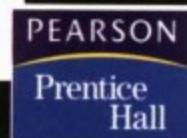


世界著名计算机教材精选



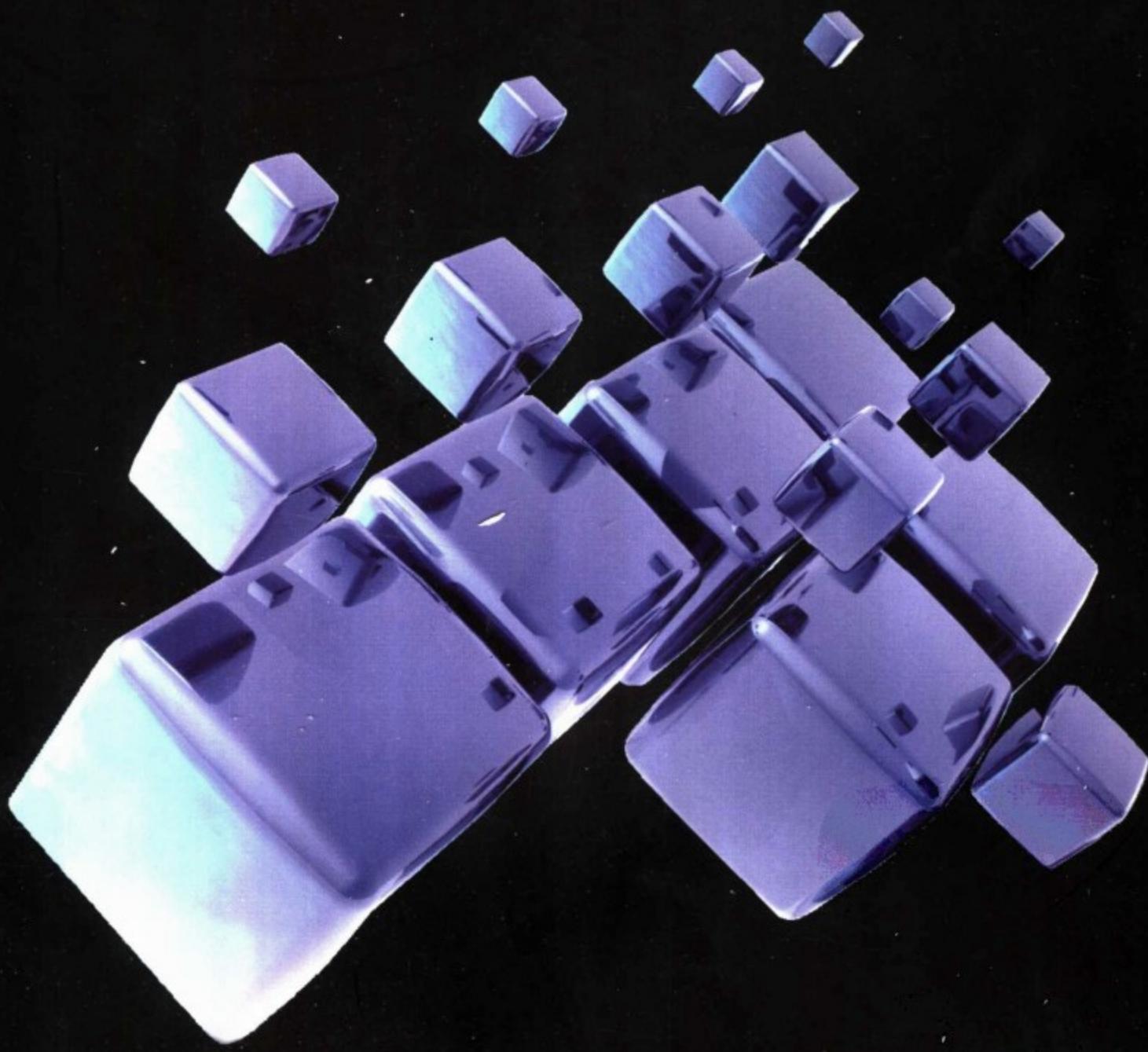
分布式系统

原理与范型

Andrew S. Tanenbaum
Maarten van Steen

著
译

杨剑峰 常晓波 李敏



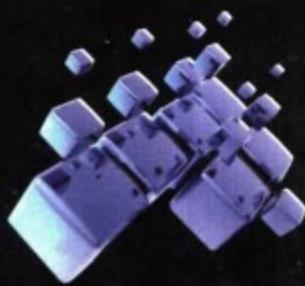
DISTRIBUTED SYSTEMS

Principles and Paradigms



清华大学出版社

分布式系统原理与范型



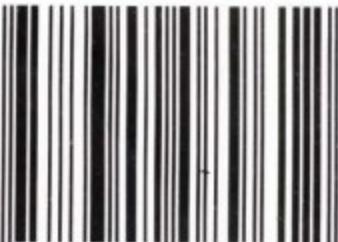
本书是Tanenbaum先生对所著的《分布式操作系统》的升级更新，是分布式系统的权威教材。全书分为两部分：原理和范型。第一部分详细讨论了分布式系统的原理、概念和技术，其中包括通信、进程、命名、同步、一致性和复制、容错以及安全。第二部分给出了一些实际的分布式系统：基于对象的分布式系统、分布式文件系统、基于文档的分布式系统以及基于协作的分布式系统，介绍了一些实际系统的设计思想和实现技术。全书结构清晰，内容全面经典，系统性与先进性并茂。

本书适用对象广泛。对于学习分布式计算的本科生和研究生，本书是优选教材。对于从事分布式计算研究和工程应用的科研人员和工程技术人员，本书也是一本优秀的基础性读物。

世界著名计算机教材精选

- 计算机网络（第4版）
Computer Networks (Fourth Edition)
- 数据结构 C++语言描述
Data Structures with C++
- 计算机组织与结构：性能设计（第4版）
Computer Organization and Architecture:
Design for Performance (Fourth Edition)
- 数据库系统基础教程
A First Course in Database Systems
- 面向对象系统分析与设计
Object-Oriented Systems Analysis and Design
- 计算理论基础（第2版）
Elements of the Theory of Computation (Second Edition)
- 多媒体技术：计算、通信和应用
Multimedia: Computing, Communications & Applications
- 因特网和万维网的基本原理与技术
In-line/On-line:Fundamentals of the Internet
and the World Wide Web
- TCP/IP协议族
TCP/IP Protocol Suite
- 通信网基本概念与主体结构
Communication Networks: Fundamental Concepts and Key
Architectures
- 计算机组成和设计（硬件/软件接口）
Computer Organization & Design(The Hardware/Software Interface)
- 分布式系统原理与范型
Distributed Systems Principles and Paradigms
- 数据结构与抽象（Java语言版）
Data Structures and Abstractions with Java
- 安腾体系结构：理解64位处理器和EPIC原理
Itanium® Architecture for Programmers:Understanding 64-Bit
Processors and EPIC Principles

ISBN 7-302-08961-2



9 787302 089612 >

定价：68.00元

PEARSON
Prentice
Hall

<http://www.pearsoned.com>

世界著名计算机教材精选

分布式系统原理与范型

Andrew S. Tanenbaum, Maarten van Steen 著

杨剑峰 常晓波 李 敏 译

清华大学出版社
北京



Simplified Chinese edition copyright © 2004 by PEARSON EDUCATION ASIA LIMITED and TSINGHUA UNIVERSITY PRESS.

Original English language title from Proprietor's edition of the Work.

Original English language title: Distributed Systems: Principles and Paradigms, by Andrew S. Tanenbaum and Maarten van Steen. Copyright © 2002

EISBN: 0-13-088893-1

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).

本书中文简体翻译版由 Prentice Hall 授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。

北京市版权局著作权合同登记号 图字: 01-2003-0557

版权所有, 翻印必究。举报电话: 010-62782989 13901104297 13801310933

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签, 无标签者不得销售。

图书在版编目(CIP)数据

分布式系统原理与范型/特南鲍姆(Tanenbaum, A. S.), 范施特恩(van Steen, M.)著; 杨剑峰等译. —北京: 清华大学出版社, 2004. 9

(世界著名计算机教材精选)

书名原文: Distributed Systems: Principles and Paradigms

ISBN 7-302-08961-2

I. 分… II. ①特… ②范… ③杨… III. 分布式操作系统—教材 IV. TP316. 4

中国版本图书馆 CIP 数据核字(2004)第 063337 号

出版者: 清华大学出版社

<http://www.tup.com.cn>

社总机: 010-62770175

地址: 北京清华大学学研大厦

邮 编: 100084

客户服务: 010-62776969

责任编辑: 袁勤勇

印 刷 者: 北京四季青印刷厂

装 订 者: 北京市密云县京文制本装订厂

发 行 者: 新华书店总店北京发行所

开 本: 185×260 印张: 39.75 字数: 915 千字

版 次: 2004 年 9 月第 1 版 2004 年 9 月第 1 次印刷

书 号: ISBN 7-302-08961-2/TP·6340

印 数: 1~4000

定 价: 68.00 元

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题, 请与清华大学出版社出版部联系调换。联系电话: (010)62770175-3103 或 (010)62795704

译者序

随着计算机网络,特别是 Internet 的迅猛发展,传统的信息系统概念发生了巨大的变化,这些变化突出地表现在信息的存储、传递、发布以及获取方式所发生的革命性变革。与此同时,基于网络的分布式信息系统在各个领域得到了广泛的应用,在整个社会生活中正发挥着日益突出的作用。Internet 已经越来越多地成为构建信息系统的一个关键组成部分。如何在更为广域和异构的计算环境中有效地发布和获取信息,已成为亟待解决的问题。分布式系统正是解决了上述问题。现在分布式系统的研究、应用日益广泛深入,分布式系统的学习也成为计算机及相关专业必不可少的教学环节。

本书是 Tanenbaum 先生在所著的《分布式操作系统》的基础上,总结了分布式系统方面的最新进展,重新撰写的力作,是分布式系统的权威教材。本书循序渐进地、全面地、深入地讲解了分布式系统的原理,并列出了大量的范型。本书的结构分为两部分:原理和范型。第一部分(第 1~8 章)详细讨论了分布式系统的原理、概念和技术,其中包括通信、进程、命名、同步、一致性和复制、容错以及安全。第二部分(第 9~12 章)给出了一些实际的分布式系统,即基于对象的分布式系统、分布式文件系统、基于文档的分布式系统以及基于协作的分布式系统,介绍了一些实际系统的设计思想和实现技术。全书结构清晰,内容全面经典,系统性与先进性并茂。

本书的目标读者是计算机及相关专业的高年级学生或研究生。从事分布式计算研究和工程应用的科研人员和工程技术人员也会从本书中受益匪浅。

本书是多人共同努力的成果,参与本书翻译、审稿、录排的人员包括:杨剑峰、常晓波、梁金昆、张丽萍、汪青青、朱志博、李敏、李静、李娟、张颖、朱剑平、刘颖、吴东升、杨战伟、郭宁宁、李楠、聂晶、刘恒、刘敏、刘洋、吕喜熹、马睿倩等。全书由杨剑峰、常晓波和李敏负责统稿。

限于译者水平,难免有错误和疏漏之处,恳请读者不吝指正。希望这本书能成为您工作的好帮手。

杨剑峰 常晓波
2004 年 5 月

前　　言

本书的出发点是对 Distributed Operating Systems 一书进行再版修订,但笔者很快就发现自 1995 年以来很多技术发生了改变,要完全体现出这些变化,仅仅对该书进行修订是不够的,而是需要写一本全新的书。因此,这本新书有了一个新的标题:《分布式系统原理和范型》。标题的改变体现了对重点的调整。虽然我们仍然讨论一些操作系统的问题,但现在这本书还从更广泛的意义上研究分布式系统。例如,WWW 作为已建立的最大的分布式系统,在 Distributed Operating Systems 一书中完全没有提到,因为它并不是一个操作系统。而在本书中,它几乎占去整整一章。

本书分为两部分:原理和范型。第 1 章是对主题的总体介绍。接下来的第 2~8 章分别讨论我们认为最重要的原理:通信、进程、命名、同步、一致性和复制、容错以及安全性。

实际的分布式系统通常是围绕一些范型来组织的,例如“所有事物都是文件”。接下来的第 9~12 章分别介绍一个不同的范型,并描述使用该范型的一些重要系统。涉及到的范型包括基于对象的系统、分布式文件系统、基于文档的系统以及基于协作的系统。

第 13 章包含一份附有说明的参考书目,可供该主题的进一步学习使用,还包含本书中引用的著作列表。

本书是作为计算机科学的大学高年级学生或研究生课程而编写的。因此,本书有一个 Web 站点,站点中以各种格式放置了本书中用到的 PowerPoint 表和图。要访问该站点,在 <http://www.prenhall.com/tanenbaum> 页面上点击本书标题即可。将本书作为教材使用的教授可以通过联系当地的 Prentice Hall 代理机构得到一本习题解答手册。当然,本书也十分适合希望更多地了解这一重要主题的社会人士。

许多人以多种方式对本书作出了贡献。我们尤其要感谢 Arno Bakker、Gerco Ballintijn、Brent Callaghan、Scott Cannon、Sandra Cornelissen、Mike Dahlin、Mark Darbyshire、Guy Eddon、Amr el Abbadi、Vincent Freeh、Chandana Gamage、Ben Gras、Bob Gray、Michael van Hartskamp、Philip Homburg、Andrew Kitchen、Ladislav Kohout、Bob Kutter、Jussipekka Leiwo、Leah McTaggert、Eli Messenger、Donald Miller、Shivakant Mishra、Jim Mooney、Matt Mutka、Rob Pike、Krithi Ramamirtham、Shmuel Rotenstreich、Sol Shatz、Gurdip Singh、Aditya Shivram、Vladimir Sukonnik、Boleslaw Szymanski、Laurent Therond 和 Leendert van Doom,感谢他们阅读了部分书稿并提出了宝贵意见。

最后,我们还要感谢我们的家庭。Suzanne 已经经历过很多次这样的情况了。她从未说过“我受够了”,尽管这个念头肯定在她脑海里出现过。谢谢你!Barbara 和 Marvin 现在对教授们为谋生所做的工作有了更好的了解,并且认识到好教材和坏教材之间的差别。现在,对我来说他们是我努力创作出更多好教材的动力所在。

本书使用指南

我们使用本书中的材料已经很多年了,主要是用作大学高年级学生和研究生的教材。而且,这些材料还曾经作为为时 1~2 天的有关分布式系统和中间件的研讨会的基本资料,参加这些研讨会的人包括 ICT 专家(技术上的)。下面是我们根据经验对本书使用方式提出的一些建议。

大学高年级学生和研究生教材

如果作为大学高年级学生和研究生的教材,本书通常可以在 12~15 周内完成教学。我们发现,在大多数学生看来,分布式系统由很多似乎彼此紧密结合的主题所组成。在本书的组织上,我们按照不同的原理介绍这些主题,分别讲授各个原理,这对学生领会重点内容有很大帮助。这样安排的效果是当第一部分(第 1~8 章)结束时,即在讨论范型之前,学生已经对本书主题在整体上有了一个相当好的把握。

然而,分布式系统的领域涵盖许多不同的主题,其中一些主题在初次学习时很难理解。因此,我们强烈建议学生们随着课程的进展学习适当的章节。从 Web 站点(<http://www.prenhall.com/tanenbaum>)可以获得所有 PowerPoint 表,将它们预先分发下去,以便学生在课堂中能够积极参与讨论。这种方法非常成功,并得到了学生们的高度评价。

所有的材料都包括在一个为时 15 周的课程中。大多数时间花费在讲授分布式系统的原理,也就是前 8 章所包括的材料上。在讨论范型时,我们的经验是:只需要介绍要点。直接从书中学习每个案例的详细内容比在课堂上听授更加容易。例如,尽管书中有关基于对象的系统的内容达 80 页之多,但我们只用一周的时间讲授这类系统。下面是一个课程进度安排建议表(表 0.1),其中包括每次讲座中包括的主题。

表 0.1 课程进度安排建议

周	主题	章	讲授内容
1	绪论	1	全部
2	通信	2	2.1~2.3
3	通信	2	2.4~2.5
4	进程	3	全部
5	命名	4	4.1~4.2
6	命名	4	4.3
6	同步	5	5.1~5.2

续表

周	主题	章	讲授内容
7	同步	5	5.3~5.6
8	一致性和复制	6	6.1~6.4
9	一致性和复制	6	6.5~6.6
9	容错	7	7.1~7.3
10	容错	7	7.4~7.6
11	安全性	8	8.1~8.2
12	安全性	8	8.3~8.7
13	基于对象的系统	9	全部
14	文件系统	10	全部
15	基于文档的系统	11	全部
15	基于协作的系统	12	全部

并不是所有材料都需要在课堂上讲授;我们希望学生能够自学特定的部分,尤其是细节部分。在讲授时间少于 15 周的情况下,我们建议跳过有关范型的章节,让感兴趣的学生自己学习这些部分。

如果用于低年级的课程,我们推荐将本书的学习延长至两个学期,并增加实验作业。例如,可以通过让学生修改一些组件,使这些组件具有容错性、处理多播 RPC 等功能来使学生理解简单的分布式系统。

行业的专业研讨会

在 1~2 天的研讨会上,通常将本书作为主要的背景材料使用。然而,如果跳过所有细节,仅将重点放在分布式系统的本质上,则有可能在两天内讲完整本书。此外,要使内容的表达更加生动实用,有必要重新安排章节的顺序,以提早说明原理是如何得到应用的。对于研究生来说,一般是在了解原理的应用之前(有时甚至根本不了解原理的具体应用)先对原理进行为期 10 周的学习,但专业人士如果能了解这些原理的实际应用,就会有更大的学习动力。下面是一个为期 2 天课程的试验性进度表(表 0.2),该表按照逻辑单元进行划分。

表 0.2 按逻辑单元划分的课程进度

第 1 天				
单元	时间(分)	主 题	章	重 点
1	90	绪论	1	客户-服务器体系结构
2	60	通信	2	RPC/RMI 和消息传递
3	60	基于协作的系统	12	消息传递问题
4	60	进程	3	移动代码和代理
5	30	命名	4	位置跟踪
6	90	基于对象的系统	9	CORBA

第 2 天				
单元	时间(分)	主 题	章	重 点
1	90	一致性和复制	6	模型和协议
2	60	基于文档的系统	11	Web 缓存/复制
3	60	容错	7	进程组与 2PC
4	90	安全性	8	基本思想
5	60	分布式文件系统	10	NFS v3 和 v4

个人学习

本书同样也适用于个人学习。如果具有足够的时间和动力,建议读者仔细阅读整本书。

如果没有足够的时间仔细阅读所有材料,我们建议只集中学习最重要的主题。下面的表格中列举一些章节,我们认为这些章节涵盖了关于分布式系统的最重要的主题(表 0.3)。

表 0.3 自学内容

章	主 题	小 节
1	绪论	1.1、1.2、1.4.3、1.5
2	通信	2.2、2.3、2.4
3	进程	3.3、3.4、3.5
4	命名	4.1、4.2
5	同步	5.2、5.3、5.6
6	一致性和复制	6.1、6.2.2、6.2.5、6.4、6.5
7	容错	7.1、7.2.1、7.2.2、7.3、7.4.1、7.4.3、7.5.1
8	安全性	8.1、8.2.1、8.2.2、8.3、8.4
9	基于对象的系统	9.1、9.2、9.4
10	分布式文件系统	10.1、10.4
11	基于文档的系统	11.1
12	基于协作的系统	12.1、12.2 或 12.3

比较好的做法是对学习这些建议的材料需要花费的时间进行估算,但这在很大程度上取决于读者的背景知识,对各种背景的读者很难做一个一般性的估计。然而,如果一个具有全职工作的人抽出晚上的时间阅读本书,则可能至少花费几周时间。

目 录

第 1 章 绪论	1
1.1 分布式系统的定义	1
1.2 目标	3
1.2.1 让用户连接到资源	3
1.2.2 透明性	4
1.2.3 开放性	6
1.2.4 可扩展性	7
1.3 分布式系统的硬件	12
1.3.1 多处理器系统	13
1.3.2 同构式多计算机系统	15
1.3.3 异构式多计算机系统	16
1.4 分布式系统的软件	17
1.4.1 分布式操作系统	18
1.4.2 网络操作系统	26
1.4.3 中间件	28
1.5 客户-服务器模型	33
1.5.1 客户与服务器	33
1.5.2 应用程序的分层	38
1.5.3 客户-服务器体系结构	40
1.6 小结	43
习题	43
第 2 章 通信	45
2.1 分层协议	45
2.1.1 低层协议	48
2.1.2 传输协议	50
2.1.3 高层协议	52
2.2 远程过程调用	54
2.2.1 基本的 RPC 操作	55
2.2.2 参数传递	58
2.2.3 扩展的 RPC 模型	61
2.2.4 实例: DCE RPC	64
2.3 远程对象调用	68
2.3.1 分布式对象	68

2.3.2 将客户绑定到对象	70
2.3.3 静态远程方法调用与动态远程方法调用	72
2.3.4 参数传递	73
2.3.5 实例 1: DCE 远程对象	74
2.3.6 实例 2: Java RMI	76
2.4 面向消息的通信	79
2.4.1 通信中的持久性和同步性	79
2.4.2 面向消息的暂时通信	83
2.4.3 面向消息的持久通信	86
2.4.4 示例: IBM MQSeries	91
2.5 面向流的通信	95
2.5.1 为连续媒体提供支持	95
2.5.2 流与服务质量	98
2.5.3 流同步	101
2.6 小结	103
习题	104

第 3 章 进程	107
3.1 线程	107
3.1.1 线程简介	107
3.1.2 分布式系统中的线程	112
3.2 客户	114
3.2.1 用户界面	114
3.2.2 客户端软件与分布透明性	116
3.3 服务器	117
3.3.1 设计上常见的 important 问题	117
3.3.2 对象服务器	120
3.4 代码迁移	125
3.4.1 代码迁移方案	125
3.4.2 迁移与本地资源	128
3.4.3 异构系统中的代码迁移	131
3.4.4 实例: D'Agents	132
3.5 软件代理	136
3.5.1 分布式系统中的软件代理	136
3.5.2 代理技术	138
3.6 小结	140
习题	141

第 4 章 命名	144
4.1 实体的命名	144
4.1.1 名称、标识符和地址	144

4.1.2 名称解析	148
4.1.3 名称空间的实现	152
4.1.4 示例：域名系统	158
4.1.5 示例：X.500	161
4.2 移动实体的定位	165
4.2.1 实体命名与定位	165
4.2.2 简单方法	167
4.2.3 基于起始位置的方法	169
4.2.4 分层方法	171
4.3 删除无引用的实体	176
4.3.1 无引用对象的问题	177
4.3.2 引用计数	178
4.3.3 引用列表	181
4.3.4 标识不可到达实体	182
4.4 小结	187
习题	188
 第 5 章 同步	190
5.1 时钟同步	190
5.1.1 物理时钟	191
5.1.2 时钟同步算法	194
5.1.3 使用同步时钟	197
5.2 逻辑时钟	198
5.2.1 Lamport 时间戳	199
5.2.2 向量时间戳	201
5.3 全局状态	203
5.4 选举算法	206
5.4.1 欺负(Bully)算法	206
5.4.2 环算法	207
5.5 互斥	208
5.5.1 集中式算法	208
5.5.2 分布式算法	209
5.5.3 令牌环算法	211
5.5.4 三个算法的比较	212
5.6 分布式事务	213
5.6.1 事务模型	213
5.6.2 事务的分类	216
5.6.3 实现	218
5.6.4 并发控制	220
5.7 小结	226
习题	227

第 6 章 一致性和复制	229
6.1 简介	229
6.1.1 复制的目的	230
6.1.2 对象复制	230
6.1.3 作为扩展技术的复制	232
6.2 以数据为中心的一致性模型	233
6.2.1 严格一致性	234
6.2.2 线性化和顺序一致性	236
6.2.3 因果一致性	239
6.2.4 FIFO 一致性	240
6.2.5 弱一致性	242
6.2.6 释放一致性	244
6.2.7 入口一致性	245
6.2.8 一致性模型小结	247
6.3 以客户为中心的一致性模型	248
6.3.1 最终一致性	249
6.3.2 单调读	250
6.3.3 单调写	251
6.3.4 写后读	252
6.3.5 读后写	253
6.3.6 实现	254
6.4 分发协议	256
6.4.1 副本放置	256
6.4.2 更新传播	259
6.4.3 epidemic 协议	262
6.5 一致性协议	264
6.5.1 基于主备份的协议	264
6.5.2 复制的写协议	267
6.5.3 高速缓存相关性协议	270
6.6 实例	271
6.6.1 Orca	272
6.6.2 因果一致的懒惰复制	276
6.7 小结	279
习题	280

第 7 章 容错性	283
7.1 容错性简介	283
7.1.1 基本概念	283
7.1.2 典型故障	285
7.1.3 使用冗余来掩盖故障	287
7.2 进程恢复	288

7.2.1	设计问题	288
7.2.2	故障掩盖和复制	290
7.2.3	故障系统的协议	290
7.3	可靠的客户-服务器通信	293
7.3.1	点到点通信	293
7.3.2	出现失败时的 RPC 语义	293
7.4	可靠的组通信	298
7.4.1	基本的可靠多播方法	298
7.4.2	可靠多播中的可扩展性	299
7.4.3	原子多播	301
7.5	分布式提交	307
7.5.1	两阶段提交	307
7.5.2	三阶段提交	312
7.6	恢复	313
7.6.1	简介	314
7.6.2	检查点	316
7.6.3	消息日志	318
7.7	小结	320
	习题	321
第 8 章	安全性	323
8.1	安全性介绍	323
8.1.1	安全威胁、策略和机制	323
8.1.2	设计问题	328
8.1.3	加密	331
8.2	安全通道	337
8.2.1	身份验证	338
8.2.2	消息完整性和机密性	344
8.2.3	安全组通信	346
8.3	访问控制	349
8.3.1	访问控制中的一般问题	349
8.3.2	防火墙	352
8.3.3	保护移动代码	354
8.4	安全管理	359
8.4.1	密钥管理	359
8.4.2	安全组管理	363
8.4.3	授权管理	364
8.5	实例：KERBEROS	368
8.6	实例：SESAME	370
8.6.1	SESAME 组件	370
8.6.2	PAC	372
8.7	实例：电子付费系统	373

8.7.1	电子付费系统	373
8.7.2	电子付费系统中的安全性	375
8.7.3	协议实例	377
8.8	小结	381
	习题	382

第9章	基于对象的分布式系统	384
9.1	CORBA	384
9.1.1	CORBA 概述	385
9.1.2	通信	390
9.1.3	进程	395
9.1.4	命名	399
9.1.5	同步	402
9.1.6	缓存与复制	403
9.1.7	容错性	404
9.1.8	安全性	406
9.2	分布式组件对象模型(DCOM)	408
9.2.1	DCOM 概述	408
9.2.2	通信	413
9.2.3	进程	415
9.2.4	命名	417
9.2.5	同步	420
9.2.6	复制	420
9.2.7	容错性	420
9.2.8	安全性	421
9.3	Globe	423
9.3.1	Globe 概述	423
9.3.2	通信	430
9.3.3	进程	430
9.3.4	命名	432
9.3.5	同步	435
9.3.6	复制	435
9.3.7	容错性	437
9.3.8	安全性	438
9.4	CORBA、DCOM 和 Globe 的比较	439
9.4.1	基本原理	439
9.4.2	通信	440
9.4.3	进程	441
9.4.4	命名	441
9.4.5	同步	442
9.4.6	缓存与复制	442
9.4.7	容错性	442

9.4.8 安全性	442
9.5 小结	444
习题	444

第 10 章 分布式文件系统 446

10.1 SUN 网络文件系统	446
10.1.1 NFS 概述	447
10.1.2 通信	450
10.1.3 进程	451
10.1.4 命名	452
10.1.5 同步	458
10.1.6 缓存和复制	462
10.1.7 容错性	464
10.1.8 安全性	466
10.2 Coda 文件系统	469
10.2.1 Coda 概述	469
10.2.2 通信	471
10.2.3 进程	472
10.2.4 命名	473
10.2.5 同步	474
10.2.6 缓存和复制	477
10.2.7 容错性	480
10.2.8 安全性	482
10.3 其他分布式文件系统	484
10.3.1 Plan 9: 资源统一为文件	485
10.3.2 xFS: 无服务器的文件系统	489
10.3.3 SFS: 可扩展的安全性	494
10.4 分布式文件系统的比较	496
10.4.1 设计理念	497
10.4.2 通信	497
10.4.3 进程	497
10.4.4 命名	498
10.4.5 同步	499
10.4.6 缓存和复制	499
10.4.7 容错性	499
10.4.8 安全性	500
10.5 小结	501
习题	501

第 11 章 基于文档的分布式系统 503

11.1 WWW	503
----------------	-----

11.1.1	WWW 概述	504
11.1.2	通信	511
11.1.3	进程	515
11.1.4	命名	520
11.1.5	同步	522
11.1.6	缓存和复制	522
11.1.7	容错性	526
11.1.8	安全性	526
11.2	Lotus Notes	527
11.2.1	Lotus Notes 概述	527
11.2.2	通信	529
11.2.3	进程	530
11.2.4	命名	531
11.2.5	同步	533
11.2.6	复制	533
11.2.7	容错性	535
11.2.8	安全性	535
11.3	WWW 和 Lotus Notes 的比较	538
11.4	小结	542
	习题	542

第 12 章	基于协作的分布式系统	544
12.1	协作模型介绍	544
12.2	TIB/Rendezvous	546
12.2.1	TIB/Rendezvous 概述	546
12.2.2	通信	548
12.2.3	进程	551
12.2.4	命名	551
12.2.5	同步	553
12.2.6	缓存和复制	554
12.2.7	容错性	554
12.2.8	安全性	556
12.3	Jini	557
12.3.1	Jini 概述	558
12.3.2	通信	560
12.3.3	进程	561
12.3.4	命名	563
12.3.5	同步	565
12.3.6	缓存和复制	567
12.3.7	容错性	567
12.3.8	安全性	567
12.4	TIB/Rendezvous 和 Jini 的比较	568

12.5 小结	571
习题	571

第 13 章 阅读材料和参考书目 573

13.1 对进一步阅读的建议 573
13.1.1 介绍性和综述性的著作 573
13.1.2 通信 574
13.1.3 进程 575
13.1.4 命名 576
13.1.5 同步 576
13.1.6 一致性与复制 577
13.1.7 容错性 578
13.1.8 安全性 579
13.1.9 面向对象的分布式系统 580
13.1.10 分布式文件系统 581
13.1.11 基于文档的分布式系统 582
13.1.12 基于协作的分布式系统 583
13.2 参考书目列表 583

第 1 章 绪 论

计算机系统正在经历一场革命。1945 年是现代计算机时代的开始。从那时起一直到 1985 年,计算机一直是庞大笨重而昂贵的。即便是小型机每台也要上万美元,因而大多数机构都只有很少的几台计算机。而且由于没有把这些计算机连接起来的手段,所以只能单独使用它们。

然而,从 20 世纪 80 年代中期开始,技术领域中两方面的进步开始使这种局面有所改观。第一项进步是高性能微处理器的开发。起初这些机器是 8 位机,但是很快 16 位、32 位乃至 64 位的 CPU 纷至沓来,得到了普及。很多使用这些 CPU 的机器的计算性能都可以与大型机相媲美,而价格却比大型机便宜得多。

近半个世纪以来,在计算机技术改进方面所取得的成果数量之多对于其他工业来说是完全无法想象的。一台计算机的价格从 1 亿美元下降到了 1000 美元,而运算速度却从每秒 1 条指令提升到每秒 1 千万条指令,性能价格比提升了大约 10^{12} 倍。如果汽车的性能价格比在同一时期以同样的速率提升,那么一辆“劳斯莱斯”牌轿车现在只值 1 美元,而且每消耗 1 加仑(1 加仑 = 4.5461 升)汽油可以行驶 10 亿英里(1 英里 = 1.6093 公里)。(不幸的是,如果果真如此,光是说明如何打开车门这一件事,就可能在用户手册中占用长达 200 页的篇幅。)

第二项进步是高速计算机网络的发明。利用局域网技术可以将位于同一座建筑物里的数百台机器连接起来。使用这种连接方式可以在几 μs 内将少量数据从一台机器传输到另一台机器。如果要传输大量的数据,传输速率可以达到 $10 \sim 1000 \text{ Mb/s}$ 。利用广域网技术可以连接全球范围内数百万台机器,机器间的数据传输速率从 64 Kb/s 到若干 Gb/s 不等。

现在,由于有以上这些技术可供使用,因此把若干由大量计算机组成的计算系统彼此通过高速网络相连接,不但是可行的,而且是很容易实现的。这种系统一般称为计算机网络或者分布式系统,以突出其与传统的集中式系统(centralized system,也称为单处理器系统,single processor system)的差异。传统的集中式系统一般由单个计算机及其外围设备构成,也可以包含一些远程终端。

1.1 分布式系统的定义

已经有很多文献给出了分布式系统的定义,但是不同文献给出的定义彼此不尽相同,而且没有一种令人满意。考虑到本书的目标,我们只给出一个粗略的描述:

分布式系统是若干独立计算机的集合,这些计算机对于用户来说就像是单个相关

系统。

这个定义包含了两方面的内容。第一个方面是关于硬件的：机器本身是独立的。第二个方面是关于软件的：对用户来说他们就像在与单个系统打交道。这两个方面共同阐明了分布式系统的本质，缺一不可。我们先介绍关于软硬件的一些背景材料，在本章稍后再回到这两个要点上来。

与继续探讨分布式系统的定义相比，把注意力集中在分布式系统的重要特性上可能更好一些。其重要特性之一是，各种计算机之间的差别以及计算机之间的通信方式的差别对用户是隐藏的。同样，用户也看不到分布式系统的内部组织结构。另一个重要特性是，用户和应用程序无论在何时何地都能够以一种一致和统一的方式与分布式系统进行交互。

分布式系统的扩展或者升级应该是相对比较容易的。这一特性是由于下述原因：分布式系统由独立的计算机组成，但同时隐藏了其中单个计算机在系统里所承担任务的细节。即使分布式系统中某些部分可能暂时发生故障，但其整体在通常情况下总是保持可用。用户和应用程序不会察觉到哪些部分正在进行替换和维修，以及加入了哪些新的部分来为更多的用户和应用程序提供服务。

为了使种类各异的计算机和网络都呈现为单个的系统，分布式系统常常通过一个“软件层”组织起来，该“软件层”在逻辑上位于由用户和应用程序组成的高层与由操作系统组成的低层之间，如图 1.1 所示。这样的分布式系统有时又称为中间件(middleware)。

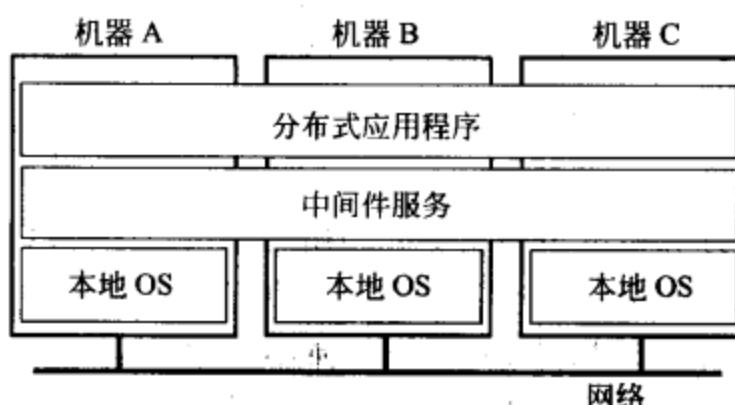


图 1.1 作为中间件组织的分布式系统(请注意，中间件层延伸到了多台机器上)

现在让我们考察一下分布式的几个例子。第一个例子是位于一所大学或者某个公司部门里的工作站网络。该系统除了包括每个用户自己的工作站以外，还应该包括机房内的一个处理器池。这些处理器并不分配给特定的用户，而是根据需要进行动态调配。这样的系统可能包含一个单一的文件系统，允许所有的机器通过相同的方法并且使用相同的路径名来访问所有文件。并且，当用户键入一个命令的时候，系统将寻找执行该命令的最佳位置，也许会在用户自己的工作站上直接执行该命令，也可能会在别人的一个空闲工作站上执行，还有可能由机房中某个尚未分配的处理器执行。如果系统整体外观和行为与传统的单处理器分时系统(即多用户系统)相似，那么这个系统就可以看作是分布式的系统。

第二个例子是某个工作流信息系统，该系统支持对订单的自动处理。一般情况下，会有来自多个不同部门的人员在不同的地点使用这样的系统。例如，销售部人员可能遍布

在很大的一个区域,甚至全国。可以通过电话网络(或者蜂窝电话)连接到系统的膝上型计算机下达订单。收到的订单由系统自动传送到计划部,接着新的内部调货订单就会送达仓储部,同时由财务部处理账单。该系统自动将订单传送到相关人员手中。用户根本看不到系统中订单处理的物理流程;对于用户来说,这些订单像是由一个集中式数据库处理的一样。

最后一个例子是万维网。它提供了一种简单、一致并且统一的分布式文档模型。要查看某个文档,用户只须激活一个引用(即链接),文档就会显示在屏幕上。理论上(但是目前在实际中并不是这样)并不需要知道该文档来自于哪个服务器,更用不着关心服务器所在的位置。要发布一个文档也很简单:只需要赋予它一个惟一的 URL 名,让该 URL 指向包含文档内容的本地文件即可。如果万维网向用户呈现的是一个庞大的集中式文档系统,也可以认为它是一个分布式系统。不幸的是,我们现在还无法做到这一点。例如,用户明白这样一个事实:文档位于不同的地点,并由不同的服务器处理。

1.2 目 标

如果仅仅是因为有能力组建分布式系统就去组建它,这并不是明智之举。以现在的技术来说,可以在一台个人计算机上安装 4 个软驱,但是这样做毫无意义。在本节中,我们将讨论要使得一个分布式系统真正物有所值必须具有的 4 个关键目标,即分布式系统必须能够让用户方便地与资源连接;必须隐藏资源在一个网络上分布这样一个事实;必须是开放的;必须是可扩展的。

1.2.1 让用户连接到资源

分布式系统的最主要目标是使用户能够方便地访问远程资源,并且以一种受控的方式与其他用户共享这些资源。资源几乎可以是任何东西,其典型例子包括打印机、计算机、存储设备、数据、文件、Web 页以及网络,但这些只是资源中的一小部分。有很多理由要求资源共享,一个显而易见的理由是降低经济成本。例如,让若干用户共享一台打印机比为每一位用户购买并维护一台打印机要划算得多。基于同样原因,共享超级计算机和高性能存储系统这样昂贵资源也是很有意义的。

用户连接到资源以后,协作和信息交换也会变得更加方便,在这方面 Internet 是一个最好的例子。它通过采用交换文件、邮件、文档、音频以及视频等数据的简单协议而获得了成功。Internet 的连接能力造就了众多的虚拟团体,这些团体由地域上分散的人们组成,他们通过群件(groupware)协同工作。群件是支持协作编辑、远程会议等活动的软件。Internet 的连接能力还使电子商务成为可能,现在通过电子商务我们不用去商店就可以买卖各种货物。

然而,在连接能力和共享功能不断加强的同时,安全性也变得愈发重要。就当前的实际情况而言,系统在通信窃听或入侵方面进行的防范是极为有限的。密码以及其他敏感信息常以明文(未加密)形式通过网络发送或者存储在服务器上,我们只能期望这些服务器是可信任的。从这个意义上来说,还有许多需要改进的地方。例如,现在只需要提供信

用卡号码就可以订购货物,极少对顾客是否确实拥有该信用卡进行验证。将来,只有在使用读卡机证实顾客确实拥有该信用卡实物的情况下,顾客才能下订单。

另外一个安全方面的问题是对通信的跟踪行为,其目的是建立一个针对特定用户的档案以记录该用户的偏好(Wang 等 1998)。这样的跟踪行为显然侵犯了隐私权,特别是在用户不知情的情况下更是如此。相关的问题是日益增长的连接能力会导致发生不希望的通信,比如垃圾电子邮件。在这些情况下,我们需要利用专门的信息过滤器来保护自己,这些过滤器可以根据传入信息的内容来决定允许哪些信息通过。

1.2.2 透明性

分布式系统的重要目标之一是将它的进程和资源实际上在多台计算机上分布这样一个事实隐藏起来。如果一个分布式系统能够在用户和应用程序面前呈现为单个计算机系统,这样的分布式系统就称为透明的。我们首先研究一下分布式系统中存在哪些种类的透明性,然后再分析是否在任何情况下都必须保证透明性。

1. 分布式系统的透明性

透明性的概念可以运用到分布式系统中的各个方面,如图 1.2 所示。

透明性	说明
访问	隐藏数据表示形式的不同以及资源访问方式的不同
位置	隐藏资源所在位置
迁移	隐藏资源是否移动到另一个位置
重定位	隐藏资源是否在使用过程中移动到另一个位置
复制	隐藏是否对资源进行复制
并发	隐藏资源是否由若干相互竞争的用户共享
故障	隐藏资源的故障和恢复
持久性	隐藏资源(软件)位于内存中还是硬盘中

图 1.2 分布式系统中不同形式的透明性(ISO 1995)

访问透明性指对不同数据表示形式以及资源访问方式的隐藏。例如,要将一个整型数从基于 Intel 处理器的工作站传输到一台 Sun SPARC 机器上,我们必须考虑到 Intel 处理器传输数据字节的顺序是 little endian 格式(先传输数据的高位字节),而 SPARC 处理器却使用 big endian 格式(先传输数据的低位字节)。在数据表示形式上还会存在其他差别。例如,分布式系统中的多个计算机系统可能运行的是不同的操作系统,这些操作系统的文件命名方式不同。文件命名方式的差异以及由此引发的文件操作方式的差异都应该对用户和应用程序隐藏起来。

透明性类型中很重要的一类是关于资源位置的。位置透明性指的是用户无法判别资源在系统中的物理位置。命名在确保位置透明性方面扮演了重要角色。特别是,位置透明性可以通过仅为资源分配逻辑名来取得,这种逻辑名和资源的位置完全无关。举一个这种名字的例子,从 <http://www.prenhall.com/index.html> 这个 URL 看不出 Prentice

Hall 的主 Web 服务器的所在位置,同样也看不出 index.html 是一直位于现在所在的位置还是只是最近才移到那里的。如果分布式系统中的资源移动不会影响该资源的访问方式,就可以说这种分布式系统就能提供迁移透明性。如果资源可以在接受访问的同时进行重新定位,而不引起用户和应用程序的注意,拥有这种资源的系统无疑会更加强壮,就可以说这种系统能提供重定位透明性。重定位透明性的一个例子是,当移动通信用户从一个地点移动到另一个地点时,可以一直使用他们的无线膝上型计算机,甚至连暂时的中断连接都不需要。

就像我们即将介绍的那样,复制在分布式系统里扮演了重要角色。例如,如果要增加系统的可用性或者提升系统性能,可以对资源进行复制,然后把其拷贝放置在访问该资源的位置附近。复制透明性指对同一个资源存在多个拷贝这样一个事实的隐藏。为了对用户隐藏复制行为,所有拷贝的名字都必须相同。这样,支持复制透明性的系统必须同时也支持位置透明性,不然系统就无法引用位于不同位置的多个拷贝。

我们已经提到过,分布式系统的重要目标之一是实现资源共享。多数情况下,资源共享是通过协作方式完成的,就像在通信中那样。然而,实际中还有许多竞争式资源共享的例子。例如,两个独立用户将各自的文件存储在同一台文件服务器上,或者访问位于一个共享数据库中的同一批表。在这种情况下,重要的是让任何一个用户不会感觉到他人也在使用自己使用的资源。这种现象称为并发透明性。必须确保对共享资源的并发访问不会破坏资源的一致状态。一致性可以通过使用锁定机制来保证。通过这种机制,用户轮流对请求的资源进行独占访问。更加精致的技术是运用事务处理机制,然而就像在后续章节中将会介绍的那样,在分布式系统中要实现事务处理机制是很困难的。

关于分布式系统还有一个流行的定义,是由 Leslie Lamport 提出的:“如果你从未听说过的某台计算机的崩溃导致你什么也干不了了,就说明你所打交道的是一个分布式系统”。这个描述道出了分布式系统设计上的另一个重要问题:故障处理机制。如果一个分布式系统提供故障透明性,就意味着用户不会注意到某个资源(也许他从未听说过这个资源)无法正常工作,以及系统随后从故障中恢复的过程。屏蔽故障是分布式系统中最困难的问题之一。如果我们提出某些接近现实的假设,那么这种屏蔽甚至是不可能的,我们将在第 7 章中讨论这个问题。在故障屏蔽方面最大的困难在于无法区别出现故障的资源和反应速度极慢的资源。例如,在连接一台繁忙的 Web 服务器时,浏览器将会超时并且报告该 Web 页不可用。在这种情况下,用户无法判定该服务器是否真正崩溃了。

与分布式系统相关的透明性的最后一种类型是持久性透明性,它指的是对资源位于易失性存储器中还是位于磁盘上的隐藏。例如,许多面向对象的数据库提供直接调用存储对象的方法的功能。在这个调用过程中发生的事情是,数据库服务器首先将对象的状态数据由磁盘复制到主存储器中,执行操作,然后可能会把状态写回到辅助存储装置中去。然而用户觉察不到服务器将状态数据在主存储器和辅助存储器之间进行了移动。持久性在分布式系统中扮演着重要角色,而且它对于非分布式系统也是同样重要的。

2. 透明度

虽然对于任何分布式系统来说,实现分布透明性通常是可取的,但是在某些情况下,

把所有分布情况都对用户屏蔽得严严实实并不见得有好处。这样的一个例子是,如果您订阅了一份电子报刊,并且要求它在每天本地时间早上 7 点之前就寄到邮箱里,而现在您呆在地球另一端的一个不同时区,您就会发现“早报”不再是早上寄到的了。

同样,一个将分别位于旧金山和阿姆斯特丹的两个进程连接起来的广域分布式系统不可能在 35ms 内将某个消息从其中一个进程发送到另一个进程,这是由自然规律(光信号或者电信号传输速度的限制)决定的。实践表明,通过计算机网络发送这样的消息实际上需要数百 ms。信号传输的速度不仅受光速限制,而且还要受到线路中交换机处理能力的限制。

另外,还必须在高度的透明性和系统性能之间进行权衡,得出折衷方案。例如,许多 Internet 应用程序会不断尝试连接到某台服务器,多次失败后才最终放弃。这种在用户转向另一台服务器之前竭力隐藏服务器短暂故障的企图会导致整个系统变慢。在这种情况下,最好是让用户早一些放弃,或者至少允许用户取消这种连接的尝试。

另一个例子是,我们必须使位于各大洲的某个资源的多个拷贝在任何时候都保持一致。换句话说,如果对其中一个拷贝进行了改动,在进行其他任何操作之前必须先将这个改动传播给所有其他拷贝。很明显,这时即使是一个简单的更新操作也可能需要花费数秒钟才能完成,这是不可能对用户隐藏的。

结论是,在设计并实现分布式系统时,把实现分布的透明性作为目标是正确的,但是应该将它和其他方面的问题(比如性能)结合起来考虑。

1.2.3 开放性

分布式系统还有一个重要目标,即开放性。一个开放的分布式系统就是这样的系统,它根据一系列准则来提供服务,这些准则描述了所提供服务的语法和语义。例如,在计算机网络中,准则规定了发送和接收的消息的格式、内容及含义。对这些准则进行形式化,就产生了协议。在分布式系统中,服务通常是通过接口(interface)指定的,而接口一般是由 IDL(interface definition language, 接口定义语言)来描述的。用 IDL 编写的接口定义基本上只是记录服务的语法,即这些接口定义明确指定可用函数的名称、参数类型、返回值,以及指出可能出现的异常等。比较困难的是确切地说明这些服务所要完成的工作,也就是接口的语义。在实际应用中,这样的接口说明一般是用一种非正式的方式,通过自然语言给出的。

说明良好的接口定义允许需要某个接口的任意进程与提供该接口的进程进行通信。同时它也允许独立的双方各自完成截然不同的接口实现,这样就会出现运行方式完全相同的两个独立的分布式系统。良好的接口规范说明是完整且中立的。“完整”意味着完成接口的实现必不可少的所有内容都已经规定好了。然而,有许多接口定义都不完整,这样就需要开发人员添加针对特定实现的细节。同样重要的是,接口规范说明不应描述接口的实现应该是什么样子,它应该是中立的。完整性和中立性对于实现互操作性和可移植性很重要(Blair 和 Stefani 1998)。互操作性刻画了来自不同厂商的系统或组件的两种实现能够在何种程度上共存并且协同工作,这种共存和协同工作只能依赖于通过双方在公共标准中规定的各自所提供的服务来完成。可移植性刻画了这样的性能:如果为分布式

系统 A 开发了某个应用程序，并且另一个分布式系统 B 与 A 具有相同的接口，该应用程序在不作任何修改的情况下在 B 上执行的可行程度。

开放的分布式系统的另一个重要目标是，它必须是灵活的，要能够方便地把由不同开发人员开发的不同组件组合成整个系统。同时，还必须能够方便地添加新组件、替换现有的组件，而不会对那些无须改动的组件造成影响。换句话说，开放的分布式系统应该是可扩充的。例如，在灵活的系统中，要添加可运行于另一个操作系统上的部件应该是比较容易的，即使要更换整个文件系统也不应该太困难。但根据我们从日常实践中得到的经验，要实现灵活性是比较困难的。

将策略与机制分离

要在开放的分布式系统中获得灵活性，关键是要把系统组织成规模相对较小而且容易替换或修改的组件的集合。这意味着我们应该不只是提供最高层的接口定义（也就是对用户和应用程序可见的那部分内容），还应该提供系统内部各部分间的接口定义，并且描述它们的交互方式。这种方案是比较新颖的。许多比较老的系统，甚至包括某些现代的系统，都是使用一种单一的形式构建起来的。在这种较老的方案中，逻辑上组件是彼此分离的，但是作为一个整体加以实现，成为一个庞大的程序。在按这种方案实现的系统中，要在不影响整个系统的前提下更换或者修改某个组件是很困难的。单一系统并不是开放的，而是趋向于封闭的。

在分布式系统中，进行改动的需求常常出自于某个不能对特定用户或者应用程序提供最优策略的组件。举一个例子，考虑一下万维网中的缓存机制。浏览器一般允许用户修改缓存策略，例如可以指定缓存的大小，以及指定不断检查缓存的文档以确保与真正的文档一致，还是每次会话只检查一次。然而，用户无法操纵缓存的其他参数，比如一个文档应该在缓存中保持多久，以及缓存空间填满时应该删除哪个文档。同时，也无法根据文档内容来决定是否对其进行缓存。例如，用户可能想要缓存铁路时刻表，因为他知道时刻表一般极少改动，但他不会缓存当前高速公路上的交通状况信息。

我们需要的是将策略与机制相分离。在以上 Web 缓存的例子中，理想情况下浏览器应该仅提供存储文档的功能，同时允许用户决定每个文档是否需要保存以及保存多久。在实际应用中，这可以通过为用户提供大量可以动态设置的参数来实现。更好的做法是用户可以通过某个能插入到浏览器中的组件来定制实现自己的策略。当然，这种组件必须具有浏览器可以识别的接口，以允许浏览器调用与该接口相对应的过程。

1.2.4 可扩展性

通过互联网，世界范围的互联正在变得像给世界任何地方的人寄张明信片一样平常。考虑到这种情况，分布式系统开发者都将可扩展性作为最重要的设计目标之一。

系统的可扩展性至少可以通过三个方面来度量 (Neuman 1994)。首先，系统要能在规模上扩展，也就是说可以方便地把更多的用户和资源加入到系统中去。其次，如果系统中的用户和资源可以相隔极为遥远，这种系统就称为地域上可扩展的系统。最后，系统在管理上是可扩展的，也就是说，即使该分布式系统跨越多个独立的管理机构，仍然可以方

便地对其进行管理。不幸的是,在以上这三个方面或其中某一两个方面可扩展的系统常常在扩展之后表现出性能的下降。

1. 可扩展性问题

当一个系统需要进行扩展时,必须解决多方面的问题。首先考虑规模上的扩展。在需要支持更多的用户或资源时,我们常常受到集中的服务、数据以及算法所造成的限制(如图 1.3 所示)。例如,许多服务是以集中式的方式实现的,它们由分布式系统中一台特定的计算机上运行的单个服务器来提供。这种方案存在的问题是显而易见的:用户增多时该服务器将成为系统的瓶颈。即使它拥有无限的处理能力和存储能力,在系统达到一定规模后与该服务器的通信也将发生困难,从而使得系统规模无法继续增长。

不幸的是,有时不可避免地要使用单个服务器。设想这样一种情况,有一项服务用来管理高度机密的信息,例如医疗记录、银行账户、个人贷款等。这时最好把这项服务放在高度安全的一台服务器上,并且通过专门的网络组件阻止分布式系统的其他部分对其进行访问。把服务器复制到多个位置以提高性能的做法是不可取的,因为这样会使得该服务更容易受安全攻击。

概念	实例
集中式服务	供所有用户访问的单个服务器
集中式数据	单个在线电话簿
集中式算法	根据完整的信息来安排路由

图 1.3 可扩展性限制的实例

集中式数据与集中式服务一样有缺陷。如果我们要保存 5000 万人的电话号码和地址,会出现什么情况?假设每条数据记录占 50 个字符,一个 2.5GB 的硬盘就可以提供足够的存储空间。但是这里有一个同样的问题,如果只用一个数据库,无疑会使得这个数据库的进出通信线路中都充满了数据。同样,设想一下如果 Internet 的 DNS(domain name system, 域名系统)只用单个表来实现,会出现什么情况?DNS 维护世界范围内数百万台计算机的信息,提供用于定位 Web 服务器的关键服务。如果每个解析 URL 的请求都必须传送到唯一的 DNS 服务器,那么显然不会再有人使用 Web 了(当然,如果真是这样的话,DNS 服务器的拥挤问题也许就会解决了)。

最后,集中式算法也有缺陷。在大型分布式系统中,海量的信息必须在许多线路之间进行路由传送。从理论上来说,完成这种传送工作最好的办法是收集关于所有计算机及线路负载的完整信息,然后利用图论的算法来计算最优的路由线路,计算结果随即应用到系统中以改善信息路由的性能。

但是,收集并传送所有的输入和输出信息的做法本身就有弊端,因为这些消息会造成部分网络过载。实际上,如果任何一个算法是通过从所有站点收集信息,然后将这些信息发送到某个计算机加以处理,随后又向系统发布计算结果的方式来执行的,那么就不应该采用这种算法,只应该采用分散式算法。分散式算法一般有下列与集中式算法不同的特性:

- (1) 没有任何计算机拥有关于系统状态的完整信息。
- (2) 计算机只根据本地信息做出决策。
- (3) 某台计算机的故障不会使算法崩溃。
- (4) 没有这样的假设：存在全局性的时钟。

前三个特性来自于我们前面已经强调过的内容，最后一个特性的重要性也许不那么明显，但它也同样重要。如果某个算法以这样的语句开头：“在 12: 00: 00 所有计算机必须记录其输出队列的大小”，这样的算法必定失败，因为无法使各个计算机的时钟完全同步。算法必须考虑到时钟无法完全同步的问题。系统越大，这种不确定度就越大。在一个小型局域网内，通过相当的努力可以把所有时钟的同步误差缩小到几个 ms，但是要在全国范围甚至全球范围的网络里做到这一点是极其困难的。

地域上的可扩展性也存在着问题。难以对现有的为局域网设计的分布式系统进行扩展的主要原因之一是这些系统基于同步通信。在这种通信方式下，请求服务的一方（一般称为客户）在得到回应以前是处于阻塞状态的。这种方法通常适用于局域网，因为局域网内两台机器之间的通信最多只需要数百 μs 。然而，在广域系统中，必须考虑到进程间通信可能需要花费数百 ms，比局域网慢三个数量级。如果要在广域系统中使用同步通信建立交互式应用程序，应非常小心，还需要极大的耐心。

另一个困扰地域可扩展性的问题是，广域网中的通信从本质上说是不可靠的，而且几乎总是点对点的。相比之下，局域网一般提供高度可靠的基于广播的通信功能，使得在局域网中开发分布式系统要相对容易得多。例如，服务定位的问题。在局域系统中，进程可以简单地向每台计算机广播一条消息，询问每台计算机是否正运行该进程所需的服务。只有那些提供该服务的机器才会做出响应，在应答消息中包含该机器的网络地址。这种定位方案在广域系统中是不可想象的。在广域系统中必须设计专门的定位服务，这种服务必须能够扩展到世界范围来使用，并且能够向数十亿用户提供服务。在第 4 章中我们将详细介绍这种服务。

地域扩展与困扰规模扩展的集中式解决方案中存在的问题也息息相关。如果一个系统使用了许多集中式组件，显然地域扩展性将会由于广域通信造成的性能和可靠性上的问题而受到制约。另外，集中式组件还会导致网络资源的浪费。设想一下整个地区只使用一台邮件服务器会发生什么情况，这意味着如果要向你的邻居发送一封电子邮件，必须先连接到中心邮件服务器，它可能距离你的位置数百英里之遥。很明显，不能采取这种方法。

最后，有一个难以解决的问题，它在许多情况下还没有答案，这就是如何对一个跨越多个独立管理域的分布式系统进行扩展。要解决的关键问题是在资源使用（以及付费）、管理和安全问题上各域有着相互冲突的策略。

例如，一个位于单个域中的分布式系统的许多组件通常可以得到在同一个域中工作的用户的信任。在这种情况下，系统管理部门可能已经对应用程序进行过测试和认证，并且采取了专门的措施以确保这些组件不会受到破坏。从本质上说，用户信任的是系统管理员。然而，这种信任无法自动拓展到域的边界之外。

如果分布式系统拓展到了另一个域，必须采取两种类型的安全措施。首先，分布式系

统必须保护自己免受来自新域的恶意攻击。例如,来自新域的用户可能只应该拥有对于系统原始域中文件服务的读权限。同样的道理,像昂贵的图像排版设备或者高性能计算机这样的设备不应该允许其他用户使用。其次,新域也必须保护自身免受来自分布式系统的恶意攻击。典型的例子是下载类似 Web 浏览器中的 Java 小程序时要小心。基本上,新域无法预料这样的外来代码会导致什么后果,因而可能会严格限制对这种代码的访问权限。问题是如何实施这种限制,对此我们将在第 8 章中加以讨论。

2. 扩展技术

我们已经讨论了一些关于可扩展性的问题,现在面对的问题是在一般情况下如何解决这些问题。由于分布式系统中的可扩展性问题表现为服务器和网络能力有限所造成的性能问题,因此基本上只有三种扩展技术:隐藏通信等待时间、分布技术以及复制技术(Neuman 1994)。

隐藏通信等待时间的技术对于地域扩展是比较适用的。基本的想法很简单:尽量避免等待远程服务对请求的响应。例如,当对远程计算机的某个服务发出请求时,在发出请求端,除了等待服务器响应之外,还可以利用这段时间做其他工作。本质上,以这种方式构建的应用程序使用的是异步通信方式。当响应到来时,应用程序产生中断,并且调用专门的处理程序对前面发出的请求进行处理。异步通信方式通常用于批处理系统以及并行应用程序中,这些系统或程序或多或少都拥有在某项任务等待通信完成时调度其他独立任务执行的能力。另外一种选择是启动一个新的控制线程来执行请求,虽然该控制线程因为等待响应而受到阻塞,但是进程中的其他线程可以继续执行。

然而,有许多应用程序无法充分利用异步通信的优点。例如,在交互式应用程序中,如果用户发送了一个请求,他一般只会无所事事地等待着回答。在这种情况下,较好的解决方案是减少通信的总量,例如,把通常由服务器完成的部分计算工作交由请求服务的客户端进程来进行。在使用表单来访问数据库的情况下,这种方案是行之有效的。通常,填写表单时需要为每个字段发送一条单独消息,然后等待服务器的响应,如图 1.4(a)所示。例如,服务器可能在接受某个项目之前先检查该条目是否存在语法错误。更好的方案是先把用于填写表单(可能还有检查条目)的代码传输给客户,然后由客户返回完整的表单,如图 1.4(b)所示。这种传输代码的解决方案目前在 Web 中通过使用 Java 小程序的形式得到了广泛支持。

另一项重要的扩展技术是分布(distribution)技术。分布技术把某个组件分割成多个部分,然后再将它们分散到系统中去。使用分布技术的一个例子是 Internet DNS。DNS 名字空间是由域(domain)组成的分层树状结构,域又划分为互不重叠的区(zone),如图 1.5 所示。每个区内的名字都由单个域名服务器处理。在不涉及太多细节的情况下,可以把每个路径名想象成 Internet 上的主机名,与该主机的网络地址相关。从根本上来说,解析一个名字意味着返回与该名字相关联的主机的网络地址。例如,要解析名字 nl.vu.cs.flits,首先要把它发送给 Z1 区的服务器(如图 1.5 所示),该服务器会返回 Z2 区的服务器地址,该名字的其余部分 vu.cs.flits 会发送给 Z2 区的服务器。而 Z2 服务器将返回 Z3 区服务器的地址,Z3 区服务器将处理名字的最后一部分并且返回相应的主机地址。

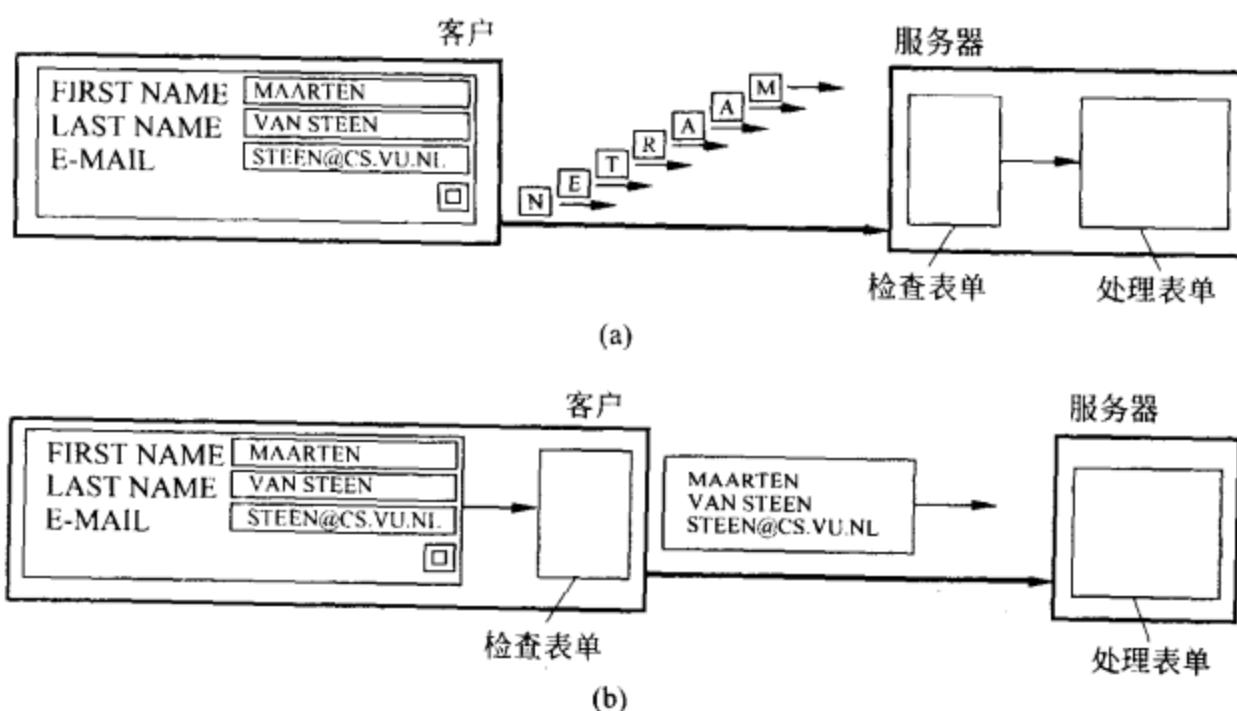


图 1.4 两种方案的区别

(a) 由服务器检查正在填写的表单; (b) 由客户检查正在填写的表单

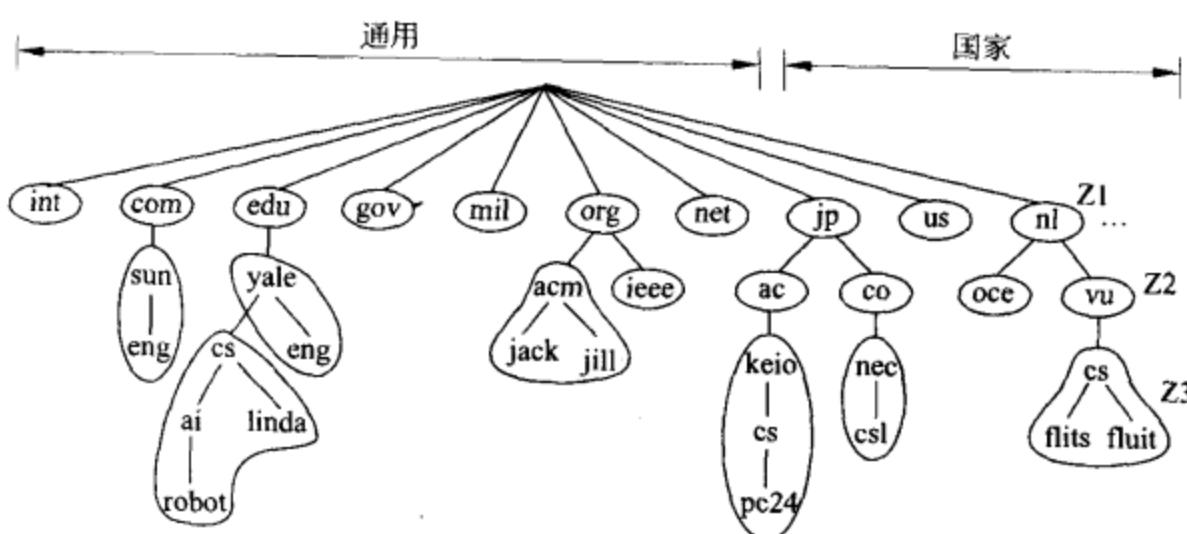


图 1.5 将 DNS 名字空间划分为区的例子

以上这个例子阐明了由 DNS 提供的命名服务是如何分布到多台计算机上进行的,这种做法避免了单个服务器不得不处理所有名字解析请求所面临的困境。

再来看看另一个例子:万维网(WWW)。对于大多数用户来说,Web 呈现给他们的是一个庞大的基于文档的信息系统,其中的每个文档都拥有以 URL 形式给出的惟一名字。从概念上讲,Web 看起来就像只有一个单独的服务器一样。然而,Web 在物理上是分布到非常多的服务器上的,每台服务器处理一部分 Web 文档。通过编码,把处理文档的服务器的名字变成该文档的 URL 中的一部分。正是因为对文档进行了分布处理,Web 才得以扩展到现在的规模。

考虑到可扩展性的问题通常表现为性能的下降,对组件进行复制并将拷贝分布到系统各处通常是一个好办法。复制不仅能够增加可用性,而且还能有助于组件间的负载平衡,从而使性能得到提高。同样,对于在地域上比较分散的系统来说,在请求者附近有一份拷贝可以在很大程度上隐藏前面提到的通信等待时间的问题。

缓存(caching)是复制的一种特殊形式,二者之间的区分通常很难划定。与复制相同,缓存一般是在访问资源的客户附近制作该资源的拷贝。然而,与复制不同的是,是否进行缓存是由要访问资源的客户决定的,而不是由资源拥有者决定的。

缓存和复制都存在严重的缺点,这些缺点可能会对可扩展性造成不良影响。进行缓存或者复制以后,由于资源存在多个拷贝,修改其中的一个会导致它与其他拷贝不相同,从而导致一致性(consistency)方面的问题。

可以在多大程度上容忍不一致性主要取决于资源的用法。例如,许多 Web 用户认为如果浏览器返回一个在最近几分钟内正确性未经验证的缓存文档是可以接受的。然而,在许多情况下必须确保高度的一致性,比如在进行电子股票交易时就是如此。高度一致性的关键问题是,对某个拷贝的更新必须立即传播到其他所有拷贝上。此外,如果两个更新操作并发执行,一般会要求每个拷贝都按照相同的次序进行更新。对这些情况的处理一般需要采取某些全局同步机制。不幸的是,要以一种可扩展的方式来实现这样的机制是极为困难的,甚至是不可能的。因此,通过复制来提高可扩展性将会引入另一种本质上不可扩展的解决方案。我们将在第 6 章中详细讨论复制和一致性问题。

1.3 分布式系统的硬件

尽管所有的分布式系统都包含多个 CPU,但硬件的组织仍然存在多种不同方式,特别是硬件相互连接以及通信的方式更是多种多样。在本节中,我们对分布式的硬件进行简要的讨论,着重讨论计算机间相互连接的方式。下一节中我们将讨论与分布式的软件相关的部分问题。

近些年来提出了多种针对多 CPU 计算机系统的分类方案,但是没有一种能够真正流行起来并得到广泛采用。这里,我们只关心那些由独立计算机的集合构建起来的系统。在图 1.6 中,我们把所有计算机分为两类:共享存储器的计算机系统,和不共享存储器的计算机系统。前者一般称作多处理器系统(multiprocessors),而后者有时称作多计算机系统(multicomputers)。它们之间的本质区别是:在多处理器系统中,所有 CPU 共享一个物理地址空间。例如,如果其中某个 CPU 向地址 1000 写入值 44,那么随后任何其他 CPU 在读取自己的地址 1000 时都会得到 44 这个值。也就是说,存储器是由所有机器共享的。

相比之下,在多计算机系统中,每台机器都拥有自己的专用存储器。如果某个 CPU 向其地址 1000 写入 44 这个值,随后其他 CPU 在读取地址 1000 时会得到该地址单元内原来存储的值,因为那个将值 44 写入自己存储器的 CPU 并不能影响到其他 CPU 的存储器。多计算机系统的一个常见例子是通过网络连接起来的一组个人计算机。

在这两个类别中,还可以根据网络互联体系结构进行进一步划分。在图 1.6 中我们将这两类分别称为总线型(bus)和交换型(switted)。总线型指存在一个连接所有机器的网络、底板、总线、电缆或者其他介质。有线电视采用的就是总线型结构,即有线电视公司在街道下面敷设一条线路,所有的用户使用分接头从该线路把信号引到自己的电视机。

交换型系统不像有线电视那样有一条单独的主干线,而是各机器之间都用独立的线

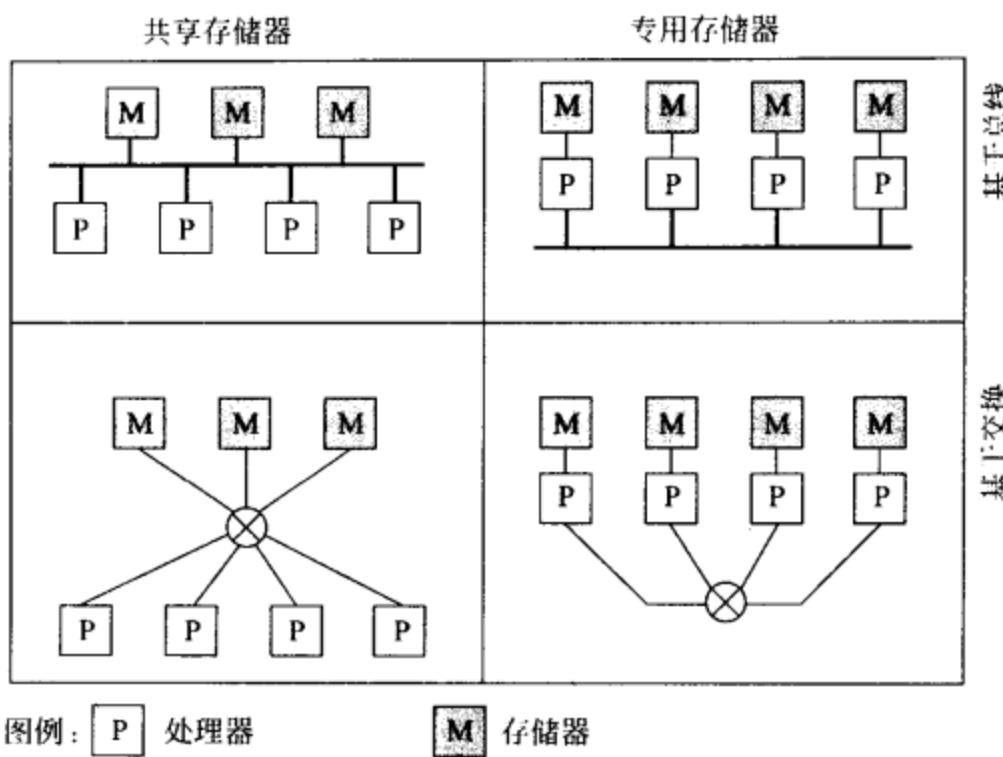


图 1.6 分布式计算机系统中处理器和存储器的不同组织方式

路相连。在实际使用中可以有多种不同的布线方式。消息沿着这些线路传输，在传输的每一步都需要做出专门的交换决策以路由消息沿着特定的线路发送。国际公用电话系统采用的就是这种方式。

可以把分布式计算机系统进一步划分为两类，一类是同构的(homogeneous)，另一类是异构的(heterogeneous)。这种划分只对多计算机系统有意义。在同构式多计算机系统中，本质上系统只有一个单独的互联网络，该网络在系统中全部使用同一种技术。同样，所有处理器也都是相同的，而且一般能够访问相同大小的专用存储器。同构式多计算机系统更多地作为并行系统(计算单个问题)使用，就像多处理器系统一样。

相比之下，异构式多计算机系统中可能包含有许多不同的独立计算机，它们通过不同的网络相联。例如，分布式系统可能由一组不同的局域计算机网络构成，这些局域的计算机网络通过 FDDI 或者 ATM 交换主干系统相联。

在接下来的三节中，我们将对多处理器系统以及同构式和异构式多计算机系统进行更详细的讨论。虽然这些主题与我们主要讨论的分布式系统没有直接联系，但是它们还是对我们的主题介绍有所帮助的，因为分布式系统的组建常常依赖于底层的硬件。

1.3.1 多处理器系统

多处理器系统有一个共同的关键属性：系统中所有 CPU 都能够直接访问共享的存储器。基于总线的多处理器系统包含多个连接到一条公用总线的 CPU 以及一个存储器模块。一种简单的系统构造实例是使用高速的底板或者主板，把 CPU 和存储器卡插接在上面。

由于只有存储器是共享的，所以如果 CPU A 将一个字写到存储器， $1\mu s$ 后 CPU B 读出该字，B 将会得到由 A 刚刚写入的那个值。具有这种性质的存储器称为相关的

(coherent)。这种方案的问题是,如果 CPU 的数目增加到 4~5 个,总线就会不堪重负,性能将急剧降低。解决方案是在 CPU 和总线之间加装高速缓冲存储器(cache memory,以下简称为缓存——译者注),如图 1.7 所示。缓存中存有最近被访问的字。所有对存储器的请求都要通过缓存。如果所请求的字位于缓存内,缓存将对发出请求的 CPU 做出响应,而不需要向总线发出请求。如果缓存足够大,成功的概率(称作命中率)将会很高,每个 CPU 产生的总线通信量将会显著下降,这样就可以在系统中使用更多的 CPU。缓存的大小一般有 512KB~1MB,在这种规模下命中率一般为 90% 或者更高。

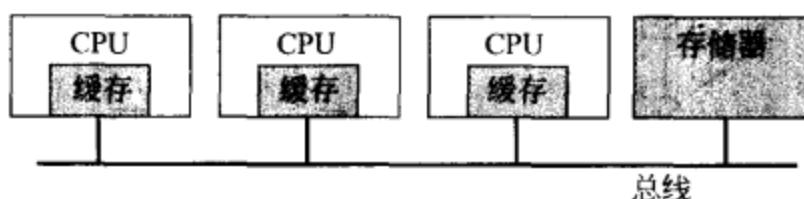


图 1.7 基于总线的多处理器系统

然而,引入高速缓冲存储器也会带来一个严重的问题。如果有两个分别称作 A 和 B 的 CPU 都将同一个字读入它们各自的缓存中。A 改写了该字,当 B 再次读入该字时,它读到的是位于缓存中的旧的结果,而不是 A 刚刚写入的新值。这样一来,存储器就变为不相关的,对这种系统很难进行编程。缓存技术在分布式系统中也得到了广泛应用,因而在分布式系统中我们同样要面对存储器非相关性的问题。我们将在第 6 章中详细讨论缓存技术以及存储器相关性的问题。如果想要更多地了解基于总线的多处理器系统,请参见文献(Lilja 1993)。

基于总线的多处理器系统存在的问题是,即使采用了缓存,其可扩展性仍然很有限。如果要组建的多处理器系统需要使用超过 256 个处理器,就必须采用另外的技术将 CPU 与存储器连接起来。一种可能的方案是,把存储器划分成若干个模块,通过纵横式交换器将这些存储器模块连接到 CPU,如图 1.8(a)所示。在每个 CPU 和每个存储器模块间都存在一条连接线路。在每个交叉处都有一个微型电子交叉点开关(crosspoint switch),可以在硬件上实现断开和闭合。如果某个 CPU 想要访问某个特定的存储器,连接该 CPU 和相应的存储器模块的交叉点开关将会立即闭合,访问便可进行。这种交叉点开关的优点在于多个 CPU 可以同时访问存储器。当然,如果两个 CPU 想要同时访问相同的存储器模块,其中的一个就必须等待。

纵横式交换器的缺点是,如果两边连接的分别是 n 个 CPU 和 n 个内存模块,则需要 n^2 个交叉点开关。当 n 很大时,需要的交叉点开关的数目可能大得令人难以承受。所以人们寻求并且找到了替代方案,这些方案的交换网络需要的开关数目比较少。图 1.8(b)中的 omega 网络就是这样一个例子。该网络包含 2×2 共 4 个交换器,每个交换器都有两路输入和两路输出,可以将每一路输入送到每一路输出。如果仔细地观察这幅图,就可以看到,如果正确设置了这些交换器,就可以让所有 CPU 访问每个存储器地址。这样的交换网络的缺陷在于 CPU 和存储器之间存在多个交换级别。因此,为了保证在 CPU 和存储器之间通信的等待时间不至于太长,交换器必须有极快的速度,但这样的交换器将会很贵。

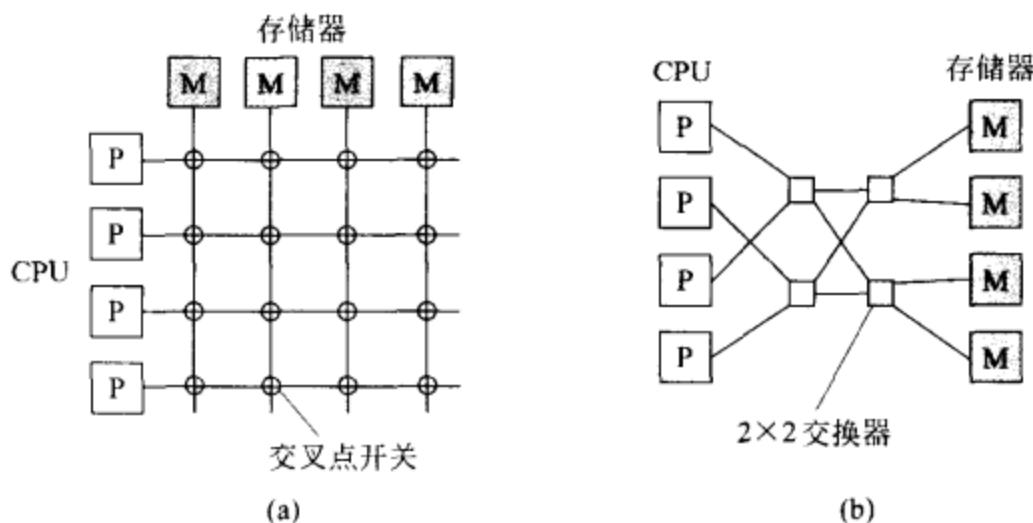


图 1.8 两种交换方式

(a) 交叉点开关; (b) omega 交换网络

人们曾经尝试转向分层系统以减少在交换上的开销。每个 CPU 都与特定的存储器相关联,CPU 访问自己拥有的本地存储器速度将会很快,但是访问属于其他 CPU 的存储器速度将会很慢。这种设计导致了非均匀存储器访问(nonuniform memory access,NUMA)机器的诞生。虽然 NUMA 机器的平均访问时间比基于 omega 网络的机器更短,但是它们又引入了新的复杂性:为了使大多数访问指向本地存储器,程序和数据的布局位置就变得很重要。

1.3.2 同构式多计算机系统

与多处理器系统不同,组建多计算机系统相对来说容易一些。每个 CPU 都与自己的本地存储器直接相连。惟一的问题是 CPU 之间的通信问题。很明显,这也需要某种互联方案,但是由于只用于 CPU 之间通信,其通信量要比 CPU 与存储器之间的通信量少几个数量级。

我们首先看看同构式多计算机系统,这种系统也称为系统域网络(system area network,SAN)。在这种系统中各结点安装在一个大框架上,通过一个单一的(通常是高性能的)互联网络连接起来。与多处理器系统类似,我们把同构式多计算机系统划分为基于总线的系统和基于交换的系统。

在基于总线的多计算机系统中,处理器通过快速以太网这样的共享多重访问网络彼此相联。典型的网络带宽为 100Mb/s。与基于总线的多处理器系统类似,基于总线的多计算机系统的可扩展性也很有限。根据实际需要通信的结点数目,一般说来系统中的结点超过 25~100 个之后性能就会下降到很低。

在基于交换的多计算机系统中,处理器之间的消息通过互联网络进行路由,而不像在基于总线的系统中那样通过广播来发送。人们提出并建造了互联网络的多种不同的拓扑结构。两种较为流行的拓扑结构是网状拓扑(mesh)和超立方体拓扑(hypercube),如图 1.9 所示。网格状的拓扑方式比较易于理解,也便于在印刷电路板上进行布局设计。这种方式最适用于解决本质上是二维的问题,例如图论和视觉方面的问题(比如机器人的眼睛设计或者照片分析)。

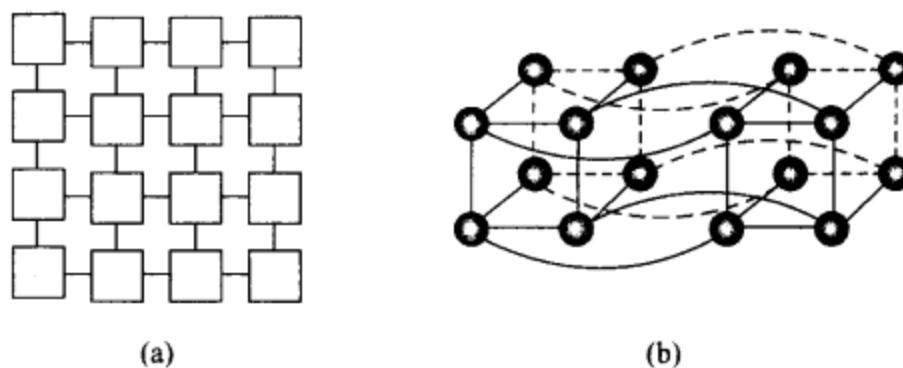


图 1.9 网状拓扑和超立方体拓扑

(a) 网状拓扑; (b) 超立方体拓扑

超立方体(hypercube)是一个 n 维的立方体。图1.9(b)中的超立方体是4维的,可以把它想象成两个普通的立方体,每一个有8个顶点和12条边。每个顶点代表一个CPU,每条边代表两个CPU之间的连接。两个立方体之间的相应顶点是相连的。如果要把这个超立方体拓展到5维,就要将原来的两个半部分中相应的边连接起来,从而加入另外1组互联立方体对,依此类推。

基于交换的多计算机系统是多种多样的,具有由大到小的各种规模。规模最大的是大规模并行处理器(massively parallel processors,MPP)系统,这种系统是价值上百万美元的巨型超级计算机,它包含数千个CPU。多数情况下,这些CPU与工作站或者PC中使用的CPU并没有两样。这种系统与其他多计算机系统的区别在于其使用了高性能的专用互联网络,网络的设计目的是尽量缩短等待时间并且达到高带宽。另外,还采取了专门的措施来确保容错性。由于系统中的CPU达到数千个之多,不可避免的是每个星期都会有一些CPU损坏。由于单个CPU的故障而导致整个系统的崩溃无疑是不可接受的。

规模较小的基于交换的多计算机系统中有一种流行形式,通常称为工作站群集(clusters of workstations,COW),这种系统一般是一组通过像Myrinet板(Boden等1995)这样的通信成品组件连接起来的标准PC或者工作站。COW和MPP之间的区别主要在于互联网络的不同。而且,在COW中没有专门的措施来保证高速I/O带宽,也没有专门的系统故障防护措施。总的说来,正是这种特点使得COW价格低廉且简单易用。

1.3.3 异构式多计算机系统

目前使用的多数分布式系统是在异构式多计算机系统的基础上组建起来的。这就意味着,系统各部分的计算机可能有着巨大的差异,这种差异体现在处理器类型、存储器大小以及I/O带宽等方面。实际上,系统中的某些计算机可能本身就是高性能并行系统,比如多处理器系统或者同构式多计算机系统。

另外,系统中的互联网络也可以是高度异构的。举例来说,笔者曾经参与构建一个自制的分布式计算机系统。该系统名叫DAS,包括4个多计算机系统群集,通过一个广域ATM交换主干系统互联。该系统的图片以及研究参考信息可以在<http://www.cs.vu.nl/~bal/das.html>上找到。这些群集也可以利用Internet的标准功能进行通信。每个群集使用的CPU类型都相同(Pentium III),群集中的互联网络也一样(Myrinet),只是处

理器的数目不同(64~128个不等)。

另一个使用异构方式的例子是在现有的网络和主干的基础上构建大规模多计算机系统。例如,常有这样的情况,校园范围的分布式系统运行于各系的局域网之上,用高速主干把各系的局域网联接起来。在广域系统中,不同的站点又可能会联接到公共网。该联接是由商业通信公司使用 SMDS 或者帧中继这样的网络服务来提供的。

与前一节讨论的系统不同,许多大规模的异构式多计算机系统没有一个整体的系统视图,也就是说应用程序不能假定在系统中各处都提供相同的性能和服务。例如,在 I-way 工程中(Foster 和 Kesselman 1998),若干高性能计算中心通过 Internet 互联。该系统的总体模型是,应用程序可以在每个站点预订并使用资源,但是不可能对应用程序隐藏不同站点之间的差异。

鉴于异构式多计算机系统的规模问题、固有的异构性以及(大多数系统)缺乏一个整体的系统视图,因而必须使用复杂的软件来建立供这类系统使用的应用程序。这正是分布式系统的用武之地。为了使应用程序开发人员不必关注底层的硬件,分布式系统提供一个软件层,从而对应用程序屏蔽了硬件层的细节(也就是实现了透明性)。

1.4 分布式系统的软件

分布式的硬件是重要的,但在更大程度上决定分布式的外在特征的则是软件。分布式系统很像传统的操作系统。首先,它们在行为上就像底层硬件的资源管理器,允许在多个用户和应用程序间共享资源。这些资源包括 CPU、存储器、外围设备、网络以及各种数据。其次,也许是更重要的一点,分布式系统试图通过向应用程序提供一个虚拟机让应用程序在上面运行,以隐藏底层硬件的不但错综复杂而且异构的本质。

为了理解分布式的本质,我们要先考察分布式的计算机使用的操作系统。供分布式的操作系统可以粗略地划分为两类:紧耦合系统与松耦合系统。在紧耦合系统中,操作系统主要是以一种简单的全局视图来管理其资源。而松耦合的系统可以看作是一组计算机,其中每台计算机都各自运行自己的操作系统,这些操作系统进行协作,使分属于各操作系统的资源和服务得到共享。

紧耦合系统与松耦合系统之间的区别与上一节中介绍的硬件有关。紧耦合系统一般又称为分布式操作系统(distributed operating system,DOS),用于管理多处理器系统和同构式多计算机系统。与传统的单处理器操作系统类似,分布式操作系统的主要目标是隐藏底层硬件管理的复杂性,例如硬件可以由多个进程共享这样的特性。

松耦合的网络操作系统(network operating system,NOS)用于管理异构式多计算机系统。虽然底层硬件管理也是 NOS 的重要任务之一,但是它与传统操作系统的区别主要在于它允许远程客户使用本地服务。在下面的各小节中,我们将先讨论紧耦合操作系统和松耦合操作系统。

必须对网络操作系统的服务进行改进,对分布透明性提供更好的支持,以使其真正符合分布式的系统的要求。这些改进导致了中间件的诞生。中间件位于现代分布式的系统的中心。中间件也将在本节中进行讨论。图 1.10 总结了关于 DOS、NOS 和中间件的主

要问题。

系统	说明	主要目标
DOS	紧耦合的操作系统,用于多处理器系统和同构式多计算机系统	隐藏及管理硬件资源
NOS	松耦合的操作系统,用于异构式多计算机系统(LAN 和 WAN)	向远程客户提供本地服务
中间件	位于 NOS 通用服务实现层之上的附加层	提供分布透明性

图 1.10 DOS(分布式操作系统)、NOS(网络操作系统)和中间件概述

1.4.1 分布式操作系统

有两种类型的分布式操作系统。多处理器操作系统(multiprocessor operating system)用来管理多处理器系统的资源;多计算机操作系统(multicomputer operating system)是一种为同构式多计算机系统开发的操作系统。分布式操作系统的功能在本质上与单处理器系统使用的传统操作系统没有两样,但是它还提供了操作多个 CPU 的功能。让我们先简单看一下单处理器操作系统:关于单处理器系统和多处理器系统使用的操作系统的详细介绍可以在(Tanenbaum 2001)中找到。

1. 单处理器操作系统

传统上,操作系统是用来管理只有单个 CPU 的计算机的。这种系统的主要目标是让用户和应用程序能够方便地共享资源。这些资源包括 CPU、主存储器、磁盘和外围设备。资源共享意味着不同的应用程序可以用一种彼此隔离的方式访问同一个硬件,对于其中某个应用程序来说,就好像这个资源是属于它自己的一样。同一个系统上可以有几个应用程序同时运行,每个程序使用属于自己的一组资源。在这个意义上可以说,操作系统实现了虚拟机(virtual machine),向应用程序提供多任务的功能。

在这种虚拟机内共享资源的一个重要特征就是应用程序彼此隔开。例如,不允许出现这种情况:两个独立的应用程序 A 和 B 同时执行,应用程序 A 只简单地通过访问应用程序 B 的数据当前所在的主存储器就可以修改属于 B 的数据。同样,我们需要确保应用程序只能以操作系统允许的方式来使用提供的功能。例如,通常应该阻止应用程序直接向网络接口复制消息的行为,操作系统应该提供通信原语,只有使用这些原语才能够在位于不同机器的应用程序之间传递消息。

因此,操作系统必须对硬件资源的使用方式及共享方式进行完全控制。所以大多数 CPU 都至少支持两种操作模式。在内核模式(kernel mode)下,允许所有指令的执行,在执行过程中可以访问所有存储器和寄存器。而在用户模式(user mode)下,对存储器和寄存器的访问受到限制。例如,会禁止应用程序访问超出某个地址范围(由操作系统设定)的存储器区域,并且禁止对设备寄存器的直接访问。在执行操作系统代码的过程中,CPU 切换到内核模式。然而,从用户模式切换到内核模式的唯一途径是通过由操作系统实现的系统调用。由于系统调用是操作系统提供的唯一的公共服务,并且由于有硬件协

助对存储器和寄存器的访问加以限制,操作系统可以对硬件资源做到完全控制。

尽管存在有两种操作模式,但实际上,操作系统的组织结构常常使所有的操作系统代码都以内核模式运行。结果操作系统变为一个庞大且单一的程序,运行于单独的一个地址空间。这种方案的缺陷是,常常难以对操作系统进行修改。换句话说,很难在不彻底关闭系统的情况下替换或者修改操作系统组件,有时甚至还需要完全重新编译并且重新安装。从开放性、软件工程、可靠性和可维护性的角度来说,采用单一的操作系统并不是一个好办法。

一个更加灵活的方案是将操作系统分成两个部分。第一个部分包括一组硬件管理模块,它们也能在用户模式下执行。例如,存储器管理主要包括对已分配给进程的存储器和空闲存储器的跟踪。惟一例外是在设置 MMU(存储管理部件)寄存器的时候,必须以内核模式执行。

操作系统的第二个部分包含一个小型的微内核(microkernel),其中含有必须以内核模式执行的代码。实际上,微内核只需要包含那些用来设置设备寄存器、在进程间切换CPU、操作MMU以及捕获硬件中断的代码。另外,微内核还要包含用来将系统调用传递给相应的用户级操作系统模块的代码,微内核对这些用户级的模块进行调用,然后返回它们得到的结果。这种方案的结构如图1.11所示。

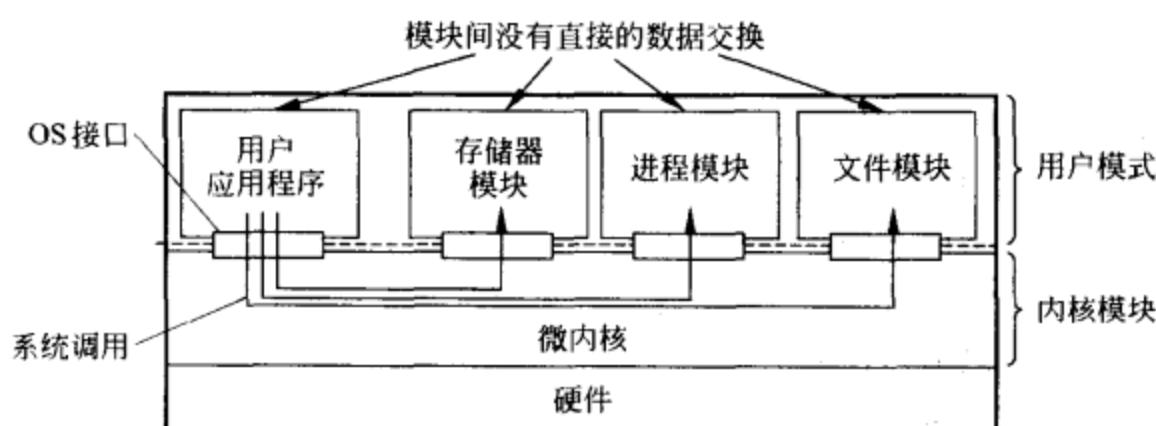


图 1.11 通过微内核分隔应用程序与操作系统代码

使用微内核有诸多好处。其中重要的一点是灵活性:由于操作系统中有一大部分代码以用户模式执行,要在不重新编译或者重新安装整个系统的情况下替换某个模块相对来说更加容易。另一个重要的好处是,原则上用户级的模块可以放置在不同的机器上。例如,可以很方便地把文件管理模块放置在一台机器上,而把目录服务放在另一台机器上。换句话说,采用基于微内核的方案可以方便地把单处理器操作系统扩展到分布式计算机上。

但是微内核也有两个严重的缺点。首先,微内核系统的工作方式与当前通用的操作系统不同,而尝试改变任何现状的努力总会遭遇重重阻力(“如果这种操作系统对我祖父来说足够好,那它对我来说也就足够好”)。其次,微内核需要进行额外的通信,这样就会损失少许的性能。然而,现在的CPU速度都很快,即使有20%的性能损失也并不严重。

2. 多处理器操作系统

对于单处理器操作系统的一个重要的但并不是很明显的扩展是提供对访问共享存储器的多处理器的支持。从概念上来说这种扩充是简单的,因为操作系统管理硬件(包括多个CPU)所需的所有数据结构都放在共享存储器中。主要的区别在于,这些数据现在可以由多个处理器来访问,这样就必须避免并发访问的发生,以确保数据的一致性。

然而,许多操作系统,特别是那些供PC和工作站使用的操作系统,无法方便地支持多个CPU。主要的原因是,这些操作系统被设计成为单一的程序,只能作为单个控制线程执行。要对这样的操作系统进行改造以供多处理器系统使用意味着要重新设计并实现整个内核。现代操作系统从开始设计的时候就把支持多CPU作为设计要求之一。

多处理器操作系统的目地是通过多个CPU支持高性能。一个关键的目地是使CPU的数目对应用程序透明。要达到这样的透明性相对比较容易,因为在不同的应用程序(或者应用程序的各部分)之间通信时使用的原语与在多任务单处理器操作系统中的原语相同。所有的通信是通过操纵位于共享存储器地址中的数据来进行的,只需要保护数据不在同一时刻受到多个访问。这种保护措施是通过同步原语实现的。两个重要的(并且等价的)原语是信号量(semaphore)和监控器(monitor)。

可以把信号量想象成一个整型数和两种操作:向下操作和向上操作。向下操作检查信号量的值是否大于0,如果是的话它就将值减1,然后继续。如果值为0,就将调用该操作的进程阻塞。向上操作则正好相反,它首先检查当前是否有在调用向下操作时被阻塞的进程;如果有的话,它将会释放被阻塞的进程之一,然后继续。否则,它将信号量的值加1。被释放的进程在从向下操作中返回后可以继续执行。信号量操作的一个重要特性是,这种操作是原子性的(atomic),也就是说一旦开始向下或者向上操作以后,在操作完成之前(或者在进程阻塞之前),任何别的进程都不可以访问该信号量。

众所周知,用信号量编程来进行进程间同步很容易出错,但用于简单的保护共享数据除外。主要的问题是,使用信号量很容易造成代码结构混乱,这和大量使用声名狼藉的goto语句所造成的情况一样。许多支持并发程序设计的现代操作系统提供了另外一种解决方案,它们提供用于实现监控器的库。

形式上,监控器是一种编程语言结构,与面向对象程序设计中的对象类似(Hoare 1974)。可以把监控器想象成一个包含变量和过程的模块,变量只能通过调用监控器中的某个过程来访问。从这种意义上说来,监控器与对象有相似之处:对象拥有自己的私有数据,只能通过由该对象实现的方法来访问。监控器与对象的不同之处在于,监控器在同一时刻只允许单个进程执行监控器中的一个过程。换句话说,如果进程A正在执行监控器中的某个过程(称为“A已经进入监控器”),而此时进程B也调用监控器中的某个过程,那么B将会被阻塞,直到A从调用返回为止(相应地称为“A离开监控器”)。

下面举一个例子。考虑图1.12中用于保护一个整型数据的简单监控器。该监控器含有一个(私有)变量count,该变量只能通过三个(公共)过程访问。这三个过程分别用于读取该变量的当前值、将该变量值加1、将该变量值减1。监控器结构确保了调用这三个过程之一的任何进程能够对监控器中包含的私有数据进行原子性的访问。

```

monitor Counter {
private:
    int count = 0;
public:
    int value() {return count;}
    void incr() { count = count + 1;}
    void decr() { count = count - 1;}
}

```

图 1.12 用于保护一个整数免受并发访问的监控器

到此为止可以看到,监控器对于简单地保护共享数据是有用的。然而,有时还需要在特定条件下阻塞某个进程。例如,假定我们希望能够这样:某个进程在调用操作 `decr` 时,如果发现 `count` 的值已经下降到 0 则将其阻塞。为了达到这样的目的,监控器中还包含有称为条件变量的特殊变量,这些特殊变量可以接受两种操作: `wait` 和 `signal`。当进程 A 正在监控器内部执行并且调用监控器中某个条件变量的 `wait` 操作时,A 将被阻塞,并且让出对监控器的独占访问权。这样,正在等待进入监控器的进程 B 就可以继续执行了。在某个特定执行点上,B 可以通过对 A 等待的条件变量执行 `signal` 操作来将 A 释放。为了避免两个活动进程同时位于监控器中,要求执行 `signal` 操作的进程必须离开监控器。现在我们可以对上一个例子进行改写。可以证明,图 1.13 中的监控器确实是对前面讨论过的信号量的一种实现。

```

monitor Counter{
private:
    int count = 0;
    int blocked_procs = 0;
    condition unblocked;
public:
    int value() {return count;}

    void incr() {
        if (blocked_procs == 0)
            count = count + 1;
        else
            signal(unblocked);
    }

    void decr() {
        if (count == 0) {
            blocked_procs = blocked_procs + 1;
            wait(unblocked);
            blocked_procs = blocked_procs - 1;
        }
        else
            count = count - 1;
    }
}

```

图 1.13 用于保护整数免受并发操作的监控器,它将阻塞某个进程

监控器的缺陷在于，它们是由程序设计语言所定义的结构。例如，Java 提供的监控器的原理从本质上来说，是让每个对象通过使用同步语句并对对象进行 wait 和 notify 操作来保护自己免受并发访问的。用于支持监控器的库一般都是通过使用只能取值为 0 和 1 的信号量来实现的，这种信号量一般称作互斥变量，与 lock(锁定)和 unlock(解锁)操作有关。对某个互斥量的锁定操作只有在互斥量为 1 时才会成功，否则调用该操作的进程将会被阻塞。同样，对某个互斥量的解锁操作将会释放某个正在等待(被阻塞)的进程，如果没有进程在等待则将该互斥量的值置为 1。条件变量及其相关操作也以库例程的形式提供。关于同步原语的更多资料可以在文献(Andrews 2000)中找到。

3. 多计算机操作系统

供多计算机系统使用的操作系统与多处理器操作系统在结构上截然不同，它比多处理器操作系统具有更高的复杂性。这种差异是由于系统范围内的资源管理所需的数据结构无法放置在物理上共享的存储器中，从而无法方便地共享这些数据结构而造成的。其惟一的通信方法是消息传递。因此，多计算机操作系统的组织结构通常如图 1.14 所示。

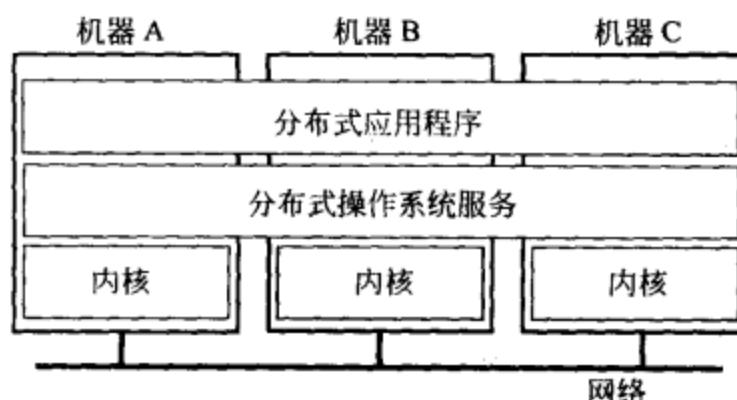


图 1.14 多计算机操作系统的常见结构

每个结点都拥有自己的内核，内核中有负责管理本地资源(比如存储器、本地 CPU、本地磁盘等)的模块。同时，每个结点还包含一个单独的模块，负责对处理器间通信进行处理，也就是向其他结点发送消息，并从其他结点接收消息。

在每个本地内核上面是一个公共软件层，它实现了操作系统作为支持各种任务并行执行和并发执行的虚拟机的功能。实际上，该层甚至可以提供多处理器计算机的抽象，我们将对此进行简短的讨论。换句话说，它提供了共享存储器的软件实现。该层中还实现了另外一些功能，例如，向处理器指派任务、屏蔽硬件故障、提供透明存储和常规进程间通信。也可以说，该层具备了一般操作系统应该具有的所有功能。

不提供共享存储器的多计算机操作系统只能向应用程序提供消息传递功能。不幸的是，消息传递原语的语义在不同系统间可能有着巨大的差异。要说明它们之间的差异，最简单的方法就是考察是否对消息进行了缓冲。另外，需要考虑发送或者接收消息的进程在何时被阻塞。图 1.15 中显示了缓冲和阻塞可能发生的位置。

只可能在两个位置对消息进行缓冲：发送端或者接收端。因此就产生了 4 个可能的同步点(同步点就是发送进程或者接收进程可能发生阻塞的点)。如果在发送端进行缓冲，那么只有在缓冲区满时阻塞发送进程才有意义，这就是图 1.15 中的同步点 S1。另一

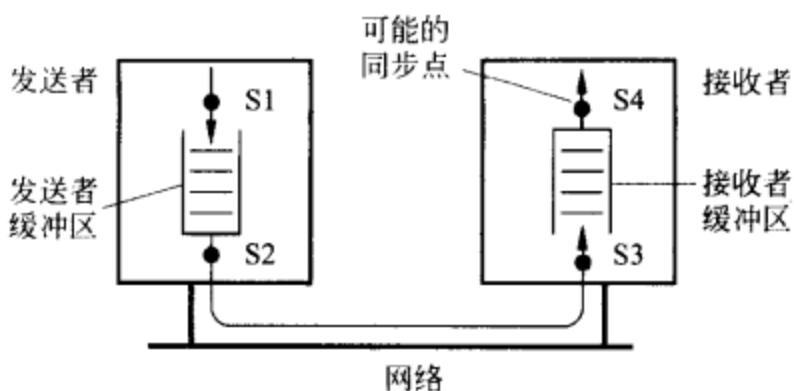


图 1.15 消息传递中可能发生阻塞和缓冲的位置

种方案是,将消息放入缓冲区的操作将会返回一个标志操作是否成功的状态值,这就避免了在缓冲区满的情况下对发送进程进行阻塞。此外,当不存在发送缓冲时,可以在下列三点之一阻塞发送进程:消息已发送(点 S2);消息已到达接收者(点 S3);消息已经交付给接收者(点 S4)。注意,如果阻塞发生在 S2、S3 或者 S4 这三个点中的一个,那么发送者的缓冲就没有任何意义。

对接收进程的阻塞只在同步点 S3 有意义,这种情况只在不存在接收者缓冲或者接收者缓冲为空时才会发生。也可以让接收者不断查询是否有传入的消息,但这会导致 CPU 资源的浪费,或者对传入消息响应过慢,这种过慢的响应可能会使缓冲区溢出,从而不得不丢弃输入的消息(Bhoedjang 等 1998)。

还有一个问题对于理解消息传递语义很重要,这就是通信是否可靠。可靠通信的关键特征是发送者发送消息后,可以从接收者获得消息已收到的确认反馈。在图 1.15 中,这意味着确保所有消息都能够到达同步点 S3。对于不可靠通信来说,没有这样的确认反馈。如果在发送端有缓冲区存在,通信就可能是可靠的,也可能是不可靠的。同样,如果发送进程在 S2 点被阻塞,操作系统就不必保证通信的可靠性。

然而,如果操作系统在消息到达 S3 或者 S4 后才对发送进程进行阻塞,就必须确保通信可靠性,否则可能会处于这样一个尴尬境地:发送者正在等待接收者的确认或者准备进行交付,其间,它的消息却已在传输过程中丢失。阻塞、缓冲以及通信可靠性保证彼此之间的关系总结见图 1.16。

同步点	发送者缓冲	是否确保通信可靠性?
在缓冲区满时阻塞发送进程	是	不必要
阻塞发送进程,直到消息发送出去为止	否	不必要
阻塞发送进程,直到消息到达接收者为止	否	必要
阻塞发送进程,直到消息交付给接收者为止	否	必要

图 1.16 阻塞、缓冲和通信可靠性之间的关系

在构建多计算机操作系统方面存在的许多问题对于任何分布式系统来说都是一样重要的。多计算机操作系统和分布式系统的主要区别在于,前者一般假定底层硬件是同构的,而且受到完全控制。然而,许多分布式系统却常常构建在现有操作系统的路上。下面将对此进行简短的讨论。

4. 分布式共享存储器系统

实践证明,多计算机系统的程序设计比多处理器系统的程序设计要困难得多。这是由于,与只有消息传递功能可供使用的情况相比,以访问共享数据和使用诸如信号量和监控器的同步原语的进程表示通信要容易得多。而像缓冲、阻塞、可靠通信等问题将使事情更为糟糕。

由于以上原因,人们做了大量研究,力图在多计算机系统上模拟共享存储器,目标在于提供一个运行在多计算机系统上的共享存储器虚拟机,这样就可以根据共享存储器模型来编写应用程序,但是目前这一目标还没有实现。多计算机操作系统在这里发挥了关键作用。

方法之一是,利用每个结点的虚拟存储器容量来实现对一个巨大的虚拟地址空间的支持。这导致了基于分页的分布式共享存储器(distributed shared memory, DSM)的诞生。基于分页的分布式共享存储器的原理如下:在 DSM 系统中,地址空间被划分为页(大小一般为 4KB 或者 8KB),将各页分配给系统中的各个处理器。当某个处理器引用当前不属于本地页的某个地址时,将引发一个软中断(trap),操作系统会取来包含该地址的存储器页,并且让出错的指令重新执行;由于要求的地址已经可用,该指令将会成功执行。图 1.17(a)进一步说明了这一原理。图中的系统有 16 页的地址空间和 4 个处理器。从本质上来说,这种方法与通常的分页机制没有什么不同,只是用来作为后备存储的不是本地磁盘而是远程 RAM。

在图 1.17 中,如果处理器 1 引用位于第 0、2、5 或 9 页中的指令或者数据,就会在本地完成操作。而对其他各页的引用将会引发软中断。例如,引用第 10 页中的地址会引发一个软中断,并被操作系统捕获,操作系统随即将第 10 页由机器 2 移到机器 1,如图 1.17(b)所示。

对这个基本系统的改进之一是对只读存储器页(比如那些包含程序文本、只读常数或者其他只读数据结构的页)进行复制,这通常可以显著提高性能。例如,如果图 1.17 中的第 10 页存储的是一部分程序文本,处理器 1 对其发出请求将会导致系统将该页的一份拷贝发送给处理器 1,而原来属于处理器 2 的页不变,如图 1.17(c)所示。在这种方式下,处理器 1 和 2 可以分别任意引用第 10 页中的内容,而用不着再引发软中断来获取缺失的存储器。

另一个可能的方法是复制所有的页,而不限于只读页。只要是进行读取操作,复制只读页与复制可读写页实际上没有什么不同。然而,如果对某个页面进行过复制,而对该页面又进行了修改,就必须采取专门的步骤来防止存在该页的多个不一致拷贝。典型情况下,在写操作进行之前,除了要写的拷贝外的所有拷贝都会被系统标记为无效。

如果放松对复制页面之间的严格的一致性要求,也就是说,允许某一个拷贝与其他拷贝暂时不同,可以取得进一步的性能提升。实践表明,这种方法确实有效,但不幸的是,这可能导致程序员的工作更加困难,因为他必须时刻注意到这种不一致性。考虑到开发 DSM 系统的首要原因就是为了给程序设计带来方便,因此削弱一致性是不可取的。我们将在第 6 章中继续讨论一致性问题。

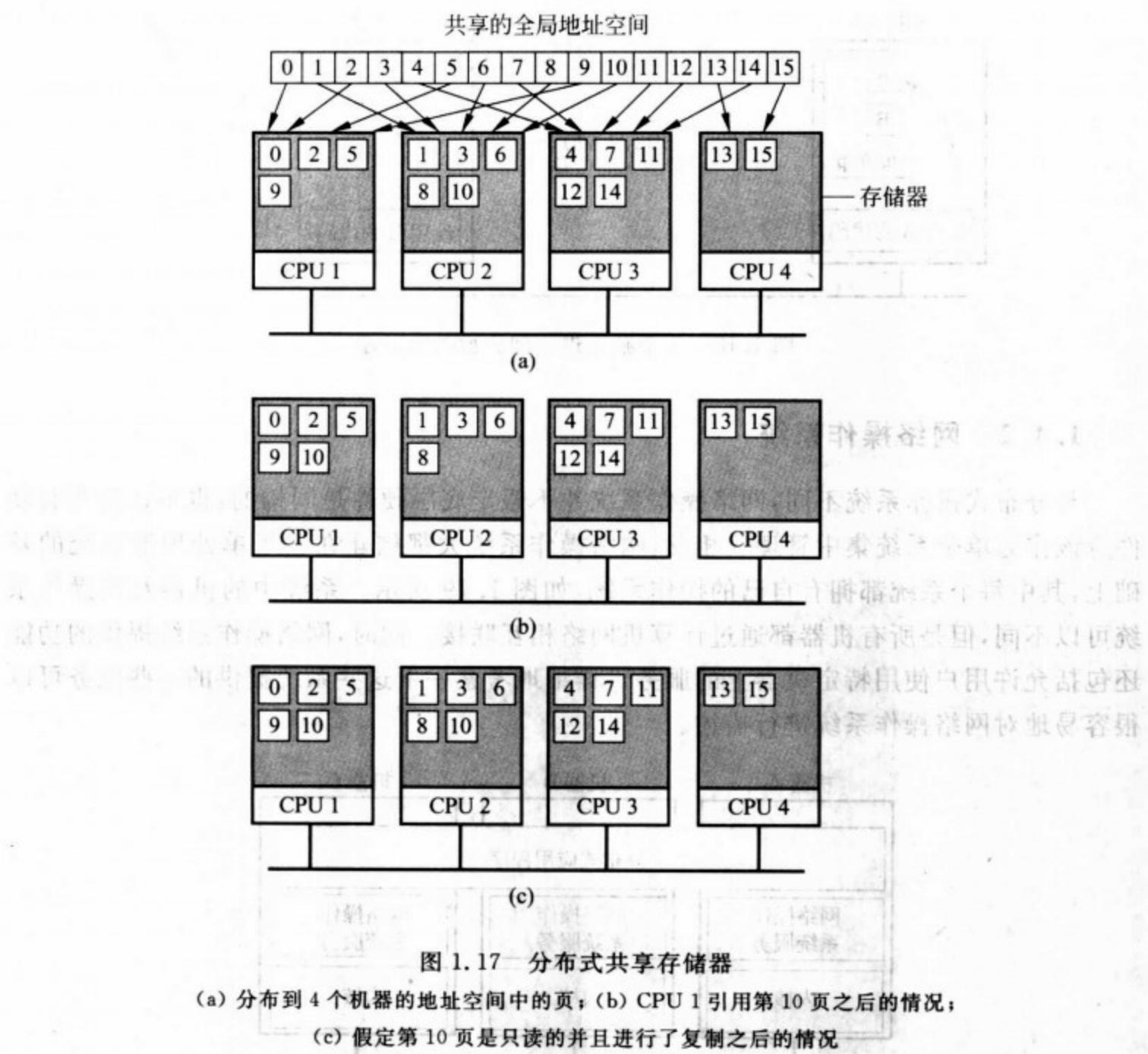


图 1.17 分布式共享存储器

- (a) 分布到 4 个机器的地址空间中的页;
- (b) CPU 1 引用第 10 页之后的情况;
- (c) 假定第 10 页是只读的并且进行了复制之后的情况

在设计高效的 DSM 系统方面存在另一个问题,就是如何决定页的大小。这与决定单处理器虚拟存储器系统中页面大小时所遇到的折衷问题相似。例如,通过网络传输某一页的代价主要取决于建立传输的代价,而不是所传输的数据量。因而,如果页面较大,那么在经常需要访问大量连续数据的情况下可以减小传输的总次数。另一方面,如果某页含有不同处理器上的两个独立进程使用的数据,操作系统可能会不断地在这两个处理器间传输该页,如图 1.18 所示。将分别属于两个独立进程的数据放在同一页内的行为称为伪共享(false sharing)。

对分布式共享存储器的研究已经长达 15 年。目前,DSM 的研究人员还在竭力想把高效率和可编程性结合到系统中去。为了在大规模多计算机系统上获得高性能,程序员们仍然采用消息传递机制,尽管它与(虚拟)共享存储器系统相比具有更高的复杂性。这样我们似乎可以得出如下结论:用于高性能并行编程的 DSM 无法实现它的预定目标。如果想要了解关于 DSM 的更详细资料,请参见文献(Protic 1998)。

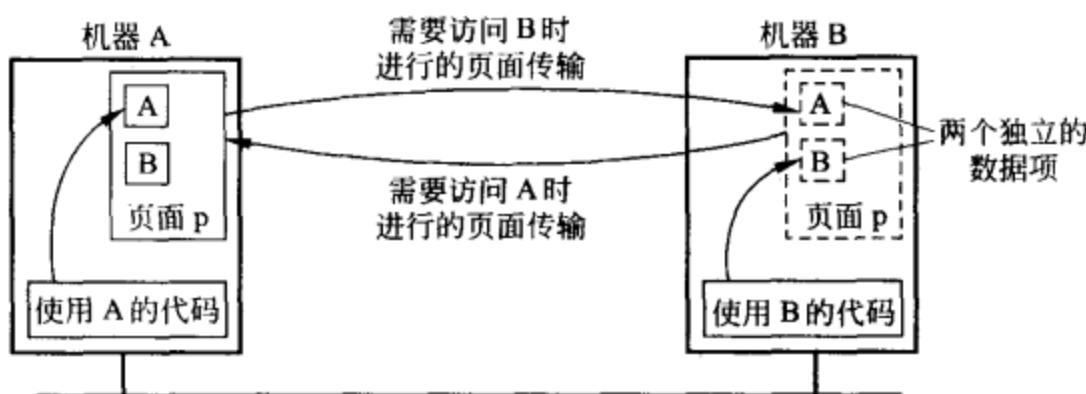


图 1.18 两个独立进程间页面的伪共享

1.4.2 网络操作系统

与分布式操作系统不同，网络操作系统并不假定底层硬件是同构的，也不认为所有硬件应该作为单个系统集中管理。相反，网络操作系统大都构建在一组单处理器系统的基础之上，其中每个系统都拥有自己的操作系统，如图 1.19 所示。系统中的机器及其操作系统可以不同，但是所有机器都通过计算机网络相互联接。同时，网络操作系统提供的功能还包括允许用户使用特定机器上的服务。详细地考察一下这些系统提供的一些服务可以很容易地对网络操作系统进行描述。

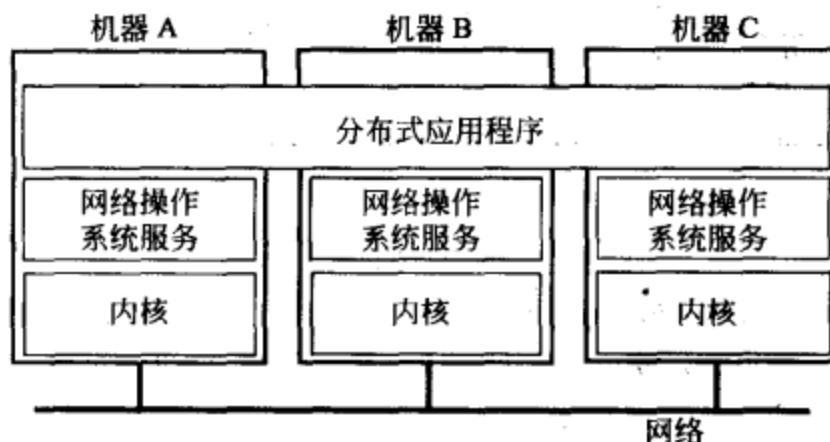


图 1.19 网络操作系统的一般结构

网络操作系统提供的一项常见服务是允许用户使用下面的命令远程登录到另一台机器上：

```
rlogin machine
```

这个命令执行的结果是将用户自己的工作站变为一台登录到远程机器上的远程终端。假定用户使用的是图形工作站，用键盘输入的命令将发送到远程机器，随后远程机器的回应将显示在用户屏幕上的一个窗口中。如果要切换到另一台远程机器，必须先打开一个新窗口，然后使用 rlogin 命令来连接到另一台机器。因而机器的选择完全是通过手动完成的。

网络操作系统还提供了远程复制命令，用来把文件从一台机器复制到另一台机器。例如：

```
rcp machine1,file1 machine2,file2
```

上面这条命令将会把文件 file1 从机器 machine1 复制到机器 machine2，并且将复制到 machine2 上的文件命名为 file2。在这里，文件的移动过程同样是显式的，要求用户完全掌握文件所在位置以及命令执行位置。

虽然有这种通信方式总比没有好，但是它实在太原始了，系统设计者开始寻找更为便捷的通信与信息共享的方式。一种方法是提供一个共享的可供所有工作站访问的全局文件系统，该文件系统由一台或数台称为文件服务器(file server)的机器支持。文件服务器接受来自其他(非服务器)机器(这些机器称作客户, client)上运行的用户程序的请求，对文件进行读写访问。服务器对每一个接收到的请求进行检查并加以执行，完成后将应答发送给请求者，如图 1.20 所示。

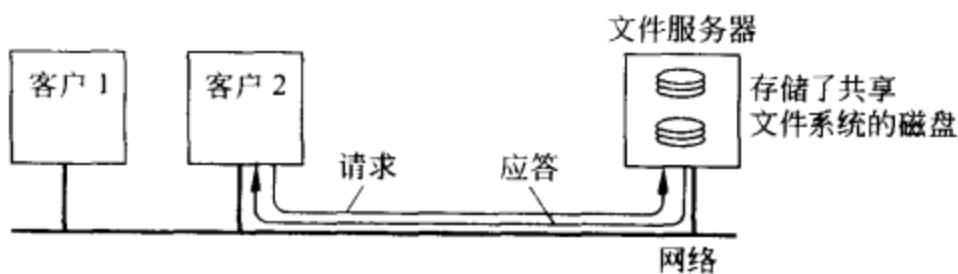


图 1.20 位于网络操作系统中的两个客户和一个服务器

文件服务器通常维护着多个分层文件系统，每个文件系统中都有一个根目录，根目录下有子目录和文件。工作站可以导入或者挂载(mount)这些文件系统，用服务器上的文件系统来扩充本地文件系统。例如，图 1.21 中显示了两台文件服务器，其中一台拥有一

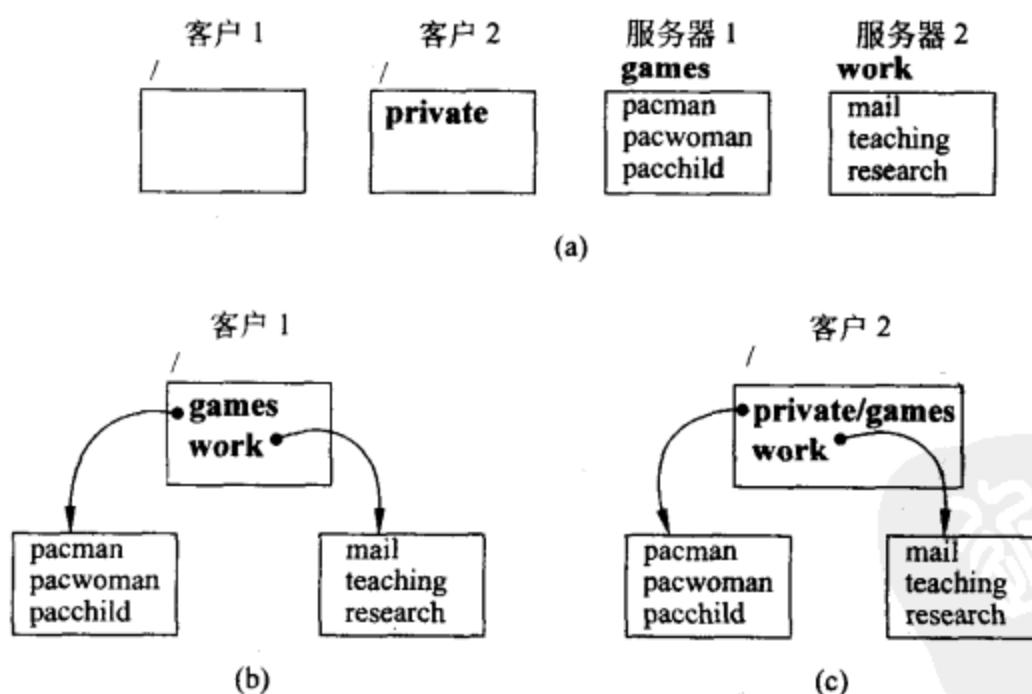


图 1.21 不同的客户可能把服务器中的文件系统安装到不同位置

个名叫 games 的目录，另一台有一个名叫 work 的目录(目录名以粗体字显示)，这两个目录中都含有若干文件。图中的两个客户都安装上了分别属于两台服务器的两个目录，但是它们把这些目录安装在本机文件系统中不同的位置。客户 1 将服务器目录安装在自己

的根目录下,可以用/games 和/work 来分别访问这两个目录。而客户 2 把目录 work 安装到自己的根目录下,但是考虑到像玩游戏这样的事要保密,于是就创建了一个名为/private 的目录,然后把 games 目录安装到该目录中。因此,可以使用路径 /private/games/pacwoman 来访问 pacwoman 文件,玩“吃豆豆”游戏,而不是使用路径/games/pacwoman。

客户把服务器上的文件系统安装到本地目录里哪一层一般是无关紧要的,但是必须注意到这会造成文件系统在不同的客户端看起来不同。文件的名称与访问该文件的机器以及该机器上文件系统的建立方式有关。由于所有客户机器都以彼此相对独立的方式运行,所以无法保证它们对程序呈现出相同的目录层次结构。

网络操作系统无疑比分布式操作系统更原始。这两种操作系统之间的主要区别在于,分布式操作系统做了相当大的努力以实现完全的透明性,也就是说,力图给出一种单一系统视图。

由于网络操作系统中缺乏透明性,造成了一些明显的缺陷。例如,这种系统难以使用,要求用户显式地登录进远程机器,或者显式地把文件从一台机器复制到另一台机器。管理上也存在问题,由于网络操作系统中的所有机器是彼此独立的,通常只能对它们进行独立管理。结果是,用户只有在机器 X 上拥有账号才能登录到 X 上。同时,如果用户想在所有机器上使用同一个密码,对这个密码的改动就会导致要求在所有机器上显式修改该密码。同样的道理,对所有访问许可的维护通常是逐台机器进行的。一旦访问许可各处一致,就没有简单的方法来对访问许可进行修改,只能逐个进行。这种分散的安全方法有时会导致网络操作系统难以防范恶意攻击。

网络操作系统与分布式系统相比也有一些优势。由于网络操作系统的结点是彼此高度独立的,添加或者删除机器非常方便。在某些情况下,要添加一台机器所需要做的工作只是把机器连接到公用网络,然后使网络中其他机器知晓这台新机器的存在。在 Internet 内添加一台新服务器的过程正是如此。为了让 Internet 上的机器了解到新服务器的存在,需要给出它的网络地址。更好的办法是给新服务器取一个象征性的名字,再把这个名字和它所对应的网络地址一并添加到 DNS 中。

1.4.3 中间件

无论是分布式操作系统还是网络操作系统都不能真正符合 1.1 节中给出的分布式系统的定义。分布式操作系统并不是用来管理一组“独立”的计算机的,而网络操作系统也没有提供“单个一致的系统”这样的视图。问题来了,是否可能开发出集这两种系统的优点于一身的分布式系统呢?最好既能够具有网络操作系统的可扩展性和开放性,又能够具有分布式操作系统的透明性和与之相关的易用性。该问题的解决方案可以通过在网络操作系统中使用的附加软件层来实现,这种附加软件层用来或多或少地隐藏网络操作系统中底层平台集合的异构性,并且提高分布透明性。许多现代的分布式系统都是通过使用这种附加软件层实现的。该附加层称为中间件(middleware)。在本节中,我们将阐明中间件的一些特性,对中间件的组成进行详细探讨。

1. 中间件所处的地位

很多分布式应用程序直接使用网络操作系统提供的编程接口。例如,通信常常是通过对套接字的操作进行的,套接字允许位于不同机器上的进程互相传递消息(Stevens 1998)。此外,应用程序常常利用本地文件系统接口。就像我们说过的那样,这种方法的问题在于使得分布难以做到透明。解决方法是在应用程序和网络操作系统之间放置一个附加的软件层,该软件层提供比操作系统更高层次的抽象。这种软件层位于应用程序和网络操作系统之间,因此被称为中间件(middleware),如图 1.22 所示。

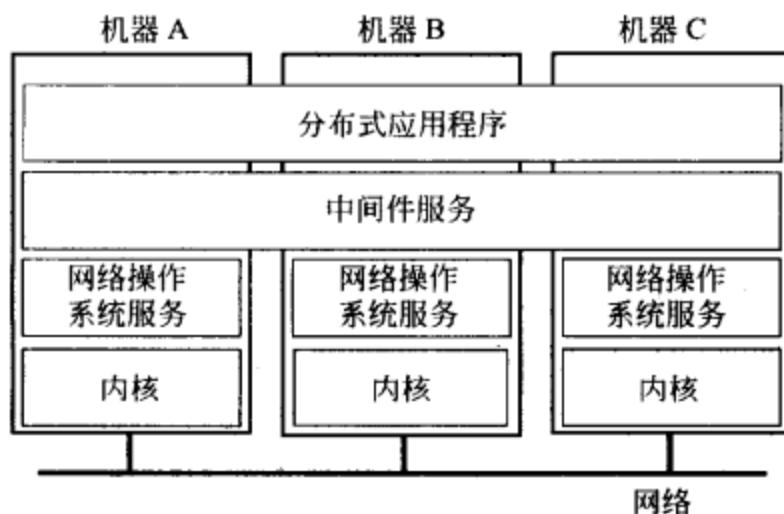


图 1.22 以中间件形式组织的分布式系统的常见结构

每个本地系统都是底层网络操作系统的一部分,它们被假定提供连接到其他计算机的通信手段,还负责进行本地资源管理。也就是说,中间件并不对结点本身进行管理,结点的管理工作完全由本地操作系统负责。

中间件系统的一个重要目标是对应用程序隐藏底层平台的异构性,因此许多中间件系统都提供一组完整程度不同的服务集。这些服务必须使用系统提供的接口来访问,除此以外没有别的访问方法。也就是说,一般禁止跳过中间件层直接调用底层操作系统的服务。我们将很快回到中间件服务这个主题上面来。

注意到以下事实是很有趣的:中间件并不是作为理论上达到分布式透明性的措施而发明的。在网络操作系统诞生并且得到广泛应用之后,许多组织发现他们有许多网络应用程序无法方便地集成到单个系统中去(Bernstein 1996)。此时,厂商开始在其系统中构建高层的独立于应用程序的服务。这方面的典型例子包括对分布式事务和高级通信功能的支持。

当然,要就中间件究竟应该是什么样的这个问题达成一致不容易。解决这个问题的一种方法是建立一个组织,由它来制定中间件解决方案的通用标准。现在已经有了很多这样的标准,但它们彼此之间通常互不兼容,更有甚者,来自不同厂商的遵循同一个标准的产品也很少能够协同工作。当然,用不了很久就会有人提供“上层件”来解决这个问题。

2. 中间件模型

为了使分布式应用程序的开发和集成尽可能简单,多数中间件是基于某种模型(或称范型)来对分布及通信进行描述的。其中一种相对简单的模型是将任何东西都作为文件来处理。这是一种在 UNIX 中率先采用的方法,在第 9 计划(Plan 9)中得到严格遵守(Pike 等 1995)。在第 9 计划中,包括 I/O 设备(比如键盘、鼠标、磁盘、网络接口等)在内的所有资源都作为文件来处理。文件在本地还是在远程从本质上来说并没有区别。应用程序打开文件,读出或写入一定数量的字节,然后将文件关闭。由于文件可以由多个进程共享,只须通过访问同一个文件就可以实现通信。

以分布式文件系统(distributed file system)为中心的中间件所采用的方式与第 9 计划中的方式相近,但没有那么严格。在许多情况下,从只对传统文件(也就是只用于存储数据的文件)支持分布透明性这个意义上来说,这样的中间件实际上只比网络操作系统前进了一小步。例如,仍然要求进程在特定机器上显式启动。基于分布式文件系统的中间件已得到证明具备一定的扩展能力。正是这个优势使它流行起来。

另一种重要的早期中间件模型是基于 RPC(remote procedure call,远程过程调用)的。在这种模型中,重点在于通过允许进程调用位于远程机器上的过程实现来隐藏网络通信。在进行这样的过程调用的时候,调用参数被透明地传输到远程机器上,随后在远程机器上执行该过程,执行完毕后将结果发送给调用者,因此看起来过程调用像在本地执行的一样,除了也许有一定的性能损失之外,发出调用的进程并不知道有网络通信发生。我们将在下一章中讨论远程过程。

随着面向对象的编程方式越来越流行,很明显,如果过程调用可以超越机器之间的界限,那么以一种透明的方式调用驻留在远程机器上的对象也应该是可以做到的。这种想法导致很多中间件系统提供一种分布式对象(distributed object)的概念。分布式对象本质上是实现了一种特殊接口的对象,这种接口向对象的用户隐藏了对象的所有内部细节。接口包含而且只包含由对象实现的方法。进程只能看到对象的接口。

分布式对象的实现方式常常是把每个对象本身放置在一台机器上,并允许在其他机器上使用对象的接口。当某个进程调用一个方法时,进程所在机器上的接口实现只是简单地把方法调用转换成一个消息,再把该消息发送给对象。该对象执行所请求的方法,将执行结果送回给接口实现。接口实现再将收到的应答消息转换成结果,把结果交给进行调用的进程。就像使用 RPC 一样,进程本身并不知道进行了网络通信。

万维网例子最有力地证明了模型给网络系统使用所带来的简化。Web 的成功主要应该归功于分布式文档(distributed document)这种极为简单但却高效的模型。在 Web 的这种模型中,信息被组织成文档,每个文档驻留在世界上某处位置透明的机器上。文档中包含有指向其他文档的链接。通过打开某个链接,可以从该链接指向的文档所在的位置将文档取回,并把它显示在用户屏幕上。文档的概念并不限于基于文本的信息。比如,Web 也支持音频和视频文档,以及各种基于图形的交互式文档。

我们将在本书的后半部分继续对中间件范型进行详细讨论。

3. 中间件服务

很多中间件系统通常提供多种通用的服务。不管采取的是何种方式,各种中间件都有一个共同的目标,那就是力图实现访问透明性。这个目标是通过提供高层通信功能(communication facility)以隐藏通过计算机网络进行的低层消息传递来达到的。这些功能彻底取代了由网络操作系统提供的传输层程序设计接口。支持通信的具体方式随中间件系统向用户和应用程序提供的分布模型的不同而有较大差异。我们已经提到过远程过程调用和分布式对象调用。另外,许多中间件系统;比如分布式文件系统和分布式数据库,还提供对远程数据的透明访问功能。像 Web 那样以透明的方式取回文档是高层(单向)通信的另一个例子。

中间件提供的一种常见服务是命名(naming)服务。命名服务允许实体被共享和查询(就像在目录中一样),这种服务可以比作电话簿和黄页。虽然命名服务乍看起来好像很简单,但是考虑到可扩展性困难就来了。问题是这样出现的:在大规模系统中,如果要高效地进行名字查询,就必须假设命名的实体所在地址是固定的。在万维网中就做了这个假设,其中的文档目前是用 URL 命名的。URL 中含有该 URL 指向的文档所在的服务器名,因此如果文档移到了另一台服务器上,它原来的 URL 就失效了。

许多中间件系统提供了存储方面的特殊功能,这些功能也称为持久性(persistence)。如果采用最简单的形式,持久性就可以由分布式文件系统来提供,但是更高级的中间件已经在本身的系统中集成了数据库,或者提供了将应用程序连接到数据库的工具。

在数据存储发挥重要作用的环境中通常提供一些分布式事务(distributed transactions)的功能。事务的重要属性之一就是,它允许多个读写操作以原子方式进行。原子性意味着事务要么成功(在这种情况下所有的写操作可以得到实际执行),要么失败(在这种情况下事务涉及到的所有数据都不受影响)。分布式事务用于对分散在多台机器上的数据进行操作。特别是在面临要求实现故障屏蔽这一分布式系统难以做到的功能时,很有必要提供分布式事务这样的服务。不幸的是,分布在多个本地机器上的事务难于进行扩展,更不用说分布到地域上分散的机器上了。

最后,几乎所有用于非实验性环境的中间件系统都提供安全功能。与网络操作系统相比,中间件存在的安全问题更加普遍。在原理上,中间件层不能依赖底层的本地操作系统来提供对整个网络安全的支持。因此,部分安全功能必须由中间件层本身进行重新实现。由于必须与系统扩展性方面的要求结合起来考虑,安全性已成为分布式系统中最难实现的服务之一。

4. 中间件与开放性

现代分布式系统一般构建成适合于多种操作系统的中间件。在这种方式下,为某个特定分布式系统建立的应用程序就与操作系统无关了。不幸的是,这种无关性常常演变为对特定中间件的强依赖性。产生这个问题的原因是中间件的开放性往往不像开发商宣称的那么强。

就像前面介绍过的那样,真正开放的分布式系统是由完整的接口定义进行说明的。

完整意味着实现系统所需的一切内容都确实得到了说明。接口定义的不完整性将会导致系统开发人员被迫添加自己的接口，结果就会发生这种情况：两个遵循同一标准的开发组各自开发了一个中间件系统，然而为其中一个系统编写的应用程序无法方便地移植到另一个系统上去。

同样糟糕的还有这样的情况：接口定义的不完整性导致两种不同的实现无法互操作，尽管它们实现了完全相同的一组接口，不同的只是底层协议。例如，如果两种不同的实现都依赖于各自底层网络操作系统提供的互不兼容的通信协议，就不能指望能实现互操作性。必须采用相同的中间件协议和中间件接口，如图 1.23 所示。

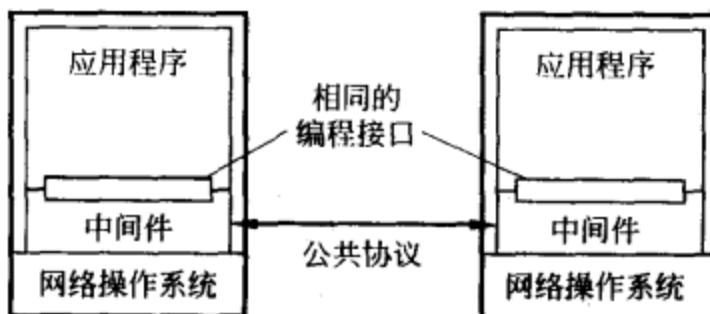


图 1.23 在一个基于中间件的开放分布式系统中，各中间件层所使用的协议及其向应用程序提供的接口必须相同

另一个例子是，为了确保不同实现间的互操作性，必须以相同的方式来引用不同系统中的实体。如果某个系统中的实体通过 URL 来引用，而其他系统中的实体却通过网络地址来引用，这种交叉引用就会出问题。在这种情况下，接口定义必须准确规定引用实体的形式。

5. 各种系统的比较

分布式操作系统、网络操作系统以及基于中间件的分布式系统之间的简单比较如图 1.24 所示。

项 目	分布式操作系统		网 络 操 作 系 统	基 于 中 间 件 的分 布 式 系 统
	多 处 理 器 系 统	多 计 算 机 系 统		
透明度	非常 高	高	低	高
所有结点使用的操作系统是否相同	是	是	否	否
操作系统的拷贝数目	1	无	无	无
通信基于何种实体	共享存储器	消息	文件	取决于特定的模型
资源管理	资源是全局的，集中管理	资源是全局的，分布管理	各结点自行管理	各结点自行管理
可扩展性	否	部分	是	各系统不同
开放性	封闭的	封闭的	开放的	开放的

图 1.24 多处理器操作系统、多计算机操作系统、网络操作系统和基于中间件的分布式系统之间的比较

在透明性方面,很明显分布式操作系统比网络操作系统要好。在多处理器系统中只须对存在多个 CPU 这个事实加以隐藏,这是相对容易的。难点在于要隐藏存储器的物理分布,这就是难以构建支持完全分布透明性的多计算机操作系统的原因所在。分布式系统常常通过采用用于分布和通信的特定模型来提高透明性。例如,分布式文件系统一般可以有效地隐藏文件所在位置和对文件访问的差异性。然而,这种系统的通用性会有所损失,因为用户被迫按照特定模型的要求表示所有内容,有时这会导致某些应用程序的实现不那么方便。

分布式操作系统是同构的,这意味着每个结点运行相同的操作系统(内核)。在多处理器系统中,不需要对表之类的内容进行复制,因为这些内容可以在主存储器中共享。在这种情况下,所有通信都通过主存储器进行,而在多计算机操作系统中则使用消息来完成通信。在网络操作系统中,可以说通信几乎完全是基于文件来完成的。例如,在 Internet 上多数通信是通过文件传输完成的。然而,像电子邮件系统和电子公告栏这样的高层消息传递机制也得到了广泛应用。基于中间件的分布式系统中的通信方式主要取决于系统采用的特定模型。

网络操作系统和分布式系统中的资源是由每个结点分别管理的,这就使得它们相对容易扩展。然而,实践表明,在分布式系统中实现的中间件层的可扩展性一般很有限。分布式操作系统要负责进行全局资源管理,因此它难以扩展。由于多处理器系统采用集中式方案(管理所需的全部数据都放在主存储器中),它一般也是难以扩展的。

最后,如果要比较开放性的话,网络操作系统和分布式系统则更胜一筹。通常说来,结点都支持某种标准通信协议(比如 TCP/IP),这样就很容易实现互操作性。然而,如果使用了多种操作系统,在应用程序移植方面可能会存在一系列问题。分布式操作系统一般不设计为开放的,它们为了提高性能经常进行优化,结果产生了多种专用的解决方案。这些方案会影响系统的开放性。

1.5 客户-服务器模型

到目前为止,我们还几乎没有谈到分布式系统的实际组织结构。分布式系统组织结构的核心是系统中的进程组织问题。尽管关于分布式系统的许多问题众说纷纭,但是许多研究人员和业内人士都赞同这样一个观点:以客户向服务器请求服务这样一种模式来进行思考将会有助于理解和管理分布式系统的复杂性。在本节中,我们将对客户-服务器模型进行详细探讨。

1.5.1 客户与服务器

在基本的客户-服务器模型中,可以将分布式系统内的进程划分为两组(它们之间可能有相互重叠的部分)。服务器(server)是实现某个特定服务的进程。这种服务的例子包括文件系统服务和数据库服务。客户(client)是向服务器请求服务的进程,它向服务器发送一个请求,随后等待服务器的应答。这种客户-服务器的交互也称为请求-应答行为,如图 1.25 所示。

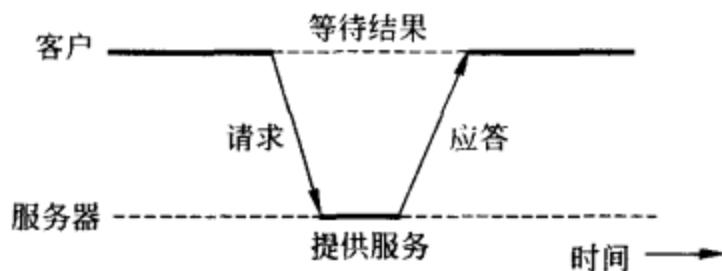


图 1.25 客户与服务器之间的一般交互

如果底层网络非常可靠(如在多数局域网中)，客户与服务器之间的通信可以利用简单的非连接协议来实现。在这种情况下，如果客户想请求某个服务，只要打包一条消息，在消息中写明它需要的服务以及必要的输入数据，然后将打包的消息发送给服务器。服务器在收到请求之后对其进行处理，然后将处理结果打包成一条应答消息，将该应答消息送回客户。

使用非连接的协议有一个明显的好处，即效率高。只要消息不丢失也没有遭到破坏，上面描述的请求/应答协议就能够工作得很好。不幸的是，使协议能够处理偶然的传输故障不是一件简单的事情。我们所能做的可能只是让客户在未收到应答消息的情况下重新发送请求。然而问题在于，客户无法探知究竟是原来的请求消息丢失了，还是应答消息的传输出现了问题。如果是应答消息丢失了，那么重新发送请求将导致相同的操作被执行两次。如果这个操作类似于“从我的银行账户转出 10 000 美元”，那么，无疑更好的办法还是报告发生了一个错误，而不是重新发送请求。另一方面，如果这个操作是“请告知我的账户还有多少钱”，那么重新发送请求当然是可以接受的。不难看出，这个问题没有一定的解决方法。我们将在第 7 章中详细讨论传输故障的处理。

还可以选择另一种方法：许多客户-服务器系统采用可靠的面向连接的协议。虽然把这种性能相对较低的协议用在局域网内并不是很合适，但是在通信不可靠的广域网系统中，这种协议工作得非常好。例如，几乎所有 Internet 应用协议都是基于可靠的 TCP/IP 连接的。这种情况下，当客户请求服务的时候，它首先与服务器之间建立一个连接，然后再发送请求。服务器一般使用同一个连接把应答消息送回给客户，然后断开该连接。问题是建立连接和断开连接的代价相对来说较高，特别是在请求消息和应答消息的尺寸相对较小时更是这样。我们将在下一章中讨论另一个解决方法，该方法将连接管理与数据传输结合在一起进行。

客户与服务器模型示例

为了对客户和服务器的工作原理有一个更加深入的了解，本节中我们将用 C 语言简单地实现一个客户和一个文件服务器。客户和服务器都需要共享一些定义，我们把这些定义放到 header.h 文件中，该文件的内容如图 1.26 所示。客户和服务器的源文件需要用到这些定义，因而都必须包含下面这个语句：

```
#include <header.h>
```

这个语句可以让编译预处理器将 header.h 中的全部内容加进源程序中,然后编译器再对源程序进行编译。

```
/* 客户和服务器需要的定义 */
#define TRUE          1
#define MAX_PATH      255    /* 文件名的最大长度 */
#define BUF_SIZE      1024   /* 每次传输的数据量 */
#define FILE_SERVER   243    /* 文件服务器的网络地址 */

/* 定义允许的操作 */
#define CREATE        1      /* 创建一个新文件 */
#define READ          2      /* 从一个文件中读取数据并返回之 */
#define WRITE         3      /* 向一个文件中写入数据 */
#define DELETE        4      /* 删除一个已有的文件 */

/* 错误代码 */
#define OK            0      /* 操作正确完成 */
#define E_BAD_OPER    -1     /* 请求了未知操作 */
#define E_BAD_PARAM   -2     /* 参数错误 */
#define E_IO           -3     /* 磁盘错误或其他 I/O 错误 */

/* 消息格式的定义 */
struct message {
    long source;           /* 发送者的标识 */
    long dest;             /* 接收者的标识 */
    long opcode;           /* 请求的操作 */
    long count;            /* 传输的字节数 */
    long offset;           /* 文件中 I/O 操作的开始位置 */
    long result;           /* 操作结果 */
    char name[MAX_PATH];  /* 待操作文件的名称 */
    char data[BUF_SIZE];  /* 读取或写入的数据 */
}
```

图 1.26 供客户和服务器源文件使用的 header.h 文件

我们先看一下 header.h 文件,它以两个常量定义开始。这两个常量是 MAX_PATH 和 BUF_SIZE,它们定义了消息中包含的两个数组的大小。MAX_PATH 指定文件名(即类似/usr/ast/books/opsys/chapter1.t 这样的带路径名的文件名)最长可以包含多少个字符。而 BUF_SIZE 指定缓冲区的大小,它决定了每次操作可以读出或者写入的数据量。常量 FILE_SERVER 代表文件服务器的网络地址,客户可以根据它来向服务器发送消息。

第二组常量定义了与各种操作对应的编号值。这些定义可以确保各种操作所对应的编号对客户和服务器都是相同的,比如代表读操作的编号都为 2,代表写操作的编号都为 3,等等。我们在这里只列出了 4 种操作,在实际的系统中一般会更多。

每个应答消息都包含一个结果码。如果操作成功完成,结果码中通常包含有用的信息(比如实际读出的字节数)。如果没有值需要返回(比如创建了一个文件),则返回值OK。如果由于某种原因致使操作没能成功完成,结果码中会包含代表操作失败原因的数值,比如 E_BAD_OPER 或 E_BAD_PARAM 等。

最后,我们看一下 header.h 中最重要的部分,即对消息的定义。本例中的消息是一个由 8 个字段组成的结构,客户向服务器发送的所有消息都使用这一格式,服务器的应答消息也一样。在实际系统中,消息的格式也许并不固定(因为不是在所有情况下都需要结构中的所有字段),这里这样做是为了使说明更加简单。source 和 dest 字段分别用来标识消息的发送者和接收者;opcode 字段的值是上面定义的各种操作之一,也就是 create(创建)、read(读取)、write(写入)和 delete(删除)中的一种;count 和 offset 字段中保存的是请求附带的参数;result 字段不是在客户向服务器发出请求时使用的,而用来保存服务器发给客户的应答消息中包含的结果值。最后还有两个数组,第一个是 name 数组,存放要访问的文件的名字,第二个是 data 数组,在进行读操作时用于存放从服务器应答消息中读出的数据,而在进行写操作时用于存放写入的发送数据。

现在看一下图 1.27 中列出的代码。图 1.27(a)是服务器的代码;图 1.27(b)是客户的代码。服务器的代码很简单。主循环由对 receive 的调用开始,该调用用来接收请求消息。调用的第一个参数指明了调用者地址,第二个参数指向用于存储传入消息的缓冲区。receive 过程会阻塞服务器进程,直到有消息到达才将其释放。消息到达之后服务器进程将会继续执行,根据 opcode 的值进行处理,不同的 opcode 值会导致调用不同的过程。传入的消息内容和为发送消息准备的缓冲区作为参数传递给这些过程。过程分析传入的消息 m1,生成应答消息并将其放在 m2 中,同时还把返回的函数值放在 result 字段中送回给发送者。在应答消息发送完毕之后,服务器回到循环开头,再次执行 receive 调用,等待下一个传入的消息。

图 1.27(b)中的代码实现了一个通过服务器复制文件的过程。该过程由一个循环组成,在每次循环中读出源文件中的一个块并将它写入到目标文件中,循环反复执行,直到源文件全部复制完毕为止。read 过程的返回值会是 0 或者负值,可以根据这一点来判定文件是否已经全部复制完毕。

循环中的开始的一部分代码用于生成代表读操作的消息,并将该消息发送到服务器。在接收到应答信息之后,循环中后面那部分代码开始执行,以写操作的方式将接收到的数据发送回服务器,并写到目标文件中去。图 1.27 中的程序是非常粗略的,省略了非常多的细节。例如,没有给出 do_xxx 过程(就是那些实际完成工作的过程)的内容,也没有进行错误检查。但是它们确实清晰地体现了客户与服务器交互的总体思想。在下面几节里,我们将详细讨论客户-服务器模型在组织结构方面的一些问题。

```

#include <header.h>
void main(void) {
    struct message m1, m2;           /* 输入的消息和发出的消息 */
    int r;                          /* 结果码 */
    while(TRUE) {                   /* 服务器一直运行 */
        receive(FILE_SERVER, &m1);   /* 阻塞,等待消息的到来 */
        switch(m1.opcode) {          /* 根据请求的类型进行分派 */
            case CREATE:             r = do_create(&m1, &m2); break;
            case READ:                r = do_read(&m1, &m2); break;
            case WRITE:               r = do_write(&m1, &m2); break;
            case DELETE:              r = do_delete(&m1, &m2); break;
            default:                  r = E_BAD_OPER;
        }
        m2.result = r;               /* 向客户返回结果 */
        send(m1.source, &m2);       /* 发送应答消息 */
    }
}

```

(a)

```

#include <header.h>
int copy(char *src, char *dst){           /* 通过服务器复制文件的过程 */
    struct message m1;                  /* 消息缓冲区 */
    long position;                     /* 当前的文件位置 */
    long client = 110;                 /* 客户的地址 */
    initialize();                      /* 为执行做准备 */
    position = 0;
    do{
        m1.opcode = READ;              /* 读操作 */
        m1.offset = position;         /* 文件中的当前位置 */
        m1.count = BUF_SIZE;          /* 要读取的字节数 */
        strcpy(&m1.name, src);       /* 复制要读取的文件的名称 */
        send(FILESERVER, &m1);       /* 向文件服务器发送消息 */
        receive(client, &m1);         /* 阻塞,等待响应 */

        /* 将刚接收到的数据写入目标文件 */
        m1.opcode = WRITE;            /* 写操作 */
        m1.offset = position;         /* 文件中的当前位置 */
        m1.count = m1.result;         /* 要写的字节数 */
        strcpy(&m1.name, dst);       /* 复制要写入的文件的名称 */
        send(FILE_SERVER, &m1);      /* 向文件服务器发送消息 */
        receive(client, &m1);         /* 阻塞,等待响应 */
        position += m1.result;        /* m1.result 是写入的字节数 */
    } while(m1.result > 0);           /* 反复执行,直至文件复制完毕 */
    return(m1.result >= 0 ? OK : m1.result); /* 返回 OK 或错误码 */
}

```

(b)

图 1.27 一个服务器和客户示例

(a) 一个服务器; (b) 一个通过该服务器复制文件的客户

1.5.2 应用程序的分层

在客户-服务器模型上存在许多争论。其中一个主要问题就是,如何清楚地区分客户和服务器。有时候,服务器和客户之间的界限很模糊。例如,用于分布式数据库的服务器需要将它收到的请求转发给负责实现数据库表的多台文件服务器,此时它又成了客户。在这种情况下,数据库服务器本身在本质上不过是一个发出查询的进程。

然而,考虑到很多客户-服务器应用程序的目标是为用户访问数据库提供支持,因此许多人提议从以下三个层次进行区分:

- (1) 用户界面层
- (2) 处理层
- (3) 数据层

用户界面层包含与用户直接交互所需的全部内容,比如显示管理。处理层一般包含应用程序。数据层包含要处理的实际数据。在以下各节中,我们将分别对这些层次进行讨论。

1. 用户界面层

一般由客户实现用户界面层。该层由允许最终用户与应用程序之间进行交互的程序组成。不同用户界面程序的复杂程度差异很大。

最简单的用户界面程序是基于字符的屏幕界面。这种界面通常用于大型机环境。由于大型机控制了包括键盘和监视器在内的所有交互设备,这种环境几乎称不上是客户-服务器环境。然而在多数情况下,用户终端可以进行回显键入字符之类的本地处理;或者为类似表单的界面提供支持。在这样的界面中可以对完整的条目进行编辑,然后再将其发送到主机。

现在,即使是在大型机环境中也可以看到比较先进的用户界面。客户的机器一般至少提供图形显示界面,使用弹出式或者下拉式菜单,许多屏幕控件是通过鼠标而不是用键盘来控制的。这种界面的典型例子包括许多 UNIX 环境中使用的 X-Windows 界面,以及为 MS-DOS PC 机和 Macintosh 苹果机开发的用户界面。

现代用户界面都提供了更加强大的功能,允许不同应用程序共用同一个图形窗口,并且在该窗口中根据用户的操作进行数据交换。例如,如果要删除某个文件,通常只须把代表该文件的图标移到代表垃圾箱的图标上就行了。还有多种字处理器允许用户使用鼠标将文档中的某段文本移动位置。我们将在第 3 章中继续讨论用户界面问题。

2. 处理层

许多客户-服务器应用程序大致由三部分构成:一部分与用户交互,另一部分操作数据库或者文件系统,还有一个中间部分包含应用程序的核心功能。这个中间部分在逻辑上位于处理层。与用户界面和数据库不同,处理层并没有太多共同之处。因此,我们多举几个例子,以便更清楚地说明处理层。

第一个例子,考虑一下 Internet 上的搜索引擎。如果忽略掉各种动画横幅、图像和其

他浮华的装饰,搜索引擎的用户界面实际上非常简单:用户输入关键字字符串,随后就可以看到一个 Web 页标题的列表。后台是存放 Web 页的一个巨型数据库,这些 Web 页被预先取出,并且编制了索引。搜索引擎的核心是将用户输入的关键字字符串转换为一个或多个数据库查询,数据库查询在查询完成后将查询结果排列成一个列表,再将该列表转换为一组 HTML 页面。在客户-服务器模型中,这种进行信息检索的部分一般放置在处理层。图 1.28 显示了搜索引擎的整个组织结构。

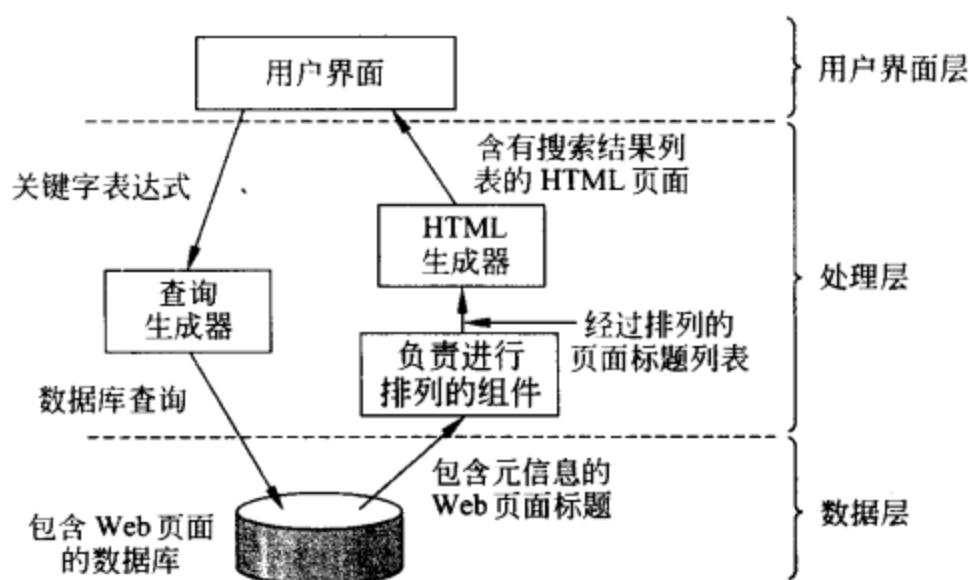


图 1.28 Internet 搜索引擎通常组织为三个不同层次

第二个例子,考虑股票经纪人使用的决策支持系统。与搜索引擎相似,这种系统可以分为前端和后端,前端实现了用户界面,后端访问存放财务数据的数据库,分析程序位于前端和后端之间。财务数据分析必须采用统计学和人工智能方面的复杂方法和技术。有时财务决策支持系统的核部分必须放到高性能计算机上来执行,这样才能取得用户所期望的处理速度和响应时间。

最后还有一个例子,考虑一个典型的桌面应用程序包,其中包括字处理器、电子表格应用程序、通信工具等。这种“办公软件”套装一般都通过一个公共的用户界面集成在一起,这种用户界面可以支持复合文档,并且可以对用户根目录中的文件进行处理。在这种情况下,处理层包含相对较多的程序,而其中每个程序的处理功能相对比较简单。

3. 数据层

客户-服务器模型中的数据层包含用于维护由应用程序操作的实际数据的程序。该层有一个重要特性,即数据是持久性的(persistent)。也就是说,即使没有应用程序运行,数据也会存储起来供以后使用。最简单的数据层由文件系统组成,但更一般的情况是使用一个完备的数据库。在客户-服务器模型中,数据层一般由服务器端实现。

除了数据存储之外,数据层一般还负责维护不同应用程序之间的数据一致性。如果使用了数据库,维护一致性意味着像对表的描述、条目约束这样的元数据以及针对特定应用程序的元数据也在该层存储。比如在银行里,如果某位顾客的信用卡透支额达到一定值,就需要生成一个通知。这种类型的信息可以通过一个数据库触发器来维护,它会在适

适当的时候激活相应的处理程序。

在传统的面向商务的环境中,常常把数据层组织成关系型数据库的形式。采用这种形式的关键原因是可以做到使数据具有独立性,即数据组织方式是独立于应用程序的。这样,数据组织方式的变化不会影响到应用程序,应用程序的变化也同样不会影响到数据组织方式。由于处理过程和数据是分开来考虑的,因此在客户-服务器模型中使用关系型数据库有助于将处理层与数据层分隔开来。

然而,对于有些应用程序来说,使用关系型数据库并不是理想的选择。这样的应用程序还日益增多。这类应用程序的典型特征是,它们处理的是复杂的数据类型,而这种数据类型更适合于用对象模型来表述,而不太适合用关系模型表述。在计算机辅助设计系统中有许多这种数据类型的示例。简单的例子有多边形和圆,复杂的有飞机设计描述。同样,对于多媒体系统来说,通过针对音频流和视频流的特定操作来处理诸如此类的音频流和视频流复杂数据类型,这无疑要比用关系表的形式对此类流数据进行建模要容易得多。

如果在某些情况下对数据的操作用对象操作的形式来表示更为容易,那么就有必要使用面向对象数据库来实现数据层。这种数据库不仅支持将复杂数据组织为对象,而且还可以存储针对这些对象的操作的实现。这样,处理层的部分功能就转移到了数据层中。

1.5.3 客户-服务器体系结构

上一节中讨论了三个逻辑层次之间的区别。这种区别的存在带来了将客户-服务器应用程序在物理上分布到若干台机器上的可能性。最简单的组织结构是仅使用两类机器:

- (1) 客户机器只包含实现(部分)用户界面层的程序;
- (2) 服务器机器包含其余程序,也就是实现处理层和数据层的程序。

这种组织结构存在的问题是它并非是真正分布式的:所有任务都由服务器处理,而客户实质上只不过是一台哑终端。还有许多其他类型的体系结构可以考虑,下面我们将对其中几种较为常见的体系结构进行介绍。

1. 多层体系结构

有一种组织客户和服务器的方案是这样的:把位于应用程序层上的程序分布到各台机器上去,如图 1.29 所示,请参见(Umar 1997, Jing 等 1999)。第一步,只需区分两类机器,即客户机器和服务器机器。这种体系结构因而得名为(物理上的)双层体系结构。

一种可能的组织结构是,在客户机器上只放置用户界面中与终端有关的那些部分,如图 1.29(a)所示,让应用程序远程控制其数据的表示。另一种方法是把用户界面软件全部放在客户端,如图 1.29(b)所示。在这种情况下可以把应用程序从本质上划分为图形前端和其余部分,前者通过特定应用程序所规定的协议与后者(驻留在服务器上)通信。在这种模型中,前端除了显示应用程序的界面外,并不进行其他的处理工作。

沿着这条思路走下去,我们还可以把应用程序中更多的部分移到前端,如图 1.29(c)所示。为介绍这种方案,以下的例子很有意义:应用程序要处理一个表单,而这个表单必须先等待用户填好后再进行处理。在这种情况下,可以由前端检查表单的正确性和一致

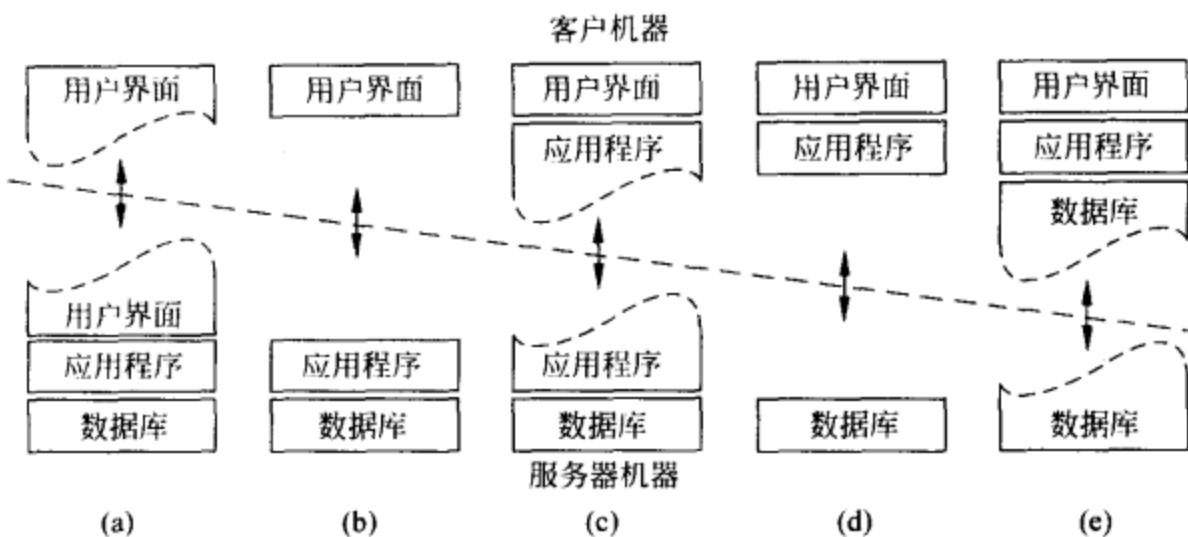


图 1.29 客户-服务器模型的可能组织结构

性，在必要的情况下还可以与用户交互。这种组织结构的另一个实例是一个字处理器，其基本编辑功能在客户端执行，客户端处理的是位于本地缓存或者存储器中的数据，而诸如拼写检查和语法检查这样的高级支持功能则放在服务器端执行。

在许多客户-服务器环境中，如图 1.29(d) 和图 1.29(e) 所示的组织结构特别流行。这些组织结构适用的情况是，客户机器是 PC 或者工作站，并且这些机器通过网络连接到一个分布式文件系统或者数据库。从本质上说，大多数应用程序运行于客户机器上，但是所有对于文件或者数据库数据条目的操作还是要通过服务器来执行。在图 1.29(e) 描述的情况下，客户机器的本地磁盘上含有部分数据。例如，在浏览 Web 结点的过程中，客户可能会在本地磁盘上逐渐建立起一个庞大的缓存，其中保存有最近浏览过的 Web 页。

在我们仅仅关注服务器和客户之间的区别时，我们就会遗漏重要的一点：服务器有时需要行使客户的职能，如图 1.30 所示。在这种情况下就形成了（物理上的）三层体系结构。

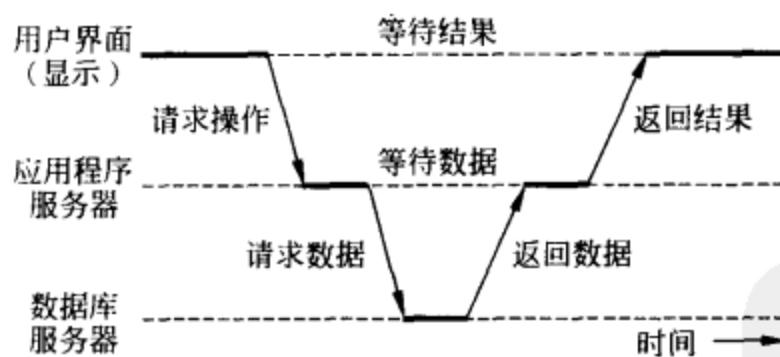


图 1.30 服务器行使客户职能的示例

在这种体系结构中，组成处理层的程序驻留在一个单独的服务器上，但也可以部分分布到客户机和服务器机器上。在事务处理中有一个典型的三层体系结构应用实例，即由一个单独的进程来协调可能分布在多个数据服务器上的所有事务，该进程称为事务监控器（transaction monitor）。在后面一些章节中还会对事务处理进行讨论。

2. 现代体系结构

多层客户-服务器体系结构的形成是将应用程序划分为用户界面、处理部分及数据层的直接结果。各层直接与应用程序的逻辑结构相对应。在许多商业环境中，分布式处理就是将客户-服务器应用程序组织成多层次体系结构。我们把这种分布叫做纵向分布 (vertical distribution)。纵向分布的典型特征在于，它是通过将逻辑上不同的组件分别放置在不同的机器上实现的。纵向分布的概念与分布式关系数据库中使用的纵向分割 (vertical fragmentation) 的概念有关。纵向分割指按列来拆分表，然后将它们分布到多台机器上。

然而，纵向分布仅仅是组织客户-服务器应用程序的方法之一，而且在多数情况下是最不引人注目的一种方法。在现代体系结构中，客户和服务器的分布通常更有价值，这种分布称为横向分布 (horizontal distribution)。在这种分布中，客户或者服务器可以在物理上分割成几个部分，这几个部分在逻辑上拥有同等地位，但是每个部分都处理自己拥有的完整的数据集，从而使负载得到平衡。

有一个流行的横向分布的例子，就是把 Web 服务器复制到某个局域网中若干台机器上，如图 1.31 所示。每台服务器上都有一组相同的 Web 页，在每次对 Web 页进行更新的时候，都立即把更新后的拷贝送到每一台服务器上。当接收到请求时，系统依据一种循环 (round-robin) 策略将该请求转发给某个服务器。只要有足够的带宽，采取这种横向分布的形式对于极度繁忙的 Web 站点来说将是非常有效的。

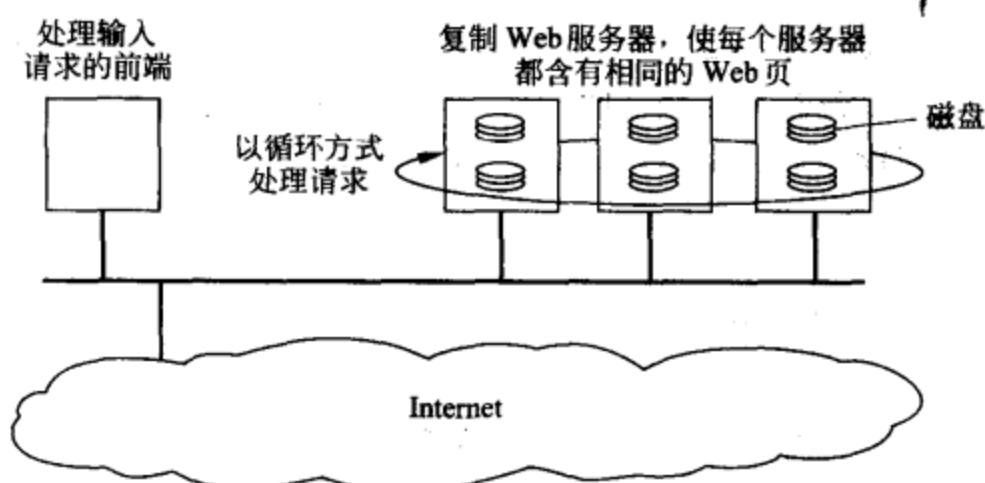


图 1.31 Web 服务的横向分布示例

客户也可以是分布的，尽管不是那么明显。对于简单的协作式应用程序来说，有些情况下甚至根本不存在服务器。这种情况下一般采用对等分布 (peer-to-peer distribution)。例如，一个用户与另一个用户进行联系，随后它们共同启动同一个应用程序来进行会话。随后也可以有第三个客户与前面二者之一相互联系，然后启动相同的应用程序软件来加入会话。

客户-服务器系统还有许多其他的组织结构，在文献 (Adler 1995) 中对这些组织结构进行了讨论。在后面的章节中也将涉及许多其他的分布式系统组织结构。我们将看到，通常，有些系统既是纵向分布的，又是横向分布的。

1.6 小 结

分布式系统由许多台独立的计算机组成,这些计算机协同工作使系统以单个一致系统的面貌出现。分布式系统的重要优点之一在于,这种结构使得将不同计算机上运行的各种应用程序集成为一个系统变得更加方便。另一个优点是,如果设计得合理,分布式系统的底层网络规模上的扩展将变得很容易。获得这些优点所付出的代价是软件更为复杂、性能有所损失,并且安全性降低。无论如何,分布式系统的构建和安装在全球范围内引起了人们的广泛兴趣。

存在不同类型的分布式系统。分布式操作系统与其他系统的区别在于,它管理的是紧耦合计算机系统的硬件,包括多处理器系统和同构式多计算机系统。这些分布式系统并不真正支持独立自主的计算机,但很好地实现了一个单一系统的视图。在另一方面,网络操作系统的长处在于将使用不同操作系统的计算机连接起来,这样用户就可以方便地使用每个结点的本地服务。然而,网络操作系统并不像分布式操作系统那样提供一个单一系统视图。

现代分布式系统一般是通过在网络操作系统之上添加一个附加的软件层来构建的。该层称为中间件,用来隐藏底层计算机集合的异构性和分布性。基于中间件的分布式系统通常采用某个特定的模型来对分布和通信进行描述。比较流行的模型是基于远程过程调用、分布式对象、文件和文档的。

内部组织结构对于任何分布式系统都很重要。在这方面一个广泛应用的模型是客户进程向服务器进程请求服务。客户向服务器发送一个消息,然后等待服务器的回应。这种模型与传统的程序设计方式是紧密相关的,在这种设计方式中服务作为单独模块中的过程来实现。更好的做法是将系统划分为用户界面层、处理层和数据层。服务器一般负责数据层的工作,而用户界面层在客户端实现。处理层既可以由客户实现,也可以由服务器实现,或者由两者共同实现。

对于现代分布式系统来说,客户-服务器应用程序如果只采用纵向组织结构是不足以构建大规模系统的,还需要采用横向分布,将客户和服务器在物理上分布并且复制到多台计算机上。万维网是成功应用横向分布的典型例子。

习 题

1. 中间件在分布式系统中扮演什么角色?
2. 请解释(分布)透明性的含义,并且给出各种类型透明性的例子。
3. 在分布式系统中,为什么有时难以隐藏故障的发生以及故障恢复过程?
4. 为什么有时要求最大程度地实现透明性并不好?
5. 什么是开放的分布式系统?开放性带来哪些好处?
6. 请对可扩展系统的含义做出准确描述。
7. 可以通过应用多种技术来取得可扩展性。请说出这些技术。

8. 多处理器系统与多计算机系统有什么不同?

9. 某多计算机系统中的 256 个 CPU 组成了一个 16×16 的网格方阵。在最坏的情况下,消息的延迟时间有多长(以跳(hop)的形式给出,跳是结点之间的逻辑距离)?

10. 现在考虑包含 256 个 CPU 的超立方体,最坏情况下消息的延迟有多长(同样以跳的形式给出)?

11. 分布式操作系统与网络操作系统有什么不同?

12. 请解释如何使用微内核将操作系统组织成客户-服务器的方式。

13. 请解释基于分页的分布式共享存储器系统主要有哪些操作。

14. 为什么要开发分布式共享存储器系统? 你认为是什么问题给这种系统的高效率实现造成了困难?

15. 请解释什么是分布式共享存储器系统中的伪共享。你有没有针对这个问题的解决方案?

16. 由于存在错误,某个试验性文件服务器有 $3/4$ 的时间能够正常工作,而另外 $1/4$ 的时间无法工作。如果要确保服务至少在 99% 的时间可用,需要将该文件服务器复制多少次?

17. 什么是三层客户-服务器体系结构?

18. 纵向分布与横向分布有什么不同?

19. 考虑一个进程链,该进程链由进程 P_1, P_2, \dots, P_n 构成,实现了一个多层次客户-服务器体系结构。进程 P_i 是进程 P_{i+1} 的客户, P_i 只有得到 P_{i+1} 的应答之后才能向 P_{i-1} 发出应答。如果考虑到进程 P_1 的请求-应答性能,这种组织结构主要存在什么问题?

第 2 章 通 信

进程间通信是一切分布式系统的核心。如果不仔细分析分布式系统中各机器间信息交换的机制,对分布式系统的研究就成了空谈。分布式系统中的通信都是基于底层网络提供的低层消息传递机制的。就像我们在第 1 章里谈到的那样,通过消息传递来描述通信过程比使用基于共享存储器的原语来描述要更困难。现代分布式系统中常常含有数千甚至数百万个进程,这些进程分散在诸如 Internet 的不可靠网络中。除非用其他的技术来替代计算机网络的原始通信功能,否则要开发大规模分布式应用程序是极为困难的。

在本章的开头,我们将讨论进行通信的进程必须遵守的规则(这种规则称作协议),并且把讨论重点放在协议的分层结构上。然后将讨论 4 个广泛使用的通信模型:远程过程调用(remote procedure call, RPC)、远程方法调用(remote method invocation, RMI)、面向消息的中间件(message-oriented middleware, MOM)以及流(stream)。

首先要讨论的是分布式系统中的 RPC(远程过程调用)通信模型。RPC 的目的在于将消息传递的大部分复杂性隐藏起来,它比较适用于客户-服务器应用程序。RMI(远程方法调用)是 RPC 模型的一种改进形式,它是基于分布式对象概念的。我们将在不同的节中对 RPC 和 RMI 分别进行讨论。

在许多分布式应用程序中,通信并不遵循严格的客户-服务器交互模式。在这种情况下,从消息的角度来考虑会更恰当。然而,计算机网络的低层通信功能由于缺乏分布透明性,从许多方面来说并不适用。替代的方案是使用高层消息队列模型,在这种模型中进行通信的方式与电子邮件系统的通信方式非常类似。因此 MOM(面向消息的中间件)是一个重要的主题,值得用专门的一节来讨论它。

多数系统都缺乏对诸如音频和视频这样的连续媒体的通信支持。随着多媒体分布式系统的出现,这个缺陷表现得愈发明显。必须引入流(stream)的概念,流可以在有时间限制的条件下支持消息的连续流。我们将在最后一节中对流进行讨论。

2.1 分 层 协 议

由于没有共享存储器,分布式系统中的所有通信都是基于(低层)消息交换的。如果进程 A 要与进程 B 通信,A 必须首先在自己的地址空间中生成该消息,再执行一个系统调用,通知操作系统将该消息通过网络发送给 B。虽然这个基本思路听起来很简单,但是做起来并不简单,为了避免混乱,A 和 B 必须先就传输的所有位所代表的含义达成一致。如果 A 发送了一部精彩的法文新小说,采用的是 IBM 的 EBCDIC 字符编码方式,而 B 等待接收的却是一张采用 ASCII 编码的超级市场的英文存货清单,显然,这样的通信是毫

无用处的。

需要制定多种这样的协定(agreement)。比如,用多高的电压来表示0(低电平)信号,又用多高的电压来表示1(高电平)信号?消息接收者如何得知哪一位标志着消息的结尾?如何检测消息是否丢失或者损坏,发现后该怎么办?数值、字符串以及其他数据项的长度是多少,表示方式是什么?简而言之,必须在不同层次上制定多种协定,其中包括从位传输的低层细节到信息表示的高层细节在内的各层。

为了更加方便地对通信中涉及的多个不同层次进行处理,并解决其中存在的问题,国际标准化组织(ISO)颁布了一个参考模型,该模型清楚地标明了通信涉及的各个层次,为这些层次给出了标准名称,并且指出各层执行的特定任务。该模型称作开放式系统互联参考模型(open systems interconnection reference model)(Day 和 Zimmerman 1983),通常简称为 ISO OSI 模型,有时进一步简称为 OSI 模型。需要强调指出的是,作为 OSI 模型的一部分而开发的协议从未得到过广泛应用。然而,学习底层模型本身对于理解计算机网络是十分有用的。虽然我们并不想在这里给出该模型的完整描述以及它涉及的所有内容,但是进行简短的讨论还是很有好处的。如果想知道关于该模型的更多细节,请参见文献(Tanenbaum 1996)。

OSI 模型是设计用来支持开放式系统间的通信的。所谓开放式系统是准备通过一系列标准规则来与其他开放式系统通信的系统,这些规则规定了发送和接收的消息的格式、内容以及相应的含义。对这些规则进行归纳总结,加以形式化,就形成了协议(protocol)。为了使一组计算机能够通过网络相互通信,它们必须使用相同的协议。可以把协议划分为两大类。一类是面向连接(connection-oriented)的协议,使用这种协议,消息发送者和接收者必须首先显式地确立连接,可能还需要就采用的协议进行协商,然后二者才能进行数据交换。在通信完毕之后,它们必须释放(终止)连接。电话系统就是一个面向连接的通信系统。另一类是无连接(connectionless)协议,使用这种协议,交换数据之前不需要有建立连接的过程,消息发送者只需要在准备好的时候开始传送第一个消息即可。将信件投入邮箱就是无连接通信的一个例子。对于计算机通信来说,无论面向连接的通信还是无连接的通信都是很常用的。

在 OSI 模型中,通信过程划分为 7 级(层),如图 2.1 所示。每一层负责处理通信中某个特定方面的问题。这样就可以把要解决的问题划分成易于处理的多个部分,每个部分都可以相对独立地进行处理。每一层都规定了与上面一层之间的接口,接口中包含一组操作,这些操作共同定义了该层向其用户提供的服务。

如果机器 1 上的进程 A 想与机器 2 上的进程 B 通信,它先生成一个消息,然后将该消息传递给机器 1 上的应用层。该层可能是一个程序库,也可能通过其他方式实现(比如在操作系统内部实现,或者由外部网络处理器实现)。应用层软件随后在消息前面加入一个报头(header),并通过第 6 层与第 7 层之间的接口将处理后的消息传递给表示层。表示层在得到的消息中加入自己的报头,并将得到的消息传给下一层即会话层。依此类推,逐层向下传递。某些层不仅要在消息开头加入自己的报头,而且还要在消息末尾加入报尾(trailer)。当消息最终到达底层物理层时,由该层执行实际的消息传输任务,最终得到的消息如图 2.2 所示。

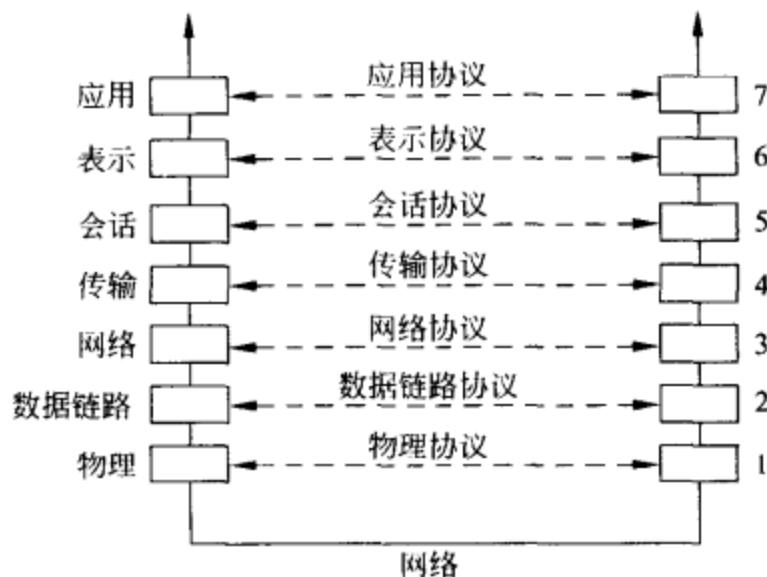


图 2.1 OSI 模型中的层、接口以及协议

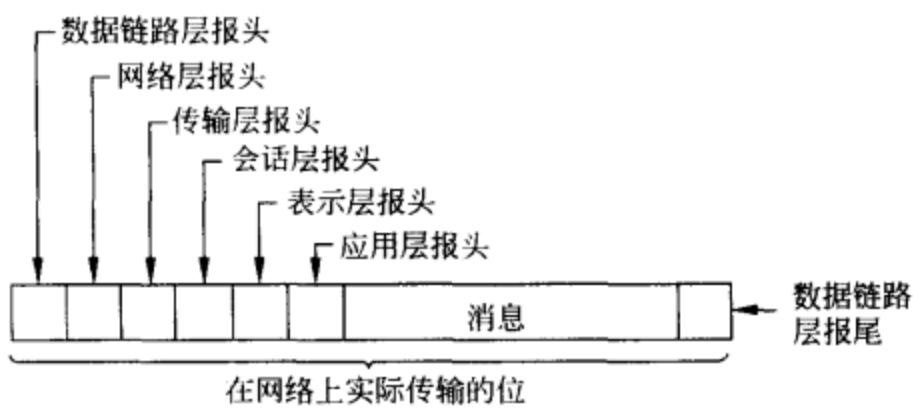


图 2.2 在网络上传输的典型消息

当消息送达机器 2 时，又被逐层向上传递。每一层都对该层原来加上的报头进行检验，并将其去掉。最终消息抵达接收者即进程 B，进程 B 可使用相反的路径进行应答。第 n 层加入的报头中的信息是供第 n 层协议使用的。

以下这个例子说明了分层协议的重要性。我们来考虑两家公司之间的通信问题，这两家公司分别是 Zippy 航空公司以及它的服务商 Mushy 餐饮国际有限公司。每个月 Zippy 公司的客运服务负责人会打发她的秘书与 Mushy 公司的销售经理秘书进行联系，订购 10 万份鸡肉。按照传统的做法，订单应该通过邮局来递送，然而由于邮政服务很糟糕，两位秘书决定放弃这种方式，转而采用传真方式进行通信。采用哪种通信方式并不需要征得他们各自老板的同意，因为他们之间达成的协议只涉及订单的物理传输方式，而与订单的内容无关，所以不会对订单的内容造成任何影响。

同样，客运服务负责人也可以决定不再订购鸡肉，而订购 Mushy 公司新推出的羊上肋肉，他所做出的这个决定并不会对秘书的工作造成影响。要注意的是：这里分为两层，即老板和秘书。每一层都有自己的协议（通话的主题和采用的技术），可以单独改变某一层的协议，而不会对另一层造成影响。分层协议的可贵之处正是在于它带来的这种独立性。随着技术的发展，可以对任何层单独进行改进，而不会对其他层造成影响。

OSI 模型中不止有两个层次，从图 2.1 中可以看出，它共分为 7 层。特定系统中使用

的协议集称为协议组(protocol suite)或者协议栈(protocol stack)。把参考模型与该参考模型使用的实际协议区分开来是很重要的。我们曾经指出,OSI 协议从未流行过。相反,得到广泛应用的是那些为 Internet 开发的协议,比如 TCP 协议和 IP 协议。在以下各节中,我们将从底层开始,依次对 OSI 各层进行简要分析。然而,我们并不给出 OSI 协议的例子,而是在适当的地方指出各层中使用的某些 Internet 协议。

2.1.1 低层协议

我们先讨论 OSI 协议组中的三个最低层协议。这三层共同实现了计算机网络的基本功能。

1. 物理层

物理层负责对 0(低电平)信号和 1(高电平)信号的传输。物理层要解决的是 0 和 1 信号分别使用多高的电压,每秒传输多少位,以及是否可以同时进行双向传输等关键问题。另外该层还需要定义网络连接器(插头)的尺寸和形状,以及插头上针脚的个数和各引脚的作用。

物理层负责为接口制定电气、机械和信号方面的标准,以确保某台机器发送的 0 信号在接收机器上仍然被识别为 0 信号,而不会变成 1 信号。已经制定了许多(在不同介质中使用的)物理层标准,其中包括供串行通信线路使用的 RS-232-C 标准。

2. 数据链路层

物理层只负责传输位。只要没有错误发生,就不会有任何问题。然而,实际的通信网络中会发生错误,所以需要采用某种机制来检测并且纠正这些错误,实现这种机制是数据链路层的主要任务。该层将若干位组成一个有时被称作帧(frame)的单元,并检查每一帧是否被接收者正确接收。

为了完成这些工作,数据链路层在每一帧的开头和结尾分别放置特殊的位模式来对头尾作出标记,并且用某种方法将帧中所有的字节相加,计算出校验和(checksum)。数据链路层将得到的校验和加进帧中一同传输。当帧送达的时候,接收者对数据校验和进行重新计算,并将得出的结果与帧中附带的校验和进行比较。如果两者一致,说明该帧传输过程中没有发生错误,于是接受该帧。如果两者不一致,接收者就要求发送者重发该帧。每一帧都带有一个序列号(在其报头中),所以可以很容易地对不同帧作出区分。

在图 2.3 中,我们可以看到一个(稍稍有些不合理的)例子。A 试图向 B 发送两个消息,即消息 0 和消息 1。在时刻 0 发送了数据消息 0,但是该消息在时刻 1 送达的时候,由于传输线路上的噪声而损坏了,因而它的校验和发生了错误。B 注意到了这一点,于是在时刻 2 向 A 发送一个控制消息,要求重发消息 0! 不幸的是,在同一个时刻,即时刻 2,A 正在发送数据消息 1。当 A 收到重发请求的时候,它重新发送数据消息 0。然而,B 先接收到的是数据消息 1 而不是重发的数据消息 0,因此它会向 A 发送控制消息 1,声明它要求重发的是消息 0 而不是消息 1。A 接收到该控制消息后,只能无可奈何地再次发送消息 0,至此它已经发送了 3 遍消息 0。

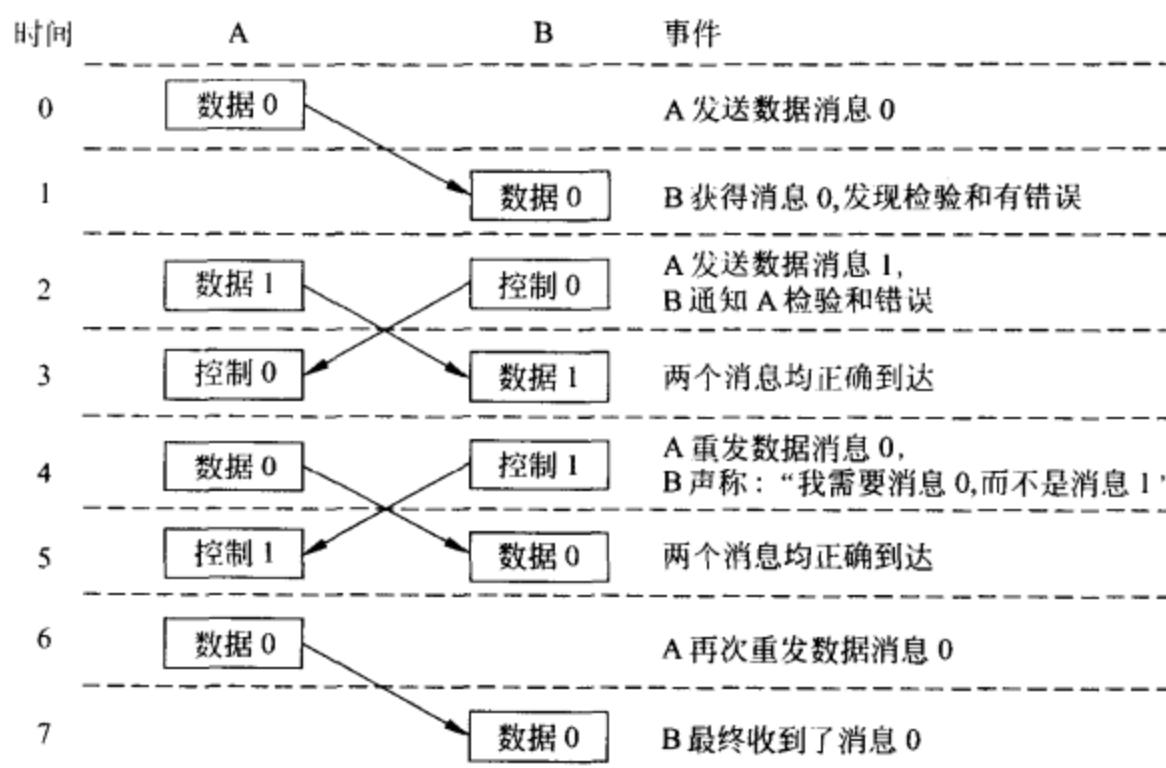


图 2.3 数据链路层上发送者与接收者之间的对话

该例所强调的重点不在于图 2.3 中的协议是否是一个很好的协议(实际上它不是),而主要表示的是每一层都需要在发送者和接收者之间进行通话。典型的消息有“请重发消息 n ”,“我已经重发过消息 n 了”,“你没有重发”,“确实重发了”,“好吧,也许你是对的,请再重发一遍”,等等。这些通话的内容包含在报头字段中,该字段定义了各种请求及响应,还可以加进参数信息(比如帧的编号)。

3. 网络层

在局域网(LAN)中,发送者一般并不需要了解接收者所在的位置,它只须将消息放到网络上,然后接收者会取走该消息。然而在广域网中情况就不同了。广域网由大量计算机组成,其中每台机器都拥有连接到其他机器的若干条线路,就像一幅显示大城市以及大城市之间相连的道路的大范围地图。消息从发送者传送到接收者的过程中,可能要经过好几次转发,每次转发都要选择一条传出线路。如何选择最佳路径的问题称作路由选择(routing),这是网络层的首要任务。

实际上,最短的路线并不总是最佳路线,这就使得路由选择问题变得愈加复杂。最重要的是所选择路径的传输延迟时间,这与各条线路上的网络通信流量以及排队等候发送的消息数量有关。因此,传输延迟会随时间的推移而发生变化。一些路由选择算法试图根据动态的负载值选择路由,而其他的算法则只是根据一段较长时间内的负载均值来作出决策。

目前应用最广泛的网络协议可能就是无连接的网际协议(IP),它是网际协议组的一个组成部分。不需要先建立连接就可以发送 IP 包(包是表示网络中消息的术语称呼)。所有 IP 包彼此独立地发送到目的地,而且在包内部不指定或记录路径。

现在有一种面向连接的协议日趋流行起来,这就是 ATM 网络中的虚拟信道(virtual

channel)。ATM 中的虚拟信道是从信号源指向目的地的单向连接,中途可能会经过若干台 ATM 中介交换机。可以将一组虚拟信道组合成一条所谓的虚拟路径(virtual path),而不需要在两台主机之间单独建立每个虚拟信道。虚拟路径可以比作两台主机之间预先确定好的路线,沿着该路线来“铺设”所有的虚拟信道。某个路径的变更意味着所有与该路径相关联的信道都自动变更。关于 ATM 的更多资料可以在文献(Handel 等 1994)中找到。

2.1.2 传输协议

有一些服务对于构建网络应用程序是必需的,但在网络层接口中却没有提供,这些服务由传输层来实现。从这个意义上说,传输层构成了基本网络协议栈的最后一部分,也就是说传输层将底层网络转换成可供应用程序开发人员使用的形式。

1. 传输层的功能

发送者发给接收者的包有可能在途中丢失。虽然某些应用程序可能拥有自定义的错误恢复手段来进行处理,但是多数程序还是希望连接是可靠的。传输层的任务就是提供这种可靠的连接服务。这样,应用层在把消息交付给传输层的时候,消息将会按预期的那样被无损地送达接收者。

当从应用层接收到消息时,传输层将消息分为适于传输的小块,每一块都分配一个序列号,然后将它们发送出去。在传输层报头中涉及的内容包括哪些包已发送、哪些包已接收到、接收者所拥有的空间还能够容纳多少包、哪些包应该重发以及其他一些类似的主题。

可靠的传输连接(根据定义它是面向连接的)既可以建立在面向连接的网络服务之上,也可以建立在无连接的网络服务之上。如果是前者,所有的包都将依发送时的次序到达接收者(如果所有的包都能安全到达的话);而如果是后者,由于各包经过的路线可能不同,后发送的包有可能先到达。由传输层中的软件负责将所有的包依照原来的顺序排列好,以维持传输连接就像一根大管子的假象:将消息从一端放入,随后从另一端按照放入时的顺序将消息无损地提取出来。提供这种点对点的通信方式是传输层的重要特征之一。

Internet 的传输协议称为传输控制协议(transmission control protocol, TCP),在文献(Comer 2000a)中对其进行了详细描述。TCP 与 IP 结合成的 TCP/IP 协议是目前网络通信中事实上的标准。Internet 协议组也支持无连接的传输协议,该协议称为通用数据报协议(universal datagram protocol, UDP),从本质上说它只不过是附加了某些次要内容的 IP 协议。不需要面向连接协议的用户程序通常使用 UDP。

官方的 ISO 传输协议有 5 种变体形式,分别称为 TP0、TP1、TP2、TP3 和 TP4。这些变体之间的区别主要在于错误处理过程的不同,以及在单个低层连接上建立多个传输连接的能力(在 X.25 中明确规定)的不同。究竟使用哪一种传输协议取决于位于底层的网络层特性。实际上,这些协议都没有得到广泛的使用。

其他的传输协议都是为了某种专门用途而制定的。例如,为了对实时数据传输提供支持,制定了实时传输协议(real-time transport protocol, RTP)。RTP 是一种框架性协

议,也就是说它规定了实时数据所用的包的格式,但是并没有提供确保数据成功送达的实际机制。另外,它还指定了另一种协议,用于监督并控制 RTP 包数据传输(Schuizrinne 等 1996)。

2. 客户-服务器 TCP

在分布式系统中,客户与服务器间的交互常常是利用底层网络的传输协议来进行的。随着 Internet 日趋流行,使用 TCP 来构建客户-服务器应用程序和系统如今已经变成很平常的事情。与 UDP 相比,TCP 的优点在于它可以在任何网络上可靠工作。TCP 的明显缺点在于 TCP 引入的开销显著增加,尤其是与那些底层网络极为可靠的情况,比如局域网系统相比,这种开销的增加更加明显。

当性能与可靠性的要求发生冲突时,可以考虑的另一种方案是采用 UDP,并且辅之以针对特定应用程序进行了优化的附加错误控制及流程控制手段。这种方案的缺点在于需要进行许多额外的开发工作,并且由于引入了一种专有的解决方案,将会对系统开放性带来影响。

TCP 之所以在许多情况下不如人意,原因在于它并不适合用于支持多数客户-服务器交互过程所使用的同步请求-应答方式。在正常的情况下,当没有丢失消息时,使用 TCP 进行客户-服务器交互的过程如图 2.4(a)所示。首先,客户启动连接建立过程,这个过程是通过一个三向握手协议来完成的,如图 2.4(a)中开始的三个消息所示。为了使通信双方能够就通过连接发送的包的次序编号方式达成一致(要了解详细资料,请参见文献(Tanenbaum 1996)),这个握手协议是必需的。连接建立之后,客户发送请求(消息 4),随后立即发出另一个包(消息 5),告诉服务器关闭该连接。

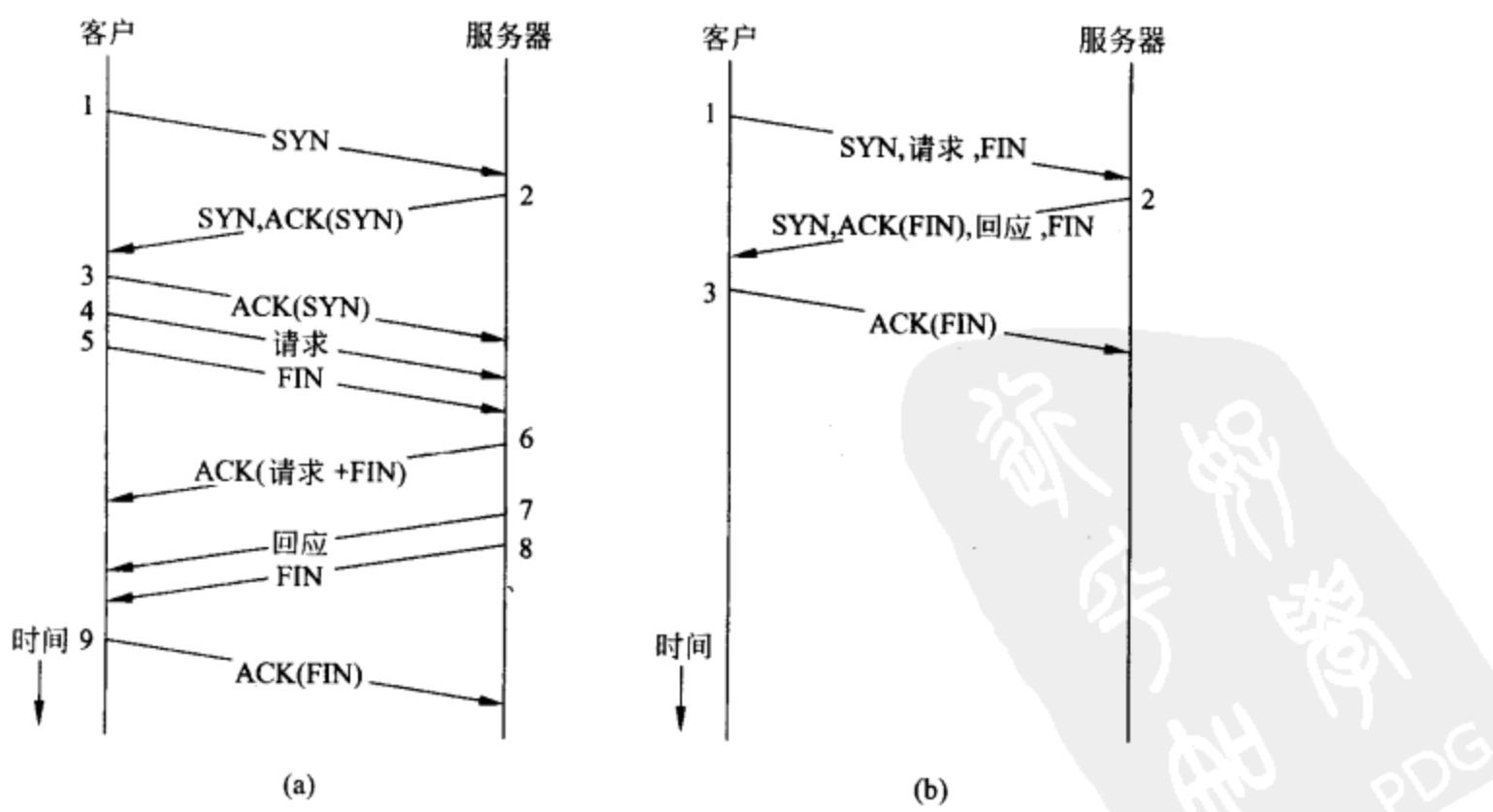


图 2.4 正常的 TCP 操作和事务性 TCP

(a) 正常的 TCP 操作; (b) 事务性 TCP

服务器随即做出确认响应,向客户证实它已经接收到客户的请求,附带确认连接将会关闭(消息 6)。随后服务器执行请求的操作,并将回应发送回客户(消息 7),接着发送请求要求客户释放连接(消息 8)。客户进行响应,确认与服务器的通信终止(消息 9)。

很明显,TCP 中的许多开销都耗费在连接的管理上。在使用 TCP 进行客户-服务器交互时,更经济的方式是将发送请求当作连接的建立标志,同样地把送回结果当作连接的关闭标志。这样的协议称作事务用 TCP(TCP for transaction),简写为 T/TCP。通常情况下,它的基本工作过程如图 2.4(b)所示。

通常情况下,客户首先发送一个单个消息(图中以消息 1 表示),其中包含三条信息:建立连接的请求、实际的服务请求本身,以及一个关闭连接的请求(该请求告知服务器可以随后立即拆断该连接)。

服务器在完成了实际请求的操作之后才进行响应,因此它可以把操作结果和接受连接时所需的数据放在一起发送,并且在消息中要求客户立即释放连接,如图中消息 2 所示。同样,客户的响应只是简单确认连接的终止(消息 3)。

这个协议是作为 TCP 的增强版本而设计的,也就是说,T/TCP 进程在与之通信的另一方没有实现 T/TCP 的情况下,将会自动切换到普通的 TCP 方式。Stevens(1996)对 TCP/IP 进行了详细讨论。

2.1.3 高层协议

在传输层之上,OSI 又划分了三个层次,在实践中只用到其中的应用层。事实上,在 Internet 协议组中,传输层之上的所有内容都合到一起了。在本节中我们将会看到,对于中间件系统来说,OSI 方案和 Internet 方案都不真正适用。

1. 会话层与表示层协议

会话层本质上是传输层的增强版本,它提供了用于追踪正在谈话的一方的对话控制,还提供了同步功能。同步功能对于长时间的传输过程非常有用,它允许用户在传输过程中插入若干检验点,这样如果传输崩溃,只需要回退到最近的一个检验点,而不需要从头开始重新传输。在实践中,应用程序很少用到会话层,几乎不对它提供支持。在 Internet 协议组中没有与会话层对应的层次。

低层主要考虑的问题是如何将要传输的所有位从发送者可靠而高效地传送给接收者,而在表示层中主要考虑的则是这些位所表示的意义。多数消息并不是由随机的位串组成的,而是由更结构化的信息,比如人名、地址、金额等组成。在表示层中,有可能对包含这种结构化信息字段的记录加以定义,随后由发送者通知接收者消息中包含有符合某种特定格式的记录。这种做法可以使得内部数据表示方法不同的机器间的通信更加方便。

2. 应用层协议

OSI 应用层起初是准备用来容纳一组标准网络应用程序的,比如那些供电子邮件、文件传输以及终端仿真使用的应用程序。但是现在,应用层变成了所有由于各种原因不适

于归纳到某个较低层中去的应用程序和协议的大杂烩。从 OSI 参考模型的角度来看,几乎所有的分布式系统都只不过是应用程序而已。

这个模型中缺乏对应用程序、针对特定应用程序的协议以及通用协议的准确区分。例如,Internet 的文件传输协议(file transfer protocol,FTP)(Postel 和 Reynolds 1985; Horowitz 和 Lunt 1997)规定了一种用于在客户机器和服务器机器间传输文件的协议。该协议不应与 ftp 程序相混淆,ftp 程序是指一种供最终用户使用的应用程序,用来传输文件,而且这种程序恰好(当然不完全是偶然的巧合)实现了 Internet FTP 协议。

另一个关于针对特定应用程序的协议的例子是超文本传输协议(hypertext transfer protocol,HTTP)(Fielding 等 1999),它是设计用来对 Web 页进行远程管理和传输操作的。该协议由 Web 浏览器以及 Web 服务器这样的应用程序实现。然而,HTTP 目前也用在与 Web 没有直接联系的系统中。例如,Java 的 RMI 就使用 HTTP 来向受防火墙保护的远程对象发出调用请求(Sun Microsystems 1998)。

还有为数不少的对多数应用程序十分有用的通用协议,但是这些协议并不能算是传输协议。在许多情况下,这种协议只能归入中间件协议的范畴。下面讨论中间件协议。

3. 中间件协议

中间件是一种应用程序,它在逻辑上位于应用层,但在其中包含有多种通用协议,这些协议代表各自所在的层,独立于其他更加特别的应用。可以在高层通信协议和用于建立各种中间件服务的协议间作出区分。

有多种协议可以对各种中间件服务提供支持。比如说,有很多方法可以进行身份验证(authentication)——为声称的身份提供证明,我们将在第 8 章中对身份验证进行讨论。身份验证协议与任何特定的应用程序都没有紧密的联系,但是可以作为一种通用服务集成到中间件系统中。同样,授权协议(这种协议允许通过验证的用户和进程对其拥有授权的资源进行访问)本质上也可以是通用的、独立于应用程序的。

作为另一个例子,我们来考虑一下第 7 章中的多种分布式提交协议。提交协议确定了在一组进程中,某种操作要么由全体进程共同完成,要么不执行。这种现象也称作原子性(atomicity),在事务中广泛应用。就像下面介绍的那样,除了事务以外,其他应用程序也可以利用分布式提交协议,比如具有容错性的应用程序。

最后一个例子是关于分布式锁定协议的,这种协议可以保护资源免受分布于多台机器上的一组进程的并发访问。我们将会在第 5 章中介绍这种协议。同样的,该协议也可以用于实现通用中间件服务,但它又是高度独立于任何特定应用程序的。

中间件通信协议支持高层通信服务。例如,在接下来的两小节我们将要讨论允许进程以高度透明的方式调用远程机器上的过程或者调用远程机器上的对象的协议。同样也有用于对实时数据传输进行设定并使其保持同步的协议,这种实时数据传输是多媒体之类的应用程序所需要的。另外,某些中间件系统还提供可靠的多播服务,这种服务的规模可以达到遍及广域网上数以千计的接收者的水平。

某些中间件通信协议也可以归入传输层中,而把它们归入更高层还有一些特殊的原因。比如,可扩展性得到保证的可靠的多播服务只在考虑应用程序需求的情况下才能实

现。因而中间件系统可以提供不同的(可调整的)协议,每种协议通过不同的传输协议依次实现,但是各协议提供的接口可以是单一的。

采用这种方案进行分层将形成一种改进的通信参考模型,如图 2.5 所示。与 OSI 模型相比,会话层和表示层由一个单一的中间件层代替,该层中包含有独立于应用程序的协议,这些协议不属于我们刚才讨论过的低层。原来的传送服务可以作为中间件服务来提供,而无需加以改动。这种方案与在传输层提供的 UDP 有异曲同工之处。同样的,中间件通信服务中可以加入消息传递服务,它与由传输层提供的服务相似。

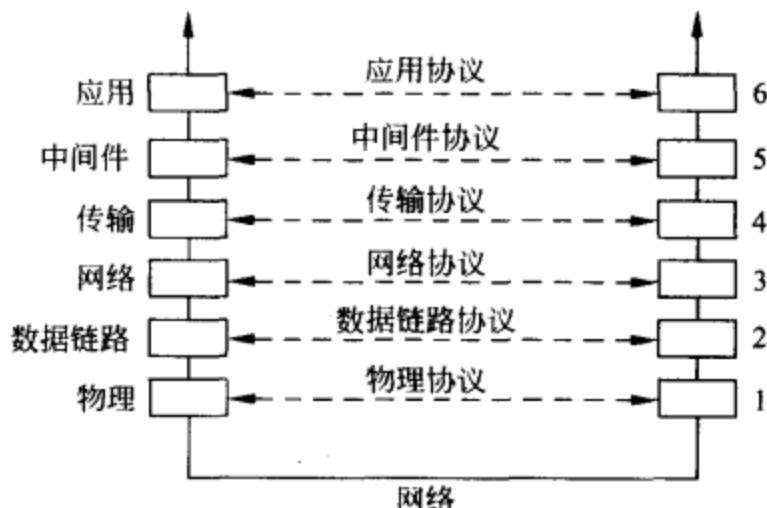


图 2.5 调整后的网络通信参考模型

在本章的余下部分,我们将集中讨论 4 种高级的中间件通信服务:远程过程调用、远程对象调用、消息队列服务以及通过流为连续媒体通信提供的支持。

2.2 远程过程调用

许多分布式系统是基于进程间的显式消息交换的,然而消息的发送(send)和接收(receive)过程无法隐藏通信的存在,而通信的隐藏对于在分布式系统中实现访问透明性是极为重要的。这个问题由来已久,但是人们在很长一段时间内都没有找到合适的解决办法,直到 Birrell 和 Nelson(1984)在一篇论文中引入了一套与传统方法截然不同的通信处理手段。虽然其中蕴含的思想很简单(一旦经过思考以后就很容易理解),但其中的含义非常复杂精妙。在本节中,我们将详细分析这个概念,包括它的实现、长处以及弱点。

简要地说,Birrell 与 Nelson 认为应该允许程序调用位于其他机器上的进程。当机器 A 上的进程调用机器 B 上的进程时,A 上的调用进程被挂起,而 B 上的被调用进程开始执行。调用方可以通过使用参数将信息传送给被调用方,然后可以通过传回的结果得到信息。编程人员看不到任何消息传递过程。这种方法称为远程过程调用(remote procedure call,RPC)。

虽然基本思想听起来很简单也很不错,但是还存在很多的具体问题。首先,调用过程和被调用过程运行在不同机器上,它们在不同的地址空间内执行,这就带来了复杂性。其次,需要进行参数和结果传递,这种传递可能是复杂的,特别是在机器间存在差异的情况下更是这样。最后,双方所在的机器都可能发生崩溃,任何一种潜在的故障都可能引发不

同的问题。尽管存在这么多问题,但是其中的大多数是能够解决的。RPC 作为一种广泛使用的技术,已成为许多分布式系统的基础。

2.2.1 基本的 RPC 操作

我们先讨论常规的过程调用,然后解释如何把调用本身划分为分别在不同机器上执行的客户部分和服务器部分。

1. 常规过程调用

为了理解 RPC 的工作机制,首先必须彻底理解常规(也就是单机)过程调用的工作机制。考虑下面的 C 语言调用:

```
count = read (fd, buf, nbytes);
```

其中 fd 是一个整型数,代表某个文件。buf 是一个字符数组,用于存放读入的数据。nbytes 是另一个整型数,用来记录实际读入的字节数。如果该调用位于主程序中,那么在调用之前堆栈的状态将如图 2.6(a)所示。为了进行调用,调用方首先把参数反序压入堆栈,即最后一个参数最先压入,如图 2.6(b)所示。(C 编译器将参数按相反顺序压入堆栈的原因与函数 printf 有关。因为只有这样做,才能保证 printf 能找到它的第一个参数,即格式字符串。)在 read 操作运行完毕后,它将返回值放在某个寄存器中,移出返回地址,并将控制权交回调用方。调用方随后将参数从堆栈移出,使堆栈还原到初始的状态。

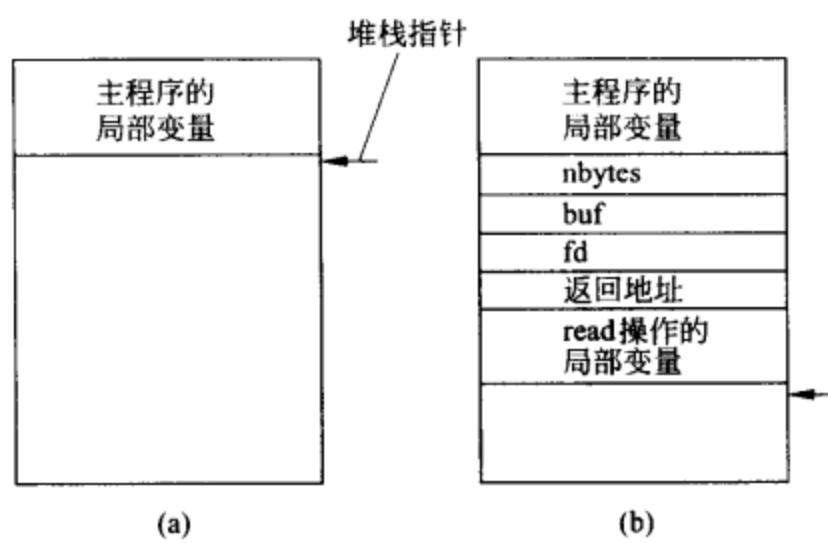


图 2.6 过程调用中的参数传递

(a) 本地过程调用中的参数传递: 调用 read 之前的堆栈状态; (b) 被调用过程执行时的堆栈状态

有一些问题需要注意。其中一个问题是,在 C 语言中,可以采用传值调用(call-by-value)和引用调用(call-by-reference)来传递参数。传值调用的参数(以下简称为值参数),如上面例子中的 fd 和 nbytes,只是简单地复制到堆栈中,如图 2.6(b)所示。对于被调用过程来说,值参数仅仅相当于初始化过的局部变量,被调用过程可以对这种参数进行修改,但是所作的改动并不会影响到调用端该参数原来的值。

C 中的引用参数实际上是指向某个变量的指针(也就是该变量所在的地址),而不是该变量的值。在上例对 read 的调用中,第二个参数就是一个引用参数,因为 C 语言中的数组

总是通过引用传递的。实际上,压入堆栈的是字符数组的地址。如果被调用过程使用该参数来将某些内容存储到字符数组中去的话,它确实修改了位于调用者过程中的数组。对于 RPC 来说,传值调用与引用调用的区别显得尤为重要,我们将在下文中看到这一点。

还存在另一种在 C 语言中未使用的参数传递机制,称作复制-还原调用(call-by-copy/restore)。这种调用的过程是:首先由调用方将变量复制到堆栈中,这一步与传值调用相同;随后在调用完毕之后将堆栈中的变量复制回去,覆盖掉调用方中该变量原先的值。多数情况下,这种调用的结果与引用调用相同,但是某些情况下这种调用会有不同的语义,比如说如果同一个参数在参数列表中出现了多次就会造成不同。许多语言中没有使用复制-还原调用机制。

采用哪种参数传递机制的决策通常由语言的设计者作出。参数传递机制是语言的固定特性之一,有些时候它还依赖于要传递的数据类型。在 C 语言中,整型数以及其他的基本量类型总是通过值传递的,而数组总是通过引用传递的,正如我们刚才看到的那样。某些 Ada 编译器对于入出(in out)参数采用复制-还原调用机制,而其他编译器则采用引用调用机制。由于语言本身的规定同时允许这两种参数传递机制,这就使得在语义上显得有些模糊。

2. 客户存根及服务器存根

RPC 背后隐含的思想是尽量使远程过程调用具有与本地调用相同的形式。也就是说,希望 RPC 是透明的,即调用过程不应该发现被调用的过程实际在另一台机器上执行,反过来也一样。假定程序需要从某个文件读取数据,程序员在代码中执行 read 调用来取得数据。在传统(单处理器)系统中,read 例程由连接器从库中提取出来,然后连接器再将它插入目标程序中。read 过程是一个短过程,一般通过执行一个等效的 read 系统调用来实现。也就是说,read 过程是一种位于用户代码与本地操作系统之间的接口。

虽然 read 中执行了系统调用,但是它本身依然是通过将参数压入堆栈的常规方式调用的,如图 2.6(b)所示。这样,程序员就不会知道 read 究竟干了些什么。

RPC 是通过类似的途径达到透明性的。当 read 实际上是一个远程过程时(比如在文件服务器所在机器上运行的过程),库中就放入 read 的另一种版本,称为客户存根(client stub)。这种版本的 read 过程同样遵循图 2.6(b)中的调用次序,这一点与原来的 read 过程相同,另外一个相同点是其中也执行了本地操作系统调用。惟一的不同点就是它并不要求操作系统提供数据,而是将参数打包成一个消息,然后请求将此消息发送到服务器,如图 2.7 所示。在对 send 的调用之后,客户存根调用 receive 过程,随即阻塞自己,直到收到响应消息。

当消息送达服务器时,服务器上的操作系统将它传递给服务器存根(server stub)。服务器存根是客户存根在服务器端的等价物,它也是一段代码,用来将通过网络输入的请求转换为本地过程调用。服务器存根一般先调用 receive,然后被阻塞,等待消息输入。收到消息后,服务器存根将参数由消息中提取出来,然后以常规方式调用服务器上的相应过程(如图 2.6 所示)。从服务器的角度来看,过程就好像是由客户直接调用一样:参数和返回地址都位于堆栈中,一切都很正常。服务器执行所要求的操作,随后将得到的结果

以常规的方式返回给调用方。以 read 为例,服务器将用数据填充 read 中第二个参数指向的缓冲区,该缓冲区是属于服务器存根内部的。

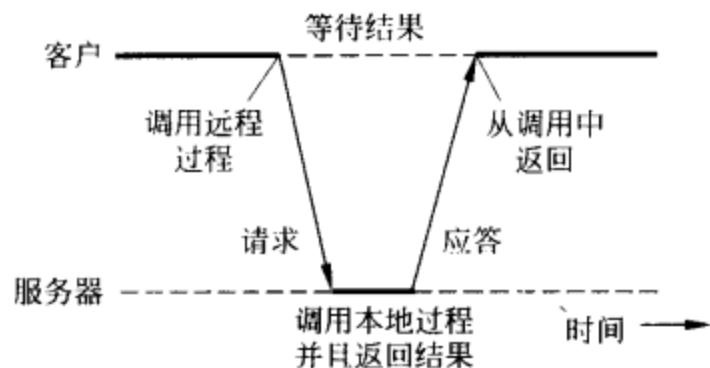


图 2.7 客户与服务器之间的 RPC 的原理

调用完毕之后,服务器存根要将控制权交回给客户发出调用的过程,它将结果(缓冲区)打包成消息,随后调用 send 将结果返回给客户。此后,服务器存根一般会再次调用 receive,等待下一个输入的请求。

客户机器接收到消息以后,客户操作系统会发现该消息属于某个客户进程(实际上该进程是客户存根,但是操作系统无法对二者作出区分)。操作系统将消息复制到相应的缓冲区中,随后解除对该客户进程的阻塞。客户存根检查该消息,将结果提取出来并复制给调用者,然后以通常的方式返回。当调用者在 read 调用进行完毕后重新获得控制权时,它所知道的惟一的事就是已经得到了所要的数据。它并不知道操作不是通过本地操作系统进行的,而是在远程完成的。

整个方法的精彩之处在于客户方面可以简单地忽略不关心的内容。客户所涉及的操作只是通过执行普通的(本地)过程调用来访问远程服务,它并不需要直接调用 send 和 receive。消息传递的所有细节都隐藏在双方的库过程中,就像传统库中隐藏了执行实际系统调用的细节一样。

总的说来,远程过程调用包含下列步骤:

- (1) 客户过程以正常的方式调用客户存根;
- (2) 客户存根生成一个消息,然后调用本地操作系统;
- (3) 客户端操作系统将消息发送给远程操作系统;
- (4) 远程操作系统将消息交给服务器存根;
- (5) 服务器存根将参数提取出来,然后调用服务器;
- (6) 服务器执行要求的操作,操作完成后将结果返回给服务器存根;
- (7) 服务器存根将结果打包成一个消息,然后调用本地操作系统;
- (8) 服务器操作系统将含有结果的消息发送回客户端操作系统;
- (9) 客户端操作系统将消息交给客户存根;
- (10) 客户存根将结果从消息中提取出来,返回给调用它的客户过程。

所有这些步骤总的效果是,将客户过程对客户存根发出的本地调用转换成对服务器过程的本地调用,而客户和服务器都不会意识到有中间步骤的存在。

2.2.2 参数传递

客户存根的功能是将得到的参数打包成消息,然后将消息发送给服务器存根。虽然这听起来很清楚明了,但是实际上并不那么简单。在本节中,我们将讨论 RPC 系统中涉及的一些参数传递方面的问题。

1. 传递值参数

把参数打包进消息中的过程称作参数编组(parameter marshaling)。作为一个非常简单的例子,我们来考虑一下远程过程 $\text{add}(i,j)$ 。该过程有两个整型参数 i 和 j ,在结果中返回 i 和 j 的算术和(实际上,基于开销方面的考虑,一般不会对这种简单过程进行远程调用,它只是作为一个简单例子)。在客户进程中对 add 的调用如图 2.8 中左边部分所示。客户存根将 add 的两个参数放入消息中,同时还在消息中加入要调用的过程的名称或者编号,这是为了在服务器支持多个不同调用的情况下指明要求执行的是哪一个调用。

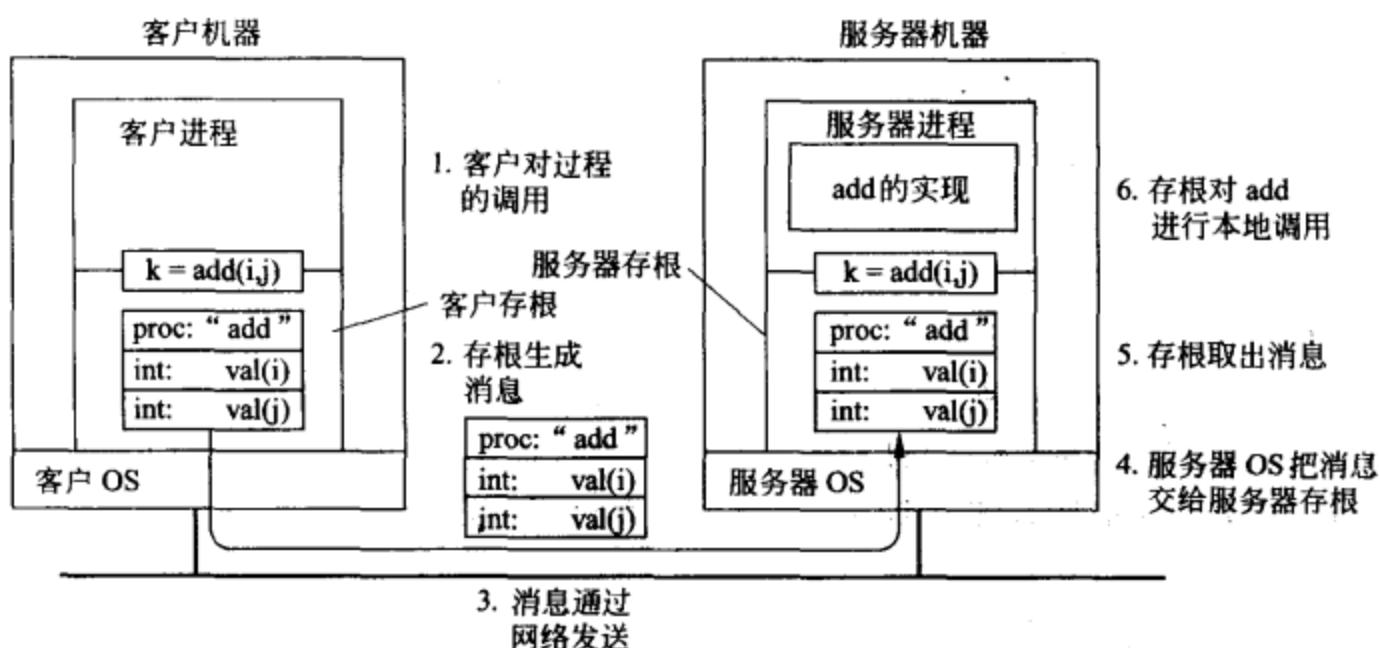


图 2.8 通过 RPC 进行远程计算的步骤

消息送达服务器之后,服务器存根对该消息进行分析,以判明需要调用哪一个过程,随后执行相应的调用。如果服务器还支持其他的远程过程,服务器存根中就需要使用 switch 语句,以根据消息中首字段的内容来选取需要调用的过程。服务器存根对服务器的实际调用与原来客户进程发出的调用十分相似,不同的只是前者的参数是依据输入的消息来完成初始化的变量。

服务器运行完毕后,服务器存根重新收回控制权,并将服务器得到的结果打包成消息送回给客户存根,客户存根随即将结果从消息中提取出来,把结果值返回给客户过程。

只要客户机器与服务器机器没有差异,并且所有的参数和结果都是标量类型,比如整型、字符型和布尔型,以上模型就可以正常工作。然而,在大型分布式系统中,存在多种不同类型的机器是司空见惯的。不同的机器对于数字、字符以及其他种类的数据项的表示方式常有不同。比如,IBM 大型机使用 EBCDIC 字符码,而 IBM 个人电脑使用 ASCII 字

符码。因此要以图 2.8 中所示的简单方式从 IBM PC 客户向 IBM 大型机传递字符型参数是不可能的，服务器将会错误地解释该参数中的字符。

类似的问题在整型数表示方式(是以 1 的补码还是以 2 的补码表示)以及浮点数表示方式方面也存在。另外，还存在另一个更为棘手的问题，就是字节排列次序问题。包括 Intel Pentium 在内的某些处理器是从右向左排列字节的，而 Sun SPARC 等处理器则相反。Intel 采用的格式称为 little endian 格式，而 SPARC 采用的格式称为 big endian 格式，这些格式的称呼来自于《格利佛游记》，书中描述的政治家决定开战的理由只是一方从大头来敲鸡蛋，而另一方从小头敲鸡蛋(Cohen 1981)。举例来说，考虑一个使用两个参数的过程，参数分别是一个整型数和一个包含 4 个字符的字符串。每个参数都需要占用一个 32 位的字。图 2.9(a)显示了基于 Intel Pentium 处理器的客户存根生成的消息中的参数部分的内容：第一个字包含整型参数，其值为 5，而第二个字中包含字符串“JILL”。

由于消息是逐字节(实际上是逐位)通过网络进行传输的，第一个发送的字节将第一个送达。在图 2.9(b)中显示了由 SPARC 接收到的消息内容。SPARC 将字节从左至右排列，0 字节放在最左边(高阶字节)；而不是像所有 Intel 处理器那样将字节从右至左排列，0 字节放在最右边(低阶字节)。当 SPARC 机器上的服务器存根读出地址分别是 0 和 4 的参数值时，它将得到一个值为 $83\ 886\ 080 (5 \times 2^{24})$ 的整型数，以及一个字符串“JILL”。

简单地将接收到的每个字中的字节顺序逆转过来，如图 2.9(c)所示，虽然很容易想到，但却是错误的。如果这样做，整型数虽然变回了 5，但是字符串却变成了“LLIJ”。关键问题是虽然机器间字节排序的差异把整型数弄错了，但是字符串并没有受到影响。如果没有用来识别字符串和整数的附加信息，就无法对该损害进行修复。

<table border="1"><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>5</td><td></td></tr><tr><td>7</td><td>6</td><td>5</td><td>4</td></tr><tr><td>L</td><td>L</td><td>I</td><td>J</td></tr></table>	3	2	1	0	0	0	5		7	6	5	4	L	L	I	J	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td>5</td><td>0</td><td>0</td><td>0</td></tr><tr><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>J</td><td>I</td><td>L</td><td>L</td></tr></table>	0	1	2	3	5	0	0	0	4	5	6	7	J	I	L	L	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td>0</td><td>0</td><td>0</td><td>5</td></tr><tr><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>L</td><td>L</td><td>I</td><td>J</td></tr></table>	0	1	2	3	0	0	0	5	4	5	6	7	L	L	I	J
3	2	1	0																																															
0	0	5																																																
7	6	5	4																																															
L	L	I	J																																															
0	1	2	3																																															
5	0	0	0																																															
4	5	6	7																																															
J	I	L	L																																															
0	1	2	3																																															
0	0	0	5																																															
4	5	6	7																																															
L	L	I	J																																															
(a)	(b)	(c)																																																

图 2.9 原始的消息和逆转后的消息(方框中的小数字代表每个字节的地址)

(a) Pentium 处理器上原始的消息；(b) SPARC 接收到的消息；(c) 进行逆转以后的消息

2. 传递引用参数

现在我们面临着一个难题：指针(广义来说还包括引用)如何传递？答案是：即使这种传递是可能的，也必定是极为困难的。要知道，指针只在使用该指针的进程所在的地址空间内才有意义。回头看看早先那个关于 read 的例子，如果第二个参数(缓冲区地址)在客户端为 1000，仅仅把 1000 这个数值传给服务器是没有用处的。服务器上的地址 1000 可能正好位于程序文本的中间。

解决这个问题的一种方法是完全禁止使用指针和引用，但是由于指针和引用是非常重要的，因此这种解决方法极为不可取。事实上，在上面那个 read 调用的例子中，客户存根知道作为第二个参数的指针指向的是一个字符数组。假定此时它还知道该数组的大

小,下面的解决策略显而易见:将数组复制到消息中,发送给服务器。服务器存根随后使用指向数组的指针来调用服务器程序,即使该指针拥有与 read 的第二个参数不同的数字值。服务器使用指针所作的改动(例如,在其中存储数据)直接影响了属于服务器存根的缓冲区。当服务器完成操作之后,服务器存根将原先的消息送回给客户存根,客户存根再将它复制回客户进程。这种方法实际上是用复制-还原调用来代替引用调用,虽然不完全相同,但是一般来说是行之有效的。

以下的优化策略可以使这种机制的效率提高一倍。如果存根知道缓冲区对服务器来说是输入参数还是输出参数,就可以省略其中的一个复制步骤。如果数组对服务器来说是输入参数(比如在 write 调用中就是这样),就不需要将数组复制回客户存根。如果数组是输出参数,那么客户存根最初发送的消息中就不用加进数组。

最后要指出的一点是:要充分认识到,虽然目前我们提出了处理指向简单数组和结构的指针的办法,但是我们还是无法处理一般意义上的指针,即指向任意数据结构,例如复杂图形的指针。某些系统试图将指针直接传递给服务器存根,然后在该服务器过程中生成特殊代码以使用这种指针。例如,服务器可以向客户发回一个请求,要求提供引用的数据。

3. 参数说明和存根生成

根据前面的解释可以清楚地看到,要隐藏对远程过程调用,必须使调用者和被调用者就互相交换的消息格式达成一致,并且在进行诸如传递复杂数据结构之类的操作时遵循相同的步骤。换句话说,RPC 的双方必须遵循相同的协议。

作为一个简单的例子,看一下图 2.10(a)中的过程。该过程包含 3 个参数,分别为字符、浮点数和包含 5 个整数的数组。假定一个字的长度为 4 个字节,RPC 协议可以规定:把字符放在一个字最右边的字节内传输(其余 3 个字节为空);浮点数占一个字;而数组由一组字组成,字的个数与数组长度相同,前面加上一个说明数组长度的字,如图 2.10(b)所示。在给出这些规则之后,名为 foobar 的客户存根就会知道它必须使用图 2.10(b)中规定的格式,而服务器存根也会知道输入的消息将使用图 2.10(b)中规定的格式。

```
foobar(char x;float y; int z[5])
{
    ...
}
```

(a)

foobar的局部变量	
x	
y	
5	
z[0]	
z[1]	
z[2]	
z[3]	
z[4]	

(b)

图 2.10 过程及其对应的消息

(a) 一个过程; (b) 与该过程相对应的消息

对消息格式的规定只是 RPC 协议内容中的一个方面。还需要让客户和服务器就诸如整型数、字符、布尔量等简单数据结构的表示方法达成一致。例如，协议可以规定整型数用 2 的补码形式表示，字符用 16 位 Unicode 编码表示，浮点数使用 IEEE 754 号标准规定的格式，并且所有数据都使用 little endian 格式存储。有了这些信息，就可以对消息进行明确的解释，而不会产生歧义。

在编码规则已经详尽到每一位之后，惟一剩下的事情就是让调用者和被调用者就消息的实际交换达成一致。例如，可以决定采用诸如 TCP/IP 之类的面向连接的传输服务，也可以采用不可靠的数据报服务。如果是后者的情况，必须由客户和服务器自己实现一套错误控制方案，并将该方案作为 RPC 协议的一部分。在实践中还存在许多其他方案。

在完整地定义了 RPC 协议之后，需要实现客户存根和服务器存根。幸运的是，相同协议所使用的存根在用于不同的过程时，不同点仅仅在于面向应用程序的接口。接口由一组由服务器实现的可供客户调用的过程组成，可以用编写客户或者服务器的同一种编程语言来编写（虽然严格说来这并不必要）。为了进一步简化，接口通常使用接口定义语言(interface definition language, IDL)来说明。用 IDL 说明的接口可以与适当的编译时接口或者运行时接口一起编译到客户存根过程和服务器存根中。

在实践中发现，使用接口定义语言可以显著简化基于 RPC 的客户-服务器应用程序。由于生成客户存根和服务器存根很容易，所有基于 RPC 的中间件系统都提供 IDL，以对应用程序开发提供支持。在某些情况下，更是要强制使用 IDL，这一点将在以后的章节中介绍。

2.2.3 扩展的 RPC 模型

远程过程调用已经成为分布式系统中事实上的通信标准。该模型之所以得以流行，应该归功于它显著的简明性。在本节中，我们将对原始 RPC 模型的两种扩展形式进行简要的讨论，设计这两种扩展形式的目的是为了克服 RPC 的某些缺点。

1. 门(door)

原始 RPC 模型假定调用者和被调用者之间通信的惟一手段是通过网络传递消息，一般说来这个假定是正确的。然而，如果客户和服务器驻留在同一台机器上，我们通常希望使用本地的进程间通信(IPC)功能，该功能是由底层操作系统向运行在同一台机器上的进程提供的。例如，在 UNIX 下这种功能包括共享内存、管道以及消息队列（要了解 UNIX 系统中 IPC 的详细信息，请参见文献(Stevens 1999)）。

本地 IPC 功能与基于网络的类似功能相比要高效得多，即使在后者用于同一台机器上的进程间通信的情况下也是如此。因此，如果性能方面要求较高，就需要根据要处理的进程是否位于同一台机器上的具体情况，结合使用多种进程间通信机制。

作为一种折衷的方法，一些操作系统提供一个与 RPC 等价的机制，供位于同一台机器上的进程使用，这种机制称作门(door)。门是一类过程的总称，这种过程位于某个服务器进程的地址空间内，可由与该服务器进程位于同一台机器上的其他进程调用。门原先是为 Spring 操作系统设计的(Mitchell 等 1994)，在文献(Hamilton 和 Kougouris 1993)中对门进行了详细描述。Bershad 等人于 1990 年开发了另一种称作 Lightweight RPC

的类似机制。

要对门进行调用，必须有本地操作系统提供支持，如图 2.11 所示。具体来说，首先，服务器进程必须对门进行注册，随后其他进程才能调用这个门。在注册一个门的时候，系统返回该门的标识符，该标识符可在以后用来为门赋予一个符号名称。注册是通过调用 door_create 完成的。如果要让其他进程调用注册过的门，只需给该门取个名字，并将该名字与该门在注册时得到的标识符关联起来即可。例如，在 Solaris 中，每个门都拥有一个文件名，该文件名是通过调用 fattach 而与该门的标识符关联起来的。客户利用系统调用 door_call 来请求对门进行调用，并在 door_call 中给出要调用的门的标识符以及其他必需的参数。操作系统随后向上调用注册该门的服务器进程。对服务器进程的向上调用导致服务器调用门。调用门所得到的结果通过系统调用 door_return 返回客户进程。

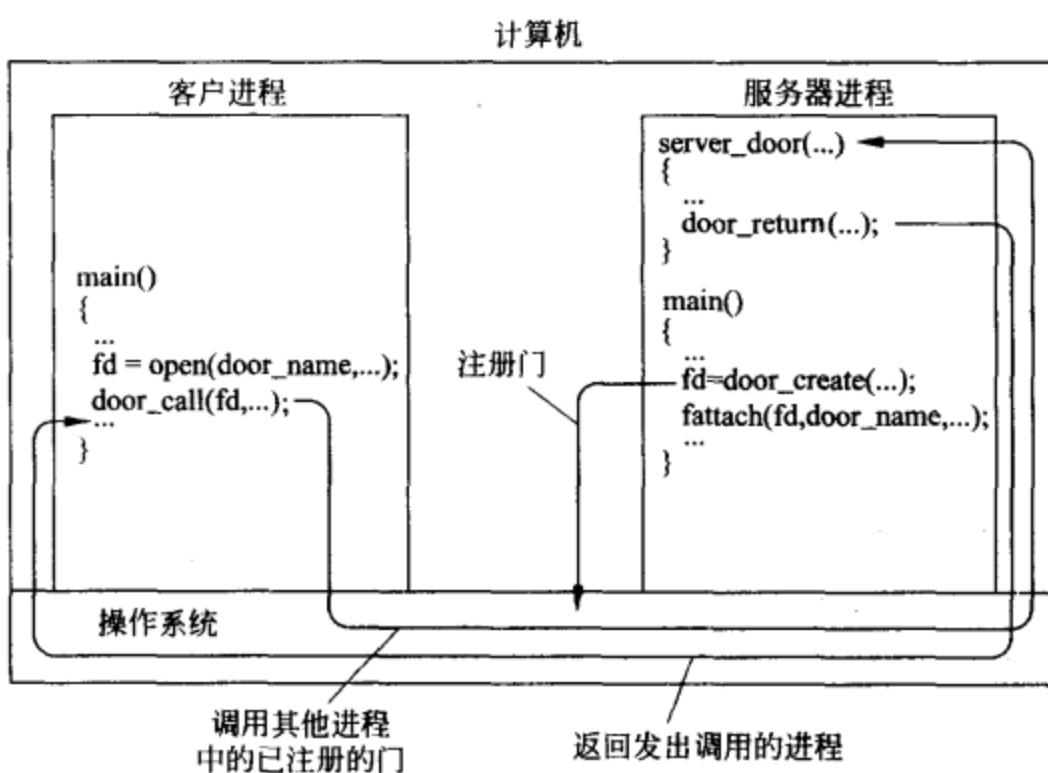


图 2.11 门(一种 RPC 机制)的使用原理

门的主要好处在于它允许在分布式系统的通信中使用一种单一的机制，即过程调用。不幸的是，这种情况下应用程序开发人员必须了解哪些调用是在当前进程内部完成的，哪些调用是对本地机器上的不同进程发出的，而又有哪些调用是向远程进程发出的。

2. 异步 RPC

和在常规过程调用中的情形一样，当客户调用远程过程时，客户将会阻塞，直到有应答返回为止。在没有结果要返回的情况下这种严格的请求-应答方式是不必要的，它只会导致客户过程向远程过程发出调用请求之后处于阻塞状态，从而无法进行本来能够进行的其他有用的工作。不需要等待应答的例子有：从某个账户向另一个账户进行转账操作、向数据库添加条目、启动远程服务以及执行批处理操作，等等。

针对以上这些状况，RPC 系统可以提供称为异步 RPC(asynchronous RPC)的功能。利用该功能，客户可以在发出 RPC 请求后立即继续执行。在异步 RPC 中，服务器在接收到

到 RPC 请求后立即向客户送回应答,之后再调用客户请求的过程。应答的作用是向客户确认服务器已准备开始处理该 RPC 请求。客户接收到服务器的确认消息之后,将不会阻塞,而是继续向下执行。图 2.12(b)中显示了异步 RPC 中客户和服务器的交互过程。作为对比,图 2.12(a)中显示了常规 RPC 的请求-应答过程。

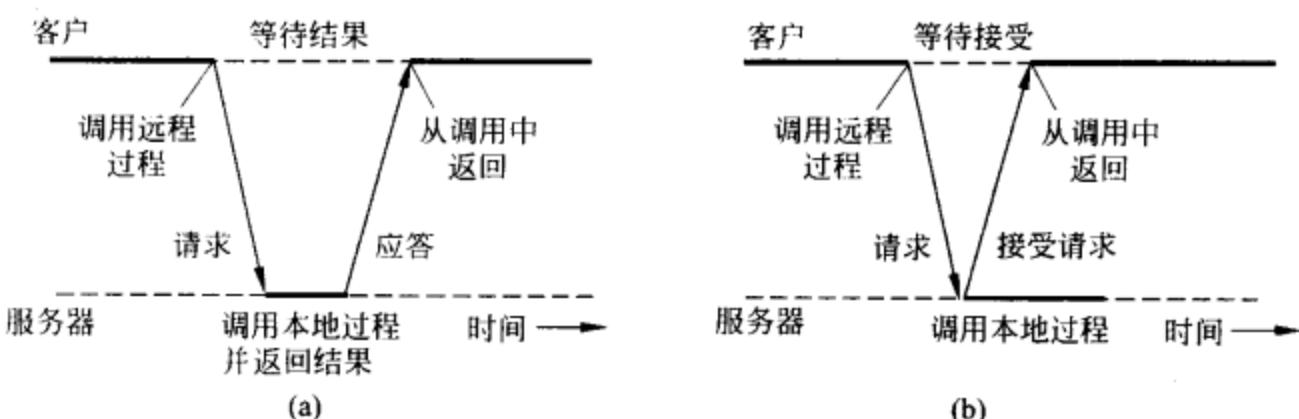


图 2.12 RPC 中客户-服务器间的交互过程
(a) 传统 RPC 中客户-服务器间的交互过程; (b) 使用异步 RPC 时的交互过程

如果在应答返回时客户还未做好接收的准备,从而没有做出任何动作,在这种情况下异步 RPC 也很有用。例如,客户可能想预取一组主机的网络地址以便随后与它们进行联络。当命名服务正在搜集这些地址的时候,客户可能想做其他事情。在这种情况下,通过两步异步 RPC 来进行客户与服务器间的通信是很有意义的,如图 2.13 所示。客户首先对服务器进行调用,把要查询的主机名清单交给服务器,并且客户在接收到服务器关于已收到主机名清单的确认之后继续执行其他的程序。第二步调用是由服务器发出的,它对客户进行调用,将查询到的地址清单交给客户。两步异步 RPC 结合起来也称作延迟的同步 RPC(deferred synchronous RPC)。

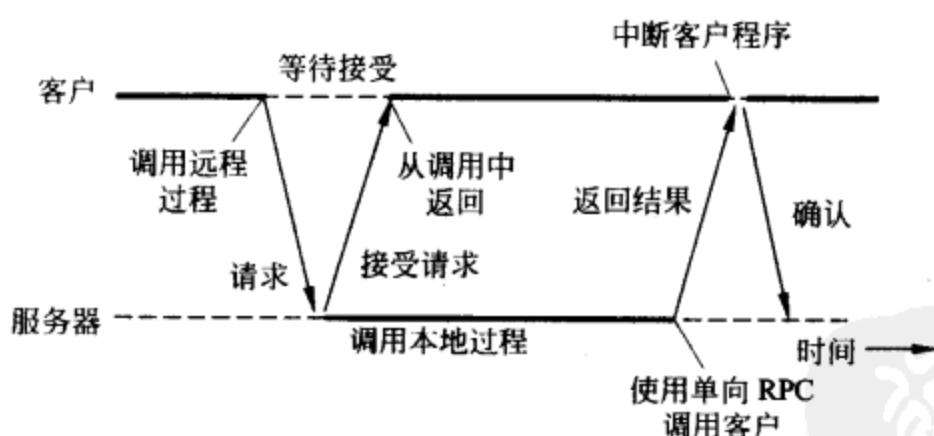


图 2.13 客户和服务器通过两个异步 RPC 进行交互

需要指出的是,还存在一些异步 RPC 的变异形式。在这些异步 RPC 中,客户向服务器发送请求之后立即继续执行其他的程序。也就是说,客户不等待服务器返回接受请求的确认。我们称这种 RPC 为单向 RPC(one-way RPC)。它的问题在于,如果无法确保可靠性,客户就无法确定它发出的请求是否将得到处理。我们将在第 7 章中继续讨论这些问题。

2.2.4 实例：DCE RPC

远程过程调用已广泛用作大部分中间件和分布式系统的基础。在本节中，我们重点讨论一个具体的 RPC 系统：分布式计算环境 (distributed computing environment, DCE)。DCE 是由开放软件基金会 (Open Software Foundation, OSF) 开发的，该组织现在称作开放开发组 (Open Group)。DCE RPC 的流行程度比不上其他某些 RPC 系统，特别是 Sun RPC。但是，DCE RPC 是 RPC 系统的一个典型代表，其规范得到了 Microsoft 的分布式计算基本系统的采用。另外，DCE RPC 可以很好地说明 RPC 系统与分布式对象间的联系，这点在后面一节中可以看出。我们首先简要介绍 DCE，随后仔细考虑 DCE RPC 的主要工作机制。

1. DCE 介绍

DCE 是设计用来作为在现有的(网络)操作系统与分布式应用程序之间的中间抽象层来执行的。从这个意义上来说，DCE 是真正的中间件系统。DCE 最初是为 UNIX 设计的，但是现在已经移植到所有主要操作系统上，包括 VMS、Windows NT 以及桌面操作系统在内。它的主要思想是，使用者可以在现有的一组机器上安装 DCE 软件，然后就可以运行分布式应用程序，而不会对现有的(非分布式的)应用程序造成影响。虽然大部分 DCE 包在用户空间运行，但是在采用某些特定配置的情况下，必须在内核中添加一部分内容(分布式文件系统的一部分)。开放开发组自己只是出售源代码，由销售商将源代码集成进他们的系统中去。

所有 DCE 的底层编程模型都是客户-服务器模型，该模型在第 1 章中已经详细讨论过了。用户进程就像客户那样对服务器进程提供的远程服务进行访问。其中的某些服务属于 DCE 的一部分，而其他服务属于应用程序，由应用程序开发人员编写。客户和服务器间的所有通信都通过 RPC 进行。

DCE 本身的一部分是由多个服务构成的。分布式文件服务 (distributed file service) 是一种全球规模的文件系统，可以提供一种透明的访问方式，允许使用相同手段访问系统中任意文件。分布式文件服务既可以建立在主机的本地文件系统之上，也可以用来代替本地文件系统。目录服务 (directory service) 是用来跟踪系统中所有资源的位置的，这些资源包括机器、打印机、服务器、数据等，它们在地域上可以分布在全世界范围内。目录服务允许一个进程在不了解某个资源所在位置的情况下(除非进程关心这件事)请求该资源。安全服务 (security service) 允许对所有类型的资源进行保护，这样就可以将访问权提供给经过授权的用户。最后，分布式时间服务 (distributed time service) 可以使位于不同机器上的时钟保持全局同步。正如我们将在后续章节中介绍的那样，全局时钟的思想使得在分布式系统中确保一致性变得更加容易。

2. DCE RPC 的目标

DCE RPC 系统的目标是比较传统的。首先同时也是最重要的是，RPC 系统使客户可以通过简单的本地过程调用来自访问远程服务。这种接口使得客户程序(例如应用程序)

能够以一种简单的、为多数程序员所熟悉的方式来编写,还使得大量现有的代码不需要进行改动(或者只需要极少的改动)就能够在分布式环境下运行。

RPC 系统负责向客户隐藏所有细节,并且也在某种程度上对服务器进行细节隐藏。RPC 系统首先自动找到恰当的服务器,随后在客户软件和服务器软件之间建立通信(这个过程一般称为绑定(binding))。它还能处理双向数据传输,在必要的情况下把消息分割成片段,然后再将片段重组还原成消息(比如说,在参数之一是大数组的情况下)。最后,RPC 系统可以自动完成客户和服务器间的数据类型转换,这样,即使客户和服务器运行在体系结构不同而且字节排序方式不同的机器上也没有问题。

由于 RPC 系统具有隐藏细节的能力,所以客户和服务器是彼此高度独立的。可以用 Java 编写客户程序而用 C 编写服务器程序,反之亦然。客户和服务器可以运行于不同的硬件平台上,并且可使用不同的操作系统。它还支持非常多的网络协议和数据表示方式,而不需要客户和服务器进行任何干预。

3. 编写客户程序和服务器程序

DCE PRC 系统由许多组件组成,其中包括语言组件、库、守护程序以及工具程序等。配合使用这些组件就可以编写客户程序和服务器程序。本节将对各个部分以及它们相互配合的方式进行描述。图 2.14 总结了编写以及使用一个 RPC 客户程序和服务器程序的完整过程。

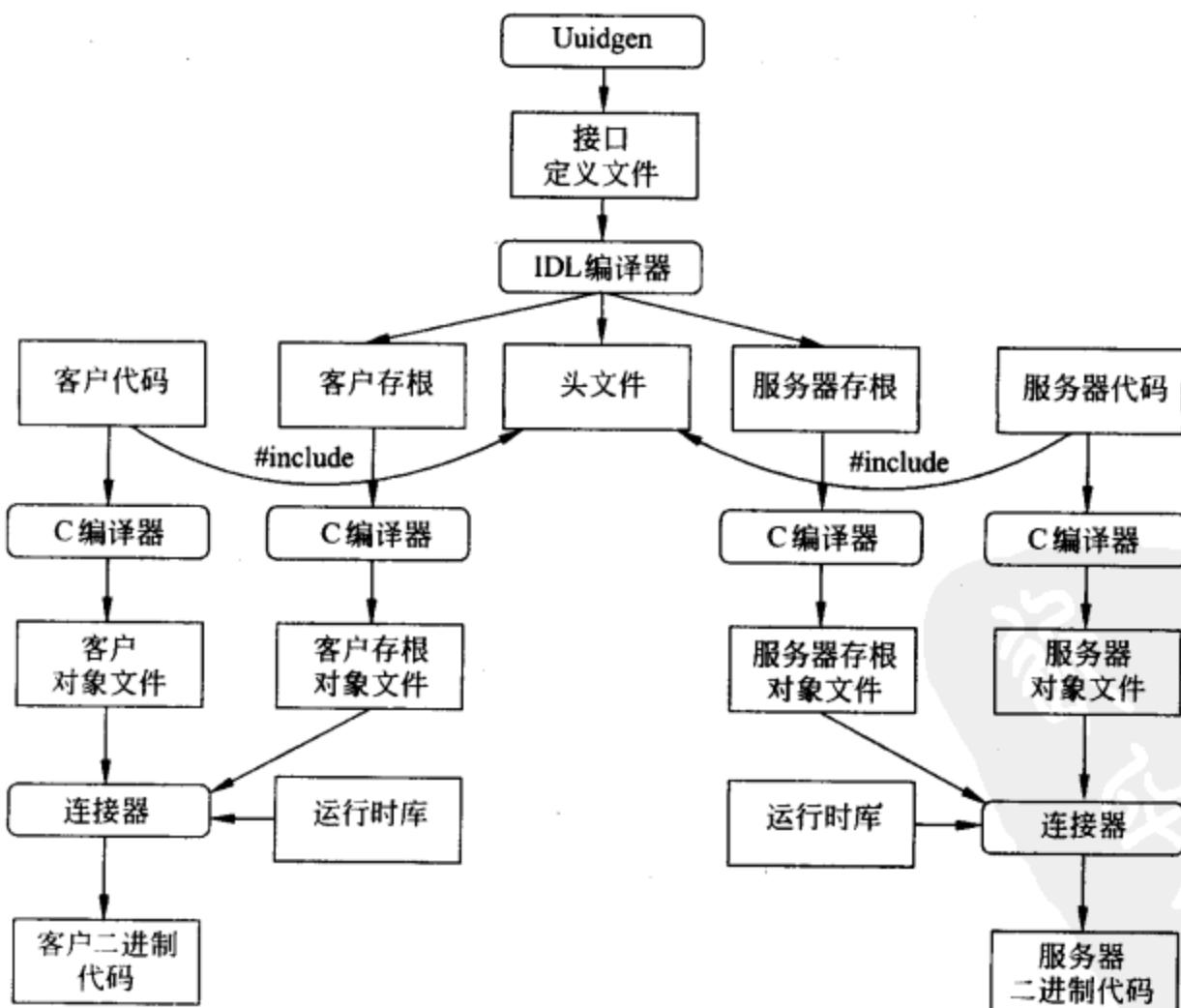


图 2.14 用 DCE RPC 编写客户程序和服务器程序的步骤

在客户-服务器系统中,将各个部分结合到一起的是使用接口定义语言 IDL 描述的接口定义。这种语言允许以极其类似于 ANSI C 语言中的函数原型的形式来对过程进行说明。IDL 文件也可以包含类型定义、常量声明以及正确进行参数编组和结果解编时所必需的其他信息。理想情况下,接口定义中还应该含有过程所完成的工作的形式化定义,但是这样的定义在目前的技术条件下还是无法做到的。因此接口定义只是规定了调用的格式,而不包括调用的语义。编写者顶多能够加入少量注释,以描述该过程要完成的工作。

每个 IDL 文件中关键的部分是指定接口的全局惟一标识符。在第一个 RPC 消息中,客户把该标识符发送给服务器,由服务器检验该标识符是否正确。如果当某个客户无意中试图绑定到错误的服务器上或者是正确服务器的一个过时版本上时,服务器将会检测到该错误,绑定将不会建立。

接口定义和惟一标识符在 DCE 中是紧密联系在一起的。就像图 2.14 中显示的那样,编写服务器或客户应用程序的第一步常常是调用 `uuidgen` 程序,要求该程序生成一个原型 IDL 文件,其中包含有一个接口标识符,该标识符保证不会在由 `uuidgen` 生成的任何其他接口中再次使用。在该标识符的编码过程中加进了它所在的位置和创建时间,从而确保了它的惟一性。该标识符是一个 128 位的二进制数,在 IDL 文件中表示为一个十六进制的 ASCII 字符串。

下一步是编辑 IDL 文件,填入远程过程名以及它们使用的参数。值得注意的是,RPC 并不是完全透明的(例如,客户和服务器无法共享全局变量),而 IDL 的定义规则已经保证不可能表示那些不支持的结构。

IDL 文件编写完毕之后,可以调用 IDL 编译器来编译它。IDL 编译器的输出包括下列 3 个文件:

- (1) 头文件(例如,C 语言形式的 `interface.h`);
- (2) 客户存根;
- (3) 服务器存根。

头文件中包含惟一标识符、类型定义、常量定义以及函数原型。应该在客户和服务器代码中使用 `#include` 语句将该头文件包含进去。客户存根包含供客户程序调用的实际过程,这些过程负责参数搜集,并且将搜集到的参数打包进准备送出的消息中,随后调用运行时系统来发送该消息。客户存根还负责将结果值从应答中提取出来,返回给客户。服务器存根包含有一些过程,在输入的消息到达时,由服务器所在机器上的运行时系统调用这些过程,这些过程随后再调用服务器过程来完成实际操作。

接下来是由应用程序编写人员编写客户和服务器代码。要对服务器、客户以及双方的存根进行编译。然后将得到的客户代码和客户存根的目标文件与运行时库连接起来,从而得到客户程序的二进制可执行形式。与之相仿,将服务器代码和服务器存根的代码进行编译和连接,可以得到服务器程序的二进制可执行形式。在运行时,把客户程序和服务器程序都启动起来,这样应用程序就可以正确执行了。

4. 将客户绑定到服务器

为了让客户能够调用服务器,必须先对服务器进行注册,然后服务器做好接收输入调用的准备。服务器进行了注册之后,客户就能够对服务器进行实际定位,然后绑定到服务器。服务器定位分两步进行:

- (1) 定位服务器所在的机器;
- (2) 定位该机器上的服务器(也就是正确的进程)。

第 2 步稍微有些复杂,但基本上可以归结为这样一个问题:为了与服务器通信,客户需要知道服务器所在机器上的一个端点(endpoint),以便向服务器所在机器发送消息。服务器的操作系统使用端点(通常称作端口,port)来判别输入的消息要调用哪一个进程。在 DCE 中,每一台运行服务器的机器上都由 DCE 守护程序(DCE daemon)维护一张(服务器,端点)对列表。服务器在可以接收输入请求之前,必须先请求操作系统分配一个端点,随后通过 DCE 守护程序注册该端点。DCE 守护程序在端点列表中记录下有关信息(包括服务器采用何种协议)供以后使用。

服务器同时还需要通过目录服务进行注册,方法是给出服务器所在机器的网络地址以及在查询该服务器时使用的名字。把客户绑定到服务器的过程如图 2.15 所示。

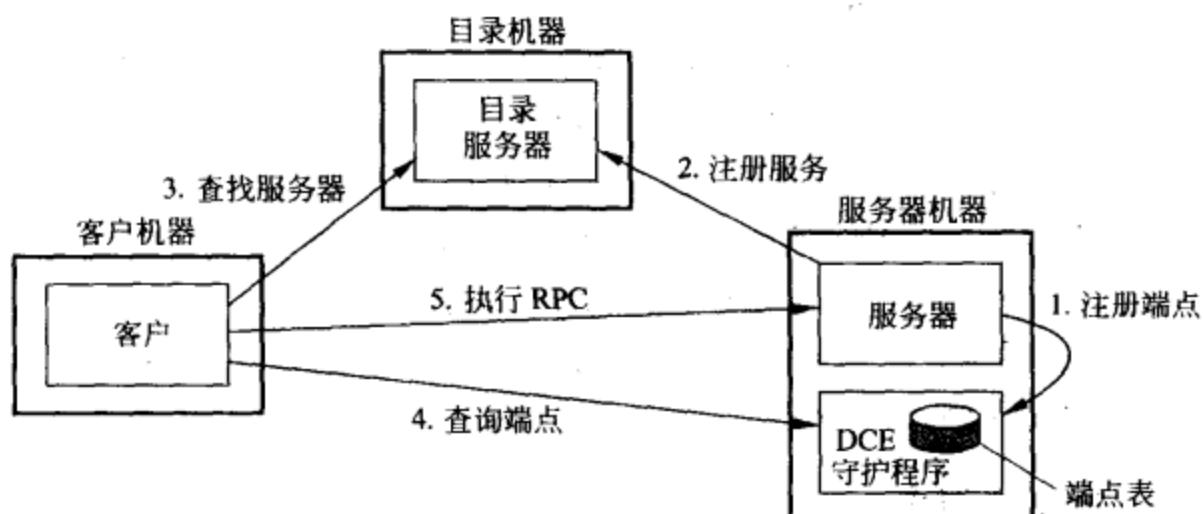


图 2.15 DCE 中的客户-服务器绑定过程

假定客户希望绑定到一个视频服务器上,该服务器的本地名是/local/multimedia/video/movies。客户把该名字传递给目录服务器,目录服务器返回运行视频服务器的机器的网络地址。客户随后向该机器上的 DCE 守护程序(它的端点是公开的)发出请求,要求它在端点列表中查询视频服务器使用的端点。利用所获得的信息,就可以执行 RPC 了。在随后的执行 RPC 的过程中,不再需要进行查询。如果有必要,DCE 还允许客户为了查找一个合适的服务器执行较为复杂的搜索。另外,还可以选择执行安全的 RPC。

