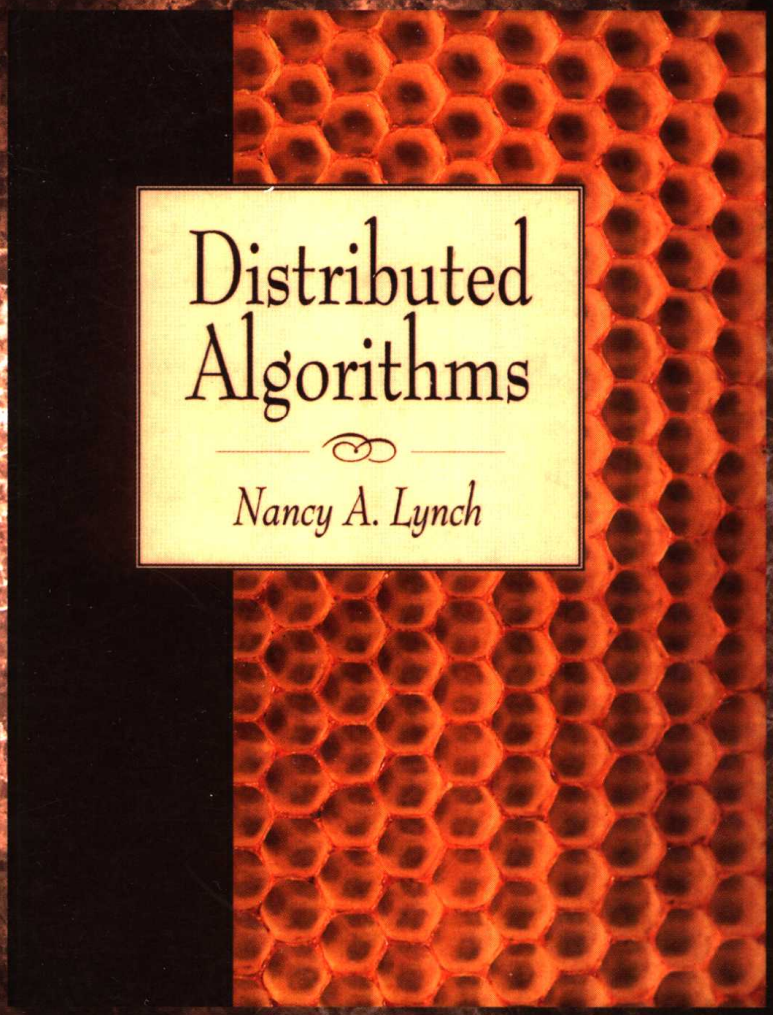
计 算 机 科 学 丛 书

分布式算法

（美） Nancy A. Lynch 著　　舒继武　李国东　余华山　译



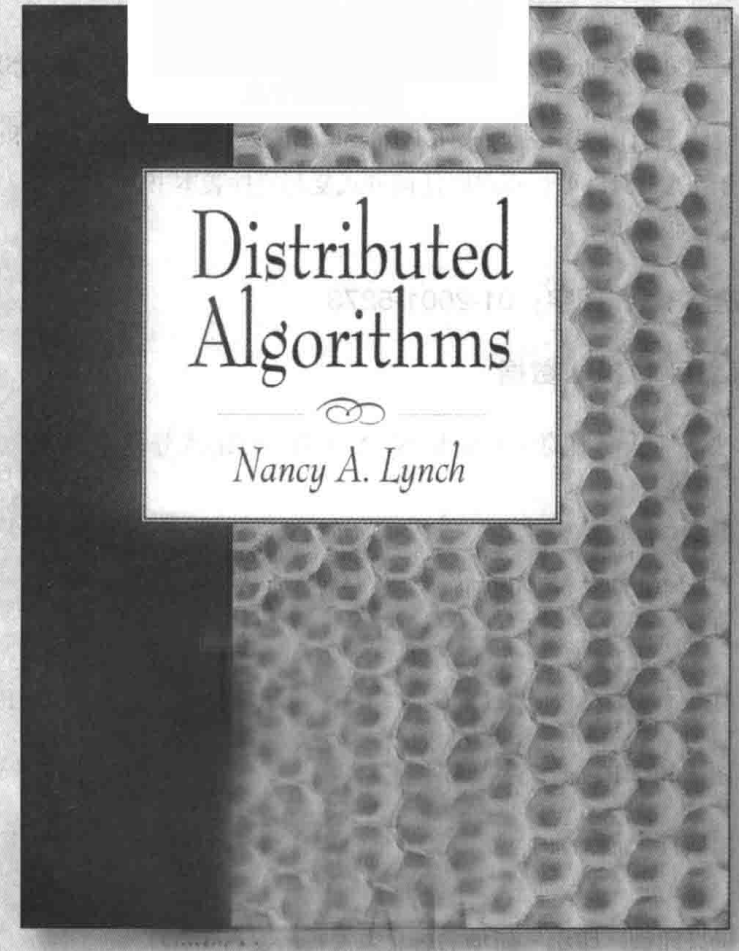
Distributed Algorithms

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

计 算 机 科 学 丛 书

分布式算法

(美) Nancy A. Lynch 著　舒继武 李国东 余华山 译



Distributed Algorithms

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

本书对分布式算法进行全面介绍，包括最为重要的算法和不可能性结果。绝大部分的解都给出了数学证明。这些算法都根据精确定义的复杂度衡量方法进行分析。本书还讲述针对许多典型问题的算法、各类系统模型及其能力。章后提供大量习题并列出了详细的参考文献。

本书可作为高等院校计算机系研究生的教材，尤其适合对计算机理论或体系结构感兴趣的学生学习，还适合分布式设计人员、研究人员及其相关技术人员参考。

Nancy A. Lynch: Distributed Algorithms (ISBN 1-55860-348-4).

Original English language edition copyright © 1996 by Morgan Kaufmann Publishers, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher.

Chinese simplified language edition published by China Machine Press and CITIC PUBLISHING HOUSE.

Copyright © 2003 by China Machine Press and CITIC PUBLISHING HOUSE.

本书中文简体字版由美国Morgan Kaufmann公司授权机械工业出版社和中信出版社合作出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书版权登记号：图字：01-2001-5273

图书在版编目（CIP）数据

分布式算法 /（美）林奇（Lynch, N. A.）著；舒继武等译. – 北京：机械工业出版社，2004.1

（计算机科学丛书）

书名原文：Distributed Algorithms

ISBN 7-111-13127-4

Ⅰ. 分… Ⅱ. ① 林… ② 舒… Ⅲ. 电子计算机 – 算法理论 Ⅳ. TP301.6

中国版本图书馆CIP数据核字（2003）第087861号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：姚 蕾

北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行

2004年 1月第1版第1次印刷

787mm × 1092mm 1/16 · 34印张

印数：0 001- 4 000册

定价：59.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线电话：（010）68326294

出版者的话

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

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

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

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

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

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

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

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

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

专家指导委员会

（按姓氏笔画顺序）

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

译 者 序

分布式计算是随着计算机网络的发展而兴起的，现已成为提高问题求解规模和速度、提高系统可靠性的重要手段，在数值模拟和生物工程等应用领域中被广泛应用。随着网络技术的发展以及网络计算的兴起，分布式计算技术也在不断地发展和完善中，在计算机技术的发展和应用中发挥着越来越重要的作用。在我国科学工程计算部门和高等院校中，有越来越多的科技工作者开始学习和研究分布式计算技术。

分布式计算包括三个层面的内容：作为底层的分布式系统，作为理论指导的分布式算法，以及结合具体问题的程序实现。其中，分布式算法处于重要地位，它是分布式系统的体现，更是分布式程序设计的基础和灵魂。分布式算法的一个重要特点是，它并不仅是抽象的理论研究，而且是与具体的分布式系统和应用问题密切相关的。

然而，在分布式算法的研究中，国内的有关资料十分缺乏，因此我们翻译了本书，它有几个显著特点：

全面：本书分三部分，分别对同步算法、异步算法和部分同步算法进行全面的介绍，可以作为一本分布式算法的完全手册。

严谨：书中的算法和概念都给出准确的定义，性能的分析评价都给出严格的证明，可以作为进一步深入理论研究的基础。

深入浅出：虽然算法理论有很强的抽象性，但是本书能够用浅显的语言和大量的图示作出详尽的讲解，阅读本书只需要读者具备一些基本的离散数学和概率知识。因此，本书可以适合不同层次的读者。

本书的翻译是专门从事分布式算法的科研工作者通力合作的结果，其中清华大学计算机系舒继武副教授翻译了序言和第1章至第7章，南京大学计算机系李国东副教授翻译了第8章至第13章和第15章至第20章，北京大学计算机系余华山博士翻译了第14章和第21章至第25章，全书由舒继武和李国东统稿和审校。

此书的翻译过程中，我们深切体会到本书作者在分布式算法方面的造诣，自身也获得了提高。希望本书能使国内的学者共享本书作者的思想和结晶。

由于分布式算法是一个蓬勃发展的领域，译者水平有限，加上时间仓促，书中的错误在所难免，竭诚欢迎广大读者批评指正。

译 者

2003年9月

前　言

分布式算法是用于解决多个互连处理器运行问题的算法。分布式算法的各部分并发和独立地运行，每一部分只承载有限的信息。即使处理器和通信信道以不同的速度运作，或即使某些构件出了故障，这些算法仍然应该工作正常。

分布式算法有广泛的应用：电信、分布式信息处理、科学计算以及实时进程控制。例如，今天的电话系统、航班订票系统、银行系统、全球信息系统、天气预报系统以及飞机和核电站控制系统都严重依赖于分布式算法。很明显，确保分布式算法准确、高效地运行是非常重要的。然而，由于这种算法的执行环境很复杂，所以设计分布式算法就成为了一项极端困难的任务。

本书对分布式算法这个领域做了全面的介绍——包括最为重要的算法和不可能性结果，且都是在一种简单的自动机理论环境中呈现。几乎所有的解都给出了数学证明（至少是粗略的）。这些算法都根据精确定义的复杂度衡量方法进行了分析。总之，这些材料为更深入地理解分布式算法打下了牢固的基础。

本书面向不同层次的读者。首先，本书可以作为计算机系一年级研究生的教材，尤其适合于对计算机系统、理论或两者怀有浓厚兴趣的学生。第二，本书可作为分布式系统设计人员的短期培训教材。最后，它也可作为参考手册，供设计人员、学生、研究人员以及任何对该领域感兴趣的个人使用。

本书包含了针对很多典型问题的算法，如在几种不同系统环境下的一致性(consensus)、通信、资源分配和同步问题。这些算法和结论基于分布式环境的基本假设来组织。组织的第一层基于时序模型（timing model）——同步、异步或部分同步；第二层基于进程间的通信机制——共享存储器或消息传递。每种系统模型都用数章来阐述：每一组的头一章提出所述系统类型的形式化模型，余下各章介绍了算法和不可能性结果。从头至尾，我们进行了严密的论述，然而非常简明易懂。

由于该领域很广阔也很活跃，因此本书不想去包罗万象。我们只是将最根本的结果选进了本书。若以复杂度来衡量，这些结果并不总是最优的；但它们比较简单且能够阐明重要的通用设计和推理方法。

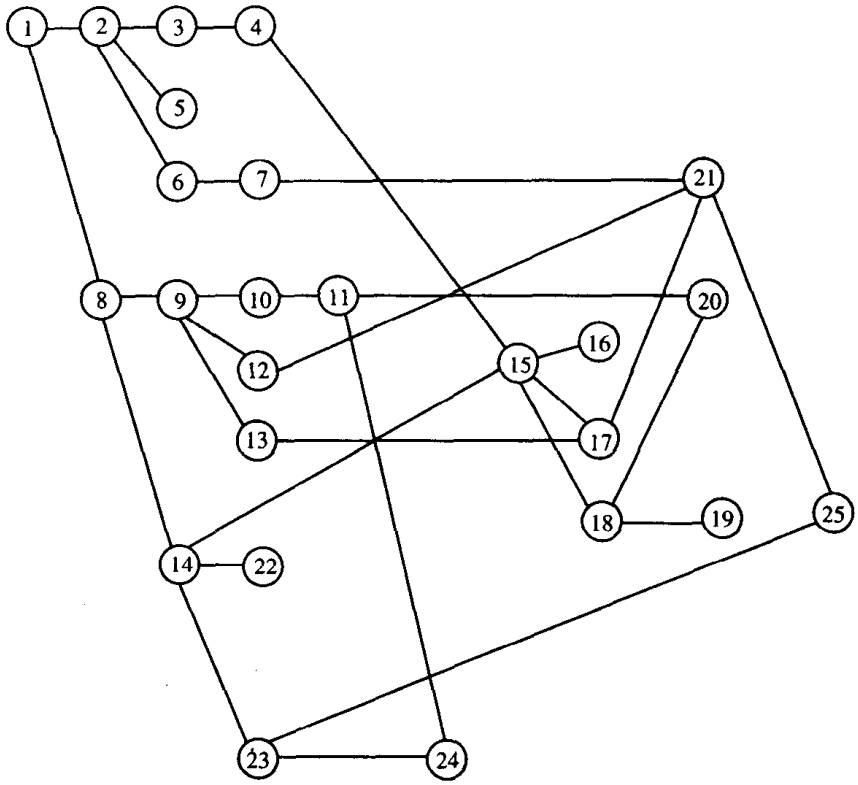
本书将会介绍分布式计算领域中许多最重要的问题、算法和不可能性结果。当实际系统中出现这些问题的时候，你就能将它们识别出来，并进而利用本书介绍的算法来解决它们，或者应用不可能性结果来证明它们是不可解的。本书还介绍各类系统模型及其能力。这样一来，你自己就可以设计出新算法（甚至还可以证明出新的不可能性结果）。最后，本书还会让你相信，严格推导分布式算法和系统是可行的：形式化建模，给出其所需行为的精确规格说明，严格证明它们符合规格说明，确定合适的复杂度衡量标准以及按照这些标准进行分析。

使用本书

预备知识　阅读本书所需的预备知识是基本的本科离散数学（包括数学归纳和渐进分析）、一些编程技能以及对计算机系统相当熟悉。有关随机算法的部分还需要基本的概率知识。有关串行算法及其分析的本科课程对阅读本书有帮助，但并不是必需的。

章节关系　本书的编排原则是使读者能比较独立地阅读不同模型的各章。各章之间的依赖

关系如图A所示。例如，如果想尽快了解异步网络，就可以跳过第5~7章。还可以只读算法部分，而不必先阅读算法所依赖的建模部分。



图A　各章之间的依赖关系

带星号的小节　在本书目录中，有几个小节的标题打了星号。它们的内容不太基本或者说比其他部分更深。第一次阅读的时候可以忽略这些内容，不会有什么影响。

课程　本书的第1版已经在MIT（麻省理工学院）的研究生导论课中用了很多年，并且在一些计算机软件和应用公司的系统设计师夏季课程中用了三年。本书包括足够一年课程的内容，所以对一些短期课程来说必须有所取舍（注意看章节之间的关系）。

例如，在强调异步网络计算的一个学期的课程中，可以选择第3、4、6章、7.2节、第12章、和第14~21章，参考一些有关建模的章节（第2、8和9章），并根据需要加入第10、11和13章中的一些定义。在强调对分布一致性进行详细研习的一个学期的课程中，可以选择第2~9、12章、13.1节、第15、17、21、23和25章。还有其他多种可能组合。如果你是这个领域的研究者，你可以用所在领域的研究报告中更新或者更特别的结论来补充本书。

在为系统设计师提供的一两周的短期课程中，可以涉及全部章节的重点，在较高的层次上讨论关键结论和关键证明思想，而无需讲解太多细节。

错误　如果在本书中发现了错误，以及对本书有什么建设性建议，请告诉我。特别欢迎对额外问题的建议。请发送email到：distalgs@theory.lcs.mit.edu。

致谢

我们很难一一列举出所有对本书的出版做出贡献的人们，因为本书是多年教学和研究的成果，得到了许多学生和研究人员的帮助。即使这样，我还是想尽力而为。

本书是MIT的研究生课程6.852（分布式算法）讲稿的最终版本。在我早期组织材料的过程中，学生们学过这门课。这些学生在1990和1992年给予了特别的帮助，当时是他们帮助我完成了讲稿的在线版本。有几位课程助教对我整理笔记给予了极大的帮助，他们是：Ken Goldman、Isaac Saias和Boaz Patt-Shamir。助教Jennifer Welch 和Rainer Gawlick也帮了我很大的忙。

许多同事和学生与我一起研究本书中的一些结果，或者与我一起讨论其他人的工作，这对我充分理解资料帮助很大。其中包括：Yehuda Afek、Eshrat Arjomandi、Hagit Attiya、Baruch Awerbuch、Bard Bloom、Alan Borodin、James Burns、Soma Chaudhuri、Brian Coan、Harish Devarajan、Danny Dolev、Cynthia Dwork、Alan Fekete、Michael Fischer、Greg Frederickson、Eli Gafni、Rainer Gawlick、Ken Goldman、Art Harvey、Maurice Herlihy、Paul Jackson、Jon Kleinberg、Leslie Lamport、Butler Lampson、Victor Luchangco、Yishay Mansour、Michael Merritt、Michael Paterson、Boaz Patt-Shamir、Gary Peterson、Shlomit Pinter、Stephen Ponzio、Isaac Saias、Russel Schaffer、Roberto Segala、Nir Shavit、Liuba Shrira、Jørgen Søgaard-Andersen、Eugene Stark、Larry Stockmeyer、Mark Tuttle、Frits Vaandrager、George Varghese、Bill Weihl、Jennifer Welch和Lenore Zuck。尤其感谢其中的两位：我的导师Michael 和我的学生Mark Tuttle。从1978年以来，Michael就开始与我一起致力于研究这个当时还很小但看起来很有前途的领域，而Mark Tuttle的硕士论文定义并发展了I/O自动机模型。

我还要感谢Ajoy Datta、Roberto De Prisco、Alan Fekete、Faith Fich、Rainer Gawlick、Shai Halevi、Jon Kleinberg、Richard Ladner、John Leo、Victor Luchangco、Michael Melliar-Smith、Michael Merritt、Daniele Micciancio、Boaz Patt-Shamir、Anya Pogosyants、Stephen Ponzio、Sergio Rajsbaum、Roberto Segala、Nir Shavit、Mark Smith、Larry Stockmeyer、Mark Tuttle、George Varghese、Jennifer Welch和Lenore Zuck，他们审阅了全书的各部分草稿并提出了很多有用的建议。特别是Ajoy、Faith和George，他们使用本书的早期版本作为教材来教学，给出了很多宝贵的意见。此外，我要感谢 Joanne Talbot 不厌其烦的排版、画图、搜集参考文献，以及不停地打印文稿等。David Jones也参予了排版工作。在此，我还要感谢 John Guttag、Paul Penfield 和其他MIT EECS的成员，他们为我安排了写书的时间。Morgan Kaufmann的Bruce Spatz又一次鼓励并帮助我做这个艰巨的工作，他总能给我正确的建议。在本书最后成型阶段，Morgan Kaufmann的Julie Pabst和Diane Cerra给了我很大帮助。同时，也感谢Babel Press的Ed Sznyter的LaTeX技术。

最后也是最重要的，我要感谢体贴我的家人Dennis、Patrick和Mary Lynch，他们体谅我为本书所做的一切工作。同时为我料理了其他的事情。特别感谢Dennis，当我把大部分时间花在电脑前时，Dennis却在为我准备美味的海鲜晚餐，甚至把我的浴室和洗衣房翻修一新！

Nancy A. Lynch
Cambridge, Massachusetts

目　录

出版者的话
专家指导委员会
译者序
前言
第1章　引言 ……1
1.1　相关主题……1
1.2　我们的观点……2
1.3　本书内容综述……3
1.4　参考文献注释……7
1.5　标记……7

第一部分　同步网络算法

第2章　建模I：同步网络模型 ……10
2.1　同步网络系统 ……10
2.2　故障 ……11
2.3　输入和输出 ……11
2.4　运行 ……11
2.5　证明方法 ……12
2.6　复杂度度量 ……12
2.7　随机化 ……12
2.8　参考文献注释 ……13
第3章　同步环中的领导者选择 ……14
3.1　问题 ……14
3.2　相同进程的不可能性结果 ……14
3.3　基本算法 ……15
3.4　通信复杂度为$O(n \log n)$的算法 ……17
3.5　非基于比较的算法 ……19
3.5.1　时间片算法 ……20
3.5.2　变速算法 ……20
3.6　基于比较的算法的下界 ……21
3.7　非基于比较的算法的下界* ……25
3.8　参考文献注释 ……26
3.9　习题 ……27
第4章　一般同步网络中的算法 ……29
4.1　一般网络中的领导者选举 ……29
4.1.1　问题 ……29
4.1.2　简单的洪泛算法 ……29
4.1.3　降低通信复杂度 ……31
4.2　广度优先搜索 ……32
4.2.1　问题 ……32
4.2.2　基本的广度优先搜索算法 ……33
4.2.3　应用 ……34
4.3　最短路径 ……35
4.4　最小生成树 ……36
4.4.1　问题 ……36
4.4.2　基本定理 ……36
4.4.3　算法 ……37
4.5　最大独立集 ……39
4.5.1　问题 ……40
4.5.2　随机化算法 ……40
4.5.3　分析* ……42
4.6　参考文献注释 ……43
4.7　习题 ……43
第5章　链路故障时的分布式一致性 ……46
5.1　协同攻击问题——确定性版本 ……46
5.2　协同攻击问题——随机化版本 ……48
5.2.1　形式化模型 ……49
5.2.2　算法 ……49
5.2.3　不一致的下限 ……52
5.3　参考文献注释 ……54
5.4　习题 ……54
第6章　进程故障下的分布式一致性 ……56
6.1　问题 ……56
6.2　针对停止故障的算法 ……58
6.2.1　基本算法 ……58
6.2.2　减少通信 ……59
6.2.3　指数信息收集算法 ……61
6.2.4　带鉴别的Byzantine一致性 ……66
6.3　针对Byzantine故障的算法 ……66
6.3.1　举例 ……66
6.3.2　Byzantine一致性问题的EIG算法 ……68
6.3.3　使用二元Byzantine一致的一般的Byzantine一致性问题 ……71
6.3.4　减少通信开销 ……72

6.4 Byzantine一致性问题中进程的个数 ……74
6.5 一般图中的Byzantine一致性问题 ……78
6.6 弱Byzantine一致性 ……81
6.7 有停止故障时的轮数 ……82
6.8 参考文献注释 ……88
6.9 习题 ……89
第7章 更多的一致性问题 ……93
7.1 k一致性问题 ……93
7.1.1 问题 ……93
7.1.2 算法 ……93
7.1.3 下界* ……95
7.2 近似一致性 ……102
7.3 提交问题 ……105
7.3.1 问题 ……105
7.3.2 两阶段提交 ……106
7.3.3 三阶段提交 ……107
7.3.4 消息数的下界 ……109
7.4 参考文献注释 ……111
7.5 习题 ……111

第二部分 异步算法

第8章 建模II：异步系统模型 ……114
8.1 输入/输出自动机 ……114
8.2 自动机的操作 ……118
8.2.1 合成 ……118
8.2.2 隐藏 ……121
8.3 公平性 ……121
8.4 问题的输入和输出 ……123
8.5 属性与证明方法 ……124
8.5.1 不变式断言 ……124
8.5.2 轨迹属性 ……124
8.5.3 安全与活性属性 ……125
8.5.4 合成推理 ……126
8.5.5 层次化证明 ……128
8.6 复杂度衡量 ……130
8.7 不可区分运行 ……131
8.8 随机化 ……131
8.9 参考文献注释 ……131
8.10 习题 ……132

第二部分A 异步共享存储器算法

第9章 建模III：异步共享存储器模型 …136
9.1 共享存储器系统 ……136
9.2 环境模型 ……138
9.3 不可区分状态 ……140
9.4 共享变量类型 ……140
9.5 复杂度衡量 ……144
9.6 故障 ……144
9.7 随机化 ……145
9.8 参考文献注释 ……145
9.9 习题 ……145
第10章 互斥 ……146
10.1 异步共享存储器模型 ……146
10.2 问题 ……148
10.3 Dijkstra的互斥算法 ……151
10.3.1 算法 ……151
10.3.2 正确性证明 ……154
10.3.3 互斥条件的一个断言证明 ……156
10.3.4 运行时间 ……157
10.4 互斥算法的更强条件 ……158
10.5 锁定权互斥算法 ……159
10.5.1 双进程算法 ……159
10.5.2 n进程算法 ……163
10.5.3 锦标赛算法 ……167
10.6 使用单写者共享存储器的算法 ……170
10.7 *Bakery*算法 ……171
10.8 寄存器数量的下界 ……173
10.8.1 基本事实 ……174
10.8.2 单写者共享变量 ……175
10.8.3 多写者共享变量 ……175
10.9 使用读-改-写共享变量的互斥 ……179
10.9.1 基本问题 ……179
10.9.2 有界绕过次数 ……180
10.9.3 锁定权 ……185
10.9.4 模拟证明 ……187
10.10 参考文献注释 ……189
10.11 习题 ……190
第11章 资源分配 ……194
11.1 问题 ……194
11.1.1 显式资源说明和互斥说明 ……194

11.1.2 资源分配问题 ……195
11.1.3 哲学家用餐问题 ……196
11.1.4 解法的受限形式 ……197
11.2 对称哲学家用餐算法的不存在性 ……197
11.3 右-左哲学家用餐算法 ……199
11.3.1 等待链 ……199
11.3.2 基本算法 ……200
11.3.3 扩展 ……202
11.4 哲学家用餐随机算法* ……204
11.4.1 算法* ……205
11.4.2 正确性* ……207
11.5 参考文献注释 ……212
11.6 习题 ……213
第12章 一致性 ……215
12.1 问题 ……215
12.2 使用读/写共享存储器的一致性 ……217
12.2.1 限制 ……218
12.2.2 术语 ……218
12.2.3 双价初始化 ……218
12.2.4 无等待终止的不可能性 ……219
12.2.5 单故障终止的不可能性结果 ……221
12.3 读/改/写共享存储器上的一致性问题 ……223
12.4 其他共享存储器类型 ……224
12.5 异步共享存储器系统中的可计算性* ……224
12.6 参考文献注释 ……225
12.7 习题 ……226
第13章 原子对象 ……229
13.1 定义和基本结论 ……229
13.1.1 原子对象的定义 ……229
13.1.2 规范无等待原子对象自动机 ……235
13.1.3 原子对象的合成 ……237
13.1.4 原子对象和共享变量 ……237
13.1.5 显示原子性的一个充分条件 ……241
13.2 用读/写变量实现读-改-写原子对象 ……242
13.3 共享存储器的原子快照 ……243
13.3.1 问题 ……243
13.3.2 带无界变量的一个算法 ……244
13.3.3 带有界变量的一个算法* ……247
13.4 读/写原子对象 ……250
13.4.1 问题 ……250
13.4.2 证明原子性的其他引理 ……250
13.4.3 带无界变量的一个算法 ……251
13.4.4 两个写者的有界算法 ……254
13.4.5 使用快照的算法 ……258
13.5 参考文献注释 ……259
13.6 习题 ……260

第二部分B 异步网络算法

第14章 建模IV：异步网络模型 ……264
14.1 发送/接收系统 ……264
14.1.1 进程 ……264
14.1.2 发送/接收通道 ……264
14.1.3 异步发送/接收系统 ……268
14.1.4 使用可靠FIFO通道的发送/接收系统的特征 ……268
14.1.5 复杂度度量 ……269
14.2 广播系统 ……269
14.2.1 进程 ……269
14.2.2 广播通道 ……270
14.2.3 异步广播系统 ……270
14.2.4 采用可靠广播通道的广播系统的特征 ……270
14.2.5 复杂度度量 ……271
14.3 多播系统 ……271
14.3.1 进程 ……271
14.3.2 多播通道 ……271
14.3.3 异步多播系统 ……272
14.4 参考文献注释 ……272
14.5 习题 ……272
第15章 基本异步网络算法 ……274
15.1 环中的领导者选举 ……274
15.1.1 *LCR*算法 ……275
15.1.2 *HS*算法 ……278
15.1.3 *Peterson Leader*算法 ……278
15.1.4 通信复杂度的下界 ……281
15.2 任意网络中的领导者选举 ……286
15.3 生成树的构造、广播和敛播 ……287
15.4 广度优先搜索和最短路径 ……290
15.5 最小生成树 ……295
15.5.1 问题描述 ……295
15.5.2 同步算法：回顾 ……296

15.5.3 *GHS*算法：概要……296
15.5.4 更详细的算法 ……297
15.5.5 特殊消息 ……299
15.5.6 复杂度分析 ……301
15.5.7 *GHS*算法的正确性证明……301
15.5.8 简单“同步”策略 ……302
15.5.9 应用到领导者选举算法中 ……302
15.6 参考文献注释 ……303
15.7 习题 ……303
第16章 同步器 ……307
16.1 问题 ……307
16.2 局部同步器 ……309
16.3 安全同步器 ……313
16.3.1 前端自动机 ……314
16.3.2 通道自动机 ……315
16.3.3 安全同步器 ……315
16.3.4 正确性 ……315
16.4 安全同步器的实现 ……316
16.4.1 同步器*Alpha* ……316
16.4.2 同步器*Beta*……317
16.4.3 同步器*Gamma* ……317
16.5 应用 ……320
16.5.1 领导者选举 ……321
16.5.2 深度优先搜索 ……321
16.5.3 最短路径 ……321
16.5.4 广播与确认 ……321
16.5.5 最大独立集 ……321
16.6 时间下界 ……321
16.7 参考文献注释 ……324
16.8 习题 ……324
第17章 共享存储器与网络 ……326
17.1 从共享存储器模型到网络模型的转换 ……326
17.1.1 问题 ……326
17.1.2 无故障时的策略 ……327
17.1.3 容忍进程故障的算法 ……332
17.1.4 对于$n/2$故障的不可能性结果 ……335
17.2 从网络模型转换到共享存储器模型 …336
17.2.1 发送/接收系统 ……336
17.2.2 广播系统 ……338
17.2.3 异步网络中一致性的不可能性 …338
17.3 参考文献注释 ……339
17.4 习题 ……339
第18章 逻辑时间 ……341
18.1 异步网络的逻辑时间 ……341
18.1.1 发送/接收系统 ……341
18.1.2 广播系统 ……343
18.2 使用逻辑时间的异步算法 ……344
18.2.1 时钟的走动 ……344
18.2.2 延迟未来事件 ……345
18.3 应用 ……346
18.3.1 银行系统 ……346
18.3.2 全局快照 ……348
18.3.3 模拟一台单状态机器 ……349
18.4 从实际时间算法到逻辑时间算法的变换*……352
18.5 参考文献注释 ……352
18.6 习题 ……353
第19章 一致全局快照和稳定属性检测 …355
19.1 发散算法的终止检测 ……355
19.1.1 问题 ……355
19.1.2 *DijkstraScholten*算法 ……356
19.2 一致全局快照 ……360
19.2.1 问题 ……360
19.2.2 *ChandyLamport*算法 ……361
19.2.3 应用 ……364
19.3 参考文献注释 ……366
19.4 习题 ……367
第20章 网络资源分配 ……369
20.1 互斥 ……369
20.1.1 问题 ……369
20.1.2 模拟共享存储器 ……370
20.1.3 循环令牌算法 ……370
20.1.4 基于逻辑时间的算法 ……372
20.1.5 *LogicalTimeME*算法的改进 ……374
20.2 通用资源分配 ……376
20.2.1 问题 ……376
20.2.2 着色算法 ……377
20.2.3 基于逻辑时间的算法 ……377
20.2.4 无环有向图算法 ……378
20.2.5 哲学家饮水* ……379
20.3 参考文献注释 ……383

20.4 习题 ……383
第21章 带进程故障的异步网络计算 ……386
21.1 网络模型 ……386
21.2 有故障环境中一致性的不可能性 ……387
21.3 随机算法 ……388
21.4 故障检测器 ……390
21.5 k—致性 ……393
21.6 近似一致性 ……394
21.7 异步网络的计算能力* ……395
21.8 参考文献注释 ……396
21.9 习题 ……396
第22章 数据链路协议 ……399
22.1 问题阐述 ……399
22.2 Stenning协议 ……400
22.3 位变换协议 ……403
22.4 可重排序的有界标志协议 ……406
22.4.1 关于重排序和复制的不可能性结论 ……407
22.4.2 容许丢失和重排序的有界标志协议 ……408
22.4.3 不存在容许消息丢失和重排序的高效协议 ……412
22.5 容许进程崩溃 ……414
22.5.1 简单的不可能性结论 ……415
22.5.2 更复杂的不可能性结论 ……415
22.5.3 实用的协议 ……418
22.6 参考文献注释 ……423
22.7 习题 ……423

第三部分 部分同步算法

第23章 建模V: 部分同步系统模型 ……428
23.1 MMT 定时自动机 ……428
23.1.1 基本定义 ……428
23.1.2 操作 ……432
23.2 通用定时自动机 ……434
23.2.1 基本定义 ……434
23.2.2 将MMT自动机转化为通用定时自动机 ……437
23.2.3 操作 ……440
23.3 属性和证明方法 ……441
23.3.1 不变式 ……441
23.3.2 定时轨迹属性 ……443
23.3.3 模拟 ……444
23.4 构造共享存储器和网络系统的模型 ……449
23.4.1 共享存储器系统 ……449
23.4.2 网络 ……449
23.5 参考文献注释 ……449
23.6 习题 ……450
第24章 部分同步的互斥 ……452
24.1 问题 ……452
24.2 单寄存器算法 ……453
24.3 对时间故障的回复性 ……459
24.4 不可能性结果 ……461
24.4.1 时间下界 ……462
24.4.2 最终时间界限的不可能性结果* ……462
24.5 参考文献注释 ……463
24.6 习题 ……463
第25章 部分同步的一致性 ……466
25.1 问题 ……466
25.2 故障检测器 ……467
25.3 基本结论 ……468
25.3.1 上界 ……468
25.3.2 下界 ……469
25.4 有效算法 ……470
25.4.1 算法 ……471
25.4.2 安全属性 ……472
25.4.3 活性和复杂度 ……473
25.5 涉及时间不确定性的下界* ……475
25.6 其他结果* ……480
25.6.1 同步进程、异步通道* ……480
25.6.2 异步进程、同步通道* ……481
25.6.3 最终时间界限* ……481
25.7 小结 ……483
25.8 参考文献注释 ……483
25.9 习题 ……483
参考文献 ……486
索引 ……512

第1章 引 言

1.1 相关主题

分布式算法（*distributed algorithm*）的概念包括大量并发算法，这些算法有着广泛的应用。最初，分布式算法这个术语用来指在那些分布在一个大的地理区域中的多个处理器上运行的算法。但多年之后，该术语的应用更为广泛。现在的分布式算法不仅包括运行在局域网上的算法，甚至还包括针对共享存储器多处理器的算法。造成这种状况的原因是：人们已经逐渐认识到，用在上述不同环境下的算法有许多共同之处。

分布式算法得到了广泛的应用：电信、分布式信息处理、科学计算以及实时进程控制。当我们为某项应用搭建一个系统时，其中一个重要的环节是设计、实现和分析分布式算法。这些算法和它们要解决的问题构成了本书中涉及到的研究领域的相关主题。

分布式算法有许多种类型。它们的分类所依据的属性包括：

- 进程间通信(IPC)的方法：分布式算法运行在一组处理器上，而这些处理器需要某种方式的通信。一些常规的通信方法包括访问共享存储器、发送点对点或广播的消息（在广域网或局域网上)以及执行远程过程调用。
- 时序模型：关于系统中事件的时序可作几种不同的假设，这反映了算法可能用到的不同时序信息类型。一种极端情况是处理器完全同步，通信和计算在完美的锁一步同步中进行。另一种极端情况是它们完全异步，以任意的速度和次序运行。两者之间有大量可能的假设，这些假设可以归为部分同步的，在这些情况下，处理器具有关于事件时序的部分信息。例如，处理器的相对速度可能有界限，或者处理器可以访问近似同步的时钟。
- 故障模型：算法运行时的底层硬件可能被假设为是完全可靠的。或者，算法可能需要容忍一定限度的故障行为。这些故障行为可能包括处理器故障，即处理器可能在给出或不给出警告的情况下停止工作；也可能暂时发生错误；或表现出严重的*Byzantine*故障，即一个出错的处理器可以作出任意动作。出错行为也包括通信机制的故障，包括消息丢失或消息重复。
- 需解决的问题：当然，算法也视它们试图解决的问题而相异。我们考虑的典型问题就是在上面提到的应用领域内产生的问题。这些问题包括资源分配、通信、分布式处理器之间的一致性、数据库并发控制、死锁检测、全局快照、同步以及各种对象类型的实现等。

本书不讲述某些并发算法，如并行随机存取机(PRAM)算法和针对固定连接网络(如数组、树和超立方体)的算法。与大部分并发算法不同，本书提到的算法具有更高程度的不确定性（*uncertainty*）和行为独立性（*independence of activity*）。本书中的算法必须面对的几种不确定性和行为独立性包括：

- 处理器数目未知
- 网络拓扑结构未知
- 不同位置上的独立输入
- 几个程序立即运行，在不同时间开始，以不同速度运行
- 处理器的不确定性

- 不确定的消息传递次数
- 不确定的消息顺序
- 处理器和通信故障

幸运的是，并不是每个算法都要面对所有这些不确定性！

由于这些不确定性，分布式算法的行为经常很难理解。即使算法的代码很短，由于多个处理器并行执行代码，其执行步骤以某种不确定的方式相互交错，也会意味着即使在输入相同的情况下，算法有多种不同的表现。因此，通过准确预测算法到底如何执行来理解算法通常是不可能的。这可以对比其他并行算法，例如，在PRAM算法中，我们通常能够预测某一时刻算法将要做什么。对于分布式算法，我们最好是理解它的行为中某些已选定的属性，而不是去理解它的行为的一切。

在过去的15年里，分布式算法的研究已经发展成一个相当一致的领域。该领域的研究风格大致如下：首先，确认在实际分布式计算中的重要问题，并定义适合对问题进行数学研究的抽象版本。然后，开发出解决问题的算法。精确地描述这些算法，证明它们能解决出现的问题，并且根据不同度量标准来分析其复杂度。算法的设计者通常会试图降低算法的复杂度。同时，要证明不可能性结果和下限，给出问题如何才能可解的限制以及其求解代价。所有这些工作的基础是分布式系统的数学模型。

这些结论构成了十分有趣的数学理论。但是它们不仅仅是数学理论：这里讲到的问题声明可用于对实际系统的某些部分建立形式化规格说明；这里讲到的算法（很多情况下）可用于实际的设计；这里给出的不可能性结果会告诉设计者应在何时停止做一些事情。所有这些结果，
3 加上底层的数学模型，有助于设计者理解他们构建的系统。

1.2 我们的观点

本书研究的领域是分布式算法。因为这是一个非常广阔和活跃的领域，我们不能作全面的研究。必须有所取舍，所以我们试图选出本领域在理论上和实际上最基本的一些结果。就复杂度来讲，它们不一定是最优的结果，但我们更倾向于选出那些简单且体现了重要的设计和推理方法的结论。我们给出的结果包括一部分在本领域内相当典型的问题，如领导者选举、网络搜索、构造生成树、分布一致性、互斥、资源分配、构造对象、同步、全局快照以及可靠通信。这些问题在不同应用中重复出现。我们会在几个不同的系统模型中考虑这些问题。

本书的一大特点是，我们根据一个几乎统一的形式化框架来给出所有的算法、不可能性结果和下限。这个框架包括少量针对不同类型分布式系统的形式化自动机理论模型，以及使用这些模型来对系统进行推理的标准方法。我们的框架基于自动机理论，而不是基于某种特定的形式语言或形式证明逻辑；这使得我们可以用基本的集合论数学来表达结论，而不用过多地担心语言细节。这样也有灵活性，因为可以在同一框架内运用多种语言和逻辑来对算法进行描述和推理。使用形式化框架可以对所有的结论进行严密的处理。

对于严密性还需要多说几句。在分布式算法领域内，严密的处理十分重要，因为其中有许多微妙的因素。如果不注意这一点，就很难避免错误。然而，很难说清怎样作出完全严密的表达既足够简练又易于直观理解。在本书中，我们综合使用了直观的和严密的推理。也就是说，我们给出相应形式化模型的精确描述。对于算法，我们有时用形式化模型精确描述，有时用文字描述，有时两者都用。在讨论算法正确性时，严密的程度可能变化很大：有时给出很形式化的证明，有时仅仅是直观的概述。然而，我们希望提供足够的工具，以便在需要时可以将直观概述扩展为形式化证明。我们一般根据形式化模型来给出严密的不可能性证明。

因为有许多不同的环境和问题要考虑，所以很难说怎样组织这些材料最好。我们首先是按照形式化模型组织的——特别是根据模型中对结论影响最大的方面描述，其次是根据抽象的问题描述。模型之间最深层的区别在于对时序的假设，同时IPC机制和故障假设也是重要的因素。 4

我们考虑的时序模型如下：

- 同步模型：这种模型最容易描述、编程和推理。我们假设各部分同时执行各个步骤，也就是说，运行依照同步轮前进。当然，在大部分分布式系统中，实际情况并非如此，但是无论如何，同步模型是很有用的。理解怎样在一个同步模型中解决问题，往往是理解怎样在更现实的模型中解决它的有用的中间步骤。例如，有时一个实际的分布式系统可以“模拟”一个同步系统。而且，同步系统的不可能性结果可以直接用于表现更差的模型。然而，在许多种类的分布式系统中实现同步模型是不可能或无效的。
- 异步模型：这里我们假设不同部分以任意的顺序、按任意速度执行各个步骤。尽管有几个细微之处，主要涉及活性（liveness）的考虑，这种模型还是比较容易描述的。因为有事件顺序的额外不确定性，它比同步模型更难编程。然而，异步模型允许程序员忽略特定的时序问题。与典型的分布式系统相比，由于异步模型对时间所作的假设更少，因此为异步模型设计的算法是通用和可移植的：它们可以保证按任意的时序在网络中正确运行。然而，异步模型有时不能提供高效解决问题的能力，有的问题甚至不能解决。
- 部分同步（基于时序）模型：这里我们假设对事件的相对时序作一些限制，但运行并不像同步模型那样严格遵循锁-步机制。这些模型是最实际的，但也是最难编程的。利用事件时序知识所设计的算法可以是高效的，但它们也可能是脆弱的，因为如果时序假设被打破的话它们就不能正确运行。 5

我们用来分类的下一个基础是IPC机制。在本书中，我们考虑了共享存储器和消息传递。我们首先给出了共享存储器模型，因为它更为强大，且更易理解，也因为共享存储器环境中的许多技术和结论修改后可以用于网络环境。接下来，我们根据研究的问题来组织材料。最后，我们研究了许多在不同的故障假设下的问题。当我们在不同的模型下提出相同的问题时，将会看到，假设中很小的不同会导致结果产生巨大的差别。我们试图指出并强调这些差别。

通过将算法组合成其他算法，使用层次抽象来设计算法，以及将针对一个模型的算法转换成为针对其他模型的算法，我们尽量将我们的表达模块化。这有助于极大地降低设计思想的复杂性，使我们事半功倍。在实际的分布式系统设计中，相同种类的模块可以用于同样的目的。

1.3 本书内容综述

本书包括的具体主题如下。

模型和证明方法 有关形式化模型和证明方法的内容分散在第2、8、9、14和23章中，这几章是本书中各主要部分（同步网络算法、异步共享存储器算法、异步网络算法和部分同步算法）的起始章节。这些内容独立成章，以便读者参考。首次阅读时你可能更愿意跳过建模部分的某些内容，以后在理解算法的相关内容时根据需要再回来参考它们。我们尽量合理组织本书，使得算法易读和易理解，而不需要太多的形式化建模工作。

我们用到的模型都是基于状态机的，通常有无限多个状态和与其迁移相关的显式名字。一个状态机既可以用于对分布式系统的一个组件建模，也可以对整个分布式系统建模。状态机的每个状态表示这个组件或系统的一个瞬时快照，它包括诸如每一处理器的存储状态、每一运行程序的程序计数器以及通信系统中传递的消息等信息。迁移描述了系统中发生的变化，例如消息的发送和接收引起的变化，或者由本地计算引起的变化。我们对同步网络、异步系统和基于 6

时序的系统分别给出不同的模型。

对于分布式系统而言，形式化模型的一个重要用途是作为所要解决问题的规格说明以及算法正确性证明的基础。这样的规格说明和证明可以用许多特定风格的和特别的方法来做。然而，某些方法会经常采用，所以我们在有关模型的章节中明确地描述了它们。这些方法包括了*不变式断言*和*模拟*。不变式断言是在系统的所有可达状态中都为真的属性。断言通常通过对系统运行的步数做归纳来证明。模拟是一对系统之间的形式化关系，一个代表要解决的问题，另一个代表解决方案；或者一个代表高级的、抽象的解决方案，另一个代表具体的方案。模拟关系一般也用归纳法证明。

第2章包含了第一个模型，它是针对同步网络的。它是一个很简单的模型，仅仅描述了消息交换和计算的同步轮。第8章包含一个异步网络的通用模型——*输入/输出自动机*（*I/O 自动机*）模型。模型的名字表明它明确区分输入和输出迁移，即环境对系统的通信和系统向环境的通信。在一个I/O自动机中，从某一给定的状态开始可能有几个迁移；例如，涉及不同处理器的迁移能够以任意次序执行。因为模型允许迁移次序具有如此大的灵活性，我们引入*活性*的概念以表示某些迁移最终一定会发生。该模型一个有用的特点是它具有一个并行的合成操作，使得将那些以I/O自动机表示的系统组件进行合并后，仍然可以使用I/O自动机来表示合并后的系统。通常，可以基于组件正确性的证明，以模块化的方式来证明合成自动机的正确性。

第8章中的模型很通用，它足以描述异步的共享存储器系统和异步网络（以及许多其他类型的异步系统）；第9章和第14章中分别包含针对共享存储器系统和消息传递系统模型进行裁剪所需要的额外结构。

最后，第23章给出了基于时序的系统的模型。这些模型同样是状态机，但是这里的状态包
7 括时序信息，比如当前时间和不同事件的调度时间。这些模型使我们可以描述基于时序的系统的典型结构，比如本地时钟和超时。

同步网络算法　我们考虑的最简单的（即有最少不确定性的）模型是同步网络模型，模型中所有的处理器都在同步的轮中进行通信和计算。我们不考虑同步的共享存储器算法，因为它们构成了一大研究分支（见本章末尾的参考资料注释）。在网络环境中，我们假设处理器在图或有向图G中的节点上，并通过沿G的边发送消息与它的邻接节点通信。

在第3~7章中，我们考虑同步网络中几个典型的分布式问题。在第3章中，我们从一个涉及环网中计算的简单例子开始。问题是：假设节点上的每个处理器除了一个*惟一的标识符*（*UID*）之外都相同，怎样在环网中选出一个惟一的领导者节点？这里主要的不确定性是处理器实际拥有的UID集合是未知的（尽管已知没有两个处理器有相同的UID）；而且，网络的规模通常是未知的。这个问题主要用于令牌环的局域环网上，其中假设总有一个令牌在循环，使得当前所有者拥有发起通信的特权。然而，令牌有时会丢失，因此处理器有必要执行一个算法来重新产生丢失的令牌。这个重新产生的过程就等于是选出一个领导者。关于领导者选举问题，我们给出了一些基本的理论复杂度的结论。特别是，我们证明了所需的时间和通信量（即消息数）的界限。

接着，在第4章中，我们简略探讨了在更一般的网络中用到的基本算法。特别地，我们给出了一些用于解决诸如*领导者选举*、*广度优先搜索*、*寻找最短路径*、*寻找最小生成树*、以及*寻找节点的最大独立集*等基本问题的算法。这里，典型的不确定性来自未知的UID和未知的网络图。

然后，在第5章和第6章，我们考虑在一个分布式网络中达成一致的问题。在有些问题中，一些处理器需要达成一个共同的决定，即使开始时它们对决定应该是什么样子存在分歧。现实中会产生许多一致性问题，例如：处理器能监测飞机上不同的高度计，并且试图就高度达成一
8 致。或者处理器能对某个其他的系统组件进行故障诊断，并且试图把各个处理器的诊断综合成

一个共同的决定，即是否替换这一组件。

这里，我们考虑的不确定性不仅源于初始判断的差异，而且源于链路或处理器的故障。在第5章中我们考虑链路因丢失消息而产生故障的情况。在第6章中我们考虑两种不同类型的处理器故障：停止故障，即发生故障的处理器在某一点停止执行本地的协议；Byzantine故障，即发生故障的处理器表现出完全随意的行为（其限制是不能破坏系统中它不能访问的部分）。对可以容忍的错误数目、时间以及通信量，我们都给出了界限。

最后，在第7章中，我们考虑了一些基本的一致性问题的扩展和变形，包括：在一个小的值集合上的一致，而不是仅仅对单个值的一致；还包括对一个实数值的近似一致；以及分布式数据库的提交问题。

异步的共享存储器算法 经过同步算法的热身（其中只有少量的不确定性），我们开始学习难度更大的异步算法。现在我们不再假设处理器按锁-步同步，而是可以按任意次序交错执行各步，对单个处理器的速度也没有限制。典型情况下，它和外部环境的交互（通过输入和输出事件）是不断进行的，而不是仅仅包含初始的输入和最终的输出。在这种环境中，得到的结果与同步网络中的大不相同。

第10~13章包括了异步的共享存储器算法。在第10章中，我们考虑的第一个问题是互斥问题。这是在分布式算法领域中最基本的问题之一，也是历史上第一个得到深入的理论研究的问题。本质上讲，这个问题涉及对单一、不可分的资源进行访问时的管理，这种资源在某一时刻只能支持一个用户。从另一角度看，我们可以认为这个问题是要确保某些程序段在临界区内执行，临界区内不允许两个进程同时执行。这个问题在集中式和分布式操作系统中都会遇到。除了与步骤次序有关的基本的不确定性，还有与哪些用户何时访问资源有关的不确定性。

我们给出了一系列共享存储器的互斥算法，从Dijkstra在1965年提出的经典算法开始，列出了一系列能更好地保证正确性的算法。大部分结果基于那些只能用读操作和写操作来访问的存储器；对于这种读/写共享存储器模型，我们也给出了必须用到的共享变量数目的下限。我们也考虑了使用一个更强的共享存储器——读-改-写存储器的问题；在这种情况下，我们给出了所需存储器大小的上下限。除了给出算法和下限，我们把互斥问题作为一个实例来阐明异步分布式算法的许多重要概念。这些概念包括通用建模方法；原子性、公平性、演进性和容错的概念；不变式断言和模拟证明；以及时间分析技术。

在第11章中，我们将互斥问题扩展为更复杂的资源分配问题，涉及更多的资源和对资源使用方式的更详尽的要求。例如，我们考虑哲学家用餐问题，这是一个典型的资源分配问题，涉及在处理器环中分配成对的共享资源。

在第12章中，我们在异步共享存储器模型中重新考虑一致性问题。这一章的主要结论是一个基本的事实，即如果共享存储器仅仅支持读写操作，那么在发生故障的情况下，基本的一致性问题在这样的环境中不可解。相反，对于较强的共享存储器类型，如读-改-写存储器，存在对这个问题的简单解法。

接着，在第13章中，我们介绍了原子对象。在此之前，我们假设处理器对存储器的访问是即时的。原子对象分别接受调用和作出响应动作，但除此之外它的表现和即时访问的共享变量很相似。我们定义了原子对象，并证明了一些基本结论，这些结论说明了怎样使用它们来构造系统；特别是它们可用于代替共享变量。同时我们考虑了使用更弱原语（共享变量或较弱类型的原子对象）来实现更强原子对象的几个算法。这些算法的一个有趣的属性是无等待，这意味着不管其他并发操作是否出错，在已实现的对象上的任一操作必须完成。

我们显示了怎样用读/写共享变量来实现快照原子对象；快照原子对象接受快照操作，立

即返回所有存储位置的值。我们也说明了怎样用单写者的读/写共享存储器来实现多写者/多读者的原子对象。

10 **异步网络算法**　在第15~22章中，我们进一步研究了异步网络中的算法。和同步网络一样，系统以图或有向图来表示，节点是处理器，边是通信链路，但是系统不在同步轮中运行。现在，消息可以在任意时间到达，处理器可以以任意速度运行。相对于同步网络环境和异步共享存储器环境，这里的系统组件之间的耦合更松散。因此，这种模型的不确定性又增加了。

我们从第15章开始，在异步网络环境中重新考虑第4章里的问题和算法。例如，我们重新考虑领导者选举问题、广度优先搜索和最短路径、广播和聚播以及最小生成树。尽管有些算法几乎不用修改就可以用于新的环境，但大部分需要较大的改动。把第4章的简单同步最小生成树算法扩展到异步环境尤其困难。

第15章说明对异步网络编程是困难的，接下来的四章就是要解决这个问题。在第16~19章中，我们介绍了四种简化技术，这些技术被形式化为*算法变换*，允许异步网络模拟更简单或更强的模型。这些变换使得针对更简单或更强的模型而设计的算法可以运行在更复杂的异步网络模型中。

在第16章中介绍的第一种技术是引入*同步器*。同步器是这样的一种系统组件，它使得（无故障的）异步网络可以模拟第2~4章中的同步网络（没有故障的那些）。我们给出了有效的实现，并且把这些实现和一个下限结果做比较，其中这个下限结果似乎表明任何这样的模拟都是无效的。这种表面上的矛盾其实依赖于待解决问题的类型。

在第17章中介绍的第二种技术是用异步网络模型模拟异步共享存储器模型。这使得第10~13章中介绍的那些异步共享存储器算法可以用在异步网络中。

在第18章中介绍的第三种技术是给异步分布式网络中的事件分配一致的*逻辑时间*。这种技术允许异步网络模拟一个网络，该网络中的节点可以访问一个完全同步的实时时钟。它的重要
11 应用是允许一个异步网络模拟一个集中式（非分布式）状态机。

第19章包含了我们的第四种技术，在异步网络算法运行时监控它们。例如，这可以用于调试目的，用于产生备份版本，或者用于检测算法的*稳定属性*。稳定属性是一旦发生就会永远保持的属性，例如系统*终止*或*死锁*。有助于检测稳定属性的一种基本原语其实就是这样的一种能力，即产生分布式算法的状态的*一致全局快照*。我们给出了几种产生这样的快照的方法，并且描述了怎样使用快照来检测稳定属性。

有了这些有力的工具之后，我们考虑异步网络环境中的特定问题。在第20章中，我们重新考虑资源分配问题。例如，我们给出了在异步网络中解决互斥和哲学家用餐问题的方法。

在第21章中，我们考虑了出现停止故障时异步网络中的计算问题。首先，用第17章中的变换，我们证明了一致性问题从共享存储器环境到网络环境的不可能性结果。围绕这个固有的限制我们考虑了几种方法；例如，我们给出了一个*随机化算法*来解决一致性问题，介绍了怎样用所谓的*故障检测器*来解决一致性问题，说明了怎样达到*近似一致*而不是精确一致。

在第22章中，我们考虑*数据链路*问题。数据链路协议用于在不可靠的底层通道上实现可靠的通信链路。首先我们给出**交替位**（Alternating Bit）协议，这个简单协议不但本身有趣，而且在并发算法证明领域也是一个著名的标准案例研究。对于底层通道出现不同类型故障行为的环境，我们也给出了这个问题的其他各种算法和不可能性结果。

部分同步算法　部分同步模型处于同步和异步模型之间。在部分同步模型中，我们假设处理器知道一些时间信息，例如能够访问实际时间或近似时间，或者某种类型的超时机制。或者，我们可以假设处理器每步的时间和/或消息发送的时间在已知的上下限之间。由于部分同步系统

具有比异步系统更少的不确定性，你可能以为它们更容易编程。然而，时序引起了更多的复杂问题——例如，算法的正确性通常严重依赖于时序假设。因此，部分同步环境的算法和证明通常比异步环境更复杂。 12

在第24章中，我们给出在时序环境中解决互斥问题所需时间的上下限，在第25章中，我们给出一致性问题的上下界。因为部分同步分布式算法是当前研究的一个主题，我们只给出这种模型的初步结论。

1.4 参考文献注释

书中材料的主要来源是学术文献，特别是在计算机协会（ACM）的分布式计算原理（PODC）年会论文集上发表的论文。在这一领域包含大量论文的其他年会论文集包括计算机科学原理（FOCS）、计算理论（STOC）、并行算法和体系结构（SPAA）、以及分布式算法（WDAG）。其中很多内容已经出现在一些计算机科学杂志中，如《*Journal of the ACM*》，《*Distributed Computing*》，《*Information and Computation*》，《*SIAM Journal on Computing*》，《*Acta Informatica*》和《*Information Processing Letters*》。这些论文中的结论是根据大量不同的模型和以不同的严密程度来描述的。

以前已经有过几次对这一领域的资料进行收集和总结的尝试。在《*Handbook of Theoretical Computer Science*》[185]中，由Lamport和Lynch写的有关分布式计算的章节综述了一些建模和算法的思想。Raynal的两本书[249，250]分别描述了互斥算法和异步网络算法。Raynal和Helary的另一本书[251]给出了网络同步器的有关结论。Chandy和Misra[69]用UNITY编程模型给出了大量的分布式算法。Tel[276]表达了本领域的另一种观点。

对于同步共享存储器系统的PRAM模型的结论，被Karp和Ramachandran[166]收集在论文中。对于固定连接的网络的同步并行算法，Leighton[193]在一本书中作了总结。对于分布式数据处理系统的并发控制和恢复，Lynch、Merritt、Weihl和Fekete[207]，以及Bernstein、Hadzilacos和Goodman[50]提出了许多算法。Hadzilacos和Toueg[143]给出了分布式系统的实现结果，它们基于具有原子广播原语的通信系统。 13

本书用到了许多图论中的概念。相关的标准参考资料是Harary[147]的经典著作。

1.5 标记

这里列出了本书中用到的数学标记。

$\mathbb{N}$ 表示自然数，{0，1，2，...}

$\mathbb{N}^+$ 表示正整数，{1，2，...}

$R^{\geq 0}$表示非负实数。

R^+表示正实数。

λ表示空字符串。

如果β是任意一个序列，S是任意一个集合，那么$\beta|S$表示β的一个子序列，它由S在β中的所有元素组成。

14

第一部分 同步网络算法

本书的第一部分由第2~7章组成，这几章包括同步网络模型的算法和时间下限，其中网络中的处理器在同步轮中执行步骤和交换消息。

作为这一部分的头一章，第2章只给出了同步网络的形式化模型。现在可以跳过本章，而在阅读介绍算法的章节第3~7章时，如有需要再返回来阅读。第3章讲述一个简单的问题——如何在环网中选取惟一的领导者。第4章概述基于任意图的同步网络中使用的基本算法。第5章和第6章分别介绍了出现链路故障和处理器故障时，如何在同步网络中达到一致性的基本问题。最后，第7章介绍了基本一致性问题的扩展和变形。

第2章 建模I：同步网络模型

本章是书中最短的一章，因为其所有任务在于给出针对同步网络算法的简单计算模型。我们单独介绍这个模型，以便在阅读第3~7章时，可以方便地参考本章。

2.1 同步网络系统

同步网络系统由一组位于有向网络图节点位置的计算元素组成。在第1章，我们称这些计算元素为“处理器”，这说明它们是硬件。但是，把它们看成是逻辑软件“进程”往往是很有用的，这些进程在实际的硬件处理器上运行（而不是等同于处理器）。我们给出的结论适用于上述两种情况。在本书中，自现在开始，我们约定称计算元素为“进程”。

为了形式化地定义同步网络系统，我们从有向图G开始，$G=(V, E)$。我们用n表示网络有向图中的节点个数，即$|V|$。对于G中的任意节点i，我们用$out\text{-}nbrs_i$表示i的“出向邻接节点”，也就是说，在G中存在i到这些节点的有向边；同样，$in\text{-}nbrs_i$表示“入向邻接节点”，即在G中存在这些节点到i的有向边。我们用$distance(i, j)$表示在G中从i到j的最短有向路径的长度；如果不存在这样一条路径，则设$distance(i,j)=\infty$。我们定义$diam$，即直径，为所有(i,j)中
17 $distance(i,j)$的最大值。我们假设M是一个固定的消息集合，$null$是一个占位符，表示没有消息。

对应每个$i \in V$，存在一个进程，它在形式上由下列几个部分组成：

- $states_i$，一组状态的集合（不一定是有限集）
- $start_i$，$states_i$的非空子集，称为开始状态集或初始状态集
- $msgs_i$，一个消息生成函数，从$states_i \times out\text{-}nbrs_i$映射到$M \cup \{null\}$
- $trans_i$，一个状态迁移函数，从状态集$states_i$和$M \cup \{null\}$中元素组成的向量（以$in\text{-}nbrs_i$为下标）映射到$states_i$。

可见，每个进程都有一个状态集合，其中又分出一个子集——开始状态集合。状态集不一定是有限集，这一点很重要，因为这样我们可以表示某些系统，其中包含像计数器这样的无界数据结构。对于每个状态和出向邻接节点，消息生成函数定义了，从给定状态开始，进程i向指定的邻接节点发出的消息。状态迁移函数则定义了，针对每个状态和所有来自入向邻接节点的消息的集合，进程i所要迁移到的状态。

对应于G中的每一条边(i, j)，存在一条通道，也称为链路，它是在任意时刻持有M中最多一条消息的一个场所。

整个系统的运行以所有进程处于任意开始状态和所有通道都为空开始。之后，所有的进程在每个时间步重复执行下列步骤：

1）对当前状态应用消息生成函数，生成所有向出向邻接节点发送的消息。把这些消息放在相应的通道上。

2）对当前状态和入向消息应用状态迁移函数，获得新的状态。删除通道上的所有消息。

上述两步的组合称为一轮(*round*)。注意，进程往往需要通过计算来确定消息生成函数和状态迁移函数的值，通常我们不对计算量加以限制。同时，我们讨论的模型是确定性模型，也就是说消息生成函数和状态迁移函数是单值函数。因此，如果给出初始状态的一个特定集合，计
18 算将以惟一的方式展开。

停止 到目前为止，我们还没有对进程的停止做任何规定。然而，分辨出哪些状态是停止状态并指出从这些状态开始不会有进一步的活动是很容易的。也就是说，不会有任何消息生成，同时惟一的状态迁移是自循环。注意，这些停止状态在我们的系统中所扮演的角色与其在传统的有限状态自动机中是不同的。在有限状态自动机中，停止状态是作为接受状态存在的，用来决定哪些字符串属于该机器所计算的语言。而在这里，它只是用来停止进程；进程所计算的内容必须取决于其他的约定。接受状态这种概念往往不适用于分布式算法。

变量开始时间 有时，我们需要考虑在同步系统中进程有可能从不同的轮开始执行。我们通过扩展网络图来表示这种情况，使网络图中包括一个特殊的环境节点，该节点与所有的普通节点相连。相关的环境进程的工作是向其他所有节点发送特殊的唤醒消息。每个其他进程的所有初始状态都要求是休眠的，即这些状态不会生成任何消息，并且，只有当它从环境进程收到唤醒消息或者从其他进程收到非空消息时，它才迁移到新的状态。这样，一个进程可以被外界的唤醒消息直接唤醒，也可以被其他已经唤醒的进程的非空消息唤醒。

无向图 有时，我们需要考虑底层的网络图是无向图的情况。把无向图看作是具有双向边的有向图，从而使用我们已经针对有向图定义的模型来表示这种情况。在这种情况中，用$nbrs_i$来代表i的邻接节点。

2.2 故障

我们将考虑同步系统的各种故障，包括进程故障和链路（通道）故障。

一个进程可能在执行期间突然停止而产生停止故障。根据模型，进程可能在上面的步骤1或步骤2的某个实例之前或之后发生故障；另外，我们允许进程在执行步骤1的过程中发生故障。这意味着进程有可能只把它预期要产生的消息的一个子集成功地放入消息通道。我们假设该子集可以是任意子集——我们并不认为进程是顺序地产生它的消息并且在序列中间某处发生故障。 19

一个进程还可能发生*Byzantine*故障，也就是说，它可能以随意的方式产生下一个消息和下一个状态，而不遵循它的消息产生和状态迁移函数所指定的规则。

一个链路可能由于丢失消息而发生故障。根据模型，一个进程可能在步骤1期间试图把消息放入通道，但是，一个出错的链路则可能不记录该消息。

2.3 输入和输出

我们还没有提供表示输入输出的机制。我们利用了简单的约定，将状态中的输入输出编码。特别地，输入放在开始状态中指定的输入变量中；一个进程可以有多个开始状态，这一事实在这里很重要，这样就能适应不同的可能输入。事实上，我们通常假定有多个开始状态的惟一原因在于输入变量中有可能存在不同的输入值。输出则出现在指定的输出变量中；每个输出变量只记录所执行的第一个写操作的结果（也就是说，它是一次写入变量）。然而，输出变量可被读任意多次。

2.4 运行

为了推断同步网络系统的行为，我们需要对系统“运行”有一个形式化的描述。

一个系统的状态赋值 (*state assignment*)被定义为对该系统中的每个进程赋值一个状态。同样地，消息赋值(*message assignment*)是对每个通道赋值一条消息（可以是空消息）。系统的运行定义为一个无限的序列：

$$C_0,\ M_1,\ N_1,\ C_1,\ M_2,\ N_2,\ C_2,\ \ldots,$$

其中，C_r是状态赋值，M_r和N_r为消息赋值。C_r代表r轮之后的系统状态，而M_r和N_r则分别代表第r轮所发送和接收的消息。（因为通道有可能丢失消息，所以它们有可能不同）。我们通常把C_r
20 当成时刻r的状态赋值；也就是说，时刻r是指执行r轮之后的时间点。

设α和α'是一个系统的两次运行。如果在α和α'两次运行中，进程i有相同的状态序列、相同的输入消息序列和相同的输出消息序列，则称α和α'对于进程i是不可区分的（*indistinguishable*），记作$\alpha \sim_i \alpha'$。如果在α和α'两次运行中，进程i直到第r轮都有相同的状态序列、相同的输入消息序列和相同的输出消息序列，则称α和α'对于进程i是r轮不可区分的。我们还把这一概念扩展到两次运行处于两个不同同步系统中的情形。

2.5 证明方法

对同步系统作出推断的最重要方法是对*不变式断言*的证明。一个不变式断言是指系统状态（尤其是所有进程的状态）的一个属性，该属性在每一轮之后的每次运行中都成立。我们允许在断言中提到已完成的轮数，这样就可以声明每次r轮之后的状态。同步系统的不变式断言一般通过对已完成的轮数r的归纳来证明，从$r = 0$开始。

另一个重要的方法是*模拟*。粗略地说，其目的是为了表明一个同步算法A“实现”了另外一个同步算法B，即两者产生相同的输入/输出行为。当两个算法从相同的输入开始，在相同的轮数中出现相同的故障形式时，A和B之间的对应关系用与A和B的状态相关的断言来表述。这样的断言被称为*模拟关系*。与不变式断言相似，模拟关系一般通过对已完成的轮数的归纳来证明。

2.6 复杂度度量

对于同步的分布式算法，我们通常考虑两种复杂度度量：时间复杂度和通信复杂度。

同步系统*时间复杂度*的度量是根据所有要求的输出都生成或者进程全部停止时已运行的轮数。如果系统允许可变的开始时间，则时间复杂度是从任一进程被唤醒的第一轮开始度量。

21 *通信复杂度*通常是按已发送的非空消息的个数来度量。有时，我们也会考虑到消息的比特数。

在实践中，时间度量更为重要一些，不仅是对同步的分布式算法，对所有的分布式算法都是这样的。通信复杂度只有当拥塞严重到使处理进程缓慢时才变得重要。这说明我们似乎可以忽略通信复杂度而只考虑时间复杂度。然而，通信负载对时间复杂度的影响不仅仅是某个单独的分布式算法的函数。在一个典型网络中，许多分布式算法同时运行，共享同一网络带宽。任一单个算法加入链路的消息负载都会增加该链路总的消息负载，从而有助于形成影响所有算法的拥塞。因为要量化任一算法的消息对其他算法性能的影响是很难的，这里我们只简单分析单个算法所产生的消息数（同时试图使其最小化）。

2.7 随机化

我们不要求进程是确定性的，有时候，允许它们基于某一给定的概率分布进行随机选择也是很有用的。由于基本的同步系统模型不允许这一点，我们对其进行扩展，即除了消息产生和状态迁移函数之外，又引进一个随机函数来表示随机选择步骤。用形式化描述就是在每一节点i的自动机描述中增加一个$rand_i$部分；对于每个状态s，$rand_i(s)$是$states_i$的某个子集上的一个概率分布。现在，在每轮运行中，首先用随机函数$rand_i$选出新的状态，然后按常规应用$msgs_i$和$trans_i$函数。

在随机化算法中，运行的形式化表示不仅包括状态赋值和消息赋值，还包括关于随机函数的信息。特别地，系统的一次运行被定义为一个无界序列

$$C_0, D_1, M_1, N_1, C_1, D_2, M_2, N_2, C_2,...,$$

其中，C_r和D_r是状态赋值，M_r和N_r是消息赋值。D_r表示第r轮随机选择后进程的新状态。

对随机系统计算内容的声明通常是概率化的，断言某些结果至少有一定的出现概率。当作出这样的声明时，一般认为它对所有输入和所有故障模式（如果系统存在故障）都成立。为了 22
对输入和故障模式建模，通常假设有一个称为“对手”的假想实体控制着输入的选择和故障的发生，那么，概率性声明断言系统在与任意合法对手的竞争中表现良好。对这些情况的处理超出了本书的范围，我们只在需要时提供特殊情况定义。

2.8 参考文献注释

状态机模型的一般概念源于传统的有限状态自动机。关于有限状态自动机的基本内容在许多大学本科的教材中都有，如Lewis和Papadimitriou[195]，以及Martin[221]的著作。这里定义的某种特定的状态机模型是从大量分布式计算理论的论文中提取出来的，如Fischer和Lynch[119]的Byzantine一致性论文。

不变式断言的思想可能是Floyd[124]关于串行程序中首次提出的，后来被Ashcroft[15]和Lamport[175]推广到并发程序。其他许多地方也用了相似的思想。模拟的思想也有大量来源。最重要的一个来自串行程序中数据抽象方面的早期工作，例如Liskov的编程语言CLU[198]，以及Milner[228] 和Hoare[158]的工作。后期的工作将这种概念扩展到并行程序，这包括Park[236], Lamport[177], Lynch[203], Lynch和Tuttle[218], 以及Jonsson[165]的论文。 23 ~ 24

第3章 同步环中的领导者选择

在这一章中，我们给出了利用第2章的同步模型要解决的第一个问题：从一个网络的各个进程中选举一个惟一的领导者。作为开始，我们考虑网络图为环形的简单情形。

这个问题源于对局域令牌环网的研究。在这样的一个网络中，一个“令牌”在网络中循环，令牌的当前持有者拥有发起通信的惟一特权。（如果网络中的两个节点同时尝试通信，就会产生互相干扰）。然而，有时候，这个令牌会丢失，所以进程就有必要执行一个算法来重新产生丢失的令牌。这个重新产生的过程就等于选举一个领导者。

3.1 问题

我们假设网络有向图G是一个由编号为1到n的n个节点构成的顺时针方向的环（见图3-1）。我们经常使用模n计数，所以0是进程n的另一个名字，而$n+1$是进程1的另一个名字，依此类推。与G的节点相关的进程并不知道自己的编号（即下标），也不知道它们邻接节点的编号；我们假设消息产生和迁移函数都根据本地的、相对于邻接节点的名字来定义。然而，每个进程都必须能够将它的顺时针邻接节点和逆时针邻接节点区分开来。最后的要求是：应该恰好只有一个进程输出这个决定：它就是领导者，将其状态中的特定部分改为
25 值“领导者”。这个问题有如下几个版本：

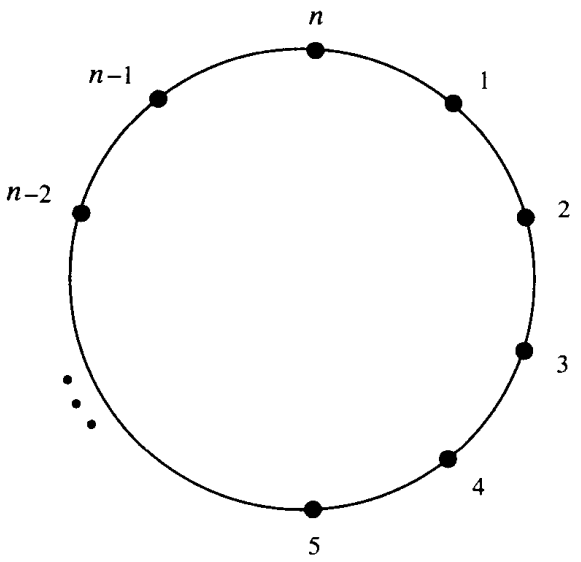


图3-1 进程环

1）可能要求所有非领导者的进程把其状态中的特定部分改为“非领导”值，最终输出自己不是领导者。

2）环有可能是单向也有可能是双向的。如果是单向的话，每条边都从一个进程指向其顺时针的邻接节点，也就是说，消息只能依照顺时针的方向发送。

3）在环中，对于进程来说，节点的个数可能是已知的，也有可能是未知的。如果是已知的，则意味着进程只需要在规模为n的环中正确工作，因此，它们可以在程序中利用值n。如果是未知的，就意味这个进程必须能在节点数不同的环中工作，此时不能使用有关环规模的信息。

4）进程可以是相同的，也可以用进程开始时的惟一标识符（UID）来区分，UID取值于某个大的全序标识符空间，如正整数集$\mathbb{N}^+$。我们假设在环中每个进程的UID都是不一样的，但
26 是并不限定哪些UID实际出现在环中。（例如：它们并不一定是连续的整数。）这些标识符可以只限于接受某些特定操作，如比较操作，或者它们允许不受限制的操作。

3.2 相同进程的不可能性结果

一个简单的观察告诉我们，如果所有的进程都相同的话，那么领导者选举问题就根本不可

能在给定的模型中解决，即使这个环网是双向的，并且对进程而言环的规模是已知的。

定理3.1 令A是一个拥有n个进程的双向环系统，其中$n>1$。如果在A中的所有进程都是相同的，那么A不可能解决领导者选举问题。

证明 假设有这么一个系统A解决了领导者选举问题，我们来导出矛盾。不失一般性，我们可以假设A的每个进程都只有一个开始状态。因为如果每个进程有多于一个开始状态的话，我们就可以选择其中一个，并得到其中每个进程中只有一个开始状态的新解。在这个假设条件下，A只有一个运行。

考虑A的（惟一）运行。通过对已经运行的轮数r做归纳，容易验证所有的进程在r轮后都会处于相同状态。所以，如果任一进程到达了领导者状态，那么A中的其他进程也会在同一时间到达领导者状态。但是这就违反了惟一性的要求。 □

定理3.1暗示着解决领导者选举问题的惟一途径就是打破对称性。在实践中得到的一个合理假设就是除了UID之外，进程都是相同的。这也是我们在本章剩下部分所作的假设。

3.3 基本算法

第一个解法相当显而易见，我们称其为*LCR*算法，以表彰Le Lann、Chang和Roberts，这种算法正是取自他们的论文。这个算法只使用单向通信，并且不依赖于是否知道网络的规模。只有领导者才会执行输出。算法只需要对UID使用比较操作。以下是*LCR*算法的非形式化描述。 27

***LCR*算法（非形式化）：**

每个进程都绕环发送自己的标识符。当一个进程接收到一个输入标识符时，就把它和自己的标识符作比较。如果输入的标识符比自己的标识符大，就继续传递这个标识符；如果这个标识符比自己的标识符小，就丢弃输入标识符；如果和自己的标识符相同，就宣称自己是领导者。

在这个算法中，拥有最大UID的进程是惟一的领导者。为了证明这个直觉是正确的，我们将按第2章的模型来给出一个更加详细的算法描述。

***LCR*算法（形式化）：**

字母表M是UID的集合。

对于每个i，$states_i$中的状态由以下部分组成：

u，一个UID，初值等于i的UID

send，一个UID或者*null*，初值等于i的UID

status，从{*unknown*,*leader*}中取值，初值为*unknown*。

开始状态$start_i$的集合由给定的初始化中定义的单个状态组成。

对于每个i，消息产生函数$msgs_i$按如下定义：

send the current value of *send* to process $i+1$

事实上，进程i会使用一个和进程$i+1$相关的名字，比如“顺时针邻接节点”，为简便起见，我们写成$i+1$。回忆在第2章中，我们用值*null*来作为一个占位符，表示消息不存在。所以，如果*send*的值为*null*的话，那么这个msg_i函数实际上并没有发送任何消息。

对于每个i来说，状态转移函数$trans_i$用下面的伪码定义： 28

send := *null*

if the incoming message is v, a UID, then

```
case
  v > u: send := v
  v = u: status := leader
  v < u: do nothing
endcase
```

这个转移函数的第一行清除了以前的消息传递（如果有的话）对状态产生的影响。其余的代码包括了以下工作——决定是继续传递还是丢弃输入的UID，或者接受它，使自己成为领导。

算法的描述应当使用可读性好的程序语言，但是注意它可以直接翻译成在第2章的模型中的进程状态机。在这样的翻译中，每个进程状态由所有变量的值组成，而且状态转移可以根据变量的变化来描述。注意，作为单独一轮中的处理部分，整个$trans_i$函数的代码不可分割地执行。

我们怎样才能形式化地证明这个算法是正确的呢？正确性的含义是恰好有一个进程最终产生“领导者”输出。用i_{max}表示UID最大的进程的下标，用u_{max}表示它的UID。需要证明：（1）进程i_{max}在第n轮结束时，输出“领导者”；（2）没有别的进程输出“领导者”。我们分别在引理3.2和3.3中证明这两个属性。

在书中的很多地方，我们都给状态分量加了下标i来表示这个分量是属于进程i的。比如：我们用符号u_i来表示进程i的状态分量u的值。但是一般我们会在编写代码的时候忽略这些下标。

引理3.2　进程i_{max}在第n轮结束时输出“领导者”。

证明　注意到u_{max}是初始化时变量$u_{i_{max}}$的初始值，即变量u在进程i_{max}处的值。同时注意到变量u的值从不（被代码）改变，（根据假设）它们都是各不相同的，而且（由i_{max}的定义）i_{max}具有u的最大值。从代码看，已经足够来说明以下的不变式断言：

断言3.3.1　在n轮以后，$status_{i_{max}} = leader$。

通常我们对轮数做归纳来证明这样的不变式。然而，为此我们需要一个预备不变式来说明少数几轮之后的情况。我们加入以下断言：

断言3.3.2　对$0 \leqslant r \leqslant n-1$,在$r$轮之后，$send_{i_{max}+r} = u_{max}$。

29　(回想一下，加法是模n加法。)这个断言说明$send$组件的最大值出现在环中与i_{max}距离为r的位置。

对r做归纳就可以很简单地证明断言3.3.2。当$r = 0$时，初始化表明在0轮后，$send_{i_{max}} = u_{max}$成立，得证。归纳步基于这样的事实：除i_{max}之外的每个节点接受最大值并把它放到$send$分量中，因为u_{max}大于其他所有的值。

在证明了断言3.3.2后，我们用它的特殊情况$r = n-1$和对单一轮中情况的讨论来说明断言3.3.1。这里的关键在于进程i_{max}把u_{max}作为设置自己为领导者的一个信号。　□

引理3.3　除进程i_{max}之外没有其他进程输出“领导者”值。

证明　这只要证明其他进程总有$status=unknown$就足够了。它可以有助于声明一个更强的不变式。如果i和j是环中任意两个进程，而且$i \neq j$,则定义$[i, j)$为下标集$\{i, i+1, \ldots, j-1\}$，其中加法是模n的。也就是说，$[i, j)$是一个进程集合，它包含从i开始沿顺时针方向直到j的逆时针邻接节点的进程。以下的不变式表明，没有UID v能够到达处于i_{max}与v的初始位置i之间任一位置的$send$变量：

断言3.3.3　对于任意的r和任意的i, j，在r轮之后，如果$i \neq i_{max}$且$j \in [i_{max}, i)$,那么$send_j \neq u_i$。

同样，使用归纳法来证明以上断言也是很容易的。证明中用到的关键事实在于一个非最大值不会通过i_{max}。这是因为i_{max}把输入的值和u_{max}比较，而u_{max}比其他任何UID都要大。

最后，断言3.3.3可以用于说明只有进程i_{max}可以在消息中接收自己的UID，所以只有进程i_{max}可以输出“领导者”。 □

引理3.2和引理3.3一起说明了以下的结论：

定理3.4 *LCR*解决了领导者选举问题。

停止和“非领导者”输出 如上所述，从所有进程都达到一个停止状态这个意义上讲，*LCR*算法永远都无法完成其工作。可以像2.1节中描述的那样扩充每个进程，使它们都包括停止状态。然后就可以修改算法，让选出的领导者在网络中发送一条特殊的报告消息。任何收到这个报告消息的进程可以在传递该消息之后停止。这样的策略不仅可以允许进程停止，还可以让不是领 30
导者的进程输出“非领导者”。更进一步，在消息上如果附上领导者进程的下标，这个策略可以让所有参与的进程都输出领导者的身份。注意，让每个不是领导者的节点在看到一个UID比自己的大时立即输出“非领导者”是可能的；但是这并不能告知非领导者节点何时停止。

一般说来，停止是一个分布式算法应满足的重要属性，但并不都是像本例一样容易做到。

复杂度分析 直到宣布一个领导者为止，基本*LCR*算法的时间复杂度是n轮，而通信复杂度在最坏的情况下是$O(n^2)$。在带停止的算法版本中，时间复杂度度是$2n$，但是通信复杂度还是$O(n^2)$。停止和“非领导者”声明所需的额外时间是n轮，通信所需要的额外消息是n个。

变形 以上两部分描述和分析了一个基本的算法变形，即从只有领导者进程产生输出且没有进程停止的任意一种领导者选举算法，到一个领导者进程和非领导者进程都有输出并且所有进程都停止的算法。获取额外输出和停止的额外开销就是n轮和n个消息。对于其他任意的假设组合，这个变形都成立。

可变的开始时间 注意在具有可变的开始时间的同步模型中，*LCR*算法无需改变。参看2.1节中关于这种模型的描述。

打破对称性 在一个环中选举领导者这个问题中，关键的难点是打破对称性。在分布式系统中需要解决许多其他问题，包括资源分配问题（见第10~11和20章）和一致性问题（见第5~7、12、21和25章），打破对称性也是其中一个重要部分。

3.4 通信复杂度为$O(n \log n)$的算法

虽然*LCR*算法的时间复杂度比较低，但是用到的消息的数目比较大，共有$O(n^2)$。这看起来好像不是很重要，因为在任一时刻任一链路中的消息不会多于一个。但是，在第2章中，我们已经讨论了为什么消息条数是一个值得注意的应该越少越好的衡量标准；这是因为对于许多并 31
发运行的分布式算法，其总的通信负载可能导致网络的拥塞。在本节中，我们将要介绍一个算法，它把通信复杂度降为$O(n\log n)$。

第一个将最坏情况下的复杂度降到$O(n\log n)$的算法由Hirschberg和Sinclair提出，所以称这个算法为*HS*算法。我们再次假定只有领导者需要执行输出，尽管在3.3节结尾中提到的变形暗示着这个限制并不重要。另外，我们假设环的规模是未知的，但是现在允许双向通信。

就像*LCR*算法所作的那样，*HS*算法也选出具有最大UID的进程。每个进程，不是像*LCR*算法那样把自己的UID在网络中一直传递下去，而是把它传到某个距离之外，然后反转回来，再回到原来的进程。它如此反复地逐步增大距离。*HS*算法如下所述。

***HS*算法（非形式化）：**

每个进程i在阶段0,1,2...中操作。在每个阶段l中，进程i向两个方向发出包含自己UIDu_i的“令牌”。这些令牌会前进2^l的距离，然后再回到自己原来的进程i（见图3-2）。如果两个令牌都

能够安全回来，那么进程i就会继续下一个阶段。但是，令牌可能不会安全返回。当一个u_i的令牌在外出方向前进时，任何在u_i的路径上的进程j都会把自己的UID u_j和u_i进行比较。如果$u_i<u_j$，那么进程j就会丢弃这个令牌；如果$u_i>u_j$，它就会继续传递u_i。如果$u_i=u_j$，则说明进程j在令牌反转之前就已经接收到自己的UID，所以进程j就会把自己选为领导者。

所有进程总是继续传递所有进入方向的令牌。

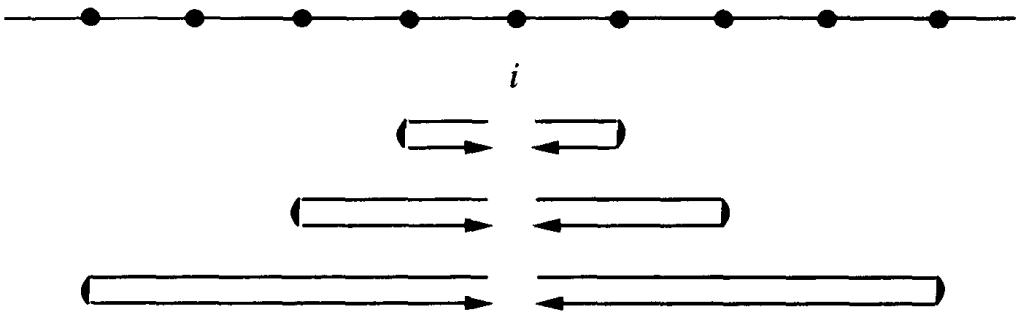


图3-2　*HS*算法中进程i处发起的连续令牌的轨迹

下面我们来更形式化地描述这个算法。这次，需要一些簿记（bookkeeping）以保证令牌
32 能沿着正确的轨迹运动。比如：令牌会带上一个标志来表示它们到底是外出还是进入。同样，令牌也会带上跳跃计数（hop count）来记录它们在外出方向上必须走的距离；这允许进程计算出应在何时反转令牌的方向。一旦算法以这样的方式进行形式化，那么我们就可以给出类似于*LCR*算法中的正确性证明。

***HS*算法（形式化）：**

消息字母表M是一个三元组的集合，三元组包括一个UID、一个$\{out, in\}$中的标志值和一个正整数“跳跃计数”。

对每一个i，状态$states_i$包括以下分量：

u，UID类型，初始值为i的UID
$send+$，包含M中的一个元素或*null*，初始值为由i的UID、*out*和1构成的三元组
$send-$，包含M中的一个元素或*null*，初始值为由i的UID、*out*和1构成的三元组
status，$\{unknown, leader\}$中的一个值，初始值为*unknown*
phase，一个非负整数，初始值为0

开始状态$start_i$的集合包含给定的初始化定义的单个状态。

对每一个i，消息产生函数$msgs_i$定义如下：

send the current value of $send+$ to process $i+1$
send the current value of $send-$ to process $i-1$

对每一个i，转移函数$trans_i$用下面的伪码定义：

$send+ := null$
$send- := null$
if the message from $i-1$ is (v, out, h) then
　case
　　$v > u$ and $h > 1$: $send+ := (v, out, h-1)$
　　$v > u$ and $h = 1$: $send- := (v, in, 1)$
　　$v = u$: $status := leader$
　endcase
if the message from $i+1$ is (v, out, h) then

```
        case
          v > u and h > 1: send− := (v, out, h − 1)
          v > u and h = 1: send+ := (v, in, 1)
          v = u: status := leader
        endcase
    if the message from i − 1 is (v, in, 1) and v ≠ u then
        send+ := (v, in, 1)
    if the message from i + 1 is (v, in, 1) and v ≠ u then
        send− := (v, in, 1)
    if the messages from i − 1 and i + 1 are both (u, in, 1) then
        phase := phase + 1
        send+ := (u, out, 2^phase)
        send− := (u, out, 2^phase)
```

33

像以前一样，前两行用来清除状态。接下来的两段代码用来处理外出方向的令牌：如果令牌的UID大于u_i，则根据跳跃计数决定继续传递或者折回；如果u_i到来，i就把自己选举为领导者。再下面的两段代码用来处理进入方向的令牌：它们只是简单地被继续传递（跳跃计数1用于进入的令牌）。如果进程i接收到了自己的两个返回令牌，那么它就会进入到下一个阶段。

复杂度分析 我们首先来分析通信复杂度。每个进程在阶段0发出一个令牌；因为令牌双向出去和返回，这总共需要$4n$条消息。对于$l > 0$，如果进程接收到了在$l-1$阶段的两个返回令牌，就会发送它在阶段l的令牌。如果在沿着环的两个方向且距离在2^{l-1}以内它都没有被另一个“进程”击败，就出现这种情况。这就暗示着在任意一组$2^{l-1}+1$个连续进程内，最多只有一个进程可以在阶段l发送令牌。因此在阶段l最多只有

$$\left\lfloor \frac{n}{2^{l-1}+1} \right\rfloor$$

个进程可以发送令牌。那么在阶段l发送的总的消息数就限于

$$4\left(2^l \cdot \left\lfloor \frac{n}{2^{l-1}+1} \right\rfloor\right) \leqslant 8n$$

这是因为阶段l的令牌能前进2^l的距离。而因子4源于这样的事实：令牌同时被双向发送（顺时针和逆时针），而且每一个外出的令牌必须反转并返回。

在选举出领导者和所有通信都停止之前要执行的总的阶段数最多是$1+\lceil \log n \rceil$（包括阶段0），所以总的消息数最多是$8n(1+\lceil \log n \rceil)$，即$O(n \log n)$，其常数因子接近于8。

这个算法的时间复杂度仅为$O(n)$。因为对于每个阶段l的时间是$2 \times 2^l = 2^{l+1}$(因为令牌要出去并返回)。最后阶段需要的时间为n——这是一个未完成阶段，其中令牌只向外发送。倒数第二个阶段是$l=\lceil \log n \rceil - 1$，它的时间复杂度至少和前面所有阶段的总和一样大，所以总的时间复杂度最多是 $2 \times 2^{\lceil \log n \rceil}$。如果$n$为2的幂，则总的时间复杂度最多为$3n$，否则是$5n$。其他的细节留做一道习题（习题3.6）。

34

可变的开始时间 *HS*算法在具有可变的开始时间的同步模型中不需修改。

3.5 非基于比较的算法

下面考虑是否可能以少于$O(n \log n)$的消息数来选出领导者。这个问题的答案是否定的，正如我们将会简短证明一个不可能性结果——$\Omega(n \log n)$的下界。但是这个结论只适用于那些使用比较操作来处理UID的算法（基于比较的算法在节3.6中定义）。

在本节中，我们允许UID为正整数，且允许对它们进行一般的数学操作。在这种情况下，我们给出两个算法，时间片（*TimeSlice*）算法和变速（*VariableSpeeds*）算法，它们都是消息数为$O(n)$的算法。这两个算法的存在说明了在一般情况下，下限低于$\Omega(n\log n)$是可能的。

3.5.1 时间片算法

第一个算法用到了一个很强的假设：环的大小n对所有进程来说是已知的，但只假定是单向通信。我们所说的时间片算法就在这样的设定下工作的。它选择具有最小UID的进程。

注意，这个算法比*LCR*和*HS*算法更深入地利用了同步性。也就是说，它在某些轮处利用了非到达消息（即到来的是*null*消息）来传递信息。

时间片算法：

在阶段1,2...中进行计算，每一个阶段都由n个连续的轮组成。每个阶段都有一个携带着特
35 殊UID的令牌一直在个环中流转。更特别地，在包含轮$(v-1)n+1$，...，vn的阶段v中，只有带着UID v的令牌才被允许流转。

如果存在UID为v的进程i，且在$(v-1)\times n+1$轮到达之前进程i没有收到任何非空消息，那么进程i就会把自己选举为领导者，并沿着环发送一个包含其UID的令牌。当这个令牌在传递的时候，所有其他进程都接收到它，这样它们就不会选举自己为领导者或在以后的阶段中创建令牌。

在这个算法中，最小的UID u_{min}最终会环绕整个路径，使得它的原始进程被选为领导者。在第$(u_{min}-1)n+1$轮之前，不会有任何消息发送，并且在$u_{min}\times n$之后，也不会发送任何消息。总共发送的消息的数量是n。如果想选出具有最大UID而不是最小UID的进程，那么我们可以让具有最小UID的进程在被发现后发送一个特殊的消息来确定最大UID的进程。通信复杂度还是为$O(n)$。

时间片算法的好处就是消息的总数是n个。但不幸的是，时间复杂度大约是$n\times u_{min}$，这在固定规模的环中也是一个无界数。这个时间复杂度限制了这个算法的实际应用。它只能用于那些小的环网络中，其中的UID都是很小的正整数。

3.5.2 变速算法

时间片算法表明，在进程知道环的规模为n的那些环中，消息数为$O(n)$就足够了。但是如果n是未知的呢？在这种情况下，同样存在消息数为$O(n)$的算法，我们称之变速算法。这个算法只使用单向的通信。

但是，变速算法的时间复杂度甚至比时间片算法还要差，为$O(n\times 2^{u_{min}})$。当然，没有人会在实际中运用这样的算法。我们称变速算法为一个反例算法。一个反例算法的主要目的是说明某个推想的不可能性结果是错误的。这种算法本身并无意义——不实用，从数学的观点来看也不精确。但是，它却可以说明一个不可能性结果不成立。

下面就是这个算法。

变速算法：

每个进程i创建一个令牌，令牌带着发起进程i的UID u_i在网络中流动。不同令牌的流动速
36 度是不一样的。一个带着UID v的令牌的速度为每2^v轮传递一条消息，也就是说每个进程在接受这个令牌以后需要等到2^v轮之后才能将它发出。

同时，每个进程都会保存自己见到的最小UID，并丢弃任何包含比最小UID更大的UID的令牌。

如果令牌返回到创建进程，那么这个进程就被选为领导者。

和时间片算法类似，变速算法保证了拥有最小UID的进程会被选出。

复杂度分析 变速算法保证了最小UID u_{min}的令牌会流过整个网络，而次小的令牌会流过大约一半的网络，第三小的令牌流过大约四分之一的网络，一般来说，按从小到大排序UID，排在第k位的令牌会流过 $\frac{1}{2^{k-1}}$ 的路程。直到领导者被选出为止，最小UID的令牌使用的消息数比所有其他令牌用到的消息数的和还要多。因为u_{min}正好使用n个消息，所以总的消息数小于$2n$。

但是，注意当u_{min}流过了整个网络的时候，所有的节点都会知道这个值，并会拒绝发出其他令牌。在这个算法中，$2n$是发送消息数目的上限。

如上所提到的，时间复杂度是$n \times 2^{u_{min}}$，因为每个节点都会将UID为u_{min}的令牌延迟$2^{u_{min}}$个时间单位。

可变的开始时间 不像*LCR*和*HS*算法，变速算法不能像具有可变开始时间的同步模型中的那样使用。但是可以通过修改算法来实现：

改进的变速算法：

如果一个进程在接收到任一普通（非空）消息之前（必须正好在提前一轮中)接收到一条唤醒消息，则定义它为*开始者*（starter）。

每个开始者i发出一个带着自己的UID u_i的令牌在网络中流动；非开始者不能创建令牌。开始时，令牌走得很“快”，每轮传递一次，它通过所有被令牌的到来而唤醒的非开始者进程，直到首次到达一个开始者（可以为不同的开始者，也可以为i自己）。当令牌到达开始者以后还会继续传递，但是速度减慢到每2^{u_i}轮传递一次。

37

同时，每个进程都保存所见到的最小UID的开始者，并且丢弃任何比这个UID大的进程的令牌。如果令牌回到原始进程，那么该进程就会被选为领导者。

这个改进的变速算法保证了具有最小UID的开始者进程能够被选出。用$i_{min\text{-}start}$来表示这个进程。

复杂度分析 我们分三类来对消息计数。

1）与开始时令牌的快速传递过程相关的消息。共有n条。

2）直到$i_{min\text{-}start}$的令牌首次到达一个开始者为止与令牌的慢速传递过程相关的消息。从第一个进程被唤醒时开始，这至多需要n轮。在这段时间中，一个带着UID为v的令牌可能最多用到 $\frac{n}{2^v}$ 条消息，一共最多 $\sum_{v=1}^{n} \frac{n}{2^v} < n$ 条消息。

3）在$i_{min\text{-}start}$的令牌首次到达开始者之后与令牌的慢速传递过程相关的消息。这里的分析和基本变速算法差不多。在获胜令牌走完整个路程时，按从小到大排序UID，排在第k位的令牌只走了至多 $\frac{1}{2^{k-1}}$ 的路程。所以，到选出领导者时，发送的总消息少于$2n$。但是在获胜令牌走完整个路程时，所有的节点都会知道它的值，并会拒绝发出其他令牌，所以$2n$是消息数的上限。所以总的通信复杂度最大为$4n$。

时间复杂度为$n+n \times 2^{u_{min\text{-}start}}$。

3.6 基于比较的算法的下界

到现在为止，我们已经列出了好几种在同步环中的领导者选举算法。*LCR*和*HS*算法都是基于比较的，并且后者的通信复杂度为$O(n\log n)$，时间复杂度$O(n)$。另一方面，时间片和变速算法不是基于比较的，消息数为$O(n)$，但是需要很长的运行时间。在本节中，我们证明基于比较的算法的消息数下界为$\Omega(n\log n)$。即使我们假设通信是双向的而且环的大小n是已知的，这个下界也不会改变。在下一节中，我们将针对具有受限时间复杂度的非基于比较的算法，证明一个类似的消息数下界$\Omega(n\log n)$。

38

本节的结果是基于打破对称性的难度。回想一下定理3.1中的不可能性结果，因为对称性，如果缺少如UID这样的区别信息，就不可能选出一个领导者。下述证明的主要思想是，即使存在UID，也有可能发生某些对称性。在这种情况下，UID允许打破对称性，但是需要大量通信来实现。

在整章中，我们假设所有的进程除了它们的UID之外都是相同的。所以，除了那些包含进程UID的分量之外，进程的开始状态都是相同的。总之，我们并没有强行限制消息产生和状态迁移函数应该怎样使用UID的信息。

在本章的剩余部分中（本节和下一节），我们假设对于每个UID，只有一个开始状态包含这个UID（如定理3.1的证明中所示，此假设不失一般性）。这样假设的好处就是，它意味着系统（具有固定的UID指派）恰好只有一次运行。

一个基于比较的算法会遵守一些额外的约束，下面用不太形式化的定义来表示。在一个基于UID的环算法中，如果进程处理UID的惟一方法是拷贝、在消息中发送和接送以及用{<, =, >}来比较它们，那么算法就是基于比较的。

举例来说，这样的定义允许一个进程储存它所碰到的各种各样的UID，并且可能与其他的信息组合起来在消息中发送它们。进程还可以比较这些存储的UID，并用比较的结果在消息产生和状态迁移函数中选择。比如，这些选择可以包括是否要向它的每个相邻节点发送消息、是否要选举自己作为领导者以及是否需要保存UID，等等。一个重要的事实就是，一个进程的所有活动都只是依赖于它所面对的UID的相对次序，而不是这些UID的具体值。

下面的形式化表示用于阐述即使具有UID，对称性还是可以存在。设$U=(u_1,u_2,\ldots,u_k)$和$V=(v_1,v_2,\ldots,v_k)$为两个UID序列，其长度都为k。对于所有的i，j，$1\leqslant i,\ j\leqslant k$，我们称$U$次序等价
39 于V当且仅当$v_i\leqslant v_j$时有$u_i\leqslant u_j$。

例 3.6.1　次序等价

如果UID集是按正常排序的自然数，则序列（5，3，7，0），（4，2，6，1）和（5，3，6，1）都是次序等价的。

注意，当且仅当对应UID的相对次序相同时，两个UID序列才能认为是次序等价的。下面有两个技术性的定义。如果在运行的一轮中至少有一条（非空）消息被发送，那么这一轮就被称为*活动的*。在规模为n的环R中，进程i的k邻接节点，$0\leqslant k<\lfloor n/2\rfloor$被定义为由$2k+1$个进程$i-k$，…，$i+k$组成，即那些和进程$i$距离在$k$之内的进程（包括$i$本身）。

最后我们需要定义的是，除了进程包含的特定UID之外，进程状态是相同的。如果以下条件被满足，我们就称两个进程状态s和t对于UID序列$U=(u_1,u_2,\ldots,u_k)$和$V=(v_1,v_2,\ldots,v_k)$是*对应的*：s中的所有UID都是从U中选出，t中的所有UID都是从V中选出，且对于所有的i，$1\leqslant i\leqslant k$，将s中的u_i用t中的v_i代替后，t和s是相同的。同样可以给出*对应消息*的定义。

现在我们可以来证明关于下限的重要引理3.5了。它说明具有次序等价的k阶邻接节点的不同进程，会表现出基本相同的行为，除非信息有机会从k阶邻接节点以外向这些进程传播。

引理3.5　设A是一个基于比较的算法，它在一个大小为n的环R中执行。令k为一个整数，$0\leqslant k<\lfloor n/2\rfloor$。令$i$和$j$是$A$中的两个进程，在它们的$k$阶邻接节点中具有次序等价的UID序列。那么在至多$k$个活动轮之内的任一点，进程$i$和$j$对于其$k$阶邻接节点中的UID序列是对应的。

例 3.6.2　对应状态

假设进程i的3阶邻接节点中的UID的序列为（1，6，3，8，4，10，7）（进程i的UID为8），进程j的3阶邻接节点的UID的序列为（4，10，7，12，9，13，11）（进程j的UID为12）。因为这两个序列是次序等价的，按照引理3.5，只要已发生的活动的轮数不大于3，进程i和j对于其3

阶邻接节点处于对应状态。粗略地说，其原因在于如果只有3个活动的轮，次序等价的3阶邻接节点以外的信息就没有机会到达i和j。

（引理3.5的）证明 不失一般性，我们可以假设$i \neq j$。对运行中已经执行的轮数r做归纳。对于每个r，我们证明引理对所有的k成立。 40

基础：$r=0$。从基于比较的算法的定义来看，i和j的初始化状态除了各自的UID外都是相同的，所以就它们的k阶邻接节点（对任意的k）而言，它们的初始状态是对应的。

归纳步：假设引理对所有的$r'<r$都成立。固定k，使得i和j有次序等价的k阶邻接节点，并假设最初的r轮中包含最多k个活动轮。

如果i和j都没有在r轮接收到消息，那么根据归纳假设（对$r-1$和k）得知，i和j正好在$r-1$轮后对于其k阶邻接节点是对应的。因为i和j没有新的输入，因此它们会做对应的状态转移，并在第r轮后处于对应状态。

假设i或j在r轮时接收到了消息。那么r轮就是活动的，所以开始的$r-1$轮最多包括了$k-1$个活动轮。注意i和j拥有次序等价的$k-1$阶邻接节点，这对于$i-1$和$j-1$，以及$i+1$和$j+1$也都成立。所以，根据归纳假设（对$r-1$，$k-1$）得知，进程i和j在$r-1$轮后对于其$(k-1)$阶邻接节点是对应的。类似地，这对于$i-1$和$j-1$，以及$i+1$和$j+1$也是成立的。

我们继续对不同情况进行分析。

1）在第r轮，$i-1$和$i+1$都没有发送消息给i。

因为$i-1$和$j-1$在$r-1$轮以后处于对应状态，并且$i+1$和$j+1$也一样，所以$j-1$和$j+1$都没有在第r轮中发送消息给j。这与i或者j在第r轮接收到消息的假设相矛盾。

2）在第r轮，$i-1$给i发送消息，但是$i+1$没有。

那么，既然$i-1$和$j-1$在第$r-1$轮以后处于对应状态，$j-1$也在第r轮给j发送了消息，而且相对于$i-1$和$j-1$的$(k-1)$阶邻接节点而言，这个消息与$i-1$发送给i的消息是对应的，因此这两个消息对于i和j的k阶邻接节点也是对应的。鉴于类似的理由，$j+1$在第r轮没有给j发消息。因为i和j在$r-1$轮后处于对应状态，并且接收到了对应消息，所以现在它们对于k阶邻接节点依然是对应的。

3）在第r轮，$i+1$给i发送消息，但是$i-1$没有。

类似于对前面情况的分析。

4）在第r轮，$i-1$和$i+1$都给i发送了消息。 41

同理。 □

引理3.5告诉我们，如果存在很大的次序等价的邻接节点，那么需要有许多活动轮来打破对称性。现在我们定义具有特殊属性的环，它们有许多不同大小的次序等价的邻接节点。设c为常数，$0 \leqslant c \leqslant 1$，并设$R$是规模为$n$的环。如果对每一个$l$，$\sqrt{n} \leqslant l \leqslant n$，以及对于$R$中长度为$l$的每一段$S$，在$R$中至少存在$\left|\frac{cn}{l}\right|$个次序等价于$S$的段（包括$S$本身），那么，我们称$R$是$c$对称的[⊖]。

如果n是2的幂，那么构造一个1/2对称的环是很容易的。特别地，我们定义大小为n的*位反转环*如下：假设$n=2^k$，那么，给每一个进程赋值$[0, n-1]$之间的一个整数，使其k位二进制表示恰好是i的k位二进制表示的反转（我们用0^k作为n的k位二进制表示，使0与n相同）。

例 3.6.3 位反转环

对于$n=8$，可得$k=3$，且赋值如图3-3所示。

⊖ 可以忽略平方根的下限条件——这仅是个技术问题。

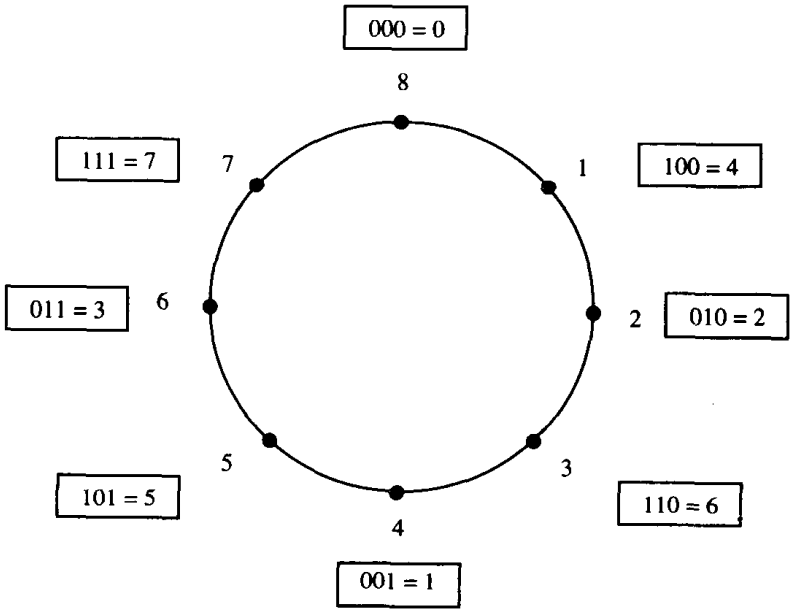


图3-3　大小为8的位反转环

引理3.6　任意一个位反转环都是1/2对称的。

证明　留做一道习题[⊖]。　□

对不是2的幂的n值，也总是存在c对称环，不过一般情况下需要一个较小的常数c。

定理3.7　存在一个常数c，使得对于所有$n \in \mathbb{N}^+$，存在大小为n的c对称环。

定理3.7的证明需要一个很复杂的递归构造[⊜]。所需的环不可能通过简单的方法产生，比如从一个2的次小次幂的位反转环出发并加入额外的进程；这些额外的进程会破坏对称性。

所以我们可以假设，对任意n，存在一个大小为n的c对称环R。接下来的引理说明，如果在
42 这样的环中选出领导者，那么需要许多活动轮。

引理3.8　令A是一个在大小为n的c对称环中执行的基于比较的算法，并假设A选出了一个领导者。设k是满足$\sqrt{n} \leqslant 2k+1$和$\left|\frac{cn}{2k+1}\right| \geqslant 2$的整数，则$A$有多于$k$个活动轮。

证明　采用反证法。假设A在最多k个活动轮中选出了领导者进程i。令S为i的k阶邻接节点；S是长度为$2k+1$的一段。因为环是c对称性的，所以在环网中至少有$\left|\frac{cn}{2k+1}\right| \geqslant 2$个片段与$S$是次序等价的，包括$S$本身。因此至少有一个其他片段与$S$次序等价；令$j$为那个片段的中点。现在，根据引理3.5，直到选出点为止，i和j在整个运行中都处于等价状态。我们可以得到结论j也被选出了。矛盾。　□

现在我们来证明下界。

定理3.9　令A是一个在大小为n的环中选出一个领导者的基于比较的算法。则存在A的一个运行，
43 其中在领导者被选出之前有$\Omega(n \log n)$条消息被发送[⊛]。

⊖　注意，对位反转环不需要平方根的下限条件。

⊜　这里有平方根下限的问题。

⊛　表达式$\Omega(n \log n)$隐藏了一个与n无关的固定常数。

证明 固定一个由定理3.7断定存在的常数c，并使用定理3.7来获得一个大小为n的c对称环R。我们考虑在R中的算法运行。

定义$k=\left|\frac{cn-2}{4}\right|$，那么$\sqrt{n}\leqslant 2k+1$（假设$n$足够大），且$\left|\frac{cn}{2k+1}\right|\geqslant 2$。根据引理3.8，有多于$k$个活动轮，亦即至少有$k+1$个活动轮。

考虑第r个活动轮，其中$\sqrt{n}+1\leqslant r\leqslant k+1$。因为这是活动轮，所以某个进程$i$在$r$轮中发送了消息。令$S$为$i$的（$r-1$）阶邻接节点。因为$R$是$c$对称的，所以至少有$\left|\frac{cn}{2r-1}\right|$个片段和$S$等价。根据引理3.5，在第$r$个活动轮即将执行的时候，所有这些片段的中点都处于对应状态，所以它们都发送消息。

现在令$r_1=\lceil\sqrt{n}\rceil+1$和$r_2=k+1=\left\lfloor\frac{cn-2}{4}\right\rfloor+1$。以上的讨论说明了消息总数至少为

$$\sum_{r=r_1}^{r_2}\left\lfloor\frac{cn}{2r-1}\right\rfloor\geq\sum_{r=r_1}^{r_2}\frac{cn}{2r-1}-r_2\ 。$$

第二项为$O(n)$，因此表明第一项是$\Omega(n\log n)$就足够了。

我们有

$$\begin{aligned}\sum_{r=r_1}^{r_2}\frac{cn}{2r-1}&=\Omega\left(n\sum_{r=r_1}^{r_2}\frac{1}{r}\right)\\&=\Omega\left(n\left(\ln r_2-\ln r_1\right)\right)\end{aligned}$$

由和的整数近似，

$$\begin{aligned}&=\Omega\left(n\left(\ln\left(\left\lfloor\frac{cn-2}{4}\right\rfloor+1\right)-\ln(\lceil\sqrt{n}\rceil+1)\right)\right)\\&=\Omega(n\log n)\end{aligned}$$

正是所求。 □

3.7 非基于比较的算法的下界*

我们能够描述非基于比较的算法的消息数的下限吗？虽然$\Omega(n\log n)$的界限可以被打破，但是可能表明这样做只能以大的时间复杂度为代价。举个例子，假设选出领导者的时间界限为t。如果标识符空间中UID的总数足够大，比如，大于某个快速增长的函数$f(n,t)$，那么就存在一个标识符的子集U，其上的算法可能表现得像基于比较的算法，至少在t轮中如此。这意味着针对比较的下限也适用于使用U中标识符的时间受限算法。 44

我们会给出更多细节，但表示形式仍然只是一个大概。用*Ramsey定理*定义一个快速增长的函数$f(n,t)$，Ramsey定理是一种扩展的鸽巢原理。在定理的叙述中，一个n维子集就是有n个元素的子集，一个着色就是给每一个集合指定一种颜色。

定理3.10（Ramsey 定理） 对于所有的整数n、m和c，总存在一个整数$g(n,m,c)$满足以下性质：对于每个大小至少为$g(n,m,c)$的集合S，当使用最多c种颜色来对S的n维子集进行着色时，总有一个大小为m的S的子集C，C的所有n维子集的颜色相同。

作为开始，我们把每个算法都放入*规范式*（*normal form*），其中每个状态都以LISP的S表

达式来简单地记录初始的UID和所有曾经收到的消息，而且每个非空消息都包含其发送者的完整状态。然后将其中的某些S表达式指定为选举状态，这种状态表示进程已被选为领导者。如果原来的算法是一个正确的领导者选举算法，那么新的（带有修改后的输出约定）也是，通信复杂度也是一样的。

下界定理描述如下。

定理3.11　对于所有的整数n和t，存在满足如下性质的整数$f(n, t)$：令A为任意（不必基于比较）的算法，该算法于时间t内在大小为n的环中选出领导者，并且使用了大小至少为$f(n, t)$的UID空间，则存在一个A的运行，其中领导者被选出之前发送的消息数为$\Omega(n \log n)$。

证明概要　固定n和t。不失一般性，我们只考虑规范式中的算法。因为算法中只涉及n个进程且只执行t轮，所以所有出现的S表达式都有最多n个不同的参数和最多t层括号深度。

现在对于每个算法A，我们定义一个UID的n维集合（大小为n的集合）上的等价关系$\equiv_A$。粗略地说，如果算法A在两个n维集合上的行为是一样的，那么就称这两个集合是等价的。更精
45 确地，如果V和V'是UID的两个n维子集，并且对于V上每个深度最多为t的S表达式，在算法A中，V'上对应的S表达式（通过用V'中的元素代替V中排位相同的元素得到）产生相同的决定，即是否在每个方向发送一个消息和是否该进程已被选为领导者，则称$V \equiv_A V'$。

因为在等价关系定义中，S表达式最多有n个参数和t的深度，所以$\equiv_A$等价类的数目有限；事实上，有一个不依赖于算法A的等价类数的上界，它只依赖于n和t。令$c(n, t)$为这个上界。

现在固定算法A。我们来描述一个给UID的n维子集着色的方法，这样就可以应用Ramsey定理。也就是说，我们把n维集合的每一个$\equiv_A$等价类与一种颜色相联系，类中所有的n维子集都用那种颜色着色。

现在我们定义$f(n, t) = g(n, 2n, c(n, t))$，其中$g$是定理3.10中的函数，并且考虑任一包括至少$f(n, t)$个标识符的UID空间。定理3.10暗示着存在一个含有至少$2n$个元素的UID空间的子集C，使得C的所有n维子集都着同一种颜色。令U为由C中n个最小元素组成的集合。

当从U中选择UID时，我们可以声称在t轮内，算法的行为完全像一个基于比较的算法。每个进程做出的有关是否要发送消息或者要让哪个进程成为领导者的决定只依赖于在当前状态下所含参数的相对次序。为了弄清楚其原因，我们固定U的任意两个子集W和W'，大小都为m。假设S是一个深度最多为t的S表达式，其UID从W中选取，而S'是W'上对应的S表达式。那么W和W'通过包括进C中最大的$n-m$个元素，就可以被扩展成为V和V'，每个大小都是n。因为V和V'有相同的着色，因此这两个S表达式就会对于是否在每个方向发送消息和选择某个进程为领导者作出相同决定。

由于算法在t轮内的行为完全像一个基于比较的算法，因此当从U中选择UID时，定理3.9就产生了这个下界。　□

3.8　参考文献注释

3.2节中的不可能性结论是这个领域内早期理论的一部分，Angluin[13]的论文中给出了这
46 种结论对于一个不同模型的一种版本。*LCR*算法来自于Le Lann[191]，并由Chang和Roberts[71]做了优化。*HS*算法来自于Hirschberg和Sinclair[156]。

对于上界$O(n \log n)$的常数因子已经有了一系列的改进，最后大概接近于$1.271n\log n+O(n)$，由Higham和Przytycka[155]提出，然而这个边界只适用于单向环的情况。Peterson[239]与Dolev、Klawe和Rodeh[97]已经给出了单向情况下的$O(n \log n)$的算法。

时间片算法似乎来自民间，但是它和MIT令牌环网络系统中使用的领导者选择策略相似。变速算法由Frederickson和Lynch[127]提出，同时还有Vitanyi[282]。

下限的结论，包括基于比较的和非基于比较的算法，都是由Frederickson和Lynch[127]提出。另外一种c对称环的构造由Attiya、Snir和Warmuth[27]提出，Ramsey定理是一个组合理论的标准结论，并且在Berge[47]的图论书中有描述。

Attiya、Snir和Warmuth[27]写的论文包括了在同步网络中计算能力受限的一些结论，并用到了类似3.6节中的证明方法。

3.9 习题

3.1 对*LCR*算法的正确性给出详细的归纳证明。

3.2 对*LCR*算法，

a）给出一个发送消息数为$\Omega(n^2)$的UID赋值方案。

b）给出一个发送消息数为$O(n)$的UID赋值方案。

c）证明发送消息的平均数为$O(n\log n)$，其中对于环上所有可能的进程排序，该平均数都有效，假设所有可能的进程排序的概率相等。

3.3 修改*LCR*算法，使得所有的非领导者进程输出“非领导者”，并且最终停止所有的进程。用描述*LCR*算法的相同代码形式描述修改后的算法。

3.4 证明*LCR*算法在允许可变开始时间的同步模型中仍然适用（可能需要对代码稍做修改）。

3.5 详细证明*HS*领导者选举算法的正确性，利用*LCR*算法中用到的不变式断言。 47

3.6 证明*HS*算法在允许可变开始时间的同步模型中仍然适用（可能需要对代码稍做修改）。

3.7 假设对*HS*领导者选举算法做了修改，使用k的连续次幂，而不是2的幂，作为路径长度（$k>2$）。用类似于书中分析原始的*HS*算法的方法，分析新算法的时间和通信复杂度，并将结果与原算法进行比较。

3.8 修改*HS*算法，使进程只向一个方向发送令牌。

a）证明这样对书中算法最直接的修改不会产生$O(n \log n)$的通信复杂度。通信复杂度的上限是多少?

b）在算法中稍微增加一些技巧，使其保证$O(n \log n)$的复杂度限制。

3.9 设计一个单向的领导者选举算法，使其可在大小未知的环中工作，最坏情况下的消息数为$O(n \log n)$。算法只对UID作比较操作。

3.10 给出描述时间片领导者选举算法的状态机代码。

3.11 描述一个时间片断算法的变形，它在每一阶段允许k个而不是1个UID在环中传递，通过额外的消息开销来节约时间。证明算法的正确性，并分析其复杂度。

3.12 给出描述变速领导者选举算法的状态机代码。

3.13 证明未修改的变速算法在进程唤醒时间不同的情况下，不一定有$O(n)$的通信复杂度。

3.14 证明在大小为n的环中，在最坏情况下，选举领导者所需轮数的最佳下限。请准确描述你的假设。 48

3.15 直观地表示出$n = 16$的位反转环。

3.16 证明对任意的$k \in \mathbb{N}^+$，大小为$n=2^k$的位反转环是1/2对称环。

3.17 对某一$c > 0$，设计一个非2的幂的c对称环。

3.18 考虑大小为n的同步环中的领导者选举问题，其中所有的进程都知道n，但进程没有UID。

设计一个随机化领导者选举算法，也就是说，算法中的进程除了遵循确定的代码外，可以作出随机的选择。请准确描述算法满足的属性。例如，它能否绝对保证选出一个惟一的领导者，还是有失败的可能？算法期望的时间和通信复杂度是多少？

3.19 考虑一个双向的同步环，大小n未知，其中的进程具备UID。给出一个基于比较的算法所
49 要求的消息数的上限和下限，在这个算法中，所有的进程都计算n模2。

第4章　一般同步网络中的算法

在第3章中，我们给出了对简单同步网络——单向环和双向环——中的领导者选举问题的算法及其下限。在这一章中，我们要考虑在更大类同步网络中的更多问题。特别地，我们将在基于任意图和有向图的网络中，讨论领导者选举、广度优先搜索（*BFS*）、寻找最短路径、寻找最小生成树（*MIS*）和寻找最大独立集（*MIS*）的算法。

当需要选举一个进程来负责整个网络的计算时，就出现了领导者选举问题。而广度优先搜索、寻找最短路径和最小生成树这些问题的动机来源于构建适合于支持有效通信的结构这一需要。寻找最大独立集的问题源于网络资源分配的需要（在第15章的异步网络环境中，我们还会讨论这些问题）。

在这一章中，我们考虑一种任意的强连通网络有向图$G = (V, E)$，它有n个节点（有时我们只考虑所有边都是双向边的情况，即无向图的情况）。像通常的同步网络系统一样，我们假设进程仅仅通过图的有向边通信。为了给节点命名，给它们指派下标1, ..., n，但是与环的下标不同，这些下标与图中节点位置并无特殊联系。进程并不知道自己的下标，也不知道邻接节点的下标，但是能够通过本地名字来指称邻接节点。我们假定，如果进程i的出向和入向邻接节点都为同一进程j，那么i就知道这两个进程是相同的。

4.1　一般网络中的领导者选举

我们重新考虑领导者选举问题，这一次的网络基于任意的强连通有向图。

4.1.1　问题

我们假设进程都有惟一的标识符（UID），UID从某个标识符的全排序空间中选择；在网络中，每个进程的UID都和其他的不一样，但是没有规定哪个UID必须出现。和在第3章中一样，这里的要求是，最终有且只有一个进程将自己选为领导者，把自己状态中一个特定的状态分量的值改为“领导者”。和第3章一样，这里有以下几个问题：

1）可能要求那些非领导者的进程通过把特定的状态分量的值改为“非领导者”，最终输出自己不是领导者的事实。

2）进程知道或者不知道节点数n和直径$diam$的值；或者已知这些量值的一个上界。

4.1.2　简单的洪泛算法

我们给出一个简单算法用于让领导者和非领导者识别自己。算法要求进程都知道直径$diam$。算法把最大的UID在整个网络中像水流一样传播，所以我们称它为*洪泛最大值（FloodMax）算法*。

洪泛最大值算法（非形式化）：

每个进程都保持有一个最大UID的记录（最初是自己的UID）。在每轮中，每个进程都要在自己的所有出向边上传播最大的UID。在$diam$轮后，如果得到的最大值是进程自己的UID，那么进程就把自己选为领导者，否则就为非领导者。

进程i的代码如下。

洪泛最大值算法（形式化）

消息字母表是UID的集合。

$states_i$ consists of components:
u, a UID, initially i's UID
max-uid, a UID, initially i's UID
status $\in$ {*unknown*, *leader*, *non-leader*}, initially *unknown*
rounds, an integer, initially 0

$msgs_i$:
if *rounds* < *diam* then
　send *max-uid* to all $j \in$ *out-nbrs*

$trans_i$:
rounds := *rounds* + 1
let U be the set of UIDs that arrive from processes in *in-nbrs*
max-uid := max({*max-uid*} $\cup U$)
if *rounds* = *diam* then
　if *max-uid* = u then *status* := *leader*
　else *status* := *non-leader*

容易看出，洪泛最大值算法选出具有最大UID的进程。更确切地讲，我们定义i_{max}为具有最大UID的进程的下标，u_{max}为那个最大的UID。我们有如下定理：

定理4.1　在洪泛最大值算法中，在*diam*轮内，进程i_{max}输出自己为领导者，其他的进程输出自己为非领导者。

证明　很容易证明以下断言：

断言4.1.1　在*diam*轮后，$status_{i_{max}} = leader$，并且对于每一个不等于$i_{max}$的$j$，$status_j = non\text{-}leader$。

对于断言4.1.1证明的关键是，在r轮过后，沿着G中的有向路径，最大UID已经到达了那些与i_{max}的距离在r之内的所有进程。这个条件可由下面的不变式断言得到：

断言4.1.2　对于每一个j和$0 \leqslant r \leqslant diam$，在$r$轮后，如果从$i_{max}$到$j$的距离不超过$r$，那么$max\text{-}uid_j = u_{max}$。

尤其是从图的直径的定义来看，断言4.1.2暗示着每个进程在*diam*轮后就会有最大的UID。为了证明断言4.1.2，我们还需要有以下辅助的不变式断言。

断言4.1.3　对于每个r和j，在r轮后，$rounds_j = r$。

53

断言4.1.4　对于每个r和j，在r轮后，$max\text{-}uid_j \leqslant u_{max}$。

将断言4.1.2、断言4.1.3和断言4.1.4中的r定为$diam - 1$，再加上对发生第*diam*轮的情况的证明，就能证明断言4.1.1。　□

洪泛最大值算法可以认为是在3.3节中*LCR*算法的推广，因为*LCR*算法也是把最大的UID值在整个环形网络中传递。但是，注意*LCR*算法并不需要任何与网络有关的特殊信息，如网络直径等。在*LCR*中，一个进程在消息中接收到自己的UID之后，就会把自己选为领导者，而洪泛最大值算法在特定轮数之后才把自己选为领导者。这样特殊的策略只在环形网络中才能成立，而不适应于一般的有向图。

复杂性分析　很容易可以看出，直到领导者被选出（并且其他进知道它们不是领导者），所需时间为*diam*轮。消息的数量为$diam \cdot |E|$，其中$|E|$是有向图中有向边的数目，这是因为在开始的

*diam*轮中，每轮都要在每条有向边上发送一个消息。

直径的上界。注意，如果所有进程都知道一个直径的上界*d*而不知道直径本身，算法也可以有效地工作。此时算法复杂度将随*d*而不是*diam*而增长。

4.1.3 降低通信复杂度

在多数情况下，有一个简单的优化[⊖]可以用来减少通信复杂度，虽然它不能降低最坏情况下复杂度的数量级。也就是说，进程可以只在它们初次知道*max-uid*的时候发送消息，而不是在每轮中都要发送。我们把这个算法叫做*优化洪泛最大值算法*(*OptFloodMax*)。对洪泛最大值算法的代码修改如下：

优化洪泛最大值算法：

$states_i$ has an additional component:

new-info, a Boolean, initially *true*

***$msgs_i$*:**

if *rounds* < *diam* and *new-info* = *true* then
  send *max-uid* to all $j \in$ *out-nbrs*

***$trans_i$*:**

rounds := *rounds* + 1
let U be the set of UIDs that arrive from processes in *in-nbrs*
if max (U) > *max-uid* then *new-info* := *true* else *new-info* := *false*
max-uid := max $(\{max\text{-}uid\} \cup U)$
if *rounds* = *diam* then
  if *max-uid* = *u* then *status* := *leader* else *status* := *non-leader*

很容易相信这个改进算法是正确的，然而怎样形式化地证明它呢？一个方法就是使用在*洪泛最大值算法*中用过的不变式断言。但这涉及对许多已做工作的重复。与从头开始不同，我们给出了一个证明，基于优化洪泛最大值算法和洪泛最大值算法在形式化上的相关性。这个简单例子表明，可以使用模拟的方法来证明分布式算法的正确性。

定理4.2 在优化洪泛最大值算法中，在*diam*轮内，进程i_{max}输出自己为领导者，其他的进程输出自己为非领导者。

证明 和断言4.1.1相似，我们可以很容易地证明以下断言。

断言4.1.5 在*diam*轮后，对于每一个$j \neq i_{max}$，$status_{i_{max}}$=*leader*和$status_j = non-leader$。

我们现在先证明一个预备的不变式，它表明只要进程预期在下一轮中需要发送新信息，进程的*new-info*标志就设置为*true*。特别地，如果*i*的任一出向邻接节点不知道这样一个UID，该UID不小于*i*已知的最大UID，那么*i*的*new-info*标志就必定为*true*。

断言4.1.6 对任意的r，$0 \leqslant r \leqslant diam$，和任意的$i$，$j$，其中$j \in out\text{-}nbrs_i$，在$r$轮之后，如果$max\text{-}uid_j < max\text{-}uid_i$，则$new-info_i = true$。

断言4.1.6可以通过对r的归纳来证明。基本情况是$r = 0$为真，因为所有的*new-info*标志都被初始化成为真。下一步，我们考虑进程*i*和*j*，$j \in out-nbrs_i$。如果$max-uid_i$在r轮中变大，那么$new-info_i$就设置为真，满足结论。另一个方面，如果$max-uid_i$没有增大，那么归纳假设表明，要么$max-uid_j$已经足够大，要么$new\text{-}info_i$刚好在r轮前为真。在前一种情况下，$max-uid_j$保持

54

55

⊖ “优化”用在这里并不恰当，“改进”更好一些，但“优化”是标准的用法。

足够大，因为这个值从来不减小。在后一种情况下，新的信息在第r轮会被从i发送到j，导致$max-uid_j$变得足够大。

现在，来证明优化洪泛最大值算法的正确性，我们假想优化洪泛最大值算法和洪泛最大值算法一起运行，以同样的UID赋值开始。证明的核心部分是模拟关系，它涉及相同轮数之后两种算法所处状态的不变式断言。

断言4.1.7　对于任意的r，$0 \leqslant r \leqslant diam$，在$r$轮后，$u$、$max-uid$、$status$和$rounds$分量的值在两种算法中都是一样的。

断言4.1.7的证明是通过对r的归纳，就像只包括单个算法的常见的断言一样。在归纳步骤中重要的部分就是表明$max-uid$的值保持相同。

所以考虑任意的i和j，其中$j \in out-nbrs_i$。如果在第r轮前$new-info_i = true$，那么就和在洪泛最大值算法中一样，优化洪泛最大值算法的进程i也在第r轮向j发送相同的信息。另一方面，如果在第r轮前，$new-info_i = false$，那么在优化洪泛最大值算法中，i不向j发送任何东西，但是在洪泛最大值算法中，i向j发送$max-uid_i$。断言4.1.6表明，在r轮前，如果$max-uid_j \geqslant max-uid_i$，那么在洪泛最大值算法中，消息对$max-uid_j$是无影响的。在两个算法中，$i$对$max-uid_j$有相同的影响。既然这个对所有的$i$和$j$来说都是真的，那么在两个算法中，$max-uid$的值就会保持相同。

断言4.1.7和断言 4.1.1共同证明了断言4.1.5。 □

刚才我们证明优化洪泛最大值算法正确性的方法在证明“优化的”分布式算法的正确性时很有用。首先，证明一个低效率但是简单的算法的正确性。然后通过证明两个算法之间的一个形式化关系，来证明更有效但是更复杂的算法的正确性。对于同步的网络算法来说，这种关系一般采取上面的形式——即包括相同轮数之后两个算法的状态的一个不变式。

另一个改进　我们可以在洪泛最大值算法中稍微减少消息个数。也就是说，如果进程i接收到来自进程j的一个新的最大值，其中j既是i的出向邻接节点也是i的入向邻接节点，即i与j有双向通
56 信，那么i在下一轮的时候就不需要向j的方向发送消息。

在一个具有UID的一般有向图网络中，如果进程不知道节点数n和直径$diam$，选出领导者也是可能的。现在我们建议你停下来去构造一个这样的算法。一种可能的方法是引入一种辅助协议来允许每个进程计算网络的直径。本章中稍后讲到的思想或许有用。

4.2　广度优先搜索

下一问题是，如何基于一个任意的强连通有向图在网络上进行广度优先搜索（BFS），有向图上具有明确区分的源节点。更精确地讲，如何给有向图创建一棵广度优先生成树。创建这种树的目的是为了有一个方便的结构用做广播通信的基础。BFS树可以使一个特定节点上的进程向网络中所有其他进程发送消息的最大通信时间最小化（假设消息在每个通信信道中的传递时间相等）。

在每一对邻接节点都是双向通信的时候，也就是网络是无向图的时候，BFS问题和它的解决方法就比较简单。我们会指出对这种情况的简化。

4.2.1　问题

有向图$G=(V, E)$的有向生成树被定义为一棵带根树，包括E中的所有有向边和G中的每一个顶点，树中所有边都由父节点指向子节点。对于G中根节点为i的有向生成树，如果所有与i的距离为d的节点在树中深度都为d（也就是说在树中与节点i的距离为d），则该树是广度优

先的。每个强连通的有向图都有一棵广度优先有向生成树。

在BFS问题中，我们假设网络是强连通的，并且有一个单独的源节点i_0。这个算法应该能够输出网络图的广度优先有向生成树的结构，根节点为i_0。输出应该是以分布式的形式：除了i_0，其他的每个进程都应该设置一个*parent*分量，用来表示在树中的父节点。

与通常一样，进程只在有向边上通信。进程都有UID，但是不知道整个网络直径的大小。 57

4.2.2 基本的广度优先搜索算法

这个算法的基本思想和标准的串行广度优先搜索算法是一样的。我们称这个算法为*SynchBFS*算法。

***SynchBFS*算法：**

在运行中的任一时刻，总有一个进程集合是被“标记的”，最初的时候是i_0。进程i_0在第一轮时向所有出向邻接节点发出一个搜索消息。在每一轮中，如果一个未标记的进程接收到一个“搜索”消息，就标记它自己，并且从传来“搜索”消息的进程里选择一个作为父节点。在一个进程被标记之后的第一轮，它向所有的出向邻接节点发送“搜索”消息。

不难看出，*SynchBFS*算法建立了一棵BFS树。为了形式化地证明它，我们可以证明一个不变式，即在过了r轮以后，每个在图中与i_0的距离为d（$1 \leqslant d \leqslant r$）的进程都已经定义了自己的“父节点”指针，且“父节点”指针指向与i_0的距离为$d-1$的节点。和往常一样，这个不变式可以通过对轮数作归纳来证明。

复杂度分析 时间复杂度最多为*diam*轮（事实上，这个分析可以更精确一点，即一个特定节点i_0到其他节点的最大距离）。消息的数目正好是$|E|$——因为每个有向边上都正好有一个“搜索”消息要传递。

降低通信复杂度 如同洪泛最大值算法一样，有可能稍微减少消息数目：一个新标记的进程不需要再向传来“搜索”消息的进程发送消息。

消息广播 *SynchBFS*很容易被扩充以实现消息的广播。如果一个进程有一个消息m想要发送到网络中的所有进程，那么它只要把自己当成根节点来执行*SynchBFS*算法，并把m附在第一轮中发送的“搜索”消息上。其他的进程也继续把m附在它们的“搜索”消息上。因为树中包含了所有的节点，所以消息m最后会被送到所有进程。

子节点指针 在BFS问题的一个重要变形中，每一个进程不仅仅需要知道谁是它的父节点，还要知道哪些是它的子节点。在这种情况下，每个接收到“搜索”消息的进程就必须回复一个“父节点”或“非父节点”消息，告诉消息的发送者，是否它已经被接收者选择为父节点。 58

如果在每一对邻接节点中双向通信都是被允许的，也就是说，如果网络是无向图，那么就没有什么难度了，增加额外的通信只会增加很小的开销。但是，因为我们只允许邻接节点之间进行单向通信，所以一些“父节点”消息和“非父节点”消息就可能需要通过间接的路径来传递了。举个例子，一个“父节点”消息或“非父节点”消息可以通过上面这种附加消息的方法，重新执行一次*SynchBFS*来传送。为了让接收者能够认出这个消息来，消息必须带着接收方的UID（还有能使接收者知道发送者的本地名字），并把它附在“搜索”消息中。注意，许多*SynchBFS*的子程序都是可以并行执行的。为了能够符合我们的形式化模型，使得每一轮的每条链路上最多只有一条消息被发送，有必要把很多消息合并成一个。

对于在某些边上单向通信的一个有向图来说，除了输出父节点和子节点，有可能需要输出子节点到父节点的最短路径信息。例如，可以通过额外执行*SynchBFS*来产生这样的信息。

复杂度分析 如果图是无向的，那么计算BFS树（包括子节点指针）的总时间为$O(diam)$，通

信复杂度为$O(|E|)$。

即使有的邻接节点对之间是单向通信的，计算树和子节点指针的时间复杂度还是$O(diam)$，因为额外的BFS运行可以并行地进行。在这种情况下，总的消息数为$O(diam|E|)$，因为在$O(diam)$轮的每一轮中，最多有$|E|$条消息被发送。然而，由于一条消息可能包含多达$|E|$个并发BFS运行的信息，所以每条消息可能多达$|E|b$位，其中b为表示单个UID所需的最大位数。这将导致总共$O(diam|E|^2b)$位的通信量。注意每一个并发BFS运行（最多$|E|$个）最多使用$|E|$条消息，每条消息最多有b位，由此可以得到更小的总位数界限。所以总的通信位数最多为$O(|E|^2b)$。

终止　源进程i_0怎样才能告知树的构造已经完成了呢？如果每个“搜索”消息都得到了“父节点”或“非父节点”消息的回复，那么任一进程接收到“搜索”消息的回复以后，就知道谁是它在
59 树中的子节点，并知道这些子节点已经被标记了。所以，从BFS的叶节点开始，完成构造的通知可以一直“扇入”到源节点：当一个进程（a）已经接收到了所有发出的搜索消息的回复(使它知道谁是它的子节点，并知道它们已经被标记了)，和(b)已经接收到了所有它的子节点发来的完成构造的通知时，它可以马上给它的父节点发送完成构造的通知。这种过程叫做聚播。

如果图为无向的，那么计算一棵BFS树（包括子节点指针）并把完成构造的通知传播回源节点的全部时间约为$O(diam)$，通信复杂度约为$O(|E|)$。如果允许单向通信，那么总的时间（包括完成构造的通知的传播）约为$O(diam^2)$。成平方级增长的原因在于，完成构造的通知需要在树中每次一层地串行传递。因此，总的消息数目为$O(diam^2|E|)$，总的通信位数最多为$O(|E|^2b)$。

4.2.3　应用

广度优先搜索是分布式算法的一块基石。在这里，我们给出了一些例子，显示如何扩充*SynchBFS*算法来完成其他的任务。

广播　正如我们前面提到的，一个消息的广播可以在建立一棵BFS树的过程中实现。另外一个办法就是先用上面的方法建立一个带子节点指针的BFS树，然后用这个树来进行广播。消息只需要沿着父节点到子节点的边进行广播。这就使得构造的BFS树是可重用的，因为很多消息可以在这棵树上传播。一旦BFS树建立起来以后，广播一个消息的时间开销为$O(diam)$，消息的数目只有$O(n)$。

全局计算　BFS树的另外一个应用就是从整个网络中收集信息，或者更一般地说，就是对基于分布式输入的函数的计算。举个例子，假设有这样一个问题，每个进程都有一个非负整数的输入，我们想要在这个网络中计算所有这些输入的和。利用BFS树就可以很容易地（并且高效地）做到。从树叶开始，在聚播过程中“扇入”结果，如下所示。每个叶节点都把值传给自己的父节点；每个父节点都会把从所有子节点中得到的值与自己的输入值求和，再把结果传给自己的父节点。由BFS树的根节点计算出来的值就是最后的结果。

60 假设BFS树已经建立了，并且所有的边的通信都是双向的，这种计划会要求$O(diam)$的时间和$O(n)$的消息数。可以用相同的方法来实现其他的很多函数，例如，求最大或最小的输入整数（这里要求函数满足结合律和交换律）。

选出领导者　使用*SynchBFS*，可以设计一个算法在具有UID的网络中选举领导者，而且各个进程并不需要知道n和$diam$。也就是说，所有的进程都可以并行地进行广度优先搜索。每个进程i用建立的BFS树和全局计算过程来决定整个网络中进程的最大UID。拥有最大UID的那个进程宣布自己为领导者，而其他的进程宣布自己不是领导者。如果图是无向的，那么时间开销就是$O(diam)$，消息数目就是$O(diam|E|)$，同样是因为在diam轮中，每轮最多只有$|E|$条消息被发送。消息位数最多为$O(n|E|b)$，其中b是表示一个UID所需的最大位数。

计算直径　整个网络的直径可以通过让所有进程并行地执行广度优先搜索来计算。每个进程i用

建立的BFS树来决定$max-dist_i$，$max-dist_i$定义为i到网络中其他进程的最远距离。然后每个进程i重用各自的广度优先搜索树做全局计算来找到$max-dist$的最大值。如果图为无向的，那么时间开销为$O(diam)$，消息数为$O(diam|E|)$，消息位数为$O(n|E|b)$。这样计算出来的直径可以用在领导者选举的洪泛最大值算法中。

4.3 最短路径

现在来考察BFS问题的扩充。考虑一个强连通有向图，其中一些邻接节点对之间可能存在单向通信。这次假设每个有向边$e=(i,j)$都与一个非负的实数值“权”（weight）相关联，权用$weight(e)$或者$weight_{i,j}$来表示。一条路径的权可以被定义成所有边的权的和。问题是要找到一条从源节点i_0到图中其他各个节点的最短路径，其中最短路径定义为权最小的一条路径[⊖]。由i_0到其他所有节点的最短路径集合构成了图的一棵子树，所有的边都是由父节点指向子节点。

对广度优先搜索来说，建立一棵BFS树的动机就是为了能够有一个方便的结构来完成广播 61
通信。权代表在边上通信的开销，比如通信延迟或收费。一棵最短路径树的作用是将某一进程与网络中其他进程 在最坏情况下通信的最大开销最小化。

假设开始时每个进程都知道与它相连的所有边的权，或者更精确地说，每条边的权会出现在两个端点进程的特定*weight*变量中。我们还假设每个进程都知道图中节点的个数n。要求每个进程都应确定它在最短路径树中的父节点以及到i_0的距离（它的最短路径的权的和）。

如果所有边的权都相等，那么BFS树也是一棵最短路径树。所以在这种情况下，对做*SynchBFS*做一个小小的改动，就可以使它产生距离信息和父节点指针。

权不相等则是一种更有趣的情况。解决这个问题的一种方法就是用下面的算法——Bellman-Ford串行最短路径算法。

***BellmanFord*算法：**

每个进程i都记录目前所知的从i_0出发的最短距离*dist*，以及作为入向邻接节点的父节点parent（这个父节点位于i之前，与i之间的路径的权重为*dist*）。开始的时候，$dist_{i_0}=0$，对$i\neq i_0$，则$dist_i=\infty$，parent分量还没有定义。在每一轮，每个进程都把*dist*发送给所有的出向邻接节点。然后每个进程通过一个“松弛步”（relaxation step）来更新自己的*dist*，即在前一步的*dist*和所有的$dist_j+weight_{j,i}$中取最小值，其中j是它的入向邻接节点。如果*dist*改变了，那么它的*parent*分量也会相应更新。在$n-1$轮以后，*dist*就包含了最短的距离，*parent*则指向最短路径树中的父节点。

容易看出，经过$n-1$轮后，*dist*收敛到一个正确的距离值。证明BellFord正确性的一种方法就是（通过对r归纳）证明r轮后有以下结论：每一进程i的*dist*和parent分量，与i_0到i的包含最多r条边的最短路径对应（如果没有这样的路径，则$dist=\infty$，*parent*没有定义）。具体证明留做一道习题。

复杂度分析 *BellmanFord*算法的时间复杂度是$n-1$，消息的数目是$(n-1)|E|$。

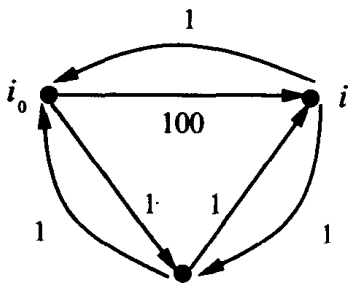


图4-1 虽然*diam*=1，最短路径却只在2轮后稳定 62

例4.3.1 *BellmanFord*的时间复杂度

通过和*SynchBFS*类比，有可能会怀疑*BellmanFord*的

⊖ 权和距离不应当混用，但传统上却是这样。

时间复杂度为*diam*。为什么不是这样呢？图4-1就是一个例子。在这个例子中，需要2轮才能确定从i_0到i的真正距离为2，因为其路径上有两条边。但是图的直径只有1。

*BellmanFord*算法利用n的上界而不是n本身也可以有效工作。如果不知道这样的界限，那么有可能用一些像4.2节中所提到的技术来找到一个。

4.4 最小生成树

下一个要考虑的问题就是在一个具有带权边的无向图网络中找到一棵最小（或最小权）的生成树（MST）。当然这棵生成树的主要用途也是作为广播通信的基础。一棵最小权生成树使任一源进程与网络中其他所有进程通信的总开销最小。

4.4.1 问题

一个无向图$G=(V, E)$的生成森林（spanning forest）是这样的一个森林，它只由E中的无向边组成，并且包括了G的每个顶点。如果无向图G是连通的，那么G的一棵生成树就是它的一个生成森林。如果在E中的无向边是带权的，那么G的任一子图（如G的生成树或生成森林）的权就定义为子图中所有边的权之和。

63 回想在有向图模型中，我们把无向图形式化为每对邻接节点之间都有双向边的有向图。和4.3节一样，假设每条有向边$e=(i, j)$都和一个非负的实数值*weight*相关联，$weight(e)=weight_{i,j}$，不过这次，我们假设对所有的i和j，有$weight_{i,j}=weight_{j,i}$。假设开始时每个进程都知道与它相连的所有边的权，或者更精确地说，每条边的权会出现在两个端点进程的*weight*变量中。我们还假设每个进程都有UID并知道图中节点的个数n。现在的问题是为整个网络找到一棵最小权（无向）的生成树；更精确地说，要求每一个进程确定与它相连的哪些边是最小生成树的一部分。

4.4.2 基本定理

无论串行MST算法还是并行MST算法都是基于一个简单的理论，在这一节中我们要描述这个理论。构造一棵最小生成树的基本策略是：从包含n个孤立节点的生成森林开始，重复地将沿着连接边的各组件合并起来，直到产生最小生成树。其中最重要的是沿着某些选定的边进行合并——也就是选择某个节点中权值最小的出向边进行合并。下面的引理证明了这种选择方法的正确性。

引理4.3　$G=(V, E)$是一个带权无向图，$\{(V_i, E_i): 1\leqslant i\leqslant k\}$为$G$的任一生成森林，其中$k>1$。固定任一$i$，$1\leqslant i\leqslant k$。令$e$为集合$\{e': e'$只有一个端点在$V_i$中$\}$中权最小的一条边，那么就有一棵包括$\cup_j E_j$和$e$的$G$的生成树，且在$G$的包括$\cup_j E_j$的所有生成树中，这棵树的权值最小。

证明　用反证法。假设结论是错误的——也就是说，存在一棵生成树T，它包括$\cup_j E_j$，但不包括e。而且它的权严格小于任意包括$\cup_j E_j$和e的生成树。现在我们把e加到T中得到图T'。显然，T'包括一个环，其中有另外一条边e'，$e'\neq e$，且e'是V_i的出向边。

根据e的选择过程，我们知道$weight(e')\geqslant weight(e)$。现在从$T'$中把$e'$删除得到$T''$。那么$T''$是
64 G的一棵生成树，它包括$\cup_j E_j$和e，而且它的权不大于T的权。这与假设中T的性质矛盾。　□

引理4.3为下面的构建MST的通用策略提供了依据。

MST的通用策略：

从包含n个独立节点且不包含边的生成森林开始，重复以下步骤：在树中选择一个任意的组件C，并且从C的所有出向边中选出权值最小的边e。把C与e的另一端点所在的组件合并起来，

并且把e包括在新合并后的组件中。当森林变成只有一个组件时停止。

在归纳证明中，引理4.3可以被用来说明在构造过程中的任一阶段，所得森林是MST的一个子图。一些有名的串行MST算法属于这个通用策略的特殊情况。举个例子，*Prim-Dijkstra*算法就是开始时只有一个单独的节点，然后重复地把当前组件中具有最小权的出向边加进来，每次都加上一个单独的新节点直到产生一棵完整的生成树为止。再举个例子，*Kruskal*算法重复地在连接当前生成森林的两个独立组件的所有连接边中，选出并加入具有最小权的边，以合并各组件，直到最后只剩下一个组件，这就是生成树。

为了能在分布式环境中使用这样的通用策略，最好能够并发地确定几条边来扩展森林，也就是说，每个组件都能独立地确定它的最小权出向边，然后那些被确定的边就可以被加入到森林中，这样使得几个组件可以一次合并。但是引理4.3并不保证这个并行策略的正确性。事实上，在一般情况下，这个策略是不正确的。

例 4.4.1 在MST算法中环的产生

考虑图4-2中的图。其中点代表了生成森林中的各个组件。三条权为1的边都是出向边。如果每个组件都按照箭头所示的那样来选择权最小的出向边，一个环就会产生了。

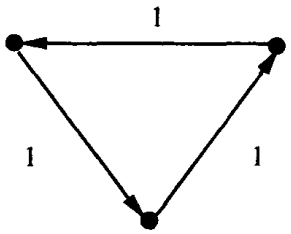


图4-2 一个由于并发选择最小权值的出向边而产生的环

在所有的边都有不同权值的特殊情况下，环的问题是可以避免的。因为有以下引理。

引理4.4 如果图G的所有边都有不同的权，那么只存在惟一一个G的MST。

65

证明 这个证明和引理4.3的证明类似。假设有两个不同的最小生成树，T和T'，令e是只出现在其中的一棵树中的权最小的边。（不失一般性）假设$e \in T$，那么把e加到T'中形成的图T''就包含有一个环，而且在这个环中，至少有另一条边e'不属于T。因为所有边的权都是不同的，且e'只在其中的一棵树中，那么必有$weight(e') > weight(e)$。然后把e'从T''中删除就会产生一棵权比T'更小的生成树，这就产生了矛盾。 □

现在考虑当图的边有不同权值时的通用策略，由引理4.4可知，存在着一个惟一的MST。在这样的情况下，在构造的任一阶段，森林的任一组件都只有一条最小权的出向边（Minimum-Weight Outgoing Edge，MWOE）。引理4.3意味着如果开始时有一个森林，它所有的边都在惟一的MST中，那么对于所有的组件，所有MWOE也在惟一的MST中。所以我们可以把它们一次性加入，不会有产生环的危险。

4.4.3 算法

按照前一小节描述的通用策略，我们给出了一个分布式的算法，在任意的带权无向图中构建MST。因为要求各组件可以并发地合并，假设边的权都是不同的；在本小节的最后部分，可以看到如何消除这个假设。我们称这个算法为*SynchGHS*，因为这个算法是基于由Gallager、Humblet和Spira提出的异步算法（在15.5节中，我们将给出这个简称为*GHS*的异步算法）。

66

***SynchGHS*算法：**

这个算法分层构造各组件。对于每个k，第k层的各组件构成了一个生成森林，其中k层每一组件都包含一棵树，该树是MST的一个子图。k层的每个组件至少有2^k个节点。在每一层中，每个组件有一个独特的领导者节点。进程允许按一个固定的轮数$O(n)$来完成每层的工作。

算法从0层开始，各组件由单独的节点组成且不包含边。我们归纳地假设k层的各个组件已

经确定了（包括它们的领导）。更特别地，假定每个进程都已经知道其组件的领导的UID；这个UID就被用作整个组件的标识符。每个进程也知道它的哪条邻接边在该组件的树中。

为了得到$k+1$层的各个组件，k层的每个组件就会沿着其生成树的边搜索组件的MWOE。领导者使用在4.2节所描述的那种消息广播策略，沿着树的边广播搜索请求。每个进程都在自己的邻接边中找到一条权最小的组件的出向边（如果有的话）；它为了完成这个工作，在所有的非树中的边上传递*test*消息，询问边的另一端点是否属于相同的组件（通过比较组件的标识符来确定）。然后各进程就会把本地最小权边的信息聚播给领导者，途中选择最小的。领导者得到的最小值就是整个组件的MWOE。

当k层的各组件都找到了它们的MWOE后，这些组件就通过所有的MWOE合并为$k+1$层的组件。这涉及k层的每个组件的领导者和与其MWOE相邻的进程通信，告知该相邻进程把这条边标记为属于新树；边的另一端进程也被告知做同样的工作。

然后对于$k+1$层的每个组件，按如下方式产生一个新的领导者。可以证明：对于由每组被合并到$k+1$层组件中的k层各组件，必定存在一条惟一的边e，e是该组的两个k层组件的共同MWOE（以后我们会讨论。）我们让e的两个端点中具有更大UID的端点成为新的领导者。注意这个新的领导者可以只利用本地可见的信息来标识自己。

67 最后，使用广播在新的组件中传播这个新的领导者的UID。

在若干层之后，这个生成森林最终就会只包含由网络中所有节点组成的单个组件。然后，新的寻找MWOE的尝试就会失败，因为没有一个进程能够找到一条出向边。当领导者知道这个情况以后，就可以广播一个消息，告诉大家算法已经完成了。

这个算法的关键就是，在每一组被合并的k层组件中，两个端点组件的公共MWOE是一条惟一的（无向）边。为了证明这一点，我们考虑一个组件有向图G'，它的节点是将要合并成一个k+1层组件的各个k层组件，边是MWOE。G'是一个弱连通有向图，其中每一节点只有一个出向边（如果一个有向图忽略所有边的方向后得到的无向图是连通的，则有向图是弱连通的）。因此我们可以用下面的性质。

引理4.5　设G为一个弱连通图，其中每个节点只有一条出向边，那么G包含且只包含一个环。

证明　此证明就留做一道习题。　□

例 4.4.2　每个节点只有一条出向边的图

图4-3显示了一个例子，其中的每个节点只有一条出向边。

我们对组件有向图G'使用引理4.5，得到组件的惟一环。由构造G'的方法可知，在环中连续的边不应

68 该有递增的权，所以，这个环的长度不可能超过2。所以这个环的长度只可能为2。但这就对应一条边，即两个相邻组件的公共MWOE。

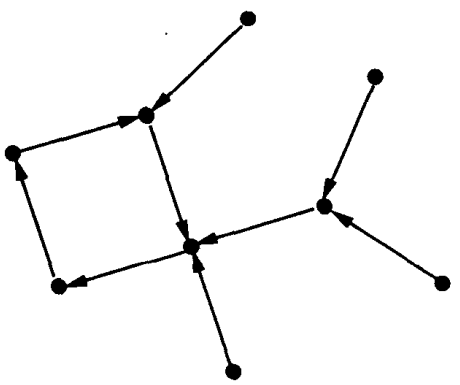


图4-3　每个节点只有一条出向边的图，注意惟一的环

在*SynchGHS*算法中，保持每一层的同步是很重要的。这需要保证当进程i试图确定候选边的另一个端点j是否在同一个组件中的时候，i和j都有最新的组件UID。如果观察到j的UID和i的UID不同，那么我们就会确定i和j确实在不同的组件中，而不只是认为它们尚未从各自的领导者那里接收到组件的UID。为了能保证每层的同步运行，进程允许对每一层有一个预

先设定的轮数。为保证一轮中的所有计算能够完成，这个数目是$O(n)$；注意$O(diam)$不总是足够的。节点需要知道n的惟一原因就是需要计算轮数。（在15.5节中，当我们在异步网络环境中改写这个算法的时候，将使用一个不同的同步组件的策略）。

复杂度分析 首先注意到在每个k层组件的节点数至少为2^k。这个可以用归纳法证明，因为在每一层中，每个组件都会至少和另一个同层的组件合并。所以，层数最多为$\log n$。因为每层需要的时间为$O(n)$，所以*SynchGHS*的时间复杂度就是$O(n \log n)$。通信复杂度是$O((n+|E|)\log n)$，因为在每层中，共有$O(n)$条消息沿着树的所有边发送，而且需要$O(|E|)$条额外消息来寻找局部最小权边。

减少通信 如果采用一种更加精细的策略来寻找局部最小权边，就有望将消息的数量减少到$O(n+|E|\log n)$。这样的改进会导致时间复杂度的增加，但是不会改变其数量级。其思想如下。

当一个进程发现它的邻接边的另一端点属于同一组件时，就将邻接边标记为“剔除”，之后，就不需要再测试它们。同样，在每一层中，剩余的候选边按权递增的顺序，每次测试一个，直至找到组件的第一条出向边（或没有候选边）为止。

有了这样的改进，树的边上发送的消息数和以前一样，是$O(n \log n)$。我们来逐步分析用于寻找局部最小权边的消息数。每条边最多一次被测试和被剔除，总共为$O(|E|)$。一条边如果被测试并确定为局部最小权边，但不是整个组件的MWOE，有可能还会被测试。然而，在每一轮的每一个节点上，这种情况最多发生一次，共有$O(n \log n)$次。所以总的通信复杂度为$O(n \log n+|E|)$。 69

以上策略还有一个好处。因为每一节点都将其在MST中的和不在树中的邻接边作了标记，所以不需要一个完成阶段来让领导者通知每一节点算法已经结束。每一节点仅仅输出其邻接边的信息。

边的权非惟一 现在来考虑这样一个图中的MST问题，其中各边的权不必明确确定。在这种情况下，可对*SynchGHS*算法作一个小的修改。首先注意到*SynchGHS*算法只是对权进行{<,=,>}的比较操作。

如果给定任意的边权，我们就可以利用UID得到一个独特的边标识符集合。边(i, j)的标识符是三元组$(weight_{i,j}, v, v')$其中v和v'是i和j的UID，并且$v<v'$（这样，(i, j)和(j, i)有相同的边标识符）。在边标识符中，基于三元组的字典顺序来定义一个全序关系。

由于*SynchGHS*算法在对权操作时只是用了比较，因此我们可以用边的标识符来替代这些（实数值的）权，最后，*SynchGHS*算法的运行结果，与它在一个符合同样顺序关系的权值确定的集合上的运行结果是一样的。故此生成一棵树。证明这棵树实际上就是原始图的一个MST的任务留做一道习题。

领导者选举 对于一个基于无向图的网络来说，一旦一个MST（或任一生成树）是已知的，那么给定UID，就很容易选举一个领导者。也就是说，从叶节点开始沿着树的路径做聚播；每一个内部节点在向它的邻接节点发送消息前，等待其他所有邻接节点的消息。如果一个节点收到了它的所有邻接节点的消息，而自己没有发出任何消息，那么它就声明自己是领导。如果两个邻接节点在同一轮中都得到了对方的消息，那么它们之中UID大的那个宣布自己为领导者。这个领导者选举过程的总开销（MST构造之后）只需$O(n)$的时间和$O(n)$条消息。

把这和MST的复杂度分析相结合，我们可以看到，在一个无向带权图中，如果每个节点知道n（而不是$diam$），就可以在$O(n \log n)$时间内，用$O(n \log n+|E|)$次通信选出领导者。 70

4.5 最大独立集

本章考虑的最后一个问题是寻找一个无向图的节点的最大独立集（MIS）。如果一个节点

集不包含相邻的节点，则称之为独立集；如果一个独立集不能通过加入任一其他节点来形成更大的独立集，则称之为最大独立集。注意，一个无向图可以有许多不同的最大独立集。我们并不需要所有可能的最大独立集中最大的一个——任一个最大独立集就足够了。

MIS的问题源自在网络的进程间分配共享资源的问题。图G中的邻接节点可能代表不能同时对共享资源进行某些操作（如数据存取或无线广播）的进程。我们可能希望选择一个集合，集合中的进程可以同时进行活动，为了避免冲突，这些进程应该由G中的一个独立集组成。另外，鉴于性能的考虑，如果一个进程没有邻接节点是活动的，就没有必要阻塞这个进程；所以所选择的进程集合应该是最大的。

4.5.1 问题

令$G=(V, E)$是一个无向图。对于一个节点集合$I \subseteq V$，如果所有的节点$i, j \in I$都满足$(i, j) \notin E$，则称I为*独立的*。如果任何真包含I的集合I'都不是独立的，那么称独立集I为*最大的*。我们的目标就是要计算出G的最大独立集。更确切地说，就是下标在I中的每个进程最后都得输出*winner*，即将其状态中一个特定的状态分量的值设置为*winner*。而且，任何下标不在I中的进程应该输出*loser*。

假定所有进程都知道节点的个数n（也可以用n的一个上限）。我们并不假定UID必须存在。

4.5.2 随机化算法

不难证明，在一些图中，如果进程是确定的，则MIS问题不能被解决。其证明过程和定理3.1的证明过程相类似。在本节中，我们给出了一个简单的解决办法，使用随机化来克服这个确定性系统固有的限制。更精确地说，我们注意到随机化算法可以解决比上面所说的稍微弱一点的问题，该问题存在一个无法终止的概率（概率为0）。我们根据这个算法的发明者Luby的
71 名字将这个算法命名为*LubyMIS*。

*LubyMIS*算法基于以下的迭代方案，从所给的图G中选择一个任意的非空独立集，从图中删除这个集合中所有的节点和它们的邻接节点，重复这一过程。如果W是图的节点的一个子集，那么我们就用*nbrs*（W）来表示W中节点的邻接节点集。

定义*graph*为一个记录，包含字段*nodes*，*edges*和*nbrs*，初始值为源图G中指定的组件。

令I为一个节点集，初始化为空。

```
while graph.nodes ≠ φ do
  choose a nonempty set I' ⊆ graph.nodes that is independent in graph
  I := I ∪ I'
  graph := the induced subgraph^Θ of graph on graph.nodes − I' − graph.nbrs(I')
end while
```

不难看出，这种方法能够产生出最大独立集。要理解为什么它是独立的，请注意在每一阶段，选择的集合I'是独立的，并且我们从剩下的图中明确去除那些放进I的顶点的所有邻接节点。要理解为什么它是最大的，请注意我们删除的仅仅是放进I的节点的邻接节点。

在一个分布式网络中实现这个方案的关键问题是，如何在每次循环中选择I'。这里就用到了随机化。在每个阶段中，每个进程i按均匀分布在范围$\{1, \ldots, n^4\}$中随机选择一个整数val_i。用n^4作为边界的原因是它大得足以令图中的进程以高概率选择不同的值（我们在本书中不展开计算，而是建议你参考Luby的论文）。一旦进程选择了这些值，我们就定义I'由所有作为局部*winner*的节点i组成，也就是在i的所有的邻接节点j中，$val_i > val_j$。这显然就产生了一个独立集，

Θ 在其节点的一个子集W上的图G的归纳子图被定义为子图(W, E')，其中E'是W中连接节点的G的边的集合。

因为两个邻接节点不可能同时击败对方。

在这样的实现中，如果随机选择不巧的话，有可能出现集合I'在某些阶段是空的；这些阶段将没有用处，什么都不能完成。如果算法不会到达这样的点，在该点之后算法将无休止地在无用阶段运行，那么我们可以简单地忽略无用阶段并且断定LubyMIS可以按一般模式正确运行。 72
但是在我们的分析中必须考虑这些无用阶段。算法如下。

LubyMIS算法（非形式化）：

算法分阶段运行，每个阶段由三轮构成。

第1轮：在一个阶段的第一轮中，各个进程都选择它们各自的*val*，并把它们发送给自己的邻接节点。到第一轮的最后，当所有的*val*消息都被接收到时，*winner*——也就是在I'中的进程——将会知道它们是谁。

第2轮：在第二轮中，*winner*通知它们的邻接节点。在第二轮的最后，*loser*——也就是有邻接节点在集合I'中的进程——知道它们是谁。

第3轮：在第三轮中，每个*loser*通知它的邻接节点。然后所有相关进程——*winner*和*loser*的邻接节点——从图中删除合适的节点和边。更精确地说，在这个阶段之后，*winner*和*loser*不再参与以后的工作，*loser*的邻接节点也删除所有与新删除节点相关的边。

我们现在在模型中更加形式化地讨论这个算法。正如在2.7节中所说的那样，每个进程都有一个特定的随机函数$rand_i$，在每轮中它在应用$msgs_i$和$trans_i$函数之前使用。这里我们使用*random*来表示按均匀分布从$\{1, \ldots, n^4\}$随机选择。

LubyMIS算法（形式化）：

$states_i$:
$round \in \{1,2,3\}$, initially 1
$val \in \{1,\ldots,n^4\}$, initially arbitrary
awake, a Boolean, initially *true*
rem-nbrs, a set of vertices, initially the neighbors in the original graph G
$status \in \{unknown, winner, loser\}$, initially *unknown*

$rand_i$:
if *awake* and $round = 1$ then $val :=$ *random*

$msgs_i$:
if *awake* then
  case
    $round = 1$:
      send *val* to all nodes in *rem-nbrs*
    $round = 2$:
      if $status = winner$ then 73
        send *winner* to all nodes in *rem-nbrs*
    $round = 3$:
      if $status = loser$ then
        send *loser* to all nodes in *rem-nbrs*
  endcase

在以下代码中，我们按模3计数，用0来标识3。

$trans_i$:
if *awake* then
  case

round = 1:
　if *val* > *v* for all incoming values *v* then *status* := *winner*
round = 2:
　if a *winner* message arrives then *status* := *loser*
round = 3:
　if *status* ∈ {*winner*, *loser*} then *awake* := *false*
　rem-nbrs := *rem-nbrs* − {*j* : a *loser* message arrives from *j*}
endcase
round := (*round* + 1 mod 3)

注意，如果在某些阶段，某些相邻进程选择了相同的随机值，那么*LubyMIS*仍可以正确地运行。

4.5.3 分析*

我们已经指出，如果LubyMIS算法不停滞在无用阶段中运行，它将会产生一个MIS。现在我们声明事实上这个算法不会停滞的概率为1。更确切地说，我们声明在算法的任一阶段，预期从剩余的图中删除的边数，至少占剩下的总边数中一个不变的比例；也就是说有恒定的概率表明，至少会有一部分的边被删除。因此，这意味着直到终止为止算法的期望运行轮数为$O(\log n)$。它还表明这个算法的确会终止。

对*LubyMIS*算法的完整分析可以在Luby最初的论文中找到。论文中包含了很多关于图的计数证明。我们只是列出了主要的引理，但不给出证明，并指出怎样用它获得所需的结果。对下面的三个引理，我们确定$G=(V, E)$，并且对于任意的节点i，定义

$$sum(i) = \sum_{j \in nbrs_i} \frac{1}{d(j)}$$

其中$d(j)$是j在G中的度数。下面是引理：

74

引理4.6　设I'是*LubyMIS*算法在某个阶段中的集合，那么对于图中的每个i，在该阶段之前，

$$\Pr[i \in nbrs(I')] \geq \frac{1}{4} \min\left(\frac{sum(i)}{2}, 1\right)$$

用引理4.6，我们得到从图中删除边数的期望值的界限：

引理4.7　在*LubyMIS*算法的一个阶段中，从G中删除的边的期望数目至少是$|E|/8$。

证明　算法保证那些至少有一个端点在$nbrs(I')$中的每条边都被删除。因此期望删除的边数至少为

$$\frac{1}{2} \sum_{i \in V} d(i) \cdot \Pr[i \in nbrs(I')]$$

这是因为每一个顶点i有一个指定的概率表明其在I'中有一个邻接节点；如果确是如此，则删除i，这会导致它的$d(i)$条邻接边都被删除。引入因子1/2是考虑到被删除边有可能被重复计数而做的补偿，因为每条边的两个端点都可能导致它被删除。

接下来我们代入引理4.6中的界限，可得被删除边数的期望值至少为

$$\frac{1}{8} \sum_{i \in V} d(i) \cdot \min\left(\frac{sum(i)}{2}, 1\right)$$

根据*min*中的最小项，分解上式，得

$$\frac{1}{8}\left(\frac{1}{2}\sum_{i:sum(i)<2} d(i)\cdot sum(i)+\sum_{i:sum(i)\geq 2} d(i)\right)$$

