长，需要传递的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，（在前端和副本管理器之间）通常需要两个消息。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$2 + (R-1) / G$。式中的第一项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副本管理器的gossip消息的更新消息。提高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要等待更多的更新到达。

631 为了增强基于gossip的服务的可伸缩性，一个方法是将大多副本管理器设置为只读的。换言之，这些副本管理器只通过gossip消息进行更新，并不直接从前端接收更新。当更新/查询率很小时，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管理器完成。因

为只读副本管理器没有gossip消息传播，所以gossip流量会降低。同时向量时间戳只需要包含那些更新副本的条目。

15.4.2 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，但Bayou系统提供的一致性保证弱于顺序一致性。与那些系统类似，Bayou的副本管理器通过成对地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采用了一个非常不同的方法，它能够进行领域特定的冲突检测和冲突解决。

考虑一个离线工作时需要更新日记的用户。如果需要严格的一致性，在gossip体系结构中，更新必须用强制的（全序）操作执行。但那样的话，只有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。预定不冲突约会的用户和无意中在一个时间段进行两次预约的用户会被一视同仁。

相比之下，在Bayou中，火车上的用户和办公室中的用户都可以进行他们希望的任何更新。所有更新将被应用，并且记录到它们到达的副本管理器中。当任何两个副本管理器接收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特定领域准则。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会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被取消或改变以解决冲突的过程被称为**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，是一个“存储过程”调用，它影响着每个副本管理器中的一些对象，但它遵循ACID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应用这些更新，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。 632

提交的更新和临时更新　当更新首次应用于数据库时，它们被标记为**临时的**。Bayou最终将临时的更新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，可以通过将某些副本管理器设为主副本管理器来获得提交的次序。通常，这决定提交的次序为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。

在任何时刻，数据库副本的状态来自一个（可能空的）提交的更新序列，后跟着一个（可能空的）临时的更新序列。如果第二个更新到达，或如果某个临时更新已经被执行变为下一个提交的更新，那么必须对更新进行重排序。在图15-9中，t_i已经变为提交的。c_N后的所有更新都必须撤销。然后，t_i在c_N后执行，并且$t_0 \sim t_{i-1}$和t_{i+1}等在t_i后被重新执行。

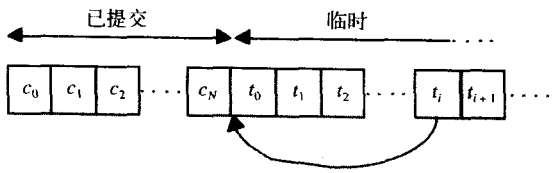


图15-9　Bayou中的提交更新和临时更新

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。考虑到这种可

能性，除了操作规约（包括操作类型和参数）外，每一个Bayou更新还包含一个依赖检查和一个合并过程。所有这些更新的组件都是领域特定的。

一个副本管理器在执行操作前调用依赖检查过程。该过程用来检查是否一个更新执行时会产生冲突，为检查冲突，它可能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。最简单的情况是，依赖检查可以检查写-写冲突，即是否另外一个客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。

如果依赖检查发现了一个冲突，那么Bayou将调用操作的合并过程。该过程会改变将要执行的
633 操作以获得相似效果，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，或者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，然后留下重要的预约。合并过程可能无法找到一个操作的合适替代，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。

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而言是不透明的。它利用应用语义的知识提高数据的可用性，同时维持一个复制状态，我们称之为*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工作的复杂度，他必须提供依赖检查和合并过程。当需要检查并解决大量可能的冲突时，生成这两者非常复杂。第二个不足是增加了用户工作的复杂度。用户不仅要处理所读的临时数据，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。应给用户一个清晰的指示，说明哪些数据是临时的，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用于支持CSCW（计算机支持的协同工作）的计算机系统中，该系统中地理上分离的用户可能发生更新冲突 [Kindberg et al. 1996, Sun and Eills 1998]。该方法的实际应用限于冲突较少的应用、基本数据语义较简单的应用以及用户可以处理临时信息的应用。

15.4.3 Coda文件系统

Coda文件系统的前身是AFS系统（参见第8.4节），其目标是解决一些AFS不能解决的需求，特别是除断链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（只限于只读卷），使它的规模受限，不适用于访问大规模共享的文件，如电子公告板系统和其他系统范围的数据库。

另外，AFS提供的服务仍然有提升可用性的空间。AFS用户所经历的最常见问题是服务器和网络组件的故障（或调度中断）。在CMU的系统规模下，每天会发生一些服务故障，这些故障在几分钟到数小时内给用户造成了极大的不便。

634 最后，AFS没有迎合计算机使用的一种新趋势——便携式计算机的移动使用。这种趋势导致了下列需求：在计算机断链时，用户不必借助手工方式管理文件的位置，而是能继续自己的工作。

Coda就是为满足这三个需求而开发的，这三个需求流称为*稳定的数据可用性*。目标是提供一个共享文件存储，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的容错性。另外，Coda扩展了AFS使用的在客户计算机上缓存文件副本的机制，使客户在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15.4.2节），它也采用了乐观策略。也就

是说，它允许客户在有网络分区的情况下更新数据，只要冲突发生的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称为Venus进程，在文件服务器上运行的进程称为Vice进程。Vice进程就是我们所说的副本管理器，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，因此尽管它们和Vice进程类型有所不同，它们仍是副本管理器。

持有一个文件卷副本的服务器集合称为*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，该子集被称为*可用的卷存储组*（AVSG）。由于网络或服务故障使服务器变得可访问或不可访问，AVSG的成员关系也在变化。

正常情况下，Coda文件访问过程和AFS的文件访问过程相似。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，通过一个*回调承诺机制*，客户被告知文件的变化，而Coda依靠一个附加机制对每个副本管理器进行更新分布。当文件关闭时，修改过的拷贝并行广播到AVSG中的所有服务器。

在Coda中，断链操作被认为发生于AVSG为空时。这可能是由于网络或服务器故障造成的，也可能是客户计算机（比如一台笔记本电脑）有意离线的结果。断链操作的有效性依赖于客户计算机缓存中是否有用户继续工作所需的所有文件。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，用它来记录网络连接时文件使用的历史表，并以这个表为基础预测离线时要使用的文件。

635

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拷贝更可靠。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缓存的文件拷贝，但这样的系统不大可能提供令人满意的服务质量。Coda服务器的目标是提供必要的服务质量。客户计算机缓存中的文件拷贝被认为是有效的，只要它们的当前数据能定期与服务器上的拷贝进行验证。在断链操作的情况下，重新验证在断链操作停止并且将缓存文件和服务器上的文件重新整合时发生。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——在网络分区和断链操作期间，仍然可以进行文件修改。它依靠每个版本的文件上附加的Coda*版本向量*（CVV）。CVV是一个向量时间戳，其中每一个元素对应着在相关VSG中的每个服务器。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每个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应的CVV（11.4节给出了对于向量时间戳v_1和v_2而言$v_1 \geqslant v_2$的定义），那么不会发生冲突。旧的副本（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。

如果不是这种情况，对于两个CVV，即当$v_1 \geqslant v_2$和$v_2 \geqslant v_1$均不成立时，表示存在一个冲突：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不会自动解决冲突。文件被标记为“不可操作”并且向文件所有者告知有冲突。

当一个修改的文件关闭后，由客户的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，如果这个CVV比当前它持有的CVV大，则存储文件新内容并返回一个肯定的确认。然后Venus进程计算一个新的CVV：对更新消息进行肯定应答的服务器，增加它的修改记数，并且发布新的CVV给AVSG中的成员。

由于消息仅仅发送给AVSG的成员而不是VSG的成员，因此不在当前AVSG中的服务器收不到新的CVV。因此，对本地服务器，CVV经常包含一个准确的修改记数，但对于非本地的记数一般

是更小些的下界，因为仅当服务器收到一个更新消息时它们才更新。

下面的例子说明了在三个站点上使用CVV来管理文件副本的更新。可以在[Satyanarayanan et al. 1990]中找到使用CVV管理更新的更多细节。CVV基于Locus系统使用的复制技术[Popek and Walker 1985]。

例子：考虑对卷中的文件F的一个修改序列，它在三个服务器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）起初，F的CVV在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进程将一个更新消息广播给它的AVSG，即$\{S_1, S_2\}$。最后产生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上出现任何改变。

3）同时，C_2运行两个进程，每一个进程都打开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，产生了F的一个新的版本和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，C_2通过某个例程检查以前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稍后描述），发现S_1和S_2现在可达了。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]，而S_3有[1，1，3]。这是一个冲突，需要手工干预以使F能以信息丢失最少的方式进行更新。

另一方面，考虑一个相似但是更简单的情况，即事件顺序相同，但删去了第3条，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S_1和S_2的CVV([2，2，1])控制了S_3。S_1或S_2的文件的版本应该替代S_3上的文件版本。

在正常的操作中，Coda的行为和AFS相似。一次缓存访问未命中，对于用户而言是透明的，并且仅仅是性能上的问题。在多个服务器上复制某些或全部文件卷，所获得的好处有：

- 对于至少可以访问一个副本的客户，可访问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。这个请求作用于所有具有副本的服务器的一个复制卷上。

在断链操作（客户不能访问卷中的任何服务器）中，一次缓存访问未命中会阻止进一步的操作，计算被挂起直到重新连接上或用户放弃了进程。因此，在断链操作开始前加载缓存非常重要，这样可以避免缓存访问未命中。

636 ~ 637

简而言之，和AFS相比，Coda通过文件在多个服务器上复制和客户能在缓存范围之外操作，改善了可用性。两种方法都取决于乐观策略的使用，从而在有网络分区的情况下检测出更新冲突。这两种机制既是相互补充的，又是相互独立的。例如，一个用户可以利用断链操作的好处，即使需要的文件卷被存储在单个服务器上。

更新语义 当客户打开一个文件时，Coda提供的传播保证比AFS要弱，这反映了乐观更新策略的特点。在AFS的传播保证中的单个服务器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）。

通俗地说，在Coda中一个成功的“open”提供的保证如下：它从当前的AVSG中提供F的最近拷贝，并且如果没有服务器是可访问的，并且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；如果没有服务器可用，这个文件便被加上标记以便在第一时间传播出去。

考虑到丢失回调的影响，通过扩展在AFS中应用的标记，可以产生这些保证的更精确的定义。

除了最后一个定义外，每个定义都有两种情况：首先，$\overline{s} \neq \emptyset$，代表所有AVSG不为空的情景；然后处理断链操作：

在一个成功的 open之后：　($\overline{s} \neq \emptyset$ and (latest(F, $\overline{s}$, 0)
or (latest(F, $\overline{s}$, T) and lostCallback ($\overline{s}$, T) and
inCache(F))))
or ($\overline{s} = \emptyset$ and inCache(F))

在一个失败的open之后：　($\overline{s} \neq \emptyset$ and conflict (F, $\overline{s}$))
or ($\overline{s} = \emptyset$ and ¬ inCache(F))

在一个成功的close之后：　($\overline{s} \neq \emptyset$ and updated (F, $\overline{s}$))
or ($\overline{s} = \emptyset$)

在一个失败的 close之后:　$\overline{s} \neq \emptyset$ and conflict(F, $\overline{s}$)

上述模型假定是一个同步系统。T是客户不知道在其他地方对它缓存中的文件做了一次更新的最长时间，latest(F, $\overline{s}$, T)指客户C的文件F的当前值是最近T秒$\overline{s}$的所有服务器中的最新值，与该时刻F的拷贝没有冲突。lostCallback ($\overline{s}$, T)指在最近T秒由$\overline{s}$的一些成员发送了一个回调，但在C端没有收到。conflict (F, $\overline{s}$)指当前$\overline{s}$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，也可以基于性能准则（比如物理上接近或根据服务器负荷）进行选择。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 638
首选站点，文件内容将被重新获取，并且告知AVSG成员一些成员有过时的副本。当完成读取时，在那个首选服务器上建立一个回调承诺。

当客户的一个文件在修改后关闭时，将使用一个组播远程过程调用协议将它的内容和属性并行传递到AVSG的所有成员。这将使一个文件在每个复制站点都有当前版本的可能性最大。它并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。正常情况下，通过让客户负责传播文件的修改到各个复制场地，可以将服务器负载减到最小（只有在open操作发现一个过时的副本时，才需要服务器帮忙）。

因为在所有的AVSG成员中维持回调状态是非常昂贵的，所以回调承诺仅维持在首选服务器上。但这样做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。如果出现这种情况，第二个客户的一个更新将不会导致对第一个客户的回调。下一小节将讨论这个问题的解决方法。

缓存一致性　Coda的传播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（由于一个先前不可访问的服务器变得可访问）。
- 收缩一个AVSG（由于一个服务器变得不可访问）。
- 回调事件丢失。

为了实现这个目标，Venus每隔T秒发送一个探测消息给文件的VSG中的所有服务器，表示文件已经在它的缓存中。Venus只能从可访问的服务器那里收到应答。如果Venus从一个先前不可访问的服务器收到应答，那么它会扩大对应的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拷贝可能不再是新的AVSG中的最新可用版本了。

如果Venus不能从一个先前可访问的服务器处接收到应答，则它收缩对应的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引起，在这种情况下，那个服务器的所有回调

承诺必须丢弃。如果一个响应显示已发送了一个回调消息但没有被收到，那么相应文件上的回调承诺将被丢弃。

剩下的问题是，一个服务器没有收到更新，因为该服务器不在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*（卷CVV）响应每个探测消息。卷版本向量包含一个卷中所有文件的 CVV的摘要。如果Venus检测到卷CVV间的任何不匹配，则说明一些AVSG成员肯定有一些过时的文件版本。尽管过时的文件可能不是在本地缓存的，但由于Venus使用悲观的假设，因此会丢弃所有它持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
639 更新AVSG并检查某一文件卷中的所有文件的回调。这（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使Coda在大量服务器和广域网方面具有可伸缩性的障碍。

断链操作 如果出现短暂的断链，诸如由于不可预料的服务干扰而导致的离线，Venus采用最近最少使用的缓存替代策略，避免断链的文件卷上的缓存不命中。但除非采取另外的策略，否则，一个客户在断链模式下长期工作时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应该努力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，能够容纳所有的高层对象的话，那么用户可一直访问它们。由于要精确地知道某种次序的用户动作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分组；Venus 记录由访问序列生成的文件引用并且为它们标上一个给定的优先级。

在断链操作结束时，开始*重新整合过程*。对于每个在断链操作期间进行了修改、创建或删除的缓存文件或目录来说，Venus执行一系列更新操作以使得AVSG副本和缓存拷贝相同。重新整合从每个缓冲文件卷的根起自顶向下进行。

在重新整合期间，由于其他客户更新了AVSG副本，因此可能会检测到冲突。一旦发生了这样的情况，缓存的拷贝被存储在服务器上的一个临时位置，并且通知发起重新整合的用户。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拷贝的优先级。临时拷贝存储在一个*合作卷*中，它和服务器上每一个卷相关。合作卷很像传统UNIX系统中的lost+found目录。合作卷仅镜像部分，用于存放临时数据的文件目录结构。它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个到50个）的基准负载，比较了Coda和AFS的性能。

如果没有复制，AFS和Coda的性能没有太大的差别。若采用复制三次的策略，Coda在5个典型用户负载的基准下，完成负载的时间只超过无复制的AFS 5%，但是，同样是三次复制，在50个典型用户的负载基准下，Coda完成负载的时间增加了70%，对无复制的AFS，完成负载的时间只增加了16%。这个差别部分归因于与复制相关的开销，实现的不同也是造成性能差异的原因。

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，不仅仅是因为一个管理文件，另一个管理数据库）。我们也描述了Coda如何使用
640 CVV检查冲突，但不用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。之所以说是“潜在”的写-写冲突，是因为在应用语义的层次上来说，并不存在实际的冲突：客户可能无冲突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所用的语义无关的冲突检测和手工解决的方法在许多情况下是可行的，尤其在需要人为判断的应用或者是没有数据语义知识的系统中。

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：目录发生的变化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。

15.5 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一次请求一个单独的操作。第13章和第14章解释了事务是一个或多个操作的序列，并具有ACID性质。对15.4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的对象的事务一样。在无复制的系统中，事务以某种次序执行一次。这是通过客户事务的串行等价交错执行来保证的。作用于复制对象的事务应该和它们在某一对象集上执行具有一样的效果。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它自己的对象提供并发控制和恢复。本节假定为并发控制应用两阶段加锁。

一个副本管理器出现故障，不能再提供服务，但是同一个副本管理器集合中的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。当副本管理器从故障中恢复后，考虑到在它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的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；是否某个客户相关的 641
副本管理器能够推迟转发请求，直到事务提交；以及如何实现两阶段提交协议。

单拷贝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，本节提出了三种复制方案。在出现网络分区，副本管理器集合被分为子组时，这三种方案均可正确工作。

- *带验证的可用拷贝*：在每一个分区中应用可用拷贝复制，当修复分区后，通过一个验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），以便在出现分区时能够继续提供服务。当分区修复后（并且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。

15.5.1 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[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，我们假定前端发送客户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，前端可以和任何一个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。

收到针对特定对象执行操作请求的副本管理器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，read请求可以由单一的副本管理器来执行，而write请求必须由组
642 中所有副本管理器来执行。如图15-10所示（不同对象可以有不同数目的副本）。法定数共识方案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read only操作的副本管理器的数目。

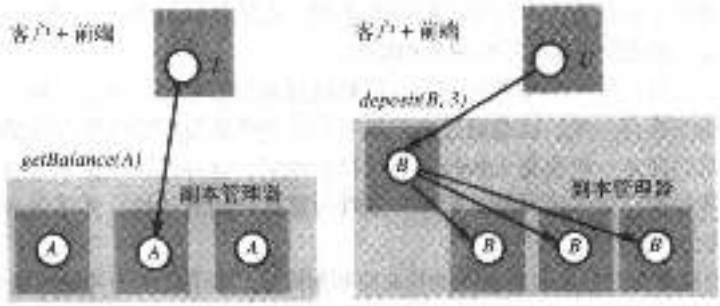


图15-10 复制数据上的事务

另一个问题是和前端联系的副本管理器是否应该延迟转发更新请求到别的管理器，直到一个事务提交为止，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将每一个更新请求转发到所有的管理器——及时方法。惰性方法是一个很好的选择：它降低了响应更新客户之前发生的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，及时方法是唯一可用的方案。

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，参与者再将这个消息传递给其他副本管理器，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，这个请求将传递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15-4）。对于主拷贝复制，并发控制被应用于主副本管理器上。
643 当提交一个事务时，主副本管理器和备份副本管理器通信，然后用及时方法应答用户。这种形式的复制可以在主副本管理器出故障时，由一个备份副本管理器一致地接管它。在惰性方法中，主副本管理器在它更新备份前就响应前端。此时，一个替代了故障前端的备份不一定有数据库的最新状态。

读一个/写所有 我们使用这个简单的复制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，前端可以和任何副本管理器通信。每一个write操作必须在任何副本管理器上执行，它在操作影响到的每个对象上加一个写锁。每个read操作由单个副本管理器执行，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write操作需要在所有副本管理器上请求冲突锁；在单一的副本管理器上，一个read操作和一个write操作将请求冲突锁。结果，获得了单拷贝串行化。

15.5.2 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，因为当副本管理器因为崩溃或发生通信故障而变得不可用时，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read请求可以由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由具有那个对象拷贝的副本管理器组中的所有可用副本管理器执行。"副本管理器组中可用成员"的概念和15.4.3节描述的Coda中的可用卷存储组非常相似。

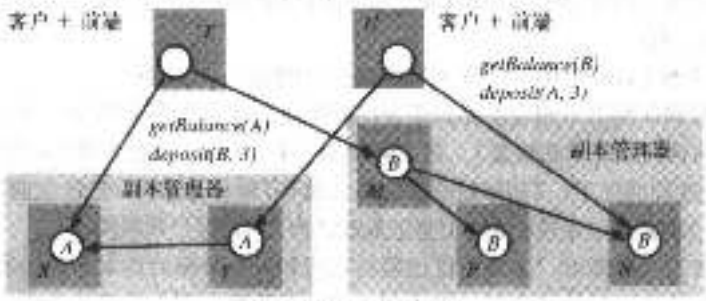


图15-11　可用的拷贝

在正常情况下，一个正常工作的副本管理器接收并执行客户的请求。read请求可由收到请求的副本管理器执行。write请求由收到请求的副本管理器和组中其他可用的副本管理器执行。例如，在图15-11中，事务T的getBalance操作由X执行，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执行的操作。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完成为止。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/写所有复制一样可获得单拷贝串行化。遗憾的是，如果相冲突的事务在进行过程中，副本管理器出了故障或正在恢复，就不是这种情况了。

644

副本管理器故障　我们假定副本管理器的故障是良性崩溃。崩溃的副本管理器被一个新的进程取代，它用一个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，并重新尝试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，但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，这时前端将重新发送请求到组中的另一个副本管理器。

就事务而言，单拷贝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，将无法获得单拷贝串行化。

考虑图15-11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后出现故障。假定在T和U执行deposit操作以前副本管理器X和N出现故障。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上对于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的并发控制也不会阻止T在副本管理器M和P上更新B。

这种现象与单拷贝串行化需求是相违背的。如果这些操作在对象的单一拷贝上执行，那么它们应该是可串行化的，即要么事务T在U之前执行，要么T在事务U之后执行。这保证一个事务可以读取另一个事务设置的值，对象拷贝的本地并发控制不足以在可用拷贝复制方案中保证单拷贝串行化。

由于write操作直接作用于所有可用的拷贝上，因此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的。相反，一个事务的read操作和另一个事务的write操作没必要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read操作和另一个的write操作相互依赖而形成一个环。如果对事务而言，故障和对象副本的恢复是串行化的，那么不会产生这样的依赖。

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
645 会在事务的执行过程中发生。在我们的例子中，当T已经对X上的一个对象进行了read操作，X的故障一定出现在T完成以后。同样的，当T试图更新对象时发现N出了故障，那么N的故障一定在T之前出现，即

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

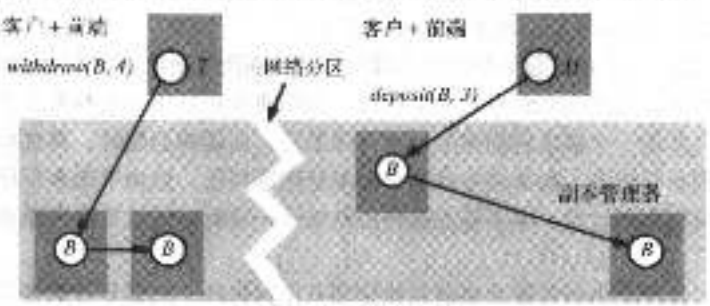
本地验证过程确保两个不相容的序列不会同时发生。在一个事务提交以前，它检查事务已访问的副本管理器的任何故障（和恢复）。在上面的例子中，T通过检查发现N仍然不可用，而X、M和P仍然可用。在这种情况下，T能够提交。这暗示着在T验证之后，U验证之前X出现故障。换言之，U的验证是在T的验证之后进行的。U验证失败是因为N已经出现故障。

每当某个事务发现故障时，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用于测试访问对象时，副本管理器是否发生故障，这些部分操作可以并入两阶段提交协议中。

当正常工作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。

15.5.3 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，在这种情况下，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15-12中，收到deposit的副本管理器不能将其发送给收到withdraw请求的副本管理器。



646 图15-12 网络分区

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[1985]讨论了多种不同的方法。按照是否容易发生不一致，这些方法可分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，然而悲观方法却对此有所限制。

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，该问题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，对更新加以验证——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，悲观算法也对可用性有所限制，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，它允许在主副本服务器所在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。

15.5.4 带验证的可用拷贝

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可以维持read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。如果没有发生分区，相互冲突的两个事务之一将被延迟或放弃。遗憾的是，当分区存在时，冲突的事务可以在不同的分区中提交。这种情况发生后的唯一选择就是放弃其中的一个事务。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。当会发生这种补偿行为时，乐观方法才是可行的。

可利用版本向量来验证相互冲突的write操作。这些方法在15.4.3节中已经描述过，并且已被用于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这类应用，因为对这种应用而言，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read和write操作影响的对象的日志。这个日志用来构建一个前驱图，图的节点是事务，它的边代表事务read和write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。 647

15.5.5 法定数共识方法

一种阻止分区中的事务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，因此每一个分区中的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。法定数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。该子组的其他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。如果可以使用版本，对象的初始状态是第一个版本，并且经过每一次变化后，我们有一个新的版本。每个对象的每个拷贝有一个版本号，只有最新的版本有当前版本号，而过时的拷贝有一个较早的版本号，操作只能应用于具有当前版本号的拷贝。

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的权重。每个read操作必须在它对任何最新拷贝进行读之前，先获得一个有R个选票的读法定数，每个write操作必须在它对任何最新拷贝进行更新之前，获得一个有W个选票的写法定数。其中，R和W是副本管理器组，它们满足下面条件：

W > 总选票的一半

R + W > 组选票的总数

这就确保了任何一对（由一个读法定数和一个写法定数或两个写法定数组成），必须包含相同的拷贝。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。

为了进行一个read操作，首先必须通过足够多的版本号查询来发现一组拷贝，从而收集一个读法定数，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。由于每个读法定数和每个写法定数存在重叠，每个读法定数必定至少包括一个当前拷贝。read操作可在任何最新的拷贝上执行。

为了进行一个write操作，首先必须通过足够多的版本号查询来收集一个写法定数，法定数中的成员必须具有最新的拷贝，并且选票的数量不得少于W。如果没有足够的最新拷贝，那么一个非当前的文件会被一个当前文件的拷贝所替代，以使法定数得以建立。由写法定数中的每个副本 648

管理器进行write操作中指定的更新，增加所有对象副本的版本号，write操作的完成要报告给客户。

然后，在剩下可用的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，如果它的文件拷贝的写版本号比写法定数拥有的文件拷贝的版本号旧时，整个文件由来自最新更新过的副本管理器的一个副本来替换。

在Gifford的复制方案中，两阶段读-写加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数R时，每个副本管理器都被设置了一个读锁。当在写法定数W中执行write操作时，每个被涉及的副本管理器上设置了一个写锁（这里，锁的粒度与版本号有相同的粒度）。由于一个读法定数和一个写法定数重叠，并且两个写法定数也重叠，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 加权投票算法的一个重要性质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得到一个副本管理器组的可靠性和性能，write操作的可靠性和性能的提高可以通过减少W而得以增加，同样可以通过减少R来提高读操作的可靠性和性能。

该算法既允许使用客户机本地磁盘的文件拷贝，也允许使用文件服务器上的文件。客户机上的文件拷贝被认为是*弱代表*，并且经常给它们分配0个选票。这就确保它们不会包含在任何法定数中。一旦某个读法定数构造成功，一个read操作就可以在任何最新的拷贝上执行。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。弱代表可用来加快read操作速度。

Gifford的例子 Gifford给出了三个例子，这三个例子通过给一个组上的不同副本管理器分配权重和分配适当的R和W，从而显示出不同的特性。现在基于下面的表再现Gifford的例子。阻塞概率表示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，任何副本管理器不可用的概率均为0.01。

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流量。

例2用来配置一个有中读写率的文件，该文件主要通过局域网被访问。局域网上的副本管理器被分配两个选票，远程网络上的每个副本管理器被分配一个选票。读可以在本地副本管理器上执行，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。如果本地副本管理器出现故障，
649 文件在只读模式下仍然是可用的。客户为了获得更低的读延时，可以创建本地的弱代表。

例3用来配置一个具有非常高读写率的文件，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，为了降低读操作延时，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
法定数大小	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

法定数共识方法的主要缺点是，由于需要从R个副本管理器中收集一个读法定数，因此read操作的性能被降低了。

Herlihy[1986]引入抽象数据类型扩展了法定数共识方法。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的方法使用时间戳而不是版本号，这样做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声称的主要好处是使用语义知识可以提高法定组选择的数量。

15.5.6 虚拟分区算法

该算法由El Abbadi等[1985]提出，结合了法定数共识和可用拷贝两种算法。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read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”是指将副本管理器分成许多部分的屏障，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 650
还是很像15.2.2节介绍的组视图。如果虚拟分区包含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，这样做的好处是read操作只要访问某个对象的单一拷贝，因此可以通过选择“最近”的拷贝来提高性能。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发生变化，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，从而确保了事务的单拷贝可串行化。

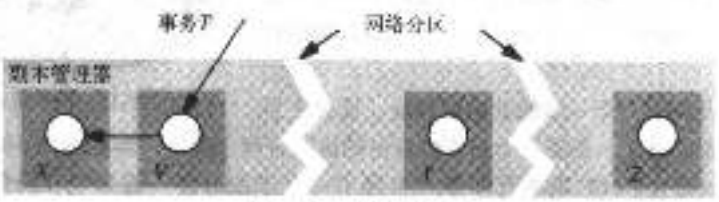


图15-13 两个网络分区

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write操作没被确认时），它试图创建一个具有读、写法定数的新虚拟分区。

例如，设想有4个副本管理器V、X、Y和Z，每一个副本管理器都有一个选票，并且写和读法定数是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。例如，一个事务T由读操作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read操作，并且所有的4个副本管理器上进行write操作。

假设事务T开始在V上执行read操作时，V仍和X、Y、Z相连。发生网络分区后（如图15-13所示）后，V、X在一部分，Y、Z各在不同的分区。那么，当事务T试图执行write操作时，V将注意到它已不能连接到Y和Z了。

当一个副本管理器不能和它先前连接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为止。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为止，像图15-14所示Y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数和写法定数。

在一个事务执行期间（比如事务T已经在一个副本管理器上执行一个操作），这时创建了一个新的虚拟分区，那么这个事务必须放弃。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read操作可在任何副本上执行。 651

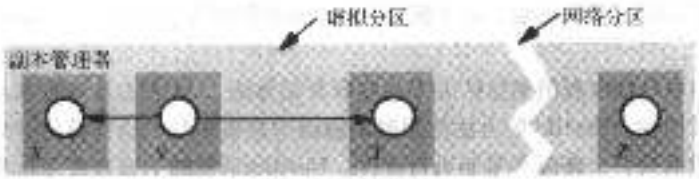


图15-14 虚拟分区

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（一个它们可以与之通信的副本管理器的共享视图）。例如，在图15-14中，潜在的成员是V、X、Y、Z，而实际的成员是V、X和Y。

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15-13中的副本管理器Y、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，虽然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。此时存在的一个危险是创建两个相互重叠的虚拟分区，如图15-15中的V_1和V_2。

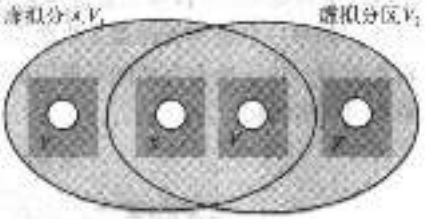


图15-15 两个重叠的虚拟分区

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。在这种情况下，它的读锁将不会和另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。

652 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15-16。

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Confirmation消息为止。

阶段1：

- 发起者发送一个Join请求给每个潜在的成员。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，那么它同意加入并回复Yes。
  - 如果请求中的逻辑时间戳小，那么它拒绝加入并回复No。

阶段2：

- 如果发起者收到了足够的Yes应答，从而获得多数和写法定数，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戳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15-16 创建一个虚拟分区

在上面的例子中，图15-13中显示的副本管理器Y、Z都试图创建一个虚拟分区，具有较高逻辑时间戳的副本管理器，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，

15.6 小结

复制对象是在分布式系统中获得具有高性能、高可用性和容错性质的服务的重要手段。我们描述了复制服务的体系结构，其中副本管理器掌管着对象的副本，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一组物理副本来实现。可以通过组通信来非常方便地更新这些副本。我们扩展了组通信内容，以包括组成员关系服务和视图同步通信。

我们定义了线性化能力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，服务必须如何保证它们与逻辑对象集合的单个映像等价。最有实际意义的准则是顺序一致性。 653

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副本管理器代替它，便可以获得容错。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都允许客户在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来获得它所具有的最高的可用性。Bayou提供了更强的最终一致性保证，采用了自动冲突检测以及操作变换技术来解决冲突。Coda是一个高可用文件系统，它使用版本向量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。即使在某些并不是所有的副本管理器都可达的环境下，可用拷贝、法定数共识和虚拟分区的技术仍能使事务中的操作继续进行。

练习

15.1 三台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如何？ （第604页）

15.2 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。 （第607页）

15.3 在一个多用户的游戏中，多个玩家在一个公用屏幕上进行游戏。游戏的状态被复制到玩家各自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，例如碰撞检测等。更新被组播给所有的副本。

(1) 这些游戏人物之间可能相互射弹，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。

(2) 玩家可能使用一个外接的操作设备来玩这个游戏，那么对这个设备的操作需要什么样的次序？ （第608页） 654

15.4 一个路由器将进程p与另外两个进程q和r分开。p组播消息m后路由器就出现故障。如果组通信系统是视图同步的，接下来进程p将会怎样？ （第612页）

15.5 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组件，来构造一个视图同步组通信系统？ （第612页）

15.6 同步有序的组播操作的传递排序语义和视图同步组通信系统中的传递视图的语义相同。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这个服务上执行操作。试解释为什么从列表中删除用户应该是同步有序操作？ （第612页）

15.7 由状态迁移引起的一致性问题是什么？ （第613页）

15.8 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。 (第614页)

15.9 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？ (第616页)

15.10 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会导致顺序一致性，而不是线性化执行？ (第619页)

15.11 gossip体系结构能够应用于练习15.3描述的分布式游戏吗？ (第622页)

15.12 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？ (第626页)

15.13 在gossip系统中，前端有一个向量时间戳（3，5，7），代表着它从一个有三个副本管理器的组中的成员接收到的数据。相应的，这三个副本管理器有向量时间戳（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？ (第627页)

15.14 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？ (第631页)

15.15 对于一个简单的房间预定应用，写出（如Bayou中使用的）依赖性检查和合并过程的伪代码。 (第633页)

15.16 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？ (第640页)

15.17 请设计一种方案来集成文件系统目录的两个副本，它们能够在断链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。 (第641页)

655

15.18 在数据项A和B上应用可用拷贝复制，因此具有副本A_x、A_y和B_m、B_n。事务T和U定义如下：

T：Read(A)；Write(B,44)；U：Read(B)；Write(A, 55)

假定在副本上应用两阶段加锁，设计一个T和U的交错序列。解释为什么在T和U的执行中出现副本故障时，只用锁不能确保单拷贝串行化。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。 (第644页)

15.19 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副本的初始值是100，并且在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。

（1）当客户执行这些操作时，出现了一个分区，将服务器X和Y与服务器Z分离了。描述当客户能访问服务器X和Y时，获得的法定数和发生的操作。

（2）描述当客户仅能访问服务器Z时，获得的法定数和发生的操作。

（3）分区修复了，然后另一个分区发生了，结果X和Z与Y分离了。描述当客户能访问服务器X和Z时，获得的法定数和发生的操作。 (第649页)

656

第16章 移动计算和无处不在计算

本章将探讨移动计算和无处不在计算领域中的问题，这些问题是由于设备小型化和无线连接的出现而产生的。从广义上说，移动计算主要研究关于便携设备之间的连通问题；无处不在计算则研究日常物理世界中计算设备的增量集成问题。

本章将介绍一种常用的系统模型，它强调移动系统和无处不在系统的易变性：在任何给定环境中的用户、设备和软件组件都在频繁地改变。之后，本章还将研究涉及易变性和易变性物理基础的几个研究领域，包括当实体移动、失效或自发出现时，软件组件之间如何实现互连和互操作；系统如何通过感知和上下文敏感信息与物理世界集成；集成系统中的安全和私密问题以及适合计算能力和I/O资源相对缺乏的小型设备的技术。本章的最后会以Cooltown项目进行实例研究，Cooltown为移动计算和无处不在计算设计了一种面向人的、基于Web的体系结构。 657

16.1 简介

设备的小型化和无线连接的出现导致了移动计算和无处不在计算的产生。从广义上说，移动计算研究的是日常物理世界中移动设备的连通问题；无处不在计算研究的是物理世界中设备的增量集成问题。随着设备越来越小，我们能将它们带在身边或穿戴它们，而且我们能将它们嵌入到物理世界的许多物体中——不只是安装在我们熟悉的桌面设备中或服务器架上。而且，随着无线连接越来越普遍，我们可以将这些新型便携设备互连，或者连接到传统的个人计算机或服务器上。

本章将概述移动计算（第15章的断链操作处理已经涉及这个主题）和无处不在计算的各个方面。本章侧重它们的共有特性和它们同传统分布式系统的不同。虽然本章将给出该领域最新的进展，但本章将更关注开放性问题，而不是解决方案。

本章将首先介绍移动计算和无处不在计算的起源，并介绍若干子领域，包括可穿戴计算、手持计算和上下文敏感计算。之后，本章还将围绕这些领域的特性——易变性，即在任一给定环境下用户、设备和软件组件都在频繁地改变——给出一个系统模型。本章随后将讨论涉及易变性和易变性物理基础的几个主要研究领域，包括当实体移动、失效或自发出现时，软件组件之间如何实现互连和互操作；系统如何通过感知和上下文敏感信息与物理世界集成；易变的、物理上集成的系统所引发的安全和私密性问题以及适合计算能力和I/O资源相对缺乏的小型设备的技术。本章最后以Cooltown项目作为实例研究对象，Cooltown为移动计算和无处不在计算设计了一种面向人的、基于Web的体系结构。

移动计算 移动计算最初作为一种能够保持用户所携带的个人电脑与其他机器的连接的计算范型出现。大约到1980年才出现适合携带的个人电脑，并且它们可以使用调制解调器通过电话线与其他电脑连接。技术进化大致沿着这个理念，并获得了更好的功能和性能：今天的便携产品有笔记本电脑或小型的笔记本电脑，它们均与无线连接（包括红外线、WiFi、蓝牙、GPRS或3G通信技术）相结合。

技术进化的另一条路径产生了*手持计算*。利用手持设备，包括个人数字助理(PDA)、手机和其他的更专门化的手持操作设备。PDA是通用计算机，可以运行许多不同类型的应用程序，但是与笔记本电脑和笔记本相比，它们必须在大小、电池容量和相对有限的处理能力、小屏幕和其他资源限制间进行折衷。制造商陆续为PDA装配了与笔记本电脑相似的无线连接功能。 658

手持计算的一个有趣的趋势是模糊了PDA、手机和专业手持设备（例如照相机）之间的差别。某些类型的手机通过运行Linux、Symbian或Microsoft Smartphone操作系统获得了类似于PDA的计算功能。PDA和手机可以装配摄像头、条形码阅读器和其他类型的特殊配件，使它们成为某种专业手持设备的替代品。例如，想拍数码照片的用户可以使用专业的相机，也可以使用一个带摄像头的PDA，或可拍照手机。以上所有设备都自带（或购买）短距离或长距离的无线连接能力。

Stojmenovic[2002]介绍了无线通信的原理和协议，包括本章所研究的系统需要解决的网络层的两个主要问题。第一个问题是当移动设备进出基站覆盖范围时，如何保证它们的持续连接，基站是提供无线覆盖区域的基础设施。第二个问题是在没有基站的地方，设备集合之间如何进行无线通信（见16.4.2节自组织网络给出的简洁的处理方法）。通常，在两个给定设备之间无法建立直接的无线连接时，上面两个问题都会出现。通信需要经由几个无线或有线网段来完成。下面两个因素导致无线覆盖必须划分为若干子覆盖。第一，无线网络的范围越大，就有越多的设备竞争网络的有限带宽。第二，考虑能量的使用，传输一个无线信号所需的能量与它传输距离的平方成正比，但是我们关注的很多设备的能量有限。

无处不在计算 1988年，Mark Weiser提出了无处不在计算的概念[Weiser 1991]。无处不在计算有时也称为普适计算，这两个术语通常被认为是同义的。“无处不在”的意思是“处处存在的”。Weiser看到了计算设备的普及，并相信它们会让我们使用计算机的方式产生一场革命性的变革。

首先，世界上每个人会使用多个计算机，我们可以把它与之前的个人计算机革命——每个人拥有一台计算机——相提并论。尽管这听起来简单，但是与之前的主机时代（那时是一台计算机有多个用户）相比，这种改变将对我们使用计算机的方式产生巨大的影响。Weiser的“一个人，多台计算机”的理念与通常的理解有很大的不同，通常的理解是每人有多台计算机——一台在单位，一台在家里，一台笔记本电脑和一个可能随身携带的PDA。更确切地说，在无处不在计算中，为了适应不同的任务，计算机的增加是在形式上和功能上，而不只是在数量上。

例如，一个房间内的所有固定的显示屏和书写工具——白板、书、纸张、便签等等——被几十个、甚至上百个带有电子屏幕的计算机代替。白板可以协助人们表达、组织和归纳他们的想法；书本可以变成设备，使读者可以搜索文本、查找词意、在Web上搜索相关的想法并查看连接的多媒体内容。现在，设想在所有的写作工具中嵌入计算功能。例如，笔和各种标记工具能存储用户
659 写的和画的内容，并且可以在周围的多台计算机之间收集、拷贝和移动这些多媒体内容。该场景没有考虑可用性和经济性问题，而且它只涉及我们生活的一小部分，但它给了我们关于“计算处处存在”可能是什么样子的一个想法。

Weiser预测的第二个转变是计算机将要“消失”——它们将“渗透于日常生活中，直至不可或缺”。这在很大程度上只是一种心理概念，正如人们认为家具是理所当然该有的，因而很少注意到它们。这反映出计算机将融入到我们的日常用品中去——正常情况下，我们不认为这些物品具有计算能力，正如我们并不认为洗衣机或车辆是“计算设备”，即使它们由嵌入在其中的微处理器控制——有些汽车中有大约100个微处理器。

虽然某些设备是不可见的——例如在汽车中嵌入计算机系统的情况——但是我们所关心的设备并不都是这样的，尤其是移动用户经常携带的设备。例如，手机在当前是一种最普遍的设备，它的计算能力是可见的，并且无疑的，它也应该是可见的。

可穿戴计算 用户能够在他们身上携带可穿戴设备，可以附在他们衣服的外面或里面，也可以像戴手表、珠宝和眼镜一样戴在身上。与我们上面提到的手持设备不同，这些设备经常在无需用户操纵的情况下运行。它们通常具有特定的功能。一个早期的例子是“Active Badge”，它是一种可以夹在用户衣服上的小型计算设备，其功能是定期通过一个红外线发送装置广播一个（与用户相关的）标识符[Want et al 1992;Harter and Hopper 1994]。标记点的作用是使环境中的设备对标记发送的信

息做出响应，从而对用户的出现做出响应；红外线发送的作用范围有限，所以只有用户在附近时才能被设备发现。例如，一个电子屏幕可以依照用户的偏好（例如默认的绘画颜色和线宽度）定制行为来响应用户的出现（如图16-1所示）。可以根据房内的人来调整房间的空调和灯光的设置。

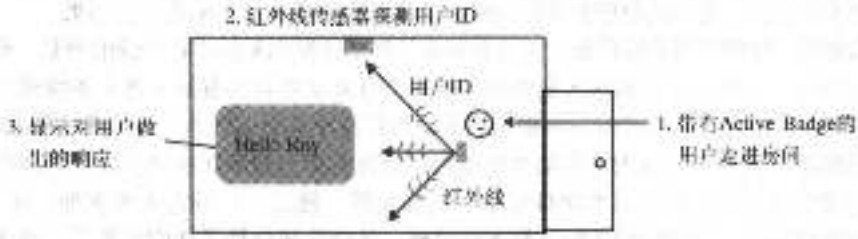


图16-1 一个对带有Active Badge的用户能做出响应的房间

上下文敏感计算 Active Badge——或者说是其他设备对它的出现做出的反应——作为一个例子解释了上下文敏感计算。上下文敏感计算是移动计算和无处不在计算中一个很重要的子领域。这就是计算机系统根据物理环境自动调整它们的行为。这些环境原则是物理可测量的或可觉察的，例如用户的出现、一天的时间或天气的状况。一些依赖条件确定起来比较简单，例如根据时间、日期和地理位置判断现在是不是晚上。但其他依赖条件需要经过复杂的处理来检测。例如，考虑一个上下文敏感的手机，它只在适合的时候才响铃，特别是，在电影院，它应该自动切换到“振动”而不是“响铃”。但是要它感知用户是在电影院里看电影，还是站在电影院的门廊上，却不是件容易的事情（假定位置传感器的测量并不准确）。16.4节将会更详细地介绍上下文。 660

易变系统

从分布式系统的观点看，移动计算和无处不在计算或我们已经介绍的子领域（更确切的说，还包括我们省略的子领域（例如可触摸计算[Ishii and Ullmer 1997]和类似Wellner数字桌面的强真实感领域[Wellner 1991]）之间没有本质的区别。在本节中，我们将给出一个称为易变系统的模型，它包括了以上所有领域中的分布式系统的本质特征。

我们之所以称本节所描述的系统是“易变的”，是因为与本书其他部分描述的大多数系统不同，某些在其他系统中异常的改变在该系统中是很平常的。移动系统和无处不在系统中的用户、硬件和软件组件是高度动态的，并且其变化不可预计。我们对这些系统有时使用另一个词：自发的，在文献中该词出现在词组自发网络中。易变性的相关形式包括：

- 设备和通信链接故障。
- 通信特征（例如带宽）的改变。
- 设备上的软件组件之间的关联——逻辑上的通信关系——的建立和中断。

这里，术语“组件”包括任何的软件单元，例如对象或进程，而不管它是进行互操作中的一个客户、一个服务器还是一个对等点。

第15章已经介绍了一些处理改变的方法，即处理故障和断连操作的方法。但是那里的解决方案将进程和通信故障作为异常而不是规则，而且是以存在冗余处理资源为前提的。易变系统不但违反了前述假设并且在组件间的关联中加入了甚至更多的变化现象，变化频率更显著。

在我们深入介绍易变性之前，有必要澄清一些误解。易变性不是移动系统和无处不在系统定义的一个属性；其他类型的系统也显示出一种或多种形式的易变性，但是它们既不是移动的也不是无处不在的。一个很好的例子是对等计算，例如文件共享应用（见第10章），其中参与进程集合和集合元素之间的关联都在频繁地发生改变。移动计算和无处不在计算的不同之处在于，由于它们与物理世界集成的方式，它们表现出了上述易变性的所有形式。我们将对物理集成和如何产生 661

易变性进行更多介绍。但是物理集成不是分布式系统的属性，而易变性则是分布式系统的属性。因此我们采用易变性这个术语。

我们将在本节余下的部分描述智能空间，它是易变系统存在的环境；之后我们将描述移动和设备无处不在设备，它们的物理和逻辑连接，以及在低信任和低私密性下的后果。

智能空间 物理空间非常重要，因为它形成了移动计算和无处不在计算的背景。移动性发生在物理空间之间；无处不在计算嵌入在物理空间内。智能空间是具有嵌入服务的物理空间，也就是说，服务只在或原则上在物理空间内提供。可以将计算设备引入野外——那里没有基础设施——来执行应用程序，例如环境监测。但是更典型的移动设备和无处不在系统在任何时间都存在于一个有计算能力的建筑的一部分或车辆环境中，例如房间、建筑、广场或火车车厢。在这些情况下，智能空间通常包括一个相对稳定的计算基础设施，基础设施包括常规的服务器、设备（例如打印机和显示器）、传感器和一个无线网络基础设施（能够连接到因特网）。

在智能空间中存在几种移动或“出现和消失”。第一，*物理移动性*。智能空间可以作为访问和离开它们的设备的环境。例如，用户可以携带或穿戴设备进入和离开；机器人甚至可以自己移入和移出空间。第二，*逻辑移动性*。移动进程或代理可能移入或移出智能空间，或者移入或移出用户的个人设备。而且，设备的物理移动可能导致其内部组件的逻辑移动。然而，不论组件的移动是否是由于它的物理设备的移动造成的，不会发生有意义的逻辑移动，除非组件改变了它与其他组件的关联。第三，用户可以加入相对静止的设备（例如多媒体播放器）使其在空间中长期存在。相反，用户也可以从空间中撤出旧设备。例如，考虑*智能家居*的发展，居住者经过一段时间之后，就会以一种相对无计划的方式改变其中的设备布置[Edwards and Grinter 2001]。最后，设备可能失效并从空间中“消失”。

从分布式系统观点看，有一些现象是类似的。在每种情况下，一个软件组件可以*出现*在一个业已存在的智能空间中，并且如果有感兴趣的事件形成，那么它将与空间（至少是暂时的）集成；或者一个组件通过移动、关闭或者失效从空间中*消失*。在上述情况下，任何特定的组件不一定能够区分“访问”设备和“基础设施”设备。

然而，在设计一个系统时，要抽取最重要的区别。易变系统之间的一个重要区别是变化的频率。用于处理一天中少量组件出现或消失的算法（例如，在智能家居内）与用于处理任何时候至
662 少有一个组件变化的算法（例如，在一个拥挤的城市，手机之间使用蓝牙通信实现的系统）有很大的不同。此外，虽然以上的出现和消失现象看上去相似，但依然存在很大的区别。例如，从安全性的角度看，用户的设备进入智能空间是一件事，外面的软件组件进入属于该空间的基础设施设备中则是另一件事。

设备模型 随着移动计算和无处不在计算的出现，一种新型的计算设备成为分布式系统的一部分。该设备的能量供给和计算资源有限；它有与物理世界交互的途径：传感器（例如光线检测器）或制动器（例如可以控制移动的可编程工具）。

有限能量：物理世界中的便携式或嵌入式设备通常使用电池，并且设备越小、越轻，它的电池容量就越小。从时间（可能一个用户拥有几百台这样的设备）和物理访问而言，更换电池或充电是不方便的。计算、访问内存或其他形式的存储动作都会消耗宝贵的能量。无线通信是典型的能量密集型应用。此外，接收一条消息所消耗的能量与发送消息所需能量相差不多。即使处于“待机”模式，在此模式下网络接口准备接收消息，也需要消耗少许的能量[Shih et al.2002]。因此，如果设备要在给定的电池容量下持续尽可能长的时间，算法就需要考虑设备消耗的能量，特别是根据消息复杂度来考虑，最后，由于电池放电，设备故障的可能性也会增加。

资源限制：就处理器速度、存储能力和网络带宽而言，移动设备和无处不在设备的计算资源有限。部分原因是我们若提高这些特性，能量消耗就会随之增加。而且，若要实现设备便携或嵌

入到日常物品中，就需要将它们做得很小，并对制作过程加以限制，限制每个节点上晶体管的数目。这会引发两个问题：尽管资源有限，如何设计出可在节点上用合理时间执行完成的算法；如何利用环境中的资源来增强节点贫乏的资源。

传感器和制动器：为了使设备与物理世界集成——特别地，使得它们是上下文敏感的——给设备配备了*传感器*和*制动器*。传感器是测量物理参数并把值传给软件的设备；制动器是影响物理世界的由软件控制的设备。每种组件都有很多类型。例如，有测量位置、方向、负载以及光线和声音的传感器。制动器包括可编程的空调控制器和发动机。传感器的一个重要的问题是精度，因为精度很有限，所以可能导致虚假的行为，例如一个不合适的响应导致位置错误。不精确可能是这类设备的一个特性，因为它们要足够廉价以便可以到处部署。

以上描述的设备听起来有些奇特，然而，它们不但在商业上是可用的，有些甚至是批量生产的。这样的例子有mote和拍照手机。

663

mote：mote（微尘）[Hill et al. 2000; www.xbow.com]是用于满足应用程序（例如环境感知）的自治操作的设备。它们被设计成嵌入式的、可编程的，这样它们能无线地发现彼此并且在它们之间传送感知到的数据。例如，倘若有一场森林火灾，那么散布在森林周围的多个mote就能感知到不正常的高温，并且通过其他mote在节点把信息通知给一个较高功率的设备，该设备能够把这种情况传达给紧急服务。mote的最基本形式是：具有一个低功率处理器（一个微控制器），该处理器在内部闪存中运行TinyOS操作系统[Culler et al. 2001]；用于记录数据的内存；一个短程、双向地使用“工业、科学和医学”（ISM）波段的无线电收发机。还可以包含多种传感器模块。mote也称为“智能尘埃”，这反映了这些设备微小的尺寸，尽管在写书时它们的尺寸大约是6×3×1厘米（不包括电池组和传感器）。Smart-its以一种类似的形式因素（form factor）提供类似mote的功能[www.smart-its.org]。16.4.2节将讨论类似mote的设备在无线传感器网络中的使用。

拍照手机：拍照手机在我们考虑的系统中是一种非常不同的设备。它们的主要功能是通信和照相。但是，运行诸如Symbian操作系统后，它们就可编写多种应用程序。除了它们的广域数据连接之外，通常有红外线（IrDA）或蓝牙短程无线网络接口使得它们可以彼此连接，或与PC和感知设备连接，例如用于确定它们位置的GPS或其他卫星导航装置。此外，它们可以运行软件从相机图像中识别符号（例如条形码），使得它们成为物理物体（例如产品）上“编码数值”的传感器，这些传感器信息就可用于访问相关的服务。例如，用户可以用他们的手机从产品盒子上的条形码发现商店产品的说明[Kindberg 2002]。

易变连接 本章感兴趣的设备都有某种形式的无线连接，而且可能会有多种。连接技术（蓝牙、WiFi、GPRS等）根据它标称的带宽和延迟、能量消耗以及是否需要为通信支付费用而变化。但是连接的易变性——在运行时设备之间的连接或断链状态的易变性和服务质量的易变性——对系统属性也有很大的影响。

断链：无线断链与有线断链有很大的不同。我们描述的很多设备是移动的，并且可能超出了与其他设备的操作距离或者遇到了无线电障碍物，例如建筑物。即使设备静止时，也可能有移动的用户或车辆成为阻塞物而导致断链。设备间还存在多跳无线路由的问题。在*自组织路由*中，一组设备之间彼此通信，而不依赖于其他设备：它们协作路由所有的包。以森林中的mote为例，mote可能与其他mote在无线电范围内直接通信，但是不能将它的高温信息传送给紧急服务，因为所有包都要经过的较远的那个mote出现了故障。

664

变化的带宽和延迟：导致完全断连的因素也可能导致带宽和延迟的显著变化，因为它们带来错误率的变化。随着错误率的上升，越来越多的包被丢弃。这在本质上导致了吞吐率降低。然而由于高层协议的超时，情况可能会更加恶化，很难设置合适的超时数值来适应明显变化的环境。如果与当前的错误情况相比，超时值太大，那么延迟和吞吐量就会受影响。如果超时值太小，可

能会增加拥塞而且浪费能量。

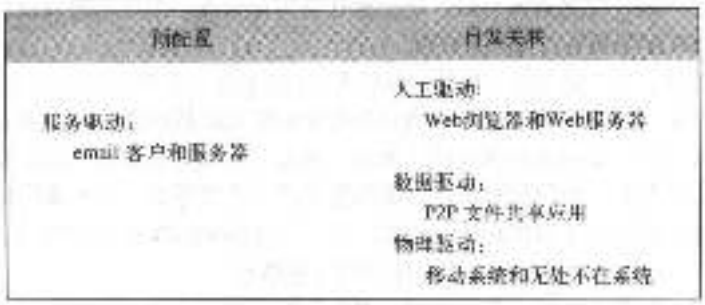


图16-2 预配置与自发关联的例子

自发互操作 在易变系统中，随着组件的移动或其他组件的出现，就会改变进行组件间通信的组件集合。我们使用术语关联来表示逻辑关系，当一对给定的组件中的至少一个组件与另一个组件在某一定义好的时间段内通信时，逻辑关系就会产生；使用术语互操作表示它们在关联中的相互作用。注意，关联不同于连接：两个组件（例如，笔记本电脑的邮件客户端和邮件服务器）可能在当前已经断连，但仍然可以保持关联。

在一个智能空间中，组件可以充分利用与本地组件相互作用的机会而改变关联。举例来说，无论设备出现在哪儿，它都可以使用本地打印机。类似地，设备可能想为本地环境中的客户提供服务——例如，用户穿戴的（例如，在他的皮带上）的个人服务器[Want et al. 2002]，它为空调装置提供用户的匿名属性，这样该装置能根据用户的喜好调整房间的环境。当然，某些静态关联在易变系统中仍然有意义。例如，一台笔记本电脑跟随它的主人走遍世界，但仍然仅与一个固定的邮件服务器通信。

为了将这种类型的关联放到因特网上服务的大背景中，图16-2给出了三种类型的自发关联的例子（在右边），并与预配置关联（在左边）相比较。

665 预配置关联是服务驱动的。也就是说，客户需要长期使用一种特殊服务，所以他们通过预配置以便与之关联。配置客户的开销（包括用所需服务的地址建立它们）与使用某种服务的长期收益相比是很小的。

在图的右边是按常规变化的关联类型，包括由人工操作驱动，由对特殊数据的需求驱动或由物理环境的改变驱动。我们认为，Web浏览器和Web服务之间的关联是自发的和人工驱动的，用户作出动态的和（从系统角度看）不可预测的点击，这样便可访问服务实例。Web是个真正的易变系统，对于它的成功而言，最重要的是关联的改变通常涉及可忽略的开销——Web网页的作者已经做了配置工作。

因特网上的对等网络应用程序（例如文件共享程序）也是易变系统，但它们主要是数据驱动的。数据经常来源于人（例如，所寻找的内容的名字），但是，正是提供给它的数据值导致对等节点通过一个基于数据的分布式发现算法与另一个对等节点关联，这个节点与它以前可能从未关联过，并且以前也未存储过该节点的地址。

本章讨论的移动和无处不在系统与现有的大规模物理驱动的自发性关联不同。关联的建立和破坏（有时由人完成）依据组件当前的物理环境，尤其是它们的靠近程度。

低信任和私密性 正如第7章所述，分布式系统的安全基于可信任的硬件和软件——可信任的计算基础。但是由于自发性互操作，易变系统中信任是成问题的。能自发关联的组件间的信任基础是什么？在智能空间之间移动的组件可能属于独立的个体或组织并且对彼此或可信任的第三方了解很少。

私密性是用户关心的主要问题，用户可能由于对系统的感知能力而不信任系统。在智能空间中，传感器的出现意味着它可以在先前无法看到的、潜在巨大的范围内跟踪用户。通过上下文敏感服务（如前面房间的例子，可依据用户的喜好来设置空调），用户可以让他人知道他们在哪儿和他们在那儿做什么。更糟糕的是，他们可能并没有意识到他们被感知了。即使用户没有暴露他们的身份，其他用户也可能了解到并查找出到底是哪一个人。例如，通过观察工作地点与住址之间的有规律的路径，并且将那些和住址与工作地点间信用卡的偶尔使用相关联。

16.2 关联

正如上面所描述的，设备容易在智能空间中出其不意地出现和消失。尽管如此，易变组件需要互操作——最好没有用户的干预。也就是说，突然出现在智能空间中的设备需要能通过自举将自 666
己引导进局域网络，从而与其他设备通信，并且能在智能空间中适当地关联：