5. 执行 RPC

实际的 RPC 过程是以常规方式透明地执行的。客户存根对参数进行编组,然后把它们交给运行时库,以使用在绑定时选定的协议来传输这些参数。当消息送达服务器端的时候,系统根据输入的消息中包含的端点信息将该消息发送给相应的服务器。运行时库

将消息传递给服务器过程,由服务器过程将参数解编,然后调用服务器。应答消息的返回过程正好相反。

DCE 提供了一些语义选项。默认的选择是至多一次操作(at-most-once operation),这种情况下同一个调用的执行不能超过一次,即使系统崩溃时也是如此。在实践中,这意味着如果服务器在 RPC 过程中崩溃,随后又迅速恢复,客户不再重复刚才的操作,因为担心该操作可能已经执行过一次了。

作为另一种解决方法,也可以把某个远程过程标记为幂等的(idempotent)(在 IDL 文件中声明),这意味着可以重复多次执行该过程,而不会造成任何不良后果。例如,从文件中读取指定的某个块的操作就可以反复尝试多次,直到成功为止。如果由于服务器崩溃而导致一个幂等的 RPC 失败,那么客户可以等待服务器重启完毕,然后再次进行尝试。还有其他一些语义(但是极少使用),包括对本地网络上的所有机器进行 RPC 广播等。在第 7 章中讨论 RPC 的故障问题时会继续探讨 RPC 语义。

2.3 远程对象调用

基于对象的技术已经展示了它在开发非分布式应用程序方面的价值。对象最重要的特征之一就是通过定义良好的接口向外界隐藏其内部结构。这种机制保证了只要保持对象接口不变,就可以方便地替换或者修改对象。

当 RPC 机制逐渐成为分布式系统通信处理的事实标准的时候,人们开始认识到 RPC 的原理同样可以应用于对象。在本节中,我们把 RPC 的思想拓展到对远程对象的调用上,并且说明与 RPC 相比,这种方法是如何增强分布透明性的。本节仅仅集中于讨论相对简单的远程对象。在第 10 章中,将详细讨论包括 CORBA 和 DCOM 在内的多种基于对象的分布式系统。CORBA 和 DCOM 提供的对象模型比在本节中要讨论的模型更加重要,并且具有更强大的功能。

2.3.1 分布式对象

对象的关键特性是对数据的封装,这些被封装的数据称作状态(state),对这些数据的操作称作方法(method)。只有通过接口(interface)才能使用方法。理解以下这一点很重要:一个进程除了可以通过对象的接口来引用它可用的方法以外,不存在别的合法途径来访问或者操纵对象的状态。一个对象可以实现多个接口。同样,在给定一个接口定义的情况下,也可以有若干个对象提供该接口的实现。

将接口与实现这些接口的对象分离开来对于分布式系统是至关紧要的。由于进行了严格的分离,因此可以将接口放在一台机器上,而让对象本身驻留在另一台机器上。这种组织结构如图 2.16 所示,通常称作分布式对象(distributed object)。

当一个客户绑定(bind)到一个分布式对象的时候,该对象接口的一种实现——称作代理(proxy)——被加载到客户的地址空间中。代理与 RPC 系统中的客户存根相类似。代理惟一的工作是将对方法的调用编组成消息,并且对应答消息进行解编,将调用的方法得到的结果返回给客户。实际对象驻留在服务器所在机器上,它向服务器提供的接口与

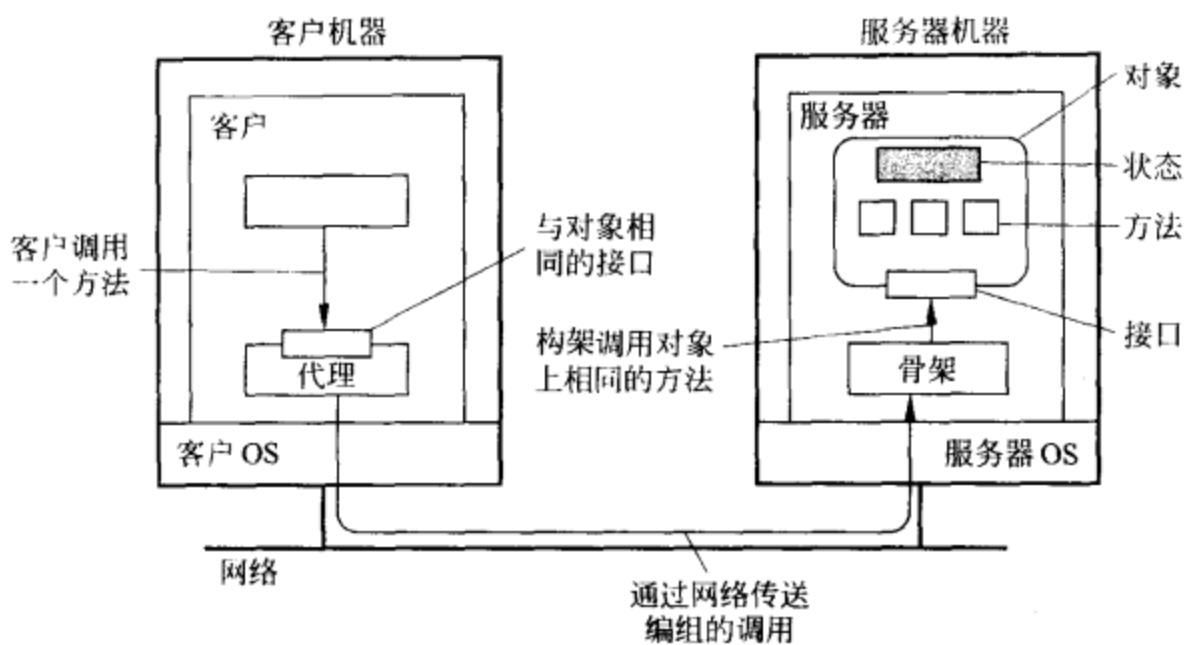


图 2.16 使用客户端代理的远程对象的一般组织结构

向客户提供的接口相同。输入的调用请求首先传递给服务器存根——这种存根也称作骨架(skeleton)，它将调用请求解编成对服务器上对象接口的恰当的方法调用。服务器存根同时还负责对应答消息进行编组，并且将应答消息发送回客户端代理。

多数分布式对象所特有的、但同时又是直观难以想象的特性是：它们的状态并不是分布的；它驻留在单个机器上，只有由该对象实现的接口可以在其他机器上使用。这种对象也称为远程对象(remote objects)。我们将会在后续章节中介绍，在普通的分布式对象中，状态本身可能是物理上分布于多台机器之上的，但是这种分布对于客户端来说是隐藏在对象接口后面的。

1. 编译时对象与运行时对象

分布式系统中的对象可能以多种形式出现。最常见的一种对象形式直接与语言级对象(比如那些 Java、C++ 以及其他面向对象语言直接支持的对象)相关联，这种对象称作编译时对象。在这种情况下，对象定义为类的实例。类(class)是对一种抽象类型的描述，是含有数据元素以及对这些数据元素的操作的模块(Meyer 1997)。

在分布式系统中使用编译时对象常常使得构建分布式应用程序更加方便。比如说，在 Java 中，可以通过定义类及该类实现的接口来完整地定义一个对象。对该类定义进行编译，得到可以实例化 Java 对象的代码。对接口进行编译可以得到客户端存根以及服务器端存根，用来从远程机器上引用 Java 对象。Java 开发人员大多数时候都看不到对象的分布，所看到的只有 Java 程序的代码。

编译时对象的一个显而易见的缺陷是对特定编程语言的依赖性。因此，另一种构建分布式对象的方法是在运行时显式生成对象。由于这种方案独立于编写分布式应用程序时所使用的编程语言，许多基于对象的分布式系统都使用它。应特别指出的是，可以使用由多种不同语言编写的对象来构建应用程序。

在与运行时对象打交道的时候，实现实际对象的方法是多种多样的。比如说，开发人

员可以选择用 C 语言编写一个库,其中包含许多函数,这些函数可以处理常规的数据文件。根本问题在于,如何让这样的实现表现为一个对象,以便从远程机器调用其方法。一种常用的方案是使用一种对象适配器(object adapter),它可以作为实现的一种包装,惟一的作用就是让该实现呈现出对象的外部特征。适配器(adapter)这个术语来自于(Gamma 等 1994)中描述的一种设计模式,这种设计模式用来将接口转换成客户期望的某种东西。例如,动态绑定到上面所说的那个 C 库,随后打开代表对象当前状态的对应数据文件的适配器就是对象适配器。

对象适配器在基于对象的分布式系统中扮演了一个重要角色。为了使包装尽可能简单,对象根据它们实现的接口来惟一定义。接口的实现可以在适配器中注册,随后适配器可以使该接口可供(远程)调用。适配器会注意到有调用请求发出,随后向客户提供远程对象的映像。我们将在下一章中继续讨论对象服务器和对象适配器的组织结构。

2. 持久对象和暂时对象

除了可以将对象分为语言级对象和运行时对象以外,还可以将对象分为持久对象(persistent object)和暂时对象(transient object)。持久对象无论当前是否位于服务器进程的地址空间内,都始终保持存在。换句话说,持久对象并不依赖于当前的服务器。这意味着当前管理持久对象的服务器在退出运行之前,可以先把持久对象的状态存储到辅助存储器上;之后,新启动的服务器可以由存储器将对象的状态读到自己的地址空间中,对调用请求进行处理。与持久对象相反,暂时对象只在管理该对象的服务器存在的期间存在,服务器退出运行后,该对象也就不再存在。在是否应该使用持久对象的问题上存在争论;有些人认为使用暂时对象就足够了。在第 9 章中讨论基于对象的分布式系统时,我们将继续详细讨论这个问题。

2.3.2 将客户绑定到对象

传统 RPC 系统与支持分布式对象的系统有一个很有趣的不同点,就是后者一般都提供系统范围内的对象引用。这种对象引用(比如作为方法调用的参数)可以在位于不同机器上的进程之间自由传递。通过将对象引用的实际实现隐藏起来,也就是说使其变得不透明,甚至变成引用对象的惟一途径,与传统 RPC 系统相比,分布透明性增强了。

如果进程获得了某个对象的引用,那么在调用该对象的任何方法之前必须首先绑定到该对象。绑定将产生一个位于该进程地址空间内的代理,它实现了包含有此进程所能调用的方法的接口。多数情况下,绑定是自动进行的。如果给底层系统一个对象引用,它就需要通过某种途径来定位管理该实际对象的服务器,随后将代理放置在客户地址空间中。

通过隐式绑定(implicit binding),可向客户提供一种简单机制,该机制允许客户在只使用对象引用的情况下直接进行方法调用。比如说,C++ 允许重载一元成员选择操作符(“`>>`”),这样就可以引入对象的引用,就好像它们是通常的指针一样,如图 2-17(a)所示。通过隐式绑定,可以在引用被连接到实际对象上的时候将客户透明地绑定到对象上。与之相反,如果使用显式绑定(explicit binding),客户必须首先调用某个特殊函数来绑定到

对象上,随后才能调用该对象的方法。显式绑定一般返回指向代理的指针,该代理可以在本地使用,如图 2.17(b)所示。

```
Distr_object * obj_ref; // 声明一个系统范围的对象引用  
obj_ref = ...; // 对一个分布对象初始化引用  
obj_ref->do_something(); // 隐式绑定并调用方法
```

(a)

```
Distr_object obj_ref; // 声明一个系统范围的对象引用  
LocalObject * obj_ptr; // 声明一个指向本地对象的指针  
obj_ref = ...; // 对一个分布对象初始化引用  
obj_ptr = bind(obj_ref); // 显式绑定并得到本地代理指针  
obj_ptr->do_something(); // 在本地代理上调用方法
```

(b)

图 2.17 隐式绑定和显式绑定的例子

(a) 隐式绑定的例子,仅使用全局引用;(b) 显式绑定的例子,同时使用全局引用和局部引用

对象引用的实现

很明显,对象引用必须包含足够的信息,以便使客户可以绑定到服务器上。一个简单的对象引用包含实际对象驻留的机器的网络地址,以及端点(用来标识管理该对象的服务器),还有对象的标号。对象的标号一般由服务器提供,比如说可以用 16 位数来表示。这里的端点与 DCE RPC 系统中的端点完全一样,对应于由服务器的本地操作系统动态分配的某个本地端口。然而,这种对象引用存在诸多缺陷。

首先,如果服务器所在机器崩溃,而在机器恢复后又给服务器分配一个与原先不同的端点的话,所有原来的对象引用就都变得无效了。这个问题可以通过与在 DCE 中所使用的同样的办法解决:在每台机器上驻留一个本地守护程序,监听某个公开的端点,并且通过端点表来追踪服务器与端点的对应关系。在将客户绑定到对象上之前,首先要请求守护程序给出服务器当前所用的端点。如果使用这种方案,就需要将服务器 ID 编码进对象引用中,该对象引用可作为端点表中的索引,还要求服务器通过本地守护程序进行注册。

然而,将服务器所在机器的网络地址编码进对象引用中并不是一个好主意。这种方法存在的问题是,在不破坏服务器所管理的所有对象引用的前提下,无法将服务器移到另一台机器上去。一个容易想到的解决方法是对维护端点表的守护程序的思路进行拓展,使用定位服务器(location server)来追踪管理对象的服务器当前在哪一台机器上运行。在对象引用中不仅包含定位服务器的网络地址,而且还含有服务器的系统级标识符。我们在第 4 章中将介绍,这种解决方法同样也存在很多严重缺陷,特别是在侧重考虑可扩展性的情况下更是如此。

前面我们做了这样一个默认的假定:客户和服务器都已经配置为使用相同的协议栈。这不仅意味着它们使用相同的传输协议,比如 TCP,而且意味着它们必须使用相同的协议进行参数编组和解除编组。而且还必须使用相同的协议来建立初始连接,以相同

方式进行错误处理和流控制，等等。

如果我们能够从对象引用中得到更多的信息，我们就能够放心地丢掉这个假定。所需要的信息包括用来绑定到一个对象的协议标识，以及那些由负责对象管理的服务器支持标识。比如说，某个服务器可能支持通过 TCP 连接和 UDP 数据报同时传入数据，此时客户就必须负责为对象引用中标出的至少一种协议提供代理实现。

我们对以上方案进行进一步的改进，在对象引用中加入实现句柄(implementation handle)，它指向代理的完整实现，客户可以在绑定到对象的时候动态加载这些代理。比如说，实现句柄可以采用 URL 的形式，指向档案文件，例如 `ftp://ftp.clientware.org/proxies/java/proxy-v1.la.zip`。绑定协议随后只需要规定这种文件必须动态下载、解包、安装并且实例化。这种方案的优点在于，客户端不需要关心它是否拥有针对于某种特定协议的实现。另外，它还赋予对象开发者自主设计针对特定对象的代理的自由。然而，我们必须采取专门的安全措施来确保客户能够信任下载的代码，我们将在第 8 章中讨论这一点。

2.3.3 静态远程方法调用与动态远程方法调用

在将客户绑定到对象之后，就可以通过代理来调用对象的方法。这种远程方法调用(remote method invocation)简称为 RMI，它在参数编组及传递方面十分接近于 RPC。RMI 与 RPC 本质上的不同在于，RMI 一般支持系统级对象引用，就像我们前面所介绍的那样。另外，RMI 不需要使用通用的客户端和服务器端存根，而可以更方便地使用针对特定对象的存根，这一点我们前面也谈到了。

提供 RMI 支持的常规方法是利用接口定义语言来指定对象的接口，与 RPC 中采用的方法相似。另外，我们也可以使用诸如 Java 之类的基于对象的语言来自动生成存根。使用预先确定的接口定义的方法一般称作静态调用(static invocation)。静态调用要求在开发客户应用程序的时候就已知对象接口，这也意味着如果接口发生变化，就必须重新编译客户应用程序，然后才能使用新的接口。

另一种方法是采用更为动态的方式来进行方法调用。在实践中，有时在运行时建立方法调用会更为方便，这种做法称为动态调用(dynamic invocation)，它与静态调用的本质区别在于，使用动态调用的应用程序直到运行时才对需要在远程对象上调用的方法做出选择。动态调用的常用形式如下面的例子所示：

```
invoke(object, method, input_parameters, output_parameters);
```

其中参数 `object` 代表分布式对象，参数 `method` 指定要调用的方法，`input_parameters` 是存储方法需要的输入参数值的数据结构，而 `output_parameters` 是存储输出值的数据结构。

举个例子，考虑在文件对象 `fobject` 的末尾添加整型数 `int`，该对象实现了方法 `append`。在这种情况下，静态调用的形式是：

```
fobject.append(int)
```

而相应的动态调用的形式可能会是：

```
invoke(fobject, id	append), int)
```

其中操作 `id.append` 返回方法 `append` 的标识符。

为了进一步说明动态调用的用处,我们来研究一下对象浏览器,它可以用来察看一组对象。我们假定该浏览器支持远程对象调用。这种浏览器能够绑定到分布式对象上,而且可将分布式对象的接口提供给用户,随后要求用户选择一个方法,并且提供该方法的输入参数值,然后浏览器就可以执行实际调用了。典型地,开发这种对象浏览器时应该考虑到为任何可能的接口提供支持,这就要求在运行时检查接口并动态建立方法调用。

另一个应用了动态调用的例子是批处理服务。这种服务可以处理指定了调用执行时间的调用请求。该服务可以通过调用请求队列的形式实现,在队列中根据请求中指定的执行调用时间的先后来对调用请求进行排序。该服务中主循环的结构是:简单地等待直到下一次调用的时刻来临,在调用前将该调用请求从队列中移出,然后调用上面代码中的 `invoke`。

2.3.4 参数传递

由于多数 RMI 系统都支持系统级对象引用,因此对方法调用中的参数传递的要求不像 RPC 中那么严格。然而,也有一些细微的东西会使得 RMI 变得比预想的复杂,我们将在下面对这些问题进行简要的讨论。

我们首先考虑以下这种状况:只有分布式对象存在,换句话说也就是系统中的所有对象都可以通过远程机器访问。在这种情况下,我们可以一成不变地使用对象引用作为方法调用的参数。引用通过值来进行传递,从一台机器复制到另一台机器上。当把作为方法调用的结果的对象引用赋予进程时,该进程就可以在必要的时候简单地绑定到引用的对象上去。

不幸的是,只使用分布式对象可能造成效率降低,特别是在对象规模比较小的情况下,比如当对象是整型数或者布尔值的时候。如果客户引用的是位于另一服务器上的对象,那么它执行的每一次调用所生成的请求都要跨越不同地址空间,甚至要跨越不同的机器。因此,指向远程对象的引用和指向本地对象的引用要区别对待。

在使用对象引用作为参数来进行方法调用时,只有在该引用指向远程对象的情况下,才将对象引用作为值参数来复制和传递。在这种情况下,对象是完全通过引用来传递的。然而,如果引用指向的是本地对象,也就是位于与客户相同的地址空间中的对象,那么被引用对象将完整地进行复制,并且与调用一起传递。也就是说,此时对象是通过值传递的。

这两种不同的情况如图 2.18 中所示,图中显示了运行于机器 A 上的客户程序,以及运行于机器 C 上的服务器程序。客户有一个对本地对象 O1 的引用,在调用位于机器 C 上的服务器程序时将该引用作为调用的参数。另外,它还有一个对驻留在机器 B 上的远程对象 O2 的引用,这个引用也是用来作为参数的。在对服务器进行调用的时候,系统将 O1 的一份拷贝以及指向 O2 的引用的拷贝一同传递给机器 C。

要注意,正在处理的引用指向的是本地对象还是远程对象这一点可以是高度透明的,比如在 Java 中就是这样。在 Java 中,二者间可见的惟一不同点只是:本地对象的数据类

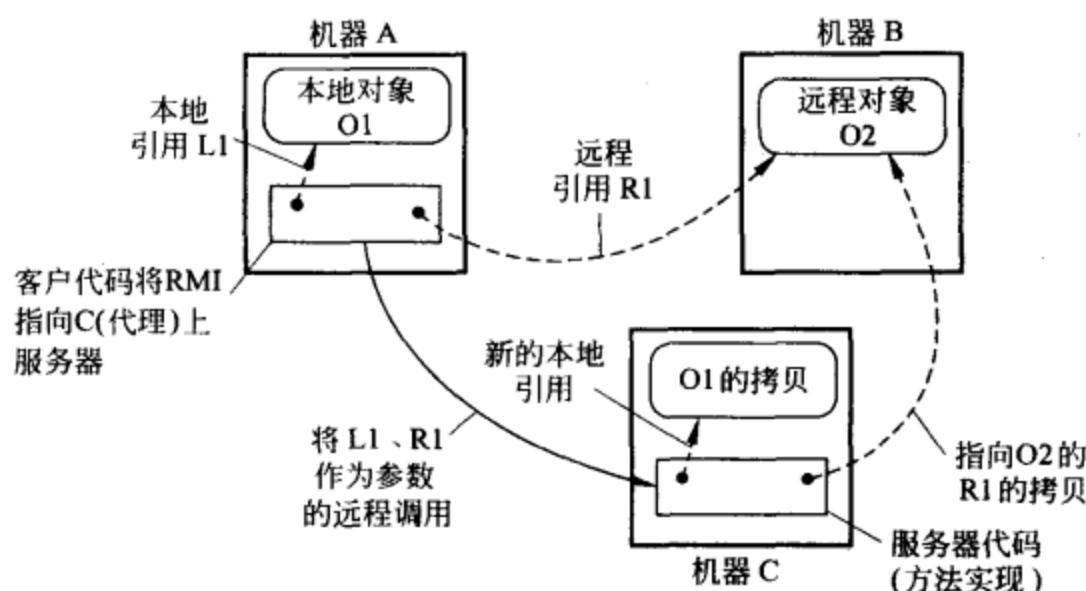


图 2.18 通过值来传递对象与通过引用传递对象

型与远程对象在本质上是不同的,而在其他方面,这两种引用几乎一样(Wollrath 等,1996)。但是,如果使用的是 C 语言这样的常规编程语言的话,指向本地对象的引用可以是简单的指针,而这种简单指针是不可能用来指向远程对象的。

将对象引用作为参数来进行方法调用的做法也会带来副作用,即可能要复制整个对象。因为无法对这种情况进行隐藏,所以我们被迫显式地区分本地对象和分布式对象。很明显,这种区分不但损害了分布透明性,而且导致编写分布式应用程序变得更加困难。

2.3.5 实例 1: DCE 远程对象

虽然 DCE 把分布式系统用在了正确的地方,但是有点时机不对。DCE 是第一个作为位于现有操作系统之上的中间件建立起来的分布式系统,它需要一个相对来说比较长的接受期来证明自己的价值。但不幸的是,还在接受期时它就由于远程对象的出现而受到了挑战,远程对象简直成了构建分布式系统的灵丹妙药。作为传统的基于 RPC 的系统,DCE 在确立合理的应用基础之前,就已经被认为过时了,因此有一段时期 DCE 举步维艰。实际情况是不使用对象无人理会,而 DCE 就没有使用对象。

很长一段时间以来,人们一直建议在 DCE 中引入对象技术,但是 DCE 研究人员总是争辩说他们已经在 DCE 中支持对象技术了。比如,他们声称 RPC 系统本质上就是基于对象的,因为其所有的实现和分布方面的特征都通过接口加以隐藏。然而,这种辩解并没有说服力,DCE 被迫更显式地采用对象技术。在本节中,我们将仔细探讨 DCE 是如何为分布式对象提供支持的。我们之所以对 DCE 对象感兴趣,是因为它们是对基于 RPC 的客户-服务器模型进行直接改进的产品,并且体现了从远程过程调用到远程方法调用的提升。

1. DCE 分布式对象模型

分布式对象已经以接口定义语言(IDL)的扩展形式和 C++ 语言绑定一起加入到 DCE 中。也就是说,DCE 中的分布式对象使用 IDL 的格式来说明,而用 C++ 语言实现

的。分布式对象表现为远程对象的形式,其实际的实现驻留在服务器上。服务器负责在本地创建 C++ 对象,并且将方法向远程客户所在的机器公开。除此之外不存在其他创建分布式对象的方法。

一共有两类分布式对象受到支持。其中分布式动态对象由服务器以客户的名义在本地创建,原则上只能由所代表的客户访问。客户如果要创建某个对象,必须先对服务器发出请求。因此,每一个动态对象所属的类都必须有一个相关的 create 过程,该过程可以通过标准 RPC 来调用。在动态对象创建以后,DCE 运行时系统将拥有对新对象的支配权,并且系统会将该动态对象与相应的客户关联起来。

与动态对象相反,分布式命名对象(distributed named objects)由服务器创建后不仅仅与一个客户相关联,而是供多个客户共用。命名对象通过目录服务中注册,这样客户可以先查找到该对象,然后再绑定到该对象上。注册完成后,得到的该对象的惟一标识符连同关于如何与该对象所在服务器联系的信息被一同存储起来。动态对象与命名对象之间的区别如图 2.19 所示。

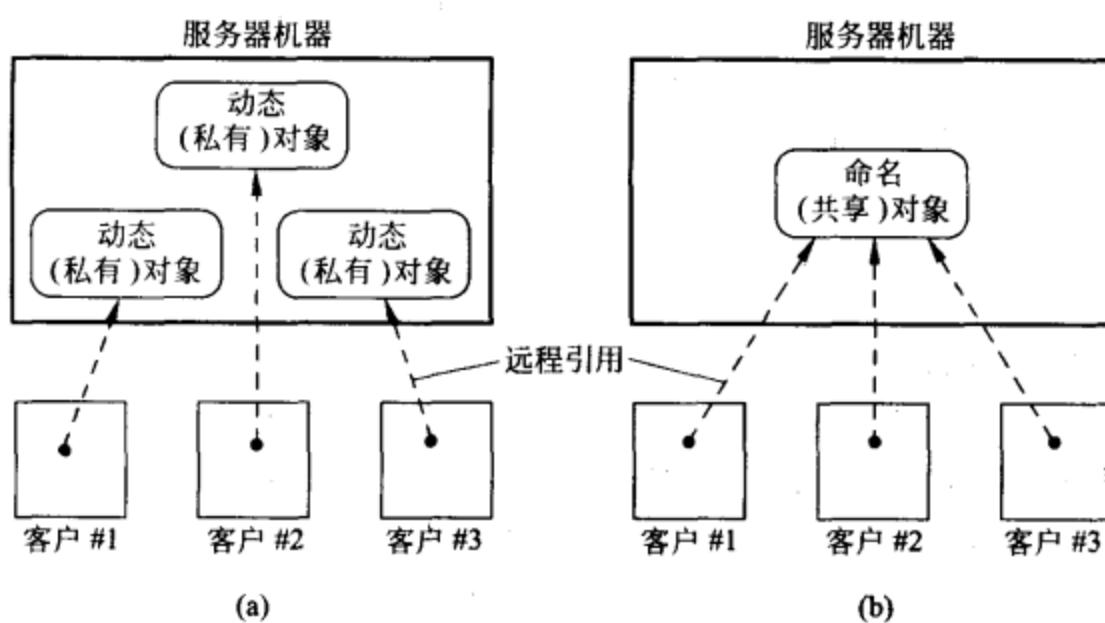


图 2.19 动态对象与命名对象

(a) DCE 中的分布式动态对象; (b) 分布式命名对象

2. DCE 远程对象调用

可以预料,DCE 中的每个远程对象调用都是通过 RPC 这种方式来进行的。当客户调用某个对象的方法的时候,它要传递对象的标识符、包含该方法的接口的标识符、方法本身的标号以及服务器需要的参数。服务器维护一个对象列表,在给定对象标识符以及接口标识符的情况下,从表中可以判断出应该调用哪一个对象。这样服务器就可以正确地使用参数来分派所请求的方法。

由于一个服务器有时要向上千个对象提供服务,因此 DCE 提供了将对象放置在辅助存储器上的能力,以代替将所有对象都保持在主存储器中的做法。如果输入的调用请求所调用的对象在服务器的对象列表中找不到,运行时系统就会调用针对特定服务器的查询函数来将对象由辅助存储器取出,并将其放入服务器的地址空间中去。在将对象放入

主存储器后,就可以对其中的方法进行调用了。

DCE 中的分布式对象存在一个问题,这个问题与其强烈的 RPC 背景有关:没有透明对象引用的机制。客户端顶多利用绑定句柄(binding handle)来与命名对象进行关联。绑定句柄中含有对象接口的标号、与对象所在服务器通信时所使用的传输协议以及服务器的主机地址和终点。绑定句柄可以转换成字符串,并且在不同进程间传递。

由于缺少合适的系统级对象引用机制,DCE 中的参数传递比其他许多基于对象的系统要更加困难。现在应用程序开发人员必须为 RPC 中的对象传递设计专门的解决方法,在实践中这就意味着必须将参数显式编组以进行值传递,从而需要开发针对特定对象的编组例程。

另一种方案是,开发人员可以使用委托(delegation)来传递参数。通过委托,可以根据对象的接口规范说明生成专门的存根。存根用来作为对实际对象的包装,其中仅包含有那些需要由远程进程调用的方法。随后必须将存根连接进需要使用相应用对象的进程中去。如果认识到 DCE 允许将指向存根的远程引用作为 RPC 中的参数来传递,这种方法的优点就一目了然了。因此,要通过指向存根的引用来在整个系统范围内访问对象是可以做到的。

关于 DCE 对象编程的详细信息可以在文献(Viveney 1998)以及文献(OSF 1997)中找到。

2.3.6 实例 2: Java RMI

在 DCE 中,添加进去的分布式对象本质上是远程过程调用的改进产物。客户标识的不是位于服务器上的远程过程,而是位于服务器的对象中的远程过程。它缺乏合适的系统级对象引用机制,这一点揭示了我们所面对的不过是 RPC 的改进产物。

让我们从一个完全不同的角度来审视分布式对象。在 Java 中,分布式对象已经集成进了语言本身中。这么做有一个重要的目的,那就是让尽可能多的非分布式对象的语义保持不变。换句话说,Java 语言开发人员的目标是取得高度的分布透明性。然而,就像我们即将看到的那样,Java 开发人员有时也不得不让分布变得可见,这种情况下要在实现高度的透明性会导致效率降低,透明性难以实现甚至不可能实现的时候。

1. Java 分布式对象模型

Java 也采用远程对象作为分布式对象的惟一形式。回忆一下,远程对象,这种分布式对象的特点是:它的状态一直驻留在单台机器上,而其接口可供远程进程使用。接口是通过代理以常规的方式实现的,代理提供了与远程对象完全相同的接口,而其本身呈现为位于客户地址空间中的本地对象。

在远程对象和本地对象之间还存在少量细微但却重要的差别。首先,对远程对象和本地对象的复制过程不同。如果要复制某个本地对象 O,会导致一个与 O 类型相同的新对象产生,而且新对象的状态与 O 也完全相同。复制过程返回与被复制的对象完全相同的一份拷贝。而这些语义难以应用到远程对象上去。如果要制作某个远程对象的一个完全相同的拷贝,不仅要复制位于服务器上的实际对象,而且还要复制当前绑定到该远程对

象上的每一个客户所使用的代理。因此对远程对象的复制操作只能由服务器执行,而且只是在服务器地址空间中制作与实际对象完全相同的拷贝,而不复制实际对象的代理。如果位于远程机器上的客户想要访问服务器上的对象拷贝,就必须首先重新绑定到该对象上。

Java 中本地对象与远程对象之间更重要的区别在于对象的阻塞这个语义上。如果某个对象中存在声明为 synchronized(同步)的方法,Java 就允许该对象作为监控器(monitor)来构建,如果两个进程同时调用声明为 synchronized 的方法,其中只会有一个进程继续执行,另一个将会被阻塞。这样就可以确保以完全串行的方式来访问对象的内部数据。在监控器中,也可以在对象内部阻塞一个进程,等到某些条件成立时再让它继续执行,这些内容已经在第 1 章中讨论过了。

从逻辑上说,在远程对象内部阻塞进程是很容易的。假定客户 A 调用远程对象的某个声明为 synchronized 的方法。因为 A 直接访问的是实现了对象接口的客户端存根,所以为了使对远端对象的访问看起来总是与对本地对象的访问完全相同,必须在客户端存根中阻塞 A。同样,对于位于另一台机器上的客户来说,必须在它的请求被发送给服务器之前将其在本地阻塞。这样做的结果是,需要对位于不同机器上的客户进行同步。与分布有关的同步是相当复杂的,我们将在第 5 章中对其进行讨论。

另一种方案是只在服务器上进行阻塞。原则上这是可行的,但是如果服务器正在处理客户的调用时,客户发生崩溃,就会出问题。就像我们将在第 7 章中看到的那样,要处理这种情况需要相对来说更为复杂的协议,而且这也会严重影响远程过程调用的综合性能。

因此,Java RMI 的设计者选择将对远程对象的阻塞限制在代理上(Wollrath 等 1996)。在实践中,这就意味着,通过使用声明为 synchronized 的方法无法保护远程对象免受并发访问的影响,因为访问可能来自于不同代理上的进程。此时必须使用显式的分布式锁定技术。

2. Java 远程对象调用

由于本地对象与远程对象之间的差异在语言层次上几乎不可见,Java 也可以隐藏远程方法调用中的大部分差异。比如说,任何原语或者对象类型只要可以编组,就都可以作为参数传递给 RMI。在 Java 术语中,可以编组意味着是可串行的(serializable)。虽然原则上大多数对象都可以串行,但是串行化并不是永远都是允许的。典型地说,与平台相关的对象,比如文件描述符和套接字就无法串行化。

RMI 中本地对象与远程对象的惟一差别是,本地对象(包括数组等大型对象)通过值传递,而远程对象通过引用传递。也就是说,先对本地对象进行复制,然后将得到的拷贝用作参数值。对于远程对象来说,传递的参数是指向该对象的引用,而不是该对象的拷贝,如图 2.18 所示。

Java RMI 中本质上实现了对远程对象的引用,这已经在 2.3.2 节介绍了。这种引用包含服务器的网络地址以及端点,还包含位于服务器地址空间中的实际对象的本地标识符。这种本地标识符只能由服务器使用。我们已经说过,对远程对象的引用中也需要加进客户端和服务器通信所使用的协议栈的编码。为了理解在 Java RMI 中是如何对协议

栈进行编码的,必须了解 Java 中的每个对象都是类的实例这样一个事实。类中包含有一个或者多个接口的实现。

从本质上说,远程对象是由两个不同的类创建的。其中一个类包含有服务器端代码的实现,称作服务器类(server class)。该类中包含远程对象在服务器上运行部分的实现,即包含对象状态的描述以及操纵这些状态的方法的实现。利用对象的接口规范说明可以生成服务器端存根,也就是骨架。

另一个类中包含有客户端代码的实现,称作客户类(client class)。该类中包含有代理的实现。与构架相同,客户类也是通过对象的接口规范说明生成的。最简单的代理所完成的惟一的任务就是:将每一个方法调用转换成消息,然后把消息发送给位于服务器端的远程对象的实现,将应答消息转换成方法调用的结果。对于每一次调用,代理都与服务器建立一个连接,并在调用完成之后拆除该连接。为了完成这些工作,代理需要了解服务器的网络地址和所使用的端点。这些信息以及位于服务器上对象的本地标识符都始终作为代理状态的一部分来存储。

因此,代理拥有让客户调用远程对象的方法所需的全部信息。在 Java 中,代理是可串行化的。也就是说,可以对代理进行编组,然后将它作为字节序列发送给另一个进程,再用解除编组后得到的代理来调用远程对象上的方法。即,代理可以用作对远程对象的引用。

这种做法与 Java 集成本地对象以及分布式对象的方式是一致的。不要忘记在 RMI 中,本地对象是通过制作拷贝来传递的,而远程对象是通过系统范围对象引用的方法来传递的。代理只是作为本地对象来看待。因此,可以将可串行的代理作为 RMI 参数来传递。这样做造成的另一种影响是,可以把代理作为对远程对象的引用来使用。

原则上,在对代理进行编组的时候,代理的完整实现(也就是它的所有状态和代码)都被转换成字节序列。这种代码编组方式的效率并不是很高,而且会导致庞大的引用。因此,在 Java 中对代理进行编组时,实际的做法是生成实现句柄,由实现句柄来准确指定构建代理所需要使用的类,其中某些类可能需要从远程端下载。实现句柄取代了作为远程对象引用一部分的已编组代码。在采用这种方法以后,Java 中的远程对象引用的大小一般为数百字节。

这种引用远程对象的方法灵活性很强,是 Java RMI 区别于其他 RMI 的特征之一(Waldo 1998)。特别是它允许采用针对特定对象的解决方法。比如说,考虑一个状态在一段短时间内只会变化一次的远程对象。我们可以将该对象的全部状态在绑定时复制给客户,这样该对象就变成了真正的分布式对象。每一次客户需要调用方法的时候,它都对本地拷贝进行引用。为了确保一致性,每一次调用时还要检查服务器的状态是否发生了变化,如果是的话就要对本地拷贝进行更新。同样,对需要修改状态的方法的调用将被转给服务器。这样远程对象的开发人员需要做的只是实现必需的客户端代码,并且让客户端在绑定到对象的时候下载该代码。

只有在所有进程都在同一个 Java 虚拟机上执行的情况下,也就是说所有进程都运行于相同执行环境中的情况下,才能将代理作为参数传递。编组后的代理在接收端被简单地解除编组,随后执行得到的代码。但情况并不总是这样,比如在 DCE 中,要传递存根是

根本不可能的,因为不同进程所在的执行环境在语言、操作系统和硬件等方面都不同。DCE 过程必须首先动态链接到本地的存根中,该存根是针对进程所在的执行环境编译的。只要将引用作为 RPC 参数传递给存根,就可以引用属于别的进程的对象。

2.4 面向消息的通信

远程过程调用和远程对象调用都有助于隐藏分布式系统中的通信,也就是说增强了访问透明性。不幸的是,这两种机制并不总是适用的。特别是当无法保证发出请求时接收端一定正在执行的情况下,就必须有其他的通信服务。同时,RPC 和 RMI 的同步特性也会造成客户在发出的请求得到处理之前被阻塞,因而有时也需要采取其他办法。

这里所说的“其他方法”就是消息传递机制。在本节中,我们重点讨论分布式系统中面向消息的通信。首先详细讨论同步方式的本质及其含义;随后讨论在通信过程中参与通信的各方都处于执行状态的消息传递系统;最后分析消息队列系统,即使通信的另一方在通信开始时并未执行,该系统也能允许进程互相交换信息。

2.4.1 通信中的持久性和同步性

为了更好地理解面向消息的通信的多种不同方式,我们假定通信系统的组织结构都采用如图 2.20 所示的计算机网络的形式。应用程序总是在主机上执行,每台主机都向通信系统提供接口,通过该接口可以提交待传输的消息。主机通过由通信服务器组成的网络相连,通信服务器负责在主机间传递消息(并进行路由)。为了不失一般性,可以假定每台主机都只连接到一个通信服务器上。在第 1 章中,我们假定只能在主机上放置缓冲区。在更为一般的情况下,需要考虑在底层网络的通信服务器上也可以放置缓冲区。

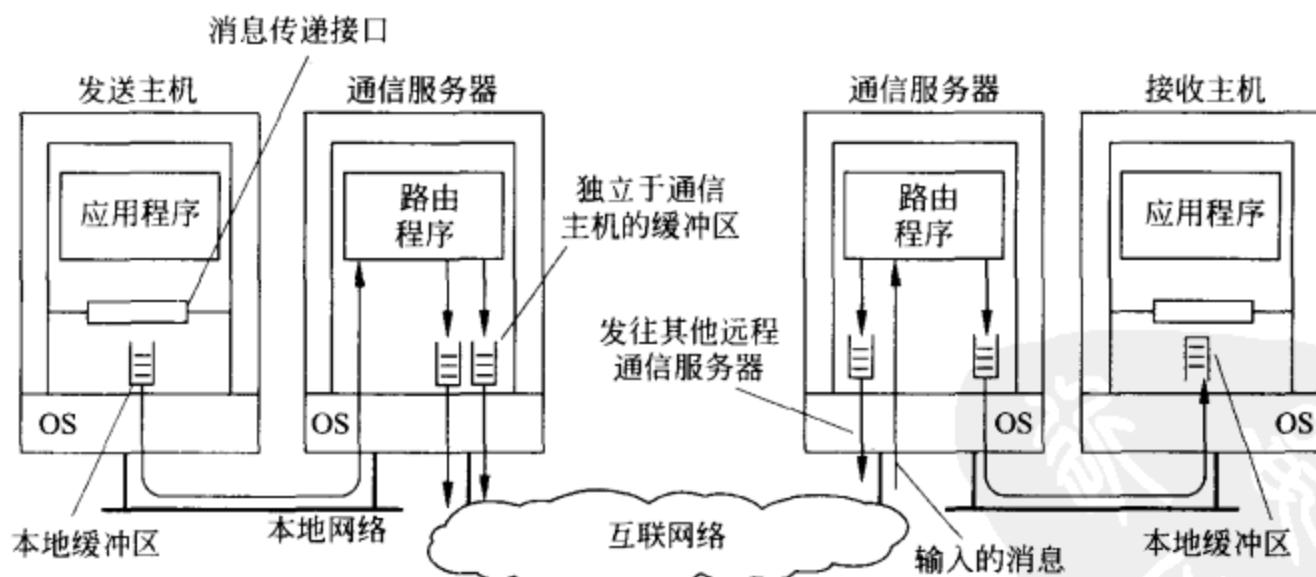


图 2.20 主机通过网络相连的通信系统的一般组织结构

作为例子,我们来看一个基于这种设计的电子邮件系统。用户代理在主机上运行,用户代理是供用户撰写、发送、接收及阅读消息的应用程序。每台主机只与一个邮件服务器相连接,邮件服务器相当于上面所说的通信服务器。用户代理可以利用用户主机上的接口来向特定的目的地发送消息。当用户代理将要传输的消息提交给主机以后,主机一般

会先将消息转发给本地邮件服务器,然后将消息暂时存储在本地邮件服务器的输出缓冲区中。

邮件服务器从输出缓冲区中移出消息,然后查询消息的目的地。对目的地所在地址的查询将会得到消息应当发送到的目标邮件服务器的(传输层)地址。邮件服务器随后与目标邮件服务器建立连接,并将消息传递给它。目标邮件服务器将消息存储在为指定的接收者准备的输入缓冲区中,该缓冲区也称作接收者的邮箱。如果暂时无法联系到目标邮件服务器,比如在目标邮件服务器已关闭的情况下,将继续由本地邮件服务器存储该消息。

接收主机上的接口向接收者用户代理提供服务,通过这种服务,接收者用户代理可以查看收到的邮件。用户代理可以直接在本地邮件服务器上的用户邮箱中对邮件进行处理,但在很多情况下它会将收到的新消息复制到所在主机的本地缓冲区中。也就是说,消息一般都在通信服务器上进行缓存,有时也会在接收主机上进行缓存。

电子邮件系统是持久通信(persistent communication)的典型例子。通过持久通信,需要传输的消息在提交之后由通信系统来存储,直到将其交付给接收者为止。在图 2.20 中,在将消息成功交付给下一个通信服务器之前,消息一直存储在通信服务器中。因此发送消息的应用程序不必在提交消息后保持运行。同样,要接收消息的应用程序在消息提交时也可以不处于运行状态。

持久通信的工作方式与以前经常使用的驿马快递制度相似,如图 2.21 所示。要投递一封信件,首先将它交给当地的邮局。邮局负责将信件根据下一站的不同进行分类,准备将信件传送给位于通往其投递目的地路线上的一站邮局。邮局同时也根据下一站的不同把信件保存在相应的邮包中,直到运送邮件的马和骑手到达。在下一站将对到达的信件重新进行分类,有些信件将被取走,其他的信件将由邮递员继续传递。重要的是,邮件从不会丢失或者被弃置在一旁。虽然运输邮件的手段以及邮件分类的手段在最近几百年中有了很大的进步,但是这种对邮件进行分类、存储以及转发的原理并没有发生变化。

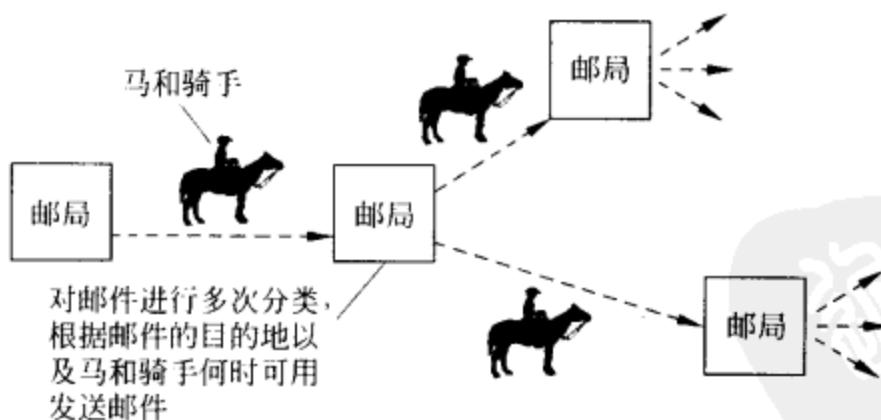


图 2.21 驿马快递时代使用的持久通信

与之相反,在暂时通信(transient communication)中,通信系统只在发送和接收消息的应用程序的运行期间存储消息。更准确地说,在如图 2.20 所示的情况下,如果通信服务器无法将消息递送到下一个服务器或者接收者,消息将会被简单地丢弃。所有典型的传输层通信服务都仅仅提供暂时通信,这种通信服务器与传统的存储转发式路由器相似。

如果路由器无法把消息发到下一个路由器或者目的主机,它将会简单地丢掉该消息。

通信除了分为持久通信和暂时通信之外,还可以分为同步通信(synchronous communication)和异步通信(asynchronous communication)。异步通信的典型特征在于发送者把要传输的消息提交之后立即继续执行其他程序,这意味着该消息存储在位于发送端主机的本地缓冲区中,或者存储在送达的第一个通信服务器上的缓冲区中。而对于同步通信来说,发送者在提交消息之后会被阻塞,直到消息已经到达并存储在接收主机的本地缓冲区中以后,也就是消息确实已经传送到接收者之后,才会继续执行其他程序。同步通信的最强的形式是,直到接收者处理完消息之后,才将发送者从阻塞中释放。

在实践中可以采用这些通信类型的多种组合形式。在持久异步通信中,消息可以持久地存储在本地主机的缓冲区中,或存储在送达的第一个通信服务器的缓冲区中。电子邮件系统一般采用这种通信方式。而如果是持久同步通信,消息就只能持久地存储在接收主机上。发送者在消息送达并存储在接收者的缓冲区中之前一直是阻塞的。要注意,并不是接收端应用程序一定要处于运行状态才能把消息存储到其所在的本地缓冲区中去。持久同步通信的一种较弱的形式是,在消息存储到与要接收该消息的主机相连的通信服务器上之后,就将发送者从阻塞中释放。

暂时异步通信一般由传输层数据报服务(例如 UDP)提供。当发送端应用程序提交消息以进行传输时,消息暂时存储在发送主机的本地缓冲区中,随后发送者立即继续执行其他程序。与此同时,通信系统将消息向目的地,也就是下一站发送,并要求下一站继续传递消息直到消息抵达接收者。如果在消息送达接收者所在主机的时候接收者并不在执行中,传输将失败。异步 RPC 是采用暂时异步通信的另一个例子。

暂时同步通信有多种不同形式。最弱的形式是基于消息接收的,即在消息送达接收主机并且存储在那里的本地缓冲区中之后就解除对发送者的阻塞,此时发送者会收到一个确认消息,于是继续执行其他程序。在基于交付的暂时同步通信中,在消息送达接收者以进行进一步处理之前,发送者则一直处于阻塞状态。在讨论异步 RPC 的时候,我们已经提到了这种同步方式。在异步 RPC 中,在客户的请求被接受以进行进一步处理之前,客户一直处于等待状态,因而客户与服务器是同步的。最强的形式是基于响应的暂时同步通信,即发送者在收到另一方发来的回应消息之前一直阻塞,这类似于客户-服务器交互中的请求-应答行为。RPC 和 RMI 遵循这种方案。

图 2.22 中总结了通信的持久性与同步性的各种不同的组合形式。文献(Tai 和 Rouvellou 2000)中讨论了另一种类似的分类方法。

直到最近,许多分布式系统依然只支持基于应答的暂时同步通信,采用的方法一般是远程过程调用或者远程对象调用。设计者认识到这种通信形式并不总是适用的,因此添加了一些较弱形式的暂时同步通信功能,例如异步 RPC 或者延迟的同步操作,如图 2.13 所示。

在消息传递系统中采用的则是一种完全不同的方法,这种方法以暂时异步通信为出发点,并可能在其中添加了同步通信的功能。然而,在使用消息传递的任何时候,都将通信假定为暂时的。也就是说,只提供那些适用于并发执行的进程的通信功能。但在很多情况下仅仅拥有符合这种要求的功能是不够的,特别是在考虑到地域可扩展性时。

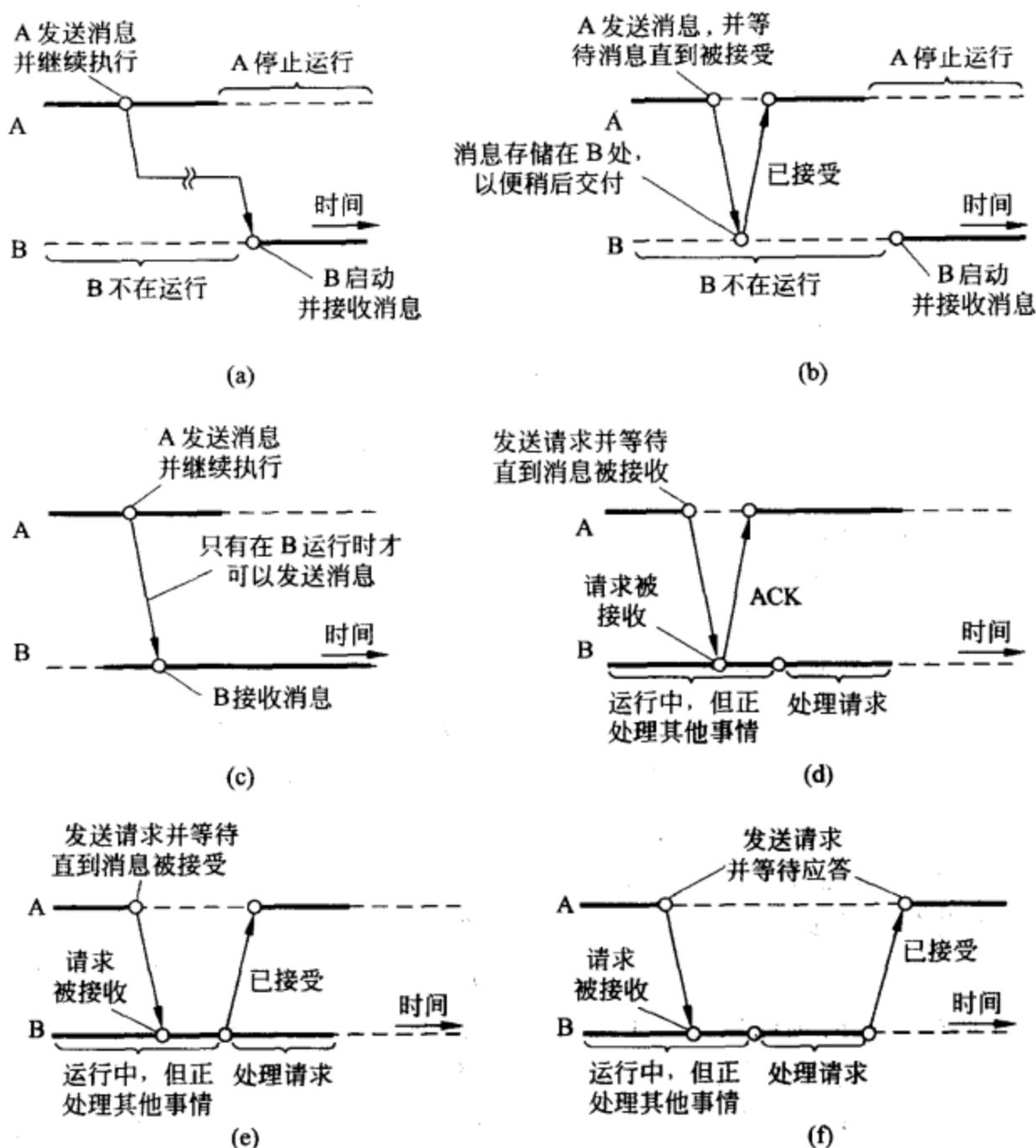


图 2.22 通信的 6 种不同形式

- (a) 持久异步通信；(b) 持久同步通信；(c) 暂时异步通信；
- (d) 基于接收的暂时同步通信；(e) 基于交付的暂时通信；(f) 基于响应的同步通信

当中间件开发人员需要将应用程序集成到大规模而且分散的互联网络中去的时候，就迫切需要持久通信服务。这种互联网络一般分布到不同的部门以及管理域，其中的某些部分可能有时会无法立即访问。例如，可能由于网络故障或者进程故障导致访问受到限制。为了解决这类问题，已经为持久通信开发了一些专用的解决方法。但是可以想象，这些解决方法在互用性和可移植性方面并不能令人满意。

暂时通信的另一个缺陷在于，一旦发生故障，必须立即将其屏蔽并启动恢复过程。不能拖延故障维修工作，因为这样会损害故障透明性。然而，对于持久通信来说，在开发应用程序时就设计好了如何处理发送请求到收到应答之间的长延迟时间，这样就可以选择采用更为简单但是反应更慢的故障屏蔽及恢复的解决方法。

有一点应该很清楚：在许多情况下，只有暂时通信或者只有持久通信都是不够的。

同样,只有同步通信功能或者只有异步通信功能也不行。所有类型的通信形式都是有用的,应该根据分布式系统的不同用途来加以取舍。到这里为止,我们主要讨论了 RPC 以及 RMI 使用的暂时同步通信。其他通信形式一般是由面向消息的通信系统提供的,我们将在下一节中对它们进行讨论。我们将分别讨论暂时通信和持久通信。

2.4.2 面向消息的暂时通信

许多分布式系统和应用程序直接构建在由传输层提供的简单的面向消息模型之上。面向消息的系统是中间件解决方案的一部分,为了更深入地理解和评价它,我们首先讨论通过传输层套接字进行的消息传递。

1. Berkeley 套接字

人们已经对传输层接口的标准化给予了充分的重视,以便使程序员通过一个简单的原语集就可以使用传输层提供的全部(消息传递)协议。同时,标准化的接口还使得不同机器之间的应用程序移植变得容易。

作为一个例子,我们将对 Berkeley UNIX 中引入的套接字接口(socket interface)进行简要讨论。另一种重要接口是 XTI,它是 X/open transport interface(X/开放传输接口)的缩写。它的正式名称是 TLI(transport layer interface,传输层接口),是由 AT&T 开发的。套接字与 XTI 采用类似的网络编程模型,但是它们的原语集有些差异。

从概念上说,套接字是一种通信端点。如果应用程序要通过底层网络发送某些数据,可以把这些数据写入套接字,然后再从套接字读出数据。对应于每一种特定的传输协议,本地操作系统都要使用一个实际的通信端点,而套接字形成了位于实际通信端点之上的一个抽象层。在下文中,我们将重点讨论用于 TCP 协议的套接字原语,如图 2.23 所示。

服务器一般执行图 2.23 中的前 4 个原语,而且一般按照图中给出的顺序执行。调用套接字原语的时候,调用者创建一个新的通信端点,该端点是用于某种特定的传输协议的。从内部来说,创建一个通信端点意味着本地操作系统储备一定的资源,以供特定协议在发送及接收消息时存放消息使用。

原语	含义
socket	创建新的通信端点
bind	将本地地址附加(attach)到套接字上
listen	宣布已准备好接受连接
accept	在收到连接请求之前阻塞调用方
connect	主动尝试确立连接
send	通过连接发送数据
receive	通过连接接收数据
close	释放连接

图 2.23 TCP/IP 套接字原语

bind 原语将本地地址与新创建的套接字相关联。例如,服务器应该将其所在机器的 IP 地址以及一个(可能是公开的)端口号绑定到套接字上。所执行的绑定操作将告知操

作系统,服务器希望只在指定的地址和端口上接收消息。

listen 原语只能在面向连接的通信中调用。这是一种非阻塞的调用,它允许本地的操作系统保留足够大小的缓冲区来存放用户希望接受的最大数目的连接。

对 accept 的调用将阻塞调用者,直到有连接请求到达为止。当请求到达的时候,本地操作系统将创建一个与原先的套接字属性相同的套接字,随后将新套接字返回给调用者。这种方法可以让服务器派生出一个新进程,由它来处理新连接的实际通信过程;在此期间服务器则返回到原来的状态,并等待原先套接字上新的连接请求。

我们现在来看一下客户端的情况。同样,首先必须使用 socket 原语创建一个套接字,但是不必将套接字显式绑定到本地地址上,这是因为操作系统可以在连接建立的时候为其动态分配端口。connect 原语要求其调用者指定一个传输层地址,以便将连接请求发送到该地址。调用 connect 后客户被阻塞,直到连接成功建立为止。随后双方使用 write 和 read 原语开始进行信息交换,其中 write 原语用来发送数据,而 read 原语则用来接收数据。最后要关闭连接。使用套接字来关闭连接是一个对称的过程,只有客户和服务器都调用了 close 原语,连接才真正关闭。客户和服务器通过套接字进行面向连接通信的一般模式如图 2.24 所示。在 UNIX 环境下使用套接字以及其他接口进行网络编程的详细资料可以在文献(Stevens 1998)中找到。

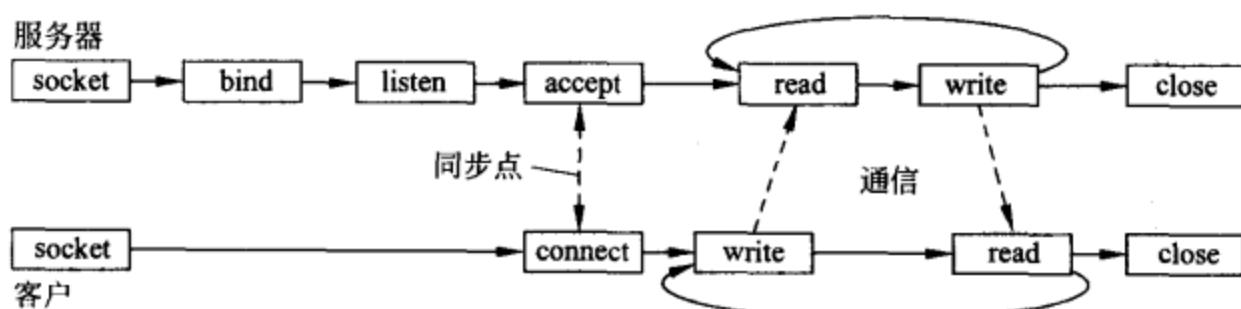


图 2.24 使用套接字的面向连接通信模式

2. 消息传递接口(MPI)

随着高性能多计算机系统的出现,开发人员开始寻求有助于更加方便地编写高效率应用程序的面向消息原语。这就要求原语应该提供一个位于适当层次上的抽象(以便简化应用程序开发),而且实现这些原语所花费的开销应该尽量小。由于两方面的原因,套接字被认为不能满足要求。首先,套接字所在的抽象层不对,它只支持简单的 send 以及 receive 原语。其次,套接字被设计为利用通用的协议栈(例如 TCP/IP)进行网络通信,而不适用于为高速互联网络开发的专用协议,比如那些用于 COW 和 MPP 中的协议(在 1.3 节中已对此进行了讨论)。这些协议要求接口能够提供一些更为高级的特性,比如不同的缓冲和同步方式。

因此,多数互联网络和高性能多计算机系统都附带有专用通信库。这些库提供了丰富的高层通信原语,而且这些原语通常是高效率的。当然,这些库互不兼容,导致应用程序开发人员需要面对程序的可移植性问题。

程序的硬件独立性的需要最终导致了消息传递方面标准的出台。该标准称作消息传递接口(message-passing interface, MPI)。MPI 是为并行应用程序设计的,因而是为暂时

通信量身定做的。它使用的是底层网络,不存在图 2.20 中的通信服务器那样的概念。同时,它还假定诸如进程崩溃或者网络分区之类的严重故障都是致命的,因此不要求这些故障能够自动恢复。

MPI 假定通信在一个已知进程组内发生。每个组都分配有一个标识符(ID),而组内的每个进程也分配有一个局部标识符。因此,一个(组 ID, 进程 ID)对就可以惟一地确定消息的来源或者目的地,可以用它来代替传输层地址。在一次计算过程中也许会涉及多个可能相互重叠的进程组,它们在相同时间执行。

MPI 的核心是一些消息传递原语,用于支持图 2.22 的(c)至(f)子图中大部分形式的暂时通信。最直观的一部分原语列在图 2.25 中。

原语	含义
MPI_bsend	将要送出的消息追加到本地发送缓冲区中
MPI_send	发送消息,并等待直到消息复制到本地或远程缓冲区中为止
MPI_ssend	发送消息,并等待直到对方开始接收为止
MPI_sendrecv	发送消息,并等待直到收到应答消息为止
MPI_isend	传递要送出的消息的引用,随后继续执行
MPI_issend	传递要送出的消息的引用,并等待直到对方开始接收为止
MPI_recv	接收消息,如果不存在等待接收的消息则阻塞
MPI_irecv	检查是否有输入的消息,但是无论有没有消息都不会阻塞

图 2.25 MPI 中一些最直观的消息传递原语

在本质上,只有图 2.22(d)所示的同步通信不受 MPI 支持。换句话说,MPI 不支持发送者与接收者在消息已经通过网络传输完毕这一点上进行同步。

暂时异步通信(如图 2.22(c))受 MPI_bsend 原语支持。发送者提交要传输的消息,该消息首先复制到 MPI 运行时系统的本地缓冲区中,消息复制完成后发送者继续执行。本地的 MPI 运行时系统将会把消息从本地缓冲区中移出,并在接收者调用了 receive 原语之后进行消息传输。

还有一个会导致阻塞的发送操作,称作 MPI_send,其语义是依赖于实现的。MPI_send 原语对调用方的阻塞有可能直到指定的消息被复制到发送者一端的 MPI 运行时系统为止,或者直到接收者启动接收操作为止。前一种情况与图 2.22(d)所示的异步通信相对应,而后一种情况与图 2.22(e)对应。

MPI_ssend 原语支持如图 2.22(e)所示的同步通信。在这种通信方式中,发送者将被阻塞,直到其请求被接收者接受并准备进行进一步处理为止。

最后,MPI 还支持如图 2.22(f)所示的同步通信的最强形式:当发送者调用 MPI_sendrecv 时,它在向接收者发出请求之后就被阻塞,直到接收者返回应答为止。该原语基本上相当于标准的 RPC。

MPI_send 和 MPI_ssend 各自拥有一种变体形式,使用这些变体时不需要将消息从用户缓冲区复制到本地 MPI 运行时系统的内部缓冲区中。这两种变体各自对应于一种异步通信形式。如果使用的是 MPI_isend,发送者会传递指向消息的指针,随后由 MPI 运行时系统来管理通信,而发送者则立即继续运行其他程序。为了避免在通信完成以前

就错误地覆盖消息, MPI 提供了用于检查通信是否完毕的原语, 甚至在必要时阻塞调用者。如果使用的是 MPI_send, 那么消息是确实传送给了接收者还是仅仅由本地 MPI 运行时系统复制到了内部缓冲区中, 它都不做说明。

同样, 使用 MPI_issend 的发送者只向 MPI 运行时系统传递指向消息的指针。在运行时系统告知发送者它已经处理完该消息之后, 发送者就能够确认接收者已经接受该消息并正在对它进行处理。

可以调用 MPI_recv 操作来接收消息, 这时, 调用者将被阻塞, 直到有消息送达为止。它也有一种采用异步方式的变体, 称作 MPI_irecv, 接收者调用 MPI_irecv 就说明它已经准备好接受消息。接收者可以检查是否确实有消息送达, 或者一直保持阻塞状态直到有消息送达为止。

MPI 通信原语的语义并不总是简明易懂的。不同的原语有时可以相互替换使用, 而不会影响程序的正确性。对于 MPI 为何支持如此之多的通信方式的官方解释是, 这可以赋予 MPI 系统的开发实现者充分的机会来进行性能优化。批评者认为是委员会无法统一大家的意见, 于是把 MPI 变成了包罗万象的大杂烩。MPI 是为高性能并行应用程序设计的, 了解了这一点就会明白 MPI 中的通信原语为何如此多样。

关于 MPI 的更多资料可以在文献(Gropp 等 1998b)中找到。而在文献(Snir 等 1998)和(Gropp 等 1998a)中有 MPI 的完整参考, 其中详细解释了 MPI 中的上百个函数。

2.4.3 面向消息的持久通信

现在要讨论一类重要的面向消息的中间件服务, 一般称为消息队列系统(message-queuing system), 或者面向消息的中间件(message-oriented middleware, MOM)。消息队列系统为持久异步通信提供多种支持。这类系统的本质是, 提供消息的中介存储能力, 这样就不需要消息发送者和接收者在消息传输过程中都保持激活状态。与 Berkeley 套接字及 MPI 的重要区别之一在于, 消息队列系统的设计目标一般是支持那些时间要求较为宽松的消息传输, 比如那些要求几分钟完成的传输, 而不适用于那些必须在几秒内甚至几微秒内完成的传输。我们首先解释消息队列系统的一般实现方法, 然后通过把它们与较为传统的系统(如 Internet 的电子邮件系统)相比较来结束本节的内容。

1. 消息队列模型

消息队列系统隐藏的基本思想是, 应用程序可以通过在特定队列中插入消息来进行通信。消息由一系列通信服务器依次进行转发, 最终送达目的地。即使在消息发送过程中接收者的机器未处于运行状态, 消息也能送到。在实践中, 多数通信服务器彼此直接相连, 也就是说消息一般直接传递到目的服务器。原则上, 每一个应用程序都拥有归其私有的消息队列, 其他应用程序可以发送消息到该队列中。队列只能由相应的(也就是拥有该队列的应用程序)读取, 但是也可能有多个应用程序共享单个队列。

消息队列系统的重要特征之一是, 通常只能确保发送者发出的消息最终能够插入到接收者的队列中, 并不保证消息到达的时间, 甚至不保证消息一定会得到读取, 这完全由接收者来决定。

这些语义允许使用松散耦合的通信方式。因此在消息进入接收者的消息队列时，接收者不必处于运行状态。同样，当发送者所发送的消息被接收者从消息队列中取出的时候，发送者也不必处于运行状态。发送者和接收者可以彼此完全独立地运行。事实上，某个消息一旦进入到队列中，就将一直保留到从队列中移出为止，而与其发送者或者接收者是否正在运行无关。根据发送者和接收者运行状态的不同，一共有 4 种组合，如图 2.26 所示。

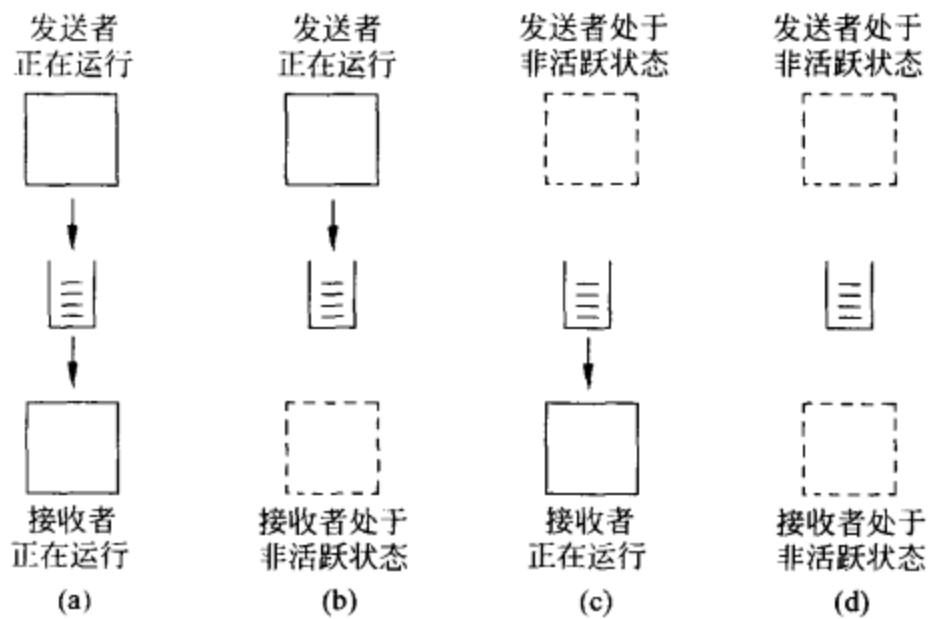


图 2.26 使用队列的松散耦合通信的 4 种组合方式

在图 2.26(a)中，发送者和接收者在消息的整个传输过程中都在运行。在图 2.26(b)中，只有发送者在运行，而接收者并不在运行，也就是说，是处于一种无法进行消息交付的状态，但是发送者仍然可以发送消息。图 2.26(c)中显示的是一种发送者没有运行而接收者在运行的状态。在这种情况下，接收者可以读出发送给它的消息。而在图 2.26(d)中的情况是，虽然发送者和接收者都不在运行，但是消息可以由系统存储(还可能进行传输)。

原则上说，消息中可以包含任何数据。惟一重要的是，应该对消息进行正确的编址。在实践中，编址是通过给出目的消息队列的名字来实现的，该名字在系统范围内是惟一的。在某些情况下，可能对消息的大小做出限制，不过底层系统也可能会以一种对应用程序完全透明的方式来对较大消息的分段及重组进行管理。采用这种方法将使得向应用程序提供的基本接口变得特别简单，如图 2.27 所示。

原语	含义
put	将消息追加进指定队列。
get	调用进程阻塞，直到指定队列变为非空为止，然后取出队列中第一个消息。
poll	查看指定队列中的消息，并且取出队列中的第一个消息。不阻塞调用进程。
notify	注册一个处理程序，在有消息进入指定队列时调用该处理程序。

图 2.27 消息队列系统中队列的基本接口

发送者调用 put 原语把要加入到某个指定队列中的消息传递给底层系统。我们曾经说过，这个调用是非阻塞性的。而 get 原语是一个阻塞性调用，调用 get 原语，获得授权

的进程将取出指定队列中等待最久的消息。只有在队列为空的时候，进程才被阻塞。这个调用还有几种变体形式，使用这些变体形式可以在队列中搜索给定的消息，比如可以根据优先级或者某种匹配模式来搜索。poll 原语是 get 原语的一种非阻塞性的变体形式。调用 poll 时，如果队列为空，或者找不到指定的消息，调用进程将简单地直接继续运行。

最后，多数消息队列系统也允许进程注册一个处理程序，称作回调函数(callback function)，在有消息进入队列时系统将自动调用该处理程序。回调还可以用于在没有进程运行的时候，自动启动一个进程以从队列中获取消息。这常常是通过在接收者一端驻留一个守护程序(daemon)来实现的，该程序对队列进行持续的监视，如果有消息进入就进行相应的处理。

2. 消息队列系统的一般体系结构

我们现在来详细讨论一般消息队列系统的外部特征。我们所做的第一项限制是，消息只能够放入发送者的本地队列，也就是与发送者位于同一台机器上的队列，至少也应该是附近的机器(比如同一个局域网中的机器)上的队列。这种队列称作源队列(source queue)。同样，只能从本地队列中读出消息。然而，放入队列的消息中将会包含对其将要传输到的目的队列(destination queue)的说明。由消息队列系统负责向发送者和接收者提供队列，并对消息由源队列向目的队列传输过程进行管理。

意识到下面这点很重要：全部队列的集合是分布在多台机器上的。因此，对于要传输消息的消息队列系统来说，它应该维护一个由队列到其所在网络位置之间的映射关系，这在实践中也就意味着它应该维护一个(也许是分布式的)数据库，其中存储网络位置所对应的队列名(queue name)，如图 2.28 所示。应该注意到，这种映射关系与在 Internet 上用来传输电子邮件的域名解析系统(domain name system, DNS)是完全类似的。比如说，如果向逻辑邮件地址 steen@cs.vu.nl 发送邮件，邮件系统将会请求 DNS 找到接收者的邮件服务器所在的网络地址(也就是 IP 地址)，服务器用该地址来进行实际的消息传输。

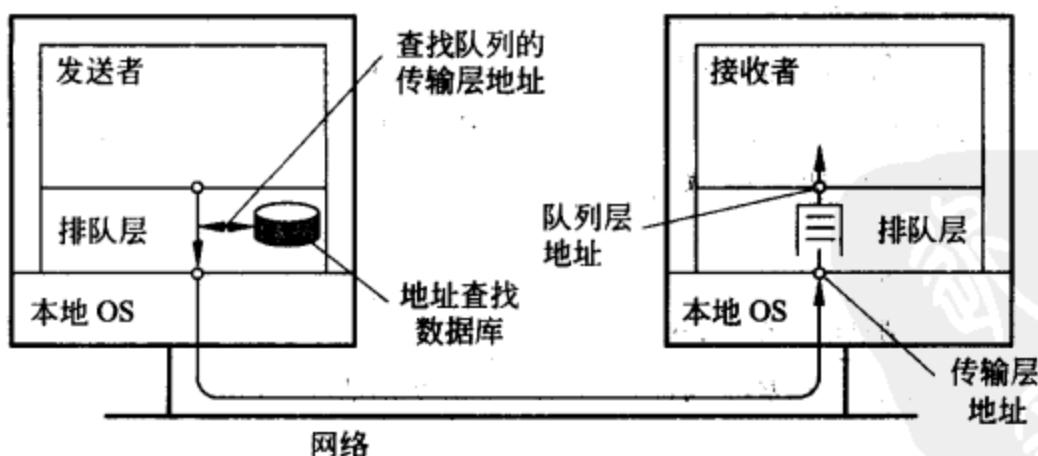


图 2.28 队列级编址与网络级编址之间的关系

队列由队列管理器(queue managers)来管理。一般说来，队列管理器与发送或者接收消息的应用程序直接交互。然而，也有一些特殊的队列管理器发挥了路由器或者中继器的作用：它们将输入的消息转发给其他的队列管理器。通过这种方式，消息队列系统

可以逐渐演变成为完整的应用级覆盖网络(overlay network),该覆盖网络位于原来的计算机网络之上。这种方案与 Internet 早期的 MBone 架构方案相似,在 MBone 中把普通用户进程都设定为多播路由器。如今,有许多路由器都已经可以支持多播,这就使得覆盖多播变得不那么重要了。

由于多种原因,使用中继器可以带来方便性。比如说,在很多消息队列系统中,没有能对队列-位置映射进行动态维护的通用命名服务。而如果队列网络的拓扑是静态的,每个队列管理器就都需要一份队列-位置映射的拷贝。不用说,在大规模队列系统中,这种方案很容易导致网络管理方面的问题。

一种解决方案是,使用若干了解网络拓扑的路由器。如果说,发送者 A 将目的地为 B 的消息放入其本地队列中,该消息将会首先传输到最邻近的路由器,在图 2.29 中是 R1。在这里,路由器 R1 知道如何将该消息往 B 的方向转发。比如说,R1 可以从 B 的名字推断出应该将消息转发给路由器 R2。采用这种方式,只有在路由器需要添加队列或者删除队列时进行更新操作,而其他队列管理器都只需要知道最邻近的路由器所在的位置即可。

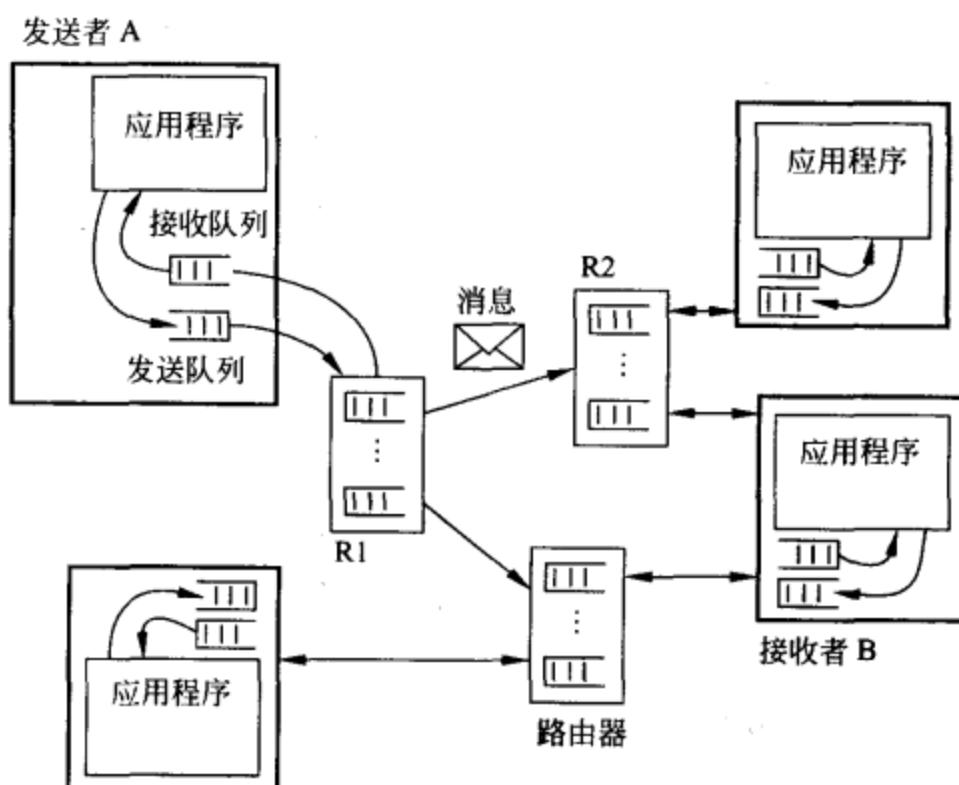


图 2.29 含有路由器的消息队列系统的一般组织结构

中继器一般有助于构建可扩展的消息队列系统。然而,显而易见的是,随着队列网络规模的迅速增长,如果还用人工来配置网络,网络很快就会变得难以管理。惟一的解决方法是采用动态路由,这种方法在计算机网络中早已采用了。令人惊讶的是,某些流行的消息队列系统并没有采用这种解决方法。

使用中继器的另一个原因是,中继器允许对消息进行二次处理。比如说,基于安全方面或者容错的考虑,可能要在日志中记录消息。我们将在下一节中讨论中继器的一种特殊形式,它发挥了网关的作用,将消息转换成接收者能够理解的格式。

最后,中继器还可以用于多播的目的。在这种情况下,输入的消息被简单地放入每一个发送队列。

3. 消息转换器

消息队列系统的重要应用领域之一就是将现有的应用程序与新应用程序一同集成进一个单一的、结构紧密的分布式信息系统中去。要进行集成,就要求应用程序必须能够理解所接收到的消息,这实际上也就是要求发送者送出的消息所用的格式必须与接收者的消息格式相同。

这种方法存在的问题是,每一次在系统中添加需要不同消息格式的应用程序时,每个可能的接收者都必须进行改动,以便能够识别新的格式。

另一种方法是,各方法达成一致,都采用一种公用的消息格式,就像传统网络协议那样。不幸的是,这种方法在消息队列系统中一般并不适用。问题在系统运行所在的抽象层次。公用消息格式只在使用该格式的所有进程确实有足够的共同之处的情况下才有意义。如果组成分布式信息系统的应用程序间的差异非常大(这是常有的情况),那么即使最好的公用格式也只不过意味着一串字节序列。

虽然规定了少数供特定的应用程序域使用的公用消息格式,但是一般的方法还是倾向于提供几种共存的不同格式,并提供在各种格式之间尽可能简单地转换手段。在消息队列系统中,转换是由队列网络中特定结点完成的,这些结点称作消息转换器(message broker)。消息转换器在消息队列系统中扮演了应用层网关的角色,其主要目的在于将输入消息的格式转换为目的应用程序能够理解的格式。要注意,对于消息队列系统来说,消息转换器也只不过是一个应用程序而已,如图 2.30 所示。换句话说,一般不认为消息转换器是队列系统的一个主要部分。

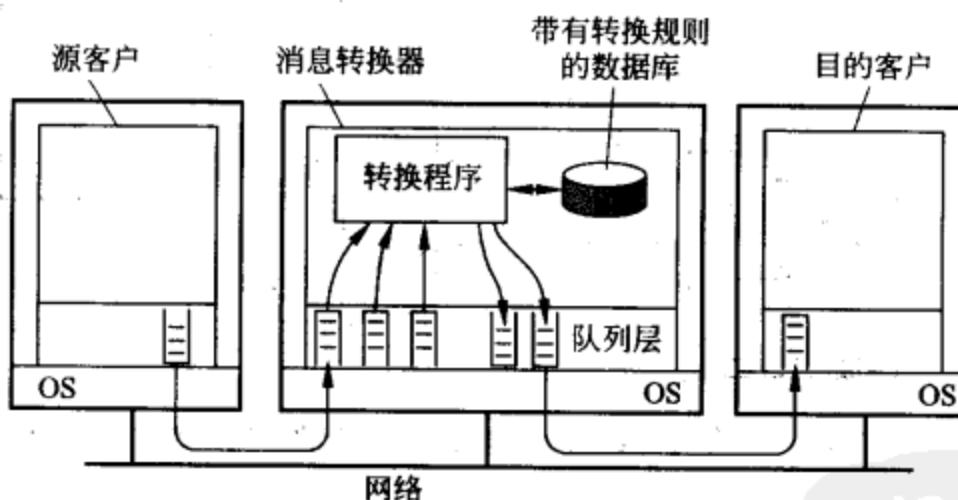


图 2.30 消息队列系统中消息转换器的一般结构

消息转换器可以像一个消息格式重新编排工具一样简单。比如说,假定输入的消息包含有数据库中的一张表,其中,记录之间通过特殊的代表记录结束的分隔符来分隔,而记录中的字段长度已知且固定。如果目的应用程序希望使用另外的分隔符来分隔记录,并且希望字段是可变长度的,就需要使用消息转换器来将消息的格式转换为目的应用程序所能够接受的格式。

拥有更高级设置的消息转换器可以扮演应用层网关的角色;比如处理 X.400 消息格式与 Internet 电子邮件格式之间的转换。在这种情况下,常常不能够保证输入消息中包

含的所有信息都能够实际转换为输出消息中相对应的内容。也就是说,可能不可避免地要在转换过程中损失某些信息。要了解这方面的更多内容,请参阅文献(Houttuin 1993)和(Alvestrand 1998)。

消息转换器的核心是一个数据库,数据库中的规则规定了如何将格式为 X 的消息转换为格式为 Y 的消息。问题在于如何定义这些规则。多数消息转换器的产品附带有复杂的规则开发工具,但是仍然要求手工将规则输入数据库。虽然可以用专门的转换语言来描述规则,但是许多消息转换器产品也允许使用常规编程语言来编写转换规则。因此,建立消息转换器是一项非常艰苦的工作。

4. 消息队列系统说明

请回顾一下我们谈论过的关于消息队列系统的内容,看起来好像这种系统早就以电子邮件服务的方式实现了。电子邮件系统一般是通过一组邮件服务器实现的,这些服务器代表与其直接相连的主机上的用户来存储并转发消息。电子邮件可以直接使用底层传送服务,而并不涉及路由问题。比如说,在 Internet 主要协议之一——SMTP(Postel 1982)中,可以直接与目的邮件服务器之间建立 TCP 连接来传输消息。

电子邮件系统与消息队列系统相比较,特别之处在于前者的目的主要在于为最终用户提供直接支持。这就说明了为什么有多种群件应用程序是直接基于电子邮件系统的(Khoshafian 和 Buckiewicz 1995)。另外,电子邮件系统可能有一些特殊的需求,比如自动消息过滤、高级消息数据库支持(比如,为了更方便地取得先前存储的数据的需要)等。

普通消息队列系统的目标并不仅仅在于为最终用户提供支持。建立这样的系统要解决的重要问题是:要为进程间的持久通信提供支持,无论该进程是在运行用户程序,在处理对数据库的访问,还是在进行计算等,都必须能够支持其通信。这就导致消息队列系统的需求与纯粹的电子邮件系统不同。比如说,电子邮件系统一般不需要提供可靠的(有保证的)消息传递、消息的优先级、日志功能、高效的多播、负载平衡、容错性,等等。

因此,通用的消息队列系统的应用范围很广,包括电子邮件、工作流、组件以及批处理。然而,它最重要的应用领域是将一组数据库(也许分散在广阔区域)或者数据库应用程序集成进多个数据库(信息)系统中去。如果想了解这方面的更多资料,请参阅文献(Oszu 和 Valduriez 1999)以及(Sheth 和 Larson 1990)。比如说,一个涉及若干个数据库的查询可能需要切分成为几个子查询,然后将这些子查询转发给相应的数据库。在这个过程中可以利用消息队列系统所提供的基本方法来将每个子查询打包进消息中去,然后将消息发送给相应的数据库。在本章中讨论过的其他通信工具在这方面都远没有这么适用。

2.4.4 示例: IBM MQSeries

为了帮助理解消息队列系统的实际工作方式,我们来研究一个实际的系统——IBM 的 MQSeries 系统(Oilman 和 Schreiber 1996)。这种系统在 IBM 主机的传统领域渐渐流行起来,IBM 主机主要是用来访问和操纵大规模数据库的。金融领域是 MQSeries 的重要应用领域之一。

1. 概述

MQSeries 的基础体系结构是非常简单易懂的,如图 2.31 所示。所有队列由队列管理器(queue managers)管理。队列管理器负责从其发送队列中取出消息,然后将消息转发给其他队列管理器。同时,队列管理器还负责从底层网络获取输入的消息并进行处理,随后将每个消息存储进恰当的输入队列中。

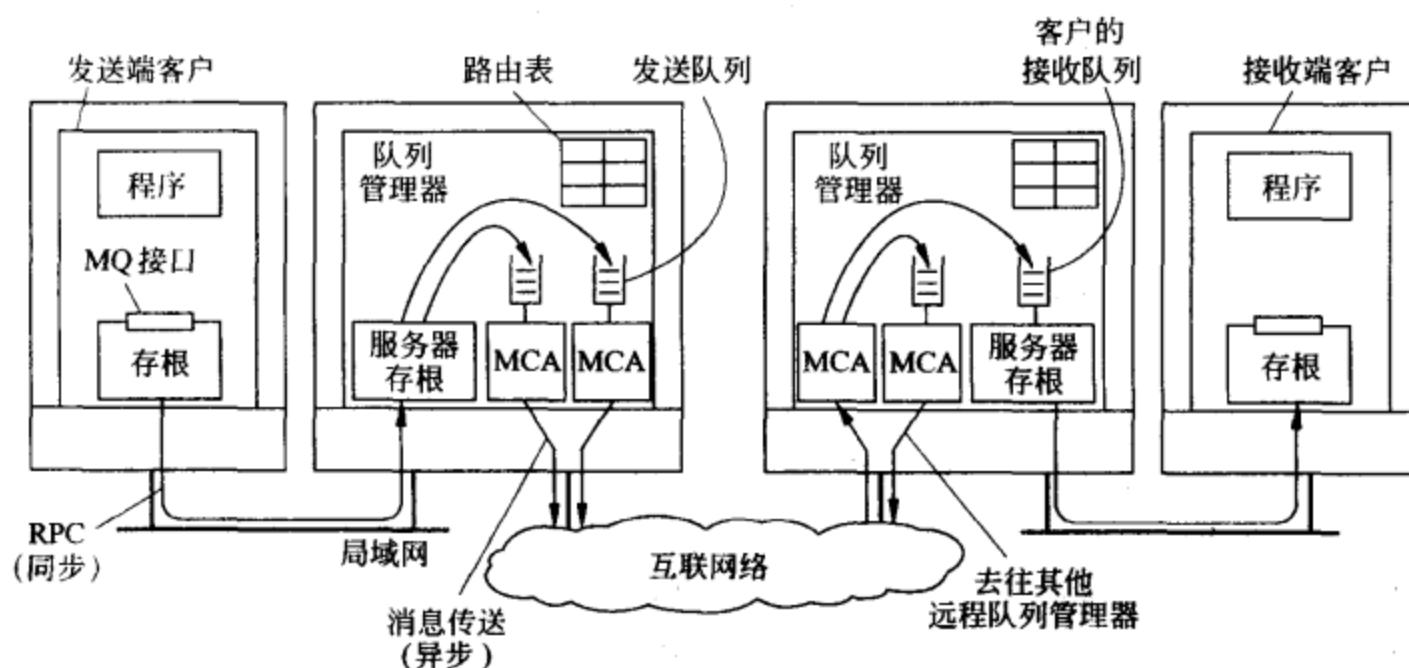


图 2.31 IBM MQSeries 消息队列系统的一般组织结构

一对队列管理器通过消息通道(message channel)相连,消息通道是传输层连接的抽象。消息通道是在发送者队列管理器和接收者队列管理器之间单向的可靠连接,队列中的消息通过该通道传输。比如说,基于 Internet 的消息通道是作为 TCP 连接实现的。消息通道的两端各自由一个 MCA(message channel agent,消息通道代理)来管理。发送端 MCA 所做的基本上只是:检查发送队列中是否有消息,如果有的话将消息包装成传输层的包,然后将包通过连接发送给对应的接收端 MCA。同样,接收端 MCA 的基本任务只是监听是否有包送达,如果有的话将包解开,随后将由包中得到的消息存储进恰当的队列中。

队列管理器可以连接到其管理的队列所属的应用程序所在的进程中去。在这种情况下,队列通过标准接口对应用程序隐藏,但是应用程序可以有效地对队列进行操纵。另一种组织结构是,队列管理器与应用程序运行在不同的机器上。在这种情况下,向应用程序提供的接口与第一种情况——也就是应用程序与队列管理器位于同一台机器上时相同。然而,不同的是这时接口是作为代理实现的,它使用基于 RPC 的传统同步通信来与队列管理器通信。通过这种方式,MQSeries 基本保留了这样的模型,就是只能够访问应用程序的本地队列。

2. 通道

消息通道构成了 MQSeries 的一个重要部分。每个消息通道都拥有且只拥有一个与

其相对应的发送队列,它从中取出消息,传输给另一端。只有在通道两端的 MCA——发送端 MCA 和接收端 MCA 都已启动运行的情况下,消息才能沿通道进行传输。除了可以手工启动两端的 MCA 之外,还可以采取某些别的办法来激活一个通道,下面我们将讨论其中的几种方法。

一种方法是,由应用程序通过激活发送端或者接收端 MCA 来直接启动通道的一端。然而,如果考虑到透明性,这种方法并不十分可取。启动发送端 MCA 的更好的办法是,对通道的发送队列进行配置,使其在第一个消息放入队列时触发一个触发器(trigger),该触发器随之启动与其相关联的一个处理程序,该处理程序负责启动发送端 MCA,随后就可以由 MCA 从发送队列中取出消息。

还有一种方法是通过网络来启动 MCA。特别是,如果通道一端已经激活,它就可以发送一条控制消息,请求另一端的 MCA 启动。该控制消息是发送给与要启动的另一端 MCA 位于同一台机器上的守护程序的,该守护程序监听某个公开的地址。

如果不再有消息送入发送队列,经过一段时间后通道将会自动关闭。

每一个 MCA 都有一组相关的属性,这些属性决定了通道的全部特性。其中某些属性如图 2.32 所示。发送端 MCA 与接收端 MCA 的属性值应该是一致的,并且应该在建立通道之前就协商好。比如说,两端的 MCA 都应该明确地支持同一传输协议。有的属性值无法进行协商,比如消息是否应该按照放入发送队列的次序来进行传输就是这样的属性。也就是说,如果一端的 MCA 要以 FIFO(先入先出)的次序来传输,另一端就必须遵守这个次序。而如果属性值包含了消息最大长度,则可以采用两端 MCA 对应的该属性值中较小的那个。

属性	描述
transport type(传输类型)	决定要采用的传输协议
fifo delivery(先入先出传输)	表明消息将按照发送的次序到达
message length(消息长度)	单个消息的最大长度
setup retry count(建立连接的重试次数)	指定启动远程 MCA 的最大重试次数
delivery retries(消息交付重试次数)	MCA 将收到的消息放入队列的最大重试次数

图 2.32 某些与消息通道代理相关的属性

3. 消息传输

为了从一个队列管理器向另一个队列管理器传输消息,每一个消息中都必须包含它的目的地址,也就是说必须使用传输报头。MQSeries 中的地址由两部分构成:第一个部分包含将要接收消息的队列管理器的名字,而第二部分是目的队列的名字,队列管理器将把消息添加到该队列中去。

除了目的地址之外,还必须指定消息传输的路由。路由的说明是通过给出其本地发送队列的名字来实现的,消息将被添加到该队列中。因此,不必在消息中包含完整的路由。回忆一下,每个消息通道都拥有且只拥有一个发送队列。通过指定消息应添加到哪一个发送队列中,就可以有效地规定应该将消息转发到哪一个邻近的队列管理器。

多数情况下,路由被显式地存储在队列管理器中的路由表中。路由表中的条目是(destQM, sendQ)对,其中 destQM 是目的队列管理器的名字,而 sendQ 是本地发送队列的名字,队列管理器将把消息放入该队列中去(路由表中的条目在 MQSeries 中称为别名)。

消息在传输过程中可能需要经过多个队列管理器才能到达目的地。当消息被中途遇到的队列管理器收到时,该队列管理器简单地从消息报头中提取出目的队列管理器的名字,随后在路由表中进行查询,以决定要将该消息放入哪一个本地发送队列中去。

认识到下面这点是很重要的:队列管理器拥有一个系统范围内惟一的名字,该名字可以有效地用作队列管理器的标识符。但是这样做带来的问题是:替换队列管理器或者更改队列管理器的名字会对向该管理器发送消息的所有应用程序造成影响。这个问题可以通过为队列管理器取一个本地别名(local alias)而部分地得到解决。如果由队列管理器 M1 为队列管理器 M2 定义了一个别名,那么只有使用 M1 接口的应用程序可以使用该别名。即使管理某个队列的队列管理器发生了改变,还是可以使用该队列原来的逻辑名来访问该队列。如果队列管理器的名字改变了,就需要改变它在其他所有队列管理器中的名字,但不会对应用程序造成影响。

路由表以及别名的使用规则如图 2.33 所示。比如说,连接到队列管理器 QMA 的应用程序可以使用本地别名 LA1 来引用远程队列管理器。QMA 将会首先在别名列表中查询消息实际的目的地,发现该别名代表的是队列管理器 QMC。随后,在路由表中找出到 QMC 的路由,其中指明了要发往 QMC 的消息应该放入队列 SQ1 中,而 SQ1 是用来向队列管理器 QMB 传输消息的。在收到该消息后,QMB 将会利用它自己的路由表将消息转发到 QMC。

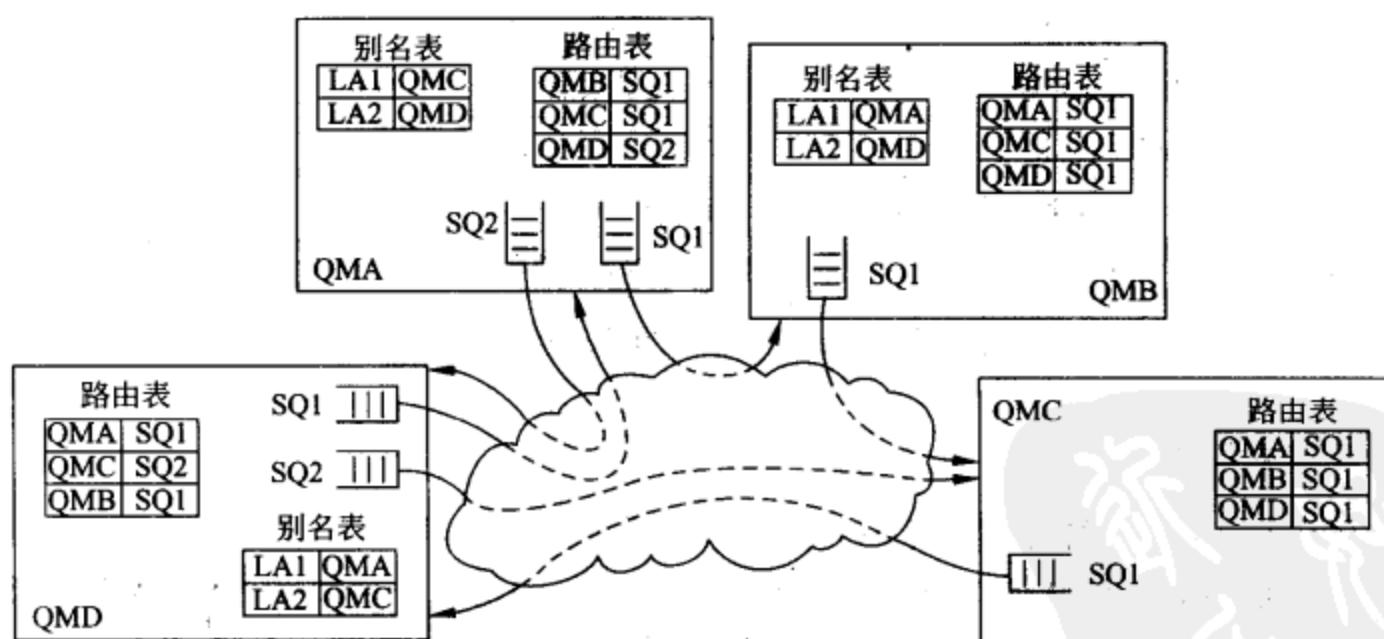


图 2.33 MQSeries 队列网络的一般组织结构——使用了路由表和别名

如果采用这种路由和别名方案,就会使得应用程序编程接口相对简单,这时的接口称作消息队列接口(message queue interface, MQI)。MQI 中最重要的原语如图 2.34 所示。

原语	描述
MQopen	开启一个队列(可能位于远程)
MQclose	关闭一个队列
Mqput	将消息放入开启的队列中
Mqget	从队列(队列可以在本地)中获取消息

图 2.34 IBM MQSeries MQI 中的部分原语

如果要将消息放入队列中,应用程序就需要调用 MQopen 原语,同时指定特定队列管理器中的某个目的队列。队列管理器可以使用可用的本地别名来命名。目的队列是否位于远程的问题对应用程序是完全透明的。如果应用程序要从本地队列中取出消息,也需要调用 MQopen。只能够打开本地的队列来读取送达的消息。当应用程序结束对队列的访问之后,需要调用 MQclose 来关闭该队列。

可以调用 MQput 向队列中写入消息,调用 MQget 来从队列中读出消息。理论上说,是根据消息优先级来决定将哪一条消息移出队列的。如果消息优先级相同,就按照先入先出的原则来决定移出哪一条消息,等待时间最长的消息将首先被移出。也可以要求取出某一条特定消息。最后,MQSeries 提供了在特定消息送达时通知应用程序的功能,这就避免了应用程序为了等待接收某个消息而不断对消息队列进行查询。

2.5 面向流的通信

到目前为止,我们讨论的通信都是对某种程度上独立且完整的信息单元进行交换。比如,调用某个过程或者方法的请求、对这种请求的响应,以及消息队列系统中应用程序间的消息交换都属于这种类型的通信。这种类型通信的典型特征是,它并不在乎通信究竟在哪个确切时间发生。虽然系统可能会运行得过快或者过慢,但是同步对通信的正确性没有影响。

但是,在某些形式的通信中,同步扮演了关键的角色。考虑一下这种情况:比如一个采用 16 位样本序列表示的音频流,每一个样本通过脉冲码调制(pulse code modulation, PCM)来表示声波的振幅。假设该音频流需要达到 CD 音质,也就是说应该对原始声波进行频率为 44 100Hz 的采样。为了复现原声,关键在于音频流中的样本必须按照次序来播放,而且播放的每两个样本之间的时间间隔必须为严格的 1/44 100s。如果以别的速率来播放,就会造成播放的声音与原声不同。

我们在本节中要讨论的问题是,分布式系统应该提供怎样的功能,来为时间敏感的信息交换(比如音频流和视频流)提供支持。文献(Halsall 2001)中讨论了用来处理面向流通信的协议。而文献(Steinmetz 和 Nahrstedt 1995)对多媒体方面内容进行了综合介绍,面向流的通信是其中的一部分。

2.5.1 为连续媒体提供支持

为时间敏感的信息交换提供支持一般表示为连续媒体提供支持。在这里,媒体是指

传送信息的手段,其中包括存储以及传输介质、如监视器之类的显示媒介,等等。媒体的重要类型是信息的表示方式,也就是信息在计算机系统中的编码方式。对于不同类型的信息要使用不同的表现形式。比如说,文本一般用 ASCII 或者 Unicode 来进行编码。图像可以采用 GIF 或者 JPEG 等格式来表示。而计算机系统中的音频流可以采用诸如 PCM 的 16 位样本来编码。

在连续(表示)媒体中,了解不同数据项之间在时间上的联系是正确解释数据含义的基础。我们已经给出过这样的例子:通过播放某个音频流以复现原来的声音。作为另一个例子,我们来考虑一下运动。运动可以通过一系列图像来表现,这些图像必须以相同的间隔时间 T 来连续显示, T 一般为 $30\sim40\text{ms}$ 。如果要正确地表现运动过程,不但要以正确的顺序显示图像,还要以恒定的显示速率——每秒 $1/T$ 幅图像来显示。

与连续媒体相反,离散(表示)媒体的特征是:数据项之间的时间联系对于正确解释数据含义并不重要。离散媒体的典型例子包括文本、静态图像的表示,还有对象代码及可执行文件等。

数据流

为了对时间敏感的信息进行交换,分布式系统一般都提供对数据流(data stream)的支持。数据流是数据单元的序列,可以应用于离散的媒体,也可以应用于连续媒体。比如说,UNIX 中的管道或者 TCP/IP 连接就是面向字节的离散数据流的例子。而播放音频文件时一般要求在文件与音频设备之间建立连续数据流。

对于连续数据流来说,同步是极为关键的。为了捕捉同步状况,要对不同的传输模式做出区分。在异步传输模式(asynchronous transmission mode)下,流中的数据项是逐个传输的,但是对某一项在何时进行传输并没有进一步的限制。这是采用离散数据流时常见的。比如说,文件可以作为数据流来传输,但是每一数据项传输完成的确切时间通常是无关紧要的。

在同步传输模式(synchronous transmission mode)下,数据流中每一个单元都定义了一个端到端最大延迟时间。数据单元的传输时间是否远远小于最大允许延迟并不重要。比如说,传感器以某个特定的采样率对温度值进行采样,然后将采样结果通过网络传给操作员。在这种情况下,很重要的一点是必须确保端到端的网络传输时间小于采样间隔时间,但是如果采样结果的传输速度大大超过所需的最低值是没有关系的。

最后,还有一种等时传输模式(isochronous transmission mode),在这种模式中数据单元必须按时传输,也就是数据传输的端到端延迟时间必须同时受到上限和下限的约束,端到端延迟时间上限和下限也称为边界延迟抖动。等时传输模式在表现视频和音频方面扮演了关键角色,因此它对分布式多媒体系统极为有用。在本章中,我们只考虑使用等时传输的连续数据流,简称为流。

流可以很简单,也可以很复杂。简单流(simple stream)只包含有单个数据序列,而复杂流(complex stream)由若干相关的简单流——子流(substreams)构成。复杂流中各子流之间的关系常常是时间敏感的。比如说,立体声音频可以使用包含两个子流的复杂流来传输,其中每个子流代表一个声道。然而,这两个子流必须是始终保持同步的,这一点

很重要。换句话说,来自两个流的数据单元都必须成对传输,以确保产生立体声的效果。另一个复杂流的例子是关于影片传输的,这种流应该包含单个视频流,两个用于以立体声方式传输电影中声音的音频流,还有第4个流用来传输那些不在对白中出现或者是不同语言的字幕。在这里各子流间的同步依然是重要的。如果无法做到同步,就无法完整地复现影片。我们将在稍后继续讨论流同步问题。

流一般可以看作信源和信宿之间的虚拟连接。信源或信宿可以是进程,也可以是设备。比如说,通过网络发送数据的时候,可能会由某个发送端进程来从磁盘上读取音频文件,随后逐字节地通过网络发送该文件中的内容。接收者进程将会获取收到的字节,并将它们传递给本地音频设备。整个过程如图2.35(a)所示。另一方面,在多媒体分布式系统中,可以通过在信源和信宿之间建立直接连接来为流提供支持。比如说,可以将由照相机生成的视频流直接发到显示设备,如图2.35(b)所示。

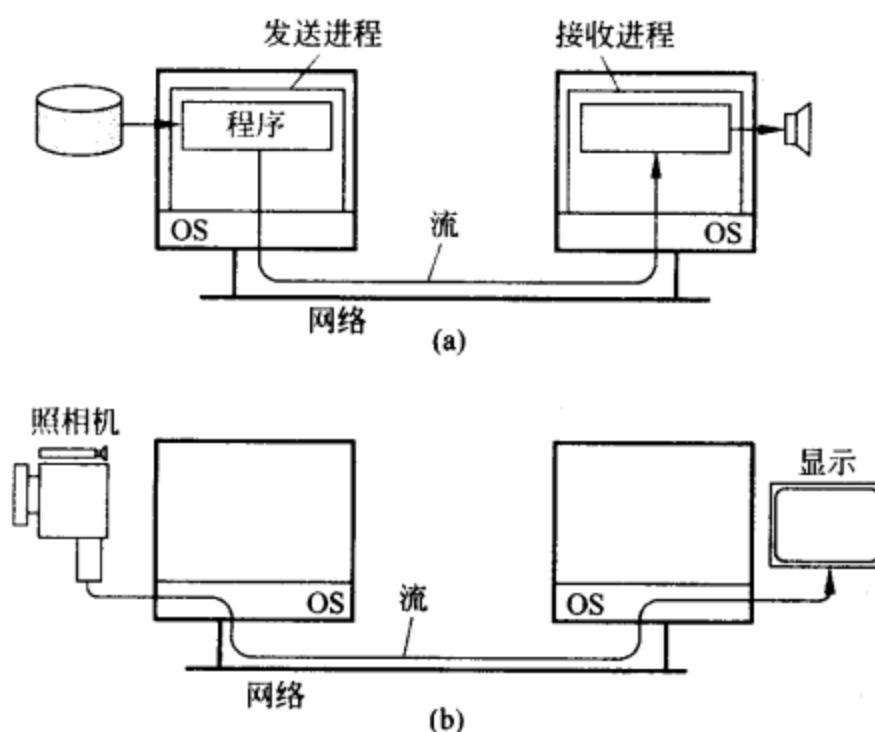


图2.35 两种流连接

(a) 通过网络在两个进程间建立流连接; (b) 在两个设备间直接建立流连接

另一个问题是:是只能有单个的信源和信宿,还是可以建立多方通信。最常见的多方通信情况是将多个信宿连接到单个流上,这时数据流是对若干接收者进行多播,如图2.36所示。



图2.36 流对多个接收者进行组播的示例

流的多播存在的主要问题在于,如果接收者对流的质量提出不同的要求该怎么办?举个例子,考虑一下某个传输高画质立体声电影的信源,这种情况下必须使用复杂流,其中包含频率为 50Hz 的用于图像传输的视频子流以及两个提供 CD 音质的音频子流。即使使用了先进的压缩技术,要接收该复杂流所需带宽仍然高达 30Mb/s(Steinmetz 和 Nahrstedt 1995)。不是所有接收者都有能力处理如此大量的数据。因此,流应该与过滤器(filter)一同使用(Yeadon 等 1996),过滤器能够将输入流调整为各种不同质量的输出流,如图 2.36 所示。我们将在稍后继续讨论流质量控制问题。

2.5.2 流与服务质量

时间敏感的(以及其他非功能性的)需求一般统称为服务质量(quality of service, QoS)需求。这种需求描述了底层分布式系统及网络在确保传输质量方面的需要。连续数据的 QoS 主要涉及时间、容量以及可靠性。在本节中我们将详细讨论 QoS 以及它与流的建立之间的关系。

1. 描述 QoS

可以通过多种方法来表达 QoS 需求。其中一种方法是,提供精确的流规格说明,其中指定了要求的带宽、传输速率、延迟等。图 2.37 中给出了流规格说明的一个示例,该规格说明是由 Partridge 于 1992 年开发的。

输入特征	请求的服务
最大数据单元大小(B)	丢失敏感度(B)
令牌存储桶速率(B/s)	丢失间隔(μs)
令牌存储桶大小(B)	突发丢失敏感度(数据单元)
最大传输速率(B/s)	最小可察觉延迟(μs) 最大延迟范围(μs) 要求确保的质量

图 2.37 流规格说明

在 Partridge 开发的模型中,通过令牌存储桶算法(token bucket algorithm)的方式来表述流的特征。这种算法刻画了流在网络通信方面的形态。令牌存储桶的原理如图 2.38 所示,它的基本思想是以恒定的速率来产生令牌,每个令牌代表允许应用程序向网络传递的固定数量的字节。令牌在存储桶中进行缓冲存储,存储桶容量有限,当它装满之后就开始丢弃令牌。每当应用程序希望向网络传递一个大小为 N 的数据单元时,它必须首先从存储桶中删去总共 N 字节。因此,假定每个令牌占 k 字节,应用程序就必须从存储桶中删去至少 N/k 个令牌。

令牌存储桶算法所要达到的目的在于用相对恒定的速率向网络传输数据,传输速率由令牌产生的速率决定。但是,它也允许进行某些突发性的传输,比如允许应用程序提供一个完整的存储桶,该存储桶足以容纳在单次操作中所有要向网络传输的令牌。为了避免过度的突发传输,数据流也可以为传输速率指定一个最大值。在流规格说明中,应用程

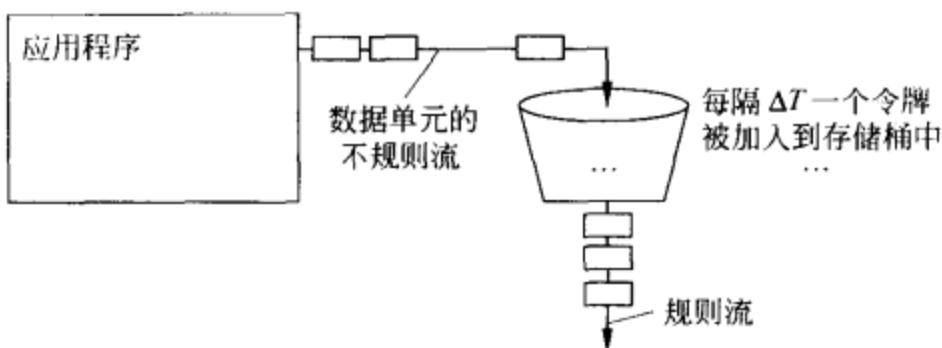


图 2.38 令牌存储桶算法的原理

序必须保证将依照令牌存储桶算法的输出结果来向通信系统送出数据单元。

除了指定数据单元之间的时间关系以外,流规格说明中还包括有对服务的要求。丢失敏感度(loss sensitivity)以及丢失间隔(loss interval)共同规定了可接受的最大丢失率(比如 1b/s)。突发丢失敏感度规定了允许连续丢失数据单元的数量。

最小可察觉延迟(minimum delay noticed)规定了网络最多可以将数据单元传输延迟多久而不引起接收者注意,如果实际延迟大于该时间,接收者将会察觉到该延迟。与这项指标有关的是最大延迟范围(maximum delay variation),它规定了可容忍的延迟时间范围。对于视频和音频数据来说,指定信号延迟范围显得尤为重要。

最后,要求保证的质量(quality of guarantee)是一个数,代表了要求得到何种质量等级的服务。基本上说,较小的数表示的是,如果通信系统无法提供所请求的服务质量,并没有什么关系。而较大的数表示的是,如果无法保证所请求的服务等级,系统就不要确立流连接,因为低于请求等级的服务无法满足客户要求。

在流的规格说明中所存在的问题是,应用程序也许并不了解自身的需求。特别是,如果强迫每一个用户都按照令牌存储桶参数的格式来指定服务质量,比如丢失敏感度等,将会很快让服务提供者忙不过来。因此可以采取另一种方案,即将流分类,进而为具体的流规范提供合理的默认值。比如说,用户可能只需要指定需要的流是视频流还是音频流,如果是音频流,则可以在高质量、中等质量还是低质量中进行选择。同样,对于视频流也可以采取类似的分级方案。

就像文献(Partridge 1994)中讨论过的一样,分类与进行详细的流规格说明并没有太多不同,不同之处仅仅在于需要指定的参数的数目以及每个参数可选值的数目。

2. 建立流

一旦以流规格说明之类的方式对数据流进行了描述,分布式系统就可以为流的建立以及满足其在 QoS 方面的需求而分配资源了。与流管理相关的资源主要包括带宽、缓冲区以及处理能力等。一旦确认数据单元已经做好传输(比如指定了传输的优先级)安排以后,就为它保留带宽。在路由器和操作系统中为数据单元分配缓冲区之后,数据单元就可以进入队列等待进一步处理。最后,必须保证数据单元得到及时处理,这要求与之相关的任务,比如调度程序、编码器和解码器、过滤器等都必须做好适当的调度,以保证它们能够分配到合适的 CPU 时间。

用来指定数据流特性以及 QoS 需求的参数不必要与类似的资源参数直接相一致。比如说,如果指定了网络应该确保至多不超过 k 个连续的数据单元丢失,从这个要求就可以得出从源到目的地路径上的路由器所要分配的静态缓冲区。但是实际上这种分配方案可以通过其他的流特性计算出来,从而得到由网络提供的稳定的并且从统计角度来说有保证的服务。

不幸的是,当前没有适合以下用途的单一的最佳模型:(1)QoS 参数指定;(2)对通信系统中资源的一般性描述;(3)将 QoS 参数转化为对资源的使用。由于缺乏这样的模型,导致了在服务质量表述和确定方面的困难,也导致了不同系统使用不同的彼此互不兼容的方案。

为了说得更具体一些,并且考虑到分布式系统中的 QoS 主要依赖于底层网络提供的服务,让我们简单地研究一下一个具体的 QoS 协议,它是用来为连续的流保留资源的。RSVP(resource reservation protocol, 资源保留协议)是一种传输层控制协议,它用于在网络路由器上保留资源(Zhang 等 1993, Bradenetal 1997)。

RSVP 中的发送者给出流规格说明,其中描述了数据流的特性,包括带宽、延迟、信号时间不稳定范围等,与图 2.37 中的流规格说明极为类似。该规格说明被移交给与发送者位于同一台机器上的 RSVP 进程,如图 2.39 所示。RSVP 进程并不对流规格说明进行解释。实际上,在收到发送者送来的流规格说明之后,RSVP 仅仅是将该说明存储到本地。RSVP 是一种由接收者启动的 QoS 协议,这就是说要求接收者沿传输路径向发送者发送保留资源的请求。通过将流规格说明存储起来,RSVP 可以防止所保留的资源超出本要求。

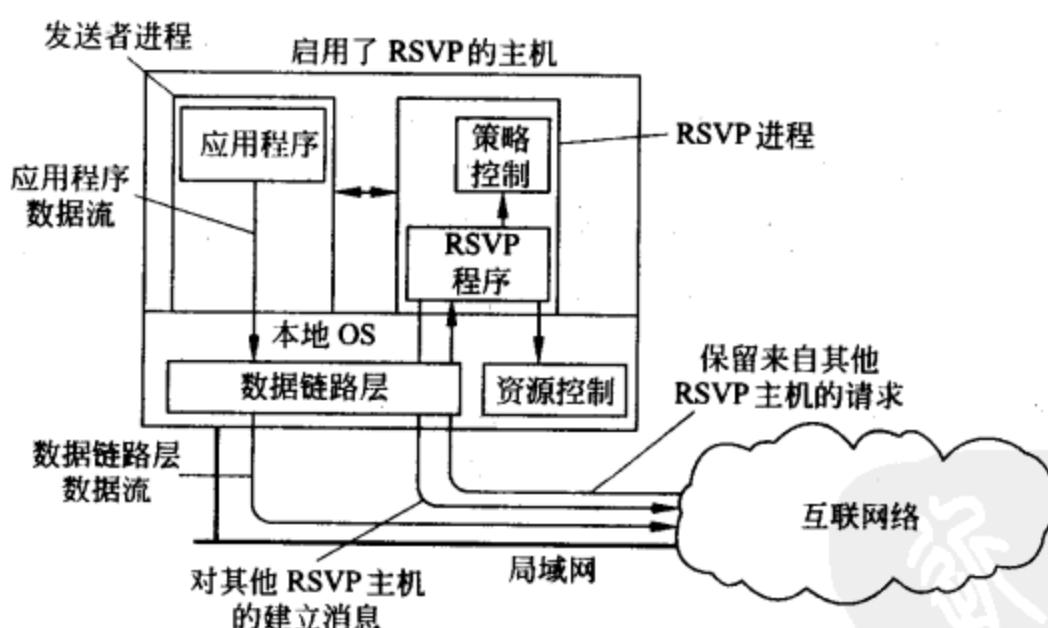


图 2.39 在分布式系统中用于资源保留的 RSVP 的基本组织结构

在 RSVP 中,发送者首先建立连接到可能的接收者的路径,并向路径中途的每个中介点提供数据流的流规格说明。接收者准备接收输入的数据单元时,它首先将资源保留的请求沿上行路径传给发送者。这种请求的格式在本质上与原来的流规格说明是一样的,只是其中设定的参数值代表的 QoS 等级可能会比发送者指定的低,因为发送者必须考虑到最苛刻的接收者的要求。

在 RSVP 进程接收到资源保留请求之后, 它将该请求传递给资源供应控制模块以核实是否有足够的资源可用。该请求同时还被传递给策略控制模块, 以检查接收者是否拥有保留资源的许可。如果这两项检查都通过了, 就可以对进行资源保留了。

资源保留高度依赖于数据链路层。实际上, 要使 RSVP 能够工作, RSVP 进程必须将流规格说明中的 QoS 参数转化为数据链路层能够识别的数据。比如说, 将一个要求大量带宽的请求转换为赋予每一个携带流数据的帧以最高优先级。这种转换是基于原始的流规格说明(代表了发送者产生数据的最高速率)以及数据链路层的可用带宽的, 它足以满足接收者的 QoS 需求。

当数据链路层提供了自己的一套参数, 并用它们来描述 QoS 需求时, 就必须采用其他方法, ATM 网络就是这样的。在 ATM 网络中, 数据以单元的形式传输, 这些单元称作信元(cell), 信元中含有 48 字节的有效载荷字段以及 5 字节的报头。ATM 允许 RSVP 进程指定最大信元传输速率、信元长时间平均传输速率、可接受的最低信元传输速率、最大可接受的信元间抖动, 以及多种其他的 QoS 参数。在本例中, 由 RSVP 进程负责将面向流的流规格说明转换成 ATM 专用的参数值, 而 ATM 层将会负责满足实际的 QoS 需求。

2.5.3 流同步

多媒体系统中一个重要的问题是, 不同的流(可能都是复杂流), 可能要互相保持同步。流同步涉及的问题是要在流之间保持时间上的关联, 流同步有两种类型。

最简单的一种同步是在离散数据流与连续数据流之间保持同步。我们来考虑一下 Web 上的带有音频的幻灯片放映。幻灯片以离散数据流的形式由服务器传输给客户, 而同时客户要播放特定的(一部分)音频以配合当前放映的幻灯片, 而音频也是从服务器取得的。在这种情况下, 音频流必须与幻灯片的演示保持同步。

要求更加苛刻的一类同步是连续数据流之间的同步。日常生活中就有这方面的例子: 放映影片时视频流必须与音频流保持同步, 这常常称作口型同步。另一个例子是放送立体声。立体声音频流由两个子流组成, 每个子流代表一个声道。要正确地表现立体声, 就要求两个子流保持紧密的同步: 只要子流间相差 $20\mu s$ 就足以导致立体声失真。

同步是在建立流的数据单元这个层次上的。也就是说, 只要让两个流的数据单元保持同步, 就可以让两个流同步。对数据单元的选择高度依赖于看待数据流的抽象层次。让我们来更具体地考虑一下前面提过的 CD 音质的(单声道)音频流。在最适当的粒度上, 这种流呈现为 16 位样本序列。如果采样频率为 $44\ 100\text{Hz}$, 在理论上每约 $23\mu s$ 就必须进行一次同步。实践证明, 如果要达到高质量的立体声效果, 这种级别的同步确实是必需的。

然而, 当我们考虑音频流与视频流之间的口型同步时, 就必须采用比较粗糙的粒度。我们曾经说过, 视频帧必须以 25Hz 以上的速率显示。广泛采用的 NTSC 标准规定的是 30Hz , 如果采用这个标准, 我们就必须将音频样本组成逻辑单元, 音频样本的长度必须与视频帧的显示时间相同(33ms)。如果音频采样频率为 $44\ 100\text{Hz}$, 那么音频数据单元就必须包含 1470 个样本, 也就是 11 760 字节(假定样本为 16 位)。在实践中, 可以允许更大

的长度为 40 或 80ms 的单元(Steinmetz 1996)。

同步机制

现在的问题是,到底是如何做到同步的。需要先搞清楚两个问题:(1)两个流同步的基本机制;(2)在网络环境下这些机制的分布式版本。

可以在不同的抽象层次上来观察同步机制。从最低的层次上看,同步是通过显式地对单个流的数据单元进行操纵来实现的,其原理如图 2.40 所示。在本质上,存在一个专门在少数几个简单流上执行读写操作的进程,该进程确保这些操作遵守指定的时间及同步约束条件。举个例子,我们来考虑一下一部用两个输入流来表示的影片。视频流中包含有未经压缩的低质量图像,分辨率为 320×240 像素,每个像素由一个字节表示,这样的图像中每个视频数据单元的大小为 76 800 字节。假定图像要以 30Hz 的频率显示,那么每幅图像就要持续 33ms。假定音频流包含的音频样本组成了大小为 11 760 字节的单元,那么每个单元就相当于 33ms 的音频,这些我们在前面已经讨论过了。如果输入进程的输入速率为 2.5MB/s,只要每 33ms 读取一次图像并读取一次音频数据块就可以做到口型同步了。

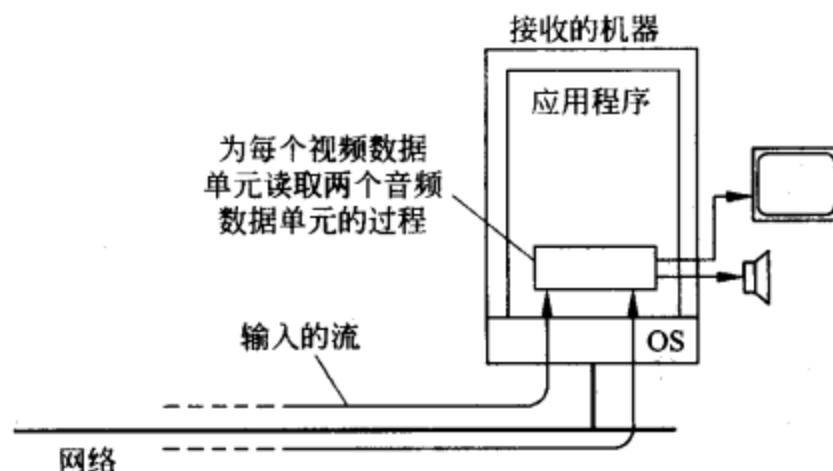


图 2.40 在数据单元层次上进行显式同步的原理

这种方法的缺陷在于,完全由应用程序来负责实现同步,但它只有低层功能可用。更好的做法是,向应用程序提供接口,允许它更加方便地对流和设备进行控制。回到上一个例子,假定视频显示有一个控制接口,允许指定图像显示的速率。该接口还提供注册用户处理程序的功能,每当有 k 个新图像到达的时候该用户处理程序就被调用。音频设备也可以提供相似的接口。通过这些控制接口,应用程序开发人员可以编写出简单的监控程序,它由两个处理程序组成,每个处理程序负责处理一个流,并由两个处理程序相配合,共同检查视频流和音频流是否完全同步,并且在必要的时候调整视频或音频单元的显示或播放速率。

最后一个例子如图 2.41 所示,它对于许多多媒体中间件系统来说是很典型的。多媒体中间件提供了一组接口用于控制视频流和音频流,其中包括控制诸如显示器、照相机和麦克风等设备的接口。每个设备和每个流都有自己的高级接口,其中包括在发生某些事件时通知应用程序的接口,后者用来编写用于流同步的处理程序。在文献(Blair 和

Stefani 1998)中给出了这种接口的例子。

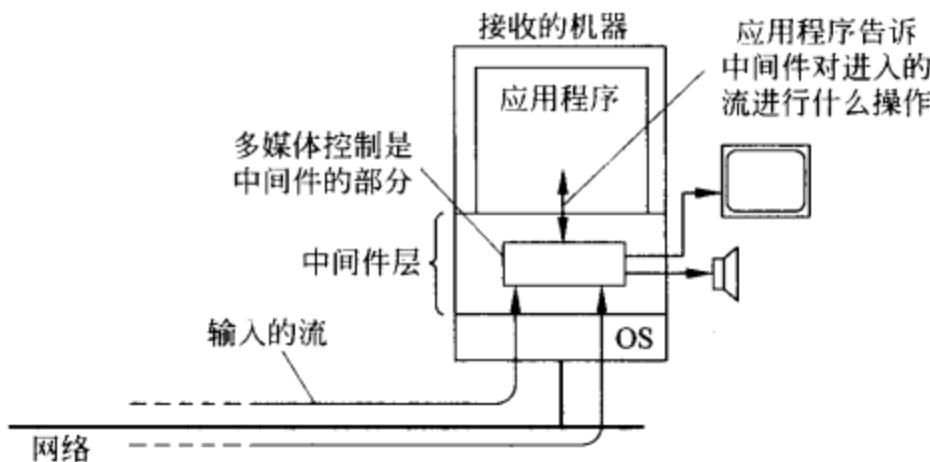


图 2.41 通过高级接口支持同步的原理

需要注意的另一个问题是，同步机制的分布性。首先，由请求同步的子流组成的复杂流的接收端需要了解同步究竟要达到哪些要求。也就是说，它必须使完整的同步规格说明(synchronization specification)在本地可用。在实践中一般是隐式地将不同流多路复用到包含所有数据单元(其中含用于同步的数据)的单个流中。

MPEG 流采用了另一种同步方案。活动图像专家组(motion picture experts group, MPEG)标准定义了一组用于视频及音频压缩的算法。现在已经有了若干种 MPEG 标准。比如 MPEG-2, 该标准原来是用来将广播级质量的视频压缩到 4~6Mb/s。在 MPEG-2 中，可以将数目不限的连续流和离散流合并到单个流中。每个输入流先转换成包的流，包中带有由基于 90kHz 的系统时钟给出的时间戳，这些包的流又被多路复用为一个节目流(program stream)，节目流中的包长度是可变的，而这些包的共同点在于其带有相同的时基。在接收端需要对节目流进行多路分离，此时再一次利用包中的时间戳作为流间同步的基本机制。

另一个重要的问题是，应该选择在发送端还是接收端进行同步。如果由发送端来处理同步，就可以将许多流合并成单个流，并且使用另一种类型的数据单元。让我们来再次考虑一下前面例子。例子中，由两个各代表一个声道的子流构成了立体声音频流。一种可能采取的做法是将两个子流彼此独立地传输给接收者，由接收者进行两个流之间的同步。很明显，由于每个子流的延迟可能不同，要做到同步是非常困难的。更好的做法是在发送端将两个子流合并起来，得到的流中的数据单元是由成对的样本构成的，分别来自左右声道的两个样本组成一对，这样接收者只要读取数据单元并将它分离成左声道样本和右声道样本就行了，在这种情况下，两个声道的延迟相同。

2.6 小结

强大灵活的进程间通信功能是任何分布式系统中必需的。在传统的网络应用中，通信常常是通过基于传输层提供的低层消息来传递原语的。中间件系统要解决的一个重要问题是：提供更高层次的抽象，与传输层接口所提供的支持相比，这能更加方便地表述进

程间的通信。

使用最为广泛的抽象方法之一就是远程过程调用(remote procedure call, RPC)。RPC的本质是一种通过过程来实现的服务,该过程的主体在服务器上执行。只向客户提供过程的特征,也就是该过程的名字以及参数。当客户调用该过程的时候,其客户端实现——称作客户存根(stub),将会负责将参数值包装进消息中,然后将消息发送给服务器。由服务器来调用实际过程并在消息中返回结果,由客户存根从返回的消息中提取结果值,然后将结果值传递回调用它的应用程序。

RPC的目的在于实现访问透明性。然而,它对引用传递的支持比较薄弱。从这个意义上说,远程对象可以提供更好的透明性。远程方法调用(RMI)本质上就是RPC,但是它是专门针对远程对象的。二者主要的不同在于RMI允许将系统范围的对象引用作为参数来传递。

RPC和RMI提供了同步通信功能,这些功能可以将客户堵塞,直到服务器送回应答为止。虽然这种机制的某些变化放松了模型中对同步的要求,但是实践证明通用的高级面向消息模型会更加方便。

在面向消息的模型中,问题是通信是否是持久的以及通信是否是同步的。持久通信的本质是提交进行传输的消息,由通信系统来存储,直到系统将其传输给接收者为止。也就是说,发送者和接收者都不必保持运行来等待消息传输。而在暂时通信中,不提供存储功能,因此发送消息时接收者必须等待接收该消息。

在异步通信中,允许发送者在将消息提交并进行传输后立即继续运行——甚至可以在消息实际发送出去之前就继续运行。而在同步通信中,发送者会被阻塞,至少要到消息得到接收之后才能释放。发送者还有可能要等到消息已经交付给接收者之后甚至等到接收者进行响应之后才能继续运行,最后一种情况就像RPC一样。

面向消息的中间件模型一般提供持久异步通信,用在RPC和RMI不适用的场合。它们主要用来协助将高度分散的数据库集成进大规模信息系统中。其他的应用还包括电子邮件和工作流。

流是一种完全不同的通信方式,它的主要问题是两个连续的消息是否有时间上的联系。在连续数据流中,每个消息都规定了端到端的最大延迟时间。另外,发送的消息还要受端到端最小延迟时间的约束。这种连续数据流的典型例子包括视频流和音频流。这种时间上的联系以及希望底层通信子系统提供的服务质量常常是难于说明和实现的。使得事情变得复杂的因素之一是信号的延迟不稳定性。即使平均性能是可接受的,传输时的实际延迟时间的变化可能会导致性能恶化到不可接受的程度。

习题

1. 在许多分层协议中,每一层都有自己的报头。如果每个消息前部只有单个报头,其中包含了所有的控制信息,无疑会比使用单独的多个报头具有更高的效率。为什么这么做?

2. 为什么传输层通信服务常常不适于构建分布式应用程序?

3. 一种可靠的多播服务允许发送者向一组接收者可靠地传递消息。这种服务是属于中间件层还是属于更低层的一部分？
4. 考虑一个带有两个整型参数的过程 incr。该过程将两个参数的值分别增加 1。现在假定调用它时使用的两个参数是同一个变量，比如 incr(i,i)。如果 i 的初始值是 0，在执行引用调用之后 i 将变为什么值？如果使用复制—还原调用呢？
5. C 语言中有一种称为联合(union)的构造，其中的记录（在 C 语言中称作结构）的字段可以用来保存几种可能值中的一个。在运行时，没有可靠的办法来分辨其中保存的是哪一个值。C 的这种特性是否与远程过程调用有某些相似之处？请说明理由。
6. 处理 RPC 系统中参数转换的一种方法是，每台机器以自己系统使用的表示方式来发送参数，由另一方在必要的情况下进行转换。可以通过首字节中的代码来表示发送消息机器所用的系统。然而，由于要在首个字中找到开头的字节这本身也是一个问题，这种方法能行得通吗？
7. 假定客户通过异步 RPC 对服务器进行调用，随后等待服务器使用另一异步 RPC 返回结果。这种方法与客户执行常规的 RPC 有没有什么不同？如果使用的是同步 RPC 而不是异步 RPC，情况又如何呢？
8. 在 DCE 中，服务器在守护程序中注册自身。如果换一种方法，也可以总是为它分配一个固定的端点，然后在指向服务器地址空间中对象的引用中就可以使用该端点。这种方法的缺陷在哪里？
9. 给出一种用来让客户绑定到暂时远程对象的对象引用的实现实例。
10. Java 和其他语言支持异常处理，当错误发生时会引发异常。如何在 RPC 和 RMI 中实现异常处理？
11. 将静态和动态 RPC 区分开来有用吗？
12. 某些分布式中间件系统的实现是完全基于动态方法调用的。甚至连静态的调用也被编译成动态的调用。这种方法的优点在哪里？
13. 描述一下客户和服务器之间使用套接字的无连接通信是如何进行的。
14. 说明 MPI 中 mp_bsend 原语和 mp_isend 原语之间的区别。
15. 假定只能使用暂时异步通信原语，再加上异步 receive 原语，如何实现用于暂时同步通信的原语？
16. 假定只能够使用暂时同步通信原语，如何实现用于暂时异步通信的原语？
17. 通过 RPC 实现持久异步通信有意义吗？
18. 在本章中我们讲过，为了自动启动一个进程以从输入队列中获取消息，常常要使用守护程序来监视输入队列。请给出一种不使用守护程序的实现方法。
19. IBM MQSeries 以及许多其他消息队列系统中的路由表是人工配置的。描述一种自动完成配置工作的简单方法。
20. 如何将持久异步通信加入到基于远程对象 RMI 的通信模型中去？
21. 在持久通信中，接收者一般拥有自己的本地缓冲区，如果接收者不在运行状态，可以将消息放入该缓冲区中去。为了创建这种缓冲区，必须指定它的大小。分成两方进行辩论：一方认为这种指定缓冲区大小的行为是可取的，而另一方反对这种指定大小的

行为。

22. 请说明为什么暂时同步通信在可扩展性方面存在固有的问题,以及如何解决这些问题。

23. 给出一个将多播应用于离散数据流的例子。

24. 当一组计算机组成一个逻辑上或者物理上的环时,如何确保传输延迟不超过允许的最大端到端延迟时间?

25. 当一组计算机组成一个(逻辑上或者物理上的)环时,如何确保传输延迟不小于允许的最小端到端延迟时间?

26. 想象一下,某个令牌存储桶规范说明中的最大数据单元大小是 1000B,令牌存储桶速率是 10MB/s,令牌存储桶的大小是 1MB,而最大传输速率是 50MB/s。突发的传输可以以最大传输速率持续传输多长时间?

27. 在这道练习题中,要求实现一个使用 RPC 的简单客户-服务器系统。服务器提供一个名为 next 的过程,该过程接受一个整型数作为输入,并且返回紧接着该整型数之后的一个数。编写一个客户使用的名为 next 的存根,它的任务是使用 UDP 将参数发送给服务器,随后等待服务器响应,如果过长时间未收到响应就认为超时了。服务器过程必须在一个公开的端口监听并接受请求,完成这些请求所要求的操作,然后送回结果。

第3章 进 程

在第2章中,我们重点讨论了分布式系统中的通信。通信是在进程间进行的,在本章中我们将详细讨论各种进程在分布式系统中是如何发挥重要作用的。进程的概念源自于操作系统,它定义为执行中的程序。从操作系统的角度来说,进程管理和调度也许是它要处理的最重要的问题,但是对于分布式系统来说,还存在许多与这些问题同等重要,甚至更加重要的问题。

例如,为了有效地组织客户-服务器系统,通常使用多线程技术更为方便。在分布式系统中,线程的主要作用就是以适当的方式来构建客户和服务器,使得通信和本地处理过程可以并行进行,从而获得性能上的提高。我们将在3.1节中讨论有关内容。

我们在第1章中曾经指出,客户-服务器组织结构在分布式系统中是很重要的。在本章中,我们将对客户和服务器的典型组织结构作详细的剖析,并介绍服务器的一般设计问题。另外,我们还将考察通用的对象服务器,它是实现分布式对象的基本手段。

线程在不同机器之间的迁移是一个重要问题,对于广域分布式系统来说,这个问题显得特别重要。进程的迁移(更明确地说是代码的迁移)有助于获得可扩展性,也可以帮助动态地配置客户和服务器。本章将讨论代码迁移的确切含义及其相关内容。

本章的最后一个主题涉及一种称为软件代理(agent)的新兴机制。与非对称的客户-服务器模型相反,多代理系统大致上是由一组同等重要的代理组成的,这些代理协同工作以实现某个公共目标。软件代理也是一种类型的进程,它能够以不同的形式出现。在本章的最后一节中,将从分布式系统的角度讨论代理的实质以及代理之间的协作方式。

3.1 线 程

虽然进程(process)构成了分布式系统中的基本组成单元,但是实践表明,操作系统提供的用于构建分布式系统的进程在粒度上还是太大了。而就粒度而言,将每个进程细分为若干控制线程(thread)的形式则更加合适,可以使构建分布式应用程序变得更加方便,并可获得更好的性能。在本节中,我们将详细分析线程在分布式系统中所扮演的角色,并且说明线程的重要性。关于线程以及使用线程构建应用程序的更详细内容可以在文献(Lewis 和 Berg 1998)以及(Stevens 1999)中找到。

3.1.1 线程简介

为了理解线程在分布式系统中所扮演的角色,重要的是要了解进程是什么,以及进程与线程之间的关系。为了程序执行的需要,操作系统创建多个虚拟处理器,每个虚拟处理器

器运行一个程序。为了保持对这些虚拟处理器的追踪，操作系统中有一张进程表（process table），其包含的条目中存储着 CPU 寄存器值、内存映像、打开的文件、统计信息、特权信息等。进程一般定义为执行中的程序，也就是当前在操作系统的某个虚拟处理器上运行的一个程序。操作系统特别注意确保独立的进程无法有意或者无意地破坏其他独立进程运行的正确性。也就是说，多个进程并发地共享同一个 CPU 以及其他硬件资源这样一个事实是透明的。一般说来，操作系统需要硬件支持来实现这种隔离。

要得到这种并发透明性需要付出相对较高的代价。例如，每次创建一个进程的时候，操作系统必须创建一个完整的独立地址空间。空间分配意味着要对内存段进行初始化，比如先对数据段清零，将有关的程序复制到文本段中，随后为临时数据建立堆栈。在两个进程之间切换 CPU 的开销同样会比较大，除了要保存 CPU 环境（包括寄存器值、程序计数器、堆栈指针等）以外，操作系统还必须修改内存管理单元（memory management unit，MMU）的寄存器，并且将位于转换后备缓冲器（translation lookaside buffer，TLB）中的地址转换缓存内容标记为无效。另外，如果操作系统支持同时运行的进程数目超出主存容纳能力，则必须在切换进程之前先在主存和磁盘之间进行交换。

线程与进程从下面意义来说是非常相近的：线程同样可以看作是程序的一部分在虚拟处理器上的执行。然而，线程与进程不同的是，如果要取得高度的并发透明性会导致性能降低的话，那么线程就不会有这样的企图。因此，线程系统一般只维护用来让多个线程共享 CPU 所必需的最少量信息。特别是，线程环境（thread context）中一般只包含 CPU 环境以及某些其他的线程管理信息。比如说，线程系统可能会追踪线程当前正在某个互斥变量上阻塞的事实，这样，系统就不会选择该线程执行。那些对于多线程管理不是完全必要的信息通常被忽略。由于这个原因，在单个进程中防止数据遭到线程错误访问的任务完全落在了应用程序开发人员的肩上。

这种方法有两点需要特别注意。首先，多线程应用程序的性能至少不会比其单线程版本差。事实上，在很多情况下，多线程能够提高性能。其次，由于线程并不像进程那样彼此隔离并受到系统自动提供的保护的，因此多线程应用程序的开发需要付出更多的努力。设计合理并实行简单的原则会大有益处。不幸的是，现在的实践却证明这条原则并没有得到很好的理解。

1. 非分布式系统中的线程用法

在讨论线程在分布式系统中的作用之前，首先让我们来考察它们的传统用法——在非分布式系统中的用法。由于在进程中使用多线程有许多好处，这就使得多线程系统的使用变得流行起来。

多线程最显著的好处来自以下事实，那就是在只拥有单线程的进程中，一旦执行了造成阻塞的系统调用，整个进程就被阻塞了。为了说明这一点，我们来考虑一下诸如电子表格这样的应用程序，并且假定用户需要连续并且交互地改变值。电子表格程序的重要特点是，需要维护不同单元格之间的函数依赖关系，而这些单元格常常位于不同的数据表中。因此，一旦修改了某个单元格，所有关联的单元格都要自动更新。如果用户对某个单元格中的值进行改动，将会触发程序的大量计算操作。如果只有单个控制线程，在程序等

待输入的时候就无法进行计算,同时也很难在进行计算的时候接受输入。最方便的解决办法是使用至少两个控制线程:其中一个管理与用户之间的交互,而另一个进行数据表的更新工作。

多线程的另一个优点是,在多处理器系统上执行多线程程序的时候,可以使用并行操作技术。在这种情况下,为每一个线程分配一个CPU,同时将共享的数据存储在共享主存中。如果设计得当,这种并行操作可以是透明的:进程可以与运行在单处理器系统上一样好,只是比较慢。随着相对便宜的多处理器工作站的出现,并行操作多线程技术变得愈发重要。这种计算机系统的典型应用是在客户-服务器应用程序中用来运行服务器程序。

多线程技术在大型应用程序上下文环境中也是很有用的。这种应用程序一般是作为一组相互协作的程序开发出来的,其中每一个程序都通过独立的进程执行。这种方法的典型例子是 UNIX 环境。程序间协作是通过进程间通信(IPC)机制实现的。对于 UNIX 系统来说,这套机制中通常包括已命名管道、消息队列以及共享内存段,请参阅文献(Stevens 1992)。所有 IPC 机制都有一个主要的缺陷,就是其中的通信需要开销庞大的上下文环境切换,如图 3.1 中三个点所示。

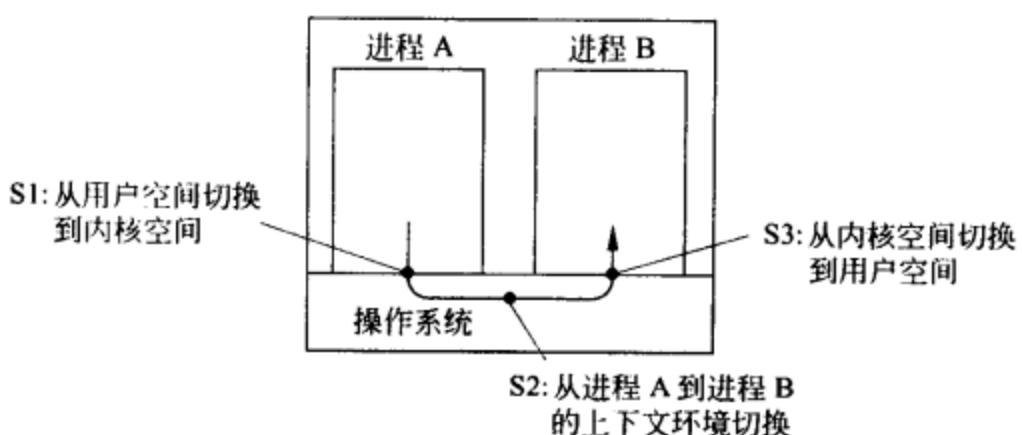


图 3.1 IPC 造成的上下文环境切换

由于 IPC 需要内核干预才能进行,因此,要进行 IPC 的进程一般首先要从用户模式切换到内核模式,如图 3.1 中的 S1 所示。这就需要改变 MMU 中的内存映像,同时还要刷新 TLB。在内核中进行进程上下文环境的切换(图 3.1 中的 S2),随后就可以从内核模式切换回用户模式以使得通信的另一方能够激活。后一次切换也同样需要改变 MMU 映像并且刷新 TLB。

除了使用进程之外,应用程序也可以这样构建:其中不同的部分都通过独立的线程来执行。各个组成部分之间的通信完全通过数据共享来处理。线程切换可以完全在用户空间中完成,也可以由内核来掌管线程并对它们进行调度。这样在性能上就可以得到极大的提升。

最后,使用线程还出于纯粹的软件工程上的考虑。有许多应用程序如果用一组相互协作的线程来构建,常常是比较简便的。想象一下那些需要完成若干任务(这些任务或多或少是独立的)的应用程序。比如说,在字处理器中,可以用独立的线程来分别处理用户输入、拼写检查及语法检查、文档布局、索引生成等工作。

2. 线程的实现

线程一般是以线程包的形式提供的。这种包中包含有创建与销毁线程的操作,以及对诸如互斥变量以及条件变量之类的同步化变量的操作。有两种实现线程包的基本方法:第一个方法是可以构造一个完全在用户模式下执行的线程库;另一个方法是由内核来掌管线程并进行调度。

采用用户级线程库有很多好处。首先,创建和销毁线程的开销很小。由于所有线程管理工作都保持在用户地址空间中进行,线程创建的开销主要取决于为线程堆栈的建立而分配内存的开销。与此类似,销毁线程的工作主要是释放线程堆栈所占用的内存,因为线程销毁后它将不会再使用这些内存。这些操作的开销都不大。

采用用户级线程的好处还有,可以通过不多的几条指令来实现线程上下文环境的切换。基本上,只有 CPU 寄存器值需要存储,并随后用将要切换到的线程的原先存储的值重新加载到 CPU 寄存器中去。并不需要内存映像的改变、TLB 的刷新以及 CPU 统计等。线程上下文环境的切换是在两个线程需要同步的时候发生的,比如说在进入共享数据段的时候。

然而,用户级线程的主要缺陷在于,对引起阻塞的系统调用的调用将会立即阻塞该线程所属的整个进程,也就阻塞了所属进程中的所有其他线程。我们曾经说过,如果要将大型应用程序分为若干部分来构建,所有部分在逻辑上可以同时执行,那么线程是非常有用的。在这种情况下,在 I/O 过程中发生的阻塞不应该妨碍此时其他部分的执行。对于有这种需要的应用程序,用户级线程没什么益处。

这些问题大都可以通过在操作系统的内核实现线程的方法来解决。不幸的是,这种解决方法的代价非常大:每一个线程操作(创建、删除、同步等)都必须由内核来执行,并且需要进行系统调用。线程上下文环境的切换的开销变得与进程上下文环境切换的开销一样大。结果,用线程代替进程的大多数优点都不复存在了。

另外一种解决方法是采用用户级线程和内核级线程的混合形式,一般称作轻量级进程(lightweight processes,LWP)。LWP 运行在单个重量级进程的上下文环境中,每个进程中可以包含多个 LWP。除了 LWP 以外,系统还提供用户级线程包,向应用程序提供了创建和销毁线程等普通操作。另外,包中还提供了用于线程同步的工具,比如互斥变量和条件变量(请参阅 1.4 节)。重要的是,线程包是完全在用户空间中实现的。也就是说,执行这些线程操作不需要内核的干预。

线程包可以由多个 LWP 共用,如图 3.2 所示。这就意味着每个 LWP 可以运行于自己的用户级线程上。建立多线程应用程序时首先要创建线程,随后为每个 LWP 分配一个 LWP。将线程分配给 LWP 的行为一般是隐式的,并且向程序员隐藏。

用户级线程和 LWP 一般是通过以下方式结合起来:线程包中有一个用于调度下一个线程的简单例程。在创建 LWP(一般通过系统调用来创建)的时候,LWP 得到了自己的堆栈,并且将会得到通知执行调度例程,该调度例程会寻找下一个线程来执行。如果有多个 LWP 存在,每一个 LWP 都会执行该调度例程。用来追踪当前线程集的线程表是由各 LWP 共享的。通过完全在用户空间中实现的互斥标志来对该表进行保护,以确保只

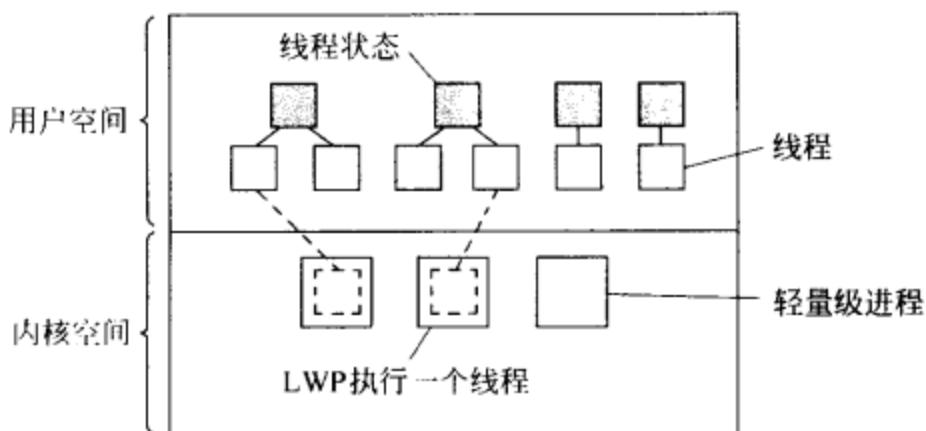


图 3.2 将内核模式的轻量级进程与用户级线程结合使用

能对它执行互斥访问。也就是说，各 LWP 间的同步并不需要任何内核支持。

如果 LWP 找到了一个可运行的线程，它就将上下文环境切换到该线程。此间，其他 LWP 也会寻找其他可运行的线程。如果线程由于互斥变量或者条件变量需要阻塞，它就会在完成必要的管理工作之后调用调度例程。如果找到了另一个可运行的线程，就会将上下文环境切换成该线程拥有的上下文环境。这种方法的优点在于，LWP 在执行线程时不需要得到通知：上下文环境切换完全在用户空间中实现，并且对于 LWP 来说切换程序所使用的都是普通的程序代码。

现在，我们来看一下线程进行阻塞系统的调用时的情形。在这种情况下，执行过程不再处于用户模式，而是转变到内核模式中，但是仍然在当前的 LWP 上下文环境中继续执行。在当前的 LWP 无法继续执行的时候，操作系统会将上下文环境切换到另一个 LWP，这就意味着上下文环境重新切换回了用户模式。被选中的 LWP 将会简单地从上一次停止的地方继续执行。

将 LWP 与用户级线程包结合起来使用有很多优点。首先，线程创建、销毁及同步工作的开销相对较小，并且根本不需要内核干预。其次，如果某个进程中有多线程，则阻塞的系统调用将不会导致整个进程被挂起。第三，应用程序并不需要了解 LWP，它能见到的只是用户级线程。最后，通过在不同 CPU 上执行不同 LWP，LWP 可以在多处理器系统中方便地应用。拥有多个处理器这个事实可以对应用程序完全隐藏。轻量级进程与用户级线程组合的唯一缺点是，仍然必须进行 LWP 的创建和销毁工作，这个工作的开销并不比内核级线程的小。然而值得庆幸的是，只是偶尔需要进行 LWP 的创建和销毁工作，而且受到操作系统的完全控制。

另一种与轻量级进程相似的方法是使用调度激活 (scheduler activations) (Anderson 等 1991)。调度激活和 LWP 之间的本质区别在于，在调度激活中，当线程在系统调用中被阻塞时，内核对线程包进行上行调用 (upcall)，调用调度例程来选择下一个可执行的线程来执行。当线程释放的时候，也要重复相同的步骤。然而，使用上行调用是一种不太好的做法，因为它破坏了分层系统的结构，在分层系统中只允许对紧接着的下一层进行调用。

3.1.2 分布式系统中的线程

线程的重要特性之一是,它提供了一种方便的方法,使得在进行会导致阻塞的系统调用时不阻塞该线程所属的整个进程。这种特性使得在分布式系统中使用线程变得特别有吸引力,因为利用它可以极为方便地将通信表述为同时维护多个逻辑连接的形式。我们将通过对多线程的客户和服务器分别进行详细考察来说明这一点。

1. 多线程客户

为了确立高度的分布透明性,广域网中的分布式系统需要隐藏较长的进程间消息传播的时间。在广域网中,传输的延迟很容易达到上百 ms,甚至几 s。

隐藏通信时间延迟的常规方法是启动通信后立即进行其他工作。这方面典型的例子是 Web 浏览器。在很多情况下,Web 文档是由 HTML 文件组成的,HTML 文件中包含有纯文本文件以及图像组、图标等。为了获得 Web 文档中每一个组成部分,浏览器必须建立 TCP/IP 连接,读取输入数据并将数据传递给显示部件。建立连接和读取输入的数据在本质上都是可能导致阻塞的操作。在进行远程通信的时候,需要面对的问题是每个操作所花费的时间可能会相对很长。

Web 浏览器一般首先获取 HTML 页面,随后再显示它。为了尽量隐藏通信时延,某些浏览器在接收数据的过程中就开始显示这些数据。首先将文本显示出来以便用户查看,并且提供页面滚动之类的功能,同时继续获取组成页面的其他文件,比如图像等。在收到这些文件之后再显示它们。用户不必等待浏览器取得整个页面的所有组件就能够查看页面。

从效果上来说,看起来就好像 Web 浏览器在同时进行多项任务一样。实践已经证明,以多线程客户的模式来开发浏览器可以显著地使问题得到简化。只要取得了主 HTML 文件,就可以激活多个独立的线程,它们分别负责取得页面的各个部分。每个线程都与服务器建立一个独立连接以获取数据。只要假定进行导致阻塞的调用不会将整个进程挂起,与服务器建立连接和读取数据的过程就可以使用标准的(可能导致阻塞的)系统调用来编制。在文献(Stevens 1998)中也说明了,用于每一个线程的代码都是相同的,并且首先是简单的。在浏览页面期间,用户只会注意到图像等内容显示上的延迟,但是可以对文档进行浏览。

在可以同时打开多个连接的情况下,使用多线程的 Web 浏览器还有另一个明显的好处。在上面的例子中,建立了到同一个服务器的多个连接。但是,如果该服务器的负载过重,或者比较慢,这种方法与逐个获取组成页面的文件的方法相比,在性能上并不会有什么真正的提升。

然而,在许多情况下,Web 服务器会复制到多台机器上,每个服务器负责提供一组完全相同的 Web 文档。复制的服务器位于同一个站点,并且名字是相同的。当对 Web 页的访问请求到来时,该请求被转发到其中的一个服务器,转发到哪一个服务器的决策通常是通过循环策略或者某些其他负载平衡技术做出的(Katz 等 1994)。在使用多线程客户的时候,可以与不同的服务器副本建立连接,这样就可以并行地进行数据传输了,并且确

保整个 Web 文档完全显示出来所需的时间与使用非复制的服务器的情况相比要短得多。这种方案只在客户真正能够处理输入数据的并行流时才能够发挥作用,而使用线程正是达到这个目的的理想方式。

2. 多线程服务器

虽然我们已经看到多线程客户有许多优点,但是分布式系统中多线程技术主要还是在服务器端发挥作用。实践表明,多线程技术不仅能够显著简化服务器代码,还能够使得应用并行技术来开发高性能的服务器变得更加容易,即使在单处理器系统上也是如此。而且,目前多处理器计算机广泛地用作通用工作站,所以多线程技术就更加有用了。

为了理解使用线程给编写服务器代码带来的好处,考虑一下文件服务器的组织结构,该文件服务器可能会偶尔由于等待磁盘操作而阻塞。文件服务器一般等待输入的文件操作请求,随后执行该请求,最后送回应答。其中一种特别流行的组织结构如图 3.3 所示。图中有一个称作分发器(dispatcher)的线程,由它来读取文件操作请求。由客户将该请求发送给服务器的某个公开端点。在对请求进行检查以后,服务器选择一个空闲的(也就是阻塞的)工作者线程(worker thread),由它来处理该请求。

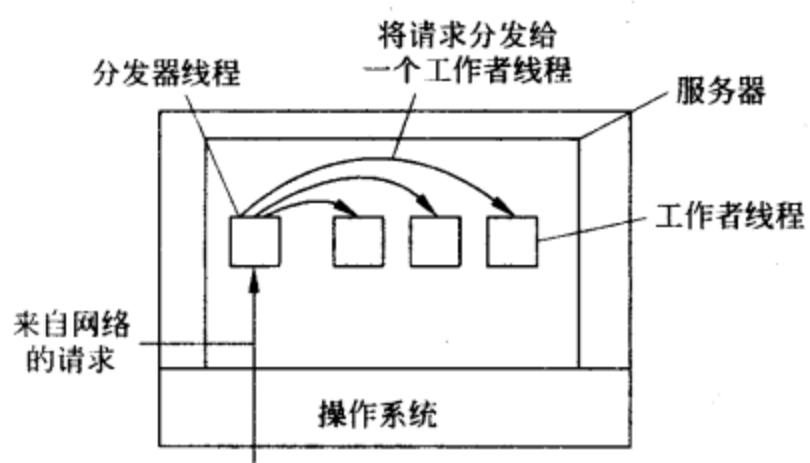


图 3.3 以分发器/工作者模型组织起来的多线程服务器

工作者在本地文件系统上执行阻塞的 read 调用,执行该调用将会导致该线程被挂起直到数据从磁盘上读出为止。如果该线程被挂起了,就选择另一个线程接着执行。比如说,可以选择执行分发器线程以完成更多工作。此外,也可能选择执行已经准备好运行的另一个工作者线程。

现在来考虑在不使用多线程的情况下文件服务器的编写方式。一种可行的做法是,让服务器作为单个线程来工作。由文件服务器主循环程序来获得请求,检验该请求,随后执行该请求直至完成,然后再接受另一个请求。在等待磁盘操作的时候,服务器是空闲的,但也不对其他请求进行处理。也就是说,来自其他客户的请求不能得到处理。另外,如果文件服务器运行于专用机器上(大多数情况下是如此),文件服务器等待磁盘操作的时候 CPU 将会完全空闲。这样做的结果是,服务器每秒能够处理的请求数目大大减少。而使用多线程能够显著提升性能,而且每个线程都是以常规方式顺序编写的。

至此我们已经看到了两种可行的设计方法:多线程文件服务器以及单线程文件服务

器。假定无法使用多线程,但是系统设计者又发现使用单线程所带来的性能损失是不可接受的,那么第三种可行方法是将服务器作为一个大的有限状态机来运行。当请求输入的时候,由惟一的线程对请求进行检查,如果可以利用缓存中的内容来满足请求当然好,如果缓存的内容无法满足就必须向磁盘发出消息,请求执行磁盘操作。

然而,向磁盘发出消息之后线程并不阻塞,而是在表中记录下该请求的当前状态,随后接收下一个消息。下一个消息可能是一个开始新工作的请求,也可能会是磁盘对之前的操作请求所做的应答。如果是新工作请求,就进行该工作。如果是来自磁盘的应答,就从表中取出之前记录下的有关信息,对应答进行处理并把结果发送给客户。如果采用这种方法,服务器必须使用非阻塞性的 send 和 receive 调用。

在这种设计中,前两种方法使用的“顺序处理”模型不见了。对于每一个发送和接收的消息来说,计算状态都必须显式地保存下来,存储在表中。从效果的角度来看,这么做是在以一种复杂的方法模拟线程及其堆栈。进程是作为有限状态机来运行的,它在接收到一个事件之后根据其内容来决定采取的操作。

现在,多线程的作用已经很清楚了。多线程能够保留顺序处理的思路,使用阻塞性的系统调用(比如,RPC 与磁盘通话),并且仍然能达到并行处理的目的。使用阻塞性系统调用能使编程更加容易,而且使并行处理能够提升性能。单线程服务器保留了使用阻塞性系统调用所带来的方便之处,但是放弃了性能。而有限状态机的方法能够通过并行获得高性能,但是由于使用非阻塞性系统调用,编程上比较困难。这些模型总结在图 3.4 中。

模型	特征
多线程	并行, 使用会导致阻塞的系统调用
单线程进程	非并行, 使用会导致阻塞的系统调用
有限状态机	并行, 使用非阻塞系统调用

图 3.4 构建服务器的 3 种方式

3.2 客户

在前面的章节中我们讨论过客户-服务器模型、客户和服务器所扮演的角色以及它们之间的交互。让我们对客户和服务器分别进行详细剖析,在本节中我们先讨论客户,服务器将在下一节中讨论。

3.2.1 用户界面

大多数客户的主要任务是与个人用户和远程服务器相交互。对用户界面的支持是多数客户的关键特征。用户与远程服务器之间的接口在多数情况下都相对简单,并且与客户端硬件集成在一起。比如说,蜂窝电话有一个简单的显示装置,它与一组传统的拨号按键组合在一起。更复杂的蜂窝电话提供电子邮件功能,这种电话也许会配上一个完整的键盘、电子便笺簿或者语音识别装置。

图形用户界面是很重要的一个类别。下面,作为一个更为传统的图形用户界面的例子,我们将首先讨论 X Window 系统。随后我们将考虑用于支持应用程序间通信的现代界面。

1. X Window 系统

X Window 系统通常也简称为 X,它用来控制按位映射的终端,这种终端包括监视器、键盘以及鼠标之类的定点设备。从这种意义上说,X 可以看作操作系统中负责控制终端的一部分。系统的核心由称作 X 内核(X kernel)的部分构成,它包括该类终端专用的全部设备驱动程序,而且对硬件有高度的依赖性。

X 内核提供相对低级的屏幕控制接口,也负责捕获键盘和鼠标事件。接口放在称作 Xlib 的库中供应用程序调用,其一般的组织结构如图 3.5 所示。(在 X 术语中,X 内核称作 X 服务器,而调用该内核功能的程序称作 X 客户。为了避免与标准客户-服务器术语相混淆,我们不采用 X 服务器和 X 客户这样的说法)。

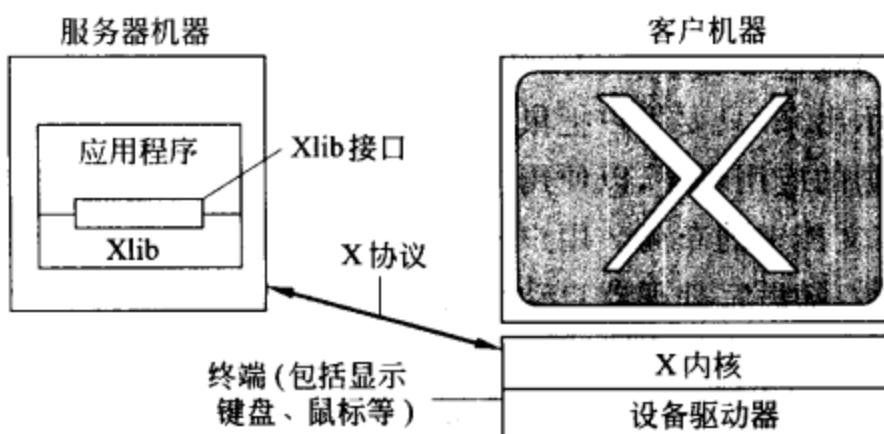


图 3.5 X Window 系统基本组织结构

X 将应用程序区分为两类:普通应用程序和窗口管理器。普通应用程序一般(通过 Xlib)请求在屏幕上创建窗口,随后它就可以使用该窗口来进行输入输出处理。另外,X 会确保在某个应用程序窗口处于激活状态的时候,也就是鼠标指针进入该窗口的时候,将来自键盘和鼠标的所有事件都传递给该应用程序。

窗口管理器(window manager)是得到特殊许可来管理整个屏幕的应用程序。普通应用程序在进行屏幕操作时,必须遵守由窗口管理器制定的约束规则。比如说,窗口管理器可能规定窗口不能相重叠,或者窗口必须始终以统一的颜色显示。因此,是窗口管理器决定了窗口系统的整体“外观和效果”。

X 有一个特性非常有趣:X 内核和 X 应用程序并不一定要驻留在同一台机器上。应该特别指出的是,X 提供了 X 协议,该协议是一个面向网络的通信协议,通过该协议 Xlib 实例可以与 X 内核交换数据和事件。这就导致了不同类型的客户-服务器组织结构的产生。在这种组织结构下,客户的复杂度会有显著的差别。在最简单形式下,客户只运行 X 内核,所有应用程序代码都在远程机器上运行。采用这种配置的终端常称作 X 终端(X terminal)。而在最复杂形式下,客户机器上包含众多应用程序,其中包括窗口管理器,基本不需要网络通信。

意识到以下这点是很重要的：X 这样的用户界面系统本质上只不过是应用程序的图形界面。应用程序从这种系统中所能够得到的信息不过是用来识别一些基本用户操作的事件，这些用户操作与连接到终端的设备直接相关，它们产生的事件包括击键、鼠标定位、按钮操作等。

2. 复合文档

我们在第 1 章中已经提到过，现代用户界面的功能要比 X 这样的系统强得多。特别是，现代用户界面允许应用程序共享单个图形窗口，并在用户指挥下利用该窗口来在应用程序间交换数据。用户可以进行的操作一般包括拖放式(drag-and-drop)操作和在位编辑(in-place editing)。

拖放功能的典型例子有：将代表文件 A 的图标移到代表垃圾箱的图标上去以删除该文件。这种情况下，用户界面所做的不仅仅是重新排列和显示图标这么简单的事情：它必须在将代表 A 的图标移到代表垃圾箱应用程序的图标的时候，将文件 A 的名字传递给与垃圾箱相关的应用程序。我们还可以想出更多此类的例子。

在位编辑可以通过包含文本和图形的文档的例子来说明。想象一下，如果该文档在标准字处理器中显示出来，只要用户把鼠标指针放到某幅图片上，用户界面就必须将有关信息传递给负责绘画的应用程序，以便用户修改该图片。比如说，用户可能想要旋转该图片，这会影响图形在文档中的布局。用户界面会计算出图片新的高度和宽度，随后将有关信息传递给字处理器，随后字处理器自动更新文档的页面布局。

隐含在这些用户界面背后的关键思想是复合文档(compound document)的概念。复合文档可以定义为一组文档，这些文档的类型可以不一致(比如文本、图片、数据表等)，但是在用户界面层上做到了无缝集成。由于用户界面可以处理复合文档，这就隐藏了这样一个事实：文档的不同部分实际上是由不同应用程序来处理的。而对于用户来说，文档中所有部分都是无缝地集成在一起的。

与 X Window 系统的情况相类似，与复合文档相关联的应用程序也不必在客户机器上执行。然而，很明显支持复合文档的用户界面必须比不支持复合文档的用户界面做更多的处理工作。

3.2.2 客户端软件与分布透明性

我们在 1.5 节中已经说过，客户端软件不仅仅包含用户界面。在许多情况下，客户-服务器应用程序中的部分处理和数据级工作是在客户端执行的。嵌入式客户软件是一类特殊的软件，用于自动取款机(ATM)、现金记录器、条形码阅读机、电视机顶盒等。在这些情况下，用户界面与本地处理和通信功能相比较，在客户端软件中所占的比重很小。

除了用户界面和其他应用相关软件之外，客户端软件中还包含用于获得分布透明性的组件。理论上说，客户端不应该察觉到它与远程进程的通信。相反，对于服务器来说，出于性能和正确性方面的考虑，分布常常不那么透明。比如说，在第 6 章中我们将介绍，复制的服务器有时需要相互通信，以确定操作应该按照什么顺序在各服务器副本上执行。

访问透明性通常是由客户端存根处理的，该存根由服务器提供的接口定义生成。存

根提供的接口与服务器所提供的相同,但是前者隐藏了不同机器的体系结构上以及实际通信上可能存在的差异。

有几种不同的方法可以用来处理定位透明性、迁移透明性以及重定位透明性。关键是要使用一个便利的命名系统,这点我们将在下一章中介绍。很多时候与客户端软件的协作也是很重要的。比如说,如果客户已经绑定到服务器,改变服务器的位置时就可以直接通知它。这时可以由客户的中间件负责对用户隐藏服务器的当前位置,并且在必要的时候透明地重新绑定到服务器上。即使在最坏的情况下,也只不过是客户应用程序会察觉到暂时的性能下降而已。

与上面的方法相类似,许多分布式系统利用客户端解决方案来实现复制透明性。比如说,想象一下一个带有远程对象的分布式系统。可以通过将调用请求转发给每一个远程对象的副本达到复制透明性,如图 3.6 所示。客户代理将会透明地搜集所有对象的响应,并且只向客户应用程序送回一个返回值。

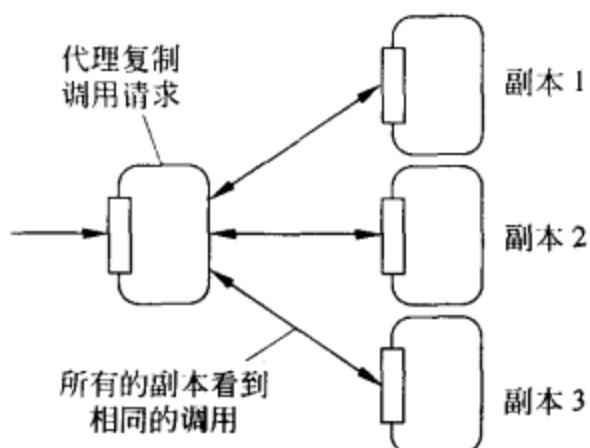


图 3.6 使用客户端解决方案来透明地复制远程对象的可行方法

最后,考虑一下故障透明性。对服务器通信故障的屏蔽一般是通过客户中间件完成的,比如说可以对客户中间件进行配置,使它不断尝试连接到某服务器,或者在进行几次尝试(并失败)后尝试连接到其他服务器。有时甚至可以让客户中间件返回它在前一次与该服务器的会话中缓存的内容,Web 浏览器在无法连接到某服务器时就是这样处理的。

并发透明性可以通过专门的中介服务器来实现,特别是可以由事务监视器来实现,而且需要客户软件的支持很少。同样,持久性透明性一般完全由服务器来处理。

3.3 服 务 器

现在我们来详细讨论服务器的组织结构。下面,我们首先重点讨论服务器设计上的几个常见的关键问题,随后讨论对象服务器。对象服务器很重要,因为它们是用来实现分布式对象的“积木块”。

3.3.1 设计上常见的主要问题

服务器是实现特定服务的进程,这些服务是为一组客户提供的。本质上,每个服务器的组织方式都一样:等待来自客户的请求,随后负责处理该请求,最后等待下一个输入的

请求。

服务器有几种不同组织结构。重复服务器(iterative server)自己处理请求，并且在必要的情况下将响应返回给发出请求的客户。并发服务器(concurrent server)并不自己处理请求，而是将请求传递给某个独立线程或者其他进程来处理，自身立即返回并等待下一个输入的请求。多线程服务器就是并发服务器的一个特例。并发服务器还有另一种实现形式，那就是每收到一个输入请求都派生出一个新进程来对其进行处理。许多 UNIX 系统采用了这种方法。在这种方法中，由处理请求的线程或者进程负责向发出该请求的客户返回响应。

另一个问题是客户联系服务器的地方。客户总是向服务器所在机器上的端点(endpoint)发送请求，端点也称作端口(port)。每个服务器都监听一个特定的端口。那么客户如何知道某个服务所对应的端点呢？一种方法是为公开的服务分配一个统一的端点。比如说，处理 Internet FTP 请求的服务器总是在 TCP 端口 21 上监听的。同样，万维网 HTTP 服务器总是在 TCP 端口 80 上监听。这些端口是由 Internet 号码分配管理局(IANA)来组织分配的，并且存档在(Reynolds 和 Postel 1994)中。由于已分配的端口是已知的，客户只需要找出运行服务器的机器的网络地址就可以了，而网络地址可以通过命名服务来取得，我们将在下一章中说明这一点。

有许多服务不需要预先分配好端点。比如说，一个昼间运行的服务器可能会使用由本地服务器动态分配给它的端点。在这种情况下，客户首先必须查询到该端点号。一种解决方法与 DCE 中使用的相同，就是在运行服务器的每台机器上都运行一个特殊的守护程序，该守护程序负责追踪由位于同一台机器上的服务器实现的每一项服务所使用的当前端点。守护程序是通过监听一个公开的端口来进行这项工作的。客户首先与该守护程序进行联系，请求得到指定服务器的端点号，随后再与该服务器进行联系，如图 3.7(a)所示。

通常把某个端点与特定的服务关联起来。然而，如果每一项服务都由单独的服务器来实现，无疑是对资源的一种浪费。比如说，在典型 UNIX 系统中，一般同时运行多个服务器，这些服务器大多被动地等待客户端的请求输入。与其对这么多被动进程进行追踪，不如由一个超级服务器(superserver)来负责监听所有与一个特定的服务关联的端点更有效率，如图 3.7(b)所示。UNIX 中的 inetd 守护程序就采用了这种方法。inetd 监听许多与 Internet 服务有关的公开端口。当收到请求的时候，它派生出一个进程以对该请求进行进一步处理，这个派生出的进程在处理完毕后将会自动退出运行。

在设计服务器时需要考虑的另一个问题是，应该在何时以何种方式来中断服务器的工作。比如说，考虑一下这种情况：用户刚刚决定向 FTP 服务器上载一个庞大的文件。突然该用户意识到选错了文件，想要中断服务器程序，取消数据传输。有几种方法可以做到这一点。有一种方法在当前的 Internet 环境下工作得非常好(有时这是惟一的可选方法)，就是让用户立即强行退出客户应用程序(这将自动中断与服务器之间的连接)，然后马上重新启动客户应用程序，就好像什么都没有发生过一样。服务器将拆除客户应用程序原来的连接，并且认为客户端发生了崩溃。

处理这类通信中断的更好的方法是在开发客户程序和服务器程序的时候考虑到对带外(out-of-band)数据发送的支持，服务器在处理客户发送的其他所有数据之前会优先处

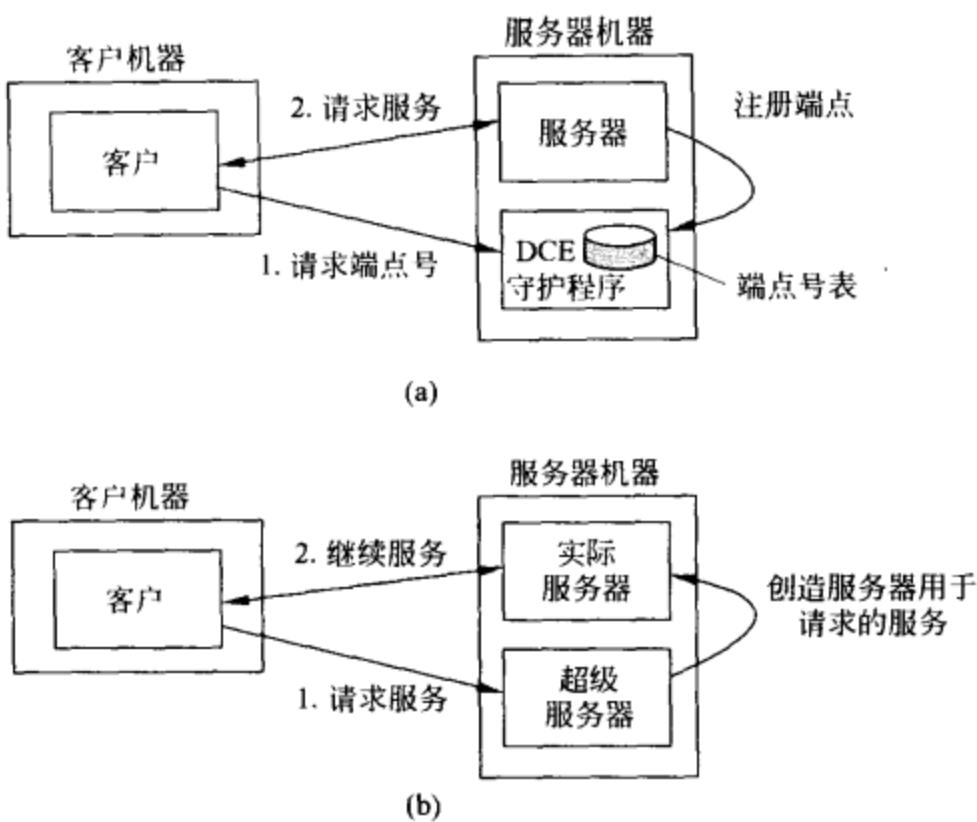


图 3.7 DCE 中和 UNIX 中的客户-服务器绑定

(a) 在 DCE 中,利用守护程序来进行客户-服务器绑定;

(b) 在 UNIX 中,使用超级服务器来进行客户-服务器绑定

理带外数据。一种解决方法是让客户将带外数据发送到独立的控制端点,服务器则监听该端点,同时也监听(以较低的优先级)常规数据传输的端点。另一种方法是通过发送原始请求所使用的连接来发送带外数据。比如在 TCP 中,传输紧急数据是可能的。当服务器接收到紧急数据时,服务器就被中断(比如通过 UNIX 系统中的信号机制),随后服务器可以检查紧急数据并且根据数据类型决定采取何种处理方式。

最后还有一个设计上的重要问题,就是服务器是否是状态无关的(stateless)。状态无关服务器(stateless server)不保存其客户的状态信息,而且也不将自身的变化告知任何客户(Birman 1996)。比如 Web 服务器就是状态无关的,它仅仅对输入的 HTTP 请求作出响应,这些请求包括向服务器上载文件或者更常见的从服务器获取文件,在处理请求的时候,Web 服务器彻底忽略了客户的存在。同样,Web 服务器所管理的一组文件(也许和文件服务器协作管理)的变化可以不必告知客户。

相反,状态相关服务器(stateful server)保存客户端的信息,并对其进行维护。这种服务器的典型例子是文件服务器,它让客户保留文件的本地拷贝,并且进行更新操作。文件服务器维护一张表,表中条目是(客户,文件)对,服务器可以利用该表来跟踪目前哪些客户拥有对哪些文件的更新许可,并且跟踪哪里的文件的版本是最新的。对于客户端来说,这种方法可以提升读写操作的性能。如果要提升状态无关服务器的性能,采用状态相关设计可以带来很大的帮助。然而,这个例子同时也体现了状态相关服务器的一个主要缺陷。即如果服务器崩溃,就必须恢复那些包含(客户端,文件)对条目的表,否则无法确保能够处理文件的最新版本。一般说来,状态相关服务器必须将自身的整个状态都恢复

到崩溃之前。我们将在第 7 章中讨论,要建立恢复机制将引入相当大的复杂性。如果采用状态无关设计,就不需要采取任何特殊措施来使崩溃的服务器恢复,只要简单地重新启动运行服务器,然后继续等待客户请求到达就行了。

在设计服务器的时候,无论选择采用状态无关设计还是状态相关设计都不应该对服务器所提供的服务造成影响。比如说,如果在读取文件或者写入文件之前必须打开该文件,那么状态无关服务器就必须用这样或那样的方法模拟这一打开文件的行为。通常的解决方法是,服务器响应对文件的读写请求时首先打开指定的文件,然后再执行实际的读写操作,随后立即关闭该文件。我们将在第 10 章中进行更为详细的讨论。

在某些情况下,服务器需要保留客户的活动记录,以便更加有效地响应客户的请求。比如说,Web 服务器有时可以立即将客户引导到该客户最常浏览的页面。只在服务器拥有关于该客户的历史信息的情况下,这种方法才能够实施。一种常用的方法是让客户在发送请求的同时发送关于前一次访问的额外信息。这种信息常常由客户端浏览器透明地存储在 cookie 中,cookie 是一小段数据,其中包含有对服务器有用的针对特定客户的信息。浏览器本身永远不会执行 cookie,它只对 cookie 进行存储。

在客户首次访问服务器时,服务器将会把 cookie 和所请求的 Web 页面一起发送给浏览器,随后浏览器将会安全地将 cookie 隐藏起来。随后客户每次访问该服务器的时候,关于该服务器的 cookie 将会与请求一起发送给服务器。虽然原则上来说这种方法可以有效工作,但是浏览器将 cookie 送回服务器存储这个事实是对用户完全隐藏的,这就会引发许多隐私保护方面的问题。与奶奶做的 cookie(此处意为小甜饼——译者)不同,浏览器生成的 cookie 最好放在原处不动为妙。

3.3.2 对象服务器

在讨论完设计中常见的一些重要问题之后,我们现在来考察一类渐趋重要的服务器——对象服务器(object server),这是一种为支持分布式对象而定制的服务器。一般的对象服务器与其他更为传统的服务器之间的区别在于,对象服务器本身并不真正提供特定的服务,特定的服务是由驻留在服务器中的对象实现的。本质上来说,服务器仅仅提供根据远程客户的请求来调用本地对象的手段。因此,要改变所提供的服务相对来说比较简单,只要添加并删除某些对象就行了。

对象服务器用来作为对象居留的场所。对象由两部分构成:一个部分是代表自身状态的数据,另一个部分是构成其方法实现的代码。这两个部分是否互相分离,以及方法实现是否由多个对象共享等问题取决于对象服务器。同时,不同对象服务器调用其对象的方法也存在不同。比如说,在多线程服务器中,每个对象都分配给一个单独的线程,或者对每一个调用请求都使用一个单独线程。下面将讨论以上问题以及其他问题。

1. 调用对象的不同方法

如果要调用某个对象,对象服务器必须了解应该执行哪些代码,应该对哪些数据进行操作,是否应该启动一个单独的线程来对调用进行管理,等等。一种简单的方法是假定所有对象都相似,并且只存在一种调用对象的方法。DCE 在本质上采取的就是这种方式。不幸的是,这种方法一般欠缺灵活性,并且常常对分布式对象的开发人员造成不必要的约束。

更好的方法是让服务器支持不同策略。比如说，考虑一下暂时对象。回忆一下，暂时对象的生存时间至多不会超过其服务器的生存时间，还可能更短。文件在内存中的只读拷贝可以通过暂时对象实现。同样，计算器（可能运行在高性能服务器上）也可以作为暂时对象来实现。一种合理的策略是，在收到第一个调用请求时创建一个暂时对象，随后如果不再有客户绑定到该对象上就尽快将其销毁。这种方法的优点在于，暂时对象只在它真正有用时才需要占用服务器的资源。而该方法的缺陷在于，因为需要首先建立对象，所以调用可能需要花费一段时间才能完成。因此，另一种改进方法是有时在服务器初始化的时候就一次性创建全部暂时对象，但是这么做的代价是，某个对象即使不再被客户使用，它仍然要消耗资源。

另一种相近的方式是，服务器可以采取另一种策略，就是将每个对象都放置在属于该对象自己的内存段中。也就是说，对象间不共享数据，也不共享代码。如果对象的实现并不将代码和数据相分隔开来，或者出于安全上的原因必须将对象相互隔离，那么这种策略就是必需的。如果是出于安全原因，服务器还必须提供特殊措施，或者请求到底层操作系统的支持，以确保访问不会超越段边界。还有一种方法可以让对象共享代码。比如说，如果数据库中含有的对象都是属于同一个类的，就可以通过将类实现一次性加载到服务器上来实现该数据库。这样，当有对象调用的请求输入时，服务器只需从数据库中取得对象状态，随后执行请求的方法就行了。

同样，关于线程也有多种不同策略。最简单的方法是实现单控制线程的服务器。服务器也可以拥有多个线程，每一个线程管理一个对象。只要有对象的调用请求输入，服务器将会把请求传递给负责管理该对象的线程。如果该线程当前正忙，请求就会暂时存放在队列中。这种方法的好处在于，对象可得到自动的保护以免受到并发的访问，即所有对该对象的调用都由与该对象相关联的线程来串行处理。当然，也可以为每一个调用请求都使用一个单独的线程，但这就要求采取措施来保护对象使其免遭并发访问。是根据需要来创建线程还是由服务器来维护一组线程并不依赖于每个对象使用一个线程还是每个方法使用一个线程。一般来说，并没有一个万能的最佳策略。

2. 对象适配器

对如何调用对象所做的决定一般称作激活策略（activation policies），这么称呼它是为了强调：在许多情况下，在实际调用对象之前，必须先将对象本身加载到服务器地址空间（也就是激活该对象）。

随后需要一种机制来将对象根据策略进行分组。这种机制有时称作对象适配器（object adapter）或者对象包装器（object wrapper），而许多用于构建对象服务器的工具都经常将它隐藏起来。在这里我们使用对象适配器这个术语。最好能将对象适配器想象成实现特定激活策略的软件。然而，公认的说法是，对象适配器是用来协助分布式对象开发人员的普通组件，只需要根据特定的策略来加以配置。

对象适配器控制一个或者多个对象。由于服务器应该能够为请求不同激活策略的对象同时提供支持，其中某些适配器还可能同时驻留在同一个服务器上。当一个调用请求传送到服务器时，该请求首先被分派到适合的对象适配器，如图 3.8 所示。

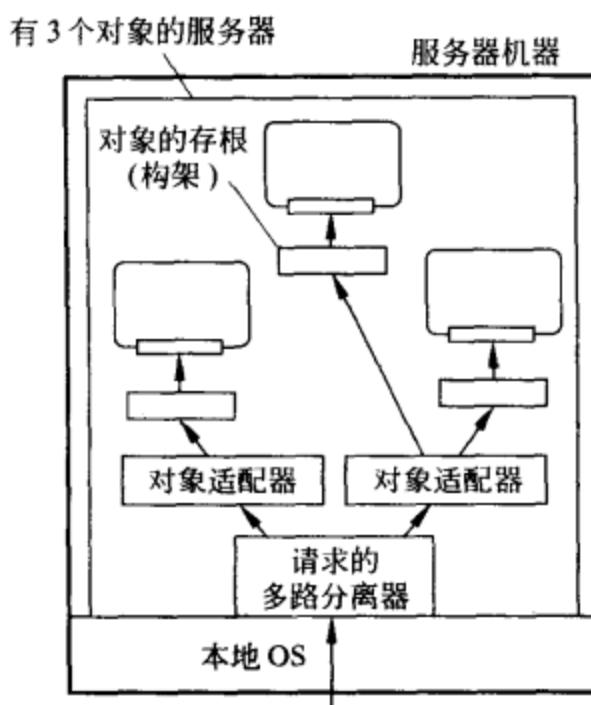


图 3.8 支持多种激活策略的对象服务器的组织结构

重要的是,对象适配器对受其控制的对象的接口一无所知,否则,对象适配器就不是通用的。对于对象适配器来说,惟一件重要的事就是它可以从调用请求中得到指向对象的引用,随后将请求分派给引用所指向的对象,但是首先需要遵循特定的激活策略。如图 3.8 所示,对象适配器不是将请求直接传递给对象,而是将调用请求交给该对象所在的服务器端存根。该存根也称作骨架(skeleton),它一般是利用该对象的接口定义生成的,负责对请求进行解除编组并且调用相应的方法。

作为一个例子,考虑一下一个管理多个对象的对象适配器的情况。适配器实现的策略是为每个对象分派单个控制进程来管理。为了与负责编组和解编请求的特定对象的骨架进行交互,它希望由每个骨架来实现以下操作:

```
invoke(unsigned in_size, char in_args[], unsigned * out_size, char * out_args[])
```

其中的 `in_args` 是一个字节数组,它需要由存根来解编组。该数组中包含有方法的标识以及该方法使用的所有参数值。只有存根知道该数组的确切格式,存根同时也负责执行实际的调用操作。参数 `in_size` 指定了 `in_args` 的长度。所有的输入都以相似的方式由存根编组进数组 `out_args`,该数组是由存根动态创建的。该数组长度由输出参数 `out_size` 指明(注意,这个 `invoke` 语句与第 2 章中讨论过的版本相近,但是后者是用来进行动态调用的)。

图 3.9 展示了适配器的头文件。头文件中最重要的一部分是适配器与远程客户交换的消息的定义。每个客户都希望将调用请求编组进一个含有 5 个字段的消息中去。同样,适配器将会返回一个结构相同的消息进行响应。`source` 字段标识了消息的发送端。`object_id` 和 `method_id` 字段分别唯一地标识了要调用的对象及方法。要传递给存根的输入数据包含在 `data` 数组中,`data` 数组的实际长度由 `size` 字段给出。随后调用结果被放入新消息的 `data` 字段中去。

```

/* 适配器及其调用者需要的定义 */
#define TRUE    1
#define MAX_DATA 65536

/* 一般消息格式的定义 */
struct message {
    long      source;          /* 发送的标识 */
    long      object_id;       /* 所请求对象的标识符 */
    long      method_id;       /* 所请求方法的标识符 */
    unsigned size;            /* 参数列表的总字节数 */
    char     * data;           /* 字节顺序参数 */
}

/* 对象构架中调用操作的一般定义 */
typedef void (*METHOD_CALL)(unsigned, char *, unsigned *, char **);

long register_object(METHOD_CALL call);        /* 注册对象 */
void unregister_object(long object_id);         /* 未注册对象 */
void invoke_adapter(message * request);         /* 调用适配器 */

```

图 3.9 适配器以及调用该适配器的程序使用的 header.h 文件

头文件中还包含有一些定义,定义了适配器希望能够在对象的服务器端存根中通过 METHOD_CALL 类型定义执行的调用。

最后,适配器还提供两个过程,服务器可以调用它们在适配器中注册或者注销对象。可以通过向 invoke 过程的针对特定对象的实现传递指针来进行注册,对象存根采用的就是这种方法。注册将会返回一个号码,该号码可以供适配器用作对象标识符。如果要注销一个对象,服务器只要调用 unregister_object 过程并且向其传递要注销的对象号码即可。对适配器真正的调用是通过 invoke_adapter 过程完成的,该过程需要对象标识符和调用请求。调用结果放入单独的缓冲区中,我们将随后进行进一步说明。

为了实现适配器,我们假定已经有了线程包,其中包含有创建(并删除)线程以及线程通信等必需功能。线程间通信是利用缓冲区来完成的。特别是,每个线程都拥有与之相关联的缓冲区,线程可以利用阻塞性操作 get_msg 从缓冲区中移出消息。使用非阻塞性操作 put_msg 将消息添加到缓冲区末尾。线程包头文件的主要部分列出在图 3.10 中。

```

typedef struct thread THREAD;      /* 被隐藏的线程定义 */

THREAD * create_thread(void (* body)(long tid), long threadId);
/* 通过给出一个指向定义线程实际行为的函数的指针,以及一个用来惟一定义这个线程的 */
/* 整数来创建一个线程 */

void get_msg(unsigned * size, char ** data);
void put_msg(THREAD * receiver, unsigned size, char * data);
/* 在一个消息被放入与之相关联的缓冲区前,调用 get_msg 会阻塞线程。将消息放入线程的 */
/* 缓冲区是非阻塞性操作 */

```

图 3.10 使用多线程的适配器所使用的 thread.h 文件

现在该讨论适配器的实际实现了,如图 3.11 所示。每个对象都拥有与自身相关联的线程,这些线程是由过程 thread_per_object 指定的。线程开始是阻塞的,直到有调用请求放入属于它的缓冲区时才启动。使用恰当的参数来调用 invoke[object_id] 可以将请求立即传递给对象存根。对象调用的结果放在变量 result 中返回,随后结果复制进响应消息中去。通过先填充响应消息中的 object_id 和 method_id 字段,随后再将结果复制到消息的 data 字段中去,响应消息就建立起来了。在这时,响应可以交由多路分离器处理,如图 3.8 所示。在我们的这个例子中,多路分离器可以通过一个单独的线程实现,这个线程通常由变量 root 来代表。

```
# include <header.h>
# include <thread.h>
# define MAX_OBJECTS    100
# define NULL          0
# define ANY           -1

METHOD_CALL invoke[MAX_OBJECTS];      /* 存根指针数组 */
THREAD * root;                      /* 多路分离器线程 */
THREAD * thread[MAX_OBJECTS];        /* 每个对象一个线程 */

void thread_per_object(long object_id) {
    message      * req, * res;          /* 请求/响应消息 */
    unsigned      size;                /* 消息大小 */
    char       * results;             /* 所有结果数组 */

    while(TRUE) {
        get_msg(&size, (char *) &req);   /* 阻塞调用请求 */
        /* 将请求传送给适当的存根,这个存根被假设为存储结果分配存储器 */
        (invoke[object_id])(req->size, req->data, &size, &results);

        res = malloc(sizeof(message)+size); /* 创建响应消息 */
        res->object_id = object_id;        /* 标识对象 */
        res->method_id = req.method_id;    /* 标识方法 */
        res->size = size;                /* 设置调用结果大小 */
        memcpy(res->data, results, size); /* 将结果复制到响应中 */
        put_msg(root, sizeof(res), res);  /* 将响应添加到缓冲区 */
        free(req);                     /* 释放请求所用的存储器 */
        free(results);                 /* 释放结果所用的存储器 */
    }
}

void invoke_adapter(long oid, message * request) {
    put_msg(thread[oid], sizeof(request), request);
}
```

图 3.11 实现每个对象一个线程策略的适配器的主要部分

`invoke_adapter` 的实现现在很简单。执行调用的线程(也就是例子中的多路分离器)将它的调用请求添加在与被请求调用的对象相关的线程的缓冲区中。随后,多路分离器可以从自己的缓冲区中获取结果,然后将结果返回给原先发出调用请求的客户。

重要的是要注意到,适配器的实现是独立于它为之处理调用的对象的。事实上,在示例的实现中没有包含针对特定对象的代码。正因为这样,才有可能构建通用对象适配器并且在概念上把这些适配器放在中间件层。对象服务器的开发人员只要把精力集中放在对象开发上,并简单地指定对象的调用应该由哪个适配器来控制就行了。

最后要说一下,虽然在图 3.8 中我们展示了一个独立的多路分离器组件,它是负责将输入的调用请求分派给恰当的对象服务器的,但是实际上并不需要这种多路分离器,可以使用对象适配器来达到同样的目的。CORBA 中就采用了后者,我们将在第 9 章中对其进行讨论。

3.4 代码迁移

到此为止,我们所涉及的分布式系统所进行的通信仅限于传递数据。然而,在有些情况下需要传递程序(甚至可能需要传递正在执行中的程序),以简化分布式系统设计。在本节中,我们将详细考察代码迁移的实质。我们首先考察进行代码迁移的几种不同方案,随后讨论如何处理要迁移的程序所使用的本地资源。还要讨论一个特别难以解决的问题——异构系统中的代码迁移。为了说得更加具体一些,我们将在本节末尾讨论移动代理使用的 D'Agents 系统。关于代码迁移的安全性方面的问题将在第 8 章中讨论。

3.4.1 代码迁移方案

在考察代码迁移的各种不同形式之前,让我们首先考虑一下代码迁移究竟有什么用处。

1. 进行代码迁移的理由

传统上,分布式系统中的代码迁移是以进程迁移(process migration)的形式进行的,在这种形式下整个进程被从一台机器搬到另一台机器上去。将正在运行的进程移到另一台机器上去是一项复杂并且开销庞大的任务,所以要这么做必须有充分的理由。理由当然永远是性能上的考虑。基本的思想是:如果把进程由负载较重的机器上转移到负载较轻的机器上去,就可以提升系统的整体性能。负载常常是用 CPU 队列长度或者 CPU 利用率来衡量的,也可以使用其他性能指标衡量。

制定决策所使用的是负载分布算法,还要考虑到任务在一组处理器上的分配和重新分布。这种算法对于计算任务繁重的系统尤其有用。然而,在许多现代分布式系统中,对计算能力的优化不如通信量的减少重要。此外,由于底层平台和计算机网络的异构性,代码迁移所能够带来的性能提升常常是基于定性推导得出的,而不是通过数学模型计算出来的。

举个例子,考虑一下一个客户-服务器系统,其中由服务器来管理一个巨型数据库。

如果客户应用程序需要执行大量的涉及大量数据的数据库操作,最好先将客户应用程序的一部分搬运到服务器上运行,只通过网络发送结果。如果不这样做,服务器与客户之间传输的大量数据就会挤垮网络。在这种情况下,之所以要进行代码迁移是基于这样一个假定,那就是处理数据的地点离数据所在地点越近越好。

将服务器的一部分迁移到客户端的原因也是一样的。比如说,在许多交互式数据库应用程序中,客户必须填写表单,随后表单被转换成一系列数据库操作。比较好的做法是先在客户端处理表单,然后再把处理好的表单发送给服务器,这种做法可以避免在网络上上传输大量的小型消息。这样客户就可以获得更好的性能,而同时服务器花费在处理和通信上的时间也会更少。

对于代码迁移的支持,还可以通过并行的使用来提升性能,而不会带来通常的并行编程所具有的复杂性。典型的例子是 Web 信息的搜索。以一种便于在站点之间移动的小移动程序的形式来实现这种搜索查询会相对简单。可以制作这种程序的多份拷贝,并且将每一份拷贝发送给不同的站点。与只使用单个程序实例相比较,这样做可以使速度得到线性提升。

除了可以提升性能之外,支持代码迁移还有其他方面的原因,其中最重要的是灵活性。传统的构建分布式应用程序的方法是将应用程序分为不同部分,预先确定每一个部分应该在何处执行。依照这种方案执行将会产生第 1 章中讨论过的各种多层次客户-服务器应用程序。

然而,如果代码可以在不同机器之间移动,就可以动态地配置分布式系统。比如说,假定某个服务器实现了访问某个文件服务器的标准化接口。为了让远程客户访问文件服务器,服务器必须使用专门的协议。一般说来,基于这种专门协议的文件系统接口的客户端实现需要与客户端应用程序链接起来。这种方案要求在开发客户应用程序时就必须有可用的协议软件。

另一种方法是只在确实必要的时候让服务器提供客户的实现,也就是让客户绑定到服务器上。这种情况下,客户动态地下载该实现,经过某些必需的初始化步骤后调用该服务器。其原理如图 3.12 所示。这种模型将代码由远程站点动态移来,它要求有标准化的下载协议和初始化代码,同时还要求下载的代码能够在客户所在机器上执行。后面将在

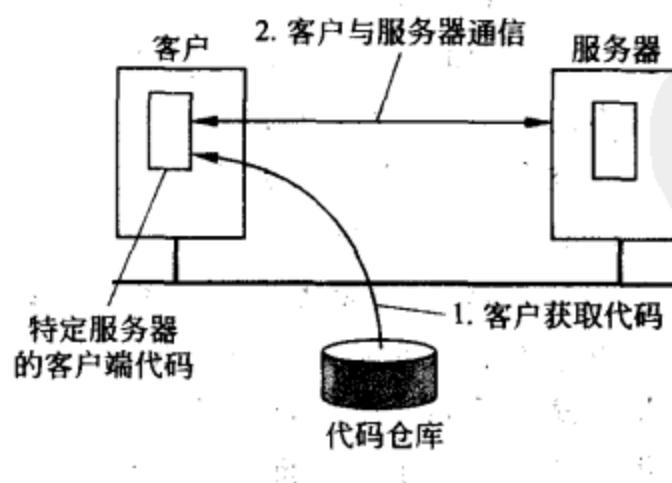


图 3.12 对客户进行动态配置使其能够与服务器通信的原理。客户首先获取必需的软件,随后调用服务器

不同章节中讨论不同的解决方法。

这种动态下载客户端软件的显著优点在于,客户不需要预先安装与服务器通话所需的所有软件。可以在必要的时候将软件移来,同样也可以在不再需要该软件的时候将其丢弃。另一个优点是,只要接口是标准化的,就可以不受限制地改变客户-服务器协议及其实现,所做的改动并不会影响到现有的依赖于服务器的客户应用程序。当然它也存在缺点,最严重的缺点我们将在第 8 章中进行讨论,那就是安全性方面的问题。盲目地信任下载的代码只是实现了声称的接口是极为危险的,它可能会访问您的毫无防备的硬盘,天知道它会将其中最敏感的信息发给什么人。

2. 代码迁移模型

虽然代码迁移所说的只是在机器间移动代码,但是这个名词中包含的思想其实要丰富得多。传统上,分布式系统中的通信是关于进程间数据交换的。最广义的代码迁移可以涉及在机器间移动程序,目的在于在目标机器上运行该程序。在某些情况下,与进程迁移相同,必须同时移动程序执行状态、未处理信号以及其他部分环境内容。

为了更好地理解代码迁移的不同模型,我们采用文献(Fugetta 等 1998)中提出的框架结构。在该框架结构中,进程包含 3 段。代码段(code segment)部分包含构成正在运行的程序的所有指令。资源段(resource segment)包含指向进程需要的外部资源的指针,这些外部资源包括文件、打印机、设备、其他进程等。最后是执行段(execution segment),它用来存储进程的当前执行状态量,这些状态量包括私有数据、栈和程序计数器等。

代码迁移的最弱形式仅仅提供弱可移动性(weak mobility)。在这种模型中,可以只传输代码段以及某些初始化数据。弱可移动性的典型特征是,传输过来的程序总是以初始状态重新开始执行的。Java 小程序(applet)就是这种情况。这种方法的好处在于其简单性。弱可移动性只要求目标机器能够执行代码,这在本质上可以归结为使代码可移植。我们将在讨论异构系统中的迁移时继续讨论这些内容。

与弱可移动性相反,在支持强可移动性(strong mobility)的系统中,还可以传输执行段。强可移动性的典型特征是,可以先停止运行中的进程,然后将它搬到另一台机器上去,再从刚才中断的位置继续执行。很明显,强可移动性要比弱可移动性强大得多,但是也更加难以实现。D'Agents 是支持强可移动性的系统的一个例子,我们将在本节中对其进行讨论。

可以通过是由发送者启动迁移还是接收者启动迁移来对迁移作出进一步区分,这与可移动性的强弱无关。在发送者启动(sender-initiated)的迁移中,代码当前驻留在哪台机器上或者正在哪台机器上执行,就由该机器来启动迁移。一般来说,在向计算服务器上载程序时进行的就是发送者启动的迁移。通过 Internet 向 Web 数据库服务器发送搜索程序以在该服务器上进行查询,也是发送者启动迁移的一个例子。在接收者启动(receiver-initiated)的迁移中,代码迁移的主动权掌握在目标机器手中。Java 小程序是这种迁移的一个例子。

接收者启动的迁移一般比发送者启动的迁移要更容易实现。在许多情况下,在客户和服务器之间进行的代码迁移的主动权由客户端掌握。如果要像发送者启动的迁移那样

安全地向服务器上载代码，常常要求客户端预先在服务器上注册并通过验证。也就是说，要求服务器了解它的所有客户，这是因为客户很可能会访问服务器的资源，比如磁盘，而对这些资源的保护是极为重要的。相反，如果是接收者启动的迁移，代码下载工作一般可以匿名进行。此外，服务器一般并不对客户资源感兴趣，而向客户端进行代码迁移只是为了提升客户端的性能。实际上，客户端只有有限的少数资源需要保护，比如内存和网络连接等。我们将在第 8 章中继续详细讨论代码迁移的安全问题。

在弱可移动性的情况下，迁移过去的代码是否由目标进程执行，还是重新启动一个单独的进程来执行，这方面也存在区别。比如说，Java 小程序可以简单地由 Web 浏览器下载之后在浏览器的地址空间中执行。这种方法的好处是不需要启动单独的线程，也就避免了与目标机器进行通信，而主要缺陷是目标进程必须得到保护，以免在迁移来的代码的执行过程中受到有意或者无意的侵害。一种简单的解决方法是，由操作系统来进行管理，创建一个单独的进程来执行迁移来的代码。要注意，这种解决方法并没有解决刚才指出的资源访问方面的问题。

如果不想直接移动运行中的进程，也就是说不进行进程迁移，那么可以通过远程克隆 (remote cloning) 的方式来支持强可移动性。与进程迁移不同，克隆将制作一份与原始进程完全相同的拷贝，但运行在不同的机器上。在 UNIX 系统中，远程克隆的发生在派生出一个子进程并让它在远程机器上继续运行的情况下。克隆的优点是克隆得到的进程与原来已经在多个应用程序中使用的进程极为相似，惟一的不同点在于，克隆得到的进程是在另一台机器上运行。从这个意义上说，通过克隆来迁移是一种增强分布透明性的简单方法。

代码迁移的各种不同方法如图 3.13 所示。

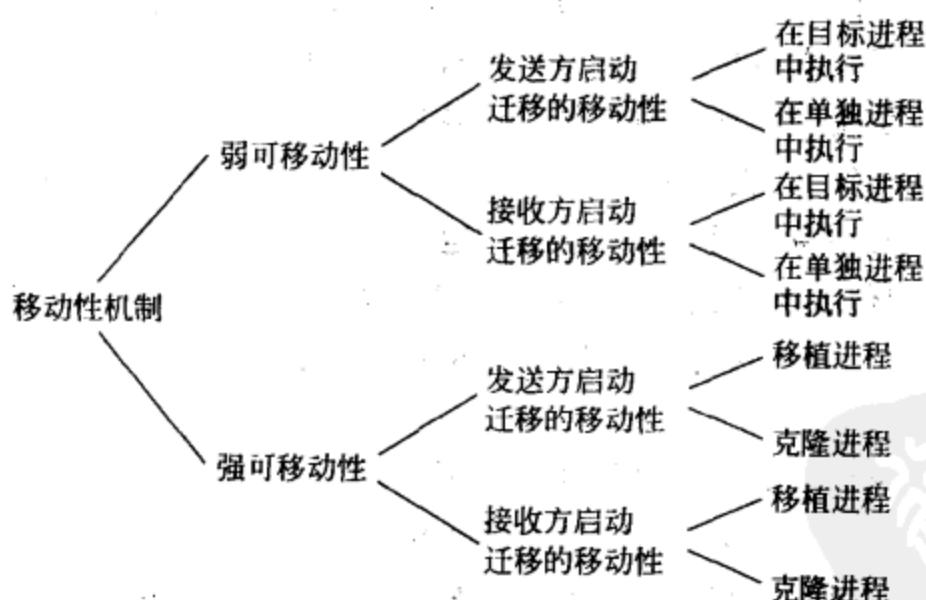


图 3.13 代码迁移的各种不同方法

3.4.2 迁移与本地资源

到此为止，我们只讨论了代码段和执行段的迁移。资源段也应该受到特别的关注。代码迁移之所以如此困难，就是因为有时资源段在不作改动的情况下难以与其他段一同

迁移。比如说,如果有一个进程中保存指向某个 TCP 端口的引用,通过该端口它可以与其他远端进程通信。该引用保存在进程的资源段中。当进程迁移到另一个地方时,它必须放弃使用该端口,然后在迁移到的地方请求一个新端口。而在别的情况下转移一个引用是不成问题的。比如说,如果一个进程中保存有以绝对 URL 的形式表示的文件引用,该进程无论驻留在哪一台机器上,该文件引用都有效。

为了理解代码迁移与资源段的关系,Fuggetta 等人将进程对资源的绑定方式分为三类。最强的绑定方式是进程使用资源的标识符来引用资源。在这种情况下,进程明确指定的只是要引用哪些资源。这种绑定称作按标识符绑定(binding by identifier),举个例子,进程根据服务器所在的 Internet 地址使用 URL 来引用特定 Web 站点或者 FTP 服务器时就使用这种绑定。同样的,使用指向本地通信终点的引用也会导致采用这种绑定方式。

进程对资源的绑定的一种较弱方式是,只使用资源的值。这种情况下,只要另一个资源能够提供一样的值,进程的执行就不会受到影响。依赖标准库的程序,比如那些使用 C 语言或者 Java 编程的程序都是这种按值绑定(binding by value)的例子。本地必须保证有这些库可用,但是这些库在本地文件系统中的确切位置依据站点不同可能会不同。为了保证进程能够正确执行,重要的不是用到了哪些文件,而是文件中的内容要正确。

进程对资源的绑定的最弱方式是,进程只指明它需要哪一种类型的资源。这种绑定叫做按类型绑定(binding by type),对诸如监视器、打印机等本地设备的引用都是按类型绑定的例子。

在迁移代码的时候,我们常常需要在不影响进程对资源的绑定的条件下改变指向资源的引用。改变引用的具体方式取决于资源能否与代码一起迁移到目标机器上去。更明确地说,需要对资源对机器的绑定方式进行考察,并区分为下列几种情况进行考虑。未连接资源(unattached resources)可以方便地在机器之间移动,这种资源一般是与要迁移的程序相关的数据文件。而附着连接资源(fastened resource)也可以进行复制或者移动,但是开销相对来说比较大。附着连接资源的例子包括本地数据库以及完整的 Web 站点。虽然这种资源在理论上并不依赖于当前的机器,但是将它们移动到另外的环境中去的做法通常并不可行。最后还有一种紧固连接资源(fixed resources),这种资源紧密地绑定到特定的机器或者环境,无法进行移动。本地设备一般是紧固连接资源。本地通信终点是另一个紧固连接资源的例子。

如果将三种类型的进程对资源的绑定与三种类型的资源对机器的绑定结合起来,就会产生 9 种组合形式,这些形式可供我们在考虑代码迁移时采用。在图 3.14 中列出了这些组合形式。

我们首先考虑一下进程能不能利用资源标识符来绑定到资源上。如果一个资源是未连接的,那么最好将它和要迁移的代码放在一起移动。然而,如果有其他进程共享该资源,就必须建立关于该资源的全局引用,即超越机器界限在全系统中都适用的引用。URL 是这种引用的一个例子。如果资源是附着的或者是紧固的,最好的解决方法还是建立全局引用。

		资源对机器绑定		
		未连接	附着连接	紧固连接
进程对资源绑定	按标识符	MV(或 GR)	GR(或 MV)	GR
	按值	CP(或 MV,GR)	GR(或 CP)	GR
	按类型	GB(或 MV,CP)	RB(或 GR,CP)	RB(或 GR)

GR 建立一个全局系统范围内引用
MV 移动资源
CP 复制资源的值
RB 将进程重新绑定到本地可用资源上

图 3.14 在向另一台机器迁移代码时,根据引用本地资源方式的不同所应该采取的不同做法

意识到以下这一点是很重要的:要建立全局引用并不仅仅是使用 URL,而且使用这种引用的开销有时昂贵得无法接受。举个例子,考虑一下某个供多媒体工作站使用的生成高画质图像的专用程序。实时制作高画质图像是一项非常繁重的计算任务,所以程序也许需要移动到高性能计算服务器上去执行。建立多媒体工作站的全局引用意味着要在计算服务器和工作站之间建立连接通道。另外,为了满足图像传输的带宽需求,服务器和工作站还必须进行一些重要的处理工作。结果是将程序移动到计算服务器上的效果并不好,这主要是因为使用全局引用的开销太高了。

在以下这个例子中,建立全局引用也不是很容易的。如果要迁移的是使用本地通信终点的进程,就必须涉及对紧固连接资源的处理,进程是使用标识符绑定到该资源上的。这可以有两种解决方法。一种方法是,在进程迁移之后,在进程所在源机器上建立一个单独的进程并由迁移后的进程建立连往源机器的连接,由单独的进程负责将所有传入消息通过连接转发给迁移后的进程。这种方法的主要缺点在于,无论何时只要源机器出错,与迁移后的进程的通信也就会失败。另一种方法是,让与迁移进程通信的所有进程都改变它们的全局引用,将消息都发送到目标机器上的新的通信终点。

如果是按值绑定,情况就不同了。先考虑紧固连接资源的情况。比如说,如果是按值绑定的紧固连接资源,假定进程允许内存由多个进程共享。在这种情况下,建立全局引用意味着需要实现第 1 章中讨论过的分布式共享存储器机制。很明显,这种解决方法并不真正可行。

运行时库是典型的通过值引用的附着连接资源。一般说来,这种资源的副本在目标机器上是很容易得到的,也可以先将副本复制到机器上再进行代码迁移。如果要复制的数据的量非常大,比如要复制文本处理系统中的字典或者词典,那么更好的做法是建立全局引用。

如果要处理的是未连接资源,情况最简单。只要没有多个进程共享该资源,最好的做法就是把资源复制(或者移动)到新地点。如果该资源被共享,那就只好建立全局引用了。

最后,还有按类型绑定。无论使用的资源对机器的绑定方式是何种类型,解决方法都是显而易见的,就是将进程重新绑定到本地可用的同一类型的资源上去。只在本地没有同一类型的资源的情况下,才需要将原来的资源复制到新位置,或者建立全局引用。

3.4.3 异构系统中的代码迁移

到此为止,我们都默认这样一个假定,那就是迁移后的代码能够在目标机器上很好地执行。如果是同构系统,这个假定是成立的。然而分布式系统通常是建立在一组异构平台的基础上的,这些平台各自的操作系统和机器体系结构都不同。在这种系统内的迁移要求每一个平台得到支持,也就是说代码段必须能够在所有平台上执行(也可能要对原始的代码重新编译)。同时,必须确保在所有平台上执行段都能够正确表示。

如果只涉及弱可移动性,问题还不那么严重。在这种情况下,基本不需要在机器间传输运行时信息,所以可以对源代码进行重编译,生成不同的代码段,分别用于不同的目标平台。

如果要支持强可移动性,需要解决的主要问题在于执行段的传输。问题是执行段是高度依赖于执行进程的平台的。事实上,只有在目标机器的体系结构与源机器相同,并且运行的是完全一样的操作系统的情况下,才可能在不作改动的情况下直接迁移执行段。

执行段包含的数据是进程私有的,包括当前堆栈以及程序计数器等。临时数据占用了一部分堆栈,比如局部变量值,但是堆栈中也含有诸如寄存器值等平台相关的信息。从这里得到的一点重要启发是,如果执行是不依赖于针对特定平台的数据的,就能够更加方便将执行段转移到不同机器后再恢复执行。

一种针对诸如 C 或者 Java 之类的过程性语言的解决方法如图 3.15 所示,它的工作原理如下:只在程序执行过程中特定的执行点允许进行代码迁移。特别是,只在调用下一个子例程时才允许进行迁移。C 中的子例程就是函数,而 Java 中的子例程则是方法,别的语言中可能还有别的称呼。运行时系统负责维护自身的程序栈拷贝,但是这种维护是独立于机器的。我们将这种拷贝称为迁移栈(migration stack)。在调用子例程时或者执行权从子例程返回时对迁移栈进行更新。

在调用子例程时,运行时系统将对自上一次调用以来压入堆栈中的数据进行编组。这些数据代表的是局部变量的值以及新调用的子例程的参数值。随后将编组后的数据以及调用的子例程的标识符压入迁移栈。另外,当调用者由子例程中返回后,继续执行的起始位置也以跳转标号的形式压入迁移栈中。

如果在调用子例程的执行点上发生代码迁移,运行时系统首先将所有针对特定程序的全局数据(这些数据形成了执行段的某些部分)都进行编组。针对特定机器的数据和当前堆栈都被忽略。编组后的数据和迁移栈一同传输到目的机器。此外,目的机器还要根据自己机器的体系结构和操作系统加载代码段。属于执行段的编组数据被解除编组,并对迁移栈解除编组以建立新的运行时栈。随后只要进入在原先机器上调用的子例程中就可以恢复执行了。

很明显,只有在编译器进入或者退出子例程时都生成了用于更新迁移栈的代码的情况下,这种方法才是可行的。编译器还必须在调用者代码中生成标号,以保证从子例程中的返回能够像与机器无关的跳转一样实现。另外,还需要合适的运行时系统。不过,有多种系统已经成功地应用了这些技术。比如说,参考文献(Dimitrov 和 Rego 1998)就演示了如何通过仅对语言本身作少量改动,并使用预处理器插入维护迁移栈的必要代码来支

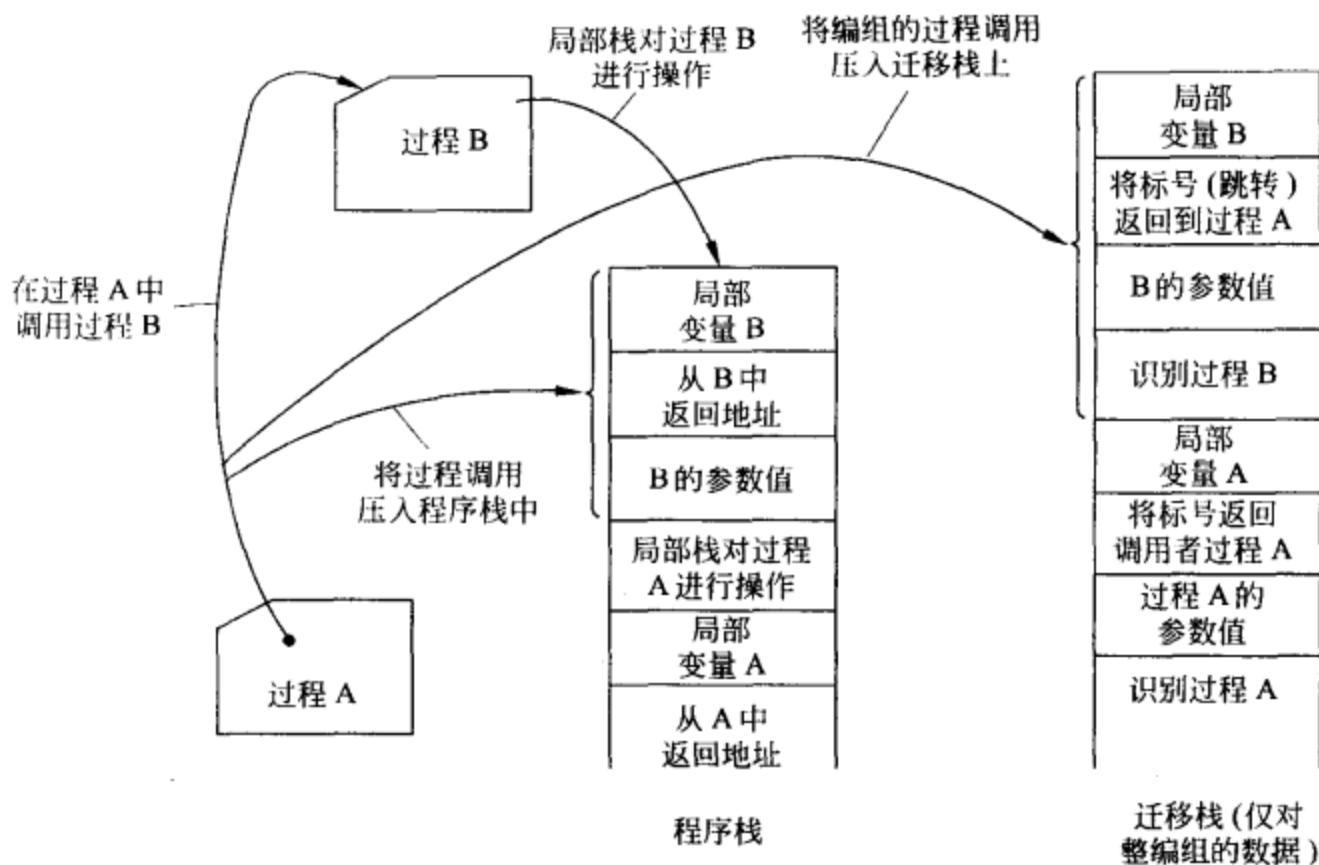


图 3.15 维护迁移栈以支持异构环境下的执行段迁移的原理

持异构系统中的 C/C++ 程序的迁移。

异构性所导致的问题在很多方面与可移植性相同,解决方法也非常相近,对此并不应该感到惊奇。比如说,在 19 世纪 70 年代末,产生了一种可以部分解决 Pascal 在不同机器间移植问题的简单方法,就是生成用于抽象虚拟机的与机器无关的中间代码(Barren 1981)。当然需要在多种平台上实现该虚拟机,然后就可以允许 Pascal 程序在各处运行。虽然这种简单的思想已经沿用好多年了,但它从未真正成为解决其他语言(特别是 C 语言)的移植性问题的通用流行方法。

大约 20 年以后,以上这种异构系统中代码迁移的解决方法受到脚本语言及诸如 Java 之类的高度可移植语言的冲击。这类解决方法的共同点是它们都依赖于虚拟机,该虚拟机要么直接解释源代码(比如脚本语言就是这样),要么就解释由编译器生成的中间代码(Java 就是这样)。可见对于语言开发者来说,选择正确的时间和在正确的领域进行开发也是很重要的。

虚拟机方法的惟一严重不足在于,我们常常将其与指定的语言联系在一起,而这种语言在过去并没有使用过。由于这个原因,为了让某些语言具有可迁移性,必须让它们向现有的语言提供接口,这是很重要的。

3.4.4 实例: D'Agents

为了更好地说明代码迁移,我们先来看一下支持多种形式的代码迁移的一种中间件平台——D'Agents。D'Agents 的正式名称是 Agent Tel,它是一种建立在代理的概念上的系统。D'Agents 中的代理是一个程序,该程序可以在异构系统中的机器间迁移。这里

我们只讨论 D'Agents 系统中迁移方面的功能,在下一节中再详细讨论软件代理。同时,在这里我们还将忽略系统中的安全方面的内容,把相关的讨论放到第 8 章中去。如果要了解关于 D'Agents 的更多信息,请参阅文献(Gray 1996b)和(Kotz 等 1997)。

1. D'Agents 中的代码迁移概述

D'Agents 中的代理是一个程序,该程序可以在异构系统中的机器间迁移。原则上说,可以用多种语言来编写程序,只要目标机器能够执行迁移过来的代码就行。这在实践中就意味着 D'Agents 中的程序要用可解释型语言,特别是工具命令语言(Tool Comm and Language),也就是 Tel(Ousterhout 1994)、Java 和 Scheme(Rees 和 Clinger 1986)。只使用可解释型语言使得对异构系统的支持变得更加方便。

程序(或者称作代理)是由运行语言解释器的进程来执行的,该语言解释器负责解释编写该程序所用的语言。通过三种方式来支持可迁移性:发送者启动的弱可移动性,进程迁移所代表的强可移动性,以及进程克隆所代表的强可移动性。

弱可移动性是使用 agent_submit 命令来实现的。给出目标机器的标识符作为参数,并且给出在目标机器上执行的脚本。脚本就是一系列指令。脚本与过程定义和变量一起传输到目标机器上,这些过程定义和变量是目标机器在执行脚本时所必需的。随后,在目标机器上运行对应的语言解释器的进程就将开始执行该脚本。就如同在图 3.13 中显示的代码迁移方法一样,D'Agents 提供对发送者启动的弱可移动性的支持,迁移后的代码放在单独的进程中执行。

为了给出 D'Agents 中弱移动性的实例,图 3.16 中展示了一个简单的 Tel 代理的一部分,该代理向远程机器提交脚本。在该代理中,过程 factorial 只有一个参数,它递归地求其参数阶乘表达式的值。假定变量 number 和 machine 已经进行过合理的初始化(比如说要求用户提供它们的值),随后代理调用 agent_submit。脚本 factorial \$ number、过程 factorial 的描述还有变量 number 的初始值都被发送到变量 machine 指向的目标机器。D'Agents 会自动进行安排,将得到的结果送回代理。对 agent_receive 的调用将会确保正在进行提交的代理被阻塞直到收到计算结果为止。

```
proc factorial n {
    if { $n < 1 } { return 1; }                                # fac(1)=1
    expr $n * [ factorial [ expr $n - 1 ] ]                # fac(n) = n * fac(n-1)
}

set number...          # 告知哪一个阶乘用来计算集合
machine ...           # 标识目标机器

agent_submit $machine -procs factorial -vars number -script {factorial $number}
agent_receive ...     # 接收结果(为了简单化,不考虑未指定的)
```

图 3.16 D'Agents 中 Tel 代理行为的一个简单示例:向远程机器提交脚本(改编自 Gray 1995)

D'Agents 也支持发送者启动的强可移动性,而且对进程迁移和进程克隆这两种方式都支持。如果要迁移正在运行的代理,代理先调用 agent_jump 以指定要迁移到的目标机器。在调用 agent_jump 之后,在源机器上执行的代理被挂起,其资源段、代码段和执行段被编组进消息中去,随后将消息发送给目标机器。在消息抵达的时候,就启动一个运行对应的解释器的新进程,由该进程来解除消息的编组,并且让代理从调用 agent_jump 之后的下一条指令开始继续执行。而在源机器上运行该代理的进程则退出。

图 3.17 中给出了代理迁移的一个例子,在该示例中,一个简化版本的代理通过在所有主机上执行 UNIX 命令 who 找出当前登录的所有用户。过程 all_users 给出了代理的行为模式。它维护一个用户列表,该列表开始时是空的。参数 machines 指定了要进行用户查询的一组主机。代理轮流向不同主机进行查询,将执行 who 命令得到的结果放在变量 users 中,然后添加到用户列表末尾。在主程序中,代理是在当前机器上通过提交创建的,也就是使用前面讨论过的弱可移动性机制实现的。在这种情况下,需要调用 agent_submit 来执行脚本。all_users \$ machines 得到了过程和作为附加参数的一组主机。

```
proc all_users machines {
    set list ""
    foreach m $ machines {
        agent_jump $ m
        set users [exec who]
        append list $ users
    }
    return $ list
}
set machines...
set this_machine ...
# 通过对这个机器提交脚本来创建一个移植代理,位置在 $ machines 中它将跳转到所有其他主机的地方
agent_submit $ this_machine -procs all_users -vars machines \
             -script { all_users $ machines }

agent_receive ... # 接收结果(为了简单化,不考虑未指定的)
```

图 3.17 D'Agents 中的 Tel 代理行为示例: 迁移到不同的机器上并执行 UNIX 的 who 命令
(改编自(Gray 1995))

最后,通过 agent_fork 命令来支持进程克隆。该命令与 agent_jump 的作用几乎相同,只是源机器上运行代理的进程在调用 agent_fork 后将会从下一条指令开始继续执行。与 UNIX 中的 fork 操作相似,agent_fork 返回一个值,调用方可以根据该值来检查进行调用的代理是克隆出来的版本还是原来的那个调用者。

2. 实现方面的问题

为了说明某些内部实现细节,让我们看一下使用 Tel 编写的代理程序。D'Agents 系

统在内部包含 5 层,如图 3.18 所示。其中的最底层可以比作 Berkeley 套接字,因为它们都实现了用于访问底层网络提供的通信功能的接口。在 D' Agents 中,假定底层系统提供了处理 TCP 消息和电子邮件的功能。

第 2 层由服务器构成,这些服务器在执行 D' Agents 代理的机器上运行。服务器负责代理的管理、验证以及代理之间的通信管理。为了处理代理之间的通信,服务器向每一个代理都指派一个惟一的本地标识符。如果知道了服务器的网络地址,可以使用(地址,本地标识符)对来对代理进行引用。这个低级的名字(标识符)是用来在两个代理之间建立通信的。

第 3 层居于 D' Agents 系统的核心,由语言无关的内核构成,该内核支持基本的代理模型。比如,该层包含启动和终止代理的实现,各种迁移操作的实现,以及用于代理间通信的功能。很明显,内核的运行方式与服务器相近,它与服务器的不同之处在于内核并不负责管理运行于同一台机器上的所有代理。

第 4 层由解释器构成,每种解释器负责解释 D' Agents 支持的一种语言。解释器由一个用来解释语言的组件、一个安全模块、一个面向内核层的接口以及一个用于捕捉运行中代理的状态的单独模块构成。最后一个模块对于支持强可移动性是非常重要的,将在下面进行详细讨论。

最高层是由使用某种支持的语言编写的代理构成的。D' Agents 中的每个代理都由一个单独的进程来执行。比如说,当某个代理迁移到机器 A 上的时候,A 上的服务器会派生出一个进程来执行适用于迁移过来的代理的解释器,随后该新进程得到迁移过来的代理的状态,然后它就会让代理在原先中止执行的位置继续往下执行。服务器负责使用局部管道(local pipe)对它创建的进程进行跟踪,以便将传入的调用传递给合适的进程。

D' Agents 实现中更为困难的部分在于捕捉运行中代理的状态,并将该状态移动到另一台机器上去。Tcl 中的代理状态由两部分构成,如图 3.19 所示。本质上,有 4 张表用于保存变量和脚本的全局定义,还有两个栈用来追踪执行状态。

状态	描述
全局解释器变量	代理解释器需要的变量
全局系统变量	返回码、错误码、错误字符串等
全局程序变量	某个程序中用户定义的全局变量
过程定义	要由代理执行的脚本定义
命令堆栈	存储当前正在执行的命令的堆栈
调用框架堆栈	存储激活记录的堆栈,每个记录对应于一个正在运行的命令

图 3.19 D' Agents 中包含代理状态的组成部分

有一张表用于存储供解释器使用的全局变量。比如说,可能由一个事件处理程序来告知解释器在来自某个特定代理的消息到来时应该调用哪一个过程。在解释器表中存储

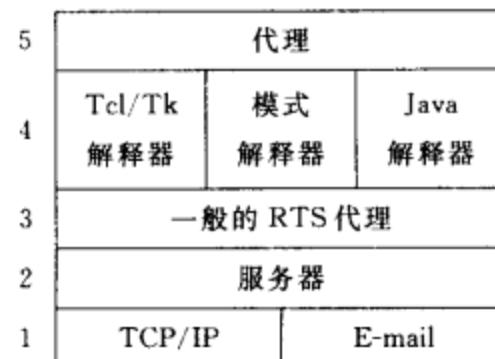


图 3.18 D' Agents 系统的体系结构

有(事件,处理程序)对条目。另一张表中包含系统变量,用于存储错误码、错误字符串、结果码、结果字符串等。还有单独的一张表,其中含有所有用户定义的全局程序变量。最后,另外一张单独的表中存放了与代理相关联的过程的定义,这些过程定义需要与代理一同迁移,只有这样才能够在目标机器上进行解释。

与代理迁移相关的部分中,更有意思的是有两个堆栈用来保存对代理实际执行状态的正确记录。基本上来说,代理可以看作是一系列的 Tcl 命令,这些 Tcl 命令嵌入在循环、case 语句等结构中。另外,还可以把多条命令合起来组成过程。就像任何可解释型语言一样,代理是一条命令接着一条命令执行的。

先考虑一下在执行基本 Tcl 命令时会发生的情况,基本 Tcl 命令也就是那些不是对用户定义过程进行调用的命令。解释器将会对命令进行解析,随后建立一条记录,把该记录压入命令栈(command stack)中去。这种记录包含了实际执行命令所需的所有字段,比如参数值、指向实现该命令的过程的指针等。系统将该记录压入堆栈之后,把它交给负责实际命令执行的组件。也就是说,命令栈准确记录了代理当前的执行状态。

Tcl 也支持用户定义的过程。除了命令栈之外,D'Agents 运行时环境还追踪一个存储激活记录的堆栈,激活记录也称作调用框架。D'Agents 中的调用框架包含过程中的局部变量列表,以及调用该过程时使用的参数值。只有进行过程调用才会创建调用框架,它在被压入命令栈时与过程调用命令有关。调用框架中保存对相关联命令的引用。

现在来考虑一下当代理调用 agent_jump 时发生的事情,agent_jump 将代理迁移到另一台机器上。这种情况下,代理的整个状态都被编组成字节序列,也就是说 4 张表和两个堆栈都合起来放进单个字节序列中,并传输到目标机器上。目标机器上的 D'Agents 服务器随后会创建一个运行 Tcl 解释器的新进程。该进程得到传来的编组数据,随后解除数据的编组,使之变为代理在之前调用 agent_jump 时的状态。只要简单地从命令栈顶部弹出下一个命令,代理就可以从原先中止的地方继续往下执行。

3.5 软件代理

到此为止,我们已经从多个不同角度考察了进程。首先,我们重点讨论了一个基本问题,也就是进程中的控制线程(单个或者多个)问题。我们还从通信的角度详细考察了客户和服务器的一般组织结构。最后,我们还讨论了程序和进程的可迁移性。这些多少彼此独立的针对进程的观点凑起来就组成了软件代理的概念:软件代理是一些独立的单元,能够与其他(可能是远程的)代理进行协作,一同执行任务。

代理在分布式系统中扮演着日趋重要的角色。然而,就像对于进程究竟是什么的问题有一个直观的答案一样(Organick 1972),软件代理也已经有了精确的定义。在本节中,我们将详细讨论代理的本质以及它们在分布式系统中扮演的角色。

3.5.1 分布式系统中的软件代理

关于代理究竟是什么的问题存在着很多争论。按照文献(Jennings 和 Woolridge 1998)中给出的描述,软件代理(software agent)定义为能够对环境中的变化作出反应,并

且启动这种变化的自治进程,而且可能与用户代理或者其他代理协同。代理区别于一般进程的特征是它能够对自己执行操作,特别是能够在适当的时候采取主动。

我们对软件代理的定义要更宽一些,许多类型的进程都可以称作代理。与其把精力放在寻找更好的定义上,不如对各类代理进行考察更有意义。同时,在文献中记载了人们在开发软件代理的分类方法上所做的努力,但是研究者似乎难以就究竟采用哪种分类方法达成一致。

除了自治性之外,代理还有一个重要特征,就是必须能够与其他代理协作。自治与协作相结合,就产生了合作代理(Nwana 1996)。合作代理构成了多代理系统的一部分,在多代理系统中代理通过相互协作来寻求达到某个共同目标。合作代理的一种典型应用是会议的安排。每个出席会议者都由一个了解该用户个人日程安排的代理来代表。如果确定了每个人的时间、旅程、地点等方面安排,各个独立的代理就可以相互协作以制定整个会议的日程。从分布式系统开发的角度来说,交换的究竟是哪些信息以及如何处理这些信息是放在较次要的地位上考虑的,而重要的是要考虑通信究竟是如何进行的。我们将在下面继续讨论代理间通信问题。

许多研究者把移动代理(mobile agent)与其他类型的代理区分开来。移动代理是能够在不同机器间迁移的代理。根据前一节中对代码迁移的讨论,移动代理常常需要对强可移动性的支持,虽然这种支持并不是必需的。这种对于强可移动性的需求出自于这样一个事实,那就是代理是自治的,同时又是与所在环境积极交互的。如果不考虑代理的执行状态,那么将代理迁移到另一台机器上的工作是难以完成的。然而,就像 D'Agents 系统所说明的那样,代理与弱可移动性的结合也是有用的。要注意,可移动性是代理的普遍特征,并不能根据可移动性划分出一个与众不同的代理类别来。比如说,讨论移动合作代理是有意义的。在文献(Brewington 等 1999)中给出了关于在实际中使用移动代理的一个很好的例子,作者描述了如何使用移动代理来获取分布在 Internet 这样的大型异构网络上的信息。

与其他代理协作的能力或者在不同机器间移动的能力是代理的系统属性,而与代理的实际功能无关。在考察代理功能时,还可以采取其他的分类方法。

一个得到普遍承认的类别是接口代理(interface agent),这种代理协助最终用户使用一个或者多个应用程序。接口代理的一种得到普遍认可的关键特征就是它拥有学习(learning)能力(Maes 1994 和 Nwana 1996)。它与用户交互得越频繁,它的协助能力就越强。在分布式系统的环境中,有一些代理寻求与位于同一个领域中的用户使用的代理相交互,它们是接口代理的有趣的例子。比如说,存在一种主动寻求买方和卖方并让他们进行接洽的特殊代理。这种代理通过不断熟悉所有者在买进物品(或者卖出物品)方面的需求,可以提升选择合适卖方(或者买方)的能力。

信息代理(information agent)与接口代理有着紧密的联系。这些代理的主要功能是管理来自多个不同信息源的信息。这些信息管理任务包括排序、过滤、比较等。信息代理在分布式系统中之所以如此重要,是因为它们要处理的信息来自于物理上不同的信息源。固定信息代理一般处理输入的数据流。比如说,电子邮件代理能够把不希望接收的邮件从主人的邮箱中过滤掉,或者把收到的分散信件按照主题分门别类地放到相应的邮箱中

去。相反，移动信息代理一般在网络上漫游，替主人搜集所需的信息。

总的说来，可以通过多种属性来描绘代理，如图 3.20 所示（请参阅文献（Franklin 和 Graesser 1996））。可以从人工智能的角度来考察代理的运行方式，以对代理进行进一步分类。如果想浏览这方面的简要介绍，请参阅文献（Hayes 1999；Jennings 和 Woolridge 1998；Woolridge 1998）。

属性	所有代理都具有吗？	描述
自治性	是	可以对自身执行操作
反应性	是	对所在环境中的变化作出及时响应
主动性	是	能够采取影响其所在环境的行动
通信	是	能够与用户和其他代理交换信息
连续性	否	有一个相对较长的时期
移动性	否	可以从一个站点迁移到另一个站点
适应性	否	学习能力

图 3.20 代理的部分重要属性，可以根据这些属性对代理进行分类

3.5.2 代理技术

如果在代理系统实际开发上无法获得充分支持，光有代理的概念是没有什么用的。现在面临的重要问题是，如何将分布式系统中的通用代理组件隔离开来并将这类组件嵌入到中间件等系统中去。首先，应开发出智能物理代理基础（foundation for intelligent physical agents, FIPA）作为软件代理的通用模型。在这个模型中，代理在代理平台中注册，并且按照代理平台的规定来运行，如图 3.21 所示。代理平台提供了任何一个多代理系统都需要的基本服务，这些服务包括创建和删除代理的功能、定位代理的功能以及代理间通信功能。

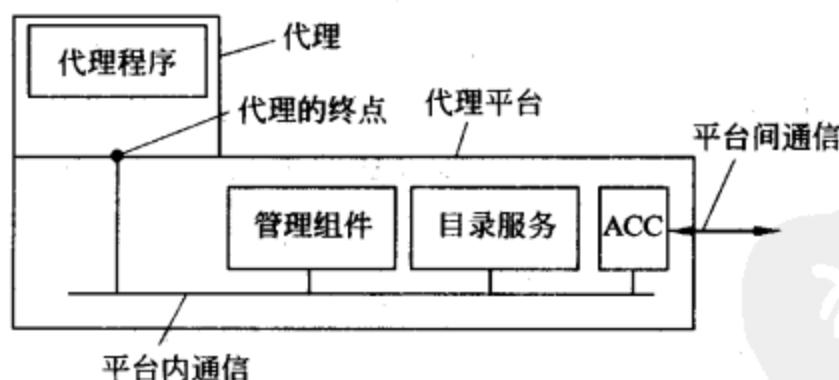


图 3.21 普通代理平台模型（改编自（FIPA 1998b））

代理管理组件负责跟踪相关平台的代理。它提供了创建和删除代理的功能，以及查询某个特定代理当前终点的功能。从这个意义上说来，它提供了一种命名服务，这种服务将全局唯一的标识符映射为本地通信终点。我们将在下一章中对命名服务进行详细讨论。

也有一种独立的本地目录服务，代理可以通过该服务来查询平台上的其他代理所提

供的服务。在 FIPA 模型中,目录服务是基于属性的使用的,也就是说代理以属性名的形式来给出对它所提供的服务的描述,这与“黄页”的工作机制极为类似。远程代理——也就是驻留在不同代理平台上的代理也可以访问目录服务。

代理通信通道(agent communication channel, ACC)构成了代理平台上一个重要组件。在多数多代理系统模型中,代理之间通过互相交换消息来通信。FIPA 模型也不例外,由 ACC 来掌管不同代理平台之间的所有通信。也就是说,由 ACC 负责与其他平台进行可靠的并且是有序的点到点通信。可以简单地利用一个服务器来实现 ACC,该服务器监听到达某个特定端口的消息,并将该消息转发给其他服务器或者代理平台上的代理。为了确保平台间互操作性,不同平台的 ACC 间的通信遵循互联网 Inter-ORB 协议(Internet inter-ORB protocol, HOP),我们将在第 9 章中讨论该协议。D'Agents 系统体系结构中的服务器是 ACC 的一个例子。

代理通信语言

至此为止,代理平台已经没有什么特殊之处了,只要考察一下代理之间通信所交流的信息的类型,就可以清楚它与分布式系统的其他方法之间的区别。代理间通信是使用应用程序级通信协议进行的,这种协议也称作代理通信语言(agent communication language, ACL)。ACL 把消息的用途(purpose)与消息的内容(content)二者严格区分开来。比如说,消息的用途可以是请求接收者提供特定的服务,也可以是对之前发送来的请求消息作出响应。另外,还可以发送某些消息以通知事件的接收者,或者在协商过程中提出建议。FIPA 中的 ACL 所规定的消息用途如图 3.22 所示。

消息用途	描述	消息内容
INFORM	告知已经批准提出的建议	建议
QUERY-IF	查询提出的建议是否得到批准	建议
QUERY-REF	查询给定对象	符号
CFP	征求建议	建议说明
PROPOSE	提出建议	建议
ACCEPT-PROPOSAL	告知提出的建议已得到采纳	建议 ID
REJECT-PROPOSAL	告知提出的建议已被拒绝	建议 ID
REQUEST	请求采取行动	行动说明
SUBSCRIBE	预订信息源	信息源引用

图 3.22 FIPA ACL(FIPA 1998a)中规定的各种消息类型示例。图中给出了消息的目的以及消息实际内容的描述

ACL 最基本的要求就是进行发送的代理和进行接收的代理至少要就消息的用途达成共识。而且,消息的用途常常决定了接收者的反应。比如说,如果发出一个报头中含有 CFP 的消息来征求建议,就希望该消息的接收者发出一个含有 PROPOSE 的消息来提出一个建议。从这种意义上说,ACL 实际上定义了一组代理间的高级通信协议。

与多数通信协议相似,ACL 消息由报头和实际内容组成。报头中包含有标识消息用

途的字段,还有标识发送者和接收者的字段。消息内容是与其他部分分开并且是独立的,也就是说,可以认为不同代理进行通信的消息内容可以不同。ACL 并不规定表述消息内容时一定要使用什么样的语言或者格式。

随后必须向负责接收的代理提供足够的信息,以便它对消息内容作出正确解释。最后,ACL 消息头中也可以包含有一个用于标识内容所使用语言或者编码方式的字段。只要发送者和接收者就数据(更准确地说,是消息中的符号)的解释方式达成了共识,这种方法就是可行的。如果尚未达成共识,就要使用另外一个字段来标识符号与其所表示含义的确切映射关系,这种映射关系一般称作实体(ontology)。

图 3.23 中给出了一个简单的例子,该例子表示的是用一个以 FIPA ACL 表述的消息,把关于荷兰王室的家族信息通知给代理。为了标识进行发送的代理和进行接收的代理,每个代理都拥有一个由若干部分组成的名字。比如说,如果某个名为 max 的代理驻留在 DNS 名为 fanclub-beatrix. royalty-spotters. nl 的代理平台上,就可以使用 max@http://fanclub-beatrix. royalty-spotters. nl:7239 来引用该代理。如果要与代理通信,必须首先将平台名通过 DNS 解析成 IP 地址。而且,该名字还指定了通信必须通过向主机上监听 7239 端口的服务器发送 HTTP 消息来进行。在本例中,代理 max 把携有信息的消息发送给名为 elke 的代理。该代理驻留在名为 royalty-watcher. uk 的平台上。必须使用 IIOP 协议(该协议将在第 9 章中讨论)来发送消息,并且要发送到 5623 端口。

字段	值
用途(purpose)	INFORM
发送者(sender)	max@http://fanclub-beatrix. royalty-spotters. nl:7239
接收者(receiver)	elke@iiop://royalty-watcher. uk:5623
语言(language)	Prolog
实体(ontology)	家谱
内容(content)	female(beatrix), parent(beatrix, juliana, bernhard)

图 3.23 两个代理之间传递的 FIPA ACL 消息示例,该消息使用 Prolog 来表述家谱信息

消息中的其他字段与消息内容有关。language 字段指定了消息内容是用一系列 Prolog 语句来表述的,而 ontology 字段指定 Prolog 语句的语义要作为家谱信息来解释。结果,进行接收的代理现在明白语句 female(beatrix)用来说明有一位妇女的名字叫做 beatrix,而语句 parent(beatrix, juliana, bernhard)说明 beatrix 的母亲名叫 juliana,父亲名叫 bernhard。

3.6 小结

在分布式系统中,进程是一个基本的部分,它们构成了不同机器间通信的基础。一个重要的问题是,进程内部的组织结构究竟是什么样的,更具体的说就是它们是否支持多个控制线程。分布式系统中的线程对于在执行阻塞性 I/O 操作时继续使用 CPU 是非常有用的。如果采取了多线程方式,就可以构建效率更高的服务器,让服务器并行运行多个线

程,其中某几个线程可以由于阻塞而处于等待状态,直到磁盘 I/O 或者网络通信完成为止。

以客户端-服务器的方式来组织分布式应用程序已经证实是很有用的。客户过程一般实现用户界面,这种用户界面可以只提供简单的显示,也可以提供能够处理复合文档的高级界面。而且客户软件还通过隐藏与服务器通信的细节——还包括服务器当前所在位置和服务器是否被复制等细节来取得更好的分布透明性。另外客户端软件还可以负责进行部分故障隐藏及恢复工作。

服务器一般要比客户更加复杂,但是与设计相关的问题相对较少。比如说,服务器可以是重复的也可以是并发的,可以实现一种服务也可以实现多种服务,可以是状态无关的也可以是状态相关的。其他设计方面的问题涉及寻址服务,以及在服务请求已经提出甚至已经在处理过程中的情况下中断该服务的机制。

对象服务器是一个特殊的类别。在本质上说,对象服务器是一种在其地址空间中容纳了多个对象的进程,它接收针对这些对象的调用请求。对象服务器的特殊之处在于它可以使用多种方法来调用对象。比如说,服务器可以为每一个调用请求启动一个单独的线程来处理,也可以用一个线程来处理一个对象,甚至还可以用一个单独的控制线程来处理所有对象。同一个服务器可以使用对象适配器来执行多种调用策略。本质上来说,对象适配器是只实现一种调用策略的组件,每个服务器可以拥有多个对象适配器。

在不同机器间进行代码迁移是分布式系统中的另一个重要主题。支持代码迁移的理由是为了提高性能及灵活性。由于通信的开销很大,有时需要把计算工作由服务器移到客户端进行,让客户尽可能多地进行本地处理,以减少通信量。如果客户能够动态下载与特定服务器通信所需的软件,就可以提高灵活性。从服务器下载的软件是该服务器专用的,而不需要客户预装该软件。

代码迁移会带来与本地资源使用相关的问题,因为迁移时要求资源同时迁移,并且在目标机器上重新绑定到本地资源,或者使用系统范围的网络引用。另一个问题是,在迁移代码时要考虑到异构性。目前的实践证明,最好的解决方法是使用虚拟机来处理异构性,虚拟机通过对代码进行解释可以有效地隐藏异构性。

最后来回顾一下软件代理,它是一类特殊的进程,是作为自治单元来运行的,但是它能够与其他代理互相合作。从分布式系统的角度来说,代理与普通进程的区别在于,代理通过称为代理通信语言(ACL)的应用层协议进行交互。ACL 把消息的用途和消息的内容二者严格区分开来。ACL 定义的是高层通信协议:发送的消息规定了消息接收者所应该作出的反应,而该反应只能基于消息的用途。

习 题

1. 比较使用单线程文件服务器读取文件和使用多线程服务器读取文件有什么不同。花费 15ms 来接收请求、调度该请求并且完成其他必须的处理工作,假定需要的数据存放

在主存储器的缓存中。如果需要磁盘操作,就需要额外多花 75ms,在磁盘操作的过程中线程处于睡眠状态。如果服务器采用单线程的话,它每秒能处理多少个请求?如果采用多线程呢?

2. 对服务器进程中的线程数目进行限制有意义吗?
3. 在文中我们描述了一个多线程的文件服务器,说明了为什么它比单线程服务器和有限状态机服务器更好。有没有这样的环境,在其中使用单线程服务器会更好?给出这种环境的例子。
4. 将轻量级进程与单个线程静态关联起来并不好,为什么?
5. 如果每个进程只使用单个轻量级进程也不好,为什么?
6. 描述一种使用与可运行线程数目相等的轻量级进程的方法。
7. 本章解释过,代理(proxy)可以通过调用所有副本来自支持引用透明性。能否对(服务器端的)对象的副本进行调用?
8. 通过生成进程来构建并发服务器与使用多线程服务器相比有优点也有缺点。给出部分优点和缺点。
9. 粗略地设想一种多线程服务器的设计,该服务器必须使用套接字作为面对底层操作系统的传输级接口,以支持多种协议。
10. 如何防止应用程序绕过窗口管理器破坏屏幕显示?
11. 解释对象适配器的概念。
12. 举出几个用于支持持久性对象的对象适配器设计方面的问题。
13. 改编对象适配器示例中的 `thread_per_object` 过程,使用单个线程来处理受该适配器控制的所有对象。
14. 维护到客户的 TCP/IP 连接的服务器是状态相关的还是状态无关的?
15. 想象一下,某个 Web 服务器维护一个列表,该列表中的内容是 IP 地址与该地址最近访问过的 Web 页的映射关系。当一个客户连接到该服务器的时候,该服务器在列表中查找该客户,如果找到的话就返回注册过的页面。这个服务器是状态相关的还是状态无关的?
16. Java RMI 对代码迁移依赖到何种程度?
17. 在 UNIX 系统中,可以通过让进程在远程机器上派生出一个子进程来支持强可移动性。请说明这种机制的工作机理。
18. 图 3.13 指出,强可移动性不能与在目标进程中已迁移代码的执行结合在一起。请举出一个反例。
19. 考虑某个进程 P,它请求访问与自己位于同一台机器上的本地文件 F。在 P 迁移到另一台机器上以后,它还需要访问 F。如果文件对机器的绑定是坚固的,如何实现对 F 的系统级引用?
20. D'Agents 系统中的每个代理都由单独的进程实现。代理通信的主要方式是共享文件以及消息传递,而文件无法在机器之间传递。如果按照 3.4 节中给出的移动框架结构,代理的哪一部分状态(在图 3.19 中)包含了资源段?

21. 将 D'Agents 的体系结构与 FIPA 模型所实现的代理平台的体系结构进行比较。
22. 代理通信语言(ACL)在哪些方面与 OSI 模型相符?
23. 在系统顶层实现代理通信语言以进行电子邮件处理时(比如在 D'Agents 系统中),这种代理通信语言又有哪些方面与 OSI 模型相符? 这种方案有什么优点?
24. 为什么 ACL 消息中通常必须指定实体(ontology)?