现在，我们扩展$sum(i)$的定义并且把$d(i)$写为和式，得到

$$\frac{1}{8}\left(\frac{1}{2}\sum_{i:sum(i)<2}\sum_{j\in nbrs_i}\frac{d(i)}{d(j)}+\sum_{i:sum(i)\geq 2}\sum_{j\in nbrs_i}1\right)$$

注意，每一条无向边(i,j)产生扩号中的两个和式，每个方向一个和式；在每一种情况下，两个和式的和大于1。所以总和至少为$|E|/8$。□ 75

由引理4.7可以得到

引理4.8 在*LubyMIS*的单个阶段中，从图G中删除的边数至少为$|E|/16$的概率至少为1/16。

根据引理4.7和引理4.8，可得：

定理4.9 算法*LubyMIS*最后终止，而且直至终止时的期望轮数为$O(\log n)$的概率为1。

随机化的算法 在分布式算法中经常用到随机化的方法。它的主要用途就是用来打破对称性。例如，在一般的图中，不具有UID的确定性进程不能解决领导者选举和MIS问题，原因在于无法打破对称性。相反，这样的问题在使用随机化以后就可以解决。即使有UID，随机化也可能更快地打破对称性。

但是，随机化算法的一个问题就是，它们的正确性和/或性能只能以比较高的概率来保证，不能完全确定。在设计这样的算法时，重要的是应该保证算法的关键属性是确定的，而不是概率性的。举个例子，不管随机选择的输出是什么，*LubyMIS*算法的每一个运行都保证产生一个独立集。然而，性能却依赖于随机选择的运气。甚至有一种（近乎于零的）概率使得所有的进程都会不停地反复选择相同的值，这样程序就会永远停滞。这些是否属于算法的严重缺点取决于算法的具体应用。

4.6 参考文献注释

*FloodMax*和*OptFloodMax*算法很早就产生了。Afek和Gafni[6]研究了在完全同步网络系统中领导者选举问题的复杂度界限。*SynchBFS*算法基于标准的串行广度优先搜索算法，例如[83]中有这样的算法。*BellmanFord*算法是一个串行算法的分布式版本，分别由Bellman和Ford[43, 125]提出。

*SynchGHS*算法是著名的异步MST算法的并行版本（因此做了很大简化），由Gallager、Humblet和Spira提出。*LubyMIS*算法和它的分析由Luby[200]提出。76

一个随机化算法（概率几乎为0）运行的例子出现在[271]中，其中进程总是作出相同的选择。

4.7 习题

4.1 详细证明*FloodMax*算法的正确性。

4.2 就n来讲，易于看出*FloodMax*算法中用到的消息数$diam|E|$为$O(n^3)$。构造一类有向图，使其中的$diam|E|$为$\Omega(n^3)$，或者证明这样的有向图不存在。

4.3 对于*OptFloodMax*算法，证明一个小于$O(n^3)$的消息数的上限，或者给出一类有向图及其对

应的UID赋值，使其中的消息数为$\Omega(n^3)$。

4.4 对4.1.3节中的*OptFloodMax*算法做进一步优化，防止进程又向先前给它发送*max – uid*的进程发送相同的*max – uid*信息。

a）用本章的代码格式给出该算法的代码。

b）使用证明*OptFloodMax*算法正确性的模拟策略（在定理4.2的证明中），通过联系*OptFloodMax*，证明算法的正确性。

4.5 a）写出*SynchBFS*算法的代码。

b）用不变式断言证明算法的正确性。

c）对有子节点指针的*SynchBFS*算法做a）和 b）部分。

d）对有子节点指针和完成构造通知的*SynchBFS*算法做a）和 b）部分。

4.6 考虑4.2.2节中描述的*SynchBFS*的优化版本，它防止进程向它先前收到*search*消息的进程发送同样的消息。

77 a）给出算法的代码。

b）使用证明*OptFloodMax*算法正确性的模拟策略（在定理4.2的证明中），通过联系*SynchBFS*，证明算法的正确性。

4.7 详细描述一个扩展的*SynchBFS*算法，它不仅产生子节点指针，而且产生从BFS树中的子节点到父节点的最短路径的信息。这些信息分布在路径上，使得路径上的每一进程知道路径上的下一进程。分析其时间和通信复杂度。

4.8 详细描述一个扩展的*SynchBFS*算法，它允许源进程i_0向其他所有进程广播消息，并得到所有进程已经收到消息的确认。算法用$O(|E|)$条消息和$O(diam)$的时间。可以假设网络是无向图。

4.9 对4.2节结尾的全局计算模式、领导者选举模式和直径计算模式，分析时间和通信复杂度，假设某些邻接节点之间允许单向通信。

4.10 给定一个强连通的有向图，其中进程具有UID但不知道进程数和网络直径，设计出所能想到的最高效的领导者选举算法。

a）假设每一对邻接节点之间的通信都是双向的，即网络是无向图，完成这个工作。

b）不做上面的假设而完成这个工作。

分析这个算法。

4.11 在进程拥有UID的强连通有向图中，设计所能想到的最高效的计算节点数的算法。

a）假设每一对邻接节点之间的通信都是双向的，即网络是无向图，完成这个工作。

b）不做上面的假设而完成这个工作。

分析这个算法。

78 4.12 在进程拥有UID的强连通的有向图中，设计所能想到的最高效的计算边数的算法。

a）假设每一对邻接节点之间的通信都是双向的，即网络是无向图，完成这个工作。

b）不做上面的假设而完成这个工作。

分析这个算法。

4.13 在任意无向图中设计一个所能想到的最高效的算法，确定有最小高度的带根的生成树。假设进程具有UID，但是没有独特的领导节点。

4.14 a）给出*BellmanFord*最短路径算法的代码。

b）利用不变式断言证明其正确性。

4.15 给出*SynchGHS*算法的代码。

4.16 证明引理4.5。

4.17 在*SynchGHS*算法中，证明$O(diam)$轮并不总是足以满足每一轮计算。

4.18 证明在用边标识符代替边权的*SynchGHS*版本中（在4.4节结尾处有描述），的确能够产生MST。

4.19 研究性问题：设计一个最小生成树算法，使它的时间复杂度或者通信复杂度或两者均好于*SynchGHS*算法。

4.20 给出4.4节最后聚播算法的代码，给定一个无向图网络的任意生成树，选出领导者。

4.21 对于在无向图网络中构造任意生成树的问题，给出最好的上限和下限。假设进程有UID，边没有权。仔细说明所用到的有关图的进程知识。

4.22 考虑一个线形网络，也就是1,...,n个进程的线性集合，其中每个进程和它的邻接节点双向连接。假设每一进程能区分它的左右邻接节点，并且知道它是否为端节点。 79

假设每一进程i在初始时有一个很大的整数值v_i，而且任意时刻只能在它的内存中保存常数个这样的值。设计一个算法在进程间对这些值排序，也就是说，每个进程返回一个输出值o_i，其输出集合等价于输入集合，并且$o_1 \leqslant \cdots \leqslant o_n$。依据消息数和轮数，设计所能想到的最有效的算法。证明你的结论。

4.23 在4.5节的假设条件下证明，如果进程是确定性的而不是概率性的，那么在有些图中不可能解决MIS问题。对你的不可能结果，找出满足它的最大的图类。

4.24 假设*LubyMIS*在一个大小为n的环网中执行。估计在算法的一次循环中，将某一特定的边从图中删除的概率。 80

第5章 链路故障时的分布式一致性

在这一章和下面两章中，我们将要学习有关分布式网络中达成一致（*reaching consensus*）的问题。在这个问题中，网络中的每个进程以特定类型的初始值开始，并最终输出相同类型的值。输出必须是相同的——进程必须统一意见（*agree*）——即使输入可以是任意的。对于每种输入模式，通常有一个有效性条件（*validity condition*）来描述允许的输出值。

当系统的组成部分不存在故障时，一致性问题很容易解决，用一些简单的消息交换就可以了。为了使得事情更为有趣，我们一般在故障存在的环境中考虑这个问题。在这一章中，我们考虑通信故障存在时的基本一致性问题；而在第6章中，我们考虑进程故障。第7章中包括了一些基本问题的变形，此时进程故障也是存在的。

在许多的分布式计算应用中都存在一致性问题。举个例子，进程们可能会试图就是否接受或者放弃分布式数据库事务处理的结果达成一致意见。或者进程可能试图就基于多个高度计的度数来对飞机高度进行估计，达成一致意见。或者进程试图在给定不同进程给出的不同测试结果之后，就是否把一个系统组成部分确定为存在故障而达成一致。

在这一章我们提出的特殊的一致性问题叫做协同攻击问题（*coordinated attack problem*），
81 这是在消息可能丢失的环境中达成一致的基本问题。首先我们给出一个针对确定性系统的不可能性结果，然后探索随机化解法的可能性。我们可以证明该问题可以使用一个随机化算法来解决，但是有一定的出错概率。而且出现错误的可能性是不可避免的。

5.1 协同攻击问题——确定性版本

首先我们根据战场场景来给出一个非形式化的（事实上是含糊的）问题叙述。

> 几位将军想要从不同的方向对同一个目标发起协同攻击。他们知道取胜的惟一方法就是所有将军一起攻击；如果只是其中几位进行了攻击，那么他们的军队会被消灭。每个将军都会有一个初始的意见，即军队是否做好了进攻的准备。
>
> 将军们都处在不同的地点。相邻的将军可以互相通信，但是只能靠通信兵步行传递消息。但是通信兵有可能会迷失或者被捕，那么消息就有可能会丢失。依靠这样不可靠的通信，将军们必须为是否要攻击达成一致。而且，如果有可能的话，他们就发动攻击。
>
> （我们假设将军的“通信图”是无向连通图，所有的将军都知道这张图，并且存在一个通信兵成功传递消息的时间上限。）

如果所有的通信兵都是可靠的，那么将军可以派他们向其他所有将军送信息（可能经过几站），告诉他们是否愿意进攻。在经过与“通信图”的直径相等的“轮”数后，所有的将军就会知道这个信息。那么他们就可以用一个大家统一的原则来决定是否攻击：比如，如果所有的将军都想发动攻击，他们就可以决定发动攻击。

在通信兵可能迷失的模型中，这个简单算法就不能工作了。其实，并不只是这个算法存在这种问题：我们将说明没有一个算法能够总是正确地解决这个问题。

上述问题背后的实际计算机科学问题是分布式数据库的提交问题。这个问题涉及到那些参
82 与数据库事务处理的进程集合。在事务处理之后，对是否要提交（即结果是永久性的且释放给其他事务使用）或者取消处理结果（即丢弃结果），每个进程都会形成一个最初“意见”。如果

一个进程中代表那个事务的所有局部计算成功完成，它可能会趋向于提交这个事务，否则趋向于取消事务。进程们会互相通信并且最终同意其中一个结果（提交或取消）。如果可能的话，结果应该是提交。

在证明这个不可能性结果之前，我们先来更正式地叙述它并去除含糊的地方。我们考虑在一个任意的无向图网络中，有n个进程，其索引是1，…，n；并且所有的进程都知道整个图，包括进程的索引。每个进程在开始时，其指定的状态组件中有一个在{0, 1}中的输入。我们用1来表示“进攻”或者提交，用0来表示“不进攻”或者取消。我们采用先前一直使用的同步模型，但是允许任意数量的消息在运行过程中丢失。（见2.2节中的定义）我们的目标在于让进程通过把特定决定状态组件设置为0或1，最终输出一个在{0, 1}中的决定。进程所做的决定必须满足三个条件：

一致性 没有两个进程决定出不同的值。

有效性

1）如果所有进程都以0开始，那么0就是惟一可能的决定值。

2）如果所有进程都以1开始而且所有消息都被传递，那么1就是惟一可能的决定值。

终止性 所有进程最后都会做出决定。

其中一致性和终止性的要求都是很自然的，有效性的要求则只是一种可能性——还存在几种有用的替代。一般说来，有效性条件表明所有的决定值都应该是“合理的”。比如，在这种情况下，根据有效性要求的第2点，总是决定出0的协议就被排除了。我们上述的有效性条件是很弱的。举例来说，如果有一个进程以1开始，那么算法允许可以决定出1，如果所有的进程都以1开始但是丢失了一个消息，那么算法可以决定出0。由于本章重点放在不可能性结果上，这个弱表达式是适当的。其实，即使是这样的弱表达式都不可能在含有两个或者更多节点的图中解决。

83

我们来证明针对只有两个节点且只有一条边这种特殊情况的不可能性结果。我们把证明两个或者多于两个节点的情况留做一道习题。在这个证明中，我们用了在第2章中给出的运行和不可区分性（~）的形式定义。

定理5.1 令G是由节点1和节点2和连接这两个节点的一条边组成的图，则不存在一个可以解决G上协同攻击问题的算法。

证明 采用反证法。假设存在一个解法，比如说算法A。不失一般性，我们可以假设对于每个进程，每个输入值只含于一个开始状态中。这意味着对于固定的输入指派和固定的成功消息模式，系统只有一个运行与之对应。我们再假设在A中的每一轮中两个进程都发送消息，这是因为我们可以强迫它们发送虚拟（dummy）消息。

令α为当两个进程都从1开始，并且所有的消息都被成功发送时所得到的运行。根据终止性要求，两个进程最终都做决定；又根据有效性条件的第2点，两个进程都决定出1。假设两个进程都是在r轮内做出决定。现在令α_1和α相同，但是在r轮后所有消息都丢失了。那么在α_1中，两个进程在r轮内还是决定出1。α_1中的通信模式见图5-1。边表示传递的消息，消息发出但是没有被传递的消息在图中没有画出。

从α_1开始，我们现在来构造出一系列运行，每个运行与它在序列中的前一个运行对于一个进程是不可区分的。因此这些运行必然做出相同的决定值。

令α_2和α_1是一样的运行，但是进程1向进程2发送的最后一个（第r轮）消息没有被送到（见图5-2）。那么在α_1和α_2中，虽然在r轮以后进程2的状态可能不同，但是这种不同与进程1无关。

所以 $\alpha_1 \overset{1}{\sim} \alpha_2$。进程1在$\alpha_1$中决定出1，它在$\alpha_2$中还是决定出1。根据终止性和一致性属性，在$\alpha_2$中进程2（最后）也决定出1。

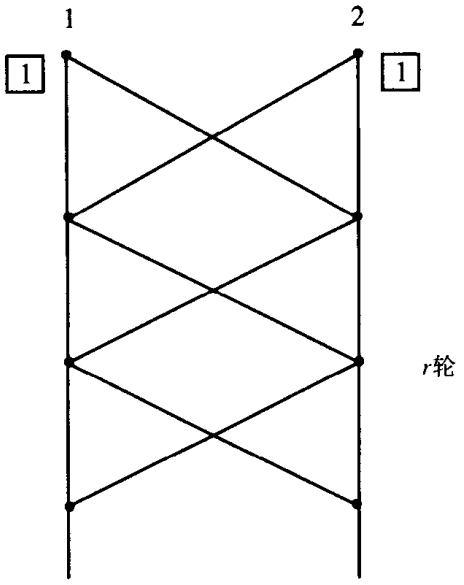


图5-1 运行α_1中的消息交换模式

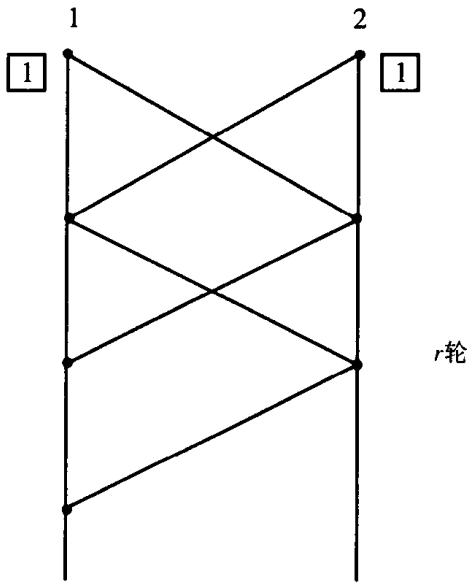


图5-2 运行α_2中的消息交换模式

84 下一步，让α_3和α_2相同，但是最后从进程2到进程1的消息丢失了。因为 $\alpha_2 \overset{2}{\sim} \alpha_3$，在$\alpha_3$中进程2决定出1，根据终止性和一致性属性，进程1也决定出1。

继续这个过程，我们交替去掉从进程1发送给进程2的最后一条消息，最后可以得到一个运行α'，两个进程都由1开始但是没有消息发送。就像上述原因，两个进程都决定出1。

现在我们考虑运行α''，其中进程1是从1开始，进程2是从0开始，而且没有消息被发送。那么 $\alpha_1'' \overset{1}{\sim} \alpha_2'$，所以进程1在$\alpha''$中还是决定出1，根据终止性和一致性，进程2也决定出1。但是 $\alpha'' \overset{2}{\sim} \alpha'''$，其中$\alpha'''$是两个进程都由0开始并且没有消息发送的运行。所以进程2在α'''中决定出1。但是这就产生了矛盾，因为有效性的第一点说明，在α'''中两个进程都决定出0。 □

定理5.1描述了分布式网络能力的一个基本限制。我们对解决诸如不可靠通信的分布式数据库提交问题等基本一致性问题能做的工作很少。但是在实际系统中这个问题的某些版本必须解决。为了能够对付定理5.1的限制，我们有必要加强模型或者放松问题的要求。

一种方法是在保持进程确定性的同时，对消息丢失做某些概率性假设。然后我们必须允许以一定概率来违反一致性和有效性条件。我们把这种环境中的算法设计留做一道习题。第二种方法就是允许进程使用随机化，并且允许以一定概率来违反一致性和有效性条件。我们将在
85 5.2节中讨论这种方法。

5.2 协同攻击问题——随机化版本

在这一节中，我们考虑在进程可被随机化的条件下的协同攻击问题。与上一节中一样，我们考虑n个进程被安放在一个任意的、已知的无向图网络中。每个进程在开始时，在其指定的状态组件中有一个输入在{0, 1}中；我们假设对于每个进程，每个输入值只含于一个开始状态中。在本节中，我们还假设协议在固定的r轮内终止，$r \geq 1$，准确的说，在r轮结束以后，每个

进程都通过把其特定决定状态组件设置为0或1，最终输出一个在{0, 1}中的决定。我们假设在第k轮中消息都沿着每条边发送，$1 \leqslant k \leqslant r$，而且可能有任意数量的消息丢失。

目标还是像以前那样，只是我们弱化了问题的叙述，允许出现错误的可能性。也就是说，我们还是用以前的有效性条件，但是弱化了一致性条件，允许以小概率ε发生不一致。我们根 86
据轮数r来得到ε的可取值的上限和下限结果，你将会看到ε的这个可取值并不小。

5.2.1 形式化模型

为了将这个问题形式化，我们必须搞清楚概率性声明的意义——这里的情况比在4.5节中讲到的MIS问题的情况还要复杂。复杂性在于产生的运行不仅依赖于随机化选择的结果，并且还依赖于哪些消息被丢失。我们并不假定消息的丢失也是随机确定的，而是想象它们由“对手”决定，这些对手试图把事情弄得对于算法来说尽量难。我们通过考虑在所有可能对手上的最坏情况时的行为来评估算法。

通信模式的形式化定义为一个以下集合的任一子集：

$\{(i, j, k) : (i, j)$ 是图中的一条边，且$1 \leqslant k\}$。

如果对于每个 $(i, j, k) \in \gamma$，都有$k \leqslant r$，则通信模式γ被定义为良好的（该概念只适用于本章——在第6章中我们将使用另一个“良好性”的概念）。一个良好的通信模式代表了在某个运行中被传递的消息的集合：如果 (i, j, k) 在通信模式中，那么一个在第k轮从i送到j的消息被成功传递了。

我们在这里用的“对手”的概念是以下两点之一：

1）将输入值赋给所有进程。

2）一个良好的通信模式。

对于任意的对手，进程所做的任意一组随机选择就决定了一个惟一的运行。所以对任一特定对手，进程所做的随机选择导致了运行集合上的一个概率分布。利用这个概率分布，我们可以表达诸如所有进程达成一致等事件的概率。为了强调对手的作用，我们用记号Pr^B来表示由给定对手B引起的概率函数。

现在，我们在这种概率化环境中重新叙述协同攻击问题，其中用到了参数ε，$0 \leqslant \varepsilon \leqslant 1$。

一致性 对于任一对手B，Pr^B[某些进程决定出0而某些进程决定出1]$\leqslant \varepsilon$。 87

有效性 和前面一样。

我们并没有要求终止性条件，因为我们已经假设了所有的进程都在r轮之内做出决定。我们的目标就是找到一个算法使它有最小的ε值，并且证明不会有更小的ε。

5.2.2 算法

为了简单起见，在这一节和下一小节中，我们将注意力限于n节点完全图的特殊情况。我们把任意图的情况留做一道习题。对于这种特殊的情况，我们有一种简单的算法，其ε为$1/r$。

这个算法基于哪些进程互相知道初值、它们如何得到相互的初值知识，等等。我们需要一些定义来捕捉这些知识。

首先，对任意通信模式γ，我们定义一个 (i, k) 对上的自反的偏序关系$\leqslant_\gamma$，其中i是进程索引，k是时间，$0 \leqslant k$。（回忆第2章中，时间k是指在运行中，经过k轮后到达的一点。）这种偏序代表了在不同时刻、不同处理器之间的信息流。我们定义其关系如下：

1）对所有的$i, 1 \leqslant i \leqslant n$和所有的$k, k', 0 \leqslant k \leqslant k'$，有$(i, k) \leqslant_\gamma (i, k')$。

2）若 $(i, j, k) \in \gamma$，则$(i, k-1) \leqslant_\gamma (j, k)$。

3）若 $(i, k) \leqslant_\gamma (i', k')$ 且$(i', k') \leqslant_\gamma (i'', k'')$，则 $(i, k) \leqslant_\gamma (i'', k'')$。

第一种情况描述了在同一个进程中的信息流。第二种情况描述了从消息的发送者到接收者的信息流。第三种情况是一个传递闭包。相似的信息流思想会在本书的后面几章中遇到，比如

在第14、16、18和19章。

现在对任意良好的通信模式γ，我们递归地定义在任意时间k的任一进程i的信息等级$level_\gamma(i,k)$，其中$0 \leq k \leq r$。这里分三种情况：

1）$k=0$: 则定义$level_\gamma(i,k)$为0。

2）$k>0$且存在某个$j \neq i$使得$(j,0) \not\leq_\gamma (i,k)$不成立，则定义$level_\gamma(i,k)$为0。

3）$k>0$且对于任意$j \neq i$，$(j,0) \leq_\gamma (i,k)$。

则对于每个$j \neq i$，令l_j代表$\max\{level_\gamma(j,k') : (j,k') \leq_\gamma (i,k)\}$。

88

（这是i知道的j达到的最大等级。）注意对于所有j，有$0 \leq l_j \leq k-1$。定义$level_\gamma(i,k)$为$1+\min\{l_j : j \neq i\}$。

换句话说，每一个进程从0级开始；当它收到其他所有进程的消息后，就进入到等级1。当它知道所有其他的进程都已经到了1，那么它就进到2，依次类推。如果B是通信模式γ的一个对手，我们有时用$level_B(i,k)$来表示$level_\gamma(i,k)$。

例5.2.1　信息级别

设$n=2$和$r=6$。令γ为良好的通信模式，它由以下三元组组成。

$(1,2,1)$, $(1,2,2)$, $(2,1,2)$, $(1,2,3)$, $(2,1,4)$, $(1,2,5)$, $(2,1,5)$, $(1,2,6)$

通信模式γ见图5-3。在时间k，$0 \leq k \leq 6$，进程1和2的信息级别见图中标号。

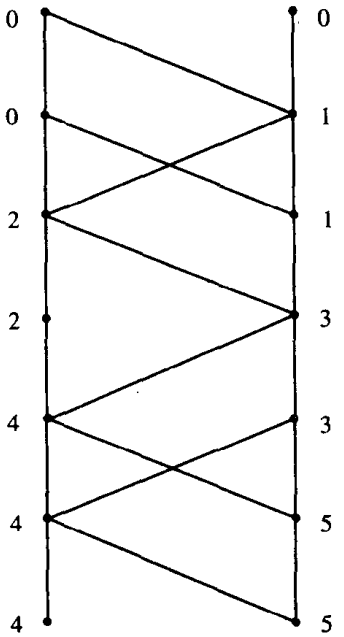


图5-3　良好的通信模式

89

以下的引理说明不同进程的信息级别的差不会超过1。

引理5.2　对于任一良好的通信模式γ，和k，$0 \leq k \leq r$，以及任何的i和j，有$|level_\gamma(i,k) - level_\gamma(j,k)| \leq 1$。

证明　此证明留做一道习题。　□

接下来的引理说明，在所有的信息都传递的情况下，信息级别等于轮数。

引理5.3　如果γ是一个由所有(i,j,k)，$1 \leq k \leq r$，组成的“完整”通信模式，那么对于所有i和k，有$level_\gamma(i,k)=k$。

证明　此证明留做一道习题。　□

随机攻击算法（*RandomAttack*）的思想如下：

***RandomAttack*算法（非形式化）：**

每个进程i根据在运行中产生的通信模式，都明确地把它的级别保存在变量*level*中。进程1从$[1, r]$的整数中选择一个随机的*key*值，这个值附于所有消息中。另外，所有进程的初值也附于所有消息中。

在r轮之后，如果每个进程计算出来的*level*值不少于*key*，且它知道所有进程的初值都是1，那么它决定出1。

***RandomAttack*算法（形式化）：**

消息表由形如(L, V, K)的三元组组成，其中L是一个向量，负责分配$[0, r]$中的整数

给每个进程索引，V是一个向量，负责分配{0，1，*undefined*}中的值给每个进程索引，k要么是[1，r]中的一个整数，要么是未定义（*undefined*）。

***states*$_i$:**

rounds $\in \mathbb{N}$, initially 0
decision $\in \{unknown, 0, 1\}$, initially *unknown*
key $\in [1, r] \cup$ *undefined*, initially *undefined*
for every j, $1 \leq j \leq n$:
  val(j) $\in \{0, 1,$ *undefined*$\}$; initially *val*(i) is i's initial value and
    val(j) = *undefined* for all $j \neq i$
  level(j) $\in [-1, r]$; initially *level*(i) = 0 and *level*(j) = -1 for all $j \neq i$

变量*level*（j）用来记录已被进程i所知（通过消息链）的进程j的最大等级。对于$j \neq i$，在i听到来自j中的消息前，*level*（j）的默认值为 -1。在随机化函数*rand*$_i$中，随机表示使用均匀分布从[1，r]中随机选出一个整数。

90

***rand*$_i$:**

if $i = 1$ **and** *rounds* $= 0$ **then** *key* := *random*

***msgs*$_i$:**

send $(L, V,$ *key*$)$ **to all** j, **where** L **is the** *level* **vector and** V **is the** *val* **vector**

***trans*$_i$:**

rounds := *rounds* + 1
let (L_j, V_j, k_j) **be the message from** j, **for each** j **from which a message arrives**
if, for some j, $k_j \neq$ *undefined* **then** *key* := k_j
for all $j \neq i$ **do**
  if, for some i', $V_{i'}(j) \neq$ *undefined* **then** *val*(j) := $V_{i'}(j)$
  if, for some i', $L_{i'}(j) >$ *level*(j) **then** *level*(j) := $\max_{i'} \{L_{i'}(j)\}$
level(i) := 1 + min $\{$*level*(j) : $j \neq i\}$
if *rounds* $= r$ **then**
  if *key* $\neq$ *undefined* **and** *level*(i) $\geq$ *key* **and** *val*(j) = 1 **for all** j **then**
    decision := 1
  else *decision* := 0

在这些代码中，第三行是用来设置*key*组件。对*key*设置多次并不要紧，因为所有通过的*key*值都是一样的。第五行为不等于i的进程j设置*val*组件，同样不会有赋值冲突。第六行的代码为不等于i的进程j更新*level*组件，希望包含i知道的其他进程的最大级别。下一步，i更新自己的*level*组件，设置成比i所知的其他进程的最小级别大一级。最后，如果这是在最后的第r轮，那么i就根据前面所说的原则做出决定。

定理5.4 *RandomAttack*算法以ε=1/r解决了协同攻击问题的随机化版本。

证明 这个证明的关键就是证明算法能够正确地计算*level*。也就是，在*RandomAttack*算法的任意运行中，在良好的通信模式γ下，对于任何的k，$0 \leqslant k \leqslant r$，和任意$i$，在$k$轮以后，*level*（$i$）$_i$的值等于*level*$_\gamma$（$i$, k）。同样的，在k轮以后，如果*level*（i）$_i \geqslant 1$，那么*key*$_i$已被定义，对于所有j，*val*（j）$_i$也已被定义。而且，它们的值分别和进程1所选择的实际*key*值以及实际初始值相等。

*RandomAttack*算法的终止性是很明显的。对于有效性来说，如果所有的进程都有初始值0，那么很明显决定的值只可能是0。现在假设所有进程以1开始，而且所有的消息都被传递了。那么引理5.3和算法能够正确计算级别的事实就意味着对于任意i，在做出决定的第r轮，*level*（i）$_i$

91

$=r$。因为$level(i)_i=r\geq 1$，所以key_i已被定义，对于所有j，$val(j)_i$也已被定义。因为所有可能的key的值小于等于r，所以1是惟一可能的决定值。

最后，我们考虑一致性。令B为任意对手；我们证明

$$Pr^B[\text{某些进程决定出0且某些进程决定出1}]\leq\varepsilon。$$

对于每个i，令l_i代表当进程i(在r轮内)做决定时的$level(i)_i$值。那么引理5.2意味着所有的l_i值都是相差不超过1的。如果选择的key的值严格大于$max\{l_i\}$，或者某些进程以初值0开始，那么所有进程都决定出0。另一个方面，如果$key\leq min\{l_i\}$，且所有进程以初值1开始，那么所有进程都决定出1。所以只有当$key=max\{l_i\}$时才可能出现不一致。这种情况的概率为$\varepsilon=1/r$，因为$\max\{l_i\}$是由对手B决定的，而且key是在[0，r]中均匀分布的。□

例5.2.2　*RandomAttack*算法的行为

考虑当$n=2$和$r=6$的情况。假设对手B为两个进程提供了输入1，并且具有例题5.2.1中的良好通信模式γ。令$\varepsilon=1/6$。定理5.4说明对手B的不一致概率最多为ε。事实上，这个概率正好是ε：如果进程1所选择的key值为5，那么进程1决定出0而进程2决定出1；如果$key\leq 4$，那么两个都决定出1；如果$key=6$，那么两个都决定出0。

另一方面，如果对手B提供了其他输入组合和通信模式γ，那么不一致的概率为0，因为两个进程都决定出0。

使用定理5.4的证明中的思想，我们可以看出*RandomAttack*满足比我们原来声明更强的有效性条件。也就是说，我们可以展示：

有效性：

1）如果任意进程以0开始，那么0是惟一可能的决定值。

2）对于所有初值都为1的任意对手B，

$$Pr^B[\text{所有进程决定1}]\geq l\varepsilon$$

92　其中l是在r时B中的任意进程的最小等级。

第二个性质对于某些应用来说是很有用的，比如分布式数据库的提交，这样的环境中人们希望有肯定结果。举个例子，如果只有一个消息丢失了，那么协同攻击的概率保证很高，至少有$(r-1)/r$。*RandomAttack*算法能够满足更强的有效性条件，其证明留做一道简单的习题。

5.2.3　不一致的下限

现在我们来展示不存在比定理5.4给出的下限好得多的结果。（回想在上一节我们仅仅考虑n个节点的完全图。）

定理5.5　任何解决随机化协同攻击问题的r轮算法，其不一致概率至少为$1/(r+1)$。

在这一节剩下的部分中，我们假设有一个r轮算法A，它以ε的不一致概率解决了n个节点的完全图中的协同攻击问题。我们来证明$\varepsilon\geq 1/(r+1)$。

为了能证明这个定理，我们需要更多的定义。如果B是任一对手，γ是它的通信模式，i是任意进程，我们定义另一个对手$prune(B,i)$，它“裁剪”i在B中没有收到的信息。$B'=prune(B,i)$，定义如下：

1）如果$(j,0)\leq_\gamma(i,r)$，那么j在B'中的输入和在B中的输入是相同的，否则为0。

2）如果(j,j',k)在通信模式B中且$(j',k)\leq_\gamma(i,r)$，那么(j,j',k)就在通信模式B'中。

这就是说，对手B'包括了B中i知道的所有消息，但是没有其他的；而且B'指出，所有B中i不知道的输入都为0。下面的引理说明对手裁剪版本便足以确定输出的概率分布。

引理5.6 如果B和B'是两个对手，i是一个进程，而且$prune(B, i) = prune(B', i)$，那么$Pr^B[i决定1] = Pr^{B'}[i决定1]$。

证明 此证明留做一道习题。 □

定理5.5的证明依赖于以下引理。 93

引理5.7 令B为所有初值都为1的任意对手，i是任意进程，那么有

$$Pr^B[i决定1] \leqslant \varepsilon(level_B(i, r)+1)$$

证明 对$level_B(i, r)$做归纳。

基础 设$level_B(i, r) = 0$，定义$B'=prune(B, i)$，那么有$prune(B', i) = B' = prune(B, i)$，所以由引理5.6，

$$Pr^B[i决定1] = Pr^{B'}[i决定1] \tag{5-1}$$

因为$level_B(i, r) = 0$，那么一定存在某个进程j，$j \neq i$，使得$(j, 0) \not\sim_\gamma (i, r)$不成立，其中$\gamma$是$B$的通信模式。那么$B'$就规定了$j$的初值为0，而且在它的通信模式不包含目的地为$j$的消息。因此$prune(B', j)$是所有初值为0且通信模式中没有消息的对手。令$B''$来代表这个对手，那么$prune(B'', j) = B'' = prune(B', j)$，所以由引理5.6得

$$Pr^{B'}[j决定1] = Pr^{B''}[j决定1]$$

根据有效性条件

$$Pr^{B''}[j决定1] = 0$$

所以

$$Pr^{B'}[j决定1] = 0$$

但是因为不一致概率最多为ε，所以我们有

$$|Pr^{B'}[i决定1] - Pr^{B'}[j决定1]| \leqslant \varepsilon$$

所以

$$Pr^{B'}[i决定1] \leqslant \varepsilon$$

由式（5-1）得到

$$Pr^B[i决定1] \leqslant \varepsilon$$

得证。

归纳步：设$level_B(i, r) = l > 0$，并设引理对小于1的所有等级都成立。定义$B' = prune(B, i)$.那么引理5.6意味着

$$Pr^B[i决定1] = Pr^{B'}[i决定1] \tag{5-2}$$ 94

因为$level_B(i, r) = l$，由$level$的定义可知，一定有某个进程j使得$level_{B'}(j, r) \leqslant l - 1$。由归纳假设得到

$$Pr^{B'}[j决定1] \leqslant \varepsilon(level_{B'}(j, r)+1) \leqslant \varepsilon l$$

但是因为存在一个最大不一致概率ε，我们有

$$|Pr^{B'}[i决定1] - Pr^{B'}[j决定1]| \leqslant \varepsilon$$

所以

$$Pr^{B}[i\text{决定}1] \leqslant \varepsilon(l+1)。$$

由式（5-2）可得

$$Pr^{B}[i\text{决定}1] \leqslant \varepsilon(l+1)。$$

得证。□

现在我们可以证明定理了。

（定理5.5的）证明　令B为所有输入都是1且没有消息丢失的对手。所有进程都决定出1的最大概率不大于其中任意进程决定出1的概率。如引理5.7所说，最多为$\varepsilon(level_B(i, r)+1) \leqslant \varepsilon(r+1)$。但是有效性条件要求，在这个对手$B$产生的运行中所的进程都必须决定出1；所以所有进程决定出1的概率为1。这意味着$\varepsilon(r+1) \geqslant 1$，也就是$\varepsilon \geqslant 1/r+1$，得证。□

5.3　参考文献注释

协同攻击问题是由Gray[142]提出来的，目的是为了表示分布式数据库的提交问题。该问题的确定性版本的不可能性结果也归功于Gray[142]，而随机化协同攻击的结论来自于Varghese和Lynch[281]的工作。

5.4　习题

5.1 证明：对任意非平凡连接图中（确定性）协同攻击问题的一个解法，意味着在包括两个节点和一条连接它们的边的简单图中，也有一个解法。（因此，协同攻击问题在任何非平凡
95 图中都是不可解决的。）

5.2 考虑以下（确定性）协同攻击问题的变形。假设网络是$n>2$的完全图，终止性和有效性要求和5.1节一样。但是一致性要求削弱为：如果任意进程决定出1，那么至少有两个进程决定出1。（即我们排除一个将军单独进攻的情况，但是允许两个或更多将军一起进攻。）这个问题是否可解？给出证明。

5.3 对两个进程用一条边连起来的简单情况，考虑有链路故障时的协同攻击问题。假设进程是确定性的，但消息系统是概率性的。每条消息都有一个独立的成功传递的概率p，$0<p<1$。（和往常一样，在每轮中我们只允许进程发送一条消息。）

在这种情况下，设计一个算法，它在固定轮数r内终止，最大的不一致概率是ε，同样，违反有效性条件的概率也是ε。尽量得到最小的ε。

5.4 在上题描述的环境中，证明可以得到的ε的下限。

5.5 证明引理5.2。

5.6 证明引理5.3。

5.7 详细证明定理5.4的证明过程中的第一个论断，即*RandomAttack*算法正确地计算*level*值，并正确传递初始值和*key*值。

5.8 对*RandomAttack*问题，证明5.2.2节末尾给出的更强的有效性属性。即证明

a）如果任意一个进程从0开始，那么0是惟一可能的决定值。

b）对初始值是1的任意对手B，Pr^{B}[所有进程决定出1]$\geqslant l\varepsilon$，其中l是B中任意进程在时间r的最
96 小等级。

5.9 扩展协同攻击的随机化版本，使其允许以ε的概率违反有效性条件和一致性条件。根据修改后的问题描述调整*RandomAttack*算法，使其得到尽可能小的ε。做相应分析。

5.10 把*RandomAttack*算法扩展到任意的无向图（不一定是完全图）中，并做分析。

5.11 证明引理5.6。

5.12 把定理5.5的下限结论扩展到任意的无向图（不一定是完全图）中。

5.13如果对手决定的通信模式并不像我们假设的那样是预先确定的，而是在线确定的，那么本章中随机化环境中的结论将会怎样？精确的说，就是对手在决定某一个第k轮传递消息之前，可以检查k轮之前所有的运行。

a）在有任意的在线对手的情况下，*RandomAttack*算法可以保证的不一致界限ε是什么？

b）对于能达到的ε的值，你能证明它的下限吗？

第6章 进程故障下的分布式一致性

在这一章中，我们继续学习在第5章提出的同步模型中的一致性问题。这次我们来考虑进程故障而不是链路故障的情况。当然，讨论物理“处理器”的故障比逻辑“进程”的应该更有道理，但是为了保持本书中的术语一致性，我们就用了进程这个词。我们研究了两种故障模型：停止故障模型（进程会在没有警告的前提下停止）和*Byzantine*故障模型（故障进程可能会出现各种各样的无法预测的错误）。停止故障模型可以用来表示不可预测的进程崩溃，而Byzantine故障模型可以用来表示任意类型的处理器功能紊乱，如处理器中单独组件的故障。

*Byzantine*这个术语首先是在Lamport、Pease和Shostak的一篇论文中被使用，在这篇论文中一致性问题根据*Byzantine*将军们来做形式化。就像在第5章中的协同攻击问题，Byzantine将军们想就是否应该发起进攻达成一致。这次并不用担心消息会丢失，但要注意某些将军的可能叛国行为。*Byzantine*可以作为双关语——在古代Byzantium帝国中的战场，某些叛变的将军的行为也被称为Byzantine。

在本章中考虑的我们称之为一致性问题（*agreement problem*）的特定一致性问题中，进程的输入都是从一个特定的集合V开始的。所有的无故障进程都被要求从同一个值集合V中产生
99 输出，并且符合简单的一致性和有效性条件。（对于有效性来说，我们假设所有的进程都是以相同的值v开始的，那么惟一允许的决定值也是v）。

一致性问题是最初开发飞机控制系统时产生的问题的一个简化版本。在这个问题中，有一个处理器的集合，每个处理器都可以访问一个分离的高度计，其中一些可能发生故障，这些处理器试图就飞机的高度达成一致。Byzantine一致性算法已经被加到容错的多处理器系统的硬件中，其中少数处理器执行相同的计算，就每一步的结果达成一致。这种冗余允许处理器忍受一个处理器的（Byzantine）故障。Byzantine一致性算法在处理器故障诊断中也很有用，它们可以使一个处理器集合就它们中的哪些发生了故障达成一致（所以应该替换或者忽略这些处理器）。

在两个故障模型中，我们需要对进程故障发生频度的界限做出假设。怎样来表达这样的界限呢？在其他的对有处理器故障的系统的分析中，这些限制总是以控制故障发生的概率分布的形式出现。在这里，我们不用概率的形式，而是简单地预先假设一个故障数的上限为一个确定的数字f。这是一个比较简单的假设，因为这样就可以避免讨论复杂的故障发生概率了。在实际中，这个假设可以是实际的，因为不大可能会有超过f个故障发生。但是我们必须记住这个假设是有问题的，在大部分的实际的情况下，如果故障的数目已经很大的话，那么很有可能会发生更多的故障。对故障数目的假设意味着故障是负相关的(*negatively correlated*)，但是，实际上故障往往是相互独立或者正相关的。

在定义了一致性问题后，我们对停止故障和Byzantine故障给出了一系列算法。然后证明解决Byzantine故障所需要的进程数目的下限和解决每种故障类型所需要的轮数的下限。

6.1 问题

我们假设网络是一个有n个节点组成的无向连通图，其中的进程为进程1,...,n，而且每个进程都知道整张图。每个进程开始时都有固定集合V中的一个输入值，放在指定的状态组件中。

我们假设对于每个进程来说，每个开始状态只有包含一个输入值。所有的进程的目的就是，通过把状态中的*decision*状态组件设置为V中的值，最终输出集合V中的结果。我们使用与第3~5章中相同的同步模型，只是这次我们允许其中有限数目（最多为f）的进程发生故障。在本章中，我们假设所有的链路都是极可靠的——所有发送的消息都能被传递。我们来考虑两种不同的进程故障：停止故障和Byzantine故障。 100

在停止故障模型中，在算法执行的任何一点上，进程都有可能停止。特别地，进程有可能在发送消息的过程中停止。也就是说，在进程停止的时候，进程应当发送的消息中只有一部分被实际发送了。在这种情况下，我们假设消息的任何子集都有可能被发送。一个进程有可能在某一轮发送完消息后停止，但是还没有进行那一轮的状态转换。

对于停止故障模型，一致性问题的正确性条件是：

一致性 没有两个进程会决定出不同的值。

有效性 如果所有的进程都是从相同的初始值$v \in V$开始的，那么v就是惟一可能的决定值。

终止性 所有的没有故障的进程最后都要做出决定。

在Byzantine 故障模型中，一个进程故障不仅仅是停止，还有可能出现任意的行为。也就是说，它可能会从任意一个状态启动，而不是从它的开始状态之一启动；有可能发送任意的消息，而不是由消息函数所规定的那些消息；还有可能表现出任意的状态转换，而不像转换函数所规定的那样。（作为技术上的一个方便的特例，我们允许Byzantine进程的行为是完全正常的可能性。）惟一的对故障进程的行为的限制就是，它只能影响到它能够控制的系统组件，也就是它自己发出的消息和它自己的状态。比如，它不能破坏另一个进程的状态，修改或者替换另一个进程的消息。

对于Byzantine 故障模型，一致性和有效性条件就和停止故障模型稍微有点不同：

一致性 没有两个无故障进程会决定出不同的值。

有效性 如果所有无故障进程都以相同的初始值$v \in V$开始，那么对于无故障进程来说，v就是惟一可能的决定值。

终止性 这个条件与以前的相同。

这个修改过的条件反映了在Byzantine模型中，不可能对出错进程如何开始和做出什么决定施加限制。我们把对于Byzantine 故障模型的一致性问题称为*Byzantine一致性问题*。 101

停止故障和Byzantine故障一致性问题之间的关系 不能说能够解决Byzantine一致性问题的算法就一定能够自动解决停止故障的一致性问题。区别在于，在停止模型中，我们要求所有做出决定的进程都必须达到一致，即使那些随后发生故障的进程也必须达成一致。如果停止故障模型的一致性条件被Byzantine 故障的一致性条件代替，那么这个可以满足。或者，如果在Byzantine算法中所有无故障进程都在相同的轮上做出决定，那么算法也对停止故障适用。其证明留做习题。

停止故障的更强有效性条件 另一个有时会用到的停止故障模型的有效性条件如下所述：

有效性 任何进程的决定值都是某个进程的初始值。

容易看出，这个条件意味着我们已经描述过的有效性条件。在第7章对一致性问题的扩展中，我们会在定义k一致性问题时用到这个强化的条件。在这一章中，我们用所给出的弱一点的条件；这样会稍稍减弱算法的结论，并且会稍稍加强不可能性结果。对于这一章的算法，我们会指出它们是否能够满足这个加强的有效性条件。

复杂度 对于时间复杂度来说，我们计算直到所有无故障进程做出决定时经过的轮数。对于通信复杂度来说，我们要计算消息的数目和通信的位数。在停止模型下，我们的计算基于所有进

程发送的消息。在Byzantine模型下，我们仅仅基于无故障进程发送的消息。因为在Byzantine模型中，没有办法对故障进程的通信提供限制。

6.2 针对停止故障的算法

在这一节中，我们在特殊的n节点完全图中，给出针对停止故障模型的一致性算法。我们从一个基本算法开始，在这个算法中每一个进程只是重复地广播它所见到的所有值。我们将继续看到这个基本算法的一些更低复杂度的版本。最后我们给出一个使用了一种叫做指数信息收集（*Exponential Information Gathering*，*EIG*）策略的算法。EIG算法虽然花销大而且有点复杂，但是可以扩展到表现更差的故障模型中。

102

约定　在本节和下节中，我们用v_0来表示输入集合V中的一个默认值。表示V中任意值所需的位数的上界定义为b。

6.2.1 基本算法

对于停止故障的一致性问题，有一个简单的算法叫做*FloodSet*。进程只是传播它们看见的所有的V中的值，而且在最后使用一个简单的规则。

***FloodSet*算法（非形式化）：**

每个进程都有一个变量W，它包含V的一个子集。开始时，进程i的变量W只包含i的初始值。在$f+1$轮中的每一轮，每个进程都广播W，然后把所接收到的元素加到W中。

在$f+1$轮后，进程i使用下面的决定规则。如果W是一个单元素集合（singleton set），那么i就决定出W中的惟一元素；否则i就决定出默认值v_0。

代码如下：

***FloodSet*算法（形式化）：**

V子集所包含的消息字母。

***states*$_i$：**

rounds $\in \mathbb{N}$, initially 0
decision $\in V \cup \{unknown\}$, initially *unknown*
$W \subseteq V$, initially the singleton set consisting of i's initial value

***msgs*$_i$：**

if *rounds* $\leq f$ then send W to all other processes

***trans*$_i$：**

rounds := *rounds* + 1
let X_j be the message from j, for each j from which a message arrives
$W := W \cup \bigcup_j X_j$
if *rounds* $= f + 1$ then
　if $|W| = 1$ then *decision* := v, where $W = \{v\}$
103　else *decision* := v_0

为了检验*FloodSet*算法的正确性，我们用符号$W_i(r)$来表示在进程i中的变量W在r轮后的值。和往常一样，我们用一个下标i来表示一个组件是属于进程i的。如果一个进程在r轮结束以后没有出错，我们说它是*活动的*。

第一个简单引理说明，如果存在一轮其中没有一个进程出错，那么在此轮之后所有活动的进程都有相同的W值。

引理6.1　如果在某一轮r中，$1 \leqslant r \leqslant f+1$，没有一个进程发生故障，那么对于$r$轮以后仍活动的

所有i和j，$W_i(r) = W_j(r)$。

证明 假设没有进程在第r轮出错，令I为在r轮后（或者等价地说，在$r-1$轮之后）所有活动进程的集合。那么因为在I中的每个进程都会发送自己的W集合到别的进程中去，在r轮以后，I中每个进程的W集合就是在r轮前I中所有进程包含的值的集合。 □

下一个结论是，如果在r轮以后所有的活动的进程都有相同的W值，那么在以后的轮中，情况是相同的。

引理6.2 假设对所有在r轮后都是活动的i和j来说，$W_i(r) = W_j(r)$。那么对于任意的r'轮，$r \leqslant r' \leqslant f+1$，有相同的情况。也就是说，对于所有的活动的$i$和$j$，在$r'$轮之后，同样有$W_i(r') = W_j(r')$成立。

证明 此证明留做一道习题。 □

下面的引理在一致性问题中是至关重要的。

引理6.3 如果进程i和j在$f+1$轮后都是活动的，那么在$f+1$轮结束后，有$W_i = W_j$。

证明 因为最多只有f个故障进程，所以必然存在某一轮r，$1 \leqslant r \leqslant f+1$，在这一轮中没有进程发生故障。引理6.1说明，对于所有的活动的i和j，在r轮以后有$W_i(r) = W_j(r)$。那么引理6.2说明对于所有的活动的i和j在$f+1$轮以后$W_i(f+1) = W_j(f+1)$。 □

定理6.4 *FloodSet*算法解决了停止故障的一致性问题。

证明 从决定规则来看，终止性条件是很明显的。对于有效性，假设所有的初始值都是v，那么v就是可以发送到任何地方的惟一值。每个$W_i(f+1)$都是非空的，因为它包含i的初始值。所以每个$W_i(f+1)$都必然是$\{v\}$，所以我们由决定规则得到v是惟一可能的决定。 104

对于一致性来说，让i和j为任意两个做出决定的进程。既然决定只在$f+1$轮之后发生，这就意味着i和j在$f+1$轮之后都是活动的。引理6.3说明$W_i(f+1) = W_j(f+1)$。由决定规则得i和j都做出相同决定。 □

复杂度分析 *FloodSet*算法中，直到所有的无故障进程做出决定，需要$f+1$轮。总共的消息的数目是$O((f+1)n^2)$。每个消息包含了至少n个元素（因为每个元素是某个进程的初始值），所以每个消息的位数是$O(nb)$。所以总共的通信位数是$O((f+1)n^2b)$。

另一个决定规则 *FloodSet*的决定规则有点随意性。因为*FloodSet*算法保证了所有的不出错进程在$f+1$轮后都有相同的集合W值，所以只要所有的进程都使用相同的规则，其他不同的决定规则也会正确工作。举个例子，如果值集合V有一个全序关系，那么所有的进程就可以在W中选择一个最小的值。这个替代规则有一个优点，就是它能够保证在6.1节中提到的加强的有效性条件。*FloodSet*算法的决定规则没有保证这个加强条件，因为默认值v_0可能不是任何进程的初始值。

进程故障与通信故障 *FloodSet*算法表明，对于停止故障，一致性问题是可以解决的。这个结果应该和有通信故障的环境中的协同攻击问题的不可能性结果做对比。（见定理5.1和习题5.1）

6.2.2 减少通信

*FloodSet*算法用到的消息数是$O((f+1)n^2)$，消息位数是$O((f+1)n^2b)$，要减少它的通信量是可能的。比如，通过使用以下的办法，消息的个数可以减少到$2n^2$，通信的位数可以减

少到$O(n^2b)$。注意到在最后，如果$|W_i|=1$，每个进程i只需要知道W_i的确切元素，否则，i仅仅需要知道$|W_i|\geq 2$的事实。所以，如果所有的进程都只广播自己看到的前两个的值，而不是所有的值，结果会更好。这个思想是以下算法的基础：

*OptFloodSet*算法：

105 进程就像*FloodSet*算法一样操作，只是每个进程i最多广播两个值。第一次广播发生在第1轮，每个进程i都广播自己的初始值。第二次广播发生在第r轮，$2\leq r\leq f+1$（如果这样的轮存在的话），其中在第r轮的开始，i知道某个v的值和自己的初始值不同。那么i就广播这个新值v。（如果在这一轮有两个或两个以上新的值，可以选择其中的任意一个广播。）

在*FloodSet*算法中，如果最后的W_i是单元素的集合$\{v\}$，进程i就决定出v，否则就决定出v_0。

复杂度分析　*OptFloodSet*算法的轮数也和*FloodSet*算法一样，是$f+1$。消息的数目最多是$2n^2$，因为每个进程最多发送两条非空消息给其他的进程。通信的位数是$O(n^2b)$。

我们用*模拟关系*（*simulation relation*），通过把*OptFloodSet*算法和*FloodSet*算法相联系，来证明它的正确性（在4.1.3节中我们曾使用相似的策略，通过把*OptFloodMax*与*FloodMax*相联系来证明它的正确性）。这首先需要像*FloodSet*中那样详细的描述*OptFloodSet*算法，包括轮（*round*）、决定（*decision*），以及W变量。我们分别用$W_i(r)$和$OW_i(r)$表示经过r轮后*OptFloodSet*算法和*FloodSet*算法中的W_i。下面的引理描述了*FloodSet*中的消息的广播。

引理6.5　在*FloodSet*算法中，假设i把第$r+1$轮的消息发送给j，j接收到并处理了这个消息。那么$W_i(r)\subseteq W_j(r+1)$。

证明　此证明留做练习。　□

*OptFloodSet*算法的最重要的性质是由以下的引理得到的。

引理6.6　在*OptFloodSet*算法中，假设i把第$r+1$轮的消息发送给j，j接收到并处理了这个消息。那么有

1）如果$|OW_i(r)|=1$，那么$OW_i(r)\subseteq OW_j(r+1)$。

2）如果$|OW_i(r)|\geq 2$，那么$|OW_j(r+1)|\geq 2$。

而且，如果i在开始的r轮中没有出错，而且没有发送第$r+1$轮的消息给j，而只是因为*OptFloodMax*算法并没有指定任何消息要被发送，那么这两个结论也成立。

证明　此证明留做一道习题。　□

现在我们用相同的输入和相同的故障模式，一起运行*OptFloodSet*算法和*FloodSet*算法。也
106 就是说，在两个不同的运行中，相同的进程故障会在相同的轮中发生。而且，如果在一个算法中进程i只发送了一些第r轮的消息，那么在另一个算法中它会把第r轮的消息发送给相同的进程。更准确地说，如果i在一个算法中想要给j发送一个消息但是失败了，那么在另一个算法中也不会成功发送。我们给出联系两个算法状态的不变式断言。

引理6.7　在任意的r轮后，$0\leq r\leq f+1$：

1）$OW_i(r)\subseteq W_i(r)$。

2）如果$|W_i(r)|=1$，那么$OW_i(r)=W_i(r)$。

证明　此证明留做一道习题。　□

引理6.8　在任意的r轮后，$0\leq r\leq f+1$：如果$|W_i(r)|\geq 2$，那么$|OW_i(r)|\geq 2$。

证明 用归纳法证明。$r=0$时的基本情况是正确的。假设现在引理对于r是正确的，那么我们来证明引理对于$r+1$也是正确的。假设$|W_i(r+1)|\geq 2$。如果$|W_i(r)|\geq 2$，那么用归纳假设可以得到$|OW_i(r)|\geq 2$，也就是说$|OW_i(r+1)|\geq 2$。

假设$|W_i(r)|=1$。引理6.7说明$OW_i(r)=W_i(r)$。我们来考虑下面两种情况：

1）对于所有在*FloodSet*算法中发送第$r+1$轮消息给i进程的j，$|W_j(r)|=1$成立。则对于所有的j，我们根据引理6.7有$OW_j(r)=W_j(r)$，所以有$|OW_j(r)|=1$。引理6.6告诉我们对于所有的j，$OW_j(r)\subseteq OW_i(r+1)$。于是我们就有$OW_i(r+1)=W_i(r+1)$，这个足以用来证明归纳步骤。

2）对于一些在*FloodSet*算法中发送第$r+1$轮消息给i进程的j，$|W_j(r)|\geq 2$成立。则由归纳假设得到，$|OW_j(r)|\geq 2$。然后由引理6.6得到$|OW_i(r+1)|\geq 2$。 □

引理6.9 在任意的r轮后，$0\leq r\leq f+1$，轮数和*decision*变量在*FloodSet*算法和*OptFloodSet*算法中都有相同的值。

证明概要 有趣的事情是证明在两个算法中，进程i在第$f+1$轮会做相同的决定。这只需对$r=f+1$和两个算法的决定规则应用引理6.7和引理6.8就行了。 □

定理6.10 *OptFloodSet*算法解决了停止故障模型的一致性问题。 107

证明 由引理6.9和定理6.4（*FloodSet*的正确性理论）可得。 □

减少通信复杂度的其他办法 减少*FloodSet*算法的通信复杂度有其他的方法。比如，如果V有一个全序，那么决定规则就可以修改成在W中简单选择一个最小的值。那么就有可能修改*FloodSet*算法，使得每个节点都只要记住并传递它遇到的最小的值，而不是所有的值。这个算法用了$O((f+1)n^2b)$通信位。这个可以通过把它和（修改了决定规则的）*FloodSet*算法相联系，用模拟关系来证明它的正确性。这个算法可以满足6.1节中提到的加强有效性条件。

6.2.3 指数信息收集算法

在这一节中，我们给出一个解决停止故障一致性问题的策略——*EIG*算法。在*EIG*算法中，进程在几轮中发送和传递初始值，把它们从不同通信链路接收到的值记录在一个特殊的数据结构中，称为*EIG*树。最后，它们根据记录在树中的数据，使用一个通用的决定规则。

无论是从通信的位数还是从本地存储量来说，*EIG*算法对于解决停止故障的一致性问题的开销是很大的。我们给出这个算法的主要原因就是*EIG*树这个数据结构也可以用来解决Byzantine一致性问题，这将在6.3.2节中给出。停止故障模型对这个数据结构给出了一个简单的介绍。我们给出这个算法的第二个原因是，*EIG*算法很容易经过修改，用来解决一个Byzantine的受限形式的一致性问题，称为带鉴别的Byzantine故障（*authenticated Byzantine failure*）模型。

*EIG*算法的一个基本的数据结构就是带标号的*EIG*树，$T=T_{n,f}$，其源自根节点的路径代表初始值在进程中传播的链。所有的链都是由不同的进程组成的。树T有$f+2$层，从第0层（根）开始一直到第$f+1$层（叶子）。在k层上的每个节点，$0\leq k\leq f$，都有$n-k$个子节点。在树T中的每个节点都被标注有一个序号。根节点以一个空串λ标注，每个节点的标号为$i_1\ldots i_k$，每个节点有 108
$n-k$个子节点，标号为$i_1\ldots i_k j$。其中j的取值范围是$\{1,\ldots,n\}-\{i_1,\ldots,i_k\}$。如图6-1所示。

在我们称之为*EIGStop*的针对停止故障的*EIG*算法中，进程在所有可能的路径上传递值。每个进程维护一个*EIG*树$T=T_{n,f}$的副本。计算会持续$f+1$轮。在计算过程中，进程会在k轮后为所

有k层节点标注上V中的值或空值*null*。进程i树的根节点以i的输入值标注。在进程i的树中，如果标号为$i_1 \ldots i_k, 1 \leqslant k \leqslant f+1$的节点有一个标注值$v \in V$，那么就意味着在第$k$轮$i_k$已经告知了$i$，在$k-1$轮$i_{k-1}$已经告知了$i_k$，……，在第1轮$i_1$已经告知了$i_2$这一信息：$i_1$的初始值为$v$。另一方面，如果标号为$i_1 \ldots i_k$的节点有一个标注值*null*，那么意味着通信链$i_1, i_2, \ldots, i_k$，$i$被某个故障打断了。在$f+1$轮后，进程使用各自标注了的树来决定出$V$中的一个值，这基于（以下描述的）通用的一致决定规则。接着我们给出更细致的算法描述。

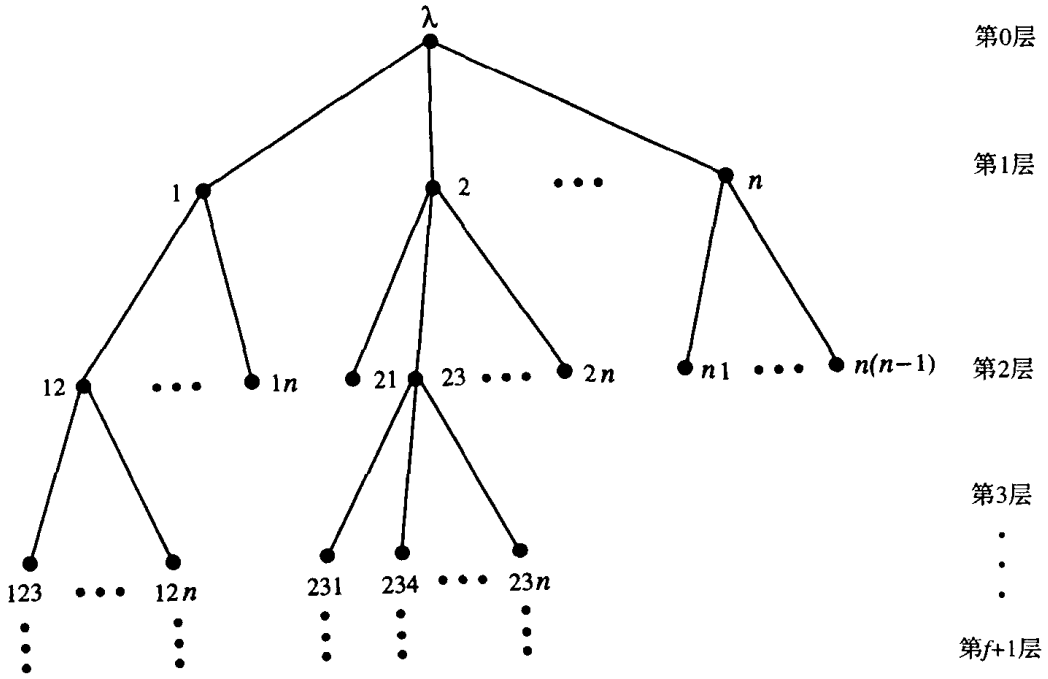


图6-1　*EIG*树$T_{n,f}$

在这个算法的叙述和以后的一些叙述中，我们可以很方便地假装每个进程i除了向其他进
109 程发送信息之外还能向自己发送信息。这可使得算法的描述更加统一。这些消息在这个模型中是不被允许的，但是这样做并无害处，因为虚构的传递可以用本地计算来模拟。

***EIGStop*算法：**

对于每个作为T的节点的标号串x来说，每个进程都有一个变量$val(x)$。$val(x)$用来存放标号为x的节点的标注值。最初，每个进程i把自己的树的根节点标注上自己的初始值，也就是给$val(\lambda)$赋值。

第1轮：进程i对所有的进程广播$val(\lambda)$，包括i自己。然后进程i记录下接收到的信息。

1）如果有一个带着$v \in V$的消息从j到达i，那么i就把$val(j)$设置为v。

2）如果没有带着$v \in V$的消息从j到达i，那么i就把$val(j)$设置为*null*。

第k轮，$2 \leqslant k \leqslant f+1$：进程$i$广播所有数对$(x, v)$，其中$x$是不包含索引$i$的$T$中第$k-1$层的标号，$v \in V$，且$v = val(x)$[⊖]。然后进程$i$就记录下接收到的信息。

1）如果xj是在T中第k层节点的标号，其中x是一个进程索引串，j是一个单独索引，且有一

⊖ 为了适合我们的形式化模型，其中每一轮只有一条消息从i发送到其他每个进程，我们把目的地址相同的所有消息包装在一起，构成一条大的消息。

个从j来到i告知$val(x)=v\in V$的消息，则i就会把$val(xj)$设置为v。

2）如果xj是第k层节点的标号，但没有从j来到i的带有V中值的消息，那么i就会把$val(xj)$设置为$null$。

在$f+1$轮之后，进程i应用一个决定规则。令W为i的树中节点的非空值的集合。如果W是一个单元素集合，那么i的决定值是W中的惟一值；否则，i决定出v_0。

不难看出，树会被标注上我们先前指出的值。也就是说，进程i的根节点被标注上i的输入值，而且如果标号为$i_1\ldots i_k$，$1\leqslant k\leqslant f+1$的节点有一个标注值$v\in V$，那么就意味着在第$k$轮$i_k$已经告知了$i$、在$k-1$轮$i_{k-1}$已经告知了$i_k$，……，在第1轮$i_1$已经告知了$i_2$这一信息：$i_1$的初始值是$v$。而且，如果标号为$i_1\ldots i_k$，$1\leqslant k\leqslant f+1$的节点有一个标注值$null$，那么在第$k$轮$i_k$没有将$i_1\ldots i_{k-1}$的值发送给$i$。 110

例 6.2.1 *EIGStop*的运行

这是一个*EIGStop*算法如何运行的例子，考虑有三个进程（n=3）的情况，其中一个可能是有故障的（f=1）。那么就会执行两轮，树有三层。这棵*EIG*树$T_{3,1}$结构如图6-2所示。

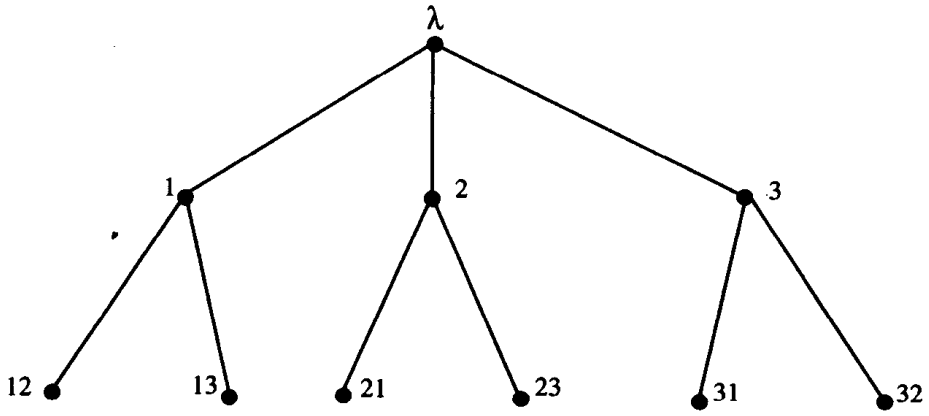


图6-2 *EIG*树$T_{3,1}$的结构

假设进程1、2、3的初始值分别为0、0、1。假设进程3出了故障，而且它是在第一轮发送消息给进程1之后和发送消息给进程2之前出错。那么三个进程的树就如图6-3所示。

注意，在第2轮收到从1发来的消息之前，进程2并不知道进程3的初始值是1。

为了看清*EIGStop*算法的正确性，我们首先给出两个引理，它们给出不同树中的值的联系。第一个引理给出了初始化和树中相邻层上不同进程的*val*的关系。

引理6.11 在*EIGStop*算法中，$f+1$轮后，以下成立：

1）$val(\lambda)_i$是i的输入值。

2）如果xj是一个节点的标号而且$val(xj)_i=v\in V$，那么$val(x)_j=v$。 111

3）如果xj是一个节点的标号而且$val(xj)_i$为null，那么或者$val(x)_j$为null，或者j在第$|x|+1$轮给i发送消息失败。

证明 此证明留做一道习题。 □

第二个引理描述了在树中不一定相邻的层上的*val*的联系。前两种情况追溯树中任意地方出现的值的来源。第三种情况是技术性的，说明树中出现的任意值v，一定在树中某个标号不包含索引i的节点上出现。简略的说，这意味着进程i第一次得知的值不会是它向自己传播值的结果。

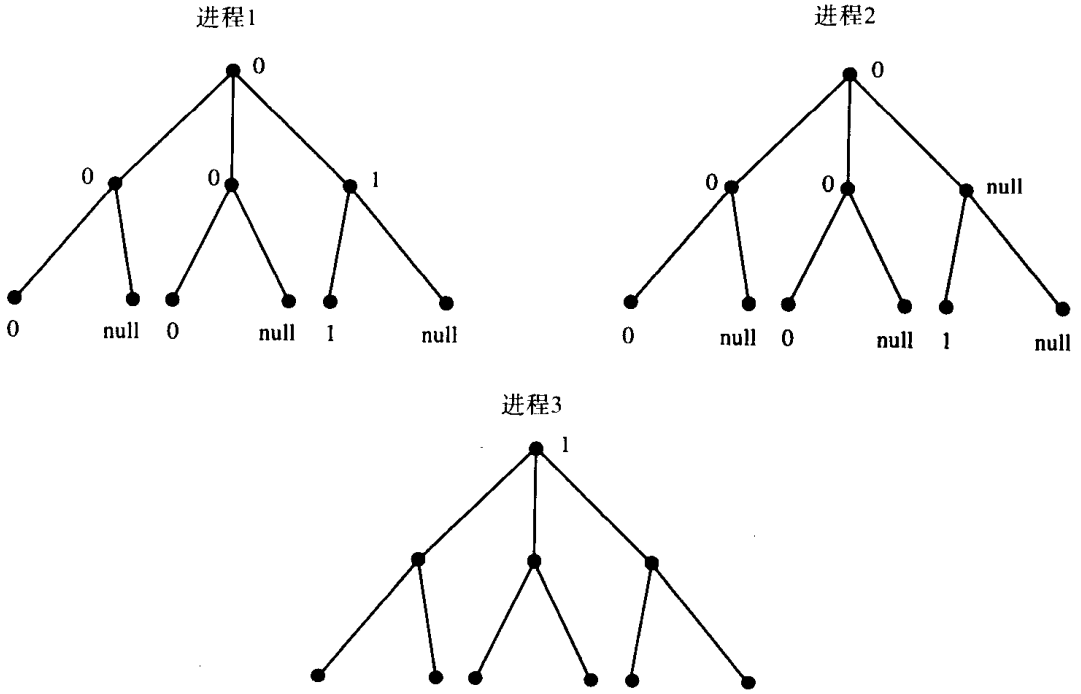


图6-3　*EIGStop*的运行；进程3在第一轮失败

引理6.12　在*EIGStop*算法中，$f+1$轮后，以下成立：

1）如果y是一个节点的标号，$val(y)_i = v \in V$，而且xj是y的前缀，那么$val(x)_j = v$。

112　2）如果$v \in V$出现在任意进程的val中，那么对于某些i来说$v = val(\lambda)_i$。

3）如果$v \in V$出现在进程i的val集合中，那么一定有一个不包括i的标号y，使得$v = val(y)_i$。

证明　第一部分可以再次使用引理6.11的第二部分得到。

对于第二部分，假设$v = val(y)_i$。如果$y = \lambda$，得证。否则，让j成为y中的第一个索引。第一部分证明了$v = val(\lambda)_j$。

对于第三部分，我们做相反的假设，v只是包含i的标号的val值，令y是满足$v = val(y)_i$的最短标号。那么y有形如xi的一个前缀。但是第一部分意味着$val(x)_i = v$，这就和y的选择相矛盾了。　□

下面一个引理提供了一致性的关键。

引理6.13　如果进程i和j都是不出故障，那么$W_i = W_j$。

证明　我们可以假设$i \neq j$，我们从两方面来证明

1）$W_i \subseteq W_j$

假设$v \in W_i$，那么引理6.12意味着对于一些没有包含i的标号x来说，$v = val(x)_i$成立。我们考虑两种情况。

a）$|x| \leqslant f$

那么$|xi| \leqslant f+1$，那么因为串x没有包含索引i，（没有故障的）进程i就会在第$|xi|$轮把v传递给进程j。这就是说$val(xi)_j = v$，所以$v \in W_j$。

b）$|x| = f+1$

因为最多只有f个故障进程和所有在x中的索引是不同的，所以一定有某个无故障进程l，它

的索引是在x中出现的。因此，x有形如yl的前缀，其中y是一个串。那么引理6.12意味着$val(y)_l = v$。既然进程l是没有故障的，那么它就在第$|yl|$轮给进程j传递消息。所以$val(yl)_j = v$，所以$v \in W_j$。

2）$W_j \subseteq W_i$

和前面的情况对称。

两种情况合在一起证明了需要证明的等式。 □

例 6.2.2 证明引理6.13中的情况

例6.2.1说明了两种情况，a）和b），在引理6.13的证明中也这样考虑。进程1首先在第一轮给自己的树标志上1，就像在情况a）中那样。进程2在第二轮给自己的树标志上1。特别地， 113
$val(3)_1 = 1$，所以$val(31)_2 = 1$。

另一方面，在最后一轮，即第二轮，进程2首先给自己的树标志上1，设置$val(31)_2 = 1$。这就意味着某个无故障进程索引，在这种情况下是1，一定会出现在节点的标号中。在情况b）中，1出现在进程1树的节点31上。也就是$val(31)_2 = 1$，所以$val(31)_1 = 1$。

定理6.14 *EIGStop*算法解决了停止故障的一致性问题。

证明 从决定规则来看，终止性是显然的。

对于有效性条件，假设所有的初始值都是v，那么由引理6.12，标识任意进程的树的值只可能是v或者为*null*。每个W_i都是不空的，因为它都包含了i的初始值。因此，每个W_i必须是$\{v\}$，所以根据决定规则得，v是惟一可能的决定值。

对于一致性条件来看，令i和j是任意两个做出决定的进程。因为决定只是在最后才做出，也就是说i和j是正确的。那么由引理6.13得，$W_i = W_j$。根据决定规则得i和j都做出了相同的决定。□

复杂度分析 轮数是$f+1$，发送得消息数是$O((f+1)n^2)$。（这种计算把任意进程在任意轮中发送给任意其他进程的组合消息作为单个消息。）通信位数与故障数呈指数关系是$O(n^{f+1}b)$。

其他决定规则 因为*EIGStop*算法保证了相同的W出现在无故障进程的树中，其他的决定规则也可以正确地起作用。举个例子，如果V有一个全序，那么所有的进程就可以在W中选择一个最小值。和以前一样，这有一个优点，就是可以保证6.1节中提到的加强的有效性条件。

就像我们在*FloodSet*算法中做的那样，可以减少*EIGStop*算法的通信开销。和以前一样，在$|W_i| = 1$的情况下，每个进程i只需要知道W_i的元素。所以进程就可能只需要广播它们遇到的开始的两个值。

***OptEIGStop*算法：**

进程就像在*EIGStop*算法中一样操作，只是每个进程i最多广播两个值。第一次广播是在第一轮，进程i广播它的初始值。第二次广播是在第一个r轮，$2 \leqslant r \leqslant f+1$，使得在第$r$轮的开始， 114
进程i知道了某个和自己的初始值不同的v值（如果任意这样的轮存在）。于是i就会广播这个新的v值和在$r-1$层上任何标志上v的节点x。（如果有两个以上的(x, v)可以选择，那么可以任选一个。）

就像在*EIGStop*算法中一样，令W为标志i树节点的非空*val*的集合。如果W是一个单元素集合，i就决定出W的惟一元素；否则，i决定出v_0。

复杂度分析 *OptEIGStop*算法用了$f+1$轮。消息的个数最多是$2n^2$，因为每个进程最多只能发送两个非空的消息给其他的进程。通信位数是$O(n^2(b+(f+1)\log n))$，每个消息的值部分使用$O(b)$位，标号部分使用$O((f+1)\log n)$位。

*OptEIGStop*算法的正确性可以通过联系*EIGStop*算法做一道习题，用模拟方法来证明。证明过程和*OptFloodSet*算法的正确性证明过程差不多。此外，给出将*OptEIGStop*和*OptFloodSet*联系的正确的证明，具体的就留做一道习题。

6.2.4 带鉴别的Byzantine一致性

虽然*EIG*算法在本节仅仅被设计成仅能容忍停止故障，它们也能解决一些更差类型的故障。它们不能应付Byzantine故障模型的全部困难，其中进程可能出现随机的行为。但是它们能够应付一种有趣的Byzantine的受限模型，其中进程有特殊的能力，可以通过数字签名来鉴别通信。所谓进程i的一个数字签名，就是进程i可以应用在任意出向消息上的一种转换，用来证明消息的确是来自进程i的。其他的进程在没有i的合作下都不能产生i的数字签名。在现代通信网络中，数字签名是一个应当具备的能力。

我们没有一个形式化的带鉴别的Byzantine模型的定义（事实上我们并没有一个完美的定义），只是非形式化地描述它。在这个模型中，假设任何进程都可以用数字签名来鉴别自己发送的消息。从字面上，通常假设有某个公共的数据源，初始值从它产生，并由它签名。这里，
115 我们假设每一个非故障进程从一个初始状态开始，其中包含单个被数据源签名的输入值；而故障进程从某个状态开始，其中包含某个被数据源签名的输入值集合。故障进程可以发送任意的消息和实行任意的状态转换，惟一的限制是它不能产生非故障进程或数据源的签名。

这个模型满足的正确性条件有：Byzantine一致性问题的终止性和一致性条件，再加上如下的有效性条件。

有效性　如果所有进程都从一个初始值$v \in V$开始，并被数据源签名，那么v就是无故障进程的惟一可能决定值。

不难看出做出如此改动（所有的消息被签名，而且只有被正确签名的消息才可以被接收）的*EIGStop*算法和*OptEIGStop*算法可以解决带鉴别的Byzantine故障模型的一致性问题。证明过程就和停止故障模型的证明过程差不多，我们将其留做一道习题。

6.3 针对Byzantine故障的算法

在本节中，我们给出Byzantine一致性问题的算法，对于特殊的n节点完全图来说。我们从使用*EIG*的一个算法开始。然后我们来说明，一个对于二值集合$V = \{0, 1\}$上的解决Byzantine一致性问题的算法，如何用做解决一般值集合V上的Byzantine一致性问题的“子过程”。最后我们来讨论减少了通信复杂度的Byzantine一致性算法。

所有算法都有的一个共同特点就是，它们使用的进程的数目要大于出错进程数目的三倍，$n>3f$。这个情况与我们在停止故障中看到的不同，因为那里没有对n和f的关系做特定的要求。这个进程界限反应了Byzantine故障模型所增加的难度。事实上，我们将在6.7节中可以看到该性质是固有的。这个乍看起来可能有点奇怪，因为你可能想象，使用某种多数选举算法，$2f+1$个进程可以容许f个Byzantine故障。（有一种标准的容错技术称为模3冗余，其中任务数乘以3，占多数的结果被接收。你可能以为这种方法可以解决只有一个故障进程的Byzantine一致性问题，但你将会看到它不能。）

116 6.3.1 举例

在我们给出*EIG* Byzantine一致性算法前，我们先来说明一下为什么Byzantine一致性问题会比停止故障的一致性问题更难。特别地，我们给出了一个例子，表明（不是证明）如果三个进程中的一个可能发生故障，则Byzantine一致性问题不能被解决。

假设进程1、2和3解决了Byzantine一致性问题，允许其有一个出错。假设它们在第二轮结

束后决定，而且它们用一个特殊的方法来处理：在第一轮，每个进程都简单地广播自己的初始值，在第二轮每个进程向其他的进程报告它们在第一轮中接到了第三个进程的什么信息。考虑以下运行：

运行α_1：

进程1和2是正确的，而且初始值都为1，而进程3是出错的，而且初始值是0。在第一轮中，所有的进程都正确地广播自己的初始值。在第二轮中，进程1和2正确地报告它们在第一轮中所接收的消息，但进程3（错误地）告诉进程1“进程2在第一轮发送了0”，或做出正确的报告。图6-4显示了在运行a_1中发送的消息。在这个运行中，有效性条件要求进程1和2都决定出1。

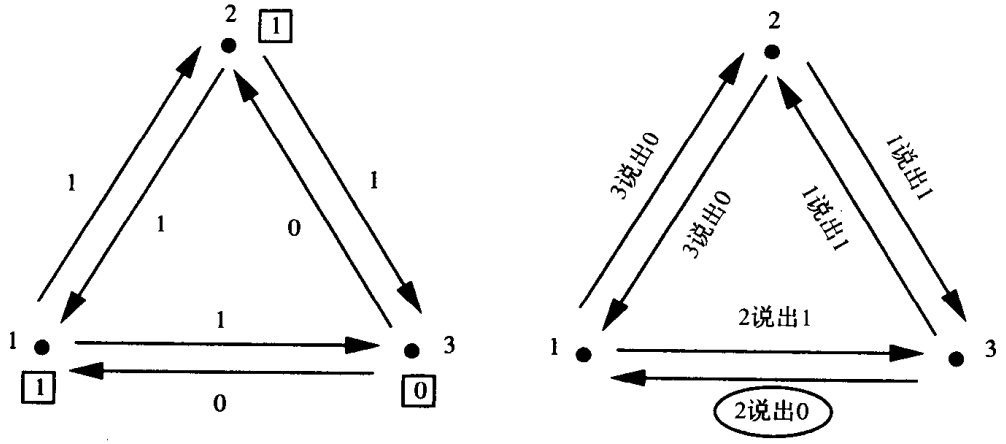


图6-4 运行α_1——错误消息被圈出

现在考虑第二种运行情况：

117

运行α_2：

这个和a_1相对称。这次，进程2、3是无故障进程，而且初始值为0，进程1发生故障，初始值为1。第一轮时，所有进程都正确地报告自己的值。第二轮时，进程2、3正确地报告它们在第一轮中所接收到的消息，但进程1（错误地）告诉进程3“进程2在第一轮给它发送了1”，或做出正确的报告。图6-5显示了在运行α_2中发送的消息。在这个运行中，有效性条件要求进程2和3都决定出0。

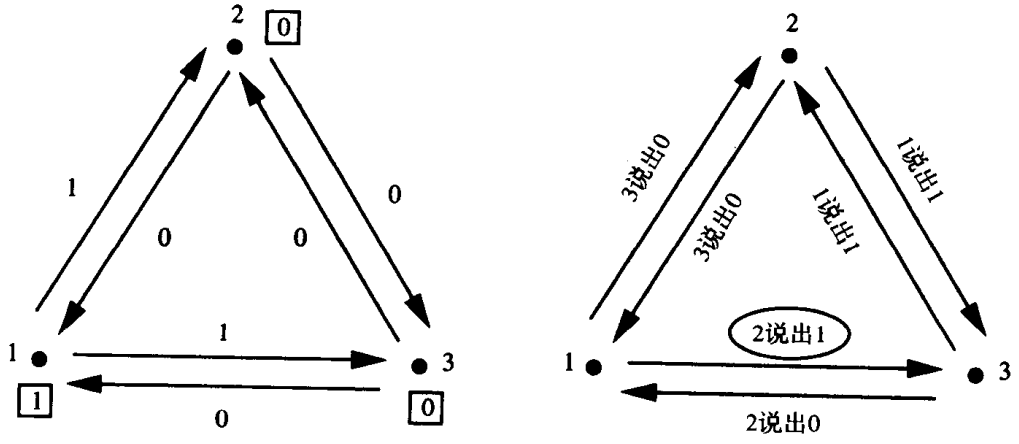


图6-5 运行α_2——错误消息被圈出

为了能得到矛盾，我们来考虑第三种运行情况。

运行α_3：

现在假设进程1、3都是无故障的，初始值分别是1和0。进程2发生故障，它告诉进程1其初始值为1，并告诉进程3其初始值为0。所有的进程的行为在第二轮中都是正确的。图6-6显示了这种情况。

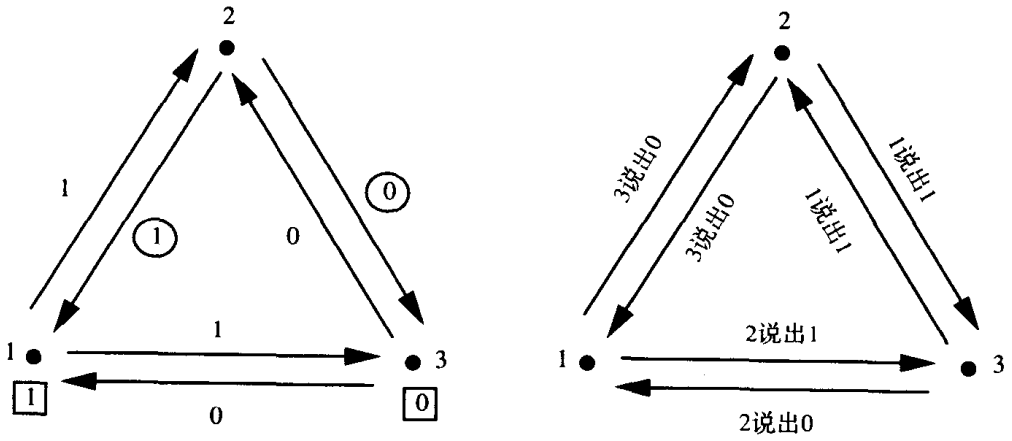


图6-6　运行α_3——冲突消息被圈出

注意到进程2在运行α_1和运行α_3中，发送了相同的消息给进程1，而在运行α_3和运行α_2中发送了相同的消息给进程3。事实上，很容易可以知道运行α_1和运行α_3对于进程1来说是不可区分的，即$\alpha_3 \overset{1}{\sim} \alpha_1$。运行$\alpha_3$和运行$\alpha_2$对于进程3来说也如此，即$\alpha_3 \overset{3}{\sim} \alpha_2$。既然进程1在运行$\alpha_1$中决定出1，它在运行$\alpha_3$中也如此，而且进程3在运行$\alpha_2$中决定出0，它在运行$\alpha_3$中也如此。但是这样就违反了运行$\alpha_3$的一致性条件，这就和进程1、2、3能解决Byzantine一致性问题产生了矛盾。我们已
118 经表明了没有一个简单形式的算法可以解决Byzantine一致性问题。

注意进程1在运行α_3中可以说出某个进程出错，因为进程2告诉进程1说它的值是1，但是进程3告诉进程1说进程2的值是0。问题就是进程1无法说出进程2和3到底哪个出错。

这个例子不能证明：三个进程，其中一个可能发生故障，不能解决Byzantine一致性问题。因为以上的讨论预先假设算法仅使用两轮，并且发送特定类型的消息。但是有可能把它扩展到更多的轮数和任意的消息类型。事实上，在第6.4节我们将会看到，这里的思想可以被扩展为，在有f个故障出现的情况下，需要有$n>3f$个进程才能解决Byzantine一致性问题。

6.3.2　Byzantine一致性问题的EIG算法

我们现在给出了Byzantine一致性问题的*EIG*算法，我们把它叫做*EIGByz*。不像*EIGStop*算法，*EIGByz*算法预先假设了进程数目相对比故障数目大很多，特别有$n>3f$。这个是必须的，因为有6.31节和6.4节中所描述的限制。在你读这个算法之前，我们建议你自己试着构造一个特殊情况下的算法，比如$n=7$，$f=2$。

*EIGByz*算法对于有f个故障的n个进程，使用与*EIGStop*相同的*EIG*树数据结构$T_{n,f}$。传播策略本质上和*EIGStop*算法使用的是一样的；惟一的区别就是进程收到错误形式的消息后会把它改成看起来是正确的。然而，决定规则是大不相同的，进程不再相信出现在树中任意地方的值。
119 现在进程必须采取措施来屏蔽掉那些错误消息传来的值。

***EIGByz*算法：**

进程在$f+1$轮中传播消息就像在*EIGStop*算法中一样，但有以下的例外：如果进程i曾经从其他进程j那里收到一个不是规定形式的消息（例如，它包含垃圾或包含j的树中相同节点的重

复值），那么进程i就会丢弃那个消息，就像进程j在那一轮没有给进程i发送过消息一样。

在$f+1$轮的最后，进程i调整自己的val，使得任何空值都被默认值v_0代替。

然后做出决定，进程i在它的进程调整并做了标注的树中，从叶子节点向上遍历，用一个$newval$标注每一个节点，过程如下：对每一个标注为x的叶子，有$newval(x):=val(x)$。对每一个标注为x的非叶子节点，$newval(x)$定义为x的大多数子节点持有的$newval$值，也就是，如果存在大多数形式为xj的节点满足$newval(xj)=v$，$v\in V$，则$newval(x)=v$。如果没有这样的大多数节点存在，则进程i令$newval(x):=v_0$。进程i的最终决定是$newval(\lambda)$。

为了证明$EIGByz$算法的正确性，我们从一些预备断言开始。第一个说的是所有无故障进程都会就无故障进程直接传来的值达成一致。

引理6.15 $EIGByz$算法中，在$f+1$轮以后，有以下结论成立。如果i,j和k都是无故障进程，$i\neq j$，那么对于所有的以k结尾的标号x来说，有$val(x)_i=val(x)_j$。

证明 如果$k\notin\{i,j\}$，那么结论就会由以下事实得到：因为k是无故障的，它会在第$|x|$轮给i和j发送相同的消息。如果$k\in\{i,j\}$，那么可以从进程将值传给自身的假设得出结论。 □

下一个引理声明，所有无故障进程就某些节点计算出来的$newvals$们达成一致，其中节点的标号以无故障进程索引结尾。

引理6.16 在$EIGByz$算法中，在$f+1$轮以后，以下结论成立。假设x是一个以无故障进程的索引作为结尾的标号，那么对于所有无故障进程i来说，存在一个值$v\in V$，使得$val(x)_i=newval(x)_i=v$。 120

证明 通过对树的标号进行归纳，从树的叶子节点开始向上进行，也就是从长度为$f+1$的标号一直到长度为1的标号。

基础：假设x是叶子，也就是说$|x|=f+1$。那么引理6.15意味着所有无故障进程i都有相同的$val(x)_i$；称之为公共值v。而且对于每个无故障进程i，由$newval$的定义可知$newval(x)_i=v$。所以v是所需的值。

归纳步：假设$|x|=r$，$1\leqslant r\leqslant f$。那么引理6.15意味着所有无故障进程i都有相同的值$val(x)_i$；称之为v。因此，在第$r+1$轮的时候，每个无故障进程l都会发送相同的x的值v到所有进程，所以对于所有无故障进程i和l来说，有$val(xl)_i=v$。那么根据归纳假设，对于所有无故障进程i和l，$newval(xl)_i=v$成立。

我们现在可以说节点x的大部分子节点的标号都是以无故障进程的索引来结尾的。这是正确的，因为x的子节点的个数是$n-r\geqslant n-f$。因为我们已经假设了$n>3f$。这个数字必须严格大于$2f$。因为最多有f个子节点的标号以故障进程的索引结尾，所以我们有所需的大多数节点。

故此，对于任意无故障进程i，对节点x的多数子节点xl，有$newval(xl)_i=v$。那么对于所有无故障进程，算法中大多数的规则意味着$newval(x)_i=v$。所以v是所需的值。 □

现在我们来讨论有效性。

引理6.17 如果所有的无故障进程都以相同的初始值$v\in V$开始，那么对于一个无故障进程来说，v是惟一可能的决定值。

证明 如果所有的无故障进程都是以v开始的，那么所有的无故障进程就会在第一轮广播v，所以对于任意的无故障的i和j来说，$val(j)_i=v$。引理6.16意味着对于任意无故障进程i和j，有

$newval(j)_i = v$。那么算法中使用的多数规则决定了，对所有的无故障进程i，$newval(\lambda)_i = v$。所以，i的决定值就是v。□

为了说明一致性，我们还需要两个定义。首先，一个有根树的节点的子集C被称为*路径覆盖*，是指从根节点到叶子节点的每条路径上都至少包含C中的一个节点。

第二，考虑*EIGByz*算法的任意一个运行α。如果α中f+1轮之后，所有的无故障进程i有相同的$newval(x)_i$，那么树节点x被称为*公共的*。如果α中一个树的节点集合中的所有节点是公共的，那么树节点集合（比如一个路径覆盖）被称为*公共的*。注意引理6.16意味着，如果i是正确的，
121 那么对任意x，xi是一个公共节点。

引理6.18　在*EIGByz*算法的任意运行α中，在f+1轮过后，存在一个公共的路径覆盖。

证明　令C为形如xi的节点的一个集合，其中i是无故障的。就像上面所说的，所有C中的节点都是公共的。为什么C是一条路径覆盖呢？考虑任意一条从根节点到叶子节点的路径。它包含有正好$f+1$个非根节点，而且根据T的构造，每个这样的节点都以一个不同的进程索引结尾。因为最多只有f个故障进程，所以一定有某个节点的标号是以非故障的进程索引结尾的。这个节点一定在C中的。□

下面的引理显示了公共的节点怎样在树中传播。

引理6.19　在*EIGByz*算法中，在$f+1$轮以后，有以下的结论成立。令x为*EIG*树中任意节点的标号。如果在以x为根节点的子树中有一个公共的路径覆盖，那么x就是公共的。

证明　对树标号用归纳法，从叶节点开始证明。

基础：假设x是叶子，那么覆盖x的子树的惟一的路径就是由x自己组成的。所以x是公共的。

归纳步：假设$|x| = r$，$0 \leqslant r \leqslant f$。假设$x$的子树中有一个公共的路径覆盖。如果$x$本身在$C$中的话，那么$x$就是公共的。所以假设$x \notin C$。

考虑x的任意子节点xl。因为$x \notin C$，对xl为根节点的子树，C产生了一个公共的路径覆盖。所以由归纳假设得到，xl是公共的。因为xl是x的任意选择的子节点，所以所有的x的子节点都是公共的。那么$newval(x)$的定义就意味着x是公共的。□

作为一个简单的结果，我们得到

引理6.20　*EIGByz*算法中，在$f+1$轮以后，根节点λ是公共的。

证明　可以由引理6.18、6.19直接得到。□

我们现在把这些引理合在一起形成主要的正确性定理。

定理6.21　在n个进程、f个故障进程、$n>3f$的情况下，*EIGByz*算法能够解决Byzantine一致性问题。

证明　终止条件是显而易见的。有效性条件可以由引理6.17得到。一致性条件可以由引理6.20
122 和决定规则得到。□

复杂度分析　这个算法的开销和*EIGStop*算法一样，共有$f+1$轮、$O((f+1)n^2)$的消息和$O(n^{f+1}b)$的通信位数。另外增加的要求是，进程数目相对于故障数目很大，$n>3f$。

6.3.3 使用二元Byzantine一致的一般的Byzantine一致性问题

在这一节中，我们来说明怎样将一个解决了输入在{0，1}中的Byzantine一致性问题的算法作为解决一般的Byzantine一致性问题的“子过程”。额外开销为2轮、$2n^2$消息数，但只需总共$O(n^2b)$位的通信。这会导致通信所需位数的极大节省，因为执行子过程的时候，它不需要发送V中的值，仅仅需要发送二进制的值。然而，这个改进并不足以将通信位数从指数级减小到f的多项式级。

根据算法的设计者，我们把它叫做*TurpinCoan*算法。这个算法假设$n>3f$。像以前一样，我们假设每个进程都可以像把消息发送给其他进程一样发送给自己。

*TurpinCoan*算法：

每个进程都有局部变量x、y、z和$vote$，其中x被初始化为进程的输入值，y、z和$vote$被初始化为任意值。

第1轮：进程i给所有进程（包括自己）发送x的值。如果在本轮收到的消息中，有不小于$n-f$个特定值$v \in V$的副本，那么i将设置成$y:=v$，否则$y:=null$。

第2轮：进程i给所有的进程发送y的值，包括它自己。如果在本轮收到的消息中，有不小于$n-f$个V中特定值的副本，那么i将设置$vote:=1$，否则就设置$vote:=0$。同样进程i把z设置成i在本轮接到的所有消息中最常出现的非空值。如果所有的消息都是空的，那么z就保持为未定义的。

第r轮：$r \geq 3$。进程运行二元Byzantine一致子例程，使用$vote$作为输入值。如果在子例程中进程i决定了1，并且z已经有定义，那么算法的最后的决定值就是z。否则就是默认值v_0。

*TurpinCoan*算法的关键事实是：

引理6.22 在第二轮中至多有一个$v \in V$的值被无故障进程发送。 123

证明 为了构造矛盾，假设进程i和j在第二轮的时候分别发送了包含v和w的消息，而且v和w都属于V，且$v \neq w$。那么进程i接收到了至少$n-f$个第一轮的包含v的消息。因为最多只有f个故障进程，而且无故障进程在第一轮向所有进程发送相同的消息。所以进程j一定是接收到至少$n-2f$个包含v的消息。因为$n>3f$，也就是说j接收到了至少$f+1$个包含v的消息。

但是，因为进程j在第二轮发送了w，进程j接收了至少$n-f$个第一轮的包含w的消息。所以一共有$(f+1)+(n-f)>n$条消息。但是进程j收到的第一轮消息总共只有n个，所以这是一个矛盾。 □

定理6.23 如果给定一个二元Byzantine一致性算法作为子例程，且$n>3f$，*TurpinCoan*算法能解决一般Byzantine一致性问题。

证明 终止条件很容易看到。

为了说明有效性条件，我们必须证明，如果所有的无故障进程都以相同的初始值v开始，那么所有的无故障进程都会决定出v。所以假设所有的无故障进程都以v开始。那么有不少于$n-f$个无故障进程在第一轮向所有进程成功地广播了包含有v的消息。所以在第一轮，所有无故障进程把它的y设置成了v。然后在第二轮，每个无故障进程接收到了至少$n-f$个包含v的消息，也就是说将会把z设置成v，把$vote$设置成1。因为所有无故障进程都用输入值1作为二元Byzantine一致子例程的输入，所以根据二元算法的有效性条件，它们的子例程的最后决定的都是1。这就是说它们都在主算法中决定了v，这就证明了有效性。

最后，我们来看一致性。如果子例程的决定值为0，那么v_0就被所有无故障进程选择为最后的决定值，默认地保持了一致性。

假设子例程的决定值是1，那么根据它的有效性条件，某个无故障进程i必须从$vote_i = 1$开始子例程。这意味着进程i收到了至少$n-f$个第2轮包含特定值$v \in V$的消息。因为最多只有f个出故障的进程i，所以进程i接收到了至少$n-2f$个第二轮的包含v的从无故障进程来的消息。如果j是任意的无故障进程，那么j必然也收到至少$n-2f$个第二轮的包含v的从那些相同无故障进程来的消息。根据引理6.22，在第二轮中，除了v，V中没有其他的值被任何无故障进程发送，所以进程j收到的第二轮的不是v的V中的值不多于f个（它们必须是从故障进程来的）。因为$n>3f$，我们
124 有$n-2f>f$，所以v是j收到的第二轮消息中最经常出现的值。因此在第二轮中，j设置$z:=v$。因为子例程的决定值为1，这意味着j决定v。因为这里的讨论对任意无故障进程j都成立，所以一致性成立。□

在*TurpinCoan*算法的证明中，对故障进程的数目的限制用来得到一个运行中不同的进程的视图之间的相似性。这样的讨论同样也出现在其他的一致性算法的证明中，比如在7.2节中的近似一致性算法中。

复杂度分析　轮数是$r+2$，其中r是二元Byzantine一致性子例程所用的轮数。*TurpinCoan*算法用的额外的通信加上子例程用的，是$2n^2$个消息，每个至多b位，一共有$O(n^2b)$位。

6.3.4　减少通信开销

虽然*TurpinCoan*算法在一定程度上可以减少Byzantine一致的通信位的复杂性，但是它的开销仍然与故障数f成指数关系。在Byzantine模型中，与故障数成多项式关系的算法比在停止failure模型中更难得到。在这一节中，我们给出了一个例子，就时间复杂性来讲，这个算法不是最佳的，但是它确实比较简单，而且用了一些有趣的技巧。这个算法是针对在{0，1}中的一个值上的Byzantine一致的特殊情况。6.3.3节中的结论说明了怎样使用这个算法，来得到一个对于一般值域的多项式的算法。

这个算法用了一个一致广播的机制。这种机制是保证不同进程接收的消息满足一定量的一致性的一种方法。用了这种机制，进程i在第r轮可以广播一条(m, i, r)形式的消息，而且在以后的任意轮中，消息可以被任意进程接收（包括i自己）。一致广播的机制要求满足以下的三个条件：

1）如果无故障进程i在第r轮广播了消息(m, i, r)，那么消息就在第$r+1$轮被所有无故障进程接收（也就是说，它在第r轮或第$r+1$轮被接收）。

2）如果无故障进程i没有在第r轮广播消息(m, i, r)，那么(m, i, r)就不会被任何无
125 故障进程所接收。

3）如果在第r轮，任意一个消息(m, i, r)被任意的无故障进程j接收，那么在第$r'+1$轮，它就会被所有的无故障进程接收。

第一个条件说的是无故障进程的广播会被很快的接收，而第二个条件说的是没有一个消息会被错误地发给无故障进程。第三个条件说的是一个被无故障进程接收的消息（无论来自无故障还是出错的发送者）应该立刻就会被其他的所有的无故障进程所接收。

一致广播的机制可以很容易地实现。

***ConsistentBroadcast*算法：**

为了在第r轮广播(m, i, r)，进程i在第r轮给所有的进程发送消息（“*init*”，m，i，r）。如果进程j在第r轮从进程i接收到了（“*init*”，m，i，r）消息，那它就在第$r+1$轮发送（“*echo*”，m，i，r）消息给所有的进程。

如果在第r'轮以前，$r' \geq r+2$，进程j就从至少$f+1$个进程接收到了（“*echo*”，m，i，r）消息，那么进程j在第r'轮发送（“*echo*”，m，i，r）消息（如果它以前没有这样做过的话）。

如果在第r'轮最后，$r' \geq r+1$，进程j从至少$n-f$个进程接收到了（“*echo*”，m，i，r）消息，那么进程j在第r'轮接收通信（如果它以前没有这样做过的话）。

定理6.24 如果$n>3f$，一致广播的算法解决了一致广播的问题。

证明 我们来验证这三个性质。

1）假设无故障进程i在第r轮广播消息（m，i，r）。在r轮，进程i就发送（“*init*”，m，i，r）给所有的进程，而且不小于$n-f$个无故障进程中的每一个进程都在第$r+1$轮向所有的进程发送（“*echo*”，m，i，r）消息。那么在第$r+1$轮最后，每个无故障进程从至少$n-f$个进程接收到（“*echo*”，m，i，r）消息并且接收这个消息。

2）如果无故障进程i在第r轮没有广播（m，i，r）消息，那么它就不发送（“*init*”，m，i，r）消息，所以就没有无故障进程发送（“*echo*”，m，i，r）消息。那么就没有无故障进程接收消息，因为接收消息要求从至少$n-f>f$个进程收到*echo*消息。 126

3）假设消息（m，i，r）被无故障进程j在第r'轮接收。那么在r'轮之前，j从至少$n-f$个进程接收到了（“*echo*”，m，i，r）消息。在这$n-f$个进程中，至少有$n-2f \geq f+1$个无故障进程。因为无故障进程对所有进程都发送相同的消息，所以每个无故障进程在r'轮之前至少接收到了$f+1$个（“*echo*”，m，i，r）消息。这就说明了每个无故障进程在第$r'+1$轮之前发送了（“*echo*”，m，i，r）消息，所以在第$r'+1$轮之前，每个进程至少接收到了$n-f$个（“*echo*”，m，i，r）消息。所以在第$r'+1$轮之前，消息被所有的无故障进程接收。 □

复杂度分析 一个消息的一致广播用了$O(n^2)$个消息。

现在我们来描述一个简单的二元Byzantine一致性算法，其中对它的所有通信都使用一致广播。这个算法叫做*PolyByz*算法。它只发送初始值为1的信息。它对广播消息使用增长的阈值。

***PolyByz*算法：**

这个算法在$f+1$个阶段中完成，每一阶段都由两轮组成。所有（使用一致广播）发送的消息都是（1，i，r）的形式，其中i是进程的索引，r是一个奇数的轮数。也就是说，消息只是在每一阶段的第一轮中发送，而且发送的惟一的信息就是1。

进程i广播消息的情况如下：在第一轮，如果进程i的初始值就是1，它广播一个消息（1，i，1）。在第$2s-1$轮的时候，就是s阶段的第一轮，其中$2 \leq s \leq f+1$，如果进程i在$2s-1$轮以前，已经从至少$f+s-1$个不同的进程接收了消息，并且i并没有广播消息，进程i只广播消息（1，i，$2s-1$）。

在$2(f+1)$轮的最后，如果在这之前进程i已经接收了至少$2f+1$个不同进程的消息，进程i决定值为1。否则，进程i决定值为0。

定理6.25 如果$n>3f$，*PolyByz*算法解决了二元Byzantine一致性问题。

证明 终止条件是显而易见的。

对于有效性条件，有两种情况。第一种，如果所有的无故障进程都以1开始，那么至少有$n-f \geq 2f+1$个进程在第一轮广播。根据一致广播的第一个性质，在第二轮之前所有无故障进程都接收这些消息，使得在第二轮最后，每一个无故障进程至少从$2f+1$个不同的进程接收了消息。这足以说明每一个无故障进程决定值为1。 127

另一方面，如果所有的进程从初始值0开始，那么没有无故障进程做广播。这是因为，能

引起广播的最小的接收数目是$f+1$，如果先前没有无故障进程的广播，这个条件不可能达到。（这里我们使用了一致广播的第二个性质。）这就意味着每一个无故障进程决定值为0。

最后，我们讨论一致性。假设无故障进程i决定出1，只要证明其他的每一个无故障进程也决定出1就够了。因为i决定出1，在$2(f+1)$轮最后，i必须从至少$2f+1$个不同的进程接收消息。令I为其中无故障进程的集合，那么$|I| \geq f+1$。

如果I中所有进程初始值都为1，那么根据一致广播的第一个性质，它们在第一轮做出广播，并且所有的无故障进程在第二轮之前接收这些消息。所以在第三轮[⊖]之前，每一个无故障进程至少从$f+1$个不同的进程接收了消息，这足以引发第三轮的广播。同样根据一致广播的第一个性质，在第四轮之前，所有的无故障进程接收了这些消息。所以在第四轮最后，每一个无故障进程至少从$n-f \geq 2f+1$个不同的进程接收了消息，因此决定出1。

另一方面，假设I中的一个进程，比如说j，没有初始值1。那么j必然在某个$2s-1$轮做出广播，其中$2 \leq s \leq f+1$，这意味着在$2s-1$轮之前，j至少从$f+s-1$个不同的进程接收了消息；而且这些消息中没有一个是从j自己发出的。根据一致广播的第三个性质，在第$2s-1$轮最后，所有这$f+s-1$个进程被所有的无故障进程接收，同时根据第一个性质，在第$2s$轮最后，j广播的消息被所有的无故障进程接收。因此，在第$2s$轮最后，每一个无故障进程至少从$(f+s-1)+1=f+s$个不同的进程接收了消息。

现在有两种情况。如果$s=f+1$，那么在$2(f+1)$轮最后，每一个无故障进程至少从$2f+1$个不同的进程接收了消息，这足以保证它们都决定出1。另一方面，如果$s \leq f$，那么每一个无故障进程在第$2s+1$轮之前接收了足够多的消息，使得它如果没有做过广播，就在第$2s+1$轮做出广播。
128 那么根据一致广播的第一个性质，在第$2s+2$轮最后，所有的无故障进程从所有的无故障进程接收消息。同样，这足以保证它们都决定出1。 □

复杂度分析　*PolyByz*算法要求有$2f+2$轮。最多有n个广播，每个要求$O(n^2)$消息，所以消息数是$O(n^3)$。因为消息包含有进程的索引，每个消息的位数是$O(\log n)$。所以总共的位的复杂度为$O(n^3 \log n)$。

与带鉴别的Byzantine故障模型的关系　如果给普通的二元Byzantine模型增加一个一致广播能力，就产生了一个类似于6.2.4节讨论的带鉴别的Byzantine故障模型。但是两个不是完全相同的。例如，一致广播只用于广播，不用于发送单个消息。更重要的是，一致广播不会防止进程i广播一条（错误的）消息，这个消息声称无故障进程j以前发送了一条消息；无故障进程都会接收这条消息，即使这条消息的内容是错误的。在带鉴别的Byzantine故障模型中，数字签名可以使它立即拒绝这样的消息。但是，尽管这两个模型是不同的，一致广播有足够强大的能力使得它可以用于在普通Byzantine模型中实现一些为带鉴别的Byzantine故障模型设计的算法。

6.4 Byzantine一致性问题中进程的个数

对于存在停止故障甚至Byzantine故障的完全网络图中的一致性问题，我们已经给出了一个算法来解决。你可能已经注意到这些算法的开销都比较大。对于停止故障模型来说，我们给出的最好的算法就是*OptFloodSet*算法，它需要$f+1$轮、$2n^2$个消息和$O(n^2 b)$的通信位数。对于Byzantine故障模型的情况，*EIGByz*算法用了$f+1$轮和一个指数级的通信量，而*PolyByz*算法用了$2(f+1)$轮和一个多项式级的通信量。两个Byzantine一致性算法都要求$n>3f$。

⊖　我们假设$f \geq 1$，使得的确存在第三轮。

在这一章的剩余部分，我们将会指出这些高开销不是无关紧要的。首先，在这一节中，我们说明$n>3f$这个限制对于解决任意的Byzantine一致性问题都是必须的。后面的两节包含了相关的结论：6.5节描述了在一个非完全图网络中为了解决Byzantine一致性问题所需要的连接量。 129
6.6节说明$n>3f$这个界限可以扩展到比Byzantine一致性问题较弱的问题中。本章的最后一节说明，即使对于简单的停止故障情况来说，轮数的下界$f+1$也是必要的。

为了证明在故障数为f的情况下，$n\leqslant 3f$的进程数不能解决Byzantine一致性问题，我们先给出一个最简单的特殊例子：就是在有一个可能故障进程的情况下，三个进程不能解决Byzantine一致性问题。这个结果在6.3.1的例子中已经说明了，尽管那个例子不能构成证明。接下来我们对于任意的n和f，$n\leqslant 3f$，通过把问题减小为只有三个进程的情况，给出一般的结论。

引理6.26 三个进程中有一个进程发生故障的情况下，不能解决Byzantine一致性问题。

证明 用反证法。假设有一个三进程算法A可以解决Byzantine一致性问题，三个进程分别为进程1、2、3，其中一个可能为故障进程。我们用算法A的两个副本来建立一个新的系统S，并显示S必定会表现出矛盾的行为。也就是说，算法A是不存在的。

特别地，我们取A中每个进程的两个副本，并把它们配置在一个六角形的系统S中。开始时，我们把1、2、3进程的一个不重要的副本的输入值设为0，另一个重要副本的输入值设为1。如图6-7所示。

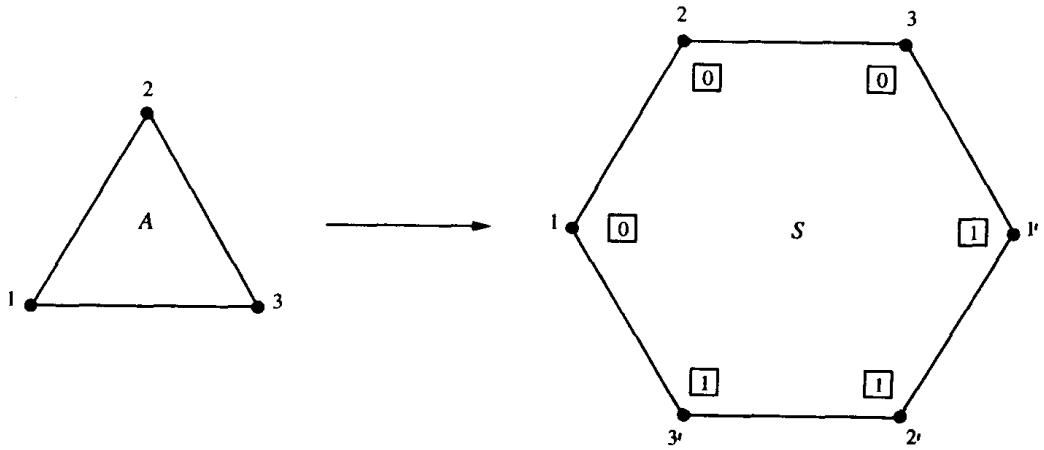


图6-7 合并A的两个副本得到S

什么是系统S呢？形式化的讲，它是一个同步系统，建立在一个六角形的网络上，属于第2章中的通用模型。注意这不是用来解决Byzantine一致性问题的系统——我们并不关心它做了什么，事实上，它只是某种类型的同步系统。我们不会在S中考虑任何故障进程的行为。 130

记住，在我们作为Byzantine一致性问题的解法的系统中，我们假设进程都“知道”整个网络的情况。比如，在A中，进程1知道2和3的名字，并且预先假定有三个节点，叫做1、2、3，被安排在三角形中。在S中，我们并没有假设进程知道整个（六角形）网络图的情况，但是每个进程都有对于它的邻居的本地名字。比如，在S中，进程1知道它有两个邻居，并且知道名字是2和3，即使其中一个的名字是3'。它不知道在网络中有两份副本。这种情况也和第4章的差不多，每个进程只知道网络中的本地情况。特别地，注意S中的网络对于每个进程来说就像A中的网络一样。

我们不要求系统S表现任何特殊的行为。但是注意，S有了特殊的输入赋值就会展现出某种

定义良好的行为。我们会说明，对于以上指出的特定的输入的赋值，这样的定义良好的行为是不存在的，从而得到一个矛盾。

假设S中的进程开始时有图6-7中的输入值，也就是带撇的进程是1，不带撇的进程是0。令α为S的运行结果。

我们首先从进程2、3的角度考虑运行α。对于进程2和3来说，就好像它们在一个三角形系统A中运行一样，在这个运行α_1中，进程1发生故障的。也就是说，根据2.4节中定义的“不可区分”，α和α_1对于进程2和3来说是不可区分的，即 $\alpha \overset{2}{\sim} \alpha_1$ 和 $\alpha \overset{3}{\sim} \alpha_1$，如图6-8所示。在$\alpha_1$中，进程1表现了一个特殊的故障行为，就像它是进程1'、2'、3'的结合一样，而且进程1在α中。尽管这种行为很特殊，但是根据Byzantine故障的假设，A中的一个故障进程允许有这样的行为。

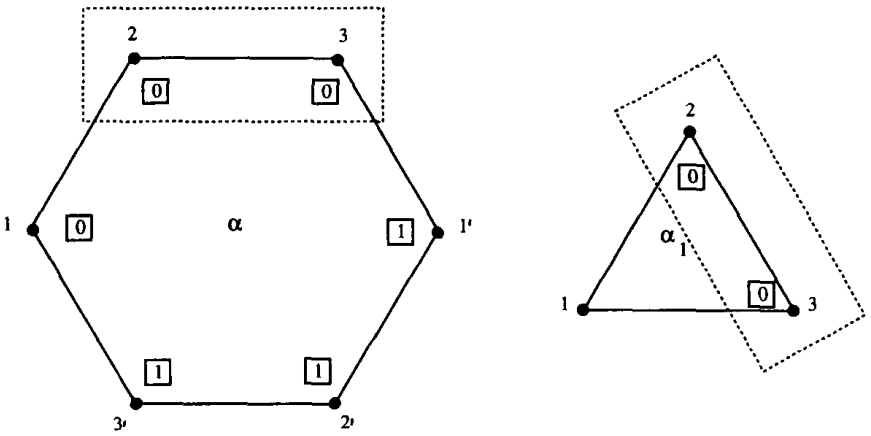


图6-8　对于进程2和3来说α和α_1是不可区分的

因为α_1是A的一个运行，其中进程1发生故障，进程2和3都以0开始，而且因为我们假设A是可以解决Byzantine一致性问题的，所以Byzantine一致性的正确性条件就意味着在α_1中，进程2和3必然决定出0。因为α和α_1对于进程2和3是不可区分的，所以在α中两个都决定出0。

接下来从进程1'和2'的角度考虑运行α。对于进程1'和2'来说，就好像它们在一个三角系统A
131 中运行一样，在这个运行α_2中，进程3发生故障。也就是 $\alpha \overset{1'}{\sim} \alpha_2$，$\alpha \overset{2'}{\sim} \alpha_2$。如图6-9所示。由如上的讨论，在$\alpha$中进程1'和2'最终决定出1。

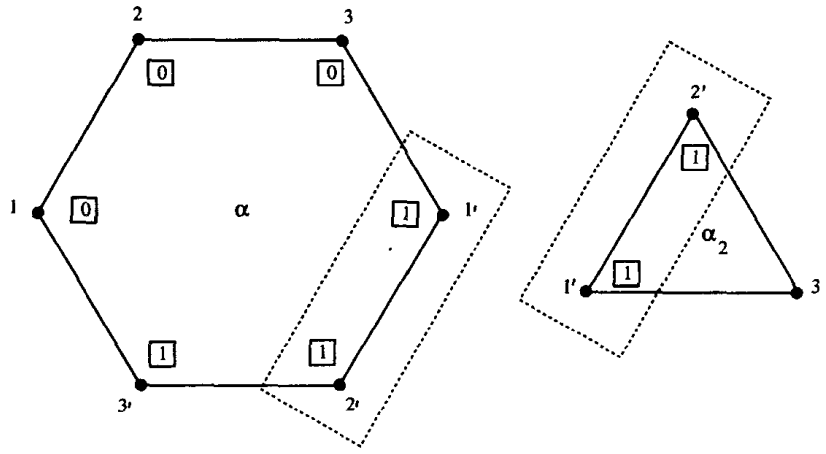


图6-9　α和α_2对于进程1'和2'来说是不可区分的

最后从进程3和1'的角度考虑运行α。对于进程3和1'来说，好像它们在三角系统A中运行，

在这个运行α_3中，进程2发生故障。也就是$\alpha \overset{3}{\sim} \alpha_3$，$\alpha \overset{1}{\sim} \alpha_3$，如图6-10所示。根据Byzantine一致性问题的正确性，进程3和1'必须最终在α_3中做出决定，而且它们的决定必须是相同的。因为进程3开始的输入值是0而进程1'的是1，并没有要求它们决定于什么值，但是一致性条件要求它们必须要一致。因此，它们也在α中决定了相同的值。 132

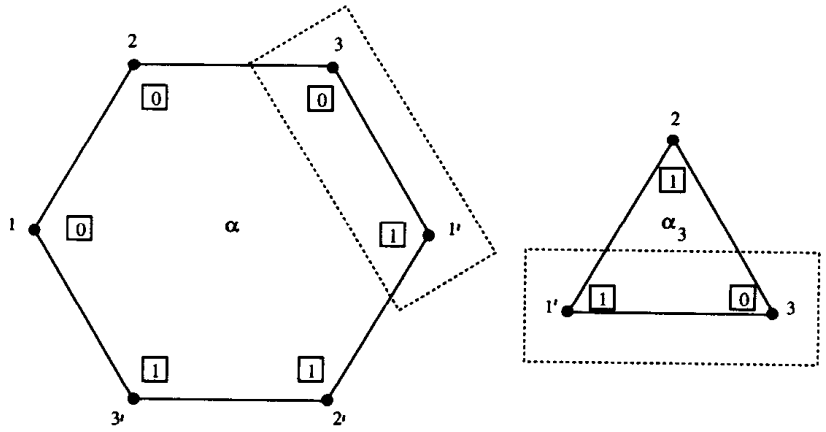


图6-10 α和α_3对于进程1和3来说是不可区分的

但是这是一个矛盾，因为在α中，进程3决定出0而进程1'决定出1。

我们现在用引理6.26来说明，当有$n \leqslant 3f$个进程时，Byzantine一致性问题是不可能解决的。我们使用的方法是：如果存在对于$n \leqslant 3f$个进程的能容许f个Byzantine故障的解法，暗示着存在对于三个进程的能容许一个Byzantine故障的解法，这与引理6.26矛盾。

定理6.27 如果一共有n个进程，其中有f个Byzantine故障进程，而且$2 \leqslant n \leqslant 3f$，那么Byzantine一致性问题是不可解的。

证明 先考虑当$n = 2$时的特殊情况，容易看到，这个问题是不能解的。简单的说，假设一个进程是以0开始，而另一个进程是以1开始，那么每个进程都存在着必须允许另一个进程出错且决定自己的值的可能性。但是如果没有一个出错，那么就违反了一致性的性质。所以我们可以假设$n \geqslant 3$。

为构造矛盾，假设存在一个Byzantine一致性问题的解法A，而且$3 \leqslant n \leqslant 3f$。我们来看怎样把$A$转换成$B$，$B$是针对三个进程1、2和3（其中一个发生故障）的Byzantine一致性解法。B中的三个进程中的每个进程会近似地模拟A中的三分之一的进程。 133

特别地，我们把A中的进程分成三个非空的子集I_1、I_2、I_3，每个进程的大小至多是f。我们让在B中的每个进程i模拟在I_i中的进程。如下所示：

B：

每个进程i保持I_i中所有进程的状态，把它的初始值分配给I_i的每个成员，模拟I_i中所有进程的每个步骤，以及I_i中进程对之间的消息。从I_i中的进程发到另一子集中的进程的消息就从i发送到模拟那个子集的进程。如果I_i中任意被模拟的进程决定了值v，那么i就决定为v。(如果有多于一个这样的值，i就任意选择一个。)

我们来说明B能够正确地解决对于三个进程的Byzantine一致性问题。指定A中的故障进程就是B中故障进程模拟的那些进程[⊖]。固定B的某个特定的运行α，其中最多有一个故障进程，

⊖ 为了适合这里的分类，我们使用的技术是允许Byzantine故障进程的行为完全正确。

并且令α'为A的模拟运行。因为B的每一个进程最多模拟A的f个进程，所以α'中最多有f个故障进程。因为假设A对于最多f个故障的n*个进程*，能解决Byzantine一致性问题，所以Byzantine一致性问题中一般的一致性、有效性和终止性条件在α'中成立。

我们说明这些条件可以在α中满足。对终止性，令i为B中的一个无故障进程。那么i至少模拟A中的一个进程j，并且因为i是无故障的，j也必然是无故障的。α'的终止性条件意味着j最终必然做出决定。只要j做出了决定，i就会做出决定（如果它以前没有做过的话）。

对于有效性，如果B中所有的无故障进程从一个v值开始，那么A的所有的无故障进程也会从v开始。α'的有效性意味着对α'中一个无故障进程来说，v是惟一的决定值。那么对α中一个无故障进程来说，v是惟一的决定值。

对于一致性，假设i和j是B中的无故障进程，那么它们仅仅模拟A中的无故障进程。α'的一致性意味着所有这些被模拟的进程是一致的，因此i和j也是一致的。

我们得到的结论是，B能够解决三个进程的Byzantine一致性问题，可以容许一个故障。但
134 是，这与引理6.26矛盾。 □

6.5 一般图中的Byzantine一致性问题

本章到目前为止只在完全图中考虑了一致性问题。对于n节点完全图来说，我们在6.3节和6.4节中说明，Byzantine一致性问题只能在$n>3f$的情况下才能解决。在本节中，我们考虑在一般网络图中的Byzantine一致性问题。我们描述了其中问题可解图的特征。

首先，如果网络图是一个有至少三个节点的树，那么即使只有一个故障进程，我们也不能期待它能够解决Byzantine一致性问题，因为任意非叶子节点故障进程都可能把树中的一部分进程和其他的进程“断开”。不同部分中的无故障进程甚至不能可靠地联系，更不用说达成一致了。相似的，如果f个节点能够断开网络图，那么有f个故障进程的Byzantine一致性问题就无法解决。

把这个直觉形式化，我们用到了在图论中的以下概念。图G的*连通性*，$conn(G)$被定义为一个最小的节点数：如果移去这些节点，图形就变成非连通的图或者平凡的只有一个节点的图。如果$conn(G) \geq c$，就称图G是c连通的。

例6.5.1 连通性

任何至少含有两个节点的树都是1连通的，一个n个节点的完全图的连通性是$n-1$。图6-11显示了一个2连通的图。如果节点2和节点4移走了，那么就只剩下两个孤立的节点1和3了。

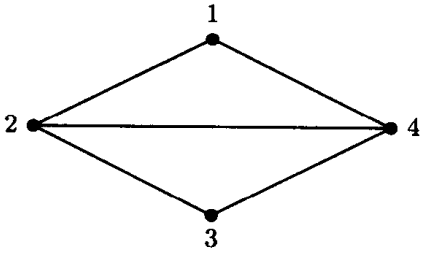


图6-11　$conn(G)=2$的图G

我们使用图论中一个叫做*Menger*定理的经典定理。

定理6.28（Menger定理） 图G是c连通的，当且仅当G
135 中的每对节点之间都有至少c条节点不相交的路径。

现在我们可以来描述那些可能解决给定故障数的Byzantine一致性问题的图。其特征就在于图的节点数和连通性。这个特征中不可能性部分的证明方法，和在6.4节中用于证明故障进程数的下限的方法很相似。

定理6.29 Byzantine一致性问题可以在一个n节点的网络中解决，并且允许有f个故障，当且仅当以下的两个条件都得到满足：

1）$n>3f$

2）$conn(G)>2f$

证明　在定理6.27中，我们已经证明了，要在完全图中解决Byzantine一致性问题，就必须有$n>3f$。那么在一个任意的（不必是完全的）网络图中，不难想像也是需要$n>3f$。这是因为一个非完全图中的$n\leqslant 3f$的算法也可以在一个n节点完全图中运行。

下面我们给出充分性的证明，即Byzantine一致性问题在$n>3f$和$conn(G)>2f$的条件下是可能解决的。因为G是$2f+1$连通的，那么由Menger定理和引理6.28得，在G中的任意两个节点间必定存在着至少$2f+1$条节点不相交的路径。所以在任意一对正确节点i和j之间，i可以通过它和j之间的$2f+1$条路径发送消息，从而可能实现可靠的通信。因为最多只有f个故障进程，j沿着大多数路径接收到的消息一定是正确的。

一旦我们可以在每对无故障进程之间建立可靠的通信，那么我们就可以模拟n节点完全图中的可以解决问题的任意算法，从而解决Byzantine一致性问题。上面给出的可靠通信的实现办法是用来代替完全图中点对点通信的。当然，复杂度会有所增加，但是那不是这里要解决的问题，我们需要的是算法能够正确地工作。

我们现在转到证明中最有趣的部分，说明Byzantine一致性问题只能在$conn(G)>2f$的情况下解决。为了简化问题我们只讨论$f=1$的情况，对更大的f值的（相似）讨论留做一道习题。

假设有一个图，其中$conn(G)\leqslant 2$，在有一个故障的情况下，Byzantine一致性问题不能用算法A解决。那么在G中有两个节点或者断开G，或者可以缩减成一个节点。但是如果它们缩减成一个节点，也就意味着G只有三个节点，我们已经知道，在有一个故障的情况下，Byzantine一致性问题在三个节点的图中是不能解决的。所以我们可以假设这两个节点断开G。 136

那么这个图就一定如图6-11所示，只是有可能节点1和3被任意的连通子图代替，在节点2和4与两个连通子图之间可能有一些边。（节点2和4之间的链路也可能出错，但这会使问题更复杂。）同样为了简单起见，我们只考虑节点1和3是单独的节点的情况。我们通过合并A的两个副本来建立一个系统S。开始时，每一个进程的一个副本的输入值为0，另一个副本的输入值是1，如图6-12所示。正如引理6.26的证明中所说，给定输入的S会有定义良好的行为。同样我们会指出这样的良好行为是不可能的。

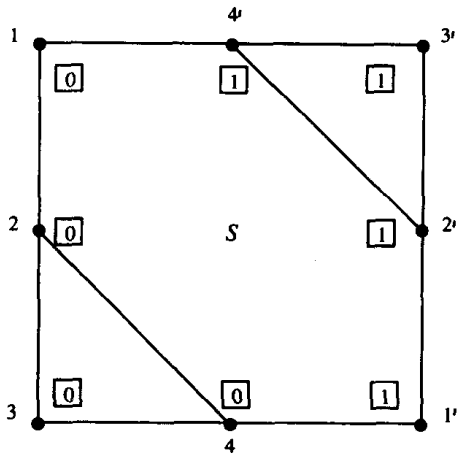


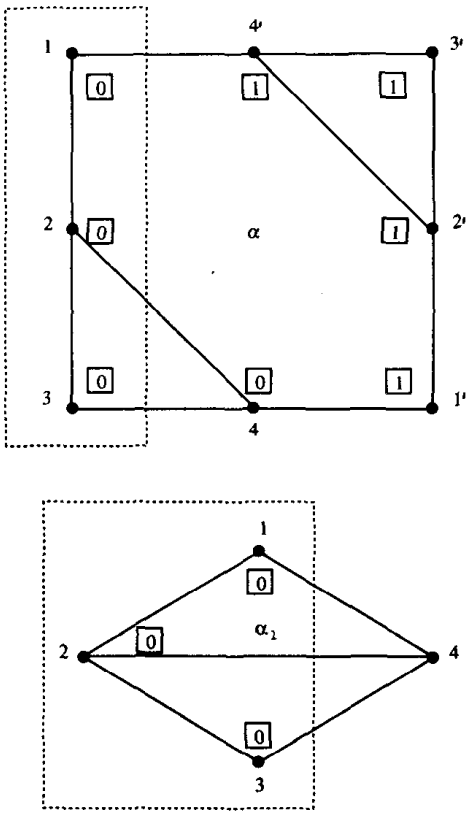
图6-12　把A的两个副本合并得到S

假设S中的进程都以图6-12中指示的输入值开始，也就是不带撇的进程为0，带撇的进程为1。令α为S的一个运行。

我们从进程1、2、3的角度考虑α。这些进程好像在系统A的运行α_1中运行，其中进程4发生故障。如图6-13所示。那么Byzantine一致的正确性条件意味着在α_1中，最后进程1、2和3都必须决定出0。因为对进程1、2、3来说，α和α_1是不可区分的，所以三个进程都会在α中最终决定出0。

下面从进程1'、2'、3'的角度考虑α。三个进程好像在系统A的运行α_2中运行，其中进程4发生故障，如图6-14所示。由相同的讨论，进程1'、2'、3'最终都决定出1。

最后我们从进程3、4和1'的角度考虑α。这些进程好像在系统A的运行α_3中运行，其中进程2发生故障，见图6-15所示。那么Byzantine一致性问题的正确性意味着，在α_3中，最终进程1、2和3都必须做出同一决定。在α中有同样的情况。


1
4'
3'
2
α
2'
3
4
1'
0
1
1
0
0
1
0
0
1
0
1
1
0
1
1
0
1

但这是一个矛盾，因为我们已经说明了，在α中，进程3必须决定出0而进程1'必须决定出1。所以在$conn(G) \leqslant 2$和$f=1$的情况下Byzantine一致性问题是不可解决的。

为了能够得到$f>1$时的结论，我们可以使用相同的图，其中进程2和4被最多有f个节点的集合I_2和I_4代替，进程1和3被任意的节点集合I_1和I_3代替。移去I_2和I_4中的所有节点，使得I_1和I_3被孤立。图6-11中的边可以看做代表不同节点组I_1、I_2、I_3和I_4之间的一束边。 □

137
~
138

6.6 弱Byzantine一致性

我们在6.4、6.5节中用到的证明Byzantine一致性在$n \leqslant 3f$或$conn \leqslant 2f$的情况下不可解的方法也可以用来证明其他一致性问题的不可能性结果。作为一个例子，在本节中，我们说明了怎样证明Byzantine一致性问题的一个弱变形的不可能性结果，这个变形称为*弱Byzantine一致性*。

弱Byzantine一致性问题和普通Byzantine一致性问题的惟一的区别在于有效性的条件。弱Byzantine一致性问题的有效性条件是 139

有效性条件 如果没有故障进程，而且所有进程都以相同初始值$v \in V$开始，那么v是惟一可能的决定值。

在普通Byzantine一致性问题中，如果所有无故障进程都以同一初始值v开始，那么即使故障进程存在，这些无故障进程都必然决定出v。在弱Byzantine一致性问题中，只有在没有发生无故障情况下它们才需要决定出v。

因为新问题声明比原来的要弱，所以我们为普通Byzantine一致性问题给出的算法同样也适用于弱Byzantine一致性问题。但是，不可能解的结果不能直接拿过来用，因为对于弱Byzantine一致性问题，可能存在更有效的算法。然而，事实证明（除了有一点技术性要求）对 140
进程的数目和图的连通性的限制还是成立的。（技术性要求是：现在我们需要假设$n \geqslant 3$，因为对于$n=2$的弱Byzantine一致性问题，存在一个平凡解法。）

定理6.30 假设$n \geqslant 3$。在一个允许f个故障进程的n节点网络图G中，弱Byzantine一致性问题是可解的，当且仅当：

1）$n>3f$

2）$conn(G)>2f$

证明 正如定理6.29所说，充分性条件可通过对于普通Byzantine一致性的协议的存在性得到证明。对于三个进程，其中一个发生故障，这里证明了弱Byzantine一致性问题是不可解的；对$f>1$的情况和连通性的证明则留做练习。为简单起见，我们假设$V=\{0,1\}$。

假设存在一个三进程算法A可以解决弱Byzantine一致性问题，其中三个进程分别为1、2、3，且其中一个是可能出故障的。令α_0为A的一个运行，其中三个进程都是以0开始的，而且没有出故障的进程。终止性和有效性条件意味着在α_0中三个进程最后都决定出0。令r_0为所有进程都做出决定所需的最小轮数。类似的，令α_1为A的一个运行，其中三个进程都是以1开始，而且没有出故障的进程。则所有进程最后在α_1中都决定出1。令r_1为所需的轮数，并选取$r \geqslant \max\{r_0, r_1, 1\}$。

通过把A的$2r$个副本放到一共有$6r$个进程的环中，其中$3r$个在“上半部分”，$3r$个在“下半部分”，我们建立一个新系统S。在上半部分的进程都以输入值0开始，在下半部分的进程都以输入值1开始。如图6-16所示。（这里，对A的相同进程的多个副本，我们不再使用带撇的符号或者其他用于区别的标记。）令α为S的一个运行。

通过像引理6.26的证明中一样的讨论，我们可以证明S中任意两个相邻进程在运行α中必须都决定出相同的值；因为在这两个进程看来，它们是在一个三角形中，且与第三个进程（出故

障的进程）交互。因此，S中所有进程必然在α中达成相同决定。（不失一般性）假设它们都决定出1。

现在来导出一个矛盾，我们证明在上半部分的某个进程必然决定出0。让B为S的上半部分任意$2r+1$个连续进程的“块”；在α中它们都是以初值0开始的。现在B中的所有进程在α中的开始状态与同名进程在α_0
141 中的相同，而且在第一轮发送相同的消息。因此，在两个运行中的第一轮，B中的所有进程（可能要除了两个端点中的一个）收到的消息与它们的同名者在α_0中收到的一样，因此也就与同名者处于相同的状态，并在第二轮发送相同的消息。在两个运行的第二轮，B中除了两个端点，所有进程接收相同的消息并处于相同的状态。如此继续下去，我们看到在第k轮，$1 \leqslant k \leqslant r$，$B$中除了端点处的$k$个进程，其他所有进程都在$\alpha$与$\alpha_0$中收到相同消息并处于相同状态。也就是说，在第$k$轮，对于$B$中除了端点处$k$个进程之外的所有进程，$\alpha$与$\alpha_0$是不可区分的。简单的说，这是因为信息来不及从$B$块外传播到那些进程。

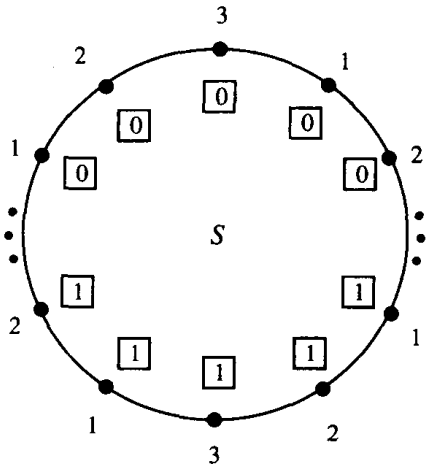


图6-16　合并A的$2r$个副本得到S

特别地，在第r轮，对于块B中间的进程i来说，α和α_0是不可区分的，但是因为在α_0中，进程i在r轮的最后决定出0，故它在α中也决定出0。这与进程i在α中决定出1相矛盾。 □

6.7　有停止故障时的轮数

在这章的最后，我们来证明在少于$f+1$轮的情况下一致性问题是不能解决的，这对Byzantine故障或停止故障都成立。也就是说，对于每种故障，不存在一个一致性协议使得所有无故障进程都在第f轮做出决定。

142 我们采用的方法是：假设存在一个f轮的一致性算法，然后导出矛盾。对于我们来说，给假设的算法加上限制是很容易的，不会引起一般性的丧失。首先，我们假设网络图是完全连接的，一个非完全图的快速算法在完全图中也可以用，所以这种限制就不会丧失一般性。我们还假设所有的进程在第f轮末做决定，然后马上就停止。在这种情况下，一个Byzantine一致性问题的算法就必然就是一个停止一致性问题的算法（参考6.1节中对两个问题关系的说明）。所以，为了得到一个不可能解，我们可以把注意力只放在停止一致性问题上。我们还假设每个进程都在每k轮，$1 \leqslant k \leqslant f$，向其他进程发送消息（除非它发生故障）。最后我们把注意力放在$V=\{0,1\}$情况上。

就像第5章中的协同攻击问题一样，我们使用通信模式的概念，指出在每一轮哪个进程向其他哪个进程发送消息。把以前的定义特殊化为完全图中的情况，我们定义通信模式为如下集合的任意子集

$$\{(i,j,k):1 \leqslant i,j \leqslant n, i \neq j, 1 \leqslant k\}$$

一个通信模式并不描述消息的内容，它只是描述在哪一轮从哪一个进程向另外哪一个进程发送消息。

我们在通信模式上考虑三个限制。首先，因为我们考虑的算法有f轮，所以在我们考虑的通信模式中，所有的三元组（i，j，k）都满足$k \leqslant f$。第二，因为我们考虑的是停止故障模型，所有的通信模式都满足以下的限制：如果任意三元组（i，j，k）在模式中丢失了，那么所有

(i, j', k')也丢失了，其中$k'>k$。也就是说，如果进程i在第k轮没有成功地发送消息，那么它就不会在以后的轮中发送消息。第三，因为我们考虑的运行最多有f个故障，所以所有的通信模式最多含有f个故障进程（我们定义通信模式中的一个进程i是“有故障的”，是指某个形如(i, j, k)的元组在模式中丢失了，其中$k \leqslant f$）。我们把满足以上的三个条件的通信模式（仅仅在本章的剩余部分中）称为良好的通信模式。

例 6.7.1 良好的通信模式

一个良好的通信模式（$n=f=4$）的例子如图6-17所示。在这个模式中，进程3在第一轮给进程4发送消息，但是给进程1和2发送消息时失败了。所以，进程3一定是在第一轮停止，并且在后来的轮中没有发送消息。同样，进程2在第二轮末尾停止。进程1和4是无故障的。 143

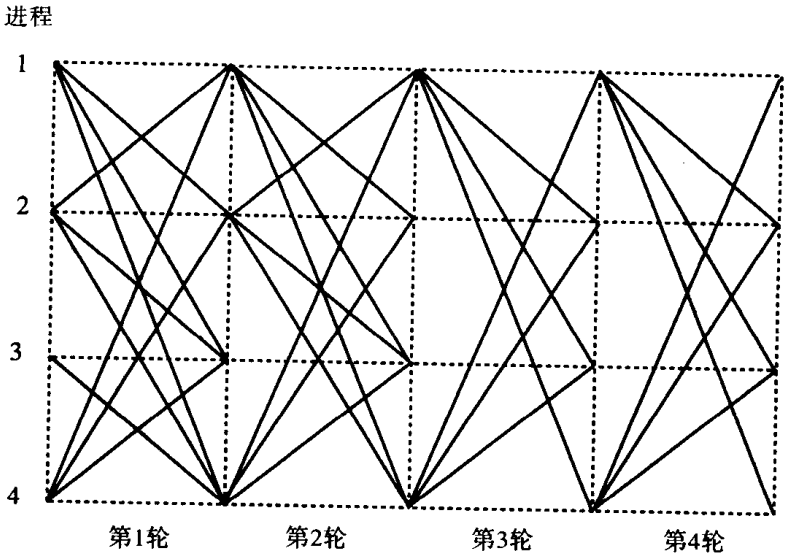


图6-17 一个良好的通信模式

现在我们定义运动为以下两条性质的结合：

1）一个对所有进程的输入值的赋值

2）一个良好的通信模式

（这与5.2.1节中的对手相似）

对于一个特定一致性算法A，每个运行ρ都定义了相应的运行$exec(\rho)$。通过根据ρ中给出的输入赋值来对进程的输入状态部分设置，从而定义了进程的初始状态。发送的消息由ρ的通信模式决定，这需要对发送进程的先前状态使用A的消息转移函数；初始状态之后的状态由A的状态转移函数决定。（但是，当任意进程发送消息失败后，就不再使用它的状态转移函数。）

为了给出下界的直观认识，开始我们先证明$f=1$的特殊情况。 144

定理6.31 假设$n \geqslant 3$。那么不存在能够容忍一个故障的n进程停止一致性算法，其中所有无故障进程都在第一轮结束前做决定。

证明 为了得到矛盾，假设存在一个这样的算法A。我们假设A满足本节开始列出的所有限制。

构造A的一个运行链，其中每个运行至多有一个故障进程，使得a）链的第一个运行包含0作为它的独一无二的决定值。b）链的最后一个运行包含1作为它的独一无二的决定值。c）对在两个运行中都不出故障的某个进程来说，链中任意两个连续运行都是不可区分的。那么因为

任意两个连续运行对于某个无故障进程i来说是一样的，所以进程i在两个运行中必须做出相同的决定。所以这两个运行必须有相同的惟一决定值。因此链中的每个运行必须有相同的惟一决定值，这就和性质a）、b）矛盾了。

链的开始为由运动ρ_0决定的运行$exec\ (\rho_0)$，其中所有进程的输入值都是0并且没有故障进程。运动如图6-18所示。根据有效性，$exec\ (\rho_0)$中惟一决定值必然是0。从运行$exec\ (\rho_0)$开始，我们通过删除一条消息构造下一运行，删除的消息是从进程1到进程2的。结果如图6-19所示。对于
145 除了进程1和2之外的所有进程来说，这个运行和$exec\ (\rho_0)$是不可区分的。因为$n \geq 3$，故至少存在一个这样的进程。这个进程在两个运行中都没有发生故障。

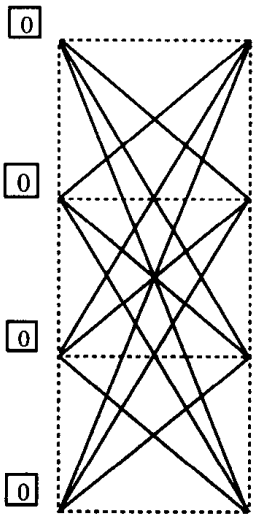


图6-18　运动ρ_0—所有输入为0，并且没有故障

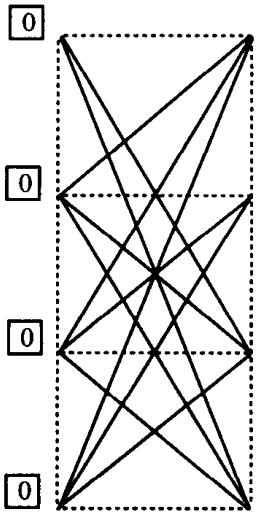


图6-19　从ρ_0中删除一条消息的结果

下面，我们删除从1到3的消息；对于除了进程1和3之外的所有进程来说，这个运行和先前运行是不可区分的，而且至少有一个这样的进程。我们继续这样的做法，每次删除一条从进程1发出的消息，这样使得对于某个无故障进程来说，每两个连续的运行都是不可区分的。

一旦我们移走了所有的从进程1发送的消息，我们把进程1的输入值从0改为1。当然，除了进程1，对于每个进程来说，最后得到的运行和先前运行是不可区分的，原因在于进程1没有在运行中发送消息。接着，我们来逐个替代进程1的消息，同样对于某个无故障进程来说，每一对连续运行都是不可区分的。采用这种方法，我们以$exec\ (\rho_1)$结束，其中ρ_1是这样的运动：其中进程1有输入值1，其他进程输入值为0，并且没有故障发生。

接着，我们对进程2重复这个构造过程，首先逐个移走所有的从进程2发出的消息，然后把进程2的输入值从0改为1，然后替代它的消息。得到的运行是ρ_2，其中进程1和2输入值为1，其他进程输入值为0，并且没有故障发生。对进程3到n做重复这个构造过程，最后得到$exec\ (\rho_n)$，
146 其中ρ_n是之中所有进程从1开始且没有故障发生的运动。

所以我们就构造了一条从$exec\ (\rho_0)$到$exec\ (\rho_n)$的链，它满足性质c）。但是有效性条件意味着在$exec\ (\rho_0)$中惟一的决定值是0，在$exec\ (\rho_n)$中惟一的决定值是1，这产生了a）和b）。所以我们就有了导出矛盾所需的链。 □

在说明一般情况之前，我们再做一步预备工作，说明$f = 2$时的情况。

定理6.32　假设$n \geq 4$。那么不存在能够容忍两个故障的n进程停止一致性算法，其中所有无故障进程都在第二轮结束前做决定。

证明 同样假设存在一个这样的算法。和前面证明中一样，我们用类似的构造方法构造一条链，使得它满足a）、b）、c）性质。对于每个i来说，$0 \leqslant i \leqslant n$，令$\rho_i$代表（两轮的）运动，其中进程1,..., i有输入值1，进程i+1,..., n有输入值0，并且没有故障发生。链以$exec(\rho_0)$开始，以$exec(\rho_n)$结束，之间是所有运行$exec(\rho_i)$。

从$exec(\rho_0)$开始，我们想要在开始时就杀掉进程1。当我们在处理一轮的时候，可以简单地逐条删除从进程1发送的消息。现在逐条删除进程1的第二轮消息是没有问题的。但是，如果在链的一步中，我们删除了进程1在第一轮从进程1发给其他某个进程i的消息，那么对于某个无故障进程来说，这两个连续运行就不再是不可区分的了。这是因为在第二轮中，进程i可以告诉其他的进程是否在第一轮中从进程1接收到了消息。

我们分几步来删除从进程1到进程i的第一轮的消息，从而解决这个问题。在链中间的运行中，进程1和进程i都是发生故障的；因为$f=2$，这是允许的。特别的，在我们开始的运行中，进程1在第一轮给进程i发送消息，而且进程i是无故障的。我们逐条删除进程i发送的第二轮的消息，直到得到一个运行，其中进程1在第一轮给进程i发送消息，而进程i在第二轮没有发送消息。接着，我们删除从进程1到进程i的第一轮的消息；对于除了进程1和i之外所有进程来说，得到的运行和先前运行是不可区分的。然后我们来替代进程i在第二轮中发送的消息，直到我们得到一个运行，其中进程1在第一轮不给进程i发送消息，而且进程i是无故障的。这就实现了我们要移走从进程1到进程i的第一轮消息的目的，同时保证了对于某个无故障进程来说，每对连续的运行是不可区分的。

按照这样的方法，我们逐条删除进程1发送的第一轮的消息，直到进程1不发送消息为止。然后和以前一样把进程1的输入值从0改为1。我们“按相反的方向”继续这个过程，逐条替代进程1的第一轮的消息。对进程2，...，n重复这样的过程，从而得到所需的链。 □ 147

现在我们来证明一般的定理。

定理6.33 假设$n \geqslant f+2$。那么不存在能够容忍f个故障的n进程停止一致性算法，其中所有无故障进程都在第r轮结束前做决定。

定理6.31和定理6.32的证明包含了证明定理6.33的主要思想。在这个通用证明中，我们使用f个进程故障来构造一个更长的链。相对于定理6.31和定理6.32，我们给出更加形式化的证明。因此我们需要一些记号。

首先，如果ρ和ρ'是两个运动，其中进程i是无故障的，那么我们用$\rho \overset{i}{\sim} \rho'$表示$exec(\rho) \overset{i}{\sim} exec(\rho')$，亦即由运动$\rho$和$\rho'$产生的运行对于进程$i$是不可区分的。如果对于某个在$\rho$和$\rho'$中都没有发生故障的进程$i$有$\rho \overset{i}{\sim} \rho'$，则记作$\rho \sim \rho'$。~关系的传递闭包记住$\rho \approx \rho'$。

其次，注意在定理6.31和6.32的证明中，链中产生的所有通信模式都有一个特别简单的形式。我们把这个形式定义如下。我们定义一个良好的通信模式是规则的，是指对于每一个k，$0 \leqslant k \leqslant f$，在$k$轮结束之前最多有$k$个进程出现故障（至少发送一条消息）。如果一个运动或一个运行的通信模式是规则的，我们说这个运动或运行是规则的。

最后，如果ρ是一个运动，并且$0 \leqslant k \leqslant f$，我们定义$ff(\rho, k)$为$\rho$的变形，它和$\rho$有相同的输入赋值，在时间$k$后没有故障发生，在前$k$轮中和$\rho$有相同的通信模式且之后不再发生故障。下面是关于$ff$的一些明显事实。

引理6.34 如果ρ是一个规则的运行，那么

1）对于任意的k，$0 \leqslant k \leqslant f$，$ff(\rho, k)$是规则的。

2）如果除了在ρ中出错的某个进程i也会在ρ'中的以后轮中出错之外，ρ'和ρ是相同的，那么ρ'是规则的。

3）如果没有进程在第$k+1$轮出错，那么$ff(\rho, k) = ff(\rho, k+1)$。

证明定理6.33的核心部分在于以下的更强引理，它说明有可能构造一个在任意两个具有相
148 同输入赋值的规则运动之间的链。

引理6.35　假设A是一个能够容忍f个故障的n进程停止一致性算法，其中所有无故障进程都在第f轮结束前做出决定。令ρ和ρ'为A的两个具有相同输入赋值的规则运动，那么$\rho \approx \rho'$。

证明　我们通过证明以下参数化声明来证明这个定理。$k=0$的情况就直接证明了这条引理。

声明6.36　令k为一个整数，$0 \leqslant k \leqslant f$。让$\rho$和$\rho'$是$A$的两个规则运动，它们具有相同的输入赋值，而且在前$k$轮中有相同的通信模式。那么有$\rho \approx \rho'$。

证明　声明6.36的证明是通过对k做反向归纳来完成的，从$k=f$开始，以$k=0$结束。

基础：$k=f$。因为我们假设在f轮之前ρ和ρ'具有相同的输入和相同的通信模式，这就意味着ρ和ρ'是相同的。显然此时声明成立。

归纳步：假设声明对$k+1$成立，现在来证明$0 \leqslant k \leqslant f-1$的情况。在这种情况下，只要证明任意规则运行$\rho$都满足$\rho \approx ff(\rho, k)$就足够了，因为我们可以把这个结果应用两次来得到所需声明。所以固定某个规则运行ρ，由引理6.34得到$ff(\rho, k)$是规则的。

由归纳假设得，$ff(\rho, k+1) \approx \rho$，所以只需要证明$ff(\rho, k) \approx ff(\rho, k+1)$就足够了。如果没有进程在$\rho$中的第$k+1$轮发生故障，那么引理6.34就意味着$ff(\rho, k) \approx ff(\rho, k+1)$。所以我们假设至少有一个进程在$\rho$中的第$k+1$轮中发生故障。令$I$为这些进程的集合。

令ρ_0为除了I中所有进程在$k+1$轮后发生故障之外与$ff(\rho, k)$相同的运动。那么引理6.34的第二部分（用于ρ）意味着ρ_0是规则的。

因为ρ_0和$ff(\rho, k)$是其中前$k+1$轮都相同的规则运行，那么我们可以用归纳假设来证明$\rho_0 \approx ff(\rho, k)$。因此，要证明$ff(\rho, k) \approx ff(\rho, k+1)$，只需证明$\rho_0 \approx ff(\rho, k+1)$。

现在，我们构造一个从ρ_0到$ff(\rho, k+1)$的规则运动的链。ρ_0和$ff(\rho, k+1)$的惟一区别是：I中进程在ρ_0中的第$k+1$轮发送的消息在$ff(\rho, k+1)$中会被丢失。所以我们把那些消息一次性删除，而保持运动不变。

例如，考虑删除从i到j的一条消息，其中$i \in I$。令σ为包含这条消息的运动，令τ为不包含这条消息的运动；我们必须证明$\sigma \approx \tau$。如果$k+1=f$，那么对于除了i和j之外的所有进程来说，σ和τ是不可区分的；因为$n \geqslant f+2$，并且i是发生故障，所以这必然包括至少一个无故障进程。故有$\sigma \approx \tau$，得证。

149 另一方面，如果$k+1 \leqslant f-1$，那么定义σ'与τ'分别和σ与τ相同——除了在σ'与τ'中j在第$k+2$轮的开始发生故障之外（如果它以前没有发生故障的话）。如图6-20所示。

σ'与τ'都是规则的，原因在于σ与τ都最多包含$k+1 \leqslant f-1$个故障，而且在第$k+2$轮中我们只引入一个新故障。那么根据归纳假设，$\sigma \approx \sigma'$，$\tau \approx \tau'$。且有$\sigma' \approx \tau'$，原因在于它们对于除了i和j之外的所有进程是不可区分的。故此，我们又得到$\sigma \approx \tau$。

这说明了可以构造从ρ_0到$ff(\rho, k+1)$的链，因此$\rho_0 \approx ff(\rho, k+1)$，所以$\rho \approx ff(\rho, k)$，得证。　□

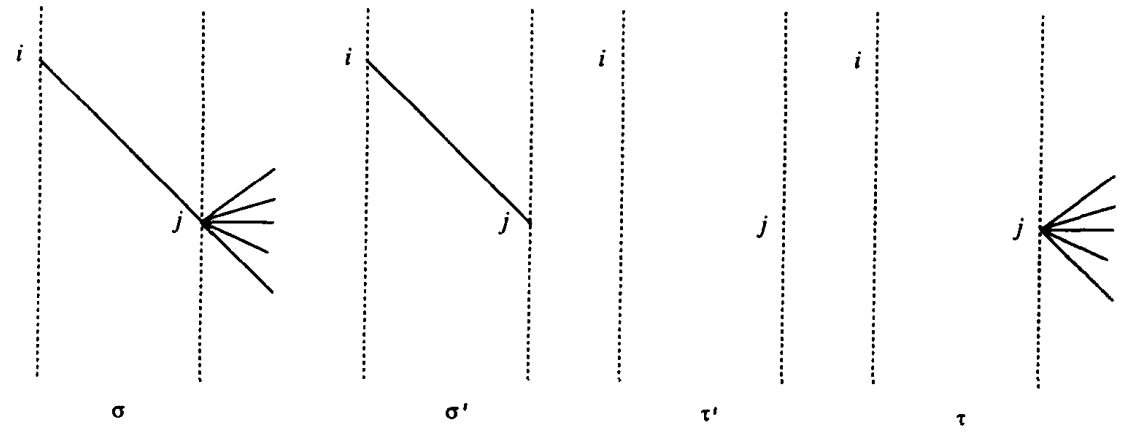


图6-20 在声明6.36的证明中删除从i到j的k+1轮消息

正如以前提到的，由声明6.36可以直接得到引理6.35。

现在我们把引理6.35扩展到不同输入赋值的情况。

引理6.37 假设A是一个能够容忍f个故障的n进程停止一致性算法，其中所有无故障进程都在第f轮结束前做出决定。令ρ和ρ'为A的两个规则运动，那么$\rho \approx \rho'$。

证明 根据引理6.35，每个ρ都和它的无故障版本相关联，也就是$\rho \approx ff(\rho, 0)$。所以在不失一般性的情况下，我们可以假设引理中叙述的ρ和ρ'都是无故障的。

如果ρ和ρ'是有相同的输入赋值，那么它们是相同的，这不需要什么证明。

假设ρ和ρ'只在一个进程i的输入上是不同的；比如，i在ρ中的输入是0，而在ρ'中的输入是1。那么定义σ和σ'为与ρ和ρ'分别相同的运动，只是i在一开始就发生故障。那么由引理6.35可 150
知，$\rho \approx \sigma$，$\rho' \approx \sigma'$。同样有$\sigma \approx \sigma'$，因为对于除了进程i之外的所有进程来说，σ和σ'是不可区分的。所以$\rho \approx \rho'$。

最后，假设ρ和ρ'在多于一个进程上的输入中是不同的。那么我们就可以构造一个从ρ到ρ'的无故障运动链，链中的每一步都只改变一个进程的输入值。把前面的情况用于链中的每一步。我们又得到$\rho \approx \rho'$。 □

使用引理6.37，很容易证明定理6.33。我们已经知道，所有的规则运行都被链相联系；现在我们考虑在这些运动中的决定值。假设$n>f$，终止性和一致性性质意味着对于每个ρ，$exec(\rho)$中都产生一个惟一决定值$dec(\rho)$。以下的引理说明，满足~或≈关系的运动必然产生相同的决定值。

引理6.38

1）如果$\rho \sim \rho'$，那么$dec(\rho) = dec(\rho')$

2）如果$\rho \approx \rho'$，那么$dec(\rho) = dec(\rho')$

证明 对于1），回忆$\rho \sim \rho'$意味着在ρ和ρ'中都存在一个无故障进程i，使得 $exec(\rho) \overset{i}{\sim} exec(\rho')$。这说明进程$i$在$exec(\rho)$和$exec(\rho')$中决定出相同的值。因此$dec(\rho) = dec(\rho')$。

2）可以从1）得到。 □

（定理6.33的）证明 假设存在这样一个算法A；我们假设A满足本节开始所列出的限制。

令ρ_0为A的一个运动，其中所有进程都是以0开始，而且没有发生故障，令ρ_1为A的另一个

运动，其中所有进程都是以1开始的，而且没有发生故障。由引理6.37可知，$\rho_0 \approx \rho_1$。由引理6.38 2)可知，$dec(\rho) = dec(\rho')$。但是有效性条件意味着$dec(\rho_0) = 0$，$dec(\rho_1) = 1$，矛盾。 □

弱有效性条件 注意，如果我们把有效性条件降到6.6节中用于弱Byzantine一致性问题的那种程度，这里不可能性的证明还是可以用的。也就是说，假设$n \geq f+2$，我们已经证明了解决弱Byzantine一致性问题也至少需要$f+1$轮。

151

6.8 参考文献注释

本章中的许多思想都是源于两篇奠基性的论文，一篇来自Pease、Shostak和Lamport[237]，另一篇来自Lamport、Shostak和Pease [187]。这两篇论文包含了Byzantine一致性要求的进程数目的上下限$3f+1$，以及一个带鉴别的一致性算法，所有这些都是针对全连通图的。第二篇论文中的描述使用了攻击将军而不是进程的形式。正是第二篇论文为这种故障模型创造了术语*Byzantine*。

更详细地说，这两篇论文定义了Byzantine一致性问题，并且使它成为软件实现的容错(Software-Implemented Fault Tolerance，SIFT)飞行器控制系统中出现的问题的一种抽象[289]。[237]中算法使用的指数数据结构与EIG树相似；Byzantine一致性算法与*EIGByz*算法相似，而带鉴别的算法与*EIGStop*相似。[187]中的算法几乎与它们完全相同，但采用了递归的形式。[237]中的$n \leq 3f$个进程的不可能性证明涉及详细情况的明确构造。[187]中的不可能性证明介绍了缩减到三对一情况的方法，这在定理6.27的证明中出现了。

对于可以使用鉴别的Byzantine一致性问题，Dolev和Strong[93]给出了与*FloodSet*和*OptFloodSet*算法相似的算法。Dolev[94]考虑了在不一定是全连通的图中的Byzantine一致性问题。他用清晰的构造方法，证明了定理6.29中给出的连通性界限。Dolev、Reischuk和Strong[99]针对特定的通信模式，给出了“提前停止”的算法。另外的提前停止算法由Dwork和Moses[105]，以及Halpern、Moses和Waarts[145]开发。

Bar-Noy、Dolev、Dwork以及Strong定义了*EIG*树的数据结构，并给出了*EIGByz*算法，本质上其形式与本书一样[39]。*TurpinCoan*算法来自[279]。

对Byzantine一致性问题的第一个多项式通信复杂度的算法是Dolev和Strong[101]提出的；接着被Dolev、Fischer、Fowler、Lynch和Strong[96]提高以得到$2f+3$的时间界限。Coan[82]开发了一个折衷算法，其中对任意$\varepsilon>0$，轮数降低为$(1+\varepsilon)f$；通信复杂度是多项式级，但多项式的级数取决于ε。一致广播原语和*ConsistentBroadcast*算法是Srikanth和Toueg[269]提出的。*PolyByz*算法的基础是Srikanth和Toueg[269]以及Dolev等人[96]提出的算法。接下来，Moses和Waarts[231]，Berman和Garay[49]，以及Garay和Moses[133]的研究，产生了$f+1$轮的多项式级通

152

信的Byzantine一致性算法；其中Garay和Moses[133]的研究得到了进程数的下限$n=3f+1$。然而这些算法都很复杂。

正如已经提到的，Byzantine一致性要求的进程数下限$n>3f$最初是在[237，187]中证明的，而连通性的下限最初是在[94]中证明的。然而本书中给出的证明是由Fischer、Lynch和Merritt[122]提出的。Menger定理最初是由Menger[225]证明的，在Harary的书[147]中可以看到。

弱Byzantine一致性问题是Lamport[178]定义的。他还给出了进程数下限的结论。但本书中给出的证明是由Fischer、Lynch和Merritt[122]提出的。

对Byzantine故障，达成一致所需轮数的第一个下限结论是由Fischer和Lynch[119]证明的。以后又被Dolev和Strong[93]，以及DeMillo，Lynch和Merritt[88]扩展到带鉴别的Byzantine故障中。Merritt[226]第一个把Dolev和Strong[101]的思想用于停止故障。Dwork和Moses[105]给出

了对这个结果的另一种证明；他们的证明更好地分析了不同运动所需的时间。Feldman和Micali[113]使用“秘密共享”技术来获得一个常数时间随机化解法。

Fischer[117]的一篇论文调查了关于一致性问题的大量早期工作。

Draper实验室的大量工作[172，173]涉及利用Byzantine一致性来设计容错多处理器和处理器故障诊断算法。这些设计已经被用于一些安全性要求很高的应用程序，如无人海底运输器、核攻击潜艇以及核电站控制。

6.9 习题

6.1 如果停止故障的有效性条件改为只要求无故障进程达成一致，证明任意解决Byzantine一致性问题的算法可以解决停止一致性问题。

6.2 证明对于任意解决了Byzantine一致性问题的算法，其中所有无故障进程总是在同一轮中做出决定，那么它也可以解决停止一致性问题。 153

6.3 证明引理6.2。

6.4 追踪存在四个进程和两个故障情况下的*FloodSet*算法的运行，其中进程的初始值分别为1，0，0，0。假设故障进程为1和2，进程1在第一轮中只向进程2发送消息后失败，进程2在第二轮中只向进程1和3而不是进程4发送消息后失败。

6.5 考虑f个故障的*FloodSet*算法。假设算法仅仅运行f轮，而不是$f+1$轮，决定规则相同。描述违反正确性要求的一个特定运行。

6.6 a）除了课本中讨论的决定规则，描述另一个决定规则，使它可以在*FloodSet*算法中正确工作。

b）对可以正确工作的决定规则的集合，给出其准确特征。

6.7 把*FloodSet*算法扩展到任意（不必是完全）连通的图中，对它进行分析，并证明其正确性。

6.8 给出*OptFloodSet*算法的代码，通过证明引理6.5、6.6和6.7来完成课本中给出的证明。

6.9 在一个值域V中，考虑下面的简单算法，它是针对存在停止故障的一致性的算法。每一个进程保持一个变量*min-val*，初始值为它自己的初始值。在$f+1$轮中的每一轮，所有的进程广播它们的*min-val*，然后在它原来的*min-val*和收到的消息中的所有的*min-val*中选择一个最小的，设置为它的*min-val*的新的值。最后的决定值是*min-val*。给出算法的代码，并证明（直接证明或通过模拟证明）其正确性。

6.10 追踪针对四个进程和两个故障情况的*EIGStop*算法的运行，其中进程的初始值分别为1，0，0，0。假设故障进程为1和2，进程1在第一轮中只向2发送消息后失败，进程2在第二轮中只向进程1和3而不是进程4发送消息后失败。

6.11 证明引理6.11。

6.12 证明引理6.12的1）。 154

6.13 考虑f个故障的*EIGStop*算法。假设算法仅仅运行f轮，而不是$f+1$轮，决定规则相同。描述违反正确性要求的一个特定运行。

6.14 证明*FloodSet*算法正确性的另一种方法是通过模拟关系把它和*EIGStop*相联系。为此，我们首先扩展*EIGStop*算法，允许每个进程i在所有轮中广播所有的值，而不仅仅是那些与标号不包含i的节点相联系的值。必须说明这种扩展不会影响算法的正确性。另外，在*EIGStop*算法的描述中必须加入一些细节，比如*rounds*和*decision*变量必须显式地操作。那么*FloodSet*和修改后的*EIGStop*就可以并排运行，它们从相同的初始值集合开始，而且故障在相同进程和相同时间中发生。

用这种方法证明*FloodSet*算法的正确性。证明的核心是下面的模拟关系，它是关于两个算法中相同轮数后的状态的。

断言 6.9.1　对任意r，$0 \leqslant r \leqslant f+1$，在$r$轮之后，下列结论成立。

a）在两个算法的状态中，变量*rounds*和*decision*的值是相同的。

b）对每一个i，在*EIGStop*中标注i的树的节点的值*vals*的集合与*FloodSet*中的集合W_i相等。

对于建立模拟关系所需的*EIGStop*的任意附加不变式，确保包括了它们的描述和证明。

6.15 使用下列两种方法之一，证明*OptEIGStop*的正确性。

a）通过*EIGStop*模拟方法，使用的证明方法与联系*OptFloodSet*和*FloodSet*的模拟证明方法类似。

b）通过把它和*OptFloodSet*联系。

6.16 对于带鉴别的Byzantine故障模型，证明*EIGStop*算法和*OptEIGStop*算法的正确性。可以用于证明*EIGStop*的一些重要事实在下面的断言中，与引理6.12的叙述类似。

155

断言6.9.2　在$f+1$轮之后：

a）如果i和j是非故障进程，$val(y)_i = v \in V$，而且xj是y的前缀，那么$val(x)_j = v$。

b）如果v在任意无故障进程的*vals*集合中，那么v是某个进程的初始值。

c）如果i是一个无故障进程，$v \in V$在i的*vals*集合中，那么必然存在某个不包含i的标号y，使得$v = val(y)_i$。

这些事实可以从数字签名的属性得到。

6.17 研究问题：形式化地定义带鉴别的Byzantine故障模型，并对它的能力和局限的有关结论进行证明。

6.18 给出*EIGStop*的一个运行例子，以说明对Byzantine故障*EIGStop*不能解决一致性问题。

6.19 考虑7个进程的3轮的*EIGByz*算法。任选两个进程作为故障进程，并假设所有进程的输入和故障进程的消息值都是随机选择的。计算运行中产生的所有信息，并证明它满足正确性条件。