- **网络自举** 通常，通信发生在局域网内。设备必须首先在局域网内获得一个地址（或注册一个已存在的地址，例如移动IP地址）。它可能还要获得或注册一个名字。
- **关联** 设备上的组件或者关联到智能空间中的服务，或者为智能空间的其他组件提供服务，或者两者都有。

网络自举 目前已有很好的方法来解决网络中设备的集成问题。某些解决方案必须访问智能空间中的服务器。例如，DHCP服务器（见3.4.3节）能提供IP地址、其他网络和DNS参数，设备通过给一个众所周知的广播地址发出一个查询获得这些信息。智能空间中的服务器也可以给设备分配一个唯一的域名；如果已经开放了对因特网的访问，那么设备可以使用一个动态DNS更新服务来注册它的新的IP地址，以取代一个静态的域名。

一个很有趣的情况是在没有任何服务基础设施的情况下，如何在智能空间中（甚至其外）分配网络参数。这对于简化智能空间和避免可能失败的服务间的依赖是很有帮助的。IPV6标准包括一个无服务器地址分配协议。IETF的零配置联网工作组[www.zeroconf.org]正在为无服务器地址分配、域名查找、组播地址分配和服务发现（见下一节）开发标准。苹果电脑公司（Apple）的Rendezvous [www.apple.com]是一个包含上述大多数功能的商业实现。就像DHCP访问一样，所有的这些方法通过使用一个众所周知的地址在局域网内进行广播和组播。任何设备都能向这个地址发送消息，也能监听发往这个地址的消息，并且只涉及设备自己的网络接口。

关联问题和边界原理 一旦设备能在智能空间中通信，它就面临着关联问题：在智能空间中如何适当地关联。解决关联问题方案必须处理好两个方面的问题：规模和范围。第一，智能空间中每立方米内有几十甚至几百个设备，并且在这些设备上可能有更大数量级的软件组件。突然出现的设备上的组件应该与原智能空间中的哪些组件交互（如果有的话）？如何保证这种选择是高效的？

第二，当解决了上面的问题后，我们如何限制范围，以便只需要考虑智能空间中的组件（或者智能空间中的所有组件）而不是其外的成千上万个组件？范围并不仅仅指规模问题。一个智能空间通常有管理和领域边界，这对用户和管理者有非常大的区别。例如，如果宾馆房间内一个设备想发现一个服务（例如打印机），它就必须要在它的用户房间内（而不是隔壁房间）找到一个打印机。同样的，如果在用户房间内有合适的打印机，那么解决方案应该将它作为一个关联的候选包括在内。

*边界原理*就是指智能空间需要有系统边界，它精确地对应于一个有意义的空间，因为它是在空间范围上和管理上有正规定义的[Kindberg and Fox 2001]。那些“系统边界”是系统定义的标准 667
范围，在其内不必限制关联。

一个尝试解决关联问题的方案是通过使用下文描述的一个发现服务，它具有Jini发现服务的一个账户。发现服务通常基于子网组播，其缺点是子网范围可能与智能空间中可用的服务不一致——

它们破坏了边界原理。16.2.2节将描述一些解决方案，这些解决方案依赖物理参数和人工输入以提供更精确范围的关联。

16.2.1 发现服务

客户使用发现服务来发现智能空间提供的服务。发现服务是一个目录服务（见9.3节），智能空间的服务在其中注册，并将它们的属性作为查找的关键字，但是它们的实现要考虑易变系统的特性。这些特性包括：第一，某个客户所要求的目录数据（即将要查询的服务的属性集合）在运行时才能确定，作为客户上下文（指发生查询的智能空间）的一个功能，目录数据是动态确定的。第二，智能空间中可能没有存放目录服务器的基础设施。第三，目录中注册的服务可能自发消失。第四，访问目录使用的协议应该了解它们所消耗的能量和带宽。

目前，存在设备发现服务和服务发现服务，蓝牙包括这两者。客户通过设备发现获得设备的名字和地址，通常，他们随即根据额外信息（例如由人选择）选择一个单独的设备，并查询它所提供的服务。另一方面，当用户不关心他们所需要的服务由哪个设备提供，而是只是关心服务的属性时，他们就可以使用服务发现服务。本节的描述将集中在服务发现服务上，除非特别说明，否则这就是我们之前所说的发现服务的意思。

可以用接口来自动注册和注销可用于关联的发现服务，客户也可以通过接口查找当前可用的服务，从而继续与合适的服务关联。图16-3给出这些接口的一个假想的、简化的例子。首先，通过函数调用用给定的地址和属性注册可用的服务，并且通过调用来管理随后的注册。之后，若有函数调用查找与所需属性匹配的服务，则可能有零个或多个服务匹配，每一个服务返回它的地址和属性。注意，仅有发现服务不能启用关联：可能需要服务选择——从返回集合中选择一个服务。这可以由编程实现，或者通过列出匹配的服务让用户来选择。

已开发的发现服务包括Jini发现服务（见下面）、服务位置协议[Guttman 1999]、意图命名系统
668 [Adjie-Winoto et al.1999]、简单服务发现协议（它是通用即插即用项目[www.upnp.org]的核心）和安全服务发现服务[Czerwinski et al. 1999]。此外还有链路层发现服务，例如蓝牙。

服务注册/注销方法	说明
lease := register(address, attributes)	用给定的属性、在给定的地址注册服务，返回一个租期
refresh(lease)	刷新注册时返回的租期
deregister(lease)	删除在给定租期下注册的服务记录
查找服务的方法	
serviceSet := query(attributeSpecification)	返回一个注册服务的集合，其属性与给定的说明匹配

图16-3 发现服务的接口

设计一个发现服务所需处理的问题如下：

- 低能耗、合适的关联。理想情况下，合适的关联应该在无任何人为控制因素下进行。第一，查询操作（参见图16-3）返回的服务集合应该是合适的——它们应该是智能空间中精确符合查询的服务。第二，应该通过编程或者利用少量的人工输入选择满足用户需要的服务。
- 服务描述和查询语言。整体目标是匹配客户请求的服务。预先假设一种语言用于描述可用服务，另一种语言用于表示服务请求。查询和描述语言应该一致（或可翻译），并且它们的表达能力应该紧跟新设备和服务的发展。
- 智能空间特定的发现。我们需要一种机制以便让设备访问适合它们当前物理环境的发现服务的一个实例（或范围），这是一种设备不需要预先知道该服务的名字或地址的机制。实际上，

发现服务仅通过在与它交互的子网的有限范围内的组播才能与某一智能空间产生关联，稍后我们将说明这一点。

- 目录实现。逻辑上，发现服务的每个实例都包括可用服务的一个可查询的目录。有多种方式可实现这样一个目录，这些方法所需的网络带宽、所提供的服务发现的及时性和能量消耗都不相同。
- 服务易变性。易变系统的任何服务都要有效而妥善地处理一个客户的消失。发现服务作为客户服务，应恰当地处理服务消失。

669

作为通过发现而关联的一个例子，考虑一个偶然或第一次去主人家或宾馆的客人，他需要从笔记本电脑上打印一份文档。用户当然不能认为在他的笔记本电脑上配置了当地某打印机的名字，或者能猜测出它们的名字（例如，\\myrtle\titus和\\lionel\frederick）。与强制用户在访问时配置他们的机器相比，更好的方法是让笔记本电脑使用发现服务的*查询*调用来查找满足用户需要的可用的网络打印机集合。可以通过与用户的交互或者参考用户的偏好记录来选择某个打印机。

打印服务所要求的属性可能有说明它是“激光”还是“喷墨”，它是否提供彩色打印，以及它相对于用户的物理位置（例如，房间号）。

相应地，服务通过*注册*调用将它的地址和属性提供给发现服务。例如，打印机（或管理它的服务）可进行如下注册以便向发现服务提供它的地址和属性：

```
serviceAddress=http://www.hotelDuLac.com/services/printer57;resource-
Class=printer, type=laser, colour=yes, resolution=600dpi, location=room101
```

在运行时，在无手动配置情况下，自举访问本地目录服务的一般方法就是使用局域子网的可达范围。具体地说，在局域子网内向一个众所周知的IP组播地址组播（或广播）查询。所有需要访问发现服务的设备事先知道这个众所周知的IP组播地址。只基于网络可达范围的发现服务有时明确地称为*网络发现服务*。

注意，有些网络（例如蓝牙）使用跳频，所以不能在物理层同时与所有相邻设备通信。蓝牙使用“众所周知地址”的等价方法实现发现，即一个众所周知的跳频次序来实现。可发现设备频率周期比试图发现它们的设备慢，这样发送者（发现者）和接收者最后可以在频率上达成一致并建立通信。

实现发现服务时有几种设计选择，不同的设计对实现会有很大的影响。

第一个选择是发现服务应该由*目录服务器*实现，还是不需要服务器。目录服务器保存注册过的服务的描述，并对客户发出的服务查询做出响应。任何想使用本地目录服务器的组件（服务器或客户）发出一个组播请求来定位它，目录服务器以它的单播地址作为响应。之后，组件与目录服务器进行点对点通信——避免了采用组播通信时对无关设备的干扰。这种方法在提供了基础设施的智能空间中运作良好。目录服务经常运行于一个有稳定电源的机器。但是，在简单会议室这样的简单智能空间中，没有用于目录服务器的设施。原则上，可以从现场的设备中选择一台服务器（参见12.3节），但是这样的服务器可能自发地消失。这可能导致发现服务的客户端的实现的复杂性，因为客户不得不去适应一个变化的注册服务器。此外，在高易变系统中重新选举带来的负载可能很大。

670

另一种方法是*无服务器发现*，其中参与的设备通过协作实现一个分布式发现服务，以此代替目录服务器。至于分布式目录，分布式发现服务有两种主要的实现。在*推*模型中，服务定期组播（“广告”）它们的描述。客户监听这些组播并对它们发起查询，并可能缓存描述以备以后使用。在*拉*模型下，客户组播他们的查询。提供服务的设备比较它们的描述和这些查询是否匹配，只响应那些匹配的描述。如果没有响应，客户间隔一段时间后重复它们的查询。

推和拉两种模型都涉及带宽和能量的使用。每次设备发出一个组播消息，都令消耗带宽并且所有的监听设备都要消耗能量来接收消息。在纯推模型下，设备要定期广告它们的服务，以便客

户发现它们。但是如果没有客户需要服务，就会对带宽和能量造成浪费。而且客户等待服务的时间要与消耗的带宽和能量相权衡，而带宽和能量消耗随广告的频率的增加而增加。

在纯拉模型下，可用的服务一出现，客户就能够发现它。在给定的间隔内若没有发现需求就不会有组播浪费，客户可能收到多个响应，但只要一个响应就够了。在默认情况下，对频繁需要的服务、无法利用包含相同查询的请求来提高效率。

可以设计混合协议来解决上面的缺陷，习题16.2将涉及这个问题。

一个服务在它消失之前会调用注销函数（参见图16-3），但同样地它也可能自发地消失。根据目录实现的体系结构，服务易变性有多种处理方法。注册的服务消失后，目录服务器需要尽快知道这个情况，这样它才不会错误地发出它的描述。通常使用一个称为租期的通用机制实现这一目标。租期是服务器给客户临时分发一些资源，在租期过期之前，客户要进一步发生请求加以续租。如果客户续租失败（例如，使用图16-3的refresh函数），服务器就会收回（并且可能重新分配）资源。我们在5.2.6节作为Jini的一部分介绍了租期，而且DHCP服务器在分配IP地址时也使用了租期。只有当服务定期与目录服务器通信并更新项的租期时，目录服务器才会保存服务的注册。在这里，我们看到一个类似在及时性与带宽和能量消耗之间进行权衡的方法——租期越短，服务消失的通知就发得越快，但是需要的网络和能量资源越多。在无服务器体系结构中，不需要采取任何步骤（除了清除缓存服务的设备中的陈旧项信息），因为已经消失的服务不再广告自己，并且使用基于拉协议的客户只能发现当前的服务。

Jini Jini[Waldo 1999; Arnold et al.1999]是用于移动系统和无处不在系统的，它完全基于
671 Java——它假定Java虚拟机在所有的计算机中运行，允许它们之间通过RMI或事件（见第5章）进行通信并且能在必要时下载代码。接下来我们描述Jini的发现系统。

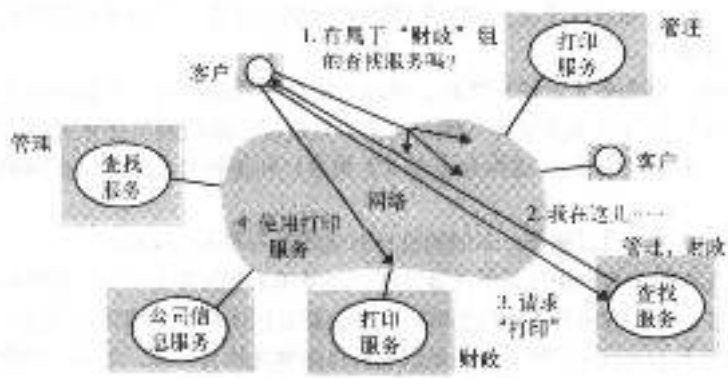


图16-4 Jini中的服务发现

Jini系统中与发现相关的组件是查找服务、Jini服务和Jini客户（参见图16-4）。查找服务实现了我们所说的发现服务，尽管Jini只为发现"查找服务"本身使用术语"发现"。查找服务允许Jini服务注册它们所提供的服务，并允许Jini客户请求与它们需求相匹配的服务。一个Jini服务（例如打印服务）可以在一个或多个查找服务上注册。Jini服务所提供的同时也是查找服务所存储的是提供该服务的对象以及该服务的属性。Jini客户为了查找符合它们需求的Jini服务，首先查询查找服务；如果找到匹配的服务，就从查找服务上下载提供访问该服务的一个对象。给客户请求提供的服务可以基于属性或者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。

当Jini客户或服务启动时，它发送一个请求给一个众所周知的IP组播地址。接收到该消息，并可以响应它的查找服务都返回自己的地址，使得请求者执行一个远程调用来查找或注册一个服务（在Jini中注册称为加入）。查找服务通过发送数据报到同一组播地址宣布它们自身的存在，Jini客

户和服务也可能监听组播地址以便它们发现新的查找服务。

从一个给定的Jini客户或服务的组播通信可能到达多个查找服务的实例。每个这样的服务实例配置了一个或多个组名字，例如“管理”、“财政”和“销售”，这些将作为范围标签。图16-4展示了一个Jini客户发现和使用打印机服务的例子。客户需要一个“财政”组中的查找服务，所以它组播一条带有该组名字的请求（图中的请求1）。只有一个绑定到“财政”组的查找服务（该服务也 672
绑定到“管理”组），该服务做出响应（2）。查找服务的响应包括它的地址，然后客户通过RMI直接与它通信来定位所有类型为“打印”的服务（3）。只有一个打印服务注册到“财政”组的查找服务上，因此返回一个用于访问该服务的对象。最后，客户利用返回的对象直接使用该打印服务（4）。图16-4显示了另一个位于“管理”组的打印服务，还有一个不绑定到任何组的公司信息服务（并且它可能向所有的查找服务注册了）。

关于网络发现服务的讨论 刚刚描述过的基于网络可达区域的发现服务——网络发现服务——采取了一些措施来解决关联问题。存在有效的目录实现，包括那些不依赖于基础设施的实现。在很多情况下，就计算和网络代价而言，通过子网可到达的客户和服务的数目是可管理的，所以规模通常不是问题。我们已经描述了用于处理系统易变性的措施。

然而，从边界原理的角度来看，网络发现服务引发了两个困难：子网的使用和服务描述方式的不完备。

子网近似于一个智能空间。第一，网络服务可能错误地包括并不属于智能空间的服务。例如考虑一个宾馆房间。基于无线射频（RF）信号（例如802.11或蓝牙）的传送通常会穿透墙壁到达另一个客房。效仿Jini例子，将服务在逻辑上按组划分——每个宾馆房间为一组。但是它避开了用户房间号如何成为发现服务的一个参数的问题。第二，网络发现可能错误地忽视了“在”智能空间中但超出子网的服务。Cooltown实例研究（见16.7.1节）说明了非电子实体（例如智能空间中要打印的文档）是如何与智能空间外的服务关联的。

此外，网络发现服务不总是能产生合适的关联，因为用于描述服务的语言可能在下面两个方面是不完备的。第一，发现可能是*脆弱的*：不同组织使用的服务描述词汇的微小变化都可能导致发现失败。例如，宾馆房间有个名为“Print”的服务，然而房客的笔记本电脑搜索的是名为“Printing”的服务。人类语言在词汇上的各种变化更容易加剧这个问题。第二，可能会*失去机会*——错失访问服务的机会。例如，在宾馆房间的墙壁上有个“数码相框”，它会以JPEG格式显示假日快照。房客的相机有无线连接，并且生成该格式的图像，但是它没有该服务的描述——它没有随着最近的发展进行更新。因此相机不能利用该服务。 673

16.2.2 物理关联

网络发现系统的缺陷在一定程度上可使用物理方法解决，尽管解决方案经常在要很大程度上需要人为的参与。目前已经开发的技术如下：

采用人工输入界定发现的范围 人向设备提供输入，以便设置发现范围。该技术的一个简单例子是输入或选择智能空间的标识符，例如房客的房间号。之后，设备可以将标识符作为一个额外服务“组”属性（就像在Jini中那样）。

采用传感技术和物理受限通道界定发现的范围 一个可能减少用户参与的方法是在他们的设备上使用传感器。例如，智能空间可能有一个标识符，在文档和空间表面——例如，显示在旅客房间的电视屏幕上——基于标识符进行编码的符号（称为*象形文字*）表示。房客使用他们的可拍照手机或其他图像设备来解码这种符号，接着设备以我们描述的用户直接输入的方式使用该结果标识符。为了使用具有卫星导航信号的智能空间，另一种可能的方法是使用传感器来得到智能空间的纬度和经度坐标位置，并且把那些坐标发送给一个众所周知广域服务，该服务返回本地发现服务的地址。然而，在卫星

导航不精确的情况下，如果旁边有其他的智能空间，该方法对智能空间的识别就可能不准确了。

另一种避免人工输入的技术是使用物理受限通道（参见16.5.2节）——它是一种通信通道，可以近似地认为它只作用于智能空间所在的物理范围内。例如，在客房内，电视可能正在以低音播放背景音乐，房间标识符的数字编码被叠加为音乐信号的一个听不见的微扰[Madhavapeddy et al. 2003]；房间内也可能有一个红外线发射器（一个信标）用来传送标识符[Kindberg et al. 2002a]。这两种通道遇到房间边界的材料时会极大地衰减（假设门和窗是关闭的）。

直接关联 最后我们考虑的技术是让人使用物理机制直接关联两个设备，而不使用发现服务。通常，这种情况下的设备只提供可供一个人或一小部分人选择的服务。在下面提到的每种技术中，用户都能够使用他们携带的设备获得“目标”设备的网络地址（例如蓝牙或IP地址）。

地址感知：使用一种设备直接感知目标设备的网络地址。这可能包括：读取设备的象形文字，形成网络地址编码；或者将一个设备靠近另一个设备，并使用短程无线通道读取它的地址。这种短程通道的例子有近距离通信[www.nfc-forum.org]（一种双向无线电通信标准，它拥有多个可选的较短的作用范围，但是通常只能在3厘米之内变化）和短程的红外线传输。

674 *物理触发器*：使用一个物理触发器让目标设备发送它的地址。例如，将一种数字调制的激光束（另一种物理受限通道）照射到目标设备上[Patel and Abowd 2003]，就能将它的地址传送给目标设备，目标设备用它的地址加以响应。

时间或物理相关性：使用时间上或物理上相关的触发器来关联设备。SWAP-CA规范[SWAP-CA 2002]用于引入协议的家庭环境内的无线网络，它有时也被称为两按钮协议，让人将两个无线设备关联起来。每个设备监听一个众所周知的组播地址。用户几乎同时按下各自设备上的按钮，同时设备向组播地址发送它们的网络地址。由于不可能在同一子网、同一时刻进行该协议的下一轮，因此设备与在按钮按下这一小段时间间隔内到达的任何地址进行关联。有一个有意思但不太实际的两按钮协议的实现方法[Holmquist et al. 2001]，用户一只手握着两个设备并同时摇动它们。每个设备有一个加速计以感知它的运动状态。设备记录摇动模式，从中计算出一个标识符，并且将标识符连同它的单播地址组播到一个众所周知组播地址。只有精确体验该加速模式的两个设备——并在直接通信范围内——才会识别出彼此的标识符并由此知道双方的地址。

16.2.3 小结和前景

本节描述了易变系统组件的关联问题，并提出从网络发现到多人监控技术等方法来试图解决该问题。移动系统和无处不在系统引发了特有的难题，因为它们是与我们日常的、凌乱的物理世界空间集成的（例如家庭和办公室），这使得研究解决方案很困难。人们在考虑一个特殊智能空间内部有什么和外部有什么时容易在头脑中有很多的领域和管理方面的考虑。边界原理提出的解决关联问题的方法需要在某种用户可以接受的程度上匹配底层的物理空间。我们看到，由于网络发现系统的缺陷，通常需要一定程度上的人工监控。Cooltown实例研究（参见16.7节）描述了一种有人参与的模型。

在关联问题的解决方案中，由于地球通常被分成可管理大小的智能空间，因此我们很大程度上忽略了规模因素。但是，有一些针对发现服务的研究——毕竟，有些应用可能将整个地球看作一个智能空间。例如INS/Twine[Balazinska et al, 2002]，它将目录数据划分到一系列对等服务器上。

16.3 互操作

我们已经描述了易变系统的两个或多个组件关联的方式，现在讨论它们如何互操作。组件基
675 于它们中的一个或两个拥有的某些属性或数据进行关联。下面的问题是：它们使用什么协议进行通信？在较高层次上，什么样的编程模型最适合它们之间的交互？本节将解决上述问题。

第4、5章描述了用于互操作的模型，包括各种形式的进程间通信、方法调用和过程调用。其中一些模型包含的隐含假设是互操作组件被设计为在一个特定系统或应用中共同工作，还有互操作组件集合的改变要么是长期配置问题，要么是一个偶尔被处理的运行时错误条件。但是那些假设在移动系统和无处不在系统中是不成立的。幸运的是，正如本节我们阐述的，第4章和第5章中介绍的互操作的某些方法（除一些新方法外）非常适合那些易变系统。

理想情况下，移动系统或无处不在系统中的组件可以与变化的服务类型关联，而不只是与相同类型服务的不同的实例集合关联。也就是说，最好避免前一节描述的“失去机会”问题。例如，数码相机不能将它的图像发送到一个数字相框，因为它不能与相框的图像使用服务互操作。

从另一个角度看，无处不在计算和移动计算的一个目标是组件应该有机会与功能匹配组件互操作，即使后者处于一个不同于最初开发它的智能空间中。这需要软件开发者之间具有全局协定。给定达成协定所需的努力，最好能将需要达成一致的内容减到最少。

易变互操作的最主要的困难是软件接口的不兼容。例如，如果数码相机希望调用操作pushImage，但是在数字相框接口中没有这样的操作，那么它们就不能互操作——至少，不能直接互操作。

该问题有两种解决方法。第一种方法是允许接口异构，但是要适应彼此的接口。例如，如果数字相框的sendImage操作与pushImage操作有相同的参数和语义，那么就能构造一个组件作为数字相框的代理，将相机的pushImage调用转换成相框的sendImage调用。

然而，这种方法很难实现。通常，操作的语义可能随语法而变化，并且解决语义不兼容性一般是很困难的并且容易造成错误。该问题的规模是：如果有N个接口，则潜在的适配器要有N^2个，并且随时间的增长可能需要创建更多的接口。此外，在易变系统中组件重新关联时，存在它们如何获得合适的接口适配器的问题。组件（或拥有它们的设备）不能预装载所有可能的N^2个适配器，所以要在运行时确定并装载正确的适配器。尽管有上述困难，但仍然有关于如何在实际中解决接口适配问题的研究。例如，Ponnekanti和Fox[2004]。

另一种关于互操作的方法是限制接口，使其在语法上尽可能像一种组件类型一样一致。这听起来可能不现实，但实际上它已经广泛、成功地应用了几十年了。最简单的例子是UNIX中的管道。管 676
道只有两个操作：read和write，用于管道两端组件（进程）间的数据传送。多年来，UNIX程序员创建了许多程序能从管道中读数据/或向管道中写数据，或二者兼而有之。因为这些程序使用标准接口和通用的文本处理功能，所以其中任何一个程序的输出都能作为另一个程序的输入；用户和程序员发现了许多有用的方法用于合并程序——那些可独立开发的、不涉及其他程序的特定功能的程序。

一个已经比较成功的、通过一个固定接口达到高度互操作的例子是Web。HTTP规约（见4.4节）定义的方法集合规模很小并且很固定。通常，Web客户只使用GET和POST操作访问Web服务器。使用固定接口的好处是能通过一个相对稳定的软件（通常是浏览器）与服务集合互操作。服务之间变化的是交换的内容的类型和数值，以及服务器的处理语义。但是每个交互依然是一个GET或POST操作。

16.3.1 易变系统的面向数据编程

我们称使用一个不变的服务接口（如UNIX管道和Web）的系统为*面向数据*（或*面向内容*）的系统。选择这个术语是为了与*面向对象*区别开来。面向数据系统中的组件可以被知道固定接口的其他组件调用。另一方面，对象或过程的集合可能有一个变化多样的接口集合，并且只能被那些知道它的特定接口的组件调用。分布和开发一个不确定的特殊接口定义比发布和使用一个接口规约（比如HTTP规约）更困难。这就解释了为什么我们熟知的广泛使用的异构分布式系统是Web，而不是类似的范围有限的，比方说，CORBA对象集合。

但是要在面向数据系统的灵活性与健壮性之间做出权衡。两个组件的互操作不总是有意义的，并且很难用程序核查兼容性。在面向对象或面向过程系统中，程序至少能够核查它们的特有的接口签名匹配。但是面向数据组件只能通过验证发送给它的数据类型来增强兼容性。这种验证要么通过作为元数据提供的（例如Web内容的MIME类型）标准数据类型描述符，要么通过检查传递给它的数据值（例如，JPEG类型的数据有一个可识别的信息头）来完成。

我们现在分析几种编程模型，由于它们具有面向数据互操作特性，所以适用于易变系统。首先介绍用于间接关联组件间互操作的两种模型：事件系统和元组空间。之后我们将描述用于直接关联的设备间互操作的两种设计：JetSend和Speakeasy。

事件系统 我们在5.4节中介绍了事件系统[Bates et al.1996]。事件系统提供事件服务的实例。
677 每个系统提供一个固定的、通用的接口，名为*发布者*的组件通过该接口发布称为事件的结构化数据，同时，称为*订阅者*的组件接收事件。每个事件服务与某个物理的或逻辑的事件传送范围相关联。订阅者只接收（“处理”）这样的事件：(1) 在同一个事件服务中发布的，(2) 匹配它们对所感兴趣事件的注册说明。

对于发布和处理组件在易变系统中或在易变系统间移动时所经历的变化，事件是一个很自然的编程范型。事件可以被组织以说明事情的新状态，比如，设备位置的改变。最近一个用于无处不在计算的使用事件的例子系统是one.world[Grimm 2004]。但是事件在开发的早期就用于无处不在系统了。在Active Badge系统[Harter and Hopper 1994]中，应用可以订阅用户移动时发生的位置改变事件。位置事件也提出了检测同时或相邻发生的事件（也称为*合成事件*）模型的问题。例如，考虑检测两个用户相邻时的定位问题，所知的仅是单个用户何时进入或离开一个特定位置。位置系统并不亲自检测这些事件：有必要制定规则，从基本事件（比如“到达（用户，位置，时间）”和“离开（用户，位置，时间）”）的角度来说明合成事件（比如协同定位）何时发生。

尽管已给定事件的发布、订阅和处理接口（事件系统之间的变化相对很小），但是发布者和订阅者只有在对使用的事件服务（可能有许多实例）和事件的类型、属性（它们的语法和语义）达成一致时，才能正确地进行互操作。因此，事件系统转换了而不是解决了无处不在互操作问题。对于给定的组件，要在种类繁多的智能空间中实现互操作，需要将事件类型标准化，并且理想中的事件应该用一种独立于编程语言的标记语言（比如XML）描述。

另一方面，事件产生者和消费者不需要识别彼此。这在易变系统中是一个优势，在该系统中跟踪其他组件的位置是很困难的。两个组件通过发布和订阅匹配的事件，以及通过对事件传递范围达成一致来进行通信——换句话说，它们间接关联。

在移动系统和无处不在系统中，事件传递范围本身就是一个有趣的话题。随着服务发现的应用，出现了事件服务的范围如何与智能空间的物理范围联系的问题。该问题可参见习题16.7。

元组空间 同事件系统类似，元组空间也是一个成熟的编程范型，而且它在易变系统中已得到应用。组件使用一个固定的、通用的接口来增加或检索元组空间中称为元组的结构化数据（见18.2.1节）。元组空间系统允许交换应用特定的元组，并且关联和互操作的基础是组件对元组结构和元组所包含数值的约定。

例如，数码相机可以发现本地智能空间（如一个宾馆房间）的元组空间，并且使用元组比如：
678 *<'The leaning tower', 'image/jpeg',<jpeg data>>*

将它的图像放到元组空间中。

相机软件的设计者有一个用于将图像以一定格式放置其中的元组空间的模型，而没有那些图像的特殊处理模型。

相应的，图像使用设备（比如数字相框）可以通过编程发现它的本地元组空间，并试图从带有如下模板格式的元组中检索，其中“*”代表通配符：

<*, 'image/jpeg', *>

相机的元组匹配相框要求的模板——它有三个域，并且第二个域包括要求的MIME类型字符串。相框将检索相机的元组并且能显示图像和相关的标题。另一个例子是用户可以激活一个打印机来打印元组空间的图像。

有几种特别为移动系统和无处不在系统开发的基于元组空间的编程系统。尽管名为事件堆[Johanson and Fox 2004]，但它是为称为iRoom的一类智能空间开发的基于元组的编程系统。iRoom中包括多个大显示器和其他基础设备。对于每个iRoom，有一个对应的事件堆，在该事件堆中，iRoom中的组件（包括带入房间内的移动设备上的组件）可以发现或配置使用。组件通过事件堆交换元组来实现互操作，并且事件堆提供一定程度的间接关联以便于设备间的动态关联。一个例子是，放在iRoom中的一个远程控制设备可以与不同的显示器动态地关联。例如，一段视频可以在几个大显示器中的任何一个上显示。当用户按下远程控制的“暂停”按钮时，控制器在事件堆中放置一个“暂停”元组。显示视频的设备编程来查找并检索“暂停”元组，进而加以响应。远程控制器还可以以完全相同的方式与一个音频输出设备一起工作，不需要重新编程。

LIME系统（移动环境中的Linda）[Murphy et al.2001]是作为移动系统的编程模型而开发的。在LIME中，参与的设备拥有元组空间，并且与基础设施无关。每个设备拥有它自己的元组空间。当LIME拥有的设备关联时，LIME共享各自的元组空间，对共享元组空间的聚合操作形成了元组集合的并集。例如，这可用于服务发现。请求服务的组件能够编程得到描述它所需服务实例的元组；实现相应服务的设备能够编程在它的本地元组空间中放置描述性元组。当二者连接时，LIME将建立匹配，并且为潜在客户提供服务的详细资料。

虽然LIME模型易于描述，但是面对任意的连接和断链实现合适的一致性语义是很困难的。LIME的实现者做出了有争议的、不现实的假设来简化他们的设计，包括组播连通性在元组空间被聚集的设备间保持一致；到聚集集合的连接和从聚集集合的断开是串行化的、有序的。

事件系统和元组空间的比较 如果我们把“事件”看作“元组”，把“兴趣的说明”看作“元组匹配模板”，那么两种互操作模型是一致的。两者都提供一定程度的间接性，这对易变系统是有 679
用的，因为默认情况下产生或使用事件或元组的组件的标识符对彼此是透明的。这样组件集合就可以透明地改变了。然而，也有重要的区别。第一，事件模型是绝对异步的，而元组空间系统支持同步操作以检索匹配的元组。同步操作的编程比较容易。另一方面，期望某个组件（例如，动态遇到的图像产生设备）最终会提供匹配的元组是错误的想法，因为断链在任何时候都可能发生。

第二个重要的区别是事件和元组的生命期。默认情况下，事件在它在发布者和订阅者间的传播后消亡。然而，元组空间的元组可能比放置它的组件和任何读取（除了破坏性的使用）它的组件的生命期长。这种持久性是一种优点。例如，用户相机的电池可能在他将图像上传到宾馆房间的元组空间后，但在将它们分配给另一个设备前用完。同时，持久性也可能是一种缺点，如果一组不可控制的设备将元组放到空间中，但因为使用它们的组件已经断链而不会被使用，这将会产生什么样的结果呢？这种空间中的元组集合将变得不可控制。没有易变组件集合的全局知识，不可能确定哪些元组是无用的。

事件堆的设计者认识到了iRoom的持久性问题。他们选择让元组在事件堆中待一段时间后到期（也就是垃圾回收），特定时间通常与人的交互时间间隔一致。例如，当用户第二天试图播放视频时，该机制阻止了一个来自远程控制器的未使用的“暂停”事件。

设备的直接互操作 前面的编程模型用于间接关联组件间的互操作。JetSend和Speakeasy是用于由人为因素导致直接关联的两个设备间的互操作的系统。

JetSend：JetSend协议[Williams 1998]用于应用（比如照相机、打印机、扫描仪和电视）间的

交互。JetSend明确地设计为面向数据的，这样应用就不需要根据将要交互的特定设备装载特殊的驱动程序了。例如，JetSend相机能发送图像到JetSend图像使用设备，比如打印机或电视，而不用考虑使用者的特殊功能。相连接的JetSend设备间的主要通用操作是同步一方呈现给另一方的状态。这意味着以设备协商的格式传送状态。例如，图像产生设备（比如扫描仪）可通用使用来自JPEG格式的产生者的图像与图像使用设备（比如数字相框）同步。JPEG格式是从产生者可提供的几种图像格式中挑选出来的。同一个扫描仪可能在使用一种不同的图像格式时与一个电视同步。

JetSend的设计者认识到，他们的同步操作只对异构设备间的简单操作（数据传递）有利。它
680 避开了如何在特定设备间达到更复杂的交互的问题。例如，在传递要打印的图像时，如何在单色和彩色间做出选择？假定源设备没有特定打印机的驱动程序。并且对于该设备不可能编程得到它可能要连接的任意设备（包括未发明的设备）的语义的先验知识。JetSend对该问题的回答是使用目标设备指定的、在源设备（比方说，相机）上显示的用户界面，通过人为选择目标设备（比方说，打印机）的特定功能来实现。这就是说，用户通过他们的浏览器与高度异构的服务交互时，在Web上如何发生互操作：每个服务将它的接口以标记脚本的形式发送给浏览器，浏览器以通用窗口小部件集合的形式将服务呈现给用户，而不需要关于服务特有语义的知识。Web服务（见第19章）试图以程序代替人（甚至在一些复杂的交互中也是如此）。

Speakeasy：Speakeasy项目[Edwards et al.2002]采用了与JetSend相同的设计原理来实现设备间互操作，但有一点不同：它使用了移动代码。使用移动代码有两个原因。第一，设备（比如打印机）能够将任何用户接口发送给另一个设备（比如PDA）的用户。用户接口的移动代码的实现能够执行本地处理（比如输入验证），并且它能够提供在用户界面上不可用的交互模式，该模式必须要在标记语言中说明。然而，与该优点相对的是，必须要设置执行移动代码的安全性，移动代码需要复杂的保护机制来预防特洛伊木马，并且为了处理有更多限制的标记脚本，要有在虚拟机上运行移动代码的许可。

在设备互操作中使用移动代码的第二个动机是优化数据传送。虽然Speakeasy的移动代码必须在主机设备的API限制内运作，但它可以与发送它的远程设备进行任意交互。因此，移动代码能够为传送内容（内容类型是特定的）实现优化的协议——例如，视频可能在传送前被压缩。比较而言，JetSend只能使用预定义的内容传送协议。

16.3.2 间接关联和软状态

当服务的资源能提供足够高的可用性（比如基础设施服务）时，组件与它的关联就有意义了。也就是说，组件可以获得服务的地址。当组件随后使用该地址与服务互操作时（比方说，关联后10分钟），它们仍然可以希望它是可达的并且是可响应的。然而，通常情况下，系统易变性使得依赖某特定组件提供的服务是不可能的，因为该组件可以在任何时候离开或失效。从这个区别中得到的教训是：有必要告诉程序员哪些服务是高可用的，哪些服务是易变的。此外，为了处理易变性，需要给他们提供不依赖特定组件的编程技术。

681 以上几种面向数据编程系统的例子涉及间接关联和匿名关联。特别是，通过事件系统或元组空间互操作的组件不需要知道彼此的名字或地址。只要事件服务或元组空间存在，单个组件可以进入、离开，或被替代。这需要小心维护整个系统的正确操作，但至少组件的编写者不需要管理与经常消失节点的单独关联。

使用间接关联的客户-服务器系统的一个例子是意图名字系统（Intentional Name System，INS）[Adjie-Winoto et al.1999]。组件发起请求，说明所需服务的属性、要调用的操作和参数。组件不需要说明所需服务实例的名字或地址，因为INS自动将操作和参数路由到一个合适的——比如，本地的——匹配所需属性的服务实例。因为指向相同属性说明的连续操作可能由不同的服务器组件处理，

所以INS假设那些服务器是无状态的，或者使用第15章描述的某种技术复制它们的状态。

这导致了一个常见问题：在易变系统中程序员如何设法管理状态？第15章的复制技术允许资源的冗余，在易变系统中资源可能不可用——至少，不是持续可用的。复制技术也导致了额外通信，从而导致相关的能量消耗和性能降低，所以额外通信可能是不实际的。

Lamport的“Part-time Parliament”算法[1998]提供了一种忽略易变性而达到分布式一致的方法——假定参与的进程有规律地、独立地消失和重新出现。不过，该算法依赖于每个进程访问自己的持久性存储。

相比之下，一些实现使用软状态来提供不太严格但有用的一致性保证，甚至在没有持续可用的持久性存储的情况下。Clark[Clark 1998]引入了软状态的概念来作为一种管理因特网路由的配置方式，而且不考虑故障情况。路由器集合是一个易变系统，即使系统中没有路由器能够保证总是可用，但是系统也必须能持续运作。软状态的定义一直处于争论之中[Raman and McCanne 1999]，但从广义上讲，它是提供提示的数据（也就是说，它提供的数据可能是过时的，并且从严格意义上讲不应该被依赖），而且，最重要的是，软状态的源会自动更新它。一些发现系统（见16.2节）给出了软状态在管理服务注册项集合上的应用。第一，项只是提示——可能有服务的一个已经消失的项。第二，项通过服务的组播自动地更新——增加新项和保持现有的项。

16.3.3 小结和前景

本节描述了易变系统组件间的互操作模型。如果每个智能空间都开发它自己的编程接口，那么移动性的好处就无法体现。如果一个组件不是源于给定的智能空间而是移动到该空间的，那么它与智能空间内服务互操作的唯一方法将是使它的接口自发地适应新环境的接口。实现这个目标需要非常复杂的运行支持，除实验中已有几个例子外，这仍是不现实的。 682

通过上面几个例子的描述可知，另一种解决方法是使用面向数据编程。一方面，Web显示了该范型的可扩展性和极大的可应用性。另一方面，没有“尚方宝剑”能够解决易变系统互操作的所有问题。面向数据系统是在接口的函数集合上达成一致而不是在作为参数传递给那些函数的数据的类型上达成一致。尽管XML（见4.3.3节）使得数据能够“自我描述”，所以有时用作一种便于数据互操作的方式，但是实际上它仅提供了表示结构和词汇的框架。XML本身对什么是语义问题没有贡献。有些作者认为“语义Web”[www.w3.org XX]将在未来的解决方案中占据一席之地。

16.4 感知和上下文敏感

前面几节重点介绍了移动系统和普适系统的易变性方面。本节将着重介绍系统的其他特征：与物理世界集成的特征。特别地，将会考虑用于处理从传感器收集的数据的体系结构和对（感知的）物理环境作出响应的上下文敏感系统。我们还将详细论述位置感知——一个重要的物理参数。

因为我们考虑的用户和设备是经常移动的，并且物理世界为跨越时空的大量交互提供了不同的机会，所以它们的物理环境通常是系统行为的决定性因素。Active Badge系统提供了一个例子：在移动电话出现之前，用户的位置（也就是他们穿戴的标记的位置）用于识别他们打出的电话应该路由到哪个电话机上（Want et al. 1992）。汽车的上下文敏感刹车系统应该能根据路面是否覆盖着冰来调整它的行为。个人设备应该能够自动利用在它的环境内探测到的资源，比如一个大显示器。

实体（人、位置或物体，不论是否是电子的）的上下文是与系统行为相关的物理环境的一个方面。它包括相当简单的值，比如位置、时间、温度；关联的用户的标识符（例如，操作设备的用户或附近的用户）；物品的存在和状态（比如另一个设备，如显示器）。可以通过规则编撰和作用于上下文，比如“如果用户是Fred并且他在IQ实验室的会议室，同时如果在他1米范围内有一个显示器，那么就将设备上的信息显示在显示器上——除非有非IQ实验室的员工在场”。上下文也包

括比较复杂的属性，比如用户的活动。例如，上下文敏感手机决定是否要响铃时，需要以下问题
683 的答案：用户正在电影院看电影，还是正在放映前与他们的朋友聊天？

16.4.1 传感器

上下文数值的确定依赖于传感器，传感器是用来测量上下文数值的硬件和/或软件的结合物。下面给出一些例子：

位置、速度和方位：卫星导航（例如，GPS）装置提供全球坐标和速度；加速计用来监测运动；磁力计和陀螺仪提供方位数据。

周围环境：温度计；测量光线强度的传感器；感受声音强度的麦克风。

存在：用来测量物理负载的传感器，例如探测到人坐在椅子上或走过某块地板；读取靠近它们的标签的电子标识符的设备，比如RFID（无线射频识别）阅读器[Want 2004]，或红外线阅读器（比如那些用于感知Active Badge的）；用于检测计算机的按钮按下的软件。

以上分类只有作为用于某种目的的传感器的例子时才有意义。一个给定的传感器可能有多种用途。例如，在会议室用麦克风可监测人的存在；可通过在已知地点检测对象的主动标记来确定它的位置。

传感器的一个很重要的方面是它的误差模型。任何传感器产生的数值都带有一定程度的误差。有些传感器（例如温度计）可以通过精密的制造将误差控制在一个已知的可容忍的（如Gaussian分布）范围内。其他传感器，比如卫星导航装置，有一个依赖于当时的环境的复杂的误差模型。第一，在一定环境下，它们可能根本无法产生数值。卫星导航装置依赖于当时可见的卫星集合。它们在建筑物内可能根本无法工作，建筑物的墙壁可以大大削弱卫星信号。第二，装置位置的计算依赖于一些动态因素，包括卫星位置、附近的障碍物和电离层。即使在建筑物外，在不同的时间，装置通常会为同一位置给出不同的数值，这些数值只是当前精确度的一个最好估计。在遮蔽或反射无线电信号的建筑物或其他高物体的附近，只有当可见卫星足够多时才能产生一个读数，但是精确度可能很低甚至读数可能是完全虚假的。

用来陈述传感器的错误行为的一个有用的方法是引用精确度，说明达到度量的一个制定的比例。例如，“在给定区域内，在90%的测量中，卫星导航装置的精确度在10米以内”。另一种方法是用*置信值*进行特定测量——依据测量中遇到的不确定性选择的一个数值（通常在0～1间）。

16.4.2 感知体系结构

Salber et al. [1999]认为在设计上下文敏感系统中有四个功能性方面的挑战：

684 *异质传感器的集成* 上下文敏感计算需要的一些传感器的结构和编程接口是不同的。可能需要特殊的知识才能在感兴趣的物理场景中正确的部署它们（例如，要测量用户的胳膊姿势，加速计应该放在哪？），并且可能产生系统问题，比如标准操作系统的驱动程序的可用性。

传感器数据的提取 应用程序要求对上下文属性进行抽象，以避免涉及单个传感器的特殊点。但问题是，即使用于相似用途的传感器也会提供不同的原始数据。例如，一个给定的位置可能被卫星导航传感器感知成纬度/经度对，或者被附近的红外线源感知成“Joe's Café”字符串。应用程序所需要的可能是两者之一或两者都需要，或两者都不需要。需要对上下文属性的含义达成一致，需要软件从传感器原始数值中推断出这些属性。

传感器输出可能需要结合 可靠地感知一种现象可能需要结合来自多个误差检测源的数据。例如，检测人的出现可能需要：麦克风（用来检测声音，但附近的声音会产生干扰）、地板压力传感器（用来监测人的活动，但很难区分不同用户的模式）以及录像机（用来监测人的形体，但很难区分面部特征）。*传感器融合*是指结合传感器源以减少错误。同样的，应用为了收集操作需要的多种上下文属性，可能需要不同类型的传感器输出。例如，一个上下文敏感PDA为了决定是否将它

的数据投影到附近的一个显示器上，需要来自不同传感器源的数据，包括监测现场有谁和什么设备的传感器以及一个或多个感知位置的传感器。

上下文是动态的 上下文敏感应用通常需要对上下文的变化做出响应，并且不是只读取上下文的一个快照。例如，如果非员工进入或者如果Fred（设备的主人）离开房间，上下文敏感PDA必须清除房间里显示器上的数据。

研究者设计出了各种软件体系结构以支持上下文敏感应用，同时处理上面提到的一些或所有问题。我们将给出一些体系结构的例子，它们用于传感器几乎是已知的和静态的情况，或者用于确定来自易变传感器集合的上下文属性——此时非功能性需求（比如能量节约）也变得很重要。

基础结构中的感知 Active Badge传感器最初部署在英国剑桥Olivetti研究室，位于建筑物内已知的、固定的地点。最初的上下文敏感应用之一是电话接线员帮助系统。如果有人打电话说让Roy接电话，接线员就会在屏幕上查找Roy的房间位置，随后将电话转到一个最合适的范围。系统从上次感知Roy穿戴的标记的信息中确定Roy的位置，并将信息显示给接线员。Olivetti研究室和Xerox PARC进一步精化了用于处理Active badge数据和其他上下文数据的系统。Harter 和Hopper[1994]描述了一种用于处理位置事件的完整的系统。Schilit 等[1994]也描述了一种能处理Active badge感知事件的系统，通过这些事件，它们调用上下文触发动作。例如，下面的说明表示：

685

Coffee Kitchen arriving play-v.50 / soundstrooster.au

当感知到标记靠近安装在厨房的咖啡机上的传感器时，就会发出一种声音。

Context工具包[Salber et al. 1999]是比那些基于特殊技术（如Active Badges）更通用的上下文敏感应用使用的一种系统体系结构。正是Context工具包的设计者们阐述了上面列出的上下文敏感系统面临的四个挑战。他们的体系结构遵循了如何通过可重用窗口部件库构成图形用户界面的模型。这种方式对应用的开发人员隐藏了对底层硬件处理的大多数细节和大部分的交互管理。Context工具包定义了上下文窗口部件。那些可重用软件组件给出了某些上下文属性的抽象表示，同时隐藏了实际使用的传感器的复杂性。例如，图16-5显示了IdentityPresence窗口部件的接口。它通过轮询窗口部件将上下文属性提供给软件，并在上下文信息改变时（用户到达或离开）发起调用。如上所述，可以从给定实现中的多个传感器的组合的任何一个中得到人员出现信息；抽象使得应用程序编写者忽略了那些细节。

属性（通过轮询访问）	说明
Location	窗口部件正在监控的位置
Identity	上一个被感知的用户ID
Timestamp	上一次到达的时间
回调	
PersonArrives(location, identity, timestamp)	用户到达时触发
PersonLeaves(location, identity, timestamp)	用户离开时触发

图16-5 Context工具包的IdentityPresence窗口部件类

部件是从分布式组件中构建的。生成器从传感器（比如地板压力传感器）中得到原始数据，并且将数据提供给窗口部件。窗口部件使用解释器服务，该服务从生成器的原始数据中提取上下文属性，得到较高级的值，比如，从人的不同脚步声中判断出人的身份。最后，由服务器部件提供更高级抽象，该抽象是通过从其他小部件收集、存储并解释上下文属性得到的。例如，建筑物的一个PersonFinder部件能够由该建筑物内各个房间的IdentityPresence部件构成（见图16-6）。

686 IdentityPresence部件又可以使用地板压力传感器提供的脚步解释或视频捕捉的脸部识别来实现。PersonFinder窗口部件为应用程序编写者封装了建筑物的复杂性。

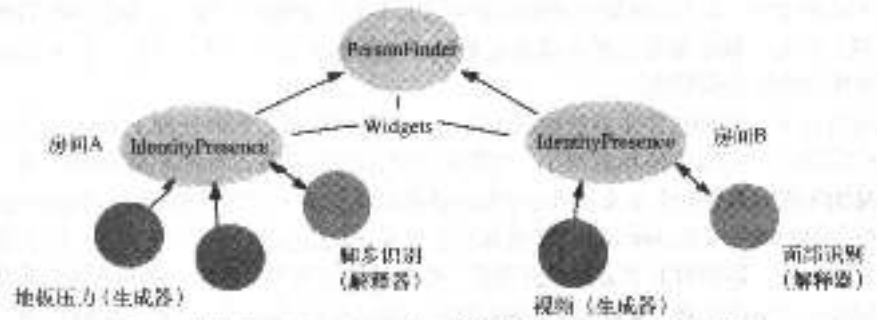


图16-6 使用IdentityPresence窗口部件构建PersonFinder窗口部件

关于上面提到的由Context工具包的设计者提出的四个挑战，它们的体系结构容纳了各种类型的传感器，它从传感器的原始数据中获得抽象的上下文属性，上下文敏感应用程序通过轮询或回调的方式来获知上下文的变化，但该工具包的实用性有限。它并没有帮助用户和程序员集成异质的传感器，也没有为特定情况解决任何解释和合成进程所固有的难题。

无线传感器网络 我们讨论过了传感器集合相对稳定的应用的体系结构。例如，传感器安装在建筑物的房间内，通常有外部电源和有线网络连接。我们现在转而研究由传感器集合形成一个易变系统的情况。无线传感器网络包括很多（通常有大量的）小的、低成本设备或节点，每一个节点都带有用于感知、计算和无线通信的设施[Culler et al.2004]。它是自组织网络的一个特例：节点在物理上几乎是随意安排的，但它们可以通过节点间的物理多跳进行通信。这些网络的一个重要的设计目标是在无全局控制条件下运作，每个节点通过发现它的无线邻居和与它们的通信来自举它自己。3.5.2节描述了802.11网络中的自组织配置，但是在这里我们更关注低功率技术，比如ZigBee（IEEE 802.15.4）。

节点不在单跳中与其他节点进行通信，而只与相邻的节点直接通信。原因是，无线通信在能量消耗上是很大的（与无线电射程的平方成正比）。另一个限制单个无线电射程的主要原因是减少网络竞争。

无线传感网络被设计成加入到一个已存在的自然或构建的环境中，并且不依赖环境运作（即不依赖基础设施）。由于它们无线电波和感知范围有限，节点必须安装得足够密集从而使得能够在
687 任意两节点间多跳通信和感知重大的现象。

例如，考虑设备遍布整个森林，它们的任务是监控火灾和其他环境条件，比如动物出现。这些节点非常像16.1.1节介绍的设备。它们各自有附属的传感器，例如，用于检测温度、声音和亮度的传感器，它们使用电池组。此外，它们与其他设备以对等方式通过短程无线电通信进行交流。易变性源于设备会因为电池耗尽或者事故（比如火灾）而停止运转；并且他们的连通性可能由于节点故障（节点在其他节点间传递包）或影响无线电传播的环境条件而改变。

另一个例子是用于监控交通和路面条件的节点安装在车辆上的什么位置。观察到不佳情况的节点可以将该消息通过车辆上的节点传送。由于连通性足够充分，因此该系统能够警告出现在该问题前方附近的驾驶者。在这里，出现易变性主要是由于节点的运动，运动会迅速地改变每个节点与其他节点的连通性。这是移动自组织网络的一个例子。

通常，无线传感网络专用于应用特定目的，等同于检测某种事件，即感兴趣的情况（比如火灾或糟糕的路面情况）。该网络中至少包括一台功能强大的设备，称为汇节点，用于与响应警报的常规系统进行远距离通信，例如发生火灾时的呼叫紧急服务。

构造传感器软件体系结构的一种方法是通过从高层中分离出网络层，并视它们类似于传统网络。特别地，当它们动态地发现它们自己已经通过直接无线电链接进行连接，并且每个节点都能够作为与其他节点通信的路由器时，就可以对节点图采用已有的路由算法。自适应路由试图适应网络的易变性，目前它已是研究的热点，Milanovic等[2004]提供了某些技术的概述。

然而，对网络层的关心引发了下面的问题。第一，自适应路由算法不必考虑低能量（和带宽）消耗。第二，易变性影响了传统的网络层以上层的一些假设。无线传感网络软件体系结构的另一种首要方法由两个主要需求驱动：能量节约和在易变性条件下持续运作。这两个因素导致了三个主要体系结构特征：网络内处理。容中断网络和面向数据编程模型。

网络内处理：无线通信不但能量消耗过高，而且与处理相比代价也很高。Pottie和Kaiser[2000]计算了能量消耗，发现无线电将1K比特数据传输100米所使用的能量（3J）可以令一台通用处理器执行300万条指令。因此，通常情况下，处理优于通信：最好花费几个处理器周期来决定是否仍需要通信，而不是盲目地传送感知到的数据。毫无疑问，这是传感器网络的节点有处理能力的原因——否则，它们可能只由将感知的数值发送到根节点处理的感知通信模块组成。 688

网络内处理是指在传感器网络内处理。也就是说，在网络节点上处理。传感器网络中的节点执行如下任务：求来自邻近节点的数值的总和或平均值，从而为一个区域而不是单个传感器检查数值；过滤掉不感兴趣的或重复的数据；检查数据以检测警报；根据感知的数据接通或切断传感器。例如，如果低功率光线传感器指示可能有动物出现（由于影子的投射），那么影子投射的位置附近的节点就会接通它们的高功率传感器，比如用于探测动物声音的麦克风。在该方案中，为了节约能量，应该在其他情况下关闭麦克风。

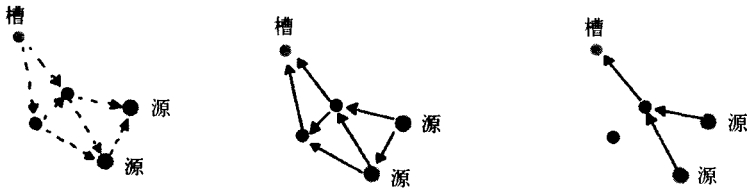
容中断网络：端对端的争论（见2.2.1节）是分布式系统的一个重要的体系结构方面的原则。然而，在易变性系统（比如传感器网络）中，可能没有持续存在时间足够长的端对端路径去实现一个操作（比如块数据跨系统移动）。术语容中断网络和容延迟网络用于实现较高层跨易变（通常异构）网络的传输[www.dtnrg.org]。这个技术不仅用于传感器网络，而且可以用于其他易变网络，比如空间研究需要的星球间通信系统[www.ipnsig.org]。通信并不是依赖两个固定端点间的持续连通性进行的，而是寻找机会进行的：数据在它能够传输的时候传输，并且节点以存储转发方式承担起传输数据的责任，直至达到端对端目标（比如整批运输）。节点间的传输单元称为束[Fall 2003]，它包括源端应用程序数据和描述如何在终点和中间节点管理和处理它的数据。例如，一个束可能通过逐跳可靠传输来传送。束一旦交付，接收节点就承担起随后的传送责任，以此类推。这个过程不依赖任何持续的路由，而且资源不足的节点将数据传送给下一跳之后，就将存储数据释放了。为了预防故障，数据可以冗余地转发给多个相邻节点。

面向数据编程模型：考虑应用层的互操作，面向数据技术包括定向扩散和分布式查询处理，简单地说，它用于传感器网络的应用。这些技术通过将处理分布在节点间的方法的协作来识别网络内处理的需要。此外，这些技术通过消除节点的标识（和其他组件的名字，比如与节点相关的进程和对象）来识别传感器网络的易变性。正如我们在16.3.2节所讨论的，任何依赖于节点或组件的持续存在的程序在易变系统中不会健壮地运行，因为存在与节点或组件无法通信的可能。

在定向扩散[Heidemann et al. 2001]中，由程序员说明兴趣——它是要注入到系统中一个称为槽（sink）的节点的任务的声明。例如，一个节点可能对动物的出现感兴趣。每个兴趣包括若干属性-值对，它们是将要执行该任务的节点的“名字”。那么，对节点的引用不是通过它的标识而是 689
通过执行所要求任务所需要的特性，比如在某个可感知的范围内的值。

运行系统在称为扩散（参见图16-7a）的进程中将兴趣从一个槽传遍网络。槽将兴趣转发给相邻节点。任何收到兴趣的节点在将它向前传播以搜索匹配该兴趣的节点之前保存它的一个记录，以及回传数据给槽节点所需要的信息，源节点是一个匹配兴趣的节点，它的特性与兴趣中说明的

属性-值对相匹配——例如，它可能装配了合适的传感器。对于一个给定兴趣，可能有多个源节点（就像将兴趣注入多个槽）。当运行系统找到一个匹配的源节点时，它将兴趣传给应用程序，应用程序接通它所需要的传感器并产生槽节点所需要的数据。运行系统沿着由从槽到源节点构成的路径回送数据给槽。



a）兴趣的传播　b）建立梯度　c）数据传送

图16-7 定向扩散

一般来说，没有节点事先了解其他哪个节点可作为源，因此定向扩散可能包括大量多余的通信。最坏情况下，一个兴趣可能遍布整个网络。然而，有时候兴趣只与某个物理区域相关，比如森林中的一个特定区域。如果传感器节点知道它们的位置，那么兴趣只需传播到目标区域。从原则上讲，节点为了达到该目的应装配卫星导航接收器，尽管自然覆盖（比如茂密的树木）可能阻碍接收数据。

从源到槽的数据回流由*梯度*控制，梯度是节点间的（*方向*，*值*）对，它是当某个兴趣在整个网络中扩散时建立的（见图16-7b）。方向是数据流动的方向，值是应用特定的但可用于控制流动的速率。例如，槽每小时可能只需要一定次数的动物监测数据。从给定源到给定槽可能有多条路径。系统可以采用各种策略来进行选择，包括万一发生故障时使用冗余的路径，或者使用启发式算法找到一条最短路径（见图16-7c）。