第 4 章 命 名

名称在所有计算机系统中都起着重要的作用。它们用来共享资源、惟一标识实体、指向位置等。命名的一个重要结果是可以把名称解析为它所指向的实体。因而，名称解析允许进程访问命名的实体。要解析名称，需要实现命名系统。分布式系统中的命名与非分布式系统中的命名之间的区别在于命名系统实现的方式。

在分布式系统中，命名系统的实现本身通常是分布在多台计算机上的。该分布是如何完成的对命名系统的效率和可扩展性起着关键的作用。在本章中，我们集中讨论在分布式系统中使用名称的三种不同的重要方法。

首先，在讨论一些与命名有关的常见问题之后，我们将进一步研究易于理解的名称的组织和实现。这样的名称的典型例子包括用于文件系统和万维网的名称。建立世界范围的、可扩展的命名系统是这类名称主要关注的问题。

其次，名称用来定位移动的实体。实践表明，用于建立易于理解的名称的命名系统并不特别适合支持大量移动实体（这些实体还可能分散在大规模的网络中）。还需要其他的命名机制，例如用于移动电话的机制，在这种命名机制中，名称是与位置无关的标识符。

我们的第三个，也即最后一个主题是名称的组织。特别要指出的是，不再被引用的名称（因此不能再被定位和访问）应该自动被删除。该主题也称为垃圾收集，起源于一些编程语言中。但是，随着大规模分布式基于对象的系统的引入，自动收集不被引用的对象变得愈加重要了。

4.1 实体的命名

在本节中，我们首先集中讨论不同类型的名称，以及名称是如何组成名称空间的。然后我们继续讨论一个重要的问题，即如何解析一个名称，使得它所指向的实体可以得到访问。此外，我们还将介绍在多台计算机上分布和实现大型名称空间的多种方法。Internet 域名系统和 OSI 的 X.500 在这里将作为大规模命名服务的实例进行讨论。

4.1.1 名称、标识符和地址

分布式系统中的名称是由位或者字符所组成的串，它用来指向一个实体。分布式系统中的实体几乎可以是任何事物。典型的实例包括如主机、打印机、磁盘和文件等资源。通常得到明确命名的众所周知的其他实体实例还有进程、用户、邮箱、新闻组、Web 页面、图形窗口、消息、网络连接，等等。

对实体可以进行操作。例如，像打印机这样的资源可以提供一个界面，在打印机的接

口包含打印文档、查询打印工作的状态等操作。此外，像网络连接这样的实体可以提供发送和接收数据、设置服务质量参数、查询状态等操作。

如果要对实体进行操作，就需要访问实体，因此需要一个访问点(access point)。访问点是分布式系统中另一种特殊类型的实体。访问点的名称称为地址(address)。一个实体的访问点的地址也简称为该实体的地址。

一个实体可以提供多个访问点。打个比方，电话号码可以视为一个人的访问点，其中电话号码相当于地址。实际上，如今许多人都拥有多个电话号码，每个号码都相当于可以联系到他们的一个访问点。在分布式系统中，访问点的一个典型实例就是运行某个特定服务器的主机，可以通过 IP 地址和端口号(即服务器的传输层地址)的组合来构成其地址。

一个实体经过一定的时间后可能改变其访问点。例如，当一台移动式计算机移动到另一个地方时，通常会为它分配一个与以前不同的 IP 地址。同样，一个人到达另一个城市或国家后，通常也需要改变电话号码。与此类似，变换工作或者改变 Internet 服务提供商(ISP)都意味着要改变电子邮件的地址。

因此，地址就是一种特殊类型的名称：它指向实体的访问点。因为访问点是与实体紧密关联的，所以使用访问点的地址作为相关实体的常规名称似乎是非常方便的。但是，在实际中却很少这样做。

将地址作为一种特殊类型的名称进行处理有许多益处。例如，定期地重新组织一个分布式系统，使一台特定服务器(例如一台处理 FTP 请求的服务器)运行于与以前不同的主机上是很常见的工作。而该服务器以前所在的计算机可能会被重新指定一个完全不同的服务器(例如本地文件系统的备份服务器)。换句话说，一个实体很可能会改变访问点，或者一个访问点可能会重新分配给不同的实体。

如果把地址用于指向实体，那么由于访问点随时会发生改变或者重新分配给另一个实体，我们可能会得到一个无效的查询结果。例如，一个公司的 FTP 服务可能仅通过运行 FTP 服务器的主机地址来标识。一旦该服务器移动到另一台主机上，那么整个 FTP 服务会变得无法访问，直至其所有用户都知道了新的地址为止。在这种情况下，应该让 FTP 服务由一个与相关的 FTP 服务器地址无关的单独名称来标识。

同样，如果一个实体提供了多个访问点，那么哪个地址用来作为引用是不清楚的。例如，我们在第 1 章中所讨论的，许多公司将其 Web 服务分布在几台服务器上。如果用这些服务器的地址来作为指向 Web 服务的引用，那么哪个地址应该作为最佳选择是不明显的。更好的解决方法是让 Web 服务具有一个与不同 Web 服务器的地址无关的单一名称。

这些实例说明一个独立于实体地址的名称通常是比较简单的，而且使用更为灵活。我们称这样的名称是与位置无关(location independent)的。

除了地址之外，还有其他类型的名称值得特别注意，例如用来惟一标识实体的名称。真正的标识符(identifier)是具有以下属性的名称(Wieringa 和 de Jonge 1995)：

- (1) 一个标识符最多指向一个实体；
- (2) 每个实体最多由一个标识符指向；

(3) 一个标识符始终指向同一个实体(也就是说,标识符永远不会重新使用)。

通过使用标识符,明确地指向某个实体变得更加容易。例如,假设两个进程都通过标识符指向一个实体。如果要检查两个进程是否指向同一实体,那么检验两个标识符是否相等就可以了。如果这两个进程使用的是常规的非标识性名称,那么仅做这样的检验是不够的。例如,名字“John Smith”不能作为指向某个人的惟一引用。

同样,如果一个地址可以重新分配给不同的实体,那么我们就不能使用地址作为标识符。考虑电话号码的使用,一个号码通常指向同一个人或公司,这是相当稳定的。但是,不能使用电话号码作为一个标识符,因为经过一段时间电话号码可能会重新分配。因此,Bob 的新面包店可能会长期接到找 Alice 的旧五金商店的电话。在这种情况下,最好让 Alice 使用一个真正的标识符,而不是使用她的电话号码。

地址和标识符是两个重要的名称类型,每个类型用于不同的目的。在许多计算机系统中,地址和标识符仅使用计算机可读的形式(即位串的形式)来表示。例如,以太网地址就是任意的 48 位的位串。同样,内存地址通常用 32 位或 64 位位串来表示。

另一种重要的名称类型是为用户使用而定制的,也称为易于理解的名称(human-friendly name)。相对于地址和标识符而言,易于理解的名称通常用字符串来表示。这些名称表现为各种各样的形式。例如,UNIX 系统中的文件具有最多 255 个字符的字符串名称,它完全是由用户来定义的。与此类似的是,DNS 名称是用相对简单的不区分大小写的字符串来表示的。

名称空间

分布式系统中的名称都组织在通常所说的名称空间(name space)中。名称空间可以表示为带有标注的具有两种类型的节点的有向图。叶节点(leaf node)表示一个命名的实体,其性质是不具有分支边。叶节点通常存储关于它所表示的实体的信息(例如实体的地址),以便客户访问该实体。另外,叶节点还可以存储实体的状态。例如在文件系统中,叶节点实际包含它所代表的完整文件。我们在下文中还会继续对节点内容的讨论。

与叶节点相比,目录节点(directory node)具有一定数量的分支边,每条边用一个名称来标注,如图 4.1 所示。命名图中的每个节点都认为是分布式系统中的一个实体,具有一个相关联的标识符。目录节点用于存储一个表,其中每条分支边用一个(边标签,节点标识符)对来表示。这样的表称为目录表(directory table)。

图 4.1 所示的命名图具有一个特殊的节点,即 n_0 ,它只具有分支边,而没有进入边。这样的节点称为命名图中的根(节点)。尽管命名图可以含有多个根节点,但为了简单起见,许多命名系统只含有一个根节点。命名图中的每个路径可以通过路径中的边的标签序列来指向,例如:

$N: <label-1, label-2, \dots, label-n>$

其中 N 指向路径中的第一个节点。这样的序列称为路径名(path name)。如果路径名中的第一个节点是命名图的根,那么该路径名称为绝对路径名,否则,称为相对路径名。

认识到名称始终组织在名称空间中这一点是很重要的。因此,名称始终仅相对于目录节点进行定义。在这个意义上,绝对名称这个术语很容易使人误解。同样,全局和局部

名称之间的差异有时也会混淆。全局名称(global name)无论在系统中何处使用,都表示同一个实体。换句话说,全局名称始终根据相同的目录节点进行解释。相比之下,局部名称(local name)需要根据使用名称的位置来进行解释。当放置在不同的位置上时,局部名称实质上是相对名称,包含它的目录是隐式已知的。我们在讨论名称解析时还会回来讨论这些问题。

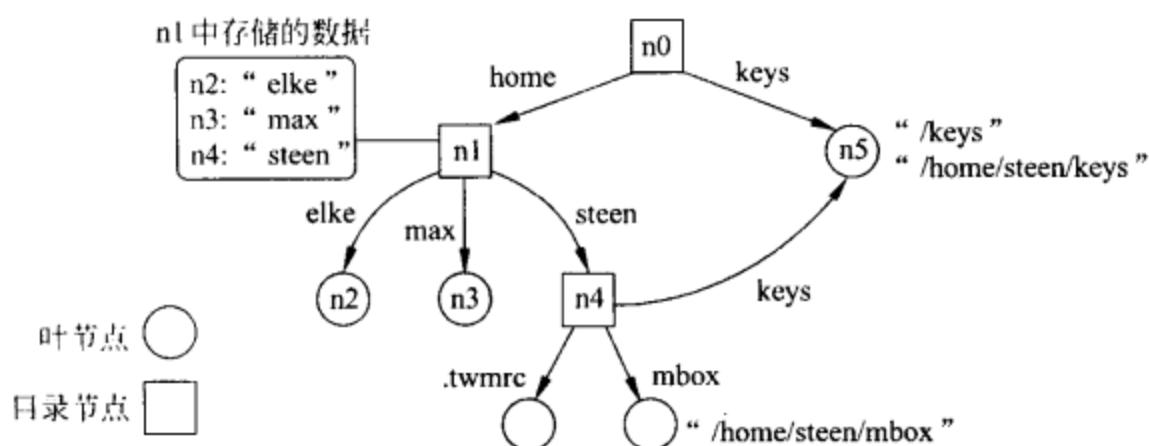


图 4.1 具有单个根节点的一般命名图

命名图的描述与其在许多文件系统中的实现是非常接近的。但是,在具体实现中,不是用边标签序列来表示路径名的。文件系统中的路径名通常用一个字符串来表示,其中边标签之间由特定的分隔字符来分隔,例如斜杠(“/”)。该字符也用来表明路径名是否是绝对的。例如,在图 4.1 中,通常不使用路径名 $n_0: \langle \text{home}, \text{steen}, \text{mbox} \rangle$,而是使用其字符串表示法/home/steen/mbox 来表示。还要注意的是,当有多个路径通向同一节点时,该节点可以由不同路径名来表示。例如图 4.1 中的节点 n_5 ,使用/home/steen/keys 和/keys 都可以指向该节点。路径名的字符串表示法除了用于文件系统之外,也可以用于其他命名图中。在 Plan 9(Pike 等 1995)中,所有的资源(例如进程、主机、I/O 设备和网卡)都是以与传统文件相同的方式进行命名的。这个方法类似于为在一个分布式系统中的所有资源实现一个单一的命名图。

组织名称空间有许多不同的方法。如我们所提到的,大多数名称空间仅有一个根节点。在许多情况下,名称空间也严格地进行分层,将命名图组织成一棵树。这意味着除了根节点之外,每个节点都只有一条进入边,而根节点没有进入边。因此,每个节点只有一个相关联的绝对路径名。

图 4.1 所示的命名图是有向非循环图(directed acyclic graph)的一个实例。在这样的组织中,一个节点可以具有多条进入边,但是图中不允许有循环。也有些名称空间没有此限制。

为了更具体一些,我们来考虑一下在传统 UNIX 文件系统中对文件进行命名的方式。在 UNIX 的命名图中,一个目录节点表示一个文件目录,而一个叶节点表示一个文件。惟一的根目录在命名图中用根节点表示。命名图的实现是文件系统的完整实现的一部分。该实现由逻辑磁盘上一系列连续的块组成,通常分为一个根块、一个超级块、一系列索引节点(称为信息节点,inode)和一些文件数据块。请参见文献(Crowley 1997; Nutt 2000; Tanenbaum 和 Woodhull 1997)。该组织如图 4.2 所示。

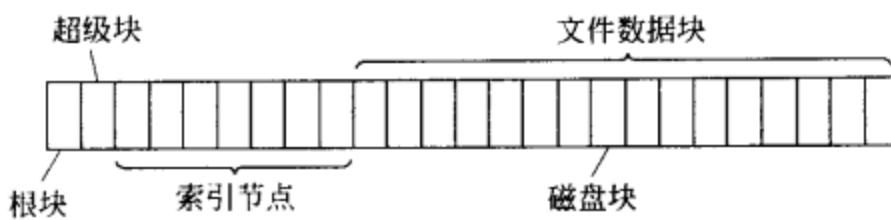


图 4.2 在逻辑磁盘的连续磁盘块上实现 UNIX 文件系统的一般组织结构

根块是一个特殊的数据和指令块，在系统启动时自动装载到主存储器中。根块用于将操作系统装载到主存储器中。

超级块保存整个文件系统的信息，如文件系统的大小，磁盘上还有哪些块没有分配，以及哪些信息节点还没有使用，等等。信息节点由索引号来指向，从数字 0 开始（数字 0 是为表示根目录的信息节点保留的）。

每个信息节点包含关于在磁盘上哪个位置可以找到与节点相关联的文件数据的准确信息。另外，一个信息节点包含关于其所有者、创建时间和最后一次修改的时间、保护信息等诸如此类的信息。因此，如果给定了一个信息节点的索引号，那么就可以访问到与其关联的文件。每个目录都作为一个文件来实现，对于根目录也是如此。它包含文件名与信息节点的索引号之间的映射。因此，信息节点的索引号可以看作相当于命名图中的节点标识符。

4.1.2 名称解析

名称空间为通过名称来存储和检索实体的信息提供了一套便利的机制。更一般地讲，给定一个路径名，应该能够查找出存储在由该名称所指向的节点中的任何信息。查询名称的过程称为名称解析(name resolution)。

为了解释名称解析是如何进行的，请考虑一个路径名，如 $N: < \text{label-1}, \text{label-2}, \dots, \text{label-n} \rangle$ 。该名称的解析是从命名图的节点 N 开始的，在其目录表中查询名称 label-1 ，并返回 label-1 所指向的节点的标识符。然后，解析过程在标识出的节点的目录表中继续查询名称 label-2 ，依此类推。假设该命名路径确实存在，那么解析会在由 label-n 指向的最后一个节点上停止，并返回该节点的内容。

名称查询从名称解析程序继续运行的地方返回节点的标识符。应特别指出的是，访问标识节点的目录表是必要的。再次考虑 UNIX 文件系统中的命名图。如前面所提到的，节点标识符是作为信息节点的索引号来实现的。访问目录表意味着首先要读取信息节点，以找到实际数据在磁盘上存储的位置，然后再读取含有目录表的数据块。

1. 终止机制

仅当我们知道如何启动以及在何处启动时才能进行名称解析。在我们的实例中，开始节点已经给出，而且我们假设已经访问了目录表。知道如何启动以及在何处启动名称解析通常称为终止机制(closure mechanism)。从本质上讲，终止机制处理从名称空间中选择初始节点，名称解析从初始节点开始(Radia 1989)。有时难以理解终止机制的原因

是它们必须是部分隐式的,而且进行比较时彼此截然不同。

例如,在 UNIX 文件系统命名图的名称解析中,根目录的信息节点是代表文件系统的逻辑磁盘的第一个信息节点。其实际的字节偏移量由超级块的其他字段中的值和操作系统中有关超级块的内部组织的硬性编码信息一起计算得出。

为了理解这个内容,请考虑文件名的字符串表示,如/home/steen/mbox。要解析该名称,需要事先访问适当命名图的根节点的目录表。作为根节点,节点自身不能被查询,除非它作为其他命名图中的不同节点来实现,譬如说 G。但是在这种情况下,需要事先访问 G 的根节点。因此,解析文件名需要事先实现一些机制,通过它们解析程序才可以启动。

一个完全不同的实例是使用字符串“0031204430784”。许多人都不知道如何处理这些数字,除非他们被告知这个数列是一个电话号码。该信息足够启动解析程序,特别是通过拨打号码。然后,电话系统来完成剩下的工作。

最后我们考虑一个实例,即在分布式系统中使用全局和局部名称。局部名称的一个典型实例是环境变量。例如,在 UNIX 系统中,名为 HOME 的变量用于查询用户的主目录。每个用户都有该变量的副本,它初始化为与用户的主目录的全局系统名称相对应。与环境变量相关联的终止机制通过在特定于用户的表中查询变量的名称以确保该名称可被适当地解析。

2. 链接和挂载

与名称解析紧密联系的是别名的使用。别名是同一实体的另一个名称。环境变量是别名的一个例子。在命名图中,主要有两种不同的方法来实现别名。第一个方法是允许多个绝对路径名来指向命名图中的同一节点。该方法如图 4.1 所示,其中节点 n5 可以由两个不同的路径名来查询。在 UNIX 术语中,图 4.1 中的两个路径名/keys 和/home/steen/keys 都称为到节点 n5 的硬链接。

第二种方法是用叶节点来表示实体(譬如说 N),而不是存储实体的地址或位置,该节点存储绝对路径名。当第一次解析指向 N 的绝对路径名时,名称解析将返回存储在 N 中的路径名,在这点它可以继续解析新的路径名。这个原理对应于 UNIX 文件系统中的符号链接,如图 4.3 所示。在这种情况下,指向包含绝对路径名/keys 的节点的路径名/home/steen/keys 是对节点 n5 的符号链接。

到目前为止,所介绍的名称解析都是在单个名称空间中进行的。但是,名称解析也可以用来以透明方式来合并不同的名称空间。首先,让我们考虑一个已挂载的文件系统。在我们的命名模型中,已挂载的文件系统于让目录节点存储来自不同名称空间(我们称之为外部名称空间)的目录节点的标识符。存储节点标识符的目录节点称为挂接点(mount point)。因此,外部名称空间中的目录节点称为挂载点 mounting point。通常,挂载点是名称空间的根。在名称解析的过程中,挂载点被查询,并通过访问目录表来完成解析。

挂载的原理可以推广到其他名称空间。值得注意的是,作为挂接点的目录节点存储识别和访问外部名称空间中的挂载点的所有必需的信息。在 Jade 命名系统(Rao 和 Peterson 1993)中遵循的是该方法,在许多分布式文件系统中实际上也是遵循该方法的。

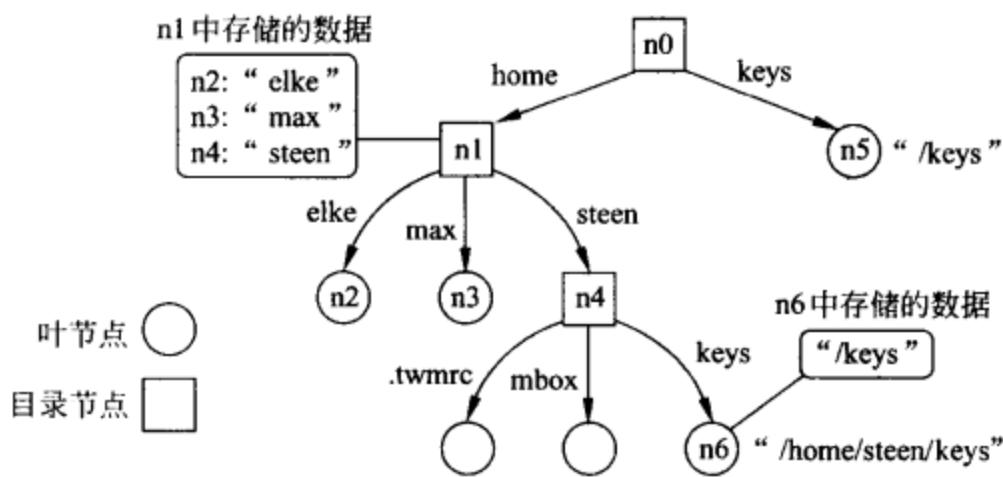


图 4.3 命名图中所讲述的符号链接的概念

请考虑分布在多台计算机上的名称空间的集合。特别是，每个名称空间由不同的服务器来实现，每个服务器可能运行在单独的计算机上。因此，如果想要将外部名称空间 NS2 挂载到名称空间 NS1 中，可能需要在网络上与 NS2 的服务器进行通信，因为服务器可能运行在不同的计算机上而不是 NS1 的服务器上。在分布式系统中挂载外部名称空间至少需要以下信息：

- (1) 访问协议的名称；
- (2) 服务器的名称；
- (3) 外部名称空间中挂载点的名称。

注意，这些节点每一个都需要解析。访问协议的名称需要解析成协议的执行，通过协议的执行可以与外部名称空间的服务器进行通信。服务器的名称需要解析成可以到达服务器的地址。作为名称解析中的最后部分，挂载点的名称需要解析成外部名称空间的节点标识符。

在非分布式系统中，不需要这三点中的任意一点。例如，在 UNIX 中，没有访问协议和服务器。同样，挂载点的名称也是不需要的，它仅是外部名称空间的根目录。

挂载点的名称由外部名称空间的服务器来解析。但是，我们还需要名称空间，以及访问协议的实现和服务器名称。一个可能性是将上面所列的三个名称表示为一个 URL。

为了使概念更具体，请考虑这样一种情况，使用笔记本电脑的用户想要访问远程文件服务器上存储的文件。客户计算机和文件服务器都使用 Sun 公司的网络文件系统 (NFS) 来配置，我们将在第 10 章中对 NFS 进行讨论。NFS 是一种分布式文件系统，它使用一种明确描述客户如何访问存储在远程 NFS 文件服务器上的文件的协议。特别是，如果允许 NFS 通过 Internet 来工作，那么客户可以通过 NFS URL 来确切指定它所要访问的文件，例如 nfs://flits.cs.vu.nl//home/steen。该 URL 命名 NFS 文件服务器 flits.cs.vu.nl 上名为/home/steen 的文件（对目录也是一样），客户可以通过 NFS 协议来访问它（Callaghan 2000）。

就如何解释 nfs 名称这个问题，在全世界范围内已经达成协议，从这个意义上来说，nfs 是一个众所周知的名称。换句话说，如果我们正在处理 URL，那么名称 nfs 将解析成 NFS 协议的实现。使用域名系统 (domain name system, DNS) 可以把服务器名称解析成

它的地址,我们将在后面章节中讨论域名系统。正如我们说过的那样,/home/steen 由外部名称空间的服务器进行解析。

图 4.4 部分显示了在客户机器上的文件系统的组织方式。根目录包含许多用户定义的条目,其中有名为/remote 的子目录。这个子目录用来包含外部名称空间的挂载点,例如 Vrije Universiteit 上的用户根目录。为此,可使用一个名为/remote/vu 的目录节点来存储 URL nfs://flits.cs.vu.nl/home/steen。

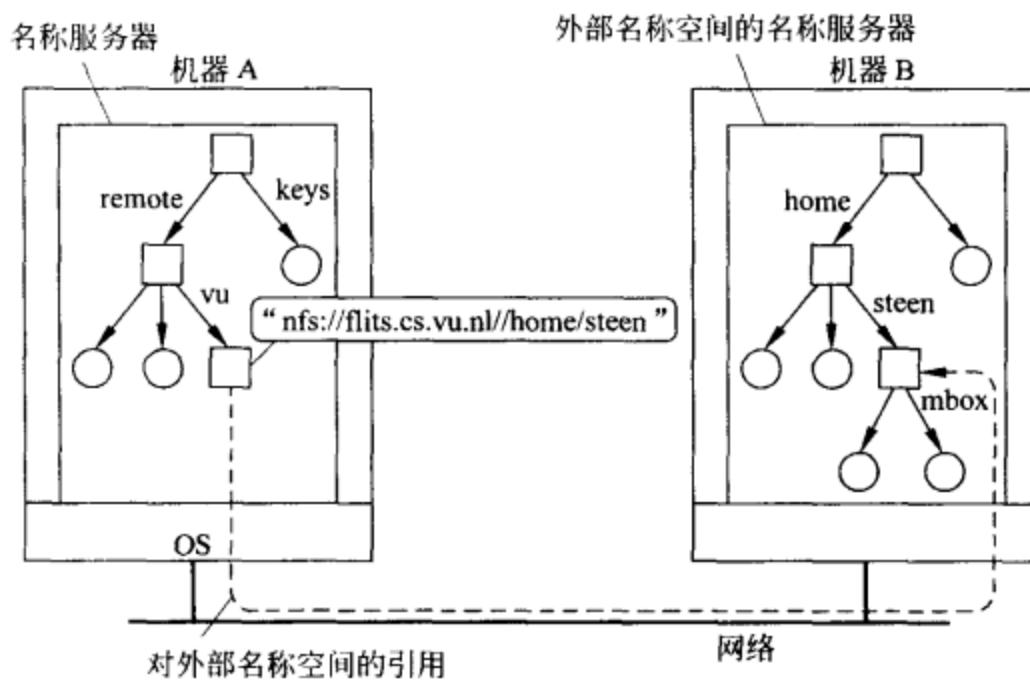


图 4.4 通过指定的访问协议装配远程名称空间

现在来考虑名称/remote/vu/mbox。在解析这个名称时,将从客户机器的根目录开始,一直到节点/remote/vu 为止。然后名称解析会返回 URL nfs://flits.cs.vu.nl//home/steen,接着会引导客户机器使用 NFS 协议与文件服务器 flits.cs.vu.nl 联系,随后会访问目录/home/steen。接下来的名称解析将读取该目录下的 mbox 文件。

举例来说,刚才描述的允许挂载远程文件系统的分布式系统允许客户机器执行如下命令:

```
cd /remote/vu  
ls-1
```

随后这些命令将列出远程文件服务器上/home/steen 目录下的文件。这样做的优点是用户不必了解访问远程服务器的实际细节。在理想情况下,与访问本地可用文件相比,仅仅存在一些性能损失。实际上,在客户看来,来自于本地机器的名称空间和来自远程机器上/home/steen 下的名称空间共同形式了一个单一的名称空间。

挂载是合并不同名称空间时所使用的一种方式,另一种方式来自 DEC 公司的全局名称服务(global name service, GNS)。它的做法是添加一个新的根节点,然后把现有的根节点变成它的子节点(Lamson 1986)。图 4.5 显示了这种方式,下面是对它的解释。

这种方式存在的一个问题就是需要修改现有的名称。举例来说,在图 4.5 中,要将名称空间 NS1 下的绝对路径名/home/steen 转换成一种相对路径名,这个相对路径名将从节

点 n0 开始解析,它对应于绝对路径名/vu/home/steen。为了解决这些问题并允许将来添加其他的名称空间,GNS 中的名称始终隐含地包括节点的标识符,解析通常会从该节点标识符开始。例如,在图 4.5 所示的名称空间 NS1 中,名称/home/steen/keys 总是被扩展以包括节点标识符 n0,结果就是 n0:/home/steen/keys。用户一般看不到扩展过程。这里假定节点标识符是全局惟一的。因此,即使是来自不同名称空间的节点也被假定为总是包含不同的节点标识符。

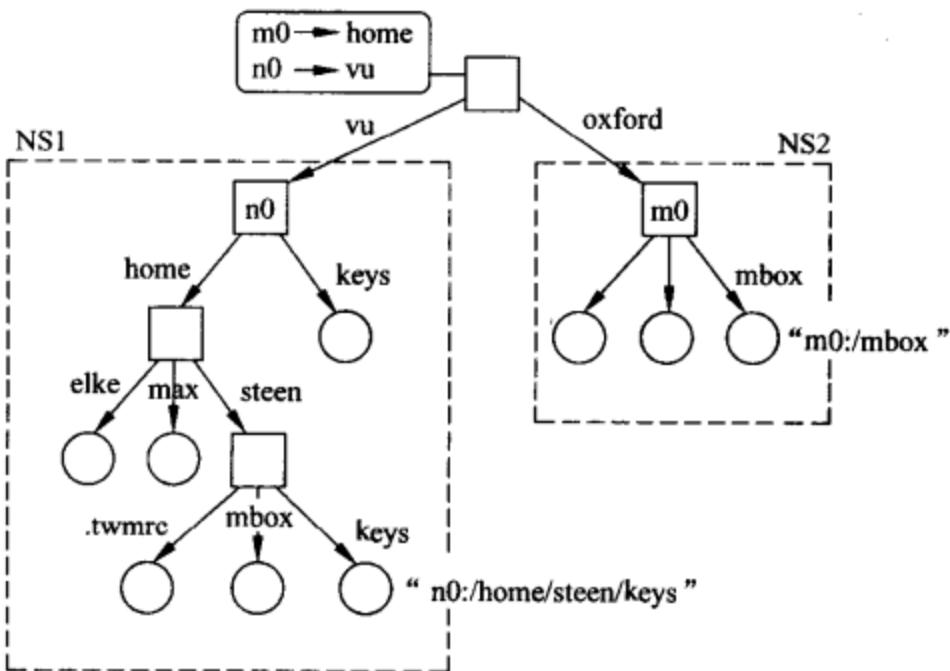


图 4.5 DEC GNS 的组织

现在在不改变现有名称的情况下合并 NS1 和 NS2 这两个名称空间,步骤如下。在添加新的根节点时,新节点保存了一张表,这张表把 NS1 根节点的标识符映射到该根节点在新名称空间中的名称,如图 4.5 所示。NS2 根节点也一样。通过始终从新名称空间中的根节点开始解析名称,可以首先把 n0:/home/steen 这样的名称转换为/vu/home/steen,这是通过在根节点的表中查找根节点的标识符 n0 来实现的。

GNS 存在一个潜在问题是:在合并后的名称空间中,根节点需要维护原根节点的标识符到新名称之间的映射。如果合并了几个名称空间,那么这种方式最终将导致性能问题。

4.1.3 名称空间的实现

名称空间形成了命名服务的核心,准确地说,命名服务就是一种允许用户和进程添加、删除和查找名称的服务。命名服务由名称服务器实现。如果一个分布式系统局限于局域网,那么可以只使用一台名称服务器来实现命名服务。可是,大型分布式系统包含许多实体,可能会跨越很大的地理区域,在这样的系统中,需要使用多台名称服务器来分散名称空间的实现。

1. 名称空间分类

在大型的、可能是世界范围的分布式系统中,名称空间通常分层组织。和以前一样,我们仍然假定这样的名称空间只有一个根节点。为了有效地实现这样的名称空间,把它

分成逻辑上的层会很方便。(Cheriton 和 Mann 1989)把名称空间区分为以下三层。

全局层(global layer)由最高级别的节点组成,即由根节点以及其他逻辑上靠近根节点的目录节点,也就是根节点的子节点组成。全局层的特点就是它通常是稳定的,也就是说目录表很少改变。这样的节点可以代表组织或组织群,这是因为它们的名称被存储在这个名称空间中。

行政层(administrational layer)由那些在单个组织内一起被管理的目录节点组成。行政层中的目录节点所具有的特点是它们代表属于同一组织或行政单位的实体组。举例来说,这样的目录节点可能是一个组织中每个部门都拥有的目录节点,或者是通过它可以找到所有主机的目录节点。还有可以用来命名所有用户起点的目录节点,等等。行政层的节点是相对稳定的,尽管说它们的改变一般要比全局层节点频繁得多。

最后是管理层(managerial layer),它由那些经常改变的节点组成。例如,代表本地网络主机的节点就属于这一层。同样,本层还包括那些代表共享文件(如库文件或二进制文件)的节点。另一类重要节点包括那些代表用户定义目录和文件的节点。与全局层和行政层相比,管理层的节点不仅要由系统管理员维护,而且还要由分布式系统的各个最终用户维护。

为了更具体地介绍这些概念,图 4.6 显示了一个例子,它说明了把 DNS 名称空间划分为若干部分的方法,该空间包括了组织内文件的名称,可以通过 Internet 访问这些文件,如 Web 页面和可传输文件等。名称空间划分成不重叠的几部分,在 DNS 中称为区域(Zone)(Mockapetris 1987)。区域是名称空间的一部分,它是由单独的名称服务器实现的。图 4.6 展示了一些区域。

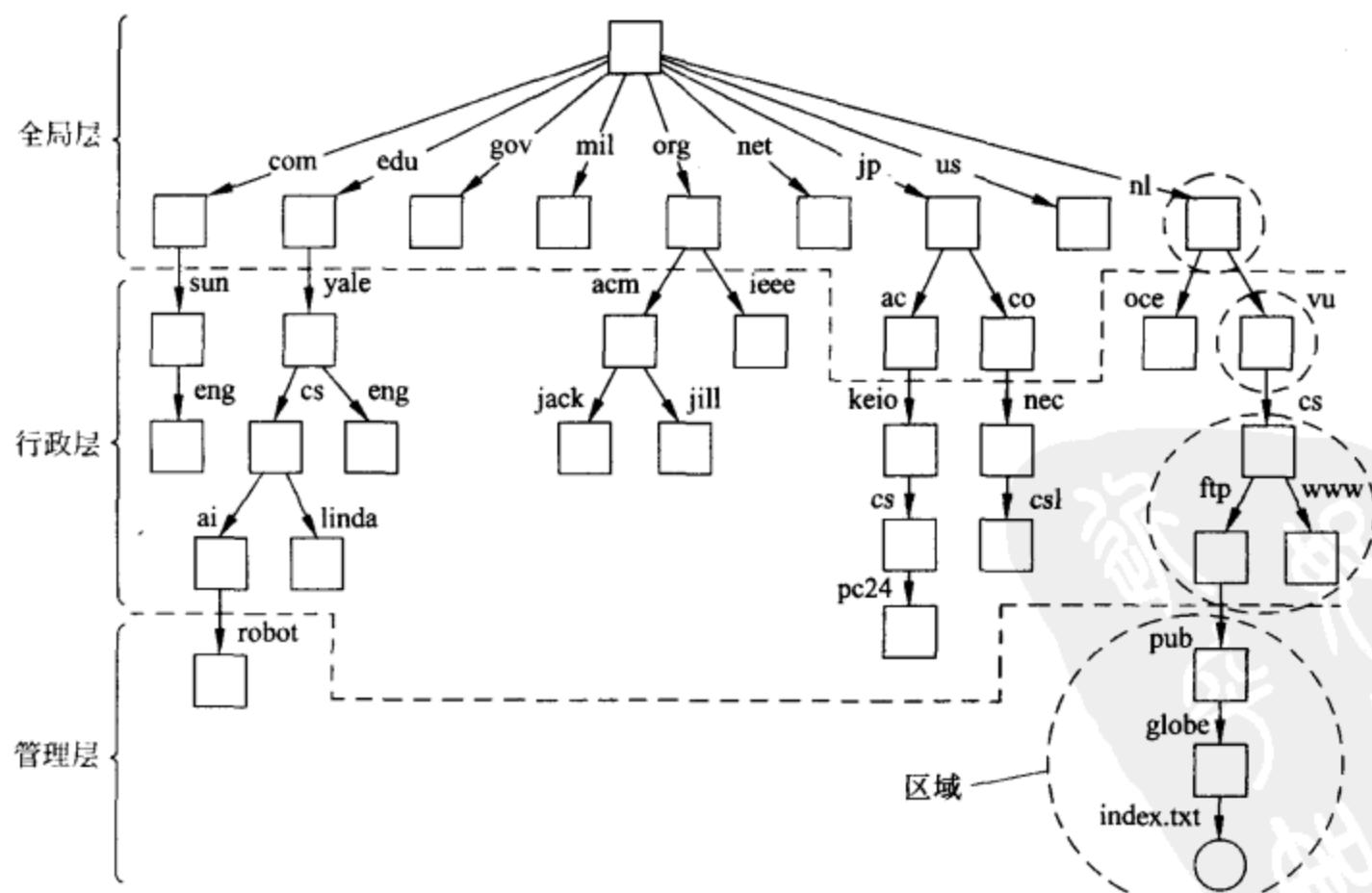


图 4.6 DNS 名称空间划分示例,该空间包括了可以从 Internet 上访问的文件,共分三层

在可用性和性能方面,每层的名称服务器都必须满足不同的要求。在全局层中,良好的可用性对名称服务器特别重要。如果名称服务器发生故障,那么名称空间中很大部分将无法到达,这是因为不能越过发生故障的服务器进行名称解析。

性能问题有点微妙。根据全局层节点变化不频繁的特点,查找操作的结果一般会长期有效。因此,客户可以有效地缓存(也就是本地存储)这些结果。当再次执行同样的操作时,可以从客户的缓存中直接获取结果,而不用让名称服务器返回结果。因此,全局层的名称服务器并不需要快速响应单一的查询请求。相反,重要的可能是其吞吐能力,特别是在拥有上百万用户的大型系统中。

将复制服务器与客户端缓存结合起来可以满足全局层名称服务器在可用性和性能方面的要求。正如我们将在第 6 章讨论的那样,这一层的更新通常不需要立即生效,让复制保持一致会使事情更为轻松一些。

对于与名称服务器同在一个组织内的客户来说,行政层名称服务器的可用性是最为重要的。如果名称服务器发生故障,那么组织内的许多资源将因无法查找而变得不可到达。然而,对于组织外的用户来说,暂时无法到达组织内的资源可能没有那么重要。

在性能方面,行政层的名称服务器与全局层的名称服务器有类似的特点。由于节点的变化不是经常发生,所以缓存查询结果可能是非常有效的,这使得性能不那么重要。不过,与全局层相比,行政层应该注意查询结果在几 ms 内返回的情况,不管是直接从服务器返回,还是从客户的本地缓存返回。同样,更新处理一般应该比全局层更迅速。举例来说,需要几个小时来让新用户账号生效是不能接受的。

通过使用高性能的机器来运行名称服务器,上述要求一般可以得到满足。另外,应该采用客户端缓存的方法,并与复制相结合,从而提高整体可用性。

对管理层名称服务器的可用性要求一般比较低。特别是,使用一台专用的机器运行名称服务器来应付临时不可用的风险通常已经足够了。不过,在这里,性能是关键。用户希望操作能够立即完成。由于经常进行更新,所以客户端缓存一般不太有效,除非采用特殊措施,第 6 章将对此进行讨论。

图 4.7 显示了不同层的名称服务器之间的比较。在分布式系统中,全局层和行政层的名称服务器非常难以实现,困难来自于复制和缓存。复制和缓存对于提高可用性和性能来说是必需的,但它们也导致了一致性问题的出现。跨越广域网络的缓存和复制使问题进一步恶化,这是因为广域网络造成了较长的通信延迟,从而使同步变得更加困难。第 6 章将全面地讨论了复制和缓存。

内容	全局层	行政层	管理层
网络的地理范围	世界范围	组织	部门
节点数量	少	许多	极多
查询响应	s	ms	立即
更新的传播情况	延迟	立即	立即
复制数量	许多	没有或很少	没有
是否采用客户端缓存	是	是	有时

图 4.7 为大型名称空间所属的全局层、行政层和管理层实现节点的名称服务器之间的比较

2. 名称解析的实现

在多个名称服务器上名称空间的分散性影响了名称解析的实现。为了说明名称解析在大型名称服务中的实现,我们假定不复制名称服务器,也不使用客户端缓存。每个客户都使用本地的名称解析程序,名称解析程序负责确保名称解析过程得以执行。请参考图 4.6,假定绝对的路径名

root:<nl,vu,cs,ftp,pub,globe,index.txt>

将被解析。如果使用 URL 表示法,那么这个路径名将对应于 `ftp://ftp.cs.vu.nl/pub/globe/index.txt`。现在可以用两种方法实现名称解析。

在迭代名称解析中,名称解析程序把完整的名称转发给根名称服务器。假定可以联系根服务器的地址是众所周知的。根服务器会尽量深入地解析路径名,然后把结果返回给客户。在我们的例子中,根服务器只能解析标识符 nl,然后根服务器将返回与 nl 相关的名称服务器所用的地址。

接着,客户会把剩下的路径名(也就是 `nl:<vu,cs,ftp,pub,globe,index.txt>`)发送给一台名称服务器。这台服务器只能解析标识符 vu,然后返回相关名称服务器的地址,同时剩下路径名 `vu:<cs,ftp,pub,globe,index.txt>`。

接下来客户的名称解析程序将同下一台名称服务器联系,该服务器响应的结果是解析标识符 cs,随后还有 ftp,最后会连同路径名 `ftp:<pub,globe,index.txt>`一起返回该 FTP 服务器的地址。客户端接着再与该 FTP 服务器联系,请求它解析原始路径名的最后一部分。随后 FTP 服务器会解析标识符 pub、globe 和 index.txt,并发送被请求的文件(这里是使用 FTP)。图 4.8 显示了这个迭代名称解析过程。符号 #<cs>用来指明一台服务器的地址,该服务器负责处理<cs>涉及到的节点。

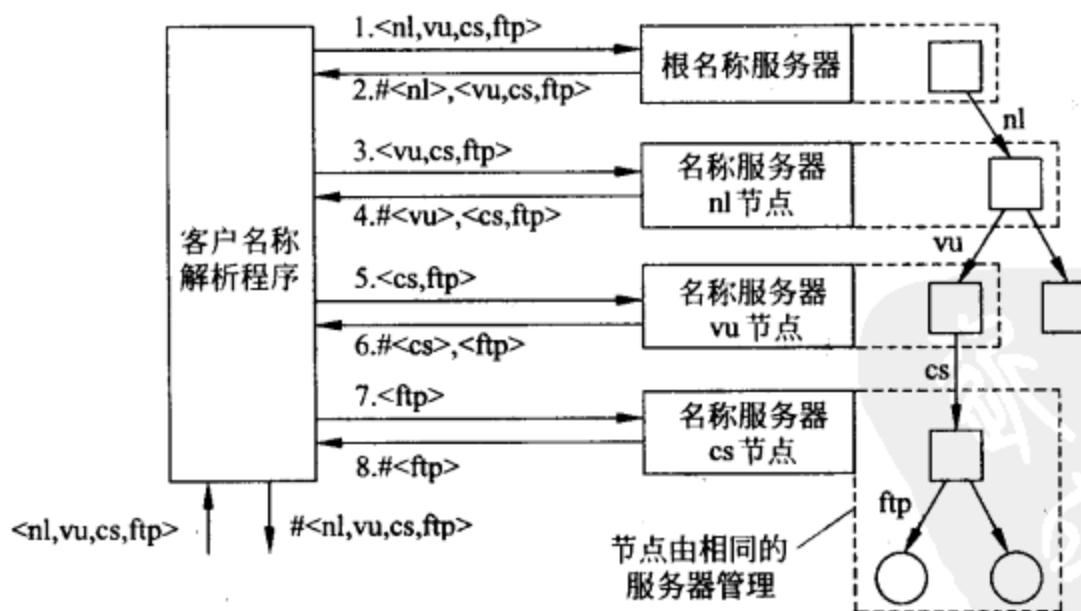


图 4.8 迭代名称解析原理

实际上,最后一步,即联系 FTP 服务器并请求它发送路径名 `ftp:<pub,globe,index.txt>` 所指文件的那一步,是由客户进程独立完成的。换句话说,客户通常只把路

径名 root:<nl,vu,cs,ftp>转交给名称解析程序,希望从名称解析程序那里得到可以联系的那一台 FTP 服务器的地址,图 4.8 也说明了这一点。

迭代名称解析(recursive name resolution)的一种替代方法是在名称解析过程中使用递归。与返回客户解析程序每个中间结果不同,在使用递归名称解析时,名称服务器会把结果传递给它找到的下一台服务器。举例来说,当根名称服务器找到实现 nl 节点的名称服务器所用的地址后,它会请求该名称服务器解析路径名 nl:<vu,cs,ftp,pub,globe,index.txt>。在同样使用递归名称解析的情况下,下一台服务器将解析完整的路径名,最终会把 index.txt 文件返回根服务器,根服务器再依次把该文件传递给客户的名称解析程序。

图 4.9 说明了递归名称解析。和迭代名称解析一样,最后的解析步骤,即联系 FTP 服务器并请求它传送指定文件的那一步,一般情况下是由客户的进程独立实现。

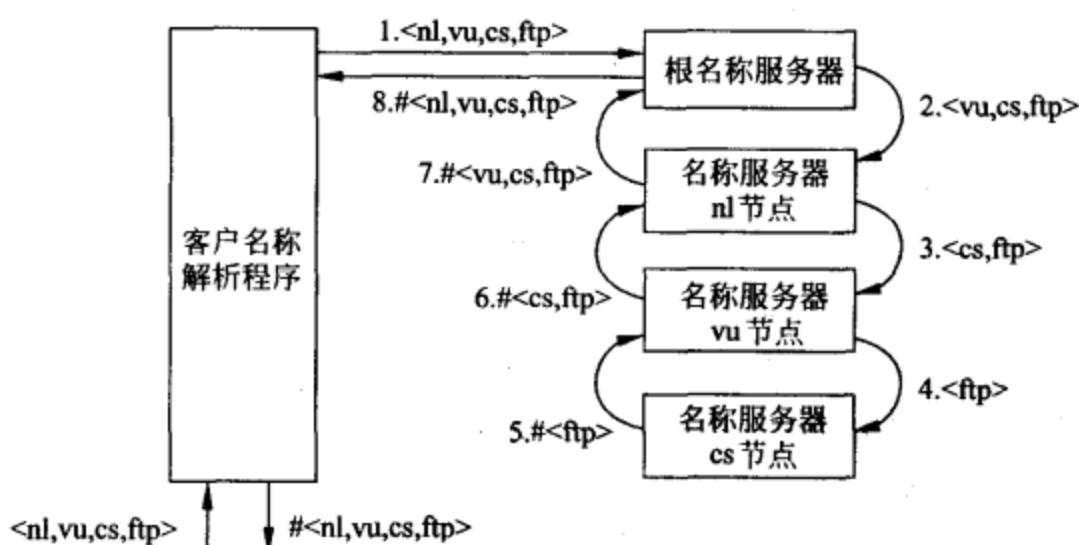


图 4.9 递归名称解析原理

递归名称解析的主要缺点是它要求每台名称服务器都具有较高的性能。本质上,需要名称服务器对路径名进行完整的解析,尽管说它可能会与其他名称服务器一起完成这项工作。这种额外的负担非常大,以至于在名称空间的全局层中,名称服务器只支持迭代名称解析。

递归名称解析拥有两个重要的优点。第一个优点就是与迭代名称解析相比,递归名称解析的缓存结果要更为有效一些。第二个优点是可以减少通信开销。为了说明这些优点,我们假定客户的名称解析程序接受的路径名只涉及名称空间中全局层或行政层节点。为了解析路径名中对应于管理层节点的那一部分,客户将独立联系名称解析程序返回的名称服务器,正如我们上面讨论的那样。

递归名称解析让名称服务器逐步负责实现更底层节点名称解析的名称服务器的地址。其结果是,可以有效地使用缓存来提高性能。例如,在请求根服务器解析路径名 root:<nl,vu,cs,ftp>时,根服务器最终会得到一台服务器的地址,这台服务器负责实现那个路径名所涉及的节点的名称解析。为了达到这一点,实现 nl 节点的名称服务器必须查找实现 vu 节点的名称服务器所用的地址,而后者必须查找实现 cs 节点的名称服务器所用的地址。

由于全局层和行政层节点不会经常改变,所以根名称服务器可以有效地缓存返回的地址。此外,由于地址也被递归返回负责实现 vu 节点以及 nl 节点的名称服务器,所以同样可以在这些服务器上缓存地址。

同样,也可以返回和缓存中间名称查询结果。例如,实现 nl 节点的服务器必须查找实现 vu 节点的服务器的地址。当实现 nl 的服务器返回原始名称的查询结果时,可以把地址返回根服务器。图 4.10 显示了解析过程的完整概括,以及各个名称服务器可以缓存的结果。

服务器所在的节点	需要解析的标识符	查询	传递给下一个服务器	递归与缓存	返回请求者
cs	<ftp>	#<ftp>	--	--	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs,ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

图 4.10 对<nl,vu,cs,ftp>进行的递归名称解析。名称服务器缓存用于后续查询的中间结果

从根本上说,这种方法的优点就是可以非常有效地完成查询操作。例如,假设另一个客户后来请求解析路径名 root:<nl,vu,cs,flits>。该名称被传递给根服务器,根服务器可以立即把它转发给用于 cs 节点的名称服务器,并请求它解析剩下的路径名 cs:<flits>。

如果使用迭代名称解析,那么就需要把缓存限制在客户的名称解析程序上。因此,如果客户 A 请求解析一个名称,而另一个客户 B 随后又请求解析同一个名称,那么 B 请求的名称解析所经过的名称服务器将与 A 相同。作为一种折衷方案,许多组织都使用一台本地的中间名称服务器,所有客户都共享它。这台本地名称服务器处理所有的名称请求并缓存结果。从管理的角度来看,使用这样一台中间服务器也是很方便的。例如,只有这台服务器才知道根服务器所在的位置,其他机器都不需要这种信息。

递归名称解析的第二个优点就是它在通信方面的开销通常较小。同样,设想要解析路径名 root:<nl,vu,cs,fp>,并且假设客户位于旧金山。假设客户知道用于 nl 节点的服务器的地址,在使用递归名称解析的情况下,通信路线从位于旧金山的客户主机开始到位于 The Netherlands 的 nl 服务器为止,如图 4.11 中的 R1 所示。从那里开始,依次需要 nl 服务器与 Vrije Universiteit 名称服务器(位于 The Netherlands 的 Amsterdam 一家大学的校园内)之间的通信。它们之间的通信如图 4.11 中的 R2 所示。最后,需要 vu 服务器与位于 Computer Science Department 的名称服务器之间的通信,如图 4.11 中的 R3 所示。返回路线是一样的,不过方向相反。很明显,决定通信开销的是客户主机与 nl 服务器之间的信息交换。

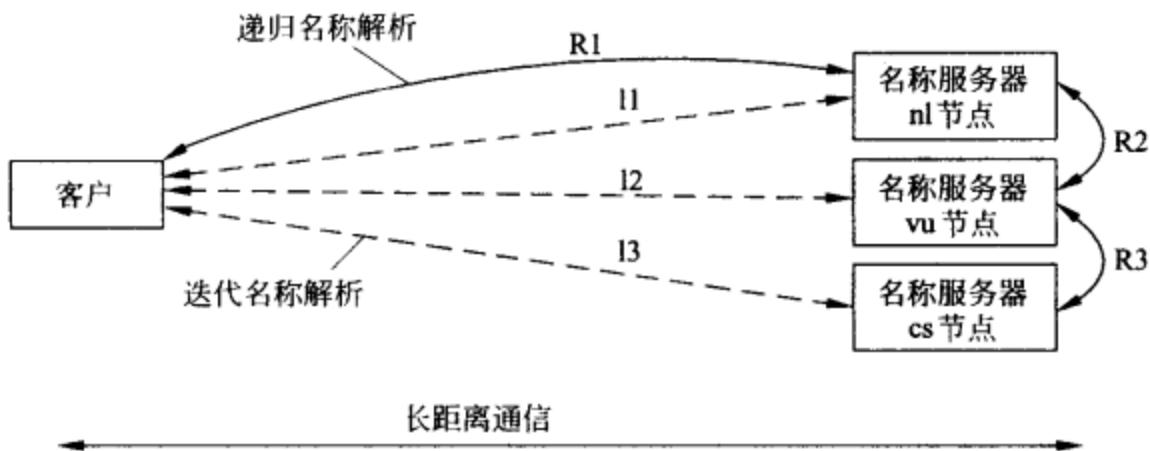


图 4.11 递归名称解析与迭代名称解析在通信开销方面的对比

与之相反，在使用迭代名称解析的情况下，客户主机必须分别与 nl、vu 和 cs 等服务器进行通信，这种通信的总开销大约是递归名称解析的通信开销的 3 倍。图 4.11 中标有 I1、I2 和 I3 的箭头显示了迭代名称解析的通信路线。

4.1.4 示例：域名系统

当今使用的最大分布式名称服务之一是 Internet 域名系统 (domain name system, DNS)。DNS 主要用来查找主机地址和邮件服务器。在下面的内容中，我们将集中介绍 DNS 名称空间的组织，以及存储在其节点上的信息。另外，我们还要深入了解 DNS 的实际实现。有关这方面的更多信息，请参考文献 (Mockapetris 1987; Albitz 和 Liu 1998)。

1. DNS 名称空间

DNS 名称空间是分层组织的，就像一棵有根的树。标识符是区分大小写的字符串，字符串由字母和数字组成。标识符的最大长度是 63 个字符，完整路径名的最大长度是 255 个字符。路径名的字符串表示由一列标识符组成，从最右边开始，使用圆点“.”分隔各个标识符。根由圆点代表。举例来说，路径名 root:<nl, vu, cs, flits>用字符串 flits. cs. vu. nl 表示，这个字符串包括最右边的圆点，该圆点象征着根节点。为了方便阅读，我们通常省略这个圆点。

由于 DNS 名称空间的每个节点都包含一个进入边（根节点除外，它不包含进入边），所以节点进入边的标识符也可用作该节点的名称。子树被称为域，指向根节点的路径名称为域名。注意，域名可以是绝对的，也可以是相对的，就像路径名一样。

节点的内容由一组资源记录组成。有各种不同类型的资源记录，图 4.12 列出了最重要的一些记录。

DNS 名称空间的节点经常同时代表几个实体。例如，像 vu.nl 这样的域名可以用来表示域和区域。在这种情况下，域由几个区域实现。

SOA (start of authority, 即授权起点) 资源记录包含的信息有：负责区域的系统管理员的电子邮件地址，可以从它上面获取区域数据的主机名称，等等。

记录类型	相关实体	描述
SOA	区域	容纳表示区域的信息
A	主机	容纳节点所代表主机的 IP 地址
MX	域	指向邮件服务器,该服务器处理发往该节点的邮件
SRV	域	指向处理某种服务的服务器
NS	区域	指向实现指定区域的名称服务器
CNAME	节点	包含指定节点主名称的符号链接
PTR	主机	容纳主机的规范名称
HINFO	主机	容纳节点所代表主机的信息
TXT	任意类型	容纳任何被认为有用的、特定于实体的信息

图 4.12 形成 DNS 名称空间节点内容的一些最重要的资源记录类型

A(address, 即地址)资源记录代表 Internet 上的特定主机。A 记录包含允许通信的主机所用的 IP 地址。如果主机拥有多个 IP 地址,那么与多宿主计算机一样,该节点将包含每个地址的 A 记录。

MX(mail exchange, 即邮件交换)记录是一种重要的资源记录类型,它本质上是指向代表一台邮件服务器的节点的符号链接。例如,代表域 cs. vu. nl 的节点拥有一条包含名称 zephyr. cs. vu. nl 的 MX 记录,名称 zephyr. cs. vu. nl 指向了一台邮件服务器。该服务器会处理所有发给域 cs. vu. nl 用户的邮件。一个节点中可以存储多条 MX 记录。

SRV 记录与 MX 记录有关,SRV 记录包含处理特定服务的服务器名称。(Vixie 1996)描述了 SRV 记录。服务本身由名称和协议名一起标识。例如,cs. vu. nl 域的 Web 服务器可以用一条类似于 http. tcp. cs. vu. nl 的 SRV 记录来命名。这条记录指向服务器的真正名称(即 soling. cs. vu. nl)。

代表区域的节点包含一条或多条 NS(name server, 名称服务器)记录。和 MX 记录类似,NS 记录包含一个服务器的名称,该服务器实现该节点代表的区域。原则上,名称空间的每个节点都可以存储一条 NS 记录,这条 NS 记录指向实现节点的名称服务器。不过,正如我们下面将要讨论的那样,在 DNS 名称空间的实现中,只有代表区域的节点才需要存储 NS 记录。

DNS 区分别名和规范名称(canonical name)。每个主机都假定拥有一个规范名称,或者说主名。别名由存储 CNAME 记录的节点实现,CNAME 记录包含主机的规范名称。因而存储这种记录的节点名称与符号链接相同,如图 4.3 所示。

DNS 使用 PTR(pointer, 指针)记录来维护 IP 地址到主机名称之间的反向映射。为了在只有 IP 地址的情况下提供主机名称查询,DNS 维护着一个名为 in-addr. arpa 的域,这个域中的节点代表 Internet 主机,这些节点使用代表主机的 IP 地址命名。例如,假设主机 www. cs. vu. nl 的 IP 地址是 130. 37. 24. 11。DNS 会创建一个名为 11. 24. 37. 130. in-addr. arpa 的节点,这个节点用来在 PTR 记录中存储主机的规范名称(正好是 soling. cs. vu. nl)。

最后两种记录类型是 HINFO 记录和 TXT 记录。HINFO(host info, 主机信息)记录用来存储主机的其他信息,如机器类型和操作系统等。与此类似,TXT 记录用来存储任意类型的数据,存储这些节点所代表实体的有关信息对用户会有所帮助。

2. DNS 的实现

DNS 的实现与我们在前面小节中描述的那些实现方式很类似。本质上,DNS 名称空间可以划分成如图 4.6 所示的全局层和行政层。管理层一般由本地文件系统组成,在形式上它不是 DNS 的一部分,因此也不使用 DNS 来管理。

每个区域都由一个名称服务器来实现,出于可用性考虑,这个名称服务器常常被复制。区域更新一般由主名称服务器完成。通过修改主服务器的 DNS 数据库,可以完成更新操作。辅名称服务器不直接访问该数据库,相反,它请求主服务器把内容传送给它。后一种做法在 DNS 术语中称为区域传送。

DNS 数据库由一组(数量很少)文件实现,其中最重要的一个文件包含了特定区域中所有节点的所有资源记录。这种方法允许只用节点的域名来标识节点,这样,节点标识符就变成了一个(隐含的)文件索引。

为了更好地理解这些实现方式,图 4.13 显示了一个文件的一部分,这个文件包含了有关 cs. vu. nl 域的大多数信息要注意,为了直观,我们对该文件进行了编辑。该文件显示了 8 个不同节点的内容,这些节点是 cs. vu. nl 域的一部分,在这个域中,每个节点都由它的域名来标识。

名称	记录类型	记录值
cs. vu. nl	SOA	star 1999121502,7200,3600,2419200,86400
cs. vu. nl	NS	star. cs. vu. nl
cs. vu. nl	NS	top. cs. vu. nl
cs. vu. nl	NS	solo. cs. vu. nl
cs. vu. nl	TXT	"Vrije Universiteit-Math. & Comp. Sc."
cs. vu. nl	MX	1 zephyr. cs. vu. nl
cs. vu. nl	MX	2 tornado. cs. vu. nl
cs. vu. nl	MX	3 star. cs. vu. nl
star. cs. vu. nl	HINFO	Sun Unix
star. cs. vu. nl	MX	1 star. cs. vu. nl
star. cs. vu. nl	MX	10 zephyr. cs. vu. nl
star. cs. vu. nl	A	130.37.24.6
star. cs. vu. nl	A	192.31.231.42
zephyr. cs. vu. nl	HINFO	Sun Unix
zephyr. cs. vu. nl	MX	1 zephyr. cs. vu. nl
zephyr. cs. vu. nl	MX	2 tornado. cs. vu. nl
zephyr. cs. vu. nl	A	192.31.231.66
www. cs. vu. nl	CNAME	soling. cs. vu. nl
ftp. cs. vu. nl	CNAME	soling. cs. vu. nl
soling. cs. vu. nl	HINFO	Sun Unix
soling. cs. vu. nl	MX	1 soling. cs. vu. nl
soling. cs. vu. nl	MX	10 zephyr. cs. vu. nl
soling. cs. vu. nl	A	130.37.24.11
laser. cs. vu. nl	HINFO	PC MS-DOS
laser. cs. vu. nl	A	130.37.30.32
vucs-das. cs. vu. nl	PTR	0.26.37.130.in-addr.arpa
vucs-das. cs. vu. nl	A	130.37.26.0

图 4.13 区域 cs. vu. nl 使用的 DNS 数据(摘录)

节点 cs. vu. nl 既代表区域,也代表域。它的 SOA 资源记录包含了该文件的有效性信息,以后我们不必再关心这个文件。这个区域有三个名称服务器,可使用 NS 记录中的规范主机名引用它们。TXT 记录用来提供这个区域的其他信息,不过任何一台名称服务器都不能自动对它们进行处理。此外,还有三个邮件服务器,它们可以处理发给该域用户的邮件。邮件服务器名称前面的数字指定了选择顺序。发送邮件的服务器始终会首先与编号最小的邮件服务器联系,在这个例子中,也就是 zephyr. cs. vu. nl。

主机 star. cs. vu. nl 作为这个区域的一个名称服务器运行。名称服务器对于任何名称服务都是很关键的。可以看到,为了增强这个服务器的健壮性,提供了两个分开的网络接口,每一个都由一条独立的 A 资源记录来代表。使用这种方式可以避免网络连接中断带来的影响。

接下来的 4 行提供了有关邮件服务器的必要信息。要注意,这台邮件服务器也由其他邮件服务器提供备份,其路径是 tornado. cs. vu. nl。

下面 4 行显示了一种典型的配置,一台单独的机器使用这种配置实现了部门的 Web 服务器和 FTP 服务器,这台机器名为 soling. cs. vu. nl。通过在同一台机器上运行这两种服务器(本质上是只把该机器用于 Internet 服务),系统管理可以变得更为简单一些。例如,两种服务器将包含相同的文件系统视图,从效率角度考虑,文件系统的一部分可以在 soling. cs. vu. nl 上实现。这种方法通常应用于 WWW 和 FTP 服务中。

下面两行显示的信息与一台连接到本地网络的激光打印机有关。最后两行演示了从地址到规范名称的反向映射。在这个例子中,该部门的一台超级计算机的名称可以通过它在 in-addr. arpa 域的地址来查询。

因为 cs. vu. nl 域被实现为单个区域,所以图 4.13 不包含对其他区域的引用。对在另一个区域里实现的子域中的节点,其引用方法如图 4.14 所示。所需要做的就是指定该子域的名称服务器,只要提供它的域名和 IP 地址就可以完成这项工作。在为 cs. vu. nl 域中的节点解析名称时,名称解析过程将在某一点上持续进行,方法就是读取 DNS 数据库,该数据库存储在 cs. vu. nl 域的名称服务器上。

名 称	记录类型	记录值
cs. vu. nl	NS	solo. cs. vu. nl
solo. cs. vu. nl	A	130. 37. 24. 1

图 4.14 vu. nl 域的部分描述,vu. nl 域包含 cs. vu. nl 域

4.1.5 示例: X.500

DNS 是一个传统命名服务方面的例子:在指定一个(可能是分层结构)名称的情况下,DNS 把该名称解析成一个命名图中的节点,然后以资源记录的形式返回该节点的内容。从这种意义上说,可以把 DNS 比作一部用来查找电话号码的电话簿。

目录服务采用一种不同的方法。目录服务是一种特殊类型的名称服务,在这种服务中,客户可以基于属性描述来查找实体,而不用基于完整的名称。这种方式与人们在需要找人来修理打破的窗户时使用黄页的方式非常类似。在这种情况下,用户可以在标题“窗

户修理”下查找,以便获得修理窗户的店铺(名称)列表。

在这一节中,我们要简单地介绍 OSI X.500 目录服务。尽管这种目录服务已经使用十几年了,但是直到最近它的轻量级版本被实现成 Internet 服务以后,它才被广泛接受。有关 X.500 的详细信息,请参阅文献(Chadwick 1994; Radicati 1994)。有关各种目录服务(包括 X.500)的实用信息,请参阅文献(Sheresh 和 Sheresh 2000)。

1. X.500 名称空间

从概念上来说,X.500 目录由许多记录组成,这些记录通常称为目录项。X.500 中的目录项可以比作 DNS 中的资源记录。每条记录都由一组(属性,值)值对组成,其中每个属性都有一个对应的类型。属性之间有单值属性和多值属性之分。后者通常代表数组和列表。作为例子,图 4.15 显示了一些简单的目录项,这些目录项标识了图 4.13 所显示的一些通用服务器的地址。

属性	缩写	值
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Math. & Comp. Sc.
CommonName	CN	Main server
Mail_Servers	—	130.37.24.6,192.31.231.42,192.31.231.66
FTP_Server	—	130.37.24.11
WWW_Server	—	130.37.24.11

图 4.15 X.500 目录项方面的简单示例,这些目录使用 X.500 命名约定。

在我们的例子中,使用了一个 X.500 标准描述的命名约定,这个约定适用于头 5 个属性。Organization 和 OrganizationalUnit 属性分别描述与存储在记录中的数据相关联的组织和部门。同样,属性 Locality 和 Country 提供目录项存储位置方面的信息。CommonName 属性通常用来作为一个不确定的名称,这个名称用来标识目录中限定部分的目录项。举例来说,在为其他 4 个属性 Country、Locality、Organization 和 OrganizationalUnit 都提供指定值的情况下,名称“Main servers”足以找到我们的示例目录项。在我们的例子中,只有属性 Mail_Servers 拥有多个与它相关联的值。所有其他属性都只有一个值。

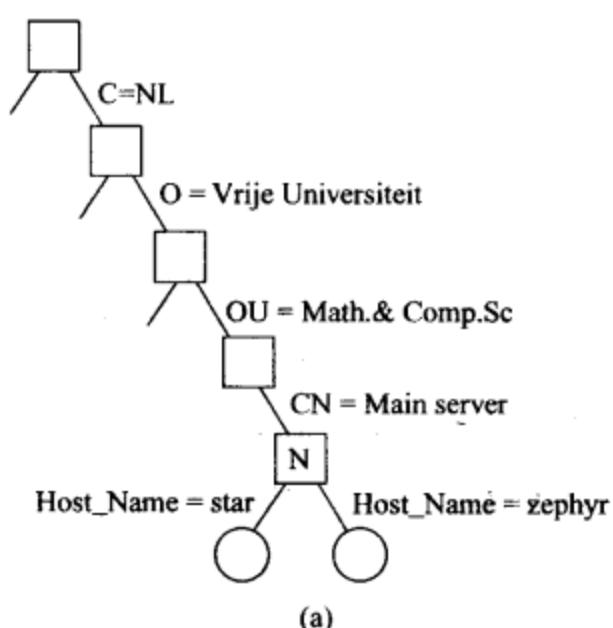
在 X.500 目录服务中,所有目录项的集合称为 DIB(directory information base, 目录信息库)。DIB 的一个重要特征就是每条记录都被唯一地命名,以便能够查找它。这样一种全局唯一的名称表现为每条记录的命名属性所组成的序列。每个命名属性称为相对可识别名称(relative distinguished name),或者简称为 RDN。在图 4.15 所示的例子中,前 5 个属性全部是命名属性。可通过使用约定的缩写形式来代表 X.500 中的命名属性,就像图 4.15 所示的那样,属性 Country、Organization 和 OrganizationalUnit 可以用来构成全局唯一名称:

/C=NL/O=Vrije Universiteit/OU=Math. & Comp. Sc.

这类似于 DNS 名称 nl.vu.cs。

和 DNS 类似,使用 RDN 序列组成的全局惟一名称导致了目录项集合的分层,这种分层也被称为 DIT(directory information tree,目录信息树)。DIT 本质上形成了 X.500 目录服务的命名图。在这种命名图中,每个节点代表一个目录项。另外,节点可以作为传统意义上的目录,在这种情况下,一个父节点可以拥有几个子节点。为了便于理解,请参见图 4.16(a)所显示的部分命名图。

节点 N 对应于图 4.15 所显示的目录项。同时,这个节点还是许多其他目录项的父节点,这些其他的目录项拥有另外的命名属性 Host_Name,这个命名属性被用来作为一个 RDN。举例来说,这类的目录项可以用来代表图 4.16(b)所示的主机。



(a)

属性	取值
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Math. & Comp. Sc.
CommonName	Main server
Host_Name	Star
Host_Address	192.31.231.42

属性	取值
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Math. & Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	192.31.231.66

(b)

图 4.16 目录信息树和目录项

(a) 目录信息树的一部分; (b) 使用 Host_Name 作为 RDN 的两个目录项

这样,X.500 命名图中的节点既可以代表 X.500 记录,又可以代表传统意义上的目录,就像我们刚才讨论过的那样。区别是它们要用两种不同的查询操作来支持。read 操作作用在指定 DIT 路径名的情况下读取一条记录,而 List 操作则用来列出指定 DIT 节点的所有外出边名称。每个名称都对应于一个指定节点的子节点。注意,List 操作不返回任何记录,它只返回名称。换句话说,使用如下名称调用 read 操作:

/C=NL/O=Vrije Universiteit/OU=Math. & Comp. Sc./CN=Main server

将返回图 4.15 所示的记录,而调用 List 操作将返回名称 star 和 zephyr,它们来自图 4.16(b) 所示的实体,同时还会返回已经以类似方式注册的其他主机所用的名称。

2. X.500 的实现

除了 X.500 支持更多的查询操作之外(稍后我们将对此进行讨论),实现 X.500 目录服务的方式与实现 DNS 之类的命名服务所采用的方式非常类似。在处理大型目录时,通常会分割 DIT,然后分布到几个服务器上,这些服务器在 X.500 术语中称为 DSA (directory service agents, 目录服务代理)。这样一来,DIT 的每个分割部分都对应于 DNS 的一个区域。同样,除了要实现许多典型的目录服务(如高级搜索操作等)之外,每个 DSA 的行为都非常类似于普通的名称服务器。

客户用目录用户代理(directory user agent, DUA)来表示。DUA 类似于传统命名服务中的名称解析程序。DUA 通过标准化访问协议与 DSA 交换信息。

X.500 的实现不同于 DNS 的实现的地方在于它通过 DIB 进行搜索时表现出来的灵活性。特别是,可以通过指定一组被搜索目录项属性应该满足的条件来搜索目录项。例如,假设我们需要一张由所有位于 Vrije Universiteit 的主服务器组成的表。通过使用 (Howes 1997) 定义的表示法,可以使用如下搜索操作来返回这样一张表:

```
answer=search("&(C=NL)(O=Vrije Universiteit)(OU=*)(CN>Main server)")
```

在这个例子中,虽然我们已经指定了查找主服务器的位置是名叫 Vrije Universiteit 的组织,它位于国家 NL 中,但是我们对特定的组织单位不感兴趣。不过,每条返回结果都应该有等于 Main server 的 CN 属性。

一项重要的观察结果是:目录服务中的搜索一般都是一种开销巨大的操作。例如,为找到 Vrije Universiteit 中的全部主服务器需要搜索各个部门的所有实体,并把搜索结果合并成一个答案。换句话说,为了获得答案,我们往往需要访问多个 DIT 叶节点。实际上,这也意味着要访问多个 DSA。而在实现命名服务时,一次查询请求则只需要访问一个叶节点。

和许多其他的 OSI 协议一样,根据正式的规则访问 X.500 是很重要的。为了在 Internet 中实现 X.500 服务,人们设计了一种简化的协议,这种协议称为轻量级目录访问协议(lightweight directory access protocol, LDAP)。

LDAP 是一种应用层协议,它直接在 TCP 上实现(Yeong 等 1995; Wahl 等 1997),与正式的 OSI 访问协议相比,它惟一的贡献就是它的简单性。另外,查询和更新操作的参数可以简单地作为字符串传递,而不需要 OSI 协议要求的单独编码。LDAP 逐渐成为基于 Internet 目录访问的事实标准。它正在被集成到许多分布式系统当中,包括 Windows 2000(将在第 9 章中对此进行讨论)。有关 LDAP 的应用信息,请参阅文献 (Johner 等 1998)。

4.2 移动实体的定位

到现在为止,我们讨论的命名服务主要用于位置固定的命名实体。传统的命名系统天生就不能很好地支持经常改变的名称—地址映射,比如说移动实体。本节将要讨论这些问题,同时还要给出移动实体定位的解决方案。

4.2.1 实体命名与定位

正如我们在上一节所说的那样,实体都必须命名,以便能够对它们进行查找和访问。名称共分为三种:易于理解的名称、标识符和地址。由于创建分布式系统的目的是让人来使用,同时由于实体也必须有地址,以便访问它,因而所有的命名系统实质上都要维护易于理解的名称与地址之间的映射。

我们还说过,为了高效地实现诸如 DNS 之类的大型名称空间,把名称空间划分为三个层是很有益的。全局层和行政层的特点是它们所包括的名称不经常改变。说得更准确一点,也就是这部分名称空间的节点内容相对稳定。结果是,通过复制和缓存可以完成高效的实现。

管理层的节点内容经常改变。因此,其更新和查找的性能变得很重要。实际上,通过在本地高性能名称服务器上实现节点,性能要求可以得到满足。

让我们深入探讨一下我们到底做出了哪些假设,以及为什么这种大型名称系统实现方式能够行之有效。首先,再来看一下对远程 hostftp.cs.vu.nl 所用地址的查找。如果假设全局层和行政层节点内容是稳定的,那么客户就可以在本地缓存中查找 cs.vu.nl 域的名称服务器所用的地址。因此,为了找到 ftp.cs.vu.nl 的地址,只需要向该名称服务器发送一次请求就行。

接下来,假设要更新 ftp.cs.vu.nl 的地址,因为该 FTP 服务器要被转移到另一台机器上。只要这个服务器是被转移到了 cs.vu.nl 域内的一台机器上,就可以有效地完成更新操作。在这种情况下,只需要修改 cs.vu.nl 所用名称服务器的 DNS 数据库。查询操作的效率会和以前一样。

由此看来,如果假设全局层和行政层节点不经常改变,并且更新只限于一个服务器,那么就可以高效地实现 DNS 这样的命名系统。

现在考虑一下如果把 ftp.cs.vu.nl 转移到一台名为 ftp.cs.unisa.edu.au 的机器上,那么将会发生什么事情(其中 ftp.cs.unisa.edu.au 机器位于一个完全不同的域内)。首先,名称 ftp.cs.vu.nl 最好不要改变,这是因为许多应用程序和用户可能会拥有指向它的符号链接。换句话说,这个名称可能被用作标识符。修改它可能导致所有指向它的符号链接失效。

针对现在的情况,基本上有两种方法。一种是在 cs.vu.nl 的 DNS 数据库中记录新机器的地址。另一种替代方法就是记录新机器的名称,而不是它的地址,同时有效地把 ftp.cs.vu.nl 转化成一个符号链接。但两种方法都存在严重的缺点。

让我们先考虑记录新机器的地址。很明显,在使用这种方法时查询操作不受影响。

然而,当再次把 `ftp.cs.vu.nl` 转移到另一台机器上时,必须同时更新它在 `cs.vu.nl` 所用 DNS 数据库中的项。重要的是这种更新不再是本地操作,完成这种操作实际上可能需要几百 ms。换句话说,这种方法违背了一条假设,这条假设就是对管理层节点进行的操作是高效的。

使用符号链接的主要缺点就是查询操作会变得比较低效。实际上,每次查询都被分成了两步:

- (1) 找到新机器的名称;
- (2) 查找与该名称相关联的地址。

然而,如果 `ftp.cs.vu.nl` 再次被转移,比如说转移到 `ftp.cs.berkeley.edu` 上,那么通过把 `name.ftp.cs.unisa.edu.au` 转换成指向 `ftp.cs.berkeley.edu` 的符号链接,我们就可以完成本地更新操作,并且保持 `cs.vu.nl` 所用数据库的项保持不变。这样做的缺点是必须在查询操作中添加一个步骤。

对于非常容易移动的实体来说,问题只会变得更糟糕。每当实体转移的时候,要么需要执行非本地更新操作,要么需要向查询操作添加另一个步骤。

到现在为止,我们所介绍的方法还存在另一个严重的问题,那就是名称 `ftp.cs.vu.nl` 不允许改变。因此,为实体选择合适的名称是极其重要的,选择的名称在它所代表实体的生存周期内应该不发生改变。另外,这个名称不能用于任何其他实体。在实践中,选择这样的名称,特别是为长期存在的实体选择这样的名称是困难的,WWW 中的命名已经说明了这一点。特别是,许多实体以不同的名称为人所知,并且所有这些名称应该保持有效,更精确地说,就是应该始终指向同一个实体,即使是面对易移动实体也是如此。

出于这些原因,DNS 之类的传统命名服务不能很好地处理移动实体,从而需要各种解决方法。本质上,出现问题的原因是传统的命名服务维护着易于理解的名称与实体地址之间的直接映射。每当名称或地址发生改变时,映射也同样需要修改,如图 4.17(a)所示。

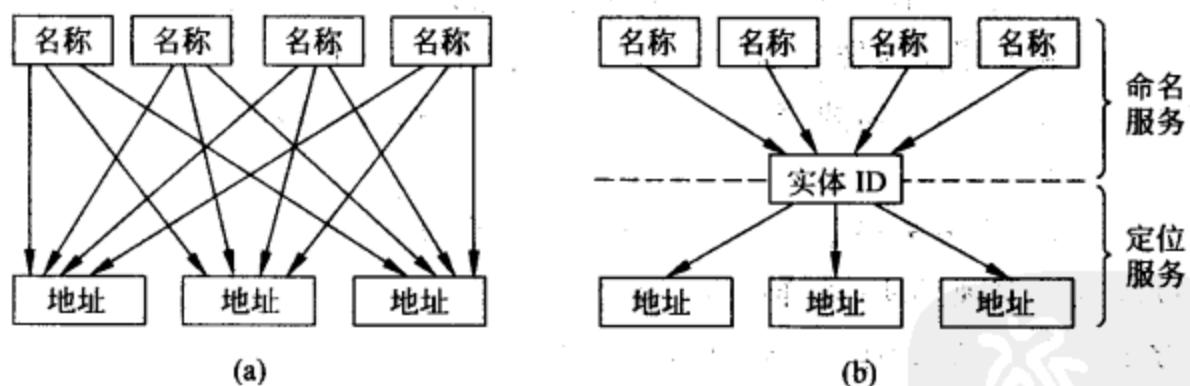


图 4.17 名称与地址之间的映射
(a) 名称与地址之间的直接、一级映射; (b) 使用标识符的两级映射

一种更好的解决方法是通过引入标识符来把命名实体与查找实体分开,如图 4.17(b)所示。回想一下从不改变的标识符,每个实体正好有一个标识符,这个标识也从不会指派给其他的实体(Wieringa 和 de Jonge 1995)。一般情况下,标识符不必拥有易于理解的表示。换句话说,它只为机器处理进行优化。

在通过命名服务查找实体时,命名服务会返回一个标识符。只要有必要就可以在本

地存储该标识符,这是因为已经知道它永远不会指向其他实体,也永远不会改变。它在哪个名称下本地存储并不重要。这样,当下一次需要该标识符时,只要从本地获取它就行了,不需要通过命名服务查找它。

实体的定位由独立的定位服务(location service)处理。定位服务实际上是以标识符为输入,然后返回被标识实体的地址。如果地址存在多份副本,那么可能会返回多个地址。在本节中,我们只集中讨论实现高效定位服务方面的问题。

4.2.2 简单方法

首先来考虑两个用于实体定位的简单方法。每种方法都只能应用于局域网。不过在这种环境中,它们通常能够很好地工作,这使得它们的简便性特别吸引人。

1. 广播和多播

设想这样一个分布式系统:它创建于一个能够提供高效广播功能的计算机网络之中。通常情况下,这种功能是由局域网提供的,在这样的网络中,所有的机器都连接在一根电缆上。同样,局域无线网络也属于这一类。

在这种环境中定位实体是一件简单的事情:包含该实体所用标识符的消息会广播到每台机器上,并且请求每台机器查看它是否拥有该实体。只有能够为该实体提供访问点的机器才会发送回复消息,回复消息中包含访问点的地址。

在 Internet ARP 协议(Address Resolution Protocol,地址解析协议)中,这个原理用于在仅仅指定 IP 地址的情况下查找机器的数据链路地址。实质上,机器会在本地网络中广播一个数据包,以查找拥有指定 IP 地址的机器。当消息到达一台机器以后,接收者会查看一下自己的 IP 地址是否与监听到的被请求的 IP 地址相同。如果是,那么它会发送一个回复数据包,其中包含它的 Ethernet 地址。

随着网络的膨胀,广播开始变得低效。不仅网络带宽被请求数据包浪费,而且更为严重的是,可能有太多的主机被它们不能回答的请求中断。一种可能的解决方法是转换成多播,通过使用多播,只有符合条件的一组主机才会接收到请求。例如,Ethernet 网络直接在硬件上支持数据链路层多播。

也可以在点到点网络中使用多播来定位实体。例如,通过允许主机加入特定的多播组,Internet 就可以支持网络层多播。这样的组由多播地址来标识。当主机向一个多播地址发送消息时,网络层会提供最尽力的服务来把消息发送给全部组成员。文献(Deering 和 Cheriton 1990; Deering 等 1996)讨论了多播在 Internet 中的高效实现。

多播地址可以用作对多播实体的通用定位服务。例如,设想这样一个组织:组织中的每个员工都拥有自己的移动电脑。当这样的电脑连接到本地可用的网络中时,网络会动态地把一个 IP 地址分配给它。另外,它还将加入一个特定的多播组。当一个进程需要查找计算机 A 时,它会向多播组发送一个“A 在哪里?”的请求。如果 A 连接到了网络上,那么它就会用它当前的 IP 地址进行响应。

使用多播地址的另一种方式就是让它与一个复制的实体相关联,并且使用多播查找最近的复制实体。向该多播地址发送请求时,每个复制实体都会用它当前的(通常的)IP

地址进行响应。选择最近的复制实体的一种粗糙方法就是选择最先回复的复制实体。文献(Guyton 和 Schwartz 1995)介绍了更多精确的方法。实践证明,选择最近的复制实体通常没那么容易。

2. 转发指针

另一种用于移动实体定位的流行方法是使用转发指针(Fowler 1985)。它的原理很简单:当实体从 A 移动到 B 时,它将在后面留下一个指针,这个指针指向它在 B 中的新位置。这种方法的主要优点是它很简便:一旦找到实体以后(比如使用传统的命名服务),客户就可以顺着转发指针形成的链查找实体的当前地址。

转发指针同时也存在许多重要的缺点。首先,如果不采取特殊措施,那么链可能会特别长,以致定位实体的开销会变得很大。其次,只要需要,链中的所有中间位置就必须维护它们的那一部分转发指针链。第三,一个相关的缺点是它所在链很脆弱,易于断开。不管是什么原因,只要有一个转发指针丢失,就无法再到达实体。因此,一个重要的事情就是让指针链相对短一些,并且确保转发指针是健壮的。

为了更好地理解转发指针的工作方式,让我们来考虑一下它们与分布式对象一起使用的情况。按照 SSP 链(Shapiro 等 1992)中的方法,每个转发指针都以(代理,骨架)值对的形式实现,如图 4.18 所示。在 SSP 中,代理被称为存根,而骨架被称为后裔,结果就变成了(stub,scion)值对,即(存根,后裔)对,这就是缩写 SSP 的来源。骨架(即服务器端存根)要么包含实际对象的本地引用,要么包含该对象所用代理(即客户端存根)的本地引用。为了强调骨架是远程引用的入口项(entry item),而代理是远程引用的出口项(exit item),我们使用了图 4.18 所示的符号。

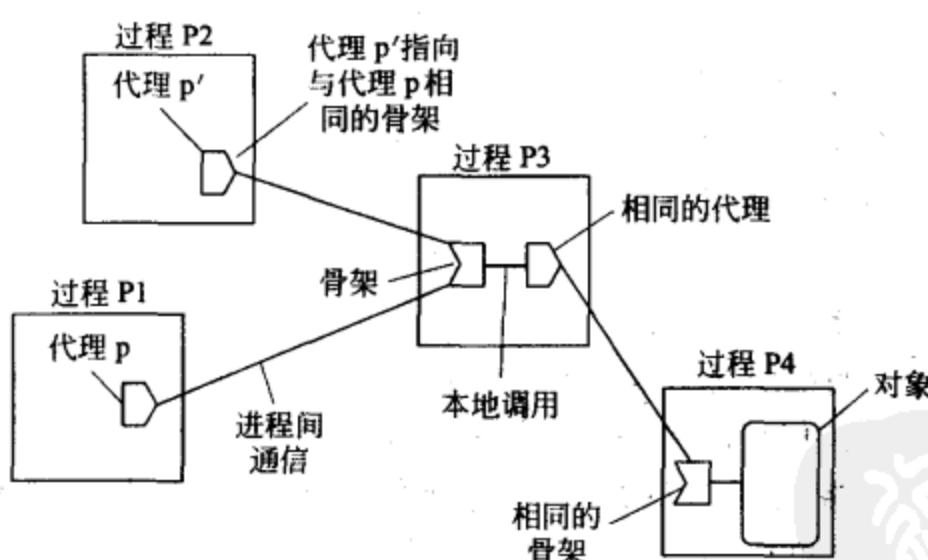


图 4.18 使用(代理,骨架)值对的转发指针

当对象从地址空间 A 移到地址空间 B 时,它会把一个代理留在 A 中,并在 B 中安装一个引用代理的骨架。这种方法存在一个有趣的特点:移动细节对客户是完全透明的。客户惟一能看到的就是一个代理。客户无法看到代理转发调用的方法,也不知道代理把调用转发到了什么位置。同时要注意,这种转发指针的使用与查找地址是不同的。相反,客户的请求顺着链被转发到实际对象上。

为了简化(代理,骨架)值对组成的链,调用会携带一个代理的标识符,这个代理是调用开始的地方。这个代理标识符由客户的传输层地址加上一个本地产生的、用于标识代理的号码组成。当调用到达位于当前位置的对象以后,会向发起调用的代理反馈一个答复。答复中包含了对象的当前位置,然后代理会把自己的对应骨架调整为对象当前位置中的那一个骨架。图 4.19 显示了这里介绍的原理。

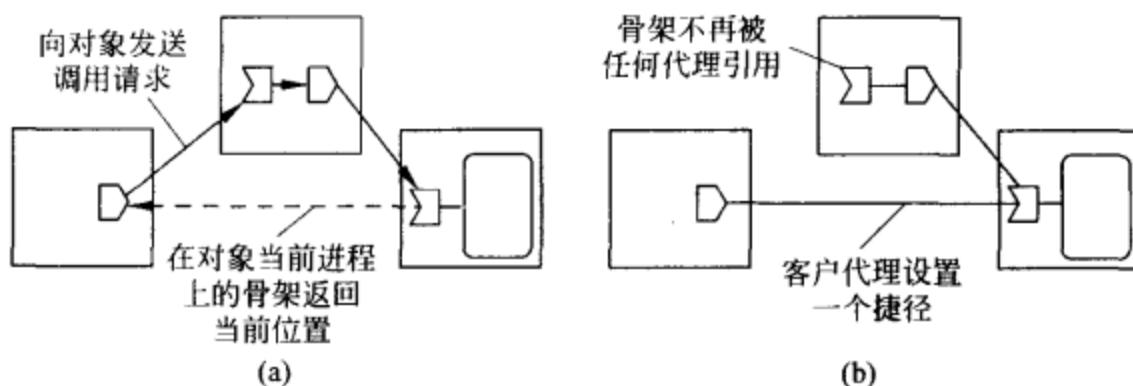


图 4.19 通过在代理中存储一个捷径来重定向转发指针

直接向起点代理发送响应和沿着转发指针的相反路线发送响应二者各有所长。对于前者来说,通信要更快一些,这是因为需要经过的过程较少。另一方面,使用前者时只有起点代理可以调整,而后者可以调整所有中间代理。

当骨架不再被任何代理引用时,它可以被删除。4.3 节将讨论自动完成这种操作的方法。

正如我们在第 2 章介绍的那样,在分布式对象系统中,可以用代理的形式实现对象的引用,代理会作为参数传递给方法调用。这种方法还可以用于转发指针。假设图 4.18 所示的进程 P1 要把它对对象 O 的引用传递给进程 P2。引用传递的实现方法是在进程 P2 的地址空间中安装代理 p' 的一份副本,即 p。因为代理 p' 和 p 引用同一个骨架,所以转发调用机制和以前一样起作用。

如果(代理,骨架)值对组成的链中有进程崩溃,或者因其他原因而无法到达,那么将会出现问题。存在几种解决方法。其中一种在 Emerald (Jul 等 1988) 和 LII system (Black 和 Artsy 1990) 里面介绍过,这种方法是让创建对象的机器(它被称为对象的起始位置)始终保留一个对对象当前位置的引用。这个引用以一种容错方式进行存储和维护。当链断开时,可以向对象的起始位置提出请求,询问它对象当前所在的位置。为了允许对对象的起始位置进行修改,可以使用传统的命名服务来记录当前的起始位置。下面讨论这种基于起始位置的定位方法。

4.2.3 基于起始位置的方法

使用广播和转发指针带来了可扩展性的问题。在大型网络中,很难有效地实现广播和多播,而过长的转发指针链会导致性能问题,并且容易受到链断开的影响。

有一种在大型网络中支持移动实体定位的流行方法,这就是引入起始位置的方法。这种方法持续跟踪实体的当前位置,可以使用特殊的技术来预防网络故障或进程失效。在实践中,通常选择创建实体的位置作为起始位置。

正如前面讨论的那样,对于基于转发指针的定位服务来说,基于起始位置的方法用来作为一种回退机制。Mobile IP(Perkins 1997)介绍了一个例子,这个例子同样采用了基于起始位置的方法。每个移动主机都使用固定的IP地址,所有与该IP地址进行的通信一开始都被转发到移动主机的起始位置代理中。起始位置代理位于局域网中,与包含在移动主机IP地址中的网络地址相对应。当一台移动主机转移到另一个网络中时,它会请求一个可以用来通信的临时地址。这种转交地址(care-of address)在起始位置代理中注册。

当起始位置代理收到发给移动主机的数据包时,它会查找主机的当前位置。如果主机是在当前本地网络中,那么就简单地转发数据包。否则,它会建立一条通往主机当前位置的通道,准确地说,它会把数据重新组装成IP包,然后发送给转交地址。同时,将把主机的当前位置告诉数据包的发送者。图4.20显示了这里介绍的原理。注意,IP地址被有效地用作移动主机的标识符。

图4.20还说明了在大型网络中基于起始位置的方法存在另一个缺点。即为了与移动实体通信,客户首先必须与起始位置进行联系,而起始位置可能与实体本身处于完全不同的位置。结果是增加了通信延迟。

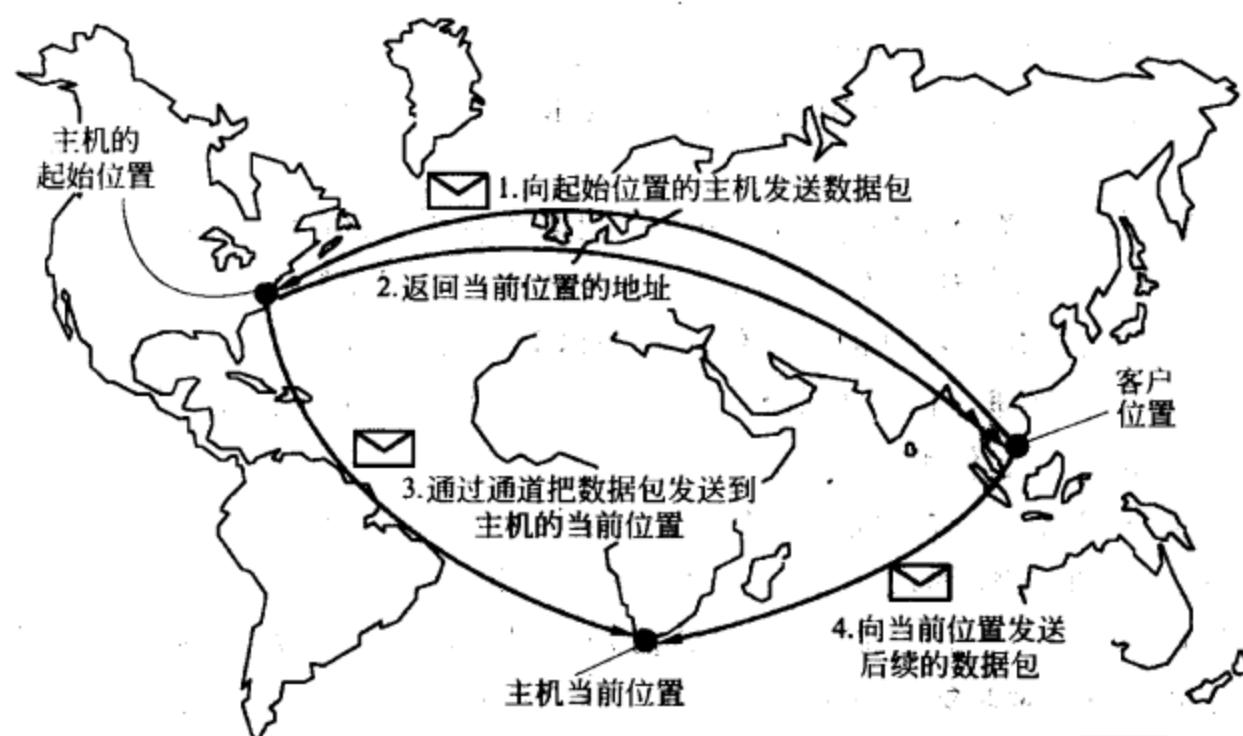


图4.20 移动IP原理

有一种应用于移动电话的解决方案,它使用了一种双层设计,(Mohan 和 Jain 1994)介绍了这种设计。在与移动实体建立连接时,客户首先查看本地注册机构,看看移动实体是否在本地。如果不是,那么就联系实体的起始位置,以便找到它的当前位置。下面,我们讨论一下对这种设计进行的扩展,扩展后的设计会跨越多个层次。

基于起始位置方法的另一个重要缺点是它使用了固定的起始位置。首先,必须保证起始位置始终存在。否则,将无法与实体联系。如果决定把一个长期存在的实体永久转移到网络的一个完全不同的部分中,而不是转移到起始位置所在的网络部分,那么问题会进一步恶化。在这种情况下,如果能让起始位置随着主机一起转移就好了。

针对这个问题的一种解决方法是：在传统的命名服务中注册起始位置，然后让客户首先查找起始位置所在的位置。由于可以假定起始位置是相对稳定的，所以在查找到它以后可以有效地缓存它。

4.2.4 分层方法

定位实体所使用的基于起始位置的两层方法可以推广到多层。在本节中，我们首先讨论一种用于分层定位设计的一般方法，然后再提出多种优化方法。我们提供的方法基于 Globe 定位服务，(van Steen 等 1998b)对这种服务进行了介绍。这是一种通用定位服务，是典型的分层定位服务，这些服务都是为个人通信系统(personal communication systems)提出的(Pitoura 和 samaras 2001; Wang 1993)。

1. 通用机制

在分层设计中，网络被划分为一组域(domain)，这与 DNS 的分层组织非常类似。有一个覆盖整个网络的顶级域。每个域都可以进一步划分成多个更小一些的子域。最低层的域称为叶域，叶域通常与计算机网络中的局域网相对应，或者对应于移动电话网络中的单元。

与 DNS 和其他分层命名系统类似，每个域 D 都拥有关联的目录节点 $\text{dir}(D)$ 。 $\text{dir}(D)$ 会持续跟踪域的实体，这样就形成了一棵目录节点树。顶级域的目录节点称为根(目录)节点，它包括了全部实体。图 4.21 显示了这种把网络划分为域和目录节点的总体组织方法。

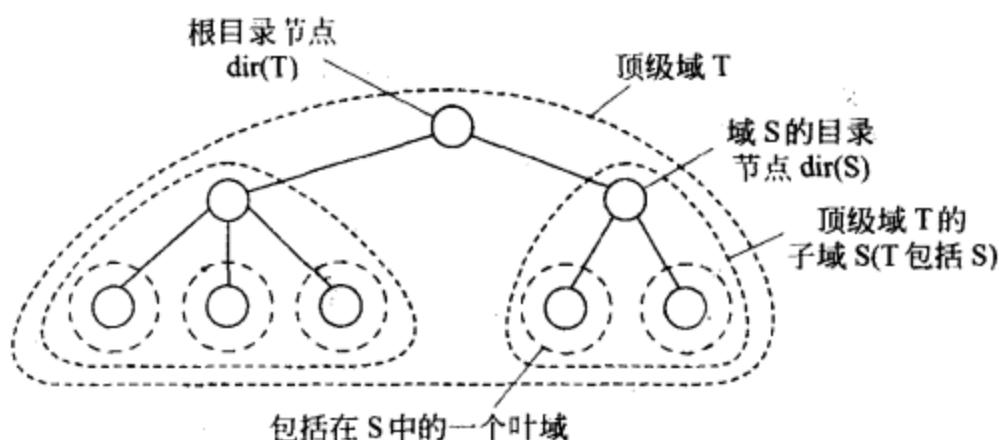


图 4.21 把定位服务划分为域的分层组织，每个域都拥有关联的目录节点

为了持续跟踪实体的位置，每个目前位于域 D 内的实体都由一条在目录节点 $\text{dir}(D)$ 中的位置记录来代表。在叶域 D 的目录节点 N 中，实体 E 的位置记录包含该实体当前在域中的位置。与之相对应，包含 D 的、更高一级的叶域 D' 使用目录节点 N'，N' 也拥有一条关于 E 的位置记录，但是这一条记录只包含一个指向 N 的指针。同样，N' 的父节点也将存储一条关于 E 的位置记录，这条位置记录也只包含一个指向 N' 的指针。因此，根节点将拥有每个实体的位置记录，其中每条位置记录都存储一个指向更低层子域目录节点的指针，而这里所说的更低层子域就是记录的关联实体当前所在的子域。

实体可以拥有多个地址，比如说它被复制了，就会出现这种情况。如果实体分别在叶域 D1 和 D2 中拥有地址，那么同时包含 D1 和 D2 的最小域的目录节点将包含两个指针，

每个指针都指向一个包含地址的子域。这样就产生了图 4.22 所示的通用组织树。

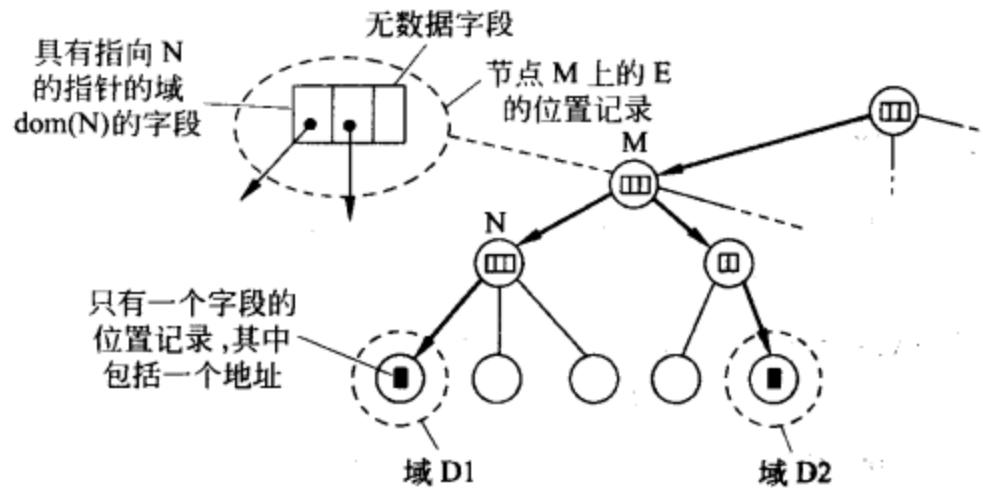


图 4.22 实体信息存储示例,这里的实体在不同的叶域中拥有两个地址

现在让我们考虑一下在这种分层定位服务中查询操作的实现方法。在图 4.23 中,希望定位实体 E 的客户向它所在的叶域 D 的目录节点发送了一个查找请求。如果这个目录节点没有存储该实体的位置记录,那么就说明该实体现在不在 D 中。因此,这个节点会把请求转发给它的父节点。注意,父节点代表一个比它的子域更大的域。如果父节点也没有 E 的位置记录,那么就会把查找请求转发给更高一层的域,以此类推。

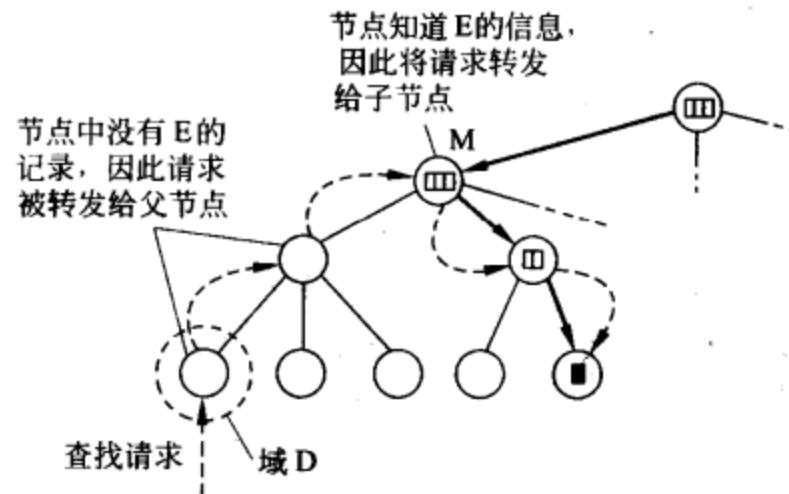


图 4.23 在分层组织的定位服务中的位置查找

如果节点 M 存储了 E 的位置记录,那么一旦请求到达 M 后,我们就可以知道 E 位于节点 M 代表的域 $\text{dom}(M)$ 中。如图 4.23 所示,M 存储了一条位置记录,其中包含一个指向其子域的指针。然后就会把请求转发给那个子域的目录节点,那个子域会依次进一步向树的下方转发请求,直到请求最终到达叶节点为止。存储在叶节点中的位置记录会包含 E 在哪一个叶域中的地址。这样就可以把这个地址返回发起请求的客户。

一个与分层定位服务有关的重要发现就是查找操作是在局部进行的。原则上,对实体的搜索是在一个以发出查找请求的客户为中心、逐步增大的环中进行的。每当查找请求被转发到更高一层的目录节点时,都会扩大搜索区域。在最差的情况下,搜索会连续进行,直至请求到达根节点为止。由于根节点拥有所有实体的位置记录,所以此时可以简单

地沿着一条向下的指针路线把请求转发给一个叶节点。

更新操作以类似的方式在局部进行,如图 4.24 所示。假设实体 E 在叶域 D 中创建了一个复制实体,需要在这个复制实体中插入 E 的地址。插入操作从 D 的叶节点 $\text{dir}(D)$ 开始,然后 D 会立即把插入请求转发给它的父节点。父节点同样会转发插入请求,直到插入请求到达已经为 E 存储了位置记录的目录节点 M 为止。

随后节点 M 会在 E 的位置记录中存储一个指针,这个指针指向转发插入请求的那个子节点。这时候,那个子节点会建立一条关于 E 的位置记录,这条位置记录中包含一个指针,这个指针指向转发请求的下一层节点。这个过程会连续进行,直至到达发起请求的叶节点为止。最后,那个叶节点会建立一条记录,这条记录包含实体在关联叶域中的位置。

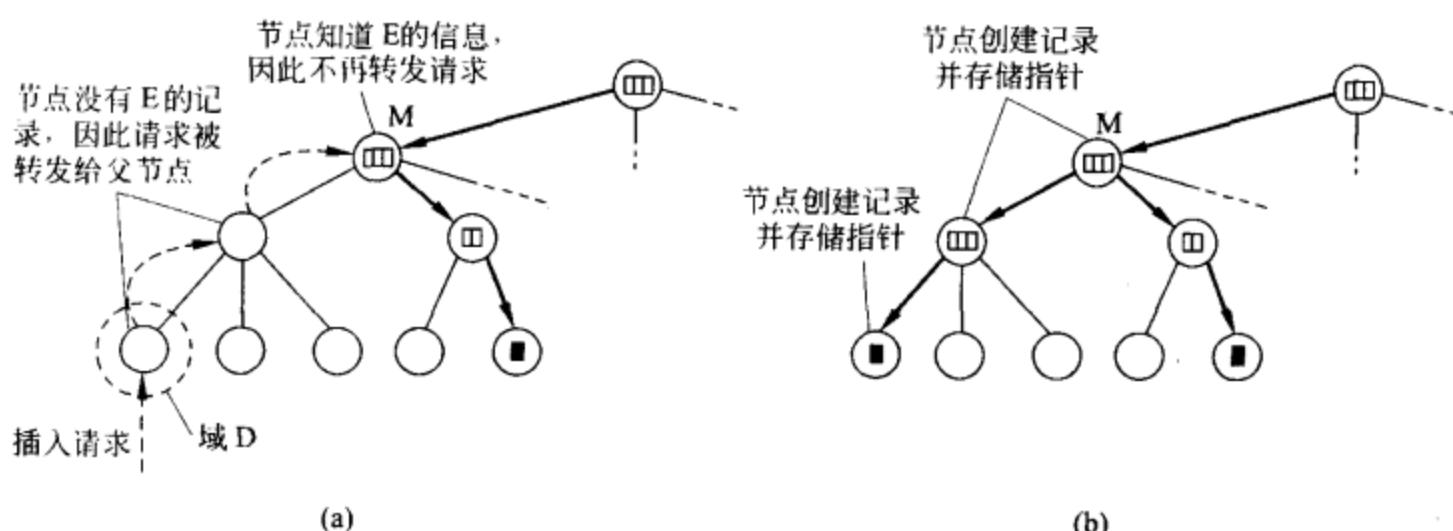


图 4.24 更新操作

(a) 插入请求被转发到第一个知道实体 E 的节点; (b) 转发指向叶节点的指针所形成的链

刚才介绍的插入地址操作会导致指针链的建立,这条指针链是一条结构严密的链,它从一个最低层的目录节点开始,该节点拥有实体 E 的位置记录。另一种选择是在向父节点发送插入请求之前创建一条位置记录。换句话说,就是以颠倒的顺序建立指针链。后者的优点是地址会尽快变得可用。这样的话,即使父节点暂时无法到达,仍然可以在当前节点所在的域内查找实体的地址。

删除操作与插入操作类似。如果需要删除叶域 D 内实体 E 的地址,那么需要将目录节点 $\text{dir}(D)$ 从它为 E 存储的位置记录中删除该地址。如果说那条位置记录变成了空的,准确地说,就是说它不再包含其他 E 在 D 中的地址,那么这条记录也可以删除。在这种情况下, $\text{dir}(D)$ 的父节点需要删除它的那一个指向 $\text{dir}(D)$ 的指针。如果父节点中用于 E 的位置记录也变空了,那么也应该删除这条位置记录并通知更高一层的节点。以此类推,这个过程将连续进行,直到从一条位置记录中删除指针后该位置记录仍然非空或者是到达根节点为止。

2. 指针缓存

分层定位服务的目的是为了支持移动实体的定位,所谓移动实体就是当前位置经常改变的实体。在传统的命名服务中,名称到地址之间的映射假定是稳定的,至少在全局层

和行政层的节点中是这样的。因此，在本地缓存中存储来自这些节点的查询结果可能是非常有效的。

对于定位服务来说，在本地缓存地址一般不会非常有效。当再次查找移动实体的地址时，移动实体很可能已经转移到另一个位置。因此，我们必须进行上述的全部查找操作。采用这种方法会使得分层定位服务不可避免地要比大多数命名服务开销更大。

只有缓存数据很少改变时缓存才会有效。假设移动实体 E 经常在域 D 内移动，这意味着 E 会经常改变它的当前地址。不过，根节点到 $\text{dir}(D)$ 之间用于实体 E 的指针路线不需要改变。换句话说，就是有一个位置保持不变，这个位置存储了与 E 最近的行踪有关的信息，在这里，这个位置就是目录节点 $\text{dir}(D)$ 。因此，缓存指向这个目录节点的引用是有效的。

一般情况下，如果 D 是一个最小的域，在这个域中有一个移动实体有规律地移动，那么在 $\text{dir}(D)$ 中而不是任何其他节点中查找 E 的当前位置是有意义的。这种方法本质上遵循了文献(Jain 1996)介绍的定位服务以及文献(van Steen 等 1998b; Baggio 等 2000)介绍的 Globe 定位服务，这种方法也称作指针缓存(pointer caching)。原则上，从发起查询的叶节点开始，一路上经过的所有节点都可以缓存指向 $\text{dir}(D)$ 的引用，如图 4.25 所示。

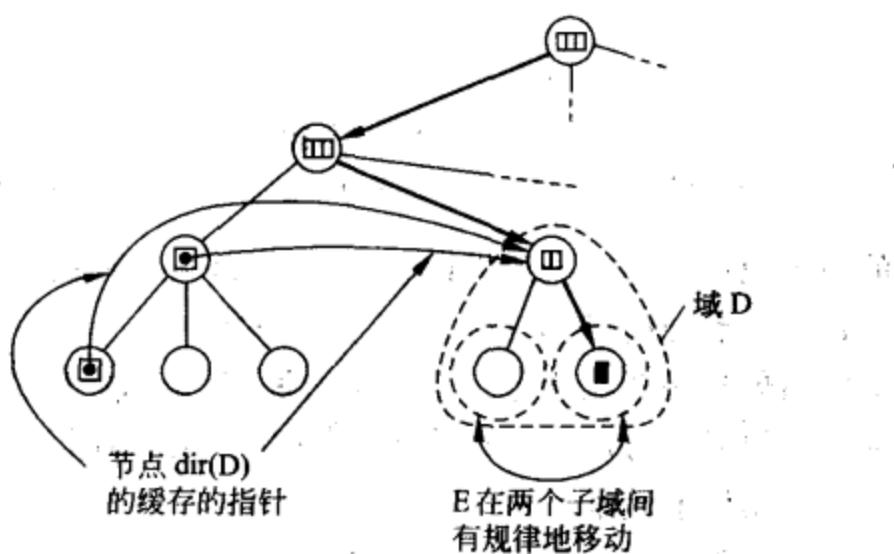


图 4.25 缓存指向最低层域所用目录节点的引用，大多数时间，实体都存在于最低层域中

通过不让 $\text{dir}(D)$ 存储指向 E 当前所在子域的指针，而是让它直接存储 E 的实际地址，情况可以进一步改善。再加上指针缓存，查找操作只用两步就可以实现。第一步要求检查本地指针缓存，这样就会直接通往正确的目录节点。第二步要求这个节点返回 E 的当前地址。

尽管在分层定位服务中指针缓存原理可以起到作用，但是还有许多未决问题需要进一步关注。问题之一就是如何找到最合适的目录节点来存储移动实体的当前地址。可以想象这么一个用户：他经常带着移动电脑在两个城市之内(或之间)来回走动，比如说是旧金山和洛杉矶。

当这个用户在旧金山的时候，他可能会经常在这个城市内转移位置。因此，在代表旧

金山域的目录节点内存储他的当前位置比较合适。当我们这位忙碌的用户到洛杉矶时也会发生类似的情况。

不过,与此同时,这个用户一直都在旧金山和洛杉矶之间飞来飞去。在这种情况下,不管他是在旧金山还是在洛杉矶,把他的当前位置存储在一个更高一层的目录节点,比如代表加利福尼亚州的目录节点中会更有效。

另一个未决问题是什么时候让缓存项失效。在我们的例子中,假设那位用户收到很多来自纽约的申请,因而他决定在纽约曼哈顿开一个办事处,并让他的一位朋友处理所有来自纽约地区的申请。对于定位服务来说,所发生的事情就是要在曼哈顿域的叶节点中有一个永久地址可用。对于任何来自纽约的查询请求都应该返回那一个新地址,而不能随着缓存指针指到加利福尼亚的目录节点去,如图 4.26 所示。

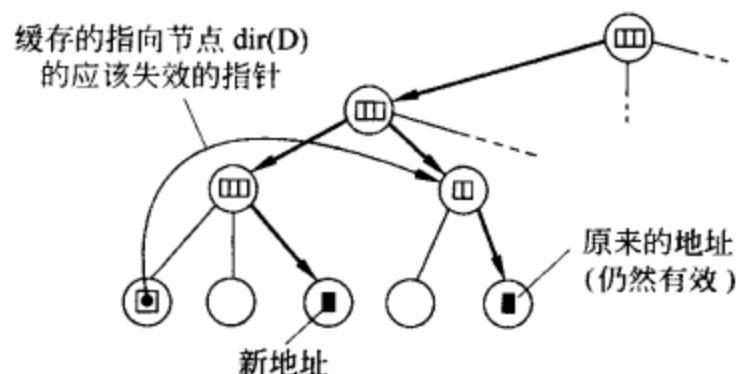


图 4.26 缓存项需要失效,因为它返回了一个非本地地址,尽管这个地址仍然有效

3. 可扩展性问题

分层定位服务存在的一个主要问题是根节点需要存储所有实体的位置记录并为每个实体处理请求。存储本身不是主要问题,每条位置记录可能相对较小,这是因为它只包含实体的标识符,再加上一个或多个指向较低层目录节点的指针。如果每条位置记录的大小大约是 1KB,那么需要的存储空间(假设有 100 万个实体)也只有一个 TB。10 张 100GB 的磁盘就可以提供这么大容量的存储空间。

真正的问题是:如果不采用特殊技术,根节点需要处理太多的查找和更新请求,以致它会成为瓶颈。这个问题的解决方法是把根节点以及其他高层目录节点划分成多个子节点。每个子节点负责处理与定位服务支持的所有实体的特定子集相关的请求。

不幸的是,简单地划分高层节点还不够。为了理解目前的可扩展性问题,假设把根节点划分成 100 个子节点。问题是在定位服务覆盖的网络中,在物理上如何放置每个子节点。

用于放置子节点的一种可能是集中放置。按照这种方法,各个子节点相互保持很近的距离,比如说是在一个簇(cluster)中。这样,就可以用一台并行计算机,如 COW 或 MPP(第 1 章简单地介绍过)等来有效地实现根节点。不过,尽管现在的处理能力可能已经足够,但是与根节点之间的网络连接可能没有足够的能力来处理全部请求。

因此,一种更好的选择是把子节点均匀地扩散到网络中。不过,如果处理不当,这种方法就有可能在可扩展性方面出现问题。再次假设那个移动用户主要在旧金山和洛杉矶

之间移动。根节点已经被划分成了多个子节点,现在需要解决的问题是哪个子节点应该负责这个用户。

假设有一个子节点放在了芬兰,并且选择它来始终存储这个用户的位置记录。在不考虑指针缓存的情况下,这意味着查询请求(比如说来自巴西)在被转发到加利福尼亚目录节点之前将通过位于芬兰的根节点。不过,正如图 4.27 显示的那样,如果这样一个请求经过一个位于加利福尼亚的子节点,那么将会更为有效。在非常大型的定位服务中,让哪一个子节点来处理哪个实体仍然是一个悬而未决的问题。

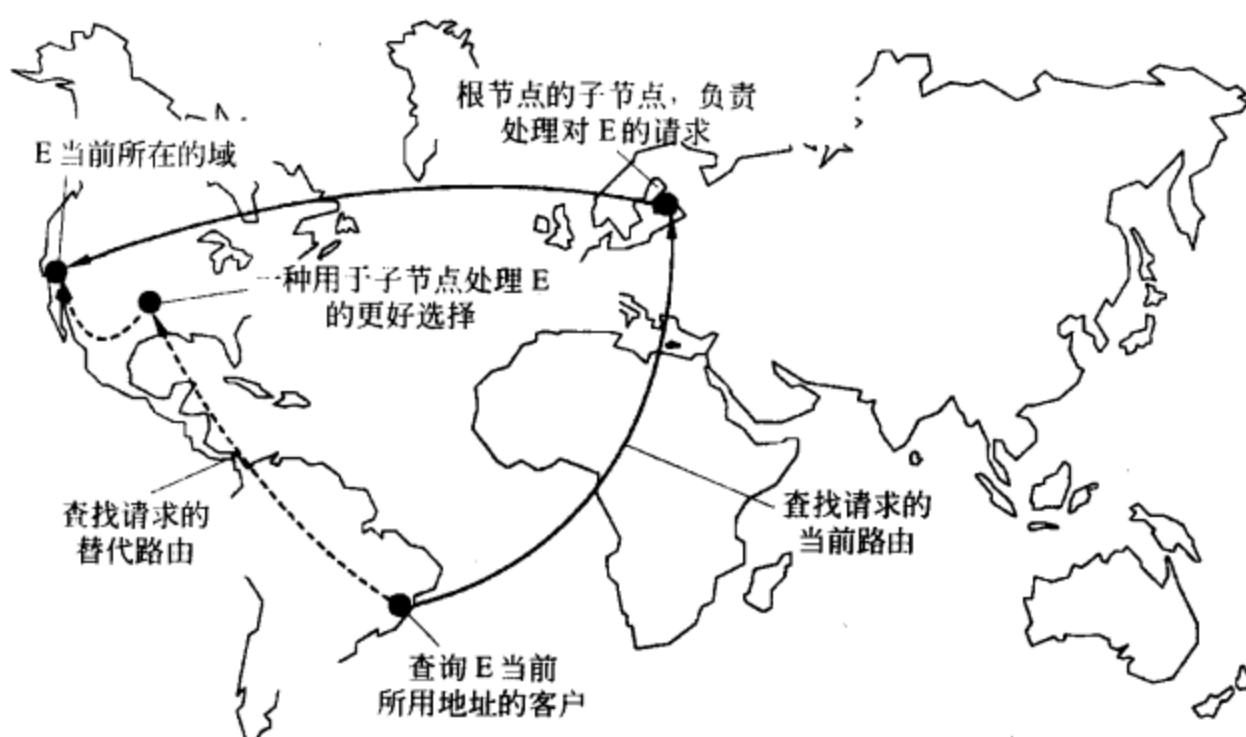


图 4.27 在定位服务覆盖的网络中均匀放置根节点的子节点所存在的可扩展性问题

一种可能的解决方法是考虑使用创建实体 E 的位置。特别是,如果根节点的一个子节点与创建 E 的位置接近,那么就让它负责处理根一级的、与 E 相关的请求。这个解决方法适合这样一些实体:这些实体往往停留在离创建它的位置很近的地方。不过,如果实体转移到一个很远的位置,那么问题依然存在。有关这种方法的细节,请参考(Ballintijn 等 1999a)。

4.3 删除无引用的实体

命名和定位服务提供一种对实体的全局引用服务。只要实体被这样一种服务引用,那么它就可以被访问和使用。一旦实体不再被访问,就应该把它删除。

在许多系统中,对实体的删除是显式进行的。举例来说,如果进程 P 知道自己是最后一个使用某个文件的进程,并且没有任何其他进程将来还需要使用该文件,那么 P 在完成它的工作后可以放心地删除这个文件。不幸的是,在分布式系统中管理实体的删除通常是困难的。特别是,通常不知道对一个实体的引用是否还存在于某个地方,也不知道以后是否还会通过这个引用来访问该实体。如果不搞清这些问题,那么删除该实体将会

在以后对它的访问中导致问题。

另一方面,仅仅因为不能确定实体的引用是否存在就从不删除实体也是不能接受的。如果没有任何引用存在,那么就会出现这样的情况:有一个实体将来再也不会使用了,但是它很可能还在消耗资源。很明显,这样的实体是一种垃圾,应该把它们删除掉。

为了解决与删除无引用实体相关的问题,分布式系统可以提供一种功能,这种功能就是不再需要实体时就自动删除实体。所有这些功能也称为分布式垃圾收集器(distributed garbage collector)。在本节中,我们将深入探讨命名实体、引用实体以及自动收集那些不再需要的实体三者之间的关系。

4.3.1 无引用对象的问题

为了说明垃圾收集的工作方式,我们将集中介绍分布式对象,特别是远程对象的垃圾收集。请回忆一下远程对象的实现方式,这是通过将它的全部状态放在一个对象服务器中来实现,而客户只拥有一个代理。正如我们在2.3节中介绍的那样,对远程对象的引用一般情况下可以用(代理,骨架)值对来实现。客户端代理包含了联系对象所需要的全部信息以便通过服务器实现的关联骨架与对象进行联系。在我们的例子中,骨架将和代理一起参与垃圾收集所需要的管理工作。换句话说,客户端和真正的对象看不到完成垃圾收集所需要的工作。注意,对象本身可以包含对另一个远程对象的引用,这种引用可依靠指针来实现,而该指针指向远程引用中的代理。同样,通过把与远程引用关联的代理复制到另一个进程,可以把远程引用传递给另一个进程。

在下面的介绍中,我们假定只要存在对象的远程引用,就可以访问对象。不存在远程引用的对象应该从系统中删除。另一方面,有对象的远程引用并不意味着将会访问该对象。由于各种原因,可能会存在这样两个对象,其中每一个都存储着对另一个的引用,除此之外,根本不存在其他的引用。这种情况很容易导致对象循环引用,这时,对象只是相互引用。应该检测到这种对象并删除它们。

一般情况下,可以用一张图来表示这种情况。在这张图中,每个节点都代表一个对象。从节点M到节点N的一条边代表对象M包含对对象N的引用。存在一个与众不同的对象子集,可以称之为根集(root set),其中的对象不需要自我引用。根集中的对象通常代表系统内的服务、用户等。

图4.28显示了一个例子,这个例子说明了这种引用图。所有白色节点都代表没有被根集对象直接或间接引用的对象,应该删除这类的对象。

与分布式系统相比,在单处理器系统中检测和删除不被引用的对象要相对简单一些(有关单处理器系统中的垃圾收集综述,请参考文献(Wilson 1994))。由于对象分散在多台机器上,所以分布式垃圾收集需要进行网络通信。结果是,通信对方法的效率和可扩展性起着决定性作用。另外,通信、机器和进程都受故障的影响,这会使问题更加复杂化。

在本节中,我们要讨论一些著名的分布式垃圾收集方法。大多数情况下,这些方法只能解决部分问题。我们遵循的是文献(Plainfosse 和 Shapiro 1995)介绍的一种方法,这种方法在分布式垃圾收集方面提供了更进一步的概括。要了解更多这方面的信息,请参考文献(Abdullahi 和 Ringwood 1998)。

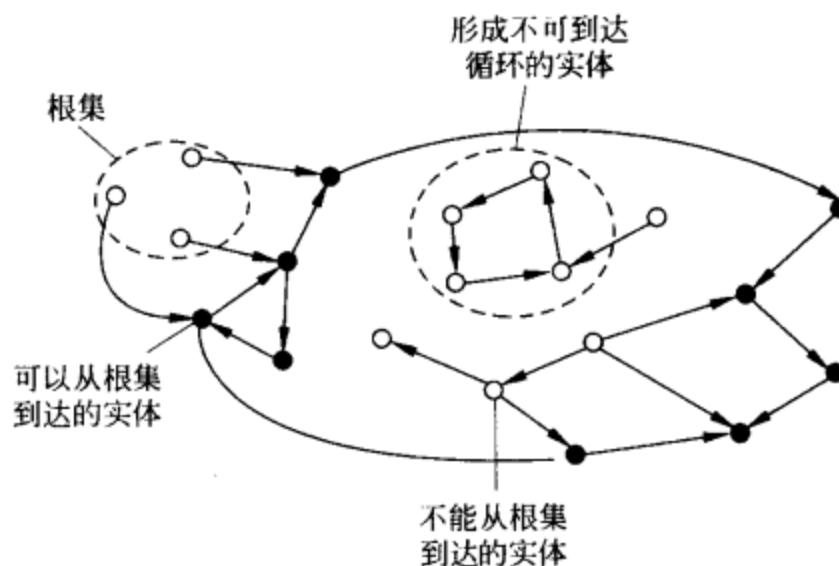


图 4.28 对象图例,这里的对象相互包含对对方的引用

4.3.2 引用计数

在单处理器系统中,有一种用于检测对象是否可以被删除的流行方法,这种方法就是简单地对对象引用进行计数。每当创建对象引用时,用于该对象的引用计数就会递增。同样,在删除引用时,引用计数就要递减。一旦计数达到 0,就可以删除该对象。

1. 简单的引用计数

在分布式系统中,简单的引用计数会产生许多问题,这些问题在一定程度上是由于通信不可靠而引起的。为了保持普遍性,我们假定对象把它的引用计数存储在关联的骨架中,骨架由负责该对象的对象服务器维护。如图 4.29 所示。

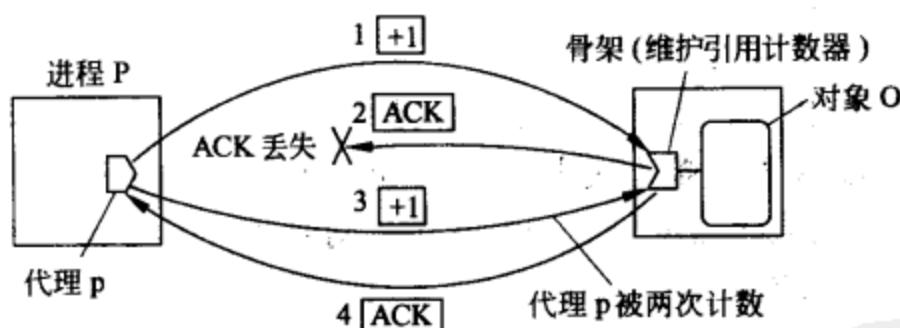


图 4.29 在通信不可靠的情况下维护正确的引用计数所存在的问题

当进程 P 创建远程对象 O 的引用时,它会在自己的地址空间中安装 O 的代理,如图 4.29 所示。为了递增引用计数,代理会向对象的骨架发送一条消息 m,并希望它返回确认信息。不过,如果确认消息丢失,那么代理将重新发送消息 m。如果不采取特殊的措施来检测重复消息,那么骨架可能会错误地再次递增它的引用计数。在实践中,检测重复消息相对比较容易。

同样,在删除远程引用时也可能产生问题。这时候,代理会发送一条要求递减引用计数的消息。如果确认消息再次被丢失,那么再次发送这条消息可能导致错误地再次递减引用计数。因此,在分布式引用计数中,必须检测重复消息然后在它们到达时丢

弃它们。

另一个需要解决的问题发生在向其他进程复制远程引用时。如果进程 P1 向进程 P2 传递了一个引用，那么对象，准确地说应该是骨架。将不会知道建立了新的引用。因此，如果进程 P1 决定删除它自己的引用，那么引用计数可能会降到 0，而且 O 在 P2 与它联系之前可能已经被删除。图 4.30(a)说明了这个问题。

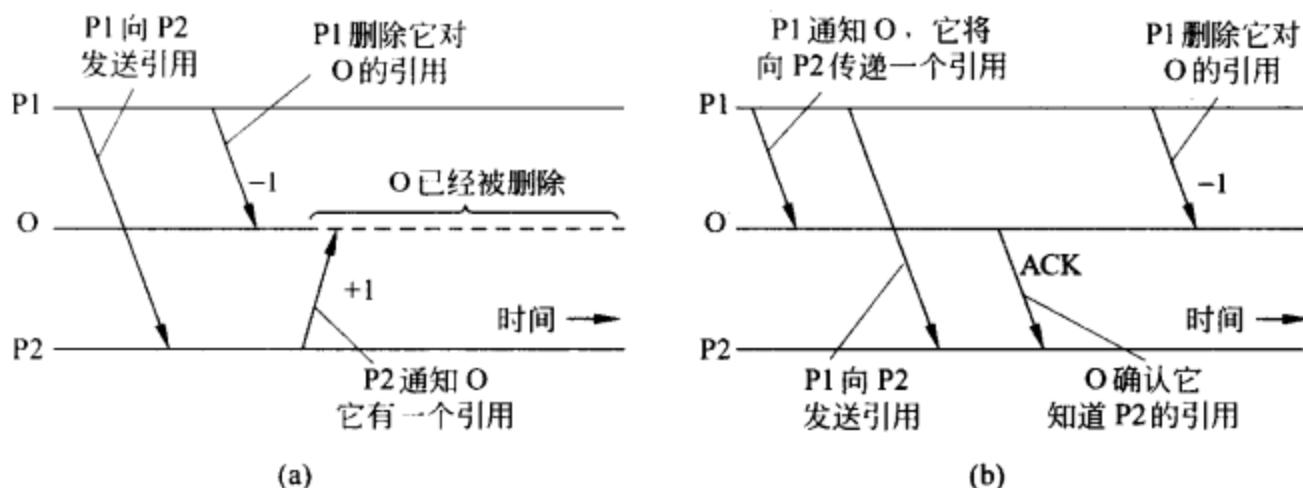


图 4.30 引用计数复制和递增及其解决方法

(a) 向其他进程复制引用计数，过一段时间以后再递增引用计数；(b) 解决方法

一种解决方法是让 P1 通知对象的骨架，告诉骨架自己将向进程 P2 传递一个引用。此外，在骨架确认它已经知道引用的存在之前禁止进程删除引用。图 4.30(b)说明了这种解决方法。对象 O 发送给 P2 的确认消息向 P2 证实 O 已经注册了该引用，这条确认消息将允许 P2 删除它对 O 的引用。只要 P2 不能确定 O 已经知道了它对 O 的引用，就不允许 P2 请求 O 递减引用计数。

注意，除了可靠的通信之外，传递引用现在需要三条消息。显而易见，在大型分布式系统中这很容易产生性能方面的问题。

2. 高级引用计数

正如刚才介绍的那样，简单的分布式引用在递增和递减引用计数之间造成了一种竞争状态。如果只采用递减操作，那么就可以避免这种竞争状态。加权引用计数 (weighted reference counting) 采用了这种解决方法，在加权引用计数中，每个对象都拥有一个固定的总权数。在创建对象时，总权数连同部分权数一起存储在与它的关联的骨架（我们称它为 s）中，这时部分权数被初始化成总权数，如图 4.31(a)所示。

在创建新的远程引用 (p, s) 时，有一半存储在对象骨架中的部分权数被赋给了新的代理 p，如图 4.31(b) 所示。剩下的一半留在骨架 s 中。在复制远程引用时（比如说把它从 P1 传递到 P2），在 P1 所用的部分权数中有一半被复制给 P2，而另一半留在 P1 所用的代理中，如图 4.31(c) 所示。

在删除引用时，一条递减消息被发送给对象的骨架，然后对象的骨架就会从总权数中减去被删除引用的部分权数。一旦总权数为 0，就表明不再存在任何远程引用，因而可以安全地删除对象。注意，在这种情况下，我们同样假定消息不会丢失或多次发送。

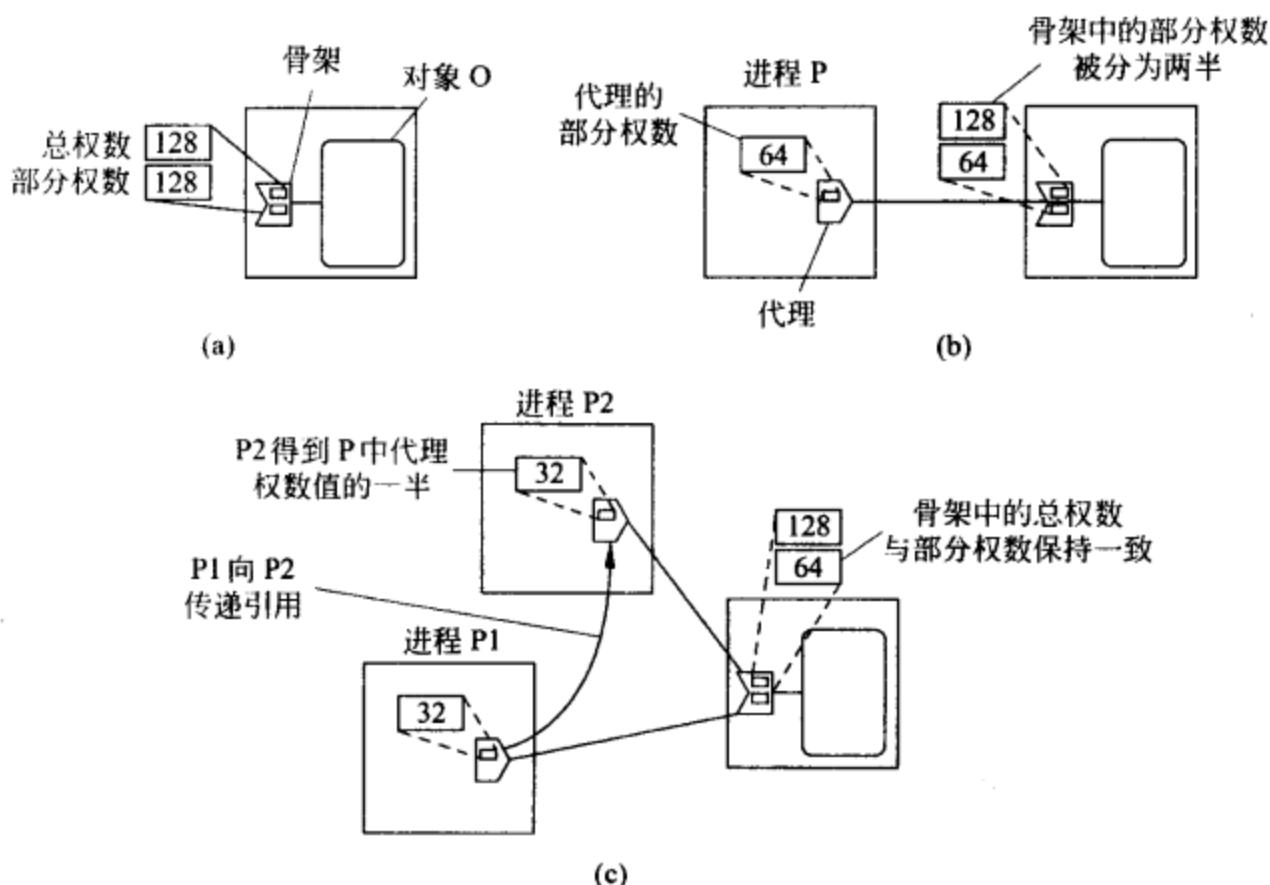


图 4.31 加权引用计数中的权数值

(a) 加权引用计数中权数的初始值; (b) 创建新引用时的权数值; (c) 复制引用时的权数值

加权引用计数存在的主要问题是只能创建有限的引用数量。一旦对象骨架的部分权数以及远程引用的部分权数降到 0, 就不能再创建或复制更多的引用。解决这个问题的方法是使用间接权数。假设进程 P_1 需要向进程 P_2 传递引用, 但是进程 P_1 自己的代理所拥有的部分权数已经达到 1, 如图 4.32 所示。在这种情况下, P_1 可以使用一个适当的总权数在自己的地址空间里创建一个骨架 s' , 同时让部分权数等于总权数。这和在对象的地址空间里创建骨架 s 完全类似。然后发送给 P_2 一个代理, 这个代理带着骨架 s' 的一半部分权数。另一半权数留在 s' 中, 以便将来分散到其他代理中。

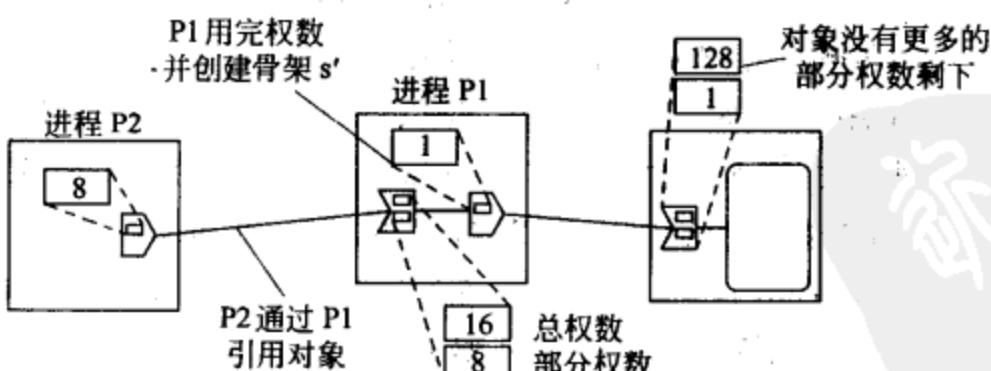


图 4.32 在引用的部分权数达到 1 时创建一个间接权数

注意, 如果骨架 s' 的总权数设成了 1, 那么这种方法与创建从进程 P_2 到进程 P_1 的转发指针相同。以此类推, 如果需要传递它的引用, 那么它必须创建另一个转发指针。使用转发指针存在的最严重问题是长指针链严重地降低了性能。同时, 指针链很容易受到故

障的影响。

可以使用世代引用计数(generation reference counting)替代间接权数。同样,我们假定每个远程引用都由(proxy, skeleton)(代理,骨架)值对构成,(惟一的)骨架和对象位于相同的地址空间中。除了一个世代是之外,每个代理还存储一个计数器,这个计数器用来记录代理被复制的次数。在创建新引用(p, s)时,相应代理 p 的世代是被设成 0。由于还没有从 p 复制任何拷贝,所以它的复制计数器也被设成 0。

向其他进程复制远程引用(p, s)时还是使用通常的方式,即发送 p 的一份拷贝 p' 。在这种情况下, p 的复制计数器会递增,从而将 p' 设置为 0。不过,由于 p' 是从 p 复制过来的,所以说它属于下一世代,出于这个原因, p' 的世代是要比 p 高,如图 4.33 所示。

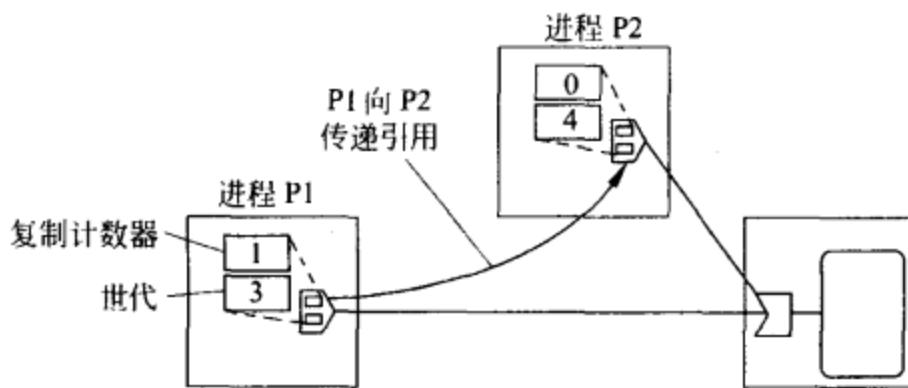


图 4.33 在世代引用计数中创建和复制引用

骨架维护着一张表 G ,在这张表中, $G[i]$ 表示世代 i 未完成的拷贝数量。如果代理 p 被删除,那么将向一个骨架发送一条删除信息,这个骨架包含该代理的世代数(比如说是 k),以及从 p 复制的拷贝数量(比如说是 n)。该骨架将让 $G[i]$ 减去 1,表示有一个属于第 k 个世代的引用被删除。然后,它会把 $G[k+1]$ 加上 n ,以表明被删除的引用已经生成了 n 个同类,换句话说,也就是它已经被复制了 n 个下一世代的引用。(注意,该骨架可能首先需要创建 $G[k+1]$,这是因为它还不知道有世代 $k+1$ 。)只要 $G[i]$ 变成 0,就表示该对象已经不再被引用,因而可以把它删除。

世代引用计数同样需要可靠的通信,不过在进行复制时,可以在不与骨架联系的情况下处理引用的复制。

4.3.3 引用列表

另一种管理引用的方法是让骨架持续跟踪引用它的代理。换句话说,骨架不是对引用进行计数,而是维护一张明确的列表,列表中列出了所有指向它的代理。这样一张引用列表拥有以下重要特征:如果一个代理已经被列出,那么把这个代理添加到引用表中不会产生任何影响。同样,删除列表中不存在的代理也不会产生任何影响。因此,添加或删除代理都是幂等(idem potent)操作。

幂等操作的特征是它们可以在不影响最终结果的情况下重复进行。特别是,在创建新的对象引用时,创建过程可以反复向对象骨架发送消息,请求骨架把代理添加到引用列表中。一旦发送被确认,创建过程就不再发送这样的消息。与此类似,引用的删除也可通过(可能是重复进行)向骨架发送消息,请求它从列表中删除代理来完成。递增和递减操

作很明显不是幂等操作。

这样,引用列表的管理就不再需要可靠的通信,也不需要检测和丢弃重复消息了(不过,需要对插入或删除操作进行确认)。这是引用列表胜过引用计数的一个重要优点。

Java RMI 采用了引用列表方法,Java RMI 基于(Birrell 等 1993)中描述的一种方法。在这种方法中,当进程 P 创建对象的远程引用时,它会向对象骨架发送它的标识,然后对象骨架会把 P 添加到引用列表中。当确认消息返回以后,进程会在自己的地址空间里创建该对象的一个代理。

向另一个进程复制引用(更准确地说,就是发送一份代理拷贝)可使用类似的方法进行处理。当进程 P1 向进程 P2 发送一份对象 O 的代理拷贝时,P2 会首先请求对象骨架把 P2 添加到自己的引用列表中。当这些完成以后,进程 P2 会在自己的地址空间里安装该代理。

如果在 P2 请求把自己插入对象引用列表之前,进程 P1 删除了它自己的代理,那么就可能会发生问题。在这种情况下,如果 P1 发送给对象骨架的删除请求在来自 P2 的插入请求之前完成,那么引用列表就可能会变成空的,结果是骨架错误地认为可以删除对象了。这种竞争状态与引用计数下的竞争状态(如图 4.30(a)所示)完全类似,并且可以用相同的方法加以解决。

引用列表的另一个重要优点是:在进程发生故障时,它更容易保持引用表的一致性。对象骨架会经常查看每个列出的进程,看它们是否依然存在和运行,实现方法是向进程发送 ping 消息,询问它是否还处于激活状态并且拥有该对象的引用。进程应该迅速响应这条消息。如果没有收到响应,那么可能再经过几次尝试之后,骨架将把该进程从表中删除。

引用列表的主要缺点是:如果骨架需要持续跟踪许多引用,那么引用列表的规模可能会严重地扩张。让列表保持简短的方法就是让骨架承诺它注册的引用只在有限的时间内有效。如果在超时之前代理没有更新注册,那么该引用就会被简单地从列表中删除。这种方法也称作分发租用(lease)。在第 6 章我们将再次介绍租用。

4.3.4 标识不可到达实体

正如图 4.28 所示,分布式系统中的实体集合可能包括这样的实体:这些实体相互引用,但是所有这些实体都无法从根集中的一个实体到达,这样的实体也应该被删除。不幸的是,前面介绍的垃圾收集技术无法查找这些实体。

需要一种方法来跟踪分布式系统中的所有实体。一般情况下,这可以通过查看哪些实体能够从根集实体到达并删除所有其他实体来完成这项工作。这样的方法一般叫做基于跟踪的垃圾收集(tracing-based garbage collection)。与到现在为止所讨论过的分布式系统中的各种垃圾收集相比,基于跟踪的垃圾收集存在固有的可扩展性问题,这是因为它需要跟踪分布式系统中的全部实体。

1. 分布式系统中的简单跟踪

为了理解分布式基于跟踪的垃圾收集,考察一下在单处理器中跟踪的实现方法是很有用的。最简单的单处理器跟踪方式标记—清除收集器采用了这种收集器分为两个

步骤。

在标记阶段,使用源于根集实体的引用链对实体进行跟踪。可以通过这种方式到达的实体都被作上了标记,比如说是把实体记录在一个单独的表中。在清除阶段会对内存进行全面的检查,以便查找没有作上标记的实体。这种实体被认为是应该清除的垃圾。

可以用另一种方式来了解标记—清除收集器,这种方式就是用三种颜色来标记实体。最初,需要检查的每个实体都是白色的。到标记阶段结束时,所有标记成黑色的实体都是可以从根到达的实体,而那些不可到达的实体仍然是白色的。如果实体是可到达的,但是该实体存储的引用还需要进行检查,那么该实体就被标记成灰色的。当实体的所有引用都被标记成灰色以后,这个实体就会变成黑色。

文献(Jul 等 1998)介绍的 Emerald 系统实现了分布式版本的标记—清除收集器。在 Emerald 系统中,每个进程都会启动一个本地垃圾收集器,所有收集器都是并行运行的。收集器会给代理、骨架和真正的对象加上颜色。开始的时候,它们都被标记成白色。当进程 P 的地址空间中的某个对象可以从同样在 P 中的根到达时,这个对象就会被标记成灰色。在把对象标记成灰色的时候,该对象包含的所有代理也都被标记成灰色。把代理标记成灰色意味着本地垃圾收集器记录下了如下内容:被引用的远程对象仍然需要与它关联的本地垃圾收集器对它进行检查。

在把代理标记成灰色时,会向与代理关联的骨架发送一条消息,让骨架把自己也标记成灰色。一旦骨架变成了灰色,与骨架关联的对象也就标记成了灰色。以此类推,该对象的所有代理都会标记成灰色。这时候,骨架以及与它关联的对象都会被标记成黑色,并且向与它关联的代理发回一条消息。注意,尽管在这种方法中,骨架知道是哪些代理与它连接,但是这并不说明代理是可以从骨架到到达的。在逻辑上,(代理,骨架)值对是严格的、从代理到骨架的单向引用。

当代理收到一条消息,被告知与它关联的骨架现在是黑色的以后,代理也会标记成黑色的。换句话说,本地垃圾收集器现在知道了那个通过代理引用的远程对象已经被记录为可到达的。

当所有本地垃圾收集器都完成了它们的标记阶段以后,它们就可以独立地收集所有被当成垃圾的白色对象了。当所有对象、骨架和代理都被标记成白色或黑色以后,标记阶段就结束了。删除白色的对象也意味着要删除与它关联的骨架以及该对象包含的所有代理。

标记—清除算法的主要缺点是它需要让可到达性图在两个阶段保持一致。换句话说,需要将原来创建的进程暂时停下来,并把执行切换到垃圾收集上。在分布式标记—清除实现中,这意味着首先需要同步系统中的所有进程,然后让每个进程切换到垃圾收集上,完成以后它们都可以继续它们原来的工作。

这种情形也被称为“停止一切”同步。对于分布式垃圾收集器来说,这通常是无法接受的。可以通过增加垃圾收集器来改善这种状况,增加垃圾收集器可以让程序执行与垃圾收集交叉进行。不幸的是,在分布式系统中,这种收集器不能很好地扩展。由于它们是和修改可到达图的程序一起运行,通常需要把对象标记成灰色的,这就使得灰色标记需要被传播到远程进程。结果会出现很高的信息通信量,这可能降低系统的

整体性能。

2. 组内跟踪

为了解决许多基于跟踪的垃圾收集器所固有的可扩展性问题,Lang 等人设计了一种方法,通过使用这种方法,大型分布式系统中的进程(包含对象的进程)按层次组织到组(Lang 等 1992)中。通过标记—清除和引用计数的联合使用,垃圾收集在组内进行。现在我们集中介绍一下在进程组内进行垃圾收集的基本算法。

组就是进程集合。使用组的惟一原因就是为了提高可扩展性。它的基本思想是先在组内收集所有垃圾,其中包括任何完全存在于组内的引用循环。下一步就考虑更大一些的组,这些更大的组包含许多子组,但那些刚刚使用垃圾收集器进行过清除操作的组除外。

为了适应组内跟踪,我们要假定远程引用还是用(代理,骨架)值对实现。对于每个对象来说,在对象存在的地址空间内都是只有一个骨架,不过可以有多个代理与骨架通信。骨架维护着 4.3.2 节中介绍的引用计数器,引用计数器对关联的代理进行计数。对于每个分布式对象来说,一个进程最多只能拥有该对象的一个代理。

进程组形成以后,需要使用如下 5 个步骤来构成组内收集垃圾的基本算法,下面将对它们进行详细讨论:

- (1) 初始化标记,这个步骤只标记骨架;
- (2) 在进程内从骨架向代理传播标记;
- (3) 在进程间从代理向骨架传播标记;
- (4) 通过重复前两个步骤来获得稳定;
- (5) 收集垃圾。

在开始详细解释每个步骤之前,理解标记实体的含义是很重要的。这个算法本质上只标记代理和骨架。还有重要的一点就是要注意代理和骨架都不会属于根集。

骨架既可以标记成“软性的”,也可以标记成“硬性的”,而代理可以标记成“无”、“软性的”或“硬性的”。如果骨架标记成了硬性的,那么就意味着骨架既可以从组外进程内的代理到达,也可以从组内的根对象到达,所谓组内的根对象,就是该组所属进程的根集包含的对象。如果骨架被标记成软性的,那么表明只能从组内的代理到达。修改骨架的标记时,只可以将软性的改成硬性的。

如果代理被标记成硬性的,那么它可以从根集内的对象到达。如果代理被标记成软性的,那么它对于同样被标记成软性的骨架是可到达的。这种代理有可能存在于一个循环中,无法从根集中的对象到达这样的循环。只有被标记成无的代理才可以改变成硬性的。正如我们将看到的那样,一旦代理被标记成了软性的,就不能再修改它的标记了。

第 1 步只标记骨架。骨架可以标记成软性的或硬性的,这要根据是否可以从组外代理到达来决定。这种可到达性很容易检测,只要检查骨架的引用计数器就行了。这个计数器的值表明有多少其他进程的代理在引用这个骨架。这些进程中有一些是组内的,还有一些是组外的。如果其中有一个组外进程的代理,那么骨架就应该被标记成硬性的。

通过简单地计算有多少与骨架关联的代理存在于组内，然后从引用计数器中减去这个数字，就可以确定是否还有组外的关联代理。这样就产生以下算法：

- (1) 对于每一个组内代理，只要骨架也在组内，就递减关联骨架的引用计数；
- (2) 如果组内骨架的引用计数降到了 0，那么骨架就被标记成软性的。否则，它就可以从组外代理到达，并且被标记成硬性的。

图 4.34(a)说明了第一步，在该图中，除了一个骨架之外，所有其他骨架都被标记成了软性的。惟一被标记成硬性的骨架对组外代理是可到达的。

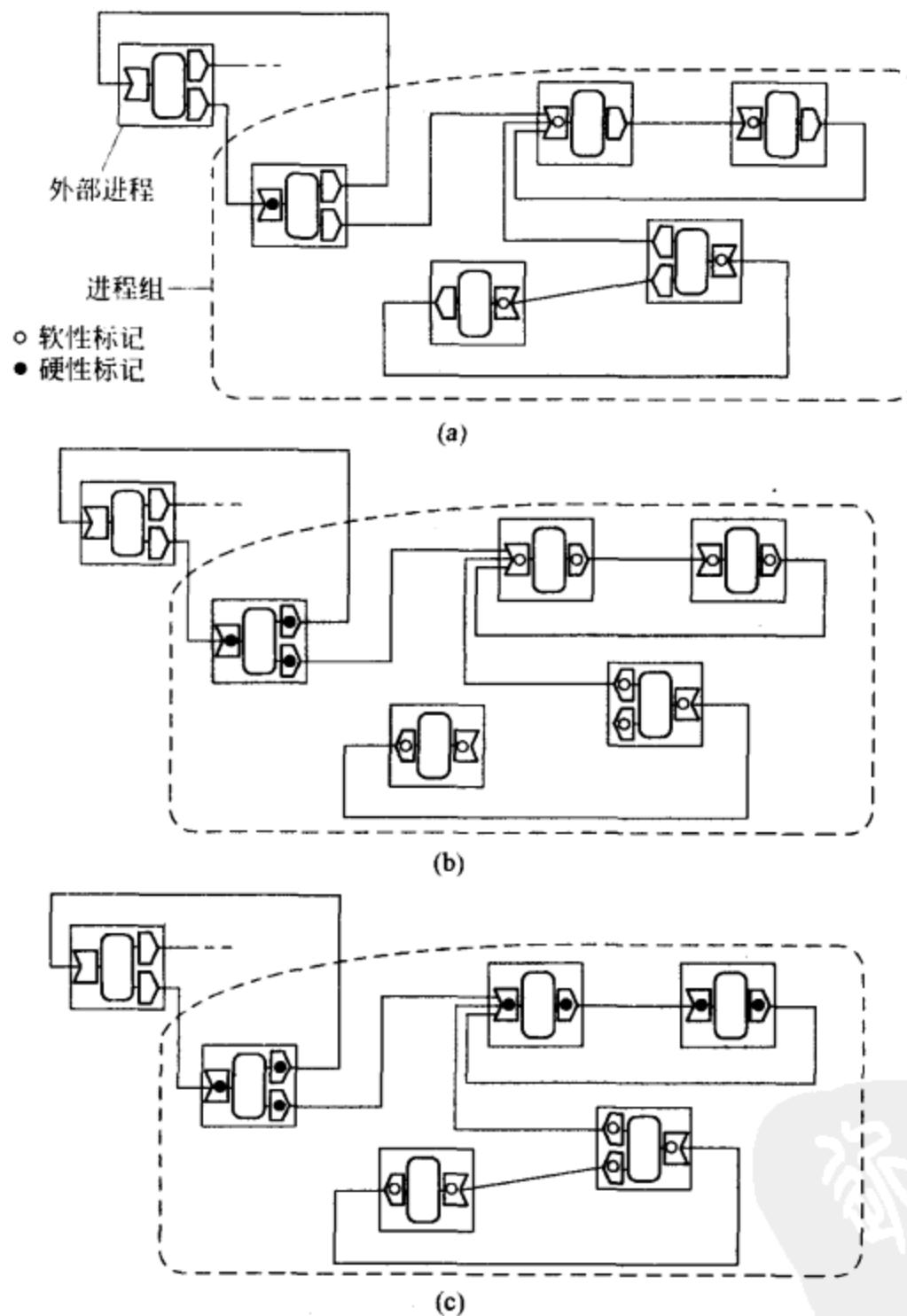


图 4.34 标记骨架的算法
(a) 骨架的初始标记；(b) 在每个进程内完成本地传播以后；(c) 最后的标记

第 2 步是让每个进程都运行自己的本地垃圾收集器。这个收集器的工作方式与全局垃圾收集无关。惟一的要求就是本地垃圾收集器在它运行的进程中从骨架向代理传播标

记。更准确地说，就是假定在单个进程中，可以从骨架到达代理。注意，代理和骨架属于不同的对象。本地标记传播的结果就是代理的标记至少会和骨架一样为硬性的。此外，如果代理可以从根集中的对象到达，那么它应该被标记成硬性的。

可以用下面的方法完成进程 P 内的本地传播。一开始，将 P 中所有的代理都标记为“无”。本地收集器同时从一个骨架集合（这个骨架集合里面的骨架预先都被标记成了硬性的）和根集内的对象开始跟踪。硬性的标记被传播给所有可以从这个集合到达的对象（更准确地说，就是本地对象或代理）。第二次跟踪从那些被标记成软性的骨架开始。如果一个被标记为无的代理现在可以到达，那么它的标记就会被改成软性的。如果代理被标记成了硬性的，那么它就保持它的标记不变。因此，经过本地传播之后，进程中的每个代理都会被标记成无、软性的或是硬性的。图 4.34(b) 显示了对图 4.34(a) 进行本地标记传播以后得到的标记。

第 3 步是从代理向与它们关联的骨架传播标记。换句话说，要在不同的进程之间传播标记。特别是，如果代理被标记成了硬性的，那么应该向与它关联的骨架发送一条消息，告诉骨架如果它还没有标记成硬性的，那么也应该标记成硬性的。只有骨架位于组内时才会发送消息。软性的标记不需要传播，因为初始标记阶段已经让每个组内的骨架标记成软性的或硬性的了。

第 4 步来自于上一步对硬性的标记进行的全局传播。假设进程 P 内的一个骨架现在已经把它的标记从软性的改变成了硬性的。这种改变来自于这样的事实：可以从远程进程的根集对象到达该骨架。因此，首先需要把这种硬性的标记本地传播到 P 内的代理，紧接着，再全局性地传播到相邻的进程。换句话说，只要标记可以被本地或全局传播，就会不断重复步骤 2 和步骤 3。一旦进入稳定状态以后（也就是说，就是组内进程不再发生任何标记变化），这种算法就会进行下一步。在我们的例子中，重复步骤 2 和步骤 3 的结果是产生了图 4.34(c) 所示的最终标记。

第 5 步也就是最后一步的工作是删除不可到达的对象，包括不可到达的代理，还有那些被标记成软性的代理和骨架。重要的一点是要注意被标记为软性的代理和骨架无法从组外到达，也无法从根集内的对象到达。换句话说，由于被标记成软性的代理和骨架只是相互引用，所以应该把它们删除。

垃圾收集实际上可以作为本地传播的副产品来完成。不是在最后一步显式删除实体，而是把骨架的软性的标记改变成无。结果是，可以在以后再次运行本地垃圾收集的时候再收集该骨架。此外，如果与这个骨架有关的对象现在变成了不可到达的，那么同样应该收集它。如果被这个对象本地引用的代理也不再可到达，那么应该把它们标记成无，并保持这样的标记。这样，在向远程进程中与代理有关的骨架发送了一条让计数器减数的消息以后，就可以让本地垃圾收集器安全地收集标记成无的代理了。

通过把进程分层组织到组中，可以为分布式垃圾收集实现一种更加容易扩展的解决方法。其基本思想就是让低层的组收集垃圾，然后把组间引用分析留给更高一层的组。通过让更低层的组减少需要跟踪的对象，更高层的组基本上只需要处理与子组数量大致相同的对象，这样一来，垃圾收集就扩展到了更大的网络上。我们省略了细节问题，要了解这些细节，请参考文献 (Lang 等 1992)。

4.4 小结

名称用来表示实体。实质上,一共有3种类型的名称。地址是与实体关联的访问点名称,也可以简称为实体地址。标识符是另一种类型的名称。标识符拥有三个属性:每个实体都严格地使用标识符引用;一个标识符只能表示一个实体;标识符永远不会再赋给其他实体。最后,易于理解的名称其目标是由人来使用,因此被表示成字符串。

名称在名称空间内组织。名称空间可以用命名图表示,在命名图中,节点代表被命名的实体,边上的标签代表用来了解实体的名称。节点拥有多条外向边,这些边代表实体集合,它们也被称为上下文环境节点或目录。大型的命名图通常组织成有根非循环有向图。

命名图方便地以结构化的方式组织用户友好名称。实体可以使用路径名引用。名称解析是在某个时间通过查找路径名代表的组件来遍历命名图的过程。把节点分散到多台名称服务器上可以实现大型的命名图。在通过遍历命名图来解析路径名时,一旦到达了这一台服务器实现的节点,名称解析就会到下一台服务器上继续。

用于用户友好命名系统不适合经常移动的实体。使用位置无关的标识符可以更为有效地实现对移动实体的定位。大体上有4种用于定位移动实体的方法。

第1种方法是使用广播和多播。实体的标识符被广播给分布式系统的每个进程。为实体提供访问点的进程通过提供访问点的地址进行响应。很明显,这种方法的可扩展性有限。

第2种方法是使用转发指针。每当一个实体转移到另一个位置时,它就会留下一个指针,说明它下一步所在的位置。定位实体需要遍历转发指针形成的路线。为了避免形成太长的指针链,定期缩短指针链是很重要的。

第3种方法就是给实体指定一个起始位置。每当实体转移到另一个地方时,它都会通知起始位置,告诉起始位置自己当前的位置。在定位实体时,首先询问起始位置,以便了解实体的当前位置。

第4种方法是创建一棵分层搜索树。网络划分成不重叠的域。域可以组合成更高层的(不重叠)域,以此类推。只有一个顶级域,它覆盖了整个网络。每个层次的每个域都有关联的目录节点。如果一个实体位于域D中,那么更高一层的域所用的目录节点将拥有一个指向D的指针。最低层目录节点存储了实体的地址。最高层的目录节点知道所有的实体。

应该删除无法定位的实体。在分布式系统中,名称的一种重要应用就是使用这样一种方式组织实体的引用:这种方式就是自动删除无引用实体。这种垃圾收集需要引用计数和跟踪的支持。

在使用引用计数的情况下,实体简单地计算尚在使用的引用数量。当计数器降到0时,就可以把实体删除。作为对引用计数的替代,也可以设置一个引用实体的进程列表。引用列表比引用计数更为健壮一些,不过同样存在可扩展性问题。

在使用基于跟踪方法的情况下,所有被根集实体直接或间接引用的实体都被标记成可到达的。不可到达的实体将被删除。分布式跟踪是难以实现的,这是因为它需要检查

系统的所有实体。有各种各样的解决方法,不过它们一般都基于在单处理器系统中使用的传统垃圾收集器。

习 题

1. 举出一个例子,在这个例子中,为了真正访问实体 E,需要把它的地址进一步解析成另一个地址。
2. <http://www.acme.org/index.html> 这样的 URL 是不是独立的位置? <http://www.acme.nl/index.html> 呢?
3. 请列举一些正确标识符。
4. 在大多数 UNIX 系统中,如何查找挂载点(mounting point)?
5. Jade 是一种分布式文件系统,它使用按用户分配名称空间(Rao 和 Peterson 1993)的方式。换句话说,就是每个用户都拥有自己的私有名称空间。这种名称空间的名称是否可以用于在两个不同的用户之间共享资源?
6. 在 DNS 中,一个子域被实现成了不同于当前域的另一个区域,为了引用这个子域的节点 N,需要指定用于该区域的名称服务器。是否总是需要包括用于该服务器地址的资源记录? 还是只提供域名就足够了?
7. 标识符是否可以包含它所引用实体的信息?
8. 简要介绍一下全局惟一标识符的有效实现。
9. 举例说明 URL 闭合机制的工作方式。
10. 说明 UNIX 系统中硬链接与符号链接之间的区别。
11. DNS 中的高层名称服务器(更准确地说,就是那些在接近根的 DNS 名称空间中实现节点的名称服务器)一般不支持递归名称解析。如果让它们支持递归名称解析,那么是否可以进行大幅度的性能改善?
12. 说明如何使用 DNS 来实现定位移动实体所需要的基于起始位置的方法。
13. 有一种定位实体的特殊方式,它被称为 anycasting,通过这种方式,可以使用 IP 地址标识服务(请参考文献 Partridge 等 1993)。向 anycast 地址发送请求时,实现该 anycast 地址所标识服务的服务器会返回一条响应消息。简要说明如何实现基于分层定位服务(在第 4.2.4 节介绍)的 anycast 服务。
14. 在专门用于分层定位服务的两层基于起始位置方法中,根在哪里?
15. 假设某个移动实体几乎从不会离开域 D,即使真的离开了,它也会很快返回。如何使用这条信息在分层定位服务中加快查询操作的速度?
16. 在深度为 k 的分层定位服务中,当移动实体改变它的位置时,最多需要更新多少条位置记录?
17. 假设一个实体从位置 A 转移到 B,期间经过了几个中间位置,在每个中间位置都只停留相对较短的时间。到达 B 后,它停留了一段时间。在分层定位服务中更改地址可能还需要相对较长的时间,因此在访问中间位置时应该避免更改地址。那么,当实体位于中间位置时应该如何查找它?

18. 在分布式引用计数中,如果从进程 P1 向进程 P2 传递远程一个引用,那么是否可以让 P1 代替 P2 递增计数器?
19. 假设通信是可靠的,解释一下为什么加权引用计数要比简单引用计数更有效。
20. 在世代引用计数中,如果一个对象仍然拥有引用,但是这些引用属于对象不知道的世代,那么是否可以作为垃圾收集该对象。
21. 在世代引用计数中,项 $G[i]$ 是否可能小于 0?
22. 在引用列表方式下,如果向进程 P 发送了 ping 消息以后没有收到任何响应,那么将从对象的引用列表中删除该进程。删除该进程是否始终正确?
23. 在 Lang 等人介绍的基于跟踪垃圾收集器中,最后会达到一种稳定阶段,请说明一种能够确定这个阶段已经形成的简单方法。

第 5 章 同步

在前面的几章中,我们已经讨论了进程及进程间的通信。虽然进程间通信很重要,但它并不是分布式系统的全部内容。与之紧密相关的问题是进程间如何协作和同步。进程协作通过命名方式得到部分支持,进程协作使得进程可以共享资源,或者一般地说是共享实体。

本章中,我们主要集中精力研究进程间是如何同步的。例如,多个进程不能同时访问一个共享资源(例如打印机),而是相互协作,彼此授权暂时地独占访问。另一个例子是多个进程有时可能需要就事件的顺序达成一致,比方说来自进程 P 的消息 m₁ 是在来自进程 Q 的消息 m₂ 之前还是之后被发送出去。

实践表明,分布式系统的同步常常比单处理器或者多处理器系统中的同步更加困难。本章中讨论的问题及其解决方法本质上是比较常见的,它们出现在分布式的许多不同情况下。

我们从基于真实时间的同步问题开始讨论,接着讨论以相对排序的方式而不是以绝对时间的方式来实现的同步。我们也要讨论分布式全局状态的概念,以及如何通过进程同步来记录全局状态。

在许多情况下,一组进程能指定一个进程作为协调者是重要的,这可以通过选举算法来实现。我们将在单独的一节中讨论各种各样的选举算法。

与同步相关的两个主题是分布式系统中的互斥和分布式事务。分布式互斥保护共享资源不被多个进程同时访问。分布式事务也作类似的事情,但是它通过高级的并发控制机制来优化访问。互斥和事务将在单独的小节中讨论。

许多不同的分布式系统有多种风格迥异的分布式算法。许多例子(以及更多的参考资料)可以在文献(Andrews 2000; Singhal 和 Shivaratri 1994; Wu 1998)中找到。大量算法的更为正式的方法可以在文献(Lynch 1996)中找到。

5.1 时钟同步

在集中式系统中,时间是明确的。当进程想知道时间时,它就进行一次系统调用,然后系统内核就会告诉它。如果进程 A 询问时间,稍后进程 B 也询问时间,那么进程 B 获得的时间值将比进程 A 获得的时间值大,或者可能相等,但无论如何不会小于后者。在分布式系统中,取得时间上的一致是不容易的。

作为一个例子,考虑 UNIX 中的 make 程序在缺少全局时间时的情况。通常在 UNIX 系统中会把较大的程序分成多个源文件,当一个源文件发生变化时,只需重新编译

一个文件即可,不需要对所有的文件进行重新编译。如果一个程序包含 100 个文件,在修改一个文件时不需要重新编译所有的文件,这将大大提高编程人员的工作效率。

通常 make 程序工作的方式很简单。程序员修改完了所有的源文件后启动 make 程序,make 程序检查所有的源文件和目标文件最后被修改的时间。如果源文件 input.c 的时间是 2151,而相应的目标文件 input.o 的时间是 2150,那么 make 程序知道在 input.o 创建之后曾经对 input.c 进行过修改,因此必须对 input.c 进行重新编译。相反,如果 output.c 的时间是 2144,而 ouput.o 的时间是 2145,那么这种情况下就不需要重新编译。make 程序就是这样检查所有的源文件,找出需要重新编译的,然后调用编译器重新编译它们。

现在设想一下在没有全局统一时间的分布式系统中会发生什么。假设 output.o 的时间如上为 2144,紧接着 output.c 被修改,但是由于它所在机器上的时钟略慢,结果 output.c 的时间值被赋为 2143,如图 5.1 所示。make 程序将不调用编译器。最终的可执行的二进制程序将包含由旧的源文件和新的源文件所产生的混合目标文件。该程序可能崩溃,而程序员很难找出错误出在哪里。

由于时间是人们思考问题的基础,并且我们已经看到,所有的时钟没有完全同步的后果是如此可怕,所以我们先从一个简单问题开始有关同步的学习:是否有可能对分布式系统中的所有时钟进行同步?

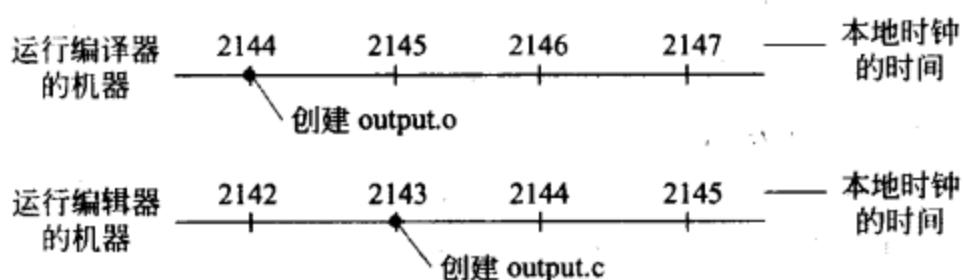


图 5.1 当每台机器都有它自己的时钟时,一个发生于另一个事件之后的事件可能会被标记上较早的时间

5.1.1 物理时钟

几乎所有的计算机都有一个计时电路。尽管一般使用“时钟”这个词来表达这些设备,但它们实际上不是通常意义的时钟,把它们称为计时器(timer)可能更恰当一点。计算机的计时器通常是一个精密加工过的石英晶体。石英晶体在其张力限度内以一定的频率振荡,这个频率取决于晶体本身如何被切割及其受到张力的大小。有两个寄存器与每个石英晶体相关联,一个是计数器(counter),另一个是保持寄存器(holding register)。石英晶体的每次振荡使计数器减 1。当计数器减为 0 时,产生一个中断,计数器从保持计数器中重新装入初始值。这种方法使得对一个计时器进行编程,令其每秒产生 60 次中断(或者以任何其他希望的频率产生中断)成为可能。每次中断称作一个时钟滴答(clock tick)。

系统初次启动时通常要求用户输入日期和时间,然后将它们转换成某一个已知起始时间后的时钟滴答次数,并将它存储在存储器中。许多计算机都有一个用特殊的电池支

持的 CMOS RAM, 其目的是为了以后启动时不再需要输入日期和时间。时钟每滴答一次, 中断服务程序就使存储在存储器里的时间值加 1。用这种方法进行(软)时钟计时。

在单机单时钟的情况下, 如果这个时钟有少许偏差是不会有多大问题的。因为这台机器上的所有进程使用同一个时钟, 所以它们内部仍然会保持一致。例如, 如果文件 input.c 的时间是 2151, 文件 input.o 的时间为 2150, 那么 make 程序将重新编译源文件。即使时钟偏差了 2 个单位, 即两个文件的真实时间应该分别为 2153 和 2152, make 程序仍然会重新编译源文件。真正重要的是相对时间。

一旦引入多 CPU 系统, 每个 CPU 都有自己的时钟, 情况将发生变化。尽管石英晶体振荡的频率通常是相当稳定的, 但仍不可能保证不同计算机中的石英晶体都以完全相同的频率在振荡。实际上, 当一个系统有 n 台计算机时, 所有 n 个晶体都将以略微不同的速度振荡, 导致(软)时钟逐渐不同步; 当同时读这些时钟值时, 将得到不同的值。这种时间值的不同称作时钟偏移(clock skew)。时钟偏移的后果如在上面所举的有关 make 的例子中所看到的那样, 那些期望与文件、对象、进程或消息相关的时间(即所使用的时钟)不仅正确, 而且独立于产生它们的机器的程序可能失败。

在一些系统(例如实时系统)中, 真实的时钟时间是很重要的。对于这些系统, 外部物理时钟是必须的。考虑到效率和冗余, 通常认为使用多个物理时钟比较合适, 但有两个问题: (1) 我们如何使它们与真实世界的时钟同步呢? (2) 我们如何使这些时钟彼此同步?

在回答这些问题之前, 让我们稍稍离题来介绍在实际中是如何测量时间的。它不像人们想像的那样简单, 尤其是当精度要求很高时更是如此。自从 17 世纪机械钟发明以来, 人们就一直用天文学的方法测量时间。每天太阳都是从东方地平线上升起, 然后升到天空的最高处, 最后再落到西边。太阳到达天空中它出现的最高点时称为中天(transit of the sun), 它在每天正午发生。两次连续的太阳中天之间的时间称为一个太阳日(solar day)。因为每天有 24h(小时), 每小时有 3600s, 所以一个太阳(solar second)被精确地定义为 1/86 400 个太阳日。平均太阳日的几何算法如图 5.2 所示。

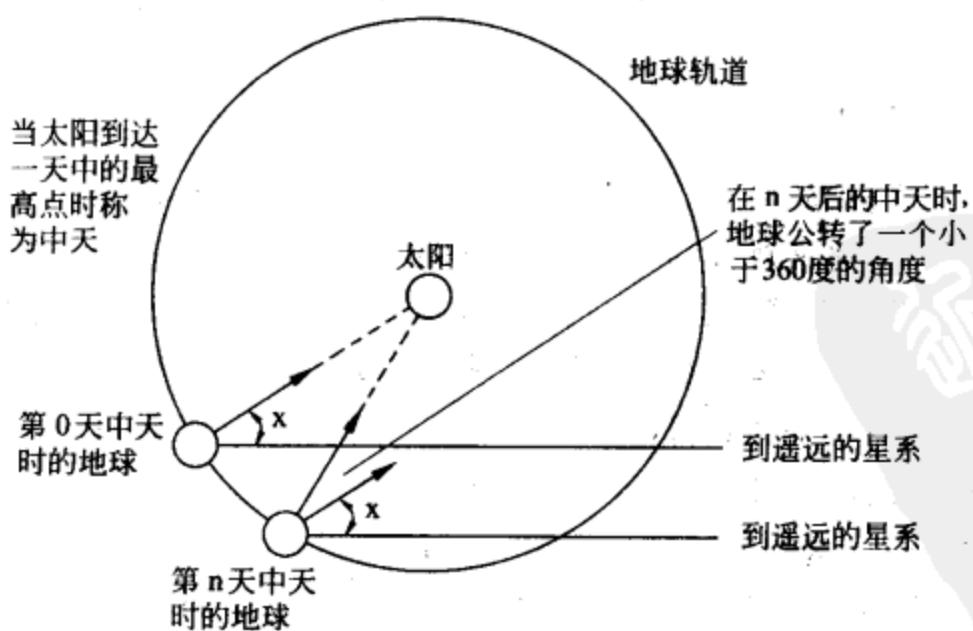


图 5.2 平均太阳日的计算

在 20 世纪 40 年代, 科学家们证实了地球的自转周期并非常数。由于潮汐摩擦和大气的阻力, 地球自转速度正在变慢。基于对远古时代珊瑚的生长图案的研究, 地质学家现在相信, 在 3 亿年前每年大约有 400 天。年的长度(地球绕太阳一周的时间)被认为是不变的, 那么每天的时间就简单地变长了。除了这种长期变化趋势, 也存在一天长短的短期变化, 这可能是由地球地核层熔岩的剧烈沸腾引起的。这些发现促使天文学家们在计算天的长度时测量很多天的长度, 然后对它们取平均值; 最后除以 86 400 得到的结果称为平均太阳秒(mean solar second)。

1948 年原子时钟诞生, 它使更精确地测量时间成为可能。原子时钟不受地球的摆动和振动的影响, 而是通过铯 133 原子的跃迁计时。物理学家从天文学家的手中接管了计时的工作, 定义 1 秒是铯 133 原子作 9 192 631 770 次跃迁所用的时间。选择 9 192 631 770 是为了使原子秒与引入原子秒那一年的平均太阳秒相等。目前, 世界上大约有 50 个实验室拥有铯 133 时钟。每个实验室都定期向巴黎的 Bureau International de l'Heure(BIH) 报告其时钟的滴答次数。BIH 将这些值平均起来产生国际原子时间(international atomic time), 简称为 TAI。这样 TAI 就是铯 133 时钟从 1958 年 1 月 1 日午夜(起始时间)以来被 9 192 631 770 除后的平均滴答数。

尽管 TAI 相当稳定, 并且任何人只要愿意都可以买到一只铯时钟, 但是它仍然存在一个严重的问题, 那就是 86 400 个 TAI 秒现在比一个平均太阳日少 $3\mu\text{s}$ (因为平均太阳日越来越长)。使用 TAI 计时将意味着多年以后, 中午会出现得越来越早, 直到最终出现在凌晨。人们也许会注意到这一点, 今后可能将会发生与 1582 年古罗马教皇 Pope Gregory 十三世宣布从日历中删除 10 天时类似的情况。这一事件导致了街头暴动, 因为地主要收一个整月的地租, 银行家要收一个整月的利息, 而雇主拒绝向雇员支付他们没有工作的那 10 天的工资, 这里只是提到了其中的几个冲突。在一些新教区, 人们拒绝任何与教皇法令有关的事情, 有长达 170 年不接受罗马教皇颁布的日历。

BIH 通过引入闰秒(leap second)解决了该问题, 即当 TAI 和太阳秒计时之间的差增加到 800 微秒时使用一次闰秒。闰秒的使用如图 5.3 所示。这种修正产生了一种时间系统, 该时间系统基于恒定长度的 TAI 秒, 但是却和太阳的运动保持一致; 它被称作统一协调时间(universal coordinated time), 简称为 UTC。UTC 是所有现代人计时的基础。它从根本上取代了原有的标准, 即格林尼治平均时间(greenwich mean time), 后者是一种天文时间。



图 5.3 TAI 秒长度恒定, 与太阳秒不同。当需要与太阳保持一致时引入闰秒

大多数电力公司将他们的 60Hz 或者 50Hz 时钟的计时基础放在了 UTC 上。因此当 BIH 宣布闰秒后,电力公司将他们使用的频率分别从 60Hz 或者 50Hz 增加到 61Hz 或者 51Hz,以便于把他们分布地区的时钟向前拨。因为对于计算机来说 1 秒是一个相当大的时间间隔,所以需要在几年间保持精确时间的操作系统必须有专门的软件根据闰秒的定义来计算闰秒(除非使用电力线的频率来计时,而这种计时方法通常很粗糙)。到目前为止,使用闰秒的次数大约是 30 次。

为了向需要精确时间的人们提供 UTC,NIST(National Institute of Standard Time,国际时间标准化组织)使用了一个位于科罗拉多州柯林斯堡的称为 WWV 的短波广播站。WWV 在每个 UTC 秒开始时广播一个小脉冲。WWV 本身的准确度为±1ms,但由于随机空气波动能影响信号传播路径的长度,实际的精确度仅能达到±10ms。英格兰的 MSF 站(位于沃里克郡的拉格比)也提供类似服务,其他一些国家也有类似的站点。

一些地球卫星也提供 UTC 服务。同步环境操作卫星(geostationary environment operational satellite)能提供精确到 0.5ms 的 UTC,其他一些卫星提供的精度甚至更高。

使用短波广播或者卫星服务需要准确地了解发送信号和接收者的相对位置,以便对信号传输延迟进行补偿。接受 WWV、GEOS 和其他 UTC 信号的无线电接受器在市场上可以买到。

5.1.2 时钟同步算法

如果一台机器上有 WWV 接收器,那么我们需要使其他所有的机器与它同步。如果没有一台机器有 WWV 接收器,每台机器都按自己的机器时间计时,这时我们的目标是尽可能使所有机器的时间保持同步。专家们已经提出了许多同步算法(例如 Cristian 1989, Drummond 和 Babaoglu 1989, Kopetz 和 Ochsenreiter 1987)。在文献(Ramanathan 等 1990)中给出了各种算法的综述。

所有算法都有相同的系统基础模型,我们下面对其进行描述。假设每台机器都有一个每秒产生 H 次中断的计时器。当计时器产生中断时,中断处理程序将软件时钟加 1,软件时钟记录从过去某一约定时间开始的滴答(中断)数。我们将这个时钟值称为 C 。更具体地说,当 UTC 时间为 t 时,机器 P 上的时钟值为 $C_p(t)$ 。最理想的情况是对所有的 p 和 t ,都有 $C_p(t)=t$;换言之, dC/dt 的理想值为 1。

真正的计时器并不是每秒精确地中断 H 次。理论上,当 $H=60$ 时,计时器应每小时产生 216 000 个滴答。实际上,现代计时器芯片所能达到的错误率大约是 10^{-5} ,这意味着一台具体机器的计时器每小时可以产生的滴答次数在 215 998 到 216 002 之间。更准确地说,若存在某一常数 ρ ,使得: $1-\rho \leq \frac{dC}{dt} \leq 1+\rho$ 。

那么可以说计时器在它的规定范围内工作。这个常量 ρ 由厂商规定,称为最大偏移率(maximum drift rate)。稍慢的、精确的及稍快的时钟如图 5.4 所示。

若两个时钟以相反的方向偏离 UTC,在它们同步后的 Δt 时,它们间的最大可能差值是 $2\rho\Delta t$ 。如果操作系统设计者要保证每两个时钟之间的差值不超过 δ ,那么时钟必须至少每 $\delta/(2\rho)$ 秒重新同步一次。各种算法的差异在于实现重新同步的方法不同。

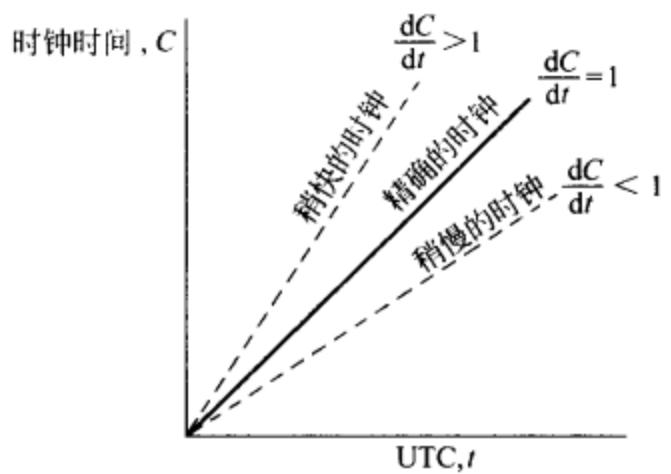


图 5.4 当时钟以不同的速率滴答时,时钟时间和 UTC 之间的关系

1. Cristian 算法

现在开始讨论一种算法,它适合于只有一台机器上有 WWV 接收器,而其他所有机器都要与那台机器同步的系统。我们将拥有 WWV 接收器的机器称为时间服务器(time server)。这种算法基于 Cristian(1989)的工作和以前的一些工作。每台机器以不大于 $\delta/2\rho$ 秒的周期定期地向时间服务器发送消息询问当前时间。时间服务器接到消息后就尽快发送含有当前时间 C_{UTC} 的消息来应答,如图 5.5 所示。

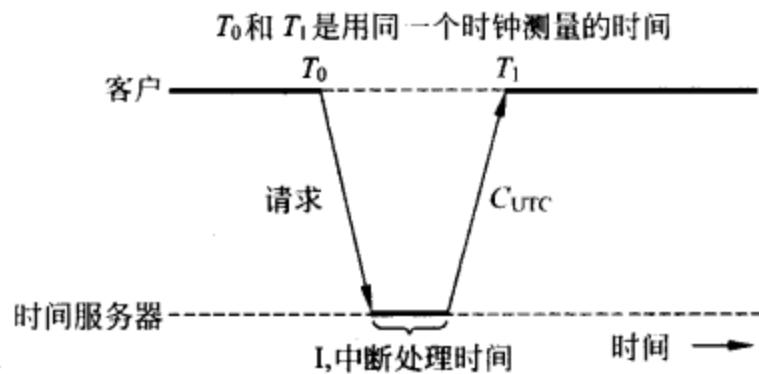


图 5.5 从一个时间服务器得到当前时间

第一种近似情况是发送请求者在得到应答后,就将它的时钟设为 C_{UTC} 。然而这种算法有两个问题:一个主要,一个次要。主要的问题是时间决不能倒退。如果发送请求者的时钟快,那么 C_{UTC} 小于发送请求者的时间值 C 。只是简单地接受 C_{UTC} 将会导致严重的错误,例如在时钟改变后,编译产生的目标文件的时间会早于时钟改变前源文件的修改时间。

这种改变必须逐步进行。一种方法如下:假设计时器每秒产生 100 次中断。正常情况下,每次中断将时间加 10ms。当需要慢下来时,中断服务程序每次仅加 9ms,直到调好为止。与此类似,时钟要加快时,每次中断服务程序将时钟加 11ms,而不是立即将时间调到所需的值。

次要问题是从时间服务器发送的应答到达发送请求者需要花费一定时间。然而更糟的是这种延迟可能较大,而且随着网络负载的改变而改变。Cristian 的处理方法是试图测量这个延迟值。发送者很容易准确地记录从向时间服务器发送请求到接收到应答的时

间间隔。起始时间 T_0 和结束时间 T_1 都使用同一个时钟来测量，所以即使发送请求的时钟与 UTC 有较大偏差，它所测得的时间间隔仍然是比较准确的。

在没有其他任何信息时，消息传送时间的最佳估计值是 $(T_1 - T_0)/2$ 。当应答消息到达时，消息中的时间值加上此值就得到了时间服务器当前时间的估计值。如果知道了理论上最小的传送时间，那么时间估计的其他属性也可以推算出来。

如果知道时间服务器处理中断和处理收到的消息的近似时间，那么估计的准确度还能进一步提高。假设中断处理时间为 I ，那么用于传输的总时间为 $T_1 - T_0 - I$ ，单向传输时间估计为总时间的一半。也有一些系统，其中从 A 到 B 的消息和从 B 到 A 的消息传输的路径不同，因此往返传输时间就会有所不同，但我们这里不考虑这样的系统。

为了提高精度，Cristian 建议不要只做一次测量，而要做一系列的测量，然后丢弃那些 $T_1 - T_0$ 超过某一阈值的测量值，因为这些值是由于网络阻塞而出现的，是不可靠的。对剩余的测量值取平均值会得到较好的估计值。另外，可以把最快返回的消息认为是最精确的，因为该消息遇到的阻塞最少，所以它最能代表纯粹的传输时间。

2. Berkeley 算法

在 Cristian 算法中，时间服务器是被动的。其他机器定期向时间服务器询问时间，时间服务器所做的就是回答它们的请求。Berkeley UNIX 系统采取了完全相反的方法 (Gusella 和 Zatti 1989)。该系统中的时间服务器(实际上是时间守护程序)是主动的，它定期地询问每台机器的时间。基于这些回答，它计算出一个平均时间，并告诉所有其他机器将它们的时钟拨快到一个新的时间，或者拨慢，直到某个指定的减少量达到为止。这种方法适合于没有 WWV 接收器的系统。时间守护程序的时间必须由操作者定期手工设置。图 5.6 描述了这种方法。

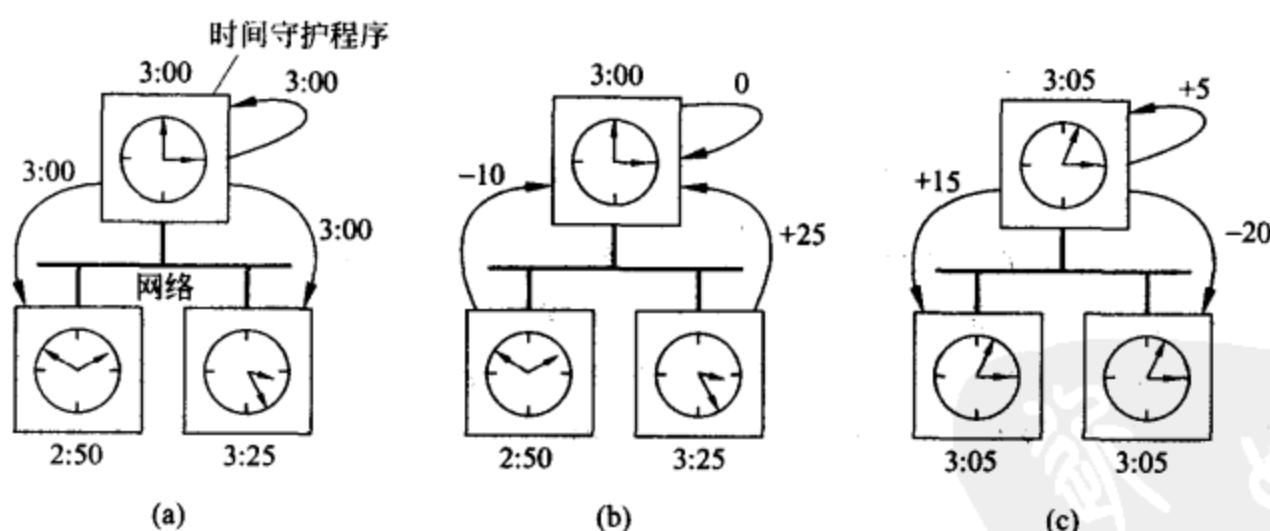


图 5.6 Berkeley 时钟同步算法

- (a) 时间守护程序向所有其他机器询问时钟值；(b) 其他机器做出应答；
- (c) 时间守护程序通知每台机器如何调整它们的时钟

在图 5.6(a)中，在 3:00 时，时间守护程序把它的时间告诉其他机器，并询问它们各自的时间。在图 5.6(b)中，各台机器将它们各自的时间与时间守护程序时间的差值告诉时间守护程序。有了这些值，时间守护程序计算出它们的平均值，并通知各台机器如何调

整各自的时钟,如图 5.6(c)所示。

3. 平均值算法

上述两种算法都是高度集中的,并具有集中式算法常有的不足之处。也有一些非集中式算法。一种非集中式时钟同步算法是将时间划分成固定长度的再同步间隔。第 i 次间隔开始于 $T_0 + iR$,结束于 $T_0 + (i+1)R$,这里的 T_0 是过去某一约定的时间, R 是系统参数。在每次间隔的开始处,每台机器根据自己的时钟广播当前时间。由于不同机器上的时钟不可能完全同速工作,所以这些广播将不会正好同时发生。

在一台机器广播了它的时间后,它启动本地计时器接收在某个时间间隔 S 里到达的其他所有广播。当所有广播到达后,执行一个算法从这些值中计算得到一个新的时间值。最简单的算法是取其他所有机器值的平均值。这个方法的稍微不同的变种是先除去 m 个最高值和 m 个最低值,然后取剩下的平均值。丢弃两端值可认为是一种自我保护措施,以防受到 $2 \times m$ 个错误时钟发出的毫无意义的时间值的影响。

另一种改进是给每条消息加上一个从源到目的地的传送时间的估计值。该估计值可参考网络的拓扑结构,或统计探测消息的响应时间而得到。

另外一些时钟同步算法在一些文献中讨论(例如,Lundelius-Welch 和 Lynch 1998; Ramanathan 等 1989; Srikanth 和 Toueg 1987)。在 Internet 上使用最广泛的算法是网络时间协议(network time protocol,NTP),文献(Mills 1992)中描述了这一协议。NTP 可以取得 1~50ms 的(世界范围)精度,该精度是通过使用高级的时钟同步算法取得的,其进一步的改进在文献(Mills 1995)中描述。

4. 多个外部时间源

对于要求与 UTC 特别精确地同步的系统,可为之配置多个 WWV、GEOS 或其他 UTC 源的接收器。但是,由于时间源固有的不准确性,以及信号传送路径的不稳定性,操作系统最好确立一个 UTC 范围(时间间隔)。一般来说,不同的时间源会产生不同的 UTC 范围,这就要求与之相连的机器和它们达成一致。

为了达成这种一致,每个具有 UTC 源的处理器定期地广播它的时间范围,例如恰好在每一个 UTC 分钟开始时。没有处理器能够瞬间即得到该时间包,更糟糕的是,传输和接收之间的延迟时间依赖于缆线的距离和包所经过的路由器数,这些对于每一对 UTC 源和处理器都是不同的。还有其他一些因素,如由于多台机器同时在以太网上传输发生冲突造成的延迟。另外,如果一个处理器正在忙于处理以前的包,它可能在相当长的几 μ s 内不去理会到来的时间包,从而增加了时间的不确定性。

5.1.3 使用同步时钟

最近几年,在大规模系统(例如整个 Internet)中同步时钟所需要的硬件和软件已经变得很容易得到了。这种新技术能够使得数百万个时钟得到同步,其偏差仅在几个 UTC ms 之内。利用同步时钟的新算法正在纷纷出现。文献(Liskov 1993)中的一个例子讨论了如何至多一次地将消息传递到服务器上,即使在系统面临崩溃时也是如此。传

统的方法是让每一条消息都带有惟一的一个消息编号,每个服务器存储它所收到的所有消息的编号,这样它就可以区分新消息和重发的消息。该算法的问题是如果服务器发生故障并重新启动,它就会丢失消息编号信息表。另外,还需要考虑应每隔多长时间保存一次消息编号。

利用时间,可以对算法做如下修改。让每个消息都携带一个连接标识符(由发送者选择)和一个时间戳。对于每个连接,服务器在一张表里记录下它所见到的最近一次的时间戳。如果通过某个连接接收到的消息的时间戳早于服务器为该连接保存的时间戳,则认为这个消息是复制品,从而拒绝接受。

为了删除旧的时间戳,每个服务器要一直维护一个全局变量:

$$G = \text{Current Time} - \text{Max Lifetime} - \text{Max Clock Skew}$$

这里 Max Lifetime (最大生存时间)指一条消息能够存在的最长时间, Max Clock Skew (最大时钟偏移)指时钟与 UTC 的最大可能偏差值。任何比 G 早的时间戳都可以安全地从表中删除,这是因为所有如此陈旧的消息都已经不起作用了。如果到来的消息具有一个未知的连接标识符,并且它的时间戳比 G 新,那么就接收该消息;如果它的时间戳比 G 老,就拒绝它,理由是如此陈旧的消息无疑是复制品。实际上, G 是新旧消息的一条分界线。每隔 ΔT ,当前时间就会被写入磁盘一次。

当服务器崩溃并重新启动时,服务器根据存储在磁盘上的当前时间重新计算 G ,并令它增加一个更新周期 ΔT 。接收到的时间戳早于 G 的任何消息都作为一个复制品被拒绝。结果,所有可能在崩溃前接收的消息都被拒绝。虽然一些新的消息可能被不正确地拒收,但是任何情况下,该算法都保证“至多一次”的原则得到了遵守。

除了该算法,Liskov(1993)还描述了如何利用同步时钟来取得高速缓存的一致性,如何在分布式系统身份验证中使用超时票据,以及如何在原子性事务中处理提交。在下面的章节中我们将讨论其中的一些算法。随着计时器的同步逐步得到改进,新的时钟同步的应用必定还会出现。

5.2 逻辑时钟

在许多应用中,只要所有的机器具有相同的时间就够了。这个时间无须与广播中每小时播报的真实时间相同。例如,对于 make 程序的运行,即使当前的真实时间是 10:02,但只要每台机器都认为当前时间是 10:00,它就可以正常运行。对于某类算法而言,重要的是时钟的内部一致性,而不是它们是否与真实时间接近。这类算法通常将时钟称为逻辑时钟(logical clock)。

在一篇著名的论文中,Lamport(1978)阐明了尽管时钟同步是可能的,但它不是绝对必要的观点。如果两个进程不进行交互,那么它们的时钟也无须同步,这是因为即使没有同步也觉察不出来,并且也不会产生问题。他进一步指出,通常重要的不是所有的进程在时间上完全一致,而是它们在事件的发生顺序上要达成一致。在上一节给出的 make 例子中,有用的信息是 input.c 的创建时间比 input.o 的创建时间早还是晚,而不是它们创建的绝对时间。

本节我们将讨论 Lamport 算法,这是一个同步逻辑时钟的算法。我们还要讨论一个称为向量时间戳的 Lamport 扩展算法,Lamport 在文献(Lamport 1990)中扩展了他自己的算法。

5.2.1 Lamport 时间戳

为了同步逻辑时钟,Lamport 定义了一个称作“先发生”(happens-before)的关系。表达式 $a \rightarrow b$ 读作“ a 在 b 之前发生”,意思是所有进程一致认为事件 a 先发生,然后事件 b 才发生。这种先发生关系有两种情况:

- (1) 如果 a 和 b 是同一个进程中的两个事件,且 a 在 b 之前发生,则 $a \rightarrow b$ 为真。
- (2) 如果 a 是一个进程发送消息的事件,而 b 为另一个进程接收这个消息的事件,则 $a \rightarrow b$ 也为真。消息不可能在发送之前就被接收,也不能在发送的同时被接收,这是因为消息需要一定时间才能到达接收端。

先发生关系是一个传递关系,所以若 $a \rightarrow b$ 且 $b \rightarrow c$,则 $a \rightarrow c$ 。如果事件 x 和 y 发生在两个互不交换消息的进程中(也不通过第三方间接交换消息),那么 $x \rightarrow y$ 不真, $y \rightarrow x$ 也同样不真。这两个事件称为并发的(concurrent),这意味着无法说(或者不必说)这两个事件什么时候发生,哪个事件先发生。

我们需要一种测量时间的方法,使得对于每个事件 a ,我们都能为它分配一个所有进程都认可的时间值 $C(a)$ 。这些时间值必须具有如下性质:如果 $a \rightarrow b$,那么 $C(a) < C(b)$ 。重述我们前面陈述的情况,若 a 和 b 是同一进程中的两个事件,且 a 在 b 之前发生,则 $C(a) < C(b)$ 。与此类似,如果 a 是一个进程发送消息的事件,而 b 是另一个进程接收该消息的事件,那么 $C(a)$ 和 $C(b)$ 必须赋予大家都认可的具有 $C(a) < C(b)$ 关系的值。另外,时钟时间值 C 必须总是前进(增加),不能倒退(减少)。校正时间的操作是给时间加上一个正值,而不能是减掉一个正值。

现在我们来看一看 Lamport 算法怎样为事件分配时间。考虑如图 5.7(a)所示的三个进程。各个进程运行在不同的机器上,每台机器都有自己的时钟,它们以各自不同的速率工作。如图 5.7 所示,当进程 0 的时钟滴答了 6 次时,进程 1 的时钟滴答了 8 次,进程 2 的时钟则滴答了 10 次。每个时钟均以一个不变的速率工作,但是由于晶体之间的差异,各时钟的工作速率不同。

在时刻 6,进程 0 将消息 A 发送给进程 1,消息的传输时间取决于信任哪个时钟。不管怎样,当它到达时,进程 1 的时钟值为 16。如果消息携带有自身的起始时间 6,那么进程 1 将会推算该消息从进程 0 到进程 1 需要 10 个滴答。这个值的确是可能的。依此类推,消息 B 从进程 1 到进程 2 需要 16 个滴答,这也是可能的。

现在出现了有趣的问题。从进程 2 到进程 1 的消息 C 在时刻 60 离开,却在时刻 56 到达。与之类似,从进程 1 到进程 0 的消息在时刻 64 离开,却在时刻 54 到达。这些值显然是不可能的。必须防止这种情况出现。

Lamport 的解决方法直接遵循先发生关系。既然 C 在时刻 60 离开,那么它只能在时刻 61 或更晚的时刻到达。所以每个消息都应携带依据发送者时钟的发送时间。当消息到达并且接收者时钟显示的时间值比消息的发送时间早时,接收者就将它的时钟调到一

个比发送时间大 1 的值。在图 5.7(b)中,我们看到 C 现在到达的时间是 61。与此类似,D 到达的时间是 70。

对这个算法稍作补充就可以满足全局时间的需要。即在每两个事件之间,时钟必须至少滴答一次。如果一个进程以相当快的速度连续发送或接收两个消息,那么它的时钟必须在这之间至少滴答一次。

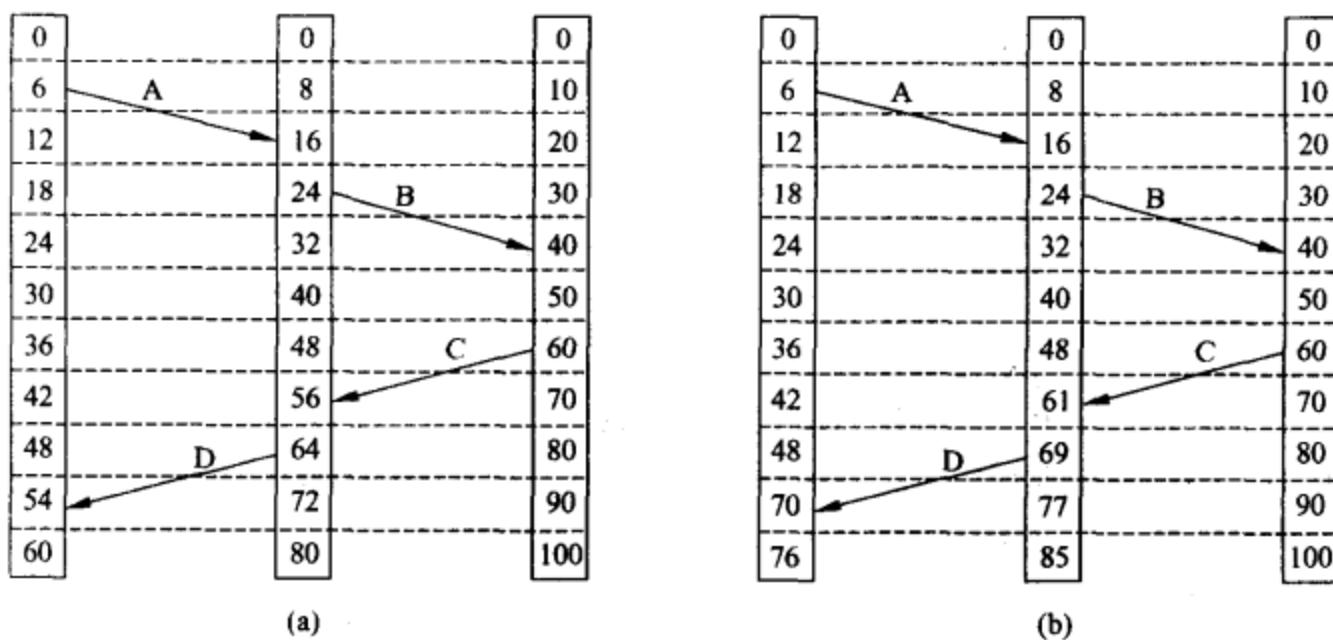


图 5.7 Lamport 算法校正三个进程的不同时钟

(a) 三个进程,每个进程都有自己的时钟,这些时钟以不同的速率工作;(b) Lamport 算法校正时钟

在某些情况下还需要一个附加条件,即两个事件不会精确地同时发生。为了达到这个目标,我们可以将事件发生所在的进程号附加在时间的低位后,并用小数点分开。这样,如果进程 1 和进程 2 中的事件都发生在时刻 40,那么前者记为 40.1,后者记为 40.2。

使用这种方法,我们现在有了一个为分布式系统中的所有事件分配时间的方法,它遵循下面的规则:

- (1) 若在同一进程中 a 在 b 之前发生,则 $C(a) < C(b)$ 。
- (2) 若 a 和 b 分别代表发送一个消息和接收该消息的事件,则 $C(a) < C(b)$ 。
- (3) 对于所有不同的事件 a 和 b , $C(a) \neq C(b)$ 。

这个算法为我们提供了一种对系统中所有的事件进行完全排序的方法。许多其他的分布式算法都需要这种排序以避免混淆,所以文献中广泛引用此算法。

例子: 全序多播

作为 Lamport 时间戳的一个应用,让我们来考虑一下已经把一个数据库复制到几个站点的情况。例如,为了提高查询的效率,银行可能在两个不同的城市都保存一个账户数据库的拷贝,例如纽约和旧金山。一个查询总是被发送到最近的那个拷贝。查询的快速应答在一定程度上是以较高的更新开销为代价的,这是因为每次更新操作都必须在每个副本上执行。

事实上,对更新的要求更严格。假设一位旧金山的客户想在他的账户中存进 \$100,且当前他账户上有 \$1000。与此同时,纽约的银行职员要向该账户加进 1% 的利息。此

数据库的两个拷贝都应该执行这两次更新操作。但是,由于网络中的传输延迟,这两次更新可能以图 5.8 所示的顺序到达。

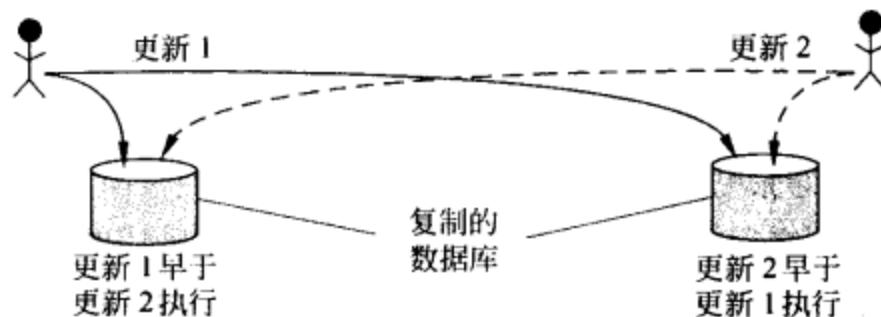


图 5.8 更新一个复制的数据库使它处于了一种不一致的状态

在旧金山,客户存款数的更新操作是在利息加入之前完成的。相反,纽约的账户拷贝则是先加入了 1% 的利息,然后存入了 \$100。结果,旧金山的数据库记录的存款总数是 \$1111,而纽约的数据库记录的存款总数是 \$1110。

我们面临的问题是在每个拷贝上的两个更新操作应该以相同的顺序执行。尽管先存款还是先更新利息有很大的不同,但是对于一致性来说,遵循哪个顺序都是无关紧要的。重要的是两个拷贝要完全相同。一般来说,像上面的情况需要进行一次全序多播 (totally-ordered multicast),即一次将所有的消息以同样的顺序传送给每个接收的多播操作。Lamport 时间戳可以用于以完全分布式的方式实现全序多播。

考虑一组彼此互相多播消息的进程。每个消息都以它的发送者的当前逻辑时间作为时间戳。当一个消息被多播时,理论上它也被传送给它的发送者。另外,我们假定来自同一个发送者的消息以它们被发送的顺序被接收,并且没有消息丢失。

进程接收到一个消息后,将它放进一个本地队列中,并根据它的时间戳进行排序。然后接收者向其他所有进程广播一个确认消息。如果我们按照 Lamport 算法来校正本地时钟,那么接收到的消息的时间戳总是早于确认消息的时间戳。

该方法令人感兴趣的地方是所有进程最终将有相同的本地队列拷贝。每个消息(包括确认消息)都被广播到所有的进程,并认为被所有的进程接收。回想一下,我们认为消息是以它被发送的顺序被传送的。每个进程根据接收到的消息的时间戳将消息放到它的本地队列中。Lamport 时钟确保没有两个消息具有相同的时间戳,并且时间戳反映了事件的全局一致顺序。

只有当队列中的一个消息处于队列头,并且已经被所有其他进程确认了时,进程才可以将它交付给运行中的应用程序。此时,把消息从队列中删除并传给应用程序,相关的确认消息可以被简单地删除。因为每个进程都有相同的队列,所以在任何地方,所有的消息都以相同的顺序交付。换言之,我们已经建立了全序多播。

5.2.2 向量时间戳

Lamport 时间戳导致分布式系统中的所有事件都要经过排序以具有这样的性质:如果事件 a 在事件 b 之前发生,那么 a 也应该排在 b 之前,即 $C(a) < C(b)$ 。

然而,使用 Lamport 时间戳后,只通过比较事件 a 和 b 各自的时间值 $C(a)$ 和 $C(b)$,

无法说明它们之间的关系。换句话说, $C(a) < C(b)$ 不能说明事件 a 就是在事件 b 之前发生。还需要另外一些信息。

为了便于理解, 我们考虑一个消息传递系统, 该系统中的进程张贴文章, 并对张贴出来的文章做出应答。最流行的消息传递系统之一是 Internet 电子公告板服务——网络新闻(network news), 请参见文献(Comer 2000b)。用户, 或者说进程, 加入特定的讨论组。在组内张贴的内容(无论张贴的是文章还是应答)都会广播到全组的所有成员。为了保证应答在与之相关的张贴之后被传送, 我们可能决定使用上面描述的全序多播方法。然而, 这样的方法不意味着如果消息 B 在消息 A 之后发布, 那么消息 B 就是对消息 A 内容的应答。事实上, 这两个消息可能完全没有关系。在这种情况下, 全序多播的顺序性显得过强了。

问题在于 Lamport 时间戳不能捕获因果关系(causality)。在我们的例子中, 根据因果关系, 应该总是先接收到文章, 再张贴应答。因此, 如果要在一个进程组内维持因果关系, 那么收到一篇文章的应答应该总是刚好发生在收到那篇文章之后。如果两篇文章或者两个应答是彼此无关的, 那么它们的交付顺序就无关紧要。

因果关系可以通过向量时间戳(vector timestamp)来捕获。分配给事件 a 的向量时间戳 $VT(a)$ 具有下列性质: 如果对某一事件 b , 有 $VT(a) < VT(b)$, 那么认为事件 a 在因果关系上处于事件 b 之前。向量时间戳的创建是通过让每个进程 P 维护一个向量 V 来完成的, 该向量具有下面两个性质:

- (1) $V_i[i]$ 是到目前为止进程 P_i 发生的事件的数量;
- (2) 如果 $V_i[j] = k$, 那么进程 P_i 知道进程 P_j 中已经发生了 k 个事件。

第一个性质是通过在进程 P_i 中的新事件发生时递增 $V_i[i]$ 来维护的。第二个性质是通过在所发送的消息中携带向量来维护的。当进程 P_i 发送消息 m 时, 它将自己的当前向量作为时间戳 vt 一起发送。

使用这种方式, 接收者可以得知进程 P_i 中已经发生的事件数。更重要的是, 接收者可以得知在进程 P_i 发送消息 m 之前其他进程已经发生了多少个事件。换句话说, m 的时间戳 vt 告诉接收者其他进程中有多少事件发生在它之前, 并且 m 可能在因果关系上依赖于这些事件。当进程 P_j 接收到 m 时, 它调整自己的向量, 将每项 $V_j[k]$ 设置为 $\max\{V_j[k], vt[k]\}$ 。该向量现在反映了进程 P_j 必须接收的消息数, 该消息数目至少是在发送 m 之前见到的消息。此后将 $V_j[i]$ 项增 1, 这表示接收消息 m 的事件是来自于进程 P_i 的下一个事件(Raynal 和 Singhal 1996)。

只在不违背因果关系限制时, 才能使用向量时间戳来传递消息。我们来再次考虑一下电子公告板的例子。当进程 P_i 张贴一篇文章时, 它将该文章作为消息 a 广播出去, 并且在该消息上附加一个时间戳 $vt(a)$, 其值等于 V 。当另一个进程 P_k 接收到 a 时, 它将调整自己的向量, 以使 $V_k[i] > vt(a)[i]$ 。

现在假设进程 P_j 张贴了一个该文章的回复。回复是通过该进程广播一个消息 r 实现的, 消息 r 携带值等于 V_j 的时间戳 $vt(r)$ 。注意 $vt(r)[i] > vt(a)[i]$ 。假设通信是可靠的, 包含文章的消息 a 和包含回复的消息 r 最终都到达了另一个进程 P_k 。因为我们没有对消息的顺序关系做出假设, 所以消息 r 可能在消息 a 之前到达进程 P_k 。进程 P_k 接收

到消息 r 时检查其时间戳，并决定推迟提交消息 r ，直到因果关系上位于 r 之前的消息都接收到了才提交 r 。消息 r 只有在下列条件满足时才得到交付：

- (1) $vt(r)[j] = V_k[j] + 1$;
- (2) 对于所有满足 $i \neq j$ 的 i 和 j , $vt(r)[i] < V_k[i]$ 。

第一个条件说明 r 是进程 P_k 正在等待的下一条来自进程 P_j 的消息。第二个条件说明当进程 P_j 发送消息 r 时，进程 P_k 只看到被进程 P_j 看到的消息。这意味着进程 P_k 已经看到了消息 a 。

关于有序消息交付的说明

一些中间件系统，尤其是 ISIS 和它的后继 Horus(Birman 和 van Renesse 1994)，提供了对全序和因果排序的可靠多播的支持。对于应由消息通信层支持排序还是由应用层处理排序(请参见 Cheriton 和 Skeen 1993; Birman 1994)存在一些争议。

让通信层处理消息排序存在两个主要问题。首先，因为通信层不能判断消息包含的内容，所以只能捕获潜在的因果关系。例如，来源相同的两个完全独立的消息总是被通信层标记为因果相关的。这种方法过于严格，可能会导致效率问题。

第二个问题是，并不是所有的因果关系都可以捕获。让我们再次考虑一下新闻系统。假设 Alice 张贴了一篇文章。如果随后她打电话给 Bob，告之她刚才所写的内容，Bob 可能没有看见 Alice 发在新闻系统上的文章就发了一篇文章作为回复。换句话说，由于外部通信，Bob 和 Alice 的张贴之间有了因果关系。这个因果关系不会被网络新闻系统所捕获。

本质上，像许多其他特定于应用的通信问题一样，排序问题可以根据通信的具体应用而得到充分的解决。在系统设计中，这也称为端对端参数(end-to-end argument)(Saltzer 等 1984)。只具有应用层解决方案的系统的一个缺点是，开发者不得不关注与应用系统的核心功能不直接相关的问题。例如，当开发一个诸如网络新闻这样的消息系统时，排序问题可能并不是最重要的问题。在这种情况下，让底层的通信层处理排序问题可能比较方便。我们还将多次提到端对端参数，尤其是在讨论分布式系统中的安全问题时。

5.3 全局状态

在许多情况下，知道分布式系统所处的全局状态是很有用的。分布式的全局状态(global state)包括每个进程的本地状态和当前正在传输中的消息，所谓正在传输中的消息即该消息已经被发送但没有被交付。进程本地状态的内容取决于我们所关心的内容(Helary 1989)。在分布式数据库系统的情况下，它可能只包括那些构成数据库的记录，而不包括用于计算的临时记录。在上一章我们讨论的基于跟踪的垃圾收集的例子中，本地状态可以由变量组成，这些变量代表代理的标记、骨架(skeleton)和包含在进程地址空间里的对象。

有很多原因表明，知道分布式的全局状态是有用的。例如，当已知本地计算已经

停止并且也没有消息在传输时,系统显然进入了一个不能再前进的状态。通过分析这样的全局状态,可能得出结论,我们正在处理死锁(参看 Bracha 和 Toueg 1987),或者一个分布式计算已经正确地结束了。下面讨论如何正确地进行这样的分析。

(Chandy 和 Lamport 1985)提出了一个简单直接的记录分布式系统全局状态的方法,该方法引入了分布式快照(distributed snapshot)的概念。分布式快照反映了该分布式系统可能处于的状态。一个重要的性质是分布式快照反映了一个一致的全局状态。这意味着如果已经记录了一个进程 P 收到了来自进程 Q 的一条消息,那么也应该记录了进程 Q 确实已经发送了那条消息。否则,快照将包含已经被接收了但从来没有被发送的消息的记录,这显然不是我们所希望的。然而相反的情况(Q 发送了一条消息,但是 P 没有接收到)则是允许的。

全局状态的概念可以用称为切口(cut)的示意图来表达,如图 5.9 所示。在图 5.9(a)中,穿越进程 P1、P2 和 P3 时间轴的虚线表示的是一个一致的切口。该切口表示了为每个进程记录的最后事件。本例中,很容易验证所有被记录的消息接收事件都有其相应的发送事件。作为对比,图 5.9(b)显示了一个不一致的切口。快照中记录了进程 P3 接收到消息 m2 这一事件,但却没有包含相应的消息发送事件。

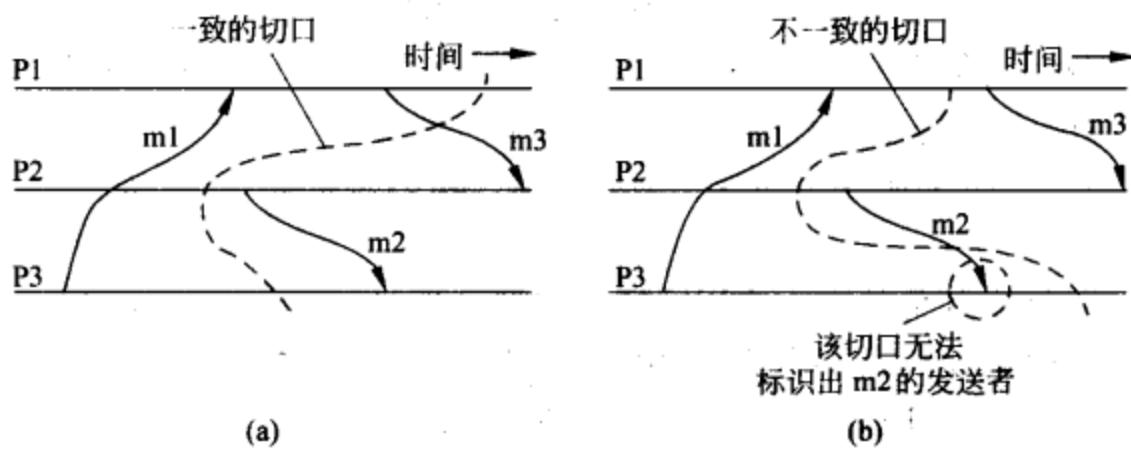


图 5.9 一致的切口和不一致的切口

(a) 一致的切口; (b) 不一致的切口

为了简化对分布式快照捕获算法的解释,我们假设分布式系统可用一个彼此通过单向点对点通信通道相连的进程集合来表示。例如,进程可能在任何进一步通信发生前首先建立 TCP 连接。

任何进程都可以启动该算法。启动算法的进程 P 通过记录它自己的本地状态而启动。然后,它通过每个流出通道发送一个标记,表明接收者应参与记录全局状态。

当进程 Q 通过一个进入通道 C 接收到一个标记时,该进程根据它是否已经保存了本地状态来决定下一步动作。如果尚未保存其本地状态,它就先记录本地状态,然后也通过它自己的每个流出通道发送一个标记。如果 Q 已经记录了本地状态,则通道 C 上的标记表明 Q 应该记录该通道的状态。该状态是由从上次 Q 记录了它自己的本地状态开始,到它接收到该标记为止,Q 所收到的消息序列组成。对这一状态的记录如图 5.10 所示。

当一个进程接收并处理了它的所有进入通道的标记时,就认为该进程已经完成了算法中与它有关的部分。此时可以将它记录的本地状态和它为每个进入通道记录的状态收

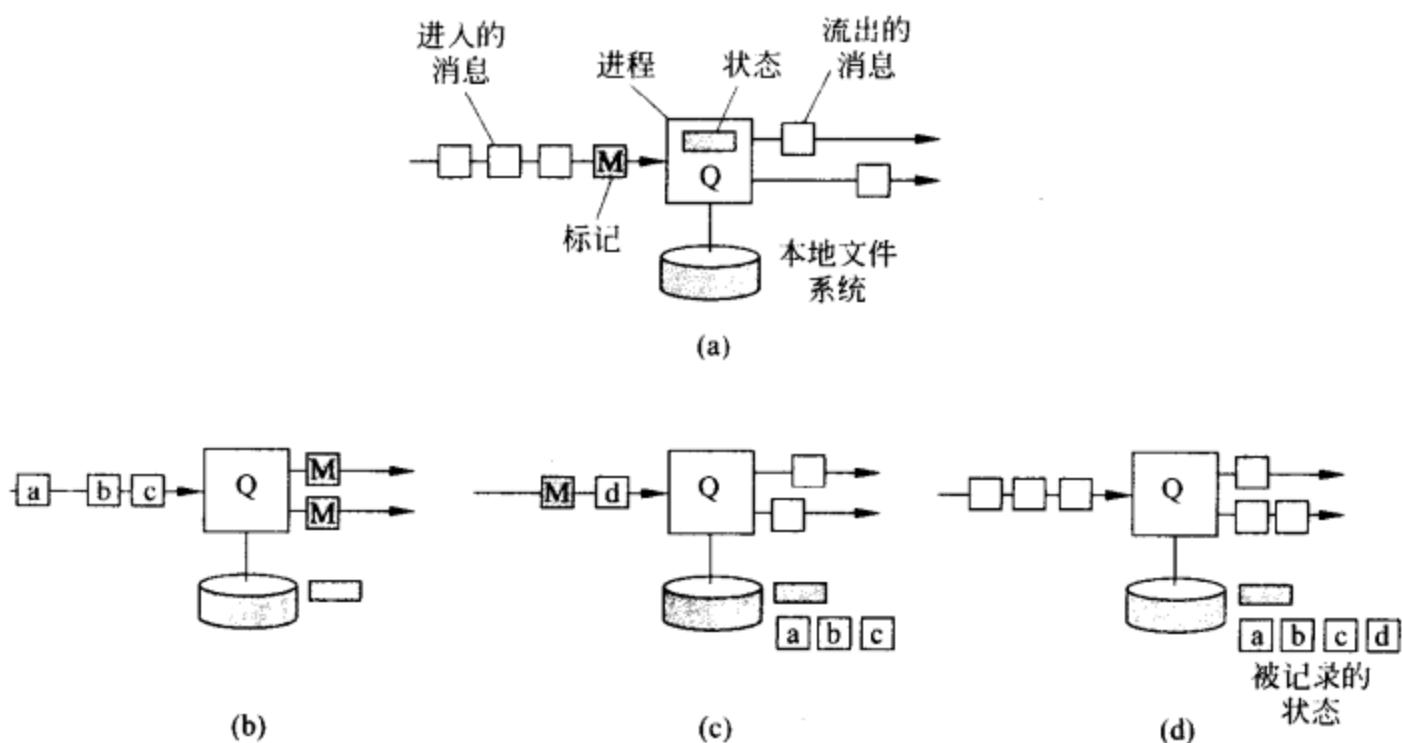


图 5.10

- (a) 进程和分布式快照的通道；(b) 进程 Q 第一次接收到一个标记，并记录它的本地状态；
(c) Q 记录所有进入的消息；(d) Q 在其进入通道中接收到一个标记，并记录该进入通道的状态

集起来，发送给发起此快照的进程。后者随后分析当前状态。注意，与此同时，分布式系统可以作为一个整体继续正常运行。

需要注意的是，因为任何进程都可以发起该算法，所以可能同时存在几个快照。为此，标记上附有发起该快照的进程的标识符（可能还有一个版本号）。只有在进程已经通过它的每个进入通道接收到了某个标记后，它才能完成与该标记相关的快照的创建。

例子：终止检测

作为捕获快照的一个应用，我们来考虑一下如何检测分布式计算的终止。如果进程 Q 第一次接收到了请求快照的标记，它把发送该标记的进程作为它的前趋者（predecessor）。当 Q 完成快照中与它相关的一部分时，它发送给它的前趋者一个 DONE 消息。依次类推，当分布式快照的发起者从它的所有后继者（successor）那里都接收到了一个 DONE 消息时，它知道快照已经完全完成了。

然而，有可能快照显示的是一个其中含有正在处于传输中的消息的全局状态。特别是，假设一个进程记录下：在它记录了本地状态到它通过某一进入通道接收到一个标记为止的这段时间里，在该通道上已经接收到了消息。很显然，在这种情况下，我们不能断定分布式计算已经完成，因为这些消息可能已经产生了不包含在快照中的其他消息。

为了确定分布式计算已经完成，需要一个所有通道都为空的快照。下面是对上面描述的算法的简单修改。当进程 Q 完成快照中与它相关的一部分时，它或者向它的前趋者返回一个 DONE 消息，或者返回一个 CONTINUE 消息。只有当下面两个条件满足时，才会返回 DONE 消息：

- (1) 所有 Q 的后继者都返回了一个 DONE 消息；
- (2) 从 Q 记录了它的状态到通过它的每个进入通道已经接收到了一个标记的这段时

间内,它没有收到任何消息。

在所有其他情况下,Q 给它的前趋者发送一个 CONTINUE 消息。

最后,快照的原始发起者,假设是进程 P,将从其后继者或者接收到一个 CONTINUE 消息,或者仅接收到 DONE 消息。当 P 接收到的所有消息都是 DONE 消息时,表明没有消息处于传输之中,计算已经终止。否则,进程 P 发起另一个快照,继续重复上述操作,直到最终返回的都是 DONE 消息为止。

人们开发了许多类似的终止检测的解决方案。要想获得更多的例子和参考资料,请参见文献(Andrews 2000; Singhal 和 Shivaratri 1994)。对各种解决方案的总结和比较可以在文献(Mattem 1987; Raynal 1988)中找到。

5.4 选举算法

许多分布式算法需要一个进程充当协调者、发起者或者其他某种特殊的角色。通常,由哪个进程充当这个特殊的角色并不重要,重要的是它们中要有一个进程来充当。本节讨论用于选举出一个协调者的算法(这里使用协调者来作为那个特殊进程的通用名字)。

如果所有的进程都完全相同,没有任何区别,那么就无法选择其中的一个作为特殊进程。因此,我们假设每个进程有一个唯一的编号,例如它的网络地址(为了简单起见,我们假设每台机器只有一个进程)。通常,选举算法试着找出进程号最大的进程,将它指定为协调者。各个算法在找出这个进程时使用的方法有所不同。

此外,我们还假设每个进程都知道所有其他进程的进程号。但进程并不知道当前哪些进程正在运行,以及哪些进程崩溃了。选举算法的目标是确保发起选举后,当所有的进程都同意选出的新协调者时选举结束。比较知名的算法有许多,请参见文献(Fredrickson 和 Lynch 1987, Garcia-Molina 1982, Singh 和 Kurose 1994)。

5.4.1 欺负(Bully)算法

第一个例子,研究一下 Garcia-Molina(1982 年)提出的欺负算法(bully algorithm)。当任何一个进程发现协调者不再响应请求时,它就发起一次选举。进程 P 按如下过程主持一次选举:

- (1) P 向所有编号比它大的进程发送一个 ELECTION 消息;
- (2) 如果无人响应,P 获胜成为协调者;
- (3) 如果有编号比它大的进程响应,则由响应者接管选举工作。P 的工作完成。

任何时刻,一个进程只能从编号比它小的进程得到一个 ELECTION 消息。当该消息到达时,接收者发回一个 OK 消息给发送者,表明它仍然在运行,并且接管选举工作。然后接收者主持一个选举。最终,除了一个进程外,其他所有进程都将放弃,那个进程就是新的协调者。它将选举获胜的消息发送给所有进程,通知它们自己是新的协调者。

当一个以前崩溃了的进程现在恢复过来时,它将主持一次选举。如果该进程恰好是当前正在运行的进程中进程号最大的进程,它将赢得此次选举,接管协调者的工作。这样,最大的进程总是取胜,故称为“欺负算法”。

在图 5.11 中,我们看到了欺负算法如何工作的例子。图中是由编号从 0 到 7 的 8 个

进程组成的进程组。以前进程 7 是协调者,但是它崩溃了。进程 4 第一个注意到这一点,所以它发送 ELECTION 消息给所有比它大的进程,即进程 5、6、7,如图 5.11(a)所示。进程 5 和 6 都用 OK 消息进行应答,如图 5.11(b)所示。进程 4 接到第一个应答就知道它的工作已经结束了。它知道有进程号大的进程将接管它的工作成为协调者。它就等着看谁将是获胜者(尽管此时它已经可以猜到结果了)。

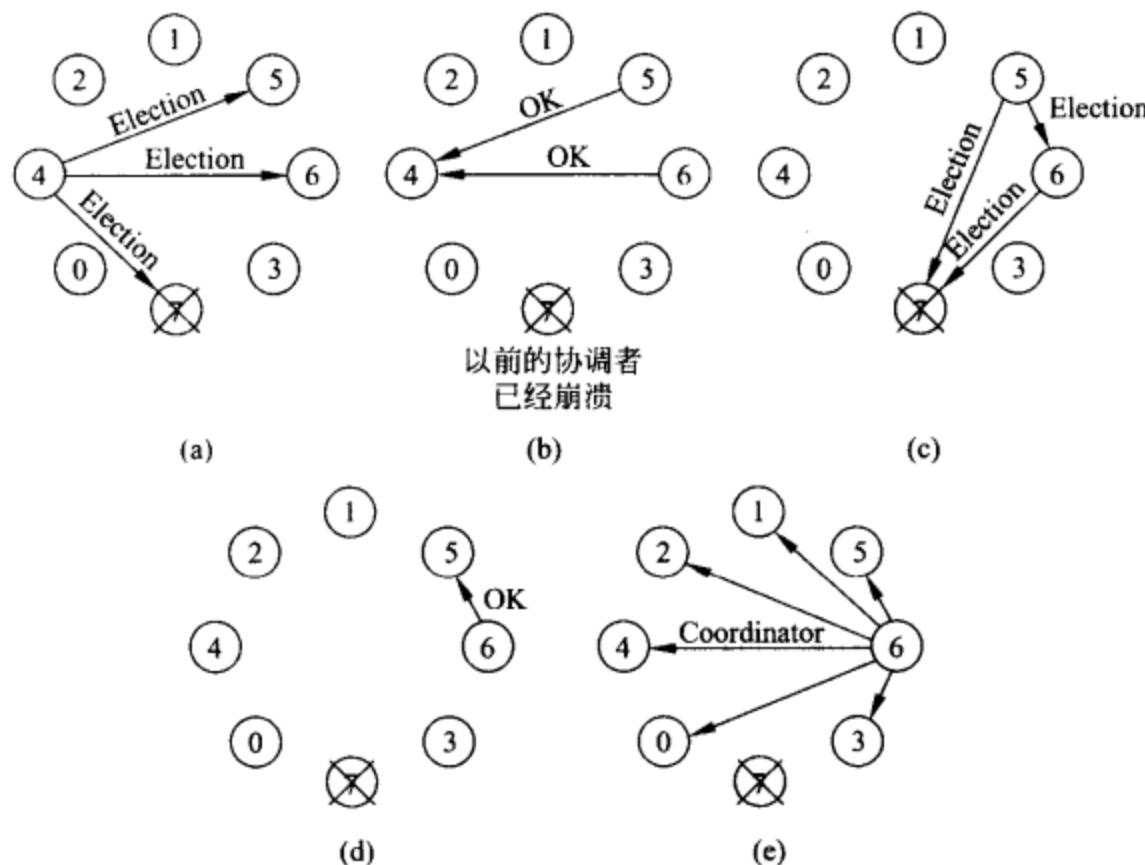


图 5.11 欺负选举算法

(a) 进程 4 主持一个选举; (b) 进程 5 和 6 进行响应,告诉进程 4 停止选举; (c) 进程 5 和 6 此时各自主持一个选举; (d) 进程 6 通知进程 5 停止选举; (e) 进程 6 获胜,并通知每个进程

在图 5.11(c)中,进程 5 和 6 都主持选举,每个进程只将消息发送给比自己进程号大的进程。在图 5.11(d)中,进程 6 告诉进程 5 它将接管成为协调者。此时进程 6 知道进程 7 已经崩溃了,它(进程 6)将是获胜者。如果从磁盘或其他地方可以获得一些表明原有的协调者在哪里失效的状态信息,那么此时进程 6 就必须承担所需工作。当进程 6 准备好接管时,它向所有正在运行着的进程发送一个 COORDINATOR 消息。当进程 4 接收到这个消息后,它就可以继续当发现进程 7 崩溃了时它正要做的工作,但是这次是以进程 6 作为协调者了。这样,进程 7 的故障得到了处理,工作可以继续了。

如果进程 7 重新启动,它将向其他所有进程发送一个 COORDINATOR 消息,让它们服从自己的协调。

5.4.2 环算法

另一个选举算法是基于环(ring)的使用。不像其他一些环算法,该算法不使用令牌。假设进程按照物理或逻辑顺序进行了排序,那么每个进程就都知道它的后继者是谁了。当任何一个进程注意到协调者不工作时,它就构造一个带有它自己的进程号的

ELECTION 消息，并将该消息发送给它的后继者。如果后继者崩溃了，发送者沿着此环跳过它的后继者发送给下一个进程，或者再下一个，直到找到一个正在运行的进程。在每一步中，发送者都将自己的进程号加到该消息列表中，以使自己成为协调者的候选人之一。

最终，消息返回到发起此次选举的进程。当发起者进程接收到一个包含它自己进程号的消息时，它识别出这个事件。此时，消息类型变成 COORDINATOR 消息，并再一次绕环运行，向所有进程通知谁是协调者（成员列表中进程号最大的那个）以及新环中的成员都有谁。这个消息在循环一周后被删除，随后每个进程都恢复原来的工作。

图 5.12 示意了当进程 2 和 5 同时发现以前的协调者进程 7 崩溃了时，将会发生什么。这两个进程各自构造一个 ELECTION 消息，并且让它们相互独立地绕环运行。最后，两个消息都将绕环走过全程，进程 2 和 5 分别将它们转化为 COORDINATOR 消息，这两个消息拥有相同的成员，相互顺序也相同。两个消息再绕环一周后都被删除。有多余的消息循环没有害处，最多是花费了一点带宽，但这也不是浪费。

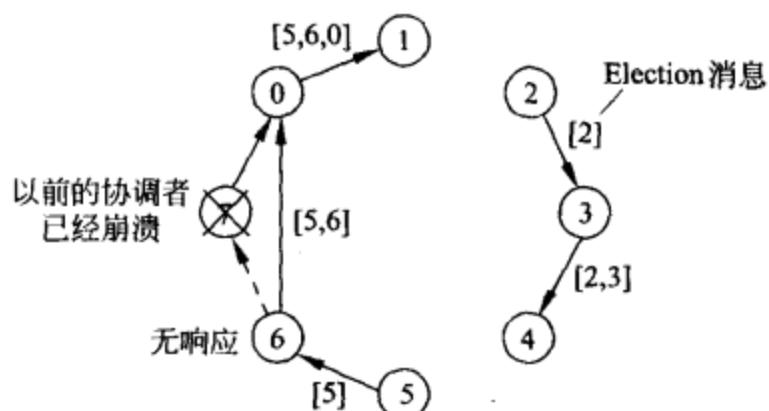


图 5.12 使用环的选举算法

5.5 互 斥

对于包含多进程的系统，使用临界区最容易编程。当一个进程必须读或者更新某个共享数据结构时，它先进入一个临界区以达到互斥，以保证没有其他进程同时使用该共享数据结构。在单处理器系统中，使用信号量、管理（monitor）及类似的结构来保护临界区。我们看几个如何在分布式系统中实现临界区和互斥的例子。关于其他方法的分类和参考书目，请参见文献（Raynal 1991；Singhal 1993）。

5.5.1 集中式算法

在分布式系统中达到互斥的最直接的方法是仿照单处理器系统中的方法，选举一个进程作为协调者（例如，运行在具有最大网络地址号的机器上的进程）。无论何时一个进程要进入临界区，它都要向协调者发送一个请求消息，说明它想要进入哪个临界区并请求允许。如果当前没有其他进程在该临界区内，协调者就发送允许进入的应答消息，如图 5.13(a)所示。收到应答后，该请求进程即进入临界区。

现在假设另一个进程,即进程 2,请求进入同一个临界区,如图 5.13(b)所示。协调者知道一个与此不同的进程已经在临界区内,所以它不能同意该请求。拒绝进入的方法依系统而定。在图 5.13(b)中,协调者只是不进行应答,以便阻塞进程 2,因为进程 2 正在等待应答。另一方面,协调者也可以发送一个“拒绝请求”的应答。不管采用哪种方法,协调者都将进程 2 的请求放到队列中,并等待更多的消息到来。

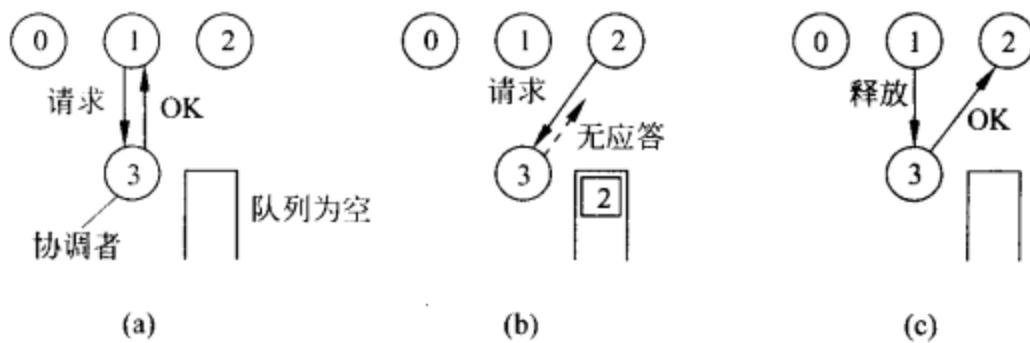


图 5.13

(a) 进程 1 请求协调者允许它进入一个临界区。请求得到了批准; (b) 进程 2 也请求进入同一个临界区。协调者不应答; (c) 进程 1 在退出临界区时通知协调者,协调者然后对进程 2 做出应答

当进程 1 退出临界区时,它向协调者发送一个消息以让协调者释放它的独占访问,如图 5.13(c)所示。该协调者从处于等待状态的请求队列中取出第一个进程,并向该进程发送一个允许进入消息。如果该进程仍然处于阻塞状态(也就是说这是发送给它的第一个消息),它将解除阻塞并进入临界区。如果已经明确发送一个消息拒绝它进入临界区,那么该进程将不断查询进入的消息,或者稍后进入阻塞状态。不管使用了哪种方法,当进程看到允许进入消息时它就能够进入临界区。

显然,该算法保证了互斥的实现,即对于每个临界区,协调者在某一时刻只让一个进程进入。它也很公平,因为允许请求的顺序同接收它们的顺序是一致的。没有进程会处于永远等待状态(不会出现饿死的情况)。此方法也易于实现,每使用一次临界区只需 3 条消息(请求、允许和释放)。该方案不仅能用于管理临界区,也可以用于更一般的资源分配。

集中式方法也存在缺点。协调者是一个单个故障点,所以如果它崩溃了,整个系统可能瘫痪。在一般情况下,如果进程在发出请求之后被阻塞,那么请求者就不能区分“拒绝进入”和协调者已经崩溃这两种情况,因为上述两种情况都没有消息返回。此外,在规模较大的系统中,单个协调者会成为性能的瓶颈。

5.5.2 分布式算法

拥有单个故障点往往是不可接受的,所以研究者开发了分布式互斥算法。Lamport 在 1978 年发表的关于时钟同步的论文中首次提出了一种分布式互斥算法。Ricart 和 Agrawala(1981)对它作了进一步的改进。本节将讨论他们的方法。

Ricart 和 Agrawala 算法要求系统中的所有事件都是完全排序的。也就是说,对于每对事件,比方说消息,哪个事件先发生必须非常明确。5.2.1 节中给出的 Lamport 算法是一种完成这种排序的方法,该算法还能用于为分布式互斥提供时间戳。

该算法的工作过程如下：当一个进程想进入一个临界区时，它构造一个消息，其中包含它要进入的临界区的名字、它的进程号和当前时间。然后它将该消息发送给所有其他的进程，理论上讲也包括它自己。假设消息的传送是可靠的，也就是说，每个消息都能得到确认。如果可能，可以使用可靠的组通信来代替单个消息。

当一个进程接收到来自另一个进程的请求消息时，它根据自己与消息中的临界区相关状态来决定它要采取的动作。可以分为三种情况：

(1) 若接收者不在临界区也不想进入临界区，它就向发送者发送一个 OK 消息。

(2) 若接收者已经在临界区中，它不进行应答，而是将该请求放入队列中。

(3) 如果接收者想进入临界区但尚未进入时，它将对收到的消息的时间戳与包含在它发送给其余进程的消息中的时间戳进行比较。时间戳最早的那个进程获胜。如果收到的消息的时间戳比较早，那么接收者向发送者发回一个 OK 消息。如果它自己的消息的时间戳比较早，那么接收者将收到的请求放入队列中，并且不发送任何消息。

在发送了请求进入临界区的请求消息后，进程进行等待，直到其他所有进程都发回了允许进入消息为止。一旦得到所有进程的允许，它就可以进入临界区了。当它退出临界区时，它向其队列中的所有进程发送 OK 消息，并将它们从队列中删除。

我们来理解一下该算法的工作原理。如果没有冲突，该算法显然正常工作。然而，假设两个进程同时试图进入同一个临界区，如图 5.14(a) 所示。

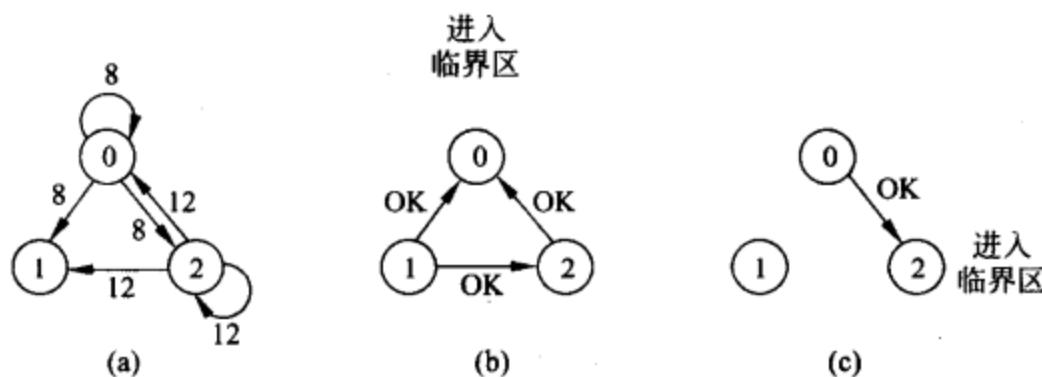


图 5.14

(a) 两个进程同时希望进入同一个临界区；(b) 进程 0 具有最早的时间戳，所以它获胜；

(c) 当进程 0 退出临界区时，它发送一个 OK 消息，所以进程 2 现在可以进入临界区

进程 0 给每个进程发送一个具有时间戳 8 的请求，然而与此同时，进程 2 给每个进程发送了一个具有时间戳 12 的请求。进程 1 不想进入临界区，所以它给两个发送者都发回 OK 消息。进程 0 和进程 2 都发现了它们之间有冲突，然后都比较时间戳。进程 2 发现它失败了，所以它向进程 0 发送 OK 消息允许其进入。进程 0 现在把来自进程 2 的请求放入它的队列中，以便以后处理，然后就进入临界区，如图 5.14(b) 所示。当进程 0 退出临界区时，它将进程 2 的请求从队列中删除，并给进程 2 发送 OK 消息，允许进程 2 进入临界区，如图 5.14(c) 所示。该算法之所以能工作是因为它规定了在产生冲突的情况下，具有最短时间戳的进程获胜，并且每个进程都对时间戳的顺序达成了一致。

注意，如果进程 2 比进程 0 早发送消息，使得进程 0 在自己发送请求前收到了进程 2 的请求，并允许进程 2 进入临界区，那么这种情况和图 5.14 中所示的情况从根本上是不

同的。在这种情况下,进程 2 在接收到进程 0 的请求时,会发现自己已经在临界区内,它不给进程 0 发送应答,而是将该请求放入自己的队列中。

与上文中讨论的集中式算法一样,该算法实现的分布式互斥也不会发生死锁或饿死现象。每次进入临界区需要 $2(n-1)$ 个消息,这里的 n 是系统中的进程数目。这种算法的一个最大的优点是不存在单个故障点。

不幸的是,单个故障点被 n 个故障点所取代。如果任何一个进程崩溃,它就不能回答请求。这种不应答被错误地解释为拒绝请求,这样就阻塞了所有请求进入任何一个临界区的后续进程。因为 n 个进程之一发生故障的可能性至少是单一协调者发生故障的 n 倍,所以我们所做的只是设法用一种糟糕了 n 倍,并且要求更多网络通信的算法来代替前面那种集中式算法。

该算法的这个弱点可用我们前面提到的技巧来修补。即当请求到达时,接收者无论是允许还是拒绝都发送应答。一旦请求或应答消息丢失,发送者超时,它就继续发送请求消息,直到收到一个应答消息为止,或者直到发送者做出目的进程已崩溃的结论为止。在一个请求被拒绝后,发送者将进入阻塞状态,以等待 OK 消息。

该算法的另一个问题是,要么必须使用组通信原语,要么每个进程都必须自己维护组成员的清单,清单中包括进入组的进程、离开组的进程以及崩溃的进程。该方法最适用于进程数目较少并且从不改变组成员的情况。

最后,回顾集中式算法存在的一个问题,即处理所有请求时会产生瓶颈问题。在分布式算法中,所有进程要参与做出与进入临界区有关的所有决定。即使有一个进程不能承担这样的负载,那么强迫每个进程都做完全相同的事情也是不可能的。

可以对该算法进行各种小的改进。例如,为了进入临界区,要得到所有进程的允许其实并不必要,真正需要的是一种防止两个进程同时进入临界区的方法。算法可以改成当一个进程从大多数进程获得允许(而不需获得所有进程的允许)时,它就可以进入临界区。当然,在该算法的这种变种中,一个进程允许另一个进程进入临界区后,它在第一个进程释放许可之前是不能再允许其他进程进入同一临界区的。还有其他一些改进方法,例如 Maekawa(1985)提出的改进方法,但是这些方法很容易变得比较复杂。

但是,这种算法与原来的集中式算法相比更慢、更复杂、花费更高,而且更不健壮。既然如此,为什么还要研究该算法呢?原因之一是它说明了分布式算法至少也是可能实现的,这一点我们开始并不知道。另外,通过指出它的不足,可以鼓励未来的理论家们试着创造出更具实用性的算法。

5.5.3 令牌环算法

图 5.15 示意了在分布式系统中实现互斥的一种完全不同的方法。这里是一个总线式网络(例如以太网),如图 5.15(a)所示,进程没有固有的顺序。可以用软件的方法构造出一个逻辑环,环中为每个进程都分配了一个位置,如图 5.15(b)所示。环的位置可以按照网络地址或其他方式来分配。按照什么顺序进行排列并不重要。重要的是每个进程要知道谁在它的下一个位置上。

当环初始化时,进程 0 得到一个令牌(token)。该令牌绕着环运行,用点对点发送消

息的方式把它从进程 k 传到进程 $k+1$ (以环大小为模)。进程从它邻近的进程得到令牌后, 检查自己是否要进入临界区。如果自己要进入临界区, 那么它就进入临界区, 做它要做的工作, 然后离开临界区。在该进程退出临界区后, 它沿着环继续传递令牌。不允许使用同一个令牌进入另一个临界区。

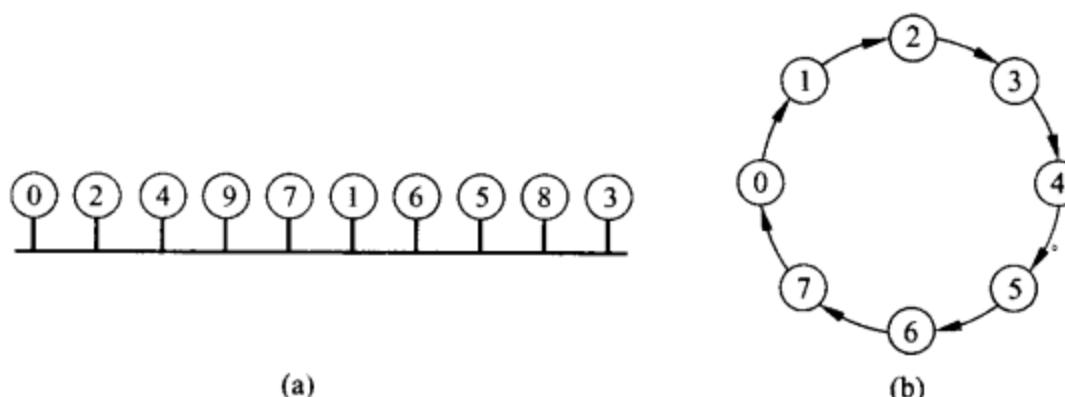


图 5.15 无序的进程组与构成逻辑环的进程组

(a) 网络中一组无序的进程; (b) 以软件方式构造的一个逻辑环

如果一个进程得到了邻近进程传来的令牌, 但是它并不想进入临界区, 那么它只是将令牌沿环往下传递。因而, 当没有进程想进入临界区时, 令牌就绕环高速传递。

该算法的正确性是显而易见的。在任何时刻都只有一个进程有令牌, 所以实际上只有一个进程能进入临界区。由于令牌以固定顺序在进程间循环传递, 所以不会发生饿死现象。如果一个进程想进入一个临界区, 那么最差的情况是等待其他所有进程都进入这个临界区然后再从中退出后它再进去。

该算法也照样存在问题。如果令牌丢失了, 那么它必须重新生成令牌。实际上, 检测令牌丢失是很困难的, 这是因为网络中令牌两次出现的时间间隔是不确定的。一小时没有发现令牌并不意味着它就丢失了, 也许某个进程还在使用它。

如果有进程崩溃, 该算法也会出现麻烦, 但是恢复起来却比其他算法容易。如果我们要求每个进程在收到令牌后发出确认信息, 那么当一个进程的邻近进程试图把令牌传递给它但是没有成功时, 这个崩溃的进程就会被检测到。这时就将这个崩溃的进程从组中删除, 令牌的持有者将令牌沿环传给这个崩溃进程的下一个进程, 如果必要, 传给再下一个。当然, 这样做需要每个进程都维护当前环的配置信息。

5.5.4 三个算法的比较

下面简略地比较一下我们已经讨论的三个互斥算法是很有意义的。在图 5.16 中我们列举这三个算法及它们的三个关键性质: 进程进入和退出一个临界区所需的消息数目, 进入临界区前的延迟(假设消息在网络中顺序传递), 以及每个算法存在的问题。

集中式算法最简单也最有效。使用这种算法, 进入并离开临界区一次只需要三个消息: 一个请求进入消息、一个允许进入消息和一个释放退出消息。分布式算法向其他每个进程发送一个请求进入消息, 这样需要 $n-1$ 个请求进入消息, 还有另外 $n-1$ 个允许进入消息, 总共有 $2(n-1)$ 个消息(假设只使用点到点通信通道)。令牌环算法的消息数目是可变的。如果每个进程都总想进入临界区, 那么令牌每传递一步就导致一次进出临界

区,这样平均每进一次临界区就需要一个消息。在另一种极端的情况下,有时令牌在环中绕行了几个小时也没有进程想进入临界区。这种情况下,每进入一次临界区需要的消息数目是不确定的。

算 法	每次进/出需要的消息数	进入前的延迟(按消息数)	问 题
集中式	3	2	协调者崩溃
分布 式	$2(n-1)$	$2(n-1)$	任何进程崩溃
令牌环	$1 \sim \infty$	$0 \sim n-1$	令牌丢失,进程崩溃

图 5.16 三种互斥算法的比较

对于三种算法来说,从进程想进入一个临界区到它真正进入临界区这段延迟时间也是不同的。当使用临界区的时间很短且很少使用时,延迟的主要因素是如何进入临界区的实际机制。当临界区使用时间较长且经常使用时,延迟的主要因素是等待其他进程使用临界区的时间。在表 5.16 中,我们说明的是前一种情况。假设消息是一个接一个地发送,那么在集中式算法的情况下每进入一次临界区需要两个消息,而在分布式算法的情况下需要 $2(n-1)$ 个消息。对于令牌环算法,所需的时间从 0(恰好接到令牌)到 $n-1$ (恰好释放令牌)个消息间变化。

最后,这三种算法在进程崩溃的情况下都损失惨重。为了避免进程崩溃造成的系统瘫痪,必须引入专门的措施和额外的复杂性。分布式算法甚至比集中式算法对进程的崩溃更敏感,这一点似乎具有讽刺意味。在容错系统中,这些算法都不适用,但是如果进程不经常崩溃,这些算法还是可以使用的。

5.6 分布式事务

与互斥紧密相关的一个概念是事务。互斥算法保证了某一时刻至多一个进程访问一个共享资源,如文件、打印机等。事务同样也保护一个共享资源不会被几个并发进程同时访问。事务尤其用于保护共享数据,然而它的功能远不止这些。特别是,事务允许进程把访问和修改数据项作为一项单独的原子操作来完成。如果进程在事务处理期间中途退出,所有数据都恢复到事务开始前的状态。本节详细研究事务的概念,并重点讨论事务对多个进程进行同步以保护共享数据的能力。

5.6.1 事务模型

事务的最初模型来自商业界。假设国际 Dingbat 公司需要一批装饰品。他们与一家潜在的供应商,即以装饰品的质量而闻名遐迩的 U. S. Widget 公司联系,希望后者 6 月份交付 100 000 件 10cm 的紫色装饰品。U. S. Widget 公司提出 12 月份交付 100 000 件 4 英寸紫红色装饰品。Dingbat 同意对方开出的价格,但是不喜欢紫红色,并希望 6 月份交货,并因为自己的客户是国际用户,所以坚持要 10cm 的产品。U. S. Widget 公司答复说 10 月份提供 $3\frac{15}{16}$ 英寸的淡紫色装饰品。经过进一步的谈判,双方最后同意 8 月 15 日交付 $3\frac{959}{1024}$ 英寸的紫罗兰色装饰品。

在此时之前，双方可以自由终止此讨论，无论谁终止，都会返回他们开始谈判前的状态。然而，一旦两个公司签订了合同，他们在法律上都有责任完成该合约。因此在双方还未签字之前，任何一方都可以停止，就像什么都没有发生一样，但是在他们签字的那一刻，他们通过了无法返回的那一点，交易就必须得到执行。

计算机模型与此相似。一个进程宣布它想和一个或多个其他进程开始一个事务。它们可以就不同的选项进行协商、创建并删除实体，以及执行一段时间的操作。然后发起者宣布它希望所有其他进程提交目前已做的工作。如果所有进程都同意，那么工作结果就成为永久性的。如果一个或更多进程拒绝（或者在同意前崩溃），那么形势就回到了事务开始前的状态，对文件或数据库等的副作用都会神奇地消失。这种要么全有、要么全无（all-or-nothing）的特性简化了编程人员的工作。

在计算机系统中使用事务可以追溯到 20 世纪 60 年代。在磁盘和联机数据库出现前，所有文件保存在磁带上。设想一个具有自动盘点系统的超级市场。每天关门后，计算机以两盘磁带作为输入进行处理。第一盘磁带存有当天早晨开门前的所有库存，第二盘存有当天的更新列表：已销售给顾客的产品和供应商交付的产品。计算机从两盘磁带上读取数据，并生成新的主库存磁带，如图 5.17 所示。

这种方案的好处（尽管使用它的人们可能并没有意识到）在于，对于任何原因引起的运行错误，所有的磁带都可以倒卷回去，工作可以毫无损失地重新开始。尽管原始，但是老的磁带系统具有事务的这种要么全有、要么全无的特性。

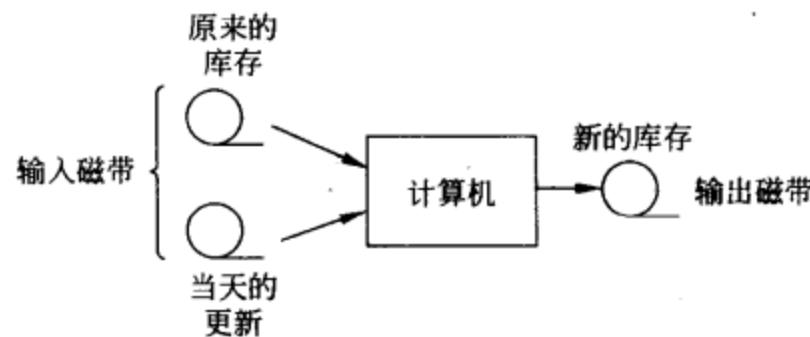


图 5.17 更新主磁带是具有容错功能的

现在来看一个现代银行应用系统，它适当地更新联机数据库。客户通过带有调制解调器的 PC 机连接到一个银行，想从一个账户取出钱，并将它存到另一个账户上。该操作通过下面两步完成：

- (1) 从账户 1 提取金额 a；
- (2) 向账户 2 存入金额 a。

如果电话连接在第一步后、第二步前中断，第一个账户已经记入借方，但是第二个账户却没有记入贷方，那么钱就消失在未知空间了。

将这两个操作组织在一个事务里将解决这个问题。要么两个操作都执行，要么两个操作都不执行。所以，关键在于如果事务执行失败，那么应能回退到初始状态。我们真正需要的是一个像使用磁带时那样的对数据库进行恢复的方法。事务必须提供这种能力。

使用事务进行编程需要一些专门的原语，这些原语或者由底层分布式系统提供，或者由语言的运行时系统提供。典型的事务原语的例子如图 5.18 所示。更准确的原语表取

取决于事务中正在使用的对象类型。在邮件系统中,可能有发送、接收和转发原语。在账目系统中,原语可能会有很大不同,然而一般都会有 READ 和 WRITE。事务中也允许使用普通语句、过程调用等。

原语	说明
BEG_IN_TRANSACTION	标志事务的开始
END_TRANSACTION	终止事务并试着提交
ABORT_TRANSACTION	销毁事务并恢复原来的值
READ	从文件、表或其他地方读数据
WRITE	向文件、表或其他地方写数据

图 5.18 事务原语的例子

BEGIN_TRANSACTION 和 END_TRANSACTION 用来限定事务的范围。介于它们之间的操作构成了事务体。所有这些操作要么全部执行,要么一个都不执行。这些操作可能是系统调用、库例程或者用某种语言编写的用括号括起来的语句块,这取决于实现的需要。

例如,让我们考虑在航空订票系统中,预订一张从纽约 White Plains 到肯尼亚 Malindi 的座票的进程。一个可能的路线是从 White Plains 到 JFK,再从 JFK 到 Nairobi,最后从 Nairobi 到 Malindi。在图 5.19(a)中,我们看到可以把预订这三个航班看成是 3 个不同的操作。

现在假设前两个航班已经预订成功,但是第三个航班已经订满了。这个事务被取消,并且前两个预订的结果也被撤销,航线数据库恢复到事务开始前的状态,请参见图 5.19(b),就好像什么也没有发生似的。

<pre>BEGIN_TRANSACTION reserve WP → JFK; reserve JFK → Nairobi; reserve Nairobi → Malindi; END_TRANSACTION</pre>	<pre>BEGIN_TRANSACTION reserve WP → JFK; reserve JFK → Nairobi; Nairobi → Malindi full ⇒ ABORT_TRANSACTION</pre>
(a)	(b)

图 5.19

(a) 预订三个航班的事务得到提交;(b) 当订不到第三个航班时,事务中止

要么全有、要么全无的特性是事务所具有的 4 个典型特性之一。更具体地说,事务的 4 个特性是:

- (1) 原子性(atomic): 对外界世界来说,事务的发生是不可分割的;
- (2) 一致性(consistent): 事务不能破坏系统的恒定性;
- (3) 独立性(isolated): 并发的事务不会互相干扰;
- (4) 持久性(durable): 一旦事务得到了提交,改变将是永远存在的。

这些属性通常按其首字母简称为 ACID。

所有事务表现的最关键的特性是它们的原子性(atomic)。该特性保证了每个事务要么全部发生,要么全部不发生。如果发生,事务是不可分割的瞬间动作。当一个事务正处

于执行中时,其他进程(无论它们是否与事务有关)看不到任何中间状态。

例如,假设某个事务开始向一个具有 10 字节长的文件添加数据。当该事务正在进行时,如果其他进程读该文件,无论事务已经添加了多少个字节,它们都只看到原来的 10 个字节。如果事务成功提交,此文件瞬间增加到事务提交时的新长度,无论这中间有多少个操作存在,都不会有中间状态。

第 2 个特性说明事务是一致的(consistent)。这意味着如果系统拥有某种必须总是保持的不变性,那么如果在事务开始之前保持这样的性质,则事务结束后该性质还应该存在。例如,在银行系统中,一个关键的不变性是资金守恒规则。任何内部转账后,银行的资金必须与转账前的数目一致。在事务执行的短暂时刻内,可能损害这个不变性。然而,在事务的外部是看不到这种损害的。

第 3 个特性说明了事务是独立的(isolated)或者串行的(serializable)。这意味着如果两个或者更多的事务同时执行,对于它们中的每个事务和其他进程,最后的结果看起来好像所有的事务以某种依赖于系统的次序顺序执行一样。下面我们还会回来讨论串行性。

第 4 个特性说明事务是持久的(durable)。它是指这样的事实,即一旦一个事务提交,无论发生什么事情,这个事务都会向前进行,其结果都会变成永久性的。事务提交之后的任何故障都不可能使结果取消或者丢失。第 7 章将详细讨论持久性。

5.6.2 事务的分类

到目前为止,我们基本上把事务看成是满足 ACID 特性的一系列操作。这种类型的事务也称为单层事务(flat transaction)。单层事务是最简单的,也是最常用的。然而,单层事务有许多局限性,因而产生了一些其他模型。下面我们将讨论两类重要的事务:嵌套事务和分布式事务。其他类型的事实在(Gray 和 Reuter 1993)中有广泛讨论。

1. 单层事务的一些局限

单层事务的主要局限是它们不允许提交或取消部分结果。换句话讲,单层事务很强的原子性在某种程度上也是它的缺点。

我们再次考虑一下预订从纽约到肯尼亚的航班,如图 5.19 所示。假设一次购买全程票较便宜,为此这三个部分被组织在一个单独事务里。当我们发现只有最后一部分订不上时,我们可能决定仍然要订前两部分。例如,我们可能也发现很难预订从 JFK 到 Nairobi 的航班。取消整个事务将意味着我们将不得不再次预订那个航班,到那时有可能会导致预订失败。因此,在这种情况下我们需要部分地提交事务。单层事务是不允许这样的。

作为另外一个例子,我们考虑一下在 Web 站点中把一个超级链接作为一个双向引用实现。换句话说,如果一个 Web 页 W_1 包含一个到页 W_2 的 URL,并且 W_2 知道 W_1 引用它(例如,请参见文献(Kappe 1999))。现在假设页 W 被移到另外的位置或者被其他的页代替。在这种情况下,所有连接到 W 的超级链接都应该得到更新,并且最好在一个原子操作中完成,否则将会暂时出现对 W 的引用不一致的现象。理论上,单层事务可以用于此处。该事务包括更新 W 和一系列的操作,每一个操作更新一个含有连接到 W 的超级链接的 Web 页。

然而,问题是像这样的事务可能要花几个小时才能完成。不但引用 W 的页分散在 Internet 上,而且还可能有成千上万的页面需要更新。把每一个更新作为一个独立的事务也不好,因为这样会出现一些 Web 页可能有正确的链接,而其他页可能没有的情况。本例的一种可能的解决方法是提交更新,同时也为那些还没有更新链接的页面保留原有的 W。

2. 嵌套事务

上面提到的局限性可以通过使用嵌套事务(nested transaction)来解决。一个嵌套事务由许多子事务构成。可以使顶层事务分支为在不同的机器上并行运行的子事务,以提高性能或简化编程。这些子事务中的任何一个都可以执行一个或多个子事务,或者经过分支拥有自己的子事务。

子事务引发了一个微小但很重要的问题。设想一个事务启动了一些并行子事务,其中的一个子事务进行了提交,使它的结果对父事务是可见的。经过进一步计算之后,父事务中止了,需要把整个系统恢复到顶层事务开始前的状态。因此,子事务提交的结果必须被撤销。这样上面所提到的事务的持久性只应用于顶层事务。

由于事务可以嵌套任意多层,因此需要相当多的管理工作来保证一切正常运行。然而,语义是清晰的。当任何事务或子事务开始时,概念上它得到了整个系统全部数据的一份私有副本,它可以按照自己的意愿对该副本进行操作。如果中止了事务,那么它的私有空间就会消失,好像事务从来没有发生过似的。这样,如果一个子事务提交后接着又开始了一个新的子事务,那么第二个事务可以看到前一个事务产生的结果。同样,一个外部(高层)事务中止,它下面的所有子事务也不得不中止。

3. 分布式事务

嵌套事务在分布式系统中是很重要的,因为它们提供了一种将事务分布到多个机器上的很自然的方法。然而,嵌套事务通常根据原来事务的工作来逻辑地划分事务。例如,如图 5.19 所示的预订三个不同航班的事务可以在逻辑上分成三个子事务。这些子事务中的任何一个都可以单独进行管理,并且独立于其他两个子事务。

然而,嵌套事务在按照逻辑关系划分成子事务时并不考虑所有的分布因素。例如,处理预订从纽约到 Nairobi 的座位的子事务可能仍然需要存取两个数据库,这两个数据库分别位于两个城市中。在这种情况下,子事务不再划分成更小的子事务了,这是因为在逻辑上已经没有子事务存在,预订本身是一个不可分割的操作。

本例中,我们遇到的是一个单层子事务需要处理分布在多台机器上的数据的情况。这样的事务称为分布式事务(distributed transaction)。嵌套事务和分布式事务之间的差别很微小,但却很重要。嵌套事务是一个按照逻辑关系分解成一层层子事务的事务。相比之下,分布式事务逻辑上是一个单层的、不可分割的事务,它的操作对象是分布式的数据。图 5.20 揭示了这个差别。

分布式事务的主要问题是需要用分布式算法来锁定数据和提交整个事务。下面将讨论分布式锁定问题。我们将在第 7 章详细介绍分布式提交协议,因为提交协议与第 7 章讨论的容错和恢复机制有关。

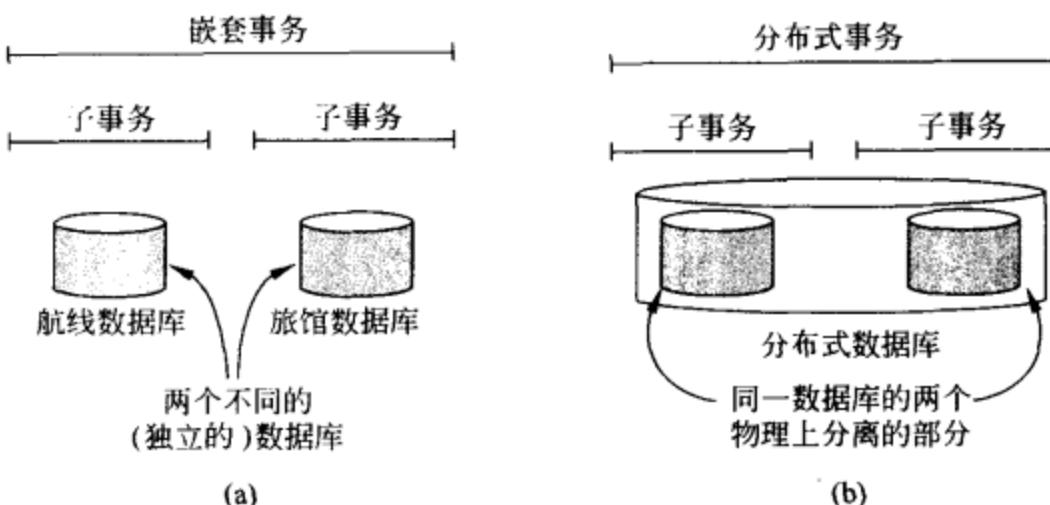


图 5.20 嵌套事务与分布式事务

(a) 嵌套事务; (b) 分布式事务

5.6.3 实现

事务听起来是个好主意,但是如何实现它们呢?我们本节将解决这个问题。为了简化问题,我们考虑一个文件系统的事务。如果执行事务的进程只是在适当位置更新它使用的文件,那么事务将不具有原子性,并且当事务被中止时,改变将不会神奇地消失。很显然,需要某种其他的实现方法。通常采用两种方法,下面将依次进行讨论。

1. 私有工作空间

在概念上,当一个进程开始一个事务时,它被分配一个私有工作空间,该工作空间包含所有它有权访问的文件。在事务提交或中止前,它的所有读写操作都在私有工作空间内进行,而不是直接对文件系统进行操作。这就直接导致了第一种实现方法的产生;在进程开始一个事务的时刻,实际上为该进程分配了一个私有工作空间。

此技术带来的问题是把所有的东西都复制到私有工作空间的开销是十分大的,但是各种各样的优化使这种方法可行。第一种优化方法是基于这样的认识,即当一个进程只读取一个文件而不对它作修改时,就不需要私有拷贝。该进程可以直接使用真正的文件(除非自事务开始以来文件已被改动)。因此,当进程开始一个事务时,就为它创建一个私有的工作空间,该空间是空的,除非当一个指针回指到它的父辈工作空间,这样就足够了。当事务处于顶层时,父辈工作空间就是文件系统。当进程为了读取而打开文件时,指针将回指,直到可以在父辈(或者更老的祖先)工作空间中找到文件为止。

当打开一个文件进行写入时,它可用同读取时一样的方法对文件进行定位,除非它是第一次被复制到私有工作空间。然而,第二种优化方法可以大大减少复制工作量。因为它不是复制整个文件,而是只将索引复制到私有工作空间。索引是与判断文件所在磁盘块位置有关的数据块。在 UNIX 中,索引是 i 节点。通过私有索引,文件可以按通常方式读取,这是因为索引中包含的地址是原始的磁盘块。然而,当一个文件块第一次被修改时,将生成该块的副本,其地址也被插入索引中,如图 5.21 所示。然后就可以在不影响原始块的情况下更新这个块。添加块也是用这种方法解决。新块有时被称为影像块(shadow block)。

如图 5.21(b)所示,运行事务的进程看到了修改的文件,但是其他所有进程看到的仍然是原始的文件。在更复杂的事务中,私有工作空间可能包含大量的文件而不仅仅是一个。如果事务中止,私用工作空间就被简单地删除,它所指向的所有私有块也将被释放回自由列表(free list)中。如果事务提交了,那么私有索引被移到父辈工作空间中,如图 5.21(c)所示。不能再被获取的块被放到自由列表中。