6.20 证明在*EIGByz*算法中，并不需要*EIG*树中的每一个节点都是公共的。

6.21 考虑*EIGByz*算法。构造一个明确的运行，说明在下列情况下运行时，算法会产生错误的结果。

a）7个节点，两个故障，2轮。

b）6个节点，两个故障，3轮。

6.22 *TurpinCoan*算法在第一轮和第二轮使用了阈值$n-f$，有哪些能使算法正确工作的其他阈值对？

6.23 假设我们考虑*TurpinCoan*算法有两个故障进程集合F和G，而不是只有一个。每一个集合最多有f个进程。F中的进程除了能在第1、2轮发送错误消息外，没有其他的故障。G中的进程只允许在运行二元Byzantine一致子例程时有错误行为。在这些故障假设下，组合算法怎样保证正确性条件呢？给出证明。

156

6.24 现在我们假设$n>4f$。使用二元Byzantine一致子例程，解决一个算法多值的Byzantine一致性问题。算法通过要求一个额外轮，而不是两个额外轮，来改善*TurpinCoan*算法。

6.25 证明*ConsistentBroadcast*算法中，在一个无故障进程接收到(m, i, r)消息之前，不存在时间上限。也就是说，对任意t，有一个*ConsistentBroadcast*的运行，其中某个无故障进程在$r' \geqslant r+t$轮收到消息。

6.26 你能够给出一个在Byzantine故障模型中实现一致广播机制的满足以下要求的算法吗？要求故障$f>>1$，而且在$r+1$轮之后，没有无故障进程严格地接收(m, i, r)消息。

要么给出这样的算法并证明其正确性，要么说明为什么不存在这样的算法。

6.27 描述一个*PolyByz*的最坏情况下的运行，也就是说在这个运行中，存在某个无故障进程i，使得进程i从$2f+1$个不同的进程接收消息的最早轮数恰是$2(f+1)$。

6.28 Flaky计算机公司的一个程序员修改了*PolyByz*算法的实现，使得形如$2s-1$的每一轮的接收阈值为$s-1$，而不是$f+s-1$，而且决定阈值从$2f+1$变为$f+1$。这种修改正确吗？给出证明或者反例。

6.29 不使用二元Byzantine一致的子例程，为通用输入值集合的Byzantine一致性问题设计一个多项式级通信复杂度的算法。你的算法会用到一致广播机制，但你必须设计一个比*ConsistentBroadcast*算法更好的实现。

6.30 为停止一致故障设计一个算法，使它满足以下的提前停止（*early stopping*）属性：如果在算法的运行中，仅仅$f' < f$个进程发生故障，那么在所有的无故障进程做出决定之前，所需的时间最多是kf'，其中k是某个常数。对Byzantine一致做同样的工作。

6.31 为全连通图中的四个进程设计一个协议，要求能够容忍一个Byzantine故障，或者三个停止故障。尽量减小轮数。

157

6.32 研究问题：设计一个解决Byzantine一致的简单的$f+1$轮的协议，要求只有$3f+1$个进程，通信复杂度是多项式级。

6.33 这个练习是为了弄清引理6.26中的构造方法，它把两个三角系统粘贴在一起构成一个六角形系统。

a）对三个进程的完全图，详细描述一个解决无故障一致问题的算法A。即Byzantine一致性问题在没有进程故障时的特殊情况。

b）现在像引里6.26证明中那样，通过把算法A的两个副本合起来构造系统S。详细描述S的一个运行，其中进程1、2、3从输入值0开始，1'、2'、3'从1开始。

c）S（在六角形网络中）能否解决无故障一致性问题？或者给出它能解决的证明，或者给出说明它不能解决的一个运行。

d）是否存在一个3个进程的算法A，使得A的任意多的副本可以合成一个环，而且这个环可以解决无故障一致性问题？

6.34 在如下网络图中运行的Byzantine一致性算法，可以容许的最大故障进程数是多少？

a）大小为n的环。

b）三维立方体，一个面上m个节点，其中节点仅仅在三个维度上与它们的邻接节点相连。

c）完全二分图，其中每一部分有m个节点。

6.35 对$n=2$，$f=1$时的Byzantine一致，给出更详细的不可能性证明。

6.36 对定理6.29中描述的一般图的Byzantine一致性算法，分析其时间、消息数，以及通信位数。你能对它们进行改善吗？

6.37 详细说明在定理6.29证明中的假设的简化，证明当$f=1$时不可能解决Byzantine一致性问题，而且$conn(G) \leq 2$实际上被调整了。也就是说，在进程1和3被任意连通子图取代的情况下，如果存在一个算法，就意味着在单节点情况下也存在一个算法。

158

6.38 重新考虑不能在图6-11的网络中达到Byzantine一致的证明。为什么这个证明不能扩展到如图6-21所示的网络中呢？

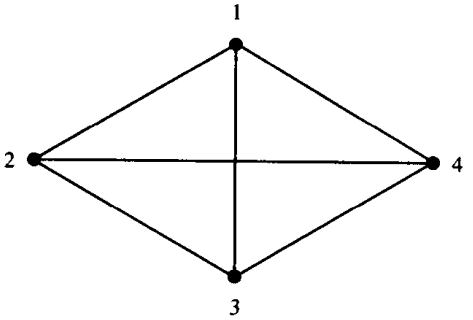


图6-21 习题6.38的网络图

6.39 证明有f个故障的Byzantine一致性，其中$f>1$，不能在$conn\ (G) \leq 2f$的图中解决。证明可以使用在定理6.29的证明的最后给出的进程分组的方法，也可以使用在定理6.27中的证明相似的缩减方法。

6.40 在由两个节点和一条连接节点的边构成的网络中，给出一个弱Byzantine一致的简单算法。

6.41 完成定理6.30的不可能性证明

a）当$n \leq 3f$，$f>1$时。

b）当$conn\ (G) \leq 2f$时。

6.42 考虑*Byzantine*射击队问题，定义如下。在一个全连通网络中有n个进程，没有输入值，开始时间可变。即每一个进程从一个安静（*quiescent*）状态开始，其中安静状态不包含信息，只发送*null*消息。除非从外面收到一个特殊的*wakeup*消息，或者从其他进程收到一个非空消息，状态才会改变。当进程醒来时，它不知道当前的轮数。模型与第2.1节的很相似，但是这里我们并不假设所有的进程都收到*wakeup*消息，只要求进程的任意子集如此。而且，我们允许Byzantine故障。

159 问题是对发送开火信号的进程，要满足下列条件：

一致性 如果在某一轮任意无故障进程都发送了*fire*信号，那么所有的无故障进程在同一轮发送*fire*信号，而在任意其他轮，没有无故障进程发送*fire*信号。

有效性 如果所有无故障进程都收到*wakeup*消息，那么所有无故障进程最终开火；如果没有无故障进程收到*wakeup*消息，那么没有无故障进程会开火。

a）设计一个算法解决$n>3f$的Byzantine 射击队问题。

b）证明如果$n \leq 3f$，问题不能解决。

6.43 对$f=3$的特殊情况，叙述定理6.33，并给出直接证明。

6.44 如果不要求运动是规则的，引理6.37还成立吗？给出证明或反例。

6.45 在6.7节中，我们已经证明了允许f个故障的停止一致性不能在f轮中解决。这包括构造连接两个运动的链，其中所有进程都是正确的，并且有相同的输入。然而链只能隐含地构造。

a）运动链有多长？

b）如果针对Byzantine故障而不是停止故障，你能把链缩短多少？

6.46 研究问题：在一般（不必是完全）网络图中，推导解决停止一致性问题和/或Byzantine一

160 致性问题所需时间的下限。

第7章 更多的一致性问题

在前面两章中我们都在研究一致性问题——第5章关于协同攻击问题，第6章关于一致性问题。在这一章中，我们对同步分布式一致性问题的学习，考虑了另外三种一致性问题：*k*一致性问题、近似一致性问题和分布式数据库提交问题。就像在第6章中一样，我们只考虑进程故障。

7.1 *k*—致性问题

首先我们来考虑*k*—致性问题，其中*k*是非负的整数。*k*—致性问题是在第6章中考虑的一致性问题的自然扩充。但是现在，我们不需要所有进程都只决定出相同的值，而是要求它们的决定限于*k*个不同的值，其中*k*是一个很小的数值。

这个问题源于一个纯粹的数学问题——它试图研究当问题的要求以这种简单方式变化时，第6章中的结果到底会怎么变化。但是我们可以想像出这个算法的实际用途。比如，考虑分配共享资源（如通信网络中的广播频率）的问题。对于一些进程来说，用少数广播频率来广播大量的数据（如录像带）是有必要的。因为通信是通过广播进行的，任意数量的进程都可以接收到采用相同频率的数据。为了使总的通信负载最小，我们就倾向于保留一个小的频率数*k*。

在本节中，我们在一个完全图网络中，在只有停止故障的情况下，证明解决*k*—致性问题中所需要的轮数的上界和下界。这些界限是根据进程的数目*n*、可以容忍的故障数*f*和允许的决定值的个数*k*来给出的。

7.1.1 问题

就像在普通一致性问题中一样，在*k*—致性问题中，我们假设网络为*n*个节点组成的无向图，进程为1,...,*n*，并且每个进程知道整张图。每个进程都以固定集合*V*中的一个输入值开始，并且最终输出集合*V*中的一个值。（同样的，我们假设对于每一个进程，每个开始状态包含每个输入值。）我们假设最多可能有*f*个进程发生故障。我们只考虑停止故障。所需条件如下：

一致性 存在*V*的一个子集*W*，|*W*|=*k*，所有的决定值都在*W*中。

有效性 任意进程的任意决定值都是某个进程的初始值。

终止性 最后所有无故障进程都做出决定。

这里的一致性条件是普通一致性问题的一致性条件的自然扩展。注意，在6.1节的最后，我们对停止故障使用了更强的有效性条件，而不是我们在第6章的大部分中所用的较弱的有效性条件；在证明7.1.3节中的下界时，我们需要这个更强的条件。具有更强一致性条件的普通一致性问题，事实上就是*k*—致性问题在*k* = 1时的特殊情况。

对于我们在本节中给出的结果，我们考虑完全图网络的特殊情况。我们还假设*V*是全序的。

像在6.2.1节中一样，如果在*r*轮后一个进程在*r*轮结束之前不出故障，0≤*r*，我们就定义它在*r*轮后是*活动的*。

7.1.2 算法

我们提出一个非常简单的算法，叫做*FloodMin*算法，事实上它就是我们在习题6.9中描述的算法，但是它运行轮数更少。正如我们在习题6.9中说明的，当算法执行了*f* + 1轮后，它会保证通常的停止一致性。结果表明，在只执行了 $\left\lfloor \frac{f}{k} \right\rfloor + 1$ 轮后，它能保证*k*—致性。因此，粗略地

说，允许k个而不是1个决定值，使得运行时间减小为原来的k分之一。

162 **FloodMin算法（非形式化）：**

每个进程都有一个变量*min-val*，开始设置成它自己的初始值。对每个$\left\lfloor \frac{f}{k} \right\rfloor + 1$轮，所有进程都广播它们的*min-val*值，然后每个进程都在它自己原来的*min-val*值和收到的消息的所有值中选择最小者，设置为它的新*min-val*值。最后，决定值为*min-val*。

代码如下：（把它的结构与6.2.1节的*FloodSet*相比较。）

FloodMin算法（形式化）：

消息字母是V。

***states*$_i$:**

rounds $\in \mathbb{N}$, initially 0
decision $\in V \cup \{unknown\}$, initially *unknown*
min-val $\in V$, initially i's initial value

***msgs*$_i$:**

if *rounds* $\leq \lfloor \frac{f}{k} \rfloor$ then send *min-val* to all other processes

***trans*$_i$:**

rounds := *rounds* + 1
let m_j be the message from j, for each j from which a message arrives
min-val := min($\{$*min-val*$\} \cup \{m_j : j \neq i\}$)
if *rounds* $= \lfloor \frac{f}{k} \rfloor + 1$ then *decision* := *min-val*

我们讨论正确性；证明就和在6.2.1中的*FloodSet*算法的证明相类似。令$M(r)$为在r轮后活动进程的*min-val*值集合。我们首先观察到*min-val*值只能随着时间递减。

引理7.1　对所有的r，$1 \leqslant r \leqslant \left\lfloor \frac{f}{k} \right\rfloor + 1$，有$M(r) \subseteq M(r-1)$。

证明　假设$m \in M(r)$。那么在r轮以后，对于某个在r轮以后活动的进程i来说，m就是在r轮以后*min-val*$_i$的值。那么，要么在r轮之前m = *min-val*$_i$，要么m在r轮的某个消息中从j到达i。但是在这种情况下，在$r-1$轮后，*min-val*$_j = m$，而且j在$r-1$轮后必然是活动的，因为它在r轮发送了消息。因此有$m \in M(r-1)$。□

引理7.2　令$d \in \mathbb{N}^+$。如果在一个特定轮r，$1 \leqslant r \leqslant \left\lfloor \frac{f}{k} \right\rfloor + 1$中，最多有$d-1$个进程发生故障，那么

163 $|M(r)| \leqslant d$，也就是说，在r轮后的活动进程中，最多有d个不同的*min-val*。

证明　为了构造矛盾，假设在r轮中最多只有$d-1$个进程发生故障，但$|M(r)| > d$。令m为$M(r)$中的最大元素，$m' \neq m$的是$M(r)$中其他任意元素。由引理7.1可知，m'是$M(r-1)$中的元素。令i为在$r-1$轮后任意活动进程，并且m'=*min-val*$_i$。如果i没有在r轮中发生故障，那么在r轮中，每个进程都会收到来自i的包含m'的消息。但是这是不可能发生的，因为某个进程在r轮以后把$m > m'$作为它的*min-val*。于是i在r轮中就发生故障。

但是m'是在$M(r)$中的除了最大值m之外的任意元素。所以对于$M(r)$每个元素$m' \neq m$来说，在$r-1$轮之后就有某个活动进程的*min-val*值和m'一样大，而且这个进程在r轮中发生故障。根据假设，在r轮中最多只有$d-1$个进程发生故障，所以在$M(r)$中除了m之外最多可能有$d-1$个元素。所以$|M(r)| \leqslant d$，这是矛盾的。□

现在我们可以来证明主要的正确性定理。

定理7.3 *FloodMin*算法解决了停止故障模型的k一致性问题。

证明 其中终止性和有效性是显而易见的。我们来证明新的一致性条件。为了得到矛盾，我们假设在一个最多有f个故障的特定运行中，不同的决定值的数目大于k。那么在$\left\lfloor \frac{f}{k} \right\rfloor + 1$轮之后，活动进程的*min-val*的数目至少是$k+1$，也就是$|M(\left\lfloor \frac{f}{k} \right\rfloor + 1)| \geq k+1$。由引理7.1，对于所有的$r$，$0 \leq r \leq \left\lfloor \frac{f}{k} \right\rfloor + 1$，有$|M(r)| \geq k+1$。那么引理7.2意味着在每一个$r$，$1 \leq r \leq \left\lfloor \frac{f}{k} \right\rfloor + 1$轮中，最少有$k$个进程发生故障。这导致了总故障数最少是$(\left\lfloor \frac{f}{k} \right\rfloor + 1)k$。但这是严格大于$f$的，因此产生了矛盾。

复杂度分析 轮数是$\left\lfloor \frac{f}{k} \right\rfloor + 1$。消息的个数最多是$(\left\lfloor \frac{f}{k} \right\rfloor + 1)n^2$，消息的位数最多为$(\left\lfloor \frac{f}{k} \right\rfloor + 1)n^2 b$，其中$b$是表示$V$中的一个元素所需位数的上界。

7.1.3 下界*

在本节中，我们来说明上界$\left\lfloor \frac{f}{k} \right\rfloor + 1$是严格的，我们将证明在$|V| \geq k+1$时，它也是一个下界。这给出了通过允许$k$个而不是1个输出值而得到的加速比的特征——本质上讲，时间除以k。你 164
可能会预料到，证明思想源于定理6.33中对普通一致性的下界的证明，但它们更先进，更有趣。事实上，它们会使我们进入代数拓扑学领域。

在本节的剩余部分，我们固定A为一个解决k一致性问题的n进程算法，它最多可以容许f个进程的停止故障。假设A在$r < \left\lfloor \frac{f}{k} \right\rfloor + 1$轮中停止，也就是说$r \leq \left\lfloor \frac{f}{k} \right\rfloor$。为了能够得到矛盾，我们需要进一步假设$n \geq f+k+1$，这就意味着至少$k+1$个进程从不发生故障。

不失一般性，我们假设所有进程都在r轮的最后做出决定，并且马上停止。我们还假设在每个k轮中，$1 \leq k \leq r$，每个进程都给其他进程发送消息（除非它发生故障）。最后，我们假设值域V恰包含有$k+1$个元素，0，1，...，k，这就是我们要得出矛盾的地方。

在A的一个运行中（最多有f个故障），$k+1$个进程选择了$k+1$个不同的值，这违反了k一致性，从而得到了矛盾。

概括 回想定理6.33的证明，其中我们证明普通停止一致性问题的轮数的下界为$f+1$。当时使用了链证明（*chain argument*），产生了一个运行链，范围从一个运行（其中：0是惟一的允许的决定值）到另一个运行（其中：1是惟一的允许的决定值）。我们来把这个证明扩展到其他的k的值。不幸的是，在k一致性问题中，不像其他的通常一致性问题，进程在一个运行中的决定值并不可以决定其他紧密相关的运行中的决定值。例如，在普通一致性算法中，如果两个运行α和α'对非故障进程i来说是不可区分的，那么不但i的决定值，而且其他所有无故障进程的决定值，在两个运行中都是一样的。但是在k一致性算法中，如果两个运行α和α'对进程i来说是不可区分的，那么i的决定值在两个运行中保证相等，其他进程就不确定了。即使α和α'对$n-1$个进程来说是不可区分的，那么剩下的进程还是不能确定。

我们使用的关键思想就是来构造一个k维的运行集合，而不是一个（一维）的链，其中相邻运行对于指定的无故障进程来说是不可区分的。我们把组织这些运行的k维结构称为*Bermuda*三角形（因为任意假设的k一致性算法都会消失在它内部某个地方）。 165

例 7.1.1　Bermuda三角形

图7-1是Bermuda三角形的一个例子，其中$k = 2$。它包含一个大三角形，被分割成很多小三角的集合。

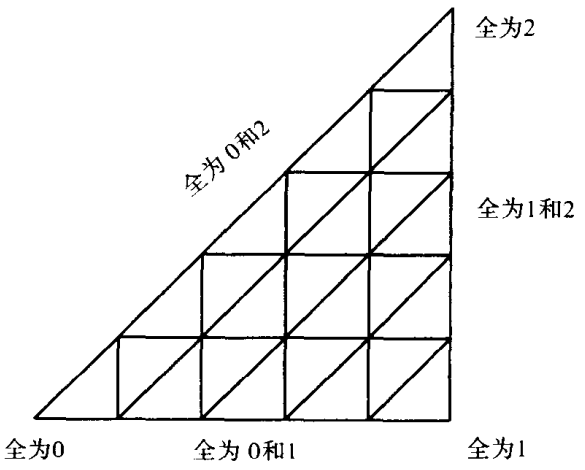


图7-1　k=2的Bermuda三角形

对于k>2，我们需要一个k维的三角形。幸运的是，这样的问题在代数拓扑学中早已存在了。它叫做k维的单纯形（*simplex*）。比如，一维的单纯形就是一条边，二维的单纯形就是一个三角形，三维的单纯形是一个四面体。（到了三维以上，单纯形就很难想像了。）

所以对于任意k，我们从k维的欧几里德空间中的k维单纯形开始。这些单纯形包含了一些格点(*grid point*)，格点是欧氏空间中有整数坐标的点。要得到k维*Bermuda*三角形B，可以根据这些格点来对这个单纯形进行三角分割，从而得到小的k维单纯形的集合。

证明中首先为B中的每一个顶点（格点）分配一个运行。在运行中，所有的进程从{0，1，...，k}中的相同输入开始，而且在B的k+1个角上的顶点上没有故障。例如，当k=2时，我们为左下角顶点分配一个所有进程输入值为0的运行，为右下角顶点分配一个所有进程输入值为1
166 的运行，为右上角顶点分配一个所有进程输入值为2的运行（如图7-1所示）。而且，对于B的（任意维的）任意面上的每一顶点x来说，分配给x运行的那些输入值就是那些出现在分配给这个面的角的运行中的值。例如，当k=2时，分配给底边顶点的所有运行的输入值都是从{0，1}中选取的。

接下来，对B中的每一个顶点，我们给它分配某个无故障进程的索引，其中进程处于分配给这个顶点的运行中。进程分配方法使得，对于每一个小单纯形T，存在一个运行α，它最多有f个故障，而且它与分配在T的角上的运行和进程满足下列关系：

1）分配给T的角上的所有进程在α中是无故障的。

2）如果运行α'和进程i分配在T的某个角上，那么对于i来说，α和α'是不可区分的。

这种对B的顶点的运行和进程的分配有一些很好的性质。假设α和i与顶点x相联系。如果x是B的一个角，那么α中所有的进程以相同的输入值开始，因此根据有效性条件，i在α中必然决定出这个值。如果x在B的外边的一条边上，那么在α中，每一个进程的输入值都是与边的两个（在B的角上的）端点相联系的两个输入值之一；有效性条件意味着i必须决定出这两个值之一。更一般的，如果x在B的（任意维）的任意面上，那么在α中，每一个进程开始时的输入值，都是与这个面上的角相联系的输入值之一；有效性条件意味着i必须决定出这些值之一。最后，如果x在B的内部，那i允许决定出k+1个值中任意一个。

我们按以上描述给顶点分配运行和索引的能力依赖于在每一个运行中轮数r最多为$\left\lfloor \frac{f}{k} \right\rfloor$的事实，也就是说，$f \geq rk$。这是因为运行的分配使用了定理6.33的证明中讨论的链的k维一般化。构造过程对k维中的每一个使用了r个进程故障。

在给节点分配运行和索引之后，我们给每一个顶点用集合{0，...，k}中的一种“颜色”着色。也就是说，对于和运行α与进程i相联系的顶点x，我们用对应α中i的决定值的颜色对x着色。这种着色有以下性质：

1）B的k+1个角上的颜色都不同。

2）B的外部边上的每一个节点的颜色，与边的端点的角的颜色之一相同。

3）更一般的说，在B的（任意维的）任意外部面上，每一个节点的颜色是这个面的角的颜色之一。

167

在代数拓扑学领域，已经对具备这些性质的k单纯形着色问题进行了研究，叫做*Sperner*着色。

在这里，我们先使用一个著名的组合学结论，就是在1928年证明的*Sperner*引理，这个结论说明，对一个三角分割的k维单纯形的任意Sperner着色，必然包含至少一个小单纯形，它的k+1个角被染上k+1种不同的颜色。在这里，这个单纯形对应一个运行，其中最多有f个故障，k+1进程选择k+1个不同的值。但是这与k一致性问题的一致性条件相矛盾。

因此假设的算法不存在，也就是说，对k一致性问题，不存在一个算法可以容许f个故障并在$r \leqslant \left\lfloor \frac{f}{k} \right\rfloor$轮内停止。本节剩余部分包括更多的细节。

定义 我们使用6.7节中的通信模式定义。现在我们重新定义的良好的通信模式为：对所有的三元组（i, j, k）有$k \leqslant r$，并且丢失的元组和停止故障模型是一致的。（也就是说，我们使用6.7节中良好的通信模型的定义中的前两个条件，但现在轮数的上界是r而不是f。目前，我们不限制故障的数目。）基于这个良好通信模式的新的定义，我们定义运动ρ和$exec(\rho)$，使用的方法和6.7节相同。如果一个进程i在某个t+1轮或这以后的轮不发送消息，我们就说在一个运行中t轮后i是*安静的*。

Bermuda三角形 我们从k维欧氏空间中的k单纯形开始，它的角上的顶点是长度为k的向量$(0, \ldots, 0), (N, 0, \ldots, 0), (N, N, 0, \ldots, 0), \ldots, (N, \ldots, N)$，其中$N$是下面要定义的大的整数。*Bermuda*三角形$B$是这个单纯形加上如下分割的小单纯形。$B$的顶点是包含在单纯形内的格点，即形如$x=(x_1, \ldots, x_k)$的点，其中向量的分量是0到$N$之间满足$x_1 \geqslant x_2 \geqslant \ldots \geqslant x_k$的整数。小单纯形定义如下：选取任意一个格点，以任意顺序在每一维上沿正方向走一步。将在走动过程中遍历的点定义为小单纯形的顶点。

例7.1.2 Bermuda三角形中的节点的坐标

二维Bermuda三角如图7-2所示。

168

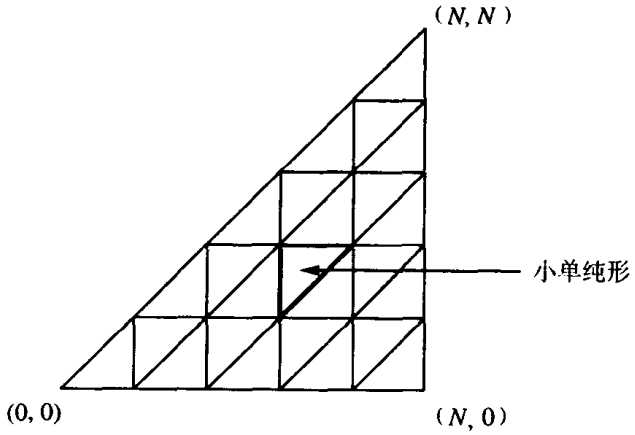


图7-2 二维Bermuda三角形

用运行和运动标注B 在本节中，我们描述怎样为B中的顶点分配运行（即用运行“标注”顶点）。首先我们要扩展运行，给运行中的某个（进程，轮数）对（i,t）附上令牌（*token*）。这样的令牌被认为是允许进程i在t轮或以后发生故障。可能有一个以上的令牌被附上相同的数对（i，t）。

更准确的说，对任意l>0，我们定义l运动（l-run）为一个增强的运动，其中对于每一轮t，$1 \leqslant t \leqslant r$，有$l$个令牌。扩展的方法是，如果进程$i$在某个$t$轮失败，那么就把一个令牌附到某个数对（$i$，$t'$）上，其中$t' \leqslant t$。一个$l$运动共包含$lr$个令牌。我们只关心$l$=1和$l$=$k$的两种情况，即1运动和$k$运动。我们定义*无故障l运动*(*failure-free l-run*)为没有故障并且所有的令牌附在形如（1，t）的数对上（也就是说只允许进程1失败）的l运动。

因为每一个扩展运动都是由一个运动构造的，所以每个扩展运动都明显地产生一个运行。我们扩展记号$exec(\rho)$，以前它是为运动定义的，现在ρ是扩展的运动。为了用运行标注B的顶点，我们用k运动来标注它们。

现在我们定义l运动上的四个操作，每一个只有很小的差别。每一个操作只能删除或添加
169 一个三元组，改变一个进程的输入值，或者在同一轮中有相邻的索引的进程之间移动令牌。这些操作与定理6.33的证明中用到的很相似。四种操作定义如下：

1）$remove(i,j,t)$，其中i是进程索引，t是轮数，$1 \leqslant t \leqslant r$。

如果元组(i,j,t)存在（元组表示t轮中从i到j的消息），这个操作删除它，否则没有影响。只有当i和j在t轮之后是安静的，并且有一个令牌附在某个(i, t')上时（$t' \leqslant t$），才能使用这个操作。

2）$add(i,j,t)$。

如果元组(i,j,t)不存在，这个操作增加元组，否则没有影响。只有当i和j在t轮之后是安静的，并且i在$t-1$轮之后是活动的，才能使用这个操作。

3）$change(i,v)$。

如果i的输入值是v，这个操作没有影响，否则它把i的输入值变成v。只有当i在0轮之后是安静的，并且$(i, 1)$有一个令牌时，才能使用这个操作。

4）$move(i,j,t)$。

这个操作把一个令牌从(i, t)移动到(j, t)，其中j是$i+1$或者$i-1$，只有当(i, t)有一个令牌，并且所有的故障都能从其他令牌得到允许时，才能使用这个操作。

从这些定义可以明显看出，当这些操作的任意一个用于l运动时，结果仍然是l运动。

现在，对任意$v \in \{1,...,k\}$，我们可以定义一个$remove$，add，$change$，$move$四种操作的序列$seq(v)$，这些操纵可以用于任意无故障1运动ρ，把ρ转换成所有进程输入为v的无故障1运动。事实上，同一序列$seq(v)$可以用于所有的无故障1运动ρ。这可以通过定理6.33证明中所用的方法得到；主要的区别是明确移动允许故障的令牌。在这个构造过程中，从进程1开始，每一次有一个进程的输入变成v。和以前一样，构造过程在r轮中使用了r个故障。

结果表明，对不同的v，构造的序列$seq(v)$是同构的——也就是说，除了v的选择，它们是相同的。现在我们可以对定义B的大小时用到的参数N做出定义：N是（对任意v）序列$seq(v)$的长度。

我们将用几个序列$seq(v)$来标注B的顶点。元素的值域是0，1，...，k。对每一个$v \in \{0,1,...,k\}$，定义τ_v所有进程初始值为v的无故障1运动。对无故障1运动τ_{v-1}，我们使用每一个序
170 列$seq(v)$，$1 \leqslant v \leqslant k$，产生一个1运动的序列，以作为第$v$维（$v$轴）的$B$的边上的顶点的初步标号（preliminary label）。把B的每一个顶点x在k轴上的投影的初步标号的k个1运动合并起来，得到k运动，并分配给顶点x。

例7.1.3　用k运行标注Bermuda三角形

为了直观地说明合并过程，我们对$k=2$（因此$V=\{0,1,2\}$），$n=5$的情况给出一个简单的图形。如图7-3所示。

图形中没有描绘出所有顶点，只画出了那些用无故障k运动标注的顶点。这样我们需要提供的信息只是每一个画出的顶点的输入值的向量。标注B的每一个角的k运动是一个无故障k运动，其中所有的输入值相等，0在左下角，1在右下角，2在左上角。沿着水平轴的序列是由$seq(1)$构造的，从全是0到全是1，而沿着垂直轴的序列是由$seq(2)$构造的，从全是1到全是2。

注意B中出现的输入的模式。沿着水平轴，从进程1开始，进程的输入从0变到1。沿着垂直
171 轴，从进程1开始，进程的输入从1变到2。在B的内部，同时在两个方向上发生变化。例如，考

虑由输入变量22100指出的内部顶点。标注它在水平轴和垂直轴上的投影的向量分别是11100和22111。通过把最后两个进程的输入从0改为1，在B中水平移动，向量22100可以变成22111。同样，通过把开始两个进程的输入从1改为2，在B中垂直移动，向量11100可以变成22100。标注B中每一个节点的向量由{0，1，2}中的值组成，而且以非递增顺序出现。

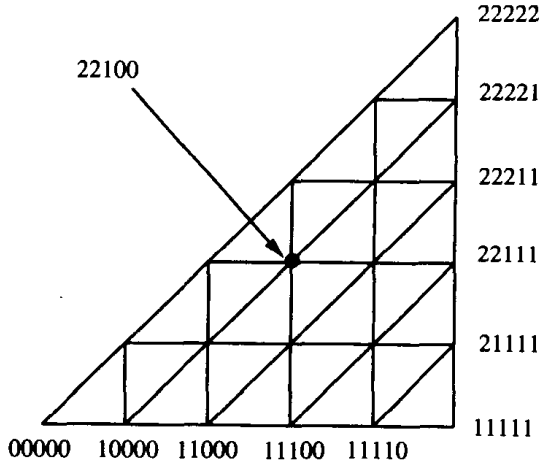


图7-3 用k运动标注Bermuda三角形

现在我们给出合并的形式化定义。1运动的序列$\sigma_1,...,\sigma_k$的合并是如下定义的k运动ρ

1）在ρ中进程i有输入值v，其中v是$\{1,...,k\}$中的使i在σ_v中有输入值v的最大值。如果没有这样的v，则为0。

2）如果三元组(i,j,t)包含在所有的σ_v中，$1 \leqslant v \leqslant k$，那么它也包含在$\rho$中。

3）在ρ中分配给数对（i，t）的令牌数，是所有的σ_v中分配给数对（i，t）的令牌数之和。

对于第一个条件，我们重新考虑所用的合并操作。每个σ_v是通过对τ_{v-1}使用$seq(v)$的一个前缀得到的。在这个序列中的某一点，进程i的输入值从$v-1$变成v。如果这在σ_v中已经发生了，那么我们说进程i在v维中已经“转变”了。第一个条件就是选择使进程i在v维转变的最大的v（如果有的话）。

第二个条件是说，一条消息在新的运动ρ中丢失，当且仅当这条消息在某个被合并的运动σ_v中丢失。第三个条件说明了令牌的累积。不难看出一系列1运动的合并实际上是一个k运动。

现在，我们概括起来，说明怎样用k运动来标注B的顶点。令$x=(x_1,...,x_k)$为B的任意顶点。对每一$v \in \{1,...,k\}$，令σ_v为通过对τ_{v-1}使用$seq(v)$的前x_v个操作得到的1运动。那么标注x的k运动是$\sigma_1,...,\sigma_k$的合并。注意，在合并的运动中最多有$rk \leqslant f$个令牌，因此最多有f个故障。在证明的剩余部分，我们固定用k运动（和运行）来标注B。

在本小节的最后，对于标注B中任意小单纯形的节点的k运动，我们给出它们之间的紧密联系。令$y_0,...,y_k$为T的顶点，其顺序是由（在Bermuda三角形的定义中描述的）产生T的方法决定的。令$\rho_0,...,\rho_k$为标注这些顶点的k运动。 172

第一个引理说明，如果任意一个进程在这些k运动中的其中一个中出错，那么它必然在所有的k运动中有一个令牌。

引理 7.4 如果进程i在某个ρ_v，$0 \leqslant v \leqslant k$中发生故障，那么$i$在每一个$\rho_v$中都有一个令牌。

证明概要 这是因为在每一个序列seq中的变化是逐步进行的，特别是因为令牌的移动和三元组的删除发生在两个分开的步骤中。详细的证明留做一道习题。 □

第二个引理限制了所有运行中的总故障数。

引理7.5 对任意$v \in \{0,...,k\}$，令F_v表示在ρ_v中发生故障的进程的集合。令$F=\cup_v F_v$。那么$|F| \leqslant rk \leqslant f$。

证明 留做一道习题。证明中要使用引理7.4。 □

最后，我们考虑用进程的索引来标注T的顶点。T的本地进程标注（*local process labelling*）是指，把不同进程的索引$i_0,...,i_k$分配给T的顶点$y_1,...,y_k$，使得对每一个v，i_v在ρ_v中没有令牌。标

注T的顶点的k运动的最后一个重要性质是，如果T有一个本地进程标注，那么T和一个运动一致。

引理7.6　令$i_0,\ldots,i_k$是T的本地进程标注。那么存在一个运动ρ，它最多有f个故障，对所有的v，i_v在ρ中是无故障的，并且$exec(\rho_v)$和$exec(\rho)$对进程i_v是不可区分的。

证明概要　我们定义ρ如下。对ρ中的每个进程i，我们定义初始值为i在ρ_v中任意一个中的初始值。对$1\leqslant t\leqslant r-1$，如果三元组（$i$，$j$，$t$）在所有的$\rho_v$中，那么我们把它包含在$\rho$中。同样，如果（$i$，$j$，$r$）在所有的$\rho_v$中，接收者$j$不是进程$i_v$中的一个，那么我们也把（$i$，$j$，$r$）包含进来。最后，对（$i$，$j$，$r$），其中（对特定的$v$）$j=i_v$，如果它（对同一$v$）在$\rho_v$中，我们也把它包含进来。

对ρ的性质的证明留做一道习题。它是一个所需的运动，它最多有f个故障，对每个v，i_v在
173 ρ中是无故障的，$exec(\rho_v)$和$exec(\rho)$对进程i_v是不可区分的。证明使用引理7.5来限制故障数。□

以进程索引标注B　我们要做的是给B的顶点分配进程索引，使得对每一个小单纯形T，存在一个运行，它与标注T的顶点的运行和进程兼容。引理7.6给出了一种做法：对B的每个顶点，我们选取一个在对应k运动中没有令牌的进程，使得为任意小单纯形的顶点选取的进程都不相同。那么，引理7.6就意味着对每个小单纯形满足兼容性条件。

我们定义B的*全局进程标注*（*global process labelling*）为B的节点的一个进程分配，使得对每个顶点x，分配给x的进程在标注x的k运动中没有令牌，并且对每个小单纯形T，分配给T的顶点的所有进程是不同的。对B的全局进程标注，导致对B的每个小单纯形的本地进程标注。

现在我们构造一个B的全局进程标注。（因为这种构造是理论上的，你可以在第一次阅读时跳过这部分，直接看引理7.10。）在构造过程中，我们首先把一个进程的集合$live(\rho)$和标注B的顶点的每个k运动ρ联系起来。然后从每个$live(\rho)$集合中选择一个进程。集合$live(\rho)$满足如下性质：

1）每个集合$live(\rho)$包含$n-rk$个进程。（因为我们已经假设$n\geqslant f+k+1$，$f\geqslant rk$，所以每个$live(\rho)$集合最多包括$k+1$个进程。）

2）$live(\rho)$中的进程是从那些在ρ中没有令牌的进程中选取的。

3）如果ρ和ρ'是标注B中同一个小单纯形的两个顶点的两个k运动，如果进程$i\in live(\rho)\cap live(\rho')$，那么$i$在两个集合中有相同的等级[⊖]。

所以固定某个k运动ρ。它包含rk个令牌；令*tokens*为进程索引的多元素集合，描述与每个进程相联系的令牌数。我们把多元素集合*tokens*“压扁”，得到一个新的多元素集合*newtokens*，它有相同的令牌数，但是其中只有一个令牌和某个进程相联系。并且，任意在*tokens*中有一个令牌的进程也在*newtokens*中有一个令牌。这个压扁的过程是如下工作的。

压扁的过程

newtokens := *tokens*
while *newtokens* **has a duplicate element do**
　select such an element, say i
174 **if there exists** $j<i$ **such that** $newtokens(j)=0$, **then**
move a token from i **to the largest such** j
else move a token from i **to the smallest** $j>i$ **such that** $newtokens(j)=0$

然后我们定义$live(\rho)$为使$newtokens(i)=0$的所有进程i。

容易看出，*live* 的这个定义满足以上要求的前两个性质。对第三个性质，我们固定一个小单纯形T，令$y_0,\ldots,y_k$为T的顶点，顺序由产生T的过程决定，并且，令$\rho_0,\ldots,\rho_k$为标注这些顶点的k

⊖　元素i在一个有限的全序的集合L中的等级，是指L中小于或者等于i的元素数。

运动。首先，我们注意到，当我们按顺序遍历T的节点的时候，如果进程i得到一个令牌，那么在以后的遍历中它总是有令牌。

引理7.7 令$v<v'<v''$。如果进程i在$\rho_{v'}$中没有令牌，但在$\rho_{v''}$中有一个令牌，那么i在$\rho_{v'}$中有一个令牌。

证明 留做一道习题。 □

现在我们可以证明*live*集合的第三个性质。

引理7.8 如果$i \in live(\rho_v) \cap live(\rho_w)$，那么$i$在$live(\rho_v)$和$live(\rho_w)$中有相同的等级。

证明 不失一般性，假设$v<w$，因为$i \in live(\rho_v)$并且$i \in live(\rho_w)$，所以i在ρ_v和ρ_w中都没有令牌。那么由引理7.7，i在$\rho_v, ..., \rho_w$中的任意一个运动中都没有令牌。

因为放在相邻k运动中的令牌的区别最多是一个令牌从一个进程到相邻进程的移动，而且i在这些运动中的任意一个中都没有令牌，所以在比i小的进程上的总令牌数，比如说s，在所有的运动$\rho_v, ..., \rho_w$中是相等的。因为$i \in live(\rho_v)$，*Flatten*过程的工作方法意味着$s<i$。（如果$s \geq i$，那么进程上开始比i小的令牌将在*Flatten*过程中溢出，使得一个将会在i结束。）因此，i在$live(\rho_v)$和$live(\rho_w)$中有相同的等级$i-s$。 □

现在我们可以用进程的索引来标注B的顶点。令$x=(x_1,...,x_k)$是B的任意顶点，令ρ是它的k运动；我们从集合$live(\rho)$中选择一个特殊的进程索引。也就是说，令$plane(x)=\sum_{i=1}^{k} x_i \pmod{k+1}$；我们用$live(\rho)$中等级为$plane(x)$的进程来标注$x$。这种选择是由以下关于$B$的事实得到的。

引理7.9 如果x和y是同一小单纯形的不同顶点，那么$plane(x) \neq plane(y)$。 175

现在我们达到目的。

引理7.10 B的这个用进程索引的标注是一个全局进程标注。

证明 因为每个节点x的索引是从集合$live(\rho)$中选取的，其中ρ与k运动相联系，因此那个索引在ρ中必然没有令牌。对任意固定的小单纯形T来说，引理7.8、引理7.9放在一起意味着选取的索引都不相同。 □

我们总结一下，对于我们构造的标注，我们知道什么。

引理7.11 给定的B的用k运动和进程的标注有以下性质。对每个运动标号为$\rho_0,...,\rho_k$，进程标号为$i_0,...,i_k$的小单纯形T，存在一个运动ρ，它最多有f个故障，对所有v，i_v在ρ中是无故障的，$exec(\rho_v)$和$exec(\rho)$对进程i_v是不可区分的。

证明 这个可以由引理7.6、引理7.10得到。 □

Sperner 引理 我们即将得出证明了。只剩下描述Sperner 引理（用于Bermuda三角的特殊情况），并且用它得到一个矛盾。这将会得到解决k一致性问题所需轮数的下界。B的一个Sperner着色是给B的每个顶点分配$k+1$种颜色中的一种，使得

1）B的$k+1$个角上的颜色都不相同。

2）在B的外部边上的每个点的颜色，是边的两个端点的角的颜色之一。

3）更一般的说，在B的（任意维的）一个外部面上的每个内部节点的颜色，是B的相邻的角的颜色之一。

Sperner 着色有一个重要的性质：必然存在至少一个小单纯形，它的$k+1$个顶点用所有$k+1$种颜

色着色。

引理7.12（*B*的Sperner 引理）　对B的任意Sperner着色，在B中至少存在一个小单纯形，它的$k+1$个角都用不同的颜色着色。

现在回想假定的k—致性算法A，假定它容许f个故障，并且最多在$\left\lfloor \frac{f}{k} \right\rfloor$轮停止。我们定义$B$的一个着色$C_A$如下。给定一个用运动$\rho$和进程$i$标注的顶点$x$，用进程$i$在$A$的运行$exec(\rho)$中的决
176 定对x着色。

引理7.13　如果A是k—致性算法，它容许f个故障，并且在$\left\lfloor \frac{f}{k} \right\rfloor$轮停止，那么$C_A$是$B$的一个Sperner着色。

证明　根据k—致的有效性条件。　□

现在我们证明主要的定理。

定理7.14　假设$n \geqslant f+k+1$，那么不存在n个进程的k—致性算法，它容许f个故障，所有的无故障进程总是在$\left\lfloor \frac{f}{k} \right\rfloor$轮内做出决定。

证明　引理7.13暗示C_A是一个Sperner着色，所以Sperner引理和引理7.12意味着存在一个小单纯形T，它的所有顶点通过C_A染上了不同的颜色。

假设T的k运动的标号是$\rho_0,\ldots,\rho_k$，它的进程标号是$i_0,\ldots,i_k$。根据C_A的定义，这意味着所有的$k+1$个不同决定是由$k+1$个进程i_v分别在运行$exec(\rho_v)$中做出的。但是引理7.11指出，存在一个最多有f个故障的运动ρ，使得对所有的v，在ρ中i_v是无故障的，并且对进程i_v来说，$exec(\rho_v)$和$exec(\rho)$是不可区分的。但是这意味着在ρ_v中，$k+1$个进程$i_0,\ldots,i_k$决定$k+1$个不同的值，从而违反了k—致性问题的一致性条件。　□

7.2　近似一致性

现在我们考虑存在Byzantine故障时的近似一致问题。在这个问题中，进程从实数值输入开始，最终要决定出实数值的输出。它们可以在消息中发送实数值数据。这里的要求不像是在普通一致性问题中那样必须达成确切的一致，而是要求它们的决定在一个小的正实数范围ε内。更精确的说，这里要求

一致性　任意一对无故障进程的决定值，相互之间的差别都在ε内

有效性　任意无故障进程的决定值都在无故障进程的初始值范围之内。

177 **终止性**　所有的无故障进程最终做出决定。

例如，这个问题会在时钟同步算法中出现，其中每个进程需要保持一个时钟值，这些值很接近，但不必确切一致。许多实际的分布式网络算法都在有近似同步时钟的情况下工作，所以时钟值的近似一致通常很重要。

这里我们只考虑在完全图中的近似一致性问题。解决问题的一种方法就是用普通Byzantine一致性算法作为子例程。解决方案假设$n>3f$。

***ByzApproxAgreement* 算法：**

每个进程运行普通Byzantine一致性算法决定一个值。所有这些算法并行运行。在进程i的算法中，i在第一轮中向所有进程发送消息，然后所有进程把收到的值作为它们在Byzantine一致性算法中的输入。当这些算法终止时，所有无故障进程对所有进程有相同的决定值。每一个

进程在其决定值多元素集合中选出第 $\left\lfloor \frac{n}{2} \right\rfloor$ 个最大值来作为最终决定值。

要说明它的正确性，注意如果i是无故障的，那么Byzantine一致的有效性条件保证所有无故障进程得到的i的值都是i的实际输入值。因为$n>3f$，所以多元素集合中的中间值必然在所有无故障进程的初始值范围内。

定理7.15 *ByzApproxAgreement* 解决了n节点完全图的近似一致性问题，其中$n>3f$。

现在我们给出第二种解法，它不使用Byzantine一致。我们给出这种解法的主要原因是它很容易扩展到异步网络模型中，这会在第21章中讲到。而Byzantine一致性问题不能在异步网络中解决。第二种解法还有一个性质，就是有时它能在少于Byzantine一致所需的轮数内终止，这取决于无故障进程的初始值距离多么远。这个算法是基于连续近似的。为简单起见，我们先描述算法的不终止情况，然后再讨论终止性。算法同样假设$n>3f$。

我们还需要一些记号和术语：首先，如果U是一个至少有$2f$个元素的有限实数值多元素集合，$u_1,\ldots,u_k$是U的元素按非递减顺序的排序，那么令$reduce(U)$表示从U中删除f个最小的和f个最大的元素后的结果，即$u_{f+1},\ldots,u_{k-f}$组成的多元素集合。同样，如果U是一个非空有限实数值多元素集合，$u_1,\ldots,u_k$是U的元素按非递减顺序的排序，那么令$select(U)$为$u_1,u_{f+1},u_{2f+1},\ldots$组成的多元素集合，即它包含$U$的最小元素和以后的每第$f$个元素。最后，如果$U$是一个非空有限实数值多元素集合，那么$mean(U)$是$U$中元素的平均值。 178

我们也说，一个实数的非空有限多元素集合的范围（*range*）是包含所有元素的最小区间，这个多值集的宽度（*width*）是范围区间的大小。

第二种解法如下：

***ConvergeApproxAgreement*算法：**

进程i保存一个包含它最新的预测值的变量val。开始时，val_i包含i的初始值。在每一轮，进程i做如下工作。

首先，它向所有的进程，包括它自己[⊖]，广播它的val值。然后把那一轮收到的所有值加到多元素集合W中；如果进程i没有收到另外某个进程的值，那么它选择任一缺省值分配给多元素集合中的那个进程，从而保证$|W|=n$。

然后，进程i把val设置为$mean\ (select\ (reduce\ (W)))$。也就是说，进程$i$去掉$W$中$f$个最大的和$f$个最小的元素。从剩下的元素中，$i$选出最小的元素和以后每第$f$个元素。最后$val$设置为选取的元素的平均值。

我们要证明：在任意轮中，所有无故障进程的val都在上一轮的无故障进程的val范围内。而且，在每一轮中，无故障进程的val的多元素集合的宽度，被一个至少为 $\left\lfloor \frac{n-2f-1}{f} \right\rfloor + 1$ 的因子缩减。如果$n>3f$，这个因子大于1。

引理7.16 假设在*ConvergeApproxAgreement*的一个运行r轮后，有$val_i=v$，其中i是无故障进程。那么在r轮之前，v在无故障进程的val的范围之内。

证明 如果W_i是进程i在r轮得到的多元素集合，那么W_i中最多有f个元素不是无故障进程送来的值。那么在r轮之前，$reduce(W_i)$中的所有元素在无故障进程的val的范围之内。因此对于val_i的

⊖ 和通常一样，给自己发送消息是通过本地转换模拟的。

新值$mean(select(reduce(W_i)))$，情况是相同的。 □

179

引理7.17 假设在*ConvergeApproxAgreement*的一个运行的r轮后，有$val_i=v$，$val_{i'}=v'$，其中i和i'是无故障进程。那么

$$|v-v'| \leq \frac{d}{\left\lfloor \frac{n-2f-1}{f} \right\rfloor + 1}$$

其中d是r轮前无故障进程的val范围的宽度。

证明 令W_i和$W_{i'}$分别为r轮中进程i和i'得到的多元素集合。令S_i和$S_{i'}$分别是多元素集合$select(reduce(W_i))$和$select(reduce(W_{i'}))$。令$c=\left\lfloor \frac{n-2f-1}{f} \right\rfloor+1$。注意，$c$是$S_i$和$S_{i'}$中的元素数。令$S_i$中的元素用$u_1,\ldots,u_c$表示，$S_{i'}$中的元素用$u'_1,\ldots,u'_c$表示，两者都以非递减序排列。首先我们给出一个断言，说明缩减的多元素集合最多有f个元素不同。

声明7.18 $|reduce(W_i)-reduce(W_{i'})| \leq f$。

证明 因为无故障进程向W_i和$W_{i'}$提供相同的值，所以$|W_i - W_{i'}| \leq f$。可以证明，从两个多元素集合中删除最小元素不会使不同元素的数目增加，而且删除最大元素也如此。使用这两个事实f次就得到了结论。 □

声明7.18可以用来证明

声明7.19 对于所有的j，$1 \leq j \leq c-1$，有$u_j \leq u'_{j+1}$，$u'_j \leq u_{j+1}$。

证明 我们只证明第一个结论；第二个是对称的。注意到，u_j是$reduce(W_i)$中由小到大第$((j-1)f+1)$个元素，而u'_{j+1}是$reduce(W_{i'})$中由小到大第$(jf+1)$个元素。根据断言7.18，因为最多有f个$reduce(W_i)$中的元素不是$reduce(W_{i'})$中的元素，所以必然有$u_j \leq u'_{j+1}$。 □

现在我们通过计算所需的轮数来完成对引理7.17的证明，我们有

$$\begin{aligned}
|v-v'| &= |mean(S_i) - mean(S_{i'})| \\
&= \frac{1}{c}|(\Sigma_{j=1}^{c}(u_j - u'_j))| \\
&\leq \frac{1}{c}(\Sigma_{j=1}^{c}|u_j - u'_j|) \\
&= \frac{1}{c}(\Sigma_{j=1}^{c}(\max(u_j,u'_j) - \min(u_j,u'_j)))
\end{aligned}$$

180

根据声明7.19，对于所有的j，$1 \leq j \leq c-1$，有$\max(u_j, u'_j) \leq \min(u_{j+1}, u'_{j+1})$。所以后面的表达式小于或等于

$$\frac{1}{c}(\Sigma_{j=1}^{c-1}(\min(u_{j+1},u'_{j+1}) - \min(u_j,u'_j))) + \frac{1}{c}(\max(u_c,u'_c) - \min(u_c,u'_c)),$$

这就得到

$$\frac{1}{c}(\max(u_c,u'_c) - \min(u_1,u'_1))$$

但是，在第r轮之前，所有的值u_i、u'_i、u_j和u'_j都在无故障进程的val的范围之内，原因在于$reduce(W_i)$和$reduce(W_j)$中所有的元素都在这个范围之内。因此，最后的表达式小于或等于$\frac{d}{c}$。

终止性 我们把*ConvergeApproxAgreement*算法改成一个终止性算法，其中所有进程最终都做出决定。（实际上，所有的进程最终停止。）每个无故障进程使用它在第一轮收到的所有值的范围来，计算一个轮数，确保在这轮之前任意两个无故障进程的val最多相距ε。每个进程都能做到这点，因为它知道ε的值，知道可以保证的收敛速率，而且，它知道它在第一轮收到的值的范围包含所有无故障进程的初始值。然而，不同的无故障进程可能计算出不同的轮数。

任意一个进程i，达到它计算的轮数时决定它自己当前的val。在这之后，进程i广播它的val，其中带一个特殊的*halting*标记，然后停止。任意进程j收到i的带有*halting*标记的val之后，j就把这个val作为它从i中收到的消息，不仅在当前这一轮，而且在以后的所有轮中都如此。（直到j根据它自己计算的轮数决定停止。）

尽管无故障进程计算的轮数可能不同，但是这样的最小的估计是正确的。因此，在第一个无故障进程停止的时候，val的范围已经足够小了。在以后的轮中，尽管不能保证无故障进程的val的范围不断减小，但是这个范围绝不会增大。

定理7.20 有了上面加上的终止性，在$n>3f$的情况下，*ConvergeApproxAgreement*算法可以解决n个节点的完全图中的近似一致性问题。

复杂度分析 在*ConvergeApproxAgreement*算法中，对于所有无故障进程做出决定的时间，不存在一个依赖于n、f、ε和无故障进程的初始值的宽度的上界。这是因为故障进程可以在第一轮 181
发送随意的值，这可能导致无故障进程计算任意多轮数才终止。

习题讨论了解决近似一致性问题所需的进程数和连接度的限制。我们将在第21章的异步网络环境中再次考虑这个问题。

7.3 提交问题

在关于同步系统的分布式一致性问题的最后一节中，我们给出关于分布式数据库提交问题的一些关键思想。正如在5.1节中讨论的，当一个进程集合参与数据库事务处理时，就会有这个问题。处理完成之后，每个进程就这个事务是否要提交（即它的结果永久确定，并且发送给其他事务使用。）或者中止（即丢弃它的结果）形成初始“意见”。如果一个进程的所有的代表那个事务的本地计算已经成功完成，那么它倾向于提交事务，否则它倾向于中止事务。进程要互相通信，最终就提交和中止中的一个输出达成一致。如果可能的话，输出应该是提交。

对于实际的分布式网络，这个问题已经有了解法，其中可以同时有进程故障和链路故障。然而，第5章说明，在链路故障不受限制的情况下，不存在这样的解法。因此必须假设消息的丢失有一定限制。

7.3.1 问题

我们考虑一个简化的提交问题，网络中没有消息丢失，只有进程故障。如果你有兴趣在一个实际系统中实现本章的算法，那么你必须增加一些其他的机制（如重复事务处理）来处理丢失的消息。我们允许任意数量的停止故障。

我们假设输入域是{0，1}，其中1代表提交，0代表中止。在这里，我们把注意力集中在网络是完全图的情况。正确性条件是

一致性 没有两个进程决定出不同的值。 182

有效性

1）如果任一进程从0开始，那么0是惟一可能的决定值。

2）如果所有的进程从1开始，并且没有故障发生，那么1是惟一可能的决定值。

终止性　这里有两种形式。*弱终止性条件*，即如果没有故障，那么所有的进程最终做出决定。*强终止性条件*（也称*非阻塞条件*），即所有的无故障进程最终做出决定。

满足强终止性条件的提交算法有时被称为*非阻塞提交算法*，满足弱终止性条件而不满足强终止性条件的提交算法有时被称为*阻塞提交算法*。

注意，我们的一致性条件是，没有两个进程决定出不同的值。因此，即使是故障进程，我们也不允许它的决定值与其他进程不同。我们做这样的要求是因为，在一个提交协议的实际应用中，一个进程可能发生故障然后恢复。举个例子，假设进程i在它发生故障之前决定提交，之后，另一个进程决定中止。如果进程i恢复并保持它的决定为提交，那么将会产生不一致。

这个问题的形式化描述和我们已经考虑过的另外两个——5.1节的协同攻击问题和6.1节的停止故障的一致性问题——是很相似的。提交问题和协同攻击问题的最大区别是，这里我们考虑的是进程故障，而不是链路故障；有效性条件也有区别。提交问题和停止一致性问题之间的最重要的区别是：特殊的有效性条件的选择；考虑到弱终止性。6.7节中的关于停止一致性问题的结论意味着，要解决有强终止性条件的提交问题，需要的轮数的下界是$n-1$。（注意，对提交的有效性条件，定理6.33的证明仍然起作用。）

在本节的剩余部分，我们（对只有进程故障的简化的环境）给出两种标准实用提交算法。第一个是*两阶段提交*，这是一个阻塞算法；第二个是*三阶段提交*，它是一个非阻塞算法。然后我们给出解决问题（即使只要求弱一致性）所需消息数的下界。

183

7.3.2　两阶段提交

最著名的实用的提交算法是*两阶段提交*（*TwoPhaseCommit*算法），勿庸讳言，这个简单的算法只能保证弱终止性。

***TwoPhaseCommit*算法：**

算法假设有一个与众不同的进程，比如说进程1。

第1轮：除了进程1，所有进程都向进程1发送它们的初始值，任意初始值为0的进程决定出0。进程1把所有收到的值和它自己的初始值放入一个向量中。如果这个向量中的所有位置都是1，那么进程1决定出1。否则，如果向量中某个位置包含0，或者某个位置没有填充（因为没有从对应的进程收到消息），那么进程1决定出0。

第2轮：进程1向其他所有进程广播它的决定。除了进程1，任意在第2轮中收到消息并且在第1轮不做决定的进程都决定出它收到的消息中的值。

见图7-4，它给出了一个例子，说明在*TwoPhase Commit*算法的无故障运行中的通信模式。⊖

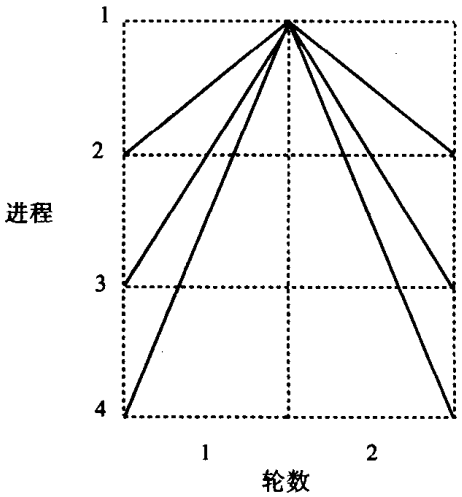


图7-4　*TwoPhaseCommit*算法中的通信模式

定理7.21　*TwoPhaseCommit*算法解决了弱终止条件下的提交问题。

⊖　我们轮数的指定不确切对应通常的两阶段提交协议的阶段的指定。一般情况下，开始时要加一个额外的轮，在这轮中，进程1从其他进程要求提交或中止值。那么第一阶段由这个额外的轮和我们的第1轮组成。对我们的简化模型和问题描述，不需要这个额外的轮。

证明 一致性，有效性和弱终止性都容易证明。 □

然而，*TwoPhaseCommit*算法不满足强一致性条件，也就是说，它是一个阻塞算法。这是因为，如果进程1在第二轮开始广播之前发生故障，那么就没有初始值为1的无故障进程做出决定。实际上，如果进程1发生故障，余下的进程通常使用它们之间的某种终止协议，并且有时能做出决定。例如，如果进程1发生故障，但另外某个进程i已经在第一轮决定了0，那么进程i可以告诉剩下的无故障进程它的决定是0，因此它们也可以安全地决定0。但是，终止协议不是在所有的情况下都能成功。例如，假设除了进程1之外，所有进程从输入1开始，但是进程1在没有发送任何消息之前就发生故障了。那么就没有其他进程知道进程1的初始值，所以根据有 184
效性条件，没有进程决定出1。另一方面，没有进程决定出0，因为就其他进程所知，进程1可能在发生故障之前就决定出1，这里的不一致将违反一致性条件。

复杂度分析 *TwoPhaseCommit*算法只使用了两轮。回忆停止一致性问题的轮数下界，在定理6.33中给出的是$f+1$。*TwoPhaseCommit*算法的时间界限与这个下界并不矛盾，因为*TwoPhaseCommit*算法只满足弱终止性条件。通信复杂度就是在任一运行中最坏情况下发送的非空消息数，它是$2n-2$；特别的，这个消息数是在无故障运行中发送的。

7.3.3 三阶段提交

现在我们描述的是提交算法是三阶段提交（*ThreePhaseCommit*算法）；它是对*TwoPhaseCommit*算法的修饰，可以保证强终止性。

这里的关键是，除非所有还没有发生故障的进程都做好了决定出1的“准备”，否则进程1不会决定出1。要确保它们做好了准备，就需要一个额外轮。我们首先描述和分析算法的前三 185
轮，算法的剩余部分需要获得非阻塞属性，这在后面描述。

***ThreePhaseCommit*算法的前三轮：**

第1轮：除了进程1之外，所有进程都向进程1发送它们的初始值，而且任意初始值为0的进程决定出0。进程1把所有收到的值和它自己的初始值放入一个向量。如果这个向量中的所有位置都是1，那么进程1做好了决定的准备，但不做出决定。否则，如果向量中某个位置包含0，或者某个位置没有填充（即没有从对应的进程收到消息），那么进程1决定出0。

第2轮：如果进程1决定了0，那么它广播*decide*(0)。否则，进程1广播*ready*。收到*decide*(0)的任意进程决定出0。收到*ready*的任意进程变为*ready*。如果进程1还没有决定，它就决定出1。

第3轮：如果进程1决定出1，它就广播*decide*(1)。收到*decide*(1)的任意进程决定出1。

见图7-5，它给出了一个例子，说明在*ThreePhaseCommit*算法的无故障运行中的通信模式。[⊖]

在给出终止协议之前，我们先分析前三轮之后的情况。我们把每个进程（不管是发生故障）的状态分成4个排斥与穷举类型：

1）*dec*0：其中的进程决定出0。

2）*dec*1：其中的进程决定出1。

3）*ready*：其中的进程没有决定，但是变成了*ready*。

4）*uncertain*：其中的进程没有决定，也没有*ready*。

下面的引理给出了*ThreePhaseCommit*算法的关键属性，它描述了不能共存的特定状态组合。

⊖ 同样，我们轮数的指定不确切地对应通常的三阶段提交协议的阶段的指定。开始时要加一个额外的请求轮，还有一些显式的确认。

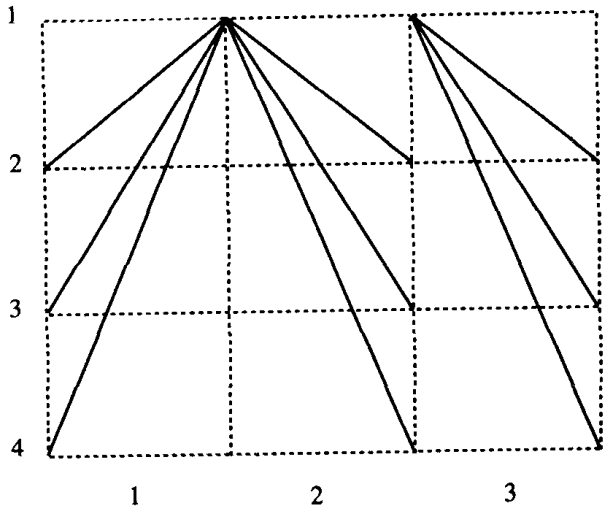


图7-5　*ThreePhaseCommit* 算法中的通信模式

引理7.22　在*ThreePhaseCommit*算法的三轮之后，以下成立：

186　1）如果任一进程的状态是*ready*或*dec*1，那么所有进程的初始值都为1.

2）如果任一进程的状态是*dec*0，那么没有进程的状态是*dec*1，而且没有无故障进程的状态是*ready*。

3）如果任一进程的状态是*dec*1，那么没有进程的状态是*dec*0，而且没有无故障进程的状态是*uncertain*。

证明　证明很直接。最有意思的部分是第三个条件的证明。我们注意到在第二轮的最后，进程1在广播了*ready*消息之后只能决定出1。这意味着在第二轮最后，进程1知道其他进程或者接收并处理了*ready*，然后进入*ready*状态或者已经发生故障了。（这里模型的同步性很重要。）　□

现在我们可以证明，在前三轮之后，大部分条件仍然成立。

引理7.23　在*ThreePhaseCommit*算法的三轮之后，有下面的结论成立：

1）一致性条件成立。

2）有效性条件成立。

187　3）如果进程1没有发生故障，那么所有无故障进程已经做出了决定。

证明　一致性条件可以从引理7.22得到，同样，有效性条件的一半——如果某个进程从0开始，那么0是惟一可能的决定值——也可以从这个引理得到。有效性条件的另一半可以通过检验得到。

最后，如果进程1没有发生故障，那么所有无故障进程都已经做出了决定。这是因为：进程1的决定不能被其他进程的任何行为阻止，而且一旦进程1作出了决定，它立即向其他进程广播它的决定。其他进程以同样的方法作出决定。　□

然而，因为这三轮不能保证强终止性，所以它们不足以解决非阻塞提交问题。正如引理7.32所说，如果进程1不发生故障，那么每个无故障进程都会做出决定。但是，如果进程1失败了，那么其他进程可能处于不确定的状态。考虑到这种情况，其他进程必须在前三轮之后执行一个终止协议。其细节可能有所不同；下面我们描述了一个可能性。

***ThreePhaseCommit*算法，终止协议：**

第4轮：所有（还未发生故障的）进程向进程2发送它们当前的状态，或者是*dec*0，或者是

*dec*1、*ready*、*uncertain*。进程2把这些状态值和它自己的状态放入一个向量。向量中的位置不需要全部填满，进程2可以忽略那些没有填充的位置。如果向量包含任意*dec*0值并且进程2还没有做出决定，那么进程2就决定出0。如果向量包含任意*dec*1并且进程2还没有做出决定，那么进程2决定出1。如果向量中填充的位置包含的值都是*uncertain*，那么进程2决定出0。如果向量中的值只有*uncertain*和*ready*，并且至少有一个*ready*，那么进程2变成*ready*，但不做出决定。

第5轮：在这一轮和下一轮，进程2的行为和进程1在第二轮和第三轮中的行为很相似。如果进程2已经做出了决定，那么它在一个*decide*消息中广播它的决定。否则，进程2广播*ready*。任意没有决定的进程收到*decide*(0)或*decide*(1)之后，决定出0或1。任意接收到*ready*的进程变成*ready*。如果进程2还没有决定，那么它决定出1。

第6轮：如果进程2已经决定出1，它就广播*decide*(1)。任意进程收到*decide*(1)之后，如果它还没有决定，就决定出1。

第6轮之后，协议继续，进程3，...，*n*中的每一个执行三个相似的轮。

定理7.24 包含了终止协议的完整的*ThreePhaseCommit*算法，是一个非阻塞提交算法。 188

证明概要 我们首先证明，在完整的*ThreePhaseCommit*算法中，不仅在前三轮，而且在任意轮数之后，引理7.22中列出的三个性质总是成立的。这可以通过对轮数的归纳来证明。

然后和以前一样，一致性条件可以从扩展的引理7.22得到，同样，有效性条件的一半——如果某个进程从0开始，那么0是惟一可能的决定值——也可以从这个引理7.22得到。有效性条件的另一半也是正确的，因为如果没有故障发生，所有进程都会在前三轮中做出决定。

我们来讨论强终止性。如果所有进程都发生故障，那么这个性质是正确的。否则，假设*i*是一个无故障进程，那么当*i*作为一个协调者的时候，每个无故障进程都做出了决定。 □

复杂度分析 在这里给出的*ThreePhaseCommit*算法中，需要的轮数为3*n*。即使我们允许所有进程发生故障，相对于第6章学习的停止一致性算法，这里的轮数仍然高于那里近似为*n*轮的界限。当然，停止一致性算法导致了一个不同的有效性条件，但是我们可以稍加修改得到提交有效性条件。那么在实际中，为什么像*ThreePhaseCommit*这样的算法被认为更好呢？

主要的原因是：在没有故障的情况下，*ThreePhaseCommit*算法可以改为复杂度很低的算法。如果没有进程发生故障，那么所有的进程在第三轮之前做出决定。那么就有可能增加一个简单的协议，使得进程可以探测到每一个进程做出了决定，然后就不再继续参与以后的终止协议。在这样的修改之后，整个算法只需要一个小常数的轮数和$O(n)$的消息。

7.3.4 消息数的下界

在本章（和第一部分）的最后，我们考虑解决提交问题所需发送的消息数。回想在无故障的情况下，*TwoPhaseCommit*算法使用了$2n-2$条消息。*ThreePhaseCommit*算法使用的消息多一些，但是如果算法被修改成提前停止，那么复杂度仍然是$O(n)$。在本节中，我们将证明，即 189
使是一个阻塞算法，在无故障的情况下，也不可能有比$2n-2$好的结果。

定理7.25 任意解决提交一致性问题的算法，即使有弱终止性，在所有输入为1的无故障运行中，也要至少使用$2n-2$条消息。

在本节的剩余部分，我们固定一个特定的提交算法*A*，并令α_1为*A*的一个无故障运行，其中所有的输入都是1。我们的目的是证明α_1至少包含$2n-2$条消息。

我们还要用到6.7节中的通信模式的定义。这里，我们用通信模式描述无故障运行中发送消息的集合。（与过去不同，我们并不假设在每一轮所有的进程向其他所有的进程发送消息。）

对A的任意无故障运行α，我们从中显式地提取出通信模式$patt(\alpha)$。

对一个通信模式r，我们还使用了5.2.2节给出的排序$\leq_r$的定义，用来获取不同时刻不同进程之间的信息流。如果对某个k，有$(i,0) \leq_r (j,k)$，我们就说在通信模式r中进程i影响进程j。下面的引理叙述了关于下界的关键思想。

引理7.26　对每两个进程i和j，在$patt(\alpha_1)$中，i影响j。

例 7.3.1　提交的下界

在证明引理7.26之前，我们给一个例子来说明为什么它是正确的。假设α_1(所有输入为1的A的无故障运行)恰好包含图7-6中左边描绘的消息。

根据有效性和弱终止性，α_1中的所有进程最终必然决定出1。注意到在$patt(\alpha_1)$中，进程4不影响进程1；让我们看一下结果将会产生什么问题。考虑另一个运行α_1'，它与α_1相同，但进程4的输入是0，而且每个进程在第一次受到进程4的影响之后发生故障。运行α_1在图7-6的右边描绘出来了，故障用Xs表示。可以直观的看出$\alpha_1 \sim^{1} \alpha_1'$，这意味着在$\alpha_1'$中进程1也决定出1。但是这违反了$\alpha_1'$的有效性条件，导致了矛盾。

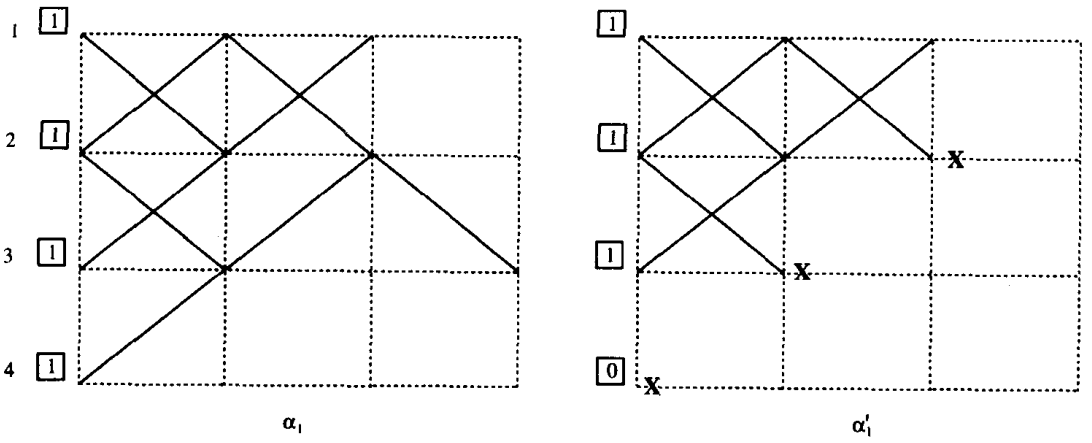


图7-6　α_1和α'_1中发送的消息

引理7.26的证明使用了和例7.3.1中相同的证明方法。

190

证明　根据有效性和弱终止性条件，α_1中所有进程最终必然都决定出1。假设引理是错误的，固定两个进程i和j，使得在$patt(\alpha_1)$中i不影响j。根据定义必然有$i \neq j$。构造α_1'，其中进程i的输入变为0，每个进程在第一次受到进程i的影响之后发生故障。那么$\alpha_1 \sim^{j} \alpha_1'$，所以进程$j$在$\alpha_1'$中也决定出1。这违反了有效性条件，矛盾。　□

为了完成定理7.25的证明，我们必须说明，每个进程影响其他每个进程的要求，意味着至少要有总共$2n-2$条消息。我们使用一个关于通信模式的引理。

引理7.27　令γ为任一通信模式。如果在γ中，一个$m \geq 1$个进程的集合中的每一个进程，对系统中n个进程中的每一个都产生影响，那么在γ中至少有$n+m-2$条消息（三元组）。

证明　通过对m归纳。

基础：$m=1$。令i为我们假设的进程，它影响n个进程中的每一个。由于i影响所有的n个进程，γ中必然包含发送给除了i之外的$n-1$个进程的消息。总消息数至少为$n-1$。

归纳步：假设引理对m成立，我们证明$m+1$的情况。令I为$m+1$个进程的集合，其中的每个

进程影响γ中所有n个进程。不失一般性，我们可以假设在第一轮中，I中至少有一个进程向某个进程发送消息。因为如果不是这样的话，我们就能去掉那些I中没有进程发送消息的开始轮；在剩余的通信模式中，I中所有进程仍然会影响所有n个进程。令i是I中的某个进程，它在γ中的第一轮发送消息。 191

现在考虑通信模式γ'，它是通过从γ中去掉i在第一轮发送的消息得到的。那么在γ'中，$I-\{i\}$中的所有进程影响所有n个进程。根据归纳假设，在γ'中至少有$n+m-2$条消息。所以γ至少包含$n+m-1=n+(m+1)-2$条消息。得证。 □

现在我们来完成定理7.25的证明。

（定理7.25）的证明 根据引理7.26，对于每两个进程i和j，在$patt(\alpha_1)$中i影响j。然后引理7.27意味着在$patt(\alpha_1)$中至少有$2n-2$条消息。 □

7.4 参考文献注释

k—致性问题通常被称为“k集合一致性问题”。这个问题是在以前研究的基础上作为普通一致性问题的自然扩展，首先由Chaudhuri[73]提出的。*FloodMin*算法来自Chaudhuri、Herlihy、Lynch和Tuttle[75]的著作，它基于一个最初由Chaudhuri[73]设计的算法。对k—致性的下界的讨论来自[75，76，77]。在讨论下界时用到的代数拓扑学的背景知识可以在Spanier的教科书[266]中找到。Sperner引理首先是由Sperner[267]提出的，在[266]中进行了讨论。

近似一致性的问题源于Dolev、Lynch、Pinter、Stark和Weihl[98]的一篇论文。关于这个问题的其他工作是Fekete[110,111]，以及Attiya、Lynch和Shavit[24]完成的。关于提交问题的资料，包括*TwoPhaseCommit*和*ThreePhaseCommit*算法，来自Bernstein、Hadzilacos和Goodman[50]的一本关于数据库理论的书。这本书讲的比这里更加深入，它讨论了对协议的实际实现中的问题，包括怎样恢复故障进程。提交问题的消息数的下界来自Dwork和Skeen[106]的著作。

7.5 习题

7.1 如果k—致性问题的*FloodMin*算法只执行$\left\lfloor \frac{f}{k} \right\rfloor$轮，而不是$\left\lfloor \frac{f}{k} \right\rfloor+1$轮，那么无故障进程得到的不同决定数最多是多少？ 192

7.2 在定理7.14的证明中，给出序列$seq(v)$的长度的上界。为此，你需要明确描述序列的构造。

7.3 证明一系列1运动的合并实际上是一个k运动，这需要说明它满足运动的定义中要求的条件，以及有关令牌的条件。

7.4 证明引理7.4。

7.5 证明引理7.5。

7.6 证明引理7.6。

7.7 证明引理7.7。

7.8 令$n=5$，$k=f=2$，$r=1$。

a）对于这些参数值，详细描述Bermuda三角形，以及它的k运动和进程索引标注。

b）考虑一个算法A，它如下工作：所有进程都进行交换一次值，每个进程选择它收到的最小值。描述Sperner着色C_A。

c）对算法A，你能否找出一个特定的小单纯形，其中有三个不同的决定值。

7.9 固定任意n和f，其中$n>3f$，任意ε，任意$w\in R^{\geq 0}$，以及任意$r\in \mathbb{N}$。对n，f和ε，描述有终止性的*ConvergeApproxAgreement*算法的一个特定运行，其中无故障进程的初始值多元素集合的

宽度最多为w，终止需要的轮数大于r。

7.10 研究问题：修改*ConvergeApproxAgreement*算法，使得所有进程做出决定的时间受一个n的函数、f、ε和无故障进程初始值多元素集合的宽度w的限制。

7.11 假设在*ConvergeApproxAgreement*中，进程不是计算*mean*(*select*(*reduce*(*W*)))，而是计算以下之一：

a）*mean*(*select*(*W*))

b）*mean*(*reduce*(*W*))

193 c）*mean*(*W*)

算法仍然能够解决近似一致性问题吗？为什么能或不能？

7.12 证明容许f个Byzantine故障的近似一致性问题能够在网络图G中解决，当且仅当以下条件成立：

a）$n>3f$

b）$conn(G)>2f$

7.13 为停止故障情况设计一个近似一致性算法。

a）相对于故障数，所需的进程数尽量最小化。

b）所需轮数尽量最小化。

7.14 构造近似一致性问题的一个变形，它使用固定的轮数r，且ε不是预先确定的。和以前一样，每个进程从一个实数值开始。在r轮后，进程要输出最终值。有效性条件和以前一样。这里的目的是保证最大可能的一致性，即无故障进程最终值的宽度与无故障进程初始值宽度的比率的上界。

a）在这样的环境中，*ConvergeApproxAgreement*算法能得到什么样的比率？

b）证明可以得到的比率的下界，用n，f，r表示。（提示：与定理6.33的证明相似，用链证明的思想。你得到的上界和下界可能并不匹配。）

7.15 写出完整的（包括终止协议的）*ThreePhaseCommit*算法的代码。

7.16 详细证明引理7.22可以扩展到*ThreePhaseCommit*的任意轮数的情况。

7.17 修改*ThreePhaseCommit*算法，以在无故障的情况下进程做出决定并很快停止。详细描述这种修改。在无故障的情况下，你的算法使用的轮数应该是一个小常数，消息数应该为$O(n)$。

194 证明其正确性。

7.18 按第6章中停止一致性算法的样子设计一个算法来解决有强终止性的提交问题。尽量减小轮数。

7.19 研究问题：设计一个算法来解决有强终止性的提交问题。同时，你能否得到一个最坏情况下的轮数$n+k$（对于某个常数k）？你能否得到在无故障的情况下做出决定和停止所需的小常数轮数？你能否在无故障的情况下得到一个较低的通信复杂度？

7.20 给出引理7.26的详细证明。如果我们构造α'_1时不强迫任何进程发生故障，而只是把进程i的初始值从1变为0，这样的话证明将在哪一步失败？

7.21 设计一个非阻塞提交算法，对于无故障运动，它使用尽可能少的消息。你能证明这个消息

195 ~ 196 数是最优的吗？

第二部分 异步算法

第二部分包括第8~22章，是该书的主要部分。这些章的计算对象从第2~7章中讨论的锁步同步模型转变为异步模型。在异步模型中，系统组件以任意速度运行。

与同步模型一样，异步模型不难描述。该模型与前者之间的细微差别主要在于它涉及到活性条件，例如，该模型要求每一组件不断地得到机会来运行。然而，由于事件发生顺序的额外不确定性，该模型比同步模型更难编程。与典型分布式系统所能提供的相比，异步模型对时间的要求更低。因而，为异步模型所设计的算法是通用的、可移植的，能够保证在具有任意时序行为的网络中正确运行。

第二部分的第1章，即第8章，提出了一个异步系统的通用模型——输入/输出自动机模型。你可先快速浏览此章，然后在必要时查阅具体内容。其余各章再分为两部分：一部分是第9~13章，覆盖异步共享存储器算法；另一部分是第14~22章，覆盖异步网络算法。

第8章 建模 II: 异步系统模型

本章的目的是介绍一种异步计算的正式模型：输入/输出（I/O）自动机模型。这是一种非常通用的模型，适合于描述几乎任何类型的异步并发系统，其中包括将要在本书中学习的异步共享存储器系统和异步网络系统这两类系统。I/O自动机模型本身的结构极为简单，可用于表示多种不同类型的分布式系统。该模型在描述特殊类型的异步系统时，必须在基础模型之上补充附加结构。该模型本身则提供一种用以精确描述和推理那些彼此交互的、以任意相对速度操作的系统组件（如进程通道或通信通道）的方法。

我们首先给出I/O自动机及其运行的定义，然后给出合成操作的定义，该操作能将多个I/O自动机合成为一个代表并发系统的较大型自动机。我们展示合成操作拥有其应有的优良属性。再次，我们引入“公平性”这一重要概念，规定系统中所有的组件都有“公平”的机会来频繁地执行操作。公平性代表着对系统组件的任意相对速度的一个限制——它排除了某些组件永远无法执行操作的可能性。我们将展示“公平性”与合成操作是如何交互的。本章其余内容则描述了一些通俗约定来阐明I/O自动机描述的系统所要解决的问题，以及一些用以证明该系统具有解决这些问
199 题的能力的方法。

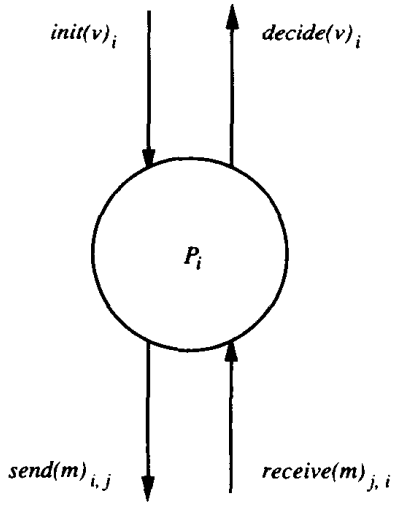


图8-1 一个进程I/O自动机

本章旨在为模拟异步系统的方法提供参考，这些方法不仅适用于本书中提到的异步系统，也适用于许多其他系统。从这一点来说，你并不需要仔细阅读本章。我们建议你先阅读后面关于算法的那些章，如第10、11、12、15章，然后在必要时返回本章（以及第9、14章）以获取形式化基础知识。

8.1 输入/输出自动机

输入/输出（I/O）自动机用来表示一个与其他系统组件相交互的分布式系统组件。它是一种简单的状态机，其中事务与带名字的动作相关。动作可以分成输入动作、输出动作和内部动作三种。其中，输入动作和输出动作用于自动机与外部环境的交流，而内部动作则只对自动机本身可见。输入动作来自外部，不受自动机控制；而输出动作和内部动作则由自动机本身规定。

一个典型的I/O自动机例子是异步分布式系统中的一个进程。图8-1显示了一个典型的进程自动机与其环境之间的接口。图中，自动机P_i被画成一个圆，在指向圆的箭头旁边标有“输入动作”字样，而在离开圆方向的箭头旁边标有“输出动作”字样。内部动作则未在图中标明。
200 自动机从外界接收形如$init(v)_i$的输入，代表着对输入值v的接收；自动机具有形式为$decide\ (v)_i$的输出，用来表示对v的决定。为了取得对v的决定，P_i可能需要使用消息系统来与其他程序通信。它与消息系统之间的界面包括形如$send(m)_{i,j}$的输出动作和形如$receive(m)_{j,i}$的输入动作，进程P_i使用前者来将内容为m的消息发送给进程P_j，使用后者来接收进程P_j发来的内容为m的消息。当自动机执行任何所示的动作（或任何内部动作）时，其状态有可能随之改变。

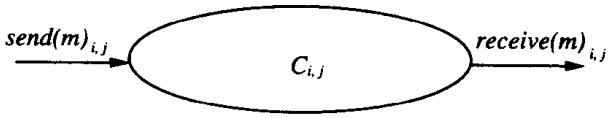


图8-2 一个通道自动机

另外一个典型的I/O自动机例子是一个FIFO消息通道。图8-2显示了一个名为$C_{i,j}$的典型通道自动机，其输入动作的形式为$send(m)_{i,j}$，输出动作的形式为$receive(m)_{i,j}$。通常，在使用I/O自动机来描述分布式系统时，系统包括一系列的进程自动机和通道自动机，其中一个自动机的输出与另一个自动机的同名输入相匹配。由此，进程P_i执行的$send_{i,j}$输出动作与通道$C_{i,j}$执行的$send_{i,j}$输出动作相一致（亦即同时执行）。值得注意的是，不同的动作以不可预知的顺序执行，且每次只执行一个动作。这与同步系统相悖，因为在同步系统中，所有的程序在每一轮的运算中都同时发送消息，随后又同时接收消息。

正式的，I/O自动机中首先被详述的元素是它的“签名”，即对它的输入、输出和内部动作的描述。我们假定存在一个动作全集。签名S是一个三元组，由三个互不相交的动作集合组成：输入动作集合$in(S)$、输出动作集合$out(S)$和内部动作集合$int(S)$。外部动作集合$ext(S)$被定义为$in(S) \cup out(S)$；局部控制动作集合$local(S)$被定义为$out(S) \cup int(S)$；而$acts(S)$则包括S中所有的动作。外部签名$extsig(S)$被定义成为签名$(in(S), out(S), \phi)$。我们通常把外部签名称为外部接口（*external interface*）。

一个I/O自动机A，简称为自动机，由五部分组成：

- $sig(A)$，一个签名。
- $states(A)$，一个状态集合（不一定是有限的）。
- $start(A)$，$states(A)$的一个非空子集，被称为起始状态或初始状态。
- $trans(A)$，状态转移关系，这里 $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$；对于任意状态$s$及任意输入动作$\pi$，都有$transition(s, \pi, s') \in trans(A)$。
- $task(A)$，一个任务划分，是$local(sig(A))$上的一个等价关系，拥有可数的等价类。

我们将$acts(sig(A))$简写成$acts(A)$，同理将$in(sig(A))$简写成$in(A)$，如此类推。如果A没有输入，即 $in(A) = \phi$，则称它是封闭的。

该定义看似与第2章的同步网络模型中进程自动机的定义相似。然而，这里的签名允许使用更通用的动作类型，而不仅仅是同步系统中的消息发送和消息接收动作。与同步网络模型进程状态集合一样，异步网络模型的进程状态集合不必是有限的。这种通用性非常重要，因为这样的话系统可以具有无界数据结构，如计数器和无界长度队列等。与同步情况的情况一样，我们允许存在多个起始状态，这样起始状态可以包括一些输入信息。

我们把$trans(A)$中的一个元素(s,π,s')称为A的一个转移或转移函数，或是A的一步。根据动作π是否为输入动作、输出动作等，我们将转移(s,π,s')相应地称为输入转移、输出转移等。与同步模型不同，这些转移不一定与一系列消息的接收相关，而是可以与任意动作相关。

如果对于一个特定的状态s与动作π，A的转移的形式为(s,π,s')，则称π在s中是使能的。由于我们要求每种输入动作都必须在每个状态中使能，故称自动机为输入使能的。输入使能的假设意味着自动机不能“阻止”输入动作发生，例如，当消息到达时，进程不得不准备以某种方式来处理任意可能的消息值。当在s中使能的惟一动作是输入动作时，认为状态s是静态的。

你可能会认为这种输入使能的属性给通用模型施加了太多的限制，因为很多系统组件认为特定输入只在特定时间发生。例如，一台表示资源分配系统的自动机（在第11章中提及）认为用户在系统完成第一条请求的回应之前，不会连续地提交第二个请求。然而，我们可以使用其他方式来表达这些施加在环境之上的限制，没有必要去禁止环境执行输入操作。例如，在资源分配的例子中，我们可以认为在接收到第一个请求的应答之前，环境不会提交第二个请求，万

一出现意外输入，我们也不限制系统的行为。或者，我们可以要求系统进行意外输入检测，并在检测到意外输入时返回一个错误消息。

输入使能属性具有两大优点。首先，系统组件开发中的一个严重错误源在于，意外输入发生时组件们往往不知所措。如果模型允许任意输入，则有助于这些错误的消除。其次，允许输入使能使得模型的理论基础更为扎实；尤其是，允许输入使能使得采用简单概念来描述自动机的外部行为更为合理（定理8.4就是在未假定输入使能时出错的一个例子）。

I/O自动机定义的第五个组件，即任务分割*tasks* (*A*), 是对自动机中的“任务（task）”或是“控制线程”的一个抽象描述。此任务分割用于定义自动机运行中任务的公平性条件，即在自动机执行任务时，自动机必须能够一直赋予公平的执行给每个任务。这对于一个执行多个任务的系统组件的表示是很有用的。这些组件的例子包括：某组件在参与一个持续运行的算法的同时又周期性地向其环境报告状态信息。在几个自动机组合成一个大自动机以代表整个系统的情况下，任务分割也是有用的，它能够表明参与合成的每个自动机都可以继续在合成系统中运行。任务分割的另一用途是用来表示异步共享存储器算法（见第9章）。通常我们将任务分割类称为任务。

如果任务C中的某些动作在状态*s*中使能，则称任务*C*某些动作在状态*s*中使能，这是一种简便说法。

下面是一个简单I/O自动机的例子。在大部分对I/O自动机的描述中，包括在本例中，我们都以前提-结果的形式来表示转移关系。与每种特定动作类型相关的所有转移都被合成一组，每组以一段代码来表示。代码使用前置状态谓词*s*表达式来表明允许动作发生的条件，随后，它以一个简单程序的形式来描述动作所导致的变化；此程序被应用在*s*上来产生*s'*。整段代码作
203 为一个单独的、不可分割的转移来执行。根据动作类型来对转移分组有助于代码的简化，因为与每种动作相关的转移只涉及一小部分状态。

以前提-结果形式写成的程序通常只使用非常简单的控制结构，这使得从程序到自动机的翻译相当透明，且对自动机的正式推理将变得更为容易。

例8.1.1　通道I/O自动机

作为一个I/O自动机的例子，考虑一个通信通道自动机C_{ij}。令M是一个固定的消息字母集。首先给出签名*sig* ($C_{i,j}$)。现在约定：如果不提及某种签名组件（通常是内部动作），则该动作集合是空集。

签名：

输入：　　输出：

$send(m)_{ij}$, $m \in M$　　$receive(m)_{ij}$, $m \in M$

状态*states*(C_{ij})与开始状态*start*(C_{ij})通常以状态变量及其初始值的列表来描述，这与同步环境中的情况是一样的。

状态：

队列*queue*，一个M中元素组成的FIFO队列，初始队列是空队列

C_{ij}的转移函数的代码如下：

转移：

$send(m)_{ij}$

结果：

将m加入队列中

$receive(m)_{ij}$

前提：

m处于队列头

结果：

删除队列的第一个元素

代码的意思很明显：动作*send*可以在任何时间发生，执行结果是将消息添加到队列的尾部；而动作*receive*只有在当前消息*m*处于队列的头部时才能发生，其执行结果是从队列删除该消息。

任务划分$task(C_{i,j})$将所有*receive*动作归成一个单独任务，亦即所有接收（传送）信息的工作被视为一个单独任务。

任务：

$\{receive(m)_{i,j} : m \in M\}$

204

例8.1.2 进程I/O自动机

I/O自动机的第二个例子是一个进程自动机P_i。该自动机的外部接口如下，其中*V*是一个固定的值集合，*null*是一个不在*V*中的特殊值，而*f*是一个固定函数，$f: V^n \to V$。

签名：

输入：	输出：
$init(v)_i, v \in V$	$decide(v)_i, v \in V$
$receive(v)_{j,i}, v \in V, 1 \leqslant j \leqslant n, j \neq i$	$send(v)_{i,j}, v \in V, 1 \leqslant j \leqslant n, j \neq i$

状态与开始状态如下：

状态：

val，一个下标在$\{1, ..., n\}$中、值在$V \cup \{null\}$中的向量，初值为*null*

转移函数如下：

转移：

$init(v)_i, v \in V$

结果：

$val(i) := v$

$send(v)_{i,j}, v \in V$

前提：

$val(i) = v$

结果：

none

$receive(v)_{j,i}, v \in V$

结果：

$val(j) := v$

$decide(v)_i, v \in V$

前提：

对于所有的$j, 1 \leqslant j \leqslant n$:

$val(j) \neq null$

$v = f(val(1), \cdots, val(n))$

结果：

none

P_i使用*init*动作来在向量*val*中自己所属位置填入指定值，*receive*动作则用于对其他位置填值，这些值可以通过多个*init*动作或*receive*动作来更新任意多次。P_i可以任意次地将值发送到任意通道中去，还可以任意次地根据*f*在向量上的应用来做决定。

任务划分$task(P_i)$包含*n*个任务：所有$j \neq i$的$send_{i,j}$动作都被归为一个任务，另一个任务由所有的*decide*动作组成。因此，所有在通道上发送消息的动作被认为是一个单独任务，而所有报告决定的动作也被认为是一个单独任务。

任务：

对每个$j \neq i$:

$\{send(v)_{i,j} : v \in V\}$

$\{decide(v)_i : v \in V\}$

现在来描述I/O自动机*A*是如何运行的。*A*的一个运行片断是一个*A*中状态和动作交替出现

205

的有限序列$s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$，或者是一个无限序列$s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r, \ldots$，这里$(s_k, \pi_{k+1}, s_{k+1})$是$A$中的一个转移，$k \geq 0$。注意当一个序列是有限的时候，序列的结尾必为状态。一个从初始状态开始的运行片断称为运行。A的运行集合以$execs(A)$来表示。如果A中的一个状态是A的一个有限运行的最终状态，则称该状态是可到达的。

如果α是A的一个有限运行片断，α'是A的另一个有限运行片断，α'以α的最终状态作为开始状态，则$\alpha \cdot \alpha'$代表由α和α'连接而成的序列，其中α的最终状态因重复出现而被删除。显然，$\alpha \cdot \alpha'$也是A的一个运行片断。

有时，我们只对I/O自动机的外部行为感兴趣。因此，A的运行α的轨迹，用$trace(\alpha)$来表示，它是α随后的包含所有外部动作。如果ρ是A的运行轨迹，则称ρ是A的轨迹。我们用$traces(A)$表示轨迹集合。

例8.1.3　运行

以下是例8.1.1中的自动机$C_{i,j}$的三种运行（假设信息字母表M为集合{1,2}）。方括号里面的是队列的状态序列，λ表示空序列。

$$[\lambda], send(1)_{i,j}, [1], receive(1)_{i,j}, [\lambda], send(2)_{i,j}, [2], receive(2)_{i,j}, [\lambda]$$

$$[\lambda], send(1)_{i,j}, [1], receive(1)_{i,j}, [\lambda], send(2)_{i,j}, [2]$$

$$[\lambda], send(1)_{i,j}, [1], send(1)_{i,j}, [11], send(1)_{i,j}, [111], \ldots$$

后两个是合法的，即使它们包含已被发送但未被接收的消息，因为我们没有规定使能的动作必定在运行中发生。为了表达这种规定，我们将在8.3节中介绍公平性要求。

8.2　自动机的操作

206　本节定义I/O自动机的合成操作和隐藏操作的输出动作。

8.2.1　合成

合成操作允许一个自动机表示由多个自动机组成的复合系统，其中每个自动机分别代表各个系统组件。合成操作标识出不同组件自动机中具有相同名字的动作。当任意一个组件自动机执行与π相关的一步时，签名中具有π的所有组件自动机也执行该步。

我们对要合成的自动机加以一定的限制。首先，由于自动机A的内部动作不愿被其他自动机所见。除非A的内部动作与B的动作互不相交，否则我们不允许A与B合成（否则A中内部动作的执行会迫使B执行一步）。其次，为了让合成操作具有良好的性质（如以后的定理8.4所示），我们约定：给定一个动作，最多一个组件自动机能够“控制”这个动作的性能，即只有A和B的输出动作不相交时，才能合成A和B。再次，我们并不排除对可数无限自动机集合进行合成的可能性，但在这种情况下我们要求每个动作都最多属于有限多个组件自动机，否则定理8.3将不成立。

为什么我们不简单地禁止无限多个自动机的合成呢？毕竟物理计算机系统是由有限多个组件（计算机、消息通道等）组成的。原因在于I/O自动机既用来模拟*逻辑系统*，又用来模拟*物理系统*。一个逻辑系统可以包含大量的逻辑组件，并在拥有更少组件的物理系统上实现。事实上，一些逻辑系统允许在运行过程中动态地创建组件，使得在一个无限运行的轨迹上可能有无限多个组件（例如，数据库系统允许在系统运行时创建新的事务实例）。我们使用I/O自动机来表示模型组件，这种方法认为：所有可能被创建的组件事实上一开始便存在，只不过在组件需

要被建立的时候系统才使用特定的唤醒输入动作来唤醒它们。通过这种技巧，一般的合成操作符便足以描述动态建立的组件与系统其他部分进行交互的方法。

正式地，对于所有的$i, j \in I, i \neq j$，如果下列所有条件成立，则称可数签名集合$\{S_i\}_{i \in I}$是兼容的：

1）$int(S_i) \cap acts(S_j) = \phi$。

2）$out(S_i) \cap out(S_j) = \phi$。

3）没有动作被包含在无限多个$acts(S_i)$集合中。

如果一个自动机集合中的相应自动机签名集合是兼容的，则该自动机集合是兼容的。

当我们合成一个自动机集合时，原来组件的输出动作变成合成自动机的输出动作，原来组件的内部动作变成合成自动机的内部动作，而那些既非任一组件的输出动作又是某些组件的输入动作的动作则变成合成自动机的输入动作。正式地，可数兼容签名集合$\{S_i\}_{i \in I}$的合成$S = \prod_{i \in I} S_i$是具有以下组件的签名：

- $out(S) = \cup_{i \in I} out(S_i)$
- $int(S) = \cup_{i \in I} int(S_i)$
- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$

现在，可数兼容自动机集合$\{A_i\}_{i \in I}$的合成$A = \prod_{i \in I} A_i$如下[⊖]：

- $sig(A) = \prod_{i \in I} sig(A_i)$
- $states(A) = \prod_{i \in I} states(A_i)$
- $start(A) = \prod_{i \in I} start(A_i)$
- $trans(A)$是三元组(s, π, s')的集合，其中对于所有$i \in I$，如果$\pi \in acts(A_i)$，则$(s_i, \pi, s'_i) \in trans(A_i)$；否则$s_i = s'_i$
- $tasks(A) = \cup_{i \in I} tasks(A_i)$

因此，合成自动机的状态是各个组件自动机的状态上的笛卡儿积（也称为向量），合成自动机的开始状态是各个组件自动机的开始状态的上的笛卡儿积（也称为向量）。合成的转移函数为：所有签名中具有特定动作π的组件自动机同时执行涉及π的步，而其他自动机不参与执行。合成自动机中局部控制动作的任务划分是各组件的任务划分的并集，即每个组件自动机的每个等价类都是合成自动机的一个等价类，这意味着当组件被合成时各个组件的任务结构被保持。注意自动机A_i是输入使能的，其合成也是输入使能的。因此$\prod_{i \in I} A_i$也是一个I/O自动机。

当I是一个有限集合时，可用中缀操作符$\times$来表示合成。例如，若$I = \{1, \ldots, n\}$，可用$A_1 \times \ldots \times A_n$来表示$\prod_{i \in I} A_i$。

注意，当一个动作π是一个组件的输出和另一个组件的输入时，在合成中它被归为输出动作而不是内部动作。原因在于π被允许用作进一步的通信。例如，假设π在自动机A中是输出动作，而在自动机B和C中是输入动作。则在三个自动机的合成$A \times B \times C$中，π是一个从A到B和C的广播动作。我们以模块化的方式来考察这次合成，首先构造$A \times B$，然后将所得结果与C合成。根据合成的定义，$A \times B \times C$与$(A \times B) \times C$同构，后者是先合成$A \times B$，再与C合成。如果π在$A \times B$中被归为内部动作，则模块性不复存在：合成$A \times B$不能够与C合成，因为这会违反第一个兼容性条件。

用于组件间通信的动作可以被“隐藏”，从而避免它们被用于以后的通信。这可以通过合

⊖ 在$start(A)$和$states(A)$的定义中的记号$\prod$指通常的笛卡儿积，而在$sig(A)$定义中的记号$\prod$指以上定义的符号合成操作。另外，这里使用记号s_i来代表状态向量s的第i个元素。

成操作和8.2.2节中的定义隐藏操作来实现。

例8.2.1　自动机的合成

设有一个固定的编号集合$I = \{1, ..., n\}$，令A是例8.1.2中所有进程自动机P_i, $i \in I$, 和例8.1.1中所有通道自动机$C_{i,j}$, $i, j \in I$的合成。为了合成它们，我们必须假定通道自动机的消息字母表M包含进程自动机的值集合V。图8-3显示了$n = 3$时合成的“体系结构”。

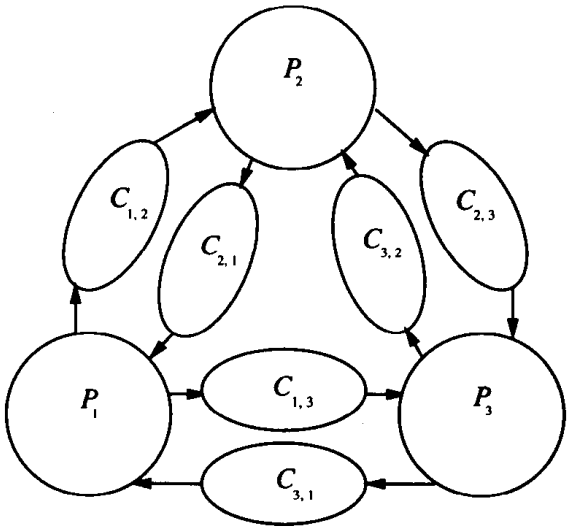


图8-3　P_i和$C_{i,j}$的合成

由此生成的合成是一个表示分布式系统的单个自动机。系统的状态由每个进程的所有状态（所有值向量）和每个通道的所有状态（所有处于转换之中的消息队列）组成。系统的每个转
209 移涉及以下动作中的一个：

1）一个$init\ (v)_i$输入动作，将一个值存放在P_i的变量$val(i)$中，即$val(i)_i$[⊖]。

2）一个$send(v)_{i,j}$输出动作，将P_i的值$val(i)_i$送到通道$C_{i,j}$中。

3）一个$receive(v)_{i,j}$输出动作，将$C_{i,j}$的第一个消息移除，同时将消息值存放在P_j的变量$val(i)_j$中。

4）一个$decide(v)_i$输出动作，被P_i用来宣布当前的计算值。

该合成的一个简单轨迹如下，其中$n = 2$，值集合V为$\mathbb{N}$，而f是加法：

$init(2)_1, init(1)_2, send(2)_{1,2}, receive(2)_{1,2}, send(1)_{2,1},$
$receive(1)_{2,1}, init(4)_1, init(0)_2, decide(5)_1, decide(2)_2$

在此轨迹到达的惟一状态中，P_1拥有val向量(4,1)，P_2拥有val向量(2, 0)，而两个通道都是
210 空的。当然，该合成系统的运行还可以生成很多其他轨迹。

本节的结尾包括三个与组件自动机合成的运行和轨迹相关的基本结论。第一个结论指出我们可以通过“投影”合成的一个运行或轨迹来产生组件自动机的多个运行或轨迹。给出一个A的运行$\alpha = s_0, \pi_1, s_1, ...$，令$\alpha | A_i$是通过删除所有$\pi_r, s_r$对（这里$\pi_r$不是$A_i$的一个动作）并以$(s_r)_i$（即自动机$A_i$的状态$s_r$）替代每个剩下的$s_r$而得的序列。给出$A$的一个轨迹$\beta$（或者更通用的，给出任意动作序列），令$\beta | A_i$是由$\beta$中$A_i$的所有动作组成的$\beta$的子序列。记号“|”用来代表序列$\beta$的一个子序列，该子序列由$\beta$中给定集合中的所有动作组成。

⊖　与同步模型中一样，变量下标是变量所在进程的索引。

定理8.1 令$\{A_i\}_{i\in I}$是一个兼容自动机集合，令$A=\prod_{i\in I}A_i$

1. 如果$\alpha\in execs(A)$，则对于任意$i\in I$，$\alpha|A_i\in execs(A_i)$
2. 如果$\alpha\in traces(A)$，则对于任意$i\in I$，$\beta|A_i\in traces(A_i)$

证明 此证明留做一道习题。 □

另外两个定理是定理8.1的逆定理。第二个定理指出：在特定条件下，组件自动机的运行能够被“粘贴”成为合成的一个运行。

定理8.2 令$\{A_i\}_{i\in I}$是一个兼容自动机集合，令$A=\prod_{i\in I}A_i$。对于任意$i\in I$，设α_i是A_i的一个运行，β是一个由$ext(A)$中动作组成的序列，使得对于任意$i\in I$，$\beta|A_i=trace(\alpha_i)$成立，则存在一个$A$的运行$\alpha$，使得对于任意$i\in I$，$\beta=trace(\alpha)$且$\alpha_i=\alpha|A_i$成立。

证明 此证明留做一道习题。 □

定理8.3 令$\{A_i\}_{i\in I}$是一个兼容自动机集合，令$A=\prod_{i\in I}A_i$。设β是一个由$ext(A)$中动作组成的序列。如果对于任意$i\in I$，有$\beta|A_i\in traces(A_i)$，则$\beta\in traces(A)$。

证明 此证明留做一道习题。 □

定理8.3意味着：为了显示一个序列是系统的一个轨迹，只要说明其在每个单独的系统组件上的投影是该组件的一个轨迹就行了。 211

8.2.2 隐藏

现在我们定义一种操作来“隐藏”I/O自动机的输出动作，这通过将输出操作归为内部动作来实现。该操作防止这些输出动作被用来进行进一步的通信，即它们不能在以后的轨迹中出现。

我们来定义签名的隐藏操作：如果S是一个签名且$\Phi\subseteq out(S)$，则$hide_\Phi(S)$是一个新签名S'，$in(S')=in(S)$, $out(S')=out(S)-\Phi$, $int(S')=int(S)\cup\Phi$。

I/O自动机的隐藏操作很容易定义：如果A是一个自动机且$\Phi\subseteq out(A)$，则$hide_\Phi(A)$是一个以$sig(A')=hide_\Phi(sig(A))$代替$sig(A)$而得的自动机$A'$。

8.3 公平性

在分布式系统中，我们通常仅对其中所有组件都得到公平执行机会的合成的运行感兴趣。在本节中，我们定义了I/O自动机的公平性概念。

如前所述，每个I/O自动机都有一个对局部控制动作的划分，划分中的每个等价类代表了自动机准备执行的一些任务。这里的公平性概念指的是让每个任务都得到无限多的机会来执行一个动作。

正式地，倘若$tasks(A)$中的每个类C都满足以下条件，则I/O自动机A的运行片断α是公平的。

1）如果α是有限的，则C在α的最终状态中不是使能的。

2）如果α是无限的，则α要么包含无限多的C中事件；要么包含某些状态的无限次发生，C在这些状态中是不使能的。

自此，序列（如运行或者轨迹）中一个动作的发生以术语事件（*event*）来表示。

公平性的定义易于明白：每个任务（即等价任务）C可以得到无限多的回合。当一个回合到来时，C或者执行一个动作，或者由于没有使能的动作而不执行任何动作。如果在一个运行中，自动机反复地以轮换的顺序来给所有任务予执行机会，但由于没有使能的动作存在而导致没有动作被成功执行时，它是一个公平的运行。

A的公平运行的集合记为$fairexecs(A)$。如果β是A的一个公平运行的轨迹，则称β为A的一个
212 公平轨迹；A的公平轨迹的集合记为$fairtraces(A)$。

例 8.3.1　公平性

在例8.1.3中，第一个运行是公平的，因为在结束状态中没有使能的*receive*动作。第二个不是公平的，因为它是有限的，且在结束状态中有一个*receive*动作是使能的。第三个也不是公平的，因为它是无限的且不包含*receive*事件，但在第一步之后的每个点上都有一个*receive*动作是使能的。

例 8.3.2　公平性

为了进一步说明公平性的定义，考虑以下的时钟I/O自动机，该自动机代表一个离散时钟。

Clock automaton:

Signature:

Input:
　request

Internal:
　tick

Output:
　$clock(t)$, $t \in \mathbb{N}$

States:

counter $\in \mathbb{N}$, initially 0
flag, a Boolean, initially *false*

Transitions:

tick
　Precondition:
　　true
　Effect:
　　counter := *counter* + 1

request
　Effect:
　　flag := *true*

$clock(t)$
　Precondition:
　　flag = *true*
　　counter = t
　Effect:
　　flag := *false*

Tasks:

{*tick*}
$\{clock(t) : t \in \mathbb{N}\}$

这个时钟自动机简单地一直“跳动”，不断增加计数器的值。当一个需求到达时，*Clock*
213 （在独立的一步中）返回计数器的当前值。以下是*Clock*的一次公平运行中的动作序列：

tick, tick, tick, ...

以下是一次不公平运行中的动作序列：

tick, tick, tick

事实上，*Clock*不存在有限的公平运行，因为*tick*动作总是使能的。如下是公平的：

tick, tick, request, tick, tick, clock(4), *tick, tick, ...,*

这是因为一旦*Clock*对单个需求做出回应，没有时钟动作是使能的。最后，以下是不公平的：

tick, tick, request, tick, tick, tick, ...,

因为在*request*事件之后，时钟任务是使能的，但一直没有时钟动作发生。

以下定理类似于定理8.1~8.3。

定理8.4 令$\{A_i\}_{i\in I}$是一个兼容自动机集合，$A=\prod_{i\in I}A_i$

1. 如果$\alpha\in fairexecs(A)$，则对于任意$i\in I$，$\alpha|A_i\in fairexecs(A_i)$。
2. 如果$\beta\in fairtraces(A)$，则对于任意$i\in I$，$\beta|A_i\in fairtraces(A_i)$。

定理 8.5 令$\{A_i\}_{i\in I}$是一个兼容自动机集合，令$A=\prod_{i\in I}A_i$。对于任意$i\in I$，设α_i是A_i的一个运行，β是一个由$ext(A)$中动作组成的序列，使得对于任意$i\in I$，$\beta|A_i=trace(\alpha_i)$成立，则存在一个$A$的公平运行$\alpha$，使得对于任意$i\in I$，$\beta=trace(\alpha)$且$\alpha_i=\alpha|A_i$成立。

定理8.6 令$\{A_i\}_{i\in I}$是一个兼容自动机集合，$A=\prod_{i\in I}A_i$。设β是一个由$ext(A)$中动作组成的序列。如果对于任意$i\in I$，有$\beta|A_i\in fairtraces(A_i)$，则$\beta\in fairtraces(A)$。

证明 此证明留做一道习题。 □

使用合成来表示分布式系统时，根据定理8.1~定理8.3和定理8.4~定理8.6，我们可以使用模块化的方法来对系统的行为进行推理。 214

例 8.3.3 公平性

我们考虑例子8.2.1中由三个进程和三个通道组成的系统的公平运行。在每个公平运行中，每个被发送的消息最终都被收到。对于任意i，在包含至少一个$init_i$事件的公平运行中，每个进程都发送无限多的消息给其他所有进程，每个进程执行无限多次*decide*步。

另一方面，在不包含至少一个*init*事件的公平运行中，没有进程执行*decide*步。注意公平性并不对*init*事件的发生做任何要求，与P_i相关的$init_i$事件的数目既可以是有限的（甚至为0），也可以是无限的。

我们以定理8.7结束本节，这个定理说明每个有限运行（或轨迹）可扩展成一个公平运行（或轨迹）。

定理8.7 令A为任意的I/O自动机。

1）如果α是A的一个有限运行，则存在A的一个公平运行，该运行以α开始。
2）如果β是A的一个有限轨迹，则存在A的一个公平轨迹，该轨迹以β开始。
3）如果α是A的一个有限运行而β是由A的输入动作组成的任意（有限或无限）序列，则存在A的一个公平运行$\alpha\cdot\alpha'$使得由α'中输入动作组成的序列正好是β。
4）如果β是A的一个有限运行，而β'是由A的输入动作组成的任意（有限或无限）序列，则存在A的一个公平运行$\alpha\cdot\alpha'$使得$trace(\alpha)=\beta$且由α'中输入动作组成的序列正好是β'。

证明 具体证明留做一道习题。 □

8.4 问题的输入和输出

能以I/O自动机解决的问题都有某种类型的输入和输出。在同步模型中，我们一般使用特殊的状态变量来表示这些输入和输出，在初始状态中输入值被存放到指定的变量中，而输出值则在指定的“写一次”变量中出现。异步模型也可以采用这种方法。然而，由于I/O自动机具有输入和输出动作，利用它的输入输出动作来表示系统的输入和输出显得更为自然。 215

8.5 属性与证明方法

I/O自动机不但能够用来精确地描述异步自动机，还可以对描述系统工作的断言进行形式化和证明。在本节中，我们描述一些异步系统的典型属性和一些证明这些性质的典型方法。

在有关异步算法的几章中（第10~13、15~22章），我们在这儿使用方法（和特别的参数）来证明异步算法的属性。参数是否使用一个典型的方法，它们都可使用I/O自动机。

8.5.1 不变式断言

需要证明的最基本性质是不变式断言，或简称为不变式。在本书中，自动机A的不变式断言被定义为对A的所有可达状态都为真的属性。

当前问题的状态经由一定的运行步数产生，若要证明不变式，可对这些运行步数进行归纳。更一般地，利用已证的不变式并进行归纳证明，可以证明另一个（或几个）不变式。

如前所述，不变式断言也用来证明同步算法的性质。在同步环境中，不变式的证明与任意轮数之后的系统状态相关。而在异步环境中，不变式的证明与任意步数之后的系统状态相关。由于异步算法的推理粒度更小，其证明通常更长、更详细和更困难。

8.5.2 轨迹属性

从用户的角度来看，I/O自动机可被视为“黑盒子”。用户只能看见自动机运行（或公平运行）的轨迹。自然的，一些待证的I/O自动机属性被形式化成其轨迹或公平轨迹的属性。

216 正式地，一个*轨迹属性*P的组成如下：

- $sig(P)$，一个不包含内部动作的签名。
- $traces(P)$，一个动作序列（有限或无限）集合，这些动作在$acts(sig(P))$中。

也就是说，一个轨迹属性指定了一个外部接口和一个在该接口中可见的序列的集合，也可以干脆把序列集合称为轨迹属性。将$acts\ (sig(P))$简写成$acts\ (P)$，将$in\ (sig(P))$简写成$in(P)$，如此类推。

当以下条件中的一个（至少一个）成立时，称I/O自动机A满足轨迹属性P：

1）$extsig(A) = sig(P)$ 且 $traces(A) \subseteq traces(P)$。

2）$extsig(A) = sig(P)$ 且 $fairtraces(A) \subseteq traces(P)$。

在任一条件中，一个直观的想法是属性P允许A能够产生的每个外部行为。注意，我们并不要求该命题的逆命题也成立，即不要求P的所有轨迹都能在A中呈现。然而，这里给出的包含性语句并不简单：事实上由于A是输入使能的，确保对于任意可能的输入动作序列，$fairtraces(A)$（和$traces(A)$）中都必须含有A的一个响应与之对应。如果$fairtraces(A) \subseteq traces(P)$，则所有结果序列都必须被包含在属性$P$中。

由于自动机“满足一个轨迹属性”这种说法具有一定的歧义性，我们将在提到它时加以明确说明。

例 8.5.1 自动机和轨迹属性

考虑输入集为{0}和输出集为{1,2}的自动机及其轨迹属性。首先假定$traces(P)$是一个由{0,1,2}上至少包含一个1的序列组成的集合。则$fairtraces(A) \subseteq traces(P)$的意思是：在每个公平运行中，$A$必须输出至少一个1。很容易设计出一个相应的I/O自动机——例如，它可以包括一个全部工作就是输出1的任务。公平性条件确保这个任务能够得到机会去输出1。另一方面，并不存在自动机A使得$traces(A) \subseteq traces(P)$，因为$traces(A)$总是包含空字符串$\lambda$，而空字符串不包含1。

现在假设$traces(P)$是由{0,1,2}上至少包含一个0的序列组成的集合。这种情况下，则不存在自动机A（和给定的外部接口）使得$fairtraces\ (A) \subseteq traces\ (P)$，因为$fairtraces\ (A)$必然包含一

些不包含任何输入的序列。

现在来定义轨迹属性的合成。当可数集合$\{P_i\}_{i\in I}$中的进程都是兼容的时候，该集合的轨迹属性是兼容的。合成$P=\prod_{i\in I}P_i$是满足如下条件的轨迹属性： 217

- $sig(P)=\prod_{i\in I}sig(P_i)$
- $traces(P)$是由P中外部动作序列组成的集合β，使得对于任意$i\in I$，$\beta|acts(P_i)\in traces(P_i)$

8.5.3 安全与活性属性

本节定义两种重要而特殊的轨迹属性：安全属性（*safety properties*）与活性属性（*liveness properties*），并给出涉及这些属性的两个基本结果及其证明方法。

安全属性 如果轨迹属性P满足以下条件，则P是一个轨迹安全属性，简称为安全属性：

1）$traces(P)$非空。

2）$traces(P)$是前缀封闭的，即如果$\beta\in traces(P)$且β'是β的一个有限前缀，则$\beta'\in traces(P)$。

3）$traces(P)$是极限封闭的，即如果无数个序列$\beta_1,\beta_2,\ldots$，都是$traces(P)$中的有限序列，且对于任意i，β_i是β_{i+1}的前缀，令序列β是这种连续次序中β_i的极限，则β也在$traces(P)$中。

安全属性常常被解释为一些特定的“坏”事从不发生。如果轨迹中发生一件坏事，则它是由轨迹中某一特定事件引起的，因此将极限封闭条件包含在定义中是合理的。而且，如果轨迹中没有坏事发生，则轨迹的任意前缀中也没有坏事发生，因此加入前缀封闭条件是合理的。最后，坏事必然可能发生在某一事件之后，即没有坏事可以在空序列λ中发生，因此非空性条件也是一个合理的条件。

例 8.5.2 轨迹安全属性

设$sig(P)$由输入$init(v)$和输出$decide(v)$组成，其中，$v\in V$。假设$traces(P)$是由$init$和$decide$动作组成的序列的集合，其中$decide(v)$只能在$init(v)$之后发生（$v\in V$），则P是一个安全属性。 218

如果P是一个安全属性，则语句$traces(A)\subseteq traces(P)$等价于语句$fairtraces(A)\subseteq traces(P)$，或者又等价于这样的语句：$A$的有限轨迹全在$traces(P)$中（具体证明留做一道习题）。对于一个给定的自动机$A$，证明这三个语句的最简单方法通常是去证明$A$的有限轨迹都在$traces(P)$中。这一般通过对产生给定轨迹的有限运行的长度进行归纳来证明。其策略与证明不变式的策略相似。事实上，通过增加一个状态变量到A中来跟踪当前产生的轨迹，安全属性P可以被转换成为一个关于该自动机状态的不变式。

活性属性 对于一个轨迹属性P，如果$acts(P)$上的每个有限序列在$traces(P)$中都有某个扩展，则P被称为轨迹活性属性，或者简称为活性属性。

活性属性常常被非正式地解释为一些特定的“好”事总是发生（尽管正式的定义包含比这更复杂的命题）。我们假设无论当前发生什么，好事总会在未来的某一时刻发生。

例 8.5.3 轨迹活性属性

设$sig(P)$由输入$init(v)$和输出$decide(v)$,其中$v\in V$组成，$traces(P)$是由$init$和$decide$动作的序列组成的集合β。如果对于β中的每个$init$事件，都有一些β中的$decide$事件在这个事件后发生，则P是一个活性属性。如果对于β中的每个$init$事件，都有无数个β中的$decide$事件在这个事件之后发生，则P也是一个活性属性。

若要证明对于自动机A和活性属性P，$fairtraces(A)\subseteq traces(P)$，即$A$的公平轨迹都满足轨迹属性。可以使用基于时序逻辑（*temporal logic*）的方法来加以证明。一个时序逻辑由一种逻辑语言和一个证明规则的集合组成，逻辑语言包括诸如“最终地”和“总是”等时序概念的签名，

证明规则用于描述和证明运行性质。

另外一种用于证明活性的方法是*演进函数方法*，该方法专门为证明某些特定目标最终可达到而设计。它定义一种从自动机状态到合式集的“进程函数”，并确保特定动作能够不断地减
219 少函数的值直至达到目标。可以使用时序逻辑来对演进函数方法进行形式化。在本书中，活性属性被非形式化地证明，但所有活性证明都可以使用时序逻辑来形式化。

下面两个定理说明了安全属性和活性属性之间的基本连接。第一个定理说明不存在非平凡的轨迹属性，它既具有安全属性，又有活性属性。

定理8.8　如果P既是安全属性又是活性属性，则P是$acts(P)$中所有（有限或无限）动作序列的集合。

证明　设P既是安全属性又是活性属性，令β是$acts(P)$中动作的任意序列。如果β是有限的，由P是活性属性得知，$traces(P)$必有β的扩展β'。因为P是安全属性——特别的，$traces(P)$是前缀封闭的——所以$\beta \in traces(P)$。因此，$acts(P)$中动作的任意序列都在$traces(P)$中。

另外，如果是β无限的，则对于任意$i \geqslant 1$，将β的长度为i的前缀定义为β_i。如前所述，每个β_i都在$traces(P)$中。因为P是安全属性——特别的，由于$traces(P)$是极限封闭的——可得$\beta \in traces(P)$。　□

第二个定理指出每个轨迹属性都可以使用一个安全属性和一个活性属性的交集（或合取）来表达。

定理8.9　若P是满足$traces(P) \neq \phi$的任一轨迹属性，则存在安全属性S和活性属性L使得：

1）$sig(S) = sig(L) = sig(P)$。

2）$traces(P) = traces(S) \cap traces(L)$。

证明　令$traces(S)$是$traces(P)$的前缀封闭和极限封闭，即是$acts(P)$上包含$traces(P)$的序列且满足前缀封闭和极限封闭条件的最小集。显然，S是安全属性。令

$$traces(L) = traces(P) \cup \{\beta : \beta\text{是一个有限序列且}traces(P)\text{中不包含}\beta\text{的扩展}\}。$$

则L是一个活性属性。为了说明这点，考察$acts(P)$中一个动作有限序列β，如果$traces(P)$中有β的扩展，则由于$traces(P) \subseteq traces(L)$，该扩展肯定也在$traces(L)$中。另一方面，如果$traces(P)$中没有$\beta$的扩展，则$\beta$在$traces(L)$中被显式地定义。在这两种情况中，$traces(L)$中存在一个$\beta$的
220 扩展，所以L是一个活性属性。

现在来证明$traces(P) = traces(S) \cap traces(L)$。显然$traces(P) \subseteq traces(S) \cap traces(L)$，这是因为$S$和$L$的轨迹被显式地定义为必须包括$P$的轨迹。下一步要证明$traces(S) \cap traces(L) \subseteq traces(P)$。使用反证法，假设$\beta \in traces(S) \cap traces(L)$，但$\beta \notin traces(P)$，根据$L$的定义，$\beta$是一个在$traces(P)$中不存在扩展的序列。但是，$\beta \in traces(S)$，而$traces(S)$是$traces(P)$的前缀极限封闭；由于$\beta$是有限序列，则$\beta$必是$traces(P)$中一个动作序列的前缀。矛盾。　□

到此为止，我们只定义了轨迹的安全属性和活性属性，可以类似地定义运行的安全属性和活性属性，其结论与轨迹的结论类似。在以后的几章中，运行属性常常被分为安全属性和活性属性。

8.5.4　合成推理

逐一地对组件自动机进行推理，常常有助于自动机合成系统属性的证明。本节给出一些这种“合成”推理的例子。

首先，如果$A = \prod_{i \in I} A_i$且每个A_i满足轨迹属性P_i，则A满足乘积轨迹属性 $P = \prod_{i \in I} P_i$。定理8.10对此进行了详细说明。

定理8.10 令$\{A_i\}_{i \in I}$是一个兼容自动机集合，令$A = \prod_{i \in I} A_i$；而$\{P_i\}_{i \in I}$是一个（兼容的）轨迹属性集合，$P = \prod_{i \in I} P_i$。

1）如果对于任意i，$extsig(A_i) = sig(P_i)$且$traces(A_i) \subseteq traces(P_i)$，则$extsig(A) = sig(P)$且$traces(A) \subseteq traces(P)$。

2）如果对于任意i，$extsig(A_i) = sig(P_i)$且$fairtraces(A_i) \subseteq traces(P_i)$，则$extsig(A) = sig(P)$且$fairtraces(A) \subseteq traces(P)$。

证明概要 可以利用定理8.1（即A_i的轨迹由合成系统A的轨迹在A_i上的投影而得）来证明第1点；第2点的证明与定理8.4的证明类似。 □

例 8.5.4 满足乘积轨迹属性

考察例子8.2.1中的合成系统，每个进程自动机P_i（以轨迹包含的方式）满足这样的轨迹属性：任意$decide_i$事件都有一个前驱$init_i$事件。另外，每个通道自动机C_{ij}满足这样的轨迹属性：$receive_{ij}$事件中消息的序列是$send_{ij}$事件中消息的序列的一个前缀。 221

由定理8.10得知，合成系统满足乘积轨迹安全属性，这意味着在合成系统的任意轨迹中，以下条件成立：

1）对于任意i，任意$decide_i$事件都有一个前驱$init_i$事件。

2）对于任意i和j，$i \neq j$，$receive_{ij}$事件中消息的序列是$send_{ij}$事件中消息的序列的一个前缀。

其次，若要表明一个特定的动作序列是合成系统$A = \prod_{i \in I} A_i$的一个轨迹（这常见于A是一个用做问题阐述的抽象系统的时候），根据定理8.3，只需表明该序列在每个系统组件上的投影是该组件的一个轨迹就足够了。根据定理8.6，公平轨迹的情况与之类似。

再次，考虑安全属性的合成证明。若要表明一个合成系统$A = \prod_{i \in I} A_i$满足安全属性P，可采用的一个策略是去表明组件A中没有组件首先违反P。该策略是有用的，例如，若要证明一对组件在交替地发收信号的时候遵守彼此之间的“握手协议”。如果能够证明这两个组件都不会首先违反握手协议，则可以得出该协议被遵守的结论。

正式地，先来定义自动机“保持”安全属性的概念。设A是一个I/O自动机，P是一个满足$acts(P) \cap int(A) = \phi$和$in(P) \cap out(A) = \phi$的安全属性。对于任意$\pi \in out(A)$和每个不包含$A$中内部动作的动作序列$\beta$，如果$\beta \lceil acts(P) \in traces(P)$且$\beta\pi \lceil A \in traces(A)$，则$\beta\pi \lceil acts(P) \in traces(P)$。这说明了$A$不会第一个违反$P$：只要$A$的环境给$A$提供输入时，累计行为满足$P$，则$A$执行输出动作时累计行为必然满足$P$。

保持安全属性的关键是如果合成系统中的所有组件都保持一个安全属性，则整个系统也保持该属性。此外，如果合成系统是封闭的，则它实质上满足安全属性。 222

定理8.11 令$\{A_i\}_{i \in I}$是一个兼容自动机集合，$A = \prod_{i \in I} A_i$。令P是一个满足$acts(P) \cap int(A) = \phi$和$in(P) \cap out(A) = \phi$的轨迹属性。

1）如果对于任一$i \in I$，A_i保持P，则A保持P。

2）如果A是一个封闭自动机，A保持P，且$acts(P) \subseteq ext(A)$，则$traces(A) \lceil acts(P) \subseteq traces(P)$。

3）如果A是一个封闭自动机，A保持P，且$acts(P) = ext(A)$，则$traces(A) \subseteq traces(P)$。

证明　此证明留做一道习题。　□

例 8.5.5　自动机保持属性

设A是一个带有输出a和输入b的自动机，而B是一个带有输出b和输入a的自动机。考察这样一个安全属性P：$sig(P)$没有输入，但a和b都是其输出，而$traces(P)$是由所有从a开始的a和b交替出现的无限或有限序列（加上空序列λ）组成的集合。令P代表A与B之间的握手协议，由A启动握手操作。

假设A有一个变量$turn$，其值在集合$\{a, b\}$中，初值为a。A的转换函数如下：

转移:

a
　Precondition:
　　$turn = a$
　Effect:
　　$turn := b$

b
　Effect:
　　$turn := a$

所以，A能够在开始时或者接收到一个b输入时运行a。如果A在得到机会对下一个a作出反应之前已经连续接收到两个b，则只对一个a做出反应。

自动机B有一个变量$turn$，其值在集合$\{a, b\}$中，初值为a，另有一个初值为$false$的布尔变量$error$。B的转换函数如下：

转移:

b
　Precondition:
　　$turn = b$ or $error = true$
　Effect:
　　if $error = false$ then $turn := a$

a
　Effect:
　　if $error = false$ then
　　　if $turn = a$ then $turn := b$
　　　else $error := true$

223

因此，只要B的环境不连续提交两个a，B只能在接收到一个输入a时执行一次b。如果B的环境连续提交两个a，则B设置一个$error$标志，用于允许B在任意时间输出b。

A和B都保持P，由定理8.11可知合成$A \times B$的所有轨迹都在$traces(P)$中。

8.5.5　层次化证明

本节描述一个基于自动机层次的重要证明策略。自动机层次代表在不同抽象层对系统或算法的一系列描述。在抽象系列中从高层到低层的移动被称为*连续求精*。顶层可能只是以自动机形式编写的问题阐述。其下一层通常是系统的一个高度抽象描述：系统可能是集中式而不是分布式的，或者可能具有大粒度的动作或简单而低效的数据结构。更低层越来越像实际的系统或算法：可能更为分布和具有较小粒度的动作，还可能包含优化工作。鉴于这些额外的细节，低层通常比高层更难明白。证明底层自动机属性的最好方法是将它们与高层自动机相联，而不是直接进行证明。

第4、6章包含了在同步环境中这种求精过程的例子。例如，第6章首先提出一个针对停止故障的协议算法*FloodSet*，其形式类似于通信描述形式；然后提出一个该算法的改进（“底层”）版本*optFloodSet*，在该版本中许多消息被删除，使得通信范围更窄。两个算法相关状态之间的模拟关系被用来证明改进算法的正确性。证明过程通过对回合数的迭代来表明模拟关系在整个计算过程得到保持，关键在于使用相同的输入和故障模式来同时运行两个算法，并观察两种运行之间的相似性。

如何将这种模拟手段扩展到异步系统中去呢？与同步模型相比，异步模型在组件执行顺序和伴随每次动作的状态变化方面允许更多的自由。故更难决定应该比较哪些运行。其实只需建

立两种算法之间的一对一关系就足够了，即对于低层自动机的任意运行都有一个高层自动机的运行与之对应。我们通过定义两个自动机之间的模拟关系来达到这种目的。 224

令A和B是具有两个相同外部签名的I/O自动机，A为底层自动机而B为高层自动机。设f是$states(A)$和$states(B)$上的二元关系，即$f \subseteq states(A) \times states(B)$，$(s, u) \in f$ 可用$u \in f(s)$表示。如果以下条件成立，则f是从A到B的模拟关系。

1）若$s \in start(A)$，则$f(s) \cap start(B) \neq \phi$。

2）若s是A的可达状态，$u \in f(s)$是B的可达状态，且$(s, \pi, s') \in trans(A)$，则存在一个以$u$开头并以$u' \in f(s')$结尾的$B$的运行片断$\alpha$，使得$trace(\alpha) = trace(\pi)$。

第一个条件，即*开始条件*，指出A的任意开始状态都有B中的开始状态与之对应。第二个条件，即*步条件*，指出对于A的任意步和B中与该步的初始状态对应的任意状态，都有B中的一个步序列与之对应。这种对应序列可以由一步、多步或者零步组成，只要保持状态之间的对应关系且两自动机的外部动作相同。图8-4显示了一个当π是外部动作时的步对象。以下定理给出了模拟关系的关键属性。

定理8.12 如果存在一个从A到B的模拟属性，则$traces(A) \subseteq traces(B)$。

证明 此证明留做一道习题。 □

特别地，定理8.12意味着被B满足的安全属性也被A满足：如果P是一个轨迹安全属性使得$extsig(A) = sig(P)$且$traces(A) \subseteq traces(P)$，则$extsig(B) = sig(P)$且$traces(B) \subseteq traces(P)$。由于模拟关系的式样整齐，可以应用计算机辅助证明来证明其正确性。

例 8.5.6 模拟证明

一个模拟证明的简单例子是将两个通道自动机进行合成来实现另外一个通道自动机。

令C是例8.1.1中的自动机（在本例中下标被省略）。重命名C中的某些动作来分别得到A和B：B的输出$receive(m)$被重命名为$pass(m)$，A的输入$send\ (m)$被重命名为$pass\ (m)$。令D是合成A和B并隐藏$pass$动作而得到的自动机，注意C和D具有相同的外部接口。

现在来证明$traces(D) \subseteq traces(C)$，为此，定义如图8-5所示的从$D$到$C$的模拟关系。

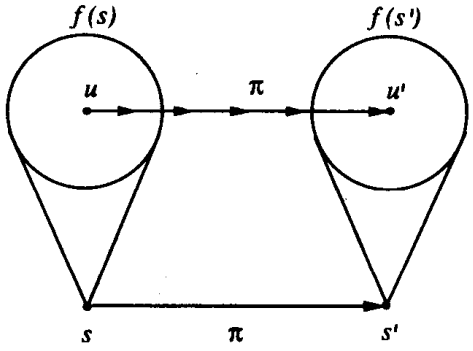


图8-4 模拟关系的步对应关系

225

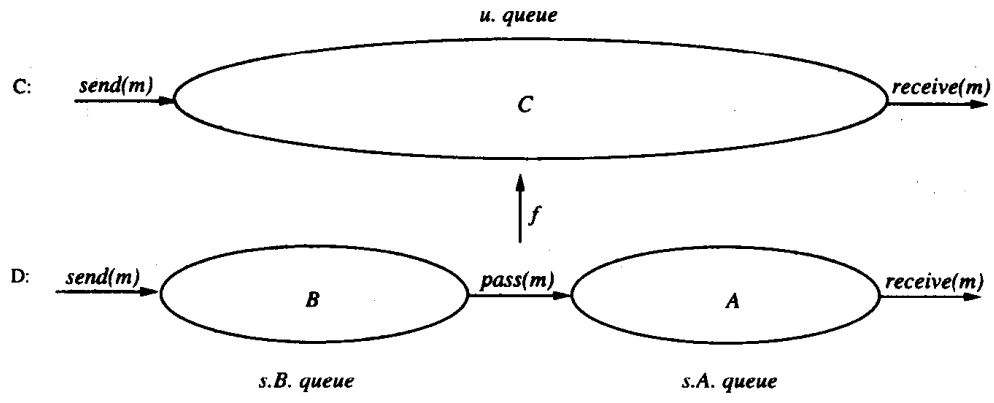


图8-5 从D到C的模拟关系

如果s是D的状态，u是C的状态，假设如下条件成立，则$(s, u) \in f$（签名。用于表示状态的给定值和合成中给定的自动机）。

226 $u.queue$是$s.A.queue$和$s.B.queue$的连接（$s.A.queue$在前）

为了证明f实质上是一个模拟关系，我们需要检查定义中的两个条件。第一个条件易于证明，因为A、B和C的开始状态都是空队列。对于第二个条件，设s是D的状态，$u \in f(s)$是C的状态，$(s, \pi, s') \in trans(D)$。以下根据被执行动作的类型来分情况考虑。

1）$\pi = send(m)$

令C中对应的运行片断由单个$send(m)$步组成，D中的给定步将m加到$s.B.queue$的尾部，C中的步将m加到$u.queue$的尾部，这保持了f的定义中的状态对应关系。

2）$\pi = receive(m)$

令C中对应的运行片断由单个$receive(m)$步组成，D中的给定步从$s.A.queue$的头部删除m。s和u的对应关系意味着m也在$u.queue$的头部，所以$receive(m)$动作实际上在u中使能。然后C中的步从$u.queue$的头部删除m，这保持了f的定义中的状态对应关系。

3）$\pi = pass(m)$

令C中对应的运行片断由0步组成。由于D的步不会影响两个队列的连接，状态对应关系被保持。

因此f是一个模拟关系，由定理8.12可知，$traces(D) \subseteq traces(C)$。

模拟关系有时可以用于证明A满足B的活性属性，其思想在于：从A到B的模拟关系不只是意味着轨迹包含——它意味着在A的每个运行和B的某些运行之间涉及轨迹和状态的紧密关系。这种紧密关系和A的公平性假设一起可用于证明所需的活性属性。

例如，这里介绍一个运行之间更强对应关系的正式描述方法。设A和B是两个具有相同输入和输出的I/O自动机，α和α'分别是A和B的运行，f是$states(A)$和$states(B)$上的二元关系。如果
227 存在一个从α中状态的下标（发生）到α'中状态的下标的映射关系满足以下属性，则称α和α'对应于f：

1）g一致非递减。

2）g覆盖所有α'（即g的范围上限是α'中状态的下标的上限）。

3）g对应的状态对都与f相关。

4）在连续的g对应状态对中，α和α'中的轨迹是相同的。

不难看出模拟关系产生了运行之间的这种对应关系。

定理8.13　如果f是从A到B的模拟关系，则对于A的任意运行α，存在B的运行α'使得α和α'对应于f。

定理8.13可用来根据B的活性属性来证明A的类似活性属性。例如，我们在互斥算法（*TicketME*，定理10.40）和数据链协议（*Stenning*，引理22.2）的证明概要中运用了该策略。

8.6 复杂度衡量

即使I/O自动机模型是异步的，它仍有时间复杂度这一自然概念。给定自动机A，我们为任务划分$task(A)$中等价类的每一个子集定义时间上界。特别地，为任意任务C定义值为正实数或∞的上界$upper_C$。对于A的任一公平运行α，遵循以下条件，一个实数值时间可以与α中任一事件关联：

1）α中事件的时间是单调非减的。

2）若α是无限的，则时间逼近∞。

3）从α的任意点开始，在C的动作发生之前，任务C的使能时间最多为$upper_C$。

粗略地说，任务C的连续执行机会之间的时间间隔有一个上限$upper_C$，以这种方式来与时间关联的运行被称为定时运行。 228

注意，给定$upper_C$上界的集合，将时间与α中事件关联的方法有许多，即有许多定时运行。在所有定时运行中π可能被赋予不同的时间，α中指定事件π的时间以这些时间的上界来衡量。类似的，两个事件可能被分别赋予各自的时间，α中两个事件之间的时间以两个时间之差的上界来衡量。

例 8.6.1 时间分析

令α是例8.2.1的系统中的任意公平运行，系统中所有进程都接收*init*输入。每个进程中的每个任务的时间上界为ℓ，每个通道中的每个任务的时间上界为d。则从最后一个进程接收α中第一个*init*输入动作开始，到所有进程执行*decide*输出为止，所花的时间为$\ell + d + \ell = d + 2\ell$。原因在于：接收*init*输入的最后一个进程对所有邻居执行*send*事件，最多需要时间ℓ，所有信息的传送时间最多为d，每个进程执行*decide*的时间最多为ℓ。

8.7 不可区分运行

现在来定义不可区分（或称为不可分）概念，该概念将会在某些不可能性证明中用到。它类似于在2.4节中针对同步系统的运行而定义的不可分概念。

设α和α'分别是两个合成自动机系统的运行，其中每个系统都包含A，若$\alpha|A = \alpha'|A$，则称α和α'对于A不可分。

8.8 随机化

如同同步系统的情况一样，允许异步系统中的组件基于一些给定的概率分布来做随机选择有时是有用的。为了表示这种随机选择，我们将I/O自动机模型扩展成为一种新的模型——*概率I/O自动机*。概率I/O自动机与I/O自动机相似，不同之处在于其转换：它的转移的形式为(s, π, P)而不是(s, π, s')，其中P是在状态子集上的概率分布。（如果有并不涉及随机选择的步，则使用一个平常分布P来表示它。）每个概率I/O自动机A都有一个*不确定版本*$\mathcal{N}(A)$，该版本通过将每个转换(s, π, s')替换成(s, π, P)而得，其中s'是P的值域中的一个元素。因此，$\mathcal{N}(A)$只是将随机选择换成不确定选择，它还是一种普通的自动机。 229

概率I/O自动机A的运行包含一系列选择对。在每一选择对中，不确定选择在前，并决定下个转换(s, π, P)；随机选择在后，并使用P来决定下个状态。关于选择的限制只有一个：不确定选择（确定下个转换）必须是“公平”的，即所有可能的随机选择序列产生的所有运行都必须是I/O自动机$\mathcal{N}(A)$的公平运行。

如同同步系统的情况一样，关于随机系统计算内容的说法通常是概率化的。当一个说法被提出时，其意图通常是：这个说法对于所有输入和所有不确定选择的公平模式都成立。如同第5章中的做法，我们发明假想“对手”来描述这些输入和不确定选择，我们要求自动机在与任意假想对手的竞争中表现良好。

8.9 参考文献注释

I/O自动机模型最先在Tuttle的硕士论文[217]中提出，Lynch和Tuttle[217, 218]的论文总结了

这一模型的重要特征，对用I/O自动机表示算法的描述和证明零星地散布在关于分布式算法的研究文献中；Afek等和Bloom提出了一些代表性例子[3, 4, 53]。一个使用I/O自动机来表示允许动态进程创建的系统的例子出现在Lynch, Merritt, Weihl和Fekete[207]编写的书《*Atomic Transactions*》中，该例子提出一个表示数据库并发控制算法的框架。I/O自动机模型受许多其他分布式系统模型的影响，其中最著名的是Lynch和Fischer[216]的异步共享存储器模型、Hewitt的Actor模型[7, 81]和Hoare的通信顺序处理模型[159]。

在第2章结尾的参考资料部分我们讨论了不变式断言概念的来源。轨迹属性这一概念来自"调度模块"的定义[217, 218]。安全属性和活性属性的概念来自Lamport[175]以及Alpern和Scheneider[8]的工作。定理8.9来自Scheneider[8]。

230 Manna和Pnueli[219]的书是时序逻辑方面极好的参考资料。Lamport在动作时序逻辑(TLA)方面的工作包括提出一个有用的时序逻辑框架[184]，以及使用该框架来验证算法的成熟方法论。

文献[207]提出一个策略，其中序列通过投影来给出合成系统A中所有组件的轨迹，从而证明序列实际上是整个系统A的轨迹，其中系统A是对串行地运行所有事务的数据库系统的一个抽象说明。通过分析投影，可以证明那些并发地运行事务的数据库系统产生的特定序列事实上是A的轨迹。这对于这些数据库系统的正确性来说是非常关键的。关于保持安全属性的工作来源于[218]。

模拟关系的来源很多，包括：将Lamport[177]中使用的求精映射概念进行一般化，对Owicki和Gries[235]的历史变量的抽象，并且他们与Park[236]的模拟很相似，Lynch[203, 214]以及Lynch和Tuttle[217, 218]的可能性映射，以及Jonsson[165]的模拟。使用模拟方法来验证异步系统安全性的价值已经得到公认。许多论文和书，如[217, 288, 69, 233, 214, 207, 189, 190]，都包含大量的实用例子。现在相当多的模拟证明依靠计算机辅助和计算机检查来进行，代表性例子包括Nipkow[233]的工作和Sogaard-Andersen、Garland、Guttag、Lynch和Pogosyants[265]的工作，前者使用Isabelle证明器，后者使用Larch证明器。

对随机系统的模拟来源于Segala和Lynch[257]的工作。

并发理论年会上陈述了关于并发系统模型的一般结果。

8.10 习题

8.1 对于例8.2.1中的自动机P_i和$C_{i,j}$，$1 \leqslant i, j \leqslant n$，考虑它们的合成：

a）例子（$n = 2$时）给出的轨迹有惟一一个运行，描述出现在该运行中出现的所有状态。

b）令$n = 3$，设m是任一自然数，描述一个运行，其中所有进程都对m做决定。在给出的运行中，连续$init_1$值组成的串是序列0, 4, 8, 12, ...的前缀，连续$init_2$值组成的串是序列0, 2,
231 0, 2, ...的前缀，连续$init_3$值组成的串是序列0, 1, 0, 1, ...的前缀。

c）令n =3，设m_1, m_2, m_3是任意三个自然数。找出一个运行，其中进程P_i对m_i做决定，$i \in \{1,2,3\}$。*init*值的情况与b）中的一样。

8.2 证明定理8.1、8.2和8.3。应在何处使用兼容性条件？

8.3 证明定理8.4、8.5和8.6。应在何处使用兼容性条件？应在何处使用输入使能条件？

8.4 考察以下两个I/O自动机，注意它们并不使用前提－结果的形式来书写，而是列出了所有组件。

• 自动机A：

$in(A) = int(A) = \emptyset$, $out(A) = \{go\}$,
$states(A) = \{s, t\}$,
$start(A) = \{s\}$,
$trans(A) = \{(s, go, t)\}$,
$tasks(A) = \{\{go\}\}$.

- 自动机B：

$in(B) = \{go\}$, $out(B) = \{ack\}$, $int(B) = \{increment\}$,
$states(B) = \{on, off\} \times \mathbb{N}$,
$start(B) = \{(on, 0)\}$,
$trans(B)) = \{((on, i), increment, (on, i+1)), i \in \mathbb{N}\} \cup$
$\{((on, i), go, (off, i)), i \in \mathbb{N}\} \cup$
$\{((off, i), go, (off, 0)), i \in \mathbb{N}\} \cup$
$\{((off, i), ack, (off, i-1)), i \in \mathbb{N} - \{0\}\}$, and
$tasks(B) = \{\{increment\}, \{ack\}\}$.

描述自动机A、自动机B和自动机$A \times B$的轨迹和公平轨迹的集合。

8.5 a）定义代表可靠消息通道的I/O自动机A，A接收并传递来自字母表M_1和字母表M_2的并集的消息。设在消息通道中，同一字母表的消息保持原来的顺序。而且，如果一个来自字母表M_1的消息在另一来自字母表M_2的消息之前发送，则对应的传递中消息的顺序保持不变。但是如果一个来自M_1的消息在另一来自M_2的消息之后发送的话，则对应的传递中消息的顺序逆转。给出的自动机必须能够呈现所有被允许的外部行为。给出A的所有组件：签名、状态、开始状态、步和任务。 232

b）对于给出的自动机，给出以下部分的例子：一个公平运行、一条公平轨迹、一个非公平运行和一条非公平轨迹。

8.6 描述这样一个I/O自动机：它没有输入动作，输出动作是{0, 1, 2, ...}，公平轨迹是集合S中的序列，其中S由输出集合上长度为1的所有序列组成，也就是说，每个序列由一个非负整数构成。

8.7 证明定理8.7。

8.8 设A为任意自动机，证明存在另一个只包含一个任务的I/O自动机B使得$fairtraces(B) \subseteq fairtraces(A)$（证明包含关系就足够了，不需要证明$fairtraces(B) = fairtraces(A)$）。

8.9 设A为任意自动机，证明存在另一个只包含一个任务的I/O自动机B，B是“确定性”的，即B满足以下条件：

a）有且只有一个初始状态。

b）对任意状态s和任意动作π，最多有一个形式为(s, π, s')的转换。

c）在任意状态中，最多有一个局部控制动作是使能的。

另外，$fairtraces(B) \subseteq fairtraces(A)$。（只需证明包含关系，不需要证明$fairtraces(B) = fairtraces(A)$）。

8.10 给出并证明一条综合了习题8.8和习题8.9的定理。

8.11 重新考虑习题8.8、习题8.9和习题8.10，要求$fairtraces(B) = faritraces(A)$成立。如果这些习题在这个条件下可解，则求解。否则，证明它们是不可解的。

8.12 设P是一个安全属性，证明以下三个关于I/O自动机A的命题等价：

a）$traces(A) \subseteq traces(P)$。 233

b）$fairtraces(A) \subseteq traces(P)$。

c）A的有限轨迹全在$traces(P)$中。

8.13 考虑以下情况的轨迹属性P，在每种情况中，$sig(P)$是没有输入和输出为{1,2}的签名：

a）设$traces(P)$是在{1, 2}上的序列集合，其中2不能立即在1之后出现，证明P是活性属性。

b）设$traces(P)$是在{1, 2}上的序列集合，其中在每个1之后，最终都有一个2出现，证明P是活性属性。

c）设$traces(P)$是在{1, 2}上的序列集合，其中在每个1之后都紧跟着一个2，证明P既不是安全属性也不是活性属性。要求明确地用一个安全属性和一个活性属性的交集来表示P。

8.14 参照对形式化轨迹的做法，将运行的安全属性和活性属性的定义形式化。证明定理8.8和定理8.9之间的类似性。

8.15 证明定理8.11。

234 8.16 证明定理8.12。

第二部分A　异步共享存储器算法

以下几章，从第9~13章，研究异步共享存储器模型上的算法，其中进程异步地执行，并通过共享存储器通信。

这个部分的第1章，即第9章，简单地描述异步共享存储器系统的形式模型。与以前一样，你可先快速浏览此章，然后在必要时查阅具体内容。第10章研究关于互斥的基本问题，第11章研究更通用的分布式资源分配问题。第12章包含针对易错异步系统中一致性问题的基本结论。最后，第13章对原子对象（用于对分布式系统编程的强力抽象对象）进行了研究。

第9章 建模III：异步共享存储器模型

在本章中，我们给出异步共享存储器系统的形式化模型。与第8章中的异步系统一样，该模型采用通用的I/O自动机模型来描述。

与网络系统一样，一个共享存储器系统由一个通信进程集合组成。不同的是，这里的进程在共享变量上执行即时操作，而不是在通信通道上发送和接收消息。

9.1 共享存储器系统

非正式的说，一个异步共享存储器系统由一个有限的进程集合组成，这些进程在一个有限的共享变量集合上进行交互。变量只能用来协助系统中进程的通信。然而，为了使得其他部分也能够与共享存储器系统交互，假定每个进程都拥有一个端口，通过这个端口，进程能够使用输入和输出动作来与外部世界交互。交互过程如图9-1所示。

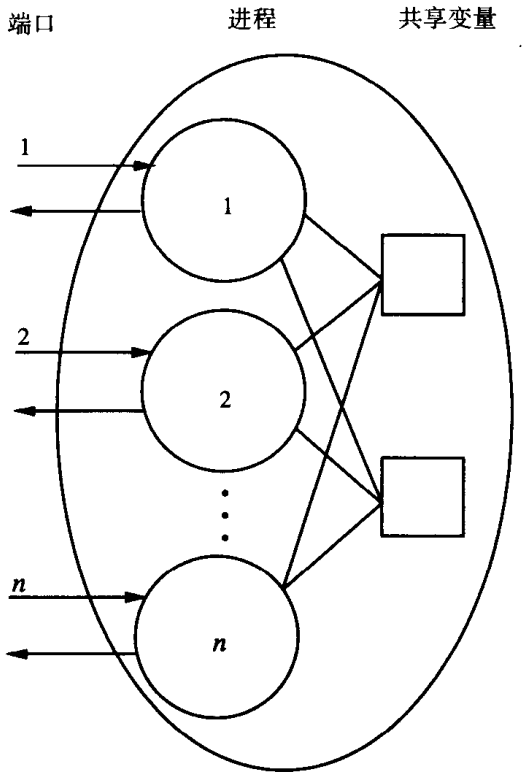


图9-1 一个异步共享存储器系统

我们使用I/O自动机来表示共享存储器系统，事实上，我们只使用单个I/O自动机，其外部接口由所有端口上的输入动作和输出动作构成。如果使用多个自动机（为每个进程和每个共享变量设置一个自动机），可能显得更为自然，但这将引起不必要的复杂化，故我们不采用这种方式。例如，若每个进程和每个共享变量都是一个I/O自动机，则使用通常的I/O自动机合成方法来将它们混合起来后，我们得到一个系统。在该系统中，进程i
237 在共享变量x上的一个操作可以用一对事件来建模：一个调用事件，作为进程i的输出和变量x的输入；和随后的一个回应事件，作为变量x的输出和进程i的输入。但系统可能有一些运行，其中这些事件对被拆开。例如，在第一个调用返回之前可能又有几个调用发生。这种行为不应该在我们对之进行建模的共享存储器系统中出现。

解决这种困境的方法之一是只考虑所有可能运行的特定子集，其中调用和相应的回应连续发生。另外一种方法是只有进程被表示成为I/O自动机，而共享变量以其他类型的状态机器（其中调用和回应合起来作为一个事件）来表示。这样，必须定义一种将进程和变量自动机合成为I/O自动机的合成操作。由于这两种方法都引入了各自的复杂性，如运行集合的受限子集、
238 事件对、新类型状态机或者新操作等，我们采用另一方法：用一个大I/O自动机A来表示整个系统。我们通过在事件上施加局部性限制来捕捉自动机A中进程和变量结构的信息。

与同步网络模型相似，假设系统中的进程从1到n编号。进程i的状态集为$state_i$，其中包含

初始状态$start_i$；系统中每个共享变量x的值集为$value_x$，其中包含初始值$initial_x$。在系统自动机A的状态集$states(A)$中，每个状态都由任意进程i的状态集$state_i$中的一个状态和任意共享变量x的值集$value_x$中的一个值组成。类似的，$start(A)$中的每个状态都由任意进程i的$start_i$中的一个状态和任意共享变量x的$initial_x$中的一个值组成。

$act(A)$中的每个动作都与某一进程相关。此外，$int(A)$中的某些内部动作可能与某个共享变量相关。进程i使用相关的输入动作和输出动作与外界交互，我们称这些动作在端口i上发生。进程i 中不与共享变量相关的内部动作用于局部计算；而与共享变量x相关的内部动作则用于在x上执行操作。

转移集合$tran(A)$拥有一些局部性限制，用于表示系统中的进程和共享变量结构。首先，考虑与进程i相关但不与任意变量相关的动作π。如前文所述，π用于局部计算，任意π步都只涉及进程i的状态。也就是说，π转移集合可由一些形式为(s, π, s')，$s, s' \in state_i$的三元组生成，其中其他进程的状态和共享变量的值的合成被粘贴到s和s'中（相同的合成被粘贴到s和s'中）。

另一方面，考虑既与进程i相关又与变量x相关的动作π。如前文所述，π被进程i用于在x上执行操作。任意π步都只涉及进程i的状态和x的值。也就是说，π转移集合由一些形式为$((s, v), \pi, (s', v'))$，$s, s' \in state_i$，$v, v' \in value_x$的三元组组成，其中其他进程的状态和其他共享变量的值的合成被粘贴到s和s'中。这里有一个技巧：如果π既涉及进程i的状态又涉及x的值，尽管最终变化可能取决于x的值，但π能否使能只取决于进程i的状态。也就是说，如果当i的状态是s而x的值是v的时候，π是使能的，则当i的状态是s而x变成其他值v'时π也是使能的。 239

任务划分$task(A)$必须与进程结构一致：即每个等价类（任务）只能包括一个进程的若干局部控制动作。很多时候进程和任务一一对应，比如进程是串行程序时的例子。这样，在8.3节中给出的I/O自动机公平性标准定义指明每个进程都得到无限多步的执行机会。更一般的情况是，当每个进程有几个任务时，公平性定义指明每个任务都得到无限多的执行机会。

例 9.1.1 共享存储器系统

设V为一个固定的值集合。共享存储器系统由n个从1到n编号的进程和一个共享变量x组成，其中x的值在$V \cup \{unknown\}$之中，且初始值为$unknown$。输入形式为$init(v)_i$，其中$v \in V$，i是进程的编号；输出形式为$decide\ (v)_i$；内部动作的形式为$access_i$。所有下标为i的动作都与进程i相关；另外，$access$动作与变量x相关。

进程i接收一个$init(v)_i$输入后访问x，如果它发现$x = unknown$，则将值v写到x中并决定v的值；如果它发现$x = w$（这里$w \in V$），则不把任何值写入到x中，但决定w的值。

正式地讲，每个集合$states_i$都由局部变量组成。

i的状态：

$status \in \{idle, access, decide, done\}$, initially $idle$
$input \in V \cup \{unknown\}$, initially $unknown$
$output \in V \cup \{unknown\}$, initially $unknown$

转移函数为：

i的转移函数：

$init(v)_i$
  Effect:
    $input := v$
    if $status = idle$ then
      $status := access$

$access_i$
  Precondition:
    $status = access$
  Effect:
    $status := decide$

$decide(v)_i$
  Precondition:
    $status = decide$
    $output = v$
  Effect:
    $status := done$

status = *access*
Effect:
if $x = unknown$ then $x := input$
output := x
240 *status* := *decide*

任务与进程一一对应，任务中包含进程的全部*access*和*decide*动作。

不难看出，在A的每个公平运行α中，任意接收*init*输入的进程最终都产生一个*decide*输出。此外，每个运行（无论公平与否，且任意数目的*init*事件可以发生在任何位置）都满足“一致性属性”（即没有两个进程决定出不同的值）和“有效性属性”（即每个决定值都是某个进程的初始值）。

我们可以根据8.5.2节中的定义，用轨迹属性来对这些正确性声明进行形式化。例如，令P是这样的轨迹属性：$sig(P) = extsig(A)$，且$trace(P)$是由$act(P)$中动作组成的、满足以下条件的序列β的集合。

1）对于任意i，如果有且只有一个$init_i$事件出现在β中，则有且只有一个$decide_i$事件出现在β中。

2）对于任意i，如果没有$init_i$事件出现在β中，则没有$decide_i$事件出现在β中。

3）（一致性）如果$decide(v)_i$事件和$decide(w)_j$事件都出现在β中，则$v = w$；

4）（有效性）如果$decide(v)_i$事件出现在β中，则某个$init(v)_j$事件（对于同一个v）出现在β中。

然后可以证明$fairtrace(A) \subseteq trace(P)$。具体证明留做一道习题。

9.2 环境模型

有时候，把系统的环境也模拟成以自动机来表示是有用的。这提供了一种简单的方法来描述那些关于系统行为的假设。例如，在例9.1.1中，我们希望阐述：对于每个i，环境只提交一次$init_i$输入，或者最少提交一次$init_i$输入。对于现实的共享内存系统，环境常常可以被描述成为互不相关的用户自动机的集合，其中每个端口对应一个自动机。

例 9.2.1　环境模型

我们来描述一个在例9.1.1中描述过的共享存储器系统A。环境以单个I/O自动机表示，该自动机由一个用户自动机U_i组成（使用8.2.1节中针对I/O自动机定义的组成操作），其中每个进程
241 i都有一个用户自动机U_i。U_i的代码如下：

U_i automaton:

Signature:

Input:
　$decide(v)_i$, $v \in V$
Internal:
　$dummy_i$
Output:
　$init(v)_i$, $v \in V$

States:

status $\in \{request, wait, done\}$, initially *request*
decision $\in V \cup \{unknown\}$, initially *unknown*
error, a Boolean, initially *false*

Transitions:

$init(v)_i$　　$decide(v)_i$

Precondition:
status = *request* or *error* = *true*
Effect:
if *error* = *false* then *status* := *wait*

dummy$_i$
Precondition:
error = *true*
Effect:
none

Effect:
if *error* = *false* then
if *status* = *wait* then
decision := *v*
status := *done*
else *error* := *true*

Tasks:
All locally controlled actions are in one class.