690

应用程序员也可以使用称为*过滤器*的软件，过滤器在每个节点上运行，截取经过该节点的匹配的数据流。例如，过滤器可以压缩重复的动物监测警报，它们来自于感知到同一动物的不同节点（可能是图16-7c中源和槽之间的节点）。

另一种用于传感器网络编程的面向数据方法是分布式查询处理[Gehrke and Madden 2004]。在这种情况下，使用一种类似SQL的语言来声明将要由节点共同执行的查询。考虑到与使用某种传感器节点有关的所有已知开销，执行查询最好的方案通常是在用户PC或网络外的*基站*上处理。考虑到处理查询细节的通信模式，基站沿着动态发现路径将优化的查询分发到网络中的节点，比如为了计算平均数据，需要将数据发送到收集节点。使用定向扩散，数据能在网络中聚集以分担通信代价。结果流回到基站，等待进一步处理。

16.4.3 位置感知

在无处不在计算使用的各种感知中，位置感知是最受关注的。位置是移动和上下文敏感计算的一个基本参数。它可以使应用和设备很自然地以依赖于用户在什么地方的方式运转，比如上下文敏感手机。但位置感知有很多其他的用途，从帮助用户在城市或乡下导航到根据地理信息决定网络路由[Imielinski and Navas 1999]。

位置感知系统用于得到对象在某种感兴趣区域内的位置（居住地或其他位置）的数据。这里我们将关注对象位置，然而利用一些技术也可以得到对象的方位值或更多信息，比如它们的速度。

由对象或用户确定自己的位置，还是由其他物体确定位置有很大的区别，特别是谈到私密性时更为重要。后一种情况称为*跟踪*。

图16-8（基于[Hightower and Borriello 2001]中类似的图）显示了某些位置技术和它们的主要特性。其中一个特性是用于得到位置的机制。该机制有时会限制技术的部署（比如技术是运行在

室内还是室外）和本地基础设施需要的装置。该机制也与精确度有关，图16-8中以数量级次序给出。其次，不同技术产生关于对象位置的不同数据类型。最后，技术在给要定位的实体提供的信息方面有区别，这与用户关心的私密性是相关的。在Hightower和Borriello[2001]中概述了额外的技术。

类型	机制	局限性	精确度	位置数据的类型	私密性
GPS	卫星射频源的多时段定位法	室外（卫星可见）	1～10m	绝对地理坐标（纬度、经度、高度）	有
无线电信标	无线基站的广播（GSM、802.11、蓝牙）	无线覆盖区域	10m～1km	接近已知的实体（通常是语义上的）	有
Active Bat	无线电和超声波的多时段定位法	安装了传感器的天花板	10cm	相对（房间）坐标	暴露Bat身份
超宽带	无线电脉冲接收的多时段定位法	接收器装置	15cm	相对（房间）坐标	暴露标记身份
Active badge	红外传感	日光或荧光	房间大小	接近已知的实体（通常是语义上的）	暴露Badge身份
自动识别标记	RFID、近距离通信、可视标记（比如，条形码）	阅读器装置	1cm～10m	接近已知的实体（通常是语义上的）	暴露标记身份
Easy Living	视觉，三角测量	照相机	不定的	相对（房间）坐标	无

图16-8 一些位置感知技术

美国全球定位系统（GPS）是卫星导航系统的一个最著名的实例——一个通过卫星信号确定接收器或装置的近似位置的系统。其他的卫星导航系统有俄罗斯的GLONASS系统和计划中的欧洲Galileo系统。由于在建筑物内信号会削弱，所以GPS只作用于室外，它通常用于车辆和手持设备的导航，并且逐渐用于非传统的应用，比如在城市内将依赖于位置的媒体数据传送给人们[Hull et al. 2004]。接收者的位置是根据绕地球6个平面运行的24颗卫星的一个子集计算的，每个平面4颗。每颗卫星每天绕地球运行2圈。每颗卫星广播其上原子钟的当前时间，以及在一段时间内它的位置信息（通过来自地面站的观测判断）。接收者（它的位置是确定的）根据信号被接收到的时间和它被广播的时间的差值（也就是信号编码的时间）以及无线电从卫星到地面传播的速度估计来计算它到每个可见卫星的距离。然后，阅读器利用称为“多时段定位法”的三角计算来计算它的位置。要得到一个位置，那么至少有3颗卫星对接收者是可见的。如果刚好有3颗卫星可见，那么阅读器就只能计算出它的纬度和经度；若有更多可见卫星，也可以计算高度。

另一种可能在广域范围，至少在人口高度密集区域工作的定位方法是，监听从范围有限的无线基站发出的（定期广播的）信标。设备可以用信号强度度量哪个是最近基站。移动电话的每个GSM基站都有一个单元标识符；802.11接入点有一个基本服务区标识符（BSSID）。基站定期广播它们标识符的信标，除非为了安全的原因而配置为不发送。蓝牙“信标机”是一种设备，它为另一台发现它的设备提供它的标识符，但它实际上不会广播它的标识符。

691～692

无线电信标自身不会确定实体的位置，除非它接近另一个实体时。如果发信标的实体的位置已知，那么就知道目标实体的位置是在无线电射程内。因此绝对定位是在数据库中查找信标标识符。管理无线电源的机构（比如电信提供商）一般不会公开位置的细节。但是在一些社区项目中，用户可以自己提供详细的位置。

接近度本身就是一个有用的属性。例如，使用接近度可以创建位置敏感的应用，该应用在用户返回到以前访问过的位置时触发。例如，在火车站等待的用户可以创建一个警告，当他在每月的第一天走近火车站时（也就是说，当他们的设备接收到相同的信标标识符时），提醒他去买新的火车月票。蓝牙是另一种新颖的无线电技术，它具有一个有趣的特征，就是有些无线电信标（例

如，某些与移动手机集成的信标）本身是移动的。这仍然有用。例如，火车乘客会通过他们的移动手机从经常一起旅行的人，——“熟悉的陌生人”那里接收数据。

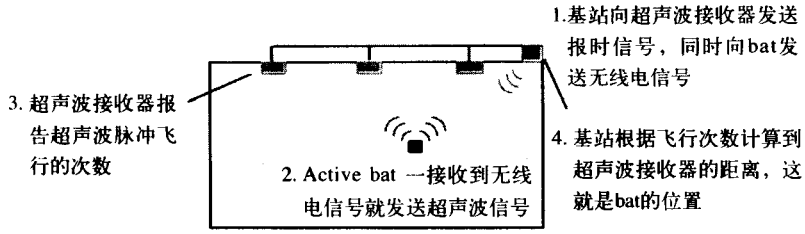


图16-9 在房间内定位一个Active Bat

考虑更为明确的定位形式，GPS得到室外对象的*绝对*（也就是全球）坐标。相比之下，Active Bat系统[Harter et al. 2002]能够产生对象或人在室内的位置的*相对*坐标，也就是相对于该对象所在房间的位置得到的坐标（见图16-9）。Active Bat系统的精确度大约是10cm。相对精确的室内位置对应用而言是有用的，比如检测移动用户距离哪个屏幕最近，并且使用VNC协议（见2.2.3节）将PC的桌面“远距离运输”到该屏幕。Bat是附着在要定位的用户或对象上的一种设备，它能接收无线电信号并发射超声波信号。该系统依赖位于天花板已知位置的一个超声波接收器子网，该子网通过有线连接连到一个*基站*。为了定位Bat，基站向bat发射一个包括其标识符的无线电信号，同时向安装在天花板的超声波接收器发射一个有线信号。当带有给定标识符的bat接收到基站的信号时，
693 它发射一个短超声波脉冲。当天花板上的接收器接收到基站的信号时，它启动一个计时器。因为电磁波传播的速度远远大于声音的传播速度，所以超声波脉冲的发射和计时器的启动实际上是同时的。当一个天花板接收器接收到相应的超声波脉冲（来自Bat）时，它计算间隔的时间，并将时间转发给基站，基站根据对声音速度的一个估计来推断接收器到bat的距离。如果基站接收到至少三个非共性超声波接收器的距离，它就能计算出bat在三维空间的位置。

超宽带（Ultra Wide Band，UWB）是用来在短程内（不超过10m）以高比特率（100Mbps或更多）传播数据的技术。比特通过低功耗但很宽的频谱，使用细脉冲（宽度为1ns）进行传播。给定脉冲的大小和形状，可以测量脉冲的变化的次数从而得到很高的精确度。通过在环境中放置接收器和使用上述技术以及多时段定位法，可以确定一个UWB标记的坐标，其精确度大约为15cm。与其他技术不同，UWB信号可以穿过建筑物内的墙壁或其他类型的障碍物。它的另一个优点是功耗低。

GPS、Active Bat和UWB都能提供对象的*物理位置数据*：在物理区域中的坐标。知道物理位置的一个好处是，通过*地理信息系统*（GIS）和建筑空间的*世界模型数据库*，一个位置能与对象的很多其他类型的信息或其他对象相联系。然而，不利之处是生成和维护数据库所需的开销，这些数据库可能会有很高的变化率。

相比之下，Active Badge系统（见16.1节）产生对象的*语义位置*：位置的名字和描述。例如，如果一个Badge被101室的红外线接收器感知到，则Badge的位置确定为“101房间”（与大多数无线电信号不同，建筑材料会极大地削弱红外线信号，所以Badge不可能在房间外被识别）。除了在空间中的位置，该数据什么也没有告诉我们，但是它为用户提供了与他居住的世界相关的信息。相对而言，相同地点的纬度和经度51°27.010 N 002°37.107W可以用于计算到其他地点的距离，但人们使用它很困难。注意，无线电信标（它与Active Badge截然不同，它将接收器放置在将要定位的目标上而不是在基础设施上）可以用于提供语义位置，或是（非常接近的）物理位置。

Active Badges是*自动标识标记*的一种特殊形式，自动标识标记是电子可读的标识符，它通常是为大规模工业应用设计的。自动标识标记包括RFID[Want 2004]、近距离通信（NFC）

[www.nfc-forum.org]和象形文字或其他可视符号（比如条形码），特别是那些设计成可被远处的照相机读取的符号[de Ipiña et al. 2002]。这些标记附着在要确定位置的对象上。当在被作用范围有限且位置已知的阅读器发现时，目标对象的位置也就可知了。

最后，Easy Living项目[Krumm et al. 2000]使用视觉算法来定位对象，比如被多个照相机观测的一个人。一个目标对象如果能被一个在已知位置的照相机识别，它就可以被定位。原则上，如果一个已知位置有多个照相机，所摄图片上对象外观之间的区别可以用来确定对象的物理位置。 694

正如在Cooltown案例研究（见16.7.2节）中将要介绍的，上面某些位置技术（特别是自动标识标记和红外线信标）通过可用的标识符，可以对它们所依附的实体的信息和服务进行访问。

比较以上技术的私密性，GPS解决方案提供了绝对的私密性：GPS操作从不将关于接收设备的信息传送到其他地方。无线电信标能提供绝对的私密性，但它依赖于使用的方式。如果设备只是监听信标，并且从不与基础设施进行其他的通信，那么可以保证它的私密性。相比之下，其他技术属于跟踪技术。Active Bat、UWB、Active Badge和自动标识方法都产生一个属于基础设施的标识符（在已知的地点、已知的时间给出）。即使相关的用户没有公开他们的身份，也能推断出来。Easy Living的视觉技术依赖于识别出用户并定位他们，所以用户的身份更容易暴露。

用于位置感知的体系结构 定位系统需要的两个主要特征是：(1) 用于位置感知的各种传感器的类型的一般性，(2) 关于要定位对象的数目和当对象（比如人和车辆）移动时发生位置更新事件的速率的可伸缩性。研究人员和开发人员为小的（单独的）智能空间（比如覆盖了传感器网络的房间、建筑物或自然环境）的位置感知设计了相应的体系结构；也为高可伸缩地理信息系统开发了相应的体系结构，该系统覆盖广大区域并包含大量对象的位置。

位置栈[Hightower et al. 2002, Craumann et al. 2003]用来满足一般性的需求。它将用于单个智能空间的位置感知系统分成若干层。*传感器层*包含用于从各种位置传感器中抽取原始数据的驱动程序。*测量层*将原始数据转化为常见的测量类型（包括距离、角度和速度）。*融合层*是应用程序可用的最低层。它结合来自不同传感器（通常是不同类型的）的测量数据，从而推断对象的位置，并通过一个统一接口提供给应用。因为传感器产生不确定的数据，所以对融合层的推理是基于概率的。Fox 等[2003]概述了可用的贝叶斯技术。*安排层*推断出对象间的关系，比如它们是否是位置相关。以上诸层是为了合并来自其他类型传感器的位置数据，以确定较复杂的上下文属性，比如一栋房子中的一群人是否全部睡着了。

可伸缩性是地理信息系统关心的一个主要方面。*时空查询*说明了可伸缩性需求。比如“最近60天内谁住在这座楼内?”，“是否有人跟着我?”或者“该区域内哪种活动的对象最容易发生碰撞?”就是时空查询。要定位的对象数目（尤其是移动对象的数目）和并发查询的数目可能很大。此外，上述查询的最后一个例子要求实时响应。使位置系统具有可伸缩性的一个简单方法是使用数据结构（比如四叉树）递归地将感兴趣的区域分成子区域。这种时空数据库的索引技术是一个热门研究领域。 695

16.4.4 小结和前景

本节描述了几种为上下文敏感计算而设计的基础设施。我们重点讨论了如何利用传感器产生应用所依赖的上下文属性的方法。我们介绍了用于静态传感器集合的体系结构和用于高易变传感器网络的体系结构。最后，我们描述了一些关于位置感知方面的技术。

通过上下文敏感，我们将日常物理世界与计算机系统集成到一起。还需要解决的一个主要问题是：与人类对物理世界的细微理解相比，我们描述的系统有些粗糙。不但传感器（至少是那些足够便宜以至广泛部署）不可避免地不够精确，而且从原始传感器数据所包含的丰富信息中产生精确的语义也是极其困难的。机器人世界（除了感知，它还包括刺激，这是我们忽略的一个话题）

解决该问题已经好多年了。在严格受限区域内，比如室内吸尘器清洁或工业产品，机器人能妥善地执行任务。但是从受限领域的应用到普遍化的应用依然是很困难的。

16.5 安全和私密性

易变系统引发了许多新的安全和私密性问题。第一，易变系统的用户和管理员要求他们的数据和资源具有安全性（机密性、完整性和可用性）。但是，正如我们在16.1节描述易变系统的模型时所指出的，在易变系统中，信任度（所有安全性的基础）经常被降低。降低信任度是因为组件自发交互的法则少，如果只有很少关于对方的先验知识，则可能没有共同的可信任的第三方。第二，很多用户关心他们的私密性。粗略地讲，是他们控制对自身信息访问的能力。但是由于在智能空间中能感知用户经过，因此私密性比以前更可能受到威胁。

尽管存在这些挑战因素，但确保人们的安全和私密性的措施必须是轻量级的，一方面是为了保存交互的自发性，另一方面由于很多设备的用户界面条件有限。例如，在办公室内使用智能笔之前，人们不想“登录”到智能笔。

本节我们将概述易变系统的安全和私密性的几个主要问题。Stajano[2002]给出了一些问题的较详细的处理措施。Langheinrich[2001]从历史的和法律的角度上分析了无处不在计算的私密性问题。

16.5.1 背景

由于与硬件相关的问题，比如资源缺乏，安全和私密性在易变系统中是很复杂的，并且由于
696 它们的自发性导致了新型的资源共享。

硬件相关的问题 传统的安全协议往往基于对设备和连通性的假设，这些假设在易变系统中是不成立的。第一，便携设备（比如PDA、手机和传感器节点）通常比PC类的设备更容易被偷和受到干扰（即使在锁着的房间里）。易变系统的安全性设计应该不依赖于任何容易失效的设备子集的完整性。例如，如果智能空间跨越一个足够大的物理区域，那么保护系统整体完整性的一种方法是：让攻击者必须在大约同一时间访问它内部的许多位置，否则攻击不可能成功。

第二，易变系统中的设备有时没有足够的计算资源用于非对称（公钥）加密，即使使用椭圆曲线加密也是如此（见7.3.2节）。SPINS[Perrig et al.2002]为无线传感器网络中的低功率节点在有潜在攻击的环境中交换数据提供安全性保证。该协议只使用对称密钥加密，与非对称密钥加密不同，它在那些低功率的设备上使用方便。然而，它避开了下面的问题：无线传感器网络中的哪个节点应该共享相同的对称密钥。一种极端情况是，如果所有节点共享相同的密钥，那么在一个节点上攻击成功将毁掉整个系统。另一种极端情况是，如果每个节点享有一个不同于其他所有节点的密钥，那么将有太多的密钥要保存在只有有限内存的节点中。一个折衷方案是节点只与距离它最近的邻居共享密钥，并且依赖于成熟的可信赖节点链，该链对消息按跳加密，而不是使用端对端的加密。

第三，和之前一样，能量也是个问题。不但要使安全协议尽量减少通信负荷以延长电池的使用时间，而且有限的能量是一种新型拒绝服务攻击的基础。Stajano和Anderson[1999]描述了在电池供能的节点上的“睡眠剥夺折磨攻击”：攻击者可以通过发送伪造的消息使设备耗尽它们的电能（因为设备在接收消息时要消耗能量），以此完成拒绝服务的攻击。Martin 等[2004]描述了更进一步的“睡眠剥夺”攻击，包括隐蔽地给设备提供数据和代码来浪费设备的资源。例如，攻击者可以提供一个GIF动画，对用户而言，它看起来是静态的，但实际上需要反复渲染。

最后，断链操作意味着最好不要使用依赖于持续在线访问服务器的安全协议。例如，假设休息站的自动售货机只为某公交公司的乘客提供免费的点心和饮料。与其假设这样的机器总是要连接到公司的总部来验证权限，不如设计出一个协议，给用户设备（比如手机）发放一个证书使得自动售

货机使用蓝牙或其他短程通信手段来验证权限[Zhang and Kindberg2002]。遗憾的是，不使用在线服务器意味着证书不能撤回，只能设计一个过期时间来解决这个问题，但这又会产生一个新的问题：离线设备如何安全地保证精确的时间。

新型资源共享：问题例子 易变系统引发了新型的资源共享，这需要新的安全性设计，如下面的例子所示。 697

- 智能空间的管理者使得访问者可以通过无线网络访问服务，例如发送幻灯片到会议室的投影仪或使用咖啡屋内的打印机。
- 同一公司的在会议上遇见的两个员工可以在他们的手机或其他便携设备之间通过无线连接交换文档。
- 护士从一个类似盒子的设备中取出一个无线心率监控器，将它装在病人身上，并且将它关联到诊所中该病人的数据日志服务上。

这些事例都是自发互操作例子，每个例子都提出了安全或私密性问题。它们与在有防火墙保护的企业内部网或者开放的因特网上的资源共享模式不同。

投影服务和打印服务只能由访问者使用，但无线网络可能越出建筑物的边界，这样攻击者可以窃听、干扰展示或发送伪造的打印任务。所以，服务需要保护，类似于Web服务器只为俱乐部成员服务。但是登录（输入用户名和密码）和处理登录的注册过程开销比较大。此外，用户可能会出于私密性考虑而反对。

两个员工间的文档交换在某些方面类似于在公司内部网内发送一封邮件。这种交互通过一个公共的无线网络发生在一个几乎充满陌生人的地方。原则上存在可信任的第三方（员工所在公司），但实际上第三方可能不可达（在会议室，员工的手机也许不能得到足够好的无线电通信信号）或者第三方可能没有配置在所有用户的设备上。

护士的工作在某些方面类似于第一个例子（指访问者可以使用一个投影仪或打印机），她可以临时但安全地使用一个可信任的设备。但是这个例子要强调的是重用问题。可能有很多的无线传感器在不同的时间用于不同的病人，如何安全地在设备和各个病人日志间建立和打破关联是最基本的问题。

16.5.2 一些解决办法

为了解决在易变系统中提供安全和私密性的问题，我们现在来看一下已有的尝试：安全自发设备关联、基于位置的认证和私密性保护。我们将要描述的安全技术很明显地脱离了分布式系统的标准方法。它们利用了如下事实：我们考虑的系统被集成到了日常的物理世界中，是通过使用物理证据而不用密码学证据来自提升安全性特性。

安全自发设备关联 前面例子提出的一个重要问题是如何在通过无线电网络W（蓝牙或802.11）相连的两个设备间保护自发关联。这就是*安全自发设备关联问题*，也称为*安全短暂关联问题*。目标是通过在两个设备间安全地交换会话密钥并使用该密钥来加密W上的通信来创建安全通道。因 698
为关联是自发的，所以首先要假设设备（或它的用户）既不与其他设备共享密钥，也没有其他设备的公钥，并且设备也不访问可信任的第三方。即使可信任的第三方存在，它也有可能是脱机的。攻击者可能试图在W上窃听，并重播或合成消息。特别是攻击者可能试图发起一个中间人攻击（见7.1.1节）。

该问题的解决方法能够使访问者安全连接到投影仪或打印服务；参加会议的同事能够安全地在他们的携带设备间交换文档；护士能够安全地通过病床将无线心率监控器连接到数据日志装置上。

W上的任何通信都不能完成安全密钥交换，所以需要紧急通信。特别地，在以蓝牙连接的两个设备间建立链路层密钥的标准方法依赖于多个用户的紧急控制动作。在一个设备上选择的数字

串必须由用户在其他设备上输入。但是这种方法通常不会安全地实施，因为简单的短数字串（例如"0000"）易于被攻击者通过穷举搜索而得到。

另一种解决安全关联问题的方法是使用带有一定物理特性的侧通道。特别地，经由此侧通道传播的信号在角度、范围或时序（或它们的某种组合）上会受到限制。为了尽可能接近，我们可以推断出此通道上消息的发送者和接收者的特性，这使得我们能够使用一种物理可示范的设备（我们下面将介绍）建立安全关联。Kindberg 等[2002b]称它们为物理受限通道，我们在本书中使用此术语；Balfanz等[2002]称之为有限位置通道。Stajano和Anderson[1999]首先以物理关联的形式开发了此种侧通道。在16.2.2节介绍物理设备关联的时候，我们介绍了这些通道的几个例子。

在一个场景中，某个设备产生一个新会话密钥，并通过某种受限通道将它发送到其他设备，接收受限通道提供一定程度的安全性。也就是说，它限制哪些设备可以接收密钥。下面是用于属于接收受限通道的一些技术：

- 物理接触。每个设备有终端来进行直接电子连接 [Stajano and Anderson 1999]，如图16-10所示。
- 红外线。红外线方向在60°以内，但会被墙壁和窗户大大削弱。用户可以通过红外线传送密钥给距离1米之内的接收设备[Balfanz et al. 2002]。



a）通过物理接触交换新会话密钥K　　b）在W上使用K构建安全通道的设备通信

图16-10 使用物理接触的安全设备关联

699

- 音频。数据可以作为一种音频信号（例如在房间内轻柔地播放的音乐）的调制传送，但它传送的距离很短[Madhavapeddy et al. 2003]。
- 激光。一个用户将设备的携带数据的窄激光束指向另一个设备上的接收器来传送数据[Kindberg and Zhang 2003a]。该方法具有比其他远程技术具有更高的精确度。
- 条形码和照相机。一个设备在它的屏幕上将密钥显示成条形码（或其他可解码的图像），另一个设备（带有摄像头，比如可拍照手机）读取并解码。该方法的精确度与设备间的距离成反比。

通常，物理上受限的通道只提供有限的安全性。具有高灵敏度接收器的攻击者就能以红外线或音频窃听。具有功能强大的照相机的攻击者可以读取条形码（即使在小屏幕上）。激光会受大气散射的限制，尽管量子调制技术可以使得发散的信号对窃听者没有用处[Gibson et al.2004]。然而，当技术部署在合适的环境中时，攻击者就需要付出极大的努力才能完成攻击，这样的安全性足以满足日常工作的需要。

第二种用于安全交换会话密钥的方法是使用受限通道在物理上认证设备的公钥，并将它发送给其他设备。之后设备参与到一个标准协议中，使用认证的公钥以交换会话密钥。当然，该方法假定设备功能足够强大，可以执行公钥加密。

对于设备而言，认证公钥最简单的方法是将它通过一个发送受限通道发送出去。这使用户能够认证密钥是从物理设备中得到的。有几种方法实现发送受限通道。例如，物理接触提供了一个发送受限通道，因为只有一个直接连接的设备可以在该通道上发送。习题16.14将请读者考虑以上描述的哪个接收受限通道技术也提供发送受限通道。此外，如何使用一个接收受限通道实现一个发送受限通道，或者反之亦然。见习题16.15。

对于设备而言，第三种实现物理受限通道的方法是最优化地但不安全地交换会话密钥，之后使用物理受限通道来验证密钥。也就是说，使用物理受限通道来证实密钥只由所请求的物理实体拥有。

首先，我们考虑如何用自发但可能包含错误的方式交换会话密钥，随后介绍几种技术来验证交换。如果验证失败，那么过程还可以重复。

在16.2.2节中，我们描述了用于关联两个设备的物理的和由人控制的技术，比如两按钮协议，

当人按下设备上的按钮时，设备几乎同时交换它们的网络地址。可以方便地修改，使设备使用 700
Diffie-Hellman协议[Diffie and Hellman 1976]交换会话密钥，但是，该方法是不安全的：由于同时运行该协议，单独的用户组仍有可能意外地错误关连设备，并且怀有恶意的团体仍有可能发起中间人攻击。

以下技术使得我们在使用前验证密钥。尽管也使用接收受限通道，但它们也涉及发送受限通道（见习题16.15）。中间人不能（几乎不可能）同每个设备交换相同的密钥，这是Diffie-Hellman协议的一个特性。所以，我们可以通过比较两个设备运行Diffie-Hellman协议后得到的密钥的安全散列值来验证关联（见图16-11）。

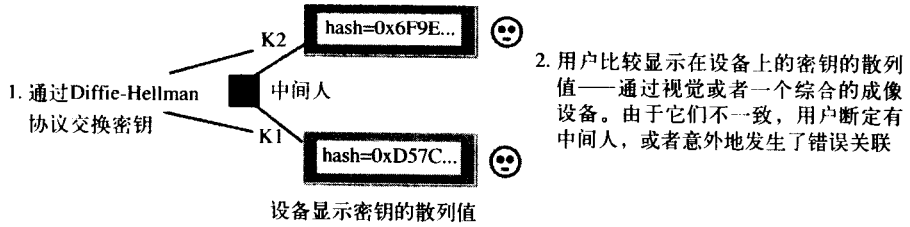


图16-11 检测中间人

- 显示的散列值。Stajano和Anderson指出，每种设备都可以以十六进制字符或其他人工可以比较的形式显示它的公钥的散列值。然而，他们认为这种人为参与的类型很有可能产生错误。上面的条形码方法可能更可靠。该方法是使用发送受限通道的另一个例子：一个设备的显示屏和接近条形码的另一个设备的摄像头间的光路安全地从所请求的设备传播安全散列值。
- 超声波。通过使用类似于16.4.3节描述的Active Bat[Kindberg and Zhang 2003b]使用的技术，将超声波信号与无线电信号结合，可以推断发送散列值的设备的距离和方向。

考虑以上方法，由于受限通道的特性，它们提供的安全性的程度是不同的，但是所有方法都适合于自发关联。没有一个方法要求在线访问其他组件。没有一个方法要求用户去认证他们自己或查找设备的电子名称或标识符，相反，要向用户提供关于哪个设备已安全关联的物理证据。通过假设，用户已在那些设备（和它们的用户）中建立了信任。当然，已获得的安全性只是同设备涉及的可信任性一样：可能将一个设备“安全地关联”到另一个实际上发起攻击的设备上。

Stajano和Aderson[1999，Stajano 2002]在“复活鸭”协议的上下文中使用了物理受限通道。该协议与无线心率监控器的例子是相关的，在那个例子中，几个同样的设备在病人间关联或重新关 701
联。该协议的名字指下面的事实：（实际的）鸭在*可盖印*的状态下开始，随后受控于任何它们首先认出的实体（理想情况下，是它们的母亲！），该过程称为*盖印*。在我们的情况中，“鸭”设备受控于第一个与它关联的设备，并且之后拒绝来自任何其他实体的请求，也就是在盖印时不知道“鸭”与它的“母亲”交换的密钥的主体。重新关联只能通过先“杀死该鸭的灵魂”来进行。例如，当“母亲”指示“鸭”重新呈现可盖印的状态，该情况下它的记忆被安全地擦除了。从这一点看，“鸭”准备受控于下一个与它关联的设备。

基于位置的认证 访问者使用会议室的投影服务以及用户在咖啡店打印文档的例子可以从访问者和管理者两个角度来看。从访问者角度看，他们可以使用前面介绍的一种物理受限通道安全地将他们的设备关联到投影仪或打印机上，从而保护数据的私密性和完整性（尽管在咖啡店内打印敏感的文档是不明智的）。

但是那些智能空间的每个管理者都有附加的需求：在让他们的访问者享有安全性的同时，他们需要实现访问控制。只有物理上位于他们空间内的人（会议室的讲演者，在咖啡店喝咖啡的人）才能使用他们的服务。可是，正如我们已经解释过的，由于访问者私密性的需求和管理者对自发

出现和消失的用户和设备集成一体的需要，认证用户的身份可能是不合适的。

满足那些需求的一种认证方法是使访问控制基于服务的客户的位置，而不是它们的身份。Kindberg等 [2002b] 描述了一种协议，该协议为了认证客户的位置，使用一种遍及但没有超越智能空间的物理受限通道。例如，该通道可以使用咖啡店内播放的音乐或会议室内的红外线来构建。还有一个嵌入相应智能空间（即直接连接到同一个受限通道）的位置认证代理，位置特定的服务信任该代理。例如，Acme咖啡公司想通过它们的连锁店让它们的客户免费下载多媒体信息作为奖励，但是又希望保证Acme咖啡店外的任何人都不可以访问该多媒体信息，即使下载服务是集中的并且连接到因特网。该协议假定用户通过Web浏览器访问服务并使用Web重定向，这样访问者的设备从位置认证代理可透明地得到证实（即客户设备位于它所声明的位置），并将它转发给目标服务。

Sastry 等[2003] 使用由超声波实现的临时受限通道验证位置声明。该协议的基础是：因为声波的速度是物理受限的，在应答包含在请求包中的信息时，只有一个恰好声明在此的设备可以通过超声波（足够快地）传送消息到目的地。

702 同安全设备关联相比，位置认证只能在有限的程度上保护系统的安全。即使服务已证实客户在一个真实的位置，该客户也可能仍然是恶意的或者作为其他位置客户的代理。

私密性保护 基于位置的认证进行了权衡，这种权衡使得保护易变系统的私密性变得很难：即使用户拒绝提供他们的身份，他们也会暴露与其他类型的潜在标识信息的无意关联的位置。用户信息流经的所有通道都需要安全措施。例如，即使用户在咖啡店匿名地访问电子服务，他们的私密性也可能被破坏（如果照相机捕捉到的话）。如果用户需要为一项服务支付费用，那么他们必须提供电子支付细节，即使通过第三方来完成这项工作。他们也可能要购买那些需要运送到他们指定地址的货物。

在系统层中，最基本的威胁是，当用户走进智能空间并访问其中的服务时，用户会有意地或无意地提供各种各样的标识符。第一，他们在访问服务时可能提供名字和地址。第二，他们个人设备上的蓝牙或IEEE802.11网络接口都有一个固定的MAC层地址，该地址对其他设备（比如接入点）是可见的。第三，如果用户携带标记，比如RFID标记（例如，嵌入到衣服中的RFID标记使智能洗衣机能够自动的选择一种合适的洗衣周期），那么智能空间可以潜在地在门口和其他“关键点”感知那些标记。RFID是全局唯一的，除了用于跟踪外，还可以用于识别用户的什么物品带有该标记（比如他们穿的衣服类型）。

无论标识符的来源是什么，在给定时间内它们都可以与一个位置和一个活动关联，这样就可以潜在地关联用户的个人信息。在智能空间中的用户可能窃听并收集标识符，如果他们与智能空间（或嵌入到智能空间的服务）串通，那么他们就可以跟踪标识符，推断出用户的活动，所有这些都有可能导致私密性的丧失。

科学家们正在如何把当前的“硬”标识符（比如无线MAC地址和RFID）做成“软”地址，这些软地址可以时不时被替代以阻止追踪。改变MAC地址（和更高层的网络地址，比如IP地址）的困难在于会导致通信中断，这就要在它和私密性之间进行权衡[Gruteser and Grunwald 2003]。改变RFID的困难在于，虽然生成RFID的用户不想被“错误”的传感器跟踪，但用户希望他们的RFID标记被某些“正确”的传感器读取（比如他们洗衣机内的传感器）。解决该问题的技术是对于标记使用（单向）散列函数，代替已存储的标识符和每次读取它时给出的标识符[Ohkubo et al. 2003]。只有一个知道标记的原始唯一标识符的可信赖方能够使用发出的标识符来证实哪个标记被读取。此外，因为标记通过一个单向散列函数在给出它们前传递它们存储的标识符，所以攻击者不能（除非他们篡改了标记）获得存储的标识符，也不能欺骗标记。例如，故意错误地声称一个带标记的用户出现在犯罪现场。

考虑客户提供给服务的软件标识符，维护私密性的一种显而易见的方法是要么用一个匿名标

识符（为每个服务请求随机送择的）要么用一个假名来替代。假名是一个不真实的标识符，但是 703
在一段时间内一直用于同一个客户主体。假名相对于匿名标识符的优势是它使客户在不必暴露真实身份就可与给定服务建立一种信任的关系。

对于用户来说，管理匿名或假名标识符太麻烦，所以这项工作通常由一个称为*私密性代理*的系统组件完成。私密性代理是一个用户信任的可以将所有服务请求都匿名发送的组件。每个用户设备有一个到私密性代理的安全的专用通道。该代理将服务请求中所有的真实标识符用匿名标识符或假名代替。

私密性代理的一个问题是它是被攻击的中心：如果代理被击破，那么所有的客户服务将被泄露。另一个问题是代理不会隐藏用户访问了哪个服务。一个窃听者或一群窃听者可以使用*流量分析*，即观察流向或流出一个特定用户设备的消息和流向或流出一个特定服务的消息间的流量的相关性，检查消息的时序和大小等因素。

*混合*是一种统计技术，它以一种攻击者很难整理出用户的动作规律的方式，综合来自许多用户的通信，从而维护用户的私密性。混合的一个应用是构建代理的一个重叠网络，在消息进入网络以后，代理对消息加密、聚集、重排以及在它们之间使用多跳传送消息，从而对服务或客户而言，很难在来自客户或服务的进入网络的消息和离开网络的消息间建立联系 [Chaum 1981]。每个代理只信任它的邻居，且只与它们共享密钥。没有攻破所有代理就很难危害到网络。Al-Muhtadi等[2002]描述了一种用于在智能空间中将客户的消息匿名地路由到服务的体系结构。

混合的另一种应用是利用在每个位置出现许多用户的存在来隐藏用户的位置。Beresford 和 Stajano[2003]描述了一种使用*混合地带*隐藏用户位置的系统，混合地带是一个用户不会访问位置敏感服务的区域，比如智能空间之间的走廊。其基本思想是用户在混合地带内改变他的匿名身份，在那里没有任何用户的位置信息。如果混合地带足够小并且有足够多的人经过混合地带，那么混合地带可以扮演一个类似于匿名代理的混合网络的角色。习题16.16在更深层次上考虑了混合地带。

16.5.3 小结和前景

本节介绍了在易变系统中提供安全性和私密性的问题，并简要介绍了几种尝试性的解决方案，包括安全自发关联、基于位置的认证和各种保护私密性的技术。广泛的感知、与硬件有关的问题（比如资源缺乏）和自发关联是产生困难的根源。感知增加了用户对私密性的关注，因为不但他们的服务访问被监控，而且他们的位置等基本信息也被监控；与硬件有关的问题和自发性限制了我们提供安全方案的能力。这是一个重要的研究领域：安全性，尤其是私密性可能成为使用易变系 704
统的障碍。

16.6 自适应

本章所研究的易变系统的设备在处理能力、I/O能力（比如屏幕大小、网络带宽、内存和能量容量）方面与PC相比，异构性更加明显。由于我们为设备设定了多个目标，所以异构很难消除。便携和嵌入设备的不同要求意味着资源（比如能量和屏幕大小）最缺乏和最丰富的设备之间有着巨大的差别（在资源上仅有的正面总体趋势是逐渐增容的但是可负担得起的持久存储[Want and Pering 2003]）。可能唯一不变的就是（运行时的）变化本身：运行条件（如可用的带宽和能量）是易于动态变化的。

本节将介绍*自适应系统*，它们基于资源变化模型，并且使它们的运行时行为适应当时的资源可用性。自适应系统的目标是通过允许软件在上下文中重用以容纳异构（上下文会根据一些因素变化，这些因素包括设备功能和用户偏好）。同时，通过适应应用行为但不牺牲关键的应用特性来容纳变化的运行时资源条件。但是，实现这些目标是极其困难的。本节给出了自适应领域的一些特点。

16.6.1 内容的上下文敏感自适应

在16.3.1节，我们看到易变系统的某些设备为其他设备提供多媒体内容。多媒体应用（见第17章）通过交换或传输多媒体数据（比如图像、音频和视频）运作。

交换内容的一种简单办法是，内容产生者向所有内容应用设备发送相同的内容，而内容应用设备根据它的需要和限制对内容进行适当的呈现。确实，该方法有时有效，只要内容被指定得足够抽象，接收设备就总能找到一个适合它需要的具体表现方法。

然而，一些因素（比如带宽限制和设备异构）使得该方法通常不可行。与PC不同，易变系统的设备接收、处理、存储和显示多媒体内容的能力差别非常大。它们屏幕大小的差异（有的甚至没有屏幕）会导致在发送固定大小的图片、固定字体的文本，以及固定布局的内容时得到不满意的结果。设备可能有（也可能没有）PC机所应有的其他类型的I/O：键盘、麦克风、音频输出设备等。即使一个设备有I/O硬件来呈现某种形式的内容（比如视频），它也可能没有某种编码（例如MPEG或Quicktime）所需的软件或者没有足够的内存及处理资源来以完全的保真度（对视频信号
705 而言指全分辨率或帧率）呈现多媒体。最后，设备可能具有足够的资源来呈现给定的内容，但是如果设备的带宽太低，内容就不能发送到设备（除非内容被适当地压缩）。

更普遍的情况是，服务需要传送到给定设备的内容是上下文的一个功能：媒体制作人不但应该考虑应用设备的能力，还要考虑设备用户的偏好，以及他或她的任务的本质等因素。例如，一个用户可能更希望在小屏幕上显示文本而不是图像；另一个用户更喜欢音频输出而不是视频输出。此外，服务传递的内容片断中的项可能是用户任务的一个功能。例如，某一区域的地图上的特征依赖于用户是参观景点的旅游者还是寻找基础设施接入点的工作者而确定[Chalmers et al. 2004]。在一个屏幕大小有限的设备上，如果地图只包括一种特征类型，那么该地图就比较易读了。

对多媒体内容的作者来说，可能会因为开销太大以至于不能为很多不同的上下文配置各自的解决方案。替代方法是改变最初的数据使之符合一种适当的形式，可采用的手段有从中选择、从中生成内容或对其进行转换，或以上三个过程的任何结合。有时，原始数据与应该如何表示它无关。例如，数据可能是XML格式，脚本是扩展样式表转换语言格式（XSLT），它用于在给定的上下文中创建可呈现的形式。在其他情况下，原始数据已经是一种多媒体数据，比如图像，在这种情况下，自适应过程称为*转码*。自适应性可以在媒体类型内（例如，选择地图数据或降低图像的分辨率）和媒体类型间（例如，根据用户的偏好或根据使用的设备是有屏幕还是有音频输出而将文本转化成语音，反之亦然）发生。

在因特网（特别是Web）上的客户/服务器系统中，内容自适应问题已得到很多的关注。Web模型使得自适应发生在资源丰富的基础设施上（要么在服务本身中，要么在代理中），而不是在资源缺乏的客户端进行自适应。HTTP协议允许就内容类型进行协商（见4.4节）：客户在它的请求头中为它可以接收的内容的MIME类型声明偏好，接着服务器设法在它返回的内容中匹配这些偏好。但是该机制对于上下文敏感自适应的作用很有限。例如，客户能声明可接收的图像编码，但不能指定设备的屏幕大小。万维网联盟（W3C）通过它的设备独立工作小组[www.w3.XIX]和开放移动联盟（OMA）[www.openmobilealliance.org]正在开发标准，通过这些标准能比较详细地表达设备的功能和配置。W3C开发了复合功能/偏好设置文件（CC/PP）使得不同类型的设备可以描述它们的功能和配置，比如屏幕大小和带宽。OMA的*用户代理偏好*规约为手机提供了CC/PP词汇。它非常详细，以至于对某一给定设备，它的大小能达到10KB以上。这样的偏好在带宽和能量方面显得太过昂贵，以至于不能同请求一起发送，所以移动手机只能在请求头中发送它的偏好的URL。服务器检索规约以获得匹配的内容，并且为了将来的使用，会将该规约缓存起来。

706 对带宽受限设备，一类很重要的自适应是类型特定的压缩。Fox 等[1998]描述了一种体系结构，其中代理在服务（它可能是或可能不是Web的一部分）和客户之间完成压缩。它们的体系结构中

有3个主要特征：

- 为了适应有限带宽，压缩应该特定于媒体类型但会有损耗。这样语义信息可用来决定哪种媒体特征比较重要，应予以保留。例如，通过去掉颜色信息来压缩图像。
- 转码应该在传输中进行，因为静态预先准备的内容形式不会提供足够大的灵活性去处理动态数据和不断增加的客户和服务器组合的情况。
- 转码应该在代理服务器上进行，这样客户和服务器两者都无需关注转码。不必重写代码，计算密集的转码活动可以在合适的可伸缩的硬件上运行（比如机架固定的计算机集群），从而使延迟保持在可接受的范围内。

当谈论到易变系统（比如智能空间）时，我们要回顾一下为Web和其他因特网规模的自适应作出的一些假设。易变系统的要求更加苛刻，因为它们可能要求在任何一对动态关联的设备间有自适应性，这样自适应性不再受限于基础设施中某个服务的客户。现在，有很多潜在的提供商，他们的内容需要适配。此外，这些提供商也可能因为缺乏资源而不能自行完成某些类型的自适应。

对智能空间的一个建议是，在它们的基础设施中提供代理以实现它们拥有的易变组件间的内容适配[Kiciman and Fox 2000; Ponnekanti et al. 2001]。第二个建议是应该更密切地观察哪种类型的内容适配可以并且应该在小型设备上完成。特别地，压缩是一个很重要的例子。

即使在基础设施中只有一个强大的自适应代理，设备仍然需要将它的数据发送给代理。上面我们讨论过，与处理相比通信是非常昂贵的。原则上，在发送前压缩数据是最有效的节约能量的方法。然而，压缩时的内存访问模式对能量消耗有很大的影响。Barr和Asanovic[2003]证实，第一次压缩数据时使用默认的实现可能要消耗更多能量，但是对压缩和解压缩算法（特别是对内存访问模式）进行细致的优化后，与传送未压缩数据相比可能会极大地节约能量。

16.6.2 适应变化的系统资源

虽然设备间的硬件资源（比如屏幕大小）是不同的，但它们至少是稳定的并且是我们熟悉的。相比之下，应用还依赖于运行时变化的、难预测的资源，比如可用的能量和网络带宽。在本节中，我们将讨论在运行时处理资源级别变化的技术。我们将讨论操作系统对运行在易变系统中的应用的支持和在智能空间基础设施中为应用增强资源可用性的支持。 707

对易变资源自适应性的OS支持 Satyanarayanan[2001]描述了自适应性的三个方法。一个方法是为应用请求资源并得到资源预留。虽然资源预留对应用而言很便利（见第17章），但在易变系统中满足QoS保证有时很难，甚至在某些情况下（比如能量耗尽的时候）是不可能满足的。第二种方法是通知用户可用资源的级别发生了变化，这样他们可以根据应用作出反应。例如，如果带宽变低，视频播放器的用户可以操作应用程序中的滑动条来改变帧率或分辨率。第三种方法是为OS通知应用资源条件发生了变化，应用根据它的特定需求进行适应。

Odyssey[Nobel and Satyanarayanan 1999]为应用适应资源（如网络带宽）可用级别的变化提供了操作系统支持。例如，如果带宽降低，那么视频播放器可能切换到颜色较少的视频流，也可能调整分辨率或帧率。在Odyssey体系结构中，应用程序管理数据类型，比如视频或图像，并且随资源条件的变化调整保真度（类型特定的质量），根据保真度呈现数据。称为总督的系统组件将设备所有的资源分配给运行在设备上的几个应用。在任何时候，每个应用运行时带有一个容忍窗口以容忍资源条件的改变。容忍窗口给出了资源级别的一个区间，根据实际资源变化，它要选得足够宽以便与实际适应，但也要足够窄以使得应用始终运行在该限制内。当总督要将资源等级变成容忍窗口外的一个值时，它对应用发出一个向上调用，之后应用做出相应反应。例如，如果带宽降到较低级别，视频播放器可能变成黑白色；它可能平稳地调整帧率和/或分辨率。

利用智能空间资源 在Cyber Foraging[Satyanarayanan 2001; Goyal and Carter 2004; Balan et al.

2003]中，处理受限的设备在智能空间中发现了一个计算服务器，并将它的一些处理负载转给该服务器。例如，将用户的语音转化成文本是一项处理密集的活动，并且利用便携设备不能满意地实现。利用智能空间资源的一个目标是增加应用对用户的响应度——基础设施中的计算机的处理能力是便携设备的许多倍。但是这也是*能量敏感自适应*的一个例子：该系统的另一个目标是通过将工作分配给大功率计算服务器来保存便携设备的电量。

cyber foraging仍然面临一些挑战。需要将应用分解，以便有效地在计算服务器上处理分解后的子应用。但是如果没有计算服务器可用，应用应该仍能正确运行（虽然比较慢或保真度有所降低）。计算服务器应该运行应用的一部分，这部分应用与便携设备间的通信相对很少，否则在低带宽连接上通信所占用的时间会超过处理所用的时间。此外，便携设备全部的能量消耗必须是令人满意的。因为通信需要消耗大量的能量，所以使用计算服务器不一定能节约能量。与计算服务器
708 通信耗费的能量可能超出转移处理所节约的能量。

Balan等[2003]讨论了如何划分应用来解决上述挑战，并描述了一个用于监控资源级别（比如计算服务器的可用性、带宽和能量）的系统，以及使用一个小的分解选项集合，以适应在便携设备和计算服务器之间划分应用。例如，考虑下面的情景，用户通过对着移动设备说话来显示文字，再将文字翻译为一种外语（他们访问国家的语言）。有多种方法可以在移动设备和计算服务器之间划分这个应用（不同的划分具有不同的资源利用含义）。如果有多个计算服务器可用，那么可以让它们分担识别和翻译的不同的阶段的工作；如果只有一个计算服务器可用，那么这些应用应该在该机器上共同运行或在移动设备和计算服务器之间运行。

Goyal和Carter[2004]采用了一种更静态的方法来划分应用，这种方法假定应用已分解成独立的通信程序。例如，移动设备可以用两种方式执行语音识别。第一种方式是，应用完全运行在移动设备上（但速度很慢）。第二种方式是，移动设备只运行用户界面，用户界面将用户声音的数字音频装载到计算服务器上运行的一个程序中；该程序将识别后的文本回送给移动设备加以显示。对移动设备而言，发送识别程序到计算服务器的能量代价可能很高，所以设备改为发送程序的URL，计算服务器从外部资源中下载该程序并运行。

16.6.3 小结和前景

本节描述了易变系统的两种自适应，自适应是由它们的异质和它们的运行条件的易变性造动的。有根据媒体消费者的上下文（比如设备的特征和设备用户的任务）的多媒体数据适配；还有根据系统资源（比如能量和带宽）的动态等级进行的自适应。

我们指出，在原则上最好制造出一种自适应软件，它可以根据一个深刻理解的变化模型适应变化的环境，而不是随需要被迫以自组织方式进化软件和硬件。然而，制造这样的自适应软件很困难，并且在制造这样的软件方面没有通用的协议。第一，变化模型本身（关于资源等级如何变化和当它们变化时如何反应）就很难得到。第二，存在软件工程方面的挑战。在已存在的软件中，找到合适的自适应的地方需要具有该软件工作的固有知识，并且不一定能成功。然而，当从头开始创建新的自适应软件时，可以利用软件工程领域的技术（比如面向方面编程[Elrad et al. 2001]）
709 帮助程序员管理自适应性。

16.7 Cooltown实例研究

Hewlett-Packard的Cooltown项目[Kindberg er al. 2002a;Kindberg and Barton 2001]的目标是为*游牧计算*提供基础设施，游牧计算是用于面向人的移动和无处不在计算的项目的术语。“游牧”指人在他们的日常生活中，在不同地点（比如家、工作地点和商店）间移动。“计算”指提供给游牧用户的服务——不只是那些只要连通就可以提供的服务（比如邮件），更多的是那些与用户进行移

动的日常物理世界中的实体集成的服务。为了访问这些服务，假定用户携带或穿戴带有传感器的无线设备，比如手机、PDA或智能手表。

特别地，通过下面两个目标，项目把从Web中学到的成功经验运用到游牧计算中。第一，因为Web在虚拟世界中提供了丰富的、可扩展的资源集合，因此大多数资源可以潜在地通过将Web的体系结构和Web中已有的资源扩展到物理世界中而获得。Cooltown的一个目标可以总结为"任何事物都有一个Web页面"：我们物理世界中的每个实体，不论是否是电子的，都有一个相关联的Web资源（称为Web存在），当存在这样的实体时，用户就能够方便地访问Web存在。Web存在可能只是一个包括实体信息的Web页，也可能是与该实体关联的任何服务。例如，一个物理产品的Web存在可以是得到替换部件的一个服务。

第二个目标是为了与设备交互以达到Web高度的互操作性。游牧用户可能需要在以前从未到达的地方与他们以前从未遇到的Web存在交互。为了从这些服务中获益，用户不得不在他们的便携设备上装载新软件或重新配置已有的软件，这对用户而言是不可接受的。

本书我们关注Cooltown体系结构的主要方面（参见图16-12）有：Web存在、物理超链接（它从物理实体连接到Web存在，从而连接到该Web上的超链接资源）、eSquirt以及一个与Web存在设备互操作的协议。



图16-12 Cooltown的各层

710

16.7.1 Web存在

Cooltown考虑将物理实体划分为三种类型：人、位置和物体。人、位置或物体的Web存在是为适合特定应用而选择的任何可能的Web资源。但是Cooltown为人和位置的Web存在选定了某些角色。物体和人的Web存在被集中在位置的Web表示中。所以下面的描述遵循这个顺序。

物体："物体"是指设备或者非电子的物理实体。通过将Web服务器嵌入物体中或者在Web服务器内拥有它们的Web表示，物体就成为Web存在。如果物体是设备，那么URL就是它实现的服务。例如，"因特网收音机"是一个音乐播放设备，它拥有自己的Web存在。发现因特网收音机的URL的用户得到一个可控的Web页面，使得用户可以将它"调"到一个因特网广播源，还可以调整它的设置（比如音量），或者上载用户自己的声音文件。非电子物体也可以有Web存在，这种Web存在是与该物体关联的一个Web资源，但由其他地方的Web服务器所拥有。例如，一个打印出来的文档的Web存在可以是它对应的电子文档；用户可以从物理制品上发现它的Web存在（正如我们在16.7.2节解释的）并请求新的打印，而不是去影印打印的文档（那样会降低质量）。音乐CD的Web存在可能是某些相关的数字内容（如额外的音乐剪辑和图像），这些内容保存在其拥有者的个人媒体集合中。

人：人通过提供带有方便通信的服务的Web主页以及通过提供关于他们当前上下文的信息而成为Web存在。例如，没有移动电话的用户可以通过它们的Web存在使得本地电话号码可用，这个Web存在是一个数值，当他们到处走动时，他们的Web存在自动更新该电话号码值。但是他们也可能选择他们的Web存在来显式地注册当前的位置（通过到他们所处的物理位置的Web存在的一个链接）。

位置：使用本章的术语，位置是智能空间。位置变成Web存在是通过注册在其中的人和物体的Web存在——甚至嵌入的位置或其他相关的位置的Web存在，注册工作可由带有一个位置特定的目录服务完成（见9.3节）。位置的目录也包括静态的信息，比如对位置的物理特征和功能的描述。目录服务使得组件发现位置内的动态Web存在集合，并与之交互。它也可作为有关位置和它的内容的一个信息源，并以Web页面的形式展示给用户。

位置内Web存在的目录条目可以通过两种方法建立，第一，由网络发现服务（见16.2.1节）自动注册该位置子网内设备所实现的任何Web存在——位置内无线连接的设备，或该位置的基础设施服务器。然而，即使网络发现服务很有用，它们也不得不面对不是所有的Web存在都由该位置子网内的设备拥有的事实。非电子物理实体的Web存在（包括人、文档和音乐CD，后两者是被移入位置或被带进位置的）可能属于其他地方。这些Web存在必须在一个称为*物理注册*的过程中手动地注册到那里或通过感知机制（例如，通过感知它们的RFID标记）进行注册。

711

称为*Web存在管理器*[Debaty and Caswell 2001]的服务管理着Web存在的位置（例如，一个建筑物的所有房间），它也管理人和物体的Web存在。位置是Cooltown的上下文抽象的一个特殊实例：一组相关的Web存在实体为某些目的（比如浏览）链接在一起。Web存在管理器将每个Web存在的实体与该实体的上下文中实体的Web存在关联。例如，如果实体是物体，它的相关实体可能是携带它的人和放置它的位置。如果实体是人，他的相关实体可能是他携带的物体、他当前所处的位置和周围的人。

16.7.2 物理超链接

Web存在与其他的Web资源类似，所以Web页也可以像其他链接一样包括到其他Web存在的文本或图像链接。但是，在那些标准Web链接模型中，用户通过*信息源*（即Web页）找到人、位置或物体的Web存在。Cooltown设计使人从他们的*物理源*（也就是他们每天在物理世界移动时遇到的具体的人、位置或物体）直接到达相应的Web存在。

*物理超链接*是一种手段，通过它用户能够从物理实体本身或它邻近的环境检索实体的Web存在的URL。我们现在考虑实现物理超链接的方式。首先，考虑Web页中一个典型链接的HTML标记，比如：

< a href="http://cdk4.net/ChopSuey.html" >Hopper's painting Chop Suey

表示将Web页中的文本“Hopper's painting Chop Suey”链接到位于http://cdk4.net/ChopSuey.html的关于爱德华霍珀的作品Chop Suey的网页。现在，考虑这个问题：博物馆的访问者看到一幅绘画作品，如何通过“点击”该作品以在他们的手机、PDA或其他便携设备的浏览器上得到有关该绘画的信息，可能需要一种方法从作品本身的物理配置中发现该作品的URL。一种可能的方法是将作品的URL写在墙上，这样用户可以将该URL键入他们的设备的浏览器中。但是这种方法是笨拙的、费力的。

Cooltown利用用户拥有与他们的设备集成的传感器，研究出了两种通过传感器发现实体的URL的方法：*直接感知*和*间接感知*。

直接感知：在该模型下，用户设备直接从标记（“自动标识”标记）或附在感兴趣的实体上的信标或在实体旁边的信标感知到URL（见16.4.3节）。一个相对较大的实体（比如房间）在容易看见的位置可能有几个标记或信标。标记是一个被动的设备或制品，当用户将他们设备的传感器放置在它旁边时，它会显示URL。例如，可拍照手机在原理上可以对写在标签上的URL实行视觉字符识别或者读取编码成二维条形码的URL。另一方面，信标定期发射实体的URL，通常通过（定向的）红外线而不是无线电，因为无线电通常是全方位的，因此会导致不确定哪个URL属于哪个实体。



图16-13 捕获和打印绘画作品的Web存在

特别地，Cooltown项目以小设备形式（仅几厘米长）开发信标。这些小设备每几分钟使用一种单程触发无连接协议通过红外线发射一个字符串（参见图16-13a）。发射的字符串是类似XML的文档，包括实体的Web存在的URL和一个简短的标题。许多便携设备（比如手机和PDA）都集成了红外线接收器，因此能够接收这样的字符串。当客户程序接收字符串时，它可以使用设备的浏览器直接连接到接收到的URL。也可以用接收到的标题创建一个到接收到的URL的超链接，并将该超链接加入到接收到的超链接列表中，之后用户可以点击相应的超链接。

间接感知：间接感知是用户设备从标记或信标中得到一个标识符，通过查询它得到一个URL。感知设备知道一个解析器的URL，这个解析器是一个名字服务器，它维护一个从标识符到URL（在第9章术语中叫名字上下文）的绑定集合，并返回绑定到给定标识符的URL[Kindberg 2002]。在理想情况下，实体标识符的名字空间将足够大以使得每个物理实体有一个唯一的标识符，从而消除了二义性。然而，原则上，本地标识符只要曾经被本地解析器查找过，就可以被使用。否则，可能得到伪造的结果，因为其他人可能为另一个实体使用了相同的标识符。

有时需要使用间接感知，因为标记技术中的限制意味着不可能直接感知URL。例如，线性条形码没有足够能力存储任意一个URL；便宜的RFID标记只能存储一个定长的二进制标识符。在这些情况下，必须查询存储的标识符以得到Web存在的URL。

不过，使用间接感知还有一个根本的原因：它允许给定的物理实体有一组Web存在而不是只有一个Web存在。正如短语“Hopper's painting Chop Suey”可能出现在多个Web网页上，而且链接到不同的Web网页，给定的物理绘画作品可能根据解析器的不同导致不同的Web存在。例如，某个作品的一个Web存在可以是一个到服务的链接，该服务是在博物馆中邻近的打印机上打印一个副本，该作品的另一个Web存在可以是一个提供该作品信息的网页，这些信息来自一个与博物馆无关的独立的第三方。

712
|
713

解析的实现遵循Web体系结构，其中每个解析器是一个独立的Web站点，客户软件是一个浏览器加上一个简单的插件。解析器提供Web表单（包括一个用户填写的域）作为感知产生的结果，而不是将域显示给用户让其手工填写。也就是说，当用户扫描条形码时，作为结果的标识符自动地填写到表单中，并且客户将表单发送给解析器。解析器返回相应的URL（如果存在）。

因为解析器本身也是Web资源，当它们可能是其他Web页面时用户可以导航到它们[Kindberg 2002]，并用客户使用的解析器更新信息。特别地，用户可能使用本地物理超链接得到本地解析器的URL。例如，博物馆的管理者可以建立Cooltown信标让其发射本地解析器的URL，这样访问者能够使用解析器得到博物馆内与绘画相关的Web存在。同样的，如果绘画作品的标识符是众所周知的，那么访问者就可以利用Web上其他地方的解析器。例如，一个西班牙的访问者在访问北美的一个博物馆时可能利用保存成书签的西班牙艺术博物馆站点的解析器。