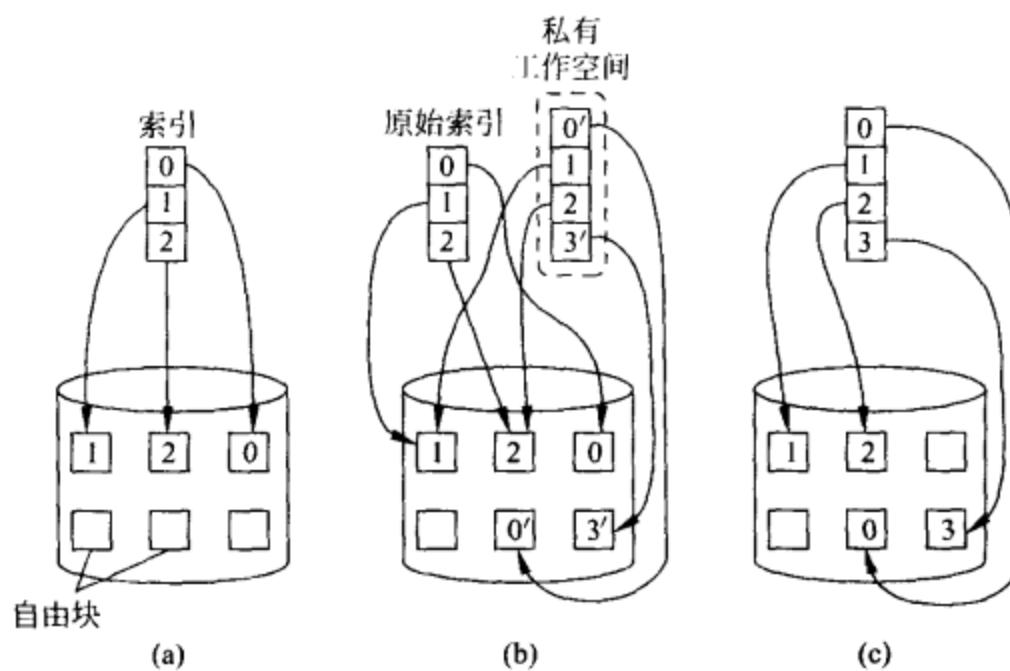


图 5.21

(a) 有三个数据块的文件的文件索引和磁盘数据块; (b) 一个事务修改了第 0 块和向第 3 块增加了数据后的情况; (c) 提交以后的情况

这种方法也可用于分布式事务。在这种情况下,进程在每台包含事务要存取的文件的机器上启动。每个进程都被分配了如上所描述的私有工作空间。如果事务中止,所有进程简单地丢弃它们的私有工作空间。另一方面,当事务提交时,更新在本地发生,此时事务作为整体而完成。

2. 写前日志

另一个实现事务的常用方法是写前日志(writeahead log)。使用这种方法时,文件将真正被修改,但是在任何一个数据块被修改前,一条记录被写到了日志中以说明哪个事务正在对文件进行修改,哪个文件和哪个数据块被改动了,旧值和新值各是什么。只有当日志被成功写入后,此改动才可以被写入文件。

图 5.22 给出了一个日志如何工作的例子。在图 5.22(a)中,我们有一个使用两个初值均是 0 的共享变量(或者别的对象) x 和 y 的简单事务。对于事务里三个语句中的任何一个,在执行该语句前都要写入一条日志记录,记录旧值和新值。在图 5.22(b)~(d)中,这些值使用斜线来区分开。

如果事务执行成功并被提交,那么一条提交记录被写进日志,但是数据结构不需要变动,这是因为它们已经被更新了。如果事务中止,那么可以使用日志来回退到原来的状态。从日志的末尾开始向前读取每条记录,同时将在每条记录中描述的改动撤销。这个

动作成为回退(rollback)。

```
x=0;  
y=0;          Log      Log      Log  
BEGIN_TRANSACTION;  
x=x+1;        [x=0/1]  [x=0/1]  [x=0/1]  
y=y+2;        [y=0/2]  [y=0/2]  [y=0/2]  
x=y*y;        [x=1/4]  [x=1/4]  [x=1/4]  
END_TRANSACTION;
```

(a) (b) (c) (d)

图 5.22

(a) 事务; (b)~(d) 每条语句执行前的日志

这种方法也适用于分布式事务。在分布式事务的情况下,每台机器都保持关于本地文件系统改变的日志。在事务中止时进行的回退要求每台机器都独立地回退以恢复原有的文件。

5.6.4 并发控制

到目前为止,我们已经解释了实现事务原子性的实质。在存在故障的情况下,实现原子性和持久性是一个重要的话题,我们将在第 7 章讨论,这是因为它不仅仅跟事务有关。通过正确地控制并发事务的执行基本上可以解决一致性和独立性问题,并发事务也就是同时对共享数据进行操作的事务。

并发控制的目的是允许几个事务同时执行,但是被操作的数据项集合(例如,文件或者数据库记录)要保持一致的状态。通过允许事务以某一特定的顺序存取数据项,藉此使最后得到的结果跟事务顺序执行的一样,从而实现这种一致性。

根据如图 5.23 所示的以分层方式组织的三个不同管理器,我们可以很好地理解并发控制。最底层由数据管理器(data manager)组成,它完成对数据的读写操作。数据管理器不关心哪个事务正在做读或写操作。实际上,它对事务一无所知。

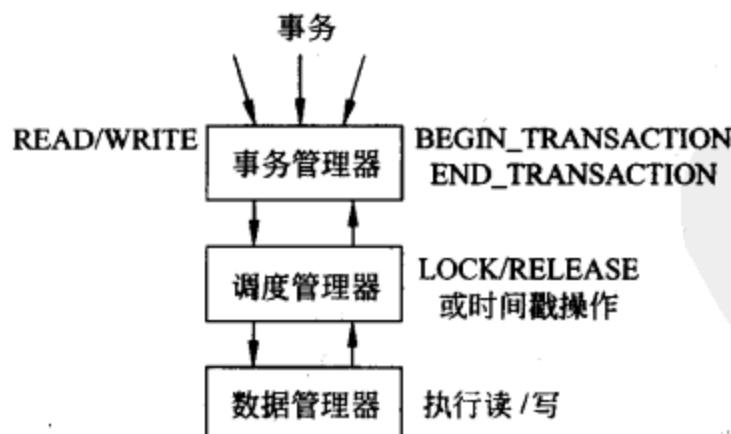


图 5.23 处理事务的管理器组织

中间层由一个调度管理器(scheduler)组成,它的主要任务是正确地控制并发。它决定哪个事务,在何时被允许将读或写操作传给数据管理器。它是通过以满足事务的独立

性和一致性的方式调度单个的读写操作实现的。下面我们将讨论基于时钟的调度算法和基于时间戳的调度算法。

最高层由事务管理器组成,该层主要责任是保证事务的原子性。它通过将事务原语转化成调度器的调度请求来处理事务原语。

图 5.23 所示的模型可以用于图 5.24 所示的分布式情况下。每个站点都有它自己的调度管理器和数据管理器,它们一起来保证本地数据的一致性。每个事务由一个事务处理器来处理。事务处理器与个人站点的调度管理器通信。根据并发控制算法,调度管理器也可以与远程数据管理器通信。我们下面回到分布式并发控制的讨论。

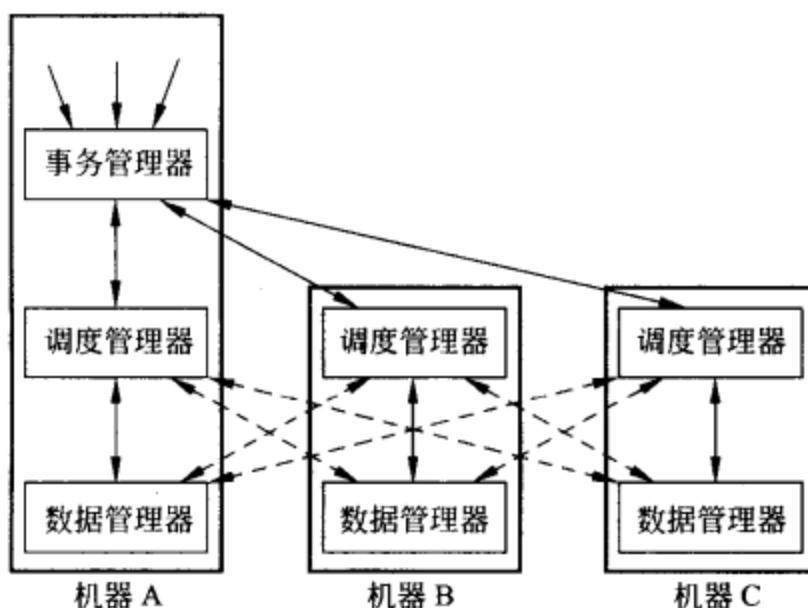


图 5.24 分布式事务管理器组织

1. 串行性

并发控制算法的主要目的是保证多个事务可以同时被执行并且仍然保持独立。这意味着最终的执行结果跟事务以某种特定顺序一个接一个串行执行得到的结果相同。

在图 5.25(a)~(c) 中,我们有三个事务由三个分离的进程同时执行。如果它们顺序执行,最终 x 的值依赖于哪个事务最后执行(因为 x 可能是一个共享变量、文件或者某种实体),其值最终将为 1、2 或 3。在图 5.25(d) 中,我们看到了各种各样的顺序,称为调度(schedule)。它们可能被交错执行。调度 1 实际上是串行进行的。换句话说,事务严格地顺序执行,所以它满足定义的串行条件。调度 2 没有串行进行,但是它仍然是合法的,因为它得到的 x 值在事务严格串行执行时是可以得到的。第三个调度是不合法的,因为它将 x 的值设为 5,而该值是事务执行任何顺序都得不到的。各个操作正确地交错执行是由系统来保证的。让系统自由选择它想要的操作顺序,并假设该顺序得到的答案是正确的,我们就为程序员消除了自己必须完成互斥工作的要求,从而简化了编程工作。

为了理解调度和并发控制,必须确切地知道正在执行什么。换句话讲, x 的值被加 2 还是加 3 并不重要,重要的是 x 的值正在被改变。因此,我们可以用一系列对具体数据项的读写操作来表示事务。例如,图 5.25(a)~(c) 中 T_1 、 T_2 和 T_3 三个事务都各自被表示成这样的序列:

$\text{write}(T_i, x); \text{read}(T_i, x); \text{write}(T_j, x)$

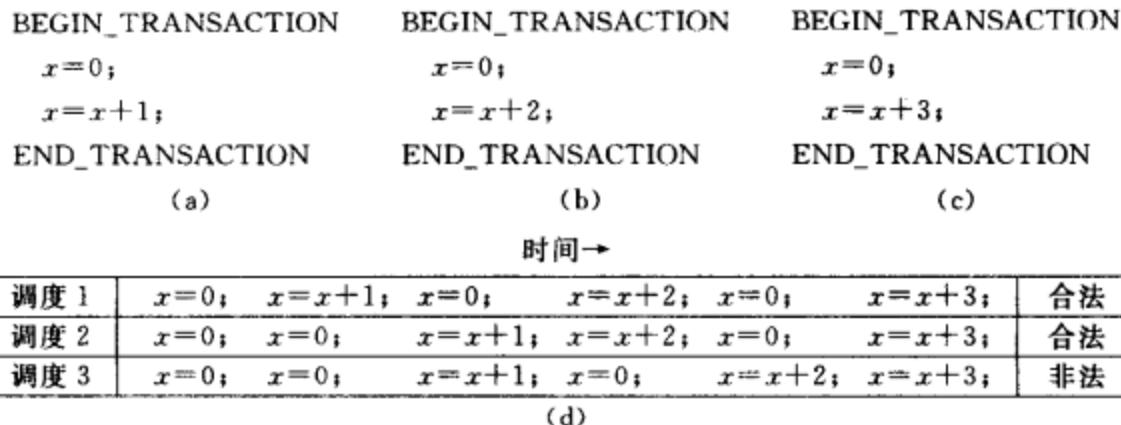


图 5.25

(a)~(c) 3 个 T_1, T_2 和 T_3 ; (d) 可能的调度

并发控制的总体思想是正确地调度相冲突的操作(conflicting operations)。如果两个操作都是对同一个数据项进行操作，并且至少一个是写操作，那么这两个操作就互相冲突。在一个读写冲突(read-write conflict)中只能有一个操作是写操作，否则，我们就要处理写写冲突(write-write conflict)。冲突操作来自同一个事务还是不同的事务并不重要。两个读操作不会发生冲突。

并发控制算法通常根据读写操作同步的方式来分类。同步可以通过共享数据上的互斥机制(例如, 锁定)，或者通过显式地使用时间戳排序来实现。

并发控制算法可以进一步区分为悲观算法和乐观算法。悲观方法(pessimistic approaches)的基本原则是 Murphy 定律：如果某事物可以出错，那么它就会出错。在悲观方法中，操作是在它们被执行前同步的，这意味着冲突在允许发生之前就解决了。相反，乐观方法(optimistic approaches)是基于错误一般不会发生的观点。所以操作被简单地执行，在事务结束的时候再进行同步。如果那时确实发生了冲突，一个或更多的事务将被迫中止。下面，我们将讨论两个悲观方法和一个乐观方法。文献(Bernstein 和 Goodman 1981)中给出了有关这些方法的各种机制的概括。

2. 两阶段锁定

最古老也是最广泛使用的并发控制算法是锁定(locking)。在这种最简单的方式中，当一个进程要作为事务的一部分来读或写一个数据项时，它请求调度管理器允许它给该数据项加锁。同样，当它不再需要一个数据项时，就请求调度管理器释放该锁。调度管理器的任务是以一种可以得到正确调度结果的方式来允许加锁和释放锁。换句话说，需要使用一种可以提供串行调度的算法。这样一个算法是两阶段锁定算法。

在图 5.26 所示的两阶段锁定(two-phase locking, 2PL)中，调度管理器先在增长阶段(growing phase)获得它所需的所有锁，然后在收缩阶段(shrinking phase)释放它们。尤其要遵守下面三个规则，这三个规则在(Bernstein 等 1987)中被更明确地解释。

(1) 当调度管理器收到来自事务管理器的 $\text{oper}(T, x)$ 操作时，它检测该操作是否跟它已经允许锁定的另一个进程冲突。如果存在冲突，操作 $\text{oper}(T, x)$ 被延迟(这样事务 T 也被延迟)。如果没有冲突，调度管理器允许对数据项 x 加锁，并将这个操作传递给数据

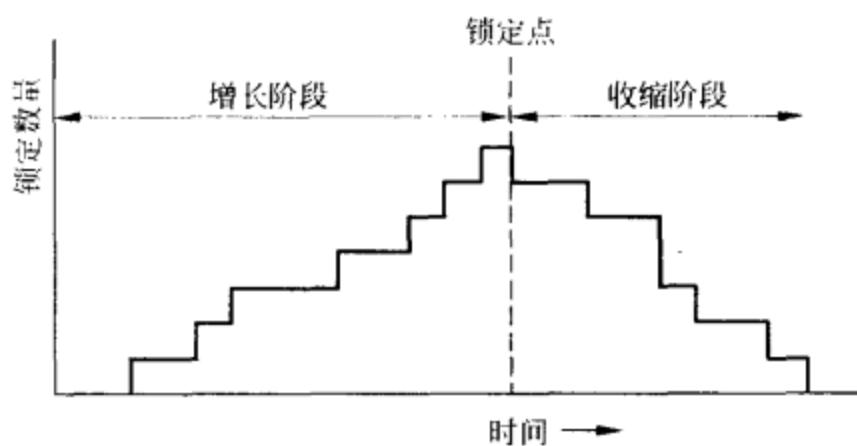


图 5.26 两阶段锁定

管理器。

(2) 直到数据管理器通知它已经完成了对锁定数据项 x 的操作, 调度管理器才会释放数据项 x 的锁。

(3) 一旦调度管理器为事务 T 释放了锁, 那么无论事务 T 请求为哪个数据项加锁, 调度管理器都不会允许 T 加另外一把锁。 T 获取另外一把锁的任何企图都是一个程序错误, 都会中止事务 T 。

可以证明(Eswaran 等 1976)如果所有事务都使用两阶段锁定, 那么通过交错执行事务所形成的调度都是串行的。这也是两阶段锁定被广泛使用的原因。

在许多系统中, 收缩阶段是在事务结束运行之后要么提交要么中止时出现, 它导致锁的释放, 如图 5.27 所示。这种策略称为严格的两阶段锁定(strict two-phase locking), 它有两个主要优点。第一, 事务总是读提交事务写入的值, 所以我们从来都不会因为事务的计算是基于一个它不应该看到的数据项而中止它。第二, 所有锁的获得和释放都可以由系统来处理而无须事务关心。要访问一个数据项就要获得锁, 当事务完成时就释放锁。这种策略消除了瀑布型中止(cascaded aborts), 即不得不撤销一个已经提交的事务, 因为它看到了它不该看见的数据项。

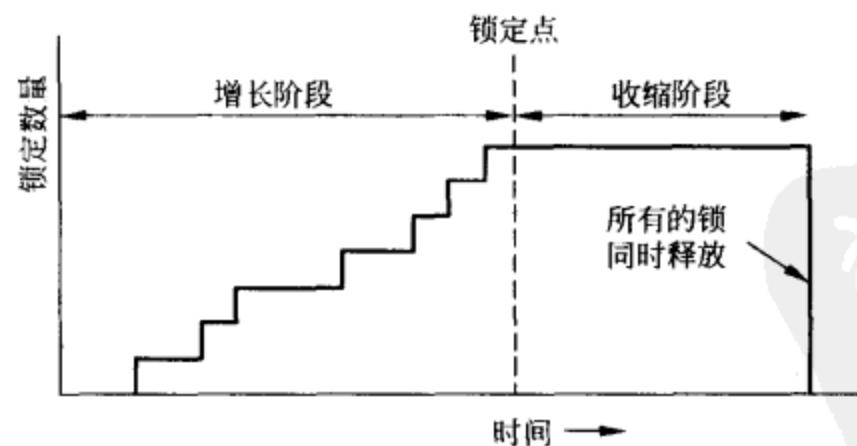


图 5.27 严格的两阶段锁定

两阶段锁定和严格的两阶段锁定都可能导致死锁。如果两个进程都试图获得同一对锁, 但是它们是以相反的顺序请求, 那么就可能导致死锁。这可采用一般性的技术来解决这一个问题, 比如以某种规范的顺序获得所有的锁以免出现保持—等待循环。也可以采

用死锁监测，即通过维护一张清楚地描述哪个进程有哪个锁和它还需要什么锁的图，并且检查该图是否有循环来检测是否存在死锁。最后，如果事先知道一个锁的拥有时间不会超过 t 秒，那么也可以使用定时方法，即如果一个锁持续在同一个拥有者的时间超过 t 秒，那么一定是出现了死锁。

有几种方法可以在分布式系统中实现基本的两阶段锁定方法。假设被操作的数据分布在多台机器上。在集中式 2PL(centralized 2PL)中，由一个站点负责允许加锁和释放锁。每个事务管理器都与这个集中的锁管理器通信获得加锁授权。当允许加锁时，该事务管理器随后就直接与数据管理器通信。注意，在这种方法中，数据项也可以被复制到多台机器上。当操作完成时，事务管理器将锁归还给锁管理器。

在基本的 2PL(primary 2PL)中，每个数据项被分配了一个主要的拷贝。拷贝所在机器上的锁管理器负责允许加锁和释放锁。主要 2PL 除了加锁是分布在多台机器上以外，它本质上跟集中式 2PL 相同。

在分布式 2PL(distributed 2PL)中，假设数据可以复制到多台机器上。与主要的 2PL 和集中式 2PL 相比，分布式 2PL 的特点是，每台机器上的调度管理器不但负责授权加锁和释放锁，也负责将操作转发给本地数据管理器。在这种意义上，分布式 2PL 更接近基本的 2PL 方案，但是分布式 2PL 是在数据所在的每个站点上执行的。

通常数据库系统和并发控制的两阶段锁定的典型处理方法可以在文献(Bernstein 等 1987)中找到。

3. 悲观的时间戳排序

一个完全不同的并发控制方法是在每个事务 T 开始时给它分配一个时间戳 $ts(T)$ 。使用 Lamport 算法，我们可以保证时间戳是惟一的，这一点很重要。事务 T 的每个操作都被盖上时间戳 $ts(T)$ ，并且系统中的每个数据项 x 都有一个相关的读时间戳 $ts_{RD}(x)$ 和写时间戳 $ts_{WR}(x)$ 。读时间戳被设置为最近读 x 的事务的时间戳，而写时间戳是最近修改 x 的事务的时间戳。使用时间戳排序，如果两个操作冲突，则数据管理器先处理时间戳最早的操作。

现在假设调度管理器从具有时间戳 ts 的事务 T 收到一个操作 $read(T, x)$ 。但是 $ts < ts_{WR}(x)$ 。换句话说，调度管理器发现一个对 x 的写操作在事务开始后已经完成。在这种情况下，事务 T 简单地被中止。相反，如果 $ts > ts_{WR}(x)$ ，那么让读操作发生。另外， $ts_{RD}(x)$ 被设置成 $\max\{ts, ts_{RD}(x)\}$ 。

同样，假设调度管理器收到一个具有时间戳 ts 并包含写操作 $write(T, x)$ 的事务 T 。如果 $ts < ts_{RD}(x)$ ，那么它只能取消事务 T ，这是因为 x 的当前值已经被更晚的事务读过。事务 T 太晚了。另一方面，如果 $ts > ts_{RD}(x)$ ，那么它改变 x 的值，因为没有更晚的事务读过它。 $ts_{WR}(x)$ 也被设置为 $\max\{ts, ts_{WR}(x)\}$ 。

为了更好地理解时间戳排序，我们考虑下面的例子。假设有三个事务 T_1 、 T_2 和 T_3 。 T_1 运行了很长时间了，并且已经使用了所有 T_2 和 T_3 ，所需要的数据项，所以所有数据项将读和写时间戳都设置为 $ts(T_1)$ 。事务 T_2 和 T_3 并发开始，并且 $ts(T_2) < ts(T_3)$ 。

我们先考虑事务 T_2 正在写数据项 x 。除非 T_3 已经潜入并已经提交，否则 $ts_{RD}(x)$ 和

$ts_{WR}(x)$ 将被设置为 $ts(T_1)$, 比 $ts(T_2)$ 小。在图 5.28(a) 和(b) 中, 我们看到 $ts(T_2)$ 比 $ts_{RD}(x)$ 和 $ts_{WR}(x)$ 都大 (T_3 还没有提交), 所以写操作被接受并被暂时执行。当 T_2 提交时, 这个操作结果将变成持久性的。 T_2 的时间戳现在作为暂时的写操作被记录在数据项中。

在图 5.28(c) 和(d) 中, T_2 是不幸的。 T_3 已经读了 $x(c)$ 或者已经改写了 $x(d)$ 并且已经提交, 因此 T_2 事务被中止。 T_2 事务仍然可以请求一个新的时间戳并再次开始。

现在看看读操作。在图 5.28(e) 中, 如果没有冲突, 可以立即读。在图 5.28(f) 中, 某个闯入者已经进入并试图改写 x 。这个闯入者的时间戳比 T_2 的小, 所以 T_2 简单地等待直到闯入者提交, 此时它便可以读新的文件并继续执行。

在图 5.28(g) 中, T_3 改变了 x 并已经提交了, T_2 必须再次中止。在图 5.28(h) 中, T_3 在修改 x 的过程中, 虽然它还没有提交, 但是 T_2 仍然太晚了, 因此必须中止。

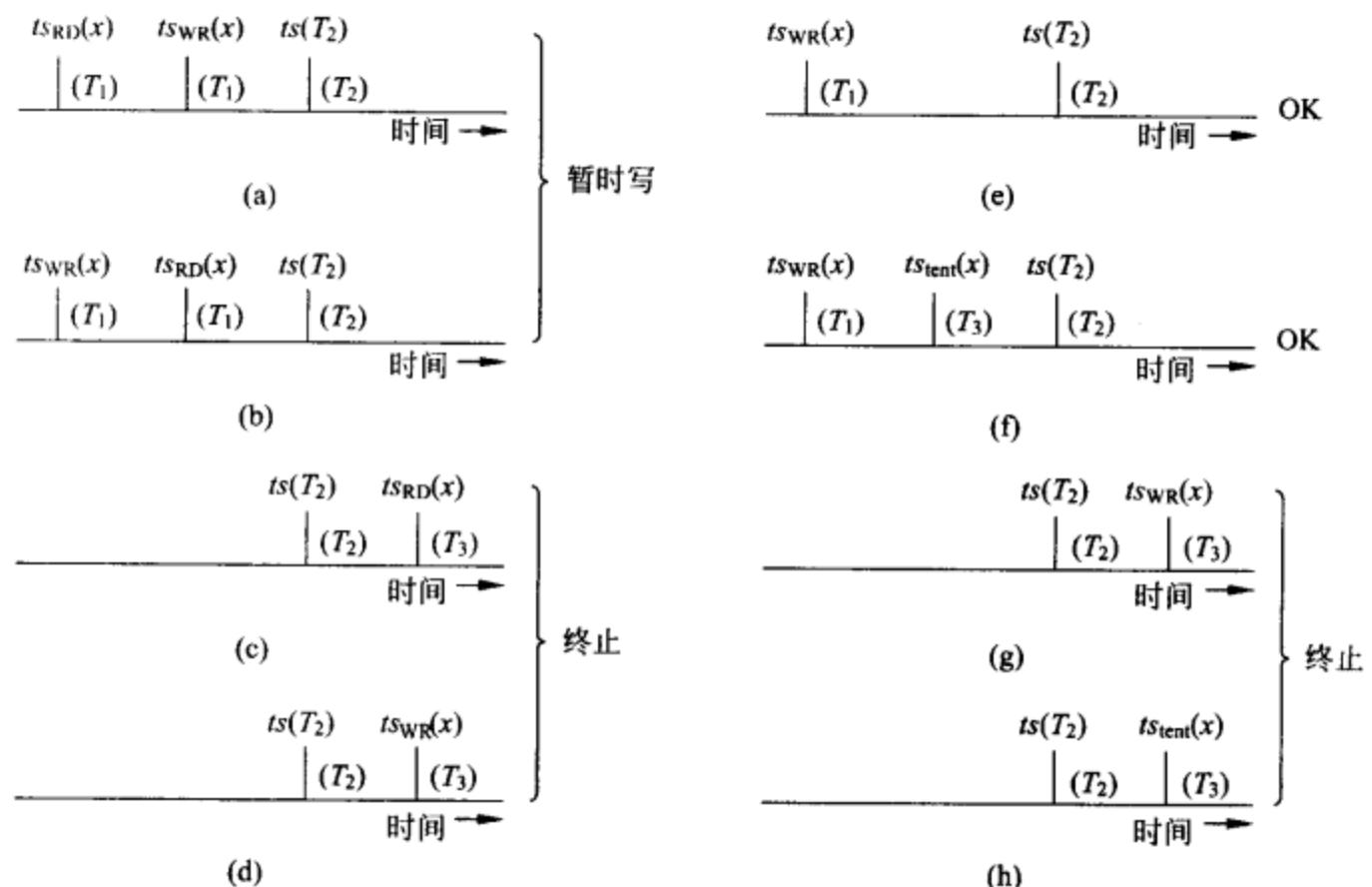


图 5.28 使用时间戳的并发控制

时间戳具有与锁定不同的特性。当事务遇到一个更大(更晚)的时间戳时, 它中止, 而在同样的情况下, 如果使用锁定方法, 事务将等待或立即继续执行。另一方面, 它不会造成死锁, 这是一个很大的优点。

基本的时间戳排序有几种, 典型的有保守的时间戳排序和多版本时间戳排序。详细的介绍可以在文献(Gray 和 Reuter 1993; Oszu 和 Valduriez 1999)中找到。

4. 乐观的时间戳排序

第三种处理多个事务同时执行的方法是乐观的并发控制(optimistic concurrency) (Kung 和 Robinson 1981)。此技术的思想是相当地简单: 不关心别人在干什么, 继续做自己要做的事情。如果有问题, 等到后面再考虑(许多政客也使用这个算法)。实际上, 相

对来讲冲突是很少的,所以系统大部分时间都运行正常。

尽管冲突可能很少发生,但是也并不是不可能发生的,所以需要某种方法来处理冲突。乐观的并发控制所做的事情是跟踪哪些数据项被读写了。在某个事务提交的时候,它检查其他所有事务,看看是否这些数据项中的某些从这个事务开始后已经被改变了。如果被改变了,该事务被中止。如果没有被改变,该事务就被提交。

乐观的并发控制算法最适合基于私有工作空间的实现。在这种方式下,每个事务私下改变自己的数据,不受别的事务的干涉。最终,新数据要么被提交要么被释放,这是一个相对简单直接的方法。

乐观的并发控制的最大优点是它不会发生死锁,允许最大的并行性。这是因为没有进程需要等待一个锁。缺点是有时它可能失败,那时事务将不得不再次执行。在负载较重的情况下,失败的可能性可能增大,使得乐观的并发控制成为了一个较差的选择。

像(Oszu 和 Valduriez 1999)指出的一样,对乐观的并发控制的研究主要集中在非分布式系统中。另外,在商业的或原型的系统中,它都很难实现,所以很难与我们讨论过的其他方法进行对比评价。

5.7 小结

分布式系统中的进程同步问题与进程间通信有密切的关系。同步是在适当的时刻做恰当的事情。分布式系统和计算机网络中的一个普遍问题是没有全局共享时钟。换句话说,不同机器上的进程都有自己的时间。

分布式系统中有很多同步时钟的方法,但是所有方法本质上都是基于交换时钟值,同时关注发送和接受消息时所采用的时间。通信延迟的变化和对这些变化处理的方式很大程度决定了时钟同步算法的准确性。

许多情况下是不需要知道绝对时间的。重要的是不同进程中的相关事件以正确的顺序发生。Lamport 算法说明通过引入逻辑时钟的概念,可以将进程集合起来,让这些进程就事件的正确顺序达成全局的一致。本质上,每个事件 e ,例如发送或者接收消息,都被分配给一个全局唯一的逻辑时间戳 $C(e)$,目的是当事件 a 在事件 b 之前发生时, $C(a) < C(b)$ 。Lamport 时间戳可以被扩展成向量时间戳:如果 $C(a) < C(b)$,那么我们知道事件 a 在因果关系上发生在 b 之前。

因为分布式系统中没有共享存储器的概念,所以常常很难准确地确定系统当前的状态是什么。可以通过让所有的进程都同步来确定一个分布式的全局状态。这样,每个进程都收集自己的本地状态,将该状态与消息一起传输。同步不需要强迫进程停下来收集它们的状态就可以完成。相反,所谓的分布式快照可以在分布式系统继续运转时收集。

进程间的同步常常要求一个进程扮演协调者。在这些情况下,协调者不是固定的,需要分布式计算来决定谁将成为协调者。可通过选举算法来做这样的决定。选举算法主要用于协调者可能崩溃的情况。

一类重要的同步算法是分布式互斥。这些算法确保分布式进程每次至多一个进程可

以访问共享资源。如果我们使用一个协调者来跟踪应该轮到谁来访问时,分布式互斥就可以容易地实现。也有完全的分布式算法,但是它们有缺点,它们更容易使通信和进程发生故障。

与互斥相关的是分布式事务。一个事务包括一系列对数据的操作,所以事务要么完全执行,要么全不执行。另外,许多事务可以同时执行,结果,其整体效果就好像事务以某种任意的但却是串行的顺序被执行。最后,事务是持久的,这意味着如果它完成了,它的影响就是持久的。

习 题

1. 说出至少三种可在 WWV 广播时间和在分布式系统中处理机设置内部时钟之间引入的延迟源。
2. 考虑分布式系统中的两台机器的行为。这两台机器的时钟假设每毫秒滴答 1000 次。但实际上只有一台是这样的,另一台每毫秒滴答 990 次,如果 UTC 每分钟更新一次,那么时钟的最大偏移量是多大?
3. 向图 5.7 中加入一个与消息 A 并发的新消息,即它既不发生在 A 的前面也不发生在 A 的后面。
4. 要使用 Lamport 时间戳实现全序多播,是不是每个消息都必须要被严格地确认?
5. 考虑一个消息只是以它们被发送的顺序被传递的通信层。给出一个不需要这种限制的例子。
6. 假设两个进程同时检测到协调者崩溃了,并且它们都使用欺负算法主持一个选举。这时将发生什么。
7. 在图 5.12 中,我们有两个 ELECTION 消息同时在循环。当对它们都没有什么不利影响时,杀掉一个将更好。设计这样一个算法,该算法在不影响基本的选举操作算法的情况下完成此项任务。
8. 许多分布式算法需要使用协调进程。讨论一下,这样的算法实际上可以在什么程度上被看作为分布式的?
9. 在集中式互斥方法中(图 5.13),协调者在收到一个进程释放它独占访问的临界区的消息后,通常给等待队列中的第一个进程授权,允许其访问临界区。请给出协调者另一个可能的算法。
10. 请再考虑图 5.13。假设协调者崩溃了。这会必然导致整个系统瘫痪吗?如果不会,那么在什么情况下会呢?有什么方法避免这个问题的发生,使系统能够忍受协调者的崩溃?
11. Ricart 和 Agrawala 算法会有这样的问题:如果一个进程崩溃,并且没有对另一个要求进入临界区的进程的请求作应答,没有应答意味着拒绝请求。我们建议所有的请求应当立即被应答,以便很容易地检测到崩溃的进程。是否存在一些情况,即使使用这种方法也还不够?请讨论。
12. 如果我们假设该算法可以在支持硬件广播的 LAN 中实现,那么图 5.16 中的实

体将如何改变？

13. 分布式系统可能有多个互相独立的临界区。假设进程 0 想进入临界区 A 而进程 1 想进入临界区 B。Ricart 和 Agrawala 的算法会导致死锁吗？请解释原因。

14. 在图 5.17 中，我们看到一种用磁带实现对存货表自动更新的方法。因为磁带可以很容易地用磁盘上的一个文件来模拟，你认为为什么不使用这种方法（用磁盘文件代替）呢？

15. 在图 5.25(d)中有三个调度策略，两个合法的，一个非法的。对于同一个事务处理，给出最终 x 所有的可能值和合法的与非法的状态的一个完整表。

16. 当一个私有工作空间用于实现事务处理时，可能需要将大量的文件索引复制到父辈工作区。怎样在不引入竞争条件下实现这种操作？

17. 给出一个完整的算法判定一个试图锁住一个文件的操作是否成功。要同时考虑到读锁和写锁，以及文件被解锁、读锁定或写锁定的可能性。

18. 用锁定的方法实现并发控制的系统通常区分读锁和写锁。如果一个进程已经得到了一个读锁但现在又想将它换成写锁，将会怎么样呢？

19. 在分布式事务中使用时间戳排序，假设写操作 $\text{write}(T_1, x)$ 被传给数据管理器，因为仅有的可能发生冲突的操作 $\text{write}(T_2, x)$ 的时间戳较早。为什么让调度管理器推迟传递 $\text{write}(T_1, x)$ 直到 T_2 完成呢？

20. 乐观并发控制与使用时间戳相比是更严格还是更不严格？为什么？

21. 使用时间戳来进行并发控制能保证串行性吗？请讨论。

22. 我们常说当事务被中止时，一切都恢复到它以前的状态，就好像事务从来没有发生一样。实际上我们在说谎。请给出一个重新恢复一切是不可能的例子。

第6章 一致性和复制

分布式系统的一个重要问题是数据的复制。对数据进行复制一般是为了提高系统的可靠性或性能。实现数据复制的一个主要难题是保持各个副本的一致性。简单地说，这个难题就是更新一个拷贝时，必须确保同时更新其他拷贝；否则，数据的各个副本将不再相同。本章将深入讨论复制数据的一致性的真正含义，以及实现这种一致性的各种方法。

首先，我们总体性地介绍数据复制，讨论复制的用途以及它与可扩展性的关系。其中将特别讨论基于对象的复制，对象复制已逐渐成为许多分布式系统的重要部分。

为了实现对共享数据的高性能操作，并行计算机的设计人员已经深入研究了分布式共享存储器系统的各种一致性模型。这些模型同样适用于其他类型的分布式系统。本章将广泛地讨论这些模型。

通常，在大规模分布式系统中有效地实现共享数据的一致性模型是困难的，很多情况下可以采用较为简单的模型。通常这些较简单的模型比较容易实现。以客户为中心的一致性模型就是其中一种。这种模型从一个单一的（可能是移动的）客户的角度关注一致性。我们将在单独的一节中讨论这种以客户为中心的一致性模型。

一致性仅仅是问题的一个方面。我们还要考虑如何真正地实现一致性。有两个问题在保持副本一致性方面举足轻重。第一个问题是数据更新的实际分发问题，它需要关心副本的位置，以及如何在副本之间传播更新。我们将在本章介绍不同的分发协议，并对它们进行比较。

第二个问题是如何保持多个副本的一致性。在大多数情况下，应用程序都要求数据保持很强的一致性。也就是说，这意味着数据更新必须很快地在副本之间得到传播。实现这种强一致性有多种不同的方法，这些方法将在单独的章节中进行讨论。另外，我们还将讨论一类特殊的一致性协议，即缓存协议。

最后，我们将以两个应用实例结束本章。这两个实例广泛地应用了数据一致性和复制。第一个实例来源于并行编程领域，它也有助于理解第9章中关于基于对象的分布式系统的讨论。第二个实例结合了因果一致性和称为懒惰复制（lazy replication）的技术。

6.1 简介

本节首先讨论进行数据复制的主要原因。特别值得注意的是对象复制，因为它是现代分布式系统中越来越重要的一个话题。最后，我们将复制作为一种实现可扩展性的技术进行讨论，并指出为什么一致性如此重要的原因。

6.1.1 复制的目的

进行数据复制主要出于两个目的：可靠性和性能。首先，数据复制可以提高系统的可靠性。如果一个文件系统已经实现数据复制，那么当一个副本被破坏后，文件系统只需转换到另一个数据副本就可以继续运行下去。同样地，通过维护多个拷贝，系统可以为被破坏的数据提供更好的保护。例如，假设一个文件存在三个拷贝，而且每个读写操作都在各个拷贝上执行。这样，可以保护数据不会因为一个失败的写操作而受到破坏，因为此时至少两个拷贝的值是正确的。

进行数据复制的另一个目的是提高性能。当分布式系统需要在服务器数量和地理区域上进行扩展时，复制对于提高性能是相当重要的。例如，当需要访问由单一服务器管理的数据的进程数不断增加时，系统就需要进行服务器数量上的扩充。在这种情况下，通过对服务器进行复制，随后让它们分担工作负荷可以提高性能。在第1章对复制的Web服务器群集进行简要讨论时，我们已经看到了这样的一个实例。

地理区域上的扩展也可能需要进行数据复制。其基本思想是，如果在使用数据的进程附近放置一份该数据的拷贝，那么进程访问数据所花费的时间将减少。对于这个进程来说，性能就相对地提高了。但这个实例也说明了复制给性能带来的益处是难以评估的。虽然一个客户进程可能获得较好的性能，但是这也可能导致为维持所有副本的一致性而消耗了更多的网络带宽。我们将在讨论分发协议时再次讨论这种性能上的权衡。

如果复制有助于提高可靠性和性能，那么没有人会对采用它提出反对意见。然而不幸的是，进行数据复制需要付出一定的代价。复制所带来的问题是多个拷贝可能导致一致性方面的问题。当修改了一个拷贝后，它将不同于其他所有的拷贝。因此，必须对所有的拷贝进行同样的修改以确保一致性。执行这些修改的确切时间和方式决定了复制的代价有多大。

为了理解这个问题，我们来考虑一下改善Web页面访问时间的实例。如果不采用特别的方法，从远程Web服务器获取一个页面有时甚至需要几秒的时间才能完成。为了提高性能，Web浏览器通常都在本地存储一个以前获取的该Web页面的拷贝（也就是说，Web浏览器对Web页面进行缓存）。如果用户需要再次访问那个页面，浏览器会自动返回本地拷贝。这样，用户所觉察到的访问时间是非常短的。然而，如果用户总是想获得最新版本的页面，他可能就会不那么走运了。问题在于，如果此时对页面进行了更新，而这些更新还没有传播到被缓存的拷贝，那么缓存的拷贝将成为过时的版本。

为了解决这个问题，一种方法是禁止浏览器保存本地拷贝，由服务器完全负责复制事宜。然而，如果用户附近没有数据副本，这种方法会导致访问时间的延长。另一种方法是由Web服务器标识出无效的缓存拷贝并对它们进行更新，但是这种方法要求服务器跟踪所有的高速缓存，并向它们发送消息。这可能会降低服务器的整体性能。我们将在下面接着讨论性能与可扩展性的问题。

6.1.2 对象复制

为了更好地理解分布式系统在管理共享的复制数据方面的作用应考虑对象的复制，而不仅是数据的复制。对象具有封装数据及对数据的操作的优点。通过使用对象，我们

更容易区分针对某些数据的操作和与数据无关的操作。后一种类型的操作一般属于通用的分布式系统,例如,使用由本书所讨论的很多中间件系统来实现的分布式系统。

我们分析一下如图 6.1 所示的由多个客户所共享的分布式远程对象。在考虑多台机器之间复制一个远程对象之前,我们需要解决的问题是如何保护这个对象,以防止多个客户同时访问它。这个问题基本上有两种解决方法(请参看 Briot 等 1998)。

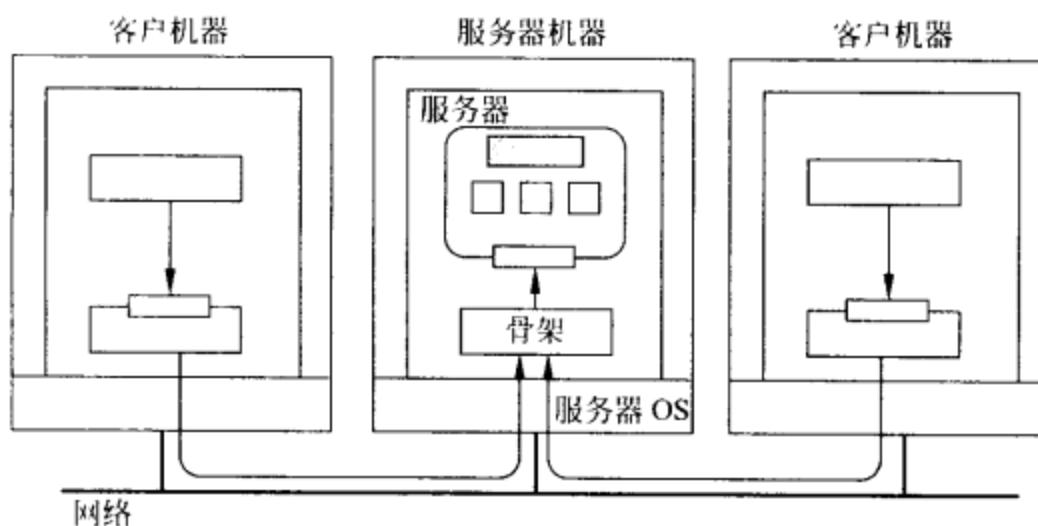


图 6.1 由两个不同客户共享的分布式远程对象的组织

第一种解决方法是让对象自身可以处理并发请求。我们以 Java 对象为例说明这一方案。通过将对象的方法声明为同步的(synchronized),可以把一个 Java 对象构造成一个监控器。假设两个客户同时请求引用同一对象的方法,这将导致对象所在服务器上产生两个并发线程。在 Java 中,如果对象的方法已经声明为同步的,那么只能允许这两个线程中的一个线程继续执行,而另一个线程将被阻塞,直到获得进一步的通知。可以存在不同级别的并发性,但问题的重点是对象自身实现了处理并发请求的方法。这一原理如图 6.2(a)所示。

第二种解决方案是对象不对防止并发引用进行任何保护,而是由对象所在的服务器负责并发控制。特别是,通过使用适当的对象适配器,可以确保并发引用不会使对象遭到破坏。例如,这种对象适配器可以是一种对每个对象使用单一线程的适配器,它可以对其管理的每个对象的访问进行有效的串行化,如图 6.2(b)所示。

在复制一个共享的远程对象时,如果不采取任何专门的方法处理并发调用,则可能导致出现一致性方面的问题。这些一致性问题是由于忽略以下事实引起的:需要有附加的同步操作来确保并发调用在每个副本上都以正确的顺序执行。这种同步问题的一个实例就是第 5.2 节中讨论的银行账号数据库问题。通常,这个问题同样只有两种解决方法。

第一种方法是对象知道自己可以被复制。在这种情况下,发生并发调用时,对象负责维护其副本的一致性。这种方法与自身负责处理并发调用的对象非常相似。这种基于对象的分布式系统基本不需要提供任何对复制的一般性支持。支持可能局限于提供有助于构建感知复制的对象的服务器和适配器,如 6.3(a)所示。这种系统的实例 SOS(Shapiro 等 1989)和 Globe(van Steen 等 1999a)将在第 9 章进行讨论。支持复制的对象的一个优点是可以采用特定于对象的复制策略,这种情况与并发对象可以自己处理并发调用的情

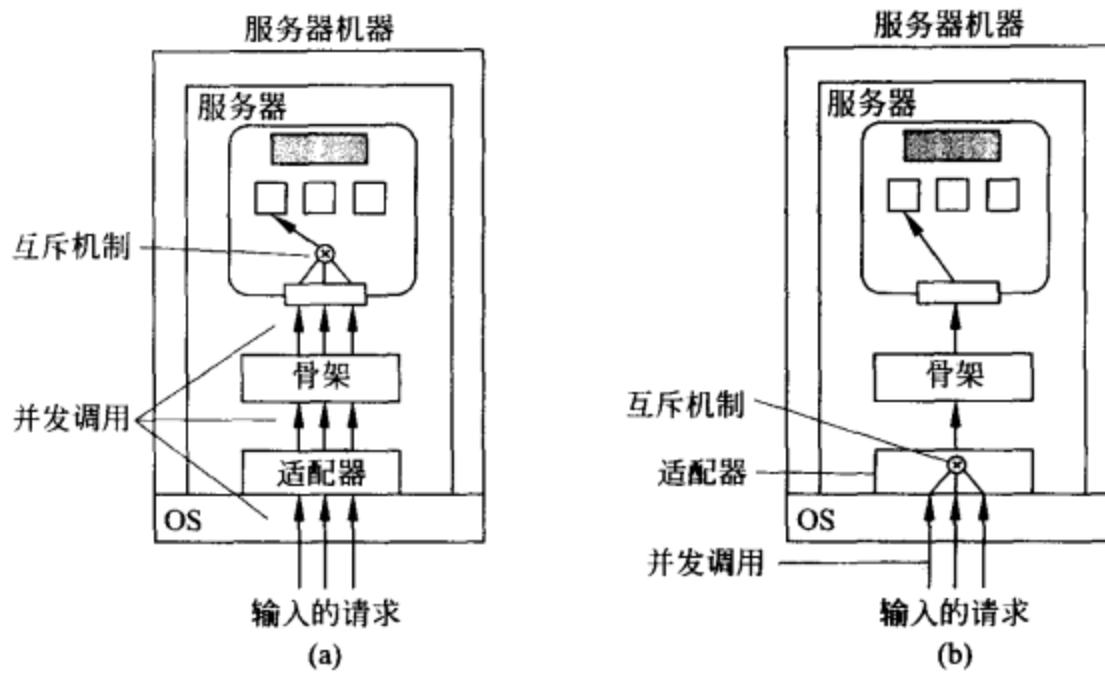


图 6.2 两种远程对象

(a) 自身能够处理并发调用的远程对象；(b) 需要对象适配器处理并发调用的远程对象

况相似。

处理并发对象一致性问题的第二种方法更为常见,那就是让分布式系统负责管理复制,如图 6.3(b)所示。具体来说,分布式系统负责确保并发调用按照正确的顺序传递给各个副本。例如,Piranha(Maffeis 1997)就采用了这种方法,Piranha 在 CORBA 中提供容错、完全排序和因果排序的对象调用机制。让分布式系统管理副本的优点是可以简化应用程序开发人员的工作。有时,采用针对对象的解决方法是比较困难的。这可能成为针对对象的解决方法的一个缺陷。正如我们将看到的,在解决可扩展性问题时,通常需要使用第二种解决方法。

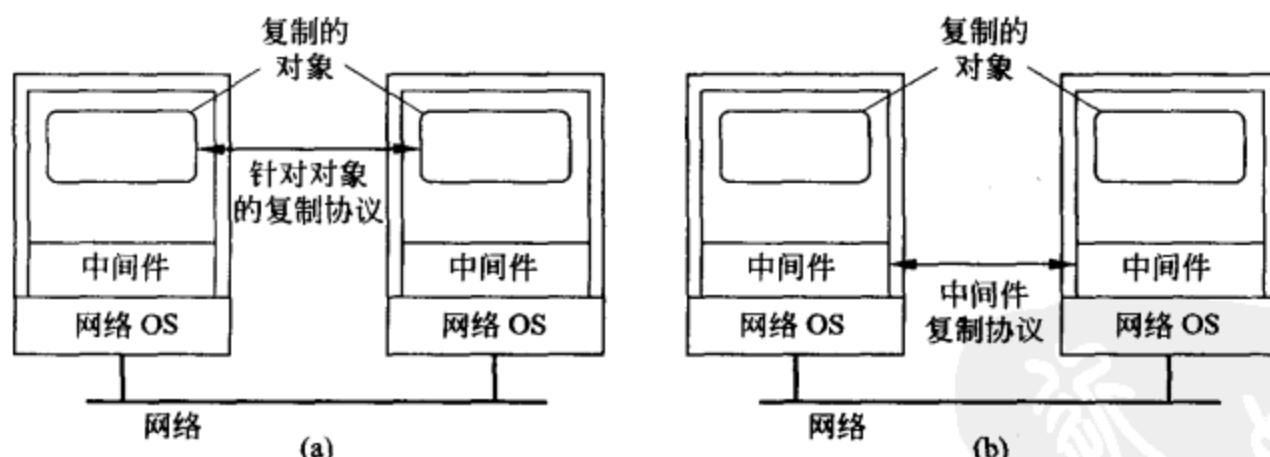


图 6.3 两种分布式系统

(a) 基于感知复制的分布式对象的分布式系统；(b) 负责副本管理的分布式系统

6.1.3 作为扩展技术的复制

为了提高性能,复制和缓存作为扩展技术得到了广泛的应用。可扩展性问题往往以性能问题的形式出现。将数据和对象的拷贝放置在使用它们的进程的附近,可以降低访

问时间,提高性能,从而解决可扩展性问题。

一个可能付出的代价是使拷贝保持为最新的数据需要更多的网络带宽。例如,一个进程 P 每秒访问一个本地副本 n 次,而这个副本自身每秒更新 m 次。假设每次更新都完全刷新本地副本的原有版本。如果 $n < m$,也就是说,访问与更新的比率很低,那么我们所面临的情况是很多本地副本的更新版本从未被 P 访问过,使得与这些版本更新有关的网络通信毫无用处。在这种情况下,最好不要在 P 附近安装本地副本,或者应该采用其他更新副本的策略。后面还会继续讨论这一问题。

然而,一个更为严重的问题是,保持多个拷贝间的一致性本身就可能存在严重的可扩展性问题。直观地看,当多个拷贝总保持相同的时候,这些拷贝的集合是一致的。这意味着在任何拷贝上执行读操作都将返回相同的结果。因此,在一个拷贝上执行更新操作时,无论这一操作是在哪个拷贝上启动或执行的,这一更新操作都应该在后续操作发生前传播到所有拷贝。

这种类型的一致性有时不规范地(也不准确地)称为强一致性,如同所谓的同步复制(Buretta 1997)所提供的一致性。(我们将在下一节中给出一致性的准确定义,并引入一系列一致性模型)。其关键思想是以单个的原子操作或事务的形式在所有拷贝上执行更新。不幸的是,当这种原子操作涉及到大量的、可能广泛散布在大型网络中的副本,同时要求原子操作快速完成时,实现这些操作是相当困难的。

其中的困难来自于需要同步所有副本这一事实。从本质上说,这意味着所有副本首先需要就本地执行更新的确切时间达成共识。例如,副本可能需要使用 Lamport 时间戳来决定执行操作的全局顺序,或由一个协调器来分配这样一个顺序。全局同步需占用大量通信时间,特别是副本散布于广域网时更是如此。

现在我们处于进退两难的境地。一方面,可扩展性问题可以通过采用复制和缓存技术得到缓解。而另一方面,保持所有拷贝的一致性通常要求全局同步,而全局同步必然造成性能的严重下降。这似乎有些得不偿失。

在很多情况下,惟一可行的解决方法是放宽在一致性方面的限制。也就是说,如果可以放松“更新必须以原子操作的方式执行”这一要求,就可以避免进行瞬间的全局同步,从而可以获得较好的性能。所付出的代价是各个拷贝可能并不总是相同的。事实表明,一致性可被放宽的程度主要取决于复制数据的访问和更新模式,同时还取决于这些数据的用途。

在以下几节中,我们将先讨论一系列一致性模型。这些模型提供一致性的准确定义。然后,我们将讨论通过分发和一致性协议实现一致性模型的不同方法。有关一致性和复制的不同分类方法可以参见(Gray 等 1996; Wiesmann 等 2000)。

6.2 以数据为中心的一致性模型

通常,总是在讨论共享数据的读操作和写操作时讨论一致性问题。这些共享数据是通过分布式共享存储器、分布式共享数据库或分布式文件系统实现的。在本节中,我们将使用更广义的术语——数据存储(data store)。数据存储可以物理地分布在多台机器上。

特别是,假设每个可以访问数据存储中的数据的进程都有整个数据存储的一个本地或邻近的拷贝,写操作将传播到其他拷贝,如图 6.4 所示。一个更改数据的数据操作被归类为写操作,否则归类为读操作。

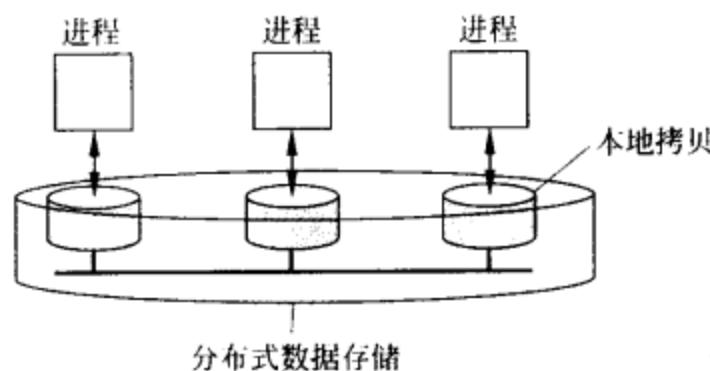


图 6.4 逻辑数据存储的一般组织,该数据存储在物理上是分布的,并被复制到多个进程中

一致性模型实质上是进程和数据存储之间的一个约定。即,如果进程同意遵守某些规则,那么数据存储将正常运行。正常情况下,一个进程在一个数据项上执行读操作时,它期待该操作返回的是该数据在其最后一次写操作之后的结果。

在没有全局时钟的情况下,精确地定义哪次写操作是最后一次写操作是十分困难的。作为替代的方法,我们需要提供其他的定义,因此产生了一系列的一致性模型。每种模型都有效地限制了在一个数据项上执行一次读操作所应返回的值。正如可以预料到的,带有较少限制的模型比较容易使用,而带有较多限制的模型使用起来则比较困难。当然,在另一方面,容易使用的模型的执行效果不如较复杂的模型那样好。生活往往就是这样。关于一致性模型的其他信息,请参见文献(Mosberger 1993, Adve 和 Gharachorloo 1996)。

6.2.1 严格一致性

最严格的一致性模型称为严格一致性(strict consistency)。它由以下条件定义:

对于数据项 x 的任何读操作将返回最近一次对 x 进行的写操作的结果所对应的值。

这一定义既自然又明显,尽管它隐含地假设存在绝对的全局时间(就像在牛顿物理学中那样)才致使“最近的”没有任何不确定性。通常,单一处理器系统遵守严格一致性。单一处理器的程序员也预期这种行为方式成为理所当然的事情。如果一个系统对程序

```
a=1;a=2;print(a);
```

输出 1 或其他非 2 的数值,那么它将使许多程序员不安,这种不安也是理所当然的。

对于一个系统,如果其数据散布在多台机器上,并且多个进程可以访问其数据,那么情况就相对比较复杂。假设 x 是仅存储在机器 B 上的数据项。如果机器 A 上的一个进程在 T_1 时间读取 x ,那就意味着向 B 发送一条读取 x 的消息。在稍后的 T_2 时间,机器 B 上的一个进程对 x 进行了一次写操作。如果系统保持严格一致性,那么无论这两台机器的相对位置如何,时间 T_2 多么接近 T_1 ,读操作应该总是返回旧数值。然而,如果 $T_2 - T_1$ 为 1ns,两台机器相距 3m,那么,为了在写操作之前,将读请求从 A 传送到 B 以获得数据,

信号必须以 10 倍光速的速度传输。显然这违背了爱因斯坦的相对论。那么，程序员要求系统保持严格的一致性是否合理呢？这种要求违背了物理定理吗？

严格一致性中存在的问题是它依赖于绝对的全局时间。从本质上说，在分布式系统中为每个与准确的全局时间对应的操作分配唯一的时间戳是不可能的。我们可以放宽条件，将时间分割成一系列连续的、不重叠的时间间隔。假设每个操作发生在一个时间间隔内，并获得与那个时间间隔对应的时间戳。根据时钟进行同步的精确程度，我们可以实现每个时间间隔最多只有一个操作的情况。

不幸的是,无法保证每个时间间隔内最多只发生一个单一操作。因此,我们仍需处理在同一时间间隔内所发生的多个操作。与解决分布式的事务处理时所讨论的并发控制相似,如果同一时间间隔的两个操作是对相同的数据进行操作,并且其中一个操作是写操作,那么我们称这两个操作是冲突的。定义一致性模型的一个重要问题就是准确定义出现冲突操作时,哪些行为是可接受的。

为了详细地研究一致性,我们将给出几个实例。为准确地描述这些实例,我们引入一个特殊的标记法。在这种标记法中,我们将进程的操作标在时间轴上。时间轴总是水平方向的,且时间从左向右递增。符号 $W_i(x)a$ 和 $R_i(x)b$ 分别表示进程 P_i 对数据项 x 写入值 a ,以及进程 P_i 读取数据项 x 得到值 b 。我们假设每个数据项的初始值为 NIL。当进程访问数据而未发生冲突时,我们忽略符号 W 和 R 的下标。

例如,图 6.5(a)对数据项 x 进行了一次读操作,将其值修改为 a。注意,从原理上说,操作 $W_1(x)a$ 首先在进程 P_1 本地的数据存储的副本上执行,然后再将结果传送到其他本地副本上。在这个例子中,进程 P_2 接着(从数据存储的本地副本中)读取 x,并得到值 a。对于严格的一致性数据存储来说,这一行为是正确的。与之相对照,图 6.5(b)中的 P_2 在写操作之后执行读操作(读操作也许仅比写操作晚 1ns,但毕竟在写操作之后),它得到 NIL。接下来的读操作得到 a。对于严格的一致性数据存储来说,这种行为是不正确的。



图 6.5 两个对同一数据项进行操作的进程。水平轴为时间轴

(a) 严格的一致性存储；(b) 非严格的一致性存储

总之,当数据存储是严格一致的时候,对于所有的进程来说,所有写操作都是瞬间可见的,系统维持着一个绝对的全局时间顺序。如果一个数据项被改变了,那么无论数据项改变之后多久执行读操作,无论哪些进程执行读操作,无论这些进程的位置如何,所有在该数据项上执行的后续读操作都将得到新数值。同样,如果执行了读操作,那么无论多快地执行下一个写操作,该读操作都将得到当前的值。

下一节中，我们将考虑用时间间隔代替绝对时间以放宽一致性模型的要求，并精确定义哪些行为是冲突操作中可接受的行为。

6.2.2 线性和顺序一致性

尽管严格一致性是理想的一致性模型,但是在分布式系统中,它无法实现。而且,经验表明,程序人员通常可以相当好地控制要求较弱的模型。例如,所有关于操作系统的教科书都有关于临界区和互斥问题的讨论。这些讨论通常包括以下限制:正确地写出的开发程序(例如,生产者—消费者问题)不应该做任何关于进程的相对速度以及进程的语句在时间上交叉的假设。考虑在一个进程中快速地发生两个事件以至于其他进程不能在其间做任何事情是自找麻烦。况且,对于读程序来说,语句执行(实际是内存引用)的精确顺序无关紧要。当事件的执行顺序十分关键时,可以使用信号量或其他同步操作。接受这个观点意味着可以使用要求较弱的一致性模型。

与严格一致性相比,顺序一致性(sequential consistency)是一个要求稍弱的一致性模型。它是由 Lamport(1979)在解决多处理器系统的共享存储器时首次提出的。通常,数据存储满足以下条件时,称为是顺序一致的:

任何执行结果都是相同的,就好像所有进程对数据存储的读、写操作是按某种序列顺序执行的,并且每个进程的操作按照程序所制定的顺序出现在这个序列中。

以上定义的意思是,当进程在多台(可能)不同机器上并发运行时,任何读、写操作的有效交叉都是可接受的行为,但是所有进程都看到相同的操作交叉。注意,这里并未提及时间,也就是说,这里并未提及“最近一次”对一个对象的写操作。在这里,进程可以“看到”所有进程的写操作,但是只能看到它自己的读操作。

从图 6.6 可以看出,时间不起任何作用。假设有 4 个进程对同一数据项 x 进行操作。图 6.6(a)中的进程 P1 对 x 执行 $W(x)a$ 写操作,随后(绝对时间),进程 P2 也执行了一次写操作,将 x 的值设置为 b 。然而,进程 P3 和 P4 先读到值 b ,然后才读到值 a 。换句话说,进程 P2 的写操作发生在进程 P1 的写操作之前。

P1: W(x)a	P1: W(x)a
P2: W(x)b	P2: W(x)b
P3: R(x)b R(x)a	P3: R(x)b R(x)a
P4: R(x)b R(x)a	P4: R(x)a R(x)b
(a)	(b)

图 6.6 顺序一致的与非顺序一致的数据存储

(a) 顺序一致的数据存储; (b) 非顺序一致的数据存储

与之对照,图 6.6(b)违背了顺序一致性,因为不是所有的进程看到了同样的写操作的交叉。特别是对进程 P3 而言,数据项好像首先变为 b ,然后变为 a 。而进程 P4 却得到数据项的最终值 b 。

一种弱于严格一致性但又强于顺序一致性的一致性模型是线性化。这种模型假设操作具有一个全局有效时钟的时间戳,但是这个时钟仅具有有限的精确度。假设进程使用上一章所讨论的宽松的同步时钟,那么可以在分布式系统中实现这样的时钟。设 $ts_{op}(x)$ 表示给操作 OP 分配的时间戳,操作 OP 对数据项 x 进行读(R)操作或写(W)操作。当数据存储上的每个操作都具有时间戳并满足以下条件时,称这个数据存储是可线性化的

(Herlihy 和 Wing 1991):

任何执行结果都是相同的,就好像所有进程对数据存储的读、写操作是按某种顺序执行的,并且每个进程的操作按照程序所制定的顺序出现在这个顺序中。另外,如果 $ts_{op1}(x) < ts_{op2}(y)$,那么在这个顺序中,操作 $OP1(x)$ 出现在操作 $OP2(y)$ 之前。

注意,可线性化的数据存储也是顺序一致的。它们的区别在于:线性化是根据一系列同步时钟确定序列顺序的。在实际应用中,线性化主要用于并发算法的形式验证(Herlihy 和 Wing 1991)。关于根据时间戳维护顺序的附加限制使得线性化的实现比顺序一致性的实现开销更大,请参阅文献(Attiya 和 Welch 1994)。

正如上一章所述,在事务处理方面,顺序一致性可以与串行化相比。回忆一下,如果最后结果也可以通过按照某种序列顺序逐个地执行事务处理获得,那么这个并发执行的事务处理集合是可串行化的。主要区别在于定义的粒度不同:顺序一致性是根据读操作和写操作定义的。而串行化是根据事务处理定义的。事务处理是一系列读、写操作的集合。

为了更具体地理解顺序一致性的含义,让我们讨论一下图 6.7(Dubois 等 1988)所示的三个并发执行的进程 P1、P2 和 P3。在这个例子中,数据项是三个整型变量 x、y 和 z,它们存储在顺序一致(也可能是分布的)的共享数据存储中。假设每个变量的初始值都是 0。在本例中,赋值语句与写操作对应,而打印语句与该时刻它所带两个参数的读操作对应。

进程 P1	进程 P2	进程 P3
x=1;	y=1;	z=1;
print(y, z);	print(x, z);	print(x, y);

图 6.7 三个并发执行的进程

各种交叉执行序列都是可能的。对于 6 个独立的语句而言,将有 720(即 $6!$) 种可能的执行序列,其中某些序列可能违背了程序顺序。假设 120(即 $5!$) 种序列以 $x=1$ 开始,一半的序列在 $y=1$ 之前执行了 $print(x, z)$,那么这些序列就违背了程序顺序。还有一半的序列在 $z=1$ 之前执行了 $print(x, y)$,也违背了程序顺序。这样,只有 120 种的四分之一的序列,也就是 30 种序列是有效的。以 $y=1$ 开始的序列可能有 30 种有效序列,以 $z=1$ 开始的有效序列也有 30 种。因此,总共有 90 种有效的执行序列。图 6.8 列出了其中 4 种有效序列。

在图 6.8(a) 中,三个进程以如下顺序运行:首先运行 P1,然后运行 P2,最后运行 P3。而另外三个实例则说明了虽然在时间上不同但同时有效的语句的交叉。三个进程都输出两个变量的值。因为每个变量可取的惟一值都是初始值(0),或所赋的值(1),所以每个进程生成一个两位的字符串。Prints 后的数字是显示在输出设备上的实际输出。

如果将 P1、P2 和 P3 的输出按照输出顺序连接起来,那么所得到的 6 位字符串说明了每种语句交叉的特点,如图 6.8 所示的 Signature(签名)字符串。下面我们将使用该字符串(而不是输出结果)来表示语句的执行顺序。

不是所有 64 个签名模式都是允许的。例如,000000 就是不允许的,因为它意味着显示语句在赋值语句之前执行,这违反了语句必须按程序顺序执行的要求。另一个例子是 001001。前两位为 00,意味着当 P1 执行显示语句时,b 和 c 的值都是 0。只有 P1 在 P2

或 P3 启动之前执行这条语句才会出现这种情况。接下来的两位为 10, 意味着 P2 必须在 P1 启动后而 P3 仍未启动前执行。最后两位为 01, 意味着 P3 必须在 P1 启动前完成。而我们已经看到 P1 必须先执行, 所以, 001001 也是不允许的。

x=1;	x=1;	y=1;	y=1;
print(y,z)	y=1;	z=1;	x=1;
y=1;	print(x,z);	print(x,y);	z=1;
print(x,z);	print(y,z);	print(x,z);	print(x,z);
z=1;	z=1;	x=1;	print(y,z);
print(x,y);	print(x,y);	print(y,z);	print(x,y);
Prints:001011	Prints:101011	Prints:010111	Prints:111111
Signature:001011	Signature:101011	Signature:110101	Signature:111111

(a)

(b)

(c)

(d)

图 6.8 图 6.7 中进程的 4 个有效执行顺序。垂直轴是时间轴

简而言之, 90 个不同的有效语句顺序产生各种各样不同的程序结果(虽然少于 64 种), 这些都是满足顺序一致性假设前提的结果。进程与分布式的共享数据存储间的协议是, 进程必须只接受有效的程序结果。换句话说, 进程必须只将图 6.8 所示的 4 种结果以及所有其他有效结果视为正确答案。如果出现其中任何一种结果, 进程都必须正常地运行。仅对其中部分结果可以运行, 而对另一些结果不能运行的程序违背了数据存储的协议, 是不正确的。

顺序一致性(和其他模型)都有许多的形式表示方法。下面介绍的方法(Ahamad 等 1992, Mizuno 等 1995)是一种通用的方法。每个进程 P_i 与执行过程 E_i 相关联, 其中执行过程 E_i 是进程 P_i 对数据存储 S 所执行的读操作和写操作的序列。这个序列符合与 P_i 关联的程序顺序。

例如, 图 6.6(a)中的 4 个线程的执行过程如下:

$E_1 : W_1(x)a$
 $E_2 : W_2(x)b$
 $E_3 : R_3(x)b, R_3(x)a$
 $E_4 : R_4(x)b, R_4(x)a$

为了获得所执行的操作的相对顺序, 我们必须将 E 中的操作字符串合并成一个单独的字符串 H , 其中每个操作表示为 E_i , 它在 H 中仅出现一次。 H 也被称为历史(history)。直观地看, 如果数据存储是一个单独的、集中的数据存储, H 给出了可能的操作执行顺序。 H 所有的合法的值必须满足两个条件限制:

- (1) 必须维持原有的程序顺序;
- (2) 必须考虑数据相干性。

第一个限制条件意味着如果在执行过程 E_i 的字符串中, 一个读操作或写操作 OP_1 出现在另一个操作 OP_2 之前, 那么在 H 中, OP_1 也必须出现在 OP_2 之前。如果所有的操作组合都满足这个限制条件, 那么所得的 H 必定不会出现任何违背程序顺序的操作

顺序。

第二个限制条件我们称之为数据相关性(data coherence),意味着读取数据项x的读操作R(x)必须总是返回最近写入x的值,即在读操作R(x)执行之前最近执行的写操作W(x)v所写入的值v。数据一致性孤立地检查每个数据项,以及该数据项上的操作序列,而不考虑其他数据项。与之相对照,一致性处理的是对不同数据项的写操作及它们的顺序。特别在处理分布式共享存储器和存储位置而非数据项时,数据相关性被称为存储相关性(memory coherence)。

在图6.6(a)所示的4个进程的实例中,一个H的合法值是:

$$H = W_1(x)b, R_3(x)b, R_4(x)b, W_2(x)a, R_3(x)a, R_4(x)a$$

另一方面,对于图6.6(b)中的4个进程的执行过程,则无法得到任何合法的历史,因为一个顺序一致的系统无法使进程P3先执行R₃(x)b,然后执行R₃(x)a,而进程P4却以不同的顺序读出值a和b。

更复杂的实例中,H可能有多个合法的值。如果程序的操作序列与H的某个合法值相对应,那么程序的行为就被认为是正确的。

尽管顺序一致性是一个对程序员友好的一致性模型,但是它存在着严重的性能问题。Lipton和Sandberg(1998)证明了:如果执行读操作的时间是r,执行写操作的时间是w,节点间最小的数据包传送时间是t,那么始终存在r+w>t。换句话说,对于任何的顺序一致性存储,改变协议以提高读操作性能必将降低写操作性能,反之亦然。因此,研究人员已经开始研究其他(较弱的)模型。我们将在以下几节讨论其中一些模型。

6.2.3 因果一致性

因果一致性(causal consistency)模型(Hutto和Ahamad 1990)代表一种弱化的顺序一致性模型,因为它将潜在具有因果关系的事件和没有因果关系的事件区分开来。上一章讨论向量时间戳时,我们已经遇到了因果关系的问题。如果事件B是由事件A引起的,或受到事件A的影响,那么因果关系必然要求其他每个人先看到事件A,再看到事件B。

考虑一个存储器的实例。假设进程P1对变量x执行了写操作。然后进程P2先读取x,然后对y执行写操作。这里,对x的读操作和对y的写操作具有潜在的因果关系,因为y的计算可能依赖于P2所读取的x的值(也就是,P1所写的值)。相反,如果两个进程自发且同时地对两个不同的变量执行写操作,这两个事件就不具备因果关系。当一个读操作后面跟着一个写操作时,这两个事件就具有潜在的因果关系。同样,读操作也与为读操作提供数据的写操作因果相关。没有因果关系的操作被称为并发的。

如果数据存储是因果一致的,那么它必须满足以下条件:

所有进程必须以相同的顺序看到具有潜在因果关系的写操作。不同机器上的进程可以以不同的顺序被看到并发的写操作。

图6.9是一个因果一致性的实例。这里的事件顺序是因果一致的存储所允许的,但该顺序是顺序一致的存储和严格一致的存储所禁止的。值得注意的是,写操作W₂(x)b

和 $W_1(x)c$ 是并发的, 所以没有必要要求所有进程以相同的顺序看到它们。

P1:	$W(x)a$	$W(x)c$	
P2:	$R(x)a$	$W(x)b$	
P3:	$R(x)a$		$R(x)c$
P4:	$R(x)a$		$R(x)b$

图 6.9 因果一致性存储所允许的, 但顺序和严格一致性存储不允许的顺序

现在, 我们看看另一个实例。图 6.10(a)中, $W_2(x)b$ 潜在地依赖于 $W_1(x)a$, 这是因为 b 可能是使用 $R_2(x)a$ 所读取的值进行计算的结果。两个写操作是因果相关的, 所以所有进程必须以相同的顺序看到这两个操作。因此, 图 6.10(a)是不正确的。但是, 图 6.10(b)删除了读操作, 所以 $W_1(x)a$ 和 $W_2(x)b$ 现在是并发的写操作了。因果一致的存储并不要求并发的写操作是全局有序的, 所以图 6.10(b)是正确的。

P1:	$W(x)a$			P1:	$W(x)a$		
P2:	$R(x)a$	$W(x)b$		P2:		$W(x)b$	
P3:		$R(x)b$	$R(x)a$	P3:		$R(x)b$	$R(x)a$
P4:		$R(x)a$	$R(x)b$	P4:		$R(x)a$	$R(x)b$

(a)

(b)

图 6.10 事件存储顺序

(a) 违背因果一致的事件存储顺序; (b) 符合因果一致的正确的事件存储顺序

实现因果一致性要求跟踪哪些进程看到了哪些写操作。这就意味着必须构建和维护一张记录哪些操作依赖于哪些操作的依赖关系图。一种实现方法是使用上一章所讨论的向量时间戳。我们将在本章后面再继续讨论使用向量时间戳捕获因果关系的方法。

6.2.4 FIFO 一致性

因果一致性允许不同的机器以不同的顺序看到并发的写操作, 但是所有机器必须以相同的顺序看到因果相关的写操作。如果放弃后一条要求, 这样就得到 FIFO 一致性 (FIFO consistency)。它满足以下条件:

所有进程以某个单一进程提出写操作的顺序看到这些写操作, 但是不同进程可以以不同的顺序看到不同进程提出的写操作。

在分布式共享存储器系统中, FIFO 一致性被称为 PRAM 一致性 (PRAM consistency), Lipton 和 Sandberg (1998) 对其有所描述。PRAM 代表管道化 RAM (pipelined RAM), 因为单一进程的写操作可以用管道处理, 也就是说, 进程不必在开始进行下一个写操作之前, 停止以等待前一个写操作的完成。图 6.11 对 FIFO 一致性与因果一致性进行对比。图中所示的事件顺序是 FIFO 一致的数据存储所允许的顺序, 但不是我们已经学过的任何更强模型所允许的顺序。

FIFO 一致性受到广泛关注, 这是因为它很容易实现。实际上, 这种模型不需要保证不同进程看到写操作的顺序, 除非两个以上的写操作是同一个进程提出的。这种情况下, 写操作必须按顺序达到。换句话说, 在这种模型中, 所有由不同进程产生的写操作都是并发的。这种模型可以通过如下方法实现: 简单地使用(进程, 序列号)对作为每个操作的

标签，并根据序列号执行每个进程的写操作。

P1:	W(x)a			
P2:		R(x)a	W(x)b	W(x)c
P3:				R(x)b R(x)a R(x)c
P4:				R(x)a R(x)b R(x)c

图 6.11 对 FIFO 一致性有效的事件顺序

现在，我们重新考虑图 6.7 所示的三个进程，这次我们使用 FIFO 一致性代替顺序一致性。在 FIFO 一致性中，不同的进程可以看到以不同的顺序执行的语句。例如，图 6.12(a) 表明 P1 所看到的事件，图 6.12(b) 表明 P2 所看到的事件，图 6.12(c) 表明 P3 所看到的事件。而对于顺序一致的存储来说，这三种不同的事件顺序都是不被允许的。

x=1;	x=1;	y=1;
print(y,z);	y=1;	print(x,z);
y=1;	print(x,z);	z=1;
print(x,z);	print(y,z);	print(x,y);
z=1;	z=1;	x=1;
print(x,y);	print(x,y);	print(y,z);
Prints:00	Prints:10	Prints:01;

(a)

(b)

(c)

图 6.12 图 6.7 的三个进程所看到的语句执行顺序，粗体的语句是产生所示输出的语句

如果我们把三个进程的输出连接起来，那么我们得到的结果是 001001。我们已经在前面看到，顺序一致性不可能产生这个结果。顺序一致性和 FIFO 一致性的主要区别在于：对于前者而言，尽管语句的执行顺序是非确定的，但是至少所有的进程都对某个顺序达成一致，对于后者而言，各个进程不需要达成一致，不同进程可以以不同的顺序看到操作。

有时，FIFO 一致性可能导致与直觉相反的结果。下面的实例假设整型变量 x 和 y 的初始值为 0，参见文献(Goodman 1989)。图 6.13 中，人们可能很自然地认为会出现以下三种可能的结果之一：P1 被终止、P2 被终止或两者都没被终止（如果先执行两个赋值语句）。但是，对于 FIFO 一致性而言，两个进程都可能被终止。如果 P1 看到 P2 执行写操作 W₂(y)1 之前执行读操作 R₁(y)0，并且 P2 看到 P1 执行写操作 W₁(x)1 之前执行读操作 R₁(x)0，那么将产生这种情况。对于顺序一致的数据存储而言，可能存在 6 种可能的语句交叉，没有任何一种语句交叉会导致两个进程都被终止的情况。

进程 P1	进程 P2
x=1;	y=1;
if(y==0)kill(P2);	if(x==0)kill(P1);

图 6.13 两个并发的进程

6.2.5 弱一致性

尽管 FIFO 一致性的性能优于较强的一致性模型的性能,但是它仍然对许多应用存在不必要的限制,这是因为这些应用要求从任何位置都可以按顺序看到某个单一进程所产生的写操作。然而,并不是所有的应用程序都要求看到所有的写操作,更不会要求按顺序看到它们。考虑一下这个例子,一个临界区内的进程向复制的数据库写入记录。尽管其他进程在第一个进程离开临界区之前,不能接触数据,但是数据库系统无法了解进程何时处于临界区内,何时处于临界区外,所以系统必须向该数据库的所有拷贝传播所有的写操作。

一个较好的解决方法是让进程结束它对临界区的占用,然后确保向所有地方发送最后结果,这种方法不太关心,甚至毫不关心所有中间结果是否已经按顺序传播到所有拷贝。通常,引入同步变量(synchronization variable)可以实现这种方法。同步变量 S 仅有一个关联操作 synchronize(S),该操作同步数据存储的所有本地拷贝。让我们回忆一下,进程 P 仅在存储的本地有效拷贝上执行操作。数据存储同步时,进程 P 的所有本地写操作都将被传播到其他拷贝,而其他进程的写操作也将被传播到 P 的本地拷贝。

使用同步变量来部分地定义一致性就得到称为弱一致性(weak consistency)模型(Dubois 等 1988)。弱一致性模型具有三个属性:

- (1) 对数据存储所关联的同步变量的访问是顺序一致的;
- (2) 每个拷贝完成所有先前执行的写操作之前,不允许对同步变量进行任何操作;
- (3) 所有先前对同步变量执行的操作都执行完毕之前,不允许对数据项进行任何读或写操作。

第一点说明所有进程都以相同的顺序看到对同步变量进行的所有操作。换句话说,如果进程 P1 调用 synchronize(S1),与此同时进程 P2 调用 synchronize(S2),那么实际效果好像是 synchronize(S1) 发生在 synchronize(S2) 之前,或 synchronize(S2) 发生在 synchronize(S1) 之前。

第二点说明同步“清空管道”。它强迫在所有拷贝上完成所有的写操作,其中包括正在执行的、部分完成的或已经在某些本地拷贝上完成却未在其他拷贝上完成的写操作。同步完成同样保证了完成所有先前进行的写操作。在更新共享数据后,通过执行一次同步,进程可以迫使新数值传送到该存储的所有其他本地拷贝。

第三点说明访问数据项时,无论读数据或写数据,所有先前的同步都已经完成。在读共享数据之前,通过执行一次同步,进程可以确信所得到的数值是最新的数值。

与前面的一致性模型不同,弱一致性不是使单个的读操作和写操作服从一致性,而是迫使一组操作服从一致性。如果对于共享数据来说,很少有访问是孤立的,大部分访问是群集的(即一段短时期内有很多访问,然后很长一段时间内没有访问),那么这种情况下,弱一致性模型非常有用。

与前面的一致性模型不同的另一个重要不同点是,它只限制了保持一致性的时间,而没有限制一致性的形式。实际上,我们可以说,弱一致性是在一组操作,而非单个操作上强迫执行顺序一致性。同步变量用于划分操作的组。

现在,对于存储器出现错误人们都习以为常了。许多编译器也在编译时出现“欺骗”现象。例如,图 6.14 的程序段,所有变量都被初始化为适当的数值。优化的编译器可能决定在寄存器内计算 a 和 b 的值,并将它们的值保存在寄存器中一段时间,而不更新它们所在存储器位置的值。仅当调用函数 f 时,编译器才不得不把 a 和 b 的当前值写回存储器,这是因为 f 可能试图访问它们。这是一种典型的编译器优化技术。

```

int a,b,c,d,e,x,y;           /* 变量 */
int * p, * q;                /* 指针 */
int f(int * p,int * q);      /* 函数原型 */

a=x * x;                     /* a 存放在寄存器中 */
b=y * y;                     /* b 同上 */
c=a * a * a+b * b+a * b;    /* 供以后使用 */
d=a * a * c;                 /* 供以后使用 */
p=&a;                        /* p 获得 a 的地址 */
q=&b;                        /* p 获得 b 的地址 */
e=f(p,q);                   /* 调用函数 */

```

图 6.14 其中部分变量存放在寄存器中的程序段

这里出现的存储器错误是可以接受的,因为编译器知道它正在做什么(即,因为软件并不坚持存储器必须实时更新)。显然,如果存在可能以不受控制的方式读取存储器内容的第二个进程,那么这种方法将不能正确工作。例如,如果在给 d 赋值的期间,第二个进程读取 a、b 和 c 的值,那么它将得到不一致的值(a 和 b 是旧数值,而 c 是新数值)。我们可以使用以下特殊方法避免发生这种混乱:编译器首先写一个特殊的标志位,用以标识存储器内容已经过时。如果另一个进程试图访问 a,它将等待标识位的复位。如果同步是通过软件完成的,并且各方遵从相同规则,那么这种方法使系统可以采用较差一点的一致性。

下面,我们讨论一个有些牵强的情况。在图 6.15(a)中,进程 P1 对数据项执行了两次写操作,然后执行同步(用字母 S 表示)。如果 P2 和 P3 还没有同步,对它们看到的东西就无法做任何保证,所以其中的事件顺序是有效的。

P1:W(x)a	W(x)b	S	P1:W(x)a	W(x)b	S
P2:	R(x)a	R(x)b	S		
P3:	R(x)b	R(x)a	S		
(a)					(b)

图 6.15 对于弱一致性有效及无效的事件顺序

(a) 对于弱一致性有效的事件顺序; (b) 对于弱一致性无效的事件顺序

但是,与图 6.15(b)不同。这里,P2 已经被同步了,这意味着它的数据存储的本地副本已经被更新了。当它读取 x 时,它必须得到值 b。弱一致性不允许它得到如图所示的值 a。

6.2.6 释放一致性

弱一致性存在一个问题，即当同步变量被访问时，数据存储不知道此次访问是因为进程已经结束对共享数据的写操作还是因为进程将开始读数据而进行的。因此，它必须对两种情况都采取一定措施，以保证所有本地启动的写操作都已经完成（也就是说，传播到所有其他拷贝），以及收到来自其他拷贝的所有写操作。如果存储可以分辨进程进入临界区和离开临界区之间的区别，那么可以采用一种更有效的实现方法。为了提供这一信息，需要使用两种类型的同步变量来代替原先的一种类型的同步变量。

释放一致性（release consistency）（Gharachorloo 等 1990）提供这两种类型的同步变量。获取（acquire）操作是用于通知数据存储进程进入临界区的操作，而释放（release）操作是表明进程刚刚离开临界区的操作。这两个操作可以通过以下两种方法实现：①对特殊变量的一般操作；②特殊操作。无论哪种方法，程序人员都将负责在程序中加入显式的代码，用以说明执行操作的时间，例如，通过调用诸如 acquire 和 release 一类的库函数或诸如 enter_critical_region 和 leave_critical_region 一类的函数。

释放一致性也可以使用障碍代替临界区。障碍（barrier）是一种同步机制，它可以防止任何进程在所有进程都已完成程序的第 n 阶段之前进入程序的第 n+1 阶段。当一个进程到达一个障碍，它必须一直等待，直到其他所有进程也都到达这个障碍。当最后一个进程到达这个障碍时，说明所有的共享数据都已经同步，然后所有进程才可以继续执行。获取操作表明进程离开障碍，而释放操作表明进程到达障碍。

除了这些同步操作外，对共享数据的读、写操作也是可以执行的。不需要对存储内的所有数据应用获取和释放操作，它们可以仅保护某些特殊的共享数据。在这种情况下，只有那些被保护的数据项是保持一致的。保持一致的共享数据被称为受保护的数据。

提供释放一致性的数据存储保证：当进程执行获取操作时，如果有必要它将确保受保护的数据的所有本地拷贝都被更新为最新数据以便与远程数据拷贝保持一致。在进程执行释放操作时，已经改变的受保护数据被传播到该存储的其他本地拷贝。执行获取操作并不能确保本地所做的改变会被立即传播到其他本地拷贝。同样，执行释放操作也不会必然地从其他拷贝引入改变。

图 6.16 显示了一个对于释放一致性有效的事件顺序。进程 P1 执行获取操作，两次改变一个共享数据项的值，然后执行释放操作。进程 P2 执行获取操作，并读取数据项 x 的值。可以保证，进程 P2 得到执行释放操作时 x 的值，也就是 b（除非 P2 在 P1 执行获取操作之前执行获取操作）。如果 P1 执行释放操作之前，P2 已经执行了获取操作，那么获取操作将等待释放操作的执行。因为 P3 在读取共享数据之前，没有执行获取操作，数据存储没有必要给出 x 的当前值，所以返回 a 是允许的。

P1:	Aca(L)	W(x)a	W(x)b	Rel(L)
P2:			Acq(L)	R(x)b
P3:				Rel(L) R(x)a

图 6.16 对于释放一致性有效的事件顺序

为了更清楚地理解释放一致性,下面简要地描述其在复制的数据库中的一种可能的简化实现。要执行获取操作,进程需向中央同步管理器发送一条消息,请求对一个特定的锁执行获取操作。在没有任何竞争的情况下,同步管理器将同意请求,完成获取操作。然后,进程可以对共享数据的本地拷贝进行任意顺序的读、写操作。这些操作都不会传播给数据库的其他拷贝。当释放操作被执行后,被修改的数据将被发送到使用这些数据的其他拷贝。每个拷贝都确认已经收到数据后,同步管理器将被通知释放锁。通过这种方式,可以在固定开销的情况下,对共享数据进行任意数量的读、写操作。不同锁的获取和释放操作是相互独立的。

尽管上述的集中算法可以完成任务,但是它绝不是惟一的方法。通常,如果一个分布式的数据存储满足以下规则,那么它就是释放一致的:

- (1) 对共享数据执行读操作或写操作之前,所有进程先前执行的获取操作都必须已经成功完成;
- (2) 在释放操作被允许执行前,所有进程先前执行的读操作和写操作都必须已经完成;
- (3) 对同步变量的访问是 FIFO 一致的(不需要顺序一致)。

如果满足所有这些条件,并且进程正确执行获取和释放操作(也就是说,成对地执行获取和释放操作),那么任何执行顺序的结果都与它们在顺序一致的数据存储下执行的结果相同。实际上,获取和释放原语使对共享数据上的操作块成为原子操作,以防止出现交叉。

一种不同的释放一致性的实现是懒惰释放一致性(lazy release consistency)(Keleher 等 1992)。在常规的释放一致性中(我们称其为急切释放一致性(eager release consistency)以区别于懒惰释放一致性),在执行释放操作时,执行释放操作的进程将所有修改过的数据发送到所有拥有该数据的拷贝而潜在需要该数据的其他进程。因为没有办法区分哪些是否真的需要该数据,所以为了保险起见,任何改变过的数据都将被发送给所有进程。

尽管这种发送所有数据的方法是直截了当的,但是它的效率通常很低。在懒惰释放一致性中,在执行释放操作时,不向任何地方发送任何数据。相反,执行获取操作时,试图执行获取操作的进程必须从持有数据的进程那里获得数据的最新值。可以使用一个时间戳协议来确定哪些数据项是实际必须被传输的。

在很多程序中,临界区位于一个循环内部。在急切释放一致性下,每次执行循环的释放操作时,所有修改过的数据都必须被发送到所有维护这些数据的拷贝的进程那里。这种算法浪费网络带宽,也引入了不必要的延迟。在懒惰释放一致性下,执行释放操作时,不执行任何其他操作。在下一次执行获取操作时,如果进程确定它已经拥有所有需要的数据,那么就不会产生任何消息传输。最终结果是,在懒惰释放一致性下,其他进程执行获取操作前根本不会产生任何网络通信。在没有外部竞争的情况下,同一进程可以自由地重复执行获取—释放操作对。

6.2.7 入口一致性

另一种用于实现临界区的一致性模型是入口一致性(entry consistency)(Bershad 等 1993)。与释放一致性的两种变体一样,它需要程序员(或编译器)在每个临界区的开始

和结束处分别使用获取和释放操作。然而,与释放一致性不同的是,人口一致性要求每个普通的共享数据项都要与某种同步变量(如锁或障碍)关联。如果需要并行地、独立地访问数组的多个元素,那么不同的数组元素必须与不同的锁关联。对一个同步变量执行获取操作时,只有该同步变量保护的数据是保持一致的。人口一致性与懒惰释放一致性不同,因为后者并不把数据项与锁、障碍关联起来,而且在执行获取操作时,它必须以经验确定它需要哪些变量。

将一系列共享数据项与各自的同步变量关联起来可以减少获取和释放一个同步变量所带来的额外开销,这是因为只有少数共享数据项必须同步。这也使得多个包含不同共享数据的临界区可以同时执行,从而增加系统的并行度。所付出的代价是将每个共享数据项与某个同步变量关联所带来的额外负载和复杂性。以这种方式进行程序设计也可能更加复杂、易于出错。

同步变量可以按如下方法使用:每个同步变量都有一个当前的所有者,即最后获取它的进程。所有者不必在网络上发送任何消息,就可以重复地进入、退出临界区。而一个当前不拥有一个同步变量的进程想要获取这个同步变量,它必须向当前的所有者发送消息,请求获得这个同步变量的所有权及其关联数据的当前值。多个进程也可以以非独占的方式同时拥有一个同步变量,此时这些进程只能读,而不能写这个同步变量关联的数据。

形式上,如果一个数据存储满足以下所有条件,那么它符合人口一致性(Bershad 和 Zekausas 1991):

(1) 在一个进程可以获取一个同步变量之前,所有的由此同步变量保护的共享数据的更新都必须已经相对于该进程执行完毕;

(2) 在一个进程对一个同步变量的独占访问被允许执行之前,其他的进程不可以拥有这个同步变量,甚至也不能以非独占的方式拥有这个同步变量;

(3) 一个进程对一个同步变量执行独占访问之后,在对该同步变量的所有者进行检查之前,任何其他的进程都不能执行下一个非独占访问。

第一个条件表明,当一个进程执行获取操作时,在所有的受保护的共享数据还没有更新之前,获取操作不能完成(也就是将控制权转到下一条语句)。换句话来说,在执行获取操作时,所有的受保护数据的远程改变都必须已经可见。

第二个条件表明,在更新一个共享数据项之前,进程必须以独占的方式进入临界区,以确保没有其他进程会试图同时更新该数据。

第三个条件表明,如果一个进程想以非独占的方式进入临界区,那么它必须首先检查保护这个临界区的同步变量的所有者,以获得受保护的共享数据的最新副本。

图 6.17 显示了一个人口一致性的实例。在这个实例中,锁操作不是在整个共享数据上进行,而是与每个数据项进行关联。在这种情况下,进程 P1 对 x 执行一次获取操作,改变 x 的值一次,然后它又对 y 执行一次获取操作。进程 P2 对 x 执行一次获取操作,而没有对 y 执行获取操作,所以,对于 x, 它将读出值 a, 而对于 y, 它可能读出值 NIL。因为 P3 进程先对 y 执行了一次获取操作,所以,进程 P1 释放 y 之后,它将读出 y 的值为 b。

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)	
P2:					Acq(Lx)	R(x)a	R(y)NIL
P3:					Acq(Ly)	R(y)b	

图 6.17 对人口一致性有效的事件顺序

关于人口一致性的一个程序设计问题是如何正确地将数据与同步变量关联起来。解决这个问题的一种方式是使用分布式的共享对象。它的工作方式如下所述：每个分布式对象都有一个关联的同步变量，底层的分布式系统在创建一个分布式对象时为其提供一个同步变量，但是这个变量对客户是完全隐藏的。

当客户调用一个分布式对象的方法时，底层系统先对该对象关联的同步变量执行一次获取操作。其效果是将该对象的状态的最新值引入到该对象的客户副本，此状态可能被复制并分布在多台机器上。此时，执行客户的调用，而对象仍然处于锁定状态以防止并发操作。调用完成后，隐含地执行一次释放操作，它将对象解锁以执行待处理的操作。

因此，所有的对分布式共享对象的访问是顺序一致的。幸运的是，客户不用考虑同步变量，这是因为它们完全地由底层分布式系统处理。与此同时，每个对象都自动地受到保护，以防同时执行并发调用。

这种方法已经在 Orca 程序设计语言(Bal 等 1992, Bal 等 1998)中实现，我们将在本章后面详细讨论该语言。本章后面也将讨论另一种相似的方法。在这种方法中，对象以分布式共享存储器的非重叠区的形式出现。每个区有一个底层运行时间系统提供的关联的同步变量。一个区被访问时，运行时间系统负责同步。

6.2.8 一致性模型小结

尽管许多其他(以数据为中心的一致性模型已经被提出来研究，但主要的一致性模型还是上面所讨论的那些模型。它们之间的主要区别是它们限制性的强弱、实现的复杂程度、程序设计的难易程度以及性能的优劣。严格一致性是限制性最强的模型，但是因为在分布式系统中实现这种模型基本上是不可能的，所以它从未应用于实际系统。

线性化是一种较弱的基于同步时钟的一致性模型。它更加容易证明并发程序的正确性，但它很难用于实际程序。从这个角度而言，顺序一致性是一种较好的模型，它的可行性已经得到证明，它深受程序员的喜爱，也确实得到了广泛的应用。但是，它存在性能较差的问题。解决这一问题的方法是放宽一致性模型的要求。图 6.18(a)列出了一些可能的模型，它们大致以限制性逐渐降低的顺序排列。

因果一致性和 FIFO 一致性都在以下方面表现出一定的弱化，即，它们不再要求对哪些操作以哪种顺序出现达成总体一致的观点。不同的进程可以看到不同的操作顺序。这两种模型的区别在于，二者在哪些顺序是允许的，哪些顺序是不允许的方面是不同的。但是两种模型中，都是由程序员负责避免执行只有顺序一致的数据存储才允许的事件顺序。

另一种不同的方式是引入显式的同步变量，例如弱一致性、释放一致性和人口一致性模型就这样做了。图 6.18(b)总结了这三种模型。当一个进程对一个普通的共享数据项

执行操作时,它们不保证其他进程何时看到这一操作。只有在执行一次显式的同步时,数据的改变才被传播。这三种模型的区别在于同步的工作方式,但是在所有模型中,一个进程都可以在一个临界区内在不进行任何数据传输的情况下,执行多个读、写操作。当临界区结束时,最后的结果被传播到其他进程,或为传播做好准备,如果某个进程对那些数据感兴趣,它们将被传播到这个进程。

一致性	描述
严格	所有共享访问按绝对时间排序
线性化	所有进程以相同顺序看到所有的共享访问。而且,访问是根据(非唯一的)全局时间截排序的
顺序	所有进程以相同顺序看到所有的共享访问。访问不是按时间排序的
因果	所有进程以相同的顺序看到因果相关的共享访问
FIFO	所有进程以不同进程提出写操作的顺序相互看到写操作。来自不同进程的写操作可以不必总是以同样的顺序出现

(a)

一致性	描述
弱	只有在执行一次同步后,共享数据才被认为是一致的
释放	退出临界区时,使共享数据成为一致
入口	进入临界区时,使属于一个临界区的共享数据成为一致

(b)

图 6.18 不使用同步操作的一致性模型与使用同步操作的一致性模型

(a) 不使用同步操作的一致性模型; (b) 使用同步操作的一致性模型

简而言之,弱一致性、释放一致性和入口一致性都需要附加的程序设计结构,当按指示的方式使用它们时,它们允许程序员声称一个实际不是顺序一致的数据存储是顺序一致的。原则上,这三种显式地使用同步的模型都应该能够提供最好的性能,但是不同的应用程序可能会导致完全不同的结果。

6.3 以客户为中心的一致性模型

上一节讲述的一致性模型的目的在于从数据存储的角度提供系统级别的-致性。一个重要的假设是并发进程可能同时更新数据存储,而在发生这种并发更新时,必须提供一致性。例如,在基于对象的入口一致性的情况下,数据存储保证向调用对象的进程提供该对象的一个拷贝。这个拷贝可以反映出对该对象所做的所有改变,而这些改变可能是其他进程做出的。在调用的过程中,也可以保证其他的进程不会干扰调用,也就是说,向调用进程提供的是独占性访问。

维持共享数据的顺序一致性的同时能够处理共享数据上的并发操作,对于分布式系统而言是非常重要的。由于性能原因,可能只有当进程使用诸如事务处理和锁一类的同步机制时,系统才能保证顺序一致性。

本节中,我们将讨论一类特殊的分布式数据存储。这里讨论的数据存储的特点是:不会出现同时发生的更新操作,或者当出现同时发生的更新操作时,可以容易地化解它

们。大部分操作是读取数据的操作。我们将讨论的数据存储提供一种很弱的一致性模型,称为最终一致性。通过引入特殊的以客户为中心的一致性模型,我们得出结论:可以通过一种开销相对低的方法隐藏许多不一致性。

6.3.1 最终一致性

进程实际以并发方式操作的程度与所需保证的一致性的程度可能不同。并发操作只以有限的形式出现的实例很多。例如,在许多数据库系统中,大多数进程几乎从不执行更新操作,而只从数据库读取数据,有时只有一个或少数几个进程会执行更新操作。那么,现在的问题是应该以多快的速度进行更新操作,才能使更新对只读进程有效。

另一个实例是世界范围的命名系统,例如 DNS。DNS 名称空间被划分成域,每个域分配一个命名授权机构,由该命名授权机构担当这个域的所有者。只有授权机构才被允许更新名称空间中它所负责的部分。因此,这个系统根本不会出现两个更新相同数据的操作所导致的冲突(即,写-写操作冲突)。惟一需要处理的情况是读-写操作冲突。事实证明,这种情况通常允许以懒惰方式传播更新操作,也就是说,读进程只有在更新操作执行一段时间后才能看到这个更新操作。

还有一个实例是 WWW。实际上,在所有的情况下,Web 页面是由单一的授权机构更新的,例如,Web 站点管理员或页面的实际所有者。通常情况下,不存在需要解决的写-写操作冲突的问题。但是,为了提高效率,通常配置浏览器和 Web 代理在本地高速缓存中保存一份已下载页面,并在用户下次请求该页面时返回高速缓存中的拷贝。两种类型的 Web 高速缓存都存在一个重要问题,那就是它们可能返回过时的 Web 页面。也就是说,与实际可从 Web 服务器获得的页面相比,向客户返回的高速缓存页面是一个较旧版本的页面。事实证明,许多用户认为这种不一致性是可接受的。

这些实例可以被看作是可以容忍相对较高程度的不一致性的(大规模的)分布式复制的数据库的案例。它们的共同之处是,如果在一段很长的时间内没有更新操作,那么所有的副本将逐渐地成为一致的。这种形式的一致性称为最终一致性 (eventual consistency)。

因此,满足最终一致性的数据存储具有以下属性:没有更新操作时,所有副本逐渐成为相互完全相同的拷贝。最终一致性实际上只要求更新操作被保证传播到所有副本。如果假设只有一小部分进程可以执行更新,那么写-写操作冲突就相对比较容易解决了。因此,最终一致性的实现通常开销比较小。本章后面将讨论特殊的实现细节。

只要客户总是访问同一副本,最终一致的数据存储就会工作得很好。但是,当客户访问不同的副本时,问题就出现了。图 6.19 所示的移动用户访问分布式数据库的实例较好地描述了这一问题。

移动用户是以透明的方式连接到一个数据库副本访问数据库的,也就是说,运行在用户便携式计算机上的应用程序不知道它实际在哪个副本上进行操作。假设用户执行几个更新操作,然后断开了与数据库的连接。后来,他要再次访问数据库时,他可能移动到了其他位置,或者他可能使用不同的接入设备访问数据库。此时,用户可能连接到一个与上次连接不同的副本,如图 6.19 所示。然而,如果先前执行的更新操作还没有传播到这

个副本,那么用户就会注意到这个不一致。特别是,他可能期望看到所有先前所做的改变,但是相反,他所看到的可能是好像根本没有做过任何改变一样。

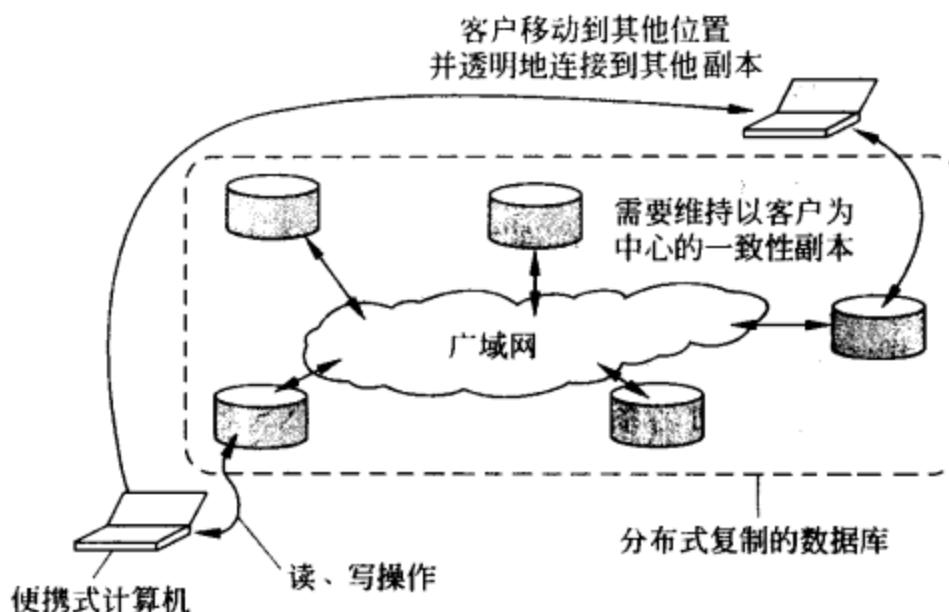


图 6.19 移动用户访问分布式数据库的不同副本的原理

对于最终一致性的数据存储而言,这个实例很有代表性。问题是由于用户有时可能对不同的副本进行操作的事实引起的,引入以客户为中心的一致性可以缓解这一问题。本质上,以客户为中心的一致性为单一的客户提供一致性保证,保证该客户对数据存储的访问的一致性。它并不为不同客户的并发访问提供任何一致性保证。

以客户为中心的一致性模型起源于关于 Bayou 的工作(请参看文献(Terry 等 1994, 和 Terry 等 1998))。Bayou 是为移动计算开发的数据库系统,该系统假设网络连接是不可靠的,而且易于受到多种性能问题的影响。无线网络和跨越范围较广的网络,例如 Internet,都属于这类网络。

Bayou 本质上区分了 4 种不同类型的一致性模型。为了解释这些模型,我们将再次讨论物理上分布于多台机器上的数据存储。当一个进程访问这个数据存储时,它通常连接到其本地(或最近)可用的拷贝上,尽管原则上任何拷贝都可以很好地为其提供数据。所有读、写操作都是对这个本地拷贝进行操作。数据更新最终将被传播到其他拷贝。为了简化问题,我们假设数据项拥有一个关联的所有者,这个所有者就是那个唯一被允许修改该数据项的进程。通过这种方法,我们可以避免写写操作冲突。

我们使用以下的标记法描述以客户为中心的一致性模型。设 $x_i[t]$ 表示 t 时刻本地拷贝 L_i 中数据项 x 的版本。初始化后,对 L_i 中的 x 进行一系列写操作将得到版本 $x_i[t]$ 。我们将这些操作的集合记为 $WS(x_i[t])$ 。如果在随后的 T_2 时刻, $WS(x_i[T_1])$ 中的操作也已经在本地拷贝 L_i 上执行完毕,那么我们记为 $WS(x_i[T_1]; x_i[T_2])$ 。如果从上下文环境可以清楚地判断出操作顺序或时间顺序,那么时间下标将被省略。

6.3.2 单调读

第一个以客户为中心的一致性模型是单调读的一致性模型。如果数据存储满足以下条件,那么称该数据存储提供单调读一致性(monotonic-read consistency):

如果一个进程读取数据项 x 的值,那么该进程对 x 执行的任何后续读操作将总是得到第一次读取的那个值或更新的值。

也就是说,单调读一致性保证,如果一个进程已经在 t 时刻看到 x 的值,那么以后它不再会看到较老版本的 x 的值。

下面,我们讨论一个分布式的电子邮件数据库,作为应用单调读一致性的一个实例。在这个数据库中,每个用户的邮箱可能分布式复制于多台机器上。邮件可以被插入到在任何位置的邮箱中。但是,数据更新是以一种懒惰(即,按需更新)方式传播的。假设一个用户在旧金山读取他的邮件。假定只读取邮件不会影响邮箱,也就是说,消息不会被删除,不会被存储到子目录下,甚至不会被标记为已读。当用户飞到纽约后,再次打开他的邮箱时,单调读一致性可以保证当他在纽约打开他的邮箱时,邮箱中仍然有旧金山的邮箱里的那些消息。

图 6.20 以图形的方式描述了单调读一致性,其中的标记法与以数据为中心的一致性模型所用的标记法相似。在垂直轴上,L1 和 L2 表示数据存储的两个不同的本地拷贝。水平轴表示时间。

L1: WS(x_1)	R(x_1)		L1: WS(x_1)	R(x_1)	
L2: WS($x_1; x_2$)	R(x_2)		L2: WS(x_2)	R(x_2)	WS($x_1; x_2$)

(a)

(b)

图 6.20 单一进程 P 对同一数据存储的两个不同本地拷贝所执行的读操作

(a) 提供单调读一致性的数据存储; (b) 不提供单调读一致性的数据存储

在图 6.20 中,进程 P 先在 L1 对 x 执行了一次读操作,得到值 X_1 (在那一时刻)。该值是在 L1 执行的 $WS(x_1)$ 中的写操作的结果。后来,P 在 L2 对 x 执行了一次读操作,即图中的 $R(x_2)$ 。为了保证单调读一致性, $WS(x_1)$ 中的所有操作都应该在执行第二个读操作前传播到 L2。也就是说,我们需要确保 $WS(x_1)$ 是 $WS(x_2)$ 的一部分,记为 $WS(x_1; x_2)$ 。

与之相对照,图 6.20(b) 表示不保证单调读一致性的情况。进程 P 在 L1 读取 x_1 后,又在 L2 执行 $R(x_2)$ 操作。然而,L2 只执行了 $WS(x_2)$ 中的写操作。此时,不能保证集合 $WS(x_2)$ 同样包含 $WS(x_1)$ 中的所有操作。

6.3.3 单调写

在很多情况下,写操作以正确的顺序传播到数据存储的所有拷贝是非常重要的。这种性质被描述为单调写一致性。单调写一致性(monotonic-write consistency)的数据存储应满足以下条件:

一个进程对数据项 x 执行的写操作必须在该进程对 x 执行任何后续写操作之前完成。

因而,完成一个写操作意味着不管后续操作的启动位置,执行这个后续操作的拷贝都能反映出同一进程先前执行的写操作的结果。也就是说,在数据项 x 的拷贝上执行的写操作只有在该拷贝已经通过任何先前的写操作进行更新之后才能被执行,而这些先前执行的写操作可能发生在 x 的其他拷贝上。

注意,单调写一致性与以数据为中心的 FIFO 一致性相似。FIFO 一致性的本质是,同一进程执行的写操作必须在任何地方以正确的顺序执行。这一顺序限制也适用于单调写一致性,只是我们这里考虑的是仅为单一进程维持的一致性,而不是为许多并发进程维持的一致性。

每个写操作完全覆盖 x 的当前值时,没有必要将 x 的拷贝更新为最新值。然而,写操作通常只修改数据项的部分状态。让我们来考虑一个软件库。在很多情况下,更新这种库是通过替换一个或多个函数来实现下一版本的。单调写一致性保证,如果在库的一个拷贝上执行数据更新,那么前面(在其他拷贝上)执行的所有数据更新都将首先执行。此后,所得的库将确实是最新版本的库,并将包括所有以前各个版本的更新。

图 6.21 是单调写一致性的例子。图 6.21(a)中,进程 P 在本地拷贝 L1 对 x 执行一次写操作,记为操作 $W(x_1)$ 。然后,进程 P 对 x 再执行一次写操作,但是这次是在 L2 处执行,如图中的 $W(x_2)$ 。为了保证单调写一致性,先前在 L1 执行的写操作必须已经传播到 L2。这就解释了图中 L2 处出现操作 $W(x_1)$,以及它出现在 $W(x_2)$ 之前的原因。

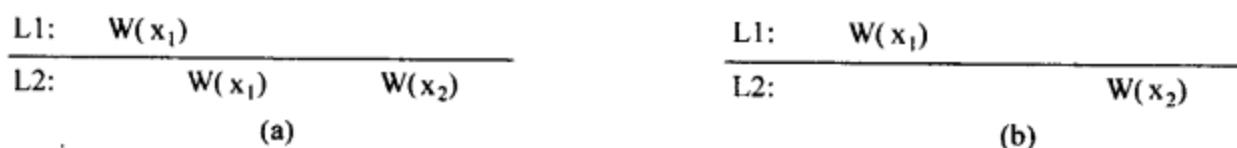


图 6.21 单一进程 P 对同一数据存储的两个不同本地拷贝所执行的写操作

(a) 提供单调写一致性的数据存储; (b) 不提供单调写一致性的数据存储

与之相对照,图 6.21(b)表示不保证单调写一致性的情况。与图 6.21(a)相比,图中没有向拷贝 L2 传播的 $W(x_1)$ 。也就是说,它不能保证在 x 的拷贝上执行第二次写操作时, x 的拷贝的值与执行完 $W(x_1)$ 时的值相同或比之更新。

注意,在单调写一致性的定义中,同一进程的写操作的执行顺序与这些操作的启动顺序相同。一个相对较弱的单调写形式是写操作的结果仅当所有先前的写操作也都已经执行完毕后才会被看到,但是它们的执行顺序可能与这些操作的最初启动顺序不同。这种一致性可应用于写操作是可交换的情况,此时,写操作的顺序的确是不必要的。详述请参阅文献(Terry 等 1994)。

6.3.4 写后读

下面介绍一种与单调写一致性有密切关系的以客户为中心的一致性模型。如果数据存储满足以下条件,那么称该数据存储提供写后读一致性(read-your-writes consistency):

一个进程对数据项 x 执行一次写操作的结果总是会被该进程对 x 执行的后续读操作看见。

也就是说,一个写操作总是在同一进程执行的后续读操作之前完成,而不管这个后续的读操作发生在什么位置。

在更新 Web 的 HTML 页面后浏览更新效果时,经常遇到缺乏写后读一致性的情况。更新操作通常是通过标准的编辑器或文字处理器实现的,它们将新版本页面存储在 Web

服务器所共享的文件系统上。用户的 Web 浏览器可能在它从本地 Web 服务器请求这个文件之后,再访问这个文件。但是,一旦文件已经被访问过,服务器或浏览器通常都会为以后的访问缓存一个本地拷贝。因此,当 Web 页面被更新时,如果浏览器或服务器返回高速缓存的拷贝而不返回原文件的话,用户将看不到更新的结果。写后读一致性可以保证,如果编辑器和浏览器被集成到一个单一的程序中,那么页面更新后,高速缓存是无效的,所以将获取和显示被更新的文件。

更新密码时也会出现相同的情况。例如,Web 上的电子图书馆通常要求输入账号和相应的密码。但是,密码的修改可能需要一些时间才能生效,这可能导致用户有几分钟的时间不能进入图书馆。这种延迟可能是由于系统使用一个单独的服务器管理密码,而加密的密码传播到图书馆的多个服务器可能需要一定时间造成的。在 Grapevine 中已经发现了这一问题,Grapevine 是最早使用最终一致性的分布式系统之一。同样,实现写后读一致性可以解决这一问题(Birrell 等 1982)。

图 6.22(a)表示了一个提供写后读一致性的数据存储。注意,图 6.22(a)与图 6.20(a)非常相像,只是这里的一致性是通过进程 P 执行的最后一次写操作确定的,而不是通过进程 P 的最后一次读操作确定的。



图 6.22 两种数据存储

(a) 提供写后读一致性的数据存储; (b) 不提供写后读一致性的数据存储

图 6.22(a)中,进程 P 执行了一次写操作 $W(x_1)$,然后在一个不同的本地拷贝处执行了一次读操作。写后读一致性保证,写操作的结果可以被所有后续的读操作看到。这一过程表示为 $WS(x_1; x_2)$,它表示 $W(x_1)$ 是 $WS(x_2)$ 的一部分。与之相对照,在图 6.22(b)中, $W(x_1)$ 不在 $WS(x_2)$ 之中,这意味着进程 P 执行的前一个写操作的结果还没有传播到。

6.3.5 读后写

最后一种以客户为中心的一致性模型是这样的模型,即更新是作为前一个读操作的结果传播的。如果数据存储满足以下条件,那么称该数据存储提供读后写一致性(writes-follow-reads consistency)。

同一个进程对数据项 x 执行的读操作之后的写操作,保证发生在与 x 读取值相同或比之更新的值上。

也就是说,进程对数据项 x 所执行的任何后续的写操作都会在 x 的拷贝上执行,而该拷贝是用该进程最近读取的值更新的。

读后写一致性可以用于保证,网络新闻组的用户只有在已经看到原文章之后才能看到它的回应文章(Terry 等 1994)。为了理解这个问题,我们假设一个用户先读了一篇文章 A。然后,他回应了一篇文章 B。通过要求保证读后写一致性,B 被写入新闻组的任何拷贝之前,A 也必须已经被写入那个拷贝。注意,只读取文章的用户不需要任何特定的

以客户为中心的一致性模型。读后写一致性保证，当且仅当原文章存储在某个本地拷贝上时，该文章的回应文章才被存储在这个本地拷贝上。

图 6.23 表示了这种一致性。图 6.23(a)中，一个进程在本地拷贝 L1 处读取 x 。写入刚才所读取值的写操作也出现在 L2 处的写操作集合中。稍后，同一进程在 L2 处执行了一次写操作。注意，L2 处的其他进程也看到了那些写操作。与之相对，图 6.23(b)中的数据存储无法保证 L2 处执行的操作是在与 L1 处所读的那个拷贝一致的拷贝上执行的。

L1:	WS(x_1)	R(x_1)	L1:	WS(x_1)	R(x_1)
L2:	WS(x_1, x_2)	W(x_2)	L2:	WS(x_2)	W(x_2)

(a)

(b)

图 6.23 两种数据存储

(a) 提供读后写一致性的数据存储；(b) 不提供读后写一致性的数据存储

6.3.6 实现

如果忽略性能问题，那么实现以客户为中心的一致性相对比较直接。下面，我们首先描述这样一个实现，然后描述一个更现实的实现。

1. 一种简化实现

在以客户为中心的一致性的一种简化实现中，每个写操作都被分配一个全局惟一的标识符。该标识符是第一次接受操作的服务器为其分配的。我们也称该操作是在那台服务器上启动的（注意，生成全局惟一标识符可以作为一种本地操作实现，请参看第 4 章最后的习题）。这样，对于每个客户，我们跟踪写标识符的两个集合。一个是客户的读集合，它由与客户所执行的读操作相关的写标识符组成；一个是客户的写集合，它由客户所执行的写操作的标识符组成。

单调读一致性以如下方式实现。当一个客户在一台服务器上执行一个读操作时，这台服务器获得客户的读集合，检查所有标识的写操作是否已经在本地执行（这个集合的大小可能引入性能问题，下面将讨论这一问题的解决方案）。如果没有执行，那么它就联系其他服务器以确保执行读操作之前将本地拷贝更新。另一种方法是将读操作转发到已经执行了这些写操作的服务器那里执行。读操作执行后，在所选择的服务器上执行的写操作以及与读操作相关的写操作会被加入客户的读集合。

注意，这里应该可以确定在读集合中标识的写操作所发生的确切位置。例如，写操作标识符可能包括启动该操作的服务器的标识符。例如，要求那台服务器将写操作记入日志以便在其他服务器上重新执行这个写操作。另外，写操作应该以启动它们的顺序执行。让客户生成一个全局惟一的序列号，例如 Lamport 时间戳，并将这个序列号包含于写操作标识符之中，可以实现写操作的排序。如果每个数据项只能被它的所有者修改，那么后者就可以提供这个序列号。

单调写一致性的实现方式与单调读一致性的实现方式类似。每当一个客户在服务器上启动一个新的写操作，客户的写集合都会转交到这台服务器（同样，面对性能需求，这个

写集合的大小可能过大。下面将讨论另一种可选择的解决方案)。然后,这台服务器确保先执行被标识的写操作,并按正确的顺序执行这些写操作。执行新操作之后,该操作的写标识符将被加入写集合。注意,使用客户的写集合更新当前服务器可能会造成客户的响应时间的大幅增加。

同样,写后读一致性要求执行读操作的服务器已经看到客户的写集合中的所有写操作。执行读操作之前,可以简单地从其他服务器获取这些操作,尽管这可能导致较差的响应时间。或者,客户端软件可以查找已经执行了已标识写操作的服务器,写操作在客户的写集合中标识。

最后,读后写一致性可以通过这一方法实现,即先使用在客户的读集合中标识了的写操作更新所选择服务器,然后将写操作的标识符以及读集合中的标识符(这些操作现在已经成为与刚刚执行的写操作相关的操作)加入到写集合中。

2. 提高效率

很容易看出,与每个客户关联的读集合和写集合可能变得非常大。为了维持这些集合的可控制性,客户的读操作和写操作根据会话分成组。通常,一个会话与一个应用程序关联:应用程序启动时,打开会话,应用程序结束时,关闭会话。但是,会话也可能与暂时结束的应用程序关联,例如用于电子邮件程序的用户代理就是这样的。每当客户结束会话,其关联的集合将被完全清空。当然,如果一个客户打开一个从不关闭的会话时,与该客户所关联的读集合和写集合仍可能变得非常大。

简化实现的主要问题在于如何表示读集合和写集合。每个集合由一些写操作的标识符组成。每当客户向服务器转发一个读请求或写请求时,标识符的集合也将被转交给这台服务器,以查看该服务器是否已经执行了所有与该请求相关的写操作。

按照如下方式使用向量时间戳可以更高效地表示这一信息。首先,每当服务器接受一个新的写操作 W 时,它首先按上面所述方法为该操作分配一个全局惟一的标识符 WID 和一个时间戳 $ts(WID)$ 。随后在该服务器上执行的写操作将被分配一个更高值的时间戳。每台服务器 S_i 都维护一个向量时间戳 $RCVD(i)$,其中 $RCVD(i)[j]$ 的值等于 S_i 已接受并执行的、由服务器 S_i 启动的最后一个写操作的时间戳。

每当客户发出在一个特定的服务器上执行读或写操作的请求时,该服务器都返回该操作当前的时间戳和该操作的结果。然后,用向量时间戳表示读集合和写集合。对于任何这样的集合 A ,我们构造一个向量时间戳 $VT(A)$ 来高效地表示这些集合,其中 $VT(A)[i]$ 被设置为集合 A 中所有由服务器 S_i 启动的操作的时间戳的最大值。

两个写标识符集合 A 和 B 的并集用向量时间戳 $VT(A+B)$ 表示,其中, $VT(A+B)[i]$ 等于 $\max\{Vr(A)[i], VT(B)[i]\}$ 。另外,为了检查标识符集合 A 是否包含于另一个集合 B ,我们只需检查对于每个下标 i ,是否有 $VT(A)[i] < VT(B)[i]$ 。

当服务器将其当前的时间戳传回给客户时,该客户根据所执行的操作调整它自己的读集合或写集合的向量时间戳。我们来考虑一下服务器 S_i 向客户返回向量时间戳 $RCVD(i)$ 的单调读一致性情况。如果客户的读集合的向量时间戳是 $VT(Rset)$,那么对于每个 j , $VT(Rset)[j]$ 被设置为 $\max\{VT(Rset)[j], RCVD(i)[j]\}$ 。因而,客户的读集

合将反映出它所看到的最新的读操作。相应的向量时间戳可能随着该客户提出的下一个读操作被传送到另一个不同的服务器。

6.4 分发协议

到目前为止,我们的注意力只集中在不同的一致性模型上,几乎没有接触到实现问题。我们将在本节讨论数据传播的不同方法,即将数据更新发送给各个副本的方法,这些方法独立于它们所支持的一致性模型。我们将在下一节讨论特殊的一致性协议。

6.4.1 副本放置

分布式数据存储的一个主要设计问题是决定数据存储的拷贝将被放置的位置、时间,以及谁来放置这些拷贝(请参阅文献 Kermarrec 等 1998)。数据存储的拷贝可以分为三种类型,图 6.24 表示从逻辑上组织这三种类型的拷贝。

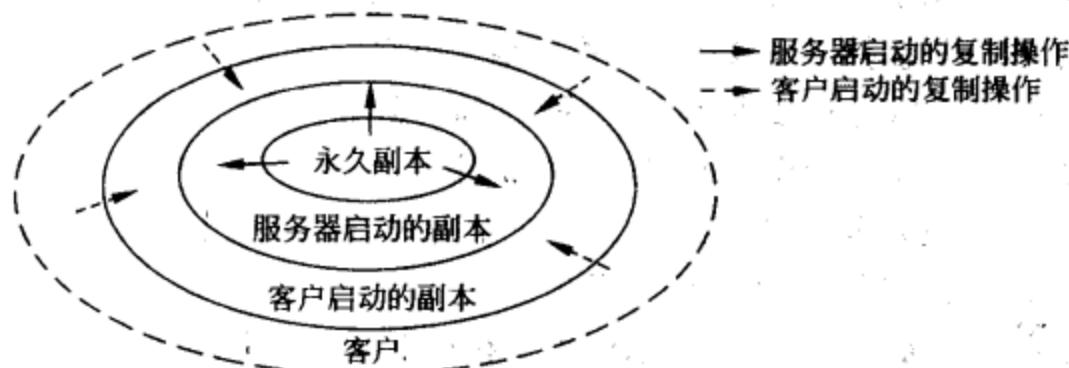


图 6.24 将数据存储的不同类型的拷贝逻辑地组织成三个同心环

1. 永久副本

永久副本可以被视为构成分布式数据存储的副本的初始集合。很多情况下,永久副本的数量很小。以 Web 站点为例,一个 Web 站点的分布通常采用以下两种形式之一。第一种形式的分布是在单一局域网中的有限数量的服务器上复制构成站点的文件。每当接收到请求,就将该请求转发到其中一台服务器,例如,使用循环提示器策略来进行转发。请参阅文献(Chawathe 和 Brewer 1998)。

分布式 Web 站点的第二种形式是一种被称为镜像(mirroring)的形式。在这种情况下,Web 站点被复制到有限数量的服务器上,这些服务器被称为镜像站点(mirror site),它们在地理上散布于 Internet 之中。在很多情况下,客户只是从提供给他们的镜像站点列表中选择一个镜像站点。被镜像的站点与基于群集的 Web 站点有共同之处,即站点只有少量的副本,而这些副本或多或少是使用静态配置的。

相似的静态组织方式也出现在分布式数据库中(Oszu 和 Valduriez 1999)。同样,数据库可以被分布地复制在很多服务器上,这些服务器一起形成工作站群集(COW)。通常将这种体系结构称为无共享体系结构(shared-nothing architecture),该体系结构强调处理器既不共享磁盘,又不共享内存。另外,数据库也可能分布地被复制在地理上分散的几

台服务器上。在联合数据库(Sheth 和 Larson 1990)中通常部署这种体系结构。

2. 服务器启动的副本

与永久副本相对,服务器启动的副本是为提高性能而存在的数据存储的拷贝,该拷贝是在初始化数据存储的所有者时创建的。例如,位于纽约的一个 Web 服务器就是这样的。一般,这台服务器可以相当方便地处理输入的请求,但是可能发生以下情况:它突然在几天内接收到来自远离服务器的意外位置的大量请求(最近的 Web 历史中已经在一些场合出现过这种突发大量请求情况)。在这种情况下,在产生请求的地区安装一些暂时副本可能是值得的。这些副本也称为推高速缓存(push cache)(Gwertzman 和 Seltzer 1995)。

最近,Web 托管服务正提出动态放置副本的问题。这些服务在本质上提供了一个相对静态的散布于 Internet 的服务器的集合。这些服务器可以维护属于第三方的 Web 文件,并提供对这些文件访问。为了提供优化的功能,这些托管服务可以动态地把文件复制到需要这些文件以提高性能的服务器,也就是接近发出请求的客户的服务器那里。

服务器启动的副本的一个问题是决定创建或删除副本的确切位置和时间。文献(Rabinovich 等 1999)描述了一种 Web 托管服务中实现文件动态复制的方法。设计该算法的目的是为了支持 Web 页面可在哪些原因下假设更新的个数与读请求的个数相比相对较少。该算法使用文件作为数据单元,其工作方式如下所述。

动态复制的算法需要考虑两个问题。第一,复制可能是为了减轻一台服务器的负载而进行的;第二,一台服务器上的指定文件可能被转移或复制到对这些文件提出很多请求的客户附近的服务器。下面,我们只集中考虑第二个问题。我们也不考虑一些细节问题,关于这些细节问题请参阅文献(Rabinovich 等 1999)。

每台服务器跟踪每个文件的访问计数以及提出这些访问请求的位置。特别是,假设对于一个给定的客户 C,每台服务器都可以确定 Web 托管服务中的哪台服务器最接近 C(例如,可以从路由数据库获得这一信息)。如果客户端 C_1 和客户端 C_2 同时最接近服务器 P,那么所有来自 C_1 和 C_2 对服务器 Q 上的文件 F 的访问请求一起被 Q 记入单一的访问计数 $cnt_Q(P, F)$ 之中。这一情形如图 6.25 所示。

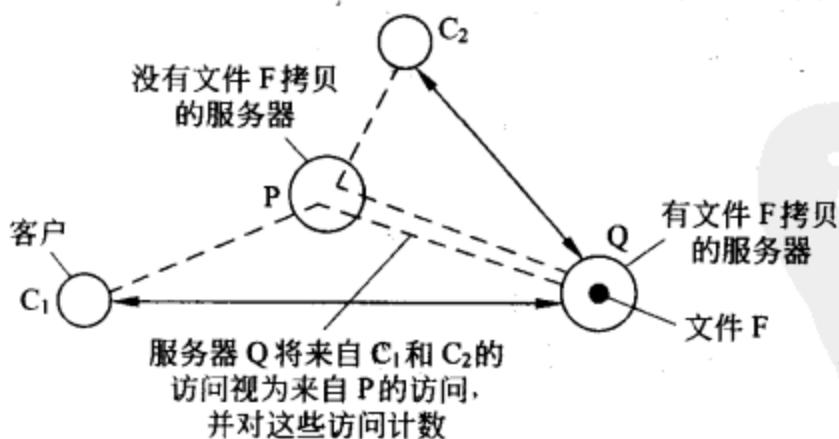


图 6.25 对来自不同客户的访问请求计数

当对服务器 S 上的指定文件 F 的请求数量下降到删除阈值 $del(S, F)$ 之下时,服务器 S 可以删除该文件。因而,该文件的副本的数目减少了,这可能导致其他服务器的工作负

载增加。可采用特殊的方法来保证每个文件至少仍存在一个拷贝。

复制阈值 $rep(S, F)$ 指出对指定文件的请求数量过高,以至于值得将该文件复制到另一台服务器上。通常所选择的复制阈值比删除阈值高。如果请求的数量位于删除阈值和复制阈值之间,那么只允许转移该文件。也就是说,在这种情况下,至少保持该文件的副本的个数不变,这一点是十分重要的。

当服务器 Q 决定重新评估它存储的文件的位置时,它检查每个文件的访问计数。如果服务器 Q 上文件 F 的访问请求的总数低于删除阈值 $del(Q, F)$,那么它将删除 F,除非 F 是最后一个拷贝。此外,如果对于某台服务器 P, $cnt_Q(P, F)$ 超过 Q 上 F 的请求总数的半数以上,服务器 Q 请求服务器 P 接管 F 的拷贝。也就是说,服务器 Q 将试图把 F 转移到 P。

向服务器 P 转移文件 F 不可能总能成功。例如,因为 P 的负载总是很重或者 P 没有磁盘空间就会造成文件转移不成功。在这种情况下, Q 将试图在其他服务器上复制 F。当然,只有对 Q 上的 F 的访问请求的总数超过复制阈值 $rep(Q, F)$ 时才会发生复制。服务器从 Web 托管服务中最远的服务器开始,检查其中所有的服务器。如果对于某台服务器 R, $cnt_Q(R, F)$ 超过对 Q 上的 F 的所有请求,并达到某个比例, Q 就会试图把 F 复制到 R。

服务器启动的复制正在逐渐地流行起来,特别是在上面介绍的 Web 托管服务的环境下。注意,只要可以保证每个数据项至少由一台服务器托管,那么只使用服务器启动的复制而不使用任何永久副本就足够了。然而,永久副本仍常常用作备份工具或用作允许被修改以保证一致性的惟一副本,而服务器启动的副本则被用于在客户附近放置只读拷贝。

3. 客户启动的副本

一类重要的副本是在客户初始化时创建的副本。普遍将客户启动的副本称为客户高速缓存(Client cache)。其实,高速缓存是一种本地存储工具,客户使用它暂时存储刚刚请求的数据的拷贝。在原则上,高速缓存的管理是完全由客户负责的。客户从中获取数据的数据存储不负责保持高速缓存数据的一致。然而,如同我们所看到的,在很多情况下,客户可能要依靠数据存储通知它被缓存的数据什么时候过时。

客户高速缓存只用于改善数据的访问时间。通常,当一个客户想要访问某些数据时,它连接该数据存储的最近的拷贝,从该拷贝获得它想读取的数据,或者将它刚刚修改的数据写入该拷贝。当大部分操作仅是读取数据时,客户在附近的高速缓存中存储所请求的数据可以提高性能。这个高速缓存可以位于客户所在的机器中,也可以位于客户所在局域网中的另一台单独的机器中。下一次需要读取该数据时,客户可以简单地从这个本地高速缓存中获得数据。只要读取所获得的数据时,数据没有被改变,这种模式会工作得很好。

例如,通常数据将在高速缓存中保存一段有限的时间,这样做是为防止使用完全过时的数据,或只是为其他数据腾出空间。当被请求的数据可以从本地拷贝获得时,我们称其为高速缓存命中(cache hit)。为了提高高速缓存命中数量,可以让多个客户共享高速缓存。其基本假设是来自客户 C₁ 的数据请求可能也对来自客户 C₂ 附近的其他客户的请求

有用。

这一假设的正确性在很大程度上取决于数据存储的类型。例如，传统文件系统几乎不共享文件，参阅文献(Muntz 和 Honeyman 1992, Blaze 1993)，这使得共享高速缓存毫无意义。相反，Web 中的共享高速缓存已被证明非常有用，尽管它们对性能的影响正在逐渐下降。这是因为 Web 页面的总数不断增加，不同客户访问相同的页面的情况不常发生(Barford 等 1999)。

客户高速缓存的放置也相对简单：通常将高速缓存放置在客户所在的机器上，或者将其放置在客户所在的局域网上由各个客户共享的机器上。然而，在某些情况下，系统管理员会引入更高级的高速缓存。他们在一些部门或组织之间放置共享高速缓存，或者甚至在整个地区，例如省或国家内放置共享高速缓存。

另一种方法是在广域网的几个特定点放置高速缓存服务器，让客户查找距离最近的服务器。当客户查找到最近的服务器后，它请求这台服务器保留它先前从其他服务器获得的数据的拷贝(Noble 等 1999)。我们将在本章后面讨论一致性协议时再继续讨论缓存问题。

6.4.2 更新传播

通常，对分布的并且是复制的数据存储上进行的更新操作是在客户上启动的，然后它们被转发到数据存储的一个拷贝。更新应该从这个拷贝传播到其他拷贝，同时保证拷贝的一致性。下面，我们将讨论传播更新所需考虑的一些不同的设计问题。

1. 状态与操作

一个重要的设计问题是将要实际传播哪些信息。基本上有三种可能：

- (1) 只传播更新的通知；
- (2) 把数据从一个拷贝传送到另一个拷贝；
- (3) 把更新操作传播到其他拷贝。

无效化协议(invalidation protocol)使用传播通知这种方式。无效化协议通知其他拷贝已发生了更新操作，这些拷贝所包含的数据不再有效。无效化消息可以指定数据存储的哪些部分被更新了，那么实际上只有部分拷贝是无效的。重要的问题是只传播一个通知而不传播别的消息。每当请求在无效的拷贝上执行操作时，一般需要先根据数据存储支持的特定的一致性模型更新那个拷贝。

无效化协议的主要优点是它几乎不占用网络带宽。所需传送的惟一信息是关于哪些数据不再有效的说明。当更新操作与读操作相比较多时，也就是说，读对写的比率相对很小时，这种协议通常工作得非常好。

例如，在一个数据存储中，更新是通过向所有副本发送被修改的数据传播的。如果被修改的数据很多，并且更新操作与读操作相比发生得更频繁，那么我们可能遇到以下情况：接连发生两个更新操作，而这两个更新操作之间没有读操作。因而，第一次向所有副本传播的更新实际上是无用的，因为它将被第二个更新操作的结果覆盖。相反，在这种情况下，发送数据已被修改的通知可能更加高效。

在多个副本间传送被修改的数据是第二种选择，在读对写比率相对较高时，这种方法十分有用。在这种情况下，更新是有效的可能性较高，因为从某种意义上来说，被修改的数据会在下一个更新操作发生前被读取。不传播被修改的数据，而将这些修改记入日志，然后仅传送那些日志以节约带宽也是可以的。另外，更新经常是通过将多个修改压缩到一个消息中的方式被组合传送的，从而减少通信开销。

第三种方法根本不传送任何数据修改，而告诉每个副本它应该执行的更新操作，这种方法也被称为主动复制(active replication)。它假设每个副本由一个进程代表，该进程能够通过执行操作来“主动地”保持它所关联的数据为最新的数据(Schneider 1990)。主动复制的主要优点是假设一个操作所关联的参数相对较少，传播更新的带宽代价通常可以达到最小。另一方面，每个副本可以获得更多的处理能力，在操作相对复杂的情况下更是如此。

2. 拉协议与推协议

另一个设计问题是，更新是拉式的还是推式的。基于推式的方法(push-based approach)也被称为基于服务器的协议(server-based protocol)。在这种方法中，甚至不需要其他副本请求更新，这些更新就被传播到那些副本那里。永久副本和服务器启动的副本之间通常使用基于推式的方法。基于服务器的协议应用于多个副本常常需要维持相对较高程度的一致性的时候，也就是说，当副本需要保持完全相同的时候。

需要保持较高程度的一致性是基于这样一个事实：通常有许多客户共享永久副本和服务器启动的副本，以及较大的共享的高速缓存，而这些客户轮流地执行操作，主要执行读操作。因而，每个副本上读与更新之比率相对较高。在这些情况下，基于推式的协议是高效的，因为每个被推的更新可以对一个或更多读程序有用。另外，基于推式的协议使一致的数据在被请求时立即有效。

相反，在基于拉式的方法(pull-based approach)中，一台服务器或客户请求其他服务器向它发送该服务器此时持有的任何更新。基于拉式的协议，也被称为基于客户的协议(client-based protocol)，通常被用于客户高速缓存。例如，Web 高速缓存使用的一种通用策略是先检查被缓存的数据项是否仍然是最新的。当高速缓存接收到对数据项的请求，而该数据项仍对本地有效的时候，高速缓存查看原始的 Web 服务器，以确定那些数据项自从被缓存后是否被修改过。如果那些数据项被修改过，那么被修改的数据首先被传送到高速缓存，然后才被返回给请求数据的客户。如果那些数据项没有被修改过，那么就向客户返回被缓存的数据。也就是说，客户轮询服务器以查看是否需要更新高速缓存。

在读与更新之比率相对较低时，基于拉式的方法是高效的。非共享的客户高速缓存通常属于这种情况，此时客户高速缓存只有一个客户。然而，即使在许多客户共享一个高速缓存，在被缓存的数据项很少被共享时，也可证明基于拉式的方法是高效的。与基于推式的方法相比，基于拉式的策略的主要缺点是在高速缓存没有命中时，响应时间会增大。

比较基于推式的方法和基于拉式的方法时，需要做出一些权衡，如图 6.26 所示。简单起见，我们以一个客户-服务器系统为例，该系统由一个单一的、非分布式的服务器和一些客户进程构成，每个客户进程都有它们自己的高速缓存。

问 题	基于推式	基于拉式
服务器的状态	客户副本和高速缓存的列表	无
发送的消息	更新(以及以后可能获取的更新)	轮询和更新
客户响应时间	立即(或获取更新的时间)	获取更新的时间

图 6.26 在多客户、单一服务器的系统中,基于推式的协议和基于拉式的协议的比较

基于推式的协议中的一个重要问题是服务器需要跟踪所有的客户高速缓存。状态较多的服务器不但通常缺少容错能力(如第 3 章所述),而且跟踪所有的客户高速缓存也可能在服务器端引入相当大的额外开销。例如,在基于推式的方法中,Web 服务器可能需要跟踪数以万计的客户高速缓存。每次 Web 页面被更新时,服务器将需要遍历持有该页面的拷贝的客户高速缓存的列表,然后向这些客户高速缓存传播更新。但是更糟的是,如果客户由于缺少空间而清除了一个页面,那么它必须通知服务器,这又导致了更多的通信。

在这两种方法中,需要在客户和服务器之间发送的消息也有所不同。在基于推式的方法中,惟一的通信是服务器向每个客户发送更新。当更新实际上仅是无效化消息时,客户需要进行其他通信以获取被修改的数据。在基于拉式的方法中,客户必须轮询服务器,并在必要时获取被修改的数据。

最后,在两种方法中客户响应时间也是不同的。当服务器将被修改的数据推入客户时,客户的响应时间显然是 0。当推入无效化消息时,响应时间与基于拉式的方法中的响应时间相同,该响应时间由客户从服务器获取被修改的数据的时间决定。

这些代价权衡导致出现了一种的更新传播的混合形式——基于租用的更新传播。租用(lease)是服务器所作的承诺,它将在指定的时间内把更新推给客户。当租用到期时,客户被迫轮询服务器以实现更新,并在必要时拉出被修改的数据。另一种方法是在前一个租用到期时,客户请求一个新的租用以实现更新的推入。

租用最初是由 Gray 和 Cheriton(1989)提出的。这是他们为基于推式的策略和基于拉式的策略之间的动态转换提供的一种便利机制。Duvvuri 等(2000)描述了一种灵活的租用系统,该系统允许租用期限根据不同租用标准动态调整。他们对以下三种类型的租用进行区分(注意,在所有的情况下,只要租用没有到期,更新都是由服务器负责推入的)。

首先,是一种基于数据项“年龄”的租用,数据项的“年龄”取决于该数据项最后一次被修改后的延续时间。其基本假设是长时间内未被修改的数据可能会在将来的一段时间内仍不会被修改。基于 Web 的数据已表明这一假设是合理的。与所有租用具有相同过期时间的方式相比,为预期保持不变的数据项授予一个长期的租用可以使更新消息的个数大大地减少。

另一种租用是基于特定客户请求更新高速缓存副本的频率的租用。使用基于更新频率的租用时,服务器将为需要经常刷新其高速缓存的客户分配一个长期的租用,而为只偶尔请求特定数据项的客户分配一个使用该数据项的短期租用。该策略的效果是服务器基本上只跟踪其数据受欢迎的那些客户,此外,那些客户端也被提供了较高程度的一致性。

最后一种租用是基于服务器的状态空间开销的租用。当服务器意识到它会逐渐过载

时,它会降低分配给客户的新租用的使用期限。这一策略的效果是服务器需要跟踪的客户变少,这是因为租用到期很快。也就是说,服务器动态地转换到一个状态较少的操作模式,从而减轻了服务器负载以使其可以更高效地处理请求。

3. 单播与多播

与推式更新或拉式更新相关的一个问题是决定应使用单播还是使用多播。在单播通信中,当作为数据存储的一部分的服务器向其他 N 台服务器发送其更新时,它通过发送 N 个单独的消息实现,即向每台服务器发送一个消息。使用多播通信时,底层的网络负责向多个接收者高效地发送一个消息。

在很多情况下,使用可用的多播工具是比较廉价的。一种极端的情况是所有副本都位于同一个局域网,此时,硬件广播是可用的。在这种情况下,广播或多播一个消息的开销并不比一个单一的点对点消息传送开销大。但单播更新的效率会较低。

与基于推式的方法结合时,多播常常可以高效地传播更新。在这种情况下,一个决定向其他一些服务器推出更新的服务器仅使用一个单一的多播组来发送它的更新。相反,使用基于拉式的方法时,通常只有一个单一的客户或服务器请求更新其拷贝。在这种情况下,单播可能是最高效的方法。

6.4.3 epidemic 协议

我们已经提到,对于许多数据存储,只为其提供最终一致性就足够了。也就是说,在没有更新时,它只需要保证所有副本最终是一致的。在最终一致性的数据存储中,更新传播通常使用一类称为 epidemic 协议的算法实现。epidemic 算法不解决任何更新冲突,更新冲突的问题由单独的解决方法解决。它所关心的只是尽可能使用最少的消息将更新传播到所有副本。为了达到这一目的,通常将多个更新压缩为一个单一的消息,然后在两台服务器之间交换这些更新。epidemic 算法构成了前面描述的 Bayou 系统的基础,它的各种更新传播方案在文献(Petersen 等 1997)中有所描述。

为了解释这些算法的一般原理,我们假定一个特定的数据项的所有更新都是在一个单一的服务器上启动的。在这种情况下,我们仅需避免写写操作冲突即可。下面的讲解基于 Demers 等 (1987) 关于 epidemic 算法的经典论文。

1. 更新传播模型

顾名思义,epidemic 算法基于 epidemics 理论,epidemics 理论是研究传染病的扩散的。复制的数据存储扩散的不是疾病,而是更新。关于数据存储的 epidemics 研究也旨在达到完全不同的目标:尽管健康组织尽其最大努力防治疾病在人群中传染,但是分布式数据存储的 epidemic 算法的设计者却试图尽可能快地用新的更新“感染”所有的副本。

作为分布式数据存储的一部分的服务器,如果它持有它愿意向其他服务器散布的更新,使用 epidemics 中的术语,则称该服务器是有传染性(infective)的。尚未更新的服务器被称为易感的(susceptible)。最后,已更新的服务器如果不愿意或不能扩散其更新,则称为被隔离的(removed)。

一个流行的传播模型是反熵(anti-entropy)传播模型。在这个模型中,服务器 P 随机选取另一台服务器 Q,然后与 Q 交换更新。交换更新的方法有以下三种:

- (1) P 只把它自己的更新推入 Q;
- (2) P 只从 Q 拉出新的更新;
- (3) P 和 Q 相互发送更新(即,推拉方法)。

在快速扩散更新中,只有推入更新被证明是较差的选择。可以按如下方式直观地理解这一点。首先,值得注意的是在纯粹的基于推式的方法中,只有有传染性的服务器能够传播更新。但是,如果很多服务器是有传染性的,那么每台服务器选择一个易感的服务器的可能性相对较低。因而,一个特定的服务器很可能在一段较长时间内仍为易感的,只因为它没有被任何有传染性的服务器选中。

相反,基于拉式的方法在很多服务器有传染性的时候工作得更好。在这种情况下,扩散更新本质上是由易感服务器触发的。这样的服务器接触到一个有传染性的服务器,随后提取更新并变成同样有传染性的服务器的机会很大。

可以证明,如果只有一个单一的服务器是有传染性的,使用哪种形式的反熵模型都可使更新最终被扩散到所有服务器上。但是,为了保证更新被快速地扩散出去,使用额外的机制使一些服务器马上或多或少有些传染性是合理的。具有代表性的一种方法是直接将更新推入到一些服务器。

这种方法的一种特殊变体称为 rumor spreading,或简单地称为 gossiping。它的工作方式如下。如果服务器 P 刚刚因为数据项 x 而被更新,那么它联系任意一个其他服务器 Q,并试图将更新推入到 Q。但是,Q 可能已经被其他服务器更新了。在这种情况下,P 会失去继续扩散更新的兴趣,这种情况的可能性是 $1/k$ 。也就是说,它变成被隔离的。

gossiping 完全类似于现实生活的情况。当 Bob 有些最新消息要散布出去的时候,他可能给他的朋友 Alice 打电话,告诉她所有的最新消息。Alice 也会像 Bob 一样,十分兴奋地将这些消息告诉她的朋友。但是,当她告诉她的朋友 Chuck,却得知他已经知道这个消息时,她会很失望。她可能会不再给其他朋友打电话,因为如果他们已经知道这个消息,打这个电话还有什么意思呢?

gossiping 的确是一种快速扩散更新的好方法。但是,它不能保证所有服务器都确实被更新了(Demers 等 1987)。例如,当数据存储由大量的服务器组成,一部分服务器 s 满足以下等式时就会未被更新,即仍为易感的。

$$s = e^{-(k+1)(1-s)}$$

例如,如果 $k=3$, s 小于 0.02,即少于 2% 的服务器仍为易感的。不过此时需要特殊的措施来保证那些服务器也将被更新。将反熵和 gossiping 结合起来就可完成这一任务。

epidemic 算法的一个主要优势是它的可扩展性,这是因为进程间的同步数量比其他传播方法相对较少。对于广域系统,Lin 和 Marzullo 表示考虑实际网络拓扑以获得更好的效果是合理的(Lin 和 Marzullo 1999)。在他们的方法中,只与少数其他服务器相连的服务器被联系的可能性相对较高。其基本假设是,这样的服务器形成了到达网络中其他远程服务器的桥,因此其他服务器会尽可能与它们联系。

2. 删除数据

epidemic 算法对于在最终一致的数据存储中扩散更新十分有效。但是,它们都具有一个相当危险的副作用:难以扩散一个数据项的删除。这个问题的本质在于删除一个数据项时破坏了该数据项的所有信息。因此,当一个数据项简单地在一台服务器删除时,那台服务器将最终接收到该数据项的旧拷贝,并把这些旧拷贝当作对其先前未持有的数据的更新。

解决这一问题的窍门就是像记录其他更新一样也记录数据项的删除,并保持删除的记录。通过这种方式,旧拷贝就不会被视为新数据,而只是被看作已被删除操作更新的数据的版本。删除记录的传播是通过散布死亡证书(death certificate)实现的。

当然,死亡证书的问题是它们最终应该被清除,否则每台服务器将逐渐建立关于被删除数据项的历史信息的巨大本地数据库,而这个数据库别无它用。Demers 等提议使用称为休眠的死亡证书的方法。每个死亡证书在创建时获得一个时间戳。如果假设更新在一个已知的有限时间内传播到所有服务器,那么最大传播时间之后死亡证书就可以被删除了。

但是,为了确实保证删除的确被扩散到所有服务器,只有极少数的服务器将保留休眠的死亡证书,并从不删除它们。假设服务器 P 对数据项 x 有这样一个证书。如果万一有对 x 的过时更新到达 P,那么 P 只需再次散布 x 的死亡证书即可。

6.5 一致性协议

到目前为止,我们的注意力主要集中在各种一致性模型和一致性协议的一般设计问题。本节我们通过几个一致性协议的实例来集中讨论一致性模型的实际实现。一致性协议(consistency protocol)描述特定的一致性模型的实现。基本上,最重要且应用最广的模型是其操作是全局串行的一致性模型。这些模型包括顺序一致性、使用同步变量的弱一致性,以及原子性事务处理。本节我们介绍实现这些一致性模型的各种方法。

文献(Stumm 和 Zhou 1990)提出的早期方法根据是否存在一个数据的主拷贝,即所有写操作都被转发到的拷贝,将顺序一致性的协议分类。当不存在这种主拷贝时,写操作可以在任何副本上启动。

6.5.1 基于主备份的协议

在基于主备份的协议中,数据存储中的每个数据项 x 都有一个关联的主备份,该主备份负责协调 x 上的写操作。根据主备份是否被固定在一个远程服务器上,或将主备份移动到启动写操作的进程那里之后写操作是否可在进程本地执行,可以区分各种基于主备份的协议。

1. 远程写协议

最简单的基于主备份的协议是一种所有读操作和写操作都在单一的远程服务器上执行的协议。实际上,此时数据根本没有被复制,而是被放置在一个单一的服务器上,而且

数据不能从该服务器被移走。这一模型通常用于客户-服务器系统，此时服务器可以是分布式的。该协议如图 6.27 所示。

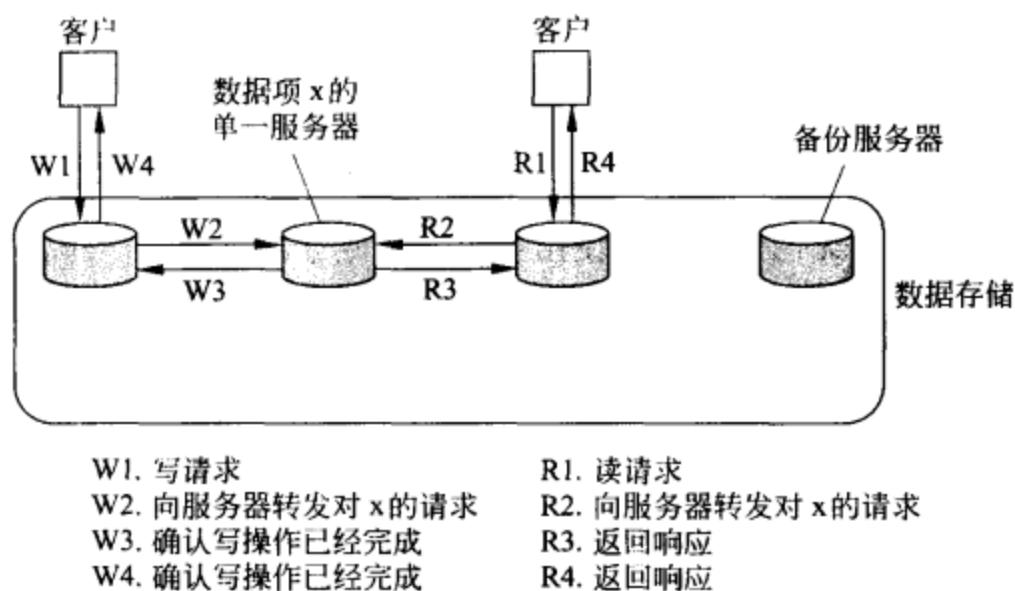


图 6.27 基于主备份的远程写协议，所有读操作和写操作都被转发到一个固定的服务器上

从一致性的角度来看，更有趣的协议是允许进程在本地可用的副本上执行读操作，但必须向一个（固定的）主拷贝转发写操作的协议。这种方法常称为主机备份协议（primary-backup protocol）（Budhiraja 等 1993）。图 6.28 表示了一个主机备份协议的工作原理。要对数据项 x 执行写操作的进程把该操作转发给 x 的主服务器。主服务器在其 x 的本地拷贝上执行更新，然后将更新转发给备份服务器。每个备份服务器也都执行这个更新，然后向主服务器返回一个确认消息。当所有的备份服务器都更新了它们的本地拷贝后，主服务器向初始进程返回一个确认消息。

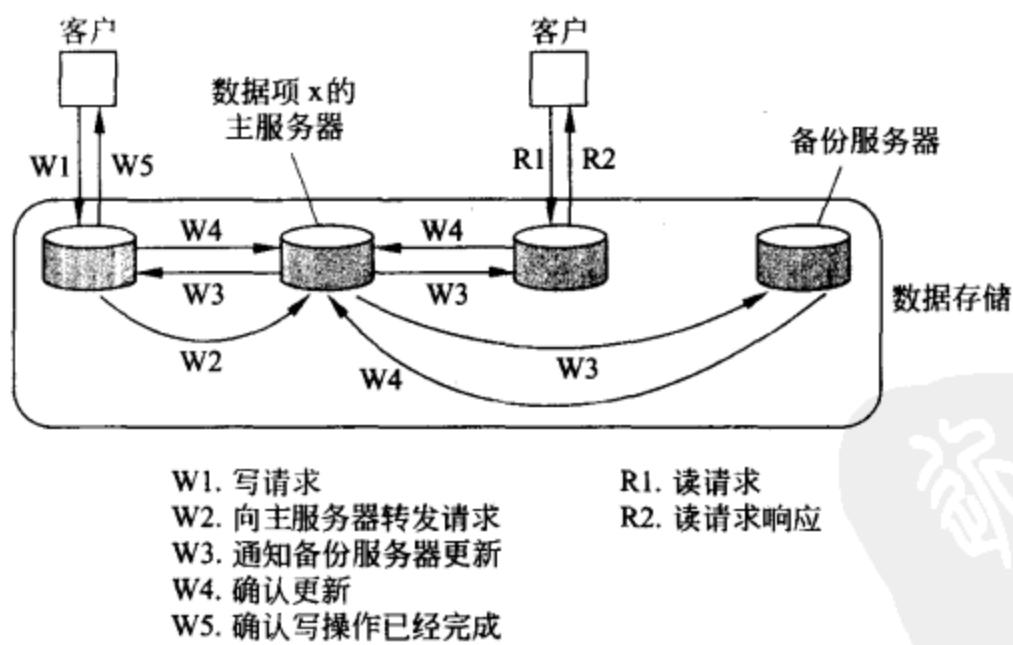


图 6.28 主机备份协议的原理

这种方法存在一个潜在性能问题，就是启动更新的进程被允许继续执行前可能需要等待相对较长的时间。实际上，在这种方法中，更新是以一种阻塞操作的方法实现的。另一种方法是使用非阻塞的方法。只要主服务器已经更新了其 x 的本地拷贝，它就返回确

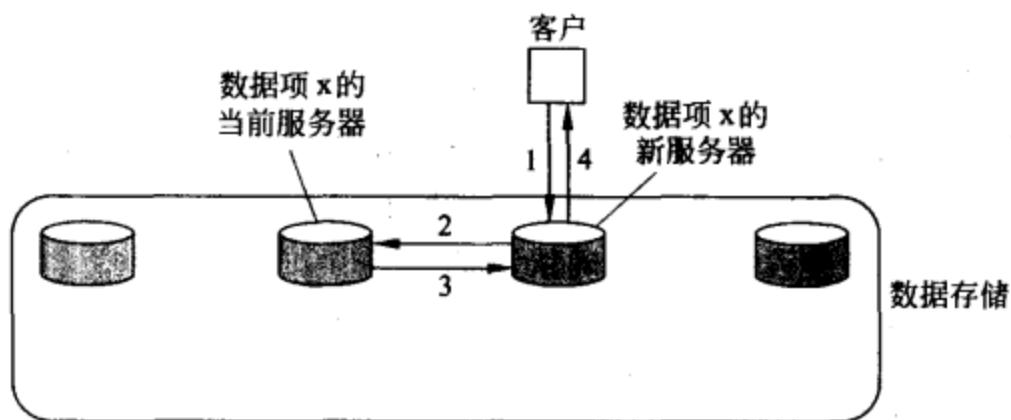
认消息。然后,它再通知备份服务器也执行这个更新。非阻塞的主机备份协议在文献(Budhiraja 和 Marzullo 1992)中有所讨论。

非阻塞的主机备份协议中的主要问题是它的容错能力。在阻塞的方法中,客户端进程确切地知道其他若干台服务器备份了更新操作。但是在非阻塞的方法中,情况并不是这样。我们将在下一章广泛地讨论容错问题。

主机备份协议提供了一种实现顺序一致性的直接方法,因为主服务器可以对所有进来的写操作排序。显然,无论各个进程使用哪个备份服务器执行读操作,所有进程都会以相同的顺序看到所有的写操作。在阻塞协议中,进程将总会看到它们最近执行的写操作的结果(注意,非阻塞协议不采取特殊措施时不能保证这一点)。

2. 本地写协议

基于主备份的本地写协议有以下两种。第一种协议是每个数据项 x 都只有一个单一的拷贝。也就是说不存在副本。每当一个进程要在某个数据项上执行操作时,先将那个数据项的单一拷贝传送到这个进程,然后在执行相应操作。这个协议本质上建立了一个完全分布式的、非复制的数据存储。其一致性是直截了当的,因为每个数据项总是只有一个单一的拷贝。图 6.29 表示了这个协议。



1. 读请求或写请求;
2. 向 x 的当前服务器转发请求;
3. 将数据项 x 移动到客户的服务器上;
4. 返回在客户的服务器上执行的操作的结果

图 6.29 基于主备份的本地写协议,其中一个单一的拷贝在多个进程间移动

这个完全迁移的方法的一个主要问题是它需要跟踪每个数据项的当前位置。如第 4 章所述,对于局域方法来说可使用底层的广播工具来解决这个问题。这个问题的另一种解决方法是使用转发指针和基于原点的方法。这种解决方法已应用于分布式共享存储器系统(例如,可参阅 Li 和 Hudak 1989)。但是,使用大规模的、分布较广的数据存储时,需要使用其他机制,例如第 4 章所讨论的层次定位服务。

上述的本地写协议的一种变形是在主机备份协议中,主拷贝在多个要执行写操作的进程之间迁移。如前所述,每当一个进程要更新数据项 x 时,它先定位 x 的主拷贝,然后将该拷贝移动到它自己的位置,如图 6.30 所示。这种方法的主要优点是多个相继的写操作可在本地执行,而执行读操作的进程仍然可以访问它们的本地拷贝。但是如上所述,只有使用非阻塞协议,即在主备份接收到更新后,通过该协议将更新传播到其他副本时,才

可能实现这个优点。这个协议已经被应用于各种分布式共享存储器系统。

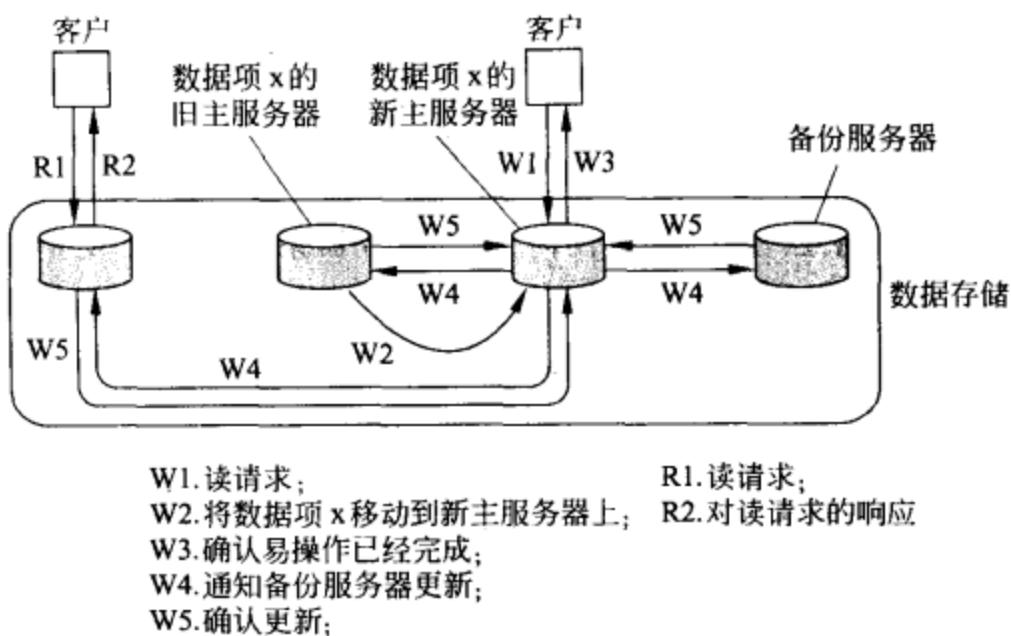


图 6.30 主机备份协议,其中主备份移动到要执行更新的进程那里

这种主机备份的本地写协议也可应用于能够以离线方式操作的移动计算机。离线前,移动计算机成为每个它期望更新的数据项的主服务器。离线时,所有更新操作都在本地执行,而其他进程仍可以执行读操作(但不能执行更新操作)。稍后,再次连接时,更新从主服务器传播到备份服务器,使数据存储再次达到一致的状态。我们将在第 10 章讨论分布式文件系统时,再讨论离线方式的操作。

6.5.2 复制的写协议

在复制的写协议中,写操作可以在多个副本上执行,而不是像基于主备份的副本那样只在一个副本上执行。主动复制和基于多数表决的一致性协议的区别在于:主动复制中的操作被转发到所有副本。

1. 主动复制

在主动复制中,每个副本有一个关联的进程,该进程执行更新操作。与其他协议相比,更新通常是通过造成更新的写操作传播的。也就是说,操作被发送到每个副本。但是,使用前面讨论的方法发送更新也是可以的。

主动复制中的一个潜在问题是操作需要在各地以相同的顺序执行,因而需要一种全序的多播机制。这种多播可以用上一章所讨论的 Lamport 时间戳实现。不幸的是,Lamport 时间戳用于大型分布式系统时的扩展性不好。一种替代的方法是使用中央协调器,也称为定序器(sequencer),来实现全序。其中一种方法是先将所有操作转发到定序器,由定序器为每个操作分配一个唯一的序列号,随后将这些操作转发给所有副本。操作按照它们的序列号的顺序执行。显然,这种全序多播的实现与基于主备份的一致性协议非常相似。

值得注意的是,使用定序器并不能解决可扩展性问题。实际上,如果需要全序的多播机制,那么结合 Lamport 时间戳和定序器的对称多播可能是必不可少的。这种解决方法

在文献(Rodrigues 等 1996)中有所描述。

另一个需要解决的问题是被复制的调用。例如,如图 6.31 所示,对象 A 调用另一个对象 B。假设对象 B 调用另一个对象 C。如果对象 B 是复制的,那么原则上,B 的每个副本都会独立地调用对象 C。问题是对象 C 现在被调用了多次,而不是只被调用了一次。如果用 C 的被调用的方法转账 \$100 000,那么显然迟早会有人抱怨。

复制的调用不是对象独有的问题,任何客户-服务器系统都可能发生这种情况。每当一台服务器既是复制的,又依赖于它可能对之发出请求的其他服务器时,我们会陷入同样的困境:请求也是复制的,复制的请求可能导致各种有害的效果。

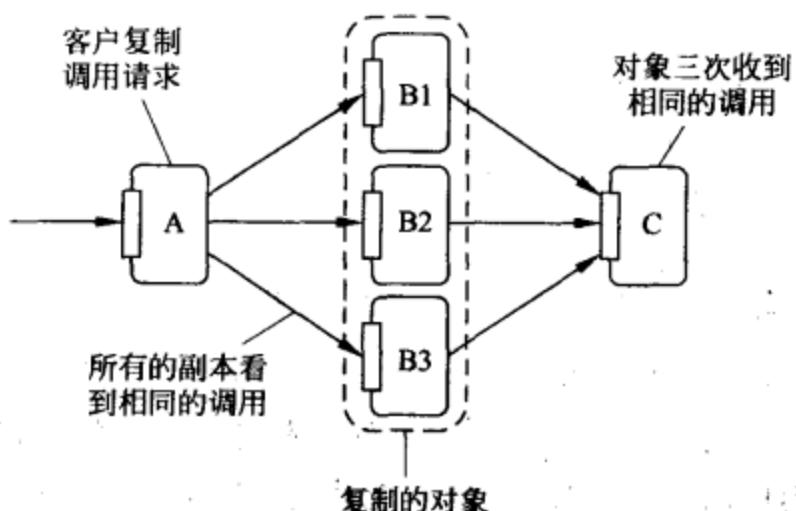


图 6.31 复制的调用问题

解决复制的调用问题的通用方法不多。文献(Mazouni 等 1995)描述了一种解决方法,此方法与复制策略无关,也就是说,与副本如何保持一致的确切细节无关。Mazouni 等的解决方法是 GASF 系统(Garbinato 等 1994)的一部分。其方法的本质是提供支持复制的通信层,(复制的)对象运行在该层之上。一个复制的对象 B 调用另一个复制的对象 C 时,B 的每个副本都先给该调用请求分配一个相同的、惟一的标识符。此时,B 的副本的协调器将 B 的请求转发给对象 C 的所有副本,而 B 的其他副本隐瞒它们的调用请求,如图 6.32(a)所示。这样做的结果是只有一个单一的请求被转发到 C 的每个副本。

相同的机制也被用于保证向 B 的各个副本只返回单一的应答消息。图 6.32(b)表示了这种情况。C 的副本的协调器注意到它正在处理复制的应答消息,这个应答消息是由 C 的每个副本产生的。但是,只有协调器将这个应答消息转发给对象 B 的各个副本,而 C 的其他副本隐瞒它们的应答消息。

B 的一个副本接收到一个调用请求的应答消息时,不管该调用请求是被转发给 C 还是因为该副本不是协调器而被隐瞒,该应答消息都被传递给真实的对象。

实际上,虽然上述的方法基于使用多播通信,但是它防止了不同的副本多播相同的消息。就这一点而论,它本质上是一种基于发送端的方法。另一种解决方法是让接收副本检查接收到的属于同一调用的输入消息的多个拷贝,并只将一个消息拷贝传递给其关联的对象。这种方法的细节留作练习。

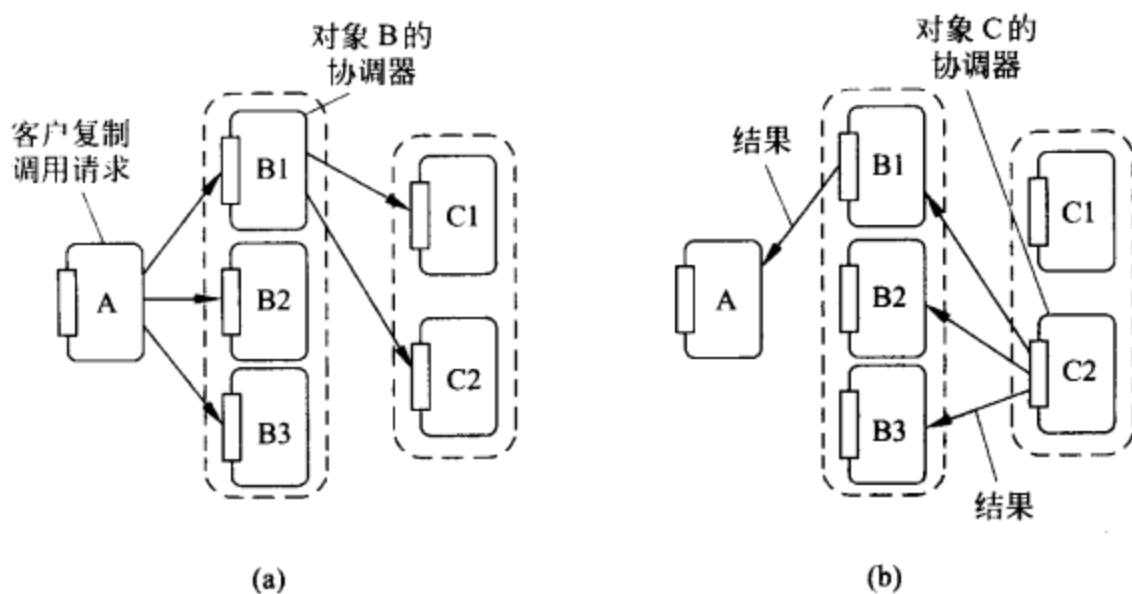


图 6.32

- (a) 从一个复制的对象向另一个复制的对象转发调用请求
- (b) 从一个复制的对象向另一个复制的对象返回应答消息

2. 基于法定数量的协议

一种不同的支持复制的写协议的方法是使用 Thomas(1979)最先提出并由 Gifford(1979)推广的表决方法。其基本思想是要求客户在读或写一个复制的数据项之前向多个服务器提出请求，并获得它们的许可。

下面以一个简单的实例说明该算法工作原理。以一个分布式文件系统为例，假设其文件被复制在 N 个服务器上。我们规定：要更新一个文件，客户必须先联系至少半数加一个服务器（多数服务器），并使它们同意它执行更新。一旦它们同意更新，该文件将被修改，这个新文件也将与一个新版本号关联。该版本号用于识别文件的版本，对于所有新更新的文件，它们的版本号是相同的。

为了读一个复制文件，客户也必须联系至少半数加一个服务器，请求它们返回该文件关联的版本号。如果所有的版本号一致，那么该版本必定是最新的版本，这是因为剩余服务器的个数不够半数以上，试图只更新剩余服务器的请求将会失败。

例如，如果系统有 5 个服务器，一个客户测定其中 3 个服务器持有第 8 版本的文件，那么其余 2 个服务器不可能持有第 9 版本的文件。毕竟，所有成功地从第 8 版本到第 9 版本的更新需要得到 3 个服务器的同意，而不能只得到 2 个服务器的同意。

实际上，Gifford 的方法比这个方法更加通用。在 Gifford 的方法中，一个客户要读取一个具有 N 个副本的文件，它必须组织一个读团体（read quorum），该读团体是 N_R 个以上服务器的任意集合。同样地，要修改一个文件，客户必须组织一个至少有 N_w 个服务器的写团体（write quorum）。 N_R 和 N_w 的值应满足以下两个限制条件：

- (1) $N_R + N_w > n$;
- (2) $N_w > N/2$ 。

第一个限制条件用于防止读写操作冲突，而第二个限制条件用于防止读读操作冲突。只有在适当个数的服务器同意参与文件的读写后，客户才能读或写该文件。

为了理解该算法的工作方式,我们以图 6.33(a)为例,其中 $N_R=3$ 且 $N_W=10$ 。假设最近的写团体由服务器 C 到 L 的 10 个服务器组成。任何随后由三个服务器组成的读团体必须至少包含一个该集合中的服务器。客户查看版本号时,它会知道哪个服务器是最新的,并选择那个服务器。

在图 6.33(b)和(c)中,我们又看到了两个实例。图 6.33(b)中的实例可能发生写写操作冲突,这是因为 $N_W \leq N/2$ 。特别是,如果一个客户选择 {A,B,C,E,F,G} 作为它的写集合,而另一个客户选择 {D,H,I,J,K,L} 作为它的写集合,那么显然会遇到麻烦,因为我们没有检测这两个更新实际上是否冲突而直接接受了这两个更新。

图 6.33(c)所示的情形特别有趣,因为它设置 N_R 为 1,这使它可以通过找到并使用复制的文件的任何拷贝来读取该文件。但是,它所付出的代价是写更新需要获取所有拷贝。这种方法通常称为“读一个,写全部”(read-one, write-all, ROWA)。基于法定数量的复制协议有多种变形。文献(Jalote 1994)很好地总结了这些协议。

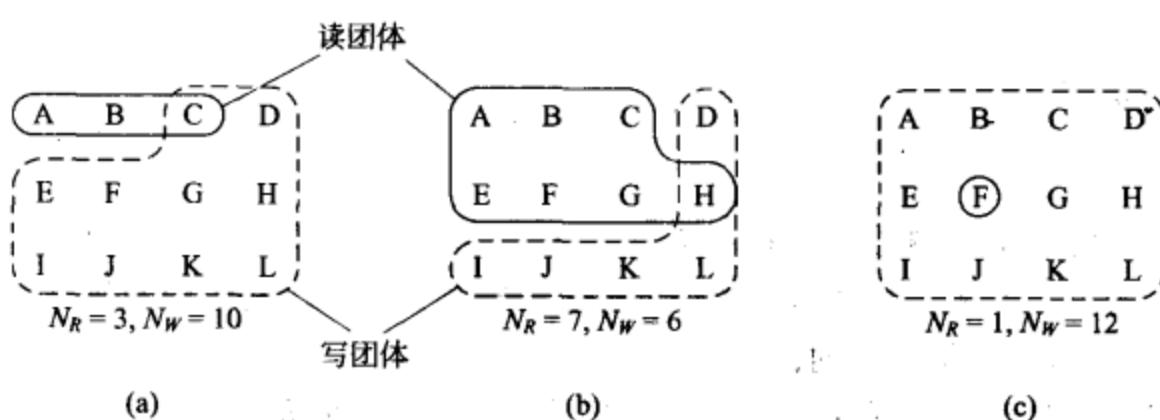


图 6.33 表决算法的三个实例

- (a) 读集合和写集合的正确选择;
- (b) 可能导致写写操作冲突的读集合和写集合;
- (c) 读集合和写集合的正确选择,称为 ROWA(read-one, write-all)

6.5.3 高速缓存相关性协议

高速缓存形成了一类特殊的复制,因为它们通常由客户而不是服务器控制。高速缓存相关性协议保证高速缓存与服务器启动的副本一致,在原理上,高速缓存相关性与迄今为止我们讨论的各种一致性协议有没有多大差别。

关于高速缓存的设计与实现方面的研究很多,特别是在共享存储器的多处理器系统环境中。很多方法基于底层硬件的支持,例如假设可以实现侦听或高效的广播。基于中间件的分布式系统建立在通用的操作系统之上,在这种环境中,基于软件来解决高速缓存问题的方法备受关注。此时,通常使用两个单独的标准来划分高速缓存协议(Min 和 Baer 1992, Lilja 1993, Tartalja 和 Milutinovic 1997)。

首先,不同的高速缓存解决方法在相关性检测策略上有区别,即实际检测不一致性的时间不同。静态的解决方法假定编译器在运行之前执行必要的分析,并确定哪些数据可能因为它们被缓存而实际导致了不一致性。编译器仅插入一些避免不一致性的指令。动态解决方法通常应用于本书所研究的各种分布式系统。在这些解决方法中,不一致性是在运行时检测的。例如,检查服务器,查看被缓存的数据自从被缓存后是否被改变过。

在分布式数据库的情况下,根据事务处理期间进行检测的确切时间,可将动态的基于检测的协议进一步精确分类。Franklin 等(1997)将其分为以下三类。第一类,在事务处理期间访问高速缓存的数据项时,客户需要检验该数据项是否仍与(可能是复制的)服务器中的该数据项一致。在检查完一致性之前,事务处理不能开始使用缓存的数据。

第二类方法是在进行一致性检验时,继续执行事务处理,该方法是最优的方法。它假设事务处理启动时高速缓存的数据是最新的。如果稍后证实该假设是错误的,那么事务处理将不得不异常中止。

第三类方法是只有事务处理委托检验时才检验高速缓存的数据是否是最新的。这类方法与上一章所讨论的乐观并发控制方法类似。实际上,事务处理只是启动对缓存的数据的操作,并希望得到令人满意的结果。在所有工作都完成后,再检验被访问的数据的一致性。如果事务处理使用的是过时的数据,那么事务处理被异常中止。

高速缓存相关性的另一个设计问题是相关性实施策略(coherence enforcement strategy),该策略决定高速缓存如何与服务器存储的各拷贝保持一致。最简单的解决方法是完全不允许缓存共享数据,而是将共享数据仅保存在服务器上。服务器使用前面讨论的基于主备份的协议或复制的写协议来维护一致性,客户只允许缓存私有数据。显然,这种解决方法只能提供有限的性能改善。

当共享数据可以被缓存时,实现高速缓存的一致性可采用两种方法。第一种方法是每当一个数据项被修改后,服务器都向所有高速缓存发送无效化消息。第二种方法是仅传播更新。大多数高速缓存系统使用其中一种方法来实现高速缓存的一致性。有时,一些客户服务器数据库动态地选择发送无效化消息或发送更新。

最后,我们还需要考虑进程修改被缓存的数据时可能发生的情况。使用只读高速缓存时,更新操作只能由服务器执行,然后服务器根据某个分发协议来确保更新被传播到各个高速缓存。很多系统使用基于拉式的方法。此时,如果客户检测到它的高速缓存已经过时,它就向服务器请求更新。

另一种方法是允许客户直接修改被缓存的数据,并将更新转发给服务器。直写式高速缓存(write-through cache)采用这种方法,分布式文件系统常使用这种直写式高速缓存。实际上,直写式高速缓存类似于基于主备份的本地写协议,客户的高速缓存成为一个临时的主备份。为了保证顺序一致性,客户必须被授予独占写的特权,否则可能发生写写操作冲突。

直写式高速缓存潜在地提供较其他方法更好的性能,这是因为所有的操作都可以在本地执行。如果我们通过允许在通知服务器更新之前执行多个写操作来推迟传播更新,那么性能可以得到进一步的改善。这种方法导致了回写式高速缓存(write-back cache)的出现,这种回写式高速缓存也主要应用于分布式文件系统。

6.6 实例

一致性和复制应用于很多分布式系统。本节我们详细讨论两个实现一致性和复制的实例。这两个实例大不相同。首先,我们考虑一个称为 Orca 的分布式对象系统。Orca

将物理上分布的对象与强一致性模型相结合。但是在这种方法中,用户完全不知道对象是分布式复制的。因而,Orca 提供了较高程度的透明性。

第二个实例是分布式数据库的实例,该数据库的副本保持因果一致。之所以介绍这个实例是因为:第一,该实例表明确实能以或多或少相互独立的方式处理更新传播和一致性。第二,它演示如何使用向量时间戳高效地实现因果一致性。第三,这个实例中向量时间戳的用法与下一章将讨论的虚拟同步系统的某些实现方面相似。

6.6.1 Orca

Orca 是一个程序设计系统,它允许不同机器上的进程可以受控地访问分布式共享存储器,该存储器由受保护的对象组成(Bal 等 1992,Bal 等 1998)。Orca 系统包括:Orca 语言、编译器及运行期间实际控制共享对象的运行时系统。尽管语言、编译器和运行时系统被设计成一起工作,但是运行时系统独立于编译器,它已经用于其他语言(Bhoedjang 等 1993,Bhoedjang 2000)。引入 Orca 语言后,我们将讨论运行时系统如何实现可物理分布于多台机器的共享对象。

1. Orca 语言

在某些方面,Orca 是一种传统的语言,它的顺序语句基本上以 Modula-2 为基础。但是,它不支持指针和别名,是一种类型安全的语言。数组边界的检查是在运行时进行的(可以在编译时进行的检查除外)。这些特性及其相似的特性可以消除或检测出许多通用的程序设计错误,例如存储器混乱。其语言特性是精心选择过的,它们使各种优化更易实现。

Orca 语言对于分布式程序设计很重要的两个特性是:共享的数据对象和 fork 语句。对象封装内部数据结构的过程和用户为操作内部数据结构而编写的过程称为操作(operation)。对象是被动的,也就是,它们不包含可接收消息的线程。而进程则通过调用对象的操作来访问它的内部数据。对象不能继承其他对象的属性,所以 Orca 不是面向对象的语言,而是基于对象的语言。

每个操作包括一系列(guard,程序块)对。其中,guard 可以是不含任何副作用的布尔表达式,也可以是空的,空 guard 的值与 true(真)相同。一个操作被调用时,它的所有的 guard 以未知的顺序被赋值。如果这些 guard 的值都是 false(假),那么调用该操作的进程将被挂起直到其中一个 guard 变为 true。当发现一个 guard 被赋为 true 时,执行该 guard 后面的程序块。图 6.34 描述了一个 stack(栈)对象,它有两个操作:push(入栈)和 pop(出栈)。

一旦定义了 stack(栈)对象,就可以使用如下语句声明这种类型的变量:

```
s,t: stack;
```

该语句创建两个栈对象,并将每个对象的 top 变量初始化为 0。以下语句可以把整型变量 k 推入栈内

```
s: s $ push(k);
```

其他情况以此类推。pop 操作有一个 guard, 所以试图从空栈弹出变量的调用者将被挂起, 直到其他进程向栈内推入某变量。

```
OBJECT IMPLEMENTATION stack;
    top: integer;                                # 指示栈顶的变量
    stack: ARRAY [integer 0..N-1] OF integer;      # 栈的存储空间

    OPERATION push(item: integer);                # 无返回的函数
    BEGIN
        GUARD top < N DO
            stack[top] := item;                      # 向栈中推入数据项
            top := top + 1;                          # 栈指针加 1
        OD;
    END;

    OPERATION pop(): integer;                     # 返回整数的函数
    BEGIN
        GUARD top > 0 DO
            top := top - 1;                        # 如果栈为空, 挂起
            RETURN stack[top];                     # 栈指针减 1
        OD;
    END;
    BEGIN
        top := 0;                                  # 初始化
    END;
```

图 6.34 用 Orca 定义的一个简化的栈对象, 该对象带有内部数据和两个操作

Orca 用 fork 语句在用户指定的处理器上创建新进程。这个新进程运行 fork 语句指定的过程。用户可以向这个新进程传送包括对象在内的各种参数。这就是如何使对象分布于多台机器的方法。例如, 下面的语句

```
FOR i IN 1..n DO FORK foobar(s) ON i; OD;
```

在从 1 到 n 的每台机器上生成一个新进程, 该进程在每台机器上运行程序 foobar。因为这 n 个新进程(以及父进程)是并行执行的, 所以它们好像运行在共享存储器的多处理器上一样, 都可以向共享栈 s 推入数据项或从 s 弹出数据项。运行时系统负责维护这个假想的共享存储器。

共享对象上的操作是原子的、顺序一致的。系统负责保证, 在多个进程几乎同时对同一共享对象进行操作时, 最终的效果是这些操作好像以严格的顺序(某种未指定的顺序)发生, 前一个操作没有结束前不会开始执行其他操作。

对于所有进程而言, 这些操作以相同的顺序出现。例如, 假设我们用新操作 peek 扩充图 6.34 中的 stack 对象, 该操作检查堆栈的栈顶数据项。如果两个独立的进程同时向堆栈推入 3 和 4, 随后所有进程都使用 peek 查看栈顶的数据项, 那么系统负责保证每个进程都看到 3, 或者每个进程都看到 4。在共享存储器的多处理器系统中不可能出现某些进程看到 3, 而其他进程看到 4 的情况, Orca 中也不会出现这种情况。如果只维护一份栈拷贝, 那么很容易实现上述效果, 但是如果栈在所有机器上复制的, 那么需要采取更多措施来实现上述效果。下面将介绍这些措施。

Orca 集成共享数据和同步的方式与入口一致性相似。第一种同步方式是互斥同步,

即禁止两个进程同时执行同一临界区。在 Orca 中,共享对象上的每个操作都与临界区非常相似,因为系统负责保证操作的最后结果是相同的,就好像所有的临界区是一个接着一个地(即,顺序地)被实施的。从这方面来说,Orca 对象就像是第 1 章讨论的监视器的分布式形式。

另一种同步方式是条件同步,即阻塞等待某个条件被满足的进程。在 Orca 中,条件同步是通过 guard 实现的。在图 6.34 的实例中,试图从空栈弹出数据项的进程将被挂起,直到该栈不再是空栈为止。

2. Orca 中共享对象的管理

在 Orca 中,对象管理是由运行时系统处理的。它既可在广播(或多播)网络上运行,又可在点对点网络上运行。运行时系统处理对象的复制、转移、一致性和操作调用。

每个对象可以处于以下两种状态之一:单一拷贝状态或被复制的状态。单一拷贝状态的对象只存在于一台机器。被复制的对象出现在所有使用该对象的进程所在的机器上。所有对象都处于相同的状态是不必要的,所以一些被进程使用的对象可能处于被复制的状态,而其他对象处于单一拷贝状态。在运行期间,对象可以在单一拷贝的状态和被复制的状态之间来回转变,从而提供了很强的灵活性。

在每台机器上复制对象的最大优点是读操作可以不需要任何网络传输或延时而在本地完成。当一个对象没有被复制时,所有操作必须被发送到这个对象,而且调用者也必须被阻塞以等待回应。复制的第二个优点是增加并行性,即多个读操作可以同时发送。在单一拷贝上,同一时间只能发生一个操作,降低了执行速度。复制的主要缺点是保持所有拷贝一致需要额外开销。

当程序在对象上执行操作时,编译器调用运行时系统过程 `invoke_op`,并指定对象、操作、参数和一个标志。其中的标志说明对象会被修改(称为写操作)还是不会被修改(称为读操作)。过程 `invoke_op` 所采取的行动取决于对象是否是复制的,是否存在本地可用的拷贝,对象上的操作是读操作还是写操作,以及底层系统是否支持可靠的、完全有序的广播。这 4 种情况必须加以区分,如图 6.35 所示。

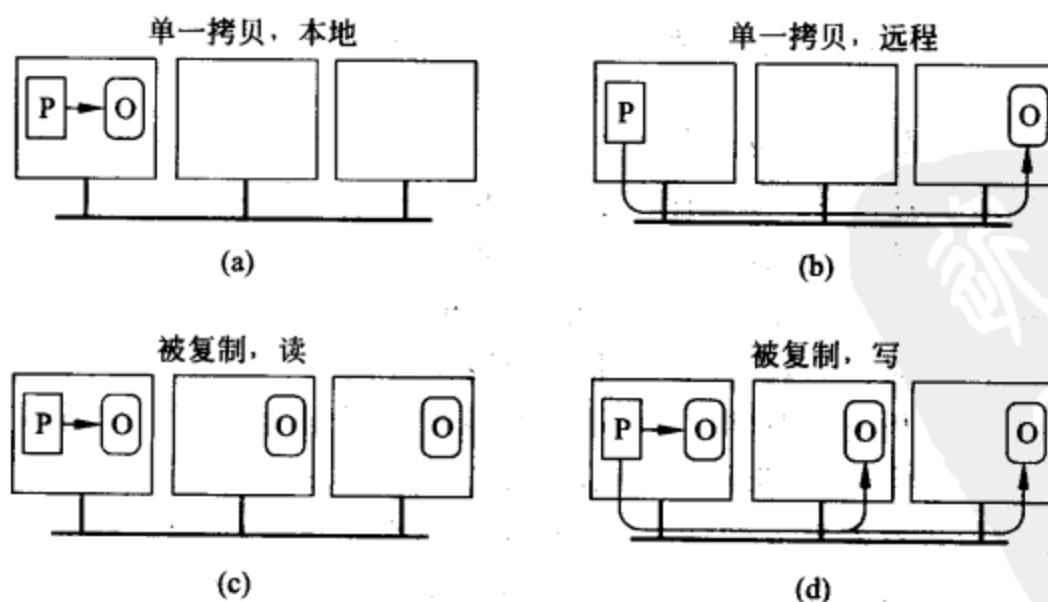


图 6.35 在 Orca 中,进程 P 在对象 O 上执行操作的 4 种情况

在图 6.35(a)中,进程要在一个非复制的对象上执行操作,这个对象恰好在该进程自己的机器上。该进程只需锁定对象,调用操作,然后给对象解锁,即可完成操作。锁定对象的目的是为了在本地操作执行期间,暂时禁止任何远程调用。

在图 6.35(b)中,我们仍使用一个只有单一拷贝的对象,但是这次这个对象不在本地机器上。运行时系统执行一次 RPC 调用,请求远程机器执行这个操作。此时,如果远程机器接到请求时该对象是被锁定的,那么 RPC 调用可能会有一些延时。在这两种情况下,读操作和写操作都没有什么区别(除非写操作可以唤醒被阻塞的进程,因为读操作可以改变 guard 所用变量的值)。

如果对象是复制的,如图 6.35(c)和(d)所示,那么该对象总是有一份本地可用的拷贝,但现在重要的是该操作是读操作还是写操作。读操作不需要任何网络传输及任何额外开销就可在本地执行。

在复制对象上进行写操作则比较棘手。如果底层系统提供可靠的、全序的广播机制,那么运行时系统可广播该对象的名字、操作和参数,并在广播完成前阻塞操作。广播完成后,包括其自身在内的所有机器都计算新值。

注意,广播原语必须是可靠的,这就意味着由较低层自动监测和恢复丢失的消息。Orca 是在 Amoeba 系统之上开发的,Amoeba 系统就具有这种特性。每个广播消息都被发送到一个称为序列器(sequencer)的特殊进程,该序列器为这个广播消息分配一个序列号,然后使用不可靠的广播硬件将其广播出去。每当进程注意到序列号不连续时,它就知道一个消息丢失了,并采取措施进行恢复。

如果系统不具有可靠的广播机制(或者根本不能进行广播),那么更新是通过使用基于主备份的协议完成的。每个对象都有一个主拷贝。进行更新的进程先向对象的主拷贝发送一个消息,锁定该对象,并进行更新。然后主拷贝再分别向所有其他持有该对象的机器发送消息,要求它们锁定它们的拷贝。当所有机器都应答已经锁定对象时,源进程进入第二个阶段,向它们发送另一个消息,通知它们执行更新并给对象解锁。

死锁是不可能发生的,这是因为即使两个进程企图同时更新同一对象,其中一个进程也会先到达主拷贝并锁定它,而另一个进程的请求将排队等候该对象再次被释放。同样需要注意的是,在更新过程期间,对象的所有拷贝都是被锁定的,所以任何其他进程都不能读取旧值。这种锁定机制保证所有操作是以全局惟一的顺序执行的。

现在,我们简要地介绍一下决定对象状态的算法,该算法决定对象是处于单一拷贝状态还是处于被复制的状态。最初,Orca 程序由一个进程组成,该进程拥有所有对象。当它派生新进程时,将向所有其他机器通知这个派生事件,并将所有子进程的共享参数的当前拷贝发送给这些机器。然后,每台机器的运行时系统计算复制每个对象与不复制每个对象的期望代价。

为了进行这种计算,系统需要知道读操作与写操作的期望比率。编译器通过检查程序来估算这一信息。检查程序时,它会考虑把内部循环访问计数较其他访问的计数相对增加,而把条件语句内部的访问的计数相对减少。通信代价也被包括在估算公式之内。例如,广播网络上读/写比率为 10 的对象将会被复制,而点对点网络上读/写比率为 1 的对象将会被设置为单一拷贝的状态,并且将这个单一拷贝转移到对其执行写操作最多的

机器上。关于如何实际解决这些问题,请参阅文献(Bal 和 Kasshoek 1993)。

因为所有运行时系统都进行相同的计算,所以它们也会得到相同的结论。如果一个对象当前只出现在一台机器上,但是它需要出现在所有机器上,那么它就被扩散到其他机器上。如果它当前是复制的,但是复制不再是最优选择,那么除一台机器之外的所有机器都要丢弃它们的拷贝。对象可以使用这种机器在两种状态之间转换。

下面我们看看顺序一致性是如何实现的。对于处于单一拷贝状态的对象而言,所有操作都是纯线性化的,所以顺序一致性是被无偿实现的。对于复制的对象而言,无论使用可靠的、完全有序的广播机制,还是使用基于主备份的算法,写操作都是完全有序的。在这两种方法中,写操作都具有全局一致的顺序。读操作是在本地执行的,它们也能以任意方式与读操作交叉,而不影响顺序一致性。

该实例可以进行多方面的优化。例如,同步不是在执行操作后进行,而是在启动操作时进行的,就像人口一致性或懒惰释放一致性那样进行同步。其优点是如果进程反复地(例如,循环)执行一个共享对象上的操作,那么在其他进程需要该对象之前,系统根本不需要发送广播。

另一方面的优化是在执行不返回数值的写操作(例如,栈实例中的 push 操作)后进行广播时不挂起调用者。当然,这种优化必须以透明的方式实现。编译器提供的信息使得进行其他方面的优化成为可能。

总之,分布式共享存储器的 Orca 模型以一种自然的方式集成了优良的软件工程规范(封装的对象)、共享数据、简单语义和同步机制。在很多情况下,它的实现也可以像释放一致性那样高效。在底层硬件和操作系统支持高效可靠的、完全有序的广播,以及应用程序对于共享对象的访问具有固有的较高读/写比率时,该模型工作得最好。

6.6.2 因果一致的懒惰复制

下面我们考虑一种复制方法,将其作为一致性和复制的一种完全不同的实例。该方法实现最终一致性,但是它同时跟踪操作间的因果关系。该方法用于最终一致的数据存储,但是它使用一种懒惰复制形式来传播更新,如 Ladin 等(1992)所述。

1. 系统模型

为了解释因果一致的懒惰复制如何工作,我们采用前面一直使用的分布式数据存储的模型(参看图 6.4)。因果一致的懒惰复制的本质是通过向量时间截捕获读操作和写操作间潜在的因果关系。上一章已经讨论了向量时间截。下面,我们假设数据存储被分布复制于 N 个服务器上。与前面类似,我们假设客户通常连接本地(或最近的)可用的服务器,但是这一原则不必严格遵循。

客户之间可以相互通信,但是它们必须交换它们在数据存储上执行的操作的信息。(因此,实际应用中,客户的集合常被组织为数据存储的前端和“纯”客户的集合,其中,前端负责交换数据存储上的操作的信息,而“纯”客户完全不知道在数据存储中如何处理一致性和复制。这里,我们对这一点不做区分。)图 6.36 表示了数据存储的通用组织方式。

如图 6.36 所示,数据存储的每个服务器由一个本地数据库和两个待处理操作的队列

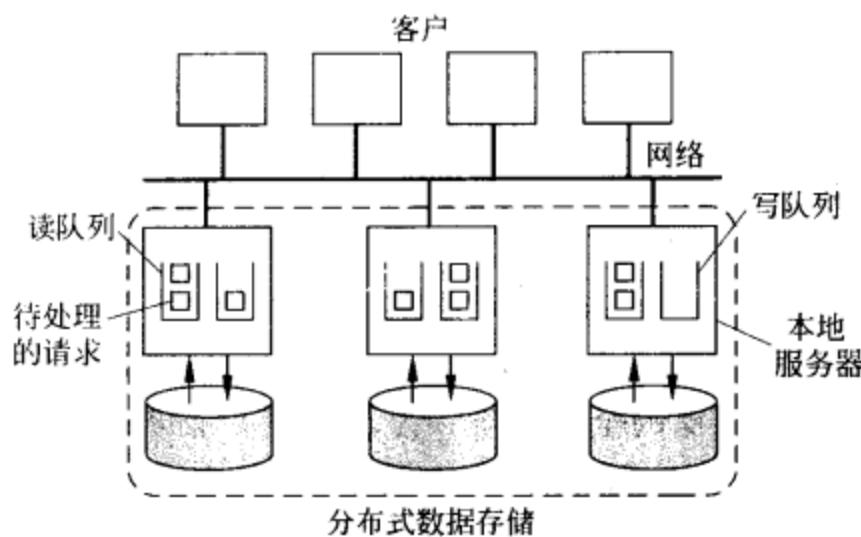


图 6.36 分布式数据存储的通用组织方式。假设客户也处理与一致性相关的通信

组成。本地数据库含有权威性数据，即该数据可以永久地存储于数据库之中而不违背数据存储的全局一致性模型。也就是说，它恰恰含有与客户和数据存储之间的软件合约相匹配的那些数据，软件合约以数据为中心的一致性模型的形式表示。在本实例中，这个合约强制本地拷贝满足因果一致性。

每个拷贝维护两个待处理操作的队列。读队列由在本地数据库与读操作期待的状态一致之前，需要被禁止的读操作组成。例如，如果读操作表明启动操作的客户已经看到某个写操作的结果，那么在执行读操作之前，本地数据库应该被那些操作更新。这一要求与前面讨论的以客户为中心的一致性模型非常相似。本地一致的实际意义将在后面解释。

同样，写队列由待处理的写操作组成。这些写操作是在数据库与写操作期待的状态一致之前，需要被禁止的写操作。特别地，只有在本地数据库已经被写操作所依赖的所有前面的有因果关系的操作更新后，该写操作才能被执行。细节将在后面讨论。

为了表示本地数据库的确切状态并精确地指定需要该状态的操作，应使用向量时间戳。首先，每个本地拷贝 L_i 都要维护两个向量时间戳。向量 $VAL(i)$ 表示拷贝 L_i 的当前状态。 $VAL(i)[i]$ 是已经被直接发送到拷贝 L_i 和已经在 L_i 完成的写请求的总数。另外， L_i 跟踪它已从拷贝 L_i 接收到（并处理）的更新操作的个数。这个数字记录在 $VAL(i)[j]$ 之中。

第二个向量时间戳 $WORK(i)$ 反映仍需在拷贝 L_i 处执行的操作。特别是， $WORK(i)[i]$ 是来自客户端的写操作的总数，它们包括那些已经被执行的操作。一旦写操作队列变为空，就表示这些写操作已经被执行了。同样， $WORK(i)[j]$ 反映来自 L_i 的更新的总数，当写队列变为空时，表示它们将已被完成。

除了这两个向量时间戳之外，每个客户还跟踪迄今为止它所看到的数据存储。为此，客户 C 维持一个向量时间戳 $LOCAL(C)$ ， $LOCAL(C)[i]$ 被设置为 C 所看到的 L_i 处的状态的最新值。（注意，每次客户想要读数据或写数据时，它可以联系不同的副本。）每次客户 C 向本地拷贝转发一个读或写请求 RW 时，它把 $LOCAL(C)$ 作为时间戳 $DEP(RW)$ 发送，以指明 RW 所依赖的操作。

2. 处理读操作

现在, 我们能够描述如何处理不同操作了。首先, 我们先描述读操作的处理方式, 图 6.37 显示了处理读操作的不同步骤。每当客户 C(本例中的前端)要执行读操作 R 时, 其关联的读请求的时间戳 $DEP(R)$ 被设置为 $LOCAL(C)$ 。这个时间戳反映了该客户当前所知道的关于数据存储的情况。然后, 该请求被发送到一个拷贝 L_i 处。

拷贝 L_i 处的输入读请求 R 总是存储在该拷贝的读队列中。时间戳 $DEP(R)$ 反映了提出 R 请求的时刻该数据存储的全局状态。为了处理 R 请求, L_i 也知道该状态是必要的。特别是对于每个 j , $DEP(R)[j] \leq VAL(i)[j]$ 。

一旦读操作可以执行, 拷贝 L_i 就向客户返回被请求的数据项的值, 以及 $VAL(i)$ 。然后, 客户通过将向量的每个项目 $LOCAL(C)[j]$ 置为值 $\max\{LOCAL(C)[j], VAL(i)[j]\}$ 来调整它自己的向量时间戳 $LOCAL(C)$ 。

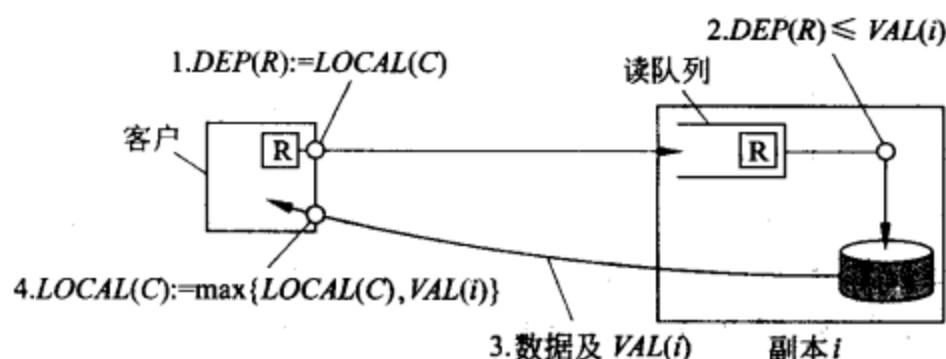


图 6.37 在本地拷贝执行读操作

3. 处理写操作

处理写操作的过程与处理读操作的过程多少有些相似, 如图 6.38 所示。当客户 C 要执行写操作 W 时, 它向拷贝 L_i 发送写请求, 并像前面一样将时间戳 $DEP(W)$ 设置为它的本地向量时间戳 $LOCAL(C)$ 。

当一个拷贝接收到来自客户的写请求 W 时, 它将 $WORK(i)[i]$ 加 1, 但是保持其他项目不变。另外, 写请求从 L_i 接收到时间戳 $ts(W)$, 其中 $ts(W)[i]$ 设置为 $WORK(i)[i]$, 而其他项目 $ts(W)[j]$ 设置为 $DEP(W)[j]$ 。这个时间戳将作为确认被发送回客户, 客户随后通过将每个第 k 个项目设置为 $\max\{LOCAL(C)[k], ts(W)[k]\}$ 来调整它的 $LOCAL(C)$ 。

与处理读请求的情况类似, 写请求 W 只有在 L_i 已经处理了 W 所依赖的所有更新时才能被执行。对于每个项目都有 $DEP(W)[j] \leq VAL(i)[j]$ 时, 就满足上述条件。此时, 执行操作, 并通过将每个第 j 个项目设置为 $\max\{VAL(i)[j], ts(W)[j]\}$ 来调整 $VAL(i)$ 。因为 L_i 接收到 W 并随后为其分配 $ts(W)[i]$ 时, $WORK(i)[i]$ 已经被加 1, 所以满足以下条件时, 很容易证实 L 处理了 W:

- (1) $ts(W)[i] = VAL(i)[i] + 1$
- (2) $ts(W)[j] \leq VAL(i)[j], \# j \neq i$

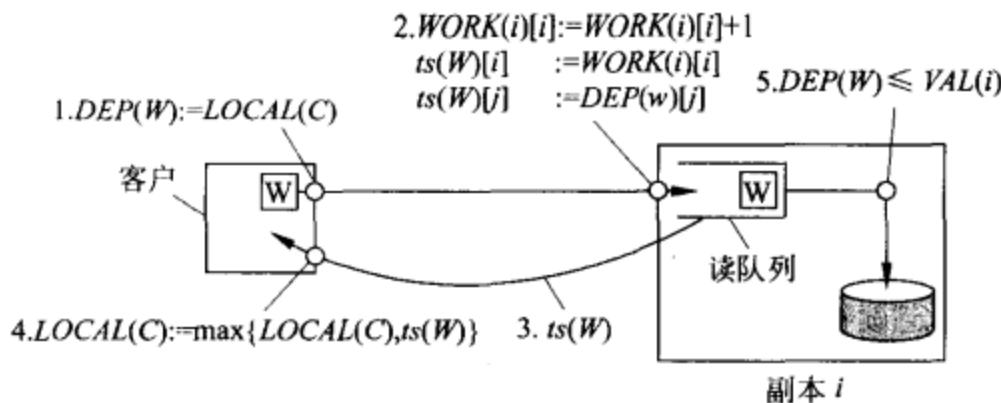


图 6.38 在本地拷贝执行写操作

第一个条件说明所有 W 之前来自客户端并被直接发送到 L_i 的操作已经被处理了。第二个条件说明 W 所依赖的所有更新也被 L_i 处理了。注意，这两个条件完全类似于上一章所讨论的实现因果消息传递的条件。

已执行的操作仍保留在队列中(但不会被再次执行)。它们仍被保留在队列中的原因是它们可能需要被转发到其他拷贝，以使该操作也在那些拷贝上被执行。下面我们将讨论更新传播的问题。

4. 更新传播

为了完成操作的处理，我们需要讨论不同拷贝之间更新操作的传播。更新传播是通过 epidemic 协议中应用的反熵实现的。一个拷贝 L_i 偶尔联系其他拷贝 L_j ，并交换各自写队列中的操作。

更具体地说，当一个拷贝 L_i 联系另一个拷贝 L_j 时，它向 L_j 发送它的写队列中当前待处理的所有操作。另外，向量时间戳 $WORK(i)$ 也被发送到 L_j 。对于每个操作， L_j 首先调整它自己的向量 $WORK(j)$ ，将每个第 k 个项目设置为 $\max\{WORK(i)[k], WORK(j)[k]\}$ 。然后， L_j 按如下方式将 L_i 发送来的待处理的写操作合并到它自己的写队列中：如果 L_j 已经包含 W ，那么仅将 W 丢弃，否则，将 W 加入到队列中。

交换过更新操作后，拷贝 L_i 检查是否可执行某个接收到的操作。设 U 表示所有待处理的写操作 W 的集合，且对于每个写操作 W 都满足：每个第 k 个项目， $DEP(W)[k] \leq VAL(i)[k]$ 。接着，拷贝 L_i 选择一个不依赖于 U 中任何其他操作的操作 W' 。也就是说， U 中没有其他操作 W 满足：对于每个第 k 个项目， $DEP(W)[k] \leq DEP(W')[k]$ 。然后，从 U 中删除 W' ，并处理 W ，此后再从 U 中选择另一个操作。

还存在很多未讨论的问题，其中，最重要的是到目前为止给出的描述中，写队列可以无限地增加的问题。显然，一旦知道某个操作已经在某处被执行了，那么就可以从队列删除该操作。检测这种情况需要引入其他管理机制及额外信息的交换，这方面的详细内容可参阅文献(Ladin 等 1992)。

6.7 小结

复制数据主要有两个原因：提高分布式系统的可靠性或提高性能。复制导致了一致性问题：每当一个副本被更新，该副本就变得与其他副本不同。为了保持各副本一致，我

们需要以一种不让人注意到的暂时的不一致的方式传播更新。不幸的是,这样做可能会严重地降低性能,特别是在大规模分布式系统中。

这一问题的惟一解决方法是确定一致性是否可以被稍微放松。在强一致性模型中,一致性是根据共享数据上单独的读操作和写操作定义的。它可分为严格一致性、顺序一致性和线性化、因果一致性以及 FIFO 一致性。

严格一致性规定读操作总是返回最新写入的值。由于分布式系统缺少全局时间,严格一致性无法实现。顺序一致性和线性化本质上提供了程序开发人员在并发程序设计中期望的语义:所有写操作总是以相同的顺序被每个进程看到。线性化稍强一点,因为它根据一个全局时钟(其精确度有限)来对操作排序。

因果一致性表现为潜在相互依赖的操作以这个依赖关系的顺序执行。FIFO 一致性仅规定只有来自单一进程的操作才按照那个进程指定的顺序执行。

弱一致性模型考虑读操作和写操作的序列。特别是,它们假定每个序列根据同步变量(例如锁)伴随的操作被适当地“分类”。尽管这要求程序开发人员显式地说明,但是弱一致性模型一般比强一致性模型更容易以高效的方式实现。

与这些以数据为中心的模型相对,移动用户的分布式数据库领域的研究人员已经定义了一系列以客户为中心的一致性模型。这种模型不考虑数据可能被多个用户共享的事实,相反,它们集中考虑一个单独的客户应被提供的一致性。其基本设想是客户在一段时间内要连接不同的副本,而这些不同副本的差别应该是透明的。实际上,以客户为中心的一致性模型确保每当客户连接一个新副本时,那个新副本使用该客户先前操作的数据更新,虽然这些数据有可能存储在其他副本处。

传播更新可使用不同的技术。不同的技术根据实际传播什么信息、传播到什么地方以及谁启动传播加以区分。我们可以决定传播通知、操作或状态。同样,不是每个副本都需要立即被更新。什么时间更新哪个副本取决于分布式协议。最后,传播更新技术还可以选择更新是被推入到其他副本,还是一个副本从其他副本处提取更新。

一致性协议描述一致性模型的特定实现。考虑到顺序一致性及其各种变型,一致性协议可分为两种:基于主备份的协议和复制的写协议。在基于主备份的协议中,所有更新操作都被转发到一个主拷贝,然后该主拷贝确保更新被正确地排序和转发。在复制的写协议中,更新同时被转发到多个副本。在这种情况下,正确地对操作进行排序变得更加困难。

习 题

1. 对共享的 Java 对象的访问可以通过将其方法声明为同步的而被串行化。当这种对象被复制时,这种方法足以保证访问的串行化吗?
2. 对于第 1 章所讨论的监视器,如果允许在一个复制的监视器中阻塞多个线程,那么给条件变量发信号时,需要保证什么?
3. 请用自己的语言解释实际考虑弱一致性模型的主要原因。
4. 请解释 DNS 如何进行复制,以及它实际运行很好的原因。

5. 讨论一致性模型时,我们经常提及软件和数据存储间的合约。为什么需要这一合约?
6. 线性化假设存在一个全局时钟。但是,我们已经在严格一致性中指出,这种假设对于大多数分布式系统都是不现实的。线性化可以应用于物理分布的数据存储吗?
7. 如果多处理器使用单一总线,那么可以实现严格一致的存储器吗?
8. 对于图 6.5(b),为什么 $W_1(x)a \quad R_2(x)NIL \quad R_3(x)a$ 是非法的?
9. 在图 6.7 中,000000 是仅满足 FIFO 一致的分布式共享存储器的合法输出吗?请解释您的答案。
10. 在图 6.8 中,001110 是顺序一致的存储器的合法输出吗?请解释您的答案。
11. 我们在 6.2.2 节的最后讨论了一个形式模型,该模型规定顺序一致的数据存储上的每个操作集合可以用一个字符串 H 表示,从这个字符串 H 可以衍生出所有的单独进程的顺序序列。对于图 6.9 中的进程 P1 和 P2,给出 H 所有可能的值。忽略进程 P3 和 P4,也不包括它们在 H 中的操作。
12. 在图 6.13 中,一个顺序一致的存储器允许 6 种可能的语句交叉。请列出这 6 种可能情况。
13. 通常认为弱一致性模型给程序开发人员强加了额外的负担。这一命题在何种程度上确实是正确的?
14. 在分布式共享存储器系统中的(急切)释放一致性的大多数实现中,共享变量是在执行释放操作时被同步的,而不是在执行获取操作时被同步的。那么,为什么还需要获取操作呢?
15. Orca 提供顺序一致性还是人口一致性?请解释您的答案。
16. 使用序列器并为维持主动复制的一致性而进行的全序的广播是否违背系统设计中的端到端理论?
17. 您会选择哪种类型的一致性来实现电子股票市场?请解释您的答案。
18. 如果要将一个移动用户的个人邮箱作为广域范围的分布式数据库的一部分来实现,哪种类型的以客户为中心的一致性最为合适?
19. 请描述一个用于显示刚被更新的 Web 页面的写后读一致性的简单实现。
20. 请给出以客户为中心的一致性易于导致写写操作冲突的实例。
21. 使用租用时,客户和服务器的时钟分别紧密同步是否是必要的?
22. 考虑一个用于保证分布式数据存储上顺序一致性的非阻塞的主机备份协议。这样的数据存储总能提供写后读一致性吗?
23. 为了使主动复制正常工作,所有操作必须以相同的顺序在每个副本处被执行。这个顺序总是必需的吗?
24. 为了使用序列器实现全序的广播,一种方法是先将操作转发给序列器,然后该序列器为这个操作分配一个唯一的号码,随后广播这个操作。请提出另外两种可选择的方法,并比较这三种方法。
25. 一个文件被复制在 10 个服务器上。请列出表决算法允许的所有读团体和写团体。

26. 为防止复制的调用,我们在书中讨论了基于发送方的方法。在基于接收方的方法中,是由接收副本辨认属于同一调用的输入消息的多个拷贝。请描述基于接收方的方法如何防止复制的调用。

27. 考虑在因果一致的懒惰复制中,一个操作被从一个写队列中删除的确切时间。

28. 在实现一个支持广播 RPC 的简单系统时,我们假设系统有多个复制的服务器,每个客户可以通过 RPC 与一个服务器通信。但是,处理复制时,客户需要向每个副本发送一个 RPC 请求。设计客户程序,以使客户好像只向应用程序发送单一的 RPC。假设复制的目的是为了提高性能,而那些服务器可能是易于出故障的。

第7章 容 错 性

分布式系统区别于单机系统的一个特性是它可能部分失效。当分布式系统中的一个组件发生故障时就可能产生部分失效。这个故障也许会影响到其他组件的正确操作,但同时也有可能完全不影响其他组件。而非分布式系统中的故障通常会影响到所有的组件,可能很容易就使整个应用程序崩溃。

分布式系统设计中的一个重要目标是以这样的方式来构造系统:它可以从部分失效中自动恢复,而且不会严重地影响整体性能。特别是,当故障发生时,分布式系统应该在进行恢复的同时继续以可接受的方式进行操作,也就是说,它应该能容忍错误,在发生错误时某种程度上可以继续操作。

在本章中,我们将更详细地学习使分布式系统具有容错性的技术。在提供了一些有关容错的背景知识后,我们将介绍进程恢复与可靠多播。进程恢复涉及到的技术可以使一个或多个进程发生故障而不会严重影响到系统的其余部分。可靠多播与这个问题相关,它确保可以成功地把信息传输到一个进程集合。要保持进程同步,可靠多播通常是必需的。

像我们在第5章中讨论过的一样,原子性在很多应用程序中都是一个重要的特性。例如在分布式事务中,就必须要保证一个事务中的所有操作全部发生或一个都不发生。在分布式系统中原子性的基础是分布式提交协议,我们将在本章中用单独一节来讨论它。

最后,我们将介绍如何从故障中恢复。特别是,我们将考虑何时以及如何对一个分布式系统的状态进行保存以确保在以后恢复到该状态。

7.1 容错性简介

容错涉及到计算机科学中的很多研究。在本节中,我们将从介绍与容错相关的基本概念开始,然后讨论故障的模式。我们也要讨论处理故障的关键技术——冗余。要获得关于分布式系统中有关容错的更多信息,请参阅文献(Jalote 1994)。

7.1.1 基本概念

要理解容错在分布式系统中的作用,我们首先需要深入地了解分布式系统中的容错到底意味着什么。容错与被称为可靠的系统(dependable system)紧密相关。可靠性是一个术语,它包含了分布式系统中很多有用的需求,列举以下(Kopetz and Verissimo 1993):

- (1) 可用性;
- (2) 可靠性;

(3) 安全性；

(4) 可维护性。

可用性(availability)被定义为系统的一个属性,它说明系统已准备好,马上就可以使用。通常,它指在任何给定的时刻,系统都可以正确地操作,可根据用户的行为来执行它的功能。换句话说,高度可用的系统在任何给定的时刻都能及时地工作。

可靠性(reliability)是指系统可以无故障地持续运行。与可用性相反,可靠性是根据时间间隔而不是任何时刻来进行定义的。高度可靠的系统可以在一个相对较长的时间内持续工作而不被中断。这很微妙,但是与可用性相比,这是一个重要的不同。如果系统在每小时中崩溃 1ms,那么它的可用性就超过 99.9999%,但是它还是高度不可靠的。与之类似,如果一个系统从来不崩溃,但是要在每年 8 月中停机两个星期,那么它就是高度可靠的,但是它的可用性只有 96%。这两种属性并不相同。

安全性(safety)是指在系统偶然出故障的情况下能正确操作而不会造成任何灾难。例如,很多进程控制系统,比如那些用来控制核电站或把人送入太空的控制系统,就必须提供高度的安全性。这样的控制系统即使只是非常短时间瞬时故障,结果也将是灾难性的。过去的很多例子(以后还会有更多的例子)都说明了要建立安全的系统是多么困难。

最后,可维护性(main tainability)是指发生故障的系统被恢复的难易程度。高度可维护的系统可能具有高度的可用性,特别是在可以探测到故障并自动恢复时。但是,像我们在本章后面将看到的那样,从故障中自动恢复说起来简单,做起来就难了。

通常也要求可靠的系统提供高度的安全性,特别是在要处理诸如完整性这样的问题时。我们将在下一章中讨论安全问题。

当一个系统不能兑现它的承诺时就被认为是失败的。尤其是,如果一个分布式系统被设计为为它的用户提供大量的服务,当这些服务中的一个或多个不能被(完整地)提供时,系统就发生故障了。而错误是系统状态的一部分,它可能会导致故障发生。例如,当从网络上传输数据包时,可能有一些数据包在到达接收者时已经被破坏了。在这种环境下,被破坏了意味着接收者可能错误地得到某位的值(例如把 0 读成 1),甚至无法探测到有什么东西到达了。

造成错误的原因被称为故障(fault)。无疑,找到是什么引起了错误是很重要的。例如,不好的传输介质可能很容易使得数据包被破坏。在这种情况下要解决故障是相对容易的,但是无线网络中的传输错误可能是由恶劣的天气条件引起的。要改变天气来减少或防止错误是不可能的。

建立一个可靠的系统与控制故障紧密相关。防止、解决和预报故障三者之间是有差别的(Laprie 1995)。对我们来说最重要的问题是容错(fault tolerance),它意味着系统即使在发生故障时也能提供服务。

故障通常被分为暂时的、间歇的和持久的。暂时故障(transient fault)只发生一次,然后就消失了,即使重复操作也不会发生。一只鸟从微波传输的电波中飞过可能会使一些网络上的数据丢失。如果传输超时重发,第二次就会正常工作。

间歇故障(intermittent fault)发生,消失不见,然后再次发生,如此反复进行。连接器接触不良通常会造成间歇故障。间歇故障会造成情况的恶化,因为它们很难诊断,通常,

当解决故障的人到来时系统工作良好。

持久故障(permanent fault)是那些直到故障组件被修复之前持续存在的故障。芯片燃烧、软件错误和磁盘头损坏都是持久故障的例子。

7.1.2 典型故障

发生故障的系统不能充分地提供所设计的服务。如果我们把分布式系统视为一个彼此之间且与它们的客户进行通信的服务集,那么不能充分提供服务就意味着服务器、通信通道或两者都不能正常地进行工作。但是,出现故障的服务器本身并不总是我们应该查找错误的地方。如果这样的服务器要依赖其他服务器才能充分提供它的服务,那么错误的原因就可能需要到别的地方去寻找。

这样的依赖关系大量出现在分布式系统中。如果一个文件服务器的设计目的是提供高度可用的文件系统,那么失效的磁盘会使该服务器运行困难。如果这样的文件服务器是分布式数据库的一部分,那么整个数据库的正常工作都将被打乱,因为只有一部分数据真正可以访问。

为了更好地理解故障到底有多严重,人们开发了一些分类方法。其中一种方法如图7.1所示,它基于(Cristian 1991)和(Hadzilacos 和 Toueg 1993)中描述的方法。

故障类型	说明
崩溃性故障	服务器停机,但是在停机之前工作正常
遗漏性故障	服务器不能响应到来的请求
接收故障	服务器不能接收到来的消息
发送故障	服务器不能发送消息
定时故障	服务器的响应在指定的时间间隔之外
响应故障	服务器的响应不正确
值故障	响应的值错误
状态转换故障	服务器偏离了正确的控制流
随意性故障	服务器可能在随意的时间产生随意的响应

图 7.1 不同类型的故障

如果服务器过早停机但在停止之前工作正常,就发生了崩溃性故障。崩溃性故障的一个重要方面是,一旦服务器停机,就不再提供任何服务。崩溃性故障的一个典型例子是操作系统崩溃,这时只有一种解决方法:重新启动。虽然人们期望个人计算机系统能正常运行,但它经常遭遇崩溃性故障。在这个意义上说,把复位按钮从机箱背后移到前面是很有道理的。可能有一天还会把它移到背后,甚至完全去掉。

当服务器不能对请求进行响应时就发生遗漏性故障。导致这种问题可能有多种原因。在发生接收遗漏性故障的情况下,首先服务器可能永远不会接收到请求。注意,这可能是由于尽管在客户和服务器之间正确地建立起连接,但是没有线程监听到来的请求。接收遗漏性故障通常不会影响到服务器的当前状态,因为服务器不知道有信息发送给了它。

与之类似,如果服务器能正常工作,但是在发送响应时失败,则产生发送遗漏性故障。例如,当发送缓冲区溢出而服务器又没有为这样的情况做好准备时就发生此类故障。注意,与接收遗漏性故障,服务器现在的状态可能说明它已经完成了对客户的服务。因此,如果响应发送失败,那么服务器可能需要为客户重新发送先前的请求而做好准备。

另外一种遗漏性故障与通信无关,它可能由诸如无限循环或不正确的内存管理的软件错误引起,这时服务器的状态被称为“挂起”。

另外一种故障类型与时间有关。如果响应是在指定的实间隔之外,就发生了定时故障。像我们在第 2 章中介绍的同步数据流那样,如果提供数据的速度过快,而在接收者又没有足够的缓冲空间来保存所有到来的数据,那么就很容易在接收端引起问题。但是,更通常的情况是服务器的响应太慢,这种情况被称为发生了“性能故障”。

一种严重的故障类型是响应故障,就是说服务器的响应不正确。可能发生两种响应故障。在值故障的情况下,服务器为请求提供错误的响应,例如搜索引擎系统返回了与使用的搜索项无关的 Web 页面,就是这种情况。

另外一种响应故障被称为状态转换故障。当服务器对到来的请求做出意想不到的响应时就发生这种故障。例如,如果服务器接收到一个它不能识别的信息,也没有采取措施来处理这样的信息,那么就会发生状态转换故障。特别是,发生故障的服务器可能会错误地采取一种从来没有初始化的默认行为来进行处理,此时就发生这种故障。

最严重的故障是随意性故障,也被称为拜占庭故障。实际上,当发生随意性故障时,客户应该做好最坏的准备。特别是,服务器可能产生它从来没有产生过的输出,但是又不能检测出错误。更坏的情况是发生故障的服务器恶意地与其他服务器共同工作来产生恶意的错误结果。这种情况说明了为什么谈到可靠系统时安全被认为是一个重要的需求。术语“拜占庭”是指拜占庭帝国,它存在的时间是 330 到 1453 年,地点在巴尔干半岛和现在的土耳其,当时在统治阶级中充斥着无休止的阴谋诡计和谎言。拜占庭故障的问题由 Pease 等(1980)和 Lamport 等(1982)首先进行了分析。我们将在下面对此类故障进行分析。

随意性故障与崩溃性故障紧密相关。崩溃性故障的定义在上面已经给出,它是服务器停机的最“好”方式。它也被称为故障停机故障。实际上,因故障停机的服务器只是简单地停止产生输出,而且它的停机能够被其他的进程探测到。例如,服务器可以“友好地”宣布它将要崩溃。

当然,在现实中,服务器通过呈现遗漏性故障或崩溃性故障而停机,它们并不“友好地”宣布将要停机,而要等其他的进程来判定服务器是否已经过早停机。但是,在这样的故障沉默系统中,其他的进程可能错误地认为服务器已经停机,而服务器只不过是意外地慢,也就是说发生了性能故障。

最后,服务器也有可能产生随机的输出,但是这个输出能简单地被其他进程识别为垃圾。这时服务器以一种“仁慈”的方式来出现随意性故障,这些故障也被称为安全失败(fail-safe)。

7.1.3 使用冗余来掩盖故障

如果系统是容错的,那么它能做的最好的事情就是对其他进程隐藏故障的发生。关键技术是使用冗余来掩盖故障。有三种可能:信息冗余、时间冗余和物理冗余,请参阅文献(Johnson 1995)。在信息冗余中,添加额外的位可以使错乱的位恢复正常。例如可以在传输的数据中添加一段 Hamming 码来从传输线路上的噪声中恢复数据。

在时间冗余中,执行一个动作,如果需要就再次执行。使用事务(在第 5 章中已经介绍)是这种方法的一个例子。如果一个事务中止,那么它就可以无害地重新执行。当错误是临时性或间歇性时,时间冗余特别有用。

在物理冗余中,通过添加额外的装备或进程使系统作为一个整体来容忍部分组件的失效或故障成为可能。物理冗余可以在硬件上也可以在软件上进行。例如,可以在系统中添加额外的进程,这样如果少数进程崩溃,系统还是可以正常工作。换句话说,通过冗余的进程可以获得高度容错性。

物理冗余是提供容错性的著名技术。它用在生物学(哺乳动物具有两只眼睛、两个耳朵、两个肺等)、飞行器(747 具有四个引擎,而只使用三个)和体育(有多个裁判)中。物理冗余用在电子电路的容错中已经有多年了,了解如何在电子电路中使用它是有意义的。例如,考虑图 7.2(a)中的电路。这里的信号依次通过设备 A、B 和 C。如果它们中的一个发生故障,最后结果就可能是错误的。

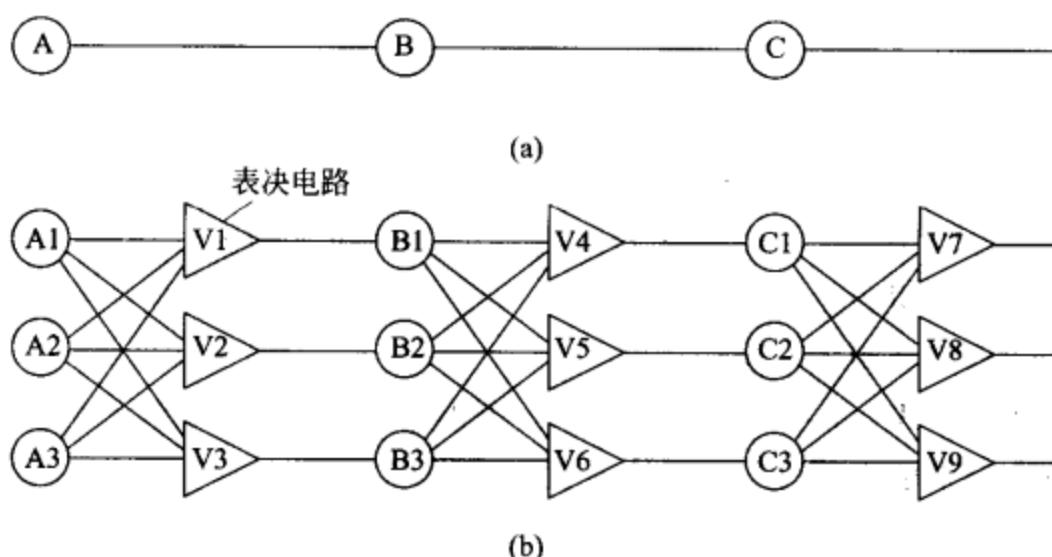


图 7.2 三倍的模块冗余

在图 7.2(b)中,每个设备都被复制了三个。这样电路中的每一级都有三个表决电路,每个表决电路都是具有三个输入和一个输出的电路。如果输入中的两个或三个是相同的,那么输出就等于输入,如果所有的输入都是不同的,那么输出就是未定义的。这种设计被称为 TMR(三倍模块冗余,Triple Modular Redundancy)。

假定元件 A2 发生故障,则每个表决电路 V1, V2 和 V3 获得两个好的(一致的)输入和一个无用的输入,它们中的每个都将正确值输出到第二级。这样 A2 故障的影响就完全被掩盖了,到 B1, B2 和 B3 的输入就与没有发生故障时完全相同。

现在考虑一下如果除了 A2 之外 B3 和 C1 也都发生了故障时的情形。即使发生这种

情况，其影响也会被掩盖，三个最后的输出还是正确的。

为什么在每一级都需要三个表决电路呢？毕竟一个表决电路就可以发现并传递主要的观点。但是，一个表决电路只是一个组件，它可能发生故障。例如，假定 V1 发生故障，到 B2 的输入就是错误的，但是只要别的部分正常工作，B2 和 B3 就会产生相同的输出，V4, V5 和 V6 都会产生到第三级的正确结果。V1 发生故障的影响与 B1 故障没什么区别，在两种情况下 B1 都产生错误的输出，但是两种情况下都在后面才进行表决。

尽管不是所有的容错分布式系统都使用 TMR，但是这种技术还是很普遍的，它对容错系统是什么给出了一个清楚的解释。它与使用高度可靠的单个组件的系统不同，后者的组织不是容错的。当然，TMR 也可以递归使用，例如在芯片内部使用 TMR 来使它高度可靠，而对于使用芯片的电路设计者来说，他们并不知道芯片的冗余性。

7.2 进程恢复

已经讨论了容错的基本问题，现在我们集中讨论在分布式系统中如何获得容错性的问题。我们首先要讨论的主题是防止进程失败，这是通过把进程复制到组中来获得的。在下面的内容中，我们考虑进程组中的一般设计问题，并讨论容错组到底是什么。我们也将讨论当一个或多个程序不能给出正确的回答时，如何在进程组中达到一致。

7.2.1 设计问题

容忍失败进程的关键方法是把多个同样的进程组织到一个组中。所有组都具有的关键特性是当信息发送到组本身时，组中的所有成员都接收它。通过这种方式，如果组中的一个进程失败，其他的一些进程可以接管它(Guerraoui, Schiper 1997)。

进程组可以是动态的。可以创建新的组也可以删除旧的组。在系统操作过程中，一个进程可以加入一个组也可以离开一个组。一个进程可以同时是多个组的成员。因此需要一些机制来管理组和组的成员。

组类似于社会组织。Alice 可能是一个读书俱乐部、一个网球俱乐部和一个环境组织的成员。在一个特别的日子里，她可能从读书俱乐部收到一本生日蛋糕食谱的邮件(消息)，从网球俱乐部收到一年一度的母亲节网球锦标赛的消息，还从环境组织收到保护南方土拨鼠运动开始的消息。在任何时刻，她都可以自由地离开任何一个或所有这些组，也可能加入其他的组。

引入组的目的在于允许把进程的集合作为单一的抽象概念来处理。这样，一个进程就可以把消息发送给一个服务器组而不用知道有多少个进程以及它们在哪里，而这些可以在两次调用之间进行改变。

1. 平等组与等级组

不同的组之间一个重要的区别在于它们的内部结构。在一些组中，所有的进程都是平等的。没有指挥，所有的决定都是共同做出的。在其他的组中存在一些等级关系。例如，一个进程是协调者而其他进程都是工作者。在这种模式中，当外部客户或一个工作者

产生一个工作请求时,请求被发送给协调者。协调者决定哪个工作者最适合,然后把请求转发给它。当然也可能有更复杂的等级关系。在图 7.3 中说明了这些通信模式。

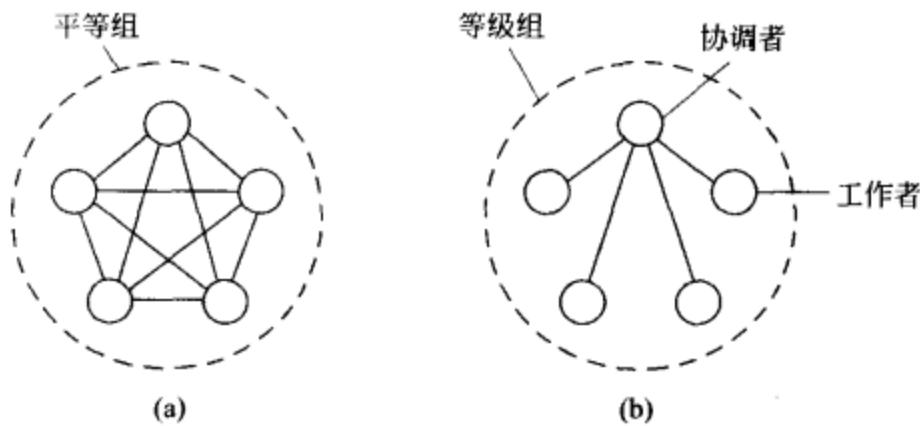


图 7.3 平等组中和简单等级组中的通信

(a) 平等组中的通信; (b) 简单等级组中的通信

每种组织都具有它的优点与缺点。平等组是对称的,没有单独的失败点。如果一个进程崩溃,组只是简单地变得更小,但是还可以继续。它的缺点在于做出决定比较复杂。例如,要决定某事常常需要进行表决,这导致了一些延迟和开销。

等级组则相反。协调者的故障会使整个组崩溃,但是只要它保持运行,就可以独自做出决定,不需要其他进程参加。

2. 组成员

当组通信发生时,需要一些方法来创建和删除组,以及允许进程加入和离开组。一种可能的方法是使用组服务器(group server),所有这些请求都发送给它。这个组服务器保持着所有组及其成员的完整的数据库。这种方法直接、有效、容易实现。不幸的是它具有所有集中式技术的缺点:单一的失败点。如果组服务器崩溃,组管理就不再存在。可能大多数或所有组都不得不从头开始重建,所有正在进行的工作都可能终止。

与之相反的方法是以分布式的方法来管理组成员。例如,如果可靠的多播可用,一个外部进程就可以发送消息给所有的组成员表示它希望加入该组。

要离开一个组,理想的情况是,成员只需要给所有成员发送一个再见消息。在容错环境中,故障停机的语义一般并不确切。问题在于当一个进程崩溃时不能像一个进程自动离开那样进行通知。其他成员不得不通过注意到崩溃的成员不再进行响应来发现这一点。一旦确定崩溃的成员是真正的崩溃(而不是速度慢),就从组中删除它。

另外一个棘手的问题是进程的离开与加入必须跟数据消息的发送同步。换句话说,从进程加入一个组开始,它就必须接收发送给该组的所有消息。与之类似,一旦进程离开一个组,它就不能接收来自该组的任何消息,其他的成员也不能接收来自它的任何消息。一种确保把进程的加入或离开集成到消息流中的正确位置的方法,是把这个操作转换为一个发送到整个组的消息序列。

与组成员相关的最后一个问题是如果很多机器停机使得组完全不能工作时该怎么办。这时需要通过一些协议来重建组。一些进程不得不进行初始化来重新启动,但是如

果有两个或三个进程同时进行尝试会发生什么？协议必须能够处理这些问题。

7.2.2 故障掩盖和复制

进程组是建立容错系统方法的一部分，特别是在有一组相同的进程允许我们掩盖组中的一个或多个发生故障的进程的情况下。换句话说，我们可以复制进程并把它们组织在一个组中来用一个容错的组取代一个脆弱的进程。像前面讨论过的那样，有两种方法来进行复制：通过基于主进程的协议或通过复制写协议。

在容错的情况下，基于主进程的复制通常以主进程后备协议的形式出现。在这种情况下，以等级方式来组织一个进程组，其中一个主进程协调所有的写操作。在实践中，这个主进程是固定的，尽管如果需要可以用一个后备进程来接管它。实际上，如果主进程崩溃，后备进程执行一些选举算法来选择一个新的主进程。

像我们在上一章中解释的那样，与基于团体的协议相同，复制写协议以主动复制的形式使用。这些解决方法把相同进程的集合组织到一个平等组中。这样做的主要优点在于这样的组没有单一的失败点，代价是分布式的协调。

使用进程组进行容错中一个重要的问题是需要多少复制。为了简化我们的讨论，我们只考虑复制一写系统的情况。如果系统能够经受 k 个组件的故障并且还能满足规范的要求，那么就被称为 k 容错(k fault tolerant)。如果这些组件(进程)是失败沉默的，那么具有 $k+1$ 个组件就足够提供 k 容错。如果 k 个组件停止工作，还可以用剩下的一个组件来得到响应。

另一方面，如果进程发生拜占庭失败，继续错误运行并发送出错误或随机的应答，那么最少需要 $2k+1$ 个进程才能获得 k 容错。在最坏的情况下， k 个失败的进程可以意外地(甚至是故意地)产生同样的应答，但是剩余的 $k+1$ 个进程也会产生同样的回答，这样，客户或表决电路还可以相信多数进程的回答。

当然，说一个系统是 k 容错的，让 $k+1$ 个相同的进程以票数胜过 k 个相同的应答在理论上是好的，但是在实践中很难想象这样的情况，即 k 个进程失败了，但是 $k+1$ 个进程没有失败。这样，即使在容错系统中也需要某种统计分析。

与这种模式有关的一个不明显前提假设是所有请求按相同的顺序到达所有的服务器，这个问题被称为原子多播问题(atomic multicast problem)。实际上，这个条件可以稍微放松，因为读操作不会有问题而一些写操作也可以替换，但是还留有一般的问题。原子多播将在后面详细讨论。

7.2.3 故障系统的协议

把复制的进程组织在一个组中有助于提高容错性。正如我们提到的那样，如果客户可以根据它的表决机制做出决定，我们甚至可以容忍 $2k+1$ 个进程中的 k 个进程得出错误的结果。但是，我们做出的假定是这些进程没有组合成团队来共同产生错误的结果。

通常，如果我们要求进程组达成一致，事情会变得更复杂。在很多情况下需要达到一致，例如，选择一个协调者，决定是否提交一个事务，在工作者之间划分任务以及同步。当通信和进程都很完美时，达成这样的一致通常是简单而又直接的，但是如果不是这样，就

会出现问题。

分布式协议算法的一般目标是使所有的非故障进程就一些问题达成一致，而且在有限的步骤内就达成一致。根据系统参数的不同可能有不同的情况，包括通信是否可靠或进程崩溃性失败的语义。

在考虑故障进程的情况之前，我们先看简单的情况：进程运行很好但是通信线路丢失消息。有一个称为两军问题(two-army problem)的著名问题，它说明了在两个运行良好的进程之间即使只就 1 位信息达成协议也是非常困难的。红军具有 5000 个士兵，在一个山谷中扎营。两支蓝军各有 3000 人，分别在山谷两边的山坡上扎营。如果两支蓝军可以协调他们对红军的攻击，那么他们就能取得胜利。但是如果只有一支蓝军独自发动攻击，则会被消灭。蓝军的目的是就攻击达成协议。他们只能使用不可靠的方法进行通信：派一个可能会被红军抓获的使者。

假定蓝军 1 的司令官 Alexander 发送一个消息给蓝军 2 的司令官 Bonaparte：“我有一个计划——我们在明天凌晨发起攻击。”使者到达蓝军 2，Bonaparte 让他带回一个消息：“好主意，明天凌晨见。”使者安全回到他的基地传达了这个消息，Alexander 通知他的士兵准备在凌晨进行战斗。

但是后来 Alexander 意识到 Bonaparte 并不知道使者是否安全返回，而不知道这一点就可能不敢发动攻击。然后 Alexander 让使者去告诉 Bonaparte 他(Bonaparte)的消息已经到达，可以进行战斗。

使者再次穿过山谷传达这个确认。但是现在 Bonaparte 在担心 Alexander 不知道确认是否能够到达，他担心的原因是如果 Alexander 认为使者会被抓住，他就不能确认他(Alexander)的计划，可能不会冒险发起攻击，所以他派使者再次回去。

很容易看出，即使使者每次都能成功通过山谷，不论他们发送了多少确认，Alexander 和 Bonaparte 也永远不会达成协议。假定有某个可以在有限步骤内结束的协议，这是一种在结尾去掉任何额外的步骤而得到的可以工作的最小协议。现在某个消息是最后的消息，它对达成协议是关键性的(因为这是最小的协议)。如果这个消息没有到达，行动就被取消。

但是，最后一个消息的发送者并不知道最后一个消息是否到达。如果没有到达，就没有完成协议，其他的司令官也不会发起攻击。由于最后一个消息的发送者不能知道是否一定发动攻击，所以也就不能调动他的军队。既然最后一个消息的接收者知道发送者不能确认，他也就不会冒险，因为没有达成协议。在不可靠传输的条件下，即使是无错误的进程，在两个进程之间达成协议也是不可能的。

现在我们假定通信良好，但是进程却并不好。这里也有一个经典的军事问题称为拜占庭将军问题(Byzantine generals problem)。在这个问题中，红军还是在山谷中扎营，但是在附近的山上有 n 个带领部队的蓝军将领。通信是通过电话双向进行的，及时而且质量很好。但是有 m 个将军是叛徒(故障)，他们通过给忠诚的将军发送错误的和矛盾的信息(模拟故障进程)来阻止他们达成协议。现在问题在于忠诚的将军是否还能达成协议。

为了一般化，我们在这里按一种稍微不同的方式来定义协议。假定每个将军都知道他有多少士兵，将军的目的是交换兵力，这样在算法结束时每个将军都得到一个对应于所

有军队的兵力情况的长度为 n 的向量。如果将军 i 是忠诚的,那么他的兵力中的元素就为 i ,否则就是未定义的。

Lamport 等(1982)建议使用一种递归算法,这种算法可以在一定条件下解决这个问题。在图 7.4 中说明了在 $n=4$ 而 $m=1$ 的情况下这种算法的工作情况。对这样的参数来说,这个算法需要进行 4 步操作。在第 1 步中,每个将军发送一个(可靠的)消息给其他所有将军以宣布他的兵力,忠诚的将军说的是实话;而叛徒则向别的将军说谎。在图 7.4(a)中,我们看到将军 1 报告有 1000 兵力,将军 2 报告有 2000 兵力,将军 3 向每个人都说谎,分别给出 x 、 y 和 z ,将军 4 报告有 4000 兵力。在第 2 步中,第 1 步里宣布的结果按图 7.2(b)中的向量格式收集在一起。

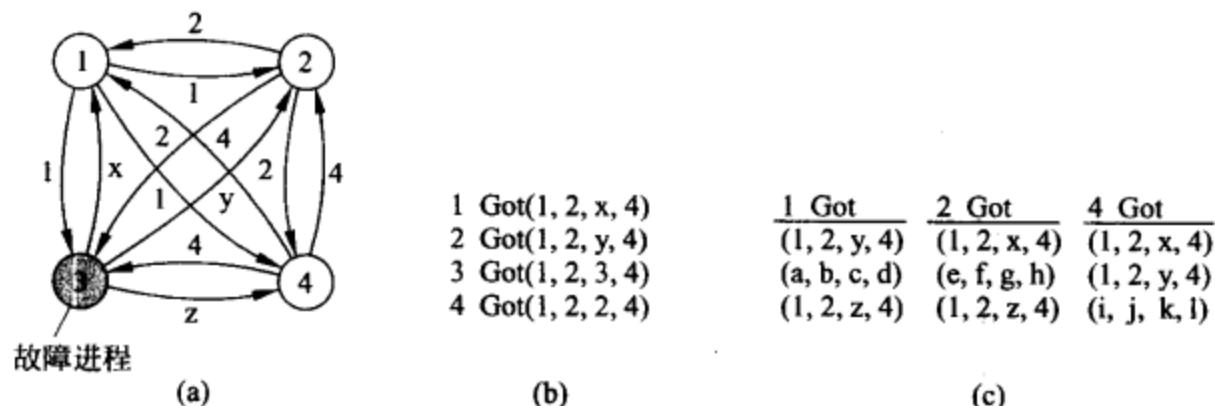


图 7.4 有三个忠诚的将军和一个叛徒的拜占庭将军问题

(a) 将军宣布他们的兵力(以千为单位);(b) 在(a)的基础上每个将军的向量;(c) 第 3 步中每个将军接收到的向量

第 3 步中,每个将军把他从图 7.4(b)中得到的向量传递给其他所有将军。每个将军都得到来自其他三个将军的三个向量。将军 3 在这里还是说了谎,编造了 12 个新的值。第 3 步的结果在图 7.4(c)中显示。最后在第 4 步中,每个将军都检查新收到的向量中的第 i 个元素。如果哪个值为多数,就把它放到结果向量中。如果没有值为多数,那么结果向量中对应的元素就被标记为 UNKNOWN。从图 7.4(c)中,我们看到将军 1、2 和 4 都对正确结果

$(1, 2, \text{UNKNOWN}, 4)$

达成了一致。叛徒不能破坏忠诚的将军的信息,因此他也不能破坏这个工作。

现在让 $n=3$ 、 $m=1$,来重新看一下这个问题,也就是说只有两个忠诚的将军和一个叛徒,像图 7.5 说明的那样。我们在图 7.5(c)中看到没有一个忠诚的将军能看到占多数的元素 1、元素 2 和元素 3,所以它们都被标记为 UNKNOWN。算法无法达成一致。

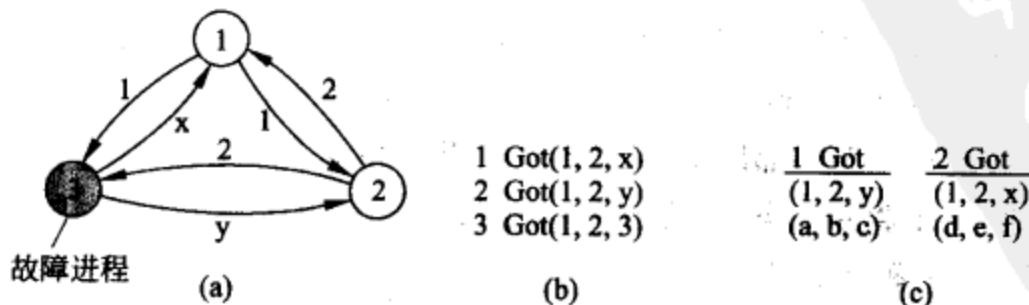


图 7.5 除了有两个忠诚的将军和一个叛徒之外与图 7.4 相同

Lamport 等(1982)在他的论文中证明在具有 m 个故障进程的系统中,只有存在 $2m+1$ 个正确工作的进程才能达成协议,这样总共就有 $3m+1$ 个进程。换句话说,只有当三分之二以上的进程正常工作时才可能达成协议。

下面换一个角度来看这个问题。基本上,我们需要得到的就是在一个忠诚将军组里的多数选票,而不管在他们中是否有叛徒。即使有 m 个叛徒,他们与那些已经被叛徒误导的忠诚者在一起表决,我们也要确保表决的结果是忠诚者的票数占多数。具有 $2m+1$ 个忠诚者,就可以达到这一点,因为只有当三分之二以上的选票相同才能达成协议。换句话说,如果三分之二以上的将军做出同样的决定,那么这个决定就符合在忠诚将军组的表决中占多数的原则。

不幸的是,达成协议的情况甚至会更坏。Fischer 等(1985)证明了在不能保证消息在已知的有限时间内交付的分布式系统中,即使只有一个进程故障(虽然该进程发生的是沉默性故障)也不可能达成协议。这种系统中的问题在于无法区分任意慢的进程与崩溃的进程。很多其他的理论的研究成果是关于何时可以达成协议,何时不可以达成协议的。有关这些成果的调查在文献(Barborak 等 1993)和(Turek, Shasha 1992)中给出。

7.3 可靠的客户-服务器通信

在很多情况下,分布式系统中的容错关注的是故障进程。但是我们也需要考虑通信故障。前面讨论过的大多数故障模型都很好地应用到通信通道上。特别是,通信通道也可能出现崩溃性故障、遗漏性故障、定时故障和随意性故障。在实践中建立可靠的通信通道时,焦点应该放在掩盖崩溃性故障与遗漏性故障上。随意性故障可能以重复消息的形式发生,造成它的原因是在计算机网络中可能把消息缓冲一个相对较长的时间,然后在原始发送者发出重新传送命令时把消息重新发到网络中(请参阅文献 Tanenbaum 1996)。

7.3.1 点到点通信

在很多分布式系统中都使用像 TCP 这样可靠的传输协议来建立可靠的点到点通信。TCP 可以通过确认和重传来掩盖遗漏性故障。这样的故障对 TCP 客户是完全隐藏的。

但是通常不能掩盖连接的崩溃性故障。崩溃性故障发生在不论什么原因 TCP 连接突然中断时,这时就不能再通过该通道传输更多的消息了。在大多数情况下,可通过抛出一个异常信号来通知客户通道已经崩溃。要掩盖这样的故障,惟一的方法就是让分布式系统尝试自动建立一个新的连接。

7.3.2 出现失败时的 RPC 语义

现在我们来看一下在使用像远程过程调用(RPC)或远程方法调用(RMI)这样的高级通信工具时的客户-服务器通信。在下面内容中,我们主要讨论 RPC,但是这些讨论也同样适用于与远程对象的通信。

RPC 的目标是通过使远程过程调用与本地过程调用看上去相同来隐藏通信。除了一些例外情况之外,现在我们已经对它相当熟悉了。确实,只要客户和服务器都正常工

作,那么 RPC 就可以很好运转。当发生错误时会出现一些问题。然后就不容易掩盖本地调用与远程调用之间的区别了。

为了使我们的讨论结构化,首先区分一下在 RPC 系统中发生的 5 种失败形式,如下所示:

- (1) 客户不能定位服务器;
- (2) 客户到服务器的请求消息丢失;
- (3) 服务器在收到请求之后崩溃;
- (4) 从服务器到客户的响应消息丢失;
- (5) 客户在发送请求之后崩溃。

每种类型的失败都会引起不同的问题,需要用不同的方法来解决。

1. 客户不能定位服务器

首先是客户可能不能定位适当的服务器。例如,服务器可能因发生故障而停止运行。作为一种选择,我们假设客户程序是使用特定的客户存根版本进行编译的,而且在相当长的时间里不使用二进制。同时服务器程序有了改进,安装了一个接口的新版本,产生了新的存根并投入使用。当客户最终运行时,绑定器可能不能使它与服务器相匹配并报告失败。这种机制用来保护客户不会意外地试图与那些与它所要求的参数以及所假定的动作并不相符合的服务器进行通信,关于如何处理这种失败的问题仍然存在。

一种可能的方法是让错误抛出一个异常,在一些语言中(例如 Java),程序员可以编写只有在发生特定错误(例如被零除)时才调用的过程。在 C 中可以使用信号处理器。换句话说,我们可以定义一种新的信号类型 57G-NOSERVER,允许它以与其他信号相同的方式进行处理。

但这种方法也有缺点。首先,不是每种语言都具有异常或信号。还有一点是不得不编写异常或信号处理器程序,这破坏了我们努力想要获得的透明性。假定你是一位程序员,你的老板要你编写一个求和的过程。你微笑着告诉她会在 5 分钟内完成编写、测试并进行文档化。然后她说你还应该编写一个异常处理程序以防万一。这时就很难再继续幻想远程过程与本地过程没有区别了,因为在单处理器系统中,为“不能定位服务器”编写异常处理器是很少见的请求。关于透明性,我们就介绍这么多。

2. 请求消息丢失

第 2 类失败类型是请求消息丢失。这是最容易处理的一种失败:只需要使操作系统或客户存根在发送请求时开启一个定时器即可。如果在返回应答或确认之前定时器超时,那么就重新发送消息。如果消息真的丢失了,则服务器不能辨别原始的与重传的消息,但每件事情都顺利进行。当然,如果请求消息很频繁地丢失,使得客户放弃重传而错误地认为是服务器停机,那么这种情况就回到了“不能定位服务器”的问题。如果请求没有丢失,我们需要做的惟一的事就是让服务器知道它正在处理重传的消息。不幸的是,这件事并不简单,我们将在讨论应答丢失时讨论它。

3. 服务器崩溃

第3个失败是服务器崩溃。服务器中事件的通常顺序如图7.6(a)所示。请求到达、执行请求、发送应答。现在考虑图7.6(b)。请求到达并且被执行,但是在发送应答之前服务器崩溃了。最后,看一下图7.6(c),请求再次到达,但是这次服务器在执行请求之前就崩溃了。

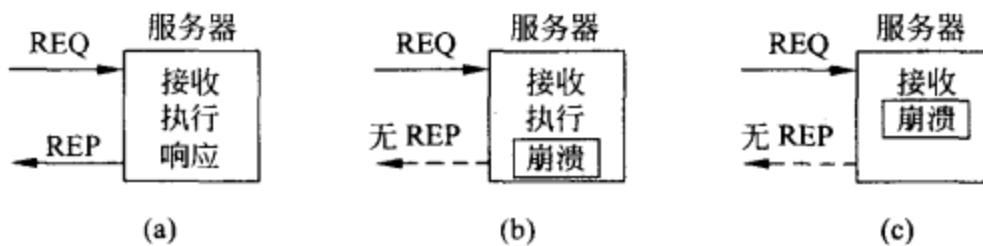


图 7.6 客户-服务器通信中的服务器
(a) 通常情况; (b) 执行之后崩溃; (c) 执行之前崩溃

图7.6中比较麻烦的部分在于正确对待(b)和(c)之间的不同。在(b)中,系统不得不向客户报告失败(也就是抛出一个异常),而在(c)中只是重新传输请求。问题在于客户的操作系统不能区分它们。它所知道的所有事情就是定时器超时。

关于如何处理这种情况存在三种方法(Spector 1982)。一种方法是在服务器重启之前(或重新绑定到一个新的服务器之前)等待并再次尝试操作。这种方法是在得到应答之前不断尝试,然后将应答传递给客户。这种技术被称为至少一次语义(at least once semantics),它保证PRC至少执行一次,但是有可能执行多次。

第二种方法是立刻放弃并报告失败。这种方法被称为最多一次语义(at most once semantics),它保证RPC最多执行一次,但是有可能一次也不执行。

第三种方法是什么都不保证。当服务器崩溃时,客户得不到有关发生了什么的任何帮助和承诺。RPC有可能执行任意多次。这种方法的主要优点在于它很容易实现。

这些方法没有一个是很吸引人的。吸引人的方法应该具有确切一次的语义(exactly once semantics),但是一般来说没有办法可以做到这一点。想象由打印一些文本组成的远程操作,当文本被打印之后服务器向客户发送一条完成消息。假定当客户提交请求时收到一个确认消息表明请求已经被提交给服务器。服务器可以采取两种策略:或者在通知打印机进行工作之前发送完成消息,或者在文本已经被打印之后发送。

假定服务器崩溃之后又恢复了。它向所有的客户宣布它刚才崩溃了但是现在又再次运行。问题在于客户并不知道它的打印请求是否被执行了。

客户可以采取4种策略。首先,客户可以决定决不重发请求,这可能要冒着文本不被打印的危险。第二,它可以决定总是重发请求,但是这种方法可能导致文本被打印两次。第三,它可能只有在没有接收到打印请求已经传送到服务器的确认时才重发请求。在这种情况下,客户认为服务器在打印请求被传送到之前崩溃。第四种也是最后一种策略是只有接收到打印请求的确认时才重发请求。

由于服务器具有两种策略而客户具有4种,所以共有8种结合方法。不幸的是,没有

一种是令人满意的。注意，在服务器上可能发生 3 种事件：发送完成消息(M)、打印文本(P)和崩溃(C)。这些事件可能以 6 种不同的次序发生：

- (1) M→P→C：在发送完成消息和打印文本之后发生崩溃；
- (2) M→C(→P)：在发送完成消息之后，打印文本之前发生崩溃；
- (3) P→M→C：在发送完成消息和打印文本之后发生崩溃；
- (4) P→C(→M)：首先打印文本，然后在发送完成消息之前发生崩溃；
- (5) C(→P→M)：在服务器做任何事之前发生崩溃；
- (6) C(→P→M)：在服务器做任何事之前发生崩溃。

括号表示由于服务器已经崩溃所以不会发生该事件。图 7.7 说明了所有可能的组合。很容易验证，没有一种客户策略和服务器策略的结合可以在所有可能的事件顺序下正确工作。根本在于客户从来都不能知道服务器是在打印文本之前还是之后发生崩溃。

客户		服务器					
		策略 M→P			策略 P→M		
重发策略	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)	
	DUP	OK	OK	DUP	DUP	OK	
	OK	ZERO	ZERO	OK	OK	ZERO	
	DUP	OK	ZERO	DUP	OK	ZERO	
	OK	ZERO	OK	OK	DUP	OK	

OK=文本被打印一次
DUP=文本被打印两次
ZERO=文本根本没有被打印

图 7.7 出现服务器崩溃时客户与服务器策略的不同结合

简而言之，服务器崩溃的可能性从根本上改变了 RPC 的本性，很明显地把单处理器系统与分布式系统区别开来。在前者中，服务器崩溃也暗示着客户的崩溃，所以恢复不但不可能的也是不需要的。而在后者中有可能也有必要采取行动。

4. 应答消息丢失

应答丢失处理起来也比较困难。直接的解决方法就是再次依赖客户的操作系统设置的定时器。如果客户在合理的时间内没有得到响应，那么就再次发送请求。这种方法的问题在于客户不能真正确认为什么没有得到响应。是请求或响应丢失还是服务器慢？这造成了一些不同。

特别是，一些操作可以安全地重复多次而不会造成任何损害。例如请求一个文件的前 1024 个字节就没有副作用，可以执行多次而没有危害。这种请求被称为是幂等的 (idempotent)。

现在考虑请求银行服务器从一个账户向另一个账户转账 100 万美元的请求的情况。如果请求到达并执行了，但是响应丢失了，而客户不知道这一点，它就会重发消息。银行服务器将把这个请求解释为一个新的请求，并再次执行它。这样就转账了 200 万美元。

想象一下这种响应丢失 10 次吧！传输货币不是幂等的。

解决这个问题的一种方法是按幂等的方式组织所有的请求。但是在实际中，很多请求（例如货币兑换）在本质上是非幂等的，所以就需要一些别的东西。另外一种方法是在客户为每个请求分配一个序列号。通过在服务器上跟踪从每个客户收到的最近序列号，服务器可以分辨原始的请求与重发的请求并拒绝执行第二次发出的请求。但是服务器还是要向客户发送响应。注意这种方法需要服务器维护对每个客户的管理。一个附加的防卫措施是在消息头中添加 1 位以区分是最初的请求还是重发的请求（这种思想在于执行原始请求是安全的，而执行重发请求就需要更多的小心）。

5. 客户崩溃

最后一种失败是客户崩溃失败。如果客户向服务器发送请求，请求做一些事情，但在服务器回复之前崩溃了，那么会发生什么呢？这时，虽然计算是活动的，但没有双亲等待结果，这种不需要的计算被称为孤儿（orphan）。

孤儿会引起多种问题。它们至少是在浪费 CPU 周期。它们也可以锁定文件或占用有价值的资源。最后，如果客户重新启动并再次进行 RPC，而孤儿的结果又立即返回了，那么就会造成结果的混淆。

如何处理孤儿呢？Nelson(1981)提出了 4 种方法。第一种方法是在客户存根（stub）发送 RPC 消息前进行日志记录来说明要做什么。日志被保留在崩溃之后仍然存在的磁盘或其他介质上。在重新启动之后，对日志进行检查然后明确地杀死孤儿。这种方法被称为消灭（extermination）。

这种方法的缺点在于为每个 RPC 都进行写磁盘操作要付出的昂贵代价。另外它也可能不能工作，因为孤儿本身也可以进行 RPC，这样就创建了更多的后代，因而对它们进行定位很难，甚至是不可能的。最后，网络可能由于网关故障而被分隔，这使得即使定位了它们，要杀死它们也是不可能的。总之，这不是一种有前途的方法。

在称为再生（reincarnation）的第二种方法中，不需要书写磁盘记录就可以解决所有这些问题。它的工作方式是把时间分为顺序编号的时期。当客户重启时，就向所有的机器广播一个消息说明一个新时期的开始。当这样的广播到达时，所有与那个客户有关的远程计算都被杀死。当然，如果网络被分隔，那么一些孤儿还能活下来。但是，当它们向回报告计算结果时，其中包含一个过期的时期号，这就使得检测到它们很容易。

第三种方法是第二种方法的变种，但是没那么严厉，它被称为优雅再生（gentle reincarnation）。当时期广播到达时，每台机器都进行检查来查看是否存在远程计算，如果有，那么就尝试定位它的拥有者。只有当不能找到拥有者时才杀死该计算。

最后一种方法称为到期（expiration），其中每个 RPC 都被给定一个标准的时间量 T 来进行工作。如果到时不能结束，那么就必须显式地请求另外的时间量。这是很麻烦的。另一方面，如果在崩溃之后，客户在重启之前等待了时间 T，那么就可以确认抛弃所有的孤儿。这里要解决的问题是选择一个合理的 T 值以满足不同 RPC 的需要。

在实践中，这些方法都令人满意。更糟的是，杀死一个孤儿可能带来无法预料的结果。例如，假定一个孤儿获得了一个或多个文件或数据库记录上的锁。如果突然杀死孤

儿,那么这些锁就会被永远保持。而且孤儿可能已经在不同的远程队列中设置了实体以便将来某个时候启动其他进程,这样即使杀死孤儿也不能删除它的所有痕迹。在Panzieri 和 Shrivastava (1988)中更详细地讨论了孤儿消灭问题。

7.4 可靠的组通信

如果考虑一下由复制获得的进程恢复有多重要,那么就会知道可靠的多播服务也是重要的了。多播服务保证了消息被传送给进程组中的所有成员。不幸的是,可靠多播的实现被证明是令人惊讶的困难。在本节中,我们更深入地来探讨与把消息可靠地传递到一个进程组相关的问题。

7.4.1 基本的可靠多播方法

尽管大多数的传输层都提供了可靠的点到点通道,它们很少能提供到一组进程的可靠通信。最好的也只是让每个进程都建立到想要与之通信的每个其他进程的点到点连接。显然,这样的组织不是有效的,因为它浪费了网络带宽。不过,如果进程数目较少,通过多个可靠的点到点通道来获得可靠性是一种简单而直接的方法。

要做到这一点,我们需要准确地定义什么是可靠多播。直观地说,它意味着发送到一个进程组的消息被传递给该组中的每个成员。但是,如果在通信期间有一个进程加入该组会发生什么呢?这个进程也应该接收消息吗?同样,我们也应该确定如果一个(发送)进程在通信期间崩溃会发生什么?