因此，U_i最先执行一个$init_i$动作，然后等待一个决定。如果共享存储器系统产生一个没有对应前继$init_i$的决定，或者产生两个决定，则U_i设置一个*error*标志，该标志允许系统在任意时刻输出任意数目的*init*（$dummy_i$动作的存在允许U_i选择不产生输出）。当然，给出的共享存储器系统不会产生这些错误。

图9-2显示共享存储器系统A和所有U_i，$1 \leqslant i \leqslant n$的合成图。这种合成表现良好：在合成的每个公平运行中，对于所有i，都有且只有一个$init_i$事件和只有一个*decide*(i)事件。另外，*decide*(i)事件满足相应的一致性条件和有效性条件。

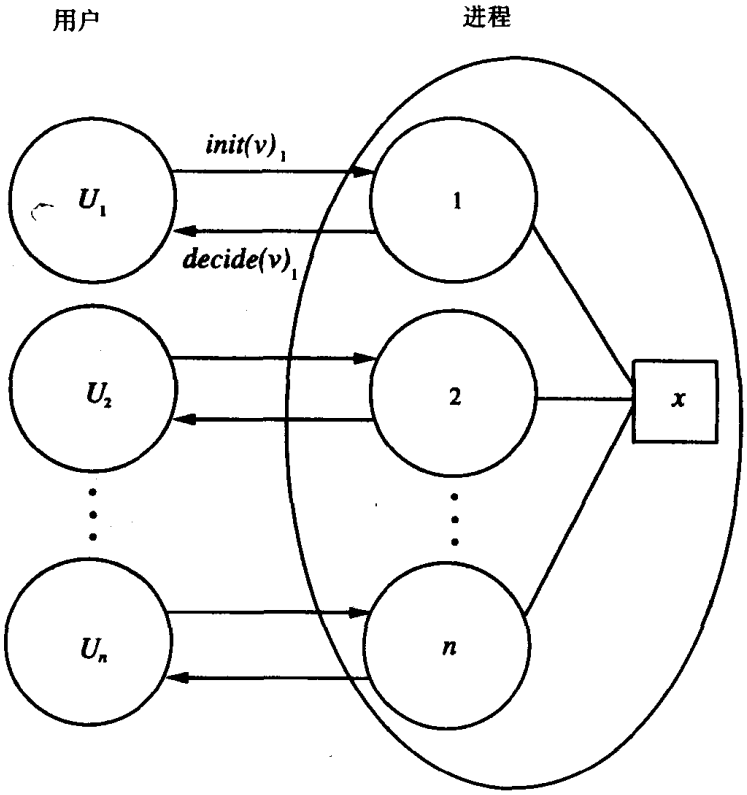


图9-2 用户和共享存储器系统

更正式地讲，令Q是一个轨迹属性，其中$sig(Q)$由所有i和u的输出$init(v)_i$和$decide(v)_i$组成，且$trace(Q)$是由$act(Q)$中动作组成的、满足以下条件的序列β的集合： 242

1）对于任意i，β中有且只有一个$init_i$事件和随后的一个$decide_i$事件；

2）（一致性）若$decide(v)_i$和$decide(w)_j$都出现在β中，则$v = w$。

3）（有效性）如果$decide(v)_i$事件出现在β中，则某个$init(v)_j$事件（对于同一个v）出现在β中。

243 然后可以证明$fairtrace(A \times \prod_{1 \leq i \leq n} U_i) \subseteq trace(Q)$。具体证明留做一道习题。

9.3 不可区分状态

我们定义一个在第10章的一些不可能性证明中会用到的概念——不可区分性。

对于一个n进程共享存储器系统A和一个由用户U_i, $1 \leq i \leq n$组成的集合，令s和s'是合成系统$A \times \prod_{1 \leq i \leq n} U_i$的两个状态。如果进程$i$的状态、$U_i$的状态和所有共享变量的值在$s$和$s'$中都是相同的，则称$s$和$s$对于进程$i$是不可区分的，记为$s \overset{i}{\sim} s'$。

9.4 共享变量类型

在给出的共享存储器系统的通用定义中，我们并没有限制进程访问共享变量时所执行操作的类型。也就是说，当一个进程i访问变量x时，根据i的旧状态和x的旧值，可以随意地改变i的状态和x的值。但实际上，共享变量一般只支持一些固定操作，例如读 - 写操作或者混合读 - 改 - 写操作等。在本节中，我们定义了变量类型的概念，对于共享存储器系统来说，变量存取必须遵守类型限制[⊖]。

一个变量类型包括：

- 一个值集合V。
- 一个初始值$v_0 \in V$。
- 一个调用集合。
- 一个回应集合。
- 一个函数f: 调用 $\times V \rightarrow$ 回应 $\times V$

函数f说明当一个给定的调用到达一个已有给定值的变量时发生的情况；它描述了变量的新值和返回的回应。注意一个变量类型并不是一个I/O自动机，即使它的一些组件看起来颇像I/O自

244 动机组件。更重要的是，在变量类型中，调用和回应都是函数的一部分，在函数应用中一起发生；而在I/O自动机模型中，输入和输出是相互独立的动作(其他动作可能在它们之间发生)。

设有一个共享存储器系统A，称系统A中的共享变量x属于给定类型是什么意思呢？首先，这意味着集合$value_x$必须等于该类型的值集合V，x的初始值集合$initials_x$也只包含一个元素v_0。另外，所有涉及x的转移都必须能够使用该类型允许的调用和回应来描述。换而言之，所有涉及x的动作都必须与该变量类型的某个调用a相关，涉及i和a的转移的集合必须能够使用以下形式来描述，其中p是一些$state_i$上的谓词表达式，而g是满足$g \subseteq state_i \times$ 回应 $\times states_i$的关系（代码中记号$state_i$代表进程i的状态）。

涉及i和a的转移

Precondition:

$p\ (state_i)$

Effect:

$(b, x) := f(a, x)$

$state_i :=$ 任意满足$(state_i, b, s) \in g$的s。

⊖ 这里的定义要求变量的行为是确定性的。可以扩充到非确定性的情况，但本书的结果并不需要这种扩充，故我们避免了这种复杂性。

这些代码意味着，当进程i决定使用调用a来访问变量x时，这个决定是根据谓词表达式p(与i的状态相关)而做出的。若要实施这种访问，变量类型的函数f应用在调用a和变量x的值上，以决定回应b和变量x的新值；然后在关系g的允许下，进程i使用回应b来更新状态。

在本书对共享存储器模型的描述中，涉及访问特定类型共享变量的转换并不以上述谓词表达式p和关系g的形式来书写。但是，理论上它们都可以用这种方式表达。

例 9.4.1 读/写共享变量（寄存器）

在多处理机中使用得最频繁的变量类型是只支持读写操作的类型。这种类型的变量被称为读/写变量，或读/写寄存器，或干脆称为寄存器。

一个读/写寄存器具有任一值集合V和任一初始值$v_0 \in V$，其调用是$read$和$write(v)$, $v \in V$，回应是$v \in V$和ack[⊖]，函数f被定义为$f(read, v) = (v, v)$ 和 $f(write(v), w) = (ack, v)$。 245

注意例9.1.1中的x不能被描述成读/写寄存器，因为例子中给定的进程不能使用上述形式来编写。可以通过重写算法，例如，将每次访问分成读步骤和写步骤，来使x成为一个寄存器，结果的进程代码如下所示。$status$的值$access$被两个新的$status$值（即$read$和$write$）取代。

Transitions:

$init(v)_i$
  Effect:
    $input := v$
    if $status = idle$ then $status := read$

$read_i$
  Precondition:
    $status = read$
  Effect:
    if $x = unknown$ then
      $output := input$
      $status := write$
    else
      $output := x$
      $status := decide$

$write(v)_i$
  Precondition:
    $status = write$
    $v = input$
  Effect:
    $x := v$
    $status := decide$

$decide(v)_i$
  Precondition:
    $status = decide$
    $output = v$
  Effect:
    $status := done$

进程i的所有局部控制动作再次组成一个任务。尽管这些代码并不以谓词表达式p和关系g的形式来书写，要以这种形式书写也是很简单的。例如，对于动作$read_i$，谓词表达式p为"$status = read$,"，关系g为三元组(s, b, s')的集合，其中$(s, b, s') \in states_i \times (V \cup \{unknown\}) \times state_i$，$s'$由以下代码从$s$取得：

```
if b = unknown then
  output := input
  status := write
else
  output := b
  status := decide
```

对于动作$write(v)_i$，谓词表达式p为"$status = write$和v=$input$,"，关系g是三元组(s, b, s')的集合，其中$(s, b, s') \in states_i \times (V \cup \{unknown\}) \times states_i$，$s'$由以下代码从$s$取得： 246

```
status := decide
```

所以x是一个读/写共享变量。

⊖ 调用和回应有时候也包括诸如寄存器名字等附加信息。这里忽略这种复杂性。

注意以这种方式来重写算法时，不能继续保证例9.1.1中的一致性条件。

例 9.4.2　读-改-写共享变量

另一重要变量类型允许强大的读 - 改 - 写操作。在一个对共享变量x的即时读 - 改 - 写操作中，进程i能够做以下任何动作：

1）读x。

2）进行计算，可能用x的值来修改i的状态并决定x的新值。

3）将新值写回x。

使用多处理机提供的一般原语来实现通用的读 - 改 - 写操作并不容易。共享存储器模型不但要求对变量的每个访问都不可分割，还要求所有进程都应该得到公平的回合来进行访问操作。实现这种公平性需要某些底层仲裁机制。

如前所述，以变量类型来表示读-改-写变量的可能性并不明显：读 - 改 - 写操作看起来涉及对变量的两种访问（而不是所需的一种访问）。一种方法是让意欲访问变量的进程根据自身状态在调用变量时使用一个函数h。函数h以应用在变量上的形式来表达，它提供进程状态信息以决定转移。函数h在值为v的变量上的应用结果是将变量的值改为$h(v)$并返回变量的旧值v给进程，然后进程根据自己的旧状态和v来改变自己的状态。

正式地，一个读 - 改 - 写变量包括任一值集合V和任一初始值$v_0 \in V$，其调用是所有函数h，$h: V \to V$；回应为$v \in V$；函数f为$f(h, v) = (v, h(v))$。换而言之，f返回旧值并根据提交的函数来更
247 新变量的值。

例如，在例9.1.1中，由进程提交给变量的函数的形式为h_v，这里

$$h_v(x) = \begin{cases} v, & \text{if } x = unknown \\ x, & \text{其他} \end{cases}$$

进程提交的特定h_v将进程的输入作为v的值。返回值*unknown*导致输出值等于输入值，而返回值$v \in V$导致输出值等于v。两种情况都相应地修改进程状态*status*。

例 9.4.3　其他变量类型

在共享存储器多处理器中，许多变量类型都包括读 - 改 - 写的受限形式以及读写等基本操作。读 - 改 - 写的一些通用受限形式包括比较 - 交换（compare-and-swap）、交换（swap）、测试 - 设置（test-and-set）、取 - 加（fetch-and-add）操作。这些操作的定义如下，其中集合V和初始值v_0是固定的。

比较 - 交换操作的调用的形式为*compare-and-swap*(u,v)，$u, v \in V$，而回应是V中的元素。其函数f的定义如下：

$$f(compare\text{-}and\text{-}swap(u,v), w) = \begin{cases} (w, v), & \text{if } u = w \\ (w, w), & \text{其他} \end{cases}$$

也就是说，如果变量的值与第一个参数u相等，则该操作将变量的值置为第二个参数v；否则，该操作不改变变量的值。无论如何，变量的原值被返回。

交换操作调用的形式为*swap*(u)，$u \in V$，而回应是V中的元素。针对交换调用的函数f被定义为

$$f(swap(u), v) = (v, u)$$

也就是说，该操作将输入值u写到变量中并返回变量的原值v。

测试－设置操作的调用形式为*test-and-set*，而回应是V中的元素。针对测试－设置调用的函数f被定义为

$$f(test\text{-}and\text{-}set, v) = (v, 1)$$

248

也就是说，该操作将1写到变量中并返回变量的原值v（假定$1 \in V$）。

最后，取－加操作的调用的形式为*fetch-and-add*(u), $u \in V$，而回应是V中的元素。针对取－加调用的函数f被定义为

$$f(fetch\text{-}and\text{-}add(u), v) = (v, v+u)$$

也就是说，该操作将输入值u加到变量v中并返回变量的原值v（该操作要求集合V支持加法）。

我们以一种很自然的方式来定义变量类型的运行。变量类型的运行是有限序列$v_0, a_1, b_1, v_1, a_2, b_2, v_2, \ldots, v_r$或者无限序列$v_0, a_1, b_1, v_1, a_2, b_2, v_2, \ldots$。其中$v$是$V$中的值，$v_0$是变量类型的初始值，$a$是调用，$b$是回应，而四元组$v_k, a_{k+1}, b_{k+1}, v_{k+1}$满足该类型的函数（亦即，$(b_{k+1}, v_{k+1}) = f(a_{k+1}, v_k)$.）。另外，变量类型的轨迹是从该类型的运行中得到的a和b的序列。

例 9.4.4 读/写变量类型的轨迹

以下是满足$V = \mathbb{N}$和$v_0 = 0$的读/写变量类型的轨迹

$$read, 0, write(8), ack, read, 8$$

作为本节的结束部分，我们为变量类型定义一种简单的合成操作，使得一个独立变量类型集合可被视为一个简单变量类型，其中独立变量类型具有各自的操作，而简单变量类型拥有几个组件，且拥有作用在各个组件之上的操作。

对于一个变量类型的可数集合$\{\mathcal{T}_i\}_{i \in I}$，如果所有调用集合和回应集合是互不相交的，则称它为兼容的。一个变量类型的可数兼容集合的合成$\mathcal{T} = \prod_{i \in I} \mathcal{T}_i$的定义如下：

- 集合V是$\mathcal{T}_i$值集合的笛卡儿积。
- 初始值v_0由$\mathcal{T}_i$中的初始值组成。
- 调用集合是$\mathcal{T}_i$中调用集合的并集。
- 回应集合是$\mathcal{T}_i$中回应集合的并集。

249

- 函数f“组件状的”操作。换而言之，考虑$f(a, w)$, 其中a是$\mathcal{T}_i$的一个调用。函数f将a应用到w的第i个组件中，并使用$\mathcal{T}_i$的函数来获取(b, v)；它返回b并将w的第i个组件的值设置为v。

当I是一个有限集时，可以使用中缀操作符$\times$来表明合成操作。

例 9.4.5 变量类型的合成

我们来描述两个读/写变量类型$\mathcal{T}_x$和$\mathcal{T}_y$的合成（你可以认为x和y是两个寄存器的名字）。设$\mathcal{T}_x$和$\mathcal{T}_y$的值集分别为V_x和V_y，初始值分别为$v_{0,x}$和$v_{0,y}$。

只有兼容的类型才能被合成。我们通过(文字)下标x或y来区分不同类型的调用和回应。合成类型$\mathcal{T}_x \times \mathcal{T}_y$的值集为$V_x \times V_y$，初始值为$(v_{0,x}, v_{0,y})$；其调用为$read_x$, $read_y$, $write(v)_x$, $v \in V_x$和$write(v)_y$, $v \in V_y$。回应为v_x, $v \in V_x$和v_y, $v \in V_y$，以及ack_x和ack_y。

现在考虑函数f。设$w = (v, v')$是$V_x \times V_y$中的任意元素，则对于w，f为$f(read_x, w) = (v_x, w)$, $f(read_y, w) = (v'_y, w)$, $f(write(v'')_x, w) = (ack_x, (v'', v'))$和$f(write(v'')_y, w) = (ack_y, (v, v''))$。因此，读操作返回向量中所指的组件，而写操作更新所指的组件。

9.5 复杂度衡量

为了衡量异步共享存储器系统的复杂度，设进程步时间的上界为 ℓ，该上界用于证明事件的发生所需时间的上界（例如，进程接收$init_i$输入来产生$decide_i$输出）。

更精确地讲，共享存储器系统的时间复杂度衡量是8.6节中定义的通用I/O自动机的时间复杂度衡量的一个特例。即我们为每个进程的任务C设定一个上界 ℓ，该上界是任务C在两次连续执行步之间的时间的上限。我们用遵循时间上界要求而赋予指定事件π的时间上限来衡量直到π为
250 止的时间；类似地，用能够赋予两个事件的时间之间的间隔的上限来衡量两个事件之间的时间。

注意，我们的时间衡量并不考虑进程间访问共同变量时引起的冲突的开销。在考虑了冲突的多处理器环境中，必须对时间衡量方法做相应的改动。

共享存储器系统其他有意义的衡量还包括静态的衡量（如共享变量的数量及其值集的大小）。

9.6 故障

我们以输入动作$stop_i$来建立共享存储器系统中进程i的停止故障模型，该故障使得进程i的所有任务都失败，但它不会影响其他进程。更精确的说，即使我们不给状态变化以任何限制（除了要求状态变化会导致进程i的所有任务永久失效之外），一个$stop_i$事件只能改变进程i的状态。对于一些问题诸如：进程i之后的输入是被忽略，还是会引起进程i发生如同$stop_i$没有发生时的同样变化，或者还是会引起其他状态变化，等等，都作为公开问题。答案如何并不要紧，因为这些状态变化的效果不会传送给其他进程。

图9-3描述了具有停止故障的异步共享存储器系统的体系结构。

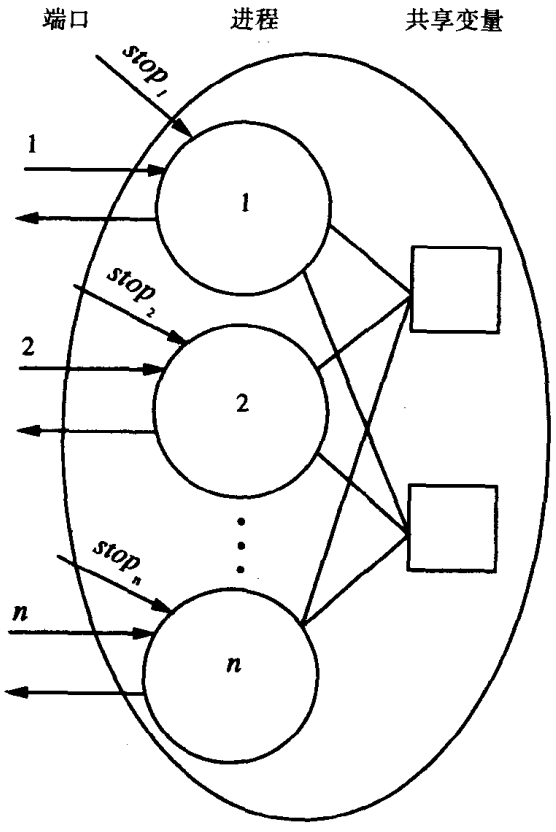


图9-3　具有停止故障的异步共享存储器系统的体系结构

9.7 随机化

在8.8节中的概率I/O自动机通用模型的定义中，当I/O自动机是共享存储器系统时，所得的系统便是概率共享存储器系统。

9.8 参考文献注释

本章描述的基本模型并没有具体的相关文献。它是一种使用I/O自动机结构来进行形式描述的garden-variety共享存储器模型。Lynch和Fischer[216]定义了另一种共享存储器系统模型，其中进程通过对共享变量的瞬时访问而不是通过外部事件来相互通信。Kruskal、Rudolph和 251
Snir [171]定义了在共享存储器多处理器中使用的多种变量类型。

Dwork、Herlihy和Waarts [103] 提出了一种将共享存储器访问冲突考虑在内的时间复杂度衡量方法。概率共享存储器系统的形式定义来源于Lynch，Saias，and Segala [208]的工作。

9.9 习题

9.1 令A是例9.1.1中描述的共享存储器系统

a）证明$fairtraces(A) \subseteq traces(P)$，其中$P$是例9.1.1描述的轨迹属性。

b）定义一种有趣的轨迹安全属性Q，并表明A的（不一定公平）轨迹满足它。即表明 252
$traces(A) \subseteq traces(Q)$。该属性必须提到当同一个进程$i$具有多个$init_i$动作时会发生什么。

9.2 证明$fairtraces(A \times \prod_{1 \le i \le n} U_i) \subseteq traces(Q)$，其中$A$是例9.1.1中描述的共享存储器系统而$Q$是例9.1.2中描述的轨迹属性。

一种证明方法是重新将Q表示为安全属性S和活动属性L的交集。S能够包含一致性条件和有效性条件，还能够包含一致性条件的一部分，即对于所有i，i的动作序列是形如$init_i$, $decide_i$的序列的前缀。对于每个i，L认为有至少有一个$init_i$事件和至少一个$decide_i$事件发生。给出保持S的所有组件，并利用定理8.11来证明$(A \times \prod_{1 \le i \le n} U_i) \subseteq traces(S)$（可以从$traces(A) \subseteq traces(P)$来推出$A$保持$S$）。然后使用公平性假设来证明其活性属性。

9.3 证明以下是例9.2.1中系统$A \times \prod_{1 \le i \le n} U_i$的一个不变式：如果$decision_{U_i} \neq unknown$和$decision_{U_j} \neq unknown$，则$decision_{U_i} = decision_{U_j}$ [⊖]。用两种方法来证明：

a）基于习题9.2中证明的$(A \times \prod_{1 \le i \le n} U_i) \subseteq traces(S)$；

b）使用证明不变式的通用方法——对产生给定的系统状态的运行长度进行归纳来完成。

9.4 例9.4.1中基于读/写寄存器的系统是否满足与例9.1.1中系统相同的轨迹属性P？如果满足，给出证明；否则给出一个反例并证明你所能给出的关于系统行为的最强论断。

9.5 研究问题。使用以下方式来为共享存储器系统定义另一种模型：使用I/O自动机来模拟进程，并为共享变量定义一种新的状态机（类似于变量类型的模型）。定义一种相关的合成操作来将“兼容”的进程和共享变量自动机合成为单个I/O自动机以表示整个系统。需要做出什么样的修改使其合乎以后几章所使用的新定义？ 253

⊖ 我们使用下标来指定属于特定自动机的变量。

第10章 互　斥

本章开始学习异步算法。由于异步算法必须处理由异步性和分布性所带来的不确定性，它与同步算法在总体上有很大的差别。例如，在异步网络中，进程步与信息传送无需在锁－步同步中发生，而是可能以任意顺序发生。

在直接进入异步网络算法的学习之前，我们先学习在异步共享存储器环境中的算法。这主要是因为这种环境相对简单一些。当然，如第17章所述，异步共享存储器模型和异步网络模型之间存在着紧密联系。例如，为异步共享存储器模型所写的算法可以被翻译成能在异步网络模型中运行的算法。在本章及第11章中，我们将不过多地考虑故障问题，因为光是异步性就足够复杂了。

本章讨论的是*互斥*（*mutual exclusion*）问题，即对单个不可分资源（如打印机）的访问进行管理的问题，该资源每次只能支持一个用户。另外，互斥也被视为这样一个问题：程序代码的特定部分在*临界区*中运行，在同一时刻不允许有两个程序在临界区中运行。哪些用户将要求使用资源，以及他们何时将发出要求都是不可知的。这个问题在集中式系统和分布式操作系统中都存在。

首先，我们在Dijkstra提出的早期算法的基础上提出几个针对读/写共享存储器模型的互斥算法，并通过确保不同用户之间的公平性和削弱所用的共享存储器类型来对后来的算法进行提
255 高。随后，我们给出解决问题所需的读/写共享变量的数目的一个基本下限。最后，我们给出针对特定存储器的一系列上限和下限结果，这种共享存储器包含更强大的读－改－写共享变量。

本章篇幅较长，其主要原因在于我们不仅要在这一章中提出一组算法和不可能性结果，而且要介绍在其余几章中会用到的很多思想。其中包括表示共享存储器系统和其环境的技术、证明异步算法的正确性条件(包括安全性、演进性和公平性条件)、对异步算法的证明技术（包括操作式断言、不变式断言和模拟关系证明）、定义和分析异步算法中时间复杂度的方法，以及证明下界的技术。

10.1 异步共享存储器模型

在描述任何算法之前，我们先来描述一下在本章和随后三章中所要用到的计算模型。由于第9章已经对此模型进行了全面、正式的描述，我们在此只做简单、粗略的介绍。

该系统被建模为一组进程和共享变量，两者之间的交互关系如图10-1所示。与同步系统类似，每个进程i都是一种状态机，拥有一个状态集合$states_i$，和一个代表开始状态的$states_i$的一个子集$start_i$。然而，现在进程i具有带标记的*动作*，用来描述它所参与的活动。这些动作被分为*输入动作*、*输出动作*和*内部动作*。在图10-1中，进入和离开进程环的箭头分别表示不同进程的输入和输出动作。我们进一步区分两种不同类型的内部动作：涉及共享存储器的动作和仅仅严格地涉及局部运算的动作。如果一个动作涉及共享存储器，那么我们假定它仅涉及一个共享变量。

与同步系统不同，在异步模型中没有任何消息，因而也没有信息生成函数。进程之间的所有交流都是通过共享存储器来进行的。

整个系统有一个转移关系$trans$，它是一个由三元组(s, π, s')组成的集合，其中s和s'是自动

机状态，也就是所有进程的状态和所有共享变量的值的合成；而π则是一个输入、输出或者内部动作的标记。我们将进程状态和变量值的这些合成称为“自动机状态”，因为在第9章的正式模型中，整个系统都以单个自动机来表示。状态$(s, \pi, s') \in trans$意味着在执行了动作π之后，从自动机状态s到达自动机状态s'是可能的。值得注意的是，*trans*是一种关系，而非一种函数——为简单起见，我们允许模型中包含不决定性。

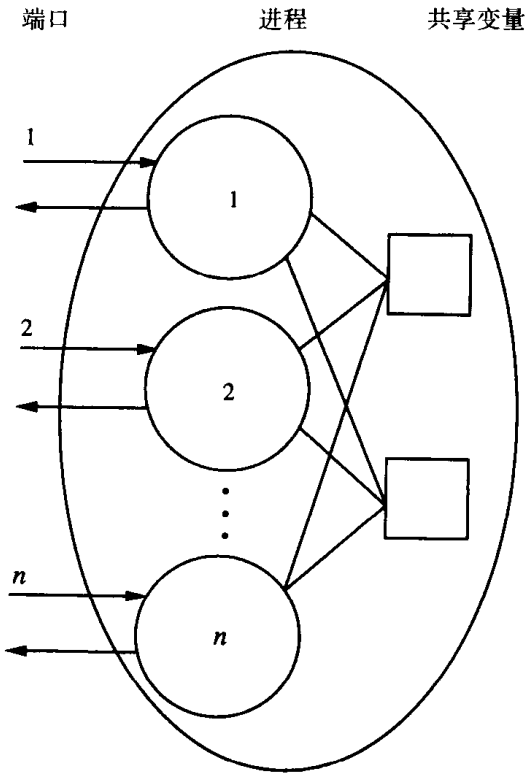


图10-1 一个异步共享存储器系统

我们假定在系统中输入动作始终可以发生，即该系统是输入使能的。正式的，这意味着对于每一个自动机状态s和输入动作π，都存在着一个s'，使得$(s, \pi, s') \in trans$。相反的，输出步和内部步可能只在状态的一个子集中使能。输入使能属性的直观想法是输入动作由任一外部用户控制，而内部动作和输出动作则由系统本身控制。

这些转移具有某些“局部性”限制。首先，如果某个转移并不涉及共享存储器，那么它只涉及执行该动作的进程的状态。相反，如果某个转移涉及进程i和共享变量x，那么它只与进程i的状态和变量x的值相关。我们假定共享存储器动作的使能与否只与进程状态相关，而与被访问的共享变量的值无关。然而，由此带来的进程状态和变量值的变化可能还是与变量值相关的。

通常对共享变量步做进一步的限制，使得运行只包括特定类型的操作，如读操作和写操作。对变量x的读步根据进程以前的状态和x中的值来改变进程的状态，但变量x的值保持不变。写步涉及在一个共享变量上写入一个给定值，变量以前的值将被覆盖。我们将主要考虑以读操作和写操作来访问变量的系统，但同时也会考虑一些功能更为强大的操作，如读－改－写操作。

异步共享存储器系统的运行大大不同于同步系统的运行。在这种模型中，进程以任意顺序而不按照同步轮来执行步。任意顺序是异步模型所必需的。一个运行被形式化为一个自动机状态和自动机动作交替出现的序列：$s_0, \pi_1, s_1, \ldots$，其中每个动作都属于一个特定进程，且连续的（状态、动作、状态）三元组满足转移关系。每一个运行都可能是有限序列或无限序列。

进程步顺序的任意性具有一个重要的例外情况。当一个进程应该执行步的时候，即当这个进程状态中的局部控制动作（即非输入动作）是使能的时候，我们不允许进程停止执行步。（虽然输入动作总是使能的，但我们并不假设它们发生过这种情况）。精确地描述这种情况需要一定的技巧。

例如，我们可能尝试这样的表达：“如果一个进程只执行有限多步，则在它的最终状态中，没有任何局部控制动作是使能的”。但是这并不足够——我们可能也想排除这种情况：一个进程执行了无限多步，但在某一点之后，所有剩下的步都是输入步。我们必须保证进程本身也得到机会来执行局部控制动作。

所以，我们可能尝试这样表达：“如果一个进程只执行有限多步，则在它的最终状态中，

没有任何局部控制动作是使能的；如果一个进程执行许多无限多步，则其中包含无限多的局部控制步。”然而，这又可能不尽正确——考虑这种情况：进程接收到无限多输入，且未执行任
258 何局部控制动作，但事实上没有局部控制动作是使能的。这种情况看起来是合理的，因为如果我们可以说“轮到”这个进程来执行局部控制步，但仅仅是它没有“想”执行的步。

以下定义考虑了所有这些可能性。对于每一个进程i，我们假定以下条件中有一个成立：

1）整个运行是有限的，在最后的状态中，进程i中没有局部控制动作是使能的。

2）整个运行是无限的，要么运行中进程i的局部控制动作发生无限多次；要么运行中有无限多的位置，这些位置中没有使能的局部控制动作。

我们将这种条件称为这个共享存储器系统的公平性条件。（用第8章中的输入/输出自动机定义来解释，这种“公平性条件”将一个进程中所有局部控制的动作都归到同一任务中）。

10.2 问题

互斥涉及到一个单个的、不可分的、不可共享的资源在n个用户（U_1, ..., U_n）之间的分配。可以把这些用户看做应用程序。例如，这种资源可以是一台打印机或其他外设，为了保证输出的正确，这些外设必须排它地访问。这种资源也可能是一个数据库，或是其他数据结构，为了避免不同用户的操作的互相干扰，它们也要求排它地访问。

访问资源的用户处于一个临界区中，即用户状态的一个指定子集。而当用户与资源没有任何关系时，它处于剩余区中。为了进入它的临界区，用户执行一个尝试协议。在使用完资源后，用户执行一个退出协议。这个过程是可以重复的，因此每个用户都遵循一个循环，即从它的剩余区（R）移动到尝试区（T），随后是临界区（C）、退出区（E），最后又到它的剩余区。图10-2显示了这一循环。

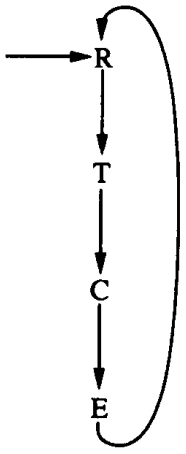


图10-2　单个用户的区域循环

我们考虑如上所述的共享存储器模型中的互斥算法，系统结构参见图10-1。共享存储器系统包含n个进程，编号为1, ..., n, 其中每一个进程都对应于一个用户U_i。进程i的输入包括try_i动作及$exit_i$动作，前者表示用户U_i访问这个资源的要求，后者表示用户U_i宣布完成对资源的操作。进程i的输出是$crit_i$和rem_i，前者表示将资源提供给用户U_i，后者告诉U_i它可以继续余下的
259 工作。try、$crit$、$exit$及rem等动作是共享存储器系统中仅有的外部动作。这些进程负责执行尝试和退出协议。每一个进程i都是用户U_i的“代理”。

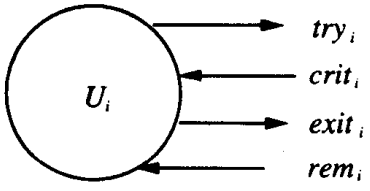


图10-3　用户U_i的外部界面

每一个用户U_i ($1 \leqslant i \leqslant n$)都被描述为一个使用$try_i$、$crit_i$、$exit_i$及$rem_i$动作与它的代理进程进行通信的状态机（正式说法是I/O自动机）。U_i的外部接口（正式说法是外部签名），见图10-3。

我们认为每一个用户U_i都在执行一些应用程序。我们对于用户U_i所作的惟一假设是它遵循循环区域协议，也就是说，在它与它的代理进程之间，用户U_i不会首先违反try_i, $crit_i$, $exit_i$, ...（以try_i开始）的动作循环顺序。正式地，对于一个try_i、$crit_i$、$exit_i$和rem_i动作序列，如果它是循环排序序列try_i, $crit_i$, $exit_i$, rem_i try_i ...的一个前缀，则我们将这个序列定义为对用户i是良构的。
260 然后，我们要求U_i保持由一个序列集合所定义的轨迹属性，其中集合中的序列对用户i是良构

的。(这里我们使用了8.5.4节中轨迹属性和保持的定义。)

在遵循动作循环顺序的U_i的运行中，我们称U_i：

- 起始时处于剩余区，且处在任意rem_i事件和随后的try_i事件之间。
- 处在尝试区，且处于任意try_i事件和随后的$crit_i$事件之间。
- 处在临界区，且处于任意$crit_i$事件和随后$exit_i$事件之间。在这段时间内，U_i可以随意使用资源(虽然我们并不明确地把资源表示出来)
- 处在退出区，且处于任意$exit_i$事件和随后的rem_i事件之间。

图10-4描述了在系统的所有交互。

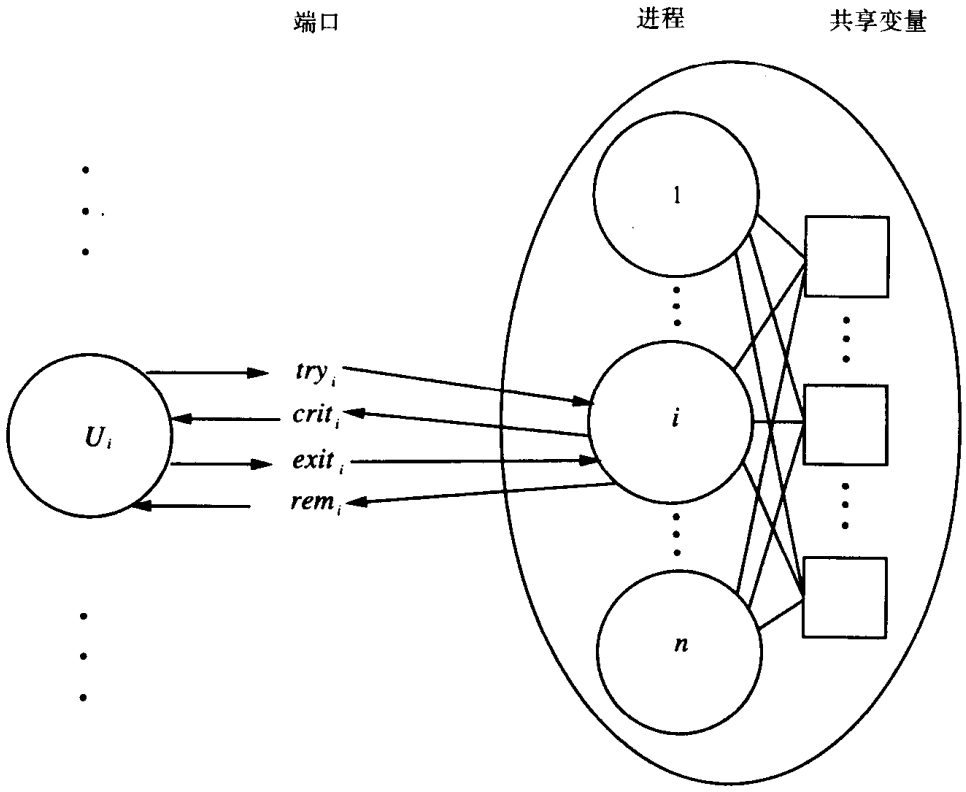


图10-4 互斥问题的各组件之间的交互

现在，我们可以说明共享存储器系统A解决了给定用户集合的互斥问题的含义。也就是说，A与用户的合并(正式说法是合成)必须满足下列条件：

良构性 在任何运行中，对于任意i，描述U_i和A之间交互的子序列对i是良构的。

互斥性 不存在可达系统状态(即A的一个自动机状态和所有U_i的状态的合成)，其中一个以上的用户处于临界区C中。

演进性 在一个公平运行的任一点中。

1)(尝试区的演进)如果至少有一个用户在T中而没有用户在C中，则在其后的某一点中某用户进入C。

2)(退出区的演进)如果至少有一个用户在E中，则在其后的某一点中某用户进入R。

如果共享存储器系统A解决了所有用户集合的互斥问题，则称它解决了互斥问题。

注意我们已经根据用户区域说明了正确性条件。正式的，进程状态也可以根据其区域来分类，且这些区域与用户区域一一对应。故此我们可以等价地根据进程区域来说明正确性条件。在本章的余下部分我们不再区分用户区域和进程区域。

注意演进性条件假定系统的运行是公平的，即假设所有进程（和用户）继续执行步。如果不做此假定，则要求*crit*或*rem*输出最终被执行是不合理的。另一方面，为了需要系统保证良构性和互斥性条件并不要求系统的公平性。区别在于良构性或互斥性条件是*安全属性*（即声称特定“坏”事情从不发生的属性），而演进性条件是*活性属性*（即声称某件“好”事情必将发生的属性）。

轨迹属性　如同8.5.2节定义的，有另外一种根据*轨迹属性*来表示这些正确性条件的等价方法。

262 例如，我们可以定义一个轨迹属性P，其中$sig(P)$以所有*try*、*crit*、*exit*和*rem*动作作为输出，而$trace(P)$是满足以下三个条件的动作序列β的集合：

1）β对于任意i是良构的。

2）β不包含没有*exit*事件夹在中间的两个*crit*事件。

3）在β的任一点中，

a）如果某个进程的最后事件是*try*，而没有进程的最后事件是*crit*，则之后必有一个*crit*事件；

b）如果某个进程的最后事件是*exit*，则之后必有一个*rem*事件。

然后，互斥问题的一个等价描述是这样一个要求：对于A与用户的所有组合B，$fairtrace(B) \subseteq trace(P)$。(注意$B$的外部动作是*try*、*crit*、*exit*和*rem*动作)。轨迹属性P可以分成两部分：一个包含良构性条件和互斥性条件的安全属性，以及一个针对演进性条件的活性属性。

演进的共同责任　根据我们给出的正确性条件，整个系统持续演进的责任不但依赖于协议，还依赖于用户。如果一个用户U_i得到资源（通过一个$crit_i$事件）但不归还它（通过一个$exit_i$事件），则整个系统被迫停止。但如果每个用户最终都归还所得的资源，则演进性条件意味着这个系统继续演进，不停地将进程移至新区域（除非从某点之后所有用户都停留在剩余区中）。

锁定　我们声明的演进性条件并不意味着任一特定用户曾经成功地到达其临界区。相反，演进性是一个“全局”概念，只声称某个用户到达其临界区。例如，如下情形并不违反演进性条件：从一个初始状态开始，用户U_1进入T。然后用户U_2在其四个区域中循环无数多次，而U_1留在T中，且剩下的进程仍然在R中。我们的演进性条件并不保证U_1曾经到达C。

限制进程的活动　在本章中我们还假设存在一个其他限制，这个限制是技术上的：对于共享存

263 储器系统中的一个进程，只有它的用户处于尝试区或退出区中时，它才能够拥有一个使能的局部动作。这是说一个进程收到具有活动的请求时才能运行协议。这与每个用户只是其相应用户的一个代理这一观点相一致。

在实际环境中，这种假设可能是合理的，也可能是不合理的。互斥问题最先在分时单处理器的环境中被研究，其中用户是对一个单处理器进行共享的逻辑独立进程。在这种环境中，如果允许一个永久进程去管理对资源的访问，那将在管理者进程和用户进程之间引起额外的上下文切换。在真正的多处理器环境中，可以使用一个专用的处理器来管理资源，从而有可能避免上下文切换。然而，通常有很多资源需要管理，且专用于资源管理的处理器将无法参与其他计算任务。

读/写共享变量　在本章的绝大部分（除了10.9节）中我们都假设共享变量是读/写变量，即*寄存器*。在一步中，进程可以读或者写单个共享变量，但不能同时进行读和写。因此，涉及进程i和寄存器x的两个动作是：

1）（读）进程i读寄存器x，并使用读回的值来改变进程i的状态。

2）（写）进程i将进程i的状态决定的一个值写到寄存器x中。

我们以一条简单的引理来结束本节，该引理声称当进程处于尝试区或退出区时，它们不能够停止执行步。

引理10.1 令A是一个（对于所有用户集合）解决了互斥问题的算法。令$U_1, ..., U_n$是特定的一组用户，而B是A与给定的一组用户的组合。令s是B的一个可达状态。

如果进程i在状态s中处于尝试区或退出区，则进程i的某个局部控制动作在s中是使能的。

证明 不失一般性，我们假设所有用户$U_1, ..., U_n$都总是归还资源。

令α是结尾为s的B的有限运行，用反证法，假设在状态s中，进程i要么在尝试区中，要么 264
在退出区中，且没有进程i的局部控制动作在s中使能。则在作为α的扩展的B的任一运行中，在前缀α之后，涉及i的事件不再发生。这是因为局部扩展动作的使能与否只决定于局部进程状态，且当进程i处于T或E中时，良构性条件阻止了对进程i的输入。

现在令α'是对α进行扩展的B的一个公平运行，其中在前缀α之后没有*try*事件发生。反复使用演进性条件和用户总是返回资源的事实，可知进程i最终必然执行一个$crit_i$动作或者一个rem_i动作。但这与α'不包含i的进一步动作的事实相矛盾。 □

10.3 Dijkstra的互斥算法

第一个适用于异步读/写共享存储器模型的互斥算法是由Edsger Dijkstra于1965年提出的，该算法建立在Dekker于早先提出的双进程解法之上。这个算法并不是到目前为止最优美、最有效的算法，也不能满足最强条件。然而，由于以下几个原因，我们给出这个算法。首先，它是可以归为“分布式”算法的最早的例子。其次，此算法中包含几个有趣的算法思想。再次，对于演示异步存储器算法的基本推理技巧来说，此算法是一个极好的例子。

10.3.1 算法

与Dijkstra的原文一样，我们开始时以传统的“伪码”风格来给出这个算法的代码。虽然这个代码是可行的，但在它如何被翻译成我们模型的一个实例这一点上，也许并不是非常清楚。我们将此算法称为*DijkstraME*算法。 265

***DijkstraME* algorithm:**

Shared variables:
$turn \in \{1, \ldots, n\}$, initially arbitrary, writable and readable by all processes
for every i, $1 \leqslant i \leqslant n$:
  $flag(i) \in \{0, 1, 2\}$, initially 0, writable by process i and readable by all processes
Process i:

```
    ** Remainder region **

    try_i
L:  flag(i) := 1
    while turn ≠ i do
        if flag(turn) = 0 then turn := i
    flag(i) := 2
    for j ≠ i do
        if flag(j) = 2 then goto L
    crit_i

    ** Critical region **

    exit_i
    flag(i) := 0
    rem_i
```

共享变量为 *turn*, 它是一个从{1,...,*n*}中取值的整数，以及*flag*(*i*), 1≤*i*≤*n*, 每个进程都有一个*flag*(*i*)变量，每个变量都从{0,1,2}中取值，初始为0。变量*turn*是一个*多写者/多读者寄存器*，可被所有进程读写。每个*flag*(*i*)都是一个*单写者/多读者寄存器*，只能由进程*i*写入，但可被所有进程读。

在进程*i*的第一步，它首先将*flag*设为1，然后重复检查变量*turn*，看变量*turn*是否等于*i*。如变量*turn*不等于*i*，并且*turn*的当前拥有者看起来并不活动，则进程*i*将*turn*设为*turn*:=*i*。一旦看到*turn* = *i*, 进程*i*便前进到第二步。

在第二步中，进程*i*再次设定*flag*，将它设为2，然后检测是否没有其他变量的*flag* = 2。这种对其他进程的*flag*的检测可以以任何顺序进行。如果检测顺利结束，进程*i*进入临界区；否则，它将重新回到第一步。在离开临界区时，进程*i*将它的*flag* 重新设为0。

在我们对*DijkstraME* 进行任何证明之前，我们需要将之理解为我们的正式状态机器模型的一个实例。如何将代码翻译为一个自动机这一点并不明显。

首先，你可以想到，每个进程的状态都应该包括其局部变量的值，另外还有加上一些在代码中没有明确写出的其他信息，包括：

266

- 用以记住刚从共享变量中读出的值的临时变量。
- 一个程序计数器, 用于说明进程处于代码的什么位置。
- 由程序的控制流引入的临时变量（例如，for 循环可以引入一个集合变量来跟踪已经被成功检测过的进程的索引）。
- 一个区域指定，用于指定进程处于区域*R*、*T*、*C*或*E*（*R*指剩余区，*T*指从try_i事件直到下一个$crit_i$事件之间的代码部分，*C*指临界区，而*E*指$exit_i$事件直到下一个rem_i事件之间的代码部分）中的一个。

每个进程的惟一开始状态必然由局部变量的指定初值、临时变量的任意值、程序计数器和表明处于剩余区的区域指定组成。每个共享变量的初值如代码所示。

自动机的步依照代码来执行；然而，自动机必须处理代码中某些含糊之处。尽管代码描述了局部变量和共享变量的变化情况，但它没有明确描述隐含变量（临时变量、程序计数器和区域指派）所发生的情况。例如，当一个try_i动作发生时，*i*的程序计数器将会移到代码中的语句*L*，且*i*的区域指定将变为*T*。这些改变在自动机中必须显式地描述。

代码也没有明确指出不可分割步由代码的哪些部分组成。然而，这一点对于仔细地对算法进行推理是重要的。对于*DijkstraME*算法，不可分割步由处于用户界面的*try*、*crit*、*exit*和*rem*步、对共享变量的各自读写以及某些局部计算步组成。这里有个巧妙之处：对*flag*(*turn*) = 0的测试并不需要两次独立的读操作——由于*turn*刚刚在前一行中被读过，故可以使用*turn*的一个局部副本。

我们采用第8章中的*前提－结果风格*，对*DijkstraME*算法进行手工重写以克服这些含糊性。这种重写使得代码大大变长，但所有的转移都被明确地描述出来。鉴于可读性的考虑，我们将不同动作的代码大体上按照它们的执行顺序来排放；但是，在形式化模型中这种顺序并无意义——每个动作只要在使能时都可以发生。区域指定*R*、*T*、*C*和*E*被编入程序计数器的值中：*R*对应*rem*；*T*对应*set-flag*-1, *test-turn*, *test-flag*, *set-turn*, *set-flag*-2, *check*和*leave-try*；*C*对应*crit*；

267

而*E*则对应*reset*和*leave-exit*。注意每个代码段都是不可分割地执行。

DijkstraME algorithm (rewritten):

Shared variables:
$turn \in \{1, \ldots, n\}$, initially arbitrary
for every i, $1 \leq i \leq n$:
  $flag(i) \in \{0, 1, 2\}$, initially 0

Actions of i:

Input:	Internal:
try_i	*set-flag-1*$_i$
$exit_i$	*test-turn*$_i$
Output:	*test-flag*$(j)_i$, $1 \leq j \leq n$, $j \neq i$
$crit_i$	*set-turn*$_i$
rem_i	*set-flag-2*$_i$
	check$(j)_i$, $1 \leq j \leq n$, $j \neq i$
	reset$_i$

States of i:
$pc \in \{rem, set\text{-}flag\text{-}1, test\text{-}turn, test\text{-}flag(j), set\text{-}turn, set\text{-}flag\text{-}2, check, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially *rem*
S, a set of process indices, initially $\emptyset$

Transitions of i:

try$_i$
  Effect:
    $pc := set\text{-}flag\text{-}1$

set-flag-1$_i$
  Precondition:
    $pc = set\text{-}flag\text{-}1$
  Effect:
    $flag(i) := 1$
    $pc := test\text{-}turn$

test-turn$_i$
  Precondition:
    $pc = test\text{-}turn$
  Effect:
    if $turn = i$ then $pc := set\text{-}flag\text{-}2$
    else $pc := test\text{-}flag(turn)$

test-flag$(j)_i$
  Precondition:
    $pc = test\text{-}flag(j)$
  Effect:
    if $flag(j) = 0$ then $pc := set\text{-}turn$
    else $pc := test\text{-}turn$

crit$_i$
  Precondition:
    $pc = leave\text{-}try$
  Effect:
    $pc := crit$

set-turn$_i$
  Precondition:
    $pc = set\text{-}turn$
  Effect:
    $turn := i$
    $pc := set\text{-}flag\text{-}2$

set-flag-2$_i$
  Precondition:
    $pc = set\text{-}flag\text{-}2$
  Effect:
    $flag(i) := 2$
    $S := \{i\}$
    $pc := check$

check$(j)_i$
  Precondition:
    $pc = check$
    $j \notin S$
  Effect:
    if $flag(j) = 2$ then
      $S := \emptyset$
      $pc := set\text{-}flag\text{-}1$
    else
      $S := S \cup \{j\}$
      if $|S| = n$ then $pc := leave\text{-}try$

reset$_i$
  Precondition:
    $pc = reset$
  Effect:
    $flag(i) := 0$

```
                                        S := ∅
exit_i                                  pc := leave-exit
    Effect:
        pc := reset                 rem_i
                                        Precondition:
                                            pc = leave-exit
                                        Effect:
                                            pc := rem
```

转移代码大部分都是清晰易明的。注意这种新的风格易于用来表示轻微的改进；例如，动作*set-turn*$_i$可以允许进程*i*直接到达第二阶段，而无需重新测试*turn*的值。

10.3.2 正确性证明

在本节中我们粗略地给出一个对*DijkstraME*算法的正确性证明。这种方法是一种蛮力操作式证明，即由关于运行的特别论证组成的证明。在接下来的10.3.3节中，我们使用不变式断言来给出另外一个更流行的互斥条件证明。

我们来给出三条表明*DijkstraME*算法满足所求的引理。

引理10.2 *DijkstraME*算法保证了对于任意用户的良构性。

更确切地说，在*DijkstraME*算法与任意一组用户的组合（组成）的任一运行中，对任一用户U_i和*DijkstraME*算法之间的交互进行描述的子序列对于用户*i*是良构的。

证明 通过观察代码，容易证实*DijkstraME*算法对于任一用户都保证了良构性，因此，由假设可知，用户也保留了良构性。定理8.11意味着系统只产生良构的序列。 □

引理10.3 *DijkstraME*算法满足互斥性。

更确切地说，在*DijkstraME*算法与任意一组用户组合的任一运行中，不存在一个可达状态，其中一个以上的用户处于临界区*C*内。

269 **证明** 采用反证法。假设在某个可达状态中U_i和U_j, $i \neq j$同时处于临界区*C*中。考虑一个导致该状态的运行。根据代码，进程*i*和进程*j*都在进入临界区前执行了*set-flag*-2步。考虑每个进程的最后*set-flag*-2步，为了保持一般性，可设*set-flag*-2$_i$先执行。则从这点起直至进程*i*离开*C*为止（根据两个进程同时处于*C*的假设，必在进程*j*进入*C*之后进程*i*才能离开*C*）都有*flag*(*i*) = 2，所以从*set-flag*-2$_j$事件直至进程*i*进入*C*这一事件段内，恒有*flag*(*i*) = 2。参见图10-5。但是，在这个时间段内，进程*j*必然测试*flag*(*i*)的值并发现它不等于2，矛盾。 □

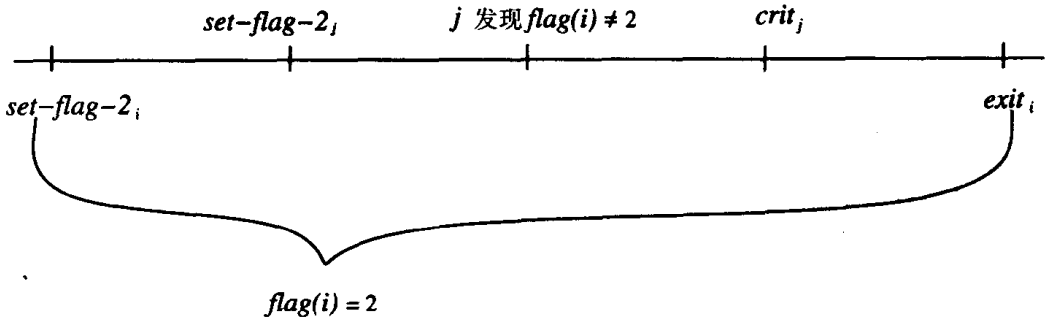


图10-5 引理10.3的证明中的事件顺序

引理10.4 *DijkstraME*算法保证演进性。

证明 对于退出区的证明很容易：在一个公平运行的任一点中，如果U_i处于退出区中，则进程i继续执行步。在最多执行额外两步后，进程i将执行一个rem_i动作，将U_i送入剩余区。

现在来考虑尝试区的演进。采用反证法。假设α是一个公平运行，在其到达的一点中，最少一个用户处于T中且没有用户处于C中，并假设在该点之后，没有用户会进入C。

我们从去除某些复杂性开始。首先，E中的每个进程都不断执行步，在最多两步后，这些进程必然到达R。故此，在α的此后某点中，每个进程都在T或R中。其次，由于系统中只有有限多个进程，α后的某点，没有进程进入T。因此，在a此后的某点，每个进程或在T中或在R中，且没有进程会再次改变所在区域。这意味着α有一个后缀α_1，其中存在一个由处于T中的进程组成的非空集合，这些进程永远不停地执行步，且没有区域变化发生。把这些进程称为竞争者。 270

在α_1的最多一步之后，每个竞争者i都保证$flag(i) \geqslant 1$，且在α_1的余下部分中都保持$\geqslant 1$。故此，不失一般性，我们可以假设在整个α_1中，所有竞争者都保持$flag(i) \geqslant 1$。

显然，如果在α_1期间*turn*被修改过，则它将等于某一竞争者的索引。另外，我们拥有以下声明。□

声明10.5 在α_1中，*turn*最终等于某个竞争者的索引。

证明 假设在整个在α_1中，*turn*的值始终等于某个非竞争者的索引。考虑任一竞争者i。

如果pc_i曾经到达*test-turn*（即在原来代码中while循环的开始），则我们来证明i将会把*turn*设为i。这是因为i最先执行$test\text{-}turn_i$并发现*turn*等于某个$j \neq i$。然后它执行$test\text{-}flag(j)_i$并发现$flag(j) = 0$，这是因为j是一个非竞争者。进程i因此执行$set\text{-}turn_i$，将*turn*设为i。

现在来证明i将到达*test-turn*。i不能到达*test-turn*的惟一可能是i成功地测试出其他进程的*flags*都不为2（在原来代码的第二阶段），故i前进到*leave-try*。但是根据对α_1的假设，我们知道i没有到达C。所以一定有某些测试失败，使得i回到*set-flag*-1并前进到*test-turn*。

故此，i到达*test-turn*且设置$turn := i$。由于i是一个竞争者，这就是所需的矛盾。 □

一旦*turn*被设置成一个竞争者的索引，其后它必将等于某个竞争者的索引，尽管*turn*的值可能变成不同竞争者的索引。（这是因为有可能几个进程同时进行*set-turn*）然后任意之后的*test-turn*和*test-flag*会导致$flag(turn) \geqslant 1$，这是因为所有竞争者i满足$flag(i) \geqslant 1$。因此，作为这些测试的结果，*turn*不会改变。最终*turn*稳定成为一个最终（竞争者的）索引。令α_2是α_1的一个后缀，其中*turn*的值稳定为某个竞争者的索引，比如i。

下一步我们声明在α_2中，对于任意不等于i的竞争者j，其程序计数器最终在*test-turn*和*test-flag*之间死循环。（即在while循环中永久循环）。原因在于如果它曾经到达*check*（在第二阶段），由于它不到达C，则它最终必会回到*set-flag*-1。之后它处于死循环中，因为在整个α_2中$turn = i \neq j$且$flag(i) \neq 0$。令α_3是α_2的一个后缀，其中除了i之外的所有竞争者都在*test-turn*和*test-flag*之间循环。这意味着在整个α_3中除了i之外的所有竞争者的*flag*变量都等于1。 271

在α_3中，没有任意障碍能够阻止进程i（其索引在*turn*中）到达C。例如，如果i执行*test-turn*，则i发现$turn = i$并前进到*set-flag*-2。然后，由于没有进程的*flag* =2，进程i第二阶段的检查成功并进入C。

图10-6描述了在证明中出现的连续后缀。

定理10.6 *DijkstraME*算法解决了互斥问题。

尽管上述的证明是正确的，它们相当复杂和特别。采用更系统化的证明将会更好。在下一节中，我们使用不变式来给出另一个对互斥条件的证明。可以使用时序逻辑来更系统化地证明演进性条件，但在本书中我们不这样做。

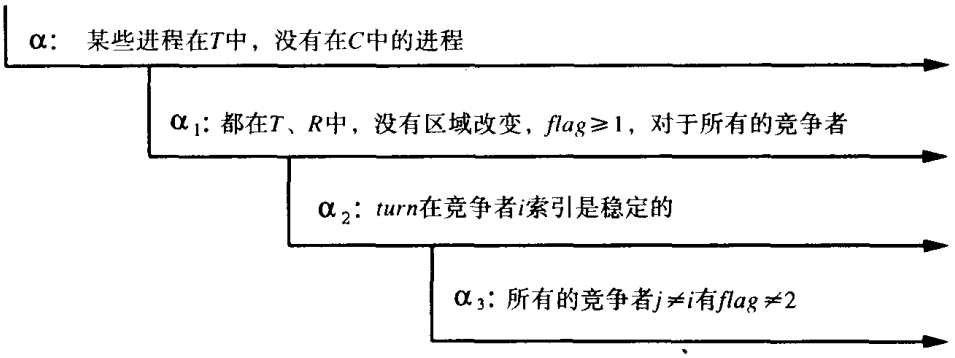


图10-6　引理10.4的证明中的连续后缀

10.3.3　互斥条件的一个断言证明

在同步网络模型中，许多最整洁、最系统化的证明是基于关于某轮数之后系统状态的不变式断言的。异步系统中不存在轮的概念，但不变式依然适用。该方法以更细粒度来应用，证实关于任意独立进程步数之后系统状态的声明。当然，对任意步数之后异步系统状态的声明一般
272 比对任意轮数之后同步系统状态的声明要难，证明起来也更难。但是鉴于不变式的优点，这种努力还是值得的。不变式断言是对异步算法的正确性进行推理的单独的最重要的形式化工具。

现在来给出*DijkstraME*算法的互斥条件的不变式断言证明。

（引理 10.3的）证明　为了证明互斥性，我们必须展示

断言10.3.1　在任意的可达系统状态中[⊖]，$|\{i : pc_i = crit\}| \leq 1$。

可以对运行的步数的归纳来证明该断言。但是已给出的声明并不强大到足以单独地证明出所需的结论，故我们需要某些辅助不变式。断言10.3.1是以下两个不变式的推论。

断言10.3.2　在任意的可达系统状态中，如果$pc_i \in \{leave\text{-}try, crit, reset\}$，则$|S_i| = n$。

断言10.3.3　在任意的可达系统状态中，不存在i和j, $i \neq j$，使得$i \in S_j$和$j \in S_i$。

如果断言10.3.2和断言10.3.3为真，则断言10.3.1立刻可得：采用反证法，假设在某个可达系统状态中，存在两个不同进程i和j，使得$pc_i = pc_j = crit$。则根据断言10.3.2，$|S_i| = |S_j| = n$。则有$i \in S_j$和$j \in S_i$，但这与断言10.3.3矛盾。

可以对运行的长度做归纳来轻松证明断言10.3.2。基础步是显然的，因为在系统开始状态中所有进程都处于R中。归纳步分情况讨论，需要考虑同时发生的所有动作类型。此时，惟一能够违反断言的步是那些使得pc_i进入所示值集合的步和那些将S_i重置为ϕ的步，即$check_i$和$reset_i$。对于$check_i$，在该步之后条件$pc_i \in \{leave\text{-}try, crit, reset\}$为真的惟一可能是$|S_i| = n$，得证。对于$reset_i$，进程在该步之后离开所示的值集合，结论成立。

剩下的是要证明断言10.3.3，这需要两个事实。第一个事实限制了在$S_i = \phi$时进程i可在代码的什么位置。

273 **断言10.3.4**　在任意的可达系统状态中，如果$S_i = \phi$，则$pc_i \in \{check, leave\text{-}try, crit, reset\}$。

也可以对运行的长度做归纳来证明该断言。基础步是显然的，因为在系统开始状态中$S_i = \phi$。在归纳步中，能够导致对断言的违反的惟一事件是那些使得S_i变成不等于ϕ的事件和那些使得pc_i离开所示值集合的事件，即$set\text{-}flag\text{-}2_i$、$check_i$和$reset_i$。但是$set\text{-}flag\text{-}2_i$设置$pc_i := check$。还

⊖　系统状态是用户状态、进程状态和共享变量值的组合。

有，当$check_i$导致pc_i离开所示值集合时，它也设置$S_i := \phi$。最后，*reset*设置$S_i := \phi$。因此，所有事件都保持了对条件的满足。

第二个事实说明当进程i处于代码的特定位置时，$flag(i) = 2$。

断言10.3.5 在任意的可达系统状态中，如果$pc_i \in \{check, leave\text{-}try, crit, reset\}$，则$flag(i) = 2$。

同样可以对运行的长度做归纳来证明该断言。综合这两个事实，可得：

断言10.3.6 在任意的可达系统状态中，如果$S_i \neq \phi$，则$flag(i) = 2$。

现在通过对运行长度的归纳来证明断言10.3.3。基础步是显然的，因为在系统开始状态中所有S_i集合都为空。在归纳步中，能够导致对断言的违反的惟一事件是对于某个i和某个$j(i \neq j)$，将元素j加到S_i的事件，即对于某个i和某个j的$check(j)_i$事件。考虑$check(j)_i$事件之后j被加到S_i中的情况，这个事件发生时必有$flag(j) \neq 2$。但是断言10.3.6意味着$S_j = \phi$，故$i \notin S_j$。因此，该步不能导致对断言的违反。 □

10.3.4 运行时间

在本节中，我们证明在运行中，从某个进程处于T中但没有进程处于C中的一点直到某个进程进入C的一点之间的时间的上界。

要证明这样的上界，我们面对的第一个困难是我们并不清楚这种“时间”是什么意思——与同步环境不同，这里没有轮数可以计算。相反，我们只是假设运行从实际时间0开始，每一步在某个实际时间发生。我们为每个进程的连续步（当这些步是使能的时候）之间的时间赋予一个上界1；试回想起一个动作的所有前提-后果代码被视为一步。我们还假设任意用户花在临界区中的时间有一个上界c。根据这些假定的上界，我们可以推导出要使我们感兴趣的活动发生所需时间的上界。 274

定理10.7 在*DijkstraME*算法中，假设在一个特定时间中某个用户处于T中且没有用户处于C中，则在时间$O(\ell n)$之内，某个用户进入C。

大O涉及的常数与ℓ，c和n无关。这个证明是特别的，需要一些技巧，可以借用演进性条件证明中的思想。

证明 假设引理为假，考虑一个运行，在该运行的某点中，进程i处于T中但没有进程处于C中，而且在最少$k\ell n$时间内没有进程进入C中，这里k是一个特定的大常数。在以下的分析中，选出的常数k比大O中的常数要大得多。

第一，容易看出，从分析的起始点直至没有进程在C或E中，流逝的时间最多为$O(\ell)$。

第二，我们来证明直到进程i执行$test\text{-}turn_i$为止的额外时间最多为$O(\ell n)$。这是因为进程i在返回*set-flag*-1之前，在第二阶段最多花费这么多时间来测试*flag*。我们知道它必然回到*set-flag*-1，否则它进入C，而我们并不假设它这么快便进入C。

第三，我们来证明从进程i执行$test\text{-}turn_i$到*turn*的值变成某个竞争者的索引的额外时间为$O(\ell)$。我们需要分情况讨论。如果在进程i执行$test\text{-}turn_i$的时候，*turn*已经拥有某个竞争者的索引，则证明完成。所以我们假设这一点不成立，即假设$turn = j$，其中j是一个非竞争者的索引。在该测试之后的$O(\ell)$时间之内，i执行$test\text{-}flag(j)_i$。如果进程i发现$flag(j) = 0$，则i将*turn*设为i，由于i是竞争者的索引，故得证。如果它发现$flag(j) \neq 0$，则在$test\text{-}turn_i$和$test\text{-}flag(j)_i$之间，进程j进入尝试区并变成一个竞争者。如果在这个间隔之内*turn*的值没有发生改变，则*turn*的值等于竞争者(j)的索引，得证。如果在这个间隔之内*turn*的值发生了改变，则它必然被设置成为某个竞争者的索引，得证。

第四，在额外的$O(\ell)$时间之后，必存在某点，其中*turn*的值稳定为某个特定竞争者的索引，比如j，然后没有进程能够再次前进到*set-turn*或*set-flag*-2（最少直到从分析的起始点之后的$k\ell n$时间内）。

第五，然后在额外$O(\ell n)$时间之内，除了j之外的所有竞争者的程序计数器都在{*test-turn*, *test-flag*}中。否则它们到达C，但我们不希望这种情况会很快发生。

第六，在额外$O(\ell n)$时间之内，j必定成功进入C。这与在这些时间内没有进程进入C的假设矛盾。

275 证明中的事件顺序和事件之间的时间界限参见图10-7。

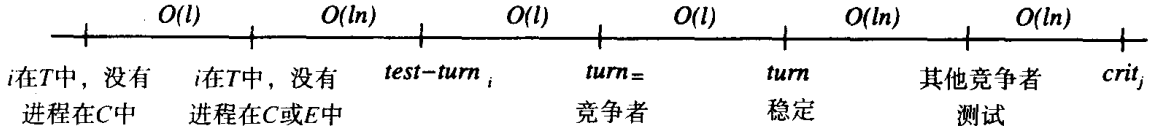


图10-7　定理10.7的证明中的事件顺序和时间界限

10.4　互斥算法的更强条件

尽管*DijkstraME*算法保证了互斥性和演进性，它不能保证另外一些我们期待的条件。它不能保证临界区被公平地分给不同用户；例如，它允许一个用户反复地得到授权来访问其临界区，而其他用户永远无法得到访问的机会。这种情况有时称为锁定（*lockout*）或者饿死（*starvation*）。

注意这里讨论的公平性与之前讨论的不同。到目前为止，我们已经讨论了进程步（和用户自动机步）的公平执行，但现在我们要讨论对资源的公平分配。为了区分这两种公平性，我们把进程步和用户自动机步的公平执行称为*低层次公平*，把资源的公平分配称为*高层次公平*。实际上，高层次公平可能并不重要；在许多使用了互斥的实际情形中，用户间的冲突并不常见，一个用户可以一直等到所有冲突用户都得到机会。高层次公平考虑的重要性依赖于资源冲突的程度和各个用户程序的敏感程度。

Dijkstra算法还有一个不怎么吸引人的性质——它使用了*共享的多写者/多读者寄存器*（*turn*）。在许多类型的多处理器系统（和绝大部分信息传送系统中）中实现这种变量是困难和昂贵的。最好设计出只使用*单写者/多读者寄存器*或甚至*单写者/单读者寄存器*的算法。

现在有许多互斥算法以不同方法来改进*DijkstraME*算法。本章的余下部分对这些算法进行概览。

276 在进入对这些算法的介绍之前，我们先详细定义互斥算法保证高层次公平的意思。根据算法所在的上下文，可能会用到不同的高层次公平概念。我们来定义三种概念，每种概念都是针对特定的互斥算法A与特定用户$U_1, \ldots, U_n$的集合的组合来声明的。

锁定权　在任意的低层次公平运行中，以下成立：

1）（尝试区的锁定权）如果所有用户都总是归还资源，则任意到达T的用户最终都进入C。

2）（退出区的锁定权）任意到达E的用户最终都进入R。

如同基本的良构性条件、互斥性条件和演进性条件一样，锁定权条件也可以使用轨迹属性来表达。

时间界限b　在任意带时间信息的低层次公平运行中，以下成立：

1）（尝试区的时间界限b）如果所有用户都总是在得到资源之后的时间c内归还资源，且每

个在T或E中进程的连续步之间的时间间隔最多为ℓ，则任意到达T的用户最终在时间b内进入C。

2）（退出区的时间界限b）如果每个在T或E中进程的连续步之间的时间间隔最多为ℓ，则任意到达E的用户最终在时间b内进入R。

（注意，这里b的值是ℓ和c的函数。）

绕过次数a 考虑一个运行中的任一时间间隔，该间隔从进程i在T中执行一个局部控制步开始，其间进程i一直留在T中。在这个时间间隔内，任意其他用户j，$j \neq i$进入C的次数最多为a。

在前两种情况中，退出区的高层次公平条件与尝试区的类似。然而，在大多数算法中，退出区是相当麻烦的。

如果算法A对所有用户集合都保证锁定权，则称它为具有*锁定权*。同样，我们扩展其他高层次公平定义。在这些公平性条件中有一些简单的含义：

277

定理10.8 令A为一个互斥算法，令$U_1,\ldots,U_n$为一个用户集合，令B为A与$U_1,\ldots,U_n$的合成。如果B具有任一有限的绕过次数界限，且对于退出区是具有锁定权，则B是具有锁定权。

证明 考虑B的一个低层次公平运行，其中所有用户总是归还资源，且假设在运行的某一点，i处于T中。采用反证法，假设i从不进入C。

引理10.1意味着最终i必定在尝试区中执行局部控制动作。反复使用演进性条件和用户总是归还资源的假设，可知共有无限多的区域发生变化。则当i处于C中时，除了i之外的某个进程进入C无限多次，这违反了绕过次数的界限。 □

定理10.9 令A为一个互斥算法，令$U_1,\ldots,U_n$为一个用户集合，令B为A与$U_1,\ldots,U_n$的合成。如果B具有任一时间界限b（即对于尝试区也对于退出区），则B是具有锁定权。

证明 考虑B的一个低层次公平运行，其中所有用户总是归还资源，且假设在运行的某一点，i处于T中。

为运行中的事件分配单调非降低的无界时间值，使得每个进程步的时间最多为ℓ，而所有针对临界区的时间最多为c。

由于算法满足时间界限b，在最多时间b内i进入C，特别的，i最终进入C，从而满足锁定权条件。 □

在以下节中，我们将看到某些协议满足这些更强高层次公平条件中的其中一些条件。

10.5 锁定权互斥算法

我们给出的第一个改进是由Peterson提出的三个算法，这三个算法都保证了无锁定。第一个算法只针对两个进程，但是它演示了大多数基本思想。这个算法可以以两种方法来扩充到$n > 2$的情况：第一种在$n-1$次竞争中使用双进程算法；第二种在选出胜者的锦标赛中使用双进程算法。

10.5.1 双进程算法

我们从双进程的解法开始，该解法被称为*Peterson2P算法*。通常，双进程系统中的两个进程被命名为进程1和进程2。这一次，我们将进程对2取模，进程2与进程0是一样的。所以我们把两个进程分别称为进程0和进程1。如果$i \in \{0, 1\}$，则令$\bar{i} = 1 - i$，即另一进程的索引。传统风格的代码如下。

278

Peterson2P algorithm:

Shared variables:
$turn \in \{0,1\}$, initially arbitrary, writable and readable by all processes
for every $i \in \{0,1\}$:
  $flag(i) \in \{0,1\}$, initially 0, writable by i and readable by $\bar{i}$

Process i:

  ** Remainder region **

  try_i
  $flag(i) := 1$
  $turn := i$
  waitfor $flag(\bar{i}) = 0$ or $turn \neq i$
  $crit_i$

  ** Critical region **

  $exit_i$
  $flag(i) := 0$
  rem_i

在*Peterson2P*算法中，进程i开始时把其*flag*设置为1，这与*DijkstraME*算法中的进程所做的一样。但是在这一次中，进程i立刻前进到语句$turn := i$。然后它等到另一进程的*flag*变为0或者$turn \neq i$。也就是说，要么另一进程目前根本不参与竞争，要么自从进程i最近对变量*turn*的值进行设置之后，另一进程又设置了*turn*的值。然后（有点奇怪地），如果变量*turn*的值等于另一进程的索引，则i进入临界区。

如何将这个程序翻译成为形式化模型中的状态机呢？与前面一样，我们需要引入程序计数器、临时变量和区域指定。代码中需要解决的含糊性之一在于进程在waitfor语句中是先检查*flag*变量还是先检查*turn*变量。为了保证正确性，两种检查都必须反复地进行；为了保证简单
279 性，我们假设两个检查交替地进行，尽管采用更松的假设也没有问题。

为了令证明更为容易，我们采用前提- 结果风格来重写这个算法。在这里，R对应*rem*；T对应*set-flag*, *set-turn*, *check-flag*, *check-turn*和*leave-try*；C对应*crit*；而E对应*reset*和*leave-exit*。

Peterson2P algorithm (rewritten):

Shared variables:
$turn \in \{0,1\}$, initially arbitrary
for every $i \in \{0,1\}$:
  $flag(i) \in \{0,1\}$, initially 0

Actions of i:

Input:	Internal:
try_i	*set-flag*$_i$
$exit_i$	*set-turn*$_i$
Output:	*check-flag*$_i$
$crit_i$	*check-turn*$_i$
rem_i	*reset*$_i$

States of i:

$pc \in \{rem, set\text{-}flag, set\text{-}turn, check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset, leave\text{-}exit\}$,
initially *rem*

Transitions of i:

try$_i$
Effect:
$pc := set\text{-}flag$

set-flag$_i$
Precondition:
$pc := set\text{-}flag$
Effect:
$flag(i) := 1$
$pc := set\text{-}turn$

set-turn$_i$
Precondition:
$pc = set\text{-}turn$
Effect:
$turn := i$
$pc := check\text{-}flag$

crit$_i$
Precondition:
$pc = leave\text{-}try$
Effect:
$pc := crit$

exit$_i$
Effect:
$pc := reset$

check-flag$_i$
Precondition:
$pc = check\text{-}flag$
Effect:
if $flag(\bar{i}) = 0$ then
$pc := leave\text{-}try$
else
$pc := check\text{-}turn$

check-turn$_i$
Precondition:
$pc = check\text{-}turn$
Effect:
if $turn \neq i$ then
$pc := leave\text{-}try$
else
$pc := check\text{-}flag$

reset$_i$ 280
Precondition:
$pc = reset$
Effect:
$flag(i) := 0$
$pc := leave\text{-}exit$

rem$_i$
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

引理10.10 *Peterson2P*算法满足互斥性。

证明 我们采用基于不变式断言的证明。很容易通过归纳法来证明以下断言

断言10.5.1 在任意的可达系统状态中，如果$flag(i) = 0$，则$pc_i \in \{leave\text{-}exit, rem, set\text{-}flag\}$。

使用断言10.5.1，可以归纳法来证明以下断言：

断言10.5.2 在任意的可达系统状态中，如果$pc_i \in \{leave\text{-}try, crit, reset\}$且$pc_{\bar{i}} \in \{check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset\}$，则$turn \neq i$。

也就是说，如果 $\bar{i}$ 是竞争者，且i在竞争中取胜，则变量$turn$被设置为i受青睐的情况，即设为 $\bar{i}$ 的值。在断言10.5.2的证明的归纳步中，要检查的关键事件包括：

1）“成功的” *check-flag*$_i$事件，即那些导致pc_i到达*leave-try*的事件。

2）“成功的” *check-turn*$_i$事件。

3） *set* $-$ *turn*$_i$事件，该事件导致 $pc_{\bar{i}}$ 的值为*check-flag*。

4） *set–turn*$_i$事件，该事件使得结论$turn \neq i$不成立。 281

当i的*check-flag*$_i$成功时，必有$flag(\bar{i}) = 0$，则断言10.5.1意味着 $pc_{\bar{i}} \notin \{check\text{-}flag, check\text{-}turn,$

leave-try, crit, reset}，从而结论成立。当i的*check-turn*$_i$成功时，必有$turn \neq i$，得证。当$\bar{i}$执行$set-turn_i$时，它明确地设置$turn \neq i$，得证。最后，当i执行*set-turn*$_i$时，可得$pc_i = check\text{-}flag$，结论显然成立。

如此断言10.5.2得证。现在互斥性条件是显而易见的：假设在某可达状态中，i和$\bar{i}$都在C中，则应用断言10.5.2两次——对i和$\bar{i}$各一次——可得$turn \neq i$和$turn \neq \bar{i}$。矛盾。 □

引理10.11　*Peterson2P*算法保证演进性。

证明　采用反证法，假设α是一个低层次公平运行，它到达一点，在这点中至少一个进程（比如i）在T中且没有进程在C中；且假设在这点之后没有进程会进入C。分两种情况来讨论。首先，如果在α中给定点之后的某个时候进程$\bar{i}$处于T中，则由于两个进程都不会进入C，故它们都必然永远陷于各自的*check*循环中。但这是不可能发生的，因为*turn*的值必然稳定为一个受青睐进程的索引。

另一方面，如果在α中给定点之后的进程$\bar{i}$始终不在T中，我们可以证明$flag(\bar{i})$最终变成并保持为0，这与i陷于*check*循环中的假设矛盾。 □

引理10.12　*Peterson2P*算法是具有锁定权。

证明　对于退出区的证明很容易，故我们来考虑尝试区。我们来展示双界限绕过的更强条件并引用定理10.8。

假设*Peterson2P*算法不是具有锁定权，即在运行α的某点中，进程i在执行*set-flag*$_i$之后处于T中，其后当i留在T中时，进程i三次进入C。注意，在第二次和第三次时，$\bar{i}$必然先设置$turn := \bar{i}$，然后看见$turn = i$；它看不到$flag(i) = 0$，因为$flag(i)$仍为1。这意味着在α中的给定点之后*set-turn*$_i$最少发生两次，因为只有进程i才能把*turn*设置为i。但*set-turn*$_i$只在i的一个尝试区中执行了一次。矛盾。 □

这样我们得到定理10.13。

282 **定理10.13**　*Peterson2P*算法解决了互斥问题并保证锁定权。

复杂度分析　如同对*DijkstraME*算法的分析，令ℓ和c分别为进程步时间和临界区时间的上界。你可以重读10.3.4节开头部分的讨论以保证你准确地理解这些界限的意思。

定理10.14　在*Peterson2P*算法中，一个特定进程i从进入T到进入C之间的时间最多为$c + O(\ell)$。

证明概要　假设这个界限不成立，考虑一个运行，其中在某点上进程i处于T中，且在该点之后的至少$c + k\ell$时间内不进入C，这里k是一个特定的大常数，它比以下分析中的大O项中的常数要大得多。

首先，在最多3ℓ时间内，进程i执行*check-flag*$_i$。这可以根据i在其尝试区中的不同位置分情况讨论。注意在这个时间段内i的任一测试都不会成功，否则它会在时间$O(\ell)$内进入C，我们假设这种情况不会很快就发生。那么当进程i完成*check-flag*$_i$；它一定会找到$flag(\bar{i}) = 1$，否则i将在$O(\ell)$时间内到达C。根据断言10.5.1，在该点中必有$pc_i \in$ {*set-turn, check-flag, check-turn, leave-try, crit, reset*}。

然后我们来证明要么*crit*$_i$在额外的$O(\ell)$时间内发生，要么*reset*$_i$在额外的$c + O(\ell)$时间内发生。这可以根据*turn*的值和进程在代码中的位置分情况讨论来证明；关键在于一旦*turn*变量稳定下

来，其中一个进程将会受到青睐。前一种情况意味着i很快到达C，故考虑后一种情况，即 $reset_{\bar{i}}$ 在额外的$c + O(\ell)$时间内发生。

现在在额外的$O(\ell)$时间内i又一次执行$check\text{-}flag_i$，它再次发现$flag(\bar{i}) = 1$。这意味着 $\bar{i}$ 已经在 $reset_{\bar{i}}$ 之后再次进入T。那么要么$turn$的值已经为 $\bar{i}$，要么在额外的时间ℓ内$turn$的值变为 $\bar{i}$。然后，在另一时间$O(\ell)$内，进程i发现它得到青睐来进入C。这与在所给的时间内i不进入C的假设矛盾。图10-8显示了证明中的事件顺序和事件之间的时间界限。 □

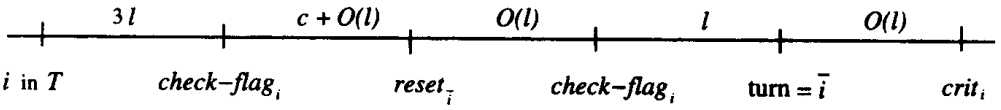


图10-8 定理10.14证明中的事件顺序和时间界限

10.5.2 n进程算法

对于n个进程，我们可以在一系列$n-1$次竞争中，在层1, 2, ..., $n-1$上反复使用*Peterson2P*算法的思想。在每个相继竞争中，算法保证至少有一个输者。所以，所有进程都可以参与层1 283
的竞争，但最多只有$n-1$个进程能够获胜。一般的，在层k上最多有$n-k$个进程能够获胜。所以在层$n-1$上最多只有一个进程能够获胜，从而产生互斥条件。

代码如下。在这里我们采用以前的约定，进程的编号（索引）为1, ..., n。我们把该算法称为*PetersonNP*算法。

***PetersonNP* algorithm:**

Shared variables:
for every $k \in \{1, \ldots, n-1\}$:
  $turn(k) \in \{1, \ldots, n\}$, initially arbitrary, writable and readable by all processes
for every i, $1 \leq i \leq n$:
  $flag(i) \in \{0, \ldots, n-1\}$, initially 0, writable by i and readable by all $j \neq i$

Process i:

```
** Remainder region **

try_i
for k = 1 to n − 1 do
    flag(i) := k
    turn(k) := i
    waitfor [∀j ≠ i : flag(j) < k] or [turn(k) ≠ i]
crit_i

** Critical region **

exit_i
flag(i) := 0
rem_i
```

进程i参与每一层k, $1 \leq k \leq n-1$上的竞争。每一层k都有自己的$turn$变量，即$turn(k)$。在每一层k上，进程i的行为与*Peterson2P*算法中进程的行为相似：它设置$turn(k) := i$, 然后等到所有其他进程的$flag$变量严格小于k或者$turn(k) \neq i$。也就是说，要么当前没有其他进程参与层k的竞 284

争，要么在i最近设置$turn(k)$变量之后其他变量也设置了$turn(k)$变量。

与前面一样，代码中也有需要解决的含糊之处。首先，waitfor语句中的其中一个条件涉及所有其他进程的*flag*变量。在我们的模型中，这些变量不能同时检测。而我们希望逐个地检查变量，当所有被检测的变量的所见值都小于k的时候，条件便满足。其次，我们必须明确在waitfor语句中进程i检测不同的*flag*变量和检测$turn(k)$变量的顺序。为了简单起见，我们假设进程i循环地进行检测，在每个循环中先以任意顺序检测所有*flag*变量，然后检测$turn(k)$变量。

细节如下。代码类似于*Peterson2P*算法的代码。局部变量*level*用于跟踪进程参与（或准备参与）哪个竞争，S用于跟踪那些已被观察到其*flag*值小于k的进程。

PetersonNP algorithm (rewritten):

Shared variables:
for every $k \in \{1, \ldots, n-1\}$:
　$turn(k) \in \{1, \ldots, n\}$, initially arbitrary
for every i, $1 \leqslant i \leqslant n$:
　$flag(i) \in \{0, \ldots, n-1\}$, initially 0

Actions of i:

Input:	Internal:
try_i	$set\text{-}flag_i$
$exit_i$	$set\text{-}turn_i$
Output:	$check\text{-}flag(j)_i$, $1 \leqslant j \leqslant n$, $j \neq i$
$crit_i$	$check\text{-}turn_i$
rem_i	$reset_i$

States of i:
$pc \in \{rem, set\text{-}flag, set\text{-}turn, check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially *rem*
$level \in \{1, \ldots, n-1\}$, initially 1
S, a set of process indices, initially $\emptyset$

285 **Transitions of i:**

try_i
　Effect:
　　$pc := set\text{-}flag$

$set\text{-}flag_i$
　Precondition:
　　$pc = set\text{-}flag$
　Effect:
　　$flag(i) := level$
　　$pc := set\text{-}turn$

$set\text{-}turn_i$
　Precondition:
　　$pc = set\text{-}turn$
　Effect:
　　$turn(level) := i$
　　$S := \{i\}$
　　$pc := check\text{-}flag$

$check\text{-}turn_i$
　Precondition:
　　$pc = check\text{-}turn$
　Effect:
　　if $turn(level) \neq i$ then
　　　if $level < n-1$ then
　　　　$level := level + 1$
　　　　$pc := set\text{-}flag$
　　　else
　　　　$pc := leave\text{-}try$
　　else
　　　$S := \{i\}$
　　　$pc := check\text{-}flag$

$crit_i$
　Precondition:
　　$pc = leave\text{-}try$
　Effect:

```
                                        pc := crit

check-flag(j)_i
  Precondition:                       exit_i
    pc = check-flag                     Effect:
    j ∉ S                                 pc := reset
  Effect:
    if flag(j) < level then           reset_i
      S := S ∪ {j}                      Precondition:
      if |S| = n then                     pc = reset
        S := ∅                          Effect:
        if level < n − 1 then             flag(i) := 0
          level := level + 1              level := 1
          pc := set-flag                  pc := leave-exit
        else
          pc := leave-try               rem_i
    else                                Precondition:
      S := ∅                              pc = leave-exit
      pc := check-turn                  Effect:
                                          pc := rem
```

现在我们来证明*PetersonNP*算法的正确性。良构性是显然的。对于互斥性，关键思想在于层k上的竞争只允许产生$n-k$个胜者。

在*PetersonNP*算法的任意系统状态中，如果$level_i > k$或者$level_i = k$且$pc_i \in \{leave\text{-}try, crit, reset\}$，则称进程$i$为一个层$k$上的胜者（后一个条件只有当$k = n-1$时才会出现）。如果进程$i$是一个层$k$上的胜者，或者$level_i = k$且$pc_i \in \{check\text{-}flag, check\text{-}turn\}$，则称进程$i$为层$k$上的竞争者。 286

引理10.15 *PetersonNP*算法满足互斥性。

证明 为了证明互斥性，我们来证明以下与*Peterson2P*算法中的断言10.5.2类似的断言。一个重要的区别在于现在的断言必须在检测标志的过程中处理中间阶段。

断言10.5.3 在*PetersonNP*算法的任意可达系统状态中，以下成立：

1）如果进程i是层k上的竞争者，$pc_i = check$-flag，且S_i中任意不等于i的进程j都是层k上的竞争者，则$turn(k) \neq i$。

2）如果进程i是层k上的胜者且其他进程都是层k上的竞争者，则$turn(k) \neq i$。

证明通常使用归纳法来完成，具体过程留做一道习题。使用断言10.5.3，我们可以证明：

断言10.5.4 在*PetersonNP*算法的任意可达系统状态中，如果存在层k上的竞争者，则$turn(k)$的值是层k上的某个竞争者的索引。

基于归纳法的具体证明也留做一道习题。最后，我们证明以下断言，它直接意味着互斥条件的成立。

断言10.5.5 在*PetersonNP*算法的任意可达系统状态中，对于任意k，$1 \leqslant k \leqslant n-1$，最多存在$n-k$个层$k$上的胜者。

断言10.5.5的证明也是基于归纳法的，但不是基于运行的长度，而是基于k的值进行归纳。

基础：$k = 1$。如果$k = 1$时命题为假，即所有n个进程都是层1上的胜者。则断言10.5.3意味着$turn(1)$的值不能为任何进程的索引，矛盾。

归纳步：假设对于k，$1 \leqslant k \leqslant n-2$命题成立，现在欲证明对于$k+1$命题也成立。采用反证法，

假设命题对于$k+1$为假，即存在严格多于$n-(k+1)$个层$k+1$上的胜者；令W为由这些胜者组
287 成的集合。每个层$k+1$上的胜者也是层k上的胜者，根据归纳假设，层k上的胜者数目最多为$n-k$。所以W也是层k上的胜者的集合，且$|W| = n-k \geq 2$。

则断言10.5.3意味着$turn(k+1)$的值不能够等于W中进程的索引。而断言10.5.4意味着$turn(k+1)$的值等于层$k+1$上某个竞争者的索引。但是层$k+1$上的每个竞争者都是层k上的竞争者，故都在W中。这是一个矛盾。 □

为了证明演进性，只需证明锁定权（见习题10.6）就足够了。定理10.9意味着锁定权可用一个时间界限来保证。对于退出区的时间界限是显然的；以下定理给出针对尝试区的时间界限。警告：我们并没有声称这个界限是严格的——收紧这个界限的任务留做一道习题——但是任一界限都足以证明锁定权。

定理10.16　在*PetersonNP*算法中，一个特定进程i从进入T到进入C为止的时间最多为$2^{n-1}c + O(2^n n\ell)$。

证明　我们使用递归来证明这个界限。定义$T(0)$为一个进程从进入T到进入C为止的最大时间。对于k, $1 \leq k \leq n-1$，定义$T(k)$为从进程从成为层k的胜者直到进入C为止的最大时间。我们希望界定$T(0)$的大小。

根据代码，我们知道$T(n-1) \leq \ell$，因为在赢得最后的竞争之后进程进入C只需一步。为了界定$T(0)$，我们根据$T(k+1)$来建立$T(k)$的递归式，其中$0 \leq k \leq n-2$。

假设进程i刚刚在层k（如果$k \geq 1$）上获胜，或者它刚刚进入T（此时$k=0$）。则在时间2ℓ之内，进程i执行$set\text{-}turn_i$，设置$turn(k+1) := i$。以π来表示这个$set\text{-}turn_i$事件。我们考虑两种情况。

第一种情况，如果$turn(k+1)$在π之后的时间$T(k+1) + c + (2n+2)\ell$内被设为除了i之外的值，则i在额外时间$n\ell$内在层$k+1$获胜。然后在额外的时间$T(k+1)$内，i进入C。在这种情况中，从π到i进入C的总共时间最多为$2T(k+1) + c + (3n+2)\ell$。

第二种情况，假设$turn(k+1)$在π之后的时间$T(k+1) + c + (2n+2)\ell$内没有被设为除了i之外的任何值，则没有进程能够在时间$T(k+1) + c + (2n+1)\ell$之内把其$flag$的值设置为$k+1$。令I为由不同于i的进程j组成的集合，其中当π发生时$flag(j) \geq k+1$。则I中的每个进程都在π之后（自从它发现$turn(k+1)$不等于其索引之后）的最多时间$n\ell$内在层$k+1$上获胜，然后在额外时间$T(k+1)$内进入C，之后在额外的时间c内离开C并在额外的时间ℓ内执行$reset$。故此，在π之后的时间$n\ell +$
288 $T(k+1) + c + \ell = T(k+1) + c + (n+1)\ell$内，$I$中的所有进程把它们的$flag$值设置为0。

因此，在π之后的时间$T(k+1) + c + (n+1)\ell$内，那些当π发生时满足$flag(j) \geq k+1$的所有不同于i的进程j都把它们的$flag$值设置为0。根据我们以上所做的假设，在此后的额外时间$n\ell$内，没有进程把其$flag$值设为$k+1$。进程i有足够的时间来检测出所有$flag$变量的值都小于$k+1$并在层$k+1$上获胜。故此，进程i在π之后的时间$T(k+1) + c + (2n+1)\ell$内获胜。进程i在另一$T(k+1)$内又进入C。在这种情况中，从π到i进入C的总共时间最多为$2T(k+1) + c + (2n+1)\ell$。

因此最坏情况的时间是以上两种情况的最大时间最多加上2ℓ，即$2T(k+1) + c + (3n+4)\ell$。我们需要求解如下$T(0)$的递归式：

$$T(k) \leq 2T(k+1) + c + (3n+4)\ell, \text{ 对于} 0 \leq k \leq n-2$$
$$T(n-1) \leq \ell$$

解出这个递归式便得到所需的时间界限。（参见10.5.3节中的一个类似递归式的更详细解法） □

可得

定理10.17 *PetersonNP*算法解决了互斥问题且是具有锁定权的。

10.5.3 锦标赛算法

另一种将*Peterson2P*算法扩充到更多进程中的方法是把基本双进程算法作为锦标赛中的建造砖块。为了简单起见，我们假设进程数目n是2的幂。进程的编号又是从0开始，为0, .., $n-1$，而不是1, ..., n。为了得到资源，每个进程参与$\log n$个竞争。这些竞争安排在一个拥有n个叶子的完全二叉锦标赛树中，树的n个叶子从左到右地对应n个进程0, ..., $n-1$。

我们需要某些记号来命名不同竞争、在所有竞争中进程的角色和所有进程在所有竞争中的潜在对手集合。对于$0 \leq i \leq n-1$和$1 \leq k \leq \log n$，定义以下记号：

- *comp*(*i*, *k*)，进程*i*的层*k*上的竞争，是由*i*的二进制表示中的前$\log n - k$位组成的字符串。在锦标赛树中，*comp*(*i*, *k*)可用做叶子*i*在层*k*上的内部节点的名字。特别的，根的名字为空字符串λ。 289
- *role*(*i*, *k*)，进程*i*在*i*的层*k*竞争中的角色，是*i*的二进制表示中的第$(\log n - k+1)$位。在锦标赛树中，*role*(*i*, *k*)指出叶子*i*是竞争*comp*(*i*, *k*)的左孩子的后代还是右孩子的后代。
- *opponents*(*i*, *k*)，进程*i*在*i*的层*k*竞争中的对手，是这些进程组成的集合：进程的索引与*i*的索引相比，前$\log n - k$位相同，但第$\log n - k + 1$位相反。在锦标赛树中，*opponents*(*i*, *k*)中的进程是节点*comp*(*i*, *k*)的另一儿子的后代，注意这个*comp*(*i*, *k*)的另一儿子不是叶子*i*的祖先节点。

例 10.5.1 锦标赛树

图10-9显示了$n = 8$时的锦标赛树。例如，*comp*(5, 2) = 1，*role*(5, 2) = 0且*opponents*(5, 2) = {6, 7}。

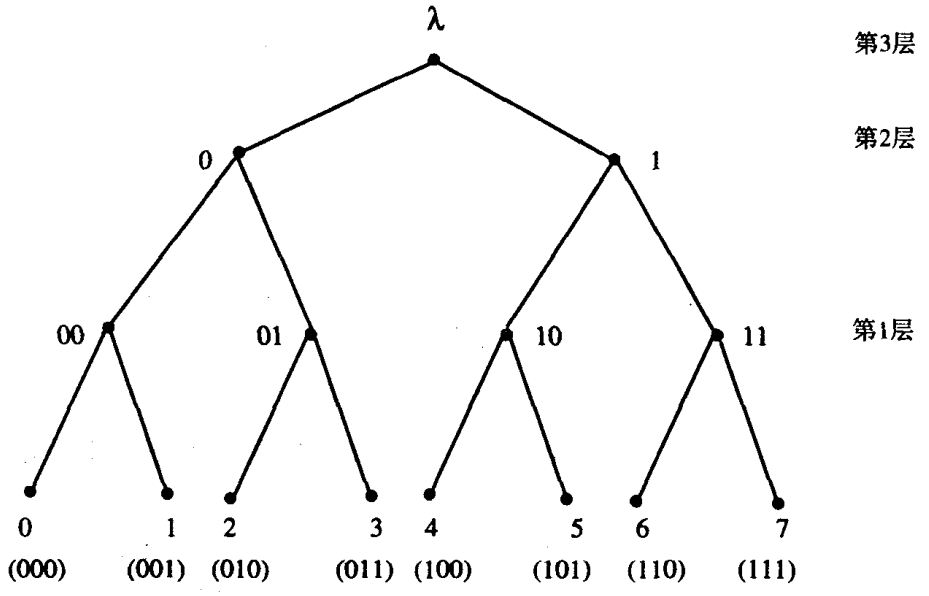


图10-9 锦标赛算法中竞争的名字

我们把该算法称为锦标赛算法（*Tournament algorithm*） 290

***Tournament* algorithm:**

Shared variables:
for every binary string x of length at most $\log n - 1$:
$turn(x) \in \{0, 1\}$, initially arbitrary, writable and readable by exactly those

processes i for which x is a prefix of the binary representation of i
for every i, $0 \leq i \leq n-1$:
$flag(i) \in \{0, \ldots, \log n\}$, initially 0, writable by i and readable by all $j \neq i$

Process i:

** Remainder region **

try_i
for $k = 1$ to $\log n$ do
　$flag(i) := k$
　$turn(comp(i,k)) := role(i,k)$
　waitfor $[\forall j \in opponents(i,k) : flag(j) < k]$ or $[turn(comp(i,k)) \neq role(i,k)]$
$crit_i$

** Critical region **

$exit_i$
$flag(i) := 0$
rem_i

这段代码与*PetersonNP*算法的代码非常相似。主要不同在于：在每个竞争中，进程只检查其竞争对手的标志。如同*PetersonNP*算法中，我们假设进程以任意顺序来检测对手，每次检测一个。检测以某种系统化的形式交替进行；例如，可以在所有*flag*都被检测的循环之后检测*turn*。由于锦标赛算法的证明思想与*Peterson2P*和*PetersonNP*的类似，我们只给出粗略证明。

首先，可以采用前提－结果的风格来重写算法，从而明确表示出程序计数器和那些把已经检测过*flag*的进程聚集起来的变量。锦标赛算法必须定义诸如“层k上的胜者”和“层k上的竞争者”等概念，这可以参考*PetersonNP*算法。

291 **引理10.18**　锦标赛算法满足互斥性。

证明概要　这里的证明用到了与*Peterson2P*和*PetersonNP*算法中不变式断言证明一样的思想。这次的关键不变式为：

断言10.5.6　在锦标赛算法的任意可达系统状态中，对于任意k, $1 \leq k \leq \log n$，以层k上节点为根的任一子树中最多有一个进程能够成为层k上的胜者。

接下来是从不变式算法模拟到断言10.5.3的第二部分。

断言10.5.7　如果进程i是层k上的胜者，并且如果i在层k上的任何对手是层k的竞争者，那么$turn(comp(i,k)) \neq role(i,k)$。

如同断言10.5.3的第二部分的做法，我们不能直接使用归纳法来证明断言10.5.7。我们必须增加以下信息来增强它：在进程发现它的某些对手的*flag*变量值严格小于k时，在waitfor循环中会发生什么。具体的增强与归纳证明工作留做一道习题。　□

为了证明演进性和锁定权，我们来证明一个时间界限。

定理10.19　在锦标赛算法中，特定进程i从进入T到进入C为止的时间最多为$(n-1)c + O(n^2\ell)$。

证明　这里的证明与定理10.16的证明类似。定义$T(0)$为一个进程从进入T到进入C为止的最大时间。对于k, $1 \leq k \leq \log n$，定义$T(k)$为进程从在层k上获胜直到进入C为止的最大时间。我们希望界定$T(0)$的大小。根据代码，我们知道$T(\log n) \leq \ell$，因为在赢得最后的竞争之后进程进入C

只需一步。我们根据$T(k+1)$来界定$T(k)$，其中$0 \leq k \leq \log n - 1$。

假设进程i刚刚在层k（如果$k \geq 1$）上获胜，或者刚刚进入T（此时$k = 0$）。令x代表$comp(i, k+1)$。则在时间2ℓ之内，进程i将变量$turn(x)$的值设置为$role(i, k+1)$。以π来表示这个事件。我们考虑两种情况。

第一种情况，如果$turn(x)$在π之后的时间$T(k+1) + c + (2^{k+1} + 4)\ell$内发生改变，则$i$在额外时间$(2^k+1)\ell$内在层$k+1$获胜。然后在额外的时间$T(k+1)$内，$i$进入$C$。在这种情况中，从$\pi$到$i$进入$C$的总共时间最多为$2T(k+1) + c + (2^{k+1} + 2^k + 5)\ell$。

第二种情况，假设$turn(x)$在π之后的时间$T(k+1) + c + (2^{k+1} + 4)\ell$内没有发生改变，则没有层$k$上的对手能够在$\pi$之后的时间$T(k+1) + c + (2^{k+1} + 3)\ell$内把其$flag$的值设置为$k+1$。如果$j$是$i$在 292
层$k+1$上的对手，且当π发生时有$flag(j) \geq k+1$。则在π之后的时间$(2^k+1)\ell + T(k+1) + c + \ell = T(k+1) + c + (2^k+2)\ell$内，进程$j$把它的$flag$值设置为0。

因此，在π之后的时间$T(k+1) + c + (2^k+2)\ell$内，那些当π发生时满足$flag(j) \geq k+1$的层$k+1$上的i的对手都把它们的$flag$值设置为0。根据我们以上所做的假设，在此后的额外时间$(2^k+1)\ell$内，没有进程把其$flag$值设为$k+1$。进程i有足够的时间来检测出所有层$k+1$上i的对手的$flag$值都小于$k+1$并在层$k+1$上获胜。故此，进程i在π之后的时间$T(k+1) + c + (2^{k+1} + 3)\ell$内获胜。进程$i$在另一个$T(k+1)$内又进入$C$。在这种情况中，从$\pi$到$i$进入$C$的总共时间最多为$2T(k+1) + c + (2^{k+1} + 3)\ell$。

因此最坏情况的时间是以上两种情况的最大时间加上最多2ℓ，即$2T(k+1) + c + (2^{k+1} + 2^k + 7)\ell$。我们需要求解如下$T(0)$的递归式：

$$T(k) \leq 2T(k+1) + c + (2^{k+1} + 2^k + 7)\ell, \text{对于} 0 \leq k \leq \log n - 1$$
$$T(\log n) \leq \ell$$

选出某个常数α，使得$(2^{k+1} + 2^k + 7) \leq \alpha \cdot 2^k$。然后，

$$\begin{aligned}
T(0) &\leq 2T(1) + 2^0 c + a2^0\ell \\
&\leq 2^2T(2) + (2^0 + 2^1)c + a(2^0 + 2^2)\ell \\
&\leq 2^3T(3) + (2^0 + 2^1 + 2^2)c + a(2^0 + 2^2 + 2^4)\ell \\
&\vdots \\
&\leq 2^kT(k) + (2^0 + 2^1 + \ldots + 2^{k-1})c + a(2^0 + 2^2 + \ldots + 2^{2k-2})\ell \\
&\vdots \\
&\leq 2^{\log n}T(\log n) + (2^0 + 2^1 + \ldots + 2^{\log n - 1})c + a(2^0 + 2^2 + \ldots 2^{2(\log n - 1)})\ell \\
&\leq (n-1)c + n\ell + O\left(n^2\ell\right) \\
&= (n-1)c + O\left(n^2\ell\right)
\end{aligned}$$

定理10.20 锦标赛算法解决了互斥问题且是具有锁定权的。 □

有界绕过次数 锦标赛算法并不保证绕过次数的任意界限。为了看到这一点，考虑一个运行，其中进程0进入锦标赛的叶子，并以假定的上界ℓ来执行中间步。同时，进程$n-1$进入锦标赛的 293
叶子，并执行得更快。进程$n-1$可以到达顶部并获胜，且在进程0在层1上获胜之前，它可以重复这个过程很多次。由于我们不假定进程步时间的下界，这是可能的。

注意在无界绕过次数和时间上界之间并无矛盾。没有进程会永远等待——无界绕过次数只有在某个进程非常快地操作时才会发生。

10.6 使用单写者共享存储器的算法

目前为止，我们研究的算法都是使用多写者共享寄存器（变量*turn*）和单写者共享寄存器（变量*flag*）。因为多写者寄存器常常难于实现，设计只使用单写者共享寄存器的算法是值得的。在本节和下一节中，我们给出两个这样的算法。

本节的算法解决了互斥问题（和演进性问题），但不保证任何高层次公平性条件。该算法的共享存储器都是二进制的。10.7节中的算法是具有锁定权的，但其缺点是使用了无界大小的变量。

我们称Burns之后的第一个算法为*BurnsME*。

***BurnsME* algorithm:**

Shared variables:
for every i, $1 \leq i \leq n$:
　$flag(i) \in \{0,1\}$, initially 0, writable by i and readable by all $j \neq i$

Process *i*:

```
        ** Remainder region **

        try_i
L:      flag(i) := 0
        for j, 1 ≤ j ≤ i − 1 do
            if flag(j) = 1 then goto L
        flag(i) := 1
        for j, 1 ≤ j ≤ i − 1 do
            if flag(j) = 1 then goto L
M:      for j, i + 1 ≤ j ≤ n do
            if flag(j) = 1 then goto M
294     crit_i

        ** Critical region **

        exit_i
        flag(i) := 0
        rem_i
```

---

*BurnsME*算法中的*flag*值是0和1，而不是*DijkstraME*算法中的0、1和2。每个进程执行三个循环。前两个循环检测那些索引比当前进程的索引小的进程的*flag*值，而第三个循环检测索引更大的进程的*flag*值。如果进程i通过三个循环中的所有检测，进入临界区。

引理10.21 *BurnsME*算法满足互斥性。

证明 这里的证明与满足互斥性的*DijkstraME*算法（见引理10.3）的第一个（操作式）证明相类似。不同之处在于现在*flag*变量被设为1，而在*DijkstraME*算法中它们被设为2。

因此，如果进程i和j同时处于C中，则假设i先把其*flag*设为1。然后直至进程i离开C为止，$flag(i)$保持为1。但在j把$flag(j)$设为1之后，j能够进入C之前，j必须检测到$flag(i) = 0$。（如果$i < j$，则这一步在第二个循环中完成；如果$i > j$，则在第三个循环中完成。）该检测必然在$flag(i) = 1$时的时间间隔中发生，从而产生矛盾。 □

注意第一个循环的代码对于互斥性条件来说是不必要的。

引理10.22 *BurnsME*算法保证演进性。

证明 对于退出区的证明相当容易。对于尝试区，采用反证法，假设α是一个低层次公平运行，在其到达的一点中，最少一个进程处于T中且没有进程处于C中，并假设在该点之后，没有进程会进入C。与引理10.4的证明相似，不失一般性，假设在α中每个进程都在T或R中，且没有进程改变在α中的区域。把在T中的进程称为竞争者（contender）。

现在把竞争者分成两个集合：曾经到达标号M的集合和未曾到达标号M的集合。把第一个集合称为P，把第二个集合称为Q。则α中必有一点，该点之前所有P中的进程已经到达标号M；注意它们再也不会回到标号M之前的代码中任一点中去。令α_1为α的一个后缀，其中在标号M之后P中的所有进程都处于最后一个循环。 295

我们来证明最少一个进程在P中。特别的，所有竞争者中具有最小索引的那个进程自从到达标号M之后不会被阻塞。

令i为P中具有最大索引的进程。我们来证明最终在α_1中，任意满足$j > i$的进程$j \in Q$都永久地把$flag(j)$设置为0。这是因为：每当j执行前两个循环中的一个时，它发现最小索引竞争者的存在并返回L。每当它这样做的时候，它设置$flag(j) := 0$，而且，一旦它这样做之后，它再也不能前进到设置$flag(j) := 1$。令α_2为α_1的一个后缀，其中Q中所有索引大于i的进程都总是把它们的$flag$设为0。

在α_2中，没有东西能够阻止进程i到达C：每个更大索引的进程j的$flag(j) = 0$，故i成功地完成第三个循环。因此，i进入C，矛盾。 □

定理10.23 *BurnsME*算法解决了互斥问题。

10.7 *Bakery*算法

本节给出针对互斥问题的*Bakery*算法。它的工作方式与面包坊的工作方式相似，客户进入面包坊时得到面包票，面包坊按照票号的顺序对客户进行服务。

*Bakery*算法只使用单写者/多读者共享寄存器。事实上，它使用一种称为安全寄存器的更弱形式的寄存器，当读和写同时进行时，这种寄存器允许为读操作提供任意回应。

*Bakery*算法保证锁定权且具有一个良好的时间界限。它保证有界绕过次数，还满足一个相关条件：它满足“无等待门廊之后的FIFO”（定义见下面）。*Bakery*算法的一个缺点是它使用了无界大小的寄存器。

代码如下。如果我们只对正常的寄存器（而不是诸如安全寄存器等更弱类型的寄存器）感兴趣的话，代码可以简化。我们把简化工作留做习题。 296

***Bakery* algorithm :**

Shared variables:
for every i, $1 \le i \le n$:
  $choosing(i) \in \{0,1\}$, initially 0, writable by i and readable by all $j \neq i$
  $number(i) \in \mathbb{N}$, initially 0, writable by i and readable by all $j \neq i$

Process i:

  ** Remainder region **

  try_i
  $choosing(i) := 1$
  $number(i) := 1 + \max_{j \neq i} number(j)$

```
choosing(i) := 0
for j ≠ i do
    waitfor choosing(j) = 0
    waitfor number(j) = 0 or (number(i), i) < (number(j), j)
crit_i

** Critical region **

exit_i
number(i) := 0
rem_i
```

在*Bakery*算法中，尝试区中直至进程i设置$choosing(i) := 0$为止的代码部分被称为门廊（doorway）。在门廊中，进程i选出一个比它读到的所有其他进程的号码更大的*number*。它每次一个地以任意顺序读取其他进程的*number*值，然后写自己的*number*值。当进程i读取并选出*number*时，它确保$choosing(i) = 1$，这作为给其他进程的信号。

注意有可能两个进程同时处于门廊中，这样的话会导致它们选出同一个*number*。为了打破均衡，进程不只是比较它们的*number*，而是比较它们的(*number*, *index*)对。比较按字典顺序进行，*number*相同时具有更小索引的进程受到青睐。

在尝试区的余下部分，进程等待其他进程完成选择并等待其(*number*, *index*)对变得最小。

为了证明正确性，用D来表示门廊（即由处于门廊中的进程的状态组成的集合），用$T - D$
297 来表示尝试区的余下部分。良构性是易见的。为了证明互斥性条件，我们用到一条引理。

引理10.24　在*Bakery*算法的任意可达系统状态中，对于任意进程i和j, $i \neq j$，以下成立：如果i在C中且j在$(T - D) \cup C$中，则$(number(i), i) < (number(j), j)$。

我们给出一个操作式证明，因为它更容易被扩充到安全寄存器中。

证明　固定一个运行中的某点s，其中i在C中且j在$(T - D) \cup C$中（正式的，s是某个系统状态的发生）。把在点s中的$number(i)$和$number(j)$的值称为它们的当前值。

在进入C之前，进程i在第一个循环中必然读到$choosing(j) = 0$。用π来表示这个事件；则π在s之前发生。当π发生时，j不在“选择区”（即在设置$choosing(j) := 1$之后的门廊部分）中。但是由于j在$(T - D) \cup C$中，j必然在某一点中通过选择区。分两种情况来考虑。

1）j在π之后进入选择区。则在j开始选择之前正确的$number(i)$被选出，保证j在进行选择的时候会看到正确的$number(i)$值。因此，在点s中，$number(j) > number(i)$成立，得证。

2）j在π之前离开选择区。则只要i在第二个循环中读取j的*number*值，它就得到正确的$number(j)$值。但是由于i决定进入C，则必有$(number(i), i) < (number(j), j)$。得证。　□

引理10.25　*Bakery*算法满足互斥性。

证明　假设在某个系统可达状态中，进程i和进程j都在C中。则应用引理10.24两次，可得$(number(i), i) < (number(j), j)$和$(number(j), j) < (number(i), i)$。矛盾。　□

引理10.26　*Bakery*算法保证演进性。

证明　通常，对于退出区的证明很容易。对于尝试区，我们再次采用反证法来证明。假设演进性得不到满足。则最终到达一点，其中所有进程都在T或R中，且没有新的区域变化发生。根据代码，T中的
298 所有进程最终完成门廊并到达$T - D$。然后具有最小(*number*, *index*)对的进程毫无阻碍地到达C。　□

引理10.27 *Bakery*算法保证锁定权。

证明 考虑在T中的特定进程i，假设它从未到达C。进程i最终完成门廊并到达$T-D$。之后，任意进入门廊的新进程都看见i的最新*number*并选出更高的号码。因此，由于i不到达C，这些新进程也不会到达C，因为每个新进程都在第二个等待循环中检测*number*(i)时被阻塞。

但是，反复使用引理10.26，可知必有进程不断地执行，无数多的*crit*事件发生，这与所有进入尝试区的新进程都被阻塞起来的事实相矛盾。 □

定理10.28 *Bakery*算法解决了互斥问题且是具有锁定权的。

复杂度分析 进程i从进入尝试区到进入临界区的时间上界为$(n-1)c+O(n^2\ell)$。这不易证明，我们只给出大概的证明，具体细节留做一道习题。

首先，进程i完成门廊需要$O(n\ell)$时间；我们必须界定i花在$T-D$中的时间间隔I的长度。令P为在i时进入$T-D$时已经处于T中的其他进程组成的集合。则在i进入C之前，只有P中的进程能够进入C，且每个这样的进程只能进入一次。在时间间隔I中，有某个进程处于C中这一情形的总时间最多为$(n-1)c$，而有某个进程处于门廊中的情形的总时间最多为$O(n^2\ell)$。

现在来界定时间间隔I中的驻留时间，即没有进程在C或门廊中的总时间。我们通过考虑在$P\cup\{i\}$中进程的演进来界定这段驻留时间。在这段时间期间，$P\cup\{i\}$中的进程不会被阻塞在第一个waitfor循环中，原因在于所有的*choosing*变量都为0。另外，$P\cup\{i\}$中的某进程没有被阻塞在第二个waitfor循环中，它将在时间$O(n\ell)$之内进入C。在它完成之后，$P\cup\{i\}$中的另一进程不会被阻塞，它将在额外的$O(n\ell)$时间内进入C，如此类推。这种过程一直重复，直到i进入C。总的驻留时间为$O(n^2\ell)$。

无等待门廊之后的FIFO *Bakery*算法保证一个比锁定权更强的高层次公平性条件。也就是说，如果进程i在j进入T之前完成门廊的工作，则在i进入C之前j不能进入C。注意算法的FIFO特性并不是真正基于进程进入T的时间或进程在T中执行第一个局部控制步的时间。例如，进程1进入T并设置*choosing*(1) := 1；然后进程2进入T，选出一个*number*，完成门廊；然后进程1选出自己的*number*。在这种情况中，进程1将会选出一个比进程2的*number*更大的*number*，使得进程2首先进入C。 299

声明一个算法满足“门廊之后FIFO”属性可能并无用处，因为我们没有限制门廊可以在什么地方结束。（如果门廊在进入C后立刻结束，则这个声明完全是平凡的）。然而，在*Bakery*算法中的门廊拥有一个有意思的属性：它是*无等待的*，即不管其他进程是否继续执行步，如果一个进程继续执行步，它保证最终完成门廊。

因此，*Bakery*算法满足“无等待门廊之后FIFO”的属性，该属性是一个非平凡但有意思的高层次公平条件。

10.8 寄存器数量的下界

我们已经给出几个使用读/写共享存储器的互斥算法。这些算法都保证了互斥性和演进性的基本条件，且大多数算法都保证了某种高层次公平性条件：锁定权、时间界限或绕过次数界限。所有算法的一个共同点是它们都使用了最少n个共享变量。

本节将证明这不是偶然的：解决互斥问题最少需要n个读/写共享变量！即使只要求基本的条件——互斥性和演进性——而不要求高层次公平性的话结论也成立。另外，这个不可能性结果与共享变量的大小（以所容纳的变量数目来衡量）无关——共享变量的大小可以只是一个单

独位或者甚至可以是无界的。这个结果代表了共享存储器系统的一个根本限制。

我们需要两个定义。第一个定义，如同9.3节中的做法，对于两个系统状态s和s'，如果进程i的状态、U_i的状态和所有共享变量的值在s和s'中都是相同的，则称s和s'对于进程i是不可区分的，记作$s \overset{i}{\sim} s'$。第二个定义，如果在一个系统状态s中，所有进程都处于剩余区中，则定义这个状态为空闲的。

在证明中，我们考虑一个固定的用户自动机集合。我们假设某个用户U_i都是尽可能地不确
300 定的——在遵循良构性条件的前提下，它可以在任意时间执行*try*和*exit*输出。把注意力限制在这个用户集合中并不会导致一般性的丢失。对于每个i，事实上都有一个单独的I/O自动机U_i确实表现出所允许的不确定性，具体证明留做一道习题。

10.8.1 基本事实

证明中用到两个基本事实。第一个是从空闲状态开始，一个运行着的进程能够到达临界区。

引理10.29　假设算法A只使用读/写共享变量来解决了$n \geq 2$个进程的互斥问题（即保证良构性、互斥性和演进性）。并假设s是一个可达的空闲系统状态，令i是任一进程。

则存在一个从状态s开始的且只涉及进程i的步的运行片断[⊖]，其中进程i到达C。

证明　根据演进性条件可得。（正式地，演进性条件被应用到一个包含s的低层次公平运行上，其中i在s的发生后进入C。）　□

作为一个简单的推论，我们知道从某个将会变成空闲状态的系统状态开始，一个运行着的进程能够到达C。

引理10.30　假设算法A只使用读/写共享变量来解决了$n \geq 2$个进程的互斥问题。令s和s'为对于进程i不可区分的可达系统状态，假设s'是一个空闲状态。

则存在一个从状态s开始的且只涉及进程i的步的运行片断，其中进程i到达C。

第二个事实是任意到达C的进程之前必然对共享存储器执行了写操作。

引理10.31　假设算法A只使用读/写共享变量来解决了$n \geq 2$个进程的互斥问题。假设s是一个可达的空闲系统状态，其中进程i处于剩余区中。假设在一个从状态s开始的且只涉及i的步的运行
301 片断中，进程i到达C。则期间i必然写入某个共享变量。

证明　令α_1任一有限运行片断，它从状态s开始，只涉及i的步，且以进程i在C中作为结束。采用反证法，假设α_1不包含对任何共享变量的写。用s'来表示α_1的结尾状态。由于进程i不写入任何共享变量，s和s'之间的惟一区别在于进程i的状态和用户U_i的状态不同。所以对于任意$j \neq i$，有$s \overset{j}{\sim} s'$。

反复使用演进性条件，可知存在一个从状态s开始但不包含进程i的步的运行片断，其中某个进程进入C。因为对于任意$j \neq i$，有$s \overset{j}{\sim} s'$，所以也存在从状态s'开始的这样一个运行片断。

但是这样很容易产生一个反例运行α。运行α以导致可达状态s的一个有限运行片断作为开始，随后为α_1，使得i不写入任意共享变量就进入C。则从s'开始，另一进程无需经过i的任意步就进入C。这违反了互斥性条件，因为在α的结尾有两个进程进入C。　□

⊖　即从任意状态开始（不必从算法的初始状态开始）的运行。

10.8.2 单写者共享变量

如果共享存储器被限定为单写者/多读者的读/写寄存器（如在*BurnsME*算法和*Bakery*算法中使用的变量），则引理10.29和引理10.31立刻意味着一个下界：

定理10.32 如果算法A只使用单写者/多读者的读/写共享变量来解决了$n \geq 2$个进程的互斥问题，则A最少使用了n个共享变量。

证明 考虑任一进程i。根据引理10.29，从A的一个初始（空闲）系统状态开始，i自己进入C。然后引理10.31意味着期间I必定写入某个共享变量。因为该结论对于任意进程i成立，且每个共享变量只有一个写者，所以必然最少有n个共享变量。 □

10.8.3 多写者共享变量

注意在我们给出的算法中，即使是那些使用了多写者寄存器的算法（例如，*DijkstraME*算法和*Peterson*算法）也需要最少n个变量。在本节中，我们把定理10.32扩展到多写者寄存器的情况。也就是说，我们来证明：

302

定理10.33 如果算法A只使用读/写共享变量来解决$n \geq 2$个进程的互斥问题，则A最少使用n个共享变量。

为了给出直观证明，我们从对两种特殊情况的证明开始。首先我们证明针对两个进程和一个变量的不可能性结果，然后针对三个进程和两个变量来证明。之后，我们把这些思想扩展到一般情况中去。

三个进程和两个变量 证明中多次用到以下定义：如果在系统状态s中进程i能够写入x（即进程i能够在下一步中写入x），则称i覆盖s中的共享变量x。

定理10.34 没有算法能够只使用一个读/写共享变量来解决了两个进程的互斥问题。

证明 采用反证法。假设存在这样的一个只使用了单个共享寄存器x的算法A。令s为一个初始（空闲）系统状态。我们来构造一个违反互斥性的A的运行。

引理10.29和引理10.30意味着存在一个只涉及进程1的运行，该运行从状态s开始，其中进程1进入C，且在此之前进程1写入单个共享变量x。在进程1写入x之前的一步，它覆盖x。令α_1是该运行的一个前缀，其结束点是进程1覆盖x的第一点；记α_1的最后状态为s'。由于在α_1中进程1没有把任何东西写入到共享存储器中，所以 $s \overset{2}{\sim} s'$。则引理10.30意味着从状态s'开始，进程2可以独自到达C。

反例运行α以α_1开始，它将进程1带至状态s'，此时进程1覆盖x。然后它令进程2从s'开始运行到C。接着运行进程1，允许它写入x，即对进程2在到达C的过程中可能写入的任意值进行改写。这消除了进程2的运行的所有轨迹。则进程1可以像只有自己在运行一样继续运行到C。这导致两个进程都处于C中，从而与互斥条件矛盾。

图10-10描绘了运行α。它把进程2的几步“连接”成一个只涉及进程1的步的运行。 □

三个进程和两个变量 现在给出针对三个进程和两个变量的不可能性结果。

定理10.35 没有算法能够只使用两个读/写共享变量来解决三个进程的互斥问题。

303

证明 采用反证法。假设存在这样的一个只使用了共享寄存器x和y的算法A。令s为一个初始（空闲）系统状态。我们来构造一个违反互斥性的A的运行。

我们使用以下策略。从s开始，我们只操纵进程1和2，直到在一点中每个进程都分别覆盖一个变量（x或y）；另外，对进程3来说，这个结果状态s'与一个可达状态是不可分的。然后进

程3从状态s'开始自己运行到C，引理10.30意味着这是可能的。

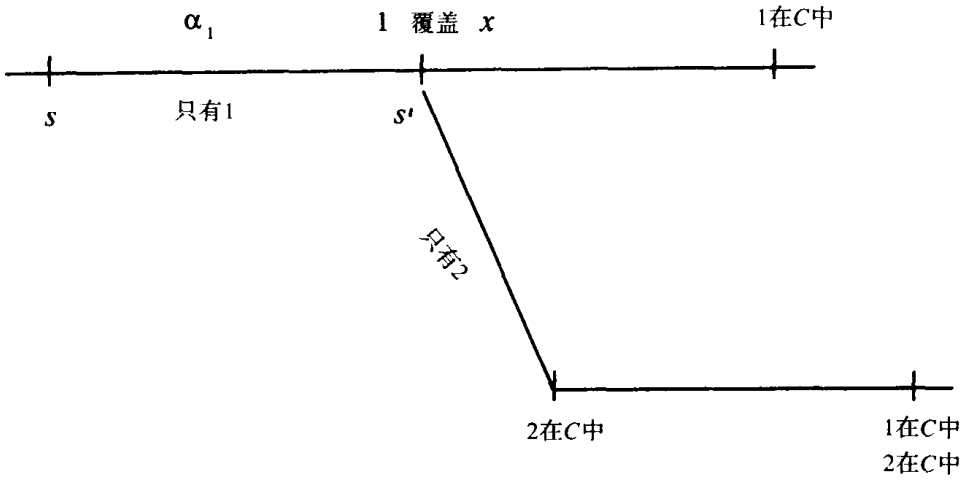


图10-10　定理10.34的证明中的运行α

接着，让进程1和2都执行一步。由于每个进程都分别覆盖一个共享变量，因此可以消除进程3的运行的所有轨迹。让进程1和2都继续执行步，由于已经消除了进程3所有轨迹，故此可以认为进程3从未进入其尝试区。因此，根据演进性条件，进程1或者进程2最终到达C。但这样的话两个进程都在C中，这与互斥条件相矛盾。

剩下的工作是显示如何操纵进程1和2，使得当进程3仍处于R中时，进程1和2覆盖两个共享变量。过程如下（见图10-11）。

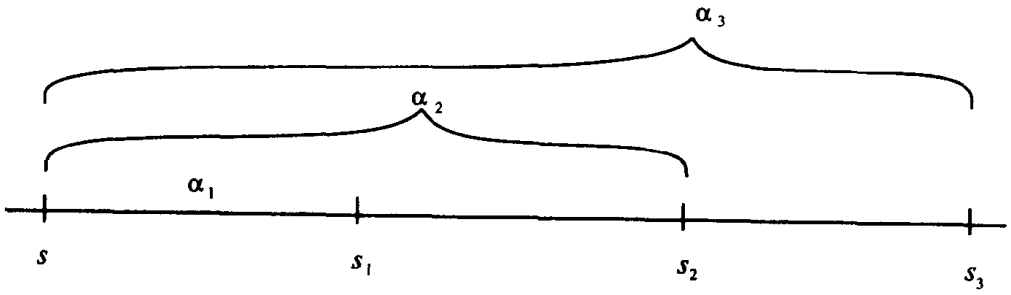


图10-11　进程1独自运行

首先，通过从s独自运行进程1直至它第一次覆盖一个共享变量为止，构造一个运行α_1。然后，通过继续独自运行进程1直至它进入C、再到E、再到R、又到T，又一次覆盖某个共享变量
304 为止，将α_1扩展为α_2。以同样方法把α_2扩展为α_3。令α_1，α_2和α_3的最终状态分别为s_1、s_2和s_3。

由于只有两个共享变量，在状态s_1、s_2和s_3的其中两个状态中，进程1必然覆盖同一个变量。假设在s_2和s_3中，进程1覆盖变量x。（其他情况的证明是类似的）。

现在来考虑从状态s_2开始单独运行进程2的情况。根据引理10.30，因为状态s_2和α（其中进程1在R中）的之前最后状态对于进程2来说是不可区分的，所有进程2将会进入C。另外，其间进程2必定写入另一个共享变量y。否则，进程2会到达C，然后进程1执行一步，把进程2对变量x的写入结果进行改写，从而消除进程2的所有轨迹，之后进程1继续执行并违反互斥性条件。

305 现在来构造一个反例运行α（见图10-12），它以α_2开始，然后把进程1带到覆盖x的一点，再令进程2运行到它首次覆盖y的一点。在这一点中，进程1和2分别覆盖变量x和y。但是任务还未完成，因为对于进程3来说，结果状态与某个空闲状态必须是不可区分的。（到目前为止，自

进程2最后一次离开R之后它可能写入x，这可能被进程3检测到）

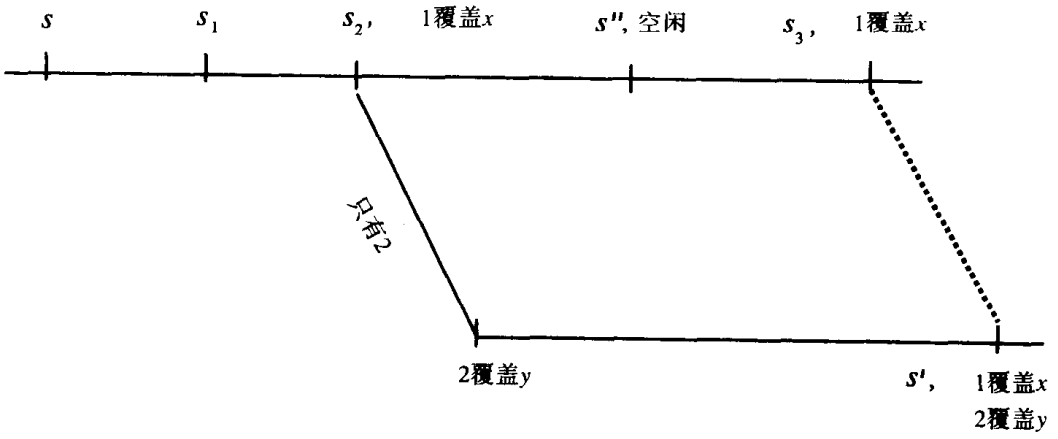


图10-12 α的构造

所以我们继续构造过程；从进程2覆盖y的那一点开始，我们继续运行进程1。进程1首先写入x，从而消除进程2的所有轨迹。然后进程1可以像只有自己在运行一样继续运行并到达一点，该点看起来像是α_3之后的一点，其中进程1又一次覆盖x。现在对α的构造完成；令s'为α的最后状态。

我们来证明α具有我们希望的所有性质。显然在状态s'中，进程1和2分别覆盖变量x和y。剩下要证明的是对于进程3来说，s'与某个空闲状态必然是不可区分的。令s''为在α_3发生的最后一个空闲状态，则s''与s_3的惟一区别在于进程1和用户U_1的状态不同，而s'与s_3的惟一区别在于进程2和用户U_2的状态不同。故有 $s' \overset{3}{\sim} s''$，得证。 □

一般情况 对一般情况的证明是对上述两种特殊情况的证明的自然扩充，它基于对变量数目的归纳来完成。我们需要另外一个基本事实——引理10.31的增强版本。该事实说明一个进程不仅在到达C的过程中必然写入某个变量，它还必然写入一个不被其他进程所覆盖的变量。在定理10.35的证明中，我们已经应用了这种思想。

引理10.36 假设算法A只使用读/写共享变量来解决了$n \geq 2$个进程的互斥问题。假设s是一个可达的空闲系统状态，其中进程i处于剩余区中。假设在一个从状态s开始的且只涉及i的步的运行片断中，进程i到达C。则进程i必然写入某个不被s中其他进程所覆盖的共享变量。

证明概要 与引理10.31的证明类似。主要不同在于现在我们必须保证那个涉及其他进程的运行片断以每个进程的单独步开始，这些步覆盖一个共享变量，并因而改写这个变量。这允许其他进程删除i的计算的所有轨迹。详细证明留做一道习题。 □

现在来证明一条主要引理。对于任意k，$1 \leq k \leq n$，如果从一个状态s开始，我们能够只使用进程1, ..., k的步来到达一个系统状态s'，则称s'为从s开始k可到达的。 306

引理10.37 假设算法A只使用读/写共享变量来解决了$n \geq 2$个进程的互斥问题。假设恰好有$n-1$个共享变量。令s为任一可达空闲系统状态。设$1 \leq k \leq n-1$。则存在两个满足以下性质的系统状态s'和s''，其中每个状态都是从s开始k可到达的。

1）在s'中进程1, ..., k覆盖k个不同的变量。

2）s''是一个空闲状态。

3）对于所有$i, k+1 \leqslant i \leqslant n$，有 $s' \overset{i}{\sim} s''$。

证明　对k做归纳。

基础：$k = 1$。从s开始独自运行进程1直至它第一次覆盖一个共享变量；引理10.29和引理10.30意味着这是可能的。将结果状态记为s'，而$s'' = s$，命题成立。

归纳步：假设引理对k, $1 \leqslant k \leqslant n-2$成立，我们来证明它对$k+1$也成立。使用归纳假设，我们得到一个从$s$开始$k$可到达的状态$t_1$，其中进程1, ..., k覆盖k个不同变量；然而，对于进程$k+1$, ..., n来说，t_1和某个也从s开始k可到达的空闲状态是不可区分的。接着，令所有进程1, ..., k都从t_1开始执行一步，从而写入其覆盖的变量。之后令所有进程1,⋯, k前进到R，得到一个新空闲状态u_1。

现在再次使用归纳假设来得到一个从u_1开始k可到达的状态t_2，其中进程1, ..., k覆盖k个不同变量；然而，对于进程$k+1$, ..., n来说，t_2和某个也从u_1开始k可到达的空闲状态是不可区分的。接着，也令所有进程1, ..., k写入其覆盖的变量并回到一个新空闲状态u_2中。

我们将该过程重复$\binom{n-1}{k}+1$次，产生“覆盖状态”$t_1,\cdots, t_{\binom{n-1}{k}+1}$。现在，根据鸽巢原理，在这$\binom{n-1}{k}+1$个覆盖状态中，必定存在两个覆盖状态，其中在每个状态中进程1,..., k覆盖相同的k个共享变量。把这k个变量组成的集合记为X。对于这两个覆盖状态，第一个记做s_1，第二个记作s_2。又令s_1'和s_2'为分别对应s_1和s_2而构造的空闲状态。因此，对于所有i, $k+1 \leqslant i \leqslant n$，有 $s_1 \overset{i}{\sim} s_1'$ 和 $s_2 \overset{i}{\sim} s_2'$。

现在考虑从状态s_1开始单独运行进程$k+1$时的情况。由于 $s_1 \overset{k+1}{\sim} s_1'$ 且s_1'是一个可达空闲状态，引理10.30意味着进程$k+1$能够最终进入C。根据引理10.36，期间进程$k+1$必然写入某个不在X中的变量。

307　现在准备定义两个所需的从原来状态s开始$k+1$可到达的状态s'和s''。（见图10-13）。首先，为了定义s'（其中$k+1$个变量被覆盖），从s_1开始，运行进程$k+1$直至它第一次覆盖某个不在X中的变量。然后继续运行进程1, ..., k，让它们先写入覆盖变量，然后前进到与s_2对应的点，其中它们又覆盖X。结果状态记为s'。注意除了进程$k+1$的状态之外，s'与s_2是相同的。为了定义s''（空闲状态），我们简单地令$s'' = s_2'$。

我们来证明s'和s''具有所有所需性质。首先，注意构造（包括归纳步的假设）中只涉及到进程1, ..., $k+1$，所以s'和s''都是从s开始$k+1$可到达的。而且，容易看出在s'中$k+1$个变量被覆盖：X中的k个变量加上进程$k+1$覆盖的新变量x。另外，根据s_2'的定义，$s''=s_2'$是一个空闲状态。

剩下的工作是证明s'和s''对于所有进程i, $k+2 \leqslant i \leqslant n$，是不可区分的。但是$s_2$和$s_2'$的定义意味着对于所有进程$i$, $k+1 \leqslant i \leqslant n$来说，$s_2$和$s''=s_2'$是不可区分的。另外，我们已经注意到$s_2$和$s'$对于除了进程$k+1$之外的进程都是不可区分的。综合这两个事实可得所需条件。□

现在来证明定理10.33。

（定理10.33的）证明　采用反证法，假设算法A使用最多$n-1$个读－写共享变量来解决$n \geqslant 2$个进程的互斥问题。不失一般性，假设A有且只有$n-1$共享变量。

308　令s为A的任意初始系统状态。则引理10.37意味着存在两个从s开始$n-1$可到达的系统状态s'和s''，使得所有$n-1$个共享变量被s'中的进程1, ..., $n-1$覆盖，而s''是一个满足 $s' \overset{n}{\sim} s''$ 的空闲状态。引理10.30意味着存在一个从s'开始且只涉及进程n的步的运行片断，其中进程n到达C。引理

10.36意味着在这个运行片断中，进程n必定写入某个在s'中不被覆盖的变量。但是在s'中所有$n-1$个变量都被覆盖，从而产生矛盾。 □

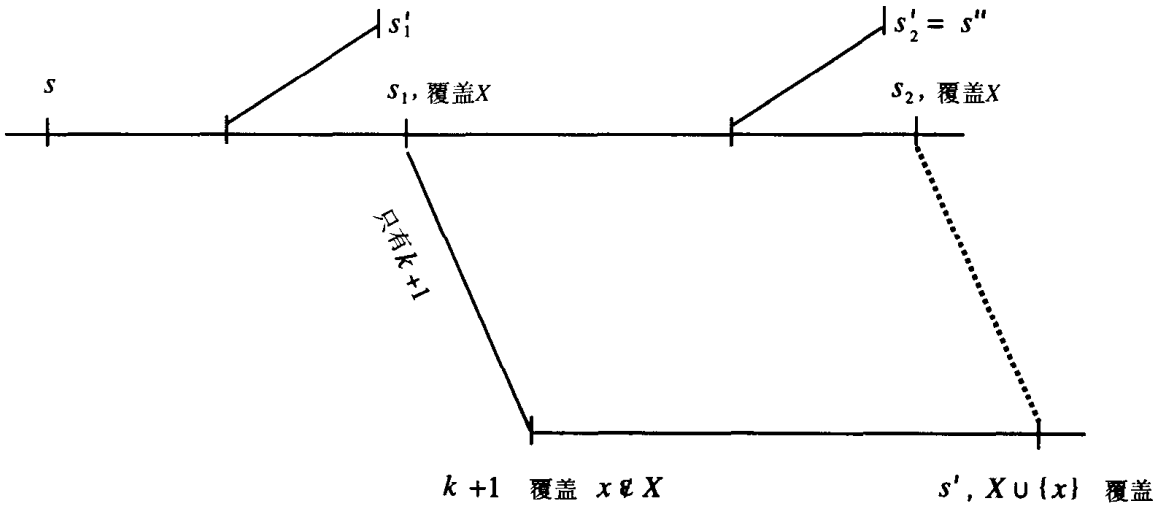


图10-13 一般情况的构造

我们再次强调不管共享变量的尺寸大小如何定理10.33都成立：共享变量的尺寸大小甚至可以是无界的。另外，这里并不需要高层次公平性假设，演进性条件是这个不可能性结果所需的惟一活性条件。

10.9 使用读-改-写共享变量的互斥

本节考虑使用读-改-写共享存储器的互斥问题。也就是说，在一个瞬时步中，进程可以访问一个共享变量并使用变量的值和进程状态来决定变量的新值和新进程状态。9.4节中给出了形式化定义。

你可能认为在读-改-写模型中考虑互斥问题是很容易的，因为这种模型相当强大。读-改-写模型为每个共享变量都提供了公平互斥的访问——每个进程都得到公平机会来访问变量，且在释放变量前可以执行任意计算。这非常接近于公平互斥算法的要求，即对临界区的公平排他访问。这看起来正是我们尝试解决的问题的一个解。

实际上，拥有如此强大的共享存储器确实令情况大大简化，但并没有令所有困难都消失。除了一组算法之外，我们还给出一些非平凡的下界结果。

我们首先考虑基本互斥问题，然后考虑当加入高层次公平要求（有界绕过次数或锁定权）时的情形。

在本节的余下部分，我们假设共享存储器系统只包含一个共享变量。在读-改-写模型中也不失一般性，因为几个读-改-写变量毕竟可以合并为单个多部分读-改-写变量。这与 309
读-写模型不同，对于读-写模型，我们已经证明了互斥问题解法的存在与否对读-写变量的个数是很敏感的。

10.9.1 基本问题

为了看出读-改-写模型和读-写模型的不同，考虑以下简单的单变量算法——*TrivialME*算法。在这个算法中，如果资源已经被分配给某个进程，则共享变量x的值为1。任意在尝试区中的进程都简单地测试x的值直至它发现$x = 0$，之后它立刻设置$x := 1$。在退出前，进程重新设置$x := 0$。容易看出*TrivialME*算法解决了互斥问题。

TrivialME algorithm :

Shared variables:
$x \in \{0,1\}$, initially 0

Actions of i:
Input: try_i, $exit_i$
Internal: $test_i$, $reset_i$
Output: $crit_i$, rem_i

States of i:
$pc \in \{rem, test, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially rem

Transitions of i:

try_i
Effect:
$pc := test$

$test_i$
Precondition:
$pc = test$
Effect:
if $x = 0$ then
$x := 1$
$pc := leave\text{-}try$

$reset_i$
Precondition:
$pc = reset$
Effect:
$x := 0$
$pc := leave\text{-}exit$

$crit_i$
Precondition:
$pc = leave\text{-}try$
Effect:
$pc := crit$

$exit_i$
Effect:
$pc := reset$

rem_i
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

310

定理10.38 *TrivialME*算法解决了互斥问题。

10.9.2 有界绕过次数

*TrivialME*算法不保证任意高层次公平性条件。然而，我们仍然可以使用单个共享变量很容易获得非常强的高层次公平性条件，甚至可以获得FIFO条件（基于每个进程在其尝试区中执行的第一个局部控制步）。例如，我们有*QueueME*算法。

QueueME算法（非形式化）:
进程在共享变量中保持一个进程索引队列，该队列初始时为空。进入T的进程把其索引添加到队列的尾部；发现自己在队列头部的进程进入C；在一个进程离开C时，它把其索引从队列中删除。以下用前提－结果风格来更形式化地表示这个算法。

QueueME algorithm (formal):

Shared variables:
queue, a FIFO queue of process indices, initially empty

Actions of i:

Input:	Internal:
try_i	$enter_i$
$exit_i$	$test_i$
Output:	$reset_i$
$crit_i$	
rem_i	

States of i:

$pc \in \{rem, enter, test, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially rem

Transitions of i:

311

try_i
- Effect:
  - $pc := enter$

$enter_i$
- Precondition:
  - $pc = enter$
- Effect:
  - add i to $queue$
  - if i is first on $queue$ then
    - $pc := leave\text{-}try$
  - else $pc := test$

$test_i$
- Precondition:
  - $pc = test$
- Effect:
  - if i is first on $queue$ then
    - $pc := leave\text{-}try$

$crit_i$
- Precondition:
  - $pc = leave\text{-}try$
- Effect:
  - $pc = crit$

$exit_i$
- Effect:
  - $pc := reset$

$reset_i$
- Precondition:
  - $pc = reset$
- Effect:
  - remove first element of $queue$
  - $pc := leave\text{-}exit$

rem_i
- Precondition:
  - $pc = leave\text{-}exit$
- Effect:
  - $pc := rem$

容易看出*QueueME*算法保证良构性、互斥性和演进性。而且它还满足高层次公平性条件，临界区的进入是根据每个进程在其尝试区（*enter*动作）中执行的第一个局部控制步、以FIFO的顺序进行。这意味着*QueueME*算法保证有界绕过次数（界限为1）。

定理10.39 *QueueME*算法解决了互斥问题且保证有界绕过次数。

*QueueME*算法简单且快，至少根据我们的时间衡量方法，它没有共享变量过大这一问题。由于$n!+(n-1)!+\cdots$不同的队列包含最多n个不同索引，所以变量有许多不同的值。变量的长度最少需要$\Omega(n \log n)$位。最好能够减少共享变量的尺寸，这不但出自节省共享存储器的需要，还因为对如此大的变量进行瞬间访问是不合理的。一个有意思的问题是需要多大的共享存储器以保证高层次公平性。当变量的个数线性于n的时候能否解决这个问题？如果只使用常数个值呢？ 312

不难使用一个只有n^2个值（$2\log n$位）的变量来获得在*QueueME*算法中所表现的相同类型的FIFO行为。例如，我们可以使用一个基于为临界区派发“票”的算法。

*TicketME*算法（非形式化）：

共享变量保存一个(*next*, *granted*)对，初值为(0, 0)，其中*next*和*granted*的值在{0, ..., $n-1$}中。*next*部分代表将要发给进程的、允许进程进入临界区的下一张“票”，而*granted*部分代表已经授权许可进入临界区的最后一张“票”。当一个进程进入尝试区时，它“拿到一张票”，也就是说，它以*next*部分的值作为票号，并将*next*部分的值加1。如果一个进程的票号等于*granted*部分的值，则它进入临界区。如果进程离开，则它将*granted*部分的值加1，并对n取模。

现在我们给出前提－结果风格的代码。

***TicketME* algorithm (formal):**

Shared variables:
$(next, granted)$, a pair of elements of $\{0, \ldots, n-1\}$, initially $(0, 0)$

Actions of i:

Input:	Internal:
try_i	$enter_i$
$exit_i$	$test_i$
Output:	$reset_i$
$crit_i$	
rem_i	

States of i:
$pc \in \{rem, enter, test, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially *rem*
$ticket \in \{0, \ldots, n-1\} \cup \{null\}$, initially *null*

Transitions of i:

try_i
　Effect:
　　$pc := enter$

$enter_i$
　Precondition:
　　$pc = enter$
　Effect:
　　$ticket := next$
　　$next := next + 1 \bmod n$
　　if $ticket = granted$ then
　　　$pc := leave\text{-}try$
　　else $pc := test$

313

$test_i$
　Precondition:
　　$pc = test$
　Effect:
　　if $ticket = granted$ then
　　　$pc := leave\text{-}try$

$reset_i$
　Precondition:
　　$pc = reset$
　Effect:
　　$granted := granted + 1 \bmod n$
　　$ticket := null$
　　$pc := leave\text{-}exit$

$crit_i$
　Precondition:
　　$pc = leave\text{-}try$
　Effect:
　　$pc = crit$

rem_i
　Precondition:
　　$pc = leave\text{-}exit$
　Effect:
　　$pc := rem$

$exit_i$
　Effect:
　　$pc := reset$

---

*TicketME*算法满足*QueueME*算法的正确性条件，包括基于每个进程在其尝试区中执行的第一个局部控制步的FIFO属性。以下定理的证明出现在10.9.4节中。

定理10.40 *TicketME*算法使用n^2个共享存储器值来解决互斥问题且保证有界绕过次数。

能否有比n^2更好的算法？以下定理给出解决互斥问题（保证有界绕过次数）所需的共享变量值数目n的一个简单下界。

定理10.41 令A为一个使用单个读-改-写共享变量并保证有界绕过次数的n进程互斥算法，则变量的不同值个数最少为n。

证明 设A为一个保证有界绕过次数的n进程互斥算法，且其绕过数量界限为a。不失一般性（如同10.8节中的情况），设用户U_i是可能是最不确定的。我们采用反证法来证明，即构造一个其中某个进程绕过数量次数多于a的运行。

我们先定义一系列有限运行$\alpha_1, \alpha_2, \ldots, \alpha_n$，其中每个运行都是前一运行的扩展。运行$\alpha_1$通过令进程1单独地从一个初始系统状态一直运行到进入C而得。（演进性条件意味着这是可能的）为了得到α_2，我们通过令进程2进入尝试区并执行一个局部控制步来扩展α_1。显然，为了避免 314
违反互斥条件，进程2在α_2之后停留在尝试区中。然后以类似方式构造α_i, $3 \leq i \leq n$：从α_{i-1}的尾部开始，令进程i进入尝试区并执行一个局部控制步。每个进程i, $3 \leq i \leq n$也停留在尝试区中。

把α_i, $1 \leq i \leq n$之后的系统状态和共享变量的值分别记为s_i和v_i。我们来证明对于$1 \leq i, j \leq n$, $i \neq j$，有$v_i \neq v_j$，从而证明所需结论。

采用反证法，假设对于特定的i和j, $i \neq j$，$v_i = v_j$成立，且不失一般性，设$i < j$，则对于任意进程k, $1 \leq k \leq i$，$s_i \overset{k}{\sim} s_j$成立。（也就是说，对于进程和用户1, ..., i，α_i和α_j之后的系统状态包括了相同的状态和相同的共享变量值。）

现在，存在某个只涉及进程1, ..., i的扩展了α_i的低层次公平运行，使得某个进程无数多次进入C。这由演进性假设（只适用于低层次公平运行）可得。可以在α_i之后应用同样的步，则又产生一个运行，其中同一进程无数多次进入C。注意这个新的运行不是低层次公平的：进程$i+1$, ..., j在α_j之后的运行部分中不再执行任何步，即使它们全部在T中。但这并不要紧：低层次公平性不是造成对绕过数量界限的违反的必要条件。运行这个运行中足够大的部分就足以使得进程j的绕过次数比某个其他进程的要多a次，这就产生了所需的矛盾。 □

图10-14显示构造过程。

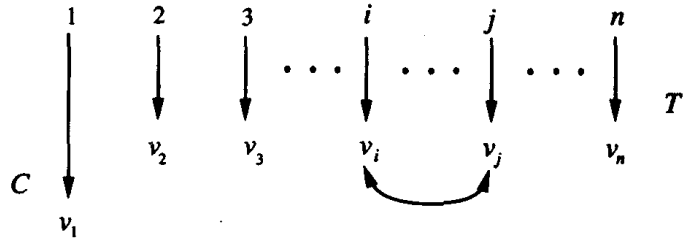


图10-14 定理10.41的证明中运行的构造

这个界限是否严格呢？能否提高这个界限，例如提高到$\Omega(n^2)$？这是不成的——存在一个*反例算法*（即本身并不有意思的算法，它既不实用，也不优雅，但是能够作为一个不可能假设的 315
反例），它只需$O(n)$个值。实际上，这个算法只需要$n+k$个值，这里的k是一个小常数。我们把

这个算法称为*BufferMainME*算法，命名原因很快就会出现。

BufferMainME算法：

*BufferMainME*算法的基本思想如下。尝试区被分为缓冲区和主区两部分。当一个进程进入尝试区时，它进入缓冲区；缓冲区并不保存进程的顺序信息。有时，当主区变空时，缓冲区中的所有进程都进入主区，然后清空缓冲区。从主区开始，进程以任意顺序逐个进入临界区。

实现这个思想需要某些通信机制，使得进程能够知道何时去变换区域。在第一个具体实现中，除了一般的“代理”进程1, ..., n，我们允许存在一个总是执行步的专一主管（*supervisor*）进程。我们来设计一个方法，把系统控制集中在主管进程上：主管对进程应该何时变换区域进行跟踪，并相应地通知它们。之后，我们描述了不使用特定主管进程的实现。

使用主管进程，我们可得以下性质。首先，令变量包括两部分：计数值*count* $\in \{0, ..., n\}$，和从指定控制消息有限集中选出的消息*message*。对于某个恒定整数k，变量共有kn个值，但我们可以使用优先权机制，允许针对通信的不同类型来复用变量，从而把值的个数优化为$n + k$。

主管拥有局部变量*buffer-count*和*main-count*，分别用来保存它听道的处于缓冲区和主区中的进程的数目。当一个进程进入尝试区时，它把共享变量中*count*部分的值加1，以通知主管某个新进程已经进入，然后这个进程在缓冲区中等待。只要主管看到在共享变量中有一个非零*count*，它就把这个*count*值提取到自己的*buffer-count*中，并把变量的*count*部分的值清0。

主管能够计算出应该在什么时候把缓冲区中的进程移到主区中，也就是说，在它的*main-count*为0之后把缓冲区中的进程移到主区中。它通过把*enter-main*消息放在共享变量的*message*
316 部分来移动这些进程，每次移动一个。当主管看到它的*buffer-count*和变量中的*count*值都等于0的时候，它停止把进程从缓冲区移到主区的工作。然后主管通过把*enter-crit*消息放在共享变量的*message*部分中，把进程从主区移到临界区。

所用的控制消息为

- *enter-main*：主管把这个信息放到共享变量的*message*部分中，从而把进程从缓冲区移到主区。缓冲区中看见这个消息的第一个进程捡起这个消息并前进到主区。
- *ack-main*：从共享变量中捡起*enter-main*消息的进程把*ack-main*信息作为对主管的确认。主管捡起这个消息。
- *enter-crit*：主管把这个消息放到共享变量的*message*部分中，从而把进程从主区移到临界区。主区中看见这个消息的第一个进程捡起这个消息并前进到临界区。
- *ack-crit*：从共享变量中捡起*enter-crit*消息的进程把*ack-crit*消息作为对主管的确认。
- *done*：在临界区中存在的进程把这个消息放到共享变量的*message*部分中，从而宣布它已经完成工作了。

现在我们来说明怎样避免在共享变量中出现两个独立部分的情况。注意变量有以下两个目的：记录新近进入的进程的数目和负责控制消息的通信。我们将“分时使用”这个变量，使得它满足这两个目的，但不是同时满足。在任一点中变量的值或者是一个*count*消息，或者是一个控制消息，但这两个信息不会同时出现。

注意，在目前为止描述的算法中，控制信息的通信根据一个单独串行“控制线程”来进行，参见图10-15。

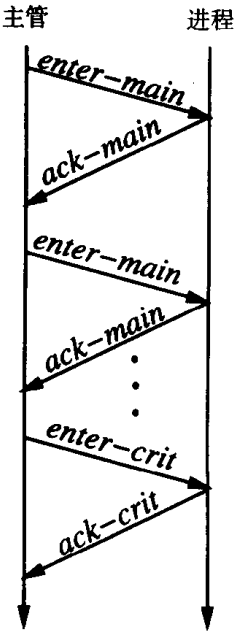


图10-15　有界绕过次数互斥算法的控制线程

假设这条线被新进入的进程i打断，其中进程i把一个控制消息

改写成为一个计数值（计数值1，用于宣布进程自己的到达）。然后（因为只有有限多的进程能够进入系统），系统最终到达一个稳定状态。主管最终吸入共享变量中的所有*count*信息，使得变量中的*count*值永远为0。在这一点中，进程*i*可能会把重写后的消息放回变量中，从而允许控 317
制线程延续下去。

更确切地说。当进程*i*进入尝试区并看到共享变量中的一个控制消息时，它记住这个消息并以计数值1来代替这个消息。进程*i*一直保持这个消息，直至它看到*count*为0，之后它以记住的消息来重写0。假设专用主管进程可用的情况下，结果为一个互斥算法，其有界绕过次数使用了$n+6$个共享变量值。

现在修改这个算法，使它可以在我们已经研究过的模型中工作——即不需要一个专用主管进程。思想在于允许进程协作地分布式模拟主管（模拟是分布式的，由于没有进程能够保证一直可用）。进程简单地执行步来进行模拟；特别的，只要进程处于临界区中，它将作为模拟主管的进程。

模拟的主要难点是离开C的进程必须把主管模拟的任务传递给下一进入C的进程。这涉及 318
到把主管需要的所有状态信息（特别是*buffer-count*和*main-count*的值）通知给下一进程。我们必须针对这种新通信类型和其他两种已经讨论过的通信类型来使用共享变量。我们再次使用分时技术。注意新的状态通信不和控制消息通信同时进行，所以现在没有干扰发生。另外，我们可以采用对计数消息和控制消息之间干扰的管理方法，来管理新到达进程的计数消息和状态信息的通信之间的干扰。

最后一个细节：有时，当进程离开C时，不存在其他进程负责传递主管模拟。此时，没有进程处于尝试区中。但这时在主管状态中没有有意思的信息，所以进程能够放弃责任，并在共享变量中留下一个特定标志。

定理10.42 *BufferMainME*算法使用单个读－改－写共享变量来解决了互斥问题，并保证有界绕过次数，其中的共享变量只有$n+k$个值（k为某个小常数）。

10.9.3 锁定权

只有当高层次公平要求是有界绕过次数时，定理10.41的下界才成立。对于更弱的锁定权要求，以上证明便行不通了。问题在于锁定权是低层次公平运行的一个属性，而证明中构造的坏运行对进程$i+1, \ldots, j$并不公平。事实上，定理10.41对锁定权并不成立。我们给出另一个反例算法，这个算法具有一个小的令人吃惊的界限。我们把它称为*Executive*算法。

***Executive*算法：**

算法的思想如下。如同*BufferMainME*算法中的情况，每个到来的进程都将共享变量中的*count*值加1，但这一次*count*的值在折回0之前只能为$0, \ldots, \frac{n}{2}$。与前面一样，*count*被一个（模拟）主管提取。

当*count*折回0的时候，$\frac{n}{2}+1$个进程对系统的余下部分暂时“隐藏”起来；对于所有其他进程来说，此时的系统状态与这些隐藏进程都进入之前的系统状态是不可区分的。如果这些隐藏 319
进程不执行更多的步，则系统的余下部分继续前进，好像所有隐藏进程仍然在R中。然而，在一个低层次公平运行中，隐藏进程会执行更多的步，系统终会知道它们的存在。

为了确保进程不会一直隐藏，我们把执行了从$\frac{n}{2}$到0的迁移的进程指定为执行官（*executive*），它负责管理隐藏进程。执行官发送特别的*sleep*消息给处于缓冲区中的(任意)$\frac{n}{2}$个进程，让它们

休眠一段时间。接着，把$\frac{n}{2}$个进程从竞争中去除之后，执行官重新进入系统，再次代表自己把*count*值加1。现在，$\frac{n}{2}$个进程在休眠，算法的运行正像*BufferMainME*算法一样；共享变量现在不会溢出，因为最多剩下$\frac{n}{2}$个活动进程，而变量中的计数值允许为$\frac{n}{2}$。特别的，不会建立第二个并发的执行官。根据*BufferMainME*算法的行为可知，执行官最终到达*C*。

当执行官到达*C*时，它发送*wakeup*消息给休眠进程并通知主管。我们必须再次针对这些新通信类型来分时使用变量，不过这次需要一个稍微复杂些的优先级方案。

定理10.43　*Executive*算法使用单个读－改－写共享变量来解决了互斥问题，并保证锁定权，其中的共享变量只有$\frac{n}{2}+k$个值（k为某个小常数）。

本小节的结尾部分给出锁定权互斥条件所需的值数目的一个约为$\sqrt{n}$的下界。根据$\frac{n}{2}+k$个上界可知，这个下界并不严格，但其证明中包含了一个如何构造坏低层次公平运行的方法。

定理10.44　令$k\geq 2$。令*A*是使用单个读-改-写共享变量、解决了互斥问题、保证锁定权、包含$n \geq \frac{k^2-k}{2}+1$个进程的任意系统，则在*A*的可达状态中变量能取的不同值最少为k。

证明　我们再次假设用户是尽可能不确定的。证明以对k的归纳来进行。

基础：$k=2$。则定理描述中的不等式说明$n\geq 2$。容易证明变量必然取最少两个值，否则
320 进程无法通信。正式证明类似于引理10.31中的证明。

归纳步：假设结果对$k\geq 2$成立，以下证明对k+1也成立。设$n\geq \frac{(k+1)^2-(k+1)}{2}+1$，采用反证法，设共享变量的值数目严格小于$k$+1。根据归纳假设，值数目最少为$k$，故此值数目等于$k$。现在构造一个坏运行来导出矛盾。

通过一直运行进程1直至它进入*C*来定义确定运行α_1，结果系统状态记为s_1。然后通过运行进程2直至到达系统状态s_2来将α_1扩展成为α_2，其中在s_2中共享变量有一个值，进程2能够通过从状态s_2开始运行来令这个值重复无数次。由于变量只有有限个值，这样的状态必定存在。类似的，通过在α_{i-1}之后运行进程i直至到达系统状态s_i来定义α_i，其中$3\leq i\leq n$，其中在s_i中共享变量有一个值，进程i能够通过从状态s_i开始运行来令这个值重复无数次。对于所有i，$1\leq i\leq n$，令v_i是在状态s_i中变量的值。由于共享变量只能取k个值，根据鸽巢原理，必有两个进程i和j，其中$n-k\leq i<j\leq n$，使得$v_i=v_j$。固定这个i和j。注意对于所有进程m，$1\leq m\leq i$，有$s_i \overset{m}{\sim} s_j$。

现在，进程1, ..., i组成了一个最少包括$\frac{k^2-k}{2}+1$个进程的系统，该系统解决了锁定权的互斥问题。所以，根据归纳假设，它们必然使用了共享存储器的所有k个值。事实上，我们可以更精确地描述这个声明：对于每个从系统状态s_i开始i可到达的[⊖]系统状态s和共享存储器的每个值v，必然存在一个从s开始i可到达的系统状态，其中共享变量的值为v。（否则，对于一个涉及进程1, ..., i的系统，其中变量取值少于k个，我们可以使用任一从s开始i可到达的空闲状态作

⊖　在10.8节，我们定义状态，i是可达的，即只使用进行1，…，n步。

为开始状态）。使用这种描述，我们可以生成一个扩展了α_i的、涉及进程1, ..., i的低层次公平运行，其中共享变量的所有k个值重复无限多次。

现在来构造所需的坏运行α。α以α_j开始，将进程1, ..., j 带到系统中，并将系统状态带到s_j中，而变量的值$v_j = v_i$。然后，按照上述方法运行进程1, ..., i，不过这次从状态s_j而不是状态s_i开始；这些进程再次导致共享变量的所有k个值都重复无数次。

现在回想从s_m中的局部状态开始，每个进程m, $i+1 \leq m \leq j$能够导致状态s_m中的变量的值重复无数次。然后我们将进程1, ..., i的主运行和进程$i+1$, ..., j的某些步按以下方式连接：在主运行中共享变量每次都被设为某个值v_m, $i+1 \leq m \leq j$，令进程n运行恰好多步（最少一步），使得它把共享变量的值返回给v_m。这些插入产生了一个无限运行，它对所有进程都是公平的，且所有进程m, $i+1 \leq m \leq j$被锁定。 □

321

定理10.44的证明中的关键思想是通过连接运行片断来构造坏的低层次公平运行。而且，注意定理10.44乍看之下将导致一个悖论。也就是说，解决锁定权互斥问题需要非平凡的内在代价，即使我们的模型已经包含与目标很相近的东西——任意计算对共享变量的公平排他访问。

注意在定理10.43的上界和定理10.44的下界之间的差异。如何缩小这个差异可作为一个研究问题。

10.9.4 模拟证明

作为结束，本节给出对前节介绍的*TicketME*算法的正确性证明。证明中用到了8.5.5节中描述的模拟方法。我们已经使用这种模拟方法来证明了同步模型中几个算法的正确性——例如，*OptFloodSet*算法的正确性；然而，这是我们首次使用这个方法来证明异步算法的正确性。

我们希望证明*TicketME*算法保证了*QueueME*算法所保证的正确性条件：良构性、互斥性、演进性和基于T中第一个局部控制事件的FIFO行为。证明了这些属性之后，定理10.40就成立了。一种了解*TicketME*算法的好方法是将它和一个新的*InfiniteTicketME*算法（而不是*QueueME*算法）联系起来。除了使用一个无限票队列而不是对n取模之外，*InfiniteTicketME*算法与*TicketME*算法是相同的。那么*TicketME*算法可被视为*InfiniteTicket*算法的低复杂度版本。为了得到*InfiniteTicketME*算法，需对*TicketME*算法做以下修改：

***InfiniteTicketME* algorithm:**

Shared variables:
$(next, granted) \in \mathbb{N} \times \mathbb{N}$, initially $(0,0)$

Actions of i:
As for *TicketME*.