最后，尽管我们已经指出间接感知相对于直接感知的几个优点，但它也有一个主要的缺点：客户连接到解析器会产生额外的往返，同时有延迟和能量消耗。

16.7.3 互操作和eSquirt协议

Web存在的目标设备和用户便携设备之间进行互操作的一种方法是使用标准Web协议。用户的便携设备发起一个HTTP GET或POST操作；目标设备以Web页面形式的用户界面作为响应，便携设备负责呈现这个Web页面。回到前面的例子，一个Web存在的因特网收音机可以通过面向用户的信标给出它的Web服务的URL。用户走到收音机前，将他们的便携设备（也就是PDA）上的红外线接收器指向它；PDA上的客户程序从收音机上接收到URL，并将URL传送到它的浏览器。结果，PDA上出现收音机的“主”页，带有控制面板用以调节它的音量、可以从PDA上载和播放声音文件，等等。

博物馆内的Web存在的打印机也具有类似的行为。用户通过打印机的信标得到打印机的主页，这样能够上载内容到打印机并通过Web页面指定打印机的设置。当然，设备（比如打印机）可能有物理用户界面，但是简单的应用（比如数码相框）则可能没有，那么虚拟的用户界面就是最基本的了。

以上互操作的形式是面向数据的并且是与设备无关的，这大体上与Web类似。因为目标设备提供了自己的用户界面，所以用户可以通过他们的浏览器控制设备，而不需要目标特定的软件。例如，PDA上带有图像文件的用户可以在任何图像呈现设备上呈现它，不论是打印机还是数码相框都可以；PDA上带有声音文件的用户可以在任何音频播放设备上收听该文件，不论它是一个因特网收音机还是一个“智能”HiFi系统。

714

那些场景的问题在于用户的便携设备的资源相对匮乏——它可能有低带宽的无线连接——处在内容源和目的地之间的内容传递路径上。假设用户在安装了Cooltown的博物馆中已得到绘画作品的一个图像，或者已得到某人评论该作品的一段音频剪辑。在这种情况下，可以将16.6节介绍的自适应技术运用到有限的资源（比如屏幕大小和带宽）上，从而在便携设备上得到图像或音频剪辑的较低失真的版本。当用户将图像传送到博物馆的打印机，或者将声音剪辑传送到宾馆房间的网上电台，它们的质量将会降低，即使那些设备有能力进行高质量的渲染，并且可能有高带宽的有线网络连接。

用于设备间互操作的Cooltown eSquirt协议解决了低失真的问题，并避免了宝贵的带宽和能量消耗，方法是将内容的URL从一个设备传送到另一个，而不是传送内容本身。事实上，该协议与通过红外线将URL（和标题）从Cooltown信标发送到设备的协议相同（参见图16-13b）。设备通过低能量红外线介质传送少量数据，并且这是eSquirt协议唯一的网络操作，所有设备都涉及该操作。然而，接收设备可以作为Web客户使用URL检索内容，并执行操作（比如呈现结果数据）。

例如，用户从霍珀的作品旁的信标得到该绘画作品的URL，通过使用具有eSquirt功能的PDA可将该URL发送到一个打印机。eSquirt使用的协议是不可靠的，但是，和一个电视远程控制一样，如果传输失败，那么用户可再次按下“squirt”按钮，直到打印机返回确认成功为止。之后打印机（更确定的说是打印服务，它可能在基础设施中实现）作为一个Web客户从URL中检索内容（以高保真的形式）并打印它。

这样，用户的便携设备可以作为该URL的一个与设备无关的剪切板，类似于桌面用户界面上的复制-粘贴操作中与应用无关的数据剪切板。用户采用设备在源和目的地之间“复制和粘贴”URL，从而在它们之间传送内容。

与设备无关是eSquirt范型最重要的优势。eSquirt协议经常以相同的方式工作，不同的是接收器对URL的处理过程。然而，用户必须对发射的URL和接收设备怎样结合才会有意义要有理性的认识。接收设备的设计者必须想到一些可能的错误：用户可能错误地将一个音频文件的URL发射到打印机。不过，我们不提倡在设计时就解决这些错误。采取预防措施（比如类型检查）可能导致我们在16.2.2节提到的丢失机会和脆弱的互操作现象。

虽然简单性是eSquirt协议的一个优点，但它的缺点是它依赖于接收设备的默认设置，或者依赖使用物理控制器来输入其设置。也就是说，eSquirt不能使用我们在本节开始提到的互操作范型。在该范型中，客户设备为控制目标设备的设置获得一个虚拟用户界面。例如，将一个声音文件或一个流式无线电电台的URL发射到网上电台后，用户如何用他们的便携设备控制音量？习题16.19将讨论这个问题。 715

16.7.4 小结和前景

我们已经概述了Cooltown体系结构的主要特点。该项目的目标是通过扩展Web（即超链接内容的一个虚拟集合）到物理世界中的实体，而不管那些实体本身是否有电子功能，从而方便游牧用户。体系结构考虑了物理实体（包括人、位置和物体）如何关联到Web存在。其次是物理超链接——从物理实体上感知Web存在的URL的体制。项目使用红外线信标、标记（比如条形码和RFID标记）和解析器将标识符转变为URL，从而实现物理超链接。最后，eSquirt是一个与设备无关的互操作协议，它将低功耗便携设备从内容源和目的地之间的内容路径中解脱出来。

Cooltown在很大程度上达到了它的目标，但只建立在人会“不断重复指令”的假设上。人通过物理超链接发现与他们遇到的与实体关联的服务。人可能也需要注册贴有标记的非电子实体的Web存在，比如，音乐CD，当这些实体被放在一个Web存在方式的位置（比如一座房子）的上下文中，这样它们在那就变成电子可发现的了。最终，人不但通过“点击”物理超链接将他们的便携设备关联到Web存在实体，而且通过eSquirt协议引入了与设备无关的互操作。人的参与增加了灵活性，并解决了丢失交互机会的问题。然而，在简单eSquirt互操作模型中，用户无法控制接收设备如何处理用户发送的URL。

另一项研究是关于自动关联和Web存在实体的互操作。每个物理实体可以有统一类型的Web存在实例，它将记录该实体的语义的细节（可能使用语义Web技术），包括该实体和其他实体之间的关系，特别是Web存在的人或物体和包括它们的Web存在位置之间的关系。因此，给定位置的Web存在可以互相发现并进行互操作。例如，在会议上，秘书的Web存在可以在会议室内发现需要打印的文档、发现出席的成员、发现附近的打印机并且打印一定数目的副本。Cooltown的Web存在管理器[Debaty and Caswell 2001]已经开始实现这个目标，不只是对地点而是对Web存在的物体和人统一管理，这些物体和人链接到相关的实体，比如他们所在的Web存在的地点。例如，当一个实体进入一个新位置并在那里注册时，实体用一个到它新位置的Web存在的链接自动地进行更新。理想情况下，实体关系将全部编程建立，而不是由当前可用的有限的支持来建立。但是，由于我们日常世界的复杂语义，使得以一种实际有用而没有错误的方式实现一个应用（比如自动会议支持）还有很长的路要走。同时，在交互中包括人为参与将可能取得进展。 716

16.8 小结

本章给出了移动计算机系统和无处不在计算系统的主要挑战，并给出了一些解决方案（因为可用的方案不多）。大多数挑战源于下面的事实：系统是易变的，这很大程度上是由于系统与我们日常物理世界集成在一起造成的。易变的系统是在给定智能空间中用户、硬件和软件组件集合会发生不可预测的改变的系统。当它们从一个智能空间移入另一个智能空间时或者由于失效，组件通常有规律地建立和破坏关联。连接带宽随时间会产生很大变化。组件可以因电量耗尽或其他原因而失效。16.1～16.3节全面地讨论了易变性的这些方面和一些用于关联组件并使得它们互操作（尽管有“不断的改变”的困难）的技术。

设备与我们的物理世界的集成涉及感知和上下文敏感（见16.4节），并且我们已经描述了处理感知数据的一些体系结构。但存在一个可以描述为*物理保真度*的挑战：带有感知和计算行为的系

统如何精确地按照我们以及与我们居住的物理世界联系的敏感语义运转？当我们在位置间移动时，上下文敏感手机是否能够真的如我们所希望的那样禁止响铃？在Cooltown中，位置（比如宾馆房间）的Web存在实际上是否记录了所有的Web存在实体，人可以说成是位于那个位置内——或者不是，例如，人在相邻房间内？

安全和私密性（见16.5节）成为移动系统和无处不在系统研究的一大特色。易变性使安全性问题更加复杂，因为它避开了在想要建立安全通道的组件间有什么可信赖的基础的问题。幸运的是，物理受限通道的存在对构建有人存在的安全通道有一定作用。物理集成对私密性有影响：如果对用户进行跟踪并提供给他们上下文敏感服务，那么可能导致严重的私密性丢失。我们描述了一些用于标识符管理的方法，概述了用于减少该问题的统计技术。

物理集成也意味着对设备能量、无线带宽和用户界面等因素的新的限制，传感器网络中的节点对前两个因素有很少要求，对最后一个因素则没有要求；手机对三者都有要求，但仍比桌面机器少很多。16.6节讨论了一些体系结构，在这些体系结构中，组件能够适应资源限制。

16.7节将Cooltown项目作为一个实例研究，描述了其体系结构。该体系结构独特之处在于它将从Web中学到的经验运用到无处不在计算中。它的优势是高度的互操作性。但是，Cooltown主要用于人监控交互的情况。

最后，本章重点讨论了移动系统和无处不在系统与本书其他章节介绍的更传统的分布式系统的差异，主要集中在易变性和物理集成方面。习题16.20将请读者列出它们的一些相似点，并考虑
717 采用哪种传统分布式系统解决方案的扩展。

练习

16.1 什么是易变系统？列举易变系统中会发生的变化的主要类型。（第661页）

16.2 讨论是否可能通过组播（或广播）和缓存对查询的应答改进服务发现的“拉”模型。（第671页）

16.3 解释为什么要在发现服务中使用租期以便处理服务易变性问题？（第671页）

16.4 Jini查找服务基于属性或Java类型以提供匹配用户请求的服务。举例说明这两种匹配方法的差异。这两种匹配各有什么优势？（第672页）

16.5 描述允许客户和服务器定位查找服务器的Jini“发现”服务中IP组播和组名字的使用。（第672页）

16.6 什么是面向数据编程？它与面向对象编程有何不同？（第677页）

16.7 讨论下列问题：事件系统的范围应该如何联系到使用它的智能空间的物理范围。（第679页）

16.8 比较和对比智能空间基础设施中与事件系统和元组空间相关的持久性需求。（第679页）

16.9 描述感知显示器旁用户存在的三种方法，从而给出上下文敏感系统的体系结构的一些特色。（第684页）

16.10 解释无线传感器网络的网络内处理。（第688页）

16.11 在Active Bat定位系统中，为得到一个三维位置默认情况下只使用三个超声波接收器，而在卫星导航中为得到一个三维位置却需要四颗卫星。为什么会有这样的差异？（第693页）

16.12 在一些定位系统中，被跟踪的对象把它们的标识符送交基础设施。解释这如何引发了对私密性的关注（即使标识符是匿名的）？（第695页）

16.13 许多传感器节点遍布一个区域，节点进行安全地通信。解释密钥分发问题并概述一种用于分发密钥的概率论策略。（第697页）

16.14 我们描述了几种为安全自发设备关联提供接收受限通道的技术。其中哪些技术也提供了发送受限通道？（第699页）

16.15 说明如何从一个接收受限通道构建一个发送受限通道，反之亦然。（提示：使用一个连接到

给定通道的可靠节点。） （第699页）

16.16 一组智能空间只通过它们之间的空间（比如走廊或广场）相连。讨论判断该中间空间是否可以作为一个混合地带的因素。 （第704页） 718

16.17 解释采用多媒体内容时要考虑的上下文因素。 （第705页）

16.18 假设使用无线电在100米距离发射或接收1K比特数据所消耗的能量可以使设备执行300万条指令。设备可以选择发送100K字节的二进制程序到100米远的计算服务器上，服务器在运行时将要执行600亿条指令，并与设备交换10000 1K比特的消息。如果只考虑能量，设备应该卸载计算还是自己执行计算呢？假设在卸载情况下忽略设备的计算。 （第708页）

16.19 一个Cooltown用户将声音文件或流广播站的URL发送给网上电台。建议对eSquirt协议进行修改，使得用户可以使用他们的便携设备控制音量。（提示：考虑发射设备应该另外提供什么样的数据。） （第715页）

16.20 讨论将以下领域的技术运用到移动和无处不在系统的适用性：（1）点对点系统（第10章）；（2）协调和协定协议（第12章）；（3）复制（第15章）。 （第717页） 719

第17章 分布式多媒体系统

多媒体应用程序实时生成和应用连续的数据流。它们包含大量的音频、视频和其他基于时间的数据元素，并且及时处理和发送单独数据元素（音频采样、视频帧）是非常重要的。延迟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约通常包含如下部分：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时（延迟）以及数据元素的丢失率。在交互式应用程序中，延迟是特别重要的。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求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（用来缓冲那些提前送到的数据元素）都很重要。系统根据应用对服务质量的请求来分配上述资源。一个成功的QoS请求向应用程序发送一个QoS保证，并且将被请求的资源预留，以便日后进行调度[⊖]。

720
~
721

17.1 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连续的、基于时间的数据流。这种处理能力促进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通用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意。当前的网络和分布式系统技术不能实现许多对实时数据要求很高的应用程序（例如，大范围的视频会议、数字电视产品、交互式电视以及视频监视系统）。

多媒体应用程序需要及时地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应用。同时对于应用程序的完整性而言，及时地传输数据元素（音频采样、视频帧）是非常重要的。简单地说，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]），但是它们都没有被集成为一个通用的操作系统和网络。航空电子设备、航空控制、制造过程控制和电话交换这些现有的实时系统所执行的任务的本质和多媒体应用程序执行的任务的本质不同。前者通常处理的数据量比较小，并且*硬时间限制*相对较少，但是如果超过这个时间限制，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，以保证在最坏的情况下也能满足其需要。

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，特别是在视频点播服务、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也会带来严重的后果。但是这些多媒体应用与其他实时应用程序的需求相比有很大差别：

- 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

⊖ 本章参考了Ralf Herrtwich[1995]的文章，并引用了其中的材料，在此对Ralf Herrtwich提供的帮助表示感谢。

站和服务器上，它们要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。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）需要显示的视频数据流的数目会发生变化。多媒体应用程序可能涉及其他变化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仿真活动。 722
- 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，或者在参加会议时能同时运行一个字处理程序，用户会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17-1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，例如桌面会议或者提供对已保存的视频片段的访问、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。我们将采用资源考虑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（网络、中央处理器、磁盘子系统）可提供的能力。

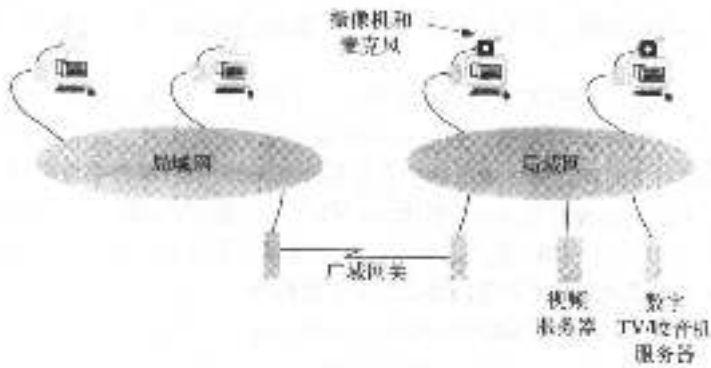


图17-1 一个分布式多媒体系统

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 723
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。QoS管理用于保证应用程序能在所需的时间内获得必要质量的资源，甚至当其他应用程序竞争资源时，也能保证这一点。

一些多媒体应用程序甚至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的计算机系统上。它们包括：

基于Web的多媒体：这些应用程序最大限度地访问通过Web发布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不同地点上的同步的数据流。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能力不强都限制了它的性能。在音频和低质量的视频应用中，系统使用目的地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，但是资源到达目的地的延迟可能会长达数秒。

网络电话和音频会议：这种应用程序对带宽的要求相对较低，特别是使用有效的压缩技术后这一点更为明显。但是它的交互性本质要求往返延迟比较小，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这些数据传输到终端用户的显示器上。当有充足的网络带宽及视频服务器和接收站是专用的时，这些应用可被成功执行。它们也要在目的地采用相当大的缓存。

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）。它们的应用环境和场景各种各样，例如：

- 因特网电话，稍后详述。
- 一个简单的涉及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]、NetMeeting[www.microsoft.com III]）、iChat AV[Apple Computer 2004]），但它的性能受到带宽和延迟的限制。
- 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。



724

这样的应用程序有如下需求：

低延迟的通信：往返延迟为100～300ms，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止在给定的帧上，那么其他用户也应该看到视频在该帧上停止。

媒体同步：音乐合奏的所有参与者都应该几乎在同一时间（Konstants等[1997]指出它的同步需要限制在50ms内）内听到其演奏。独立的声道和视频流应该保持"音唇同步"，例如，当用户对于一段视频回放进行实时评论时，或者用户在使用非本地的卡拉OK伴唱应用时。

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。

因特网电话——VoIP

因特网并不是为实时交互应用（例如电话）设计的，但是随着因特网核心组件——主干网的速度达到10～40Gbps，而且连接主干网的路由器也有相当强的性能——的功能和性能增强，因特网已经可以为实时交互应用提供服务。因特网核心组件经常运行在较低的负载下（<10%带宽利用率），而且很少因资源竞争造成IP传输延迟或丢失。

这样，就可以先将数字化的声音采样流作为没有服务质量要求的UDP分组，将其从声音源传输至目的地，从而实现在公共因特网上构建电话应用。Voice-over-IP（VoIP）应用，例如Skype和Vonage，就是基于这项技术；体现声音特点的即时通信应用，例如AOL Instant Messaging，Apple iChat AV和Microsoft NetMeeting也是基于此项技术。

当然，那些都是实时交互应用，因而延迟性是一个很重要的问题。正如在第3章讨论的，IP分组在经过每一个路由器时，都不可避免地发生延迟。在路径较长的情况下，这些延迟的积累很容易超过150ms，于是用户在对话交互中，便可以察觉这种延迟。正是由于这个原因，长距离的（尤其是洲际间的）因特网电话服务产生的延迟会比使用常规电话网络的电话服务的延迟大得多。

然而，更多的声音传输仍然是通过因特网进行的，而且其与传统电话网的整合也正在进行中。SIP（会话初始化协议，RFC 2543定义[Handley et al. 1999]）是一个在因特网上建立声音电话（以及其他服务，例如即时通信）的应用层协议。在世界各地，都有连接常规电话网络的网关，它们允许从连接在因特网上的设备发出的呼叫，通过因特网的传输，到达常规电话或个人电脑。

匮乏区 许多当今的计算机系统都提供了处理多媒体数据的能力，但是所提供的资源还非常有限。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（参见图17-2）。在到达匮乏区之前，系统用于执行相关应用程序的资源并不充足。在20世纪80年代中期前，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，系统就可以有充足的资源来提供此服务，甚至在最坏的情况下和没有专门的管理机制下也是如此。

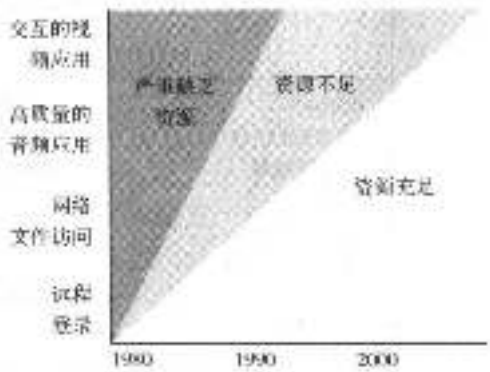


图17-2 计算和通信资源的匮乏区

系统性能的最新进展主要在于提高多媒体数据的质量，以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，一些对多媒体数据质量要求极高的应用，例如虚拟现实和实时流处理（"特殊效果"），对于多媒体数据质量的要求几乎是无止境的。

17.2节我们将回顾多媒体数据的特征，17.3节将介绍为了实现QoS管理而采取的匮乏资源的分配方法，17.4节讨论资源调度方法，17.5节介绍在多媒体系统中优化数据流的方法。17.6节介绍Tiger视频文件服务器，这是一个用于将视频流并发地传输到大量客户的系统，该系统的可扩展开销很低。 725 ~ 726

17.2 多媒体数据的特征

我们已经说过，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到的数据特征。实际上，连续的媒体是由一系列离散值组成的，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更新8000次。

因为音频和视频数据流中的定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（或者时）的。由于数据值被播放和记录的时间会影响数据的正确性，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数据吞吐量，图17-3给出了一些常用的数据速率和帧/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，必须消耗较大的带宽。例如，一个标准的TV视频数据流需要120Mbps以上的带宽，它超过了100Mbps以太网所能提供的带宽。它对CPU处理能力的需求也随之扩大。例如，对标准TV视频数据流的每一个数据帧进行拷贝和简单转换的程序至少要消耗一个400MHz CPU处理能力的10%。在处理高清电视数据流，这个数字会更高，并且许多像视频会议这样的应用程序需要同时处理多个视频和音频流。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。

压缩可以将对带宽的需求减少到原来的1/100～1/10，但它不会影响连续数据的时序需求。为了设计出高效、通用的多媒体数据流表示和压缩方法，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]，

并且站点[Multimedia Directory]给出了当前多媒体标准的文档以及其他资源。

	数据速率（近似值）	采样或帧大小	频率
电话交谈	64kbps	8比特	8000/秒
CD质量的声音	1.4Mbps	16比特	44000/秒
标准TV视频（未压缩）	120Mbps	最高640×480像素×16比特	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）	1000～3000Mbps	最高1920×1080像素×24比特	24～60/秒
HDTV视频（用MPEG-2压缩）	10～30 Mbps	可变的	24～60/秒

图17-3 典型多媒体数据流的特征

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处理资源的负担。系统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功能的增强，系统可以用软件编码和解码过滤器来完成上述功能。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提供了更好的支持，所以具有更好的灵活性。

用于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，通常是由硬件编码器来执行压缩，而由软件对到达每个用户计算机的多个数据流解压，这样可以不必考虑每个用户计算机上的解码器的个数，而会议的参与者数目可以动态地变化。

17.3 服务质量管理

当多媒体应用程序运行在由个人计算机组成的网络上时，它们需要对位于应用程序的工作站（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）上的资源进行竞争。工作站和网络必须同时支持多个多媒体程序和传统应用程序。在多媒体应用程序和传统应用程序间有竞争，在不同的多媒体应用程序之间甚至在单个应用程序的多媒体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，用于不同任务的物理资源是可以并发使用的。在多任务操作系统中，中央处理器轮流执行每一个任务（或进程），或者采用某种基于“尽可能优化调度的”调度方案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。

727～728

网络用来使不同来源的消息交织在一起传输，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佳方式来管理共享的传输介质。当介质空闲时，任何节点都可以使用它。但是可能会发生分组冲突，当发生冲突时，节点会等待随机的一段时间，然后重发分组，以便防止冲突。当网络负载很重时，很容易发生冲突，这种方案在这种情况下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方法和其他最优方法都不能满足多媒体应用程序的需求。显而易见，及时地处理和传输多媒体数据流对它们而言是非常关键的。延迟的传输数据是没有价值的。为了实现及时传输，应用程序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称为*服务质量管理*。图17-4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。白色方框表示其资源需求会影响应用程序的服务质量的软件组件。

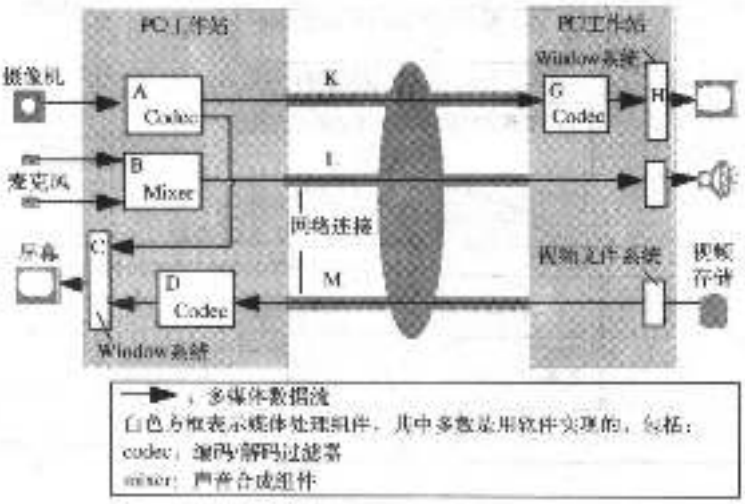


图17-4 多媒体应用程序典型的底层组件

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。这些进程间的连接使多媒体元素的源端和目标端连接在一起，在目标端，多媒体数据被输出或者被消耗。进程间的连接可以由网络连接实现，当源和目标端的进程位于同一台计算机上时，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。 729

在图17-5中，我们列出了图17-4中主要的软件组件和网络连接所需要的资源（两幅图中的字母是相对应的）。显然，系统中需要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器。

组件	节点	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比特	—	零	—
A Codec	输入：10帧/秒，原始视频 输出：MPEG-1数据流	交互的	低	每100ms需CPU10ms， 10MB RAM
B Mixer	输入：2×44kbps音频 输出：1×44kbps音频	交互的	很低	每100ms需CPU1ms， 1MB RAM
H窗口系统	输入：可变 输出：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	交互的	非常低	44kbps，非常低丢失率 的数据流协议

图17-5 图17-4中应用程序组件的QoS规约

图17-6以流程图的形式说明了QoS管理器的职责。在下面的两小节中，我们将介绍QoS管理器

的两个主要任务：

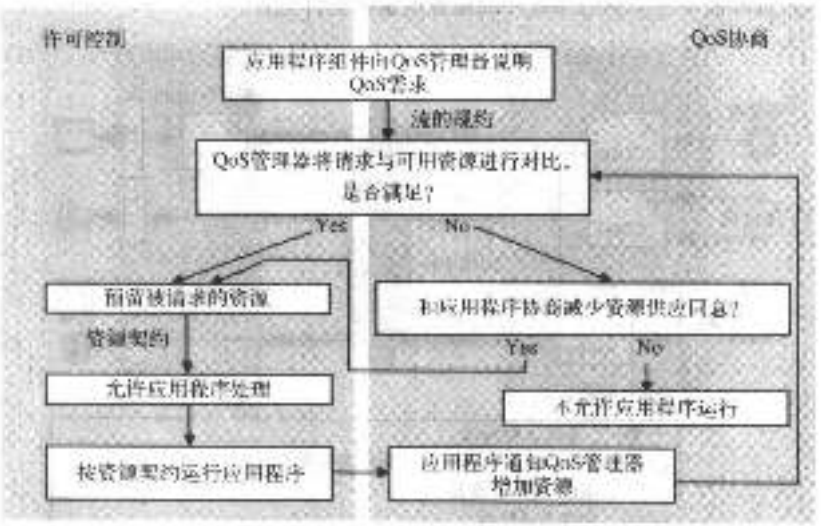


图17-6 QoS管理的任务

服务质量协商：应用程序向QoS管理器提出自己的资源需求。QoS管理器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。如果答复是否定的，那么应用程序会被重新配置，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，则预留被请求的资源。同时应用程序获得一个资源契约。用于说明被预留使用的资源。该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管理器。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。

在本节的最后，我们将进一步描述执行这些任务所使用的技术。当然，当一个应用程序运行时，它需要细粒度的资源调度（如处理器时间和网络带宽），以保证实时进程能及时接收到已分配的资源。17.4节将介绍这些技术。

17.3.1 服务质量协商

为了在应用程序和底层系统之间进行QoS协商，应用程序必须向QoS管理器说明自己的QoS需求。这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，有三个参数非常重要，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指数据流过的速度。

延迟：延迟是指单个数据元素从源端传输到目的端的时间间隔。当系统中的数据量大小以及系统负载中的其他特征变化时，延迟也会发生变化。这种变化被称为抖动，抖动是延迟产生的第一个问题。

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，数据元素将被丢掉。在管理良好的QoS环境中，这种情况是不会发生的。但是因为前面所说的原因，现在很少有这样良好的环境。而且，保证每个数据元素都能及时传输所耗费的资源通常是难以承受的——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这个数字会更低。

这三个参数可以用来：

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，并且因为用于会议应用程序，为了避免会话间隔，传输延迟最多为150ms。在丢失率小于1%时，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，一个网络可以提供带宽为64kbps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：

- 现代系统的丢失率很少与因为噪声、失效等引起的比特错误相关，而是与缓冲区溢出和与时间相关的数据延迟有关。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应越大以避免信息丢失。缓冲区越大，就可能有更多的信息等待被服务，因此，系统就可能有更大的延迟。

为数据流设定QoS参数 QoS参数的值可以显式地给出（例如，图17-4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，对于网络连接K，输入数据流的带宽等于对摄像机输出流采用MPEG-1压缩而得到的结果）。

但是更常见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，压缩比会动态变化。例如，在内容变化很快时，需要的带宽会很高。因此，通常以最大值、平均值和最小值三种类型的值来表示QoS参数，选择哪种值依赖于当前使用哪种QoS管理制度。

与带宽规约相关的另一个问题是网络的*数据爆发*。假设有三个1Mbps的数据流。其中一个数据流每秒传输一个1M比特的帧，第二个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，第三个数据流每微秒发送100比特的声音采样信号。尽管这三个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。这个爆发参数指定可能提前到达（也就是说，它们先于平常的速率到达）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内的数据流最大消息数目定义为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以块方式传输，而且从网络接收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：如果不能以和数据流传输同样的速度在到达点处理数据帧，则等待处理的数据会越来越多，最终会超出缓冲区的容量。为了避免这个问题，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，缓冲区就可能出现数据堆积。如果发生了数据堆积，除了处理和传播时间之外，积压数据的数目和大小也会影响数据流端到端延迟的最大值。

另一种延迟需求来源于应用程序环境。在会议应用程序中，为了达到参与者之间的即时交互，数据流端到端的延迟应不超过150ms，这样用户在谈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。

关系到多媒体数据传输延迟的第三种情况是抖动——传输两个相邻帧的时间间隔的变化。尽管

大多数多媒体设备都确保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上面提到的种种考虑限制的。所以媒体序列的回放要求媒体数据元素必须在固定的时间限制内到达。

730 ~ 733

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消息的概率统计。这种计算要么基于最坏情形的假设，要么基于标准分布。这两种方法都不能很好地反映实际情况。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，它们的差别可能就会很大。

在带宽参数说明中，不仅仅是在一段时间内发送的数据总量很重要。在这段时间间隔内数据的分布也很重要。丢失率参数说明需要确定数据丢失的时间间隔的期望值。特别的，在无穷长时间间隔内考虑丢失率是没有用的，这是因为在一个短期内丢失的数据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：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。这一特点就会越明显。

带宽的LBAP模型要求对多媒体数据流的爆发进行管理。通过在源端加入一个缓冲区并定义数据元素离开缓冲区的方法，该模型可管理任何数据流。漏桶图（见图17-7a）形象化地说明了这种方法：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速率不超过R，缓冲区B的大小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。

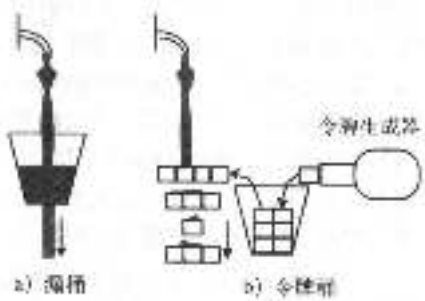


图17-7 流量调整算法

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（参见图17-7b）通过在数据流空闲时允许较大的数据爆发来做到这一点。它是漏桶算法的变种，其中发送数据的令牌以固定的速率R生成。令牌被收集到一个大小为B的桶中。只有当桶里至少有S个令牌时，大小为S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。

734

在令牌桶系统中，仅当数据流空闲了一段时间后，大小为B的传输高峰才会出现。为了避免数据爆发，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必须远大于R。它的唯一目的是分解大的数据爆发。

流规约 QoS参数的集合通常称为流规约。现有的几个流规约都比较相似。在因特网RFC 1363[Partridge 1992]中，流规约被定义为11个16位的数字值（参见图17-8），它以下面的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数据流所需要的最大带宽。
- 令牌桶大小和速率决定数据的爆发量。
- 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丢失总数和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，通过一个最大超前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协议规约[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，电话质量和高保真音频，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，当其中一些通信类型混合时，系统也可以被配置。

	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
	可以被发觉的爆发丢失
	丢失间隔
	服务质量保证

图17-8 RFC 1363的流规约

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节点上。在每个节点上有一个QoS管理器，进行一个QoS协商的最简单办法是跟随从源端到目的端的数据流。源端组件通过向本地QoS管理器发送一个流规约来启动协商过程。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其他系统，那么流规约被传送到下一个请求资源的节点。这一流规约遍历所有的节点，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节点上的并发QoS协商之间可能会发生冲突。为了彻底解决这种问题，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的方式，另一种更好的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以接受。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带宽的情况下它也能处理；或者延迟应该为200ms，但300ms是它可接受的最坏情况。因为一次只能优化一个参数，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，并由系统来优化第二个参数。

如果一个数据流包含多个槽，系统将根据数据流分支确定协商路径。作为以上方案的扩展，中间节点可以聚集来自目的端的QoS反馈消息并生成QoS参数在最坏情况下的值。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过程。

在目的端是异质的情况下，通常不适合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。这就要求由接收端发起协商，而不是由发送方发起协商过程。RSVP[Zhang et al. 1993]就属于这一类QoS协商协议，其中由目的端连接数据流。源端将现有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节点，并从那里获得数据。为了使它们能获得适合QoS的数据，它们可能需要使用过滤（见17.5节）这样的技术。

735
↓
736

17.3.2 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。如果新的多媒体数据流的资源需求违反了已有的QoS保证，它涉及关闭服务请求。

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约会反映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。相应的，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，那么许可控制是非常简单的。如果资源分布在不同节点上，例如许多局域网环境，那么可以使用一个集中式访问控制实体，也可以使用一个分布式的许可控制算法来避免并发许可控制的冲突。工作站的总线仲裁算法就属于这一类算法。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级别的一个常用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。这是给提供应用程序有保障的QoS的唯一可能的方法——至少不会发生灾难性的系统故障。在应用程序不能适应不同级别的QoS或者当其质量下降使程序不可用的情况下，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光视频中，某症状图像可能正好位于丢失的帧中）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是非常简单的：当控制访问一个带宽为B的网络时，仅当$\Sigma b_s \leqslant B$时，带宽为b_s的多媒体数据流s才能允许访问。这样，一个16Mb/s的令牌环网可能支持10个1.5Mb/s的数字视频流。

遗憾的是，容量计算并不总是和在网络中一样简单。为了以同样的方式分配CPU带宽，系统需要知道每个应用程序进程的执行概貌。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。虽然存在几种自动计算执行时间的方法[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，但它通常有很大的错误率并且移植性很有限。

对于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得多。预留最大带宽方法可能会导致带宽资源的浪费：尽管新的资源申请可以被满足，即，使用已被保留
737 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（经常很高）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，所以统计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，那么瞬时的负载高峰可能会导致服务质量的下降；应用程序必须能处理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较小的数据量，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。

正像试验所显示的那样[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。

17.4 资源管理

为了向应用程序提供一定等级的QoS服务，系统不仅要有充分的资源（性能），它还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本质。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，因此在不会造成离散媒体访问和其他交互应用程序饥饿的情况下，如何为依赖时间的数据流提供充分的服务是一个巨大的挑战。

调度方法必须应用到（并协调）影响多媒体应用程序性能的所有资源。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，并最终显示出来。在这个例子中，系统需要的资源包括磁盘、网络、CPU以及内存和总线带宽。

738

公平调度 如果有多个数据流竞争同一资源，系统必须考虑公平性，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，这一方法是基于包而提出的。在Demers等[1989]中，这种方法是基于比特提出的，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比特进行发送，但是在给定的帧速率并且要求必须完全发送的情况下，系统可以计算用于每个包的时间。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基于比特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，这时它会阻塞小数据包的传输，但这违背了基于比特的调度方案。然而，任何包的延迟都不会高于最大包的传输时间。

所有的基本轮转方案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，可以将基于比特的方法进行扩展，一些数据流可以在每个周期传输更多的比特。这种方法称为基于权值的公平排队。

实时调度 人们已经开发出一些实时调度算法来满足一些应用程序（如航空工业中过程控制）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，而这种分配方式必须保证进程能及时完成任务。

传统的实时调度方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）调度方法可以作为这些方法的代名词。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调度方法被证明在基于时序标准分配资源方面是最优的：如果系统中有一个调度满足了所有的时序需求，那么EDF调度将能找到它[Dertouzos 1974]。

EDF调度方法需要在每个消息（即每个多媒体元素）上进行调度决策。如果它用于调度生存期较长的元素会更有效。速率单调（RM）调度方法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调度方法是最优的[Liu and Layland 1973]。使用这样的分配方案，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，但是只有在一条消息的正常到达时刻才对这一消息使用EDF调度。

739

17.5 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的性能，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，但是收听者会立刻发现这种丢失。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从错误中恢复，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丢失，那么会发生延迟。对于非交互式的程序而言，这是可以接受的，尽管它可能会因为数据累积在源端或目的端最终导致缓冲区溢出。对于会议和其他交互式的应用程序而言，延迟增加是不可接受的，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输出。

17.5.1 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非但没有被减少，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瓶颈前就将其调整到可用带宽可以承受的范围。这称为流调整。

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压缩数据的难易程度。如果数据流已经被解压缩，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重采样。这个例子也说明，不同的流调整方法在不同的粒度上工作。

下列的流调整方法适合视频数据流：

时态调整：通过减少一个时间间隔传输的视频帧的数目，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可独立访问的视频数据流，它很难处理Delta压缩技术生成的数据流，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个图像的像素数。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视频。因此，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视频数据流。JPEG和MPEG-2支持不同的空间图像分辨率，因此它们适合使用这种调整方法。

740

频度调整：改变应用于每个图像的压缩算法。这样会导致一部分图像质量的损失，不过在通常的图像中，可以在察觉到图像质量降低前，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，所以即使图像内容发生变化，也可以保持恒定的吞吐率。

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位于目的端的监视进程和一个位于源端的调整进程组成。监视进程监视数据流中消息的到达时间。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消息，源端就减少数据流的带宽。过一段时间以后，源端将数据流向上调整回原来的水平。如

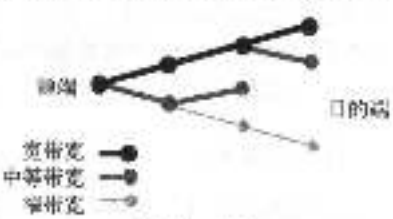


图17.9 过滤

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]。调整系统的问题包括：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。

17.5.2 过滤

流调整在源端改变了数据流，但它并不适合于涉及多个接收者的应用程序：当瓶颈出现在源端到其中一个目的端的路径上时，目的端向源端发送一个向下调整消息，这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尽可能好的QoS，它是通过在从源端到目的端的路径的相关节点上采用流调整技术（参见图17-9）来实现的。RSVP[Zhang et al. 1993]是支持过滤的QoS协商协议 741
的一个例子。过滤将一个数据流分解为一个层次性的子数据流集合，其中每一个子数据流增加更高级别的质量。路径节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。如果一个中间节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节点上就不会被过滤。

17.6 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原型系统，并且其中的一些已经成为实际产品（见[Cheng 1998]）。其中一个最先进的系统是Tiger视频文件服务器，它是由Microsoft研究实验室开发的[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：这一应用程序是向付费的用户提供电影的服务。系统从大容量的数字电影库中选择电影。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。尽管库中电影的数量很多，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地访问，这就可能导致同时播放它们，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。
- 可伸缩性和分布性：目的是设计一种具有可扩展的体系结构的系统，它（通过增加计算机）可以同时支持10000个客户。
- 硬件成本低：这个系统是由低价的硬件构建的（具有标准磁盘驱动的“大众用的”PC机）。
- 容错性：在单个服务器计算机或者是磁盘驱动器发生故障时，系统可以继续运行并且执行性能不会明显下降。

总而言之，这些需求要求有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17-10中显示了Tiger系统的硬件体系结构。所有的组件都是可以从市面上买到的产品。图中所示的cub计算机是包含同样数目的标准硬盘驱动器（通常是2～4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。它不负责处理多媒体数据，其职责只是处理客户请求和管理cub计算机的工作调度。

存储组织 主要的设计问题是：如何在cub计算机的磁盘上分布视频数据，以实现计算机共享负载的目的。因为系统负载涉及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。这可以通过复制数据的镜像方法和容错机制来实现，下面将介绍这一内容：

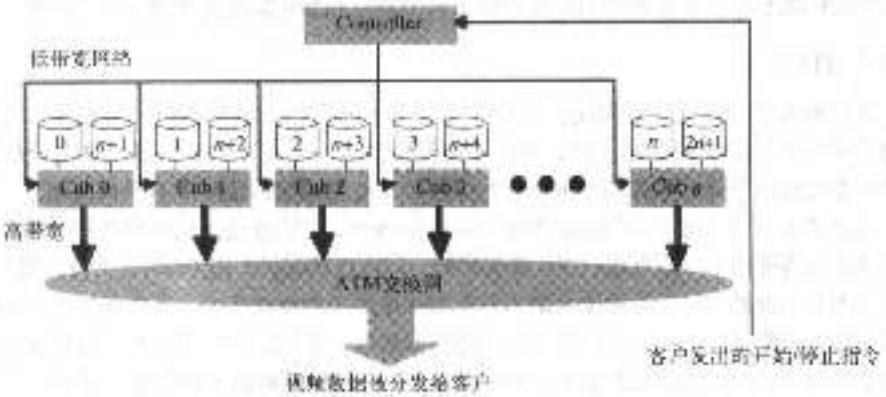


图17-10 Tiger视频文件系统的硬件配置

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1秒左右，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17-10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。如果当前数据块存储在编号最大的磁盘上，则下一数据块会被存储在第0号磁盘上，然后按顺序存储。

镜像：镜像方案将每个数据块划分为几个部分，称为二级备份。当一个cub计算机失效后，这样做能保证本来由此cub计算机提供块数据的任务被分配到其他的cub计算机上。每个数据块上二级备份的数目取决于数列因子d，它的值通常在4～8之间。存储在磁盘i上的数据块的二级备份被存储在磁盘$i+1$到$i+d$上。注意，只要系统中的cub计算机的数目多于d个，那么这些磁盘中没有一个和磁盘i属于同一个cub计算机。如果数列因子的值为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的资源应该足够处理出现故障的任务。

742
~
743

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的客户（被称为收看者）只能收到一个槽的信息，并且每个被占据的槽表示收看者正在接收实时视频数据流。在调度表中，收看者的状态通过以下信息表示：

- 客户计算机的地址。
- 被播放文件的标识。
- 文件中收看者的位置（数据流中下一个要传送的数据块）。
- 收看者播放的序号（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17-11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，其值通常是1s，并且假设对所有存储的电影来说，该值都是相等的。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，由客户计算机上的可用缓冲大小来决定系统可以允许的抖动。

每一个cub计算机维持一个指针，该指针指向每个磁盘的调度表。在播放数据块时，cub计算机必须处理属于它控制的磁盘上的数据块号所对应的槽，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：

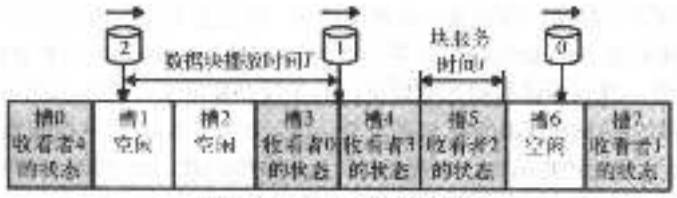


图17-11 Tiger系统的调度

1）将下一个数据块从磁盘上读出，放在此cub计算机的缓冲区内。

2）将此数据块打包并记下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）更新调度表中收看者的状态，这样可以给出下一个数据块和播放序号。然后将更新的槽传送给下一个cub计算机处理。

假设这些活动的最大时间开销为t（称为块服务时间）。如图17-11所示，t实际上小于磁盘块的播放时间。t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，直到到达下一个播放时间为止。实际上，磁盘并没有为数据块提供固定的延迟时间，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/t的结果通常大于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系统大约同时可以传送70个视频数据流。

744

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（磁盘驱动器或cub计算机）的故障都可能导致系统不能为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计算机发生故障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。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个磁盘的工作负载。镜像收看者状态与普通的收看者状态类似，但是它具有不同的磁盘号和时序需求。因为这一额外的工作负载被d个磁盘和d个计算机共享，所以如果在调度表中发现有空余能力，这项任务最好就利用这一空余，而不要打扰其他槽的任务。一个cub计算机的故障与它所有的磁盘的故障相似，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，就依赖于ATM网络协议（参见3.5.3节）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，而另一个数据块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号，并将它们传送到显示处理程序上。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。

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某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可视反馈。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。

745

在Tiger系统的最初实现中，controller计算机处理调度表的管理和发布。因为这会导致单点故

障以及潜在的性能瓶颈，后来，调度表管理被设计为一个分布式算法[Bolosky et al. 1997]。根据controller计算机发出的命令，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。

性能和可伸缩性 这一系统的初始原型在1994年被开发出来，使用了5个133MHz的奔腾PC，每个PC上装有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。通过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系统的数据传输性能相当不错——没有数据块被丢失或被延迟。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%，这在设计所容忍的范围内。

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数据块所在的磁盘最近的一个空闲槽。启动延迟度量值在2～12s范围内。最近已经研究出一个新的槽分配算法[Douceur and Bolosky 1999]，它能降低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，但后来在有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计算机都正常工作时，系统可以同时发送602个2Mbps的数据流，丢失率小于1/180000。当有一个cub计算机失效时，数据丢失率小于1/40000。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30000～40000个收看者。

17.7 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，同时它们必须对这些资源的使用进行调度，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，决定是否有足够的非预留资源来满足新的请求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，同时系统给应用程序发送一个保证信息。

系统必须调度分配给应用程序的处理器的处理能力和网络带宽以满足应用程序的需要。系统需
746 要像最早时间限制优先和速率单调这样的实时处理器调度算法以保证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，它的目的是消除数据流中数据元素之间的间隔时间的不均匀，而这种不均匀是不可避免的。通过减少源端的输出带宽（调整）或减少中间节点的带宽（过滤），数据流可以被调整以适应较少的资源。

Tiger视频文件服务器是一个很好的可扩展系统，它能为大范围的用户提供具有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

17.1 简要描述一个支持分布式音乐演奏的排练系统。请对这一系统的QoS需求以及硬件和软件配置提出建议。 （第724页，第730页）

17.2 当前因特网没有提供资源预留和服务质量管理设施。现有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？ （第724页，第734页，第740页）

17.3 请说明分布式多媒体应用程序可能需要的三种形式的同步（分布式状态同步、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而言，请对其实现同步的机制提出自己的建议。（第725页）

17.4 假设有一个由ATM网络连接多个桌面计算机以支持多个并发的多媒体应用程序的系统，简要描述其QoS管理器的设计。为你描述的QoS管理器定义API接口，同时给出其操作以及相应的参数和可能返回的结果。（第730～732页）

17.5 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？（第730～732页）

17.6 当有大量的客户同时访问同一部电影时，Tiger系统如何处理？（第742～746页）

17.7 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的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。（第742～746页）

17.8 当Tiger系统要对一个失效的磁盘或cub计算机进行操作时，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）。数据块的大小改变后，系统如何适应？（第745页）

第18章 分布式共享内存

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。开发DSM的动机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，程序员不需要对数据项运行编码。

实现DSM的一个核心问题是：在系统包含大量的计算机时，它如何获得良好的性能。对DSM的访问可能会涉及网络的通信。竞争同一数据项或相邻数据项的进程可能会产生很大的通信量。通信量和DSM的一致性模型紧密相关。当一个进程读取DSM中的数据时，一致性模型决定在多个写入数据中选择返回哪一个值。

本章将讨论DSM中的设计问题（像一致性模型）以及实现问题（例如，在写一个数据拷贝时，同一数据的其他拷贝哪些将失效，哪些将被更新）。接着本章还将详细讨论失效协议。最后描述了释放一致性——一个相对弱一点的一致性模型，在许多情况下，达到这种一致性就足够了，同时它也较容易实现。

748
~
749

18.1 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，有一个在底层的运行时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18-1）。

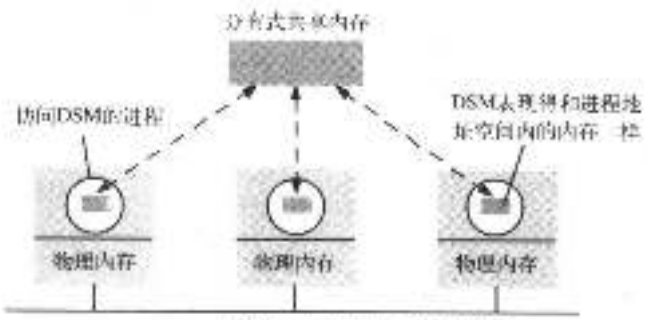


图18-1 分布式共享内存抽象

DSM的最关键一点是，它使得程序员不必关心数据通信，而使用传统方式编程时，就不可避免地要涉及到消息传递。并行应用程序、分布式应用程序以及应用程序组都可以使用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看作抽象数据并且通过发送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在运行时将不得不依靠在计算机之间传递的消息来发送数据更新。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拷贝，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15章讨论的问题有关，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同时映射到各自的地址空间来同时共享这些文件。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。

750

随着共享内存的多处理器结构（见6.3节）的发展，DSM的重要性也开始显现。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小的内存访问延时和较大的吞吐量的同时，增加处理器的数目 [Dubois et al. 1988]。在进程和内存模块通过普通总线连接的系统中，因为存在总线竞争，故处理器数目限制在10个左右，否则系统的性能将会明显下降。共享内存的处理器通常是四个一组，它们通过一个电路板上的总线来共享内存。在*非一致内存访问*（NUMA）体系结构的主板上，系统可以支持多达64个处理器。它是一种层次型的体系结构，其中系统使用高性能的开关或高级别的总线将多个拥有4个处理器的主板连接起来。在NUMA体系结构中，处理器看到一个包含所有主板的所有内存的地址空间。但是处理器访问本主板上内存的延迟比访问其他主板上内存的延迟小——这就是这一体系结构名称的来源。

在*分布式内存多处理器结构*和非定制计算组件集群（见6.3节）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，这些系统中处理器的数目可能很大，比共享内存的多处理器结构的64个处理器的上限大很多。DSM和多处理器结构的研究所面临的一个核心问题是：共享内存算法和其相关软件能否直接应用于更大规模的分布式共享内存体系结构。

18.1.1 消息传递机制和DSM

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，而且使用的是异步通信。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也是如此，因此不需要单独的通信操作。大多数实现允许系统可以以访问非共享变量的方式对存储在DSM中的变量进行命名和访问。另一方面，使用消息传递的好处是：通过进程的专用地址空间，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，在异质的计算机系统之间进行消息传递时，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？

消息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13章介绍的锁服务器这样的技术。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，如锁和信号量（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。

751

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保留它的数据，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：试验证明，一些为DSM开发的并行程序可以完成在相同硬件上——至少是相对少量的

计算机（大约10台左右）——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上的程序的性能取决于多个因素，特别是数据共享的方式（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，因此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，一个读或更新操作可能会涉及底层运行时支持的通信，也可能不涉及。是否涉及通信取决于若干因素，包括该数据之前是否被访问过，以及属于不同计算机的进程之间的内存共享模式。

DSM是否比消息传递机制更适合于某个应用程序，这一问题没有确定的答案。DSM是一个非常有发展前景的工具，它的最终应用情况取决于实现效率。

18.1.2 DSM的实现方法

通过使用专门的硬件组合、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专门的硬件为处理器提供一致共享内存。它们通过与远程内存和缓存模块通信来处理内存LOAD和STORE指令，这些远程内存和缓存存储着所需的数据。这种通信是在一种和网络类似的高速连接上进行的。Dash多处理器的原型拥有用NUMA体系结构连接方式连接的64个节点。

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.cdk4.net/oss）、Choices
752 [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格式的同构计算机。

中间件：一些像Orca[Bal et al. 1990]的语言和一些像Linda[Carriero and Gelernter 1989]以及其后的JavaSpaces[Bishop and Warren 2003]和TSpaces[Wyckoff et al. 1998]的中间件可以以与平台无关的方式支持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实例的通信来实现共享。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的实例访问本地数据，并且在必要时进行通信以维持数据一致性。

本章主要讨论在标准计算机上使用软件来实现DSM。即使使用有硬件支持的DSM系统，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给DSM强加某种结构，DSM表现为一系列的字节。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——www.cdk4.net/mach介绍Mach的虚拟内存设施）。现在，基于分页的DSM系统通常是在用户级实现的，这样可以充分利用系统提供的灵活性。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。具有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管理的限制[Bartoli et al. 1993]。

图18-2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]提供的分页式DSM通信。Writer程序更新在Mether DSM段开始处（从地址METHERBASE处开始）的一个结构中的两个域的值，并且Reader程序定期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工作站和网络硬件上。

```
#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) {   /*每秒读一次域的值*/
                printf("a = %d, b = %d\n", p ->a, p ->b);
                sleep(1);
        }
}
```

图18-2 Mether系统程序

使用中间件实现DSM的方法和使用专门硬件及分页的方法完全不同，它并不使用现有的共享内存代码。它能使我们开发出共享对象的更高级别抽象，而不是共享内存地址位置。

18.2 设计和实现问题

本节将讨论与DSM系统主要特征有关的设计和实现问题。它们是：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，它负责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；颠簸问题。

18.2.1 结构

第15章讨论了复制像日记和文件这样的对象的系统。这些系统允许客户程序操纵这些对象，就像每个对象只有一个拷贝一样，但实际上，它们访问的是多个不同的物理拷贝。系统对对象副本所允许的差异的程度做出保证。

DSM系统就是如上所述的一个复制系统。每一个应用程序进程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，可以用不同的方式访问对象。访问DSM的不同的方式的区别在于：DSM是如何被看成“对象”的以及是如何找到对象的。我们将介绍三种不同的访问方法，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。

753 ~ 754

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——它是一组连续的字节。Mether系统就采用了这种方法。其他一些DSM系统也采用了这一方法，包括Ivy系统，我们将在

18.3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是两个内存位置，下面是使用这两种操作的例子：

R(x)a——从位置x读出值a的read操作。

W(x)b——将值b写入位置x的write操作。

一个执行序列是：W(x)1，R(x)2 。该进程在位置x写入值1，然后从这一位置读出值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，而不能通过直接访问成员变量来改变内存的值。这种管理内存的方式的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，同时它可以自动对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，也可添加和删除数据项。这类系统包括Agora系统[Bisiani and Forin 1988]和更为著名的Linda系统以及它派生的TSpaces和JavaSpaces系统。

Linda这类系统向程序员提供了元组的集合，称为*元组空间*（见16.3.1节）。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不同类型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write操作来向元组空间内加入元组，并通过read或take操作来读出或抽取元组。write操作可以在不影响元组空间内已有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的值。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。

当从元组空间内读出或取出一个元组时，进程提供一个元组规约，系统从元组空间内返回符合这一规约的元组，这是一种关联寻址。为了使进程间的活动能同步，read操作和take操作会被阻塞，直到在元组空间内找到符合规约的元组为止。一个元组规约包括元组的域的数目和特定域中所需的值或域的类型。例如，take（<String，integer>）操作可能会获取<“fred”，1958>或<“sid”，1964>；take（<String，1958>）操作只会获取以上两个元组中的<“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，因此进程为了修改元组必须采
755 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”，counte+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出现竞争。

18.2.2 同步模型

许多应用程序会给共享内存存储的值加上某些约束。和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，其约束可以是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。这就违背了以上约束。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同步服务，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，对象的实现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13章中对锁服务器的描述）。在共享内存的多处理器系统中为进行同步而使用的像testAndSet这样的特殊指令也可以应用于基于分页的DSM，但这些操作在分布式环境下的效率很低。采用应用程序级同步机制的DSM可以减少更新传递量。这种DSM系统将同步机制看作是它的一个集成的组件。

18.2.3 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15章中所描述的那样，像DSM这样的系统会产生一致性问题。按照第15章给出的术语，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。

系统通过结合中间件（在每个进程的DSM运行时层）和内核的功能来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策略。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。

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15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。

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，这样客户可以选择负载最小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，系统不需要在任何时刻在所有计算机上都保持此信息的一致性。

然而，大多数应用程序需要更强的一致性，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。在详细介绍内存的一致性需求之前，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。

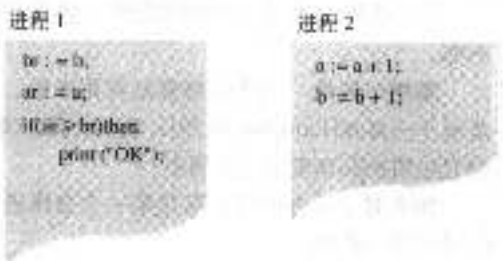


图18-3 访问共享变量的两个进程

假设一个应用程序中有两个进程（参见图18-3），它们访问两个变量a和b，a、b的初始值都为0。进程2按顺序将a和b的值加1，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。注意，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，这取决于进程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≥br总能满足，进程1总能打印“OK”。然而，DSM实现可能不按照进程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

756
/
757

也可能出现。

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计算机上，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模型。

描述某个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，由于可能对同一地址进行过多次写操作，那么应将哪一个值返回给读操作？在最弱的一致性模型中，答案是将读操作前任意一个写操作的值返回给读操作。如果副本管理器以一种不确定的方式延迟更新其数据，系统就只能达到这种一致性。这种一致性太弱了，以至于没有实际用处。

而在最强的一致性模型中，所有更新的值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种一致性模型。读者可能已经对它有所了解。在第15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，我们将重申在第15章中的线性能力的定义。

如果对于任何执行，存在某一个满足下列条件的全体客户操作的序列，那么一个复制的共享对象服务被认为是可线性化的。

L1：操作的交错执行序列符合对象的（单个）正确副本所遵循的规约。

L2：操作之间的交错执行序列和实际运行中的次序一致。

这一定义具有普遍性，它能应用到任何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，所以我们对它作详细说明。举一个简单的例子，假设一个共享内存由一组可读可写的变量组成。所有的操作都是读或写操作，在这里我们使用18.2.1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：

758

L1'：如果交错执行序列中有一个R(x)a操作，那么，或者在发生这一操作之前的最后一个write操作是W(x) a，或者在此之前没有发生过写操作并且x的初始值为a。

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。DSM应用中最强的一致性模型是*顺序一致性*[Lamport 1979]，我们在第15章曾经介绍过它。这里，我们将采用第15章的定义，并将它应用在共享变量的特例中：