要解决这样的问题,就必须区分存在故障进程时的可靠通信与假定所有进程都正确操作时的可靠通信。在前一种情况下,如果能保证所有正常的组成员都接收到消息,那么就认为多播是可靠的。其中的窍门在于除了不同的次序约束之外,还应该就在传递消息之前组看上去是什么样子达成协议。我们在下面讨论原子多播,然后再回到这个问题上来。

如果假定存在一个协议说明谁是组的成员,那么问题就会简单一些。特别是,如果我们假定进程不会失败,而且在通信进行期间不会有进程加入或离开组,那么可靠多播就简单地意味着每个消息都应该被传递到组的每个当前成员处。在最简单的情况下,不要求所有的组成员都按同样的顺序接收消息,但是有时需要这个特性。

只要接收者的数目有限,那么这种较弱形式的可靠多播实现起来就相对容易。考虑一下单个发送者想要把一个消息多播发送给多个接收者的情况。假定底层的通信系统只需要不可靠的多播,那么这意味着多播消息会以某种方式丢失,它只能传送给其中一些接收者而不是全部。

一种简单的方法如图 7.8 所示。发送进程为它发送的每个多播消息分配一个序列号。假定消息按它们被发送的次序进行接收。在这种方式中,接收者很容易探测到消息丢失。每个多播消息都在发送者本地的一个历史缓存器中进行存储。假定发送者知道接收者,那么发送者就简单地在每个接收者都返回一个确认之前在历史缓存器中保留消息。如果接收者探测到它丢失了一个消息,那么就返回一个否定确认,请求发送者重发。作为

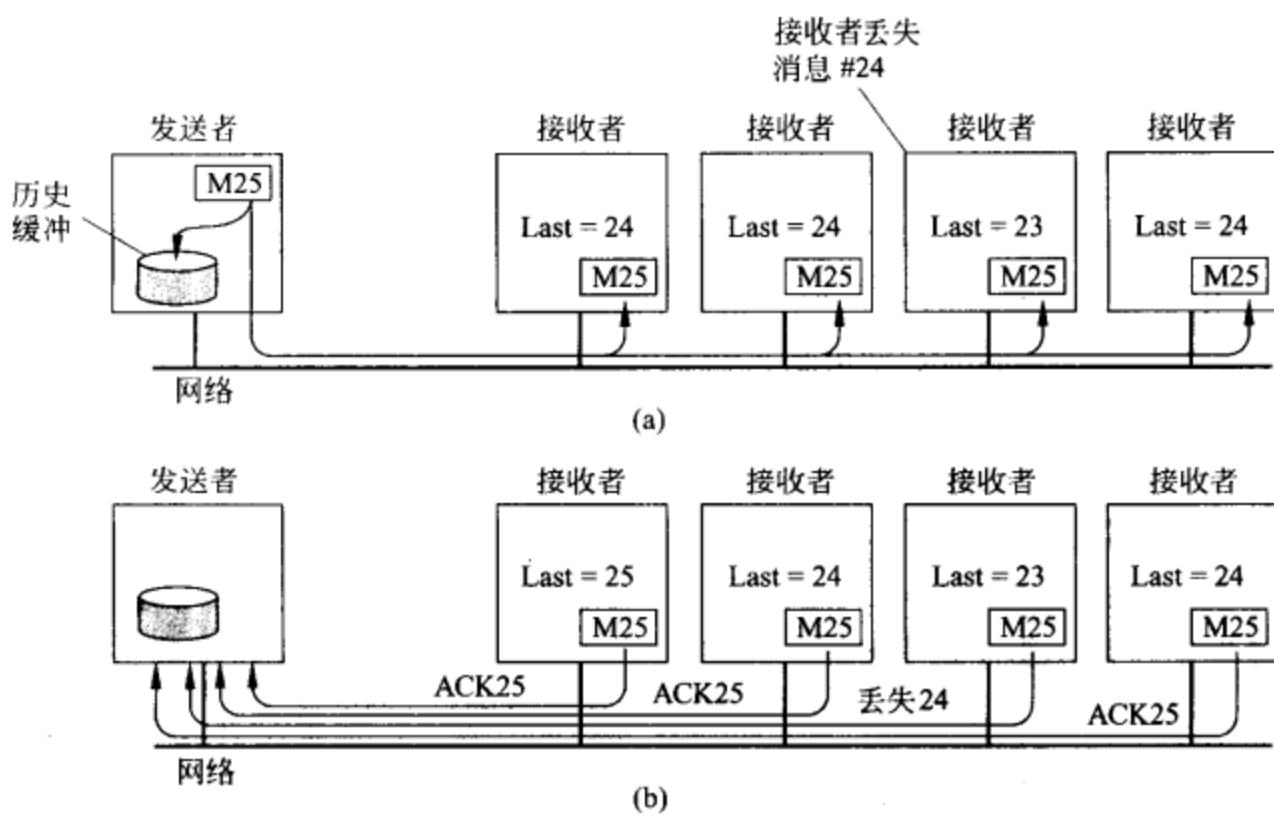


图 7.8 当所有的接收者已知而且假定不会失败时的简单的可靠多播方法
(a) 消息传递; (b) 反馈

选择,发送者可以在某个时间内没有接收到所有确认的情况下自动重发消息。

可以在多种设计中进行平衡。例如,要减少返回给发送者的消息数目,确认可能被附加在其他的消息中。重发消息也可以使用到每个请求进程的点到点通信来进行,或使用发送到所有进程的单一多播消息。要了解更多的细节,请参阅文献(Chang 和 Maxumchuk 1984)。

7.4.2 可靠多播中的可扩展性

刚才介绍的可靠多播方法的主要问题是它不支持有很多接收者的情况。如果有 N 个接收者,那么发送者必须准备接收至少 N 个确认。如果有很多的接收者,那么发送者可能被大量的反馈消息淹没,这称为反馈拥塞。另外,我们还需要考虑接收者可能发布在广域网上的情况。

解决这个问题的一种方法是:接收者不对消息接收进行反馈,而是只在通知发送者消息丢失时才返回一个反馈消息。只返回否定确认通常可以减少反馈的规模(例如,请参阅 Towsley 等 1997),但是并不能保证永远不会有反馈拥塞。

只返回否定确认在理论上还存在另外一个问题,发送者不得不永远在历史缓存器中保留消息。因为发送者永远不会知道消息是否已被传送到所有的接收者,它应该总是准备处理来自接收者的要求重发旧消息的请求。在实践中,发送者经过一段时间之后就从历史缓存器中删除消息以避免缓存器溢出。但是,删除一条消息是在冒险,因为重发的请求有可能不能得到响应。

对于可扩展的可靠多播有若干种建议,可以在(Levine, Garcia-Luna-Aceves 1998)中

找到对不同方法的比较。我们现在只简单地讨论两种非常不同的方法,它们在现存的很多方法中是具有代表性的。

1. 无等级的反馈控制

要获得可扩展的可靠多播方案,关键问题在于减少返回给发送者的反馈消息的数目。在一些广域网应用中流行的一种模式是反馈抑制(feedback suppression)。这种方法是 Floyd 等(1997)所开发的 SRM(可扩展可靠多播,Scalable Reliable Multicasting)协议的基础,它以以下方式进行工作:

首先,在 SRM 中,接收者从来都不会对多播消息的成功传送进行确认,而是只在丢失消息时进行报告。探测消息如何丢失的问题留给应用程序处理。只有否定确认作为反馈返回。当接收者发现它丢失了一条消息时,它就向组中的其他成员多播它的反馈。

多播反馈可以使组中的其他成员抑制自己的反馈。假定多个接收者都丢失了消息 m,它们中的每一个都需要向发送者 S 返回一个否定确认以使 m 被重发。但是,如果我们假定重发总是向整个组进行多播,那么只要有一个重发请求到达 S 就足够了。

由于这个原因,一个没有接收到消息 m 的接收者 R 延迟一个随机的时间然后发送反馈消息。也就是说,在过了一段随机时间之后再请求重发。如果同时有其他对 m 的重发请求到达 R,那么 R 就抑制自己的反馈,因为它知道 m 会短期内进行重发。在理想的情况下,这种方法中只有一个反馈消息到达 S,然后 S 重发 m。这种方法如图 7.9 所示。

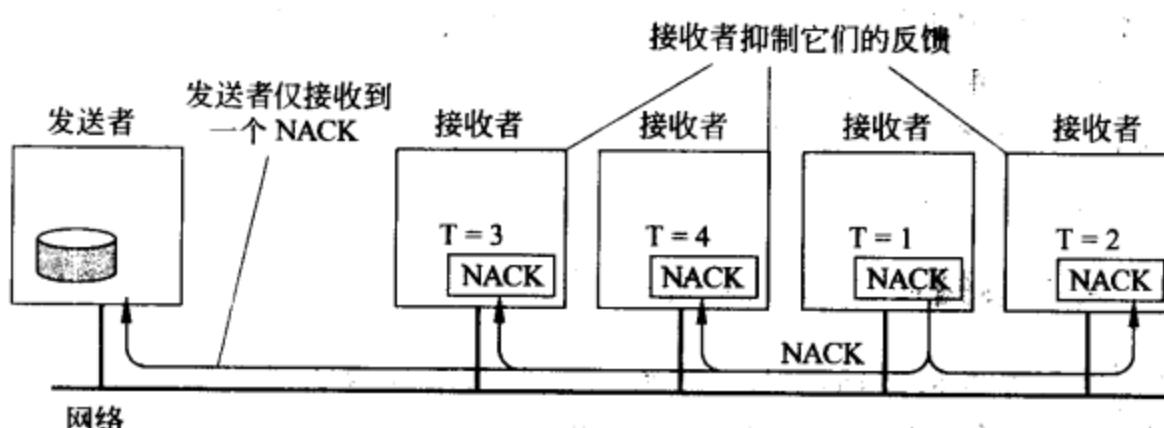


图 7.9 几个接收者要发送重发请求,但是第一个重发请求抑制了其他的请求

反馈抑制被证明具有相当好的可扩展性,它被大量的 Internet 应用程序,例如共享白板用作底层机制。但是这种方法也有一些严重的问题。首先,要确保只有一个重发请求返回给发送者需要每个接收者都对反馈消息进行相当准确的调度。否则还是会有很多接收者同时返回它们的反馈。在散布于广域网上的多个进程组中设置定时器不是容易的事。

另外一个问题是由多播反馈也会中断其他成功接收消息的进程。换句话说,其他接收者不得不接收并处理对它们无用的消息。要解决这个问题,惟一的方法就是使没有接收到消息 m 的进程加入到一个独立的多播组来接收 m,在(Kasera 等 1997)中对此进行了说明。不幸的是,这样的方法要求以一种效率很高的方式来管理组,这在广域系统中很难做到。一种更好的方法是把丢失相同消息的接收者编组,共享同一个反馈消息和重发的

通道。可以在(Liu 等 1998)中找到这种方法的细节。

使接收者进行本地恢复有助于提高 SRM 的可扩展性。特别是,如果一个成功地接收到消息 m 的接收者收到一个重发请求,它就可以决定甚至在重发请求到达原始的发送者之前就多播 m 。可以在(Floyd 等 1997,Liu 等 1998)中找到更多的有关细节。

2. 分等级的反馈控制

刚才介绍的反馈抑制基本上是一种无等级的方法。但是,要在非常大的接收者组中获得可扩展性,就需要采用分等级的方法。从根本上说,可靠多播的分等级方法以图 7.10 所示的方式进行工作。

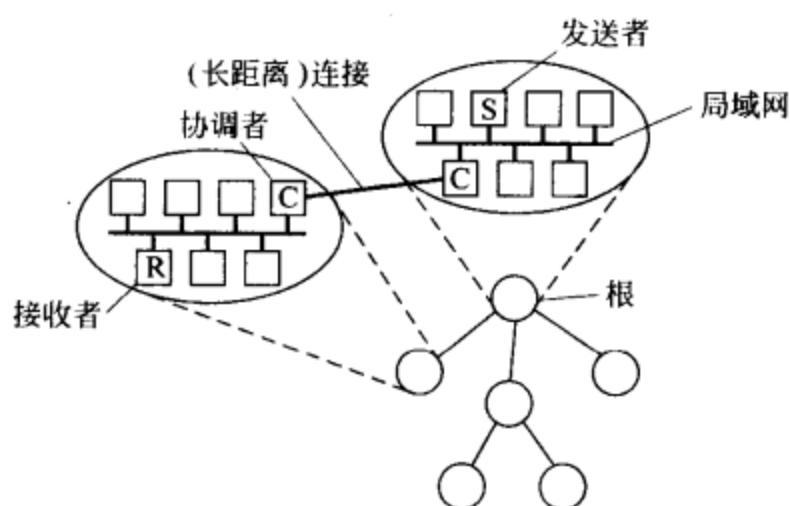


图 7.10 分等级的可靠多播。每个本地协调者都把消息转发给它的孩子然后再处理重发请求

为了简化,我们假定只有一个发送者需要向一个非常大的接收组进行多播。接收组分为很多子组,组织成树的形式。包含发送者的子组构成了树的根。在每个子组中,任何一种可以为小的组工作的可靠多播方案都可以使用。

每个子组都指定一个本地协调者,它负责处理子组中包含的接收者的重发请求(请参阅 Hofmann 1996)。本地协调者具有自己的历史缓存器。如果协调者本身丢失了消息 m ,它就请求父子组的协调者重发 m 。在基于确认的方法中,如果接收到消息,本地协调者就向它的父亲发送一个确认。如果协调者从它的子组的所有成员以及它的孩子那里接收到对消息 m 的确认,那么就可以从历史缓冲中删除 m 。

分等级的方法中的主要问题在于树的建立。在很多情况下需要动态建立树。一种方法是使用底层网络的多播树(如果存在的话)。在原理上,这种方法强化网络层中的每个多播路由器的作用,可以按照刚才所讲的方式把它们作为本地协调者来使用。不幸的是,对现存的计算机网络进行改选并不容易。

总之,建立可以扩展到跨越广域网络的大量接收者的可靠多播是一个困难的问题。不存在一个最好的方法,每种方法都会导致新的问题出现,所以在我们被这些问题难住之前还需要在这个领域中进行大量的研究。

7.4.3 原子多播

现在我们返回到需要在存在进程失败的情况下获得可靠多播的情况。特别是,在分

布式系统中经常需要保证消息要么就被发送给所有的进程,要么就不向任何进程一个也不发送。另外,如果传送的话,通常还需要所有的消息都按相同的顺序发送给所有的进程。这种方式称为原子多播(atomic multicast)。

为了理解为什么原子性这么重要,我们来考虑一个作为在分布式系统上建立的应用程序基础的复制数据库。分布式系统提供了可靠的多播功能。特别是,它允许建立可以可靠接收消息的进程组。因此,复制的数据库作为一组进程被建立,每个进程用于一个副本。更新操作总是向所有副本进行多播,然后在本地执行。换句话说,我们假定使用的是活动的复制协议。

假定现在要执行一系列的更新,但是在一个更新执行期间,一个副本崩溃了,于是该副本上的更新就丢失了。但是另一方面,在其他的副本上的更新被正确执行了。

当崩溃的副本恢复时,它可以恢复到它崩溃之前的状态,但是它可能会错过一些更新。在这一点上,让它与其他副本保持状态一致是基本的要求。使一个副本与其他副本保持相同的状态需要我们确切地知道错过了哪些操作以及它们是按什么次序执行的。

现在假定底层的分布式系统支持原子多播。在这种情况下,在一个副本崩溃之前发送给所有副本的更新操作或者是在所有的正常副本上都执行,或者是一个都不执行。特别是,在使用原子多播的情况下,只有在所有正确操作的副本就组成员关系达成一致时,操作才能执行。换句话说,如果其余的副本达成一致,认为崩溃的副本不再属于该组,那么更新就可以执行。

当崩溃的副本恢复时,它就被强迫再次加入该组。在它重新注册为组成员之前没有更新操作转发给它。加入一个组需要使它的状态与其他的组成员保持一致。因此,原子多播可确保没有故障的进程对数据库保持一致的视图。当一个副本从故障中恢复并重新加入组时,原子多播强制它与组的其他成员一致。

1. 虚拟同步

存在进程失败时的可靠多播可以根据进程组和组成员关系的变化进行准确的定义。下面我们来区分一下接收消息和发送消息。特别指出的是,我们采用如图 7.11 中的模式,其中的分布式系统由通信层组成。消息在这个通信层中进行发送和接收。接收到的消息在发送到高层的应用程序之前放在通信层的本地缓存器中。

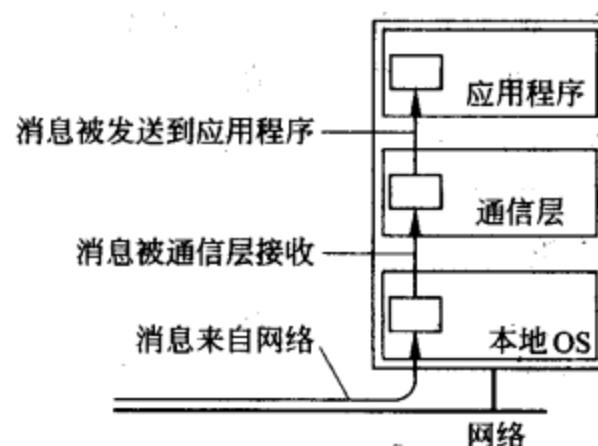


图 7.11 分布式系统中的消息接收和消息发送的逻辑组织

原子多播的总体思想是多播消息 m 应该惟一地跟它应该传送到的一个进程表相关联。这个表对应于一个组视图，也就是关于包含在组中的进程集的视图。很重要的一点是列表中的每个进程都具有相同的视图。换句话说，它们都同意 m 应该被传送给它们中的每一个而且不被传送到别的进程。

现在假定 m 在发送者具有组视图 G 时进行多播。此外，假定在发生多播时另一个进程加入了或离开了组。组成员关系的这个改变被宣告给 G 中的所有进程。换句话说，通过多播一个 vc 消息宣布一个进程加入或离开，视图的改变就发生了。现在有两个多播消息在同时进行传输： m 和 vc 。我们需要保证要么在 vc 传送到 G 中的每个进程之前把 m 传送到每个进程，或者根本不传送 m 。要注意到这个要求有些类似于第 5 章中讨论过的全序多播。

这里出现了一个问题，如果 m 没有被传送到任何一个进程，怎么能说这种协议是可靠的多播协议呢？原则上，只有一种情况下 m 的传送允许失败：当组成员关系的改变是由于 m 的发送者崩溃造成的结果时。在这种情况下，或者是 G 里的所有成员都知道这个新成员的异常结束，或者没有一个成员知道。作为选择， m 可以被每个成员忽略，这对应于在发送 m 之前发送者崩溃的情况。

这种更强的可靠多播保证多播到组视图 G 的消息被传送给 G 中的每个正常进程。如果消息的发送者在多播期间崩溃，那么消息或者被投递给所有剩余的进程，或者被每个进程忽略。具有这种属性的可靠多播被称为虚拟同步(Birman, Joseph 1987a, Birman, Joseph 1987b)。

考虑一下图 7.12 所示的 4 个进程。在某个时刻，进程 P_1 加入了一个进程组，然后这个进程组就由 P_1, P_2, P_3 和 P_4 组成。在多播了一些消息之后，进程 P_3 崩溃了。但是在崩溃之前它成功地将消息多播到了进程 P_2 和 P_4 ，但是没有多播到 P_1 。但是，虚拟同步保证这个消息根本不会被传送，这就有效地建立起一种情况，使得在 P_3 崩溃之前消息从来没有被发送。

在从组中删除 P_3 之后，通信在剩余的组成员之间进行。后来 P_3 恢复了，它可以在它的状态进行更新之后再次加入该组。

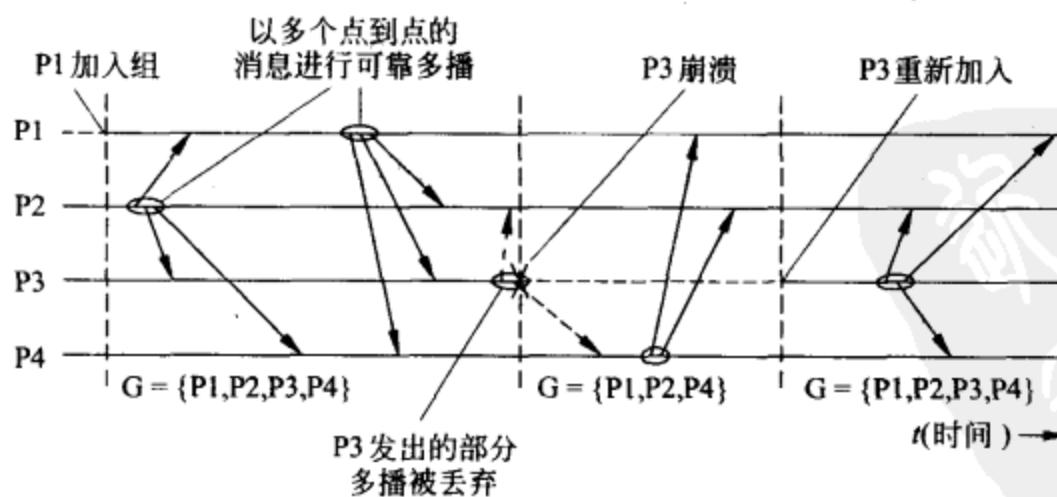


图 7.12 虚拟同步多播的原理

虚拟同步的原理是所有的多播都在视图改变之间进行。换句话说，视图改变作为一个屏障，不能跨越它进行多播。它类似于在上一章中讨论过的分布式数据存储中的同步变量。在视图改变时进行的所有多播都在视图改变生效之前完成。虚拟同步的实现并不烦琐，下面我们进行详细的讨论。

2. 消息排序

虚拟同步允许应用程序开发者认为多播是在不同的时期中发生的，不同的时期可以按照组成员关系的改变来划分。然而还没有谁发表关于多播的排序的论文。通常有 4 种不同的排序方法：

- (1) 不排序的多播；
- (2) FIFO 顺序的多播；
- (3) 按因果关系排序多播；
- (4) 全序多播。

可靠的、不排序多播是一种虚拟同步多播，它对接收不同进程发送的消息的次序不做任何保证。要解释这一点，我们假定可靠多播由一个提供发送和接收原语的库支持。接收操作在消息发送到它之前阻塞调用进程。

现在假定发送者 P1 把两条消息多播给一个组，同时该组中的两个其他进程正在等待消息到达，如图 7.13 所示。假定在多播期间进程不会崩溃也不会离开该组，有可能 P2 的通信层先接收到消息 m1，然后是 m2。因为没有消息顺序约束，所以消息可能是按它们被接收的顺序被传送给 P2。相反，P3 的通信层有可能首先接收到 m2，然后才是 m1，然后把它们按相同的顺序传送给 P3。

进程 P1	进程 P2	进程 P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

图 7.13 同一个组中的通信进程。每个进程中时间的顺序按垂直顺序排列

在可靠的 FIFO 多播的情况下，通信层被强制按照消息发送的顺序传送来自同一进程的消息。考虑如图 7.14 所示的在一个四个进程的组中的通信。使用 FIFO 的顺序，只会介意一件事情，消息 m1 总是在 m2 之前传送，同样，消息 m3 总是在 m4 之前传送。组中的所有进程都必须遵守该规则。换句话说，当 P3 的通信层先接收到 m_out2 时，在把它传送给 P3，之前必须等待 m_out1 的接收与传送。

进程 P1	进程 P2	进程 P3	进程 P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

图 7.14 同一组中的 4 个进程，具有两个不同的发送者，在 FIFO 多播下可能的消息传送顺序

但是,关于由不同进程发送的消息的传送顺序没有约束。换句话说,如果进程 P2 在 m3 之前接收到了 m1,它可能以那种顺序传送这两个消息。同时,进程 P3 可能在接收到 m1 之前接收到了 m3。FIFO 顺序说明 P3 可能在传送 m1 之前传送 m3,尽管这种传送顺序与 P2 不同。

最后,采用可靠的按因果关系排序多播来传送消息可以保留不同的消息之间的潜在因果关系。换句话说,如果在因果关系的消息 m1 在另一个消息 m2 之前,那么无论它们是否由同一个发送者多播的,每个接收者的通信层都应该在接收并传送 m1 之后传送 m2。注意按因果关系排序的多播可以使用第 5 章中讨论的向量时间戳来实现。

除了这三种排序,还有一种对消息传送的附加约束的排序被称为完全排序。完全排序的传送意味着不论消息传送是无序、FIFO 顺序还是按因果关系排序,都需要在传送消息时,对所有的组成员按照相同的次序来传送。

例如,使用 FIFO 与完全排序多播的结合,图 7.14 中的进程 P2 和 P3 都可以首先传送消息 m3,然后传送消息 m1。但是,如果 P2 在 m3 之前传送了 m1,而 P3 在 m1 之前传送了 m3,那么就可能违背全序的约束。注意也应该考虑采用 FIFO 排序。换句话说,m2 应该在 m1 之后传送,而 m4 应该在 m3 之后传送。

提供了全序的消息传送的虚拟同步可靠多播称为原子多播。使用上面讨论的三种不同的消息排序约束,可以有 6 种形式的可靠多播,如图 7.15 所示(Hadzilacos, Toueg 1993)。

多播	基本的消息排序	完全排序传送?
可靠多播	无	不
FIFO 多播	FIFO 排序传送	不
按因果关系多播	按因果关系传送	不
原子多播	无	是
FIFO 原子多播	FIFO 排序传送	是
按因果关系的原子多播	按因果关系传送	是

图 7.15 6 种不同的虚拟同步可靠多播

3. 实现虚拟同步

现在我们来考虑虚拟同步可靠多播的实现。实现这种多播的一个例子是 Isis,一个已经实际使用了几年的容错分布式系统。我们首先来讨论一下在(Birman 等 1991)中说明的有关它的实现的一些问题。

Isis 中的可靠多播使用了底层网络中可用的可靠的点到点通信工具,特别是 TCP。把消息 m 发送给一个进程组是通过把 m 可靠地发送给每个组成员来实现的。作为结果,尽管可以保证每个传输都成功,但还是不能保证所有的组成员都接收到 m。特别是,发送者可能在把 m 传输给每个成员之前崩溃了。

除了使用可靠的点到点通信以外, Isis 还假定来自同一个源的消息在通信层按照发送它们的顺序接收。在实践中, 可以使用点到点通信的 TCP 连接来解决这个问题。

需要解决的主要问题是保证发送到视图 G 的所有消息在组成员关系发生改变之前被传送到 G 中的所有正常进程。需要注意的第一个问题是确保 G 中的每个进程都接收到发送给 G 的所有消息。注意, 因为将消息 m 发送到 G 的发送者可能在完成多播之前崩溃, 有可能 G 中的进程永远不会接收到 m 。因为发送者已经崩溃, 这些进程应该从其他的地方得到 m 。下面解释进程如何探测到它丢失了消息。

要解决这个问题, 可以让 G 中的每个进程在确认 G 中的所有进程接收到 m 之前保留 m 。如果 G 中的所有进程都接收到 m , 那么 m 就被称为是稳定的。稳定的消息只允许传送一次。要确保稳定性, 在 G 中选择一个任意的(可操作的)进程并要求它向其他所有进程发送 m 就足够了。

更明确地说, 假定当前视图为 G_i , 但是有必要建立下一个视图 G_{i+1} 。为了不失一般性, 我们可以假定 G_i 和 G_{i+1} 最多相差一个进程。进程 P 在接收到视图改变的消息时意识到视图发生改变。这样的消息可能来自想要加入或离开组的进程, 或者来自 G_i 中被探测到失败并要被删除的进程, 如图 7.16(a) 所示。

当进程 P 接收到 G_{i+1} 的视图改变消息时, 它首先把它拥有的 G_i 中任何不稳定消息的拷贝转发给 G_{i+1} 中的每个进程, 然后把它标记为稳定的。回忆一下, Isis 假定点到点的通信是可靠的, 这样转发的消息就不会丢失。这样的转发保证, G_i 中已经至少被一个进程接收到的所有消息可以被 G_i 中的所有正常进程接收。注意, 选择单一的协调者转发不稳定消息就足够了。

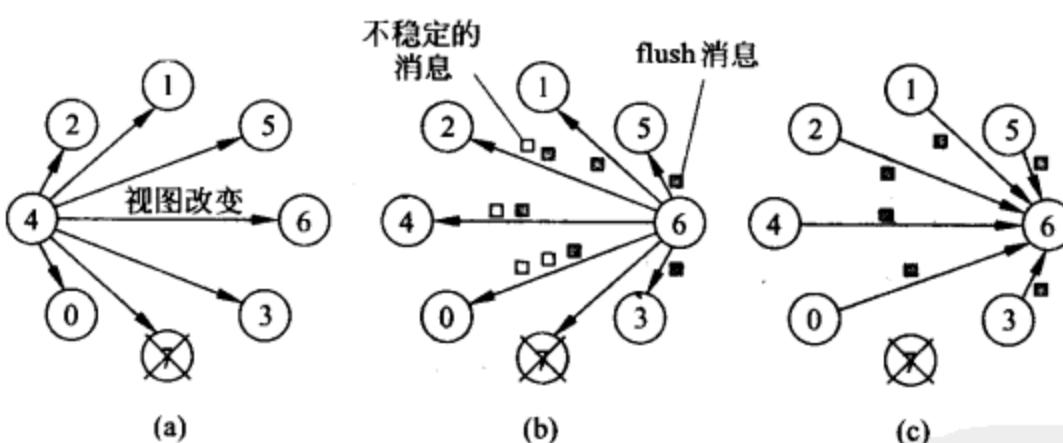


图 7.16 实现虚拟同步的过程

(a) 进程 4 注意到进程 7 已经崩溃并发送一个视图改变; (b) 进程 6 发送它的所有不稳定消息, 然后发送一个 flush 消息; (c) 当进程 6 从其他每个进程那里都接收到一个 flush 消息时就建立了一个新的视图

为了表示进程 P 不再具有任何不稳定的消息, 以及当其他进程可以建立 G_{i+1} 时它也可以建立 G_{i+1} , 它就多播一个 G_{i+1} 的 flush 消息, 如图 7.16(b) 所示。在进程 P 从其他每个进程那里都接收到 G_{i+1} 的 flush 消息后, 就可以安全地建立新的视图, 如图 7.16(c) 所示。

当进程 Q 接收到在 G_i 中发送的消息 m 而且 Q 还认为当前视图为 G_i 时, 它就传送 m 而不考虑任何附加的消息排序约束。如果它已经接收到 m , 它就认为消息是复制的, 然后丢弃它。

因为进程 Q 最后会接收到 G_{i+1} 的视图改变消息, 它首先转发它的不稳定消息, 然后通过发送一个 G_{i+1} 的 flush 消息来屏蔽别的消息。注意, 由于底层通信层进行消息排序, 一个进程的 flush 消息总是在接收到同一进程的不稳定消息之后进行接收。

迄今为止介绍过的协议的一个主要缺点在于它不能处理当宣布新的视图改变时的进程失败。特别是, 它假定在 G_{i+1} 中的每个成员都建立起新的视图 G_{i+1} 之前, G_{i+1} 中的进程不会失败(这将导致下一个视图 G_{i+2})。这个问题可以通过在所有的进程还没有建立上一个改变时就宣布任意视图 G_{i+k} 的视图改变来解决。其中的细节留作练习。

7.5 分布式提交

前面讨论的原子多播问题是一个称为分布式提交的更一般化问题中的一个例子。分布式提交的问题涉及到要使一个操作被进程组中的每个成员都执行或一个成员都不执行。在可靠多播的情况下, 操作就是消息的传送。在分布式事务中, 操作可能是单个站点上事务的提交, 它是整个事务的一部分。分布式提交中的其他例子及其解决办法在 (Tanisch 2000) 中进行了讨论。

分布式提交通常以设立协调者的方式建立。在简单的方案中, 协调者通知所有涉及到的称为参与者的其他进程是否(本地)按要求执行操作。这种方法称为单阶段提交协议。如果有一个参与者不能真正执行该操作, 那么就说明它存在着明显的缺陷, 因为它没有办法来通知协调者。例如, 在分布式事务的情况下, 不可能进行本地提交, 因为这会破坏并发控制约束。

在实践中需要更复杂的方法, 最常用的是两阶段提交协议, 下面将对此进行详细讨论。这种协议的主要缺点是不能有效地处理协调者失败的情况。后来又研究出了三阶段提交, 我们也要讨论它。

7.5.1 两阶段提交

两阶段提交协议(2PC)是由 Gray (1978) 提出的。为了不失一般性, 我们考虑一个分布式事务, 其中有很多进程作为参与者, 每个进程都运行在不同的机器上。假定没有故障发生, 协议就由以下两个阶段组成, 每个阶段又由两步组成(请参阅 Bernstein 等 1987):

(1) 协调者向所有的参与者发送一个 VOTE_REQUEST 消息。

(2) 当参与者接收到 VOTE_REQUEST 消息时, 就向协调者返回一个 VOTE_COMMIT 消息通知协调者它已经准备好本地提交事务中属于它的部分, 否则就返回一个 VOTE_ABORT 消息。

(3) 协调者收集来自参与者的所有选票。如果所有的参与者都表决要提交事务，那么协调者就进行提交。在这种情况下它向所有的参与者发送一个 GLOBAL_COMMIT 消息。但是，如果有一个参与者表决要取消事务，那么协调者就决定取消事务并多播一个 GLOBAL_ABORT 消息。

(4) 每个提交表决的参与者都等待协调者的最后反应。如果参与者接收到一个 GLOBAL_COMMIT 消息，那么它就在本地提交事务，否则当接收到一个 GLOBAL_ABORT 消息时，就在本地取消事务。

表决阶段的第一阶段由第(1)、(2)步组成。第二阶段是决定阶段，由第(3)、(4)步组成。这 4 个步骤由图 7.17 中有限状态图来说明。

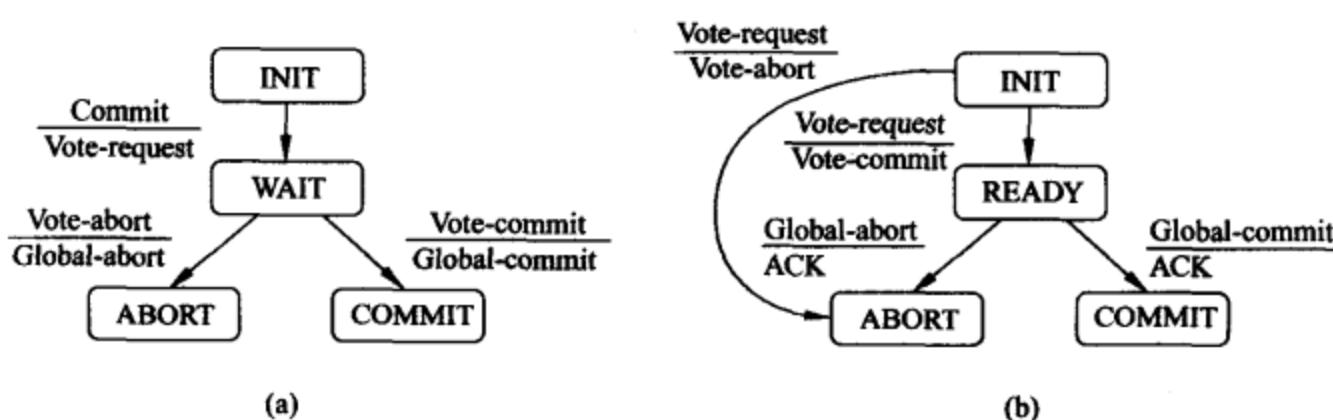


图 7.17 2PC 中协调者参与者的有限状态机

(a) 2PC 中协调者的有限状态机；(b) 参与者的有限状态机

当在发生故障的系统中使用这个基本的 2PC 协议时就会出现一些问题。首先，注意到协调者和参与者都具有阻塞等待消息的状态。因为当一个进程崩溃，而其他进程又正在无限等待来自该进程的消息时，这个协议很容易崩溃。因此使用了定时机制。在以下内容中将对这个机制进行解释。

看一下图 7.17 中的有限状态机，可以看出在其中三个状态中，协调者和参与者都被阻塞，等待某个消息到来。首先，参与者可能在 INIT 状态等待来自协调者的 VOTE_REQUEST 消息，如果在一段时间之后还没有接收到这个消息，那么参与者就简单地在本地中止事务，然后向协调者发送一个 VOTE_ABORT 消息。

同样，协调者可能在 WAIT 状态阻塞，等待来自每个参与者的表决。如果在某段时间之后没有接收到所有的表决，那么协调者就决定中止表决，然后向所有的参与者发送 GLOBAL_ABORT 消息。

最后，参与者可能在 READY 状态阻塞，等待协调者发送的全局表决消息。如果在给定的时间内没有接收到消息，参与者不能简单地决定中止事务。它必须查明协调者发送的是什么消息，要解决这个问题，最简单的方法就是让每个参与者在协调者再次恢复之前阻塞。

一种更好的方法是让一个参与者 P 与另一个参与者 Q 联系，以便根据 Q 的当前状态来决定做什么。例如，假定 Q 到达了状态 COMMIT，那么就有可能协调者在崩溃之前只

向 Q 发送了 GLOBAL_COMMIT 消息。显然这个消息没有发送给 P。因此, P 应该决定进行提交。同样,如果 Q 的状态为 ABORT,那么 P 也可以安全地进行中止。

现在假定 Q 还在状态 INIT,当协调者已经向所有的参与者发送了 VOTE_REQUEST 消息,但是这个消息只到达 P(然后它用 VOTE_COMMIT 消息作为应答),而没有到达 Q 时就是这种情况。换句话说,协调者在多播 VOTE_REQUEST 时崩溃了。在这种情况下,中止事务是安全的,P 和 Q 都可以把状态转换为 ABORT。

最困难的情况发生在当 Q 也在 READY 状态等待协调者的响应的时候。特别是,如果所有的参与者都处在 READY 状态,那么它们就无法做出决定。问题在于尽管所有的参与者都想要提交,但是还需要协调者的表决才能做出最后的决定。因此,协议在协调者恢复之前阻塞。

在图 7.18 中对不同的操作进行了总结。

要确保进程可以真正恢复,必须在持久性存储器中保存它的状态。(如何以容错的方式保存数据在本章后面进行讨论)。例如,如果一个参与者处于 INIT 状态,它就可以在恢复时安全地决定在本地中止事务并通知协调者。同样,如果它已经做出了诸如处于 COMMIT 或 ABORT 状态时何时崩溃的决定时,它将按顺序恢复到该状态,并向协调者重发它的决定。

当参与者在 READY 状态下崩溃时就会出现问题。在这种情况下,当它恢复时不能确定自己下一步做什么,即不能确定是提交还是中止事务。因此,就要强制它与其他参与者进行联系来看下一步要做什么,这与上面介绍的在 READY 状态下超时的情况类似。

Q 的状态	P 采取的行动
COMMIT	转换到 COMMIT
ABORT	转换到 ABORT
INIT	转换到 ABORT
READY	与其他参与者联系

图 7.18 参与者 P 在 READY 状态下与另一个参与者 Q 联系时采取的行动

协调者只有两个需要跟踪的关键状态。当开始 2PC 协议时,应该记录它进入了 WAIT 状态,这样就可以在恢复之后向所有的参与者重发 VOTE_REQUEST 消息。同样,如果它在第二阶段中做出决定,那么只要记录该决定就足够了,这样就可以在恢复时进行重发这个决定。

在图 7.19 中给出了对协调者执行的操作的总结。协调者首先向所有的参与者多播一个消息以收集它们的表决。随后记录它进入了 WAIT 状态,然后等待来自参与者的表决。

协调者的操作:

```

        write START_2PC to local log;
        multicast VOTE_REQUEST to all participants;
        while not all votes have been collected {
            wait for any incoming vote;
            if timeout {
                write GLOBAL_ABORT to local log;
                multicast GLOBAL_ABORT to all participants;
                exit;
            }
            record vote;
        }
        if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
            write GLOBAL_COMMIT to local log;
            multicast GLOBAL_COMMIT to all participants;
        } else {
            write GLOBAL_ABORT to local log;
            multicast GLOBAL_ABORT to all participants;
        }
    }

```

图 7.19 在两阶段提交协议中协调者采取的操作

如果不仅没有收集到所有的表决,而且在给定的时间内没有再接收到更多的表决,那么协调者就假定有一个或多个参与者崩溃了。因此,它就中止事务,然后向(剩余的)参与者多播一个 GLOBAL_ABORT 消息。

如果没有崩溃发生,那么协调者就会收集到所有的表决。如果所有的参与者和协调者都表决要提交,那么协调者就首先把一个 GLOBAL_COMMIT 记入日志,然后发送给所有的进程。否则协调者就多播一个 GLOBAL_ABORT(在把它记入本地日志之后)。

图 7.20(a)说明了参与者执行的步骤。首先,进程等待来自协调者的表决请求。注意这个等待可以由在该进程的地址空间中运行的一个独立线程来进行。如果没有消息到来,那么事务就简单地被中止。显然协调者已经崩溃了。

在接收到表决请求之后,参与者决定为提交一个事务进行表决,首先在本地日志中记录它的决定,然后通过发送一个 VOTE_COMMIT 消息来通知协调者。然后参与者必须等待全局决定,假定这个决定(来自协调者)按时到达,那么就简单地把它记录到本地日志中,然后执行。

但是,如果参与者在等待协调者的决定到来时超时,它就执行一个终止协议,首先向其他进程多播一个 DECISION_REQUEST 消息,然后阻塞,等待响应。当响应到达时(可能来自协调者,假定它最后恢复了),这个参与者就把决定写入它的本地日志,然后处理它。

每个参与者都应该准备好接收来自其他参与者为做全局决定而发出的请求。假定每个参与者都开启了一个线程与参与者的主线程并发执行,如图 7.20(b)所示。这个线程在接收到决定请求之前处于阻塞状态。只有在与之相关的参与者已经做出最后决定时该线程才对其他进程有帮助。换句话说,如果协调者已经把 GLOBAL_COMMIT 或 GLOBAL_ABORT 写到了本地日志中,那么它就至少向这个进程发送了它的决定。另外,线程也要在

与之相关的参与者还处在 INIT 状态时发送 GLOBAL_ABORT, 就像前面讨论的那样。在所有其他的情况下, 接收线程都没有帮助作用, 请求的参与者也不能得到响应。

我们已经看到参与者有可能需要在协调者恢复之前阻塞。当所有的参与者都从协调者那里接收到 VOTE_REQUEST 消息并进行处理, 而在同时协调者崩溃时, 就会发生这样的情况。在这种情况下, 参与者在要采取的最后行动的问题上不能协调一致地做出决定。因此, 2PC 被称为阻塞提交协议。

参与者的操作:

```
write INIT to local log;
wait for VOTE-REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send VOTE_ABORT to coordinator;
}
```

(a)

actions for handling decision requests: /* executed by separate thread */

```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
}
```

(b)

图 7.20 2PC 中参与者采取的步骤和处理输入的决定请求的步骤

(a) 2PC 中参与者争取的步骤; (b) 处理输入的决定请求的步骤

有一些方法可以避免阻塞。Babaoglu 和 Toueg (1993) 描述了一种方法, 它使用一个广播原语, 该原语中接收者立刻把它接收到的消息向其他所有进程进行广播。这种方法允许参与者即使在协调者没有恢复时也能到达最后决定。另一种方法是三阶段提交协议, 它是本章的最后一个主题, 我们下面就要讨论它。

7.5.2 三阶段提交

两阶段提交的一个问题在于当协调者崩溃时, 参与者不能做出最后的决定。因此参与者可能在协调者恢复之前保持阻塞。Skeen (1981) 开发了一种 2PC 的变种, 称为三阶段提交协议(3PC), 它避免了在出现故障停机时的阻塞过程。尽管 3PC 在文章中广泛出现, 但它在实际中用得并不多, 因为在 2PC 中阻塞的情况很少发生。我们讨论这个协议, 是因为它提供了解决分布式系统中的容错问题的更多看法。

像 2PC 一样, 3PC 也可以用一个协调者和一些参与者来进行说明。它们各自的有限状态机如图 7.21 所示。该协议的本质在于协调者和每个参与者都满足以下两个条件:

- (1) 没有一个可以直接转换到 COMMIT 或 ABORT 状态的单独状态;
- (2) 没有一个这样的状态: 它不能做出最后决定, 而且可以从它直接转换到 COMMIT 状态。

(Skeen, Stonebraker 1983) 中说明了这两个条件是使提交协议不阻塞的充分必要条件。

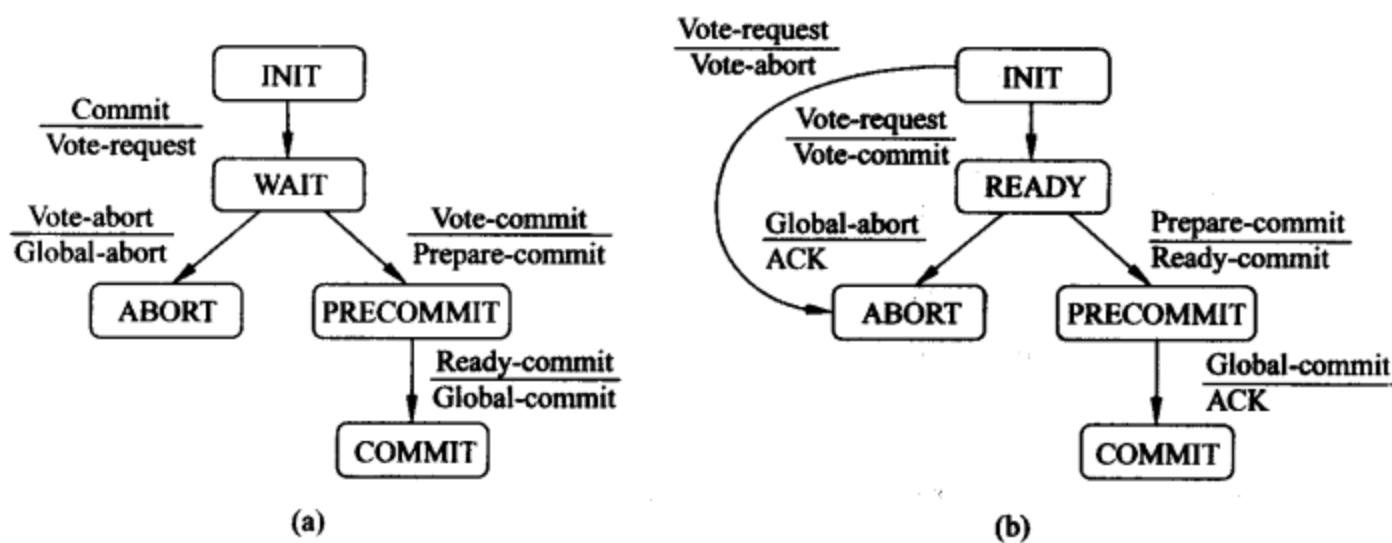


图 7.21 3PC 中协调者和参与者的有限状态机
(a) 3PC 中协调者的有限状态机; (b) 参与者的有限状态机

3PC 中的协调者首先向所有的参与者发送一个 VOTE_REQUEST 消息, 然后等待到来的应答。如果任意一个参与者投票中止事务, 那么最后决定就是中止该事务, 协调者发送一个 GLOBAL_ABORT 消息。但是如果事务可以提交, 就发送一个 PREPARE_COMMIT 消息。只有在每个参与者都确认它已经准备提交时, 协调者才发送最后的 GLOBAL_COMMIT 消息, 这时事务才真正提交。

同样会有一些进程在等待消息到来时阻塞的情况。首先, 如果参与者在 INIT 状态时等待来自协调者的投票要求, 假定协调者已经崩溃, 那么它最终会转换到 ABORT 状态。

态。这种情况下与 2PC 中相同。与此类似,协调者可能在 WAIT 状态等待来自参与者的投票。当超时发生时,协调者就认为参与者已经崩溃,通过多播一个 GLOBAL_ABORT 消息来中止事务。

现在假定协调者在 PRECOMMIT 状态阻塞,当超时时它就认为一个参与者已经崩溃,但是可以知道那个参与者已经投票要提交事务。因此,协调者可以通过多播一个 GLOBAL_COMMIT 消息来指示可操作的参与者提交。另外,当崩溃的参与者再次启动时,它要依赖恢复协议来最终提交事务中属于它的部分。

参与者 P 可能在 READY 状态或 PRECOMMIT 状态阻塞。当超时时,P 只能认为是协调者崩溃,所以它需要决定下一步做什么。与 2PC 中相同,如果 P 与其他处于 COMMIT(或 ABORT)状态的参与者联系,那么 P 也应该转换到该状态。另外,如果所有的参与者都处于 PRECOMMIT 状态,那么事务就可以安全提交。

与 2PC 相同,如果另一个参与者 Q 还处于 INIT 状态,那么就可以安全地中止事务。只有别的参与者都不处于 PRECOMMIT 时 Q 才能处于 INIT 状态,记住这一点很重要。只有协调者在崩溃前已经到达 PRECOMMIT 状态并接收到每个参与者的提交表决时,参与者才可能到达 PRECOMMIT 状态。换句话说,没有参与者处于 INIT 状态而其他的参与者处于 PRECOMMIT 状态。

如果 P 可以联系的每个参与者都处于 READY 状态,那么事务应该被中止。要记住的一点是另外的参与者可能崩溃然后又恢复。但是,P(而不是任何其他可操作的参与者)知道崩溃的参与者在恢复之后的状态。如果进程恢复到 INIT 状态,那么决定中止事务是惟一正确的决定。在最坏的情况下,进程可以恢复到 PRECOMMIT 状态,但是在这种情况下,中止事务不会造成任何损害。

与 2PC 相比,3PC 的主要不同点在于以下的这种情况:崩溃的参与者可能恢复到了 COMMIT 状态而所有其他参与者还处于 READY 状态。在这种情况下,其余的可操作进程不能做出最后的决定,不得不在崩溃的进程恢复之前阻塞。在 3PC 中,只要有可操作的进程处于 READY 状态,就没有崩溃的进程可以恢复到 INIT、ABORT 或 PRECOMMIT 之外的状态。因此存活进程总是可以做出最后决定。

最后,如果进程 P 可以到达状态 PRECOMMIT(并形成多数),那么提交事务就是安全的。这里说明在所有其他的进程至少处于 READY 状态的情况下,崩溃的进程可以恢复到 READY、PRECOMMIT 或 COMMIT 状态。

有关 3PC 的更多细节可以在文献(Bernstein 等 1987)和(Chow, Johnson 1997)中找到。

7.6 恢复

迄今为止我们主要集中于讨论允许容错的算法。但是,一旦发生了故障,使发生故障的进程恢复到正确的状态就是基本的问题了。下面我们首先说明恢复到正确状态到底意味着什么,然后说明如何通过检查点(check point)和消息日志的方式记录并恢复分布式系统的状态。

7.6.1 简介

容错的基本要求是从错误中恢复。回忆一下，错误是系统中可能导致失败的部分。错误恢复的总体想法是用正确的状态取代错误的状态。从本质上说，有两种形式的错误恢复。

在回退恢复(backward recovery)中，主要问题是要将系统从当前的错误状态回到先前的正确状态。要做到这一点，必须定时记录系统的状态，以便当发生错误时恢复到记录的状态。每次记录系统的当前状态时，就称为设置一个检查点。

错误恢复的另一种形式是前向恢复(forward recovery)。在这种情况下，当系统进入错误状态时，不是回退到以前的检查点处的状态，而是尝试从可以继续执行的某点开始把系统带入一个正确的新状态。前向错误恢复机制的关键在于它必须预先知道会发生什么错误。只有在这样的情况下才有可能纠正错误并转到新的状态。

考虑一下可靠通信的实现，就很容易解释前向错误恢复与回退错误恢复之间的区别。要从分组丢失中恢复，通常的方法是让发送者重发该分组。实际上，分组重发就是尝试着回退到以前的正确状态，也就是发送丢失的分组时的状态。因此，通过分组重发获得的可靠通信就是应用回退错误恢复技术的一个例子。

还有一种可供选择的方式是使用一种称为擦除修正(erasure correction)的方法。这种方法是从其他成功传送的分组中建立丢失的分组。例如，在一个 (n, k) 块的擦除码中，一组 k 个源分组被编码为一组 n 个已编码的分组，这样任意 k 个已编码的分组都可以重建出原来的 k 个源分组。典型的值为 $k=16$ 或 $k=32$ ，而且 $k < n \leq 2k$ ，请参阅(Rizzo 1997)。如果没有足够的分组被传送，发送者就不得不继续发送分组直到先前丢失的分组可以被重建为止。擦除修正是一种前向错误恢复方法的典型例子。

一般说来，回退错误恢复技术作为一种从分布式系统故障中恢复的机制被广泛应用。回退错误恢复的一个重要优点在于它是一种通用的方法，不依赖于任何特定的系统或进程。换句话说，它可以作为一项通用的服务集成到分布式系统的中间层中。

但是，回退错误恢复也有一些问题(Singhal, Shivaratri 1994)。首先，把一个系统或进程恢复到原先的状态通常是一个相对开销较大的操作。例如，如接下来的部分讲述的那样，要从进程崩溃或站点失败中恢复通常需要做很多工作。

第二，因为回退错误恢复机制不依赖于实际使用的分布式应用程序，不能保证一旦进行了恢复就不再发生相同或类似的失败。如果需要这样的保证，那么通常需要应用程序在恢复中参与错误处理。换句话说，回退错误恢复机制通常不能提供完全的故障透明性。

最后，尽管回退错误恢复需要设置检查点，但也决不会回退到某些状态。例如，如果一个(可能是恶意的)人从自动柜员机上取走了1000美元时，那么即使自动柜员机立即回退这些钱也很难追回。与此类似，在大多数UNIX系统中输入/bin/rm -fr *之后，要从错误的工作目录中恢复到先前的状态绝大多数人可能是无能为力的。有些事情是不可逆的。

通过检查点可使系统恢复到先前的正确状态。但是，设置检查点通常是一个开销很大的操作，可能引起严重的性能问题。因此，很多容错的分布式系统把检查点与消

息日志相结合。在这种情况下,在设置检查点后,一个进程在发送消息之前对它进行记录(称为基于发送者的日志)。一种可供选择的方法是让接收进程在把到来的消息传送到它所执行的应用程序之前先记录消息。这种方法也称为基于接收者的日志。当接收进程崩溃时,需要恢复到最近的检查点的状态,并从这个状态重放被发送的消息。把检查点与消息日志相结合使得有可能恢复到最近检查点的状态而无需付出设置检查点的代价。

在仅使用检查点方法与另外还使用日志的方法之间还有另一个重要的不同。在只使用检查点的系统中,进程将被恢复到设置检查点时的状态,它们的行为将与在发生故障之前不同。例如,因为通信时间是不确定的,消息可能以不同的次序发送,而导致接收者的不同响应。但是,如果使用了消息日志,那么就会发生从上次检查点之后事件的真实重放。这样的重放使得与外部世界相互作用更加容易。

我们来考虑一下由于用户提供了错误的输入而导致故障的情况。如果只使用检查点,那么系统为了恢复到真正相同的状态,就不得不在接受用户输入之前设置检查点。而如果还使用消息日志,就可以使用一个较老的检查点,然后在用户应该提供输入的那一点之前进行事件的重放。在实践中,使用较少的检查点与消息日志相结合的方式比使用很多检查点的方式更有效。

稳定存储

要恢复到先前的状态,就有必要安全地存储恢复所需的信息。在这种情况下,安全性意味着恢复信息不仅要经得住进程崩溃与站点故障的考验,而且还要能经得住各种存储介质故障的考验。当在分布式系统中进行恢复时,稳定存储起着重要的作用。我们在这里对它进行简单的讨论。

存储可以分为三类。首先是普通的 RAM 存储,当掉电或机器崩溃时其中的信息会丢失。然后是磁盘存储,它不受 CPU 故障的影响,但是在磁盘头损坏的情况下会丢失信息。

最后介绍一种稳定存储,它被设计可以经得住除了诸如洪水或地震之类的灾难以外的任何情况的考验。稳定存储可以使用一对普通的磁盘来实现,如图 7.22(a)所示。驱动器 2 中的每个块都是驱动器 1 中对应块的准确拷贝。当更新一个块时,首先对驱动器 1 中的块进行更新及验证,然后对驱动器 2 上的相同块进行相同的工作。

假定驱动器 1 已经更新,但是系统在更新驱动器 2 之前崩溃,如图 7.22(b)所示。在恢复时磁盘可以逐块进行比较。只要两个对应的块不同,就可以假定驱动器 1 是正确的(因为驱动器 1 总是在驱动器 2 之前进行更新),所以就把新的块从驱动器 1 拷贝到驱动器 2。完成恢复过程之后,两个驱动器就又是相同的了。

另外一个潜在的问题是块的自然损坏。微小的尘埃或日常的磨损都可以使以前有效的块突然在没有原因或警告的情况下发生校验和错误,如图 7.22(c)所示。当探测到这样的错误时,损坏的块可以从另一个驱动器上对应的块中重新生成。

由于它的特点,稳定存储很适合于诸如原子事务这样需要高度容错的应用程序。数据被写入稳定存储,然后再读出以验证它们是否被正确写入,这样以后数据丢失的机会就

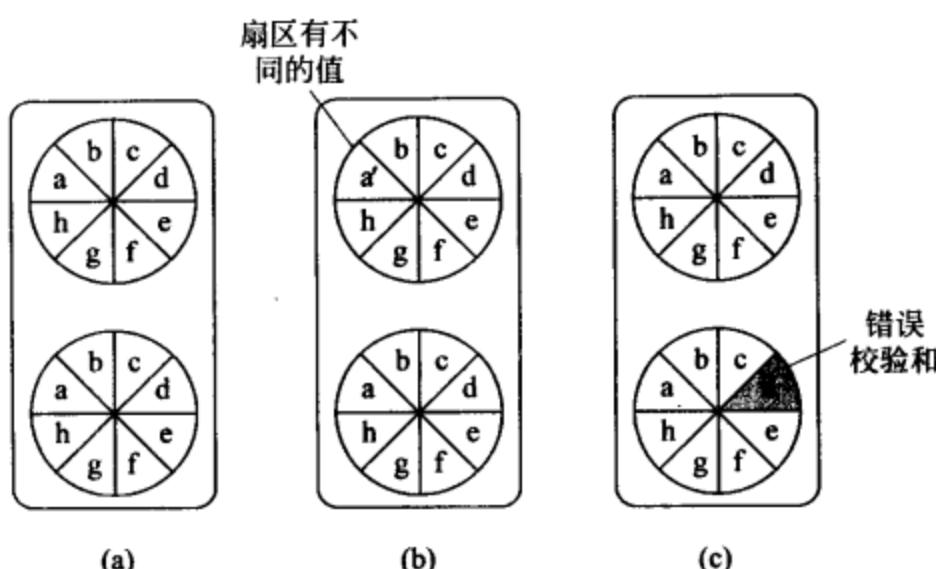


图 7.22

变得很小。

在下面两节中,我们将讨论与检查点和消息日志有关的更多细节。Elnozahy 等(1996)提供了分布式系统中检查点与日志的一个调查。可以在(Chow, Johnson 1997)中找到不同的算法细节。

7.6.2 檢查點

在容错的分布式系统中，回退错误恢复需要系统把它的状态有规律地保存到稳定存储中。分布式系统的状态已经在第 5 章进行了讨论。特别是，我们着重强调了记录一致全局状态的必要性，它也被称为分布式快照。在分布式快照中，如果进程 P 记录了一条消息的接收，那么就应该有一个进程 Q 记录了该消息的发送。毕竟消息必须是来自某个地方的。

在回退错误恢复方法中，每个进程都按时把它的状态保存到本地可用的稳定存储中。要在进程或系统失败之后进行恢复，就需要我们从这些局部状态中建立一致的全局状态。特别是，最好恢复到最近的分布式快照，这种快照也被称为恢复线。换句话说，恢复线就对应于最近的一致性切面，如图 7.23 所示。

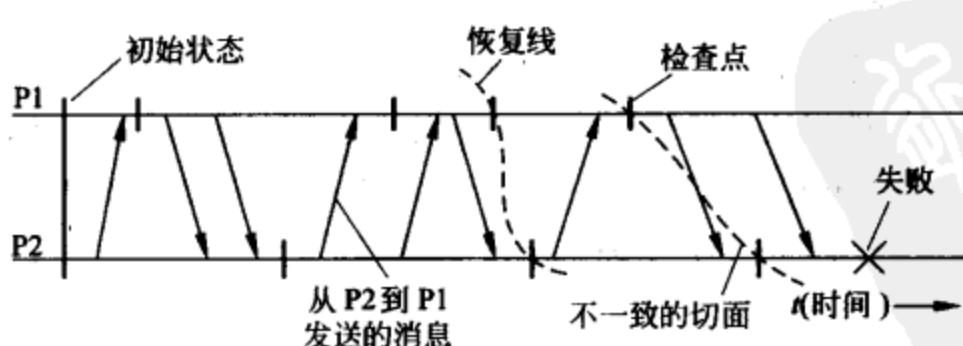


图 7.23 恢复线

1. 独立的检查点

不幸的是,每个进程的检查点是以一种不协调的方式来按时记录它的本地状态的,这种分布式本性使得要找到一个恢复线很困难。要找到恢复线需要每个进程都回退到它最近保存的状态。如果这些本地状态不能组成一个分布式快照,那么就需要进行进一步的回退。下面我们要说明一种寻找恢复线的方法。这种折叠回退的过程可能会导致多米诺效应,如图 7.24 所示。

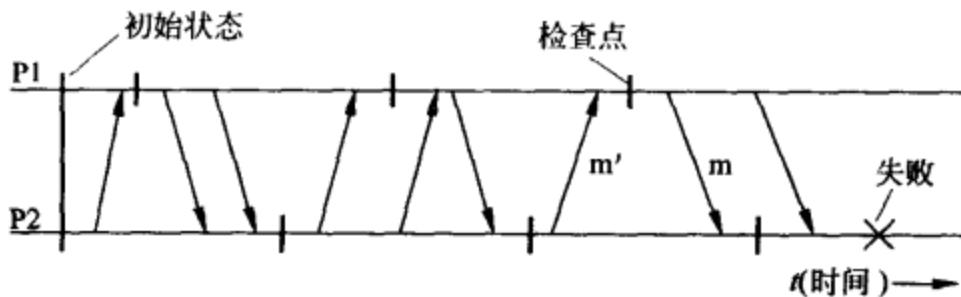


图 7.24 多米诺效应

当进程 P2 崩溃时,我们需要把它的状态恢复到最近保存的检查点。与之类似,进程 P1 也需要进行回退。不幸的是,这两个最近保存的本地状态不能组成一个一致的全局状态,P2 保存的状态表明了消息 m 的接收,但是没有其他进程说明其为发送者。因此,P2 需要回退到更早的状态。

但是,P2 要回退到的下一个状态不能用作分布式快照的一部分。在这种情况下,P1 将记录消息 m' 的接收,但是没有这一消息被发送的记录事件,因此有必要把 P1 回退到以前的状态。在这个例子中,恢复线就是系统的最初状态。

由于进程独立地设置本地检查点,所以这种方法也被称为独立的检查点方法。另一种可以选择的方法是像我们下面要讨论的那样设置全局协调的检查点,但是协调需要全局同步,这可能会引起一些性能问题。独立检查点的另一个缺点是每个本地存储都需要按时清理,例如通过运行一个特定的分布式垃圾收集器来进行清理。但是它主要的缺点还是恢复线的计算。

实现独立的检查点需要记录依赖关系,应按照进程可以共同回退到一个一致的全局状态的方法来进行记录。用 $CP[i](m)$ 来表示进程 P_i 采用的第 m 个检查点,用 $INT[i](m)$ 来表示检查点 $CP[i](m-1)$ 与 $CP[i](m)$ 之间的间隔。

当进程 P_i 按 $INT[i](m)$ 的间隔发送消息时,它就在接收进程中加入 (i, m) 对。当进程 P_i 按 $INT[j](n)$ 的间隔接收消息时,就使用 (i, m) 的索引对,它记录了 $INT[i](m) \rightarrow INT[j](n)$ 的依赖关系。然后,当 P_i 采用检查点 $CP[j](n)$ 时,它就把这个依赖关系与恢复信息的其他部分一起写入它的本地稳定存储中。

现在假定进程 P_i 在某个时刻需要回退到检查点 $CP[i](m-1)$ 。要确保全局一致性,我们需要确保所有从 P_i 那里接收到了按间隔 $INT[i](m)$ 发送的消息的进程都回退到在接收这个消息之前的检查点状态。特别是,在我们的例子中,进程至少需要回退到检查点

$CP[j](n-1)$ 。如果 $CP[j](n-1)$ 导致了不一致的全局状态,那么还需要进行更多的回退。

计算恢复线需要对每个进程在设置检查点时记录的时间间隔依赖关系进行分析。我们不讨论更多的细节,可以证明:这样的计算是相当复杂的,与协调检查点相比,独立检查点对这些计算的需求也是不尽合理的。另外,它不是支配性能因素的进程之间的协调,相反,需要把状态保存到本地稳定存储中,它还造成一些额外开销(Einozahy 等 1992)。因此,协调检查点使用得更为普遍,它也比独立检查点更为简单。

2. 协调检查点

顾名思义,在协调检查点中,所有的进程都同步地把它们的状态写到本地稳定存储中。协调检查点的主要优点是所保存的状态自动保持全局一致,这样就可以避免导致多米诺效应的折叠回退。可以使用第 5 章中讨论的分布式快照算法来协调检查点。这种算法是非阻塞的检查点协调的一个例子。

一种比较简单的方法是使用两阶段的阻塞协议。首先协调者向所有进程多播一个 CHECKPOINT_REQUEST 消息。当进程接收到这样的消息时就设置一个本地检查点,通过它正在执行的应用程序将随后传递给它的消息进行排队,然后向协调者确认它已经设置了检查点。当协调者接收到所有进程的确认时,它就多播一个 CHECKPOINT_DONE 消息让(阻塞的)进程继续执行。

很容易看出这种方法可以使全局状态一致,因为没有即将进来的消息注册为检查点的一部分。其原因在于任何跟随在设置检查点的请求之后的消息都不被认为是本地检查点的一部分。同时,在接收到 CHECKPOINT_DONE 消息之前对即将发出的消息(当通过运行的应用程序传递给检查点进程的时候)在本地进行排队。

对这种算法的一种改进是只对那些依赖协调者的恢复的进程多播检查点请求而忽略其他进程。如果一个进程在设置了上个检查点之后接收到了与协调者发送的消息直接或间接相关的信息,那么这个进程就是依赖于协调者的。这称为增量快照。

要设置一个增量快照,协调者只对那些在上次设置检查点之后对之发送了消息的进程多播检查点请求。当进程 P 接收到这样的请求时,它就把请求转发给那些在上次检查点之后 P 本身向之发送了消息的所有进程,等等。一个进程只转发一次请求。当确认所有的进程之后,使用第二次多播来引发实际的检查点,然后所有的进程继续执行。

7.6.3 消息日志

考虑到设置检查点是一个开销很大的操作,特别是该操作涉及到向稳定存储中写入状态,需要寻找一种可以减少检查点的数目但是还允许恢复的技术。在分布式系统中有一种重要的技术消息日志。

消息日志的基本思想是:如果消息的传输可以重放,那么我们就能够到达一个全局一致的状态而不需要从稳定存储中恢复该状态。它使用一个检查点状态作为开始点,然后简单地把从该点以后发送的所有消息都重发并进行处理。

假定采用分段确定模式(piecewise deterministic model),这种方法可以很好地工作。

在这种模式中,假定每个进程的执行都有一连串的间隔,事件是在这些间隔中发生的。这些事件与我们在第5章中讨论过的Lamport的先发生关系中的事件相同。例如,一个事件可能是一条指令的执行、一条消息的发送等。假定分段确定模式中的每个间隔都以一个非确定性的事件,例如一条消息的接收开始。但是,这个时刻之后进程的执行就是完全确定的。一个时间间隔以发生非确定性的事件之前的最后一个事件结束。

实际上,如果以相同的非确定性事件作为开始进行重放,那么一个间隔的重放可以具有已知的结果,也就是说,以一种完全确定的方式进行重放。因此,如果我们在这样的模式中记录下所有的非确定性事件,就有可能以一种确定的方式完全重放整个进程的执行。

考虑到要从进程崩溃中恢复到一个全局一致的状态时消息日志是必需的,准确地知道什么时候对消息进行记录是很重要的。Alvisi和Marzullo(1998)介绍的方法说明,如果我们关注现存的消息日志方案如何处理孤儿进程,那么可以容易地描述消息日志方案的特征。

孤儿进程就是在其他进程崩溃之后还存在的进程,但是它的状态与恢复之后的崩溃进程不一致。图7.25中的情况就是一个例子。进程Q从进程P与R那里分别接收到消息m₁和m₂,然后把消息m₂发送给R。但是,与其他消息不同,消息m₂没有进行日志记录。如果进程Q崩溃然后恢复,那么要恢复Q只需要重放记录了的消息,在我们的例子中就是m₁,因为m₂没有进行记录,它的传输也就不会被重放,这意味着m₂的传输不会发生。

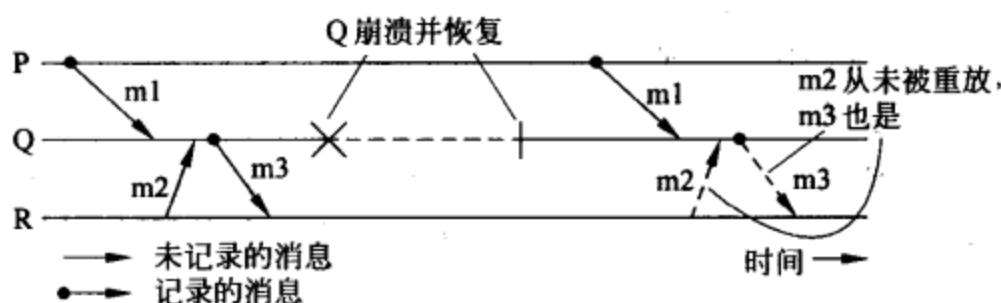


图7.25 在恢复之后不正确的重放导致孤儿进程

但是Q恢复之后的情况是与它恢复之前不一致的。特别是,R保留了在崩溃之前发送的消息,但是在重放崩溃之前发生的事情时该消息的接收与传送还没有发生。很明显,这样的不一致应该避免。

描述消息日志方案的特征

为了描述不同的消息日志方案的特征,我们使用在文献(Alvisi, Marzullo 1998)中说明的方法。认为每个消息m都具有头部,头部中包含了重发m并正确处理它所需的所有信息。例如,每个头部都包含了发送者和接收者的标识符以及一个用来识别它的序列号。另外还有一个传送号码用来确定它应该在何时被传送给接收应用程序。

如果消息不再被丢失就被称为是稳定的,例如因为它被写入了稳定存储。因此可以使用重放稳定消息的传输来进行恢复。

每个消息m都导致产生一个进程组DEP(m),它依赖于m的传送。特别是,

$DEP(m)$ 由那些接收 m 的进程组成。另外,如果另一个消息 m' 由于因果关系也要依赖 m 的传送,而且 m' 被传送给进程 Q ,那么 Q 也包含在 $DEP(m)$ 中。注意, m' 是按因果关系依赖于 m 的传送的,如果它由先前传送 m 的同一个进程发送,或者它传送了另一个按因果关系依赖于 m 的传送的消息。

$COPY(m)$ 进程集由那些具有 m 的拷贝,但是该拷贝不在它的本地稳定存储中的进程组成。当一个进程传送消息 m 时,它就也成为 $COPY(m)$ 的成员。注意, $COPY(m)$ 由那些可以移交 m 的拷贝的进程组成,这些拷贝可以用来重放 m 的传输。如果所有这些进程都崩溃了,那么重放 m 的传输显然就是不可能的。

使用这些表示,现在很容易就可以准确定义孤儿进程是什么了。假定一些进程在分布式系统中崩溃了, Q 是幸存的进程之一。如果 Q 包含在 $DEP(m)$ 中,在 $COPY(m)$ 中的每个进程都崩溃了的同时存在消息 m ,那么 Q 就是一个孤儿进程。换句话说,如果一个进程依赖于 m ,但是没有办法来重放 m 的传输时它就是一个孤儿进程。

要避免产生孤儿进程,需要确认:如果 $COPY(m)$ 中的每个进程都崩溃了,在 $DEP(m)$ 中没有幸存的进程。换句话说, $DEP(m)$ 中的所有进程都应该已经崩溃了。如果我们能保证不论何时一个进程成为了 $DEP(m)$ 的成员,它也要成为 $COPY(m)$ 的成员,那么就可以满足这个条件。换句话说,不论何时一个进程要依赖于 m 的传送,那么它就总要保持 m 的一份拷贝。

可以使用两种方法来做到这一点。第一种方法称为悲观日志协议(pessimistic logging protocol)。这个协议认为对于每个非稳定的消息 m ,都最多有一个进程依赖它。换句话说,悲观日志协议确保每个非稳定的消息 m 都最多被传送给一个进程。注意只要 m 被传送给进程 P , P 就成为 $COPY(m)$ 的成员。

最坏的情况就是 P 在 m 还没有被记录的时候就崩溃了。使用悲观日志,在传送 m 之后,如果还没有确保 m 被写到稳定存储就不允许 P 发送任何消息。因此,没有其他进程会依赖 m 的传送,这就没有了重放 m 的传输的可能性。这种方法可以避免孤儿进程。

相反,在乐观日志协议(optimistic logging protocol)中,在发生崩溃之后才进行实际的工作。特别是,假定对某个消息 m , $COPY(m)$ 中的每个进程都崩溃了。在乐观的方法中, $DEP(m)$ 中的任何孤儿进程都被回退到一个不再属于 $DEP(m)$ 的状态。很明显,乐观日志协议需要跟踪依赖关系,这使它们的实现变得更复杂。

像在(Einozahy 等 1996)中指出的那样,悲观日志要比乐观的方法简单得多,所以在实际的分布式系统设计中它是消息日志的首选方法。

7.7 小 结

容错是分布式系统设计中的一个重要主题。容错的定义是系统可以掩饰故障的发生并从故障中恢复。换句话说,如果系统可以在发生故障时继续操作,那么它就是容错的。

存在几种类型的故障。如果只是简单地停止了进程,那么就发生崩溃性故障。如果进程不对到来的请求做出响应,那么就发生了遗漏性故障。当进程对请求的响应过快或过慢时就发生定时性故障。如果以错误的方式响应到来的请求,那么就是响应性故障。

最难处理的失败是系统出现的任意类型故障,这时称为任意性故障或拜占庭故障。

冗余是获得容错性所需的关键技术。对于进程来说,进程组的概念很重要。一个进程组由很多进程组成,它们紧密协作来提供服务。在容错的进程组中,一个或多个进程的失效不会影响到该组实现的服务的可用性。通常,组内的通信必须高度可靠,而且为了获得容错性,应该具有严格的次序和原子属性。

可靠的组通信也称为可靠多播,它具有多种不同的形式。只要组相对较小,那么实现可靠性就是可行的。但是,如果需要支持非常大的组,可靠多播的可扩展性就成为一个问题。获得可扩展性的关键问题在于减少接收者发送的反馈消息数量,这些消息是用来报告多播消息的(不成功的)接收的。

如果需要提供原子性,那么事情就变得更复杂。在原子多播协议中,关键在于每个组成员都对多播消息应该被传送给哪个成员具有相同的视图。原子多播可以根据一种虚拟同步的执行模式进行准确的定义。这种模式引入了边界的概念;在边界之间组成员关系不发生变化,而且消息也可靠地进行传输。一条消息不会跨越边界。

组成员关系改变是每个进程都需要在组成员表上达成一致的一个例子。可以使用提交协议来达成这样的协议,其中两阶段提交协议是使用最广泛的。在两阶段提交协议中,首先一个协调者检查是否所有的进程都同意执行相同的操作(也就是是否它们都同意提交),然后在第二阶段中多播表决结果。可以使用三阶段提交协议来处理协调者崩溃的情况,而不必在协调者恢复之前使所有的进程阻塞以达成协议。

容错系统中的恢复可以通过有规律地对系统状态设置检查点来获得。检查点是完全分布式的。不幸的是,设置检查点是一个开销很大的操作。为了提高性能,很多分布式系统在检查点中结合使用了消息日志。通过记录进程之间的通信,有可能在发生崩溃之后重放系统的执行过程。

习 题

1. 可靠系统通常需要提供高度的安全性,为什么?
2. 是什么使得在崩溃性故障的情况下难以实现故障停机模式?
3. 考虑一个返回过期的缓存页而不是服务器上更近的更新版本的 Web 浏览器的情况,这种故障属于哪种故障?
4. 文中介绍的三倍模块冗余是否能处理拜占庭故障?
5. 图 7.2 可以处理多少失效的元素(设备加上表决者)? 给出一个例子说明它能掩盖的最坏情况。
6. TMR 是否能推广到每组 5 个元素而不是 3 个? 如果可以,它具有什么属性?
7. 对以下每个应用程序,你认为最多一次语义和最少一次语义哪个最好? 试讨论一下。
 - (a) 从文件服务器上读写文件;
 - (b) 编译一个程序;
 - (c) 远程银行。

8. 使用异步 RPC, 客户在它的请求被服务器接受之前阻塞(请参阅图 2.12)。失效会在什么程度上影响异步 RPC 的语义?
9. 给出一个组通信根本不需要消息排序的例子。
10. 在可靠多播中, 通信层是否总是需要为了重发的目的而保留消息的一份拷贝?
11. 原子多播的可扩展性重要到哪种程度上?
12. 我们在本章中说明, 在一个进程集上执行更新时, 原子多播可以节省时间。我们可以保证每个更新在什么程度上真正执行?
13. 虚拟同步类似于分布式数据存储中的弱一致性, 组视图的改变可以作为同步点。在这种情况下, 什么类似于强一致性呢?
14. 在图 7.14 的 FIFO 与完全排序相结合的多播情况下, 什么是可允许的传送顺序?
15. 在虚拟同步的情况下为协议建立下一个视图 G_{i+1} 以使它可以容忍进程发生故障。
16. 在两阶段提交协议中, 为什么即使在参与者们选择一个新的协调者的情况下也不会完全消除阻塞?
17. 在我们对三阶段提交的解释中, 看上去对事务的提交是基于多数表决的, 是这样吗?
18. 在分段确定执行模式中, 只记录消息就足够了吗? 我们还需要记录其他的事情吗?
19. 解释一下如何使用分布式事务中的先写日志来从故障中恢复。
20. 无状态服务器需要设置检查点吗?
21. 基于接收者的消息日志通常被认为比基于发送者的日志更好, 为什么呢?

第8章 安全性

我们要讨论的建立分布式系统的最后一个原则是安全性,但这绝不意味着安全性是最不重要的原则。然而,有人可以证明安全性是最难以处理的原则,因为安全需要普遍深入到整个系统中。有关安全的一个单纯的设计缺陷可能导致所有的安全措施无效。在本章中,我们集中讨论分布式系统中普遍加入的、支持安全的不同机制。

我们从介绍安全方面的基本问题开始。在对一个系统建立所有类型的安全机制并不都是真正有意义的,除非了解那些机制的使用方式以及该机制所预防的问题。这就需要我们了解要执行的安全策略。首先来讨论安全策略的概念以及有助于执行这样的策略的一些普通的机制设计问题。我们还简要地涉及到必要的加密方法。

分布式系统中的安全可以粗略地划分为两部分。一部分涉及可能处于不同机器上的用户或进程之间的通信。确保安全通信的主要机制是安全通道机制。安全通道,更明确地说,是身份验证、消息完整性以及机密性,这些内容将分别在一个单独的小节中讨论。

另一部分涉及授权,用以确保进程只获得授予它的对分布式系统内资源的访问权限。授权将在一个涉及访问控制的单独小节中介绍。除传统访问控制机制之外,我们还将集中讨论必须处理诸如代理一类的移动代码时的访问控制。

安全通道和访问控制需要分发加密密钥的机制,还需要向系统中添加或从系统中删除用户的机制。这些主题包括在称为安全管理的内容中。在一个单独的小节中,我们将讨论这样一些问题:加密密钥管理、安全组管理以及分发证明所有者有权对指定资源进行访问的证书。

最后,我们将通过讨论分布式系统中安全性如何显现的两个例子来使问题具体化。SESAME 是一个完整的系统,可以并入到分布式系统中来处理安全性。作为分布式系统中安全的一个完全不同的例子,我们将简要地讨论电子付费系统,该系统允许用户和商家在不同的地方安全地开始一个包括订货和付款的事务。

8.1 安全性介绍

我们以一些一般的安全性问题来开始对分布式系统中安全性的描述。首先,有必要对安全系统进行定义。我们将安全策略与安全机制区别开来,并考察明确规定了安全策略的 Globus 广域系统。其次,我们关注的是要为安全系统考虑一些一般的设计问题。最后,我们简要地讨论一些加密算法,这些算法在安全协议的设计中扮演关键的角色。

8.1.1 安全威胁、策略和机制

计算机系统中的安全性与可靠性的概念密切相关。非正式地说,一个可靠的计算机

系统是一个我们可以有理由信任来交付其服务的系统(Laprie 1995)。如第7章中所提到的,可靠性包括可用性、可信赖性、安全性和可维护性。然而,如果我们要信任一个计算机系统,就还应该考虑机密性和完整性。机密性是指计算机系统的一种属性,系统凭借此属性将信息只向授权用户公开。完整性是指对系统资源的变更只能以授权方式进行。换句话说,在安全的计算机系统中,不适当的变更应该是可以察觉的,并且是可以恢复的。任何计算机系统的主要资源都是其硬件、软件和数据。

考虑计算机系统中安全的另一种角度是我们试图保护该系统所提供的服务和数据不受到安全威胁。要考虑4种类型的安全威胁(Pfleeger 1997):

- (1) 窃听;
- (2) 中断;
- (3) 修改;
- (4) 伪造。

窃听是指一个未经授权的用户获得了对一项服务或数据的访问的情况。窃听的一个典型事例是两人之间的通信被其他人偷听到。窃听还发生在非法拷贝数据时,例如,发生在入侵者闯入文件系统中某个人的私人目录后。

中断的一个实例是文件被损坏或丢失时所出现的情况。一般来说,中断是指服务或数据变得难以获得、不能使用、被破坏等情况。在这个意义上说,服务拒绝攻击是一种安全威胁,它归为中断类,一些人正是通过它恶意地试图使其他人不能访问服务。

修改包括对数据未经授权的改变或篡改一项服务以使其不再遵循其原始规范。修改的实例包括窃听然后改变传输的数据、篡改数据库条目以及改变一个程序使其秘密记录其用户的活动。

伪造是指产生通常不存在的附加数据或活动的情况。例如,一个人侵者可能尝试向密码文件或数据库中添加一项。同样,有时可能通过重放先前发送的消息来侵入一个系统。我们稍后将在本章中讨论这样的例子。

注意,中断、修改和伪造中的每一个都可视为数据篡改的一种形式。

仅仅声明系统应该能够保护其自身免受任何可能的安全威胁并不是实际建立一个安全系统的方式。首先需要的是安全需求的一个描述,也就是说,一个安全策略。安全策略准确地描述系统中的实体能够采取的行为以及禁止采取的行为。实体包括用户、服务、数据、机器等。制定了安全策略之后,就可能集中考虑安全机制,策略通过该机制来实施。重要的安全机制包括:

- (1) 加密;
- (2) 身份验证;
- (3) 授权;
- (4) 审计。

加密是计算机安全的基础。加密将数据转换为一些攻击者不能理解的形式。换句话说,加密提供了一种实现机密性的方式。另外,加密使我们能够检查数据是否被修改,从而它还提供对完整性检查的支持。

身份验证用于检验用户、客户、服务器等所声明的身份。在客户的情况下,基本前提

是在客户可以使用一项服务之前,该服务必须了解客户的身份。通常,依靠密码对用户进行身份验证,但对客户进行身份验证还有许多其他方式。

对一个客户进行身份验证之后,有必要检查是否授予该客户执行所请求操作的权限。对医学数据库中记录的访问是一个典型实例。取决于访问该数据库的不同人员,可能分别赋予其读取记录、修改记录中特定字段或者添加或删除记录的许可。

审计工具用于追踪各个客户的访问内容以及访问方式。虽然审计并不真正提供任何防止安全威胁的保护,但审计记录对安全破坏的分析以及随后采取措施防止入侵非常有用。出于这种原因,攻击者通常不希望留下任何可能最终导致其身份暴露的痕迹。在这种意义上,对访问记录进行攻击有时成为更冒险的事情。

实例: Globus 安全结构

安全策略的概念和安全机制在分布式系统中为实施这样的策略所扮演的角色通常可以通过一个具体实例给予最好的解释。我们来考虑一下为 Globus 广域系统所定义的安全策略(Chervenak 等 2000)。Globus 是一个支持大规模分布式计算的系统,其中同时使用许多主机、文件和其他资源来进行计算。这样的环境还称为计算网格(Foster, Kesselman 1998)。这些网格中的资源通常位于不同的管理域,这些域可以位于世界的不同部分。

因为用户和资源在数量上是巨大的,而且广泛地散布在不同管理域中,所以安全是最基本的要求。要设计并正确使用安全机制,就需要理解确实需要保护的对象,以及有关安全的假定内容。稍作简化,Globus 的安全策略必须具有下述 8 条声明,我们将在下面进行解释(Foster 等 1998):

- (1) 该环境由多个管理域组成;
- (2) 局部操作(即仅在单个域内执行的操作)仅受局部网域安全策略的支配;
- (3) 全局操作(即包括若干域的操作)需要执行该操作的每个域都知道发起者;
- (4) 不同域中的实体之间的操作需要相互的身份验证;
- (5) 用全局身份验证取代局部身份验证;
- (6) 资源的访问控制仅受局部安全支配;
- (7) 用户可以将权限委派给进程;
- (8) 同一域内的一组进程可以共享凭证。

Globus 假设该环境由多个管理域组成,其中每个域都具有其自己的局部安全策略。假设仅因为该域在 Globus 内从而不能改变局部策略,而且 Globus 的全面策略不能覆盖局部安全决策。这样,Globus 中的安全就将自身限制于影响多个域的操作。

与这一问题相关的是,Globus 假设对一个域完全局部化的操作仅受该域的安全策略的支配。换句话说,如果一个操作是在单个域内开始并执行的,那么所有安全问题都将仅使用局部安全措施来执行。Globus 不会利用其他措施。

Globus 安全策略规定对操作的请求可以全局地也可以局部地开始。该发起者,不管是用户还是代表用户的进程,在执行该操作的每个域内都必须被该局部所了解。例如,一个用户可能具有一个映射为指定域的局部名称的全局名称。该映射发生的确切程度留待

每个域自己确定。

一个重要的策略声明是不同域中实体间的操作需要相互的身份验证。这就意味着，如果一个域内的一个用户使用来自另一个域的一项服务，那么就必须检验该用户的身份。同样重要的是，必须确保用户在使用其认为正在使用的服务。在本章的后面，我们将详细讨论身份验证。

我们将上面的两个策略问题与下面的安全需求结合在一起。如果已经检验过一个用户的身份，并且该用户已在一个域内被局部了解，那么该用户就可以像在该局部域内已经通过身份验证那样行动。这意味着，Globus 需要其系统范围的身份验证方法足够充分，从而认为用户在访问一个远程域(在这里用户是已知的)时已经通过该域的身份验证。由该域进行的其他身份验证应该是必要的。

用户(或代表用户的进程)通过身份验证后，仍然需要检验与资源相关的确切访问权限。例如，希望修改一个文件的用户首先必须通过身份验证，身份验证后就可以检查该用户是否确实被允许修改该文件。Globus 安全策略声明这种访问控制决策是在所访问的资源所处的域内完全局部化决定的。

要解释第 7 条声明，我们来考虑 Globus 内的一个移动代理程序，该代理程序通过逐个开始不同网域内的若干个操作来执行任务。这样的一个代理程序可能要花费很长时间完成其任务。为了避免与该代理程序所代表的用户进行通信，Globus 要求能够将用户权限的一个子集委派给进程。因此，通过对一个代理程序进行身份验证，然后检查其权限，Globus 应该能够允许代理程序开始一项操作而不必联系其所有者。

作为最后的策略声明，Globus 要求在单个域内运行并代表同一用户的进程组可以共享一个单独的凭证集。正如将在下面解释的那样，身份验证需要认证。这一声明不要求每个进程都具有其特有的惟一凭证集，从而在本质上开辟了身份验证的可扩展解决方案的道路。

Globus 安全策略允许其设计者集中开发一个全面的安全解决方案。通过假设每个域执行其自身的安全策略，Globus 仅注重涉及多个网域的安全威胁。特别是，该安全策略指出重要的设计问题是远程域内用户的表示方法，以及向用户或其代表分配来自一个远程网域中的资源。因此 Globus 首先需要的是跨网域的身份验证机制并使用户在远程网域内为人所知。

出于这一目的，该策略引入了两种类型的代理。用户代理是一个进程，系统允许该进程在有限的一段时间内代表用户进行操作。资源由资源代理表示。资源代理是在指定域内运行的一个进程，用于将一个资源上的全局操作转换为遵循特定网域的安全策略的局部操作。例如，需要访问一个资源时，用户代理通常会与该资源代理进行通信。

Globus 安全结构从本质上由诸如用户、用户代理、资源代理和一般进程这样的实体组成。这些实体位于域中，并互相影响。特别是，该安全结构定义了 4 个不同的协议，如图 8.1 所示(同时参阅文献(Foster 等 1998))。

第 1 个协议准确描述了用户创建用户代理和向该代理委派权限的方法。特别是，为了使该用户代理代表该用户行动，用户要赋予该代理一个适当的认证集。

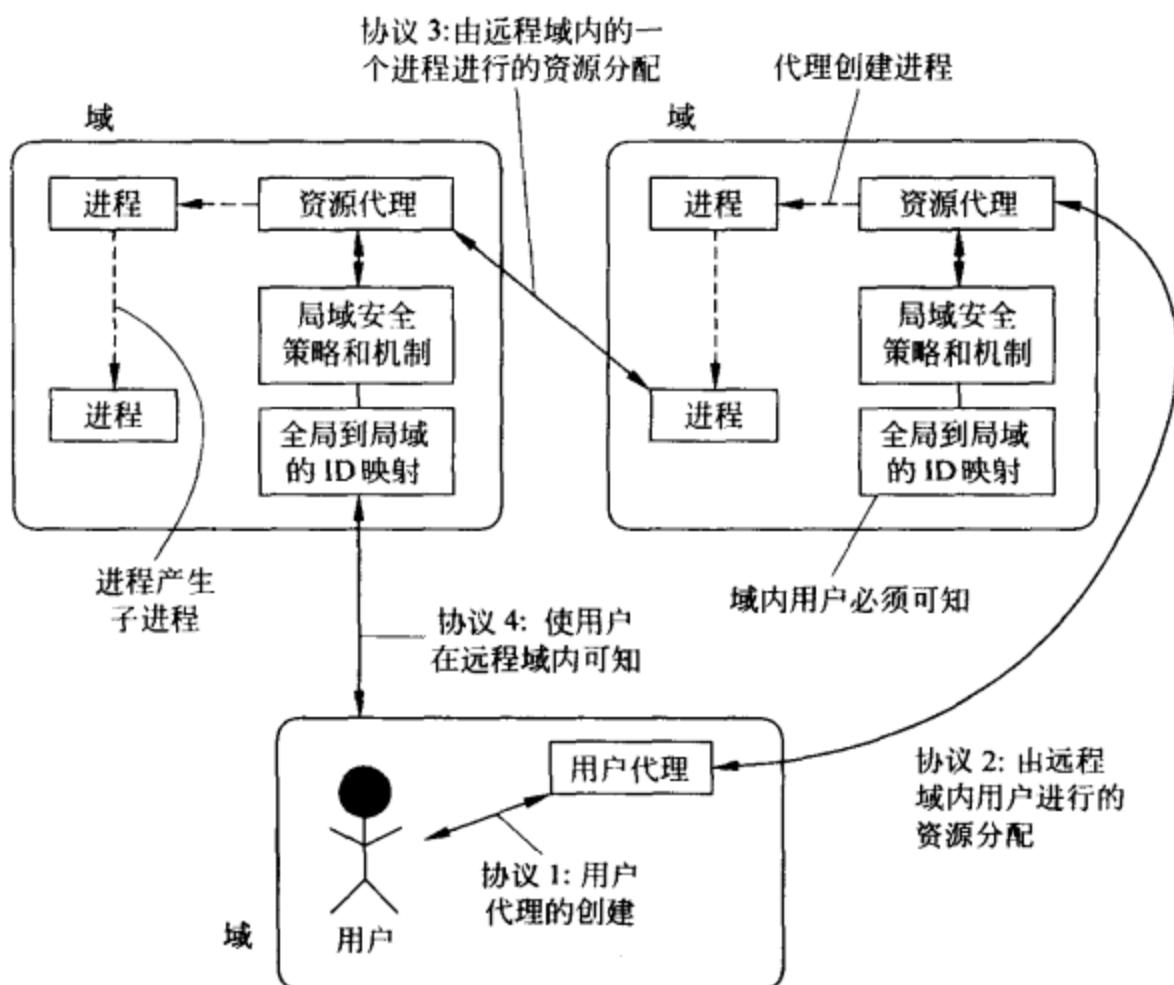


图 8.1 Globus 安全结构体系

第 2 个协议指定了用户代理对远程域中的资源提出分配请求的方式。从本质上说，该协议在进行相互身份验证后通知资源代理在远程域中创建一个进程。该进程代表该用户(正如用户代理所做的)，但在与被请求的资源所在的相同域内进行操作。有些资源由局限于该域的访问控制决策控制，系统赋予该进程访问这些资源的权限。

在远程域内创建的进程可以启动其他域内的附加计算。因此，在进程而不是用户代理请求远程域中的资源时就需要一个协议来分配该资源。在 Globus 中，这种类型的分配经由用户代理完成，通过使一个进程基本上遵循第 2 个协议来令其关联的用户代理请求资源分配来实现。

Globus 安全体系结构中的第 4 个也是最后一个协议是用户使其自身在一个域内可知的方式。假设用户在一个域内有一个账号，那么需要确认的就是用户代理所持的系统范围的凭证自动转换为指定域能识别的凭证。该协议规定了用户进行全局凭证和局部凭证之间映射的注册方式，注册在位于该域的一个映射表中进行。

每个协议的指定细节都在(Foster 等 1998)中描述。这里我们强调的是，如上所述，Globus 安全结构反映了其安全策略。用于实现该结构的机制，特别是上面所提到的协议，对许多分布式系统都是共同的，这一点将在本章中详细讨论。设计安全的分布式系统的困难并不是主要由安全机制的问题引起的，而是由如何使用那些机制以实现安全策略的决策引起的。在下一节中，我们考虑这样的一些设计决策。

8.1.2 设计问题

一个分布式系统,或者说任何计算机系统,都必须提供安全服务,使得范围广阔的安全策略可以通过这些服务来实现。实现多方面的安全服务时需要考虑很多重要的设计问题。在下面的内容中,我们讨论下面 3 个问题:控制的焦点、安全机制分层和简洁性(请参阅文献(Gollmann 1999))。

1. 控制的焦点

考虑一个(可能是分布式的)应用程序的保护时,基本上可以遵循 3 种不同的方法,如图 8.2 所示。第一种方法是把注意力直接集中于对与该应用程序相关联的数据的保护。我们使用“直接”这个词的意思是不考虑可能在数据项上执行的各种各样的操作,主要关心的是要确保数据完整性。通常,这种类型的保护出现在一些数据库系统中,在这些系统中可以规定各种完整性限制,每次修改一个数据项时系统自动检查这些限制(参见(Ullman 1988))。

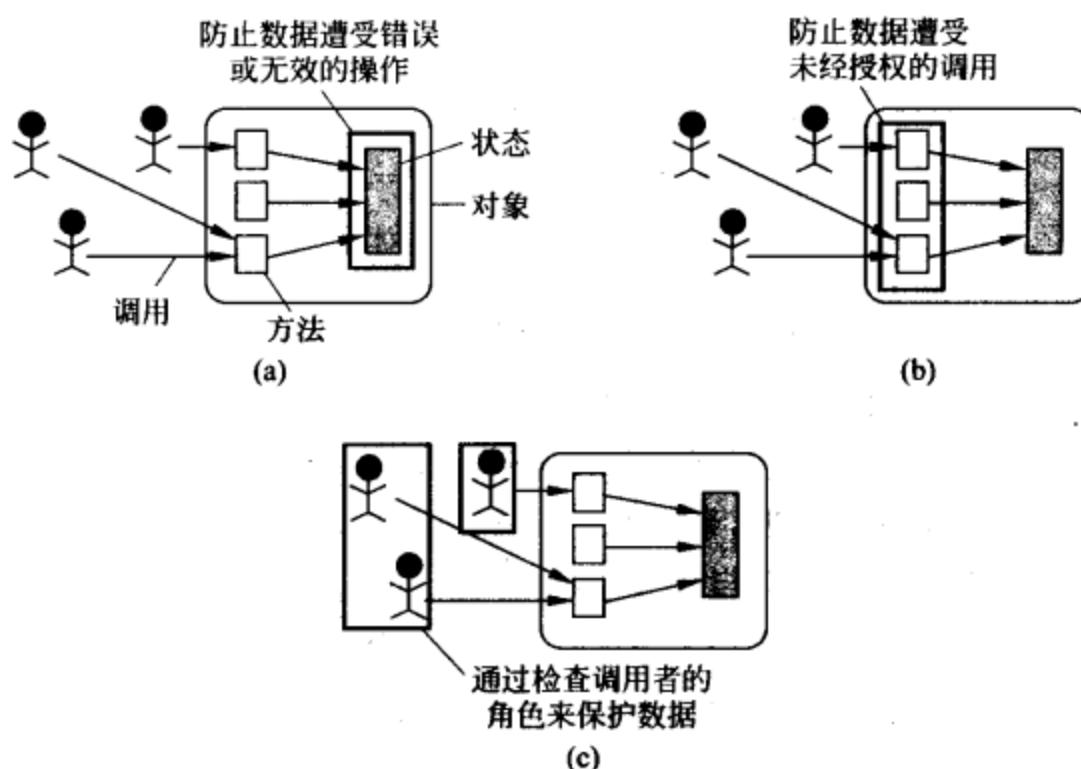


图 8.2 三种防止安全威胁的保护方法

(a) 防止无效操作的保护; (b) 防止未经授权调用的保护; (c) 防止未经授权用户的保护

第二种方法是把注意力集中于要访问特定数据或资源时,通过明确指定可以调用的操作,以及操作由谁完成来进行的保护上。在这种情况下,控制焦点与访问控制机制紧密相关,这一内容我们将在本章的后面部分进行详细讨论。例如,在一个基于对象的系统中,可以决定为客户可用的每个方法指定允许调用该方法的客户。换句话说,访问控制方法可以应用于一个对象所提供的完整接口或整个对象本身。因此这种方法考虑了访问控制的各种粒度。

第三种方法是把注意力直接集中于用户,通过采取一些措施使得只有指定的人可以访问该应用程序,而不考虑其要执行的操作来实现。例如,银行中的一个数据库可能通过

拒绝除高级管理人员外任何人的访问来获得保护。另一个例子是，在许多大学中，限制特定的数据和应用程序只能由教职员使用，而不允许学生访问。实际上，控制的焦点在于定义用户具有的角色。一个用户的角色经过检验之后，就可以确定他对一个资源的访问是允许还是被拒绝。作为安全系统设计的一部分，定义人们可能具有的角色并提供支持基于角色的访问控制的机制是必要的。在本章的后面部分我们将回到角色这个话题。

2. 安全机制分层

安全系统设计的一个重要问题是决定应将安全机制放置在哪一层。在这一环境中的层次与系统逻辑组织的层次相关。例如，计算机网络通常按照一些参考模型分层组织，正如我们在第 2 章中所讨论的那样。在第 1 章中，我们介绍了由应用程序、中间件、操作系统服务和操作系统核心这些单独的层组成的分布式系统的结构。将计算机网络和分布式系统分层的组织结合起来，一般会导致如图 8.3 所示的结果。

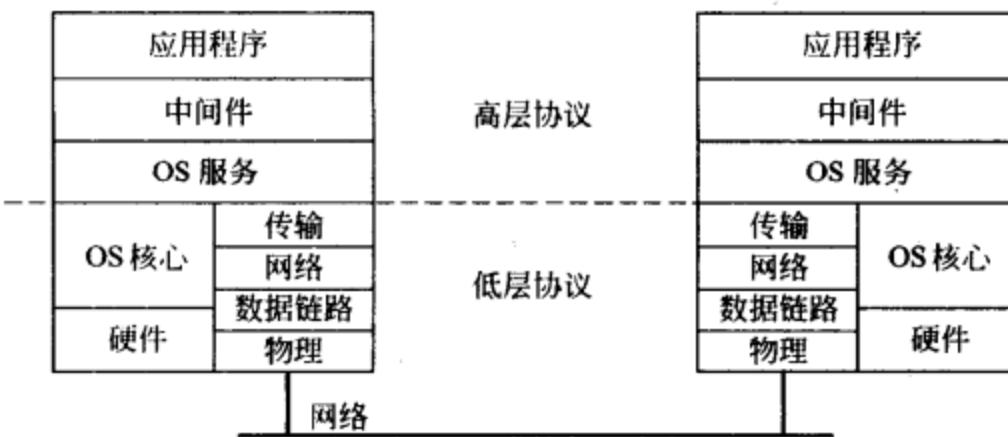


图 8.3 分布式系统的多层逻辑结构

实际上，图 8.3 将通用服务与通信服务分离开来。这一分离对理解分布式系统中的分层安全，尤其是信任的概念非常重要。信任和安全之间的差别很重要。一个系统或者是安全的，或者是不安全的（考虑各种概率的措施），但一个客户是否认为一个系统是安全的，则是一个信任问题（Pfleeger 1997）。安全机制放置在哪一层取决于客户对特定层中服务的安全程度所具有的信任度。

作为一个例子，考虑位于不同站点的一个组织，这些站点通过诸如 SMDS（switched multi-megabit data service，交换式多兆位数据服务）的通信服务连接起来。一个 SMDS 网络可以看作一个链接层主干网，该网络连接位于地理上可能分散的地点的多个局域网，如图 8.4 所示（有关 SMDS 的详细信息，请参阅（Klessig, Tesink 1995））。

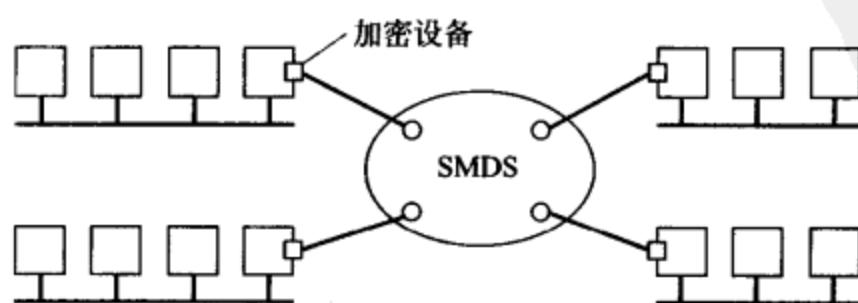


图 8.4 通过广域主干服务连接的若干站点

安全可以通过在每个 SMDS 路由器处放置加密设备来提供,如图 8.4 所示。这些设备自动对站点之间发送的数据包进行加密和解密,但并不另外对同一站点的主机之间提供安全通信。如果站点 A 的 Alice 向站点 B 的 Bob 发送一条消息,并且担心她的消息被人窃听,那么她就必须至少信任站间通信的加密工作正常。例如,这意味着她必须信任两个站点的系统管理员都已采取适当措施来防止设备损坏。

现在假设 Alice 并不信任站间通信的安全。于是她可能决定采取自己的措施,即使用一个传输层的安全服务,例如 SSL。SSL 代表加密套接字协议层(secure sockets layer),并且可以用于在一个 TCP 连接上安全地发送消息。我们将在第 11 章讨论 Web 中的安全时介绍 SSL 的细节问题。要注意的一个重要事项是 SSL 允许 Alice 建立与 Bob 的安全连接。所有传输级的消息都将被加密——SMDS 级的也是一样,但这对 Alice 无关紧要。在这一事例中,Alice 必须信任 SSL。换句话说,她相信 SSL 是安全的。

在分布式系统中,安全机制通常位于中间件层。如果 Alice 不信任 SSL,那么她可能希望使用一个本地的安全 RPC 服务。此外,她还必须信任这一 RPC 服务能够完成其许诺的任务,例如不泄露信息或适当地对客户和服务器进行身份验证。

位于分布式的中间件层的安全服务仅在其所依赖的安全服务确实安全的情况下才是可以信任的。例如,如果一个安全 RPC 服务的部分实现依靠 SSL,那么对该 RPC 服务的信任就取决于对 SSL 的信任程度。如果 SSL 的安全是不可靠的,就不可能信任该 RPC 服务的安全性。

3. 安全机制的分布

关于信任的服务之间的相关性导致了 TCB(trusted computing base, 信任计算库)概念的出现。TCB 是一个(分布式)系统中所有安全机制的集合,这些机制用于执行一个安全策略。TCB 越小越好。如果在现有的网络操作系统上将一个分布式系统作为中间件建立,那么其安全性就取决于作为基础的本地操作系统的安全性。换句话说,分布式系统中的 TCB 可以包括不同主机上的本地操作系统。

我们来考虑分布式文件系统中的文件服务器的情况。这样的一个服务器可能需要依赖其本地操作系统所提供的各种保护机制。这些机制不仅包括保护文件免受文件服务器外的进程访问的机制,还包括保护该文件服务器免受恶意破坏的机制。

因此基于中间件的分布式系统需要有对其依赖的现有本地操作系统的信任。如果不存在这样的信任,则本地操作系统的部分功能性就可能需要合并到分布式系统本身中。考虑一个微核操作系统,其中的许多操作系统服务作为普通用户进程运行。例如,在这种情况下,文件系统可以由一个为了适合分布式系统特定需求而进行裁减的系统(包括各种安全措施)完全取代。注意,正如第 1 章所述,这种方法可以用分布式操作系统逐渐取代基于中间件的分布式系统。

与这种方法一致的是根据所需要的安全性将服务分布在不同的机器上,以便将安全服务与其他类型的服务分开。例如,对一个安全的分布式文件系统来说,有可能通过把服务器放在一个具有可信操作系统并可能运行专用安全文件系统的机器上,从而将该文件服务器与客户隔离开,让客户及其应用程序位于不可信的机器上。

这一分离有效地将 TCB 简化为一个相对较少数量的机器和软件组件。通过随后保护那些机器免受外界的安全攻击,能够提升对分布式系统安全的全面信任。正如(Neumann 1995)中所描述的,RISSC(reduced interfaces for secure system components,安全系统组件简化接口)方法遵循了防止客户及其应用程序直接访问关键服务的原则。在 RISSC 方法中,任何安全性非常关键(security-critical)的服务器都放置在一台单独的机器上,与使用低级安全网络接口的终端用户系统隔离开,如图 8.5 所示。客户及其应用程序在不同机器上运行,并且仅能通过这些网络接口访问受保护的服务器。

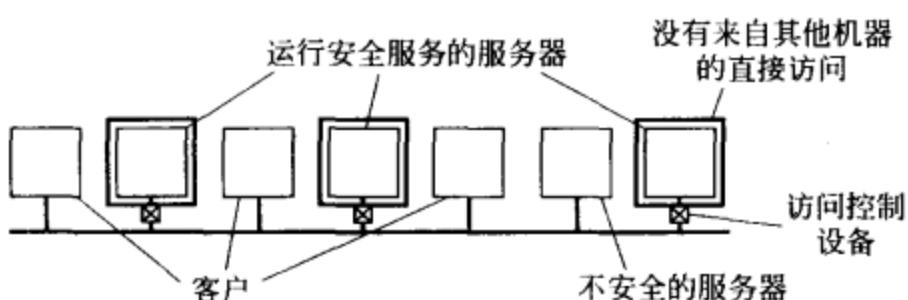


图 8.5 应用于安全分布式系统的 RISSC 原理

4. 简单性

另一个与决定将安全机制放在哪一层有关的重要设计问题是简单性问题。一般认为设计一个安全的计算机系统是一项困难的任务。因此,如果一个系统设计者可以使用一些易于理解且工作可靠的简单机制,设计工作就会较容易实现。

不幸的是,对实现安全策略来说简单机制通常不够充分。我们来再次考虑一下前面讨论的 Alice 希望向 Bob 发送一条消息的情况。链接层加密是一个简单且易于理解的机制,它用来防止对站点间消息通信的窃听。然而,如果 Alice 希望确保只有 Bob 能够接收到她的消息,就需要更多机制。在这种情况下,就需要用户级身份验证服务,并且 Alice 可能需要了解此类服务的工作方式以便投入其信任。因此尽管许多安全服务都是高度自动化且对用户隐藏的,但用户级身份验证可能至少还需要对加密密钥的概念的理解以及对诸如证书这样的机制的认识。

在其他情况下,应用程序自身本质上是复杂的,引入安全只会使事情变得更糟。复杂安全协议的一个实例应用领域(如我们在本章后面要讨论的)是一个数字支付系统。一次数字支付必须由多方进行通信才能完成,这就导致了数字支付的复杂性。在这些情况下,用于实现协议的基本机制要相对简单且易于理解,这一点很重要。简单性有助于最终用户对应用程序投入信任,更重要的是,对使设计者确信系统没有安全漏洞有很大贡献。

8.1.3 加密

在分布式系统中,安全的基本措施是使用加密技术。应用这些技术的基本思想是很简单的。我们来考虑一个发送方 S 希望向接收方 R 传送消息 m 的情况。要保护该消息免受安全威胁,发送方首先将其加密为一个难以理解的消息 m' ,然后将 m' 发送到 R。反过来,R 必须将接收到的消息解密为其原始格式 m。

加密和解密通过使用以密钥为参数的加密算法实现,如图 8.6 所示。发送的消息的原始格式称为明文,如图 8.6 中的 P 所示;已加密的格式称为密文,由 C 表示。

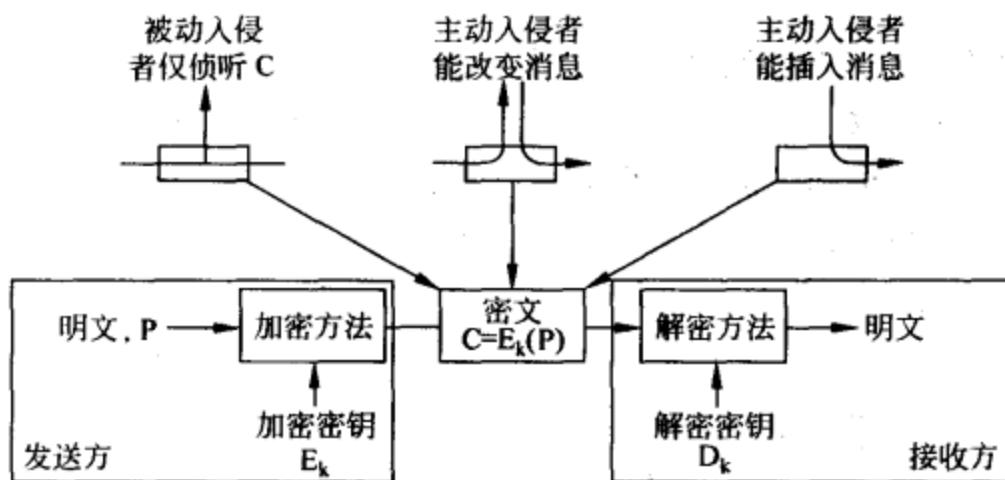


图 8.6 通信中的人侵者和窃听者

要描述为分布式系统建立安全服务所使用的各种安全协议,一个将明文、加密电文和密钥联系起来的符号表示法是很有用的。遵循普通的符号表示惯例,我们将使用 $C = E_k(P)$ 来表示通过使用密钥 K 对明文 P 进行加密获得密文 C 。同样,使用 $P = D_k(C)$ 来表示对使用密钥 K 的密文 C 的解密,其结果为明文 P 。

回到在图 8.6 中显示的例子。以密文 C 传输消息时,存在三种需要我们防止的不同攻击,加密对防止这些攻击是有帮助的。首先,人侵者可能在发送方或接收方都不知道发生窃听的情况下窃听该消息,当然,如果传输的消息已经以没有正确密钥就不易解密的方式进行了加密,那么窃听就是无用的;人侵者将只能看到不能理解的数据。(顺便说一下,仅一个消息正在传输这一事实有时就足以让人侵者得出结论。例如,如果在一次世界危机期间到白宫的通信流量突然下降到零,而到 Colorado 的一个特定山脉的通信流量以相同数量增加,那么,了解到这一点就可能获得有用的信息。)

需要处理的第二种类型的攻击是修改消息。修改明文是很容易的,修改经过适当加密的密文则困难得多。因为在能够有针对性地修改消息前,人侵者首先必须对该消息进行解密。此外,他还必须重新适当地对该消息进行加密,否则接收方就可能注意到消息已经被篡改。

第三种类型的攻击是当人侵者向通信系统中插入加密过的消息时,他们企图令 R 相信这些消息来自 S 。此外,正如我们将在本章后面所看到的,加密可以帮助防止这样的攻击。注意,如果人侵者能够修改消息,那么他同样能够插入消息。

不同加密系统之间存在一个基本区别,这一区别基于加密与解密密钥是否相同。在一个对称加密系统中,使用相同的密钥进行一个消息的加密和解密,换句话说:

$$P = D_k(E_k(P))$$

对称加密系统还称为密钥或共享密钥系统,因为需要发送方和接收方共享相同的密钥,要确保这种保护正常工作,这一共享密钥必须保密,不允许其他人看到此密钥。我们将使用符号 $K_{A,B}$ 来表示 A 和 B 共享的一个密钥。

在一个非对称加密系统中,加密和解密所使用的密钥是不同的,但两个密钥一起构成

惟一的一对,换句话说,存在一个单独的密钥 K_E 用于加密和一个用于解密的密钥 K_D ,例如

$$P = D_{K_D}(E_{K_E}(P))$$

非对称加密系统中的一个密钥是保密的,另一个是公开的。出于这种原因,非对称加密系统还称为公钥系统。在后面的内容中,我们使用符号 K_A^+ 来表示属于 A 的一个公钥, K_A^- 作为其相应的私钥。

我们来预先考虑一下本章后面有关安全协议的详细讨论。加密与解密密钥中的哪一个会实际作为公钥取决于两个密钥的使用方式。例如,如果 Alice 希望向 Bob 发送一个机密消息,那么她应该使用 Bob 的公钥加密该消息。因为 Bob 是惟一持有解密私钥的人,他同样也是惟一能够解密该消息的人。

另一方面,假设 Bob 希望确切知道他刚刚收到的消息确实来自 Alice。在这种情况下,Alice 可以保持其加密密钥为个人专用,并用它来加密其发送的消息。如果 Bob 使用 Alice 的公钥能够成功解密消息(并且该消息中的明文具有足够的信息可使其对 Bob 有意义),那么他就知道该消息确实来自 Alice,因为解密密钥与加密密钥是惟一绑定的。下面我们将回来详细介绍这样的算法。

在分布式系统中,密码学的一个最终应用是散列函数的使用。一个散列函数 H 将任意长度的消息 m 作为输入,并产生一个具有固定长度的位串 h 作为输出:

$$h = H(m)$$

一个散列 h 相当于在通信系统中向一条消息添加的附加位,这些附加位用于错误检测,例如循环冗余检查(CRC)。

加密系统中使用的散列函数具有很多属性。首先,这些函数是单向函数,意思是不可能计算出与已知输出 h 相应的输入 m 。另一方面,从 m 计算 h 是很容易的。其次,这些函数具有弱抗冲突性 weak collision resistance,意思是给定一个输入 m ,其相关输出 $h = H(m)$,不可能通过计算找到另一个不同的输入 $m' \neq m$,使 $H(m) = H(m')$ 。最后,加密散列函数还具有强抗冲突性 strong collision resistance,这意味着仅给定 H 时,不可能计算出任何两个不同的输入值 m 和 m' ,使 $H(m) = H(m')$ 。

相似的性质适用于任何加密函数 E 及所使用的密钥上。也就是说,对任何加密函数 E 来说,给定明文 P 和相关的密文 $C = E_K(P)$,都不可能通过计算找到密钥 K 。同样,类似于抗冲突性,给定明文 P 和密钥 K 时,应该不可能找到另一个密钥 K' ,使 $E_K(P) = E_{K'}(P)$ 。

为加密系统设计算法的技术和科学有很长的历史(Kahn 1967),建立安全系统通常异乎寻常地困难,或者甚至不可能(Schneier 2000)。详细讨论这些算法中的任何一种都超出了本书的范围。然而,为了对计算机系统中的加密提供一些概念,我们现在简要介绍三个有代表性的算法。有关这些算法和其他加密算法的详细信息可在(Kaufman 等 1995,Menezes 等 1996,Schneier 1996)中找到。

在我们详细研究各种协议的细节之前,图 8.7 总结了我们使用的符号和缩写。

符号	描述
$K_{A,B}$	A 和 B 共享的密钥
K_A^+	A 的公钥
K_A^-	A 的私钥

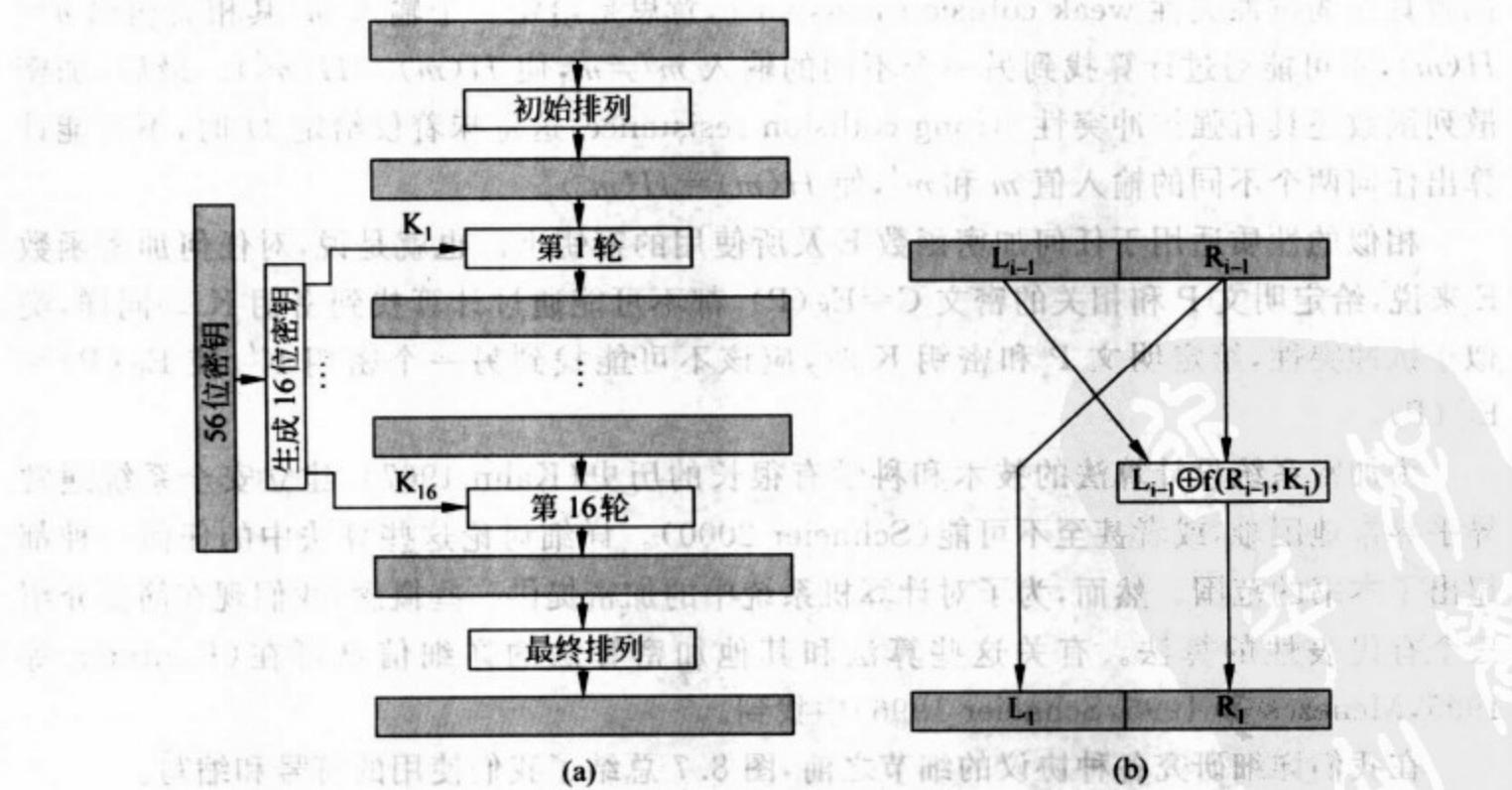
图 8.7 本章中使用的符号

1. 对称加密系统: DES

我们的第一个加密算法的例子是用于对称加密系统的 DES(data encryption standard, 数据加密标准)。DES 设计成运行在 64 位数据块上。数据块在 16 轮内转换为一个加密的(64 位)输出块, 每一轮使用一个不同的 48 位密钥进行加密。这 16 个密钥中的每一个都来源于一个 56 位的主密钥, 如图 8.8(a) 所示。一个输入块在开始其 16 轮的加密之前, 首先要进行初始排列, 其逆随后应用于加密过的输出上, 导致最终的输出数据块。

每个加密轮 i 将前一轮 $i-1$ 产生的 64 位数据块作为其输入, 如图 8.8(b) 所示。这 64 位分为左边部分 L_{i-1} 和右边部分 R_{i-1} , 每部分包含 32 位。右边部分作为下一轮的左边部分, 也就是说, $L_i = R_{i-1}$ 。

困难的工作在 mangler(切碎)函数 f 中完成。此函数将一个 32 位的数据块 R_{i-1} 作为输入, 加上一个 48 位的密钥 K_i , 产生一个 32 位的数据块与 L_{i-1} 进行 XOR 运算以产生 R_i 。(XOR 是异或运算的缩写。) mangler 函数首先将 R_{i-1} 扩展为 48 位的数据块并与 K_i 进行 XOR 运算。结果被划分为 8 块, 每块 6 位。然后将每块输入到不同的 S-box 中, S-box 是一种运算, 将有 64 种可能的 6 位输入的每一种都替换为有 16 种可能的 4 位输出的一种。然后每块 4 位的 8 个输出块结合为一个 32 位值并重新排列。



(a) DES 的原理; (b) 一个加密轮次的概要

第*i*轮的48位密钥 K_i 是从56位的主密钥中得到的，其过程如下所述。首先，排列主密钥并将其分为两个28位的等分。对每轮来说，每一半首先都左移一位或两位，之后就取出24位。先取出的24位与从移动后的另一半中取出的24位在一起，就构成了一个48位的密钥。一个加密轮次的细节如图8.9所示。

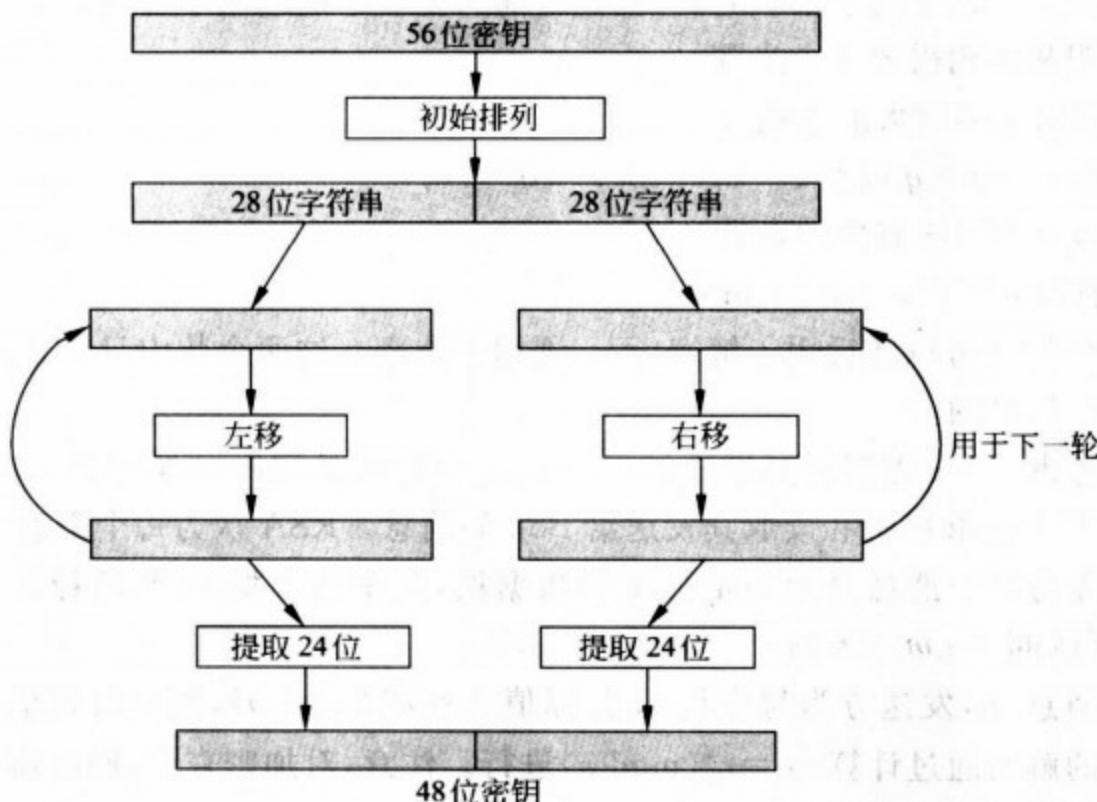


图8.9 DES中每轮密钥生成的细节

DES的原理是非常简单的，但使用分析方法难以破解该算法。正如最近所证明的，通过简单地搜索能够使用的密钥进行的粗野攻击已经变得很容易防范了。在一个具有不同密钥的特殊的加密—解密—加密模式中三次使用DES目前看来是安全的。

使DES难以通过分析进行攻击的原因是设计背后的基本原理从来没有清晰地解释过。例如，已经显示，采用其他S-box而不用当前标准中所使用的那些，会使该算法很大程度上易于破解（请参阅Pfleeger 1997以获得DES的一个简要分析）。设计的基本原理和S-box的使用仅在20世纪90年代设计出“新的”攻击模型后才公布。DES证实对这些攻击非常有抵抗力，且其设计者透露1974年他们开发DES时就已经知道新发明的攻击模型了（Coppersmith 1994）。

多年以来，DES已经作为标准加密技术使用了，但目前正处在被128位的Rijndael算法数据块取代的过程中。还有具有更大密钥和更大数据块的变体。该算法设计得非常快，以至于可以在智能卡上实现，这就形成了一个日益重要的密码学应用领域。

2. 公钥加密系统：RSA

我们的第二个加密算法实例广泛用于公钥系统，这就是RSA。RSA按其发明者Rivest、Shamir和Adieman名字的首字母命名（1978）。RSA的安全性来自这样一个事实，即没有任何已知的方法能够有效地找到大数的素因子。可以看出每个整数都可以写

成素数的乘积。例如,2100 可以写成:

$$2100=2\times2\times3\times5\times5\times7$$

这样就使 2、3、5 和 7 成为 2100 的素因子。在 RSA 中,私钥和公钥都是由非常大的素数(由上百个十进制数字组成)构成的。结果证明,破解 RSA 相当于寻找这两个素数。尽管数学家已经致力于这个问题很长时间,但到目前为止显示的结果不可能计算出来。

产生私钥和公钥需要 4 个步骤:

- (1) 选择两个非常大的素数, p 和 q 。
- (2) 计算 $n = p \times q$ 以及 $\phi = (p-1) \times (q-1)$ 。
- (3) 选择一个与 ϕ 有关的素数 d 。
- (4) 计算数 e 使得 $e \times d = 1 \pmod{\phi}$ 。

其中一个数 d 可以随后用于解密,而 e 则用于加密。这两个数中只能公开一个,取决于使用该算法的目的。

我们来考虑 Alice 希望将其发送给 Bob 的消息保密的情况。换句话说,她希望确保除 Bob 外没有人能够窃听和读取其发送给 Bob 的消息。RSA 认为每个消息 m 只是一个比特串。首先将每个消息分为固定长度的数据块,其中每个块 m_i 被解释为一个二进制数,应该位于区间 $0 \leq m_i < n$ 内。

要加密消息 m ,发送方为每个块 m_i 计算值 $c_i = m_i^e \pmod{n}$,然后将该值发送给接收方。接收方的解密通过计算 $m_i = c_i^d \pmod{n}$ 进行。注意,对加密来说,同时需要 e 和 n ,而解密则需要知道 d 和 n 的值。

将 RSA 和诸如 DES 一类的对称加密系统进行比较时,可以发现 RSA 具有计算上更为复杂的缺点。结果证明,使用 RSA 加密消息比 DES 慢大约 100~1000 倍,这取决于所使用的实现技术。结果是,许多加密系统使用 RSA 来以安全方式交换共享密钥,但很少用它来实际加密“标准”数据。稍后我们将在下面的小节中介绍这两种技术的结合实例。

3. 散列函数: MD5

作为广泛使用的加密算法的最后一个实例,我们来讨论 MD5(Rivest 1992)。MD5 是一个散列函数,用来从一个任意长度的二进制输入串计算一个 128 位固定长度的消息摘要。首先将该输入串填充至总长度 448 位(以 512 为模),然后原始比特串的长度作为一个 64 位整数加进去。实际上,该输入被转换为一串 512 位数据块。

该算法的结构如图 8.10 所示。以一些 128 位的常数值开始,该算法在 k 个阶段中进行,其中 k 是组成填充消息的 512 位数据块的数量。在每个阶段中,来自填充消息的 512 位数据块和前一阶段计算出的 128 位摘要计算出一个 128 位的摘要。MD5 中的一个阶段由 4 轮计算组成,其中每轮使用下面的 4 个函数中的一种:

$$\begin{aligned} F(x, y, z) &= (x \text{ AND } y) \text{ OR } ((\text{NOT } x) \text{ AND } z) \\ G(x, y, z) &= (x \text{ AND } z) \text{ OR } (y \text{ AND } (\text{NOT } z)) \\ H(x, y, z) &= x \text{ XOR } y \text{ XOR } z \\ I(x, y, z) &= y \text{ XOR } (x \text{ OR } (\text{NOT } z)) \end{aligned}$$

这些函数的每一个都对 32 位变量 x 、 y 和 z 进行运算。为了说明这些函数的使用方法,我们先看一个例子。

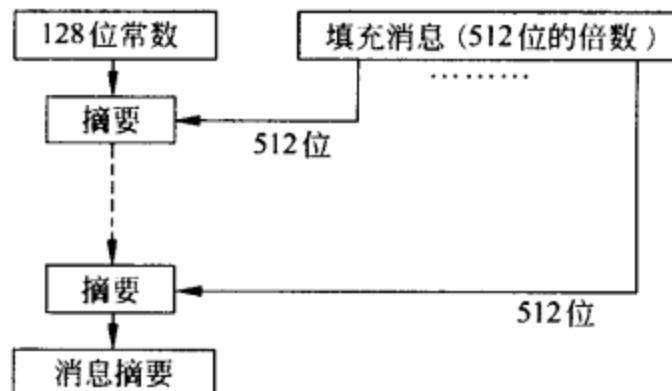


图 8.10 MD5 的结构

式,请考虑来自填充消息的一个 512 位数据块 b ,该数据块正在第 k 阶段进行处理。数据块 b 被划分为 16 个 32 位的子块 b_0, b_1, \dots, b_{15} 。在第一轮中,16 次迭代使用函数 F 来改变 4 个变量(分别表示为 p, q, r 和 s),如图 8.11 所示。这些变量传到各自的下一轮,一个阶段结束后,继续传到下一个阶段。总共有 64 个预定义常量 C_i 。符号 $x \lll n$ 表示左移, x 中的位向左移动 n 个位置,左边移出的位放在最右边的位置上。

第 1~8 次迭代	第 9~16 次迭代
$p \leftarrow (p + F(q, r, s) + b_0 + C_1) \lll 7$	$p \leftarrow (p + F(q, r, s) + b_8 + C_9) \lll 7$
$s \leftarrow (s + F(p, q, r) + b_1 + C_2) \lll 12$	$s \leftarrow (s + F(p, q, r) + b_9 + C_{10}) \lll 12$
$r \leftarrow (r + F(s, p, q) + b_2 + C_3) \lll 17$	$r \leftarrow (r + F(s, p, q) + b_{10} + C_{11}) \lll 17$
$q \leftarrow (q + F(r, s, p) + b_3 + C_4) \lll 22$	$q \leftarrow (q + F(r, s, p) + b_{11} + C_{12}) \lll 22$
$p \leftarrow (p + F(q, r, s) + b_4 + C_5) \lll 7$	$p \leftarrow (p + F(q, r, s) + b_{12} + C_{13}) \lll 7$
$s \leftarrow (s + F(p, q, r) + b_5 + C_6) \lll 12$	$s \leftarrow (s + F(p, q, r) + b_{13} + C_{14}) \lll 12$
$r \leftarrow (r + F(s, p, q) + b_6 + C_7) \lll 17$	$r \leftarrow (r + F(s, p, q) + b_{14} + C_{15}) \lll 17$
$q \leftarrow (q + F(r, s, p) + b_7 + C_8) \lll 22$	$q \leftarrow (q + F(r, s, p) + b_{15} + C_{16}) \lll 22$

图 8.11 MD5 中一个阶段的第一轮期间的 16 次迭代

第二轮以相似的方式使用函数 G ,而 H 和 I 则分别在第三轮和第四轮中使用。因此每一步都由 64 次迭代组成,随后开始下一个阶段,而且 p, q, r 和 s 在该点所具有的值也带入下一阶段。

8.2 安全通道

在前面的章节中,我们通常使用客户-服务器模型作为一种便利的方式来组织分布式系统。在此模型中,服务器可能是分布式和复制的,同时也可能充当与其他服务器相关的客户。考虑分布式系统中的安全性时,重新以客户和服务器的观点来思考是很有用的。特别是,使一个分布式系统安全性从本质上归结为两个主要问题。第一个问题是如何使客户与服务器之间的通信保持安全。安全通信需要对通信各方进行身份验证,同时还要确保消息完整性和可能的机密性。作为这一问题的一部分,我们还需要考虑保护一组服务器内的通信。

第二个问题是授权问题,即服务器一旦接受来自客户的请求,它将如何查明该客户是否得到授权使请求得以执行?身份验证与资源访问控制问题相关,这一问题我们将在下一小节中详细讨论。在本节中,我们集中于分布式系统内通信的保护。

客户和服务器之间通信的保护问题可以认为是在通信各方之间建立一个安全通道的问题(Voydock, Kent 1983)。安全通道保护发送方和接收方免受对消息的窃听、修改和伪造的攻击。对防止中断的保护不是必要的。保护消息免受窃听是通过确保机密性实现的,安全通道可确保入侵者不能窃听到其消息。防止入侵者的修改和伪造是通过相互身份验证和消息完整性的协议实现的。在下面的内容中,我们首先讨论可以用于身份验证的各种协议,这些协议使用对称加密系统和公钥加密系统。身份验证的基础逻辑的详细描述可以在(Lamson 等 1992)中找到。我们将分别讨论机密性和消息完整性。

8.2.1 身份验证

在详细阐述各种身份验证的细节之前,值得注意的是身份验证和消息完整性相互之间不能脱离开。例如,考虑一个分布式系统,该系统支持两个通信方的身份验证,但不提供确保消息完整性的机制。在这样的一个系统中,Bob 可以确切知道 Alice 是否为 m 的发送者。然而,如果 Bob 不能获得 m 在传输期间没有被修改的保证,那么即使他知道 Alice 发送了(原始版本的) m 又有什么用呢?

同样,假设系统只支持消息完整性,但不存在身份验证机制。Bob 接收到一个消息宣称他刚刚在彩票中赢得了 \$ 1 000 000 时,如果他不能检验该消息是否是由该彩票的组织者发送的,那么他能高兴到什么程度呢?

因此,身份验证和消息完整性应该结合在一起。在许多协议中,这一结合一般按下面的方式进行。我们再一次假设 Alice 和 Bob 希望进行通信,并且 Alice 主动建立通道。Alice 通过向 Bob 发送一条消息,或者另外向将帮助建立通道的一个可靠的第三方发送消息来开始通信。通道建立后,Alice 就确切知道她在与 Bob 通话,并且 Bob 也确切知道他在与 Alice 通话,于是他们可以交换消息了。

要确保进行身份验证之后交换的数据消息的完整性,常见的方法是依靠会话密钥使用密钥加密。会话密钥(session key)是一个共享密钥,用于为完整性和可能的机密性而对消息进行加密。这样的密钥一般仅在通道存在的时候使用。通道关闭时,就丢弃(或者实际上安全地破坏)与其相关的会话密钥。下面我们回来讨论会话密钥。

1. 基于共享密钥的身份验证

我们从考虑一个基于已经在 Alice 和 Bob 之间共享的安全密钥的身份验证协议开始,稍后在本章中讨论双方试图以安全方式获得一个共享密钥的实际方法。在该协议的描述中,Alice 和 Bob 分别缩写为 A 和 B,且其共享密钥表示为 $K_{A,B}$ 。该协议采用了一种普通的方法,即一方向另一方质询一个响应,而该响应只有在另一方知道共享密钥的情况下是正确的。这种解决方法也称为质询—响应协议(challenge-response protocol)。

在基于共享密钥的身份验证的情况下,该协议的进行如图 8.12 所示。首先,Alice 将其身份(消息 1)发送给 Bob,指出她希望在两人之间建立一个通信通道,随后 Bob 向

Alice 发送一个质询 R_B (消息 2)。这样的一个质询可以采取随机数的形式。需要 Alice 使用她与 Bob 共享的密钥 $K_{A,B}$ 加密该质询，并将加密过的质询返回给 Bob。此响应在图 8.12 中被表示为消息 3，它包含 $K_{A,B}(R_B)$ 。

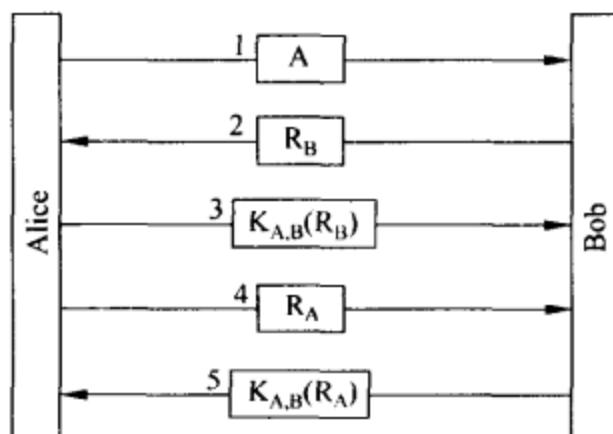


图 8.12 基于共享密钥的身份验证

Bob 接收到对其质询 R_B 的响应 $K_{A,B}(R_B)$ 时, 可以再次使用该共享密钥对该消息进行解密以查看其中是否包含 R_B 。如果包含, 那么他就知道 Alice 在另一边, 其他谁能够首先用 $K_{A,B}$ 加密 R_B ? 换句话说, Bob 现在已经检验出他确实在和 Alice 交谈。然而, 注意, Alice 还没有检验出通道的另一边确实是 Bob。因此, 她发送一个质询 R_A (消息 4), Bob 通过返回 $K_{A,B}(R_A)$ 响应该消息(消息 5)。Alice 使用 $K_{A,B}$ 解密该消息并看到 R_A 时, 她就知道在与 Bob 交谈。

安全中一个较困难的问题是设计能实际工作的协议。为了描述事情发生错误是多么容易, 我们来考虑一个“最优化”的身份验证协议, 其中的消息数量从 5 减至 3, 如图 8.13 所示。基本思想是如果 Alice 最终希望用任何方式质询 Bob, 那么她在建立通道时可能还会与其身份一起发送一个质询。同样, Bob 在单个消息中返回其对该质询的响应, 以及自己的质询。

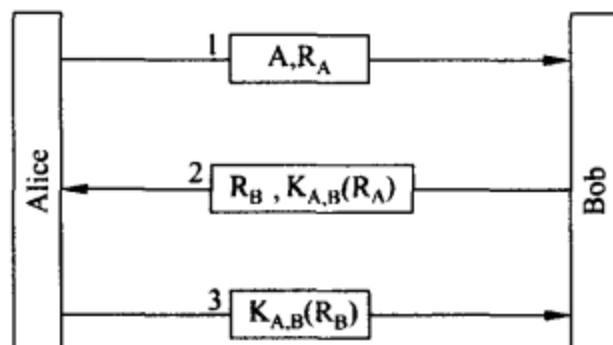


图 8.13 基于共享密钥, 但使用 3 个而不是 5 个消息的身份验证

不幸的是, 此协议不再工作。它可以轻易地被称为反射攻击(reflection attack)的行为打败。为了解释这种攻击的工作方式, 假设有一个名为 Chuck 的入侵者, 在我们的协议中将其表示为 C。Chuck 的目标是与 Bob 建立一个通道, 以使 Bob 相信他在与 Alice 交谈。如果 Chuck 正确响应由 Bob 发送的一个质询, 例如, 通过返回 Bob 发送的数字的加密版本, 那么他就可以建立这个通道。在不知道 $K_{A,B}$ 的情况下, 只有 Bob 可以进行这

样的加密,而这恰恰正是 Chuck 哄骗 Bob 所做的事。

该攻击在图 8.14 中进行了解释。Chuck 通过发送一条包含 Alice 的身份标识 A 的消息以及一个质询 R_c 开始通信,Bob 用单条消息返回其质询 R_B 和响应 $K_{A,B}(R_c)$ 。这时,Chuck 需要通过返回给 Bob 的 $K_{A,B}(R_B)$ 来证明他知道该密钥。不幸的是,他没有 $K_{A,B}$ 。作为替代,他所做的就是试图建立第二条通道以使 Bob 为其进行加密。

因此,Chuck 像以前 Alice 所做的那样用单条消息发送 A 和 R_B ,但现在假称需要第二条通道。这在图 8.14 的消息 3 中显示。Bob 并没有认识到他自己先前已经使用过 R_B 作为质询,就用 $K_{A,B}(R_B)$ 和另一个质询 R_{B2} 进行响应(消息 4)。那时,Chuck 就具有了 $K_{A,B}(R_B)$,并通过返回包含 $K_{A,B}(R_B)$ 响应的消息 5 完成了第一个会话的建立,该响应最初是由消息 2 中发送的质询请求的。

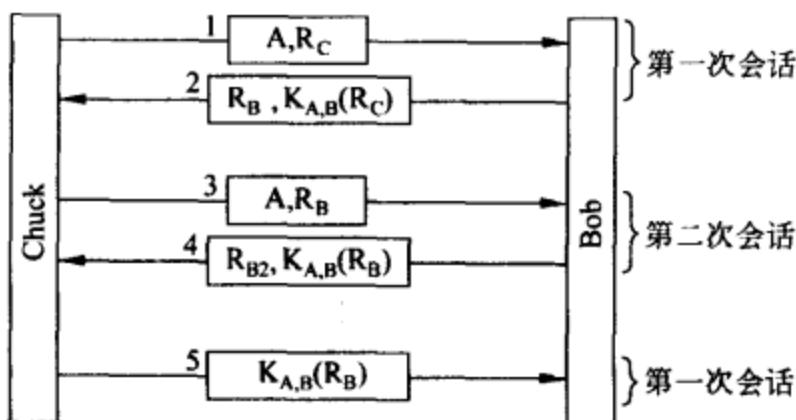


图 8.14 反射攻击

如(Kaufman 等 1995)中所解释的,原始协议改编过程中所犯的一个错误是,协议新版本中的双方在协议的两个不同方向都使用相同的质询。一个较好的设计是永远为发起者和响应者使用不同的质询。例如,如果 Alice 总使用奇数且 Bob 总使用偶数,那么 Bob 接收到图 8.14 中的消息 3 中的 R_B 时,就会认识到发生了一些可疑的事情。(不幸的是,此解决方法常遭受其他攻击,特别是称为“中间人攻击(man-in-the-middle-attack)”的攻击,该攻击在(Schneier 1996)中进行了解释)。一般说来,让建立安全通道中涉及的双方同等地做很多事情并不是个好主意。

改编协议中违背的另一个原则是,Bob 以响应 $K_{A,B}(R_c)$ 的形式泄露了有价值的信息,而没有确切了解他的信息发送给何人。这一原则在原始协议中没有违背,其中 Alice 首先需要证实其身份,然后 Bob 希望传送她的加密信息。

存在一些加密协议开发人员这些年逐渐开始学习的其他原则,我们将在下面讨论其他协议的时候介绍其中的一些原则。一个重要的教训是,将安全协议设计成能完成要其完成的任务通常比看起来困难很多。同样,正如我们上面所论述的那样,修改一个现有的协议以提高其性能可以轻易地影响其正确性。有关协议的设计原则的更多信息可以在(Abadi,Needham 1996)中找到。

2. 使用密钥分发中心的身份验证

使用共享密钥进行身份验证的一个问题是可扩展性。如果一个分布式系统包含 N

台主机，并且需要每台主机与其他 $N-1$ 台主机共享一个密钥，那么该系统作为一个整体就需要管理 $N(N-1)/2$ 个密钥，并且每台主机必须管理 $N-1$ 个密钥。对较大的 N 来说，这将导致出现问题。一个替代办法是依赖一个 KDC(key distribution center，密钥分发中心)使用集中式方法。这一 KDC 与每台主机共享一个密钥，但不需要主机对拥有共享密钥。换句话说，使用 KDC 需要我们管理 N 个密钥而不是 $N(N-1)/2$ 个密钥，这无疑是一个改进。

如果 Alice 希望与 Bob 建立一条安全通道，那么她可以通过一个(可靠的)KDC 的帮助来实现。整体思想是 KDC 向 Alice 和 Bob 两人分发一个密钥，他们可以使用该密钥进行通信，如图 8.15 所示。

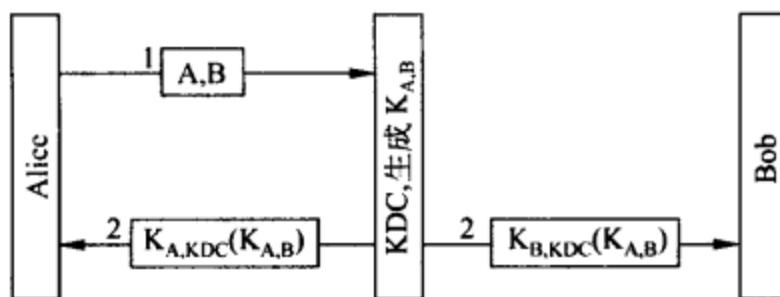


图 8.15 使用 KDC 的原理

Alice 首先向 KDC 发送一条消息，通知 KDC 她希望与 Bob 交谈。KDC 返回一个包含她能够使用的共享密钥 $K_{A,B}$ 的消息。该消息用 Alice 与 KDC 共享的密钥 $K_{A,KDC}$ 进行了加密。此外，KDC 也向 Bob 发送 $K_{A,B}$ ，但此时使用与 Bob 共享的密钥 $K_{B,KDC}$ 进行加密。

这种方法的主要缺点是 Alice 可能希望在 Bob 接收到来自 KDC 的共享密钥之前就开始与 Bob 建立一个安全通道。此外，需要 KDC 向 Bob 传送密钥来使 Bob 进入循环。如果 KDC 恰好将 $K_{B,KDC}(K_{A,B})$ 传回给 Alice，并令其关心与 Bob 的连接，就可以避开这些问题。这就导致了如图 8.16 中所示的协议。消息 $K_{B,KDC}(K_{A,B})$ 也称为票据(ticket)。将此票据传给 Bob 是 Alice 的工作。注意 Bob 仍是惟一的一个能够有意识地使用该票据的人，因为他是除 KDC 外惟一知道该票据中所包含消息的解密方法的人。

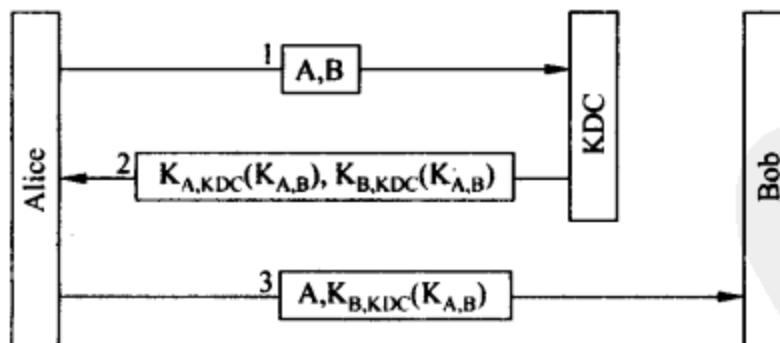


图 8.16 使用票据并让 Alice 建立一个与 Bob 的通道

图 8.16 中所示的协议是使用 KDC 的一个身份验证协议的众所周知的实例变体，该协议名为 Needham-Schroeder 身份验证协议，按其发明者(Needham, Schroeder 1978)名字命名。该协议的一个不同变体应用于我们稍后要描述的 Kerberos 系统中。Needham-Schroeder 协议，如图 8.17 所示，是一个多向质询—响应协议，其工作方式如下所述。

当 Alice 希望与 Bob 建立一个安全通道时,她就向 KDC 发送一个包含质询 R_A , 以及其身份标识 A,当然还有 Bob 的身份标识。KDC 通过给予其票据 $K_{B,KDC}(K_{A,B})$ 以及她可以随后与 Bob 共享的密钥 $K_{A,B}$ 进行响应。

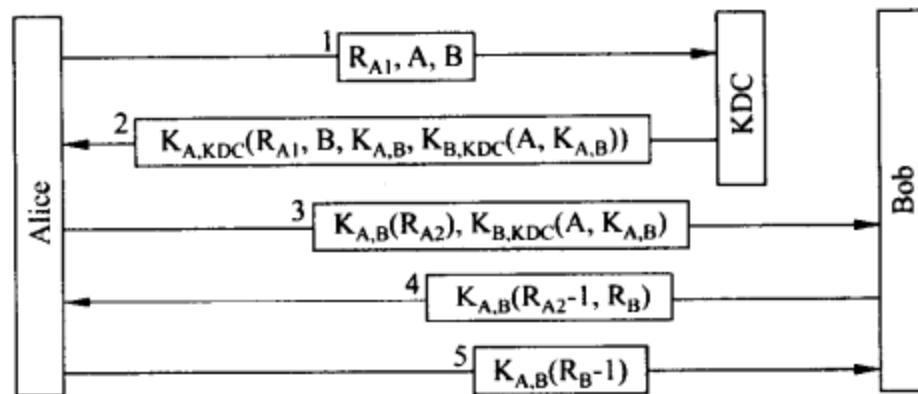


图 8.17 Needham-Schroeder 身份验证协议

Alice 发送给 KDC 的质询 R_{A1} 连同其与 Bob 建立通道的请求还被称为“现时”。现时 (nonce) 是一个仅使用一次的随机数, 比如从一个非常大的随机数集合中选取一个。现时的主要用途是将两条消息惟一地互相联系起来, 在本例中是消息 1 和消息 2。特别是, 通过在消息 2 中再次加入 R_{A1} , Alice 将确切知道消息 2 是作为消息 1 的响应发送的, 而不是较早的消息的重放。

为了理解即将发生的问题, 假设我们不使用现时, 并且 Chuck 已经窃取了 Bob 的一个原先使用的密钥, 即 $k_{B,KDC}^{\text{old}}$ 。此外, Chuck 已经窃听到了一个旧的响应 $K_{A,KDC}(B, K_{A,B}, K_{B,KDC}^{\text{old}}(A, K_{A,B}))$, 该响应是 KDC 对来自 Alice 的与 Bob 交谈的早先请求返回的响应。这时, Bob 希望已经与 KDC 协商了一个新的共享密钥。然而, Chuck 耐心地等待, 直到 Alice 再一次请求建立与 Bob 的安全通道。那时, 他就重放旧的响应, 哄骗 Alice 相信她正与 Bob 交谈, 因为这样他就可以解密该票据并证实其知道共享密钥 $K_{A,B}$ 。

通过加入一个现时, 这样的攻击就不可能成功, 因为重放较早的消息将立即被发现。特别是, 响应消息中的现时不会与原始请求中的现时相匹配。

消息 2 也包含 Bob 的身份标识 B。通过加入 B, KDC 保护 Alice 免受下列攻击。假设消息 2 中遗漏了 B。在这种情况下, Chuck 可以通过用其自身的身份 C 取代 Bob 的身份来修改消息 1。然后 KDC 会以为 Alice 希望建立到 Chuck 的安全通道, 并因此进行响应。一旦 Alice 希望与 Bob 联系, Chuck 就窃听该消息, 并哄骗 Alice 相信她正与 Bob 交谈。通过从消息 1 到消息 2 复制另一方的身份, Alice 就会立即检测到她的请求已经被修改。

KDC 将该票据传给 Alice 后, Alice 和 Bob 之间的安全通道就可以建立起来。Alice 以发送消息 3 开始, 其中包含发给 Bob 的票据, 以及一个使用 KDC 刚产生的共享密钥 $K_{A,B}$ 加密的质询 R_{A2} 。然后 Bob 解密该票据以寻找该共享密钥, 并返回一个响应 $R_{A2}-1$ 以及对 Alice 的质询 R_B 。

下面有关消息 4 的评论是适宜的。一般来说, 通过返回 $R_{A2}-1$ 而不只是 R_{A2} , Bob 不仅证实了他知道该共享密钥, 还证实了他确实解密了该质询。此外, 这将消息 4 与消息

3 联系在一起,与现时 R_A 将消息 2 和消息 1 联系在一起的方式相同。因此该协议更有效地防止重放。

然而,在这一特殊情况下,由于此消息之前还没有在该协议中的任何地方使用过这一简单原因,所以返回 $K_{A,B}(R_{A2}, R_B)$ 已经足够了。 $K_{A,B}(R_{A2}, R_B)$ 已经证实了 Bob 能够解密消息 3 中发送的质询。图 8.17 中所示的消息 4 是由历史原因引起的。

此处介绍的 Needham-Schroeder 协议仍存在弱点,如果 Chuck 曾经获得过一个旧密钥 $K_{A,B}$ 的控制,那么他就可以重放消息 3,并使 Bob 建立一个通道。然后 Bob 将相信他正与 Alice 会话,虽然实际上另一端是 Chuck。在这种情况下,我们需要将消息 3 与消息 1 联系起来,也就是说,使该密钥依赖于来自 Alice 要与 Bob 建立通道的初始请求。该解决方案如图 8.18 所示。

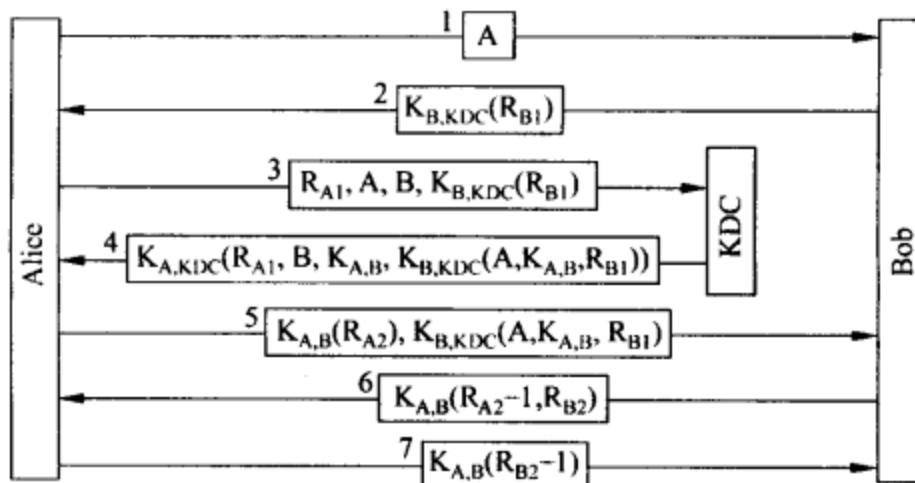


图 8.18 Needham-Schroeder 协议中对先前生成的会话密钥恶意的重新使用的防护

其中的窍门是向 Alice 发送给 KDC 的请求中加入一个现时。然而,该现时必须来自 Bob,这就使 Bob 确信希望与其建立安全通道的人将从 KDC 获得适当信息。因此,Alice 首先请求 Bob 向其发送一个使用 Bob 与 KDC 之间共享的密钥加密的现时 R_{B1} 。Alice 将此现时加入到其发往 KDC 的请求中,随后 KDC 解密该现时并将结果放在生成的票据中。这样,Bob 就会确切知道该会话密钥同 Alice 要求与 Bob 交谈的原始请求联系在一起。

3. 使用公钥加密的身份验证

现在我们来讨论使用不需要 KDC 的公钥加密系统的身份验证。我们再一次考虑 Alice 希望建立到 Bob 的安全通道,并且两人都拥有彼此的公钥的情况。基于公钥加密的一个典型身份验证协议如图 8.19 所示,我们接下来要进行解释。

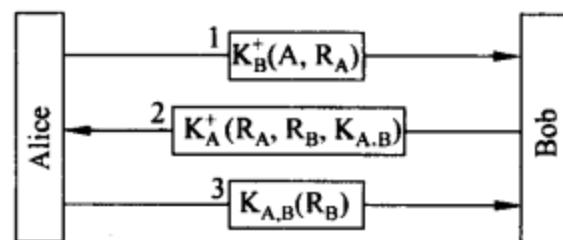


图 8.19 公钥加密系统中的相互身份验证

Alice 以向 Bob 发送一个使用他的公钥 K_B^+ 加密的质询 R_A 开始。解密该消息并向 Alice 返回该质询是 Bob 的工作。因为 Bob 是惟一可以解密该消息(使用与 Alice 所用的公钥相关联的私钥)的人,所以 Alice 就会知道她正与 Bob 交谈。注意重要的是要保证 Alice 使用的是 Bob 的公钥,而不是某个模仿 Bob 的人的公钥。这种保证的给予方式稍后在本章讨论。

Bob 接收到 Alice 建立通道的请求时,他返回解密过的质询,以及他自身用于对 Alice 进行身份验证的质询 R_B 。此外,他还产生一个可以用于更进一步通信的会话密钥 $K_{A,B}$ 。Bob 对 Alice 的质询的响应、他自己的质询以及该会话密钥都放在一个使用属于 Alice 的公钥 K_A^+ 加密的消息中,如图 8.19 中消息 2 所示。只有 Alice 能够使用与 K_A^+ 相关联的私钥 K_A^- 解密此消息。

最后,Alice 返回其对 Bob 的质询的响应,该响应使用 Bob 产生的会话密钥 $K_{A,B}$ 。在那种方式下,她就已经证实她能够解密消息 2,因此 Bob 确实是在与 Alice 交谈。

8.2.2 消息完整性和机密性

除身份验证外,安全通道还应该提供对消息完整性和机密性的保证。消息完整性是指保护消息免受修改;机密性确保了窃听者不能截获和读取消息。机密性可通过在发送消息前简单加密该消息容易地实现。加密通过与接收者共享密钥来实现,或者通过使用接收者的公钥来实现。然而,保护消息免受修改更复杂一些,就像我们接下来要讨论的那样。

1. 数字签名

消息完整性通常超出了安全通道中的实际传输的范畴。我们来考虑这样一种情况,即 Bob 刚刚以 \$500 把一个收藏家的一些电唱机唱片卖给 Alice。整个交易是通过电子邮件完成的。最后,Alice 向 Bob 发送了一个消息,确认她将用 \$500 购买那些唱片。除身份验证外,至少需要考虑两个关于消息完整性的问题。

(1) Bob 需要向 Alice 保证他不会恶意地把她的消息中所提的 \$500 改为更高的价格,并声称她答应的价格多于 \$500;

(2) Alice 需要向 Bob 保证,她不会因为她重新考虑就否认发送过该消息。

如果 Alice 的签名与该消息的内容惟一地联系在一起,那么以这种方式对该消息进行数字签名就可以处理这两个问题。消息和其签名之间的惟一关联防止了对该消息进行了修改而没有引起注意的情况出现。此外,如果可以检验 Alice 的签名是真的,那么她以后就不能否认对该消息进行签名这一事实。

放置数字签名有若干种方式。一种流行的形式是使用例如 RSA 的公钥加密系统,如图 8.20 所示。Alice 向 Bob 发送消息 m 时,她使用其私钥 K_A^- 加密该消息,并向 Bob 发出。如果她还希望保密该消息内容,那么她可以使用 Bob 的公钥并发送 $K_B^-(m, K_A^-(m))$,其中结合了 m 和 Alice 签名的版本。

该消息到达 Bob 时,他可以使用 Alice 的公钥对其进行解密。如果 he 能够得到保证该公钥确实属于 Alice,那么解密 m 的签名版本以及成功地与 m 进行比较只能意味着该

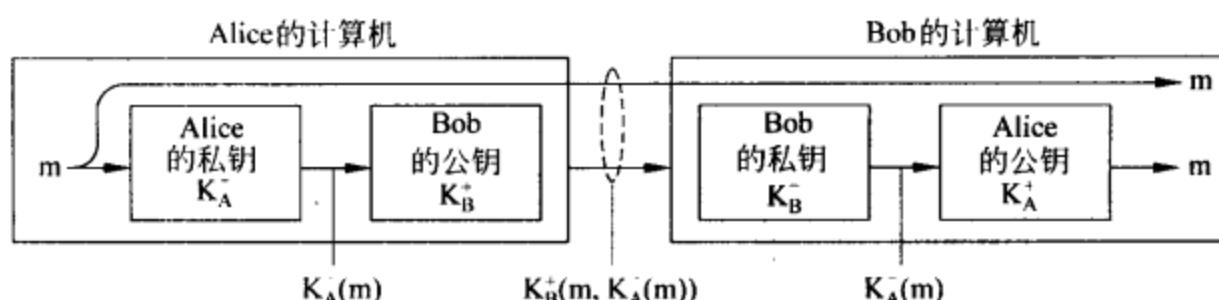


图 8.20 使用公钥加密对消息进行数字签名

消息来自 Alice。Alice 受到防止 Bob 对 m 进行任何恶意修改的保护, 因为 Bob 必须一直证明 m 的修改版本也是由 Alice 签名的。换句话说, 经解密的消息本身在本质上从来不能作为证据。保持 m 的签名版本以保护其自身防止 Alice 的否认也是出于 Bob 自己的利益。

虽然此方法中的协议是正确的, 但这一方法仍有很多问题。首先, Alice 的签名的有效性仅在 Alice 的私钥保密的情况下保持。如果 Alice 希望在向 Bob 发送了她的确认后撤销该交易, 那么她可以声称她的私钥在该消息发送前被盗了。

Alice 决定改变其私钥时会出现另一个问题。这么做本身可能不是一个坏主意, 因为不时改变密钥一般有助于防止入侵。然而, Alice 改变其密钥后, 她向 Bob 发送的信息就是无用的了。这种情况下, 除了在消息签名时使用时间戳外, 还需要跟踪密钥改变时间的一个中央授权机构。

此方法的另一个问题是 Alice 使用其私钥加密整个消息。这样的加密在处理需求上可能开销很大(或者如果我们假设转换为二进制数的消息由一个预定义的最大值限制, 那么这种加密在数学上甚至是不能实现的), 而且实际上也是不必要的。想想我们仅需要惟一地将签名与特定消息相关联。一个更容易获得且可以证明是更精致的方法是使用消息摘要。

正如我们所解释的, 消息摘要是一个固定长度的位串 h , 该位串是通过加密散列函数 H 从任意长度的消息 m 计算出来的。如果 m 变为 m' , 那么其散列 $H(m')$ 就会与 $h = H(m)$ 不同, 这就能够容易地检测出发生了修改。

为了对消息进行数字签名, Alice 可以首先计算一个消息摘要, 随后使用其私钥加密该摘要, 如图 8.21 所示。经加密的摘要与发给 Bob 的消息一同发送出去。注意, 该消息

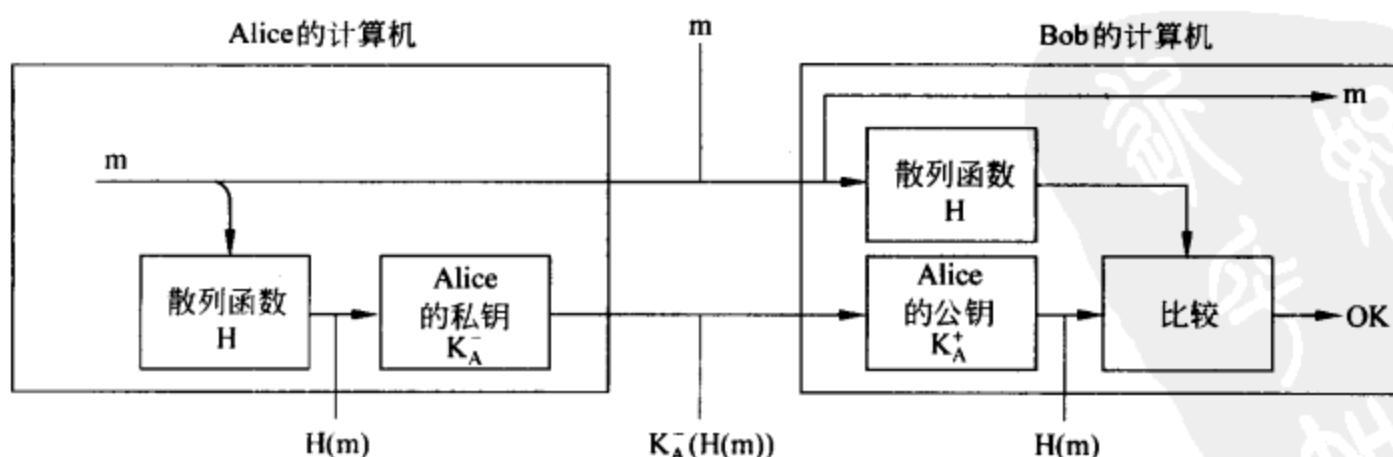


图 8.21 使用消息摘要对消息进行数字签名

本身作为明文发送：每个人都可以读取该消息。如果需要机密性，那么还应该使用 Bob 的公钥加密该消息。

Bob 接收到该消息及其经加密的摘要时，他仅需要使用 Alice 的公钥解密该摘要，并单独计算消息摘要。如果从接收到的消息计算出的摘要与经解密的摘要相匹配，那么 Bob 就确信该消息是由 Alice 签名的。

2. 会话密钥

在安全通道的建立期间，在身份验证阶段完成以后，通信各方一般使用惟一的共享会话密钥以实现机密性。而在不再使用该通道时安全地丢弃该会话密钥。还有一种替代办法是为实现机密性使用与建立该安全通道所用密钥相同的密钥。然而，使用会话密钥具有很多重要的好处(Kaufman 等 1995)。

首先，经常使用一个密钥时，就很容易泄露该密钥。在某种意义上，加密密钥就像普通钥匙一样常遭受“磨损”。使用会话密钥的基本思想是如果入侵者能够窃听到许多使用同一密钥加密的数据，就有可能发动攻击以得到所使用的密钥的某些特征，并有可能发现明文或该密钥本身。出于这种原因，尽可能不使用身份验证密钥则更为安全。此外，这样的密钥通常使用一些相对耗时的带外机制进行交换，例如定期的邮件或电话。以这种方式交换密钥应该保持最小量。

为每个安全通道产生一个单独密钥的另一个重要原因是要确保通信各方受到防止重放攻击的保护，正如我们前面碰到过很多次的那样。通过在每次建立安全通道时使用一个惟一的会话密钥，通信各方至少受到防止重放一个完整会话的保护。要保护重放上一个会话中的个别消息，一般需要其他措施，例如加入时间戳或序列号作为消息内容的一部分。

假设消息完整性和机密性是通过使用与建立会话所使用密钥相同的密钥来实现的。在这种情况下，只要该密钥被损坏了，入侵者就可能解密先前对话中传输的消息，这无疑是我们不希望看到的情况。与之相比，每个会话使用一个密钥要安全得多，因为如果这样的一个密钥被损坏了，最坏情况下，只有一个会话受到影响，其他会话期间发送的消息仍可保持机密性。

与这最后一点相关的是 Alice 可能希望与 Bob 交换一些机密数据，但她并不足够信任他，所以她不会以用长期密钥加密的数据的形式向其发送消息。她可能希望为高度机密的消息保留这样的密钥，她与真正信任的人才会交换这些消息。在这样的情况下，使用相对容易得到的会话密钥与 Bob 交谈就已足够。

一般来说，身份验证密钥通常是以这样一种方式建立的，即替换该密钥是要付出相当大的代价。因此，这样的长期密钥与更廉价且临时性更强的会话密钥的结合对实现安全通道的数据交换通常是一个很好的选择。

8.2.3 安全组通信

到目前为止，我们已经集中讨论了在通信双方之间建立安全通信通道的问题。然而，在分布式系统中，通常有必要启用多于两方之间的安全通信。复制的服务器是一个典型

的例子,其中所有副本之间的通信都应受到保护,以防修改、伪造和窃听,就像在两方安全通道的情况下一样。在本节中,我们来更彻底地讨论安全组通信。

1. 机密组通信

首先,考虑保护 N 个用户的组之间的通信免受窃听的问题。要确保机密性,一个简单的方法是令所有的组成员共享同一密钥,该密钥用于对组成员之间发送的所有消息进行加密和解密。因为此方法中的密钥是所有成员共享的,所以所有成员都应是可靠的,能够确实对该密钥进行保密,这一点很必要。这一先决条件本身就说明,对于机密组通信使用单一共享密钥比两方安全通道更易受到攻击。

一种替代的解决方法是在每对组成员之间使用各自的共享密钥。一旦一个成员被证明在泄露信息,其他成员就会简单地停止向该成员发送消息,但仍然使用他们互相通信所使用的密钥。然而,这样做不是必须保持一个密钥,现在必要的是保持 $N(N-1)/2$ 个密钥,这本身就可能是一个困难的问题。

使用公钥加密系统可以解决这个问题。在这种情况下,每个成员有其自己的(公钥,私钥)对,其中所有成员都可用该公钥发送机密消息。在这种情况下,总共需要 N 个密钥对。如果一个成员不再值得信赖,那么仅将其清除出组,而不会影响其他密钥的使用。

2. 安全的复制服务器

现在考虑一个完全不同的问题:一个客户向一组复制服务器发出了一个请求。这些服务器可能已经由于容错或性能的原因进行了复制,但无论如何,该客户都期望响应是可信的。换句话说,不管这组服务器是否像我们在上一章中所讨论的那样经常遭受拜占庭故障,客户都希望返回的响应没有受到安全攻击。如果入侵者已经成功地破坏了一台或多台服务器,那么就可能发生这样的攻击。

保护客户免受这样的攻击的一种解决方法是收集来自所有服务器的响应并对每一个服务器进行身份验证。如果响应中的大多数来自未被破坏的(也就是,通过身份验证的)服务器,那么该客户就可以相信该响应也是正确的。不幸的是,这种方法暴露了服务器的复制,从而违反了复制的透明度。

Reiter(1994)提出了安全的复制服务器的方法,其中保持了复制透明度。此方法的优点是,因为客户不知道实际的副本,所以以安全方式添加或删除副本要容易很多。下面讨论密钥管理时我们将回到管理安全组这一话题。

安全和透明的复制服务器的本质在于称为秘密共享(secret sharing)的特性。多个用户(或进程)共享一个秘密时,没有人知道整个秘密。相反,该秘密只有在所有人聚集在一起时才能揭开。这样的方法可能非常有用。例如,考虑发射一枚核弹的情形。一般这种行动需要至少两个人的授权。每个人持有一个私钥,该私钥应该与另一个人的私钥结合起来使用才可以实际发射导弹。仅使用一个密钥则不行。

在使用安全的复制服务器的情况下,我们所寻找的是对这样问题的解决方法,即 N 台服务器中最多有 k 台可以产生不正确的答案,且在这 k 台服务器中,最多有 $c \leq k$ 台实际上已经被入侵者破坏。注意,这个要求使该服务本身是 k 容错的,如上一章所述。区别

在于我们现在将被恶意破坏的服务器归为有故障的一类。

现在考虑服务器被主动复制的情况。换句话说，一个客户将请求同时发送到所有的服务器，随后由每台服务器处理。每台服务器都产生一个返回该客户的响应。对安全复制的服务器组来说，我们要求每台服务器的响应都要附有数字签名。如果 r_i 是来自服务器 S_i 的响应，就令 $md(r_i)$ 表示服务器 S_i 所计算的消息摘要。该摘要使用服务器 S_i 的私钥 K_i^- 签名。

假设我们希望保护客户，防止有多于 c 台的服务器被破坏。换句话说，该服务器组应该能够忍受至多 c 台服务器被破坏，而且还能产生客户可以给予信任的响应。如果单个服务器的签名能够以这样一种方式结合，即至少需要将 $c+1$ 个签名联合在一起为该响应构造一个有效的签名，那么这样就可以解决我们的问题。换句话说，我们希望让复制服务器产生一个保密的有效签名，这样就使在没有至少一台可靠的服务器的帮助下， c 台被破坏的服务器不足以产生有效签名。

作为一个例子，请考虑一个 5 台复制服务器的组，该组应该能够忍受 2 台被破坏的服务器，并且还产生客户可以信任的响应。每台服务器 S_i 向该客户发送其响应 r_i ，以及其签名 $sig(S_i, r_i) = K_i^-(md(r_i))$ 。因此，该客户最后会收到 5 个三元组 $\langle r_i, md(r_i), sig(S_i, r_i) \rangle$ ，从这 5 个三元组中应该得到正确的响应。这一情况如图 8.22 所示。

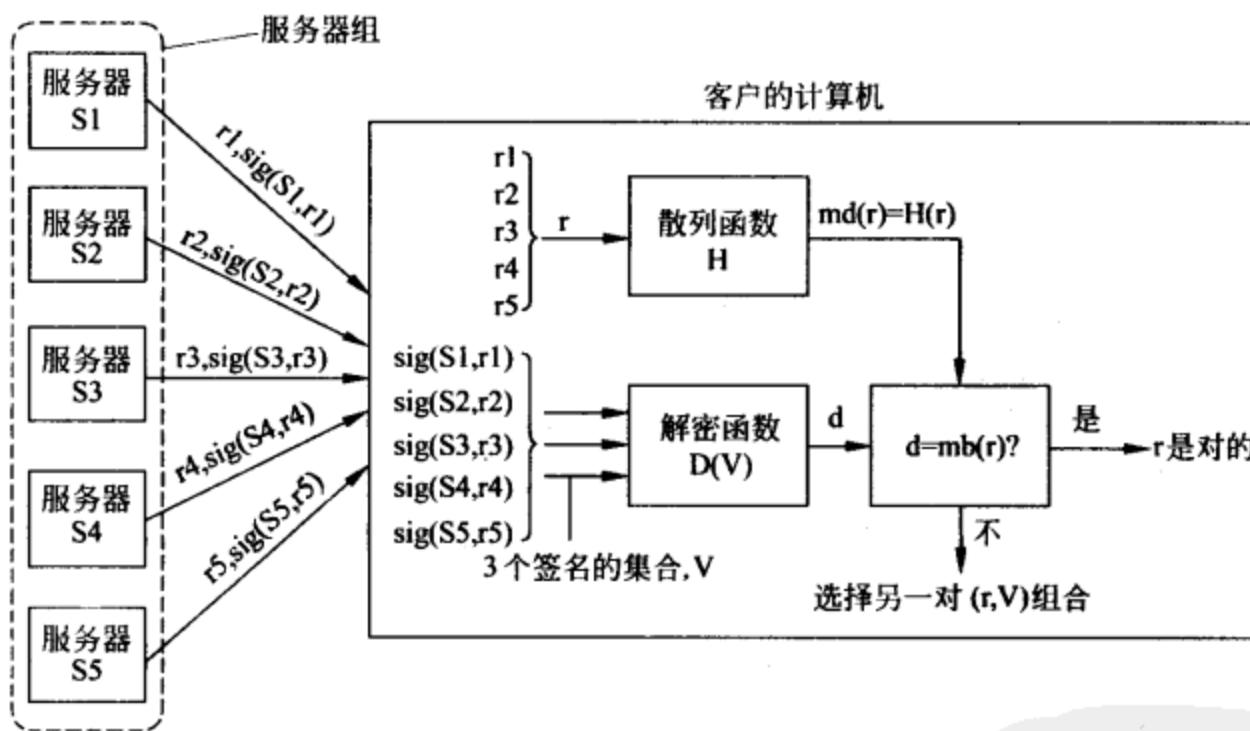


图 8.22 在一个复制服务器组中共享一个保密签名

每个摘要 $md(r_i)$ 也是由客户计算出来的。如果 r_i 不正确，那么通常可以通过计算 $K_i^+(K_i^-(md(r_i)))$ 检测出来。然而，此方法不再能够应用，因为没有单个服务器值得信赖。相反，客户使用一个特殊的众所周知的解密函数 D ，该函数以一个 3 个签名的集合 $V = \{sig(S, r), sig(S', r'), sig(S'', r'')\}$ 作为输入，并制造单个摘要作为输出：

$$d_{out} = D(V) = D(sig(S, r), sig(S', r'), sig(S'', r''))$$

有关 D 的细节，请参阅(Reiter, 1994)。3 个签名具有 $5! / (3! 2!) = 10$ 种可能的组合客户可以用它们作 D 的输入。如果其中一种组合为某个响应 r_i 产生了正确的摘要 $md(r_i)$ ，

那么客户就可以认为 r_i 是正确的。特别是,可以相信该响应是由至少 3 台可靠的服务器产生的。

为了提高复制的透明度,Reiter 和 Birman 让每台服务器 S_i 向其他服务器广播一条包含其响应 r_i 以及其相关签名 $\text{sig}(S_i, r_i)$ 的消息。一台服务器接收到至少 $c+1$ 个这样的消息(包括其自己的消息)时,该服务器就试图为其中一个响应计算一个有效的签名。如果这对响应 r 和 $c+1$ 个签名的集合 V 成功了,该服务器就以单个消息的形式向客户发送 r 和 V 。随后该客户 r 的正确性可以通过检查签名来检验,也就是说,检查 $\text{md}(r)=D(V)$ 是否成立。

我们刚刚所描述的内容还被称为 (m, n) —阈值方法 ((m, n) -threshold scheme),在我们的例子中, $m=c+1$ 且 $n=N$, N 为服务器的数量。在一个 (m, n) —阈值方法中,一条消息划分为 n 块,称为影子(shadows),因为任何 m 个影子(shadows)都可以用于重组原始消息,但使用 $m-1$ 个或更少的影子就不能重组消息。构造 (m, n) —阈值方法有多种方式。细节可在文献(Schneier 1996)中找到。

8.3 访问控制

目前为止在我们所使用的客户-服务器模型中,客户和服务器建立了一条安全通道后,该客户就可以发出要服务器执行的请求。请求包括执行对服务器控制的资源的操作。一种普通情况是一个对象服务器控制很多对象。客户发送的请求一般都包含对特定对象方法的调用。这样的请求只有在客户对该调用具有足够访问权限(access right)时才可以执行。

正式地说,检验访问权限称为访问控制(access control),而授权(authorization)则是关于授予访问权限的术语。这两个术语互相之间紧密联系,并通常以可互换的方式使用。实现访问控制有很多方式。首先我们讨论一些一般问题,把注意力集中于处理访问控制的不同模型。实际控制对资源的访问的一种重要方式是建立一个防火墙来保护应用程序甚至整个网络。防火墙将被单独讨论。随着代码移动性的出现,访问控制不再能够仅通过使用传统方法来实现,而必须设计新的技术,这一内容也在本节中讨论。

8.3.1 访问控制中的一般问题

为了理解访问控制中所包含的各种问题,通常采用图 8.23 中所示的简单模型。该模型由发出访问对象(object)的请求的主体(subject)组成。这里的对象与我们迄今为止所讨论的对象非常相像,可以认为是将自己的状态封装起来并执行对状态的操作。主体可以请求执行的对象操作可以通过接口来使用。最好认为主体是代表用户的进程,但主体也可以是为执行其工作而需要其他对象服务的对象。

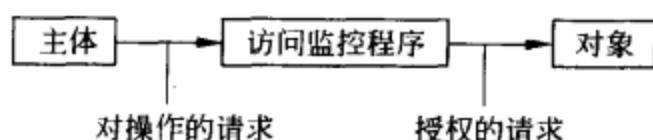


图 8.23 对象访问控制的一般模型

对对象的访问控制与对象保护有关，防止对象被那些不允许其执行特定（或者甚至任何）方法的主体所调用。同样，保护可以包括对象管理问题，例如创建、重命名或删除对象。保护通常由一个名为访问监控器的程序执行。访问监控器记录各个主体可以做的操作，并决定是否允许一个主体执行特定操作。每次调用对象时都调用此监控程序（举例来说，由底层的可信操作系统调用）。因此，该访问监控器本身是可防损害的，这一点非常重要：攻击者一定不能破坏该程序。

1. 访问控制矩阵

建立主体对对象的访问权限的模型，其中的一个普通方法是构造一个访问控制矩阵。每个主体由该矩阵中的一行表示，每个对象由一列表示。如果该矩阵由 M 表示，那么一项 $M[s, o]$ 就准确地列出了主体 s 可以请求执行的对对象 o 的操作。换句话说，只要主体 s 请求调用对象 o 的方法 m ，该访问监控程序就应该检查 m 在 $M[s, o]$ 中是否列出。如果 m 没有在 $M[s, o]$ 中列出，那么该调用失败。

考虑到一个系统可能需要支持需要保护的上千个用户和上百万个对象，所以不能以真正的矩阵实现访问控制矩阵。矩阵中的许多项将是空的，因为单个主体一般可以访问相对很少的对象。因此，应采用其他更有效的方法来实现访问控制矩阵。

一种广泛应用的方法是让每个对象保存一个希望访问该对象的主体的访问权限表。实质上，这意味着该矩阵以列方向覆盖所有对象分布，不考虑空项。这种类型的实现导致了称为 ACL（access control list，访问控制列表）。假设每个对象具有其自身的关联 ACL。

另一种方法是以行方向分布该矩阵，通过为每个主体赋予一个其对每个对象所拥有的权能（capability）列表来实现。换句话说，一个权能对应访问控制矩阵中的一项。不具备对指定对象的权能意味着没有对该对象的访问权限。

权能可以比喻为票据，赋予其持有者与该票据相关联的特定权限。应该明确的是票据应该受到保护，防止其持有者对其进行修改。一种特别适合分布式系统且广泛应用于 Amoeba（Tanenbaum 等 1990）的方法是使用签名保护（一列）权能。我们稍后在讨论安全管理时再讨论这些内容和其他问题。

使用 ACL 和使用权能这两种保护对一个对象的访问的方式，其区别如图 8.24 所示。在使用 ACL 的情况下，客户向服务器发送一个请求时，服务器的访问监控程序会检查该服务器是否了解该客户以及是否允许该客户执行所请求的操作，如图 8.24(a) 所示。

然而，使用权能时，客户仅向服务器发送其请求。该服务器对自己是否了解该客户不感兴趣，权能已经足够了。因此，该服务器仅需要检查该权能是否有效，以及所请求的操作是否在权能表中列出。这种通过权能保护对象的方法如图 8.24(b) 所示。

2. 保护域

ACL 和权能通过忽略所有的空项来帮助有效地实现访问控制矩阵。尽管如此，如果不采取进一步措施，ACL 或权能列表仍可能变得非常大。

缩小 ACL 的一种普通方式是利用保护域。确切地说，保护域是一组（对象，访问权

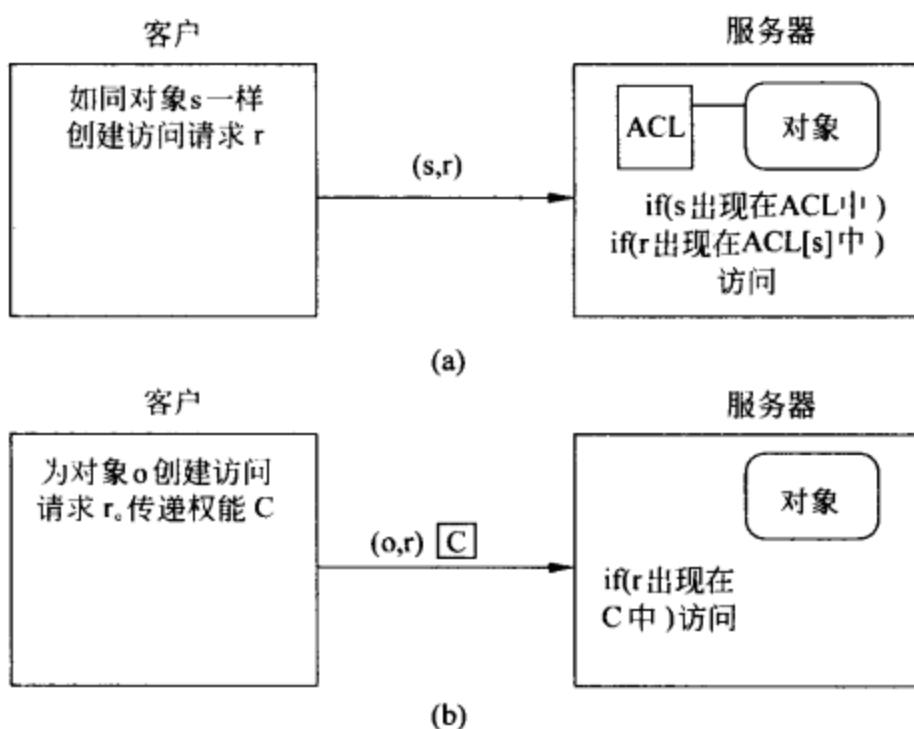


图 8.24 为保护对象而使用的 ACL 和权能之间的比较

(a) 使用 ACL; (b) 使用权能

限)对。每对为给定的对象确切指定允许执行的操作(Saltzer, Schroeder 1975)。执行一个操作的请求总是在一个域内发出。因此,只要一个主体请求对一个对象执行一个操作,访问监控程序首先都会查找与该请求相关联的保护域。然后,如果找到这个域,那么随后就可以检查是否允许执行该请求。对保护域的使用有几种不同的方法。

其中一种方法是构造用户组。作为例子,我们来考虑一个公司内部 intranet 上的一个 Web 页面的情况。这样的一个页面应该对每个雇员都是可用的,但对其他任何人都是不可用的。可能决定令一个单独组 Employee 包含所有当前的雇员,而不是为每个可能的雇员向 ACL 中为该 Web 页面添加一项。无论用户何时访问该 Web 页面,访问监控程序都仅需要检查该用户是否为一个雇员。属于组 Employee 的用户在一个单独的表中记录(当然,该表受到保护,防止未经授权的访问)。

通过引入分层组可以使事情变得更加灵活。例如,如果一个公司拥有三个不同的分公司,分别位于 Amsterdam、纽约和旧金山,那么该公司可能希望将其 Employee 组细分为子组——每个城市一个子组,结果为如图 8.25 所示的一个组织结构。

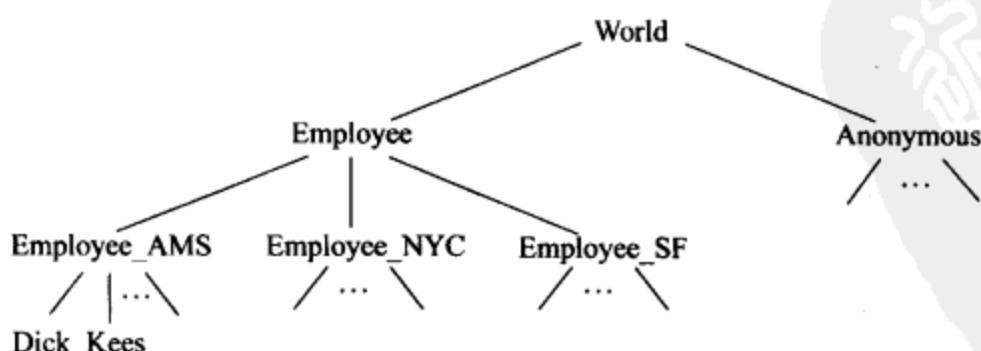


图 8.25 保护网域的用户组分层组织

应该允许所有雇员访问公司 intranet 的 Web 页面。然而,举例来说,应该只允许 Amsterdam 的雇员子集对与 Amsterdam 分公司相关联的 Web 页面进行修改。如果来自 Amsterdam 的用户 Dick 希望从 intranet 读取一个 Web 页面,那么访问监控程序首先就需要查找共同组成集合 Employee 的子集 Employee_AMS、Employee_NYC 和 Employee_SF。然后该程序必须检查 Disk 是否包含在这些集合中。分层组的优点是管理组成员相对简单,而且可以有效地构造非常大的组。一个明显的缺点是如果成员数据库是分布式的,那么查找成员代价可能相当高。

一种替代办法取代了让访问监控程序完成所有工作的方式,该办法是让每个主体携带一个证书,列出其所属的群组。这样,只要 Dick 希望从公司的 intranet 读取 Web 页面,他就将其证书传送给访问监控程序,声明他是 Employee_AMS 的成员。要保证证书是真实的而且没有被篡改,就应该通过一些手段(例如数字签名)来保护证书。证书看起来与权限差不多。我们稍后再讨论这些问题。

与让组作为保护域相关,还有可能以角色来实现保护域。在基于角色的访问控制中,用户始终使用指定的角色登录到系统中,该角色通常与该用户在公司中所具有的职能相关联(Sandhu 1996)。一个用户可以具有多个职能。例如,Dick 可以同时作为部门主管、项目经理和职员调查委员会的成员。根据他登录时所采用的角色,可以分配给他不同的特权。换句话说,他的角色决定了他所在的保护域(也就是组)。

当为用户分配角色并要求用户在登录时采用指定的角色时,用户还应该可以在必要时改变其角色。例如,可能需要允许作为部门主管的 Dick 偶尔改变为项目经理。注意,这样的改变在仅以组来实现保护域时是难以表示的。

除使用保护域外,还可以通过基于对象所提供的操作将对象(分层地)分组来进一步提高效率。例如,根据对象提供的接口对对象进行分组,而不是考虑独立的对象,可能使用子类型(也称为接口继承,参见 Gamma 等 1994)来实现分层。在这种情况下,一个主体请求对一个对象执行操作时,访问监控程序查找该对象的这个操作所属的接口。然后该程序检查是否允许该主体调用属于该接口的操作,而不是检查该主体是否能调用指定对象的操作。

将保护域和对象分组结合起来也是可能的。使用这两种技术,以及指定的数据结构和有限的对象操作,Gladney(1997)描述了如何为在数字图书馆中使用的非常大的对象集合实现 ACL 方法。

8.3.2 防火墙

到目前为止,我们已经说明了使用加密技术,结合访问控制矩阵的一些实现建立保护的方法。只要所有的通信方根据相同的规则集工作,这些方法就能发挥作用。当开发与世界其余部分隔离的独立分布式系统时,这样的规则可能还要进一步强化。然而,允许外来人访问分布式系统控制的资源时事情就变得更为复杂。这种访问的例子包括发送邮件、下载文件、上载传真表格等。

为了在这些环境下保护资源,就需要一种不同的方法。实际上,所发生的对分布式系统任何部分的外部访问都由一种称为防火墙(Cheswick, Bellovin 2000, Zwicky 等 2000)

的特殊类型的访问监控程序控制。本质上,防火墙将分布式系统的任意部分与外界分离开,如图 8.26 所示。所有出入站数据包,尤其是所有入站数据包都通过一台特殊计算机传送,并且在传送前受到检查。未经授权的通信数据将被丢弃,不允许继续传输。一个重要问题是防火墙本身应该受到很好的保护,以防受到任何类型的安全威胁,它决不应该出故障。

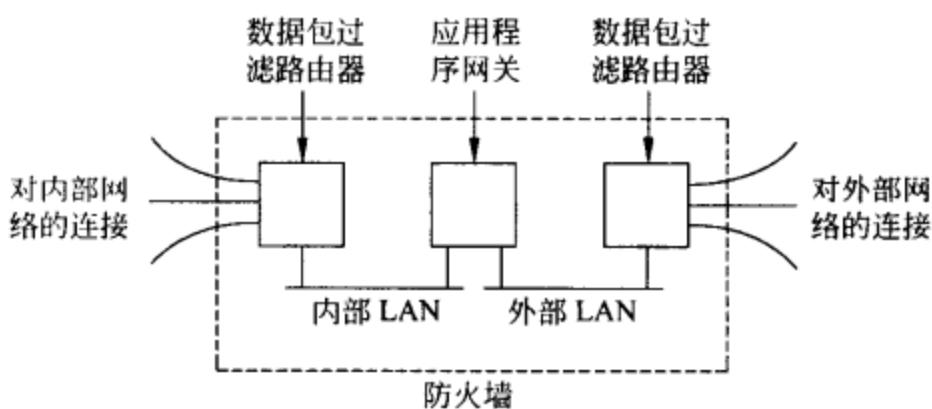


图 8.26 防火墙的一般实现方法

防火墙在本质上分为两种不同的类型,通常结合在一起使用。防火墙的一个重要类型是数据包过滤网关(packet-filtering gateway)。这种类型的防火墙作为路由器工作,并基于数据包报头中包含的源地址和目的地址制定关于是否传送一个网络数据包的决定。通常,图 8.26 中所示的外部 LAN 上的数据包过滤网关会过滤入站数据包,而内部 LAN 中的网关则会过滤出站数据包。

例如,为了保护一个内部 Web 服务器,防止来自不属于内部网络的主机的请求进入,数据包过滤网关可以决定丢掉所有发往该 Web 服务器的入站数据包。

更微妙的是这种情况,例如,公司的网络由多个局域网组成,通过我们前面所讨论的 SMDS 网络相连接。每个 LAN 都可通过一个数据包过滤网关进行保护,该网关配置为仅让源于一个其他 LAN 上的主机发送的入站数据通过。一个专用虚拟网络可以用这种方式建立。

另一种类型的防火墙是应用层的网关(application-level gateway)。与仅检查网络数据包报头的数据包过滤网关相比,这种类型的防火墙实际检查入站或出站消息的内容。一个典型实例是邮件网关,该网关丢弃超过一定大小的入站或出站邮件。也存在更复杂的邮件网关,例如,能够过滤垃圾电子邮件的网关。

应用层网关的另一个例子是允许对数字图书馆服务器进行外部访问,但仅提供文档摘要的网关。如果外部用户想获得更多内容,就要启动一个电子付费协议。防火墙内的用户可以直接访问该图书馆服务。

应用层网关的一种特殊类型称为代理网关(proxy gateway)。这种类型的防火墙作为指定类型的应用程序的前端工作,并且确保只传送符合某个准则的消息。作为实例,我们来考虑在 Web 上冲浪的问题。正如我们在下一节要讨论的,许多 Web 页面包含要在用户浏览器中执行的脚本或小应用程序。为了防止将这样的代码下载到内部 LAN 中,所有 Web 通信都应直接通过 Web 代理网关。该网关接受正常的 HTTP 请求,不论是来

自防火墙内部还是外部。换句话说，该网关对用户表现为一个一般的 Web 服务器。然而，它过滤所有的人站和出站通信信息，或者丢弃某些请求和页面，或者在页面中包含可执行代码时修改页面。

8.3.3 保护移动代码

正如我们在第 3 章中所讨论的，现代分布式系统中的一个重要发展是在主机之间迁移代码的能力，而不是仅仅迁移被动数据的能力。然而，移动代码带来了很多严重的安全威胁。首先，在 Internet 上发送一个代理程序时，其所有者将希望保护该代理程序，防止恶意主机盗窃或修改该代理程序所带的信息。

另一个问题是需要保护主机防止恶意代理程序的破坏。大多数分布式的用户都不是系统技术的专家，他们不知道是否可以信任他们从另一台主机获得的程序不会破坏其计算机。在许多情况下，即使对专家来说完全检测一个正在被实际下载的程序也是很困难的。

除非采取安全措施，否则恶意程序进入计算机之后就可以轻易地破坏其宿主机。我们面临访问控制问题：不应该允许程序对主机资源未经授权的访问。正如我们要看到的，保护主机防止下载的恶意程序的破坏并不总是容易实现的。问题并不是要避免程序下载。相反，我们所寻求的是对移动代码的支持，允许以灵活且完全可控制的方式访问本地资源。

1. 保护代理

在我们讨论保护计算机系统免受下载的恶意代码的破坏之前，首先来看相反的情况。我们来考虑一个移动代理程序，该代理程序以一个用户的名义在一个分布式系统中漫游。这样的一个代理程序可能正在搜索从 Nairobi 到 Malindi 最便宜的机票，并且已由其所有者授权，一旦找到一架班机就预订票。出于这一目的，该代理程序可以携带一张电子信用卡。

显然，在这里我们需要保护。当该代理程序移动到一台主机时，不应该允许该主机窃取该代理程序的信用卡信息。同样，该代理程序也应受到保护以防会使所有者支付比实际需要更多的金额的修改。例如，如果 Chuck 的 Cheaper Charters(便宜机票契约商)可以看到该代理程序还没有访问过价钱比其更便宜的竞争者 Alice Airlines，那么他就会防止 Chuck 改变该代理程序使其不会访问 Alice Airlines 的主机。需要保护代理程序防止来自敌对主机的攻击的另外一些例子包括恶意破坏代理程序，或篡改代理程序以便在其返回时进行攻击或从其所有者处窃取信息等。

不幸的是，不可能完全保护代理程序免受所有类型的攻击(Farmer 等 1996)。这种不可能主要是由无法确实保证主机将完成其许诺过的事情这一事实引起的。因此一种替代方法是以至少可以检测出修改的方式组织代理程序。在 Ajanta 系统中遵循了这种方法(Karnik, Tripathi 2001)。Ajanta 提供了三种机制，允许代理程序所有者检测该代理是否被篡改，即只读状态、只追加记录和有选择地揭示某些服务器的状态。

Ajanta 代理程序的只读状态由代理程序所有者签名的数据项集合组成。签名在代

理程序发送到其他主机前构建和初始化时发生。所有者首先构建一个消息摘要,然后使用其私钥加密该摘要。当该代理程序到达一台主机时,该主机能够通过与原始状态的签名消息摘要比较检验它的状态,可以很容易检测出只读状态是否已被篡改。

为了允许代理程序在主机间移动的过程中能够收集信息,Ajanta 提供了安全的只追加日志(append-only log)。这些日志的特征是数据只能追加到日志后面,在所有者不能检测的情况下不能删除或修改数据。使用只追加日志工作方式如下。最初,该日志是空的,并只有一个以 $C_{init} = K_{owner}^+(N)$ 计算的关联校验和 C_{init} ,其中 K_{owner}^+ 是该代理所有者的公钥,N 是只有所有者知道的一个秘密现时(nonce)。

当该代理程序移至希望传给其一些数据 X 的服务器 S 时,S 将 X 追加到日志后,然后使用其签名 $sig(S, X)$ 签署 X,并计算出校验和:

$$C_{new} = K_{owner}^+(C_{old}, sig(S, X), S)$$

其中 C_{old} 是以前使用的校验和。

当该代理程序回到其所有者处时,所有者可以容易地检验日志是否已被篡改。所有者从结尾开始读取记录,接连对校验和 C 计算 $K_{owner}^-(C)$ 。每次迭代都为下一次迭代返回一个校验和 C_{next} ,以及某个服务器 S 的 $sig(S, X)$ 和的 S。然后所有者就可以检验记录中的 *then-last* 元素是否与 $sig(S, X)$ 相匹配。如果匹配,就删除该元素并进行处理,然后进行下一次迭代步骤。当到达初始校验和,或因为签名不匹配使所有者注意到记录已遭篡改时停止迭代。

最后,Ajanta 支持有选择地揭示状态(selective revealing),该机制通过提供一个数据项矩阵来实现,其中每项为一个指定的服务器。每项都使用指定的服务器的公钥进行加密以确保机密性。整个矩阵由该代理的所有者签名以确保该矩阵作为整体的完整性。换句话说,如果该矩阵的任一项被恶意主机修改,那么任何指定的服务器都会注意到并采取适当行动。

除保护代理防止恶意主机的破坏外,Ajanta 还提供各种机制来保护主机防止恶意代理的破坏。正如我们接下来要讨论的,许多这种机制还由支持移动代码的其他系统所提供。有关 Ajanta 的进一步信息可在(Tripathi 等 1999)中找到。

2. 保护目标

虽然保护移动代码防止恶意主机的破坏是重要的,但人们对保护主机防止恶意移动代码的破坏则给予了更多的注意。如果认为将代理程序发往外部世界太危险,那么用户一般可选择自己完成代理所做的工作。然而,除了完全封锁,通常没有可供选择的方法使代理程序进入用户的系统。因此,如果决定代理程序可以进入系统,那么用户就需要对该代理程序所能做的操作进行完全控制。

正如我们刚刚讨论过的,虽然保护代理程序免遭修改也许不可能,但至少代理程序所有者可能检测到发生了修改。在最坏情况下,所有者将不得不在代理程序返回时丢弃该代理程序,只有这样才不会造成任何损害。然而,处理恶意入站代理程序时,仅检测您的资源是否受到攻击已经太晚了。相反,保护所有资源免受已下载代码的未经授权的访问是基本的功能。

一种保护方法是构造一个沙箱。沙箱(sandbox)是一种技术,使用这种技术下载的程序可以这样的方式执行,即它的每条指令都能够完全被控制。如果某个程序试图执行主机禁止的一条指令,那么该程序的执行就会终止。同样,指令访问主机不允许访问的某些寄存器或内存空间时该程序的执行也会终止。

实现沙箱并不容易。一种方法是在下载可执行代码时对其进行检查,并对只能在运行时检查的情况插入其他指令(Wahbe 等 1993)。幸运的是,在处理解释的代码时事情变得简单很多。我们来简要地考虑 Java 中所采用的方法(参阅(MacGregor 等 1998))。每个 Java 程序都由很多创建对象的类组成。没有全局变量和函数;所有内容都必须声明为一个类的一部分。程序执行从一个名为 main 的方法开始。一个 Java 程序编译为一组由称为 Java 虚拟机解释的指令。对一个客户来说,要下载和执行编译过的 Java 程序,该客户进程必须运行 JVM。JVM 随后将从组成 main 方法的指令开始,通过解释每条指令来处理下载程序的实际执行。

在一个 Java 沙箱中,保护是通过确保处理将程序传输到客户机器的组件是可信的开始的。Java 中的下载由一组类加载程序(class loader)负责。一个类加载程序负责从服务器上获取指定的类,并将其安装在客户的地址空间,以使 JVM 能够从中创建对象。类加载程序只是另一个 Java 类,所以一个下载程序有可能包含其自己的类加载程序。沙箱所处理的第一件事是仅使用可信的类加载程序。特别是,不允许 Java 程序创建自己的类加载程序,通过该加载程序可以不使用类加载通常的处理方式。

Java 沙箱的另一个组件由一个字节代码验证器(byte code verifier)组成,该验证器检查下载的类是否服从该沙箱的安全规则。特别是,该验证器检查该类是否包含不合法的指令或可能以某种方式破坏堆栈或内存的指令。并不是所有的类都要检查,如图 8.27 所示,需要检查的仅仅是从外部服务器下载到客户的类。客户机器上的类一般是可信的,虽然其完整性也可以很容易地进行检验。

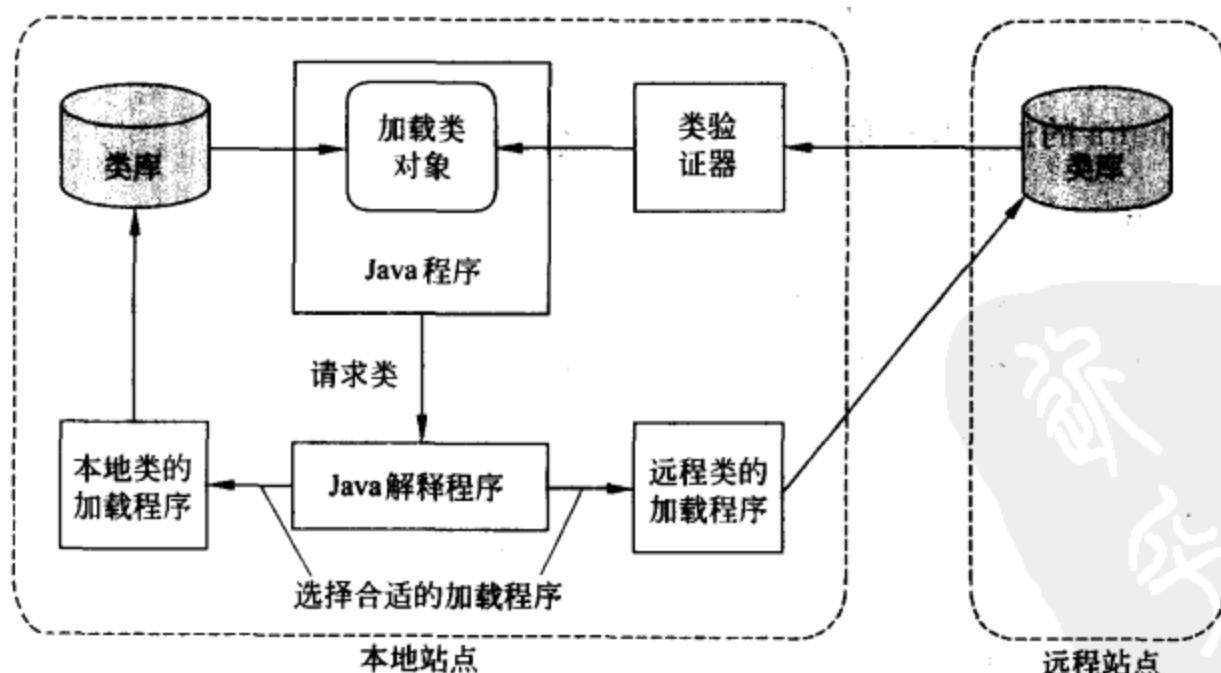


图 8.27 一个 Java 沙箱的组织

最后,一个类安全地下载下来并通过检验后,JVM 就可以从中实例化对象并执行这
• 356 •

些对象的方法。要进一步防止对象对客户资源未经授权的访问,就应使用在运行时执行各种检查的安全管理程序(security manager)。要下载的 Java 程序被迫使用安全管理程序,没有任何方法可使其避免对该程序的使用。这意味着,例如,任何 I/O 操作都受到检查以确定有效性,如果安全管理程序说“不”,就不能执行该操作。因此安全管理程序扮演我们前面讨论的访问监控程序的角色。

典型的安全管理程序禁止执行许多操作。例如,事实上所有安全管理程序都拒绝访问本地文件,且仅允许程序建立到其所在地服务器的连接。显然也不允许使用 JVM。然而,允许程序访问图形库以进行显示,并可以捕获诸如鼠标移动或单击鼠标按键这样的事件。

最初的 Java 安全管理程序实现了相当严格的安全策略,该策略中不同下载程序之间或甚至来自不同服务器的程序之间没有区别。在许多情况中,最初的 Java 沙箱模型限制太严格,需要更多的灵活性。下面,我们讨论当前所采用的一种替代方法。

符合沙箱但提供稍多灵活性的一种方法是为下载的移动代码创建运动场(Malkhi, Reiter 2000)。运动场(playground)是一台专门为运行移动代码保留的单独的指定机器。在运动场内执行的程序可以使用运动场本地的资源,例如文件或到外部服务器的网络连接,这些程序服从标准保护机制的管理。然而,其他机器本地的资源物理上与运动场分离,下载的代码不能访问到这些资源。这些其他机器上的用户可以用传统方式访问该运动场,例如,通过 RPC。然而,没有任何移动代码可以下载到运动场外的机器上。沙箱和运动场的区别如图 8.28 所示。

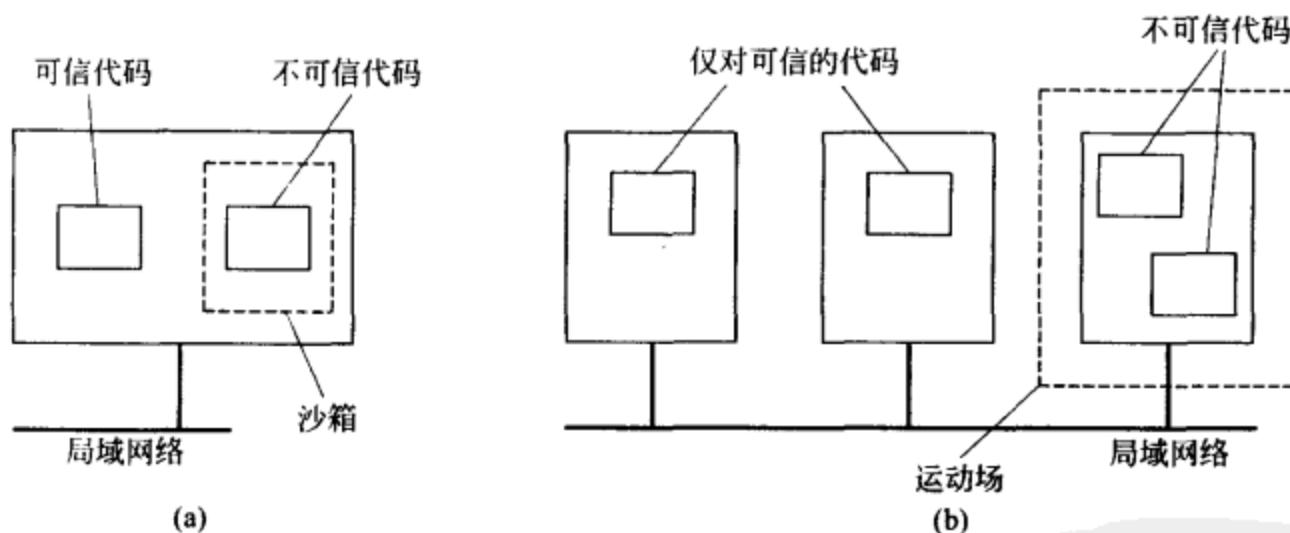


图 8.28 沙箱和运动场

(a) 沙箱; (b) 运动场

趋向更强灵活性的下一步是要求每个下载的程序能够通过身份验证,随后基于该程序的来源执行指定的安全策略。要求程序能够通过身份验证相对容易:可以对移动代码签名,正如对任何其他文档一样。这种代码签名方法通常还作为沙箱的替代方法使用。实际上,只有来自可信服务器的代码才被接受。

然而,困难部分是实施安全策略。Wallach 等(1997)对 Java 程序提出了三种机制。第一种方法使用基于作为权能的对象引用。要访问诸如文件一类的本地资源,程序必须

在下载前已经获得对处理文件操作的指定对象的引用。如果没有获得这种引用，就不可能访问文件。此原则如图 8.29 所示。

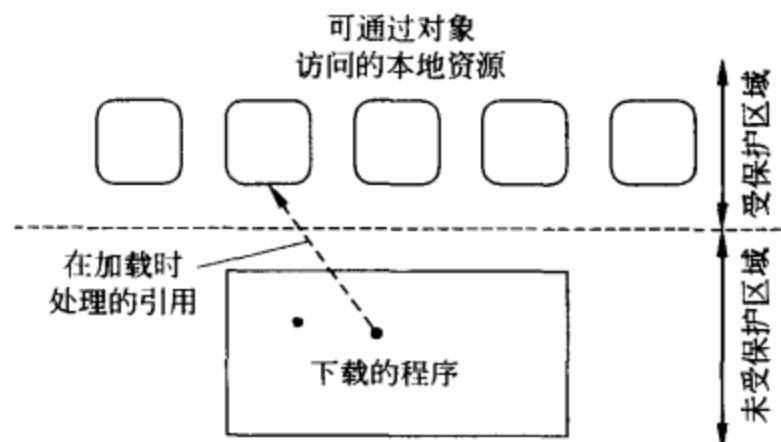


图 8.29 使用作为权能的 Java 对象引用的原则

所有实现文件系统的对象的接口最初都对程序隐藏，这是通过简单地不向这些接口分发任何引用来实现的。Java 的强类型检查确保了在运行时建立一个这种接口的引用是不可能的。此外，我们可以使用 Java 的属性来保存完全属于一个类内部的某些变量和方法。特别是，可以防止一个程序实例化其自己的文件处理对象，这可以通过从本质上隐藏由给定类创建新对象的操作来实现。在 Java 术语中，构造函数对其关联的类是私有的。

执行安全策略的第二种机制是(扩展的)栈自省((extended) stack in trospection)。本质上，对一个本地资源的方法 m 的任何调用都在一个对特殊过程 enable_privilege 的调用之后，该过程检查是否把对该资源调用 m 方法的权限授予调用者。如果该调用是经过授权的，那么该调用者就获得临时特权，持续时间为该调用期间。m 完成时，在将控制返回给调用者之前，调用特殊过程 disable_privilege 来禁用这些特权。

为了执行对 enable_privilege 和 disable_privilege 的调用，要求本地资源接口的开发人员将这些调用插人在适当的位置。然而，令 Java 解释器自动处理这些调用要好很多。举例来说，这是用于处理 Java 小应用程序的 Web 浏览器所遵循的标准方法。一个精致的解决方法如下：只要进行本地资源的一次调用，Java 解释器就会自动调用 enable_privilege，随后该过程检查是否允许进行该调用。如果允许，就将对 disable_privilege 的调用推入栈中以确保该方法调用返回时禁用特权。这种方法防止了恶意编程人员以欺诈回避规则。

使用栈的另一个重要优点是其启用了一种更好的方式来检查特权。假设一个程序调用一个本地对象 O1，该对象又调用对象 O2。虽然 O1 可能具有调用 O2 的权限，但如果不能信任 O1 的调用者调用属于 O2 的一个指定方法，那么就不应该允许这一链式调用。栈自省使检查这样的链很容易，因为解释器仅需要检查每个位于栈顶端的帧来了解是否存在一个帧具有已启用的正确特权（这种情况下允许调用），或者是否存在一个明确禁止对当前资源访问的帧（这种情况下调用被立即终止）。此方法如图 8.30 所示。

本质上，栈自省允许对类或方法的特权进行连接，并对每个调用者的那些特权的分别检查。这样，就可能实现基于类的保护域，在(Gong, Schemers 1998)中进行了详细解释。

实现安全策略的第三种方法是通过名称空间管理(name space management)。其主

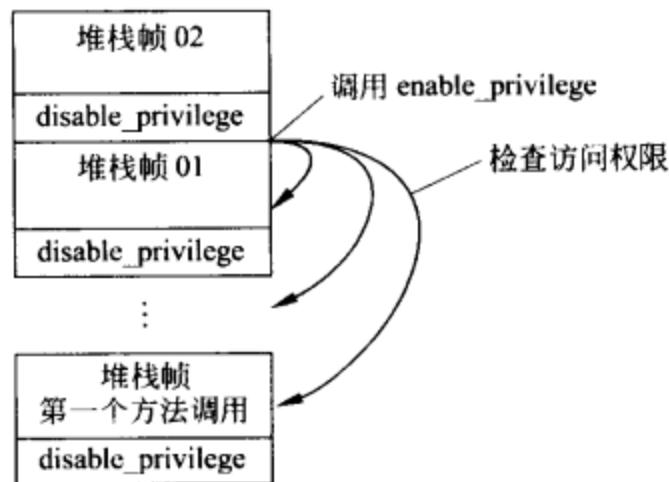


图 8.30 堆栈自省的原理

导思想如下：为了赋予程序对本地资源的访问权限，首先需要通过加入包含实现那些资源的类的适当文件来获得访问。加入过程要求赋予解释器一个名称，然后将该名称解析为一个类，随后在运行时加载该类。要为指定的下载程序实施安全策略，可以根据下载程序的来源将同一名称解析为不同的类。通常，名称解析由类加载程序处理，该程序需要进行修改以实现这种方法。细节可在(Wallach 等 1997)中找到。

到目前为止我们所描述的是基于下载程序的来源将特权与类或方法关联起来的方案。依靠 Java 解释器，通过上面所描述的机制执行安全策略是可能实现的。在这种意义上，安全结构变得高度依赖语言，并且需要为其他语言重新开发。独立于语言的解决方案，例如在(Jaeger 等 1999)中所描述的，需要一种更普通的方法来加强安全，并且还更难以实现。在这些情况下，需要安全操作系统的支持，该系统意识到下载移动代码，并且强迫所有对本地资源的调用在随后进行检查的内核中运行。

8.4 安全管理

到目前为止，我们已经考虑过安全通道和访问控制，但几乎没有谈到安全管理的问题，例如密钥的获得方法。在本节中，我们更深入地讨论安全管理。特别是，我们分成三个主题进行讨论。首先，我们需要考虑加密密钥的一般管理，尤其是公钥的分配方式。正如所证明的，证书在这里起重要作用。

其次，我们通过集中讨论添加当前成员信任的一个新的群组成员来讨论安全管理一组服务器的问题。很明显，面对分布式的和复制的服务，即使接纳一个恶意进程到一个组也不要危及安全这一点是重要的。

第三，我们通过考虑权能和称为属性证书的事物来关注授权管理。分布式系统中关于授权管理的一个重要问题是：一个进程可以将它的一些或所有访问权限委派给另一个进程。以安全方式委派权限具有其自身的微妙之处，正如我们在本节中所讨论的。

8.4.1 密钥管理

到目前为止，我们已经描述了各种加密协议，其中我们(隐含地)假设各种密钥是容易

获得的。例如，在公钥加密系统的案例中，我们假设一条消息的发送者可以使用接收者的公钥，以使其能够加密该消息来确保机密性。同样，在使用密钥分发中心(KDC)的情况下，我们假设每一方都已经与 KDC 共享一个密钥。

然而，建立和分发密钥并不是微不足道的事情。例如，通过不安全的通道分发密钥是不可接受的，在许多情况下我们需要采取带外方法。同样，也需要一些机制来废止密钥，也就是说，要防止密钥受到损害或无效后被继续使用。例如，废止受到损害的密钥是必要的。

1. 密钥建立

我们首先来考虑会话密钥的建立方式。Alice 希望与 Bob 建立一个安全通道时，她可能首先使用 Bob 的公钥来开始通信，如图 8.19 所示。如果 Bob 接受，他随后就可以生成会话密钥并将其用 Alice 的公钥加密后返回给 Alice。通过在传输前加密共享会话密钥，该密钥就可以安全地在网络中传送。

一种相似的方法可以用于在 Alice 和 Bob 已经共享一个密钥时生成并分发会话密钥。然而，这两种方法都要求通信各方已经具有可用的方法来建立安全通道。换句话说，必须已经发生了一些形式的密钥建立和分发。共享密钥通过诸如 KDC 这样的可信的第三方建立时相同的理论也适用。

在不安全的通道中建立共享密钥的一种精致且广泛应用的方法是 Diffie-Hellman 密钥交换(Diffie, Hellman 1976)。该协议工作方式如下：假设 Alice 和 Bob 希望建立一个共享密钥。第一要求就是他们约定两个比较大的数 n 和 g ，这两个数应具有一定的数学属性(我们在这里不予讨论)。 n 和 g 两者都设为公用的，没有必要对外界人员隐藏。Alice 选择一个比较大的随机数 x ，她将其保密。同样，Bob 也选择他自己的保密的比较大的数 y 。此时就有足够的信息来构造密钥，如图 8.31 所示。

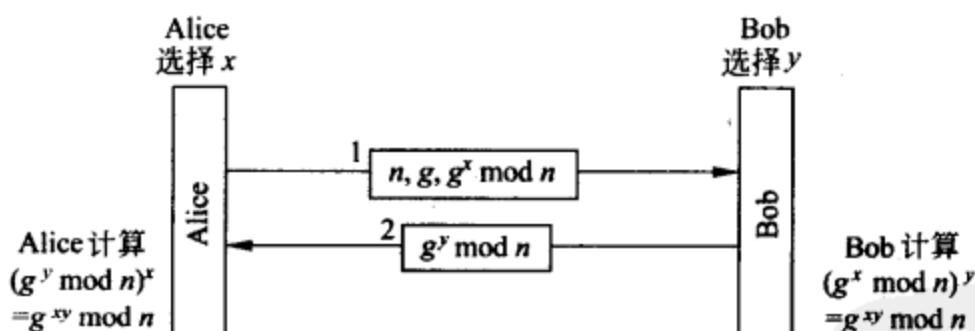


图 8.31 Diffie-Hellman 密钥交换的原理

Alice 以向 Bob 发送 $g^x \bmod n$ 以及 n 和 g 开始。注意此信息可以明文发送这一点很重要，因为事实上不可能在给定 $g^x \bmod n$ 的条件下计算出 x 。Bob 接收到该消息时，他随后计算 $(g^x \bmod n)^y$ ，相当于 $g^{xy} \bmod n$ 。此外，他向 Alice 发送 $g^y \bmod n$ ，然后 Alice 可以计算 $(g^y \bmod n)^x = g^{xy} \bmod n$ 。因此，Alice 和 Bob 两人，而且只有他们两人，此刻就具有共享密钥 $g^{xy} \bmod n$ 。注意他们都不需要让对方知道其私有数(分别为 x 和 y)。

Diffie-Hellman 可以看作一个公钥加密系统。对于 Alice 来说， x 是她的私钥，而

$g^x \bmod n$ 则是她的公钥。正如我们接下来要讨论的，安全地分发公钥对于使 Diffie-Hellman 能实际工作是最基本的。

2. 密钥分发

密钥管理中一个更困难的部分是初始密钥的实际分发。在对称加密系统中，初始共享密钥必须沿着一条提供身份验证和机密性的安全通道交流，如图 8.32(a) 所示。如果 Alice 和 Bob 没有可用的密钥建立这样的安全通道，就有必要在带外分发该密钥。换句话说，Alice 和 Bob 必须使用一些其他通信手段而不是通过网络互相联系。例如，他们中的一人可以给另一人打电话，或者使用缓慢的邮寄方式用软盘发送密钥。

在使用公钥加密系统的情况下，我们需要以这样一种方式分发公钥，即接收者能够确信该密钥肯定可以与声明的一个私钥配成对。换句话说，如图 8.32(b) 所示，虽然公钥本身可以用明文发送，但是发送该公钥所通过的通道必须可以提供身份验证。当然，私钥也需要通过提供身份验证和机密性的安全通道发送。

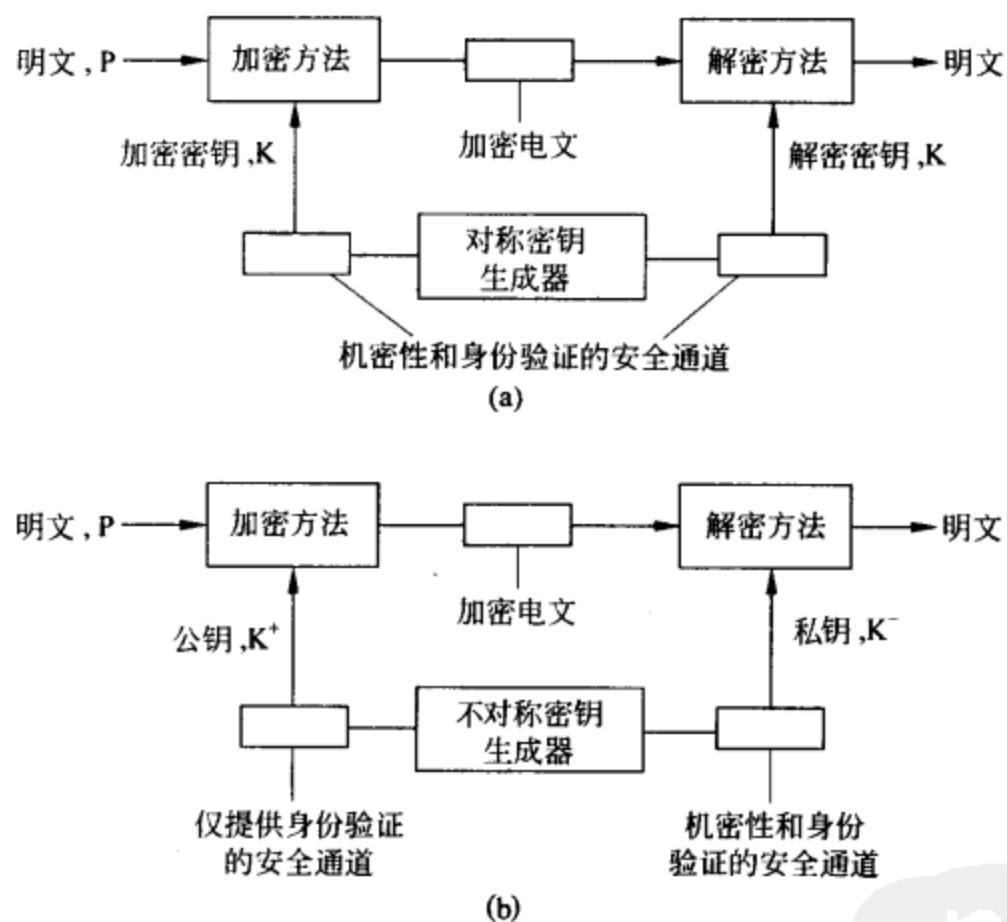


图 8.32 密钥分发和公钥分发(参见文献 Menezes 等 1996)

(a) 密钥分发；(b) 公钥分发

谈到密钥分发时，通过身份验证的公钥分发可能是最有意思的。实际上，公钥分发是通过使用公钥证书进行的。这样的一个证书由一个公钥与一个标识该公钥所关联的实体的字符串共同组成。该实体可以是一个用户，也可以是一台主机或某个特殊设备。该公钥和标识符共同由认证机构签发，并且此签名也置于该证书上。（认证机构的身份自然是证书的一部分。）签名通过一个属于该认证机构的私钥 K_{CA}^- 进行。相应的公钥 K_{CA}^+ 被认为是众所周知的。例如，各种认证机构的公钥嵌入到大多数 Web 浏览器内部并用二进制数

装载。

使用公钥证书的工作方式如下。假设一个客户希望确定在证书中找到的公钥确实属于已标识的实体。然后该客户使用关联的认证机构的公钥来检验该证书的签名。如果证书上的签名与(public key, identifier)对相匹配,那么该客户就确认该公钥确实属于该已标识的实体。

重要的一点是,注意通过确认证书是处于可用状态的,客户实际上就相信该证书不是伪造的。特别是,客户必须假设公钥 K_{CA}^+ 确实属于关联的证书授权机构。如果有所怀疑,就应该能够通过来自可能更可靠的不同认证机构的另一个证书来检验 K_{CA}^+ 的有效性。

必须每个人都信任最高级的证书颁发机构这样的分层信任模型并不是罕见的。例如,PEM(privacy enhanced mail,增强的私密电子邮件)使用一个三级信任模型,其中最低级的证书颁发机构可以由 PCA(policy certification authorities,策略认证机构)进行身份验证,而 PCA 由 IPRA(Internet policy registration authority,Internet 策略注册机构)进行身份验证。如果一个用户不信任 IPRA,或者不认为他可以安全地与 IPRA 谈话,那么他就永远不可能信任使用 PEM 时电子邮件消息能够以安全方式发送。有关此模型的详细信息可在(Kent 1993)中找到。其他信任模型在(Menezes 等 1996)中予以讨论。

3. 证书的生存期

有关证书的一个重要问题是其寿命。首先我们来考虑一下证书颁发机构分发终生证书的情况。本质上,该证书声明的是该公钥对该证书所标识的实体将一直有效。显然,这样的声明并不是我们想要的。如果被标识的实体的私钥曾经受到损害,就没有可信的客户会永远能够使用该公钥(更不用说恶意客户)。在这种情况下,我们需要一种机制,通过令公众知道该证书不再有效来吊销该证书。

吊销证书有多种方式。一种普通方法是使用证书颁发机构定期公布的 CRL (certificate revocation list,证书吊销表)。一个客户无论何时检查一个证书,都必须检查 CRL 来查看该证书是否已经被吊销。这意味着该客户至少必须在每次公布新的 CRL 时联系证书颁发机构。注意,如果 CRL 是每天公布的,那么吊销证书也需要一天。同时,一个已损坏的证书可能被错误地使用,直到在下一期 CRL 中公布。因此,公布 CRL 之间的时间不能太长。此外。获得一个 CRL 会导致一些开销。

一种替代方法是限制证书的生存期。实质上,此方法与我们在第 6 章中所讨论的租用分发相似。证书的有效性在一段时间后自动终止。如果出于任何原因要在到期前吊销证书,那么证书颁发机构依然可以在 CRL 上公布该证书。然而,此方法仍会迫使客户只要检验证书就要检查最新的 CRL。换句话说,客户需要联系证书颁发机构或包含最新 CRL 的可靠数据库。

最后的一种情况是将证书的生存期缩短到几乎为零。实际上,这就意味着不再使用证书;作为替代,客户必须一直联系证书颁发机构以检查公钥的有效性。结果是,该证书颁发机构必须持续联机。

实际上,被分发的证书都有有限的生存期。对于 Internet 应用程序来说,到期时间通常为一年(Stein 1998)。这样的方法要求定期公布 CRL,但在检查证书时也要检查 CRL。

实践证明,客户应用程序几乎没有查阅过 CRL,而只是假设证书在到期以前是有效的。看来实际涉及到 Internet 安全时,还有很多改善的余地。

8.4.2 安全组管理

许多安全系统都利用了特殊的服务机构,例如 KDC(密钥分发中心)或 CA(证书颁发机构)。这些服务机构的存在说明了分布式系统中的一个困难问题。首先,这些服务机构必须是可信的。要提高对安全服务机构的信任,就有必要对它们提供高度的保护以防止所有类型安全威胁。例如,CA 一旦受到损害,就不可能检验公钥的有效性,这会使整个安全系统完全失去价值。

另一方面,同样,许多安全服务机构也必须提供高可用性。例如,对于 KDC 来说,每当两个进程希望建立安全通道时,至少其中一个需要联系 KDC 以获得共享密钥。如果 KDC 不可用,那么安全通信就不能建立,除非可以用一种替代技术来建立密钥,例如 Diffie-Hellman 密钥交换技术。

解决高可用性的方法是复制。但另一方面,复制会使服务器更易受到安全攻击。我们已经讨论了安全组通信可以通过共享组成员之间的一个密钥的方式进行。实际上,没有单个的组成员能够破坏证书,这使组本身非常安全。仍需考虑的是实际管理一组复制服务器的方法。Reiter 等(1994)提出了下面的解决方法。

需要解决的问题是确保进程要求加入组 G 时,该组的完整性没有遭到破坏。假设组 G 使用一个所有组成员共享的密钥 CK_G 来加密组消息。此外,该组还使用一个公钥/私钥对(K_G^+, K_G^-)来与非组成员通信。

只要进程 P 希望加入组 G,它就发送一个标识 G 和 P 的加入请求 JR、P 的本地时间 T、一个生成的响应填充 RP 以及一个生成的密钥 $K_{P,G}$ 。RP 和 $K_{P,G}$ 使用该组的公钥 K_G^+ 共同加密,如图 8.33 中的消息 1 所示。RP 和 $K_{P,G}$ 的使用将在下面更详细地进行解释。加入请求 JR 由 P 签名,并与包含 P 的密钥的一个证书一起发送。我们已经使用了广泛应用的符号 $[M]_A$ 来表示消息 M 已经由主体 A 签名。

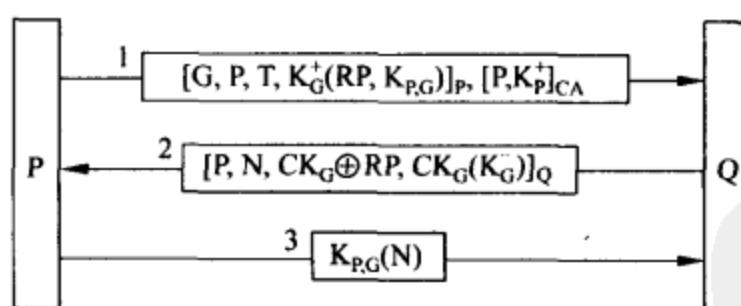


图 8.33 安全接纳一个新的组成员

一个组成员 Q 接收到这样的一个加入请求时,该成员首先对 P 进行身份验证,然后与另一个组成员进行通信来查看是否可以接纳 P 为组成员。P 的身份验证以通常方式依靠证书进行。时间戳 T 用来确定该证书在其发送时是否仍然有效。(注意我们还需要确认该时间没有受到篡改。)组成员 Q 检验该证书授权机构的签名,随后从该证书中提取出 P 的公钥以检查 JR 的有效性。到了这个时间点,就将遵循一个指定组的协议来查看是否

所有的组成员都同意接纳 P 加入该组。

如果允许 P 加入该组, Q 就返回一个组许可进入消息 GA, 如图 8.33 中的消息 2 所示, 标识 P 并包含一个现时 N。响应填充 RP 用于加密该组的通信密钥 CK_G。此外, P 还需要使用 CK_G 加密的该组的私钥 K_G⁻。消息 GA 随后由 Q 使用密钥 K_{P,G} 签名。

进程 P 现在可以对 Q 进行身份验证, 因为只有真正的组成员才能发现密钥 K_{P,G}。此协议中的现时 N 不用于安全, 而是当 P 发回使用 K_{P,G} 加密的 N(消息 3)时, Q 于是知道 P 已经接收到所有必需的密钥, 因此现在确实加入到该组中了。

注意, 如果不使用响应填充 RP, P 和 Q 也可以使用 P 的公钥加密 CK_G。然而, 因为 RP 只能使用一次, 也就是用于消息 GA 中组的通信密钥的加密, 所以使用 RP 更为安全。如果 P 的私钥曾经泄露出来, 那么就也可能泄露 CK_G, 这样可能破坏所有组通信的保密性。

8.4.3 授权管理

分布式系统中的安全管理包括访问权限管理。到目前为止, 我们几乎还没有直接涉及有关怎样把访问权限最初授予用户或用户组的方式, 以及随后以不可伪造的方式维护这些权限的方式的问题。是弥补这一遗漏的时候了。

在非分布式系统中, 访问权限管理相对容易。向该系统中添加一个新用户时, 就向该用户赋予初始权限, 例如, 可以在指定的目录中创建文件和子目录、创建进程、使用 CPU 时间等。换句话说, 用户的一个完整账号是为一台指定机器建立的, 该机器中所有权限都已由系统管理员预先指定。

在分布式系统中, 情况就很复杂, 因为资源散布在多台机器上。如果要遵循非分布式系统的方法, 就有必要为每个用户在每台机器上创建一个账号。实质上, 这就是网络操作系统中所采用的方法。通过在中央服务器上创建唯一的账号可使情况稍微简单一些。每次用户访问某些资源或机器时都会查阅该服务器。

1. 权能和属性证书

在分布式系统中广泛应用的一种好得多的方法是使用权能。正如我们上面简要解释过的那样, 权能是对于指定资源的一种不可伪造的数据结构, 它确切指定它的拥有者关于该资源的访问权限。存在不同的权能实现方式。这里, 我们简要讨论如 Amoeba 操作系统中所使用的实现方式(Tanenbaum 等 1986)。

Amoeba 是最初基于对象的分布式系统之一。其分布式对象的模型是远程对象的模型。换句话说, 一个对象存在于一个服务器中, 客户通过代理对该对象透明访问。为了调用对一个对象的操作, 客户需将权能传送给其本地操作系统, 然后该系统对该对象所在的服务器进行定位, 随后对该服务器进行 RFC。

权能是一个 128 位的标识符, 内部组织如图 8.34 所示。前 48 位由该对象的服务器在创建该对象时初始化, 并有效形成了该对象的服务器的独立于机器的标识符, 称为服务器端口(server port)。Amoeba 使用广播来定位服务器当前所在的机器。

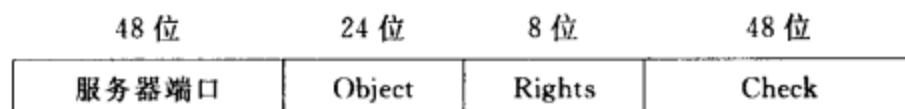


图 8.34 Amoeba 中的权能

其后的 24 位用于标识给定服务器上的对象。注意，服务器端口与对象标识符一起为 Amoeba 中的每个对象构成了一个 72 位的全系统范围的惟一标识符。下面的 8 位用于指定权能持有者的访问权限。最后，48 位的 check 字段用于使权能不可伪造，正如我们在下面几页中要解释的。

在创建一个对象时，其服务器选择一个随机 check 字段并将其同时存储在权能和其自身的表内。一个新权能的所有权限位最初都是 1，返回给客户的就是此所有者权能。权能在执行操作的请求中发送回服务器时，check 字段将被检验。

要创建一个有限权能，客户可以向服务器回传一个权能以及用于新权限的一个位掩码。服务器从其表中获得最初的 check 字段，将其与新的权限（必须是权能中权限的一个子集）进行 XOR 运算，然后通过一个单向函数运行该结果。

然后，服务器创建一个新的权能，object 字段中的值不变，但在 rights 字段具有新的权限位，在 check 字段中为单向函数的输出。新的权能随后被返回其调用者。如果愿意，客户可以将此新权能发送给另一个进程。

在图 8.35 中描述了生成有限权能的方法。在本例中，所有者关闭了除一个权限外的所有权限。例如，该有限权能可以允许读取对象，但不允许任何其他的操作。rights 字段的意义对每种对象类型是不同的，因为对象的合法操作本身也因对象类型的不同而不同。

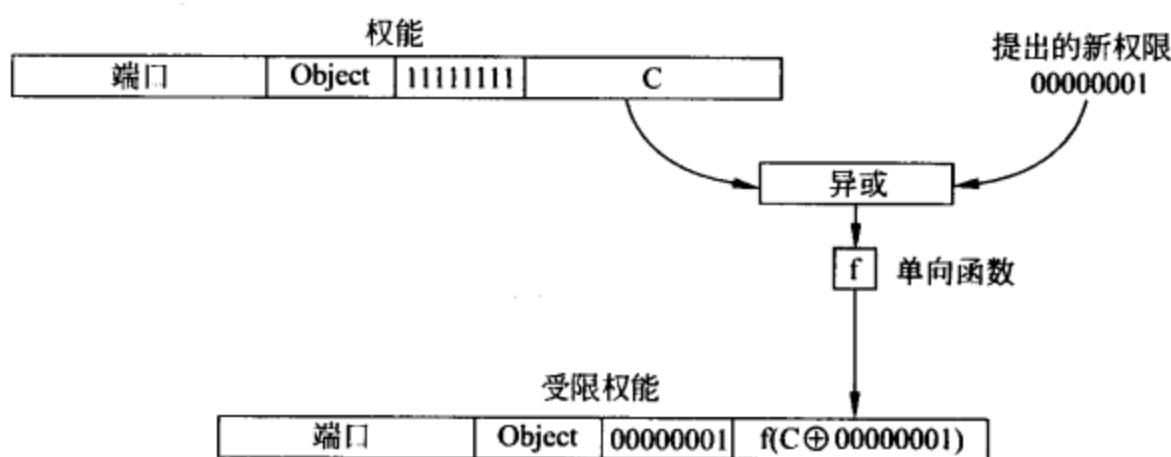


图 8.35 从一个所有者权能生成一个受限权能

有限权能返回服务器时，该服务器从 rights 字段看出其不是一个所有者权能，因为至少有一位被关闭了。然后该服务器从其表中取出最初的随机数，与该权能的 rights 字段进行 XOR 运算，通过单向函数运行该结果。如果该结果与 check 字段一致，则确认该权能是有效的。

从此算法中可以明显看出如果一个用户试图增加其没有的权限的用户只会使该权能无效。反向演算一个受限权能中的 check 字段来获取参数（图 8.35 中的 C XOR 00000001）是不可能的，因为函数 f 是单向函数。权能就是通过这种加密技术获得防止篡改的保护的。注意，在本质上，f 所做的事情与前面所讨论的计算消息摘要所做的事情

相同。改变原始消息中的任何内容(例如反转一位)都会立即检测出来。

在现代分布式系统中使用的权能的一种概括是属性证书(attribute certificate)。与上面所讨论的用于检验公钥有效性的证书不同,属性证书用于列出应用于被标识实体的某些(attribute,value)对。特别是,属性证书可以用于列出证书持有者关于该被标识资源的访问权限。

与其他证书相似,属性证书由称为属性证书颁发机构(attribute certification authority)的特殊证书颁发机构分发。与 Amoeba 的权能相比,这样的一个颁发机构与一个对象的服务器相对应。然而,一般来说,属性证书颁发机构和管理实体(为该实体已创建了一个证书)的服务器不需要是相同的。证书中列出的访问权限由该属性证书颁发机构签名。

2. 委派

现在考虑下一个问题。一个用户希望打印一个他对其有只读访问权限的大文件。为了不过多打扰其他人,该用户向打印服务器发送一个请求,请求其在凌晨 2 点以后开始打印该文件。该用户没有将整个文件发送给打印机,而是将文件名发送给打印机,这样,在如有必要在实际需要时打印机可以将该文件复制到它的 spooling 目录上。

虽然此方法看起来不错,但存在一个主要问题:打印机一般对指定文件不具有适当的访问权限。换句话说,如果不采取特殊措施,一旦打印服务器希望读取该文件来进行打印,系统就会拒绝该服务器对该文件的访问。如果用户临时将其对该文件的访问权限委派给打印服务器,这一问题就可以解决。

对于实现计算机系统,特别是分布式系统中的保护来说,访问权限的委派是一项重要技术。其基本思想很简单:通过将某些访问权限从一个进程传送到另一个进程,使得在多个进程间分布工作变得更容易,而又不会对资源保护产生明显的影响。在分布式系统的情况下,进程可以在不同机器上,甚至不同管理域内运行,就像我们对 Globus 所进行的讨论一样。委派可以避免很多开销,因为保护通常可以在本地处理。

实现委派有多种方式。(Neuman 1993)中所描述的一种普通方法是利用代理。在计算机系统的安全的环境中,代理是一个标记,允许其所有者使用与授予该标记的主体相同的或受限的权限和特权进行操作。(注意这一代理概念与作为客户存根同义词的代理不同。虽然我们试图避免重叠使用术语,但这里是个例外,因为前面定义中的术语“代理”的使用过于广泛,不能忽略。)一个进程最多可以使用其所具有的权限和特权来创建一个代理。如果进程基于当前已有的一个代理创建一个新的代理,那么派生的代理至少具有与原始代理相同的限制,而且可能更多。

考虑委托的一般方法之前,先考虑下面两种方法。首先,如果 Alice 了解每个人,委托就相对简单。如果她希望将权限委托给 Bob,那么她仅需构造一个证书说“Alice 说 Bob 具有权限 R”,例如 $[A, B, R]_A$ 。如果 Bob 希望将这些权限中的一些传送给 Charlie,他会要求 Charlie 与 Alice 联系并向其请求一个适当的证书。

在第二种简单情况中,Alice 可以只构造一个证书说“此证书的持有者具有权限 R”。然而,在这种情况下,我们需要保护该证书以防违法的复制,正如在进程间安全传送权能

中所做的那样。Neuman 的方法可以处理这种情况,而且还避免了 Alice 需要了解需要向其委派权能的每一个人。

Neuman 的方法中的代理有两部分,如图 8.36 所示。令 A 为创建该代理的进程。该代理的第一部分是一个集合 $C = \{R, S_{proxy}^+\}$,由一个已经由 A 委派的访问权限集合 R 以及一个用于对证书持有者进行身份验证的众所周知的密钥部分组成。下面我们将解释 S_{proxy}^+ 的使用。该证书带有 A 的签名 $\text{sig}(A, \{R, S_{proxy}^+\})$,用于保护其免遭修改。第二部分包含该密钥的其他部分,用 S_{proxy}^- 表示。将权限委派给另一个进程时保护 S_{proxy}^- 防止暴露是最基本的。

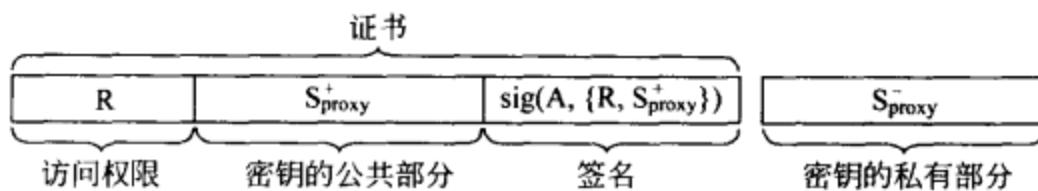


图 8.36 用于授权的一般代理结构

考虑代理的另一种方式如下。如果 Alice 希望将她的一些权限委派给 Bob,她就制作一个 Bob 可以使用的权限列表(R)。通过对该列表进行签名,她就可以防止 Bob 篡改该列表。然而,只有一个签名的权限列表通常是不够的。如果 Bob 希望使用其权限,他就必须证实其是否确实从 Alice 处获得该列表而不是从其他人处窃取的。因此,Alice 提出一个非常难以回答的问题(S_{proxy}^+),只有她知道(S_{proxy}^-)的答案。当给出问题时,任何人都可以轻易地检验答案的正确性。在 Alice 添加其签名之前将该问题追加到权限表后。

当 Alice 委派她的一些权限时,会给予 Bob 签名的权限列表以及该难以回答的问题。她还在确保没有人能够窃听到的情况下给 Bob 该问题的答案。现在 Bob 拥有一个由 Alice 签名的权限列表,他可以在必要时将该列表移交给 Charlie。Charlie 会向其询问列表底部的那个难以回答的问题的答案。如果 Bob 知道该问题的答案,Charlie 就可以确信 Alice 确实已将该表列出的权限委派给 Bob 了。

此方案的一个重要特点是不需要向 Alice 请示。事实上,Bob 可以决定将列表上的(一些)权限传递给 Dave。要这样做的话,他还要告诉 Dave 该问题的答案,以使 Dave 能够证实该列表是由某个有资格的人移交给他的。Alice 根本不需要知道关于 Dave 的事。

用于委派和使用权限的一个协议如图 8.37 所示。假设 Alice 和 Bob 共享一个可用于加密他们相互发送的消息的密钥 $K_{A,B}$ 。于是,Alice 首先向 Bob 发送使用 $\text{sig}(A, C)$ 签名的证书 $C = \{R, S_{proxy}^+\}$ (并重新以 $[R, S_{proxy}^+]_A$ 表示)。没有必要加密该消息,它可以用明

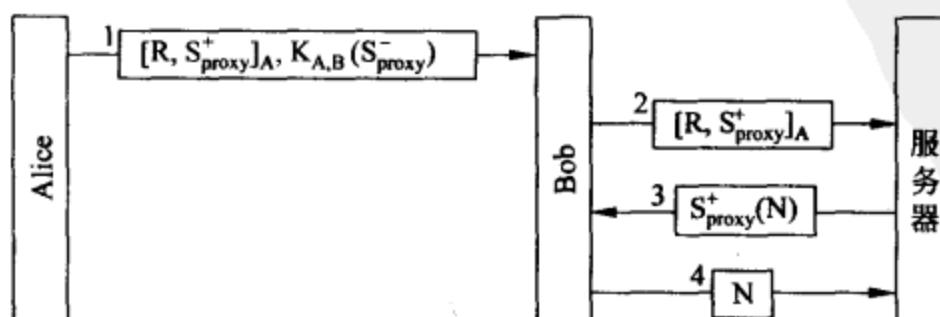


图 8.37 使用代理来授权和证实访问权限的所有权

文发送。只有该密钥的私用部分需要加密,如消息 1 中的 $K_{A,B}(S_{proxy}^-)$ 所示。

现在假设 Bob 希望对指定服务器上的一个对象执行一个操作。还要假设 Alice 被授权可以执行该操作,并且她已经将那些权限委派给 Bob。因此,Bob 将其凭证以签名证书 $[R, S_{proxy}^+]_A$ 的形式移交给该服务器。

这时,该服务器就能够检验 C 是否受到篡改,对权限表或该难以回答的问题的任何修改都会被注意到,因为两者共同由 Alice 签名。然而,该服务器还不知道 Bob 是否是该证书的合法所有者。要检验这一点,该服务器必须使用与 C 一起到来的密钥。

实现 S_{proxy}^+ 和 S_{proxy}^- 有多种方式。例如,假设 S_{proxy}^+ 是一个公钥, S_{proxy}^- 是相应的私钥。于是 Z 可以通过向 Bob 发送一个使用 S_{proxy}^+ 加密的现时 N 来对其进行质询。通过解密 $S_{proxy}^+(N)$ 并返回 N,Bob 证实了他知道该密钥并因此是该证书的合法持有者。实现安全委派还有其他方式,但基本思想始终是一样的:证明你知道一个秘密。

8.5 实例: KERBEROS

到如今应该清楚向分布式系统中加入安全性并不是简单的事情。问题是由于下列事实引起的,即安全应该是普遍的,否则整个系统的安全性可以轻易受到破坏。为了帮助构建能够实施种种安全策略的分布式系统,已经开发了很多可以作为进一步开发的基础的支持系统。一个广泛使用的重要系统是 Kerberos(Steiner 等 1988, Kohl, Neuman 1994)。

Kerberos 是在 M. I. T. 开发的,它基于我们早先描述的 Needham-Schroeder 身份验证协议。当前有两个版本的 Kerberos 在使用:版本 4 和版本 5。两个版本概念上很相似,版本 4 解释起来简单很多。版本 5 对版本 4 添加了很多特性,因此一般是首选的。在下文中,我们只集中讨论身份验证问题。

Kerberos 可以看作一个安全系统,该系统帮助客户与服务器建立安全通道。安全是基于共享密钥的。有两个不同组件:AS(authentication server,身份验证服务器)和 TGS(ticket granting service,票据授予服务)。AS 负责处理来自用户的登录请求。AS 对用户进行身份验证并提供一个可以用于与服务器建立安全通道的密钥。建立安全通道是由 TGS 处理的。TGS 分发称为票据的特殊消息,用于使服务器确信该客户正是其所声称的那个客户。下面我们给出票据的具体例子。

我们来讨论 Alice 登录到使用 Kerberos 的分布式系统的方法,以及她与服务器 Bob 建立安全通道的方法。对 Alice 来说,要登录到该系统,她可以使用任何可用的工作站。该工作站以明文的形式将其名字发送给 AS,AS 返回一个会话密钥 $K_{A,TGS}$ 和一个她将需要移交给 TGS 的票据。

由 AS 返回的票据包含 Alice 的身份,以及 Alice 和 TGS 可以用来互相通信的生成密钥。该票据本身将由 Alice 移交给 TGS。因此,重要的是除 TGS 外没有人能够理解该票据。出于这一原因,该票据使用 AS 和 TGS 之间共享的密钥 $K_{AS,TGS}$ 加密。

登录过程的这一部分如图 8.38 中的消息 1、2 和 3 所示。消息 1 并不是真正的消息,而是对应 Alice 在工作站上键入的她的登录名。消息 2 包含该名称并将其发送到 AS。消息 3 包含会话密钥 $K_{A,TGS}$ 和票据 $K_{AS,TGS}(A, K_{A,TGS})$ 。为了确保私密性,消息 3 使用

Alice 和 AS 之间共享的密钥 $K_{A,AS}$ 加密。

该工作站接收到来自 AS 的响应时,会提示 Alice 键入其密码(如消息 4 所示),随后该工作站使用该密码产生共享密钥 $K_{A,AS}$ (获得一个字符串密码,应用一个加密散列,然后取前 56 位作为密钥是相对简单的)。注意此方法不只具有 Alice 的密码决不会以明文的形式在网络中发送的优点,而且还有一个优点即该工作站甚至不必临时存储该密码。此外,一旦产生共享密钥 $K_{A,AS}$,该工作站就会找到会话密钥 $K_{A,TGS}$,而且可以忘记 Alice 的密码且只使用共享密钥 $K_{A,AS}$ 。

身份验证的这一部分进行后,Alice 可以认为其已经登录到该系统中。她现在可以联系其他用户或服务器。如果她想和 Bob 交谈,就请求 TGS 为 Bob 产生一个会话密钥,如图 8.38 中消息 6 所示。Alice 具有票据 $K_{AS,TGS}(A, K_{A,TGS})$ 这一事实证实了她就是 Alice。TGS 再次使用会话密钥 $K_{A,B}$ 进行响应,该密钥封装在 Alice 随后必须传送给 Bob 的一个票据中。

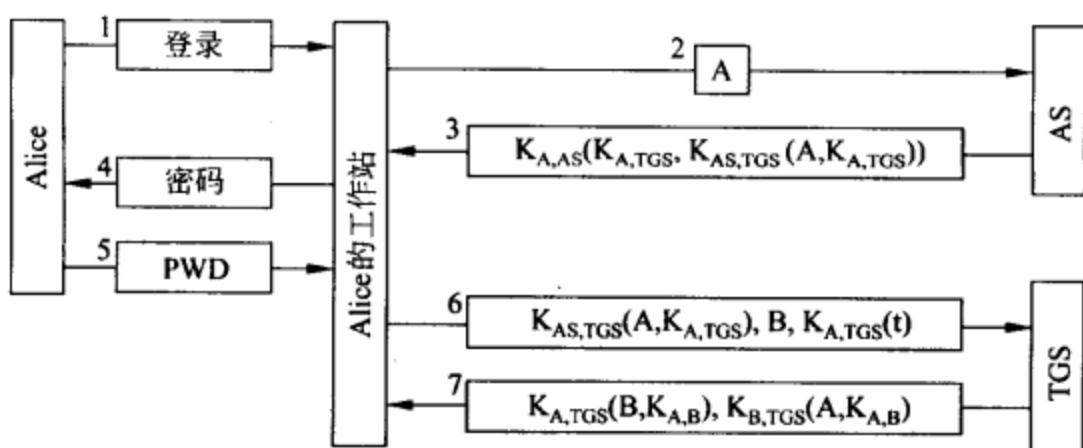


图 8.38 Kerberos 中的身份验证

消息 6 还包含一个使用 Alice 和 TGS 之间共享的密钥加密的时间戳 t 。此时间戳用于防止 Chuck 恶意再次重放消息 6,并试图与 Bob 建立通道。TGS 将在向 Alice 返回一个票据前检验该时间戳。如果它与当前时间的差别大于几分钟,就拒绝对票据的请求。

与 Bob 建立安全通道现在是简单的,如图 8.39 所示。首先,Alice 向 Bob 发送一个消息,其中包含她从 TGS 获得的票据以及一个加密的时间戳。Bob 解密该票据时,会注意到 Alice 将要与其交谈,因为只有 TGS 能够构造该票据。他还找到使其能够检验该时间戳的密钥 $K_{A,B}$ 。那时,Bob 就确信他在与 Alice 而不是恶意重放消息 1 的某个人交谈。通过使用 $K_{A,B}(t+1)$ 进行响应,Bob 向 Alice 证实他确实是 Bob。

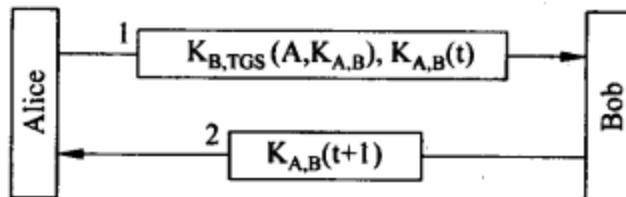


图 8.39 在 Kerberos 中建立安全通道

8.6 实例：SESAME

现在我们来讨论另一个安全系统，该系统与 Kerberos 具有相似之处，但使用公钥加密与共享密钥相结合的方法。

SESAME 最初是一些大型欧洲公司为开放系统开发安全标准的联合项目。SESAME 是 Secure European System for Application in a Multi-vendor Environment(多供货商环境下的欧洲安全应用系统)的首字母缩写。SESAME 的最初实现是在 1991 年完成的，第四次也是最近的实现是在 1995 年，也被称为 SESAME V4。我们首先来给出其主要组件的概述，然后是关于身份验证通过属性证书的处理方法的讨论。我们不集中讨论所有算法的细节，主要关注组件之间总的相互作用来阐明一个安全系统的结构。SESAME 的概述可在(Parker, Pikas 1995)中找到。

8.6.1 SESAME 组件

SESAME 可以看作一个客户-服务器系统，其中客户访问在远程服务器上运行的应用程序。此组织结构导致 SESAME 组件划分为三组，如图 8.40 所示。由一般安全组件、客户端安全组件和服务器端安全组件组成，这三个组件将在下面分别讨论。