322

States of i:
$ticket \in \mathbb{N} \cup \{null\}$, initially *null*

Transitions of i:

enter$_i$
  Precondition:
    $pc = enter$
  Effect:
    $ticket := next$
    $next := next + 1$
    if $ticket = granted$ then
      $pc := leave\text{-}try$
    else $pc := test$

reset$_i$
  Precondition:
    $pc = reset$
  Effect:
    $granted := granted + 1$
    $ticket := null$
    $pc := leave\text{-}exit$

容易证明*InfiniteTicketME*算法满足*TicketME*算法的所有相关属性，原因在于它每次只能派出一张票，且票不能被重用。然后我们可以使用模拟方法将*TicketME*算法和*InfiniteTicketME*算法正式联系起来，以证明*TicketME*算法的正确性。思想是令两个算法一起运行，证明两种运行之间的特定关系。

*InfiniteTicketME*算法的某些不变式是有用的。（在证实算法属性的过程中自然会证明它们。）

引理10.45　在*InfiniteTicketME*算法的任意可达系统状态中，以下成立：

1）如果$pc_i \in \{test, leave\text{-}try, crit, reset\}$，则进程$i$拥有一张非空票。

2）非空票的票号是在区间$[granted, next)$内的整数，且每个值都只由一个进程拥有。

3）$granted \leq next \leq granted + n$。

4）如果$pc_i \in \{leave\text{-}try, crit, exit\}$，则$ticket_i = granted$。

下一步是当两个算法和同一用户集合进行组合时，定义*TicketME*和*InfiniteTicketME*的系统状态之间的一个模拟关系f。对应关系相当简单：如果除了对应的票号取模n后是相同的外，对应状态的其他也是完全相同的，则定义$(s, u) \in f$（或者写成$u \in f(s)$）。我们使用以下的点记号
323 来指出给定状态中给定变量的值。

1）在s和u中所有用户状态都是相同的。

2）对于每个i，$s.pc_i = u.pc_i$。

3）$s.granted = u.granted \bmod n$。

4）$s.next = u.next \bmod n$。

5）对于每个i，$s.ticket_i = u.ticket_i \bmod n$。

现在来证明f是一个模拟关系。更精确的说，定义T和I分别为*TicketME*系统和*InfiniteTicketME*系统，其中每个系统都被稍微修改过，使得所有动作都被归为外部动作。我们来证明f是一个从T到I的模拟关系。我们所需的两个条件为：

1）如果s是T的一个初始状态，则$f(s)$包含I的一个初始状态。

2）如果s是T的一个可达状态，$u \in f(s)$是I的一个可达状态，且(s, π, s')是T的一个迁移，则存在I的一步(u, π, u')，其中$u' \in f(s')$。

引理10.46　f是一个从T到I的模拟关系。

证明　很容易证明上面给出的两个条件。对于第一个条件，亦即开始条件，T的一个开始状态由*TicketME*算法的惟一一个开始状态和用户的任意开始状态组成。容易看出*InfiniteTicketME*算法的惟一一个开始状态和用户有相同开始状态，并且都在$f(s)$中。

第二个条件，也就是步条件，则可以根据所执行的动作的类型分情况讨论。用户的任意局部控制动作都被精确模仿。对于算法的每个局部控制动作，步(s, π, s')在T中的存在立刻意味着在I的相应状态中同一动作π是使能的，原因在于使能条件都基于pc的值。另外，在每种情况中，新状态u'由*InfiniteTicketME*的定义惟一地决定。剩下要证明的是$u' \in f(s')$。

但这很容易证明。惟一有意思的动作是形如$test_i$的动作，其中进程i基于$ticket_i = granted$是否成立来决定是否应该进入C。我们必须证明两个算法不会做出不同决定。因为只使用了票值$0, \ldots, n-1$，且在s和u中的对应值对m取模后是相同的，做出不同决定的惟一方法是上述等式在*TicketME*中成立但在*InfiniteTicketME*中不成立。也就是说，危险来自于票值增大并对n取模后
324 可能会导致某些差异变得模糊，而这些差异对于*InfiniteTicketME*的行为的决定来说是重要的。

但是这个问题其实不会出现。假设$s.ticket_i = s.granted$，则$u \in f(s)$的事实意味着$u.ticket_i =$

$u.granted \bmod n$。引理10.45中证明的不变式意味着$u.granted \leq u.ticket_i < u.next$和$u.next \leq u.granted + n$。因此，$u.granted \leq u.ticket_i \leq u.granted + n$。故有$u.ticket_i = u.granted$，得证。□

那么如何使用引理10.46来证明*TicketME*算法的正确性呢？

（定理10.40的）证明概要 引理10.46和定理8.12意味着$traces(T) \subseteq traces(I)$。良构性、互斥性和FIFO条件都可以表达成为轨迹属性（当所有动作都被包括进来时，正如这里所做的）。所以这三个条件对I成立的事实意味着它们对T也成立。可得*TicketME*算法保证良构性、互斥性和FIFO条件。

但这还没有证明*TicketME*算法保证演进性。演进性条件与上述三个条件不同，它只对算法的公平运行成立。为了将演进性条件从*InfiniteTicketME*算法移到*TicketME*算法中，我们需要知道$fairtraces(T) \subseteq fairtraces(I)$。模拟关系$f$能够用来证明这个结论。

关键思想在于模拟关系并不只包含轨迹集合，它实际上建立两个算法的运行之间的紧密对应关系。这种对应关系的形式化定义见8.5.5节。目前，定理8.13和8.5.5节意味着对于T的任意运行α，存在I的一个运行α'按照以下形式与之对应：

1）α和α'中的动作序列相同。

2）α和α'中的同一位置的状态以f关联。

因为T和I的所有动作都是外部动作，所以我们得到这种强对应关系。

现在来证明$fairtraces(T) \subseteq fairtraces(I)$。令$\beta \in fairtraces(T)$，而$\alpha$是满足$\beta = trace(\alpha)$的$T$的任一公平运行。则根据定理8.13，存在满足以上两个条件的I的一个对应运行α'。特别的，α和α'的轨迹是相同的，所以$\beta = trace(\alpha')$。我们来证明α'是I的一个公平运行。

有两种情况可能导致不公平的发生。在第一种情况中，可能从α'中的某点开始，某个进程I是使能（以执行局部控制步）的，但在该点之后这种步却没有发生过。那么强对应关系意味 325
着进程i从α中的同一点开始是使能的，但在α中该点之后这种步却没有发生过。这违反了α的公平性，从而产生矛盾。在第二种情况中，可能某个用户任务从α'中的某点开始是使能的，但在该点之后这种任务却没有发生过。同样的，强对应关系意味着α中发生了同样的情况，从而违反了α的公平性。

所以α'是I的一个公平运行，可得$\beta \in fairtraces(I)$。因此，$fairtraces(T) \subseteq fairtraces(I)$。由于演进性条件可以表示成公平性轨迹的属性（当所有动作都被包括进来时，正如这里所做的），故演进性条件从*InfiniteTicketME*算法移到*TicketME*算法中。□

10.10 参考文献注释

*DijkstraME*算法出现在Dijkstra[90]中。它把之前的Dekker双进程算法扩展为适用于任意数目的进程的情况。在这些结果出现之前，大家甚至并不清楚能否只使用读－写共享存储器来解决互斥问题。参照Goldman和Lynch关于共享存储器模型的论文[141]，我们采用断言式证明*DijkstraME*算法满足互斥条件。Knuth、de Bruijn和Eisenberg以及McGuire [168, 86, 108]给出了Dijkstra算法的一系列扩展，后一个扩展通过加入新的高层次公平条件和/或更好的性能属性来对前一个扩展进行提高。

*Peterson2P*和*PetersonNP*算法是由Peterson[238]设计的。锦标赛算法是基于*Peterson2P*算法的思想以及Peterson和Fischer[242]的锦标赛协议思想的。我们的锦标赛算法比[242]中的锦标赛算法更为简单，且其正确性更容易证明；然而，其缺点是它使用了多写者变量，而[242]中的算法只使用了单写者变量。

*BurnsME*算法归功于Burns[60]，而*Bakery*算法归功于Lamport[174]。Lamport的后一篇论文[180]包含了额外的改进互斥算法。解决互斥问题所需的寄存器数目的下限由Burns和Lynch[63]提出。

*TicketME*算法归功于Fischer、Lynch、Burns和Borodin[120, 121]。绕过次数界限和针对读 - 改 - 写共享存储器的锁定权互斥算法都出现在Burns、Fischer、Jackson、Lynch和Peterson[62]的一篇论文中。这些结果建立在Cremers和Hibbard[84]的早期结果上；特别地，*BufferMainME*
326 算法类似于[84]中的算法。本章没有讨论[62]中给出另外一个结果，即假设进程只有单个剩余区状态时，锁定权互斥需要最少$n/2$个共享存储器值。（也就是说，它们不保存算法以前运行中的任意存储器值）。Cremers和Hibbard[85]也设计了一个$n+k$算法以取得对临界区的FIFO访问，该算法使用读 - 改 - 写共享存储器。

Manna和Pnueli[219]的书是时序逻辑方面极好的参考书。时序逻辑能够用来对本章和其他章中的活性条件进行形式化。

Fischer、Lynch、Burns和Borodin[120]首先定义了习题10.13中考虑的k互斥问题，之后Shavit[261]研究了这个问题。Raynal[249]的一本书包含许多针对异步共享存储器模型和异步网络模型的互斥算法的描述。Ben-Ari[45]的书和Peterson与Silberschatz[262]的书中也讨论了互斥问题。

10.11　习题

10.1 考虑另一种根据共享存储器系统A的轨迹而不是根据A和用户的组合来定义互斥问题的方法。也就是说，定义一个轨迹属性Q，使得$traces(Q)$是由满足如下三个条件的序列β组成的集合，其中β是由*try*, *crit*, *exit*和*rem*动作组成的序列。

a）在β中，系统不会首先违反对于任意i的良构性；

b）如果β对于每个i都是良构的，则β不包含没有干涉事件*exit*存在的两个*crit*事件；

c）如果β对于每个i都是良构的，那么以下两点成立：

i）如果在β的某点中，某个进程的最后事件是*try*，且没有进程的最后事件是*crit*，那么之后必有一个*crit*事件。

ii）如果在β的某点中，某个进程的最后事件是*exit*，那么之后必有一个*rem*事件。

证明，如果$fairtraces(A) \subseteq traces(Q)$，则与任意一组用户组合起来的$A$满足10.2节给出的互斥问题的定义。

327 10.2 定义*DijkstraME*算法的一个公平运行，其中一个特定进程发生锁定。

10.3 证明：为了正确地解决互斥问题，*DijkstraME*算法的第二阶段（其中*flag*变为2且其他进程的*flag*被测试）是必需的。

10.4 填入对*DijkstraME*算法互斥性的归纳证明中的细节。

10.5 考虑*DijkstraME*算法的计时分析，即从某个用户处于T中但没有用户处于C中的一点到其中某个用户进入C的一点为止的时间。

a）以形式$k_1 n\ell + k_2 \ell$来表示界限，其中k_1和k_2是特定的常数。令k_1和k_2尽可能小。

b）构造*DijkstraME*算法的一个运行，其时间界限尽可能大；尽量与你计算出来的上界匹配。

10.6 锁定权条件只对于那些保证良构性、但不一定保证互斥性或演进性的算法才有意义。请详细证明如果一个算法保证良构性和锁定权（对于所有用户集合），则它同时也保证演进性（对于所有用户集合）。

10.7 修改*Peterson2P*算法中的进程，使之无需严格交替执行*check-flag* 和*check-turn*，而是遵守

某个更松的规则。确定你的最终算法仍然是一个锁定权互斥算法。证明该算法的正确性，并分析其时间复杂度。

10.8 为两个仅使用单写者/多读者读－写寄存器的进程设计一种锁定权互斥算法。证明算法的正确性，且最好使用不变式断言来证明。(提示：如果感到困难，你可以参考[242]中的双进程解法。但是你必须给出自己的不变式证明。)

10.9 证明断言10.5.3。

10.10 证明断言10.5.4。

10.11 对于*PetersonNP*算法，重新考虑定理10.16中已经证明出来的时间界限。这个界限是否严格？要么明确给出一个运行，其中出现了定理10.16中描述的指数式行为；要么详细分析出一个更小的复杂度界限。 328

10.12 *PetersonNP* 算法是否保证有界绕过？给出证明或反例。

10.13 修改*PetersonNP*算法以得出k互斥问题的解法，其中$2 \leqslant k \leqslant n$。这个问题允许$k$个进程在临界区中同时共存。正式的情况下，互斥条件被修改为禁止多于k个用户同时处于C中。对于尝试区的演进性也被修改：如果至少有一个用户在T中，至多有$k-1$个用户在C中，则某个用户最终进入C。证明此算法的正确性。详细阐述该算法满足的高层次公平条件。

10.14 a）以前提－结果形式重写*Tournament* 算法。

b）根据这种重写，详细定义“胜者”和“竞争者”的概念。

c）证明断言10.5.7。(提示：增强这个断言，使之包括当进程处于waitfor循环中时会发生什么情况的信息，这在进程已发现它的一些组件的*flag*变量严格小于k之后。然后用归纳法来证明强化后的不变式和原来的不变式。)

d）完成对*Tournament*算法保证互斥性的证明。

10.15 显示如何改写*Tournament*算法使之适用于n个进程，其中n不是2的幂。时间复杂度会发生怎样的变化？

10.16 研究问题：设计一个使用单写者/多读者的寄存器而不是多写者/多读者寄存器的*Tournament*协议的变式。给出完整的正确性证明和分析。

10.17 如果将*BurnsME* 算法中循环的第二部分删除，那么此算法的行为有何改变？证明修改后的算法仍然可以解决互斥问题，或者给出反例运行。

10.18 给出一个断言式证明来说明*BurnsME*算法满足互斥条件。为此，你必须把算法改写成前提－结果形式，并定义具体的变量来跟踪for循环内的被检查进程。

10.19 给出*BurnsME* 算法的一个低层次公平运行，其中某个进程发生锁定。 329

10.20 对*BurnsME* 算法的演进性条件进行时间复杂度分析。也就是说，假设c和ℓ分别是临界区时间和进程步时间的上界，考虑从某个进程处于T中和没有进程处于C中直到某个进程进入C为止的时间段。

a）证明这个时间的一个上界。

b）给出一个特别运行，其时间尽可能长。

使a）和b）中的界限尽量精确。

10.21 描述*Bakery*算法的一个运行，其中的计数（*number*）寄存器所取的值是无界的。

10.22 为什么当*Bakery*算法中的整数对b（对于某个很大的b）取模后，此算法无效？描述一个具体的反例运行。

10.23 以前提－结果形式重写*Bakery*算法。同时，通过允许动作顺序中的最大不确定性来略微扩展此算法。(前提－结果记号通常比控制流记号更容易地表达这种不确定性)。给出一

个一般算法的互斥条件的断言证明。

10.24 证明在以下弱得多的模型中，*Bakery*算法仍然正常工作。假设读和写不再是瞬时发生的，而是会延续一段时间。假设共享寄存器只被保证为安全的，换言之，它只在读写不会同时发生的情况下才能产生正确的值。在读和写重叠的事件中，读操作可能会得到任何值。

10.25 当所有的共享寄存器都是安全寄存器时（如习题10.24中定义的那样），Burns互斥算法是否仍然正确？为什么？

10.26 假设*Bakery*算法只在瞬时访问共享存储器的情况中工作，而不是在具有安全寄存器的更通用模型中工作。给出此算法的一个简化版本，它保证与原来*Bakery*算法一样的互斥性和高层次公平条件。证明你的结论。

330 10.27详细分析*Bakery*算法的复杂性，注意一个粗略的分析出现在10.7节的结尾。

10.28 给出一个特定用户自动机U_i的代码，其中U_i呈现了用户i的所有合法非确定性——只要符合良构性条件的要求，它就可以在任意时间执行try_i和$exit_i$动作，或者从不执行这些动作。你的自动机应该具有这一属性：对于任意其他的进程i用户自动机V_i，$fairtraces(V_i) \subseteq fairtraces(U_i)$成立。

10.29 给出引理10.36的详细证明。

10.30 *研究问题*：如果我们考虑以下问题而不是互斥问题，10.8节中的结果会受什么影响？

a）在习题10.13中定义的k互斥问题，$2 \leqslant k \leqslant n$。

b）k互斥问题的一个更弱版本，它采用如上的修改的互斥条件，但保留原来的演进性条件。

10.31 Flaky计算机公司的程序员设计了以下的n进程互斥算法。他们声称他们的算法保证了互斥性和演进性，但没有声明任何高层次公平条件。

***A*:**

Shared variables:
$x \in \{1, \ldots, n\}$, initially arbitrary
$y \in \{0, 1\}$, initially 0

Process *i*:

```
        ** Remainder region **

        try_i
L:      x := i
        if y ≠ 0 then goto L
        y := 1
        if x ≠ i then goto L
        crit_i

        ** Critical region **

        exit_i
        y := 0
        rem_i
```

---

331 这个协议满足所声称的两个条件吗？给出证明或明确的反例运行。

10.32 *研究问题*：考虑将演进性条件扩展为k*并发演进性*（*k-concurrent progress*）条件，它要求：如果没有多于k个用户并发处于R之外，那么就做演进。

k并发演进性：在任意公平运行中，其中没有多于k个用户同时处于R之外

a）(尝试区的k并发演进性) 如果至少一个用户处于T中且没有用户处于C中，则在之后某点中某个用户进入C。

b）(退出区的k并发演进性) 如果至少一个用户处于E中，则在之后某点中某个用户进入R。

给出为了保证良构性、互斥性和k并发演进性所需的共享读/写变量的数目上界和下界。

10.33 为一个稍微不同于本章所用模型的读/写共享存储器模型设计一个好的互斥算法。在这个新模型中，除了通常的“代理”进程1,...,n之外，还有一个额外的进程——允许永远执行步的*主管进程*。模型应该使用单写者/多读者共享变量。证明你的算法的正确性并分析它的复杂度。

10.34 证明所有声称的*QueueME*算法的属性。你应该从标识并证明关键的系统不变式开始，然后使用这些不变式来证明互斥条件。可以使用反证法来证明演进性。FIFO条件可以使用特定操作式证明。特别的，你的证明应该导出定理10.39。

10.35 在*Bakery*算法中，不能通过对任意整数取模来减少票的无界值；然而，这种技巧可用于*TicketME*算法。解释区别产生的原因。

10.36 考虑*BufferMainME*算法

a）对于具有主管的算法版本，写出主管和“代理”进程的前提－结果代码。证明算法的正确性。

b）对于没有主管的算法最终版本，做同样的工作。（*提示*：你可以使用模拟证明来把这个版本与具有主管的算法版本联系起来）

332

10.37 设计一个使用单个读－改－写共享变量来解决互斥问题的算法，且它是FIFO的（基于尝试区中第一个局部控制步）。尽量减少共享变量所取的值的数目。你可以假定有一个专用主管进程。（*提示*：对于一个小常数k，$n+k$是可得的。）

10.38 证明*Executive*算法不保证有界绕过。

10.39 *研究问题*：写出*Executive*算法的代码并证明其正确性。它的证明能否形式化地基于*BufferMainME*算法的正确性？能否使用模拟证明？

10.40 对于以下算法，给出一个从进程进入T到进程进入C为止的时间上界：

a）*BufferMainME*算法

b）*Executive*算法

你的分析应该基于底层I/O自动机而不是代码。你可能发现写出算法的前提－后果代码后，分析起来更为方便。

10.41 为什么不能将*Executive*算法的思想扩展到允许变量只有$n/3$个值的情况中？

10.42 *研究问题*：缩小定理10.43和定理10.44中的上界和下界结果之间的差距。（*提示*：部分结果出现在[62]中）

10.43 详细证明引理10.46。

10.44 *研究问题*：使用形式化时序逻辑来重新证明本章中的活性条件（演进性和锁定权）。

333
~
334

第11章 资源分配

在第10章中，我们考虑了互斥问题，它是一个涉及并发用户访问单个非共享资源的抽象问题。在本章中，我们将该问题推扩到包括许多资源而不是只有一个资源的情况。该推扩可用于描述那些在运行中需要几个资源（例如一个打印机、一个数据库和一个网络端口）的应用程序。

本章没有考虑资源分配问题的一些更通用类型，例如：

1）本章不考虑（除非在一些通用定义和习题中）用户愿意接受其他资源组合的可能性。例如，用户可能要求“某个打印机”而不是一个特定的打印机。

2）本章不考虑资源共享的可能性。例如，数据库中的各个数据对象可被视为分配给数据库事务的资源，此时某些共享是允许的；例如，两个只读对象的事务能够同时访问对象。

我们首先定义通用资源分配问题，包括一个*哲学家用餐问题*的有趣例子；然后给出几个典
335 型的解法；最后的解法是一个随机协议——针对异步环境的随机协议的第一个例子。

11.1 问题

在本节中，我们首先给出阐明用户之间冲突关系的几个方法；然后说明如何使用这些方法来定义资源分配问题；最后定义哲学家用餐问题。

11.1.1 显式资源说明和互斥说明

可以从两个角度来看待互斥问题：对明确描述的资源进行分配的问题；或者保证在某时刻只有一个用户进入临界区的问题。也可以从这两个角度来看待通用的资源分配问题。因此，*显式资源规格说明*(*explicit resource specifications*)和*互斥规格说明*(*exclusion specifications*)是描述用户间冲突关系的两种方法。

对n个用户的一个*显式资源规格说明* $\mathcal{R}$ 包括：

1）一个通用有限对象（即资源）集合R；

2）对于每个i, $1 \leqslant i \leqslant n$, 一个集合 $R_i \subseteq R$。

R_i 中的资源是用户U_i执行工作时需要的资源。对于一个给定的显式资源规格说明，如果两个用户U_i和U_j需要一些公用的资源，即如果$R_i \cap R_j \neq \phi$，则称它们是*冲突的*。

例 11.1.1 显示资源规格说明

考虑一个针对四个用户U_1, U_2, U_3, U_4的显式资源规格说明。资源集R为{$r(1)$, $r(2)$, $r(3)$, $r(4)$}。四个用户的资源需求分别为：

U_1: {$r(1)$, $r(2)$}

U_2: {$r(1)$, $r(3)$}

U_3: {$r(2)$, $r(4)$}

U_4: {$r(3)$, $r(4)$}

由此，U_1需要排他地占有资源$r(1)$和$r(2)$ 来进行工作，其他用户的情况以次类推。用户U_1和U_2，U_1和U_3，U_2和U_4，U_3和U_4之间也发生冲突。

另一方面，*排他的规格说明*根本不提及资源，而是给出一个由用户编号的“坏集”组成的
336 集合ε。一个“坏集”是一个由不能同时执行任务的用户的编号组成的集合。排他规格说明有

一个限制：坏集集合在其超集上是封闭的。换而言之，如果用户的特定坏集E属于排他的规格说明ε，则E的任何超集都属于ε。

例 11.1.2 排他规格说明

互斥条件可以通过排他规格说明$\varepsilon = \{E \subseteq \{1, ..., n\} : |E|>1\}$来描述。

例 11.1.3 另一个排他规格说明

k-排他条件（在任意时刻，临界区中的用户个数最多为k）能够以排他规格说明$\varepsilon = \{E \subseteq \{1, ..., n\} : |E|>k\}$来描述。$k$-排他条件在习题10.13中介绍。

例 11.1.4 又一个排他规格说明

对于$n = 4$，考虑由二元集合{1,2}、{1,3}、{2,4}和{3,4}，以及包含这四个二元集合的所有集合组成的排他规格说明ε。这里用户U_1并不排斥U_4，用户U_2也不排斥U_3，这意味着U_1与U_4可以同时工作，U_2与U_3也如此。

注意任意的显式资源规格说明都产生一个等价的排他规格说明，允许相同用户的合并同时运行。排他规格说明由用户集合组成，每个用户集合最少包含两个用户，其中用户的资源需求发生重叠。

例 11.1.5 对应的规格说明

与例11.1.1中的显式资源规格说明对应的排他规格说明由二元集合{1,2}、{1,3}、{2,4}和{3,4}，以及包含这四个二元集合的所有集合组成。

然而，并不是每个排他规格说明都有一个对应的显示资源规格说明。我们将它留做习题。

11.1.2 资源分配问题

现在我们来描述如何将显式资源规格说明和排他规格说明结合到由共享存储器系统解决的资源分配问题中去。具体地考虑一个固定的排他规格说明ε（也可从显式资源规格说明中获得）。 337

这里体系结构与在第10章的互斥问题中的体系结构相同，即是用户自动机和一个共享存储器自动机的合成（参见图10-4）。同样，如图10-2所示，用户在区域*remainder* (R)、*trying* (T)、*critical* (C)和*exit* (E)（分别称为剩余区、尝试区、临界区和退出区）中循环。如果U_i和共享存储器系统之间的交互序列遵循这种循环次序，则称该序列是*良构的*。

合成系统中的良构性条件与以前定义的相同。

良构性 在任意运行中，对于任意i，描述U_i与A之间交互情况的子序列对i是良构的。

原来的互斥条件被更通用的互斥条件替代。

排他性 不存在一个可达系统状态，其中由处于临界区的用户组成的集合是ε中的一个集合。

演进条件与以前定义的一样。

演进性 在一个公平运行的任一点中，

1）（尝试区的演进）如果最少有一个用户在T中，且没有用户在C中，则在以后的某点中某个用户进入C。

2）（退出区的演进）如果最少有一个用户在E中，则在以后的某点中某个用户进入R。

如果共享存储器系统A与用户一起满足良构性、排他性和演进性条件，则称A对于给定的用户集合*解决了通用资源分配问题*。如果A对于每个用户集合都解决了通用资源分配问题，则称A*解决了通用资源分配问题*。

尝试区的演进条件比我们所希望的弱。对于通用资源分配问题，我们认为彼此不冲突的用户不应该互相妨碍彼此进入临界区，即使这些用户永久地拥有资源。对于任意排他规格说明，我们没有声明这种条件的好方法。但是，对于显示资源规格说明，我们至少能够声明以下条

件。

338 **独立演进性**　在一个公平运行的任一点中，

1）（尝试区的独立演进性）如果U_i在T中，且所有冲突的用户都在R中，则在以后某点中，或者U_i进入C，或者某个冲突用户进入T。

2）（退出区的独立演进性）如果U_i在E中，且所有冲突的用户都在R中，则在以后某点中，或者U_i进入R，或者某个冲突用户进入T。

对于高层公平性条件，我们在互斥问题中定义的锁定权和时间界限条件也适用于通用资源分配问题。这里不讨论有界绕过次数条件。这些属性之间的一些简单关系如下所示（试与定理10.9和习题10.6做比较）。

引理 11.1

1）若一个通用资源分配算法具有任一时间界限b，则它是具有锁定权的。

2）若在本章模型中的一个算法保证良构性和锁定权，则它也保证演进性。

证明　该证明留做一道习题。 □

轨迹属性　如同第10章中的做法，我们可以根据轨迹属性来表示良构性、排他性、演进性、独立演进性和锁定权条件。每个轨迹属性P都有一个由*try*、*crit*、*exit*和*rem*输出（没有输入）组成的签名。合成系统的外部动作就是这些动作。对于各个条件，我们需要证明合成系统的公平轨迹都在$traces(P)$中。

限制进程活动　如同第10章中的做法，本章假设共享存储器系统中的进程只有在处于尝试区或者退出区中的时候局部控制动作才是使能的。因此，只有当活动需求存在时，进程才能够运行协议。

11.1.3 哲学家用餐问题

*哲学家用餐问题*是分布式计算理论中最著名的问题之一，它是本章的通用资源分配问题的一

339 个简单特例，通常以显式规格说明来形式化描述。

传统上，该问题的非正式描述如下。n个哲学家（用户）坐在一张圆桌旁边，平时的工作是进行思考（即在R中）。每两个哲学家之间有一个叉子（资源）。有时，一个哲学家可能感到肚饿（即进入T）并试图用餐（即进入C）。为了用餐，哲学家必须排他地使用两把相邻的叉子。在用餐之后，哲学家放回两把叉子（即执行退出协议E）并重新进入思考状态（R）。

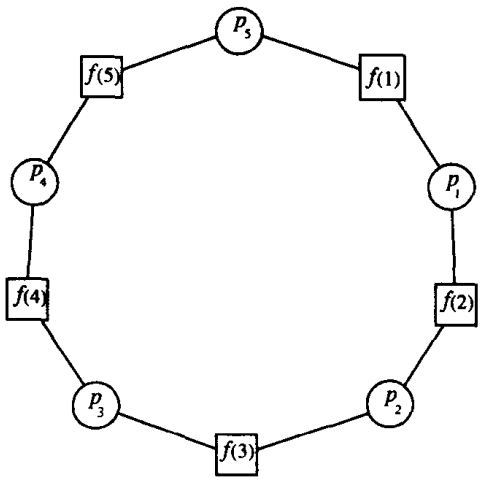


图11-1　哲学家用餐问题（$n = 5$）

对于哲学家p_i，右边（逆时钟方向）的叉子的标号为$f(i)$，左边（顺时钟方向）的叉子的标号为$f(i+1)$（与平常一样，加法对n取模，n的取模结果为0）。图11-1显示了$n = 5$时哲学家和叉子的位置情况。

在正式模型中，每个哲学家都有一个用户和一个代理进程。通常，用户决定何时要求和归还资源，而代理进程执行算法。

n个哲学家用餐问题的排他规格说明由二元集合$\{\{i, i+1\} : 1 \leq i \leq n\}$和所有包含这些二元集

340 的集合组成。

11.1.4 解法的受限形式

本章中的所有解法都属于一种特定形式：每个资源都有且只有一个相关的读－改－写共享变量。只有那些需要相应资源的进程才能访问与资源相关的变量。

这种用来解决受限形式的哲学家用餐问题的体系结构如图11-2所示。注意该图与图11-1非常相似，只不过它包括用户U_i，且以用户编号来标记进程。共享变量与叉子$f(1)$, ..., $f(5)$一一对应。注意进程i访问叉子变量$f(i)$和$f(i+1)$。

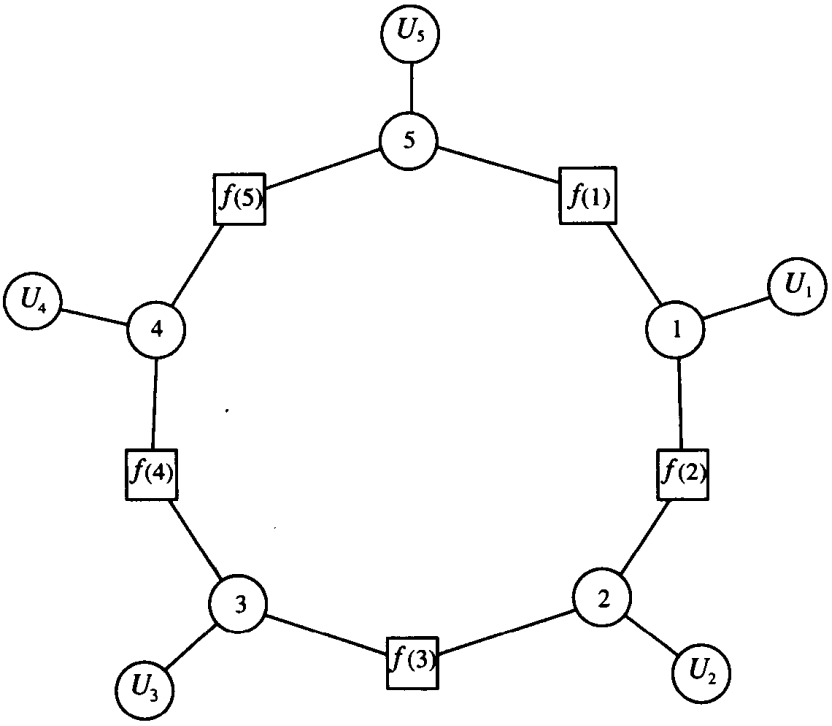


图11-2 带有用户自动机的哲学家用餐问题

11.2 对称哲学家用餐算法的不存在性

一类有意思的哲学家用餐算法是同步算法。对于给定框架中的一个算法，如果所有进程都是相同的，进程通过局部名字$f(left)$和$f(right)$可以引用它们可以使用的叉子变量，且如果所有共享变量的初值相同，则称该算法是对称的。与第3章中的领导选举问题一样，不难看出哲学家用餐算法在对称环境中是不可解的。证明过程与定理3.1的证明过程相同。 341

定理 11.2 不存在哲学家用餐算法的对称解法

证明 采用反证法。对于n个进程，设存在一个对称算法A。考虑A的一个运行α，其中开始时所有进程状态相同，所有共享变量的初始值相同。运行α以“轮换”的顺序执行进程步，即进程步按顺序1, ..., n, 1, ..., 进行，其中每个进程都以try步开始。另外，所有非决定性选择都以同一方式解决。

例如，当一个try_1动作发生时，其他每个进程也执行一个try动作，这些动作所导致的局部状态变化都与进程1的相同。又如，若进程1访问其左变量，则其他所有进程也访问它们的左变量，状态变化情况和变量值变化情况与进程1的相同。

通过对轮换轮数r的归纳，很容易证明在r轮之后所有进程都处于同一状态且所有变量都有

相同值。但演进性属性说明某个进程最终进入C。这意味着其他所有进程也在同一轮中进入C。这与互斥属性相矛盾。 □

例如，考虑以下简单的对称算法。

WrongDP算法（非形式化）：

在进入尝试区之前，每个进程都首先等待右叉子，然后等待左叉子。在得到两个叉子之后，进程进入C。当一个进程退出C时，它在返回到R之前先放下两个叉子。

形式化代码如下；进程i分别使用局部名字*right*和*left*来表示编号i和$i+1$（它的两个叉子的编号）。

WrongDP algorithm (formal):

Shared variables:

for every i, $1 \leq i \leq n$:

342 $f(i)$, a Boolean, initially *false*, accessible by processes i and $i-1$

Actions of i:

Input:	Internal:
try_i	$test\text{-}right_i$
$exit_i$	$test\text{-}left_i$
Output:	$reset\text{-}right_i$
$crit_i$	$reset\text{-}left_i$
rem_i	

States of i:

$pc \in \{rem, test\text{-}right, test\text{-}left, leave\text{-}try, crit, reset\text{-}right, reset\text{-}left, leave\text{-}exit\}$, initially *rem*

Transitions of i:

try_i
Effect:
$pc := test\text{-}right$

$test\text{-}right_i$
Precondition:
$pc = test\text{-}right$
Effect:
if $f(right) = false$ then
$f(right) := true$
$pc := test\text{-}left$

$test\text{-}left_i$
Precondition:
$pc = test\text{-}left$
Effect:
if $f(left) = false$ then
$f(left) := true$
$pc := leave\text{-}try$

$crit_i$
Precondition:
$pc = leave\text{-}try$
Effect:
$pc := crit$

$exit_i$
Effect:
$pc := reset\text{-}right$

$reset\text{-}right_i$
Precondition:
$pc = reset\text{-}right$
Effect:
$f(right) := false$
$pc := reset\text{-}left$

$reset\text{-}left_i$
Precondition:
$pc = reset\text{-}left$
Effect:
$f(left) := false$
$pc := leave\text{-}exit$

rem_i
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

由于*WrongDP*算法是对称的，定理11.2意味着它不能解决哲学家用餐问题。有意思的是去看看问题出在哪里。显然Wrong*DP*保证良构性；并满足排他性条件，因为代码保证进入*C*的进程已经明确地“获得”其两个邻近叉子。 343

但是算法不满足演进性属性。设有一个运行，其中所有进程挨个地进入各自的尝试区；然后，所有进程拿起右叉子。此时，每个进程准备尝试获取左叉子。但由于所有叉子已经被拿起，故没有进程能够得到左叉子。系统现在死锁——不能做进一步的演进。

定理11.2意味着为了解决哲学家用餐问题，必须打破环网络的对称性。具体方法有几种：进程可以使用不同的程序，或者使用相同的进程但不同的初始状态或惟一的标识符。也可以令变量的初值不同；或者使用随机化。本章的余下部分解释了这些方法。

11.3 右－左哲学家用餐算法

本节给出一个名为*RightLeftDP*算法的（正确的）哲学家算法。除了满足所需的基本属性之外，该算法还保证锁定权。它有一个良好的最坏时间界限，该界限是恒定的，与环的大小无关。*RightLeftDP*算法打破对称性的方法是将进程分成两类：“右”和“左”。两类进程运行稍微不同的程序。进程的类别用来指出进程该先拿起哪个邻近叉子。

11.3.1 等待链

恒定时间界限属性特别值得注意。在分布式系统中，与系统的大小无关的时间性能是令人向往的。但如何得到这种小的时间界限呢？

*RightLeftDP*算法是这样的通用算法中的一种：进程串行地先等待一个叉子，然后等待另一个叉子。在这些算法中，必须注意拿起叉子的顺序。例如，如果所有进程都拿起右叉子，则会存在类似于*WrongDP*算法中的死锁可能性。有些顺序不存在死锁可能性，但仍允许运行的时间性能很差。特别的，某些顺序会导致长进程等待链的出现，其中每个进程都等待链中前面进程拥有的资源。 344

例 11.3.1 等待链

设有一个由五个节点组成的环，在环上算法的一个运行中，事件按以下的顺序发生：

- 进程5得到两个叉子；
- 进程4得到其右叉子，然后等待其左叉子；
- 进程3得到其右叉子，然后等待其左叉子；
- 进程2得到其右叉子，然后等待其左叉子；

这产生了一条链：进程2等待进程3拥有的叉子，进程3等待进程4拥有的叉子，进程4等待进程5拥有的叉子。它是一条长度为3的等待链。见图11-3。

当$n \geq 3$时，以上算法产生长度为$n-2$的等待链。

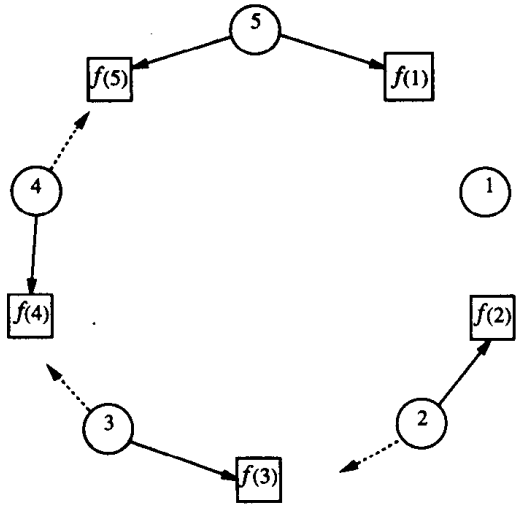


图11-3 等待链。实线箭头代表拥有叉子，虚线箭头代表等待叉子

注意等待链中的进程必须串行地进入临界区。因此，对于这种一般类型的任意算法，尝试进程进入临界区所需的最坏时间与等待链的

最大长度成正比。为了获得小的时间界限，我们必须保证等待链最大长度的小界限。实际上，
345 *RightLeftDP*产生的等待链的最大长度为3。

11.3.2 基本算法

在*RightLeftDP*算法中，与每个叉子相关的共享变量包含一个长度最多为2的进程编号FIFO队列。该队列保存那些想得到叉子的进程的编号，且按照进程开始尝试获得叉子的顺序来排放编号。由于要求使用同一个叉子的进程有两个，故队列的长度为2就足够了。

为简便起见，设环中进程数是偶数。进程数为奇数时则只需做简单修改，具体修改留做一道习题。

*RightLeftDP*算法（*n*为偶数）：

有两种不同的程序：奇数编号的进程和偶数编号的进程。基本策略相当简单：奇数编号进程先寻找右叉子，偶数进程先寻找左叉子。进程寻找叉子时，将其编号放在该叉子队列的尾部。当进程的编号到达叉子队列的头部时，进程得到叉子。当进程退出C时，在进入R之前，它通过从叉子的队列中删除自己的编号来归还叉子。

奇数编号进程i的代码以前提－结果形式来书写。偶数进程的代码是对称的。

RightLeftDP algorithm (even n, odd i):

Shared variables:
for every i, $1 \leq i \leq n$:
  $f(i)$, a queue of process indices of length at most 2, initially empty,
    accessible by processes i and $i-1$

Actions of i:

Input:	Internal:
try_i	*test-right*
$exit_i$	*test-left*
Output:	*reset-right*
$crit_i$	*reset-left*
rem_i	

States of i:
$pc \in \{rem, test\text{-}right, test\text{-}left, leave\text{-}try, crit, reset\text{-}right, reset\text{-}left, leave\text{-}exit\}$, initially *rem*

346 **Transitions of i:**

try_i
  Effect:
    $pc := test\text{-}right$

$test\text{-}right_i$
  Precondition:
    $pc = test\text{-}right$
  Effect:
    if i is not on $f(i).queue$ then
      add i to $f(i).queue$
    if i is first on $f(i).queue$ then
      $pc := test\text{-}left$

$test\text{-}left_i$
  Precondition:
    $pc = test\text{-}left$

$exit_i$
  Effect:
    $pc := reset\text{-}right$

$reset\text{-}right_i$
  Precondition:
    $pc = reset\text{-}right$
  Effect:
    remove i from $f(i).queue$
    $pc := reset\text{-}left$

$reset\text{-}left_i$
  Precondition:
    $pc = reset\text{-}left$
  Effect:
    remove i from $f(i+1).queue$

```
    Effect:
      if i is not on f(i+1).queue then
        add i to f(i+1).queue
      if i is first on f(i+1).queue then
        pc := leave-try

crit_i
    Precondition:
      pc = leave-try
    Effect:
      pc := crit

      pc := leave-exit

rem_i
    Precondition:
      pc = leave-exit
    Effect:
      pc := rem
```

现在来证明正确性。良构性是显然的，排他性也是易见的。因为代码保证一个到达C的进程在它获得的两个叉子的队列中处于队列头。我们将证明任一尝试进程到达临界区所需的确切时间上界。退出区的小时间上界（独立于n）也是易见的。根据引理11.1，这些界限足以保证锁定权，从而又足以证明演进性。

对于时间界限，如前所示，假设ℓ是每个进程的每步时间的上界，c是任意用户花在临界区中的时间的上界。

引理 11.3 在*RightLeftDP*中，特定进程从进入T到进入C之间的时间最多为$3c+18\ell$。

347

证明 关键在于：两个进程中间的叉子或者是这两个进程的第一把叉子，或者是它们的第二把叉子。这意味着（在假设的情况中，当n是偶数时）等待链的最大长度为2。

定义T为任一进程i从进入尝试区到进入临界区之间的最长时间。我们的目标在于求出T的界限。作为辅助数，S为任一进程得到第一把叉子到进入临界区之间的最长时间。正式的，当进程i处于叉子队列的头部时它获得第一把叉子（此时执行了i的一步或者共享该叉子的邻居的一步）。

我们根据S来求T的界限。设进程i进入尝试区，在时间ℓ之内，它执行测试事件π来尝试得到第一把叉子。如果它立刻得到这把叉子，则在额外的时间S之内，进程i进入临界区。所以总共的时间最多为$\ell + S$。

如果进程i不能够立刻得到叉子，则在π发生时，和i一起共享该叉子的邻居——如进程j——拥有叉子。直到j释放叉子为止最多需要时间$S + c + \ell$（该时间足以让进程j到达临界区、离开临界区和释放第一个叉子）。在j释放叉子的瞬间，进程i获得叉子；原因在于“获得叉子”的定义，以及进程i在事件π中将其编号放进队列。然后，在额外时间S内，i到达临界区。所以进程i最多在时间$\ell + (S + c + \ell) + S = c + 2\ell + 2S$内进入临界区。

可得：

$$T \leqslant \max\{\ell + S, c+2\ell + 2S\} = c+2\ell + 2S \qquad (11\text{-}1)$$

然后我们来求S的界限。设进程i刚刚获得第一把叉子（即处于该叉子的队列之首）。在时间ℓ之内它发现这一事实。在额外时间ℓ内，它对第二把叉子执行测试操作。如果它立刻获得第二把叉子，则在之后的时间ℓ之内达到临界区，故全部时间最多为3ℓ。

如果它不能立刻获得第二把叉子，则和i一起共享该叉子的邻居i拥有该叉子，且该叉子是邻居的第二把叉子。直到该邻居释放叉子为止最多需要时间$2\ell + c + 2\ell$（该时间足以让邻居发现叉子、到达临界区、离开临界区和释放第二个叉子）。从邻居释放叉子的那一刻开始，进程i最多需要时间ℓ来发现叉子，且额外的时间ℓ来进入临界区。所以进程i最多共在时间$2\ell + (2\ell +$

$c+2\ell)+2\ell = c+8\ell$之内进入临界区。

348 可得：

$$S \leqslant \max\{3\ell, c+8\ell\} = c+8\ell \tag{11-2}$$

结合式(11-1)和式(11-2)得

$$T \leqslant 3c+18\ell$$ □

容易看出该算法还满足独立演进性条件，故得到

定理 11.4 *RightLeftDP*算法解决了哲学家用餐问题，并保证锁定权和独立演进性，尝试区的时间界限为$3c+18\ell$，退出区的时间界限为3ℓ。

故此，*RightLeftDP*算法通过区分奇数编号进程和偶数编号进程来打破对称性。依赖于算法运行的环境，进程可能或者不可能拥有这些知识（即编号是奇数还是偶数）。例如，如果算法在分布式网络（见第17章）上运行，则可能需要一个额外的协议来决定这种奇偶信息（奇数编号还是偶数编号）并将它通知给所有进程。

11.3.3 扩展

以下描述一种简单直接的方法来将*RightLeftDP*中的策略扩展到任意资源分配问题（以任意显示资源规格说明来描述）中去。扩展也具有一个时间界限，也与进程数目无关；但其界限并不小——性能方面仍有改进的余地。

我们继续假设每个资源都有一个相关的共享变量，需要某一资源的所有变量都共享其相关变量。如在*RightLeftDP*中一样，设每个变量都包含一个FIFO队列来记录等待相关资源的进程；每个进程在某一时间只等待一个它所需的资源。然而，为了避免死锁，对全部资源进行排序，每个进程按次序来获得它所需的资源——号码最小的进程最迟获得资源。这种策略被称为*分级资源分配*。

不难看出分级资源分配保证演进性。粗略地说，如果进程i等待进程j持有的资源，则进程j
349 等待的资源的编号只能严格大于（按照资源次序）i等待的资源的编号；由于只有有限多的进程，持有最大资源的进程必然不会被阻塞。队列的FIFO属性也防止了锁定的发生。

尽管分级资源分配保证演进性和锁定权，其时间性能并不好。等待队列长度的惟一上界是进程的数目n，使得时间复杂度最少正比于n。例如，例11.3.1中描述的链能以一个分级资源分配算法生成，其中资源次序为$f(1)$、$f(2)$、$f(3)$、$f(4)$、$f(5)$。

我们希望得到一个“好”的资源全序，且时间界限越小越好。一个合理的策略是尝试将生成的等待链的长度最少化。

给出一个特定的包含通用资源集合R和独立进程资源需求R_i的显式资源规格说明$\mathcal{R}$，为了构造一个好的全序，我们首先为该规格说明构造*资源图*。图的节点代表资源，如果某一进程使用两个相关的资源，则有一条从一个相应节点到另一个相应节点的边。

例 11.3.2 资源图

具有六个节点的哲学家用餐问题的资源图如图11-4所示。

然后把图的节点*着色*，相邻节点的颜色不同。尽量令所需颜色的数目最小化（这里不考虑如何获得着色的最小颜色数，实际上，获得最小数目是一个NP完全性问题。找到一个较小的
350 着色数目就足够了。例如，可以使用一个贪婪算法来以不多于$d+1$种颜色对图着色，其中d是图中任意节点度数的上界）。

例 11.3.3 资源图的着色

图11-4中的资源图可以用两种颜色来着色，例如，奇数编号的资源上用颜色1，而偶数编

号的资源上用颜色2。

现在以任意方式对颜色排序，从而引入资源上的偏序关系：$r(i) < r(j)$，当且仅当$r(i)$的颜色值排在$r(j)$的颜色值之前。尽管这是一种偏序关系，但每个单独进程需要的资源之间却是全序关系。由于我们需要所有资源上的全序关系，故简单地以任意方式（使用偏序关系的拓扑排序）来把偏序关系补全为全序关系。

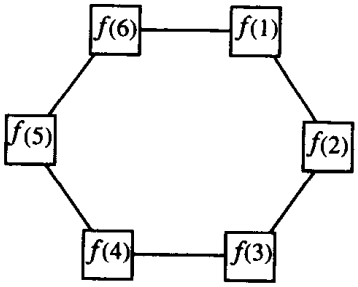


图11-4 哲学家 (n=6) 用餐问题的资源图

例 11.3.4 资源的偏序关系

例11.3.3中的着色引入了如图11-5所示的资源上的偏序关系。“更小”的资源出现在示意图的顶部。

现在来描述算法

***Coloring*算法：**

根据以上基于着色构造的全序，每个进程按升序来寻找资源。进程通过将编号放在资源队列的尾部来寻找资源。当其编号到达资源队列的头部时进程获得资源。当进程退出C时，它通过在队列中删除其编号的方式来归还所有占用资源。

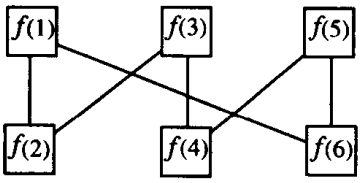


图11-5 资源上的偏序关系

由于同一进程所需的两个资源的颜色不同，*Coloring*算法的一个等价描述是：每个进程根据偏序关系按升序寻找其资源。注意，对于偶数进程环上哲学家用餐问题这一特例，*Coloring* 351
算法变成*RightLeftDP*算法。

在*Coloring*算法中，等待链的最大长度最多等于不同颜色的数目，原因在于如果进程i等待进程j持有的资源，则j只能等待“更大”颜色的资源。

*Coloring*算法的一个有意思属性是时间界限与进程和资源的总数无关。通常，令ℓ为进程步时间的上界，c为临界区时间的上界，k为用于对资源着色的颜色的总数，m为需要任意单个资源的最大进程数。我们来证明最坏时间界限为$O(m^k c + km^k \ell)$，原因在于颜色数目和每个资源上的进程数目并不依赖于系统的规模，故时间只依赖“局部”参数。如果m和k相对于n（系统的进程总数）来说较小，则这个界限代表了该策略对任意全序的分级资源分配策略（通用策略的等待链长度约为n）的提高。但是，这个界限可能不像希望中的那样小——它不正比于等待链的最大长度（即正比于k），而是k的指数。

引理 11.5 在*Coloring*算法的任意实例中，k是颜色总数，m是单个资源上进程数的上界，则特定进程i从进入T到进入C之间的时间为$O(m^k c + km^k \ell)$。

证明梗要 设颜色值为整数$1, \cdots, k$。定义$T(i, j)$, $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant m$为这样一个时间：即对于颜色值不小于$i$的资源队列，从进程到达该队列中任意不大于$j$（$\leqslant j$）的位置开始，直到它到达临界区之间所需的最坏时间。我们希望求出T的界限，即从进入尝试区到进入临界区之间的最坏时间。从进程进入尝试区开始，直到它的编号被放在某个资源队列中时，最多需要时间ℓ。因此，

$$T \leqslant \ell + T(1, m)$$

如同*RightLeftDP*算法中的做法，我们建立递归等式来求出$T(i, j)$的界限。基本情况是当进程处于具有最大颜色值的资源队列中时：

$$T(k, 1) \leqslant 2\ell$$

352

在该时间内，进程发现自己处于队首，然后进入临界区。

另一种情况是当进程处于具有非最大颜色值的资源的队首时：

$$\text{对于任意}i,\ 1 \leqslant i < k,\ T(i,1) \leqslant \max(2\ell,\ 2\ell + T(i+1,m)) = 2\ell + T(i+1,m)$$

在该时间内，进程发现自己处于队首，然后或者进入临界区，或者到达具有更大颜色值的资源的队列。

最后一种情况是当进程处于某队列的非队首位置时，

$$\text{对于任意}j,\ 1 < j \leqslant m,\ T(i,j) \leqslant T(i,\ j-1) + c + k\ell + T(i,1)$$

在该时间内，队列中当前进程的前驱到达其临界区，然后完成临界区中的操作，再释放其占用的所有$\leqslant k$个资源（使得原来进程处于给定队列的队首）；之后处于队首的当前进程到达临界区。

求解这些不等式来得到所需的界限。对于任意i, $1 \leqslant i \leqslant k$，可得

$$T(i,m) \leqslant (m-1)(c+k\ell) + mT(i,1)$$

对于任意i, $1 \leqslant i < k$, 可得

$$T(i,m) \leqslant m(c + (k+2)\ell) + mT(i+1,m)$$

和

$$T(k,m) \leqslant m(c + (k+2)\ell)$$

故

$$\begin{aligned} T(1,m) &\leqslant (c + (k+2)\ell) \sum_{i=1}^{k} m^i \\ &= O\left(m^k (c + (k+2)\ell)\right) \end{aligned}$$

从而，

$$T = O\left(m^k (c + (k+2)\ell)\right) = O\left(m^k c + k m^k \ell\right)$$

□

定理11.6　*Coloring*算法解决了资源分配问题并保证锁定权和独立演进性，尝试区的时间界限为$O(m^k c + k m^k \ell)$，退出区的时间界限为$O(k\ell)$。

353　存在*Coloring*算法的某些运行，其时间效能接近该指数界限。找出这些运行的任务留做习题。当然，将时间界限从指数级颜色数降为线性级颜色数会更好，但这需要不同的算法。

11.4　哲学家用餐随机算法*

我们最后提出一个随机哲学家用餐算法，它（肯定）保证互斥，而确保演进性的概率为1。根据其发明者来将它命名为*LehmannRabin*算法。在该算法中，所有进程都是相同的，我们使用随机化来打破对称性。

该算法有几点需要注意。首先，随机算法既可以在异步环境中使用，也可以在同步环境中使用，它有时可以完成非随机算法所不能完成的工作。例如，即使进程是相同的时候，*LehmannRabin*算法也能解决哲学家用餐问题，而定理11.2意味着非随机算法不能完成这个任务。

实际上，当说这个算法“解决了哲学家用餐问题”时要小心：被满足的正确性条件与以前阐述的不完全相同，此时进程只需以概率1而不需以绝对确定性来保证演进性条件。

其次，我们来示范如何对随机异步系统做出有意义的概率断言。如何做出这些断言并不明显，因为随机算法本身并没有给出运行上的随机分布。例如，进程在异步算法中执行步的顺序是相当任意的，不是随机确定的。为了定义一个概率分布，必须确定这种顺序。

再次，我们使用一种马可夫形式的分析技术来证明概率时间界限属性。这种属性可用于证明概率活性属性。

11.4.1 算法*

由于进程是相同的，故假设它们通过局部名字来找到叉子。与以前一样，每个进程通过局部名字$f(right)$和$f(left)$来找到叉子。采用以下记号

$$\bar{j} = \begin{cases} left, & \text{如果 } j = right \\ right, & \text{如果 } j = left \end{cases}$$

语句$first := random$的意思是$first$被设为$right$或$left$的概率都是1/2。一个*LehmannRabin*算法 354
的非形式描述如下。

LehmannRabin algorithm :

Shared variables:
for every i, $1 \le i \le n$:
$f(i)$, a Boolean, initially *false*, accessible by processes i and $i-1$

Process i:

```
** Remainder region **

try_i
do forever
        first := random
        wait until f(first) = false
        f(first) := true
        if f(first-bar) = false then
                f(first-bar) := true
                goto L
        else f(first) := false
L:
    crit_i

    ** Critical region **

    exit_i
    put down both forks
    rem_i
```

因此，尝试进程i执行一个循环，在每步中尝试获得两个叉子。在每步中，它首先随机地选择第一个叉子并一直等到获得这个叉子为止。在获得第一个叉子之后，它不会一直等待第二个叉子，而是检查一次看看第二个叉子是否可用。如果可用，则进程i获得该叉子并进入C；否则进程i放弃这一步，把第一个叉子放回，然后在下一步中重新尝试。

为了防止含糊性，我们给出前提－结果代码。

LehmannRabin algorithm (rewritten):

Shared variables:
for every i, $1 \leq i \leq n$:
355 $f(i)$, a Boolean, initially *false*

Actions of i:

Input:	Internal:
try_i	$flip_i$
$exit_i$	$wait_i$
Output:	$second_i$
$crit_i$	$drop_i$
rem_i	$reset\text{-}right_i$
	$reset\text{-}left_i$

States of i:
$pc \in \{rem, flip, wait, second, drop, leave\text{-}try, crit, reset\text{-}right, reset\text{-}left, leave\text{-}exit\}$, initially *rem*
$first \in \{right, left\}$, initially arbitrary

Transitions of i:

try_i
Effect:
$pc := flip$

$flip_i$
Precondition:
$pc = flip$
Effect:
$first := random$
$pc := wait$

$wait_i$
Precondition:
$pc = wait$
Effect:
if $f(first) = false$ then
$f(first) := true$
$pc := second$

$second_i$
Precondition:
$pc = second$
Effect:
if $f(\overline{first}) = false$ then
$f(\overline{first}) := true$
$pc := leave\text{-}try$
else $pc := drop$

356 $reset\text{-}left_i$
Precondition:
$pc = reset\text{-}left$
Effect:
$f(left) := false$
$pc := leave\text{-}exit$

$drop_i$
Precondition:
$pc = drop$
Effect:
$f(first) := false$
$pc := flip$

$crit_i$
Precondition:
$pc = leave\text{-}try$
Effect:
$pc := crit$

$exit_i$
Effect:
$pc := reset\text{-}right$

$reset\text{-}right_i$
Precondition:
$pc = reset\text{-}right$
Effect:
$f(right) := false$
$pc := reset\text{-}left$

rem_i
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

正式地，代码中描述的对象是8.8节中定义的概率I/O自动机。随机选择步就是*flip*步。每步都有一个概率分布，分布中包含两个可能状态而不只是一个新状态，每个状态的概率都是1/2。注意系统的运行通过将非确定选择和概率选择组合起来进行。非确定选择决定应由哪个进程执行下一步，从而决定下一步是什么；而概率选择决定*flip*步的新状态。

11.4.2 正确性*

容易看出*LehmannRabin*算法保证良构性、排他性和独立演进性；这些断言并不涉及概率。正式的，它们是8.8节中定义的系统的非确定性版本。然而演进性条件并不肯定成立。

例11.4.1 不做演进的LehmannRabin算法的运行

考虑*LehmannRabin*算法的一个运行α，其中进程按轮换顺序执行步，并一直做出相同的随机选择。注意α是一个（系统的非确定版本的）公平运行。在α中，没有进程能够到达C。

有意思的是证明*LehmannRabin*算法保证概率为1的演进性。实际上，我们不去证明概率为1的演进性，而去证明形如$\mathcal{T} \xrightarrow[p]{t} \mathcal{C}$的更强概率时间界限。非正式的说，这意味着从处于$T$中的某个进程的任意可达状态开始，在时间$t$之内，某进程在$C$中的最小概率为$p$。通过反复应用这个断言，可以证明概率为1的演进性条件。

为了做出与特定事件相关的断言，我们需要运行上的概率分布。到目前为止，系统包含非确定选择（即由哪个进程执行下一步）和概率选择。实际上，无论如何进行非确定选择，系统都具有所希望的属性。 357

将非确定选择视为置于对手（*adversary*）的控制之下是有用的。只要对手允许在尝试区或退出区中的每个进程都有公平的机会来执行，则对手可以选择任意进程。事实上，由于我们的目标在于证明概率时间界限的论断，对手不但要选出一个进程来执行下一步，还要决定这一步发生的时间。发生时间取决于进程步时间的上界ℓ和临界区时间的上界c，而且如果运行是无限的，这该时间必然是无穷大的。为了得到最强的结果，对手应该尽量强大；因此，当决定由谁来执行下一步和在何时执行这一步时，它具有以往运行的完整知识——包括进程状态和以往随机选择的信息。

正式地，对手$\mathcal{A}$是一个将有限运行映射到(进程、时间)对上的函数，它指出由哪个进程来执行下一步和在何时执行这一步。对于每个特定的随机序列D，具有由D给出的随机选择的对手$\mathcal{A}$产生惟一一个定时运行$exec(\mathcal{A}, D)$。对手必须符合这样一个条件：所有在$exec(\mathcal{A}, D)$中的定时运行都具有在先前段落中描述的公平性和定时属性。

一个固定的对手$\mathcal{A}$决定了在算法的定时运行集合上的一个概率分布，由于每个随机选择是“右”或“左”，且每个的概率都是1/2，故每个序列（的可数集合）都有一个相关概率。在序列D上的概率分布指出了在定时运行$exec(\mathcal{A}, D)$上的概率分布。

我们的证明还需要一个概念。如果$\mathcal{U}$和$\mathcal{U}'$都是状态集，则$\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$的意思如下：对于任意对手$\mathcal{A}$，如果算法以$\mathcal{U}$的一个状态开始，则在$\mathcal{A}$决定的运行概率分布中，在时间$t$内$\mathcal{U}'$的一个状态被达到的概率最小为$p$。可以把这些断言结合起来，例如：

引理 11.7

1）如果$\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$且$\mathcal{U}' \xrightarrow[p']{t'} \mathcal{U}''$，则$\mathcal{U} \xrightarrow[pp']{t+t'} \mathcal{U}''$

2）如果$\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$则$\mathcal{U} \cup \mathcal{U}'' \xrightarrow[p]{t} \mathcal{U}' \cup \mathcal{U}''$. 358

现在我们有足够的工具来证明演进属性。一个技术问题是：证明中的某些构造只适用于环的尺寸n最小为3的情况。从这里开始，我们假设环的尺寸n最小为3，$n=2$时的证明留做一道习题。

定义：

- 令$\mathcal{T}$是*LehmannRabin*算法的可达状态集合，在它包含的状态中，某个进程在T中。
- 令$\mathcal{C}$是由可达状态集合，在它包含的状态中，某个进程在C中。

我们来证明$\mathcal{T} \xrightarrow[\frac{1}{16}]{14\ell} \mathcal{C}$，即从处于$T$中的某个进程的任意可达状态开始，在时间$14\ell$之内，某进程将进入$C$的概率最小为1/16。我们通过五条由引理11.8~11.12表达的5个辅助断言，来证明这个断言。这些引理的表示规则与引理11.7的相同。

我们使用一些简写来对进程状态分类。令F、W、S、D和L来分别表示进程状态*pc=flip*、*wait*、*second*、*drop*和*leave-try*。这五个状态集构成了对尝试区T的划分。根据*first*的值来进一步划分W、S和D：$\vec{W}$、$\vec{S}$、$\vec{D}$分别代表了*first* = *right*时W、S、D的子集；$\overleftarrow{W}$、$\overleftarrow{S}$、$\overleftarrow{D}$分别代表了*first* = *left*时W、S、D的子集；记号$\vec{*}$代表$\vec{W} \cup \vec{S} \cup \vec{D}$，同样$\overleftarrow{*}$代表$\overleftarrow{W} \cup \overleftarrow{S} \cup \overleftarrow{D}$。现在来定义辅助引理所需的系统状态集合。

令：

- $\mathcal{L}$是可达状态集合，在它包含的状态中，某个进程在$\mathcal{L}$中(即在*leave-try*中)。
- $\mathcal{RT}$是$\mathcal{T}$的子集，在它包含的状态中，所有进程都在剩余区或尝试区中。
- $\mathcal{F}$是$\mathcal{RT}$的子集，在它包含的状态中，某个进程在F中(即在*flip*中)。
- $\mathcal{G}$是$\mathcal{RT}$的子集，在它包含的状态中，存在满足下列条件之一的进程i：

  —— $i \in \vec{W} \cup \overleftarrow{S}$且$i-1 \in \overleftarrow{*} \cup R \cup F$

  —— $i \in \vec{W} \cup \vec{S}$且$i+1 \in \vec{*} \cup R \cup F$

359 以上三个集合的意思很明显。最后一个集合$\mathcal{G}$是“好”状态集合，其中两个进程处于这样一种状态之中：一个进程很有可能很快就得到两个叉子。图11-6显示了$\mathcal{G}$允许的两个状态。直观上，在好状态中，两个相邻进程共享第二个叉子的概率很大。如果它们共享第二个叉子，则谁先访问这个叉子，谁就得到它并成功地到达C。

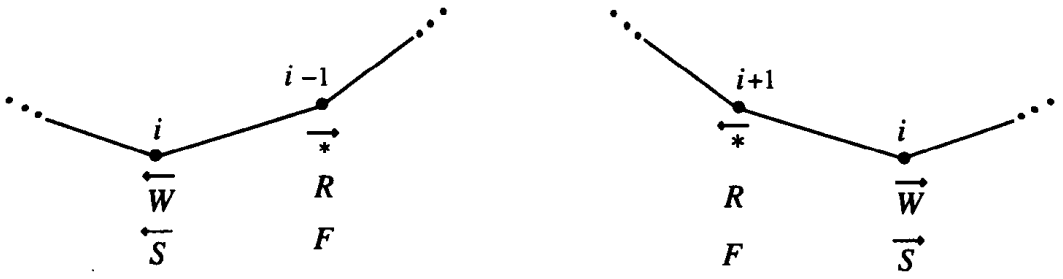


图11-6　*LehmannRabin*算法的好状态

然后我们来证明下列断言：

- $\mathcal{T} \xrightarrow[1]{3\ell} \mathcal{RT} \cup \mathcal{C}$
- $\mathcal{RT} \xrightarrow[1]{3\ell} \mathcal{F} \cup \mathcal{L}$
- $\mathcal{F} \xrightarrow[\frac{1}{4}]{2\ell} \mathcal{G} \cup \mathcal{L}$

- $\mathcal{G} \xrightarrow[\frac{1}{4}]{5\ell} \mathcal{L}$
- $\mathcal{L} \xrightarrow[1]{\ell} \mathcal{C}$

引理11.7允许我们合成这些断言来产生所需的结论 $\mathcal{T} \xrightarrow[\frac{1}{16}]{14\ell} \mathcal{C}$。

我们首先证明三个概率为1的断言，因为它们最容易证明。事实上，它们不但以概率1成立，而且肯定成立。

引理11.8 $\mathcal{L} \xrightarrow[1]{\ell} \mathcal{C}$

证明 如果进程处于*leave-trying*中，则在时间ℓ之内，同一进程将执行一步并进入C。 □ 360

引理 11.9 $\mathcal{T} \xrightarrow[1]{3\ell} \mathcal{RT} \cup \mathcal{C}$

证明 如果任一进程开始时在C中，或在时间3ℓ内进入C，则得证，所以我们不考虑这种情况。然后，任意进程在$R \cup T \cup E$中的停留时间最少为3ℓ，且没有进程在该时间内进入E（因为没有进程在C中）。但是，初时在E中的任一进程在时间3ℓ内返回到R中，这迫使所有进程在时间3ℓ之内进入$R \cup T$。得证。 □

引理 11.10 $\mathcal{RT} \xrightarrow[1]{3\ell} \mathcal{F} \cup \mathcal{L}$

证明 如果任一进程开始时在$F \cup L$中，或在时间3ℓ内进入L，则得证，所以我们不考虑这种情况。然后，没有进程在时间3ℓ之内进入C，故在时间3ℓ之内发生的所有系统状态都在$\mathcal{RT}$中。之后如果一个进程在时间3ℓ之内进入F，则它导致系统处于$\mathcal{F}$中，我们也不考虑这种情况。故此没有进程在时间3ℓ之内进入尝试区。

现在所有进程都在$R \cup W \cup S \cup D$中。如果一个进程开始时在$S \cup D$中，或者在时间ℓ之内到达$S \cup D$，则在额外的时间2ℓ之内，该进程进入$F \cup L$，矛盾。故惟一的可能是所有进程开始时都在$R \cup W$中，且（由于 $\mathcal{RT} \subseteq \mathcal{T}$）某个进程在$W$中。另外，没有进程在时间$\ell$之内到达$S \cup D$。这意味着在开始时没有叉子被进程占用。

因为某个进程在W中，某个进程在时间ℓ之内必然执行一步。令i为第一个执行步的进程。如果i开始时在R中，则它进入尝试区，矛盾。另外，如果i开始时在W中，则由于没有叉子被占用，故i立刻获得第一个叉子。但这会导致i在S中，又产生一个矛盾。 □

到此为止，我们还没有讨论任何与概率相关的证明。剩下的两个断言涉及这种证明。第一个断言解释了：从任一其中某个进程正在跳跃(flipping)状态开始，很快到达一个好状态或者某个进程很快到达*leave-try*的概率最小为1/4。

引理11.11 $\mathcal{F} \xrightarrow[\frac{1}{4}]{2\ell} \mathcal{G} \cup \mathcal{L}$

证明 如果所有进程开始时都在L中，则证明完成，所以我们不考虑这种情况。令i是初始时在F中的任一进程。则开始时下列情况之一成立：

1）$i-1 \in \vec{*} \cup R \cup F$ 361

见图11-7。i的下一个随机选择是*left*且$i-1$的下一个随机选择是*right*的概率为1/4。我们做

这种假设。

在时间ℓ内，i跳跃，因为它的下一个随机选择是*left*，故它进入状态$\bar{W}$中。分两种情况：

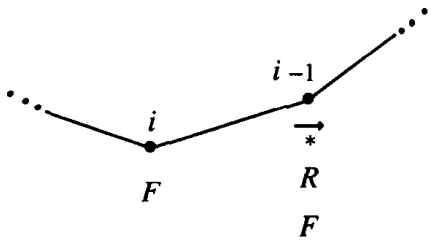


图11-7　引理11.11的证明中第1种情况的初始状态

a）期间$i-1$不访问该共享叉子。则$i-1$的状态必然在集合$\bar{*} \cup R \cup F$中，可以利用$i-1$的下一个随机选择是*right*这一事实，通过检查该状态集合之外的可能转换来证明这一点。这导致系统的状态进入$\mathcal{G}$，得证。

b）期间$i-1$访问该共享叉子。在$i-1$第一次访问叉子的时候，共享叉子必是$i-1$的第二个叉子（在$\vec{D}$、R和F情况下，这是因为$i-1$的下一个随机选择是*right*）。则$i-1$获得第二个叉子并进入L，得证。

2）$i+1 \in \bar{*} \cup R \cup F$

见图11-8。这与第1种情况是对称的。

3）$i-1 \in \bar{*}$和$i+1 \in \vec{*}$

见图11-9。

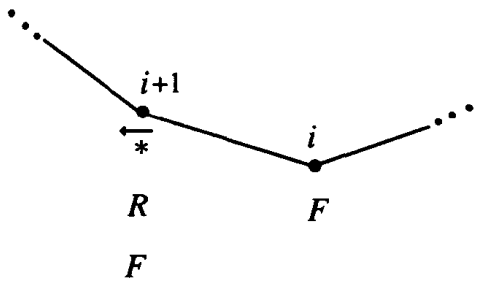


图11-8　第2种情况的初始状态

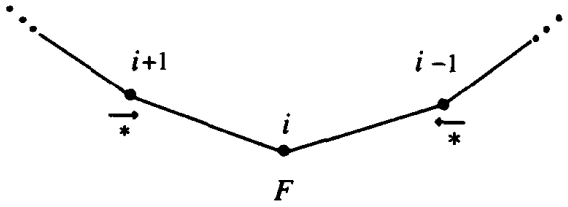


图11-9　第3种情况的初始状态

362　这种情况很有意思，因为它看起来与在好状态中发生的情况完全不同。但由于我们讨论的是一个环，如果i周围的环境是不利的话，则在此环中另一处肯定存在另一个进程j满足$j+1 \in \bar{*}$和$j \in \vec{*} \cup R \cup F$。（证明留做一道简单的习题。）见图11-10。进程j周围的情况看起来要好很多。

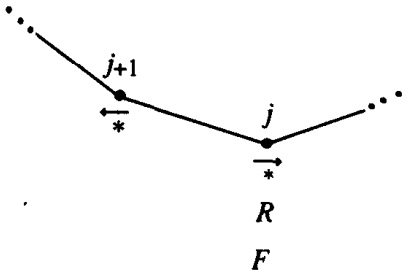


图11-10　环中的其他地方

如果$j+1 \in \bar{W} \cup \bar{S}$，则初始状态已经在$\mathcal{G}$中，得证。

另外一种惟一的可能性是$j+1 \in \vec{D}$且$j \in \vec{*} \cup R \cup F$。然而，在这种情况下，$j+1$的下一个随机选择是*left*且$j$的下一个随机选择是*right*的概率为1/4。假设这种情况成立。接着，在时间2ℓ内，进程$j+1$执行两步，从而进入$\bar{W}$。这意味着系统状态最终在$\mathcal{G}$中，除非在这段时间内进程j离开集合$\vec{*} \cup R \cup F$。但如果j离开，则j已经获得第二个叉子并到达L，得证。　□

363　最后的引理表明好状态好在哪里：从好状态开始，某个进程很快到达L的概率至少为1/4。

引理 11.12　$\mathcal{G} \xrightarrow[\frac{1}{4}]{5\ell} \mathcal{L}$

证明　由于初始状态在$\mathcal{G}$中，则它也在$\mathcal{RT}$中，且$\mathcal{G}$的定义中的两个条件之一（最少有一个）成立。不失一般性，假设第一个条件成立，即存在进程i使得$i \in \bar{W} \cup \bar{S}$和$i-1 \in \vec{*} \cup R \cup F$成立。

对第二个条件的证明是类似的。

我们引入三个基本断言。这些断言的声明并不明确涉及概率，而是通过引用确定的未来随机选择的值来隐含地涉及概率。第一个断言给出了当定位出有利邻居时进程一直等待第一个叉子的时间的界限。

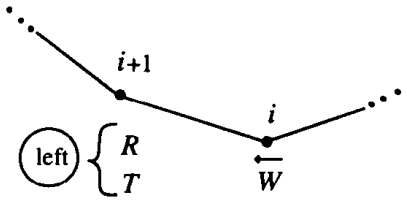


图11-11 声明11.13的情况

声明11.13 如果$i+1 \in R \cup T$，其下一个随机选择为*left*，且$i \in \tilde{W}$，则在时间4ℓ内，$i \in \tilde{S}$或$i+1 \in L$成立。

见图11-11。

证明 这是基于进程$i+1$的状态而做出的一个（比较冗长）证明。

1）$i+1 \in L$

得证。

2）$i+1 \in \tilde{*} \cup R \cup F$

开始时，$i+1$不占有共享叉子。在时间ℓ之内，i检查共享叉子。如果与此同时$i+1$没有访问这个叉子，则i获得叉子并到达$\tilde{S}$，得证。

假设同时$i+1$已经访问了共享叉子。鉴于$i+1$的状态，共享叉子必然是$i+1$的第二个叉子（在$\tilde{D}$、R和F的情况下，因为$i+1$的下一个随机选择为*left*。）。由于$i+1$获得这个叉子，它成功地到达L，得证。此时所需时间最多为ℓ。 364

3）$i+1 \in \vec{D}$

在时间ℓ之内，$i+1$放下叉子，然后$i+1 \in F$，其下一个随机选择为*left*；且$i \in \tilde{W}$。（进程i必然还在$\tilde{W}$中，因为在$i+1$放下叉子之前它不能得到叉子）。结果状态与前面所述的情况相似，故此，在额外的时间ℓ之内，$i \in \tilde{S}$或$i+1 \in L$成立。这次所需的时间最多为2ℓ。

4）$i+1 \in \vec{S}$

在时间ℓ之内，$i+1$检测左叉子。如果条件成立，则立即到达L，变成前面所述的情况；否则到达$\vec{D}$。这种情况所需的时间最多为3ℓ。

5）$i+1 \in \vec{W}$

在时间ℓ内，i与$i+1$都检测它们的共享叉子。先检测的进程先得到共享叉子。如果进程i先得到，则有$i \in \tilde{S}$，得证；否则，有$i+1 \in \vec{S}$，变成前面所述的情况。这种情况所需时间至多为4ℓ。 □

第二个断言给出了从一个进程准备测试第二个叉子（且定位出一个有利邻居的时候）到某个进程到达L之间的时间界限。

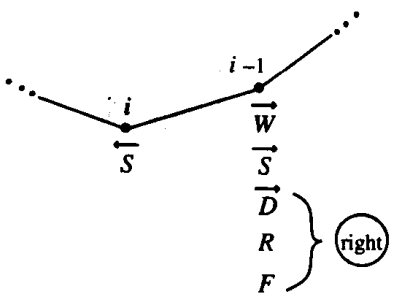


图11-12 声明11.14的情况

声明11.14 设$i \in \tilde{S}$，$i-1 \in \tilde{W} \cup \tilde{S}$或$i-1 \in \vec{D} \cup R \cup F$，$i-1$的下一个随机选择为右（right）。则在时间$\ell$内，某个进程在$L$中。

见图11-12。 365

证明 在时间ℓ内，i检测共享叉子；除非$i-1$在此时间内得到该叉子，否则i得到叉子并到达L。但是如果$i-1$已经得到该叉子，则由于该叉子肯定是$i-1$的第二个叉子，故i已经到达L。 □

最后的声明结合了前面两个声明，它给出了从进程i等待第一个叉子（且定位出两个有利的邻居）到某个进程到达L之间的时间的界限

声明11.15　若$i+1 \in R \cup T$，其下一个任意选择是*left*，$i \in \vec{W}$，且$i-1 \in \vec{*} \cup R \cup F$，其下一步的任意选择为*right*，则在时间$5\ell$内，某个进程在$L$中。

见图11-13。

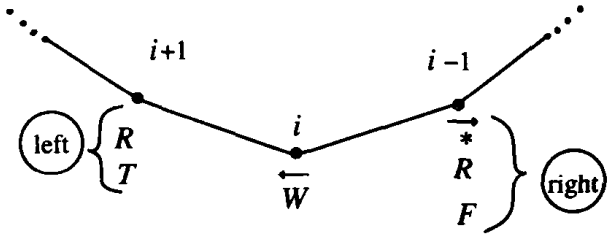


图11-13　声明11.15的情况

证明　声明11.13表明在时间4ℓ内，i到达$\vec{S}$或$i+1$到达L。后一种情况已被证明。因此假设i到达$\vec{S}$。

在该时间之内$i-1$到达L的情况也已被证明，故假设它还未到达L。则$i-1$必定还在$\vec{*} \cup R \cup F$中。此外，如果$i-1$仍在$\vec{D} \cup R \cup F$中，则它的下一步随机选择仍然是向右的。此时，根据声明11.14，至多在额外时间ℓ内，某个进程到达L。　□

现在我们回到了引理11.12的证明过程。我们已经假定有一个进程i使得$i \in \vec{W} \cup \vec{S}$和$i-1 \in \vec{*} \cup R \cup F$成立。见图11-6中左面的图。如果$i \in \vec{W}$，根据声明11.15可以得出结论——概率为1/4，等于$i+1$的下一个随机选择是*left*且$i-1$的下一个随机选择是*right*时的概率。另一方面，如果$i \in \vec{S}$，则根据声明11.14可以得出结论——概率为1/2，等于$i-1$的下一个随机选择是*right*时的概率。

366　由此我们得出：

定理11.16　对于$n \geq 3$，算法*LehmannRabin*满足属性$T \xrightarrow[\frac{1}{16}]{14\ell} C$。

我们可以反复使用定理11.16来证明最终某个进程到达临界区的概率为1。具体证明留做一道习题。

定理11.17　*LehmannRabin*算法保证良构性、互斥性和独立演进性。它同时还保证概率为1的演进性。

11.5　参考文献注释

Dijkstra[91]首先定义了哲学家用餐问题，他还针对包含全局共享信号量变量的异步共享存储器模型给出一个解法。*RightLeftDP*算法看来来自民间，Lynch[213]将它扩展为*Coloring*算法中。

Lehmann和Rabin[192]设计了*LehmannRabin*算法，并给出了一个非正式的证明概要，但没有确切说明如何对该概要进行形式化。本书的证明来自于Lynch、Saias和Segala[208]，类似风格的早期证明来自于Pnueli和Zuck[244]。Lehmann和Rabin[192]修改了*LehmannRabin*算法，从而以高概率保证锁定权。

本章的所有算法都使用共享变量。还有大量工作是针对异步网络模型中资源分配问题的，

见第20章。例如，Chandy和Misra[67]提出一个解决异步网络中通用资源分配问题的方法，并给出一个扩展来解决哲学家用餐问题的一个更动态版本——即哲学家饮水问题——其中进程的资源需求随着时间动态变化。Choy和Singh[80]、Awerbuch和Saks[37]给出了异步网络上的资源分配算法，这些算法具有良好的时间复杂度。

11.6 习题

11.1 证明并非每个互斥规格说明都有一个等价的显式资源规格说明。

11.2 扩展显式资源规格说明的定义以容纳其他资源可能性。也就是说，对于任意i，规格说明包括形式为集合R上的单调布尔公式（即只涉及∧和∨）的资源需求描述。公式$f(R_1, R_2, ..., R_k)$明确地阐明了一个由“可接受”资源集组成的集合：对于资源集S，如果S中的所有状态的值都为*真*，其他状态的值都为*假*，而公式$f(R_1, R_2, ..., R_k)$的值为*真*，则S是可接受的。该公式的意思是：可接受资源集合是由那些授权用户进入临界区的排他性所有权组成的集合。 367

a）给出适合于描述第10章的习题10.13中定义的k-排他问题的扩展显式资源规格说明。

b）证明任一扩展显式资源规格说明都有一个等价的排他规格说明。

c）证明任一排他规格说明都有一个等价的扩展显式资源规格说明。

11.3 证明引理11.1。

11.4 *研究问题*：定义一个独立演进性概念，该概念适用于以通用排他条件来表达资源分配问题。

11.5 扩展定理11.2，使之适用于比哲学家用餐问题更广的资源分配问题。尽量找出符合要求的最大的资源分配问题类。

11.6 *RightLeftDP*算法的时间上界是否严格？给出一个运行，其时间接近计算出来的界限$3c + 18\ell$。

11.7 修改*RightLeftDP*算法，使之能在进程数为奇数的环上工作。求出修改后算法的一个时间复杂度上界。这个上界应该与n无关。

11.8 构造*Coloring*资源分配算法的一个运行，其时间复杂度接近计算出来的上界$O(m^t c+km^t \ell)$。

11.9 *研究问题*：为本章的通用资源分配算法构造一个新算法，在采用的模型中，每个资源都有一个读－改－写变量与之相关，只有需要资源的进程才能够访问相关变量。你的新算法的时间性能应该比*Coloring*算法的高。 368

将你的算法扩展到更多类型的资源分配问题中，例如本章开始时介绍的两种类型。

11.10 证明存在*LehmannRabin*随机哲学家用餐算法的一个对手，其中一个特定进程锁定的概率不等于0。你能得到的最高概率是多少？

11.11 证明在引理11.11的证明中的第三种情况中提出的声明，即必定有进程j满足 $j+1 \in \tilde{*}$ 并且 $j \in \tilde{*} \cup R \cup F$。

11.12 使用定理11.16来证明关于*LehmannRabin*算法的下列结论：

a）从某个进程处于T中的任一状态开始，某个进程最终到达C中概率为1；

b）从某个进程处于T中的任一状态开始，对于任意$t \geq 0$，某个进程在时间t之内到达C中概率为$f(t)$。（你自己定义f，越小越好）。

11.13 考虑n=2时的*LehmannRabin*算法，给出并证明一个形如 $T \xrightarrow[p]{t} C$ 的有意思断言。

11.14 Flaky计算机公司的一个初级程序员在学习*LehmannRabin*算法时，提出通过删除对第一个

叉子的等待来减少时间复杂度。进程现在不等待第一个叉子，而是在测试第二个叉子的同时也测试第一个叉子。如果叉子不可用，则进程回到初始状态并重新跳跃。

耐心地向该程序员解释他的算法错在什么地方。

11.15 研究问题：能否将*LehmannRabin*算法的思想扩展到比哲学家用餐问题更通用的资源分配问题中去，且保持排他性、独立演进性和概率为1的演进性？

369

第12章 一 致 性

在本章中，我们为异步共享存储器系统的研究引入另一复杂因素：故障可能性。我们只考虑进程故障，不考虑存储器故障。实际上，我们只考虑一种最简单的进程故障：停止故障，其中进程在未给出警告的情况下停止执行。

本章研究的问题属于一种一致性问题。在第5、6和7章中我们已经仔细考察了同步消息传递系统环境中的一致性问题。对于进程故障的情况，我们已经证明了基本一致性问题是可解决的，这不但对于停止故障成立，而且对于表现更差的Byzantine故障也成立。然而，我们给出几个结果表明求解的开销（以所需进程数和所需时间来衡量）是很大的。

也许令人惊奇的是，在异步环境中，至少对于读/写共享存储器，形势变得大为不同。我们给出一个基本的不可能结果，认为在异步读/写共享存储器环境中，即使最多一个进程发生故障，基本一致性问题也是根本不可解的。我们将在第17、21章中看到，这个结果在异步网络环境中也成立。

一致性不可能性结果被认为是分布式计算理论的最基本结论之一，对任意需要某种类型一致性的分布式应用程序有着实际意义。例如，数据库系统中的进程可能会对是提交事务还是中止事务进行协商；通信系统中的进程可能就是否收到了一个消息进行协商；控制系统中的进程可能就特定的进程是否出错进行协商。不可能性结果意味着根本不存在能够达成一致性并容忍故障的异步算法。

这意味着，在实际情况中，设计者必须脱离异步模型来解决这种问题，例如，依靠定时信息或只要求一定概率的正确性来解决。

12.1 问题

我们定义共享存储器环境中的特定一致性问题。这里的描述是不正式的，但可以使用第9章中定义的模型来形式化描述。第10章包含一个针对互斥问题的类似的非正式描述，并指导如何进行形式化。现在你可以去略读一下10.1节、10.2节。

这里使用的模型与第9~11章中的模型一样，进程通过端口来与环境交互，并通过共享变量来相互通信。见图10-1的例子。设$n \geq 2$，其中n是端口的数目。进程与变量的合成以单个I/O自动机表示。如在第10、11章中的做法一样，我们用自动机U_i表示用户。一个针对一致性问题的用户集例子见例9.2.1。输入和决定的值集都为$V, |V| \geq 2$。

这次我们假定用户U_i的外部接口由输出动作$init(v)_i$（其中$v \in V$是共享存储器的一个输入值，i是一个端口名，即进程索引）和输入动作$decide(v)_i$（其中$v \in V$是一个决定值，i是一个端口名）组成。共享存储器系统的外部接口包括所有输入动作$init(v)_i$（其中$v \in V$是共享存储器系统的一个输入值）和所有输出动作$decide(v)_i$（其中$v \in V$是一个决定值，i是端口名）组成。因此，这里假设问题的输入来源于输入动作中的用户（注意该领域中的大部分研究论文都假定初始值出现在初始进程状态中的指定变量中）。每个用户自动机都必须满足一个限制：在每个运行中它最多只能执行一个$init_i$事件，即每个进程最多接收一个输入。

采用第10、11章中的方法，可以轻松地以I/O自动机来表示上述各项。在本章中，我们假设每个进程有且只有一个任务；而在习题8.8中，这个限制被放开。

我们假定进程会遭受停止故障，即在毫无预警的情况下停止执行。正式地，我们建模这种
372 情况，方法是让共享存储器系统的外部接口（外部签名）中包括特殊的$stop_i$输入动作，这些动作与进程一一对应。$stop_i$事件将会令进程i中所有之后的局部控制动作失效。$stop_i$动作并不是用户自动机外部接口的一部分，它们只是来自某个不确定的外部环境（见9.6节）。图12-1显示了完整的体系结构。通过这种故障表示方法，并根据第8章中的正式定义，系统的公平运行是这样的运行：所有不出故障的进程和任务都得到无限多的机会来执行局部控制步。如果一个系统的运行不包括*stop*事件，则称它为*无故障的*。

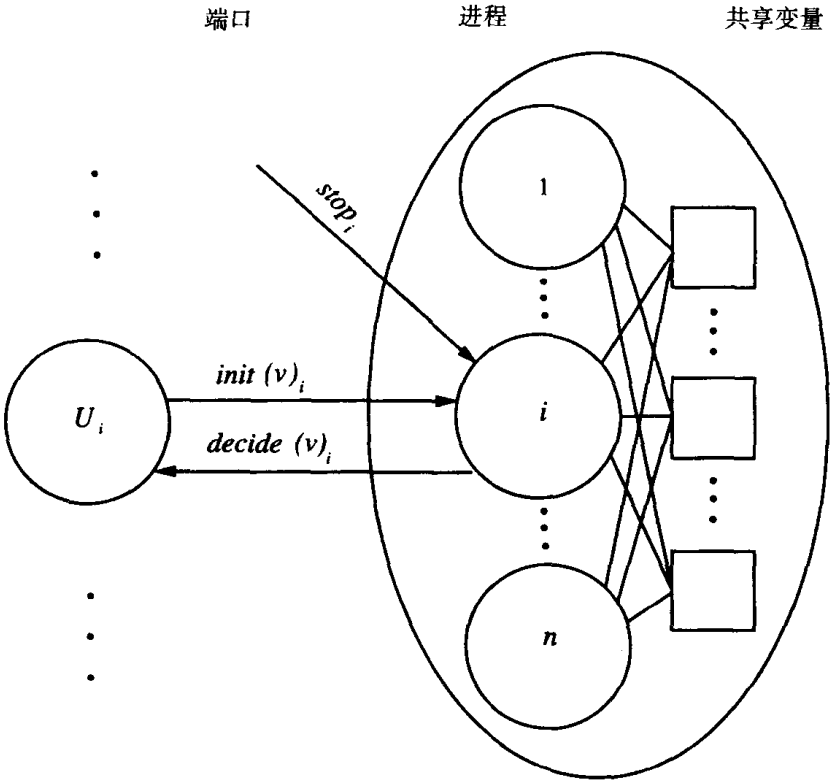


图12-1　针对一致性问题的共享存储器系统

如果一个由$init_i$和$decide_i$动作组成的序列是形如$init(v)_i$, $decide(w)_i$的序列（即空序列、由一个$init(v)_i$组成的序列，或由$init(v)_i$, $decide(w)_i$组成的双动作序列）的前缀，则称它是*良构的*。特别地，它不包含端口i上的连续输入和连续决定，也不包括那些前面没有输入与之对应的决定。
373 这里对用户自动机的假设意味着每个用户U_i都保持良构性（采用8.5.4节中的“保持”定义）。

合成系统中一个运行，无论公平与否，都必须具有以下属性：

良构性　对于任意i，U_i和系统之间的交互对i来说是良构的。

一致性　所有决定值都是相同的。

有效性　若所有发生的*init*动作都包含同一值v，则v是惟一可能的决定值。

这里的一致性和有效性的定义与6.1节中同步模型上的停止一致性问题中的定义类似，主要区别在于输入/输出约定的不同。

我们也需要某种终止条件。对于无故障运行，最基本的需求如下。

无故障终止　对于任一无故障运行，其中*init*事件在所有端口发生，则一个*decide*事件最终在每个端口上发生。

给定一个特殊的用户U_i的集合，如果共享存储器系统A保证对U_i的良构性、一致性、有效

性和无故障终止条件，则称A解决了该用户集合上的一致性问题。如果A解决了所有用户集合上的一致性问题，则称它解决了一致性问题。

对于其中任意数目的进程都可能出错的运行，以下考虑了一些涉及容错的更强终止条件：

无等待终止 对于任意公平运行，其中*init*事件在所有端口上发生，则*decide*事件在任意不出错的端口（即没有$stop_i$事件发生的端口i）上发生。

也就是说，不管其他进程是否发生故障，不出错的进程最终都做出决定。这与6.1节中针对同步环境中的停止一致性问题给出的终止条件类似。该条件被称为*无等待条件*，因为没有进程被阻塞以无休止地等待其他进程的帮助。

注意在无等待条件的声明中，输入到达所有端口，另一种等价的声明只要求输入到达端口i，对两者之间等价性的证明留做一道习题。 374

因为本章中主要的不可能性结果只涉及单个进程出故障（而不是任意进程故障）的情况，我们还需要另一个终止条件。

f故障终止，$0 \leqslant f \leqslant n$ 对于任意公平运行，其中*init*事件在所有端口上发生，如果*stop*事件最多在f个端口上发生，则一个*decide*事件在每个无故障端口上发生。

容易看出无故障终止条件和无等待终止条件分别是，f等于0和f等于n时的f故障终止条件的特例。*单故障终止条件*是$f = 1$时的特例。

引理 12.1 设A是给定的体系结构中的算法，U_i，$1 \leqslant i \leqslant n$，是用户集合。

1）若A保证针对用户U_i的无等待终止条件，则对于任意f，$0 \leqslant f \leqslant n$，$A$保证针对$U_i$的$f$故障终止条件；

2）若对于任意f，$0 \leqslant f \leqslant n$，$A$都保证针对用户$U_i$的$f$故障终止条件，则$A$保证针对$U_i$的无等待终止条件。

如果共享存储器系统保证针对所有用户集的*无等待终止条件*和f故障终止条件，则称该系统保证无等待终止条件和f故障终止条件，以次类推。

轨迹属性 如同第10、11章中的做法，我们以轨迹属性来表示本章中的正确性条件。每个轨迹属性P都有一个由$init(v)_i$、$decide(v)_i$输出和$stop_i$输入组成的签名。这些动作就是合成系统的外部动作。为了证明正确性条件，我们需要证明合成系统中的公平轨迹都在$traces(P)$中。

同步终止条件 这里的无等待条件与在6.1节的同步模型中用于讨论停止一致性问题的终止条件类似，与7.3节中用于提交问题的强终止条件也类似。而无故障终止条件则与7.3节中的弱终止条件类似。

本章的大部分内容致力于读/写共享存储器的情况，因为不可能性结果在这种情况中成立。变量可以是多写者/多读者寄存器。在本章结尾部分的12.3节和12.4节中，我们简要地讨论了其他变量类型的情况。 375

12.2 使用读/写共享存储器的一致性

在本节中，我们假定A是一个在读/写共享存储器上解决了一致性问题，并且保证1故障终止条件的算法。我们的目标在于导出矛盾，从而证明A根本不存在。

首先，不失一般性，我们对A做一些简化性限制，并给出一些所需的术语。然后，证明关于输入值的一个结果。接着，证明当强*无等待终止条件*成立时，一致性问题在读/写共享存储器上是不可解的，这较为简单。最后，证明主要结果——即系统甚至不能容忍单个故障发生。

12.2.1 限制

为简单起见且不失一般性，我们给出以下四个假设。第一，值集V为{0, 1}。第二，A在特定用户集和平凡自动机的组合中，其中每个自动机除了产生单个（任意）*init*事件之外不做其他事情。

第三，A是“确定的”，自动机只有一个初始状态；自任意自动机状态起，任一进程都最多有一个局部控制步；对于任一自动机状态和任一*init*输入，都只有一个结果自动机状态与之对应。这里并没有限制了通用性，因为如果给出不确定解法的话，可以保留每种情况中的一个状态，并删除其他所有状态（这与习题8.9中描述的确定性概念相似）。

最后，每个无故障进程都总有一个使能的局部控制步，即使在它做决定之后也如此。这也不会限制通用性，因为我们总是包括假内部步。

12.2.2 术语

*初始化*被定义为A与用户的组合的一个运行，其中用户由n个*init*步组成，每步对应一个端口，并按编号排序。则一个初始化的轨迹具有如下形式：

$$init(v_1)_1, init(v_2)_2, ..., init(v_n)_n$$

这里$v_1, ..., v_n \in V$。如果运行α的开始部分是一个初始化，则称它是*输入起始*的。我们的证明只涉及输入起始的运行。

对于有限运行α，如果在α中或在对α进行扩展的任意运行中，在
376 *decide*事件中出现的值只有0，则称α为*0价*的。另外，我们坚持认为值0实际上肯定出现在这种*decide*事件中。在一个0价运行之后，算法已经以0作为惟一的决定值，即使*decide*(0)事件实际上并没有发生也是如此。类似的，如果这种值为1，则α是*1价*的。当α是0价或1价时，称它为*单价*的；当0和1都出现在某运行中时，则称该运行是*双价*的。图12-2显示了一个双价运行。

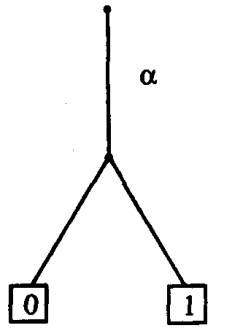


图12-2　双价运行α

以下引理指出在无故障时这种分类是完备的，亦即不能做出决定的有限无故障运行是不存在的。

引理 12.2　A的每个有限无故障运行α不是单价的，就是双价的。

证明　任意这样的α都可以被扩展成为一个无故障公平运行α'。A保证的无故障终止条件意味着：在α'中，所有进程最终做决定。　□

如果α是一个有限无故障运行而i是任意进程，则定义*extension*(α, i)为通过执行i的一步来对运行α进行扩展而得的运行。这里要用到以前所做的两个限制：每个不出错进程都总有一个使能的局部控制步，并且系统是确定性的。这个概念很容易扩展到进程编号序列中，例如，*extension*(α, ij) = *extension*(*extension*(α, i), j)。

12.2.3 双价初始化

377 我们首先证明A肯定有一个*双价初始化*，这意味着最终的决定并不仅仅取决于输入。相反，如果算法不需要容忍任何故障，则存在简单的一致性算法，使得最终值完全由输入决定。寻找这些算法的工作留做一道习题。

引理 12.3　A有一个双价初始化。

证明　如果没有的话，所有初始化都是单价的。若初始化α_0全部由0组成，根据有效性条件，α_0必然是0价的；类似的，全部由1组成的初始化α_1必然是1价的。

现在来构造一条跨越从α_0到α_1整个过程的初始化链。[⊖]在链中的每一步中，我们将一个进程的初始值从0变为1，链中两个相邻的初始化值之间的惟一区别在于，对一个进程的输入是不同的。由假设可知，链中的每个初始化都是单价的，则链中必有两个相邻的初始化，例如，α和α'，α是0价的而α'是1价的。设它们的不同之处在于进程i的初始值不同。

现在考虑对α进行扩展的任一公平运行，其中i在原来的初始化之后立即失效（即下个动作为$stop_i$），而其他进程没有失效。根据1故障终止条件，除了i之外的进程最终都做出决定。由于α是0价的，这个决定值必然是0。

现在使用相同的方法来扩展α'，可以得到决定值0。这是由于α和α'之间的惟一区别在于进程i的初始值不同，而i在两种扩展中都是在原来的初始化之后立即失效，因此剩下的进程在α'后的表现应该与在α后的表现一样。参见图12-3。

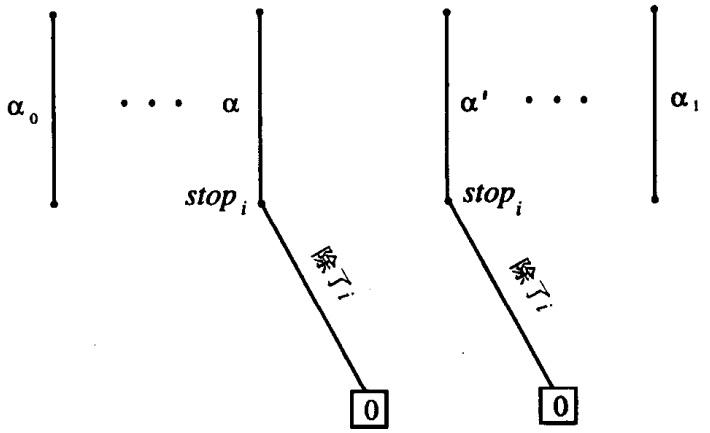


图12-3 引理12.3的构造

这与α'是1价的假设矛盾。 □

12.2.4 无等待终止的不可能性

现在来证明第一个（最简单的）不可能性结果——无等待终止的不可能性。在本小节中，算法A具有无等待终止属性，该属性比以前假定的1故障终止属性更强。无等待终止条件将用来导出矛盾。

要导出矛盾，可以去寻找一种决定方法。特别地，一个决定者运行α被定义为满足以下条件的有限无故障输入起始运行：

1）α是双价的。

2）对于所有i，$extension(\alpha, i)$是单价的。

因此，在一个决定者运行之后，尚未做出决定，但任一额外（非$stop$）进程步将确定决定。我们来证明A（具有无等待终止属性）必有一个决定者运行。

引理 12.4 A有一个决定者运行。

证明 假设该引理不成立，即任一双价无故障输入起始运行都有一个双价无故障扩展。

从一个双价初始化（引理12.3保证该初始化是存在的）开始，可以生成一个无限的无故障输入起始运行α，其中α的所有前缀都是双价的。因此，在α中，没有进程曾经做出决定。构造

378

⊖ 这个链的结构与定理6.33证明中用到的链类似。协议需要的轮数下界在同步环境下设置。

α的过程很简单：在每一阶段，从一个双价无故障输入起始运行开始，执行一步将该运行扩展成为另一个双价无故障运行。由上面的假设可知，这种构造是可行的。

由于α是无限的，它必然包含无限多的某进程步，设该进程为进程i，我们来证明i必然在α中做出决定，从而产生矛盾。

为此，对于每个只执行有限多步的进程j，将$stop_j$事件插入到α中，并放在该进程的最后一步之后的第一个位置。称产生的运行为α'，则α'是一个没有失效进程的公平运行。然后，无等待条件终止条件意味着i必然在α'中做决定。但是，由于在进程i眼中α和α'是一样的，则i也在α
379 中做决定。这就产生了证明该引理所需的矛盾。 □

现在导出矛盾来证明无等待终止条件的不可能性结果。

引理 12.5　A并不存在。

证明　根据引理12.4，固定一个决定者运行α。由于α是双价的，故存在两个进程，如i和j，使得在α之后，i的一步导致一个0价运行；j的一步导致一个1价运行。也就是说，$extension(\alpha, i)$是0价的，而$extension(\alpha, j)$是1价的。显然$i \neq j$，参见图12-4。

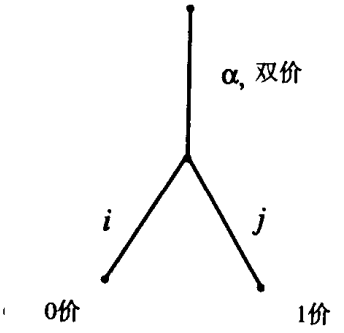


图12-4　运行α是一个决定者，$extension(\alpha, i)$是0价的，而$extension(\alpha, j)$是1价的

以下分情况来完成证明，每种情况都产生矛盾：

1）进程i的步是一个读步；

对$extension(\alpha, j)$进行扩展，其中没有进程失效，进程i不执行任何步，而其他进程执行无限多步。对于除了i之外的所有进程，该运行看起来类似于一个其中进程i立即失效，而其他进程没有失效的公平运行。因此，根据无等待终止条件（事实上，1故障终止条件就足够了），除了i之外的所有进程最终都做出决定；又因为$extension(\alpha, j)$是1价的，这些进程必然决定出1。

现在，对于除了i之外的进程来说，α之后的状态和$extension(\alpha,i)$之后的状态是不可分的（见9.3节中的定义）。由于i的步是一个读步，故它只能改变进程i的状态。将以前在α之后运行的同一后缀（该后缀以j的步开始）取出来后放在$extension(\alpha,i)$之后。此时，除了i之外的所有进程都决定出1，这与$extension(\alpha, i)$是0价的假设相矛盾。
380 参见图12-5。

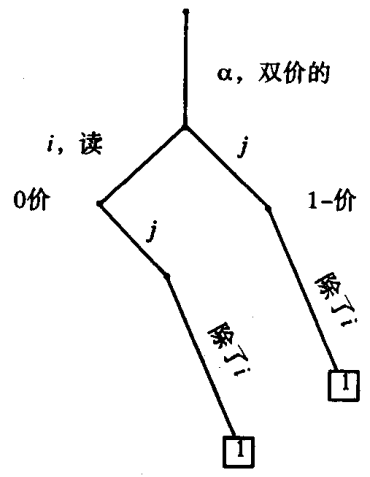


图12-5　第1种情况的构造

2）进程j的步是一个读步；

这种情况与第1种情况类似，证明过程类似。

3）进程i的步和进程j的步都是写步。

分两种情况考虑：

a）进程i和j写不同的变量；

考虑两种对α进行扩展的运行，在第一种中i先执行步，然后j执行步；另一种中j先执行步，然后i执行步。由于这两步涉及不同的进程和不同的变量，在每个运行之后的系统状态是相同的。参见图12-6。

这样在0价运行之后或者在1价运行之后到达同一系统状态。如果从该状态开始无故障地运行所有进程，则这些进程必须做出决定。然而，任何的决定都产生矛盾。例如，如果决定值是0，则我们得到一个从1价前缀扩展而来的运行，其决定值为0。

b）进程i和j写相同的变量。

如第一种情况中的做法，在$extension(\alpha, j)$之后运行除了i之外的所有进程，直到它们决定出1。此时，对于除了i之外的进程，$extension(\alpha, j)$之后的状态和$extension(\alpha, ij)$之后的状态是不可分的，原因在于j的步改写了i先前写入的值，故i的步只在进程i的状态中留下记忆。将以前在$extension(\alpha, j)$之后运行的同一后缀取出来后放在$extension(\alpha, i\ j)$之后。此时，除了i之外的所有进程都决定出1，这与$extension(\alpha, i)$是0价的假设相矛盾。参见图12-7。

381

382

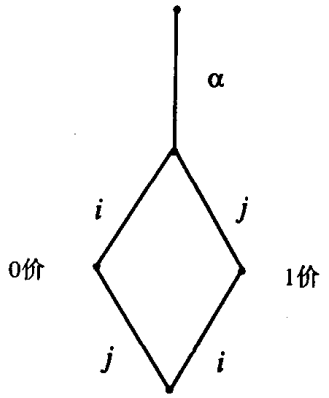


图12-6 情况3(a)的构造

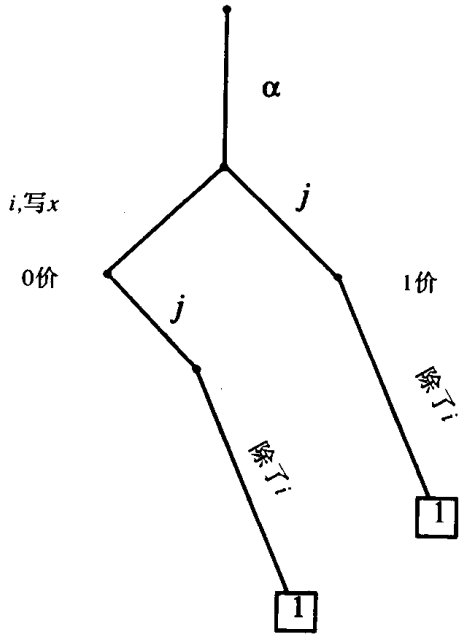


图12-7 情况3(b)的构造

在所有情况中我们都导出了矛盾，因此不存在这样的算法A。 □

这样我们证明了第一个不可能性定理：

定理 12.6 对于$n \geq 2$，不存在一个针对读/写共享存储器来解决一致性问题且保证无等待故障终止的算法。

12.2.5 单故障终止的不可能性结果

注意前一节中对定理12.6的证明并不适用于1故障终止的情况，问题出现在引理12.4的证明中，在那里使用无等待终止条件来断定进程i在不出故障的一个公平运行中必须做出决定。在本节中，我们增强定理12.6以取得针对1故障终止的系统的相应结果。

这次的证明基于以下引理。该定理认为：可以对一个双价运行进行扩展，从而允许当双价性仍然被保持时一个给定的进程执行一步。

引理 12.7 如果α是A上的一个双价无故障输入起始运行，且i是任一进程，则存在α的一个无故障扩展α'使得$extension(\alpha', i)$是双价的。

参见图12-8。

383

在钻研本引理的证明之前，你可以跳到定理12.8的证明中去看看引理12.7如何被用于证明

不可能性结果。

证明　通过反证法来证明。假设该引理不成立，则必然存在A上的双价无故障输入起始运行α和进程i，使得对于α的任意无故障扩展α'，$extension(\alpha', i)$是单价的。这意味着$extension(\alpha, i)$是单价的。不失一般性，设$extension(\alpha, i)$是0价的。

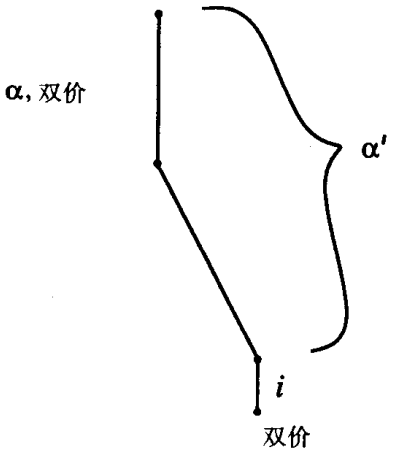


图12-8　允许进程i执行步时保持双价

由于α是双价的，α有一个包含决定值1的扩展α''。不失一般性，设$extension(\alpha, i)$是无故障的，否则可以删除任意$stop$动作，而决定不受影响。则$extension(\alpha'', i)$必然是1价的。当i在从α到α''的路径上的任意一点上执行一步时，看看会发生什么？参见图12-9。

在路径的开始，i的步是0价的；在结尾则是1价的；在中间点上是单价的。因此，路径上必有两个相邻点使得第一点是0价的而第二点是1价的。令α'是从开始点开始的运行，见图12-10。

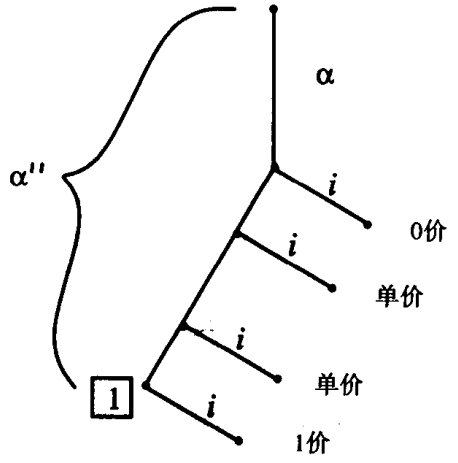


图12-9　进程i的步导致单价

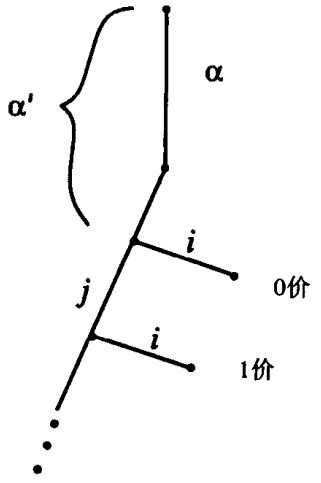


图12-10　i产生不同价的连续两点

设j是执行中间步的进程，我们认为$j \neq i$，因为如果$j = i$，则i的一步导致0价而i的两步导致1价；由于进程是确定性的，这给出一个具有1价扩展的0价运行，而这是不合理的。

384

与引理12.5的证明类似，以下分情况来完成证明，每种情况都产生矛盾：

1）进程i的步是读步。

容易证明在$extension(\alpha', ji)$和$extension(\alpha', ij)$之后的状态对于除了i之外的所有进程是不可区分的。这是因为两种运行涉及的i的步都是读步，只会影响进程i的状态。

按照以下方式来扩展$extension(\alpha', ij)$：i不再执行任何步，而其他所有进程都执行无限步。根据1故障终止条件，除了i之外的所有进程都必须最终做出决定，又因为

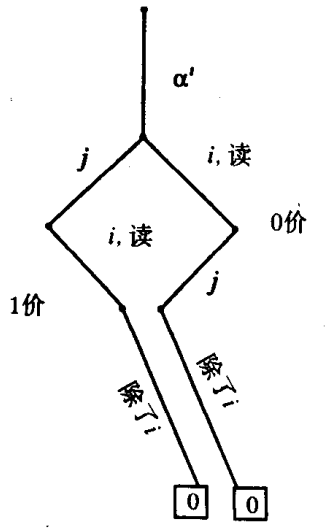


图12-11　第一种情况的构造

extension(α', *i*)是0价的，故必然决定出0。由上述的不可区分性证明可知，只需将以前在*extension*(α',*ij*)之后运行的同一后缀取出来放在*extension*(α', *ji*)之后就足够了。此时，进程也决定出0，这与*extension*(α', *ji*)是1价的假设相矛盾，参见图12-11。

2）进程*j*的步是读步。

这种情况的证明过程与第一种情况的证明过程类似。此时，在*extension*(α',*i*)和*extension*(α', *ji*)之后的状态对于除了*j*之外的所有进程都是不可区分的。让除了*j*之外的所有进程在*extension*(α', *i*)之后运行，迫使它们最终决定出0。然后在*extension*(α', *ji*)之后以相同方法运行这些进程，它们又决定出0，从而与*extension*(α', *ji*)是1价的假设矛盾。 385

3）进程*i*的步和进程*j*的步都是写步。

分两种情况考虑：

a）进程*i*和*j*写不同的变量。

这种情况的证明可以采用引理12.5的证明过程中的相同技术来进行，参见图12-6。相同的矛盾将被导出。

b）进程*i*和*j*写相同的变量。

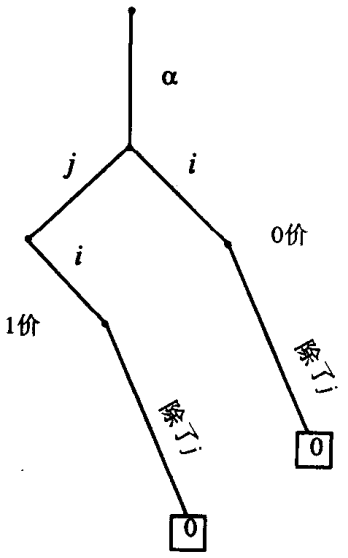
在这种情况中，在*extension*(α', *i*)和*extension*(α', *ji*)之后的状态对于除了*j*之外的进程是不可区分的，原因在于*i*的步改写了*j*的先前步。让除了*j*之外的所有进程在*extension*(α',*i*) 和*extension*(α', *ji*)之后运行时，将产生与前面相同的矛盾。参见图12-12。 □

现在可以证明出主要定理。

定理 12.8 对于$n \geq 2$，不存在一个针对读/写共享存储器来解决了一致性问题且保证1故障终止的算法。 386

证明 我们利用引理12.7来构造一个公平无故障输入起始运行，其中没有进程曾经做出决定。这与无等待终止条件的要求相矛盾。

构造过程从一个双价初始化开始，引理12.3保证这样的初始化是存在的。然后反复地扩展当前运行：以轮换的顺序，第一个扩展至少包含进程1的一步，第二个扩展至少包含进程2的一步，以次类推；所有扩展都保持双价性且无故障发生。根据引理12.7，每次扩展都是可行的。



所得的运行是公平的，因为每个进程都执行了无限多步。然而，没有进程曾经做出决定。这就导出了所需的矛盾。 □

12.3 读/改/写共享存储器上的一致性问题

与读/写共享存储器的情况相比，使用读-改-写共享存储器来解决一致性问题并保证无等待终止条件是很容易的。实际上，一个单独的读-改-写共享变量就足够了。 387

***RMWAgreement*算法：**

共享变量的初始值为*unknown*。每个进程都访问这个变量，如果看到变量的值为*unknown*，则将值改成进程的初值并在该值上做决定。另一方面，如果看到变量的值为$v \in V$，则不改变变量的值，而是将变量的这个值作为进程的决定值。

除了增加一些故障处理的代码外，进程i的*RMWAgreement*算法的前提－结果代码与例9.1.1中的代码相同。也就是说，在*RMWAgreement*算法的状态中包含额外的*stopped*部分，*stopped*部分是一个进程集合，其初值为空集；还包含一个负责将i加到*stopped*部分中去的$stop_i$动作。$access_i$和$decide_i$动作具有额外的前提：$i \notin stopped$，且$init_i$动作只在$i \notin stopped$时发生变化。

定理 12.9　*RMWAgreement*算法解决了一致性问题并保证无等待终止。

证明　这是很浅易的。无等待终止条件成立，因为进程i在接收到一个$init_i$输入之后立刻能够执行一个$access_i$动作，接着执行一个$decide_i$动作。一致性条件和有效性条件也成立，因为第一个执行*access*动作的进程建立了公共的决定值。　□

12.4　其他共享存储器类型

我们还可以考虑除了针对读/写共享存储器和读-改-写共享存储器之外的其他类型的共享存储器的一致性问题。例如，考虑变量具有交换、测试-设置、取-加、和比较-交换操作时的情况。这些操作在例9.4.3中定义。以下定理是已经取得的著名结果中的一个：

定理 12.10　使用单个只允许比较-交换操作的共享变量，对于任意n，保持无等待终止条件的一致性问题都是可解的。

定理 12.11　如果$n \geq 3$，且共享变量允许交换、测试-设置、取-加、读和写操作的任意组合，则使用这种共享变量来解决保持无等待终止条件的一致性问题是不可能的。

证明　该证明留做一道习题。　□

388

12.5　异步共享存储器系统中的可计算性*

在具有停止故障的异步共享存储器模型上的“决定问题”中，一致性问题只是其中一个例子。在本节中，我们定义了决定问题的通用概念，给出一些例子，并给出（但不证明）一些典型的可计算性结果。

一个决定问题的定义基于一个决定映射的初步定义。设输入的固定值集为V，对于长度为n的输入向量，一个决定映射D是由长度为n的合法决定向量组成的非空集合$D(w)$。向量w代表了进程1, ..., n的输入，向量中的元素按进程编号排序；类似的，$D(w)$中的每个向量都代表了进程1, ..., n的决定，向量中的元素也按进程编号排序。

决定映射D用于对异步共享存储器模型可解的问题进行形式化。与一致性问题中的情况一样，共享存储器模型的外部接口由$init(v)_i$、$decide(v)_i$、$stop_i$动作组成，良构性条件和各个终止条件的定义与一致性问题中的相应定义一模一样。不过，与一致性问题同时定义了一致性条件和有效性条件不同，这里只需要有效性条件：

有效性　在任意其中*init*事件发生在所有端口的运行中，进程做出的决定组成了$D(w)$中的某个向量，其中w是给定的输入向量。

例 12.5.1　作为决定性问题的一致性问题

基于以下定义的决定映射D，一致性问题是决定性问题的一个例子。对于任一向量$w = v_1, ..., v_n$，其中$v_1, ..., v_n$是V中的输入，由合法决定向量组成的非空集合$D(w)$的定义如下：

1）如果$v_1 = v_2 = ... = v_n = v$，则$D(w)$包含单个向量$x_1, x_2, ..., x_n$，其中$x_1 = x_2 = ... = x_n = v$。

2）如果对于某个i和某个j，有$v_i \neq v_j$，则$D(w)$由向量$x_1, x_2, ..., x_n$组成，其中$x_1 = x_2 = ... = x_n$。

容易看出基于D的决定问题等同于一致性问题（一个显著的差异在于：通用决定问题的定义中只提到了那些其中*init*输入到达所有端口的运行，而一致性问题的定义还涉及其他运行。但这个差异并不重要，因为任意有限运行都可以被扩展成为其中输入到达所有端口的运行）。 389

决定性问题的另外两个重要例子是k一致性问题和近似一致性问题，我们已经在第7章的同步网络模型中研究了这两个问题。在k一致性问题（k是任意正整数）中，一致性问题的一致性条件和有效性条件变成：

一致性 在任意运行中，存在V的一个子集W，$|W| = k$，使得所有决定值都在W中。

有效性 在任意运行中，任意进程的任意决定值都是某一进程的初始值。

这里的一致性条件比一般的一致性条件弱，因为它允许k个而不是1个决定值。有效性条件比针对一般一致性的有效性条件强一点。将k一致性问题形式化为决定问题是容易的。见以下定理，其证明过程留做一道习题。

定理 12.12 在使用单写者/多读者寄存器的异步读/写共享存储器模型中，保持$k-1$故障终止条件的k一致性问题是可解的。

定理 12.13 在使用多写者/多读者寄存器的异步读/写共享存储器模型中，保持k故障终止条件的k一致性问题是不可解的。

在近似一致性问题中，值集V是实数集合，进程可以在消息中发送实数值的数据。与一致性问题中进程必须达成一致的情况不同，这里要求进程近似地（在一个小的正容差ε之内）达成一致就行了。换而言之，一致性问题和有效性问题变成：

一致性问题 对于任意运行中的任意两个决定值，其中一个必在另一个的容差ε之内。

有效性问题 在任意运行中，任意决定值都在初始值的范围之内。

与前面一样，将这些问题形式化为决定性问题是容易的。 390

定理 12.14 在使用单写者/多读者寄存器的异步读/写共享存储器模型中，保持无故障终止条件的近似一致性问题是可解的。

证明 该证明留做一道习题。 □

作为本章的结束，我们给出一条定理来说明保持1故障终止的决定性问题是不可解的。该定理是定理12.8的扩展。

对于任一由长度为n的向量组成的集合，其中向量的元素是V中的元素，定义一个图。该图的顶点是长度为n的向量；边是向量对，其中两个向量只有一个位置的值是不同的。

定理 12.15 令D是这样一个决定映射：在异步读/写共享存储器模型中，与之相关的保持1故障终止条件的决定性问题是可解的。则必然存在一个决定映射D'，对于所有w，有$D'(w) \subseteq D(w)$，且以下条件成立：

1）如果输入向量w和w'只有一个位置的值不同，则存在$y \in D'(w)$和$y' \in D'(w')$，使得y和y'最多有一个位置的值不同。

2）对于任意w，由$D'(w)$定义的图是连通的。

定理12.15的证明留做一道习题；其证明思想类似于定理12.8的证明思想。

12.6 参考文献注释

Fischer、Lynch和Paterson[123]最先证明了容错系统中一致性的不可能性，其结果是针对1

故障终止的异步消息传送环境的。后来，该结果及其证明被Loui和Abu-Amara[199]扩展到异步共享存储器模型（一个稍强的模型）上（见第17章中介绍的异步共享存储器模型与异步网络模型的关系）。Loui和Abu-Amara[199]，以及Herlihy[150]相互独立地证明了无等待一致性的不可能性结果。本章的阐述基于[199]中的证明。

使用除了读/写变量之外其他共享变量类型的一致性问题的结果来自Herlihy[150]的工作。Herlihy的论文不但对能够解决一致性问题的变量类型进行分类，还决定了哪些类型能够“实现”
391 哪些其他类型。

k—致性问题最先由Chaudhuri[73]在异步网络环境中提出。Chaudhuri证明了保持$k-1$故障终止条件的k—致性问题是可解的，但没有回答保持k故障终止条件的k—致性问题是否可解的问题。该问题的（否定）答案由Herlihy与Shavit[152]、Borowsky与Gafni[55]和Saks与Zaharoglou[253]给出。对于能在易错异步读/写共享存储器系统解决的问题，Herlihy与Shavit在这些问题的拓扑特征上下文中给出了结果。他们进一步在[151]中扩展了这种特征。特征考虑了输入向量的受限集，而不只是本书中的完全集。

近似一致性问题最先由Dolev、Lynch、Pinter、Start和Weihl[98]定义。他们的工作针对异步网络系统。Attiya、Lynch和Shavit[24] 为近似一致性问题设计了一个无等待异步共享存储器算法。

基于Moran和Wolfstahl[230]早期取得的不可能性结果，Biran、Moran和Zaks[51]刻划了那些能在异步读/写共享存储器环境中解决的保持1故障终止条件的决定问题。这种刻划考虑了输入向量的受限集。定理12.15来自于这两篇论文的结果。这些结果首先在异步网络环境中被证明，然后扩展到异步读/写共享存储器环境中。

Chor、Israeli与Li[78]，Abrahamson[2]，和Aspnes与Herlihy[16]给出使用读/写共享存储器的一致性问题的随机化解法。

12.7 习题

12.1 证明无等待终止条件的更强形式——其中输入只需到达端口i——与书中给定的形式等价。即示范如何修改给定的保证良构性、一致性、有效性和无等待终止条件的算法A，使得修改后的版本除了保证更强无等待终止条件之外，还保证这些相同条件。

12.2 描述一个在读/写共享存储器模型中解决一致性问题的算法（没有任何容错要求），且
392 a）其中存在一个双价初始化。
b）其中所有初始化都是单价的。

12.3 判断题
a）如果A是一个在满足12.1节中限制的读/写共享存储器模型中的（无容错）一致性协议，且A拥有一个双价的初始化，则A必定有一个决定者运行。
b）如果A是一个如a）所说的一致性协议，且是一个双进程协议，则A必有一个决定者运行。

12.4 证明：即使我们用以下更弱条件来代替定理12.8中的有效性条件，定理12.8仍然成立：存在两个输入起始运行α_0和α_1，其中α_0的某些进程决定出0，而α_1的某些进程决定出1。

12.5 重新考虑使用读/写共享存储器的一致性问题。这次考虑一个比通用停止故障更受限的模型，其中进程只能在每次计算开始时发生故障（也就是说，所有*stop*事件在其他所有事件之前）。能否在该模型上解决一致性问题，且保证

a）1故障终止？

b）无等待终止？

在每种情况中，给出一个算法或者不可能性结果。

12.6 证明：在读/写共享存储器中保证1故障故障终止条件的任一一致性协议都必然具有一个双价初始化。

12.7 证明定理12.10。

12.8 证明对于 $n \geq 3$，使用任意数目的共享变量都不能解决保证无等待终止的n进程一致性问题，其中每种变量都是干扰（interfering）类型，即如果a和b是变量类型的调用，则令$f_2(a, v)$代表$f(a, v)$的第二个投影（即变量的新值），以下条件中至少有一个成立：

a）(a和b可交换) 对于所有$v \in V$，$f_2(a, f_2(b, v)) = f_2(b, f_2(a, v))$

b）(a改写b) 对于所有$v \in V$，$f_2(a, f_2(b, v)) = f_2(a, v)$

c）(b改写a) 对于所有$v \in V$，$f_2(b, f_2(a, v)) = f_2(b, v)$ 393

这里使用了9.4节中的概念。

12.9 利用习题12.8的结果来证明定理12.11。

12.10 给出决定映射以将以下问题正式地表示为决定问题：

a）k一致性问题。

b）近似一致性问题。

12.11 证明定理12.12。

12.12 证明定理12.13（提醒：本题难度较高）。

12.13 考虑 $n = 2$（进程）时的近似一致性问题。给出一个在使用单写者/多读者寄存器的异步共享存储器模型中解决该问题的无故障算法。证明其正确性，并给出时间复杂度。

12.14 将习题12.13中的结果扩展到任意进程数目n的情况，即证明定理12.14。

12.15 Flaky计算机公司中一个有经验的软件设计者想出了一个解决本章中一致性问题（对于任意数目的停止故障，$V = \{0, 1\}$）的绝妙主意：将0和1视为实数，将近似一致性问题的无等待解法作为一个“子过程”。一旦进程获得近似一致性问题子过程的答案，则简单地将答案舍入为0或者1，从而得到最终决定。解释为什么她的主意是错误的。

12.16 考虑你在习题12.13中设计的双进程无等待近似一致性算法（如同12.2.1节中的定义，假设你的算法是确定性的）。对于任一向量(v_1, v_2)，$D'(v_1, v_2)$是这样的决定向量集合：向量实际上从输入起始运行中获得，其中进程1具有输入v_1，进程2具有输入v_2。

a）描述集合$D'(0, 1)$和其相关图（在定理12.15前定义）。

b）考虑无故障无限运行，其中进程1和进程2首先分别接收输入0和1，然后交替地执行步1, 2, 1, ...。对于该运行的每个输入起始前缀α，描述实际上从α的扩展中获得的决定向量的集合。 394

c）对于任意输入向量(v_1, v_2)，描述$D'(v_1, v_2)$。

d）证明对于任意(v_1, v_2)，$D'(v_1, v_2)$定义的图是连通的。

12.17 证明定理12.15。（提示：固定在读/写共享存储器模型中解决了保持1故障终止条件的D的任一算法A。对于任意输入向量w，定义$D'(w)$为实际上从A（具有输入向量w）的输入起始运行中获得的向量的集合）。第1部分的证明类似于引理12.13的证明。第2部分使用反证法，其过程与定理12.8的证明过程相似。在每个有限无故障输入起始运行之后，考虑可获得的决定向量的集合是否连通。使用类似于引理12.7的方法（通过允许任一给定进程i执行一步，将任一“不连通”的α扩展为另一“非连通”的α'）。

12.18 使用定理12.15来证明定理12.18。

12.19 使用定理12.15来证明：除了一般一致性问题之外的其他保持1故障终止条件的决定问题，在异步读/写共享存储器模型中都是不可解的。尽量找出所有有趣问题，这些问题能以这种方法来证明其不可能性。

12.20 扩展定理12.15中的条件来得到决定问题的一个通用特征，使之保持1故障终止条件时在异步读/写共享存储器模型中是可解的（提醒：本题难度很高）。

395
~
396

第13章 原子对象

这一章是异步共享存储器模型的最后一章，在这里我们介绍*原子对象*。一个特定类型的原子对象很像一个同一类型的普通共享变量。区别在于原子对象可以被几个进程同时访问，而对共享变量的访问却只能不可分地进行。虽然对原子对象的访问是并发的，但是原子对象可以保证当过程获得回应时，在它看来每个时刻只发生一次访问，且回应的顺序和调用的顺序相一致。原子对象有时候也被称做可线性化对象。

除了具有原子性的属性之外，绝大多数已研究过的原子对象都满足有意思的容错条件。其中最强的条件是*无等待终止*条件。它要求在无故障端口上的任何调用最终获得一个回应。这个属性可以弱化为：仅当所有故障被限定在某一指定的端口集合I或某一数量f的端口上时，回应才是必需的。我们在这一章考虑的惟一故障类型是停止故障。

有人建议把原子对象作为构造多处理器系统的构件。其思想是，你可以从基本的原子对象开始，比如，单写者/单读者的读/写原子对象，它们比较简单，且可以由硬件来提供。随后，从这些基本的原子对象开始，你可以成功地构造功能更强的原子对象。最后的系统结构应该是简单的、模块化的、正确的。然而，尚未解决的问题是如何构造原子对象来在实践中提供足够快的、有用的回应。

然而作为异步网络系统的构件，原子对象的有用性是无可争辩的。很多分布式网络算法被设计成为能够给用户提供一种类似于集中的、一致的共享存储器的东西。形式上，其中很多可 397
以被视为原子对象的分布式实现。我们将在17.1节和18.3.3节中看到有关这种现象的几个例子。

在13.1节，我们引入形式框架用来研究原子对象。也就是说，我们定义原子对象并给出它们的基本性质，尤其是它们和同类型共享变量之间联系的结果，以及表明它们如何用于系统构建方面的结论。

然后在本章的余下部分，我们将给出一些算法，这些算法根据其他类型的原子对象（即共享变量）来实现一些特殊类型的原子对象。我们考虑的原子对象的种类有读/写对象、读－改－写对象和快照对象。我们给出的结果仅仅是一些例子——在相关的研究文献中还有很多这样的结果，在这个领域中也还有很多研究工作要做。

13.1 定义和基本结论

我们首先定义原子对象和它们的基本属性，然后给出一个用来构造给定类型的规范无等待原子对象的方法。然后再证明一些关于构造原子对象的基本结论，以及使用它们来代替共享存储器系统的共享变量的结论。这些结论可以用来证明从其他原子对象来分级构造原子对象的正确性。

这一节中的许多概念是相当难以捉摸的。但是它们不仅对本章中的结论，而且对第17章、第21章中涉及容错性的资料都很重要。所以我们会在这儿放慢步骤，比平常更形式化地展现我们的思想。第一次阅读的时候，你可以跳过证明而只读定义和结论。事实上，你可以只读13.1.1节的定义，然后跳到13.2节，在需要的时候再回到本节。

13.1.1 原子对象的定义

原子对象的定义基于9.4节变量类型的定义。你现在应该重新读一下9.4节。回忆一下，一个变量类型包括一个值集V、一个初始值v_0、一个调用集合、一个回应集合和一个函数f：调用 ×

$V \to$回应 $\times V$。当在一个有特定值的变量上有一个特定的调用时，f指定回应和新的值作为结果。

再回忆一下，一个变量类型的运行就是有限序列v_0，a_1，b_1，v_1，a_2，b_2，v_2，…，v_r和无限
398 序列v_0，a_1，b_1，v_1，a_2，b_2，v_2，…，其中a和b分别是调用和回应，并且相邻四元组与f一致。另外，这个变量类型的轨迹就是从这个变量类型的运行中提取出来的a和b序列。

如果T是一个变量类型，则一个类型T的*原子对象*A被定义为一个I/O自动机（应用第8章对I/O自动机的通用定义），它满足我们将在以下几页中描述的一系列属性。尤其是，它必须拥有特定类型的外部接口（外部签名），且必须满足特定的“良构性”、“原子性”和活性条件。

我们从描述外部接口开始。假设A通过编号为1, ..., n的n个端口来访问。对每个端口i，A有一些形如a_i的输入动作，其中a是变量类型的一个调用。还有一些形如b_i的输出动作，其中b是变量类型的一个回应。如果a_i是一个输入动作，它意味着在端口i上a是一个允许的（或称为合法的）调用；而如果b_i是一个输出动作，它意味着在端口i上b是一个合法的回应。我们假设一个技术性条件：如果a_i是端口i上的一个输入，并且$f(a, v) = (b, w)$对于某个v和w成立，那么b_i就是端口i上的一个输出。也就是说，如果调用a在端口i上是合法的，那么对a的所有可能回应在端口i上也是合法的。

另外，由于我们将考虑原子对象对于停止故障的复原性，所以我们假设每一个端口i上都有一个输入$stop_i$。原子对象的外部接口如图13-1所示。

例 13.1.1　读/写原子对象的外部接口

我们描述一个在域V上的1－写者/2－读者原子对象的外部接口。这个对象有三个端口，我们依次标为1、2、3。端口1是写端口，仅支持写操作；端口2和端口3是读端口，仅支持读操作。更精确地说，在端口1上，对于所有$v \in V$，有形如$write(v)_1$的输入动作和简单的输出动作ack_1。在端口2上，有简单的输入动作$read_2$和对于所有$v \in V$的形如v_2的输出动作，端口3的情况同端口2的情况类似。在端口1、2、3上，分别有$stop_1$、$stop_2$、$stop_3$输入动作，其外部接口如图13-2所示。

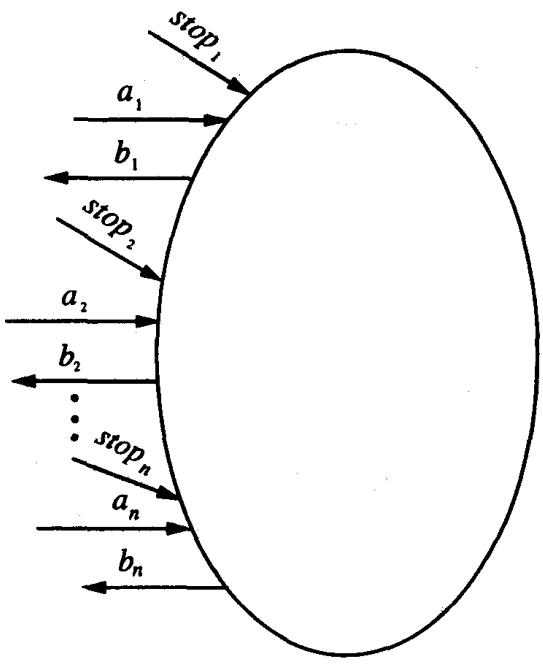


图13-1　一个原子对象的外部接口

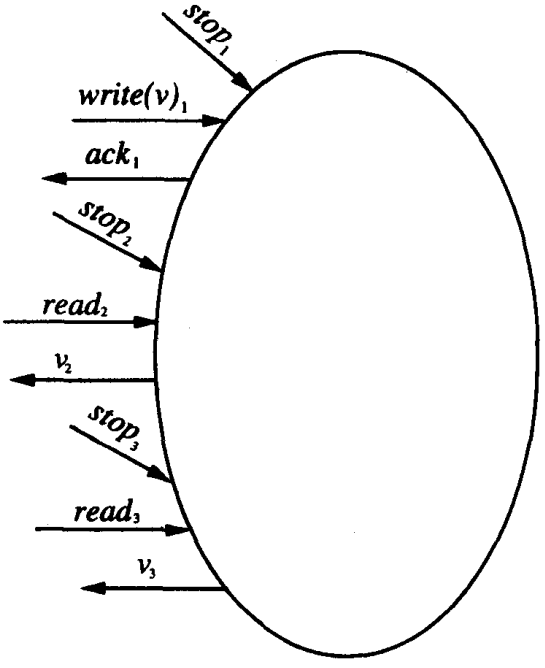


图13-2　一个1-写者/2-读者的读/写原子对象的外部接口

接下来，我们描述特定变量类型$\mathcal{T}$的原子对象自动机A的所需行为。在第10~12章中我们假设A是由一组用户自动机U_i合成的，其中用户自动机与端口一一对应。U_i的输出被认为是A在端口i上的调用，而U_i的输入则被认为是A在端口i上的回应。$stop_i$动作不是U_i的签名的一部分。我们认为它不是由U_i产生的，而是来自于未指定的外部源。

399 ~ 400

还有一个需要为U_i假设的属性是“良构性”，其定义如下：对于一个用户U_i的外部动作序列，如果序列以一个调用开始，且序列中调用和回应交替出现，则称它为良构的。我们假设对于任意i，U_i保持良构性（参照8.5.4节中“保持”的形式化定义）。也就是说，我们假设在每一个端口上的调用操作是严格串行的，每一个调用都等待对前一个调用的回应。注意这个串行要求仅限于单个端口，我们允许在不同端口上调用的并发性[⊖]。在本章中，我们用U来代表各独立用户自动机U_i的合成，$U = \prod U_i$。

我们要求包含A和U的合成系统$A \times U$满足几个性质。首先，它的良构性条件与第10、11、12章中用到良构性条件相似。

良构性 在$A \times U$的任一运行中，U_i和A之间的交互对于任意i都是良构的。

因为我们已经假定用户保持良构性，这相当于说A也保持良构性。所以在合成系统$A \times U$中，每一个端口都是从一个调用开始，调用和回应交替出现。

接下来的条件是最难理解的。对某个特定的变量类型$\mathcal{T}$，这个条件描述了操作的显著原子性。注意：$\mathcal{T}$的一条轨迹描述了所有操作串行执行时对调用序列的正确回应，也就是说，其中第一个调用之后的每一个调用都在等待对前一个调用的回应。原子性条件说明合成系统产生的每一条轨迹——允许不同端口上操作的并发调用——“看起来好像是”$\mathcal{T}$的某一条轨迹。

这些条件的形式表述稍微有点复杂，因为即使在$A \times U$的运行中某些调用——某些端口上的最后调用——未完成时，这些条件也必须成立。因此，我们规定在每个运行中，已经完成操作和某些未完成操作好像是在它们的时间间隔中的某一点上瞬时执行的。

为了定义$A \times U$系统的原子性，我们首先给出一个更基本的定义——用户动作序列的原子性。即，设β是一个对每个i都是良构的$A \times U$外部动作（有限或无限）序列（也就是说，对每个i，$\beta|ext(U_i)$对i是良构的）。如果能够做到以下各点，则称β满足$\mathcal{T}$的原子性：

401

1）对于每一个完成操作π，在β中π的调用和回应之间某处插入序列化点$*_\pi$。

2）选出一个未完成操作子集Φ。

3）对于每一个操作$\pi \in \Phi$，选出一个回应。

4）对于每一个操作$\pi \in \Phi$，在β中π的调用之后某处插入序列化点$*_\pi$。

我们选择操作和回应并插入序列化点，使得按如下方法构造的调用和回应序列正好是变量类型$\mathcal{T}$的一条轨迹：

对每个完成的操作π，将在β中出现的调用和回应事件（按照其原来顺序）移到序列化点$*_\pi$处。（也就是说，“缩小”操作π与它的序列化点之间的间隔。）同样的，对每一个操作$\pi \in \Phi$，把在β中出现的调用放在$*_\pi$处，其后跟着选出的回应。最后移除所有不属于Φ的未完成操作（即$\pi \notin \Phi$）的调用。

注意原子性条件只依赖于调用和回应事件——它并未涉及$stop$事件。我们很容易把这个定义扩展到A和$A \times U$的运行中去。即假设α是一个对每个i都是良构的运行（也就是说，对每个i，$\alpha|ext(U_i)$对i是良构的），如果α的外部动作的序列$trace(\alpha)$满足$\mathcal{T}$的原子性，我们就说α也满

⊖ 实际上，你可能想要允许对各个端口的并发访问。这需要对本章给出的定义做某些扩充。我们避免了这些复杂性，这样我们就可以相当简单地展现这些思想。

足T的原子性。

例 13.1.2　带序列化点的运行

图13-3显示了一个域$V = \mathbb{N}$、初始值$v_0 = 0$的单写者/单读者的读/写对象的某些运行，且这个对象满足读/写寄存器变量类型的原子性。序列化点用星号表示。假设端口1和端口2分别用做写和读。

在图13-3a中，一个返回0的读操作和一个$write(8)$操作重叠，而读的序列化点被放在$write(8)$序列化点的前面，如果操作间隔收缩到它们的序列化点上，那么调用和回应的序列就
402 是$read_2$，0_2，$write(8)_1$，ack_1。这是变量类型的一条轨迹（参见例9.4.4）。

在图13-3b中，与图13-3a中的相比，同一操作间隔被赋予相反顺序的序列化点，那么调用和回应的结果序列就是$write(8)_1$，ack_1，$read_2$，8_2，这又是变量类型的一条轨迹。

图13-3c和图13-3d中的运行都包含一个未完成操作$write(8)$。在每种情况下，一个序列化点被分配给$write(8)$，原因在于它的结果可以通过读操作看出来。对于c），收缩操作间隔的结果是$write(8)_1$，ack_1，$read_2$，8_2。而对于d），其序列为$read_2$，0_2，$write(8)_1$，ack_1，$read_2$，8_2。两者都是变量类型的轨迹（参见例9.4.4）。

在e）中，有无限多的读操作返回0，因此，未完成的$write(8)$不能分配一个序列化点。

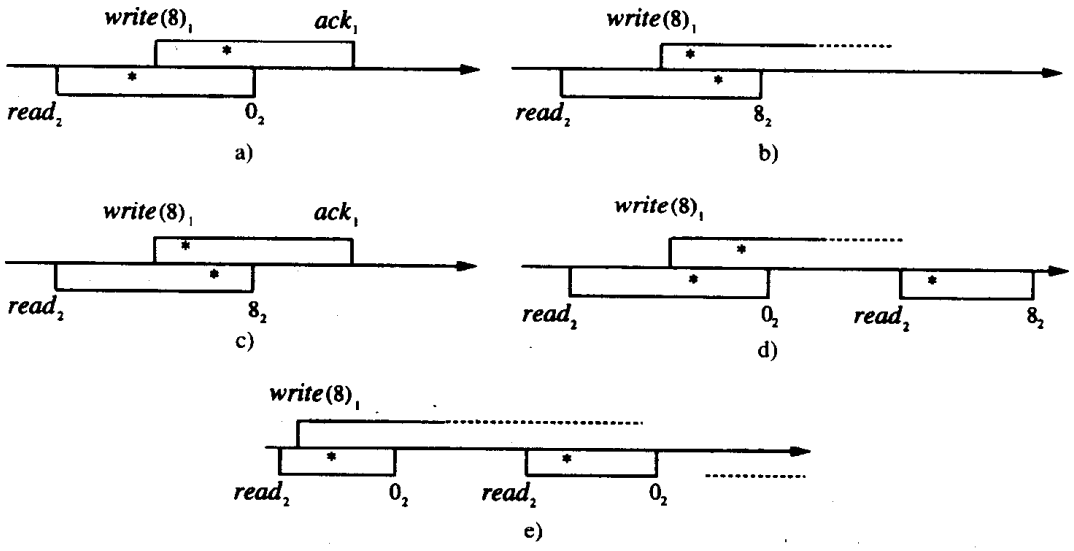


图13-3　满足原子性的单写者/单读者的读/写对象的运行

例13.1.3　无序列化点的运行

图13-4说明了单写者/单读者的读/写对象的一些运行，它们不满足原子性。在图13-4a中，
403 没有办法插入序列化点来解释为何一个读操作会返回8，而其后一个读操作却返回0。在图13-4b中，没有办法解释为何一个写8操作完成后读操作还会返回0。

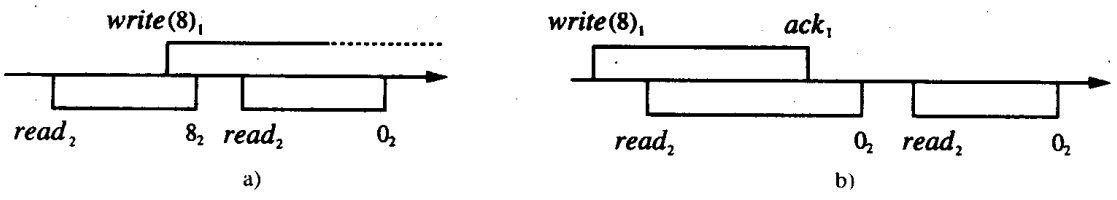


图13-4　单写/单读的读/写对象不满足原子性的运行

现在我们（最终）准备好来定义合成系统$A \times U$的原子性条件。

原子性：令α是一个对任意i都是良构的$A \times U$的（有限或无限）运行，则α满足原子性（如例13.1.2之前的定义）。

我们也可以用轨迹属性来表述原子性条件（轨迹属性的定义见8.5.2节）。即定义轨迹属性P，使得其签名$sig(P)$就是$A \times U$的外部接口，并且它的$traces(P)$集正好就是满足以下两个属性的序列集：

1）对每个i的良构性。

2）对$\mathcal{T}$的原子性。

（为了简单起见，我们把*stop*动作包括在P的签名中，虽然我们在良构性和原子性属性中并没有提到它们）。有意思的是，P是一个安全属性，其定义见8.5.3节。也就是说，$traces(P)$是非空的、前缀封闭的、极限封闭的。这并不明显，因为原子性定义比较复杂，涉及到对序列化点的恰当排列以及操作和回应的选择。

定理13.1 P（上面定义的轨迹属性，表示良构性和原子性的组合）是一个安全属性。

定理13.1的证明用到了König引理，一个关于无穷树的基本组合引理：

404

引理13.2（König 引理） 如果G是一棵无穷树，其上每一个节点只有有穷多个孩子，则G有一条从根开始的无穷路径。

（定理13.1的）证明概要 非空性是显然的，因为$\lambda \in trace(P)$。

对于前缀闭合，假设$\beta \in trace(P)$并令β'是β的一个有限前缀。因为$\beta \in trace(P)$，所以可以选出一个未完成操作集合Φ、一个对Φ中操作的回应集合和一个序列化点集合，一起用来证明β的正确性。我们来看如何为β'进行这些选择。

定义γ为通过插入选出的序列化点来从β取得的序列。令γ'是γ的前缀，且它以β'的最后一个元素结尾。则对于β'中的所有完成操作和某个未完成操作的子集，γ'含有相应的序列化点。选出由β'中未完成操作组成的集合Φ'，其中的未完成操作在γ'中具有序列化点。按照以下方法来为每一个操作$\pi \in \Phi'$选出一个回应：如果在β中π是未完成的，也就是说，如果$\pi \in \Phi$，那么就选择β中已为π选出的同一回应；否则就选择确实在β中出现的回应。不难看出，集合Φ'、选出的回应和γ'中的序列化点共同表明了β'的正确性。这就证明了前缀封闭。

最后，我们证明极限封闭。考虑一个无限序列β，并假设β的所有有限前缀都在$trace(P)$中，我们使用König引理。

为了使用König引理，我们构造一棵树G，它描述了β中序列化点的可能位置。G的每一个节点都以β的一个有限前缀作为标记，其中序列化点针对在β中被调用的某些操作来插入。我们只包括“正确的”标记，即它们要满足以下三个条件：

1）每一个完成的操作正好有一个序列化点，而且序列化点出现在操作的调用和回应之间。

2）每个未完成的操作至多有一个序列化点，而且序列化点出现在操作的调用之后。

3）对操作π的每一个回应恰好就是那个在序列化点上使用给定变量类型$\mathcal{T}$的函数对π计算出来的回应。（以初始值v_0开始，在每一个序列化点上依次使用一次函数，相应的调用作为第一个参数。对π计算出的回应就是当函数应用于序列化点$*_\pi$时得到的回应。）

此外，在G中，

405

1）根节点标记为λ。

2）每一个非根节点的标记是它双亲节点标记的扩展。

3）每一个非根节点的标记以β的一个元素结尾。

4）每一个非根节点的标记比它的双亲节点的标记恰好多含一个β中的元素（可能还有一些更加序列化的点）。

这样，在G中的每一个分支点上，就可以决定在β的两个特定标记之间，以何种顺序插入哪一个序列化点。通过考虑上面的前缀封闭构造，我们可以构造出G，使得β的每一个有限前缀β'都是G中某一节点的标记，并且β'中最后一个标记之前的序列化点都被“正确”赋值。

现在我们在树G上使用König引理。首先，容易看出G中每个节点只有有限多个子节点。这是因为只有那些已经被调用操作的序列化点才可以被插入，而只有有限多个位置可以插入这些序列化点。

其次，我们说G包含从根节点开始的任意长的路径。这是因为无限序列β的每一个有限前缀β'都在$trace(P)$中，这意味着β'有一个对序列化点的恰当的赋值。这个的赋值生成一条长度为$|\beta'|$的G中相应路径。

由于G包含从根节点开始的任意长的路径，因此是无限的。由König引理（引理13.2）可知，G包含一条从根节点开始的无限路径。对整个序列β，路径上的节点标记产生了一个正确的对整个序列β的序列化点（以及未完成操作和回应）选择。 □

我们已经为原子对象定义了安全属性——良构性和原子性——现在来看活性属性。我们考虑的活性属性是与12.1节中针对一致性问题给出的终止性条件相似的终止性条件。最简单的要求是针对无故障运行的，即那些没有*stop*事件发生的运行。

无故障终止　在$A \times U$的任意公平无故障运行中，每个调用都有一个回应。

有了这个活性属性，我们就可以定义“原子对象”。即如果A保证对$\mathcal{T}$的良构性条件和原
406 子性条件以及对所有用户集的无故障终止条件，则我们称A是变量类型$\mathcal{T}$的一个原子对象。

注意，如果我们只考虑无故障的情况，那么我们可以简化原子性条件的陈述，因为根本用不着考虑未完成的操作。我们之所以给出较复杂的原子性条件陈述，是因为我们还要考虑故障。

如同10.2节中的互斥问题一样，我们可以根据轨迹属性P来等价地表示原子对象的整个定义。这次，$sig(P)$包括原子对象的所有外部接口动作，其中除了调用和回应外，还包括停止动作；并且$trace(P)$表达了良构性、原子性和无故障终止。那么，仅当$fairtrace(A \times U) \subseteq trace(P)$对所有用户集都成立时，具有正确接口的自动机$A$就是类型$\mathcal{T}$的一个原子对象。

我们也考虑一些涉及容错性的更强终止条件。

无等待终止　在$A \times U$的任意公平运行中，在无故障端口上的任一调用都有一个回应。

也就是说，没有故障发生的端口对所有调用都提供回应，而不管在其他任何端口上发生的故障。我们扩展这个属性以描述任意数目的故障出现时的终止条件。

f故障终止，$0 \leqslant f \leqslant n$　在$A \times U$的任意公平运行中，其中*stop*事件最多发生在f个端口上，无故障端口上的每一个调用都有一个回应。

无故障终止和无等待终止是f故障终止条件的特殊情况，其中f分别等于0和n。一个进一步的推论允许我们讨论任一特定端口集上的故障。

I故障终止，$I \subseteq \{1, \ldots, n\}$　在$A \times U$的任一公平运行中，其中*stop*事件仅发生在I中的端口上，无故障端口上的每一个调用都有一个回应。

这样，当所有端口集I的规模不超过f时，f故障终止和I故障终止是一样的。如果A对所有的用户集合都保证相应的条件，则我们称A保证无等待终止、保证I故障终止，等等。

我们用一个共享存储器系统的简单例子来结束本节，它是一个原子对象。

例13.1.4 一个读/递增原子对象

这个读/递增变量类型的域为$\mathbb{N}$，初始值为0，操作包括读和递增。 407

令A是一个共享存储器系统，它有n个进程，其中每一个端口i既支持读操作也支持递增操作。A有n个共享读/写寄存器$x(i)$，$1 \leqslant i \leqslant n$，每一个的域为$\mathbb{N}$，初始值为0。共享变量$x(i)$可被进程$i$写，可被所有进程读。

当一个$increment_i$输入发生在端口i上时，进程i只递增它自己的共享变量$x(i)$。通过记住本地状态$x(i)$的值，它只要用一个写操作就可以完成这个调用。当一个$read_i$发生在端口i上时，进程i以任意顺序来逐个读取所有共享变量$x(j)$，并返回它们的和。

不难看出A是一个读/递增原子对象并且保证无等待终止。举例来说，为了证明它的原子性条件，我们考虑$A \times U$的任一运行。令Φ是由对于写操作发生在共享变量上的未完成递增操作组成的集合。对于每个完成的或在Φ中的递增操作π，把序列化点$*_\pi$放置在写操作的位置上。

现在，注意到任一完成的（高层的）读操作π返回一个值v，v不小于读操作被调用时所有$x(i)$的和，也不大于读操作完成时所有$x(i)$的和。因为每一个递增操作仅使和增加1，所以在π的间隔中必定存在一点，其上$x(i)$的和正好等于返回值v。我们就把序列化点$*_\pi$放置在这个点上。这些选择就允许了原子性所需的收缩。

13.1.2 规范无等待原子对象自动机

在本节中，对给定的变量类型$\mathcal{T}$和给定的外部接口，我们给出原子对象自动机C的例子。自动机C保证无等待终止。C是高度非确定的，有时被认为是给定类型和外部接口的“规范无等待原子对象自动机”。它可以用来证明其他自动机是无等待的原子对象。

C自动机（非形式化）：

C拥有类型$\mathcal{T}$的共享变量的一份内部拷贝，变量的初始值为v_0。它还有两个缓冲区，*inv-buffer*用来存放未决的调用，*resp-buffer*用来存放未决的回应，初始时都为空。最后，它跟踪有 408
停止动作发生的端口，并将这些端口加入*stopped*集合中。该集合初始时为空。

当一个调用到达时，C仅将它记录在*inv-buffer*中。在任何时候，C可以把未决的调用从*inv-buffer*中移出，然后在共享变量的内部拷贝上执行要求的操作。完成了要求的操作以后，它把所得回应放入*resp-buffer*中。同样的，在任何时候，C可以把未决的回应从*resp-buffer*中移出，传送给用户。

一个$stop_i$事件仅把i加入到*stopped*中，从而令一个特定的$dummy_i$动作（该动作对系统毫无影响）使能。然而，它不会导致涉及i的其他局部控制动作失效。涉及任一端口i的所有局部控制动作——包括$dummy_i$动作——都被归为一个任务。这意味着一个$stop_i$之后，涉及i的动作可以（但不必要）中止。

C automaton (formal):

Signature:

Input:
- a_i's as in the given external interface
- $stop_i$, $1 \leq i \leq n$

Output:
- b_i's as in the given external interface

Internal:
- $perform(a)_i$, a_i in the external interface, $1 \leq i \leq n$
- $dummy_i$, $1 \leq i \leq n$

States:

val, a value in V, initially v_0

inv-buffer, a set of pairs (i, a), for a_i in the external interface
resp-buffer, a set of pairs (i, b), for b_i in the external interface
stopped $\subseteq \{1, \ldots, n\}$, initially empty

Transitions:

a_i
　Effect:
　　inv-buffer := *inv-buffer* $\cup \{(i, a)\}$

b_i
　Precondition:
　　$(i, b) \in$ *resp-buffer*
　Effect:
　　resp-buffer := *resp-buffer* $- \{(i, b)\}$

409

$perform(a)_i$
　Precondition:
　　$(i, a) \in$ *inv-buffer*
　Effect:
　　inv-buffer := *inv-buffer* $- \{(i, a)\}$
　　$(b, val) := f(a, val)$
　　resp-buffer := *resp-buffer* $\cup \{(i, b)\}$

$stop_i$
　Effect:
　　stopped := *stopped* $\cup \{i\}$

$dummy_i$
　Precondition:
　　$i \in$ *stopped*
　Effect:
　　none

Tasks:
for every i:
　$\{perform(a)_i : a_i$ is an input$\} \cup \{b_i : b_i$ is an output$\} \cup \{dummy_i\}$

定理13.3　C是给定类型和外部接口上的一个原子对象，且（对所有的用户集合）保证无等待终止。

证明概要　良构性是直接的。要证明无等待性，我们考虑$C \times U$的任一公平运行α，并假设在α中端口i上没有发生故障。这样，$dummy_i$动作在α中从来就不是使能的。α的公平性表明，端口i上的每一个调用都会触发一个$perform_i$事件和随后的一个回应。

接下来证明原子性。考虑$C \times U$的任一运行α，令Φ是由对于*perform*在α中发生的未完成操作组成的集合。为每一个操作π指定一个序列化点$*_\pi$，π或者在α中是完成的，或者在Φ中：把$*_\pi$放置在*perform*的那个点上。对每一个$\pi \in \Phi$，选择*perform*返回的回应作为操作的回应。这些选择允许了证明原子性所需的收缩。　□

C可以用来证明其他自动机也是无等待原子对象，如下所述。

定理13.4　假设A是一个具有和C一样的外部接口的I/O自动机。如果对于每个用户自动机合成U，$fairtraces(A \times U) \subseteq fairtraces(C \times U)$成立，那么$A$是一个保证无等待终止的原子对象。

证明概要　这是定理13.3的推理。对于良构性和原子性，我们可以利用这两个条件的组合是安全属性（见定理13.1）的事实，加上每一个有限轨迹可被扩展成一个公平轨迹（见定理8.7）的
410 事实来证明。无等待性条件直接由定义得到。　□

定理13.4的逆定理也成立。即一个无等待原子对象允许的每条公平轨迹实际上都是由C产生的。

定理13.5　假设A是一个拥有和C一样的外部接口的I/O自动机。假设A是一个保证无等待终止的原子对象。则对于每一个用户自动机合成U，都有$fairtraces(A \times U) \subseteq fairtraces(C \times U)$。

证明 该证明留做一道习题。 □

13.1.3 原子对象的合成

在本节中，我们给出一个定理来说明原子对象的合成（采用8.2.1节中定义的普通I/O自动机合成的定义）也是一个原子对象。试回忆一下9.4节尾对兼容变量类型和变量类型合成的定义。

定理13.6 令$\{A_j\}_{j\in J}$是一个兼容变量类型$\{\mathcal{T}_j\}_{j\in J}$上的可数原子对象集合，并且所有的$A_j$都有相同的端口集$\{1, ..., n\}$。则合成$A=\prod_{j\in J}A_j$是一个变量类型$\mathcal{T}=\prod_{j\in J}T_j$和端口$\{1, ..., n\}$上的原子对象。

此外，如果每一个A_j保证I故障终止（对所有的用户集），则A也保证I故障终止。

在原子对象A中，端口i处理任一A_j中端口i上的所有调用和回应。根据合成的定义，每个A_j都作为A的状态中的一部分。来自A_j的调用和回应只涉及A的状态中与A_j有关的那一部分。然而，$stop_i$动作影响状态的所有部分。我们把定理13.6的证明留做一道习题。

13.1.4 原子对象和共享变量

原子对象的定义表明它的轨迹“看起来像”一个对底层共享变量进行顺序访问的轨迹，这有什么好处呢？

从系统构造的观点来看，关于原子对象最重要的事实是它们可以代替共享存储器系统中的共享变量。这允许系统的积木式构造：可以先设计出一个共享存储器系统，然后用给定类型的任意原子对象去代替其中的共享变量。在某种情况下，对于用户来说，所得系统和原来的共享存储器系统“表现相同”。 411

在这一节中，我们描述这种代替技术。我们首先给出一些技术条件，这些条件对需要被替代的原来共享存储器系统的正常工作是必要的。接下来，我们给出代替构造方法。最后我们定义如何才会觉得结果系统和原来系统具有相同的行为，并且证明在给定的条件下，所得系统确实和原来系统具有相同的行为。虽然基本概念相当简单，但是为了令代替工作正常，一些细节需要小心处理。

我们从第9章共享存储器模型中的任意一个算法A开始。我们假设A和用户自动机$U_i, 1\leqslant i\leqslant n$相交互。我们允许$A$中每一进程$i$有任意数目的任务。如同9.6节中的讨论，我们引入$stop_i$动作，并且假设每一$stop_i$事件都使进程$i$的所有任务永久失效。

现在，考查我们上面提到过的技术条件。考虑A与用户自动机U_i的任意组合。我们假设对于每个端口i，存在一个函数$turn_i$，对于合成系统的任意有限运行α，这个函数的输出值是*system*或者*user*。这是用来指出在α之后轮到谁来运行下一步。特别的，如果$turn_i(\alpha)$= *system*，则U_i在α之后的状态中没有使能的输出步；而当$turn_i(\alpha)$= *user*时，A的进程i没有输出步和内部步，也就是说，A在α之后的状态中没有使能的局部控制步。

举例来说，第10章中所有的互斥算法和第11章中所有的资源分配算法都满足这些条件（如果我们加上停止动作的话）。在这些情况中，在α之后U_i处于尝试区或者退出区时，$turn_i(\alpha)$ = *system*，而当U_i处于临界区或者其他区域时，$turn_i(\alpha)$ = *user*。实际上，所要求的条件已经蕴涵在对进程活动的限制中了，这些限制见10.2节尾和11.1.2节尾。

对于在第12章研究过的一致性算法，我们可以定义：当α中含一个$init_i$事件时，$turn_i(\alpha)$ = *system*，否则$turn_i(\alpha)$ = *user*。为了满足这里需要的条件，我们必须加上一个限制，即进程i在一个$init_i$发生之前不能做任何事。只有第12章的*RMWAgreement*算法满足这个条件。

现在我们进行代替。假设对于A中的每一个共享变量x，我们都有一个原子对象自动机B_x，它具有相同的类型和恰当的外部接口。也就是说，B_x有端口1, ..., n，每一个端口对应A中的一个进程。在每一个端口上，算法A中进程i用来与共享变量x进行交互的所有调用和回应都是合

412 法的。和往常一样，它也有$stop_i$输入，每一个端口对应一个$stop$。

然后，用原子对象B_x代替了共享变量的A的变换版本$Trans(A)$被定义为如下自动机。

$Trans(A)$ 自动机：

$Trans(A)$是一个I/O自动机合成，其中一个自动机对应一个进程i，一个自动机对应算法A中的一个共享变量x。针对每个变量x的自动机就是原子对象自动机B_x。针对每一个进程i的自动机就是以下定义的P_i。

P_i的输入包括A在端口i上的输入、每个B_x在端口i上的回应和$stop_i$动作。P_i的输出就是A在端口i上的输出再加上每个B_x在端口i上的调用。

P_i的步直接模仿A中进程i的步，以下除外：当A中的进程访问共享变量x时，P_i代之给B_x发送恰当的调用。之后，它暂停自己的活动，等待B_x对调用的一个回应。当一个回应到达时，P_i像平常一样模仿A中的进程i。对A中进程i的每个任务，P_i也有一个任务与之对应。

如果一个$stop_i$事件发生，则P_i的所有任务都随之失效。

例 13.1.5　A和$Trans(A)$

考虑一个双进程共享存储器系统A，它使用两个读/写共享变量x和y来解决某类一致性问题。我们假设进程1写x、读y，进程2写y、读x。A和U_i之间的接口由形如$init(v)_i$的动作和形如$decide(v)_i$的动作组成，$init(v)_i$是U_i的输出和A的输入，而$decide(v)_i$是A的输出和U_i的输入。此外，$stop_i, i \in \{1, 2\}$也是A的一个输入。这个系统的体系结构如图13-5a所示。

变形后的系统$Trans(A)$的体系结构如图13-5b所示。注意自动机B_x和B_y的外部接口。举例来说，B_x具有输入$write(v)_1$和$read_2$、输出ack_1和v_2⊖。B_x还有输入$stop_1$和$stop_2$，$stop_1$是对P_1的输入，$stop_2$是对P_2的输入。这也意味着$stop_1$除了令P_1的所有任务失效之外，还同时对B_x实现产
413 生作用。

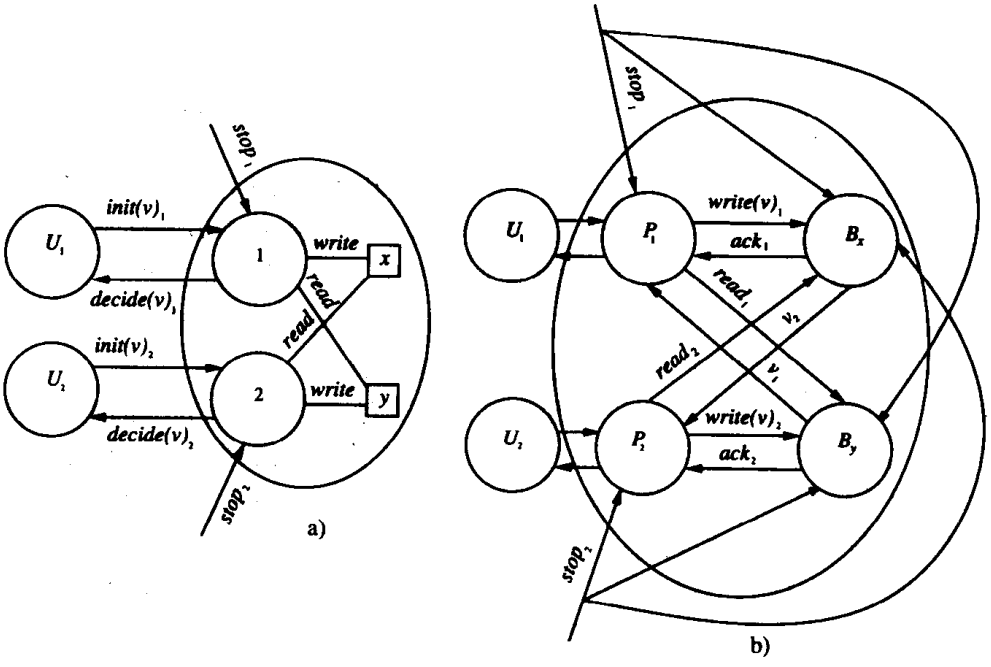


图13-5　将一个共享存储器系统转换为包含原子对象

⊖ 为了消除歧义，这样的请求和回应可以用对象的名字做下标，此处为x和y。在这个例子中，我们避免使用这种细节，因为这儿碰巧没有歧义。

现在我们给出一个定理来描述变形*Trans*所具有的性质。定理13.7首先描述了对*Trans*(*A*)中任一运行α都成立的条件。要使这些条件成立，运行α不一定要是公平的。这些条件表明在用户看来α像是*A*的一个运行α'。此外，同样的*stop*事件在α和α'中发生，尽管我们允许两个运行中的*stop*事件可能发生在不同的位置。

然后定理13.7继续标识出一些条件，在这些条件下，系统*A*的模拟运行α'肯定是一个公平运行。正如你所期望的，其中一个条件就是：α本身就是系统*Trans*(*A*)的一个公平运行。但这还不够——我们还要保证对象自动机B_x不会导致处理停止。因此，我们加入其他两个条件，它们一起保证这不会发生，即α中的所有故障被限制在一个特定的端口集*I*上，而所有的对象自动机B_x可以忍受*I*上的故障（形式化的说，它们保证*I*故障终止）。

定理13.7 假设α是系统*Trans*(*A*) × *U*中任一运行，则存在*A* × *U*的一个运行α'，使得下列条件成立： 414

1）α和α'对*U*是不可区分[⊖]的。

2）对每个*i*，如果一个$stop_i$发生在α'中，则恰好有一个$stop_i$发生在α中。

此外，如果α是一个公平运行，每个有$stop_i$出现在α中的*i*都在*I*中，且每个B_x保证*I*故障终止（对所有的用户集），则α'也是一个公平运行。

证明概要 我们修改α来得到以下的α'。首先，由于每个B_x都是原子对象，我们可以在α中B_x上的每个完成操作π的调用和回应之间插入序列化点$*_\pi$，还可以在B_x上的未完成操作子集Φ中每一个调用之后插入序列化点$*_\pi$。我们还获得对Φ中所有操作的回应。这些序列化点和回应可确保原子性条件中描述的“收缩”属性得到满足。

接下来，我们移动B_x上的每个完成操作π的调用和回应，使得它们相邻且恰好发生在$*_\pi$处。同样的，对Φ中的每一个未完成操作——即被分配了一个序列化点的未完成操作——我们把调用和新取得的回应放置在$*_\pi$处。而对不在Φ中的每一个未完成操作——即未被分配序列化点的未完成操作——我们只简单地移除调用事件。另外还有一个技术问题：如果α中任意一个$stop_i$事件发生在进程*i*的一个调用之后但发生在调用要移到的序列化点之前，则$stop_i$事件也移到这个序列化点上，且正好放在这个调用和回应之后。对所有的共享变量*x*，我们就以这样的方式移动、增加和移除事件。

我们来证明：可以不改变任意一个P_i中事件的顺序（有一个例外：某个B_x对P_i的一个回应可能被移到一个$stop_i$的前面）按照上面构造方法来移动所有的事件。原因在于两个事实。首先，根据构造可知，当P_i在等待对一个调用的回应时，它不执行任何局部控制动作。其次，当P_i在等待回应时，轮到*system*来执行步。这意味着U_i不执行任何输出步，所以P_i也没有收到任何输入。

同样，我们来证明：可以在这个构造中加入那些已经加入的回应并移除那些已经移除的调用，而P_i的行为不受影响。这是因为如果P_i在α中执行了一个未完成操作，那么此操作后它不做任何事情。P_i在发送调用之前停止，或在等待回应之时停止也好，或在收到回应之后停止，都无关紧要。 415

由于我们在移动、增加和移除事件时并未改变重要的东西，我们可以简单地如同在*a*中一样填入进程P_i的状态（一个例外情况是：如果对P_i的回应被移到一个$stop_i$之前，则可能会导致P_i状态与α中的状态不同）。结果得到系统*Trans*(*A*) × *U*的一个新运行α_1。此外，显然α和α_1对于*U*来说是不可区分的，而且α和α_1的相同端口上有相同的*stop*事件。

现在α_1是*Trans*(*A*) × *U*的一个运行，这不是我们正好想要的，我们需要系统*A* × *U*的一个运行。

⊖ 我们使用8.7节给出的不可区分的定义。

但是注意到在α_1中，对象自动机B_x的所有调用和回应都在连续匹配对中出现。我们用对相应共享变量的瞬时访问来代替这些调用和回应对，并因而得到系统$A \times U$的一个运行α'。那么α和α'对于U是不可区分的，而且它们的相同端口上有相同的$stop$事件。这证明了定理的前半部分。

对于定理的后半部分，假设α是$Trans(A) \times U$的一个公平运行，$I \subseteq \{1, ..., n\}$，满足$stop_i$出现在$\alpha$中这一条件的$i$都在$I$中，且每个$B_x$都保证$I$故障终止。那么任一$B_x$接收到的惟一$stop_i$输入必然是针对端口$i \in I$的。这样，由于每个$B_x$都保证$I$故障终止，所以，$B_x$为那些在$\alpha$中没有$stop_i$事件发生的进程$P_i$的每个调用都提供了回应。这个事实，连同对进程$P_i$的公平性假设，足以表明$\alpha'$是$A \times U$的一个公平运行。 □

因此，定理13.7意味着针对共享存储器模型（带有一些简单限制）的任一算法都可改为使用原子对象而不是共享变量的变形算法，而用户无法区分出这两个算法的不同。

我们以推论的形式给出定理13.7的特殊情况，其中B_x都保证无等待终止。在这种情况下，我们可以得出结论：如果α是公平的，那么α'也是公平的。

推论13.8　假设所有的B_x都保证无等待终止。如果α是$Trans(A) \times U$的任一公平运行，则存在$A \times U$的一个公平运行α'使得以下条件成立：

1）α和α'对U是不可区分的。

2）对任意i，如果一个$stop_i$出现在α'中，则恰有一个$stop_i$出现在α中

证明　令$I = \{1, ..., n\}$，直接从定理13.7可得。 □

在A本身就是一个原子对象的特殊情况中，定理13.7表明$Trans(A)$也是一个原子对象。考虑故障时，我们得到如下推论。 416

推论13.9　假设A和所有B_x都是保证I故障终止的原子对象，则$Trans(A)$也是保证I故障终止的原子对象。

证明　首先令α是$Trans(A)$和用户U_i集合的任意一个运行。则由定理13.7得出$A \times U$的一个运行α'使得α和α'对U是不可区分的。由于A是一个原子对象，所以α'满足良构性和原子性属性。由于这两者都是U的外部接口的属性，而α和α'对U是不可区分的，所以α也满足良构性和原子性属性。

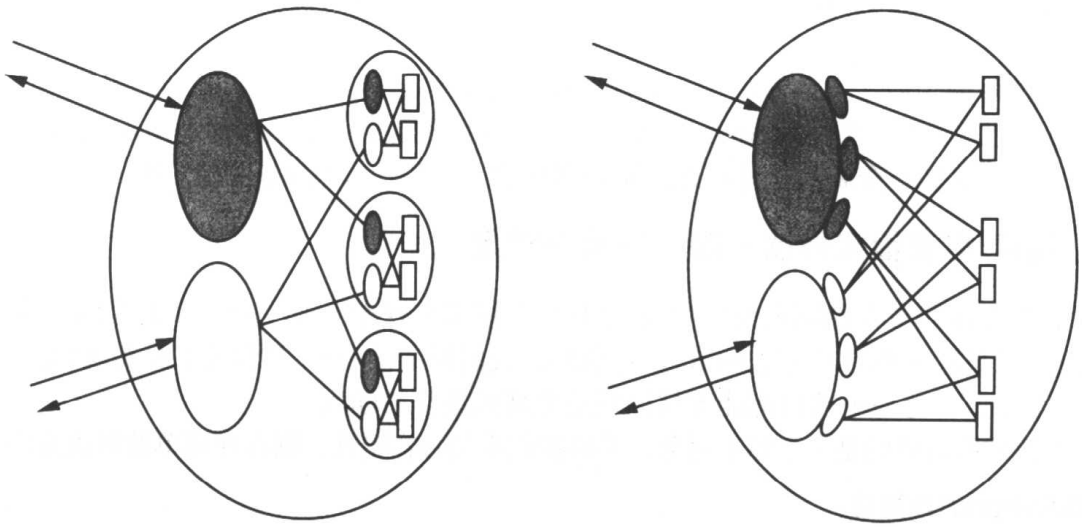
余下考虑I故障终止条件。令α是$Trans(A)$和用户U_i集合的任意一个公平运行，使得每个满足$stop_i$出现在α中这一条件的i都在I中。因为所有的B_x都保证I故障终止，根据定理13.7可得$A \times U$的一个公平运行α'，α和α'对U是不可区分的，且它们在相同的端口集上有相同的$stop$事件。这样，满足$stop_i$出现在α'中这一条件的i都在I中。

现在考虑α中端口i上的任意一个调用，此端口上没有$stop_i$事件发生——即处于一个无故障端口。由于α和α'对U是不可区分的，所以同样的调用出现在α'中。因为A保证I故障终止，所以在α'中有一个相应的回应事件。又由于α和α'对U是不可区分的，这个回应同样出现在α中。这足以证明I故障终止条件。 □

共享存储器系统的分级构造　在每一个原子对象B_x本身就是一个共享存储器系统的特殊情况中，我们来证明$Trans(A)$也可以被看做是一个共享存储器系统。即$Trans(A)$（视为一个共享存储器系统）的每一个进程i都是$Trans(A)$的进程P_i和所有共享存储器系统B_x中序号为i的进程的合成。这个合成并不恰好就是一个I/O自动机合成，因为B_x中的进程不是I/O自动机。不过，这个合成很容易描述：$Trans(A)$中进程i的状态集恰好是P_i的状态集和所有B_x中序号为i的进程的

状态集的笛卡儿乘积，开始状态的情况也同样如此。与*Trans*(*A*)中进程*i*相关的动作正好就是所有组件进程*i*的动作，任务的情况也类似。

见图13-6。a)描述了*Trans*(*A*)，包括插入到*A*的所有共享变量*x*中的共享存储器系统B_x。（为了简单起见，我们没有画出*stop*输入的箭头。）所有的阴影进程都与端口1相关。b)显示了和a)相同的系统，其中将要合成的进程被归组。这样a)中所有阴影进程就合并成为b)中的一个简单进程。 417



a) b)

图13-6 共享存储器系统的分级构造

由*Trans*(*A*)的定义可知，a)的系统中一个$stop_i$事件的效果是立即结束所有与端口*i*相关的进程的任务——P_i的任务和B_x的所有进程*i*的任务。也可以说$stop_i$停止了b)的系统中的合成进程*i*的所有任务，这正好是当把系统看成是一个共享存储器系统时，$stop_i$要做的。

原子对象的分级构造 最后我们考虑非常特殊的情况：共享存储器系统*A*是一个保证*I*故障终止的原子对象，每一个原子对象B_x都是保证*I*故障终止的共享存储器系统。则推论13.9和前面的段落表明*Trans*(*A*)是一个保证*I*故障终止的原子对象，而且它也是一个共享存储器系统。以上观察表明，共享存储器系统模型中原子对象实现的两个连续层可以压缩为一层。 418

13.1.5 显示原子性的一个充分条件

在给出特定的原子对象构造方法之前，我们给出一个充分条件以表明一个共享存储器系统保证原子性条件。这个引理可以使我们避免对（在证明对象是原子的过程中的）许多未完成操作进行明确推理。

对于这个引理，我们假设*A*是一个共享存储器系统，它具有针对变量类型$\mathcal{T}$的原子对象的合适外部接口。我们还假设U_i，$1 \leqslant i \leqslant n$是对*A*的任一用户集；通常，$U = \prod U_i$。

引理13.10 假设合成系统$A \times U$保证良构性和无故障终止，且$A \times U$的任意不包含未完成操作的（有限或者无限）运行α都满足原子性属性，则$A \times U$的每一个运行（包括那些具有未完成操作的运行）也都保证原子性属性。