如果对于任何执行，存在某一个全体进程操作的序列，满足以下两个准则，则说一个DSM系统是顺序一致的。

SC1：如果交错执行序列中有一个R(x)a操作，那么，或者在发生在这一操作之前的最后一个write操作是W(x) a，或者在此之前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错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，这一点使实

现顺序一致性成为可能。

这些准则可以用另一种说法来描述：在一个虚拟内存映像上有一个所有进程的read和write操作的虚拟的交错执行序列。该交错执行序列保持了每个操作在程序中的顺序，而且该交错序列中每个read操作读取的都是最后被写入的值。

在实际执行中，只要不违反以上定义中的约束，内存操作可以重叠，并且在不同的进程中更新的顺序可以不同。请注意，为了满足顺序一致性的条件，要考虑在整个DSM上的内存操作，而不仅仅是单个位置上的操作。

前例中的ar=0，br=1的组合不可能发生在符合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18-4给出了按顺序一致性执行的进程内存访问操作。另外，尽管这个例子显示的是read和write操作实际的交错执行顺序，但一致性定义规定了操作的执行应该像严格的交错执行一样。

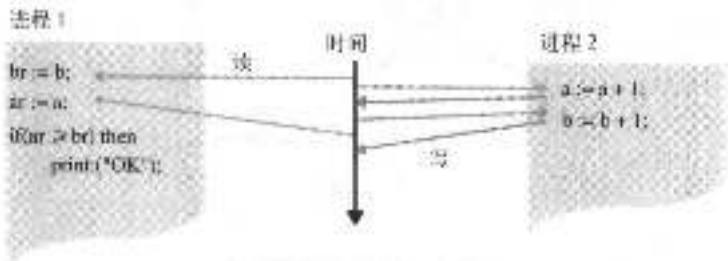


图18-4 顺序一致性下的交错执行

使用一个保存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read和write操作，这样可以实现具有顺序一致性的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，人们设计出了一个更弱一些的、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write操作的顺序达成一致，但对不同地点上的write操作的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。连贯的DSM的优点在于：因为协议单独应用到每个页上，所以对不同的页的访问是相互独立的，一个访问操作不会对其他页的访问操作造成延迟。

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结果的同时，试图避免在多处理器上顺序一致性的开销。为了放宽内存一致性，该模型利用了同步操作所隐含的信息。这样对程序员来说，系统实现了顺序一致性（至少，在某些环境下是如此，但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”，但是因为在这段时间内没有其他进程可以访问它。所以，这些执行看起来是顺序一致的。Adve和Hill[1990]推广了弱一致性概念，称为弱顺序性：“当且仅当（一个DSM系统）对所有遵守同步模型的软件都表现为顺序一致，则它相对于一个同步模型是弱顺序的”。释放一致性是弱一致性的一种改进模型，18.4节介绍释放一致性。

18.2.4 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。这两种方法可以应用到各

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：

759 ~ 760 写更新：进程所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取的数据（参见图18-5）。进程可以读数据项的本地拷贝，而不需要再进行通信。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。

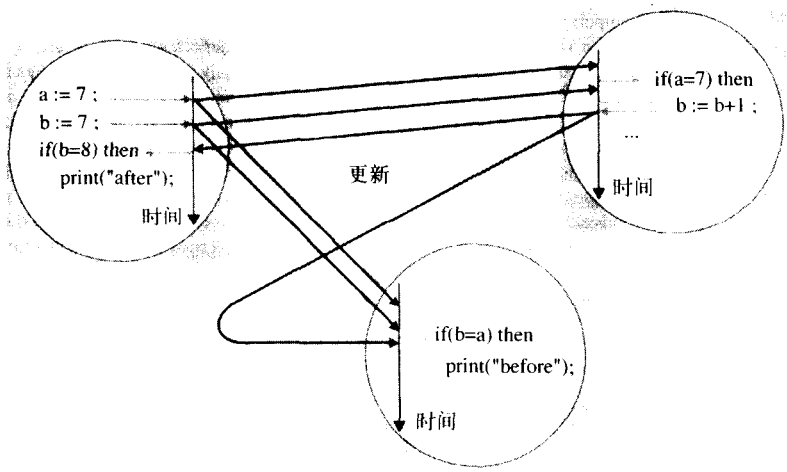


图18-5 使用写更新的DSM

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12章对全排序组播的介绍），并且要求更新消息在本地传递后才返回，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具有良好定义的，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12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.cdk4.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，使用专门的硬件支持写更新协议。

写失效：它通常应用在多个读进程/单个写进程共享中。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，或者该数据项由一个进程进行读写。当前以只读方式访问的数据项可以有多个数据拷贝。当一个进程试图写这个数据时，系统首先向该数据的所有拷贝组播一个失效消息，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取该数据项的过时数据（也就是说，这个数据项不是最新的）。如果有一个写进程正在写数据，其他
761 试图访问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。该方案是在先来先服务的基础上处理数据访问的。Lamport[1979]证明这种方案可以获得顺序一致性。在18.4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拷贝上，并且在这种更新通信前可能会进行多个更新操作。但其缺点是：在写操作发生前，使只读拷贝失效要花费一些开销。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/写率很高，那么允许多个读进程同时访问所得到的并行性可以弥补这一开销。当读/写率很低时，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。

18.2.5 粒度

与DSM结构相关的一个问题是共享的粒度。从概念上说，所有的远程共享DSM的所有内容。然而，共享DSM的程序执行时，实际上只在执行中的特定时间内共享特定的一部分数据。如果在DSM实现中进程访问和更新数据时总是传输整个DSM的内容，那么就会产生浪费。那么在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数据一致性，DSM应实时发送什么样的数据？

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18.11）。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——主要是通过在页表中放置新的页面指针（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以保持远程拷贝的一致性。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，系统都要进行整页传输。

使用更小的页面（512字节或1K字节）并不会使整个系统的性能得到明显改进。首先，当远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，这是因为每个网络分组有固定的软件开销。其次，在DSM实现中使用小一些的页，系统就必须管理更多的数据单元。

更复杂的是，当页很大时，进程间很容易产生竞争页的情况。这是因为随着页的增大，访问同一页内的数据的可能性也将增大。例如，有两个远程，一个进程只访问数据项A，而另一个进程只访问数据项B，它们位于同一页内（参见图18-6）。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。这种现象称为错误共享，即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，错误共享可能导致不必要的失效。在写更新协议中，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重复更新。 762

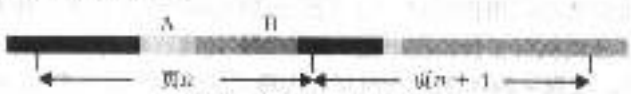


图18-6 在页上分布的数据项

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要基于可用物理页的大小。相对于页边界进行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。

18.2.6 系统颠簸

写失效协议的一个潜在问题是系统颠簸。相对于应用进程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，那么系统就会发生颠簸。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，如果有一个远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，在该方法中，计算机只在一小段时间内"拥有"页面。18.4节将介绍Munin系统是如何允许程序员为DSM系统确定访问方式的，这样它可以为每一数据项选择合适的更新方式，避免系统颠簸。

18.3 顺序一致性和Ivy实例研究

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。

18.3.1 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（参见图18-7）。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意义的指针值。在计算机上执行的进程都配备有分页的内存管理单元。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这样的进程。然而，这些进程可以直接共享DSM页（不同的进程可以使用在计
763 算机系统页表中的同一页面）。唯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。

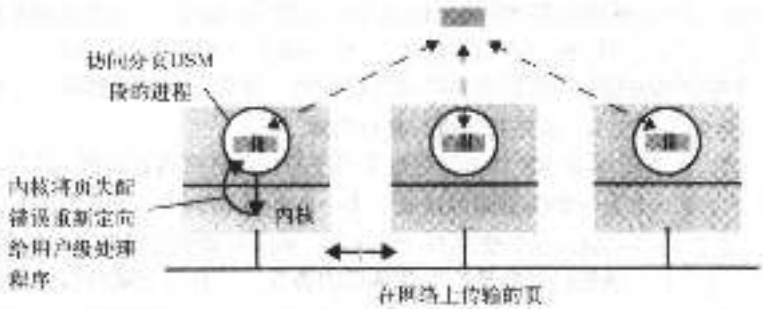


图18-7 基于分页DSM的系统模型

分页机制对于进程内的应用组件而言是透明的，它们可以在逻辑上读写DSM中的任何数据。然而，为了维护处理读写时系统的顺序一致性，DSM运行时将限制页的访问许可。分页内存管理单元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页失配或写页失配错误。内核将页失配重定向到由每个进程中的DSM运行时层指定的一个处理程序。页失配处理程序（对应用程序来说是透明的）在把控制权交还给应用程序前，以下面介绍的特殊方式处理这种失配。在像Ivy这样的DSM系统中，内核本身就执行了我们在上面介绍的许多处理。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运行时层和内核共同提供了这些处理功能。通常，进程中的DSM运行时层包含大多数这样的功能，并且不会因为改变内核而需要重新实现和优化。

这里的描述省略了常规虚拟内存实现中的页失配处理，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，那么只有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合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的数据副本上。假设已经对一个页设置了写保护，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
764 的页失配处理进程。在原则上，这个处理程序会检查产生失配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失配的指令前，它会组播所做的更新。

但是如果已经恢复写访问，随后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异常处理程序将关闭对该页的写访问许可，然后再次关闭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。

实际上，写更新应用在基于分页的实现中，并且仅仅在下列情况中页仍然保持写许可：在第一个页失配发生后，并且在传播更新的页之前，系统允许多个写操作发生。Munin使用了这种写缓

冲技术。作为一种很有效的方法，Munin系统试图避免传播整个页——可能这个页中只有一小部分被更新。当一个进程第一次试图写这个页时，Munin系统处理失配的方法是：在允许写操作之前获得这个页的拷贝，并将其放在一边。然后，当Munin准备传播这一页时，它将更新的页和保存的页拷贝进行比较，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拷贝生成更新后的页拷贝。

18.3.2 写失效

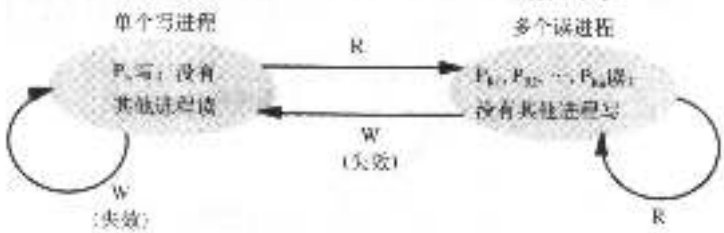
基于失效的算法使用页保护来实现一致性数据共享。当一个进程更新一个页时，该进程持有对本地数据的读和写许可，其他进程没有对该页的访问许可。当有一个或多个进程读一个页时，这些进程具有只读许可，其他进程没有访问许可（尽管它们可以获得读许可）。除此之外，没有第三种情况。拥有页p最新版本的进程被指定为这一页的拥有者，表示为owner(p)。它可能是单个写进程，也可能是多个读进程中的一个。拥有页p拷贝的进程集合称为拷贝集，表示为copyset(p)。

图18-8给出了可能的状态转换。当一个进程P_a试图写页p，并且它对这一页没有访问权限或只有只读权限时，会发生一个页失配。以下是这一页失配的处理过程：

- 如果P_a进程不拥有最新的只读拷贝，将这一页传给它。
- 这一页的其他拷贝都失效，即所有copyset(p)成员的页访问许可被设置为不允许访问。
- copyset(p) ={P_a}。
- owner(p):= P_a。
- P_a中的DSM运行时层在地址空间的合适位置给这一页加上读-写许可，并且从导致失配的指令处重新执行。

765

请注意，两个以上拥有只读拷贝的进程可能在同一时间发生写失配。当系统最终授予页所有权时，这个页的一个只读拷贝可能已经不是最新的了。为了检查当前页的只读拷贝是否是最新的，系统可以记录每个页的序列号，当页的所有权转移时，它的序列号会加1。当一个进程需要进行写访问时，它会获得这个只读拷贝的序列号。当前的拥有者就可以知道这个页是否已经被更新，是否需要发送出去。Kessler和Livny[1989]将这一方案描述为"准确算法"。



注：R = 发生读页失配；W = 发生写页失配。

图18-8 在写失效方法中的状态转换

当进程P_a试图读页p，而它没有对这一页的访问许可时，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_a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拷贝。然而，即使这一进程不再访问这一页，但因为它是这一页的拥有者，它也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_a。
- copyset(p):= copyset(p) ∪ {P_a}。

- P_R中的DSM运行时层在地址空间的合适位置给这一页加上只读许可，并且从发生失配的指令处重新执行。

系统可能在执行上面所介绍的转换算法时发生第二个页失配。为了使这一转换能一致地执行，系统不处理新的页请求，直到这一转换完成为止。

766 以上仅介绍了必须做什么。下面将介绍如何高效实现页失配处理。

18.3.3 失效协议

实现失效方案要解决协议中的两个重要问题：

1）如何为给定的页 p定位owner(p)?

2）在哪里存储copyset(p)?

在Ivy系统中，Li和Hudak[1989]描述了几种体系结构和协议，它们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，系统使用一个称为管理器的服务器来为每个页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)成员的进程标识符和传输地址。

如图18-9所示，当一个页失配发生时，本地进程（我们称为客户）向管理器发送一个包含页号和关于访问类型（读和读-写）的消息。然后客户等待应答。管理器查找owner(p)的地址并将这一请求转发给页拥有者。在发生写失配的情况下，管理器将新的拥有者设为写数据的客户，后续的请求会在客户处进行排队，直到客户完成了所有权转换再进行处理。

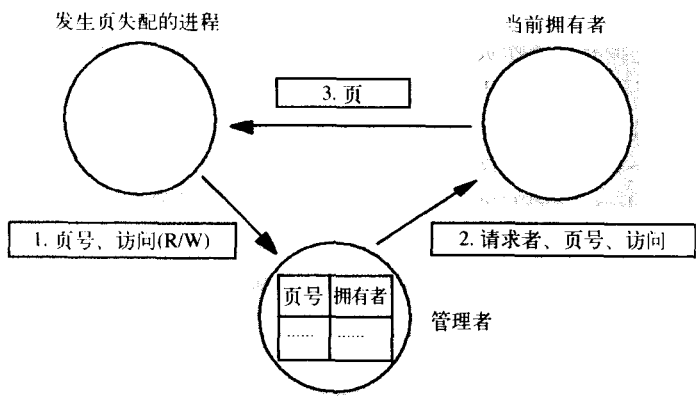


图18-9 中央管理器和与其相关的信息

页的前一个拥有者将这一页传给客户。在发生写失配的情况下，它同时也发送页的拷贝集copyset(p)。当客户得到这个拷贝集时，它就执行失效过程。它向拷贝集的成员进程组播一个请求，并等待这些进程发回“失效已经发生”的确认信息。这一组播不需要进行排序。客户不需要向前一个页拥有者发送失效消息，因为它在转移所有权时已经将自己失效了。至于拷贝集的管理问题

767 留给读者自己解决，读者可以参考上面所描述的通用失效算法来解决该问题。

管理器是系统性能的瓶颈，并且是故障的关键点。Li和Hudak介绍了三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，每一个管理器在功能上都与上面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页号被散列函数处理后落到某个值域内的页。客户计算所需页的散列数，在一个预先确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的问题，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其他进程。只有拥有此页的进程给出应答。但必须注意，可能会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进程O拥有的页。假设O首先接收到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，因为它还不是拥有者，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，而这一请求已经过期了。

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11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，这一方法拥有组播方法共同的缺点：不是页拥有者的进程也会被不相关的信息中断，从而浪费了它的处理时间。

18.3.4 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在进程之间传递页所有权，但是它没有使用组播机制，而是使用了另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 768
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位置的精确信息。然而，在一般情况下，这些值称为*提示信息*，这是因为页可以在任意时间内转换它的所有权。而在以前的算法中，系统只在发生写失配时才转换页所有权。

系统可以沿着提示信息组成的提示链（因为页的所有权会从一台计算机转换到另一台计算机）来找到页拥有者。这条提示链的长度（也就是为了找到拥有者需要转发消息的次数）可能会无限增长。通过在有更新的值可用时才更新提示信息的方法，该算法解决了这个问题。以下就是更新提示信息以及转发请求信息的过程：

- 当一个进程将页p的所有权传输给其他进程时，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) 设置为请求进程。

前三种更新只在转移页所有权和提供只读拷贝时发生，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。后面我们将再对这一内容进行讨论。

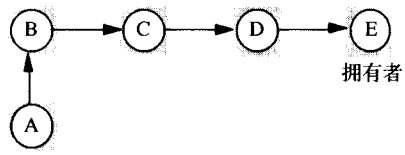
图18-10（a和b）表示了在进程A发生一个写页失配之前和之后的probOwner指针的情况。进程A的probOwner指针最初是指向进程B的，进程B、C和D的probOwner指针组成的指针链最终将请求

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。在处理完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。

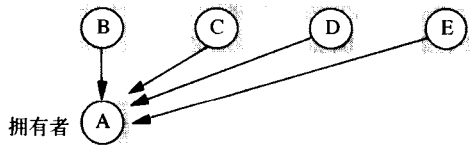
然而，如果A发生了读失配，那么进程B对这一页的访问步骤变少了（原来访问E需要三步，现在只需要两步），进程C对这一页的访问和以前一样（都是两步）。但是，进程D的情况变糟了，原来只需要一步，现在需要两步（参见图18-10c）。为了观察这一策略的性能，需要进行模拟。

通过定期向所有进程广播页的当前拥有者地址可以控制指针链的平均长度。在一次广播后，所有的指针链的长度为1。

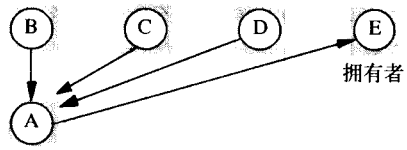
为了观察指针更新算法的效率，Li和Hudak实现了一个仿真系统，并给出了仿真结果。如果随机选择失配进程，对1024个处理器，如果系统平均每256个页失配就广播一次拥有者地址，消息到达页拥有者的平均传递步骤是2.34，而如果平均每1024个失配广播一次，那么平均传递步骤是3.64。这些数字只作为一个参考，Li和Hudak[1989]给出了完整的结果。请注意，使用中央管理器的DSM系统也需要两条消息才能到达页拥有者。



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的写请求被转发后的probOwner指针的情况



c）读失配：在A的读请求被转发后的probOwner指针的情况

图18-10 更新probOwner指针

最后，Li和Hudak描述了一种优化方法，该方法可能会提高失效操作的效率，并减少处理一个读页失配所需的消息数。在此算法中，进程不需要从页拥有者那里获取页拷贝，客户可以从任何拥有正确拷贝的进程中获得拷贝。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一个拥有正确拷贝的进程。

为了实现这一点，进程必须保留一个从该进程获得页拷贝的客户记录。拥有页的只读拷贝的进程被组织成一个树，它的根就是页拥有者，每个节点指向子节点，这些子节点从父节点获得页拷贝。一个页的失效操作从页拥有者开始，并沿着树向下传递。当一个节点收到失效消息后，它将此消息传递给其子节点，并使子节点上的页拷贝失效。最好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更为有用。

769
~
770

18.3.5 系统颠簸

有一种说法认为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运行时层尽可能减少页拷贝和所有权转换，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将每一个页与一段较小的时间间隔联系起来。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）内进行该访问。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时间片的长度。如果系统使用一个静态选择的时间长度，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对页的使用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位）来选择。另一个要考虑的因素是等待一个页的进程队列的长度。

18.4 释放一致性和Munin实例研究

前一节所介绍的算法用来实现顺序一致的DSM。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——虽然失效操作使用无序组播就足够了。定位页拥有者的开销也比较大：管理所有页拥有者的位置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，则要传递更多的消息。另外，基于失效的算法可能引起系统颠簸。

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，但它的实现开销也小一些。不过，程序员可以比较容易地使用它的语义。

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的实现可以利用这样的信息，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，不过因为使用同步对象，系统在应用程序层仍然可以保持一致性。 771

18.4.1 内存访问

为了理解释放一致性（或者其他考虑了同步的内存模型），我们首先根据内存访问在同步中的角色对其进行分类。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，我们还将给出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可以使用消息传递而不是共享变量来实现同步，这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。

```
acquire Lock(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，两个访问是竞争性访问：

- 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write访问。

所以两个read操作是不会竞争的。两个进程对同一内存位置进行的读访问和写访问，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。

我们可以将竞争性访问细分为同步访问和非同步访问两种类型：

- 同步访问是指那些与同步有关的write或read操作。
- 非同步性的访问是指那些并发但与同步无关的read或write操作。

在（上面的）releaseLock中，由lock:=0隐含的write操作是同步访问。在testAndSet中隐含的read操作也是同步访问。

同步访问是竞争性的，这是因为同步进程必须能够并发访问这些同步变量，并且它们会更新这些变量：只使用read操作不可能实现同步。但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，它们对共享变量进行竞争性的访问只是为了更新和读取另一个进程记录的结果，但它们不是同步的。 772

同步访问又可以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而进一步分为获得访问和释放访问。

执行异步操作 我们发现，在顺序一致DSM的实现中，内存操作可能发生明显的延迟。尽管有延迟，系统仍可采用多种形式的异步操作来提高进程执行的效率。首先，写操作可以异步实现。在写操作的结果被传播，并且其他进程得到写操作的效果之前，它的值可以被放置在缓冲区内。其次，DSM系统可以通过数据预取，即预先存储可能将要被访问的数据，以减少数据访问的延迟。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，只要下一条指令不能依赖于当前指令即可。

就我们所列出的异步操作而言，我们将区分read或write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DSM系统至少是连贯的。正如18.2.3节介绍的，这意味着在同一位置上每个进程就写操作的顺序达成了一致。在这一假设下，我们可以在给定的位置上明确地给出write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画出时间线（见图18-11）。

我们称一个进程P执行了一个write操作W(x)v，如果在该操作后，进程P的read操作会返回值v，这个值是由P的write操作写入的，或者返回由后续的操作对x写入的值（其他操作可能也会写入同样的值v）。

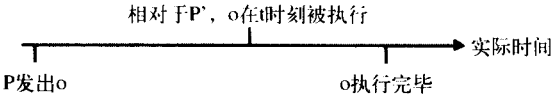


图18-11 执行DSM读或写操作的时间线

类似地，我们称进程P执行一个read操作R(x)v，如果没有后续的write操作将x的值写为v。例如，P可能领取了它所需要读出的值。

最后，如果所有进程都完成了操作o，则称操作o完成了。

18.4.2 释放一致性

我们希望满足的系统需求包括：

773

- 保持对象（如锁和屏蔽等）的同步语义。
- 为了获得较好的性能，我们允许内存操作存在一定程度的异步性。
- 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。

为此，人们设计出释放一致性模型来满足上述需求。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，以前获得的所有访问必须完成执行。

RC2：在允许其他进程执行release操作前，以前所有的普通read和write操作必须执行完成。

RC3：就acquire和release操作而言，两者必须是顺序一致的。

RC1和RC2保证了当释放操作发生时，其他获得锁的进程不能读那些由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系统知道有哪些同步访问，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，释放一致性模型允许实现采用某些异步操作。例如，当一个进程在一个临界区内执行更新时，进程没有必要阻塞它。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

发送最后一个对数据项的更新。

考虑图18-12中的进程，它们为了访问变量a和b（a和b的初始值都为0）而获得和释放锁。进程1在互斥的情况下更新a和b，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，即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。

```
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18-12 在释放一致的DSM上执行的进程

假设进程1首先获得锁，那么进程2会被阻塞，并且它不会引起与DSM相关的任意活动，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进行操作，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拷贝失效为止。

在释放一致性条件下，当进程1访问a和b时，进程1将不会被阻塞。DSM运行时系统知道这些数据被更新了，但是它没有必要立即采取进一步的行动。仅当进程1释放了锁之后，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；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。

18.4.3 Munin

Munin DSM系统[Carter et al. 1991]试图通过实现释放一致性模型来提高DSM的效率。此外，Munin系统允许程序员根据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在V内核[Cheriton and Zwaenepoel 1985]上实现，这个系统是首先允许用户级进程来处理页失配并操作页表的内核系统之一。

下面是Munin系统实现释放一致性的几个关键点：

- 当锁被释放时，Munin系统立即发送更新或失效信息。
- 程序员可以注明锁与数据项之间的关联。在这种情况下，DSM运行时系统可以利用将锁传输给等待进程的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拷贝。

774
775

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。这种惰性方法仅当下一次获得锁时才进行消息传播。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。在其他进程获得锁以前，没有必要让这些进程知道更新结果。

共享注解 Munin实现了多种一致性协议，它们可以应用在不同粒度的数据项上。这些协议的参数根据如下选项确定：

- 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发送到这一拥有者上）。
- 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。程序员也可以针对每个数据项选择使用什么参数。然而，Munin系统为程序员提供了一个标准的注解集，这些注解可以应用在数据项上，每一个注解对应一组对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，但它可以被自由拷贝。
- *迁移*：进程通常轮流访问数据项，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，而只传播那些被每个进程实际更新的部分。为了做到这一点，Munin系统（而不是写失配处理程序）在执行本地更新前拷贝该页。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。

776

- *缩影*：数据项通过加锁、读、更新和解锁而被修改。其中一个例子是，并行计算的一个全局缩影，如果它比局部缩影大，那么它必须以原子方式获得和修改数据。这些数据项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，由一个“管理者”进程处理这一数组。要指出的一点是，更新只需发送到管理者进程上，而没有必要发送到工作者进程上（它可能在上面描述的“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]详细介绍了对应于以上注解的参数选项。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。

18.5 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型，而混合型模型区分普通访问和同步访问（以及其他的访问类型）。

一些统一型模型的一致性比顺序一致性弱一些。我们在18.2.3节介绍了连贯性，连贯性是指在每一个位置上都是顺序一致的。进程在同一位置上执行写操作的顺序是一致的，但是在不同位置对不同进程执行写操作的顺序可能不一致[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11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：(1)它们是同一进程执行的操作；(2)一个进程读另一个进程写入的数值；(3)存在着用于连接两个操作的操作序列。这个模型的约束是，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（下面将介绍）。简单地说，处理器一致性就指是内存是连贯的，并且所有进程的执行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 和 Sandberg 1988]。

777

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绑定，这些同步对象管理对变量的访问。系统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，这样使其成为唯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系统避免了释放一致性中的错误共享，但是它增加了编程的复杂性。
- 范围一致性：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。在范围一致性中，变量可以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显式地关联起来。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。它的保证之一是以前所有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运行时系统必须知道同步操作，但是这并不是重大的缺点——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，他们就不必考虑某种内存一致性语义，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，要求程序员在程序中加入很多注解是很危险的。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。

18.6 小结

本章介绍了分布式共享内存的概念，我们将其看作在分布式系统中共享内存的一种抽象，它是替代基于消息通信的一种方法。DSM主要应用于并行处理和数据共享。在某些并行应用程序中，它的执行效果和消息传递机制一样好，但是高效实现它是很难的，并且它的性能随应用程序的不同变化很大。

778

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，目前DSM已经在硬件支持的基础上实现了。

DSM最主要的设计和实现问题包括：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的应用程序需要实现同步。它们使用锁这样的对象来满足这样的要求，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，人们设计出了几种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约束下来使用同步对象以获得更好的效率。本章还列出了其他一致性模型，

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拷贝上。尽管这些协议也有用全排序组播实现的，但通常还是使用硬件来实现的。写失效协议在数据项被更新时，通过使数据拷贝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的系统开销比较大。

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包含在同一共享单元中（例如页），所以有可能错误地共享数据，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，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，妨碍程序的执行。用程序层的同步，通过允许计算机在一小段时间内保留页，或者通过标记数据项使系统同时批准读写访问，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的三个主要写失效协议，它们用于处理管理拷贝集和定位页拥有者问题。这三个协议是：中央管理器协议（使用单个进程来存储每一页的当前拥有者信息）；使用组播来定位页的当前拥有者的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的当前拥有者）。

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，从而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，这样可以在指定共享方式的情况
779 下选择那些最适合这些数据项的协议选项。

练习

18.1 请说明在哪些方面DSM适合于客户–服务器系统，在哪些方面不适合该系统。 （第750页）

18.2 请讨论是消息传递机制还是DSM更适合容错应用程序呢。 （第751页）

18.3 在异构的计算机系统上可以使用中间件来实现DSM，那么应如何解决不同的数据表示问题？如果是在基于分页的DSM实现上，你又如何解决这一问题？你的解决方法涉及指针了吗？ （第753页）

18.4 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？ （第754页）

18.5 你将怎样使用元组空间来实现一个信号量？ （第755页）

18.6 下列两个进程的顺序执行后，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 （第759页）

18.7 使用R()和W()的标记方法，设计一个是连贯的但不是顺序一致的内存执行的例子。可能出现内存是顺序一致的但不是连贯的这种情况吗？ （第759页）

18.8 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拷贝上执行了，甚至组播是全排序的，系统也可能不符合顺序一致性。请讨论异步组播是否可以用来实现顺序一致性。（提示：考虑是否要阻塞后续的操作。） （第760页）

18.9 使用采用了一个异步的、全排序的组播的写更新协议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有哪些需求。 （第760页）

18.10 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。讨论该算法的性能。 （第760页）

18.11 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？ （第762页）

18.12 DSM的页替换策略寓意着什么（也就是说，为了引入一个新页，选择哪一页被淘汰）？（第763页） 780

18.13 请证明Ivy系统的写失效协议保证了顺序一致性。（第765页）

18.14 在Ivy系统的动态分布式管理器算法中，为了将找到一个页所需进行查找的次数降到最低，采取了哪些步骤？（第768页）

18.15 为什么说在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？（第771页）

18.16 请讨论释放一致性的条件RC2是如何放宽的。如何区分及时和惰性的释放一致性。（第774页）

18.17 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。一个监控器进程定期读取t。解释传播更新给t的同步需求，包括那些在应用程序层是不需要的需求。在这些进程中，由谁来执行同步操作？（第773页）

18.18 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: $R(b)2; R(a)0$ （第777页）

18.19 DSM实现知道数据项和同步对象之间的关联有何好外？将这种关联显式表示的缺点是什么？（第778页） 781

第19章 Web服务

Web服务提供了服务接口，使客户能以一种比Web浏览器更通用的方式与服务器进行交互。客户通过在HTTP上传输的XML格式的请求和应答访问Web服务接口中的操作。可以通过比基于CORBA的服务更自主的方式访问Web服务，从而可以使其更容易被因特网范围中的应用程序使用。

与CORBA和Java类似，Web服务的接口可以用接口定义语言（IDL）描述。但对于Web服务，还需要描述其他信息，包括所用的编码和通信协议以及服务位置等。

用户需要以安全的方式创建、存储和修改文档以及在因特网上交换文档，TLS提供的安全通道并不能满足这些要求，而XML安全性试图解决这些问题。

网格指的是一个基于Web服务的中间件平台，设计它是供拥有大量要处理的数据资源的大型分散用户群体来使用的。天文学家的World-Wide Telescope是典型的用于科学协作的网格应用程序。从对World-Wide Telescope的研究可以总结出数据密集型科学应用的特征，从这些特征可以求出网格架构的需求集。

782
~
783

19.1 简介

近五年网络的增长（见图1-6）证明了在因特网上使用简单协议作为大量广域服务和应用的基础是有效的。特别是HTTP的请求/应答协议（见4.4节）允许通用客户调用浏览器，通过URL引用查看网页及其他资源。（见下面对URI、URL和URN的注释。）

然而，即使可以使用下载的特定于应用程序的applet增强，客户端的通用浏览器仍限制了应用程序潜在的使用范围。在最初的客户-服务器模型中，客户和服务器从功能上是专门化的。Web服务又回到了这个模型，在Web服务中，特定于应用程序的客户与具有特定功能接口的服务在因特网上进行交互。

因此，Web服务提供了基础设施来维持客户和服务器之间的更丰富并且更加结构化的互操作性。Web服务提供了一个基础，使得一个组织中的客户程序可以在无人监督的情况下与另一个组织中的服务器交互。特别地，Web服务允许通过提供集成了几个其他服务的服务来开发复杂的应用程序。由于其交互的通用性，Web服务不能直接通过浏览器访问。

> **URI、URL和URN** 统一资源标识符（URI）是一种通用资源标识符，其值可以为URL或URN。网络用户熟知的URL包括资源定位信息，例如命名资源的服务器的域名。统一资源名称（URN）是位置独立的，它依赖查找服务来映射到资源的URL。有关URN的详细信息可参见9.1节。

将Web服务附加到Web服务器是以使用HTTP请求引发程序的执行的能力为基础的。回忆一下，当HTTP请求中的URL指向一个可执行的程序，比如搜索程序，那么该程序将生成结果并将其返回。与此类似，Web服务是Web的扩展，并且可以由Web服务器（但不必须是Web服务器）提供。不应该混淆术语“Web服务器”和“Web服务”：Web服务器提供基本的HTTP服务，而Web服务提供基于在接口中定义的操作的服务。

外部数据表示和客户与Web服务之间交换的消息的编码是以XML的形式完成的，4.3.3节中已描述过XML。简言之，XML是一种文本表示方式，虽然比其他表示所占空间更大，但由于其可读性和易于调试而被广泛采用。

SOAP协议（见19.2.1节）指定了使用XML封装消息（比如支持请求/应答协议的消息）的规则。

图19-1总结了Web服务运作的通信体系结构的要点。Web服务由URI定义，客户可以使用XML格式的消息对其进行访问。SOAP用来封装消息并在HTTP或其他协议（如TCP或SMTP）上传送这些消息。Web服务为潜在的客户部署服务描述来指定接口或服务的其他方面。

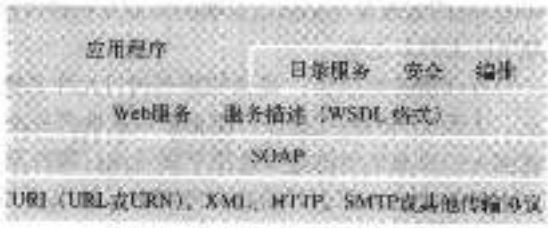


图19-1 Web服务基础设施和组件

图19-1的第一层说明：

- Web服务和应用程序可以构建于其他Web服务之上。
- 一些特定的Web服务为大量其他Web服务的操作提供所需要的通用功能，包括目录服务、安全和编排，我们将在本章稍后部分讨论这些功能。

Web服务一般提供一个服务描述，包括接口定义和其他信息，如服务器的URL。服务描述可作为客户和服务器之间对所提供的服务达成共识的基础。19.3节介绍了Web服务描述语言（WSDL）。

中间件的另一个共同需求是允许客户找到服务的名字或目录服务。Web服务的客户也有类似的需求，但经常在没有目录服务的情况下进行管理。例如，它们经常从网页上的信息（例如，从Google的搜索结果）中查找服务。然而，如果要提供适合组织内部使用的目录服务，还需要一些进一步的工作，这些内容将在19.4节中讨论。

我们将在19.5节中介绍XML安全性，在这种达到安全性的方法中，可以对文档或文档的部分进行签名或加密。然后可以传输或存储经过签名或加密的元素。之后可以生成附加信息，而这个附加信息也可以被签名或加密。

Web服务可以使远程客户访问资源，但不负责协调它们之间的相互操作。19.7节将讨论Web服务的编排，它允许Web服务在使用其他服务集时，使用预定义的访问模式。

本章的最后一节包含对网格（一个基于Web服务的应用程序）的实例研究。网格是一个基础设施，用于提供对资源（如程序、文件、数据、计算机、传感器和网络）的大规模共享的访问。最初创建Web是为了满足不同站点的物理学家小组共享实验文档的需要。最近，科学家需要一个在因特网中的更通用的分布式计算环境，网格概念得到了进一步发展。

784
785

19.2 Web服务

Web服务接口通常由客户可以在因特网上使用的操作集合组成。Web服务中的操作可以由各种不同的资源提供，如程序、对象或数据库。Web服务既可以与网页一起被Web服务器管理，也可以是完全独立的服务。

大多数Web服务的关键特征是可以处理XML格式的SOAP消息（见19.2.1节）。另一种替代方法是在788页方框中列出的REST方法。每个Web服务使用自己的服务描述处理它接收到的消息中特定于服务的特征。有关Web服务的更详细的介绍见Newcomer[2002]。

许多知名的商业网络服务器（包括Amazon、Yahoo、Google和eBay）提供允许客户操纵网络资源的Web服务接口。例如，Amazon.com的Web服务提供了一些操作，允许客户获取商品的信息，将某项商品添加到购物车中或检查交易状态。Amazon的Web服务[associates.amazon.com]可以通过SOAP（见19.2.1节）或REST访问。这可以使第三方应用程序在Amazon.com提供的服务之上开发增值服务。例如，一个库存控制和购买应用程序可以如从Amazon.com订购一样订购各种商品的供应，并自动追踪每个订单的状态变化。自从引入这些Web服务后，两年内有超过50 000的开发人员注册使用这项

Web服务[Greenfield and Dornan 2004]。

使用Web服务的应用程序另一个例子是实现了eBay拍卖中称为“snipping”的应用程序。snipping的意思是在拍卖结束前的最后几秒钟内投标。虽然用户可以在网页上执行相同的动作，但却不一定那么迅速。

Web服务的组合 提供的服务接口允许其操作与其他服务的操作组合来提供新的功能。实际上，上面提到的购买程序可能也使用其他供应商的服务。另一个体现服务组合优点的例子是：考虑到人们目前使用浏览器选择不同网站订购机票、酒店和租赁汽车，如果每个网站都提供标准的Web服务接口，那么“旅行代理服务”可以使用它们的操作为旅行者提供上述服务的一个组合，如图19-2所示。

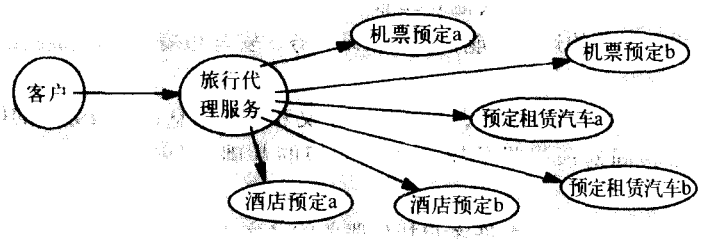


图19-2 组合其他Web服务的“旅行代理服务”

通信模式 “旅行代理”服务阐述了Web服务中可以使用的两种可以替换的通信模式：

- 处理完成预定工作需要很长时间，这可以通过文档的异步交换获得良好的支持，处理时从日期和目的地的详细信息开始，然后是不时返回状态信息直至返回完成信息。这里不考虑性能。
- 信用卡信息的检查和与客户的交互应由请求-应答协议支持。

一般来说，Web服务要么使用与其客户通信的同步请求-应答模式，要么通过异步消息进行通信。后一种通信方式甚至可以用在请求需要应答时，在这种情况下，客户发送请求，然后异步接收应答。此外，还可以使用事件方式。例如，目录服务的客户注册其感兴趣的事件，当某个事件发生时将会通知客户。这个事件可能是某个服务到达或离开。

考虑到多种通信模式，SOAP协议（见19.2.1节）基于单向消息的封装。通过使用单向消息对并指定如何表示操作、操作的参数和结果来支持请求-应答交互。

没有特定的编程模型 Web服务是为在使用多种不同的编程语言的因特网环境下支持分布式计算而设计的，它独立于任何特定的编程范型。它与分布式对象模型的主要区别如下：

- 不可实例化远程对象——事实上，Web服务由单个远程对象组成，因此
- 不存在垃圾收集
- 不存在远程对象引用

我们将在19.2.2节中对Web服务与分布式对象进行比较。REST方法提倡Web服务应具有较少的接口，如后面的REST部分的介绍所示。根据Greenfield和Dornan[2004]的统计，有80%对Amazon.com上Web服务请求是通过REST接口完成的，其余的20%是使用SOAP完成的。

786
/
787

消息表示 SOAP及其携带的数据都是用4.3.2节介绍的XML表示的，XML是一种自描述的文本格式。文本表示比二进制表示占用的空间更多，也需要更多的处理时间。在文档方式的交互中不考虑速度的问题，但在请求-应答交互中速度却十分重要。然而，使用可读的格式也有优势，它可以很容易地构造简单消息并调试更复杂的消息，还允许用户在此之前查看消息中的文本。但在某些情况下这种方式的速度太慢。

XML描述中的每一项都标注了其类型，每种类型的含义由描述中引用的模式定义。这使得格式具有可扩展性，可以传输任何类型的数据。关于XML格式文档的潜在丰富性和复杂性并没有限

制，但在解释一些过于复杂的文档时可能存在困难。

服务引用 一般来说，每个Web服务都有一个URI，客户可以使用该URI来访问服务。URI最常用的形式是URL。由于URL包含计算机的域名，因此始终可以访问其指向的在该计算机上的服务。然而，使用URN的Web服务的访问点依赖于上下文，并且有时会发生改变，其当前URL可以通过URN查找服务获得。

服务的激活 Web服务可以通过其域名包含在当前URL中的计算机访问。该计算机可能自己运行该服务，也可能在另一台服务器计算机上运行该服务。例如，一个拥有上万个客户的服务提供者需要部署几百台计算机来提供服务。Web服务可以连续运行，也可以只在需要时激活。URL是持久性引用，这意味着只要服务器存在它将永远指向某个服务。

透明性 许多中间件平台的主要任务是将程序员从数据表示和编码以及远程访问本地化的细节中解脱出来。而这些在Web服务的基础结构或平台中都没有提供。在最简单的层次上，客户和服务器直接以SOAP格式使用XML读写消息。

但是为了方便起见，SOAP和XML的细节通常被某种编程语言的本地API隐藏，如Java、Perl、Python或C++。在这种情况下，服务描述可以作为自动生成所需编码和解码程序的基础。

代理：可以通过提供客户代理或存根过程集来隐藏本地和远程调用的区别。19.2.3节给出了Java对此的实现方法。客户代理或存根提供调用的静态形式，它在任何调用前生成每个调用的框架和编码过程。

动态调用：代理的一个替代方法是给客户提供一个通用操作，在使用时不用考虑要调用的远程过程，这与图4-15中定义的DoOperation过程类似（但没有第一个参数）。在这种情况下，客户指定操作的名称和参数，并在运行中将其转化为SOAP和XML。类似地，可以通过提供给客户发送和接收消息的通用操作来实现单条消息的异步通信。

> **REST（代表性状态传输）** REST[Fielding 2000]是一种具有受约束样式的操作的方法。在该方法中，客户使用URL 和HTTP 操作GET 、PUT、DELETE和POST来操纵用XML表示的资源。重点是数据资源的操纵而不是接口，是将资源的完整状态提供给客户，而不是调用某个操作提供一部分状态。Fielding认为在因特网中，繁衍不同的服务接口不如使用操作的一个简单、最小、统一的操作集更为有用。在创建一个新的资源时，它获得一个新的URL，使用该URL可以对其进行访问或更新。

19.2.1 SOAP

SOAP旨在因特网上实现客户-服务器以及异步交互。它定义了使用XML表示请求和应答消息的模式（见图4-16），也定义了文档通信的模式。最初SOAP仅仅基于HTTP，但是当前的版本旨在使用各种传输协议，包括SMTP、TCP或UDP。本节的描述是基于SOAP版本1.2[www.w3.org IX]，它是万维网联盟（W3C）的推荐标准。SOAP是Userland XML-RPC[Winer 1999]的扩展。

SOAP规约规定了：

- 如何使用XML表示一条消息的内容。
- 如何组合一个消息对来生成请求-应答模式。
- 消息的接收者如何处理消息中的XML元素的有关规则。
- HTTP和SMTP如何传送SOAP消息。希望将来的规范能够定义如何使用其他传输协议，如TCP。

本节描述SOAP如何使用XML表示消息以及如何使用HTTP传送消息。但是，程序员通常不需要关心这些细节，因为SOAP API已经在很多编程语言中得到实现，这些语言包括Java、Javascript、

Perl、Python、.Net、C、C++、C#和Visual Basic。

为支持客户-服务器通信，SOAP规定如何对请求消息使用HTTP POST方法和其对应答消息的响应。XML和HTTP的组合使用为因特网上的客户-服务器通信提供了标准的协议。

SOAP消息在传送到管理要访问的资源的计算机的途中需要经过一些中间节点，高层中间件服务（例如，事务服务或安全服务）可以使用这些中间节点来执行处理。

SOAP消息　SOAP消息装载在一个“信封”中。在信封内有一个可选的头部和主体。如图19-3所示，消息头部可以用于建立服务所需的上下文或维持操作的日志或审计。某个中间节点可以解释并作用于消息头部的信息，如增加、更改或删除消息。消息主体携带某个Web服务的XML文档。



图19-3　envelope中的SOAP消息

XML元素envelope、header和body以及SOAP消息的其他属性和元素一起被定义为SOAP XML名字空间中的模式。有关该模式的定义可以查阅W3C的网站[www.w3.org]。由于使用了文本编码，XML模式可以使用浏览器中的“查看源文件”选项查看。头部和主体都包含内部元素。

前一节已经说过，服务描述包含客户和服务器要共享的信息。消息发送者使用这些描述生成body，并确保其包含了正确的内容。消息接收者使用这些描述分析并检查内容的有效性。

SOAP消息可以用于传送文档或支持客户-服务器通信：

- 将要传送的文档直接放在body元素中，并将对包含服务描述的XML模式的引用同时放入body元素中，该模式定义了文档中使用的名称和类型。这种类型的SOAP消息可以同步或异步发送。
- 对于客户-服务器通信，body元素包含一个Request或Reply。这两种情况均在图19-4和图19-5中说明。



在本例和下图中，每个XML元素都用矩形表示，其名称为斜体，后跟属性和内容。

图19-4　一个没有头部的简单请求示例

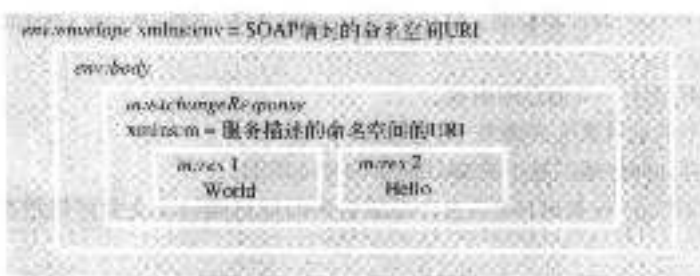


图19-5　对应于图19-4中请求的应答示例

图19-4给出了一个没有头部的简单请求消息。body中封装了元素m，以及要调用的过程的名称以及相关服务描述的名字空间（包含XML模式的文件）的URI。请求消息的内部元素包含过程的参数，该请求消息提供了两个字符串，服务器上的过程以相反的顺序返回这两个字符串。用env表示的XML名字空间包含envelope的SOAP定义。图19-5给出了相应的成功应答消息，其中包含两个输出参数。注意，在过程的名称上增加了“Response”。如果过程有返回值，则可以表示为元素rpc:result。注意，应答消息与请求消息使用相同的两个XML模式，第一个模式定义了SOAP信封，第二个模式定义了特定于应用程序的过程和参数名称。

SOAP故障　如果请求在某种情况下失败了，则在应答消息的主体中用fault元素传送故障描述。该元素包含故障的相关信息，包括代码和相关字符串以及特定于应用程序的细节。

SOAP头部　消息头部可由中间节点添加到处理装载在相应主体中消息的服务中。然而，这种用法有两个方面在SOAP规约中尚不清晰：

788 ~ 791

1）头部如何被某种更高层中间件服务使用。例如，头部可能包含：

- 事务服务使用的事务标识符。
- 消息之间互相关联（例如，实现可靠传输）的消息标识符。
- 用户名、数字签名或公钥。

2）消息如何经由中间节点集路由到最终接收者。例如，由HTTP传输的消息可以经由一系列代理服务器，其中有些可能作为SOAP。

然而，规约规定了中间节点的角色和职责。角色属性可以指定每个中间节点都处理元素或者都不处理元素，或者只是最终接收者处理元素（见[www.w3.org]）。要执行的某个动作由应用程序定义，例如某个动作可以是记录元素的内容。

SOAP消息的传输　需要使用传输协议将SOAP消息发送到其目的地。SOAP消息独立于使用的传输类型——消息的信封不包含对目的地址的引用。目的地址由HTTP或其他用于传输SOAP消息的协议指定。

图19-6阐述了如何使用HTTP POST方法传输SOAP消息。HTTP头部和主体的作用如下：

- HTTP头部指定端点地址（最终接收者的URI）和要执行的动作。Action参数用于最优化调度——通过给出操作的名称而不需要分析HTTP消息主体中的SOAP消息。
- HTTP主体封装SOAP消息。



图19-6　SOAP客户-服务器通信中HTTP POST Request的使用

由于HTTP是同步协议，它用于返回一个包含SOAP应答的应答。如图19-5所示，4.4节详述了对于成功请求和失败请求，HTTP返回的状态代码和原因。

792

如果一个SOAP Request仅仅是要返回信息的请求，它不含任何参数并且不改变服务器中的数据，那么可以使用HTTP GET方法来执行它。

上面关于Action头部和调度的特点适用于任何执行客户不同动作的服务，即使这个服务不提供

这样的操作。例如，Web服务可以处理大量不同类型的文档，如购买订单和询问，它们是由不同的软件模块处理的。Action头部使得可以不需要检查SOAP消息就能选择正确的模块。这个头部可以在HTTP内容类型指定为application/soap+xml时使用。

SOAP信封的定义和有关如何发送以及发送目的地信息的分离使得使用多种不同的底层协议成为可能。SOAP规约规定了SMTP如何作为一种替代协议传输编码为SOAP消息的文档。

但是这个优势往往是个弱点。这意味着开发者必须考虑所选择的特定传输协议的细节中。另外，这为某个消息经过的路由的不同部分使用不同协议带来了困难。

SOAP寻址和路由的进展 前面提到了两个问题：

- 如何使SOAP独立于底层使用的传输协议。
- 如何指定SOAP通过中间节点集时遵循的路由。

该领域早期的成果为Nielsen和Thatte[2001]提出的WS-Routing，它建议应该在SOAP头部指定端点地址和调度信息。这样做能有效地将消息目的地同底层协议分开。他们提出通过给出端点地址和“下一跳”地址来指定要遵循的路径。每个中间节点都将更新“下一跳”信息。

在Box和Curbera等[2004]中报告的最新成果称为WS-Addressing，它认为允许中间节点改变头部信息会破坏安全性。他们推荐了一种替代方法：头部指定端点的地址，底层的SOAP基础设施提供“下一跳”信息。另外，他们建议使用SOAP头部指明返回地址和消息标识符。

可靠通信 Web服务缺乏一种在出现故障时可靠地传送消息的协议。SOAP常用的HTTP协议运行在TCP之上，在4.2.4节中已讨论了TCP的故障模型。总结如下：TCP不能保证在所有情况下都能正确传送消息——如果等待确认的时间超时，TCP就会声明连接已经中断，这时通信进程不知道最近发送的消息是否被接收。

在提供SOAP消息的有保证传送、无重复、保证消息顺序的可靠通信方面已经取得了一些成果。
793 Ferris和Langworthy[2004]以及Evans等[2003]分别提出了两种相互竞争的规范。

检查已建立的容错措施（如确认、请求的重传、过滤重复和序列号）是否对大规模的异构的因特网环境有效，这是一件很有意思的事情。特别是如何选择超时时间将会是一个挑战。

穿越防火墙 Web服务也可被一个组织中的客户用于通过因特网访问另一个组织中的服务器。大多数组织使用防火墙来保护它们网络上的资源，Java RMI或CORBA使用的传输协议通常不能够穿越防火墙。然而，防火墙通常允许HTTP和SMTP消息通过。因此，使用这两个协议之一来传输SOAP消息是非常方便的。

19.2.2 Web服务与分布式对象模型的比较

Web服务具有一个服务接口，该接口可提供操作来访问和更新其管理的数据资源。从表面上看，客户和服务器之间的交互十分类似于RMI，在RMI中客户使用远程对象引用来调用远程对象中的操作。对于Web服务，客户使用URI来调用该URI标识的资源中的操作。有关Web服务和分布式对象之间的相似点和区别的讨论，参见Birman[2004]、Vinoski[2002]和Vogels[2003]。通过5.5节Java RMI和20.2节CORBA中使用的共享白板的例子，我们试图给出上面分析的局限性。

远程对象引用与URI不是十分类似 Web服务的URI的作用与单个对象的远程对象引用的作用看似相似。然而，在分布式对象模型中，对象可以动态创建远程对象并返回其远程引用。接收者可以使用这些远程引用调用所引用的对象中的操作。在共享白板的例子中，对 newShape工厂方法的调用创建了Shape对象的一个新实例，并返回对实例的远程引用。不能创建远程对象的实例的Web服务不能完成类似的工作。

Web服务模型 考虑到不是使用透明的Java到Java的远程调用，而是使用Web服务模型，在该模型中远程对象不能实例化，因此Java Web服务工具包(JAX-RPC)[java.sun.com VII]的用户必须为

Web服务程序建模使其满足这个限制。JAX-RPC考虑到这一点，它不允许将远程对象引用作为参数传递或作为结果返回。

图19-7给出了图5-12中接口的另一个版本，如下修改该接口使其成为Web服务接口：

- 在程序最初（分布式对象）的版本中，在服务器中创建Shape的实例，并由newShape返回对这些实例的远程引用，newShape的改进（Web服务）版本如行1所示。为避免远程对象的实例化以及相应的对远程对象的使用，删除了Shape接口并将其操作（getAllState和getGOVersion——最初的getVersion）添加到ShapeList接口中。 794
- 在程序最初（分布式对象）的版本中，服务器存储了Shape的一个向量。现在将其更改为GraphicalObject的一个向量。方法newShape的新（Web服务）版本返回一个整数，它表示该向量中GraphicalObject的偏移量。

```
import java.rmi.*;
public interface ShapeList extends Remote {
    int newShape(GraphicalObject g) throws RemoteException;        1
    int numberOfShapes()throws RemoteException;
    int getVersion() throws RemoteException;
    int getGOVersion(int i)throws RemoteException;
    GraphicalObject getAllState(int i) throws RemoteException;
}
```

图19-7 Java Web服务接口ShapeList

修改方法newShape意味着它不再是工厂方法，也就是说，该方法不再创建远程对象的实例。

伺服器 在分布式对象模型中，服务器程序通常被建模化为伺服器的集合（潜在的远程对象）。例如，共享白板应用程序对形状列表使用一个伺服器，并为创建的每个图形对象使用一个伺服器。将这些伺服器分别创建为伺服器类ShapeList和Shape的实例。服务器启动时，其main函数创建ShapeList的实例；每当客户调用newShape方法时，服务器创建Shape的一个实例。

相反，Web服务不支持伺服器。因此，在需要处理不同的服务器资源时，Web服务应用程序不能创建伺服器。为实施这一点，Web服务接口的实现既没有构造函数也没有main方法。

19.2.3 在Java中使用SOAP

用于开发SOAP上的Web服务和客户端的Java API称为JAX-RPC，在Java Web服务教程[java.sun.com VII]中介绍了JAX-RPC。该API对客户和服务的编程人员隐藏了SOAP的所有细节。

JAX-RPC将Java语言中的某些类型映射到SOAP消息和服务描述使用的XML中的定义上。允许使用的类型包括Integer、String、Date和Calendar以及java.net.uri，它允许将URI作为参数传递或作为结果返回。JAX-RPC不仅支持语言的简单类型和数组，还支持某些集合类型（包括Vector）。

另外，某些类的实例可以作为参数传递，也可以作为远程调用的结果传递，前提是：

- 每个实例变量都是上述所允许的类型之一。
- 都拥有公共的默认构造函数。
- 没有实现Remote接口。 795

一般来说（如前一节提到的），远程引用（即实现了Remote接口）的类型的值不能作为参数传递或作为远程调用的结果返回。

服务接口 Web服务的Java接口必须遵循以下规则，其中某些规则如图19-7所示：

- 必须扩展Remote接口。
- 必须不含常量声明，例如， public final static。

- 方法必须抛出java.rmi.RemoteException或其子类异常。
- 方法参数和返回类型必须符合JAX-RPC类型。

服务器程序 实现ShapeList接口的类如图19-8所示。如上所述，该类不含main方法，ShapeList接口的实现也不包含构造函数。实际上，Web服务是提供一组过程的单个对象。图19-7、图19-8和图19-9中的源程序可以从本书的网站www.cdk4.net/Webs上获得。

```
import java.util.Vector;
public class ShapeListImpl implements ShapeList {
    private Vector theList = new Vector();
    private int version = 0;
    private Vector theVersions = new Vector();

    public int newShape(GraphicalObject g) throwsRemoteException{
        version++;
        theList.addElement(g);
        theVersions.addElement(new Integer(version));
        return theList.size();
    }
    public int numberOfShapes(){}
    public int getVersion() {}
    public int getGOVersion(int i){ }
    public GraphicalObject getAllState(int i) {}
}
```

图19-8 ShapeList服务器的Java实现

```
package staticstub;
import javax.xml.rpc.Stub;
public class ShapeListClient {
    public static void main(String[] args) { /* pass URL of service */
      try {
        Stub proxy = createProxy();                                    1
        proxy._setProperty                                             2
          (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
        ShapeList aShapeList = (ShapeList)proxy;                       3
        GraphicalObject g = aShapeList.getAllState(0);                 4
      } catch (Exception ex) { ex.printStackTrace(); }
    }

    private static Stub createProxy() {                                5
        return
            (Stub) (new MyShapeListService_Impl().getShapeListPort()); 6
  }
}
```

图19-9 ShapeList客户端的Java实现

服务的接口和实现按通常方式编译。可以使用两个工具wscompile和wsdeploy生成骨架类和WSDL格式的服务描述，这将用到与服务的URL有关的信息、XML格式的配置文件中的服务名称和描述。服务的名称（在该例中为MyShapeListService）用于生成客户程序访问该服务时使用的类的名称，即MyShapeListService_Impl。

Servlet容器 服务实现作为servlet运行在Servlet容器中。Servlet容器的作用是加载、初始化并执行servlet。它包括分发器和骨架（见5.2.5节）。当请求到达时，分发器将该请求映射到某个骨架，

该骨架将请求转换为Java格式并传送给servlet中的适当方法，由该方法执行请求并生成应答，然后骨架再将该应答转换为SOAP应答。服务的URL由servlet容器的URL以及服务类别和名称连接而成，如http://localhost:8080/ShapeList-jaxrpc/ShapeList。

Tomcat[Apache 2004]是一个常用的servlet容器。运行Tomcat时，可以使用浏览器输入一个URL来查看其管理界面。该界面显示了当前部署的servlet的名称，并提供了一系列操作来管理这些servlet，以及获取每个servlet的包括服务描述在内的相关信息。一旦在Tomcat中部署了某个servlet，客户就可以访问该servlet，其操作的组合效果将存储在servlet的实例变量中。在我们的例子中生成了GraphicalObjects的一个列表，在客户请求newShape操作后，将每个GraphicalObjects作为客户请求newShape的结果添加入该列表。如果通过Tomcat管理界面停止某个servlet，那么在重启该servlet时实例变量的值将被重置。