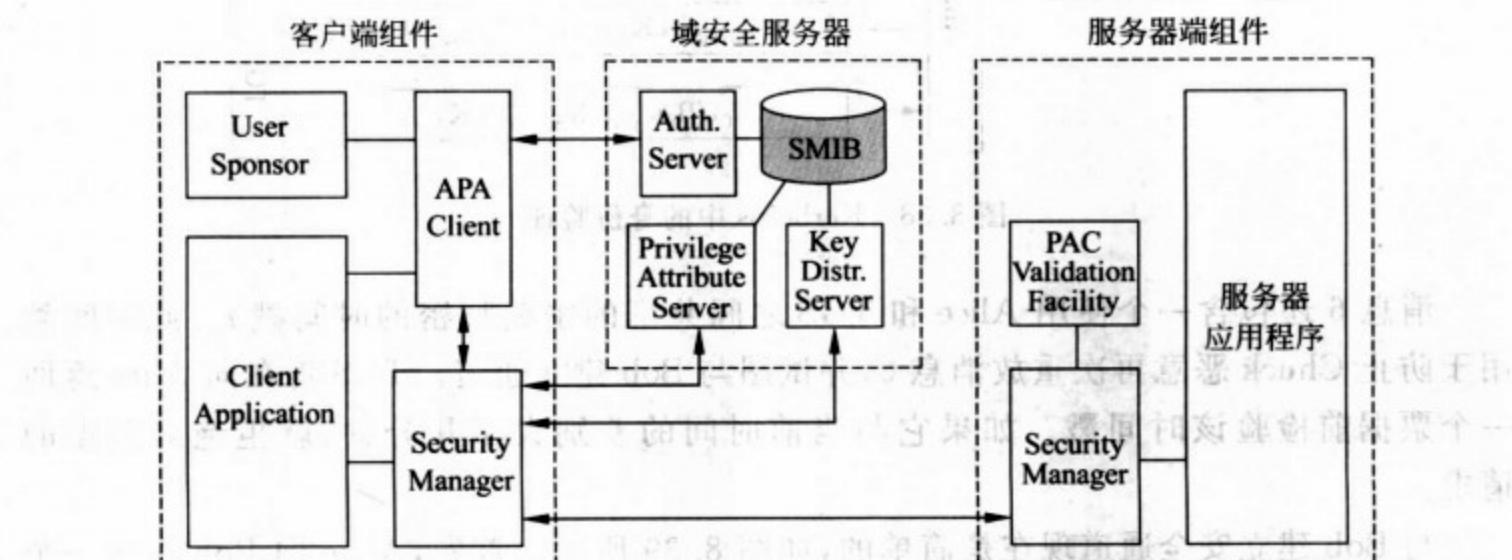


图 8.40 SESAME 中组件的概述

1. 一般安全组件

一般安全组件的集合处理身份验证、授权和密钥分发。他们通常位于单独的一台机器上，该机器称为 DSS(domain security server, 域安全服务器)。

AS 负责对用户(人)和应用程序(程序)进行身份验证。在应用程序的情况下，身份验证通过存储在一个单独数据库中的密钥进行，该数据库称为安全管理信息库(security management information base)或简称为 SMIB。SMIB 是作为本地文件的集合实现的，同时也存储与组成该域安全服务器的其他组件相关的信息。用户如同在 Kerberos 中一样通过从用户密码派生出的一个密钥进行身份验证。然而，如果一个用户拥有一个私钥，

那么该私钥就可用于身份验证。

如果 Alice 希望使用其私钥登录到该系统上,她就向 AS 发送一个登录请求以及带有其公钥并由证书颁发机构签名的一个证书。如果所有事情都正常,AS 就通过发送一个使用 Alice 的公钥加密的会话密钥来进行响应,Alice 可以使用该密钥与 PAS(privilege attribute server,特权属性服务器)(将在下面描述)通信。该响应还包含属性服务器的一个票据,正如在 Kerberos 中一样。同样,返回一个包含其自身公钥的证书。

PAS 的主要目的是准许属性证书列出一个客户拥有的对于一个应用程序的访问权限。我们再接着说 Alice,这意味着 Alice 应该将其身份验证期间返回的票据移交给 PAS。

Alice 还向 PAS 发送一个对其希望联系的一个或多个应用程序的指定访问权限的请求。此请求伴随有加密密封(Gifford 1982),加密密封本质上是一个从请求的访问权限表上计算出的散列。该密封使 PAS 能够检查该表是否受到篡改。该表本身没有加密。当接收到一个请求时,PAS 检查所请求的访问权限是否可以授予,如果可以,就将其捆绑到一个特权属性证书(privilege attribute certificate,PAC)中。下面我们会来更详细地讨论 PAC。除了返回一个 PAC(再次被密封),该属性服务器还产生一个新的会话密钥,Alice 可以使用该密钥与 KDS 通信。然后将此信息放入另一个票据中。

注意,如果此外不需要共享密钥来与客户或服务器的应用程序通信,Alice 就不必与 KDS 通信。然而,为了使采用对称密钥的应用程序更加便于使用,Alice 可以使用其 KDS 票据来请求 KDS 产生会话密钥,以与一个服务器应用程序建立安全通信。在 SESAME 中,KDS 可以使用 Kerberos 的 TGS 实现。

2. 客户端组件

在客户端,存在三个安全组件。首先,对末端用户来说,SESAME 提供一个名为 User Sponsor 的登录程序。此程序使用户能够登录、注销以及改变角色。它作为一个单独的应用程序运行,并假设用户已经登录到本地操作系统中。登录时,用户可以给出一个密码,由该密码产生一个共享密钥用于与域安全服务器的进一步通信。这一登录过程遵循 Kerberos 的登录过程。作为替代,该用户可以启动一个强登录,在这种情况下,用户的私钥在本地重新获得,并像上面所解释的那样使用。

取决于其登录时的角色,用户可获得某些默认特权,进行身份验证之后这些特权会以 PAC 的形式分发。也有可能在登录时请求某些特权。

APA Client(authentication and privilege access client,身份验证与特权访问客户)只是一个包含简单例程的库,这些例程向用户接口和客户应用程序提供对域安全服务器的访问。注意它还在需要的地方支持客户应用程序的身份验证。

第三个组件是安全相关、环境管理(secure association context management)或 SACM。此组件负责初始化和维护客户与服务器应用程序通信所需要的信息。例如,SACM 可在本地保存会话密钥,比如用户登录时获得的 PAC 等。实际上,SACM 避免了令客户应用程序共享与服务器应用程序通信所需的所有类型的安全信息。

SACM 的功能性是诸如 SESAME 一类的许多安全系统所共有的,在其他系统中也

已经以各种方式实现。因此, SACM 没有提供专用接口, 而是实现了标准通用安全服务应用程序编程接口(generic security service application program interface), 简称 GSS-API, 正如在(Linn 1997)中所定义的。

3. 服务器端组件

我们需要讨论的最后一组组件位于服务器端。与客户端相似, 服务器应用程序也拥有其 SACM, 其中保存了与客户和其他应用程序通信所用的安全信息。

服务器的 SACM 负责首先将到来的安全信息传送给 PVF(PAC validation facility)。PVF 从到来的请求中提取并验证所有必要信息。例如, 它可以解密一个消息来提取会话密钥, 或者在检验 PAC 的签名后从 PAC 中提取访问权限。作为另一个例子, PVF 对一个客户进行身份验证之后, 它可以将此信息传回到 SACM, 然后 SACM 会继续向该客户传送一个消息以对服务器进行身份验证。

PVF 可以与指定应用程序位于一台单独服务器上, 或者可以在一台单独的可信任机器上运行, 在该机器上它可以代表许多应用程序工作。在后面的情况中, PVF 服务器和应用程序之间的通信通过身份验证和机密性进行保护是有必要的。

8.6.2 PAC

SESAME 不仅仅处理客户和服务器的身份验证, 也用来支持授权。出于这一目的, 它使用如上所解释的由 PAS 分发的属性证书。为了给出属性证书的具体例子, 我们略为详细地讨论 SESAME PAC。

每个 PAC 都由很多字段组成, 如图 8.41 所示。前两个字段用于表示 PAC 的来源。PAS 的名称可以是针对域的。此外 PAC 还包括一个可用于审计的惟一序列号。

字段	描述
Issuer domain(发行者域)	发行者的安全域名称
Issuer identity(发行者身份)	发行者域中的 PAS 名称
Serial number(序列号)	此 PAC 的一个惟一号码, 由 PAS 产生
Creation time(创建时间)	此 PAC 创建时的 UTC 时间
Validity(有效性)	此 PAC 有效时的时间区间
Time periods(时间段)	附加时间段, 在此时间段外 PAC 无效
Algorithm ID(算法 ID)	用于签名此 PAC 的算法标识符
Signature value(签名值)	此 PAC 上的签名
Privileges(特权)	一个描述特权的(属性, 值)对列表
Certificate information(证书信息)	PVF 使用的其他信息
Miscellaneous(杂项)	当前仅用于审计目的
Protection methods(保护方法)	控制 PAC 使用方法的字段

图 8.41 SESAME PAC 的组成

下面的三个字段与证书生存期有关。该 PAC 创建时的 UTC 时间存储在单独的字段中。这是 PAS 所知道的创建该 PAC 的 UTC 时间。同样，PAC 有效的开始和结束时间也记录在一个字段内。有可能在 PAC 仍然有效的某些时间段中没有使用该 PAC。这些时间段在另一个字段中描述。

签署 PAC 有多种方式。确切使用哪种方法通过一个特定的 SESAME 算法标识符标识。该签名存储在另一个字段中。该 PAC 的有效性随后可与该算法标识符一起由接收者检验。

Privileges 特权字段包含了一个描述该 PAC 的持有者已获得的访问权限的 (attribute,value) 对。每个特权都与一个颁发机构相关联。在默认的情况下，此颁发机构对应发行该 PAC 的安全域，特别是创建该 PAC 的 PAS 所在的域。然而，其他安全域也可能与一个特权关联。

如果需要，可以提供其他信息来帮助正在接收的 PVF 验证该 PAC 的签名。特别是，证书包含签署该 PAC 的 PAS 的公钥，它可以存储在 Certificate information(证书信息) 字段中。最后，还有一个额外的字段存储杂项信息，当前仅用于审计目的。

一个属性证书或者可委派的或者不可委派的。在这两种情况下，都可能指定 PAC 是为了哪个目的(是可委派的或是不可委派的)有效的。所有这些信息都在一个用于该 PAC 的保护方法的单独字段中提供。

8.7 实例：电子付费系统

到目前为止，我们主要考虑了传统分布式系统中出现的安全性问题；一个客户和服务器建立了一个安全通道，这样该客户就可以随后请求执行服务。实质上，所有安全性都集中于两方的相互作用上。

分布式系统逐步发展，已经不仅仅用于客户-服务器通信。特别是，随着对 Internet 访问的增加，我们看到分布式系统正在寻找进入所有种类的电子商务应用领域。所谓电子商务就是人们在 Internet 上进行交易。在本节中，我们简要讨论电子商务的一个重要方面，即商品的实际付费是如何进行的。正如我们将看到的，这样的电子付费系统依靠为分布式系统开发的技术，尤其是在可靠性和安全方面。电子付费系统的概述可在 (Asokan 等 1997, Pfitzmann, Wadner 1996) 中找到。

8.7.1 电子付费系统

付费始终至少包括两方：顾客和商家。在大多数情况中，还会涉及一家或两家银行。无论如何，顾客都必须确保商家得到他的付款。在后面的内容中，我们不考虑正在进行付费的商品的实际交付问题。

可以有不同方式组织付费系统。日常生活中最普遍的支付方式是基于现金、支票和信用卡的。这三种组织方式如图 8.42 所示。在基于现金的系统中，顾客首先必须从其开户银行中取钱，随后他们可以将钱交给商家来进行商品交换。商家稍后可以决定是否使用那些钱来支付给某人，或者将现金存回到自己的开户银行。

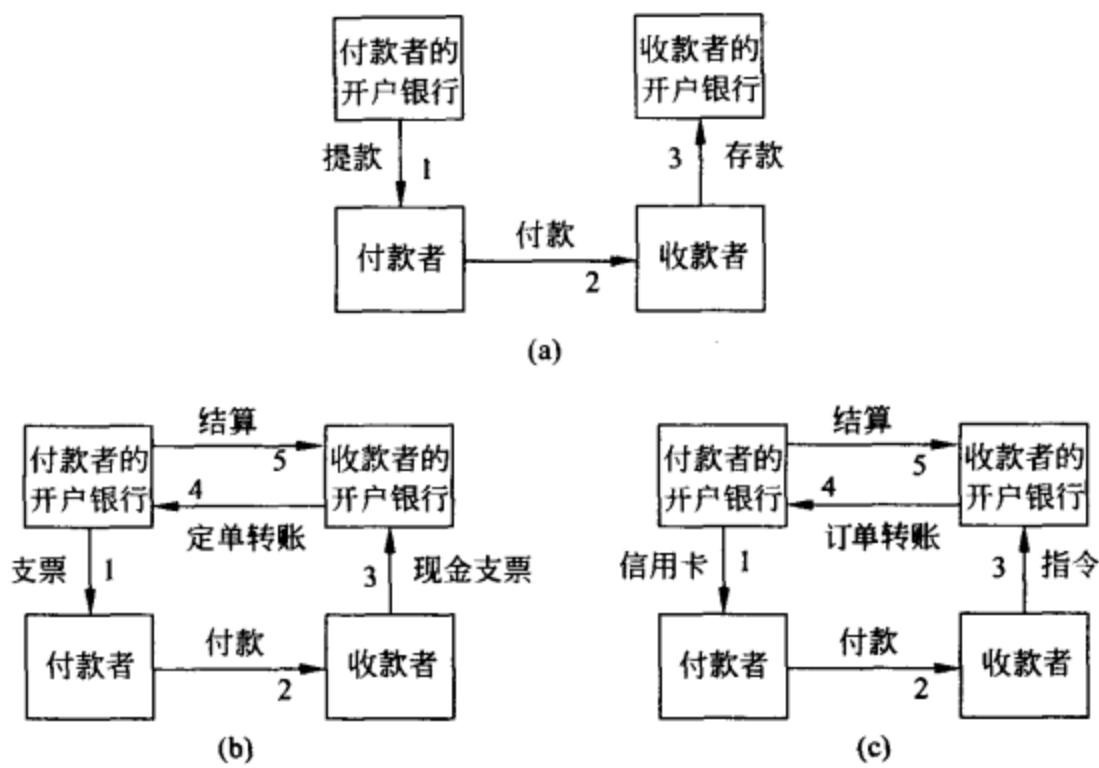


图 8.42 顾客和商家之间基于直接支付的付费系统

(a) 用现金支付；(b) 使用支票；(c) 使用信用卡

基于支票的系统遵循一种不同的方法，如图 8.42(b)所示。在这种情况下，顾客从其开户银行以支票形式获得一个签名的声明，指出向银行呈交该支票的任何人都有权从该顾客的账户中取钱。实际上，该支票被商家兑现之后，结算中就涉及了该顾客的开户银行和商人的开户银行。

第三种方法是基于信用卡的方法，它与使用支票非常类似，如图 8.42(c)所示。为了在信用卡系统中付款，顾客通过交给商家一张信用卡并在信用卡回执上签名有效地通知银行将钱转到商家的账户。钱的实际转账在商家向其自己的开户银行出示此回执，随后该银行与顾客的开户银行联系时发生。

还存在其他形式的支付。例如，通常有可能首先通过明确地将钱从顾客的开户银行转到商家的开户银行来购买商品。钱存入到商家的开户银行中时商家会获得通知，然后就可以进行商品交付了。这种付费系统如图 8.43(a)所示。此系统通常用于不了解顾客并且不希望在进行支付前就发运昂贵产品的公司。

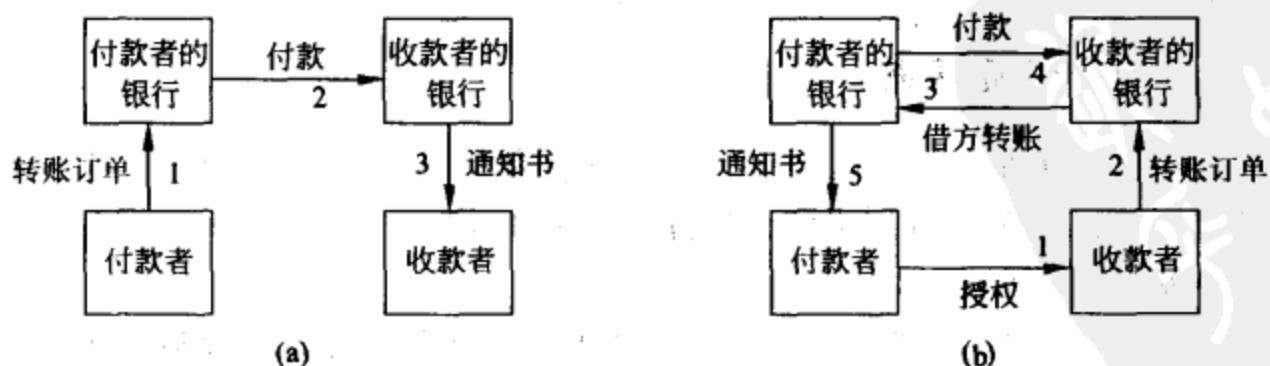


图 8.43 基于银行间资金转账的付费系统

(a) 通过款项订单的支付；(b) 通过借方订单的支付

使用如图 8.43(b)所示的借方订单时,一般由顾客对商家授权来指示商家自己的开户银行从顾客的账户中转入金额,直到到达一个确定的期限。这样的方法通常在需要进行定期付款时使用,例如会员应缴费用或每月的煤气和电的账单。

这些支付模型长期在使用,我们中的大多数人都熟悉至少其中的一些。在其传统的使用中,图 8.42 中的模型要求顾客和商家或多或少在物理上位于同一地点以使支付能够进行。在这样的联系是不可能的或不方便的情况下,可以使用图 8.43 中的模型。然而,随着人们通过 Internet 这样的系统的联系越来越多,在顾客和商家不再位于相同地方时应用图 8.42 中的传统支付模型就变得很困难或者甚至不可能。

因此,我们通常寻找的是电子付费系统的解决方法,这些方法沿用通过现金、支票或信用卡支付的模型,但不要求顾客和商家物理上位于同一地点。下面,在讨论电子付费系统的安全需求后,我们将讨论此类系统的两个实例。有关安全电子付费系统的详细信息可在(Sherif Sechrouchni 2000)中找到。

8.7.2 电子付费系统中的安全性

应该清楚安全性在电子付费系统中扮演重要的角色。没有人希望其银行账户受到未经授权的访问,也没有人会在商家错误地声明支付没有发生过时心存感激。可以证明安全性在付费系统中比在其他类型的分布式系统中扮演更加重要的角色。

1. 一般需求

首先我们通过考虑图 8.42 中所示的支付模型来研究一些较明显的安全需求。在基于现金的系统中,一种广泛应用的方法是利用与银行的一个数据库相联系的自动取款机(ATM)。无论 Alice 何时希望从银行取钱,她首先都必须通过身份验证。她可以以带有一个磁条或芯片的卡的形式使用一个特殊凭证。这种卡可以由 ATM 读取,并通过一般称为 PIN(个人识别码,personal identification number)手段获得保护。Alice 首先必须键入其 PIN,随后 ATM 联系银行。如果身份验证成功,那么 Alice 随后就被授权提取一定数量的钱。(有关基于 PIN 的 ATM 的详细信息,请参阅文献(Davies, Price 1989))。

不使用 ATM 的一种替代方法是 Alice 在家使用其个人计算机从其账户取钱。无论在什么情况下,当钱被转到 Alice 的账户上时,Alice 都应该能够使用其进行支付。电子系统中的一种方法是使用能够存储数字货币的智能卡,数字货币指比特形式的货币。作为选择,Alice 可以指出她希望将其数字货币存储在本地硬盘上。

在基于现金的系统中使用数字货币要求我们有一种保护其使用免受欺骗的方法。特别是,应该既不可能重复使用数字货币,又不可能制造假币。换句话说,明确需要完整性机制。

Alice 使用其数字货币向 Bob 付费时,Bob 应该能够检验该货币的有效性。非正式地说,Bob 能够将货币转到其自己的开户银行并存到其账户中时,数字货币才是有效的。此外,Bob 的开户银行希望确信该货币是“真的”,并且 Alice 和 Bob 都没有篡改过。在一些情况下,这意味着 Bob 的开户银行必须与最初发行该货币的银行相联系。此外,这里还需要传统的身份验证,Bob 的开户银行通过其对 Alice 的开户银行进行身份验证。

现在考虑一个使用支票或信用卡的系统。再一次说明,不难看出,为了向他人分发支票或信用卡需要标准的身份验证和授权需求。Bob 通过一张电子支票收到一个付款时,他同样要求该支票得到完整性保护。他的开户银行也希望看到没有发生篡改的证据。

然而,与基于现金的系统相比,使用支票或信用卡的系统对防止赖账的保护需求更为迫切。例如,在 Internet 上购买物品的普通方式是进行订购,并提供信用卡号。虽然该信用卡号通常在发给商家之前进行了加密,但这并不等于在付款时签名。因此,否认已经发出的订单(理论上)相对容易。可以通过要求顾客插入一个数字签名作为其事务的一部分来提供对订单认可的支持。

与否认债务有关的一个特殊需求是支付是作为(原子)事务执行的,这样就保证它们完全发生或完全失败。这一需求来自这样一个事实,Alice 向 Bob 的书店付款之后,不仅应该保护 Bob 防止 Alice 对订货的否认,也应该保护 Alice 防止 Bob 否认他曾经收到付款。如果系统向 Alice 承认发生过支付,那么就真正意味着 Bob 已经收到该付款,他以后不能否认这件事。

虽然我们的注意力集中于安全问题,但也应该清楚电子付费系统还有要求很高的可靠性需求。特别是,该系统作为一个整体应该是高度可用的,而且应该提供高度的可靠性。

2. 隐私

虽然电子付费系统中的安全需求比在传统分布式系统中要求更高,但原则上,可以使用到目前为止所讨论的技术来满足这些需求。然而,有一个问题应受到特殊关注,并通常在这个问题上将电子付费与其他应用程序分开,它就是隐私。

仅保护支付免受窃听相对简单,因为我们可以使用标准加密技术。困难的部分在于实现匿名。在标准现金系统中,匿名支付是很简单的:Alice 仅交给 Bob 一袋硬币作为她所购买的物品的回报。Alice 没有提出任何问题。Alice 一般不需要知道 Bob 是谁,而且 Bob 也确实不大关心 Alice 是谁。惟一有关系的事情是物品是为了钱而交换的。

Alice 和 Bob 使用电子付费系统时事情就变得有一些复杂。我们首先需要考虑的一件事是微数据的安全。微数据是与伴随每个事务的小数据元素,这些元素聚集在一起时,可以揭示事务中涉及的一方的身份。例如,让我们考虑匿名支付的情况,匿名支付记录了发起支付的主机以及支付时间。如此信息与登录账号相匹配则将立即揭示顾客身份。其他通常可能透露的信息是特定的属性值,可以逐渐产生特定人员所感兴趣的信息的完整轮廓。

实际上,保护系统避免隐私受侵害的焦点在于令顾客匿名。一般没有必要也令商家匿名。在这种情况下匿名意味着事务进行期间或结束后不可能识别顾客。有时候,有必要揭示该身份,例如,在纠纷情况下。这种情况被称为条件匿名。

隐藏顾客身份的另一种方法是使用假名。假名是在一个事务或一系列事务中使用的一个别名。它具有一种属性,即顾客可以标识为该名称,但原则上,不可能按身份将实际用户标识为顾客。通常,可以使用假名来实现条件匿名。

为了给出数据在付费系统中的隐藏程度的概念,我们首先考虑一个基于传统现金系
• 376 •

统的系统(参见文献 Camp 1996)。图 8.44 中的每列都显示了需要隐藏的事务属性。我们区别 5 种属性：商家的身份、顾客的身份、事务的日期、支付的金额以及销售的物品的信息。每行显示了事务中可能涉及的一方。我们区别三方：商家、顾客和银行。此外，我们假设存在一个理想地放置的观察者(即 Chuck)，他可以简单地观察事务的发生。例如，在商店购买商品的情况下，Chuck 可能是队列中的下一个人。

		信息				
		商家	顾客	日期	金额	物品
参与者	商家	完全	部分	完全	完全	完全
	顾客	完全	完全	完全	完全	完全
	银行	无	无	无	无	无
	观察者	完全	部分	完全	完全	完全

图 8.44 传统现金支付中隐藏的信息

对涉及的每一方来说，一个数据项描述了指定属性值的可见程度。例如，对基于现金的事务中的一个顾客来说，商家、事务日期、涉及的金额，还有实际购买的物品是可见的。然而，商家不总需要知道顾客的身份，在商店购买商品通常就是这种情况。

一个有趣的事情是银行实际上什么都不了解，虽然可以证明它可以仅通过观察商家最终作为事务的一部分存入的资金数量来对资金总数有一个概念。换句话说，为了使顾客完全匿名，该事务本身是不能记录的。商家在非常大的事务中不需要完全匿名。大多数银行都合法地需要在商家存钱时注意到这样的事务。

传统信用卡事务怎样呢？谈到隐藏的信息，不难看出图 8.44 中提到的几乎所有属性都对所有方可见。然而，观察者辨认顾客可能有一些困难，而银行决不会知道实际购买的商品是什么。这就导致了图 8.45 中所示的表格。

		信息				
		商家	顾客	日期	金额	物品
参与者	商家	完全	完全	完全	完全	完全
	顾客	完全	完全	完全	完全	完全
	银行	完全	完全	完全	完全	无
	观察者	完全	部分	完全	完全	完全

图 8.45 传统信用卡系统中的信息隐藏(参见 Camp 1996)

很明显，信用卡支付不提供很多隐秘性，更不用说匿名了。这样的付费系统的电子版本至少应该提供相同的安全性，而且可能应该更多。例如，一般不接受通过开放网络以明文发送信用卡号码的方式。

8.7.3 协议实例

为了更好地理解电子付费系统中的安全问题，我们简要讨论两个有代表性的系统。

首先,我们讨论一个允许使用数字货币支付的系统。其次,我们描述一个用于 Internet 上安全信用卡事务的标准化协议。

1. E-cash

有很多基于数字货币概念的电子付费系统。最有名的是由 Chaum(1985 1992)开发的 e-cash 系统,该系统主要关注的是在付费系统中实现匿名。为了理解 e-cash 的工作方式,我们首先通过描述一个非匿名的基于现金的系统来考虑其基本概念。

假设 Alice 希望通过电子手段使用数字货币从 Bob 处购买一些商品,总价为 \$12。她首先联系其开户银行并请求从其账户中提取 \$12。该银行以代表某一价值(也就是钱)的签名票据的形式交付数字货币,每一价值具有一个惟一相关的签名。例如,一个 10 美元的票据可以使用银行所有的特殊私钥 K_{10} 签名,而一个 1 美元的票据则要以另一个特殊私钥 K_1 签名。为了防止复制这些票据,每个票据中都带有一个惟一序列号,银行可以使用该序列号来检查票据是否多次使用。

在我们的例子中,Alice 从她的开户银行接收到一个 10 美元的票据和两个 1 美元的票据,随后她将这些票据交给 Bob。Bob 可以通过银行的签名来检验这些票据没有经过伪造。然而,为了检查这些票据代表的钱是否还没有消费,他需要联系银行,随后银行可以通过检查序列号来检查钱是否已经存入。如果钱没有被消费,那么就向 Bob 发送一个 OK, 表示已经接受了这些钱。注意,如果 Bob 无论如何都需要联系银行,那么同时他也可以要求银行检查票据的完整性。

此方法的明显缺点是银行需要保存票据的序列号以使其能够准确追踪钱的流向。如果 Alice 请求一个 10 美元的票据,那么银行就记录下其交付给 Alice 的票据的序列号,随后确认 Bob 将其存入。除非某一其他方接受了来自 Alice 的支付,并相信那些钱还没有消费,否则不会有除 Alice 在向 Bob 付费外的其他结论。

此系统中所需的是这样的一种方式,即防止银行能够跟踪其将哪些数字货币交给了哪些人,但仍能够保护自身免受票据伪造和二次花费的损害。这种保护可通过称为隐蔽签名(Chaum 1982)的方法实现。隐蔽签名背后的原理通过里面由复写纸覆盖的信封很容易进行解释(Chaum 1985)。如果将一张纸放在这样的一个信封中,那么在信封上写的任何字同样会出现在里面的纸上。

对 Alice 来说,要让银行签署其 10 美元的票据,她就要将该票据放在里面有复写纸的信封中,并请求其开户银行在信封上签名。银行将从 Alice 的账户中提取 \$10 并在信封上签名。这样银行就决不会看到实际的票据,但同时已将其 10 美元的签名签在上面。然后 Alice 可以使用该票据作为正常的货币。假设银行每天签署许多 10 美元票据,则此方法很有效。但在这种情况下,Bob 交给银行一个 10 美元的票据,请求将 \$10 存入其账户,而通过该票据来追查 Alice 是不可能。

为了解释这些签名如何使用 RSA 进行工作,我们使用与前面解释 RSA 的原理时相同的符号: $n = p \times q$, 其中 p 和 q 是两个比较大的素数(并且是保密的)。数字 d 是与 $z = (p-1) \times (q-1)$ 有关的素数,选择 e 使 $e \times d = 1 \pmod{z}$ 。

假设 Alice 希望从其开户银行账户取出一个 10 美元的票据 m 。实质上,她自己产生

该票据,但需要银行签名来使其有效,随后银行不能反向追查。她产生一个 1 和 n 之间的随机数 k ,通过将 m 加密成下面的消息,使 m 不可读:

$$t = mk^e \bmod n$$

银行使用其私钥 K_{10}^- ,在本例中是 d 对 t 签名,使其变成消息 t^d 。注意,我们正在使用为 10 美元票据特别准备的密钥,需要为其他面值的票据准备不同密钥。Alice 接收到 t^d 时,她将需要通过计算来解开该消息:

$$s = t^d / k \bmod n = (mk^e)^d / k \bmod n = m^d k^{d-1} \bmod n = m^d \bmod n$$

这时,她拥有一个与由银行签名的 10 美元票据一样的 10 美元票据,但银行从来没有看到过 m 。现在 Alice 可以使用 m^d 作为一个 10 美元票据支付给 Bob,如图 8.46 所示。

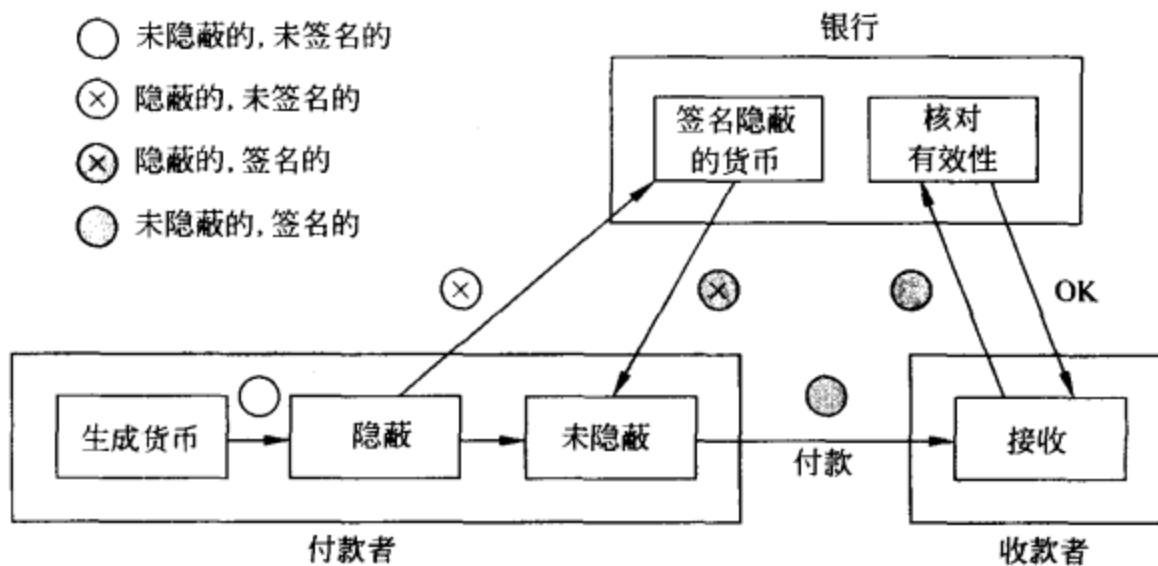


图 8.46 使用隐藏签名的匿名电子现金的原理

虽然此协议基本上是适宜的,但仍存在一些问题。主要问题是如何防止二次花费。虽然银行可以容易地检查 m 是否已经存入,但无疑它希望处罚试图两次使用 m 的人。换句话说,我们想要一种方法,通过这种方法,二次花费本身就可以提供足够的信息来揭示花费者的身份。这样的方法的细节在(Chaum 1985)中描述,也可以在(Schneier 1996)中找到。

2. 安全的电子事务

现在我们来看一个用于信用卡事务的电子付费系统。SET (secure electronic transactions, 安全电子事务) 协议是 Visa 和 Mastercard 协同诸如 Netscape 和 Microsoft 的几个其他公司共同努力,开发出来的用于在网络上使用信用卡购买商品的一种标准方式。SET 是一个开放标准,这意味着整个协议已经公布(有关 SET 的所有信息可在 <http://www.setco.org> 上找到)。

在下面的内容中,我们介绍实际协议的梗概,不考虑诸如指定消息格式、质询内容、时间戳等许多细节。我们的描述的主要目的是给出 SET 作为电子信用卡系统的一个多方协议的一个概述。

SET 中使用的一个重要且新颖的概念,而且是我们在研究组成该协议的各个步骤之前需要首先解释的是双重签名的概念。考虑下面的问题。Alice 希望使用信用卡从 Bob

处购买一些商品。她想做的是向 Bob 发送订货信息以及有关她的信用卡的信息。Bob 应该能够将后者传送到其开户银行以完成商品支付。所需要的是将订单与支付信息紧密结合在一起。

一种可能性是将此信息放在单个消息中，并让 Alice 对其签名。然而，在这种情况下，Bob 和银行都同时需要订单和支付信息以检验 Alice 的签名。因此，银行会了解 Alice 所订购的物品。通常，此信息应在 Alice 和 Bob 之间保密。同样，Alice 不希望 Bob 知道支付信息的每个细节，决定 Alice 是否可信是银行的责任，不是 Bob 的责任。

解决方法如下所示：令 m_1 和 m_2 分别表示订单和支付信息。一个双重签名包括构造 m_1 的一个消息摘要 $md_1 = H(m_1)$ 和 m_2 的 $md_2 = H(m_2)$ ，然后用 md_1 和 md_2 的连接构造第三个摘要： $md_{dual} = H(concat(md_1, md_2))$ 。然后由 Alice 使用其私钥对此摘要 md_{dual} 进行签名。现在 Alice 向 Bob 发送 $[m_1, md_{dual}, md_2, K_A^-(md_{dual})]$ 。此消息将使 Bob 能够对 Alice 进行身份验证，并检验她是否在整个消息上进行了签名。然而，对 Bob 来说不可能读取 m_2 。我们使用符号 $[m_1 | m_2]_A$ 来表示消息 m_1 和 m_2 上的双重签名，其中只有 m_1 是公开的：

$$[m_1 | m_2]_A = [m_1, H(m_2), H(concat(m_1, m_2)), K_A^-(H(concat(m_1, m_2)))]$$

现在我们来考虑 SET 协议的基础。组成该协议的不同步骤如图 8.47 所示。

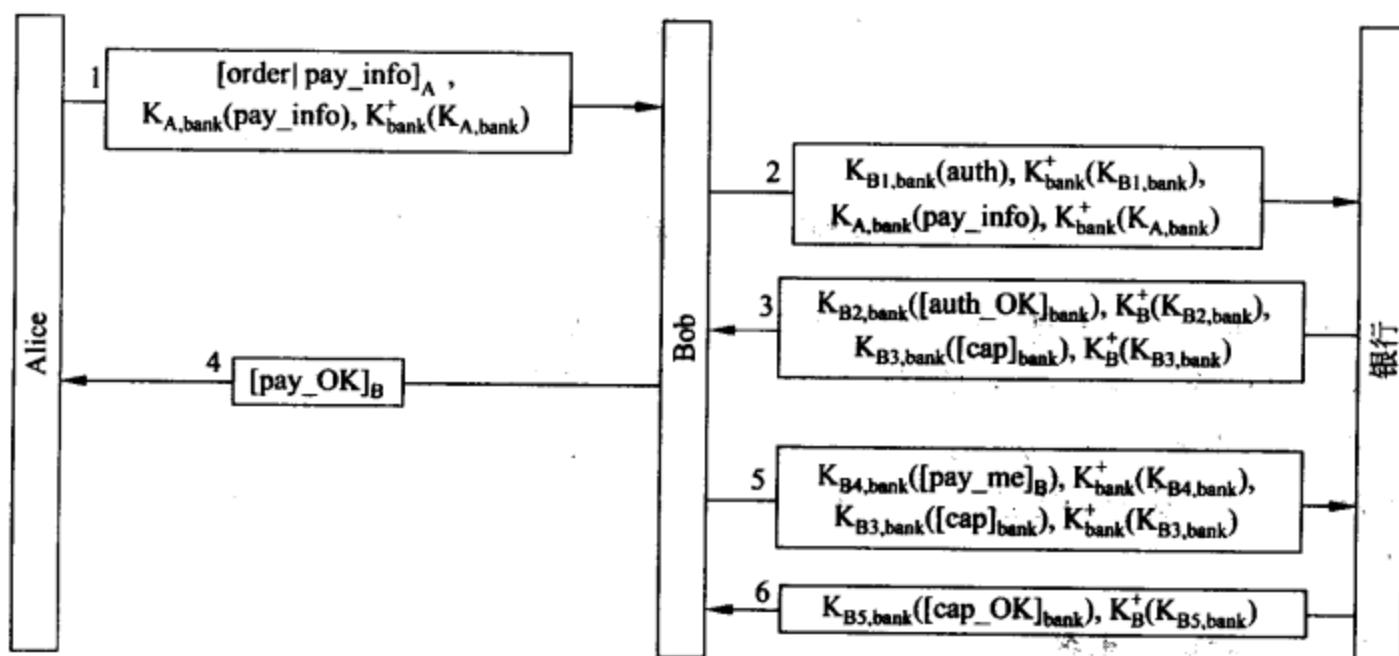


图 8.47 SET 中的不同步骤

我们假设 Alice 知道她想从 Bob 处购买的东西，她需要做的第一件事是向 Bob 发送订单和支付信息。在图 8.47 中，这作为消息 1 显示。订货信息显示为 order，而支付信息则显示为 pay_info。Alice 构造经过双重签名的消息 $[order | pay_info]_A$ 。只能由银行读取的支付信息通过 Alice 生成的会话密钥 $K_{A,bank}$ 加密。这一密钥也发送给 Bob，但使用银行的公钥 K_{bank}^+ 加密。

Bob 检验过来自 Alice 的消息后，接着他就将支付信息转发给银行，并请求对该支付进行授权。他生成一个用于加密其授权请求 auth 的会话密钥 $K_{B1,bank}$ 。此请求与 Alice 的支付信息以及使用银行的公钥加密的 $K_{B1,bank}$ 一起发送到银行，如图 8.47 中的消息 2

所示。

假设银行对支付进行了调查,它向 Bob 返回一个签名的授权消息 auth_OK,该消息使用新产生的会话密钥 $K_{B2,bank}$ 加密,而该密钥则使用 Bob 的公钥加密。此响应,如消息 3 所示,还包含称为捕获消息的内容,该内容稍后可由 Bob 使用来从银行获得实际支付。捕获消息依靠一个事务标识符与整个事务惟一关联。此标识符还在诸如 order 和 pay_info 的消息中出现。捕获消息由银行签名,并通过会话密钥 $K_{B3,bank}$ 加密。

Bob 接收到银行的调查信息时,向 Alice 发送一个签名确认,如图 8.47 中消息 4 所示。

现在 Bob 需要获取他的钱。因此他向银行发送一条消息 pay_me 以及先前返回的捕获消息。此外,产生另一个会话密钥用于加密 pay_me,这就产生了消息 5。

最后,在银行检查过捕获消息后,向 Bob 返回一个使用 $K_{B5,bank}$ 加密的确认 cap_OK。

虽然我们没有在图 8.47 中显示,但 SET 在每一步中都包含身份验证。每次需要使用公钥时,身份验证都是通过发送证书而进行的。出于这一目的,应利用分层的可信证书颁发机构,这与我们前面讨论的 PEM 中的分层可信模型相似。有关此模型以及该协议的细节可在(SET 1997)中找到。

8.8 小结

安全在分布式系统中扮演一个非常重要的角色。分布式系统应该提供一些机制来实施多种不同的安全策略。开发和正确应用这些机制一般使安全成为了一项困难的工程实践。

三个重要问题可以区别开来。第一个问题是分布式系统应该提供建立进程间安全通信的功能。原则上,安全通道提供通信各方互相进行身份验证的方法,并保护消息在传输期间免受篡改。安全通道一般还提供机密性,使得除通信参与方外没有人能够读取该通道中通过的消息。

一个重要的设计问题是只使用对称加密系统(基于共享密钥),还是将其与公钥系统结合起来使用。当前的实践显示,为分配短期共享密钥进行的公钥加密的方法使用较多。短期共享密钥称为会话密钥。

安全分布式系统的第二个问题是访问控制和身份验证。身份验证以这样一种方式处理资源保护问题,即仅具有适当访问权限的进程能够实际访问和使用那些资源。访问控制总在进程通过身份验证后发生。

实现访问控制有两种方式。首先,每个资源能够保持一个访问控制表,正确列出每个用户或进程的访问权限;另一种方式是进程可以携带一个证书明确声明其对一个特定资源集合的访问权限。使用证书的主要好处是进程可以容易地将其票据传至另一个进程,也就是说,委派其访问权限。然而,证书具有通常难以吊销的缺点。

在移动代码的情况下处理访问控制时需要特别注意。除了要能够保护移动代码免受恶意主机危害之外,一般来说更重要的是保护主机免受恶意移动代码侵害。人们已经提出了若干个建议,其中沙箱是当前应用最广泛的一个。然而,沙箱有一定的限制性,而且

基于真正的保护域的更灵活的方法也已设计出来。

安全分布式系统中的第三个问题关系到管理。本质上有两个重要的子专题：密钥管理和身份验证管理。密钥管理包括加密密钥的分配，可靠的第三方发布的证书对其扮演重要的角色。关于身份验证管理的重要点在于属性证书和委派。

Kerberos 是一个广泛使用的安全系统，它基于共享密钥。虽然其中还加入了访问控制和访问权限委派的协议，但其主要焦点在于身份验证。

SESAME 是可结合到分布式系统中的安全系统的典型实例。该系统基于公钥加密和共享密钥的组合使用。它从 Kerberos 系统中借鉴了很多东西，并且投入努力使其与 Kerberos 兼容。

最后，电子付费系统为分布式系统构成了一个重要的应用领域，并且通信各方可以跨广域网分布这一点特别有意义。通常对顾客的匿名加以特殊关注，因为这一点将传统的基于现金的系统与其电子付费系统区别开来。

习 题

1. 分布式系统应该提供哪些机制作为对应用程序开发人员的安全服务，这些开发人员仅相信系统设计中的端到端参数吗（如第 5 章所述）？
2. 在 RISSC 方法中，是否所有安全性都集中于安全服务器上？
3. 假设要求您开发一个可使老师设立考试的分布式应用程序。请给出至少三个可作为这样的应用程序的安全策略中一部分的说明。
4. 将如图 8.12 中显示的身份验证协议中的消息 3 和消息 4 结合为 $K_{A,B}(R_B, R_A)$ 是否安全？
5. 在图 8.15 中，KDC 接收到对 Alice 和 Bob 可以共享的一个密钥的请求时，为什么不需要确切知道其在与 Alice 交谈？
6. 将现时（nonce）作为一个时间戳实现有什么错误？
7. 在 Needham-Schroeder 身份验证协议的消息 2 中，票据使用 Alice 和 KDC 之间共享的密钥进行加密。这一加密是必需的吗？
8. 我们能安全地修改如图 8.19 中所示的身份验证协议，使消息 3 仅由 R_B 组成吗？
9. 设计一个使用公钥密码系统中的签名的简单身份验证协议。
10. 假设 Alice 希望向 Bob 发送一条消息 m 。她没有使用 Bob 的公钥 K_B^+ 加密 m ，而是生成了一个会话密钥 $K_{A,B}$ ，然后发送 $[K_{A,B}(m), K_B^+(K_{A,B})]$ 。为什么一般来讲，这种方法更好？（提示：考虑性能问题。）
11. 在访问控制矩阵中如何表示角色改变？
12. 在 UNIX 文件系统中 ACL 如何实现？
13. 一个公司如何加强 Web 代理网关的使用并防止其用户直接访问外部 Web 服务器？
14. 参考图 8.29，将 Java 对象引用作为权能的使用实际上在多大程度取决于 Java 语言？

15. 如本章中所解释的那样,列举出需要本地资源接口的开发人员插入调用以启用和禁用特权时会遇到的三个问题,这些特权是用来防止通过移动程序进行未经授权的访问的。
16. 列举出为密钥管理使用集中式服务器的一些优点和缺点。
17. Diffie-Hellman 密钥交换协议还可以用于建立一个三方之间的共享密钥。解释一下如何实现。
18. Diffie-Hellman 密钥交换协议中不存在身份验证。通过利用这一特性,一个恶意的第三方 Chuck 能够轻易地插入 Alice 和 Bob 之间进行的密钥交换,随后破坏安全性。解释这一行为是如何工作的。
19. 给出一种方法,说明在 Amoeba 中撤销权能的方式。
20. 限制会话密钥的生存期是否有意义?如果有,给出一个说明其建立方式的实例。
21. 图 8.38 中消息 6 中时间戳的任务是什么,为什么需要对其进行加密?
22. 通过添加 Alice 和 Bob 之间的为身份验证而进行的通信来完成图 8.38。
23. 考虑一下在 SESAME 中 Alice 和一个身份验证服务程序 AS 之间的通信。在消息 $m_1 = K_{AS}^+(K_{A,AS}(data))$ 和 $m_2 = K_{A,AS}(K_{AS}^+(data))$ 之间如果存在差别,那么差别是什么?
24. 至少定义两种不同级别的电子付费系统中的事务原子性。
25. e-cash 系统中的一名顾客在使用从银行取出的货币前最好应该等待一段时间,为什么?
26. 在任何付费系统中通常都禁止商家匿名,为什么?
27. 考虑一个电子付费系统,其中顾客向(远程)商家发送货款。给出一个类似图 8.44 和图 8.45 中所使用的表格来表示信息隐藏的情况。

第9章 基于对象的分布式系统

从本章开始,我们将从对原理的讨论转向检验各种用于组织分布式系统的范型。第一个范型由分布式对象组成。在基于对象的分布式系统中,对象的概念在分布式实现中起关键性的作用。从原理上讲,所有东西都被作为对象处理,而客户将以调用对象的方式获得服务和资源。

分布式对象之所以成为重要的范型,是因为它相对比较容易把分布的特性隐藏在对象接口后面。此外,因为对象实际上可以是任何事物,所以它也是创建系统的有力范型。分布式对象组成了两种重要而广泛应用的分布式系统的基础,本章将讨论这两种系统。

作为第一个例子,我们将研究 CORBA,一种分布式系统的行业标准。现在,已经有很多 CORBA 系统在使用,而它的发展和标准化正随着其安装数量的增长而不断进展。

第二个例子是 Microsoft 的 DCOM。DCOM 可以视为一种中间件,能够在 Windows 95 之后的各种 Windows 操作系统上实现。事实上,所有的 Windows 应用程序都使用 DCOM 提供的功能。因此,可以说 DCOM 可能是目前与分布式系统相关的使用最广泛的中间件。

除了这两种商业系统,还有各种建立基于对象的分布式系统的研究成果。本章中将讨论一种被称为 Globe 的实验系统。作为大规模广域分布式系统研究项目的一部分,它正在开发当中。使 Globe 区别于 CORBA 和 DCOM 的有趣特点是:Globe 分布式对象的状态可以在多台机器间分割和复制。

我们面临的一个困难是需要组织我们的讨论,使它能够比较不同的系统。每种系统将在一个单独的小节中讨论。首先是对系统重要概念的概述,特别是对象模型和提供的常规服务。然后逐一讨论以下 7 个方面是如何实现的:通信、进程、命名、同步、一致性与复制、容错性和安全性。

这种结构的优点是可以详细地比较在各部分提到的不同系统。但另一方面,因为关于各个方面的通用内容已经在本书第一部分讨论过,剩下的只是一些细节描述。然而,这些细节将帮助理解每种范例系统实际上如何工作,与其他系统相比如何。

有很多论述基于对象的分布式系统的通用教材。Orfali 等(1996)给出了有关 CORBA 和 DCOM 的易读的概述。Emmerich(2000)集中了大量关于分布式对象的原理,也用 CORBA、DCOM 和 Java RMI 作为例子。文献(Islam, 1996)中则讨论了分布式对象技术如何用于灵活地创建系统软件。

9.1 CORBA

对基于对象的分布式系统的学习从公共对象请求代理体系结构(common object request broker architecture,CORBA)开始。正如其名称所示,CORBA 与其说是一种分

布式系统,不如说是一种分布式的规范。这些规范由对象管理小组(object management group,OMG)发布。OMG是一个有着超过800名成员的非赢利性组织,其成员主要来自行业内。OMG制定CORBA的重要目标是:定义一个能够克服集成网络应用中大多数协同问题的分布式系统。最早的CORBA标准于九十年代初发布。目前,实现CORBA 2.4版本的系统广泛应用,而CORBA 3版本的第一批系统正在实用化。

像许多其他的委员会工作的成果一样,CORBA有着大量的特色和实用工具。核心规范有700多页,另1200页用于说明建立在核心之上的各种服务。自然,每个CORBA的实现都会有它自己的扩展,因为总有一些东西是开发商认为必需而规范中又没有包括的。CORBA再次证明了建立一个简单的分布式系统可能会是一件有些超负荷的困难工作。

下面的内容中不会涉及CORBA提供的所有东西,我们将集中讨论它作为一个分布式的本质内容和它区别于其他基于对象的分布式的部分。CORBA规范可以在(OMG 2001b)中找到,在<http://www.omg.org>上供公众访问。文献(Vinoski 1997)给出了CORBA的高度概述,而Pope(1998)则提供了从原始规范得来的详细描述。有关用C++建立CORBA应用程序的信息可以在文献(Baker 1997)和(Henning, Vinoski 1999)中找到。

9.1.1 CORBA 概述

CORBA的全局体系结构坚持了OMG在(OMG 1997)中提出的参考模型。这个参考模型如图9.1所示,由连接到被称为对象请求代理(object request broker,ORB)的4组体系结构元件构成。ORB组成了所有CORBA分布式的系统的中心;它负责实现对象与客户间的通信,同时隐藏与分布式和异质有关的问题。在许多系统中,ORB以库的形式实现,被连接到客户/服务器应用程序,提供基本的通信服务。下面回到ORB,讨论CORBA的对象模型。

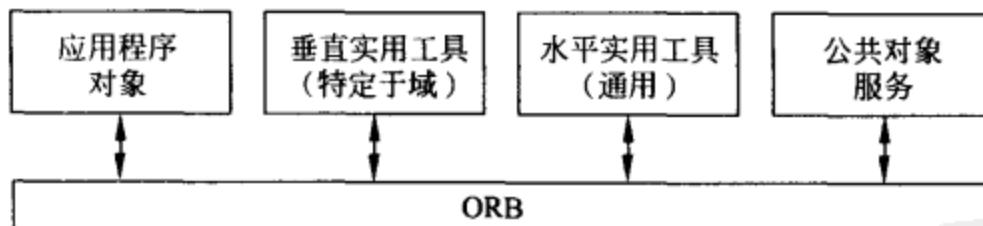


图9.1 CORBA的全局体系结构

除了为特定应用建立的对象外,参考模型还区别了所谓的CORBA实用工具(CORBA facility)。实用工具作为CORBA服务(后面将讨论)的组件,可以分为不同的两组。水平实用工具(horizontal facility)由与应用领域无关的高层通用服务组成。这些服务目前包括用户接口、信息管理、系统管理和任务管理(用于定义工作流系统)。垂直实用工具(vertical facility)由以特定应用领域,比如电子商务、银行业、制造业等为目标的高层服务组成。

我们对应用对象和CORBA实用工具将不做详细讨论,而把注意力集中在基础服务

和 ORB 上。

1. 对象模型

CORBA 使用我们在第 2 章中讨论过的远程对象模型。在这一模型中，对象的实例存在于服务器的地址空间中。可以很有趣地发现，CORBA 规范从没有明确地规定对象必须以远程对象的形式实现。然而，事实上所有的 CORBA 系统都只支持这一模型。另外，规范中经常暗示分布式对象实际上就是远程对象。在稍后讨论 Globe 对象模型时，将展示一个完全不同的对象模型如何在原理上能够同样好地得到了 CORBA 的支持。

对象和服务在 CORBA 接口定义语言 IDL 中说明。CORBA IDL 与其他的接口定义语言类似，它为表示方法及其参数提供了精确的语法。用 CORBA IDL 描述语义是不可能的。接口就是方法和对象的组合，而对象指明方法实现哪一接口。

接口规范只能通过 IDL 给出。稍后将看到，在诸如分布式 COM 和 Globe 的系统中，接口在一个较低的层次上以表格的形式说明。这些所谓的二进制接口 (binary interface) 自然地独立于任何编程语言。然而在 CORBA 中，需要为 IDL 规范到现有编程语言的映射提供准确的规则。目前，已经为很多语言提供了上述规则，其中包括 C、C++、Java、Smalltalk、Ada 和 COBOL。

CORBA 是以一组客户和对象服务器的形式组织的，CORBA 系统的一般结构如图 9.2 所示。

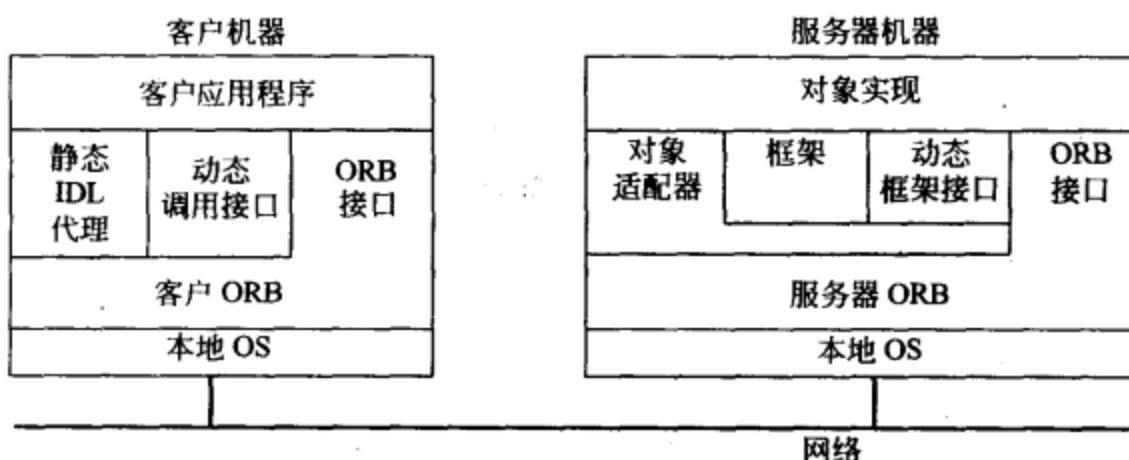


图 9.2 CORBA 系统的一般结构

在任何一个 CORBA 进程，不管是客户还是服务器的底层都是 ORB。ORB 最好被视为是负责处理客户与对象之间基本通信的运行时系统。基本通信主要包括调用请求被传送到对象服务器而响应被传回客户。

从进程的角度看，ORB 本身只提供了很少的一些服务。其中之一是处理对象引用。这种引用依赖于某一特定的 ORB。因此，ORB 提供对象引用的编组与解编操作以便在进程之间交换，同时也提供比较引用的操作。对象引用将在后面详细讨论。

ORB 提供的其他操作负责在初始化时找到该进程可用的服务。通常，它提供一种方法，可以初始化引用实现某一 CORBA 服务的对象。例如，为了使用命名服务，进程必须知道如何引用此服务。这一初始化过程对其他服务也同样适用。

除了 ORB 接口,客户与服务器几乎看不到 ORB 的任何东西。相反,他们通常只看到处理特定对象方法调用的存根。客户应用程序通常有一个可用的代理,为它使用的每一个对象实现一个相同的接口。正如在第 2 章中解释的,代理是客户端的存根,仅仅用于对调用请求进行编组并将其发送到服务器。服务器的响应则被解编并传回客户。

需要注意,代理与 ORB 之间的接口不必标准化。因为 CORBA 假设所有的接口都用 IDL 给出,CORBA 的实现为开发者提供 IDL 编译器,用于生成处理客户和服务器 ORB 之间通信的必要代码。

然而,有些时候静态定义的接口对客户是不可用的。相反,需要的是找到运行时特定对象的接口是什么样子,然后为此对象生成调用请求。为实现这一目的,CORBA 提供了动态调用接口 DII,允许客户在运行时生成调用请求。本质上说,DII 提供了通用的调用操作,以对象调用、方法标识符和一系列输入变量作为参数,而通过调用者提供的一系列输出变量返回结果。

对象服务器的组织形式如第 3 章中所述。如图 9.2 所示,CORBA 系统提供对象适配器,用于把输入的请求送至适当的对象。在服务器端,真正的解编工作通过存根来完成,在 CORBA 中称之为骨架。但对象的实例也可以完成解编。对客户来说,服务器端的存根可以根据 IDL 规范静态编译,也可以通过动态骨架的形式获得。使用动态骨架时,对象必须提供调用函数的适当实现,就像提供给客户的那样。下面继续讨论对象服务器。

2. 接口与实现仓库

对于动态创建调用请求来说,重要的是进程在运行时能找到的接口是什么样子的。CORBA 提供了接口仓库(interface repository),存储所有的接口定义。在许多系统中,接口仓库用一个单独的进程来实现。此进程提供标准接口,用于存储和检索接口定义。接口仓库也可以看作 CORBA 的一部分,辅助运行时类型检查工具。

接口定义不论何时编译,IDL 编译器都会为其分配一个仓库标识符(repository identifier)。此仓库标识符是从仓库中检索接口定义的基本方法。在默认的情况下,标识符从接口名和其方法中得到,也就是说不能保证它的惟一性。如果要求标识符惟一,默认设置可以被修改。

既然所有存储在接口仓库中接口定义都符合 IDL 语法,以标准的方式组织所有的定义就成为可能。(在数据库术语中,这意味着每一个接口仓库的概念模式都是相同的。)这样一来,CORBA 系统中的接口仓库为定位接口定义提供了相同的操作。

除了接口仓库,CORBA 系统通常还提供一个实现仓库(implementation repository)。从概念上讲,实现仓库应含有实现和激活对象需要的所有东西。因为这些功能是与 ORB 本身和底层的操作系统有密切关系,所以提供标准的实现是很困难的。

实现仓库也是与对象服务器的组织和实现紧密联系的。正如在第 3 章中解释的,对象适配器负责激活对象。它保证对象在服务器的地址空间中运行,从而可以调用它的方法。给出对象引用后,适配器就可以连接到实现仓库,找出究竟需要做什么。

例如,实现仓库可以维护一张表,说明为特定的对象应启动一个新的服务器,该服务

器应监听哪个端口。此外,还可以提供更多的信息,如该服务应加载、执行哪个可执行文件(如二进制代码)。

或者,可能不需要再启动一个独立服务器,仅需要当前服务器链接到一个含有指定方法或对象的特定库。这些信息应存储在实现仓库中。上述两个例子说明了这样的一个仓库确实是与 ORB 及其运行平台紧密相关的。

3. CORBA 服务

CORBA 参考模型的一个重要组成部分是 CORBA 服务。CORBA 服务最好被视为通用而独立的应用。因而,CORBA 服务非常类似于通常由操作系统提供的服务。如图 9.3,列出了 CORBA 服务的完整列表。可惜的是,通常不可能在不同的服务间划分明确的界限,因为其功能总是有重叠的。下面简明地描述一下每项服务,以便后面更好地与 DCOM 和 Globe 提供的服务做比较。

服务	描述
集合	把对象分组为列表、队列、集合等的工具
查询	以说明方式查询对象集合的工具
并行	允许并行访问共享对象的工具
事务	在多个对象的方法调用上的平面事务与嵌套事务
事件	事件之间异步通信的工具
通知	基于事件的异步通信的高级工具
外部化	编组与解编对象的工具
生命周期	创建、终止、复制和移动对象的工具
许可	为对象附加许可证的工具
命名	系统范围的对象命名工具
属性	为对象连接(属性,值)对的工具
交易	公布和查找对象提供的服务的工具
持久性	持久地存储对象的工具
关系	表达对象间关系的工具
安全	安全通道、授权和审核机制
时间	提供特定误差范围内的当前时间

图 9.3 CORBA 服务概观

集合服务(collection service)提供了把对象分组为表、队列、栈或集合等的方法。根据分组性质的不同,可提供各种访问机制。比如,可以用通常称为“迭代”的方法按元素来审查表。还有按关键值选择对象的工具。从某种意义上说,集合服务与面向对象编程语言通常提供的类库非常接近。

还有独立的查询服务(query service),使用说明性查询语言提供了建立可查询对象集的方法。查询可能返回指向对象的引用或对象的集合。查询服务以更加高级的查询扩

充了集合服务。它与集合服务的不同之处在于,后者提供了多种集合类型。

并行控制服务(concurrency control service),它提供先进的锁定机制,客户通过它可以访问共享对象。这一服务可用于实现事务服务(transaction service),它由另一独立服务提供。事务服务允许客户在一次事务处理中定义一系列对多个对象的方法调用。此服务支持平面事务和嵌套事务。

通常,客户调用对象的方法并等待调用的结果。为支持异步通信,CORBA 支持事件服务(event service),使客户和对象可以被特定事件的发生所中断。异步通信的高级工具由另一独立的通知服务(notification service)提供。后文将更详细地描述这些服务。

外部化(externalization)服务处理编组对象,使它们可以存储在磁盘上或通过网络发送。它可以与 Java 提供的串行化工具相比,允许对象以字节串的形式向数据流写入。

生命周期服务(life cycle service)提供创建、终止、复制和移动对象的方法。关键概念之一是工厂对象(factory object),即用于创建其他对象的特殊对象(Gamma 等 1994)。实践证明,只有对象的创建需要由单独的服务来处理,而终止、复制和移动对象通常由对象本身来方便地定义。原因是这些操作通常以各个对象特有的方式影响其状态。

许可服务(licensing service)允许对象的开发者在对象上附加许可证并执行特定的许可策略。许可证表明了客户使用对象的权限。比如,某一对象的许可证可能只允许该对象同时被一个客户使用。另一许可证可能保证该对象在一定的期限后自动失效。

CORBA 提供独立的命名服务(naming service),通过这一服务,对象可以获得映射到对象标识符的可读的名称。描述对象的基本工具由单独的属性服务(property service)提供。此服务允许客户把(属性,值)对与对象相关联。注意,这些属性不是对象状态的一部分,而是用于描述对象的。换句话说,它们不是对象一部分而仅是提供相关信息。与这两项服务有关的是交易服务(trading service),允许对象宣传他们能提供什么(通过其接口),并允许客户使用支持约束描述的特殊语言来查找服务。

独立的持久性服务(persistence service)提供了以存储对象的形式在磁盘上存储信息的工具。这里很重要的是持久性服务的透明性。客户不需要显式地在磁盘和可用主内存之间传输存储对象的数据。

到现在为止,还没有服务提供联系两个和多个对象的工具。这些工具由关系服务(relationship service)提供,此服务本质上支持按概念模式组织对象,正如在数据库中使用的那样。

安全服务(security service)提供安全。这一服务的实现类似于 SESAME 和 Kerberos 之类的安全系统。CORBA 安全服务提供身份验证、授权、审计、安全通信、非否认和管理工具。后面的内容将详细讨论安全。

最后一点的是,CORBA 提供时间服务(time service),返回特定误差范围内的当前时间。

如 Pope(1998)解释的,CORBA 服务以 CORBA 对象模型为基础来设计。这意味着所有的服务都是用 CORBA IDL 说明的,而接口规范和实现之间是相互独立的。另一重要的设计准则是服务应当最小化、简单化。在下面的部分中将更详细地讨论这些服务。从那些描述中可以看到,最后一条准则在什么范围内得以成功实现。

9.1.2 通信

最初,CORBA 有一个简单的通信模型:客户调用对象的方法并等待回答。人们认为这个模型太简单了,很快就添加了附加的通信工具。下面,将较为详细地讨论 CORBA 中的调用工具,并考虑这些对象调用的选择。正如我们将要看到的,对基本对象调用模型的扩展是由异步通信的需要推动的。这一推动同时也引出了我们在第 2 章中讨论的替代的消息传递模型。

1. 对象调用模型

在默认的情况下,当客户调用对象时,它发送一个请求给对应的对象服务器,然后等待直到接收到响应。在没有失败的情况下,当调用者与被调用者在相同的地址空间时,这些语义准确地符合正常的方法调用。

然而,有失败出现时,事情会多少有些复杂。在同步调用的情况下,如刚才描述的,客户最终将收到标志调用没有完全完成的异常。CORBA 指定,在这种情况下,调用应遵循“至多一次”(at-most-once)的原则,即要调用的方法可能被调用了一次或根本没有被调用。需要注意,这一原则应由实现版本提供。

因此,同步调用在客户需要应答时是很有用的。如果返回了适当的响应,CORBA 保证此方法只被调用了一次。然而,在不需要响应的情况下,更好的做法是客户调用方法,然后尽快继续执行本身的操作。这种类型的调用与第 2 章中讨论的异步 RPC 很相似。

在 CORBA 中,这种形式的异步调用称为单向请求(one-way request)。方法只有在不返回结果时才能声明为单向的。但是,与保证传输的异步 RPC 不同,CORBA 中的单向请求只提供尽力传输的服务。换句话说,不向调用者保证调用请求一定会传输到对象服务器。

除了单向请求,CORBA 还支持所谓的延迟同步请求(deferred synchronous request)。这种请求实际上是单向请求与服务器异步向客户送回结果的结合。客户向服务器发送请求后,立即继续工作而不等待服务器的响应。换句话说,客户并不确定地知道它的请求是否已经真正传输到了服务器。

图 9.4 总结了这三种不同的调用模型。

请求类型	失败语义	描述
同步	至多一次	调用者等待,直到返回响应或抛出异常
单向	尽力传输	调用者立即继续,不等待服务器的任何响应
延迟同步	至多一次	调用者立即继续,但稍后可等待响应的传输

图 9.4 CORBA 支持的调用模型

2. 事件和通知服务

尽管 CORBA 提供的调用模型在正常情况下应该已经满足了基于对象的分布式系统中绝大部分的通信要求,但人们仍感到只有方法调用是不够的。特别是,需要一种只是简

单地标记事件发生的服务,与该事件有关的客户可以采取合适的行动。

结果是定义了事件服务(event service)。CORBA 中的基本事件模型相当简单。每一事件与一个数据项相关联,通常是用对象引用或者应用程序特定值的方法表示。事件由生产者(supplier)产生,由消费者(consumer)接受,通过事件通道(event channel)传输。事件通道是位于生产者和消费者之间的逻辑通道,如图 9.5 所示。

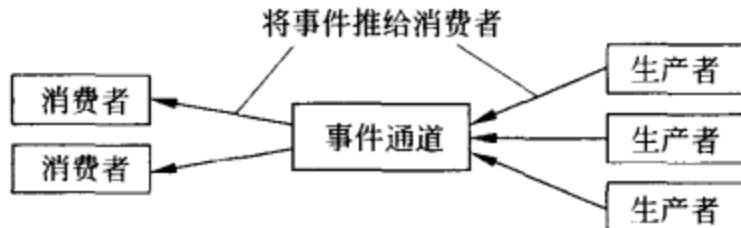


图 9.5 按照推式模型建立的事件生产者和消费者的逻辑组织

图 9.5 显示了所谓的推式模型(push model)。在这种模型中,不论事件何时发生,生产者产生事件,并把它通过事件通道推出,并传送到其消费者。推式模型与大多数人处理事件的异步行为非常接近。结果,消费者被动地等待事件传播,并在事件发生时以某种方式被中断。

CORBA 支持的另一种可选模型是图 9.6 所示的拉式模型(pull model)。在这种模型中,消费者轮询事件通道以检查事件是否发生。而事件通道则顺序轮询不同的生产者。

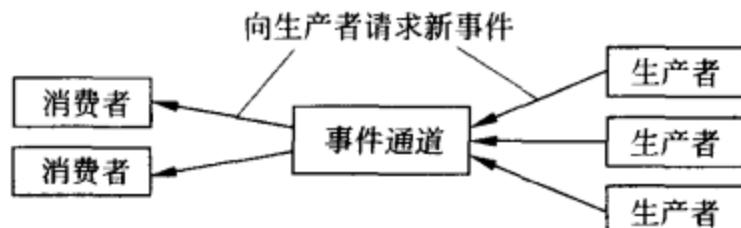


图 9.6 CORBA 中事件传输的拉式模型

虽然事件服务提供了事件传播的简单而直截了当的办法,但是它存在很多严重的缺陷。为了传播事件,生产者和消费者必须连接到事件通道。这也意味着如果消费者在事件发生之后才连接到事件通道,事件将丢失。换句话说,CORBA 的事件服务不支持事件的持续性。

更为严重的是消费者没有过滤事件的办法。从原理上说,每个事件都被送到所有的消费者。如果要区别不同的事件,就得为每一事件类型建立一条独立的事件通道。在被称为通知服务(notification service)(OMG,2000a)的扩展中添加了过滤能力。另外,这一服务还提供工具,在没有消费者对特定事件感兴趣时防止事件传播。

最后要指出,事件传播本质上是不可靠的。CORBA 规范声明对事件传输不需要给出保证。如我们将在第 12 章中讨论的,对有些应用程序来说,可靠的事件传播非常重要。这些应用程序不应当使用 CORBA 的事件服务,而应该采用其他的通信手段。

3. 消息

到现在为止,我们描述的 CORBA 中的通信都是暂时的。也就是说,只有当发送者和接收者都在执行时,消息才会被底层通信系统存储。如第 2 章中讨论的,有许多应用程序需要持久的通信,以便消息一直存储到能够传输。有了持久的通信,消息送出后发送者和接收者是否还在执行就没有关系了。在任何情况下,消息都会被存储足够长的时间。

持久通信的著名模型之一是消息传递模型(messaging model)。CORBA 支持此模型,作为附加的消息服务。CORBA 中的消息处理机制与其他系统的不同之处在于它内在的基于对象的方法。特别是,消息服务的设计者需要保留通过调用对象发生的所有通信的模型。在消息处理方面,这一设计约束造成了两种形式的异步方法调用。

在回调模型(callback model)中,客户提供一个对象来实现含有回调方法的接口。这些方法可被底层的通信系统调用,以便传送异步调用的结果。设计要点之一是异步方法调用不应影响对象原始的实现。换句话说,客户应负责把原始的同步调用转换成异步调用,而服务器给出一般(同步)的调用请求。

建立异步调用通过两步完成。首先,由对象实现的原始接口被两个新的接口替代,新接口仅由客户端软件实现。一个接口包含客户可以调用的方法说明。这些方法都不返回值,也不含有输出参数。第二个接口是回调接口。对于原始接口中每一个操作,它都含有一个由客户 ORB 调用的方法,以便在客户调用时传输相关方法的结果。

例如,考虑实现只有一个方法的简单接口的对象:

```
int add(in int i, in int j, out int k)
```

假设此方法(用 CORBA IDL 表达)接受两个非负整数 *i* 和 *j*,并用输出参数 *k* 返回 *i+j* 的值。若操作不成功则返回 -1。把原始(同步的)方法调用转换成异步调用可以通过以下步骤实现:首先生成下列一对方法说明(为示例的目的,选用了方便的名称而没有遵守 OMG, 2001b 中的严格规则):

```
void sendcb_add(in int i, in int j);           //由客户调用  
void replych_add(in int ret_val, in int k);      //由客户的 ORB 调用
```

结果,原始方法说明中所有的输出参数都被从客户调用的方法中删去,其值以回调操作的输入参数的形式返回。同样,如果原始方法声明返回值,此值也按输入参数传递给回调操作。

第二步只需编译已生成的接口。结果,客户得到一个存根,允许它异步调用 *sendcb_add*。然而,客户需要为回调接口提供实现,在本例中包括 *replycb_add* 方法。注意,这些更改不影响服务器端的对象实现。仍用此例,图 9.7 总结了回调模型。

作为另一种选择,CORBA 提供了轮询模型(polling model)。在这种模型中,客户可使用一系列操作轮询其 ORB 以得到到来的结果。和回调模型中一样,客户负责把原始的同步方法调用转换成异步调用。而且,通过从对象实现的原始接口中自动导出适当的方法说明,可以完成大多数工作。

回到我们的例子中,方法 *add* 可得到如下两个生成方法说明(再次采用了方便的

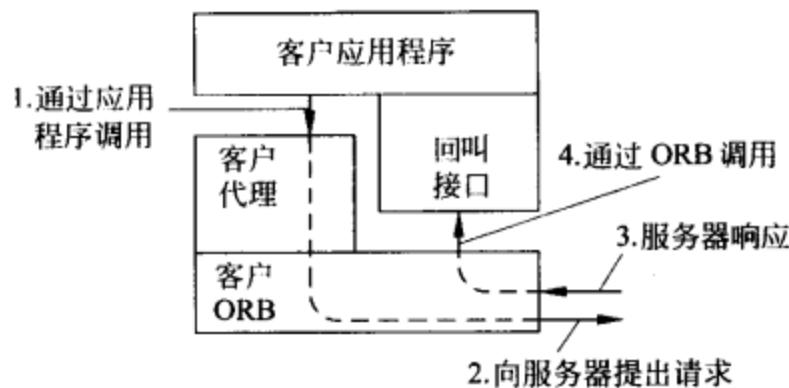


图 9.7 CORBA 异步方法调用的回叫模型

名称):

```
void sendpoll add(in int i, in int j);           // 由客户调用
void repypoll_add(out int ret_val, out int k);    // 也是由客户调用
```

与回叫模型最重要的区别是 repypoll_add 方法需要由客户的 ORB 实现。此实现可由 IDL 编译器自动生成。图 9.8 总结了轮询模型。注意,对象的原始实现在服务器端不必更改。

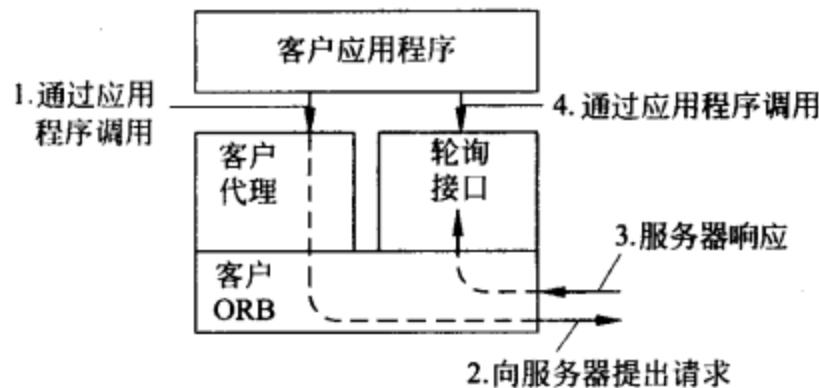


图 9.8 CORBA 异步方法调用的轮询模型

到现在为止讨论过的模型中,还缺少一些内容。那就是客户与服务器间传送的消息,包括对异步调用的响应,在客户或服务器没有运行的情况下都被底层系统存储了起来。幸运的是,与这些持久通信相关的绝大多数问题并不影响已经讨论过的异步调用模型。需要的是建立一组消息服务器,可使消息(调用请求或响应)暂存到能够传输为止。(OMG 2001b)中建议的方法与 IBM 的消息队列系统很相似。这一内容已在 2.4.4 节中讨论过,此处不再重复。

4. 互操作性

早期的 CORBA 版本把很多问题都留给了实际的实现。结果是不同制造商的 CORBA 系统都有各自的实现客户与对象服务器通信的方法。特别是,只有当客户和对象服务器使用相同的 ORB 时,客户才能调用服务器的对象。

这一缺乏互操作性的问题随着标准 ORB 间协议和统一对象引用方法的引入而得到解决。回到图 9.2,这意味着客户和服务器间的通信遵守标准协议,在 CORBA 中也被称

为通用 ORB 间协议(GIOP)。

GIOP 实际上是一个协议框架。它假设其真正的实现在一个已存在的传输协议上执行。这个传输协议应该是可靠的、面向连接的，并能提供比特流的概念和其他一些特点。不奇怪，TCP 满足这些要求，而其他一些传输协议也满足。运行于 TCP 之上的 GIOP 实现称为互联网 ORB 间协议，或简称为 IIOP。

GIOP(IIOP 或其他的 GIOP 实现)接受 8 种不同的消息类型，如图 9.9 所示。最重要的两种消息是请求(Request)和响应(Reply)，二者结合起来组成了实际的远程方法调用实现的一部分。

请求消息包含完整的编组调用请求，包括对象引用、调用方法的名称和所有必需的输入参数。对象引用和方法名是消息报头的一部分。每个请求消息还要有本身的请求标识符，用于稍后匹配相应的回应。

响应消息包含编组的返回值和与先前调用的方法相关的输出参数。不必明确地指明对象或方法，只要简单地返回与相应的请求消息相同的请求标识符就足够了。

消息类型	发起者	描述
请求(Request)	客户	含有调用请求
响应(Reply)	服务器	含有对调用的响应
定位请求(Locate Request)	客户	含有对对象准确定位的请求
定位响应(Locate Reply)	服务器	含有对象定位信息
取消请求(Cancel Request)	客户	说明客户不再需要响应
关闭连接(Close Connection)	二者均可	说明连接将关闭
消息错误(Message Error)	二者均可	含有错误信息
片段(Fragment)	二者均可	大消息的一部分(片断)

图 9.9 GIOP 消息类型

我们将在下面讨论，客户可以向实现仓库发出请求，查找特定对象在何处的细节。这一类请求通过定位请求消息发送。实现仓库以定位响应消息响应，通常说明调用请求应送到的当前对象服务器。

当客户想要取消先前发送的请求或定位请求消息时，可以向服务器发送取消请求消息。取消请求意味着客户不再等待服务器的响应。客户取消请求的原因可能多种多样，但通常是由于客户的应用程序超时。很重要的一点是，取消请求并不意味着相关的请求不会被执行。这种情况需要客户应用程序来处理。

在 GIOP 中，总是由客户建立到服务器的连接。服务器只能接受或拒绝连接请求，并不独立建立到客户的连接。然而，客户和服务器都可以关闭连接，只需向通信的另一方发送关闭连接消息即可。

如果出现失败，一方将发送消息错误类型的消息通知另一方。此类消息含有引起失败的 GIOP 消息的报头信息。(这一方法类似于 Internet 协议中用于返回错误信息的 ICMP 消息。在出错的情况下，引起错误的 IP 包报头被作为 ICMP 消息的数据传送。)

最后，GIOP 允许各种请求和回应被分段。这样，需要在客户和服务器之间传输大量

数据的调用请求可以很容易地得到支持。片段以特殊的片断消息的形式传输,通过该消息可确定原始消息并允许在接收端重新组装消息。

9.1.3 进程

CORBA 区别两种类型的进程:客户和服务器。CORBA 的重要设计目标之一是使客户尽可能简单。潜在的想法是使应用程序开发者易于利用服务器上已存在的服务。

另一方面,服务器在最初就将问题留给了各种不同的实现,仅仅以基本对象适配器的形式提供了最少的支持。不幸的是,这种最少的支持引起了可移植性问题,目前已经通过给出对象适配器提供服务的更加完善的规范而加以解决。下面,将详细讨论 CORBA 中客户端和服务器端的软件。

1. 客户

如前所述,CORBA 的客户端软件保持最小。对象的 IDL 规范被简单地编译成一个代理,负责把调用请求编组成 HOP 请求消息之类的消息,并把相应的响应消息解编成可以传回调用客户的结果。

CORBA 中的代理没有其他任务,只负责把客户应用程序连接到底层的 ORB。除了生成对象特定的代理,客户也可以通过 DII 动态调用对象。

这一方法的结果是如果对象需要一个特定的客户端接口实现,它就得告诉开发者使用一个能够生成该软件的 IDL 编译器,或者自己提供客户代理。比如,一个对象实现可能会带有一整套用于实现特定对象客户端缓存策略的代理。显然,后一种做法完全违背了 CORBA 可移植性和透明分布性的目标。

另一种方法是忽略与对象相关的问题而依赖于能提供必须支持的客户端 ORB。例如,与提供包含于客户代理中的缓存相反,对象实现可以假定缓存工作由客户的 ORB 以通用方式处理。但很清楚的是,这样的方法有其内在的局限。

需要的是一种机制,使 IDL 编译器生成的代理与已存在的客户端 ORB 协同工作,并仍然能在需要的时候修改客户端软件。CORBA 解决这个问题的方法是使用截取程序 (interceptor)。和它的名字一样,截取程序就是一种截取从客户到服务器的调用,必要时做一定的修改的机制。本质上,截取程序是一段代码,修改从客户到服务器的调用请求,相应地也修改相关的响应。有各种各样的截取程序添加到 ORB。哪一个真正被激活取决于在调用请求中所引用的对象或服务器。

CORBA 中的截取程序可以设置在一或两个逻辑层次上,如图 9.10 所示。请求层截取程序 (request_level interceptor) 逻辑上位于客户代理和 ORB 之间。调用请求传递给 ORB 之前,它首先要经过截取,并有可能被修改。在服务器端,请求层截取程序位于 ORB 和对象适配器之间。

相反,消息层截取程序 (message-level interceptor) 位于 ORB 和底层的网络之间。消息层截取程序对要发出的消息一无所知,它只处理可能修改的 GIOP 消息。消息层截取程序的典型例子是在发送端实现分段而在接收端重组原始的 GIOP 消息,比如使用特殊的片段消息。

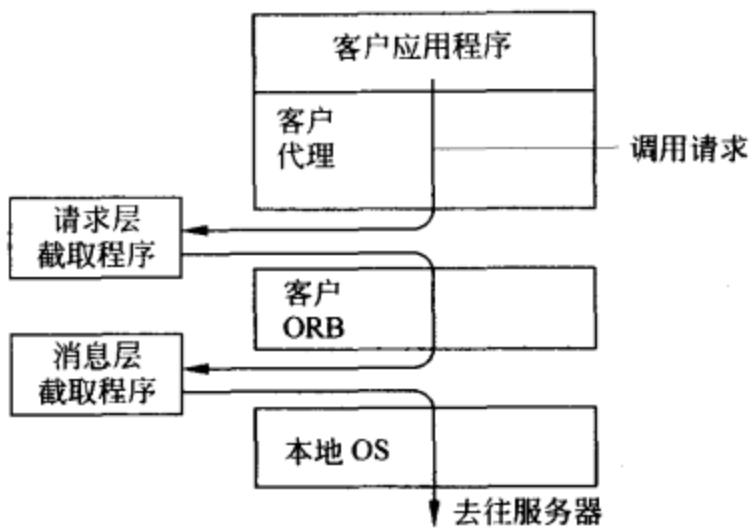


图 9.10 CORBA 中截取程序的逻辑位置

截取程序只对 ORB 是可见的,也就是说,由 ORB 负责调用截取程序。客户和服务器几乎见不到截取程序,除了客户被绑定到服务器时。注意,ORB 可能同时使用两种类型的截取程序。CORBA 中截取程序使用的概述可在(Narasimhan 等 1999)中找到,而详细的规范在(OMG 2001b)中给出。

尽管使用截取程序的主意乍一看很有吸引力,但也可以说它是侵入到执行中的一种机制,用于修补明显丢失了的东西。比如,假设 CORBA 提供了支持开发和使用特定对象代理的机制,对截取程序的需求就大大减少。然而,真正需要的是易于扩展的 ORB。截取程序提供了通用机制以支持扩展性,但问题是 CORBA 中提供的两种类型的截取程序究竟是不是我们需要的。从软件体系结构的观点讨论截取程序的更多内容可以在(Schmidt 等 2000)中找到。

2. 可移植对象适配器

第 3 章中详细讨论了对象适配器的概念。对象适配器是为一组对象实现特定激活策略的机制。例如,为了实现方法调用,一个适配器可能为每一调用使用一个独立线程,而另一个则为其管理的所有对象使用一个线程。

一般而言,对象适配器不仅仅调用对象的方法。正如其名字,对象适配器负责提供对象的稳定映像,它改编程序使其对客户而言是一个对象。适配器也被称为包装(wrapper)。

在 CORBA 中,可移植对象适配器(portable object adapter,POA)是负责把服务器端代码以 CORBA 对象的形式呈现给客户的组件。POA 的定义方式使得服务器端代码的编写可以独立于特定的 ORB。

为了支持在不同 ORB 之间的可移植性,CORBA 假定对象实现部分由被称为 servant 的组件提供的。servant 是对象中实现客户可调用方法的那一部分。servant 必须是依赖于编程语言的。例如,用 C++ 或 Java 实现 servant 的典型做法是给出类的实例。另一方面,用 C 或其他任何一种过程型语言写的 servant 的典型组成是表示对象状态的数据结构以及对它的一系列函数操作。

POA 是如何使用 servant 建立 CORBA 对象的映像的呢?第一步,每个 POA 都提供

以 CORBA IDL 描述的如下操作：

```
Objectid activate_object(in Servant p_servant);
```

这一操作以一个 servant 的指针作为输入参数, 返回一个 CORBA 对象标识符作为结果。Servant(公务员)的类型没有统一的定义; 而是被映射为依赖于语言的数据类型。例如, 在 C++ 中, Servant 被映射为预定义类 ServantBase 的指针。这个类包含了大量的 C++ Servant 需要实现的方法定义。

activate_object 操作返回的对象标识符是由 POA 生成的。它用作 POA 的活动对象映射的索引, 指向 servant, 如图 9.11(a)所示。在这种情况下, POA 为它支持的每一个对象执行一个应用程序。更具体地说, 假设应用程序开发人员写了一个 ServantBase 子类, 称为 My_Servant。作为 My_Servant 类的实例的 C++ 对象可以变成 CORBA 对象, 如图 9.12 所示。

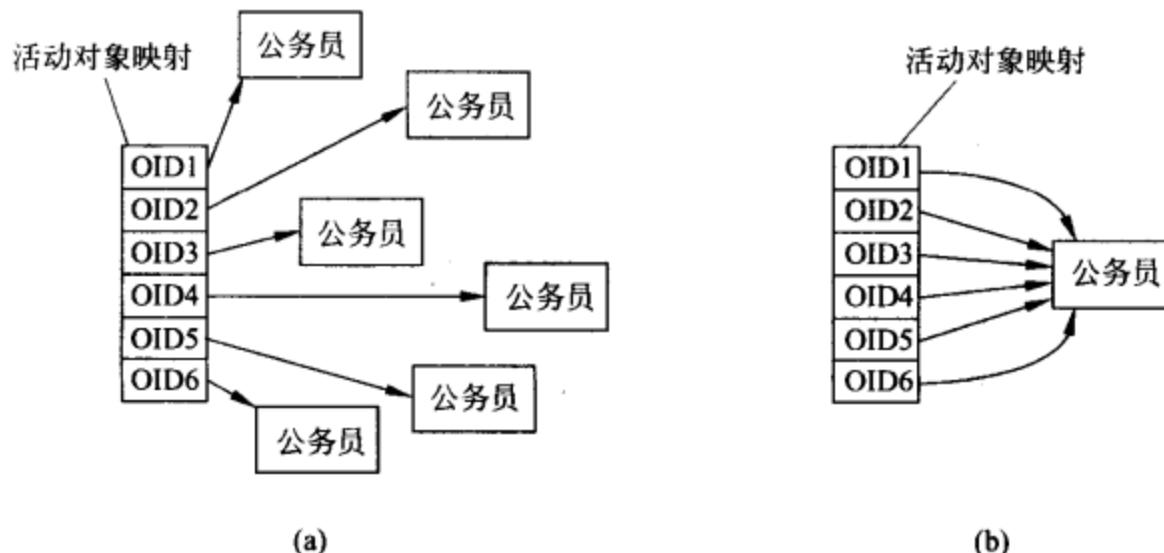


图 9.11 CORBA 对象标识符到 servant 的映射

(a) POA 支持多 servant; (b) POA 支持单一 servant

```
My_Servant * my-object; // 声明一个 C++ 对象  
CORBA::Objectid_var oid; // 声明一个 CORBA 标识符  
my_object = new My_Servant; // 创建 C++ 对象  
// 把 C++ 对象注册为 CORBA 对象  
oid = poa->activate_object(my_object);
```

图 9.12 把 C++ 对象变成 CORBA 对象

在图 9.12 的代码中,首先声明了对 C++ 对象的引用。为了创建 CORBA 标识符, 使用了 CORBA::Objectid_var, 在所有标准的 CORBA 的 C++ 版本中都会预定义的 C++ 数据类型。在这些声明之后, my_object 可以被实例化为真正的 C++ 对象。在 CORBA 术语中, 此 C++ 对象对应于一个 servant。把 C++ 对象变成 CORBA 对象通过在 POA (假定可以通过变量 poa 访问) 中注册它得到实现。注册返回 CORBA 标识符。

需要注意的是,如果又一个 My_Servant 类型的对象被创建,那么此 C++ 对象在 POA 的注册将带来几乎相同的 servant,虽然是在不同的状态下操作。在 POA 要支持从相同的类定义派生出的多个对象的情况下,注册单一的 servant 并仅用状态区别不同对

象的做法更有效率。这一原理如图 9.11(b) 所示,每一对象标识符都指向同一个 servant。在这种情况下,不论对象何时被调用,对象标识符将(隐性地)传递给 servant,以便它只对与所标识对象惟一相关的数据进行操作。

这一例子还说明了另一个重要的问题: CORBA 对象标识符是与 POA 惟一相关的。servant 不论何时注册到 POA,POA 都为它返回一个对象标识符。另一个我们没有展示的做法是应用程序开发人员先生成一个标识符再传递给 POA。然而,在两种情况下,这一标识符都会封装成一个更大的数据结构,起到系统范围的对象引用的作用,同时也标识 POA 和 POA 所在的服务器。

不论 POA 支持的是每一对象一个 servant,还是所有对象共用一个 servant,这仅仅是与 POA 相关的策略当中的一种。有很多其他的策略可被支持。例如,POA 可以像支持持久对象一样的支持暂态对象。同样地,使用线程也有不同的策略。我们略去这些策略的细节,在文献(Henning 和 Vinoski, 1999)中有对它们的深入讨论。

3. 代理

为了使基于代理的应用程序便于使用,CORBA 采用了允许来自不同系统的代理协同工作的模型。CORBA 指定了代理系统应实现的标准接口,而没有指定它自己的代理模型。这一方法有它潜在的优点,即不同类型的代理可以在一个分布式应用程序中使用。例如,在 CORBA 中可以用 Java applet 创建 D'Agents 平台上的 Tel 代理。(D'Agents 系统在第 3 章中已有论述。)

在 CORBA 中,代理通常被定义为代理系统(agent system)。代理系统是一个允许创建、执行、传输和终止代理的平台。每个代理系统都有一个相关的配置文件,描述代理系统的具体情况。例如,有一个 D'Agents 代理系统的配置文件,指定其类型(“D'Agents”),支持的语言(比如 Tel),以及代理在不同系统间移动时串行化的方法。

代理通常位于代理系统的特定地点(place)。地点对应于代理所在的服务器。代理系统中可能有多个地点。换句话说,CORBA 假设一个代理系统可以由多个进程组成,每个进程处理一个或多个代理。采用这一组织形式是为了把多个代理主机编组在同一管理域中,从而可以整体地,即以代理系统的方式引用该组。

代理系统依次可被组成区域(region),在这里区域代表代理系统所在的管理域。例如,大学的一个系可能有几个不同类型的代理系统。每个代理系统分布于多个主机或地点,而每一主机可能运行几个代理。这一模型如图 9.13 所示。

CORBA 中的代理假定是从一组类,或者至少是一个含有必需编程文本的文件创建的。在两种情况下,都应该能够对代理的实现命名,并把名字传递给代理系统,以允许创建或传输代理。

CORBA 中的每一代理系统必须实现一定数量的标准操作。例如,代理系统必须提供下列操作: 创建或终止代理、接收代理并执行、列出当前代理集合、列出代理所在的地点、以及挂起和恢复代理。注意,这些操作的实现完全依赖于实际的代理系统。然而,CORBA 要求所有的代理系统遵从整体模型。这就意味着如果一个已存在的代理系统初始时不支持地点的概念,它应当提供此项支持。

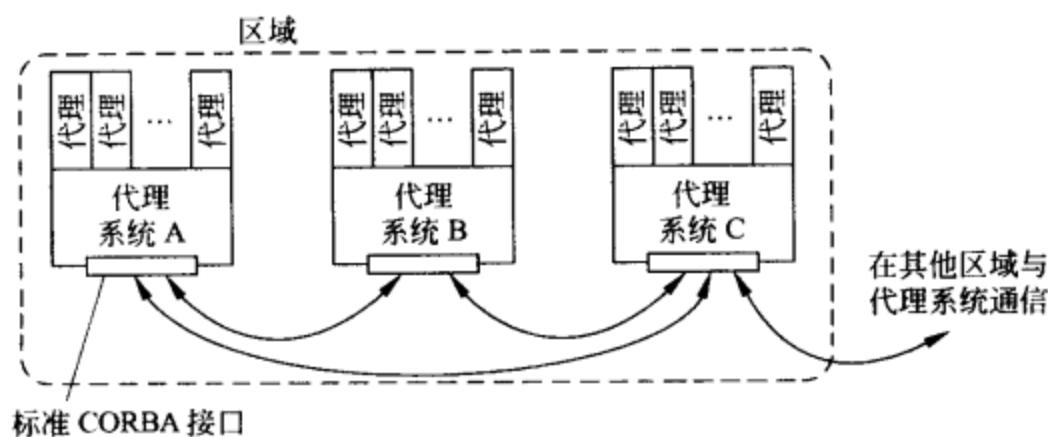


图 9.13 CORBA 中代理、代理系统和区域的整体模型

每一区域有一个相关的搜索器(finder)，允许在区域内搜索代理、地点和代理系统的位置。搜索器提供一定的标准操作，用于注册和注销代理、地点和代理系统。本质上说，搜索器就是一个简单的区域目录服务。

CORBA 中移动代理的更多细节可以在文献(OMG 2000e)中找到。

9.1.4 命名

CORBA 支持不同类型的名字。最基本的名字类型是对象引用名和基于字符的名字，如 CORBA 命名服务所支持的。此外，还有很多高级的命名工具，通过它们的命名可基于相关属性找到对象。下面，将详细讨论 CORBA 中的基本命名，重点是对象引用。有关 CORBA 交易服务中提供的高级命名工具的内容，读者可以参考文献(OMG 2001a, OMG 2000b)。

1. 对象引用

CORBA 的基础是其对象引用的方法。当客户拥有对象引用时，它可以调用由所引用对象实现的方法。重要的是要区别客户进程用于调用方法的对象引用和由底层的 ORB 实现的对象引用。

进程(可能是客户的或服务器的)只能使用一种与语言相关的对象引用的实现。在大多数情况下，是以指向对象的局部表示的指针的形式进行的。这种引用不能从进程 A 传递给进程 B，因为它只在进程 A 的地址空间中有意义。相反，进程 A 可以先把指针编组成一个独立于进程的表示。这一操作由其 ORB 提供。一旦编组，引用就可以传递给进程 B 并解编。注意，进程 A 和 B 可以是用不同语言写成的可执行程序。

相比之下，底层的 ORB 有它自己的独立于语言的对象引用的表示。这种表示甚至与它在进程间传递的用于交换引用的编组版本也不相同。重要的是当进程引用对象时，其底层 ORB 隐性地传递了足够多的信息，从而知道实际上引用的是那个对象。这种信息通常是由客户端和服务器端的存根传递的，这些存根从对象的 IDL 规范中生成。

CORBA 的早期版本中存在的问题之一是每个 ORB 都能够决定如何表示对象引用。因此，如果像上文描述的那样，进程 A 要向进程 B 传递一个引用，只有在两个进程运行于同一个 ORB 上面时才能成功。否则，进程 A 持有的引用的编组版本对进程 B 底层的

ORB 是毫无意义的。

当前的 CORBA 系统都支持相同的独立于语言的对象引用表示法,称为可互操作的对象引用(interoperable object reference)或 IOR。ORB 内部是否使用 IOR 并不重要。然而,当两个不同的 ORB 间传递对象引用时是以 IOR 的形式实现的。IOR 含有识别对象所需的全部信息。IOR 的总体结构如图 9.14 所示,图中还给出了 IIOP 的详细信息,后文中将对此进行详细讨论。

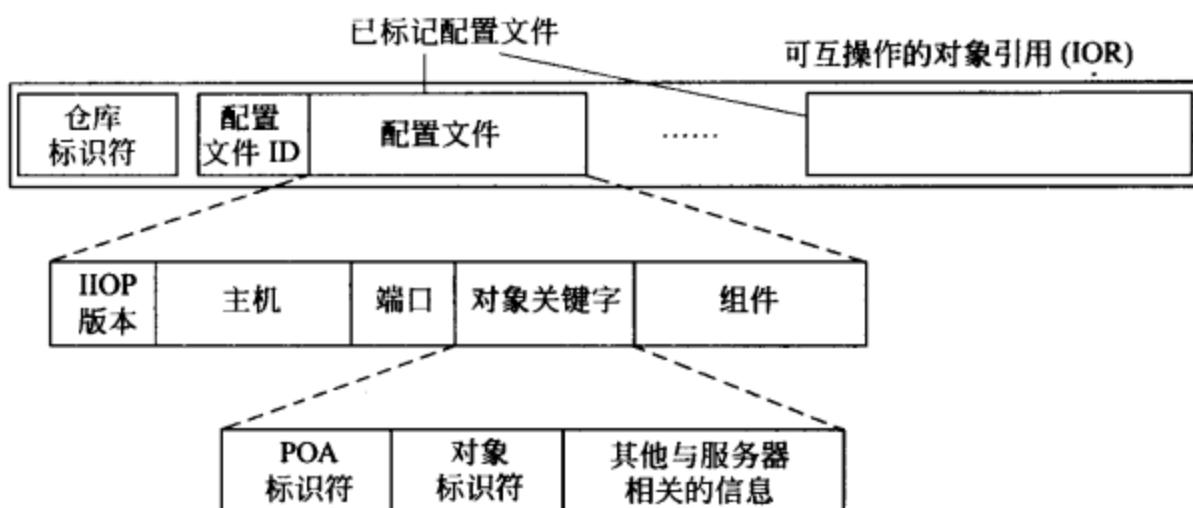


图 9.14 IOR 的组织及 IIOP 的详细信息

每个 IOR 以仓库标识符开头。如前文所述,当接口存储在接口仓库中时,此标识符被分配给一个接口。它用于在运行时检索接口信息,也可以帮助执行诸如类型检查或动态创建调用之类的操作。注意,如果此标识符是有用的,客户和服务器必须能够访问相同的接口仓库,或者至少使用相同的标识符来确定接口。

IOR 中最重要的部分由所谓的已标记配置文件(tagged profile)组成。每个标记配置文件含有调用对象所需的完成信息。如果对象服务器支持多种协议,每种协议的信息可以记录在一个单独的已标记配置文件中。用于 IIOP 的配置文件的细节如图 9.14 所示。

IIOP 配置信息用已标记配置文件中的配置 ID(ProfileID)字段确定。其主体包括 5 个字段。IIOP 版本(IIOP version)字段确定此配置文件中使用的 IIOP 版本。

主机(Host)字段是确定对象位于哪个服务器上的字符串。主机可以用完整的 DNS 域名(例如 soling.cs.vu.nl),或者主机 IP 地址的字符串表示(例如 130.37.24.11)来确定。

端口(Port)字段含有对象服务器监听输入请求的端口号。

对象关键字(Object key)字段含有服务器相关信息,用于将输入的请求多路复用给相应的对象。例如,POA 生成的对象标识符通常是该对象关键字的一部分。同时,此关键字将确定相应的 POA。

最后,组件(Components)字段选择性地包含为适当地调用引用对象的附加信息。例如,组件字段可以含有说明如何处理引用的安全信息,或者说明在引用的服务器(暂时)不可用的情况下应如何处理。下面回到这些问题。

现在我们已经解释了对象引用的细节,不难明白,客户是怎样绑定到对象以便随后调

用方法的。回忆一下第 2 章中提到,把客户绑定到对象是建立到对象的连接从而客户能够调用对象的方法的过程。只有客户拥有对此对象的引用时,绑定才可进行。

回到对 CORBA 命名服务的讨论上来,假设客户向命名服务请求解析一个可阅读的名称。命名服务将返回一个与语言相关的存储于命名服务中的 IOR 实现。只有客户的 ORB 完成了绑定过程,此实现才能够返回。具体步骤介绍如下。

客户 ORB 接受命名服务返回的 IOR,检查其中包含的仓库 ID,在客户上设置一个代理并向代理返回指针 p。在内部,ORB 存储此事实: p 与对象的 IOR 相关联。

不同的 ORB 有不同的实现,但典型的方法如下: 在向客户传递 p 之前,客户 ORB 检查 IOR 中包含的已标记配置文件。假设对象可用 IIOP 调用。在这种情况下,客户 ORB 将使用在 IOR 中找到的主机地址和端口号,与对象服务器建立 TCP 连接。此时,可以把 p 传递给客户。

不论客户何时调用对象的方法之一,客户 ORB 把调用请求编组为 IIOP 请求(Request)消息。此消息含有与服务器相关的对象关键字,IOR 中也存储此关键字。消息通过 TCP 连接传送给服务器,随后根据对象关键字传递给合适的 POA。POA 将请求转发给合适的 servant,在那里,请求被解编并转换为真正的方法调用。

在这种方法中,IOR 直接引用对象,也就是所谓的直接绑定(direct binding)。与直接绑定相对应的是间接绑定(indirect binding)。使用间接绑定时,绑定请求首先被送到实现仓库。实现仓库是由对象 IOR 确定的另一个进程。它的作用和注册表一样,通过它可以在传送调用请求之前定位并激活引用的对象。在实践中,间接绑定主要用于持久性对象,即由 POA 控制的遵从持久性生命周期策略的对象。

当客户 ORB 使用基于间接绑定的 IOR 时,它简单地从连接实现仓库开始。该仓库将注意到此请求实际上是对另一台服务器的,并在表中查找,看此服务器是否在运行。如果该服务器在运行,则对其进行定位。如果没有运行,实现仓库可以启动它。这取决于服务器是否支持自动启动。

当客户第一次调用引用对象时,调用请求被送到实现仓库。实现仓库给出对象服务器实际位置的详细情况,如图 9.15 所示。之后,调用请求被转发给适当的服务器。

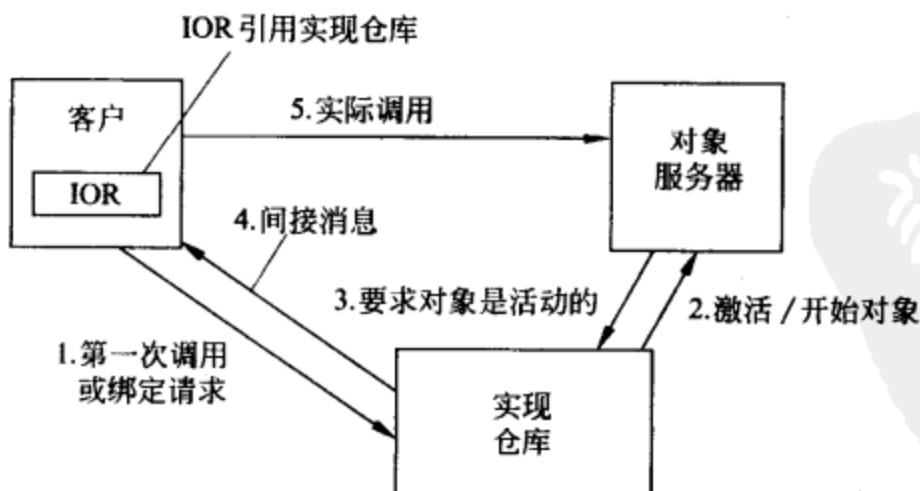


图 9.15 CORBA 中的间接绑定

2. CORBA 命名服务

和其他所有分布式系统一样,CORBA 提供命名服务,允许客户用基于字符的名称查找对象索引。CORBA 中的名称在形式上是名称组件的序列,采取(id,kind)对的形式,其中 id 和 kind 都是字符串。通常,id 用诸如“steen”或“elke”之类的字符串给对象命名。类型(kind)属性是命名对象的简单描述,类似于文件名中的扩展名。例如,称为“steen”的对象可以是“dir”类型,声明它是目录对象。

没有办法把路径名表示为一个字符串。换句话说,CORBA 没有定义名称组件之间的分隔符。而名称是明确地按名称组件序列传递的。序列的表示依赖于语言,对调用命名服务的客户来讲仍然是不透明的。

对于命名图的结构没有限制。命名图中的每一节点都被作为对象。命名上下文环境(naming context)是将名称组件存储到对象引用映射表的对象。因此,命名上下文环境和第 4 章中称作目录节点的东西是一样的。注意,命名上下文环境中一个对象索引可能指向另一个命名上下文环境。

命名图是没有“根”的。然而,每个 ORB 都被要求提供一个初始的命名上下文环境,可以有效地用作命名图的根。名称解析总是与给定的命名图有关的。换句话说,如果客户要解析一个名称,它需要调用特定命名上下文环境中的 resolve 方法。如果名称解析成功,返回的或是命名上下文环境的引用,或者是一个已命名对象的引用。在这种意义上,名称解析的过程与第 4 章中解释的完全一样。因此在这里不再重复。

9.1.5 同步

CORBA 中最重要的两种同步服务是并行控制服务和事务服务。这两种服务用两阶段锁定进行协作以实现分布式事务和嵌套事务。

CORBA 的事务模型如下。事务由客户初始化并由一系列对对象的调用组成。当此类对象第一次被调用时,它自动成为事务的一部分。结果是通知对象服务器它正在参与一项事务。当调用对象时,此信息隐性地传递给服务器。

本质上,有两种类型的对象可以成为事务的一部分。可恢复对象(recoverable object)是由能参与两阶段提交协议的对象服务器执行的对象。特别是,用于这类对象的服务器能通过回滚所有变化来支持中止事务,而这些所有变化是调用它的一个可恢复对象的结果。当然,也可以调用作为事务一部分的对象,这些对象不能恢复到事务开始前的状态。特别是,这些事务对象(transactional object)由不能参与事务两阶段提交协议的服务器执行。事务对象是典型的只读对象。

因此,可见 CORBA 的事务与第 5 章、第 7 章中讨论的分布式事务及其协议很相似。

同样,由并行控制服务提供的锁定服务是人们所期望的。实际上,此服务用中央锁定管理器实现,而没有使用分布式锁定技术。此服务区区别读锁定与写锁定,还能够支持数据库中通常需要的不同间隔的锁定。例如,区别锁定整个表和只锁定一条记录是很有意义的。间隔锁定的更多信息可参见(Gray, Reuter 1993; Garcia-Molina 等 2000)。

9.1.6 缓存与复制

CORBA 没有提供一般的缓存与复制支持。只在 CORBA 的第三版中包含了用于容错的对象复制,9.1.7 节中将讨论这一点。没有一般的缓存与复制支持意味着应用程序开发人员在需要复制时必须借助于特殊的方法。

考虑一个为了性能把复制操作并入 CORBA 的例子。CASCADE 系统实现了这个目标。在 CASCADE 系统中,目标是提供通用的、可升级的机制,允许缓存各种类型的 CORBA 对象(Chockler 等 2000)。CASCADE 提供缓存服务,该服务是作为潜在的大量对象服务器集合实现的,其中每个服务器都被称为 DCS(domain caching server,域缓存服务器)。每个 DCS 是运行于 CORBA ORB 上的对象服务器。DCS 的集合可以通过诸如 Internet 一类的广域网进行扩展。

相同对象的缓存拷贝以分层次形式组织。假设一个客户,比如对象的拥有者,可以在本地 DCS 上注册对象。此 DCS 成为层次结构的根。其他客户可以请求它们的本地 DCS 缓存这一对象的拷贝,而此 DCS 首先要加入到已经缓存了这个对象的当前 DCS 层次结构中。

CASCADE 支持以客户为中心的一致性模型,如第 6 章中所述。此外,它还支持整体排序,即所有的更新在各处都可以保证相同的顺序。每个对象可能有它自己的一致性模型,没有系统级的策略来维护缓存对象的一致性。如在第 6 章中讨论的,执行复制时,重要的是能够同时支持不同的一致性模型,因为模型的适用性非常依赖于对象的用途和访问方式。CASCADE 达到了这一要求。

作为 CORBA 服务,CASCADE 非常依赖截取程序。对客户来说,CASCADE 事实上是不可见的。所有关于一致性的问题都隐藏在对象通常提供的接口背后。客户惟一的显式访问 CASCADE 是在它请求本地 DCS 开始缓存特定对象时。不论何时调用此类对象,调用请求都被客户 ORB 截获并随后转发到 DCS。

根据引用对象的不同一致性模型,调用请求发送到 DCS 之前会加入附加的信息。例如,当客户请求读写一致性时,就需要知道客户见到的最后一个写操作是什么。

DCS 的总体结构如图 9.16 所示。一个 DCS 管理着众多的对象拷贝。这种拷贝由对

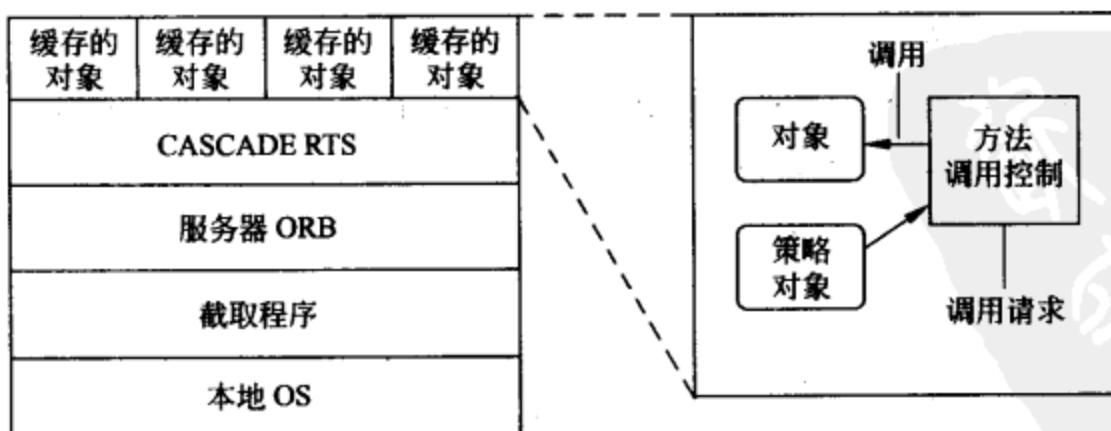


图 9.16 DCS 的(简化)结构

象状态及建立在状态上的操作实现组成。在 CORBA 术语中,DCS 与原始的对象拷贝有着相同的 servant。DCS 底层的截取程序截取输入的调用请求并提取出信息,比如客户端截取程序加入的信息。然后,请求被转发给与被引用对象惟一关联的方法调用控制模块。如图 9.16 所示,每个缓存对象有它自己的策略对象,含有控制对象调用的特定信息。由截取程序提取出的附加信息单独传递给这个策略对象。

虽然 CASCADE 或多或少提供了透明的对象服务器缓存,可以预见,为 CORBA 实现此类缓存服务还需更多的努力。尽管截取程序允许在必要时修改调用以实现这种服务,但 CORBA 没有再提供其他的支持。

9.1.7 容错性

CORBA 系统长期缺乏真正的容错性支持。在大多数情况下,失败只是简单地报告给用户,然后系统不再采取任何行动。例如,如果由于相关的服务器(暂时)不可用导致引用的对象无法获得,客户就会被搁置一旁。在 CORBA 第三版中,明确提出了容错性。容错 CORBA 的规范见(OMG 2000d)。

1. 对象组

CORBA 中处理失败的基本方法是把对象复制到对象组(object group)。这种组由一个和多个相同对象的相同拷贝组成。但是,对象组可以当做单个对象来引用。组提供与其包含的拷贝相同的接口。换句话说,复制对客户是透明的。不同的复制策略都可得到支持,包括主机—备份复制、主动复制和基于法定数量的复制。这些策略都在第 6 章中讨论过。对象组还有各种其他属性,其细节可参见(OMG 2000d)。

为了尽可能多地提供复制与失败处理的透明性,对象组不应该与普通 CORBA 对象有区别,除非应用程序有特殊要求。在这方面,一个重要问题是对象组如何被引用。下面的步骤是使用一种特殊的 IOR,称为可互操作的对象组引用(interoperable object group reference, IOGR)。IOGR 与普通 IOR 的关键不同之处是 IOGR 含有对不同对象的多重引用,在同一个对象组中引入注目地进行复制。相比之下,IOR 也可能含有多重引用,但是都是引用同一个对象,尽管有可能使用不同的访问协议。

当客户把 IOGR 传递给 ORB 时,ORB 就试图绑定到一个引用的副本上。在使用 IIOP 的情况下,ORB 很有可能使用在 IOGR 的 IIOP 配置文件中找到的附加信息。就像我们前面讨论过的那样,此类信息存储在组件(Components)字段中。例如,一个特定的 IIOP 配置文件可能会引用对象组的主机或备份,如图 9.17 所示。方法是使用单独的 TAG-PRIMARY 和 TAG-BACKUP 标记。

如果绑定到一个副本上失败,客户 ORB 将继续尝试绑定到其他的副本上,根据某种策略选择它最适合的下一个副本。对客户来说,绑定过程是完全透明的,看上去和绑定到一个普通 CORBA 对象一样。

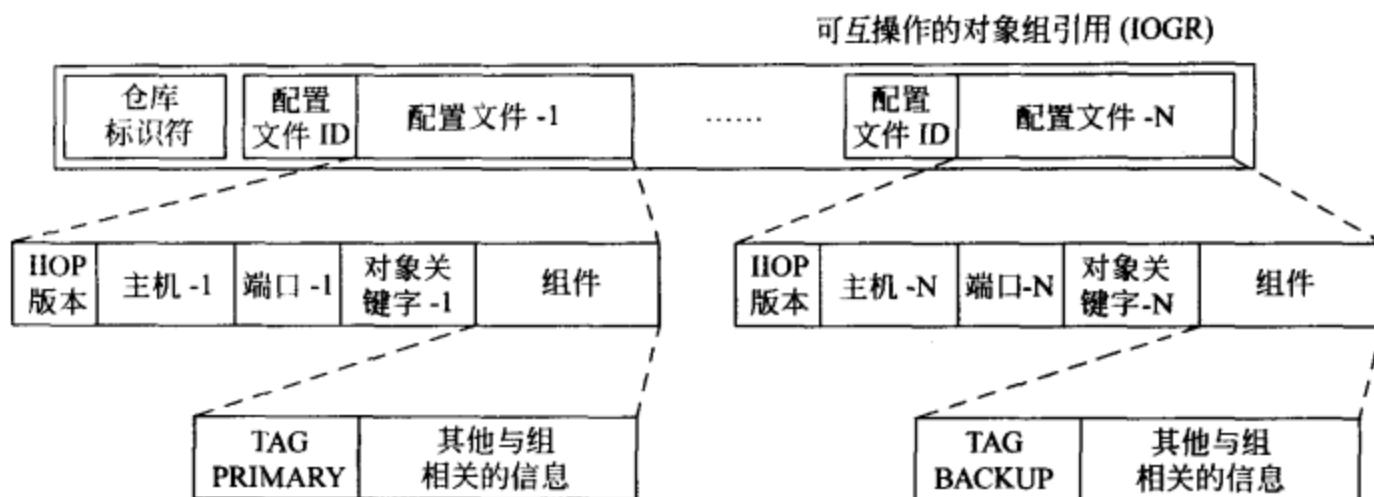


图 9.17 拥有主机和备份的对象组的 IOGR 的可能结构

2. 体系结构实例

为了支持对象组并处理更多的失败管理,需要向 CORBA 添加组件。CORBA 容错版本的一个可能体系结构如图 9.18 所示。这一体系结构是从 Eternal 系统中派生出来的(Moser 等 1998, Narasimhan 等 2000)。Eternal 系统在 Totem 可靠组通信系统的基础上提供了容错性基础结构(Moser 等 1996)。

在此体系结构中,有几个发挥重要作用的组件。目前最重要的是复制管理器(replication manager),负责创建和管理一组复制的对象。从原理上讲,只有一个复制管理器,尽管出于容错性考虑复制管理器本身也可以复制。

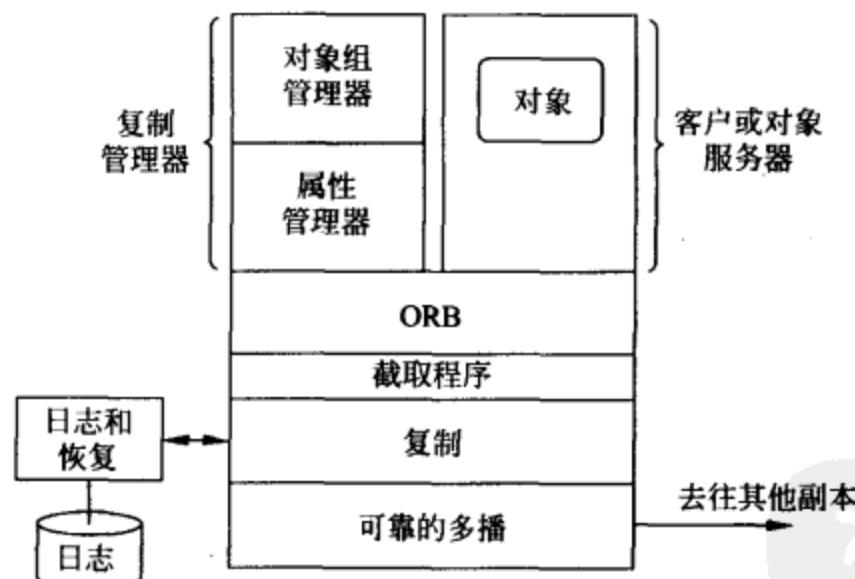


图 9.18 可容错 CORBA 系统的体系实例

如前所述,对客户来讲对象组和其他类型的 CORBA 对象没有什么不同。要创建对象组,客户只需简单地调用提供的普通 `create_object` 操作。在这种情况下,由复制管理器确定所创建的对象类型。客户仍然不知道事实上它正在隐性地创建一个对象组。开始一个新对象组时,创建的副本数量通常由系统的默认值决定。复制管理器还负责在出现失败的情况下替换副本,以保证副本的数量不低于指定的最小值。

此体系结构还展示了信息层截取程序的使用。在 Eternal 系统中，每个调用都被截取并传递给单独的复制组件。这些组件维护对象组的请求一致性并保证信息已被记录，从而可以恢复。

随后，调用请求通过可靠的、完全有序的多播发送给其他组成员。在主动复制的情况下，调用请求通过传到对象底层的 ORB，从而传递给每个对象副本。但是，在被动复制的情况下，调用请求仅传递给主机的 ORB，而其他服务器只是记录调用请求以备恢复。当主机完成了调用，其状态被多播给备份。

9.1.8 安全性

CORBA 的安全性有很长的历史。CORBA 规范的最初版本没有涉及这一主题，因为建立安全性服务的几次努力都失败了。在 CORBA 2.4 版中，安全性服务声明占用了 400 页，非常明确地说明了 CORBA 系统的安全性。下面详细讨论 CORBA 安全性。这些问题的概述见(Blakley 2000)，此书是由最初的 CORBA 安全性规范的作者之一撰写的。

安全性的重要规范问题之一是服务要提供一组合适的机制，以实现各种安全策略。这些服务需要在不同的时间和地点提供，这使问题复杂化了。例如，如果客户要安全地调用一个对象，就需要决定何时使用安全机制（比如，在绑定时、调用时、或是两种情况都用），在何处使用安全机制（比如，在应用层、在 ORB 内，或在信息传输过程中）。

CORBA 安全的核心是对安全对象调用的支持。潜在的想法是应用层对象应该对使用的各种安全服务毫不知情。但是，如果客户有特殊的安全要求，应允许它说明要求，以便在调用对象时考虑其要求。被调用的对象也会出现类似的情况。这种方法导致了如图 9.19 所示的通用结构。

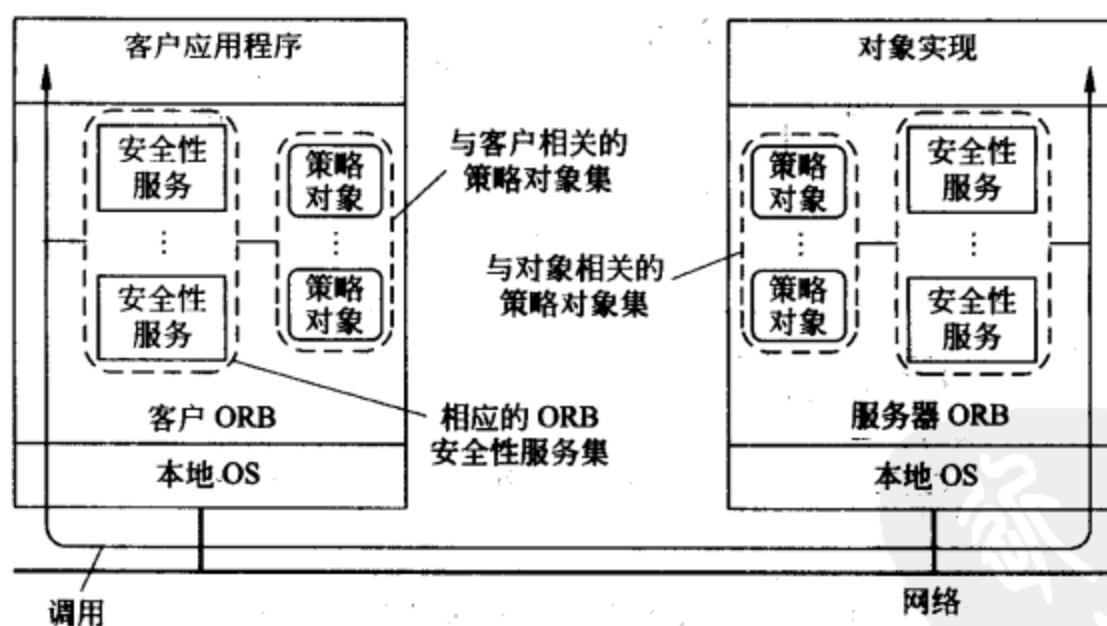


图 9.19 CORBA 安全对象调用的通用结构

当客户为了调用而绑定到某一对象时，客户 ORB 决定需要哪种安全性服务在客户端支持安全调用。服务的选择由客户在其中运行的管理域的安全性策略决定，但也受客户的特定策略的影响。

安全性策略通过与客户相关的策略对象(policy object)声明。实际上,客户运行所在的域将使其安全性策略对客户ORB可用,方法是使用一组特定的策略对象。默认的策略将自动与客户关联。策略对象的实例包括声明所要求信息保护类型的对象和有信任伙伴表的对象。其他实例及详细内容参见(Blakley 2000)。

服务器端也采用类似的结构。被调用对象在其中执行的管理域要求使用一组特定的安全性服务。同样,被调用对象本身也有一套相关的策略对象,存储特定对象的信息。

可用各种方法实现CORBA安全性。特别是,为了使ORB尽可能通用,需要通过标准接口指定不同的安全性服务,隐藏这些服务的具体实现。能够这样指定的服务在CORBA中称为可替换的(replaceable)。

可替换的安全性服务假设与两种不同的截取程序一起实现,如图9.20所示。访问控制截取程序(access control interceptor)是请求层的截取程序,检查调用的访问权限。此外,还有一个信息层的截取程序,安全调用截取程序(secure invocation interceptor),负责实现信息保护。换句话说,此截取程序能够加密请求并保证其完整性与机密性。

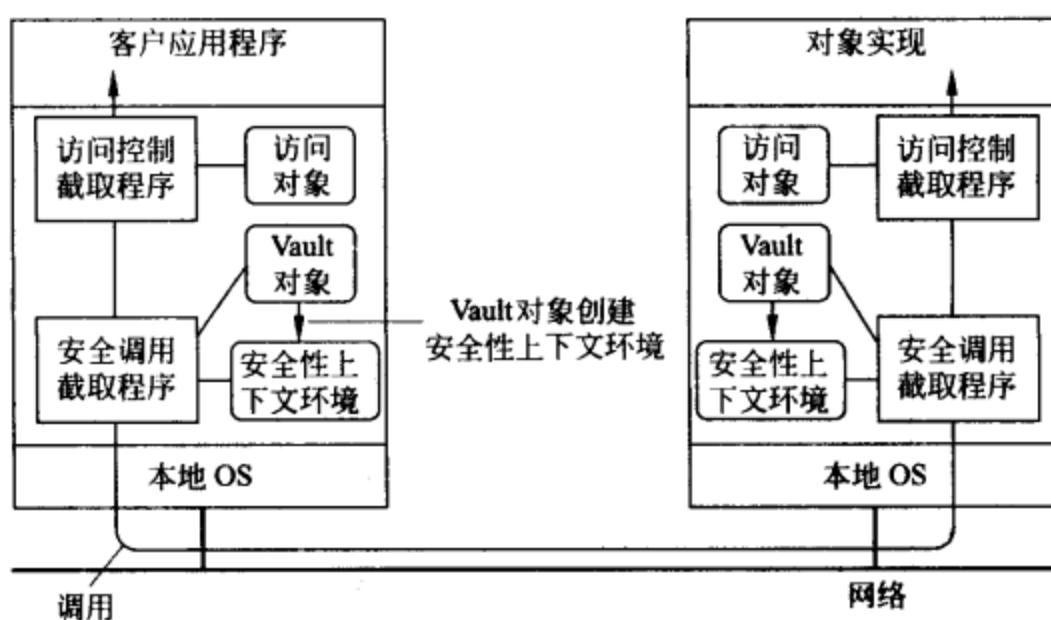


图9.20 CORBA中安全性截取程序的作用

安全调用截取程序起到了至关重要的作用,它负责为客户建立安全性上下文环境(security context),以允许对目标对象的安全调用。这个安全性上下文环境用一个安全性上下文环境对象来表示,含有安全调用目标对象所需的全部信息和方法。例如,它描述使用何种机制,提供加密解密消息的方法,存储证书的索引等。

对象服务器也需要建立自己的安全性上下文环境对象。因此,客户截取程序将首先向服务器发送含有必要信息的消息,验证客户并让服务器为随后的调用创建安全性上下文环境。注意,服务器的安全调用截取程序将检查与对象相关的策略对象,看所有的安全性要求是否能达到,如何达到。返回给客户的响应可能含有附加信息,允许客户验证服务器。

这种初始化信息交换之后,客户将绑定到目标对象上,二者将建立通常所说的安全关联(security association)。从此开始,可进行安全调用,而安全调用截取程序根据客户和对象服务器达成的策略保护请求和响应消息。

建立安全结合过程中的一个至关重要的角色是一个有标准接口的独立对象,称为 vault 对象。vault 对象由安全调用截取程序调用,创建安全环境对象。截取程序首先从与客户相关的策略对象中读取策略信息,然后传递给 vault 对象。显然,vault 对象必须以不受干扰的方式作为 ORB 的一部分来实现,并属于 CORBA 系统中可信任计算基础。

9.2 分布式组件对象模型(DCOM)

我们将要研究的第二个基于对象的分布式系统是 Microsoft 的分布式组件对象模型(DCOM)。正如其名称所示,DCOM 是从 COM(组件对象模型)扩展来的。COM 是 Microsoft 提出的一种技术,可用于 Windows 95 之后的各种 Windows 操作系统。与 CORBA 不同,DCOM 不是一个委员会的工作成果,这可以从 COM 从 1995 年 10 月(Microsoft 公司 1995)到现在只有一个 300 页的草拟可用规范的事实反映出来。一个良好的体系和设计应该只包含一套最小规模的核心元素,并在这之上建立组件和服务。遗憾的是,因为 DCOM 并不是出自委员会之手,这导致它没有这样一个良好的体系。相反,DCOM 到现在为止还是一个复杂的系统,许多相似的事情可以用很多不同的方法来完成,有时甚至不同的解决方案无法共存。

想批判 DCOM 并不难。但是,可以公正的说,与 CORBA 相比,DCOM 是在一定范围内得到证明的技术。在成千上万的每天在网络环境下使用 Windows 的人当中,DCOM 已经得到了广泛的应用。在这种意义上,CORBA 或者其他任何一个分布式系统,都望尘莫及。

在后面的部分中,我们将使用与介绍 CORBA 时同样的结构来介绍 DCOM 的重要部分。为了更好理解什么是 DCOM,最好能参考一些面向编程的书,比如(Platt 1998, Rogerson 1997)。而 Eddon 和 Eddon(1998)对 DCOM 的许多技术方面给出了很好的介绍。一个 Windows 2000 环境下的有关分布的 DCOM 概述可以在(Chappell 2000)里找到。

9.2.1 DCOM 概述

DCOM 的基础是由 Microsoft 的组件对象技术——COM 构成的。COM 的目的是支持可动态建立并能相互作用的组件的开发。COM 的组件都是可执行代码,包含于(动态链接)库中或者以可执行程序方式存在。

COM 本身是用一种与进程链接的库的形式给出的。本来,它是为了支持一种所谓复合文档(compound documents)而开发的。正如我们在第 3 章里提到的,一个复合文档就是一个由各种不同的部分,如(格式化)文件、图像、电子表格等建立起来的文档。反过来,每个部分又使用它本身的应用程序去操作。

为了支持大量的复合文档,Microsoft 需要一个通用的方法以便能够区分各种不同的部分,并能够将它们结合在一起。最开始出现的是 OLE,即对象链接和嵌入(object linking and embedding)。OLE 的第一个版本使用一种粗糙且非柔性的传递消息的方法在不同的部分之间通信。很快它就被一种新版本所取代,仍然叫 OLE,但是这种新版本

是建立在一种比较灵活的层次上的,也就是 COM 上。图 9.21 给出了它的组织结构。

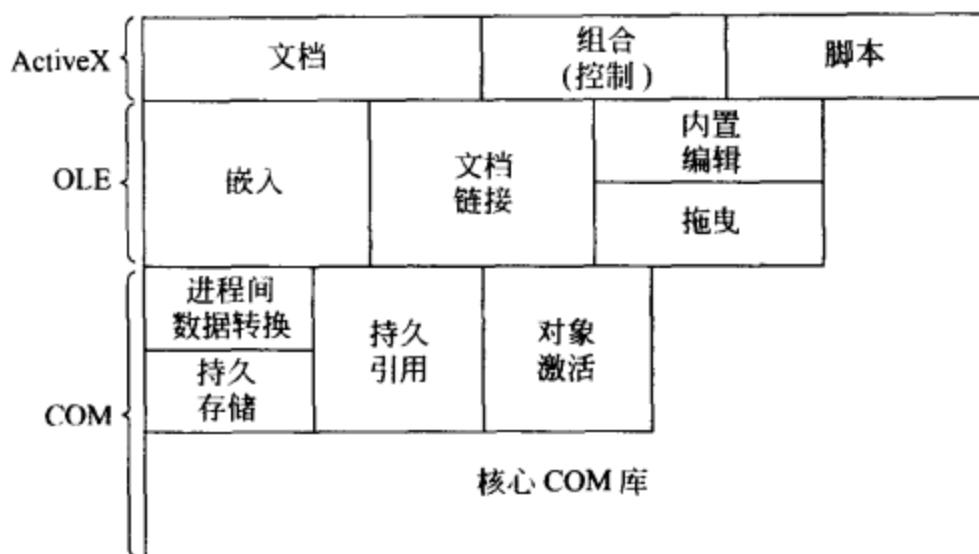


图 9.21 ActiveX、OLE 和 COM 的常规结构

图 9.21 也展示了 ActiveX。这个术语涵盖了以前所有 OLE 的内容,同时又有新的特点。这些新特点主要包括,在不同进程中建立组件的灵活性,脚本支持,以及或多或少地进行对象的标准分组,组成所谓的 ActiveX 控件。很多 DCOM 专家(甚至 Microsoft 的专家)都似乎一致认为 ActiveX 没有一个准确的定义,因此我们也不会试图定义 ActiveX。

DCOM 给这个结构新加的功能就是可以使一个进程能够与另一台机器上的组件通信。除此之外,DCOM 提供的组件间交换信息的基本机制与 COM 是完全相同的。换句话说,对于程序员来说,COM 与 DCOM 的区别常常隐藏在各种接口的后面。正如我们将要看到的,DCOM 主要提供访问的透明性。其他形式的分布式透明性没有这么明显。

1. 对象模型

就像实际中所有其他基于对象的分布式系统一样,DCOM 采用远程对象模型。实际上,DCOM 的对象只要与客户在同一个进程中,那么它既可以是本地机上的进程,也可以是远程机的进程。稍后再讨论这些不同之处。

与 CORBA 相似,DCOM 对象模型是以接口的实现为中心的。简单来说,一个 DCOM 对象就是一个接口的实现。一个单独的对象可以同时实现几个接口。但是与 CORBA 相比,DCOM 只有二进制接口(binary interface)。这样一个接口本质上就是一个指针表,其中的指针指向作为接口的一部分的方法实现。当然,用一个独立的接口定义语言(IDL)来定义接口也是很方便的。DCOM 也有这样的 IDL,叫做 Microsoft IDL(MIDL),使用它可以生成二进制接口的标准格式。

使用二进制接口的好处是这些接口是独立于编程语言。在 CORBA 中,每次需要支持一种新的编程语言时,从 IDL 规范到那种语言的映射都要标准化。而在二进制接口中,这种标准化是不必要的。图 9.22 给出了这两种方法的区别。

每个 DCOM 的接口都有一个惟一的 128 位标志符,叫接口标识符(interface identifier, IID)。每个 IID 都是全局惟一的:没有任何两个接口有相同的 IID。这样一个

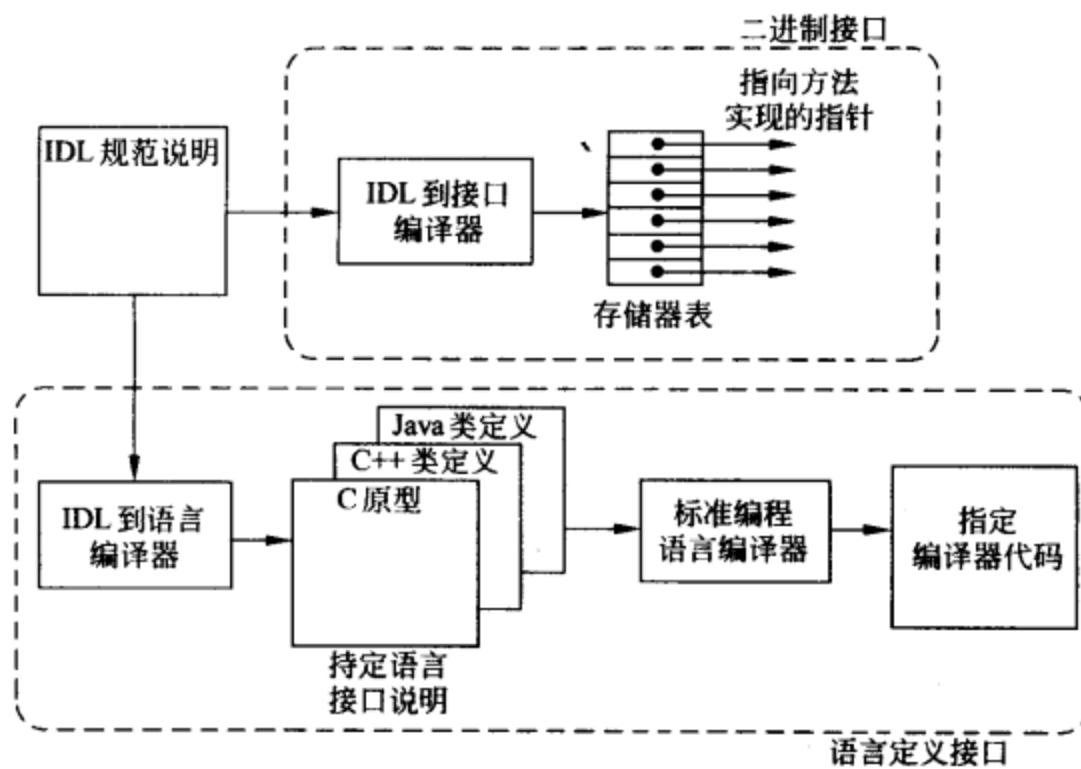


图 9.22 语言定义接口与二进制接口之间的区别

标识符是将一个较大的随机数、当地时间和当前主机的网络接口地址结合起来产生的。这样生成的两个标志符相同的几率几乎为零。

一个 DCOM 的对象是作为一个类的实例创建的。为了进行这一操作,这个类必须是可用的。为了达到这个目的,DCOM 有类对象(class object)。在形式上,这样的对象可以是能实现 IclassFactory 接口的任何东西。这个接口包括 CreateInstance(创建实例)的方法,与 C++ 和 java 中 new 操作符类似。对类对象调用 Create Instance 操作,结果是建立了一个 DCOM 对象,包括与类对象有关的接口实现。

因此,类对象代表同样类型对象,即同样类型接口实现的集合。属于同一类的对象通常只能用它们当前的状态来区分。通过将给定类对象中的一个对象实例化,就可以调用这些接口包含的方法。在 DCOM 中每个类对象都用全局惟一的类标识符 CLSID(class identifier)来引用。

所有的对象实现一个标准的对象接口,叫做 Iunknow。当通过调用 CreateInstance 创建一个对象时,这个类对象返回一个指向此接口的指针。Iunknow 包含的一个重要的方法是 QueryInterface(查询接口)方法,它可以返回一个指针,指向另一个由给出 IID 的对象实现的接口。

DCOM 与 CORBA 对象模型的一个重要区别是 DCOM 的所有对象都是暂时的。换句话说,当没有客户再引用这个对象时,它就被删除了。引用计算是要调用 Iunknow 的两个部分 AddRef 和 Release 来完成的。现在不明确的是,既然对象都是暂时的,那为什么每个对象还都要有一个代价相对较高的全局惟一标识符呢?因为上述原因,DCOM 的对象只能通过接口指针的途径来引用。因此,就需要有特殊的方法将一个对象引用传到另一个进程中去。关于这点,我们将稍后讨论。

DCOM 还支持对象的动态调用。一个可以在运行时建立调用请求的对象需要实现

IDispatch 接口。这个接口类似于 CORBA 的动态调用接口(DII)。

2. 类型库和注册

DCOM 有一个与 CORBA 的接口仓库等同的库,称为类型库(type library)。类型库通常与一个应用程序或其他较大的包含各种对象类的组件有关。库本身存在于一个独立的文件中,或者可以作为应用程序中的一部分。在任何情况下,类型库都是主要用于找出一个将要被动态调用的方法的准确特征。除此之外,编程工具也要使用类型库来帮助程序的开发,比如将界面方便地显示在屏幕上。

为了实际激活一个对象,也就是保证它被创建并被放置在一个可以使它接受方法调用的进程中,DCOM 使用 Windows 注册表并使之与一个叫做服务控制管理器的特殊进程结合。注册表用于记录一个从 CLSID(class identifier,类标识符)到包含该类实现的本地文件名的映射。无论何时一个进程想要创建一个对象,都必须首先保证加载一个合适的类对象。

当一个对象要在远程主机上执行时,可遵循不同的路线。在这种情况下,客户要与那个主机的服务控制管理器(SCM)进行联系。SCM 是一个负责激活对象的进程,与 CORBA 中的实现仓库类似。远程主机上的 SCM 在其本地注册表中查询与这个 CLSID 有关的文件,随后为这个对象开启一个进程。服务器的接口指针被编组并被返回给客户,然后解编给一个代理。稍后我们将详细讨论编组与解编接口指针。

图 9.23 给出了结合了类对象、对象和代理使用的 DCOM 的完整体系(参见 Chung 等 1998)。在客户端,一个进程可以访问 SCM 和注册表,帮助查询并绑定一个远程对象。这个客户将被提供给一个用于实现对象接口的代理。

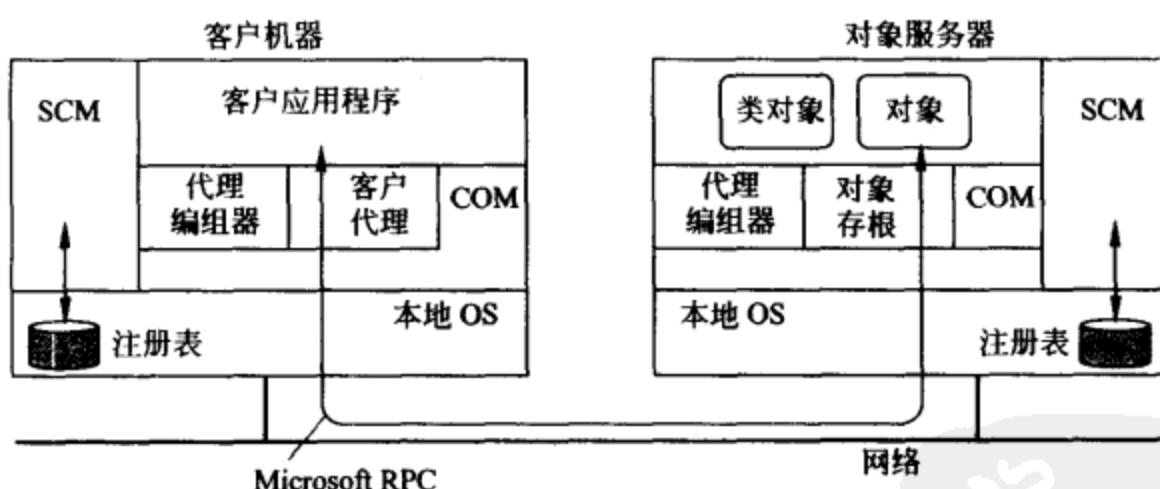


图 9.23 DCOM 的完整体系

对象服务器将包含一个存根用于编组和解编那些发往实际对象的调用。客户与服务器之间的通信通常由 RPC 方法完成,同时也提供其他通信原语。关于 RPC,我们将在本节中稍后说明。

3. DCOM 服务

DCOM 可以视为是 COM 的一个扩展,它在 COM 上增加了与远程对象通信的工具。

但是,COM 本身也发展了一个新版本 COM+。COM+可以视为 COM 的一个扩展集,由各种以前是作为 COM 附件提供的服务组成。尤其是,COM+拥有可有效处理很多对象的服务器工具。除此之外,还增加了那些要用外部服务器来实现的服务,比如由消息队列系统提供的服务。

试图在 COM、DCOM、COM+和外部服务之间画出简单的分界线是很困难的,因为这只会引起混乱。下面的篇章将简单地以 DCOM 统称 COM、COM+和 ActiveX。我们将 DCOM 依赖的外部服务与它实际包含的组件区分开来,这样是可以做到的,也更有帮助。为了更好的理解 DCOM 中包含的特殊服务,图 9.24 给出了 CORBA 服务并指出 DCOM 中与之对应的部分。Windows 2000 服务通常对 DCOM 客户是适用的,但是它们不是 DCOM 本身的一部分。图 9.24 也列出了这些服务。

CORBA 服务	DCOM/COM+服务	Windows 2000 服务
集合	ActiveX 数据对象	—
查询	无	—
并行	线程并行	—
事务	COM+自动事务处理	分布式事务协调程序
事件	COM+事件	—
通知	COM+事件	—
外部化	编组工具	—
生命周期	类工厂、JIT 激活	—
许可	特殊类工厂	—
命名	别名	活动目录
属性	无	活动目录
贸易	无	活动目录
持久性	结构化存储	数据库访问
关系	无	数据库访问
安全性	授权	SSL、Kerberos
时间	无	无

图 9.24 DCOM 服务与 CORBA 服务的简要对比

这些服务的大部分将要在下面讨论。必须注意的是 DCOM 与 CORBA 不同,它没有真正地组成一个完整的分布式系统,因为它假设了外部服务的存在性。举例说,命名服务显然是组成分布式系统的一部分(而它不支持)。Windows 2000 中的命名是通过活动目录(Active Directory)支持的。这个服务基本是由一组 LDAP 目录服务器组成的,这些服务器用 DNS 来命名和查询。我们在第 4 章曾讨论过 DNS 和 LDAP。然而,活动目录不是 DCOM 的一部分。因此,当在 UNIX 环境下运行 DCOM 应用程序时,应用程序不能使用与 Windows 2000 环境下相同的命名服务。看来移植性的限制总是存在的。

显而易见,这种方法妨碍了互操作性。另一方面,考虑到实际上所有 DCOM 应用程

序都是在 Windows 环境下运行的,所以互操作性缺乏到什么程度才真正成为问题,这尚待观察。

9.2.2 通信

和 CORBA 类似,DCOM 中的通信最初也只是同步的:调用对象的用户在接收到回应之前都是处于阻塞状态的。这种同步调用现在仍然是默认的方式。但是,DCOM 现在也支持一些其他形式的调用,稍后我们将对此进行说明。

1. 对象调用模型

正如我们所说的,对象的同步调用是 DCOM 的默认方式。如果有错误发生——这种情况在分布式环境很容易碰到的——客户就会收到一个错误代码。DCOM 本身不会试图再调用这个对象,这也就是至多一次(at-most-once)的语义。

对完全同步调用的最早扩展之一是通过回叫接口的方式提供的。回叫接口是由一种特殊的 DCOM 对象——可连接对象(connectable object)支持的。当客户实现回叫接口时,回叫接口会向这个可连接对象提交一个指针。可连接对象指定它希望客户实现的回叫接口。当客户绑定到可连接对象时,它必须提供一个指向回叫接口的指针。如果可连接对象在远程主机上,这个接口指针就要被编组并传送到那个机器上去。稍后将对接口指针的编组进行描述。

随着 COM+的引入及它集成 Windows 2000 中,其他调用模型也得到了支持。其中一个扩展就是能够取消一个未完成的同步调用。无论何时线程 A 开启了一个同步调用,一个取消对象(cancel object)就会同时被创建用以实现 cancel 方法。这个方法可以被线程 B 调用,将线程 A 的标识符作为其参数传递。结果就是线程 A 会立即解除阻塞,不再等待收到调用的响应。

异步调用也得到了支持,它对应于 CORBA 中支持异步方法调用的轮询模型。实际上,建立支持异步调用的接口的方法也是一样的。都是以接口的 MIDL 描述作为起点,对于接口的每个方法 m, MIDL 编译器都生成两个独立的方法: Begin_m 和 Finish_m。前者只包含方法 m 的输入参数,而后者只有其输出参数。

客户通过调用 Begin_m 开始,并即刻继续运行。对于对象,同步与异步调用之间并没什么区别;它惟一能见到的是方法 m 的调用。调用的结果会返回给客户,并缓存直至客户调用 Finish_m 为止。

DCOM 的异步调用要求用户和对象处在运行状态。换句话说,通信是暂时的。持久的通信是通过消息队列支持的,后面我们将讨论之。

2. 事件

在 DCOM 中,可连接对象最初是用于实现事件的,它要求客户和对象都处于激活状态。但是,将几个客户连接到同一个可连接对象上,然后让这个对象在需要的时候同时回应所有客户是可能的。在这个意义上,可连接对象提供了一种简单地将事件发布给多个客户的方法。

DCOM 通过 COM+ 提供一个更完善的事件模型。它更适合实现将在第 12 章中讨论的被称作 publish/subscribe 的系统。它的基本思想与 CORBA 的推式事件模型很相似。事件用一个只有输入参数的方法调用来模拟。事件被归入一个事件类。这个事件类是由一个有自己的 CLSID 的常规 DCOM 类对象来表示的。这里没有必要单独实现事件类,因为 DCOM 提供了一种默认实现。当将一个事件类实例化为对象后,提供者就会简单地调用对象合适的方法生成事件。注意,事件本身并不是对象。

订阅一个事件也很简单。假设一个事件是由 m_event 的方法代表的。订阅者需要提供方法的实现。要想订阅,就要向事件系统传送一个指向实现那个方法的接口的指针。这样,无论何时提供者调用 m_event,事件系统都会负责每个订阅者的 m_event 版本也都被调用。图 9.25 给出了 DCOM 事件的原理。正如所看到的,这种方法确实与 CORBA 的推式事件模型相似。

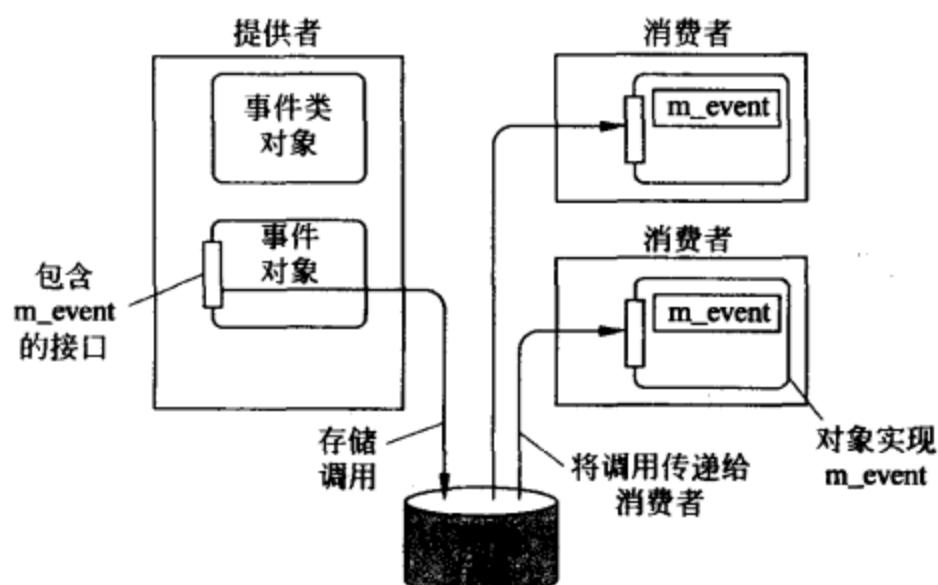


图 9.25 DCOM 中的事件处理

DCOM 中的事件可以存储。这样,如果订阅者在事件发生时没有处于激活状态,只要它愿意,仍可以稍后获得这个事件。在这种情况下,事件系统将创建提供者事件类的实例,并通过调用相关的方法依次重放这些事件。结果是,那些相关的事件方法也被订阅者调用,就像非持久性事件的情况一样。

3. 消息

除了暂时的异步通信以外,DCOM 也提供使用 QC(队列组件,queued component)的持久性异步通信。QC 实际上就是一个 Microsoft 消息队列(Microsoft message-queuing, MSMQ)的接口。MSMQ 就是一种与我们在第 2 章中讨论过的 IBM MQSeries 非常相似的消息队列系统。QC 的特性之一就是将 MSMQ 在客户与对象面前完全隐藏。方法调用以与上面讨论的异步调用同样的方式处理。主要的工作方式如下。

使用 QC 的异步调用仅限于含有只带输入参数的方法的接口。有返回值和输出参数的方法是不能接受的。换句话说,QC 只提供单向异步方法调用,这在概念上与异步 RPC 是相同的。

当客户绑定到一个支持 QC 的对象上时,它会收到一个含有能被异步调用的方法的接口指针。客户每次调用这样的方法时,此调用就会被自动编组,并被 QC 子系统存储在本地客户上。在客户调用方法结束,即调用 Release 来释放接口之后,被编组和存储的方法就会通过 MSMQ 发送出去。

一到达目的地,QC 子系统就会将每个调用解编,随后调用对象。调用是否还按它们发出的顺序排列是不能保证的,除非它们是事务的一部分。事实上,除非明确要求,就连递交都不保证发生。

就像大多数 MQSeries 之类的消息队列系统一样,MSMQ 提供事务队列。最简单形式的事务队列由一组消息的集合组成,这些消息或者全部传递给下一个队列,或者全不传递。除此之外,事务队列还保证在发生崩溃性失败后将消息恢复。在更大的分布式事务中,也存在更高级的事务处理形式。在这种形式中,单独的事务队列联合形成一个嵌套事务。这种情况要使用我们曾在第 7 章中讨论的独立协调器。

9.2.3 进程

DCOM 的远程对象模型第一眼看上去显得相对简单。但是,当进入一些细节时,(本节中)我们将了解到事情有时候并不像看起来那么简单。在接下来的篇幅中,我们将进一步分别讨论 DCOM 的客户和服务器的组织结构。

1. 客户

DCOM 的客户端支持是为人们所期望的。对于客户来讲,只要在它自己的地址空间里运行对象,什么事情看起来都差不多是一样的。换句话说,建立和调用对象的方法与 DCOM 出现之前,即只有 COM 存在时,都是相同的。

然而,有些事情需要特别注意。其中最重要的是对象引用该如何处理。回忆一下,在 DCOM 中,客户对一个对象的惟一引用是通过接口指针实现的。用户使用这个指针在引用的接口中调用可用的方法。当对象本身位于远程机器上时,客户端的接口是用代理实现的,这个代理将调用编组,并向对象发送请求。

一个进程 A 怎样向另一个进程 B 传送对象引用呢?显然,传送其接口指针没有任何意义。相反,进程 B 最终需要的是一个与进程 A 相同的接口实现的指针。那个接口是用它的 IID 惟一标识的。换句话说,传递接口指针本质上就是需要传递接口的 IID。除此之外,进程 A 必须将约束信息传递给 B,比如对象的位置和要用的传输协议。假设进程 B 在本地有可用的接口代理实现,它就可以简单地加载此实现并开始调用其方法。

大多数的代理都采取一个标准的方法来编组调用请求。这些代理可以由 MIDL 规范自动生成,或者可以用类型库里存储的信息来创建。但是,也有一些场合用标准编组来处理调用请求并不是最好的办法。例如,让代理缓存前面调用的结果也许会更方便。在代理里结合缓存需要应用程序开发人员明确地将之编入程序。

除了标准编组,DCOM 还支持自定义编组。使用它,开发人员能完全决定代理如何与和它有联系的对象通信。问题是,当使用自定义代理时,引用传递该如何进行呢?使用标准编组时,是假设接收方可获得所有编组代码,这样的话,传递一个接口的 IID 就足以

加载代码并为接口创建一个代理。

本质上讲,用自定义编组来传递对象引用就是这样进行的:先将发送者的代理编组,将这个被编组的代理传送到接收端,再将它解编。注意,这个过程与我们在第2章里讨论的Java中传递对象引用很类似。因为编组是以与特定对象相关的方式完成的,所以接收进程需要有相应的代码,将传过来的代表编组代理的字节序列解编。这些代码假设是接收方可获得的,并通过其CLSID来标识。图9.26给出了DCOM中的对象引用传递。

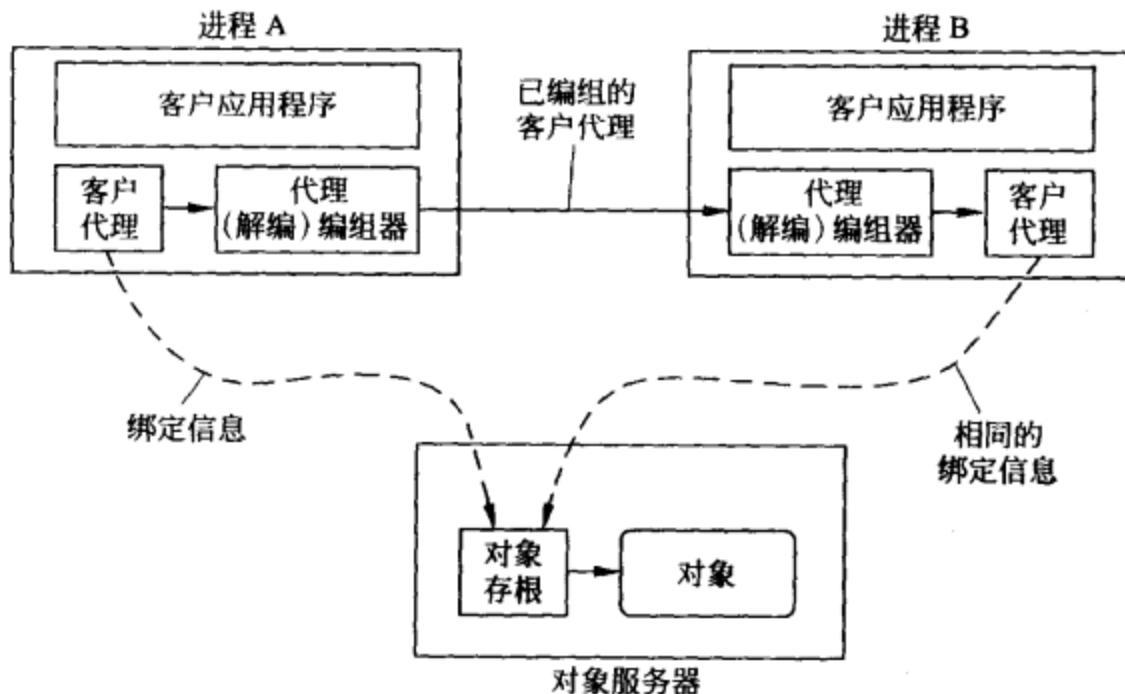


图9.26 用自定义编组传递DCOM对象引用

这里有两种情况,即“用于编组和解编的代码”和“代理自己完成的编组”,注意它们之间的细微差别是很重要的。在后一种情况下,编组的完成是与方法调用有关的,而这种方法调用是包含在由代理实现的接口中。代理负责将调用转换为消息,同样,也将包含调用结果的消息解编。而编组代理处理的是将代理的代码变换为一系列字节,并将它通过网络传送出去。

2. DCOM服务器

CORBA的可移植对象适配器提供的各种选项都被硬编码嵌入各个DCOM对象,在这个意义上,DCOM对象或多或少是自包含的。DCOM提供的是标准的激活对象的方法,即使这个对象位于远程主机上。

一个支持DCOM对象的主机将会执行前面提到的特殊的SCM(服务控制管理器)。SCM负责激活对象,其步骤如下:假设客户持有一个类对象的CLSID。为了实例化一个新的对象,它将CLSID传给一个本地注册表,在那里查询应该放置对象服务器的主机。然后CLSID就会被传送到那个主机上的SCM上。这个SCM就会在它的本地注册表中查询CLSID,查找文件,从而加载类对象,并从中实例化新的对象。

SCM通常开启一个新进程来加载指定的类对象,随后创建对象实例。新创建的服务器监听输入的调用请求的端口由SCM注册,与之一起注册的还有新创建对象的标识符。

这个绑定信息会返回给客户,此时客户可以直接联络对象服务器而不需要受 SCM 更多的干涉。

这个过程本质上把与管理服务器上对象相关的所有问题都留给了开发者。为了使事情简单些,DCOM 也提供对象管理的工具。这类工具之一是 JIT 激活(JIT activation)。JIT 激活,即及时激活,是这样一种机制,服务器通过它控制激活和撤销对象。JIT 激活按如下过程作用。

通常,对象是由客户从一个给定类中创建新对象实例的请求激活的。这个对象的服务器将其保留在内存中,直到没有任何客户持有此对象的引用为止。但是,通过 JIT 激活,服务器能在任何时候决定撤销一个对象。例如,如果服务器发现内存用完了,就可能撤销一些对象而为新对象留出空间。

当客户要调用一个已被撤销的对象的方法时,服务器就会迅速从同一个类中创建一个新对象,并调用那个对象的被请求的方法。当然,这个方法只对无状态对象起作用。无状态对象是指在调用之间不保存任何内部状态的对象。剩下的就是只有方法的对象。在面向对象编程之前,这样的对象用于调用库函数。如果想要保存状态,每次调用之后都要将它写进磁盘,新创建对象时又要重新加载。JIT 激活可以改进对象服务器的可扩展性,但它并没有使开发人员的工作更轻松。

9.2.4 命名

DCOM 本身只为对象提供相对低层的命名。在前文中,我们已经讨论过怎么使用接口指针这种基本的方法来引用对象。接口指针只对于使用它的那个进程有意义,而那个进程可以随时清除它。除了接口指针以外,DCOM 还提供了可供多个进程共享的持久性对象引用。这种引用叫做标记(moniker),稍后将对之进行讨论。DCOM 没有高层命名服务。但是,作为 Windows 2000 的一部分,DCOM 对象使用一种真正的目录服务——Windows 活动目录。在这部分中,我们将仔细研究一下活动目录。

1. 标记

与 CORBA 相比,基本的 DCOM 对象模型只支持暂时对象。换句话说,当一个对象不再被引用时,它就被清除了。为了在最后一个客户结束使用这个对象后再延长一些其使用时间,DCOM 提供了一种持久性引用。这种引用,在 DCOM 中就叫标记(moniker),可以存储在磁盘中。标记包含了所有重建这个被引用对象的必要信息,并能将此对象恢复到最后一个用户使用后的状态。

DCOM 提供各种不同种类的标记。其中重要的一种是文件标记(file moniker),它引用从本地文件系统的文件建立的对象。在这种情况下,标记将包含用于创建对象的文件的路径名。除此之外,它还包含一个类对象的 CLSID,而对象实际上都是需要从这个类对象建立的。

像其他标记一样,一个文件标记提供给客户一种称之为 BindToObject 的操作,即绑定到与标记相关的对象上的操作。使用文件标记的绑定操作步骤如下,如图 9.27 所示。

步骤	执行者	描述
1	客户	调用 BindMoniker 别名
2	标记	查找相关 CLSID 并指示 SCM 创建对象
3	SCM	加载类对象
4	类对象	创建对象并向别名返回接口指针
5	标记	指示对象加载以前存储的状态
6	对象	从文件加载它的状态
7	标记	向客户返回对象的接口指针

图 9.27 通过文件标记绑定到一个 DCOM 对象

首先要认识到,标记是持久性对象。在 DCOM 的术语中,它的意思是每个标记都提供一个接口,此接口包含将它的内容存储到磁盘的方法。在很多情况下,应用程序会将标记本身作为文件的一部分来存储。稍后,当要再次读取那个文件时,标记会重建为真实的 DCOM 对象。此时,提供给用户的是 Imoniker 接口,这个接口包含 BindToObject 操作。

假设一个客户刚刚从磁盘上读取了一个文件标记,而现在它正在请求一个被这个标记引用的对象的接口。在调用该标记的 BindToObject 时,标记代码就查询类对象来寻找本地注册表里与标记相关联的对象。这个类对象是由标记的一部分——CLSID 确定的。严格遵循以前讨论过的实例化对象的步骤,将这个 CLSID 传到本地的注册表中。从那里开始,CLSID 又被传到合适的 SCM,然后,SCM 将负责创建那个类的对象。

别名绑定代码将使用从由标记命名的文件中找到的数据对新创建的对象进行初始化。使用文件别名时要假设对象提供实现加载文件操作的接口。换句话说,假设对象本身知道怎样从它加载的文件的数据里回复它先前的状态。注意,这个文件完全独立于存储标记的那个文件。

最后,客户将得到指向它最初请求的接口的指针,绑定操作完成。

作为绑定的一个副作用,标记将在客户机器的运行对象表(running object table, ROT)中注册它的相关对象。无论何时另一个客户用标记绑定到相关对象,这个标记都会先在 ROT 中查询此对象。如果它注意到对象已经建立,就会继续绑定到那个实例上。使用这种方法,同一台机器上的不同进程就可以共享对象了。

除了文件标记之外,DCOM 也提供一些其他的标记类型,其中部分类型如图 9.28 所示。除了这些事先定义的标记类型,也可以创建自定义的标记。实现标记的细节参见文献(Eddon 和 Eddon 1998)。

别名类型	描述
文件别名	从文件创建的对象的引用
URL 别名	从 URL 创建的对象的引用
类别名	类对象的引用
复合别名	复合别名的引用
项目标记	一个复合结构的别名的引用
指针标记	对远程进程中的对象的引用

图 9.28 DCOM 定义的标记类型

2. 活动目录

随着 Windows 2000 的引入, DCOM 应用软件可以使用一种特有的世界范围的目录服务——活动目录(active directory)。回忆一下, 目录服务就是允许客户基于一组必需的性质来查询实体。例如, 目录服务可以允许诸如“返回每个运行 UNIX 的 Web 服务器的名字”的查询。在这个意义上, 目录服务是与黄页电话号簿类似的。与之对照, 仅当给出全名时才返回信息的命名服务则类似于电话簿的白页。

直到引进了活动目录, DCOM 许多的命名和目录服务才在各种 DCOM 和 Windows 组件中普及, 比如注册表、类型库、标记等。尽管活动目录不是 DCOM 的正式部分, 但简要描述一下它的机制及如何使用它命名和查询对象是很有帮助的。

活动目录与基于 Windows 2000 的分布式系统体系结构有很密切的联系。假设一个这样的系统按域(domain)划分, 每个域包含一定数量的用户和资源。每个域有一个或更多的域控制器(domain controller), 即记录这个域的用户和资源的本地目录服务器。每个域都有 DNS 名, 例如 cs. vu. nl。

每个域控制器都是按一个 LDAP 目录服务器来组织的。正如在第 4 章中所说明的, LDAP 描述的是 X.500 目录访问协议的基于互联网的版本。在活动目录中, 每个域控制器都有一个很大的事先定义实体及其属性组织的规划。例如, 有为了注册用户、主机、指针及类似内容的事先定义的实体。此规划可以在需要的地方扩展。

在第 4 章中已经介绍过, 域控制器使用 DNS SRV 记录注册为 DNS 中的 LDAP 服务。例如, cs. vu. nl 域的 LDAP 服务器可以通过使用 DNS SRV 在 ldap. tcp. cs. vu. nl 的名下注册。反过来, 这个名字将代表相关的域控制器的 DNS 主机名。

现在为给定域寻找一个域控制器是很直接的。客户只需要简单地向 DNS 查询那个域的 LDAP 服务器的主机名, 然后就可以连接到那个服务器查询用户和资源或请求更新。图 9.29 给出了这种通用的体系结构。

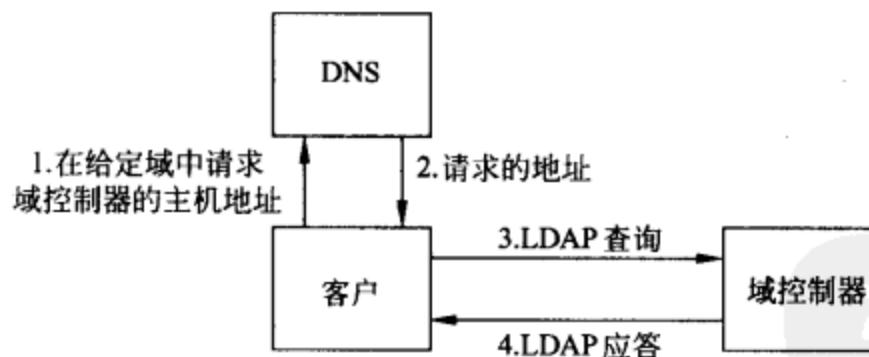


图 9.29 活动目录的通用体系结构

但到目前为止, 上面勾画的方法仍存在一个问题, 就是客户在查询之前必须知道它要去那个域。例如, 如果客户想找到所有运行 UNIX 的 Web 服务器, 就必须先连接到一个特定域的域控制器, 然后再发出一个查询请求。如果这个查询请求覆盖了几个域, 客户就不得不分别联系每个域的控制器, 然后再将各个结果合并起来。

为了把事情简化, 活动目录允许将不同的域集合成树和森林。一个域树(domain

tree)在 DNS 名称空间中形成一个子树,然而一个域森林(domain forest)只是简单地将一定数量的独立域集合到一起。域组的好处就是活动目录为那个组提供一个单独的全局索引,叫做全局目录(global catalog)。域组中域的实体都被复制到这个目录中,一起被复制的还有其最重要的(属性,值)对。这样,客户就可以简单地在目录中发送一个查询请求来查找域组了。

关于 Active Directory 的更多信息,以及关于配置、复制和安全的管理可以在文献(Lowe-Norris 2000)中找到。

9.2.5 同步

DCOM 最主要的同步机制是以事务机制的形式实现的。这个机制将在下面我们考虑 DCOM 的容错性细节时同时讨论。

9.2.6 复制

DCOM 应用程序中没有特别的对联合复制的支持。复制只能在特殊服务(如活动目录)中使用。而对于用户应用程序,开发者就得建立自己的解决方案了。

9.2.7 容错性

DCOM 的通用容错性主要是用 COM+ 的一个完整部分——自动事务处理(automatic transaction)来支持的。自动事务处理允许开发人员指定一系列的可能对不同对象的方法调用,这些调用可以被编组成一个事务。将一个单独的方法调用作为一个事务来实现也是可行的。

对象的事务是自动创建的,依赖于各个类对象如何配置。每个类对象有一个事务属性决定它的对象的事务处理行为。图 9.30 给出了可能的属性值。

属性值	描述
REQUIRES_NEW	总是在每个调用开始时开启一个新事务
REQUIRED	开启一个新事务(如果还没开启的话)
SUPPORTED	加入一个事务(只要调用者已是其中一部分)
NON_SUPPORTED	不加入事务
DISABLED	永不加入事务,即使被告知这样做

图 9.30 DCOM 对象的事务属性值

如果事务属性被设置为 REQUIRES_NEW,无论何时只要对象第一次被调用,它就自动开启一个独立于任何其他事务的新事务。此外,如果调用者已经参与了一个事务,那么请求新事务的对象就会开启另一个与调用者原有事务并行的事务。

当事务属性设置为 REQUIRE,情况就不同了。对象将参与这个调用者的事务(如果有的话)。否则,它将开启一个新事务。在任何情况下,调用都会作为一个封装事务的一部分而完成。

当一个事务的属性是 SUPPORTED 时,对象就不会开启一个新事务,而只能参与和

它的调用者相同的事务。如果这个调用对象或客户没有参与事务,那么对此对象的调用也可以以一种正常的、非事务的方式进行。

当事务不被支持时,对象调用将在调用者正在参与的可行事务之外进行。换句话说,对原子性这样的要求是不会做出保证的。一个类对象的事务属性的默认值总会被置为 NOT_SUPPORTED。

最后,当一个对象声明事务是 DISABLED 的,那么无论客户是否希望,这个对象都不会参与事务。禁用事务比声明对象不支持事务要严重。在后一种情况下,调用的客户仍可以决定是否将被调用对象与事务结合。当事务支持被禁用时,这是不可能的。

自动事务处理通常是由一个独立的事务管理器来实现的。在很多情况下,这样的管理器是作为 Windows 环境的一部分来提供的,即 Microsoft 分布式事务协调器 (distributed transaction coordinator, DTC)。DTC 是一个相当标准的事务管理器,用于实现在第 7 章讨论的两阶段提交协议。

9.2.8 安全性

与 CORBA 类似,DCOM 的安全性也是致力于保护对象的调用。但是,它处理安全的方法是完全不同的。尽管 DCOM 大部分被结合到 Windows 操作系统中,但 DCOM 的开发人员还是有理由选择尽可能多地将 DCOM 的安全性独立于 Windows 提供的安全性服务之外。这样就保留了在其他操作系统上实现 DCOM 时,仍能在安全的方式下处理调用的可能。

在 DCOM 中有两种处理安全性的方法。一个重要的角色是为注册表保留的,它为安全性提供一个声明的方法。除此之外,如果应用程序本身能够处理访问控制和身份验证问题,安全性也可以用编程的方法来处理。下面将讨论这两种方法。

1. 声明安全性

声明安全性本质上可归结为在注册表中指定与激活、访问控制和身份验证相关的组件(比如对象)的要求。DCOM 安全性模型是基于角色的使用的。准确地说,与角色识别相关的所需凭证也可以在注册表中指定。

激活和访问控制是直接的。对每个注册类,注册表将会指定允许哪些用户或用户组创建其实例,即对象。同样,注册表会为它的实体保存一个访问控制表,为用户或用户组指定访问权。

用户验证等级	描述
NONE	没有验证要求
CONNECT	对第一次连接到服务器的客户进行验证
CALL	在每次调用时验证客户
PACKET	验证所有数据包
PACKET_INTEGRITY	验证数据包并进行完整性检查
PACKET_PRIVACY	验证、完整性检查,并将数据包加密

图 9.31 DCOM 的验证等级

DCOM 区分几个验证等级,如图 9.31 所示。大多数系统都默认地要求客户第一次连接到服务器时进行验证(CONNECT 级)。但是,在每次调用时都执行验证(CALL 级),甚至每次从客户那里接收数据时执行验证(PACKET 级)都是可能的。等级 PACKET_INTEGRITY 和 PACKET_PRIVACY 实际上并不真是验证等级,只是分别在 PACKET 验证上加上了完整性检查和保密性。

服务器在何种程度上采用与调用其对象的客户相同的角色和凭证也是可以指定的,如图 9.32 所示。模仿(implementation level)用于保护客户免受恶意服务器的伤害。将模仿等级设置为 ANONYMOUS,服务器就查不出实际上是谁在调用,因而也就无法模仿这个客户。

模仿等级	描述
ANONYMOUS	对象对服务器是完全匿名的
IDENTIFY	服务器认识此客户并可以进行访问控制检查
IMPERSONATE	服务器可以代表客户调用本地对象
DELEGATE	服务器可代表客户调用远程对象

图 9.32 DCOM 模仿层次

如果客户想要访问资源,它就必须允许服务器进行访问控制,这就需要模仿等级 IDENTIFY。这个等级只对服务器显示客户的标识符,但这个标识符应该足够让服务器来检查必要的访问权限。

当服务器被允许模仿客户(IMPORSONATE 等级)时,对这个服务器的本地对象的调用就可以进行了,此时服务器会使用客户的凭证。如果调用需要跨越机器的界限,等级就应该设置为 DELEGATE,从而允许对访问权限的真实委派。

2. 编程安全性

除了声明安全性,也可以让应用程序设置安全性层次,并在不同的安全服务间进行选择。这样的设置对临时提高在注册表中声明的安全性可能是有必要的。

在进程初始化期间调用 DCOM 的 ColnitializeSecurity 函数,从而调整安全性设置。进程应该对它要用来调整前面讨论过的验证和模仿层次的组件(如类对象或对象)进行确认。

编程安全性的一个有趣的特点是进程也可以引用它期望使用的安全性服务来处理安全性。通常,这样的服务来自 DCOM 之下的操作系统。DCOM 区分 5 种身份验证服务,如图 9.33(a),以及 3 种授权服务,如图 9.33(b)所示。两类服务的数目均能很容易地扩展。服务的可用性不由 DCOM 指定,而将依赖于底层操作系统。例如,在 Windows 2000 中,诸如 SSL 服务和使用不对称密码系统的 Kerberos 变种就很容易获得。

服务	描述
NONE	无身份验证
DCE_PRIVATE	基于共享密钥的 DCE 身份验证
DCE_PUBLIC	基于公共密钥的 DCE 身份验证
WIN_NT	Windows NT 安全性
GSS_KERBEROS	Kerberos 身份验证

(a)

服务	描述
NONE	无授权
NAME	基于用户身份的授权
DCE	使用 DCE 特权属性证书(PAC)的授权

(b)

图 9.33 DCOM 支持的默认身份验证服务和授权服务
(a) DCOM 支持的默认身份验证服务; (b) DCOM 支持的授权服务

通过调整激活设置使客户被允许创建对象实例是不可能的。这样的信息只能用声明的方法来设置,因为在创建对象的机器上的 SCM 总是会检查注册表,查看客户对于激活的许可是什么。但是,在进程之间委派激活权限是可能的。有关细节可以在(Eddon 和 Eddon 1998)找到。

9.3 Globe

基于对象的分布式系统的最后一个例子是 *Globe*。*Globe* 是基于对象的全局环境 (global object-based environment) 的简称,它是一个实验性的分布式系统,本书的作者及其同事在阿姆斯特丹 Vrije Universiteit 正在对之进行开发。使它区别于 COBRA 和 DCOM 等系统的一个重要的设计目标就是它旨在支持大量遍布在 Internet 上的用户和对象,但仍提供分布透明性。相比之下,其他大多数基于对象的分布式系统主要都设计成在局域网上实现。

Globe 概述可以在文献(van Steen 等 1999a)中找到。Homburg(2001)则给出了它的体系结构的详细描述。

9.3.1 *Globe* 概述

在 *Globe* 这个基于对象的分布式对象系统中,可扩展性扮演了中心的角色。创建可支持海量用户和对象的大规模广域系统有关的各方面需求推动着 *Globe* 的设计。这个方法的基本出发点在于看待对象的方式。与其他基于对象的系统类似,*Globe* 中的对象也期望封装其状态,并在该状态下运行。

与其他基于对象的系统,特别是那些也以广域可扩展性为目标的系统,比如 Legion (Grimshaw 等 1998, Grimshaw 等 1999)等的一个重要区别是对象也要封装那些规定对象状态在多机器上的分布策略的实现。换句话说,每个对象决定它的状态如何在副本上

分布。对象也可控制自己在其他方面的策略。

总的来说, Globe 中的对象尽可能多地承担责任。比如说, 对象决定它的状态如何移植, 何时移植, 以及移植到哪里。而且, 对象还决定它的状态是否复制, 以及应该怎样复制。除此之外, 对象还可决定它的安全策略和实现。下面的章节将描述如何实现这样的封装。

1. 对象模型

与其他大多数基于对象的分布式系统不同, Globe 不采用远程对象模型。相反, Globe 对象是可物理分布的, 即对象的状态可在多进程间分布和复制。其体系结构如图 9.34 所示。图中显示了一个分布在 4 个进程上的对象, 每一个进程又都分别运行于不同机器上。Globe 对象是作为分布式共享对象(distributed shared object)引用的, 这反映对象通常是在几个进程间共享的。Globe 对象模型起源于 Orca(详情见 Bal 1989)中使用的分布式对象, 相似的对象模型也曾在 SOS 操作系统(Shapiro 等 1989, Makpangou 等 1994)中使用过。

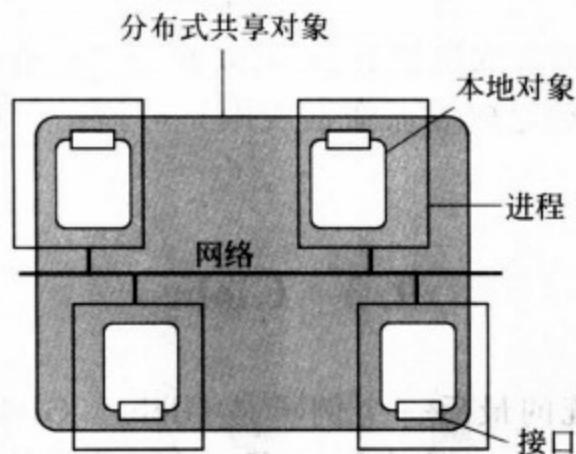


图 9.34 Globe 分布式共享对象的体系结构

绑定到分布式共享对象的进程可获得由那个对象提供的接口的本地实现。这样的本地实现称为本地代表(local representative), 或简单地说成本地对象(local object)。原则上, 无论是不是本地对象, 都拥有对绑定的进程完全透明的状态。对象的所有实现细节都隐藏在提供给进程的接口后面。

每个本地对象实现一个与 DCOM 中使用的 IUnknown 接口类似的标准对象接口, 该接口称为 SOInf。特别是, 像 DCOM 中的 QueryInterface 一样, SOInf 提供方法 getInf 可接受接口标识符输入, 并返回指向那个接口的指针, 只要那个接口是由此对象实现的。Globe 本地对象与 DCOM 对象间还有其他的相似点。例如, 每个本地对象都假设有一相关的类对象, 从而使新的本地对象可从中创建。

假设本地对象主要由指针函数表组成的二进制接口实现。接口规范由接口定义语言支持, 通常与 CORBA 和 DCOM 中使用的类似, 但本质上在很多地方都有区别。

Globe 本地对象有两种类型。基本本地对象(primitive local object)不包含其他本地对象。与之相对, 组合本地对象(composite local object)则包含多种(可能是组合)本地对象。为了支持组合, Globe 本地对象的接口表由(状态, 方法)对指针组成。状态指针指向

属于某特定本地对象的数据。对于基本本地对象的接口,所有的状态指针都将指向代表那个对象状态的相同的数据;而在组合的情况下,状态指针指向组成这个组合的不同对象的状态。

组合可用来创建实现分布式共享对象所需的本地对象。这个本地对象包括至少 4 个子对象,见图 9.35。

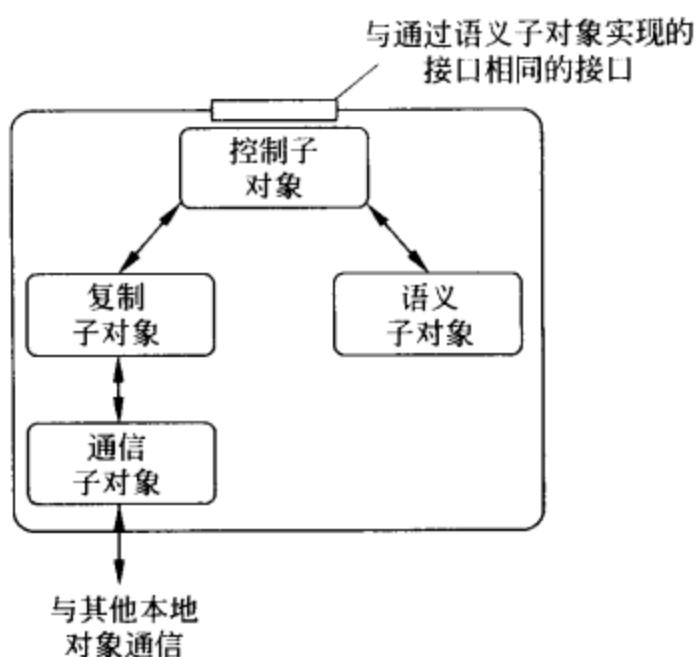


图 9.35 Globe 用于实现分布式共享对象的本地对象的通用体系结构

语义子对象(semantics subobject)实现由分布式共享对象提供的功能。例如,Globe 曾用于建立 Web 站点,在那里,一些逻辑上相关的网页、图标、图像等的集合组成一个单独的文档。这样的文档,称为 GlobeDoc(van Steen 等 1999b),通过语义子对象的方式来本地实现,这个子对象提供图 9.36 所示的接口。每个用于 Web 文档的文件都作为 GlobeDoc 对象的元素返回。元素可通过 document interface(文档接口)添加或删除。这个接口也提供方法返回所有元素的表。Web 文档假设是按有根图组织的。根元素,类似于许多 Web 应用程序中的缺省的 index.htm 文件,可以通过独立的方法设置和引用。实践证明这些接口是适当的。

每个元素都简单地表示为一个字节数组。content interface(内容接口)提供获得元素当前内容的方法,并通过这样的数组替换内容。property interface(属性接口)提供方法将元数据与元素相连。元素的元数据是以(属性,值)对的形式出现的。例如,每个 GlobeDoc 对象的元素有一相关的 MIME 类型(MIME 类型将在第 11 章讨论)。

最后,为了允许对元素进行实时修改,每个 GlobeDoc 对象都实现了 lock interface(锁接口)。这个接口说明了 Globe 的实时控制方法。考虑到大多数 Web 文档是由相对少数人来维护的,所以提供一种简单的锁定机制已足够。当一个元素需要修改时,首先要把它检出(check out)。检出元素在效果上与为修改而将元素锁定是相同的。读取仍然是可能的。当修改完成时,元素会被检入(check in),使修改生效。

我们接着讨论分布式共享对象的本地对象(如图 9.35 所示)。通信子对象(communication subobject)用于向底层网络提供标准接口。这个子对象为面向连接或无

连接的通信提供一些传递信元。也有更高级的通信子对象用于实现多点传送接口。通信子对象可以用于实现可靠通信，而其他的只提供非可靠通信。

document interface(文档接口)	
方法	描述
AddElement	向当前元素集合中添加一元素
DeleteElement	从 Web 文档中删除一元素
AllElements	返回文档中当前元素列表
SetRoot	设置根元素
GetRoot	返回根元素引用

content interface(内容接口)	
方法	描述
GetContent	以字节数组形式返回元素内容
PutContent	用给定字节数组替换元素内容
PutAllContent	替换整个文档的内容

property interface(属性接口)	
方法	描述
GetProperties	返回元素的(属性,值)对的列表
SetProperties	给一个元素提供(属性,值)对的列表

lock interface(锁接口)	
方法	描述
CheckOutElements	验出需要修改的元素序列
CheckInElements	验入改正过的元素序列
GetCheckedElements	获得当前正在检查的元素列表

图 9.36 由 GlobeDoc 对象的语义子对象实现的接口

事实上，对所有的分布式共享对象都至关重要的是复制子对象 (replication subobject)。这个子对象实现对象的实际分布策略。在通信子对象中，接口是被标准化的(将在本部分中稍后讨论)。复制子对象负责准确决定由语义子对象提供的方法何时执行。例如，一个实现主动复制的复制子对象将保证所有的方法调用在每个副本中都以同样的顺序执行。这样，子对象将不得不与组成分布式共享对象的其他本地对象的复制子对象通信。

控制子对象 (control subobject) 是语义子对象的用户定义接口和复制子对象的标准接口之间的媒介。除此之外，它还负责将语义子对象的接口输出到被绑定到分布式共享对象的进程。所以那个进程请求的方法调用都通过控制子对象编组并传递给复制子对象。

最后，复制子对象将允许控制子对象执行调用请求并将结果返回到进程。同样，远程进程的调用请求最后也被传递到控制子对象。这样的请求随后被解编，然后由控制子对

象执行调用，并将结果返回给复制子对象。Globe 中复制的细节还会在后面讨论。

2. 进程到对象的绑定

与 CORBA 和 DCOM 相比，Globe 既不提供接口仓库，也没有实现仓库的等价机制。这两个服务的缺乏部分源于 Globe 支持的对象模型。特别是，当进程绑定到一个对象时，它将在其地址空间内加载一个特定的本地对象，此地址空间是由它绑定的分布式共享对象指定的。现在我们来考虑进程到对象的绑定在 Globe 中是如何进行的。

Globe 中的绑定由 5 个不同的步骤组成，如图 9.37 所示。除了最后一步，每一步都返回下一步需要的信息。因此，如果第 k 步的信息容易得到，绑定就从该步开始，这样可以从整体上提高绑定的效率。

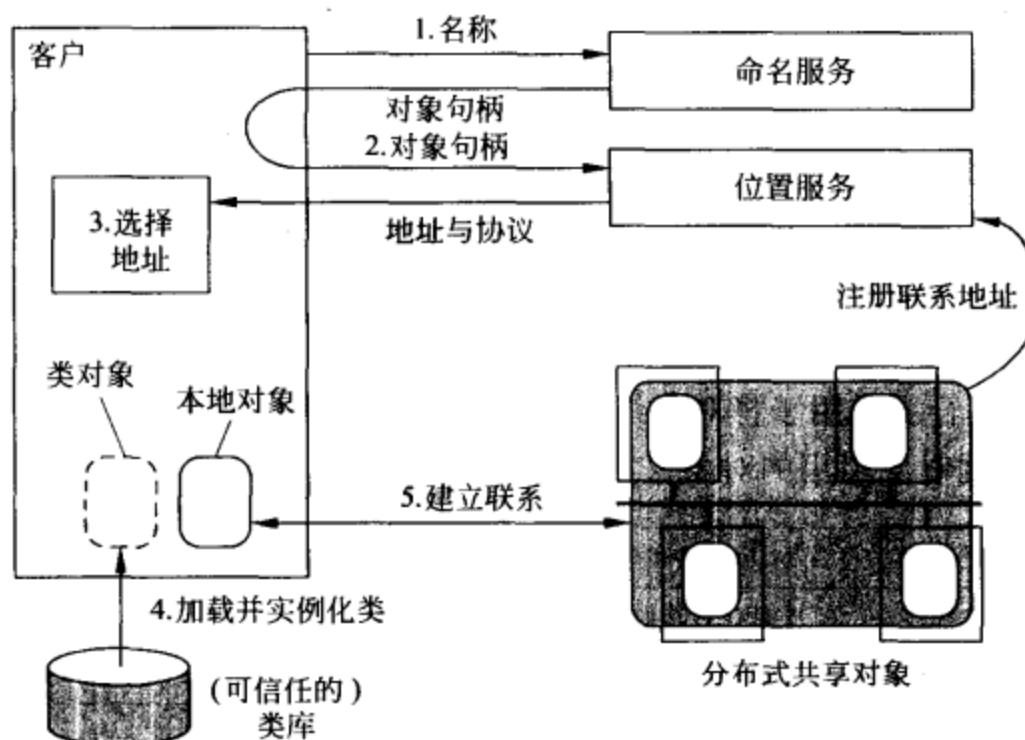


图 9.37 将进程绑定到 Globe 对象

完整的绑定以向命名服务提供(人可读的)名字为起点，如图 9.37 所示。Globe 提供基于 DNS 的命名服务，下面将对它进行讨论。这个服务返回一个全局惟一且地址无关的对象句柄(object handle)。对象句柄在 Globe 中是作为系统范围的对象引用来使用的。这个引用可用于通过使用 Globe 定位服务(见图 9.37 中的步骤 2)来定位对象。注意 Globe 以第 4 章讨论的方式分别独立实现其命名服务和定位服务。这种分别实现的好处是名字和地址可独自改变而不会影响名字到地址的映射。

Globe 定位服务为给定对象返回一组联系地址(contact address)。联系地址准确描述对象在哪里和如何到达。对象可能拥有多个联系地址，比如因为它被复制或因为它支持不同的通信协议的缘故。正如我们稍后将描述的细节所示，联系地址类似于 CORBA IOR。给定一联系地址，进程就拥有了绑定到对象的所有必需的信息。

因为定位服务可能对一个单独的对象返回一组联系地址，进程将首先选择其中一个地址。这个选择见图 9.37 的步骤 3。选择操作很简单，可以从返回的这批地址中任意挑

一个,或可以使用一些选择准则,如地址的距离或当绑定到特定地址时期望的服务质量。

每个联系地址包含了进程为了连接对象而在本地实现所需的信息。特别是,它精确地指定进程应该加载的本地对象。例如,联系地址可能包含含有进程应该加载的类对象的文件的 URL。此方法类似于 Java 中所做的类加载。从(可信任的)类库(class repository)加载和实例化一个本地对象如图 9.37 的步骤 4 所示。注意类库可以是一个简单的文件系统,可在远程站点获取,也可通过 FTP 或其他文件传输协议访问。

注意,从它们二者都提供对象实现的角度而言,类库与 CORBA 的实现仓库是类似的。但是,在 Globe 中,类库更像一个可取得代码的传统存储系统。在 CORBA 中,它是一个可以请求在对象服务器上创建和实例化对象的进程。

第 5 步即最后一步,包括初始化本地对象,和随后通过这个本地对象与形成分布式共享对象的其他本地对象进行联系。

3. Globe 服务

相比于 CORBA 和 DCOM,Globe 提供相对较少的服务。实际上,它只实现了那些不能由对象单独提供的服务。Globe 的这一特点最好这样来解释:考虑一些由 CORBA 和 DCOM 提供的服务,并让它们在 Globe 中实现。图 9.38 提供一个在分布式系统中常用服务的概述。

CORBA 的集合服务于将对象组成链表、队列等。典型情况下,这样的服务可通过一个状态包含一批对象引用的对象来实现。在 Globe 中,这样的服务也将这样实现。

服务	Globe 中的可能实现	可用性
集合	拥有对其他对象引用的独立对象	否
并行性	每个对象实现它自己的并行控制策略	否
事务	扮演事务管理者的独立对象	否
事件/通知	每个事件组的独立对象(像 DCOM 中一样)	否
外部化	每个对象实现它自己的编组程序	是
生命周期	与每个对象实现结合的独立类对象	是
许可	由每个对象独自实现	否
命名	由一命名对象集合实现的独立服务	是
属性/交易	由目录对象集合实现的独立服务	否
持久性	基于每个对象实现	是
安全性	与(本地)安全服务结合,按对象实现	是
复制	基于每个对象实现	是
容错	按与容错服务器结合的对象实现	是

图 9.38 典型的分布式系统服务的 Globe 实现的概述

CORBA 的并行性控制是按照一个接口的集合的形式来指定的,这些接口用于访问各种类型的锁定(OMG 2000c)。在 DCOM 中,并行性控制部分由事务服务管理,部分通

过对象服务器处理线程管理。在 Globe 中，并行性控制典型的处理是基于每个对象的。换句话说，需要对并行访问保护其状态的共享对象将提供一个独立的锁定接口。Globe 没有定义并行控制的标准接口。

像集合一样，事务也将对象分组。Globe 中，典型的事务是通过一个独立的作为事务管理者的对象实现的。这样的对象接口也还没有标准化。

Globe 没有事件服务。正如我们将在下面讨论的，Globe 对象总是被动的，这意味着它们没有控制自己的线程。因此，在事件发生时，对象不能向客户通报。在 Globe 中实现事件服务将要求遵从 CORBA 的拉式模型。在那种情况下，事件将组成一个单独的对象，与 DCOM 中使用的事件类是类似的。到现在为止，Globe 还没有使用事件。

外部化或编组都是基于每个对象处理的典型服务。Globe 中大多数对象需要实现本身的编组工作，通过编组，状态可以在不同机器上传输。在下面讨论 Globe 对象模型时我们将再次讨论编组。

生命周期服务提供创建、删除、复制和移动对象的手段。在 Globe 中，对象明显不能做的惟一的事情是创建它自己。所有其他的操作都由对象自己实现。Globe 不提供特殊的工具来删除、复制或移动对象。创建对象通常是直接访问对象服务器，然后请求它建立一个对象的实例。这个方法与 DCOM 中的许可(Licensing)服务有些相似。

Globe 不支持许可(Licensing)服务，但是可基于各个对象的情况来实现。

命名服务是已命名对象不能实现的服务的一个好例子，它用于查询对象。换句话说，对命名服务的访问需要在被命名对象能被访问之前。当然，命名服务自身可用对象实现，比如 CORBA 就可这样做。Globe 有一个独特的命名服务，我们将在下面详细讨论。由于同样的原因，属性(property)或交易(trading)服务也可以实现，但要与维护引用的对象分开。

许多分布式系统的持久性(persistence)服务的形式是文件系统或数据库。Globe 中不存在这样的服务。持久性被认为是对象的属性，因此也要由那个对象实现。持久性如何实现本质上是由那个对象决定的。但是，允许对象永久性存储其状态是需要支持的。在 Globe 中，这种支持主要通过对象服务器提供，但是大多对象实现也使用本地可用的文件系统。

Globe 中的安全性(Security)部分基于每个对象实现，部分通过本地和全局安全服务实现。例如，虽然对象本身可以处理大多数方法调用的访问控制，但是获取和验证密钥还是需要特殊的服务来处理。后面我们还会讨论安全性。

当考虑到诸如复制之类的问题时，Globe 和其他基于对象的分布式系统的主要区别就趋于明显了。正如我们曾提到过的，Globe 对象决定它的状态如何复制。换句话说，复制完全是基于每个对象处理的。与之相比，诸如 CORBA 之类的系统则使用复制服务器来控制一组对象的复制。Globe 中实现复制的细节将在稍后讨论。

最后，Globe 中的容错也是基于每个对象进行处理的，尽管也需要一些容错对象服务器的支持来成功地掩盖失败。稍后我们将继续讨论容错性。

9.3.2 通信

与 CORBA 和 DCOM 不同,除了同步方法调用,Globe 没有提供其他任何通信工具。当调用完成时,即当进程从作为对本地可用接口的方法调用来实现的调用中返回时,分布式共享对象内部不会再进行任何动作,除非有其他调用引起的动作。在这种意义上,Globe 对象被认为是被动的。

为了理解被动对象的意义,设想一个被三个进程共享的分布式对象,如图 9.39 所示。进程 B 和 C 各有一个由语义子对象的隐藏版本指定的副本。进程 A 没有任何状态,事实上,甚至没有语义子对象。进程 A 的地址空间里的本地对象对应于传统的客户端存根程序或代理。

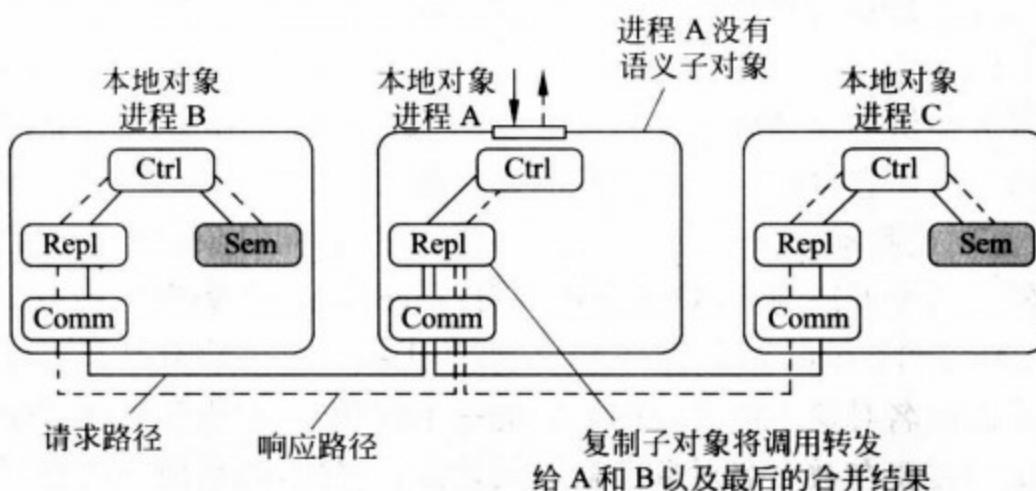


图 9.39 Globe 中使用主动复制调用对象

假设进程 A 调用对象。在一个简单的复制之后,这个调用请求被分别传送到实线代表的路径指定的进程 B 和 C。当请求到达通信子对象(比如说进程 B)时,一个线程就会被激活来处理此请求。最后,这个线程将调用语义子对象,返回一独立的结果信息并送往进程 A。进程 C 中的活动也是类似的。

在整个调用期间,作实际的方法调用的进程 A 中的线程保持阻塞。当其他进程的结果都被返回到 A 的复制子对象后,线程解除阻塞,继续运行。如果假设没有发生其他的调用,则一旦控制返回到发生调用的应用程序中,组成分布式共享对象的任何本地对象都将不再有任何活动。

Globe 不支持客户应用程序的回叫机制。只有允许作为到来请求的结果而激活的线程执行回叫时,才提供这样的支持。但是,回叫可能导致线程有不可预料的行为。当需要由一批紧密协作的复制子对象来保持其一致的复制存在时,这种行为是很危险的。在这里可看到一个例子,相对于功能的丰富性,Globe 的开发者更强调简单性。通常,简单性是 Globe 一个主要的设计目标。

9.3.3 进程

尽管在原理上 Globe 中的客户和服务器没什么区别,但是在组织分布式共享对象时做一下这样的区分是很有用的。客户进程对对象调用初始化,而服务器进程主要只响应

调用请求。在下面的内容中,这个区别将更加明确。

1. Globe 客户

Globe 客户是一个被绑定到分布式共享对象的进程,它通过调用由控制子对象接口获得的调用方法来调用那个对象。图 9.39 所示的进程 A 就是一个 Globe 客户的典型例子:没有状态,但连接到有状态的副本。

很多情况下,Globe 客户在它甚至没有语义子对象的意义上是比较简单的。相反,所有的调用都被直接编组并被传递到驻留在服务器的副本。这个模型与 CORBA 中使用的代理紧密呼应,也是类 CORBA 系统(Jansen 等 2001)中集成分布式共享对象的基础。但是,定制 Globe 中的客户也是可能的,方法与 DCOM 和 Java RMI 中的方法相同。比如,要求定制可能是为了允许客户缓存调用结果。

在 CORBA 中定制一个客户的标准方法是使用截取程序。在 DCOM 中,定制是通过提供有 DCOM 定制编组支持的代理的独立客户端实现来进行的。在 Globe 中,分布式共享对象通常提供本地对象实现的集合。这些实现存储在类库中。由用户决定使用哪个实现。

当客户绑定到一个对象,它就得到一些联系地址,每个地址支持一个或多个客户可加载的本地对象的实现。联系地址仅仅指定客户应使用的通信协议也是可以的。在这种情况下,只要遵守协议规则,客户可以任意使用想要的实现。特别是,客户可能只使用来自可信任类仓库中的实现。

需要为这种灵活性付出代价,即不得不为不同的本地对象,或者可能不同的操作系统和机器体系结构开发其实现。这些与异构相关的问题可通过使用由解释性语言(如 Java)写的可移植实现来解决。

2. Globe 服务器

对比 Globe 客户,Globe 服务器是一个仅可以处理从底层网络到来的调用请求的进程。换句话说,没有从控制子对象提供的接口来的方法调用。图 9.39 中的进程 B 和 C 是 Globe 服务器的实例。

Globe 也包含可支持分布式共享对象的对象服务器。这个服务器提供用于支持分布式共享对象的本地对象的基本功能。进程可以通过图 9.40 所示的接口与 Globe 对象服务器通信。本质上,每个 Globe 对象服务器在它的客户面前表现为另一个分布式共享对象。

一个重要的操作是指示对象服务器绑定到给定的分布式对象。这个服务器接收一个对象句柄以及当要对几个联系地址作选择时的选择准则。例如,服务器可决定选择任意地址,或符合特定属性的地址,如在主机—备份复制的情况下绑定到任一备份副本。其他的地址选择准则也很容易理解。

让服务器绑定到对象是一个幂等(idempotent)操作。如果服务器已经绑定到这个对象,那么任何事都不会发生。如果服务器打算加载已经绑定的分布式共享对象的另一本地对象,就要调用 AddBinding 方法。

方法	描述
Bind	将服务器绑定到给定对象,除非已经被绑定
AddBinding	将服务器绑定到对象,即使已经被绑定
CreateLR	让服务器为新的分布式对象创建一个本地对象
RemoveLR	让服务器删除一给定对象的本地对象
UnbindDSO	让服务器删除给定对象的所有本地对象
ListALL	返回所有本地对象的列表
ListDSO	返回给定对象的所有本地对象的列表
StatLR	获得特定本地对象的状态

图 9.40 Globe 对象服务器的操作

Globe 对象服务器还提供通过指示它建立本地对象来创建分布式共享对象的基本方法。创建通过调用 CreateLR,准确指定服务器应该加载的实现(例如,类对象)来完成。当一个类对象被加载后,服务器从那个类对象创建一个新的本地对象,并对之进行初始化。通常,初始化也意味着这个新对象在 Globe 定位服务中的联系地址可以使用了。因此,从此时起,其他进程就可以绑定到这个对象了。

RemoveLR 操作简单地将一特定本地对象从对象服务器中删除。通常不调用这个操作。相反,通过调用 UnbindDSO,服务器可被告知断开与分布式共享对象的绑定。断开绑定的结果就是给定对象的所有本地对象都从这个服务器上删除,即对象允许清除其本身。例如,可能需要从定位服务中删除联系地址。

注意,断开绑定并不一定意味着对象被消除了。在很多情况下,其他进程也可以被绑定到这个对象上,除非对象被明确地消除,否则它都将继续存在。与 CORBA 类似,而与 DCOM 相反,Globe 既支持暂时性也支持持久性对象。

最后,还有一些操作,用于检查服务器绑定到哪些对象,以及每个本地对象的状态是什么。

将 Globe 对象服务器绑定到分布式共享对象可能会发生两种情况。首先,绑定可能会导致本地对象成为持久性的。持久性本地对象(persistent local object)就是一种本地对象,Globe 对象服务器将它的状态编组并持久保存。保存持久性本地对象通常作为正确关闭服务器操作的一部分。当服务器稍后重启时将恢复它先前存储的本地对象。与之相反,绑定也可能导致暂时本地对象(transient local object),即不会被对象服务器持久保存的对象。因此,暂时本地对象在服务器退出时就会被永久消除。

绑定可以要求对象服务器建立一个与新安装的本地对象相关的联系地址。这样的地址将允许其他进程绑定到对象,但这个操作现在依赖于对象服务器中出现的本地对象。在这种情况下,对象服务器将在 Globe 定位服务中注册此联系地址,并处理从其他进程来的请求,接下来将讨论有关内容。

9.3.4 命名

命名在 Globe 中起着至关重要的作用。它与其他分布式系统中的名称的使用方法在

很大程度上是不同的。在 Globe 中,命名对象独立于定位对象,如第 4 章中所述。换句话说,使用单独的服务支持(易于理解的)字符串名称,区别于定位对象使用的位串名称。

1. 对象引用与联系地址

每个 Globe 分布式共享对象都被分配一个全局惟一的 OID(对象标识符),一个 256 位的串。Globe OID 是在第 4 章中定义的真正的标识符。换句话说,Globe OID 至多表示一个分布式共享对象,决不让另一个对象重新使用,而每个对象也至多有一个 OID。

Globe OID 只是用来比较对象引用的。例如,假设进程 A 和 B 都被绑定到一个分布式共享对象上。两个进程都可以请求被绑定到对象的 OID。当且仅当两个 OID 相同时,才认为 A 和 B 被绑定到了同一对象上。

为了定位对象,需要在 Globe 定位服务中查找对象的联系地址。对象在此定位服务中通过对对象句柄得到引用。对象句柄是一个位串,含有 OID 和其他一些定位服务使用的信息。它也是一个标识符,除了不保证同一个对象只有一个对象句柄之外,它也是一个标识符。要意识到 Globe 定位服务使用的对象句柄并不带有对象当前位置的任何信息或线索,这一点很重要。

Globe 定位服务是按分层定位服务的原理来组织的,如 4.2.4 节所述,在此不再重复。详细信息参见文献(van Steen 等 1998a,Baggio 等 2001)。定位服务器本身的设计与实现,以及性能评估在(Ballintijn 等 2001)中有描述。

Globe 定位服务返回联系地址,这可与 CORBA 及其他分布式系统中使用的依赖于地址的对象引用相比。忽略一些次要细节,联系地址主要有两部分。第一部分是地址标识符(address identifier),定位服务用它确定合适的叶节点,对相关联系地址进行的插入和删除操作被传递到该叶节点。因为联系地址是依赖于位置的,从一个适当的叶节点开始插入和删除这些地址是很重要的。

第二部分由实际的地址信息组成,但此地址对定位服务器是完全不透明的。对定位服务器来说,地址只是一个字节数组,既可以表示实际网络地址,也可以表示簇编组的接口指针,甚至是完整的编组代理。

Globe 目前支持两种类型的地址。栈式地址(stacked address)表示一个分层的协议簇,每层用一个三字段的记录表示,如图 9.41 所示。

字段	描述
协议标识符	表示(已知)协议的常数
协议地址	协议特有的地址
实现句柄	对类仓库中一个文件的引用

图 9.41 栈式联系地址中协议层的表示

协议标识符(protocol identifier)是一个表示已知协议的常数。典型的协议标识符包括 TCP、UDP 和 IP。协议地址(protocol address)字段包含协议特有的地址,比如 TCP 端口号,或者 IPv4 网络地址。最后,实现句柄(implementation handle)可是选择性地提

供的,用于确定协议默认实现的位置。通常,实现句柄以 URL 表示。

第二种联系地址是实例地址(instance address),它由两个字段组成,如图 9.42 所示。这种地址也含有实现句柄,仅仅是对能够找到本地对象的类仓库中一个文件的引用。当前绑定到对象的进程应加载此本地对象。

字段	描述
实现句柄	对类仓库中一个文件的引用
初始化串	用于初始化实现的串

图 9.42 实例联系地址的表示

加载遵从与 Java 中类加载类似的标准协议。加载实现并创建本地对象后,通过向对象传递初始化串(initialization string)进行初始化。

2. Globe 命名服务

Globe 使用相对简单的命名服务支持易于理解的名称(Ballintijn 2001a)。此命名服务基于 DNS 并允许名称查询和授权更新。Globe 中易于理解的名称用简单的 URI(uniform resource identifier,统一资源标识符)表示,比如 hfn://org/globeworld/dns/globesite。这个名称被解析为命名对象的对象句柄。

为了使用 DNS,Globe 名称首先在本地转换为合适的 DNS 名称。例如,Globe 名称 hfn://org/globeworld/dns/globesite 变成 DNS 名 globesite.dns.globeworld.org,然后传递给本地 DNS 名称解析器,如图 9.43 所示。(用符号#org 表示名为 org 的服务器地址。)假设处理的是一个有效的名称,最后,如果仍有部分名称无法解析,将联系能够处理此问题的 Globe 名称服务器。

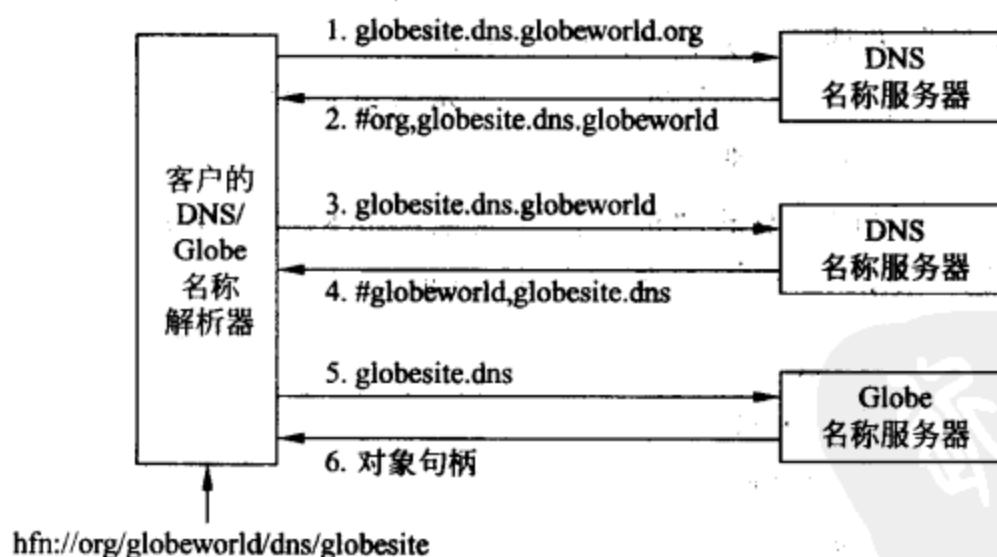


图 9.43 Globe 中基于 DNS 的迭代名称解析

上例中,DNS 名 globeworld.org 被解析到能够处理剩余部分 globesite.dns 的 Globe 名称服务器。Globe 名称服务器以常规 DNS 名称服务器的形式实现,除非 Globe 名称空间中的名称是与表示子目录或存储对象句柄的 TXT 资源记录相关的。

9.3.5 同步

Globe 不提供任何特定的同步机制。当需要同步时,通常是作为复制子对象的一部分实现的,如下文所述。在 Globe 中,对象间同步机制是不可用的。显然,实用应用程序开发需要这种机制。

9.3.6 复制

复制在 Globe 中起关键作用。Globe 中的每个分布式共享对象都通过其本地对象的复制子对象实现自己的复制策略。一个重要的设计问题是复制子对象的接口要标准化。这样,就有可能替换复制策略而不影响共同组成分布式共享对象的其他组件。

为了理解复制接口如何标准化,很重要的是要理解分布式共享对象中状态复制底层的模型。在分布式共享对象中,语义子对象实现状态以及对其的操作。状态复制通过在不同进程间传播语义子对象的拷贝而进行。

这里的一个关键问题是保证语义子对象的各种拷贝的一致性。需要何种一致性完全由分布式共享对象决定,由复制子对象集负责维护一致性。因此,复制子对象控制何时允许对语义子对象的本地方法调用。应区别两种方法:读方法(read method)和修改方法(rnmodify method),前者不修改语义子对象的状态而后者修改。

当绑定到分布式共享对象的进程调用方法时,控制子对象首先把调用通知给复制子对象。这一步通过调用复制子对象的接口的 start 方法进行,如图 9.44 所示。调用 start 方法时控制子对象提供的惟一信息是请求的对象是否修改语义子对象。基于此信息,复制对象将告诉控制对象下一步应做什么。

方法	描述
start	确定新方法调用已在本地请求
send	将编组后的调用请求传递给复制子对象
invoked	确定对语义对象的调用已完成

图 9.44 控制子对象可用的复制子对象接口

start 方法可能有两种结果,每一种都指明了控制子对象下一步的动作。第一种是控制子对象可能被指示把编组调用传送给复制子对象。换句话说,控制子对象下一步要调用 send 方法,之后等待,直到操作完成。

第二种可能的结果是控制子对象通过调用语义子对象实际执行请求。当调用完成时,控制子对象调用 invoked 方法,告诉复制子对象已经完成。它还是需要等待直到调用完成。

send 操作以编组调用为输入,通常将调用请求转发给其他副本。最后,可能来自多个副本的响应将返回给复制子对象。在任何情况下,如果一切顺利,复制子对象将给调用进程一个适当的响应(如果存在这样的响应),然后将此响应作为 send 的输出返回。

当对 send 的调用返回时,控制子对象或者将控制返回给调用进程,或者调用语义子对象。在第一种情况下,同样也被传递给控制子对象的响应被解编并传递给调用进程。

当要求控制子对象调用语义子对象时,在调用完成时必须调用 invoked。调用 invoked 之后,通常要求控制子对象把控制返回给调用进程。

对于复制子对象来说,希望控制子对象以有 4 个状态的有限状态机的方式行动,如图 9.45 所示。状态转换大部分由复制子对象控制,每一状态都有一个相关的行动。此行动既可通过调用复制子对象的特定方法实现,也可调用最初请求的语义子对象的方法。这样,复制子对象可准确地决定是否以及何时执行本地调用请求。下面研究两个复制策略,以说明这种方法。

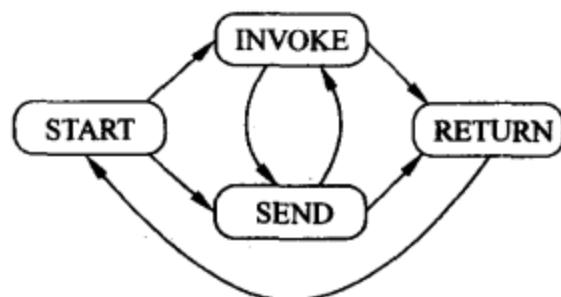


图 9.45 作为有限状态机的控制子对象的行为

Globe 中复制的实例

在主动复制的情况下,假设进程请求调用一个读方法。控制子对象首先调用 start,之后复制子对象立即指示控制子对象调用该方法,并把结果返回给调用进程。

处理修改方法需要遵从不同的步骤。假设复制子对象使用提供完全有序多播的通信子对象。在这种情况下,调用 start 时,复制子对象立即让控制子对象用 send 方法传送编组的调用。编组调用多播传送给所有的副本,包括发起多播传送的副本。当后者再次收到本身的调用请求时,它允许控制子对象继续调用该方法。调用结束时,结果传回调用进程。图 9.46 总结了这些步骤。

读方法			
状态	采取的动作	调用的方法	下一状态
START	无	start	INVOKE
INVOKE	调用本地方法	invoked	RETURN
RETURN	返回结果给调用者	无	START

修改方法			
状态	采取的动作	调用的方法	下一状态
START	无	start	SEND
SEND	传递编组的对象	send	INVOKE
INVOKE	调用本地方法	invoked	RETURN
RETURN	返回结果给调用者	无	START

图 9.46 主动复制的状态转换与动作

现在考虑一下主机一备份复制。在这种情况下,读方法仍可以本地调用,而结果可返回给调用进程。

当在备份副本调用上修改方法时,复制子对象将把请求转发给主机并等待操作的执行。它本身的状态也要在把控制传回到控制子对象之前更新。此时,要求控制子对象把控制返回给调用进程。

当在主机调用修改方法时,复制子对象让控制子对象在本地调用该方法,并通过对invoked的调用来报告该方法调用何时完成。此时,复制子对象把语义子对象的状态编组并发送到备份。所有备份一经更新,复制子对象就让控制子对象把控制返回给调用进程。图9.47总结了这一方法,并展示了刚才讨论的三种不同的可能性。

读方法			
状态	采取的动作	调用的方法	下一状态
START	无	start	INVOKE
INVOKE	调用本地方法	invoked	RETURN
RETURN	返回结果给调用者	无	START

备份副本的修改方法			
状态	采取的动作	调用的方法	下一状态
START	无	start	SEND
SEND	传递编组的对象	send	RETURN
RETURN	返回结果给调用者	无	START

主机的修改方法			
状态	采取的动作	调用的方法	下一状态
START	无	start	INVOKE
INVOKE	调用本地方法	invoked	RETURN
RETURN	返回结果给调用者	无	START

图9.47 主机一备份复制的状态转换与动作

9.3.7 容错性

目前Globe中没有广泛的容错性支持。支持主要以复制的形式存在。在许多情况下,能够在多台机器间复制分布式对象的状态就足以应付容错性要求了。然而,仍需要失败后恢复的机制。

为支持恢复,Globe对象服务器保证需要恢复的本地对象存储在磁盘上。换句话说,这些对象是持久的,如前所述。但是,还需要更多的机制保证恢复总是按需求工作。例如,需要注意对本地对象的更新应立即存储在磁盘中,或能够从副本恢复,以避免恢复陈旧的数据。Globe还没有提供这类工具。

9.3.8 安全性

Globe 中的安全性集中在保护一个分布式共享对象免受安全攻击上。实施这一保护的重要机制是作为每个本地对象一部分的附加安全子对象 (security subobject) 的设置。安全子对象的设置如图 9.48 所示。

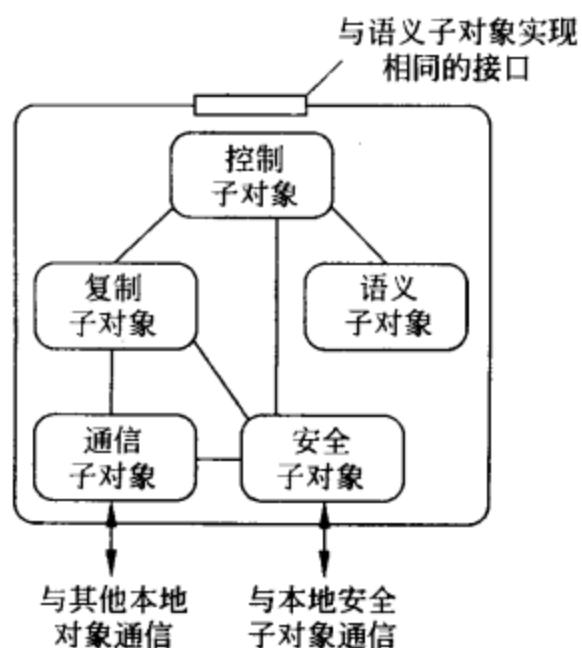


图 9.48 Globe 本地对象中安全子对象的位置

安全子对象在不同层次处理安全性问题。首先,有必要保证远程进程的调用请求是经过验证的。因此,通信子对象把每个输入的请求信息传递给安全子对象,让后者对请求者进行身份验证。

身份验证之后,复制子对象和安全子对象交换访问控制的信息。在 Globe 中,访问控制在本质上是建立在每个方法的基础上的。换句话说,对每个方法,安全子对象都可以决定是否允许来自先前标识的请求者的调用。

最后,安全子对象与控制子对象交换信息,从而检查调用请求的参数值是否在特定范围内。例如,对于用来处理提款的对象,安全子对象可以禁止超过事先定义限额的提款操作。

安全子对象和本地安全服务通信。例如,为了能够与其他本地对象建立通信,安全子对象必须获得会话密钥。在 Globe 中,与本地安全服务的通信是通过称为主体对象 (principal object) 的单独分布式共享对象进行的。这一对象在特定的安全域中代表分布式共享对象的使用者。主体对象负责与受信任的第三方通信,比如证书与身份验证服务。

作为例子,我们来考虑 Globe 分布式共享对象如何使用 Kerberos 安全系统在两个本地对象之间建立会话密钥的情况。本地对象 A 和 B 分别由进程 P_A 和 P_B 绑定到分布式共享对象上。图 9.49 表示了当 P_A 和 P_B 在不同的安全域中时,在 A 和 B 之间建立安全通信所采取的各种步骤。

首先,安全子对象请求主体对象 (P_A 代表) 以获得会话密钥,如第 1 步所示。然后,主体对象把请求发送给 Kerberos 验证与票据授予服务 (第 2 步),以获取会话密钥和本地对

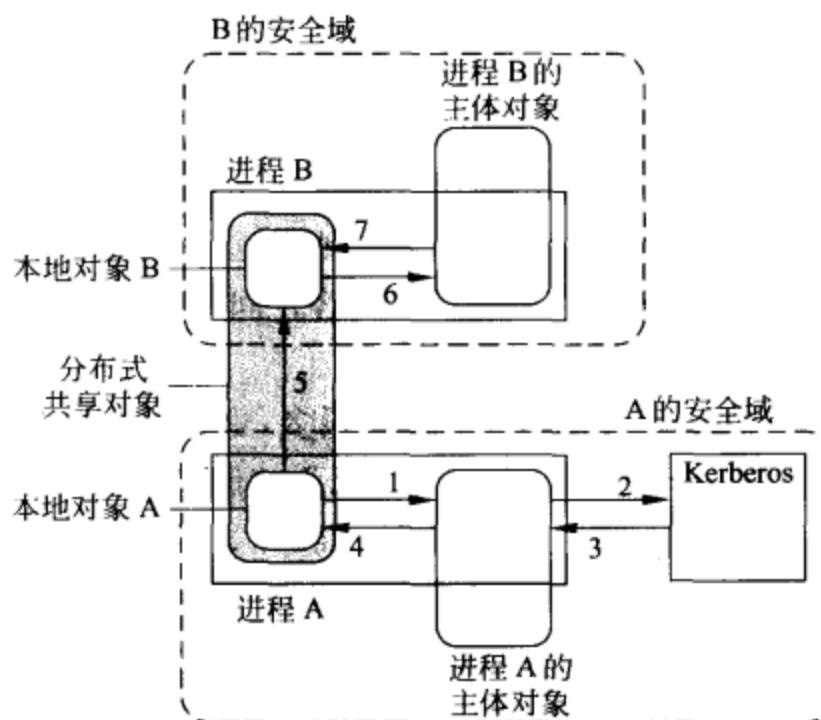


图 9.49 用 Kerberos 建立安全分布式共享对象

象 B 的票据(第 3 步),然后把它们传递给安全子对象(第 4 步)。

此时,安全子对象能够用对象的通信子对象与其他本地对象联系,并把票据传给它(第 5 步)。票据传递给代表进程 P_B的主体对象(第 6 步),它负责解释票据并把会话密钥返回给 B 的安全子对象(第 7 步)。

有关 Globe 安全性的更多信息参见文献(Homburg 2001, Leiwo 等 1999)。

9.4 CORBA、DCOM 和 Globe 的比较

CORBA、DCOM 和 Globe 是三种不同的基于对象的系统,每种都有其优点与不足。在详细讨论了各个系统的原理之后,现在比较一下三种系统的原理,首先从总体设计目标的比较开始。

9.4.1 基本原理

CORBA、DCOM 和 Globe 是为不同的目标开发的。CORBA 的努力目标主要是提供一个标准的中间件平台,使来自各个不同软件生产商的应用程序可以互操作。对各种 CORBA 组件的接口应该是什么样的问题,来自各个委员会的数以百计的参与者试图达成协议,目前有任何共识都会令人惊讶。

DCOM 的主要设计目标是改善功能性,同时与组成早期 Windows 系统的以前版本兼容。好消息是老的应用程序还能继续使用,坏消息是一旦做出错误决定,通常无法改正。考虑到它不是和 CORBA 一样的委员会工作的结果,DCOM 是非常复杂的。

Globe 是典型的研究成果。它由一个小组开发。这一小组强调简单性胜过功能性,而且能够从头开始设计系统。不必考虑其他人的利害关系,也不用考虑对以前版本的兼容性。和许多其他的研究项目一样,Globe 的设计在本质上是清晰而简单的。但是,

Globe 还不完整,其设计还需要在很多方面加以扩展。它的主要设计目标是提供可扩展性。

每个系统支持的对象模型有很大的区别。CORBA 和 DCOM 使用远程对象模型。CORBA 提供了通用而一致的对象模型。CORBA 方法的长处在于它为访问和定位对象提供了高度的透明性。对客户来说,很难看到本地和远程对象的不同。对象有状态,能够被全局识别,其引用能够很容易地在不同机器的客户间传递。此外,同时支持暂时对象和持久对象。

DCOM 的模型比 CORBA 使用的模型简单很多,也因过于简单而受到批评。DCOM 对象是暂时的,没有全局标识符,甚至在一些情况下被假设为没有状态。这一模型违反了分布式对象技术的很多基本原理。

Globe 也提供真正的对象。它区别于其他对象模型的特点是 Globe 对象可物理复制并分布于多台机器。而且,对象决定如何进行自身状态的分布与复制。换句话说,对象封装了本身的分布策略与实现。此外,它还可以封装安全、失败处理等策略与实现。

在比较它们的服务时,也可以发现很多不同。CORBA 提供了完备的服务,如此丰富以至有些相互交叠。另一方面,DCOM 提供它自己的混合服务,但也依赖于由环境提供的服务,比如命名和目录服务。Globe 仅提供了最小的服务集,一个简单的命名服务和一个先进的定位服务。这符合 Globe 的设计原则——简单化,但是如果真的把 Globe 用作通用的分布式系统,显然还需要更多的服务。

三个系统间另一个重要区别是对接口的看法。CORBA 遵从基于语言的接口方法,提供标准的 IDL 把规范映射为各种语言。由于可移植性和互操作性的原因,这些语言映射在本质上是标准化的。此方法的主要优点是 CORBA 独立于编译器所生成的代码。在本质上,CORBA 在编程语言层定义了互操作性。

相比之下,DCOM 和 Globe 都支持二进制接口方法。在这种方法中,对象接口的定义独立于编程语言。DCOM 中,二进制接口的优点已经被用诸如 C、Java 和 Visual Basic 等混合语言书写的众多应用程序所证明。Globe 采用相同的方法。

9.4.2 通信

现在考虑一下系统如何处理通信。CORBA 通信最初只有简单的同步方法调用。这一最初模型也以单向和延迟同步请求的形式提供了异步通信。目前,CORBA 通信模型一点都不简单,但提供了丰富的可能性。有以事件和通知服务形式提供的异步通信,能够产生、使用和过滤事件。此外,CORBA 提供消息服务,能支持回叫和消息轮询的结合。

在这方面,DCOM 类似于 CORBA。它提供同步远程方法调用和不同形式(不兼容)的回叫机制。像 CORBA 一样,它也支持消息发送。

Globe 只有同步方法调用。它没有事件、回叫和消息等。因为 Globe 对象能被复制,所以也不允许对象调用另一个对象。为了解决复制调用带来的问题,需要采取如(Mazouni 1995)中讨论过的特殊方法。到目前为止,对调用链的支持还未出过差错。注意,这一问题不仅针对 Globe,CORBA 和 DCOM 一旦复制对象,也将面临这一问题。

9.4.3 进程

每个系统都提供了自己的对象服务器。对象服务器相互之间有很大不同。CORBA 对象服务器在很多方面都是最通用和灵活的。它用方法明确区分了对象应实现什么,用 CORBA 对象适配器区别对象和方法如何管理。这种明确的区分为开发分布式对象以及在分布式系统中使用它们提供了很大帮助。

DCOM 对象服务器是 Microsoft 对帮助分布式对象开发做出的回答。它不像使用 POA 的 CORBA 对象服务器那样灵活,而将主要的对象管理和方法调用硬编码进运行时系统。此外,对象开发人员在处理线程支持时,在很大程度上要独立开发。DCOM 对象服务器提供的 JIT 激活可提高性能,但只在对象没有状态时可用。这种方法是否符合对象技术也存在很大的疑问。

Globe 对象服务器仍然相当粗糙,但是提供了与 CORBA POA 相似的功能。对象可以有自己的控制子对象和复制子对象的特定实现,上述功能主要来自于这一事实。Globe 对象服务器只做它支持的对象要求它做的事情。不必像 CORBA POA 那样配置服务器。但是,Globe 对象主要能够决定自身如何分布、复制和安全,仍有其他类型的策略需要对象服务器支持。例如,目前只有有限的几种办法支持持久性。

9.4.4 命名

当比较 CORBA、DCOM 和 Globe 对命名的支持时,会发现存在很大不同。

CORBA 为对象命名和引用提供了很多工具。特别是,有独立的命名服务、属性服务和交易服务。本质上,这些服务都返回一个或多个对象引用。CORBA 对象引用直接指向支持此引用的对象服务器。或者,它指向保存对象当前服务器路径的实现仓库,甚至在必要时激活对象。

换句话说,CORBA 中的对象引用是与地址有关的。结果,不论何时移动对象,其引用都会变为无效的。这一方法通常适用于相对容易实现有效定位服务的局域分布式系统。但是,作为通用解决方案,它无法扩展成大规模广域系统。

DCOM 本身没有命名服务,而是依赖于其环境提供的已有服务,如活动目录。如果此类服务不可用,按名称查找对象就不可能。

DCOM 中对象引用的形式之一是接口指针。一般说来,对这样的指针进行编组要求把客户代理编组并传送给目标。使用标准编组可以稍微简化,但整体上讲,把接口指针作为对象引用传递给另一个客户是难以处理的事情,尤其在有特殊要求的时候。

持久对象引用以标记(moniker)的形式出现,它被 Orfali 等(1996)称为“拼装品”。必须承认,标记不容易理解。它们确实解决了和对象引用相关的许多问题,但问题是它们是不是最好的解决方案。

Globe 采用独立的对象命名和编址,这一点与 CORBA 和 DCOM 是根本不同的。它使用了“长寿”而独立于地址的对象标识符,允许对象更改名称而不影响名称到地址的映射,这一映射通常存储在传统的命名服务中。同样,当地址更改时,也不影响名称到地址的映射。这一方法的优点已经在第 4 章中详细讨论过了。它用于 Globe 中的主要缺点是

需要一个独立的、世界范围的定位服务。开发这样的服务几乎是不可能的。

Globe 对象的字符串命名相对简单。所缺乏的是允许客户按所要求属性查找对象的目录服务。对许多分布式系统来说,这是基本服务。

9.4.5 同步

同步机制以传统的锁定和事务服务的形式提供,但只在 CORBA 和 DCOM 中实现。Globe 没有明确的同步机制,除了以复制子对象方法实现的对象内部同步。

9.4.6 缓存与复制

当考虑缓存与复制时,三个系统间也有重要的不同。

详细说明 CORBA 中的复制已经成为被搁置了很久的努力,主要是把它与远程对象概念的集成在一起的困难造成的。目前采用的方法是由单独的复制服务器创建一组对象,并使其看上去是一个远程对象。换句话说,对于客户来说,对象是否被复制没有本质区别。

复制服务器决定复制何时、何地、如何进行。目前,有几种复制策略可用于一组对象。所有的策略都实现连续一致性。典型的实现基于全序的多播机制。

没有特别为缓存提供支持,但是缓存可以很容易地添加到使用截取程序的已有实现中。缓存子系统的实现完全留给了应用程序开发人员。这一方法的问题是可能需要付出相当可观的努力。

DCOM 没有提供任何对缓存和复制的支持。这些方面完全留给应用程序处理。

Globe 采用与 CORBA 不同的方法。它把复制子对象并入分布式共享对象的所有本地对象。因为复制子对象的接口是标准化的,各种复制策略的不同实现能够很容易地关联到一个对象。结果是高度的灵活性,比 CORBA 支持的固定策略列表高得多。可使用相同的机制创建缓存工具。

虽然 Globe 为把复制策略关联到对象提供了通用的机制,但目前还没有一个较大的复制子对象实现集合。在这种意义上,Globe 的复制子对象可与 CORBA 的截取程序相比。它们都仅仅提供了一种机制,但还没有为分布式对象开发提供可用的子系统。

9.4.7 容错性

CORBA 中的容错性主要用上面提到的复制的办法来支持。此外,CORBA 的事务和并行控制服务对开发容错服务也有帮助。

DCOM 通过自动事务处理提供相似的支持,即由单独的事务协调进行备份。

Globe 不支持事务,其容错性支持主要用复制的办法。对于崩溃的恢复,除了能够向本地持久对象写入状态,几乎没有更多的工具。

9.4.8 安全性

最后比较 CORBA、DCOM 和 Globe 的安全性,它们之间仍然存在很多不同。

CORBA 为远程对象定义了完整的安全性体系结构。通过提供工具决定何时何地应

用安全功能,机制与策略得到了清晰的区分。安全性本身以依赖于应用程序的策略对象的形式出现,当 ORB 处理请求或回复时要调用策略对象。策略对象可由应用程序配置,通常使用本地可用的安全服务。例如,身份验证的策略对象可能是基于 Kerberos 的。

调用策略对象的机制又是截取调用。为了以安全的方式进行,CORBA 为调用策略对象和检查访问控制分别提供了安全截取。

DCOM 遵循不同的方法。安全性可由声明方式支持,即在注册表中确定对象激活、访问控制和身份验证的要求。与 DCOM 结合的操作系统将负责实现这些要求。另一种方法是通过在对象初始化时调用特殊函数来让应用程序决定需要什么安全服务。

Globe 的安全性部分由单独的安全子对象处理。这一子对象与本地安全服务(如 Kerberos)通信。此外,它还处理通信、复制和调用语义子对象方面的安全问题。但是,Globe 的安全性体系结构还不完整。例如,本地对象如何真正得到想要的安全子对象,而不是恶意的子对象,还没有规定。

图 9.50 总结了本节中比较 CORBA、DCOM 和 Globe 时所讨论的各种问题。

项目	CORBA	DCOM	Globe
设计目标	互操作性	功能性	可扩展性
对象模型	远程对象	远程对象	分布式对象
服务	本身有很多	从环境获得	很少
接口	基于 IDL	二进制	二进制
同步通信	是	是	是
异步通信	是	是	否
回叫	是	是	否
事件	是	是	否
消息	是	是	否
对象服务器	灵活的(POA)	硬编码的	依赖于对象的
目录服务	是	是	否
交易服务	是	否	否
命名服务	是	是	是
定位服务	否	否	是
对象索引	对象的位置	接口指针	真正的标识符
同步	事务	事务	仅限于对象内部
复制支持	独立的服务器	无	独立的子对象
事务	是	是	否
容错性	通过复制实现	通过事务实现	通过复制实现
恢复支持	是	通过事务实现	否
安全性	多种机制	多种机制	还需更多的工作

图 9.50 CORBA、DCOM 和 Globe 的比较

9.5 小结

基于对象的分布式系统几乎都建立在远程对象模型的基础上。这一方法有时要求在需要定制时采用特殊的措施,例如对缓存和复制的支持。在 CORBA 中,截取程序用于修改出入的调用请求。DCOM 中,定制编组允许必要时调整客户端代理。Globe 通过允许位于不同机器上的不同本地对象共存于同一个分布式对象中来处理定制。

除了同步方法调用,基于对象的分布式系统(比如 CORBA 和 DCOM)提供了替代形式的调用,包括事件和异步方法调用。Globe 不支持这些替代形式,本质上只提供同步调用。

不同系统的对象服务器结构在某种程度上很相似。在所有情况下,服务器都能支持多于一个的对象。差异存在于调整服务器的灵活性方面。CORBA 中,灵活性通过对象适配器获得。DCOM 提供标准对象服务器,可根据特定的应用程序调节。Globe 中,对象服务器相对简单,因为任何特殊之处都作为分布式对象的一部分而加以实现。

基于对象的分布式系统中命名的差异出现在对象引用的层次,而对易于理解的名称的支持几乎是一样。在 CORBA 和 DCOM 中,系统范围的对象引用是依赖于地址的。这些引用含有对象所在服务器的地址信息。相反,Globe 使用独立于地址的引用,但依赖于世界范围的定位服务把引用解析为联系地址。联系地址给出对象在何处,如何到达等信息。

CORBA 的复制只在容错性方面得到支持。DCOM 根本没有提供复制。应用程序开发人员不得不使用特定的复制服务器,或者显式地为复制编程。Globe 中,支持基于每个对象的复制,办法是作为对象一部分的复制子对象为其实现特定的协议。

与复制相关的是容错性支持。CORBA 通过基于底层可靠多播服务的复制服务提供容错支持。DCOM 中,容错性用(自动)事务处理支持。Globe 仅通过复制支持容错性,但不提供恢复工具。

最后,本章中讨论的分布式系统都解决了安全性问题。CORBA 提供了完整的安全性体系结构,允许按对象设置安全性。DCOM 中,安全性与已证明的现有安全服务,如 Kerberos 紧密结合。Globe 中,安全性也是基于每个对象定义的,同时还提供与已有安全服务连接的挂钩(hook)。Globe 的安全性仍是值得深入研究的题目。

习题

1. 为什么用接口定义语言定义对象的接口是很有用的?
 2. 给出 CORBA 中动态调用机制起作用的例子。
 3. 第 2.4 节讨论的 6 种通信形式中,哪些得到 CORBA 的调用模型支持?
 4. 给出一个简单的协议概要,实现对象调用的至多一次(at-most-once)语义。
 5. 异步方法调用中,客户端和服务器端的 CORBA 对象应该是持久性对象吗?
 6. 在 GIOP 请求(request)消息中,对象引用和方法名都是消息报头的一部分。它们
- 444 •

为什么不能仅包含在消息体内？

7. CORBA 支持第 3 章中解释的每对象一个线程的调用策略吗？
8. 假设两个 CORBA 系统都有自己的命名服务。概述如何将两个命名服务集成成为一个联合的命名服务。
9. 本章中介绍当绑定到 CORBA 对象时，附加的安全服务将由客户 ORB 基于对象引用来选择。客户 ORB 如何得知这些服务？
10. 如果 CORBA ORB 使用了一些与安全性无关的截取程序，这些截取程序的重要程度如何？
11. 说明事务队列如何成为分布式事务的一部分，就像文中提到的那样。
12. 比较 DCOM 与 CORBA，CORBA 提供标准编组还是定制编组？请讨论。
13. 在 DCOM 中，假定被包含在接口 X 的方法 m 接受到接口 Y 的指针为输入参数。客户持有指向 X 的代理实现的接口指针，解释当客户调用 m 时，编组如何进行。
14. DCOM 的定制编组是否要求接收编组代理的进程与发送进程运行于相同类型的机器上？
15. 概述把 DCOM 对象移动到另一服务器上的算法。
16. JIT 激活在多大程度上违反或遵从对象的概念？
17. 解释当不同机器上的两个客户使用同一文件标记（moniker）绑定到一个对象时，会发生什么。此对象会实例化一次还是两次？
18. 不违反对象是被动的原则，在 Globe 中实现事件的明显办法是什么？
19. 有时无意中使用回叫机制可能很容易地导致非需要的情况出现，给出例子。
20. 在 CORBA 中，命名图中的节点，如目录也被认为是对象。在 Globe 中把目录也模型化为分布式共享对象有意义吗？
21. 假设 Globe 对象服务器刚安装了一个持久性本地对象。又假设此对象提供了联系地址。当服务器关闭时，此地址也要保存吗？
22. 在 Globe 安全中使用 Kerberos 的例子里，如果用在对象和票据授予服务之间共享的密钥而不是密钥 $K_{B,TGT}$ 来加密票据，这样有用吗？

第 10 章 分布式文件系统

鉴于共享数据是分布式系统的基础,分布式文件系统是构成许多分布式应用程序的基础就不足为奇了。分布式文件系统允许多个进程在长时期内以一种安全、可靠的方式共享数据。所以,它们已被用作分布式系统和分布式应用程序的基础层。在本章中,我们把分布式文件系统看作通用分布式系统的模型。

我们将详细讨论两个不同的分布式文件系统。第一个实例是 SUN NFS(network file system,网络文件系统),它已经得到了广泛应用,现在正逐渐向基于 Internet 的大规模文件系统的方向扩展。NFS 的大多数实现基于其版本 3 规范。最近,其版本 4 规范已经制订完成,一个实验性的实现支持这一版本的规范。

另一个完全不同的分布式文件系统是 Coda。Coda 源于 AFS(Andrew 文件系统),AFS 是一个在设计时就考虑可扩展性的大规模系统。Coda 与许多其他文件系统的区别在于它在面对网络分区时支持连续操作。特别是那些故意时常断开网络连接的移动用户(比如,使用膝上型计算机的人们)会从其中获益。

我们也将简要讲述其他三个系统。Plan 9 是一个将所有资源都视为文件的分布式系统。从这种意义上来说,它可被视为一个基于文件的分布式系统。我们将讲述的另一个系统是 xFS,其与众不同之处在于它没有服务器,而是让客户实现文件系统。最后,我们会介绍 SFS,该系统强调可扩展的安全性。

本书第一部分所讨论的所有 7 条原则都适用于分布式文件系统,我们将针对本章的实例系统讨论这 7 条原则。最后,我们以各种文件系统的比较结束本章内容。

10.1 SUN 网络文件系统

我们以 SUN 微系统的网络文件系统(network file system),即通常所说的 NFS,开始讨论分布式文件系统。NFS 最初是 Sun 为在它的基于 UNIX 的工作站上的使用而开发的,但是 NFS 已在许多其他系统中实现。NFS 的基本思想是每台文件服务器提供它的本地文件系统的标准化视图。也就是说,它不关心如何实现本地文件系统,每台 NFS 服务器支持相同的模型。这个模型带有一个通信协议,该协议允许客户访问存储在一台服务器上的文件。这种方法允许大量异构进程共享一个公用的文件系统,其中的进程可能运行于不同的操作系统和机器上。

NFS 具有相对较长的历史。NFS 版本 1 从未发布,而被保留在 Sun 公司内部。NFS 版本 2 并入 Sun 的操作系统 SunOS 2.0 之中,该版本在(Sandberg 等 1985)中描述。几年后,发布了版本 3(Pawloski 等 1994)。Callaghan(2000)详细地描述了 NFS 的这两个

版本。

目前,NFS 版本 3 正在进行较大的修订,这将导致下一版 NFS 的产生(请参阅 Shepler 等 2000)。这些修订的主要目的是为将其应用于 Internet 时能提供更好的性能,将 NFS 转变为一个真正的广域文件系统。其他改进是关于安全性和互操作性的。下面,我们详细研究 NFS,主要是 NFS 版本 3 和版本 4 的各个方面。

10.1.1 NFS 概述

如前所述,与其说 NFS 是一个真正的文件系统,不如说它是共同为客户提供分布式文件系统模型的协议的集合。从这个角度来看,NFS 类似于 CORBA,CORBA 本质上也以规范的形式存在。与 CORBA 相似,NFS 有多种运行于不同操作系统和机器的实现。NFS 协议是以不同的实现之间应该易于进行互操作为目的而设计的。因此,NFS 可以运行于大量异构计算机之上。

1. NFS 体系结构

NFS 底层的模型是远程文件服务的模型。这个模型为客户提供对远程服务器所管理的文件系统的透明访问。但是,客户通常不知道文件的实际位置,相反,NFS 为客户提供访问此文件系统的接口,此接口类似于传统本地文件系统所提供的接口。也就是说,客户仅被提供包含多种文件操作的接口,而服务器负责实现这些操作。因此,这一模型也被称为远程访问模型(remote access model),如图 10.1(a)所示。

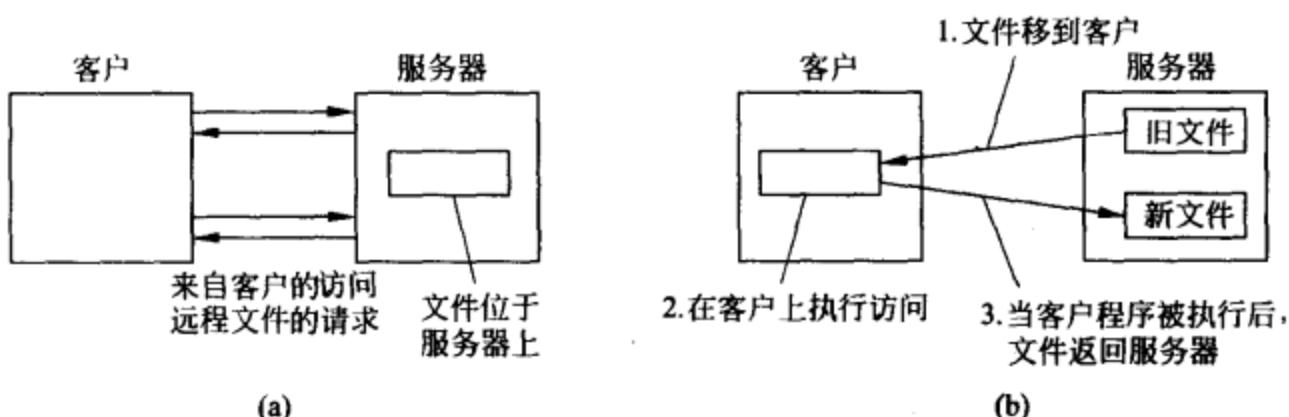


图 10.1 远程访问模型和上载/下载模型

(a) 远程访问模型; (b) 上载/下载模型

与之相对照,在上载/下载模型(upload/download model)中,客户从服务器下载文件后,在本地访问该文件,如图 10.1(b)所示。客户完成对该文件的访问后,再将该文件上载回服务器,以便其他客户使用该文件。当客户下载一个完整的文件,修改该文件,然后将其放回服务器时,可以使用 Internet 的 FTP 服务。

尽管 NFS 基于 UNIX 的版本占主流地位,但是 NFS 已开发了多种版本以用于许多不同操作系统。实际上,对所有现代 UNIX 系统而言,NFS 通常按照如图 10.2 所示的层次式体系结构实现。

客户使用其本地操作系统提供的系统调用访问文件系统。但是,本地 UNIX 文件系统接口已被 VFS(virtual file system,虚拟文件系统)的接口代替。目前,VFS 已经成为

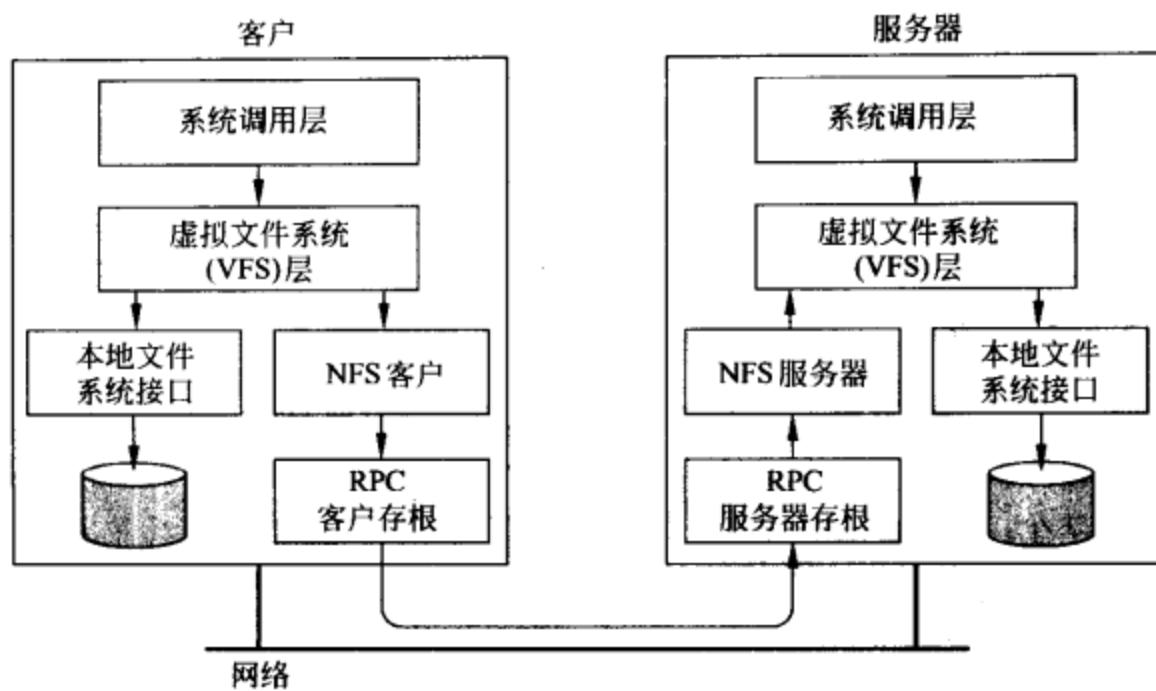


图 10.2 UNIX 系统的基本 NFS 体系结构

不同(分布式)文件系统接口的事实工业标准(Kleiman 1986)。VFS 接口上的操作或者被传送到本地文件系统,或者被传送到一个称为 NFS 客户的单独组件,该组件负责处理对存储在远程服务器上的文件的访问。在 NFS 中,所有客户-服务器通信都是通过 RPC 完成的。NFS 客户将 NFS 文件系统的操作实现为服务器的 RPC。注意,VFS 接口所提供的操作可以不同于 NFS 客户所提供的那些操作。VFS 的整体思路是隐藏不同文件系统之间的差异。

在服务器端,我们可以看到相似的组织方式。NFS 服务器负责处理输入的客户请求。RPC 存根对请求进行解编,NFS 服务器将它们转换成常规的 VFS 文件操作,随后这些操作被传送到 VFS 层。总之,VFS 负责实现真实文件所在的本地文件系统。

这种方法的一个重要优点是 NFS 在很大程度上独立于本地文件系统。本质上,它确实不关心客户端或服务器端的操作系统实现的是 UNIX 文件系统,还是 Windows 2000 文件系统,或是更老的 MS-DOS 文件系统。所关心的惟一重要问题是这些文件系统是否与 NFS 所提供的文件系统模型兼容。比如,因为 MS-DOS 使用短文件名,所以它不能被用于以完全透明的方式实现 NFS 服务器。

2. 文件系统模型

NFS 提供的文件系统模型几乎与基于 UNIX 的系统所提供的文件系统模型完全一样。文件被视为未解释的字节序列。它们被层次地组织到命名图中,图中的节点表示目录和文件。NFS 也像任何 UNIX 文件系统一样,支持硬链接和符号链接。文件具有文件名,但是访问它们是通过一种类似 UNIX 的文件句柄(file handle)实现的,我们将在后面详细讨论文件句柄。换句话说,为访问一个文件,客户必须先在命名服务中搜索该文件的文件名,然后获得该文件关联的文件句柄。另外,每个文件有许多属性,这些属性的值是可以查询和修改的。我们将在后面详细讨论命名。

图 10.3 分别列出了 NFS 版本 3 和版本 4 支持的通用文件操作。create 操作用于创建一个文件,但是它在版本 3 和版本 4 中具有不同的意义。在版本 3 中,该操作用于创建常规文件。非常规文件使用另一种单独的操作创建。link 操作用于创建硬链接,symlink 用于创建符号链接。mkdir 用于创建子目录。诸如设备文件、套接字和命名管道等特殊文件是通过 mknod 操作创建的。

操作	版本 3	版本 4	描述
create	有	无	创建一个常规文件
create	无	有	创建一个非常规文件
link	有	有	创建一个文件的硬链接
symlink	有	无	创建一个文件的符号链接
mkdir	有	无	在给定目录下创建一个子目录
mknod	有	无	创建一个特殊文件
rename	有	有	更改文件名
remove	有	有	从文件系统删除一个文件
rmdir	有	无	从一个目录删除一个空的子目录
open	无	有	打开一个文件
close	无	有	关闭一个文件
lookup	有	有	根据文件名搜索文件
readdir	有	有	读取一个目录下的项目
readlink	有	有	读取符号链接所存储的路径名
getattr	有	有	获得文件的属性值
setattr	有	有	设置文件的一个或多个属性值
read	有	有	读取文件中的数据
write	有	有	向文件写入数据

图 10.3 NFS 所支持的文件系统操作的不完全列表

版本 4 完全改变了这种情况。在版本 4 中,create 用于创建非常规文件,这些非常规文件包括符号链接、目录和特殊文件。硬链接仍使用一个单独的 link 操作创建,但常规文件是通过 open 操作创建的,该操作是 NFS 中的新操作,它是 NFS 版本 4 与老版本在文件处理方法上的主要不同。在版本 4 之前,NFS 被设计为允许它的文件服务器是无状态的。出于某些原因,版本 4 舍弃了这一设计标准。我们将在本章后面讨论这些原因。版本 4 假设服务器通常会维护同一文件上多个操作间的状态。

操作 rename 用于更改已存在文件的文件名。

删除文件是通过 remove 操作实现的。在版本 4 中,这一操作用于删除任何类型的文件。而以前的版本需要使用一个单独的 rmdir 操作删除子目录。删除文件是通过文件名进行的,其效果是该文件的硬链接个数减 1。如果其链接个数降为 0,那么该文件可能被破坏。

版本 4 允许客户打开和关闭(常规)文件。打开一个不存在的文件将导致创建一个新文件。要打开一个文件,客户需要提供文件名以及各种属性值。比如,客户可能指定以写访问方式打开文件。一个文件被成功打开后,客户可以通过该文件的文件句柄访问该文件。这个文件句柄也用于关闭该文件,客户通过这个文件句柄通知服务器,它不再需要访问该文件。然后,服务器可以释放它为客户访问该文件所保留的状态。

lookup 操作用于搜索给定路径名的文件句柄。在 NFS 版本 3 中,lookup 操作不会越过挂接点(mount point)解析名称。(回忆一下第 4 章内容,挂接点是一个目录,该目录本质上代表指向外部名称空间(foreign name space 中的一个子目录的链接))。比如,假设名称/remote/vu 代表命名图中的一个挂接点。解析名称/remote/vu/mbox 时,NFS 版本 3 中的 lookup 操作将返回挂接点/remote/vu 的文件句柄,以及路径名中的剩余部分(即 mbox)。然后,客户必须显式地装入完成名称查询所需的文件系统。这里所说的文件系统是一些文件、属性、目录和数据块的集合,它们共同地被实现为一个逻辑块设备(Crowley 1997, Tanenbaum 和 Woodhull 1997)。

版本 4 简化了这一操作。在版本 4 中,lookup 将试图解析整个名称,即使此名称越过挂接点。注意,这一方法只有在文件系统已经被装入到挂接点时才是可行的。客户通过检查 lookup 完成时所返回的文件系统标识符,能够检测到挂接点已被越过。

读取目录中的项目有一个单独的操作 readdir。该操作返回一个(文件名,文件句柄)对的列表,以及客户请求的属性值。客户也可以指定返回的项目的个数。该操作返回一个偏移量,随后的 readdir 调用使用这一偏移量读取后面的项目。

操作 readlink 用于读取符号链接所关联的数据。通常,这一数据相当于随后查询文件的路径名。注意,lookup 操作不能处理符号链接。因而,遇到符号链接时,名称解析将停止,客户必须先调用 readlink,确定继续进行名称解析的位置。

文件具有多种关联的属性。同样,NFS 版本 3 和版本 4 在这一方面也有很大的不同,我们将在后面讨论它们的区别。典型的属性包括文件的类型(表明我们所处理的是目录、符号链接、特殊文件等)、文件长度、包含文件的文件系统的标识符,以及文件最后一次修改的时间。操作 getattr 和 setattr 分别用于读取和设置文件属性。

最后,NFS 还有从文件读取数据和向文件写入数据的操作。通过 read 操作读取数据是简单而又直接的。客户可以指定将要读取的偏移量和字节数。该操作向客户返回已被读取的实际字节数,以及附加的状态信息(比如,是否到达文件结束符)。

向文件写入数据是通过操作 write 完成的。同样,客户指定在文件中开始写入的位置、将要写入的字节数以及数据。另外,它可以命令服务器确保所有数据将被写入稳定存储器(我们在第 7 章曾讨论过稳定存储器)。NFS 服务器必须支持可以抵御电力供应故障、操作系统故障和硬件故障的存储设备。

10.1.2 通信

NFS 设计中的一个重要问题是各种操作系统、网络体系结构和传输协议的独立性。比如,这一独立性确保运行在 Windows 系统上的客户可以与 UNIX 文件服务器通信。这一独立性可以在很大程度上归因于 NFS 协议本身被置于 RPC 层之上这样的事实,因为

RPC 层隐藏了各种操作系统和网络之间的差异。

在 NFS 中,客户和服务器之间的所有通信都遵循 ONC RPC (open network computing RPC,开放式网络计算 RPC) 协议以及表示编组数据的标准 (Srinivasan 1995b)。(Srinivasan 1995a) 中正式定义了开放式网络计算 RPC 协议。ONC RPC 类似于第 2 章所讨论的其他 RPC 系统。它的编程接口和实际应用在(Bloomer 1992) 中有所描述。

每个 NFS 操作都可以被实现为文件服务器的单一的远程过程调用。实际上,在 NFS 版本 4 之前,客户通过保持请求相对简单来尽可能地简化服务器的工作。比如,客户为了第一次从一个文件读取数据,通常必须先使用 lookup 操作搜索文件句柄,然后它才能提交请求,如图 10.4(a) 所示。

这种方法必然需要两个连续的 RPC。假如在广域系统中使用 NFS,这种方法显然存在缺点。此时,第二个 RPC 的额外延迟可能导致性能下降。为了解决这些问题,NFS 版本 4 支持复合过程(compound procedure),通过它,几个 RPC 可以被组合成一个单一的请求,如图 10.4(b) 所示。

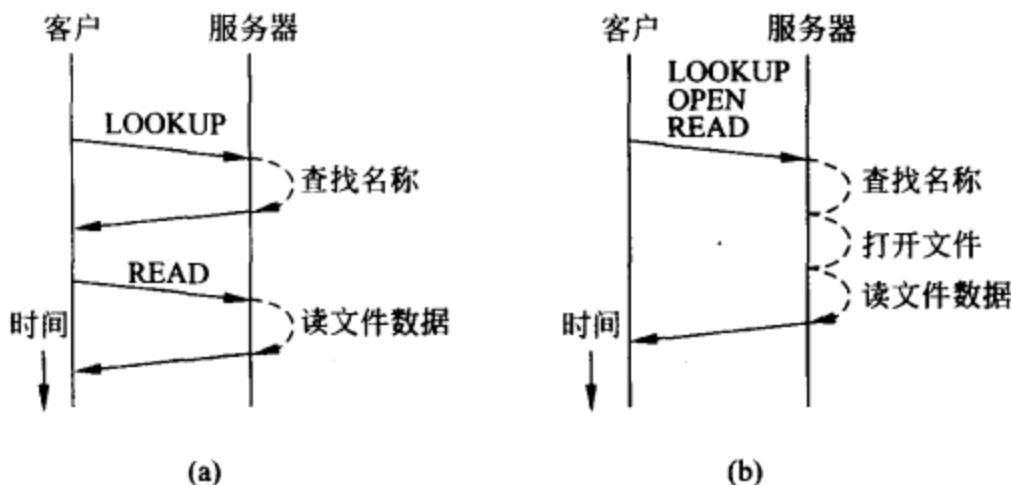


图 10.4 NFS 读取数据

(a) 在 NFS 版本 3 中,从文件读取数据; (b) 在 NFS 版本 4 中,使用复合过程读取数据

在本实例中,客户将 lookup 和 read 请求组合成一个单一的 RPC。在版本 4 中,在读文件之前打开文件也是必须的。文件句柄被找到后,它被传送到 open 操作,此后,服务器继续执行 read 操作。本实例的整体效果是,客户和服务器之间只需交换两个消息。

复合过程并不包含事务处理语义。组合在一个复合过程中的多个操作只是按照请求这些操作的顺序进行处理。如果存在来自其他客户的并发操作,那么它不采取任何措施避免冲突。无论什么原因导致一个操作失败,复合过程中的后续操作都将不能执行,目前所得的结果将被返回客户。比如,如果 lookup 失败,那么根本不会尝试执行后续的 open 操作。