证明 令α为$A \times U$的任意有限或无限运行，它可能包含未完成操作。我们必须证明α满足原子性，即$\alpha|\ ext(U)$满足原子性属性。

如果α是有限的，则在一个共享存储器系统中对*stop*事件的处理表明，存在一个有限的无故障的运行α_1，它可以通过从α中移除*stop*事件来得到（并可能改动一些和其上有*stop*事件发生的端口上的输入有关的状态变化），这样，$\alpha_1|ext(U) = \alpha|ext(U)$。由I/O自动机的基本属性（特别是定理8.7）可知，α_1可被扩展成$A \times U$中的一个公平无故障运行α_2。由于A保证无故障终止，所以α_2中的每一个操作都是完成的。由假设可知，α_2满足原子性，即$\alpha_2|ext(U)$满足原子性。但是$\alpha_1|ext(U)$是$\alpha_2|ext(U)$的一个前缀。所以根据定理13.1，原子性和良构性都是安全属性，并因此保证前缀闭合，故此$\alpha_1|ext(U)$也满足原子性。因为$\alpha_1|ext\ (U) = \alpha|ext(U)$，所以$\alpha|ext(U)$满足原子性，得证。

另一方面，假设α是无限的。由我们刚才证明可知，α的任意一个有限前缀α_1具有以下性质：$\alpha_1|ext(U)$满足原子性。但是$\alpha|ext(U)$是形如$\alpha_1|ext(U)$的序列的极限。因此，由定理13.1可知，原
419 子性和良构性都是安全属性，并因此保证极限闭合，所以$\alpha|ext(U)$满足原子性，得证。 □

13.2 用读/写变量实现读－改－写原子对象

我们考虑在共享存储器模型中用读/写共享变量来实现读－改－写原子对象时存在的问题。（读－改－写变量类型的定义见9.4节。）特别的，我们固定任一n，并假设正要实现的读－改－写对象有n个端口，每一个端口都能支持以任意更新函数作为输入。

如果我们所需要的是一个原子对象，并且我们不关心容错性，则存在简单的解决方法。例如，

***RMWfromRW*算法：**

将与正要实现的对象相对应的读－改－写变量的最近值保存在一个读/写共享变量x中。当进程希望在原子对象上执行操作时，使用一遍不同于x的读/写共享变量，进程们执行一个锁定权互斥算法（例如，10.5.2节的*PetersonNP*算法）的尝试部分。当一个进程i进入这个互斥算法的临界区时，它取得对x的独占性访问。然后，进程i用一个读步后面紧跟一个独立的写步来执行它的读－改－写操作。完成这些步之后，进程i执行这个互斥算法的退出部分。

然而，这个算法不是容错的：进程可能在临界区中发生故障，因此要防止其他任意进程访问这个模拟的读－改－写变量。实际上，这个限制不是偶然的。我们将给出一个不可能性结果，即使只有一个故障发生时这个结果也成立。

定理13.11　不存在使用读/写共享变量来实现读-改-写原子对象并保证1故障终止的共享存储器系统。

证明　采用反证法。假设存在一个这样的系统，称为B。令A是12.3节给出的读－改－写共享变量模型中的*RMWAgreement*算法，它满足一致性。由定理12.9可知，A保证无等待终止并因此保证1故障终止（见12.1节中一致性算法的定义）。我们现在对A应用13.1.4节的变换，用B来代替
420 A的共享读－改－写变量。用*Trans*(A)来代表所得的系统。

声明13.12　*Trans*(A)解决了第12章中的一致性问题并保证1故障终止。

证明　这个证明同推论13.9的证明相似。首先，令α是*Trans*(A)和用户U_i组成的集合的任一运行。则根据定理13.7可得到$A \times U$的一个运行α'，使得α和α'对U是不可区分的。由于A解决了一致性问题，所有α'满足良构性、一致性和有效性。又因为α和α'对U是不可区分的，所以α也满足良构性、一致性和有效性。

接下来考虑1故障终止条件。令α是*Trans*(A)和用户U_i组成的集合的任一运行，其中*init*事件发生在所有端口上，*stop*事件至多发生在一个端口上。由于B保证1故障终止，由定理13.7可得

到$A \times U$的一个公平运行α'，α'和α对U是不可区分的，并且*stop*事件发生在相同的端口集上。这样，α'中的*init*事件发生在所有的端口上，而α'中的*stop*事件至多发生在一个端口上。

现在考虑在α中的任意端口i上没有$stop_i$事件。由于α和α'的*stop*事件发生在相同的端口上，所以α'中也没有$stop_i$事件。因为A保证1故障终止，所以α'中存在一个$decide_i$事件。又由于α和α'对U是不可区分的，这个$decide_i$也出现在α中。这足以证明1故障终止。 □

然而，由13.1节末的段落可知，*Trans*(*A*)本身就是读/写共享存储器模型中的一个共享存储器系统。这样，*Trans*(*A*)就和定理12.8相矛盾，所以，在读/写共享存储器模型中不可能既满足一致性又保证1故障终止。

13.3 共享存储器的原子快照

在这一章的剩余部分，我们考虑使用其他类型的原子对象（或者等价地，使用共享变量）来实现特定类型的原子对象。这节将主要讨论快照原子对象，下节主要讨论读/写原子对象。

在读/写共享存储器模型中，如果一个进程能够得到共享存储器整个状态的瞬时快照，那么将是很有用的。当然，读/写模型并不直接提供这种能力——它只允许对单个共享变量的读。

在这一节，我们考虑这种快照的实现。我们把这个问题描述为使用读/写共享存储器模型来实现一个名为快照原子对象的原子对象特定类型。快照原子对象的变量类型的域V是由一个 421
更基本的域W上的某一确定长度的向量组成的集合。操作有两种：对单个向量部分的写，我们称之为*更新操作*；对整个向量的读，我们称之为*快照操作*。通过让进程把整个共享存储器看作以这些强大操作来访问的向量，快照原子对象可以简化对一个读/写系统的编程。

我们从问题描述开始，然后给出一个使用无界大小读/写共享变量的简单解法。然后说明如何改动这个构造以使它可以针对有界大小共享变量工作。13.4.5节包含了一个用快照原子对象实现读/写原子对象的应用。

13.3.1 问题

我们首先定义快照原子对象对应的变量类型$\mathcal{T}$；我们称这种类型为*快照变量类型*。

定义从基本域W开始，初始值为w_0。$\mathcal{T}$的域V是由W中元素的向量组成的集合，其中W的固定长度为m。初始值v_0是一个其中所有部分的值都为w_0的向量。存在形如*update*(*i*, *w*)的调用，其中$1 \leq i \leq m$，$w \in W$，与之相应的回应为*ack*；还有调用*snap*，与之相应的回应为$v \in V$。一个*update*(*i*, *w*)调用把当前向量的第i个分量的值赋为w，该调用还触发一个*ack*回应。一个*snap*调用不引起向量的任何变化，但会触发一个包含整个向量当前值的回应。

接下来，定义我们将要考虑的外部接口。我们假设恰好存在$n = m + p$个端口，其中m是向量的固定长度，p是某一任意正整数。前面的m个端口是*update*端口，剩下的p个端口是*snap*端口。在每一个端口i,$1 \leq i \leq m$上，我们只允许形如*update*(*i*, *w*)的调用——即只有对向量的第i个分量进行更新时才在端口i上处理。我们有时候把冗余标记$update(i, w)_i$（它表示一个发生在端口i的*update*(*i*, *w*)调用）简写为$update(w)_i$。在每一个端口i, $m+1 \leq i \leq n$上，我们只允许*snap*调用。见图13-7。

注意，我们现在考虑的只是普遍问题的一种特殊情况，其中对向量中每个分量的更新顺序地到达一个指定的端口。可以考虑一个更为通用的情况，其中许多端口都允许对同一向量分量进行更新。当然，我们也可以考虑这种情况，其中*update*和*snap*操作可以发生在同一个端口上。 422

我们考虑使用一个有n个进程（每个端口对应一个进程）的共享存储器系统来实现与这个变量类型和外部接口对应的原子对象。我们假设所有的共享变量都是1写者/n读者的读/写共享变量。我们描述的实现保证无等待终止。

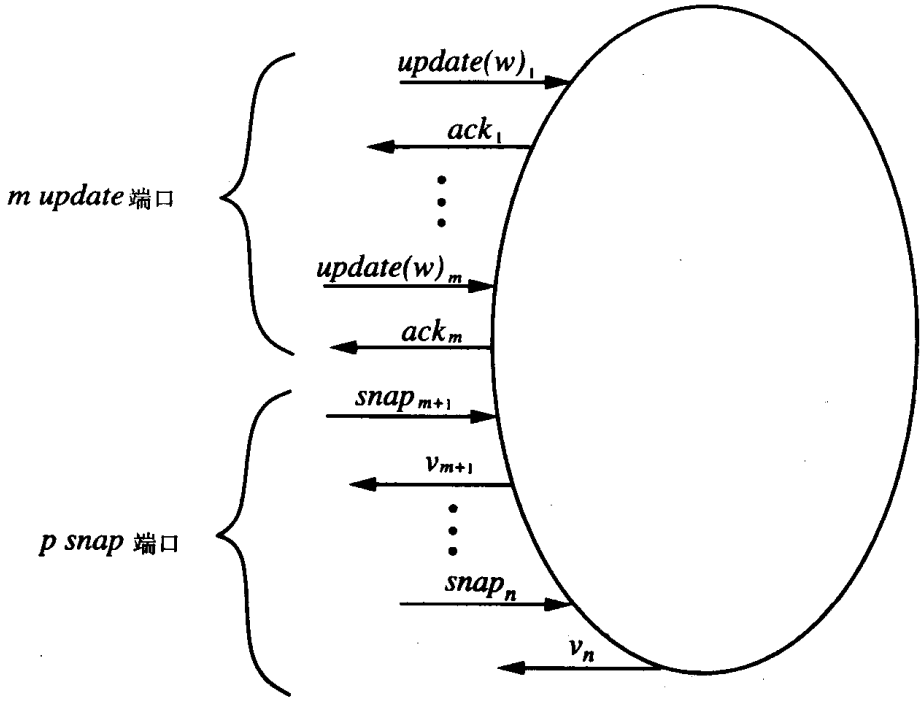


图13-7　一个快照原子对象的外部接口（*stop*动作没有画出）

13.3.2　带无界变量的一个算法

*UnboundedSnapshot*算法用到了m个1写者/n读者的读/写共享变量$x(i)$,$1 \leqslant i \leqslant m$。每一个变量$x(i)$可被进程$i$（与端口$i$相连的进程，这个端口用于*update*$(i, w)$操作）写和被所有进程读。体系结构见图13-8。每个变量$x(i)$都拥有一些值，每个值都由W的一个元素和算法需要的某些附加值组成。其中一个附加值就是无界整数“标签”。

423

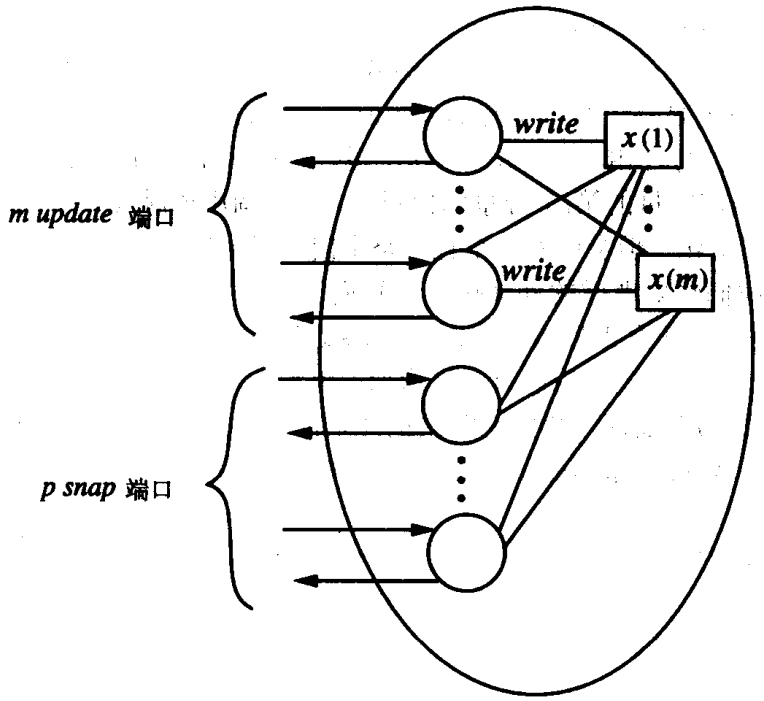


图13-8　*UnboundedSnapshot*算法的结构

在*UnboundedSnapshot*算法中，每个进程*i*都把从$update_i$调用中得到的值写入共享变量$x(i)$中。一个执行*snap*操作的进程必须以某种方式从所有共享变量取得一致值，也就是说，出现的值必须在某一时刻共存在存储器中。实现方法基于以下的两点观察。

观察1：假设只要进程*i*执行一个$update(w)_i$操作，它就不仅把值*w*写入$x(i)$，还写入一个惟一地标识这个*update*操作的“标签”。然后，如果尝试执行一个*snap*操作的进程*j*把所有共享变量读了两遍，第二遍读在第一遍读结束后开始，如果它发现每个变量$x(i)$中的标签在第一遍读和第二遍读中都是相同的，则两遍读中返回的共同向量实际上就是在整个*snap*操作期间的某一点上出现在共享存储器中的那个向量。特别的，这个向量就是第一遍读完成之后，第二遍读开始之前任一点上的值向量。

观察1让我们想到如下简单算法。每个执行$update(w)$操作的进程*i*都把*w*和一个惟一本地标 424
签写入$x(i)$，其中，第一次更新*i*时标签为1，随后每次连续更新*i*时标签加一。

每一个执行*snap*的进程*j*反复执行一遍读操作，对每个共享变量都读一次，直到连续的两遍读是“一致的”，即它们返回对于每个$x(i)$的相同标签。此时，第二遍读返回的值向量（它和第一遍读返回的值向量必定相同）作为对*snap*操作的回应返回。

显而易见，只要这个简单算法完成一个操作，回应总是“正确的”，也就是说，它满足良构性和原子性条件。然而，它甚至不能保证无故障终止：当*snap*是活动的时候，如果新的*update*操作一直被调用，那么即使没有进程故障发生，*snap*也有可能永远不会返回。一种摆脱这种困难的方法由以下提供。

观察2：如果进程*j*为执行*snap*操作而反复执行读操作时，它看到了同一变量$x(i)$的四个不同的标签——为tag_1，tag_2，tag_3，tag_4——那么，它知道某个$update_i$操作完全包含在当前*snap*的间隙中。特别的，写tag_3的$update_i$操作一定完全包含在当前的*snap*中。

为了说明为什么这样，我们首先证明写tag_3的*update*一定在*snap*操作开始之后开始。这是因为它在写tag_2的*update*操作完成之后开始，而写tag_2的*update*的完成一定在*snap*的间隙开始之后发生（因为*snap*看到了tag_1）。

然后，我们证明写tag_3的*update*一定在*snap*完成之前完成。这是因为它在写tag_4的*update*开始之前完成，而*snap*看到了tag_4。

观察1、2使人想到*UnboundedSnapshot*算法。它扩展上面的简单算法，使得一个*update*进程*i*在写$x(i)$之前，它首先运行自己的嵌入*snap*子例程，这个子例程与*snap*操作相似。然后，当它把值和标签写入$x(i)$时，它也把其嵌入*snap*的结果放入$x(i)$。在多次的反复尝试后，如果*snap*不能发现具有相同标签的两遍读时，它可以用一个嵌入*snap*的结果作为默认的快照值。一个更详细的描述如下。在这个描述中，每一个共享变量都是一个具有多个域的记录，我们用点标记表示这些域。

***UnboundedSnapshot*算法：**

每一个共享变量$x(i), 1 \leqslant i \leqslant m$可被进程*i*写，可被所有进程读。它包括下列域： 425

$val \in W$，初值为w_0

$tag \in \mathbb{N}$，初值为0

view，*w*中元素组成的向量，下标为{1，…，*m*}，元素初值全为w_0

当一个$snap_j$输入发生在端口*j*，$m+1 \leqslant i \leqslant n$上时，进程*j*的行为如下：它反复执行一遍遍读，其中每一遍包括*m*个读，这*m*个读以任何顺序将每个共享变量$x(i)$读一遍，其中$1 \leqslant i \leqslant m$。直到下列之一的情况发生：

1）对于每个*i*，两遍读返回相同的$x(i).tag$。

在这种情况下，*snap*返回第二遍读返回的向量值$x(i).val$，$1 \leqslant i \leqslant m$。（它与第一遍读返回的向量值是相同的。）

2）对于某些i，已经看到四个不同的$x(i).tag$值。

在这种情况下，*snap*返回和与第三个$x(i).tag$值相关的$x(i).view$中的向量值。

当一个$update(w)_i$输入发生时，进程i的操作如下。首先要执行一个嵌入*snap*，其工作与一个*snap*操作类似，只不过它决定出来的向量被进程i在本地记录下来，而不是返回给用户。其次，进程i对$x(i)$执行一个写操作，设置$x(i)$的三个域如下：

1）$x(i).val := w$。

2）$x(i).tag$设置为i上的最小未用标签。

3）$x(i).view$设置为嵌入*snap*返回的向量。

最后进程i输出ack_i。

定理13.13　*UnboundedSnapshot*算法是一个保证无等待终止的快照原子对象。

证明　良构性条件是显然的。无等待终止也是易见的：关键在于每个*snap*和嵌入*snap*都必定在最多$3m+1$遍读操作之后终止。这是因为$3m+1$遍读之后，要么存在两个没有变化的连续遍，要么存在某个至少具有四个不同标签的变量$x(i)$。无论如何，操作都将终止。

426　接下来证明原子性条件，给定*UnboundedSnapshot*算法和用户系统的任一运行α。根据引理13.10，我们可以不失一般性地假设α不包含未完成操作，我们来描述如何为所有操作插入序列化点。

我们为每个*update*操作在其写操作发生的位置插入序列化点。为*snap*操作插入序列化点有点复杂。为了描述这种插入，除了给*snap*操作分配序列化点之外，给嵌入*snap*操作分配序列化点也是很有用的。

首先，考虑任意通过找到两遍一致读而终止的*snap*或嵌入*snap*。对每一个这样的操作，我们把序列化点插入到第一遍读结束之后和第二遍读开始之前的任意位置。

其次，考虑那些通过在同一变量中发现四个不同标签而终止的*snap*或嵌入*snap*操作。我们按照它们回应事件的顺序来逐个为这些操作插入序列化点。对每一个这样的操作π，注意到它返回的向量是一个嵌入*sanp*ϕ的结果，其中ϕ的时间间隔完全包含在操作π的时间间隔内。注意到这个操作ϕ已经被分配了一个序列化点，因为它比π完成得更早。我们把对π的序列化点插入到ϕ的序列化点处。

容易看出所有序列化点都在要求的时间间隔内。对于*update*操作和通过找到两遍一致读而终止的*snap*和嵌入*snap*操作来说，这是显然的。对通过在同一变量中发现四个不同标签而终止的*snap*和嵌入*snap*操作来说，可以通过对α中这种操作的回应事件数目进行归纳来证明。

余下要证明的是将操作间隔收缩到各自的序列化点上的结果是底层快照变量类型的一条轨迹。为此，首先注意到α的任一有限前缀α'之后，存在惟一一个V中的向量，它是由α'中的写事件得到的。称之为α'之后的正确向量（*correct vector*）。只要证明在α的前缀（直到序列化点为止）之后，每一个*snap*操作都返回正确向量就足够了。更进一步，我们证明每个*snap*和嵌入*snap*操作对其序列化点都返回正确向量。

对于通过找到两遍一致读而终止的操作，这是显然的。对于其他*snap*和嵌入*snap*操作，可以通过α中这种操作的回应事件的数目进行归纳来证明。　□

复杂性分析　这个*UnboundedSnapshot*算法使用了m个共享变量，每一个变量都在一个无界值集
427　中取值。虽然底层域W是有限的，但因为*tag*是无界的，所以变量仍然是无界的。对于时间复杂

度，一个执行*snap*的无故障进程最多执行$3m+1$遍读，或至多访问$(3m+1)m$个共享变量，总时间为$O(m^2\ell)$，其中ℓ是进程步时间的一个上界。一个执行*update*的无故障进程也访问了$O(m^2)$个共享变量，总时间为$O(m^2\ell)$；这是因为嵌入*snap*操作的存在。

13.3.3 带有界变量的一个算法*

*UnboundedSnapshot*算法的主要问题是用无界大小的共享变量来存储无界标签。在这小节中，我们概述一种称为*BoundedSnapshot*的改进算法，它用有界数据代替了无界标签。为了有效地获得这种改进，*BoundedSnapshot*算法使用了一些比简单标签复杂得多的机制。

注意到在*UnboundedSnapshot*算法中，无界标签只是用来允许进程执行*snap*和嵌入*snap*操作以查明新的*update*操作在何时发生的。然而这个信息可以用一个不如标签强大的机制来传达，特别的，可以使用握手位（*handshake bit*）和开关位（*toggle bit*）的组合。

握手位的工作方式如下。和*UnboundedSnapshot*算法一样，存在$n+m$个变量：变量$x(i)$，$1\leqslant i\leqslant m$，另有新变量$y(j)$，$1\leqslant j\leqslant n$。和前面一样，每一个变量$x(i)$可被*update*进程i写而被所有进程读。每一个变量$y(j)$，$1\leqslant j\leqslant m$可被*update*进程j写（特别的，被*update*进程j的嵌入*snap*部分写）而被所有的*update*进程读；每一个变量$y(j)$，$m+1\leqslant j\leqslant n$可被*snap*进程$j$写而被所有*update*进程读。注意，和*UnboundedSnapshot*算法不一样，*BoundedSnapshot*中*snap*和嵌入*snap*操作的执行涉及对共享存储器的写。

对于每一个*update*进程i，$1\leqslant i\leqslant m$，存在n对握手位。每个进程j一对。(i,j)这一对允许进程i告诉进程j它执行的新*update*，也允许进程j确认自己看到了这条信息。特别的，$x(i)$包含了一个n位向量*comm*，其中变量$x(i)$中的分量$comm(j)$——我们标记为$x(i).comm(j)$——被进程i用来向进程j传达它的新*update*信息。$y(j)$包含一个m位向量*ack*，其中变量$y(j)$中的元素$ack(i)$——标记为$y(j).ack(i)$——被进程j用来向进程i确认它已经看到了进程i的新*update*这一信息。因此，(i,j)对的两个握手位就是$x(i).comm(j)$和$y(j).ack(i)$。 428

这些握手位的使用方式大致如下。当一个进程i运行一个*update*(w)时，它首先读取所有的握手位$y(j).ack(i)$。然后对$x(i)$进行写；在此期间，它写入值w和嵌入*snap*回应的*view*，这和*UnboundedSnapshot*中的做法是一样的，另外再把握手位写入*comm*。特别的，对每个j，把$comm(j)$位设置为一个与操作开始时读到的$y(j).ack(i)$不同的值。

执行*snap*和嵌入*snap*操作的进程j反复尝试执行两遍读，寻找两遍读之间没有变化发生的情况。但是这次，变化是通过握手位而不是整数标签来发现的。特别的，在每次尝试发现两遍一致读之前，进程j首先读取所有握手位$x(i).comm(j)$并把每一个握手位$y(j).ack(i)$设置为与刚才读到的$x(i).comm(j)$相同的值。（这样的话，*update*操作试图把握手位设置为不同的，而*snap*和嵌入*snap*试图把它们设置为相同的。）进程j寻找两遍读之间握手位$comm(j)$上的变化；如果它在$2m+1$次独立尝试中发现了这种变化，则它知道它看到了同一进程j执行的四个独立*update*操作的结果。并且可以采纳第三个操作产生的向量*view*。

目前描述的握手协议是简单的，感觉上是“正确的”：每当一个执行*snap*和嵌入*snap*的进程发现一个变化时，就会发生一个新的*update*。然而，握手协议并不足以发现每一个*update*——有可能进程i的两个连续*update*对其他进程j来说是不可区分的。例如，考虑下面的情况。

例 13.3.1 握手位的不足

假设在一个运行中的某一点中，$x(i).comm(j)=0$，$y(j).ack(i)=1$，也就是说，握手位用来告诉j“i的*update*是不相同的”这一信息。然而下列事件可能以指定的顺序发生。（涉及进程i和j的动作出现在不同的列中。） 429

$update(w_1)_i$
i reads $y(j).ack(i) = 1$
i writes w_1 and sets $x(i).comm(j) := 0$
ack_i
$update(w_2)_i$
i reads $y(j).ack(i) = 1$
　$snap_j$
　j reads $x(i).comm(j) = 0$
　j sets $y(j).ack(i) := 0$
　j reads $x(i).comm(j) = 0$
i writes w_2 and sets $x(i).comm(j) := 0$
ack_i
　j reads $x(i).comm(j) = 0$ and decides that no updates have occurred since its previous *read* of $x(i)$

在这个事件序列中，进程j对$x(i).comm(j)$执行三次读。第一次只是一个初步测试，第二次和第三次是尝试发现两遍一致读的一部分。这里进程j判定在第二次和第三次读之间没有*update*发生。这是错误的。

为了解决这个问题，我们用第二个机制来扩充握手协议：每个$x(i)$包含一个附加的开关位，在进程i的每个写步中，它被进程i切换。这保证了每个*update*都改变共享变量$x(i)$的值。更详细的协议工作方式如下：

BoundedSnapshot算法：

对每个共享变量$x(i)$，$1 \leqslant i \leqslant m$，可被进程$i$写和被所有进程读。它包含下面的域：

$val \in W$，初值为w_0

$comm$，是{0,1}上的一个向量，索引为$\{1, ..., n\}$，向量元素的初值全部为0

$toggle \in \{0,1\}$，初值为0

$view$，W上的一个向量，索引为$\{1, ..., m\}$，向量元素的初值全部为w_0

同样，每一个变量$y(j)$，可被进程j写和被进程i读，其中：$1 \leqslant j \leqslant n$，$1 \leqslant i \leqslant m$。它包含下列域：

ack，{0,1}上的一个向量，索引为$\{1, ..., m\}$，向量元素的初值全部为0

当一个$snap_j$输入发生在端口j上，其中$m+1 \leqslant j \leqslant n$，进程$j$操作如下。它反复尝试去获取两遍看起来“一致”的读。特别地，在每一个尝试中，对所有i，$1 \leqslant i \leqslant m$，进程$j$首先以任意顺序读
430 取所有的相关握手位$x(i).comm(j)$。然后，对于每一个i，进程j把$y(j).ack(i)$设置为与刚才读到的$x(i).comm(j)$相同的值。它以一个简单的写步完成这些工作。然后进程j执行两遍完整的读，第一遍读在第二遍读开始之前结束。如果对于每个i，两次读到的$x(i)$中$x(i).comm(j)$和$x(i).toggle$是相同的，此外，如果这个$comm(j)$共同值和进程j在开始时读取到的值是一样的，那么*snap*的返回值为最后一遍读获得的向量$x(i).val$。否则，进程j记录哪个变量$x(i)$发生了变化。

如果进程j曾在三次独立的尝试中记录到同一变量$x(i)$发生了变化，那么考虑这三次尝试中的第二次。$snap_j$操作返回那次尝试中最后一个读获得的$x(i).view$中的向量值。（这个向量肯定是在一个*update*操作期间写入的，而这个*update*的时间间隔完全包含在给定的$snap_j$的时间间隔中。）

当一个$update(w)_i$输入发生在端口i（其中：$1 \leqslant i \leqslant m$）上时，进程$i$操作如下。首先，它读取所有的相关握手位$y(j).ack(i)$，$1 \leqslant j \leqslant n$。其次，它执行一个嵌入*snap*，除了决定出的向量不返回给用户之外，这个嵌入*snap*和一个*snap*是相同的。第三，进程i对$x(i)$执行单写，设置$x(i)$的四个域如下：

1）$x(j).val := w$。

2）对于每个j，$x(i).comm(j)$被设置为与起始读$y(j)$时取得的$y(j).ack(i)$不同的值。

3）$x(i).toggle$被设置为与它的前一个值不同的值。

4）$x(i).view$被设置为嵌入$snap$返回的向量。

最后，进程i输出ack_i。

定理13.14 *BoundedSnapshot*算法是一个保证无等待终止的快照原子对象。

证明概要 与定理13.13对*UnboundedSnapshot*算法中的证明一样，良构性和无等待性是易见的。剩下的工作是证明原子性条件。这个证明同*UnboundedSnapshot*算法中原子性的证明类似。

我们再次给定运行α，并（由引理13.10可知）不失一般性地假设α中不包含未完成操作。序列化点的插入方法同*UnboundedSnapshot*算法中的完全一样。例如，对通过找到两遍一致读而终止的$snap$和嵌入$snap$操作来说，我们在第一遍读结束到第二遍读开始之间选择任意一点。 431
与前面一样，容易看出序列化点处于所需的时间间隔中。余下要证明的是将操作间隔收缩到各自的序列化点上的结果是底层快照变量类型的一条轨迹。与前面一样，也很容易表明在α的前缀（直到序列化点为止）之后，每一个完成的$snap$和嵌入$snap$操作都返回正确向量。

这次，对于通过找到两遍一致读而终止的$snap$和嵌入$snap$操作来说，这个性质是不易证明的。为此，证明下面的声明就足够了。

声明13.15 如果一个$snap$和内嵌$snap$通过找到两遍一致读而终止，则以下对所有i都为真：在对$x(i)$的第一次读和第二次读之间，没有进程i的写事件发生。

证明 采用反证法，使用一个详细一点的操作式证明。假设端口j上的一个$snap$通过找到两遍一致读而终止，然而进程i的写事件发生在第一次对$x(i)$的读π_1和第二次对$x(i)$的读π_2之间。（对嵌入$snap$操作的证明是一样的。）令ϕ是最后一次这样的写，亦即发生在π_2前面的对$x(i)$的最后一次写。

由两遍读为一致的事实可知，在π_1和π_2中读取的$x(i).comm(j)$的值是相等的，而且这个值和π_1之前最后写入$y(j).ack(i)$的值是相同的（π_1是进程j成功找到一致读的一部分）。将这个值记为b，最后的写事件记为π_0。同样地由一致性可知，从π_1和π_2读到的$x(i).toggle$的值是相等的。把这个值记为t。

因为ϕ是π_2之前对$x(i)$的最后一个写操作，所以它一定会设置$x(i).comm(j):=b$和$x(i).toggle:=t$。包含ϕ的*update*操作必定包含一个更早的对$y(j)$的读事件ψ。由*update*操作的工作方式可知，ψ读到的$y(j).ack(i)$的值一定是$\bar{b}$。（我们使用横杆来表示二进制制补。）这意味着ψ一定在π_0之前。

那么，各个读和写事件的顺序必然如下。（与前面一样，涉及到两个进程i和j的动作显示在不同的栏中。）

ψ: i的一个读看到$y(j).ack(i) = \bar{b}$。

π_0: j的一个写设置$y(j).ack(i) := b$。

π_1: j的一个读看到$x(i).comm(j) = b$和$x(i).toggle = t$。

ϕ: i的一个写设置$x(i).comm(j) := b$和$x(i).toggle := t$。

π_2: j的一个读看到$x(i).comm(j) = b$和$x(i).toggle = t$。 432

但是注意到，与写事件ϕ一样，读事件ψ也是*update*操作的一部分。这意味着两个读事件π_1和π_2必然会返回进程i的两个连续写的结果。然而，π_1和π_2返回的开关位是相同的，这和开关位的工作方式矛盾。 □

这就证明了声明13.15，它表明每一个通过找到两遍一致读而终止的$snap$和嵌入$snap$操作，在α的前缀（直到它的序列化点为止）之后都返回正确向量。对于其他的$snap$和嵌入$snap$操作来说，如同*UnboundedSnapshot*算法一样，原子性的证明是通过对α中这种操作的回应事件数目进行归纳而得的。

复杂性分析　*BoundedSnapshot*算法使用了$m+n$个共享变量。每一个变量$x(i)$可取$|W|^{m+1}2^{n+1}$种值，而变量$y(j)$可取2^m种值。对于时间复杂度，一个执行*snap*的无故障进程最多执行$2m+1$次尝试以找到两遍一致读。对于每一次尝试，最多有$4m$次共享存储器的访问，总计时间为$O(m^2\ell)$。这个界限对*update*也同样适用。

在对读/写共享存储器系统的编程中使用快照　快照共享变量代表一个强大的共享存储器类型。举例来说，使用单个快照共享变量，可以大大简化10.7节中的*Bakery*互斥算法。我们把这留做一道习题。

使用13.1.4节中的技术和一个像本节中的快照算法，一个使用快照共享变量的算法A可以被变换为一个只使用单写者/多读者的读/写共享变量的算法。这个变换需要对A做一些简单限制，这在13.1.4节讨论过。（而且，从技术上讲，对于A中的每一个进程，转换中用到的快照原子对象都有一个相应的端口；A中的进程i可以在同一个端口i上递交*update*和*snap*操作。但是我们可以改动快照原子对象的外部接口和实现以允许这种情况。）

read/update/snap变量　它是原来只支持*update*和*snap*操作的快照共享变量的一个有用变种。read/update/snap共享变量除了支持能够返回整个向量的*snap*操作之外，还支持在共享向量的各个位置执行读操作。当然，一个使用read/update/snap共享变量的模型并不比一个只使用快照变量的模型更强大，因为读可以用*snap*来实现。然而，使用read/update/snap共享变量可以更有效地编程，因为实现一个read/update/snap原子对象来加快读操作是可能的。我们把它留做一道习题。

433

13.4　读/写原子对象

读/写共享变量（寄存器）是用来构建共享存储器多处理器的最基本构件的一种。在这一节，我们考虑根据功能弱一些的寄存器（比如单写者/单读者寄存器）来实现功能强大的多写者/多读者寄存器。更确切的，我们考虑用单写者/单读者共享变量来实现多写者/多读者的读/写原子对象的问题。

13.4.1　问题

给定一个域V和一个初始值$v_0 \in V$。

在例13.1.1中，我们描述了域V上一个1写者/2读者的读/写原子对象的外部接口。通常，域V上的一个m写者/p读者的读/写原子对象具有相似的外部接口，其中端口1,..., m是写端口，而端口$m+1, ..., m+p$是读端口。我们再次令$n=m+p$。

由于我们使用读/写共享变量来实现读/写原子对象，所以我们需要一种方法来将用户在端口上提交的高级读写操作和在读/写共享变量上执行的低级读写操作区分开来。我们约定高级读写操作的名字要大写。这样，在端口i（其中$1 \leq i \leq m$）上有$WRITE(v)_i$输入和ACK_i输出，而在端口j（其中$m+1 \leq j \leq n$）上有$READ_j$输入和v_j输出，$v \in V$。（我们不想把V中的值也大写。）当然还有$STOP_i$输入，$1 \leq i \leq n$。

我们考虑使用一个具有n个进程的共享存储器系统（每个进程对应一个端口）来实现一个m写者/p读者原子对象，其中$n=m+p$。我们假设这个系统中的所有共享变量都是读/写共享变量，但是读者和写者的数目在我们给出的不同算法中将会不同。我们所描述的所有实现都保证无等待终止。

13.4.2　证明原子性的其他引理

我们以一个技术性引理开始，它可以用来证明读/写原子对象外部接口的一个动作序列β满足读/写对象的原子性。这个引理列出了涉及在β中操作上的偏序关系的四个条件。如果有满

434

足这四个条件的顺序存在，那么可以保证有某种办法来插入序列化点，使得β满足原子性。当对算法进行推理时，表明这样偏序的存在通常要比明确定义序列化点要来得简单。

引理13.16 令β是读/写原子对象外部接口的一个（有限或无限）动作序列。假设β对每一个i都是良构的，而且它不包含未完成操作。令$\prod$是由β中所有操作组成的集合。

假设$\prec$是一个满足下列性质的$\prod$中所有操作上的非自反偏序关系：

1）对任意操作$\pi \in \prod$，只存在有限多个操作ϕ，使得$\phi \prec \pi$。

2）如果在β中对π的回应事件在对ϕ的调用事件之前，那么$\phi \prec \pi$不成立。

3）如果π是$\prod$中的一个*WRITE*操作，而ϕ是$\prod$中任一操作，则要么$\pi \prec \phi$，要么$\phi \prec \pi$。

4）每个*READ*操作返回的值都是由之前最后一个*WRITE*操作根据$\prec$所写的值（如果没有这样的*WRITE*，那么为v_0）。

那么β满足原子性属性。

条件1是一个技术性条件，它排除了在一些特定的操作之前有无限多个操作的奇怪顺序。条件2说明$\prec$排序一定和用户的调用和回应顺序相一致。条件3说明$\prec$把所有*WRITE*操作排序，并根据*WRITE*操作对所有*READ*操作排序。条件4说明对*READ*们的回应与$\prec$是一致的。

证明 我们描述如何为每个操作$\pi \in \prod$插入序列化点$*_\pi$。也就是说，我们把序列化点$*_\pi$立即插到对π和所有操作ϕ的最后调用之后紧跟着的一点，其中$\phi \prec \pi$。条件1意味着这个位置是定义良好的。我们排列$*$，使得它们以某种方式相邻放置，这种方式与相关操作上的排序$\prec$是一致的；也就是说，如果π和ϕ的$*$相邻放置的操作，且$\phi \prec \pi$，则$*_\phi$在$*_\pi$之前。

我们来证明序列化点的全序与$\prec$是一致的；也就是说，对于$\prod$中的任意操作π和ϕ，如果$\phi \prec \pi$，那么$*_\phi$在$*_\pi$之前。为此，假设$\phi \prec \pi$。由构造过程可知，$*_\phi$被放置在ϕ和所有在$\prec$顺序中处于ϕ之前的操作的最后调用之后。而$*_\pi$被放置在π和所有在$\prec$顺序中处于π之前的操作的最后调用之后。但是因为$\phi \prec \pi$，所以在$\prec$中处于ϕ之前的任意操作必然也在π之前。把$*_\pi$排在$*_\phi$之前来打破关系，故此$*_\phi$在$*_\pi$之前，得证。 435

我们接下来来证明这些序列化点在所需的时间间隔内。为此，考虑任意操作$\pi \in \prod$。由构造过程可知，π的序列化点$*_\pi$一定出现在π的调用之后。我们来证明$*_\pi$出现在π的回应之前。采用反证法，假设它出现在π的回应之后。则根据构造过程可知，这意味着（在β中）对π的回应一定在某些操作ϕ的调用之前，其中$\phi \prec \pi$。但是这和条件2相矛盾。

余下要证明将操作间隔收缩到各自的序列化点上的结果是底层快照变量类型的一条轨迹。这意味着每个*READ*操作π返回的值都是*WRITE*的值，其中*WRITE*的序列化点是在$*_\pi$之前的最后一个序列化点（如果没有这样的*WRITE*，则返回的值为v_0）。

但是条件3表明根据$\prod$中的所有操作，$\prec$把所有*WRITE*操作都进行了排序。而且根据针对*READ*操作的条件4可知，任意*READ*操作π返回的值都是之前最后一个（根据$\prec$）*WRITE*操作写入的值（如果没有这样的*WRITE*，则为v_0）。由于序列化点的全序与$\prec$相一致，所以π必然返回所需的值。 □

在这一节的剩余部分，我们用引理13.16来证明对象保证原子性条件。

13.4.3 带无界变量的一个算法

我们的第一个算法是*VitanyiAwerbuch*算法，它用单写者/单读者寄存器来实现m写者/p读者的读/写原子对象。（回忆一下$n = m + p$。）这个算法相当简单，但是它的缺点是使用了无界大小的共享变量。

VitanyiAwerbuch算法：

这个算法使用了n^2个共享变量，我们可以想像这些向量被组织在一个$n \times n$的矩阵X中，见图13-9。变量命名为$x(i,j)$：$1 \leqslant i, j \leqslant n$。每个变量$x(i,j)$只能被进程$i$读和进程$j$写；这样，每个进程$i$可以读$X$中第$i$行的所有变量，以及可以写第$i$列中的所有变量。

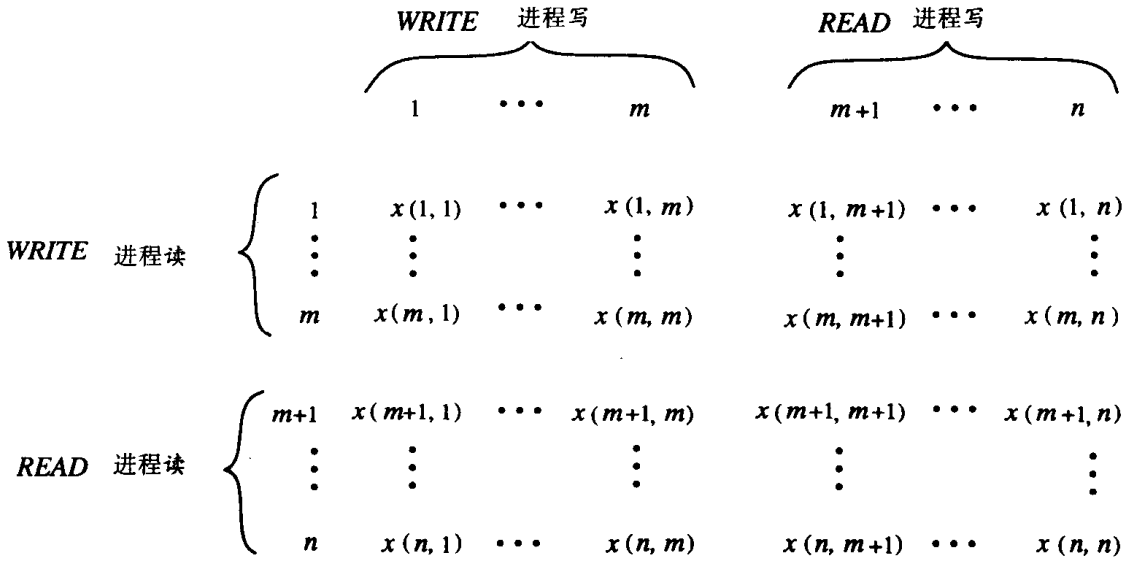


图13-9　*VitanyiAwerbuch*算法中使用的共享寄存器矩阵X

436

每一个共享寄存器$x(i, j)$有下列的域：

$val \in V$，初值为v_0

$tag \in \mathbb{N}$，初值为0

$index \in \{1, \cdots, m\}$，初值为1

把$(tag, index)$对缩写为$tagpair$。我们按词典顺序来对$tagpair$排序。

当一个$WRITE(v)_i$输入发生时，进程i的行为如下。首先，它读取所有变量$x(i, j)$，$1 \leqslant j \leqslant n$（以任何顺序）。令$k$是它发现的最大$tag$。其次，进程$i$对每一个$x(j, i)$, $1 \leqslant j \leqslant n$，执行单个写，设置$x(j, i)$的三个域如下：

1）$x(j,i).val := v$

2）$x(j,i).tag := k+1$

3）$x(j,i).index := i$

最后，进程输出ACK_i。

当一个$READ_i$输入发生时，进程i的行为如下。首先，它读取所有变量$x(i, j)$，$1 \leqslant j \leqslant n$（以任何顺序）。与它发现的与最大$tagpair=(tag,index)$对应的$(val,tag,index)$三元组记为$(v, k, j)$。其次，进程$i$对每一个$x(j,i)$, $1 \leqslant j \leqslant n$，执行单个写，设置$x(j, i)$的三个域如下：

1）$x(j,i).val := v$

437

2）$x(j,i).tag := k$

3）$x(j,i).index := j$

（即，它把读到的最好信息传播给它能够写入的所有变量。）最后，进程i输出v_i（即在端口i上输出值v）。

定理13.17　*VitanyiAwerbuch*算法是一个保证无等待终止的读/写原子对象。

为了证明*VitanyiAwerbuch*算法的正确性，我们可以像证明快照算法那样明确插入序列化点，然后证明原子性得到满足。然而，对于*VitanyiAwerbuch*算法，不容易看出（相对于前面的算法）序列化点到底应该放置在何处。一个更自然的证明策略是建立一个基于*tagpair*值的操作偏序，然后证明这个偏序满足引理13.16的条件。

证明 良构性和无等待终止是易见的。对原子性，我们使用引理13.16。

令α是*VitanyiAwerbuch*算法的任意一个运行。由引理13.10可知，我们可以不失一般性地假设α不包含未完成操作。我们以一个简单的声明开始。

声明13.18 对于任意变量$x(i,j)$，$tagpair=(tag,index)$的值在α中是单调非减的。

证明 固定i和j。注意到变量$x(i,j)$只能被进程j写，由良构性可知，进程j的所有操作必定串行发生。此外，在任意个j的完全操作之后，第j列上的所有变量包含相同的*tagpair*。

每当j执行一个操作时，它首先读取第j行上的所有变量，包含“对角线”上的变量$x(j,j)$。然后它所写的*tagpair*最后不小于它在$x(j,j)$中发现的*tagpair*。但它在$x(j,j)$中发现的*tagpair*与操作之前$x(i,j)$中的*tagpair*相等。因此，这个操作之后$x(i,j)$中的*tagpair*至少和操作之前的一样大。这足以证明这个声明。 □

接下来，将由在α中发生的操作组成的集合记为$\prod$。对每个（*WRITE*或*READ*）操作$\pi \in \prod$将它写入的惟一*tagpair*值记为$tagpair(\pi)$。

声明13.19 α中不同*WRITE*操作的所有$tagpair(\pi)$值是不同的。 438

证明 对发生在不同端口上的*WRITE*操作，这显然是正确的，因为*tagpair*中的*index*域是不同的。

因此我们考虑发生在同一端口上的操作；根据良构性，这些操作串行发生。令π和ϕ是端口i上的两个*WRITE*操作，并且不失一般性地假设π处于ϕ之前。那么在ϕ开始读第i行上的变量之前，π就完成了对第i列上所有变量的写。特别的，ϕ在“对角线”上的变量$x(i,i)$中看到了π或一个更后操作写入的*tagpair*。由声明13.18可知，这个*tagpair*至少和π中的一样大。那么ϕ为自己选择一个更大的，不同的*tagpair*。 □

现在我们来定义$\prod$中操作上的一个偏序。即当下面两者中的一个成立时，我们就说$\pi \prec \phi$：

1）$tagpair(\pi) < tagpair(\phi)$。

2）$tagpair(\pi) = tagpair(\phi)$，$\pi$是一个*WRITE*而$\phi$是一个*READ*。

只要证明这满足引理13.16要求的四个条件就足够了（其中$\beta=trace(\alpha)=\alpha|ext(A \times U)$）。

1）对任意操作$\pi \in \prod$，只存在有限多个操作ϕ使得$\phi \prec \pi$。

采用反证法，假设操作π有无限多个$\prec$前驱。声明13.19表明，它不可能有无限多个*WRITE*操作前驱，因此，它必定有无限多个*READ*操作前驱。不失一般性，我们可以假设π是一个*WRITE*。

那么无限多个*READ*操作具有相同的*tagpair*、t，其中t比$tagpair(\pi)$小。但是π在α中完成的事实意味着$tagpair(\pi)$最终被写入到每一行的某个变量中。之后，声明13.18表明随后调用的任意*READ*操作肯定看到（也因此得到）一个$\geq tagpair(\pi) > t$的*tagpair*。这与无限多个*tagpair*为t的*READ*操作的存在相矛盾。

2）在β中，如果对π的回应事件处于对ϕ的调用事件之前，那么$\phi \prec \pi$不成立。

假设π的回应在ϕ的调用之前。当π完成时，它的*tagpair*已经都写到了它的所有列变量上了。因此由声明13.18可知，当ϕ读它的行变量时，它读到的*tagpair*至少和$tagpair(\pi)$

一样大。所以，*tagpair*(ϕ)被选得至少和*tagpair*(π)一样大。此外，如果ϕ是一个*WRITE*操
439 作，那么*tagpair*(ϕ)严格大于*tagpair*(π)。

由于*tagpair*(ϕ)$\leq$ *tagpair*(π)，我们可以使$\phi \prec \pi$的惟一途径是当*tagpair*(ϕ) =*tagpair*(π)时，π是一个*READ*操作而ϕ是一个*WRITE*操作。但是这是不可能的，因为如果ϕ是一个*WRITE*操作，那么根据上面提到的，我们有*tagpair*(ϕ) > *tagpair*(π)。所以$\phi \prec \pi$不成立。

3）如果π是$\prod$中的一个*WRITE*操作而ϕ是$\prod$中的任意操作，则要么$\pi \prec \phi$，要么$\phi \prec \pi$。

由声明13.19可知，所有的*WRITE*操作都取得不同的*tagpair*。这表明所有的*WRITE*是完全有序的，而且所有和*WRITE*有关的*READ*也都是完成有序的。

4）每一个*READ*操作返回的值是基于$\prec$的倒数第一个*WRITE*操作写入的值（如果没有这样的*WRITE*，则值为v_0）。

令π是一个*READ*操作。π返回的值v正好就是它在行变量中找与最大*tagpair*, t相关的值。存在两种情况：

a）值v是被具有*tagpair* t的某一*WRITE*操作ϕ写入的。

在这种情况中，排序定义保证ϕ是$\prec$顺序中π之前的最后一个*WRITE*，得证。

b）$v = v_0$并且t =0。

在这种情况下，排序定义保证$\prec$顺序中π之前没有*WRITE*，得证。

复杂性分析　*VitanyiAwerbuch*算法用到了n^2个共享变量，每一个的大小都是无界的，尽管底层的域V是有限的。每个*READ*和*WRITE*的完成都涉及到4n次对共享存储器的访问，总时间复杂度为$O(n\ell)$。

13.4.4　两个写者的有界算法

和*UnboundedSnapshot*算法一样，*VitanyiAwerbuch*算法有同样的不足，它使用大小无界的共享变量来存储无界的标签。很多其他算法已经设计成只使用有界数据，但遗憾的是，大多数
440 都很复杂（效率也太低，以致不实用）。在这一节，我们只针对一种特殊情况来给出一个非常简单的算法。

也就是说，我们描述使用两个1写者/p+1读者的寄存器$x(1)$和 $x(2)$来实现一个2写者/p读者的读/写原子对象的*Bloom*算法。（现在n=2+p。）每个$x(i)$可被*WRITE*进程i写和被所有其他进程读。图13-10描述了有两个读者的特殊情况下的体系结构。这个算法相当简单，但难于扩充到有更多写者的情况中。

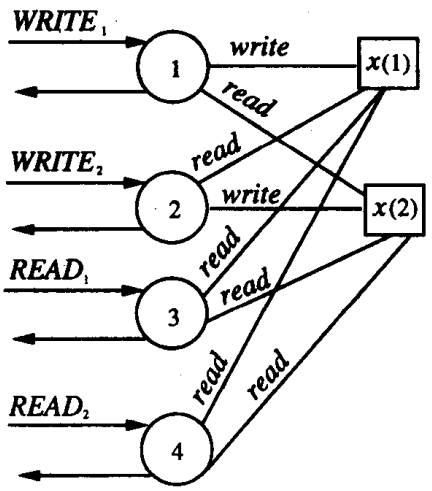


图13-10　有两个写者的*Bloom*算法的体系结构

Bloom算法：

这个算法用到了两个共享变量$x(1)$和$x(2)$，其中$x(i)$可被进程i写和被所有其他进程读。在这里，如果$i=1$，$\bar{i}$就代表2，如果$i=2$，$\bar{i}$就代表1。寄存器$x(i)$有以下域：

$val \in V$，初值为v_0

$tag \in \{0, 1\}$，初值为0

当一个$WRITE(v)_i$发生在端口i上时，$i \in \{1, 2\}$，进程i的行为如下。首先，它读$x(\bar{i})$；将它发现的tag记为b。然后它写$x(i)$，按如下方法设置域：

1）$x(i).val := v$

2）$x(i).tag := b + i \bmod 2$

最后，它输出ACK_i。

这样，当$WRITE$进程i执行一个$WRITE$时，它不仅把新值写入它的变量，还试图让这两个变量中的tag值之和等于它的索引对2取模之后的值。也就是说，进程1总是试图让这两个变量中的tag不相等，而进程2总是试图让它们相等。

当一个$READ_i$发生在端口i上时，$3 \leqslant i \leqslant n$，进程$i$的行为如下。首先，它读取两个寄存器；将它找到的两个tag之和对2取模，得到的值记为b。那么如果$b=1$，它就重读寄存器$x(1)$；如果$b=0$，它就重读寄存器$x(2)$；并返回它找到的val。

这样，所有的$READ$进程以完全一样的方式运作。每个$READ$进程都读取两个寄存器以决定它们是否包含相等的tag。如果tag是相等的，进程从$x(2)$取得返回值，否则从$x(1)$取得返回值。

定理13.20 *Bloom*算法是一个保证无等待终止的读/写原子对象。

为了证明这个算法的正确性，我们可以明确插入序列化点，然后表明它满足原子性。然而，这次我们使用一个有趣的策略，它基于引理13.16和8.5.5节定义的模拟证明的组合。我们首先定义*Bloom*算法的一个变量*IntegerBloom*，它使用整数值的tag而不是位值的tag。我们使用引理13.16来证明*IntegerBloom*是正确的。然后使用从*Bloom*到*IntegerBloom*的模拟关系来证明*Bloom*的正确性。

IntegerBloom算法：

这个算法使用了两个共享变量$x(1)$和$x(2)$，其中$x(i)$可被进程i写和被所有其他进程读。寄存器$x(i)$有以下域：

$val \in V$，初值为v_0

$tag \in \mathbb{N}$，当$i=1$时，初值为0；$i=2$时，初值为1

当一个$WRITE(v)_i$发生在端口i上时，$i \in \{1, 2\}$，进程i运转如下。首先，它读取$x(\bar{i})$；将它发现的tag记为b。然后它写$x(i)$，设置字段如下：

1）$x(i).val := v$

2）$x(i).tag := t + 1$

最后，它输出ACK_i。

当一个$READ_i$发生在端口i上时，$3 \leqslant i \leqslant n$，进程$i$运转如下。首先，它读取两个寄存器；令$t_1$和$t_2$是它发现的各自的$tag$。存在两种情况：如果$|t_1 - t_2| \leqslant 1$，那么进程$i$重新读取拥有较大$tag$的那个寄存器并返回它找到的$val$。（这个寄存器必定是惟一的，原因在于：正如我们在下面的引理13.21中陈述的那样，$x(1)$中的tag始终是偶数，而$x(2)$中的tag始终是奇数。）否则——即如果$|t_1 - t_2| > 1$——进程i非确定地任选一个寄存器来重读并返回它找到的val。

以下引理给出了*IntegerBloom*的一些基本性质。这很容易证明。

引理13.21　在*IntegerBloom*的任意可到达状态中，下列为真：

1）$x(1).tag$是偶数。

2）$x(2).tag$是奇数。

3）$|x(1).tag - x(2).tag| \leq 1$。

定理13.22　*IntegerBloom*算法是一个保证无等待终止的读/写原子对象。

证明　同定理13.17的证明相似。良构性和无等待终止是易见的。对原子性，我们使用引理13.16。令α是*IntegerBloom*算法的任意运行。和前面一样，我们不失一般性地假设α不包含未完成操作。

声明13.23　对每一个变量$x(i)$，α中的tag值是单调非减的。

把由α中发生的操作组成的集合记为$\prod$。对$\prod$中的每一个*WRITE*操作π，我们定义$tag(\pi)$为π在它的写步中写入的tag值。

现在我们定义$\prod$中操作上的一个偏序关系。首先，我们按tag值来排列所有的*WRITE*操作。如果两个*WRITE*操作有相同的tag值，那么它们一定属于同一个写者，我们按发生顺序来排列它们。接下来，我们将$\prod$中的每个*READ*操作恰好排在那个写入它所获得的值的*WRITE*之后（如果没有这样的*WRITE*时，则排在所有的*WRITE*之前）。

这足以满足引理13.16要求的四个条件（其中 $\beta = trace(\alpha) = \alpha|ext(A \times U)$）。条件3和4是直接
443 的，所以我们来证明条件1和2。以下声明对于我们的证明是有用的：

声明13.24　如果*WRITE*操作π的写步处于*WRITE*操作ϕ的调用之前，那么$\pi \prec \phi$。

证明　如果π和ϕ发生在同一个端口上，则由声明13.23可知$tag(\pi) \leq tag(\phi)$，又由$\prec$的定义可知$\pi \prec \phi$。另一方面，如果π和ϕ发生在不同的端口上，那么ϕ在π的端口上读取π或一个更后*WRITE*的结果。由声明13.23可知，ϕ读取的tag大于或等于$tag(\pi)$。因此，$tag(\pi) < tag(\phi)$，故又有$\pi \prec \phi$。 □

声明13.25　如果*WRITE*操作π的写步处于*READ*操作ϕ的调用之前，那么$\pi \prec \phi$。

证明　我们必须证明ϕ返回π的结果，或者返回其他某一满足$\pi \prec \psi$的*WRITE* ψ的结果。令$t = tag(\pi)$，且假设π发生在端口i上。

当ϕ被调用时，声明13.23意味着$x(i).tag \geq t$，而且引理13.21意味着$|x(1).tag - x(2).tag| \leq 1$。因此，当$\phi$被调用时，有$x(\bar{i}).tag \geq t - 1$。由$\prec$的定义和声明13.23可知，惟一的问题是$\phi$返回某一个带有$tag = t - 1$的*WRITE*的值。假设就是这种情况。

那么在ϕ的第一次读或第二次读中，它一定会看到$x(\bar{i})tag = t - 1$，并在它的第三次读时也一定会看到$x(\bar{i}) = t - 1$。如果ϕ在它的第一次和第二次读时看到$x(i).tag=t$，那么tag的$t - 1$和t的组合将导致ϕ去选择重读寄存器$x(i)$，而不是寄存器$x(\bar{i})$，矛盾。因此ϕ一定会看到$x(i).tag > t$。但是引理13.21意味着当ϕ看到$x(i).tag>t$之前，$x(\bar{i})>t - 1$必定成立。这表明ϕ不可能在它的第三次读时看到$x(\bar{i}) = t - 1$，矛盾。 □

使用声明13.24、声明13.25，条件1很容易证明；我们把它留做一道习题。

对条件2，假设在β中π的回应事件处于ϕ的调用之前。如果π是一个*WRITE*，那么声明13.24、声明13.25表明$\pi \prec \phi$。因此假设π是一个*READ*。采用反证法，假设$\phi \prec \pi$。

如果ϕ是一个*WRITE*，那么显然π不可能返回ϕ的结果，因为直到ϕ完成时ϕ才执行它的写步。

因此惟一的问题是π返回其他某一满足$\phi \prec \psi$的*WRITE* ψ的结果。但是在这种情况中，ψ中的写步处于π的结束之前，因此也处于ϕ的调用之前。但声明13.24意味着$\psi \prec \phi$，矛盾。

另一方面，如果ϕ是一个*READ*，则$\phi \prec \pi$的假设意味着一定存在某一个*WRITE*操作ψ，使得$\phi \prec \psi$而π获得ψ的结果。由于π获得ψ的结果，ψ中的写步必然在π的结束之前，也因此在ϕ的调用之前。但是声明13.25意味着$\psi \prec \phi$，矛盾。 □ 444

现在我们使用一个模拟关系来证明*Bloom*算法和*IntegerBloom*算法之间的对应关系。8.5.5节描述了通用策略，而在例8.5.6的证明和10.9.4节的证明中我们曾经用到过这种方法。

两个算法中的对应关系就是（非常奇怪）：*Bloom*算法中用到的从{0，1}中取值的*tag*就是*IntegerBloom*算法中用到的整数值*tag*的倒数第二位。

例 13.4.1 *Bloom*算法中的位和整数

考虑*IntegerBloom*算法的一个运行，其中*WRITE*操作在端口1和2上交替发生，且开始时发生在端口1上；每个*WRITE*只有在前一个*WRITE*完成后才开始。那么每个*WRITE*产生一个持续增大的*tag*。我们把这两个寄存器中的*tag*值写为二进制，见图13-11。最初，$x(1)$和$x(2)$的*tag*分别为0和1。第一个$WRITE_1$设置$x(1).tag$:=2，随后一个$WRITE_2$设置$x(2).tag$:=3,如此类推。

在*Bloom*的相应运行中，两个寄存器中的*tag*值见图13-12。两个寄存器都以$tag = 0$作为开始。每个$WRITE_1$设置$x(1).tag$为不等于$x(2).tag$的值，而每个$WRITE_2$设置$x(2).tag$为等于$x(1).tag$的值。

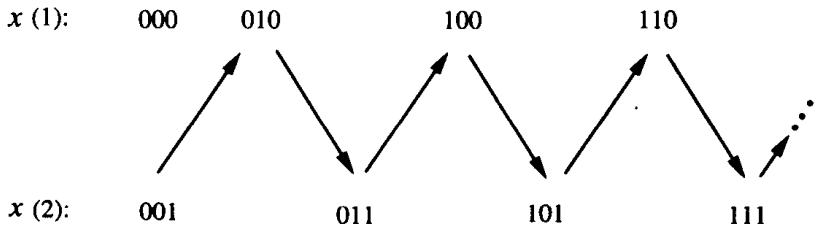


图13-11 在*IntegerBloom*算法中，两个寄存器中的连续*tag*值

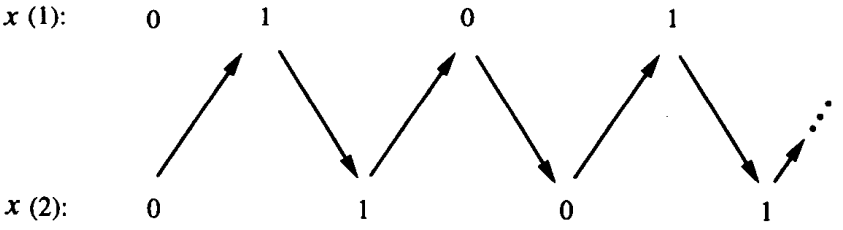


图13-12 在*Bloom*算法中，两个寄存器中的连续*tag*值

注意：在每种情况中，*Bloom*运行中的*tag*正好就是*IntegerBloom*运行中的对应*tag*的倒数第二位。

例13.4.1中描述的对应关系对两个算法中的所有运行都成立。如果s和u分别是*Bloom*和*IntegerBloom*系统（算法加上用户）的状态，如果以下条件成立，则定义$(s, u) \in f$（或$u \in f(s)$）：除了u有一个整数值的*tag*、t，而s有一个其值等于t的倒数第二位的*tag*之外，u和t的所有其他状态部分都是相同的。 445

引理13.26 f是一个从*Bloom*算法到*IntegerBloom*算法的模拟关系。

证明概要　由于这两个算法的惟一初始状态都与f相关，所以模拟关系定义的开始条件直接可得。有趣的是步条件。对于*Bloom*中任意步 (s, π, s')和任意$u \in f(s)$，其中s和u是可到达状态，如果在*IntegerBloom*中存在对应的一步 (u, ϕ, u')（其中$u' \in f(s')$且ϕ'“几乎”和π一样），那么结论就成立了。更精确的，除了ϕ涉及一个整数值，而π涉及倒数第二位之外，ϕ和π是一样的。我们基于π来进行讨论。如果π是一个调用或回应事件，那么证明起来很简单。有趣的步是那些在*WRITE*操作内的写步和在*READ*操作内的第三个读步。

假设(s, π, s')是*Bloom*中的一步，其中作为$WRITE_i$操作的一部分，进程1写入$x(1)$。进程1令$x(1).tag$不等于值b，这个b是它读$x(2).tag$后记住的值。也就是说，$s'.\ x(1).tag \neq b$。在*IntegerBloom*的对应状态u中，进程i记住从$x(2).tag$读取的整数值$tag\ t$；鉴于$u \in f(s)$，b必然是t的倒数第二位。令u'是*IntegerBloom*中生成的惟一状态。则有$u'.\ x(1).tag = t+1$。为了证明$u' \in f(s')$，我们需要证明$u'.\ x(1).tag$的倒数第二位等于$s'.\ x(1).tag$。也就是说，$t+1$的倒数第二位不等于b。这成为t是奇数（由引理13.21）和b是t的倒数第二位这两个事实的必然结果。

对于其中进程2写入$x(2)$的(s, π, s')的情况，证明过程类似。

现在假设(s, π, s')是*Bloom*的一步，其中进程2执行了*READ*之内的第三个读步。关键在于*IntegerBloom*允许进程i读同一个寄存器，$x(1)$ 或$x(2)$。假设在状态s中，*Bloom*的进程i记得在$x(1)$ 和$x(2)$中读取的tag分别为b_1和b_2；类似地，假设在状态u中，*IntegerBloom*中的进程i记得在
446 $x(1)$ 和$x(2)$中读取的tag分别是t_1和t_2。由于$u \in f(s)$，我们知道b_1是t_1的倒数第二位，而b_2是t_2的倒数第二位。存在三种情况。

1）$t_1 = t_2 + 1$

则引理13.21表明t_1和t_2的倒数第二位是不相等的。在这种情况下，*Bloom*和*IntegerBloom*都读寄存器$x(1)$。

2）$t_2 = t_1 + 1$

引理13.21表明t_1和t_2的倒数第二位是相等的。在这种情况下，*Bloom*和*IntegerBloom*都读寄存器$x(2)$。

3）$t_1 \neq t_2+1$且$t_2 \neq t_1+1$

引理13.21表明$|t_1 - t_2|>1$。在这种情况下，*IntegerBloom*可以读任意一个寄存器。

现在我们证明定理13.20，它断定了*Bloom*算法的正确性。 □

（定理13.20的）证明　引理13.26和定理8.12表明*Bloom*系统的每一条轨迹都是*IntegerBloom*系统中的一条轨迹。（回忆一下，这儿的轨迹包括端口上的调用和回应，以及*stop*事件。）定理13.22表明良构性和原子性条件对*IntegerBloom*成立。因为良构性和原子性条件可以表达为轨迹属性，它们被传到了*Bloom*算法。无等待终止条件是易见的。 □

复杂性分析　*Bloom*算法用到了两个共享变量，每个可取$2|V|$个值。每个操作对共享存储器的访问次数是一个常数，时间复杂度为$O(\ell)$。

13.4.5 使用快照的算法

在这最后一小节中，我们使用一个快照共享变量来实现一个无等待的m写者/p读者的读/写原子对象*SnapshotRegister*。（令$n = m + p$。）把*SnapshotRegister*和13.3节中快照原子对象的实现组合起来，并使用推论13.9，我们就可以使用1写者/n读者共享寄存器来实现无等待m写者/p读
447 者原子对象。

尽管针对将要实现的读/写原子对象的底层域V是有界的，*SnapshotRegister*用到的快照共享变量的大小还是无界的。尽管有一定难度，我们还是可以修改*SnapshotRegister*算法，使它只使

用一个有界的快照共享变量。

*SnapshotRegister*算法：

这个算法使用单个共享变量x，它是一个基于长度为m的向量的快照对象。对x中的每一个分量，域W由(val, tag)对组成，其中$val \in V$而$tag \in \mathbb{N}$；初始值w_0为$(v_0, 0)$。

每一个*WRITE*进程i，$1 \leq i \leq m$，在x上执行$update(i,w)$和*snap*操作，而每一个*READ*进程i，$m+1 \leq i \leq n$，只在x上执行*snap*操作。

当一个$READ_i$输入发生在端口i上时，$m+1 \leq i \leq n$，进程i的行为如下。首先，它在x上执行一个*snap*操作，由此确定一个向量u。令j，$1 \leq j \leq m$，是具有最大$(u(j).tag, j)$对（按字典顺序）的索引。那么进程i返回相关的值$u(j).val$。

当一个$WRITE(v)_i$输入发生在端口i上时，$1 \leq i \leq m$，进程i的行为如下。首先，它对x执行一个*snap*操作，由此确定一个向量u。和上面一样，令j，$1 \leq j \leq m$，是具有最大$(u(j).tag, j)$对（按字典顺序）的索引。然后进程i执行一个$update(i, (v, u(j).tag+1))$。最后进程$i$输出$ACK_i$。

*SnapshotRegister*算法和*VitanyiAwerbuch*算法有点相似。但它更加简单，因为快照共享存储器提供了额外的功能。

定理13.27 *SnapshotRegister*算法是一个保证无等待终止的读/写原子对象。

证明概要 使用引理13.16，这里的证明与对*VitanyiAwerbuch*算法和*IntegerBloom*算法的证明类似，且更加简单。具体证明留做一道习题。 □

复杂性分析 即使底层的域V是有限的，*SnapshotRegister*算法使用的快照共享变量还是无界大小的。每个操作对共享存储器的访问次数是一个常数，时间复杂度为$O(\ell)$。 448

分级构造 定理13.27与快照原子对象的任意无等待实现一起，使用1写者/$m+p$读者共享寄存器来实现m写者/p读者的读/写原子对象，这个实现是无等待的。它的证明基于推论13.9。（从技术上讲，为了应用推论13.9，我们需要一个带$n = m + p$个端口的快照原子对象，每个端口对应一个进程——举例来说，*WRITE*进程i要在同一个端口上执行它的*update*和*snap*操作。改动快照原子对象的外部接口和实现以满足这个要求是没有问题的。）

扩展 存在*SnapshotRegister*算法的几个有趣的扩展，它们也能正常工作。首先，在一个$WRITE_i$期间，如果$i = j$——即如果进程i自己拥有最大的*tag*对——那么它可以使用它上次用过的那个*tag*。其次，i可以使用非负实数值的*tag*而不是整数值的*tag*。这样写者i选出的*tag*可以是严格大于它看到的最大*tag*的任意实数。如果i自己拥有最大的*tag*对，那么它可以重用它前面的*tag*。在证明使用快照共享变量来实现读/写原子对象的方法的正确性时，这些扩展都是很有用的。

13.5 参考文献注释

“原子对象”的思想首先出现在Lamport[181,182]关于读/写原子对象的工作中。Herlihy和Wing[153]把原子性的概念扩展到任意变量类型并将它重新命名为可线性化。König引理最初由König[170]证明；在Knuth[169]的书中也有一个证明。规范的无等待原子对象自动机来自Merritt[3]的工作。原子对象和共享变量之间的联系来自Lamport和Schneider[186]的工作和Goldman和Yelick[139]的工作。用读/写对象实现读 - 改 - 写原子对象的不可能性结果归功于Herlihy[150]。

快照原子对象的思想归功于Afek、Attiya、Dolev、Gafni、Merritt和 Shavit[3]，还归功于Anderson[11,12]，这受到Chandy和Lamport[68]关于分布网络中一致全局快照方面工作的启发。

这儿给出的快照原子对象的实现，包括*UnboundedSnapshot*和*BoundedSnapshot*，归功于Afek等。用于*BoundedSnapshot*协议的握手策略归功于Peterson[240]。后来Attiya and Rachman[26]设计出
449 一个原子快照算法，它只要求$O(n \ell \log n)$的时间而不是$O(n^2)$的时间。

现在有很多算法使用更简单类型的读/写寄存器来实现读/写原子对象。*VitanyiAwerbuch*算法出现在Vitanyi和Awerbuch[283]的论文中。这篇论文还包括一个使用有界共享变量的算法，可惜那个算法是错误的。*Bloom*算法归功于Bloom[53]，而*SnapshotRegister*算法来源于Gawlick，Lynch和Shavit[135]的工作。用单写者/单读者寄存器实现单写者/多读者原子对象的有界算法已经被Singh,Anderson和Gouda[263]以及Haldar和Vidyasankar[144]设计出来了。使用单写者/多读者寄存器实现多写者/多读者原子对象的有界算法已经被Peterson和Burns[241]；Schaffer[254]；Israeli和Li[162]；Li,Tromp,和Vitanyi[196]；Dolev和Shavit[100]设计出来了。特别的，Schaffer的算法改正了Peterson和Burns算法中的错误。Gawlick,Lynch,和Shavit[135]描述了使用有界快照变量的多写者/多读者原子对象的实现，并通过把它与*SnapshotRegister*算法的扩展版本联系起来，使用模拟证明方法来证明了这个算法的正确性。这些构造中的几个使用了“有界时戳”的概念。有界时戳算法已经被Israeli和Li[162]；Dolev和Shavit[100]；Gawlick、Lynch、Shavit[135]；Israeli和Pinchasov[163]；Dwork和Waarts[107]；Dwork、Herlihy、Plotkin和Waarts[102]给出。

Attiya和Welch[28]将读/写原子对象的实现成本和具有更弱一致性要求的读/写对象的实现成本进行了比较。它们的工作针对异步网络模型。

13.6 习题

13.1 为一个2写者/1读者原子对象定义外部接口，并针对这个外部接口给出几个有趣的满足原子性和不满足原子性的序列例子。必须包括有限序列和无限序列，以及包含未完成操作的序列。

13.2 考虑一个读－改－写原子对象，它的域V是整数集，它的初始值是0（见9.4节读－改－写变量
450 类型的定义——回忆一下，一个读－改－写共享变量的返回值是这个操作之前变量中的值。）这个对象有两个端口：端口1只支持递增操作（它把对象中的值加1），端口2 只支持递减操作（它减1）。下面序列中哪一个满足原子性？

a）$increment_1$，$decrement_2$，0_1，0_2

b）$increment_1$，$decrement_2$，-1_1，0_2

c）$increment_1$，$decrement_2$，0_1，1_2

d）$decrement_2$，$increment_1$，0_1，$increment_1$，1_1，$increment_1$，2_1，$increment_1$，3_1...

e）$decrement_2$，$increment_1$，0_1，$increment_1$，0_1 $increment_1$，1_1，$increment_1$，2_1...

13.3 对定理13.1做出更详细的证明。特别的，对文中提到的“存在从根开始的任意长的路径、并且一条无限路径可以产生针对整个序列β的一个正确选择”做出更详细的描述。

13.4 扩展变量类型的定义以允许有限数量个初始值而不是只有一个，并允许有限非确定选择而不只是一个函数。扩展定理13.1和它的证明以适应这种新设置。如果我们允许无限非确定性的话又将如何？

13.5 假设我们修改例13.1.4，使得系统除了支持读和递增操作之外还支持递减操作。除了增加如下一点之外，算法和以前的相同：当一个$decrement_i$输入发生在端口i上时，进程i递减$x(i)$。所得的系统是一个读/递加/递减原子对象吗？给出证明，或给出一个反例运行。

13.6 证明定理13.4。

13.7 证明定理13.5。

13.8 证明定理13.6。

13.9 证明：如果我们不加入关于*A*的*turn*函数的特别假设，定理13.7是错误的。

13.10 给出*RMWfromRW*算法的前提－结果代码。你的描述应该模块化，把互斥组件描述成独立的自动机，并使用I/O自动机合成来把它和*RMWfromRW*算法的主要部分组成起来。证明你的算法工作正常（假设互斥组件是正确的）。 451

13.11 考虑*UnboundedSnapshot*算法的一个改动，其中每个*snap*和嵌入*snap*寻找某一$x(i)$的三个不同标签而不是书中所说的四个。这个改动的算法仍然正确吗？证明此算法是正确的，或者给出一个反例运行。

13.12 考虑*UnboundedSnapshot*算法的一个改动，其中当进程*i*运行一个*snap*操作以及运行一个*update*操作时，它将$x(i).tag$加1。（其中，$x(i).val$和$x(i).view$不变，嵌入*snap*操作不做任何改动。）

这个改动过的算法仍然正确吗？证明此算法是正确的，或给出一个反例运行。

13.13 研究问题：你能基于*UnboundedSnapshot*算法与合适的规范无等待终止原子对象自动机的形式化关系，给出*UnboundedSnapshot*算法的另一个证明吗？

13.14 设计*BoundedSnapshot*算法的一个改动，它没有*toggle*位。在你的算法中，一个*snap*进程不仅基于握手位，还基于*val*域来确定两遍读的一致性。证明你的算法是正确的。

13.15 研究问题：设计一个比*BoundedSnapshot*算法更为有效的无等待快照原子对象的实现，而且还是使用有界大小的单写者/多读者的读/写共享变量。你能设计出一个在进程数的线性时间内而不是在二次方时间内终止的这样快照原子对象吗？

13.16 研究问题：设计一个快照原子对象的良好实现，它允许对同一向量部分的更新操作发生在不同的端口上（因此，更新操作是并发的）。

13.17 给出10.7节中*Bakery*算法的简化版本，这个版本使用快照共享变量。证明其正确性。

13.18 详细陈述并证明：使用快照原子对象来解决带1故障终止的一致性问题是不可能的。 452

13.19 使用单写者/多读者的读/写共享变量，给出read/update/snap原子对象的一个有效实现。证明它的正确性并分析它的复杂性。

13.20 给出10.7节中*Barkery*算法的简化版本，这个版本使用read/update/snap共享变量。尽可能地使你的算法简单有效。证明它的正确性并分析它的复杂性。在你的复杂性分析中，根据一个底层模型来考虑实现read/update/snap变量的代价，其中这个底层模型基于习题13.19中的单写者/多读者的读/写共享变量。

13.21 扩展引理13.16以处理任意变量类型而不仅仅是读/写类型。

13.22 *VitanyiAwerbuch*算法中*READ*协议的“传播阶段”是否是必须的？证明没有它算法也正常工作，或者给出一个反例。

13.23 将序列化点明确插入到任意运行中，其中运行中所有操作都是完成的，从而给出对*VitanyiAwerbuch*算法的另一个正确性证明。然后证明原子性得到满足。

13.24 针对读/写共享变量是单写者/多读者变量的情况，设计*VitanyiAwerbuch*算法的一个简化版本。*READ*协议的传播阶段是必要的吗？证明正确性并分析复杂性。

13.25 证明*Bloom*算法中*READ*协议内的第三个读是必须的。也就是说，修改算法，使得其中每个*READ*都简单地返回它从适当的寄存器（在第一次或第二次）中刚刚读取的值。给出修改后的算法的一个错误运行。

13.26 *IntegerBloom*算法描述的结尾处指明当$|t_1 - t_2|>1$时，进程*i*不确定地任意选出一个寄存器来

进行重读。给出发生这种情况的一个特殊运行。

13.27 在定理13.22的证明中，证明引理13.16的条件1成立。

453 13.28 完善引理13.26的证明。这要求写出*Bloom*和*IntegerBloom*算法的前提 - 结果代码。

13.29 研究问题：将*Bloom*算法扩展到多于两个写者的情况。

13.30 证明定理13.27。

13.31 给出运行例子以表明当按以下任意一种方法放置序列化点时，*SnapshotRegister*算法没有被正确序列化：

a）对于*READ*：放在它的*snap*操作的那个点上；对于*WRITE*：放在它*update*操作的那个点上。

b）对每个操作：放在它*snap*操作的那个点上。

13.32 描述一个使用13.4.5节末中提到的两个方法来扩展*SnapshotRegister*的算法。也就是说，自身*tag*最大的那个*WRITE*进程可以（但不是必须）重用它的*tag*，且允许实数值的*tag*。令你的算法尽可能地不确定。

13.33 设计一个算法，使用快照共享变量来实现一个域为V和初始值为v_0的m读者/p写者的读/写原子对象。与*SnapshotRegister*算法不同，在V是有限的情况下，你的快照变量应该是有界的。（警告：这非常困难。）

13.34 研究问题：使用引理13.16来证明在研究文献中出现的某些其他原子寄存器实现的正确性。

13.35 研究问题：设计高效简单的算法以使用大小有界的单写者/单读者寄存器来实现多写者/多读者的读/写原子对象。

454 13.36 研究问题：设计一个高效简单的原子对象分级结构以用做开发实际多处理器系统的基础。

第二部分B　异步网络算法

第14~22章将介绍针对异步网络模型的算法。在这种模型中，进程的执行是异步的，通过消息交换进行通信。这些章的思想以很多有趣的方式来融合了第一部分和第二部分A中提出的思想。

按照惯例，该部分首先在第14章介绍一个形式化的模型。接下来在第15章中对异步网络的基本算法进行概述，这些算法均直接根据模型来编程。因为有些算法相当复杂，所以我们在第16~19章介绍简化异步网络编程的四种技巧。第一种技巧是引入同步器，在第16章中描述。第二种技巧是在异步网络模型中模拟异步共享存储模型，在第17章中描述。第三种技巧是在异步分布式网络中给事件分配一致的逻辑时间，在第18章中描述。第19章介绍第四种技巧，即在运行时监测异步网络算法。

然后，我们继续研究异步网络环境中的特定问题。第20章研究异步网络中的资源分配问题。第21章在有故障的异步网络中讨论计算问题。最后，在第22章讨论数据链接问题，即在一个不可靠的网络中如何实现可靠的通信。

第14章 建模 IV：异步网络模型

在本章，我们再一次改变计算范例，以异步网络系统代替异步共享存储系统。一个异步网络由一组依靠通信子系统来相互通信的进程构成。在该模型的最常见版本中，这些通信采用的是使用发送和接收操作的点对点模式。此模型的其他版本允许广播操作，即一个进程可以把某个消息发送给网络的所有进程（包括它自己）；或允许多播操作，即一个进程把某个消息发送给一个进程子集。也存在多播模型的特殊情况，例如，允许广播通信和点对点通信的组合情况。在每种情况下，都要考虑各种各样的网络故障行为，包括消息的丢失和重复。

本章包括三个主要小节，分别阐述发送/接收系统、广播系统和多播系统。

14.1 发送/接收系统

像第2章中定义的同步网络模型一样，我们从一个n节点有向图$G = (V, E)$开始。同前面一样，我们分别用符号$out\text{-}nbrs_i$和$in\text{-}nbrs_i$表示图中从节点i流出的邻接节点和向节点i流入的邻接节点，用$distance(i, j)$表示G中从i到j的最短有向路径，用$diam$表示从任意节点到任何其他节点的最大距离。

457 像同步网络模型中一样，我们用G中的节点表示进程，并允许这些进程通过有向边代表的通道进行通信。但与同步模型不同的是，发送/接收系统中没有通信同步环：现在在进程步和通信两方面都允许异步。为描述这种异步，我们将进程和通道表示成I/O自动机。以M作为固定的消息字母表。

14.1.1 进程

与各节点i相联系的进程被表示成I/O自动机P_i。P_i通常有一些输入输出动作，用来与外界用户进行通信，这使得我们能够使用在“用户接口”的轨迹来表达需要异步网络解决的问题。另外，P_i还有$send(m)_{i,j}$形式的输出，其中j是i的流出邻接节点，m是一个消息（是M的一个元素）；以及$receive(m)_{j,i}$形式的输入，其中j是i的流入邻接节点。除了这些外部接口的限制外，P_i可以是一个任意的 I/O 自动机。（为得到一些特定的结果，我们有时可能会在P_i上增加一些额外限制，如限制任务的数量或限制状态的数量。）例 8.1.2 是I/O 自动机进程的一个例子。

在节点进程部分，我们考虑两种类型的故障行为：停止故障和$Byzantine$故障。P_i的停止故障被表示为P_i外部接口中的一个$stop_i$输入动作，该操作的作用是永久停止P_i的所有任务。（这里不限制$stop_i$所引起的状态改变，也不限制后续输入行为导致的状态改变。限制这些状态改变并不重要，因为任何情况下它们的影响在P_i之外都是不可见的。）P_i的Byzantine故障被表示成允许用任意一个与P_i有相同外部接口的I/O自动机替代P_i。

14.1.2 发送/接收通道

与G的每条有向边(i, j)相关的通道被表示成I/O自动机$C_{i,j}$。通道的外部接口由一组$send(m)_{i,j}$形式的输入和一组$receive(m)_{i,j}$形式的输出组成，其中$m \in M$。通常，除了这些外部接口规格说明外，通道可以是任意的I/O自动机。然而，我们关心的通信通道对其外部行为有约束，例如，任何消息都必须在被接收之前发送出来。这种对通道外部行为的必要约束通常可以表示成8.5.2节所定义的轨迹属性P。合法通道必须是那些外部签名为$sig(P)$、公平轨迹属于$traces(P)$的I/O
458 自动机。

轨迹属性P通常有两种说明方式：一种方式是用公理集合列表表示；另一种方式是用给定的I/O自动机表示，该I/O自动机的外部接口为$sig(P)$，并且它的公平轨迹恰好为$traces(P)$。使用公理列表的一个优点是，定义一类通道时更容易，其中每个通道满足一个公理子集。另一方面，给出显式I/O自动机的优势是，整个系统由一系列进程组成，而最常见的合法通道被描述成一组I/O自动机的合成，该合成本身是另一个I/O自动机。这样，我们可以使用针对I/O自动机所开发的证明方法。例如，可以用“状态”符号表示整个系统，包括进程和通道，这种状态符号可以用于不变式断言和模拟证明中。

有时可能需要做一些烦琐的编程，以用I/O自动机说明预期的轨迹属性，在轨迹属性涉及复杂的*活性约束*（*liveness constraint*）时尤其必要。这样经常导致一种*混合策略*（*mixed strategy*），这种策略将安全属性描述成一组基本自动机（提供支持不变式证明和模拟证明所需要的机器），而用特定的活性公理来描述活性属性。这样，整个轨迹属性P就有了自己的轨迹，它们正好是基本自动机中满足活性公理的轨迹。

在本小节的剩余部分，将描述一些特殊的发送/接收通道，在第15~22章中我们将会用到这些通道。

可靠FIFO通道 在学术著作以及我们这里将使用的最常见通信通道是*可靠的FIFO通道*。这种通道所容许的行为可以简单地描述为一个有着恰当外部接口的I/O自动机的公平轨迹，其状态是一个消息队列。$send(m)_{i,j}$操作把m添加到队列的末端。如果m位于队列的前端，则激活$receive(m)_{i,j}$操作，从队列中移出首条消息。任务划分将所有局部控制操作归为一组。例8.1.1中已经给出了这种自动机的形式化定义。

这种自动机不仅是一个关于可靠FIFO通道的允许行为的规格说明，而且它自己也是可靠FIFO通道的一个例子。我们将它称做具有给定有外部接口的*通用可靠FIFO自动机*。

现在我们使用公理来给出关于可靠FIFO通道合法行为的另一种规格说明。即定义一个轨迹属性P，其中$sig(P)$等价于给定的签名，$traces(P)$等价于一组动作序列β，β由$sig(P)$中满足下列条件的动作组成。 459

存在一个函数*cause*，将β中的每个*receive*事件映射为β中的之前一个*send*事件，使得：

1）对于每个*receive*事件π，π和$cause(\pi)$包含相同的消息参数。

2）*cause*是满射函数。

3）*cause*是单射函数（一对一）。

4）*cause*具有保序性，即在β中不存在*receive*事件π_1和π_2，使得π_1位于π_2前面而$cause(\pi_2)$位于$cause(\pi_1)$之前。

*cause*函数是识别“导致”每个*receive*事件的*send*事件的工具。条件1要求只有正确的消息才能被传递；条件2的意思是消息不会丢失；条件3的意思是消息不被复制；条件4的意思是不对消息进行重排序。

需要指出的是(对于这一特定的轨迹属性P)，$traces(P)$中每个序列的*cause*函数是惟一的。

可靠重排序通道 另一种经常涉及的通道类型，它保证所有消息的传递，每个消息只传递一次，但不一定能保留消息的顺序。这种通道类型所容许的行为无法简单地用一个I/O自动机来说明，因此我们使用公理来说明。即除了要去掉*cause*函数的条件4之外，这种规格说明与上述关于可靠FIFO通道的公理规格说明P完全一致。

使用上面提到的混合策略，可以得到另外一个等价的规格说明——用一个基本的I/O自动机A来描述安全属性而用附加公理来描述活性属性。这个基本的自动机A如下所示。（这里，$\cup$和$\in$是多集合操作符。）

A automaton:

Signature:

Input: $send(m)_{i,j}$, $m \in M$

Output: $receive(m)_{i,j}$, $m \in M$

States:

in-transit, a multiset of elements of M, initially empty

460 **Transitions:**

$send(m)_{i,j}$

Effect:

$in\text{-}transit := in\text{-}transit \cup \{m\}$

$receive(m)_{i,j}$

Precondition:

$m \in in\text{-}transit$

Effect:

remove one copy of m from *in-transit*

Tasks:

Arbitrary.

任务划分方法是随意的，因为在这里不用到它。使用自动机A，我们定义了一个轨迹属性P。其签名与$sig(A)$完全相同，$traces(P)$ 是由A的满足下列条件的运行α(α不一定是公平的)组成的轨迹集合。

如果对于α中任意一点和任意一个$m \in M$都有$m \in in\text{-}transit$，那么在α中的某个后续点将发生$receive(m)$ 事件。

有故障的通道 我们也可以考虑会发生某些故障的发送/接收通道。在本书中，我们只讨论消息丢失和消息复制两种类型的通道故障。

当一个通道允许随机的消息丢失但不出现消息复制，或允许随机的消息复制但不会丢失消息，或允许随机的消息复制和消息丢失时，就可以用*cause*函数说明这个通道，说明方法与可靠重排序通道的说明方式相同。我们所要做的只是适当地删除条件2和（或）条件3。

然而，我们常希望能假设一个关于消息丢失和（或）消息复制的数量范围。例如，在考虑消息丢失的时候，我们通常不希望考虑所有消息都丢失的情况，因为这种情况下不能保证一定会发生什么事情。典型的消息丢失限制条件是一个被无数次发送的消息必然被无数次接收。为规范地说明这个问题，我们对*cause*函数使用下面的条件。

强丢失限制（SLL） （对某个特定的m）如果在β中有无数个$send(m)$ 事件，那么在*cause*函数的值域内就有无数个$send(m)$ 事件。

必须指出的是，这就是说有无数个不同的*send*事件成功地完成了它们的消息传递。这个条
461 件并不总是能够得到满足的，例如，如果在一个有无数次*receive*事件的序列中，全部*receive*事件都是由同一个*send*事件引起的，那么该序列就满足不了这个条件。

另一个典型的消息丢失限制条件不涉及任何特定的m，而仅要求无数次的*send*事件引起无数个消息的*receive*事件。

弱丢失限制（WLL） 如果在β中有无数个*send* 事件，那么*cause*函数的值域是无限的。

对于消息复制，我们可能希望限定每个消息的副本数量是有限的，或将副本数限制在某个特定的数量k之内。例如，

有限复制 *cause*函数仅把有限数量的*receive*事件映射到某个特定的*send*事件。

到现在为止，我们已经使用公理描述了所有会产生故障的通道。下面使用混合策略说明两个这样的通道。

例 14.1.1 损耗式FIFO通道

下面定义一个允许有限丢失、有限复制和无重排序的通道。（在22.3节的位变换通信协议描述中将用到该通道。）其自动机如下：

A automaton:

Signature:

As usual.

States:

queue, a FIFO queue of elements of M, initially empty

Transitions:

$send(m)_{i,j}$
Effect:
add any finite number of copies of m to *queue*

$receive(m)_{i,j}$
Precondition:
m is first on *queue*
Effect:
remove first element of *queue*

Tasks:

Arbitrary.

自动机A的定义保证通道不对消息进行重排序，而且只传递每个消息的有限个副本。然而，我们需要附加两个额外的活性条件。

1）如果在某点*queue*非空，那么在后续的另外一点将发生一个*receive* 事件。

2）如果有无限数量的*send* 事件，那么其中成功地将其消息（至少是一个副本）放到*queue* 中的*send* 事件也是无限的。

像以前一样，用A和活性条件的组合来定义一个轨迹属性。这个轨迹属性表明，如果有无限数量的*send*事件，那么其中有相应*receive*事件的*send*事件的数量也是无限的，也就是说，它隐含了弱丢失限制（WLL）条件。

例 14.1.2 损耗式重排序通道

下面定义一个允许有限丢失、有限复制和重排序的通道。（在22.2节的Stenning通信协议描述中将用到该通道。）其自动机表示如下：

A automaton:

Signature:

As usual.

States:

in-transit, a multiset of elements of M, initially empty

Transitions:

$send(m)_{i,j}$
Effect:
add any finite number of copies

$receive(m)_{i,j}$
Precondition:
$m \in$ *in-transit*

of m to *in-transit*

Effect:
remove one copy of m from *in-transit*

Tasks:
Arbitrary.

我们增加两个活性条件。

1）如果在某点*in-transit*非空，那么在后续的另外一点将发生一个*receive*事件。

2）如果有无限数量的*send*事件，那么其中成功地将其消息（至少是一个副本）放到*in-transit*中的*send* 事件也是无限的。

463

像例14.1.1中一样，所得到的轨迹属性表明，如果有无限数量的*send*事件，那么其中有相应的*receive* 事件的*send* 事件的数量也无限，也就是说，它意味着弱WLL条件。

需要提示的是，本例中的规格说明也容许例14.1.1中规格说明所允许的每个轨迹。但是，上一个规格说明并不能支持本例中规格说明所容许的全部轨迹。

14.1.3 异步发送/接收系统

采用通常的I/O自动机合成方法，对进程和通道I/O自动机进行组装，即可获得有向图G的一个异步发送/接收网络系统。图8-3所示的就是这种系统的一个结构示例。这种组合定义允许各个组件之间的正确交互，例如，当进程P_i执行一个$send(m)_{i,j}$输出动作时，通道$C_{i,j}$同时执行一个$send(m)_{i,j}$输入动作。两个组件的状态都发生相应的改变。

有时，把一个发送/接收系统的用户表示成另一个I/O自动机U会很方便。U的外部动作正好是进程的用户接口处的动作。用户自动机U通常被描述成一组用户自动机U_i的合成，每个U_i对应相应有向图的一个节点i。在这种情况下，U_i 的外部动作与P_i用户接口的动作相同。（如果考虑停止故障的情况，用户动作将不包括*stop*动作。）

14.1.4 使用可靠FIFO通道的发送/接收系统的特征

下面给出一个关于异步发送/接收网络系统的基本定理，将在第18、19章使用，定理中的网络系统采用通用可靠FIFO通道。它确定在什么环境下，可以通过对一个公平轨迹中事件的重排序得到另一个公平轨迹。（需要提醒的是，依照I/O自动机合成的形式化定义，轨迹既包括发送和接收事件，也包括用户接口的事件。）要求是重排序要满足一定的基本依赖关系：*receive*事件对相应*send*事件的依赖关系（依赖关系是由*cause*函数惟一决定的）和任何事件对同一节点进程中所有前驱事件的（可能的）依赖关系。

任取一个采用通用可靠FIFO通道的异步发送/接收系统A。令β为A的某个轨迹。我们定义β

464

中事件的无反射偏序关系$\rightarrow_\beta$如下。如果π和ϕ是β中的两个事件，π在ϕ之前，当下列条件之一成立时称$\pi \rightarrow_\beta \phi$，或$\phi$依赖于$\pi$：

1）π和ϕ是同一进程P_i的两个事件。

2）π的形式为$send(m)_{i,j}$，且ϕ是相应的 $receive(m)_{i,j}$事件。

3）π和ϕ由类型1和类型2两类关系组成的关系链所关联。

定理14.1　令A为一个采用通用可靠FIFO通道的异步发送/接收系统，β是A的一个公平轨迹。令γ 是对β中事件重排序后获得的一个序列，且γ 中保持了β中的偏序关系$\rightarrow_\beta$。那么γ 也是A的一个公平轨迹。

证明　根据定理8.4可知，对每个i有$\beta \mid P_i \in fairtraces(P_i)$。因为对于每个$i$有$\gamma \mid P_i = \beta \mid P_i$，所以对于每个$i$有$\gamma \mid P_i \in fairtraces(P_i)$。

根据定理8.4同样可知，对每个i和j有$\beta|C_{i,j} \in fairtraces(C_{i,j})$。因为$\gamma|C_{i,j}$与$\beta|C_{i,j}$有相同的事件集合，而且重排序保持了$P_i$中的事件次序、$P_j$中的事件次序，以及*receive*事件位于相应*send*事件之后，所以$\gamma|C_{i,j} \in fairtraces(C_{i,j})$。

所以根据定理8.6可知$\gamma \in fairtraces(A)$。 □

定理14.1表明，对公平执行进行特定的重排序后，其结果还是公平执行。我们可得到下列推论。

推论14.2 令A是一个采用通用可靠FIFO通道的异步发送/接收系统，α为A的一个公平执行。令γ是对$\beta=trace(\alpha)$中的事件进行重排序所得到的结果，且在γ中保持了偏序关系$\rightarrow_\beta$。那么存在A的一个公平执行α'，使得$trace(\alpha')=\gamma$，并且对任何进程P_i而言α和α'都是不可区分的[⊖]。

证明概要 根据定理14.1可知$\gamma \in fairtraces(A)$。用定理8.4 和定理8.5可以证明存在这样的α'。□

系统A的进程不能将推论14.2所证明存在的执行α'与原始执行α区别开来（即使它们附带了各自的信息）。这意味着进程不清楚一个执行中各事件的整体排序；当事件不与消息以及由偏序关系$\rightarrow_\beta$所描述的进程依赖关系相联系时，进程将确定不了不同进程中事件的顺序。 465

14.1.5 复杂度度量

我们根据所发送和（或）所接收消息的数量来度量通信复杂度。同时，在度量通信复杂度时也可以考虑消息中二进制位的数量。

为度量时间的复杂度，我们采用8.6节为I/O自动机定义的通用时间复杂度度量法的一个特例。具体来讲，就是为每个进程的每个任务设定一个上限l，从而为该任务执行一步的连续两个机会之间的时间设置上限l。我们同样需要关于消息传递时间的假设。对于这一通用可靠FIFO通道的特例，通常为由每个通道的*receive*操作组成的单一任务设置一个上限d；从而为通道中最早消息的传递时间设置上限d。这样，在我们通常的时间复杂度度量中考虑了通道中消息堆积的开销——在kd时间内确保完成通道队列中第k个消息的传递。

关于消息传递时间，我们有时也采用一个不太现实但更简单的假设：赋予通道中各消息的传递时间一个上限d，而不考虑消息堆积的影响。这个假设不能通过将时间界限与任务相关联来表达（但还是有意义的）。而且，我们显然可以把通道时间域假设推广到非通用FIFO通道。

14.2 广播系统

一个广播系统由一组编号分别为1到n的进程和一个广播通道组成，以表示广播通信子系统。同样，我们用M表示一个固定的消息字母表。

14.2.1 进程

在广播系统中，进程i被表示为一个I/O自动机P_i。像发送/接收网络系统中的进程一样，P_i通常具备若干与外部用户进行交互的输入和输出动作。并且，P_i的输出表示形式为$bcast(m)_i$，其中$m \in M$；(同前面一样)输入的表示形式为$receive(m)_{j,i}$，其中$m \in M$。除了这些外部接口的限制外，P_i可以是任意的I/O自动机。参见图14-1。

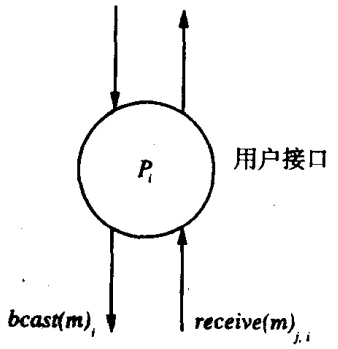


图14-1 异步广播系统的一个进程I/O 自动机 466

⊖ 这里使用的是8.7节中“不可区分”的定义。

14.2.2 广播通道

一个广播通道被表示为一个单一的I/O自动机。它的外部接口由一组$bcast(m)_i$形式的输入和$receive(m)_{i,j}$形式的输出组成，其中$m \in M$。在本书中，只考虑可靠的广播通道，但是定义显现各种形式故障的其他广播通道类型也是有可能的。

可靠广播通道　一个可靠广播通道将每个被广播的消息传递到每个进程，包括发送者本身。我们对消息的发送顺序做一个假设：在每一特定的进程对之间，消息按照FIFO的顺序传递。这样一个通道所允许的行为可以简单地被描述为单个I/O自动机B的公平轨迹，其中自动机B为每个有序进程对维护一个独立的队列。

B automaton:

Signature:

Input:
$bcast(m)_i$, $m \in M$, $1 \le i \le n$

Output:
$receive(m)_{i,j}$, $m \in M$, $1 \le i, j \le n$

States:
for every i, j, $1 \le i, j \le n$:
$queue(i,j)$, a FIFO queue of elements of M, initially empty

467 **Transitions:**

$bcast(m)_i$
Effect:
for all j do
add m to $queue(i,j)$

$receive(m)_{i,j}$
Precondition:
m is first on $queue(i,j)$
Effect:
remove first element of $queue(i,j)$

Tasks:
for every i, j:
$\{receive(m)_{i,j} : m \in M\}$

我们称 B 为有指定外部接口的通用可靠广播通道。

14.2.3 异步广播系统

将进程和一组广播通道I/O自动机组合在一起，即得到一个异步广播系统。

14.2.4 采用可靠广播通道的广播系统的特征

14.1.4节中的定义和结论经修改后可适用于采用通用可靠广播通道的异步广播系统。这里的依赖是指一个*receive*事件对相应*bcast*事件的依赖，以及任一事件对同一节点进程中所有前驱事件的（可能的）依赖。

任取一个采用通用可靠广播通道的异步广播系统A。令β表示A的某个轨迹。我们定义β中事件的无反射偏序关系如下。如果π和ϕ是β中的两个事件，π位于ϕ之前，当下列条件之一成立时，称$\pi \rightarrow_\beta \phi$、或$\phi$依赖于$\pi$：

1）π和ϕ是同一进程P_i的两个事件。

2）π的形式为$bcast(m)_i$，且ϕ是相应的$receive(m)_{i,j}$事件。

3）π和ϕ由类型1和类型2两类关系所组成的关系链所关联。

定理14.3　令A为一采用通用可靠广播通道的异步广播系统，β是A的一个公平轨迹。令γ表示对β
468 中事件进行重排序后得到的序列，且γ保持了β中的偏序关系$\rightarrow_\beta$。那么γ也是A的一个公平轨迹。

证明 此证明留做一道习题。 □

推论14.4 令A为一采用通用可靠广播通道的异步广播系统，α是A的一个公平执行。令γ是对β=$trace(\alpha)$中事件进行重排序后得到的序列，且γ保持了排序关系$\rightarrow_\beta$。那么存在A的一个公平执行α'，使得$trace(\alpha')=\gamma$，并且对任何进程P_i而言，α和α'是不可区分的。

证明 此证明留做一道习题。 □

14.2.5 复杂度度量

我们可以用*bcast*事件的数量或*receive*事件的数量来度量通信复杂度。

为度量时间的复杂度，我们采用一种特殊的I/O自动机时间复杂度度量法。具体来讲，就是为每个进程的每个任务设定一个上限l。并且针对通用可靠广播通道这一特例，我们通常为每个任务设定一个上限d；从而为处于从每个P_i到每个P_j传递途中的最早消息的传递时间设置上限d。这样，我们再次考虑了消息堆积的开销。

对每个消息的传递时间上限d，我们偶尔也会做更严格的假设，并将通道时间界限的假设推广到非通用的可靠广播通道。

14.3 多播系统

*多播系统*是发送/接收系统和广播系统的扩展，它允许每个进程将一个消息发送给网络中的一个进程子集。一个多播系统包括一组编号从1到n的进程，以及一个*多播通道*，以表示多播通信子系统。这个系统被参数化成(i, I)形式的二元组的集合$\mathcal{I}$，其中i是一个进程的索引，I是一组进程索引的集合。每个二元组(i, I)表示进程i可以使用集合I作为多播的目的进程集合。同样，M是一个固定的消息字母表。

14.3.1 进程

我们还是使用I/O 自动机P_i。除了用户接口处的若干动作之外，P_i还有$mcast(m)_{i,I}$形式的输出，其中m是一个消息，$(i, I) \in \mathcal{I}$，以及$receive(m)_{j,i}$形式的输入。除了这些外部接口的限制外，P_i可以是任意的 I/O 自动机。 469

14.3.2 多播通道

一个多播通道被模型化为一个I/O自动机。它的外部接口组成包括$mcast(m)_{i,I}$形式的输入，其中$(i, I) \in \mathcal{I}$，以及$receive(m)_{i,j}$形式的输出。我们仅考虑可靠的多播通道。

可靠多播通道 一个二元组集合为$\mathcal{I}$的可靠多播通道所容许的行为可以简单地表示为下面I/O自动机B的公平轨迹集合。

B automaton:

Signature:

Input:
$mcast(m)_{i,I}$, $m \in M$, $(i, I) \in \mathcal{I}$

Output:
$receive(m)_{i,j}$, $m \in M$, $1 \le i, j \le n$

States:

for every i, j, $1 \le i, j \le n$:
$queue(i, j)$, a FIFO queue of elements of M, initially empty

Transitions:

$mcast(m)_{i,I}$
Effect:

$receive(m)_{i,j}$
Precondition:

for all $j \in I$ do
add m to $queue(i,j)$

m is first on $queue(i,j)$
Effect:
remove first element of $queue(i,j)$

Tasks:
for every i, j:
$\{receive(m)_{i,j} : m \in M\}$

我们称 B 为给定外部接口的通用可靠多播通道。

有一类值得注意的特殊可靠多播通道，其中只有单元素集合和所有进程的集合$\{1,...,n\}$才是允许的目的进程集合。这种通道支持点到点通信和广播通信的组合。需要注意的是，即使
470 在广播通信的消息和点到点通信的消息之间，通信也是按照 FIFO的顺序执行的。

14.3.3 异步多播系统

将进程和一组多播通道I/O 自动机合成起来，即得到一个异步多播系统。14.1.4节的定义和结论能够直接推广到基于通用可靠多播通道的多播系统。同样，广播系统的复杂度度量方法也可以推广到多播系统上。

14.4 参考文献注释

通常，在表示异步发送/接收网络、广播网络和多播网络时，我们不引用特定的文献，在许多关于分布式算法和网络协议形式证明的文章中都有类似资料。用*cause*函数描述消息发送和接收事件之间的显式联系的方法源自下列人员的成果：Fekete，Lynch，Mansour，and Spinelli [112] 以及Afek，Attiya，Fekete，Fischer，Lynch，Nansour，Wang，and Zuck[4].

我们的广播和多播通道表示中只包括了基本的正确性和复杂度属性。在实现和使用具有更强属性的广播和多播通道方面已有很多成果，包括更强的排序需求和容错属性。Hadzilacos 和 Toueg的论文[143]对此做了一个很好的概括。

14.5 习题

14.1 令P为14.1.2节定义的轨迹属性，它描述了一个可靠FIFO 发送/接收通道所允许的行为。证明：$traces(P)$恰好等价于具有相同外部接口的通用可靠FIFO通道自动机的公平轨迹集合。

14.2 令A为实现通用可靠FIFO 发送/接收通道B 的某I/O自动机——也就是说，A有与B相同的外部签名，并且$fairtraces(A) \subseteq fairtraces(B)$。证明：$fairtraces(A) = fairtraces(B)$。（在这个意义上，任何可靠FIFO通道一定是通用的。）

14.3 考虑另一个轨迹属性 Q，Q 也可以作为可靠FIFO发送/接收通道所允许行为的规格说明。除了不需要$cause(\pi)$位于π之前之外，Q 与P相同。证明：对于每个有合适外部接口的I/O
471 自动机A，当且仅当$fairtraces(A) \subseteq traces(P)$时，$fairtraces(A) \subseteq traces(Q)$。

14.4 发送/接收通道C会丢失消息，但不会对消息进行复制或重排序，请用显式I/O自动机给出C的详细描述。假设C在极端情况下可以丢失所有消息。但是C必须列出满足这一条件的全部可能轨迹——例如，不必要求它丢失消息。定义一个从14.1.2节中的通用可靠FIFO 发送/接收通道到C的模拟关系（参见8.5.5节的定义），并且证明这个模拟关系。

14.5 a）证明：同一个可靠重排序发送/接收通道所容许行为的两个规格说明是等价的。

b）可靠重排序发送/接收通道所允许的行为能够等价地用一个I/O自动机定义吗？也就是说，是否存在一个具有适当外部签名的I/O自动机，其公平轨迹正好是指定的动作序列？

14.6 (通道复用技术)用单个“实际的”发送/接收通道能够实现两个或多个“逻辑的”发送/接收通道，每个“逻辑的”发送/接收通道分别用于一个独立的算法或算法的一个独立片段。假设P_1和P_2分别是描述两个独立通道的正确性需求的轨迹属性，而且它们的消息字母表M_1和M_2也不相交。那么乘积轨迹属性$P_1 \times P_2$（乘积轨迹属性的定义见8.5.2节）也可以被看做是另一个通道的规格说明，同时满足二者的需求。

例如，令P_1和P_2分别描述消息字母表M_1和M_2的可靠FIFO通道所容许的行为。令P描述消息字母表$M=M_1 \cup M_2$的可靠FIFO通道所允许的行为。

a）证明：$traces(P) \subseteq traces(P_1 \times P_2)$。

这意味着实现P的任何I/O自动机A（即$extsig(A)=sig(P)$且$fairtraces(A) \subseteq traces(P)$）实际上实现了$P_1$和$P_2$两个通道(即$extsig(A)=sig(P_1 \times P_2)$且$fairtraces(A) \subseteq traces(P_1 \times P_2)$).

b）论述$traces(P) \neq traces(P_1 \times P_2)$。

这也就是说，为了实现P_1和P_2两个通道，对P的行为做了比实际需要更多的限制。

14.7 重做习题14.6，但替换其中的可靠FIFO发送/接收通道，代之以允许随机重排序、强丢失限制（SLL）和下列条件之一的通道： 472

a）无重复。

b）有限重复。

c）任意重复。

14.8 证明：对可靠发送/接收通道而言，FIFO假设是不必要的。确切地说，就是要论述如何将一个基于可靠FIFO通道的发送/接收系统A转换成基于重排序通道的发送/接收系统$T(A)$，A和$T(A)$对外界而言是相同的。对$T(A)$的每个公平执行α，存在A的一个公平执行α'，在用户接口处α'提供与α同样的动作序列。请精确阐述你的结论。

14.9 证明：对可靠广播通道而言，FIFO假设是不必要的。也就是说，有这种假设的系统A可以被转换成没有这种假设的系统$T(A)$，A和$T(A)$对外界而言是相同的。请精确阐述你的结论。

14.10 对定理14.1进行强化，使其包括一个在用户接口需要保持什么样的断言。

14.11 证明定理14.3。

14.12 证明推论14.4。 473

第15章 基本异步网络算法

在本章中，我们描述一组算法来解决在具有可靠FIFO发送/接收通道的异步网络模型中的一些基本问题——领导者选举、构造任意生成树、广播和敛播、广度优先搜索、寻找最短路径以及构造最小生成树。上述问题的大多数都是第4章中同步网络模型所考虑的相同问题。与前面一样，这些问题的出现是由于需要选出一个进程来负责网络计算，并且需要建立合适结构来支持高效通信。本章中我们不考虑故障。

本章中所有的算法都建立在对“裸”异步网络模型的直接编程基础之上。不久我们将看到这个模型的编程工作比起同步网络模型要困难得多。这使得我们去探索一些将编程任务简化和系统化的方法。在本章之后的四章（即第16~19章）中，介绍四种这样的简化技术：同步器、模拟共享存储器、逻辑时间和运行时间监测。

15.1 环中的领导者选举

考虑在第3章同步环中的领导者选举问题。对于这个问题的异步版本，底层无向图仍是具有n个进程的环，进程按照顺时针方向的编号依次是1到n。与前面一样，我们通常对n取模，使
475 得进程n的编号为0，以此类推。这个环可以是单向的，也可以是双向的。图15-1给出异步单向环网络的结构，其中包括进程和通道。

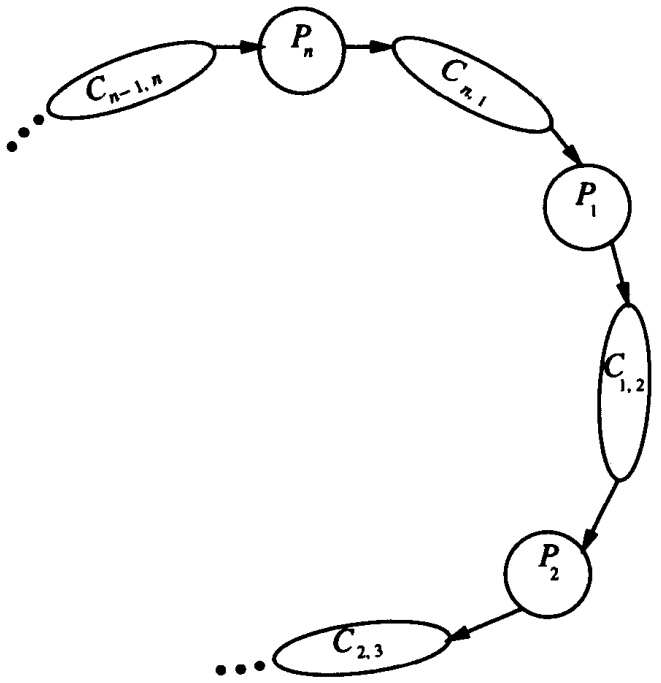


图15-1 单向环网络的体系结构

进程和通道在模型中以I/O自动机来表示。与在同步环境中一样，进程不知道它们的编号（索引），也不知道邻居的编号，而是使用局部的、相对的名字来访问邻居。这使得任意进程都能以任意顺序加到环中。另外，除了拥有*send*和*receive*指令用作进程P_i和通道的交互之外，P_i还

拥有用于宣布成为领导者的输出动作$leader_i$。在本章和以后各章中，假设通道是可靠的FIFO发送/接收通道。我们还假设进程都拥有UID。这样问题就是恰好有一个进程最终产生领导者输出。

15.1.1 *LCR*算法

在3.3节中所描述的*LCR*算法可以很容易地被修改为在异步网络中的运行。在*LCR*算法中，每个进程在环中发送自己的标识符，当进程收到发送来的标识符后，它将该标识符和自己的标识符比较：如果发送来的标识符大于自己的标识符，则让它通过；如果比自己的小，则将该标识符丢弃；如果和自己的标识符相等，则进程输出领导者。

476

在异步网络算法中采用同样的思路；主要的不同之处是现在每个进程的发送缓冲区中必须可以存放任何数目（最大为n）的消息，而不是仅仅一个消息。这一区别产生的原因在于：异步性会导致节点处UID的堆积。我们将修改后的算法称为*AsynchLCR*算法。

在以下代码中，我们把$AsynchLCR_i$作为算法*AsynchLCR*中进程P_i的另一名字。为了方便起见，讨论算法时使用$AsynchLCR_i$和P_i这两个名字；有时干脆称之为“进程i”。在其他地方我们也使用了类似约定。

$AsynchLCR_i$ automaton:

Signature:

Input:
- $receive(v)_{i-1,i}$, v a UID

Output:
- $send(v)_{i,i+1}$, v a UID
- $leader_i$

States:

u, a UID, initially i's UID
send, a FIFO queue of UIDs, initially containing only i's UID
status, with values in {*unknown*, *chosen*, *reported*}, initially *unknown*

Transitions:

$send(v)_{i,i+1}$
- Precondition:
  - v is first on *send*
- Effect:
  - remove first element of *send*

$leader_i$
- Precondition:
  - *status* = *chosen*
- Effect:
  - *status* := *reported*

$receive(v)_{i-1,i}$
- Effect:
  - case
    - $v > u$: add v to *send*
    - $v = u$: *status* := *chosen*
    - $v < u$: do nothing
  - endcase

Tasks:

{$send(v)_{i,i+1}$: v a UID}
{$leader_i$}

477

转移应该是显而易见的。进程i负责执行两个任务：发送消息给进程i+1和宣布自己是领导者。这样它就具有两个任务：一个针对所有的$send$动作，另一个针对它的$leader$动作。

*AsynchLCR*的行为基本上和*LCR*的行为相同，但可能在时间上是“歪斜的”。

为了证明*AsynchLCR*能解决领导者选举问题，如同同步*LCR*算法中的做法，我们也使用不变式断言。不变式断言证明在同步网络中的使用与在异步网络中的使用大体一样；主要的区别是现在必须以更细粒度，对单个事件而不是轮进行推导。

技术上，为了应用不变式断言证明，必须知道每个通道自动机的状态的结构。为了方便起见，假设通道$C_{i,i+1}$都是如14.4.1节中定义的通用可靠FIFO通道。然后我们知道每一$C_{i,i+1}$的状态都是由单个*queue*组件组成的，我们称这个组件为$queue_{i,i+1}$。这个假设并没有限制结论的通用性，因为一个对通用可靠FIFO通道工作正常的算法对任意可靠FIFO通道也必然工作正常。在对具有可靠FIFO通道的发送/接收系统的正确性证明中，我们将做出同样的假设。

令i_{max}表示最大UID进程的下标，u_{max}表示该UID。这里和同步情况一样，我们要证明两点：

1）除了i_{max}之外没有其他进程执行*leader*输出；

2）进程i_{max}最终执行*leader*输出；

两个条件中的第一个是安全属性，而第二个是活性属性。

引理15.1　除了i_{max}之外没有其他进程执行*leader*输出。

证明　我们使用和同步情况中断言3.3.3类似的不变式。断言3.3.3说没有UID v 能够到达处于i_{max}和v的原始位置i之间的任意*send*队列。现在，由于*AsynchLCR*算法包含通道自动机，因此我们需要一个略强的断言，它不仅涉及进程状态中的UID，还涉及通道状态中的UID。通常，把进程索引作为进程状态组件的下标；并把通道的双索引作为通道状态组件的下标。

断言15.1.1　下列情况在任何可达状态中成立：

478

1）如果 $i \neq i_{max}$并且$j \in [i_{max}, i)$，则u_i在$send_j$中不出现。

2）如果 $i \neq i_{max}$ 并且$j \in [i_{max}, i)$，则u_i在$queue_{j,j+1}$中不出现。

断言15.1.1可以通过对导致给定状态的一个有限运行的步数进行归纳来证明。该证明大体上与断言3.3.3的证明相同。这次，我们基于*send*、*receive*和*leader*事件来进行分情况讨论。关键是$receive(v)_{j-1,j}$事件的情况，其中$j=i_{max}$；此时，我们必须证明如果$v=u_j$成立，其中$i \neq i_{max}$，则v被丢弃。

断言15.1.1可用来证明断言15.1.2。

断言15.1.2　下列情况在任何可达状态中成立：

如果$i \neq i_{max}$，则$status_i = unknown$。

由于这个动作的前提条件永远得不到满足，因此容易看出除了i_{max}之外没有其他进程执行一个$leader_i$输出。 □

现在我们来看活性属性。注意，需要假设*AsynchLCR*的运行是公平的。这个形式化概念意味着进程和通道继续执行工作。

引理 15.2　在任意公平运行中，进程i_{max}最终输出*leader*。

证明　*AsynchLCR*的这一属性的证明与同步*LCR*算法对相关结果的证明（引理3.2）区别很大。在同步情况下，我们使用一个非常强的不变式断言——断言3.3.2——来精确描述在任意r轮之后最大UID处于环中的位置。现在我们没有轮的概念。并且，在计算中也不可能精确刻画计算中发生了什么，因为异步性引入了很大的不确定性。所以我们必须使用不同的方法。

我们的证明基于建立中间里程碑来达到选出领导者的主要目的。特别地，我们对r（其中：

$0 \leqslant r \leqslant n-1$)进行归纳，证明$u_{max}$最终出现在缓冲区$send_{i_{max}+r}$中。对$r = n-1$应用该结论，我们证明$u_{max}$最终被安置在通道$C_{i_{max}-1,i_{max}}$中，然后$u_{max}$最终被进程$i_{max}$接收，之后进程$i_{max}$最终输出领导者。在证明这些最终结论时用到了进程I/O自动机和通道I/O自动机的公平性属性。

例如，考虑在一个公平运行α中的状态s，其中在α中任一UID v出现于$send_i$缓冲区的头部。我们来证明$send(v)_i$最终会出现。如果不出现，那么进程$AsynchLCR_i$的转移检测将显示v永远保留在$send_i$缓冲区的头部。这也意味着任务$send_i$永远使能，根据公平性可知，之后一些$send_i$事 479
件必然发生。但由于v是在$send_i$缓冲区头部的消息，因此这意味着$send(v)_i$最终也会发生。

另外，如果对于任一$k>1$，v出现在$send_i$缓冲区的第k个位置，则可以证明$send(v)_i$最终发生。这可以通过对k的归纳来完成，其中基础$k = 1$的情况已在上面给出。对于归纳步，当缓冲区的头部被移去时，在位置$k>1$上的UID v最终到达位置$k-1$，然后根据归纳假设知$send(v)_i$最终发生。 □

可对通道中UID做类似证明。

将这些证明放到一起，我们得到：

定理15.3 *AsynchLCR*解决了领导者选举问题。

接下来我们考虑*AsynchLCR*算法的复杂度。和同步*LCR*算法一样，消息的数量是$O(n^2)$。*LCR*的时间界限是n轮。对*AsynchLCR*进行时间分析时，假定每一进程的每一任务的时间上界为l，在每一通道队列中传递最早消息的时间上界为d。

通过将时间界限整合进引理15.2的证明中，可以得到一个粗略的时间界限$O(n^2(l+d))$。也就是说，每一进程的发送缓冲区的最大长度和每一通道队列的最大长度都为n。所以，将进程的发送缓冲区中的UID放在相邻通道中最多需要时间nl，而在通道队列中的UID被下一进程接收的话最多需要时间nd。因此，总共的时间复杂度是$O(n^2(l+d))$。

然而，可以进行更加精确的分析，得出时间上界仅为$O(n(l+d))$。关键在于，虽然一些发送缓冲区和队列的长度可以达到n，但是不可能每个缓冲区和队列的长度都为n。要形成堆积的话，一些UID的走动速度必须比最坏情况上界所规定的速度要快，这样才能赶上其他UID。这样所用的时间不比当UID以相同速度来走动时的时间多。我们得到：

引理15.4 在任意公平运行中，对于任意r（其中：$0 \leqslant r \leqslant n-1$）和对于任意$i$,以下结论成立：

1）到时间$r(l+d)$为止，UID u_i要么到达$send_{i+r}$缓冲区，要么被删除。

2）到时间$r(l+d)+l$为止，UID u_i要么到达队列$queue_{i+r,i+r+1}$，要么被删除。

证明 对r进行归纳。 480

基础：$r=0$。UID在$send_i$中开始，在时间l之内被放在$queue_{i,i+1}$中，得证。

归纳步：假设引理对$r-1$成立，现要证明对r也成立。固定任意一个i。对于第一点来说，假设u_i到时间$r(l+d)$为止还没有被删除。这样，归纳假设意味着到时间$t=(r-1)(l+d)+l$为止，UID u_i到达$queue_{i+r-1,i+r}$。

声明 15.5 如果u_i到时间t为止没有被传送到进程$i+r$，那么u_i到时间t为止已经到达$queue_{i+r-1,i+r}$的头部。

证明 采用反证法。假设u_i到时间t为止还没有被传送到进程$i+r$，u_i到时间t为止也没有到达$queue_{i+r-1,i+r}$的头部。那么一定存在另一UID，u_j，在t时的$queue_{i+r-1,i+r}$中，u_j在u_i之前。这是一个堆积，其中u_i追上u_j；既然u_i在环中的走过距离小于r，那么u_j在环中的走过距离小于$r-1$。

然而，归纳假设意味着到时间$(r-1)(l+d)<t$为止，u_j要么到达$send_{j+r-1}$（即穿越距离至少为$r-1$），要么被删除。这也意味着在t时u_j不可能仍在$queue_{i+r-1,i+r}$中，这是一个矛盾。 □

因此到时间t为止，u_i要么被传送到进程$i+r$，要么已到达$queue_{i+r-1,i+r}$的头部。在后一种情况下，在额外时间d内，u_i被传送到进程$i+r$。在任何一种情况中，u_i在时间$t+d=r(l+d)$内被传送到进程$i+r$，并被放在$send_{i+r}$缓冲区中。

第二点的证明与之类似。

定理15.6　在*AsynchLCR*的任意公平运行中，直到领导者事件出现为止的所需时间最多为$n(l+d)+l$，或$O(n(l+d))$。

证明　当$r=n-1$时，引理15.4意味着在时间$(n-1)(l+d)+l$之内UID u_{max}到达$queue_{i_{max-1},i_{max}}$，采用类似于引理15.4的证明的方法，可以证明此时它到达此队列的头部。这样在额外时间d内，u_{max}被传送到进程i_{max}，然后进程i_{max}在额外时间l内输出*leader*。共花时间$n(l+d)+l$，得证。 □

唤醒　我们可以修改领导者选举问题的输入/输出约定，使得输入（这里是UID）通过特定消息
481 *wakeup*$(v)_i$而不是初始状态来到达进程，这些信息来自外部用户U。正确性条件应做修改，使得对于每个i都恰好有一个*wakeup*$(v)_i$。这样就很容易地修改*AsynchLCR*算法以满足新的正确性条件：每一进程P_i推迟执行任意局部控制动作，直到接收到其*wakeup*信息为止。如果P_i在收到它的*wakeup*之前接收到任何消息，则将这些消息放在一个新的接收缓冲区中，在收到*wakeup*后再对它们进行处理。

对本节之后的领导者选举算法也可做类似修改。更一般地，任何在初始状态中表示输入的分布式问题都可以转换成为在消息*wakeup*中表示输入的等价问题。使用上述的策略，我们对任何解决了原来问题的算法进行修改，使之满足新的正确性条件。

15.1.2 *HS*算法

在3.4节的同步*HS*算法中，每一进程都不断地向两个方向发送试探消息，并不断倍增尝试的距离（如果成功的话）。显而易见，用进程I/O自动机重写后，这一算法在异步网络模型中依旧工作正常。和前面一样，它的通信复杂度也是$O(n \log n)$。时间复杂度上界的确定则留做一道习题。

15.1.3 *PetersonLeader*算法

*HS*算法（不论同步还是异步）需要$O(n \log n)$条消息并使用双向通信。本小节介绍*PetersonLeader*算法，它只使用单向通信，通信复杂度是$O(n \log n)$。这一算法不依赖于环中节点数目n。它仅仅对UID进行比较。它选出任意一个进程作为领导者，而不必选出具有最大或最小UID的进程。$O(n \log n)$的通信复杂度只有一个很小的常数因子（约为2）。

***PetersonLeader*算法（非形式化）：**

算法运行时，每一进程都被指定为处于*活动*状态或者*转发*状态。起始时所有进程都是活动的。活动进程执行算法的“实质性工作”；转发进程只是传递消息。*PetersonLeader*算法的运行分为若干（异步决定的）阶段。在每个阶段中，活动进程的数目至少减半，因此阶段数最多
482 为$\log n$。

在算法的第一个阶段中，每一进程i都将它的UID顺时针发送两步。然后进程i将它的UID和逆时针方向的两个前驱进程的UID进行比较。如果逆时针邻居的UID是三者中最大者，即如果$u_{i-1}>u_{i-2}$且$u_{i-1}>u_i$，那么进程i仍处于活动状态，并采用逆时针方向邻居的UID u_{i-1}作为新的“暂时UID”。另一方面，如果另外两个UID中的一个是三者中的最大者，则进程i简单地变为运行的

余下部分的转发者。

每个后续阶段都以相同方式前进。现在每个活动进程i都把暂时UID发送给顺时针方向的下一个和再下一个活动进程，然后等待逆时针方向的两个前驱活动进程发来暂时UID。如果第一个前驱活动进程的UID是三个UID中的最大值，则进程i仍保持活动状态，并采用前驱UID作为新的暂时UID。另一方面，如果另两个UID中的一个是三者中的最大者，则进程i变为转发状态。

另外，如果在任何阶段中，进程i发现接收到的从直接前驱发来的UID等于自己的UID，则i知道它是惟一剩下的活动进程。在这种情况下，进程i将自己选为领导者。

显而易见，在任何拥有一个以上活动进程的阶段中，至少有一个进程会发现一个UID组合，这个组合允许它在下一阶段中保持活动状态。并且，顶多有一半的UID可以在一个给定阶段中生存下来，因为每个保持活动状态的进程都必然有一个变成转发者的直接活动前驱。

PetersonLeader$_i$ automaton (formal):

Signature:

Input:
- *receive*$(v)_{i-1,i}$, v a UID

Output:
- *send*$(v)_{i,i+1}$, v a UID
- *leader*$_i$

Internal:
- *get-second-uid*$_i$
- *get-third-uid*$_i$
- *advance-phase*$_i$
- *become-relay*$_i$
- *relay*$_i$

States:

mode $\in \{active, relay\}$, initially *active*
status $\in \{unknown, chosen, reported\}$, initially *unknown*
uid(j), $j \in \{1,2,3\}$, each a UID or *null*; initially *uid*$(1) = i$'s UID, *uid*(2) = *uid*(3) = *null* 483
send, a FIFO queue of UIDs, initially containing i's UID
receive, a FIFO queue of UIDs, initially empty

Transitions:

get-second-uid$_i$
- Precondition:
  - *mode* = *active*
  - *receive* is nonempty
  - *uid*(2) = *null*
- Effect:
  - *uid*(2) := first element of *receive*
  - remove first element of *receive*
  - add *uid*(2) to *send*
  - if *uid*(2) = *uid*(1) then *status* := *chosen*

get-third-uid$_i$
- Precondition:
  - *mode* = *active*
  - *receive* is nonempty
  - *uid*$(2) \neq$ *null*
  - *uid*(3) = *null*
- Effect:
  - *uid*(3) := first element of *receive*
  - remove first element of *receive*

become-relay$_i$
- Precondition:
  - *mode* = *active*
  - *uid*$(3) \neq$ *null*
  - *uid*$(2) \leq max\{$*uid*(1), *uid*$(3)\}$
- Effect:
  - *mode* := *relay*

relay$_i$
- Precondition:
  - *mode* = *relay*
  - *receive* is nonempty
- Effect:
  - move first element of *receive* to *send*

leader$_i$
- Precondition:
  - *status* = *chosen*
- Effect:
  - *status* := *reported*

advance-phase$_i$
　Precondition:
　　mode = *active*
　　uid(3) ≠ *null*
　　uid(2) > *max*{*uid*(1), *uid*(3)}
　Effect:
　　uid(1) := *uid*(2)
　　uid(2) := *null*
　　uid(3) := *null*
　　add *uid*(1) to *send*

send$(v)_i$
　Precondition:
　　v is first on *send*
　Effect:
　　remove first element of *send*

receive$_i(v)$
　Effect:
　　add v to *receive*

Tasks:
{*send*$(v)_{i,i+1}$: v is a UID}
{*get-second-uid*$_i$, *get-third-uid*$_i$, *advance-phase*$_i$, *become-relay*$_i$, *relay*$_i$}
{*leader*$_i$}

484

定理15.7　*PetersonLeader* 解决了领导者选举问题。

现在我们来分析复杂度。如前所述，每一阶段中的活动进程数目至少减半，直到只剩下一个活动进程为止。这意味着：到领导者被选出时，阶段总数最多为 $\lfloor \log n \rfloor + 1$。在每个阶段中，每个进程（活动的或转发的）最多发送两个消息。所以，在算法的任一运行中，最多 $2n(\lfloor \log n \rfloor + 1)$个消息被发送。复杂度为$O(n \log n)$，它有一个比*HS*算法中好得多的常数因子。

关于时间复杂度，不难证明一个未证明过的上界$O(n \log n(l+d))$。这是因为有$O(\log n)$个阶段，而且我们可以证明对于任意p，前p个阶段在时间$O(pn(l+d))$内完成。（在每个阶段中，每个UID沿着环走动$O(n)$的距离。如果一个信息没有被堆积阻塞，则它从一个节点到达另一节点至多花费$l+d$的时间。引理15.4证明中的方法也可以用来证明堆积不会损害最坏情况的界限。）

进行更加精确的分析后，可以得出一个上界$O(n(l+d))$：

定理15.8　在*PetersonLeader*算法的任意公平运行中，直到领导者事件产生为止所需的时间最多为$O(n(l+d))$。

这里只给出主要思路，具体证明留做一道稍微复杂的习题。

证明概要　首先，我们可以忽略堆积，因为引理15.4的证明中的方法可以用来证明堆积不会影响最坏情况的界限。下面的声明对分析是有用的。

声明 15.9　如果进程i和j在阶段p中都是活动的不同进程，那么一定存在某个进程k，使得从顺时针方向看k严格地在i之后和在j之前，且进程k在阶段$p-1$中是活动的。

时间复杂度正比于特定消息链的长度。这个链的最后一个消息是在最后阶段p中导致i_p被选成领导者的那个消息。这个信息中的UID源自阶段p中的i_p，且在阶段p时它已经走动的总距离为n。进程i_p在进入阶段p时启动该UID，这刚好在i_p在阶段$p-1$接收到*uid*(3)之后。而这个*uid*(3)则源自当i_{p-1}进入阶段$p-1$时i_p的第二个前驱i_{p-1}，这个前驱在阶段$p-1$中是活动的。由声明15.9可知，某个不是i_p的进程到达阶段$p-1$，这意味着在阶段$p-1$时这个UID能够走动的最大可能
485 距离是n。

我们继续沿着这条链来进行回溯。当进程i_{p-1}在阶段$p-2$收到它的*uid*(3)的时候，它进入阶段$p-1$。而这个*uid*(3)则源自当i_{p-2}进入阶段$p-2$时i_{p-1}的第二个前驱i_{p-2}，且这个前驱在阶段$p-2$中是活动的。声明15.9可以用来证明：相对i_{p-1}的位置，i_{p-2}不比i_{p-1}在阶段$p-1$的第一个活动前驱

更靠后。继续回溯以定义$i_{p-3},\ldots,i_1$，使得：相对i_q的位置，每个i_{q-1}不比i_q在阶段q的第一个活动前驱更靠后。

现在重复使用声明15.9，可以说明从i_{p-1}退回到i_1的链的总长最多为n。这意味着链的总长为$3n$，转换为时间界限的话就是$O(n(l+d))$。 □

15.1.4 通信复杂度的下界

我们描述两种异步网络领导者选举算法：*PetersonLeader*和*HS*的异步版本，它们的通信复杂度是$O(n \log n)$。在本节中，我们将证明这一问题具有更低的下界$\Omega(n \log n)$。为了保持一般性，本节我们假设通道是通用可靠FIFO通道。

我们已经在定理3.9和定理3.11中给出针对同步环境中领导者选举问题的两个$\Omega(n \log n)$下界结果。定理3.9给出基于比较算法的下界；它允许双向通信，允许进程知道网络中节点的个数。这一结果可以直接用于异步环境，因为可以将同步模型视为异步模型的一种限制模型。

定理15.10 设A是在大小为n的异步环网络中选出领导者的基于比较的算法，其中通信是双向的，且n为进程所知。那么存在A的一个公平运行，直到领导者被选出为止，其中$\Omega(n \log n)$条消息被发送。

定理3.11给出某类算法的下界，这类算法以任意方式来使用UID，但具有固定时间界限和一个大的标识符空间。另外，它允许双向通信，允许进程知道节点数目。我们也将此结果的一个版本应用于异步环境。

定理15.11 令A是在大小为n的异步环中选出领导者的任意（不一定是基于比较的）算法，其中UID的空间是无穷大的，通信是双向的，且n为进程所知。那么存在A的一个公平运行，其中直至领导者被选出为止，其中$\Omega(n \log n)$条消息被发送。 486

证明概要 如果存在A的任意运行，直到领导者被选出为止，其中多于$n \log n$条消息被发送，则证明完成，所以我们假设不存在这种情况。我们“限制”A来产生一个同步算法S，其中在每轮中都有某个消息被发送。既然在任意A的公平运行中，直到领导者被选出为止，顶多有$n \log n$条消息被发送，那么S选出领导者的所需轮数最多为$n \log n$。由于UID的空间是无穷大的，因此定理3.11就可以用来证明存在S的一个运行，其中直到领导者被选出为止共有$\Omega(n \log n)$条消息被发送。这可以转化成A的一个公平运行，其中直到领导者被选出为止共有$\Omega(n \log n)$条消息被发送。 □

由于定理3.11在本书的起始章中出现，因此我们为非基于比较的算法给出另一个更基本的下界证明。这个证明跟定理3.9和定理3.11的证明区别很大，因为它是基于异步的，并且假设进程不知道环的大小。

定理15.12 设A为在任意大小的环中选出领导者的任意（不一定是基于比较的）算法，其中UID的空间是无穷大的，通信是双向的，进程不知环的大小。那么存在A的一个公平运行，其中$\Omega(n \log n)$条消息被发送。

证明需要一些基本定义。假设我们有一个通用无穷进程自动机集$\mathcal{P}$。$\mathcal{P}$中的所有进程除了UID之外都是相同的；另外，假定它们只根据本地名字（即“右”和“左”）来知道它们的邻居。

我们的主要兴趣在于观察当$\mathcal{P}$中的一组进程自动机被放在环中时它们是如何运作的；然而，观察它们被放在一条直线上时的行为也是有用的，见图15-2。线被定义为$\mathcal{P}$中不同进程的线性合成（采用I/O自动机合成），其中每个进程都与两个方向上的可靠FIFO发送/接收双向通道相关。 487

图15-2　自动机线

如果两条线不包含相同的进程自动机，即不包含相同的UID，则称它们是*互不相交的*。如果L和M是两个互不相交的自动机线，定义*join*（L，M）为由L和M连接而成的线，其中新的可靠FIFO发送/接收通道插在L的最右端进程和M的最左端进程之间。*join*操作是可结合的，我们可以把它扩展到任意数目的线中去。如果L是任意一条自动机线，则我们定义*ring*（L）为将L围住的环，其中新的可靠FIFO发送/接收通道插在L的最右端和最左端进程之间。线中每个进程的右邻居变为它在环中的顺时针方向的邻居。*ring*和*join*操作如图15-3所示。（我们现在将通道仅仅表示为箭头，而不是椭圆。）

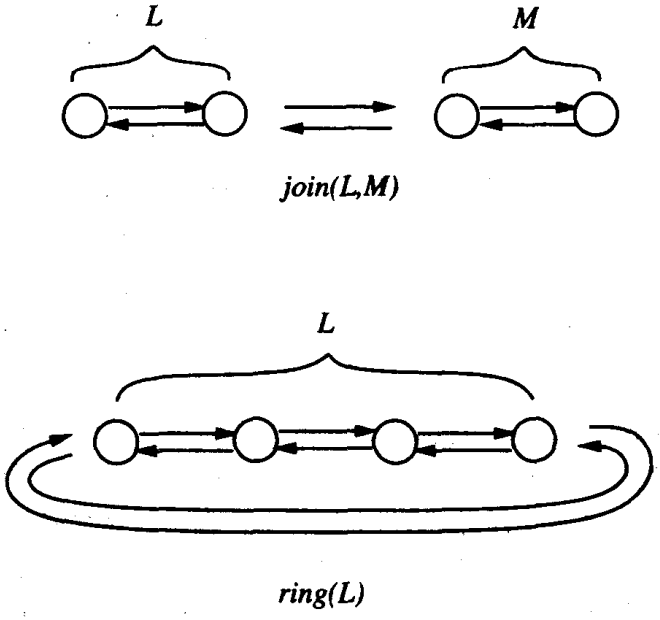


图15-3　*join*和*ring*操作

如果α是线或环中的一个运行，则我们定义$C(\alpha)$为α中发送的消息的数目。如果R是环，则我们定义$C(R)$为$sup\{C(\alpha)$：α是R的一个运行$\}$，即在R的任意运行中被发送的消息的数目上界。对于线，我们考虑当线“孤立地”工作、没有消息从线外环境来到末端进程时发送消息的数目。
488 因此，如果L是线，则我们定义$C(L)$为$sup\{C(\alpha)$：α为L的一个无输入运行$\}$，即在L的任意运行中被发送的消息的数目上界，其中在这些运行中没有信息从线之外的地方来到L的末端。

对于状态s，如果不存在以s开始的、且有新消息被发送的运行片段，则称环的状态s是*安静的*。如果不存在以s开始的、且有新消息被发送的无输入运行片段，则称线的状态s是*安静的*。注意，如果环或线处于安静状态，则并不意味着不可能出现进一步的活动——它仅仅意味着不会发生进一步的消息发送事件。进程仍旧可能收到消息并且执行内部步和*领导者*输出。

我们以一个初步引理开始。

引理15.13　存在一个由$\mathcal{P}$中进程自动机组成的无穷集，其中每个进程都可以在没有收到任何消息之前发送至少一条消息。

证明 我们给出更强的东西：$\mathcal{P}$中所有进程自动机（可能一个除外）在没有收到任何消息之前都可发送至少一条消息。

为了得到矛盾，假设$\mathcal{P}$中存在两个进程i和j，它们都只有在收到消息之后才能发送消息。然后考虑图15-4中所示的三个环R_1、R_2和R_3。（现在，为了简单起见，我们不再描绘通道自动机）。

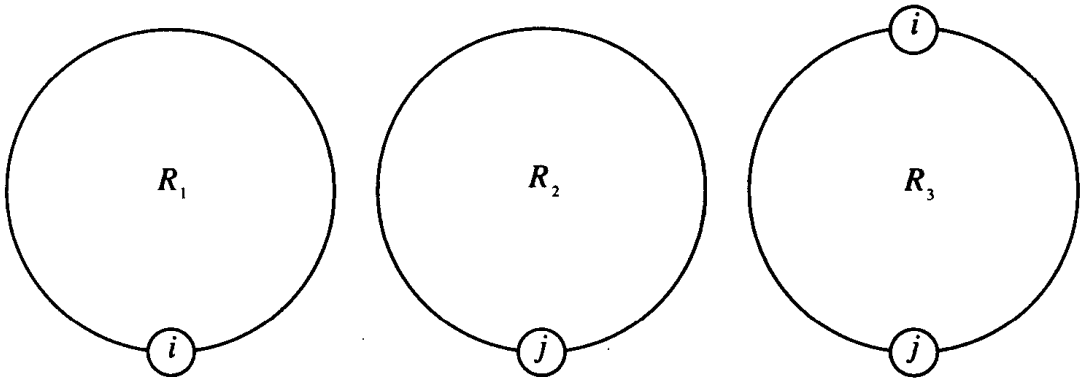


图15-4 引理15.3的证明中的环R_1、R_2和R_3

既然i和j只有在收到消息之后才能发送消息，那么在这三个环的任何运行中都没有任何消息被发送。因此，进程i和j独立运行，执行局部计算和*leader*动作，但是没有执行任何通信动作。既然R_1可解决领导者选举问题，那么i最终在R_1的任何公平运行中都选出领导者。同样地，既然R_2可解决领导者选举问题，那么j最终在R_2的任何公平运行中都选出领导者。现在考虑R_3的 489
任意公平运行α。因为没有通信，α和R_1的某个公平运行对于进程i是不可区分的（用到8.7节中正式定义的“不可区分”概念），所以，i最终在α里选出领导者。同样地，α和R_2的某个公平运行对于进程j是不可区分的。所以，j最终在α里选出领导者。但是这使得R_3中选出两个领导者，矛盾。

我们已经证明$\mathcal{P}$中不可能出现两个进程i和j，它们只能在收到消息之后才能发送消息。也就是说，$\mathcal{P}$中最多只有一个进程不能在收到消息之前发送消息。既然$\mathcal{P}$是个无穷集，从中去掉一个进程仍旧是进程的无穷集，其中每个进程都可以在没有收到消息之前发送消息。 □

定理15.12的证明用到下面的关键引理。

引理15.14 对所有$r \geq 0$，存在一个由互不相交的线组成的无穷集合$\mathcal{L}_r$，使得对每个$L \in \mathcal{L}_r$，$|L| = 2^r$和$C(L) \geq r2^{r-2}$成立。

证明 对r进行归纳。

基础：$r = 0$。令$\mathcal{L}_0$为由所有与P中所有进程对应的单节点线组成的集合。结果显而易见。

基础：$r = 1$。令$\mathcal{L}_1$为由互不相交的双节点线组成的任意无穷集合，其中双节点线由那些能够在事先没有收到消息的情况下发送消息的进程组成。引理15.13已经暗示这一集合的存在。那么如果L是$\mathcal{L}_1$中的任意线，则一定存在L的无输入运行，其中至少一条消息被发送：简单地令两个进程中的一个在事先没有收到消息的情况下发送一条消息。得证。

归纳步：假设$r \geq 2$并且该引理对$r-1$成立，即存在由互不相交线组成的无穷集合$\mathcal{L}_{r-1}$，使得对于所有$L \in \mathcal{L}_{r-1}$，$|L| = 2^{r-1}$和$C(L) \geq (r-1)2^{r-3}$成立。令$n = 2^r$。

令L、M和N为$\mathcal{L}_{r-1}$中的任意三条线，我们考虑由这三条线中的两条线组成的六种可能连接：$join(L, M)$、$join(M, L)$、$join(L, N)$、$join(N, L)$、$join(M, N)$和$join(N, M)$。我们来证明以下声明。□

声明15.15　这六条线中至少有一条拥有其中至少 $\frac{n}{4}\log n = r2^{r-2}$ 条消息被发送的无输入运行。

上面的引理可以从声明15.15中导出，因为可以从 $\mathcal{L}_{r-1}$ 的进程中选取无穷多个三线集合，其
490 中没有进程被重用。

（声明15.15的）证明　假设声明不成立，六条线中没有一个可以发送多至 $\frac{n}{4}\log n$ 条消息。根据归纳假设，存在一个满足 $C(\alpha_L) \geqslant (r-1)2^{r-3} = \frac{n}{8}\log\frac{n}{2}$ 的 L 的有穷无输入运行 α_L。为了保持一般性，我们可以假定 α_L 的最终状态是安静的，否则 α_L 可以被扩展为更长的有穷运行，其中更多的消息被产生。（这一扩展不能无止境继续下去，因为我们知道 L 自己不能发送出 $\frac{n}{4}\log n$ 条消息。）类似地，我们获得具有同样性质的 M 的有穷无输入运行 α_M 和 N 的有穷无输入运行 α_N。

现在我们构造线 $join(L, M)$ 的一个有穷运行 $\alpha_{L,M}$。运行 $\alpha_{L,M}$ 以在 L 上运行 α_L 和在 M 上运行 α_M 开始，把那些发送到线 L 和 M 之间通道上的所有消息进行延迟。在 $\alpha_{L,M}$ 的这个前缀中，至少有 $2\left(\frac{n}{8}\right)\log\frac{n}{2} = \frac{n}{4}(\log n - 1)$ 条消息被发送。

接下来，$\alpha_{L,M}$ 仍保持安静状态。注意，无论这一过程如何发生，在扩展中发送的额外消息条数一定严格小于 $\frac{n}{4}$，不然的话，$\alpha_{L,M}$ 中的消息总数至少为 $\frac{n}{4}\log n$，与我们的假设矛盾。

我们做这一扩展的具体方法是：只允许 L 中离 L 和 M 的连接点最近的 $\frac{n}{4}-1$ 个进程和 M 的 $\frac{n}{4}-1$ 个进程在 α_L 和 α_M 后执行步，直到系统到达一个状态，从这个状态开始这些进程都不能再发送任何消息。我们声明 $join(L, M)$ 的所得状态一定是安静的。因为不然的话，在初始的 α_L 和 α_M 后会有至少 $\frac{n}{4}-1$ 条消息被发送，这些信息会将与连接点有关的信息带到距离连接点 $\frac{n}{4}$ 处的地方，并使得进程发送另一条信息。（说服你自己这一点是正确的。）但是这使得在 α_L 和 α_M 的扩展中至少 $\frac{n}{4}$ 条额外消息被发送，这是不可能的。所以说 $join(L, M)$ 的状态一定是安静的。

简要地讲，在 $\alpha_{L,M}$ 之后，有关线 L 和 M 的连接点的信息既没有到达 L 的中点，也没有到达 M 的中点。连接点的每边都只有距离在 $\frac{n}{4}$ 之内的进程知道连接点的信息，并且在刚好距离连接点 $\frac{n}{4}$ 处的两个进程知道此信息后，不会再发送任何新消息。图15-5描述 n=16 的情况下 L 和 M 的连接点。

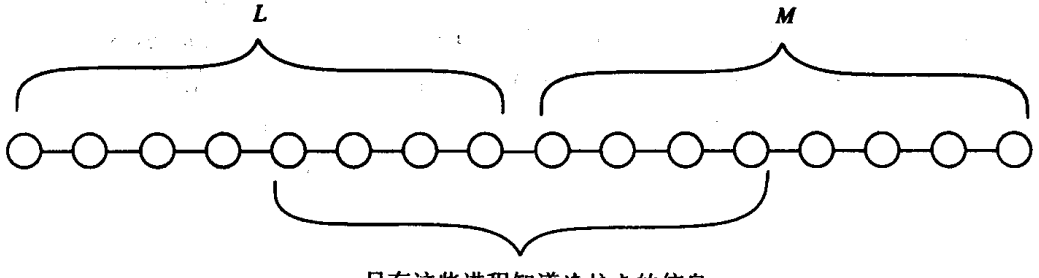


图15-5　$\alpha_{L,M}$

用类似的方法，我们定义有穷运行$\alpha_{M.L}$、$\alpha_{L.N}$等。

现在我们将L、M和N合成为几个不同的环以得到矛盾。首先，定义R_1为$ring(join(L, M, N))$，如图15-6所示。定义R_1的一个公平运行α_1如下。运行α_1从α_L、α_M和α_N开始，因此使得三个独立线L、M和N都处于安静状态。然后α_1以$\alpha_{L.M}$、$\alpha_{M.N}$和$\alpha_{N.L}$继续。由于知道每个连接点的进程最多扩展到离连接线一半距离的地方，因此在这三个运行中没有相互干扰。而且，在这三个运行之后，整个环处于安静状态。然后α_1以任何公平形式继续。正确性条件意味着在α_1中某一领导者i_1被选出。为了保持一般性，我们假设进程i_1在L的中点和M的中点之间，如图15-6所示。 491

接下来定义$R_2=ring(join(L, N, M))$，并且定义类似于α_1的R_2的一个公平运行α_2（这一次使用α_L、α_M、α_N、$\alpha_{L.N}$、$\alpha_{N.M}$和$\alpha_{M.L}$）。然后在α_2中选出某一领导者i_2（如图15-7所示）。

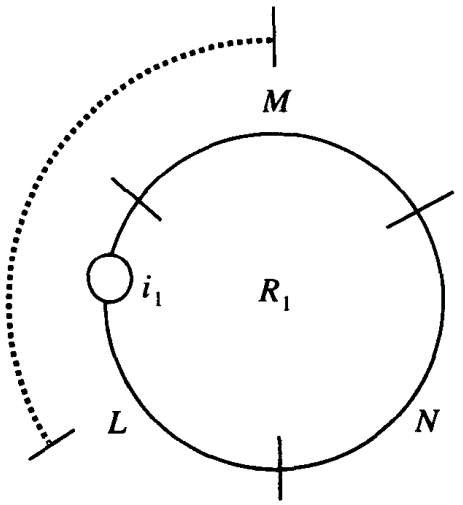


图15-6 $R_1 = ring(join(L, M, N))$

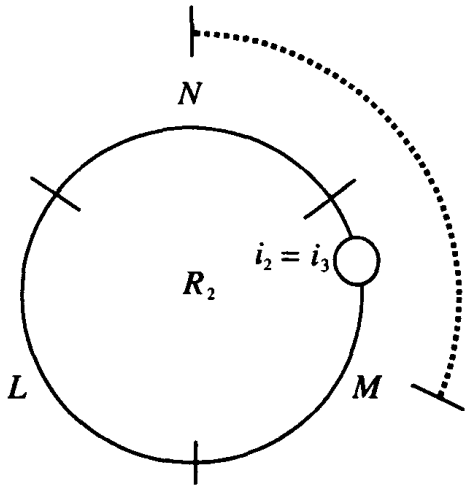


图15-7 $R_2=ring(join(L, N, M))$

接下来定义$R_3 = ring(join(M, N))$，并且定义R_3的一个公平运行α_3（使用α_M、α_N、$\alpha_{M.N}$和$\alpha_{N.M}$）。在α_3上一定可以选出某个领导者i_3(如图15-8所示)。我们声称i_3一定在R_3的下半圈，如图15-8所示，即在M的中点和N的中点之间（按顺时针方向）。因为如果i_3在R_3的上半圈，那么α_1和α_3对于进程i_3是不可区分的，这样i_3也会在α_1中被选出。但是这样的话两个不同进程i_1和i_3都在α_1中被选出，矛盾。（进程i_1和i_3是不同的，因为i_1在L和M的中点之间，而i_3在M和N的中点之间。） 492

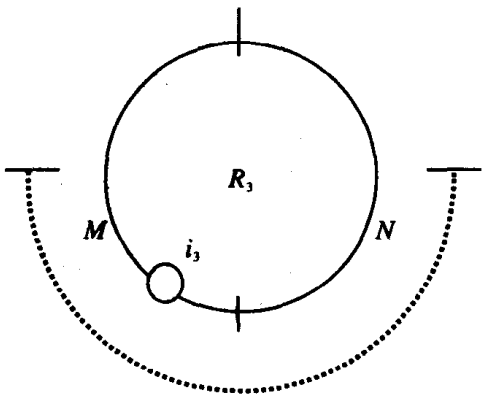


图15-8 $R_3 = ring(join(M, N))$

由于i_3在R_3的下半圈，α_2和α_3对i_3来说是不可区分的，因此i_3在R_2中也被选出。注意，i_3在R_2中N的中点和M的中点之间。既然α_2中只能选出一个领导者，那么我们得出$i_2=i_3$。见图15-7。

最后，我们定义$R_4=ring(join(L, N))$，并且定义R_4的一个公平运行α_4（用α_L、α_N、$\alpha_{L.N}$和$\alpha_{N.L}$）。见图15-9。我们说在α_4中没有领导者被选出。因为如果领导者在R_4的上半圈中被选出，那么该领导者也会被α_2选出，这样在α_2中就产生了两个领导者。如果领导者在R_4的下半圈中被选出，那么该领导者也会被α_1选中，在α_1中就产生了两个领导者。两种情况都产生矛盾。

但是，在α_4中没有选出领导者这一事实违反了问题要求，从而产生证明该声明所需的矛盾。□

正如以前声明的证明所描述的，根据声明15.15，引理
493 得证。

现在使用引理15.14，可以轻易地完成定理15.12的证明。

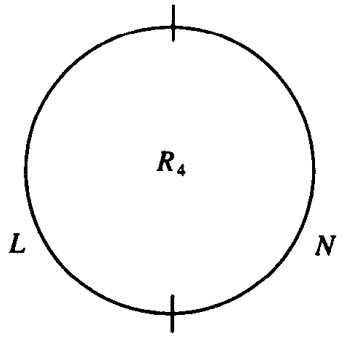


图15-9 R_4=ring(join(L, N))

（定理15.12的）证明　首先假设n是2的幂，如$n = 2^r$。令L为$\mathcal{L}$中的任意一条线。引理15.14意味着$|L| = n$和$C(L) \geq \frac{n}{4}\log n$。令$\alpha$为满足$C(\alpha) \geq \frac{n}{4}\log n$的$L$的一个无输入运行。定义$R = ring(L)$，也就是说，将$L$粘贴到一个环中。定义$R$的一个运行$\alpha'$，它运作得像和$L$上的$\alpha$一样，它将经过$L$的末端之间的连接点的所有信息进行延迟，直到看见最少$\frac{n}{4}\log n$条信息为止。然后$C(\alpha') \geq \frac{n}{4}\log n$，这证明了$C(R) \geq \frac{n}{4}\log n$。

494 对于n不为2的幂的情况，具体证明留做一道习题。 □

注意定理15.12证明中的关键部分在于对异步性和未知环大小的处理。

15.2 任意网络中的领导者选举

本节到此为止已经考虑了在异步环网络中选举领导者的算法。现在我们将考虑在基于更加通用的图网络中的领导者选举问题。在本节中我们假定底层图是无向的，即可以在所有边上进行双向通信，并且图是连通的。假定进程除了UID之外都是相同的。

考虑4.1.2节中针对同步网络的*FloodMax*算法。它要求进程知道网络的直径*diam*。在这个算法中，每个进程保存着到目前为止它所见到的最大UID。在每一同步轮中，进程在所有通道中发送该最大值。算法在*diam*轮后结束；其UID等于最大UID的惟一进程宣布自己为领导者。

*FloodMax*算法不能直接扩展到异步环境中，因为在异步模型中没有轮。但是，有可能异步地模拟轮。我们只要求每个发送第r轮消息的进程将轮号r标到消息上。接收者在执行第r轮转移之前，等待来自它的所有邻居的第r轮消息。通过模拟*diam*轮，算法可以正确终止。

在同步环境中，我们描述一个*FloodMax*的优化算法，名为*OptFloodMax*算法，其中每个进程只在有新信息时才发送信息，即在它的最大UID发生变化时才发送消息。如何在异步网络中模拟这个优化版本还不是很清楚。如果与*FloodMax*一样仅仅将轮号标于消息上，那么对于一个在第r轮中没有收到所有邻居的消息的进程来说，它便不能确定何时收到所有第r轮的流入消息，这样它不会知道应在何时执行第r轮转移。当然我们可以在不相互通信的邻居对之间加入*dummy*消息，但这样会破坏最优性。

或者，我们纯异步地模拟*OptFloodMax*——只要进程得到新的最大UID，它就在以后某个
495 时间发送这个UID给它的邻居。这一策略最终会将最大值传给所有进程。但是有个问题：现在进程无法知道应在何时停止。

使用以后章节中的许多技术，我们可以开发出许多针对通用异步网络中领导者选举问题的解法。这些技术包括：

1）基于搜索算法的异步广播和敛播（见15.3节）
2）利用生成树的敛播（见15.5节）
3）利用同步器来模拟同步算法（见16.5.1节）
4）利用一致性全局快照来检测异步算法的终止（见19.2.3节）

15.3 生成树的构造、广播和敛播

异步网络中最基本的任务之一是为网络构造一棵以所给源节点i_0为根的生成树，并且利用这一生成树进行广播和敛播通信。在本小节中，我们描述针对这些任务的协议。我们再次假设底层图$G=(V, E)$是无向的、连通的。进程无需知道网络的大小或直径。无需UID。

对生成树问题，要求是网络中每个进程通过*parent*输出动作来最终报告在图G的生成树中它的父节点的名字。4.2节中描述了同步算法*SynchBFS*，该算法构造以i_0为根的广度优先生成树。*SynchBFS*算法从i_0开始同步地搜索图，让每个非源进程把它听到的第一个邻居报告为它的父节点。这个算法可以在异步环境中运行，仍然可以保证产生出一棵生成树，但不一定是广度优先生成树。异步算法的代码如下：

***AsynchSpanningTree$_i$* automaton:**

Signature:

Input:
$receive(\text{“}search\text{”})_{j,i}$, $j \in nbrs$

Output:
$send(\text{“}search\text{”})_{i,j}$, $j \in nbrs$
$parent(j)_i$, $j \in nbrs$

496

States:

$parent \in nbrs \cup \{null\}$, initially $null$
reported, a Boolean, initially *false*
for every $j \in nbrs$:
$send(j) \in \{search, null\}$, initially *search* if $i = i_0$, else $null$

Transitions:

$send(\text{“}search\text{”})_{i,j}$
Precondition:
$send(j) = search$
Effect:
$send(j) := null$

$parent(j)_i$
Precondition:
$parent = j$
$reported = false$
Effect:
$reported := true$

$receive(\text{“}search\text{”})_{j,i}$
Effect:
if $i \neq i_0$ and $parent = null$ then
$parent := j$
for all $k \in nbrs - \{j\}$ do
$send(k) := search$

Tasks:

$\{parent(j)_i : j \in nbrs\}$
for every $j \in nbrs$:
$\{send(\text{“}search\text{”})_{i,j}\}$

定理15.16 *AsynchSpanningTree*算法构造出一棵生成树。

证明概要 证明的关键断言是：

断言15.3.1 在任意可达状态中，由所有*parent*变量定义的边形成包含i_0的G的子图的生成树；并且，如果任意通道$C_{i,j}$中存在消息，那么i在生成树中。

通常这可以用归纳法证明。为了说明活性条件——每个节点最终都将包含于生成树中——我们用到另外一个不变式：

断言15.3.2　在任意可达状态中，如果$i = i_0$或$parent_i \neq null$，并且如果$j \in nbrs_i - \{i_0\}$，则$parent_j \neq null$，或者$C_{i,j}$包含一条*search*消息，或者$send(j)_i$包含一条*search*消息。

这样我们可以证明对于任意$i \neq i_0$，在时间$distance(i_0,i) \cdot (l+d)$内都会有$parent_i \neq null$，这暗示了
497 活性条件。 □

复杂度分析　在*AsynchSpanningTree*的任意公平运行中，消息的总数是$O(|E|)$,并且除了i_0之外的所有进程在时间$diam(l+d)+l$内产生*parent*输出。（这里不存在堆积问题，因为在每一通道上只有一条消息被发送。）

注意，*AsynchSpanningTree*算法中产生的路径可能比网络的直径长。这是因为在异步网络中，消息有时在较长的路径上比在较短的路径上传得更快。尽管如此，产生树的时间仍然以直径为界，因为每个进程收到第一个*search*消息的时间不大于一条消息从i_0沿着最短路径传送到它所需的时间。

消息广播　如同*SynchBFS*的情况，很容易增强*AsynchSpanningTree*算法来实现从i_0开始的消息广播。在生成树的形成过程中，消息只需被放到所有*search*消息的尾部。因此广播的通信复杂度是$O(|E|)$，所需时间是$O(diam(l+d))$。

儿子指针　也很容易增强*AsynchSpanningTree*算法来让父节点知道它们的儿子是谁。既然这里假定通信是双向的，*search*消息的接收者所要做的就是以*parent*或*non-parent*消息来做出合适的回应。

预先计算好的具有儿子指针的生成树可用来从进程i_0对网络中所有其他进程广播消息。每条消息由i_0发送给它的所有儿子，然后继续由父节点发送给它的儿子，直到消息到达树的叶子节点。每次广播的消息数目是$O(n)$，时间复杂度是$O(h(l+d))$，其中h是生成树的高度。有一个有趣的时间异常：如果树是由*AsynchSpanningTree*算法生成，那么广播的时间复杂度是$O(n(l+d))$；即使*AsynchSpanningTree*算法本身所用时间以直径为界，广播也不需要$O(diam(l+d))$的时间。这是因为*AsynchSpanningTree*产生的树的高度有可能大于直径。

预先计算好的具有儿子指针的生成树也可用来从树中所有进程敛播消息到i_0。这个工作方式和同步环境中一样：每个叶节点上的进程将信息发送给父亲节点。除i_0外的每个内部节点上的进程在收到所有儿子节点的信息之前一直等待，收到之后则将自己的信息与之合并，然后将
498 结果发送给它的父节点。最后，i_0在收到它的所有儿子的信息之后将它们与自己的信息合并，产生出最终结果。消息的数目是$O(n)$,所用时间为$O(h(l+d))$。和同步环境中一样，这一方案可以用于对基于分布输入的函数进行计算。

广播和敛播的结合可以用于：让i_0给所有其他进程发送一条消息，然后它接收所有进程的回应，回应表示进程成功地接收到信息。每一叶节点在收到广播消息后简单地开始敛播。消息的总条数仍是$O(n)$，所用时间仍是$O(h(l+d))$。

在生成树构造过程中我们也可以让i_0广播一条消息并且接收所有进程的回应。令W为能被广播的值的集合。消息集M为$\{(\text{“}bcast\text{”},w):w \in W\} \cup \{\text{“}ack\text{”}\}$。

***AsynchBcastAck_i* automaton:**

Signature:

Input:
$receive(m)_{j,i}$, $m \in M$, $j \in nbrs$

Internal:
$report_i$

Output:
$send(m)_{i,j}$, $m \in M$, $j \in nbrs$

States:
$val \in W \cup \{null\}$, initially the value to be broadcast if $i = i_0$, else *null*
parent $\in nbrs \cup \{null\}$, initially *null*
reported, a Boolean, initially *false*
acked, a subset of *nbrs*, initially $\emptyset$
for every $j \in nbrs$:
send(j), a FIFO queue of messages in M; if $i = i_0$ then this initially contains the single element ("*bcast*", w), where $w \in W$ is the value to be broadcast; otherwise this is empty

Transitions:

$send(m)_{i,j}$
Precondition:
m is first on *send*(j)
Effect:
remove first element of *send*(j)

$receive(\text{"}ack\text{"})_{j,i}$
Effect:
acked := *acked* $\cup \{j\}$

$report_i$ (for $i \neq i_0$)
Precondition:
parent $\neq$ *null*
acked = *nbrs* $- \{parent\}$
reported = *false*
Effect:
add "*ack*" to *send*(*parent*)
reported := *true*

$receive(\text{"}bcast\text{"}, w)_{j,i}$
Effect:
if *val* = *null* then
val := w
parent := j
for all $k \in nbrs - \{j\}$ do
add ("*bcast*", w) to *send*(k)
else add "*ack*" to *send*(j)

$report_i$ (for $i = i_0$)
Precondition:
acked = *nbrs*
reported = *false*
Effect:
reported := *true*

Tasks:
$\{report_i\}$
for every $j \in nbrs$:
$\{send(m)_{i,j} : m \in M\}$

499

复杂度分析 总共的通信为$O(|E|)$，时间为$O(n(l+d))$。时间的上界依赖于n而不是$diam$，因为前面所说的时间异常——广播有可能在较长路径上传得很快，而回应有可能沿着同一条路径传送得很慢。在16章中，将看到如何得到时间复杂度只依赖于$diam$的算法。

垃圾收集 如果*AsynchBcastAck*中的树只用来发送并且确认一条消息，那么每个进程都可以在执行*report*指令和发出*acks*之后删除与算法有关的所有信息。我们将这一修改及其正确性的证明留做一道习题。

在领导者选举中的应用 对于其中没有独一无二源节点并且进程不知道节点数目或网络直径的任意图，异步广播和敛播可以用来解决领导者选举问题。现在进程需要UID。为了得到网络中最大的UID，我们允许每个节点都启动广播 - 敛播以找出网络中的最大UID。发现最大UID和自己的UID相等的那个节点宣布自己为领导者。这一算法用了$O(n|E|)$条消息。我们把时间复杂

度的分析工作留做一道习题。

在本节最后，在只有局部知识、除了UID没有任何特异节点的情况下，我们指出在无向连
500 通图网络中两个基本问题之间的紧密联系：

1）为图找出一棵（无根的）生成树

2）选出领导者节点

首先，如果给定一棵无根生成树，则有可能按如下方法选出领导者。思路与4.4节最讨论的同步情况时的思路是一样的。

***STtoLeader*算法：**

该算法使用从叶节点开始的*elect*消息敛播。每个叶节点初始时都给它的惟一邻居发送一条*elect*消息。从所有其他邻居接收到*elect*消息的任意节点都将一条*elect*消息发送给剩下的那个邻居。

最后有两种可能：要么有一个特别进程在发送出*elect*消息之前收到它的所有通道的消息，要么*elect*消息从两个方向发送到一条特殊边上。在第一种情况中，*elect*消息敛播到的进程将自己选为领导者。在第二种情况中，与这条边相邻的具有较大UID的那个进程将自己选为领导者。

定理15.17 *STtoLeader*算法能够在进程只具有局部知识和UID的情况下，在带生成树的无向图网络中选出领导者。

复杂度分析 *STtoLeader*算法用了最多n条消息，仅用时$O(n(l+d))$。

相反地，如果领导者已给出，那么我们已经显示怎样用*AsynchSpanningTree*来构造一棵生成树。这需要$O(|E|)$条消息和$O(diam(l+d))$时间。所以，对这两种基本算法的（相当小）代价取模后，领导者选举问题和寻找任意生成树问题是等价的。