Tomcat还提供对其包含的每个服务的服务描述的访问，这使得编程人员能够设计客户程序，并使客户代码请求的代理的自动编译更加方便。由于服务描述是用XML表述的，因此是人可读的。

客户程序 客户程序可以使用静态代理、动态代理或动态调用接口。在各种情况下，都可以从相关的服务描述中获得客户代码所需的信息。在我们的例子中，服务描述可以从Tomcat中获得。

静态代理：图19-9显示了ShapeList客户通过代理（将消息发送到远程服务的本地对象）发起调用。代理的代码由wscompile从服务描述中生成。代理的类名是在服务的名称后添加“_Impl”得到的。在本例中，代理类称为MyShapeListService_Impl。该名称是特定于实现的，因为SOAP规范中并没有给出代理类的命名规则。

在程序的第1行，调用creatProxy方法。该方法如第5行所示，在第6行它使用MyShapeListService_Impl创建了一个代理，然后返回了该代理（注意，由于代理有时被称为存根，所以出现类Stub的名称）。在第2行，通过命令行给出的参数将服务的URL提供给代理。在第3行，将proxy的类型强制转换为ShapeList接口的类型。第4行调用了远程过程getAllState，请求服务返回GraphicalObjects的Vector中第0个元素的对象。

由于代理是在编译时创建的，因此这种代理称为静态代理。从中生成的服务的服务描述不一定是从Java接口中生成的，它可以由各种不同语言系统的相关工具生成，甚至可以直接用XML写成。

动态代理：除了可以使用预编译的静态代理外，客户也可以使用动态代理。动态代理的类是在运行时从服务描述和服务接口的信息中生成的。这种方法避免了代理类使用特定于实现的名称的问题。

动态调用接口：允许客户调用远程过程，即使服务的基调或服务的名称在运行前是未知的。与以上方法不同的是，客户不需要代理，但必须在发起过程调用前使用一系列操作来设置服务器操作的名称、返回值以及每个参数。

Java SOAP的实现 Java API的实现方式可以用图5-7来阐述。以下各段说明了Java/SOAP环境中各个组件的作用——组件之间的相互作用跟以前一样。不存在远程引用模块。

通信模块：通信模块的任务由一对HTTP模块实现。服务器中的HTTP模块根据action头部中的URL为POST请求选择合适的分发器。

客户代理：代理（或存根）方法知道服务的URL，并将其方法的名称和参数与对该服务的XML模式的引用一起编码到SOAP请求信封中。对应答的解码包括分析SOAP信封来抽取结果、返回值或故障报告。将客户的请求方法调用作为HTTP请求发送到服务。

分发器和骨架：如上所述，分发器和骨架存在于servlet容器中。分发器从HTTP请求的action头部抽取操作的名称，然后调用合适的骨架中的相应方法，并将SOAP信封传递给该方法。骨架方法执行以下任务：分析请求消息中的SOAP信封，然后抽取其参数，调用相应的方法，生成包含结果的SOAP应答信封。

796 ~ 798

SOAP/XML中的错误、故障和正确性：HTTP模块、分发器、骨架或服务自身都可以报告故

障。服务可以通过返回值或利用服务描述中指定的故障参数报告其错误。骨架负责检查SOAP信封包含的请求以及XML（SOAP信封用良构的XML写成）。确认XML的有效性后，骨架将使用信封中的XML名字空间检查请求是否与提供的服务对应以及操作和其参数是否适合。如果请求验证在上述阶段中的任一阶段失败，则向客户返回错误。在接收到包含结果的SOAP信封时，代理也进行类似的检查。

19.2.4 Web服务和CORBA的比较

Web服务和CORBA或其他类似的中间件的主要区别是使用它们的上下文不同。CORBA用于单个组织或很少的几个协作组织。这导致设计的某些方面过于集中，不适合独立组织协作使用或在没有预先安排时的自主使用，下面我们将解释这个问题。

名字问题 在CORBA中，通过CORBA名字服务（见20.3.1节）的实例管理的名称来引用每个远程对象。这种类似DNS的服务提供名称到表示地址的值（CORBA中为IOR）之间的映射。但与DNS不同的是，CORBA名字服务旨在用于一个组织而不是整个因特网。

在CORBA名字服务中，每个服务器管理一个具有初始的名字上下文的名称图，该服务器最初独立于其他服务器。尽管各个组织可能联合他们的名字服务，但这不能自动完成。在一个服务器与另一个服务器联合之前，需要知道后者的初始名字上下文。因此，CORBA名字服务的设计将CORBA对象的共享有效地限制到已经联合名字服务的几个组织中。

引用问题 现在考虑CORBA远程对象引用（称之为IOR，见20.2.4节）是否可以以类似URL的方式用作因特网范围内的对象引用。每个IOR包含一个槽，指定其引用的对象的接口的类型标识符。不过，只有存储相应类型定义的接口库才能理解这个类型标识符。这意味着客户和服务器需要使用相同的接口库，这在全球范围应用中实际上并不可行。

相反，在Web服务模型中，服务由URL标识，这使得处于因特网上任何位置的客户都可以请求位于因特网其他位置上的组织中的服务。也就是说，客户可以通过因特网共享Web服务。URL
799 访问唯一需要的服务就是DNS，而DNS可以在因特网范围内有效地工作。

激活和定位的分离 定位和激活Web服务的任务被巧妙地分离开来。相反，CORBA持久引用指的是平台的一个组件（实现仓库），它在任何合适的计算机上按需激活相应的对象，一旦激活对象，它还将负责定位对象。

易用性 虽然用户需要一种方便的编程语言API用于SOAP通信，但Web服务的HTTP和XML基础结构易于理解和使用，并且已经安装在所有最常用的操作系统上。相反，CORBA平台需要安装和支持庞大复杂的软件。

效率 CORBA是比较高效的：CORBA CDR（4.3.1节）是二进制的，而XML是文本方式的。Olson和Ogbuji[2002]的一项研究比较了CORBA、SOAP和XML-RPC的性能。他们发现，SOAP请求消息的大小是CORBA中等价消息的14倍，SOAP请求耗费的时间是CORBA调用的882倍。但在某些应用中，SOAP的消息开销和较低的性能并不易觉察，由于廉价的带宽、处理器、内存和磁盘空间越来越普及，SOAP的低效变得更不明显。

W3C等组织一直在研究允许XML元素包括二进制数据以提高效率的可行性。有关该主题的讨论请参见www.w3.org。请注意，XML已经提供了二进制数据的十六进制和base64表示。Base64表示与XML加密联合使用（见19.5节）。将二进制数据转换为Base64或十六进制数据的时间和空间开销相当大。因此，真正需要的是能够包括数据项（如CORBA CDR或gzip生成的）的预解析顺序的二进制表示。另一个正在研究的方法是与附件一起发送SOAP消息（其中某些附件可能是二进制的）并使用复合MIME文本传输它。然而，最后一种方法并不符合SOAP模型。

CORBA的优势 CORBA服务在事务、并发控制、安全和访问控制、事件和持久对象方面的

优势使其成为在单个组织内或者相关的几个组织内使用的许多应用程序中的合适选择。通常，CORBA非常适合具有复杂交互的应用。另外，分布式对象模型对设计复杂应用很有吸引力，因此值得花费一些精力去理解CORBA对象模型（20.2节）和使用的特定编程语言之间的关系。

19.3 服务描述和Web服务接口定义语言

客户与服务进行通信需要使用接口定义。对于Web服务，接口定义是通常的服务描述的一部分，服务描述还指定了另外两个特性——消息如何通信（如通过HTTP上的SOAP）以及服务的URI。为满足多语言环境中的使用，服务描述使用XML编写。 800

服务描述构成了客户和服务器之间对提供的服务达成共识的基础，它聚集了服务方面所有与客户有关的因素。服务描述通常用于生成客户端存根，存根将自动为客户实现正确的行为。

服务描述的类似接口定义语言（IDL）组件比其他IDL更为灵活，因为服务可以按照发送或接收消息的类型指定，也可以根据所支持的操作指定，从而允许文档交换以及请求-应答形式的交互。

Web服务及其客户可以使用很多不同方法进行通信，因此通信方法由服务提供者决定并在服务描述中说明，而不是像CORBA一样将其构建进系统。

将服务的URI写入服务描述可以避免使用大部分其他中间件使用的单独的绑定器或名字服务。这意味着一旦服务描述对潜在的客户公开，将不能更改其URI。但URN模式通过引用层上的间接性从而允许位置的变化。

相反，在绑定器方法中，客户在运行时使用名称来查找服务引用，因而允许随时更改服务引用。但是这种方法需要在所有服务的名称和服务引用之间存在一个间接层，即使许多服务依旧在相同位置也是如此。

在Web服务环境中，Web服务描述语言（WSDL）通常用于服务描述，在编写本书时，WSDL 2.0 [www.w3.org XI]就是W3C的工作草案。它定义了表示服务描述组件的XML模式，包括诸如名为definition、type、message、interface、binding和service的元素。

WSDL将服务描述的抽象部分与具体部分分开，如图19-10所示。

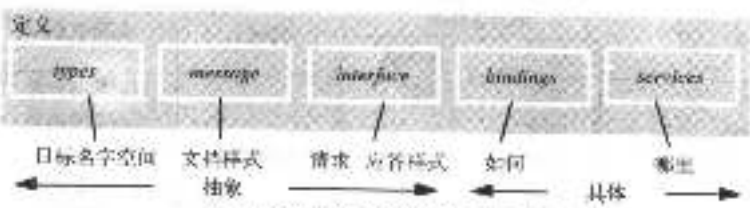


图19-10 WSDL描述中的主要元素

描述的抽象部分包括服务使用的一组类型的定义，特别是消息中交换的值的类型。19.2.3节中的Java示例的接口如图19-7所示，它使用了Java类型int和GraphicalObject，前者（跟其他基本类型一样）可以直接转换成XML中相应的类型，但GraphicalObject是用Java类型int、String和boolean定义的。为满足异构的客户使用，GraphicalObject用XML表示为复杂类型，由一组命名的XML类型组成，例如包括： 801

```
<element name="isFilled" type="boolean"/>
< element name="originx" type="int"/>
```

WSDL定义的type项内定义的名称集称为它的目标名字空间。抽象部分的message项包含交换的消息集的描述。对于文档样式的交互来说，这些消息将直接被使用。对于请求-应答样式的交互来说，每个操作有两条消息，用于描述interface项中的操作。具体部分指定了如何联系服务以及在哪里联系服务。

WSDL定义的固有的模块性允许其组件以不同的方式组合在一起，例如，相同的接口可以与不

同的绑定或位置一起使用。类型可以在type元素内定义，也可以在type元素中的URI引用的单独文档中定义。在以后的例子中，类型定义可以从几个不同的WSDL文档引用。

消息或操作 在Web服务中，客户和服务器所需要的是对要交换的信息达成共识。若服务只涉及很少几个不同类型的文档的交换，则WSDL只需描述要交换的不同信息的类型。当客户发送这些消息之一到Web服务时，服务基于其收到的消息的类型决定执行何种操作以及给客户返回何种类型的信息。在我们给出的Java例子中，为接口中的每个操作定义两条消息——一条用于请求，一条用于应答。例如，图19-11给出了操作newShape的请求和应答消息。该操作具有一个类型为GraphicalObject的输入参数和类型为int的输出参数。

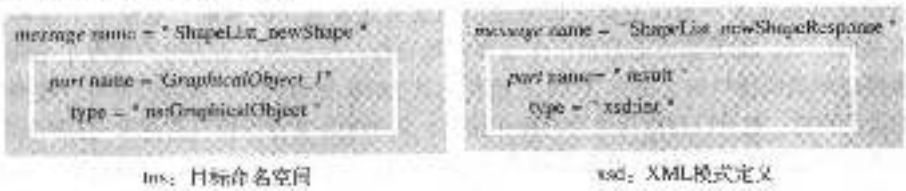


图19-11 newShape操作的WDSL请求和应答消息

但对于支持多个不同操作的服务，将交换的消息指定为对带参数的操作的请求以及相应的应答，允许服务将每个请求分派到合适的操作，这种方法效率更高。然而，在WSDL中，操作由相关的请求和应答消息组成，而不是服务接口中操作的定义构成。

接口 属于同一个Web服务的操作集合分为一组，放在名为interface（有时称为portType）的XML元素中。每个操作必须指定客户和服务器之间消息交换的模式。可选的模式如图19-12所示。第一个模式In-Out是客户/服务器通信常用的RR（请求/应答）形式。在这种模式中，应答消息可以
802 用故障消息代替。In-Only用于带有表决语义的单向消息，Out-Only用于服务器到客户的单向消息，这两种模式都不能同故障消息一起发送。Robust In-Only和Robust Out-Only是相应的有传递保证的消息，这时可以交换故障消息。Out-In是由服务器发起的请求/应答交互。

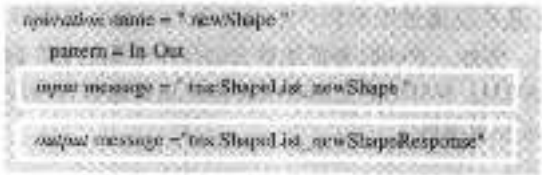
名 称	消息发送者		传递	故障消息
	客户	服务器		
In-Out	请求	应答		可以代替应答
In-Only	请求			没有故障消息
Robust In-Only	请求		有保证的	可以发送
Out-In	应答	请求		可以代替应答
Out-Only		请求		没有故障消息
Robust Out-Only		请求	有保证的	可以发送故障

图19-12 WSDL操作的消息交换模式

回到上面那个Java的例子，每个操作定义为In-Out模式。操作newShape如图19-13所示，该操作使用图19-11定义的消息。这个定义与其他四个操作的定义都嵌套在XML的interface元素中。操作也可以指定可以发送的故障消息。

但是，如果一个操作有两个参数，一个是整数，另一个是字符串，那么就没有必要定义新的数据类型，因为这些类型都已经定义为XML模式了。然而，这样就需要定义包含这两个部分的消息。该消息可以用作操作定义中的输入或输出。

继承 任何WSDL接口都可以扩展一个或多个其他WSDL接口。这是继承的一种简单形式。在继承中接口除了支持自身定义的操作外，还支持其继承的所有接口的操作。接口不允许递归定义。即如果接口B扩展了接口A，则接口A不能再扩展接口B。



tns：目标命名空间　xsd：XML模式定义

在WSDL的XML模式中定义了关于operation, pattern, input和output。

图19-13　WSDL操作newShape

803

具体部分　WSDL的其余部分（具体部分）由binding（协议的选择）和service（端点或服务器地址的选择）组成。这两者是相关联的，因为地址的形式取决于使用的协议类型。例如，SOAP端点使用URI，而CORBA端点使用特定于CORBA的对象标识符。

绑定：WSDL文档中的binding项表示要使用何种消息格式和外部数据表示形式。例如，Web服务通常使用SOAP、HTTP和MIME。但最终还是使用诸如GIOP（见20.1节）来访问CORBA对象的实例。绑定可以与特定操作或接口相关联，也可以供许多不同的Web服务自由使用。

图19-14给出嵌套了一个soap:binding的binding。soap:binding指定了传输SOAP信封的特定协议的URL：SOAP的HTTP绑定。通过该元素可选的属性还可以指定：

- 消息交换的模式，可以是rpc（请求/应答）或document交换模式，默认值为document模式。
- 消息格式的XML模式。默认值为SOAP envelope。
- 外部数据表示的XML模式。默认值为XML的SOAP编码。

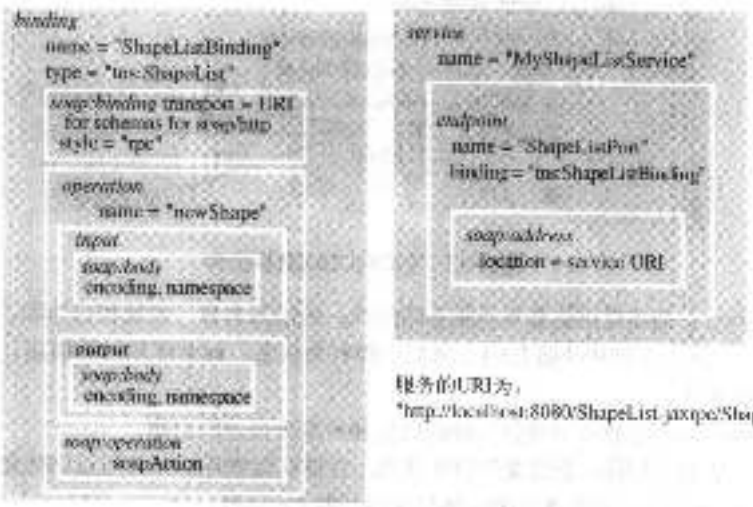


图19-14　SOAP绑定和服务定义

804

图19-14还显示了一个操作newShape的绑定的详细信息，它指定了input和output消息都应在SOAP主体内传输，使用了特定的编码样式，另外操作还应作为SOAP Action传输。

服务：WSDL文档中的每个service元素都指定了服务的名称和一个或多个端点（或端口）。在端点将与服务的某个实例联系。每个endpoint元素引用所使用的绑定的名称，在使用SOAP绑定的情况下，使用一个soap:address元素来指定服务位置的URI。

文档　在WSDL文档内的大部分地方，可以将人与机器都可读的信息插入documentation元素中。在使用WSDL自动处理之前，可以通过stub编译器将该信息删除。

WSDL的用途　直接地或间接地通过UDDI等目录服务，客户和服务器可以使用URI访问完整

的WSDL文档。有很多工具可以通过图形用户界面提供的信息生成WSDL，从而避免了用户涉及WSDL的复杂细节和结构。例如，Web服务描述语言的Java工具包允许创建、表示和操纵WSDL文档来描述服务[WSDL4J 2003]。WSDL定义还可以直接从用其他语言写成的接口定义中生成，例如前面19.2.1节中讨论的Java JAX-RPC。

19.4 Web服务使用的目录服务

客户可以通过很多方法获取服务描述，例如，19.1节讨论的旅行代理服务的高级Web服务的提供者肯定会制作一个网页宣传这个服务，潜在的客户将在查询这种类型的服务时遇到这个网页。

然而，任何计划将Web服务作为其应用基础的组织会发现使用目录服务将更便于客户找到服务。这就是统一目录和发现服务（UDDI）[Bellwood et al. 2003]的目的，它提供名字服务和目录服务（见9.3节），也就是说，可以通过名称（白页服务）或通过属性（黄页服务）查找WSDL服务描述。还可以直接通过URL访问，这大大方便了开发使用该服务的客户程序的开发人员。

客户可以使用黄页方法来查找特定类别的服务，如旅行代理或书商，或者使用白页方法根据提供服务的组织来查找服务。

数据结构 支持UDDI的数据结构旨在允许上述所有访问方式，并且可以结合任意数量的用户可读的信息。数据以图19-15所示的四种结构组织，每种结构都可以通过称为key的标识符单独访问（tModel除外，tModel可以通过URL访问）。

805



图19-15 主要的UDDI数据结构

businessEntity：描述提供这些Web服务的组织，给出其名称、地址和行为等。

businessService：存储Web服务的实例集合的有关信息，如名称和目的描述。前面的旅行代理或书商就是一个例子。

bindingTemplate：存放Web服务实例的地址和对服务描述的引用。

tModel：存放服务描述，通常是WSDL文档，存储在数据库外并且通过URL访问。

查找 UDDI基于以下两个查询操作集提供查找服务的API：

- get_xxx操作集包括get_BusinessDetail、get_ServiceDetail、get_bindingDetail和get_tModelDetail。这些操作根据给定的键检索实体。
- find_xxx操作集包括find_business、find_service、find_binding和find_tModel。这些操作检索匹配特定搜索标准的实体集，并给出名称、描述、键和URL的摘要。

因此，拥有某个键值的客户可以使用get_xxx操作直接检索相应实体。其他客户可以通过浏览来辅助查找——开始时找到一个较大的结果集，然后逐渐缩小搜索范围。例如，可以首先使用find_business操作来获得包含有关匹配提供者的信息摘要的列表。从摘要中，用户可以通过匹配所需服务的类型，使用find_service操作来缩小搜索范围。在这两种情况下，将会找到合适的bindingTemplate的键，并找到一个URL来检索合适的服务的WSDL文档。

806

另外，UDDI提供了通知/订阅接口。通过该接口，客户可以在UDDI注册表中注册感兴趣的实体集，并以同步或异步方式获得变更通知。

发布 UDDI提供了一个接口用来发布和更新Web服务的信息。当一个数据结构（见图19-5）第一次在某个UDDI服务器上发布时，该结构获得一个URI形式的键，例如uddi:cdk4.net:213，并且该服务器成为其所有者。

注册处 UDDI服务基于注册处中存储的复制数据。UDDI注册处由一个或多个UDDI服务器组成，每个服务器都有相同数据集的副本。数据将在注册处的成员间复制。每一个成员都可以响应查询并发布信息。对某个数据结构的更改必须提交到其所有者，也就是该结构第一次发布时所在的服务器。所有者可以将所有权传给同一注册处的其他UDDI服务器。

复制模式：注册处的成员按照如下方式相互传播数据结构的副本：进行了变更的服务器通知注册处的其他服务器，然后请求做出更改。使用一个向量时间戳表来确定应传播并应用的更改。与其他使用向量时间戳的复制模式（如15.4.1节的Gossip或15.4.3节的Coda）相比，这个模式十分简单，因为：

1）对某个数据结构的所有更改都在同一个服务器上进行。

2）来自某个服务器的更新按顺序被其他成员接收，但是不同服务器所做的更新之间不存在特定的顺序。

服务器之间的交互：如上所述，服务器之间通过交互来完成复制。还可以通过交互来传递数据结构的所有权。然而，对查询操作的响应由单个服务器执行，而不需要与注册处其他服务器进行任何交互，这一点与X.500目录服务（见9.5节）不同。在X.500目录服务中，数据被分布到服务器上，服务器通过互相协作来查找与特定请求相关的服务器。

19.5 XML安全性

XML安全性由一组相关的W3C提出的用于签名、密钥管理和加密的设计组成。它用于因特网上的协作，因为因特网上的文档内容可能需要认证或加密。通常，文档被创建、交换、存储，然后再次交换，这其中很可能文档被一系列不同用户修改。

WS-Security[Kaler 2002]是另一种获得安全性的方法，该方法将消息完整性、消息机密性以及单个消息的认证应用到SOAP。

考虑一个包含病人病历的文档，在这个场景中XML安全性将十分有用。在本地医生的诊疗室以及在病人去过的不同的诊所和医院分别用到病历文档的不同部分。医生、护士、医疗顾问根据 807
病情和治疗方法的历史记录来更新该文档，另外负责预约的管理人员以及提供药品的药剂师也将更新该文档。上面提到的不同角色可以查看文档的不同部分，病人也可能查看文档。将文档的某个部分（如关于治疗的建议等）归属到做出这些建议的人，并保证这个部分不被其他人更改，这种做法是很有必要的。

TLS（即从前的SSL，见7.6.3节）可以用于创建信息通信的安全通道，但它不能满足上面的需要。在通道建立的过程中以及通道的生命周期内，TLS允许通道两端的进程对认证的需求、加密和密钥以及用到的算法进行协商。例如，可以对有关金融交易的数据进行签名，然后以明文传送，直到需要传送诸如信用卡信息等敏感信息时才应用加密。

考虑到上面列出的新的用途，必须在文档内指定安全性并应用安全性，而不是将安全性作为将文档从一个用户传送到另一个用户的通道的属性。

这可以用XML以及其他结构化文档格式实现，因为在这些文档格式内可以使用元数据。XML标记可以用于定义文档中数据的属性。特别是，XML安全性依赖于可以用于指示加密或签名数据以及签名项的起止位置的新标记。一旦将必须的安全性应用到文档内，就可以将其发送到大量不

同的用户，甚至是通过组播的方式发送。

基本需求 XML安全性至少应该提供与TLS相同等级的保护，即

既能够加密整个文档，也能选择文档的某些部分进行加密：例如，考虑金融交易中的信息，包括姓名、交易类型以及使用的信用卡或借记卡的信息。一种情况是，只是将卡的信息隐藏，从而能够在解密记录之前确认交易。另一种情况是，将交易的类型也隐藏起来，这样外部的人就不能分辨这到底是订单还是付款。

既能够对整个文档签名，也能够只选择文档的某些部分签名：当文档要用于一组人的协同工作时，应该对文档中的某些关键部分签名，以保证这些部分由某个人做出修改或没有做出更改。但是文档中可以有些部分能够在使用期间更改，这是很有用的，这些部分不应该被签名。

其他基本需求 有时需要存储文档、可能修改文档然后将其发送到许多不同的接收者，这一过程产生了新的需求：

在已签名的文档上增加内容并对结果签名：例如，Alice对一个文档签名并将其传给Bob，Bob
808 通过对其添加一个标记来证明Alice的签名，然后对整个文档签名。

在包含加密部分的文档上增加内容并对新版本文档的某个部分加密，其中可能包含某些已加密的部分。

授权不同的用户来查看文档的不同部分：在病历的例子中，调查人员可以查看医疗记录的某个特定部分，管理人员可以查看个人信息，医生可以查看这两个部分。

XML记号的灵活性和结构化能力使得它能满足上述所有的需求，不需要对满足基本需求的机制做任何增加。

算法需求 XML安全文档在考虑谁将访问文档之前已经进行了签名和加密。如果没有涉及文档的创作者，就不可能对协议以及是否使用验证或加密进行协商。因此：

标准应该指定一套在任何XML的安全性实现中都提供的算法：至少应该强制提供一个加密算法和签名算法，从而实现最大可能的互操作性。应该尽可能少的提供其他可选的算法。

用于特定文档的加密和认证的算法必须从这一套算法中选择，使用的算法的名称必须在XML文档自身内引用：如果文档使用的环境不可预测，则必须使用一个所需的协议。

XML安全性定义了元素名字，这些名字可以用于指定签名或加密所用的算法的URI。因此能够在相同的XML文档内选择多种算法，指定算法的元素通常嵌套在包含签名信息或加密数据的元素内。

查找密钥的需求 当创建文档以及每一次更新文档时，都必须选择合适的密钥，而不必与以后可能访问该文档的人进行任何协商。这引发了以下需求：

帮助安全文档的用户查找必需的密钥：例如，包括签名数据的文档应该包含用来验证签名的公钥信息，如可以用于获取密钥的名称，或者一个证书。KeyInfo元素可以用于此目的。

使协作用户能够彼此帮助查找密钥：假如KeyInfo元素没有以加密方式绑定到签名上，则应在不破坏数字签名的情况下添加信息。例如，Alice对一个文档签名后将其发送给Bob，该文档中包含一个仅指定密钥的名称的KeyInfo元素。当Bob收到文档时，他检索验证签名所用的信息并在将文档发送到Carl时将这些信息添加到KeyInfo元素中。

KeyInfo元素 XML安全性指定了一个KeyInfo元素，指示用于验证签名或解密某些数据的密钥。例如，它可以包括证书、密钥的名称或密钥协定算法。其使用是可选的：签名者可能不想向文档的
809 访问者透漏任何密钥信息；而在某些情况下，使用XML安全性的应用程序可能有权访问所用的密钥。

规范的XML 某些应用程序可能会做一些对XML文档的实际信息内容没有影响的更改，这是因为有多种不同的方式来表示逻辑上相同的XML文档。例如，属性的顺序可能不同，可能使用不同的字符编码，但信息内容是等价的。规范的XML[www.w3.org X]旨在用于数字签名，而数字签名则用来保证文档的信息内容不被更改。在签名前，将XML元素规范化，并将规范化算法的名称

与签名一起存储起来。这样可以保证在验证签名时使用相同的算法。

规范形式是将XML以比特流的形式进行标准的序列化。它添加了默认属性并去除了多余的模式，并在每个元素中将属性和模式声明以词典顺序排列。它使用了标准的换行形式，字符使用UTF-8编码方案。任意两个等价的XML文档都具有相同的规范化形式。

当对XML文档的一个子集（如一个元素）进行规范化时，规范化形式应包括祖先上下文，即所声明的名字空间和属性的值。因此，在规范的XML与数字签名一起使用时，如果将元素置于不同的上下文中，对元素签名的验证将不会通过。

该算法的一个变种称为互斥的规范XML，它忽略序列化的上下文。如果应用程序想要某个签名元素能在不同的上下文中使用，则可以使用该算法。

以XML形式表示的数字签名的使用 XML形式的数字签名规约[www.w3.org XII]是W3C推荐标准，它定义了新的XML元素类型来保存签名、算法的名称、密钥和对签名信息的引用。该规范中提供的名称是按照XML签名模式定义的，包括元素Signature、SignatureValue、SignedInfo和KeyInfo。图19-16显示了XML签名的实现中必须包含的算法。

算法类型	算法名称	是否必需	参考
消息摘要	SHA-1	必需	7.4.3节
编码	base64	必需	[Freed and Borenstein 1996]
签名	使用SHA-1的DSA	必需	[NIST 1994]
（非对称）	使用SHA-1的RSA	推荐	7.3.2 节
MAC 签名	HMAC-SHA-1	必需	7.4.2节和Krawczyk等 [1997]
（对称）			
规范性	规范XML	必需	“规范的XML”部分

图19-16 XML签名所需的算法

密钥管理服务 XML密钥管理服务的规约[www.w3.org XIII]包含用于分发和注册XML签名使用的公钥的协议。虽然不需要任何公钥基础结构，但该服务仍然可以与现有的公钥基础结构兼容，如X.509证书（见7.4.4节）、SPKI（简单公钥基础结构，见7.4.4节）或PGP密钥标识符（很好的私密性，见7.5.2节）。 810

客户可以使用该服务查找某个人的公钥。例如，如果Alice想给Bob发送一封加密的电子邮件，她可以使用该服务来获得Bob的公钥。另一个例子是，Bob从Alice那里收到一个签名文档，该文档包含Alice的X.509证书，那么Bob将请求密钥信息服务来获取公钥。

XML加密 [www.w3.org XIV]中定义了XML形式的加密标准，它是W3C推荐标准，定义了用XML表示加密数据的方式，也定义了加密和解密的过程。它引入了EncryptedData元素来包含加密数据部分。

图19-17指定了应该包含在XML加密的实现中的加密算法。块密码算法用于加密数据；base64编码在XML中用来表示数字签名和加密数据。密钥传输算法是用于加密以及解密密钥本身的公钥加密算法。

对称密钥包装算法是共享密钥加密算法，用于通过另一个密钥来加密和解密对称密钥。如果密钥包含在KeyInfo元素中，则可以使用该算法。

密钥协定算法允许从一对公钥的计算结果得到一个共享的私钥。若应用程序想不进行任何交换就共享密钥，则可以使用该算法。它不适用于XML安全系统自身。

算法类型	算法名称	是否必需	参考
块密码	TRIPLEDES	必需	7.3.1节
	AES-128, AES-256		
	AES-192	可选	
编码	base64	必需	[Freed and Borenstein 1996]
密钥传输	RSA-v1.5	必需	7.3.2节
	RSA-OAEP		[Kaliski and Staddon1998]
对称密钥包装	TRIPLEDES KeyWrap	必需	[Housley 2002]
(由共享密钥签名)	AES-128 KeyWrap		
	AES-256 KeyWrap		
	AES-192 KeyWrap	可选	
密钥协定	Diffie-Hellman	可选	[Rescorla, 1999]

811 图19-17 加密所需的算法（还需要图19-16中的算法）

19.6 Web服务的协作

SOAP基础结构支持客户和服务器之间的单个请求/应答交互。然而，许多有用的应用程序往往涉及很多请求，必须以特定顺序处理。例如，在订机票时，在进行预定之前必须收集价格和剩余机票的信息。当用户通过浏览器与网页交互时，比如订机票或在拍卖中竞价，浏览器提供的接口根据服务器提供的信息来控制操作执行的顺序。

然而，如果服务是一个负责预定的Web服务，类似于图19-2显示的旅行代理服务，那么在该Web服务与其他执行汽车租赁、酒店预订和机票预定等的服务交互时，需要按照一个合适的描述来工作。图19-18给出了这样一个描述。

1. 客户向旅行代理服务请求有关一组服务的信息，如航班、汽车租赁和酒店预订。
2. 旅行代理服务收集价格和可用信息并将其发送给客户，客户代表用户选择以下一种动作：
   (a) 改进查询，可能涉及更多提供者，从而获得更多信息，然后重复步骤2。
   (b) 做出预定。
   (c) 退出。
3. 客户请求预定，旅行代理服务检查是否可以预订。
4. 要么所有服务都可以预订；
   要么对于不可用的服务；
   要么向返回步骤3的客户提供替代服务。
   要么客户返回步骤1。
5. 交纳定金。
6. 作为确认，给客户一个预定号。
7. 在最后付款前这段时间中，客户可修改或取消预定。

图19-18 旅行代理场景

这些例子阐述了在与其他Web服务进行交互时，需要给作为客户的Web服务提供一种要遵循的协议的描述。但在服务器接收和响应多个客户的请求时，还存在服务器的数据保持一致性的问题。
812 第13章和第14章讨论了事务，并通过一系列银行事务阐述了这个问题。作为一个简单的例子，在两个银行账户之间转账时，一致性要求向一个账户存钱和从另一个账户取钱这两个操作必须都执行。第14章介绍了两阶段提交协议，协同服务器使用该协议来确保事务的一致性。

在某些情况下，原子事务能够满足使用Web服务的应用程序的需求。然而，诸如旅行代理这样的活动需要花费很长时间才能完成，并且由于要在很长时间内锁定资源，因此使用两阶段提交协议来执行这些活动是很不实际的。一种可选的方案是使用更灵活的协议，在该协议中每个参与者在出现时都对持久性状态做出更改。在发生故障的情况下，使用应用程序级协议来撤销这些操作。

在传统的中间件中，基础结构提供了一个简单的请求/应答协议，允许将事务、持久性和安全性等服务作为单独的高级服务来实现，以便在需要时可以使用这些服务。对Web服务来说也是这样，W3C和其他组织已经在定义更高级的服务方面做出了许多。

在Web服务协调的通用模型方面已经取得了一些成果，该模型类似于14.2节讨论的分布式事务模型。在分布式事务模型中有协调者和参与者角色，这些角色能够执行特定的协议，如执行分布式事务。Langworthy[2004]描述了这项称为WS-Coordination的工作。该小组还给出了事务如何在该模型中得到执行的说明。若要透彻研究Web服务协调协议，请参见Alonso等[2004]。

在本节的剩余部分里，我们给出了Web服务编排的思想。考虑这样一个事实，通过工作在同一个任务（如旅行代理情况）中的Web服务对之间的交互，可以描述所有可能的有效路径。如果存在这样一个描述，则可以用它来协调共同任务。它还可以用作服务的新实例（如想加入协作的航班预定服务）要遵循的规范。

W3C使用术语编排来表示基于WSDL的用来定义协调的语言。例如，该语言可以指定参与者之间交换信息所依照的顺序和条件方面的限制。编排用于提供一组交互的全局描述，显示每个参与者的行为，从而达到增强互操作性的目的。

编排的需求 编排用于支持Web服务之间的交互，这些Web服务通常由不同的公司和组织来管理。一个涉及多个Web服务和客户的协作应按照参与者之间的一组可观测的交互来描述。一个描述可以看作参与者之间的契约，该描述有以下用途：

- 为想参与的新服务生成代码概要。
- 作为为新服务生成测试消息的基础。
- 促进对协作达成共识。
- 分析协作，例如识别可能的死锁情况。

813

一组协作的Web服务使用一个通用的编排描述，这样应该能产生具有更好互操作性的更为健壮的服务。另外，将更容易开发和引入新的服务，使得全部服务更为有用。

[www.w3.org XV]上的W3C工作草案文档建议一种编排语言应包含以下特征：

- 编排的层次和递归组成结构。
- 为现有服务和新服务增加的新实例的能力。
- 并发路径、选择路径和重复编排某一部分的能力。
- 可变的超时时间——例如，不同的预定保存时间。
- 异常，比如用来处理乱序到达的消息、用来处理撤销等用户操作。
- 异步交互（回调）。
- 引用传递，如允许汽车租赁公司向银行咨询来检查用户的信用。
- 划分同时发生的不同事务的边界，比如以便进行恢复。
- 包含可供人工阅读的文档的能力。

另一个W3C工作草案文档[www.w3.org XVI]描述了一个基于这些需求的模型。

编排语言 目的是生成一种声明性的、基于XML的语言，用来定义可以使用WSDL定义的编排。[www.w3.org XVII]中报告了编排定义语言方面的早期工作。在此之前，一些公司向W3C提交了一个Web服务编排接口规范[www.w3.org XVIII]。

19.7 实例研究：网格

“网格”指的是一种中间件，它使得文件、计算机、软件、数据和传感器等资源的大规模共享成为可能。这些资源主要是由位于不同组织中的许多用户共享，他们通过共享数据或共享计算能力来协作解决一些需要大量计算机才能解决的问题。这些资源需要得到异构的计算机硬件、操作

系统、编程语言和应用程序的支持。为了确保客户能够获得所需要的资源并且服务能提供这些资源，需要进行适当的管理来协调对这些资源的使用。在某些情况下，需要复杂的安全技术来保证
814 在该类型的环境下正确地使用资源。

19.7.1节将介绍World-Wide Telescope，它是一个数据密集型应用，是网格所解决的问题的一个例子。它阐述了一种典型的共享和用户地理分布的模式，从中可以得到科学应用的特征，这些特征表明了网格的一系列需求（见19.7.2节）。我们利用这些需求来推动体系结构的发展，这种体系结构规定了运行在Web服务之上的网格（见19.7.3节）。最后一节将介绍Globus工具包，它是网格体系结构的一种实现。

19.7.1 World-Wide Telescope——一种网格应用

World-Wide Telescope项目关注天文团体共享的数据资源的部署，在Szalay和Gray[2004]、Szalay和Gray[2001]以及Gray和Szalay [2002]的著作中介绍了这个项目。天文数据由观测档案组成，每一个观测文档包含一段特定时间、一段电磁波频谱（光学的、X射线的、无线电的）和天空的一片区域。这些观测数据是由分布在世界各地的不同设备获得的。

有关天文学家如何共享数据的研究对于得出典型的网格应用的特征是非常有用的，这是因为天文学家可以彼此自由地共享他们的成果，所以可以忽略安全的问题，从而简化了这个问题的讨论。

天文学家需要整合同一天体对象的多个不同时间段和多段频谱的数据来进行研究。能够使用独立的观测数据对研究十分重要。可视化可以使天文学家能够查看数据的2维或3维散点图。

数据收集小组将数据存储在大容量存储设备中（现在是以TB计），这样每个数据收集小组都可以对其进行逻辑管理。用于收集数据的设备服从摩尔定律，因此收集到的数据以指数方式增长。在收集数据时，使用流水线方式分析数据并存储得到的数据供全世界天文学家使用。但在其他研究人员使用数据之前，在某一领域的科学家需要协商一种方法来标记数据。

Szalay和Gray[2004]指出，在过去，科学研究数据由其作者写成论文发布在期刊上，并保存在图书馆中。但是现在，数据的数量过于巨大，出版物无法包含。这种情况不仅在天文学领域出现，粒子物理、基因和生物研究领域也存在这个问题。现在，科学家通常是共同协作，花费5～10年时间做实验，然后将生成的数据发布到基于网络的数据库中。因此，研究该项目的科学家同作者一样成了数据发布人员和管理人员。

这个额外的职责要求任何管理数据库的组织使其他研究人员能访问存储设备。这意味着在原有的数据分析的基础之上要增加了不少花费。要使这种共享成为可能，就需要元数据来描述数据收集时间、天空区域和使用的设备等原始数据。另外，导出数据需要跟描述处理数据的流水线的
815 参数的元数据一起存放。

导出数据的计算需要大量的计算。通常在技术改进时需要重新进行计算。所有这些对于拥有数据的组织而言都是不小的花费。

World-Wide Telescope的目的在于将全世界的天文数据整合到一个巨大的数据库中，这个存储设备包括天文学文献、图像、原始数据、导出数据集和模拟数据。

19.7.2 数据密集型科学应用的特征

在本节中，我们通过对World-Wide Telescope的研究给出类似的科学应用的特征，然后列出支持这些特征的需求。特征如下：

- 数据通过科学设备收集。
- 数据存储于一系列不同站点的存储设备中，这些站点可以位于世界的任何地方。
- 数据由不同组织的科学家小组管理。
- 设备生成的原始数据十分巨大且不断增长（以TB或PB计）。

- 使用计算机程序来分析和总结原始数据，如对表示天体对象的原始数据进行分类、校准和编目。

世界各地的科学家可以通过因特网访问所有的这些存储设备，他们能够获得不同地点的不同设备在不同时间采集的数据。然而，一个在其研究中使用这些数据的科学家仅仅对数据库中数据库中对象的一个子集感兴趣。

考虑到所需的传输时间和本地的磁盘空间，数据库中海量的数据如果不经过处理以抽取感兴趣的对象就传输给用户是不现实的。因此在这种环境下不适合使用ftp或Web访问。应该在收集原始数据并将其存储在数据库的位置对其进行处理。当科学家对某一数据进行查询时，可以对每一个数据库中的数据进行分析。如果需要，则在返回远程查询结果之前生成可视化图形。

数据在不同的地点被处理这一事实已经为我们提供了并行性，从而有效地分割了要处理的巨大任务。

从上面的特征可以获得以下需求：

R1：对资源的远程访问，即远程访问所需的存储设备中的信息。

R2：可以在收集数据时，也可以在响应请求时，在存储和管理数据的站点上处理数据。一个典型的查询可以得到基于不同设备在不同时间记录的某一区域的天空的数据的可视化表现。这将涉及从每个大规模的存储设备中选择少量的数据。 816

R3：数据存储的资源管理器应该能够动态地创建服务实例来处理所需的数据的特定部分，正如在分布式对象模型中，每当需要伺服器来处理服务管理的不同资源时就创建它。

R4：需要使用元数据来描述：

- 存储设备中数据的特征，比如对于天文学，这些特征有：天空区域、数据收集的日期和时间以及使用的设备。
- 管理这些数据的服务的特征，如其花费、地理位置、发布者或负载、可用的空间。

R5：基于上述元数据的目录服务。

R6：考虑到资源通常是由生成数据的组织管理，并且需要定量分配对这些资源的访问，所以需要管理查询、数据传输和提前预订资源的软件。

Web服务提供一种简单的方法允许科学家对远程存储设备中的数据进行操作，从而满足了前两个需求。这需要每个应用程序都提供一个服务描述，该服务描述包括一系列访问其数据的方法。网格中间件处理剩下的需求。

虽然World-Wide Telescope是一个典型的数据密集型应用，但网格还可用于计算密集型应用，如图像分析和19.7.4节中讨论的其他例子。当在网格上部署计算密集型应用时，资源管理将关注于分配计算资源和平衡负载。

最后，许多网格应用程序还需要安全性。例如，用于医学研究和商业应用的网格。即使在数据的私密性不是一个问题，建立数据创建者的身份标识也是十分重要的。

19.7.3 开放的网格服务体系结构

开放的网格服务体系结构（OGSA）是基于网格应用的一个标准[Foster et al. 2002 and 2001]。它提供了一个可以满足以上需求的框架。它基于Web服务，通过特定于应用的网格服务管理资源。19.7.5节讨论的Globus工具包实现了这种体系结构。

图19-19给出了网格体系结构的主要组件，它阐述了应用级网格服务的两个重要方面：

1）它们是实现了标准的网格服务接口以及应用特定接口的Web服务，特别是实现了下列网格服务接口和附加功能：

- 包含有关服务元数据的数据集（称为服务数据）的接口。元数据可以包括终止时间，也可以

817 包括需求R4中提到的任何项，以及诸如最近结果集或平均值的应用程序值。

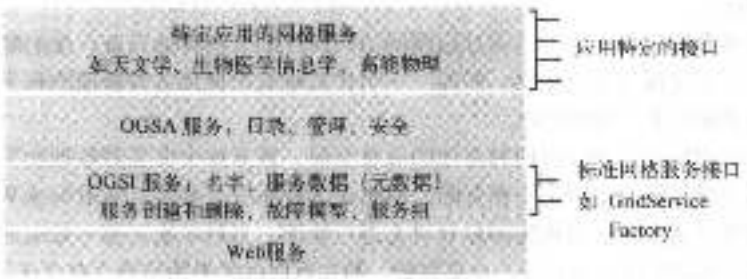


图19-19 开放的网格服务体系结构

- 服务运行的上下文必须提供一个工厂，该工厂能够创建新的服务实例并能够在时间耗尽时终止这些实例，如需求R3所示。工厂依赖于OGSI层的名字工具来管理其创建的服务实例的名称。

2）它们在Web服务之上使用两层标准的网格服务，这两层分别称为OGSI和OGSA。OGSA服务层包括：

- 目录服务：允许客户软件基于用目录服务注册的服务实例中收集的元数据，选择满足需要的服务实例。这解决了需求R5。
- 管理服务：根据服务数据中的信息监控服务，以处理故障情况；通过设置服务数据中的值来控制服务实例的生命期。这解决了需求R6。
- 安全服务：提供认证和数据加密，也提供了单点登录和委托。

开放的网格服务基础设施（OGSI）层包括：

- 服务实例的命名模式的实现。
- 标准服务数据元素的定义，这必须通过每个应用级网格服务实例以及设置和获得服务数据元素值的操作来实现。比如，这些元素包括所支持的接口的名称、用于创建实例的工厂的引用或终止时间。
- 用于创建新服务实例的工厂的接口的定义和用于设置服务实例的终止时间或删除服务实例的操作的定义。

818
- 供所有网格服务使用的故障模型。
- 通知服务：使服务成为有关服务数据的信息发布者，其他服务成为订阅者。
- 服务组：添加或删除成员的操作，供提供服务的协作小组使用。

以上定义和服务在OGSI层的GridService、Factory和其他接口中提供。

开放的网格服务基础设施 我们现在介绍两级命名模式，它将服务实例的高级名称同低级名称联系起来，然后给出OGSI层中其他服务的更详细信息。

两级命名模式：可以动态创建网格服务，为满足区分不同实例的需要，基础设施包括一个命名模式。该模式为每一个实例提供了一个长期的全局唯一标识符，称为网格服务句柄（GSH）。在用以前的状态重启服务的实例时，可以使用相同的GSH。

GSH用一个URI表示，这个URI可以映射到一个叫作网格服务引用（GSR）的短期名称上。GSR指向的是调用的目的地。基础设施包括一个句柄解析服务来执行这个映射。GSR是一个结构依赖于所使用的请求/应答机制的名称。例如，在使用SOAP的情况下，GSR指向一个包含<service>和<binding>元素的WSDL文档。GSR在其指向的服务实例不存在时变为无效。

如果用URN来表示GSH，则可以在不同时刻解析为不同的服务实例，从而允许GSH迁移甚至在不同资源处复制。

服务数据（元数据）：其思想是客户可以请求服务的一个实例来返回有关其当前状态的信息，

例如，服务的能力、剩余空间、负载或当前错误甚至是最近的结果。这要求应用级网格服务的每个实例都必须实现服务数据的存储并实现一组访问服务数据的操作。

GridService接口提供了一组标准的操作，这些操作访问服务支持的接口的名称、服务数据元素的名称、创建该服务的工厂的标识、其GSH、GSR及其终止时间。每个网格服务都必须支持GridService接口，还可以包括用于通知或服务组的OGSI接口。

服务创建和删除：这一部分的基础设施定义了一个标准的Factory接口，该接口指定了用于按需创建短暂服务实例的操作，其中使用了GSH和GSR对形式的名称。任何用于运行应用级网格服务的容器都必须实现Factory接口。

应用程序发送一个典型的请求要求创建一个新的服务实例，并将其服务描述和生命期作为参数。作为响应，工厂创建一个新的服务实例并将其注册在一个句柄解析服务上，最后向客户返回GSH和当前的GSR。

当服务的任务完成或应创建服务的客户的请求时，服务实例将会消亡。每个服务实例都具有一个生命期，以防在消息丢失的情况下实例永远存在下去。但实例的创建者可以发送“keep alive” 819
消息来延长其生命期。在客户失败的情况下，就不会再有“keep alive”消息，服务实例也将最终中止。为提供服务的自动性，一个实例可以改变自己的生命期。当服务实例的生命期到达尽头时，其使用的所有资源将释放。客户可以共享服务实例。

故障模型：基础设施定义了一个通用的方法来报告故障，所有OGSI层服务都使用这个方法，并且推荐应用级网格服务也使用该方法。它定义为一个XML模式，至少需要两个元素来指定源服务和时间戳。它提供可选的元素来以简明的语言描述故障，包括其原因和出错码。留出了空间用于扩展故障报告以包含特定于一个OGSI或应用服务的信息。

WSDL接口扩展：由于标准的WSDL不包括用于定义服务数据的机制，OGSI提供了对WSDL的扩展，使用服务描述中的某个接口将服务数据元素的名称和类型关联起来。比如，数据元素可以保存服务的能力、空闲空间、负载或当前错误指示。

OGSA服务 高层服务构建在OGSI服务之上。根据Foster等[2004]的研究，不同的开发人员将提供多种不同的OGSA服务来满足应用级服务的特定需求。他们列出了那些广泛使用以至于可以包含在任何网格系统中的组件，如目录服务、管理和监控以及安全服务。这些服务的标准接口的可用性是确保不同实现可以互相协作的基础。在19.7.5节中，我们将在Globus工具包中讨论目录和安全服务。

管理服务：Grid中的管理关注的是任何可以共享和使用的资源。这相当于服务管理，服务管理关注的是安排要使用的服务并监视服务状态。它解决的问题包括任务提交、提供某种质量的服务以及资源的提前预定的协定。这些可以通过*服务级别协定*（SLA）[Hauch and Reiser 2000]来提供，SLA为资源的客户端提供服务类型的保证，允许资源的所有者控制如何使用资源以及向客户公开多少信息。

有三种类型的SLA：任务级，用于对某项活动的性能达成共识；资源级，用于对消费资源的权力达成共识，如通过提前预定；绑定，用于对任务中资源的使用达成共识。

19.7.4 一些网格应用的例子

图19-20给出了使用网格技术的一些例子。前三个例子具有与World-Wide Telescope类似的特征：数据由科学设备收集并将之存储在收集站点。在例1中，通过飞行器引擎上的传感器收集震动 820
数据。在例2中，从测试结构中收集数据，测试结构很容易剧烈震动，可以模拟地震。在例3中，使用MRI或CT等设备来收集大脑图像。在所有的情况中，随着测量次数的增加，原始数据的量随时间不断增长。另外，随着分析的执行，被处理的数据也在不断增加。

项目描述	参考
1. 飞行器引擎维护：使用故障历史和传感器做出预测诊断	www.cs.york.ac.uk/dame
2. 远程呈现：使用仿真和测试场所预测地震对建筑物的影响	www.neesgrid.org
3. 生物医学信息学网络：使研究人员能够获得实验及其结果的图形化表示	nbcr.sdsc.edu
4. 对CERN上的CMS高能粒子探测器中的数据进行分析，这些数据是全世界物理学家用超过15年的时间收集到的	www.uscms.org
5. 通过使用空闲的桌面计算机执行并行计算，测试候选药物分子对蛋白质活动的影响	[Taufer et al. 2003] [Chien 2004]
6. 通过利用Web服务器集群上的空闲能力，使用Sun Grid Engine增强航空图像	www.globexplorer.com
7. 蝴蝶Grid通过Globus工具包支持因特网上巨大数目的玩家之间的多玩家游戏	www.butterfly.net
8. Access Grid支持小组协作，如通过提供共享工作空间	www.accessgrid.org

图19-20 Foster和Kesselman[2004]中介绍的网格项目节选

在这三个例子中，数据由属于不同组织的科学家或工程师小组管理。对原始数据的处理和模拟都在本地完成，并可以被全世界的合作者使用。

这些例子证实，World-Wide Telescope是一个典型的数据密集型应用。但是，网格应用还可以用于计算密集型应用程序，如：

- 例4指的是位于CERN的新探测器，它将于2007年投入运行。在此之前，许多物理学家团队正在对探测器所得到的预期结果进行模拟。这是一个计算密集型应用，由多台协作的计算机执行。
- 例5关于虚拟筛选——测试数以百万计的药物分子，查看是否能阻止许多蛋白质中的每一个蛋白质的活动。可以使用一组桌面计算机的空闲计算能力来运行网格软件。每台计算机测试某个药物分子对某个蛋白质的作用，从而并行执行该任务。它使用空闲的桌面计算机的闲置能力。
- 例6是关于图像分析的一个例子。GlobeExplorer公司提供高质量的卫星图像和航空照片，这些图片是从无数原始照片中得到的，每张原始照片都需要增强。它使用了Web服务器集群的空闲能力。

821

例5和例6的一个共同特征是使用空闲计算能力以及对要执行的任务进行分割。它们可以与SETI@home项目（见10.1节）相比，该项目的任务是从射电望远镜数据中搜寻外星文明。通过使用最终用户的计算机的空闲计算能力，SETI@home使用对等软件来解决计算密集型问题。巨大的数据库被分割以便并行执行任务。

之所以给出例7和例8，是因为它们阐述了网格在科学和工程领域之外的两种不同用途。它们都需要管理分布式状态，在第一种情况下是游戏的状态，第二种情况中是共享的工作空间。

19.7.5 Globus工具包

Globus项目开始于1994年，其目的是提供一种集成和标准化科学应用所需功能的软件。这些功能包括目录服务、安全性和资源管理。第一个Globus工具包在1997年出现。工具包的第2版（称为GT2）中开始出现OGSA，在Foster和Kesselman[2004]中介绍了这一点。

第3版出现于2002年，称之为GT3，它基于OGSA并构建于Web服务之上。GT3是由Globus联盟（www.globus.org）等开发的，是一种开源软件。

Sandholm和Gawor[2003]介绍了GT3的内核。它包括OGSI层中的所有接口，并以Java类方式实现了诸如GridService和Factory等接口，并且可以与JAX-RPC兼容（见19.2.3节）。将此功能添加到现有的Web服务中的最简单的方法是使服务的类成为OGSI类的子类。实际上，可以对应用级的网格服务进行配置，使其包含所需的网格服务功能。例如，可以按需增加通知服务或服务组操作。

网格服务实例和工厂是在称为网格服务容器的运行时环境中部署的。网格服务容器在某些方面类似于CORBA POA（见20.2.2节），因为它既可以执行调度，又可以处理：

- 具有全局名称的服务实例的动态创建和管理。
- 通过GridService接口中的FindServiceData操作或通过通知来访问服务实例的状态。
- 安全性，包括证书的授权、消息的签名、加密和验证。

服务实例可以在与其工厂相同的容器内创建，或者可以在其他地方部署。比如，一个容器可能基于一个servlet容器（见19.2.3节）。在启动容器时，服务实例并不立即转为活跃状态，直到第一次被使用时才变为活跃状态。容器可以使空闲的服务实例（例如通知或服务小组使用的实例）变得不活跃。 822

安全服务 GT3中的安全服务提供了对SOAP消息的保护。它基于WS-Security[Kaler 2002]、XML签名和XML加密（见19.5节）。认证使用X.509证书作为凭证，这些证书由一个可信赖的证书颁发机构以通常方式提供。它使用X.509的扩展来提供代理证书，这些代理证书可以被代表用户的服务使用。

目录服务 GT3不使用UDDI，这是因为其数据结构以及搜索标准包含不合适的信息，比如，有关协议的技术细节（见图19-15），而网格服务的客户需要找到服务实例的某种特征。例如，管理客户可能对服务负载感兴趣，天文学家可能对数据收集的时间、地点和方式等信息感兴趣。

GT3提供了索引服务作为替代，索引服务用于查找匹配一组特定需求的服务实例。索引服务基于向其注册的一组服务实例的服务数据收集信息。它可以使用FindServiceData操作来收集信息，但对于可变的服务数据，则更适合使用通知接口，以便在数据值改变时通知索引服务。

索引服务可以响应查询，它通过执行算法来决定最适合某个客户的服务实例。例如，该过程可以基于服务实例使用的处理器类型、速度、支持的客户或使用该服务实例的开销。除了响应查询外，索引服务还支持通知接口。

可以以多种方式将几个索引服务联合起来，以提供大型团体使用的信息服务。

索引服务的实现需要包括容错措施以处理各种情况。比如，一个服务实例在没有通知索引服务的情况下停止运行，则索引服务在其索引中保存的将是过期数据。

管理和可靠的文件传输服务 GT3中的管理服务关注的是监视和管理容器以及容器中的服务实例，例如，监视当前的状态或负载以及激活与使实例变得不活跃。

考虑到网格服务之间要传输大量数据，GT3还提供了一个可靠文件传输服务。

未来与Web服务的关系 关于在WS-Resource框架之下将OGSI特征集成到Web服务的可能性进行了一些讨论[Globus 2004]。Globus联盟指出，应该具有在比网格应用更广泛的环境中创建服务实例和使用服务状态的能力。然而，这种分布式对象方法对于许多现在通常作为Web服务运行的松耦合应用来说不太合适。 823

19.8 小结

本章说明了Web服务的产生源于为不同组织之间的交互提供基础设施的需要。该基础设施通常使用HTTP协议通过因特网上在客户和服务器之间传输消息，它使用URI来指向资源，使用文本格式的XML表示数据和编码数据。

两个因素导致了Web服务的出现。一个因素是：为了允许客户程序而不是浏览器以一种更丰富的交互性访问一个站点上的资源，需要将服务接口添加到Web服务器。另一个因素是希望基于现有的协议在因特网上提供一种类似RPC的结构。由此产生的Web服务提供了带有一组可以远程调用的操作的接口。与其他形式的服务类似，Web服务可以是另一个Web服务的客户，并允许一个Web服务集成或组合一系列其他Web服务。

SOAP是Web服务和其客户通常使用的通信协议，它可以用于在客户和服务器之间传送请求消息及应答，这个过程既可以通过文档的异步交换方式，也可以通过基于一对异步消息交换的请求/应答协议来实现。在这两种情况中，请求或应答消息都包含在称为信封的XML格式的文档中。虽然可以使用其他协议，但SOAP信封通常通过异步HTTP协议传送。

XML和SOAP处理程序可用于所有广泛使用的编程语言和操作系统。这使得可以在任何地方部署Web服务及其客户。由于Web服务既不绑定到任何编程语言也不支持分布式对象模型，这使得这种交互形式成为可能。

在传统的中间件服务中，接口定义为客户提供了服务的详细信息。然而，在Web服务中使用了服务描述。服务描述不仅描述了服务的接口，还指定了使用的通信协议（如SOAP）和服务的URI。接口既可以描述成一组操作，也可以描述成在客户和服务器之间交换的一组消息。