10.1.3 进程

NFS 是一个传统的客户-服务器系统,在此系统中,客户请求文件服务器在文件上执行操作。与其他分布式文件系统相比,它长期独有的一一个特性是服务器可以是无状态的。换句话说,NFS 协议不要求服务器保留任何客户的状态。版本 2 和版本 3 仍沿用这种方

法,但是版本 4 已放弃了这种方法。

无状态方法的主要优点是简单性。比如,无状态服务器崩溃时,本质上不需要进入使其回到原先状态的恢复阶段。但是,如 7.3 节所述,我们仍需要考虑客户端无法得到任何关于请求是否真正得以执行的保证的问题。我们将在后面讨论 NFS 容错。

实际的实现不可能完全遵循 NFS 协议的无状态方法。比如,无状态服务器难以实现文件的锁定。NFS 使用一个单独的锁管理器处理这种情况。同样,某些身份验证协议要求服务器保留其客户的状态。然而,NFS 服务器通常设计为只需保留很少的客户信息。对于大多数情况,这种方案已经足够了。

从版本 4 开始,NFS 放弃了无状态方法,尽管其协议仍设计为服务器不需要保留很多客户的信息。除了上面提到的那些原因外,选择有状态的方法还有一些其他原因。一个重要原因是期望 NFS 版本 4 能够应用于广域网。这就需要客户可以有效地利用高速缓存,这样就需要有效的高速缓存一致性协议。这些协议通常在与能够保留客户所用文件的某些信息的服务器合作时工作得最好。比如,一台服务器可以将它发放给客户的每个文件与一个租用关联起来,以保证在租用到期或被刷新之前给予该客户独占读和写的权限。我们将在本章后面继续讨论这些问题。

以前各版本 NFS 之间最明显的区别在于对 open 操作的支持,该操作是天生有状态的。另外,NFS 支持回叫过程,服务器可以通过它调用客户的 RPC。显然,回叫也要求服务器记录它的客户。

10.1.4 命名

与任何其他分布式文件系统一样,命名在 NFS 中也起到了极其重要的作用。NFS 命名模型的基本思想是为客户提供完全透明地访问由服务器保存的远程文件系统的机制。这种透明性是通过让客户能够在它自己的本地文件系统装入一个远程文件系统实现的,如图 10.5 所示。

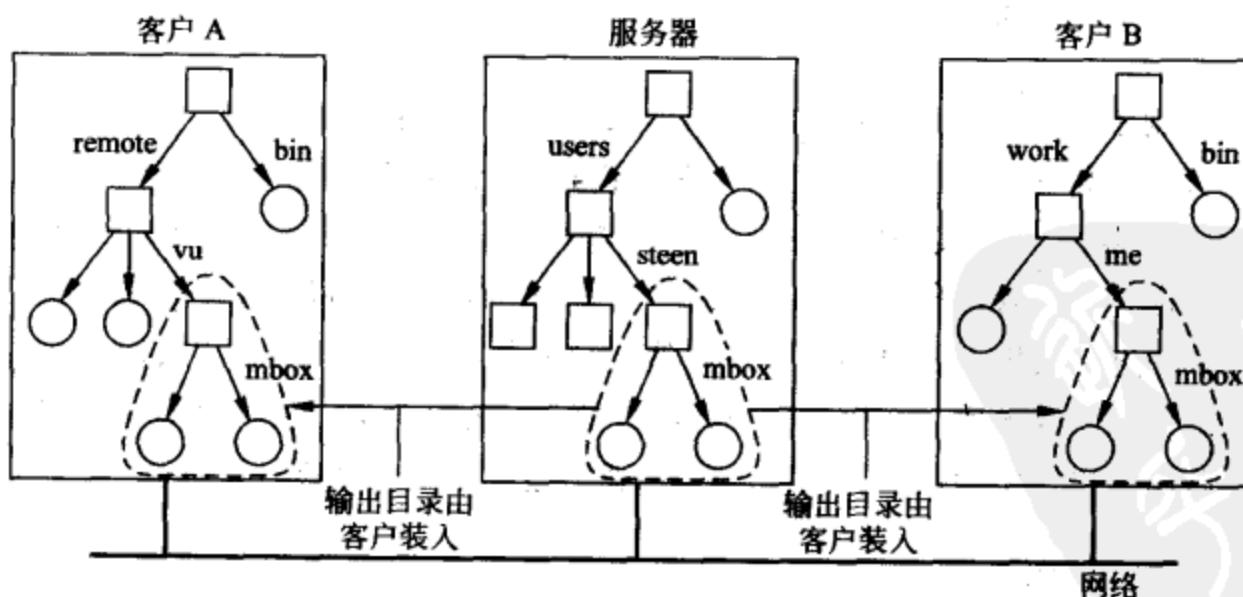


图 10.5 在 NFS 中装入(部分)远程文件系统

NFS 允许客户不装入整个文件系统,而只装入文件系统的一部分,如图 10.5 所示。

当服务器允许其他客户使用一个目录及该目录的项目时,称该服务器输出(export)那个目录。一个输出目录可以由客户装入其本地名称空间。

这种设计方法有一个重要的含义:用户基本上不共享名称空间。如图 10.5 所示,客户 A 处名称为 /remote/vu/mbox 的文件在客户 B 处被命名为 /work/me/mbox。因此,一个文件的名称依赖于客户组织他们自己的本地命名空间的方式,以及装入输出目录的位置。分布式文件系统中的这种方法的缺点是共享文件变得更困难。比如,Alice 不能用她为某个文件指定的名称来告诉 Bob 这个文件,因为那个名称可能在 Bob 的文件名称空间中具有完全不同的意义。

解决这一问题的方法有很多种,但是最通用的方法是为每个客户提供一个部分标准化的名称空间。比如,每个客户可以使用本地目录 /usr/bin 装入包含一些标准程序的文件系统,这些标准程序对每个人都是可用的。同样,目录 /local 也可用作装入位于客户主机上的本地文件系统的标准目录。

一台 NFS 服务器自身可以装入其他服务器输出的目录。但是,它不能再将这些目录输出给它自己的客户。客户必须显式地从维护目录的服务器那里装入该目录,如图 10.6 所示。加入这一限制的部分原因是出于简单性的考虑。如果服务器可以输出它从其他服务器装入的目录,那么它就可能返回包含服务器标识符的特殊文件句柄。NFS 不支持这种文件句柄。

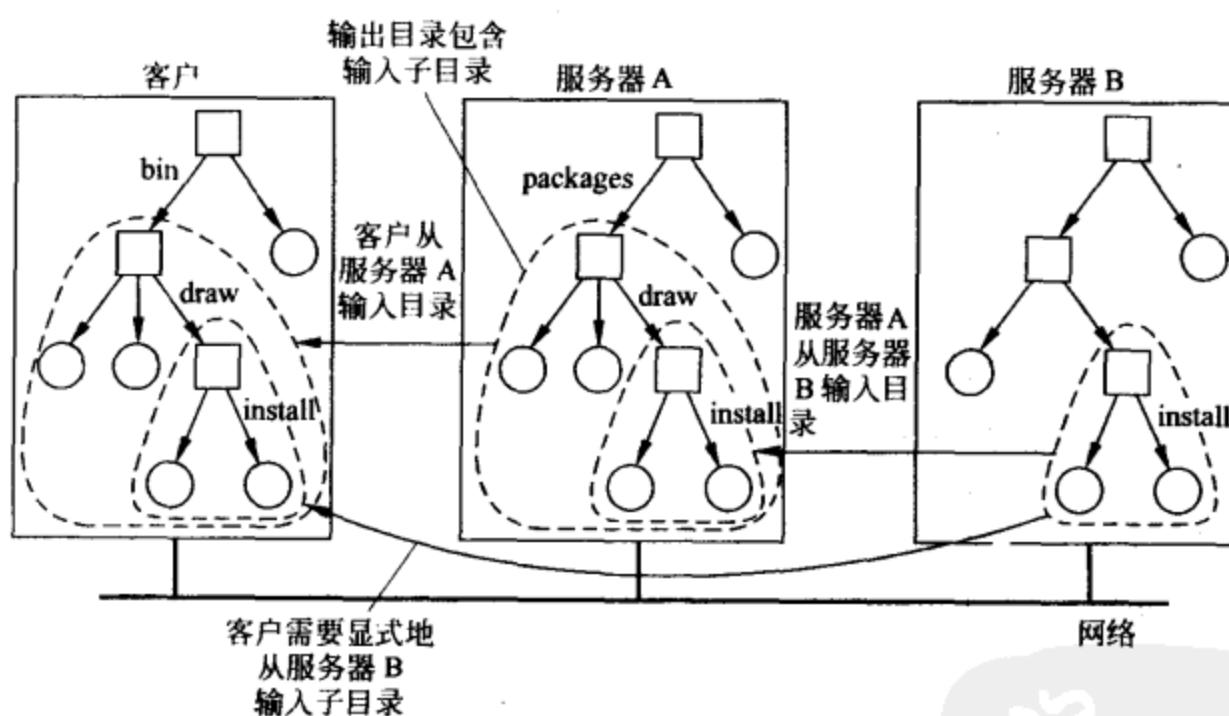


图 10.6 从 NFS 的多台服务器装入嵌套目录

为了解释这一点,我们假设服务器 A 拥有文件系统 FS_A ,并从该文件系统输出目录 /packages。此目录包含子目录 /draw,该子目录是 A 装入的文件系统 FS_B 的挂接点,文件系统 FS_B 是由服务器 B 输出的。设 A 也向它自己的客户输出 /packages/draw,并假设一个客户已经把 /packages 装入它自己的本地目录 /bin,如图 10.6 所示。

如果名称解析是迭代的(如 NFS 版本 3),那么为了解析名称 /bin/draw/install,当客户已经在本地解析了 /bin 时,它联系服务器 A,并请求 A 返回目录 /draw 的文件句柄。

此时,服务器 A 应该返回包含服务器 B 的标识符的文件句柄,因为只有 B 可以解析路径名的剩余部分,即本实例中的/install。如前所述,NFS 不支持这种类型的名称解析。

NFS 版本 3(以及更早版本)一次只能搜索一个单一的文件名,从这种意义上说,它的名称解析是严格迭代的。换句话说,解析诸如/bin/draw/install 之类的名称需要三次单独地调用 NFS 服务器。此外,客户完全负责实现路径名的解析。NFS 版本 4 还支持递归的名称搜索。此时,客户可以将完整的路径名传送给服务器,并请求服务器解析它。

NFS 版本 4 还解决了 NFS 名称搜索的另一个怪异问题。假设一台文件服务器有多个文件系统。在版本 3 的严格迭代名称解析下,每当 lookup 完成一个装入其他文件系统的目录查找时,lookup 就会返回那个目录的文件句柄。随后读取那个目录将返回那个目录下的原始内容,而不是被装入的文件系统的根目录下的内容。

为了解释这一问题,我们假设在先前的实例中,文件系统 FS_A 和 FS_B 属于一个单一的服务器。如果客户已经将/packages 装入它的本地目录/bin,那么在该服务器上搜索文件名 draw 将会返回 draw 的文件句柄。随后使用 readdir 请求该服务器列出的目录项目时,将会返回存储在 FS_A 的子目录/packages/draw 下的原始目录项目列表。只有客户也装入了文件系统 FS_B 时,才可能正确地解析相对于/bin 的路径名 draw/install。

NFS 版本 4 通过允许搜索越过服务器上的挂接点来解决这一问题。特别是,lookup 返回被装入目录的文件句柄,而不是原始目录的文件句柄。客户可以通过检查被搜索文件的文件系统标识符来检测 lookup 是否越过挂接点。如果需要,客户也可以本地装入那个文件系统。

1. 文件句柄

文件句柄是对文件系统内的文件的引用。它与它所引用的文件的名称无关。文件句柄是由该文件系统所在的服务器创建的,并且它在该服务器输出的所有文件系统中是唯一的。它是在创建文件时创建的。客户不知道文件句柄的真实内容,它是完全不透明的。NFS 版本 2 的文件句柄长度是 32 字节,但是 NFS 版本 3 和版本 4 可具有可变长度的文件句柄,版本 3 的文件句柄的最大长度为 64 字节,版本 4 的文件句柄的最大长度为 128 字节。当然,文件句柄的长度是透明的。

在理想情况下,文件句柄作为一个与文件系统有关的文件的真正标识符来实现。首先,这意味着只要文件是存在的,那么它就应该有一个不变的文件句柄。这种持久性需求允许客户只要通过文件名找到了关联的文件,就可以在本地存储文件句柄。一个好处是性能:因为许多文件操作需要使用文件句柄,而不使用文件名,所以客户就可以避免在每个文件操作之前反复地查找名称。这种方法的另一个好处是客户现在可以访问文件,而不依赖于文件的(当前)名称。

因为文件句柄可以被客户存储在本地,所以服务器不能重新使用被删除文件的文件句柄,这一点也是非常重要的。否则,客户使用它在本地存储的文件句柄时,它可能错误地访问错误的文件。

注意,迭代名称搜索和不允许 lookup 操作越过挂接点结合在一起就引出了一个如何获得初始文件句柄的问题。为了访问远程文件系统上的文件,客户需要向服务器提供应

搜索的目录的文件句柄,以及将要解析的文件或目录的名称。NFS 版本 3 使用一个单独的装入协议解决这一问题,客户实际上通过这一协议装入远程文件系统。装入文件系统后,客户获得被装入的文件系统的根文件句柄(root file handle),客户随后使用这个根文件句柄作为搜索名称的起始点。

在 NFS 版本 4 中,这一问题是通过提供一个单独的操作 `putrootfh` 解决的,该操作通知服务器解析与它管理的文件系统的根文件句柄相关的所有文件名。根文件句柄可用于搜索服务器的文件系统上的任何其他文件句柄。这种方法的额外好处是不需要使用单独的装入协议。取而代之的是文件系统的装入可以集成到搜索文件的常规协议之中。客户使用 `putrootfh` 获得文件系统的根文件句柄,通过请求服务器解析相对于这个根文件句柄的名称,就可以简单地装入一个远程文件系统。

2. 自动装入

如前所述,NFS 命名模型本质上为各个用户提供了自己的名称空间。在这个模型中,如果各个用户对同一文件的命名不同,那么可能难以实现文件共享。这个问题的一种解决方案是为每个用户提供一个部分标准化的本地名称空间,然后对每个用户以同样的方式装入远程文件系统。

NFS 命名模型的另一个问题是应该何时装入远程文件系统。以一个具有成千上万个用户的大型系统为例。假设每个用户都有一个本地目录`/home`,该目录用于装入其他用户的主目录。比如,虽然 Alice 的文件实际存储在一台远程服务器,但是她可以通过本地可用的`/home/alice` 访问她的主目录。这个目录可以在 Alice 登录到她的工作站时自动装入。另外,她可能经由`/home/bob` 访问 Bob 的目录,从而访问 Bob 的公共文件。

但是,问题是 Bob 的主目录是否也应该在 Alice 登录时自动装入。这种方法的好处是整个装入文件系统的工作都是对 Alice 透明的。但是,如果每个用户都使用这一策略,那么登录可能导致通信和管理的许多额外开销。另外,它可能要求所有用户都是事先已知的。另一个更好的方法是根据需要,也就是第一次需要使用时,透明地装入其他用户的主目录。

在 NFS 中,根据需要装入一个远程文件系统(或实际是一个输出目录)是由自动装入器(automounter)处理的,它以一个单独进程的形式运行于客户机器上。自动装入器的基本原理是相对比较简单的。我们来考虑一个作为 UNIX 操作系统上的用户级 NFS 服务器实现的简单自动装入器(关于其他可选择的实现,请参看(Callaghan 2000))。

假设每个用户可以通过本地目录`/home`访问所有用户的主目录,如上所述。客户端机器启动时,自动装入器开始装入这个目录。这个本地装入的效果是每当程序试图访问`/home`时,UNIX 核心会把 `lookup` 操作转发给 NFS 客户,此时,NFS 客户将该请求转发给作用相当于 NFS 服务器的自动装入器,如图 10.7 所示。

比如,假设 Alice 登录进入系统。登录程序会试图读取目录`/home/alice`以找到诸如登录脚本的信息。因而,自动装入器会接收到搜索子目录`/home/alice`的请求,因此它首先在`/home`中创建`/alice`。然后,它搜索输出 Alice 的主目录的 NFS 服务器以便随后

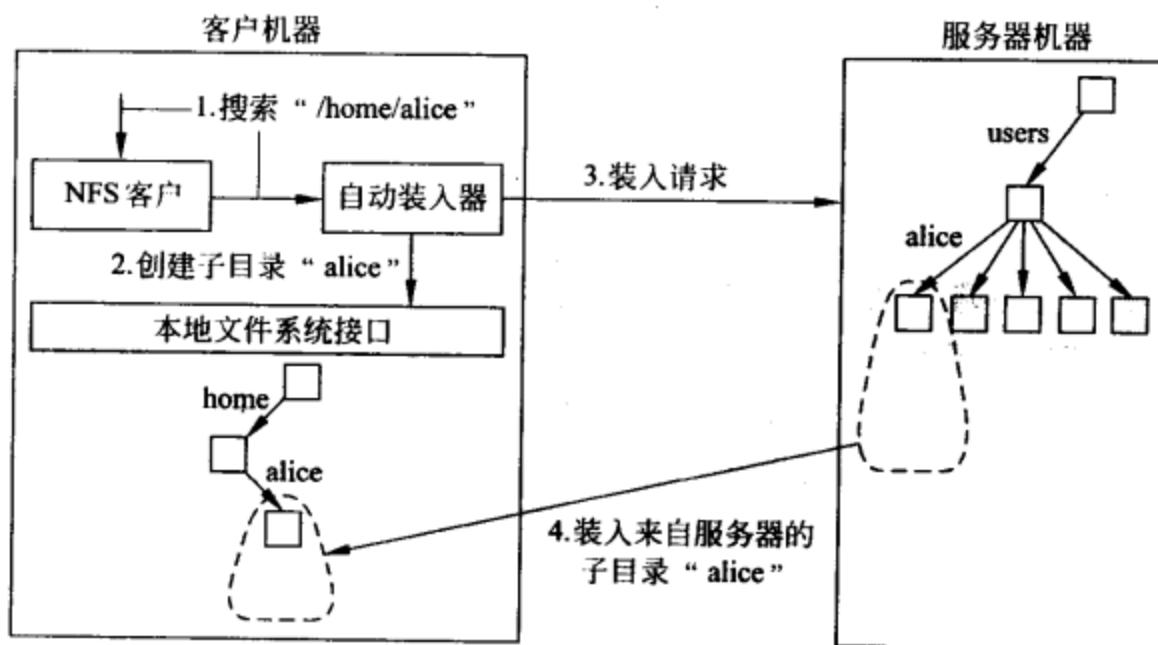


图 10.7 简单的 NFS 自动装入器

在 /home/alice 中装入那个目录。此时，登录程序可以继续执行。

这种方法的问题是自动装入器将不得不参与所有的文件操作以保证透明性。如果一个被引用的文件因为相应的文件系统还没有被装入，从而不能在本地可用，那么自动装入器将必须知道这一事实。特别是，它将需要处理所有的读请求和写请求，即使是已经装入的文件系统的读请求和写请求。这种方法可能导致严重的性能问题。使自动装入器只装入和卸除目录，而在其他时候不参与循环可能会更好。

一个简单的方法是让自动装入器在特定子目录中装入目录，并为每个被装入的目录安装一个符号链接。图 10.8 表示这种方法。

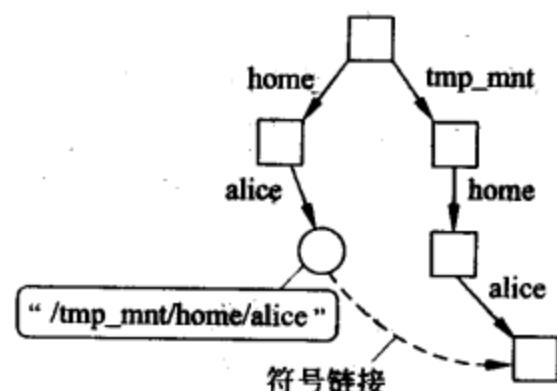


图 10.8 自动装入时使用符号链接

这个实例将用户主目录作为 /tmp_mnt 的子目录装入。当 Alice 登录时，自动装入器在 /tmp_mnt/home/alice 安装她的主目录，并创建一个指向该子目录的符号链接 /home/alice。此时，每当 Alice 执行一个命令：

```
ls -l /home/alice
```

系统不需要自动装入器的进一步参与而直接联系输出 Alice 主目录的 NFS 服务器。

3. 文件属性

NFS 文件有很多关联的属性。在版本 3 中,属性的集合是固定不变的,每个 NFS 实现都需要支持那些属性。但是,因为传统上 NFS 主要是按照 UNIX 文件系统进行模型化的,所以,在其他文件系统(比如,Windows)上完全实现 NFS 有时是非常困难的,甚至是不可能的。

在 NFS 版本 4 中,文件属性的集合分为每个实现都必须支持的强制属性的集合、希望得到支持的推荐属性的集合以及命名属性的附加集合。

命名属性实际上不是 NFS 协议的一部分,而被编码为(属性,值)对组成的数组,其中属性表示为一个字符串,其值表示为不可解释的字节序列。它们和文件(或目录)一起存储,NFS 提供各种操作来读、写这些属性值。但是,属性及其值的解释完全由应用程序负责处理。

图 10.9(a)表示一些强制文件属性(共有 12 个)。这些属性也出现在 NFS 的以前版本中。值得注意的是 CHANGE 属性,该属性可能是一个服务器定义的属性,客户可以通过该属性查看文件是否已经修改,或者最后一次修改的时间。

属性	描述
TYPE	文件的类型(常规、目录、符号链接)
SIZE	文件的长度,以字节为单位
CHANGE	供客户端查看文件是否和/或何时改变的指示器
FSID	文件所在文件系统的标识符,它在服务器上是惟一的

(a)

属性	描述
ACL	文件所关联的访问控制列表
FILEHANDLE	此文件的文件句柄,由服务器所提供
FILEID	此文件的惟一标识符,它在文件系统上是惟一的
FSLOCATIONS	可在网络中找到此文件系统的位置
OWNER	文件所有者的字符串名称
TIMEACCESS	文件最后一次访问的时间
TIMEMODIFY	文件最后一次修改的时间
TIMECREATE	文件的创建时间

(b)

图 10.9 一些 NFS 通用文件属性

(a) 强制文件属性; (b) 推荐文件属性

图 10.9(b)列出了一些推荐属性。当前的 NFS 版本 4 总共支持 43 个推荐文件属性。一个重要的属性是访问控制列表,它代替了 UNIX 类型的容许位。另一个属性列出了可找到文件系统的副本的位置。比如,如果当前服务器不可用,客户可使用这个列表联系其他服务器。(Shepler 等 2000)描述了推荐属性的完整列表。

10.1.5 同步

分布式文件系统中的文件通常由多个客户共享。如果系统从不发生文件共享,那么使用分布式文件系统首先就是不合理的。不幸的是,共享是有一定代价的:为保证共享文件保持一致,需要进行同步。如果文件保留在一台中心服务器,那么实现同步就相对比较简单。但是,这样做会造成潜在的性能问题。因此,当客户读取文件内容、向文件写入内容时,通常允许它们在本地保留一份该文件的拷贝。注意,这种实现相当于图 10.1(b) 的下载/上载模型。详细讨论同步之前,我们先看一看共享文件意味着什么。

1. 文件共享的语义

当两个或更多用户共享同一文件时,为避免出现问题,精确定义读操作和写操作的语义是十分必要的。为了解释 NFS 中文件共享的语义,我们先考虑几个与文件共享有关的普遍问题。

在允许进程共享文件的单处理器系统,比如 UNIX 中,语义通常规定 read 操作出现在 write 操作之后时,read 操作返回刚写入的值,如图 10.10(a) 所示。同样,当相继很快地发生两个 write 操作后,又发生一个 read 操作,那么所读取的值是最后一个 write 操作写入的值。实际上,系统对所有操作都强加了一个绝对时间顺序,系统总是返回最新值。我们称这个模型为 UNIX 语义(UNIX semantics)。这个模型易于理解,实现起来也相当简单。

如果在分布式系统中,只有一个文件服务器而客户不缓存文件,那么实现 UNIX 语义十分容易。所有 read 操作和 write 操作直接传送到文件服务器,该文件服务器以严格的顺序执行它们。这种方法给出了 UNIX 语义(只是它有个小问题,那就是,网络延迟可能导致落后于 write 操作 1ms 发生的 read 操作先达到服务器,从而导致 read 操作得到旧值)。

但是,实际上,所有文件请求都必须传送到一个单一服务器的分布式系统的性能通常是很差的。通常,解决这个问题的方法是允许客户在它们私有(本地)高速缓存中保留频繁使用的文件的本地拷贝。尽管我们下面将详细讨论文件缓存,但是我们现在可以指出,如果客户在本地修改被缓存的文件,而稍后很短的时间内另一个客户从该服务器读取这个文件,那么第二个客户会得到过时的文件,如图 10.10(b) 所示。

解决这一难题的方法是立即将被缓存文件的所有改动传播到服务器。尽管这种方法在概念上相当简单,但是它并不有效。另一种可选择的方法是放宽文件共享的语义。我们可以不要求 read 操作看到所有先前的 write 操作的结果,而定义一套新的规则:“对打开的文件所做的改动最初只对修改这个文件的进程(或者可能是机器)可见。只有当文件被关闭时,这些改动才对其他进程(或机器)可见。”采用这一规则不会改变图 10.10(b) 所发生的情况,但是它确实将实际行为(B 得到文件的原值)重新定义为正确的行为。当 A 关闭文件时,它向服务器发送一份拷贝以便随后的读操作像我们要求的那样得到新值。这一规则被广泛地实现,并被称为会话语义。与大多数分布式系统一样,NFS 也实现了会话语义。这意味着,尽管 NFS 在理论上遵循图 10.1(a) 中的远程访问模型,但是大多数实现使用本地高速缓存,有效地实现了图 10.1(b) 中的上载/下载模型。

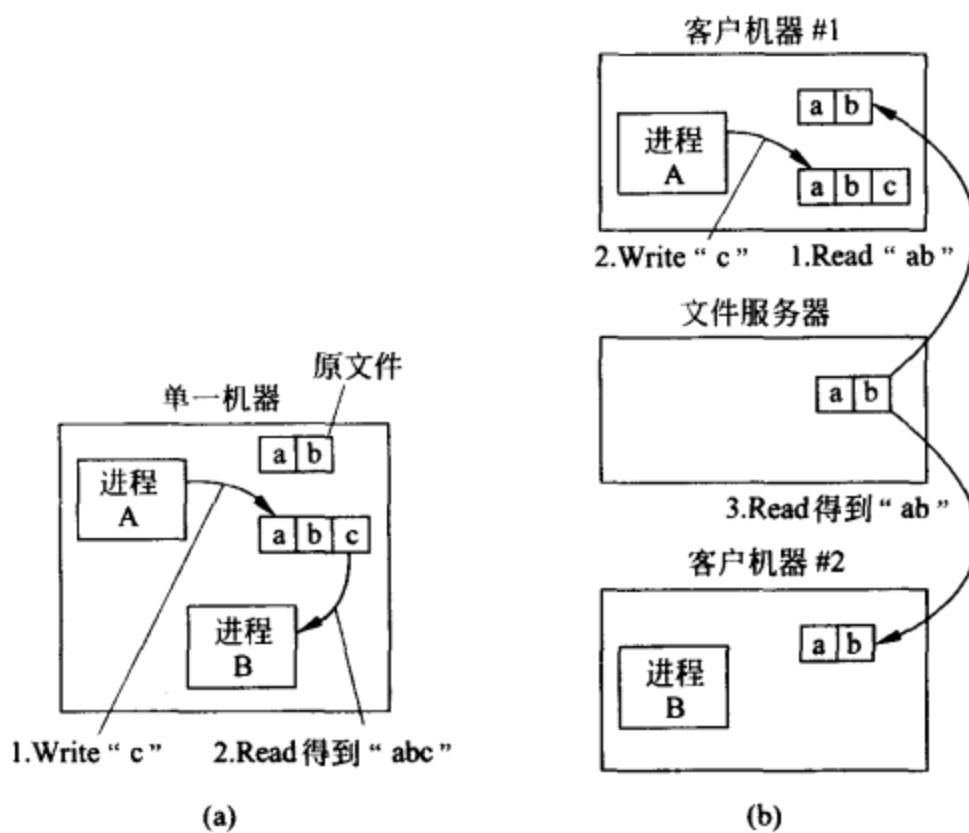


图 10.10 read 操作的返回值

- (a) 单处理器中, read 操作出现在 write 操作之后时, read 操作返回的值是刚刚写入的值;
 (b) 使用缓存的分布式系统可能返回过时的值

使用会话语义提出了以下问题,即如果两个或更多客户同时缓存和修改同一文件会发生什么情况。一种方法是在依次关闭每个文件时,将文件的值传送给服务器,所以最后的结果依赖于服务器最后处理的那个关闭请求。另一种稍差却相对易于实现的方法是最后的结果是几个候选者之一,但是并不指定选择哪个候选者。

解决分布式系统中文件共享的语义的一种完全不同的方法是使所有文件都不可改变。因而,系统无法为写操作打开文件。结果,文件上的操作仅有 create 和 read。

创建一个全新的文件并使它以原先存在的文件名存入目录系统是可能的,而此时那个原先存在的文件将变成不可访问的(至少不能以那个名称访问)。因而,尽管修改文件 x 变成不可能的,但是用一个新文件自动替换 x 仍然是可能的。也就是说,尽管文件不能更新,但是目录能更新。一旦我们决定文件根本不能改变,如何处理两个进程——其中一个进程写文件,而另一个进程读同一文件——的问题也就随之消失了。

剩下的问题是两个进程试图同时替换同一文件时会发生什么情况。因为使用会话语义,所以解决这个问题的最好方法好像是允许使用其中一个新文件替换旧文件,使用最后一个新文件或者不确定使用哪个新文件。

一个颇为困难的问题是如果一个文件被替换了,而其他进程正忙于读取该文件,那么如何处理这种情况。一种方法是设法安排阅读器继续使用旧文件,即使该文件可能不再存在于任何目录,这种方法类似于 UNIX 允许打开文件的进程,即使在该文件已经从所有目录被删除时,仍继续使用该文件。另一种方法是检测文件已经被改变并使随后试图从该文件读取数据的请求失败。

第四种处理分布式系统的共享文件的方法是使用我们在第 5 章详细讨论的事务处

理。简要地总结一下,为访问一个文件或一组文件,一个进程先执行某种类型的 BEGIN_TRANSACTION 原语以标志其后的语句必须不可分割地执行。然后是读、写一个或多个文件的系统调用。所有工作都完成后,执行一个 END_TRANSACTION 原语。这种方法的关键特性是服务器保证包含在一个事务处理内的所有调用都被顺序执行,而不会受到其他并发事务的干扰。如果两个或更多事务同时启动,那么系统保证最后的结果就像它们都按照某种(未定义的)序列顺序运行一样。

图 10.11 总结了已讨论的 4 种处理分布式系统中共享文件的方法。

方法	注释
UNIX 语义	一个文件上的每个操作对所有进程都是瞬间可见的
会话语义	文件关闭之前,所有改动对其他进程都是不可见的
不可改变的文件	不允许更新文件;简化了共享和复制
事务处理	所有改动以原子操作的方式发生

图 10.11 4 种处理分布式系统中共享文件的方法

2. NFS 中的文件锁定

对于一个可能包含无状态服务器的分布式文件系统来说,锁定文件可能令人疑惑不解。在 NFS 的传统中,文件锁定是通过一个单独协议处理的,由一个(有状态的)锁管理器实现这个协议。但是,出于多种原因,使用 NFS 锁定协议实现文件锁定的方法从未得到广泛应用,其原因是这个协议过于复杂,而且可能导致执行效果较差的实现,甚至错误的实现(Shepler 1999)。在版本 4 中,文件锁定集成到 NFS 文件访问协议之中。希望这种方法会使客户更易于使用文件锁定,而不需要求助于哪些用来代替 NFS 版本 3 的锁定方法的特殊方法。

由于客户和服务器可能在处于锁定状态时发生故障,分布式文件系统中的文件锁定变得更加复杂。因而,为确保共享文件的一致性,正确恢复变得尤为重要。我们将在下面讨论 NFS 的容错问题时再讨论这个问题。

从概念上说,NFS 版本 4 的文件锁定比较简单。它本质上只提供 4 种与锁定相关操作,如图 10.12 所示。NFS 对读操作锁和写操作锁加以区分。如果多个客户只读取数据,那么它们可以同时访问一个文件的相同部分。为了获得修改文件的某个部分的独占访问权,需要使用写操作锁。

操作	描述
lock	为一定范围的字节创建锁
lockt	测试是否已经授予冲突的锁
locku	删除一定范围的字节上的锁
renew	更新一个指定锁上的租用

图 10.12 NFS 版本 4 中关于文件锁定的操作

`lock` 操作用于请求对文件的一段连续范围的字节加读操作锁或写操作锁。它是一个非阻塞操作,如果由于另一个冲突的锁的存在而锁不能授予操作,那么客户会得到一个错误消息,并不得不在稍后的时间轮询服务器。另一种可选择的方法是,客户可以请求将其放入服务器维护的 FIFO 列表。一旦冲突的锁被删除,服务器将把下一个锁授予位于该列表前端的客户,只要该客户在一定的时间期限内轮询服务器。这种方法避免了服务器必须通知客户的操作,但对那些锁请求没能被批准的客户仍然保持了公平,因为锁的授予是按照 FIFO 顺序进行的。

`lockt` 操作用于测试是否存在冲突的锁。比如,客户在请求一个文件的指定范围的字节上的写操作锁之前,可以测试哪些字节是否已经被授予了读操作锁。在发生冲突的情况下,提出请求的客户被告知哪个客户造成这个冲突,以及在哪段范围的字节上发生冲突。这个操作的实现可以比 `lock` 的实现更为高效,因为它不需要打开文件。

删除一个文件上的锁是通过 `locku` 操作完成的。

被授予的锁在一段指定时间(由服务器决定)内有效。换句话说,它们都与一个租用相关联。除非客户更新授予其锁上的租用,否则服务器会自动删除它的锁。我们将看到,服务器端提供的其他资源也采用这种方法,这种方法也有助于故障后的恢复。客户使用 `renew` 操作来请求服务器更新其锁上的租用(实际上,其他资源也是这样的)。

除了这些操作之外,还有一种锁定文件的隐含方法,称为共享预约(*share reservation*)。共享预约完全独立于锁定,它可被用于在基于 Windows 的系统上实现 NFS。客户打开文件时,它指定它所需的访问类型(即 READ、WRITE 或 BOTH),以及服务器应该拒绝的其他客户的访问类型(NONE、READ、WRITE 或 BOTH)。如果服务器不能满足客户的需求,那么该客户的 `open` 操作将会失败。图 10.13 确切地表示了一个新客户试图打开一个已被另一个客户成功打开的文件时所发生的情况。对于一个已经打开的文件,我们用两个不同的状态变量加以区分。访问状态表明当前客户目前如何访问该文件。拒绝状态表明新客户不允许进行哪些访问。

图 10.13(a) 表示给定一个文件的当前拒绝状态时,客户试图请求以指定的访问类型打开此文件时所发生的情况。同样,图 10.13(b) 表示试图打开一个当前正被另一个客户访问的文件而所请求的访问类型不被该客户允许的情况。

当前文件拒绝状态					
请求访问	NONE	READ	WRITE	BOTH	
	READ	成功	失败	成功	失败
	WRITE	成功	成功	失败	失败
	BOTH	成功	失败	失败	失败

(a)

请求的文件拒绝状态					
当前访问状态	NONE	READ	WRITE	BOTH	
	READ	成功	失败	成功	失败
	WRITE	成功	成功	失败	失败
	BOTH	成功	失败	失败	失败

(b)

图 10.13 使用 NFS 共享预约实现 `open` 操作的结果

(a) 在当前拒绝状态下,客户请求共享访问时的情况; (b) 在当前文件访问状态下,客户请求拒绝状态时的情况

10.1.6 缓存和复制

与许多分布式文件系统一样, NFS 也广泛使用客户缓存来提高性能。另外, 它也对文件复制提供最低限度的支持。

1. 客户端缓存

NFS 版本 3 中的缓存主要在协议外部处理。这种方法导致了不同缓存策略的实现, 而其中大部分实现从不保证一致性。与存储在服务器上的数据相对照, 在最理想的情况下, 被缓存的数据可能在几秒钟之内就过时。但是, 允许被缓存的数据在客户完全不知晓的情况下在 30 秒内过时的实现也是存在的。这种情况就很难令人满意。

NFS 版本 4 解决了一些有关一致性的部分, 但是它本质上仍以实现相关的方式实现高速缓存一致性。图 10.14 表示了 NFS 假定的通用缓存模型。每个客户都可以有一个存储器高速缓存, 它包含先前从服务器读取的数据。另外, 客户端还有一个磁盘高速缓存, 它作为存储器高速缓存的扩展, 并使用相同的一致性参数。

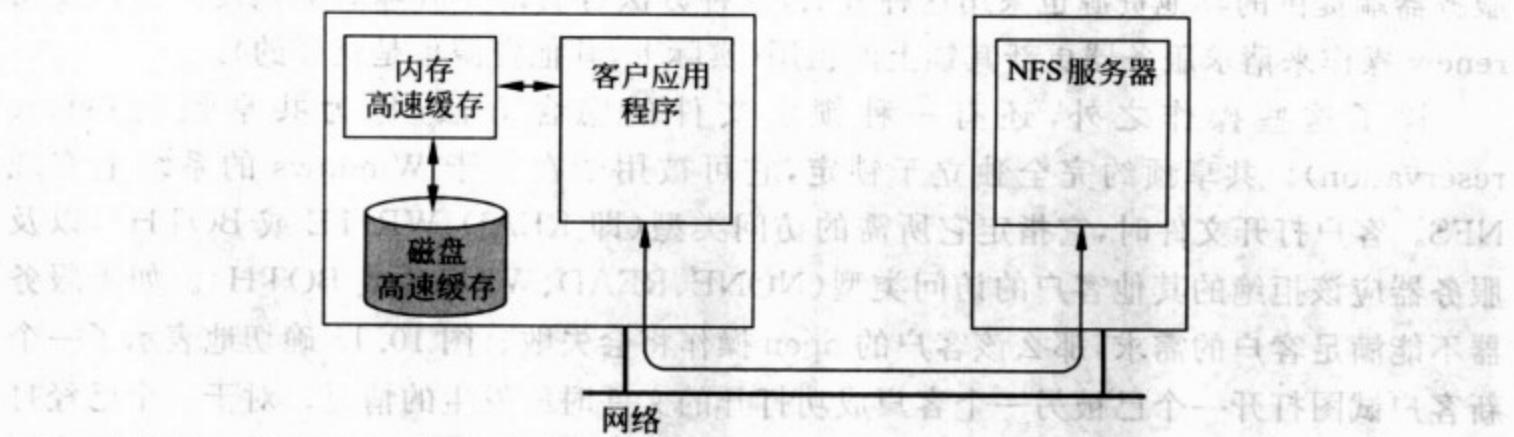


图 10.14 NFS 中的客户端缓存

通常, 客户缓存文件数据、属性、文件句柄和目录。处理被缓存的数据、被缓存的属性等的一致性可采用不同的策略。下面, 我们先看看缓存文件数据的情况。

NFS 版本 4 支持两种缓存文件数据的方法。最简单的方法是客户打开文件时, 将它从服务器获得的数据缓存, 并将这些缓存数据作为各种 read 操作的结果。另外, write 操作也可以在高速缓存中实现。客户关闭文件时, NFS 要求如果文件被修改, 那么被缓存的数据必须送回服务器。这种方法相当于实现上面讨论的会话语义。

一旦文件(文件的一部分)被缓存, 客户就可以在高速缓存中保留它的数据, 即使在关闭文件之后。同一台机器上的多个客户也可以共享一个高速缓存。NFS 要求, 每当客户打开一个已关闭但曾被(部分)缓存的文件, 客户必须马上使被缓存的数据重新有效。这是通过检查文件最后被修改的时间并使包含过时数据的高速缓存无效来实现的。

NFS 规范的一个新增特性是打开文件时, 服务器可以将它的某些权限委派给客户。打开委派发生在客户机器被允许在本地处理来自同一机器的其他客户的 open 和 close 操作时。比如, 因为需要考虑共享预约, 所以常常由服务器负责检查打开文件的操作是否成功。使用打开委派时, 客户机器有时被允许做这种检查, 以避免联系服务器的需要。

比如,如果服务器将文件的打开操作委派给请求写访问权限的客户,那么来自同一台机器上的其他客户的文件锁定也可以在本地处理。服务器仍会处理来自其他机器上的客户的锁定请求,它只拒绝那些客户对文件的访问。注意,将文件委派给请求只读访问权限的客户时,这一方法无法工作。在这种情况下,每当另一个本地客户要获得写访问权限时,该客户将不得不联系服务器,在本地处理这个请求是不可能的。

将文件委派给客户的一个重要后果是服务器需要有能力撤销这种委派,比如,当另一个位于不同机器上的客户需要获得该文件的访问权时,服务器就需要撤销文件的委派。撤销委派要求服务器能够回叫客户,如图 10.15 所示。

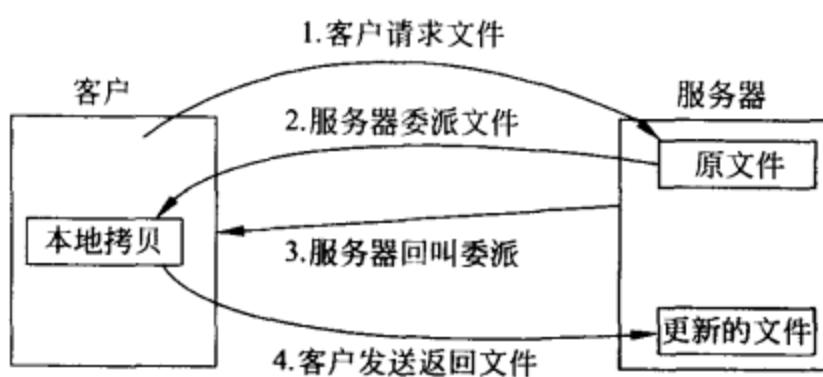


图 10.15 使用 NFS 版本 4 回叫机制撤销文件委派

在 NFS 中,回叫是通过 NFS 底层的 RPC 机制实现的。但是,注意,回叫要求服务器记录已委派文件的客户。这里,我们再看一个实例,该实例中的 NFS 服务器不再以无状态的方式实现。如同我们下面讨论的,将委派和有状态的服务器结合起来可能在出现客户和服务器故障时引起各种问题。

客户也可以缓存属性的值,但是客户何时达到保持被缓存的属性值的一致主要取决于客户自身。特别是,两个不同客户缓存的同一文件的属性值可能有所不同,除非这两个客户共同维持这些属性的一致。对属性值的修改应该立即转发到服务器,这样就遵循了直写式高速缓存相关性策略。

缓存文件句柄(确切地说,名称到文件句柄的映射)和目录也使用相似的方法。NFS 使用被缓存的属性、文件句柄和目录上的租用来减轻不一致性的影响。由此,一段时间后,客户自动使高速缓存数据项无效,再次使用这些数据项时,需要使它们重新有效。

2. 复制服务器

NFS 版本 4 对文件复制提供最低限度的支持。只有整个文件系统可以被复制(即,由文件、属性、目录和数据块组成的逻辑块设备)。对复制的支持是以 FS_LOCATIONS 属性的形式提供的,该属性是每个文件的推荐属性。这一属性提供了一个可能存在文件系统的位置列表,处于这些位置的文件系统含有该属性关联的文件。每个位置是以 DNS 名称或 IP 地址的形式给出的。注意,实际提供复制服务器的方法取决于特定的 NFS 实现。NFS 版本 4 并未规定如何实现复制。

10.1.7 容错性

在最新版 NFS 之前的各版本 NFS 中,容错根本不构成问题。其原因是 NFS 协议不要求服务器是有状态的。所以,从服务器崩溃状态恢复服务器是相当简单的,这是因为没有状态会丢失。当然,在单独的锁管理器维护状态的情况下,需要采取特殊的措施。

由于 NFS 版本 4 放弃了无状态的设计方案,所以它需要处理容错和恢复问题。特别是,以下两种情况会用到状态:文件锁定和委派。另外,我们还需要采取特殊措施来处理位于 NFS 协议底层的 RPC 机制的不可靠性。下面,我们先从 RPC 故障开始分别讨论这些问题。

1. RPC 故障

NFS 所用的 RPC 机制存在一个问题:它不能保证可靠性。实际上,所生成的客户端的和服务器端的 RPC 存根可能基于可靠的、面向连接的传输协议,如 TCP,也可能基于不可靠的、无连接的传输协议,如 UDP。

NFS 的底层 RPC 语义存在的主要问题是它缺少对重复请求的检测。因此,当 RPC 响应丢失而客户又成功地重传原先请求时,服务器最终会多次执行该请求。处理非幂等操作时,这种情况是不应该发生的。

由服务器实现的重复请求高速缓存(duplicate-request cache)(Juszczak 1990)可以减轻这些问题。每个来自客户的 RPC 请求的头部都带有一个惟一的 XID(transaction identifier 事务处理标识符),当 RPC 请求到来时,服务器缓存其标识符。只要服务器还没有发送响应,就说明服务器正在处理这个 RPC 请求。当服务器处理了某个请求后,也缓存该请求的关联响应,此后该响应才返回客户。

现在,需要处理的情况有三种。第一种情况,如图 10.16(a)所示,客户发送一个请求,并启动一个计时器。如果在计时器超时之前,客户没有收到响应,那么客户就使用与原先请求的 XID 相同的 XID 重传原先的请求。因此,如果服务器还没有处理完原先的请求,那么它只需忽略这个重传的请求。

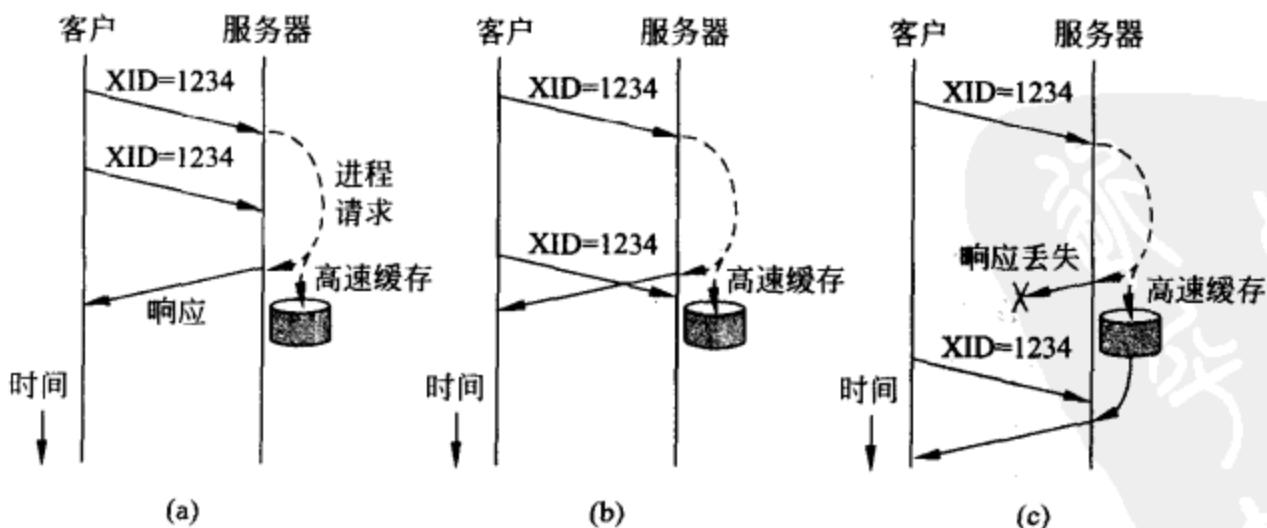


图 10.16 处理重传的三种情况

(a) 仍在处理请求; (b) 刚刚返回响应; (c) 早已返回响应,但是响应丢失

第二种情况，服务器可能在刚向客户端返回响应后就接收到客户重传的请求。如果重传请求的到达时间与服务器发送应答的时间非常接近，那么服务器断定重传和响应相互交叉，所以，服务器再次忽略重传的请求。图 10.16(b) 表示了这种情况。

第三种情况，响应确实丢失了，应该向客户端发送操作的缓存结果以响应重传的请求，如图 10.16(c) 所示。注意，文件操作的结果仍应该缓存，这是因为系统无法保证这次重传必然成功。后面的习题将系统地研究如何解决这种问题及其相关的问题。

2. 出现故障时的文件锁定

NFS 版本 3 中的文件锁定是由单独的服务器处理的。这样，NFS 协议的无状态性质仍可保留，与容错相关的问题将由锁服务器处理。但是，由于 NFS 版本 4 引入了文件锁，它必然要提供一种处理客户和服务器崩溃的基本机制，以此作为 NFS 协议的一部分。我们要解决的问题相对比较简单。为了锁定一个文件，客户向服务器提出锁定请求。假设服务器批准这个请求，那么当客户或服务器崩溃时，我们就可能陷入困境。

为了解决客户崩溃问题，服务器为它授予的每个锁分配一个租用。当租用到期时，服务器将删除对应的锁，从而释放其关联的资源（即，文件）。为了防止服务器删除锁，客户应在其租用到期之前更新租用。为了达到这一目的，NFS 提供了前面介绍的 renew 操作。

即使客户没有崩溃，还是可能发生锁被删除的情况。特别是，如果客户无法与服务器联络以更新租用时，比如由于网络（暂时）断开，就会导致锁被错误地删除。NFS 不采取任何特殊措施处理这种情况。

服务器崩溃并随后得到恢复时，它可能会丢失它授予客户的锁的信息。NFS 版本 4 采用的方法是进入宽限期（grace period），在这个宽限期期间，客户可以要求恢复曾授予它的锁。这样，服务器就会建立锁的原先状态。注意，这里的锁恢复与其他丢失数据的恢复无关。在宽限期期间，通常只接收要求归还锁的请求，宽限期结束之前，正常的锁请求将会被拒绝。

注意，使用租用引起了许多问题，NFS 只解决了其中的部分问题。比如，租用要求客户和服务器同步它们的时钟。如果服务器声明租用在时间 T 过期，那么客户需要知道服务器的当地时间。另一方面，如果服务器规定租用在某个时间段 D 之后过期，那么它向客户端发送租用的时间应是公开的。在广域系统中，这些时间问题都难以得到解决。

另一个问题是向服务器发送租用更新消息的可靠性问题。如果这个消息的传输失败或被延迟，那么租用可能意外地过期。不论哪种情况，服务器都不得不在客户可以继续使用文件之前，显式地向它分配一个新的租用。

3. 出现故障时的打开委派

当客户或服务器崩溃时，打开委派引起了另外的问题。首先，我们来考虑执行被委派文件的打开操作的客户。如果这个客户崩溃，而它可能无法立即将更新传播到服务器而一直在它的本地拷贝上工作。也就是说，客户遵循回写式高速缓存相关性策略。此时，除非客户的更新被存入本地的稳定存储器，否则完全恢复文件是不可能的。总之，客户对文件恢复负部分责任。

服务器崩溃并随后得到恢复时,它的处理过程与锁恢复类似。服务器再次启动后,被委派文件打开操作的客户将要求服务器归还打开委派。但是,与锁定不同,服务器强迫客户向服务器返回所有修改,有效地撤销委派。

这种方法带来两个结果。第一,服务器现在是最新的,它获得了委派给客户的每个文件的最新修改内容。第二,服务器再次完全控制了文件,此后,它可以决定将文件委派给其他客户。

10.1.8 安全性

如前所述,NFS的基本思想是远程文件系统应该像是客户的本地文件系统那样提供给客户。从这个角度来看,NFS中的安全性主要集中于客户和服务器之间的通信就不足为怪了。安全通信意味着两者之间的安全通道应当以我们在第8章所讨论的方式建立。

除了安全RPC之外,控制对文件的访问也是十分必要的,这是通过NFS中的访问控制文件属性处理的。文件服务器负责验证其客户的访问权限,我们将在下面解释其工作方法。结合了安全RPC,NFS安全性体系结构如图10.17所示。

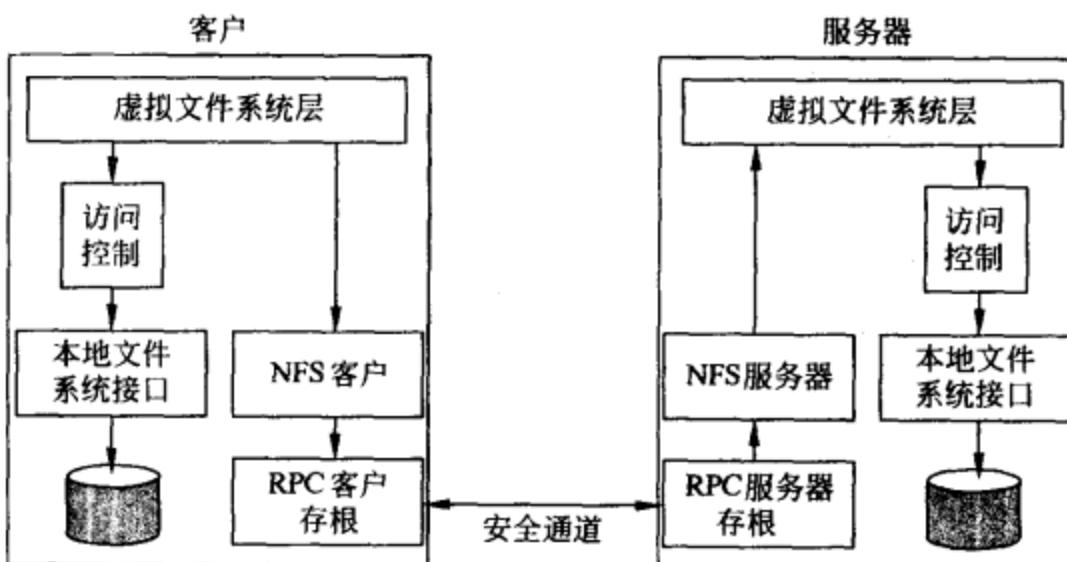


图 10.17 NFS 安全性体系结构

1. 安全 RPC

因为NFS位于RPC系统的层次之上,所以建立NFS中的安全通道的关键在于建立安全RPC。在NFS版本4之前,安全RPC仅意味着身份验证。进行身份验证可采用三种方法。现在,我们将依次研究这三种方法。

应用最广泛的方法称为系统身份验证,它实际上几乎不能进行任何身份验证。在这种基于UNIX的方法中,客户只向服务器发送它的有效用户ID、组ID以及它声称自己所属的组的列表。这一信息是以无签名的明文形式发送到服务器的。也就是说,服务器无法证实发送者声称的用户和组标识符是真正与该发送者关联的标识符。实质上,服务器假设客户已经通过一个正确的登录过程,并假设服务器可以信任该客户所在的机器。

老版本NFS所用的第二种身份验证方法是使用Diffie-Hellman密钥交换建立一个会话密钥,所以使用这种身份验证方法的NFS被称为安全NFS(secure NFS)。我们在第

8 章解释了 Diffie-Hellman 密钥交换的工作方式。这种方法比系统身份验证方法好很多,但是它比较复杂,也正是因为这一点,它较少得到实现。Diffie-Hellman 可以视为是一个公共密钥加密系统。最初,它无法安全地分发服务器的公共密钥,但是由于后来引入了安全名称服务器,这一问题得以解决。长期以来,批评家一直批评的是其相对较小的公共密钥,NFS 的公共密钥只有 192 位。事实证明,破解一个使用如此短的密钥的 Diffie-Hellman 系统几乎是毫不费力的(Lamacchia 和 Odiyko 1991)。

第三种身份验证协议是 Kerberos(版本 4),我们也在第 8 章讨论过。

随着 NFS 版本 4 的引入,由于它支持 RPCSEC_GSS,因此安全性得以提高。RPCSEC_GSS 是一个通用的安全性框架,它可以支持各种建立安全通道的安全性协议(Eisler 等 1997)。特别是,它不仅支持不同身份验证系统的挂钩(hook),而且支持消息的完整性和机密性,这两种特性是老版本 NFS 无法支持的。

RPCSEC_GSS 基于安全性服务的标准接口,该接口称为 GSS-API,我们曾在第 8 章简要介绍过此接口,(Linn 1997)完整地描述了此接口。RPCSEC_GSS 所在的层在此接口所在的层之上,图 10.18 表示其组织结构。

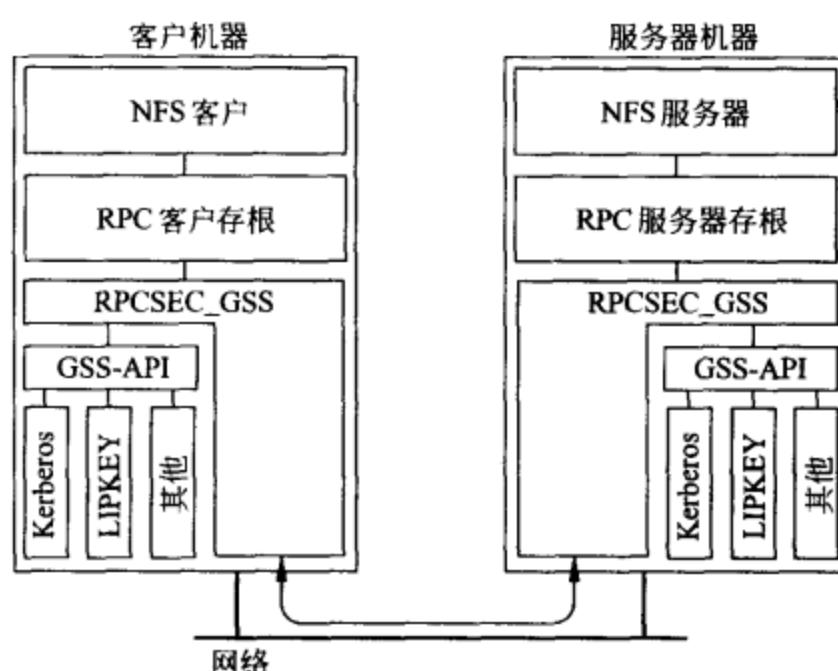


图 10.18 NFS 版本 4 中的安全 RPC

对于 NFS 版本 4,RPCSEC_GSS 应该配置为支持 Kerberos 版本 5。另外,系统也必须支持一个称为 LIPKEY 的方法,该方法在(Eisler 2000)中有所描述。LIPKEY 是一个公共密钥系统,它允许使用密码验证客户,而使用公共密钥验证服务器。

NFS 的安全 RPC 的重要部分是设计者们选择不仅支持他们自己的安全性机制,而且支持处理安全性的标准方法。因此,已被证明的安全性机制,如 Kerberos 可以结合到 NFS 实现之中,而不影响系统的其他部分。同样,如果现存的安全性机制被证明有缺陷(比如,使用小密钥的 Diffie-Hellman 的情况),那么替换它是十分容易的。

应该注意,因为 RPCSEC_GSS 是作为 RPC 层的一部分实现的,RPC 层位于 NFS 协议层之下,所以它也可以用于老版本的 NFS。但是 RPC 层的这种改变只在引入 NFS 版本 4 之后才变得有效的。

2. 访问控制

NFS 中的授权类似于安全 RPC：它提供一些机制，但是不指定任何特定的策略。访问控制是通过 ACL 文件属性实现的。这一属性是访问控制项目的列表，其中，每个项目指定某个特定用户或组的访问权限。NFS 所区分的访问控制操作是相对比较直接的，图 10.19 列出了这些操作。

操作	描述
read_data	允许读取文件中的数据
write_data	允许修改文件的数据
append_data	允许在文件最后追加数据
execute	允许执行文件
list_directory	允许列出目录的内容
add_file	允许向目录增加新文件
add_subdirectory	允许在目录中创建子目录
delete	允许删除文件
delete_child	允许删除目录中的文件或目录
read_acl	允许读取 ACL
write_acl	允许写 ACL
read_attributes	能够读取文件的其他基本属性
write_attributes	允许改变文件的其他基本属性
read_namedattrs	允许读取文件的命名属性
write_namedattrs	允许写文件的命名属性
write_owner	允许改变所有者
synchronize	允许在服务器本地使用同步的读操作和写操作访问文件

图 10.19 NFS 所识别的关于访问控制的操作的分类

与诸如 UNIX 系统中的简单访问控制机制相比，NFS 能够区分很多不同类型的操作。值得注意的还是 synchronize 操作，该操作本质上表示一个与服务器共存一处的进程是否可以为达到提高性能的目的，绕过 NFS 协议直接访问文件。NFS 访问控制模型具有比大多数 UNIX 模型丰富得多的语义。这种差异来自于 NFS 应能够与 Windows 2000 系统互操作的需求。其基本思想是使 UNIX 的访问控制模型更易于适应 Windows 2000 的访问控制模型，以及反过来使 Windows 2000 的访问控制模型更易于适应 UNIX 的访问控制模型。

访问控制有别于诸如 UNIX 之类的文件系统的另一个方面是，可以为不同的用户和不同的组指定访问。传统上，对文件的访问是为单个用户（该文件的所有者）、单个用户组（比如，项目组的成员）和所有其他人指定的。NFS 具有很多不同类型的用户和进程，如图 10.20 所示。

用户类型	描述
owner	文件的所有者
group	与文件关联的用户所在的组
everyone	任何用户或进程
interactive	任何从交互式终端访问文件的进程
network	任何通过网络访问文件的进程
dialup	任何通过服务器的拨号连接访问文件的进程
batch	任何作为批处理作业的一部分访问文件的进程
anonymous	任何不需要身份验证而访问文件的用户
authenticated	任何需要身份验证的用户或进程
service	任何服务器定义的服务进程

图 10.20 NFS 所区分的关于访问控制的各种用户和进程

10.2 Coda 文件系统

下一个分布式文件系统的实例是 Coda。Coda 是卡内基-梅隆大学(CMU)1990 年开发的,目前,它被集成到许多流行的基于 UNIX 的操作系统,比如 Linux 之中。Coda 在很多方面不同于 NFS,尤其是其追求高可用性的目标。这一目标导致了高级缓存方案的出现,该方案允许客户端即使断开与服务器连接仍可以继续操作。(Satyanarayanan 等 1990,Kistler 和 Satyanarayanan 1992)概述了 Coda。关于系统的详细描述,请参阅(Kistler 1996)。

10.2.1 Coda 概述

Coda 被设计为一个可扩展的、安全的、高可用的分布式文件系统。它的一个重要目标是实现高度的命名和位置透明性,以使系统对用户来说好像是纯本地文件系统。另外,考虑到高可用性,Coda 的设计者也试图达到高度的故障透明性。

Coda 源于 AFS(Andrew file system,Andrew 文件系统)的第二版,并从 AFS 继承了许多体系结构特性,AFS 也是在 CMU 开发的(Howard 等 1988,Satyanarayanan 1999)。设计 AFS 是为了支持整个 CMU 社团,这意味着将近 10 000 台工作站需要访问系统。为了满足这一需求,AFS 节点被分为两个组。一个组由数量相对较少的专用 Vice 文件服务器组成,这些服务器是集中管理的。另一个组由大量的 Virtue 工作站组成,用户和进程使用这些工作站访问文件系统,如图 10.21 所示。

Coda 仍沿用 AFS 的组织结构。每个 Virtue 工作站拥有一个称为 Venus 的用户级进程,它的作用类似于 NFS 客户的作用。Venus 进程负责提供访问 Vice 文件服务器所维护的文件的机制。在 Coda 中,Venus 还负责使客户即使(暂时)不可能访问文件系统仍可以继续执行操作。这一额外功能是其与 NFS 所用方法的主要不同之处。

图 10.22 显示了 Virtue 工作站的内部体系结构。关键的问题是 Venus 是作为用户

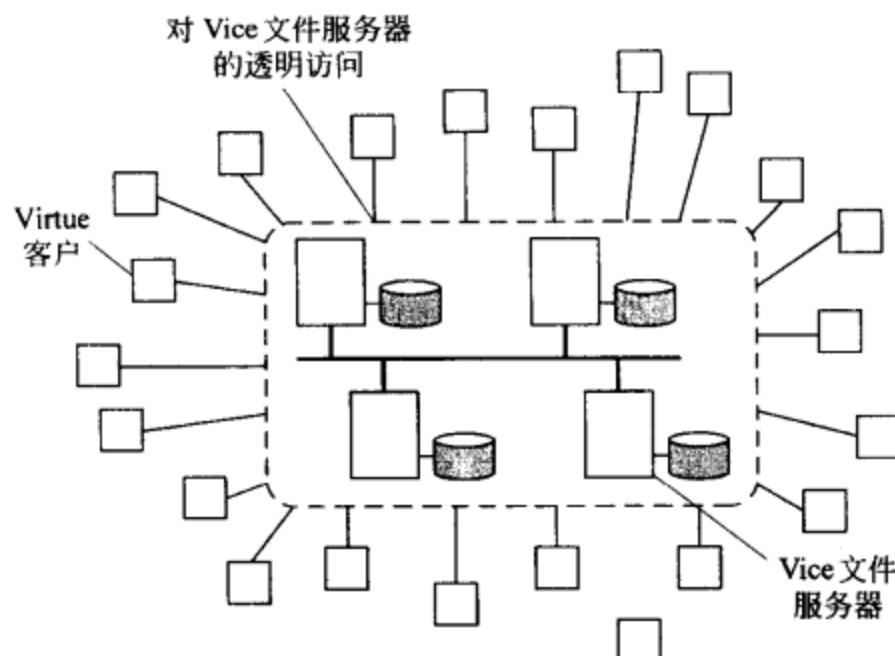


图 10.21 AFS 的总体组织结构

级进程运行的。同样，单独的 VFS(虚拟文件系统)层截获所有来自客户应用程序的调用，并将这些调用转发到本地文件系统或 Venus，如图 10.22 所示。这种使用 VFS 的组织结构与 NFS 中的组织结构相同。Venus 使用一个用户级 RPC 系统依次与 Vice 文件服务器通信。这个 RPC 文件系统构建于 UDP 数据报之上，并提供最多一次的语义。

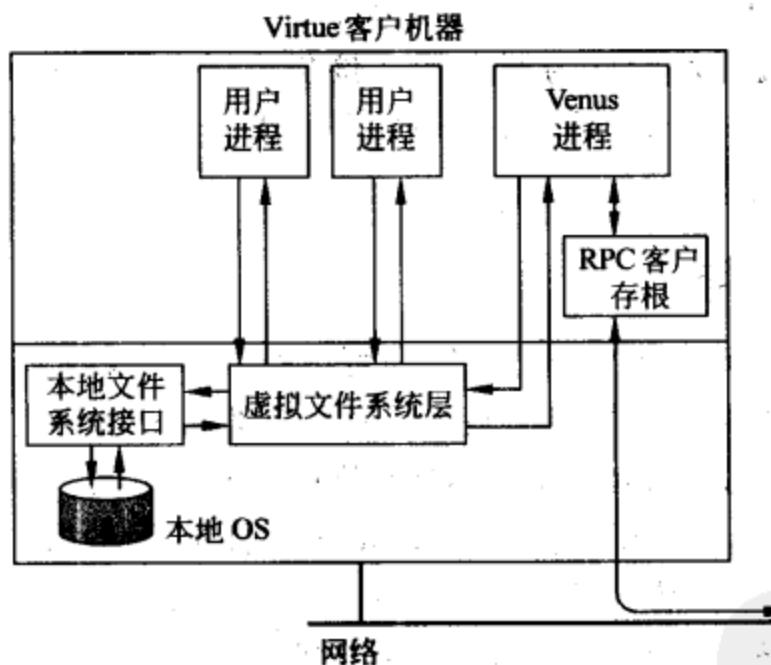


图 10.22 Virtue 工作站的内部组织结构

服务器端有三种不同的进程。绝大多数的工作是由真正的 Vice 文件服务器完成的，它负责维护本地文件。与 Venus 类似，文件服务器也是作为用户级进程运行的。另外，允许信任的 Vice 机器运行身份验证服务器，我们将在后面详细讨论身份验证服务器。最后，更新进程用于保持文件系统上的元信息 (meta information) 在每个 Vice 服务器上一致。

对于用户而言，Coda 就像一个传统的基于 UNIX 的文件系统。它支持构成部分

VFS 规范的大多数操作(Kleiman 1986),这些操作类似于图 10.3 列出的那些操作,因此,这里不再重复介绍这些操作。与 NFS 不同的是,Coda 提供了一个全局共享的名称空间,该名称空间由 Vice 服务器维护。客户通过它们本地名称空间中的特殊子目录,如/afs,访问这个名称空间。每当客户在这个子目录中搜索一个名称时,Venus 确保在本地装入这个共享的名称空间的适当部分。我们将在下面详细讨论这个问题。

10.2.2 通信

Coda 中的进程间通信是通过 RPC 实现的。但是,Coda 所用的 RPC2 系统比传统的 RPC 系统,如 NFS 所用的 ONC RPC 要复杂得多。

RPC2 在(不可靠的)UDP 协议之上提供可靠的 RPC。每当调用远程过程时,RPC2 客户节点启动一个新的线程,该线程向服务器发送一个调用请求,随后阻塞,直到它接收到服务器的响应。因为处理请求可能需要在任意时间完成,所以服务器有规律地向客户返回消息以通知客户它仍在处理请求。如果服务器死机,那么这个线程迟早会注意到这些消息停止了,并会向调用应用程序报告故障。

RPC2 的一个有趣特性是它对副作用的支持。副作用(side effect)是客户和服务器可以使用针对应用程序的协议进行通信的机制。比如,以客户打开视频服务器上的文件为例。本实例需要客户和服务器使用一种异步传输模式建立一个连续的数据流。也就是说,确保从服务器到客户的数据传输在最小的和最大的端到端延迟内完成,如第 2 章所述。

RPC2 允许客户和服务器建立单独的连接以按时将视频数据传输给客户。连接建立是作为一个服务器的 RPC 调用的副作用完成的。为此,RPC2 运行时系统提供副作用例程的接口,该例程由应用程序开发人员实现。比如,建立连接的例程和传输数据的例程。客户和服务器上的 RPC2 运行时系统分别自动地调用这些例程,但是它们的实现却完全独立于 RPC2。图 10.23 显示了副作用的这一原理。

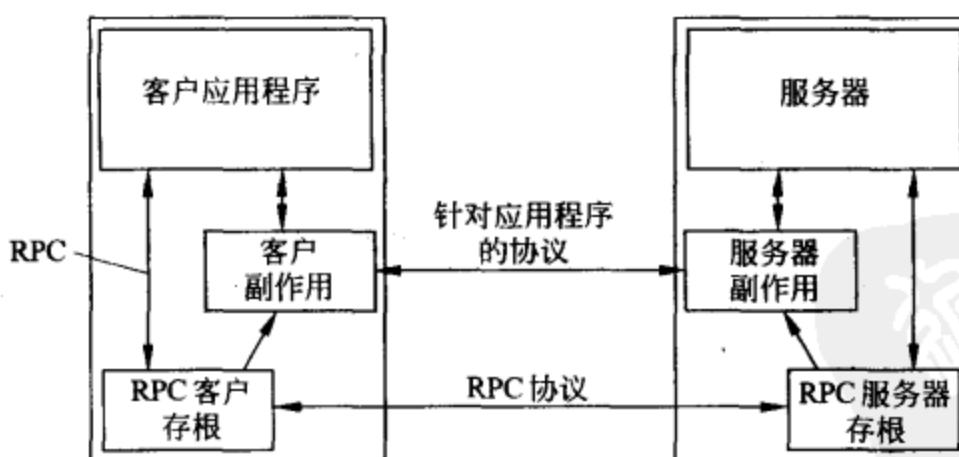


图 10.23 Coda 的 RPC2 系统中的副作用

RPC2 的另一个不同与其他 RPC 系统的特性是它支持多播。Coda 的一个重要设计问题是服务器需要记录哪些客户具有文件的本地拷贝,我们将在下面详细解释这一点。当文件被修改时,服务器通过 RPC 通知适当的客户,使这些文件的本地拷贝无效。显然,

如果服务器一次只能通知一个客户，那么使所有的客户无效可能需要一段时间，如图 10.24(a)所示。

这个问题是由 RPC 可能失败这样的事实引起的。按严格的顺序使文件无效可能被相当大程度地延迟，这是因为服务器不可能到达一个可能已崩溃的客户，但是服务器只有在经过相对较长的时间期限后才会放弃那个客户。同时，其他客户仍会从它们的本地拷贝读取数据。

图 10.24(b)显示了一个较好的解决方案。服务器不是一个接一个地使每个拷贝无效，而是并行地向所有客户发送一个无效化消息。因此，所有无故障的客户在与执行立即 RPC 的相同时间内得到通知。同样，服务器在通常的时间期限内注意到某些客户无法响应这个 RPC，从而能够宣布这样的客户已崩溃。

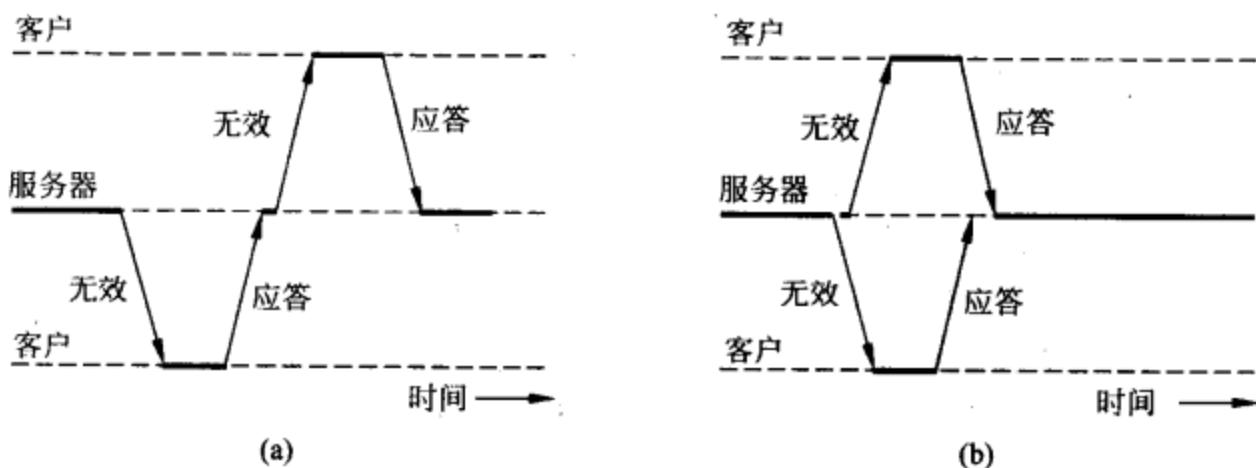


图 10.24 无效化消息的发送

(a) 一次发送一个无效化消息；(b) 并行地发送无效化消息

并行 RPC 是通过 MultiRPC 系统(Satyanarayanan 和 Siegel 1990)实现的，MultiRPC 系统是 RPC2 程序包的一部分。MultiRPC 的一种重要特性是 RPC 的并行调用对于被调用者而言是完全透明的。也就是说，MultiRPC 调用的接收者无法区分并行 RPC 和普通 RPC。对于调用方而言，并行执行也是相当透明的。比如，出现故障时的 MultiRPC 语义与普通 RPC 的语义非常相似。同样，副作用机制也可按照前面所述的方式使用。

MultiRPC 本质上是通过并行地执行多个 RPC 实现的。这意味着调用者显式地向每个接收者发送一个 RPC 请求。但是，调用者并不立即等待响应，而是在所有请求都被发送出去之后才阻塞。也就是说，调用者调用许多单向 RPC，然后阻塞，直到接收到所有无故障的接收者的响应为止。另一种在 MultiRPC 中并行执行 RPC 的方法是建立多播组，并使用 IP 多播向所有组成员发送 RPC。

10.2.3 进程

Coda 在客户进程和服务器进程之间保持了清晰的界限。客户由 Venus 进程表示；服务器以 Vice 进程的形式出现。两种类型的进程都被固有地组织为并发线程的集合。Coda 中的线程是非抢先的，并在整个用户空间操作。为了在阻塞 I/O 请求时连续操作，使用一个单独的线程处理所有 I/O 操作，它使用底层操作系统的低级异步 I/O 操作实现这一功能。该线程不阻塞整个进程，而有效地模拟同步 I/O 操作。

10.2.4 命名

如前面所述,Coda 拥有的命名系统类似于 UNIX 的命名系统。文件的分组单元称为卷(volume)。卷类似于 UNIX 的磁盘分区(比如,真正的文件系统),但是,它通常具有更小的粒度。它相当于 Vice 服务器维护的共享名称空间中的部分子树。通常,一个卷与一个用户关联的文件的集合相对应。卷的实例包括共享的二进制文件或源文件的集合,等等。与磁盘分区相同,卷也可以装入。

体现卷的重要性的原因有两个。第一,它们形成了构建整个名称空间的基本单元。名称空间是通过在挂接点上装入卷构建的。Coda 中的挂接点是卷的叶节点,该叶节点指向另一个卷的根节点。使用第 4 章引入的术语,只有根节点可以作为装入点(比如,客户只能装入卷的根目录)。卷之所以重要的第二个原因是它们形成了服务器端复制的单元。我们将在下面继续讨论卷的这个特性。

考虑到卷的粒度,我们希望名称搜索会跨越多个挂接点。也就是说,路径名通常包含多个挂接点。为了支持较高程度的命名透明性,Vice 文件服务器在名称搜索期间向 Venus 进程返回装入信息。这一信息使 Venus 能够在必要时自动地在客户的名称空间装入卷。这一机制类似于 NFS 版本 4 所支持的跨越挂接点进行搜索的机制。

值得注意的是,当来自共享名称空间的卷被装入到客户的名称空间时,Venus 遵循共享名称空间的结构。为了解释这一点,假设每个客户都能通过称为/afs 的子目录访问共享名称空间。当装入一个卷时,每个 Venus 进程确保以/afs 为根的命名图总是 Vice 服务器共同维护的完整名称空间的一个子图,如图 10.25 所示。这种方式向客户保证,共享文件确实具有相同的名称,尽管名称解析是根据本地实现的名称空间进行的。注意,这种方法从根本上不同于 NFS 所使用的方法。

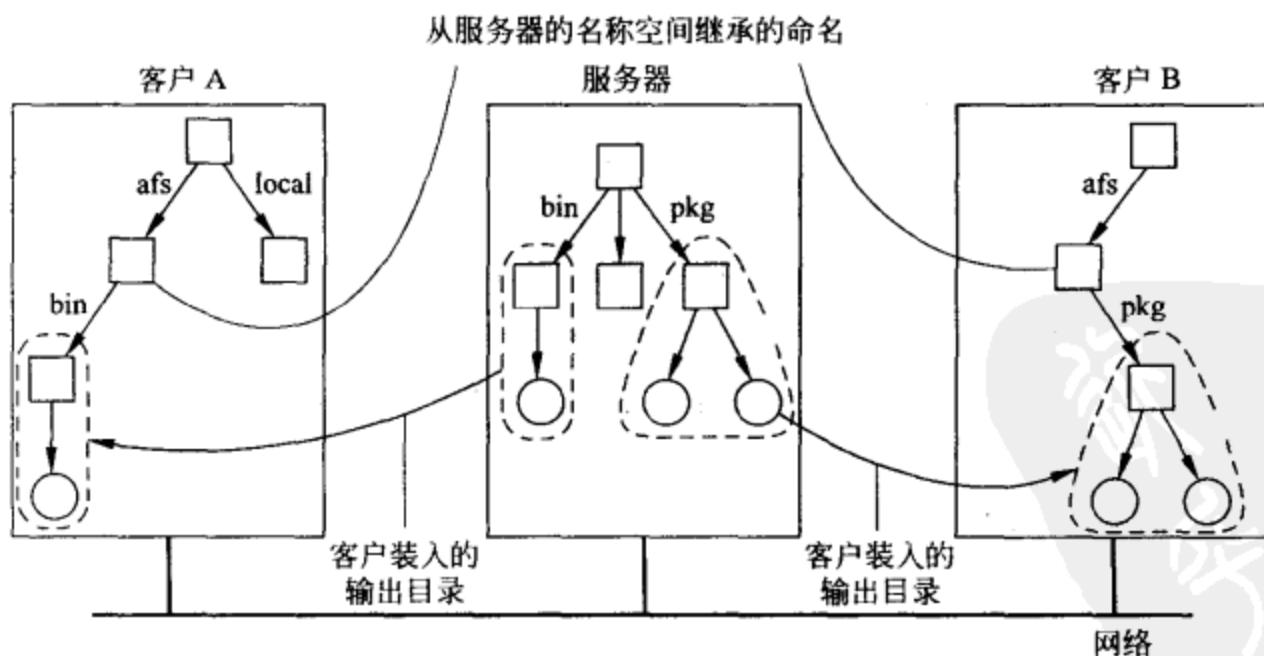


图 10.25 Coda 的客户访问单一的共享名称空间

1. 文件标识符

考虑到共享文件的集合可能被复制、分布于多台 Vice 服务器, 使用以下方法惟一地标识每个文件变得尤为重要。这种方法可以跟踪文件的物理位置, 而同时保持复制和位置的透明性。

Coda 中的每个文件包含于一个惟一的卷。如前所述, 一个卷可能被复制于多个服务器。为此, Coda 区分逻辑卷和物理卷。逻辑卷表示可能被复制的物理卷, 它具有一个关联的 RVID(replicated volume identifier, 复制卷)标识符。RVID 是一个与位置和复制无关的卷标识符。多个副本可以与相同的 RVID 关联。每个物理卷都有自己的 VID(volume identifier, 卷标识符), 该标识符以一种位置无关的方式标识一个特定的副本。

Coda 所使用的方法是为每个文件分配一个 96 位的文件标识符。一个文件标识符由两部分组成, 如图 10.26 所示。

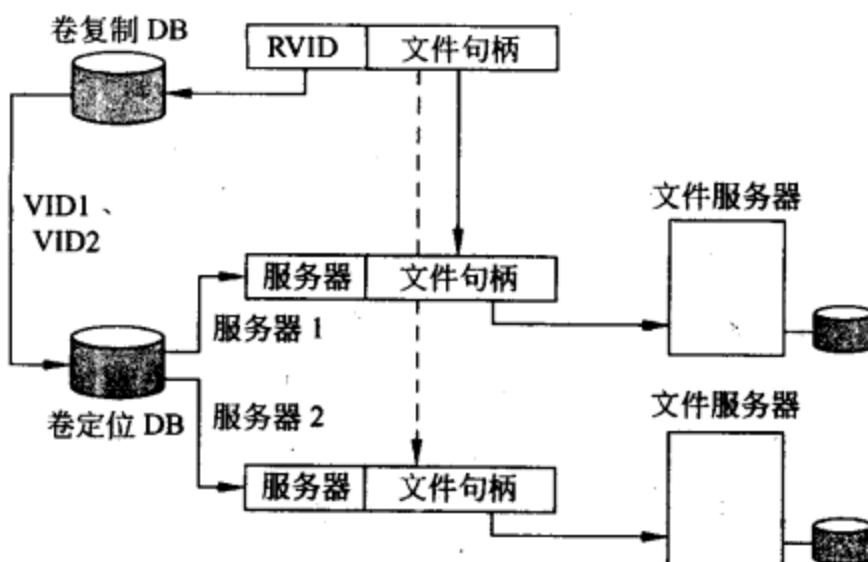


图 10.26 Coda 文件标识符的实现和解析

第一部分是文件所属的逻辑卷的 32 位 RVID。为了定位一个文件, 客户首先把文件标识符的 RVID 传送到卷复制数据库(volume replication database), 卷复制数据库返回与此 RVID 关联的 VID 的列表。然后, 对于给定的一个 VID, 客户可以搜索当前主管该逻辑卷的特定副本的服务器。这个搜索操作是通过把 VID 传送到卷位置数据库(volume location database)实现的, 卷位置数据库返回那个指定物理卷的当前位置。

文件标识符的第二部分由 64 位文件句柄组成, 文件句柄惟一地标识一个卷内的文件。实际上, 它相当于 VFS 所用的索引节点的标识。这样的 vnode, 就像其名称一样, 与 UNIX 系统中 inode 的概念相似。

10.2.5 同步

很多分布式文件系统, 包括 Coda 的祖先 AFS, 都不能提供 UNIX 的文件共享语义, 但是都能支持较弱的会话语义以替代文件共享语义。Coda 的目标是获得高可用性, 它采用了一种不同的方法, 并试图支持事务处理语义, 尽管它是一种比事务处理通常支持的语

义还弱的形式。

Coda 要解决的是大型分布式文件系统可能易于出现某些或全部文件服务器都暂时不可用的问题。这种不可用性可能是由网络或服务器故障导致的,但是它也可能是移动客户故意断开文件服务连接的结果。假如断开的客户在本地缓存了所有的相关文件,那么它应该可以在连接断开时使用这些文件,并在随后重新建立连接时使它们保持一致。

1. Coda 中的共享文件

为了提供文件共享,Coda 使用一种特殊的分配模式,该模式与 NFS 中的共享预约存在相似之处。下面的内容对于理解该模式的工作方式十分重要。当一个客户成功地打开一个文件 f 时, f 的完整拷贝都被传送到客户机器。服务器记录下客户有 f 的拷贝。到目前为止,这种方法与 NFS 中的打开委派相似。

现在假设客户 A 已为写操作打开文件 f 。当另一个客户 B 也想打开 f 时,它会失败。这是由于服务器已经有客户 A 可能已经修改 f 的记录引起的。另一方面,如果客户 A 为读操作打开 f ,那么客户 B 试图从服务器获得一份拷贝以执行读操作将会成功。B 试图为写操作打开文件也将会成功。

现在考虑一下 f 的多个拷贝已经存储于多个客户本地的情况。假定像我们刚刚所说的那样,只有一个客户能够修改 f 。如果这个客户修改 f ,随后关闭该文件,那么文件将被传回服务器。但是,其他各个客户可能继续读取它们的本地拷贝,尽管实际上它们的拷贝是过时的。

导致这种明显不一致的行为的原因在于 Coda 将会话视为事务处理。以图 10.27 为例,它显示了两个进程 A 和 B 的时间线。假设 A 已为读操作打开 f ,它导致会话 S_A 。客户 B 已为写操作打开 f ,如图所示的 S_B 。

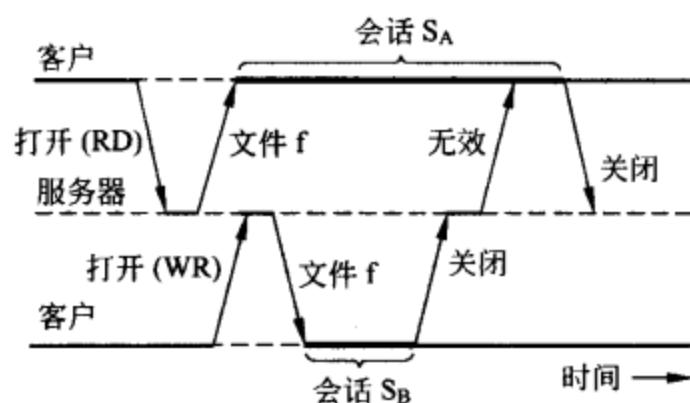


图 10.27 Coda 中共享文件的事务处理行为

当 B 关闭会话 S_B 时,它向服务器传送更新版本的 f ,然后服务器将向 A 发送一个无效化消息。此时,A 知道它正在读取较老版本的 f 。但是,从事务处理的角度来说,这真的无关紧要,因为可以认为会话 S_A 已经安排在会话 S_B 之前。

2. 事务处理语义

在 Coda 中,网络分区的概念在定义事务处理语义时起到至关重要的作用。分区

(partition)是网络的一部分,它与网络的其余部分相隔离,并且它由一些客户或服务器组成,或由二者共同组成。其基本思想是一系列的文件操作应该在出现跨不同分区的冲突操作时继续执行。回忆一下,如果两个操作都对相同的数据进行操作,并且至少其中一个操作是写操作,那么称这两个操作是冲突的。

我们首先研究一下在网络分区的情况下冲突是如何发生的。假设两个进程 A 和 B 持有多个共享数据项的完全相同的副本,恰好此时,由于网络分区造成它们分离。在理想情况下,支持事务处理语义的文件系统会实现单拷贝串行性(one-copy serializability),也就是说,操作分别被 A 和 B 执行的结果等同于它们联合串行地在这两个进程共享的、非复制的数据项上执行那些操作的结果。(Davidson 等 1985)深入讨论了这个概念。

我们已经在第 5 章中介绍了串行性的实例。使用分区时的主要问题是在一个分区内发生过串行化的执行顺序后,识别这个串行化的执行顺序。也就是说,当文件系统从一次网络分区恢复时,它所面对的是许多已经在每个分区上(可能是在浅拷贝(shadow copy)上,也就是说,发送到客户端执行试验性修改的文件的拷贝,类似于事务处理中的影像块(shadow block))执行过的事务处理。那么,为了接受这些事务处理,它需要检查联合执行是否可以串行化。通常,这是一个难以处理的问题。

Coda 采用的方法是把会话解释为事务处理。典型的会话以显式地调用 open 打开一个文件开始。通常,此后会出现一系列的 read 操作和 write 操作,然后,该会话以调用 close 关闭该文件而结束。大多数的 UNIX 系统调用自身构成一个单一的会话,Coda 也将这些系统调用看作独立的事务处理。

Coda 可识别不同的会话类型。比如,每个 UNIX 系统调用都与一个不同的会话类型关联。较复杂的会话类型是那些以调用 open 开始的会话类型。会话的类型是自动地由应用程序调用的系统调用推断出来的。这种方法的主要优点是使用标准化的 VFS 接口的应用程序不需要进行修改。对于每种会话类型,将读取或修改哪些数据都是事先已知的。

以 store 会话类型为例,该会话以打开文件的操作开始,如上所述,它代表指定的用户 u 打开文件 f 以执行写操作。图 10.28 列出了与文件 f 和用户 u 关联的、受该会话类型影响的元数据,以及这些数据是否只被读取,是否可被修改。比如,可能有必要读取用户 u 对文件 f 的访问权限。通过显式地标识指定会话所读取和修改的元数据,冲突操作的识别就变得更加容易。

与文件关联的数据	读取?	修改?
文件标识符	是	否
访问权限	是	否
最后修改时间	是	是
文件长度	是	是
文件内容	是	是

图 10.28 在 Coda 中,store 会话类型所读取和修改的元数据

从这一点看来,会话处理是相当直截了当的。我们先考虑一下发生在单一网络分区内的一系列并发会话。为了简化问题,我们假设该分区只包含一台服务器。当客户开始一个会话时,假如不违背上述的文件共享的规则,那么客户机器上的 Venus 进程将从服务器获取该会话的读集合和写集合包含的所有数据。这样,它有效地为那些数据获得了必要的读操作锁和写操作锁。

这里,我们可以看出两个重要问题。首先,因为 Coda 可以自动地由应用程序调用的(第一个)系统调用推断出会话的类型,而且它知道每种会话类型所读取和修改的元数据,所以 Venus 进程知道会话开始时应从服务器获取哪些数据。因此,它可以在会话开始时就获得必要的锁。

其次,因为文件共享语义与预先获得必要的锁非常相似,所以这里采用的方法与第 5 章所述的应用两阶段锁定(2PL)技术的方法类似。2PL 的重要特性是它使并发会话的读操作和写操作的所有调度都是可串行化的。

现在,我们考虑一下面对分区时如何处理会话。需要解决的主要问题是需要解决跨分区的冲突。为此,Coda 使用一种简单的版本记录模式。每个文件有一个关联的版本号,该版本号指出文件创建以来更新的次数。如前所述,当客户开始一个会话时,所有与那个会话相关的数据,包括每个数据元素关联的版本号,都被复制到客户机器上。

假设当客户正在执行一个或多个会话时,网络分区使客户和服务器间的连接断开。此时,Venus 允许客户继续执行完它的会话,就好像什么都没发生,这符合上述单一分区情况下的事务处理语义。随后,再次建立与服务器的连接时,更新以它们在客户上发生的顺序传送到服务器。

当对文件 f 的更新传送到服务器时,如果在客户和服务器断开连接的期间没有其他进程更新 f ,那么服务器会试探性地接受文件的更新。比较版本号可以很容易地检测到这种冲突。设 V_{client} 是文件 f 传送到客户时从服务器获得 f 的版本号。设 $N_{updates}$ 是在该会话期间服务器重新合成之后传送和接受的更新的个数。最后,设 V_{now} 表示服务器上 f 的当前版本号。那么,当且仅当

$$V_{now} + 1 = V_{client} + N_{updates}$$

来自客户会话的 f 的下一个更新可以被接受。也就是说,仅当来自客户的更新会导致文件 f 的下一版本时,它才会被接受。实际上,这意味着最终只有一个客户获胜,从而解决了冲突。

如果并发执行的会话同时更新 f ,那么就会出现冲突。发生冲突时,无法完成来自客户会话的更新,实际上,客户被迫为手工调整保存 f 的本地版本。就事务处理而言,会话无法提交,并且冲突由用户解决。我们将在下面继续讨论这个问题。

10.2.6 缓存和复制

至此,我们可以清楚地看出,缓存和复制在 Coda 中起到重要的作用。事实上,这两种方法是达到 Coda 设计人员所制定的高可用性目标的基本方法。下面,我们先介绍客户端缓存,它是处理连接断开时操作的至关重要的方法。然后,我们再介绍服务器端的卷复制。

1. 客户缓存

客户端缓存之所以对 Coda 的操作至关重要是因为以下两个原因。首先,它符合 AFS 采用的方法(Satyanarayanan 1992),进行缓存是为达到可扩展性的目的。其次,缓存提供了较高程度的容错性,这是因为客户变得较少地依赖于服务器的可用性。出于这两个原因,Coda 的客户总是高速缓存整个文件。也就是说,不管文件是为读操作还是为写操作而打开的,整个文件的拷贝被传送给客户,然后客户缓存这份拷贝。

与许多其他分布式文件系统不同,Coda 的缓存相关性是通过回叫的方法维护的。对于每个文件,客户从服务器获取文件时,服务器记录哪些客户在本地缓存了该文件的拷贝。此时称服务器为客户记录回叫承诺(callback promise)。当客户第一次更新该文件的本地拷贝时,它通知服务器,然后,服务器依次向其他客户发送无效化消息。这种无效化消息称为回叫中断(callback break),这是因为服务器将会废弃它为刚刚向其发送了无效化消息的客户保存的回叫承诺。

这个方法的有趣之处在于只要客户知道它在服务器上有未被废弃的回叫承诺,它就可以安全地在本地访问文件。具体地说,假设客户打开一个文件,它发现该文件已经在它的高速缓存中。那么,如果服务器仍为这个客户保留着该文件的回叫承诺,那么这个客户就可以使用该文件。客户必须与服务器核对那个回叫承诺是否还在。如果它还在,服务器就不需要再向客户传送文件了。

这种方法如图 10.29 所示,它扩展了图 10.27。当客户 A 开始会话 S_A 时,服务器记录一个回叫承诺。同样,B 开始会话 S_B 时也是如此。但是,当 B 关闭 S_B 时,服务器向客户 A 发送回叫中断,从而中断了它对回叫客户 A 的承诺。注意,由于 Coda 的事务处理语义,客户 A 关闭会话 S_A 时,不会发生任何特殊的事情,这个关闭操作就像所预期的情况那样被简单地接受。

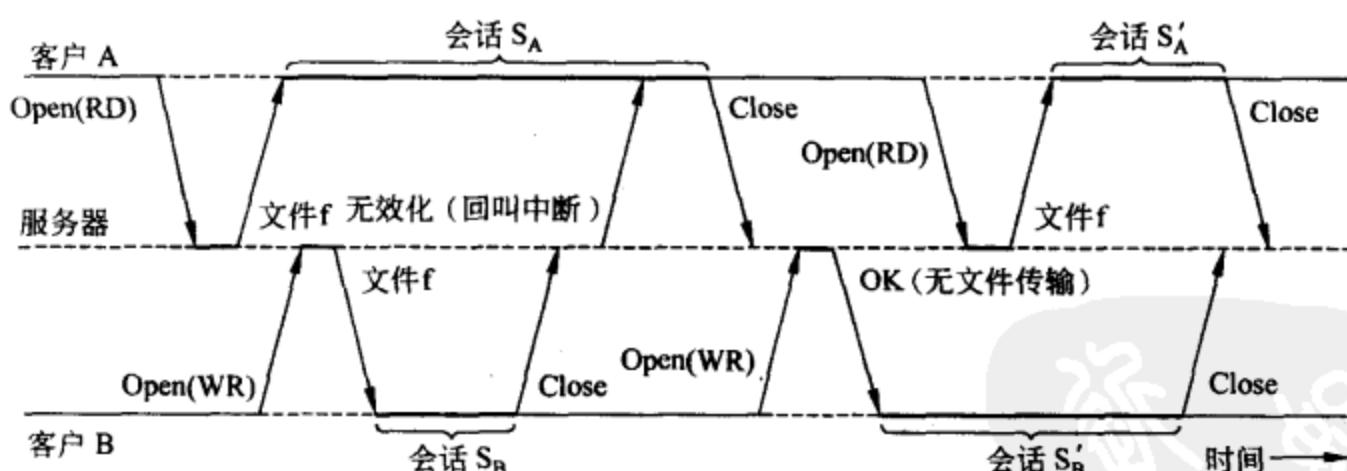


图 10.29 在 Coda 中打开会话时本地拷贝的使用

因此,当 A 随后要打开会话 S'_A 时,它会发现其本地的 f 的拷贝是无效的,所以它必须从服务器获取最新版本的文件拷贝。另一方面,当 B 开始会话 S'_B 时,它会注意到服务器仍有一个未被废弃的回叫承诺,该承诺意味着 B 可简单地重新使用从会话 S_B 获得的本地拷贝。

2. 服务器复制

Coda 允许复制文件服务器。如前所述,复制的单位是卷。拥有一份卷拷贝的服务器的集合称为该卷的卷存储组(volume storage group),或简单地称为 VSG。出现故障时,客户可能不能访问卷的 VSG 中的所有服务器。对于一个卷,客户的 AVSG(accessible volume storage group,可访问卷存储组)是由卷的 VSG 中客户可以联系的那些服务器组成的。如果 AVSG 是空的,那么称该客户是断开的。

Coda 使用复制的写协议来维护复制卷的一致性。具体地说,它使用第 6 章介绍的 ROWA(Read-One, Write-All)的一种变体形式。客户需要读取一个文件时,该客户联系该文件所属卷的 AVSG 中的一个成员。但是,在关闭一个更新的文件上的会话时,客户将这个更新的文件并行地传送到 AVSG 的每个成员。这种并行传送是通过上述的为 RPC 实现的。

在没有故障的情况下,即在对于每个客户来说,客户卷的 AVSG 与该卷的 VSG 相同的情况下,这种方案工作得很好。但是,在出现故障时,情况可能变得很糟。以一个复制于三个服务器 S_1 、 S_2 和 S_3 上的卷为例。对于客户 A,假设它的 AVSG 包括服务器 S_1 和 S_2 ,而客户 B 只有权访问服务器 S_3 ,如图 10.30 所示。

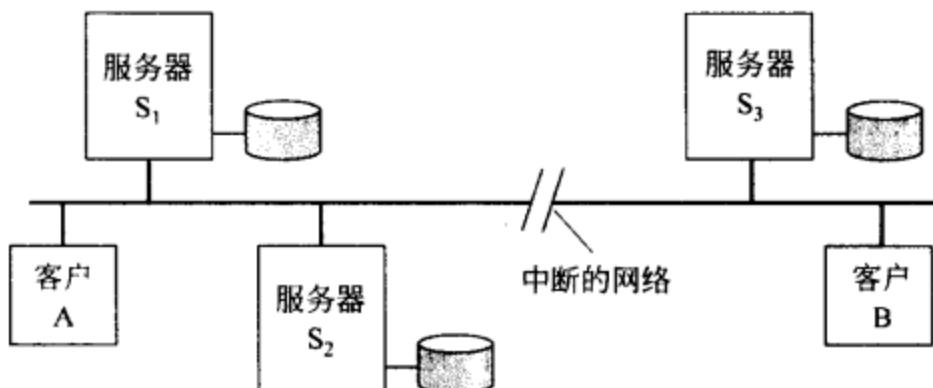


图 10.30 对于同一复制文件,具有不同 AVSG 的两个客户

Coda 使用一种优化策略实现文件复制。具体地说,A 和 B 都被允许为执行写操作打开文件 f ,更新它们各自的拷贝,并将它们的拷贝传回它们的 AVSG 中的成员。显然,在 VSG 中将存在不同版本的 f 。问题是如何检测和解决这种不一致性。

Coda 采用的方法是扩展上一节讨论的版本记录方案。具体地说,VSG 中的服务器 S_i 为该 VSG 包含的每个文件 f 维护一个 Coda 版本向量(coda version vector)($CVV_i(f)$)。如果 $CVV_i(f)[j] = k$,那么服务器 S_i 就知道服务器 S_j 至少已经看到文件 f 的版本 k 。 $CVV_i[i]$ 是存储在服务器 S_i 上的 f 的当前版本号。服务器 S_i 对 f 的更新会导致 $CVV_i[i]$ 增加。注意,版本向量完全类似于第 5 章讨论的向量时间戳。

回到刚才讨论的三个服务器的实例,对于每个服务器 S_i , $CVV_i(f)$ 的初始值是 $[1, 1, 1]$ 。当客户 A 从其 AVSG 中的一个服务器,假设 S_1 ,读取 f 时,它也收到 $CVV_1(f)$ 。更新 f 后,客户 A 向其 AVSG 中的每个服务器,即 S_1 和 S_2 多播 f 。然后,这两个服务器都会记录它们各自的拷贝已被更新,但是 S_3 的拷贝没有被更新。也就是说:

$$CVV_1(f) = CVV_2(f) = [2, 2, 1]$$

此时,客户 B 被允许打开一个会话,在该会话中,它从服务器 S_3 接收一份 f 的拷贝,并随后更新 f 。当它关闭这个会话,并向 S_3 传送更新时,服务器 S_3 会把它的版本向量更新为 $CVV_3(f)[1, 1, 2]$ 。

分区结束时,这三个服务器需要使用它们的拷贝重新合成 f 。通过比较它们的版本向量,它们就会发现出现了需要进行修复的冲突。在很多情况下,可以使用应用程序相关的方法自动地解决冲突,有关内容在文献(Kumar 和 Satyanarayanan 1995)中有所讨论。但是,很多情况下,用户不得不参与手工解决冲突,特别是在不同用户以不同的方式修改了同一文件的相同部分的情况下。

10.2.7 容错性

Coda 是为达到高可用性的目标而设计的,这一点主要体现在它对客户端缓存的高度支持和对服务器复制的支持。我们已经在前面讨论过这两个方面。Coda 的另一个需要深入讨论的有趣特性是客户如何在连接断开时继续操作,甚至在连接断开持续几小时或几天的情况下继续操作。

1. 断开连接的操作

如上所述,如果对于一个卷,客户的 AVSG 是空的,那么称该客户是与卷断开的。也就是说,客户不能联系任何持有该卷拷贝的服务器。在大多数文件系统(比如,NFS)中,客户将不能继续执行,除非它可以联系至少一个服务器。Coda 采用一种不同的方法。此方法中,客户将简单地使用文件的本地拷贝,这个本地拷贝是该客户在服务器上打开该文件时获得的。

在连接断开时,关闭文件(或实际上,关闭访问文件的会话)总会成功。但是,再次建立连接并把修改内容传回服务器时,可能会检测到冲突。如果自动的冲突解决失败,那么手工干预是十分必要的。使用 Coda 的实践经验表明,断开连接的操作通常可以正常工作,尽管存在由于不可解决的冲突而导致重新合成失败的情况。

Coda 所采用方法的成功主要归因于以下事实:在实践中,几乎不发生写操作共享文件的情况。也就是说,实际中,两个进程打开相同的文件以执行写操作是很罕见的。当然,只为执行读操作而共享文件是经常发生的,但是这样不会引起任何冲突。这些观察结果也可从其他文件系统得到(比如,请参看(Page 等 1998)中有关高度分布式文件系统中的冲突解决方面的内容)。此外,Coda 的文件共享模型底层的事务处理语义也使处理多个进程只读取一个共享文件,而同时只有一个进程并发地修改这个文件的情况更容易。

为了成功地完成断开连接的操作,需要解决的主要问题是确保客户缓存包含连接断开期间将访问的那些文件。如果采用简单的缓存方法,那么可以证明客户可能由于缺少必要的文件而不能继续执行。预先使用适当的文件填充高速缓存称为储藏(hoarding)。客户对于一个卷(及该卷上的文件)的全部行为可以用图 10.31 所示的状态转换图概括。

通常,客户处于 HOARDING 状态。在该状态时,客户连接到(至少)一个包含有卷

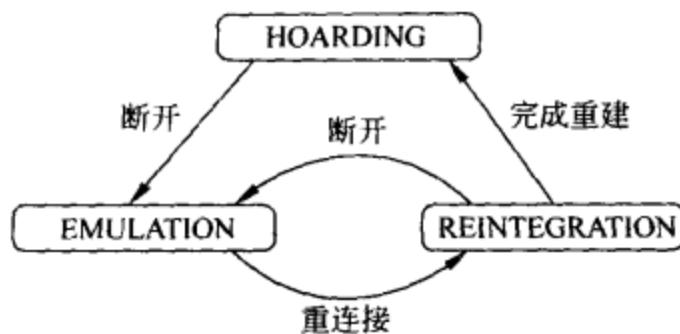


图 10.31 对于一个卷,Coda 客户的状态转换图

拷贝的服务器。客户处于该状态时,它可以联系服务器,提出文件请求以完成它的工作。同时,它也会试图用有用的数据(比如,文件、文件属性和目录)填充它的高速缓存。

在某一时刻,客户的 AVSG 中的服务器的个数降为 0,这就使客户进入 EMULATION 状态,此时,客户机器将不得不模拟服务器对卷的行为。实际上,这意味着使用本地缓存的文件拷贝直接处理所有文件请求。注意,当客户处于 EMULATION 状态时,它可能仍能联系管理其他卷的服务器。这种情况下,连接断开通常是由于服务器故障造成的,而不是由于客户的网络连接断开造成的。

最后,重新建立连接后,客户进入 REINTEGRATION 状态,此时,客户把更新传递到服务器以使这些更新永久保存在服务器上。冲突检测是在重新合成期间进行的,如果可能,是在重新合成期间自动解决的。如图 10.31 所示,在重新合成期间,与服务器的连接可能再次断开,这会使客户回到 EMULATION 状态。

对于连接断开时连续操作的成功,高速缓存包含所有必要的数据是至关重要的。Coda 使用一种复杂的优先级机制来确保有用的数据确实已经缓存。首先,用户可以在储藏数据库(hoard database)中存储路径名,显式地声明那些重要的文件或目录。Coda 为每个工作站维护这个数据库。储藏数据库中的信息和最近的文件引用信息结合起来,使 Coda 能够计算出每个文件的当前优先级,然后,它按照满足以下三个条件的优先级获取文件:

- (1) 未缓存的文件的优先级不比任何已缓存的文件的优先级高。
- (2) 高速缓存已满,或者没有任何未缓存的文件具有非零的优先级。
- (3) 每个缓存的文件都是客户 AVSG 中维护的文件的拷贝。

计算文件的当前优先级的详细方法在文献(Kistler 1996)中有所描述。如果满足所有三个条件,那么称该高速缓存处于平衡状态(equilibrium)。因为文件的当前优先级会随时间改变,而且可能需要从高速缓存删除缓存的文件从而为其他文件腾出空间,所以需要时常重新计算高速缓存的平衡。重新组织高速缓存以使其达到平衡是由称为 hoard walk(储藏走查)的操作完成的,该操作每 10 分钟调用一次。

传统的高速缓存管理技术通常基于计数和计时引用,储藏数据库、优先级函数和保持高速缓存的平衡的结合是对传统高速缓存管理技术的巨大改进。但是,这种技术不能保证客户缓存总是包含用户近期需要的数据。因此,仍然存在某些情况,连接断开的模式中的操作会因为不可访问数据而失败。

2. 可恢复的虚拟内存

除了提供高可用性之外,AFS 和 Coda 的设计人员还考虑了一些有助于建立容错过程的简单机制。RVM(recoverable virtual memory, 可恢复的虚拟内存)是一个简单、有效的机制,它使恢复变得十分容易。RVM 是一个用户级的机制,它将关键数据结构保存在内存中,但它可以保证崩溃故障后很容易地恢复这些关键数据结构。关于 RVM 的详细内容,请参看(Satyanarayanan 等 1994)。

RVM 底层的基本思想是相当简单的:崩溃故障后应继续存在的数据通常被存储在文件中,需要该文件时,它被显式地映射到内存。对那些数据的操作记入日志,这类似于事务处理所用的先写日志。实际上,RVM 所支持的模型接近于平坦(flat)事务处理的模型,只是它不支持并发控制。

一旦文件映射到内存,应用程序就可以在那些数据上执行事务处理中的操作。RVM 并不知道数据结构。因此,事务处理中的数据被应用程序显式地设置为映射文件的一段连续字节。那些数据上的所有(内存中的)操作都被记录在一个单独的先写日志中,该日志需要保存在稳定存储中。注意,因为日志通常相对较小,所以使用电池供电的内存部分是可行的,这样就使持久性和高性能结合在一起。

10.2.8 安全性

Coda 继承了 AFS 的安全性体系结构,它由两个部分组成。第一部分使用安全 RPC 和系统级的身份验证建立客户和服务器之间的安全通道。第二部分处理文件的访问控制。我们将依次研究它们。

1. 安全通道

Coda 使用秘密密钥加密系统建立客户和服务器之间的安全通道。Coda 采用的协议是从第 8 章讨论的 Needham-Schroeder 身份验证协议派生而来的。简单地说,用户需要先从 AS(身份验证服务器)获得特殊的令牌。这些令牌与 Needham-Schroeder 协议中发放的票据随后都被用于建立到达服务器的安全通道,从这种意义上说,它们有些相似。

客户和服务器之间的所有通信都基于 Coda 的安全 RPC 机制,如图 10.32 所示。如果 Alice(作为客户)想与 Bob(作为服务器)交谈,她向 Bob 发送她的身份,及使用 Alice 和 Bob 共享的秘密密钥 $K_{A,B}$ 加密的质询 R_A 。这次通信被表示为消息 1。

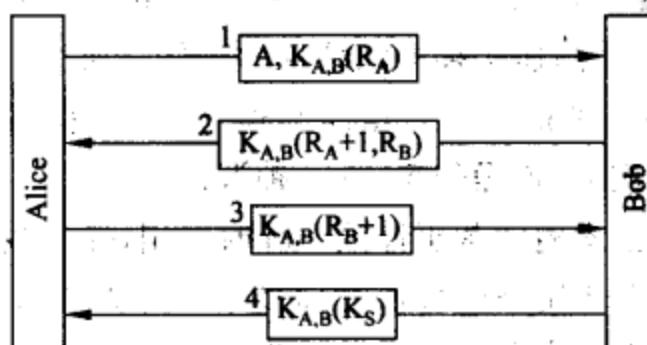


图 10.32 RPC2 中的相互身份验证

Bob 通过解密消息 1 并返回 $R_A + 1$ 来进行回应, 这证明他知道秘密密钥, 从而证明他是 Bob。他返回质询 R_B (作为消息 2 的一部分), Alice 必须解密该质询并且也返回 $R_B + 1$ (表示为消息 3)。进行相互身份验证时, Bob 生成一个会话密钥 K_S , Alice 和 Bob 之间的进一步通信可使用该密钥。

Coda 中的安全 RPC 只用于建立客户和服务器之间的安全连接, 它不足以建立可能包含多个服务器并可能持续相当长时间的安全登录会话。因此, 需要使用安全 RPC 层之上的另一个协议。前文已经简要地提及, 客户通过该协议从 AS 获得身份验证令牌。

身份验证令牌有些类似于 Kerberos 中的票据。它包括获得该令牌的进程的标识符、令牌标识符、会话密钥, 以及表明令牌有效时间和过期时间的时间戳。为保证完整性, 可以用密码封存令牌。比如, 如果 T 是一个令牌, K_{vice} 是所有 Vice 服务器共享的秘密密钥, H 是一个散列函数, 那么 $[T, H(K_{vice}, T)]$ 是 T 被密码封存的版本。也就是说, 尽管 T 是以明文的形式在非安全的通道上发送的, 但是入侵者不可能在 Vice 服务器毫无察觉的情况下修改它。

当 Alice 登录到 Coda 时, 她需要先从 AS 获得身份验证令牌。此时, 她使用她的密码执行一个安全 RPC 以生成秘密密钥 $K_{A,AS}$, 她与 AS 共享的这个秘密密钥 $K_{A,AS}, K_{A,AS}$ 用于实现上述的相互身份验证。AS 返回两个身份验证令牌。明文令牌 $CT = [A, TID, K_S, T_{start}, T_{end}]$ 验证 Alice 的身份, 它包括令牌标识符 TID、会话密钥 K_S , 以及指定令牌有效期的两个时间戳 T_{start} 和 T_{end} 。另外, AS 还发送密文令牌 $ST = K_{vice}([CT]_{K_{vice}}^*)$, 该令牌是使用所有 Vice 服务器共享的秘密密钥 K_{vice} 加密的 CT, 解密该令牌也使用相同的密钥。

Vice 服务器能够解密 ST, 获得 CT, 从而获得会话密钥 K_S 。同样, 因为只有 Vice 服务器知道 K_{vice} , 所以这个服务器可以通过执行完整性检查(计算完整性需要 K_{vice}), 容易地检测出 CT 是否被篡改过。

每当 Alice 想要与 Vice 服务器建立安全通道时, 她就使用密文令牌 ST 来验证她自己的身份, 如图 10.33 所示。AS 使用明文令牌向 Alice 发送会话密钥 K_S, K_S 用于加密她向服务器发送的质询 R_A 。

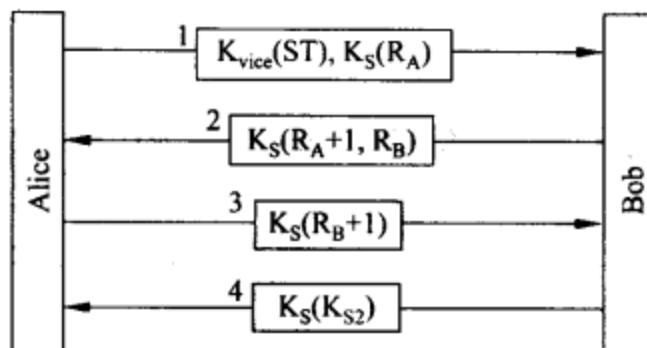


图 10.33 建立 Coda 的(Venus)客户和 Vice 服务器之间的安全通道

服务器首先使用共享的秘密密钥 K_{vice} 解密 ST, 得到 CT。然后, 它找到 K_S , 而后它用 K_S 来完成身份验证协议。当然, 服务器也对 CT 执行完整性检查, 只有在令牌当前有效时才继续执行。

产生问题的情况是: 客户访问文件前需要进行身份验证, 而此时该客户的连接是断

开的。因为客户不能联系服务器,所以身份验证无法执行。既然这样,将推迟身份验证推迟,并暂时允许文件访问。当客户重新连接上服务器时,它先进行身份验证,再进入REINTEGRATION状态。

2. 访问控制

我们简要地讨论一下 Coda 中的保护问题。与 AFS 类似,Coda 使用访问控制列表来确保只有被授权的进程才能访问文件。出于简单性和可扩展性的原因,Vice 文件服务器只将目录与访问控制列表关联,而不将文件与访问控制列表关联。同一目录下的所有普通文件(即,不包括子目录)共享相同的保护权。

Coda 可以区分图 10.34 所示的操作类型对应的访问权限。注意,没有权限与执行文件的操作相对应。省略该权限的原因很简单:文件的执行发生在客户上,因此,它超出了 Vice 文件服务器所管理的范围。一旦客户下载了文件,Vice 就无法分辨客户是执行文件还是只读取文件的内容。

操作	描述
read	读取目录中的任一文件
write	修改目录中的任一文件
lookup	搜索任一文件的状态
insert	向目录增加一个新文件
delete	删除一个已存在的文件
administer	修改目录的 ACL

图 10.34 Coda 所识别的关于访问控制的文件、目录操作的分类

Coda 维护用户和组的信息。除了指定用户和组拥有的权限之外,Coda 也支持指定负权限。也就是说,它可以显式地声明指定的用户不拥有某些访问权限。事实表明,这种方法相当方便,因为它不需要先从所有组中删除有不当行为的用户,就可以立即撤销该用户的访问权限。

10.3 其他分布式文件系统

除了上面讨论的分布式文件系统外,还有很多其他分布式文件系统,尽管其中的大多数系统与 NFS 或 Coda 存在很多相似之处。在本节中,我们简要地讨论三个系统,这三个系统在某一个或多个方面有所不同。我们先讨论 Plan 9,该系统将每种资源视为文件。XFS 是无服务器文件系统的例子。最后一个实例是 SFS,在该文件系统中,文件名也包含安全性信息。

与 NFS 和 Coda 的描述方法相反,我们故意省略本节所讨论的三个系统的许多细节问题。而将注意力主要集中于每个系统底层的一般原理,以提出实现分布式文件系统的可选择的方法。

10.3.1 Plan 9：资源统一为文件

从 20 世纪 80 年代开始，一直存在着用强大的工作站网络代替集中式分时系统的趋势，这一趋势导致了网络操作系统的出现，如第 1 章所述。这种方法的一个问题是透明性通常会丢失，而这一点又导致了诸如分布式（操作）系统的发展。

开发 Plan 9 是对网络操作系统的反应，它重新引入具有少数集中式的服务器和许多客户机器的系统理念。但是，它不用大型机或小型机作为服务器，而假定服务器是强大的、相对便宜的计算机，这些计算机使用微机技术。服务器仍像从前一样集中管理。另一方面，假定客户机器是简单的，并且只有少数任务。

该系统主要由贝尔实验室负责开发 UNIX 的人员设计。该项目起始于 20 世纪 80 年代晚期。考虑到许多现代的局部分布式系统都与相对简单的客户和一定数量的强大的服务器组织在一起，这种表面上过时的思想，即使用集中式管理的分时系统的思想，当今仍是有效的。（Pike 等 1995）给出了 Plan 9 的概述。关于详细的信息，可参看 Plan 9 程序员手册（Bell Labs 2000）。

与其说 Plan 9 是一个分布式文件系统，不如说它是一个基于文件的分布式系统（file-based distributed system）。所有资源，甚至诸如进程和网络接口之类的资源，都是以相同的方式访问的，即使用类似于文件的语法和操作访问。这种思想是从 UNIX 继承而来的，UNIX 也试图为各种资源提供类似于文件的接口，但是 Plan 9 更深入、更一致地利用了类似于文件的接口。每个服务器为它所控制的资源提供层次式的名称空间。客户可以在本地装入服务器提供的名称空间，从而建立它自己的私有名称空间，这种方法类似于 NFS 所采用的方法。为了允许共享，部分名称空间是标准化的。

这种方法产生了图 10.35 所示的组织结构。Plan 9 系统由一些服务器组成，这些服务器以本地名称空间的形式向客户提供资源。为了访问服务器的资源，客户要将服务器的名称空间装入它们自己的名称空间。尽管我们区分客户和服务器，但我们应当注意到，

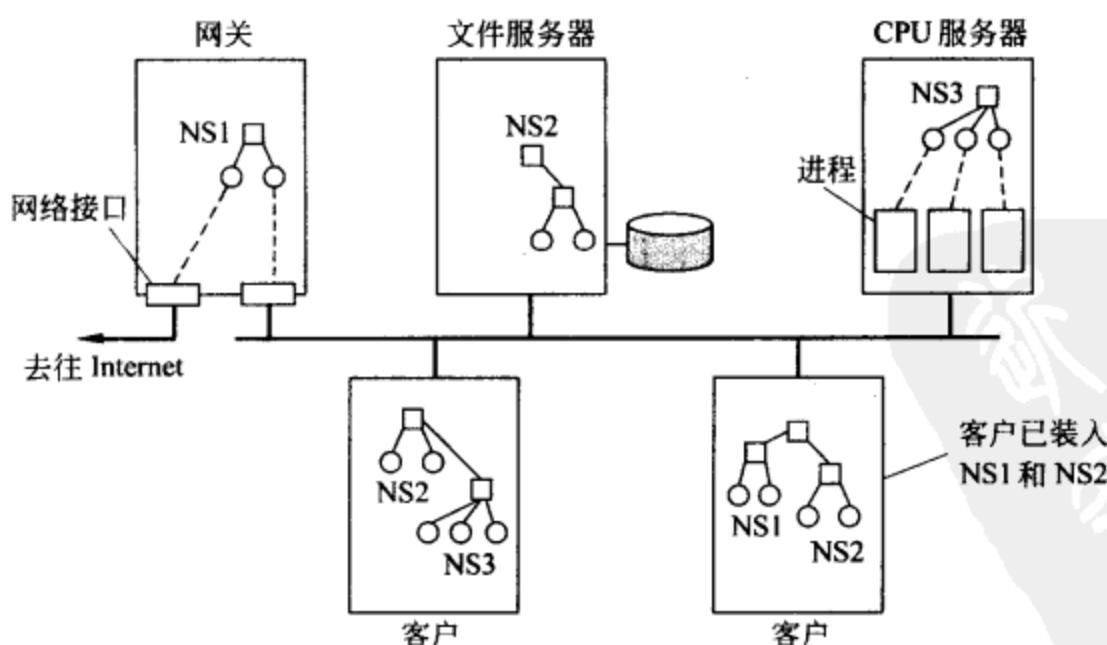


图 10.35 Plan 9 的一般组织结构

在 Plan 9 中不能将服务器和客户区分得很清楚。比如，服务器经常担当其他机器的客户，而客户也可以向服务器输出它们的资源。

1. 通信

网络上的通信是通过称为 9P 的标准协议实现的，该协议是专门为面向文件的操作设计的协议。比如，9P 提供打开、关闭文件的操作，读、写数据块的操作，以及遍历层次式名称空间的操作。9P 运行于可靠的传输协议之上。对于局域网，它使用称为 IL (Internet link, Internet 链接) 的可靠数据报协议。IL 是一种可靠的、基于消息的协议，它提供 FIFO 顺序的消息传递机制。对于广域网，9P 使用 TCP 协议。

Plan 9 的一个有趣特性是它在网络接口层处理通信的方式。网络接口是由文件系统表示的，因此，它由一些特殊文件组成。这种方法类似于 UNIX，尽管 UNIX 中的网络接口是由文件表示的，而不是由文件系统表示。注意，此处的文件系统仍是包含所有数据和元数据的逻辑块设备，而其中的数据和元数据是由一些文件组成的。比如，在 Plan 9 中，单独的 TCP 连接是由子目录表示的，该子目录包含图 10.36 所示的文件。

文件	描述
ctl	用于写协议特有的控制命令
data	用于读、写数据
listen	用于接收引入的连接建立请求
local	提供连接的呼叫方的信息
remote	提供连接的另一方的信息
status	提供连接的当前状态的诊断信息

图 10.36 Plan 9 中，与单一 TCP 连接关联的文件

文件 `ctl` 用于向连接发送控制命令。比如，为使用端口 23 与 IP 地址为 192.31.231.42 的机器建立 Telnet 会话，发送者要向文件 `ctl` 写入文本字符串“connect 192.31.231.42! 23”。而接收者应当已经在它自己的 `ctl` 文件写入字符串“announce 23”，这说明它可以接收输入的会话请求。

`data` 文件通过简单地执行 `read` 操作和 `write` 操作来交换数据。这些操作采用文件操作的通用 UNIX 语义。比如，为了向连接写入数据，进程简单地调用以下操作

```
res = write(fd, buf, nbytes);
```

其中 `fd` 是打开数据文件时返回的文件描述符，`buf` 是指向缓冲区的指针，它包含将要写入的数据，`nbytes` 是应从缓冲区提取的字节的数量。实际写入的字节的数量被返回，并存储在变量 `res` 中。

文件 `listen` 用于等待连接建立请求。进程宣布它愿意接受新连接后，它可以在文件 `listen` 上执行阻塞的 `read` 操作。如果一个请求进入，该调用向一个新的 `ctl` 文件返回一个文件描述符，这个文件描述符对应于新创建的连接目录。

2. 进程

Plan 9 有多种服务器。每种服务器都实现层次式名称空间。最简单的实例是 Plan 9 文件服务器，该服务器是运行于专用机器上的单机系统。它使用的命名图是一棵树，这棵树表示那些存储在多个磁盘上的文件。逻辑上，这个服务器被组织为图 10.37 所示的三层存储系统。

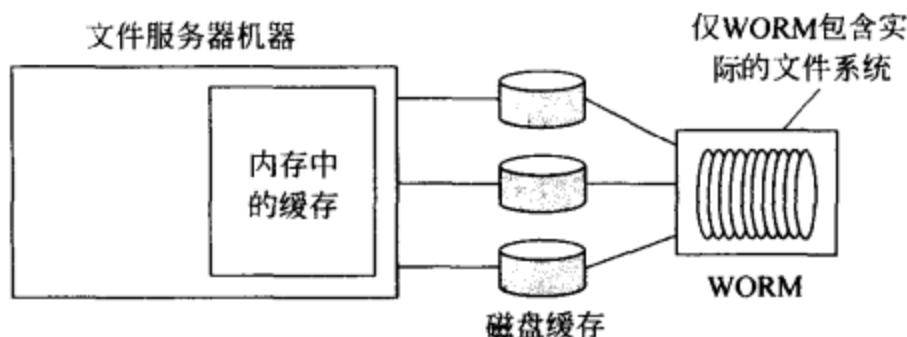


图 10.37 Plan 9 文件服务器

最底层由提供大容量存储的 WORM(Write-Once, Read-Many, 写一次, 读多次)设备构成。实际上, 它类似于可擦写 CD, 但是它的容量更大。存储在这个设备上的文件系统构成了客户所见的实际文件系统。

中间层由许多磁盘组成, 它们担当 WORM 设备的巨大缓存系统。访问文件时, 从 WORM 设备读取文件, 并将文件存放在磁盘上。对文件的修改也暂时存储在磁盘上, 这类似于 UNIX 系统中的传统的缓冲区缓存。每天一次向 WORM 设备写入所有的改动, 因而, 它以天为单位提供了整个文件系统的增量备份。

最高层由大量的内存缓冲区构成, 这些内存缓冲区以磁盘的内存高速缓存的形式操作。同样, 从磁盘读入文件, 对文件的改动存储在内存中的缓存中, 内存中的缓存不时地将这些改动写入磁盘。

一个完全不同的 Plan 9 服务器是实现 8¹/₂窗口系统的服务器(Pike 1991)。从概念上说, 这种服务器对相当于设备(比如, 鼠标或屏幕)的客户程序提供文件。这些设备的接口像文件, 也就是说, 这些设备并不是以文件的形式实现的, 但是它们表面上就像文件一样。比如, 向运行于窗口中的程序提供一个称为/dev/mouse 的文件, 以捕获鼠标在其窗口内的当前位置。每个窗口都有其私有版本的/dev/mouse。同样, 指定窗口的键盘输入也是从名称为/dev/cons 的文件读取, 该文件也是每个窗口私有的。与此类似, 其他文件为其他服务, 比如窗口控制函数和图像输出提供访问方法。

Plan 9 采用的方法提供了其他可选择的有趣方法来提供服务。比如, 称为 exportfs 的服务器允许机器在网络上输出其(部分)本地名称空间。实际上, 这台服务器接收 9P 消息, 并将那些消息翻译成本地系统调用。举例来说, 假设机器 M 提供多个网络接口, 可以通过称为/net 的本地目录访问这些网络接口。设/net/inet 表示提供外部网络的低级访问的网络接口。如果 M 输出/net, 那么客户通过在本地装入/net, 并随后打开/net/inet, 就可以将 M 用作网关。通过向那个文件写入数据, 客户将有效地向外部网络发送消

息，尽管客户机器自身并没有到达那个网络的接口。

使用类似的方式，客户可以向远程计算服务器输出它自己的文件。因此，客户可以在远程主机上启动程序。通过使客户自己的文件对于那个程序本地可用，实际计算就发生在那台主机上，而不是发生在客户机器上。但是，对于那个程序来说，它好像在使用客户的本地名称空间运行。注意，这种方法有效地使客户端机器变成文件服务器，以使程序运行于远程主机。

3. 命名

如上所述，Plan 9 的每个进程都有自己的私有名称空间，这是通过在本地装入远程名称空间构建的。Plan 9 的命名的一个有趣特性是可以在同一个挂接点装入多个名称空间，这就是所谓的联合目录（union directory）。在这种目录中，不同的文件系统好像是经过了布尔代数的或运算（尽管它们存在着强加的顺序，我们将马上讨论这一点）。比如，假设文件系统 FS_A 有子目录 /home 和 /usr，而文件系统 FS_B 有子目录 /bin、/src 和 /lib。如图 10.38 所示，如果客户在同一个挂接点，比如 /remote 装入这两个文件系统，那么随后这个目录好像包含 5 个子目录。

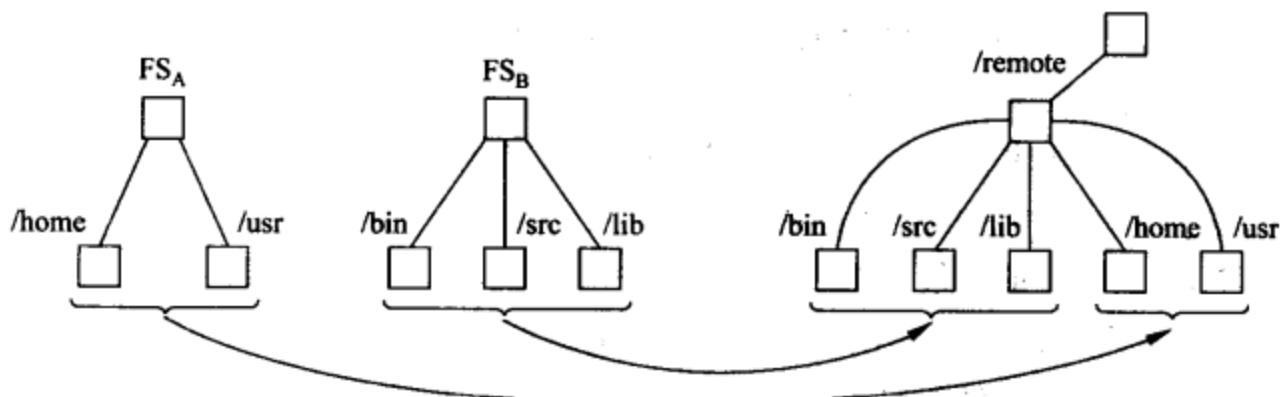


图 10.38 Plan 9 的联合目录

联合目录的创建是以指定的顺序装入文件系统的方式进行的。这个排序用于解决名称冲突。比如，如果文件系统 FS_A 也有一个 /bin 子目录，那么在装入 FS_A 之后装入 FS_B 时，把 FS_B 的子目录 /bin 置于 FS_A 的子目录之后。其效果是，搜索名称 /remote/bin/prog 时，如果 FS_A 中存在这个文件，那么将返回 FS_A 中该文件的文件标识符，否则，搜索操作继续搜索 FS_B 的 /bin 目录。

当文件服务器或设备创建文件时，该文件被标有两个整数路径和版本号。路径是一个唯一的文件号，它用于标识相对于服务器或设备的文件。每次文件被修改时，它的版本号加 1。为了识别系统范围内的文件，装入文件系统的客户为其装入的文件系统所在的服务器或设备维护类型和设备号。这些数字相当于 UNIX 所用的主设备号和辅设备号。因此，4 个数字全局唯一地标识了所有文件：两个数字用于标识文件驻留的服务器或设备，两个数字用于唯一地标识与其主机相关的文件。

4. 同步

为了提高效率，许多分布式文件系统在打开文件时就向客户传送文件（的一部分）。

因此,可能发生一个文件存在多个拷贝,而不同的客户并发地修改这些拷贝的情况。在会话语义的情况下,最后关闭文件的客户所作的更新会生效,其他客户的更新将会被丢弃。

Plan 9 通过让文件服务器总是保留一份文件的拷贝来实现 UNIX 文件共享语义。所有更新操作总会被转发到服务器。并发的客户按照服务器决定的顺序执行它们的操作,而更新从不会被丢弃。

5. 缓存和复制

Plan 9 对缓存和复制提供最低限度的支持。客户可以使用称为 cfs 的特定的用户级服务器缓存文件。通常,客户端机器启动时,cfs 被自动调用。cfs 实现直写式缓存策略,即更新操作总是立即被发送到服务器。

为了避免不必要的文件传送,如果先前缓存的拷贝仍然有效,那么客户可以使用先前缓存的拷贝。有效性是通过比较被缓存的文件和服务器上的文件的版本号检验的。如果从最后一次被缓存到客户端当前访问的期间,文件已经被修改,那么它们的版本号就会不同。如果这样,客户就使它的高速缓存无效,并从服务器获取新的拷贝。

6. 安全性

Plan 9 的用户身份验证方法类似于 Needham-Schroeder 协议中的方法。用户必须从身份验证服务获得票据,以建立到达另一个用户的安全通道。原则上,应验证用户的身份,而不是机器的身份。为了让机器或服务代表用户执行操作,Plan 也支持委派。

保护文件的方法与 UNIX 所用的方法相同。与其他文件系统相比,其不同之处在于 Plan 9 的组概念。在 Plan 9 中,组只被视为另外的用户,它可能有认同的领导者。组领导者拥有特殊的权限,比如允许它改变文件的组权限。如果组没有领导者,那么所有组成员是平等的。

10.3.2 xFS: 无服务器的文件系统

下一个与众不同的分布式文件系统的实例是 xFS(Anderson 等 1996),xFS 是作为 Berkeley NOW 项目的一部分开发的(Anderson 等 1995)。xFS 的与众不同之处在于它的无服务器设计,整个文件系统分布于多台机器,包括客户。这种方法完全不同于大多数其他文件系统,后者通常以集中式的方式组织,即使系统中存在多个用于分发和复制文件的服务器。比如,AFS 和 Coda 就属于后面一类。

需要指出,另一个称为 XFS(即首字母为大写字母“X”)的系统是一个完全不同的分布式文件系统,它也是在开发 xFS 的同一时间开发的(Sweeney 等 1996)。下面所用的“xFS”和“XFS”都是指作为 NOW 项目的一部分而开发的系统。

1. xFS 概述

xFS 设计为运行于高速局域网之上,即其中的机器通过高速连接互联。很多现代网络都满足这一要求,特别是工作站的群集(我们曾在第 1 章简要介绍过它)。通过将数据和控制完全地分布于局域网内的机器,xFS 的设计人员旨在达到比使用(可能是复制的)

文件服务器的传统分布式文件系统更高的可扩展性和容错性。xFS 的一个原型实现运行于一些通过 Myrinet 网络连接的 Sun SPARC 和 UltraSPARC 工作站之上 (Boden 等 1995)。

在 xFS 的体系结构中,有三种不同类型的进程。存储服务器是负责存储文件的某些部分的进程。多个存储服务器共同地实现虚拟磁盘的阵列,此阵列类似于以 RAID 的形式实现的磁盘阵列(Chen 等 1994)。元数据管理器(metadata manager)是负责记录文件数据块的实际存储位置的进程。注意,同一文件的数据块可能散布于多个存储服务器。管理器将来自客户的请求转发到适当的存储服务器。由此看来,元数据管理器作为文件数据块的定位服务器运行。最后,xFS 的客户是接收用户请求以执行文件操作的进程。每个客户有缓存的能力,并且可以向其他客户提供被缓存的数据。

xFS 的基本设计原则是任何机器都可以担当客户、管理器和服务器的角色。在完全对称的系统中,每台机器都运行这三个进程。但是,使用专用机器运行存储服务器,而其他机器运行客户或管理器进程也是可能的。图 10.39 显示了这种方法。

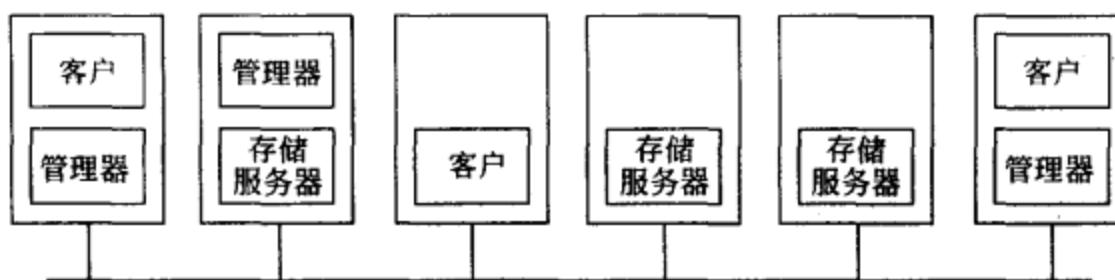


图 10.39 xFS 进程在多台机器上的一种典型分布

下面,我们简要讨论一下 xFS 的整体设计。其详细内容可参看(Anderson 等 1996)。xFS 的开发经验以及技术细节的描述发表于(Wang 等 1998)。一个有些与 xFS 相似的方法是将存储分布于多个网络附带的磁盘,如(Gibson 等 1998)所述。其作者指出传统分布式文件系统,比如 NFS 和 AFS,如何通过存储设备的分布来利用类似 xFS 的方法。

2. 通信

最初,xFS 中的所有通信都是使用 RPC 处理的。但是后来,由于某些原因,xFS 放弃了这种方法,最重要的原因是其性能低下。RPC 的另一个问题是它通常假设通信是 peer-to-peer 的。因为数据和控制完全分布于多台机器,所以单一客户的请求可能涉及一连串进程间的通信。RPC 服务器必须显式地记录未完成的请求,并把输入的响应匹配到适当的请求。因而,简单的阻塞机制是行不通的。

由于以上原因,xFS 中的通信被活动消息(active message)取代(van Eicken 等 1992, Chun 等 1998)。在活动消息中,接收方的处理程序以及调用它的必要参数都是指定的。消息到达时,处理程序被直接调用,并运行直至程序完成。处理程序执行时,不能传递其他消息。这种方法的主要优点体现在效率方面。但是,活动消息使通信的设计变得更复杂。比如,不允许阻塞处理程序。处理程序还应该相对较小,这是因为处理程序的执行妨

碍了处理程序所在主机的其他网络通信。

3. 进程

现在,我们详细地研究一下 xFS 中的各种进程。如上所述,xFS 存储服务器共同实现了虚拟磁盘的阵列。更具体地说,xFS 实现基于 Berkeley LFS(log-structured file system,日志结构化文件系统)的存储系统(Rosenblum 和 Ousterhout 1992)。

在 LFS 中,写入文件系统的各种数据,包括对 inode(信息节点)、目录块和数据块的修改都被存入缓冲区,用一个单独操作将它们一起写入磁盘。所有的写入数据共同组成一个(固定大小的)记录段,该记录段被追加到日志后面。因为数据此时可能散布于整个日志,所以这种方法的主要问题是查找文件块。为此,使用一个称为 imap 的附加表将 inode 引用映射为它们在日志中的位置。一旦找到了 inode,也就可以定位文件块了。

为了提高性能和可用性,xFS 采用 Zebra 文件系统所采用的方法(Hartman 和 Ousterhout 1995)。在 Zebra 文件系统中,日志分布于多台服务器,日志的组织结构类似于一类 RAID 中数据的组织结构。图 10.40 显示了这种方法。每当客户向日志追加包含更新数据的记录段时,它将记录段分成几个大小相等的分段,这被称为带区划分(striping)。每个段存储在不同的存储服务器上。客户还计算出一个奇偶校验分段(parity fragment),该分段存储在附加服务器上。如果任何一个服务器出现故障,那么可以从奇偶校验分段和存储在其他无故障服务器上的分段重新计算出一个段来。

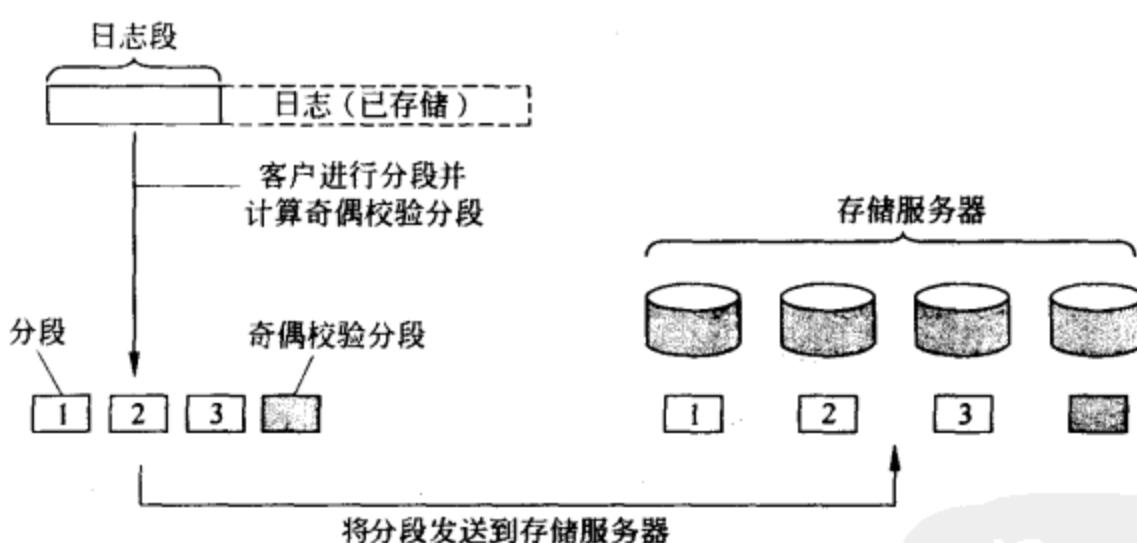


图 10.40 xFS 的基于日志的带区划分的原理

xFS 中的存储服务器被组织成带区组(stripe group)。将记录段划分成多个分段时,客户向一个带区组中的服务器写入这些分段。这种方法不同于 Zebra 所采用的方法,在 Zebra 方法中,记录段的各个分段总是被写入所有存储服务器。用带区组代替所有存储服务器,使 xFS 能够容易地添加更多的存储服务器:因为它没有记录段的各个分段需要存储于所有服务器(因而由所有服务器管理)的要求。它使用一个全局复制的表(table)来查找哪些服务器属于哪个带区组。

使用日志结构文件系统要求我们记录数据的实际存储位置。为此，xFS 中的每个文件都有一个关联的管理器。这个管理器维护 inode 引用的 imap 表，通过这个表，它可以在日志中找到文件的当前 inode。每当在文件上执行更新操作时，文件的 inode 也会改变。而这种改变要求向日志追加一个新版本的 inode，而这又要求文件的管理器必须得到通知以使它更新其 imap。

xFS 将文件的管理分布于多个管理器，而不是只使用一个单一的管理器。也就是说，系统中可能存在很多管理器，它们共同维护文件数据的存储位置的信息。那么，搜索一个文件要求客户联系该文件的管理器以访问该文件。一个单独的、全局复制的表，称为管理器映射(manager map)，用于搜索给予文件标识符的管理器。文件标识符相当于前面所说的 inode 引用。

如果我们忽略客户端缓存，那么 xFS 的客户是相当简单的进程，它只提出读、写文件的请求。图 10.41 显示从文件 f 读取数据块 b 的基本方法。客户首先在目录中搜索文件 f，搜索操作返回文件标识符 fid，如图所示的步骤 1。然后，使用该标识符在管理器映射中搜索 f 所关联的元数据管理器，如图 10.41 所示的步骤 2。

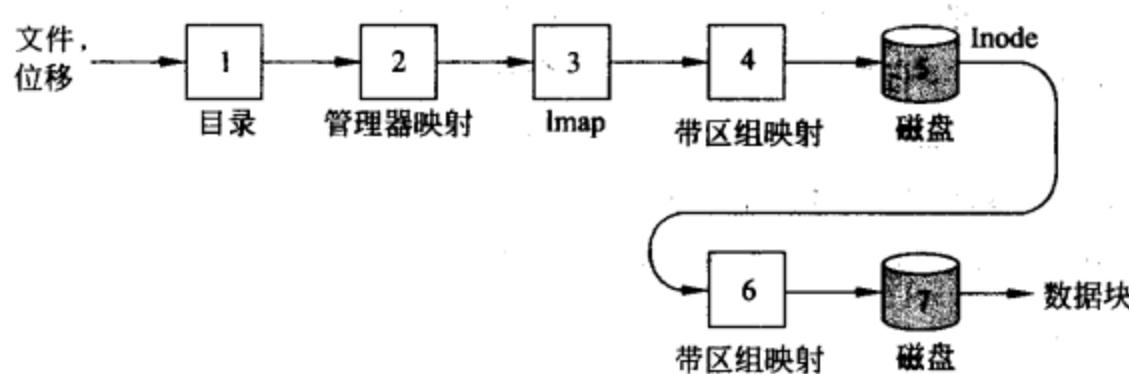


图 10.41 读取 xFS 中的数据块

定位到管理器之后，fid 和 b 被传送到管理器，管理器在它的内部表中搜索实际的 inode(步骤 3)。此时，在日志中应找到该 inode 的确切位置。这一位置信息包括带区组标识符、记录段标识符和该记录段中的位移。

现在，需要做的是确切地找到该 inode 所在的服务器。这要求搜索文件所写入的带区组，如图所示的步骤 4。包含 inode 的记录段被写入到一些服务器中，而这个搜索操作返回这些服务器的列表。通过记录段标识符和位移，客户可以计算出哪个服务器实际存储该 inode，并向它传送该 inode 的磁盘地址，如图所示的步骤 5。然后，重复执行步骤 4 和步骤 5，但这次是为了读取数据块 b。

4. 命名

除了使用各种系统范围的标识符来搜索管理器、块、inode 等的位置之外，xFS 的命名没有其他特殊之处。图 10.42 总结了 xFS 所用的各种标识符和数据结构，此图选自 (Anderson 等 1995)。

数据结构	描述
管理器映射	将文件 ID 映射到管理器
imap	将文件 ID 映射到文件的 inode 的日志地址
inode	将块号(即,位移)映射到块的日志地址
文件标识符	用于索引管理器映射的引用
文件目录	将文件名映射为文件标识符
日志地址	带区组 ID、记录段 ID 和记录段位移的三元组
带区组映射	将带区组 ID 映射为存储服务器的列表

图 10.42 xFS 所用的主要数据结构

5. 缓存

xFS 中的客户维护本地块高速缓存。其一致性方案类似于 Coda 的一致性方案,只是它缓存的是数据块,而不是整个文件。也就是说,当一个客户想要写入一块数据时,它联系该数据块所属文件的管理器,并请求写权限。管理器首先使由其他客户缓存的该数据块的所有拷贝无效,然后,它授予该客户写权限。这样,管理器就记录了数据块被缓存的位置,以及哪个客户被授予了写权限。后者也被称为当前所有者。

只要所有者拥有一个数据块的写权限,它不需要联系管理器就可以使用它的缓存拷贝来执行更新操作。当另一个客户想要访问该数据块时,所有者的写权限被撤销。此时,清洗当前所有者的高速缓存,并存储该数据块,直到它可以被转发到一个存储服务器。另外,该数据块被发送到新的客户,而这个新的客户成为下一个所有者。

xFS 进一步实现了协作缓存(collaborative caching)。在该方法中,每当客户想要访问一个数据块时,它联系适当的管理器,如上所述。但是,管理器不是在存储服务器上定位所请求的数据块,而是先检查当前是否存在其他已经缓存了所请求的数据块的客户。如果存在这样的客户,那么它就从高速缓存获取一份拷贝,并将其返回给提出请求的客户。

6. 容错

通过将存储服务器实现为保留冗余信息的带区组,使得 xFS 的可用性比传统方法的可用性要高。每当一个单独的服务器崩溃时,它可以通过获取其他服务器上的奇偶校验分段来恢复分段。如果奇偶校验分段丢失了,那么它可以容易地从现有的分段计算出来。但是,如果同一组内的多个服务器崩溃,那么需要采取特殊的措施,但是这些措施还未被实现。

管理器的恢复主要是通过不时地对管理器当前持有的数据设置检查点得到支持。一个难以处理的问题是恢复管理器的映射。设置检查点对此有所帮助,但是仍需考虑检查点后的更改。为此,各个客户记录它们自最后一个检查点以来向管理器发送的更新。这样,管理器可以请求客户重放一个检查点之后的更新,从而使它的 imap 进入与存储服务器上的信息一致的状态。

xFS 中的恢复只实现了一部分。许多必要的机制并未包括在原型实现中。详细内容请参看(Anderson 等 1996)。

7. 安全性

XFS 仅对安全性提供最低限度的支持。除了实现常见的访问控制机制外,xFS 要求客户、管理器和存储服务器运行于可信任的机器上以实施安全性。但是,它提供了一种让不被信任的 NFS 客户使用安全 RPC 访问一些 xFS 机器的简单措施,如 10.1.8 小节所述。实际上,NFS 客户使用安全 RPC 联系 xFS 客户。因而,xFS 客户将作为那个 NFS 客户的 NFS 服务器。

10.3.3 SFS: 可扩展的安全性

在最后一个实例中,我们简要地讨论一个完全不同的处理安全性的方法。在安全文件系统(secure file system,SFS)中,其主要设计原则是将密钥管理和文件系统安全性分开(Mazieres 等 1999)。也就是说,SFS 保证客户在没有适当的密钥时不能访问文件。但是,如何获得这个密钥是与文件访问完全分开的。

1. SFS 概述

图 10.43 显示了 SFS 的整体组织结构。为了确保其在多种类型的机器上的可移植性,SFS 已与多个 NFS 版本 3 组件集成在一起。客户机器有三个不同的组件,其中不包括用户程序。NFS 客户被用作用户程序的接口,它与 SFS 客户交换信息。对于 NFS 客户来说,SFS 客户就好像是另一个 NFS 服务器。也就是说,这两个组件使用 ONC RPC 系统通信,并遵循 NFS 协议。

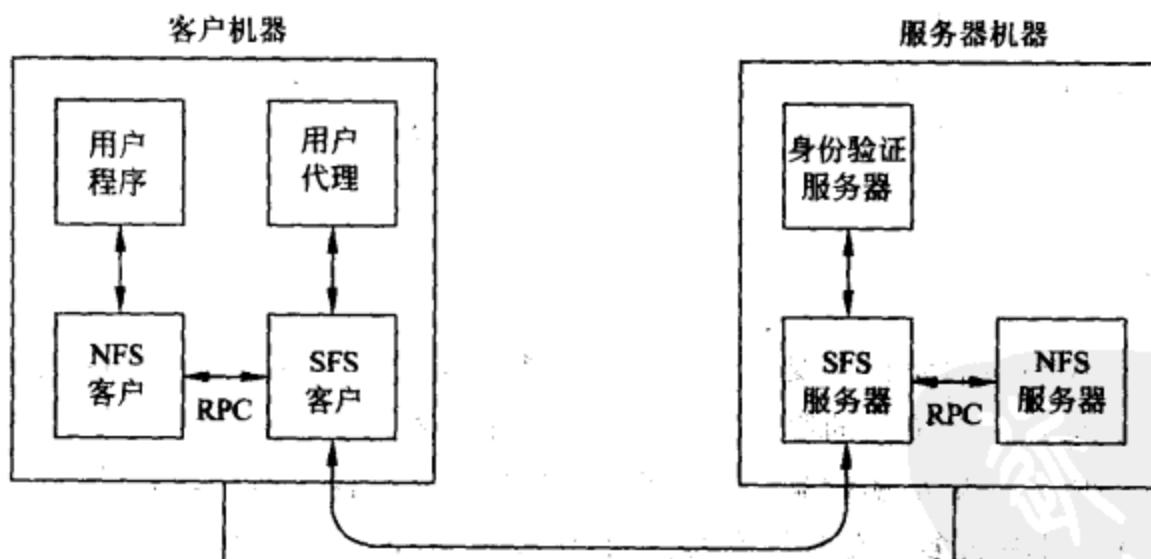


图 10.43 SFS 的组织结构

SFS 客户负责建立与 SFS 服务器通信的安全通道。它还负责与本地可用的 SFS 用户代理(SFS user agent)的通信,SFS 用户代理是一个自动处理用户身份验证的程序。SFS 并未规定如何进行用户身份验证。根据其设计目标,SFS 将这些身份验证问题分离出来,并且对于不同的用户身份验证协议,使用不同的代理。

在服务器端也有三个组件。使用 NFS 服务器也是出于可移植性的原因。NFS 服务器与 SFS 服务器通信,SFS 服务器是作为 NFS 服务器的 NFS 客户运行的。SFS 服务器构成 SFS 的核心进程。该进程负责处理来自 SFS 客户的文件请求。与 SFS 代理类似,SFS 服务器与一个单独的身份验证服务器通信来处理用户的身份验证。

2. 通信

SFS 客户和 SFS 服务器通信的协议是稍加改变的 NFS 协议版本 3。当客户打开文件时,它也接收其文件属性,然后将这些属性缓存起来。在 SFS 中,这些属性的缓存受到租用的约束。当租用到期时,SFS 客户应将这些文件属性视为无效的。另外,SFS 服务器能够回叫客户,并使缓存的文件属性无效。实际上,SFS 被认为实现了一个已并入 NFS 版本 4 的重要特性。

3. 进程

从图 10.43 还可看出,SFS 实现传统的客户-服务器系统。但是,一个重要设计特点是 SFS 客户没有针对位置的配置选项。也就是说,SFS 客户完全独立于其用户访问 SFS 的位置。这种方法与用户应能从世界的任何位置访问其文件的思想相符。SFS 服务器只可以向用户授予访问权,而不向客户授予访问权。

4. 命名

与其他分布式文件系统相比,SFS 的独特之处在于它的名称空间的组织结构。SFS 提供一个全局名称空间,该名称空间以目录 /sfs 为根。SFS 客户允许其用户在这个名称空间内创建符号链接。

更重要的是,SFS 使用自证明路径名来命名它的文件。这种路径名本质上携带所有用于验证由其命名的文件所在 SFS 服务器的身份的信息。自证明路径名由三部分组成,如图 10.44 所示。

/sfs	LOC	HID	路径名
$\underbrace{\quad}_{/sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere/home/steen/mbox}$			

图 10.44 SFS 中的自证明路径名

路径名的第一部分由位置 LOC 组成,LOC 是识别 SFS 服务器的 DNS 域名或其对应的 IP 地址。SFS 假定每台服务器 S 有一个公共密钥 K_s^+ 。自证明路径名的第二部分是主机标识符 HID,它是通过服务器的位置和它的公共密钥上的密码编译的散列 H 计算出来的:

$$HID = H(LOC, K_s^+)$$

HID 由基数为 32 的 32 位数字表示。第三部分是实际存储文件的 SFS 服务器上的本地路径名。

每当客户访问 SFS 服务器时,它可以简单地通过向该服务器请求它的公共密钥来验

证它的身份。然后，客户可以使用众所周知的散列函数 H 计算 HID ，并用该值与在路径名中获取的值进行比较。如果两个值相同，那么客户就知道它正在和名称与获取位置相同的那个服务器通信。此时，该服务器应该验证用户的身份，但是如上所述，服务器如何验证用户的身份与 SFS 无关。(Mazières 等 1999) 描述了一种实现，在这种实现中，客户使用一个本地代理管理用户的专用密钥，并在服务器端使用该密钥进行身份验证。

5. 安全性

这种方法如何把密钥管理和文件系统安全性分开呢？SFS 所解决的问题是服务器的公共密钥的获取可以完全地与文件系统安全性问题分开。一种获取服务器密钥的方法是让客户联系服务器并向其请求密钥，如上所述。

但是，客户在本地存储一些密钥也是可以的，比如，由系统管理员存储。这样，客户就不需要联系服务器了。在客户解析路径名时，在本地搜索服务器的密钥，然后，它可以使 用路径名中的位置部分来验证主机 ID。

为了简化问题，可以使用符号链接实现命名透明性。比如，假设客户想要访问名为
`/sfs/sfs.vu.cs.nl: ag62hty4wior450hdh63u62i4f0kqere/home/steen/mbox`
的文件。为了隐藏主机 ID，用户可以创建一个符号链接

`/sfs/vucs→/sfs/sfs.vu.cs.nl: ag62hty4wior450hdh63u62i4f0kqere`

然后，只使用路径名 `/sfs/vucs/home/steen/mbox`。对该名称的解析将自动将该名称扩展为完整 SFS 路径名，然后，使用本地的公共密钥，验证名称为 `sfs.vu.cs.nl` 的 SFS 服务器的身份。

与此类似，SFS 可以被证书颁发机构支持。通常，这种颁发机构会维护指向其代理的 SFS 服务器的链接。举例来说，考虑一个 CA(证书颁发机构)，它运行名为

`/sfs/sfs.certsf.com: kty83pad72qmbna9uefdppioq7053jux`

的 SFS 服务器。假设客户已经安装了一个符号链接

`/certsf→/sfs/sfs.certsf.com: kty83pad72qmbna9uefdppioq7053jux`

CA 可以使用另一个指向 SFS 服务器 `sfs.vu.cs.nl` 的符号链接

`/vucs→/sfs/sfs.vu.cs.nl: ag62hty4wior450hdh63u62i4f0kqere`

这里，客户可以简单地引用 `/certsf/vucs/home/steen/mbox`，因为它知道它正访问的文件服务器的公共密钥已经被 CA 认证。

密钥管理技术的其他实例可参看(Mazieres 等 1999)。SFS 的实现和性能方面的详细描述可参看(Mazieres 2000)。后者也包含 SFS 用以实现安全性的各种协议的详细描述。

10.4 分布式文件系统的比较

我们已经讨论了 5 种不同的文件系统。尽管这些系统有许多相同之处，但是它们也在很多方面不同。下面，我们比较一下这些系统，主要比较 NFS 和 Coda。

10.4.1 设计理念

通常,各种文件系统的设计目标是截然不同的。NFS 旨在提供一个系统使客户能够透明地访问存储在特定远程服务器上的文件系统。从这个角度而言,其目标很像只支持远程对象的基于对象的系统。在 NFS 版本 4 中,它的设计人员还要通过隐藏延迟来提供广域连接上的访问透明性。这种隐藏是通过允许客户缓存整个文件并在本地处理所有操作实现的。

Coda 的主要设计目标是高可用性,甚至是出现网络分区和断开连接的操作时的高可用性。Coda 还继承了 AFS 的设计目标,AFS 的最重要的设计目标是达到可扩展性。出现网络分区时的高可用性与可扩展性的结合,使 Coda 有别于大多数其他分布式文件系统。

Plan 9 的主要目标是提供分布式分时系统,而系统中的所有资源都以相同的方式访问,即以文件的方式访问。如前所述,Plan 9 也可以称为基于文件的分布式系统。

xFS 追求高可用性和可扩展性,从这种意义上说,xFS 具有许多与其他分布式文件系统相同的设计目标。但是,比较重要的一点是 xFS 旨在通过无服务器系统实现这些目标,这可以视为它的主要设计目标。

SFS 的目标是提供可扩展的安全性。它把管理问题和文件系统安全性分开,并允许任何用户不受中心证书颁发机构的干涉而启动它们自己的 SFS 服务,以此实现这一目标。文件的路径名有效地携带了文件所在服务器的公共密钥,从这种意义上说,名称空间的组织结构起到了重要作用。

在考虑每个系统所支持的基本文件模型时,这些系统还存在其他重要的不同之处。比如,NFS、Plan 9 和 SFS 都支持远程访问模型,在这种模型中,服务器仍实际负责文件,并执行所有的文件操作。但是,可以认为 Coda 或多或少地支持上载/下载模型,这是因为 Coda 继承了 AFS 主动缓存整个文件的特性。而 xFS 采用了另一种方法,它实现日志结构文件系统。

10.4.2 通信

下面在通信方面进行比较,大多数分布式文件系统实际上都依赖于某种形式的 RPC。NFS、Coda 和 SFS 直接使用底层 RPC 系统,它们有时优化这些 RPC 系统以处理特殊情况。

可以认为 Plan 9 也使用 RPC 系统,但是实际协议已经修改,以使其适于处理文件操作,这使它与其他协议有些不同。

一个有趣的观察结果是 xFS 最初也使用 RPC 系统来处理所有的通信。由于性能原因,以及 xFS 所需要的、固有的多方通信,RPC 系统被活动消息取代了。

10.4.3 进程

我们所列举的实例系统的区别在于总体考虑文件系统时,客户起作用的方式。使用第 1 章介绍的客户-服务器系统的组织结构,NFS 版本 3 的客户进程本质上可以认为是

“瘦的”。也就是说，实际上，大多数工作是文件服务器完成的，而 NFS 客户只是请求服务器执行操作。NFS 版本 4 的客户完成相当多的工作，这是因为 NFS 版本 4 允许它们缓存整个文件，允许它们在本地处理很多操作。也就是说，NFS 版本 4 的客户也可以被归为“胖的”而不是“瘦的”。

同样的推理也适用于代表 AFS 和 Coda 的客户软件的 Venus 进程。Venus 完成了客户的许多工作。它部分地负责整个文件的缓存、客户处于断开状态时服务器功能的模拟等。

相反，Plan 9 的设计人员却试图使客户尽可能地保持简单。在他们的设计中，服务器完成所有工作，而客户实际上只需是一个简单的终端。然而，它允许客户进程缓存文件，但是，即使根本不用缓存，系统仍会正常工作。

假如我们考虑缓存的情况，xFS 的客户进程也有资格称为“胖的”。否则，xFS 客户也不过是在存储服务器上执行文件操作。但是，xFS 的设计依赖于客户共同实现协作缓存以提高性能。

SFS 所采用的方法介于 Plan 9 的简单客户和 Coda 的复杂客户之间。SFS 客户是一个相对简单的组件，它并不依赖于它的本地位置的特定特征。它只提供对远程 SFS 服务器的访问。如果需要，它可以增加用于处理用户和服务器特定的身份验证的特殊代理。

比较各种系统时，会发现服务器是有很大不同的。实际上，只有 Coda 和 xFS 假定服务器是以组的方式组织的。每个组可以主管一个复制的文件，或者像 xFS 一样，负责一个带区文件。NFS、Plan 9 和 SFS 本质上假设文件位于一个单一的、非复制的服务器上。

10.4.4 命名

除了 SFS 之外，我们的实例文件系统都或多或少地支持相同的名称空间，此名称空间是通过装入操作构建的，这类似于 UNIX 系统。装入可以发生在目录粒度上，如 NFS 的目录（以及 SFS 的目录），也可以发生在文件系统粒度上，如 Coda、Plan 9 和 xFS（xFS 继承 LFS 和 Zebra 采用的方法）的文件系统上。

更有趣的是名称空间的组织方法。基本上，名称空间有两种组织方法。在第一种方法中，每个用户获得它们自己私有的名称空间。NFS 和 Plan 9 采用这种方法。每个用户拥有自己的名称空间的缺点是难以根据名称共享文件。为了减轻其中的一些问题，可以将部分名称空间的标准化。

第二种方法是提供一个全局共享的名称空间，如 Coda、xFS 和 SFS 的名称空间。在所有这些系统中，每个用户也能用私有的本地名称空间扩展全局的名称空间。比如，SFS 客户允许用户在本地创建符号链接。这些名称是用户私有的，而且对于不同 SFS 客户上的用户，它们是不可见的。

文件引用方面也存在不同之处。除 Coda 和 xFS 之外，没有其他系统使用系统范围唯一的文件引用。在 Coda 中，文件引用由两部分组成。第一部分是与位置无关的卷标识符，第二部分是在卷内唯一标识文件的句柄。这两部分一起构成了系统范围内唯一的名称。

xFS 使用另外的间接方法引用文件，即通过元数据管理器引用文件。本质上，文件引用指向一个管理器，而该管理器随后用于定位 inode。这些文件引用是系统范围内唯一的。

其他系统使用的文件引用基本上只在文件所在的文件系统（如 NFS 和 SFS）上是惟一的。

一的,或者只在负责文件处理的服务器(如 Plan 9)上是惟一的。实际上,这些方法都要求使用额外信息定位文件。

10.4.5 同步

考虑各种分布式文件系统时,最重要的同步特性是每个系统提供的文件共享语义。严格地说,NFS 提供会话语义,即服务器只保存最后关闭文件的进程所做的更新。Coda 只允许那些可以被串行化的会话,从这种意义上说,Coda 支持事务处理语义。但是,在断开连接的操作中,这些语义不能得到保证,这可能会导致以后需要恢复的更新冲突。

因为 Plan 9 中的所有文件操作基本上都是由文件服务器处理的。所以,Plan 9 提供 UNIX 语义。xFS 也支持这些语义,因为它提供(协作)缓存机制,根据这种机制,只允许文件块的当前所有者执行写操作。

SFS 中未规定文件共享语义。但是,因为 SFS 主要是基于 NFS 协议的,所以它至少提供会话协议。

10.4.6 缓存和复制

我们的所有实例系统都支持客户端缓存。只有 Plan 9 使用直写式高速缓存一致性协议,即每个写操作都立即被转发到服务器。其他系统实现回写式高速缓存,它们允许客户在清除高速缓存前对缓存数据执行一系列写操作。在 NFS 中(因而也在 SFS 中),采用哪种高速缓存一致性协议主要由实现决定,尽管 NFS 版本 4 要求文件关闭时向服务器传播修改。

xFS 缓存文件数据块。Plan 9 缓存数据的粒度是任意的,它取决于客户的访问模式。NFS(以及 SFS)并未解决这些问题,尽管人们预期在 NFS 版本 4 系统中,鉴于它对广域网的支持,它会支持整个文件缓存。Coda 显然支持整个文件缓存,这是它从 AFS 继承的一个特性。

除了 Coda 之外,服务器对文件复制的支持基本上未得到解决。NFS 版本 4 通过 FS_LOCATIONS 属性对复制的文件系统提供最低程度的支持。NFS 版本 3 根本没有处理复制的问题。同样,Plan 9 也没有对复制提供显式的支持。xFS 通过带区划分支持一种较弱形式的复制。Coda 显然考虑到卷可能被复制的情况,它使用 ROWA 协议维护一致性,但是出现网络分区时,它降为弱一致性协议。

10.4.7 容错性

在大多数情况下,对容错的支持是最低限度的,而且也只限于使用可靠的通信来提供支持。在 NFS 中,即使使用可靠的通信,也需要另外的支持,这是因为底层的 RPC 系统不提供最多一次的语义。Coda 采用了一种比较先进的方法,它使用它的缓存和复制能力来提供高可用性。xFS 使用带区划分来保护系统,使其不受一个带区组内的单一服务器崩溃的影响。

服务器崩溃而丢失某些资源的信息时,NFS 允许客户要求服务器归还某些资源,如锁。从这种意义上说,NFS 中的恢复主要基于客户。Coda 没有规定服务器如何恢复,但是它对于从网络分区的恢复提供复杂的支持。同样的技术也用于服务器组内的崩溃服务

器的重新集成。

xFS 使用检查点和客户记录日志的方法来帮助系统恢复,使系统恢复到管理器的数据与日志中存储的信息一致的点。但是,绝大部分的恢复机制并未实现。

10.4.8 安全性

NFS 版本 4 将安全性机制和它们的实现完全分开。为了实现安全通道,NFS 以 RPCSEC_GSS 的形式提供了一个标准接口。通过该接口,可以访问很多现存的安全性系统,实际使用哪种机制取决于 NFS 的实现。

Coda 和 Plan 9 提供安全通道,但是这两个系统都根据我们在第 8 章讨论的 Needham-Schroeder 身份验证协议实现它们的安全通道。这种方法使用共享的密钥。

xFS 不提供安全通道。由各种受到信任的机器来实现系统范围的安全性。为此,它需要采取特殊的措施,以允许不被信任的机器访问 xFS 系统。目前的方法是使用 NFS 版本 3 底层的安全 RPC。

SFS 使用自己的方法处理安全通道。实质上,它支持任何一种安全机制,但是它对密钥管理进行特殊的处理。通过使用自证明路径名,用户可以使用不同的方法验证服务器的身份。服务器使用特殊的代理验证用户的身份。

NFS 版本 4 为访问控制提供了大量的操作。ACL 是作为单独的文件属性提供的,这提供了许多灵活性。Coda 采用一种不同的方法,它只处理与目录操作有关的访问控制。Plan 9 和 xFS 主要遵循标准 UNIX 的方法,这是通过区分读、写和执行操作来实现的。最后,SFS 从 NFS 版本 3 继承其访问控制机制。

图 10.45 总结了 5 种不同分布式文件系统的各个要点。

要点	NFS	Coda	Plan 9	xFS	SFS
设计目标	访问透明性	高可用性	统一性	无服务器系统	可扩展的安全性
访问模型	远程	上载/下载	远程	基于日志	远程
通信	RPC	RPC	特殊方法	活动消息	RPC
客户进程	瘦/胖	胖	瘦	胖	中等
服务器组	无	有	无	有	无
装入粒度	目录	文件系统	文件系统	文件系统	目录
名称空间	每个客户	全局	每个进程	全局	全局
文件 ID 范围	文件服务器	全局	服务器	全局	文件系统
共享语义	会话	事务处理	UNIX	UNIX	N/S
缓存单元	文件(版本 4)	文件	文件	块	N/S
高速缓存一致性	回写式	回写式	直写式	回写式	回写式
复制	最低限度	ROWA	无	带区划分	无
容错	可靠的通信	复制与缓存	可靠的通信	带区划分	可靠的通信
恢复	基于客户	重新合成	N/S	检查点和记录日志	N/S
安全通道	现存的机制	Needham-Schroeder	Needham-Schroeder	无路径名	自证明
访问控制	许多操作	目录操作	基于 UNIX	基于 UNIX	基于 NFS

图 10.45 NFS、Coda、Plan 9、xFS 和 SFS 之间的比较。N/S 表示无特殊规定

10.5 小结

分布式文件系统形成构建分布式系统的重要模型。它们通常按照客户-服务器模型组织,使用客户端缓存和对服务器复制的支持来满足可扩展性的需求。另外,为实现高可用性,需要使用缓存和复制。

使分布式文件系统与非分布式文件系统不同的是共享文件的语义。理想情况下,文件系统可以使客户总是读取最近写入文件的数据。在分布式系统中实现这些 UNIX 文件共享语义是非常困难的。NFS 支持一种较弱形式的语义,称为会话语义,文件的最后版本取决于先前为写操作打开该文件并最后关闭该文件的客户。在 Coda 中,读数据的客户只有重新打开文件时才能看到最新的更新数据,从这种意义上说,文件共享遵循事务处理语义。Coda 的事务处理语义不涉及常规事务处理的所有 ACID 属性。在服务器控制所有操作的情况下,可以提供真正的 UNIX 语义,但是这样会造成可扩展性问题。

为了达到可接受的性能,分布式文件系统通常允许客户缓存完整的文件。NFS 和 Coda 支持这种整个文件缓存的方法,但是只存储文件的大部分数据也是可能的。一旦文件已经打开,并(部分地)传送到客户,则所有操作都在客户本地执行。当文件再次被关闭时,客户向服务器返回所有更新。另外,诸如 Plan 9 和 xFS 的系统支持块缓存,并将它与回写式高速缓存一致性协议结合。

通常不是直接在传输层之上建立分布式文件系统,而是假设存在一个 RPC 层,使所有操作都可以简单地表示为文件服务器的 RPC,而不必使用原始的消息传送操作。NFS、Coda 和 SFS 都遵循这种方法。RPC 层最好提供最多一次调用的语义,否则,这些语义将不得不作为文件系统层的一部分而被显式地实现,NFS 就是这样的。

Coda 与其他分布式文件系统不同的是它支持断开连接的操作。通过确保客户的高速缓存总是包含近期将使用的数据,即使当客户暂时断开与文件服务器的连接时,它也允许客户继续操作。这种方法需要特殊的高速缓存管理(称为储藏)的支持。储藏不容易实现,但是已经证明,有效的实现是可能的。

安全性是任何分布式系统,包括文件系统的最重要的方面。NFS 本身几乎不提供任何安全性机制,但是,它实现了标准化接口,这使得它可以使用不同的现有安全性系统,比如 Kerberos。相反,Coda 和 Plan 9 提出了它们自己的安全性机制。这些系统使用安全 RPC 来进行身份验证,通常,它们的身份验证是从 Needham-Schroeder 身份验证协议衍生而来的。SFS 允许文件名包含文件服务器的公共密钥的信息,在这种意义上,SFS 是不同的。这种方法简化了大规模系统中的密钥管理。实际上,SFS 通过将密钥包含于文件的名称之中来分发密钥。某些特殊工具可能有助于使这种方法对客户透明。

习题

1. 实现 NFS 版本 3 的文件服务器必须是无状态的吗?
2. NFS 不提供全局共享的名称空间。是否有模拟这种名称空间的方法?

3. 请简单扩展 NFS 的 lookup 操作,结合输出从其他服务器装入的目录的服务器,使它允许进行迭代名称搜索。
4. 在基于 UNIX 的操作系统中,使用文件句柄打开文件的操作只能在内核中实现。对于 UNIX 系统的用户级 NFS 服务器,请给出 NFS 文件句柄的可行实现。
5. 如本章所述,自动装入器安装符号链接,使用这样的自动装入器导致更加难以实现透明装入。为什么?
6. 假设 NFS 中一个文件的当前拒绝状态是 WRITE。另一个客户是否可以先成功地打开该文件,然后请求一个写操作锁?
7. 在考虑第 6 章所讨论的高速缓存相关性的情况下,NFS 实现的是哪种高速缓存相关性协议?
8. NFS 是否实现项目(entry)一致性?是否实现版本(release)一致性?
9. 我们陈述了 NFS 实现文件处理的远程访问模型。我们可以认为它也支持上载/下载模型。请解释其原因。
10. NFS 使用直写式高速缓存相关性策略缓存属性。它是否有必要立即转发所有的属性改变?
11. 本章所述的重复请求高速缓存在哪种程度上真正成功地实现最多一次的语义?
12. 在 NFS 中,当服务器进入宽限期时,哪种类型的安全性措施可以防止恶意的客户要求服务器归还从未授予给它的锁?
13. 图 10.21 表示客户可以完全透明地访问 Vice。这在某种程度上确实是正确的?
14. 使用 RPC2 的副作用处理连续的数据流很方便。请给出另一个实例,在该例子中使用一种仅次于 RPC 的针对应用程序的协议是有意义的。
15. 如果 Coda 中的物理卷从服务器 A 移动到服务器 B,那么需要对卷复制数据库和卷位置数据库做哪些改变?
16. 出现故障时,RPC2 提供什么调用语义?
17. 对于一个被多个读程序和单一写程序共享的文件,请解释 Coda 如何解决该文件上的读写操作冲突。
18. 在 Coda 中,是否有必要加密 Alice 登录时从 AS 接收到的明文令牌?如果有必要,应在何处进行加密?
19. 如果 Vice 服务器上的文件需要它自己指定的访问控制列表,如何实现这种情况?
20. Vice 服务器只提供对文件的 WRITE 权限是否合理?(提示:考虑客户打开文件时发生的情况。)
21. Plan 9 中的联合目录是否可以代替 UNIX 系统中的 PATH 变量?
22. Plan 9 是否可以实际用作广域分布式系统?
23. 与记录段的各个分段存储于所有存储服务器的方法相比,xFS 使用带区组的主要优点是什么?
24. 使用自证明路径名,是否总能保证客户和非恶意的服务器通信?

第 11 章 基于文档的分布式系统

WWW(World Wide Web, 万维网)的引入对计算机联网和分布式系统的普及做出了重大贡献。Web 的生命力在于其形式的相对简单性：一切内容都以文档来表示。在本章中，我们将讨论以 Web 为代表的基于文档的分布式系统。基于文档的分布式系统可以使用户把文档视为一种交换信息的简单而强大的手段。这种模型易于理解，因为它和人们日常生活中的许多模型非常相像。例如，在办公环境中，通信往往借助于备忘录、便笺和报告等来完成。

Web 是目前最重要的基于文档的分布式系统，并且在不远的未来仍将如此。实际上对于很多人来说，当他们谈论 Internet 时，他们指的就是 Web。因此，本章讨论的这种系统的第一实例就是 Web。

基于文档的分布式的另一个重要实例是 Lotus Notes，它的开发早于 Web。与 Web 不同，Notes 主要基于数据库而不是基于文件。今天，它仍然是一种广泛使用的系统，而且通常与 Web 技术相结合并提供工具来建立基于 Web 的服务。Notes 是本章要讨论的第二个基于文档的分布式系统。本章最后将对这两个系统进行比较。

11.1 WWW

可以把 WWW 视为一个巨大的分布式系统，其中包含数以百万的客户和服务器用于访问链接文档。服务器维护文档的集合，而客户为用户提供可以表示和访问这些文档的易于使用的界面。

Web 起源于位于日内瓦的欧洲粒子物理研究所 (European Particle Physics Laboratory, CERN) 的一个项目，其目的是使其地理位置上分散的众多研究者能够使用一种简单的超文本系统来访问共享文档。文档可以是任何能显示在用户的计算机终端上的内容，例如个人便笺、报告、图形、蓝图和图样等。通过文档的相互链接，无须集中更改就很容易把来自不同项目的文档集成为一个新文档。需要做的只是构建一个提供到其他相关文档的链接的文档(请参见 Berners-Lee 等 1994)。

逐渐地，Web 中的站点不再局限于高能物理领域，而是扩展到了世界范围。但是 Web 真正得到普及是在出现像 Mosaic (Vetter 等 1994) 这样的图形用户界面之后。Mosaic 提供了只需单击鼠标即可表示和访问文档的易用界面。通过这种界面，可以从服务器上获取文档，把它传递给客户，然后显示在屏幕上。对于用户而言，本地存储的文档和存储在世界上其他地方的文档没有什么概念上的区别。从这一点看，分布是透明的。

从 1994 年开始，Web 的发展主要由 WWW 协会 (WWW Consortium) 发起并控制。

该协会由 CERN 和 M. I. T. (美国麻省理工学院)合办,负责制订协议标准、改善互操作性和进一步增强 Web 的能力。它的主页为 <http://www.w3.org/>。

11.1.1 WWW 概述

WWW 实质上是一个巨大的客户-服务器系统,其数以百万的服务器分布在世界各地。每个服务器上都维护着一个文档的集合,每个文档以文件的形式存储(尽管文档也可以应请求而产生)。服务器接受获取文档的请求并把文档传送给客户。另外,服务器还接受存储新文档的请求。

指定一个文档最简单的方法是使用所谓的统一资源定位符(uniform resource locator, URL)。URL 与 CORBA 中的 IOR 或 Globe 中的联系地址相当。它指定了文档所在的位置,通常的方法是指定与文档相关服务器的 DNS 名称以及文件名,服务器可以在其本地文件系统中根据这个文件名查找文档。而且,URL 还指定了在网络上传送文档所使用的应用级协议。可用的协议有多种,我们下面会对它们进行解释。

客户与 Web 服务器通过一个称为浏览器(browser)的专门的应用程序进行交互。浏览器负责正确地显示文档。浏览器还负责接受用户的输入,通常是让用户选择对另一个文档的引用,然后去获取并显示被选文档。其总体结构如图 11.1 所示。

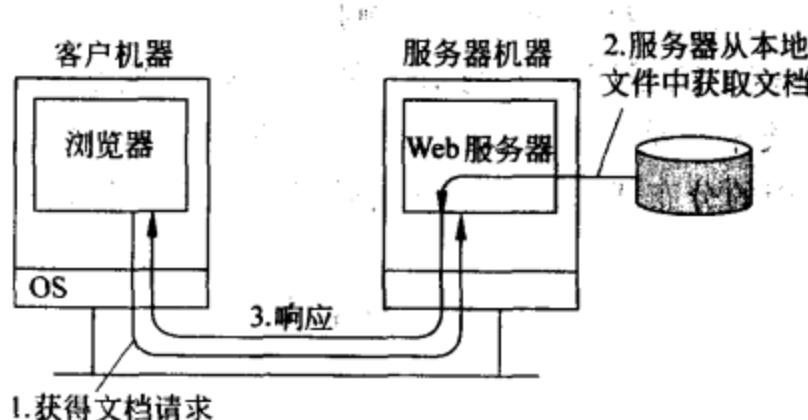


图 11.1 Web 的总体结构

10 年来,Web 的发展取得了相当可观的进步。到现在为止,已开发出大量的方法和工具以产生可以由 Web 客户和 Web 服务器处理的信息。下面,我们将深入讲述 Web 作为一个分布式系统的工作原理。但我们将不讨论实际建立 Web 文档的大部分方法和工具,因为它们通常和 Web 的分布式特性没有直接的关系。在文献(Deitel 和 Deitel 2000)中有关于如何建立基于 Web 的应用程序的详尽介绍。

1. 文档模型

Web 的基础是所有信息都以文档的形式表示。表达文档的方式是多种多样的。有些文档像 ASCII 文本文件一样简单,而有些文档由脚本集合来表达,当文档被下载到浏览器时这些脚本会自动执行。

然而,最重要的是文档中可以包含对其他文档的引用,这样的引用称为超链接(hyperlink)。当文档在浏览器中显示时,对其他文档的超链接也会显式地呈现给用户。

于是用户就可以通过在一个链接上单击来选择它。选择一个超链接将把一个获取文档的请求发送到存储该文档的服务器。然后该文档从服务器传送到用户的机器并由浏览器显示出来。新文档可能会代替当前文档,也可能会在一个新的弹出窗口中显示。

大多数 Web 文档通过一种称为超文本标记语言(hyper text markup language, HTML)的特殊语言来表达。称 HTML 为标记语言的意思是,HTML 提供把文档组织为不同部分的关键字。例如,每个 HTML 文档可以分为标题部分和正文部分。HTML 还区分文件头、列表、表格和表单。在文档的特定位置插入图像或动画也是可以的。除了这些结构化元素外,HTML 还提供指示浏览器如何显示文档的各种关键字。例如,有的关键字可以用来选择某种特定的字体或字体大小,有的关键字可以表示文本以斜体或是粗体显示,还有的关键字对部分文本进行对齐排列等。

HTML 已经不仅仅是一种简单的标记语言了。目前,它已包含用于制作出生动的 Web 网页的许多功能。其中最强大的功能之一是可以采用脚本的形式表达文档中的一些部分。例如,考虑图 11.2 所示的 HTML 文档。

```
<HTML>                                <! -- HTML 文档的开始-->
<BODY>                                 <! -- 主体的开始-->
<H1>HelloWorld</H1>                  <! -- 显示的基本文本-->
<P>                                    <! -- 开始新的段落-->
<SCRIPT type = "text/javascript">     <! -- 标识脚本语言-->
    document.writeln("<H1>Hello World</H1>") // 写一行文本
</SCRIPT>                               <! -- 脚本部分的结尾-->
</P>                                    <! -- 段部分的结尾-->
</BODY>                                 <! -- 主体的结尾-->
</HTML>                                <! -- HTML 部分的结尾-->
```

图 11.2 带有一段 JavaScript 脚本的简单 Web 页

当浏览器解释并显示这个 Web 页时,用户将看到在单独的行中文本“Hello World”显示两次。第一次显示是解释下面的 HTML 行的结果:

```
<H1>Hello World </H1>
```

而第二次显示则是运行一段短小的脚本的结果,该脚本用 JavaScript 语言(一种 Java 脚本语言)编写。这段脚本由一行代码组成:

```
document.writeln("<H1>HelloWorld</H1>");
```

尽管两次显示文本的效果是完全相同的,但显然它们之间有重要的区别。第一次是直接解释 HTML 命令来正确显示被标记文本的结果。第二次是执行作为文档的一部分下载到浏览器的一段脚本的结果。换句话说,这是一种移动代码形式。

在内部解析文档时会把它存储为一个有根树,称为一个解析树,其中每个节点代表该文档的一个元素。为了获得可移植性,人们已经对解析树的表示方法进行了标准化。例如,每个节点只能表示一个预定义元素类型集合中的一种元素类型。与此类似,每个节点都被要求实现一个标准接口,该接口包含访问这个节点内容的方法、返回对父节点和子节

点的引用的方法等。这种标准表示方法也称为文档对象模型(document object model, DOM)(le Hors 等 2000)。它也常被称为动态 HTML(dynamic HTML)。

DOM 提供了一个解析 Web 文档的标准编程接口。这个接口用 CORBA IDL 描述，并且到各脚本语言(例如 JavaScript)的映射也已实现了标准化。嵌入文档中的脚本使用这个接口遍历解析树、查看和修改节点、添加和删除节点等。换句话说，脚本可以用来查看和修改它们所在的文档。显然，这为动态调整文档创造了多种可能。

尽管大多数 Web 文档仍然是用 HTML 表达的，但还有另外一种遵循 DOM 的可选语言即 XML，它代表可扩展标记语言(extensible markup language)，可参见文献(Bray 等 2000)。与 HTML 不同，XML 只用于组织文档的结构；它不包含格式化文档(例如使段落居中，或使文本以斜体显示)的关键字。另一个与 HTML 的重要不同是 XML 可以用来随意定义结构。也就是说，它提供定义不同的文档类型(document type)的手段。

定义一个文档类型要求首先声明该文档的元素。例如，图 11.3 显示了一个对期刊文章的简单常规引用的 XML 定义。(行号不是定义的一部分)。第 1 行把一个文章引用声明为包含 3 个元素(title、author 和 journal)的文档。author 元素后面的符号“+”表明有一个或多个作者。

```
(1)      <! ELEMENT article(title, author+, journal)>
(2)      <! ELEMENT title(#PCDATA)>
(3)      <! ELEMENT author(name, affiliation?)>
(4)      <! ELEMENT name(#PCDATA)>
(5)      <! ELEMENT affiliation(#PCDATA)>
(6)      <! ELEMENT journal(jname, volume, number?, month?, pages, year)>
(7)      <! ELEMENT jname(#PCDATA)>
(8)      <! ELEMENT volume(#PCDATA)>
(9)      <! ELEMENT number(#PCDATA)>
(10)     <! ELEMENT month(#PCDATA)>
(11)     <! ELEMENT pages(#PCDATA)>
(12)     <! ELEMENT year(#PCDATA)>
```

图 11.3 引用一篇期刊文章的 XML 定义

在第 2 行中，title 元素被声明为由一系列字符组成(这是通过 XML 的基本 #PCDATA 数据类型指出的)。author 元素又被分为两个其他元素：name 和 affiliation。“?”表明 affiliation 元素是可选的，但在一个文章引用中，一个作者中最多只能有一个 affiliation 元素。与之类似，在第 6 行中，journal 元素也被分为更小的元素。每一个都没有被进一步细分的元素(即在解析树中形成一个叶节点的元素)被指定为一系列字符。

一个 XML 文章引用的实际例子是图 11.4 中所示的 XML 文档(再次强调，行号不是文档的一部分)。假设图 11.3 中的定义存储在一个名为 article.dtd 的文件中，第 2 行告诉 XML 解析器到哪里能够找到与当前文档相关的定义。第 4 行给出了文章的标题，第 5 行和第 6 行分别包含两个作者的姓名。注意，没有包含 affiliation 元素，根据文档的 XML 定义这是允许的。

```
(1)  <? xml = version "1.0">
(2)  <!DOCTYPE article SYSTEM "article.dtd">
(3)  <article>
(4)      <title>Prudent Engineering Practice for Cryptographic Protocols</title>
(5)      <author><name>M. Abadi</name></author>
(6)      <author><name>R. Needham</name></author>
(7)      <journal>
(8)          <jname>IEEE Transactions on Software Engineering</jname>
(9)          <volume>22</volume>
(10)         <number>1</number>
(11)         <month>January</month>
(12)         <pages>6-15</pages>
(13)         <year>1996</year>
(14)     </journal>
(15) </article>
```

图 11.4 一个使用图 11.3 中的 XML 定义的 XML 文档

要把一个 XML 文档显示给用户,需要给出格式化规则。一个简单的方法是把 XML 文档嵌入 HTML 文档中,然后就可以使用 HTML 的关键字进行格式化。作为可选方法,一种称为可扩展样式语言(extensible style language,XSL)的独立的格式化语言可以用来描述 XML 文档的布局。关于如何在实践中使用这种方法的详细介绍请参见文献(Deitel 和 Deitel 2000)。

除了文档传统上所包含的一些元素(例如标题和段落)之外,HTML 和 XML 还支持专门用于多媒体支持的特殊元素。例如,文档中不仅可以包含图像,还可以附带视频片段、音频文件,甚至是交互式的动画。显然,在多媒体文档中脚本也起着重要的作用。

2. 文档类型

除了 HTML 和 XML 之外还有许多文档类型。例如,一段脚本也可以认为是一个文档。其他的例子如用 Postscript 或 PDF 进行格式化的文档、JPEG 或 GIF 图像以及 MP3 格式的音频文档等。文档类型通常以 MIME 类型的形式来表示。MIME 代表多用途 Internet 邮件扩展(multipurpose internet mail extension),最初是为了提供有关作为电子邮件的一部分发送的消息正文内容的信息而开发的。MIME 区分多种类型的消息内容,在文献(Freed 和 Borenstein 1996)中有相关描述。这些类型也用在 WWW 中。

MIME 中的类型分为顶级类型和子类型。图 11.5 中是一些不同的顶级类型,其中包括文本、图像、音频和视频。有一种特殊的 Application 类型,它表示这种类型的文档中包含的数据与一个特定的应用程序相关。实际上,只有这个应用程序才能把文档转换为可以被人理解的内容。

类型	子类型	说明
Text	Plain	未格式化的文本
	HTML	包含 HTML 标记命令的文本
	XML	包含 XML 标记命令的文本
Image	GIF	GIF 格式的静态图像
	JPEG	JPEG 格式的静态图像
Audio	Basic	音频, 8 位 PCM 编码, 采样率为 8000Hz
	Tone	一种特殊的可听的音调
Video	MPEG	MPEG 格式的影片
	Pointer	用于表现的定位设备的表示法
Application	Octet-stream	未解释的字节序列
	Postscript	可打印的 Postscript 格式文档
	PDF	可打印的 PDF 格式文档
Multipart	Mixed	以指定顺序排列的独立部分
	Parallel	必须同时查看的部分

图 11.5 6 种顶级 MIME 类型和一些常见子类型

Multipart 类型用于合成文档。合成文档由多个部分组成, 每个部分还可以有各自的顶级类型。

每种顶级类型可以有多个子类型, 如图 11.5 所示。文档类型由顶级类型和子类型的组合来表示, 例如 application/PDF。这种文档类型说明需要一个独立的应用程序来处理文档, 并且该文档以 PDF 格式表示。通常, Web 页的类型是 text/HTML, 说明它们表示为包含 HTML 关键字的 ASCII 文本。

3. 体系结构概述

HTML 或 XML 与脚本的结合提供了表达文档的一种强大手段。然而, 我们还没有讨论过文档实际上在哪里进行处理和进行什么样的处理。WWW 开始时是一个如前面图 11.1 所示的相对简单的客户-服务器系统。但现在, 这种简单的体系结构已扩展为包含众多组件以支持我们刚刚描述的那些高级文档类型。

对基本体系结构的主要增强之一是通过通用网关接口 (common gateway interface, CGI) 支持简单的用户交互。CGI 定义了 Web 服务器把用户数据作为输入来执行程序的标准方式。通常, 用户数据来自 HTML 表单, 它指定了要在服务器端执行的程序, 以及由用户填写的参数值。一旦完成了表单, 程序的名称和从用户处收集的参数值会被发送到服务器, 如图 11.6 所示。

当服务器收到请求时, 它启动请求中指定的程序, 并把参数值传递给该程序。此时, 程序只是完成它自己的工作, 并且通常以文档的形式返回结果, 该文档会被送回用户的浏览器以供显示。

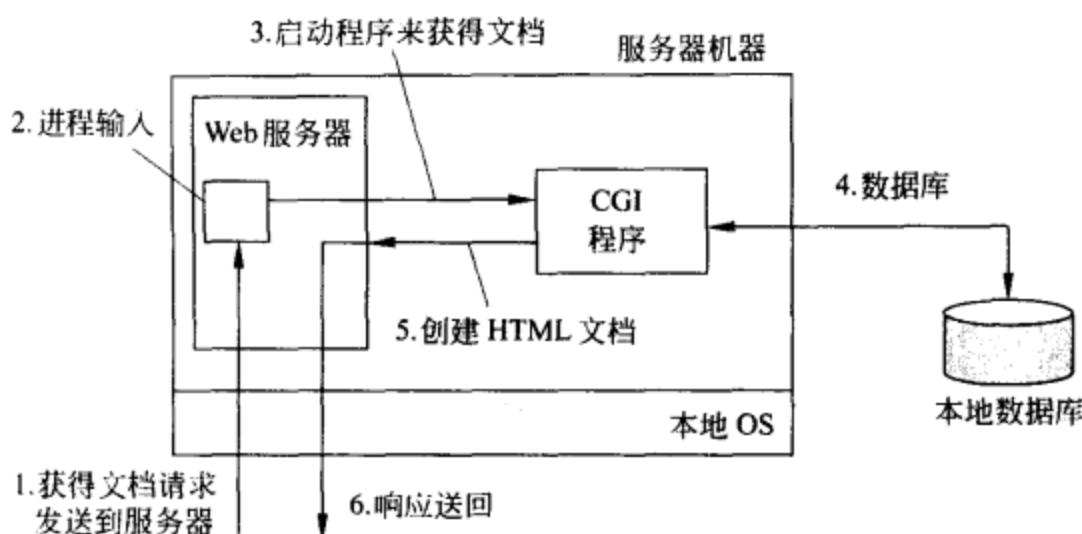


图 11.6 使用服务器端 CGI 程序的原理

只要开发人员需要,CGI 程序可以很复杂。例如像图 11.6 中那样,许多程序对位于 Web 服务器本地的数据库进行操作。在处理完数据后,程序生成 HTML 文档并把它返回给服务器。随后服务器把文档传递给客户。有趣的是,对于服务器而言,这像是它要求 CGI 程序获取文档。也就是说,服务器所做的只是把获取文档的任务委派给一个外部程序。

服务器的主要任务原先只是简单地获取文档来处理客户请求。有了 CGI 程序,服务器只需把获取文档的任务委派给这类程序即可,它不需要知道文档是动态生成的,还是从本地文件系统中实际读出的。然而,今天的服务器除了获取文档之外,还要完成许多其他任务。

最重要的功能增强之一是服务器把获取的文档传递给客户之前还可以对它进行一些处理。特别是,一个文档也许会包含一段服务器端脚本(server-side script)。在从本地获取该文档之后,服务器会执行这段脚本,脚本执行的结果将和文档的其余部分一起发送给客户。但脚本本身并不被发送。也就是说,使用服务器端脚本修改文档实质上是通过用脚本的执行结果替换脚本来完成的。

我们举一个这类脚本的简单例子,考虑图 11.7 所示的 HTML 文档(行号不是文档的一部分)。当客户请求这个文档时,服务器将执行从第 5 行到第 12 行中的脚本。服务器创建一个称为 clientFile 的本地文件对象,并用它以只读方式打开名为 /data/file.txt 的文件。只要文件中有数据,服务器就会把其中的数据作为 HTML 文档的一部分发送给客户。

服务器端脚本通过专门的、非标准化的 HTML 标记 <SERVER> 来识别。来自不同软件制造商的服务器识别服务器端脚本的方法各不相同。在本例中,通过处理脚本,对原始文档进行了修改,其中的服务器端脚本被 /data/file.txt 文件的实际内容所代替。也就是说,客户将不会看到脚本。如果文档中包含客户端脚本,那么它们会像普通内容一样被传递给客户。

```
(1)  <HTML>
(2)  <BODY>
(3)  <P>The current content of <PRE>/data/file.txt</PRE> is: </P>
(4)  <P>
(5)  <SERVER type = "text/javascript">
(6)      clientFile = new File("/data/file.txt");
(7)      if(clientFile. open("r")){
(8)          while(! clientFile. eof())
(9)              document. writeln(clientFile. readln());
(10)         clientFile. close();
(11)     }
(12)   </SERVER>
(13)   </P>
(14)   <P>Thank you for visiting this site.</P>
(15)   </BODY>
(16) </HTML>
```

图 11.7 包含要被服务器执行的 JavaScript 脚本的 HTML 文档

除了执行客户端和服务器端脚本之外,还可以把预编译程序以小程序(applet)的形式传递给客户。通常,applet 是一个短小的独立程序,可以把它发送给客户并在浏览器地址空间中执行它。applet 最常见的形式是已编译为可解释的 Java 字节码的 Java 程序。例如,下面的代码在一个 HTML 文档中包含了一个 Java applet:

```
<OBJECT codetype = "application/Java" classid = "java: welcome.class">
```

applet 是由客户执行的。在服务器端也有 applet 的对应形式,称为 servlet。与 applet 类似,servlet 是一个预编译程序,在服务器的地址空间执行。按照当前的惯例,servlet 大多用 Java 实现,但原则上并没有对使用其他语言的限制。servlet 实现的方法与 HTTP 中的方法相同。HTTP 是客户和服务器之间的标准通信协议,我们在下面会详细介绍该协议。

无论何时服务器收到“瞄准”servlet 的 HTTP 请求,它都调用该 servlet 的与该请求相关联的方法。后者会处理请求,并且一般会以 HTML 文档的形式返回一个响应。servlet 与 CGI 脚本的重要差别在于 CGI 脚本作为独立的进程执行,而 servlet 由服务器执行。

现在我们可以对 Web 上客户和服务器的结构给出一个更完整的体系结构方面的视图。这个结构如图 11.8 所示。无论何时用户发出对一个文档的请求,Web 服务器一般都可以根据该文档的要求,从三种做法中进行选择。第一种做法是可以直接从它的本地文件系统中获取文档;第二种做法是可以启动一个 CGI 程序,这个程序将生成一个文档,其中也许会用到本地数据库的数据;第三种做法是可以把请求传递给一个 servlet。

获取文档之后,也许需要通过执行文档中包含的服务器端脚本以进行一些后处理。实际上,只有从本地文件系统中直接获取的文档,即其中不涉及 servlet 或 CGI 程序的文档需要这样做。然后文档会被传递到用户的浏览器。

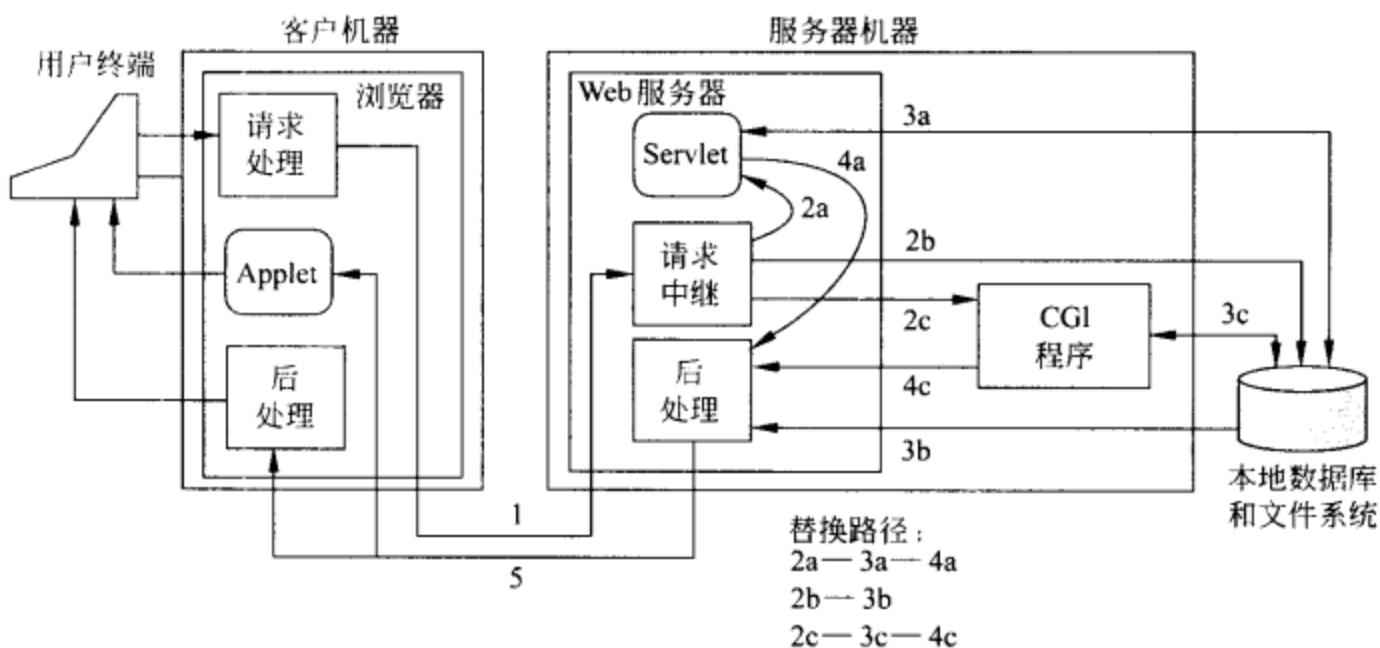


图 11.8 Web 中客户和服务器的详细体系结构

当文档返回到客户后,浏览器将执行任何客户端脚本,并且有可能获取并执行文档中引用的 applet。文档处理的结果是其内容显示在用户的终端上。

在迄今为止讨论的体系结构中,我们一直假设浏览器可以显示 HTML 文档,并有可能还可以显示 XML 文档。然而,还有许多其他格式的文档,例如 Postscript 或 PDF 格式的文档。在这些情况下,浏览器简单地请求服务器从其本地文件系统中获取文档而不做任何干预就返回它。实际上,这将导致返回给客户一个文件。根据文件的扩展名,浏览器启动合适的应用程序来显示或处理其中的内容。因为这些应用程序协助浏览器显示文档,所以它们也被称为助手应用程序(helper application)。后面我们将讨论一种方法来代替使用这类应用程序。

11.1.2 通信

Web 中客户和服务器之间的所有通信都基于超文本传输协议(hypertext transfer protocol,HTTP)。HTTP 是一个相对简单的客户服务器协议;客户向服务器发送一个请求消息并等待一个响应消息。HTTP 的一个重要属性是它是无状态的。也就是说,它没有打开连接的概念,也不要求服务器保存有关客户的信息。文献(Fielding 等 1999)中描述了 HTTP 的最新版本。

1. HTTP 连接

HTTP 基于 TCP。无论何时客户向服务器发起一个请求,它都会建立一个到服务器的 TCP 连接,并通过该连接发送它的请求消息。同一连接也被用来接收响应。通过使用 TCP 作为其底层协议,HTTP 无须考虑请求和响应的丢失问题。客户和服务器简单地假设它们发送的消息能够到达另一端。如果确实出现了错误,例如,连接断开或发生了超时,则会报告一个错误。但是,一般不会尝试从失败中恢复。

第一个 HTTP 版本的问题之一是它使用 TCP 连接的效率很低。每个 Web 文档都

是由位于同一服务器上的多个文件的集合构建而成的。要正确显示一个文档，这些文件也需要传输给客户。原则上，这些文件中的每一个都是一个文档，客户可以向存储它的服务器发出单独的请求来获取它。

在 HTTP 的 1.0 及更早版本中，对一个服务器的每个请求都要求建立一个单独的连接，如图 11.9(a)所示。服务器做出响应之后，连接再次被断开。这种连接被称为是非持久的(nonpersistent)。非持久连接的一个主要缺点是建立 TCP 连接开销比较大。因此，把一个包含所有元素的完整文档发送给客户将花费相当多的时间。

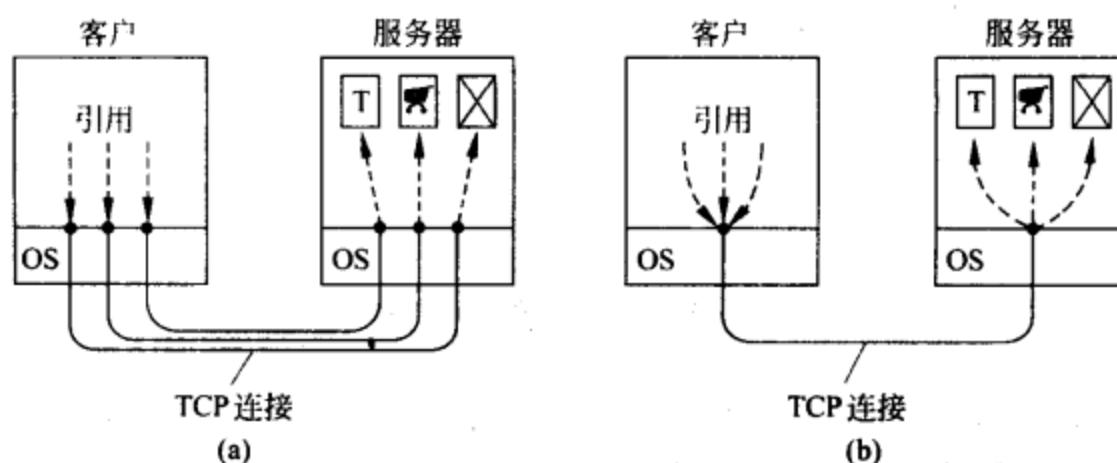


图 11.9 使用非持久连接和持久连接

(a) 使用非持久连接；(b) 使用持久连接

请注意，HTTP 允许一个客户同时和一个服务器建立多个连接。这常常用来抵消建立连接带来的延迟，使数据能够从服务器并行传输到客户。

HTTP 的版本 1.1 采用了一种更好的方法。它使用持久连接(persistent connection)，可以在一个连接中发送多个请求(以及接收它们各自的响应)，而无须为每个请求/响应对建立一个单独的连接。为了进一步改善性能，客户可以在一行中发送多个请求，而不用等待前面请求的响应，这也称为流水线(pipelining)。图 11.9(b)中示意了持久连接的使用。

2. HTTP 方法

HTTP 设计成一个通用的客户-服务器协议，设计目标是在这二者之间实现文档的双向传输。客户可以向服务器请求执行这两种操作中的任一种，方法是把包含所需操作的请求消息发送给服务器。图 11.10 中列出了最常用的请求消息。

操作	说明
head	请求返回一个文档的标题信息
get	请求把一个文档返回给客户
put	请求存储一个文档
post	提供要添加到一个文档(集合)中的数据
delete	请求删除一个文档

图 11.10 HTTP 支持的操作

HTTP 假设每个文档都有与其相关的元数据，并且文档的元数据存储在一个随请求或响应一起发送的单独的 HTML 文件头中。当客户不想得到实际的文档，只想得到与其相关的元数据时，会向服务器提交 head 操作。例如，使用该操作可以返回特定文档被修改的时间，这样可以验证由客户缓存的文档的有效性。还可以使用它检查一个文档是否存在，而无须实际传输文档。

最重要的操作是 get。用这个操作来实际从服务器上获取一个文档，并把它返回给发出请求的客户。可以指定仅当文档在某个特定时间后修改过的情况下才返回这个文档。HTTP 还允许文档具有与之相关的标记(tag，它以字符串表示)，并且仅当文档的标记与某些特定的标记相匹配时才返回这个文档。

put 操作是 get 操作的逆操作。客户可以请求服务器以某个给定的名字(该名字随请求一起发送)存储一个文档。当然，服务器一般不会盲目地执行 put 操作，它只会接受来自授权客户的请求。我们稍后将讨论如何处理这些安全问题。

post 操作与存储文档有些类似，不同之处是客户请求把数据添加到一个文档或文档集合中。一个典型的例子是把一篇文章贴到一个新闻组中。与 put 操作相比，post 的显著特征是，post 操作指出一篇文章应添加到哪个文档组中，而且文章随该请求一起被发送。与之对照，put 操作传送的是一个文档和服务器存储该文档所使用的名字。

最后，delete 操作用于请求服务器删除在发送给它的消息中指定的文档。删除操作是否真正发生仍然要取决于各种安全措施。甚至有可能服务器本身也没有删除指定文档的适当权限。毕竟，服务器只是一个用户进程。

3. HTTP 消息

客户和服务器之间的所有通信都是通过消息完成的。HTTP 只识别请求消息和响应消息。一个请求消息由三部分组成，如图 11.11(a) 所示。请求行(request line)是必不可少的，它标识出客户希望服务器执行的操作，并且指明与该请求相关的文档。使用一个单独的字段来标识客户期望使用的 HTTP 版本。后面我们会解释附加的消息报头。

响应消息以一个状态行(status line)开始，该行包含一个版本号和一个三位数字的状态编码，如图 11.11(b) 所示。作为状态行的一部分发送的一个文本短语用来简要地解释编码的意义。例如，状态编码 200 表示一个请求已得到准予，与它相关的文本短语是“OK”。其他常用的编码还有：

400(错误的请求)

403(禁止)

404(未找到)

请求或响应消息中也许会包含其他报头信息。例如，如果客户对一个只读文档请求了 post 操作，那么服务器的响应消息中会包含状态编码 405(方法不允许使用)，并且包含一个指明允许的操作(例如，head 和 get)的 Allow 报头。另一个例子是，可能仅当一个文档在某个时刻 T 之后被修改过，客户才对它有兴趣。这种情况下，客户的 get 请求可以带有指定值 T 的 If-Modified-Since 报头作为其参数。

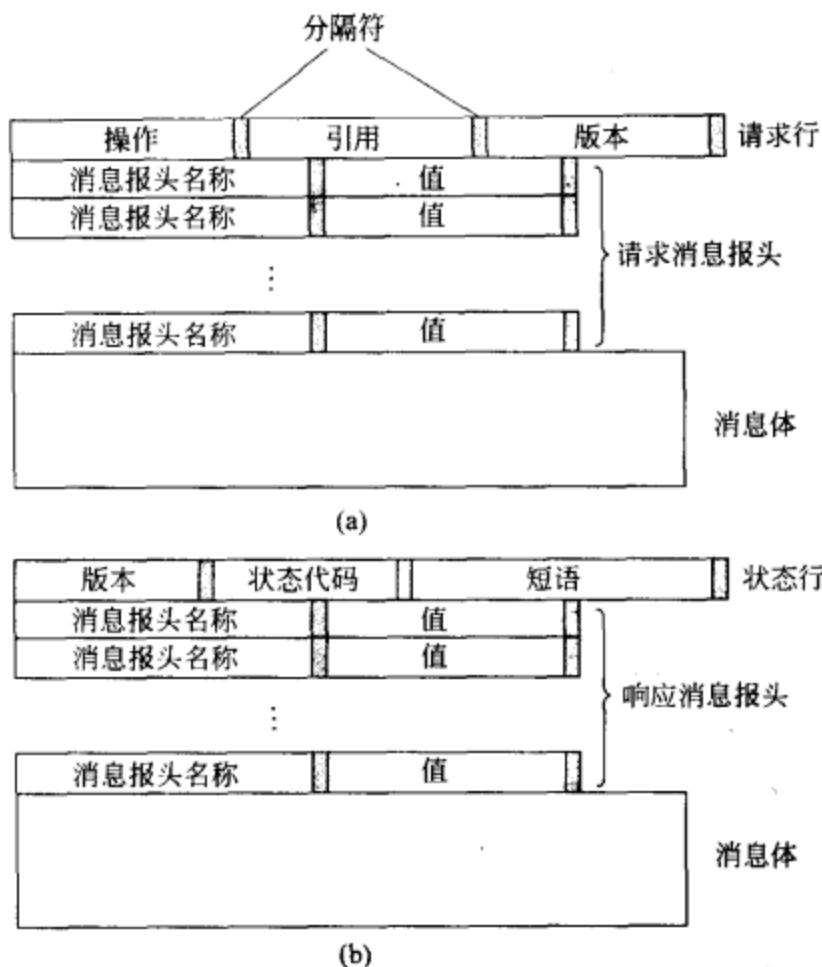


图 11.11 HTTP 请求消息和响应消息

(a) HTTP 请求消息; (b) HTTP 响应消息

报头	来源	内容
Accept	客户	客户可以处理的文档类型
Accept-Charset	客户	客户可以接受的字符集
Accept-Encoding	客户	客户可以处理的文档编码
Accept-Language	客户	客户可以处理的自然语言
Authorization	客户	客户证书列表
WWW-Authenticate	服务器	客户应响应的安全口令
Date	二者	发送消息的日期和时间
ETag	服务器	与返回的文档相关的标记
Expires	服务器	响应保持有效的时间
From	客户	客户的电子邮件地址
Host	客户	文档的服务器的 TCP 地址
If-Match	客户	文档应具有的标记
If-None-Match	客户	文档不应具有的标记
If-Modified-Since	客户	告诉服务器仅当文档在指定时间后被修改过时才返回该文档
If-Unmodified-Since	客户	告诉服务器仅当文档在指定时间后未被修改过时才返回该文档
Last-Modified	服务器	返回的文档上次被修改的时间
Location	服务器	客户应把请求重定向到的文档引用
Referer	客户	指向客户最近请求过的文档
Upgrade	二者	发送者希望转而使用的应用协议
Warning	二者	消息中数据状态的信息

图 11.12 一些 HTTP 报头

图 11.12 显示了一些可以随请求或响应发送的合法报头。大多数报头都是顾名思义的，因此我们不进行逐一讨论。

客户可以向服务器发送不同的报头来说明它所能够接受的响应。例如，客户也许可以接受使用在大多数 Windows 和 UNIX 机器上都可用的 gzip 压缩工具压缩的响应信息。这时，客户将随请求发送一个 Accept-Encoding 报头，其内容是“Accept-Encoding: gzip”。与此类似，Accept 报头可以用来指定只可以返回 HTML 格式的 Web 页。

有两个报头是有关安全方面的，但我们本节稍后会讨论到，Web 的安全性通常由一个单独的传输层协议来处理。

Location 和 Referer 报头用于把客户重定向(redirect)到另一个文档。重定向相当于使用前向指针(forwarding pointer)来定位文档，正如第 4 章所讨论的。当客户发出一个对文档 D 的请求时，服务器的响应中也许会含有一个 Location 报头，指定客户应该重新请求文档 D'。在使用对 D' 的引用时，客户可以添加一个 Referer 报头，其中包含对 D 的引用以指示导致重定向的原因。一般地，这个报头用来表示客户最近请求的文档。

Upgrade 报头用于切换到另一种协议。例如，客户和服务器可能在开始时使用普通的 HTTP/1.1 来建立连接。然后服务器可能立即在响应中通知客户它希望使用一个安全的 HTTP 版本继续通信，例如使用 SHTTP(Rescorla 和 Schiffman 1999)。这种情况下，服务器可以发送一个内容是“Upgrade: SHTTP”的 Upgrade 报头。

11.1.3 进程

实质上，Web 仅使用两类进程：一类是浏览器，用户使用它可以访问 Web 文档并把文档显示在本地屏幕上；另一类是 Web 服务器，它对浏览器的请求进行响应。如前所述，浏览器可以由助手应用程序协助完成任务。与此类似，服务器也可以带有其他程序，例如 CGI 脚本。下面我们详细介绍 Web 中使用的典型的客户端和服务器端软件。

1. 客户

最重要的 Web 客户是称为 Web 浏览器(Web browser)的软件，它通过从服务器获取 Web 页并把它们显示在用户的屏幕上允许用户访问这些页面。浏览器一般提供带有超链接的界面，用户只需在一个超链接上单击鼠标就可以选中这个超链接。

原则上，Web 浏览器是一个单一的程序。然而，由于浏览器需要能够处理多种文档类型并要向用户提供使用简单的界面，所以它们通常是复杂的合成软件。

Web 浏览器的设计者需要面对的一个问题是浏览器应该很容易扩展，以使它在原则上可以支持服务器返回的任何类型的文档。大多数浏览器采用的方法是提供一种称为插件的功能。插件(plug-in)是可以动态载入浏览器以便处理某种特定类型文档的小型程序。文档类型一般作为 MIME 类型给出。插件应该在本地可用，有可能需要事先由用户专门从远程服务器传输到本地。

插件向浏览器提供一个标准的接口，并且也希望浏览器为它提供一个标准接口，如图 11.13 所示。当浏览器遇到一个需要插件进行处理的文档类型时，它把本地的插件加载进来并为之创建一个实例。在初始化完毕后，与浏览器其余部分的交互操作就针对插件，

但只有标准接口中的方法才能使用。当不再需要一个插件时,会把它从浏览器中删除。

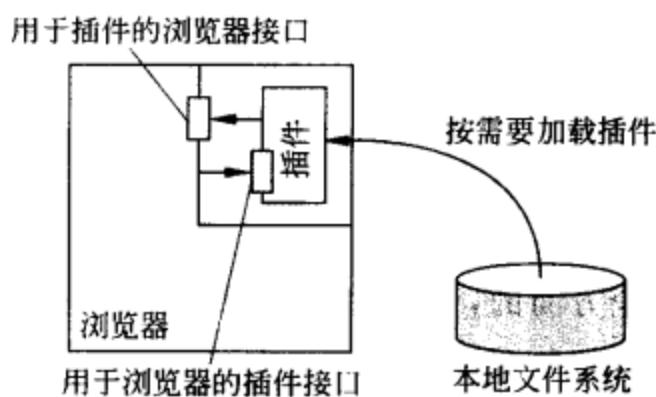


图 11.13 在 Web 浏览器中使用插件

另一个常用的客户端进程是 Web 代理 (Web proxy), 参见文献 (Luotonen 和 Altis 1994)。起初, 这种进程用于允许浏览器处理 HTTP 之外的应用级协议, 如图 11.14 所示。例如, 要从一个 FTP 服务器传输文件, 浏览器可以发送一个 HTTP 请求给本地 FTP 代理, 后者将获取文件并把它嵌入在一个 HTTP 响应消息中返回给浏览器。

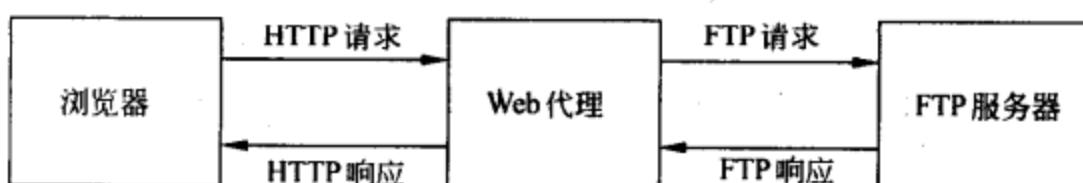


图 11.14 浏览器不支持 FTP 时使用 Web 代理

现在, 大多数 Web 浏览器都能够支持多种协议, 因而不再需要使用代理。然而, Web 代理仍然被广泛使用, 不过是出于一个完全不同的原因, 那就是提供为多个浏览器所共享的缓存。我们下面会讨论, 当请求一个 Web 文档时, 浏览器可以把请求传递给一个本地 Web 代理。这个代理在与存储文档的远程服务器联系之前会查看本地缓存中是否已有被请求的文档。

2. 服务器

正如我们已解释的, Web 服务器是一个处理输入的 HTTP 请求的程序, 它获取被请求的文档并把该文档返回给客户。为了给出一个具体的例子, 我们简要地介绍 Apache 服务器 (UNIX 平台上重要的 Web 服务器) 的总体结构。

图 11.15 是 Apache Web 服务器的总体结构。服务器由若干模块组成, 这些模块由一个核心模块所控制。核心模块接受输入的 HTTP 请求, 然后以一种流水线的方式把请求传递给其他模块。也就是说, 核心模块决定着处理请求的控制流。

对于每个输入的请求, 核心模块为它分配一个请求记录, 其中的字段用来保存 HTTP 请求中包含的文档引用、相关的 HTTP 请求报头、HTTP 响应报头等。每个模块通过读取和修改适当的字段来操作记录。最后, 当所有模块均已完成对请求的处理时, 最后的模块把被请求的文档返回给客户。注意, 原则上, 每个请求可以有它自己的流水线。

Apache 服务器是高度可配置的, 它可以包含众多协助处理输入的 HTTP 请求的模

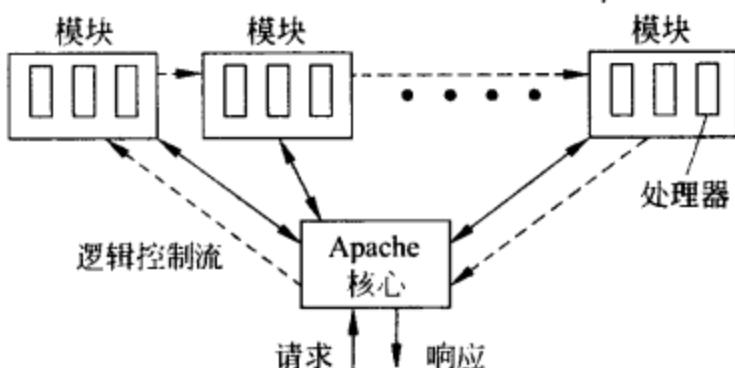


图 11.15 Apache Web 服务器的总体结构

块。为了支持这种灵活性,它采用了下述方法。要求每个模块提供可以由核心模块调用的一个或多个处理器(handler)。这些处理器都接受指向一个请求记录的指针作为它们惟一的输入参数。这些处理器的相似之处还在于它们都可以读取和修改请求记录中的字段。

为了在正确的时机调用适当的处理器,HTTP 请求的处理被分解为几个阶段。一个模块可以在某个特定阶段中注册一个处理器。当到达一个阶段时,核心模块查看该阶段中已注册了哪些处理器,然后调用这些处理器,我们很快将讨论这个过程。下面列出了各个阶段:

- (1) 把文档引用解析为一个本地文件名。
- (2) 客户身份验证。
- (3) 客户访问控制。
- (4) 请求访问控制。
- (5) 确定响应的 MIME 类型。
- (6) 处理其余事务的一般阶段。
- (7) 响应的传输。
- (8) 把请求处理过程记入日志。

引用以统一资源标识符(uniform resource identifier, URI)的形式给出,稍后我们将讨论 URI。解析 URI 的方法有许多种。大多数情况下,被引用文档的本地文件名是 URI 的一部分。但是,当文档需要生成时,例如使用 CGI 程序的情况下,URI 将包含服务器应当启动的程序以及任何输入参数值。

第二个阶段包含对客户的身份验证。原则上,这个阶段仅验证客户的身份,而不检查客户的访问权限。

客户访问控制负责检查客户是否真正为服务器所知,并且是否具有访问权限。例如,在这个阶段中,可能会发现客户属于某个用户组。

请求访问控制阶段检查客户是否拥有对被引用文档执行所发出请求的权限。

下面的这个阶段负责确定文档的 MIME 类型。可以通过多种方法确定文档类型,例如通过使用文件扩展名,或使用一个包含文档的 MIME 类型的单独文件。

不属于前面任何阶段的其他请求处理在一个一般阶段中完成。例如,检查一个被返回引用的语法,或建立用户的配置文件(profile),这些都是在该阶段处理的典型事务。

最后,将把响应传输给用户。返回结果的方法也有很多种,这取决于请求的内容和响应的生成方式。

最后一个阶段把处理请求时采取的各个行动记入日志。这些日志可用于多种用途。

为了处理这些阶段,核心模块维护一个在各阶段中注册的处理器的列表。它负责选取一个处理器并简单地调用它。处理器可以拒绝请求、处理请求,或是报告一个错误。当一个处理器拒绝一个请求时,它会明确声明自己不能处理该请求,这样核心模块将需要选取在当前阶段注册的另一个处理器。如果可以处理请求,通常会启动下一个阶段。当报告了一个错误时,请求处理将被中断,一个错误消息将返回给客户。

Apache 服务器实际包含哪些模块是在配置时确定的。最简单的情形是核心模块必须执行全部请求处理过程。这时,服务器实际上简化为一个只能处理 HTML 文档的简单的 Web 服务器。为了能够在同一时间内处理多个请求,核心模块通常会为每一个输入的请求启动一个新进程。进程的最大数目也是可以配置的参数。有关 Apache 服务器的配置与编程的详细内容请参见文献(Laurie 和 Laurie 1999)。

3. 服务器簇

与 Web 的客户-服务器特性有关的一个重要问题是 Web 服务器很容易超载。为许多设计所采用的一个实用解决方案是简单地把一个服务器复制到一个工作站簇中,并使用一个前端把客户的请求重定向到这些副本服务器之一(Fox 等 1997)。图 11.16 示意了这个原理。这是我们在第 1 章中讨论的水平分布的一个例子。

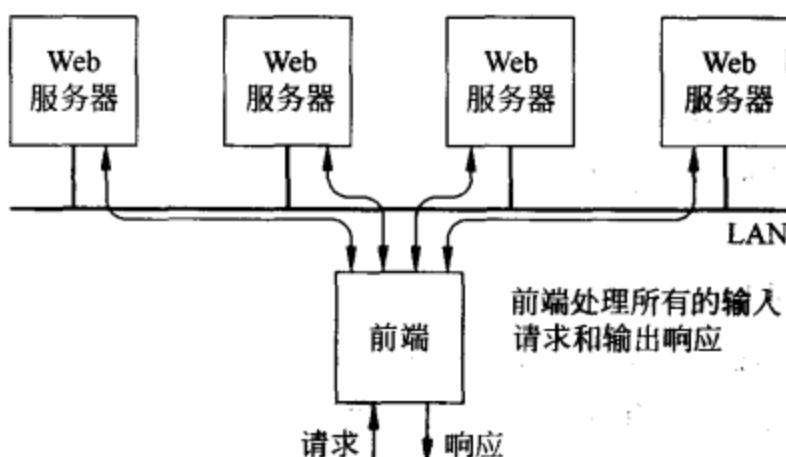


图 11.16 使用工作站簇实现 Web 服务的原理

这种结构的一个关键问题是前端的设计,因为前端很容易成为一个严重的性能瓶颈。通常,作为传输层交换器运行的前端和运行在应用级别的前端有很大的区别。

正如我们所提到过的,无论何时客户发出一个 HTTP 请求,它都需要和服务器建立一个 TCP 连接。一个传输层交换器根据对服务器负载的某种衡量,简单地把通过 TCP 连接发送的数据传递给服务器之一。这种方法的主要缺点是交换器不能考虑通过 TCP 连接发送的 HTTP 请求的内容。它只能基于服务器负载情况做出重定向决定。

一般来说,更好的方法是采用感知内容的请求分布(content-aware request distribution)。使用这种方法,前端首先查看输入的 HTTP 请求,然后决定应当把该请求转发给哪个服务器。这种方法可以与在服务器簇中分布内容相结合。参见在文献(Yang

和 Luo 2000)中的描述。

感知内容的分布方法有许多优点。例如,如果前端总是把对相同文档的请求转发给同一个的服务器,那么这个服务器就能有效地缓存这个文档,从而缩短响应时间。而且,还可以把文档集合分布在各个服务器上,而不是在每个服务器上复制每个文档。这种方法可以更有效地利用可用存储容量,并且允许使用专门的服务器处理特殊的文档(例如音频或视频文档)。

感知内容分布方式的一个问题是前端需要做很多工作。为了改进性能,Pai 等(1998)引入了一种机制;通过这种机制,到前端的 TCP 连接移交给一个服务器。这样,服务器将直接对客户发出响应,前端无须再做任何干预,如图 11.17(a)所示。TCP 连接的移交对客户是完全透明的;客户将把它的 TCP 消息发送给前端(包括确认消息等),但将从连接移交给的服务器那里接收消息。

将前端工作的分布与传输层交换相结合可以进一步改进性能,在文献(Aron 等 2000)中有相关讨论。在与 TCP 移交结合的情况下,前端有两个任务。首先,当一个请求第一次到来时,前端必须决定由哪个服务器来处理与客户的剩余通信。其次,前端应该转发与被移交的 TCP 连接相关联的客户的 TCP 消息。

这两个任务的分布如图 11.17(b)所示。调度器(dispatcher)负责决定应该把一个 TCP 连接转交给哪个服务器;分发器(distributor)监视一个被移交的连接中的到来的

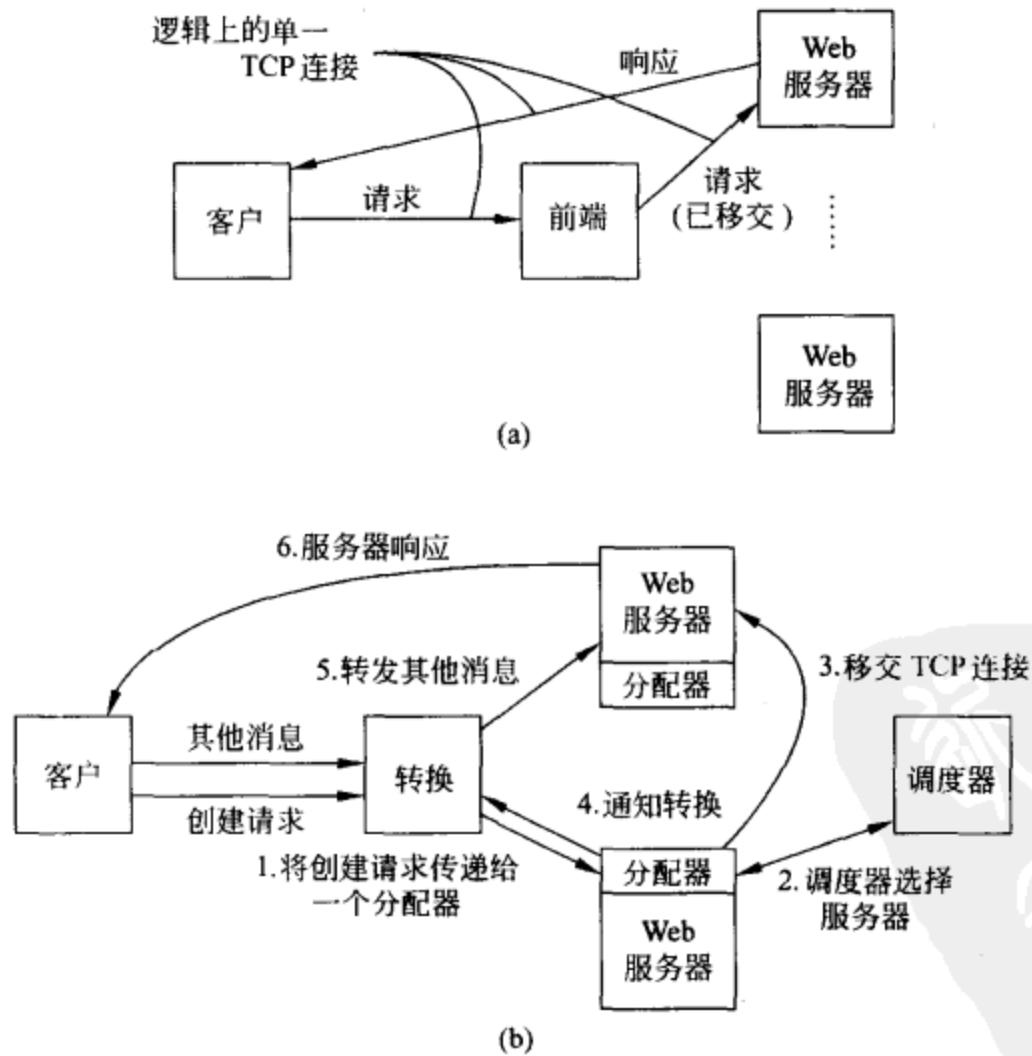


图 11.17 TCP 移交和 Web 服务器簇

(a) TCP 移交的原理; (b) 一个可扩展的内容感知的 Web 服务器簇

TCP 流量。交换器用来把 TCP 消息转发给分发器。当一个客户首次联系 Web 服务时，它的 TCP 连接建立消息转发给一个分发器；分发器继而与调度器联系，由后者决定应把该连接移交给哪个服务器。然后，交换器会得到通知，把该连接的所有后续 TCP 消息发送给选中的服务器。

11.1.4 命名

Web 使用一个单一的命名方案来确定文档，该方案称为统一资源标识符(uniform resource identifiers, URI)(Berners-Lee 等 1998)。URI 有两种形式。一种 URI 是统一资源定位符(uniform resource locator, URL)，它通过包含如何及在何处访问文档的信息来标识一个文档。相比之下，统一资源名称(uniform resource name, URN)更像第 4 章中讨论的真正的标识符。URN 用来作为对文档的全球惟一的、与位置无关的可靠引用。

一个 URI 的实际语法由与其相关的方案(scheme)决定。方案名是 URI 的一部分。已定义的方案有许多种，下面我们将介绍其中的一些方案及相关的 URI 例子。最著名的方案是 http，但它不是惟一的方案。

1. 统一资源定位符

URL 中包含有关如何及在何处访问一个文档的信息。访问文档的方法通常由作为 URL 一部分的方案名(例如 http、ftp 或 telnet)来反映。文档所在的位置通常嵌入在 URL 中，由可以向其发送访问请求的服务器的 DNS 名字指定，也可以使用 IP 地址。服务器用来监听访问请求的端口号也是 URL 的一部分，如果这部分被省略，则使用默认端口。最后，URL 还包含服务器查找文档时使用的文档名。URL 的一般结构如图 11.18 所示。

方案	主机名	路径名
http	:// www.cs.vu.nl /home/steen/mbox	

(a)

方案	主机名	端口	路径名
http	:// www.cs.vu.nl : 80 /home/steen/mbox		

(b)

方案	主机名	端口	路径名
http	:// 130.37.24.11 : 80 /home/steen/mbox		

(c)

图 11.18 URL 的常用结构

(a) 只使用 DNS 名；(b) 使用 DNS 名和端口号；(c) 使用 IP 地址和端口号

解析图 11.18 中所示的 URL 是很简单的。如果使用 DNS 名指定服务器，那么需要把 DNS 名解析为服务器的 IP 地址。利用 URL 中包含的端口号，客户可以使用由方案指定的协议来联系服务器，然后向服务器传递 URL 最后一部分中的文档名。

图 11.19 中列出的是 URL 的一些例子。如前所述，使用 http 方案的 URL 用于使用

HTTP 传输文档。与此类似,使用 ftp 方案的 URL 用于使用 FTP 传输文件。

使用 data 方案的 URL(Masinter 1998)支持直接的文档形式。在这样的 URL 中,文档本身嵌入在 URL 中,这类似于把文件的数据嵌入在一个 inode (Mullender 和 Tanenbaum 1984)中。示例 URL 中包含了代表希腊字符串 $\alpha\beta\gamma$ 的纯文本。

URL 除了用来指向一个文档外,还常常用于其他用途。例如,telnet URL 可以用来和服务器建立一个 telnet 会话。还有用于基于电话通信的 URL,在文献(Vaha-sipila 2000)中有相关描述。tel URL 如图 11.19 中例子所示,它实质上只嵌入了一个电话号码,并简单地让客户通过电话网建立一次通话。这种情况下的客户通常是手持电话之类的设备。modem URL 可以用来与另一台计算机建立基于调制解调器的连接。示例 URL 指定远程调制解调器应该遵守 ITU-T V32 标准。

名称	用途	示例
http	HTTP	http://www.cs.vu.nl:80/globe
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/READme
file	本地文件	file:/edu/book/work/chp/11/11
data	内联(inline)数据	data:text/plain;charset=iso-8859-7,%e1%e2%e3
telnet	远程登录	telnet://flits.cs.vu.nl
tel	电话	tel:+31201234567
modem	调制解调器	modem:+31201234567;type=v32

图 11.19 URL 示例

2. 统一资源名称

与 URL 不同,URN 是位置无关的对文档的引用。URN 的结构如图 11.20 所示,由三个部分组成(Moats 1997)。请注意,URN 构成了所有 URI 中使用 urn 方案的子集。而名字空间标识符决定了 URN 的第三部分的语法规则。

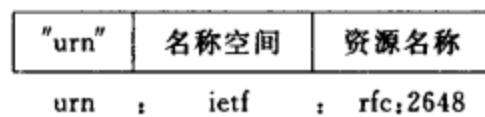


图 11.20 URN 的总体结构

URN 的一个典型例子是用书籍的 ISBN 来标识它们,例如 urn:isbn:0-13-349945-6 (Lynch 等 1998);或用于标识 IETF 文档的 URN,例如 urn:ietf:rfc:2648 (Moats 1999)。IETF 代表 Internet 工程任务组(Internet Engineering Task Force),它是负责开发 Internet 标准的组织。

尽管定义一个 URN 名字空间比较简单,但解析 URN 却困难得多,因为从不同名字空间派生出的 URN 可能会有相差很大的结构。因此,将需要为每个名字空间分别引入解析机制。不幸的是,当前设计的 URN 名字空间中还没有一个提供这样的解析机制。

11.1.5 同步

对于 Web 而言,同步不算是一个问题,这主要出于两个原因。第一,Web 采用严格的客户-服务器结构,服务器之间(或客户之间)从不相互交换信息,所以不会发生同步问题。第二,可以把 Web 看作是一个大多数时候进行读操作的系统。更新的操作一般由单个人完成,因此很少会有写-写冲突发生。

然而,这种情况正在发生变化,越来越多的应用要求对 Web 文档的协同创作提供支持。也就是说,Web 应当为由一组协同工作的用户或进程对文档的并发更新提供支持。

为此已提出了一种 HTTP 的扩展协议,称为 WebDAV (Whitehead 和 Wiggins 1998)。WebDAV 代表 Web 分布式创作和版本控制 (Web distributed authoring and versioning),它提供一种简单的手段来锁定共享文档,可以在远程 Web 服务器上创建、删除、复制和移动文档。下面我们简要介绍 WebDAV 中支持的同步。关于其他功能的更多信息可以在文献 (Whitehead 和 Golan 1999) 中找到。WebDAV 的详细说明请参见 (Golan 等 1999)。

为了同步对一个共享文档的并发访问,WebDAV 支持一种简单的锁定机制。有两类写入锁定。独占写入锁定可以赋予一个客户,它将阻止任何其他客户在共享文档被锁定时修改该文档。另一类是共享写入锁定,它允许多个客户同时更新文档。因为锁定发生在整个文档的粒度上,所以共享写入锁定对于多个客户修改同一文档的不同部分的情况很方便。但是此时,需要客户自己小心避免发生写-写冲突。

对客户赋予一个锁定通过向该客户传递一个锁定令牌完成。服务器中注册着哪个客户当前具有锁定令牌。无论何时客户希望修改文档,它会向服务器发送一个 HTTP 的 post 请求,并随请求发送锁定令牌。这个令牌说明该客户可以对文档进行写访问,因而服务器将执行该请求。

一个重要的设计问题是在客户持有锁定令牌期间无须在客户和服务器之间维护一个连接。客户在获取锁定令牌之后可以简单地和服务器断开连接,并在发送 HTTP 请求时重新连接服务器。

注意,当一个持有锁定令牌的客户崩溃时,服务器必须能以某种方式收回锁定。WebDAV 没有指定服务器如何处理这类问题,而是把它留给了具体的实现。其原因是最佳解决方法随 WebDAV 所应用于的文档类型不同而不同。采用这种方法的原因是没有圆满地解决孤儿锁定问题的通用方法。

11.1.6 缓存和复制

为了改进 Web 的性能,客户端缓存在 Web 客户的设计中一直并将继续发挥重要的作用(请参见 Barish 和 Obraczka 2000)。另外,为了缓解负载严重的 Web 服务器的压力,通常的做法是对 Web 站点进行复制并将其拷贝分布到 Internet 上。最近一段时间,兴起了复杂的 Web 复制技术,而缓存技术正在渐渐失去它的普及性。下面我们详细论述这些问题。

1. Web 代理缓存

客户端缓存一般有两种形式。首先,大多数浏览器都带有简单的缓存功能。无论何时获取了一个文档,它都会被存储在浏览器的缓存中,以供下一次请求该文档时载入。客户一般可以通过指定何时进行一致性检查来对缓存进行配置,下面论述一般情况时将对此给出解释。

其次,客户站点通常运行一个 Web 代理。如前所述,Web 代理接受本地客户的请求并将其传递给 Web 服务器。响应到来后,其结果被传递给客户。这种方法的优点是代理可以对结果进行缓存,并且在需要时可以把该结果返回给另一个客户。换句话说,Web 代理可以实现共享缓存。

除了在浏览器和代理处进行缓存之外,还可以设置覆盖一个地区,甚至一个国家的缓存,这就导致了层次性的缓存模式的出现。这种模式主要用于减少网络流量,但缺点是它的延迟时间高于非层次性缓存模式的延迟时间。造成高延迟的原因是可能需要联系多个缓存服务器,而在非层次性缓存模式中最多只需联系一个服务器。

Web 中使用了不同的缓存一致性协议。为了保证从缓存返回的文档是一致的,一些 Web 代理会先向服务器发送一个有条件的 HTTP 的 get 请求,其中附带的 If-Modified-Since 请求报头中指定缓存中的文档的最后一次修改时间。仅当文档在那个时间之后被更改过时,服务器才会返回整个文档。否则,Web 代理只是简单地将其缓存中的文档版本返回给发出请求的本地客户。根据第 6 章中引入的术语,这对应于一个基于拉的协议。

不幸的是,这种策略要求代理为每一个请求联系服务器。为了用较弱的一致性换取性能的改进,广为采用的 Squid Web 代理(Chankhunthod 等 1996)根据文档被缓存的时间距离它最后一次被修改的时间有多久来决定文档的到期时间 T_{expire} 。具体地说,如果 $T_{\text{last_modified}}$ 是文档最后被修改的时间(由其所有者进行记录),并且 T_{cached} 是它被缓存的时间,那么:

$$T_{\text{expire}} = \alpha(T_{\text{cached}} - T_{\text{last_modified}}) + T_{\text{cached}}$$

其中 $\alpha=0.2$ (这个值是根据实践经验得出的)。在时间 T_{expire} 之前,认为文档有效,代理不会与服务器联系。在时间到期之后,代理请求服务器发送一个最新的拷贝,除非文档没有被修改过。也就是说,当 $\alpha=0$ 时,这个策略就是我们刚刚讨论过的那种方法。

请注意,有很长时间未被修改的文档不会像不久前刚修改过的文档那样快地被检查。这个策略最初是针对 Alex 文件系统提出的(Cate 1992)。它的明显缺陷是代理可能会返回一个无效的文档,即一个比服务器中存储的当前版本更旧的文档。更糟糕的是,客户没有办法检测到自己收到的是一个过时的文档。

基于拉的协议的一种替代方法是让服务器通过发送失效信息来通知代理文档已被修改。这种针对 Web 代理的方法的问题是服务器可能需要保持大量代理的记录,因而不可避免地导致可扩展性的问题。然而,通过结合使用租用和失效,Cao 和 Liu(1998)证明服务器需要维护的状态信息可以被控制在可以接受的限度内。但是,Web 代理缓存的失效协议目前还很少付诸应用。

Web 代理缓存的设计者要面对的问题之一是,只有当文档确实为不同客户所共用

时,代理处的缓存才有意义。在实践中,这意味着缓存的命中率应达到 50%左右,并且缓存还必须能够非常大。构建容量很大的能为众多客户提供服务的虚拟缓存的一种方法是使用协作缓存,其原理如图 11.21 所示。

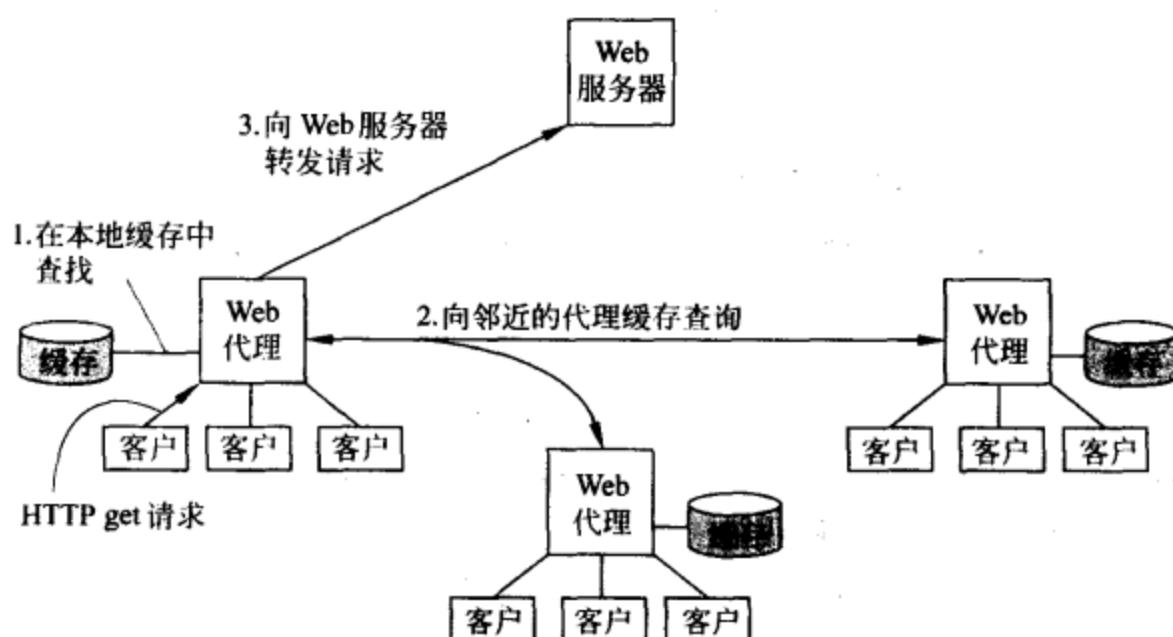


图 11.21 协作缓存的原理

在协作缓存(cooperative caching)中,只要 Web 代理的缓存未能命中,该代理首先检查它邻近的代理中是否包含被请求的文档。如果检查失败,该代理再把请求转发给文档所在的服务器(请参见 Tewari 等 1999)。Wolman 等(1999)的一项研究表明协作缓存对于较小的客户团体(数万个用户)可能是有效的。然而,这样的团体也可能会使用一个单一的代理缓存,因为这样在通信和资源的使用上更节省。

Web 代理缓存的另一个问题是它们只能用于存储静态文档。静态文档不是那种应客户的请求而由 Web 服务器动态生成的文档。来自一个客户的相同请求可能下次会得到不同的响应,从这个意义上说,静态文档是惟一的。

在某些情况下,可以把文档的动态生成从服务器转移到代理缓存。活动缓存(active cache)就使用这种方法(Cao 等 1998)。在这种方法中,当获取一个正常生成的文档时,服务器返回一个要被代理缓存的 applet(小程序)。然后由代理负责通过运行这个 applet 来生成对请求客户的响应。applet 自身被保存在缓存中,当相同的请求再次发出时,它会再次被执行。注意 applet 也可能过时,因为服务器可能会修改它。

活动缓存有多种应用。例如,当一个生成的文档非常大时,可以使用缓存的 applet 来请求服务器只发送下次需要重新生成的文档的不同之处。

作为另一个例子,我们来考虑包含一个广告条列表的文档。每次请求该文档时,只有列表中的一个广告条显示。这种情况下,第一次请求文档时,服务器向代理发送这个文档、完整的广告条列表,以及一个选择广告条的 applet。当这个文档再次被请求时,代理就可以自己处理请求,生成一个响应,但这次会返回一个不同的广告条以供显示。

2. 服务器复制

Web 中的服务器复制通常有两种形式。首先,像我们前面解释的那样,负载沉重的 Web 站点使用 Web 服务器簇来减少响应时间。通常,这类复制对客户是透明的。其次,有一种广为使用的不透明的复制形式,它令 Web 站点的一个完整拷贝在一个不同的服务器上可用。这种方法也称为镜像(mirroring)。于是,客户可以从多个服务器中选择一个来访问 Web 站点。

最近,第三种复制形式正在逐渐获得广泛的应用,它遵循我们在第 6 章讨论的服务器发起的副本放置策略。在这种方法中,分布在 Internet 上的一个服务器集合提供宿主 Web 文档的功能,方法是根据客户访问的特征在服务器之间复制文档。这个服务器集合称为内容分发网络(content delivery network,CDN),有时也称为内容分布网络。

在一个 CDN 中复制和分布文档的技术有许多种。第 6 章中我们曾给出了一个在 RaDaR 的 Web 宿主服务(Rabinovich 和 Aggarwal 1999)中使用的方案。在 RaDaR 中,一个服务器记录着来自一个特定区域的客户的请求次数。就像请求来自该区域的一个 RaDaR 服务器那样衡量这些请求,然后会决定向这个区域服务器移植或复制文档。另一种很受欢迎但较简单的方法由 Akamai 实现,其工作原理大致如下,另请参见文献(Leighton 和 Lewin 2000)。

其基本思想是每个 Web 文档由一个主要的 HTML 页和嵌入在该页中的多个其他文档(例如图像、视频和音频文档)组成。要显示整个文档,用户的浏览器也需要获取这些嵌入的文档。可以假设这些嵌入的文档很少改变,因此对它们进行缓存或复制是有意义的。

每个嵌入文档通常由一个与图 11.18 中的例子类似的 URL 引用。在 Akamai CDN 中,把这样的 URL 修改为指向一个虚拟 ghost(virtual ghost),即指向 CDN 中的一个实际的服务器。下面介绍修改后的 URL 的解析过程,同时图 11.22 中也示意了这个过程。

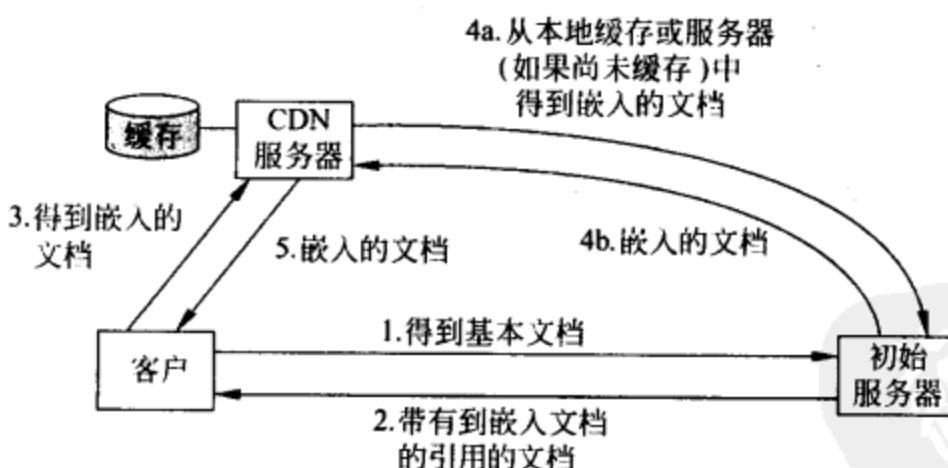


图 11.22 Akamai CDN 的工作原理

虚拟 ghost 的名字包含 DNS 名字 ghosting.corn,它被常规 DNS 名字系统解析为一个 CDN DNS 服务器,该服务器是与请求解析这个 DNS 名字的客户最接近的服务器。最近的 CDN DNS 服务器的选择是根据客户的 IP 地址做出的,客户的 IP 地址放在一个包含 Internet“地图”的数据库中。每个 CDN DNS 服务器中记录着与它接近的普通 CDN

服务器。CDN DNS 服务器继续进行名字解析,根据诸如当前负载这样的选择标准,返回与它接近的普通 CDN 服务器之一的地址。

最后,客户把对嵌入文档的请求转发给选中的 CDN 服务器。如果这个服务器没有所需的文档,它就从起始的 Web 服务器(如图 11.22 的步骤 4 所示)那里获取该文档,在本地缓存它,然后把它传递给客户。如果文档已经在这个 CDN 服务器的缓存中,它会被立即返回给客户。

对于 Akamai CDN 或任何其他 CDN 来说,如何定位一个邻近的 CDN 服务器是一个重要的问题。一个可能的方法是使用第 4 章介绍的一种定位服务。刚刚描述的 Akamai 所使用的方法依赖于 DNS 以及对一个 Internet“地图”的维护。还有一种方法在文献(Amir 1998)中有描述,它对多个 DNS 服务器赋予相同的 IP 地址,让网络底层根据最短路径路由协议自动把名字查询请求定向到最近的服务器。

11.1.7 容错性

Web 的容错性能主要通过客户端缓存和服务器复制来实现。除此之外,Web 中没有专门的容错方法。例如,HTTP 中没有加入任何帮助容错或错误恢复的功能。但要注意,Web 的高可用性是通过冗余实现的,这种冗余利用了在一些关键服务(例如 DNS)中通常可用的技术。例如,DNS 允许返回多个地址作为一个名字查询的结果。

11.1.8 安全性

考虑到 Internet 的开放特性,设计一个保护客户和服务器免遭各种攻击的安全体系结构是非常重要的。Web 中的大部分安全问题与建立客户和服务器之间的安全通道有关。在 Web 中建立一个安全通道的最主要方法是使用安全套接层(secure socket layer,SSL),该协议最初由 Netscape 提出。尽管 SSL 从未被正式标准化,但大多数 Web 客户和服务器都支持它。最近,一个 SSL 的更新已成为 RFC2246,称为传输层安全(transport layer security, TLS)协议(Dierks 和 Alien 1996)。

如图 11.23 所示,TLS 是一个与应用无关的安全协议,在逻辑上它位于传输层协议的顶部。为了简单起见,TLS(和 SSL)的实现通常基于 TCP。TLS 可以支持多种高层协议,其中包括 HTTP,我们下面还会讨论这一点。例如,可以使用 TLS 实现 FTP 或 Telnet 的安全版本。

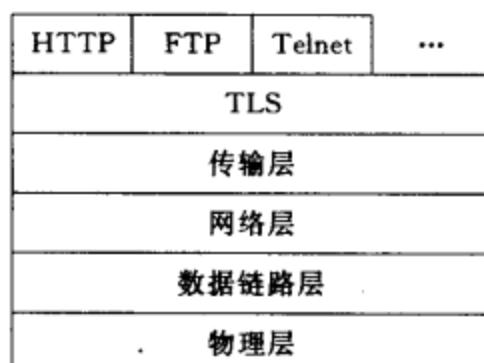


图 11.23 TLS 在 Internet 协议栈中的位置

TLS 本身组织为两层。协议的核心由 TLS 记录协议层(TLS record protocol layer)构成,它实现了客户和服务器之间的安全通道。通道的确切特征在建立通道时确定,但可能会包含消息的分组和压缩,以保证消息的身份验证、完整性和机密性。

建立一个安全通道需要经历两个阶段,如图 11.24 所示。首先,客户通知服务器自己能处理的加密算法以及支持的压缩方法。实际的选择通常由服务器做出,它把这个选择报告给客户。这些就是图 11.24 中所示的前两个消息。

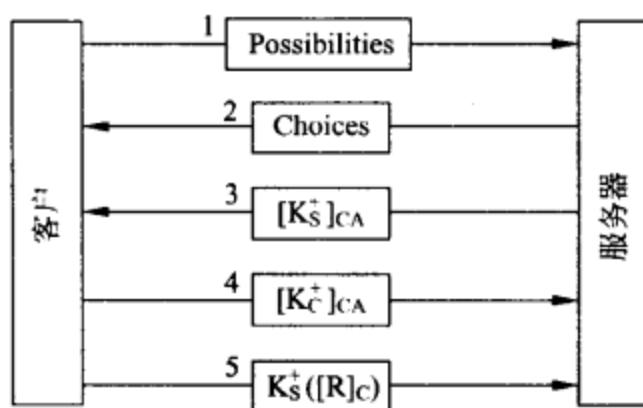


图 11.24 带有相互身份验证的 TLS

在第二个阶段中进行身份验证。总是要求服务器验证自己的身份,所以它向客户传递一个包含其公钥的证书,该证书由一个证书机构 CA 签发。如果服务器要求验证客户,客户必须也向服务器发送一个证书,如图 11.24 的消息 4 所示。

客户生成一个可供双方构造会话密钥的随机数,并使用服务器的公钥把这个随机数加密后发送给服务器。另外,如果要求对客户进行身份验证,客户就使用它的私钥对这个数进行签名,这就是图 11.24 中的消息 5。(实际上,随一个单独的消息发送随机数的混合签名版本也可起到同样的作用。)此时,服务器可以验证客户的身份,随后安全通道就建立起来了。

11.2 Lotus Notes

现在让我们研究一个与 Web 完全不同的基于文档的分布式系统。Lotus Notes 是一个面向数据库的系统,它起源于 Lotus 发展公司。但是现在它的整个销售和开发业务由 IBM 来运作。20 世纪 90 年代后期 IBM 收购了 Lotus。它运行在各种 Windows 和 UNIX 平台上。Notes 的大部分技术内幕和文件不对外公开。最近,Internet 上发布了一些 Notes 的指南和手册,以及有关其体系结构的资料(这些资料可以在 <http://www.notes.net/doc> 找到)。在文献(Lotus Development 2000)中有对 Lotus Notes 系统的概述。

11.2.1 Lotus Notes 概述

与 Web 类似,Lotus Notes 系统组织为一个(可能非常大的)客户-服务器系统。Notes 最初设计为工作在局域网中,但它现在也运行在像 Internet 这样的广域网中。它

的总体结构如图 11.25 所示。4 个主要组件联合组成了 Lotus Notes 系统：客户、服务器、数据库和中间件层。每个客户和服务器可以具有多个本地相关联的数据库。每个数据库构成一个文笺(note)的集合，文笺是所有 Notes 系统中的关键数据元素。后面我们还会讨论文笺和数据库。

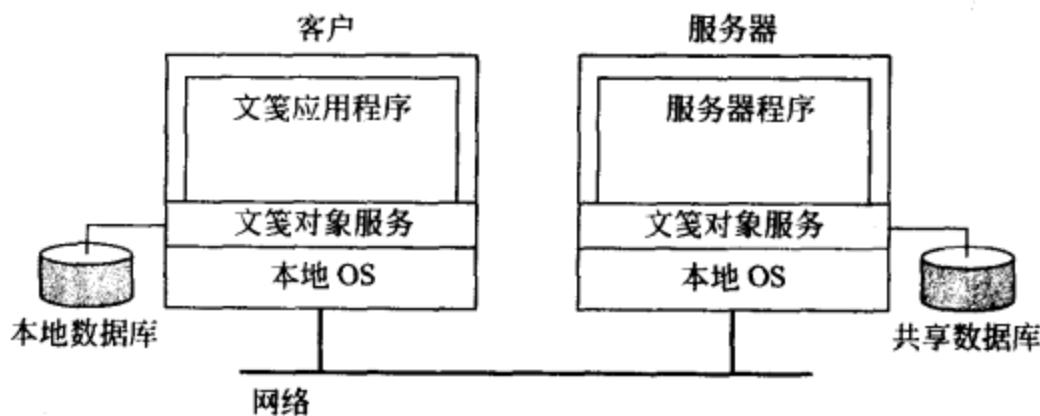


图 11.25 Lotus Notes 系统的总体结构

客户运行的应用程序可以用来访问数据库，这样的应用程序相当于 Web 浏览器。但它与浏览器的一个主要区别在于用户不仅要从数据库读取数据，还要对它们进行修改。另外，还有一个单独的工具组件供用户设计和维护他们自己的数据库。

Lotus Notes 服务器称为 Domino 服务器。一个服务器管理与其相关联的数据库集合。它的主要任务是为(远程)客户和其他服务器提供对这些数据库的访问。因此主要的服务器程序由完成下列功能的模块组成：监听来自网络上的请求，维护与远程进程的连接和会话，以及维护打开的本地数据库的信息。另外，还有许多与数据库管理有关的其他任务，它们通常作为单独的程序运行在服务器机器上。