15.4 广度优先搜索和最短路径

我们在4.2节中考虑了广度优先搜索（BFS）的问题，在4.3节中考虑了寻找最短路径的问
501 题，现在我们在异步网络中重新考虑这些问题。现在我们假定底层图$G=(V, E)$是连通无向图并且有惟一的源节点i_0。对于最短路径问题，我们还假定每条无向边$(i, j) \in E$都有一个非负实数权$weight(i, j)$，并且边两端的进程都知道这个权值。我们假定进程不知道网络的大小或直径，并且没有UID。

对广度优先搜索来说，问题是网络中每个进程如何通过*parent*输出动作来报告出在广度优先树中它的父节点的名字。回忆在同步情况中，这是通过简单*SynchBFS*算法来完成的。*SynchBFS*算法的异步版本是15.3节中的*AsynchSpanningTree*算法；它可以确保产生出一棵生成树，但是不一定是广度优先生成树。

可以对*AsynchSpanningTree*进行修改，使得进程对错误的*parent*指定进行修正。也就是说，如果进程i起始时把它的一个邻居j识别为父亲，之后在一条更短路径上找到另外一个邻居k的信息，那么进程i把它的*parent*指定改为k。在这种情况中，进程i必须将它的修改通知给其他邻居，以便它们也可以修正自己的*parent*指定。代码如下：

***AsynchBFS*$_i$ automaton:**

Signature:

Input:
$receive(m)_{j,i}$, $m \in \mathbb{N}$, $j \in nbrs$

Output:
$send(m)_{i,j}$, $m \in \mathbb{N}$, $j \in nbrs$

States:
$dist \in \mathbb{N} \cup \{\infty\}$, initially 0 if $i = i_0$, ∞ otherwise
$parent \in nbrs \cup \{null\}$, initially $null$
for every $j \in nbrs$:
$send(j)$, a FIFO queue of elements of $\mathbb{N}$, initially containing the single element 0 if $i = i_0$, else empty

Transitions:

$send(m)_{i,j}$
Precondition:
m is first on $send(j)$
Effect:
remove first element of $send(j)$

$receive(m)_{j,i}$
Effect:
if $m + 1 < dist$ then
$dist := m + 1$
$parent := j$
for all $k \in nbrs - \{j\}$ do
add $dist$ to $send(k)$

Tasks:
for every $j \in nbrs$:
$\{send(m)_{i,j} : m \in \mathbb{N}\}$

定理 15.18 在*AsychBFS*算法的任意公平运行中，系统最终稳定在一个状态，其中*parent*变量代表一棵广度优先树。 502

证明概要 首先证明

断言 15.4.1 在任意可到达状态中，以下都为真。

1）对于每一个进程$i \neq i_0$，如果$dist_i \neq \infty$，那么$dist_i$是G中某条从i_0到i的路径p的长度，其中i的前驱是$parent_i$。

2）对于通道$C_{i,j}$中的每一消息m，m是某条从i_0到i的路径p的长度。

这意味着每个进程i总是拥有关于某条从i_0到i的路径的正确信息。但是为了证明活性属性——每一个进程最终都获得关于最短路径的信息——我们需要另一个不变式，它意味着关于最短路径的信息是被“保存”了的。

断言 15.4.2 在任意可到达状态中，对于每对邻居i和j，要么 $dist_j \leq dist_i + 1$，要么 $send(j)_i$ 或$C_{i,j}$包含值$dist_i$。

然后我们可以证明对于任意i，在时间$distance(i_0, i) \cdot n(l+d)$内我们有$dist_i = distance(i_0, i)$；这个证明可由对$distance(i_0, i)$的归纳得出。（这里我们考虑了堆积）这足以证明活性需求。 □

复杂度分析 在异步BFS算法的一个运行中发送的消息的数量是$O(n|E|)$；这是因为每一个节点可以获得最多n个对它与i_0的距离的不同估值，每个估值导致恒定数量的消息通过它的邻边。直到系统达到一个稳定状态为止的时间是$O(diam \cdot n(l+d))$；这是因为从i_0到任一节点的最短路径的长度最多是$diam$，并且任一通道中最多有n个消息。（这里我们又考虑了堆积。）

终止 *AsynchBFS*的一个问题是：过程没有办法得知在什么时候就不需要做更多修正（即使网络的大小是已知的）。因此，在技术上，这个算法不是*BFS*问题的一个解法，因为它永远不能产生所需的*parent*输出。通过增加对所有消息的确认，并像在*AsynchBcastAck*一样将确认敛播给i_0，从而增强*AsynchBFS*算法来产生输出是可能的。这使得i_0知道系统何时到达稳定状态，然后它广播一个信号给所有进程以令进程执行*parent*输出。 503

这个敛播有点复杂，因为，与*AsynchBcastAck*不同，一个进程i可能需要参加多次。每当进程

i从邻居j得到一个新$dist$估值并发送修正到所有其他邻居时，它一直等待所有这些邻居的相应确认，然后发送确认给j。簿记（bookkeeping）用于分离不同的确认。我们将这个留做一道习题。

已知直径　如果$diam$是已知的，那么通过仅允许小于或等于$diam$的距离估值可以改善*AsynchBFS*算法。通过这个更改，对于它与i_0之间的距离，每个节点仅得到最多$diam$个不同估值，使得通信复杂度为$O(diam|E|)$，时间复杂度为$O(diam^2(l+d))$。加上上述终止性的话，复杂度界限不变。

现在给出另一个解法；它产生所需的*parent*输出。它不需要知道网络图直径或大小的任意知识。这个解法的通信复杂度比*AsynchBFS*算法的任何版本都低，但时间复杂度高于已知$diam$的*AsynchBFS*算法版本的时间复杂度。

***LayeredBFS* 算法：**

BFS树是在层中构造的，每个层k由树中深度为k的节点组成。这些层构建在一系列阶段中，每层对应一个阶段，所有层都由进程i_0调控。

在第一阶段中，进程i_0发送*search*消息给它的所有邻居并且等待接收确认。收到阶段1的*search*消息的进程发送一个肯定确认。这使得树中深度为1的所有进程都能够确定它们的父母，即i_0，并且理所当然地，i_0也知道它的子女。这样构造了层1。

由上归纳，我们假定k个阶段已经完成并且前面k层已经建造好：深度至多为k的每个进程都知道BFS树中它的父母，并且深度至多为$k-1$的每个进程都知道它的子女。此外，源节点i_0知道所有这些都已完成。为了在第$k+1$阶段中构建第$k+1$层，进程i_0沿着已构建出的生成树的所有边广播一条*newphase*消息，目的地是深度为k的进程。

504　一旦接收到一条*newphase*消息，每个深度为k的进程都给它的所有邻居（除了父母之外）发送*search*消息并且等待接收确认信息。当一个非i_0进程在一个运行中接收到它的第一条*search*消息时，它指定此发送者为它的父母并且返回一条肯定的确认信息。当一个非i_0进程收到一条后来的*search*消息时，它返回一个否定的确认信息。当i_0收到任意*search*消息，它返回一条否定的确认信息。当一个深度为k的进程接收到它所有*search*消息的确认时，它指定那些发送给它肯定确认信息的进程为它的子女。

然后，深度为k的进程沿着深度为k的生成树的边，将它们已经完成对儿子确认的信息敛播给i_0。它们还敛播一位，说明是否发现深度为$k+1$的节点。在没有新节点被发现的阶段之后，进程i_0便终止这个算法。

定理 15.19　*LayeredBFS*算法产生一棵广度优先生成树。

复杂度分析　*LayeredBFS*算法使用$O(|E|+n \cdot diam)$条消息。总共有$O(|E|)$条*search*和确认消息，原因在于每条边最多在每个方向上探测一次。而且，在每个阶段中，树的每条边最多被*newphase*和敛播消息通过一次；由于最多有$diam+1$个阶段，故总共产生最多$O(n \cdot diam)$条这样的消息。每个阶段用时$O(diam(l+d))$，因此时间复杂度为$O(diam^2(l+d))$。

$diam$已知的*AsynchBFS*算法和*LayeredBFS*算法揭示通信量和时间复杂度之间的一种平衡。这一平衡将在下面的*AsynchBFS*和*LayeredBFS*混合算法中得到进一步印证。*HybridBFS*算法使用一个参数m，$1 \leqslant m \leqslant diam$。如果$m=1$，那么*HybridBFS*和*LayeredBFS*是一样的，而如果$m=diam$，*HybridBFS*与已知$diam$的*AsynchBFS*算法类似。对于m取中间的那些值，通信和时间复杂度估量则介于*LayeredBFS*算法和已知$diam$的*AsynchBFS*算法的相应值之间。

***HybridBFS*算法：**

该算法工作在阶段中。在每个阶段中，BFS树中的m个层是确定的（而不似*LayeredBFS*中

那样只有一个层是确定的）。在每个阶段中，我们使用如同*AsynchBFS*中的那些修正来异步地探测下一m层。确认信息被敛播回进程i_0。到一次敛播完成的时候，进程i_0知道在正被探测的层中，所有进程都已稳定在正确的距离估值上。 505

复杂度分析 *HybridBFS*算法的通信复杂度为$O(m|E|+\frac{n\cdot diam}{m})$。共有$O(m|E|)$条*search*和确认消息，原因在于每条边只携带至多m个不同距离估值信息。并且，在每个阶段中，树的每条边至多被*newphase*和敛播消息通过一次；由于至多有$O(\frac{diam}{m})$个阶段，所以至多有$O(\frac{n\cdot diam}{m})$条这样的消息。每个阶段用时$O(diam(l+d)+m^2(l+d))$。（m^2来自一个通道中m条消息堆积的可能性。）这样，总时间复杂度为$O\frac{diam^2}{m}(l+d)+diam\cdot m(l+d)$。

我们已经给出三种算法来解决BFS问题：*AsynchBFS*算法（带终止性）、*LayeredBFS*算法和*HybridBFS*算法。为了对这三种算法进行简单比较，我们考虑带终止性的且$diam$已知的*AsynchBFS*算法。我们忽略本地处理时间l并且忽略链路中堆积的影响，使用d作为在通道中传送每条消息的上界。我们得到

	消息数	时间
AsynchBFS:	$O(diam\|E\|)$	$O(diam\cdot d)$
LayeredBFS:	$O(\|E\|+n\cdot diam)$	$O(diam^2 d)$
HybridBFS:	$O(m\|E\|+\frac{n\cdot diam}{m})$	$O(\frac{diam^2}{m}d)$

现在我们转到在基于带权无向图的异步网络中寻找最短路径的问题：在一条距源节点i_0的最短路径树中，为网络中的每个进程确定并输出它的父母和它到i_0的距离。广度优先搜索问题便是所有的权值均为1的最短路径问题的特殊情况。

回忆在同步系统中，*BellmanFord*算法解决了寻找最短路径的问题。尽管这个算法是同步的，它仍需要修正错误的路径估值。*BellmanFord*算法可以使用以下代码来异步地运行，这些代码是*AsynchBFS*代码的自然扩展。*AsynchBellmanFord*算法是1969年至1980年间ARPANET中用来建立路由的算法。 506

$AsynchBellmanFord_i$ automaton:

Signature:

Input:
$receive(w)_{j,i}$, $w \in R^{\geq 0}$, $j \in nbrs$

Output:
$send(w)_{i,j}$, $w \in R^{\geq 0}$, $j \in nbrs$

States:

$dist \in R^{\geq 0} \cup \{\infty\}$, initially 0 if $i = i_0$, ∞ otherwise
$parent \in nbrs \cup \{null\}$, initially $null$
for every $j \in nbrs$:
$send(j)$, a FIFO queue of elements of $R^{\geq 0}$, initially containing the single element 0 if $i = i_0$, else empty

Transitions:

$send(w)_{i,j}$
　Precondition:
　　w is first on $send(j)$
　Effect:
　　remove first element of $send(j)$

$receive(w)_{j,i}$
　Effect:
　　if $w + weight(j,i) < dist$ then
　　$dist := w + weight(j,i)$
　　$parent := j$
　　for all $k \in nbrs - \{j\}$ do
　　add $dist$ to $send(k)$

Tasks:

for every $j \in nbrs$:
　$\{send(w)_{i,j} : w \in R^{\geq 0}\}$

定理 15.20　在*AsynchBellmanFord*算法的任意公平运行中，系统最终稳定在一个状态，其中*parent*变量代表一棵以i_0为根的最短路径树，并且*dist*变量包含节点到i_0的正确距离。

如同对*AsynchBFS*一样，*AsynchBellmanFord*的一个问题是进程无法知道在什么时候就不需要做进一步的修正。因此，该算法并不是严格正确的，因为它永远不能产生所需的输出。我们可以用在*AsynchBFS*中使用的同一种方法（即通过敛播确认来增强*AsynchBellmanFord*），以获得所需的输出。

对*AsynchBellmanFord*的复杂度分析是有趣的，主要是因为最坏情况下消息和时间复杂度都极其糟糕——它们都达到n指数级。作为对比，回忆同步*BellmanFord*算法仅需要$(n-1)|E|$
507 条消息和$n-1$个轮，而*AsynchBFS*算法（不知道直径的值也没有终止性）仅需要$O(n|E|)$条信息和$O(diam \cdot n(l+d))$的时间。

定理 15.21　令n为任意偶数，$n \geq 4$。存在一个n节点带权图，其中在最坏情况下，*AsynchBellmanFord*算法发送至少$\Omega(c^n)$条消息并且花费至少$\Omega(c^n d)$的时间才能稳定下来，这里的常量$c>1$。（我们可取 $c = 2^{\frac{1}{2}}$）。

证明　我们假定通道是通用FIFO可靠通道。令 $k = \frac{n-2}{2}$。令G为图15-10中描绘的带权图。图G的大多数边的权值都为0；仅有的非零边是右边的斜边，并且它们的权值连续以2的次幂递减。

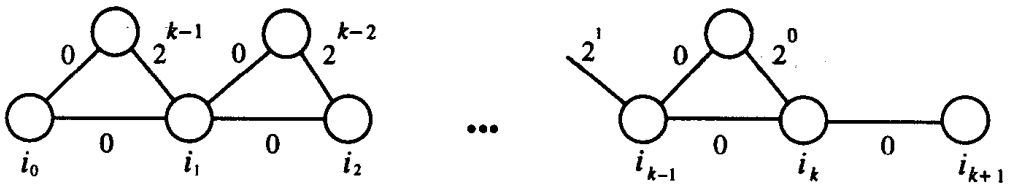


图15-10　*AsynchBFellmanFord*算法的坏有权边

我们来证明在图G上的*AsynchBellmanFord*算法的一个运行中，进程i_k可能取的有限*dist*估值恰好是集合$\{2^k-1, 2^k-2, \cdots, 3, 2, 1, 0\}$中的值。每个值都可由沿着$i_0$到$i_k$的一条特定路径的消息流产生。事实上，我们证明在同一运行中，强行使i_k按最大值到最小值的顺序来取这些估值是可能的。

假设在上面的路径中消息传播得很快，这样i_k的估值为2^k-1。接着，从i_{k-1}到i_k的消息沿着下面的路径到达i_k，使得i_k的新估值为2^k-2。再接着从i_{k-2}到i_{k-1}的消息沿着在下面的路径到达i_{k-1}，使得i_{k-1}将它的估值减少了2，从2^k-2减至2^k-4。进程i_{k-1}然后将这个减少了的估值从两条路径发

送给i_k。再一次假设在上面路径中消息传递得更快，这样进程i_k接着获得估值2^k-3，之后为2^k-4。

接着，从i_{k-3}到i_{k-2}的消息沿着下面的路径到达i_{k-2}，使i_{k-2}减少它的估值。以这样的方法继续下去，我们可以使进程i_k依次获得全部的估值2^k-1，…，0。

让系统运行在这样一个方式下是可能的，其中所有进程和所有除了$C_{i_k,i_{k+1}}$之外的通道迅速地操作。这导致$C_{i_k,i_{k+1}}$中产生一个由2^k条消息组成的队列，消息数为$\Omega(2^{\frac{n}{2}})$，或$\Omega(c^n)$。此外，如果所有消息都花费最大时间来传递的话，那么时间复杂度就为$\Omega(c^n d)$，得证。 □ 508

下面我们来考虑*AsynchBellmanFord*的算法复杂度上界。通道$C_{i,j}$发送的消息的数量与发送进程i所获得的不同估值的数量成正比。这种估值必然不大于图中从i_0到i的不同简单路径的条数，即$O(n^n)$。（实际上，它更小，但我们将此改进留做一道习题。）因此，总通信复杂度为$O(n^n|E|)$。鉴于一个通道中消息的数量的界限为n^n，时间复杂度的一个上界为$O(n^{n+1}(l+d))$。

注意时间界限依赖于消息通道中的堆积。如果我们采用更简单的假设（这种假设有时会在理论研究文献中用到），即任意消息从开始发送到被接收为止最多花费时间d（不考虑本地处理时间），那么通过计算可知*AsynchBellmanFord*算法的时间界限仅为$O(nd)$。这显然不是一个对这个算法的现实分析。

15.5 最小生成树

在本章的最后部分，我们来考虑如何为基于任意一个无向连通图的网络构造一棵最小权值生成树。在4.4节，我们曾给出一个算法——*SynchGHS*——来解决同步系统中的这一问题；现在我们示范如何修改这一算法以使得它适用于异步系统。称最终得出的算法为*GHS*算法，这种算法以其发现者Gallager、Humblet和Spira的名字命名。它是分布式计算理论中著名的算法之一。该算法是一个严格设计的复杂算法，它非常有趣，可以作为算法证明方法的一个案例研究。

我们建议此时你重新读一下4.4节；4.4节包含*GHS*算法所基于的基本原理以及*SynchGHS*算法，其中*SynchGHS*算法包含*GHS*算法所需的许多思想。

15.5.1 问题描述

如前，我们假定基本图$G=(V, E)$是连通且无向的，并且假设每条边都有权值。我们希望进程互相协作来构造图G的一棵最小权值生成树（MST），即包含G中所有顶点的一棵生成树，其所有边权值之和小于或等于G的任意其他生成树的所有边权值之和。

我们假定进程具有UID，并且与连接节点相关的进程都知道每条边的权值。我们来作一个技术性假设：所有边的权值都是惟一的。4.4节末的讨论显示这个惟一性假设并没多大意义——边的权值相同状态可以通过使用邻近进程的UID来破坏。我们假定这些进程仅具有图的局部知识；特别地，它们既不知道节点数目也不知道直径。 509

我们假定进程初始是静止的，这就是说，在起始状态下是没有局部控制动作是允许的。我们假定每个进程都有一个唤醒输入动作，环境通过该动作来发信号以告知进程开始执行MST算法。我们允许任意数量的进程在一个运行中接收唤醒输入；这样，不管开始计算的进程数目为多少，也不管这些进程何时开始计算，该算法都可以正常工作。注意我们假定只有进程的起始状态是静止的；我们允许进程在收到任意种类的输入（一条唤醒消息或来自另一个进程的消息）时苏醒[⊖]。算法的输出是由组成一棵MST的边组成的集合，特别地，每一个进程需要输出在MST中与它相邻的边的集合。

⊖ 我们在此所做的关于*wakeup*消息的假定与15.1.1节最后所做的假定不同。在那里，我们假定*wakeup*消息到达所有的进程。

15.5.2 同步算法：回顾

回忆*SynchGHS*算法是基于MSTs的两个基本特性的（由引理4.3和引理4.4给出）。这些特性用于证明策略的正确性：在任意中间状态中，该算法构造一个生成树林，森林中的所有边都在MST中。那么这个生成树林组件的任一子集可以独立确定它的最小权值流出边（minimum-weight outgoing edge，简称MWOE），所有这些边都必定被包含在惟一一棵MST中。

*SynchGHS*算法按“层”工作。0层生成树林包含各个节点且不包含边。给定层k生成树林，算法通过允许层k生成树的所有组件确定它们的MWOEs，并将沿着这些边来合并组件，来构造出层$k+1$生成树林。于是每个层k组件包含至少2^k个节点。

510 一个组件的MWOE的确定是由该组件的惟一的领导者节点来控制的，这个领导者节点的UID被用作组件的标识符。领导者节点沿着组件的边广播一条请求消息以确定MWOE，然后进程使用一个询问协议以获知哪些邻居处于同一组件中和哪些邻居不处于同一组件中，然后进程将它们的信息敛播给领导者。组件的合并涉及从领导者节点到与MWOE相邻的进程之间的通信。使用优化的簿记，进程可以确保通信复杂度保持在$O(n \log n+|E|)$并且轮数保持在$O(n \log n)$。

如果试着在一个异步网络中运行*SynchGHS*，就会有一些困难出现。比如：

难点1：在*SynchGHS*中，当一个进程i询问邻居j以了解在当前生成树林中j是否和i处于同一组件中时，它知道j处于同一构造层。因此，如果进程j具有一个不同的组件标识符，那j和i必定不在同一个组件中。然而在异步环境中，会出现这样一种情况：进程j和i确实在同一个组件中，但进程j还不知道这一点（因为包含最近组件标识符的消息还未到达）。

难点2：鉴于层是同步的这一事实，*SynchGHS*算法的一条消息花费时间$O(n \log n+|E|)$。每个层k级组件至少拥有2^k个节点，故总层数最多为$\log n$。在异步环境中，存在以不平衡方式构造组件的危险，从而导致更多的消息。组件发送来寻找其MWOE的消息数目至少正比于组件中的节点数目。我们必须避免一种状况，其中一个大组件不断发现其MWOE指向一个单节点组件并且它和该单节点合并（如图15-11所示），因为这样的话将需要$\Omega(n^2)$条消息。

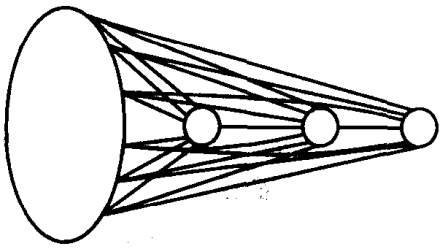


图15-11　一个大组件可能每次增加一个节点

难点3：在*SynchGHS*中，层保持同步，然而在异步环境中一些组件会到达比其他组件更高的层。这样，我们无法得知当不同层上的相邻组件同时搜索MWOE时会发生什么样的冲突。

15.5.3 *GHS*算法：概要

*GHS*算法与*SynchGHS*算法很相似。特别地，它们具有相同的通信量复杂度$O(n \log n+|E|)$，并且具有相当的时间界限$O(n \log n\ (l+d))$。

511 在*GHS*算法中，进程们自己构成组件，组件再合并成为更大的组件。初始的组件仅仅由各个节点组成。每个组件拥有一个区别于其他组件的领导者节点和MST的一个子图。

在任意组件内，进程在某一算法中互相协作来寻找整个组件的MWOE。这涉及从领导者节点开始进行广播，该广播要求组件中的所有进程去确定它们指向外部组件的最小权值边。关于所有这些边的信息被敛播回领导者节点，然后领导者节点就可以确定这个组件的MWOE。这个MWOE将被包含进MST中。

一旦找到MWOE，就沿着该条边发送一条消息到另一端的组件。这两个组件可以合并成一个新的更大组件。如此，重复整个过程以得到一个新组件。执行足够的合并之后，图中所有的节点最终被包含到一个组件中，这个组件的生成树就是所求的MST。

为使这个算法正确工作，还需要做一些工作。首先，进程i如何得知它的哪条边指向当前组件的外部？当然，我们需要给某种组件命名的方法，这样两个进程可以通过名字来确定它们是否在同一个组件中。但是，这个问题比以下这种情况更复杂：有可能像在难点1中描述的那样，一个有着不同组件名字的邻近进程j实际上已经和询问进程i处于同一组件中，但由于通信延迟它尚未了解到这一点。我们需要某种同步来确保进程j除非已经拥有所在组件的名字的当前信息，否则不做出“它在另一组件中”的响应。

第二个问题，正如上面难点2中提到的，一个不平衡的组件合并可能会产生过多的消息。 512
为了解决这个困难，我们将尽量使合并起来的组件的大小保持大致相等。更精确地说，我们将像*SynchGHS*中一样为每个组件都赋予一个*层*。如同在*SynchGHS*中一样，所有初始单节点组件都有$level = 0$，一个层k组件中的节点数至少为2^k。一个层$k+1$的组件可以通过合并两个层k组件而得，从而满足容量要求。这种策略和*SynchGHS*算法中的策略稍有不同。在*SynchGHS*中，任意多个层k组件都可以被合并到一个层$k+1$组件中去。

结果证明，这些层不仅有利于保持合并平衡，它们还提供一些鉴别消息来帮助进程确定它们是否属于同一组件中。

难点3中提到的第三个问题是：一些组件可以比其他组件到达更高的层，这可能导致一些不同层的相邻组件同时寻找MWOE时发送冲突。为了防止这种情况的发生就要求某种同步。

15.5.4 更详细的算法

*GHS*算法通过两种途径来合并组件，分别是：合并（*merging*）和吸收（*absorbing*）。

合并：这种合并操作可以作用于两个组件C和C'上，其中C的层与C'的层相等且C和C'具有一条公共MWOE。合并的结果是得到一个新组件，该组件包含C和C'中所有的节点和边以及公共的MWOE。新组件的层为$level(C)+1$。

吸收：这种合并操作也适用于两个组件C和C'，其中C的层低于C'的层且C的MWOE指向C'中的一个节点。吸收的结果将得到一个新组件，该组件包含C和C'中的所有节点和边以及C的MWOE。新组件的层与C'的层相同。事实上，我们一般不认为吸收是产生了一个新组件，而通常认为是把一个组件C添加到另一个已经存在的组件C'当中去。

吸收操作适用于某些进程落后于其他进程的情况。假设我们通过一系列合并操作，将一大堆节点合并成一个高层的大组件C'，而其他一些低层的小组件就落后了。如果其中一个小组件C发现它的MWOE指向C'，那么C就可以被吸收到C'中而无需知道关于C'的MWOE信息。这个操作耗费较小。 513

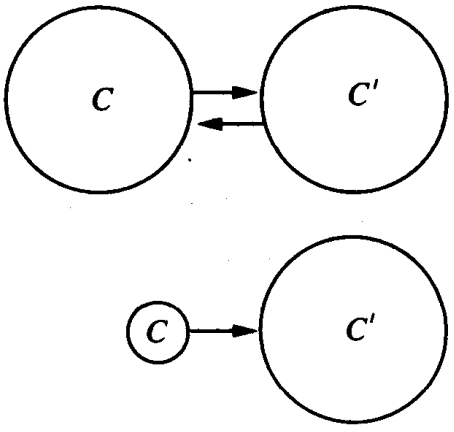


图15-12 合并和吸收

图15-12展示这两种合并策略。注意，吸收操作中C的等级小于C'的并不意味着C中节点数就少于C'中的节点数。图中所示的只是一种“典型”情况。

合并和吸收操作操纵层来处理难点2。特别地，它们保证任意层k组件都至少有2^k个节点。现在我们来证明合并和吸收操作足以将整张图的所

有组件合并成一棵MST。

引理15.22　假设我们从一个初始状态开始，其中每个组件都由单个层0节点组成。执行有限次吸收和合并操作，则在这些操作之后，要么只剩下一个组件，要么可以使用某个合并或（和）吸收操作。

证明　假设在一系列合并和吸收操作之后还剩下两个以上的组件。下面我们将证明还存在可用的操作。

我们考虑组件图G'，其节点数为当前组件数，且它的有向边对应于MWOE们；每条边都从组件出发，对此组件来说，它就是MWOE。引理4.5指出在G'的任意一个弱连通部分中，都存在惟一一个的长度为2的环，也就是说，存在两个组件C和C'，它们的MWOE均指向对方。但是很容易看出，这种情况下这两条MWOE都必须是源图G的同一条边。

现在我们来证明可以使用一个合并或一个吸收操作将C和C'合并起来。如果$level(C)=level(C')$，那么合并操作是可用的；反之如果C和C'具有不同的层值，那么具有较低层值的组件可被吸收至具有较高层值的组件中去。 □

514

现在我们更详细地考虑如何找到一个给定组件的MWOE。这涉及到每个组件中的各进程i分别确定出从该组件流出的最小权值边（如果有的话）$mwoe(i)$，然后所有进程将其信息送到一个领导者节点那里，由领导者节点选出具有最小权值的边。这需要一些附加机制。首先，我们需要一种能够选出每个组件的领导者节点的机制，第二，对每个进程而言，我们需要一种方法来确定一条给定的边是否是从组件流出的。

为了实现这些任务，对于层1或者更高层的每个组件，我们标识出一条称之为*核心边*的特殊边。这条边是根据一系列用于构造组件的合并和吸收操作来确定的。

- 在一个合并操作后，核心边是两个初始组件的共同MWOE。
- 在一个吸收操作后，核心边是最高层初始组件的核心边。

这样，一个组件的核心边是满足以下要求的一条边：最后一个用于构造组件的合并操作沿着它发生。

对于层1或更高层组件，我们将核心边对（从技术上讲，是核心边权值对）和层作为组件标识符。这是合理的，因为边的权值被假定是惟一的。我们也可以将核心边的其中一端指定为领导者，例如，将带有更高UID的那一端作为领导者。对于层0组件来说，其中的惟一节点当然就是领导者节点。

现在假设进程i希望确定它的一条通向邻居进程j的边是否是从进程i的当前组件流出的。如果进程j的当前组件标识符与i的一样，那么进程i就能确定进程j与它处于相同组件中。但是，如果进程j的组件标识符和进程i的不一样，进程i和进程j仍然可能处于相同组件中，只不过进程j尚未收到当前组件标识符的通知。我们可以解决一种特殊情况：如果进程j的组件标识符与进程i的不同，而且j的最后得知的层至少和进程i的一样高，那么我们就能够确定进程j和进程i不可能处于相同组件中。这是因为在一个运行中，一个节点在每层中最多有一个组件标识符，而且当进程i正积极地搜寻它的流出边时，i的组件的标识符必定是最新的。

515

于是，如果i和j具有相同的组件标识符，那么j回答它也在相同的组件中。同样地，如果i和j有不同的组件标识符且j的层至少要跟i的一样大，那么j回答与i在不同的组件中。剩下的一种情况就是j的层严格小于i的层；在这种情况下，进程j就会简单地延迟回答i，直到它自己的层上升到至少和i一样大。这就解决了难点1。

然而，注意：我们现在不得不重新考虑演进性证明，因为这个新的延迟确实会导致演进被阻塞。组件中某些进程对最小权值流出边的寻找可能被延迟这一事实意味着组件作为一个整体会延迟对MWOE的寻找；我们必须考虑这是否会导致系统进入一个状态，其中没有更多的合并和吸收操作能够运行。

为了证明这种情况不会发生，我们还是采用与前面本质上一样的演进性证明方法，但是这次我们仅仅考虑那些具有当前最低层k的组件。在这些组件中的所有进程都必须成功地确定出其最小权值流出边，因此，这些组件也必须成功地确定出它们的MWOE。如果每个层k组件发现它的MWOE指向一个更高层组件，那么就可以使用一个吸收操作。另一方面，如果每个层k组件发现它的MWOE指向另一个层k组件，那么定理4.5意味着我们必然得到一个涉及层k组件的长度为2的环，而且也可以使用一个合并操作。这样，即使是新类型延迟存在，算法也继续演进，直到整个MST被找到为止。

这样，我们已经看到每个进程是如何确定其最小权值流出边（如果有的话）的。然后，就像上面描述的那样，组件的领导者节点通过广播和收敛播来确定组件的MWOE，从而选出最小权值边。

我们仍然必须考虑难点3：不同层的邻近组件同时搜寻MWOE时可能会产生相互干扰。特别地，我们考虑如果一个低层组件C被一个更高层组件C'吸收进去而C'正在确定其MWOE时会发生什么样的情况。假设C的MWOE将C的节点i和C'的节点j连接起来。见图15-13。

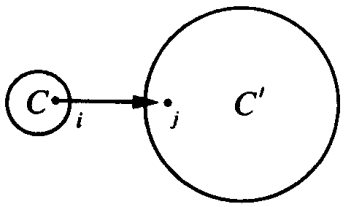


图15-13 当C'在搜索自己的MWOE时，组件C被吸收进C'

有两种情况需要考虑。第一种，假设进程j在还没有确定流出组件的最小权值边时吸收操作发生了。这种情况下，算法会同时在C和C'寻找合成组件的MWOE。j尚未有确定$mwoe(j)$的事实意味着它不会被包括在C中。

516

另一方面，假设进程j在吸收操作发生时已经确定$mwoe(j)$。这种情况下，我们要求$mwoe(j) \neq (i,j)$，也就是说，j的最小权值边不可能和C的MWOE相同。原因在于$mwoe(j)$已经被确定的事实意味着它指向一个其层至少跟C'的层一样大的组件。（一个技术点：由$mwoe(j)$的另一端点告诉j的那个层值至少和C'的层值一样大，这意味着它至少也这么大，因为进程得知的层值不会减少）。然而，因为C被吸收进C'，所以我们知道$level(C)$严格小于$level(C')$。故得$mwoe(j) \neq (i,j)$。这也意味着$mwoe(j)$的权值严格小于(i,j)的权值。

然后我们来证明合并组件的MWOE不可能与C中的节点相连。这是因为(i,j)是C的MWOE，因此不可能有权值比(i,j)的权值更小的C的流出边，也不可能有权值比已经发现的$mwoe(j)$的权值更小的C的流出边。于是，如果$mwoe(j)$在吸收操作期间已被确定，那么算法无需在C中寻找合并组件的MWOE。这是很幸运的，因为在C中搜索的话可能太迟了——进程j可能已经报告它的最小权值边，而组件C'在不知道新吸收进来的节点的情况下正处于对MWOE的确定过程中。

15.5.5 特殊消息

现在我们给出关于在*GHS*算法中发送的特殊消息的更多细节。这些消息属于如下类型：

- *初始*（*initiate*）。*initiate*消息在所有组件中广播，从领导者节点开始沿着组件生成树的边传播。通常[⊖]，它触发进程去开始试图寻找其$mwoe$。它还传播组件标识符（核心边

⊖ 这里有一个例外情况，我们将在下面提到。

和层）。

517

- 报告（*report*）。*report*消息将最小权值边信息敛播回领导者节点。
- 测试（*test*）。进程*i*给邻居*j*发送一条*test*消息来试图确定*j*是否与*i*处于同一组件中。这是进程*i*寻找它的*mwoe*的过程的一部分。
- 接收和拒绝（*accept* 和*reject*）。它们是对*test*消息的响应。它们告知测试节点响应节点是在不同组件中（*accept*）还是在同一组件中（*reject*）。
- 改变根（*changeroot*）。组件的领导者节点在确定MWOE之后发送*changeroot*消息给与MWOE相连的进程。它用来告知进程去尝试与MWOE另一端的组件合并。
- 连接（*connect*） 当组件*C*试图与另一个组件合并的时候，*connect*消息在*C*的MWOE上发送。当*connect*消息沿着同一条边的两个方向都发送后，一个合并操作就发生。当发送方将*connect*消息沿着边的一个方向发送给更高层进程时，一个吸收操作就发生。

在*test – accept – reject*协议中，测试进程*i*必须做一些簿记以使通信量复杂度保持较低水平；这与早些时候在*SynchGHS*中提及的簿记相类似。也就是说，进程*i*按权值的递增顺序来维持一个连接边列表。这些连接边可分为以下三类：

- *branch*边是那些已被确定属于MST的边。
- *reject*边是那些已被确定不属于MST的边，因为它们指向同一组件中的其他节点。
- *basic*边是其他所有边。它们是进程*i*不能确定是属于还是不属于MST的那些边。

开始时所有边都属于*basic*边。

当进程*i*寻找它的最小权值流出边时，它仅需沿着*basic*边发送*test*消息。接着按最小权值到最大权值的顺序来测试*basic*边。对每一个*basic*边，进程*i*发送一条含有组件*C*的标识符（*core*和

518

level）的*test*消息。接收到*test*消息的进程*j*检查它最近一次确定的组件标识符是否和发送者*i*的一样。如果一样，它就发送一条*reject*消息来响应。当*i*收到这条*reject*消息，它将这条边重新归类为*rejected*边。另外，如果接收进程*j*的*core*值和*i*的不同，并且它的层至少和*i*的一样大，那么它发送一条*accept*消息来响应。(这并不导致*i*将这条边重新归类。）最后，如果*j*的*core*值与*i*的不同，并且它的层严格小于*i*的，那么进程*j*简单地延迟响应，直到根据上面规则它能够发回一条*reject*或*accept*消息为止。

注意*i*可能收到边(*i*,*j*)的一条*accept*消息，但边（*i*,*j*）最终并非是整个组件*C*的MWOE。在这种情况下，同一条边（*i*,*j*）可能被进程*i*在接下来的搜索中重复测试。进程*i*只会在它发现某条边的确属于MST时，例如当进程*i*收到一条指向该边的*changeroot*消息或从该边收到一条*connect*消息时，才会将其重新归类为*branch*边。

当两条*connect*消息通过同一条边时，一个合并操作发生。然后该公共边被标识为新核心边，层增加1，并且具有较大UID的端点被选作新领导者节点。然后新领导者节点广播*initiate*消息，开始寻找通过合并操作构成的新组件的MWOE。当一个组件从更低层组件中收到一条*connect*消息时，一个吸收操作发生。接收进程知道它是否已找到它的*mwoe*，从而知道是否需要在这个新吸收组件中触发一次搜索。无论如何，都将广播一条*initiate*消息给组件的进程，以告知最新组件标识符。

注意，一旦每个进程再也不将任意连接边归为*basic*边的时候，它们便都可执行输出；输出的就是*branch*边集合。

定理 15.23　*GHS*算法解决了任意无向连通带权图网络中的*MST*问题。

15.5.6 复杂度分析

通信复杂度的分析与*SynchGHS*的类似，界限同样为$O(n \log n+(|E|)$。我们将消息分成两个集合，其消息数目分别为$O(n \log n)$和$O(|E|)$。$O(|E|)$来自所有边上导致拒绝的*test*消息数加上 519
*reject*消息数。共有$O(|E|)$条，因为每条边最多被拒绝一次：当进程i接收到一条边上的*reject*消息之后，i不会再次测试这条边。

所有其他消息——使进程接受一条边为其*mwoe*的*test-accept*对、用作广播和敛播的*initiate*和*report*消息、在组件已经确定其MWOE之后用到的*changeroot*和*connect*消息——负责为一个特定组件（即特定的核心边和层）寻找MWOE。在一个组件的此类任务中，这些消息按照如下方式与节点关联：每个节点至多与一种类型的消息关联。（特别地，每个进程发送至多一个成功的*test*消息。）这样，组件C使用的消息数为$O(|C|)$，其中$|C|$用来表示组件C中节点的个数。因此，总消息数正比于

$$\sum_{C}|C|$$

根据组件层来组织组件，重写表达式如下：

$$\sum_{k:0\le k\le \log n}\left(\sum_{C:level(C)=k}|C|\right)$$

对每个层k，内部和最多为n，因为没有节点会在出现在一个以上的层k组件中。因此，表达式的值最多为

$$\sum_{0}^{\log n} n = O(n\log n)$$

这样算法的总通信复杂度为$O(n \log n+|E|)$。

对于时间复杂度来说，包括一个初步协议来尽快唤醒所有进程是方便的。那么通过对k进行归纳可得：到所有进程都至少到达层k为止所需的时间为$O(kn(l+d))$。这样，总时间便为$O(n \log n(l+d))$。

下界 注意至少对于一些图，通信复杂度需为$\Omega(n \log n)$。例如，如果环中的MST通信复杂度比这个少，那么就有可能通过将一种高效通信的MST算法和*STtoLeader*算法合并起来，以获得一个通信复杂度也比这个小的领导者选举算法。但这样将与定理15.12矛盾，因为定理15.12说明对于大小为n的环的领导者选举算法，$\Omega(n \log n)$条消息是必要的。 520

15.5.7 *GHS*算法的正确性证明

*GHS*算法是本书中第一个没有给出正确性证明的算法，原因在于目前还没有简单的证明方法。通过各种各样的方法，该算法已被证明了至少四次，但是没有一种方法能够在短短几页内很好地说清楚。

一种有效途径是通常的不变式声明方法，这涉及收集大量的不变式和描述算法执行的所有不同任务。例如，有对广播和敛播任务的正确操作进行描述的不变式；有对*test-accept-reject*协议进行描述的不变式；还有对*changeroot-connect*协议进行描述的不变式。所有这些不变式可以通过大量的归纳证明来证明。这样的证明涉及大量实例和大量乏味冗长的细节，但是原则上是十分直截了当的。

但是这种蛮力证明似乎并没有充分利用算法中存在的模块化优点。例如，算法可以被分解成诸如广播－敛播和测试等独立任务，然而这种分解还没有被形式化地描述出来（如使用I/O

自动机合成操作）。所以我们并不清楚应该如何证明单独任务的正确性并把结果综合起来。

而且，这种蛮力不变式声明方法没有充分利用关于算法的高层次直觉知识。要注意的是，当讨论到这种算法时，我们更多的使用图、组件、层和MWOE等高层次的概念，而不是使用消息和局部变量等低层次的概念。看来一个好的证明应该尽可能使用高层次概念来进行。事实上，第二种证明方法就是如同用自动机来操纵图、组件等一样，给这个算法一个更高层次的描述并使用不变式来证明它的正确性。然后有可能证明详细算法正确地模拟了高层次描述。低层次和高层次算法之间的形式化对应关系就正如8.5.5节中定义的*模拟关系*。模拟证明方面的例子可参见10.9.4节中对*Infinite TicketME*互斥算法的证明以及第16章中对*SimpleSynch* 和*SafeSynch*同步算法的证明。对同步算法的证明优雅地示范如何使用两种方法来分解某些复杂的异步网络
521 算法：对单独进行推理的任务使用I/O自动机合成，使用模拟关系以允许在最高抽象层次上进行推理。

另一种证明*GHS*正确性的方法是尝试把它的行为与算法的同步版本（即*SynchGHS*）形式化地联系起来。非正式地说，两者十分接近。要注意的是，这种关系不是简单的模拟关系，因为在异步算法中，网络的不同部分远远不是同步的，就像它们是由当前层来决定的那样。无论使用什么对应方式，都必须允许对在网络上不同位置发生的活动进行某种重排序。

寻找*GHS*算法正确性的精确分解证明方法的任务作为一个有趣的开放问题。为了获得模块性，轻微地修改算法是可以接受的，只要修改不会影响到重要的算法思想和复杂度即可。

在第16章到第22章，你会看到各种各样的异步网络算法，它们使用各种各样的方法来进行分解。我们希望诸如*GHS*算法的复杂性能够使你相信去寻找这些分解方法是很重要的。

15.5.8 简单“同步”策略

*GHS*算法有很多*SynchGHS*算法没有的复杂因素，其中绝大部分来自于网络的不同部分可能远远不是同步的，它们的行为是由当前层来决定的。一种避免这些复杂因素的方法是尽可能接近地模拟*SynchGHS*，保持邻近进程的层大致相等。

SimpleMST算法：

这种算法基于组件的合并，每个组件都有一个相关层。初始组件都是层0单独节点。现在，使用与*SynchGHS*中相同的通用策略，层k组件只能合并到层$k+1$层组件中。

每个进程i保存一个*local-level*变量，这个变量跟踪着组件知道的最高层进程i。起初，*local-level*等于0，当进程i知道它处于新层k组件中时，它就把*local-level*增为k。

关键思想在于直到网络中所有进程的*local-level*都至少为k为止，层k进程i尽量不去参与对
522 其层k组件的MWOE寻找。实际上，要实现这一点需要昂贵的全局同步。但是事实上，弱一些的本地同步就足够了：每一个进程只是等待所有邻居的*local-level*至少为k。结果是所有进程都会发现这一点，每当进程的*local-level*增加时，进程在每条连接边上发送一条信息。

*SimpleMST*算法具有与*GHS*一样的时间复杂度上界，即$O(n \log n(l+d))$，当然，它比*GHS*要简单得多。然而，通信复杂度会更差，因为在每一层都使用同步消息：现在通信复杂度为$O(|E|\log n)$。

15.5.9 应用到领导者选举算法中

MST算法可以用来解决在一个带有UID的任意无向连接带权图中的领导者选举问题。也就是说，建构完一个MST以后，进程参与*STtoLeader*协议来选出领导者。

注意，进程不需要知道MST算法何时完成在整个网络中的运行过程。对于每个进程i来说，只要等到它局部任务完成，即输出它在MST中的连接边集合就足够了。如果进程i在执行MST

协议的输出之前收到一条消息，而这条消息是*STtoLeader*算法的一部分，则它只是简单地延迟这条消息，直到MST算法完成为止。这里的思想与15.1.1节结尾中讲到的在*wakeup*消息中处理输入的通用策略的思想相同。

如果使用*GHS*算法来建立MST，那么选出领导者所需的消息总数为$O(n \log n+|E|)$，所需的总时间为$O(n \log n(l+d))$。

15.6 参考文献注释

*AsynchLCR*算法和*HS*算法的异步版本，与这些算法的同步版本一样，都来源于LeLann[191]，Chang和Roberts[71]以及Hischberg和Sinclair[156]的论文。*Peterson Leader*算法由Peterson提出并出现在[239]中。获得$O(n \log n)$通信复杂度的另一个单向算法是Dolev、Klawe和Rodeh在[97]中提出来的。Higham和Przytycka[155]提出异步环中领导者选举问题的目前为止最小的上界$1.271n \log n+O(n)$。

523

15.1.4节开始部分中的讨论，即如何将领导者选举问题的同步通信复杂度下界带到异步环境中，出自Gafni[129]。Burns在[61]中直接证明了针对异步环境的下界。

Afek和Gafni[6]提出在完全异步的网络结构中领导者选举问题的复杂度界限。

SMT、广播和敛播算法最早是出现在Segall[258]和Chang[72]的论文中。*AsynchBFS*和*AsynchBellmanFord*算法是在Bellman和Ford[43,125]的串行最短路径算法的基础上提出的。*AsynchBellmanFord*算法本质上就是在1969年到1980年间用来建立ARPANET网中的路由的算法[223]。本章中提到的*AsynchBFS*和*AsynchBellmanFord*算法的终止协议基于Dijkstra和Scholten对“扩散计算”（diffusing computation）的终止检测方面的工作[92]；我们在19.1节中阐述该工作。*LayeredBFS*算法和它的m层版本受到Gallager[131]的工作的启发；这些结果后来被Awerbuch和Gallager[33]加以改进。还有一个有趣的最短路径算法是由Gabow[128]设计的。

Gallager、Humblet和Spira在[130]中提出*GHS*协议。他们那篇文章中的代码和本书的前提-结果代码有些不同。一个与本书的风格更接近的版本出现在Welch的博士论文[287]中。现在已经有几篇发表的论文针对*GHS*算法及其变种的正确性证明。Welch、Lamport和Lynch[288]使用模拟方法来证明正确性。Chou和Gafni[79]使用与同步算法的对应关系来证明该算法的一个轻微改进版本的正确性。Stomp和de Roever[87]，Janssen和Zwiers[164]也进行了证明。Awerbuch[31]设计一个时间复杂度为$O(nd)$、通信消息量为$O(n \log n)$的MST算法。Garay、Kutten和Peleg[132]设计一个时间复杂度为$O((diam+\sqrt{n})d)$的算法。Awerbuch、Goldreich、Peleg和Vainish[34]证明建立一棵最小生成树所需消息数为$\Omega(|E|)$的下界结果；这个结果假定消息的长度是有界的。*SimpleMST*算法出自Awerbuch的研究。

Humblet[160]设计一个异步分布式算法来在无向图网络中寻找最小生成树。

15.7 习题

15.1 将*AsynchLCR*算法与同步*LCR*算法正式联系起来，给出另一个针对*AsynchLCR*算法的正确性证明。

524

15.2 给出在15.1.1节结尾中描述的改进*AsynchLCR*算法的前提-结果代码，其中算法包括*wakeup*输入和*receive*缓冲区。

15.3 对于*HS*算法的异步版本：

a）给出前提-结果代码。

b）基于代码来证明算法的正确性。

c）假定每个进程的每个任务的时间上界为l，在任意通道中最早消息的传递时间上界为d，分析算法的时间复杂度。

d）假定任意消息的传递时间上界为l，忽略局部处理时间，分析算法的时间复杂度。

15.4 考虑一个在具有15个节点的环中的*PetersonLeader*算法，其中进程P_1, P_2, ..., P_{16}的UID分别为25，3，6，15，19，8，7，14，4，22，21，18，24，1，10，23。哪个进程将被选为领导者？

15.5 为第2章和第3章中描述的同步网络模型设计一个*PetersonLeader*算法版本。在你的算法中进程可以知道n。尽量让你的算法（编写和理解）简单，要求保持单向性和$O(n \log n)$的通信复杂度。分析你的算法的时间复杂度。

15.6 详细证明*PetersonLeader*算法的时间复杂度上界为$O(n(l+d))$。

15.7 为双向通信的环设计一个*PetersonLeader*领导者选举算法。在新算法版本中，仍在竞争的UID不需要在环中流动，而是可以停留在其产生的地方；每个进程在各个阶段都只是简单地从它的两个活动邻居中收集UID。给出你的算法的前提-结果代码，并分析它的时间复杂度和通信复杂度。

15.8 对*AsynchLCR*算法、异步*HS*算法和*PetersonLeader*算法进行扩展，使得非领导者通过*non-leader*$_i$输出动作来声明它不是领导者。分析这些算法的时间复杂度和通信复杂度。

525 15.9 详细证明定理15.11。

15.10 对声明15.15的证明的归纳步中的命题进行详细证明，即$\alpha_{L,M}$之后的状态是安静的。

15.11 扩展定理15.12的证明，使之对大小不是2的幂的环也适用。你能得到的最佳下界是多少？

15.12 考虑在基于双向线图的网络中的领导者选举问题；这种图由进程1,2,...,n组成，它们排成一条直线，邻居之间以双向边连接。假定进程通过局部变量right和left来识别它的邻居。每个进程都知道自己是否在末端上，且进程不知道n的大小：

a）使用很少数量的消息，针对这种网络来给出一个领导选举算法。

b）为什么这个结果和引理15.14中给出的下界不矛盾？

15.13 考虑15.2小节中描述的*OptFloodMax*算法的异步模拟，其中进程并不知道何时终止：

a）写出该异步模拟的前提-结果代码。

b）对于任意图G和UID分配，比较你的模拟算法和同步*OptFloodMax*算法的最大发送消息数量。

15.14 详细证明定理15.16。

15.15 详细证明*AsynchBcastAck*算法的正确性。

15.16 写出改进*AsynchBcastAck*算法的前提-结果代码，其中每个进程在执行一个报告动作和送出确认之后对关于算法的所有信息进行垃圾回收。证明它的正确性并分析它的复杂度。

15.17 为异步网络中的广播和确认设计一个算法，其时间复杂度依赖于网络直径而非节点总数。

15.18 把15.3节中的生成树、广播和收敛算法扩展到基于有向强连通图的网络中去。分析你的
526 算法的复杂度。

15.19 对在*AsynchBcastAck*的描述之后的领导者选举策略进行详细描述和复杂度分析。分析在以下两种不同假设情况中的时间复杂度：每个通道中最早消息的传递时间上界为d；每个通道中任意消息的传递时间上界为d。在第二种情况下，你可以忽略局部处理时间。

15.20 详细描述一个针对基于任意无向连接图G的异步网络的算法。该算法允许一个惟一的进程i计算G中的节点数。并证明其正确性。

15.21 详细证明定理15.18。

15.22 对于*AsynchBFS*算法：

a）构造一个运行，它使用尽可能多的消息；尝试获得给定的上界$O(n|E|)$。

b）构造一个运行，直到到达一个稳定状态为止，它花费尽可能长的时间。尝试获得给定的上界$O(diam \cdot n(l+d))$。

15.23 写出改进*AsynchBFS*算法的前提-结果代码，其中进程通过确认协议来产生*parent*输出。不要对网络图的大小和直径做任何假设。

证明你的协议的正确性并分析其复杂度。（提示：通信复杂度应与基本*AsynchBFS*算法的相同。鉴于在*AsynchSpanningTree*和*AsynchBcastAck*算法中讨论的定时异常，时间复杂度会变大。）

15.24 假定*diam*已知且进程产生*parent*输出，重新给出习题15.23中的改进*AsynchBFS*算法。

15.25 写出*LayeredBFS*算法的前提-结果代码，并证明其正确性。

15.26 使用自然语言或前提-结果代码来详细描述*HybridBFS*算法。并证明其正确性。

15.27 设计一个有效算法，允许在基于任意无向连接图G的异步网络中的一个惟一的进程i_0确定从i_0到网络中最远节点的最大距离k。分析它的时间复杂度和通信复杂度。 527

15.28 给出*AsynchBellmanFord*最短路径算法的尽可能严格的时间复杂度上界。

15.29 写出改进*AsynchBellmanFord*算法的前提-结果代码。其中进程通过确认协议来产生*parent*输出和距离输出。证明你的协议的正确性并分析其复杂度。

15.30 设计一个算法来找出从给定节点i_0到网络中其余节点的最短距离。你的算法的时间界限应该大大优于*AsynchBellmanFord*算法的时间界限$O(n(l+d))$。

15.31 扩展15.4节中的广度优先搜索算法和最短路径算法，使之适用于基于有向强连通图的网络。

15.32 给出完整的*GHS*最小生成树算法前提-结果代码。

15.33 考虑*GHS*最小生成树算法：

a）详细叙述并证明从第一个进程被唤醒到最后一个进程宣布结果之间的时间的界限。你可以假设使用一个初级协议来尽快地唤醒所有节点。

b）在a)中证明的上界有多严格？也就是说，描述给出算法的一个特定运行，该运行花费的时间接近你得到的时间上界。

15.34 描述*GHS*算法的一个运行，其中当进程i把边(i,j)归类为*branch*边时，作为对i的*test*消息的回应，*reject*消息沿着通道（i,j）到达进程i。证明算法能够正确地处理这种情况。

15.35 假定在*GHS*算法的一个运行的某点中，组件C中的进程i沿着某条边(i,j)发送出一条*connect*消息，其中边（i,j）通向一个与C处于同一层中的另一组件C'。证明组件C最终合并进C'或者被吸入某个包含C'的组件中。

15.36 研究问题：试比较*GHS*最小生成树算法的操作和*SynchGHS*的操作。例如，在这两种情况中产生的组件之间的关系如何？（在对*GHS*正确性的形式化证明中，有可能探索出这样一种联系）。 528

15.37 研究问题：试为本章和[130]中讲述的*GHS*算法寻找一个简单优美的正确性证明。你可以在需要的情况下稍微修改这个算法，只要你保持基本算法思想不变以及时间和消息复杂度不变。

15.38 对于*SimpleMST*算法：

a）写出前提-结果代码。

b）证明其正确性。

15.39 研究问题：试找出一个MST算法，其时间复杂度近似为$O(diam \cdot d)$，消息量为$O(\log n)$。

15.40 给出15.5.9小节中描述的领导者选举策略的形式化描述，即作为生成MST的I/O自动机和使用MST来进行领导者选举的I/O自动机的合成。详细描述这个自动机集合的交互情况，标识出哪些动作用于两个自动机集合之间的通信，并标识出每个自动机集合需要另一集合的哪些行为。

15.41 考虑习题15.12中描述的基于线图的网络。也就是说，图由进程1,2,...,n组成，它们排成一条直线，邻居之间以双向边连接。假定进程通过局部变量right和left来识别它的邻居。每个进程都知道自己是否在末端上，且进程不知道n的大小。

假设某个进程开始时都有一个很大的整数值v_i，且在任一时刻它只能在存储器中保存常数个这样的值。设计一个算法来在进程中对值进行排序，也就是说，使得某个进程i返回一个输出值o_i，其中输出集等于输入集且$o_1 \leq o_2 \leq \ldots \leq o_n$。尽量设计出最高效（以消息量和时间来衡量）的算法。并给出证明。

15.42 考虑一个基于任意拓扑结构的异步无向连接网络，其中每个进程都有一个UID。假定每个进程i开始时都接收到某个整数值v_i作为输入。设计一个算法，使得每个进程都返回网络中所有输入的总和。尽量使通信复杂度（以消息量来衡量）最少。并给出证明。

529

15.43 考虑一个银行系统，其中每个进程都保存一个数字以指示钱数。为简单起见，我们假定，不会发生外部存款或是提款，但是消息在任意时间中在进程之间传递，它包含着从一个地点“转移”到另一个地点的钱数。通道遵循FIFO顺序。

设计一个分布式网络算法，允许每个进程决定（即输出）自己的余额，所有余额的总和等于系统中钱的总额。假定算法的运行是由从（一个或多个系统地点中）外部到达的信号来触发的。（这些信号可在任意时刻发生，也可以在不同时间在不同地点发生。）

你的算法不应该会“不必要地”停止或延迟转账的进行。给出一个有说服力的证明，证明你的算法正确地工作。

15.44 设计一个在异步网络中工作的4.5节中描述的*LubyMIS*算法版本。给出并证明一个关于你的算法能够保证什么的严密声明。

530

第16章 同 步 器

在第15章中，我们给出几个在“裸”异步网络模型上直接编程的分布式算法示例。很显然，这种模型具有相当大的不确定性，使得直接编程非常困难。因此，我们希望得到便于编程的简化模型，而该模型的程序能够被翻译成针对通用异步网络模型的程序。

我们已经提出两种比异步网络模型更简单的模型——*同步网络模型*和*异步共享存储器模型*——并给出这两种模型的许多示例。在本章中，我们将示范如何把针对同步网络模型的算法转换为针对异步网络模型的算法；在第17章中，我们将示范如何把针对异步共享存储器模型的算法转换为针对异步网络模型的算法。由于存在这些转换，因此针对两种简单模型的算法能够在异步网络上运行。

第15章中的某些算法已经提及把针对同步网络模型的算法转换为针对异步网络模型的算法的想法，比如在15.2节中，我们使用所有消息上的轮数来模拟*FloodMax*算法，又如15.5.8节中*SimpleMST*算法的情况。

从同步网络模型算法到异步网络模型算法的转换只适用于非容错算法。事实上，如第21章所示，由于同步网络和异步网络的容错能力大相径庭，因此这种转换不适用于容错算法。

我们根据一个称为（局部）*同步器*的系统模块来表示从同步模型到异步模型的转换，然后描述同步器的几种分布式实现方法。这些实现方法涉及*在每一个同步轮中对系统进行同步*，这是必须的，因为转换的对象是任意同步算法。能否以更低频率进行同步（如*SimpleMST*算法的情况）取决于算法是否具有这样的特殊属性：如果在同步点之间进程步任意交错，算法能够正常工作。 531

同步器的实现是一个对分布式系统进行模块分解的极好例子。我们使用几种算法分解技巧，大部分技巧在第8章中描述。开始时我们以I/O自动机来表示“全局”正确性规格说明，然后抽象地定义一个实现全局规格说明的局部同步器，这种技巧基于事件的偏序关系。之后描述实现同步器的其他几种方法，每种方法都可以使用8.5.5节的模拟方法来证明其正确性。然而，大多数实现可以利用附加的分解步骤来进行。因此，我们定义另一个称为*安全同步器*的系统模块，并对该模块如何实现局部同步器进行示范，还开发出几个分布式算法来实现安全同步器。整个开发过程充分显示了分解手段在简化描述复杂的分布式算法方面的有效性。

结束部分给出用于异步网络上运行同步网络算法（假设同步性要求很强）时所需的时间开销的下界。

16.1 问题

本节描述将由同步器解决的问题。出发点是同步网络模型和由n个同步进程组成的集合，进程在无向图$G = (V, E)$的节点上运行，并在边上发送消息来进行通信。这种模型的形式化表示见第2章，其中每个进程都以一种带有消息生成函数和状态转移函数的状态自动机来表示。这里不沿用前面的表示方法，而是将每个进程都表示成为一个“用户进程”I/O自动机U_i。[⊖]

令M是在同步系统中使用的固定消息字母表。带标志信息被定义成(m,i)对，其中$m \in M$，$1 \leq i \leq n$。

用户自动机U_i具有形式为$user\text{-}send(T, r)_i$的输出动作，其中T是带标志消息集，$r \in \mathbb{N}^+$，自 532

⊖ 我们在这里称这些进程为“用户进程”，原因是它们是同步器系统（这是我们正在学习的主要的系统部分）的用户。

动机通过该动作来发送消息给它的邻居。标志消息中的标志指定消息的目的地，而r参数代表轮数。如果U_i在第r轮中没有消息要发送，那它就执行$user\text{-}send(\phi, r)_i$。$U_i$还有形式为$user\text{-}receive(T, r)_i$的输入动作，其中$T$是标志消息集，$r \in \mathbb{N}^+$，自动机通过该动作来接收邻居的消息。标志消息中的标志指出消息的来源，而r参数也代表轮数。U_i还可能有其他用于与外部世界交互的外部动作。与将输入和输出编码到状态中的做法（见第2章）不同，这里使用输入动作和输出动作来表示用户自动机的输入和输出。

例 16.1.1　*user-send*和*user-receive*动作

设n=4，$user\text{-}send(\{(m_1, 1), (m_2, 2)\},3)_4$指在第3轮中，用户$U_4$将消息$m_1$和$m_2$分别发送给用户$U_1$和$U_2$，且没有发送其他消息。另外，$user\text{-}receive(\{(m_1, 1), (m_2, 2)\},3)_4$指在第3轮中，用户$U_4$分别接收来自用户$U_1$和$U_2$的消息$m_1$和$m_2$，且没有接收其他消息。

U_i满足*良构性*条件：从$user\text{-}send_i$动作开始，$user\text{-}send_i$和$user\text{-}receive_i$动作交替出现，且动作对在轮中连续出现。即，该动作序列是以下形式的无限序列的前缀：

$$user\text{-}send(T_1, 1)_i, user\text{-}receive(T_1', 1)_i, user\text{-}send(T_2, 2)_i, user\text{-}receive(T_2', 2)_i,$$
$$user\text{-}send(T_3, 3), \ldots .$$

U_i还必须满足一个*活性*条件：在任意一个良构公平运行中，如果对于每个第r轮，其之前所有轮的$user\text{-}receive_i$事件都已经发生，那么U_i最终执行$user\text{-}send_i$。也就是说，只要系统保持回应，用户就继续在无数轮中发送消息。

将系统的剩余部分描述为*全局同步器*$GlobSynch$，其任务是在每轮中收集用户自动机在该轮中使用*user-send*动作发送的所有消息，并将这些消息传送给在该轮中使用了*user-receive*动作的所有用户自动机。它在每一轮的所有*user-send*事件之后、在所有*user-receive*事件之前进行全局同步。图16-1是一张用户和$GlobSynch$自动机的合成图，即$GlobSynch$*系统图*。注意：*user-send*动作是$GlobSynch$的输入动作，而*user-receive*是$GlobSynch$的输出动作。

533

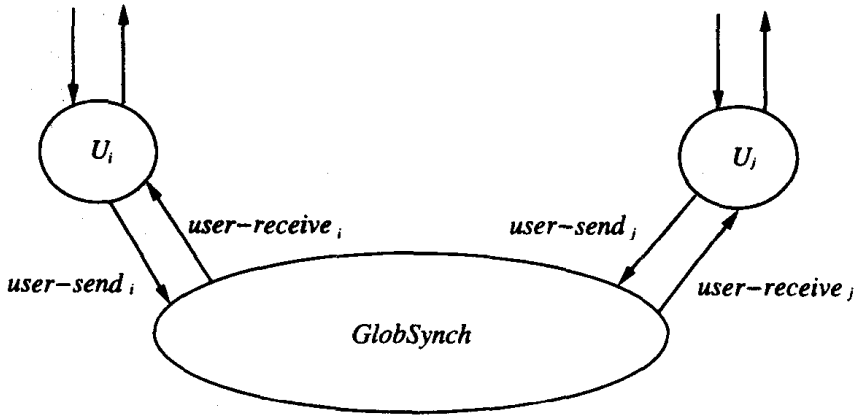


图16-1　$GlobSynch$系统的体系结构

可以容易地将$GlobSynch$表示为I/O自动机。

***GlobSynch* automaton:**

Signature:

Input:

$user\text{-}send(T, r)_i$, T a set of tagged messages, $r \in \mathbb{N}^+$, $1 \le i \le n$

Output:

$user\text{-}receive(T, r)_i$, T a set of tagged messages, $r \in \mathbb{N}^+$, $1 \le i \le n$

States:
tray, an array indexed by $\{1,\ldots,n\} \times \mathbb{N}^+$ of sets of tagged messages, initially all $\emptyset$
user-sent, *user-rcvd*, each an array indexed by $\{1,\ldots,n\} \times \mathbb{N}^+$ of Booleans, initially all *false*

Transitions:
user-send$(T,r)_i$
  Effect:
    user-sent$(i,r) := true$
    for all $j \neq i$ do
      $tray(j,r) := tray(j,r) \cup \{(m,i) | (m,j) \in T\}$

user-receive$(T,r)_i$
  Precondition:
    for all j
      user-sent$(j,r) = true$
    user-rcvd$(i,r) = false$
    $T = tray(i,r)$
  Effect:
    user-rcvd$(i,r) := true$

Tasks:
for every i, r:
  {*user-receive*$(T,r)_i$: T a set of tagged messages}

在上述代码中，*tray*(*i*, *r*)用来保存U_i的邻居提交给U_i的消息，这些消息中的标志是发送者的索引。*user-sent*和*user-rcvd*组件分别记录着*user-send*和*user-receive*事件的发生。 534

不难看出第2章中的同步网络模型上的任意算法都可以以新的风格——作为用户自动机U_i和*GlobSynch*自动机的合成来表示。具体证明留做一道习题。

同步器问题是：以图G中节点上的进程P_i和边(i, j)的每个方向上的可靠FIFO发送/接收通道$C_{i,j}$，使用一个异步网络算法来实现*GlobSynch*自动机。这种实现必须保证单个用户自动机U_i不能区分出来到底是在实现系统（即用户自动机加上分布式算法）中运行还是在*GlobSynch*系统中运行。也就是说，如果α是实现系统的一次公平运行，则*GlobSynch*系统中必有一个公平运行α'使得对于任意i，α与α'对于U_i是不可区分的[⊖]。

注意该要求只限于每个单独用户，我们并不要求不同用户的事件相对次序必须得到保持。我们将在16.6节中讨论这点。

16.2 局部同步器

所有同步器实现都是“局部”的，它们只涉及网络中邻居的同步而不是任意节点的同步。只使用局部同步器的优点在于能够节省通信和减少时间复杂度。本节定义一个*GlobSynch*的局部变种——*LocSynch*；我们将以*LocSynch*的实现来表达算法。

*LocSynch*几乎与*GlobSynch*一模一样，惟一的区别在于如下的*user-receive*转换：

***LocSynch* automaton:**

Transitions:
user-receive$(T,r)_i$
  Precondition:
    for all $j \in nbrs \cup \{i\}$
      user-sent$(j,r) = true$
    user-rcvd$(i,r) = false$
    $T = tray(i,r)$
  Effect:
    user-rcvd$(i,r) := true$

535

⊖ 这里采用8.7节中的“不可区分”定义，即两个运行投影后得出U_i的两个相同运行。

因此，在*LocSynch*中，只要已经从U_i及其邻居中接收到第r轮的消息，该消息就可以被发送给U_i，并没有必要等待来自整个网络中所有用户的消息。

引理 16.1　如果α是*LocSynch*系统（即用户加上*LocSynch*）的一个公平运行，则存在*GlobSynch*系统的一个公平执行α'，α和α'对于每个U_i都是不可区分的。

不同于对10.9节中*TicketME*的证明，这里不能像往常一样使用模拟技术来证明这条引理，因为在不同节点间发生的外部动作的相对次序在两个系统中是不同的。这里使用一种基于事件偏序关系的证明方法。

证明概要　令*L*和*G*分别代表*LocSynch*和*GlobSynch*系统，只不过*L*和*G*将用户自动机的所有内部动作归成输出动作（因此每个系统的外部动作由用户自动机的所有动作组成）。*L*的特定事件"依赖于"其他事件：*user-receive*事件依赖于同一节点或者邻居节点上的同一轮的*user-send*事件，用户自动机的任意事件可能依赖于同一自动机的以前事件。设β是*L*的任一轨迹，则定义β的事件的非自反偏序关系$\rightarrow_\beta$如下（这与在14.1.4节和14.2.4节中定义的依赖关系相似）。如果π和ϕ是β中的两个事件，π先于ϕ，则倘若以下条件成立，可以说$\pi \rightarrow_\beta \phi$，或$\phi$依赖于$\pi$。

1）π和ϕ是同一用户U_i的事件。

2）$\pi = user\text{-}send(T, r)_i$且 $\phi = user\text{-}receive(T', r)_j$，这里 $j \in nbrs_i$。

3）π和ϕ的关系遵循第1点和第2点的事件组成的关系链。

这些关系的关键为：$\rightarrow_\beta$关系充分捕捉了公平轨迹β的依赖关系，从而确保任意保持这些依赖关系的重排得到的仍是一个公平轨迹。（这与定理14.1和定理14.3类似。）

声明 16.2　如果β是一条*L*的公平轨迹，而γ是在保持$\rightarrow_\beta$次序的情况下通过重排β中事件的次序而得到的序列，则γ也是一条*L*的公平轨迹。

为了利用声明16.2来证明引理16.1，我们从*L*的任意公平运行α开始，令$\beta = trace(\alpha)$。重排β的事件来得到新轨迹γ，其中以轮为单位进行全局"排队"：特定轮r中所有*user-send*事件被放在同一轮的所有*user-receive*事件之前。新次序仍然满足$\rightarrow_\beta$，原因在于*user-send*事件和*user-receive*事件的次序并没有逆转。根据声明16.2，γ也是*L*的一条公平轨迹。另外，由于每个第r轮中所有*user-send*事件都处在同一轮的所有*user-receive*事件之前，不难证明γ是*G*的一条轨迹。为了完成证明，如同在α中填入用户状态一样，我们填充γ的状态以获取*G*的一条轨迹。形式上，这种填充可以使用与I/O自动机合成相关的通用定理（如定理8.4和定理8.5）来完成。 □

536

一个实现*LocSynch*的简单分布式算法如下：

***SimpleSynch*算法（非形式化）：**

在任意第r轮，在接收一个形式为$user\text{-}send(T, r)_i$的输入之后，进程$SimpleSynch_i$首先发送一个消息给每个邻居$SimpleSynch_j$，该消息中包含轮数r，以及在T中出现的从U_i到U_j的任何消息。当$SimpleSynch_i$接收完每个邻居的第r轮消息之后，它输出$user\text{-}receive(T', r)_i$，其中$T'$是收到的消息集合，而每个消息都带有发送者的标志。

更加形式化一些，$SimpleSynch_i$是如下的自动机：

***SimpleSynch*$_i$ automaton (formal):**

Signature:

Input:

$user\text{-}send(T, r)_i$, T a set of tagged messages, $r \in \mathbb{N}^+$

$receive(N, r)_{j,i}$, N a set of messages, $r \in \mathbb{N}^+$, $j \in nbrs$

Output:
$user\text{-}receive(T,r)_i$, T a set of tagged messages, $r \in \mathbb{N}^+$
$send(N,r)_{i,j}$, N a set of messages, $r \in \mathbb{N}^+$, $j \in nbrs$

States:
user-sent, *user-rcvd*, each a vector indexed by $\mathbb{N}^+$ of Booleans, initially all *false*
pkt-sent, *pkt-rcvd*, each an array indexed by $nbrs \times \mathbb{N}^+$ of Booleans, initially all *false*
outbox, an array indexed by $nbrs \times \mathbb{N}^+$ of sets of messages, initially all $\emptyset$
inbox, a vector indexed by $\mathbb{N}^+$ of sets of tagged messages, initially all $\emptyset$
Transitions:

$user\text{-}send(T,r)_i$
Effect:
$user\text{-}sent(r) := true$
for all $j \in nbrs$ do
$outbox(j,r) := \{m | (m,j) \in T\}$

$send(N,r)_{i,j}$
Precondition:
$user\text{-}sent(r) = true$
$pkt\text{-}sent(j,r) = false$
$N = outbox(j,r)$
Effect:
$pkt\text{-}sent(j,r) := true$

$receive(N,r)_{j,i}$
Effect:
$inbox(r) := inbox(r) \cup \{(m,j) | m \in N\}$
$pkt\text{-}rcvd(j,r) := true$

$user\text{-}receive(T,r)_i$
Precondition:
$user\text{-}sent(r) = true$
for all $j \in nbrs$
$pkt\text{-}rcvd(j,r) = true$
$T = inbox(r)$
$user\text{-}rcvd(r) = false$
Effect:
$user\text{-}rcvd(r) := true$

Tasks:
for every r:
$\{user\text{-}receive(T,r)_i : T$ a set of tagged messages$\}$
for every $j \in nbrs$ and every r:
$\{send(N,r)_{i,j} : N$ a set of messages$\}$

537

*SimpleSynch*系统由$SimpleSynch_i$ 进程、所有边的可靠FIFO发送/接收通道$C_{i,j}$和用户合成而得，见图16-2。

538

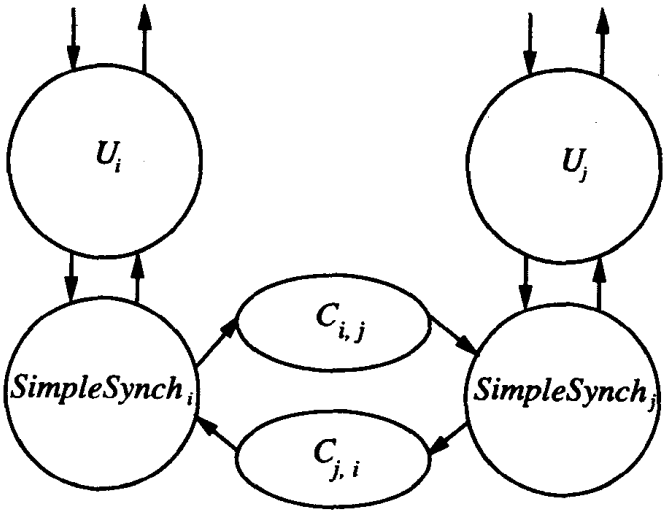


图16-2 *SimpleSynch*系统的体系结构

引理16.3　如果α是*SimpleSynch*系统的任一公平运行，则存在*LocSynch*系统的一个公平运行α'，使得α与α'对于每个U_i都是不可区分的。

证明概要　不同于引理16.1的证明，这次不用重排不同用户的事件，而是使用模拟方法来证明。令*S*和*L*分别代表*SimpleSynch*和*LocSynch*系统，它们被略微改动使得被归为外部动作的那些动作刚好是用户自动机的所有动作。（也就是说，用户的内部动作被重新归为输出动作，而*send*和*receive*动作被隐藏——重新归为内部动作。）令*s*和*u*分别是*S*和*L*的状态，如果以下各点成立，则称$(s, u) \in f$：

1）s的所有用户状态和u的所有用户状态相同

2）$u.user\text{-}sent(i, r) = s.user\text{-}sent(r)_i$

3）$u.user\text{-}rcvd(i,r) = s.user\text{-}rcvd(r)_i$

4）$u.tray(i, r) = \cup_{j \neq i}\{(m, j) : m \in s.outbox(i,r)_j\}$

为了证明*f*是一种模拟关系，我们需要以下关于*S*的不变式断言。

断言16.2.1　在*SimpleSynch*系统的任意可达状态中，如果$pkt\text{-}rcvd(j, r)_i = true$，则

1）$user\text{-}sent(r)_j = true$

2）$\{m : (m, j) \in inbox(r)_i\} = outbox(i, r)_j$

这个不变式的证明用到其他中间不变式，并涉及对转换中消息的正确性证明（如前，我们假定在这些不变式的声明和证明中的通道都是通用可靠FIFO通道）。给定断言16.2.1，证明*f*是一种模拟关系是直截了当的；惟一有趣的是对*user-receive*的证明需要利用断言16.2.1。我们将不变式和模拟证明的细节留做一道习题。

模拟关系的存在意味着*S*的任意轨迹都是*L*的一条轨迹（如前所述，包含在这些轨迹中的动作是自动机的动作）。但我们需要进一步确认*S*的公平性条件可以导出*L*的公平性条件。我们来证明$fairtraces(S) \subseteq fairtraces(L)$，然后应用关于I/O自动机的通用合成定理（定理8.4和定理8.5）
539 来填充用户状态，从而获得所需的运行之间的关系。

为了证明公平轨迹的包含关系，我们利用模拟关系的一个事实：除了保证轨迹包含之外，模拟关系还能保证运行之间的对应关系，这种关系曾在节8.5.5中定义。令$\beta \in fairtraces(S)$，α是*S*的满足$\beta = trace(\alpha)$的任意运行。则定理8.13意味着存在*L*的一个运行α'与α对应于*f*。我们称α'是*L*的一个公平运行。

有两种情况将导致不公平的发生。在第一种情况中，可能存在一个在α'的某点中使能的用户任务，该任务的步没有发生在α'以后的点中。这样根据对应关系，该任务在α的某点中也是使能的，但其任务步没有发生，这与α的公平性相矛盾。

在第二种情况中，可能存在*i*和*r*使得第*r*轮的$user\text{-}receive_i$任务在α'的某点中是使能的，但该任务的步没有发生。这意味着，从α'的给定点开始，对于所有$j \in nbrs_i \cup \{i\}$，有$user\text{-}sent(j, r) = true$和$user\text{-}rcvd(i, r) = false$。则对应关系意味着从$\alpha$的对应点开始，对于所有$j \in nbrs_i \cup \{i\}$，有$user\text{-}sent(r)_j = true$和$user\text{-}rcvd(r)_i = false$。

我们使用以下断言。

断言 16.2.2　在*SimpleSynch*系统的任意可达状态中，若$pkt\text{-}sent(i, r)_j = true$，则通道$C_{j,i}$包含一个消息或者$pkt\text{-}rcvd(j, r)_i = true$。

然后，对于任意$j \in nbrs_i$，*send*任务在第*r*轮的公平性意味着在α中，$pkt\text{-}sent(j, r)_i$最终变为*true*。而$user\text{-}receive_i$任务在*S*中第*r*轮的公平性意味着该任务的一步最终会在α中发生，如此，由对应关系可知该任务的一步最终也会在α'中发生。矛盾。　□

注意引理16.3的证明实际上揭示了$fairtraces(S) \subseteq fairtraces(L)$，并揭示了对单独用户的不可区分性。从引理16.1和引理16.3可以导出：

定理 16.4 如果α是*SimpleSynch*系统的任一公平运行，则存在*GlobSynch*系统的一个公平运行α'，α和α'对于U_i是不可区分的。

复杂性分析 每轮需要$2|E|$个消息，因为图中每条边的每个方向都有一个消息。假设c是在更小轮中所有$user\text{-}receive_i$事件发生之后到任一$user\text{-}send_i$事件发生为止之间的时间上界，l是任意进程的任意任务的时间上界，d是在任意通道中传送最早消息的时间的上界，则模拟r轮所需的总时间最多为$r(c+d+O(l))$。 540

16.3 安全同步器

尽管明显地减少*SimpleSynch*算法的时间复杂度是不可能的，但有可能减少其通信复杂度。换而言之，如果原来同步算法的第r轮中没有从U_i到邻居U_j的消息，则也许可以在异步算法中忽略第r轮中从进程i到进程j的消息。在执行第r轮的*user-receive*输出之前，进程需要确定自己已经收到邻居会在第r轮发给它的所有消息。在*SimpleSynch*算法中，进程既利用消息来做出这种确定，又要传送用户的消息。减少通信的基本策略在于将这两种功能分开。

因此，我们将*LocSynch*的实现分为几个部分：每个节点都有一个“前端”*FrontEnd*，前端通过特定的通道$D_{i,j}$与邻居节点的*FrontEnd*通信；另外还有一个“安全同步器”*SafeSynch*。图16-3显示这种新的体系结构。$FrontEnd_i$的任务是传递那些从本地用户U_i接收过来的在$user\text{-}send_i$事件中的消息。在特定的第r轮中，在接收到$user\text{-}send_i$后，$FrontEnd_i$将所有要发送的消息排序后放到“发件箱”中，然后将发件箱中的非空项经由通道$D_{i,j}$发送给相应的邻居j，之后等待$D_{j,i}$的确认。若$FrontEnd_i$收到所有消息的确认，则称它为安全的，这意味着进程i的所有消息都已被相应邻居的*FrontEnd*接收。同时，$FrontEnd_i$收集并确认其邻居*FrontEnd*发来的消息。

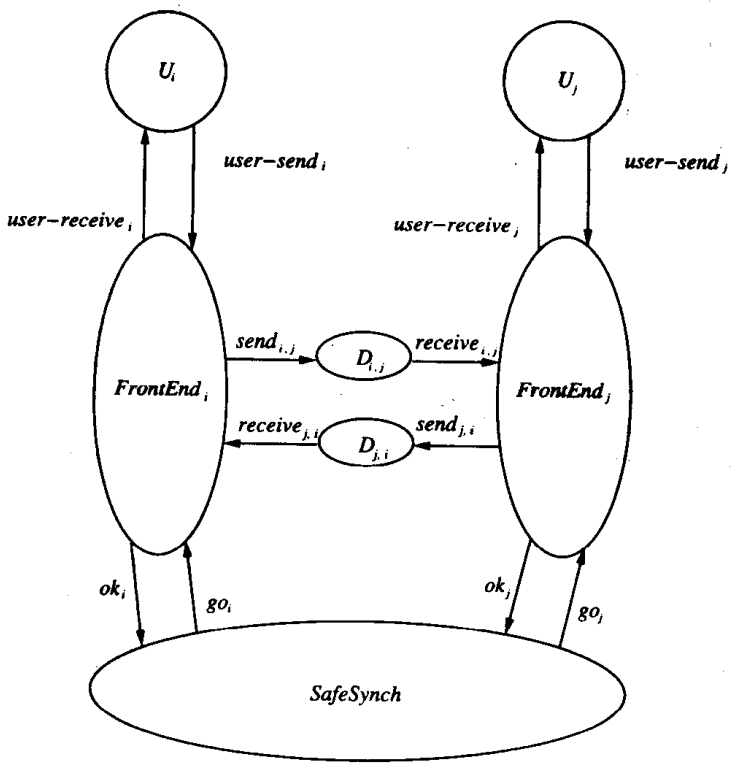


图16-3 使用*SafeSynch*对*LocSynch*的分解

在第r轮中，$FrontEnd_i$应在何时执行$user\text{-}receive_i$呢？即应何时将U_i从邻居中收集到的第r轮消息传送给邻居呢？只有当它知道自己已经收到所有第r轮的消息后，$FrontEnd_i$才能这样做。如果$FrontEnd_i$确定它的所有邻居$FrontEnd_j$在第r轮中都是安全的，那就足够了，此时所有邻居都知道自己的所有第r轮消息都已经被相应的$FrontEnd$自动机接收。

因此，安全同步器自动机$SafeSynch$的任务是告诉每个$FrontEnd$自动机它的邻居是安全的。为此，$SafeSynch$具有ok输入动作和$FrontEnd$自动机的输出，$FrontEnd$自动机通过这些动作来告诉$SafeSynch$它们是安全的。在接收到进程i的所有邻居和进程i本身发来的ok后，$SafeSynch$就发
541 送go_i给 $FrontEnd_i$。$FrontEnd_i$接收到go_i后，可以执行$user\text{-}receive_i$。在本节的余下部分中，将更为详细地描述这种分解。

16.3.1 前端自动机

$FrontEnd_i$ automaton:

Signature:

Input:

$user\text{-}send(T,r)_i$, T a set of tagged messages, $r \in \mathbb{N}^+$

$receive(\text{“}msgs\text{”}, N, r)_{j,i}$, N a set of messages, $r \in \mathbb{N}^+$, $j \in nbrs$

$receive(\text{“}ack\text{”}, r)_{j,i}$, $r \in \mathbb{N}^+$, $j \in nbrs$

542 $go(r)_i$, $r \in \mathbb{N}^+$

Output:

$user\text{-}receive(T,r)_i$, T a set of tagged messages, $r \in \mathbb{N}^+$

$send(\text{“}msgs\text{”}, N, r)_{i,j}$, N a set of messages, $r \in \mathbb{N}^+$, $j \in nbrs$

$send(\text{“}ack\text{”}, r)_{i,j}$, $r \in \mathbb{N}^+$, $j \in nbrs$

$ok(r)_i$, $r \in \mathbb{N}^+$

States:

user-sent, *user-rcvd*, each a vector indexed by $\mathbb{N}^+$ of Booleans, initially all *false*

pkt-for, *pkt-sent*, *pkt-rcvd*, *ack-rcvd*, each an array indexed by $nbrs \times \mathbb{N}^+$ of Booleans, initially all *false*

ack-sent, an array indexed by $nbrs \times \mathbb{N}^+$ of Booleans, initially all *false*

outbox, an array indexed by $nbrs \times \mathbb{N}^+$ of sets of messages, initially all $\emptyset$

inbox, a vector indexed by $\mathbb{N}^+$ of sets of tagged messages, initially all $\emptyset$

ok-given, *go-seen*, each a vector indexed by $\mathbb{N}^+$ of Booleans, initially all *false*

Transitions:

$user\text{-}send(T,r)_i$

Effect:

$user\text{-}sent(r) := true$

for all $j \in nbrs$ such that $\exists m, (m,j) \in T$ do

$outbox(j,r) := \{m | (m,j) \in T\}$

$pkt\text{-}for(j,r) := true$

$send(\text{“}msgs\text{”}, N, r)_{i,j}$

Precondition:

$pkt\text{-}sent(j,r) = false$

$pkt\text{-}for(j,r) = true$

$N = outbox(j,r)$

Effect:

$pkt\text{-}sent(j,r) := true$

$receive(\text{“}ack\text{”}, r)_{j,i}$

Effect:

$pkt\text{-}rcvd(j,r) := true$

$send(\text{“}ack\text{”}, r)_{i,j}$

Precondition:

$pkt\text{-}rcvd(j,r) = true$

$ack\text{-}sent(j,r) = false$

Effect:

$ack\text{-}sent(j,r) := true$

$ok(r)_i$

Precondition:

$user\text{-}sent(r) = true$

for all $j \in nbrs$

if $pkt\text{-}for(j,r) = true$ then

$ack\text{-}rcvd(j,r) = true$

$ok\text{-}given(r) = false$

Effect:

$ok\text{-}given(r) := true$

Effect:
$ack\text{-}rcvd(j,r) := true$

$receive(\text{“}msgs\text{”}, N, r)_{j,i}$
Effect:
$inbox(r) := inbox(r) \cup \{(m,j) | m \in N\}$
$pkt\text{-}rcvd(j,r) := true$

$go(r)_i$
Effect:
$go\text{-}seen(r) := true$

$user\text{-}receive(T,r)_i$
Precondition:
$go\text{-}seen(r) = true$
$T = inbox(r)$
$user\text{-}rcvd(r) = false$
Effect:
$user\text{-}rcvd(r) := true$

543

Tasks:
for every r:
$\{user\text{-}receive(T,r)_i : T$ a set of tagged messages$\}$
$\{ok(r)_i\}$
for every j and every r:
$\{send(\text{“}msgs\text{”}, N, r)_{i,j} : N$ a set of messages$\}$
$\{send(\text{“}ack\text{”}, r)_{i,j}\}$

16.3.2 通道自动机

每对前端自动机$FrontEnd_i$和$FrontEnd_j$使用两个通道$D_{i,j}$和$D_{j,i}$进行通信。根据14.1.2节中的定义，从i到j和从j到i的通道都是可靠发送/接收通道。

16.3.3 安全同步器

安全同步器*SafeSynch*的全部任务是：等待$FrontEnd_i$的所有邻居和$FrontEnd_i$本身发来ok，然后执行go_i。

***SafeSynch* automaton:**

Signature:

Input:
$ok(r)_i$, $r \in \mathbb{N}^+$, $1 \le i \le n$

Output:
$go(r)_i$, $r \in \mathbb{N}^+$, $1 \le i \le n$

States:
ok-seen, *go-given*, each an array indexed by $\{1,\ldots,n\} \times \mathbb{N}^+$ of Booleans, initially all *false*

Transitions:
$ok(r)_i$
Effect:
$ok\text{-}seen(i,r) := true$

$go(r)_i$
Precondition:
for all $j \in nbrs_i \cup \{i\}$
$ok\text{-}seen(j,r) = true$
$go\text{-}given(i,r) = false$
Effect:
$go\text{-}given(i,r) := true$

Tasks:
for every i, r:
$\{go(r)_i\}$

544

16.3.4 正确性

引理 16.5 如果α是*SafeSynch*系统（即图16-3中的*FrontEnd*、通道、*SafeSynch*和用户自动机）

的任一公平运行，则存在*LocSynch*系统的一个公平运行α'，α和α'对每个U_i都是不可区分的。

证明概要　使用从*SafeSynch*系统到*LocSynch*系统的模拟关系来证明，证明策略与引理16.3（该引理针对*SimpleSynch*算法）的证明策略相同，都使用同样的模拟关系f。但由于本算法较为复杂，故具体的证明也更为复杂。同样地，该模拟证明中惟一有趣的是*user-receive*动作的情况，其中需要用到以下的不变式断言。

断言16.3.1　在*SafeSynch*系统的所有可达状态中，如果$go\text{-}seen(r)_i = true$，则对于所有$j \in nbrs_i$，有

1）$user\text{-}sent(r)_j = true$

2）$\{m : (m, j) \in inbox(r)_i\} = outbox(i, r)_j$

该断言的证明需要用到一些辅助的不变式，例如

断言 16.3.2　在*SafeSynch*系统的所有可达状态中，如果$ok\text{-}seen(j, r) = true$，则[⊖]

1）$user\text{-}sent(r)_j = true$

2）对于所有$i \in nbrs_j$，$\{m : (m, j) \in inbox(r)_i\} = outbox(i, r)_j$

进一步的证明细节留给读者去做。

从引理16.1和引理16.5可以导出：　□

引理 16.6　如果α是*SafeSynch*系统的任一公平运行，则存在*GlobSynch*系统的一个公平运行α'，α和α'对任意U_i都是不可区分的。

剩下的任务是使用分布式算法来实现*SafeSynch*自动机。在以下的节中描述几种实现方法。另外还要使用实际的发送/接收通道$C_{i,j}$来实现$D_{i,j}$通道，这可以通过“复用”$C_{i,j}$来完成：$C_{i,j}$既用
545 来实现*SafeSynch*的分布式实现中的通道，又用来实现$D_{i,j}$通道。复用策略在习题14.6中描述。

16.4 安全同步器的实现

本节给出几种以分布式算法描述*SafeSynch*实现的方法。主要的两种实现是*Alpha*和*Beta*，还有一种将*Alpha*和*Beta*混合而得的混合实现*Gamma*。

如前所述，安全同步器*SafeSynch*的全部任务是：等待$FrontEnd_i$的所有邻居和$FrontEnd_i$本身发来ok，然后执行go_i。

16.4.1 同步器*Alpha*

*SafeSynch*的最简单实现是*Alpha*同步器，工作原理如下：

***Alpha*同步器：**

当进程$Alpha_i$在任意轮r接收到一个ok_i时，它将该消息发送给所有邻居。当$Alpha_i$听到所有邻居都已经收到第r轮的ok且$Alpha_i$本身也收到第r轮的ok时，$Alpha_i$输出go_i。

对每个$Alpha_i$的前提-结果代码的编写任务留给读者；代码的结构有点类似于$SimpleSynch_i$的结构。正确性——安全性和活性——易于证明，只需使用将*Alpha*系统（$Alpha_i$、$FrontEnd_i$、$D_{i,j}$和用户自动机）与*SafeSynch*系统关联起来的模拟技术。[⊜] 可得

⊖ 回想$ok\text{-}seen$是*SafeSynch*组件的状态的一部分。

⊜ 这个策略看来并不可能非常模块化，因为相同的用户（$FrontEnd_i$和$D_{i,j}$自动机）出现在两个系统中。然而，可以用平常的方式处理它们，保持模拟关系不变。一种替代的方法可以是为*SafeSynch*自动机制定一个更抽象（和更普通）的环境。

定理 16.7 如果α是*Alpha*系统的任一公平运行，则存在*GlobSynch*系统的一个公平运行α'，α和α'对于每个U_i都是不可区分的。

复杂度分析 我们分析整个系统的复杂度。通信复杂度依赖于原来同步算法发送的消息的个数：如果同步算法在第r轮共发送m个非空消息，则为了模拟r轮，*Alpha*系统最多发送$2m+2r|E|$个消息，其中$2m$来自*FrontEnds*发送的*msgs*和 *ack*消息，而$2r|E|$来自*Alpha*本身发送的消息——在每一轮中的每条边的每个方向都有一个消息。 546

如果c、l和d采用*SimpleSynch*算法中的定义，则模拟r轮所需的总时间最多为$r(c+3d+O(l))$。（这考虑了原来通道中的堆积。）因此，*Alpha*的通信复杂度和时间复杂度都比*SimpleSynch*的高。

与*SimpleSynch*相似，*Alpha*的时间复杂度较低，但通信复杂度高。在以下的子节中，我们将给出具有更好通信复杂度但更高时间复杂度的另一*SafeSynch*实现。

16.4.2 同步器*Beta*

同步器*Beta*假设图G存在一棵带根的生成树，树的高度越低越好。

***Beta*同步器：**

在第r轮，所有进程都通过生成树的边将*ok*消息汇给根，在根收集到所有进程的*ok*消息之后，它沿着生成树的边广播许可消息，许可消息允许进程执行*go*输出。

同样地，对*Beta*中每个$Beta_i$进程的前提-结果代码的编写任务留给读者；其思想类似于15.3节中的广播和汇总思想。正确性易于证明，只需使用将*Beta*系统与*SafeSynch*系统关联起来的模拟技术。

定理 16.8 如果α是*Beta*系统（$Beta_i$、*FrontEnd*、$D_{i,j}$和用户自动机）的任意公平运行，则存在*GlobSynch*系统的一个公平运行α'，α和α'对于每个U_i都是不可区分的。

复杂度分析 如果原来的同步算法在r轮中共发送m个非空消息，则为了模拟r轮，*Beta*系统最多发送$2m+2rn$个消息，其中$2m$来自*Alpha*，而$2rn$来自广播和敛播消息。故模拟r轮所需的总时间最多为$r(c+2d+O(l)+2h(d+O(l)))$，或$r(c+O(hd)+O(hl))$。 547

16.4.3 同步器*Gamma*

我们将同步器*Alpha*和*Beta*的思想混合起来得到混合算法*Gamma*，*Gamma*（依赖于图G的结构）既具有*Alpha*算法在时间上的优点又具有*Beta*在通信上的优点。

算法*Gamma*假定G的一棵生成森林，森林中每个树都是带根的。称每棵树为一个簇（*cluster*），簇C的节点集合为$nodes(C)$。（如何构建一棵合适的生成树是一个有趣的问题，但在这里不作讨论。）在*Gamma*中，簇内节点以一个*Beta*算法来同步，而簇之间则以一个*Alpha*算法来同步。

在每个簇只包含一个节点的极端情况中，*Gamma*等同于*Alpha*；在只有一个簇的情况下，*Gamma*等同于*Beta*。在其他情况中，*Gamma*的通信复杂度和时间复杂度介于*Alpha*的复杂度和*Beta*的复杂度之间。

例 16.4.1 簇分解

设有一个由p个完全图组成的网络图G，其中每个完全图的节点数为k。将完全图排列成行，相邻两个完全图的所有节点互相连接。图16-4显示$p=5$和$k=4$的情形（图中的一些边在其他边之“下”，故不可见）。现在考虑图16-5中G的簇分解。

分解后的每个簇C都是G中某个k节点完全图的一棵树。每棵簇树的根是在顶部的节点。算法*Gamma*使用一个*Beta*算法来同步每个k节点树内的节点而使用一个*Alpha*算法来同步p棵树。 548

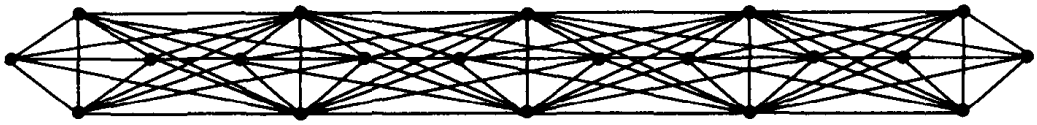


图16-4　网络图G

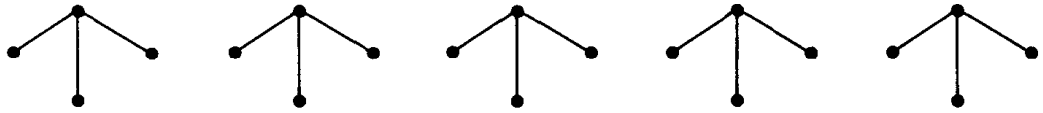


图16-5　G的簇分解

鉴于$Gamma$是两个算法的混合，我们先进行高层分解，将$SafeSynch$为两种自动机：$ClusterSynch$自动机和$ForestSynch$自动机。每个簇C_k都有一个$ClusterSynch_k$自动机，而系统还有一个单独的$ForestSynch$自动机。其体系结构如图16-6所示。

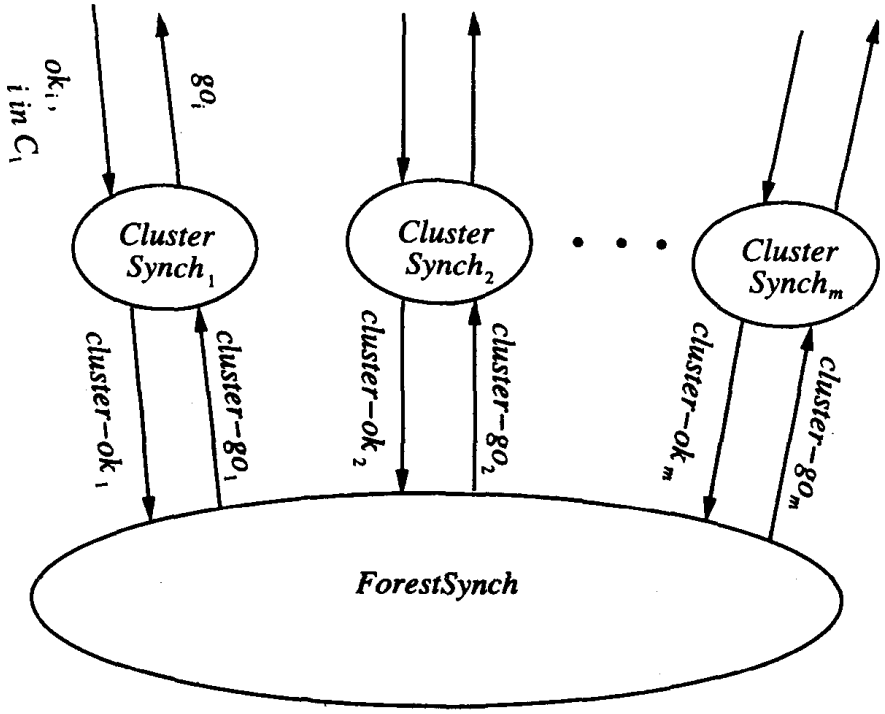


图16-6　将$SafeSynch$分解为$ClusterSynch$自动机和$ForestSynch$自动机

对于每个簇C_k和每轮r，自动机$ClusterSynch_k$有两个任务：第一个是在接收到C_k中任意节点i的输入ok_i之后，输出单个$cluster\text{-}ok_k$给$ForestSynch$；第二个（在互不依赖的任务中）是当一个$cluster\text{-}go_k$输入到达$ForestSynch$时，$ClusterSynch_k$为C_k中的每个节点i产生一个go_i。任务的合成
549 类似于$Beta$的行为，以抽象自动机表达：

$ClusterSynch_k$ automaton:

Signature:

Input:
$ok(r)_i$, $r \in \mathbb{N}^+$, $i \in nodes(C_k)$
$cluster\text{-}go(r)_k$, $r \in \mathbb{N}^+$

Output:
$go(r)_i$, $r \in \mathbb{N}^+$, $i \in nodes(C_k)$
$cluster\text{-}ok(r)_k$, $r \in \mathbb{N}^+$

States:
ok-seen, *go-given*, each an array indexed by $nodes(C_k) \times \mathbb{N}^+$ of Booleans, initially all *false*
cluster-ok-given, *cluster-go-seen*, each a vector indexed by $\mathbb{N}^+$ of Booleans, initially all *false*

Transitions:
$ok(r)_i$
Effect:
$ok\text{-}seen(i,r) := true$

$cluster\text{-}ok(r)_k$
Precondition:
for all $i \in nodes(C_k)$
$ok\text{-}seen(i,r) = true$
$cluster\text{-}ok\text{-}given(r) = false$
Effect:
$cluster\text{-}ok\text{-}given(r) := true$

$cluster\text{-}go(r)_k$
Effect:
$cluster\text{-}go\text{-}seen(r) := true$

$go(r)_i$
Precondition:
$cluster\text{-}go\text{-}seen(r) = true$
$go\text{-}given(i,r) = false$
Effect:
$go\text{-}given(i,r) := true$

Tasks:
for every r:
$\{cluster\text{-}ok(r)_k\}$
for every i, r:
$\{go(r)_i\}$

*ForestSynch*自动机（外部动作改名之后）是G的簇图G'上的安全同步器，其中G'的节点对应于G的簇，如果G中有一条从C_k的某点到C_l的某点的边，则G'中存在一条从C_k到C_l的边。*ClusterForest*系统由*ClusterSynch*、*ForestSynch*、*FrontEnd*、$D_{i,j}$和用户自动机组成。

引理 16.9 如果α是*ClusterForest*系统的任一公平运行，则存在*SafeSynch*系统的一个公平执行α'，α和α'对于任意U_i都是不可区分的。

550

证明概要 可以使用模拟证明方法，但是为了多样性，我们给出基于运行的运算证明方法。证明的关键在于：对于任意$j \in nbrs_i \cup \{i\}$，如果$go(r)_i$发生，则在此之前$ok(r)_j$必然发生。分两种情况：

1）i和j在同一簇C_k中（有可能$i = j$）。
根据$ClusterSynch_k$的代码，在$go(r)_i$之前必有一个$cluster\text{-}go(r)_k$出现。由*ForestSynch*的定义可知，在$cluster\text{-}go(r)_k$之前，必有一个$cluster\text{-}ok(r)_k$出现，这又意味着之前有一个$ok(r)_j$出现，得证。

2）i在簇C_k中，而j在簇C_l中，$k \neq l$。
由于$j \in nbrs_i$，簇C_k和簇C_l在簇图G'中必为邻居（参考对簇图中邻居簇的定义）。如前，在$go(r)_i$之前必有一个$cluster\text{-}go(r)_k$出现。根据*ForestSynch*的定义，在$cluster\text{-}go(r)_k$之前，必有一个$cluster\text{-}ok_l$出现。这又意味着之前有一个$ok(r)_j$出现。 □

为了完成对同步器*Gamma*的描述，我们示范如何用分布式算法来实现*ForestSynch*自动机和*ClusterSynch*自动机。我们可以使用在带根树C_k上的同步器*Beta*的变种来实现$ClusterSynch_k$。也就是说，先执行敛播，根先收集*ok*消息，然后输出*cluster-ok*消息。根还接收*cluster-go*，然后对$nodes(C_k)$中的所有节点进行广播，通知它们执行*go*。（实际上，我们以两个独立的自动机来形式化这两种操作。）

*SafeSynch*的任一实现经过适当的改名后都可用来实现*ForestSynch*，这里选用同步器*Alpha*。这里有一个技术难点：我们不能将*Alpha*直接运行在给定的分布式网络上，因为*Alpha*运行在那

些参与同步的实体（这里是整个簇）相对应的进程上，并使用那些与相邻实体（这里是簇）之间的边相对应的通道。给定的模型只允许使用与G中节点和边对应的进程和通道。然而，不难实现所需的进程和通道：将每个簇的进程运行在簇的根节点上，并使用簇的根之间的指定路径来模拟相邻簇的进程之间的直接通信。这样的路径是存在的，因为簇是相连的，且两簇中有些
551 节点在G中是相邻的。同样地，需要采用预处理来确定这些路径，但在这里我们不作讨论。*cluster-ok*和*cluster-go*动作被实现成为在簇的根节点上的进程的内部动作。

例 16.4.2　实现*Alpha*

考虑例16.4.1中的网络图G和簇分解。针对这种图及其分解，我们在每个簇的根节点（图16-5中的顶部节点）上运行*Alpha*进程。*Alpha*进程与相邻簇的通信可以用原来图 G（图16-4）中簇的根节点之间的有向边来模拟。

在*Gamma*的完整实现中，与G中节点i相关的进程是三个进程的合成：$FrontEnd_i$、*ClusterSynch*实现中的进程i、*ForestSynch*实现中的进程i。每个通道$C_{i,j}$用于实现三个通道：$D_{i,j}$、*ClusterSynch*实现中的从i到j的通道、*ForestSynch*实现中从i到j的通道。*Gamma*系统被定义为整个实现，可以使用模拟技术来证明以下定理：

定理 16.10　如果α是*Gamma*系统的任一公平运行，则存在*GlobSynch*系统的一个公平运行α'，α和α'对于每个U_i都是不可区分的。

正交分解　完整的*Gamma*系统有两种自然的分解：第一种是基于功能（数据通信、簇同步和森林同步）的逻辑分解；另一种是基于完整实现中进程和通道的空间分解。这两种分解方法与组成算法的原始I/O自动机的不同合成顺序相对应。由于合成操作是可结合的，因此两种分解方法的结果相同。

复杂度分析　令所有簇树的最大高度为h，用于根间通信的所有路径的总边数为e'。如果原来同步算法在r轮中发送了总数最多为m的非空消息，则*Gamma*系统发送了总数最多为$2m+O(r(n+e'))$的消息，其中$O(rn)$来自*ClusterSynch*实现中所有簇树内发送的消息；$O(re')$来自*ForestSynch*实现中根之间发送的消息。模拟r轮所需的时间为$O(r(c+O(hd)+O(hl)))$。如果$n+e' <<$
552 $|E|$，则*Gamma*的消息数比*Alpha*的消息数少，如果单个簇生成树的最大高度比整个网络的生成树的高度小，则*Gamma*比*Beta*省时间。

例 16.4.3　*Alpha*、*Beta*和*Gamma*的复杂度比较

对于例16.4.1中的网络图G及其簇分解，我们对给出的三种安全同步器实现的代价进行比较。代价的单位为轮。忽略用户、*FrontEnds*和$D_{i,j}$引起的代价，因为在三种算法中它们都是一样的。忽略局部处理时间。对于*Beta*，设树具有最小可能的高度，近似为p。

	消息	时间
Alpha:	$O(pk^2)$	$O(d)$
Beta:	$O(pk)$	$O(pd)$
Gamma:	$O(pk)$	$O(d)$

如果p和k近似相等，与*Alpha*和*Beta*相比，*Gamma*代表一种数量级提高。

16.5　应用

前面给出的同步器算法允许无错异步网络实现任意的无容错同步网络算法（同步器不适用于第6章中的容错算法）。在本节中，我们给出使用同步器建造异步算法的几个例子。

如前所述，在本章中我们只考虑无向网络，在本节的所有分析中，我们均忽略局部进程步时间。

16.5.1 领导者选举

使用同步器后，诸如*LCR*和*HS*等同步环领导选举算法可以在异步环上运行。但这并不吸引人，因为这些算法已经在异步网络上工作（不需要同步器）且没有同步器引起的开销。 553

在基于任意无向图（已知直径为*diam*）的异步网络上，同步器可用来运行*FloodMax*同步领导选举算法。使用同步器*Alpha*时，所得算法发送$O(|E| \cdot diam)$个消息并需要$O(diam \cdot d)$的时间来模拟所需的*diam*个同步轮。

与*FloodMax*算法相似，*OptFloodMax*同步领导者选举算法也可以使用同步器来运行，只不过它的节点只在有新消息要发送的时候才发送消息。如果使用同步器*Alpha*，则优化带来的好处将消失殆尽，因为同步器本身在所有轮都发送消息给全部通道。另一方面，倘若使用同步器*Beta*，则通信复杂度是相当低的（在额外时间的花费方面）。

16.5.2 深度优先搜索

在直径为*diam*的网络上，4.2节中的*SynchBFS*算法需要发送$O(|E/)$个消息和执行$O(diam)$轮；其中进程们并不需要知道*diam*的值。使用同步器，*SynchBFS*算法能够在异步网络上运行。采用同步器*Alpha*时，算法发送$O(|E/) \cdot diam)$个消息并需要$O(diam \cdot d)$的时间来模拟*diam*轮，其中所有进程输出其父亲消息。采用*Beta*算法（树的高最多为*diam*）时，与15.4节中的*LayeredBFS*算法一样，算法只需发送$O(|E/+n \cdot diam)$个消息和需要$O(diam^2 \cdot d)$的时间。也可以采用*Gamma*来在增加通信复杂度的情况下减少算法的时间复杂度。

这里有一个技术难点：使用同步器而得的*BFS*算法是如何结束的？如前所述，算法的实现不停地模拟轮，并产生无限多的消息（如果进程得知*diam*的值，则它们会在模拟*diam*轮之后停止，但我们这里假设进程并不知道*diam*的值）。一种特殊的解法是令每个决定父亲的用户自动机只执行额外一轮来通知其邻居，然后停止。

16.5.3 最短路径

对于寻找从指定点开始的最短路径问题，同步器的使用非常有利。如前所述，*AsynchBellmanFord*算法的消息和时间复杂度是节点数的指数。然而，对于已知尺寸为*n*的网络， 554
同步*BellmanFord*算法只有值为$O(n|E|)$的通信复杂度和值为$O(n)$的轮复杂度。我们可以采用同步器*Alpha*来运行*BellmanFord*算法，从而得到一个发送$O(n|E|)$个消息和使用$O(nd)$的时间来模拟所需*n*轮的算法；也可以采用同步器*SimpleSynch*。

16.5.4 广播与确认

可以设计一个允许进程广播消息到其他所有进程并接收其他进程的确认的同步算法，该算法发送$O|E|$个消息并执行$O(diam)$轮（见习题4.8）。可以采用同步器*Alpha*来运行这个算法，从而得到一个使用$O(|E| \cdot diam)$个消息和$O(diam \cdot d)$轮时间的广播与确认算法。比较该算法与15.3节中的*AsynchBcastAck*算法的复杂度。

16.5.5 最大独立集

同步器可用于*LubyMIS*等随机同步算法。我们将实现细节留给读者完成。

16.6 时间下界

对同步器的一个非正式解述如下：

任意（无故障）同步算法都可以以不大的代价被转换成相应的异步算法。

特别地，通过使用同步器*Alpha*或者*SimpleSynch*，时间开销根本不会增加。本节给出异步网络算法解决特定问题时所需时间的下界，从而揭示同步方法的一个局限。对于同一问题，有些同步求解算法的速度很快，故

并非每个同步算法都可以被转换成为具有相同时间复杂度的相应异步算法。

这两种非正式的解述似乎相互矛盾，原因在于同步器所能保证的正确性条件的局部性不同。我们将在给出下界证明后回来讨论这个问题。

555 本节的结果是本书针对异步分布式系统中时间复杂性问题给出的惟一下界结果。

我们考虑的问题称为“对话问题”。令$G = (V, E)$是一个图，$diam$是其直径。系统与环境的接口包括$flash_i$输出动作，其中G中每个节点都有一个$flash_i$输出动作。$flash_i$是在节点i的进程自动机的一个输出。我们将*flash*动作视为抽象动作，也可以视为相应进程已经完成计算任务的信号。

定义一个*对话*是*flash*事件的任何序列，对于任意i，序列中都至少包含一个$flash_i$。对于任意非负整数k，*k-对话问题*要求在任意的公平运行中，算法至少执行k个互不相交的对话。

例 16.6.1　k-对话问题的起因

k-对话问题最初来源于异步共享存储器模型上的矩阵计算问题。考虑一组异步并行进程协作计算$m \times m$布尔矩阵的传递闭包，开始时矩阵存放在共享存储器中，所有部分结果和最终输出都被写到共享存储器中。

对于任意i, j, k，都有一个进程$P_{i,j,k}$。如果位置(i, k)和(k, j)中的值都为1，则进程$P_{i,j,k}$负责将1写入到输出矩阵的位置(i,j)中。因此，每个进程在一轮循环中读取位置(i, k)和(k, j)中的值，然后（可能地）写入(i,j)。每个在共享存储器上的读操作或者写操作都被抽象成一个*flash*输出。

如果进程步之间有“足够”的交互，则矩阵的基本属性保证计算的执行是正确的。特别地，执行$O(\log n)$个对话就足够了。进程是否去作更多的读或写并不要紧——只要足够的交互发生，正确的输出将被产生。

该问题的一个具有类似下界的简化版本是要求每个进程在每个对话中只执行一次*flash*。我们使用的版本限制更少，故可以得到更强的下界结果。

同步网络环境的*k*-对话问题的求解相当容易，我们只需让每个进程i在k轮的每一轮中都执
556 行一个单独的$flash_i$输出即可。进程间的通信是不必要的。所需的轮数为k。

在异步网络环境中，进程以I/O自动机表示，进程由可靠FIFO发送/接收通道连接。为了保持一般性，设通道是全工的。与往常一样，时间与事件相关，ℓ是每个进程事件的一个上界，而d是每个通道中的最早消息的传递时间的上界。假定$\ell << d$，故在结果和证明中通常忽略ℓ。我们曾在8.6节中提到，那些在给定限制下时间与事件相关联的公平运行被称为计时运行。

接着来定义算法A的*时间衡量*$T(A)$，对于A的任意计时运行α，定义$T(\alpha)$为*flash*事件在α中发生的时间的上界（采用上界而不是最大值的原因在于这些事件的数目可能是无限的）。然后定义：

$$T(A) = \sup\{T(\alpha) : \alpha \text{是} A \text{的一个计时运行}\}$$

也就是说，$T(A)$是A的计时运行的时间上界，其中*flash*事件发生在所有这些计时运行中。

现在我们给出并证明下界。

定理 16.11　假设A是一个解决图G上*k*-对话问题的异步网络算法，则$T(A) \geq (k-1)\, diam \cdot \mathrm{d}$。

为了将这个结果与同步环境中k轮所需的简单上界作比较，我们指定每轮中的最大消息传送时间为d。定理16.11中的下界与kd的下界之差约等于$diam$的某个因子，这证明对于对话问题，异步性带来的内在开销是$diam$的一个因子。

证明　为了保持一般性，假定A的所有动作都是外部动作。采用反证法来证明。

假设存在满足$T(A) < (k-1) \cdot diam \cdot d$的算法$A$，如果所有消息传送都要花费最大时间$d$，

则A的计时运行被定义为慢的。令α是A的任一慢计时运行，注意带有时间信息的α必然是A的公平运行。由于A是正确的，则α必须包含k个对话。根据假设，从时刻$(k-1)\cdot diam\cdot d$开始，α中没有*flash*事件发生。故可以将α写成$\alpha'\cdot\alpha''$，其中α'的最后事件的时间严格地少于$(k-1)\cdot diam\cdot d$，且α''中没有*flash*事件。另外，可以将α'分解成$k-1$块，就像如下串联$\alpha_1\cdot\alpha_2,\ \ldots\ \alpha_{k-1}$，在每个块$\alpha_r$中，$1\leqslant r\leqslant k-1$第一个事件与最后一个事件的时间之差严格少于$diam\cdot d$。 557

现在来构造A的一个公平轨迹β，β是一个不计时公平轨迹，即其中的事件不与时间相关。它通过连接$\beta=\beta_1\cdot\beta_2\cdot\ldots\cdot\beta_{k-1}\beta''$产生，每个$\beta_r$通过重排$\alpha_r$中的动作（并删除时间信息）产生，$\beta''$是$\alpha''$中动作的序列（时间信息被删除）。我们将表明$\beta$包含少于$k$个对话，从而与$A$的正确性相矛盾。

构造β时的重排必须保持α的动作之间的依赖关系，特别是*receive*事件与相应的*send*事件的关系、和任意进程i的任意事件与同一进程中之前事件之间的关系。我们使用非自反偏序关系来描述这些依赖关系，并使用14.1.4节中定义的记号$\rightarrow_{trace(\alpha)}$来表示它们。定理14.1用来表明$\beta$实际上是$A$的一个公平轨迹。

以下声明描述重排后的序列β_r的属性。设j_0和j_1是G中距离为$diam$的固定两点，定义

$$i_r=\begin{cases} j_0, & \text{如果}r\text{是偶数} \\ j_1, & \text{如果}r\text{是奇数} \end{cases}$$

声明 16.12 对于任意r，$1\leqslant r\leqslant k-1$存在一个由$A$的动作组成的具有以下属性的序列$\beta_r$

1）β_r由重排α_r中的动作序列而得，并保持$\rightarrow_{trace(\alpha)}$顺序。

2）β_r可以写成连接$\gamma_r\delta_r$，其中γ_r不包含进程i_{r-1}的事件且δ_r不包含进程i_r的事件。

我们首先示范如何利用声明16.12来证明该定理。由于事件的重排只发生在单独的β_r序列上，而且遵守依赖关系$\rightarrow_{trace(\alpha)}$，所以定理14.1意味着$\beta$是$A$的一条公平轨迹。但是我们能够证明$\beta$包含最多$k-1$个对话：没有对话能够整个地被包含到$\gamma_1$中去，因为$\gamma_1$不包含事件$i_0$；类似地，没有对话能够整个地被包含在$\delta_{r-1}\gamma_r$的任意片段中。这意味着每个对话都必须包含在$\gamma_r-\delta_r$边界两边的事件。但只有$k-1$个这样的边界，故最多有$k-1$个对话。因此，$\beta$违反了$A$的正确性保证，从而产生矛盾。

剩下的任务是构造声明16.12中需要的序列β_r。固定任一r，$1\leqslant r\leqslant k-1$，考虑以下情况：

1）α_r不包含i_{r-1}事件。 558

则令β_r是α_r中动作的不重排序列，令$\gamma_r=\beta_r$和$\delta_r=\lambda$（空序列）便得到所需属性。

2）α_r不包含i_r事件。

则令β_r是α_r中动作的不重排序列，令$\gamma_r=\lambda$和$\delta_r=\beta_r$即可。

3）α_r包含至少一个i_{r-1}事件和至少一个i_r事件。

则令ϕ是α_r中的第一个i_{r-1}事件，是α_r中的最后一个i_r事件。我们认为不可能有$\pi\rightarrow_{trace(\alpha)}\phi$，即$\pi$不会依赖于$\phi$，原因在于：$\alpha$是一个慢运行，消息从$\alpha$中的进程$i_{r-1}$被广播到$\alpha$中的进程$i_r$所需的时间最少为$diam\cdot d$；然而，$\alpha_r$中第一个事件与最后一个事件的时间差严格少于$diam\cdot d$。

然后，我们可以（具体证明留做一道习题）在保持偏序关系$\rightarrow_{trace(\alpha)}$的情况下重排$\alpha_r$的事件，使得$\phi$在$\pi$之前。令$\beta_r$是得到的事件序列，$\gamma_r$是以$\phi$结尾的$\beta_r$的前缀，而$\delta_r$是$\beta_r$的余下部分。这些序列具有所需的属性。 □

我们再一次强调的是，在定理16.11的证明中构造的轨迹β没有让事件与时间相关联。主要矛盾在于β不包含足够的对话，而不是因为β的任意计时属性。在证明过程中计时信息用于推导

出在慢计时运行α中的事件不依赖其他事件的结论。

正确性的局部概念　定理16.11看起来与一些关于同步器的结果相矛盾——这些结果给出从同步算法到异步算法的只需恒定时间开销的转换。不同之处在于同步器只是保证正确性的“局部”概念。与整个地保持用户集合的行为不同，它们只保持每个单独用户的行为，并允许重排不同用户的事件。

对于许多分布式应用程序，不同用户的事件顺序并不要紧，例如，典型的数据处理应用程序和财务应用程序能够经受住对不同用户事务的乱序处理。然而，对于那些在分布式系统之外
559 的用户之间具有大量通信的应用程序，不同用户的事件顺序可能是相当重要的。

16.7　参考文献注释

Awerbuch[29]介绍同步器的一般概念，并介绍将同步器问题分解成为数据通信部分和安全同步器部分的概念。Awerbuch的论文还定义*Alpha*、*Beta*和*Gamma*同步器，并包括有效的*Gamma*簇分解算法。对于针对广度优先搜索和最大流问题的有效异步算法，其同步器应用程序出现在[29][30]中。有效簇分解方面的进一步工作出现在[35, 36 , 32]中。使用I/O自动机来表示同步器的工作出自Devarajan[89]，随后Fekete、Lynch和Shrira[109]进行了早期的开发。

下界证明来自Arjomandi、Fischer和Lynch[14]，他们给出针对共享存储器模型的结果。本章中的描述采用Attiya和Mavronicolas[17]提出的一些简化结果。Attiya和Mavronicolas[17]还将下界的结果推广到部分同步系统中去。Raynal写了一本完全讨论同步器的书[250]。

16.8　习题

16.1 对于第2章中的同步模型和16.1节中给出的由用户自动机U_i与*GlobSynch*组成的异步模型，给出并证明两者之间的一种密切对应关系。

16.2 补充引理16.1的证明细节。特别地，需要证明声明16.2，还要证明有可能在不违反$\rightarrow_\beta$次序的情况下重排β的事件来获取γ。

16.3 令*L*和*G*分别代表*LocSynch*系统和*GlobSynch*系统，只不过*L*和*G*的输出动作是用户自动机的所有动作（因此用户的内部动作被归为输出动作）。举出反例来证明$fairtraces(L) \subseteq fairtraces(G)$是不可能的。

16.4 补充*SimpleSynch*系统的证明细节和复杂度分析。特别地，

a）给出并证明所有需要的不变式。

b）证明*f*是一个模拟关系。

560 c）根据定理8.13来证明公平性。

d）给出复杂度声明的详细证明。（注意，*d*的假定界限只是指在任意通道中最早消息的传送。）

16.5 令*S*和*G*分别代表*SimpleSynch*系统和*GlobSynch*系统，只不过*L*和*G*的输出动作是用户自动机的所有动作。（因此用户的内部动作被归为输出动作，而*send*和*receive*动作被“隐藏”——被归为内部动作）。

a）举出反例来证明$fairtraces(S) \subseteq fairtraces(G)$不成立。

b）修改*S*来得到新系统*S'*，*S'*由用户自动机和分布式算法组成，使得$fairtraces(S') \subseteq fairtraces(G)$。分析其复杂性。

16.6 补充引理16.5的证明细节。

16.7 写出$Alpha_i$自动机的前提-结果代码，并证明其正确性定理——定理16.7。使用从*Alpha*系统

到*SafeSynch*系统的模拟关系。

16.8 写出$Beta_i$自动机的前提-结果代码，并证明其正确性定理——定理16.8。使用从*Beta*系统到*SafeSynch*系统的模拟关系。

16.9 判断题

令*B*和*G*分别代表*Beta*和*GlobSynch*系统，只不过*B*和*G*的输出动作是用户自动机的所有动作。则有$fairtraces(B) \subseteq fairtraces(G)$。

证明你的结论。

16.10 在*Gamma*同步器中，给出*ClusterSynch*自动机和*ForestSynch*自动机实现中的节点进程的前提-结果代码。证明定理16.10。

16.11 给出一个分布式算法，该算法在网络图*G*上操作，且产生*Beta*同步器所需的最小高度带根生成树。可以假定节点具有UID，但节点都是一样的。你设计的算法的效率如何？ 561

16.12 给出一个分布式算法，该算法在网络图*G*上操作，且产生*Gamma*同步器所需的“好”生成森林。另外，找出用于相邻簇的根之间通信的不同路径。可以假定节点具有UID，但节点都是一样的。你的算法应该能够生成高度较小的树和短通信路径。

16.13 考虑一个由$\sqrt{n} \times \sqrt{n}$个节点组成的平方网格图*G*，通过将每条边分成*k*等份得到划分P_k，P_k有k^2个大小相同的簇。根据*n*和*k*，基于P_k的同步器*Gamma*的通信复杂度和事件复杂度如何（针对分解的最优生成树和最优通信路径）？

16.14 一个在Flaky计算机公司工作的程序员具有无故障方面的大量经验，他刚刚想出一个用于无故障异步网络编程的同步器的绝妙主意。他也认为此主意只适用于完全图G，但仍然觉得此主意大大有用。

他的同步器与*GlobSynch*相似，不同之处在于：对于每个第*r*轮，在执行第*r*轮的$user\text{-}receive_i$事件之前，同步器等待来自至少*n-f*个进程（包括*i*）的第*r*轮的*user-send*，而不是等待来自全部*n*个进程的*user-send*。

在他的算法被安装到一个容错飞行器控制系统前，证明给他的上司看他的算法是错误的（提示：将一个正确的同步协议算法（如*FloodSet*）与提出的算法进行合成，然后找出合成算法的一个错误运行）。

16.15 证明具有同步器的*SynchBFS*算法的终止策略是正确的。

16.16 将*LubyMIS*运行在你喜欢的同步器上，对于得到的异步算法，给出和证明一个结果，该结果给出该异步算法保证的重要属性。

16.17 证明：例16.6.1中描述的布尔矩阵传递闭包问题的求解需要$O(\log n)$个对话。你能够证明的最佳常数是什么？

16.18 证明定理16.11的证明中所缺的声明，即有可能在保持$\rightarrow_{trace(\alpha)}$的情况下重排$\alpha_i$的事件，使得$\phi$在$\pi$之前。 562

16.19 对于*k*-对话问题的异步求解，求出你能够得到的最好的时间复杂度上界。将你的算法扩展到任意同步算法的异步实现中去。哪些正确性条件将得到保证？

16.20 重做习题15.40，这次使用本章中的一些算法分解思想。尽量使用所有的模块。例如，你应该给出抽象自动机来表示：MST算法所需的行为、和使用MST来进行领导者选举的那些算法所需的行为。 563

第17章 共享存储器与网络

在前一章中描述同步器，其中包含一个用于简化异步网络编程的方法。此方法使得诸如第4章中的（非容错）同步网络算法能够在异步网络中使用。在本章中，描述第二种用于简化异步网络编程的策略：利用异步网络来模拟*异步共享存储器系统*。这个策略使得诸如第10章、第11章和第13章中的异步共享存储器算法能够在异步网络中使用。许多其他的异步共享存储器算法（包括针对科学编程和财务数据库的实用算法等）能够被调整到在异步网络中运行。这个策略基于这样的假设：异步共享存储器系统比异步网络系统更易编程。

更一般地，本章处理异步共享存储器模型与异步网络模型之间的关系。在两种转换方向上都得出很好的结果，有些结果甚至保持一些容错属性。这导致我们得出结论：（除了在效率方面的不同）两个模型是相当相似的。

除了提供异步网络的简化编程模型之外，这些转换结果还有其他的意义。例如，从网络模型到共享存储器模型的容错转换意味着我们可以根据异步共享存储器模型中的特定不可能结果来推出异步网络模型的相应不可能结果。

565 从异步共享存储器模型到异步网络模型的另一种转换方法出现在18.3.3节中，这种方法依赖于在异步网络中建立逻辑时间的概念。

17.1 从共享存储器模型到网络模型的转换

在本节中，我们描述几种从异步共享存储器模型转换到异步发送/接收网络模型的方法。17.1.1节给出这种转换必须满足的正确性条件。17.1.2节包含非容错策略，而17.1.3节包含容错策略。我们只考虑一种故障类型——进程停止故障。

17.1.1 问题

我们从第9章的模型中的共享存储器系统A出发。与往常一样，假设A使用由n个编号从0到n的端口组成的集合来与环境交互，在端口i上，A与用户自动机U_i交互。对于第10~13章中的用户自动机，我们假定每个U_i的外部动作都是那些被U_i用于与A交互的动作。在本章中，A的进程可以具有任意数量的任务。由于某些转换保持容错性，我们加入在9.6节中讨论的$stop_i$输入动作，每个$stop_i$事件将永久性地令进程i的所有任务失效。

为了正确地实施这种转换，我们需要用到一个曾在13.1.4节中用到的在A上的技术限制：对于A与任意用户自动机集合的组合，设每个端口i都有一个函数$turn_i$，对于组合系统的每个公平运行α，该函数都产生值$system$或者值$user$。它用来标识在α之后轮到谁来执行一步。如果$turn_i(\alpha) = system$，则在α之后的状态中U_i没有可执行的输出步；如果$turn_i(\alpha) = user$，则在α之后的状态中A的进程i没有可执行的输出步或内部步——即没有可执行的局部控制步。因此，假设相同的共享存储器模型，就像13.1.4节中我们所做的一样。

这里的通用问题（包括对任意端口集I的容错要求）是去设计进程P_i的异步发送/接收网络系统B，$1 \leqslant i \leqslant n$，即是$A$的一个$I$模拟。$I$模拟的定义如下：设$B$是由用户$U_i$组成的任一集合上的

566 网络系统，对于B的任一运行α，存在同一用户集合上的A的一个满足以下条件的运行α'：

1）α和α'对U（用户U_i的合成）是不可区分的[⊖]；

⊖ 这使用了8.7节中的正式概念“不可区分”。

2）对于任意i，如果$stop_i$在α'中发生，则$stop_i$也在α中发生。

另外，如果α是一个公平运行，且那些其中$stop_i$出现在α中的所有i都在I中，则α'也是一个公平运行。如果对于任意满足$|I| \leqslant f$的I，B都是A的一个I模拟，则称B是A的一个f模拟。

你可能意识到这些条件与定理13.7中的条件相似，我们将在证明特定网络系统模拟了共享存储器系统的时候来挖掘两者之间的关系。

如14.1.1节所述，在系统B中，每个$stop_i$事件永久地令进程P_i中的任务失效。然而，$stop$事件并不影响通道。

17.1.2 无故障时的策略

在无故障发生时，我们可以使用简单的策略。根据共享变量在网络中的副本个数，大多数策略可以分为单复制（*single-copy*）方案和多复制（*multi-copy*）方案两种。

单副本方案 这种最简单的策略只涉及将A的共享变量任意地分布到B的进程中去，其中每个共享变量都在单个进程中。该策略适用于任意类型的共享变量。

***SimpleShVarSim*算法：**

A的每个共享变量x被“划给”B的单个进程P_i。P_i的任务有两个：模拟A的相应进程i，并管理它拥有的共享变量。

对于任意i，在用户接口中，进程P_i的动作与A的进程i的动作相同。P_i的步直接模拟进程i的步，除非：当A的进程i访问共享变量x时，P_i发送一个调用信息给拥有变量x的进程P_j（如果P_i本身拥有x，则它将调用请求传给一个“子程序”）；然后P_i暂停对进程i的模拟中的所有局部控制动作，等待调用的结果。当回应到达时，P_i继续模拟A的进程i。

当共享变量x的主人接收到要求调用x的消息（或者一个局部调用请求）时，它在不可分割 567
的一步中将调用作用于x，把结果放入回应消息发给调用的发送者（如果调用是本地的，则传回给主模拟任务）。

*SimpleShVarSim*算法具有一些有趣的模块化性质。我们可以将每个进程P_i表示成为一个I/O自动机Q_i和一个I/O自动机$R_{x,i}$的合成，其中Q_i负责模拟A的进程i，而$R_{x,i}$负责模拟共享变量x[⊖]。

我们将13.1.4节中的*Trans*(A)算法自动机P_i作为Q_i。更精确地，设自动机Q_i的输出包括形式为$a_{x,i}$的动作而输入包括形式为$b_{x,i}$的动作，其中a是A的进程i对共享变量x的一个调用，而b是共享变量x对进程i的调用作出的一个回应。

每个$R_{x,i}$都有输入$a_{x,i}$和输出$b_{x,i}$。为了方便起见，我们假设对于任意特定的共享变量x，所有自动机$R_{x,i}$都有用于互相通信的FIFO发送/接收通道。如习题14.6所示，我们可以使用给定的FIFO可靠通道来模拟单个x的通道。对于每个x，所有自动机$R_{x,i}$和它们之间的通道的合成组成x的变量类型的原子对象。

图17-1显示包含两个进程和两个共享变量的*SimpleShVarSim*的体系结构，我们并不明确给出$stop$动作——每个$stop_i$都假定是Q_i和所有$R_{x,i}$的输入。

$R_{x,i}$的代码如下。代码根据P_i是否为x的所有者来分成两个部分。由于$stop_i$动作被包括在$R_{x,i}$的签名之中，我们明确描述出对$stop_i$动作的处理：该动作设置一个*stopped*标志，使得所有局部控制动作失效，并防止任意与输入动作有关的变化。（这种处理并不有趣，因为我们没有指出算法在发生故障时的行为。）为了清晰起见，通道动作的下标包括变量的名字和两端的节点名字。

⊖ 我们还隐藏了它们之间的通信动作。

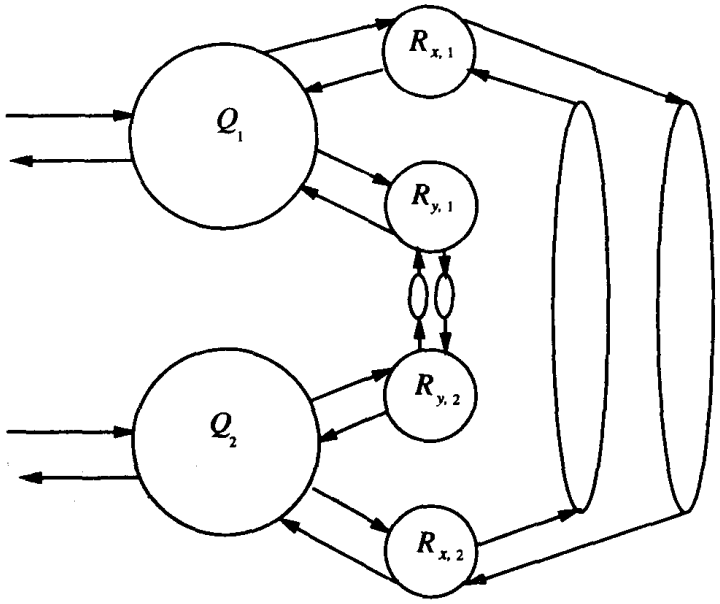


图17-1 包含两个进程和两个共享变量的*SimpleShVarSim*的体系结构

$R_{x,i}$, P_i the owner of x:

Signature:

Input:

$a_{x,i}$, a an invocation of x by process i

receive("*invoke*", $a)_{x,j,i}$, a an invocation of x by j, $j \neq i$

568 $stop_i$

Output:

$b_{x,i}$, b a response of x to process i

send("*respond*", $b)_{x,i,j}$, b a response of x to j, $j \neq i$

Internal:

perform$(a,j)_{x,i}$, a an invocation of x, $1 \leq j \leq n$

States:

val, a value in the domain of x, initially the initial value of x

inv-buffer, a set of pairs (a,j), a an invocation, $1 \leq j \leq n$, initially empty

resp-buffer, a set of responses b, initially empty

stopped, a Boolean, initially *false*

for every $j \neq i$:

send-buffer(j), a FIFO queue of responses, initially empty

Transitions:

$a_{x,i}$

Effect:

if *stopped* = *false* then

inv-buffer := *inv-buffer* $\cup \{(a,i)\}$

569 *perform*$(a,j)_{x,i}$

Precondition:

stopped = *false*

$(a,j) \in$ *inv-buffer*

Effect:

inv-buffer := *inv-buffer* $- \{(a,j)\}$

receive("*invoke*", $a)_{x,j,i}$

Effect:

if *stopped* = *false* then

inv-buffer := *inv-buffer* $\cup \{(a,j)\}$

send("*respond*", $b)_{x,i,j}$

Precondition:

stopped = *false*

b is first on *send-buffer*(j)

Effect:

remove first element of *send-buffer*(j)

$(b, val) := f(a, val)$
if $j = i$ then
$resp\text{-}buffer := resp\text{-}buffer \cup \{b\}$
else
add b to $send\text{-}buffer(j)$

$b_{x,i}$
Precondition:
$stopped = false$
$b \in resp\text{-}buffer$
Effect:
$resp\text{-}buffer := resp\text{-}buffer - \{b\}$

$stop_i$
Effect:
$stopped := true$

Tasks:
$\{b_{x,i} : b$ is a response$\}$
for every j:
$\{send(\text{“respond”}, b)_{x,i,j} : b$ is a response$\}$
$\{perform(a, j)_{x,i} : a$ is an invocation$\}$

$R_{x,i}$, P_i not the owner of x:

Signature:

Input:
$a_{x,i}$, a an invocation of x by process i
$receive(\text{“respond”}, b)_{x,j,i}$, b a response of x to i, j the owner of x
$stop_i$

Output:
$b_{x,i}$, b a response of x to process i
$send(\text{“invoke”}, a)_{x,i,j}$, a an invocation of x by i, j the owner of x

States:
resp-buffer, a set of responses b, initially empty
send-buffer, a FIFO queue of invocations, initially empty
stopped, a Boolean, initially *false*

Transitions:

570

$a_{x,i}$
Effect:
if $stopped = false$ then
add a to *send-buffer*

$send(\text{“invoke”}, a)_{x,i,j}$
Precondition:
$stopped = false$
a is first on *send-buffer*
Effect:
remove first element of *send-buffer*

$receive(\text{“respond”}, b)_{x,j,i}$
Effect:
if $stopped = false$ then
$resp\text{-}buffer := resp\text{-}buffer \cup \{b\}$

$b_{x,i}$
Precondition:
$stopped = false$
$b \in resp\text{-}buffer$
Effect:
$resp\text{-}buffer := resp\text{-}buffer - \{b\}$

$stop_i$
Effect:
$stopped := true$

Tasks:
$\{b_{x,i} : b \text{ is a response}\}$
$\{send(\text{“}invoke\text{”}, a)_{x,i,j} : a \text{ is an invocation}\}$

定理 17.1　基于A的*SimpleShVarSim*算法是A的一个0模拟。（我们并不声明任何容错性）。

证明概要　首先，对于每个x，所有自动机$R_{x,i}$的合成 $1 \leqslant i \leqslant n$，加上它们之间的通道（隐藏了*send*和*receive*动作），组成一个x的变量类型和13.1.4节的*Trans*定义中所指定接口的原子对象B_x。（然而，这里不提B_x的任何容错属性。）带有这些原子对象B_x的系统B就是系统*Trans*(A)。这使我们可以应用定理13.7，其中$I = \phi$：如果α是带有用户U_i的B的任一运行，则定理13.7产生带有相同用户的A的一个运行α'，α'满足0模拟定义中的所有条件。 □

共享变量的位置　*SimpleShVarSim*算法允许将变量划给任意进程。然而，若将变量放在对它的访问频率最高的进程中的话，性能最佳。

例如，如果对于单写者/多读者的读/写共享变量x，其写操作比读操作多得多，则应将它放在与写者对应的进程中。这样写操作在局部执行，故速度很快。当然，此时除了写者之外的所
571 有进程的读访问都比较慢，因为它们涉及到网络上消息的交换。如果写访问比读访问频繁，则这种安排是合适的；但当写访问比较罕见时，则这种安排不是最好的。

容错性　*SimpleShVarSim*算法没有有趣的容错性。例如，如果$stop_i$发生，所有之后的进程都不能访问进程P_i拥有的变量。

忙-等待　一些共享存储器算法，例如10.7节中的*Bakery*互斥算法和11.3节中的*RightLeftDP*哲学家用餐算法，采用忙-等待循环来令进程反复检查一个共享变量，直到特定条件被满足。为了删除这些循环，可以修改*SimpleShVarSim*算法，即让变量的主人在变量值发生变化时（或者等待的条件变为真时）通知忙-等待进程。这样可以减少通信复杂度。

多副本方案　让几个进程同时拥有同一共享变量x的副本有时是有用的。例如，当x是一个读操作多于写操作的读/写共享变量时（这多见于许多数据库），如果许多进程都存有x的“缓冲”副本，则大量读操作都可以在本地进行，使得代价较低。不过此时写操作必须在x的所有副本上操作，即写进程必须发送消息给所有拥有x副本的进程，因此代价变高。

情况甚至变得更为恶劣。例如，假设x是一个多写者寄存器。则进程P_1和P_2可以同时尝试写x，而存有x副本的进程P_3和P_4可能以相反的顺序来接收来自P_1和P_2的消息，以后的读操作便会得到不一致的结果。

即使x是单写者存储器时，异常仍然会发生。如果写者发送写消息，这些消息可能先到达进程P_1，然后到达P_2。P_1在接收到消息后执行的本地读操作将取回新值，而P_2在接收到消息之
572 前执行的本地读操作可能还是取回旧值。倘若P_1的读在P_2的读开始之前完成，则这种行为在读/写原子对象中是被禁止的。

因此，需要一种更明智的协议来管理写操作。例如，写者可以分两个阶段工作：第一阶段“上锁”并修改x的所有副本，第二阶段进行解锁。当本地副本被上锁时，其上的读操作被延迟。注意必须让这些操作最终得到执行。

这种类型的算法属于*并发控制算法*的例子。我们刚才浅谈过的算法是一个*读/写上锁算法*，该算法实现了一个*原子事务*——写x的所有副本。这意味着在写操作的间隔之中的某个“序列化点”上，操作x的进程即时地写x的所有副本。还有许多其他类型的并发控制协议，包括针对除了读/写变量之外的其他类型共享变量的上锁算法、基于时戳的算法、结合使用上锁和时戳

的混合算法以及乐观算法等，这里不进行一一介绍，请参考Lynch、Merritt、Weihl和Fekete编写的书《*Atomic Transactions*》（该书的风格与本书相似）来了解详细内容。

一个流行的读/写共享变量多副本算法是*MajorityVoting*算法，其核心是为每个读/写共享变量x实现了读/写原子对象，而原子对象的实现基于低层原子事务的实现。

***MajorityVoting*算法（非形式化）：**

n个进程中的每一个都保存着x的副本和一个非负整数*tag*，副本的初值等于x的初值，*tag*的初值为0。

想要读或写x的进程执行一个涉及x的某些副本的操作。原子事务由一系列操作组成，这些操作在事务的执行过程中的某个“序列化点”上即时地进行（未完成的操作可能有或没有序列化点）。事务可以使用两阶段上锁算法、基于时戳的算法、混合算法、或乐观并发方法等来实现，另外加上优先权机制来保证每个事务都能够最终完成。

P_i执行读x操作时，它至少读取x的大多数副本，然后从中选出*tag*值最大的一个并返回其x值。这些步骤都是同一原子事务的一部分，看起来好像即时地执行。 573

P_i执行写x操作时，它首先执行一个如上所述的内嵌读操作，从而得到最大的*tag*值t，然后将$(v, t+1)$写入到至少x的大多数副本中。这些步骤——包括内嵌读副本和写副本——都是同一原子事务的一部分，看起来好像即时地执行。

引理 17.2 *MajorityVoting*算法是一个读/写原子对象。

证明概要 我们来验证13.1.1节的原子对象定义中的条件。良构性和无故障终止是显然的。对于原子性条件，固定*MajorityVoting*算法（采用任意自动机集合）的一个运行α，选出一个不完全操作子集Φ，其中的操作在事务实现中被赋予序列化点，且我们采用对Φ中的操作的回应和在事务实现中的序列化点。为了证明收缩属性成立，对于每个读操作，如果之前被序列化的写操作写入了值，则读操作获取该值；否则读操作得到初始值v_0。

关键在于如下事实：

1）按照序列化点的次序，写操作获得*tag* 1, 2, ...。

2）每个读操作或者内嵌读操作都获得之前写操作写入的最大*tag*（如果写操作不存在，则取0）和相应值。

因为每个读操作或者内嵌读操作都读了大多数副本，而最大*tag*已被写入到大多数副本中去，再加上大多数必然相互重叠的特点，所以这些事实成立。 □

现在，如果共享存储器系统A使用读/写共享变量，那么我们把基于A的*MajorityVoting*算法定义为由在*SimpleShVarSim*算法中使用的相同Q_i组件和所有读/写共享变量的*MajorityVotingObject*组成。则引理17.2意味着：

定理 17.3 假设A使用读/写共享变量。则基于A的*MajorityVoting*算法是A的一个0模拟。 574

容错性 尽管*MajorityVotingObject*算法具有选择哪些大多数副本进行读和写的灵活性，它一般不提供对x的原子对象的容错实现。原因在于标准的事务实现是非容错的。例如，在读/写上锁算法中，执行读事务的进程可能发送消息来读取大多数副本，使得大多数副本被上锁。如果进程在没有解锁的情况上发生故障，则以后的写事务再也得不到所需的锁。实际上，为了应付这个问题，可以使用超时机制来检测进程故障（我们不能在异步网络模型中这样做），也可以弱化故障回复性要求。

17.1.3 容忍进程故障的算法

17.1.2节中的策略不具有容错性。本节介绍Attiya、Bar-Noy和Dolev的*ABD*算法，它用于进程故障数为f的场合，且假设网络是可靠的。设进程数n严格大于$2f$，即大多数进程都不出故障。我们只考虑单写者/多读者的读/写共享存储器的情况。

*ABD*算法的核心在于对任意读/写共享变量都实现了保证f故障终止的读/写原子对象。为了简单起见，假设在实现中端口1上只有写操作发生，端口2, ..., n上只有读操作发生，以后再对此进行轻微修改以用于通用模拟中。算法利用*MajorityVoting*算法和13.4.3节中*VitanyiAwerbuch*算法的思想，关键在于每次写的结果都在写操作完成之前被保存在网络的大多数节点中。

***ABDObject*算法（非形式化）：**

n个进程中的每一个都保存着x的副本和一个非负整数*tag*，副本的初值等于x的初值，*tag*的初值为0。

当惟一的写者进程想要执行x上的*write*(v)操作时，先令t为尚未赋给任意写操作的*tag*中最小的一个，然后将本地x副本的值和本地*tag*值分别设为v和t，并发送消息(“*write*”, v, t)给所有其他进程。如果收到该消息的进程发现t值大于自己的*tag*的当前值，则以同样的方式更新其x副
575 本和*tag*。无论如何，进程发送一个确认给写者。当写者知道（通过确认消息和本地行为的知识）大多数进程的*tag*值等于t时，返回*ack*。

当任一进程P_i想要执行读x操作时，它发送读消息给其他所有进程并读取本身的x值和*tag*值。接收到该消息的进程返回x和*tag*的最新值。当P_i得知大多数进程的x值和*tag*值后，它准备返回与最大*tag*值相关的那个x的值v，不过之前P_i先将(v, t)广播给大多数进程：它更新自身的x值和*tag*值并将一个第二轮消息发送给其他所有进程（写者进程除外）。如果收到该消息的进程发现t值大于进程的*tag*的当前值，则相应地更新其x副本和*tag*。无论如何，它都发送一个确认给P_i。当P_i知道（通过确认消息和本地行为的知识）大多数进程的*tag*值等于t时，返回v。

具体代码如下。ABD_1是写者进程，ABD_2, ..., ABD_n是读者进程。为了简单起见，我们不提及*stop*动作，该动作的处理与*SimpleShVarSim*中的情况类似。我们还省略了不同动作的显式下标x。（具体代码已经足够长，足以不用带这些细节。）设V是值的域，v_0是x的初始值。

$ABDObject_1$ automaton (formal):

Signature:

Input:
- $write(v)_1$, $v \in V$
- $receive(\text{“write-ack”}, t)_{j,1}$, $t \in \mathbb{N}^+$, $j \neq 1$
- $receive(\text{“read”}, u)_{j,1}$, $u \in \mathbb{N}^+$, $j \neq 1$

Output:
- ack_1
- $send(\text{“write”}, v, t)_{1,j}$, $v \in V$, $t \in \mathbb{N}^+$, $j \neq 1$
- $send(\text{“read-ack”}, v, t, u)_{1,j}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \neq 1$

States:

$val \in V$, initially v_0
$tag \in \mathbb{N}$, initially 0
$status \in \{idle, active\}$, initially *idle*
$count \in \mathbb{N}$, initially 0
for every $j \neq 1$:
576 - *send-buffer*(j), a FIFO queue of messages, initially empty

Transitions:

$write(v)_1$
Effect:
$val := v$
$tag := tag + 1$
$status := active$
$count := 1$
for all $j \neq 1$ do
add ("write", v, tag) to $send\text{-}buffer(j)$

$send(m)_{1,j}$
Precondition:
m is first on $send\text{-}buffer(j)$
Effect:
remove first element of $send\text{-}buffer(j)$

$receive(\text{"write-ack"}, t)_{j,1}$
Effect:
if $status = active$ and $t = tag$ then
$count := count + 1$

ack_1
Precondition:
$status = active$
$count > \frac{n}{2}$
Effect:
$count := 0$
$status := idle$

$receive(\text{"read"}, u)_{j,1}$
Effect:
add ("read-ack", val, tag, u)
to $send\text{-}buffer(j)$

Tasks:
$\{ack_1\}$
for every j:
$\{send(m)_{1,j} : m$ a message$\}$

注意：相比*MajorityVoting*和*VitanyiAwerbuch*算法，在*ABDObject*算法中选择一个新*tag*是简单的，因为只有一个写者。下面的代码是读者进程的代码。

$ABDObject_i$ automaton, $2 \leq i \leq n$ (formal):

Signature:

Input:
$read_i$
$receive(\text{"write"}, v, t)_{1,i}$, $v \in V$, $t \in \mathbb{N}^+$
$receive(\text{"read-ack"}, v, t, u)_{j,i}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \neq i$
$receive(\text{"prop-ack"}, u)_{j,i}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
$receive(\text{"read"}, u)_{j,i}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
$receive(\text{"propagate"}, v, t, u)_{j,i}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
Output:
v_i, $v \in V$
$send(\text{"write-ack"}, t)_{i,1}$, $t \in \mathbb{N}^+$
$send(\text{"read"}, u)_{i,j}$, $u \in \mathbb{N}^+$, $j \neq i$
$send(\text{"propagate"}, v, t, u)_{i,j}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
$send(\text{"read-ack"}, v, t, u)_{i,j}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
$send(\text{"prop-ack"}, u)_{i,j}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$ 577

States:
$val \in V$, initially v_0
$tag \in \mathbb{N}$, initially 0
$response\text{-}val \in V$, initially v_0
$read\text{-}tag \in \mathbb{N}$, initially 0
$status \in \{idle, active1, active2\}$, initially $idle$
$count \in \mathbb{N}$, initially 0
for every $j \neq i$:
$send\text{-}buffer(j)$, a FIFO queue of messages, initially empty

Transitions:

$read_i$
　Effect:
　　read-tag := *read-tag* + 1
　　status := *active1*
　　count := 1
　　for all $j \neq i$ do
　　　add (*"read"*, *read-tag*)
　　　to *send-buffer*(j)

$send(m)_{i,j}$
　Precondition:
　　m is first on *send-buffer*(j)
　Effect:
　　remove first element of *send-buffer*(j)

$receive(\text{"read-ack"}, v, t, u)_{j,i}$
　Effect:
　　if *status* = *active1* and u = *read-tag* then
　　　count := *count* + 1
　　　if $t >$ *tag* then
　　　　val := v
　　　　tag := t
　　　if *count* $> \frac{n}{2}$ then
　　　　response-val := *val*
　　　　status := *active2*
　　　　count := 1
　　　　for all $j \notin \{1, i\}$ do
　　　　　add (*"propagate"*, *val*, *tag*, *read-tag*)
　　　　　to *send-buffer*(j)

$receive(\text{"prop-ack"}, u)_{j,i}$
　Effect:
　　if *status* = *active2* and u = *read-tag*
　　　then *count* := *count* + 1

v_i
　Precondition:
　　status = *active2*
　　count $> \frac{n}{2}$
　　v = *response-val*
　Effect:
　　count := 0
　　status := *idle*

$receive(\text{"write"}, v, t)_{1,i}$
　Effect:
　　if $t >$ *tag* then
　　　val := v
　　　tag := t
　　add (*"write-ack"*, t) to *send-buffer*(1)

$receive(\text{"read"}, u)_{j,i}$
　Effect:
　　add (*"read-ack"*, *val*, *tag*, u)
　　　to *send-buffer*(j)

$receive(\text{"propagate"}, v, t, u)_{j,i}$
　Effect:
　　if $t >$ *tag* then
　　　val := v
　　　tag := t
　　add (*"prop-ack"*, u) to *send-buffer*(j)

578

Tasks:
$\{v_i\}$
for every j:
　$\{send(m)_{i,j} : m$ a message$\}$

在上面的代码中，*read-tag*用于跟踪哪个确认属于当前操作，*response-val*用于记住在广播时要返回的值。注意无需将回应值广播给写者，因为写者肯定已经知道最新值。

定理 17.4　当$n > 2f$时，*ABDObject*算法是一个保证f故障终止的读/写原子对象。

证明概要　这类似于第13章中*VitanyiAwerbuch*和*IntergerBloom*算法的证明。良构性是显而易见的。f故障终止也易于证明，因为每个操作只需要大多数进程的参与而$n > 2f$。故此，原子性是要证明的核心。我们需要利用引理13.16。

令α是*ABDObject*算法的任一运行，使用针对异步网络环境来重新描述引理13.10，为了保持一般性，我们可以假设α不包含未完成的操作。

将Π定义为由α中出现的操作组成的集合，定义Π的偏序关系如下：首先根据执行顺序（即*tag*值）来对写操作排序，然后，如果读操作的*tag*值是从写操作中取得的，则将它紧挨着放在该写操作之后，否则将它放在所有写操作之前。

需要证明的核心性质是：

1）如果带有$tag = t$的写操作π在读操作ϕ被调用之前完成，则ϕ包含一个至少与t一样大的tag。这是因为π的tag被大多数副本接收，而ϕ读取大多数副本，这两个大多数必然相交。

2）如果读操作π在读操作ϕ被调用之前完成，则ϕ得到的tag至少与π得到的tag一样大。与第1点类似，这是因为π将其信息广播给大多数副本。

579

使用这两点性质，可以很容易地证明引理13.16的四个条件成立，则原子性条件也成立。□

显然，我们可以修改*ABDObject*算法，使得任意端口i（而不只是端口1）都可以是写端口；读操作也可以被允许发生在单个写端口上。这些修改仍然保证f故障终止。可以像*SimpleShVarSim*和*MajorityVoting*算法一样，使用*Trans*(*A*)的进程和每个共享变量的原子对象，来构造完整的基于*A*的*ABD*算法，其中原子对象是*ABDObject*的相应修改版本。

定理 17.5 假设*A*使用单写者/多读者的读/写共享存储器且$n > 2f$，则基于*A*的*ABD*算法是*A*的一个f模拟。

证明 根据定理17.4和定理13.7来证明。 □

有界*tag* *ABD*算法使用无界*tag*值，可以修改算法以让它使用有界*tags*，具体修改细节留做一道习题。

应用 *ABD*算法可用于实现许多有趣的基于单写者/多读者寄存器的容错共享存储器算法。例如，可以使用*ABD*算法来将第13章中的原子快照和原子多写者寄存器算法转换成为在异步发送/接收网络模型上实现相同对象的算法。不过，要注意的是，原来的算法保证无等待终止，而转换后的算法只能容忍f故障，其中$n > 2f$。

17.1.4 对于n/2故障的不可能性结果

当$n \leqslant 2f$时，不难看出*ABD*算法不能只能容忍f故障，原因在于：当f个进程发生故障时，其他进程永远不能得到完成自身工作所需的大多数。这种局限是内在的，关于异步网络中读/写原子对象实现的容错性的一个关键局限如下。为了得到更强的结论，我们给出在广播系统而不是发送/接收系统中的结果。

580

定理 17.6 令$n = m + p$，其中$m, p \geqslant 1$。假设$n \leqslant 2f$，则不存在异步广播模型（具有可靠广播通道）中的一个算法，该算法实现了具有m个写者和p个读者的读/写原子对象，并保证f终止。

证明 采用反证法，假设存在这样一个算法*A*。如同往常一样，假设用户尽可能地不确定。

令G_1是由$1,\cdots, n-f$组成的集合，G_2是由$n-f+1, \cdots, n$组成的集合。由假设得知，$|G_1| \leqslant f$，$|G_2| \leqslant f$。

考虑系统（*A*加上用户）的一个公平运行α_1，该运行只包含一个端口1上的调用$write(v)_1$，其中$v \neq v_0$。假设*stop*输入在G_2中的端口上发生且恰好在运行的开始部分出现，这意味着下标在G_2中的进程从不执行局部控制动作。根据f故障终止条件，写操作最终以一个匹配的ack_1结束。令α_1'是以ack_1结尾的α_1的前缀。

现在考虑第二个公平运行α_2，该运行只包含一个在端口n上的调用$read_n$。假设*stop*输入在G_1中的端口上发生且恰好在运行的开始部分出现。根据f故障终止条件，这个读操作必然最终终止，且回应值必为v_0。令α_1'是以这个回应结尾的α_1的前缀。

现在构造一个不满足原子属性的有限运行α，从而产生矛盾。运行α满足以下条件：

1）α与α_1'对于编号在G_1中的进程是不可区分的；

2）α与α_2'对于编号在G_2中的进程是不可区分的；

3）在α中，反应事件ack_1出现在调用事件$read_n$之前。

因此读操作返回由写操作写入的值v，而不是初值v_0，这违反了原子性条件。

以如下方法来构造运行α：它不包含*stop*事件。除了编号在G_2中的进程的*stop*事件和*receive*事件之外，α_1'的所有其他事件组成α的起始部分。由于G_2的进程刚好在α_1'的开始部分出错，因此删除这些事件之后仍然得到一个运行，该运行与α_1'对G_1中的进程是不可区分的。运行α接着
581 以α_2'中除了编号在G_1中的进程的*stop*事件和*receive*事件之外的事件结束。

因此，在α中，G_1或者G_2中的进程与另一组中的进程相互独立，G_1中的进程广播的消息不会被传送到G_2的进程中去，反之亦然。容易看出α满足所需条件。 □

定理17.6指出，对于任意给定的n和f，$n \geq 2$，$f \geq n/2$，即使共享变量被限为单写者/单读者寄存器，也不存在一种通用方法来产生n进程共享存储器算法的f模拟。若要理解这个结论，则必须注意：对于任意这样的n，存在一个无等待共享存储器算法A，该算法使用单个1写者/$n-1$读者的读/写寄存器来实现1写者/$n-1$读者的读/写原子对象；而A的f模拟要求产生一个发送/接收网络算法，该算法实现了满足f故障终止条件的1写者/$n-1$读者的读/写原子对象（证明过程与推论13.9的证明过程相似），这与定理17.6矛盾。

17.2 从网络模型转换到共享存储器模型

现在我们描述相反方向的转换，即从异步网络模型到共享存储器模型的转换。这转换能够容忍进程停止故障：进程故障数最多为f的共享存储器系统能够模拟进程故障数最多为f的网络系统（网络具有可靠通道）。现在对故障数目不作特殊要求——这与相反方向的转换的情况不同，即使$n \leq 2f$时也可以构造可行的转换。另外，这些构造比相反方向转换的构造简单得多。

这些构造更为简单且功能更强，这得益于异步共享存储器模型比异步网络模型强大，原因在于共享存储器器具有可靠性。

可以使用这些转换来将异步网络算法运行在异步共享存储器系统上。但是吸引力可能不大，因为共享存储器模型更易编程。一个更重要的用途是将异步共享存储器模型上的不可能结果应用到异步网络模型上。例如，可以使用这些转换来将针对共享存储器模型的故障存在时的一致
582 性不可能性结果（在定理12.8中证明）扩展到网络模型中。

我们提出两种转换方法：一种针对发送/接收系统，另一种针对广播系统。

17.2.1 发送/接收系统

给定第14章中的基于有向图的异步发送/接收系统A，系统具有进程P_i，$1 \leq i \leq n$，和可靠FIFO通道C_{ij}。与以前一样，每个$stop_i$事件立刻令P_i的所有任务失效，但不影响通道。

问题（包括容错要求）是去产生一个具有n个进程的共享存储器系统B，该系统使用单写者/单读者共享寄存器来“模拟”A。这与反方向的转换类似。对于用户集合U_i上的B的任意运行α，必然存在相同用户集合上的A的运行α'，使得以下条件成立：

1）α和α'对于U是不可区分的。

2）对于任意i，如果一个$stop_i$在α中发生，则一个$stop_i$在α'中发生。

另外，如果α是公平运行且在α中出现的$stop_i$事件的下标i在I中，则α'也是一个公平运行。如果对于特定的I，B以这种方式模拟A，则称B是A的一个I模拟。如果对于任意满足$|I| \leq f$的I，B都是A的一个I模拟，则称B是A的一个f模拟。

我们给出一个算法*SimpleSRSim*，它能处理任意故障，即它是一个n模拟。

SimpleSRSim算法（非形式化）：

对于有向图G中的每条边(i,j)，B都包含一个可被进程i写和进程j读的单写者/单读者的读/写

共享变量$x(i,j)$。它包含一个初值为空的消息队列。进程i只能增加消息到这个队列中；没有消息会被删除。

B的进程i模拟A的进程P_i。对P_i的用户接口步和内部步的模拟是很直接的。为了模拟P_i的$send(m)_{i,j}$动作，A的进程i将消息m添加到在变量$x(i,j)$中的队列尾部（可以通过在本地保存一个队列的副本，然后使用一个写操作来实现）。另外，进程i常常检查所有到来的变量$x(j,i)$，以决定在上次检查之后是否有新的消息到达。如果有，进程i以P_i处理这些消息的方法来处理它们。 583

具体代码如下。注意每个进程都有几个任务。在$check(j)_i$的代码中，我们使用记号$receive(M)_{j,i}$作为动作序列$receive(m_1)_{j,i}$, $receive(m_2)_{j,i}$, ……，的简写，其中M是消息队列m_1, m_2, ...。在该代码片段中，序列M包含自进程i上次检查之后被放在$x(j,i)$中的新消息。

SimpleSRSim algorithm (formal):

Shared variables:
for every edge (i,j) of G:
  $x(i,j)$, a FIFO queue of messages, initially empty

Actions of i:
As for P_i, except:
Input:
  Omit all *receive* actions.
Output:
  Omit all *send* actions.
Internal:
  $send(m,j)_i$ for every $send(m)_{i,j} \in out(P_i)$
  $check(j)_i$ for every $j \in in\text{-}nbrs$

States of i:
$pstate \in states(P_i)$, initially a start state
for every $j \in out\text{-}nbrs$:
  $out\text{-}msgs(j)$, a FIFO queue of messages, initially empty
for every $j \in in\text{-}nbrs$:
  $in\text{-}msgs(j)$, a FIFO queue of messages, initially empty
  $processed\text{-}msgs(j)$, a FIFO queue of messages, initially empty

Transitions of i:

π, an input of $P_i \neq receive$
  Effect:
    $pstate$:= any s such that
      $(pstate, \pi, s) \in trans(P_i)$

π, a locally controlled action of $P_i \neq send$
  Precondition:
    π is enabled in $pstate$
  Effect:
    $pstate$:= any s such that
      $(pstate, \pi, s) \in trans(P_i)$

$send(m,j)_i$
  Precondition:
    $send(m)_{i,j}$ is enabled in $pstate$
  Effect:
    add m to $out\text{-}msgs(j)$
    $x(i,j) := out\text{-}msgs(j)$
    $pstate$:= any s such that
      $(pstate, send(m)_{i,j}, s) \in trans(P_i)$

$check(j)_i$
  Precondition:
    true
  Effect:
    $processed\text{-}msgs(j) := in\text{-}msgs(j)$
    $in\text{-}msgs(j) := x(j,i)$
    $pstate$:= last state of any execution
      fragment starting with $pstate$ and
      with action sequence $receive(M)_{j,i}$,
      where $processed\text{-}msgs(j) \cdot M = in\text{-}msgs(j)$ 584

Tasks of i:
As for P_i, except:
　replace each $send(m)_{i,j}$ by $send(m,j)_i$
　add, for every j:
　　$\{check(j)_i\}$

应该不难看出该模拟是正确的。

定理 17.7　如果A是一个具有可靠FIFO发送/接收通道的异步发送/接收系统，则*SimpleSRSim*算法是A的一个n模拟。

证明　具体证明留做一道习题。　□

17.2.2 广播系统

可以使用类似于*SimpleSRSim*的构造方法来模拟具有可靠广播通道的异步广播系统。模拟的正确性条件与对发送/接收系统的模拟的正确性相同。主要区别在于新的模拟使用单写者/多读者寄存器而不是单写者/单读者寄存器。

***SimpleBcastSim*算法：**

对于任意i，$1 \leqslant i \leqslant n$，$B$包含一个可被进程$i$写和所有进程（包括$i$）读的单写者/多读者的共享变量$x(i)$。$B$包含一个初值为空的消息队列。

与以前一样，B的进程i模拟A的进程P_i，B直接模拟P_i的用户接口步和内部步。为了模拟P_i的$bcast(m)_i$动作，A的进程i将消息m添加到在变量$x(i)$中的队列尾部。另外，进程i常常检查所有变量$x(j)$（包括变量$x(i)$），以决定是否有新的消息。如果有，进程i以P_i处理这些消息的方法来处理它们。

定理 17.8　如果A是一个具有可靠广播通道的异步广播系统，则*SimpleBcastRim*算法是A的一个n模拟。

585

17.2.3 异步网络中一致性的不可能性

定理17.8可用于证明试图在异步网络中解决第12章中的基本一致性问题是不可能的，即使网络保证：可靠广播、不超过一个进程出故障、惟一的故障类型是停止故障！这个不可能性结果代表异步网络计算能力的一个局限。

这个结果应该与第6章中针对同步模型中的停止一致性问题的结果进行对比。在那种环境，问题是可解的，尽管内在时间开销取决于能被容忍的故障的数目。在定理6.33的证明中，时间的下界基于这样一种可能性：进程可能在广播的途中停止。而在异步模型中，即使不存在局部广播的可能性，这个不可能性结果仍然成立。

我们使用12.1节中给出的问题声明方法来描述具有1故障终止的一致性问题。（注意这种声明可以使用轨迹属性来进行形式化描述，使得它适用于异步网络系统和共享存储器系统。）

定理 17.9　在具有可靠广播通道的异步广播系统中，不存在能够解决一致性问题并保证1故障终止的算法。

证明　假设存在这样一个算法A，根据定理17.8产生一个在单写者/多读者共享存储器模型中的算法B，使得B是A的一个n模拟。由n模拟的定义可知，B是一致性问题的一个解法，且它保证1故障条件。这与定理12.8矛盾，因为定理12.8指出在读/写共享存储器模型中解决一致性问题是不可能的。　□

17.3 参考文献注释

Lynch、Merritt、Weihl和Fekete[207]以及Bernstein、Hadzilacos和Goodman[50]的书是实现原子事务的并发控制算法方面较好的参考资料。

*MajorityVoting*算法来自Gifford[137]，并分别由Herlihy[154, 149]以及Goldman和Lynch[140]做了扩展，后一个扩展还出现在[207]中。

586

*ABD*算法出自Attiya、Bar-Noy和Dolev[18]。他们的论文中还包括一个使用有界标志的算法和ABD模拟的一些应用，此算法基于Israeli和Li[162]提出的思想。在本章中$n \leq 2f$时的不可能性结果是在Bracha和Toueg[56]以及Attiya、Bar-Noy、Dolev、Peleg和Reischuk[20]的类似证明的基础上做了少量修改而得到的。

定理17.9，即容错异步网络中的一致性不可能性，来自Fischer、Lynch和Paterson[123]的工作。他们直接在网络模型上证明了这个结果，而不是像本章一样通过转换来证明。

17.4 习题

17.1 对定理17.1的证明概要中的声明进行证明——即对于任意x，所有自动机$R_{x,i}$与其间通道（隐藏了*send*和*receive*动作）的合成组成一个相应类型和接口上的原子对象B_x。

17.2 将*SimpleShVarSim*算法应用到共享存储器系统A中来产生系统B，找出并证明A的时间复杂度与B的事件复杂度之间的关系。对所做的任何假设做详细说明。

17.3 令B是将*SimpleShVarSim*算法应用到10.5.2节中的*PetersonNP*算法上而得到的异步网络算法。求出你能得到的B的时间复杂度的最佳上界，即求出从任意try_i事件到相应的$crit_i$事件所需的时间的最佳上界。你的结果与习题17.2中得到的一般上界有何不同？

17.4 研究问题：当*SimpleShVarSim*转换被应用在随机共享存储器系统（如11.4节中的*LehmannRabin*算法）上时，能够保证什么？证明你的结论。

17.5 写出17.1.2节中概述的读/写上锁算法（算法的概述出现在*MajorityVotingObject*算法的描述之前的几段中）的前提-结果代码，该算法在异步网络中模拟了单写者/多读者共享存储器算法。共享变量x的每个读者都应该保存x的一个本地副本并读取这个副本（当副本可用时）。写者应该使用双阶段协议来对各个副本执行写操作。所有操作都保证能够终止。给出并证明正确性结果。

587

17.6 将习题17.5中的答案扩展到多写者/多读者共享存储器算法中。

17.7 使用以下两种不同方法来将10.7节中给出的*Bakery*互斥算法转换到在异步网络上运行：

a）使用*SimpleShVarSim*；

b）使用习题17.5中求出的双阶段上锁策略。

比较两种结果算法的时间复杂度和通信复杂度。

17.8 扩展*MajorityVotingObject*算法，允许每个读操作访问一定读配额（*read quorum*）的副本，而不是读大多数副本；每个写操作访问一定写配额（*write quorum*）的副本。读配额和写配额不必要是严格大多数的。它们应该满足什么条件？使用前提-结果记号来描述算法，并证明其正确性。

17.9 在*ABDObject*实现中的读者代码中的“广播阶段”是必需的吗？如果不是必需的，描述去除它后算法如何工作；否则给出反例。

17.10 扩展*ABD*算法，使得它实现了一个多写者/多读者的读/写原子对象，且当$n > 2f$时，保证f故障终止。示范如何将该扩展结合到具有多写者/多读者寄存器的共享存储器模型的容错

异步网络模拟中去。

17.11 修改*ABDOject*算法，使得它使用有界而不是无界标志（提示：采用对固定k取模的方法是不够的，需要使用一个具有更有趣结构的有限数据类型*D*。参考[162]来看看一种数据类型如何工作的。写者进程需要根据数据类型*D*来不断地选择“更大的”标志，并知道由慢进程保存的所有旧标志都可以被那些标志“小于”新标志的进程检测到。所以，当写者选择一个新标志时，它需要将由任意进程保存的所有标志考虑进去。为了让写者跟踪这个集合，当一个进程修改它的局部标志时，它首先保证大多数进程知道它采用了新的标志。然后写者总是能够决定所有进程中的可能标志，这可以通过向大多数进程查询
588 消息来做到。若要更多的提示，参考[18]）。

17.12 对在满足$n \leq 2f$的异步网络中实现快照原子对象的问题，给出并证明类似于定理17.6的结果。

17.13 证明定理17.7。

17.14 参照*SimpleSRSim*算法的代码风格，写出*SimpleBcastSim*算法的前提-结果代码。证明其正
589 确性（定理17.8）。

第18章 逻辑时间

在这一章里阐述简化异步网络编程工作的第三种主要方法：引入逻辑时间的概念。在我们的异步网络模型中，并没有实际时间的内建概念。然而，通过特殊的协议还是有可能推行逻辑时间概念。如果系统的用户不太在意发生在不同网络位置的事件的相关次序，逻辑事件有时候是可以代替实际时间的。

18.1 异步网络的逻辑时间

基本思想是给一个异步网络系统A的每一运行的每一个事件赋予一个“逻辑时间”，这个“逻辑时间”是某一固定全序集合T[⊖]的一个元素。典型地，这个集合T要么是一个非负整数集，要么是一个非负实数集（也可能附加其他类型的值（比如进程索引）作为附加条件）。这些逻辑时间不需要和实际时间有任何特殊的关系。然而，不同事件的逻辑时间必须能够表达这些事件在系统A中的所有可能依赖关系，就如同14.1.4节中所述的那样。基于这样的假设，能够证明：对于进程来说，逻辑时间赋值看起来就像是实际时间赋值一样。

我们将分别考虑发送/接受系统的逻辑时间和广播系统的逻辑时间。在这一章里，假设所有的通道都是第14章中定义的特殊的通用通道。我们不考虑故障的情况。

18.1.1 发送/接收系统

考虑一个具有通用的可靠FIFO发送/接收通道的异步发送/接收系统。假设底层的网络图是任意强连通有向图。这样一个系统有以下几种类型的事件：用户接口事件——使进程自动机和系统用户通信，发送和接收事件——使进程自动机和通道自动机交互，还有进程自动机的内部事件。（我们不需要考虑通道的内部事件，因为我们使用的特殊的通用通道没有任何内部事件。）

令α是异步发送/接收网络系统A的一个运行，那么对α的一次逻辑时间赋值被定义为给属于α的每一个事件赋予一个T中的值，而且这种赋值必须按照一种能够和α中事件之间依赖关系保持一致的方式进行。特别地，要求满足以下四种属性：

1）没有两个事件会被赋予同样的逻辑时间。

2）每一进程中，事件的逻辑时间必须按照它们在α中的发生次序严格递增。

3）任一发送事件的逻辑时间必须严格小于相应的接收事件的逻辑时间。

4）对任意给定的值$t \in T$,只能有有限多个事件的逻辑时间值小于t。

属性2和属性3暗示逻辑时间顺序必须和ordering$\rightarrow_{trace(\alpha)}$保持一致，就如14.1.4节中定义的那样。然而，我们允许在不同进程中的一些事件的逻辑时间顺序和它们在α中的顺序相反。

我们强调对网络中的所有进程的任何逻辑时间赋值都要“看起来”像实际时间赋值一样。特别地，对于所有进程来说，任何一个具有逻辑时间赋值*ltime*的公平运行α看起来都像另一个公平运行α', 在α'里*ltime*表现的就像实际时间一样，也就是说，在α'里所有的事件以它们的*ltime*的顺序发生。

定理18.1 令α是具有通用可靠FIFO通道的发送/接收网络系统A的一个公平运行，*ltime*是对α的一次逻辑时间赋值。那么存在A的另一个公平运行α', 使得：

⊖ T必须满足一个技术上的假设：必须存在一个递增的T的元素的序列$t_1, t_2, \ldots\ldots$，使得每一个$t \in T$都有某个上界t_i。

1）α'包含和α一样的事件。

2）α'中事件以它们在α中的*ltime*的顺序发生。

592 3）α'和α对于任何进程自动机都是不可区分的[⊖]。

定理18.1规定在α和α'中每个特定进程中的事件必须以同样的次序发生，不过，它允许属于不同进程的事件能够以不同的次序发生。

证明　令γ为根据α中事件的*ltime*的顺序重新排列事件得到的序列。逻辑时间定义的属性1到属性4暗示这个序列是惟一的。那么我们可以使用推论14.2来推导所需要的公平运行α'的存在。在应用推论14.2的时候，我们把所有的进程动作都看成是外部的（比如通过重新分类）。由逻辑时间定义的属性2和属性3可以得到重新排列后仍然保持$\rightarrow_{trace(\alpha)}$。这一点推论14.2也是需要的。　□

例 18.1.1　发送/接收图

考虑一个基于三节点完全无向图的发送/接收系统A，在A的一个运行α中，消息按照图18-1的模式被发送和接收。

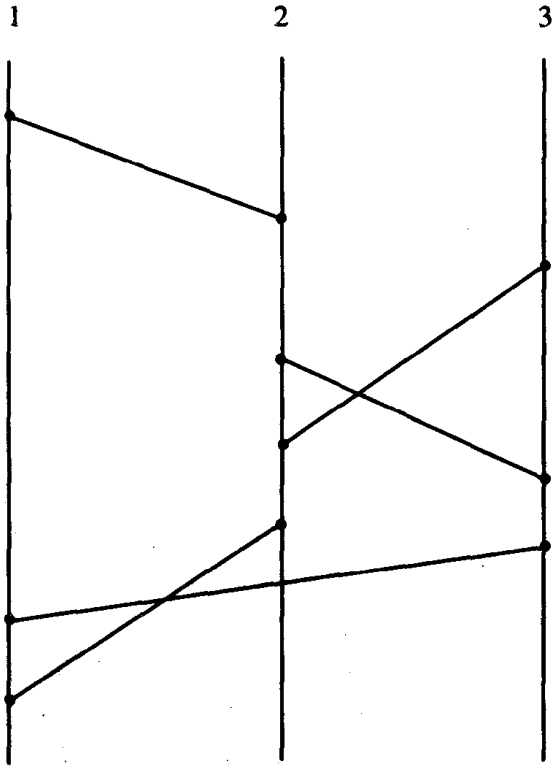


图18-1　运行α的发送/接收图

在这个发送/接收图中，用一条垂直线表示每个进程的运行，时间的方向向下。圆点表示发送和接收事件，每条斜线连接一条消息的发送事件和接收事件。在这里我们没有描述进程内部事件和进程与用户的通信事件等其他事件，它们可以由垂直线上的其他圆点表示。

图18-2展示对α的一次逻辑时间赋值*ltime*（假设α仅仅包含发送和接收事件）。因为时间按向下的方向进行，*ltime*并不符合事件在α中的次序，不过，它是和α中事件间的所有可能的依赖关系相一致的。

⊖　这里我们使用8.7节中“不可区分”的正式定义。

图18-3描述α中事件以它们*ltimes*的次序重新排列，从而产生定理18.1中描述的α'。注意到在α和α'中每个进程中事件的顺序是一样的。

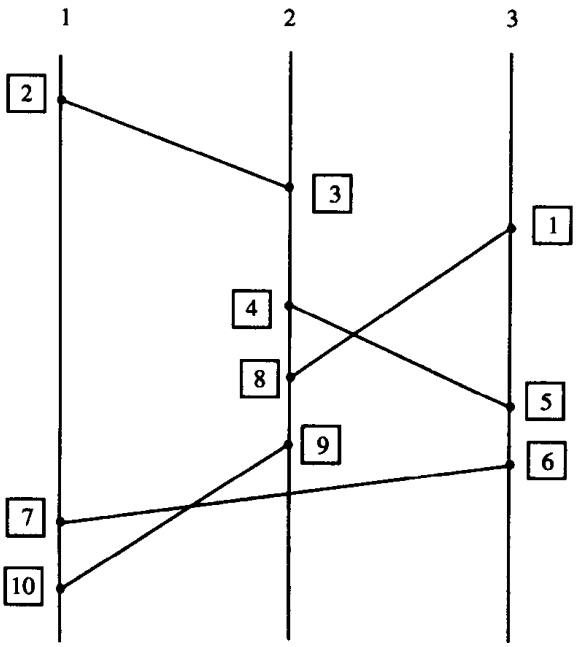


图18-2 对α的一个逻辑时间赋值

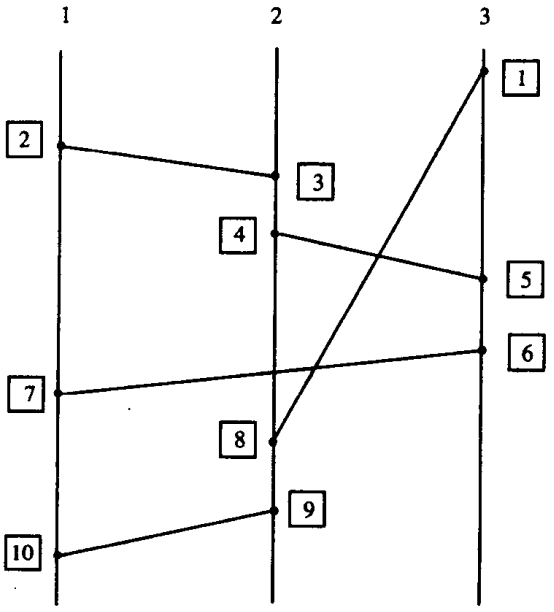


图18-3 重新排序后运行α的发送/接收图

注意，这一节的思想和16.2节中关于本地和全局同步的思想是紧密相关的。每种情形都定义了每个运行中捕获事件之间所有可能依赖性的事件的一个依赖性次序。从而，在各自的情形下，运行的事件被重新排序，这是保持了所有依赖关系但是按照一个全局概念的时间（同步周 593
期或者逻辑时间）进行的重新排列。（一个本地的同步器或者逻辑时间的定义被使用来证明这是可行的。）每种情况下，结论都是在本地重新排序后的运行与原来运行是不可区分的。这样，对原来运行中的所有参与者来说，看起来它们就好像在全局同步下运行。

18.1.2 广播系统

我们也可以为具有通用可靠广播通道的可靠异步广播系统定义逻辑时间。在这里，事件包括用户接口事件、广播和接收事件以及进程内部事件。

令α是一个异步广播系统的一个运行。对α的一个逻辑时间赋值定义为对α中的每个事件赋予一个T中的值，并且要满足发送/接收系统中的同样属性，除了属性3现在为：

3）广播事件的逻辑时间严格小于相应的每一个接收事件的逻辑时间。 594

如同发送/接收系统一样，我们有：

定理18.2 令α是一个具有通用可靠广播通道的广播系统A的一个公平运行，令*ltime*是对α的一个逻辑时间赋值。则存在这样一个公平运行α'：

1）α'包含和α一样的事件。

2）α'中事件以它们在α中*ltime*的次序发生。

3）α'和α对于任何进程自动机都是不可区分的。

证明概要 类似于定理18.1的证明，但是这次是基于推论14.4。具体证明留做一道习题。 □ 595

18.2 使用逻辑时间的异步算法

在上一节中，我们为异步接收/发送和广播系统定义了逻辑时间这个概念。现在我们给出两个算法来为一个给定的异步发送/接收网络算法A中的事件产生逻辑时间。这两个算法中的每一个算法实际上都是一个算法变换，这个变换基于同样的底层网络有向图将给定的算法A“变换”为一个新的异步发送/接收算法$L(A)$。这个变换一个进程一个进程地进行，根据A_i（A系统的进程i）来定义$L(A)_i$（$L(A)$系统的进程i）。$L(A)$中的进程以某种方式合作来“模拟”A的一个公平运行，这里每个$L(A)_i$模拟相应的A_i。当$L(A)$中的一个进程模拟A的一步时，这个进程同时也“产生”一个逻辑时间值。我们在一些术语上使用引号（比如说“变换”，“模拟”，“产生”）表示我们并没有这些术语的惟一清楚的含意，但是在不同的情况下会对这些术语作出稍微不同
596 的解释。

上面我们所描述的两个转换都可以修改后应用到广播系统中。

18.2.1 时钟的走动

下面是一个简单的算法变换，用来为一个给定的异步发送/接收网络算法A的一个运行产生逻辑时间。我们用发现人Lamport的名字来命名这个转换为*LamportTime*转换。这个转换的基本原理是维护本地时钟，当收到消息时使时钟向前走动，从而使所有的本地时钟能够获得足够的同步。这里逻辑时间域T是二元组（c，i）的集合，c是一个非负整数而i是一个进程索引；这些二元组是按照字典顺序排序的。

LamportTime变换：

进程$LamportTime(A)_i$中保存着进程A_i的状态以及一个取非负整数值、初始值为0的本地变量*clock*。每当进程i中发生一个事件（包括用户接口事件、发送和接收事件以及内部事件），这个*clock*变量就至少增加1。任一个事件的逻辑时间就定义为这个事件刚刚发生后的变量*clock*的值和作为附加条件的进程索引i所组成的二元组。

每当进程i产生一个发送事件，它首先将它的*clock*变量的值加1来得到发送事件的*clock*值v，然后它将值v作为时戳添加到即将发送的消息中。而当进程i产生一个接收事件，它增加它的*clock*变量的值，增加后的*clock*值不仅要严格大于*clock*原来的值，而且还要严格大于接收的消息的时戳。这样得到的新的*clock*的值就是接收事件的*clock*值。

更精确地，进程i中$LamportTime(A)$算法的代码如下所示：

***LamportTime*$(A)_i$:**

Signature:

As for A_i, except that $send(m)_i$ and $receive(m)_i$ actions are replaced, respectively, with $send(m,c)_i$ and $receive(m,c)_i$ actions, where $c \in \mathbb{N}$.