需要交换某个文档的多个用户都要在该文档上执行不同的任务，XML安全性旨在为该文档的内容提供必要的保护。不同的用户可以访问文档的不同部分，某些用户可以添加或更改文档内容而有些用户则只能阅读文档。为使该文档在以后的使用中更加灵活，在文档内定义了安全属性。它通过使用一种自描述的格式——XML来完成。XML元素用于指定经过加密或签名的文档部分，也指定了所使用算法的详细信息以及用于帮助查找密钥的信息。

为了支持分布在世界不同地方的组织中的科学家或工程师之间的协作，出现了运行在Web服务之上的网格中间件。这些用户的工作通常基于使用不同站点的工具收集的并在本地处理的原始
824 数据。Web服务接口使得用户可以远程访问这些数据。然而，代表远程客户开始处理数据的需求导致了需要按需创建服务实例并管理其生命期。网格体系结构向Web服务增添了工厂以及用于引用服务实例的两级命名模式。另外，它支持描述服务实例特征的元数据。它还详叙了目录、管理和安全服务。Globus工具包是该体系结构的一个实现，它已用于多种数据密集型和计算密集型应用程序。

练习

19.1 比较4.4节介绍的请求/应答协议和使用SOAP的客户-服务器通信的实现。为什么SOAP使用的异步消息更适合于因特网上的应用，请给出两个原因。SOAP通过使用HTTP在什么范围内减少了两种方法之间的差异？ （第790页）

19.2 比较Web服务使用的URL结构与4.3.4节介绍的远程对象引用的URL结构。说明在每一种情况下它们如何处理客户请求。 （第794页）

19.3 使用SOAP以及图19-4和图19-5中给出的XML的图示版本重做练习5.3。 （第791页）

19.4 列出WSDL服务描述的五个主要元素。请说明在练习5.1定义的Election服务中，请求和应答消息所使用的信息类型——这些都必须包括在目标名字空间中吗？对于vote操作，画出类似于图19-11和图19-13的图。 （第803页）

19.5 在Election服务的例子中，解释练习19.4中定义的WSDL部分被称作“抽象”的原因。为使服务描述成为完全“具体”的，需要向其中增添什么？ （第800页）

19.6 为Election服务定义一个Java接口，使其适合用作Web服务。说明你定义的接口为什么是合适的。解释如何生成该服务的WSDL文档并使客户可以使用该文档。 （第796页）

19.7 描述Election服务的Java客户代理的内容。解释如何从静态代理获得正确的编码和解码方法。 （第797页）

19.8 解释servlet容器在部署Web服务和处理客户请求时的作用。 （第796页）

19.9 在图19-8和图19-9给出的Java例子中，虽然Web服务不支持分布式对象，但客户和服务器都
825 处理对象。为什么会出现这种情况？Java Web服务接口的限制是什么？ （第796页）

19.10 概述UDDI中使用的复制模式。假设使用向量时间戳来支持该模式，定义注册处交换数据所用的一对操作。 (第807页)

19.11 考虑到询问类型，解释为什么既可以把UDDI看作名字服务又可以看作目录服务？UDDI中的第二个“D”指的是“发现”，UDDI真的是一个发现服务吗？ (第9章及第805页)

19.12 概述TLS和XML安全性之间的主要区别。解释为什么在这些区别中，XML特别适合其扮演的角色？ (第807页)

19.13 在可以预测最终接收者之前，受XML安全性保护的文档可以被签名或加密。采取什么措施可以确保后面的接收者能够访问前面接收者使用的算法？ (第807页)

19.14 解释规范的XML和数字签名之间的相关性。规范的形式中可以包括什么样的上下文信息？给出一个违背安全性的例子，其中在规范的形式中省略了上下文。 (第810页)

19.15 为协调Web服务的操作可以执行一个协作协议。分别概述（1）集中式和（2）分布式协作协议的体系结构。在每种情况下，说明在一对Web服务之间建立协作所需的交互。 (第812页)

19.16 Web服务在多大程度上满足支持网格的需求？概述OGSI服务如何添加Web服务没有提供的功能。 (第822页)

19.17 概述Globus工具包中的索引服务的功能。说明UDDI为什么不适合用作网格的目录服务。 (第822页)

第20章 CORBA实例研究

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，或简称为ORB。

CORBA规约已经由对象管理组（OMG）的成员共同提出。根据规约实现了许多不同的ORB，并支持包括Java和C++在内的各种各样的编程语言。

CORBA服务提供能在许多应用领域采用的通用机制。它们包括名字服务、事件和通知服务、
827 安全服务、事务和并发服务以及交易服务。

20.1 简介

为了从软件开发所用的面向对象程序设计中获益，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）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。

构建开放式系统的一个重要动机是：允许分布式对象以任意编程语言来实现，并且能够相互通信。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。ORB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。

1991年，一个对象请求代理体系结构的规约，也就是大家熟知的CORBA（公共对象请求代理体系结构），得到许多公司的认同和接受。紧接着，1996年又推出了CORBA 2.0规约，CORBA 2.0规约定义了一些标准，目的是使不同开发者的实现能相互通信。这些标准称为*通用ORB间互操作协议*（GIOP）。根据设计，GIOP能够在任何具有连接的传输层上实现。在因特网上实现的GIOP使用的是TCP协议，故称为*因特网ORB间互操作协议*（或IIOP）[OMG 2004a]，而CORBA 3在1999年下半年第一次出现，并且最近又添加了一个组件模型。

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IDL，它将在20.2节的开始部分加以介绍，而更完整的描述在20.2.3节给出。
- 体系结构，详见20.2.2节的讨论。
- GIOP定义了一种外部数据表示，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。除了请求和应答消息之外，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作为GIOP的一个实现定义了远程对象引用的标准格式，详见20.2.4节的描述。

CORBA体系结构还考虑了CORBA服务，即一系列可以用于分布式应用的通用服务，这些都在20.3节详细介绍，20.3节还详细地阐述了名字服务、事件服务、通知服务和安全服务。与CORBA有
828 关的论文的汇总详见CACM特刊[Seetharamanan 1998]。

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。

20.2 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。他们必须首先了解以下几个新概念：

- 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节中介绍过的共享白板程序为例，介绍如何写一个IDL规范，以及如何创建服务器和客户程序。

CORBA的对象模型 CORBA对象模型与5.2节介绍过的一个对象模型相似，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。这里，术语CORBA对象指的是远程对象。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来实现，例如没有类概念的语言。既然在不同的实现语言中有不同的类的概念，有些甚至没有类概念，故而在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，各种类型和任意复杂度的数据结构都可以作为参数传递。

CORBA IDL 一个CORBA IDL接口指定一个客户能够请求的名字和一系列方法。图20-1显示了两个分别称为Shape（第3行）和ShapeList（第5行）的接口，它们是图5-12中定义的接口的IDL版本。在它们的前面是两个struct的定义，这两个struct在方法定义中用作参数的类型。特别要注意的是，GraphicalObject也被定义为一个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域，可包含各种类型的值，比如一个对象的实例变量，但是它没有方法。关于IDL更多的介绍见20.2.3节。

```
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20-1 Shape和ShapeList的IDL接口

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是输入类型或者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20-1的第7行中，newShape的参数是一个in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参数——如果没有out参数，可以将返回值指明为void。

参数可以是简单类型中的任何一种，例如long和boolean型，或者是一种构造类型，比如struct或者array。简单类型和构造类型在20.2.3节有更详细的描述。我们的例子在第1行和第2行给出了两个struct的定义。序列和数组用typedef定义，如第4行所示，它表示一个长度为100的Shape类型的元素序列。参数传递的语义如下：

传输CORBA对象：如果某个参数的类型由一个IDL接口的名字指定，例如第7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。

传输CORBA简单类型和构造类型：简单类型和构造类型的参数是拷贝之后以值传递的。到传递到达的时候，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行）就在服务器上创建了这种结构的一个新拷贝。

829
~
830

在allShapes方法（第8行）中结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是其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）的公共超类型。

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异常。为了说明这一点，我们在服务器中将Shapes列表定义为一个定长序列（第4行），还定义了FullException（第6行）异常，如果客户试图在序列全满的情况下添加一个形状的话，newShape方法（第7行）将抛出该异常。

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有或许语义。客户不阻塞oneway请求，它只能用于没有返回结果的方法。有关oneway请求的一个例子，参见20.2.1节末尾关于回调的例子。

CORBA名字服务 CORBA名字服务将在20.3.1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，以便服务器能用名字注册CORBA对象的远程对象引用；还包括为客户提供resolve操作，以便客户能按名字查找它们。名字以层次方式构造，并且一条路径中的每个名字都放在一个称为NameComponent的结构之中。这使得简单例子中的访问看起来相当复杂。

CORBA伪对象 CORBA的实现提供了程序员需要使用的一些ORB的功能的接口。特别是它们提供了图20-6所示的两个组件：ORB核心和对象适配器的接口。这两个组件的作用将在20.2.2节中介绍。

代表那些组件的对象称为伪对象，因为它们不能像通常的CORBA对象那样使用。例如，它们不能作为RMI的参数传递。它们拥有IDL接口，并被实现为库。与我们的简单例子（使用Java 21.4版本）相关的是：

- 对象适配器，它从CORBA 2.2开始是可以移动的，因此也称为可移动对象适配器（POA）。它的接口包括：一个激活POAmanager的方法和一个用于注册CORBA对象的servant_to_reference方法。
- ORB的接口包括：init方法，该方法用于初始化ORB；resolve_initial_reference方法，它用于查询服务（例如名字服务）和根POA；以及其他用于远程对象引用与字符串之间转换的方法。

20.2.1 CORBA客户和服务器实例

本节概述采用图20-1中IDL定义的Shape和ShapeList接口来说明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，这种方法对于其他语言来说也是相似的。接口编译器idlj可以应用到CORBA接口中，用于创建如下项目： 831

- 等价的Java接口——每一个IDL接口对应于两个Java接口。第一个Java接口名称以Operations结尾，用于定义IDL接口中的操作。第二个Java接口名称与IDL接口名称相同，用于实现第一个Java接口中的操作，以及适合CORBA对象的接口中的操作。例如，IDL接口ShapeList产生两个Java接口，分别为ShapeListOperations和ShapeList（如图20-2所示）。

```
public interface ShapeListOperations {
  Shape newShape ( GraphicalObject g ) throws ShapeListPackage.FullException ;
  Shape[] allShapes ( ) ;
  int getVersion ( ) ;
}
```

```
public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
    org.omg.CORBA.portable.IDLEntity{}
```

图20-2 根据CORBA接口ShapeList由idlj生成的Java接口

- 每个idl接口的服务器骨架。骨架类的名字以POA结束，例如ShapeListPOA。
- 每一个IDL接口对应一个代理类或者客户存根。这些类的名字以Stub结尾，例如_ShapeListStub。
- IDL接口中定义的每个struct对应于一个Java类。在我们的例子中，创建了Rectangle和GraphicalObject类。其中每个类包含一个实例变量（对应于struct中的每个域）和一对构造函数的声明，但是没有其他方法。
- 每种类型都有一个定义在IDL接口中称为helper和holder的类。helper类包括narrow方法，它用于将一个给定的对象引用强制转换到它所属于的处于更低类层次的类。例如，ShapeHelper中的narrow方法强制转换到类Shape。holder类处理out和inout参数，它不能直接映射到Java。参见练习20.9，该练习是使用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。对于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。

当一个服务器创建一个伺服类的实例时，它必须用POA注册，POA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CORBA对象就不能接收远程调用。仔细研究了第5章的读者可能会认识到，通过POA注册的对象会被记录在类似于远程对象表的CORBA中的等价位置。

在我们的例子中，服务器包括Shape和ShapeList接口的实现，它的形式是两个伺服类和一个在其main方法中包含初始化部分（见5.2.5节）的服务器类。 832

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接口中定义的方法基调来实现了IDL接口中的方法。尽管也可以选择其他的名字，但还是将实现ShapeList接口的伺服类命名为ShapeListServant。该类的要点显示在图20-3中。考虑第1行中的方法newShape，它是一个工厂方法，因为它创建了Shape对象。为了使Shape对象成为一个CORBA对象，通过POA的servant_to_reference方法注册，如第2行所示。这个方法需要引用根POA，而这个根POA是在创建伺服器时通过构造器传入的。关于这个例子的IDL接口和客户、服务器类的完整版本参见www.cdk4.net / corba。

服务器：在服务器类ShapeListServer中的main方法如图20-4所示。它首先创建和初始化ORB（第1行），这个过程需要引用根POA，并且激活POAManager（第2、3行）。然后，它创建

ShapeListServant的一个实例，它是一个Java对象（第4行），同时给根POA传递一个引用。其次通过在POA中注册它而使之成为一个CORBA对象（第5行）。此后，它通过名字服务注册服务器。最后它等待客户请求的到来（第10行）。

```
import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
  private POA theRootpoa;
  private Shape theList[];
  private int version;
   private static int n=0;
  public ShapeListServant(POA rootpoa) {
      theRootpoa= rootpoa;
      //初始化其他实例变量
  }
  public Shape newShape(GraphicalObject g) throws ShapeListPackage FullException { 1
     version++;
     Shape s = null;
     ShapeServant shapeRef=new ShapeServant(g,version);
     try {
         org.omg.CORBA.Object ref=theRoopoa.servant_to_reference(shapeRef); 2
         s = ShapeHelper.narrow(ref);
     }catch(Exception e){}
      if(n >=100) throw new ShapeListPackage.FullException();
     theList[n++] = s;
     return s;
  }
  public  Shape[] allShapes() { ... }
  public int getVersion() {...}
}
```

图20-3 CORBA接口ShapeList的Java服务器程序中的ShapeListServant类

服务器使用名字服务先获得一个根名字上下文（第6行），然后创建一个NameComponent（第7行），定义一条路径（第8行），最后使用rebind方法（第9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20-5的第2行中的resolve方法。

客户程序 图20-5给出了一个客户程序的例子。它创建和初始化ORB（第1行），然后联系名字服务，利用它的resolve方法获得一个远程ShapeList对象的引用（第2行）。在它调用allShapes方法（第3行）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getAllState方法（第4行），作为返回序列中的第一个远程对象引用的参数；结果作为GraphicalObject类的一个实例提供。

getAllState方法看起来与我们最初的声明，即在CORBA中对象不能通过值传输似乎是矛盾的，因为客户和服务器都处理了GraphicalObject类的实例。然而，实际上并不是矛盾的：CORBA对象返回了一个结构，使用不同的语言的客户对它就有不同的看法。例如，在C++语言中，客户会将它看作一种结构，即使在Java中，已创建的GraphicalObject类也更像一种结构，因为它没有任何方法。

客户程序应该总是捕获CORBA的SystemExceptions异常，它报告因分布性而引起的各种错误（见第5行）。客户程序也应该捕获IDL接口中定义的异常，例如由newShape方法抛出的FullException异常。

这个例子阐述了narrow操作的使用：名字服务的resolve操作返回一个Object类型的值，这个类型被狭义化了，以适合所要求的特定类型——ShapeList。

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class ShapeListServer {
    public static void main(String args[]) {
      try{
        ORB orb = ORB.init (args, null ) ;                                          1
        POArootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));   2
        rootpoa.the_POAManager().activate();                                        3
        ShapeListServant SLSRef=new ShapeListServant(rootpoa);                      4
        org.omg.CORBA.Object ref=rootpoa.servant_to_reference(SLSRef);              5
        ShapeList SLRef=ShapeListHelper.narrow(ref);
        org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);                   6
        NameComponent nc = new NameComponent("ShapeList","");                       7
        NameComponent path[] = {nc};                                                8
        ncRef.rebind(path, SLRef);                                                  9
        orb.run();                                                                  10
      } catch (Exception e) {...}
    }
}
```

图20-4 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20-5 CORBA接口Shape和ShapeList的Java客户程序

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给客户发送一个版本号。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诉服务器。为此，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 ) ;
```

833 ~ 835 在客户获得了ShapeList对象的一个引用并创建了WhiteboardCallback的一个实例之后，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使版本号增加的时候通知这些感兴趣的客户。callback方法被声明为oneway类型，以便服务器能使用异步调用，从而避免因为通知每个客户而带来延迟。

20.2.2 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。在这种情况下，客户和服务器都能以各种编程语言实现。CORBA体系结构的主要组件如图20-6所示。

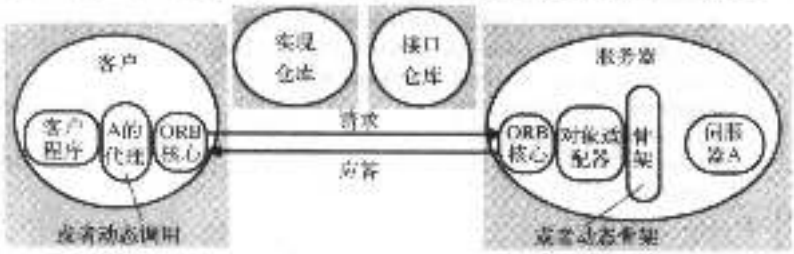


图20-6 CORBA体系结构的主要组件

将该图与图5-7对比，可以发现，CORBA体系结构包含了三种额外的组件：对象适配器、实现仓库和接口仓库。

CORBA提供动态调用与静态调用。在编译时如果知道CORBA对象的远程接口，就使用静态调用，让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核心的作用类似于图5-7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。

836 **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。它也包括图5-7中显示的远程引用模块和分发器模块的功能。对象适配器具有如下的任务：

- 它为CORBA对象创建远程对象引用（见20.2.4节）。
- 它通过适当的伺服器骨架调度每个RMI。
- 它激活或休眠伺服器。

一个对象适配器给每个CORBA对象指定一个唯一的对象名，这也是其远程对象引用的一部分。每次激活某个对象都使用相同的名字。对象名可以由应用程序来指定，也可以由对象适配器来创建。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，用于将CORBA对象名映射到它们的伺服器。

每个对象适配器有自己的名字，这也是它管理的所有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来指定，也可以自动地创建。

可移动适配器 CORBA 2.2标准中将对象适配器称为可移动对象适配器。之所以被称为是可移

动的，是因为它允许应用和伺服器在由不同开发者创建的ORB上运行[Vinoski 1998]。这种可移动性是通过骨架类的标准化以及POA与伺服器之间的交互实现的。

POA支持的CORBA对象按生存期的不同分为两类：

- 一类对象的生命周期严格限制在实例化了它们的伺服器的进程中。
- 另一类对象的生命周期可以跨越多个进程的伺服器实例化。

前一类对象拥有暂时性的对象引用，后一类对象拥有持久性的对象引用（见20.2.4节）。

POA允许透明地实例化CORBA对象。另外，POA将CORBA对象的生成与实现这些对象的伺服器的生成分开。对于那些拥有大量CORBA对象的服务器端应用（例如数据库）可以在对象被访问的时候再按需生成伺服器。在这种情况下，对象名称可以作为数据库关键字，作为一种选择，它们可以使用一个伺服器来支持所有对象。

另外，POA策略是可以指定的。例如，可以定义是否为每一个调用提供一个独立的线程，一个对象引用是持久性的还是暂时性的，或者是否为每一个CORBA对象提供一个独立的伺服器。默认情况下，一个伺服器可以支持注册到POA上的所有CORBA对象。

骨架 骨架类由IDL编译器用服务器所用的语言创建。与以前类似，远程方法调用通过某个伺服器的适当骨架来调度，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 由IDL编译器用客户所用的语言来编写。IDL编译器根据IDL接口为客户语言创建代理类（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，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。 837

实现仓库 实现仓库负责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。用对象适配器名字来引用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。当安装服务器程序的时候，对象实现和对象适配器名通常注册在实现仓库中。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对象都需要在要求的时候激活。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。

实现仓库一般允许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访问控制信息。为了提供可用性或者容错性，信息有可能会在实现仓库中复制。

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，接口仓库提拱了一种CORBA对象的反射机制。假设一个客户程序收到一个新的CORBA对象的远程引用，再假设客户还没有代理，那么它会向接口仓库询问该对象的方法和它们需要的每一个参数的类型。

当一个IDL编译器处理一个接口时，它为每个遇到的IDL类型指定一个类型标识符。对于注册的每一个接口，接口仓库提供一个该接口到它的类型标识符的映射。因此，一个接口的类型标识符有时也被称为仓库ID。因为这个类型标识符可以作为注册在接口仓库中的IDL接口的关键字。

每一个CORBA远程对象引用都包含一个“槽”，它包含接口的类型标识符，并且允许支持它的客户通过接口仓库来查询接口的类型。使用带有客户代理和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 正如在5.5节中所述，在某些应用程序中，有必要构造一个客户程序而并不需

要知道它将来可能使用的代理类。例如，一个对象浏览器可能需要显示分布式系统中各种服务器
838 上可用的CORBA对象的信息。让这样一个程序包含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不允许像在Java RMI中那样，在运行时下载代理类。CORBA的方法是使用动态调用接口。

动态调用接口允许客户动态地调用远程CORBA对象，它在不便使用代理的时候使用。客户能从接口仓库获取关于给定CORBA对象中可用方法的必要信息。客户可以利用这些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 正如5.5节所述，有时一个服务器有必要添加一个在它编译时接口还不确定的CORBA对象。如果服务器使用动态骨架，则它可以接受对那些没有骨架的CORBA对象的调用。当动态骨架接收调用的时候，它检查请求的内容以查找目标对象、待调用的方法和参数，然后它就调用这个目标对象。

遗留代码 术语遗留代码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思想设计的。遗留代码可以通过为它定义一个IDL接口并提供合适的对象适配器和必要骨架的一个实现的方式，从而整合到CORBA对象中。

20.2.3 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、属性和方法基调的机制。除了模块之外，我们已经在图5-2和图20-1中给出了上述情况的相关的例子。IDL具有与C++相同的词汇规则，而且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和raise。它也支持标准的C++预处理机制。可考虑图20-7中的用typedef定义All类型。IDL的语法是ANSI C++的一个子集，并且有支持方法标记符的构造函数。我们在这里只给出了IDL的一个简要描述。Baker [1997]、Henning和Vinoski [1999]给出了IDL的概述和许多例子。完整的规范可参见OMG站点[OMG 2002a]。

```
module Whiteboard {
    struct Rectangle {
    ... };
    struct GraphicalObject {
    ... };
    interface Shape {
    ... };
    typedef sequence < Shape , 100 > All ;
    interface ShapeList {
    ... };
};
```

图20-7 IDL模块Whiteboard

IDL模块 模块构造支持将接口和其他IDL类型定义分组为逻辑单元。一个模块定义了一个名字域，它可以避免在一个模块内定义的名字与在模块外定义的名字发生冲突。例如，接口Shape和ShapeList的定义属于一个称为Whiteboard的模块，参见图20-7。

IDL接口 正如我们已看到的，一个IDL接口描述了实现该接口的CORBA对象中可用的所有方法。CORBA对象的客户可能就是根据其IDL接口的知识进行开发的。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，并且通常依赖于的一系列其他类型用于定义该接口。例如，图5-2中的PersonList接口定义了一个属性、三种方法并且依赖于类型Person。

IDL方法 方法基调的通常格式是：

[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基调：

void getPerson (in string name , out Person p) ;

正如20.2节解释的，参数被标记为in、out或者inout，其中in参数的值从客户传输到被调用的CORBA对象，而out参数从被调用的CORBA对象传递到客户。inout类型的参数极少使用，该类型表示参数值可以双向地传输。如果没有返回值，可以将返回类型声明为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。另外，oneway调用或者被执行一次，或者根本不会被执行，即采用的是或许调用语义。我们在20.2.1节看到过下面的例子：

oneway void callback (in int version) ;

在该例子中，服务器在每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给客户带来问题，因为该调用只指出最新的版本号，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，可以抛出这些异常来终止方法的执行。例如，考虑图20-1中的例子：

exception FullException { } ;

Shape newShape (in GraphicalObject g) raises (FullException) ;

839 ~ 840

newShape方法带有raises表达式，它可以抛出一个称为FullException的异常，该异常定义在ShapeList接口中。在我们的例子中，异常不包括变量。但是，异常也可以定义为包含变量，例如：

exception FullException { GraphicalObject g } ;

当抛出包含变量的异常时，服务器可以使用这些变量，将与异常的上下文返回给客户。

CORBA也能创建与服务器问题相关的系统异常，例如由于它们太忙或者无效了而无法激活，还能创建与通信和客户问题相关的系统异常。客户程序应该处理用户定义的异常和系统异常。可选的context表达式用于提供从字符串名到字符串值的映射。参见Baker[1997]对context的解释。

IDL类型 IDL支持15种简单类型，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（它能表示任何一种简单类型或者构造类型）。大多数简单类型的常数和常数字符串都可以使用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，那么对应的参数可以指任何CORBA对象。

IDL的构造类型在图20-8中给出，这些类型在作为参数或结果时都是以值传递的。所有用作参数的序列和数组必须以typedef定义。简单或者构造数据类型都不能包含引用。

类 型	举 例	使 用
序列	typedef sequence<Shape,100> All; typedef sequence<Shape> All 有界和无界的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 ; 有界和无界的字符序列	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在参数和结果中以值传递

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值集合中

联合	union Exp switch (Rand) {
	case Exp: string vote ;
	case Number: long n ;
	case Name: string s ;
	};
		可区分的IDL联合允许一个给定的类型集合作为一个参数传输。头部由一个enum参数化，该enum指定正在使用哪种类型

图20-8 IDL构造类型

属性 除了方法，IDL接口还可以有属性。属性类似于Java中的公用类域。属性可以在需要时定义为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，一个方法获取属性的值，另一个方法设置属性的值。readonly属性只能提供getter方法。例如，定义在图5-2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、常数、异常、方法和属性。一个被扩展的接口可以重新定义类型、常数和异常，但是不允许重新定义方法。扩展类型的值作为父类参数或者结果的值是合法的。例如，类型B作为类型A的参数或者结果的值是合法的。

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的所有组件（不包括它重新定义的组件）以及它扩展定义的部分。

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具有相同名字的类型、常数或者异常。例如，假设B和C都定义了一种称为Q的类型，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不允许一个接口从两个不同接口继承带有相同名字的属性或者方法。

所有IDL接口都继承自Object类型，它意味着所有的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。名字服务中的bind和resolve操作就是这样的例子。

841 / 842

IIDL类型标识符 20.2.2节提到，由IDL编译器为IDL接口中的每种类型创建唯一的类型标识符。例如，Shape接口类型的IDL类型（见图20-7）可能是：

IDL: Whiteboard / Shape : 1.0

这个例子表明，一个IDL类型名有三个部分，即IDL前缀、类型名和版本号。由于接口标识符被用作访问接口仓库中接口定义的关键字，因此程序员必须保证提供一个从接口标识符到接口的唯一映射。程序员可以使用IDL前缀杂注将一个附加的字符串作为类型名的前缀，以便将程序员自己的类型和其他程序员的类型区分开来。

IDL杂注指令：IDL杂注指令允许为IDL接口中的组件指定附加的非IDL属性（见Henning and Vinoski [1999]）。这些属性包括（例如）定义一个仅能被本地调用的接口，或者提供一个接口仓库的ID值。每一个杂注都要通过#pragma指定，并且要声明它的类型，例如：

#pragma version Whiteboard 2.3

对CORBA的扩展 CORBA 2.3规约又添加了一些新的特征，包括以值传递方式传输非CORBA对象的能力以及RMI异步变量。Vinoski[1998]在CACM的文章讨论了上述内容。

按值传递的对象：正如我们在上面所看到的，构造类型和简单类型的IDL参数和结果都是以值传递的，而那些指向CORBA对象的参数和结果是以引用传递的。CORBA规约已经支持以值传递非CORBA对象[OMG 2002c]。那些非CORBA对象既有属性又有方法，从这个意义上说它们类似对象。但是，它们仅仅是本地对象，所以它们的操作不能被远程调用。按值传递的方式能够将非CORBA对象的副本在客户和服务器之间传递。

按值传递是通过将一种用以表示非CORBA对象的称为valuetype的类型附加到IDL中实现的。valuetype是一种附加了方法基调的结构（和接口中的那些结构类似）。valuetype类型的参数和结果

是按值传递的，也就是说，状态被传输到远程站点，并用于在目的地上创建一个新的对象。

新对象的方法可以在本地调用，使它的状态脱离了原来对象的状态。传递方法的实现不太直观，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，那么就可以下载代码。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。

按值传递带来的便利是很有用的。例如将一个对象的副本作为值传递给客户端进程，从而能够对该对象方便地使用本地调用。但是，这并不意味着我们可以将CORBA对象作为值来传递。

异步RMI：CORBA现在提供一种异步的RMI，它允许客户端实现对CORBA对象的无阻塞的调用请求[OMG 2004e]。异步RMI是在客户端实现的。因此，服务器并不能了解它是被同步调用的还 843
是异步调用的。（一个例外情况是传输服务，它必须区分这两种调用。）

异步RMI在RMI调用语义的基础上增加了两个新的变量：

- 回调是指客户在每一次调用时，使用一个额外的参数向被调用者传递一个回调的引用，这样，服务器就可以带着结果回调。
- 轮询是指服务器返回一个valuetype对象，它可以用于轮询或等待应答。

异步RMI的结构允许部署一个中间代理来保证请求的执行，并且如果需要的话，中间代理还要保存应答。因此，异步RMI非常适合在客户可能暂时无法联网的环境中使用，例如，在火车上使用笔记本电脑的用户。

20.2.4 CORBA远程对象引用

CORBA 2.0规范描述了一种实用的远程对象引用的格式，而不管该远程对象是否是被实现仓库激活的。采用这种格式的引用称为互操作对象引用（IOR）。下表基于Henning [1998]的描述，它包括了IOR的详细说明：

IOR格式

IDL接口类型ID	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	适配器名字	对象名

- IOR的第一个域说明CORBA对象的IDL接口的类型标识符。注意，如果ORB有一个接口仓库，那么这个类型名也是IDL接口的接口仓库标识符，它允许在运行时获取一个接口的IDL定义。
- 第二个域说明传输协议和该传输协议用以识别服务器的具体要求，特别是，因特网ORB互操作协议（IIOP）使用的是TCP/IP，其中，服务器地址由一个主机域名和一个端口号组成。
- 第三个域由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持续的时间只和驻留这些对象的进程一样长，而持久性IOR能在CORBA对象激活的时间中持续存在。暂态IOR包含了驻留CORBA对象的服务器的具体地址，而持久性IOR包含了它所注册的实现仓库的具体地址。在这两种情形下，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如何用于定位代表CORBA对象的伺服器。 844

暂态IOR：服务器ORB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来定位对象适配器，对象适配器再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。假设在IOR表里提供了对象适配器的名字，如果必要的话它会尝试激活IOR表中所指定的主机地址上的CORBA对象。一旦激活CORBA对象，实现仓库就将它的地址细节返回到客户ORB，客户ORB将它们用作RMI请求消息的目的地（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。

IOR格式中的第二个域可以重复出现，从而可以指定多于一个目的地的主机域名和端口号，以使对象或者实现仓库被复制到几个不同的地方。

在请求-应答协议中，应答消息包括头信息，头信息使持久IOR的过程能被执行。特别是，它包括一个状态项，该项用于标识出请求是否应该被转发到其他的服务器。在此情形下，应答体会包括一个IOR，该IOR含有最近被激活的对象的服务器地址。

20.2.5 CORBA语言映射

从上面的例子我们已经看到，从IDL的类型映射到Java类型是非常直观的。IDL中的简单类型映射到Java中相应的简单类型。struct、enum和union被映射为Java中的类，IDL中的序列和数组被映射为Java中的数组。一个IDL异常映射为一个Java类，该类提供具有这个IDL异常域的实例变量和构造函数。IDL到C++的映射同样也很直观。

但是，我们也已经看到，在将IDL的参数传递语义映射到Java的时候出现了一些困难。尤其是，IDL允许方法通过输出参数返回几个单独的值，而Java只能返回一个结果。Java提供了Holder类来克服这种差异，但是这就要求程序员来使用它们，它们的使用并不是很容易。例如，图5-2中的getPerson方法用IDL可定义成如下形式：

void getPerson (in string name , out Person p) ;

而在Java接口中该方法的定义为：

void getPerson (String name , PersonHolder p) ;

客户必须提供一个PersonHolder的实例作为它的调用的参数。Holder类有一个实例变量，它拥有参数的值，以便客户在调用返回的时候访问该值。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++程序员却要处理由参数带
845 来的与存储管理相关的一系列问题。这些困难在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出现。

例如，在Orbix[Baker 1997]中，ORB负责保存远程对象的引用计数和代理，并在不需要时释放它们。ORB为程序员提供释放或者复制它们的方法。一旦一个服务器的方法执行完毕，就释放out参数和结果，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。当不再需要的时候，必须释放传递给客户的对象引用。类似的规则也应用在变长参数中。

通常，使用IDL的程序员不仅必须学习IDL自身的表示法，还要理解它的参数怎样映射到实现语言的参数上。

20.2.6 CORBA与Web的集成

当Web服务在21世纪早期刚刚出现的时候，CORBA已经比较完善，并且在各个组织中获得了广泛应用。第19章讨论过，Web服务非常适合于基于因特网的各组织之间的协作。19.2.4节曾对Web服务和CORBA进行比较，结果显示，虽然CORBA不适合需要交互的机构的分布式应用，但是它的好处体现在效率方面以及它能提供一系列用于事务、并发控制、安全服务等方面的服务（见20.3节）。

许多组织都依赖于CORBA应用以及它在可靠性与高性能方面的优势。但是，将CORBA服务与 Web服务整合可以获得额外的好处。一个有用的方法就是为现有的CORBA服务提供一种WSDL服务描述（见19.3节）。一个CORBA对象的IDL定义可以通过XML在WSDL服务描述的抽象部分表达，通信协议（例如IIOP）可以在WSDL服务描述的具体部分描述。

这就允许客户像访问其他Web服务一样访问CORBA对象。一旦这成为可能，那么就可以通过

结合带有Web服务接口的CORBA服务同其他Web服务来构建新的Web服务。这也可以使CORBA服务的用户同时获得Web服务的灵活性与轻量级基础结构等好处。

在2004年的秋季，OMG提倡将CORBA绑定到WSDL，这些倡议包括：

- 将IDL定义的CORBA接口映射为WSDL描述。
- 将IDL类型映射为XML schema类型。
- 建立一种按照CORBA对象要求的通信机制，但却可以像访问Web服务一样访问CORBA对象实例的机制。

一旦这项工作完成，就可以用WSDL表示现有的CORBA服务的接口。客户可以像访问Web服务一样访问那些接口，而无需知道底层CORBA中间件的具体实现。客户程序可以被编译为使用 846
WSDL绑定所定义的消息格式、数据表示和通信协议。它们通过URL表示CORBA对象的地址，但是那些CORBA对象要转变为IOR。

20.3 CORBA服务

CORBA包括分布式对象可能需要的服务的规范。特别是，名字服务是对任何ORB的基本补充，正如我们在20.2节的编程例子中所看到的。所有这些服务的索引可在OMG的Web站点上找到[www.omg.org]。Orfali等[1996]对CORBA服务进行了描述，这些服务包括：

名字服务：CORBA名字服务在20.3.1节详细介绍。

事件服务和通知服务：CORBA事件服务在20.3.2节中讨论，通知服务在20.3.3节讨论。

安全服务：CORBA安全服务在20.3.4节中讨论。

交易服务：与名字服务允许CORBA对象按名字定位相反，交易服务[OMG 2000a]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的相关属性到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。客户通过指定所要求的服务类型、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[OMG 2003]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，它从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。并发控制服务[OMG 2000b]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，也可以独立地使用。

持久状态服务：5.2.5节解释了在不需要时通过在持久对象存储中以被动形式存储来实现持久对象，而在需要时激活这些对象的方法。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它们的实现，但是它们并不负责保存和恢复CORBA对象的状态。

CORBA持久状态服务适用于CORBA对象的持久对象存储 [OMG 2002d]。CORBA持久状态服 847
务取代了较早的持久对象服务。持久对象服务基于下面的结构：伺服器能够通过内部接口访问数据存储，例如数据库或文件系统。用于表示持久对象的伺服器被称为*存储对象*，它们被保存在服务器进程和数据存储的*存储家园*中。每一个存储家园仅包含一种类型的存储对象。一种类似Java的语言可以用来定义存储对象的接口，并且将这些接口与存储家园关联起来。伺服器可以创建并且访问它的存储家园中的存储对象。存储对象也可以通过一些程序设计语言，如Java和C++，透明地使用。为了保证透明持久性，预处理器插入一些在伺服器与数据存储之间传输对象的操作。持久状态服务用于事务服务中的事务的上下文中。

生命周期服务：生命周期服务定义了CORBA对象的生成、删除、复制和移动的约定。它定义

了客户如何使用工厂方法在某个位置创建对象，并且在需要的时候允许使用持久对象。它定义了允许客户删除CORBA对象，或将对象移动或复制到指定对象的接口。其中还讨论了深拷贝与浅拷贝的策略问题[OMG 2002e]。

20.3.1 CORBA名字服务

CORBA名字服务是第5章描述的绑定程序的一个更复杂的版本。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上。

正如第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唯一的。名字可能与一个应用中的CORBA对象的对象引用相关，或者与名字服务中另一个上下文相关。上下文可以嵌套在一起，提供一种层次化的名字空间，如图20-9所示，其中CORBA对象正常显示，而名字上下文以无色椭圆显示。左边的图形显示了20.2.1节编程实例中描述的ShapeList对象的项。

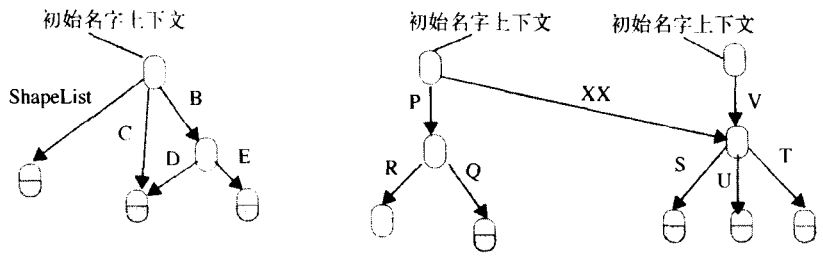


图20-9 CORBA名字服务中的名字图

*初始名字上下文*提供了用于一系列绑定的根。注意，可能有不止一个初始名字上下文指向同一个名字图。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关联的名字服务器可以形成联盟，这将在本节的后面介绍。客户和服务器程序需要从ORB获取初始名字上下文，这通过调用ORB的方法resolve_initial_references，并以“NameService”作为参数来完成，参见图20-5。ORB返回一个NamingContext类型的对象的引用，参见图20-10，它指向那个ORB的名字服务器的初始上下文。因为可能会有多个初始上下文，所以对象没有绝对的名字——名字总是相对于一个初始名字上下文来解释的。

```
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，并将它与原上下文中一个给定的名字绑定在一起。
  Object resolve ( in Name n ) ;
    在该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20-10 IDL中CORBA名字服务的NamingContext接口

一个带有一个或者多个组成部分的名字可以在任意名字上下文中开始进行解析。为了解析一个具有多个组成部分的名字，名字服务在起始的上下文中查找与第一个成分相匹配的一个绑定。如果存在这样的绑定，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，那么名字的第二个组成部分就在那个上下文中解析。这个过程不断重复，直到名字的所有组成部分都已经被解析，并且最终取得了一个远程对象引用为止，除非中途匹配失败。

CORBA名字服务使用的名字由两部分组成，称为NameComponents，其中每个NameComponents包括两个字符串，一个是*名字*，另一个是对象的*种类*。*种类*域提供一个为应用所采用的属性，并且可以包含一些有用的描述信息。它不由名字服务解释。

尽管CORBA对象由名字服务给出了层次化的名字，但是这些名字不能像UNIX文件那样表示为路径名。所以在图20-9中，我们不能用/V/T表示最右边的对象。这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20-10给出了CORBA名字服务的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 2004b]处获得。为了简单起见，我们在图中没有给出方法抛出的异常。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20-5标号为2的行。Resolve的参数是Name类型的，它被定义为一个名字组成部分的序列。这意味着客户在调用之前必须构造一个名字组成部分的序列。图20-5给出了一个客户，它创建了一个称为path的数组，其中包含了一个名字组件，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操作来注册对象的名字，并采用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，并可以在添加绑定的上下文中调用它。参见图20-4，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。

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在一起。另一个称为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在一起。unbind方法可以删除名字，也可以删除上下文。

list操作用于浏览名字服务的一个上下文中的可用信息。它从目标NameContext返回一个绑定的列表。每个绑定包括名字和类型——类型说明它是一个对象还是一个上下文。有时侯，一个名字上下文可能包含大量绑定，在此情形下，我们不希望将它们作为一次调用的结果返回。因此，list方法将最大数量的绑定作为一次list调用的结果返回，如果还要求发送更多绑定的话，它就组织成批返回结果。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20-10给出了list方法，但是为了简单起见，忽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，用于存取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，以将要得到的最大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名字空间支持名字服务的联盟，采用的机制是每个服务器提供名字图的一个子集。例如，在图20-9中，图的中间和右边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

848, 850 "XX"的绑定，指向右边图中的上下文。通过该上下文，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到中间图上的节点的绑定。一个组织可以通过提供远程名字服务器和指向它们的远程引用来对其名字空间中部分或者全部上下文进行访问。

CORBA名字服务的Java实现非常简单。它被称为是透明的是因为它将其所有绑定存储在易失存储器中。任何认真的实现至少都会把它的名字图的拷贝保留在文件中。正如我们在对DNS的研究中看到的，复制可以用于提供更好的可用性。

20.3.2 CORBA事件服务

CORBA事件服务规约定义了许多接口，用于支持兴趣对象（称为提供者）将通知传递给订阅者（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。通知有推和拉两种传播方式。"推"是由提供者到消费者，"拉"是由消费者到提供者。在"推"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"拉"方式中，提供者实现PullSupplier接口，该接口包括一个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，该方法结果是接收到一个通知。

通知本身可作为any类型的参数或结果传输，这意味着对象交换通知时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他们自己的IDL接口，从而可以带有任意类型的通知。

事件通道是用于支持多个提供者与多个消费者以异步方式进行通信的CORBA对象。事件通道作为提供者与消费者之间的缓冲区。它也能向多个消费者组播通知。通过事件通道进行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使用。例如，提供者可以将通知推到通道，再由消费者从通道拉出通知。

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名字服务或者以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通知通道获得代理提供者并与它们相连。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模型，也可以用于拉模型。当提供者采用交互的推模型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20-11所示。如果消费者采用了交互的拉模
851 型，那么消费者将调用拉代理消费者的pull方法。



图20-11 CORBA事件通道

代理消费者和代理提供者的存在使得构造事件通道链成为可能，在事件通道链中，每个通道提供被后续通道消费的通知。CORBA模型中的事件通道与图5-11中定义的观察者很相似。如果用程序实现事件通道的话，可以实现5.4节中讨论的观察者的一些功能。然而，通知不带有任何形式的标识符，因此模式识别或者通知过滤就需要基于由应用放在通知里的类型信息来完成。

Farley[1998]给出了CORBA事件服务的详细解释和主要接口概述。CORBA事件服务的完整规约可参见[OMG 2004c]。然而，规约没有指出如何创建一个事件通道和如何请求需要的可靠性，

20.3.3 CORBA通知服务

CORBA通知服务[OMG 2004d]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。事件服务不支持事件过滤或者说明传输需求。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，如果不具备说明传输需求的能力，则对于所有通过通道发送的通知，它们的传输保证都内置在实现中。

通知服务增加了以下一些新的机制：

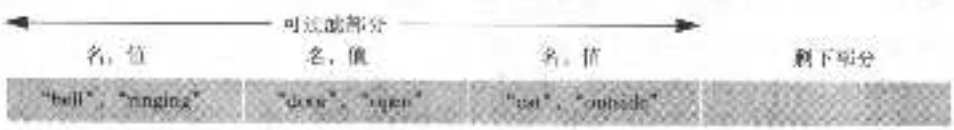
- 通知可以定义为数据结构。这增强了事件服务中通知具有的有限能力。在事件服务中，通知的类型只能是any或者是由应用程序员指定的一种类型。
- 事件消费者可以使用过滤器准确地说明它们对哪些事件感兴趣。过滤器可以连接到通道中的代理。代理将根据在过滤器中对每种通知内容指定的约束来把通知转发给事件消费者。
- 事件提供者具有一种发现消费者对哪些事件感兴趣的手段，这使它们可以只创建那些满足消费者要求的事件。 852
- 事件消费者可以发现在一个通道中由提供者提供的事件类型，这使它们可以及时订阅新出现的事件。
- 可以配置通道、代理或者某个事件的属性。这些属性包括事件传输的可靠性、事件的优先级、排序需求（例如按FIFO或者按优先级排序）以及抛弃存储事件的策略。
- 可以增加事件类型仓库。它将提供对事件结构的访问，从而方便地定义过滤约束。

通知服务引入了结构化事件类型，它提供一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。下面的例子显示了事件头的内容：



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唯一地分类（例如，"stock quote"、"breakfast time"、"burglar alarm"）。事件名唯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，它可以用于说明关于事件传输的可靠性和其他需求。

下面的例子给出了一个结构化事件体的信息：



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，用于规范用在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，当警报响起的时候，该事件会包括警铃的状态、前门是否是开的和猫在哪儿。事件体的剩下部分用于传递与特定事件相关的数据。例如，当警报响起的时候，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。一个过滤器被设计为一个约束的集合，每个约束是一个由两部分组成的数据结构：

- 一张数据结构的列表，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"。该列表包括约束应该用到的所有事件类型。 853
- 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")

我们的例子使用了非形式化的语法。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。

20.3.4 CORBA安全服务

CORBA安全服务[Blakly 1999, Baker 1997, OMG 2002b]包括如下内容：

- 主体（用户和服务器）的认证；为主体创建凭证（即申明他们权利的证书）；支持 7.2.5节中描述的凭证的委托。
- 当CORBA对象接收远程方法调用的时候可以应用访问控制。访问权限可以在访问控制列表（ACL）中说明。
- 客户与对象之间通信的安全，保护消息的完整性和机密性。
- 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远程调用的时候，服务器创建并存储一些凭证，证明已经由服务器代表请求主体执行过该调用。

为了保证这种安全性正确地应用到远程方法调用，安全服务要求利用ORB的协作代表ORB。为了进行一次安全的远程方法调用，要在请求消息中发送客户的凭证。当服务器接收到一个请求消息的时候，它就要验证客户的凭证，检查它们是否是新的、是否为可接受的权威签署的。如果这些凭证是有效的，它们就可以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利。做出这一决定要查询一个对象，该对象包含了目标对象上的每个方法都允许由哪些主体访问的信息（可能是以ACL的形式）。如果客户有足够的权利，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凭证。目标对象还可以将调用的细节记录在审计日志里或者存储在不可抵赖的凭证中。

CORBA允许根据需求指定不同类型的安全策略。消息-保护策略申明客户或者服务器（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/修改。还可以指定审计和不可抵
854 赖的政策。例如，一个策略可以申明它们应该应用哪些方法和参数。

访问控制需要考虑到，许多应用有大量的用户和更大量的对象，每个用户和对象都有它自己的方法集。根据用户的角色，为他们提供了一种专门的称为特权的凭证。将对象分组成若干个域，每个域有一个访问控制政策，规定具有某些特权的用户对该域中对象的访问权限。考虑到还有各种难以预见的方法，将每种方法都归类为四种通用方法（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方法使对象做一些工作，而manage方法完成那些一般情况下不会用到的特殊功能。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指定其访问权限。这项工作就涉及那些正致力于应用访问控制、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请求适当特权的系统设计者。

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政策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凭证和特权，以提供像口令之类的认证数据。

20.4 小结

CORBA的主要组件是对象请求代理（ORB），它允许以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互操作协议（GIOP）包括一种称为CDR的外部数据表示，它使客户和服务器之间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-应答协议的操作规约。
- 因特网ORB互操作协议（IIOP）在TCP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。就如何访问一个CORBA对象，客户所需要知道的就是那些在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，它们是由IDL接口根据客户方语言自动创建的。CORBA对象的服务器骨架也是由IDL接口根据客户方语言自 855
动创建的。对象适配器是CORBA服务器的重要组件，它的任务包括激活与解除伺服器，生成远程对象引用，并且将请求信息转发给合适的伺服器。

CORBA体系结构允许CORBA对象在需要的时候被激活，要做到这一点需借助于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接口定义的数据库，因为一个CORBA对象的IOR包含了它的接口的仓库ID，所以为了支持动态方法调用，接口仓库可以用于获得与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，从而为应用提供额外的服务（如名字和目录服务、事件通知、事务或者安全等）。

练习

20.1 Task Bag是一个对象，存储（关键字，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指定将被存储的关键字和值。

pairIn：它的第一个参数让客户指定将从Task Bag中删除的（关键字，值）对中的关键字。其中的值则通过第二个参数提供。如果没有相匹配的（关键字，值）对，就抛出一个异常。

readPair：除了把（关键字，值）对留在Task Bag中之外，其他与pairIn相同。

采用CORBA IDL定义Task Bag的接口。定义一个异常，一旦任意一个操作无法执行的时候，抛出该异常。所定义的异常应该返回一个用来标识问题号的整数和一个描述问题的字符串。Task Bag接口应该定义一个单独的属性，给出包中任务的数目。（第839页）

20.2 定义方法pairIn和readPair的另一种基调，它的返回值没有相匹配的（关键字，值）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。你会用哪个方法标识像关键字中包含了非法字符这样的错误？（第840页）

20.3 Task Bag接口中的哪个方法可以被定义为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的其他部分不同？（第840页）

20.4 IDL的union类型可以作为一种参数，在少数几种类型中只传输一个。采用union定义一种参数类型，该类型有时侯是空的，有时侯有值类型。（第842页） 856

20.5 在图20-1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。给出一些在一个IDL接口中应该选择数组还是序列的建议。（第842页）

20.6 Task Bag设计用于协作的客户，其中一部分客户把（描述任务的）（关键字，值）对添加进去，而另一些客户则把它删除掉（并执行所描述的任务）。当一个客户被通知说没有相匹配的对的时候，它就不能继续它的工作，直到出现一个可用的对为止。定义一个合适的回调接口以便用在这样的情形中。（第835页）

20.7 描述对Task Bag接口的必要修改，使得可以使用回调。（第835页）

20.8 Task Bag接口中方法的哪个参数是按值传递的？哪个参数是按引用传递的？ （第830页）

20.9 使用Java IDL编译器处理你在练习20.1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方法的基调定义。再看看为方法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。 （第845页）

20.10 给出一个例子，说明Java客户将如何访问用于给出Task Bag对象中任务个数的属性。一个属性与一个对象的实例变量不同有哪些不同？ （第841页）

20.11 解释为什么一般来说远程对象的接口，特别是CORBA对象的接口不提供构造函数。解释怎样在没有构造函数的情况下创建CORBA对象。 （第5章，第833页）

20.12 根据练习20.1用IDL重新定义任务包接口，以使它可以使用struct表示一个Pair，该结构包括一个Key和一个Value。注意，不必使用typedef定义struct。 （第842页）

20.13 从伸缩性和容错的角度讨论实现仓库的功能。 （第838页，第844页）

20.14 通过怎样扩展，CORBA对象可以从一个服务器迁移到另一个服务器？ （第838页，第844页）

20.15 讨论CORBA名字服务中的两部分名字或NameComponent的优点和缺点。 （第848页）

20.16 给出一个算法，描述在CORBA名字服务中如何解析一个多部分名字。一个客户程序需要相对于初始名字上下文解析一个带有“A”、“B”和“C”组成部分的多部分名字。在名字服务中，怎样指定resolve操作的参数？ （第848页）

20.17 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。每个公司都希望只给其他公司提供与该项目有关的CORBA对象的访问权限。描述一种合适的方法，可以让整个公司联合它们的CORBA名字服务。 （第850页）

857

20.18 在共享白板应用程序上下文中，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分别决定终止事件通信。 （第851页）