Web 中的文档大多是通过文件实现的。相比之下，Lotus Notes 通过数据库存储和操作文档。这是这两种系统之间的重要的也是主要的区别，它在很大程度上决定了 Web 服务器和 Domino 服务器之间的差异。

客户、服务器和数据库通过一个独立的中间件组件结合在一起，这个中间件称为 Notes 对象服务(Notes object services, NOS)。它在底层的操作系统和网络之上实现了一个中间层，以允许客户和服务器进行通信和访问本地及共享数据库。因此，它由远程过程调用、存储工具、回调(callback)功能等组件组成。

文档模型

Lotus Notes 系统中的关键数据元素由文笺(note)构成，文笺本身实质上是一个项目的列表。项目(item)是用于存储与文笺相关联的数据的元素，每个项目具有与之相关的类型，由类型决定它能存储的数据种类。项目类型包括存储纯文本的文本项目、存储浮点数的数值项目、存储日期和时间的时间项目、存储编译后的 Notes 命令的公式项目，以及存储脚本的脚本项目等。

每个文笺也可以有一系列相关的文笺作为其子文笺，并且每个文笺最多只能有一个父文笺，这样就形成了一个文笺的层次结构。这种层次性反映了 Notes 系统的一个最初

设计意图,即提供一个允许用户记入文笺以及记入对文笺的反应的系统。记入和反应之间的关系由父子关系来维护,这与 Network News 的工作原理类似。

数据文笺和其他类型的文笺之间还有进一步的差异。数据文笺(data note)相当于 Web 文档:它表示可以呈现给用户的文档,但它本身可能会包含不同类型的数据元素,例如音频、视频、图像、纯文本、图标等。数据文笺也被简单地称为文档。

还有许多其他类型的文笺,如图 11.26 所示,它们大体上可以分为设计文笺和管理文笺两大类别。设计文笺用于显示和控制文档;管理文笺用于管理 Notes 数据库。

文笺类型	类别	说明
Document	数据	面向用户的文档,例如 Web 页面
Form	设计	用于创建、编辑和查看文档的结构
Field	设计	定义一个在表单和子表单之间共享的字段
View	设计	用于显示一个文档集合的结构
ACL	管理	包含数据库的一个访问控制列表
ReplFormula	管理	描述数据库的复写

图 11.26 不同类型文笺的例子

Form(表单)文笺类似于 Web 用户看到的表单,也类似于传统的数据库表单。表单将其项目作为字段,提供对字段的位置和表示方法的定义,从而定义文档呈现给用户的形式。一些字段可以为不同的表单所共享,这通过 Field(域)文笺来描述。View(视图)文笺的概念也来自数据库领域,它提供表示文档集合的手段。例如,一个 View(视图)文笺可以描述如何把一个文档的集合表示为一个表,其中的列用于显示这些文档中的特定项目。

还有各种管理文笺,例如用于存储一个访问控制列表的 ACL(访问控制列表)文笺。Notes 提供了复制数据库的手段。复制的确切过程存储在一个 ReplFormula(复制公式)文笺中,这种文笺也属于管理文笺。

像 Web 中的文档一样,文笺也可以通过超链接指向其他文笺,这种超链接在 Notes 中称为文笺链接(notelink)。一个文笺链接标识数据库及其包含的文笺。也就是说,可以交叉引用不同数据库中的文笺。还可以使用 URL 来指向文笺,URL 中通常嵌入一个文笺链接和一个对控制被标识数据库的服务器的引用。我们在介绍 Notes 中的命名机制时还会讨论这个问题。

要弄明白的是文笺相当于 Web 文档,但它是以一种完全不同的方式组织的。具体地说,一个文笺的内容(由其项目来描述)和把它呈现给用户的形式(由设计元素描述)之间有严格的区分。这种区分相当于我们前面讨论过的 XML 和 XSL 之间的区分。

11.2.2 通信

像许多其他分布式系统一样,Lotus Notes 使用底层的 RPC 系统进行客户和服务器之间的通信。Notes 的 RPC 系统对所有客户基本上是透明的,它实际上位于 NOS 中间

件层内部。当一个服务器接收到一个 RPC 请求时,它启动一个单独的任务来处理与这个 RPC 相关的剩余通信过程,这与其他 RPC 系统采用的面向连接的方法类似。

Notes 为在同一机器上运行的进程之间交换信息提供了轻便的进程间通信功能。但是,与 CORBA 和 DCOM 之类的分布式系统不同,它对不同机器的进程间的通信不提供很多的其他支持。大多数的通信通过明确定义的接口进行,这些接口使用 Notes 的 RPC 系统来实现。

Notes 处理电子邮件的子系统是一个例外。Notes 的电子邮件消息总是以 MIME 格式发送,这个子系统本身可以使用诸如 SMTP 之类的各种邮件传输协议。Notes 还有一个单独的邮件协议。

为了方便高级应用程序(例如工作流系统)的开发,Notes 提供各种手段来自动发送电子邮件以响应数据库中发生的事件。例如,改变一个文笺中的特定项目可以触发包含一份已修改文笺的拷贝的电子邮件消息。与此类似,输入的电子邮件消息可以被自动处理,从而导致对 Notes 数据库的修改。

11.2.3 进程

像在 Web 中那样,Notes 的客户和服务器之间的差异很明显。客户端软件由允许用户与服务器进行交互的程序(包括 Web 浏览器)组成。另外,还有一个用于设计和实现 Notes 应用的单独程序。这个程序相当于许多现代的数据库应用开发工具,它允许用户设计文笺及其相关的表单、视图、事件、任务等。

服务器端软件由一个处理大量内置任务的主程序组成,这些任务包括处理输入请求、打开和关闭本地数据库、保持数据库一致性以及管理服务器簇(我们马上就会讨论到)等。除了这些内置的任务之外,还有许多的其他服务器任务(server task)作为单独的进程运行在主服务器所在的机器上,并受主服务器的完全控制。主服务器及其服务器任务联合构成了 Domino 服务器,如图 11.27 所示。

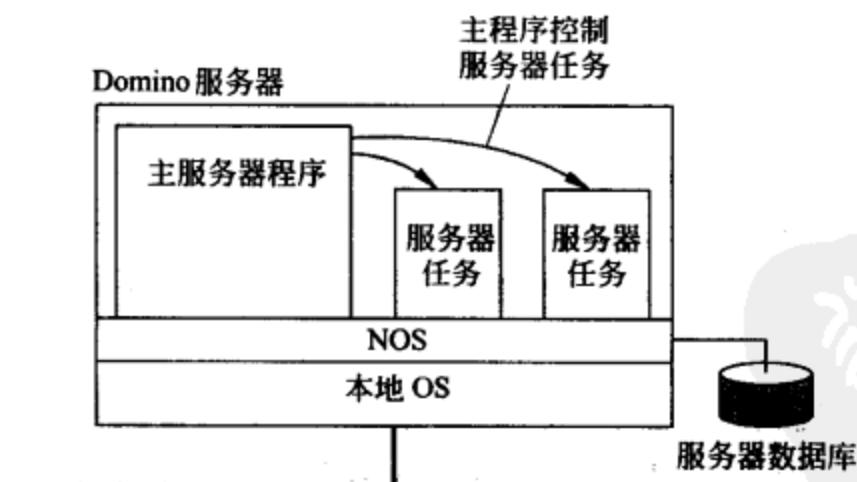


图 11.27 Domino 服务器的总体结构

Domino 服务器的进程间通信使用前面所述的 NOS 层进行。NOS 不仅提供访问和管理本地数据库的功能,还提供许多进程间通信功能。

为了增强容错能力和改进性能,多个 Domino 服务器可以组织为一个簇(cluster),即

2~6个运行相同服务器并具有相同数据库拷贝的机器的集合。Notes簇与Web服务器簇类似，但在组织方式上有所不同。例如，Notes簇对客户是不透明的；客户将知道自己在和一个服务器簇打交道，并且被要求选择一个服务器以定向请求。

客户按照如下方式得知簇的情况。一般来说，要访问一个特定数据库中的文笺，客户应具有一个对该文笺的引用，其中包含对于该数据库相关联的一个服务器的引用。客户第一次联系数据库的服务器时，服务器返回同一个簇中的服务器列表。如果客户最初联系的服务器非常忙，客户会在簇中查找其他的服务器。

如图11.28所示，第二个服务器被请求构建一个簇中可用服务器的排序列表。排列的顺序由每个服务器的当前负载决定，负载最轻的服务器位于列表的顶端。客户先联系列表中的第一个服务器，查看它是否可以处理请求。如果不可以，客户将继续联系列表中的下一个服务器。

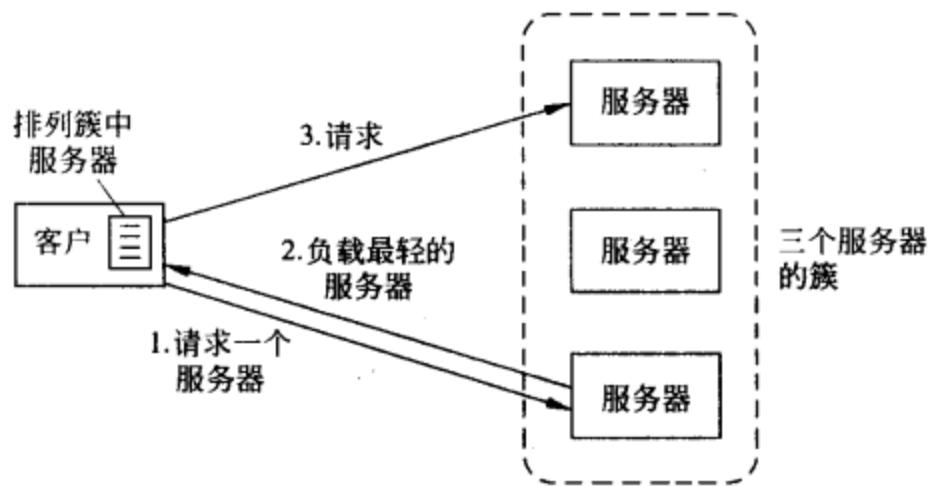


图11.28 Domino服务器簇中的请求处理

服务器的负载 W 由每个服务器进行计算，它是响应时间的函数，返回值在 0 和 100 之间。具体地说，如果 T_{response} 是当前响应时间， T_{opt} 是最佳响应时间，那么 W 的计算方法如下：

$$W = \max\{0, 100 - (T_{\text{response}}/T_{\text{opt}})\}$$

一个服务器具有较小的 W 值表明该服务器比具有较大 W 值的服务器要忙。当前响应时间通过使用一个具有代表性的函数集每分钟计算一次。如果当前没有需要处理的请求， W 的值将为 100，表明对下一个输入的请求会有最佳响应时间。

11.2.4 命名

由于 Notes 面向数据库，所以它的命名系统所起的作用不同于迄今为止我们所讨论的许多其他分布式系统。例如，由于文笺存储在数据库中，所以不能使用名字解析的方法来访问它们，而是使用更为传统的数据库操作来搜索它们，或是仅通过我们前面讨论的视图和链接来访问它们。

一个数据库本身由一个单一的文件表示，因此它的命名遵循 Notes 实现的底层文件系统的命名约定。除了数据库文件和其他文件，Notes 中还有许多需要命名的其他实体，例如用户、服务器、公钥等。这些信息包含在一个称为 Domino 目录 (Domino directory)

的特殊数据库中,该数据库的访问方式同 Notes 中的任何其他数据库一样。

Notes 实质上使用两种方式支持较传统的字符串名字。首先,Notes 提供由 LDAP 实现的识别名称服务。这些目录服务可以用来访问数据库,但也可以用来组织一个层次性的名字空间,正如我们在第 4 章中所讨论的。注意 LDAP 服务往往由 Notes 数据库来实现。

第二种形式的字符串命名是使用 URL。支持这种功能是为了允许使用 Web 访问 Domino 服务器。另外,实现基于 Notes 的 Web 服务器也需要这种功能。Notes 中的 URL 采用如图 11.29 所示的一般形式,这与 CGI 程序使用的 URL 一致。

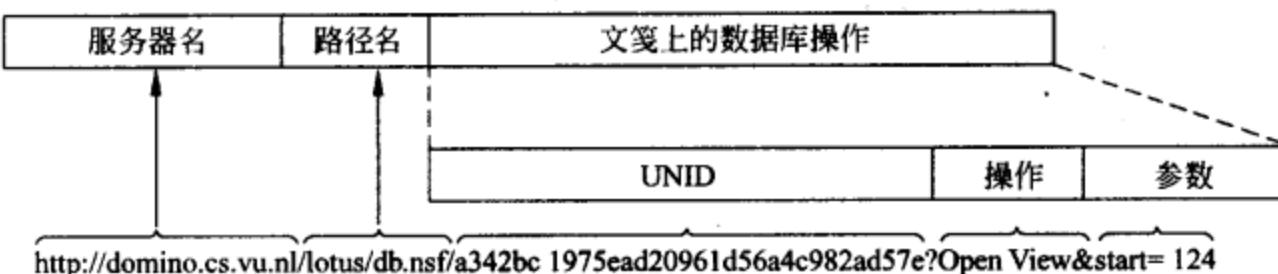


图 11.29 用于访问数据库的 Notes URL

Notes 中的一个 URL 由三个主要部分组成:能够处理 HTTP 请求的 Domino 服务器的 DNS 名字、服务器上本地 Notes 数据库的路径名以及用于实际访问该数据库中的一个被标识的文笺(即第三部分)。第三部分最为有趣,它包含一个标识符(我们马上就会讨论)、一个操作的名称和该操作的可选参数。Domino 服务器负责对被标识文笺执行指定的操作。

标识符

尽管字符串名字在 Notes 中起着重要的作用,但大多数命名问题是面向机器的标识符处理的。Notes 区分多种标识符,如图 11.30 所示。最重要的一种是通用 ID (universal ID, UNID),它用作文笺的全球惟一标识符。UNID 是赋予每个文笺的一个 16 字节的值。所有复制的文笺副本都具有相同的 UNID。

标识符	范围	说明
UNID(通用 ID)	全世界	赋予每个文笺的全球惟一的标识符
OID(创始者 ID)	全世界	文笺的标识符,但包含历史信息
database ID(数据库 ID)	服务器	数据库的依赖于时间的标识符
note ID(文笺 ID)	数据库	相对于一个数据库实例的文笺标识符
replica ID(副本 ID)	全世界	用于标识一个数据库的相同拷贝的时间戳

图 11.30 Notes 中的一些主要标识符

创始者 ID(originator ID, OID)与 UNID 有关,用于标识一个文笺的特定实例。OID 包含文笺的 UNID,此外还包含一个 2 字节的序列号和一个 8 字节的时间戳。这两者用来检测和解决对一个复制文笺进行并发更新时可能会出现的冲突。我们稍后讨论 Notes

中的复制时还会提到 OID。

数据库 ID(database ID)并不是一个真正的标识符,而是一个时间戳,它表示数据库创建的时间或上一次服务器崩溃后数据库恢复的时间。可以把它看作一个范围局限于管理它的 Domino 服务器的标识符。

文笺 ID(note ID)用于在一个特定的数据库实例中惟一标识一个文笺。文笺 ID 相当于文件系统中的文件标识符,因为它仅在一个数据库的范围内标识文笺。每个数据库都使用一个单独的查找表来把文笺 ID 映射到它在数据库中的物理位置。这样,无论何时重新组织一个数据库,只需调整其查找表,而数据库的所有文笺 ID 将保持不变。

最后,副本 ID(replica ID)用于表示参与复制的一组数据库。如果两个数据库具有相同的副本 ID,那么在一个数据库执行的所有更新都会传播到另一个数据库。原则上,副本 ID 是全球惟一的,但是由于它仅基于时间戳,因此无法保证惟一性。我们稍后会解释,数据库不必以完全相同的方式组织,尽管很多情况下都是如此。副本 ID 只是用于标识那些共享相同更新数据的数据库集合。

11.2.5 同步

与任何其他分布式系统类似,Notes 提供多种锁定机制以保证对其数据库的独占访问。而且,它还支持事务。但是,Notes 把这些同步机制限制为只能在单个 Domino 服务器上进行操作,就这个意义而言,Notes 同步机制是以非分布的方式实现的。Notes 不支持分布式的锁定或事务。

11.2.6 复制

与同类的许多其他商业系统不同,Notes 从一开始就支持复制(关于 Notes 复制的早期文章,请参见文献(Kawell 等 1988))。它使用一种懒惰形式的更新传播,在没有相互冲突的更新的情况下可保证最后的一致性。Notes 中的复制主要应用于文档(也即数据文笺),其工作原理如下所述。

连接文档(connection document)在复制过程中扮演着关键的角色。连接文档是包含在 Domino 目录中的特殊文笺,它们详细描述复制的时间、方式和对象。每个 Domino 服务器使用一个或多个相关联的复制器任务来执行一个连接文档中描述的复制方案。在图 11.31 中列出了 4 种可能的复制方案。默认的方案是拉/推方案。在这种方案中,一个复制器任务建立与目标服务器的一个连接,然后把自己的更新推给该服务器。并且,它拉入目标服务器上已发生的所有更新。请注意,这种方案和我们在第 6 章中讨论的 epidemic 算法中的推/拉方案完全相同。Notes 中的复制深受这些算法的影响。

方案	说明
拉/推	复制器任务拉入目标服务器上的更新,并且把自己的更新推给目标服务器
拉/拉	复制器任务拉入目标服务器上的更新,并响应来自目标服务器的获取更新的请求
只推	复制器任务只把自己的更新推给目标服务器,但并不拉入目标服务器上的任何更新
只拉	复制器任务只拉入目标服务器上的更新,但并不把自己的更新推给目标服务器

图 11.31 Notes 中的复制方案

也可以采用拉/拉方案进行复制,即在两个不同服务器上的两个复制器任务从对方处拉入更新。与此类似,可以采用只推方案或只拉方案。在这两种方案中,更新以一种单向的方式传播到目标服务器,反之亦然。

需要考虑的更新操作有三种:修改、添加和删除一个文档。在复制过程中,重要的是要知道一个文档是否并且何时已被修改。应该把已修改的文档传播到副本中去。记录对一个文筈的修改是通过更新文档的 OID 中的序列号和时间戳来完成的。旧的值被复制到与该文档相关联的一个历史列表中。

通过生成一个全球惟一的新 OID 可以记录一个新文档的添加。为了记录一个文档的删除,可以把一个删除存根(deletion stub)作为被删除文档的替代放在数据库中。这个删除存根类似于 epidemic 算法中引入的死亡证书。请注意,在原则上,只有确信所有相关的文档拷贝已被实际删除时才能删除一个删除存根。

在正常情况下,当发生复制时,所有的更新都会在副本之间传播。回忆一下,副本标识为具有相同副本 ID 的数据库。就以相同的方式进行内部组织而言,两个副本是否确实相等,这并不重要。惟一重要的是两个数据库中所包含的同一个文档的拷贝必须保持一致。通过与这些拷贝相关联的 UNID 可以识别它们。

1. 冲突解决

在 Notes 采用的复制方法中,一个潜在的基本问题是可能会(并且的确会)发生冲突。如果对同一文档的多个拷贝相互独立地进行更新,就会发生写写冲突;当稍后传播这些更新时,就需要解决这个冲突。Notes 采用下面介绍的方法检测和解决这类冲突。

假设按照前面讨论的拉/推方案对 A 和 B 这两个副本进行复制。B 所执行的更新会被 A 拉入,然后 A 所执行的更新会被推给 B。在复制时,会为每个副本构建一个 OID 列表。

当 B 的列表中包含一个不在 A 列表中的 UNID 时,显然是 B 中添加了一个新文档,A 应该拉入这个新文档。通过进行类似的比较,添加到 A 中的新文档也会被推给 B。如果两个列表中包含相等的 OID,那么就认为这些 ID 指向的文档在 A 和 B 中是相同的。因此不用传输任何更新。

现在假设 A 和 B 的列表中都有一个文筈 N,但与它们相关联的 OID 不同。也就是说,N 有两个不同的拷贝 N_A 和 N_B ,这两个拷贝具有相同的 UNID 和不同的 OID。这种情况下,A 上的复制器任务将查看这两个拷贝的历史列表。如果一个列表是另一个的子列表,那么不会发生冲突,因为显然只有一个拷贝曾被更新过。这时,被更新的拷贝可以传输给另一个副本,随后历史列表和 OID 可以修改为完全相等。

但是,如果两个历史列表不同,并且没有哪个是另一个的子列表,这表明存在着一个有待解决的冲突。此时有必要进行更细致的考察。特别是,Notes 允许通过检查构成一个文筈的项目来合并文档。只要修改了一个项目,OID 中的序列号部分就会增加,并赋予这个项目。通过这种方法可以准确地记录究竟哪个项目已被修改了。

现在假设两个历史列表在序列号 k 之前的部分是相同的。如果在序列号 k 之后发生的对 N_A 的所有修改应用于与 k 之后对 N_B 的修改不同的项目,那么这两个文档可以安全

地合并,如图 11.32 所示。其原因是没有相互冲突的修改。也就是说,A 所做的更改与 B 所做的更改没有发生冲突。

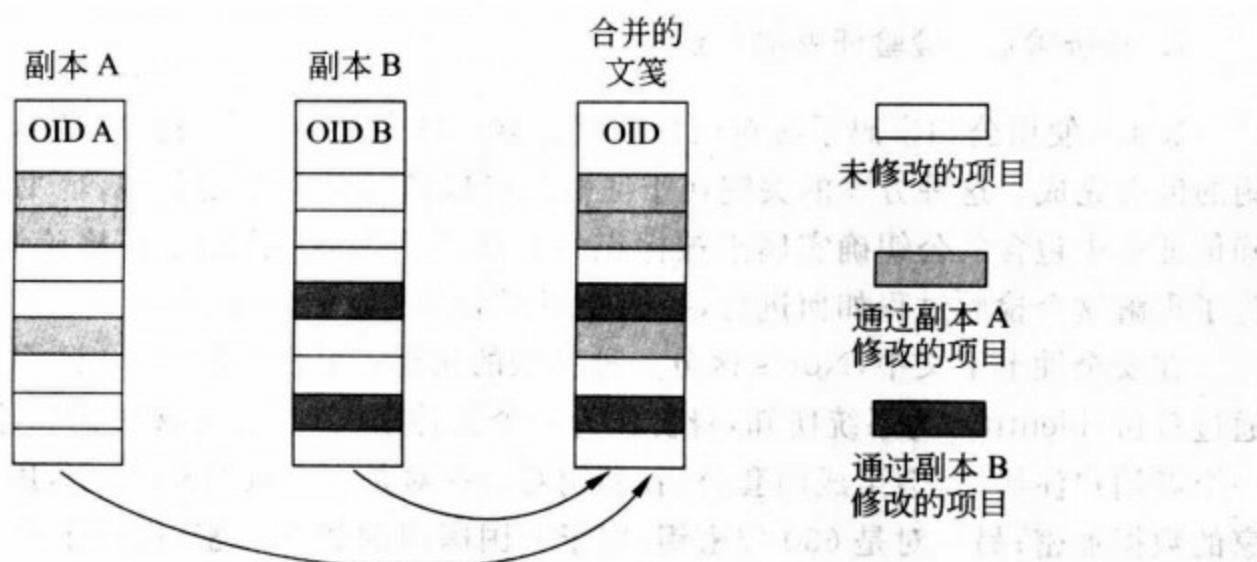


图 11.32 安全地合并两个具有相互冲突的 OID 的文档

在所有其他情况下都会记录不可解决的冲突。这时,Notes 把其中一个文档提升为“获胜者”,把另一个文档变为“失败者”。具有最高的 OID 序列号的拷贝会被声明为获胜者。当序列号相同时,OID 中的时间戳最接近当前时间的那个拷贝将成为获胜者。

我们把删除一个拷贝而修改另一个拷贝的情况下的冲突解决方法留作本章结尾处的练习。

2. 簇复制

前面讨论的复制方法通常应用于分布范围广泛的服务器集合。服务器簇总是位于同一个局域网内,因此它们的复制采用另一种方法。服务器簇不是显式地使用连接文档安排复制动作,而是把一个更新立即推给簇中的所有副本。

为此,每个服务器都维护一个更新事件的队列,在本地执行的各种数据库操作会向其中写入事件。一个专门的复制器任务每秒检查这个队列一次,以查找需要传播给簇中其他服务器的更新。这样的更新会被发送给其他副本,而相应的事件会从队列中删除。

11.2.7 容错性

从分布式系统的容错角度看,Notes 没有提供任何屏蔽失败的特殊机制。它对容错最显著的支持由一个恢复子系统组成,该子系统结合到单个服务器上的数据库实现中。这个子系统使用一种先写日志的方法,即一个事务中的每个操作在数据库中执行之前首先记录到一个日志中。由于我们在第 5 章中已经详细说明了事务日志,所以这里不再赘述。

11.2.8 安全性

所有 Notes 系统安全性的基础依赖于对其用户的身份验证。一旦用户通过了身份验

证,就可以访问服务器数据库中的文筈。Notes 用一个广泛的机制来控制用户对文筈及文筈的一部分的访问。下面我们将对身份验证和 Notes 中的访问控制分别进行详细介绍。

1. 身份验证: 检验证书的合法性

Notes 使用公钥密码系统对用户和服务器进行身份验证。身份验证通过传递包含公钥的证书完成。这种方案的关键在于证书是可以信任的。也就是说,证书的接收者必须确信证书中包含的公钥确实属于被标识的发送者。Notes 特别重视检验证书的合法性。为了理解这个检验过程如何进行,我们介绍证书的实际创建过程。

在安全性上下文中,Notes 区分三种类型的实体: 用户、服务器和证明者。每个实体通过身份(identity)为系统所知,身份作为一个文件来实现,它包含密钥和证书。例如,当一个新用户注册时,会生成两套公钥/私钥对。一对是 512 位 RSA 密钥,用于美国之外国家的数据加密;另一对是 630 位密钥,用于美国国内的数据加密,也用于身份验证和放置签名。另外,还会生成一个证书,并签署一个为用户正在加入的 Notes 系统所知的证书颁发机构的签名。显然,只有经授权的管理员才能够使用这个签名。

所有这些信息都存储在 Notes 系统中用户的身份文件中。诸如私钥一类的机密信息使用从用户的密码生成的密钥来加密。与此类似,Domino 服务器也具有与之相关的一个身份文件,该文件存储在系统中的某处。与一个证书颁发机构相关联的身份文件一般存放在一个物理上安全的地方。

Notes 支持层次性的认证。也就是说,当把一个用户添加到系统中时,系统管理员也许会决定在生成用户的证书时包含另一个含有与管理员使用的私钥相关联的公钥的证书。按照这种推理,如果系统管理员的证书需要检验,则该证书中必须包含一个更高级别的机构的证书。

现在假设 Alice 需要让 Bob 验证她自己的身份。首先我们考虑这种情况: Alice 和 Bob 都在同一组织(例如 Franeker 大学,简称 FU)中工作;Alice 在计算机科学(CS)系工作,而 Bob 在电子工程(EE)系工作。Alice 和 Bob 在他们各自的身份文件中都有一个 Franeker 大学的公钥。

Alice 把她的证书传递给 Bob,以使 Bob 可以检验这些证书中包含的公钥。Bob 进行的检验遵循下列规则:

- (1) 存储在 Bob 的身份文件中的任何公钥都为 Bob 所信任。
- (2) Bob 的身份文件中拥有一个证书颁发机构的公钥,那么来自由该机构签署的证书的公钥是可信任的。
- (3) 一个证书颁发机构拥有 Bob 信任的公钥,那么由该机构签署的证书中包含的公钥也是可信任的。

Alice 将不得不传递两个证书: $[A, K_A^+]_{EE}$ 和 $[EE, K_{EE}^+]_{FU}$ 。前者包含她的公钥并由 EE 签名;而后者包含 EE 的公钥并由 FU 签名,如图 11.33 所示。Bob 会信任 K_{EE}^+ ,因为他可以使用存储在他的身份文件中的 FU 的公钥 K_{FU}^+ 来验证与 K_{EE}^+ 相关联的证书。根据第一条规则, K_{FU}^+ 得到了 Bob 的信任;根据第二条规则, Bob 又信任了 K_{EE}^+ 。

现在根据第三条规则,Bob 可以验证 $[A, K_A^+]_{EE}$ 的合法性。Bob 将相信 EE 确实曾向

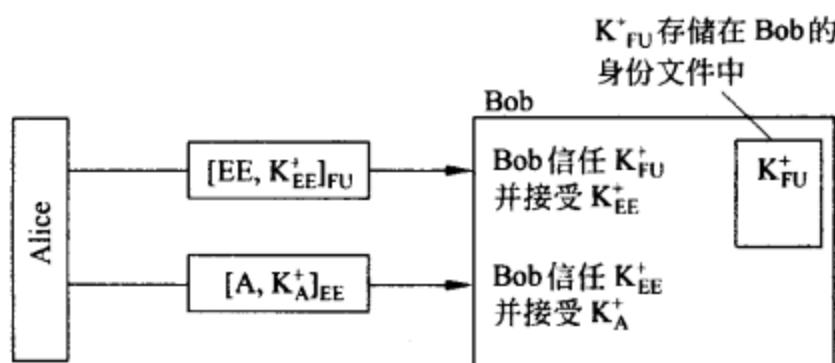


图 11.33 Notes 中的公钥检验

Alice 发出了公钥 K_A^+ , 因此他现在将信任 Alice 发送的公钥确实是她的公钥。

这种方法的一个问题是, Alice 传递给 Bob 的证书必须由某个证书颁发机构的派生机构签署, 而且在 Alice 和 Bob 的身份文件中都必须有该机构的公钥。在我们的例子中, 这是 Franeker 大学(FU)的公钥。但是, 如果 Alice 和 Bob 在不同的大学里工作, 那么对 Alice 证书的检验将会失败。

这个问题的解决方法是使用交叉证书(cross certificate), 这种证书显式地描述了一个对等信任关系。例如, 如果 Alice 在 Lommel 技术学院(TUL)工作, 她可以使用一个包含 TUL 的公钥的证书, 但必须由 FU 进行签名, 以使 Bob 信任她拥有的公钥。

在检验了证书的合法性后, 则通过发送一个使用接收者的公钥加密的口令来进行相互的身份验证。接收者必须通过把口令返回给发送者来证明它持有相应的私钥, 我们在第 8 章中已解释了这个过程。有关 Notes 中的身份验证的更多详细信息请参见文献(Nielsen 等 1999)。

2. 访问控制

Notes 为其各种组件的访问控制提供了广泛的支持, 这里我们仅作简要的讨论。系统的不同部分之间会有一些差异, 图 11.34 中总结了这些差异。对于服务器, 会维护一个访问控制列表, 其中详细指定了哪些用户、用户组和服务器可以向一个服务器的哪个端口发出请求。这是 Notes 的基本访问控制性能之一。

部分	说明
服务器	使用 ACL 指定对服务器和端口的访问权限
工作站	使用列表指定脚本等的执行权限
数据库	使用 ACL 指定不同类型用户的权限
文件	使用 ACL 控制 Web 客户的访问
设计文稿	使用 ACL 控制文档的表示及其他方面
文档	使用 ACL 控制对文档的读写访问

图 11.34 根据访问控制把 Notes 分为几个部分

对于工作站, Notes 维护执行控制列表(execution control list, ECL), 其中详细指定

了对可执行代码(这些可执行代码作为读取文笺的一部分传递到工作站)的限制。例如,一个 ECL 可以限制对工作站本地文件系统的访问、限制发送邮件的能力、限制访问某些网络地址等。可以把 ECL 看作我们在第 8 章中描述的沙箱(sandbox)模型的一个扩展,它给予下载代码一些特定的权限并剥夺其他一些权限。但是请注意,在许多情况下,这些代码的执行只是基于信任。也就是说,在执行指令时并不进行安全检查。

数据库的访问控制得到了广泛的支持。每个服务器都有一个与之相关的 ACL 来详细指定谁可以做什么。共有 7 种数据库访问级别。最高的是管理员级别,用户可以添加、删除和修改任何内容;最低的是“禁止访问”级别,用户只能读或写公共文档。对于可以存储在数据库中的不同类型的文笺,数据库访问控制有各种相关的访问特权。例如,针对文档可执行的操作和针对设计文笺的可执行操作是不同的。

文件访问控制与允许 Web 客户访问的内容有关。另外,它还控制对非数据库文件的访问。

通过定义拥有对设计元素的访问特权的角色可以细化数据库访问。例如,可以授予一个特定用户集合修改数据库中某个设计文笺子集的权限。数据库管理员并不是列出这个用户集合,而是可以指定一个角色,定义该角色所具有的特权。然后授予用户“扮演”这一角色的权限。

最后,每个文档具有它自己的相关 ACL。访问控制可以是项目级别的,这时一个项目甚至可以具有一个相关的密钥以得到进一步的保护。还可以指定一个文档的某些部分应该对未授权用户隐藏,或者可以指定当把一个文档发送到远程站点时应该进行签名。

在文献(Lotus Development 2000)中可以找到关于 Notes 中的访问控制的更完整讨论。

11.3 WWW 和 Lotus Notes 的比较

WWW 和 Lotus Notes 是两种使用最广泛的基于文档模型的分布式系统。但是它们之间也存在着很大的差异,现在我们简要地对比一下分布式系统的基本原理如何应用于这二者。

1. 基本观点

Web 和 Lotus Notes 都采用一种简单的客户-服务器模型,其中一个单一的服务器管理能够被远程客户访问的文档集合。整个文档集合分布在多个服务器中。从这方面看,二者是类似的。但是,它们的基本文档模型之间存在着重要的差别。

Web 接近于较传统的模型,其文档实质上是基于文本的文件,其中包含 HTML 或一种更新的语言(例如 XML)的各种标记命令。但是,这些文档中可以嵌入其他类型的文档,例如音频、图像和视频。而且,随着 Web 的普及,一些更复杂的元素(例如脚本和小程序)也已经加入进来。

随着 Web 的继续发展,作为标记语言的 HTML 将逐渐为 XML 所替代,因为后者允许对文档的结构进行更好的描述。而且,XML 把结构的构造和表示方法分离开来,而在 HTML 中这些是混合在一起的。

Lotus Notes 的基本模型有很大不同。Notes 中的关键数据元素是一个包含粒度可能很小的项目的列表,这种列表简单地称为文笺。与 Web 相比,文笺是一种与数据库紧密结合的数据结构。文笺的修改和表示完全通过诸如表单和视图这样的专门的数据库机制来进行。这些机制由称为设计元素的特定的文笺进行说明。

与 Web 相比的另一个不同是,Lotus Notes 尝试使用文笺来管理整个系统。例如,有用于控制复制、处理安全问题,以及完成其他功能的文笺。而 Web 则使用单独的机制来解决这些问题。

2. 通信

Web 中的通信是使用 HTTP 这一专门的文档传输协议进行的。HTTP 规定了与文档相关的可能操作,这些操作主要局限于获取文档和替换服务器上的文档。

Lotus Notes 中的通信使用传统的 RPC 子系统进行内部处理。另外,Notes 重点实现通过电子邮件进行的高级通信,提供了自动处理电子邮件的接收、处理和发送的多种工具。

还应当注意的是,两种系统对同一机器上的进程间通信的处理方法也不相同。在 Web 中,客户和服务器都对底层的操作系统有很强的依赖性,因此所有的进程间通信由操作系统提供的功能处理。相比之下,Notes 使用一个独立的层 NOS 来隐藏客户和服务器所在操作系统的不同。这种方法增强了许多 Notes 应用程序的可移植性。

3. 进程

比较 Web 和 Lotus Notes 对于进程的使用和组织,可以发现二者在许多方面颇为相似。最重要的 Web 客户是浏览器,它为用户提供可以获取和显示文档的图形界面。大多数现代浏览器还提供编辑文档的功能。而且,多数浏览器都可以通过使用插件进行动态扩展。

在 Lotus Notes 中,也向客户提供一个图形界面以供查看远程数据库中存储的文笺。而且,Notes 客户通常配备有一个单独的应用程序,用于通过表单、视图等来设计文笺。由于 Notes 客户被设计为处理所有(预定义)类型的文笺,所以无需像在 Web 中那样提供可扩展性。

服务器的组织也很相似。只是传统的 Web 服务器普遍设计为运行在文件系统上,而 Notes Domino 服务器通常近似于一个传统的数据库服务器。

在 Web 中,服务器端的灵活性通过支持 CGI 程序来实现。CGI 程序通常作为独立的进程启动并可以作用于数据库。这样,Web 服务器就可以实现类似 Domino 所提供的功能。另外,Web 服务器还可以支持称为 servlet 的可动态加载的模块。

Domino 服务器构造为一个主程序,它作为一个单独的进程运行。许多其他任务由一些服务器任务来执行。服务器任务是与主程序并行启动和执行的程序,它们相当于 Web 的 CGI 程序。

Web 和 Notes 都通过支持服务器簇来增强自己的功能。在服务器簇中,多个服务器协同处理来自客户的请求。它们之间的一个主要区别是提供的透明性程度不同。在实际运行中,Web 客户也许不知道自己的请求定向到一个服务器簇;而 Notes 客户则必须建立一个簇中的服务器列表,以便在需要时重定向自己的请求。

4. 命名

Web 使用面向文件的方法,而 Notes 使用数据库。Web 和 Notes 在命名方面的区别与此有直接的关系。在 Web 中,通过使用 URN 或 URL 命名文档。目前 Web 使用的 URL 中通常含有一个文件名,该文件名对应于包含文档的文件存储在服务器上的本地路径名。而 URN 则用作与位置无关的标识符,例如用 ISBN 表示书籍。

在 Notes 中,文档(以及其他类型的文笺)具有与之相关联的 UNID。UNID 是真正的标识符,用于命名和访问文笺。文笺没有与之相关联的字符串名字以供用户在数据库中查询。然而,Notes 却提供一个对 Web 的命名接口,方法是在 URL 中嵌入 UNID 以及存储被标识的文笺的服务器和数据库的名字。

5. 同步

对于同步,Web 和 Notes 都只提供最小化的支持。在这两种系统中,同步实质上都局限于本地的锁定机制;不支持对分布在多个服务器上的多个文档的锁定。直到最近才提出了要允许对共享文档进行锁定以支持 Web 上的协作编辑的建议。

6. 缓存和复制

在对缓存和复制的支持方面,Web 和 Notes 之间存在着较大的差异。在 Web 中缓存一向起着重要的作用,即增强可扩展性。现在层次性的缓存机制已经是很常见的做法了,在用户、站点、地区和国家的层次上都可以进行缓存。由于 Web 中的许多文档很少变化,因此缓存往往是有效的。然而,这种情况正在迅速地发生变化,使得缓存机制,特别是层次性缓存,变得不是那么有效了。

各种各样的缓存一致性协议已经提出并已在 Web 中付诸实现。许多协议实现弱一致性:一个缓存的文档在租用到期之前被认为是有效的。在此期间,有可能文档已被更改,而缓存的拷贝却没有注意到这样的更改。

在 Notes 中,客户端缓存看上去不起什么作用,因为没有 Notes 文档提到这一功能。Notes 假设文档是共享的并且是经常更改的,因此缓存不会很有效。

传统上用镜像整个站点的形式支持 Web 中的复制,这种形式很不灵活。随着内容分发网络(CDN)的出现,动态复制正变得越来越普及。CDN 允许把文档动态复制到靠近用户的服务器上。通过 CDN 特定的方式处理一致性。

与缓存相比,复制在 Notes 中有着重要的作用。Notes 中使用懒惰传播更新的 epidemic 算法来支持复制。这种方法有可能导致需要手工解决的写-写冲突。Notes 为自动解决冲突做出了许多努力,但它仍无法处理所有可能发生的情况。当需要手工干涉时,Notes 决定一个获胜的文档,并把另一个文档作为失败文档;但它也允许用户按照他们认为合适的方式决定如何解决冲突。

7. 容错

Web 和 Notes 应用相似的技术来实现容错性。在 Web 中,通信的可靠性完全基于 TCP 的使用。更为有意义的是使用了各种 Web 服务器簇来增强可用性以及改进单一

Web 服务器的性能。Web 中提出并实现了多种解决方法,这些方法大多是依赖一个专门的前端服务器把输入的请求转发给主服务器之一。

Notes 提供专门的工具支持 Domino 服务器簇,其中的数据库使用一个相对较强的一致性协议进行复制。在这个协议中,正常情况下每秒更新传播一次,但除此之外它与我们前面描述的懒惰复制的传播协议没有什么不同。

在这两种系统中,恢复机制即使可用,也是局限于单个服务器崩溃的情况,没有专门针对分布式系统的设计。

8. 安全性

在 Web 和 Notes 中,安全性都起着至关重要的作用。在 Web 中,一般通过使用 TLS 来保证安全性。TLS 允许在客户和服务器之间建立一个安全通道,随后可以进行访问控制。授权一般依赖于服务器。

Notes 使用证书作为身份验证的基础,但在检验证书的合法性上给予了格外的注意。它使用自己的信任模型决定是否可以信任一个证书中包含的公钥。通常,使用一个层次性的命名机制建立信任关系,在服务器的建立期间可以对这种机制进行部分的配置。通过使用交叉证书,两个不同的 Notes 系统可以相互信任,这样客户就可以使用另一个系统的服务器。

Notes 中的访问控制通过使用包含 ACL 的特殊文笺来实现。Notes 提供许多不同的访问权限和访问级别,从而使高度可定制的访问控制成为可能。

作为结论,图 11.35 总结了对 Web 和 Notes 的比较。

问题	WWW	Notes
基本模型	标记文本	文本项目的列表(文笺)
扩展	多媒体、脚本	多媒体、脚本
存储模型	面向文件	面向数据库
网络通信	HTTP	RPC、电子邮件
进程间通信	依赖于操作系统	Notes 对象服务(NOS)
客户进程	浏览器、编辑器	浏览器、设计编辑器
客户扩展	插件	在基本客户系统中
服务器进程	相当于文件服务器	相当于数据库服务器
服务器扩展	Servlet、CGI 程序	服务器任务
服务器簇	透明	不透明
命名	URN、URL	URL、标识符
同步	主要是本地的	主要是本地的
缓存	高级	无文档记载
复制	镜像、CDN	懒惰
容错	可靠的通信、簇	簇
恢复	没有明显的支持	单个服务器
身份验证	主要是 TLS	证书检验
访问控制	依赖于服务器	广泛使用 ACL

图 11.35 Web 和 Lotus Notes 的比较

11.4 小 结

可以证明,基于文档的分布式系统,特别是 WWW,使得网络应用程序在终端用户中间很普及。使用文档概念作为交换信息的手段与人们通常在办公室环境和其他场合进行通信的方式很接近。每个人都理解纸张上的文档是什么,因此把这个概念扩展到电子文档对于大多数人来说都是很合乎逻辑的。

建立文档模型有不同的方法,但最重要的一点是文档之间如何相互关联。Web 和 Lotus Notes 都支持超文本模型,它对基于文档的分布式系统的普及做出了极大的贡献。在这种模型中,激活对另一个文档的引用的方法是简单地获取被引用的文档,并把它显示给用户。

一般而言,服务器维护文档并使客户访问这些文档。通过允许文档之间的引用跨服务器,即提供一个全球文档链接系统,世界范围的文档分布就变得相对简单。在最简单的情况下,一个引用包含存放文档的服务器,客户在希望获取文档时会被定向到那个服务器。

与基于文档的分布式系统的可扩展性有关的一个重要问题是文档的访问模式。文档通常仅由一个惟一的拥有者或一小组用户来维护;相比之下,它可以被许多用户读取。这种访问模式使支持缓存和复制更加容易,因为常常可以采用一种懒惰的方式来传播更新。也就是说,保证良好的性能相对简单,因为可以把文档的拷贝放在靠近用户的地方而不会引起难以处理的同步问题。

在这两种系统中,安全性与在其他任何分布式系统中同样重要。采取的措施仍然包括建立安全通道,以及在建立安全通道后执行访问控制。

习 题

1. 电子邮件在 Web 的文档模型中占有什么样的地位?
2. Web 对文档使用一种基于文件的方法。使用这种方法,客户在打开和显示文档之前首先获取整个文档文件。对于多媒体文件,这种方法会造成什么后果?
3. 为什么与非持久连接相比,持久连接通常可以改进性能?
4. 解释插件、applet、servlet 和 CGI 程序之间的区别。
5. 简述一个把 URN 转换为 URL 的命名服务的通用设计,即这种服务把一个 URN 解析为由该 URN 命名的文档的一份拷贝的 URL。
6. 在 WebDAV 中,为了获取写权限,客户只向服务器出示锁定令牌,这么做足够吗?
7. 服务器可以代替 Web 代理来计算文档的期满时间。这种方法有什么好处?
8. Akamai CDN 采用的是基于拉的分布协议还是基于推的分布协议?
9. 简述一种简单的方案。使用该方案,Akamai CDN 服务器无需检查原始服务器就可以查明一个缓存的嵌入文档是陈旧的。
10. 与使用一个或几个全局策略相比,为每个 Web 文档分别关联一个复制策略是否

有意义？

11. Notes 中的 URL 包含一个文档的标识符。与 Web 中使用的文件名相比，这种方法带来了哪些改进？
12. Notes 中的 URL 方案是否意味着可以在全世界范围内惟一地对操作进行命名？
13. 考虑一个 Domino 服务器簇。假设一个 Notes 客户第一次联系了该簇中的一个忙 Domino 服务器，解释接下来会发生什么。
14. 在本章，我们描述了两个修改过的 Notes 文档之间的冲突。而一个已删除拷贝和已修改拷贝之间也会发生冲突。解释这种冲突如何发生，并且如何解决它。
15. 在 Notes 的簇复制中，写写冲突会在什么程度上发生？
16. 给出在 Notes 中使用交叉证书的一种替代解决方法。
17. 在 Notes 中执行下载的代码通常基于信任。这样做安全吗？



第 12 章 基于协作的分布式系统

在前面几章中我们介绍了分布式系统的一些不同实现方法。实质上,这些方法都采用一种单一的数据类型作为分布的基础。这些数据类型的例子有对象、文件或文档,它们都源于非分布式系统。在分布式系统中,对它们进行了修改,以使许多有关分布的问题对用户和开发者透明。

本章中我们将考虑新一代的分布式系统。它假设系统的各个组件本质上是分布的,开发这样的系统的真正问题在于对不同组件的活动进行协作。也就是说,不把重点放在组件的透明分布上,而是放在组件间的活动协作上。

本章中介绍的一些有关协作的问题在前面的章节中,特别是在考虑基于事件的系统时已有所提及。随着时代前进,许多常规的分布式系统正在逐步结合一些在基于协作的系统中发挥重要作用的机制。

在介绍具体的系统实例前,我们简要介绍一下分布式系统中协作的概念。随后,我们将讨论两个基于协作的系统:TIBCO 的 Rendezvous Bus 和 Sun Microsystem 的 Jini。它们是目前可用的或正作为研究项目处于开发中的许多基于协作的分布式系统的代表性例子。

12.1 协作模型介绍

基于协作的系统所采用方法的关键之处在于把计算和协作分离开来。如果我们把一个分布式系统视为(可能是多线程)进程的集合,那么分布式系统的计算部分由这样的进程构成:每个进程处理一个特定的计算活动,并且原则上这个活动的执行独立于其他进程的活动。

在这个模型中,分布式系统的协作部分处理进程间的所有通信和协作。它构成把由进程执行的各个活动结合为一个整体的粘合剂(Gelernter 和 Carriero 1992)。在基于协作的分布式系统中,重点放在如何进行进程间的协作上。

Cabri 等(2000)为可同等应用于许多其他类型分布式系统的移动代理提供了一种协作模型的分类法。为了把他们的术语修改为适用于一般的分布式系统,我们从两个不同的角度(现时性和引用性)对各种协作模型进行分类,如图 12.1 所示。

如果进程之间的耦合是现时性的和引用性的,那么协作就以一种直接的方式进行,这种方式称为直接协作(direct coordination)。引用性耦合一般表现为以显式引用的形式进行通信。例如,一个进程要想和其他进程交换信息,它必须知道对方进程的名字或标识符才能与之通信。现时性耦合意味着相互通信的两个进程必须都正在运行。这种耦合与我

们在第 2 章中讨论的面向消息的暂时通信类似。

		现时性	
		耦合的	解耦合的
引用性	耦合的	直接	邮箱
	解耦合的	面向会议	生成通信

图 12.1 协作模型的一种分类法(改编自(Cabri 等 2000))

如果进程之间是现时性解耦合的,但又是引用性耦合的,那么这是另一种不同类型的协作,称为邮箱协作(mailbox coordination)。这种情况下,要进行通信,两个相互通信的进程不必同时运行,而是把消息放到一个(可能是共享的)邮箱中。这种情形类似于我们在第 2 章中描述的面向消息的持久通信。显式地指明将要存放消息的邮箱是必需的。因此,这是一种引用性耦合。

引用性解耦合系统与现时性耦合系统的结合构成了面向会议的协作(meeting-oriented coordination)模型。在引用性解耦合系统中,进程并不显式地知道对方进程。也就是说,当一个进程希望与其他进程协作活动时,它不能直接引用另一个进程,而是使用一个会议的概念,这种概念把进程临时组合在一起以进行协作活动。这种模型决定了参加会议的进程必须同时执行。

基于会议的系统往往通过事件实现,就像基于对象的分布式系统所支持的那样。在本章中,我们讨论实现会议的另一种机制,即基于主题的消息机制。这种机制允许构建称为发布/订阅系统(publish/subscribe system)的结构。在这些系统中,一些进程可以订阅包含某个特定主题信息的消息,而其他一些进程则产生(也即发布)这些消息。大多数发布/订阅系统要求相互通信的进程同时处于活动状态;因此这是一种现时性耦合。然而另一方面,通信进程也可以保持匿名。

最著名的协作模型是引用性解耦合进程和现时性解耦合进程的结合,典型的例子是由 Gelemter(1985)在 Linda 编程系统中引入的生成通信(generative communication)。生成通信的关键思想是让一个独立进程的集合共享使用一个持久的元组数据空间。元组(tuple)是带有标记的数据记录,它包含若干个(但也可能是零个)各种类型的字段。进程可以把任何类型的记录放入共享的数据空间中(即,它们生成通信记录)。与黑板(blackboard)的情况不同,这里无须预先对元组的结构达成一致。只需要使用标记来区分代表不同类型信息的元组。

这些共享的数据空间的一个有趣特性是它们执行一种元组的相关搜索机制。也就是说,当一个进程想要从数据空间中提取一个元组时,它实质上是指定自己感兴趣的字段的一些值。任何符合该标准的元组都会从数据空间中删除并传递给该进程。如果没有符合条件的元组,进程可以选择被阻塞,直到出现这样的元组为止。后面讨论 Jini 系统时我们再进行详细解释。Ciancarini 等(1999)提供了 Internet 应用环境下基于生成通信的模型的综述。

有关协作方面的大多数研究集中于编程模型及其相关语言(请参见(Papadopoulos 和 Arbab 1998))。最近的研究上升到一个更为抽象的层次,即考虑软件和系统体系结构

(Bass 等 1998, Garlan 2000)。在软件和系统体系结构方面的一个重要问题是在多个组件的相互作用中把它们区分开来,这引出了体系结构风格和体系结构描述语言的概念。有关这些主题的更多信息可以在(Shaw 和 Clements 1997, Medvidovic 和 Taylor 2000)中找到。

12.2 TIB/Rendezvous

作为基于协作的分布式系统的第一个例子,我们介绍由 TIBCO 开发的 TIB/Rendezvous 系统。在(TIBCO 2000b)中可以找到该系统的概述,在各种系统附带的手册中有详细的说明。

12.2.1 TIB/Rendezvous 概述

TIB/Rendezvous 系统最初是作为一种信息总线(Skeen 1992, Oki 等 1993)描述的。信息总线(information bus)是一个进程集合的最小化通信系统,它基于下述设计原则。首先,核心通信系统是高度独立于应用的。例如,它不考虑复杂的消息排序语义,因为这类问题应当在应用层进行处理(Cheriton 和 Skeen 1993)。在第 5 章中我们已经碰到过这个问题。与此类似,基本系统没有内置对原子事务的支持。我们将看到,这种支持通过一个附加服务来提供。

第二个设计原则是消息是自描述的。实际上,这意味着一个应用程序可以通过查看一个输入的消息来了解它的结构以及它所包含的数据类型。请注意,相比之下,大多数通信系统的做法是假设一个进程已经知道一个输入消息的格式,以便它可以正确地解释消息的内容。

第三个设计原则是进程应该是引用性解耦合的。之所以提出这个原则是由于这样的要求:一个运行的系统在维护时也要能够正常工作,并且与此类似,它应该能够轻松地动态添加新的进程。如果进程并不显式地引用其他进程,那么这些要求可以更容易地得到满足。引用性解耦合通过基于主题的寻址来实现,下面我们就将进行说明。

1. 协作模型

TIB/Rendezvous 基于这样的协作模型:其中的进程主要是引用性解耦合和现时性耦合的。这种模型在上文中称为面向会议的协作。该系统还提供允许进程现时性解耦合的服务。这一模型的详细情况将在后面讨论。

TIB/Rendezvous 协作模型的基本关键思想是基于主题的寻址(subject-based addressing)。在这种方法中,希望发送消息的进程并不指定消息实际的目的地,而是为消息标记一个主题名称(subject name),然后把它传递给通信系统以在网络上传播。

接收者也并不指定它们准备接受的输入消息来自哪些进程,而是告诉通信系统它们对什么主题感兴趣。于是,通信系统保证只有那些携带关于一个接收者感兴趣主题的数据的消息才会传递给该接收者。

使用基于主题的寻址方法发送消息也称为发布(publishing)。要接收某个特定主题

的消息,一个进程必须订阅(subscribe)该主题。使用基于主题的寻址的发布/订阅系统的原理如图 12.2 所示。

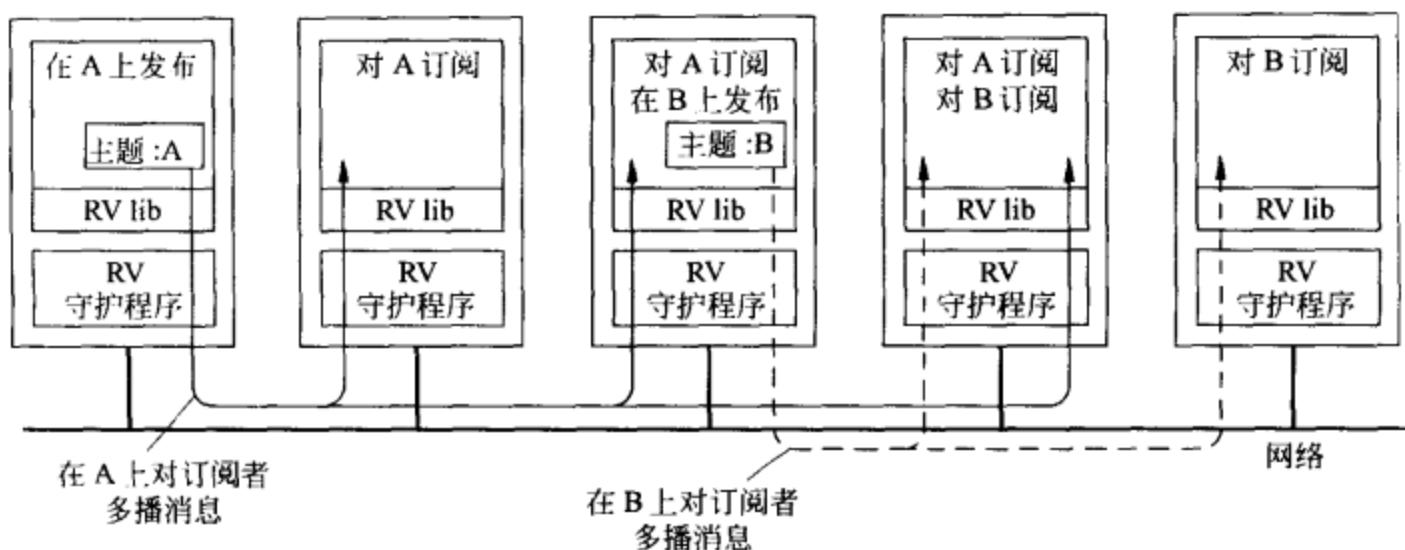


图 12.2 TIB/Rendezvous 中实现的发布/订阅系统的原理

2. 体系结构

TIB/Rendezvous 系统的体系结构相对而言比较简单。其实现的基础是一个多播网络的使用,不过如果可能,它也会使用效率更高的通信方法。例如,如果确切地知道一个订阅者所在的位置,通常就会使用点到点的消息。在这样的网络中,每个主机都将运行一个 Rendezvous 守护程序(rendezvous daemon),它负责根据消息的主题发送和传递消息。无论何时发布了一个消息,都会把它多播到网络上每个运行 Rendezvous 守护程序的主机。一般来说,多播使用底层网络提供的功能实现,例如 IP 多播或硬件广播。

订阅某个主题的进程把它们的订阅传递给它们本地的守护程序。守护程序构造一个进程,主题项目表。当一个关于主题 S 的消息到达时,守护程序简单地查看自己表中的本地订阅者,并把这个消息转发给每个订阅者。如果没有 S 的订阅者,将立即丢弃这个消息。

为了把该系统扩展到像 WAN 这样的更大的网络中,需要使用 Rendezvous 路由器守护程序(renderzvous router daemon)。通常,每个本地网络具有一个路由器守护程序。这个守护程序与其他远程网络的路由器守护程序进行通信,如图 12.3 所示。这些路由器守护程序一起形成了一个点到点的覆盖网络(overlay network),其中路由器对相互之间的连接通过一个 TCP 连接完成。覆盖网络是路由器的应用层网络。我们在第 2 章中讨论消息队列系统时已经遇到过这种类型的网络。

每个路由器都知道覆盖网络的拓扑结构,并计算出一个多播树用来向其他网络发布消息。一个路由器只对那些在它所代表的本地网络上发布的消息进行多播。来自其他网络的消息由路由器沿着该消息起源网络的多播树进行转发。

出于性能的原因,无论何时发布一个消息,它只会传递给那些已显式地配置为接受这类消息并且当前具有该消息的订阅者的远程网络。下面会详细讨论这些通信限制。

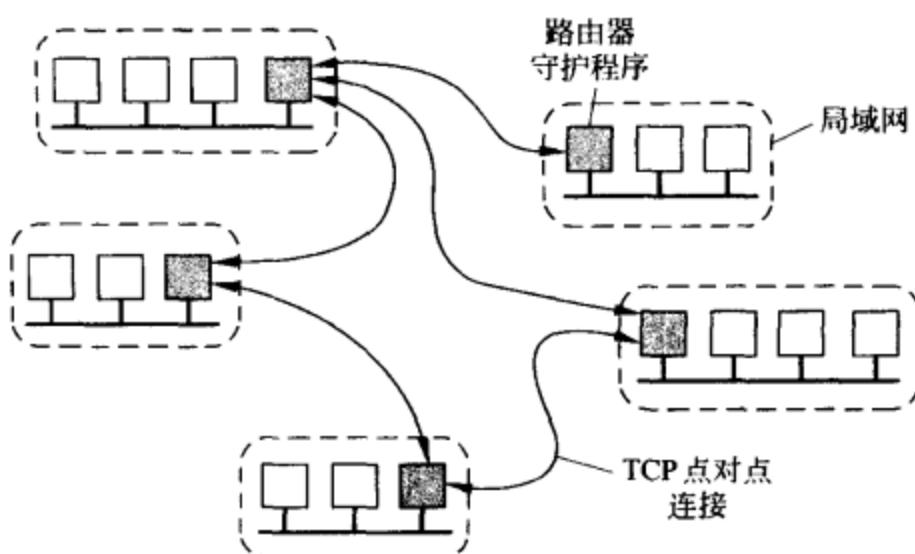


图 12.3 广域 TIB/Rendezvous 系统的总体结构

12.2.2 通信

TIB/Rendezvous 的基础是其通信功能。我们已经简要地解释了消息机制的工作原理,本节将深入进行介绍,并将说明它对广域通信的支持。发布/订阅方法很适合基于事件的编程,TIB/Rendezvous 也支持后者。本节也将讨论事件。

1. 基本通信机制

我们已经提到过,TIB/Rendezvous 中的通信通过使用基于主题的寻址发送和接收自描述的消息来进行。每个消息由一些字段(可能为零)组成,这些字段是由发送进程动态构造的。一个字段本身是具有如图 12.4 所示的属性的一个记录。

属性	类型	说明
Name	字符串	字段的名称,有可能是 NULL
ID	整数	一个在消息内惟一的字段标识符
Size	整数	字段的总大小(以字节为单位)
Count	整数	如果是数组,该属性是元素的个数
Type	常量	指示数据类型的常量
Data	任何类型	字段中存储的实际数据

图 12.4 TIB/Rendezvous 消息字段的属性

每个字段通常具有一个字符串类型的“Name”(名称)属性以简单地命名该字段。同一个消息中的不同字段可以具有相同的名称。但是,也可以通过使用“ID”属性使一个字段和一个单独的标识符相关联。一个字段标识符是一个 2 字节的整数值,该数值对于每个消息必须是惟一的。

字段的总大小由“Size”(大小)属性以字节为单位给出。如果字段由一个数组组成,那么“Count”(计数值)属性包含该数组中的元素个数。

“Type”(类型)属性是一个代表 TIB/Rendezvous 原始数据类型之一的常量。当字段的实际数据是一个数组时,该属性指明数组中每个元素的原始数据类型。TIB/Rendezvous 还提供用于交换自定义数据类型的功能。最后,“Data”(数据)属性包含字段中的实际数据。

每个消息可以构造为具有任意数目的字段。在发送一个消息之前,必须通过调用一个单独的操作来使它与一个主题相结合。一个主题本身由另一个字符串名称来表达。还可以使一个消息与一个回复主题相结合。接收者可以使用这个主题名称向发送进程返回一个回复消息。当然,发送者必须订阅它自己的回复主题。

出于性能的原因,每个进程可以使用一个专门创建的、特定于进程的主题名称,这样的主题名称叫作收件箱名称(inbox name)。如果一个发布者 P 在发布消息时使用了另一个进程 Q 的收件箱名称,那么这个消息就会通过点到点通信功能而不是多播方式直接发送给进程 Q。

为了发送和接收消息,进程需要创建所谓的传送器。一个传送器(transport)是一个本地对象,它类似于 Berkeley UNIX 中引入的套接字(我们在第 2 章中曾讨论过)。一个传送器用来向使用一个特殊的通信协议监听一个特定端口的 Rendezvous 守护程序发送消息。例如,有用于使用广播、链接级网络多播,或 IP 多播的不同传送器。

与传送器相关联的通信原语只有三个。send 操作接收一个消息并把它传递给本地的 Rendezvous 守护程序,后者将负责正确地发布这个消息。send 是非阻塞的操作。

与之类似的是非阻塞的 sendreply 操作,该操作可以在接收到一个包含前述的回复主题的消息时调用。

最后一个操作是阻塞的 sendrequest 操作。该操作向本地 Rendezvous 守护程序传递一个消息,后者把这个消息传播到底层网络中;然后该操作阻塞调用线程,直到收到一个回复消息。回复消息通过点到点通信发送回一个(动态创建的)收件箱名称。

请注意,没有单独的 receive 操作。我们后面还会解释到,消息接收主要通过事件的方法处理。当一个包含某进程已订阅的主题的消息到达时,TIB 的事件系统会调用一个该订阅进程提供的回调函数。接下来会详细讨论这种方法。

2. 事件

为了接收有关进程已订阅主题的消息,订阅者使用一种事件机制。原则上,没有处理输入消息的其他方法,不过 sendrequest 操作构成了该规则的一个例外。

订阅一个主题是通过创建一个监听器事件(listener event)完成的。监听器事件是一个本地对象,它与一个传送器和创建它的进程正在订阅的主题相关联。它还包含一个对订阅者提供的回(callback)函数的引用。当事件被分派(dispatched)时将调用这个回调函数。输入消息的事件分派按照下述原理工作,图 12.5 中也示意了该原理。

当一个符合某个监听器事件指定的主题的消息到达时,TIB/Rendezvous 系统会创建一个相应的事件对象(event object),并把它追加到一个本地的事件队列中。这样的事件对象所包含的信息与监听器事件相同。后者继续监听更多的输入消息。每个事件队列具有至少一个与之相关联的分派器线程,该线程负责删除队列头部的事件,并调用相关的回

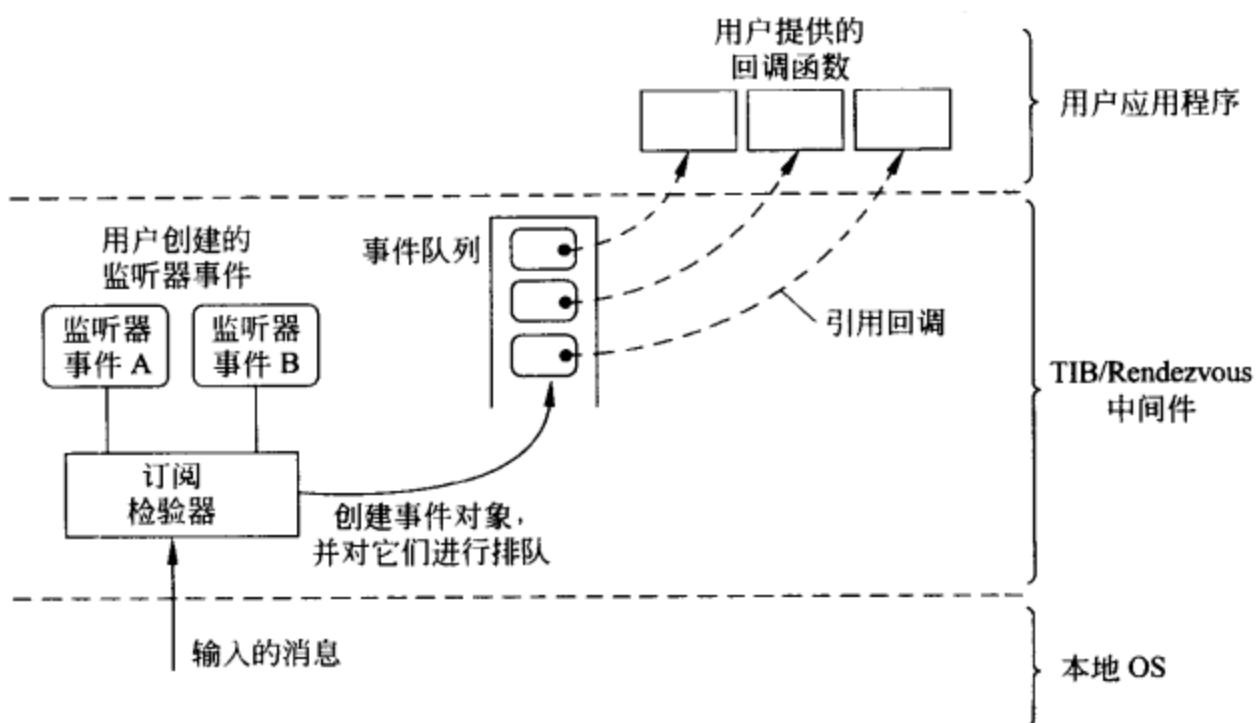


图 12.5 TIB/Rendezvous 如何处理订阅的监听者事件

调函数。在调用回调函数时,与被分派的事件相关的输入消息会传递给该函数。

如果符合一个监听器事件指定的主题的另一个消息到达,那么会创建另一个事件对象,并把它追加到本地队列中。这与分派器是在等待新的事件对象还是仍在忙于处理前面的事件无关。要取消一个订阅,订阅进程只须销毁相应的监听器事件。任何未处理的与该监听器对象相关的事件对象也将销毁。图 12.6 示意了对三个符合同一监听器事件的输入消息采取的活动。每个水平线代表一个活动(例如一个订阅或处理一个消息)的时间间隔。

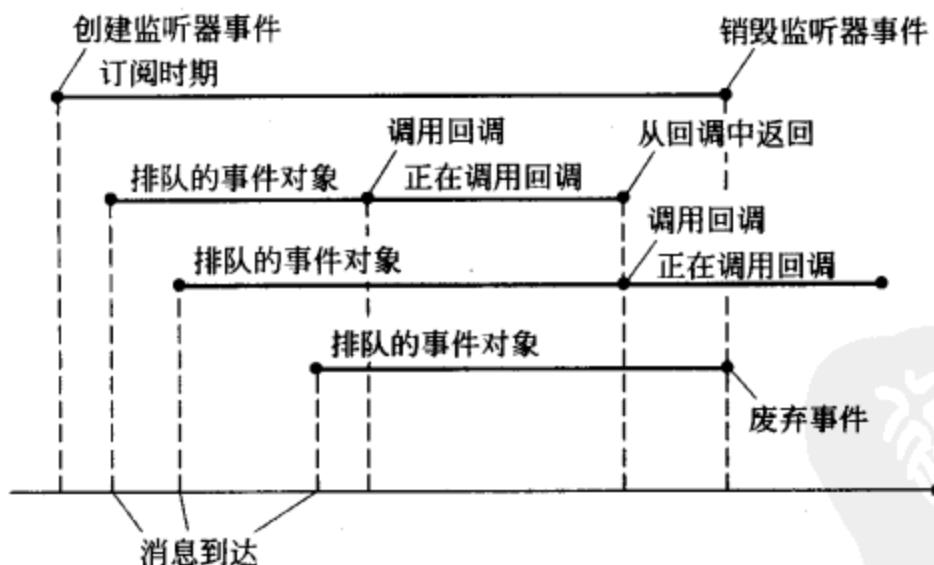


图 12.6 TIB/Rendezvous 如何处理到来的消息

如果一个输入消息符合两个或更多个监听器事件指定的主题,那么会为每个监听器事件创建一个相应的事件对象并把它们追加到队列中。也就是说,一个输入消息有可能导致产生多个事件对象。

默认的情况是使用一个队列处理所有事件。但是,为了允许对事件的处理进行粒度精细的控制,也可以创建多个队列并显式地把事件与事件队列相关联。因为每个事件队列具有它自己的相关的分派器线程,所以使用多个事件队列允许事件对象的交替处理。

而且,还可以把多个事件队列组合为一个队列组(queue group)。可以给一个队列组中的每个队列赋予一个优先级。队列组的分派器总是从具有最高优先级的队列中开始寻找事件对象。如果那个队列为空,则继续检查优先级其次的队列。图 12.7(a)示意了在一个队列组中组织 4 个事件队列。针对图中所示的特定事件对象集合,图 12.7(b)示意了一个语义上等价的事件队列。

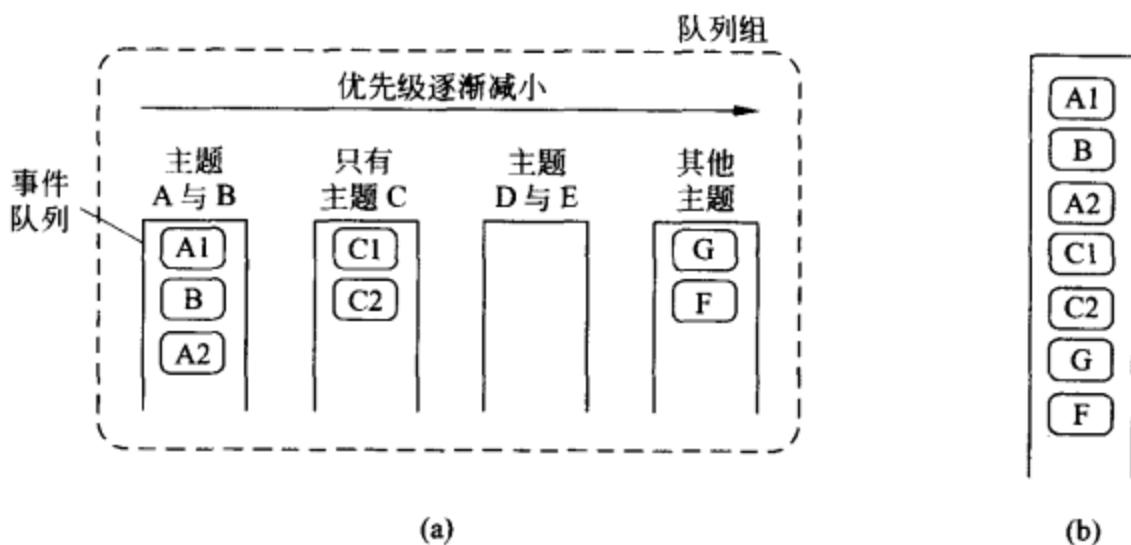


图 12.7 事件优先级安排和等价队列

(a) 通过一个队列组安排事件的优先级; (b) 与(a)中的具有特定事件对象的队列组在语义上等价的队列

如果没有任何事件对象,那么分派器将会阻塞。当一个选中的事件对象的回调函数被调用时,分派器会一直运行到调用结束。当一个新的事件对象被追加到一个队列,而该队列具有比当前选中的事件对象所在的队列更高的优先级时,并不发生先占现象。

12.2.3 进程

TIB/Rendezvous 系统的基本组织是相当简单的。每个主机通常运行一个 Rendezvous 守护程序,该程序负责处理所有的网络通信,正如我们前面解释的那样。如果需要,会用一个路由器守护程序代替 Rendezvous 守护程序。路由器守护程序可以处理广域的通信,它扩展了 Rendezvous 守护程序的功能。

每个客户程序作为一个单独的可执行进程运行在其主机上。所有通信通过与客户应用程序相链接的一个库进行处理,这个库将把消息传递给本地守护程序。有关该库和实际的 C 编程接口的详细信息请参见(TIBCO 2000a)。

12.2.4 命名

本书迄今为止所讨论的名称都用于指向明显存在于分布式系统中并被其管理的实体。典型的例子如资源、进程、文件、网络连接等。

诸如 TIB/Rendezvous 这样的发布/订阅系统的一个关键方面是使用主题名称进行寻址。这里的使用名称进行寻址和我们前面讨论的有些不同。主题名称并不显式地指向

存在于系统中并需要系统管理的某个实体,而是用来把一组发送进程(即发布者)和一组接收进程(即订阅者)进行匹配。

在 TIB/Rendezvous 中,主题名称是用点号分隔的字符串标签(label)的序列,它与 DNS 名称类似。主题名称必须总是以一个标签开始,并以一个标签结束;不允许有空标签。一个主题名称最多有 255 个字符,而一个标签最多有 252 个字符。图 12.8 中是一些合法和非法的主题名称的不同例子。

例子	是否合法?
Books. ComputerSystems. DistributeclSystenns	是
. ftp. cs. vu. nl	否(以一个“.”开始)
ftp. cs. vu. nl	是
NEWS. res. comp. os	是
Maarten. . vanSteen	否(空标签)
Maarten. R. vanSteen	是

图 12.8 合法和非法的主题名称的例子

订阅者可以使用通配符指定他们希望接收的消息所具有的主题范围。有两种通配符。在一个标签所在的位置使用一个星号(“*”)表明任何标签都可以放在这个位置。在一个标签所在的位置使用大于号字符(“>”)表明任何标签都可以作为主题名称的剩余部分。请注意,通配符总是用来代替一个标签,它们不能用作标签的一部分。使用通配符的例子如图 12.9 所示。

主题名称	匹配
*. cs. vu. nl	ftp. cs. vu. nl www. cs. vu. nl
nl. vu. >	nl. vu. cs. ftp nl. vu. cs. zephyr nl. vu. few. www
NEWS. comp. *. books	NEWS. comp. os. books NEWS. comp. ai. books NEWS. comp. se. books NEWS. comp. theory. books

图 12.9 在主题名称中使用通配符的例子

可以通过使用通配符把路由器守护程序配置为对输入或输出的消息进行过滤。对于输入的消息,这意味着路由器守护程序订阅一个主题集合,只有属于该主题集合的消息才能传递给本地网络。因此,如果一个客户程序订阅了所有主题(通过使用主题名称“>”可以做到这一点),但是路由器守护程序只接受 Network News 消息(例如,通过“NEWS. >”指定),那么这个客户只能接收到 News 消息。(为了实际触发这样的消息的接收,应当显式地告知路由器守护程序,宣布对主题“NEWS. >”感兴趣。)

与此类似,也可以限制输出的消息。实际上,路由器守护程序订阅一个输出消息的集合。符合它的订阅规定的每个消息都会传递给远程网络的路由器,而其他的消息则会保留在本地网络中。

这种对输入和输出消息的双向过滤可以极大地降低广域网的流量,在本质上它类似于 Network News 协议 NNTP(Kantor 和 Lapsley 1986, Barber 2000)中使用的洪泛算法。注意,这种多播方案假设主题是预先知道的。(Banavar 1999b)中描述了使用消息内容而不是预定义主题来匹配发布者和订阅者的一种解决方案。

12.2.5 同步

TIB/Rendezvous 核心对进程同步仅提供很少的支持。它做出的惟一保证是,来自每个消息来源的消息都以先进先出(FIFO)的顺序传递,即来自相同传送器的消息按照发送它们的顺序传递。

作为对核心的补充,TIB/Rendezvous 提供一个单独的事务服务以支持事务消息机制(transactional messaging)。在这种机制中,发送和接收消息可以作为事务的一部分完成。事务消息机制作为位于 TIB/Rendezvous 系统核心上层的一个单独的层实现,如图 12.10 所示。一个事务中的操作往往只来自一个进程。事务层不支持把不同进程的操作组合为一个事务,但通过使用一个单独的事务管理器(就像我们在第 7 章中所讨论的)可以处理这种情况。

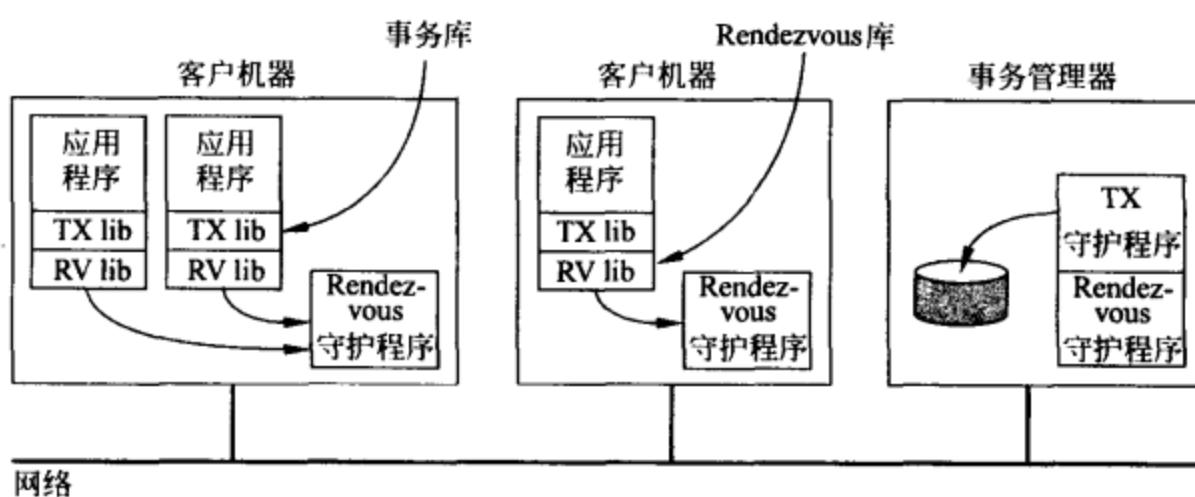


图 12.10 把事务消息机制组织为 TIB/Rendezvous 中的一个单独的层

事务守护程序(transaction daemon)在事务中起着重要的作用,它主要负责存储一个已发布的消息,直到该消息传递给所有订阅者。可以有多个事务守护程序,每个程序处理属于某个特定主题集合的消息。只有那些作为事务的一部分发送的消息才会存储。下面解释这个模型。

一个进程 P 也许会决定把一些发布和订阅操作组合为一个事务,如图 12.11 所示。一个事务开始时会向各种事务守护程序发送一个消息,明确告诉它们为 P 打开一个会话,以便作为事务的一部分发送和接收消息。随后,无论何时 P 发布一个主题为 S 的消息,这个消息会被发送给负责处理主题 S 的守护程序;该守护程序会保存这个消息,直到 P 提交或中止该事务。注意,直到提交事务时,发布的消息才会被发送给其他进程。

如果 P 订阅了存储在一个事务守护程序中的已提交事务的消息,那么这些消息会照常转发给 P。像前面提到的那样,尚未提交的事务的消息保存在守护程序中。当 P 接收到一个消息时,它把这一事实通知给与该消息相关联的事务守护程序。这个通知是保证

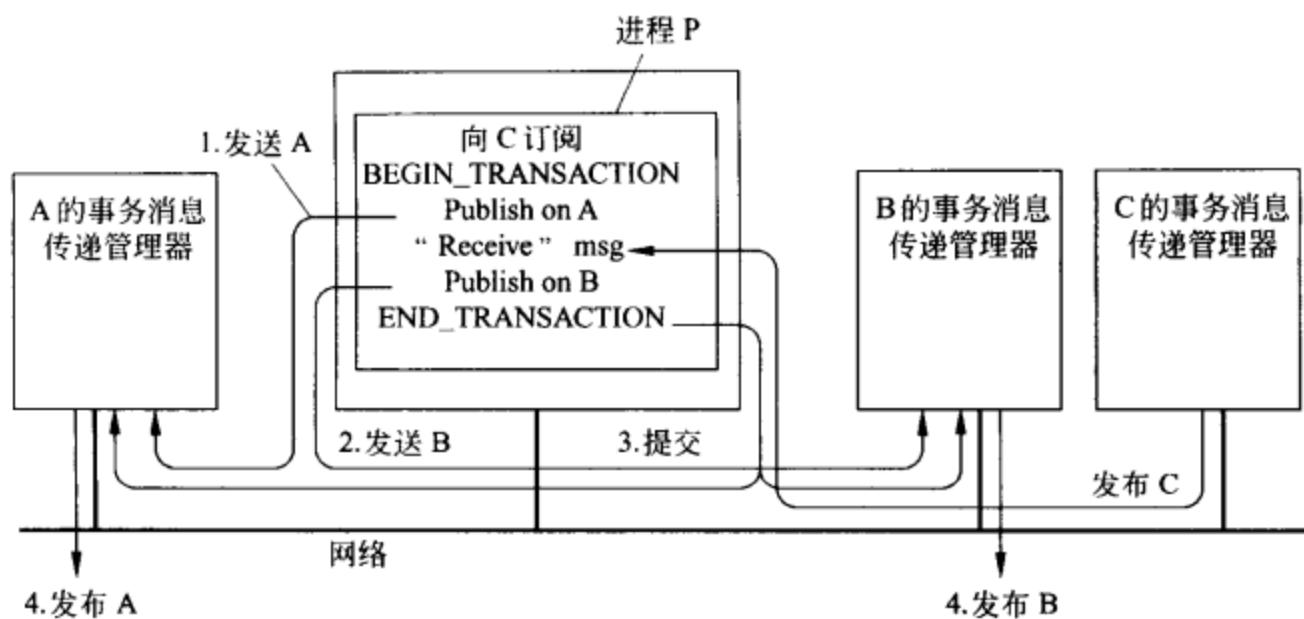


图 12.11 TIB/Rendezvous 中一个事务的结构

P 在提交事务后不会再次接收到相同的消息所必需的。

只有当一个事务提交后该事务中发送的消息才对订阅者可用。如果中止了一个事务,那么它发送的消息会被事务守护程序简单地丢弃,而不会被发布。来自一个被提交事务的消息在提交该事务时被转发给它的订阅者。没有订阅者的消息将被丢弃(即,不管它们是否参与事务消息机制)。在所有订阅者提交之前消息一直保存在守护程序中。当一个已发布的消息存储在守护程序中,在该消息被与其相关的事务提交之前,就可以接受新的订阅。

当一个事务中涉及多个事务守护程序时,TIB/Rendezvous 使用一个两阶段的提交协议来实际提交或中止事务。

事务消息机制可以与数据库操作相结合,但也可以与非事务消息机制相结合。这种灵活性可以轻易地模糊 TIB/Rendezvous 中与事务相关的语义。我们不打算对此进行深入讨论,详细内容请参见(TIBCO 2000c)。

12.2.6 缓存和复制

几乎没有什么专门的功能用于支持缓存和复制。这些问题需要由应用程序来全权处理。特别是,需要采取专门的措施以防止来自被复制进程的相同消息重复发布。TIB/Rendezvous 独立地对待这些消息并把它们每一个都传递给订阅者。

为了支持新的订阅者,可以运行一个后台进程,用作缓存有关某个特定主题的最后 n 个消息的服务器。当一个进程刚刚开始时,它可以查询缓存服务器以取得有关某个特定主题发送的最新消息,从而加速该主题消息的处理。

12.2.7 容错性

TIB/Rendezvous 以两种方式支持容错。首先,它提供多种功能来实现可靠的通信,这样就确保了已发布的消息不会由于通信故障而丢失。其次,它通过支持进程组来防止

进程的崩溃性故障。下面我们讨论这些功能。

1. 可靠的通信

TIB/Rendezvous 假设底层网络的通信功能本质上是不可靠的。为了弥补这一不可靠性，无论何时一个 Rendezvous 守护程序向其他守护程序发布一个消息时，它会把这个消息至少保存 60 秒。当发布一个消息时，守护程序把一个与主题无关的序列号附加到这个消息上。接收消息的守护程序通过查看序列号可以检测是否有消息丢失（回忆一下，消息会传递给所有守护程序）。当丢失了一个消息时，TIB/Rendezvous 会请求发布消息的守护程序重新传输该消息。

这种形式的可靠通信仍然不能防止消息的丢失。例如，如果一个接收消息的守护程序请求重新传输一个早在 60 秒之前发布的消息，则发布消息的守护程序对恢复这个已丢失的消息通常是无能为力的。在正常的情况下，TIB/Rendezvous 会通知负责发布和订阅的应用程序发生了一个通信错误，而错误可以留给应用程序以后处理。

TIB/Rendezvous 中通信的可靠性主要由其底层网络提供。最近，TIB/Rendezvous 也提供了可靠的多播功能，使用（非可靠的）IP 多播作为底层的通信手段。TIB/Rendezvous 所遵循的方案是一种称为 PGM(Pragmatic General Multicast) 的传输层多播协议，(Speakman 等 2001) 中有对该协议的描述。我们将对 PGM 进行简要讨论。

PGM 并不绝对保证多播的消息最终一定能交付给每个接收者。在图 12.12(a) 中，一个消息沿树进行多播，但是它并没有交付给两个接收者。PGM 依靠接收者来检测出它们丢失了消息，并向消息的发送者发送一个重新传输的请求(NACK)。该请求沿着以消息发送者为根的树中的反向路径发送，如图 12.12(b) 所示。当一个重新传输消息的请求到达一个中间节点时，该节点可能已缓存了该消息，此时它将重新传输该消息。否则，该节点只是把 NACK 消息转发给通向发送者的下一个节点。发送者最终负责重新传输消息。

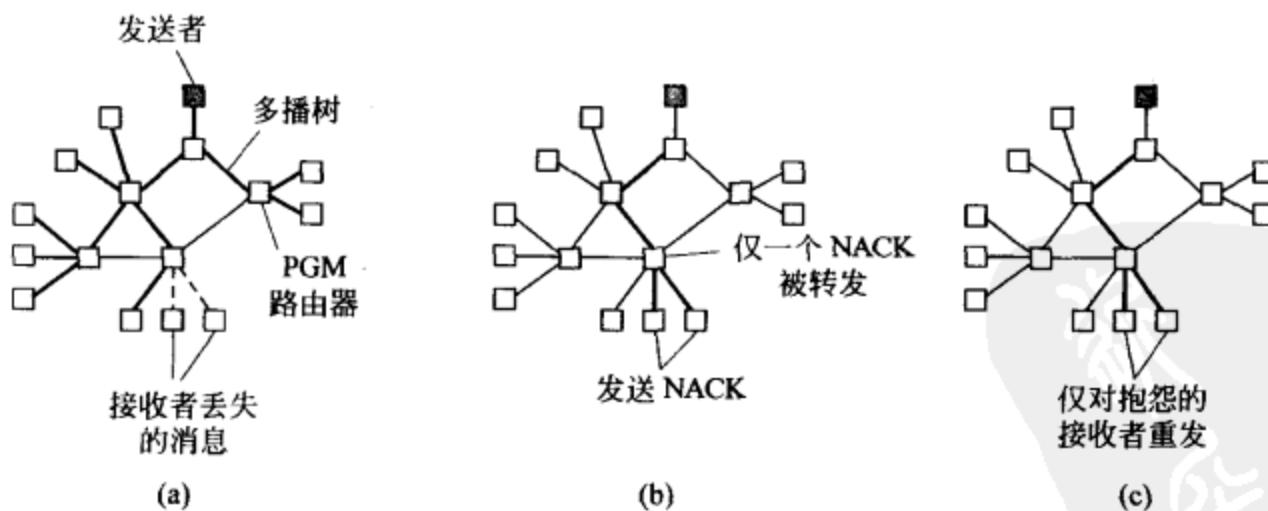


图 12.12 PGM 原理

(a) 沿多播树发送一个消息；(b) 路由器只为每个消息传递一个 NACK；(c) 消息只重新传输给发出请求的接收者

PGM 采用许多措施来为可靠的多播提供可扩展的解决方案。首先，如果一个中间节点接收到对同一个消息的多个重新传输请求，它只把一个重新传输请求转发给发送者。

通过这种方式来确保只有一个 NACK 会到达发送者,从而避免出现反馈爆炸的现象。在第 7 章讨论可靠多播的可扩展性问题时我们已经遇到过这个问题。

PGM 采取的另一个措施是记录下 NACK 从接收者到发送者所走过的路径,如图 12.12(c)所示。当发送者最后重新传输被请求的消息时,PGM 使得消息只多播给那些发出了重新传输请求的接收者。因此,已经成功接收到消息的接收者就不会受到干扰。

除了基本的可靠性方案和通过 PGM 实现的可靠多播外,TIB/Rendezvous 还通过经证明的消息交付(certified message delivery)提供更高的可靠性。此时,进程使用一个专门的传送器来发送或接收消息。该传送器具有一个相关的账簿(ledger),用于记录已发送和已接收的经证明的消息。希望接收这样的消息的进程在发送者那里进行注册。实际上,这种注册允许传送器处理更多的 rendezvous 守护进程不提供支持的可靠性问题。这些问题中的大多数由传送器处理,并对应用程序隐藏。

当把账簿实现为一个文件时,那么即使在进程出现故障的情况下也可提供可靠的消息交付。例如,当一个接收进程崩溃时,在它恢复前丢失的全部消息都会存储在发送者的账簿中。一旦恢复,接收者只需与账簿联系,并请求重新传输丢失的消息。

2. 容错进程组

为了能够隐藏进程故障,TIB/Rendezvous 提供了一种自动激活进程或停用进程的简单手段。这里,活动的进程正常情况下会对所有输入的消息做出响应,而不活动的进程则反之。不活动的进程是只能处理特殊事件的运行中进程。

可以把进程组织为进程组,其中的每个进程都有一个与之相关联的惟一的级别。进程的级别由其权重(手工赋值)决定,但是同一组中的两个进程不能具有相同的级别。对于每个组,TIB/Rendezvous 试图令其中的一些进程处于活动状态,这些进程的数目是特定于组的,称为该组的活动目标(active goal)。在许多情况下,把活动目标设置为 1,从而使得与每个组的所有通信简化为第 6 章中讨论的原始协议。

活动的进程一般会向组中的所有其他进程发送一个消息,以宣布自己仍然在运行。当不再发出这样的心跳消息(heartbeat message)时,TIB/Rendezvous 中间件将自动激活当前不活动的最高级别的进程。激活通过回调一个 action 操作完成,每个组成员都应实现该操作。同样,当一个先前崩溃的进程重新恢复并变活动时,当前最低级别的活动进程将自动变为不活动状态。

为了与活动进程保持一致,不活动进程在变为活动进程前需要采取特别的措施。一种简单的方法就是让不活动的进程订阅与任何其他组成员相同的消息。它会以通常的方式处理输入的消息,但是并不发布任何反应。这种方法与我们在第 6 章中讨论的主动复制极为相像。

12.2.8 安全性

TIB/Rendezvous 中的安全性问题主要是在发布者和订阅者之间建立一个安全通道。使用这种方法的一个后果是丧失了发布/订阅系统的一个重要的特性,即发送者和接收者之间的引用性解耦合关系。这样,模型就简化为我们以前解释的直接协作模型。在需要

安全通道这样的相互验证时,发送者和接收者之间就不可避免地需要一种更为紧密的耦合关系。

在 TIB/Rendezvous 中建立安全通道是从发布加密数据开始的。发布的数据包含发送者的身份,以允许发送者和接收者之间的进一步协商。如果把一个消息发布给多个订阅者,那么每个订阅者都需要与发送者建立一个安全通道。用于解密发布消息的密钥对于所有订阅者是相同的。

例如,在 Alice 和 Bob 之间建立一个安全通道是基于 Diffie-Hellman 密钥交换和公钥系统的。通道的建立过程如图 12.13 所示。我们假设 Alice 和 Bob 已经得到了包含彼此公钥的证书。正如我们将看到的,根据安全通道的建立协议,这些证书要得到验证。我们还假设 Alice 和 Bob 已经使用 Diffie-Hellman 密钥交换协议(在第 8 章中讨论)建立了一个秘密的共享密钥 $K_{A,B}$ 。

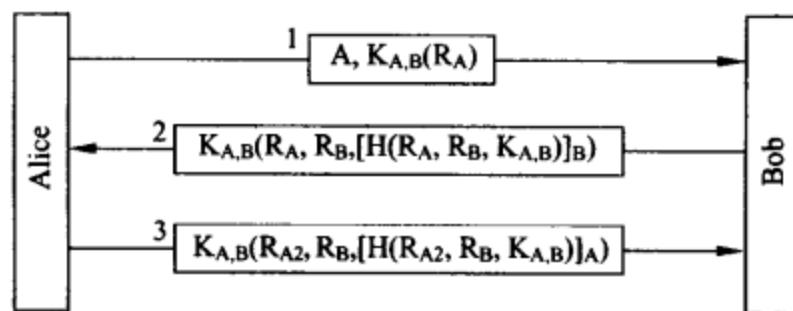


图 12.13 在 TIB/Rendezvous 中建立安全通道

为了建立安全通道,Alice 和 Bob 需要相互进行身份验证。Alice 首先向 Bob 发送 R_A ,并用共享密钥 $K_{A,B}$ 加密,如图 12.13 中的消息 1。作为响应,Bob 向 Alice 发送 R_B 。为此,他计算散列值 $H(R_A, R_B, K_{A,B})$,并用他的私钥签名。签名后的散列和 R_A, R_B 一起再用 $K_{A,B}$ 加密后返回给 Alice,如图 12.13 中的消息 2。

当 Alice 接收到消息 2 时,她可以通过计算散列值并将其与 Bob 签名后的值相比较来对 Bob 进行身份验证。为此,她需要检验 Bob 的证书,其中包含有他的公钥。检验证书包含检查该证书是否由 Alice 信任的证书机构签名。

在对 Bob 进行身份验证之后,Alice 也需要得到验证。因此,她使用 R_{A2} 构造一个与 Bob 的消息类似的消息,即图 12.13 中的消息 3。随后,Bob 就可以对 Alice 进行身份验证,从而完成安全通道的建立过程。

12.3 Jini

接下来要介绍的基于协作系统的例子是 Sun Microsystem 的 Jini。把 Jini 作为基于协作的系统主要是由于它支持生成通信,方法是使用一种称为 JavaSpace 的类 Linda 服务,这也是我们重点要讨论的内容。另外,它还支持让客户容易地发现可用的服务,就像在一个分布式事件和通知系统中那样。Jini 还提供许多其他服务和功能,而不仅仅是一个基于协作的系统。不过,从刚刚提及的这些服务来看,可以把它作为基于协作的分布式系统的案例研究。Sun Microsystem 提供关于 Jini 的详细说明,这些描述在 (Sun

Microsystems 2000b, Waldo 2000)中可以找到。Keith(2000)对 Jini 的核心功能给出了很好的介绍。

12.3.1 Jini 概述

Jini 是一个包含各种相关元素的分布式系统。它与 Java 编程语言结合紧密,不过它的许多原理在其他语言中都可以同样实现。该系统的一个重要部分由生成通信的协作模型构成。我们首先讨论这一模型,然后给出一个典型 Jini 系统的整体体系结构。

1. 协作模型

Jini 通过一个称为 JavaSpace 的类 Linda 协作系统来提供进程间的现时性和引用性解耦合。一个 JavaSpace 就是一个共享的数据空间,用于存储代表对 Java 对象的引用的有类型集合的元组。在一个 Jini 系统中可以有多个 JavaSpace 同时存在。

元组(tuple)以串行化的形式存储。也就是说,当进程存储一个元组时,首先要对该元组进行编组,这意味着它的所有字段都将被编组。这样,当一个元组包含两个引用同一对象的不同字段时,存储在一个 JavaSpace 实现中的该元组将包含该对象的两个已编组的副本。

通过一个 write 操作可以把元组放入 JavaSpace 中,该操作首先对元组进行编组,然后再存储之。每当对一个元组调用 write 操作时,这个元组的另一个已编组的拷贝会被存储到 JavaSpace 中,如图 12.14 所示。我们把每个已编组的拷贝称为一个元组实例(tuple instance)。

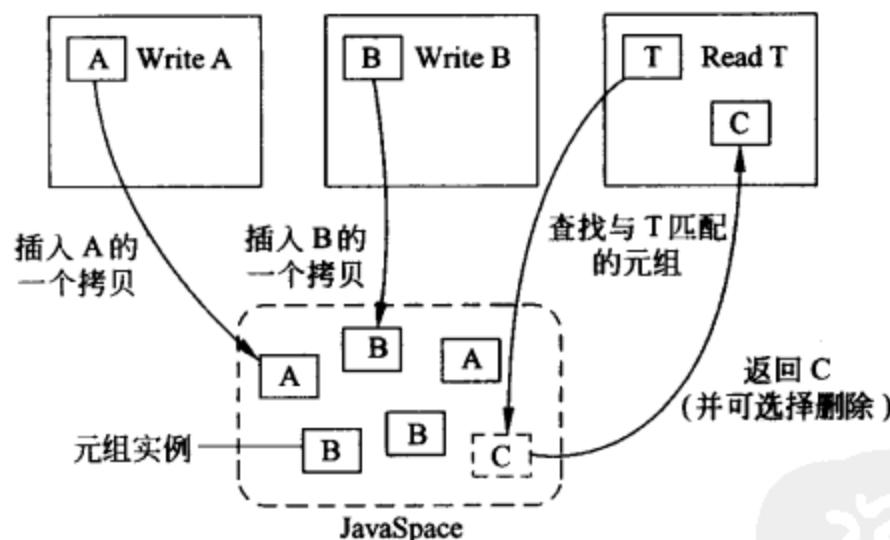


图 12.14 Jini 中的一个 JavaSpace 的总体结构

Jini 中的生成通信有趣的一面是从 JavaSpace 中读取元组实例的方式。为了读取一个元组实例,进程会提供另一个元组作为其模板(template),以对 JavaSpace 中存储的元组实例进行匹配。正如任何其他元组一样,模板元组也是对象引用的有类型集合。只有那些与模板元组类型相同的元组会被从 JavaSpace 中读出。模板元组中的一个字段可以包含对一个实际对象的引用,或是包含值 NULL。

要把 JavaSpace 中的一个元组实例与一个模板元组相匹配,需要先把模板元组像一

般元组那样进行编组(包括它的 NULL 字段)。与模板类型相同的每个元组实例和已编组的模板元组之间会进行逐字段的比较。如果两个字段具有相同的引用拷贝或者模板元组中的字段为 NULL,那么这两个字段相匹配。如果一个元组实例和模板元组的各个字段分别匹配,那么这个元组实例和模板元组相匹配。

当发现一个元组实例与一个 read 操作提供的模板元组相匹配时,该元组实例将被解编并返回给读取进程。还有一个 take 操作负责从 JavaSpace 中删除元组实例。这两个操作在找到一个匹配的元组实例之前都会阻塞调用程序。可以指定一个最大阻塞时间。另外,在不存在匹配元组时还可以简单地立即返回一些变量。

与 TIB/Rendezvous 采用的发布/订阅模型相比,使用 JavaSpace 的进程无需同时存在。实际上,如果 JavaSpace 是使用持久存储实现的,那么可以停止一个完整的 Jini 系统并在稍后重新启动,而不会丢失任何元组实例。但不幸的是,采用生成通信模型需要付出一定的代价。正如实践表明的那样,开发 JavaSpace 或类 Linda 系统的高效实现比较困难。在广域网中实现生成通信模型时通常会遇到可扩展性的问题。下文我们会回来讨论这个问题。

2. 体系结构

JavaSpace 只是 Jini 系统的一部分。正如 TIB/Rendezvous, Jini 的目标是提供一个有用的设备和服务的小型集合,以便允许构建分布式应用。人们通常把使用 Jini 的分布式应用描述为设备、进程和服务的一个松散联合。当前 Jini 系统中的所有通信都基于 Java RMI。

Jini 系统的体系结构可以视为图 12.15 所示的三层。最底层由 Jini 基础设施构成,它提供 Jini 的核心功能,即允许通过 Java RMI 进行通信的功能。Jini 模型的一个重要方面是客户可以轻松地找到服务。服务可以由常规的进程提供,但也可以由不能运行 Jini 软件的硬件设备(即 Java 虚拟机)提供。因此,注册和查找服务也属于 Jini 的基础设施。

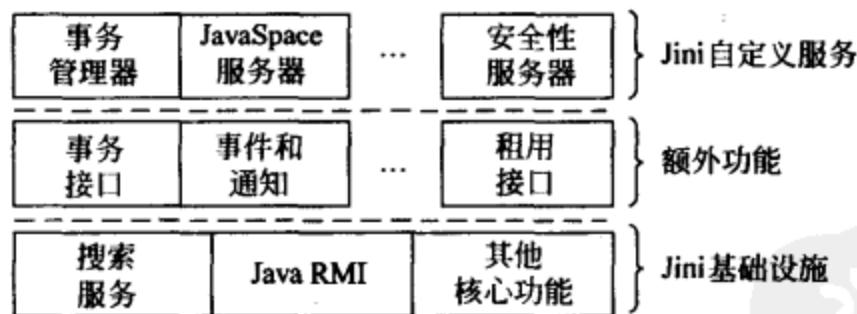


图 12.15 Jini 系统的分层体系结构

第二层由一组通用功能的集合构成,用于扩展基本体系结构的功能并更为有效地实现各种服务。目前这些功能包括一个事件和通知子系统、把租用与资源相关联的工具,以及支持事务的标准接口说明。

最高的一层由客户和服务组成。与其他两层相比不同的是,Jini 并未规定该层应当包含的内容。目前,它提供多种服务,包括一个 JavaSpace 服务器和一个实现 Jini 事务接

口的事务管理器。最高层的程序一般也会直接使用 Jini 基础设施提供的功能。

在后续小节中,我们按照本书前面讨论的分布式系统的 7 条基本原则对 Jini 进行剖析,重点是讨论在 Jini 这个基于协作的并且 JavaSpace 起着重要作用的系统中,这些原则是如何体现的。

12.3.2 通信

正如前面所谈到的,Jini 的核心通信功能都基于 Java RMI。在第 2 章中我们讨论了 RMI,这里不再赘述。除了前面讨论的 JavaSpace 固有的生成通信模型之外,Jini 还提供了一个简单的事件和通知子系统,作为其通信功能的一部分。下面我们就讨论这个子系统。

1. 事件

Jini 的事件模型比较简单。如果一个对象具有一个客户感兴趣的事件,该客户可以在这个对象中进行注册。这样,当发生了这个事件时,对象会通知已注册的客户。或者,客户可以告知对象把通知传送给其他进程。在这两种情况下,都会向对象传送对一个监听器对象(listener object)的远程引用,该引用在事件发生时可以通过 Java RMI 的进行回调。

注册总是从属于租用。当租用到期时,不会再有通知发送给注册的客户(或代表客户接收通知的进程)。使用租用可以防止注册的无限期持续,例如,在注册客户已经崩溃的情况下。下面将讨论租用。

对象通过向注册事件的监听者对象发送一个远程调用来通知事件的发生。事件下一次发生时还会再次调用监听者对象。由于 Jini 自身不保证按照事件发生的顺序交付相应的通知,所以通知通常带有一个序列号,以便监听者对象了解事件的相对顺序。

事件也可以用于 JavaSpace 中。特别是,客户可以请求在把一个特定的元组实例写入 JavaSpace 中时该客户得到这一事件的通知。这种情况下,客户调用由每个 JavaSpace 实现的 notify 操作。该操作可以接受一个模板元组作为输入,用于与被存储的元组实例进行匹配,正如 read 或 take 操作中一样。图 12.16 示意了对事件和 JavaSpace 的结合使

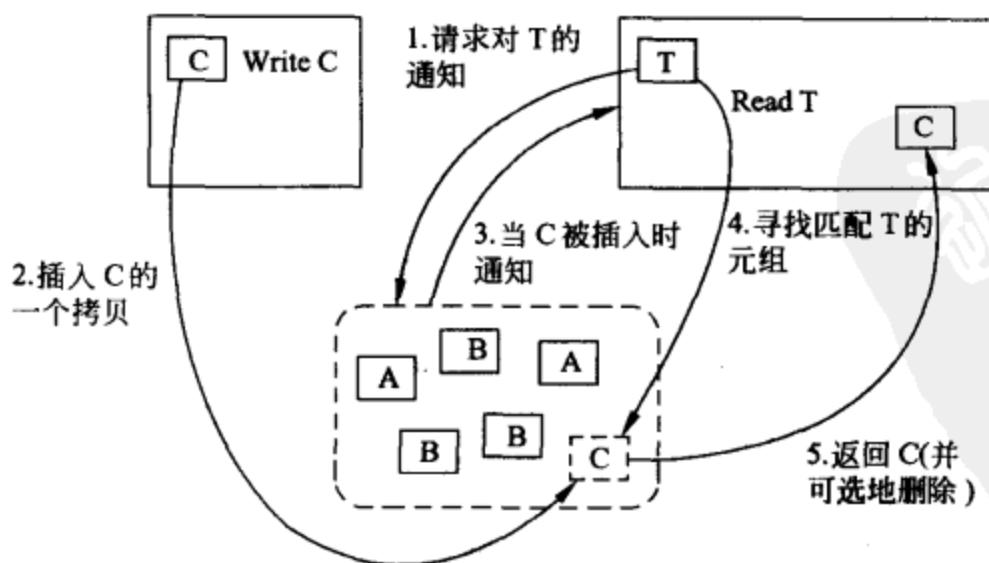


图 12.16 结合使用事件和 JavaSpace

用。注意，客户得到关于一个元组实例的通知并试图读取该实例时，也许另一个进程已经读取该元组实例，并已将其从 JavaSpace 中删除。这样的竞争情况在生成通信中是经常发生的，一般情况下很难避免。

12.3.3 进程

Jini 系统中的进程没有什么特殊的性质。不过，JavaSpace 服务器的实现还是值得加以注意的。我们简要地进行讨论。我们重点讨论 JavaSpace 服务器的分布式实现，即元组实例集合可能分布在多个机器上。Rowstron(2001)介绍了基于元组运行时系统的实现技术最新进展。

一个有效的 JavaSpace 分布式实现必须解决以下两个问题：

- (1) 如何在不进行大量搜索的情况下模拟相关寻址；
- (2) 如何把元组分布到多个机器上，并在稍后定位它们。

要解决这两个问题，关键是要认识到每个元组是一个有类型的数据结构。把元组空间分为多个子空间，每个子空间中的元组具有相同的类型，这样做可以简化程序设计，并使优化措施成为可能。例如，由于元组是有类型的，所以可以在编译时决定 write、read 或 take 操作的对象。这种划分意味着只需要搜索元组实例集合的一部分。

另外，每个子空间可以把其第 i 个元组字段（或该字段的一部分）作为散列键来构造一个散列表。回忆一下，元组实例中的每个字段是对一个对象的编组引用。Jini 没有预先定义应该如何进行编组。因此，实现可以决定在对引用进行编组时，使前面的一些字节用作编组对象的类型标识符。在执行对 write、read 或 take 操作的调用时，可以计算第 i 个字段的散列函数，以找到元组实例在表中的位置。已知子空间和表的位置，就可以免去搜索的过程。当然，在 read 或 take 操作的第 i 个字段为 NULL 时，不可能进行散列处理，因此一般需要对子空间进行完整的搜索。然而，通过谨慎地选择进行散列处理的字段，通常是可以避免搜索的。

还使用了其他的优化措施。例如，上面描述的散列方案把一个给定子空间的元组分布到一些箱子(bin)中，从而把搜索限制在单个箱子中。可以把不同的箱子放置在不同的机器上，这样既可以使负载得到更广泛的分担，也可以更好地利用局部化的优势。如果散列函数是对类型标识符取机器数目的模，那么箱子的数目是随系统大小线形增长的（请参见 Bjomson 1993）。

现在，我们简要地研究不同类型硬件上的各种实现技术。在多处理器系统中，元组子空间可以实现为全局存储器中的散列表，每个子空间对应一个散列表。当执行一个 JavaSpace 操作时，会锁定相应的子空间，然后可以写入或删除元组实例，完成后解除对子空间的锁定。

在多计算机系统中，最佳选择取决于通信的体系结构。如果存在可靠的多播，那么可以选择把所有子空间复制到所有的机器上，如图 12.17 所示。当进行一个 write 操作时，就广播新的元组实例，并把它写入每个机器上的相应子空间中。为了进行一个 read 或 take 操作，对本地子空间进行搜索。然而，由于成功的 take 操作会把元组实例从 JavaSpace 中删除，因此需要使用一个删除协议来把该元组实例从所有机器上删除。要防

止竞争和死锁,可以使用一个两阶段提交协议。

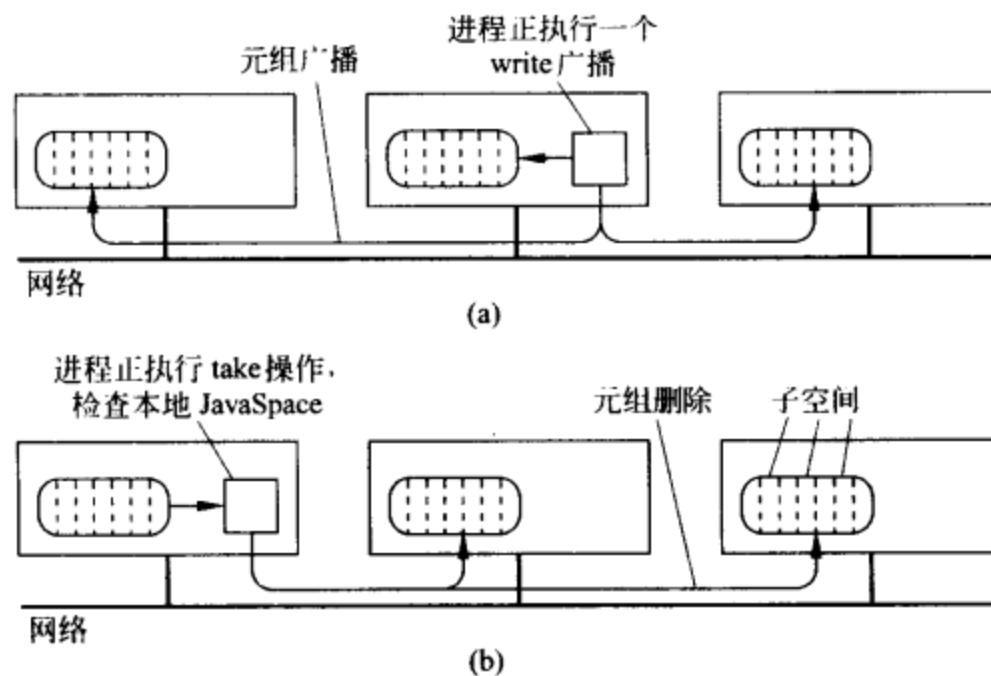


图 12.17 可以把一个 JavaSpace 复制到所有机器上。虚线表明把 JavaSpace 划分为子空间

(a) 在进行 write 时广播元组; (b) read 是本地操作,但调用 take 删除一个元组实例时必须进行广播

这种设计是直截了当的,但当系统在元组实例数目和网络规模方面增长时,它的可扩展性并不是很好。例如,在广域网中实现该方案的代价是极为昂贵的。

相反的设计方案是在本地进行 write 操作,把元组实例仅仅存储在生成它的机器上,如图 12.18 所示。在进行 read 或 take 操作时,进程必须广播模板元组。每个接收者会检查自己是否有与之匹配的元组实例,如果有,将发回一个应答消息。

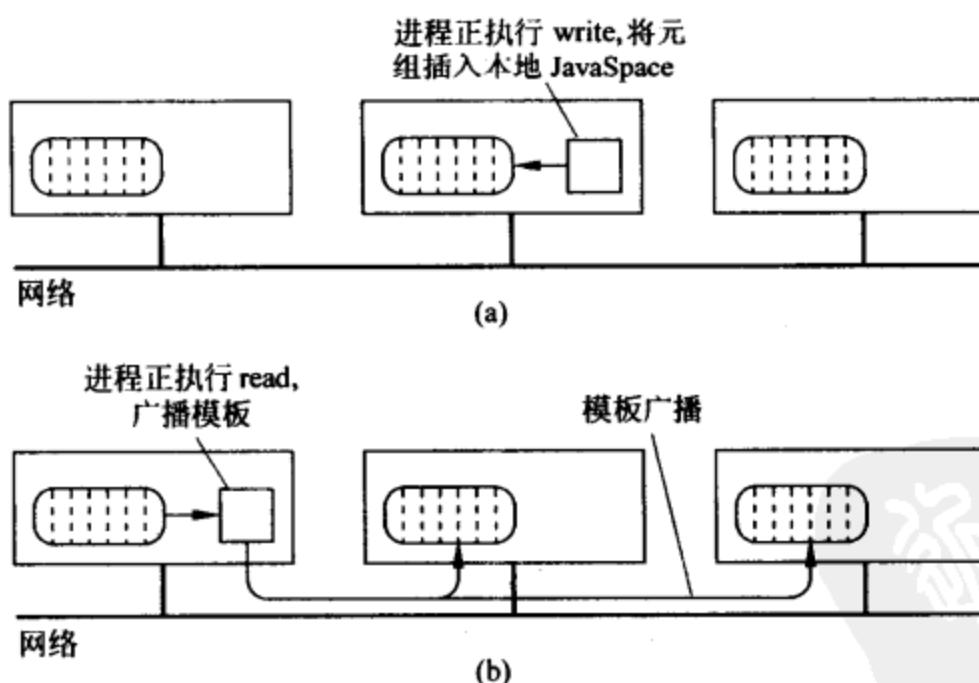


图 12.18 未复制的 JavaSpace

(a) 在本地进行 write 操作; (b) read 或 take 操作需要广播模板元组,以找到元组实例

如果不存在元组实例,或者含有元组实例的机器没有收到广播,请求的机器将无限地重新广播请求,同时逐渐增加两次广播之间的时间间隔,直到出现合适的元组实例,并且请求得到满足。如果发送了两个或多个元组实例,会像本地 write 操作那样进行处理,并

把它们从各自所在的机器上移动到发出请求的机器上。实际上,运行时系统甚至可以自主地移动元组,从而进行负载平衡。Carriero 和 Gelemter(1986)在实现局域网中实现 Linda 元组实例时使用了这种方法。

可以结合使用这两种方法,得到部分复制的系统。作为一个简单的例子,设想所有的机器逻辑上构成了一个矩形网格,如图 12.19 所示。当机器 A 上的一个进程希望进行一个 write 操作时,它把元组广播(或用点到点消息发送)到它所在的行中的所有机器。当机器 B 上的一个进程希望读或取出一个元组实例时,它把元组模板广播到它所在的列中的所有机器。根据几何学原理,总是恰有一台机器同时看到元组实例和元组模板(在本例中是机器 C),该机器进行匹配,并把该元组实例发送给请求它的进程。这种方法类似于我们在第 6 章中讨论的基于法定数量(quorum-based)的复制。这种方法已用于实现 Linda(Ahuja 等 1988),并用于实现簇中的元组空间。

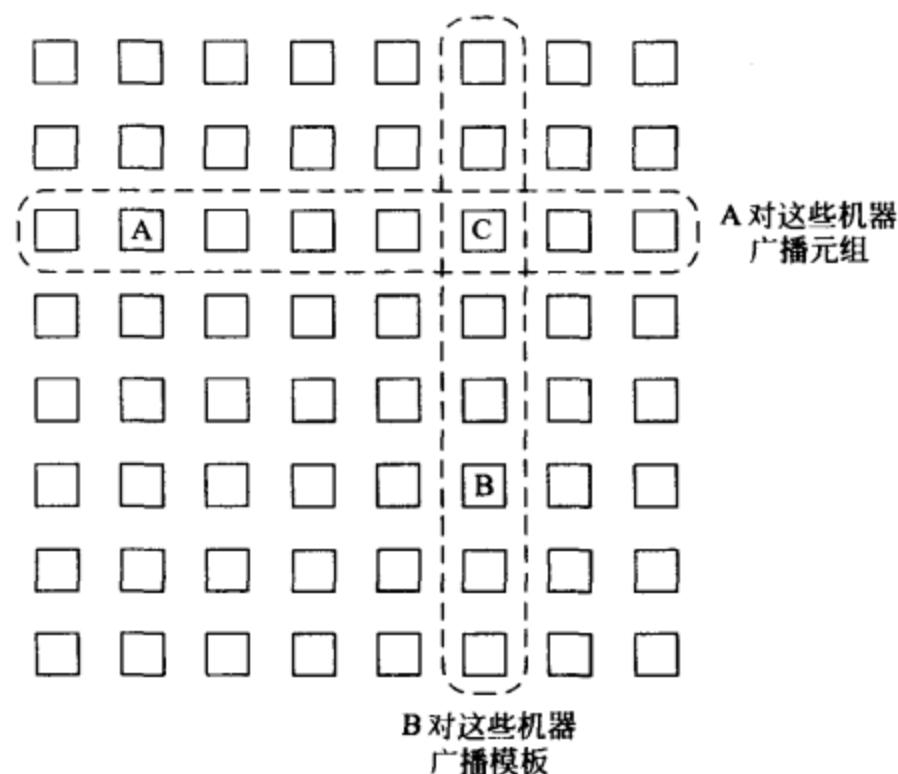


图 12.19 部分广播元组和元组模板

迄今为止讨论的实现具有严重的可扩展性问题,因为在对元组空间插入或删除元组时都会用到多播。元组空间的广域实现尚不存在。最好的情况下,多个不同的元组空间可以并存于一个单一的系统中,每个元组空间在一个单一的服务器或一个局域网中实现。这种方法用于 PageSpaces(Ciancarini 等 1998)和 WCL(Rowstron 和 Wray 1998)中。在 WCL 中,每个元组空间服务器负责一个完整的元组空间。换言之,进程总被定向到一个服务器。不过,可以把一个元组空间迁移到一个不同的服务器上,以改善性能。如何开发元组空间的有效的广域实现仍然是一个未知的问题。

12.3.4 命名

Jini 不提供基于对象的分布式系统或分布式文件系统中提供的传统命名服务。在 Jini 体系结构中,传统的命名服务作为服务的一部分可以很容易地实现,但是这不构成

Jini 的核心。Jini 提供一种服务,该服务允许客户使用一种基于属性的搜索功能查询已注册的服务。下面说明这种查询服务。

1. Jini 查询服务

正如我们所提到的,Jini 的设计目标之一是在该系统中,客户可以很容易地查询到刚刚可用的新服务。原理上,JavaSpace 可以实现这一服务。当一个服务添加到现有的系统中时,该服务在 JavaSpace 中插入一个对自己进行描述的元组实例。正在寻找特定服务的客户可以请求 JavaSpace 在某个服务插入与请求相匹配的元组时向它发出通知。

Jini 不是使用 JavaSpace,而是提供了一个单独的、专门的查询服务作为其最低级体系结构层的一部分,如图 12.15 所示。服务通过提供一个(属性,值)对的集合来注册自己,例如,服务内容、联系地址等。客户可以通过向查询服务提供一个模板来查找一个服务,该模板类似于 JavaSpace 中的模板元组。查询服务返回与之相匹配的服务的信息。我们简要地讨论一下查询服务。

每个服务具有一个相关的服务标识符(service identifier),它是由查询服务生成的一个全局惟一的 128 位值。服务使用该标识符在查询服务中注册一个服务项(service item)。可以把服务项看作具有三个字段的记录,如图 12.20 所示。

字段	说明
ServiceID	与该项相关的服务的标识符
Service	对实现该服务的对象的(可能是远程的)引用
AttributeSets	描述服务的元组集合

图 12.20 一个服务项的结构

ServiceID 字段包含查询服务分配给该服务的标识符。标识符在查询服务中用作惟一的键。因此,在查询服务中不会有两个服务项具有同一个标识符。

Service 字段包含对一个对象的引用。许多情况下,它是对一个远程对象的引用,这意味着从查询服务中获得该引用的客户可以直接使用 Java 的 RMI 引用该对象。回忆一下,在第 2 章中,Java 中的远程引用通常作为一个编组的代理来实现,只需要解编就可以引用该对象。

AttributeSets 字段是元组的集合,其元组类似于 JavaSpace 中使用的元组。每个元组实质上相当于一个 Java 对象,元组中的每个字段描述该对象的一个(属性,值)对。使用与 JavaSpace 相同的技术,客户在寻找特定的元组实例时可以提供一个模板元组。查询服务将只选择那些与模板相匹配的元组。

与 JavaSpace 中的情况相同,客户可以请求查询服务在插入了某个与客户的模板元组相匹配的服务项时向它发送一个通知。

为了帮助建立 Jini 系统,有一些用于注册服务的预定义元组,如图 12.21 所示。对这些元组,所有属性都用字符串表示,但其他表示也是允许的,从而提供所需的灵活性。

元组类型	属性
ServiceInfo	名称、厂商、提供商、版本、模型、序列号
Location	楼层、房间、建筑物
Address	街道、公司、组织单元、位置、州或省、邮编、国家

图 12.21 服务项预定义元组的例子

我们隐含地假设了只有一个查询服务。然而, Jini 允许多个查询服务同时存在。每个查询服务可以负责一组服务。这样,单个查询服务的负载可以分布到多个不同的机器上去。

迄今为止,还有一个重要的问题没有讨论,即查询服务如何进行查询。许多分布式系统采用的标准方法配置一个具有众所周知的地址的查询服务器。Jini 采用的是另一种方法,由客户多播一个消息,请求查询服务告知其位置。如果不采取特殊的措施,这种方法只能在局域网中有效地工作。

另外,查询服务也定期地用多播宣告它们的存在。客户可以记下一个查询服务的位置,以便下次需要搜索时使用。相关协议的详细内容在(Waldo 2000)中描述,其中还包含了对查询服务接口的具体规范。

2. 租用

与命名问题相关的是对对象的引用的管理。我们在第 4 章中已经解释,管理引用的一种方法是让被引用的对象记录下谁正在引用它,从而产生一个引用列表。为了保持该列表简短,同时仍可以应付引用进程崩溃这样的情况,使用租用是比较方便的做法。当租用到期时,引用变为无效,从对象的引用列表中删除。当列表变为空时,对象可以安全地销毁它自己。

Jini 广泛使用了租用来保证对象在不再被引用时得到清除。例如,当一个进程向 JavaSpace 中写入一个元组时,它会得到一个返回的租用,指明该元组在被销毁之前可以保存的时间。这种情况下,write 操作允许其调用者指定所需的租期。

租用的分发并没有得到绝对的保证。当一个进程发出一个租用时,它允诺尽力保持与该租用相关的对象起码在指定的时间内不被销毁。持有一个租用的进程总是可以请求进行续租,但是是否答应续租是由承租方决定的。

租用的接口已经得到了标准化。当在 Jini 中使用租用时,它遵循同一个规范。这些接口的详细说明请参见文献(Waldo 2000)。

12.3.5 同步

Jini 提供了一些同步机制。一类重要的机制作为 JavaSpace 的一部分实现,即阻塞操作 read 和 take。这些操作可以用于表达许多不同的同步模式,这些同步模式在(Carriero 和 Gelemter 1989)中讨论。除了通过这两种操作提供的同步功能外,Jini 还提供事务的概念,我们接下来对此进行讨论。

1. 事务

为了帮助实施在多个对象上的一系列操作,Jini 支持使用两阶段提交协议的事务。这一支持实质上以一个接口集合的形式给出,没有提供这些接口的实际实现。但是,Jini 可以用一个默认的事务管理器进行配置。

这种方法有一个重要的含义,即 Jini 自身不提供事务的 ACID 属性。Jini 假设参与事务的各个进程联合实现这些属性。然而,进程之间的交互操作遵守使用两阶段提交的事务中的模式。

Jini 中事务的整体模型如图 12.22 所示。客户可以通过向事务管理器发起一个请求来启动一个事务,事务管理器返回一个事务标识符。在许多情况下,还要求客户指定事务在中止或提交之前所花费的时间。此外,管理器将为一个新创建的事务传递一个租用,只要租用到期,便中止该事务。

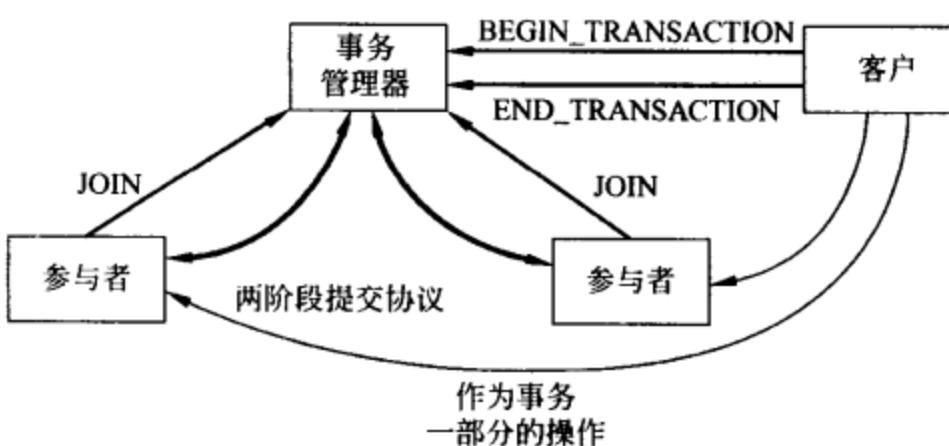


图 12.22 Jini 中事务的一般结构。粗线标明 Jini 的事务协议要求的通信

客户可以指示其他进程加入事务。这些进程必须实现一个预定义的接口,以便事务管理器进一步控制事务。该接口包含 commit 和 abort 之类的操作,事务管理器通过调用这些操作控制事务的参与者。参与者负责正确地实现这些方法。

假设事务可以在其相关租用到期之前结束,客户将告知事务管理器它应当提交或中止该事务。此时,事务管理器执行两阶段提交协议,并把结果返回给客户。实际的协议在第 7 章中已经讨论过了,这里不再赘述。

Jini 还支持嵌套事务。在嵌套事务的情况下,Jini 请求事务管理器启动一个事务,作为一个已有事务的子事务。同样,客户控制事务的组织,即它决定哪些进程应该加入一个事务。在嵌套事务中,会请求进程加入某个特定的子事务。

JavaSpace 也可以参与事务。反之,事务可以跨越多个 JavaSpace。当一个进程发起一个 JavaSpace 操作时,它可以随之传递一个事务标识符。如果 JavaSpace 尚未加入该事务,它首先加入该事务。JavaSpace 服务器和事务管理器共同确保事务的 ACID 属性得到遵守。另外,JavaSpace 的 Jini 实现和作为 Jini 的一部分提供的默认事务管理器一起执行严格的两阶段锁定,以避免级联式中止。两阶段锁定已在第 5 章中解释。

12.3.6 缓存和复制

与 TIB/Rendezvous 类似, Jini 没有提供特殊的缓存或复制措施, 而是把这些问题留给作为基于 Jini 的系统的一部分构建的应用程序。只有在查询服务中, Jini 假设应对服务进行复制以便进行容错。

12.3.7 容错性

除了上面说明的事务管理器实现了事务协议之外, Jini 自身不提供对容错的更多支持。Jini 希望使用它的组件实现它们自己所需的容错措施。

人们对于在最初的 Linda 元组空间(构成 JavaSpace 的基础)中加入容错机制做了大量的研究工作。例如, Bakken 和 Schlichting(1995)描述了一种基于元组空间的主动复制的方法。此外, 他们把程序设计模型扩展到把多个操作组合为一个单一的原子性单元。把元组空间操作组合到事务中的一种更适合 Jini 的方法在(Shasha 和 Jeong 1994)中进行了描述。

关于通信, 请注意, 几乎所有的通信都是通过 Java RMI 完成的。Java RMI 本身一般通过一种可靠的、低级通信协议(如 HTTP 或 TCP)实现。

12.3.8 安全性

Jini 中的安全性完全依赖 Java RMI 提供的安全性。Java RMI 的大多数问题已经在第 8 章中讨论, 特别是通过栈自省防护动态下载的代码的有关问题。栈自省(stack introspection)的一个重要属性是把访问特权附加给类的可能性。可以在运行时使用 Java 安全管理器检查这些特权。

Jini 中新增的功能是一个单独的服务, 称为 JAAS (Java authentication and authorization service, Java 身份验证和授权服务), 该服务处理基于 Java 系统(例如 Jini)中的用户身份验证和授权问题。与其他分布式系统采用的方法类似, JAAS 把提供给用户进行身份验证和访问控制的接口和这些服务的实际实现分离开来。这种分离通过 PAM 可插入身份验证模块(pluggable authentication module, PAM)实现(Samar 和 Lai 1996)。PAM 实质上在应用程序和安全服务之间提供了一个中间层, 向上下两层提供一个标准接口, 如图 12.23 所示。JAAS 是 PAM 的一种 Java 实现。

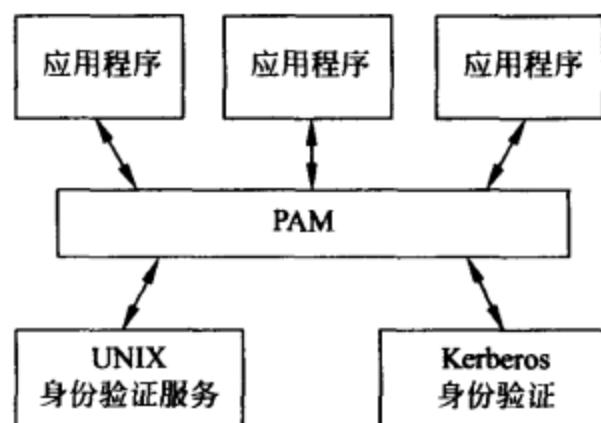


图 12.23 PAM 关于安全服务的位置

JAAS 对 Java 已有访问控制机制的增强在于,能够执行与以前已验证用户有关的访问控制。第 8 章中讨论了 Java,它提供基于类的访问控制功能,可以把特权与类相关联。JAAS 也能处理用户。实际上,JAAS 提供了必要的支持,使基于 Jini 的系统能够支持多个用户。

鉴于 JAAS 类似于我们已经讨论的身份验证和授权服务,这里不再过多地探讨。在 (Lai 等 1999) 中有详细的信息。

12.4 TIB/Rendezvous 和 Jini 的比较

本章描述的两个基于协作的系统是其他基于协作的分布式系统的代表。下面我们将对 TIB/Rendezvous 和 Jini 进行简要的比较,看看它们之间的主要差异是什么。

1. 基本观点

TIB/Rendezvous 和 Jini 都致力于提供进程的引用性解耦合,也就是说,使相互通信的进程可以或多或少地保持匿名。为了实现引用性解耦合,TIB/Rendezvous 采用了一个发布/订阅机制,而 Jini 则使用了 JavaSpace 的生成通信。此外,Jini 还提供现时性解耦合。

这两个系统间的另一个差异是,TIB/Rendezvous 处理进程间的大部分通信。但在 Jini 中,基本思想是系统应该提供把进程结合到一起的基础,一旦进程彼此可找到,通信就可以使用不同的方法进行,如使用 Java RMI。在这个意义上,Jini 更注重于动态地集成进程,而 TIB/Rendezvous 则更多地致力于维护进程间的松散耦合。

2. 通信

正如我们已提到的,TIB/Rendezvous 中的通信主要由作为其基础的发布/订阅机制完成。因此,多播在 TIB/Rendezvous 中起着重要的作用,并且采取了许多措施来保证多播也能够有效地工作在广域网中。为了使通信应用程序独立,消息应该是自描述型的。

相比之下,Jini 实质上只把多播用于对帮助进一步定位其他进程的查询服务的初始定位上。所有其他通信,包括与 JavaSpace 服务器的通信主要基于 Java RMI。换言之,一旦一个进程发现了它希望与之通信的其他进程,它会使用点到点通信功能。而且,消息的格式和内容完全由通信双方决定。

在这两种系统中,事件都起着重要的作用,但是方式有所不同。在 TIB/Rendezvous 中,一个重要的设计问题是通信是通过事件机制处理的。原则上,一个输入的消息仅当接收者显式安装了该消息的处理器时,才可以由接收者接收。对于应用程序,没有处理输入消息的阻塞接收操作。

Jini 中的事件具有不同的特性。每个进程可以提供一个事件服务,允许其他进程注册以便进行通知。当一个特定的事件发生时,将回调注册的进程,从而进行相应的处理。因此,Jini 的事件机制实质上是一个在进程对之间操作的回调服务。

3. 进程

由于 TIB/Rendezvous 和 Jini 实质上只提供进程通信的手段,所以关于得到支持的进程没有什么特殊的规定。此外,尽管两个系统都提供了一些实现事务之类服务的特定进程,但是这些进程与用户开发的应用程序进程没有什么区别。

4. 命名

在命名问题上,TIB/Rendezvous 和 Jini 之间有着重要的区别。TIB/Rendezvous 中的名称在基于主题的寻址中扮演重要的角色。所有的地址用语法上类似 DNS 的字符串名称表达。TIB/Rendezvous 提供简单而又有效功能,以允许用通配符和缩写符来代表一组主题。

与 TIB/Rendezvous 不同,Jini 中的名称采用字节串的形式,一个串代表一个编组的对象。只能把串用于比较是否相等,但也可以以 NULL 串的形式使用通配符。

由于采用了发布/订阅范型,TIB/Rendezvous 无须提供单独的命名服务来把名称解析为进程的地址。相反,Jini 提供了一个单独的查询服务,以便客户能够在基于属性的名称(仍然用字节串系列表达)的基础上定位一个进程。

5. 同步

TIB/Rendezvous 和 Jini 都提供事务机制,但二者的事务模型不同。在 TIB/Rendezvous 中,事务的关键思想是把一系列发布和接收操作(可能还有数据库操作)组成一个单一的事务。事务(实际上是事务性的消息传递)由一个单独的层支持,其中包含了发布/订阅库的扩展。此外,还提供事务管理器。TIB/Rendezvous 确保事务中的消息传递机制的 ACID 属性得到实现。

Jini 实质上只提供了一个事务协议,允许把一个客户引发的多个操作组合为一个事务。但是,事务的 ACID 属性是由参与事务的进程来确保实现的。Jini 提供了一个单独的事务管理器,还与 JavaSpace 服务器一起确保这些 ACID 属性的实现。

TIB/Rendezvous 中的事务操作局限在单个进程中。当多个进程需要参与一个事务时,需要一个额外的事务管理器来处理消息传递之外的问题。在 Jini 中,多个客户进程可以参与同一个事务,不过通常只能由一个客户发起事务的启动和结束。

除了事务以外,Jini 还通过 JavaSpace 中的阻塞性操作提供同步机制,这些操作有效地实现了一种分布式的锁定。TIB/Rendezvous 没有提供其他的同步功能。

6. 缓存和复制

TIB/Rendezvous 和 Jini 都没有对缓存和复制提供支持。这些功能留给应用程序实现。

7. 容错性

在容错方面,两个系统都提供可靠的通信。TIB/Rendezvous 不仅以确认消息交付的

形式,而且通过事务消息传递的手段支持持久性的通信。Jini 的可靠通信完全基于 Java RMI 提供的可靠性。

TIB/Rendezvous 通过显式地支持进程组来支持容错,它在中间件中构造了一个接管机制,在进程失败时发挥作用。Jini 没有针对进程组的专门功能,它假设这些功能由应用程序来实现。两个系统都没有明显的故障恢复功能,只是在实现事务时提供了这一功能。

8. 安全性

最后,TIB/Rendezvous 用一个单独的协议的形式支持安全性,该协议在发布者和订阅者之间建立一个安全通道。没有提供访问控制,而是把它留给使用 TIB/Rendezvous 作为通信中间件的应用程序来实现。

Jini 的安全性完全依赖于基于 Java 系统通常具有的安全性能。这些性能包括我们在第 8 章中讨论的基于类的访问控制,还包括由一个称为 JSSA 的单独服务提供的用户身份验证和授权功能。

为了对这些比较做一总结,图 12.24 中列出了我们在本节中讨论的一些最重要的问题,以及在每个系统中它们是如何解决的。

问题	TIB/Rendezvous	Jini
主要设计目标	进程的解耦合	灵活的集成
协作模型	发布/订阅	生成通信
网络通信	多播	Java RMI
消息	自描述型	特定于进程
事件机制	对于输入消息	作为一种回调服务
进程	通用	通用
名称	字符串	字节串
命名服务	无	查询服务
事务(操作)	消息	方法引用
事务(范围)	单个进程(参见正文)	多进程
锁定	否	作为 JavaSpace 操作
缓存和复写	否	否
可靠的通信	是	是
进程组	是	否
恢复机制	无明显支持	无明显支持
安全性	安全通道	完全基于 Java

图 12.24 TIB/Rendezvous 和 Jini 的比较

12.5 小结

基于协作的分布式系统正逐步在构建分布式应用程序时发挥重要的作用。大多数这种系统的重点在于进程的引用性解耦合,即进程在通信时无需显式地彼此引用。此外,还可能提供现时性解耦合,其中的进程可以不必为进行通信而同时存在。

一组重要的基于协作的系统由那些像 TIB/Rendezvous 那样遵循发布/订阅范型的系统组成。在这种模型中,消息中不携带接收者的地址,而是通过主题进行寻址。希望接收消息的进程应订阅某个特定的主题,中间件负责把消息从发布者传递给订阅者。

另一组基于协作的分布式系统使用生成通信,该模型在 Linda 系统中首次使用。生成通信通过一个元组的共享空间实现。一个元组与一个记录类似,是一个有类型的数据结构。为了从一个元组空间中读取一个元组,进程通过提供一个模板元组指定要寻找的元组。匹配该模板的元组会被挑选出来,并返回给请求的进程。如果找不到匹配的元组,进程会阻塞。

基于协作的系统与其他分布式系统的不同在于它们完全致力于为进程提供一种无须事先得知对方情况的便利的通信方法。通信还能够以匿名的方式继续下去。这种方法的主要优点是其灵活性,因为在系统继续运行的同时,对其进行扩展或修改变得更容易了。

本书前半部分讨论的分布式系统的原理同样可以很好地应用于基于协作的系统,尽管缓存和复制在目前的实现中所起的作用比较微弱。此外,命名与目录服务支持的基于属性的搜索有密切的联系。

习题

- 可以把第 2 章中讨论的消息队列系统归为哪种类型的协作模型?
- TIB/Rendezvous 中的引用性解耦合由基于主题的寻址支持。在该系统中引用性解耦合还得到了哪些支持?
- 略述基于消息队列系统(例如 IBM MQSeries)的发布/订阅系统的一种实现。
- TIB/Rendezvous 中的主题名称实际上被解析为什么?名称解析过程如何进行?
- 略述 TIB/Rendezvous 系统中全序消息交付的一个简单实现。
- 当作为一个 TIB/Rendezvous 的一部分把一个消息交付给进程 P 之后,P 确认该交付。P 的事务层是否可能自动向事务守护进程发送一个确认消息?
- 假设在 TIB/Rendezvous 系统中复制了一个进程。给出两种方法,避免来自复制进程的消息被不止一次地发布。
- 当在 TIB/Rendezvous 系统中复制进程时,我们在多大程度上需要全序多播?
- 描述一种简单的 PGM 方案,允许接收者检测到有消息丢失(即使丢失的是序列中的最后一个消息)。
- TIB/Rendezvous 中作为文件实现的账簿能够保证消息绝不丢失(甚至是在进程崩溃时)吗?

11. 如何在 TIB/Rendezvous 中实现一个基于生成通信的协作模型?
12. 除了 Jini 中的现时性解耦合外,在考虑到协作模型时,你认为什么是 Jini 和 TIB/Rendezvous 之间最重要的区别?
13. Jini 中的租用期限总是指定为一段持续时间,而不是一个租用到期的绝对时间。为什么?
14. Jini 中最重要的可扩展性问题是什么?
15. 考虑一个 JavaSpace 的分布式实现,其中的元组复制到多个机器上。给出一个删除元组的协议,以避免两个进程试图删除同一元组时的竞争情况发生。
16. 假定 Jini 中的一个事务 T 请求锁定一个当前正由另一个事务 T' 锁定的对象。请解释会发生什么。
17. 假定一个 Jini 客户缓存从一个 JavaSpace 获取的元组,以便下次不用再访问 JavaSpace。这种缓存是否有用?
18. 回答上一个问题,但这次是客户缓存查询服务返回的结果。
19. 略述容错 JavaSpace 的一种简单实现。

第 13 章 阅读材料和参考书目

前面的 12 章中已经介绍了很多主题。本章的目标是帮助那些感兴趣的读者进一步学习分布式系统。13.1 节是建议的阅读材料的列表,13.2 节是按字母顺序排列的本书引用的书籍和文章的列表。

除了下面给出的参考文献之外,“Proceedings of the n-th ACM Symposium on Operating Systems Principles”(SOSP, 每两年举办一次)和“Proceedings of the n-th International Conf. on Distributed Computing Systems”(ICDCS, 每年举办一次)收集了分布式系统方面的最新论文。另一个寻找参考资料的好地方是“Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms”(有关中间件的会议)。该会议每 18 个月举办一次。另外,《ACM Transactions on Computer Systems》是一本著名的期刊,常常发表一些关于分布式系统的优秀文章。

13.1 对进一步阅读的建议

13.1.1 介绍性和综述性的著作

Coulouris 等的《Distributed Systems—Concepts and Design》

一本全面介绍分布式系统的好书。该书介绍了与本书类似的内容,但组织内容的方式截然不同。包含很多关于分布式事务的内容,但在命名和安全方面则描述得比较少。案例学习包括 CORBA 和 Mach 分布式操作系统。

Chow 和 Johnson 的《Distributed Operating Systems and Algorithms》

该书的第一部分从操作系统的角度审视分布式系统,包括诸如分布式共享内存和分布式进程调度等主题。第二部分系统地讲解诸如同步、分布式协议、复制和容错性等主题。

Geihs 的“Middleware Challenges Ahead”

该文章简要地介绍分布式系统的中间件,重点放在未来中间件系统的需求上。文章还讨论编程模型、自定义能力,并阐述移动计算的相关内容。

Mullender 的《Distributed Systems》第二版

一本暑期科研报告集,其中的 21 篇论文是分布式系统权威专家的著作。主题涉及建

模、规范、容错、实时性、通信、命名、文件系统、调度、安全性以及其他内容。虽然该书没有讨论最近的发展,但仍然涵盖了与所有分布式系统相关的方方面面。

Neuman 的“Scale in Distributed Systems”

该论文是系统地阐述分布式系统扩展性问题的为数不多的论文之一。主要作为扩展时使用的技术来探讨缓存、复制以及分布技术,并提供将这些技术应用于大型系统设计时的经验法则。

Silberschatz 等的《Applied Operating System Concepts》

这是一本全面的关于操作系统的教科书,其中包括有关分布式系统的内容,重点是文件系统和分布式协作。

Umar 的《Object-Oriented Client/Server Internet Environments》

这是一本比较通俗易懂的书,主要讨论客户/服务器系统,涵盖诸如事务处理和远程数据库访问等传统主题。本书还精彩地讲述了从业人员是如何看待分布式系统的。

Verissimo 和 Rodrigues 的《Distributed Systems for Systems Architects》

这是一本有关分布式系统的高级读物,介绍的内容与本书基本相同。相对而言,该书更侧重于容错、实时分布式系统以及分布式系统的管理。

13.1.2 通信

Birrell 和 Nelson 的“Implementing Remote Procedure Calls”

这是一篇关于设计和实现最早的远程过程调用系统之一的经典论文。

Crowcroft 等的《Internetworking Multimedia》

整本书的内容都是关于互联网上的多媒体通信的。介绍了多媒体通信的传输协议、会话组织、会议控制、按需媒体以及多媒体通信中的安全性。

Halsall 的《Multimedia Communications—Applications, Networks, Protocols, and Standards》

该书对许多多媒体联网协议进行了深入的分析,同时还讨论了许多传统的联网内容。多媒体主题包括多媒体网络、信息展示和压缩技术。

Kurose 和 Ross 的《Computer Networking, A Top-Down Approach Featuring the Internet》

这是一本内容全面的好教材,提供了最新的计算机网络方面的信息。为理解分布式系统中的通信提供了详实的背景知识。

Oram 的《Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology》

分布式系统的下一代的形式可能会是由终端用户通过互联网结成松散的、特别的联盟。本书汇集了大量有关这些点对点网络的论文。涉及内容包括很多工程,以及诸如安全性、信任和可统计性等重要主题。

Waldo 的“Remote Procedure Calls and Java Remote Method Invocation”

该文章浅显易懂,清晰地介绍了 RPC 与 RMI 之间的相同点与不同点,作者是基于 Java 的分布式技术(如 Java RMI 和 Jini)的主要设计者之一。

13.1.3 进程

Andrews 的《Foundations of Multithreaded, Parallel, and Distributed Programming》

如果您需要一本透彻地介绍编写并行系统和分布式系统的指导书籍,那么该书正是您所需要的。

Lewis 和 Berg 的《Multithreaded Programming with Pthreads》

Pthreads 构成了实现操作系统线程的 POSIX 标准,并受到 UNIX 系统的广泛支持。尽管作者集中介绍了 Pthreads,但书中还是提供了线程编程的精彩指南。同样,本书还包含了开发多线程客户和服务器所需的基础知识。

Milojicic 等的“Process Migration”

本文精彩而全面地阐述了进程移植方面的内容,包括移动代码和移动代理。进程移植受到了分布式(操作)系统研究人员的广泛注意,但还没有流行。这篇文章可作为本书第 3 章中讨论的代码移植的补充内容。

Nwana 和 Ndumu 的“A Perspective on Software Agents Research”

该书由两位此领域的权威人士编写,综述了研究人员应该对基于代理的系统研究些什么,以及他们实际上应该达到什么目的。在讨论实际的代理状况时,两位作者并不乐观,主张应该开发更多具备工业强度的系统来揭示实际的问题。

Schmidt 等的《Pattern-Oriented Software Architecture—Patterns for Concurrent and Networked Objects》

只是在最近,研究人员才开始注意分布式系统中的通用设计模式。这些模式可简化分布式系统的开发过程,原因是能够使程序员将精力更集中于系统特定的问题上。该书讨论了设备访问、事件处理、同步和并发的设计模式。

13.1.4 命名

Albitz 和 Liu 的《DNS and BIND》

BIND 是广泛使用的一种 DNS 服务器实现,它是容易得到的公共软件。该书详尽地阐述了使用 BIND 建立 DNS 域的方法。同时还提供了当今用得最多的分布式命名服务的大量实际信息。

Hudson 等的《Garbage Collecting the World: One Car at a Time》

大型分布式垃圾收集中的一个最困难的问题是跟踪。这篇文章讲述了另一种垃圾收集的方法,该方法部分是依赖于本书第 3 章所讨论的 Lang 等的解决方法。

Loshin 的《Big Book of Lightweight Directory Access Protocol(LDAP)RFCs》

基于 LDAP 的系统在分布式系统中得到了迅速的普及。LDAP 服务的最终起源是由 IETF 发布的 RFC。Loshin 将相关内容汇集起来,编写出这本全面的关于 LDAP 服务的设计和实现的参考。

Needham 的“Names”

这是一篇关于分布式系统中名称角色的浅显易懂的精彩文章。其重点在于命名系统(请参阅本书 4.1 节),使用 DEC 的 GNS 作为示例。

Pitoura 和 Samaras 的“Locating Objects in Mobile Computing”

这篇文章可作为关于定位服务的全面指导。作者讨论了各种不同的定位服务,包括电信系统中使用的定位服务。本文包含了参考资料的广泛列表,可作为进一步阅读其他文献的起始点。

Saltzer 的“Naming and Binding Objects”

虽然该文章写于 1978 年,并且关注的也不是分布式系统,但应该将该文作为命名研究的起始点。文中论述了名称与对象之间的关系,尤其是将名称解析为所引用的对象。另外也着重介绍了封闭机制的概念。

13.1.5 同步

Anceaume 和 Puaut 的“Performance Evaluation of Clock Synchronization Algorithms”

这篇报告包含了几类著名的时钟同步算法。另外,还有一篇关于这几种算法的基于仿真的性能分析报告,其中考虑了容错性问题。同时该文还对各种算法及其属性进行了精彩的论述。

Babaoglu 和 Marzullo 的“Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms”

尽管对分布式中的全局化状态概念的论述理论性较强,但仍不失为一篇优秀的文章。其中包含的材料涉及逻辑时钟和矢量时钟、分布式快照以及全局谓词。

Gray 和 Reuter 的《Transaction Processing: Concepts and Techniques》

这是一本很全面的关于分布式和非分布式系统的著作。该书讲述了事务处理的实际方法,并详尽地阐述了基于事务的系统的多种选项的设计和实现。如果您正在寻找一本关于事务处理的书,那么此书正是您所需要的。

Lynch 的《Distributed Algorithms》

此书使用单一框架描述了多种不同的分布式算法。其中考虑了三种不同的定时模型:单一同步模型、无任何定时假设的异步模型以及部分接近实时系统的同步模型。如果您以前学习的是理论性的概念,那么您将发现该书包含了大量实用的算法。

Raynal 和 Singhal 的“Logical Time: Capturing Causality in Distributed Systems”

这篇文章用相对简单的术语描述了三种逻辑时钟:标量时间(例如 Lamport 时间戳)、矢量时间以及矩阵时间。另外,此文章还介绍了大量实际的和实验性的分布式系统中所使用的多种实现。

Wu 的《Distributed System Design》

对于从事实际工作的人员来说,由于该书的题目带有理论性,因此可能不太合适。然而,该书包含了多种分布式算法,包括路由选择和负载分布的算法,因此可作为本书第 5 章的补充参考材料。

13.1.6 一致性与复制

Ahamad 和 Kordale 的“Scalable Consistency Protocols for Distributed Services”

此文章讲述了一种一致性协议,在其中,更新不需要立即传播给所有的拷贝。而是允许拷贝反映出上一次的一致性全局状态,然后在状态调节到更近的全局状态后,再进行更新。允许当前全局状态和本地一致性状态之间的滞后是这种方法的可扩展性的关键所在。

Gray 等的“The Dangers of Replication and a Solution”

此文章讨论的是实现顺序一致性模型的复制(称为期望复制)与懒惰复制之间的平衡。这两种复制形式都对事务处理进行了公式化。期望复制的问题是可扩展性差,而懒惰复制则可能容易导致更难以解决或可能有冲突的解决方案。作者的目的正是促成一种

混合型的模式。

Protic 等的“Distributed Shared Memory: Concepts and Systems”

内存一致性模型在分布式共享内存(DSM)系统中扮演着重要的角色,该领域的大多数研究都源自开发中的 DSM 系统。这篇文章给出了这样一种系统的概述,并总结了各种一致性模型。

Saito 的“Consistency Management in Optimistic Replication Algorithms”

此报告展示了弱一致性模型中使用的优化复制算法的分类。报告还说明了另一种审视复制及其关联的一致性协议的方法。其中的一个有趣的话题是对各种解决方案可扩展性的讨论。该报告还包含了一个参考资料的广泛列表。

Yu 和 Vahdat 的“Design and Evaluation of a Continuous Consistency Model for Replicated Services”

此文讨论的是可支持失控于应用程序的一致性模型的系统。区分模型的需求源自于性能、可用性和一致性之间的平衡。一致性是根据三个连续的参数定义的:一个副本允许的不可见的写入次数、在进行更新前可本地执行的写入次数以及允许的更新延迟时间。

Wiesmann 等的“Understanding Replication in Databases and Distributed Systems”

一般来说,分布式数据库和常规的分布式系统中处理复制的方法有一些不同。在数据库中,复制的主要原因常常是为了提高性能。而在常规分布式系统中,复制常常是为了提高容错性。此文展示了一个能够更容易地比较解决方案的框架。

13.1.7 容错性

Cristian 和 Fetzer 的“The Timed Asynchronous Distributed System Model”

此文讨论了一种更现实的分布式系统模型,而不是单纯的同步或异步模型。两个重要的假设是:服务完全在一个指定的时间间隔内发生;通信是不可靠的,并会造成性能故障。此文阐述了这种模型在捕获实际分布式系统的重要属性时的适用性。

Guerraoui 和 Schiper 的“Software-Based Replication for Fault Tolerance”

此文简要清晰地描述了分布式系统中的复制是如何增强容错性的。此文还讨论了主机—备份复制和主动复制,以及与组通信相关的复制。

Guerraoui 和 Schiper 的“A Generic Consensus Service”

在分布式系统中实现容错性的困难之一是有太多的解决方案和协议,并且在某种情况下的适用性是很难评估的。此文描述了一种框架,用模块化的方式包含了多种容错性协议,并用分离的服务实现。

Jalote 的《Fault Tolerance in Distributed Systems》

此书是少数几本完整地阐述分布式系统中的容错性的教科书之一,内容涵盖了可靠的广播、恢复、复制和进程恢复。其中有单独的一章来解释软件设计的错误。

Obaczka 的“Multicast Transport Protocols: A Survey and Taxonomy”

目前存在着大量的旨在提供可靠性的多播协议,即使是在广域系统中。可扩展性和可靠性的组合在实际工作中是难以实现的。此文综述了在可扩展的可靠稳定多播方面进行的某些尝试。

13.1.8 安全性

Anderson 的《Security Engineering: A Guide to Building Dependable Distributed Systems》

此书是少数几本旨在阐述整个安全领域的成功教科书之一,讨论了诸如口令、访问控制和密码编译等基础知识。安全性与应用程序域紧密相关,文中讨论了下面几个领域中的安全性:军事领域、银行业、医药系统及其他系统。最后还讨论了社会、组织和政治方面的安全性。该书是进一步研究安全性的最佳起始点。

Blaze 等的“The Role of Trust Management in Distributed Systems Security”

此文阐述的主题是大型分布式系统应该使用一种比当前所用的更简单的方法来授权对资源的访问。尤其是如果请求的证书集符合本地的安全策略,那么应该准许该请求。换句话说,即不需额外的身份验证和访问控制就应该授权。此文解释了这种模型,并展示了其实现方式。

Ellison 和 Schneier 的“Ten Risks of PKI: What You’re Not Being Told about Public Key Infrastructure”

要想使用公钥,需要一个可分布和管理密钥的基础设施。构建这样的基础设施并非易事。此文正是解释了其原因所在。

Kaufman 等的《Network Security》

这本权威的、多次再版的书最先介绍了网络安全。书中详尽地阐述了密钥和公钥算法及协议、消息散列、身份验证、Kerberos 和电子邮件。最精彩的部分是作者之间的讨论,以下标形式标记出来,如 I_2 could not get me₁ to be very specific...。

Menezes 等的“Handbook of Applied Cryptography”

书的内容正如其名,提供了理解多种用于加密、散列等的不同加密解决方案所必需的数学背景知识。其中有单独的一章专门介绍身份验证、数字签名、密钥建立和密钥的

管理。

Schneier 的《Applied Cryptography》

一本优秀的全面介绍密码学的书,同时也描述了多种不同安全协议的实现。此书主要是为计算机专家编写的,因此与面向数学的教材相比,此书比较容易理解。此书包含了进一步阅读的参考书目的广泛列表。

Schneier 的《Secrets and Lies》

与《Applied Cryptography》的作者为同一人,此书主要是为非技术人员解释安全性问题。一个重要的观点是:安全性不仅仅是技术问题。实际上,从本书中可以了解到大多数与安全性相关的风险可能都是人为造成的或由我们组织事务的方式造成的。此书还提供了很多材料,可作为本书第 8 章的补充读物。

Sherif 和 Sechrouchni 的《Protocols for Secure Electronic Commerce》

电子商务是安全性扮演关键性角色的领域。此书阐述了用于在非信任网络上建立安全事务的很多协议。主题包括付款系统、B2B 商务、银行卡远程付款、小型付款系统等。

13.1.9 面向对象的分布式系统

Eddon 和 Eddon 的《Inside Distributed COM》

更好地理解分布式 COM 的最好方式是实际其内部是如何工作的。此书提供了 DCOM 的详细技术细节,这些技术细节均由代码专家实例证实。涉及组件、类型库、线程、别名和安全性等。

Emmerich 的《Engineering Distributed Objects》

一本优秀的专门书籍,全面介绍了远程对象技术,特别关注了 CORBA、DCOM 和 Java RMI。而且此书还提供了比较这三种流行的对象模型的良好基础知识。还包含与设计下述系统有关的材料:使用远程对象、处理不同形式的通信、定位对象、持续性、事务和安全性。

Grimshaw 等的“Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems”

Legion 是一种广域的、基于对象的分布式系统(类似于 Globe),目的是对范围广泛的应用程序提供支持,但重点是针对极消耗计算资源的应用程序。Legion 提供了一种远程对象模型。这篇报告提供了 Legion 实际结构的很多细节,最好是将之与第 9 章所讨论的其他系统相比较。

Henning 和 Vinoski 的《Advanced CORBA Programming with C++》

如果需要 CORBA 编程的资料,并且同时需要学习大量关于 CORBA 实质的内容,那

么此书正是最佳选择。此书是由参与了 CORBA 系统的设计与开发的技术人员编写的，此书的实际信息和技术细节性信息并不局限于某个特定的 CORBA 实现。

Siegel 的“CORBA and the OMA in Enterprise Computing”

此文精彩地、浅显易懂地介绍了 CORBA，但没有像本书第 9 章那样详尽。此文主要关注的是 CORBA 体系结构。

13.1.10 分布式文件系统

Groenvall 等的“The Design of a Multicast-based Distributed File System”

分布式文件系统有很多种，通常只在结构上有细微差别。本文描述了一种全然不同的方法，其中一个名为 JetFile 的文件系统充分利用了本书第 7 章所述的可扩展的可靠多播技术。JetFile 允许客户作为服务器运行。

Lee 等的“Operation-based Update Propagation in a Mobile File System”

在支持移动用户的文件系统中，操作中断的一个大问题是更新大型文件的代价相对较高。在这种情况下，当重新连接或使用如无线网络等窄带链接时，客户需要再次把整个文件发送到服务器。本文将给出一个解决方案，即把更新操作转移给一个可以与服务器稳定连接的代理客户。

Rao 和 Peterson 的“Accessing Files in an Internet: The Jade File System”

该文描述了一种早期的文件系统，该文件系统允许用户通过 Internet 把多种文件系统结合在一起。Jade 的命名模型与本书第 4 章的命名图很接近。用户可以在本地建立名称空间，并可依据必要的访问协议，通过指定远程文件服务器的联系地址创建远程名称空间。

Spasojevic 和 Satyanarayanan 的“An Empirical Study of a Wide-Area Distributed File System”

该文汇总了 Andrew File System 的使用特性，Coda 就是从 AFS 发展而来。该文值得注意之处在于，AFS 的能力已经发展到能够支持(AFS)散布于 Internet 上的 1000 台以上的服务器和 20000 台以上的客户。因此，它也很可能是迄今为止开发和使用过的最大的分布式文件系统。该文还阐述了从最初安装到 CMU 的校园网中起，AFS 就不断发挥着重要作用。

Thekkath 等的“Frangipani: A Scalable Distributed File System”

Frangipani 是为工作站群集设计的分布式文件系统。它是双层系统，其下层是提供虚拟磁盘的存储系统。虚拟磁盘由一个很大的稀疏地址空间构成，该空间中的模块只在需要时才进行分配。文件服务器能够很容易共享存储在相同虚拟磁盘上的文件。这种方

法使 Frangipani 的总体设计相对简单。

Triantafillou 和 Neilso 的“Achieving Strong Consistency in a Distributed File System”

为简单起见,绝大多数分布式文件系统都采用了主机一备份复制协议,以实现高度的实用性和性能。该文论述了一种可高效实现 UNIX 文件共享逻辑的复制的写协议。该协议值得注意,它表明当采用复制的写方式时,需要注意很多平衡问题。

13.1.11 基于文档的分布式系统

Dourish 等的“Extending Document Management Systems with User-Specific Active Properties”

该文描述了一种名为 Placeless 的分布式文档管理系统,系统中的文档是根据它们的属性而不是位置来组织的。文档可带附属代码,如日志访问或所生成的摘要。与 Lotus Notes 等系统相比,这些活动属性是 Placeless 的显著特征。

Pierre 等的“Self-Replicating Web Documents”

该报告描述了增强 Web 文档的作用,每个文档能够拥有与自己相关联的复制策略。该策略能动态地自动调整以改变使用模式。可自我复制的 Web 文档能在最大限度地减少带宽占用和客户访问时间的情况下保证强一致性,而这是在所有文档都使用同一个全局策略时所无法实现的。

Qiu 等的“On the Placement of Web Server Replicas”

该文对将 M 个 Web 服务器副本存放到 N 个宿主服务器的网络中的几种算法进行了评估。根据存放成本,验证了一种简单有效的算法:把第一副本以最低存放成本存储到某宿主服务器,第二副本存储到其余宿主服务器中存放成本最低的一个,依次类推,即可得到接近理想化的结果。文中考虑了多种不同的存放成本方案,包括基于距离和请求负载的方案。

Rodriguez 和 Sibal 的“SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution”

该文描述了处理 CDN 的另一种方法。在 SPREAD 中,对代理网络进行动态配置,以决定将 Web 的内容传送到最需要的地方。其显著特点是,代理服务器动态地为每一个 Web 文档做出单独决定:是使用基于拉式还是推式的更新传送,是发送无效的还是完整的更新资料。代理利用网络级的路由信息,把 Web 信息从一个代理推送到另一个代理。

Wang 的“A Survey of Web Caching Schemes for the Internet”

此文概述了缓存静态 Web 文档的几种配置。文中探讨了一些其他主题:体系结构、共享缓存、文档预取、缓存重置算法和缓存相关性。还包括一个很长的参考书目列表。

13.1.12 基于协作的分布式系统

Banavar 等的“*A Case for Message-Oriented Middleware*”

作者讨论了如果在现有组件通信基础上建立一个结构宽松的综合性分布式系统，则应当依据事件和消息的队列进行，并把重点放到发布/订阅(publish/subscribe)结构上。

Carzaniga 等的“*Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service*”

在广域网上完成基于事件的系统可扩展性是一项具有挑战性的任务。该文描述了一种系统，通过应用相对简单的数据模型使订户与发布者相匹配，事件匹配允许高效的基于内容的路由选择。它将基于多点广播的订阅和广告转发结合起来。过滤技术的使用，避免了向不必要的地址发送信息。

Eugster 等的“*The Many Faces of Publish/Subscribe*”

这是一篇优秀且浅显易懂的综述性论文，内容是正在分布式系统中使用的多种发布/订阅模式。文中解释了现有的模式，比如 RPC、消息队列和 tuple 空间，同时还阐述了一些特殊的发布/订阅变型，诸如那些基于主题、内容和类型的变型。

Papadopoulos 和 Arbab 的“*Coordination Models and Languages*”

这是一篇关于协作模型和语言的广泛综述的文章。该文章不以分布式系统为特定目标，但也包括了从分布式系统发展而来的多种模型，例如 Linda。该文以一种简单的模型分类法为基础，使数据驱动模型与控制驱动模型区别开来。

Wyckhoff 等的“*T Spaces*”

对本书第 12 章讨论的元组空间模型(tuple-space model)的一个重要扩展，是支持使用数据库的存储元组。这种应用在 T 空间的方法支持功能强大的搜索并与元组相匹配，类似于传统数据库的做法。而且，使用数据库意味着元组空间附着在一个数据模型上，因此变得更为结构化，反过来，结构化也有助于控制大型元组的复杂程度。

13.2 参考书目列表

- ABADI, M. and NEEDHAM, R. :** “Prudent Engineering Practice for Cryptographic Protocols.” *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 6-15, Jan. 1996.
- ABDULLAHI, S. and RINGWOOD, G. :** “Garbage Collecting the Internet: A Survey of Distributed Garbage Collection.” *ACM Comput. Surv.*, vol. 30, no. 3, pp. 330-373, Sept. 1998.

- ADLER, R.** : "Distributed Coordination Models for Client/Server Computing." *IEEE Computer*, vol. 28, no. 4, pp. 14-22, Apr. 1995.
- ADVE, S. and GHARACHORLOO, K.** : "Shared Memory Consistency Models: A Tutorial." *IEEE Computer*, vol. 29, no. 12, pp. 66-76, Dec. 1996.
- AHAMAD, M., BAZZI, R., JOHN, R., KOHLI, P., and NEIGER, G.** : "The Power of Processor Consistency." Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology, Dec. 1992.
- AHAMAD, M. and KORDALE, R.** : "Scalable Consistency Protocols for Distributed Services." *IEEE Trans. Par. Distr. Syst.*, vol. 10, no. 9, pp. 888-903, Sept. 1999.
- AHUJA, S., CARRIERO, N., GELERNTER, D., and KRISHNASWAMY, V.** : "Matching Languages and Hardware for Parallel Computation in the Linda Machine." *IEEE Trans. Comp.*, vol. 37, no. 8, pp. 921-929, Aug. 1988.
- ALBITZ, P. and LIU, C.** : *DNS and BIND*. Sebastopol, CA: O'Reilly & Associates, 3rd ed., 1998.
- ALVESTRAND, H.** : "Mapping between X. 400 and RFC-822/MIME Message Bodies." RFC 2157, Jan. 1998.
- ALVISI, L. and MARZULLO, K.** : "Message Logging: Pessimistic, Optimistic, Causal, and Optimal." *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 149-159, Feb. 1998.
- AMIR, Y., PETERSON, A., and SHAW, D.** : "Seamlessly Selecting the Best Copy from Internet-Wide Replicated Web Servers." *Proc. Int'l Symp. Distributed Computing (DISC)*, 1998. pp. 22-33.
- ANCEAUME, E. and PUAUT, I.** : "Performance Evaluation of Clock Synchronization Algorithms." Technical Report RR-3526, INRIA, Rennes, France, Oct. 1998.
- ANDERSON, R.** : *Security Engineering-A Guide to Building Dependable Distributed Systems*. New York: John Wiley, 2001.
- ANDERSON, T., BERSHAD, B., LAZOWSKA, E., and LEVY, H.** : "Scheduler Activations: Efficient Kernel Support for the User-Level Management of Parallelism." *Proc. 13th Symp. Operating System Principles*. ACM, 1991. pp. 95-109.
- ANDERSON, T., DAHLIN, M., NEEFE, J., ROSELLI, D., PATTERSON, D., and WANG, R.** : "Serverless Network File Systems." *ACM Trans. Comp. Syst.*, vol. 14, no. 1, pp. 41-79, Feb. 1996.
- ANDERSON, T., CULLER, D., PATTERSON, D., and THE NOW TEAM**: "A Case for NOW." *IEEE Micro*, vol. 15, no. 2, pp. 54-64, Feb. 1995.
- ANDREWS, G.** : *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, MA: Addison-Wesley, 2000.

- ARON, M., SANDERS, D., DRUSCHEL, P., and ZWAENEPOEL, W.** : "Scalable Content-aware Request Distribution in Cluster-based Network Servers." *Proc. Ann. Techn. Conf. USENIX*, 2000. pp. 323-336.
- ASOKAN, N., JANSON, P., STEINER, M., and WAIDNER, M.** : "The State of the Art in Electronic Payment Systems." *IEEE Computer*, vol. 30, no. 9, pp. 28-35, Sept. 1997.
- ATTIYA, H. and WELCH, J.** : "Sequential Consistency versus Linearizability." *ACM Trans. Comp. Syst.*, vol. 12, no. 2, pp. 91-122, May 1994.
- BABAOGLU, O. and MARZULLO, K.** : "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms." In Mullender, S. (ed.), *Distributed Systems*, pp. 55-96. Wokingham: Addison-Wesley, 2nd ed., 1993.
- BABAOGLU, O. and TOUEG, S.** : "Non-Blocking Atomic Commitment." In Mullender, S. (ed.), *Distributed Systems*, pp. 147-168. Wokingham: Addison-Wesley, 2nd ed., 1993.
- BAGGIO, A., BALLINTIJN, G., and VAN STEEN, M.** : "Mechanisms for Effective Caching in the Globe Location Service." *Proc. Ninth SIGOPS European Workshop*. ACM, 2000. pp. 55-60.
- BAGGIO, A., BALLINTIJN, G., VAN STEEN, M., and TANENBAUM, A.** : "Efficient Tracking of Mobile Objects in Globe." *Comp. J.*, vol. 44, no. 5, 2001.
- BAKER, S.** : *CORBA Distributed Objects Using Orbix*. Reading, MA: Addison-Wesley, 1997.
- BAKKEN, D. and SCHLICHTING, R.** : "Supporting Fault-Tolerant Parallel Programming in Linda." *IEEE Trans. Par. Distr. Syst.*, vol. 6, no. 3, pp. 287-302, Mar. 1995.
- BAL, H.** : *The Shared Data-Object Model as a Paradigm for Programming Distributed Systems*. PhD. Thesis, Vrije Universiteit, Amsterdam, 1989.
- BAL, H., BHOEDJANG, R., HOFMAN, R., JACOBS, C., LANGENDOEN, K., RUHL, T., and KAASHOEK, M.** : "Performance Evaluation of the Orca Shared Object System." *ACM Trans. Comp. Syst.*, vol. 16, no. 1, pp. 1-40, Feb. 1998.
- BAL, H. and KAASHOEK, M.** : "Object Distribution in Orca using Compile-Time and Run-Time Techniques." *Proc. Eighth Ann. Conf. Object-Oriented Programming Systems and Languages (OOPSLA)*. ACM, 1993. pp. 162-177.
- BAL, H., KAASHOEK, M., and TANENBAUM, A.** : "Orca: A Language for Parallel Programming of Distributed Systems." *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 190-205, Mar. 1992.
- BALLINTIJN, G., VAN STEEN, M., and TANENBAUM, A.** : "Exploiting Location Awareness for Scalable Location-Independent Object IDs." *Proc. Fifth ASCI Ann. Conf.* ASCI, 1999. pp. 321-328.

- BALLINTIJN, G., VAN STEEN, M., and TANENBAUM, A.:** "A Scalable Implementation for Human-Friendly URIs." *IEEE Internet Comput.*, vol. 5, no. 5, Sept. 2001a.
- BALLINTIJN, G., VERKAIK, P., CRAWL, D., BAGGIO, A., and VAN STEEN, M.:** "The Globe Location Server." Technical Report, Vrije Universiteit, Department of Mathematics and Computer Science, 2001b.
- BANAVAR, G., CHANDRA, T., STROM, R., and STURMAN, D.:** "A Case for Message-Oriented Middleware." In *Proc. DISC*, vol. 1693 of *Lect. Notes Comp. Sc.*, pp. 1-18. Berlin: Springer-Verlag, Sept. 1999a.
- BANAVAR, G., CHANDRA, T., MUKHERJEE, B., NAGARAJARAO, J., STROM, R., and STURMAN, D.:** "An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems." *Proc. 19th Int'l Conf. on Distributed Computing Systems*. IEEE, 1999b.
- BARBER, S.:** "Common NNTP Extensions." RFC 2980, Oct. 2000.
- BARBORAK, M., MALEK, M., and DAHBURA, A.:** "The Consensus Problem in Fault-Tolerant Computing." *ACM Comput. Surv.*, vol. 25, no. 2, pp. 171-220, June 1993.
- BARFORD, P., BESTAVROS, A., BRADLEY, A., and CROVELLA, M. E.:** "Changes in Web Client Access Patterns: Characteristics and Caching Implications." *World Wide Web*, vol. 2, no. 1-2, pp. 15-28, Aug. 1999.
- BARISH, G. and OBRACZKA, K.:** "World Wide Web Caching: Trends and Techniques." *IEEE Commun. Mag.*, vol. 38, no. 5, pp. 178-184, May 2000.
- BARRON, D.:** *Pascal-The Language and its Implementation*. New York: John Wiley, 1981.
- BASS, L., CLEMENTS, P., and KAZMAN, R.:** *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 1998.
- BELL LABS COMPUTING SCIENCE RESEARCH CENTER:** *Plan 9 Programmer's Manual*, Vol. 1. Bell Laboratories, Lucent Technologies, Murray Hill, NJ, 3rd ed., 2000.
- BERNERS-LEE, T., CAILLIAU, R., NIELSON, H. F., and SECRET, A.:** "The World-Wide Web." *Commun. ACM*, vol. 37, no. 8, pp. 76-82, Aug. 1994.
- BERNERS-LEE, T., FIELDING, R., and MASINTER, L.:** "Uniform Resource Identifiers (URI): Generic Syntax." RFC 2396, Aug. 1998.
- BERNSTEIN, P.:** "Middleware: A Model for Distributed System Services." *Commun. ACM*, vol. 39, no. 2, pp. 87-98, Feb. 1996.
- BERNSTEIN, P. and GOODMAN, N.:** "Concurrency Control in Distributed Database Systems." *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185-221, June 1981.
- BERNSTEIN, P., HADZILACOS, V., and GOODMAN, N.:** *Concurrency Control and*

- Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.
- BERSHAD, B., ZEKAUSKAS, M., and SAWDON, W.** : "The Midway Distributed Shared Memory System." *Proc. COMPCON*. IEEE, 1993. pp. 528-537.
- BERSHAD, B., ANDERSON, T., LAZOWSKA, E., and LEVY, H.** : "Lightweight Remote Procedure Call." *ACM Trans. Comp. Syst.*, vol. 8, no. 1, pp. 37-55, Feb. 1990.
- BERSHAD, B. and ZEKAUSKAS, M.** : "Midway: Shared Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors." Technical Report CMU-CS-91-170, Carnegie Mellon University, Sept. 1991.
- BHOEDJANG, R., RUHL, T., HOFMAN, R., LANGENDOEN, K., BAL, H., and KAASHOEK, F.** : "Panda: A Portable Platform to Support Parallel Programming Languages." *Proc. Symp. on Experiences with Distributed and Multiprocessor Systems IV*, 1993. pp. 213-226.
- BHOEDJANG, R., RUHL, T., and BAL, H.** : "User-Level Network Interface Protocols." *IEEE Computer*, vol. 31, no. 11, pp. 53-60, Nov. 1998.
- BHOEDJANG, R.** : *Communication Architectures for Parallel Programming Systems*. Ph. D. Thesis, Vrije Universiteit, Department of Mathematics and Computer Science, Amsterdam, June 2000.
- BIRMAN, K.** : "A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication." *Oper. Syst. Rev.*, vol. 28, no. 1, pp. 11-21, Jan. 1994.
- BIRMAN, K.** : *Building Secure and Reliable Network Applications*. Englewood Cliffs, NJ: Prentice Hall, 1996.
- BIRMAN, K. and JOSEPH, T.** : "Exploiting Virtual Synchrony in Distributed Systems." *Proc. 11th Symp. Operating System Principles*. ACM, 1987a. pp. 123-138.
- BIRMAN, K. and JOSEPH, T.** : "Reliable Communication in the Presence of Failures." *ACM Trans. Comp. Syst.*, vol. 5, no. 1, pp. 47-76, Feb. 1987b.
- BIRMAN, K., SCHIPER, A., and STEPHENSON, P.** : "Lightweight Causal and Atomic Group Multicast." *ACM Trans. Comp. Syst.*, vol. 9, no. 3, pp. 272-314, Aug. 1991.
- BIRMAN, K. and VAN RENESSE, R.** (eds.) : *Reliable Distributed Computing with the Isis Toolkit*. Los Alamitos, CA: IEEE Computer Society Press, 1994.
- BIRRELL, A., EVERS, D., NELSON, G., OWICKI, S., and WOBBER, E.** : "Distributed Garbage Collection for Network Objects." Technical Report SRC-116, Digital Systems Research Center, Palo Alto, CA, Dec. 1993.
- BIRRELL, A., LEVIN, R., NEEDHAM, R., and SCHROEDER, M.** : "Grapevine: An Exercise in Distributed Computing." *Commun. ACM*, vol. 25, no. 4, pp. 260-

274, Apr. 1982.

- BIRRELL, A. and NELSON, B.** : "Implementing Remote Procedure Calls." *ACM Trans. Comp. Syst.*, vol. 2, no. 1, pp. 39-59, Feb. 1984.
- BJORNSON, R.** : *Linda on Distributed Memory Multicomputers*. Ph. D. Thesis, Yale University, Department of Computer Science, 1993.
- BLACK, A. and ARTSY, Y.** : "Implementing Location Independent Invocation." *IEEE Trans. Par. Distr. Syst.*, vol. 1, no. 1, pp. 107-119, Jan. 1990.
- BLAIR, G. and STEFANI, J. -B.** : *Open Distributed Processing and Multimedia*. Reading, MA: Addison-Wesley, 1998.
- BLAKLEY, B.** : *CORBA Security*. Reading, MA: Addison-Wesley, 2000.
- BLAZE, M., FEIGENBAUM, J., IOANNIDIS, J., and KEROMYTIS, A.** : "The Role of Trust Management in Distributed Systems Security." In Vitek, J. and Jensen, C. (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of *Lect. Notes Comp. Sc.*, pp. 185-210. Berlin: Springer-Verlag, 1999.
- BLAZE, M.** : *Caching in Large-Scale Distributed File Systems*. PhD thesis. Department of Computer Science, Princeton University, Jan. 1993.
- BLOOMER, J.** : *Power Programming with RPC*. Sebastopol, CA: O'Reilly & Associates, 1992.
- BODEN, N., COHEN, D., FELDERMAN, R., KULAWIK, A., SEITZ, C., SEIZOVIC, J., and SU, W.** : "Myrinet-A Gigabit-per-Second Local-Area Network." *IEEE Micro*, vol. 15, no. 2, pp. 29-36, Feb. 1995.
- BRACHA, G. and TOUEG, S.** : "Distributed Deadlock Detection." *Distributed Computing*, vol. 2, pp. 127-138, 1987.
- BRADEN, R., ZHANG, L., BERSON, S., HERZOG, S., and JAMIN, S.** : "Resource Reservation Protocol (RSVP)-Version 1 Functional Requirements." RFC 2205, Sept. 1997.
- BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C., and MALER, E.** : "Extensible Markup Language (XML) 1.0 (Second Edition)." W3C Recommendation, Oct. 2000.
- BREWINGTON, B., GRAY, R., MOIZUMI, K., KOTZ, D., CYBENKO, G., and RUS, D.** : "Mobile Agents for Distributed Information Retrieval." In Klusch, M. (ed.), *Intelligent Information Agents*, pp. 355-395. Berlin: Springer-Verlag, 1999.
- BRIOT, J. -P., GUERRAOUI, R., and LOHR, K. -P.** : "Concurrency, Distribution and Parallelism in Object-Oriented Programming." *ACM Comput. Surv.*, vol. 30, no. 3, pp. 291-329, Sept. 1998.
- BUDHIJARA, N., MARZULLO, K., SCHNEIDER, F., and TOUEG, S.** : "The Primary-Backup Approach." In Mullender, S. (ed.), *Distributed Systems*, pp. 199-216.

- Wok-ingham: Addison-Wesley, 2nd ed., 1993.
- BUDHIRAJA, N. and MARZULLO, K.** : "Tradeoffs in Implementing Primary-Backup Protocols." Technical Report TR 92-1307, Department of Computer Science, Cornell University, 1992.
- BURETTA, M.** : *Data Replication: Tools and Techniques for Managing Distributed Information*. New York: John Wiley, 1997.
- CABRI, G. , LEONARDI, L. , and ZAMBONELLI, F.** : "Mobile-Agent Coordination Models for Internet Applications." *IEEE Computer*, vol. 33, no. 2, pp. 82-89, Feb. 2000.
- CALLAGHAN, B.** : *NFS Illustrated*. Reading, MA: Addison-Wesley, 2000.
- CAMP, L.** : *Privacy and Reliability in Internet Commerce*. Ph. D. Thesis, Carnegie Mellon University, Aug. 1996.
- CAO, P. ,ZHANG, J. , and BEACH, K.** : "Active Cache: Caching Dynamic Contents on the Web." *Proc. Middleware '98*. IFIP, 1998. pp. 373-388.
- CAO, P. and LIU, C.** : "Maintaining Strong Cache Consistency in the World Wide Web." *IEEE Trans. Comp.*, vol. 47, no. 4, pp. 445-457, Apr. 1998.
- CARRIERO, N. and GELENTER, D.** : "The S/Net's Linda Kernel." *ACM Trans. Comp. Syst.*, vol. 32, no. 2, pp. 110-129, May 1986.
- CARRIERO, N. and GELENTER, D.** : "How to Write Parallel Programs: A Guide to the Perplexed." *ACM Comput. Surv.*, vol. 21, no. 3, pp. 323-358, 1989.
- CARZANIGA, A. ,ROSENBLUM, D. S. , and WOLF, A. L.** : "Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service." *Proc. 19th Symp. on Principles of Distributed Computing*. ACM, 2000. pp. 219-227.
- CATE, V.** : "Alex-A Global File System." *Proc. File Systems Workshop*. USENIX, 1992. pp. 1-11.
- CHADWICK, D.** : *Understanding X. 500, The Directory*. London: Chapman & Hall, 1994.
- CHANDY, K. and LAMPORT, L.** : "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Trans. Comp. Syst.*, vol. 3, no. 1, pp. 63-75, Feb. 1985.
- CHANG, J. and MAXEMCHUNK, N.** : "Reliable Broadcast Protocols." *ACM Trans. Comp. Syst.*, vol. 2, no. 3, pp. 251-273, Aug. 1984.
- CHANKHUNTHOD, A. , DANZIG, P. , NEERDAELS, C. , SCHWARTZ, M. , and WORRELL, K.** : "A Hierarchical Internet Object Cache." *Proc. Ann. Techn. Conf.* USENIX, 1996. pp. 153-163.
- CHAPPELL, D.** : *Understanding Windows 2000 Distributed Services*. Redmond, WA: Microsoft Press, 2000.
- CHAUM, D.** : "Blind Signatures for Untraceable Payments." *Proc. Crypto '82*, 1982.

pp. 199-203.

- CHAUM, D.** : "Security without Identification: Transaction Systems to make Big Brother Obsolete." *Commun. ACM*, vol. 28, no. 10, pp. 1030-1044, Oct. 1985.
- CHAUM, D.** : "Achieving Electronic Privacy." *Scientific American*, vol. 267, no. 2, pp. 96-101. Aug. 1992.
- CHAWATHE, Y. and BREWER, E.** : "System Support for Scalable and Fault Tolerant Internet Services." *Proc. Middleware '98*. IFIP, 1998. pp. 71-88.
- CHEN, P., LEE, E., GIBSON, G., KATZ, R., and PATTERSON, D.** : "RAID: High-Performance, Reliable Secondary Storage." *ACM Comput. Surv.*, vol. 26, no. 2, pp. 145-186, June 1994.
- CHERITON, D. and MANN, T.** : "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance." *ACM Trans. Comp. Syst.*, vol. 7, no. 2, pp. 147-183, May 1989.
- CHERITON, D. and SKEEN, D.** : "Understanding the Limitations of Causally and Totally Ordered Communication." *Proc. 14th Symp. Operating System Principles*. ACM, 1993. pp. 44-57.
- CHERVENAK, A., FOSTER, I., KESSELMAN, C., SALISBURY, C., and TUECKE, S.** : "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets." *J. Netw. Comp. App.*, vol. 23, no. 3, pp. 187-200, July 2000.
- CHESWICK, W. and BELLOVIN, S.** : *Firewalls and Internet Security*. Reading, MA: Addison-Wesley, 2nd ed., 2000.
- CHOCKLER, G. V., DOLEV, D., FRIEDMAN, R., and VITENBERG, R.** : "Implementing a Caching Service for Distributed CORBA Objects." In *Proc. Middleware 2000*, vol. 1795 of *Lect. Notes Comp. Sc.*, pp. 1-23. Berlin: Springer-Verlag, 2000.
- CHOW, R. and JOHNSON, T.** : *Distributed Operating Systems and Algorithms*. Reading, MA: Addison-Wesley, 1997.
- CHUN, B., MAINWARING, A., and CULLER, D.** : "Virtual Network Transport Protocols for Myrinet." *IEEE Micro*, vol. 18, no. 1, pp. 53-63, Jan. 1998.
- CHUNG, P. E., HUANG, Y., YAJNIK, S., LIANG, D., SMITH, J. C., WANG, C.-Y., and WANG, Y.-M.** : "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer." *C++ Report*, vol. 10, no. 1, pp. 18-29, Jan. 1998.
- CIANCARINI, P., OMICINI, A., and ZAMBONELLI, F.** : "Coordination for Internet Agents." *Nordic J. Comput.*, vol. 6, no. 3, pp. 215-240, 1999.
- CIANCARINI, P., TOLKSDORF, R., VITALI, F., and KNOCHE, A.** : "Coordinating Multi-agent Applications on the WWW: A Reference Architecture." *IEEE Trans. Softw. Eng.*, vol. 24, no. 5, pp. 362-375, May 1998.

- COHEN, D.** : "On Holy Wars and a Plea for Peace." *IEEE Computer*, vol. 14, no. 10, pp. 48-54, Oct. 1981.
- COMER, D.** : *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*. Upper Saddle River, NJ: Prentice Hall, 4th ed., 2000a.
- COMER, D.** : *The Internet Book*. Upper Saddle River, NJ: Prentice Hall, 3rd ed., 2000b.
- COPPERSMITH, D.** : "The Data Encryption Standard (DES) and its Strength Against Attacks." *IBM J. Research and Development*, vol. 38, no. 3, pp. 243-250, May 1994.
- COULOURIS, G., DOLLIMORE, J., and KINDBERG, T.** : *Distributed Systems, Concepts and Design*. Wokingham: Addison-Wesley, 3rd ed., 2001.
- CRISTIAN, F.** : "Probabilistic Clock Synchronization." *Distributed Computing*, vol. 3, pp. 146-158, 1989.
- CRISTIAN, F.** : "Understanding Fault-Tolerant Distributed Systems." *Commun. ACM*, vol. 34, no. 2, pp. 56-78, Feb. 1991.
- CRISTIAN, F. and FETZER, C.** : "The Timed Asynchronous Distributed System Model." *IEEE Trans. Par. Distr. Syst.*, vol. 10, no. 6, pp. 642-657, June 1999.
- CROWCROFT, J., HANDLEY, M., and WAKEMAN, I.** : *Internetworking Multimedia*. London: UCL Press, 1999.
- CROWLEY, C.** : *Operating Systems, A Design-Oriented Approach*. Chicago: Irwin, 1997.
- DAVIDSON, S., GARCIA-MOLINA, H., and SKEEN, D.** : "Consistency in Partitioned Networks." *ACM Comput. Surv.*, vol. 17, no. 3, pp. 341-370, Sept. 1985.
- DAVIES, D. and PRICE, W.** : *Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronic Funds Transfer*. Chichester: John Wiley, 2nd ed., 1989.
- DAY, J. and ZIMMERMAN, H.** : "The OSI Reference Model." *Proceedings of the IEEE*, vol. 71, no. 12, pp. 1334-1340, Dec. 1983.
- DEERING, S., ESTRIN, D., FARINACCI, D., JACOBSON, V., LIU, C. -G., and WEI, L.** : "The PIM Architecture for Wide-Area Multicast Routing." *IEEE/ACM Trans. Netw.*, vol. 4, no. 2, pp. 153-162, Apr. 1996.
- DEERING, S. and CHERITON, D.** : "Multicast Routing in Datagram Internetworks and Extended LANs." *ACM Trans. Comp. Syst.*, vol. 8, no. 2, pp. 85-110, May 1990.
- DEITEL, H., DEITEL, P., and NIETO, T.** : *Internet and World Wide Web - How to Program*. Upper Saddle River, NJ: Prentice Hall, 2000.
- DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S.,**

- STURGIS, H., SWINEHART, D., and TERRY, D.** : "Epidemic Algorithms for Replicated Data Management." *Proc. Sixth Symp. on Principles of Distributed Computing*. ACM, 1987. pp. 1-12.
- DIERKS, T. and ALLEN, C.** : "The Transport Layer Security Protocol." RFC 2246, Jan. 1996.
- DIFFIE, W. and HELLMAN, M.** : "New Directions in Cryptography." *IEEE Trans. Information Theory*, vol. IT-22, no. 6, pp. 644-654, Nov. 1976.
- DIMITROV, B. and REGO, V.** : "Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms." *IEEE Trans. Par. Distr. Syst.*, vol. 9, no. 5, pp. 459-469, May 1998.
- DOURISH, P., EDWARDS, K., LAMARCA, A., LAMPING, J., PETERSEN, K., SALISBURY, M., TERRY, D., and THORNTON, J.** : "Extending Document Management Systems with User-Specific Active Properties." *ACM Trans. Inf. Syst.*, vol. 18, no. 2, pp. 140-170, Apr. 2000.
- DRUMMOND, R. and BABAOGLU, O.** : "Low-Cost Clock Synchronization." *Distributed Computing*, vol. 6, pp. 193-203, 1993.
- DUBOIS, M., SCHEURICH, C., and BRIGGS, F.** : "Synchronization, Coherence, and Event Ordering in Multiprocessors." *IEEE Computer*, vol. 21, no. 2, pp. 9-21, Feb. 1988.
- DUVVURI, V., SHENOY, P., and TEWARI, R.** : "Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web." *Proc. 19th INFOCOM Conf.* IEEE, 2000. pp. 834-843.
- EDDON, G. and EDDON, H.** : *Inside Distributed COM*. Redmond, WA: Microsoft Press, 1998.
- EISLER, M.** : "LIPKEY-A Low Infrastructure Public Key Mechanism Using SPKM." RFC 2847, June 2000.
- EISLER, M., CHIU, A., and LING, L.** : "RPCSEC_GSS Protocol Specification." RFC 2203, Sept. 1997.
- ELLISON, C. and SCHNEIER, B.** : "Ten Risks of PKI: What You're Not Being Told about Public Key Infrastructure." *Computer Security Journal*, vol. 16, no. 1, pp. 1-7, Jan. 2000.
- ELNOZAHY, E., JOHNSON, D., and ZWAENEPOEL, W.** : "The Performance of Consistent Checkpointing." *Proc. 12th Symp. on Reliable Distributed Systems*. IEEE, 1992. pp. 39-47.
- ELNOZAHY, E., JOHNSON, D., and WANG, Y.** : "A Survey of Rollback-Recovery Protocols in Message-Passing Systems." Technical Report CMU-CS-96-181, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Oct. 1996.
- EMMERICH, W.** : *Engineering Distributed Objects*. New York: John Wiley, 2000.

- ESWARAN, K. ,GRAY,J. ,LORIE,R. , and TRAIGER, I.** : "The Notions of Consistency and Predicate Locks in a Database System." *Commun. ACM*, vol. 19, no. 11, pp. 624-633, Nov. 1976.
- EUGSTER, P. , FELBER, P. , GUERRAOUI, R. , and KERMARREC, A. -M.** : "The Many Faces of Publish/Subscribe." Technical Report MSR-TR-2001-104, Microsoft Research Laboratories, Cambridge, UK, Jan. 2001.
- FARMER, W. M. ,GUTTMAN,J. D. , and SWARUP,V.** : "Security for Mobile Agents: Issues and Requirements." *Proc. 19th National Information Systems Security Conf.*, 1996. pp. 591-597.
- FIELDING, R. ,GETTYS, J. ,MOGUL, J. ,FRYSTYK, H. ,MASINTER, L. ,LEACH, P. , and BERNERS-LEE, T.** : "Hypertext Transfer Protocol - HTTP/1. 1." RFC 2616, June 1999.
- FIPA:** *FIPA 97 Specification, Version 2. 0 - Agent Communication Language.* Foundation for Intelligent Physical Agents, Geneva, Oct. 1998a.
- FIPA:** *FIPA 98 Specification, Version 1. 0 - Agent Management.* Foundation for Intelligent Physical Agents, Geneva, Oct. 1998b.
- FISCHER, M. , LYNCH, N. , and PATTERSON, M.** : "Impossibility of Distributed Consensus with one Faulty Processor." *J. ACM*, vol. 32, no. 2, pp. 374-382, Apr. 1985.
- FLOYD,S. ,JACOBSON,V. ,MCCANNE,S. ,LIU,C. -G. , and ZHANG, L.** : "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing." *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 784-803, Dec. 1997.
- FOSTER, I. and KESSELMAN, C.** (eds.): *Computational Grids: The Future of High Performance Distributed Computing.* San Mateo, CA: Morgan Kaufman, 1998.
- FOSTER, I. , KESSELMAN, C. , TSUDIK, G. , and TUECKE, S.** : "A Security Architecture for Computational Grids." *Proc. Fifth Conf. Computer and Communications Security.* ACM, 1998. pp. 83-92.
- FOWLER, R.** : *Decentralized Object Finding Using Forwarding Addresses.* Ph. D. Thesis, University of Washington, Seattle, 1985.
- FOX, A. ,GRIBBLE, S. D. ,CHAWATHE, Y. ,BREWER, E. A. , and GAUTHIER, P.** : "Cluster-Based Scalable Network Services." *Proc. 16th Symp. Operating System Principles.* ACM, 1997. pp. 78-91.
- FRANKLIN, M. J. ,CAREY, M. J. , and LIVNY, M.** : "Transactional Client-Server Cache Consistency: Alternatives and Performance." *ACM Trans. Database Syst.*, vol. 22, no. 3, pp. 315-363, Sept. 1997.
- FRANKLIN, S. and GRAESSER, A.** : "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents." *Proc. Third Int'l Workshop on Agent Theories, Architectures, and Languages.* 1996. pp. 21-35.

- FREDRICKSON, N. and LYNCH, N.** : "Electing a Leader in a Synchronous Ring." *J. ACM*, vol. 34, no. 1, pp. 98-115, Jan. 1987.
- FREED, N. and BORENSTEIN, N.** : "Multipurpose Internet Mail Extensions(MIME) Part Two: Media Types." RFC 2046, Nov. 1996.
- FREEMAN, E., HUPFER, S., and ARNOLD, K.** : *JavaSpaces, Principles, Patterns and Practice*. Reading, MA: Addison-Wesley, 1999.
- FUGGETTA, A., PICCO, G. P., and VIGNA, G.** : "Understanding Code Mobility." *IEEE Trans. Softw. Eng.*, vol. 24, no. 5, pp. 342-361, May 1998.
- GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J.** : *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.
- GARBINATO, B., GUERRAOUI, R., and MAZOUNI, K.** : "Distributed Programming in GARF." In Guerraoui, R., Nierstrasz, O., and Riveill, M. (eds.), *Object-Based Distributed Programming*. vol. 791 of *Lect. Notes Comp. Sc.*, pp. 225-239. Berlin: Springer-Verlag, 1994.
- GARCIA-MOLINA, H.** : "Elections in a Distributed Computing System." *IEEE Trans. Comp.*, vol. 31, no. 1, pp. 48-59, Jan. 1982.
- GARCIA-MOLINA, H., ULLMAN, J. D., and WIDOM, J.** : *Database System Implementation*. Upper Saddle River, NJ: Prentice Hall, 2000.
- GARLAN, D.** : "Software Architecture: A Roadmap." *Proc. 22nd Int'l Conf. Future of Software Engineering*. ACM, 2000. pp. 93-101.
- GEIHS, K.** : "Middleware Challenges Ahead." *IEEE Computer*, vol. 34, no. 6, pp. 24-31, June 2001.
- GELERNTER, D.** : "Generative Communication in Linda." *ACM Trans. Prog. Lang. Syst.*, vol. 7, no. 1, pp. 80-112, 1985.
- GELERNTER, D. and CARRIERO, N.** : "Coordination Languages and their Significance." *Commun. ACM*, vol. 35, no. 2, pp. 96-107, Feb. 1992.
- GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., and HENNESSY, J.** : "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors." *Proc. 17th Ann. Int'l Symp. on Computer Architecture*. ACM, 1990. pp. 15-26.
- GIBSON, G. A., NAGLE, D. F., KHALIL AMIRI, BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., and ZELENKA, J.** : "A Cost-Effective, High-Bandwidth Storage Architecture." *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*. ACM, 1998. pp. 1-12.
- GIFFORD, D.** : "Weighted Voting for Replicated Data." *Proc. Seventh Symp. Operating System Principles*. ACM, 1979. pp. 150-162.

- GIFFORD, D.** : "Cryptographic Sealing." *Commun. ACM*, vol. 25, no. 4, pp. 274-286, Apr. 1982.
- GILMAN, L. and SCHREIBER, R.** : *Distributed Computing with IBM MQSeries*. New York: John Wiley, 1996.
- GLADNEY, H.** : "Access Control for Large Collections." *ACM Trans. Inf. Syst.*, vol. 15, no. 2, pp. 154-194, Apr. 1997.
- GOLAND, Y., WHITEHEAD, E., FAIZI, A., CARTER, S., and JENSEN, D.** : "HTTP Extensions for Distributed Authoring-WEBDAV." RFC 2518, Feb. 1999.
- GOLLMANN, D.** : *Computer Security*. New York: John Wiley, 1999.
- GONG, L. and SCHEMERS, R.** : "Implementing Protection Domains in the Java Development Kit 1.2." *Proc. Symp. Network and Distributed System Security*. Internet Society, 1998. pp. 125-134.
- GOODMAN, J.** : "Cache Consistency and Sequential Consistency." Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.
- GRAY, C. and CHERITON, D.** : "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency." *Proc. 12th Symp. Operating System Principles*. ACM, 1989. pp. 202-210.
- GRAY, J., HELLAND, P., O'NEIL, P. and SASHNA, D.** : "The Dangers of Replication and a Solution." *Proc. SIGMOD Int'l Conf. on Management Of Data*. ACM, 1996. pp. 173-182.
- GRAY, J. and REUTER, A.** : *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufman, 1993.
- GRAY, J.** : "Notes on Database Operating Systems." In Bayer, R., Graham, R., and Seeg-muller, G. (eds.), *Operating Systems: An Advanced Course*. vol. 60 of *Lect. Notes Comp. Sc.*, pp. 393-481. Berlin: Springer-Verlag, 1978.
- GRAY, R.** : "Agent Tcl: Alpha Release 1.1." Technical Report, Dartmouth College, Hanover, NH, Dec. 1995.
- GRAY, R.** : "Agent Tcl: A Flexible and Secure Mobile-Agent System." *Proc. Fourth Tcl/Tk Workshop*. USENIX, 1996b. pp. 9-23.
- GRIMSHAW, A., FERRARI, A., KNABE, F., and HUMPHREY, M.** : "Wide-Area Computing: Resource Sharing on a Large Scale." *IEEE Computer*, vol. 32, no. 5, pp. 29-37, May 1999.
- GRIMSHAW, A., LEWIS, M., FERRARI, A., and KARPOVICH, J.** : "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems." Technical Report CS-98-12, University of Virginia, Department of Computer Science, June 1998.
- GROENVALL, B., WESTERLUND, A., and PINK, S.** : "The Design of a Multicast-based Distributed File System." *Proc. Third Symp. on Operating System Design*

- and Implementation.* USENIX, 1999. pp. 251-264.
- GROPP, W., HUSS-LEDERMAN, S., LUMSDAINE, A., LUSK, E., NITZBERG, B., SAPHIR, W., and SNIR, M.:** *MPI: The Complete Reference - The MPI-2 Extensions.* Cambridge, MA: MIT Press, 1998a.
- GROPP, W., LUSK, E., and SKJELLUM, A.:** *Using MPI, Portable Parallel Programming with the Message-Passing Interface.* Cambridge, MA: MIT Press, 2nd ed., 1998b.
- GUERRAOUI, R. and SCHIPER, A.:** "The Generic Consensus Service." *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 29-41, Jan. 2001.
- GUERRAOUI, R. and SCHIPER, A.:** "Software-Based Replication for Fault Tolerance." *IEEE Computer*, vol. 30, no. 4, pp. 68-74, Apr. 1997.
- GUSELLA, R. and ZATTI, S.:** "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4. 3BSD." *IEEE Trans. Softw. Eng.*, vol. 15, no. 7, pp. 847-853, July 1989.
- GUYTON, J. and SCHWARTZ, M.:** "Locating Nearby Copies of Replicated Internet Services." *Proc. SIGCOMM*. ACM, 1995. pp. 288-298.
- GWERTZMAN, J. and SELTZER, M.:** "The Case for Geographical Push-Caching." *Proc. Fifth Workshop Hot Topics in Operating Systems*. IEEE, 1996. pp. 51-55.
- HADZILACOS, V. and TOUEG, S.:** "Fault-Tolerant Broadcasts and Related Problems." In Mullender, S. (ed.), *Distributed Systems*. pp. 97-145. Wokingham: Addison-Wesley, 2nd ed., 1993.
- HALSALL, F.:** *Multimedia Communications: Applications, Networks, Protocols and Standards.* Reading, MA: Addison-Wesley, 2001.
- HAMILTON, G. and KOUGIOURIS, P.:** "The Spring Nucleus: A Microkernel for Objects." *Proc. Summer Techn. Conf.* USENIX, 1993. pp. 147-160.
- HANDEL, R., HUBER, M., and SCHRODER, S.:** *ATM Networks.* Wokingham: Addison-Wesley, 2nd ed., 1994.
- HARTMAN, J. and OUSTERHOUT, J.:** "The Zebra Striped Network File System." *ACM Trans. Comp. Syst.*, vol. 13, no. 3, pp. 274-310, Aug. 1995.
- HAYES, C. C.:** "Agents in a Nutshell - A Very Brief Introduction." *IEEE Trans. Know. Data Eng.*, vol. 11, no. 1, pp. 127-132, Jan. 1999.
- HELARY, J.:** "Observing Global States of Asynchronous Distributed Applications." In *Proc. Int'l Workshop on Distributed Algorithms*, vol. 392 of *Lect. Notes Comp. Sc.*, pp. 124-135. Berlin: Springer-Verlag, 1989.
- HENNING, M. and VINOSKI, S.:** *Advanced CORBA Programming with C++.* Reading, MA: Addison-Wesley, 1999.
- HERLIHY, M. and WING, J.:** "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Trans. Prog. Lang. Syst.*, vol. 12, no. 3, pp. 463-495.

492, July 1991.

- HOARE, C.** : "Monitors: An Operating System Structuring Concept." *Commun. ACM*, vol. 17, no. 10, pp. 549-557, Oct. 1974.
- HOFMANN, M.** : "A Generic Concept for Large-Scale Multicast." In Plattner, B. (ed.), *Broadband Communications-Networks, Services, Applications, Future Directions*. vol. 1044 of *Lect. Notes Comp. Sc.*, pp. 95-106. Berlin: Springer-Verlag, 1996.
- HOMBURG, P. C.** : *The Architecture of a Worldwide Distributed System*. Ph. D. Thesis, Vrije University Amsterdam, Department of Mathematics and Computer Science, 2001.
- HOROWITZ, M. and LUNT, S.** : "FTP Security Extensions." RFC 2228, Oct. 1997.
- HOUTTUIN, J.** : "A Tutorial on Gatewaying between X. 400 and Internet Mail." RFC 1506, Sept. 1993.
- HOWARD, J. , KAZAR, M. , MEENES, S. , NICHOLS, D. , SATYANARAYANAN, M. , SIDEBOOTHAM, R. , and WEST, M.** : "Scale and Performance in a Distributed File System." *ACM Trans. Comp. Syst.*, vol. 6, no. 2, pp. 55-81, Feb. 1988.
- HOWES, T.** : "The String Representation of LDAP Search Filters." RFC 2254, Dec. 1997.
- HUDSON, R. L. , MORRISON, R. , MOSS, E. B. , and MUNRO, D. S.** : "Garbage Collecting the World: One Car at a Time." *SIGPLAN Notices*, vol. 32, no. 10, pp. 162-175, Oct. 1997.
- HUTTO, P. and AHAMAD, M.** : "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories." *Proc. Tenth Int'l Conf. on Distributed Computing Systems*. IEEE, 1990. pp. 302-311.
- ISLAM, N.** : *Distributed Objects, Methodologies for Customizing Systems Software*. Los Alamitos, CA: IEEE Computer Society Press, 1996.
- ISO**: "Open Distributed Processing Reference Model." International Standard ISO/IEC IS 10746, 1995.
- JAEGER, T. , PRAKASH, A. , LIEDTKE, J. , and ISLAM, N.** : "Flexible Control of Downloaded Executable Content." *ACM Trans. Inf. Syst. Sec.*, vol. 2, no. 2, pp. 177-228, May 1999.
- JAIN, R.** : "Reducing Traffic Impacts of PCS using Hierarchical User Location Databases." *Proc. Int'l Conf. Communications*. IEEE, 1996.
- JALOTE, P.** : *Fault Tolerance in Distributed Systems*. Englewood Cliffs, NJ: Prentice Hall, 1994.
- JANSEN, M. , KLAVER, E. , VERKAIK, P. , VAN STEEN, M. , and TANENBAUM, A.** : "Encapsulating Distribution in Remote Objects." *Information and Software Technology*, vol. 43, no. 6, pp. 353-363, May 2001.

- JENNINGS, N. and WOOLDRIDGE, M.** : "Applications of Intelligent Agents." In Jennings, N. and Wooldridge, M. (eds.), *Agent Technology Foundations, Applications, and Markets*. pp. 3-26. Berlin: Springer-Verlag, 1998.
- JING, J., HELAL, A., and ELMAGARMID, A.** : "Client-Server Computing in Mobile Environments." *ACM Comput. Surv.*, vol. 31, no. 2, pp. 117-157, June 1999.
- JOHNER, H., BROWN, L., HINNER, F.-S., REIS, W., and WESTMAN, J.** : "Understanding LDAP." Technical Report SG24-4986-00, International Technical Support Organization, IBM, Austin, TX, May 1998.
- JOHNSON, B.** : "An Introduction to the Design and Analysis of Fault-Tolerant Systems." In Pradhan, D. K. (ed.), *Fault-Tolerant Computer System Design*. pp. 1-87. Upper Saddle River, NJ: Prentice Hall, 1995.
- JOHNSON, K., KAASHOEK, M., and WALLACH, D.** : "CRL: High-Performance All-Software Distributed Shared Memory." *Proc. 15th Symp. Operating System Principles*. ACM, 1995. pp. 1-16.
- JUL, E., LEVY, H., HUTCHINSON, N., and BLACK, A.** : "Fine-Grained Mobility in the Emerald System." *ACM Trans. Comp. Syst.*, vol. 6, no. 1, pp. 109-133, Feb. 1988.
- JUSZCZAK, C.** : "Improving the Performance and Correctness of an NFS Server." *Proc. Summer Techn. Conf.* USENIX, 1990. pp. 53-63.
- KAHN, D.** : *The Codebreakers*. New York: Macmillan, 1967.
- KANTOR, B. and LAPSLEY, P.** : "Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News." RFC 977, Feb. 1986.
- KAPPE, F.** : *Hyperwave Information Server*. Hyperwave Research & Development, June 1999.
- KARNIK, N. and TRIPATHI, A.** : "Security in the Ajanta Mobile Agent System." *Software-Practice & Experience*, vol. 31, no. 4, pp. 301-329, Apr. 2001.
- KASERA, S., KUROSE, J., and TOWSLEY, D.** : "Scalable Reliable Multicast Using Multiple Multicast Groups." *Proc. Int'l Conf. Measurements and Modeling of Computer Systems*. ACM, 1997. pp. 64-74.
- KATZ, E., BUTLER, M., and MCGRATH, R.** : "A Scalable HTTP Server: The NCSA Prototype." *Comp. Netw. & ISDN Syst.*, vol. 27, no. 2, pp. 155-164, Sept. 1994.
- KAUFMAN, C., PERLMAN, R., and SPECINER, M.** : *Network Security: Private Communication in a Public World*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- KAWELL, L., BECKHARDT, S., HALVORSEN, T., OZZIE, R., and GREIF, I.** : "Replicated Document Management in a Group Communication System." *Proc. Second Conf. Computer-Supported Cooperative Work*, 1988. pp. 226-235.
- KEITH, E. W.** : *Core Jini*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 2000.

- KELEHER, P. , COX, A. , and ZWAENEPOEL, W.** : "Lazy Release Consistency." *Proc. 19th Ann. Int'l Symp. on Computer Architecture*. ACM, 1992. pp. 13-21.
- KENT, S.** : "Internet Privacy Enhanced Mail." *Commun. ACM*, vol. 36, no. 8, pp. 48-60, Aug. 1993.
- KERMARREC, A. , KUZ, I. , VAN STEEN, M. , and TANENBAUM, A.** : "A Framework for Consistent, Replicated Web Objects." *Proc. 18th Int'l Conf. on Distributed Computing Systems*. IEEE, 1998. pp. 276-284.
- KHOSHAFFIAN, S. and BUCKIEWICZ, M.** : *Introduction to Groupware, Workflow, and Workgroup Computing*. New York: John Wiley, 1995.
- KISTLER, J.** : *Disconnected Operation in a Distributed File System*, vol. 1002 of *Lect. Notes Comp. Sc.* Berlin: Springer-Verlag, 1996.
- KISTLER, J. and SATYANARYANAN, M.** : "Disconnected Operation in the Coda File System." *ACM Trans. Comp. Syst.*, vol. 10, no. 1, pp. 3-25, Feb. 1992.
- KLEIMAN, S.** : "Vnodes: an Architecture for Multiple File System Types in UNIX." *Proc. Summer Techn. Conf. USENIX*, 1986. pp. 238-247.
- KLESSIG, R. and TESINK, K.** : *SMDS. Wide-Area Data Networking with Switched Multi-megabit Data Service*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- KOHL, J. , NEUMAN, B. , and T'SO, T.** : "The Evolution of the Kerberos Authentication System." In Brazier, F. and Johansen, D. (eds.), *Distributed Open Systems*. pp. 78-94. Los Alamitos, CA: IEEE Computer Society Press, 1994.
- KOPETZ, H. and OCHSENREITER, W.** : "Clock Synchronization in Distributed Real-Time Systems." *IEEE Trans. Comp.*, vol. C-87, no. 8, pp. 933-940, Aug. 1987.
- KOPETZ, H. and VERRISSIMO, P.** : "Real Time and Dependability Concepts." In Mulder, S. (ed.), *Distributed Systems*. pp. 411-446. Wokingham: Addison-Wesley, 2nd ed., 1993.
- KOTZ, D. , GRAY, R. , NOG, S. , RUS, D. , CHAWLA, S. , and CYBENKO, G.** : "Agent Tcl: Targeting the Needs of Mobile Computers." *IEEE Internet Comput.*, vol. 1, no. 4, pp. 58-67, July 1997.
- KUMAR, P. and SATYANARAYANAN, M.** : "Flexible and Safe Resolution of File Conflicts." *Proc. Winter Techn. Conf. USENIX*, 1995. pp. 95-106.
- KUNG, H. and ROBINSON, J.** : "On Optimistic Methods for Concurrency Control." *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213-226, June 1981.
- KUROSE, J. F. and ROSS, K. W.** : *Computer Networking*. Reading, MA: Addison-Wesley, 2001.
- LADIN, R. , LISKOV, B. , SHIRA, L. , and GHEMAWAT, S.** : "Providing Availability Using Lazy Replication." *ACM Trans. Comp. Syst.*, vol. 10, no. 4, pp. 360-391, Nov. 1992.

- LAI, C. , GONG, L. , KOVED, L. , NADALIN, A. , and SCHEMERS, R.** : "User Authentication and Authorization in the Java Platform." *Proc. 15th Ann. Computer Security Applications Conf.* IEEE, 1999. pp. 285-290.
- LAMACCHIA, B. and ODLYZKO, A.** : "Computation of Discrete Logarithms in Prime Fields." *Designs, Codes, and Cryptography*, vol. 1, no. 1, pp. 47-62, May 1991.
- LAMPORT, L.** : "Time, Clocks, and the Ordering of Events in a Distributed System." *Commun. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.
- LAMPORT, L.** : "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs." *IEEE Trans. Comp.*, vol. C-29, no. 9, pp. 690-691, Sept. 1979.
- LAMPORT, L.** : "Concurrent Reading and Writing of Clocks." *ACM Trans. Comp. Syst.*, vol. 8, no. 4, pp. 305-310, Nov. 1990.
- LAMPORT, L. , SHOSTAK, R. , and PAESE, M.** : "Byzantine Generals Problem." *ACM Trans. Prog. Lang. Syst.*, vol. 4, no. 3, pp. 382-401, July 1982.
- LAMPSON, B.** : "Designing a Global Name Service." *Proc. Fourth Symp. on Principles of Distributed Computing*. ACM, 1986. pp. 1-10.
- LAMPSON, B. , ABADI, M. , BURROWS, M. , and WOBBER, E.** : "Authentication in Distributed Systems: Theory and Practice." *ACM Trans. Comp. Syst.*, vol. 10, no. 4, pp. 265-310, Nov. 1992.
- LANG, B. , QUEINNEC, C. , and PIQUER, J.** : "Garbage Collecting the World." *Proc. Symp. on Principles of Programming Languages*. ACM, 1992. pp. 39-50.
- LAPRIE, J. -C.** : "Dependability-Its Attributes, Impairments and Means." In Randell, B. , Laprie, J. -C. , Kopetz, H. and Littlewood, B. (eds.), *Predictably Dependable Computing Systems*. pp. 3-24. Berlin: Springer-Verlag, 1995.
- LAURIE, B. and LAURIE, P.** : *Apache: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, 2nd ed. , 1999.
- LEE, Y. , LEUNG, K. , and SATYANARAYANAN, M.** : "Operation-based Update Propagation in a Mobile File System." *Proc. Ann. Techn. Conf.* USENIX, 1999. pp. 43-56.
- LE HORS, A. , LE HEGARET, P. , WOOD, L. , NICOL, G. , ROBIE, J. , CHAMPION, M. , and BYRNE, S.** : "Document Object Model(DOM) Level 2 Core Specification." W3C Recommendation, Nov. 2000.
- LEIGHTON, F. and LEWIN, D.** : "Global Hosting System." United States Patent, Number 6,108,703, Aug. 2000.
- LEIWO, J. , HAENLE, C. , HOMBURG, P. , GAMAGE, C. , and TANENBAUM, A.** : "A Security Design for a Wide-Area Distributed System." In *Proc. Second Int'l. • 600 •*

- Conf. Information Security and Cryptology*, vol. 1787 of *Lect. Notes Comp. Sc.*, pp. 236-256. Berlin: Springer-Verlag, 1999.
- LEVINE, B. and GARCIA-LUNA-ACEVES, J.** : "A Comparison of Reliable Multicast Protocols." *ACM Multimedia Systems Journal*, vol. 6, no. 5, pp. 334-348, 1998.
- LEWIS, B. and BERG, D. J.** : *Multithreaded Programming with Pthreads*. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.
- LI, K. and HUDAQ, P.** : "Memory Coherence in Shared Virtual Memory Systems." *ACM Trans. Comp. Syst.*, vol. 7, no. 3, pp. 321-359, Nov. 1989.
- LILJA, D.** : "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons." *ACM Comput. Surv.*, vol. 25, no. 3, pp. 303-338, Sept. 1993.
- LIN, M. -J. and MARZULLO, K.** : "Directional Gossip: Gossip in a Wide-Area Network." In *Proc. Third European Dependable Computing Conf.*, vol. 1667 of *Lect. Notes Comp. Sc.*, pp. 364-379. Berlin: Springer-Verlag, Sept. 1999.
- LINN, J.** : "Generic Security Service Application Program Interface, version 2." RFC 2078, Jan. 1997.
- LIPTON, R. and SANDBERG, J.** : "PRAM: A Scalable Shared Memory." Technical Report CS-TR-180-88, Princeton University, Sept. 1988.
- LISKOV, B.** : "Practical Uses of Synchronized Clocks in Distributed Systems." *Distributed Computing*, vol. 6, pp. 211-219, 1993.
- LIU, C. -G., ESTRIN, D., SHENKER, S., and ZHANG, L.** : "Local Error Recovery in SRM: Comparison of Two Approaches." *IEEE/ACM Trans. Netw.*, vol. 6, no. 6, pp. 686-699, Dec. 1998.
- LOSHIN, P.** (ed.) : *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*. San Mateo, CA: Morgan Kaufman, 2000.
- LOTUS DEVELOPMENT CORP.** : *Inside Notes: The Architecture of Notes and the Domino Server*. Lotus Development Corporation, Cambridge, MA, Oct. 2000.
- LOWE-NORRIS, A.** : *Windows 2000 Active Directory*. Sebastopol, CA: O'Reilly & Associates, 2000.
- LUNDELIUS-WELCH, J. and LYNCH, N.** : "A New Fault-Tolerant Algorithm for Clock Synchronization." *Information and Computation*, vol. 77, no. 1, pp. 1-36, Jan. 1988.
- LUOTONEN, A. and ALTIS, K.** : "World-Wide Web Proxies." *Comp. Netw. & ISDN Syst.*, vol. 27, no. 2, pp. 1845-1855, 1994.
- LYNCH, C., PRESTON, C., and DANIEL, R.** : "Using Existing Bibliographic Identifiers as Uniform Resource Names." RFC 2288, Feb. 1998.
- LYNCH, N.** : *Distributed Algorithms*. San Mateo, CA: Morgan Kaufman, 1996.

- MACGREGOR, R. , DURBIN, D. , OWLETT, J. , and YEOMANS, A. : *Java Network Security*.** Upper Saddle River, NJ: Prentice Hall, 1998.
- MAEKAWA, M. : "A Square-root(N) Algorithm for Mutual Exclusion in Decentralized Systems."** *ACM Trans. Comp. Syst.*, vol. 3, no. 4, pp. 145-159, May 1985.
- MAES, P. : "Agents that Reduce Work and Information Overload."** *Commun. ACM*, vol. 37, no. 7, pp. 31-40, July 1994.
- MAFFEIS, S. : "Piranha: A CORBA Tool For High Availability."** *IEEE Computer*, vol. 30, no. 4, pp. 59-66, Apr. 1997.
- MAKPANGOU, M. , GOURHANT, Y. , LE NARZUL, J. -P. , and SHAPIRO, M. : "Fragmented Objects for Distributed Abstractions."** In Casavant, T. and Singhal, M. (eds.), *Readings in Distributed Computing Systems*. pp. 170-186. Los Alamitos, CA: IEEE Computer Society Press, 1994.
- MALKHI, D. and REITER, M. : "Secure Execution of Java Applets using a Remote Playgrounnd."** *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1197-1209, Dec. 2000.
- MASINTER, L. : "The Data URL Scheme."** RFC 2397, Aug. 1998.
- MATTERN, F. : "Algorithms for Distributed Termination Detection."** *Distributed Computing*, vol. 2, pp. 161-175, 1987.
- MAZIERES, D. : *Self-Certifying File System*.** Ph. D. Thesis, Massachusetts Institute of Technology, May 2000.
- MAZIERES, D. , KAMINSKY, M. , KAASHOEK, M. , and WITCHEL, E. : "Separating Key Management from File System Security."** *Proc. 17th Symp. Operating System Principles*. ACM, 1999. pp. 124-139.
- MAZOUNI, K. , GARBINATO, B. , and GUERRAOUI, R. : "Building Reliable Client-Server Software Using Actively Replicated Objects."** In Graham, I. , Magnusson, B. , Meyer, B. and Nerson, J. -M (eds.), *Technology of Object Oriented Languages and Systems*. pp. 37-53. Englewood Cliffs, NJ: Prentice Hall, 1995.
- MEDVIDOVIC, N. and TAYLOR, R. : "A Classification and Comparison Framework for Software Architecture Description Languages."** *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70-93, Jan. 2000.
- MENEZES, A. J. , VAN OORSCHOT, P. C. , and VANSTONE, S. A. : *Handbook of Applied Cryptography*.** Boca Raton: CRC Press, 3rd ed. , 1996.
- MEYER, B. : *Object-Oriented Software Construction*.** Englewood Cliffs, NJ: Prentice Hall, 2nd ed. , 1997.
- MICROSOFT CORPORATION: *The Component Object Model Specification, Version 0.9*.** Redmond, WA, Oct. 1995.
- MILLS, D. : "Network Time Protocol(version 3): Specification, Implementation, and Analysis."** RFC 1305, July 1992.

- MILLS, D.** : "Improved Algorithms for Synchronizing Computer Network Clocks." *IEEE/ACM Trans. Netw.*, vol. 3, no. 3, pp. 245-254, June 1995.
- MILOJICIC, D., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., and ZHOU, S.** : "Process Migration." *ACM Comput. Surv.*, vol. 32, no. 3, pp. 241-299, Sept. 2000.
- MIN, S. L. and BAER, J. -L.** : "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps." *IEEE Trans. Par. Distr. Syst.*, vol. 3, no. 1, pp. 25-44, Jan. 1992.
- MITCHELL, J., GIBBONS, J., HAMILTON, G., KESSLER, P., KHALIDI, Y., KOUGIOURIS, P., MADANY, P., NELSON, M., POWELL, M., and RADIA, S. R.** : "An Overview of the Spring System." *Proc. 39th Int'l. Computer Conf.* IEEE, 1994. pp. 122-131.
- MIZUNO, M., RAYNAL, M., and ZHOU, J. Z.** : "Sequential Consistency in Distributed Systems." In Birman, K., Mattern, F., and Schiper, A. (eds.), *Theory and Practice in Distributed Systems*. vol. 938 of *Lect. Notes Comp. Sc.*, pp. 224-241. Berlin: Springer-Verlag, 1995.
- MOATS, R.** : "URN Syntax." RFC 2141, May 1997.
- MOATS, R.** : "A URN Namespace for IETF Documents." RFC 2648, Aug. 1999.
- MOCKAPETRIS, P.** : "Domain Names - Concepts and Facilities." RFC 1034, Nov. 1987.
- MOHAN, S. and JAIN, R.** : "Two User Location Strategies for Personal Communication Services." *IEEE Pers. Commun.*, vol. 1, no. 1, pp. 42-50, Jan. 1994.
- MOSBERGER, D.** : "Memory Consistency Models." *Oper. Syst. Rev.*, vol. 27, no. 1, pp. 18-26, Jan. 1993.
- MOSER, L., MELLIOR-SMITH, P., AGARWAL, D., BUDHIA, R., and LINGLEY-PAPADOPOULOS, C.** : "Totem: A Fault-Tolerant Multicast Group Communication System." *Commun. ACM*, vol. 39, no. 4, pp. 54-63, Apr. 1996.
- MOSER, L., MELLIOR-SMITH, P., and NARASIMHAM, P.** : "Consistent Object Replication in the Eternal System." *Theory and Practice of Object Systems*, vol. 4, no. 2, pp. 81-92, 1998.
- MULLENDER, S.** : *Distributed Systems*. Wokingham: Addison-Wesley, 2nd ed., 1993.
- MULLENDER, S. and TANENBAUM, A.** : "Immediate Files." *Software-Practice & Experience*, vol. 14, no. 3, pp. 365-368, 1984.
- MUNTZ, D. and HONEYMAN, P.** : "Multi-level Caching in Distributed File Systems." *Proc. Winter Techn. Conf.* USENIX, 1992. pp. 305-313.
- NARASIMHAM, P., MOSER, L., and MELLIOR-SMITH, P.** : "Using Interceptors to Enhance CORBA." *IEEE Computer*, vol. 32, no. 7, pp. 62-68, July 1999.

- NARASIMHAN, P., MOSER, L., and MELLiar-SMITH, P.** : "The Eternal System." In Urban, J. and Dasgupta, P. (eds.), *Encyclopedia of Distributed Computing*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2000.
- NEEDHAM, R. and SCHROEDER, M.** : "Using Encryption for Authentication in Large Networks of Computers." *Commun. ACM*, vol. 21, no. 12, pp. 993-999, Dec. 1978.
- NEEDHAM, R.** : "Names." In Mullender, S. (ed.), *Distributed Systems*. pp. 315-327. Wokingham: Addison-Wesley, 2nd ed., 1993.
- NELSON, B.** : *Remote Procedure Call*. Ph. D. Thesis, Carnegie-Mellon University, 1981.
- NEUMAN, B.** : "Scale in Distributed Systems." In Casavant, T. and Singhal, M. (eds.), *Readings in Distributed Computing Systems*. pp. 463-489. Los Alamitos, CA: IEEE Computer Society Press, 1994.
- NEUMAN, B.** : "Proxy-Based Authorization and Accounting for Distributed Systems." *Proc. 13th Int'l Conf. on Distributed Computing Systems*. IEEE, 1993. pp. 283-291.
- NEUMANN, P.** : "Architectures and Formal Representations for Secure Systems." Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, Oct. 1995.
- NIELSEN, S., DAHM, F., LUSCHER, M., YAMAMOTO, H., COLLINS, F., DENHOLM, B., KUMAR, S., and SOFTLEY, J.** : "Lotus Notes and Domino R5.0 Security Infrastructure Revealed." Technical Report SG24-5341-00, International Technical Support Organization, IBM, Austin, TX, May 1999.
- NOBLE, B., FLEIS, B., and KIM, M.** : "A Case for Fluid Replication." *Proc. NetStore'99*, 1999.
- NUTT, G.** : *Operating Systems, A Modern Perspective*. Reading, MA: Addison-Wesley, 2nd ed., 2000.
- NWANA, H.** : "Software Agents: An Overview." *Know. Eng. Rev.*, vol. 11, no. 3, pp. 205-244, Oct. 1996.
- NWANA, H. and NDUMU, D.** : "A Perspective on Software Agents Research." *Know. Eng. Rev.*, vol. 14, no. 2, pp. 1-18, Jan. 1999.
- OBRACZKA, K.** : "Multicast Transport Protocols: A Survey and Taxonomy." *IEEE Commun. Mag.*, vol. 36, no. 1, pp. 94-102, Jan. 1998.
- OKI, B., PFLUEGL, M., SIEGEL, A., and SKEEN, D.** : "The Information Bus - An Architecture for Extensible Distributed Systems." *Proc. 14th Symp. Operating System Principles*. ACM, 1993. pp. 58-68.
- OMG** : "Object Management Architecture Guide, Revision 3.0." OMG Document ab/97-05-05, Object Management Group, Framingham, MA, Jan. 1997.

- OMG:** "CORBAServices: Notification Service Specification." OMG Document formal/00-06-20. Object Management Group, Framingham, MA, June 2000a.
- OMG:** "CORBAServices: Trading Service Specification." OMG Document formal/00-06-27, Object Management Group, Framingham, MA, May 2000b.
- OMG:** "CORBAServices: Concurrency Service Specification." OMG Document formal/00-06-14. Object Management Group, Framingham, MA, Apr. 2000c.
- OMG:** "Fault Tolerant CORBA." OMG Document ptc/00-04-04, Object Management Group, Framingham, MA, Apr. 2000d.
- OMG:** "Mobile Agent Facility Specification." OMG Document formal/00-01-02, Object Management Group, Framingham, MA, Jan. 2000e.
- OMG:** "CORBAServices: Naming Service Specification." OMG Document formal/01-02-65, Object Management Group, Framingham, MA, Feb. 2001a.
- OMG:** "The Common Object Request Broker: Architecture and Specification, revision 2.4.2." OMG Document formal/00-02-33. Object Management Group, Framingham, MA, Feb. 2001b.
- ORAM, A. (ed.):** *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA: O'Reilly & Associates, 2001.
- ORFALI, R., HARKEY, D., and EDWARDS, J.:** *The Essential Distributed Objects Survival Guide*. New York: John Wiley, 1996.
- ORGANICK, E.:** *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972.
- OSF:** *OSF DCE 1.2.2 Application Development Guide-Core Components*. The Open Group, 1997.
- OUSTERHOUT, J.:** *Tcl and the Tk Toolkit*. Reading, MA: Addison-Wesley, 1994.
- OZSU, T. and VALDURIEZ, P.:** *Principles of Distributed Database Systems*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 1999.
- PAGE, T., GUY, R., HEIDEMANN, J., RATNER, R., REIHER, P., GOEL, A., KUENNING, G., and POPEK, G.:** "Perspectives on Optimistically Replicated, Peer-to-Peer Filing." *Software-Practice & Experience*, vol. 28, no. 2, pp. 155-180, Feb. 1998.
- PAI, V., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWAENEPOEL, W., and NAHUM, E.:** "Locality-Aware Request Distribution in Cluster-Based Network Servers." *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*. ACM, 1998. pp. 205-216.
- PANZIERI, F. and SHRIVASTAVA, S.:** "Rajdoot: A Remote Procedure Call Mechanism with Orphan Detection and Killing." *IEEE Trans. Softw. Eng.*, vol. 14, no. 1, pp. 30-37, Jan. 1988.
- PAPADOPOULOS, G. and ARBAB, F.:** "Coordination Models and Languages." In

- Zel-kowitz, M. (ed.), *Advances in Computers*, vol. 46, pp. 329-400. New York, NY: Academic Press, Sept. 1998.
- PARKER, T. and PIKAS, D.** : "SESAME V4 Overview." Technical Report, SESAME Consortium, Dec. 1995.
- PARTRIDGE, C.** : "A Proposed Flow Specification." RFC 1363, Sept. 1992.
- PARTRIDGE, C.** : *Gigabit Networking*. Reading, MA: Addison-Wesley, 1994.
- PARTRIDGE, C., MENDEZ, T., and MILLIKEN, W.** : "Host Anycasting Service." RFC 1546, Nov. 1993.
- PAWLOWSKI, B., JUSZCZAK, C., STAUBACH, P., SMITH, C., HEBEL, D., and HITZ, D.** : "NFS Version 3 Design and Implementation." *Proc. Summer Techn. Conf.* USENIX, 1994. pp. 137-152.
- PEASE, M., SHOSTAK, R., and LAMPORT, L.** : "Reaching Agreement in the Presence of Faults." *J. ACM*, vol. 27, no. 2, pp. 228-234, Apr. 1980.
- PERKINS, C.** : *Mobile IP: Design Principles and Practice*. Reading, MA: Addison-Wesley, 1997.
- PETERSEN, K., SPREITZER, M., TERRY, D., THEIMER, M., and DEMERS, A.** : "Flexible Update Propagation for Weakly Consistent Replication." *Proc. 16th Symp. Operating System Principles*. ACM, 1997. pp. 288-301.
- PFITZMANN, B. and WADNER, M.** : "Properties of Payment Systems: General Definition Sketch and Classification." Technical Report RZ 2823, IBM Research Division, Zurich Research Laboratory, June 1996.
- PFLEEEGER, C.** : *Security in Computing*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 1997.
- PIERRE, G., VAN STEEN, M., and TANENBAUM, A.** : "Self-Replicating Web Documents." Technical Report IR-486, Vrije Universiteit, Department of Mathematics and Computer Science, Feb. 2001.
- PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., and WINTERBOTTOM, P.** : "Plan 9 from Bell Labs." *Computing Systems*, vol. 8, no. 3, pp. 221-254, Summer 1995.
- PIKE, R.** : "8½, the Plan 9 Window System." *Proc. Summer Techn. Conf.* USENIX, 1991. pp. 257-265.
- PITOURA, E. and SAMARAS, G.** : "Locating Objects in Mobile Computing." *IEEE Trans. Know. Data Eng.*, vol. 13, 2001.
- PLAINFOSSE, D. and SHAPIRO, M.** : "A Survey of Distributed Garbage Collection Techniques." In *Proc. Int'l Workshop on Memory Management*, vol. 986 of *Lect. Notes Comp. Sc.*, pp. 211-249. Berlin: Springer-Verlag, Sept. 1995.
- PLATT, D.** : *The Essence of COM and ActiveX: A Programmers Workbook*. Englewood Cliffs, NJ: Prentice Hall, 1998.

- PLUMMER, D.** : "Ethernet Address Resolution Protocol." RFC 826, Nov. 1982.
- POPE, A.** : *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- POSTEL, J.** : "Simple Mail Transfer Protocol." RFC 821, Aug. 1982.
- POSTEL, J. and REYNOLDS, J.** : "File Transfer Protocol." RFC 995, Oct. 1985.
- PROTIC, J., TOMASEVIC, M., and MILUTINOVIC, V.** : "Distributed Shared Memory: Concepts and Systems." *IEEE Par. Distr. Techn.*, vol. 4, no. 2, pp. 63-79, Summer 1996.
- PROTIC, J., TOMASEVIC, M., and MILUTINOVIC, V.** : *Distributed Shared Memory, Concepts and Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1998.
- QIU, L., PADMANABHAN, V., and VOELKER, G.** : "On the Placement of Web Server Replicas." *Proc. 20th INFOCOM*. IEEE, 2001.
- RABINOVICH, M., RABINOVICH, I., RAJARAMAN, R., and AGGARWAL, A.** : "A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service." *Proc. 19th Int'l Conf. on Distributed Computing Systems*. ACM, 1999. pp. 101-113.
- RABINOVICH, M. and AGGARWAL, A.** : "Radar: A Scalable Architecture for a Global Web Hosting Service." *Proc. Eighth Int'l WWW Conf.*, 1999.
- RADIA, S.** : *Names, Contexts, and Closure Mechanisms in Distributed Computing Environments*. Ph. D. Thesis, University of Waterloo, Ontario, 1989.
- RADICATI, S.** : *X. 500 Directory Services: Technology and Deployment*. London: International Thomson Computer Press, 1994.
- RAMANATHAN, P., KANDLUR, D., and SHIN, K.** : "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems." *IEEE Trans. Comp.*, vol. C-89, no. 4, pp. 514-524, Apr. 1989.
- RAMANATHAN, P., SHIN, K., and BUTLER, R.** : "Fault-Tolerant Clock Synchronization in Distributed Systems." *IEEE Computer*, vol. 23, no. 10, pp. 33-42, Oct. 1990.
- RAO, H. and PETERSON, L.** : "Accessing Files in an Internet: The Jade File System." *IEEE Trans. Softw. Eng.*, vol. 19, no. 6, pp. 613-624, June 1993.
- RAYNAL, M.** : *Distributed Algorithms and Protocols*. New York: John Wiley, 1988.
- RAYNAL, M.** : "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms." *Oper. Syst. Rev.*, vol. 25, pp. 47-50, Apr. 1991.
- RAYNAL, M. and SINGHAL, M.** : "Logical Time: Capturing Causality in Distributed Systems." *IEEE Computer*, vol. 29, no. 2, pp. 49-56, Feb. 1996.
- REES, J. and CLINGER, W.** : "Revised Report on the Algorithmic Language Scheme." *SIGPLAN Notices*, vol. 21, no. 12, pp. 37-43, Dec. 1986.
- REITER, M.** : "How to Securely Replicate Services." *ACM Trans. Prog. Lang. Syst.*,

vol. 16, no. 3, pp. 986-1009, May 1994.

REITER, M., BIRMAN, K., and VAN RENESSE, R. : "A Security Architecture for Fault-Tolerant Systems." *ACM Trans. Comp. Syst.*, vol. 12, no. 4, pp. 340-371, Nov. 1994.

RESCORLA, E. and SCHIFFMAN, A. : "The Secure HyperText Transfer Protocol." RFC 2660, Aug. 1999.

REYNOLDS, J. and POSTEL, J. : "Assigned Numbers." RFC 1700, Oct. 1994.

RICART, G. and AGRAWALA, A. : "An Optimal Algorithm for Mutual Exclusion in Computer Networks." *Commun. ACM*, vol. 24, no. 1, pp. 9-17, Jan. 1981.

RIVEST, R. : "The MD5 Message Digest Algorithm." RFC 1321, Apr. 1992.

RIVEST, R., SHAMIR, A., and ADLEMAN, L. : "A Method for Obtaining Digital Signatures and Public-key Cryptosystems." *Commun. ACM*, vol. 21, no. 2, pp. 120-126, Feb. 1978.

RIZZO, L. : "Effective Erasure Codes for Reliable Computer Communication Protocols." *ACM Comp. Commun. Rev.*, vol. 27, no. 2, pp. 24-36, Apr. 1997.

RODRIGUES, L., FONSECA, H., and VERRISSIMO, P. : "Totally Ordered Multicast in Large-Scale Systems." *Proc. 16th Int'l Conf. on Distributed Computing Systems*. IEEE, 1996. pp. 503-510.

RODRIGUEZ, P. and SIBAL, S. : "SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution." *Comp. Netw. & ISDN Syst.*, vol. 33, no. 1-6, pp. 33-46, 2000.

ROGERSON, D. : *Inside COM*. Redmond, WA: Microsoft Press, 1997.

ROSENBLUM, M. and OOSTERHOUT, J. : "The Design and Implementation of a Log-Structured File System." *ACM Trans. Comp. Syst.*, vol. 10, no. 1, pp. 26-52, Feb. 1992.

ROWSTRON, A. : "Run-time Systems for Coordination." In Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R. (eds.), *Coordination of Internet Agents: Models, Technologies and Applications*. pp. 78-96. Berlin: Springer-Verlag, 2001.

ROWSTRON, A. and WRAY, S. : "A Run-Time System for WCL." In Bal, H., Belkhouche, B., and Cardelli, L. (eds.), *Internet Programming Languages*. vol. 1686 of *Lect. Notes Comp. Sc.*, pp. 78-96. Berlin: Springer-Verlag, 1998.

SAITO, Y. : "Consistency Management in Optimistic Replication Algorithms." Technical Report, University of Washington, June 2001.

SALTZER, J. and SCHROEDER, M. : "The Protection of Information in Computer Systems." *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278-1308, Sept. 1975.

SALTZER, J. : "Naming and Binding Objects." In Bayer, R., Graham, R., and

- Seegmuller, G. (eds.), *Operating Systems: An Advanced Course*. vol. 60 of *Lect. Notes Comp. Sc.*, pp. 99-208. Berlin: Springer-Verlag, 1978.
- SALTZER, J., REED, D., and CLARK, D.**: "End-to-End Arguments in System Design." *ACM Trans. Comp. Syst.*, vol. 2, no. 4, pp. 277-288, Nov. 1984.
- SAMAR, V. and LAI, C.**: "Making Login Services Independent from Authentication Technologies" *Proc. SunSoft Developer's Conf.*, 1996.
- SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., and LYON, B.**: "Design and Implementation of the Sun Network File System." *Proc. Summer Techn. Conf. USENIX*, 1985. pp. 119-130.
- SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., and YOUNMAN, C. E.**: "Role-Based Access Control Models." *IEEE Computer*, vol. 29, no. 2, pp. 38-47, Feb. 1996.
- SATYANARAYANAN, M.**: "Scalable, Secure, and Highly Available Distributed File Access." *IEEE Computer*, vol. 23, no. 5, pp. 9-21, May 1990.
- SATYANARAYANAN, M.**: "The Influence of Scale on Distributed File System Design." *IEEE Trans. Softw. Eng.*, vol. 18, no. 1, pp. 1-8, Jan. 1992.
- SATYANARAYANAN, M., KISTLER, J., KUMAR, P., OKASKAI, M., SIEGEL, E., and STEERE, D.**: "Coda: A Highly Available File System for a Distributed Workstation Environment." *IEEE Trans. Comp.*, vol. 39, no. 4, pp. 447-459, Apr. 1990.
- SATYANARAYANAN, M., MASHBURN, H., KUMAR, P., STEERE, D., and KISTLER, J.**: "Lightweight Recoverable Virtual Memory." *ACM Trans. Comp. Syst.*, vol. 12, no. 1, pp. 33-57, Feb. 1994.
- SATYANARAYANAN, M. and SIEGEL, E.**: "Parallel Communication in a Large Distributed System." *IEEE Trans. Comp.*, vol. 39, no. 3, pp. 328-348, Mar. 1990.
- SCHMIDT, D., STAL, M., ROHNERT, H., and BUSCHMANN, F.**: *Pattern-Oriented Software Architecture-Patterns for Concurrent and Networked Objects*. New York: John Wiley, 2000.
- SCHNEIDER, F.**: "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299-320, Dec. 1990.
- SCHNEIER, B.**: *Applied Cryptography*. New York: John Wiley, 2nd ed., 1996.
- SCHNEIER, B.**: *Secrets and Lies*. New York: John Wiley, 2000.
- SCHULZRINNE, H., CASNER, S., FREDERICK, R., and JACOBSON, V.**: "RTP: A Transport Protocol for Real-Time Applications." RFC 1889, Jan. 1996.
- SET**: "Secure Electronic Transactions Specification. Book 3: Formal Protocol Definition." May 1997.

- SHAPIRO, M., DICKMAN, P., and PLAINFOSSE, D.** : "SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection." Technical Report 1799, INRIA, Rocquencourt, France, Nov. 1992.
- SHAPIRO, M., GOURHANT, Y., HABERT, S., MOSSERI, L., RUFFIN, M., and VALOT, C.** : "SOS: An Object-Oriented Operating System - Assessment and Perspectives." *Computing Systems*, vol. 2, no. 4, pp. 287-337, Fall 1989.
- SHASHA, D. and JEONG, K.** : "PLinda 2.0: A Transactional/checkpointing Approach to Fault Tolerant Linda." *Proc. 13th Symp. on Reliable Distributed Systems*. IEEE, 1994. pp. 96-105.
- SHAW, M. and CLEMENTS, P.** : "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems." *Proc. 21st Int'l Comp. Softw. & Appl. Conf.*, 1997. pp. 6-13.
- SHEPLER, S.** : "NFS Version 4 Design Considerations." RFC 2624, June 1999.
- SHEPLER, S., BEAME, C., CALLAGHAN, B., EISLER, M., NOVECK, D., ROBINSON, D., and THURLOW, R.** : "NFS version 4 Protocol." RFC 3010, Dec. 2000.
- SHERESH, B. and SHERESH, D.** : *Understanding Directory Services*. Indianapolis, IN: New Riders, 2000.
- SHERIF, M. H. and SECHROUCHNI, A.** : *Protocols for Secure Electronic Commerce*. Boca Raton: CRC Press, 2000.
- SHETH, A. P. and LARSON, J. A.** : "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Comput. Surv.*, vol. 22, no. 3, pp. 183-236, Sept. 1990.
- SIEGEL, J.** : "OMG Overview: CORBA and the OMA in Enterprise Computing." *Commun. ACM*, vol. 41, no. 10, pp. 37-43, Oct. 1998.
- SILBERSCHATZ, A., GALVIN, P., and GAGNE, G.** : *Applied Operating System Concepts*. New York: John Wiley, 2000.
- SINGH, S. and KUROSE, J.** : "Electing 'Good' Leaders." *J. Par. Distr. Comput.*, vol. 21, pp. 184-201, May 1994.
- SINGHAL, M.** : "A Taxonomy of Distributed Mutual Exclusion." *J. Par. Distr. Comput.*, vol. 18, no. 1, pp. 94-101, may 1993.
- SINGHAL, M. and SHIVARATRI, N.** : *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. New York: McGraw-Hill, 1994.
- SKEEN, D.** : "Nonblocking Commit Protocols." *Proc. SIGMOD Int'l Conf. on Management Of Data*. ACM, 1981. pp. 133-142.
- SKEEN, D.** : "An Information Bus Architecture for Large-Scale Decision Support Environments." *Proc. Winter Techn. Conf.* USENIX, 1992. pp. 183-195.
- SKEEN, D. and STONEBRAKER, M.** : "A Formal Model of Crash Recovery in a

- Distributed System." *IEEE Trans. Softw. Eng.*, vol. SE-9, no. 3, pp. 219-228, Mar. 1983.
- SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., and DONGARRA, J.:** *MPI: The Complete Reference - The MPI Core.* Cambridge, MA: MIT Press, 1998.
- SPASOJEVIC, M. and SATYANARAYANAN, M.:** "An Empirical Study of a Wide-Area Distributed File System." *ACM Trans. Comp. Syst.*, vol. 14, no. 2, pp. 200-222, May 1996.
- SPEAKMAN, T., BHASKAR, N., CROWCROFT, J., EDMONSTONE, R., FARINACCI, D., GEMMELL, J., MONTGOMERY, T., LESCHCHINER, D., LIN, S., LUBY, M., RIZZO, L., SUMANASEKERA, R., VICISANO, L., and TWEEDELY, A.:** "PGM Reliable Transport Protocol Specification." Internet Draft(work in progress), Feb. 2001.
- SPECTOR, A.:** "Performing Remote Operations Efficiently on a Local Computer Network." *Commun. ACM*, vol. 25, no. 4, pp. 246-260, Apr. 1982.
- SRIKANTH, T. and TOUEG, S.:** "Optimal Clock Synchronization." *J. ACM*, vol. 34, no. 3, pp. 626-645, July 1987.
- SRINIVASAN, R.:** "RPC: Remote Procedure Call Protocol Specification Version 2." RFC 1831, Aug. 1995a.
- SRINIVASAN, R.:** "XDR: External Data Representation Standard." RFC 1832, Aug. 1995b.
- STEIN, L.:** *Web Security, A Step-by-Step Reference Guide.* Reading, MA: Addison-Wesley, 1998.
- STEINER, J., NEUMAN, C., and SCHILLER, J.:** "Kerberos: An Authentication Service for Open Network Systems." *Proc. Winter Techn. Conf. USENIX*, 1988. pp. 191-202.
- STEINMETZ, R. and NAHRSTEDT, K.:** *Multimedia: Computing, Communications and Applications.* Upper Saddle River, N. J. : Prentice Hall, 1995.
- STEINMETZ, R.:** "Human Perception of Jitter and Media Synchronization." *IEEE J. Selected Areas Commun.*, vol. 14, no. 1, pp. 61-72, Jan. 1996.
- STEVENS, W.:** *Advanced Programming in the UNIX Environment.* Reading, MA: Addison-Wesley, 1992.
- STEVENS, W.:** *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols.* Reading, MA: Addison-Wesley, 1996.
- STEVENS, W.:** *UNIX Network Programming-Networking APIs: Sockets and XTI.* Englewood Cliffs, NJ: Prentice Hall, 2nd ed. , 1998.
- STEVENS, W.:** *UNIX Network Programming-Interprocess Communication.* Englewood Cliffs, NJ: Prentice Hall, 2nd ed. , 1999.

- STUMM, M. and ZHOU, S.** : "Algorithms Implementing Distributed Shared Memory." *Computer*, vol. 23, no. 5, pp. 54-64, 1990.
- SUN MICROSYSTEMS:** *Java Remote Method Invocation Specification, JDK 1.2*. Sun Microsystems, Mountain View, Calif., Oct. 1998.
- SUN MICROSYSTEMS:** *JavaSpaces Service Specification, Version 1.1*. Sun Microsystems, Palo Alto, CA, Oct. 2000a.
- SUN MICROSYSTEMS:** *Jini Architecture Specification, Version 1.1*. Palo Alto, CA, Oct. 2000b.
- SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., and PECK, G.** : "Scalability in the XFS File System." *Proc. Ann. Techn. Conf. USENIX*, 1996. pp. 1-14.
- TAI, S. and ROUVELLOU, I.** : "Strategies for Integrating Messaging and Distributed Object Transactions." In *Proc. Middleware 2000*, vol. 1795 of *Lect. Notes Comp. Sc.*, pp. 308-330. Berlin: Springer-Verlag, 2000.
- TANENBAUM, A.** : *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall, 3rd ed., 1996.
- TANENBAUM, A.** : *Modern Operating Systems*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 2001.
- TANENBAUM, A., MULLENDER, S., and VAN RENESSE, R.** : "Using Sparse Capabilities in a Distributed Operating System." *Proc. Sixth Int'l Conf. on Distributed Computing Systems*. IEEE, 1986. pp. 558-563.
- TANENBAUM, A., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G., MULLENDER, S., JANSEN, J., and VAN ROSSUM, G.** : "Experiences with the Amoeba Distributed Operating System." *Commun. ACM*, vol. 33, no. 12, pp. 46-63, Dec. 1990.
- TANENBAUM, A. and WOODHULL, A.** : *Operating Systems, Design and Implementation*. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1997.
- TANISCH, P.** : "Atomic Commit in Concurrent Computing." *IEEE Concurrency*, vol. 8, no. 4, pp. 34-41, Oct. 2000.
- TARTALJA, I. and MILUTINOVIC, V.** : "Classifying Software-Based Cache Coherence Solutions." *IEEE Softw.*, vol. 14, no. 3, pp. 90-101, May 1997.
- TERRY, D., DEMERS, A., PETERSEN, K., SPREITZER, M., THEIMER, M., and WELSH, B.** : "Session Guarantees for Weakly Consistent Replicated Data." *Proc. Third Int'l Conf. on Parallel and Distributed Information Systems*. IEEE, 1994. pp. 140-149.
- TERRY, D., PETERSEN, K., SPREITZER, M., and THEIMER, M.** : "The Case for Non-transparent Replication: Examples from Bayou." *IEEE Data Engineering*, vol. 21, no. 4, pp. 12-20, Dec. 1998.

- TEWARI, R. , DAHLIN, M. , VIN, H. , and KAY, J.** : "Design Considerations for Distributed Caching on the Internet." *Proc. 19th Int'l Conf. on Distributed Computing Systems*. IEEE, 1999. pp. 273-284.
- THEKKATH, C. ,MANN, T. , and LEE, E.** : "Frangipani: A Scalable Distributed File System." *Proc. 16th Symp. Operating System Principles*. ACM, 1997. pp. 224-237.
- THOMAS, R.** : "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases." *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180-209, June 1979.
- TIBCO:** *TIB/Rendezvous TX Concepts, Release 1. 1.* TIBCO Software Inc. , Palo Alto, CA, Nov. 2000.
- TIBCO:** *TIB/Rendezvous C Reference, Release 6. 4.* TIBCO Software Inc. , Palo Alto, CA, Oct. 2000a.
- TIBCO:** *TIB/Rendezvous Concepts, Release 6. 4.* TIBCO Software Inc. , Palo Alto, CA, Oct. 2000b.
- TOLKSDORF, R.** : "A Machine for Uncoupled Coordination and its Concurrent Behaviour." In *Object-Based Models and Languages for Concurrent Systems*, vol. 924 of *Lect. Notes Comp. Sc.*, pp. 176-193. Berlin: Springer-Verlag, 1995.
- TOWSLEY,D. ,KUROSE,J. , and PINGALI, S.** : "A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols." *IEEE J. Selected Areas Commun.*, vol. 15, no. 3, pp. 398-407, Apr. 1997.
- TRIANTAFILLOU, P. and NEILSON, C.** : "Achieving Strong Consistency in a Distributed File System." *IEEE Trans. Softw. Eng.*, vol. 23, no. 1, pp. 35-55, Jan. 1997.
- TRIPATHI,A. ,KARNIK,N. ,VORA,M. ,AHMED,T. , and SINGH,R.** : "Mobile Agent Programming in Ajanta." *Proc. 19th Int'l Conf. on Distributed Computing Systems*. IEEE, 1999. pp. 190-197.
- TUREK,J. and SHASHA, S.** : "The Many Faces of Consensus in Distributed Systems." *IEEE Computer*, vol. 25, no. 6, pp. 8-17, June 1992.
- ULLMAN, J.** : *Principles of Database and Knowledge-Base Systems*, vol. 1. Rockville, MA: Computer Science Press, 1988.
- UMAR,A. :** *Object-Oriented Client/Server Internet Environments*. Upper Saddle River, NJ: Prentice Hall, 1997.
- VAHA-SIPILA,A. :** "URLs for Telephone Calls." RFC 2806, Apr. 2000.
- VAN STEEN,M. ,HAUCK,F. ,BALLINTIJN,G. , and TANENBAUM, A.** : "Algorithmic Design of the Globe Wide-Area Location Service." *Comp. J.*, vol. 41, no. 5, pp. 297-310, 1998a.
- VAN STEEN, M. , HAUCK, F. , HOMBURG, P. , and TANENBAUM, A.** : "Locating

- Objects in Wide-Area Systems." *IEEE Commun. Mag.*, vol. 36, no. 1, pp. 104-109, Jan. 1998b.
- VAN STEEN, M., HOMBURG, P., and TANENBAUM, A.** : "Globe: A Wide-Area Distributed System." *IEEE Concurrency*, vol. 7, no. 1, pp. 70-78, Jan. 1999a.
- VAN STEEN, M., TANENBAUM, A., KUZ, I., and SIPS, H.** : "A Scalable Middleware Solution for Advanced Wide-Area Web Services." *Distributed Systems Engineering*, vol. 6, no. 1, pp. 34-42, Mar. 1999b.
- VERISSIMO, P. and RODRIGUES, L.** : *Distributed Systems for Systems Architects*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2001.
- VETTER, R., SPELL, C., and WARD, C.** : "Mosaic and the World-Wide Web." *IEEE Computer*, vol. 27, no. 10, pp. 49-57, Oct. 1994.
- VINOSKI, S.** : "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments." *IEEE Commun. Mag.*, vol. 35, no. 2, pp. 46-55, Feb. 1997.
- VIVENEY, B.** : "DCE and Object Programming." In Rosenberry, W. (ed.), *DCE Today*, pp. 251-264. Upper Saddle River, NJ: Prentice Hall, 1998.
- VIXIE, P.** : "A DNS RR for Specifying the Location of Services(DNS SRV)." RFC 2052, Oct. 1996.
- VON EICKEN, T., CULLER, D., GOLDSTEIN, S., and SCHAUER, K.** : "Active Messages: a Mechanism for Integrated Communication and Computation." *Proc. 19th Int'l Symp. on Computer Architecture*, 1992. pp. 256-266.
- VOYDOCK, V. and KENT, S.** : "Security Mechanisms in High-Level Network Protocols." *ACM Comput. Surv.*, vol. 15, no. 2, pp. 135-171, June 1983.
- WAHBE, R., LUCCO, S., ANDERSON, T., and GRAHAM, S.** : "Efficient Software-based Fault Isolation." *Proc. 14th Symp. Operating System Principles*. ACM, 1993. pp. 203-216.
- WAHL, M., HOWES, T., and KILLE, S.** : "Lightweight Directory Access Protocol (v3)." RFC 2251, Dec. 1997.
- WALDO, J.** : "Remote Procedure Calls and Java Remote Method Invocation." *IEEE Concurrency*, vol. 6, no. 3, pp. 5-7, July 1998.
- WALDO, J.** : *The Jini Specifications*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 2000.
- WALLACH, D., BALFANZ, D., DEAN, D., and FELTEN, E.** : "Extensible Security Architectures for Java." *Proc. 16th Symp. Operating System Principles*. ACM, 1997. pp. 116-128.
- WANG, H., LO, M. K., and WANG, C.** : "Consumer Privacy Concerns about Internet Marketing." *Commun. ACM*, vol. 41, no. 3, pp. 63-70, Mar. 1998.

- WANG, J.** : "A Survey of Web Caching Schemes for the Internet." *ACM Comp. Commun. Rev.*, vol. 29, no. 5, pp. 36-46, Oct. 1999.
- WANG, J.** : "A Fully Distributed Location Registration Strategy for Universal Personal Communication Systems." *IEEE J. Selected Areas Commun.*, vol. 11, no. 6, pp. 850-860, Aug. 1993.
- WANG, R., ANDERSON, T., and DAHLIN, M.** : "Experiences with a Distributed File System Implementation." Technical Report CSD-98-986, Department of Computer Science, University of California, Berkeley, 1998.
- WHITEHEAD, J. and GOLAND, Y.** : "WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web." *Proc. Sixth European Conf. on Computer Supported Cooperative Work*, 1999. pp. 291-310.
- WHITEHEAD, J. and WIGGINS, M.** : "WEBDAV: IETF Standard for Collaborative Authoring on the Web." *IEEE Internet Comput.*, vol. 2, no. 5, pp. 34-40, Sept. 1998.
- WIERINGA, R. and DE JONGE, W.** : "Object Identifiers, Keys, and Surrogates-Object Identifiers Revisited." *Theory and Practice of Object Systems*, vol. 1, no. 2, pp. 101-114, 1995.
- WIESMANN, M., PEDONE, F., SCHIPER, A., KEMME, B., and ALONSO, G.** : "Understanding Replication in Databases and Distributed Systems." *Proc. 20th Int'l Conf. on Distributed Computing Systems*. IEEE, 2000. pp. 264-274.
- WILSON, P.** : "Uniprocessor Garbage Collection Techniques." Technical Report, University of Texas at Austin, 1994.
- WOLLRATH, A., RIGGS, R., and WALDO, J.** : "A Distributed Object Model for the Java System." *Computing Systems*, vol. 9, no. 4, pp. 265-290, Fall 1996.
- WOLMAN, A., VOELKER, G., SHARMA, N., CARDWELL, N., KARLIN, A., and LEVY, H.** : "On the Scale and Performance of Cooperative Web Proxy Caching." *Proc. 17th Symp. Operating System Principles*. ACM, 1999. pp. 16-31.
- WOOLDRIDGE, M.** : "Agent-Based Computing." *Interoperable Communication Networks*, vol. 1, no. 1, pp. 71-97, Jan. 1998.
- WU, J.** : *Distributed System Design*. Boca Raton: CRC Press, 1998.
- WYCKOFF, P., MCLAUGHRY, S., LEHMAN, T., and FORD, D.** : "T Spaces." *IBM Systems J.*, vol. pp. 454-474, Aug. 1998.
- YANG, C. and LUO, M.** : "A Content Placement and Management System for Distributed Web-Server Systems." *Proc. 20th Int'l Conf. on Distributed Computing Systems*. IEEE, 2000.
- YEADON, N., GARCIA, F., HUTCHISON, D., and SHEPHARD, S.** : "Filters: QoS Support Mechanisms for Multipeer Communications." *IEEE J. Selected Areas Commun.*, vol. 14, no. 7, pp. 1245-1262, Sept. 1996.

- YEONG, W., HOWES, T., and KILLE, S.** : "Lightweight Directory Access Protocol." RFC 1777, Mar. 1995.
- YU, H. and VAHDAT, A.** : "Design and Evaluation of a Continuous Consistency Model for Replicated Services." *Proc. Fourth Symp. on Operating System Design and Implementation*. USENIX, 2000.
- ZHANG, L., DEERING, S., ESTRIN, D., SHENKER, S., and ZAPPALA, D.** : "RSVP: A New Resource Reservation Protocol." *IEEE Network Magazine*, vol. 7, no. 5, pp. 8-18, Sept. 1993.
- ZWICKY, E., COOPER, S., CHAPMAN, D., and RUSSELL, D.** : *Building Internet Firewalls*, Sebastopol, CA: O'Reilly & Associates, 2nd ed., 2000.