20.19 根据你在20.18中的解决方案，说明如何将在提供者和消费者之间插入一个事件通道，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能得到以如下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 ) ;
};
```

使用事件通道有什么好处？ （第851页） 858

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），520
access control（访问控制），281~283
access control list（访问控制列表），281, 330
access rights（访问权限），57
access transparency（访问透明性），23, 329, 348
ACID properties（ACID特性），519
ack-implosion（确认爆炸），486
activation（激活），194
activator（激活器），194
active badge（主动标记），660, 683, 685, 694
  events（事件）678
active bat 693
active object（主动对象）194
active replication（主动复制），620~622
actuator（制动器）663
ad hoc network（自组织网络）119, 687
  mobile（移动）688
ad hoc routing（自组织路由）664
adaptation（自适应）
  energy-aware（能量敏感）708
  of content（内容）705~707
  to resource variations（资源变化）707~709
address resolution protocol（ARP）（地址解析协议），95
address space（地址空间），227, 229~230
  aliasing（别名），239
  inheritance（继承），232
  region（区域），229
  shared region（共享区域），230, 250
admission control（许可控制），730, 737~738
AES(Advance Encryption Standard)（高级加密标准），292, 303
agreement（协定）
  in consensus and related problems（在共识和相关问题中），500~501
  of multicast delivery（关于组播传递的），487
  problems of（的问题），499
  uniform（一致的），490
alias（别名），373
Amoeba
  multicast protocol（组播协议），494
  run server（运行服务器），231
Andrew File System(AFS)（Andrew文件系统（AFS）），332, 349~358
  for Linux（Linux上的），332
  in DCE/DFS（DCE/DFS上的），361
  performance（性能），358
  wide-area support（广域支持），358
anti-entropy protocol（反熵协议），630, 632
Apollo Domain（Apollo域），750
applet（小程序），14, 38
  threads within（内部线程），240
application layer（应用层），76, 78
architectural models（体系结构模型），31~46
ARP（参见address resolution protocol）
association（关联），661, 665, 666~675
  direct（直接），674
  indirect（间接），681
  physical（物理），674
  problem（问题），667
  secure spontaneous（安全自发），698~702
  spontaneous（自发），665
asymmetric cryptography（非对称密码学），286, 293~295
asynchronous communication（异步通信），134
asynchronous distributed system（异步分布式系统），50, 439, 498, 507
asynchronous invocation（异步调用），225
  persistent（持久），255
  in CORBA（CORBA中的），843
asynchronous operation（异步操作），253~256
  in DSM（DSM中的），773
at-least-once invocation semantics（至少一次调用语义），188
ATM(Asynchronous Transfer Mode)（异步传输模式），71, 74, 75, 78, 86, 113, 124~127
at-most-once invocation semantics（至多一次调用语义），188
atomic commit protocol（原子提交协议），566~578

failure model（故障模型），570
two-phase commit protocol（两阶段提交协议），570
atomic consistency（原子一致性），758
atomic operation（原子操作），514
atomic transaction （原子事务，参见transaction）
audio conferencing application（音频会议应用），724
authentication（认证），60, 276~278
location-based（基于位置的），702
authentication server（认证服务器），306
authentication service（认证服务），309
automatic identification（自动标识），694
availability（可用性），22, 604, 622~641

B

backbone（主干网），96
bandwidth（带宽），49
base station, wireless（基站，无线），119, 659, 692
Bayou, 399, 632~634
dependency check（依赖检查），633
merge procedure（合并过程），633
beacon（信标），713
infrared（红外线），674
radio（无线电），692
Bellman-Ford routing protocols（Bellman-Ford路由协议），82
best-efforts resource scheduling（最佳资源调度），728
big-endian order（大序法排序），144
binder（绑定器），193
portmapper（端口映射器），200
RMIiregistry, 210
binding（绑定）
socket（套接字），169
birthday attack（生日攻击），300
BitTorrent, 399, 409
block cipher（块密码），287~288
blocking operations（阻塞操作），134
Bluetooth network（蓝牙网络），113, 121~124
boundary principle（边界原理），667
bridge（网桥），88
broadcast（广播），484
brute-force attack（强行攻击），286
byzantine failure（拜占庭故障），55
byzantine generals（拜占庭将军问题），501, 504~507

C

cache（缓存，高速缓冲存储器），38, 45, 604
coherence of cached files（缓存文件的一致性），639
caching（高速缓冲存储）
file, write -through（文件，写透），344
files at client（客户端文件），344
files at server（服务器端文件），343
of whole files（所有文件的），350
validation procedure（验证程序），345, 355
callback（回调）
in CORBA remote method invocation（在CORBA远程方法调用中），835
in Java remote method invocation（在Java远程方法调用中），214
callback promise（回调承诺），355
camera phone（照相手机），664, 674, 700, 712
CAN routing overlay（CAN路由覆盖），399, 409
capability（权能），281
cascading abort（连锁放弃），527
case studies（实例研究）
Andrew File System(AFS)（Andrew文件系统(AFS)），349~358
ATM(Asynchronous Transfer Mode)（异步传输模式），124~127
Bayou, 632~634
Coda file system（Coda文件系统），634~640
Cooltown, 710~716
CORBA, 828~856
Domain Name System(DNS)（域名系统），378~385
Ethernet network（以太网），113
Global Name Service（全局名字服务），387~390
Gossip architecture（Gossip体系结构），622~632
Grid（网格），814~823
IEEE 802.11 wireless LAN（WiFi)（IEEE 802.11 无线局域网），118
IEEE 802.15.1 Bluetooth wireless PAN（IEEE 802.15.1蓝牙无线PAN），121
Internet protocols（因特网协议），89
Interprocess communication in UNIX（UNIX系统的进程间通信），168~171
IP routing（IP路由），96
Ivy DSM system（Ivy DSM系统），767~771
Ivy file system（Ivy文件系统），426~429
Java remote method invocation（Java远程方法调用），208~216
Kerberos（Kerberos认证），307~312
Munin, 775~777
Needham-Schroeder protocol（Needham-Schroeder协议），306~307
Network File System(NFS)（网络文件系统(NFS)），337~349
Network Time Protocol（网络时间协议），441~444

OceanStore, 422~426
Pastry routing overlay（Pastry路由覆盖）, 410~418
Squirrel web cache（Squirrel web缓存）, 420~422
Sun RPC, 198~201
Tapestry routing overlay（Tapestry路由覆盖）, 418~419
Tiger video file server（Tiger视频文件服务器）, 742
Transport Layer Security（TLS）（传输层安全）, 313~317
WiFi security（WiFi安全）, 317~319
X.500 directory service（X.500目录服务）, 390~394
catch exception（捕获异常）, 183
causal consistency（因果一致性）, 777
causal ordering（因果排序）, 445
of multicast delivery（组播传递的）, 491
of request handling（请求处理的）, 608
CDR（参见CORBA）
Common Data Representation（公共数据表示）
Cell phone（蜂窝电话，参见mobile phone）
certificate（证书）, 279~281
X.509 standard format（X.509标准格式）, 301
certificate authority（证书权威机构）, 302
certificate chain（证书链）, 280, 302
CGI（参见common gateway interface）
challenge, for authentication（质询，用于认证）, 277
checkpointing（检查点）, 593
Chord routing overlay（Chord路由覆盖）, 399
Chorus（Chorus系统）, 259
chosen plaintext attack（明文选择攻击）, 294
cipher block chaining(CBC)（密码块链接）, 287
cipher suite（密码组）, 314
ciphertext（密文）, 286
circuit switching network（电路交换网络，参见network, circuit switching）
classless interdomain routing(CIDR)（无类别域间路由）, 98, 399
clients（客户）, 8
client-server communication（客户－服务器通信）, 155~164
client-server model（客户－服务器模型）, 35~38
variations（变体）, 37~42
clock（时钟）
accuracy（精确）, 437
agreement（协定）, 437
computer（计算机）, 45, 49, 435
correctness（正确性）, 438
drift（漂移）, 49, 436
faulty（故障的）, 438
global（全局的）, 2
logical（逻辑的）, 446
matrix（矩阵）, 448
monotonicity（单调性）, 438
resolution（分辨率）, 436
skew（偏移）, 436
synchronization（同步，参见synchronization of clocks）
vector（向量）, 447
cluster, of computers（计算机集群）, 37, 231
Coda file system（Coda文件系统）, 428, 634~640
available volume storage group(AVSG)（可用的卷存储组）, 635
Coda version vector(CVV)（Coda版本向量）, 636
volume storage group(VSG)（卷存储组）, 635
codec（多媒体数字信号编/解码器）, 728
coherence（连贯性）
of distributed shared memory（分布式共享内存）, 759~760
collision detection（冲突检测）, 116
commit（提交）, 520
common gateway interface（公共网关接口）, 14
communication（通信）
asynchronous（异步）, 134
client-server（客户－服务器）, 155~164
group（组）, 164~168
operating system support for（操作系统支持）, 245~253
reliable（可靠的）, 56, 135
synchronous（同步）, 134
communication channels（通信通道）
performance（性能）, 49
threats to（威胁）, 59
commuting operations（可交换操作）, 621
competing memory accesses（竞争性内存访问）, 772
Composite Capabilities/Preferences profile（CC/PP）（复合能力/偏好文件）, 706
concurrency（并发）, 2, 23
of file updates（文件更新的）, 329
concurrency control（并发控制）, 521~563
comparison of methods（方法的比较）, 556
conflicting operations（冲突的操作）, 524
in CORBA concurrency control service（在CORBA并发控制服务中）, 534, 847
in distributed transaction（在分布式事务中）, 578~581
inconsistent retrieval（不一致检索）, 522
with locks（用锁，参见locks）

lost update（更新丢失），521
operation conflict rules（操作冲突规则），524, 532
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（冲突操作），524
confusion(in cryptography)（含混(在密码学中)），289
congestion control（拥塞控制，参见network congestion control）
connection（连接）
persistent（持久），251
consensus（共识），499~510
impossibility result for an asynchronous system（异步系统的不可能性结论），507
in a synchronous system（在同步系统中），503
related to other problems（与其他问题的关系），502
consistency（一致性）
of replicated data（复制数据的），605
of shared memory（共享内存的），757
context（上下文），683
context switch（上下文转换），238
Context Toolkit, 686
context-aware computing（上下文敏感计算），660
context-awareness（上下文敏感），683~696
cookie（Web服务器发给Web浏览器的一小段信息），340
Cooltown, 710~716
beacon（信标），713
Coordinated Universal Time (UTC)（通用协调时间），436
copy-on-write（写时复制），233
CORBA（公共对象请求代理体系结构）
architecture（体系结构），836~839
object adapter（对象适配器），836
object request broker（对象请求代理, ORB），836
proxy（代理），837
skeleton（骨架），837
case study（实例研究），828~856
client and server example（客户和服务器实例），831~836
Common Data Representation（公共数据表示），146~148
compared with web services（与web服务比较），799
dynamic invocation interface（动态调用接口），838
dynamic skeleton interface（动态骨架接口），839
efficiency compared with web services（与web服务比较的效率），800
extensions（扩展），843~844
asynchronous RMI（异步RMI），843
objects by value（按值传递的对象），843
general Inter-ORB protocol(GIOP)（通用ORB间互操作协议），828
implementation repository（实现仓库），838
integration with web services（与web服务集成），846
interface definition language（接口定义语言），181, 829~831, 839~844
attribute（属性），841
inheritance（继承），841
interface（接口），839
method（方法），840
module（模块），839
parameters and results（参数和结果），829
pseudo object（伪对象），831
type（类型），841
interface repository（接口仓库），838
Internet Inter-ORB protocol(IIOP)（因特网ORB间互操作协议），828
interoperable object reference(IOR)（互操作对象引用），844
compared with URL（与URL比较），799
persistent（持久），844
transient（暂态），844
language mapping（语言映射），845
marshalling（编码），147
object model（对象模型），829
object request broker(ORB)（对象请求代理），828, 836
remote method invocation（远程方法调用），829~836
callback（回调），835
remote object reference（远程对象引用，参见CORBA interoperable object reference）
services（服务），847~856
concurrency control service（并发控制服务），534, 847
event service（事件服务），851~852
life cycle service（生命周期服务），848
naming service（名字服务），848~851
notification service（通知服务），852~854
persistent state service（持久状态服务），325, 847
security service（安全服务），854~855
trading service（交易服务），847
transaction service（事务服务），847
crash failure（崩溃故障），53, 470
credentials（证书），283~285
critical section（临界区），471

cryptography（密码学）, 59, 275~279, 286~295
and politics（和政治学）, 304
performance of algorithms（算法的性能）, 303
CSMA/CA（具有避免冲突的载波侦听多路访问）, 120
CSMA/CD（具有检测冲突的载波侦听多路访问）, 114
cut（割集）, 451
consistent（一致的）, 451
frontier（边界）, 451
Cyber Foraging, 708

D

data compression（数据压缩）, 728
data link layer（数据链路层）, 78
data streaming（数据流，参见network, data streaming）
datagram（数据报）, 80, 94
data-oriented programming（面向数据编程）, 677~683
deadlock（死锁）, 538~541
definition（定义）, 539
detection（检测）, 449, 540
distributed（分布式的，参见 distributed deadlock）
prevention（预防）, 540, 541
timeouts（超时）, 541
wait-for graph（等待图）, 539
with read-write locks（读-写锁）, 538
dealing room system（交易所系统）, 203
debugging distributed programs（调试分布式程序）, 450, 457~463
delay-tolerant networking（容延迟网络，参见disruption-tolerant networking）, 689
delegation (of rights)（权限的委托）, 284
delivery failures（发送故障）, 158
denial of service（拒绝服务）
sleep deprivation torture attack（睡眠剥夺折磨攻击）, 697
denial of service attack（拒绝服务攻击）, 19, 61, 269
dependability（可依赖性）, 46
DES(Data Encryption Standard)（数据加密标准）, 291, 303
design requirements（设计需求）, 43
detecting failure（检测故障）, 21
device discovery（设备发现）, 668
DHCP-Dynamic Host Configuration Protocol（DHCP, 动态主机配置协议）, 99, 104, 667
Diffie-Hellman protocol（Diffie-Hellman协议）, 701
diffusion (in cryptography)（扩散（在密码学中））, 289
digest function（摘要函数）, 297
digital signature（数字签名）, 278~279, 295~302
in XML（在XML中）, 810
directed diffusion（定向扩散）, 689
directory service（目录服务）, 335~337, 386
attribute（属性）, 386
discovery service as（作为发现服务）, 668
in the Grid（在网格中）, 823
UDDI, 805~807
dirty read（读取脏数据）, 526
disconnected operation（断链操作））, 605, 632, 640, 664
discovery service（发现服务）, 386, 668~673
Jini（Sun的一种分布式技术）, 671~673
serverless（无服务器）, 671
dispatcher（分发器）, 191
generic in Java RMI（Java RMI中的通用分发器）, 215
in CORBA（CORBA中的）, 837
in web services（web服务中的）, 798
disruption-tolerant networking（容中断网络）, 689
distance vector routing algorithm（距离向量路由算法）, 82
distributed deadlock（分布式死锁）, 581~589
edge chasing（边追逐）, 584~589
transaction priorities（事务优先级）, 586
phantom deadlock（假死锁）, 583
distributed garbage collection（分布式无用单元收集）, 186, 195~196
in Java（在Java中）, 186, 195~196
use of leases（租借的使用）, 196
distributed object（分布式对象）, 183
communication（通信）, 181~197
distributed object model（分布式对象模型）, 184
compared with web services（与web服务比较）, 794
distributed operating system（分布式操作系统）, 223
distributed shared memory（分布式共享内存）, 325, 749~779
byte-oriented（面向字节的）, 755
centralized manager（中央管理器）, 767
consistency model（一致性模型）, 756~760, 777~778
design and implementation issues（设计和实现问题）, 754~763
distributed manager（分布式管理器）, 768~771
granularity（粒度）, 762
immutable data stored in（存储的不变数据）, 755
object-oriented（面向对象的）, 755
synchronization model（同步模型）, 756
thrashing（系统颠簸）, 763
update options（更新选项）, 760~762
write-invalidation（写失效）, 761, 765~771
write-update（写更新）, 760
distributed transaction（分布式事务）

atomic commit protocol（原子提交协议），566~578
concurrency control（并发控制），578~581
locking（锁），578
optimistic（乐观的），580
timestamp ordering（时间戳排序），579
coordinator（协调者），568~569
flat（平面），566
nested（嵌套），566
one-phase commit protocol（单阶段提交协议），569
two-phase commit protocol（两阶段提交协议），570.
DNS（参见Domain Name System）
Domain Name System (DNS)（域名系统），108, 378~385
BSD implementation（BSD实现），384
domain name（域名），379
name server（名字服务器），381~385
navigation（导航），383
query（查询），380
resource record（资源记录），383
zone（区域），381
domain transition（域转换），238
downloading of code（下载的代码），14
in Java remote method invocation（在Java远程方法调用中），210
DSL (digital subscriber line)（数字用户线），71, 86
DSM（参见distributed shared memory）
dynamic invocation（动态调用），192
in CORBA（在CORBA中），838
in web services（在web服务中），789
dynamic invocation interface（动态调用接口），192, 798, 838
dynamic skeleton（动态骨架），193, 839
in CORBA（在CORBA中），839
dynamic web pages（动态web页面），13~14

E

eager update propagation（及时更新传播），643, 775
eavesdropping attack（窃听攻击），269
election（选举），479~484
bully algorithm（霸道算法），482
for processes in a ring（对环中进程的），480
electronic commerce（电子商务）
security needs（安全性需要），271
elliptic curve encryption（椭圆曲线加密），295
emulation（模拟）
of operating system（操作系统的），258
encapsulation（封装），76, 91
encryption（加密），60. 275~279
in XML security（在XML安全中），811
enemy（敌方），58
enemy, security threats from an（来自敌方的安全威胁），61
energy consumption（能量消耗），663, 665
and adaptation（和适应性），708
and denial of service（和拒绝服务），697
of communication（通信的），688
of compression（压缩的），707
of discovery protocols（发现协议的），671
energy-aware adaptation（能量敏感自适应），708
entry consistency（变量项一致性），778
esquirt, 714~715
Ethernet for real-time applications（实时应用的以太网），118
Ethernet hub（以太网网络集线器），88
Ethernet network（以太网网络），88, 112, 113~118
Ethernet switch（以太网交换机），88, 118
event（事件），201~203
active badge（主动标记），678
composite（复合的），678
concurrency（并发），446
delivery semantics（传递语义），205
filtering（过滤），206
forwarding（转发），206
heap（堆），679
in CORBA event service（在CORBA事件服务中），851~852
in CORBA notification service（在CORBA通知服务中），852~854
in Jini（在Jini中），206~208
in volatile system（在易变系统中），677~678
notification（通知），201~203
object of interest（兴趣对象），204
observer（观察者），205
ordering（排序），51
patterns（模式），206
publisher（发布者），205
subscriber（订阅者），204
system, compared to tuple space（系统，与元组空间比较），679
type（类型），201, 203
exactly once（恰好一次），188
exception（异常），183
catch（捕获），183
in CORBA remote invocation（在CORBA远程调用中），189
in Java remote method invocation（在Java远程方法调用中），189
throw（抛出），183

execution environment （执行环境）, 228
Exokernel（Exokernel内核设计）, 260
external data representation （外部数据表示）, 144~149

F

factory method （工厂方法）, 193, 833
factory object （工厂对象）, 193
fail-stop（故障-停止）, 53
failure（故障）
  arbitrary（随机的）, 55
  byzantine（拜占庭）, 55
  timing（时序）, 55
failure atomicity （故障原子性）, 518, 589
failure detector（故障检测器）, 470~471
  to solve consensus （为了解决共识）, 508
failure handling（故障处理）, 21, 22
failure model（故障模型）, 53~57
  atomic commit protocol （原子提交协议）, 570
  IP multicast （IP组播）, 166
  request-reply protocol（请求-应答协议）, 158
  TCP, 141
  transaction（事务的）, 517
  UDP, 137
failure transparency（故障透明性）, 24
fairness（公平性）, 472
false sharing（错误共享）, 762
familiar stranger（熟悉的陌生人）, 693
fault-tolerant average（容错平均值）, 441
fault-tolerant service（容错服务）, 605, 615~622
fidelity（保真度）, 708
FIFO ordering（FIFO排序）
  of multicast delivery （组播传递的）, 490
  of request handling（请求处理的）, 608
file（文件）
  mapped（映射的）, 230, 750
  replicated（复制的）, 635
file group identifier（文件组标识符）, 337
file operations（文件操作）
  in directory service model（在目录服务模型中的）, 336
  in flat file service model（在平面文件服务模型中的）, 334
  in NFS server（在NFS服务器中的）, 340
  in UNIX（在UNIX中的）, 328
file-sharing application（文件共享应用）, 399
Firefly RPC（Firefly远程过程调用）, 250
firewall（防火墙）, 4, 108~112, 285, 794
flat file service（平面文件服务）, 333~335
flow control（流控制）, 107
flow specification（流规约）, 735
frame relay （帧中继，参见network, frame relay）
Frangipani distributed file system（Frangipani 分布式文件系统）, 364
Freenet, 399, 403
front end （前端）, 608
FTP（文件传输协议）, 78, 80, 90, 110, 111
fundamental models（基础模型）, 47~61

G

Galileo, satellite navigation system（伽利略，卫星导航系统）, 691
garbage collection（无用单元收集）, 183, 449
  distributed（分布式的）, 195~196
  in distributed object system（在分布式对象系统中的）, 186
  local（局部的）, 183
gateway （网关）, 72
geographical information system（地理信息系统）, 694
global clock（全局时钟）, 2
Global Name Service（全局名字服务）, 387~390
  directory identifier（目录标识符）, 387
  working root（工作根）, 389
Global Positioning System (GPS)（全球定位系统）, 437, 691
global state（全局状态）, 448~463
  consistent（一致性）, 452
  predicate（谓词）, 452
  snapshot（快照）, 453~457
  stable（稳定的）, 452
Globus toolkit（Globus工具包）, 822
GLONASS, 691
glyph（象形）, 674
GNS（参见Global Name Service）, 387
Gnutella, 399
gossip architecture（gossip体系结构）, 622~632
  gossip message （gossip消息）, 624
    processing（处理）, 629~630
    propagating（传播）, 630~631
  query processing（查询处理）, 627
  update processing（更新处理）, 627~629
GPS（参见Global Positioning System）
Grid（网络）, 814~823
  application examples（应用实例）, 820
  data-intensive application characteristics（数据密集型应用特征）, 816
  directoty service（目录服务）, 823

Globus toolkit（Globus工具包），822
Open grid services architecture（OGSA）（开放的网格服务体系结构），817
Open grid services infrastructure（OGSI）（开放的网格服务基础设施），819
naming scheme（命名方案），819
service creation and deletion（服务创建和删除），819
service data（服务数据），819
requirements for（需求），816
World-Wide Telescope, 815
group（组）
closed（封闭的），486
membership service（成员服务），609
of objects（对象的），614
open（开放的），486
overlapping（重叠的），498
view（视图），610, 611~614
group communication（组通信），164~168, 484, 609~615
view-synchronous（视图同步），612~614
GSM mobile phone network（GSM蜂窝式电话网），72, 692

H

handheld computing（手持计算），658
handle system（处理系统），372
handshake protocol, in TSL（握手协议，在TSL中），314
happened-before（发生在先），445
heartbeat message（心跳消息），416
heterogeneity（异构性），16, 17, 330, 331
name service（名字服务），375
historical notes（历史记录）
distributed file systems（分布式文件系统），324
emergence of modern cryptography（现代密码学的出现），267
history（历史），435
global（全局的），451
of server operations in request-reply（请求-应答中服务器操作的），159
hold-back queue（保留队列），489
hostname（主机名），370
HTML（超文本标记语言），10
HTTP（超文本传输协议），13, 78, 80, 90, 109, 160~164
in SOAP（在SOAP中），792
over persistent connection（基于持久连接的），255
performance of（性能的），251
hub（网络集线器），88
hybrid cryptographic protocol（混合密码协议），278, 295
hybrid memory consistency model（混合内存一致性模型），777
hyperlink（超链接），9
physical（物理的），710, 712~714

I

IANA (Internet Assigned Numbers Authority)（因特网编号管理局），80, 379
IDEA (International Data Encryption Algorithm)（国际数据加密算法），292, 303
idempotent operation（幂等操作），158, 188, 334
identifier（标识符），368, 370
IDL（参见interface definition language）
IEEE 802 standards（IEEE 802标准），112
IEEE 802.11 (WiFi) network（IEEE 802.11(WiFi)网络），113, 118~121
security（安全），317~319
IEEE 802.15.1 (Bluetooth) network（IEEE 802.15.1（蓝牙）网络），113, 121~124
IEEE 802.15.4 (ZigBee) network（IEEE 802.14（Zig Bee）网络），113
IEEE 802.16 (WiMAX) network（IEEE 802.16（WiMAX）网络），113
IEEE 802.3 (Ethernet) network（IEEE 802.3（以太网）网络），88, 112, 113~118
IEEE 802.4 (Token Bus) network（IEEE 802.4（令牌总线）网络），113
IEEE 802.5 (Token Ring) network（IEEE 802.5（令牌环）网络），112
implementation repository, in CORBA（实现仓库，在CORBA中），838
inconsistent retrieval（不一致检索），522, 533
independent failure（独立故障），2
information leakage as a security threat（信息泄露作为一个安全威胁），270
initialization vector (for a cipher)（初始化向量（用于密码）），288
i-node number（i节点号），339
input parameters（输入参数），180
integrity（完整性）
in consensus and related problems（在共识和相关问题中），500~502
of multicast delivery（组播传递的），487
of reliable communication（可靠通信的），56
intentions list（意图列表），590, 595
interaction model（交互模型），48~53
interactive consistency（交互一致性），501
interface（接口），42, 179~181, 182
in distributed system（在分布式系统中），179

remote（远程），180
service（服务），180
interface definition language（接口定义语言），179, 180
CORBA, 829~831, 839~844
CORBA IDL example（CORBA IDL例子），181
in web services（在web服务中），800~805
Sun RPC example（Sun RPC例子），198
interface repository, in CORBA（接口仓库，在CORBA中），838
International Atomic Time（国际原子时间），437
Internet（因特网），3, 4, 80, 89~112
routing protocols（路由协议），96~99
Internet address（因特网地址，参见IP adress）
Internet protocol（因特网协议，参见IP）
Internet telephony（因特网电话），725
internetwork（互联网），72, 79, 86~89
interoperable Inter-ORB protocol (IIOP)（因特网ORB互操作协议），844
interoperation（互操作）
spontaneous（自发的），665~666
interprocess communication（进程间通信）
characteristics（特征），133
in UNIX（在UNIX中），168~171
intranet（企业内部网），4, 5
invocation mechanism（调用机制），224
asynchronous（异步），255
latency（延迟，等待时间），248
operating system support for（操作系统支持），245~253
performance of（性能），247~253
scheduling and communication as part of（其中的调度和通信），224
throughput（吞吐量），249
within a computer（在一个计算机内），251~253
invocation semantics（调用语义），
at-least-once（至少一次），188
at-most-once（至多一次），188
maybe（或许），188
invokes an operation（调用一个操作），8
IOR（参见CORBA）
interoperable object reference（互操作对象引用）
IP, 78, 94~105
API（应用程序接口），133~144
IP address, Java API（IP地址, Java API），136
IP addressing（IP寻址），92~94
IP multicast（IP组播），89, 165~167, 484, 488
address allocation（地址分配），165
failure model（故障模型），166
Java API（Java API），166
router（路由器），165
IP spoofing（IP伪冒），96
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, 101~104
ISDN（综合业务数字网），78
ISIS（智能系统和信息服务），614
isochronous data streams（等时数据流），727
Ivy DSM system（Ivy DSM系统），767~771
Ivy file system（Ivy文件系统），426~429

J

Java
object serialization（对象序列化），148~150
reflection（反射），149
thread（线程，参见thread, Java）
Java API
DatagramPacket（DatagramPacket类），138
DatagramSocket（DatagramSocket类），138
InetAddress（InetAddress类），136
MulticastSocket（MulticastSocket类），166
ServerSocket（ServerSocket类），142
Socket（Socket类），142
Java remote method invocation（Java远程方法调用），208~216
callback（回调），214
client program（客户程序），213
design and implementation（设计和实现），215~216
downloading of classes（类下载），192, 210
dynamic invocation（动态调用），192
exception（异常），189
parameter and result passing（参数和结果传递），209
remote interface（远程接口），209
RMIregistry, 210
server program（服务器程序），211
use of reflection（反射的使用），215
Java security（Java安全性），270
JetSend, 680
Jini
discovery service（发现服务），671~673
distributed event specification（分布式事件规约），206~208
leases（租借），197
jitter（抖动），49

K

Kademlia routing overlay（Kademlia路由覆盖），399, 409

Kazaa, 399
Kerberos, 307~312
Kerberos authentication （Kerberos 认证）
  for NFS（NFS的）, 346
kernel（内核）, 227
  monolithic（整体的）, 257
keystream generator（密钥流产生器）, 288

L

L4 microkernel（L4 微内核）, 260
Lamport timestamp （Lamport时间戳）, 446
LAN（参见network, local area）
latency（延迟，等待时间）, 49
layers in protocols （协议中的层）, 76
lazy update propagation（惰性更新传播）, 643, 775
LDAP（参见lightweight directory access protocol）
leaky bucket algorithm（漏桶算法）, 734
leases（租借）, 196, 214
  for callbacks（用于回调）, 214
  in discovery services（在发现服务中）, 671
  in distributed garbage collection （在分布式无用单元收集中）, 196
  in Jini （在Jini中）, 197
  in Jini event service（在Jini事件服务中）, 207
Life cycle service（生命周期服务）, 848
lightweight directory access protocol（轻量级目录访问协议）, 394
lightweight RPC（轻量级RPC）, 251~253
Linda（Linda系统）, 755
linear-bounded arrival processes (LBAP)（线性限制的到达处理）, 733
linearizability（线性化能力）, 616
linearization（线性化）, 452
links（链接）, 9
link-state routing algorithms（链接状态路由算法）, 85
little-endian order（小序法排序）, 144
liveness property（活性）, 453
load balancing（负载平衡）, 44
load sharing（负载共享）, 231~232
local area network （局域网，参见network, local area）
location（位置）
  absolute（绝对）, 693
  authentication based upon（基于位置的认证）, 702
  physical（物理）, 694
  relative（相对）, 693
  semantic（语义）, 694
  sensing（传感）, 691~695
  stack（栈）, 695
location service （定位服务）, 195
location transparency（位置透明性）, 23, 329, 348
  in middleware（在中间件中）, 178
location-aware computing（位置敏感的计算）, 6
location-aware system（位置敏感系统，参见location, sensing）
lock manager（锁管理器）, 535
locking（加锁）
  in distributed transaction（在分布式事务中）, 578
locks（锁）, 530~544
  causing deadlock （引起死锁，参见deadlock）
  exclusive（排他）, 530
  granularity（粒度）, 531
  hierarchic（层次）, 543, 545
  implementation（实现）, 534
  in nested transaction （在嵌套事务中）, 537
  lock manager（锁管理器）, 534
  operation conflict rules（操作冲突规则）, 532
  promotion （提升）, 533
  read-write（读－写）, 532, 533
  read-write-commit（读－写－提交）, 542, 543
  shared（共享的）, 532
  strict two-phase（严格两阶段）, 531
  timeouts（超时）, 541
  two-phase locking（两阶段加锁）, 531
  two-version（两版本）, 542
logging （日志）, 590~593
logical clock（逻辑时钟）, 446
logical time （逻辑时间）, 52, 445~448
log-structured file storage（日志结构的文件存储, LFS）, 362
lost reply message（丢失应答消息）, 158
lost update（更新丢失）, 521, 533

M

Mach（Mach系统）, 259
MAN（参见network, metropolitan area）
man-in-the-middle attack （中间人攻击）, 269
marshalling（编码）, 145
masking failure （屏蔽故障）, 22
masquerading attack（伪装攻击）, 269
maximum transfer unit (MTU)（最大传输单元）, 79, 95, 114
maybe invocation semantics（或许调用语义）, 188
MD5 message digest algorithm（MD5消息摘要算法）, 301, 303
media synchronization （媒体同步）, 725
medium access control protocol（介质访问控制协议,

MAC）, 78, 114, 117, 119
message（消息）
destination（目的地）, 134~135
reply（应答）, 157
request（请求）, 157
message authentication code (MAC)（消息认证码）, 298
message digest（消息摘要）, 278
message passing（消息传递）, 132
message tampering attack（消息篡改攻击）, 269
metadata（元数据）, 328
metropolitan area network（城域网，参见 network, metropolitan area）,
microkernel（微内核）, 257~260
comparison with monolithic kernel（与整体内核比较）, 258
middleware（中间件）, 16, 32, 178
and heterogeneity（和异构性）, 179
operating system support for（操作系统支持）, 223
MIME type（多用途因特网邮件扩展类型）, 162
use in HTTP（在HTTP中使用）, 706
mix zone（混合区域）, 704
mixing（混合）, 704
mobile agent（移动代理）, 39
mobile code（移动代码）, 17, 19, 38, 61
security threats（安全威胁）, 270
mobile computing（移动计算）, 6~7, 658~717
origin of（起源）, 658
mobile device（移动设备）, 41
mobile phone（移动电话）, 658, 660, 661, 662, 664
and proximity（和接近度）, 693
and secure association（和安全关联）, 698
user agent profile（用户代理偏好）, 706
with camera（和照相机，参见camera phone）
mobileIP（移动IP）, 104~105
mobility（移动）
physical versus logical（物理和逻辑）, 662
transparency（透明性）, 24, 329
model（模型）
failure（故障）, 53~57
fundamental（基础）, 47~61
interaction（交互）, 48~53
security（安全性）, 57~61
mote（微生）, 664
MPEG video compression（MPEG 视频压缩）, 727, 728, 732
MTU（参见maximum transfer unit）
multicast（组播）, 484~498
atomic（原子）, 492
basic（基本的）, 486
causally ordered delivery（因果排序的发送）, 491, 496
FIFO-ordered delivery（FIFO排序的发送）, 490, 493
for discovery services（发现服务的）, 164
for event notifications（事件通知的）, 164
for fault tolerance（容错的）, 164, 618, 620
for highly available data（高可用数据的）, 164
for replicated data（复制数据的）, 168
group（组）, 484
operation（操作）, 164
ordered（排序的）, 490~498
reliable（可靠的）, 487~490
to overlapping groups（给重叠组的）, 498
totally ordered delivery（全排序的发送）, 491, 493
multicast group（组播组）, 165
multilateration（多时段定位法）, 692
multimedia（多媒体）, 722~747
admission control（任务控制）, 737~738
data compression（数据压缩）, 728
network applications（网络应用）, 124
play time（播放时间）, 74
resource bandwidth（资源带宽）, 723
stream（流）, 722~726, 727
stream adaptation（流自适应）, 740
stream burstiness（流猝发）, 733
typical bandwidths（典型带宽）, 727
web-based（基于web的）, 724
multiprocessor（多处理器）
distributed memory（分布式内存）, 751
shared memory（共享内存）, 225
Non-Uniform Memory Access (NUMA)（非一致内存访问）, 751
Munin, 775~777
mutual exclusion（互斥）, 471~479
between processes in a ring（环中进程之间的）, 474
by central server（中央服务器）, 473
Maekawa's algorithm（Maekawa算法）, 477
token（令牌）, 474
using multicast（使用组播）, 475~477

N

Nagle's algorithm（Nagle算法）, 107
name（名字）, 368, 370
component（成分）, 373
prefix（前缀）, 373
pure（纯的）, 368
unbound（未被绑定）, 372

name resolution（名字解析），368, 371, 375~376, 713
name service（名字服务），368~395
caching（高速缓冲存储），378, 381
CORBA naming service（CORBA名字服务），848~851
heterogeneity（异构性），375
navigation（导航），376~378
replication（复制），381
name space（命名空间），372
naming context（命名上下文），375
naming domain（命名域），374
Napster, 399, 402~404, 409
NAT（参见Network Addres Translation）
navigation（导航）
multicast（组播），377
server-controlled（服务器控制的），377
Near Field Communication（NFC）（近距离通信），674, 694
Needham-Schroeder protocol（Needham-Schroeder协议），306~307
negative acknowledgement（否定确认），488
Nemesis, 260
nested transaction（嵌套事务），528~530, 598
locking（加锁），537
recovery（恢复），598~599
two-phase commit protocol（两阶段提交协议），574~578
timeout actions（超时动作），578
network（网络）
ad hoc（自组织），119, 687
ATM(Asynchronous Transfer Mode)（异步传输模式），71, 74, 75, 78, 86, 113, 124~127
bridge（网桥），88
circuit switching（电路交换），75
congestion control（拥塞控制），85
CSMA/CA（具有冲突避免的载波侦听多路访问），120
CSMA/CD（具有冲突检测的载波侦听多路访问），114
data streaming（数据流），73
data transfer rate（数据传输率），67
delay-tolerant（容延迟），689
disruption-tolerant（容中断），689
frame relay（帧中继），75
gateway（网关），72
IEEE 802 standards（IEEE 802标准），112
IEEE 802.11（WiFi），113, 118~121, 317~319
IEEE 802.15.1（Bluetooth），113, 121~124
IEEE 802.15.4（ZigBee），113
IEEE 802.16（WiMAX），113
IEEE 802.3（Ethernet），88, 112, 113~118
IEEE 802.4（Token Bus），113
IEEE 802.5（Token Ring），112
Internet（因特网），80, 89~112
interplanetary（星球间），689
IP multicast（IP组播），89
IP routing（IP路由），96
latency（延迟，等待时间），67
layer（层），78
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），79
protocol（协议，参见protocol）
reliability requirements（可靠性需求），68
requirements（需求），67~69
router（路由器），88
routing（路由），71, 81~85, 96~99
scalability requirements（伸缩性需求），68
security requirements（安全性需求），68
TCP/IP, 89
total system bandwidth（系统总带宽），67
traffic analysis（流量分析），704
transport address（传输地址），80
tunnelling（隧道法），89
Ultra Wide Band（超宽带），694
wide area（广域），71
Network Address Translation（NAT）（网络地址转换），99
network computer（网络计算机），39
network discovery service（网络发现服务），670
Network File System（NFS）（网络文件系统），331, 337~349, 404, 425, 426
Automounter（自动装载器），343
benchmarks（基准程序），347
enhancements（增强），359
hard and soft mounting（硬安装和软安装），342
Kerberos authentication（Kerberos认证），346
mount service（安装服务），341
performance（性能），347
Spritely NFS, 359
virtual file system(VFS)（虚拟文件系统），338
v-node（v节点），339
WebNFS, 360
Network Information Service (NIS)（网络信息服务），619
network operating system（网络操作系统），222

network partition（网络分区），469, 646~647
  primary（主的），611
  virtual（虚拟的），650~653
network phone application（网络电话应用），724
Network Time Protocol（网络时间协议），441~444
network transparency（网络透明性），24
NFS（参见Network File System）
NIS（参见Network Information Service）
NNTP（网络新闻传输协议），90
nomadic computing（游牧计算），6, 710
non-blocking send（非阻塞发送），134
nonce（当前时间），306
non-preemptive scheduling（非抢占性调度），242
non-repudiation（不可抵赖），272, 296
notification（通知），201~203
NQNFS (Not Quite NFS), 360
NTP（参见Network Time Protocol）

O

object（对象），42
  activation（激活），194
  active（主动），194
  distributed（分布式的），183
    model（模型），184
  factory（工厂），193
  instantiation（实例化），183
  interface（接口），42
  location（位置），195
  model（模型），182
    CORBA, 829
  no instantiation in web services（在web服务中没有实例化），787, 794
  passive（被动的），194
  persistent（持久的），194
  protection（保护），57
  reference（引用），182
  remote（远程），184
  remote reference（远程引用），185
object adapter（对象适配器），836
object request broker (ORB)（对象请求代理），828
object serialization, in Java（对象序列化，在Java中），148~150
OceanStore, 422~426
OMG (Object Management Group)（对象管理工作组），828
omission failure（遗漏故障），53
  communication（通信），54
  process（进程），53
one-copy serializability（单拷贝串行化），641
one-way function（单向函数），286
one-way hash function（单向散列函数），300
open distributed systems（开放的分布式系统），18
Open Mobile Alliance（开放移动联盟），706
open shortest path first (OSPF)（开放最短路径优先算法），97
open system（开放系统），18, 256
open systems interconnection（开放系统互连，参见OSI Reference Model）
openness（开放性），17~18, 330
operating system（操作系统）
  architecture（体系结构），256~260
  communication and invocation support（通信和调用支持），245~256
  policy and mechanism（策略和机制），256
  processes and threads support（进程和线程支持），228~245
operational transformation（操作变换），632
optimistic concurrency control（乐观并发控制），545~549
  backward validation（后向有效性），547
  comparison of forward and backward validation（前向有效性与后向有效性的比较），549
  forward validation（前向有效性），548
  in distributed transaction（在分布式事务中），580
  starvation（饥饿），549
  update phase（更新阶段），546
  validation（有效性），546
  working phase（工作阶段），546
Orca, 753
OSI Reference Model（OSI参考模型），78
output parameters（输出参数），180

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（页失配），233
passive object（被动对象），194
passive replication（被动复制，参见primary-backup replication）
Pastry routing overlay（Pastry路由覆盖），399, 409, 410~418
peer-to-peer（对等），35~36, 398~430
  and copyright ownership（版权拥有者），403
  middleware（中间件），399, 404~406
  routing overlay（路由覆盖），406~410
performance issues（性能问题），43

performance transparency（性能透明性），24, 329
persistent connection （一致性连接），161, 251
persistent object （持久对象），194
  in CORBA persistent state service（在CORBA持久状态服务中），847
persistent object store（持久对象存储）
  comparison with file service（与文件服务比较），325
  persistent Java（持久Java），194~195, 326
personal digital assistant（个人数字助理, PDA），658
personal server（个人服务器），665
Petal distributed virtual disk system（Petal 分布式虚拟磁盘系统），364
PGP（参见Pretty Good privacy）
phantom deadlock（假死锁），583
physical hyperlink（物理超链接），710, 712~714
physical layer （物理层），77, 78
physically constrained channel（物理受限通道），674, 699~702
pipelined RAM（管道RAM），777
plaintext（明文），286
Plan（计划，规划），9, 375
platform（平台），32, 223
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
port mapper（端口映射器），200
POTS (plain old telephone system)（老式电话系统），75
PPP（点对点协议），78, 89, 91
precedence graph （优先级图），647
preemptive scheduling（抢占性调度），242
premature write （过早写入），527
presentation layer（表示层），78
Pretty Good Privacy (PGP)（良好隐私（一种加密电子邮件的方案））, 304
primary copy（主副本），348
primary-backup replication （主备份复制），618~620
principal（主体），57
privacy（私密性），666, 695, 696, 703~704
  proxy（代理），704
process（进程），228~245
  correct（正确的），470
  creation（生成），231~234
  creation cost（创建开销），238
  multi-threaded（多线程的），228
  threats to （威胁），58
  user-level（用户级），227
processor consistency （处理器一致性），777
promise（承诺），255
protection（保护），226~227
  and type-safe language（类型安全的语言），227
  by kernel （内核的），227
protection domain（保护域），281
protocol（协议），75~81
  application layer（应用层），76, 78
  ARP（地址解析协议），95
  data link layer（数据链路层），78
  dynamic composition（动态合成），246
  FTP （文件传输协议），78, 80, 90, 110, 111
  HTTP（超文本传输协议），78, 80, 90, 109
  internetwork layer（互联网络层），79
  IP（因特网协议），78, 79, 94~105
  IPv4（第4版因特网协议），92
  IPv6（第6版因特网协议），74, 89, 101~104
  layers（层），76
  mobileIP, 104~105
  network layer（网络层），78
  NNTP（网络新闻传输协议），90
  operating system support for （操作系统支持），246
  physical layer（物理层），77, 78
  POP （从邮件服务器获得邮件的协议），90
  PPP（点对点协议），78, 89, 91
  presentation layer（表示层），78
  session layer （会话层），78
  SMTP（简单邮件传输协议），78, 90
  stack（栈），77, 246
  suite （组），77
  TCP （传输控制协议），78, 105~108
  TCP/IP, 89
  transport（传输），76
  transport layer（传输层），78
  UDP（用户数据报协议），78, 90, 105
proxy（代理），191
  dynamic（动态），797
  in CORBA（在CORBA中），837
  in web services（在web服务中），789, 797, 798
proxy server （代理服务器），38
pseudonym（假名），703
public key （公钥），275
pubic-key infrastructure（公钥基础设施），301
public-key certificate （公钥证书），278, 280
public-key cryptography （公钥密码学），293~295
publish-subscribe（出版－订阅），201

Q

QoS（参见quality of service）

quality of service（服务质量），44
admission control（任务控制），730
management（管理），722, 728~738
negotiation（协商），730, 731~736
parameters（参数），731
query（查询）
distributed processing（分布式处理），689
spatio-temporal（时空），695
quorum consensus（法定数共识），642, 648~650

R

Radio Frequency IDentification（无线射频识别，参见RFID）
randomization（随机），509
RC4 stream cipher algorithm（RC4流密码算法），292
reachability（可达性），452
Real Time Transport Protocol (RTP)（实时传输协议），74
real-time network（实时网络），118
real-time scheduling（实时调度），739
receive-omission failures（接收遗漏故障），55
recovery（恢复），526~528, 589~599
cascading abort（连锁放弃），527
dirty read（脏数据读取），526
from abort（从放弃中），526
intentions list（意图列表），590, 595
logging（日志），590~593
nested transactions（嵌套事务），598~599
of two-phase commit protocol（两阶段提交协议的），596~599
premature write（过早写入），527
shadow versions（影子版本），594~595
strict executions（严格执行），428
transaction status（事务状态），590, 595
recovery file（恢复文件），589~599
reorganization（重组织），593, 598
recovery from failure（故障恢复），22
recovery manager（恢复管理器），589
redundancy（冗余），22
redundant arrays of inexpensive disks (RAID)（廉价磁盘冗余阵列（RAID））, 362
reflection（反射）
in Java（在Java中），149
in Java remote method invocation（在Java远程方法调用中），215
region（区域，参见address space）
registering interest（注册兴趣），201
release consistency（释放一致性），773~775
reliable channel（可靠的通道），469
reliable communication（可靠的通信），56
in SOAP（在SOAP中），793
integrity（完整性），56
validity（有效性），56
reliable multicast（可靠的组播），168, 487~490
remote interface（远程接口），180, 185
in Java remote method invocation（在Java远程方法调用中），209
remote method invocation（远程方法调用），8, 184
communication module（通信模块），190
CORBA, 829~836
dispatcher（调度程序），191
duplicate filtering（重复过滤），187
implementation（实现），190~195
Java case study（Java实例研究），208~216
null（空），247
parameter and result passing in Java（在Java中的参数和结果传递），209
performance of（性能，参见invocation mechanism, performance of）
proxy（代理），191
remote reference module（远程引用模块），190
retransmission of replies（应答重传），187
retry request message（重发请求消息），187
semantics（语义），187
skeleton（骨架），191
transparency（透明度），189
remote object（远程对象），184
instantiation（实例化），186
remote object reference（远程对象引用），154, 185
compared with URI（与URI比较），794
in CORBA（在CORBA中），844
remote object table（远程对象表），190
remote procedure call（远程过程调用），197~201
exchange protocols（交换协议），158
lightweight（轻量级），251~253
null（空），247
performance of（性能，参见invocation mechanism, performance of）
queued（排队的），255
remote reference module（远程引用模块），190
replaying attack（重发攻击），269
replica manager（副本管理器），607
replication（复制），45, 603~653
active（主动的），620~622
available copies（可用的拷贝），642, 644~646
with validation（带验证的），647~648

of files（文件的），330
primary-backup（主备份），618~620
quorum consensus（法定数共识），642, 648~650
transactional（事务性的），641~653
transparency（透明的），24, 605
virtual partition（虚拟分区），642, 650~653
reply message（应答消息），157
request message（请求消息），157
request-reply protocol（请求－应答协议），156~164
doOperation, 156
duplicate request message（重复的请求消息），158
failure model（故障模型），158
getRequest, 156
history of server operations（服务器操作的历史），159
lost reply messages（丢失应答消息），158
sendReply, 156
timeout（超时），158
use of TCP（TCP的使用），160
resolver（解析器），713
resource（资源），2
invocation upon（调用），224
sharing（共享），7~8, 398
resource management (for multimedia)（资源管理（多媒体）），738~739
Resource Reservation Protocol (RSVP)（资源保留协议），74, 736
responsiveness（响应能力），43
REST, 786, 788
resurrecting duckling protocol（复活鸭协议），701
RFC, 18
RFID（无线射频识别），684, 694, 703, 713
RIP-1（路由信息协议1），85, 96
RIP-2（路由信息协议2），96
RMI（参见remote method invocation）
RMIRegistry, 210
router（路由器），86, 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），159
RR（请求－应答协议），159
RRA（请求－应答－确认协议），159
RSA public key encryption algorithm（RSA公钥加密算法），287, 293~295
RSVP（参见Resource Reservation Protocol）
run（运行），452

S

safety property（安全性），453
sandbox model of protection（保护的沙盒模型），270
satellite navigation（卫星导航），691
参见Global Positioning System（GPS），255
scalability（可伸缩性），19~21, 329
scaling transparency（伸缩透明性），24, 329
scheduler activation（调度器激活），244
scope consistency（范围一致性），778
secret key（密钥），275
secure channel（安全通道），60
secure digest function（安全摘要函数），278
secure hash function（安全散列函数），297, 400
Secure Sockets Layer（安全套接字层，参见Transport Layer Security）
secure spontaneous device association（安全自发设备关联），698~702
security（安全性），18, 19
"Alice and Bob" names for protagonists（角色名字，如"Alice 和 Bob"），268
denial of service attack（拒绝服务攻击），269
design guidelines（设计原则），273
eavesdropping attack（窃听攻击），269
in CORBA security service（在CORBA安全服务中），854~855
in Java（在Java中），270
information leakage models（信息泄露模型），270
man-in-the-middle attack（中间人攻击），269
masquerading attack（伪装攻击），269
message tampering attack（消息篡改攻击），269
replaying attack（重发攻击），269
threats from mobile code（对移动代码的威胁），270
threats: leakage, tampering, vandalism（威胁：泄露、篡改、恶意破坏），268
Transport Layer Security (TLS)（传输层安全，参见Secure Sockets Layer），313~317
trusted computing base（可信赖计算基），274
XML security（XML安全），807~811
security mechanism（安全机制），266
security model（安全模型），57~61
security policy（安全策略），266
Semantic Web（语义Web），683
send operation, non-blocking（发送操作，非阻塞的），134
sender order（发送方顺序），135
send-omission failures（发送遗漏故障），55
sensor（传感器），663, 683~696
error mode（错误模式），684

fusion（融合），685
location（位置），691-695
network, wireless（网络，无线），687~691
sequencer（顺序者），493
sequential consistency（顺序一致性），617
of distributed shared memory（分布式共享内存的），759
serial equivalence（串行等价），519, 523
serialization（串行化），148
servant（伺服器），191, 192, 193, 211, 795, 832, 837
servant classes（伺服器类），211, 832, 837
server（服务器），8
multi-threaded（多线程的），234
multi-threading architecture（多线程体系结构），235
personal（个人的），665
throughput（吞吐量），234
server port（服务器端口），136
server stub procedure（服务器存根过程），198
serverless file system (xFS)（无服务器文件系统），363
service（服务），8
creation and deletion（创建和删除），819
fault-tolerant（容错），605, 615~622
highly available（高可用的），622~641
service discovery（服务发现，参见discovery service）
service interface（服务接口），180, 197, 796
servlet（servlet程序），240
servlet container（servlet容器），796
session Initiation Protocol（会话初始化协议），725
session key（会话密钥），277
session layer（会话层），78
SETI @ home project（SETI @ home项目），401
SHA secure hash algorithm（SHA安全散列算法），301, 303
SHA-1 secure hash algorithm（SHA-1安全散列算法），408, 410, 421, 427
shadow versions（影子版本），594~595
shared whiteboard（共享的白板）
CORBA IDL interface（CORBA IDL接口），830
dynamic invocation（动态调用），192
implementation in CORBA（CORBA中的实现），832
implemented in Java RMI（Java RMI中的实现），208~214
implemented in web services（web服务中的实现），795
signature of method（方法基调），182
Simple Public-key Infrastructure (SPKI)（简单公钥基础设施），302
SIP（参见Session Initiation Protocol）
skeleton（骨架），191
dynamic（动态），193
dynamic in CORBA（CORBA中的动态），839
in CORBA（CORBA中的），837
in web services（在web服务中），798
not needed with generic dispatcher（不需要通用调度器），215
smart dust（参见mote）
smart space（智能空间），662
SMTP（简单邮件传输协议），78, 90
SOAP（SOAP协议），786, 789~794
addressing and routing（寻址和路由），793
and firewalls（和防火墙），794
envelope（信封），790
header（头部），791
Java implementation（Java实现），798
message transport（消息传输），792
message（消息），790
reliable communication（可靠通信），793
specification（规约），789
use of HTTP（HTTP的使用），792
with Java（带Java的），795~799
socket（套接字），135
address（地址），168
bind（绑定），169
close（关闭），169
connect（连接），143
datagram communication（数据报通信），169
pair（对），168
stream communication（流通信），170
system call（系统调用），168, 169, 171
socket address（套接字地址），168
sockets（套接字），135~136
soft state（软态），682
software interrupt（软中断），237
Solaris（Solaris系统）
lightweight process（轻量级进程），243
spatio-temporal query（时空查询），695
Speakeasy, 681
speaks for relation (in security)（关系证明（在安全性中）），284
SPIN, 259
SPKI（参见Simple Public-key Infrastructure）
spontaneous association（自发关联），665
spontaneous interoperation（自发互操作），41~42, 665~666
spontaneous networking（自发网络），661
discovery service（发现服务），386
Spring naming service（Spring名字服务），375

Spritely NFS, 359
Squirrel web cache（Squirrel web缓存）, 420~422
SSL（参见Transport Layer Security）
starvation（饥饿）, 472, 549
state machine（状态机）, 607
state transfer（状态转换）, 613
stateless server（无状态服务器）, 334
stream cipher（流密码）, 288~289
strict executions（严格执行）, 528
stub procedure（存根过程）, 197
  in CORBA（在CORBA中）, 837
subscribe（订阅）, 201
subsystem（子系统）, 258
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）, 198~201, 338
  external data representation（外部数据表示）, 200
  interface definition language（接口定义语言）, 198
  portmapper（端口映射器）, 200
  rpcgen（rpcgen）, 198
supervisor（管理器）, 226
supervisor mode（管理模式）, 227
Switched Ethernet（交换式以太网）, 118
symmetric cryptography（对称密码学）, 286
symmetric processing architecture（对称处理体系结构）, 225
synchronization memory accesses（同步内存访问）, 772
synchronization of clocks（时钟同步）, 437~444
  Berkeley algorithm（Berkeley算法）, 441
  Cristian's algorithm（Cristian算法）, 439~441
  external（外部的）, 437
  in a synchronous system（在同步系统中）, 439
  internal（内部的）, 437
  Network Time Protocol（网络时间协议）, 441~444
synchronization, of server operations（同步，服务器操作的）, 516
synchronous communication（同步通信）, 134
synchronous distributed system（同步分布式系统）, 50, 439, 463, 468, 471, 498, 503
system call trap（系统调用陷阱）, 227

T

tag（标记，参见automatic identification）
Tapestry routing overlay（Tapestry路由覆盖）, 399, 409, 418~419
TCP（传输控制协议）, 78, 105~108, 140~144
  and request-reply protocols（和请求－应答协议）, 160, 250
  API, 140
  failure model（故障模型）, 141
  Java API, 142~144
  UNIX API, 170
TCP/IP（参见protocol, TCP/IP）
TEA (Tiny Encryption Algorithm)（微加密算法）, 290~291, 303
termination（终止性）
  of consensus and related problems（在共识和相关问题中）, 500~501
termination detection（终止检测）, 449
thin client（瘦客户）, 40
Third generation mobile phone network（第三代移动电话网络）, 72
thrashing（系统颠簸）, 763
thread（线程）, 134, 228, 234~245
  blocking receive（阻塞型接收）, 134
  C, 239
  in client（在客户端的）, 236
  comparison with process（与进程的比较）, 236
  creation cost（创建花费）, 238
  implementation（实现）, 242~245
  in server（在服务器端的）, 234
  Java, 239~242
  kernel-level（内核级）, 243
  on multiprocessor（多处理器上的）, 236
  multi-threading architecture（多线程体系结构）, 235
  POSIX, 239
  programming（编程）, 239~242
  scheduling（调度）, 242
  switching（切换）, 238
  synchronization（同步）, 241
  worker（工作线程）, 235
throughput（吞吐量）, 44
throw exception（抛出异常）, 183
Ticket Granting Service (TGS)（票证授予服务）, 309
ticket, of authentication（票证，认证的）, 276
Tiger video file server（Tiger视频文件服务器）, 742
time（时间）, 433~448
  logical（逻辑的）, 445~448
time-based data streams（基于时间的数据流）, 727
time-critical data（实时数据）, 44
timeouts（超时）, 53
timestamp（时间戳）
  Lamport, 446
  vector（向量）, 447
timestamp ordering（时间戳排序）, 549~556

in distributed transaction（在分布式事务中），579
multiversion （多版本），554~556
operation conflicts（操作冲突），551
read rule（读规则），551
write rule（写规则），551
timing failure（时序故障），55
TLS（参见Transport Layer Security）
Token Bus network（令牌总线网），113
Token Ring network（令牌环网），112
tolerating failure（容错），22
total ordering（全排序），435, 447
of multicast delivery（组播传递的），491
of request handling（请求处理的），608
totally ordered multicast（全排序组播），168
tracking（跟踪），691
and privacy（和私密性），695, 703
traffic analysis（流量分析），704
traffic shaping (for multimedia data)（流量调整（对多媒体数据）），734
transaction （事务），517~530
abort（放弃），520
abortTransaction （abortTransaction类），520
ACID properties（ACID属性），519
closeTransaction （closeTransaction类），520
commit（提交），520
concurrency control（并发控制，参见 concurrency control）
failure model（故障模型），517
in CORBA transaction service （在CORBA的事务服务中），847
in web services（在web服务中），812
openTransaction （openTransaction类），520
recovery（恢复，参见recovery）
serial equivalence（串行等价），523
with replicated data（复制数据的），641~653
transaction status（事务状态），590, 595
transcoding（转码），706
transparency（透明性），23~25
access（访问），23, 329, 348
concurrency（并发），23
failure（故障），24
in remote method invocation （在远程方法调用中），189
location （位置），23, 329, 348
mobility（移动），24, 329
network（网络），24
performance（性能），24, 329
replication （复制），24, 605
scaling（伸缩），24, 329
transport address（传输地址），80
transport layer（传输层），78
Transport Layer Security（传输层安全，TLS），90, 313~317
transport protocol（传输协议），76
trap-door function （陷门函数），286
triple-DES encryption algorithm（三重DES加密算法），303
trust（信任），666, 696, 701
trusted computing base （可信赖计算基），274, 666
trusted third party（可信任第三方），696, 698, 699
tunnelling（隧道法），89
tuple space （元组空间）
as distributed shared memory（作为分布式共享内存），755~756
compared to event system（与事件系统比较），679
in volatile system（在易变系统中），678~679
two-phase commit protocol （两阶段提交协议），570~573
nested transaction （嵌套事务），574~578
flat commit（平面提交），577
hierarchic commit（层次提交），576
performance（性能），573
recovery （恢复），596~599
timeout actions（超时动作），571

U

ubiquitous computing（无处不在计算），6, 658~717
origin of（起源），659
UDDI（参见universal directory and discovery service）
UDP（用户数据报协议），78, 90, 105, 136~139
failure model （故障模型），137
for request-reply communication（请求－应答通信），160, 250
Java API, 138~139
UNIX API, 169
use of （使用），137
UFID （参见unique file identifier）
Ultra Wide Band（UWB）（超宽带），694
uniform memory consistency model （统一内存一致性模型），777
uniform property （一致性），490
Uniform Resource Identifier（统一资源标识符），370
compared with remote object reference（与远程对象引用比较），794
in web services（在web服务中），784
Uniform Resource Locator（统一资源定位器），11~13, 370
Uniform Resource Name（统一资源名称），370
unique file identifier (UFID)（文件唯一标识符），333,

353
universal directory and discovery service (UDDI) (统一目录和发现服务), 805~807
Universal Plug and Play (UPnP) (通用即插即用), 668
Universal Transfer Format (通用传输格式), 149
UNIX
  i-node (i-结点), 339
  signal (信号), 237
  system call (系统调用)
    accept, 171
    bind, 170
    connect, 171
    exec, 231
    fork, 231, 232
    listen, 171
    read, 171
    recvfrom, 170
    sendto, 170
    socket, 169, 171
    write, 171
unmarshalling (解码), 145
unreliable multicast (不可靠组播), 166
upcall (上调), 244
update semantics (更新语义), 330, 335
URI (参见Uniform Resource Indentifer)
URL (参见Uniform Resource Locator)
URN (参见Uniform Resource Name)
user mode (用户模式), 227
UTC (参见Coordinated Universal Time)
UTF (参见Universal Transfer Format)

V

V system (V系统)
  remote execution (远程执行), 231
  support for groups (对组的支持), 614
validity (有效性)
  of multicast delivery (组播传递的), 487
  of reliable communication (可靠通信的), 56
vector clock (向量时钟), 447
vector timestamp (向量时间戳), 447
  comparison (比较), 448
  merging operation (合并操作), 447
Verisign Corporation (Verisign公司), 302
videoconferencing (视频会议), 724, 727
  CU-SeeMe application (CU-SeeMe 应用), 724
  iChat AV application (iChat AV应用), 724
  NetMeeting application (网络会议应用), 724
video-on-demand service (视频点播服务), 724
view, of group (视图, 组的, 参见 group, view)
view-synchronous group communication (视图同步的组通信), 612~614
virtual channel (in ATM networks) (虚通道 (在ATM网络中)), 125
virtual circuit (虚电路), 81
virtual file system (VFS) (虚拟文件系统, 参见Network File System (NFS))
virtual machine (虚拟机), 17
Virtual Network Computer (虚拟网络计算机), 41
virtual partition (虚拟分区), 650~653
virtual path (in ATM networks) (虚拟路径 (在ATM网络中)), 125
virtual private network (VPN) (虚拟私网), 69, 111
virtual processor (虚拟处理器), 243
virtualization (虚拟化), 259
virtually synchronous group communication (虚拟同步组通信), 614
VNC (参见Virtual Network Computer)
Voice over IP (通过IP的语音传输), 725
  Skype application (Skype应用), 725
  Vonage application (Vonage应用), 725
VOIP (参见Voice over IP)
volatile system (易变系统), 661~666
voting (投票), 477
VPN (参见virtual private network)

W

WAN (WAN, 参见network, wide area)
weak consistency (弱一致性), 760, 778
wearable computing (可穿戴计算), 660
Web, 9~15
  caching (高速缓存), 45, 420~422
  proxy server (代理服务器), 38
  semantic (语义的), 683
web presence (web存在), 711
web services (web服务), 14~15, 786~800
  case study the Grid (网格的案例研究), 814~823
  choreography (编排), 813
  communication patterns (通信模式), 786
  compared with CORBA (与CORBA比较), 799, 800
  compared with distributed object model (与分布式对象模型比较), 794
  coordination (协调), 812~814
  directory service (目录服务), 805~807
  dispatcher (分发器), 798
  dynamic invocation (动态调用), 789
  dynamic invocation interface (动态调用接口), 798

dynamic proxy（动态代理），797
infrastructure（基础设施），785
integration with CORBA（与CORBA集成），846
Java client program（Java客户程序），797
Java server program（Java服务器程序），796
model（模型），794
no servants（没有伺服器），795
programming model（编程模型），787
proxy（代理），789, 797
REST, 786, 788
service description（服务描述），800~805
sevice interface in Java（Java中的服务接口），796
servlet container（servlet容器），796
skeleton（骨架），798
SOAP, 786, 789~794
SOAP with Java（带Java的SOAP），795~799
Uniform Resource Identifier（统一资源标识符），784
web services description language（WSDL）（web服务描述语言），800~805
binding（绑定），804
concrete part（具体部分），804
interface（接口），802
main elements（主要元素），801
operations or messages（操作或消息），802
service（服务），805
web-based multimedia（基于web的多媒体），724
WebNFS, 360
WEP WiFi security（WEP WiFi安全），317~319
wide area network（广域网，参见network, wide area）
WiFi network（WiFi网络），113, 118~121, 317~319
WiMAX network（WiMAX网络），113
WLAN (wireless local area network)（WLAN(无线局域网)），72
WMAN (wireless metropolitan area network)（WMAN(无线城域网)），72
WWAN (wirekess wide area network)（WMAN（无线广域网），72
WorldWide Web（万维网，参见Web）
World-Wide Telescope
a grid application（一种网格应用），815
WSDL（参见web services description language）

X

X.500 directory service（X.500目录服务），390~394
directory information tree（目录信息树），391
LDAP, 394
X.509 certificate（X.509证书），301
X-11 window system（X-11窗口系统），40
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），363
XML（Extensible Markup Language）（XML（可扩展标记语言）），15, 150~154
canonical XML（单调XML），810
elements and attributes（元素和属性），151
for interoperation（用于互操作），683
namespaces（名字空间），153
parsing（解析），152
schemas（模式），153
security（安全），807~811
digital signature（数字签名），810
encryption（加密），811
requirements（需求），808
XSLT（eXtensible Style Language Transformations）（XSLT（可扩展样式语言转换）），706

Z

zero configuration networking（零配置联网），667
ZigBee network（zigBee网络），113