States:

As for A_i, plus:

597 $clock \in \mathbb{N}$, initially 0

Transitions:

As for A_i, with the following modifications:

Input action $\neq receive$

Effect:

As for A_i, plus:

$clock := clock + 1$

$send(m,c)_i$

Precondition:

As for $send(m)_i$ in A_i, plus:

$c = clock + 1$

Effect:

Locally controlled action $\neq$ *send*
  Precondition:
    As for A_i.
  Effect:
    As for A_i, plus:
    $clock := clock + 1$

  As for $send(m)_i$ in A_i, plus:
  $clock := c$

$receive(m, c)_i$
  Effect:
    As for $receive(m)_i$ in A_i, plus:
    $clock := \max(clock, c) + 1$

Tasks:
As for A_i (modulo the replacements).

因为每个进程每一步都增加它的*clock*以及因为均衡的打破，所以*LamportTime*(*A*) 明显满足逻辑时间定义的属性1和属性2。由对接收事件的处理可以得到属性3。由每一个事件都促使和它关联的*clock*变量至少加1这一点可以得到属性4。

根据在这一节开始时非正式提到的条件，每个A_i的产生*LamportTime*(*A*) $_i$的“变换”只是简单地添加新的*clock*组件，再加上维护它的一些代码段。这并没有添加全新类型的操作或者延迟事件。这个“模拟”逐步进行，直接得到*A*的一个公平运行。当进程*LamportTime*(*A*) $_i$模拟A_i的一步时，“产生”的逻辑时间正是二元组（c, i），其中*c*就是执行完这一步后的*clock*的值。

广播 很容易地修改*LamportTime*变换以应用到异步广播系统中。

18.2.2 延迟未来事件

现在我们给出另一个为发送/接收网络算法*A*的一个运行产生逻辑时间的算法变换。根据发现者Welch的名字，我们命名它为*WelchTime*。与*LamportTime*类似，*WelchTime*也是基于维护本地时钟的，只不过这次并不是一收到消息就使时钟前进，而是延迟“太快”到达的消息。从某种意义上说，这个变换比*LamportTime*更具有“侵略性”，因为它在底层执行的事件中引入了延迟。这里逻辑时间域*T*是三元组（*c*,*i*,*k*），*c*是非负实数，*i*是一个进程索引，$k \in \mathbb{N}^+$；三元组是字典顺序排列的。 598

***WelchTime*变换：**

每个进程*WelchTime*(*A*)$_i$维护一个本地变量*clock*，它的值是非负实数。我们假设进程*i*的*clock*值是由一个单独的任务维护的，这样就可以保证*clock*的值是单调非递减的并且可以是无限大的。

任一个事件的逻辑时间被定义为事件发生时的*clock*值，同时进程索引作为第一个附加判断值（当同一进程中的事件具有相同的*clock*值的时候）以及一个表示执行次序的序列号码作为第二个附加判断值。注意：在算法*A*的任何事件的执行期间*clock*值都是不会变化的。一个发送事件的*clock*值作为一个时戳被附加到即将发出的消息上。

每个进程*i*都维护着一个FIFO队列*receive-buffer*，用来保存那些时戳大于等于本地当前*clock*值的消息。当一个消息到达进程*i*时，它的时戳被首先检查。如果时戳小于本地当前*clock*值，这个消息被立刻处理；否则，这个消息被放入*receive-buffer*。在每一个本地控制的非时钟步，进程*i*首先取出并处理*receive-buffer*中所有时戳小于当前本地*clock*值的消息；这些消息按照它们在*receive-buffer*中出现的次序被处理。

这个算法可以说是模拟当*A*中相应消息被处理时（而不是当这个消息刚到达进程*i*的时候）的一个*receive*(*m*)$_i$事件。和这个接收事件相关联的*clock*值是这个消息被处理时*clock*值。

WelchTime(*A*)的属性4可以由本地*clock*变量的无上界属性得到。本地*clock*变量的无上界性质也意味着在一个*receive-buffer*中的每个消息最终都会被处理，所以每个接收事件也最终会被

模拟并被赋予一个逻辑时间。这样，每一个事件都确实拥有一个逻辑时间。属性1和属性2可以由附加条件和本地时钟的单调性得到。属性3可以由*receive-buffer*的性质保证。

根据前面非正式提到的条件，用于产生$WelchTime(A)_i$的每个A_i“变换”加入并管理*clock*、*receive-buffer*和序列号附加条件组件。在这个变换中，A_i的接收动作可以被延迟。这样这个
599 “模拟”产生一个A的公平运行，这个公平运行根据一些其他的事件重新对A中的一些接收事件排定次序。每次进程i模拟A的一步，“产生”的逻辑时间正是三元组（*clock*，i，k），k是作为第二附加条件的一个序列号。

需要注意的是，当本地时钟远远不能同步的时候，由*WelchTime*转换引入的延迟量将会非常大。这个算法在时钟紧密同步的情况下工作得最好。

广播　可以很容易地修改*WelchTime*算法变换使其在异步广播系统中工作。

18.3 应用

这一节中，我们给出一些将逻辑时间用于异步网络算法的简单应用。

18.3.1 银行系统

我们考虑在习题15.43中给出的这个问题，计算一个银行系统中所有钱款的总数，这个银行系统没有外来的存款或者提款，钱款只是通过消息在进程之间转移。

这个银行系统可以用一个用户接口中没有动作的异步发送/接收网络算法A来模仿。每个进程有一个本地变量*money*来保存当前存在本地的钱款数量。发送和接收动作使用参数来表示要操作的钱款数量。A中的进程决定何时以及向何地发送多少数量的钱款。我们做一个技术上的假设：每个进程发送给它的每个邻居的消息的数量不受限制。这不是一个严格的限制——加入包含0块钱的哑元消息总是可能的。

我们希望有一个异步发送/接收网络算法，其中每个进程决定一个本地的余额，所有的余额的总和就是系统中正确的钱款总数。这个算法的运行是由系统中一个或者多个地点收到的外部信号触发的。（这些信号可以在同一时间或者不同时间发生在不同的地点。）

所以，我们假设算法A以某种方式（比如使用*LamportTime*或者*WelchTime*）转换成一个新的系统$L(A)$，这个新的系统模拟A并且为A的事件产生逻辑时间。然后，从$L(A)$的更进一步的变换可以得到所需的算法*CountMoney*，这里*CountMoney*的每一个进程$CountMoney_i$负责“监视”
600 相应$L(A)$的进程$L(A)_i$的工作。[⊖]

***CountMoney* 算法：**

算法的核心是一个使用预先决定的逻辑时间$t \in T$的“子例程”，假设t对所有进程是已知的。假设t已知，总的策略是：

1）对A的每个进程，在所有逻辑时间小于等于t的事件之后和所有逻辑时间大于t的事件之前决定*money*变量的值。

2）对每个通道，决定所有在逻辑时间小于等于t发送但是在逻辑时间严格大于t接收的消息所包含的钱款总量。

特别地，每个$CountMoney_i$进程负责决定进程A_i的*money*变量的值以及所有引入到A_i中的通道所包含的钱款的总量。

要决定这些量，进程$CountMoney_i$将每个发送事件的逻辑时间附加到即将发送的消息上，

⊖ 这个结构对转换后的算法$L(A)$做了一些技术上的假设：这个模拟是一步一步的，并且以一种进程*CountMoney*能够识别的格式产生A的每一步和逻辑时间。

作为时戳。为了决定进程A_i的*money*变量的值，进程$CountMoney_i$跟踪A_i最近模拟的事件之前和之后的*money*的值。当模拟A_i的第一个逻辑时间严格大于t的事件时，$CountMoney_i$返回记录的这个事件之前的*money*变量的值。（肯定存在这样一个事件，因为A_i执行无限多的事件而逻辑时间小于等于t的事件是有限的。）

为了决定从j到i的通道中的钱款数量，进程$CountMoney_i$需要决定哪些消息是其中$send_j$事件的逻辑时间小于等于t的消息、哪些消息是其中$receive_i$事件的时间严格大于t的消息。这样，由A_i的第一个逻辑时间超过t的事件（比如$CountMoney_i$决定A_i的*money*值的那个事件）开始，进程$CountMoney_i$记录从通道中到达的消息。只要消息的时戳小于等于t，它就持续记录这些消息。当一个时戳严格大于t的消息到达时，$CountMoney_i$返回所有记录的消息的钱款数量的总和。（这样一个消息肯定会到达，因为A_j发送无限多的消息给A_i。）

601

每个进程$CountMoney_i$（在子例程中）计算得到的款额是进程A_i和所有引入的通道决定的值的总和。

记住所有这些都假定有一个预先决定的逻辑时间t。由于实际上并不存在这样一个预先决定的t，所以进程需要一些机制来得到一个。（选择一个任意的t是行不通的，因为这个逻辑时间可能在执行子例程之前就已经通过某个进程。）比如，进程可能使用一些预先决定的递增的逻辑时间序列t_1，t_2，...，其中每一个$t \in T$都对某些i有$t \leq t_i$，并且进程企图去完成所有处于这些逻辑时间的子例程（并行地）。通过广播它们的结果，进程可以决定第一个子例程在所有地方都成功了的第一个t_i，并且使用那个子例程的结果。

我们讨论对任何给定的t的子例程的正确性。首先，对任何给定的*CountMoney*的公平运行，可以看到总策略都可以产生正确的钱款总额。这个运行模拟了A的一个公平运行α，并给α赋予一个逻辑时间*ltime*。定理18.1暗示存在A的另一个包含和α相同事件的公平运行α'，并且对所有的进程A_i来说，α'与α是不可区分的，在α'中所有的事件是根据它们的*ltime*值的次序发生的。总策略所做的就是在发生任何*ltime*=t的事件后立即“切开”运行α'并记录这种情况下在所有进程和所有通道中的钱款。这样，总策略在运行α'中的给出整个系统的一个瞬时全局快照，这个快照当然能得出银行系统中的正确钱款总量。

应该直接可以看出分布式算法实际上正确地实现了总策略。

例 18.3.1 *CountMoney*算法的运行

图18-4显示银行算法A的一个公平运行α的一个接收/发送图，其中$L(A)$赋予了相关联的逻辑时间，在每个进程的相应的时间线的顶部都有初始的钱款的数量，而在每条消息的边上都标有传输的钱款的数额。

现在考虑模拟运行α的*CountMoney*算法的一个公平运行。假设这次运行使用了值t=7.5。图18-5在前一幅图的基础上加上虚线来表示（连接）逻辑时间7.5与进程时间线相交的地方。

602

在*CountMoney*的运行中，进程1决定进程A_1的*money*的值为\$10 – \$1+\$5=\$14；进程2决定A_2的*money*值为\$20+\$1 – \$3=\$18；进程3决定A_3的*money*值为\$30 – \$2+\$3 – \$5=\$26。除了进程3到进程2的通道包含\$2以外，所有的通道都被确定为空的。钱款的总量可以确定为\$14+ \$18+ \$26+ \$2 =\$60，这也是实际的总和。

图18-6包含被记录的运行α'的发送/接收图，事件按照逻辑时间的次序出现。这里，与t=7.5对应的虚线是水平的并且切到一条边，就是从进程3到进程2的那条边。可以很容易地看到：计算出的总和精确地刻画了7.5时刻α'中所有进程和通道的情况。

603

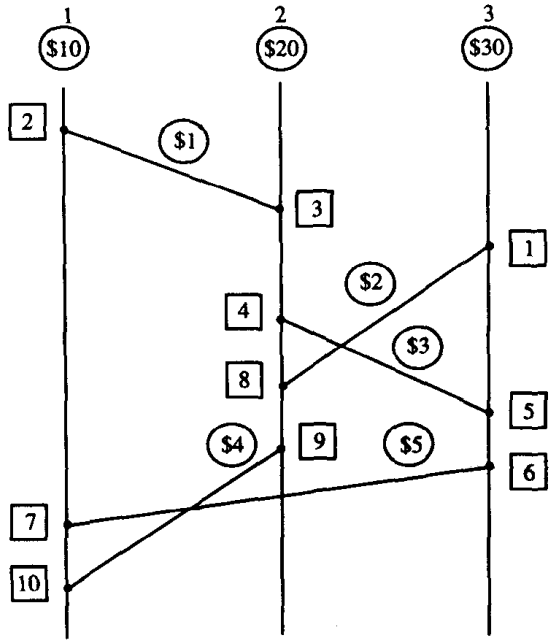


图18-4 银行算法A的运行α

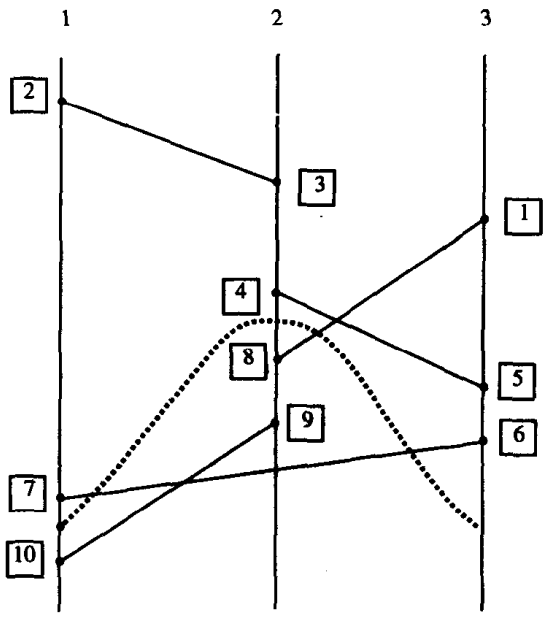


图18-5　t=7.5的虚线

我们强调*CountMoney*算法在A的操作中，除了那些已经$L(A)$引入的之外，没有引入任何新的延迟。

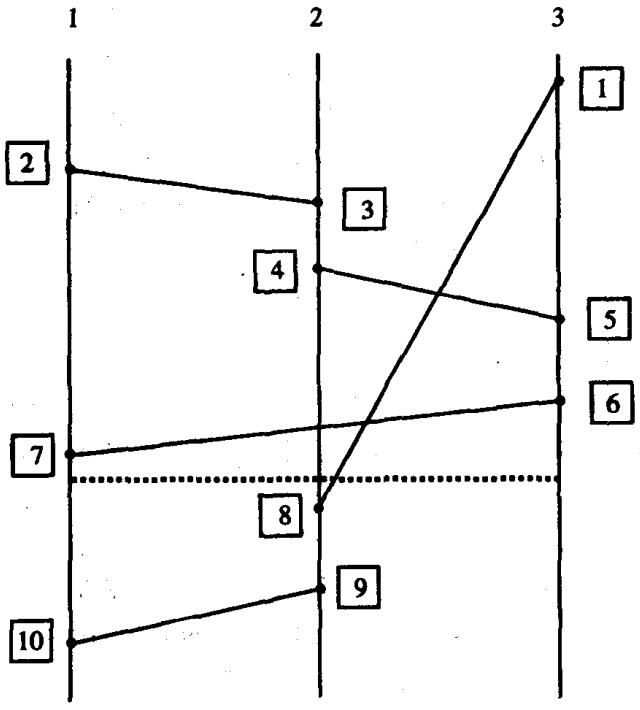


图18-6　带有t=7.5水平线的重排序运行α'

18.3.2　全局快照

*CountMoney*算法的思想可以推广到银行系统外的任一个异步发送/接收系统。（像前面一样，我们假设每一个进程A_i向它的每一个邻居发送消息数量是不受限制的。）设想我们要得到A的运

行过程中某一点的系统状态的瞬时全局快照。这可能是很有用的，比如说，为了调试、为了防止系统失败而建立的系统状态的一个备份版本、或者为了检测例如像算法是否在所有地方都结束了这样一些全局属性。通过延迟所有的进程和消息是有可能得到一个瞬时全局快照的，因为只要记录下所有需要的信息即可；不过，这种策略在绝大多数实际大小的分布式系统中都是不实用的。 604

但是对某些应用，一个真实的瞬时全局快照可能并不是必需的；一个“看起来像是”所有进程瞬时全局快照的系统状态可能就足够好。我们在上一段列举了一些这种应用的例子；在第19章中将给出其他的例子。在这种情况下，获得整个银行系统的钱款总额的策略可以改为提供一个异步发送/接收网络系统A的可以接受的全局快照。如前所述，A首先是按逻辑时间前进的。

***LogicalTimeSnapShot*算法：**

如同*CountMoney*一样，这个算法的核心思想是一个使用一个预先决定的逻辑时间$t \in T$的子例程，t被假设为被所有进程已知。假设t是已知的，则策略为：

1）决定所有逻辑时间小于等于t的事件之后和所有逻辑时间大于t的事件之前的A的每一个进程的状态。 605

2）对每一个通道，决定在逻辑时间小于等于t时刻发送而在逻辑时间严格大于t时刻接收的消息序列。

这些信息是采用在*CountMoney*中相同的分布式算法决定的。

任何*LogicalTimeSnapshot*的公平运行都模拟A的一个公平运行α，并对α进行逻辑时间赋值*ltime*。由定理18.1可以得出返回的全局状态是对A的另一个公平运行α'的一个瞬时全局快照，这个α'包含和α一样的事件，这些时间按*ltime*排序，并且α'和α对所有进程来说都是不可区分的。这些已经足够来，比如说，建立一个系统状态的可以接受的备份版本。

18.3.3 模拟一台单状态机器

逻辑时间也可以被用来使一个分布式系统去模拟一台集中式状态机，或者换句话说，一个共享的单一变量。回忆一下9.4节中关于一个变量类型的正式概念；它由一个值的集合V、一个初始值v_0、调用和回应的集合以及一个函数f：调用 $\times V \rightarrow$ 回应 $\times V$组成。我们现在展示如何在异步广播网络模型中“实现”给定变量类型的一个共享变量x。

我们考虑这样一种情况，有n个用户进程向x提交调用并从x处得到回应，网络中的节点i与用户进程U_i一一对应。我们假设每个用户进程串行地发出调用，也就是说，进程一直等待直到收到上一次调用的回应之后才发出新的调用。我们希望用户能得到这样一个视图，即对所有的被应用的操作来说只存在一份单独的x的拷贝。更精确地，网络作为一个整体（包括隐含的发送/接收操作）应该是一个给定类型的原子对象，就像13.1节中定义的那样。我们没有强制规定回复性要求；我们只要求良构性、原子性和无故障终止。

对这个问题有多种可能的解法，其中一些在17.1节中讨论过。比如，一个进程可以维护单独一份x的拷贝，并在这份拷贝上执行所有的操作——见*SimpleShVarSim*算法。这里我们考虑这样一种解法，每个进程都持有一份私有的x拷贝；每个调用都被广播给所有的进程，然后在各自的拷贝上执行。发起操作的进程在自己的本地拷贝上执行操作，并可以决定所需的回应。为了使这种策略正确工作——保证所有的进程按照一样的次序在自己的拷贝上执行操作，并且这些操作发生的点应该在各自的调用－回应间隔内——需要某些同步。我们使用逻辑时间概念来获得所需的同步。 606

***ReplicatedStateMachine*算法：**

这个算法由一个琐碎的算法A开始。每个进程都简单地收到来自用户U_i的调用并广播它们。

（当这些调用被收到时由哪些进程处理这些消息并不重要。）另外，如果有必要的话，A_i 广播哑元消息，来保证它可以无限频繁地广播。逻辑时间通过一个变换被加入到A中，如前所述一样，产生$L(A)$。

主要的算法使用$L(A)$。除了一个本地的x的拷贝外，每个进程i还使用一个本地变量*invocation-buffer*来保存它曾经所收到的所有调用以及它们*bcast*事件的逻辑时间。进程i在执行一个本地调用的*bcast*时将这个本地调用放入它的*invocation-buffer*，而在对一个远程调用（也就是发生在另一个进程中）执行*receive*时将这个远程调用放入它的*invocation-buffer*。

进程i还维护一个初始为0的向量*known-time*，以跟踪它所收到的每个进程的最大逻辑时间。这样，*known-time*$(i)_i$就是进程i中最近发生的事件的逻辑时间，而对任何$j \neq i$，*known-time*$(j)_i$就是i从j收到的最后一个消息的*bcast*事件的逻辑时间。

当下列条件都为真时，进程i可以在它的*invocation-buffer*中执行一个对它的x的拷贝的调用π。

1）调用π在还没有被进程i应用到x上的*invocation-buffer*$_i$中的所有调用中逻辑时间最小。

2）对每个j，*known-time*$(j)_i$至少和π的逻辑时间一样大。

当进程应用一个对它的x的拷贝进行操作的本地调用时，它将x的回应返回给用户。

引理18.3　*ReplicatedStateMachine*算法实现一个原子对象。

证明　良构性是显而易见的。我们讨论终止性。考虑任何公平运行α，逻辑时间的属性1表示α的每一次调用都被赋予一个惟一的逻辑时间。属性4表示对任何特定的t，α中只有有限多的调
607 用的逻辑时间小于t。这样，只存在一个惟一定义的α中的调用序列$\prod$，其中调用按照它们*bcast*事件的逻辑时间来排序。

可靠广播保证每个进程最终总能将每个调用都放入它的*invocation-buffer*。因为A的每个进程发生事件的数量是不受限制的，属性4意味着每个进程的逻辑时间无限地增长。每个进程可以广播无限多次的事实意味着每个进程的*known-time*向量的每个单元也是可以无限增长的。这样我们可以说，通过归纳序列$\prod$中调用的位置，每个调用最终都可以应用到x的每份拷贝。这表示每个调用都产生一个回应表示终止。

现在我们讨论原子性。考虑任何（有限或无限）运行α。如前所述（比如，定理17.4的证明），我们可以假设α中不存在未完成操作。

我们首先说明每个进程按照操作的逻辑时间无间隔地应用操作到它的本地x拷贝。这是因为当进程i应用一个逻辑时间为t的操作π到x的时候，它明确地检查到它并不知道有任何逻辑时间小于t的未决操作，并且它的对所有进程的*known-times*都至少等于t。然后任一对进程之间广播通道的FIFO属性意味着进程i决不会收到任何其他逻辑时间小于t的调用。

现在我们为α的每一操作定义一个串行化点。也就是说，为在每一个进程i中发起的并且*bcast*事件的逻辑时间为t的操作π，选择系统中所有进程都到达逻辑时间大于等于t的最早的点为串行化点。（通过按照逻辑时间的次序排列串行化点来打破平衡。）我们知道在α中迟早会到达这样一个点，因为我们已经讨论过进程i必定要应用操作π到它的本地x拷贝。然而，在它的所有*known-times*都至少是t之前它是不可能做到这一点的，它的所有*known-times*都至少是t意味着所有的进程逻辑时间都至少达到t。

注意因为逻辑时间的属性2以及t是π的*bcast*事件的逻辑时间的事实，任何操作π的串行点都不可能在π的任何调用之前。另一方面，π的串行点不可能在π的任何回应之后，因为发起π的那个进程在所有它的*known-times*都至少是t之前是不会向用户回应的。这样，串行点发生在操

作的间隔里。

因为串行点以逻辑时间的次序出现，这也是在本地拷贝上执行的操作的次序，所以原子性所要求的“收缩”性质满足了。 □

我们并不能很明显地看出*ReplicatedStateMachine*比简单的集中式*SimpleShVarSim*算法优越 608
的地方；毕竟，*ReplicatedStateMachine*本质上要求每个进程做到在集中式算法中一个进程所做的工作。在不同进程的逻辑时间正好紧密同步的情况下我们可以看到一个优点。这种情况下，在*SimpleShVarSim*算法中执行一个操作的时间大致相当于一个双向消息延迟。另一方面，在*ReplicatedStateMachine*算法中，只要进程i发现所有其他进程都达到由它发起的操作π的*bcast*事件赋予的逻辑时间，它就可以立即执行操作π。如果时钟被紧密同步，这仅仅需要大致相当于一个单向消息的延迟。

*ReplicatedStateMachine*可以被用来实现一个共享内存系统的分布式实现中的所有共享变量。这种方法是17.1节中建议的实现技巧之外的另一种选择。

读操作的特别处理 假设在共享变量x上实现的一些操作是读操作（或者，更通俗地，不修改变量的值、而是返回一个回应的操作）。那么可以修改*ReplicatedStateMachine*，根本不使用*invocation-buffer*机制而实现本地执行这些操作。这个修改会比那些原子对象有稍弱的正确性保证，但是对很多应用来说仍然是很合理的。

银行分布式数据库 *ReplicatedStateMachine*算法可以在共享变量x代表整个银行数据库的环境下使用。这种情况下，典型的操作将是*存款*、*取款*、*增加利息*等。这个数据库可能被复制到银行的每一个支行。对这样一个数据库的很多操作，修改的次序是非常重要的。比如，当余额低的时候，一个*取款*操作在一个*存款*操作之前或者之后调用，将会产生不同的结果。这样，由*ReplicatedStateMachine*算法保证的操作应用的一致性的次序是很重要的。

独立的支行能够从数据库的本地拷贝中读取信息常常是很有用的，即使这些信息并不完全是最新的。这种情况下，上面所述的对读操作的特别处理是很有用的。

互斥 我们已经在第10章针对异步共享内存模型和在第20章中针对异步网络模型定义过互斥问题。简单地说，用户通过*try*动作请求来对一个资源进行独占使用，而系统通过*crit*动作进行授 609
权。用户通过*exit*动作释放资源，系统通过*rem*动作作出回应。系统被设想为保证任何时刻最多只有一个用户拥有某项资源，并且只要有请求资源就会不停被授权出去。这里，我们也需要*lockout-freedom*，也就是说，每一个请求最终都会被满足。

*ReplicatedStateMachine*算法可以被使用到广播网络中来解决互斥问题。这时，共享变量x是一个进程索引的先进先出队列，支持的操作有*add*(i)、*first*(i)以及*remove*(i)。*add*(i)操作将制定的索引添加到队列的尾部。*first*(i)操作是一个查询，如果i是队列的第一个元素就返回*true*，否则返回*false*。*remove*(i)操作从队列中删除所有是索引i的元素。令B_x是x的一个原子对象，这里接口i支持所有带有参数i的操作。

当用户i通过try_i事件请求进入临界区时，进程i在原子对象B_x上调用*add*(i)操作，将i添加到队列尾部。然后进程i重复不停地调用*first*(i)操作直到返回的结果为真，这表示i到达队列的第一个位置。当i收到为*true*的结果时，它允许用户i进入临界区并产生事件$crit_i$。当用户i通过$exit_i$事件离开临界区时，进程i在原子对象B_x上调用*remove*(i)操作。当这个操作返回后，进程i通过rem_i操作允许用户i进入剩余的区域。（这本质上是10.9.2.节的*QueueME*算法。）这样通过使用原子对象B_x的实现，特别地，通过使用*ReplicatedStateMachine*解决了互斥问题（带有锁定权）。

然而，如果使用*ReplicatedStateMachine*，一点简单的优化是可能的。也就是，修改 *add*(i)操作使它返回一个值：如果队列里i有前驱j的话，就返回j，否则返回*null*。如果返回值是*null*，

那么i没有前驱，进程i就可以立刻执行$crit_i$。否则，进程i简单地等待直到它在队列的本地拷贝上对i的前驱j执行$remove(j)$（这时它知道用户j已经释放了资源）。接着它就可以执行$crit_i$，而$exit_i$在之前就已经被处理。

18.4 从实际时间算法到逻辑时间算法的变换*

到目前为止我们所论述的每个算法都是建立在一个由逻辑时间增强的异步算法A的基础上。另外一个设计策略就是使用一个“实际时间”的概念来开始设计一个算法，然后通过用逻辑时间替代实际时间来将其转换成另一个算法。

610 假设我们从一个异步发送/接收网络系统A开始，这里每个进程A_i都有一个初始为0，值域为$R^{\geq 0}$的本地变量*real-time*。假设所有进程的*real-time*变量都由一个全局的实时I/O自动机，通过$tick(t)$输出同时的设置所有进程的*real-time*变量为t，来维护。（这个I/O自动机模型允许一个单独的输出动作可以和不只一个的输入动作同步。）对这个实时自动机的惟一要求是作为输出事件的参数出现的时间在任何公平运行中应该是非递减的且无上界。[⊖] 进程A_i不能修改*real-time*变量。

然后就有可能将每一个进程A_i变换成进程B_i，B_i不使用实际时间，而是用逻辑时间。B_i没有*real-time*变量，而使用*clock*变量代替，并且就像A_i使用*real-time*变量一样使用*clock*变量。*clock*变量由B_i通过逻辑时间的一个实现来维护，这里逻辑时间域是$R^{\geq 0}$（或者$R^{\geq 0}$的一个子集）。

为了描述这个变换能保证什么，我们考虑系统A及它的变换版本B，每个都和（每个节点i一个的）用户自动操作U_i组合起来。我们可以得到：

定理18.4 对系统B的每一个公平运行α（比如，B加上用户自动操作），都存在一个对每个U_i来说都是不可区分的系统A（A加上实时自动机加上用户）的公平运行α'。

这就是说，对每个单独的用户，B的每个公平运行看起来都像是A的一个公平运行。

例 18.4.1 银行系统

可以设计一个类似于*CountMoney*但是使用实际时间的算法来计算一个银行的钱款的总量。也就是说，每个进程在它发现它的*real-time*变量超过t的那步之前记录它的*money*变量的值。然后它记录所有那些具有以下性质的引入的消息，这些消息在发送者的*real-time*变量小于等于t时发送，但是在进程i的*real-time*变量大于t时接收。

611 所得到的算法可以转换成一个使用逻辑时间的算法。

18.5 参考文献注释

逻辑时间的概念是Lamport在他的著名的论文“Time,Clocks and the Ordering of Events in a Distributed System”[176]中提出的。这篇论文也包括*LamportTime*算法变换以及对*ReplicatedStateMachine*算法核心思想的一小段描述。Lamport后来扩展了复制状态机器方法以使其能够容忍有限数量的错误[179]。Schneider[255]写了一篇关于使用复制状态机来实现容错服务的综述。

*WelchTime*算法变换由Welch[286]引入；Neiger和Toueg[232]也研究了同一变换，而Chaudhuri、Gawlick和Lynch[74]则将这个算法扩展成一个部分同步的模型。

⊖ 既然实时只是一个普通的I/O自动机，我们不能对它输出出现的“频率”做出任何假设。在第23~25章中，我们考虑一个这种频率假设可以表达的模型。

*CountMoney*和*LogicalTimeSnapshot*算法是与Chandy和Lamport[68]的一致全局快照算法紧密相关的。

这一章中诸如银行数据库这样的例子由Lynch、Merritt、Fekete和Weihl在[207]中进行了广泛的讨论；讨论集中在银行和其他数据库中的原子操作。

习题18.17中刻画的“向量时钟”算法参见Mattern[222]，Liskov和Ladin[197]，以及Fidge[115]。它应用在Isis系统[52]中。对向量时钟应用的综述参见[256]。

18.6 习题

18.1 证明定理18.2。

18.2 为*WelchTime*算法变换编写和*LamportTime*代码一样风格的“代码”。

18.3 描述一个发送／接收网络的逻辑时间的实现，逻辑时间的值域是$R \geq 0$。

18.4 在一个周五深夜工作会议上，吃着比萨饼，几个Flaky计算机公司的程序员发明了4个异步发送/接收网络系统的“逻辑时间”概念。这四个概念中的每一个都是通过删去逻辑时间的四个属性中的一个得到的。他们认为这些概念对某些应用可能会很有用。对他们这4个概念中的每一个，

a）描述加入这种逻辑时间概念到在一个给定的异步网络算法*A*的运行上的算法变换。 612

b）描述可能的应用。

18.5 *CountMoney*算法被明确表达为一个应用到底层银行系统*A*的双重算法变换，这样可能很难看出发生了什么。这道习题中，你将把不同的不同组成一个单独的算法。

a）针对18.3.1节中允许的那种类型的任何特意的银行系统*A*，编写预处理代码。也就是，你要设定所有进程初始的钱款数量，一些决定何时向谁传递钱款的规则，以及传送多少。

b）针对a)加入逻辑时间后的修订版的算法*A*，编写预处理代码。你可以选择你喜欢的算法来产生逻辑时间。

c）针对b)使用*CountMoney*策略来产生所需余额的算法的修改版本，编写预处理代码。注意确定加入用来决定一个适当的逻辑时间*t*的机制。

18.6 重新考虑18.3.1节中的银行系统例子。现在假设底层的银行系统*A*除了传输外还允许存款和取款（建模为系统用户接口的输入操作）。如果我们如前一般应用同样的*CountMoney*转换，那么产生的系统的输出是什么呐？

18.7 修改*LogicalTimeSnapshot*算法，使其适应广播系统而不是发送/接收系统。仔细表述你的算法能够保证什么？

18.8 在*CountMoney*和*LogicalTimeSnapshot*算法中，逻辑时间是附加在每条消息上的。开发另外一种算法，并不将逻辑时间附加在消息上，而是在每条通道发送一条单独的额外的*marker*消息来表示在逻辑时间小于等于*t*发送的消息和逻辑时间大于*t*发送的消息之间的分隔点。并证明你的算法的正确性。

18.9 在习题13.21的基础上给出引理18.3的另一种证明。

18.10 假设在*ReplicatedStateMachine*算法中使用“逻辑时间”，一种满足属性1、2和4但是不满足属性3' 的逻辑时间。那么能得到什么属性？ 613

18.11 开发18.3.3节中描述共享变量的修改实现，使其能本地处理读操作。总的来说，证明它并没有实现一个原子对象。详细陈述有哪些正确条件它没有满足。

18.12 在18.3.3节最后部分的优化的互斥算法是分成几部分描述的：一个加入逻辑时间的简单的异步算法*A*，*ReplicatedStateMachine*算法，以及一个使用重复队列的主算法。将这几部分

组合成一个单独的算法。为你的算法编写预处理代码并给出你的算法的证明概要。

18.13 *ReplicatedStateMachine*算法使用逻辑时间来在广播网络模型中实现一个原子对象。你怎样修改它以使其工作在发送/接收网络模型中？

18.14 给出定理18.4的详细证明；这需要准确的描述转换。

18.15 为模拟异步发送/接收网络中的单写者/多读者共享内存算法编写一个基于逻辑时间的算法。这必须是与17.1.2节中的两阶段锁策略不同的方法。共享变量x的每个读者都保持x的一份本地拷贝。每个对x的读和写操作都要被赋予一个逻辑时间，并且每个操作都必须按照它们逻辑时间的顺序在本地拷贝上执行。所有的操作都要保证能够结束。

给出预处理代码，描述结果并证明之，最后分析复杂度。

18.16 如果习题18.15变为多写者/多读者共享内存算法，概括你的回应。

18.17 考虑通过将T由一个全序集合改为一个偏序集合来弱化逻辑时间的定义为弱逻辑时间。但是，逻辑时间定义的属性1~4必须满足。这样，就不是所有时间都必须按逻辑时间顺序相关，但是相互依赖的事件（位于同一节点的事件，或者发送事件的相应的接收事件）必须仍然相关。

a）给出使用弱逻辑时间的定理18.1的版本。它必须根据和给定偏序一致的一个任意的全序来陈述。证明你的结果。

614

b）为一个给定异步网络算法A的运行开发一个产生弱逻辑时间赋值的算法转换。如果事件之间有依赖性，那么和事件关联的时间只能是根据底层偏序T相关的。（提示：一个基于长度为n的非负整数的集合T的算法。如果对所有的i有$C(i) \leq C'(i)$并且对某些i有$C(i)<C'(i)$，则我们说$C<_T C'$；也就是说，向量C'的所有元素都至少和C的相应元素一样大，在某些元素上C'严格大于C。

每个进程i维护一个本地时钟，是T中的一个向量，初始全为0。当进程i发生事件时，$clock_i(i)$至少增加1。当进程i发送一个消息时，它首先增加$clock_i(i)$，然后将得到的向量作为时戳附加在消息上。当进程i收到一个消息时，它首先增加$clock_i(i)$，然后将它的时钟向量设置为新增加的时钟向量和消息时戳向量中最大的一个。）

证明你的变换实际上产生了一个弱时间赋值并且如果时间之间有依赖的话则事件的时间是按T相关的。

615

第19章 一致全局快照和稳定属性检测

在这一章里，阐述四种简化异步网络编程方法中的最后一种，即对运行中的异步网络算法A进行监测。比如，一个监测算法可以：

- 通过检查有没有违反想要得到的不变式来协助对A的调试
- 产生A的全局状态备份版本
- 检测A何时终止运行
- 检测A的某些进程是否陷入“死锁”，这是一种几个进程都在互相等待对方的状况
- 计算由A管理的一些全局量（比如，钱款的总额）

本章聚焦在两个概念上：*一致全局快照*和*稳定属性检测*。一个全局快照返回A的一个全局状态，即由A的所有进程和通道的状态组成的一个集合。如果在一个进程看来，一个快照是*在系统中的同一瞬时到处发生的*，那么就说这个快照是“一致的”。这样一个快照对上面所列的所有任务都是有用的。A的一个稳定属性是A的全局状态的一个属性，如果这个属性曾经是真的，那么它将保持永远为真。稳定属性的例子有系统终止和死锁。

每个监测算法都被描述成原始算法A的一个变换版本$B(A)$；更详细地，$B(A)$基于和A一样的底层图，并且每个$B(A)_i$进程都是根据相应的A_i进程定义的。$B(A)_i$不是表达为一些新的I/O自动机和A_i的简单组合，因为这个新的$B(A)_i$需要获取A_i的状态。更确切地，$B(A)_i$被描述为添加了一些新的状态组件和动作并且对旧的动作做了一些修改。我们限制这些变化以使得它们不会过多干扰A的操作。 617

19.1 发散算法的终止检测

我们以考虑一个异步发送/接收算法A的*终止检测*问题开始，这个算法A属于一个特别简单的类型，称作*发散算法*。

19.1.1 问题

我们假设底层图G是任意无向连通图。我们假设在算法A中，所有进程的初始状态都是*安静的*（如8.1节中所定义）。也就是说，只有输入动作是允许的。我们考虑这样一种环境下的A，即对一个单独的（任意的）进程仅仅供给一个单独的输入事件。根据I/O自动机的定义，对一个进程来说这样一个输入的到达可以使这个进程执行局部控制动作，包括发送消息给其他的进程。这些消息随后就会唤醒接收进程，而接收进程又会发送更多的消息，如此下去。算法A被称为是*发散的*，因为所有活动都是从输入发生的地方开始并且通过消息“发散”到网络的某些部分中。

如果在A的一个全局状态中，没有进程可以执行局部控制动作，并且通道中也没有消息，则称这个全局状态是*安静的*。（这再次和8.1节中关于*安静*的定义一致，只不过这次是应用于代表整个算法A的单个I/O自动机。）A的终止检测问题[⊖]如下：如果在某个进程A_i发生某个输入之后，算法A曾经达到一个安静的全局状态，那么最终一个特殊的$done_i$输出在节点i上执行。

实际的终止检测，包括$done$输出，将由一个*监测算法*$B(A)$来执行。$B(A)$也是一个基于和A

⊖ 这一章中，我们使用“终止”来表示安静；在这本书的大多数其他地方则表示系统产生一个回答。

一样的图G的发送/接收网络算法。监测算法$B(A)$的每个进程自动机$B(A)_i$根据相应的进程自动机
618 A_i来定义。为了得到$B(A)_i$，我们允许对A_i做如下改变。

- 除了A_i的所有状态组件外，$B(A)_i$还可以包含新的状态组件。
- $B(A)_i$的开始状态在A_i的状态组件上的投影必须正好是A_i的开始状态。
- 除了A_i的动作之外，$B(A)_i$可以包含新的输入、输出以及内部动作。
- 原来A_i的动作在$B(A)_i$中可能包含新的信息。比如，一个$send(m)_i$动作可能被变换为一个$send(m,c)_i$动作。在$B(A)_i$中A_i的动作保持原来的前提条件并且属于和原来一样的任务划分类。它们对A_i的状态组件的作用和以前一样，但是它们可能还会作用在新的组件上。
- $B(A)_i$的新输入动作只能改变$B(A)_i$的新组件的值。
- $B(A)_i$的新局部控制动作的前提条件可能会牵涉到$B(A)_i$的整个状态，包括所有旧的和新的状态组件。但是，新的局部控制动作只能影响$B(A)_i$的新状态组件。它们被归类到$B(A)_i$的新任务划分类中。

19.1.2 *DijkstraScholten*算法

我们阐述解决发散算法终止检测问题的*DijkstraScholten*算法。算法的思想是改进底层的算法A，即创建和维护一个当前A牵涉到的图的节点的生成树。这棵树的根是源节点，即输入发生的那个节点。生成树的创建与15.3节中的*AsynchSpanningTree*算法相似，但是更复杂，因为它允许这棵树反复地收缩和生长，并多次融入同一个节点。（同样的复杂性出现在15.4节中*AsynchBFS*的终止协议中。）

619 ***DijkstraScholten*算法(非形式化)：**

算法使用的消息是A的消息加上一个*ack*消息。A中的消息被看作*AsynchSpanningTree*算法中的搜索消息。除了源进程之外的每个进程都指定从哪个邻居首次接收一条A的消息并将它作为生成树中的父亲节点。任何随后的A的消息都立刻得到确认；只有第一个A的消息保持不被确认（目前）。还有，源进程在收到任何A的消息后都立刻给予确认。这样，随着A的消息在网络中的发送，协议牵涉到的节点的生成树就创建成功。

现在，我们允许生成树可以“收缩”，通过一个敛播过程来向源进程报告终止。特别地，每个*DijkstraScholten*$(A)_i$进程都期待以下局部条件同时成立：

1）A_i的状态是安静的。

2）所有外出A消息都得到确认。

当它发现这样一个情况，它就“清理”：一个非源进程向它的父亲发送一个确认消息并且删除所有与这个协议有关的消息，而一个源进程报告它完成了任务。

一个类似的清理过程在15.4节的*AsynchBcastAck*算法中描述过。但是在这里，当一个进程清理后，它可能收到另一个A消息，从而促使它再一次加入生成树的创建。实际上，取决于低层算法A的消息传递模式，这可能发生任意多次。也就是说，*DijkstraScholten*(A)算法的生成树可以反复的生长和收缩并且可以在不同的时候以不同的方式生长。

例 19.1.1　生成树的生长和收缩

假设底层的图G由四个节点1、2、3和4组成，如图19-1所示，考虑以下情形。这里我们使用$DS(A)_i$来作为进程*DijkstraScholten*$(A)_i$的速记。

a）进程A_1收到一个输入，苏醒过来，并且向它的邻居A_2和A_3发送消息。

b）进程$DS(A)_2$和$DS(A)_3$收到从A_1来的消息并且设置它们的父亲指针指向节点1，随后A_2和
620 A_3醒来并互相发送消息。因为每个$DS(A)_2$和$DS(A)_3$都已经有了父亲，它们回答以确认。接着，

每个A_2和A_3发送一个消息给A_4。

c）A_2的消息首先到达进程$DS(A)_4$，所以$DS(A)_4$设置它的父亲指针指向节点2并立刻确认A_3来的消息。现在进程A_1、A_2、A_3和A_4继续它们的工作一段时间，在需要的时候互相发送消息；每个消息都立刻得到确认。接着，A_2达到一个安静状态。$DS(A)_2$还不能清理，因为它仍然没有收到它发给A_4的初始消息的确认。 621

d）现在A_4达到一个安静状态。因为$DS(A)_4$没有未确认的A消息，它发送一个确认给它的父亲，接着$DS(A)_2$清理，忘记所有关于这个协议的一切。当$DS(A)_2$收到这个确认时，A_2仍然处于安静状态并且$DS(A)_2$现在收到它发出的所有A消息的确认；因此，$DS(A)_2$发送一个确认给它的父亲，$DS(A)_1$，然后清理。接着，A_3发送消息给A_2和A_4。

e）当A_2和A_4收到这些消息，它们如同以前一样醒来，重置它们的父亲指针指向节点3，并继续执行算法A的工作。

这个执行可以以这种方式无限地继续下去，随着算法A相应部分的安静，生成树的一些部分生长和收缩。但是如果全部算法A都曾经变得安静，那么这棵树最终会收缩到源节点1。如果A_1到达一个安静状态并且$DS(A)_1$收到所有发出去的消息的确认，那么$DS(A)_1$可以宣布终止。

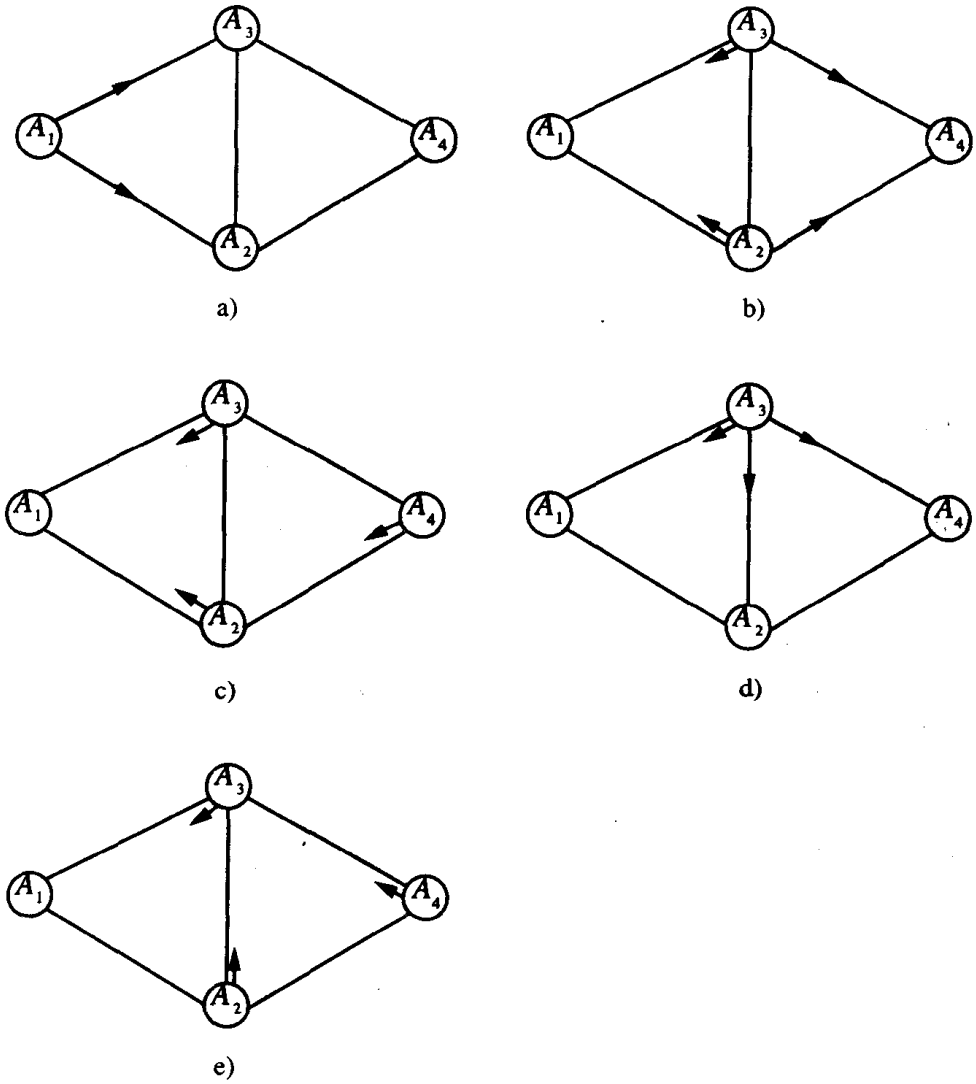


图19-1　*DijkstraScholten*(*A*)的一个运行。边上的箭头代表正在传递的消息；平行于边的箭头代表父亲指针

*DijkstraScholten(A)*中进程*i*的代码如下所示。变量*deficit*用来跟踪显著确认的数目。

DijkstraScholten$(A)_i$ automaton (formal):

Signature:

As for A_i, plus:

Input:
- *receive*("*ack*")$_{j,i}$, $j \in nbrs$

Internal:
- *cleanup*$_i$

Output:
- *send*("*ack*")$_{i,j}$, $j \in nbrs$
- *done*$_i$

States:

As for A_i, plus:

status $\in \{idle, source, non\text{-}source\}$, initially *idle*

parent $\in nbrs \cup \{null\}$, initially *null*

for every $j \in nbrs$:
- *send-buffer*(j), a FIFO queue of *ack* messages, initially empty
- *deficit*(j) $\in \mathbb{N}$, initially 0

622

Transitions:

Input of $A_i \neq$ *receive*
- Effect:
  - As for A_i, plus:
  - *status* := *source*

receive(m)$_{j,i}$, m an A message
- Effect:
  - As for A_i, plus:
  - if *status* = *idle* then
    - *status* := *non-source*
    - *parent* := j
  - else add *"ack"* to *send-buffer*(j)

Locally controlled action of $A_i \neq$ *send*
- Precondition:
  - As for A_i.
- Effect:
  - As for A_i.

send(m)$_{i,j}$, m an A message
- Precondition:
  - As for A_i.
- Effect:
  - As for A_i, plus:
  - *deficit*(j) := *deficit*(j) + 1

send("*ack*")$_{i,j}$
- Precondition:
  - *"ack"* is first on *send-buffer*(j)
- Effect:
  - remove first element of *send-buffer*(j)

receive("*ack*")$_{j,i}$
- Effect:
  - *deficit*(j) := *deficit*(j) − 1

cleanup$_i$
- Precondition:
  - *status* = *non-source*
  - state of A_i is quiescent
  - for all $k \in nbrs$
    - *deficit*(k) = 0
- Effect:
  - add *"ack"* to *send-buffer*(*parent*)
  - *status* := *idle*
  - *parent* := *null*

done$_i$
- Precondition:
  - *status* = *source*
  - state of A_i is quiescent
  - for all $k \in nbrs$
    - *deficit*(k) = 0
- Effect:
  - *status* := *idle*

Tasks:

As for A_i, plus:

{*done*$_i$}

```
{cleanup_i}
for every j ∈ nbrs:
    {send("ack")_{i,j}}
```

可以很清楚地看到*DijkstraScholten*(*A*)的任何全局状态都投影出*A*的一个全局状态，而*DijkstraScholten*(*A*)的任何公平运行都投影出*A*的一个公平运行。为了证明*DijkstraScholten*(*A*)正确地检测出*A*的终止，我们首先证明一个包含多个不变式断言的引理。关键的不变式是最后两个：倒数第二个表明父亲指针构成了非空闲进程的一棵生成树，而最后一个暗示一个*done*报告表示*A*已经变成安静的。

623

引理19.1　在一个包含节点i上的输入的运行之后的*DijkstraScholten*(*A*)的任意状态中，以下为真：

1）$status_i \in \{source, idle\}$ 并且 $parent_i = null$。
2）对每个$j \neq i$，有$status_j \in \{idle, non\text{-}source\}$，并且如果$status_j = non\text{-}source$，那么$parent_j \neq null$。
3）对每个j，如果$status_j = idle$，那么A_j的投影状态是安静的，$parent_j = null$，并且对每个k有$deficit(k)_j = 0$。
4）对每个j和k，$deficit(k)_j$是下面四个量的和：从j到k的通道里的A消息数量，$send\text{-}buffer(j)_k$里的*ack*的数量，从k到j的通道里的*ack*的数量，如果$parent_k = j$再加上1。
5）如果$status_i = source$，那么父亲指针构成了一棵以i为根的有向树并且刚好是由$status \neq idle$的节点集组成的。
6）如果$status_i = idle$，那么对所有的j和所有空的通道都有$status_j = idle$。

证明　此证明留做一道习题。　□

定理19.2　*DijkstraScholten*(*A*)算法检测到一个发散算法*A*的终止。

证明　引理19.1的第6点和第3点暗示着如果*DijkstraScholten*(*A*)宣布终止，那么实际上*A*已经变成安静的。我们还必须证明所需的活性属性：如果*A*变成安静的，那么最终*DijkstraScholten*(*A*)会宣布终止。

采用反证法。假设在*DijkstraScholten*(*A*)的一个公平运行α中，算法*A*变为安静的并且没有*done*事件发生。那么，在安静点之后，没有更一步的*A*消息被发送和接收；这可以由父亲节点构成的树（如引理19.1第5点所述）不能进一步生长而得。最终，这棵树必然停止收缩，稳定为一棵固定的树*T*。（这棵树*T*必然至少包含源节点，因为我们假设没有*done*事件曾经发生过。）既然对于这棵树来说没有进一步的*A*的消息或者变化，那么最终在全局状态里各处也没有进一步的*ack*消息。因此，引理19.1第4点所述的构成$deficit(k)_j$的和的前三项必然是0，并且任何$deficit(k)_j$可能是非0的惟一途径是$parent_k = j$。但是那样的话*T*的任何叶节点*i*都可以执行清理，所以它最终也这么做。不过这表示*T*进一步收缩，这是一个矛盾。由此可以得到在α中最终必然有一个*done*事件发生。　□

624

复杂度分析　考虑包含一个*done*事件的*DijkstraScholten*(*A*)算法的一个运行。在α中发送的消息总数是$2m$，其中m是被包含的*A*的运行中发送的消息数。从*A* 安静到*done*事件发生的时间上界是$O(m(l+d))$，l和d如常一样定义。注意通信和时间复杂度并不直接取决于网络的大小，而是取决于发送的*A*消息数。如果*A*仅仅在网络的一小部分执行一小段时间，那么它通常将发送很小数目的消息，所以*DijkstraScholten*(*A*)仅仅导致相对较小的代价。另一方面，如果*A*发送大量的

消息那么*DijkstraScholten*(*A*)会代价十分昂贵。

例 19.1.2　广度优先搜索

回想15.4节的*AsynchBFS*算法，其中进程修正错误的*parent*信息直到这些信息稳定为止。如阐述的那样，这个算法不会终止，因为进程无法知道算法什么时候会变得安静。

为了将*AsynchBFS*算法表达为一个发散算法，我们做一个微小的变动，令进程i_0初始时为安静的并且以一个唤醒输入动作来唤醒它。然后我们应用*DijkstraScholten*算法来得到一个会终止的BFS算法。这是为*AsynchBFS*介绍的特殊终止策略的系统化版本。

19.2　一致全局快照

现在我们回到为一个运行着的异步发送/接收网络算法*A*拍摄*一致全局快照*的问题。非正式地说，如果一个快照对所有进程来说都好像是在同一时间在系统内每一个地方拍摄的，那么我们就说这个快照是“一致的”。

19.2.1　问题

再一次地，我们假设底层的图*G*是一个任意连通无向图。这里的底层算法*A*是任一发送/接收网络算法。这个快照是由一个监测算法*B*(*A*)摄制的，*B*(*A*)也是一个基于图*G*的发送/接收网络算法。同样地，监测算法*B*(*A*)的每个进程自动机$B(A)_i$根据相应的A_i来定义。

625　这次我们允许变化的类型和19.1.1节中我们允许的那些变化类型相比更一般化一些，但是仍然足以保证任何*B*(*A*)的公平运行“包含”一个*A*的公平运行。不同的地方在于现在我们允许$B(A)_i$“延迟”一个A_i的$send(m)_{i,j}$动作，直到$B(A)_i$在从*i*到*j*的通道中在*m*前面放置另外一个消息为止。

我们假设每个$B(A)_i$都有一个输入动作$snap_i$，该动作给$B(A)_i$发信号来告知快照摄制开始。我们要求在*B*(*A*)的任何公平运行中都包含至少一个*snap*输入事件，最终每个$B(A)_i$都将执行一个$report_i$输出，这个$report_i$输出包含A_i的状态以及所有通向A_i的*A*的通道的状态。

由所有$B(A)_i$报告的不同状态组成*A*的一个全局状态。我们要求这个状态满足一个一致性属性。也就是说，令α为*A*的一个公平运行，它被包含在一个给定的*B*(*A*)的公平运行中。存在另一个满足以下所有条件的*A*的公平运行α'：

1）对每一个A_i进程来说，α'与α是不可区分的。

2）α'的开始部分是α的一个前缀α_1，这个α_1发生在给定的*B*(*A*)运行的第一个*snap*事件之前。

3）α'的结束部分是α的一个后缀α_2，这个α_2发生在给定的*B*(*A*)运行的最后一个*snap*事件之后。

4）返回的状态正是α'的一个前缀之后的全局状态，其中这个前缀包括整个α_1但不包括α_2的任意事件。

这样，由进程可知，返回的全局状态是在*A*的运行过程中的某一点同时拍摄的。而且，这一点处于快照算法的开始和结束之间的某处。

例 19.2.1　银行系统

令*A*为例18.3.1中的银行系统。图18-4描绘*A*的一个运行，它包含在系统中3个进程之间的5次钱款转移。（忽略图中的逻辑事件标志。）假设监测算法*B*(*A*)的一些进程在运行开始的时候收到一个*snap*输入。那么一致全局快照算法可能返回的一个全局状态例子如图18-5所示。即，作为A_1、A_2和A_3各自的状态，$B(A)_1$、$B(A)_2$和$B(A)_3$返回\$14、\$18和\$26。除了从A_3到A_2的那个通道

626　（那里$B(A)_2$报告包含一个值为\$2的消息）之外，所有其他通道都被判定为空。所需的另一个运行α'如图18-6所示。

19.2.2 *ChandyLamport*算法

我们已经阐述一致全局快照问题的一个解法——18.3.2节中的*LogicalTimeSnapshot*算法。现在我们给出另一个算法——*ChandyLamport*全局快照算法，它和*LogicalTimeSnapshot*算法非常相似但是不使用显式的逻辑时间t。相反地（如习题18.8所建议的），它使用新的*marker*消息来指出在时间$\leq t$发送的消息和时间$>t$发送的消息之间的区分点。

***ChandyLamport*算法（非形式化）：**

当一个先前还没有涉及快照算法的$ChandyLamport(A)_i$进程收到一个$snap_i$输入时，它记录当前A_i的状态。然后它立刻向它的所有外出通道发送一条*marker*消息；这个*marker*消息指出本地状态被记录之前和之后发送出去的消息之间的边界。⊖

然后$ChandyLamport(A)_i$开始记录每个输入通道里到达的消息以获得这个通道的状态；它记录这个通道里的消息直到遇到一个*marker*。这个时候，$ChandyLamport(A)_i$在通道另一端的邻居记录它的本地状态之前，$ChandyLamport(A)_i$已经记录这个通道里发送的所有消息。⊜

还剩下一种情况要考虑：设想进程$ChandyLamport(A)_i$在完成记录A_i的状态之前收到一个*marker*消息。在这种情况下，一旦收到第一个*marker*消息，$ChandyLamport(A)_i$立刻记录A_i的当前状态，发出*marker*消息，并且开始记录输入消息。它收到*marker*消息的那条通道被记录为空。

代码如下。 627

$ChandyLamport(A)_i$ automaton (formal):

Signature:

As for A_i, plus:

Input:
- $snap_i$
- $receive(\text{“marker”})_{j,i}$, $j \in nbrs$

Output:
- $report(s, C)_i$, $s \in states(A_i)$, C a mapping from *nbrs* to finite sequences of A messages
- $send(\text{“marker”})_{i,j}$, $j \in nbrs$

Internal:
- $internal\text{-}send(m)_{i,j}$, $j \in nbrs$, m a message of A

States:

As for A_i, plus:

$status \in \{start, snapping, reported\}$, initially *start*

snap-state, a state of A_i or *null*, initially *null*

for every $j \in nbrs$:
- *channel-snapped*(j), a Boolean, initially *false*
- *send-buffer*(j), a FIFO queue of A messages and *markers*, initially empty
- *snap-channel*(j), a FIFO queue of A messages, initially empty

---

⊖ 比如，如果A是一个例19.2.1那样的银行系统，那么在*marker*之前发送的钱款并不包括在发送者记录的本地状态里，但是在*marker*之后发送的钱款被包括在内。

⊜ 在银行例子中，这表示$ChandyLamport(A)_i$已经计算了所有这样的钱款数目，即这些钱款是邻接节点在记录它的本地状态之前发出的，因此没有被邻接节点计算在内。

Transitions:

snap$_i$
Effect:
if *status* = *start* then
snap-state := state of A_i
status := *snapping*
for all $j \in nbrs$ do
add *"marker"* to *send-buffer*(j)

Input of $A_i \neq receive$
Effect:
As for A_i.

receive(m)$_{j,i}$, m an A message
Effect:
As for A_i, plus:
if *status* = *snapping*
and *channel-snapped*(j) = *false* then
add m to *snap-channel*(j)

628

send(m)$_{i,j}$
Precondition:
m is first on *send-buffer*(j)
Effect:
remove first element of *send-buffer*(j)

receive(*"marker"*)$_{j,i}$
Effect:
if *status* = *start* then
snap-state := state of A_i
status := *snapping*
for all $j \in nbrs$ do
add *"marker"* to *send-buffer*(j)
channel-snapped(j) := *true*

Locally controlled action of $A_i \neq send$
Precondition:
As for A_i.
Effect:
As for A_i.

internal-send(m)$_{i,j}$
Precondition:
As for *send*(m)$_{i,j}$ in A_i.
Effect:
add m to *send-buffer*(j)

report(s, C)$_i$
Precondition:
status = *snapping*
for all $j \in nbrs$
channel-snapped(j) = *true*
s = *snap-state*
for all $j \in nbrs$
$C(j)$ = *snap-channel*(j)
Effect:
status := *reported*

Tasks:
As for A_i, except:
internal-sends are in tasks corresponding to *sends* in A_i,
plus there are new tasks:
{*report*(s, C)$_i$: $s \in states(A_i)$, C a mapping}
for every $j \in nbrs$:
{*send*(m)$_{i,j}$: m a message}

定理19.3 *ChandyLamport*(*A*)算法决定*A*的一个一致全局快照。

证明 固定*ChandyLamport*(*A*)的任意一个公平运行，其中某个进程收到*snap*输入。我们首先证明每个进程最终都会执行一个*report*输出。一旦任何*snap*输入发生在某个*ChandyLamport*(*A*)$_i$进程中，这个进程记录A_i的状态并向它的所有通道发送*marker*。一旦任何其他*ChandyLamport*(*A*)$_j$进程从任何一个通道收到一个*marker*，如果它还没有记录A_j的状态和发送*marker*到它的所有通道，它就执行这些动作。鉴于图的连通性，*marker*最终会传播到所有进程，而所有进程也会记录它们的本地状态。还有，每个*ChandyLamport*(*A*)$_i$进程最终会完成对它的所有输入通道里的消息的收集（当它从每个通道都收到一个*marker*）。那么每个*ChandyLamport*(*A*)$_i$最终都会执行一个*report*，得证。

现在我们证明返回的全局状态是一致的。也就是说，我们令α表示被包含的A的公平运行（α中的*send*事件和*ChandyLamport*(A)运行的*internal-send*事件相对应），然后我们构造出所需的另一个运行α'和它的所需前缀。即，令α_1为α中第一个*snap*之前的部分，α_2为α中最后一个*report*之后的部分。运行α'以α_1开始以α_2结束；惟一的重排涉及在第一个*snap*和最后一个*report*之间的α中事件。 629

第一个*snap*和最后一个*report*之间的每个α中事件都发生在某个*ChandyLamport*(A)$_i$进程中。这些事件可以分为两个集合：S_1——在促使A_i的状态被记录的*ChandyLamport*(A)$_i$事件（*snap*$_i$或者*receive*(*marker*)$_{j,i}$）之前发生的那些事件，和S_2——在这个事件之后发生的那些事件。重排将所有的S_1事件放置在所有的S_2事件之前，同时保持每个A_i事件的次序以及和每个相应*receive*相对的*send*（由一个*internal-send*得到）的次序。这样一个重排是可能的，原因在于一个事实：在对A_i的状态的记录之后没有*internal-send*(m)$_{i,j}$事件发生，并且和*internal-send*(m)$_{i,j}$相对应的*receive*(m)$_{i,j}$事件发生在对A_j的状态记录之前。（如果一个*internal-send*(m)$_{i,j}$事件是在对A_i的状态的记录之后发生的，那么在*send-buffer*(j)$_i$中m被放置在*marker*之后。但是这意味着*marker*在m之前到达*ChandyLamport*(A)$_j$，这表示在m到达时A_j的状态已经被记录。）以这种方式重排α的事件并填写α中每个A_i的状态产生了序列α'。

现在考虑正好在所有的S_1的事件之后结束的α'的前缀α_3。我们断言α'和它的前缀α_3满足所有所需的属性：关键因素是所有进程返回的结果准确地组成了α_3之后的A的全局状态。很明显，返回的每个A_i的状态正好是α_3之后的A_i的状态，因为α_3被定义为刚好包括在记录A_i的状态之前的A_i的事件。我们还必须验证对通道的记录刚好给出α_3之后在A的通道里传送的消息。但是α_3之后从i向j传送的消息刚好是*internal-send*(m)$_{i,j}$事件是在记录A_i的状态之前发生且*receive*(m)$_{i,j}$事件是在记录A_i的状态之后发生的那些消息。这些消息刚好就是那些在*marker*之前而在*ChandyLamport*(A)$_j$记录A_j的状态之后从*ChandyLamport*(A)$_i$到达*ChandyLamport*(A)$_j$的消息，也刚好是这个通道里被*ChandyLamport*(A)$_j$所记录的那些消息。 □

有向图 很容易可以看出*ChandyLamport*算法在有向强连通图中工作得和无向连通图中一样好。

例 19.2.2 两美元银行

令A是例18.3.1中银行系统的一个简单特例，这里底层图G只有两个节点1和节点2，系统中钱款的总额为\$2。假设每个进程开始时都有\$1。我们把*CL*(A)$_i$作为进程*ChandyLamport*(A)$_i$的速记。 630

考虑图19-2中描绘的*ChandyLamport*(A)的公平运行。在这个图中，#符号代表*marker*。

a）*snap*$_1$发生，导致*CL*(A)$_1$记录A_1的状态为\$1。然后*CL*(A)$_1$发送一个*marker*给*CL*(A)$_2$并开始记录收到的消息。

b）A_1发送\$1给A_2；这个美元进入从*CL*(A)$_1$到*CL*(A)$_2$的通道，在*marker*之后。

c）A_2发送\$1给A_1。

d）A_1收到美元并且*CL*(A)$_1$将它记录在*snap-channel*(2)$_1$中。

e）*CL*(A)$_2$收到从*CL*(A)$_1$来的*marker*，记录A_2的状态为\$0，发送一个*marker*给*CL*(A)$_1$，记录输入通道的状态为空，并报告结果。

f）*CL*(A)$_1$收到从*CL*(A)$_2$来的*marker*，记录输入通道的状态为一个消息（在*marker*之前收到的\$1）构成的序列，并报告它的结果。

g）A_2收到美元。

算法返回的全局状态如图19-2h所示。它由以下部分组成：\$1在A_1中、\$1在从A_2到A_1的通道

中、没有钱款在A_2或者在从A_1到A_2的通道中。这产生了正确的总额$2。

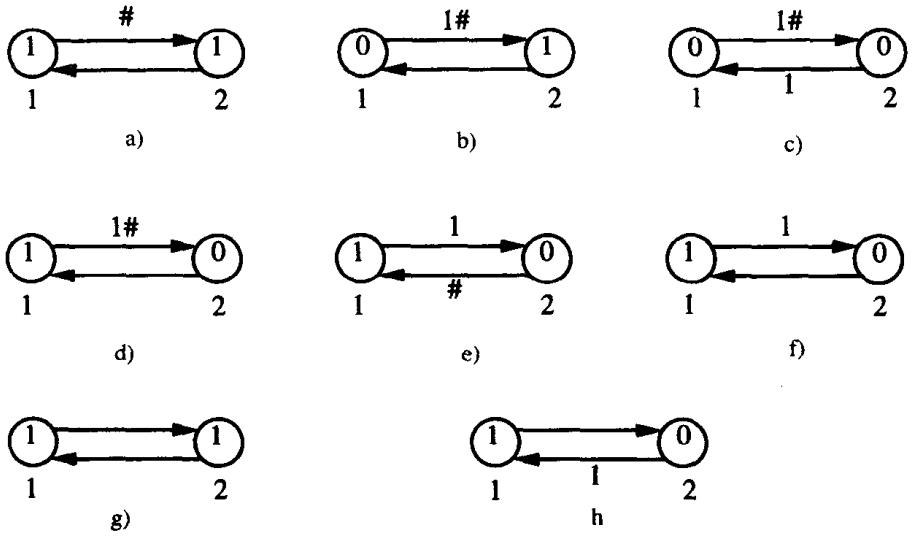


图19-2　针对两美元银行，*ChandyLamport*(*A*)算法的运行

注意：快照算法返回的全局状态实际上没有在被包含的*A*的公平运行α中表现出来。然而，它在*A*的另一个公平运行α'中表现出来，在α'中事件按照下面的顺序发生：a）A_2发送$1给$A_1$；
631 b）A_1发送$1给$A_2$；c）$A_1$收到$1；d）A_2收到$1。图19-3展示$\alpha$和$\alpha'$的发送/接收图。$\alpha$的图包括运行*ChandyLamport*(*A*)中A_1和A_2的状态在何处被记录的指示。α'的图展示在α'的创建中这两个记录点是如何排列的。快照算法返回的状态是第二幅图里由水平线表示的状态。

复杂度分析　除了*A*的消息之外，*ChandyLamport*(*A*)算法还使用$O(|E|)$条消息。从第一个*snap*事件直到最后一个*report*事件之间时间取决于通道和*send-buffer*里堆积的*A*消息数。如果我们忽略这些堆积，那么我们得到时间上界仅仅是$O(diam(l+d))$，但是忽略它们是不合理的。更实际的时间界限可以根据在快照时间里全局状态中各处出现的*A*消息的总数得到。

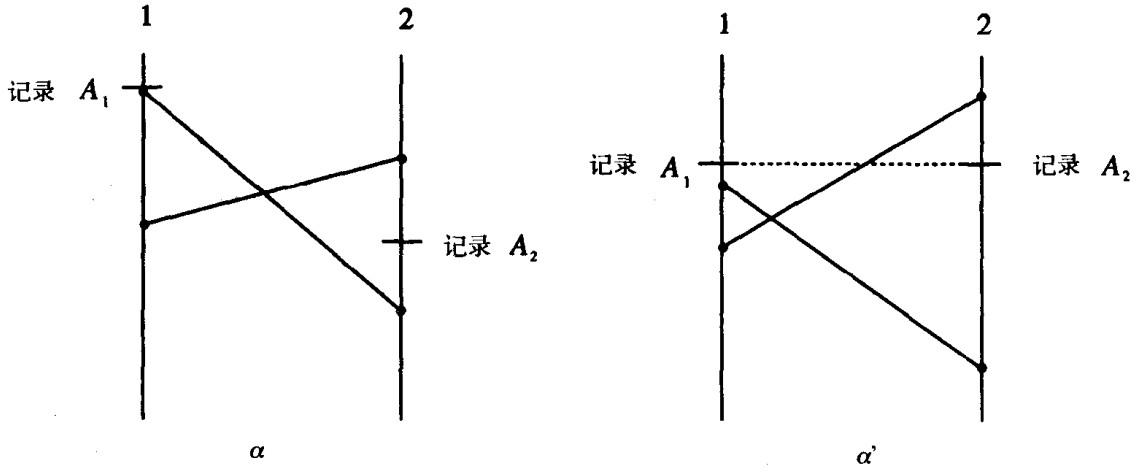


图19-3　α和α'的发送/接收图

19.2.3　应用

在这一小节里，我们给出一致全局快照的一些应用。

银行系统　*ChandyLamport*算法——或者任何其他产生一个一致全局快照的算法——可以用来计

算在这一章中描述的银行系统里的钱款总额。这个策略可以被推广到允许底层算法A管理其他 632
数量的情况。

分布式调试 一个一致全局快照算法可以用来协助调试分布式算法。一个分布式算法A的设计者可以（而且应该）通过关于A的全局状态的不变式断言来描述A的关键属性。一个调试器可以允许A运行，不时地获得一致全局快照并且检查对每个快照来说不变式是否为真。因为由快照算法返回的每个全局状态都是A的一个可达全局状态，对这些状态来说不变式应该为真。设计者可以在试图进行详细的推导证明这些不变式之前执行这些检查。比如，15.3节的*AsynchSpanningTree*算法有两个不变式，断言15.3.1和断言15.3.2，它们应该按这种方式得到检查。

需要一些工作来验证返回的全局状态中不变式是为真的。比如，全局状态信息可以传送给一个单独的进程，由它在本地验证这些不变式。或者，可以通过一个分布式算法使用快照算法返回的信息作为输入数据来验证。比如，断言15.3.1可以通过使用一个分布式算法来验证，即验证给定的父亲指针集合构成了一棵以一个给定的节点i_0为根的有向生成树；我们将这样一个算法的开发留做一道习题。断言15.3.2也可以由一个分布式算法来检查。在这里，这个分布式算法特别简单，因为不变式可以表达为一系列属性的关联，其中每个属性都可以在本地验证。（本地验证的结果可以*convergecast*给i_0。）

另一种调试策略是在单独一个处理机上使用A的一个集中化模拟。这里，使用A的模拟状态，不变式可以在A的每个模拟步后（或者经常地）得到验证。这种情况下不需要全局快照算法；这种做法的缺点是模拟会更耗时，因为全部工作都是在单独一个处理机上进行的。

稳定属性检测 一个异步发送/接收算法A的一个*稳定属性*是满足下列条件的A的全局状态的一个属性：如果对A的任何可达状态s，P为真，那么P对所有从s开始可达的状态P也为真。非正式地说，也就是如果在A的一个运行中P曾经为真，那么从那一刻起P就一直保持为真。

一个确定算法A的全局状态的一个稳定属性P是否为真的简单策略是使用一个全局快照算法获取一个一致全局快照，然后判断返回的全局状态中P是为真还是为假。同样地，这个判断可以通过在一个单独的进程中进行信息收集而作出，这个进程可以在本地对P做出判断，或者 633
通过一个分布式算法使用快照算法返回的信息作为输入数据作出。一个一致全局快照的正确性条件意味着下面几点：

1）如果快照状态中P为真，那么在快照算法的最后一个*report*之后的A的全局状态中P也为真。

2）如果快照状态中P为假，那么在快照算法的第一个*snap*之前的A的全局状态中的P也为假。

以上第一点为真是因为最后一个*report*之后的A的状态是从快照状态开始可到达的，而第二点为真是因为快照状态是可以从首次*snap*之前的A的状态开始可到达的。这个算法没有提供任何有关当快照算法在进行过程中A的全局状态中P是否为真的信息。

终止检测 现在我们回到终止检测问题。这次，考虑一个没有外部输入但是开始状态不一定必须是安静的发送/接收算法A。如果A曾经到达过一个安静的全局状态（其中没有进程能够执行局部控制动作并且通道中没有消息），一个终止检测算法最终应该输出*done*。

既然A没有外部输入，安静是一个稳定属性。所以终止可以通过使用检测稳定属性的通用策略检测到：摄制一个全局快照，然后判断是不是返回的全局状态是安静的。在这里，快照再次被进行，每个进程i可以判断它记录的A_i的状态是否是安静的以及它记录的引入通道的状态是否是为空。然后这个结果（一个每个进程都有的说明它的信息是否表示是安静的二进制位）可以汇聚发送给生成树中某个独一无二的进程。实际上，每个进程只需要汇聚发送一个单独的二进制位，说明所有在它这个子树中的进程是否报告了安静。

如果这个策略总结得出A已经终止，那么就可以保证确实是这样的。而且，如果快照被重

复地进行，这个策略可以保证终止最终被检测到。

例 19.2.3　广度优先搜索和最短路径

刚才阐述的策略可以用来为*AsynchBFS*和*AsynchBellmanFord*算法检测终止。快照可以由源
634 节点i_0初始化。如果回答是肯定的，就说明底层的算法已经终止，那么进程i_0可以向其他所有进程广播一个消息，告诉它们输出它们的结果。另一方面，如果回答是否定的，就说明底层的算法还没有终止，那么进程i_0必须继续执行快照直到某一次返回一个肯定的回答。

例 19.2.4　领导者选举

15.2节中的异步*OptFloodMax*领导者选举算法可以通过使用基于*ChandyLamport*快照算法的终止检测来改进，从而可以得到一个对任意无向连通图都能够终止的领导者选举算法。比如，一个快照可以由任意其中最大已知UID改变了的进程来初始化。在终止被检测到之前可能要执行一些快照。不同快照产生的消息可以加上这个快照的标识数字标签，从而可以区分不同的快照。

比较这个终止检测策略以及*DijkstraScholten*算法的代价是十分有趣的，尽管它们是用于不同类型的算法的。回想一下，*DijkstraScholten*(*A*)的通信和时间复杂度是依赖于发送的*A*消息的总数，而不是网络的大小。这样，如果*A*仅仅在网络的一小部分工作一小段时间，那么*DijkstraScholten*(*A*)的代价是比较小的。另一方面，快照策略总是牵涉到网络中所有的进程，所以它的代价依赖于网络的大小。但是在快照仅仅需要执行一次（没有堆积的*A*消息）的情况下，快照策略的代价并不依赖于发送的*A*消息的总数。这样，如果*A*工作的时间比较长，发送的消息数量比较多，那么快照策略应该比*DijkstraScholten*(*A*)工作得更好一点。

死锁检测　我们仅仅给出死锁检测问题的一个版本；实际上存在死锁问题的很多不变式。考虑一个发送/接收网络算法*A*，其中每个A_i进程都有本地状态来指示是否它正在“等待”邻居进程的某个子集（比方说，释放资源）。我们假设当A_i在等待一个非空邻居集合时，它处于安静的状态；实际上，直到它从每一个它等待的邻居收到一条消息（比方说，告诉它一个资源已经被释放）为止，它不能够执行任何局部控制步。在进程A_i从任何一个它等待的进程收到一个消息
635 后，它继续等待剩余的进程。我们进一步假设*A*没有外部的输入。

在*A*的全局状态中一个死锁是由两个或多个进程组成的环构成的，每个进程都在等待环中的下一个进程，从任何进程到它环中的前驱的路上也没有消息。死锁是一个稳定属性，因为一旦这样一个环模式被建立，环中没有进程可以再执行任何局部控制步。这样，我们可以使用检测稳定属性的通用策略来检测死锁：摄制一个全局快照，然后判断返回的全局状态中是否有一个死锁。这个判断可以通过在一个单独的进程收集信息并执行一个顺序的环检测算法作出（比如，使用深度优先搜索）。另外，这个判断也可以通过一个使用快照结果的分布式环检测算法作出。

这个策略可以保证仅仅检测真实的死锁。而且，如果反复地进行快照，可以保证最终检测到任何发生了的死锁。

19.3　参考文献注释

*DijkstraScholten*算法是由Dijkstra和Scholten[92]提出的。在他们论文中的阐述与这里的有很大不同；他们的论文提供算法的一个“衍生”和证明。*DijkstraScholten*的通用化是Francez和Shavit研究的，其中活动被允许在多个地点开始。其他关于终止检测的工作见Francez的论文[126]。*ChandyLamport*一致全局快照算法及其在稳定属性检测中的应用归功于Chandy和Lamport[68]。这个算法是根据Lamport早期在逻辑时间上的工作衍生而来的[176]。Fischer，

Griffetch和Lynch[118]设计了另一个一致全局快照算法，这个算法适合于基于事务的系统（如习题19.8讨论的那样）。

我们列出的对底层算法A的修改限制来自于程序语言Unity中*superposition*操作的定义，这种语言是由Chandy和Misra设计的[69]。有关分布式死锁检测的代表性的论文来自Isloor和Marsland[161]，Menasce和Muntz[224]，Gligor和Shattuck[138]，Obermarck[234]，Ho和Ramamoorthy[157]，Chandy、Misra和Haas[70],以及Bracha和Toueg[57]。这一章中的死锁检测方法和Bracha和Toueg[57]的方法很相近。Tay和Loke设计了一个模型可以用来理解某些死锁检测算法[274]。 636

19.4 习题

19.1 在*DijkstraScholten*算法里，算法牵涉到的进程生成树可以反复的生长和收缩，包含同一个进程很多次。在垃圾回收版本的*AsynchBcastAck*里没有这种行为——那里，一旦一个进程清理了自己的状态，它将永远不会再次加入到这个算法中。造成这种行为差异的原因是什么？

19.2 证明引理19.1。

19.3 对19.1节中可终止的广度优先搜索算法，通过应用*DijkstraScholten*到*AsynchBFS*，给出你能给出的最好的通信和时间复杂度上界。

19.4 描述如果通过同时使用*DijkstraScholten*算法和15.4节中的*AsynchBellmanFord*算法来得到一个可终止的最短路径算法。给出你能给出的最好通信和时间复杂度上界。

19.5 考虑一个以安静全局状态（如发散算法中一样）开始的算法A，使用一个可以在任意多的地点（每地点一个）提交输入的环境。设计一个算法来检测A何时到达一个安静的全局状态。这里我们说，当所有从环境中收到输入的进程都执行了*done*输出时，终止就被检测到。

19.6 给出定理19.3的证明的更多细节。

19.7 例19.2.1给出一个银行系统A的一个运行α、以及一个作为一致全局快照算法正确结果的全局状态。

a）描述一个特殊的返回这个全局快照的*ChandyLamport*(A)运行。你可以允许*snap*输入在任何时间发生在进程的任何子集。

b）将a)中的结果通用化。

19.8 我们考虑这一章中讨论的银行系统的扩展。假设给出一个发送/接收系统A，通过每个进程管理一些数据项，A中的进程维护一个分布式的数据库。A运行的惟一动作就是事务。这里，我们简单的定义一个事务为由一系列在数据项上的操作组成的一个串行程序；整个事务的原子性不是必须的。 637

问题是设计一个作为A的变换的新系统$B(A)$，来判断一个“合理的”A的事务一致快照。事务一致快照由某组事务完成后得到的每个A_i的一个状态组成。如果一个快照包括所有在快照算法开始之前结束的事务、以及在快照结束之前开始的其他事务的任意子集，那么就说这个快照是合理的。

变换B应该不应该过多干涉A的操作；比如，当它在获取快照的时候它不能停止所有的事务。

19.9 根据快照期间出现在全局状态里的A消息的数目，证明*ChandyLamport*(A)算法的时间复杂度上界。

19.10 考虑一个带有惟一节点i_0的连通无向图G。设计一个基于图G的异步发送/接收算法A来肯

定一个给定的固定父亲指针集合组成了以i_0为根的G的一个子图的有向生成树。更精确地，假设A的每个进程有一个值，它要么为一个邻居进程的索引，要么为*null*的父亲指针。输出应该是由进程i_0产生的。给出你的算法的前提-后果代码，证明其正确性，并分析其复杂度。

19.11 如例19.2.3所述，使用*ChandyLamport*快照算法来增强*AsynchBFS*算法以进行终止检测。

a）描述一个明确运行（对一个你选择的图G），其中进程i_0首先初始化*AsynchBFS*，然后初始化一个快照，并且由快照返回的状态不是安静的。

b）假设每当前一个快照返回一个否定的回答，i_0就初始化另一个快照。在成功地返回一个肯定的回答之前调用的快照的数目有上界吗？

19.12 对终止的*DijkstraScholten*和快照方法的比较仅仅对某种算法A有意义，其中两种终止策略
638 都能在A中应用。描述你可以找到的两种策略都适用的算法的最大类别。

19.13 考虑一组进程，其中每一个都可能在等待它的一些邻居。也就是说，每个进程都有一个固定的本地值*waiting-for*来指明这个进程等待的邻居集合。

a）为这些进程设计（比如，给出前提-后果代码）一个分布式的环检测算法。你的算法应该判断是否有两个或更多进程组成的环，环中每个进程都在等待下一个进程，而任何进程到环中它的前驱的路径上也没有消息。

b）证明你算法的正确性并分析它的复杂度。

c）参照19.2.3节中描述的问题，展示你的算法如何可以用来在一个底层异步算法A中检测死锁。

19.14 在死锁问题的另一个版本里，进程如19.2.3节所说那样等待一些邻居，不过此时等待的进程仅仅需要从这些邻居里任何一个收到消息即可，而不是从所有这些邻居都要收到消息。为这个版本的问题定义一个合适的死锁概念，并基于一致全局快照设计一个算法来检测这种新死锁类型。

639 19.15 描述用于监测发送/接收网络算法的一致全局快照的其他一些应用。

第20章 网络资源分配

在第16~19章中完成对异步网络编程通用方法的描述之后，现在继续研究异步网络中的特定问题。在本章中，再一次研究互斥问题以及更通用的资源分配问题，在第10和11章中已经研究过异步共享存储器环境中的这些问题。接着，在第21章中，考虑在某些进程可能失效情况下的异步网络中的一致性问题和其他一些问题。研究异步计算的最后一章是第22章，其中将研究在不可靠通道上的可靠通信问题。

20.1 互斥

我们从互斥问题开始。

20.1.1 问题

问题陈述和10.2节中的问题陈述差不多。如在10.2节中一样，用户$U_1, \ldots, U_n$被定义为保持良构性的I/O自动机。接着我们假设将要使用的系统A是一个异步网络系统，在A中每个用户U_i都有一个相对应的进程P_i。动作try_i、$crit_i$、$exit_i$和rem_i用于I/O自动机U_i与进程P_i的通信。在发送/接收网络环境下，进程P_i之间使用图20-1中描述的可靠的FIFO通道$C_{i,j}$通信。我们同样也将考虑广播系统以及由发送/接收和广播组合而成的系统。（这样一个组合可以看成是多播通道的一个特例——见14.3.2节。）

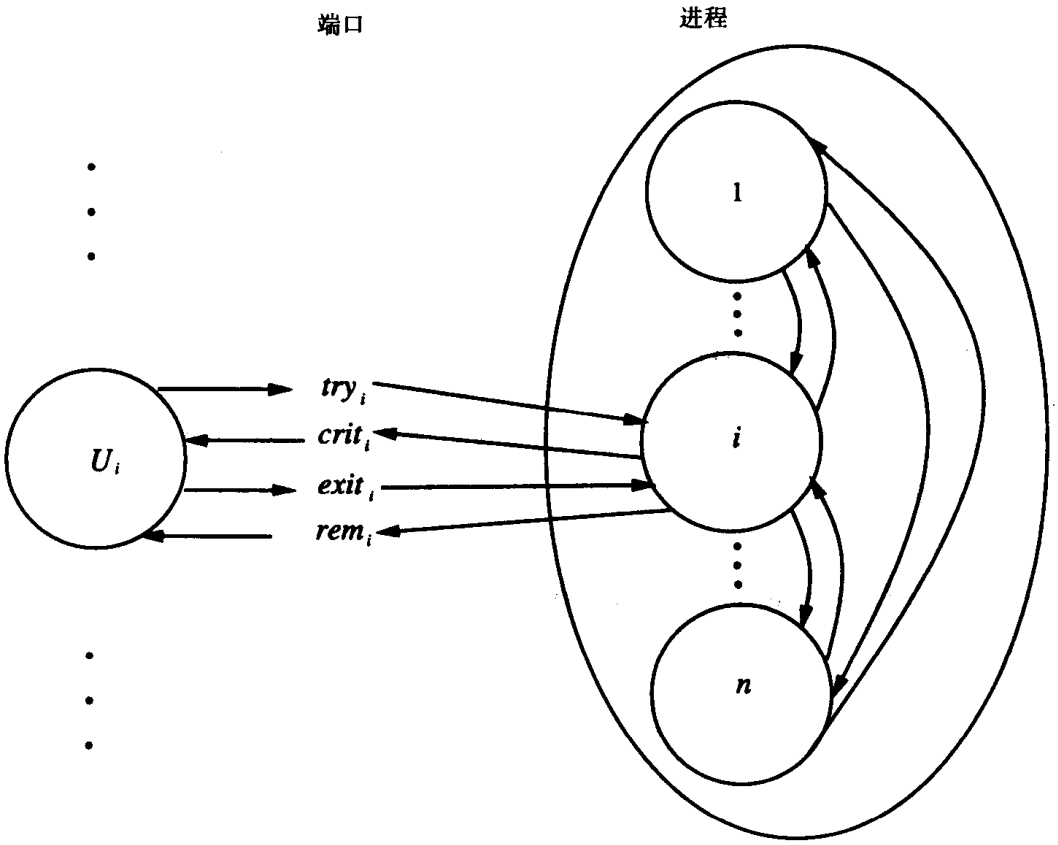


图20-1 针对互斥问题的组件之间的交互。系统椭圆内的箭头代表发送/接收通道

系统所能保证的基本正确性条件和我们在10.2节中定义的一样。也就是说，我们要求系统A和用户的组合必须满足下列属性：

良构性　在任意运行中，对任意i，描述U_i和A之间交互的序列对U_i来说必须是良构的。

互斥性　不存在其中一个以上的用户同时进入临界区C的系统状态（即一个A的全局状态以及所有U_i的状态的组合）。

演进性　在一个公平运行的任一时刻，

1）（尝试区的演进）如果有至少一个用户处于T，但是没有用户处于C，那么之后某一点中某个用户进入C。

642

2）（退出区的演进）如果至少有一个用户处于E，那么之后某一点中某个用户进入R。

如果对任意的用户组合，异步网络系统A都解决互斥问题，那么我们称A*解决了互斥问题*。

在这一章里，我们取消在10.2节中所做的限制，即仅当一个进程的用户处于尝试区或者退出区时它才可以执行局部控制动作。这个限制在共享存储器环境下是可行的，因为那里共享变量维护信息，使得这些信息对所有进程来说总是可用的。然而，在网络环境下，不存在共享变量，所以进程需要做维护这些信息的工作并且在需要的时候和其他进程交流这些信息。

我们也使用与10.4节一样的锁定权条件，即：

锁定权　在任意公平运行中，必须满足下列属性：

1）（尝试区的锁定权）如果所有用户总是释放资源，那么到达T的用户最终总会进入C。

2）（退出区的锁定权）到达E的用户最终总会进入R。

在这一章里，我们有时将会分析那些“以隔离方式”操作的请求的通信复杂度和时间复杂度。如果对一个用户发出的请求来说，在它从*try*到*crit*的这段时间里，其他用户都停留在剩余区中，那么我们就说这个请求是*隔离的*。

在这一节的剩余篇幅中，我们描述异步网络中的几种互斥算法。

20.1.2　模拟共享存储器

第10章包含很多共享存储器中的互斥算法。使用第17章中的技术，我们可以将这些算法变换成异步网络模型中的互斥算法。比如，可以有效地在异步发送/接收网络中实现10.7节中的*Bakery*算法。

20.1.3　循环令牌算法

643

异步发送/接收网络环境中的最简单互斥算法工作在单向环网络上。

***CirculatingToken*算法（非形式化）：**

一个代表对资源控制的令牌在环中不停流转。当进程P_i收到令牌时，它检查是否有来自用户U_i的显著请求。如果没有这样的请求，P_i就将令牌传给P_{i+1}。否则，P_i将资源授权给U_i，并持有令牌直到U_i释放资源。当U_i归还资源后，P_i将令牌传给P_{i+1}。

形式化代码如下。

***CirculatingToken*$_i$ automaton (formal):**

Signature:

Input:	Output:
try_i	$crit_i$
$exit_i$	rem_i
$receive(\text{“token”})_{i-1,i}$	$send(\text{“token”})_{i,i+1}$

States:

token-status $\in$ {*not-here*, *available*, *in-use*, *used*}, initially *available* if $i = 1$, *not-here* otherwise

region $\in \{R, T, C, E\}$, initially R

Transitions:

try_i
  Effect:
    $region := T$

$crit_i$
  Precondition:
    $region = T$
    token-status = *available*
  Effect:
    $region := C$
    token-status := *in-use*

$exit_i$
  Effect:
    $region := E$

rem_i
  Precondition:
    $region = E$
  Effect:
    $region := R$
    token-status := *used*

$receive(\text{“token”})_{i-1,i}$
  Effect:
    token-status := *available*

$send(\text{“token”})_{i,i+1}$
  Precondition:
    token-status = *used* or
      (*token-status* = *available* and $region = R$)
  Effect:
    token-status := *not-here*

Tasks:
Each locally controlled action comprises a task by itself.

644

定理20.1 *CirculatingToken*算法解决互斥问题并且保证锁定权。

证明概要 证明是直截了当的。因为只有一个令牌，而且只有令牌所在的用户才可以进入C，所以互斥是得到保证的。因为令牌一直流转直到发现一个请求为止，所以演进性是可以保证的。在令牌在环中流转一圈之前，同一进程的两个连续请求不能得到满足，所以每个进程都有机会获得令牌，可以保证锁定权。 □

复杂度分析 首先我们考虑*CirculationToken*算法的通信复杂度。我们并不清楚地知道应该测量什么，因为消息并不是很自然地分给特定请求的。比如，在没有活动请求时仍然有消息在发送。我们只能说在一个try_i和相应的$crit_i$之间发送的消息总数最多是n。我们还可以为“重负载”情况给出分阶段分析，“重负载”是指在每个节点上总是有一个活动请求。（形式化地说，即每个rem_i后面都紧接着一个try_i。）这种情况下，每个请求都只有恒定数量的消息。

对于时间复杂度，我们照常假设l是每个进程的任务时间的上界而d是任意通道中最早消息的延迟上界。我们还假设c是任意用户花在临界区中的时间的上界。那么从一个try_i事件到相应的$crit_i$事件所花的时间最多是$c(n-1)+dn+O(ln)$。注意，这个时间范围有一个dn项，这个项甚至在很轻的负载（比如一个隔离请求）的时候也可以出现。

虚拟环 如果进程被配置成一个虚拟环，*CirculatingToken*算法可以用于任意一个基于强连通有向图G的发送/接收网络中。在环上相邻的进程在G中不一定非要是相邻的——任意一对进程之间的通信可以由沿着底层网络上的一条有向路径的一系列通信来模拟，这一点可以由G的强连通性保障。而所得算法的性能则很大程度上取决于图G以及进程按照什么样的顺序被安排在环上——这对于在模拟中最小化总路径长度是非常重要的。

容错 在实际应用中，*CirculatingToken*算法可以从某些类型的故障中恢复过来。比如，如果一个进程发生故障，且其他进程可以探测得到这一信息，那么其他进程可以重新配置它们自己以组成一个新环。再比如，如果令牌也丢失，且其他进程可以探测到这一点，则在环上可以使用

645

一个领导者选举协议产生一个新令牌，而领导者选举协议可以是15.1节中的其中一个。

在异步模型中，不能检测普遍进程的停止故障和消息丢失，因为进程没有办法从进程或消息仅是被延迟的情形中区分出这种故障。因此，为了达到容错，有必要假定一个更强的模型，这个模型中包含可以宣告这种故障的事件。实际上，这些事件通常是由超时来实现的。

20.1.4 基于逻辑时间的算法

在18.3.3节中，为异步网络系统（特别是一个广播网络系统）中互斥问题给出另一种解法。这个方法使用*ReplicatedStateMachine*算法来实现一个代表请求进程索引队列的原子对象。而*ReplicatedStateMachine*算法又使用逻辑时间。这样，这个算法是分成几块来描述的。

这一节介绍一个类似的算法，不过为了更容易地和本章的其他算法比较，我们将算法的几块放到一起描述。为了简单起见，我们不以18.3.3中描述的那种特殊方式来处理本地操作。我们将得到的算法称为*LogicalTimeME*算法。

LogicalTimeME算法（非形式化）：

这个算法基于本地非负整数的*clock*值，使用*LamportTime*策略来产生事件的逻辑时间。逻辑时间是一个二元组(c,i)，其中$c \in \mathbf{N}$而i是一个进程索引；逻辑时间是按字典序排序的。

算法同时使用广播和发送/接收通信，所有相互不同的一对进程之间都可以使用发送/接收通信。每个进程P_i维护一个单独的*history*数据结构来代替独立的*invocation-buffer*和队列。对每个j，$history(j)_i$记录着P_i收到的来自P_j的所有消息，每个消息都有一个非负整数c，c是和这条消息的*bcast*事件或*send*事件相关的*clock*值。和以前一样，*try*和*exit*请求是广播的。但是每个进程通过一个*ack*消息确认每个*try*请求，而不再广播哑元消息。

在进程P_i的最近一个*try*请求到达它的$history(i)$、并且P_i收到的所有其他具有更小逻辑时间的请求都被已处理、以及P_i从所有其他进程都收到一个有更大逻辑时间的消息的情况下，P_i可
646 以执行一个$crit_i$。（后两个属性一起保证当前不会有具有更小逻辑时间的请求，而且以后也不会有。）只要P_i的最近的*exit*请求到达它的$history(i)$，P_i就立刻可以执行rem_i。

我们令 ≤ 代表逻辑时间二元组的字典顺序。

$LogicalTimeME_i$ automaton (formal):

Signature:

Input:
- try_i
- $exit_i$
- $receive(m)_{j,i}$, $m \in \{$"try", "exit", "ack"$\} \times \mathbf{N}$, $1 \le j \le n$,

Output:
- $crit_i$
- rem_i
- $send(m)_{i,j}$, $m \in \{$"ack"$\} \times \mathbf{N}$, $j \ne i$
- $bcast(m)_i$, $m \in \{$"try", "exit"$\} \times \mathbf{N}$

States:

$region \in \{R, T, C, E\}$, initially R
$clock \in \mathbf{N}$, initially 0
bcast-buffer, a FIFO queue of {"try", "exit"}, initially empty
for every $j, 1 \le j \le n$:
　history(j), a subset of {"try", "exit", "ack"} × N, initially ∅
for every $j \ne i$:
　send-buffer(j), a FIFO queue of {"ack"} × N, initially empty

Transitions:

try_i
　Effect:

$send(m, c)_{i,j}$
　Precondition:

$clock := clock + 1$
$region := T$
add *"try"* to *bcast-buffer*

$bcast(m, c)_i$
Precondition:
m is first on *bcast-buffer*
$c = clock + 1$
Effect:
$clock := c$
remove first element of *bcast-buffer*

$receive(m, c)_{j,i}$
Effect:
$clock := \max(clock, c) + 1$
$history(j) := history(j) \cup \{(m, c)\}$
if $m =$ *"try"* and $j \neq i$ then
add *"ack"* to *send-buffer*(j)

$exit_i$
Effect:
$clock := clock + 1$
$region := E$
add *"exit"* to *bcast-buffer*

m is first on *send-buffer*(j)
$c = clock + 1$
Effect:
$clock := c$
remove first element of *send-buffer*(j)

$crit_i$
Precondition:
$region = T$
$(\text{"try"}, c) \in history(i)$
$\nexists(\text{"exit"}, c') \in history(i)$ with $c' > c$
for all $j \neq i$
if $(\text{"try"}, c') \in history(j)$, $(c', j) < (c, i)$
then
$\exists(\text{"exit"}, c'') \in history(j)$ with $c'' > c'$
$\exists(m, c') \in history(j)$ with $(c, i) < (c', j)$
Effect:
$clock := clock + 1$
$region := C$

rem_i
Precondition:
$region = E$
$(\text{"exit"}, c) \in history(i)$
$\nexists(\text{"try"}, c') \in history(i)$ with $c' > c$
Effect:
$clock := clock + 1$
$region := R$

Tasks:
$\{crit_i\}$
$\{rem_i\}$
$\{bcast(m)_i : m \in \{\text{"try"}, \text{"exit"}\} \times \mathbb{N}\}$
for every $j \neq i$:
$\{send(m)_{i,j} : m \in \{\text{"ack"}\} \times \mathbb{N}\}$

647

定理20.2 *LogicalTimeMe*算法解决互斥问题并且保证锁定权。

证明 我们给出一个操作式证明。为了证明这个算法保证互斥性，我们使用反证法。假设在某个可达系统状态中，两个进程P_i和P_j同时处于C中。假设（为了保持一般性）P_i的最近*try*消息的逻辑时间t_i小于P_j的最近*try*消息的逻辑时间t_j。那么为了执行$crit_j$并进入C，P_j必须看到在它的$history(i)$中有一个来自P_i的消息，其逻辑时间大于t_j，从而也大于t_i。而从P_i到P_j的通信通道的FIFO属性意味着在P_j执行$crit_j$时它必须已经看到P_i的当前*try*消息。但是$crit_j$的前提条件规定P_j必须已经看到一条来自P_i的随后*exit*消息。这意味着当P_j执行$crit_j$时P_i必须已经离开C。矛盾。

接着，我们讨论锁定权，这意味着演进性。*try*区的锁定权可以由请求是根据它们的*try*消息的逻辑时间的顺序获得服务这一事实得出。我们强调一个具有当前请求中最小逻辑时间的*try*消息最终总会收到一个*crit*回应。那么既然对任意一个特定的*try*消息来说，只有有限多的*try*消息逻辑时间小于它，那么可以归纳得出每个请求都会得到授权响应。

现在假设P_i处于T并且有当前请求中逻辑时间最小的*try*消息。我们认为$crit_i$的前提条件最终总会得到满足并且保持满足直到$crit_i$发生。广播通道的公平特性意味着P_i最终总会收到自己的*try*消息并将

648

其置于自己的$history(i)_i$中。并且，因为*try*消息会收到相应的回应，且*clock*变量是根据*LamportTime*方式来管理的，那么P_i最终将从每个其他进程收到一个逻辑时间大于t_i的消息。最后，既然P_i的请求是当前逻辑时间最小的请求，那么任何逻辑时间更小的请求肯定已经有一个相应的*exit*事件。那么广播通道的公平性意味着最终P_i会收到这些*exit*消息。通过这种方式，$crit_i$的所有前提条件最后都得到满足。

退出区的锁定权可以直接得到。 □

复杂度分析　对于通信复杂度，我们注意到，和*CirculatingToken*算法不一样，在*LogicalTimeME*算法中，消息很自然地分派给特定请求。所以我们计算每个请求的消息数目。对每个请求，只有一个*try*广播和一个*exit*广播，共有$2n$条消息，加上$n-1$条回答*try*消息的*ack*消息。因此准确的说每个请求有$3n-1$条消息。

对于时间复杂度，我们首先考虑一个用户U_i发出的隔离请求的情况。实际上，我们考虑"强隔离"请求，这是指我们还要求当try_i事件发生时前面请求残留下来的消息不会保存在系统状态中。在这种情况中，从try_i到$crit_i$的时间至多只需要$2d+O(l)$，这里d是从任意进程i到任意其他进程j的最早消息（广播或者点对点）的传递时间上界。比较而言，*CirculatingToken*算法的时间复杂度有一个dn项，即使在是在一个隔离请求的情况下也是如此。

我们把从一个try_i事件到相应的$crit_i$事件的最坏情况下的时间上界作为一道习题留给读者。

20.1.5 *LogicalTimeME*算法的改进

现在我们描述被设计来降低通信复杂度的一个*LogicalTimeME*算法的简单变形。根据设计者把它命名为*RicartAgrawalaME*算法，它的每个请求只使用$2n-1$条消息。它通过以一种更仔细的方式来对请求进行确认以省略*exit*消息，从而改进*LogicalTimeME*算法。这个算法使用广播
649 和发送/接收通信，这里任意一对互不相同的进程之间都可以进行发送/接收通信。

***RicartAgrawalaME*算法：**

事件逻辑时间的产生和*LogicalTimeME*算法中的一样。惟一需要广播的消息是*try*，而惟一在发送/接收通道里发送的消息是*ok*。每条消息都携带它的*bcast*事件或者*send*事件的*clock*值。

在一个try_i输入后，P_i像在*LogicalTimeME*里一样广播*try*并且在收到从其他所有进程来的相应的*ok*消息后可以进入C。这个算法的有趣之处在于它规定一个进程P_i何时发送一条*ok*消息给另一个进程P_j的一条规则。基本思想是使用一个优先级方案。为了回答来自P_j的一条try_i消息，P_i做以下工作：

1）如果P_i处于E或者R，或者处于T且还没有广播当前请求的*try*消息，那么P_i回答以*ok*。

2）如果P_i处于C，那么推迟回答直至它到达E，此时它立即发送所有推迟的*ok*。

3）如果P_i处于T并且当前的请求已经广播出去，那么P_i把它自己请求的逻辑时间t_i（*bcast*事件的）和收到的P_j的*try*消息相关联的逻辑时间t_j进行比较。如果$t_i > t_j$，那么P_i自己的请求被给予一个较低的优先级，P_i回答一个*ok*消息。否则，P_i自己的请求拥有较高的优先级，它推迟回答直到它完成下一个临界区。这时，它立即发送任何推迟的*ok*。P_i可以在收到一个$exit_i$之后的任何时候执行一个rem_i。

也就是说，当存在冲突的时候，*RicartAgrawalaME*算法通过偏爱由逻辑时间决定的"更早"请求来解决冲突。

定理20.3　*RicartAgrawalaME*算法解决互斥问题并且保证锁定权。

证明　我们给出一个操作式证明。首先我们通过反证法来证明互斥性。假设在某个可达系统状态中，两个进程P_i和P_j在同时处于C。假设（为了保持一般性）P_i的最近*try*消息的逻辑时间t_i小于P_j的最近*try*消息的逻辑时间t_j。那么在它们进入C之前，P_i和P_j之间肯定互相发送*try*消息

和*ok*消息。另外，在每一个进程中，接收另一个进程发来的*try*消息是在它自己发送相应的*ok*消息之前的。但这仍然使不同的事件会有几种可能的不同次序，具体请参见图20-2。

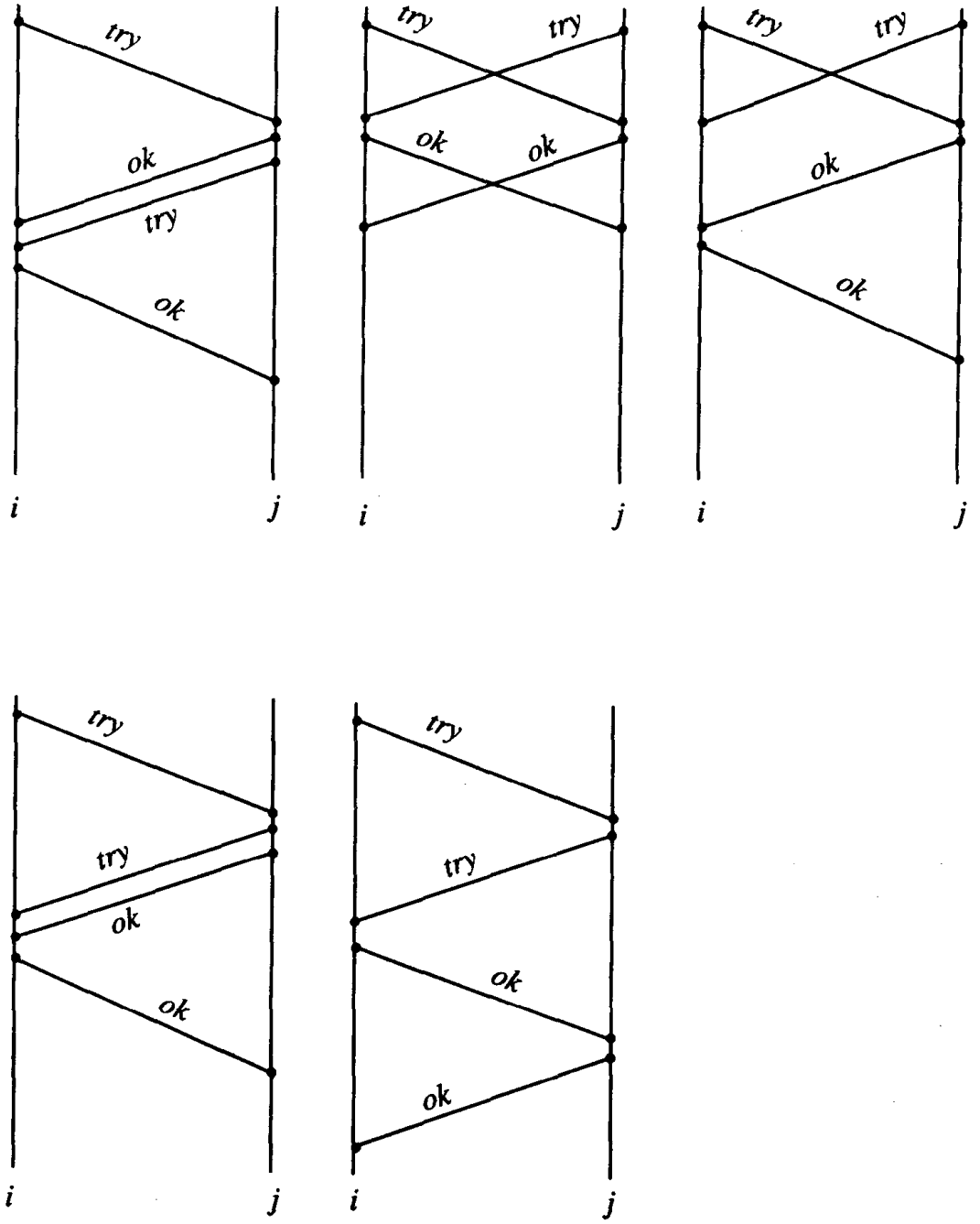


图20-2 *RicartAgrawalaME*算法中的某些可能的事件次序

现在我们断定P_j的最近*try*消息的*receive*事件发生在P_i广播它自己的最近*try*消息之后。否则，逻辑时间的属性意味着这个*receive*事件的逻辑时间大于t_j，且P_i产生的*bcast*事件的逻辑时间t_i大于这个*receive*事件的逻辑时间。这样一来有$t_i > t_j$，矛盾。 650 ~ 651

因此，在P_i收到P_j的*try*消息的时候，P_i要么处于*T*要么处于*C*。但是在每一种情况中，P_i的规则规定它应该推迟发送*ok*消息直到它完成自己的临界区。这样，在P_i离开*C*之前P_j是不能进入*C*的，矛盾。

现在我们证明演进性，同样也是使用反证法。退出区的演进是显然的。假设在公平运行α的某点中，某个用户处于T但没有用户处于C，而在该点之后没有用户曾经进入过C。那么（如同引理10.4的证明），在α的某个扩展α_1中，所有用户要么在R要么在T，且没有进一步的区域变化发生。那么在α_1的某个扩展α_2中，所有在T中的进程都被赋予一个根据它们最近请求得到的逻辑时间，并且没有消息曾经传送过。在α_2中，对于所有处于T的进程，令P_i是最近请求逻辑时间最小的那个进程，令这个最小的逻辑时间为t_i。

既然P_i一直陷于T中，那么肯定有其他某个进程从来没有对P_i的最后一个*try*消息应答以*ok*消息。只可能有两个原因使得P_j在收到P_i来的*try*消息后不立即发出*ok*消息：

1）P_j收到*try*消息时处于C。在这种情况下，因为在α_2中没有进程处于C中，所以P_j必须在α_2开始之前完成自己的临界区，并且在这之后发送推迟的*ok*消息给P_i。

2）P_j在T中收到一个请求的逻辑时间为$t_j < t_i$的*try*消息。这时，在α_2中，对于陷于T的所有进程，P_i的请求的逻辑时间是最小的，所以P_j肯定是在收到从P_i来的*try*消息之后，在α_2开始之前，进入并完成它的临界区的。但是又一次地，这意味着P_j必须发送推迟的*ok*消息给P_i。

在每个情况中，P_i都收到所有所需的*ok*消息并演进到C，矛盾。

我们把对锁定权属性的证明留做一道习题。 □

复杂度分析　很容易看出每个请求都有$2n-1$条消息被发送。时间复杂度的分析留做一道习题。

另一种优化　通过给*ok*消息以不同的解释还可能进一步改进*RicartAgrawalaME*算法。当某个进
652 程P_i发送*ok*消息给另外一个进程P_j时，它不仅同意P_j当前的请求，而且允许P_j重新进入C任意多次——直到P_j发送*ok*给P_i来回答来自P_i的*try*消息。回答*try*消息的规则和*RicartAgrawalaME*算法中的一样。

在一个用户在没有其他用户的干扰请求发生时重复请求资源的情况中，这个算法版本运行得最好。此时，发出请求的用户可以重复地进入它的临界区，不需要在第一个请求发送出后再发送任何其他消息。

20.2 通用资源分配

现在我们考虑异步网络中的更为通用的资源分配问题。

20.2.1 问题

如同11.1节中一样，这里的问题定义采用显式资源规格说明和排他规格说明。我们假设用户自动机和20.1节中的一样。

系统保证的基本正确性条件和11.1节中的一样。即，对于一个给定的排他规格说明$\mathcal{E}$，我们要求系统和用户的组合满足以下属性：

良构性　在任意运行中，对任意i，描述U_i和A之间交互的序列对U_i来说必须是良构的。

排他性　不存在一个可达的系统状态，其中由处于临界区的用户组成的集合是$\mathcal{E}$的一个子集。

演进性　在一个公平运行的任一点中，

1）（尝试区的演进）如果至少有一个用户处于T而没有用户处于C，那么在之后某点中某个用户进入C。

2）（退出区的演进）如果至少一个用户处于E，那么在之后某点中某个用户进入R。

当一个异步网络系统解决任意用户集合的通用资源分配问题时，我们就称它解决了通用资源分配问题。对于显式资源规格说明，我们还考虑：

独立演进性 在一个公平运行的任一点中，

653

1）（尝试区的独立演进）如果U_i处于T并且所有冲突用户都处于R，那么在之后某点中U_i进入C或者某个冲突用户进入T。

2）（退出区的独立演进）如果U_i处于E且所有冲突用户都处于R，那么在之后某点中U_i进入R或者某个冲突用户进入T。

我们还要考虑锁定权条件。如同实现互斥所做的那样，我们取消只有当一个进程的用户处于尝试区或者退出区时它才能够执行局部控制动作的限制。

对于一个给定的资源规格说明$\mathcal{R}$，只有满足从*try*到*crit*的这段时间所有有冲突请求的其他用户都处于它们的剩余区的条件下，我们才说一个用户请求是隔离的。

哲学家饮水 在20.2.5节中我们将要考虑的一个通用资源分配问题的变种涉及同一个用户U_i在不同的时间请求不同的资源。这个版本的问题基于一个给定的资源规格说明$\mathcal{R}$，而且我们假设对每个i，try_i动作是由R_i的一个任意子集参数化的，R_i是指定给用户U_i的资源集合。排他条件被重新解释为指那个实际上最近被请求的资源，而不是由R描述的潜在的资源。这就是说，我们要求不存在这样的可达系统状态——两个最近请求集合相交的用户同时处于它们的临界区。演进条件和锁定权条件都和前面一样。独立演进条件和隔离请求的定义都重新解释为指那些实际请求。

这个版本的资源分配问题有时被称为哲学家饮水问题，资源有时被称为瓶子。

20.2.2 着色算法

11.3.3节的着色算法可以修改来解决在一个基于连通无向图G的异步发送/接收网络和一个给定的资源规格说明$\mathcal{R}$的情况中的通用资源分配问题。做到这一点的一种方法是使用第17章中描述的共享存储器算法模拟中的一个。不过，一个目标特定的模拟可以更有效地工作。

着色算法：

除了模拟共享存储器着色算法进程的进程之外，我们再包括一个管理每个资源的进程。这个网络算法的每个进程正确地模拟一个共享存储器算法的进程及资源进程的某个子集。当用户U_i执行try_i时，进程P_i按照以前的颜色升序、每次一个地收集所需资源，但这次依靠发送消息给相应资源进程来进行。每发送完一个消息，P_i都等待接收一个回答。每个资源进程维护一个FIFO的请求用户队列，将新收到的请求加到队列的尾端。当索引i到达一个资源进程的队列的头部时，这个资源进程就向P_i发还一条消息，然后P_i就转向请求下一项资源。当P_i获得所有所需资源时，它就可以执行$crit_i$。当$exit_i$发生时，P_i向所有有关的资源进程发送消息通知它们从各自的队列里移去索引i。在发送完所有这些消息之后，P_i就可以执行rem_i，而不需要等待任何回答。

654

这个算法要求每个进程P_i能够和所有进程P_j通信，其中P_j负责管理那些根据给定资源规格说明$\mathcal{R}$赋予i的资源。通常，如果相关节点在底层的图G里都是直接相连的话，这个通信可以直接进行，否则，则可以通过G中的边组成一条路径来模拟。

对这个版本的着色算法的分析和11.3.3节的共享存储器中的分析差不多。这一次，时间范围取决于进程步时间的上限、消息传递时间、临界区时间、加上用于给资源图着色的颜色数量、以及请求同一个单独资源的最大的用户数量。不过，和前面一样，时间范围不是直接取决于底层图G的大小的。

20.2.3 基于逻辑时间的算法

*RicartAgrawalaME*算法可以推广来解决对任意一个资源规格说明R的资源分配问题。现在

假设我们使用多播和发送/接收通信的组合。（技术上说，这可以看成是多播通信的一个特例——见14.3.2节。）必须允许任意进程可以向所有与之共享资源的进程集合进行多播，而任意两个共享资源的进程之间可以进行发送/接收通信。

***RicartAgrawalaRA*算法：**

进程使用*LamportTime*算法来计算逻辑时间。

在收到一个try_i输入后，进程i向所有与之共享资源的进程多播一个带有相关联的*clock*值的
655 *try*消息。当收到从所有这些进程发来的应答*ok*消息后，进程i可以进入C。进程使用和*RicartAgrawalaME*算法一样的规则来发送*ok*消息，根据逻辑时间来决定优先级。

在进程i收到一个$exit_i$后的任何时候它都可以执行一个rem_i。

定理20.4　对任意给定的资源规格说明，*RicartAgrawalaRA*算法都解决通用资源分配问题并保证锁定权和独立演进性。

像对*RicartAgrawalaME*算法一样，我们可以修改*RicartAgrawalaRA*算法以使得*ok*消息可以延长扩展许可，直到被明显撤销为止。

20.2.4　无环有向图算法

在*RicartAgrawalaRA*算法里，逻辑时间被用来给冲突的请求赋予优先级，从而打破僵局。还有其他的策略可以用来打破僵局，比如，维护一个包含所有进程的无环有向图。

为了简单起见，我们考虑一个满足以下两个限制的显式资源规格说明$\mathcal{R}$：

1）每个资源都在正好两个用户的资源集合里。

2）每对用户至多共享一个资源。

我们将取消这些限制的扩展留做一道习题。

我们假设有一个基于无向连通图G的发送/接收网络，任何两个共享一个资源的进程在G里都有一条边直接相连。同样，为了简单起见，我们假设在G中的所有边都在共享资源的进程之间。

***AcyclicDigraphRA*算法：**

这个算法是基于以这样一种方式维护G中所有边的方向，即，在任何时刻，包含所有有向边的有向图H是没有环的。每条边的方向是存储在两个端点进程的本地*orientation*变量中的，通过从有向边的头部进程向尾部进程发送*change*消息的方式来改变边的方向。我们必须假设由初始边的方向决定的有向图是无环的。

如果进程i处于尝试区并且所有它连着的边的方向都是向内的，那么它可以执行一个$crit_i$输
656 出。如果进程i处于退出区，那么它可以执行rem_i，将它所有的*orientation*变量都设置为指向外面，并向所有向内的边发送一个*change*消息，这些都是一步完成的。（*change*消息被同时放入指向所有邻居的本地*send-buffer*中。）还有，如果进程i处于剩余区并且它的所有边都是向内的，那么进程i将它的所有*orientation*变量设置为指向外面，并向它的每条向内边发送一条*change*消息，这也是一步完成的。

定理20.5　*AcyclicDigraphRA*算法解决资源分配问题（在上面的限制1和2下）并保证锁定权。

证明概要　我们以给出在任何一个可达状态中对每条边方向的一个更详细定义开始。即，对一条边(i,j)，当P_i的关于此边的*orientation*变量为“向外”并且P_j的*orientation*变量为“向内”，或者有一条*change*消息在从P_i发向P_j（在$send\text{-}buffer(j)_i$或者在从i到j的通道里）时，我们说边(i,j)是从i指向j的。一个不变式可以用来说明这个规则在每个可达状态中为每条边决定惟一的方向。

那么我们证明这个不变式，即，当一个进程P_i处于临界区时，所有和它关联的边都是向内的，并且没有*change*消息在这些边的任一个方向上传送。这意味着排他属性。

下面我们证明关键的不变式，即有向图H是无环的。我们已经假设在开始时这是真的。惟一可以使这个断言不成立的步是改变某些边方向的那些步。但是每个改变边的方向的步同时也以这样一种方式改变一些特定节点上的相关联的所有边，即在这一步之后所有边都是指向外的。因为这步之后没有边是向内指向i，那么这一产生新方向的边的步之后肯定也是没有环的。从而不会产生环。

接着我们证明锁定权，这意味着演进性。我们只考虑尝试区的锁定权；通常，退出区的条件是显然的。因为图是无环的，在一个运行的任一点中，我们可以定义图的一个节点i的高度为在有向图H中从i开始的一条有向路径的最大长度。我们首先注意到一个节点的高度永远不会增加，除非它的高度是0（给此节点上的进程一个机会进入临界区）。我们从而可以得到任何高度为0的节点最终都会将所有指向它的边变为方向向外。使用这些论据，我们证明任何高度$h>0$的节点最终会拥有一个小一些的高度h'，这意味着任何高度$h>0$的节点最终高度会变为0。这样给予这个节点上的进程一个机会可以进入临界区。 □ 657

相同进程 *AcyclicDigraphRA*算法的一个有趣的性质是进程“几乎”都是一样的：除了所有边的初始方向之外它们不使用UID或者其他可以相互区别的信息。为了在任意图中解决这个问题，像定理11.2那样的论断意味着需要打破对称性的一些方法。这里，对称性是通过有向图H初始是无环的这样一个条件来打破的。

20.2.5 哲学家饮水*

在一个基于连通无向图G的具有可靠FIFO通道的发送/接收网络中，对于一个给定的资源规格说明$\mathcal{R}$，我们给出一个特别的哲学家饮水问题的解法。这个解法是模块化的——它使用任意一个对$\mathcal{R}$的解决通用资源分配问题的锁定权算法。这个方法我们称为*ModularDP*，它的体系结构如图20-3所示。每个U_i和相应的D_i之间的通信使用$try(B)_i$、$crit_i$、$exit_i$和rem_i动作。这里，$B \subseteq B_i$，B_i是由$\mathcal{R}$规定的i的瓶子（资源）的集合。 658

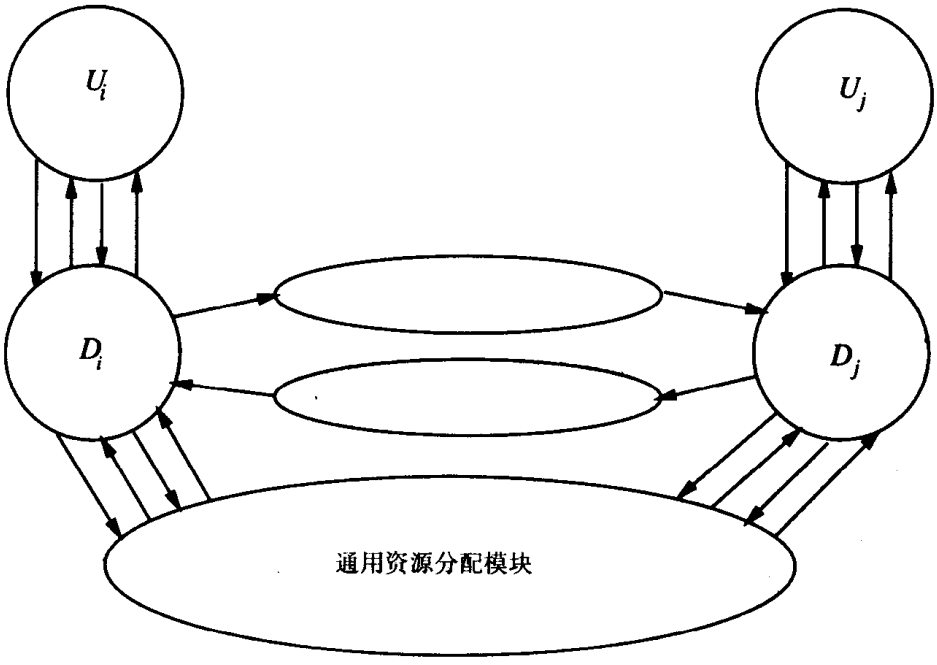


图20-3 *ModularDP*的体系结构

每个D_i和通用资源分配算法之间的通信使用$internal\text{-}try_i$、$internal\text{-}crit_i$、$internal\text{-}exit_i$和$internal\text{-}rem_i$动作；为了避免混淆，我们重命名这些动作。

在给定模型下对哲学家饮水问题的一个完全解法必须包括一个图20-3中的综合资源分配模块，这个模块是由基于同样的底层图G的一个发送/接收网络算法A实现的。完全解法中的每个进程P_i是D_i和一个相应A的进程的组合体。而这里的每个通道$C_{i,j}$必须同时实现图20-3中的从D_i到D_j的通道以及A中相应的通道。

为了简单起见，我们再次作出在$AcyclicDigraphRA$中的关于R的一个假设：每个瓶子都属于两个用户的资源集合。我们还假设任何两个共享一个瓶子的进程在G中都有一条边相连。

ModularDP算法：

当D_i收到$try(B)_i$时，它发送请求消息来请求那些需要的但是当前没有的瓶子。一个请求的接收者D_j在U_j处于E或R时满足这个请求。如果U_j处于T或者C，那么D_j推迟请求，这样当U_j完成它的临界区后D_j可以满足这个请求。

为了防止两个进程互相推迟对方的请求从而阻塞演进，一个通用资源分配模块被用来建立进程间的优先级。这样，当一个进程D_i一进入它的尝试区能够发送请求时，它就调用$internal\text{-}try_i$来试图获取优先权。当D_i收到一个$internal\text{-}crit_i$输入而它仍处于尝试区——即，当它进入自己的内部临界区时——它发送*demand*消息请求所需要的但现在还没有的瓶子。一条*demand*消息的接收者D_j在拥有这个瓶子的情况下总是满足这个请求，除非U_j正好处于使用这个瓶子的临界区；在这种情况下，D_j推迟这个*demand*并在U_j完成自己的临界区后满足它。

一旦D_i处于自己的内部临界区，我们可以证明它将最终收到所有它所需的瓶子。当D_i处于尝试区并拥有所有的瓶子，它可以进入自己的临界区。一旦D_i进入自己的临界区，它可以输出
659 $internal\text{-}exit_i$，因为它不再需要和内部临界区相关联的优先级。

D_i automaton:

Signature:

Input:
- $try(B)_i$, $B \subseteq B_i$
- $exit_i$
- $internal\text{-}crit_i$
- $internal\text{-}rem_i$
- $receive(m)_{j,i}$, $m \in \{\text{"request"}, \text{"bottle"}, \text{"demand"}\} \times (B_i \cap B_j)$, $j \in nbrs$

Output:
- $crit_i$
- rem_i
- $internal\text{-}try_i$
- $internal\text{-}exit_i$
- $send(m)_{i,j}$, $m \in \{\text{"request"}, \text{"bottle"}, \text{"demand"}\} \times (B_i \cap B_j)$, $j \in nbrs$

States:

$region \in \{R, T, C, E\}$, initially R
$internal\text{-}region \in \{R, T, C, E\}$, initially R
$need \subseteq B_i$, initially $\emptyset$
$bottles \subseteq B_i$; initially arbitrary, subject to the global restriction that the *bottle* sets for all processes partition the set of all bottles of $\mathcal{R}$.
$deferred \subseteq B_i$, initially $\emptyset$
current, a Boolean, initially *false*
for every $j \in nbrs$:

send-buffer(j), a FIFO queue of messages in $\{$ *"request"*, *"bottle"*, *"demand"*$\} \times (B_i \cap B_j)$, initially empty

Transitions:

try$(B)_i$
Effect:
region $:= T$
need $:= B$
for all $j \in$ *nbrs*,
all $b \in (need \cap B_j) - bottles$, do
add (*"request"*, b) to *send-buffer*(j)

send$(m)_{i,j}$
Precondition:
m is first on *send-buffer*(j)
Effect:
remove first element of *send-buffer*(j)

internal-try$_i$
Precondition:
region $= T$
internal-region $= R$
Effect:
internal-region $:= T$

receive(*"demand"*, b)$_{j,i}$
Effect:
if $b \in$ *bottles* and (*region* $\neq C$ or $b \notin$ *need*) then
add (*"bottle"*, b) to *send-buffer*(j)
bottles $:=$ *bottles* $- \{b\}$
deferred $:=$ *deferred* $- \{b\}$

receive(*"bottle"*, b)$_{j,i}$
Effect:
bottles $:=$ *bottles* $\cup \{b\}$

crit$_i$
Precondition:
region $= T$
need $\subseteq$ *bottles*
Effect:
region $:= C$
current $:=$ *false*

receive(*"request"*, b)$_{j,i}$
Effect:
if *region* $\in \{T, C\}$ and $b \in$ *need* then
deferred $:=$ *deferred* $\cup \{b\}$
else
add (*"bottle"*, b) to *send-buffer*(j)
bottles $:=$ *bottles* $- \{b\}$

internal-crit$_i$
Effect:
internal-region $:= C$
if *region* $= T$ then
current $:=$ *true*
for all $j \in$ *nbrs*,
all $b \in (need \cap B_j) - bottles$, do
add (*"demand"*, b) to *send-buffer*(j)

660

internal-exit$_i$
Precondition:
internal-region $= C$
current $=$ *false*
Effect:
internal-region $= E$

internal-rem$_i$
Effect:
internal-region $:= R$

exit$_i$
Effect:
region $:= E$
for all $j \in$ *nbrs*,
all $b \in$ *deferred* $\cap B_j$ do
add (*"bottle"*, b) to *send-buffer*(j)
bottles $:=$ *bottles* $-$ *deferred*
deferred $:= \emptyset$

rem$_i$
Precondition:
region $= E$
Effect:
region $:= R$

Tasks:
$\{crit_i\}$
$\{exit_i\}$

$\{internal\text{-}try_i\}$
$\{internal\text{-}exit_i\}$
for every $j \in nbrs$:
　$\{send(m)_{i,j} : m \in \{\text{“request”, “bottle”, “demand”}\} \times (B_i \cap B_j)\}$

对这段代码需要作出两点解释。第一，我们可以证明当D_i收到一个(“*request*”,b)消息时，它确实是拥有瓶子b。这样，在满足或者推迟这个请求之前D_i就不需要去确认$b \in bottles$。另一方面，这也使得当D_i没有瓶子b时仍然可能会收到一个（“*demand*”,b）消息。所以在满足这个*demand*之前，D_i要确认$b \in bottles$。

第二，标志$current_i$监视当前是否有一个内部临界区仍然被使用来给当前U_i发来的请求赋
661 予优先权。当一个$internal\text{-}crit_i$发生而$region_i$=*T*时$current_i$标志被设为*true*。当$crit_i$发生时它被设为*false*。当$current_i$=*false*时，D_i可以执行$internal\text{-}exit_i$来结束内部临界区。

定理20.6　通过使用任意锁定权的通用资源分配问题的解法，*ModularDP*算法解决哲学家饮水问题并保证锁定权。

证明概要　良构性很容易得到。排他条件源自以下事实：瓶子集合和瓶子消息明确代表瓶子，而且为了执行一个*crit*输出，进程必须拥有所有所需瓶子。我们证明锁定权，这意味着演进。为此，我们使用通用资源分配模块的特性来证明。

首先，从这段代码可以很容易看出资源分配模块的环境保持了模块的良构性。其次，模块的特性意味着系统的每一次运行都满足模块的良构性和排他条件。还有，每次公平运行也满足模块的锁定权条件。　□

声明20.7　在*ModularDP*系统的任何公平运行中，如果每个*crit*后面都紧跟着有一个相应的*exit*，那么每个*internal-crit*后面也都紧跟着一个相应的*internal-exit*。

证明概要　假设一个$internal\text{-}crit_i$在一次公平运行α的某个时刻发生而此后没有发生$internal\text{-}exit_i$；令α_1是$internal\text{-}crit_i$之后立刻开始的α的一个扩展。那么$current_i$在整个α_1中肯定始终保持为*true*。因为如果$current_i$为*false*的话，那么$internal\text{-}exit_i$的前提条件就为*true*，$internal\text{-}exit_i$最终必然会发生。还有，根据模块的排他条件，在α_1过程中没有D_i的邻居可以进入自己的内部临界区。

当$internal\text{-}crit_i$事件发生时，必然有$region_i$=*T*，因为$current_i$为*true*。因此，作为$internal\text{-}crit_i$事件的一个部分，D_i发送*demand*消息请求所有需要的瓶子。考虑这样一个（“*demand*”,b）消息的任何接收者D_j。如果D_j拥有瓶子b并且没有真正在使用它（比如，不是在$b \in need_j$时进入自己的临界区），那么它发送（“*bottle*”,b）给D_i。另一方面，如果D_j正在使用b，那么，既然每个$crit_j$都紧跟着有一个$exit_j$，D_j最终会完成临界区并满足这个推迟的*demand*。这样，D_i最终会得到所有所需的瓶子。我们断言它必须占有这些瓶子直到执行$crit_i$为止。这是因为在α_1过程中它不再接收请求这些瓶子的*demand*消息；这可以由在α_1过程中没有D_i的邻居进入自己的
662 内部临界区证明。（这里需要某些不变式。）

因为D_i获得所有需要的瓶子，它最终会执行$crit_i$。但是这个事件导致*current*被设为*false*，这是一个矛盾。　□

声明20.7可以使我们能够证明下面这个关键的断言。

声明20.8　在*ModularDP*系统的任何公平运行中，如果每个*crit*都紧跟着有一个相应的*exit*，那

么每个*try*都紧跟着有一个相应的*crit*（比如，每个请求都获得授权）。

证明概要 假设try_i在一次公平运行α的某个时刻发生而此后没有发生$crit_i$；令α_1是从try_i开始的α的一个后缀。

如果在α_1中发生一个*internal-crit*，那么声明20.7暗示有一个紧接着的*internal-exit*。但是由于有对*current*标志的处理，上述情况可能发生的惟一途径是在间隙里有一个$crit_i$发生。这是矛盾的。所以我们可以假定在α_1中没有*internal-crit*发生。

如果在α_1中*internal-region*曾经等于T，那么模块的锁定权属性意味着一个*internal-crit*最终肯定会发生，矛盾。所以我们可以假设在α_1中自始至终有$internal\text{-}region \neq T$。如果在$\alpha_1$中*internal-region*曾经等于$R$，那么最终会有一个$internal\text{-}try_i$发生，导致$internal\text{-}region=T$，又构成一个矛盾。所以我们可以假设在$\alpha_1$中自始至终有$internal\text{-}region \neq R$。使用模块的锁定权属性，我们可以得到在$\alpha_1$中自始至终有$internal\text{-}region \neq E$。

那么在α_1中自始至终惟一的可能性是$internal\text{-}region=C$。但是因为α_1是在一个try_i事件之后立刻发生的，在整个α_1中必然有$current=false$。但是这样最终会有一个$internal\text{-}exit_i$发生，导致$internal\text{-}region=E$，矛盾。 □

声明20.8产生*ModularDP*系统的尝试区的锁定权；退出区的锁定权是很容易得出的。

复杂度分析 *ModularDP*的复杂性范围取决于实现通用资源分配模块的代价。如果底层图G的任何一个节点的最大度数是k的话，算法的D_i组件的每个请求最多发送$3k$个消息。

对于时间复杂度，像前面一样定义l和d，并且令c为任何U_i的任何临界区的长度上界。假设T_1和T_2是单个进程在它的内部尝试区和内部退出区所分别花费的时间的上界。（T_1一般来说是关于一个内部临界区长度的上界的函数。内部临界区的长度的上界为$c+3d+O(l)$。）那么从一个*try*到相应的*exit*的时间界限为$T_1+T_2+c+3d+O(l)$。 663

对一个“强隔离的”请求，也就是说，是一个其中所有之前请求的残留消息都已发送出去的隔离请求，时间复杂度至多为$2d+O(l)$。

20.3 参考文献注释

*CirculatingToken*算法来自Le Lann[19]。他的论文包括一个关于互斥算法的容错的不同形式的讨论，其中包括使用领导者选举算法来重新生成一个丢失的令牌。*LogicalTimeME*算法来自Lamport[176]，*RicartAgrawalaME*算法来自Ricart和Agrawala[252]。20.1 节结尾的优化来自Carvalho和Roucairol[64]。Raynal的书[250]包含大量针对异步网络和异步共享存储器模型的互斥算法。

哲学家饮水问题是由Chandy和Misra[67]定义的。他们的论文还包括一个和*AcyclicDigraphRA*算法非常相似的通用资源分配算法，以及一个通过修改他们的通用资源分配算法构成的哲学家饮水问题的解法。Welch和Lynch[285]开发了以这里这种形式出现的*ModularDP*算法，这个算法是基于Chandy和Misra的思想。特别地，他们将隐含在Chandy-Misra算法中的模块化明确表示出来。

其他最新的网络资源分配问题的工作包括Styer和Peterson[272]，Choy和Singh[80]，以及Awerbuch和Saks[37]的工作。这些论文主要研究如何对运行时间和/或容错进行改进。

20.4 习题

20.1 给出异步发送/接收网络环境下*Bakery*互斥算法的一个实现的前提-结果代码。分析你的算

法的复杂度。（注意：你的算法不一定非要是，但是可以是，通过对原始的*Bakery*算法施加一个推广的变形来实现的。）

20.2 给出异步发送/接收网络环境下*PertersonNP*互斥算法的一个实现的前提-结果代码。分析你的算法的复杂度。

664 20.3 给出定理20.1的详细证明。

20.4 假设G是一个无向连通图。设计一个基于G的有效发送/接收网络算法，使得网络中所有进程配置它们自己以形成一个虚拟的环。特别地，假设这些进程有UID。每个进程必须输出自己在环上的后继UID，以及一条通向这个后继的路径上的所有节点的UID。尝试最小化所有路径的总长度。

20.5 将G换成有向强连通图，重做习题20.4。

20.6 对*LogicalTimeME*算法的互斥属性给出一个不变式断言证明。（提示：关键不变式表示如果一个进程i处于C，那么和它的*try*消息关联的逻辑时间小于任何其他没有后继*exit*消息的*try*消息的逻辑时间。）

20.7 对*LogicalTimeME*算法，证明一般最坏情况下一个try_i事件和相应的$crit_i$事件之间时间的界限。记住不要忽略可能的通道堆积。

20.8 “优化”*LogicalTimeME*算法以使*history*变量不再保存所有曾经收到的消息。也就是说，在压缩保存的消息的同时，使每个进程保持和以前一样的行为。使用一个涉及*LogicalTimeME*的模拟关系来证明你优化后算法的正确性。

20.9 假设我们修改*LogicalTimeME*算法，使得每个进程在收到一条消息后增加它的本地时钟，但是这个增加并不保证新的时钟大于收到的消息所携带的时钟。（这就得到习题18.4中的“非逻辑时钟”。）这个修改后的算法还保持有哪些正确性？证明你的声明（肯定的和否定的）。

20.10 给出*RicartAgrawalaME*算法的前提-结果代码并以之作为一个形式化正确性证明的基础。在你的对互斥的证明中使用不变式断言。

20.11 证明*RicartAgrawalaME*算法是具有锁定权的并证明从try_i事件到相应的$crit_i$事件之间的时间的上限。

20.12 给出改进版本的*RicartAgrawalaME*算法的前提-结果代码，在这个版本的算法里，*ok*消息
665 携带可以重复访问临界区的许可。证明你的代码的正确性。

20.13 分析20.2.2节中的改进的着色算法的通信复杂度和时间复杂度。

20.14 对于*RicartAgrawalaME*算法做以下工作：

a）写出前提-结果代码。

b）证明算法的正确性。

c）分析算法的复杂度。

d）构建一个运行使得从一个try_i事件到相应的$crit_i$事件的时间尽可能长。

20.15 给出改进版本的*RicartAgrawalaRA*算法的前提-结果代码，在这个版本的算法里，*ok*消息携带可以重复访问临界区的许可。证明你的代码的正确性。

20.16 对于*AcyclicDigraphRA*算法做以下工作：

a）写出前提-结果代码。

b）给出正确性的详细证明。

c）判断算法是否保证独立演进。

d）分析算法的复杂度。

e）构建一个运行使得使得从一个*try*事件到相应的*crit*事件的时间尽可能长。

f）证明一个隔离请求的时间上限。

20.17 解释为什么*CirculatingToken*算法可以看作是*AcyclicDigraphRA*算法的一个特例。

20.18 消除资源规格说明的两个限制以推广*AcyclicDigraphRA*算法。

20.19 给出基于连通无向图G的发送/接收网络的一个有效算法，对所有边进行定向以形成一个无环有向图H。你必须假设进程是有UID的。

20.20 陈述异步网络环境下的定理11.2并给出证明。

20.21 定义一个和11.3.1节中类似的等待链概念，不过这次针对*RicartAgrawalaRA*和*AcyclicDigraphRA*之类的算法，其中每个进程并不明确要求独立的资源。使用你的定义来分析这一章中的资源分配算法的等待链的长度。 666

20.22 Flaky计算机公司的程序员们决定要尝试去改进*AcyclicDigraphRA*算法。即，对于一个处于剩余区的进程，除非收到从某个邻居发来的一个明确*try*消息，它的所有向内的边都不会将边的方向改为向外。在从用户U_i收到一个try_i输入后进程P_i向所有邻居发送*try*消息。这个策略有什么错误？

20.23 研究问题：基于给定的资源规格说明R，为通用资源分配问题设计一个发送/接收网络算法。假设任何两个共享资源的进程在底层图中都有一条边相连。你的算法设计应该使某类请求的时间复杂度比较低，其中这类请求具有较小数量k的“重叠”冲突请求。尝试令这个时间界限线性于k。

20.24 研究问题：基于给定的资源规格说明R，为通用资源分配问题设计一个发送/接收网络算法。假设任何两个共享资源的进程在底层图中都有一条边相连。即使是在G中与i的距离大于等于k的进程发生停止故障的情况下，你的算法设计也应该能够保证任意特定进程i的锁定权。尝试最小化k。

20.25 给出定理20.6的证明的所有细节。特别地，你要证明一些不变式断言，包括

断言20.4.1 如果$b \in bottles_i$且一条(“*demand*”,b)消息正由D_j传向D_i，那么$region_j=T$, $internal\text{-}region_j=C$，并且$current_j=true$。

20.26 证明*ModularDP*的时间复杂度是$T_1+T_2+c+3d+O(l)$。

20.27 考虑使用着色算法（网络版）来实现通用资源分配模块的*ModularDP*哲学家饮水算法。陈述并证明用户等待请求得到满足为止的时间的上限。

20.28 消除对资源规格说明的限制来推广*ModularDP*算法。 667

第21章 带进程故障的异步网络计算

本章考虑在有进程停止故障的异步网络中，能够计算什么、不能够计算什么。在此，假设通信是可靠的，而只考虑进程的故障。

首先，我们证明，最终获得的计算性能结果与我们所考虑的系统是发送/接收系统还是广播系统没有关系。

接着，将针对异步网络模型中的分布式一致性问题，(重新)阐述基本的不可能性结果。这一结论表明，在异步网络中，只要存在进程故障，即使能够保证最多只有一个进程故障，也无法解决一致性问题。在第12章，曾经讨论这个问题，并针对异步共享存储环境给出一个类似的不可能性结果。正如我们在第12章的开始所述，对于那些需要一致性的分布式应用，这种不可能性结果有现实的意义。这些应用包括需要关于事务是否提交或放弃一致性的数据库系统、需要关于消息传递一致性的通信系统、以及需要关于故障诊断一致性的进程控制系统。这一不可能性结果表明，任何一个纯粹的异步算法都是不能够正确工作的。

669 在本章的余下部分，将描述一些绕过这一根本难题的方法：随机选择法、用故障检测机制强化模型法，允许用一组值的一致性而不是一个值，允许使用近似值而不是精确值。

本章对前述章节的依赖性很强，特别是第7、12和17章。具体来说，异步网络中关于计算性能的许多结果，都是通过对异步读/写共享存储系统中关于计算性能的类似结果进行普通变换直接得来的。

21.1 网络模型

我们假定，本章中的模型是一个异步广播系统，该系统采用可靠的广播通道，但会出现进程停止故障(用停止事件表示)。同样，也可以在所有的不同进程对之间考虑采用可靠FIFO 发送/接收通道的发送/接收系统：事实表明，从计算性能的角度看，这两个模型是相同的。不难看出，广播模型至少与发送/接收模型有同样的功能。下面的定理表明：广播模型的功能并不更强大。

定理21.1 如果A是某个具有可靠广播通道的异步广播系统，那么就存在一个具有可靠FIFO发送/接收通道的异步发送/接收系统B，B与A有相同的用户接口、并按照如下方法“模拟”A。对于B的每个运行α，存在A的一个运行α'，使得下列条件成立：

1）α和α'对U而言是不可区分的（U是用户群U_i的组合）。

2）任取i，仅当α'中发生停止事件$stop_i$时，α中才发生该事件。

并且，如果α是公平的，那么α'也是公平的。

证明概要 对A的每个进程P_i，系统B有一个进程Q_i。Q_i负责模拟P_i，并参加广播通道的模拟。

Q_i首先对全部的$j \neq i$执行$send(m, t)_{i,j}$输出，其中t是一个局部整型tag，然后执行一个内部步模拟$receive(m)_{i,i}$，用于模拟P_i的一个$bcast(m)_i$输出。Q_i所使用tag的值从1开始，随每个后续的$bcast$而递增。如果Q_i接收Q_j所发送的一个消息(m, t)，通过传递该消息，Q_i协助模拟P_j的广播——确切地讲，它向进程i和进程j之外的其他所有进程发送(m,t,j)。如果Q_i从k接收到(m, t, j)，它继续协助向除进程i、进程j、进程k之外的Q_i还没有向其发送(m, t, j)的所有进程发送(m,t,j)。

同时，Q_i 收集最初从每个P_j 广播出来的已标记消息(m, t)，其中$j \neq i$；这些消息或者直接从 P_j 收到、或者是通过其他进程传递来的。 在一定的时侯，允许Q_i 执行一个内部步，以模拟A系统的一个$receive(m)_{j,i}$事件。具体说，在下面的条件下Q_i 可以执行这个事件，即：Q_i 有一个最初 670
从P_j 广播的消息(m, t)，Q_i 已经将(m, t, j) 传递给除进程i 和进程 j之外的所有进程，且Q_i 对来自 P_j 、所有tag 值严格小于 t的消息都已经模拟$receive_{j,i}$事件。

这一证明需要若干关键事实。首先，对于任意的j，每个进程Q_i 只有在成功地向其他所有进程发送相应的(m,t)，从而保证所有进程都将最终从进程j收到(m, t)之后，才会模拟 $receive(m)_{j,i}$事件。其次，尽管进程Q_i 在接收最初由P_j 广播的消息时，接收消息的顺序可能与P_j 广播消息的顺序不同，但$tags$使Q_i 可以对这些消息进行正确的排序。第三，如果一个tag值为t 的消息是由某个进程Q_i 发送的，那么来源于P_i、且其tag值小于t的消息必定已经发送给所有的进程。 □

定理21.1表明，从计算性能的角度看，我们是考虑广播系统还是发送/接收系统并没有关系。当然，这两种系统的复杂性不一样——在上面的模拟中， $receive$ 事件的总数相差大约n 倍——但本章对复杂性问题涉及不多。由于广播系统中的不可能性结果看起来要略微强烈一些，也简化了算法的书写，因此本章选择广播系统。

21.2 有故障环境中一致性的不可能性

我们采用12.1节中的一致性问题定义。尽管该定义是针对共享存储系统给出的，它对异步(广播或发送/接收)网络系统同样适用。在此回顾一下这个定义。

系统A的用户界面由输入操作$init(v)_i$ 和输出操作$decide(v)_i$ 组成，其中$v \in V$、$1 \leqslant i \leqslant n$；同时，$A$还有输入操作$stop_i$。所有下标为$i$的操作在端口$i$处发生。每个用户$U_i$ 有输出操作$init(v)_i$和输入操作$decide(v)_i$，$v \in V$。在任何运行中，假定U_i 最多执行一个$init_i$ 操作。

对于一个由$init_i$ 和 $decide_i$ 操作组成的序列，当它是一个形式为$init_i(v)$，$decide_i(w)$序列的某前缀时，则称这个操作序列对i来说是良构的。在由A和用户U_i 组成的复合系统中，考虑下面的条件：

良构性条件 在任意一个运行中，对任何i， U_i 和 A 之间的交互对i来说是良构的。

一致性条件 在任意一个运行中，所有决定值是同样的。

有效性条件 在任意一个运行中，如果所发生的全部 $init$ 操作都包含同一个值v，那么 v 是惟一可能的决定值。 671

无故障终止条件 在所有端口都发生$init$ 事件的任意一个公平无故障运行中，每个端口都发生一个 $decide$ 事件。

如果一个异步网络系统(对所有用户集合)满足良构性条件、一致性条件、有效性条件和无故障终止条件，我们就说这个系统解决了一致性问题。同样考虑下面的条件

f故障终止条件，$0 \leqslant f \leqslant n$ 在所有端口都发生$init$ 事件的任意一个公平运行中，如果在最多f个端口有$stop$事件，那么在每个无故障端口都发生一个$decide$事件。

无等待终止是当$f = n$时的f故障终止特殊情况。

当然，如果没有容错需求，在异步广播模型中解决一致性问题就很容易。例如， 每个进程可以简单地广播其初始值，并对其收到的初始值向量应用某个适当的、共同认可的函数。由于所有进程都保证能收到同样的向量值，因此这些进程将获得同样的结果。

关于广播系统的主要不可能性结果（重复 17.2.3节）是：

定理21.2　在采用可靠广播通道的异步广播模型中，没有算法能解决一致性问题和保证1故障终止。

17.2.3节给出的证明基于异步广播系统到异步共享系统的转换（定理17.8）以及一个关于异步共享存储模型中一致性问题的不可能性结果（定理12.8）。使用类似于定理12.8的证明方式也可以直接证明这个不可能性结果。这个证明留做一道习题。

21.3　随机算法

定理21.2阐述在异步网络系统中不可能解决一致性问题，即使这个系统仅有一个停止故障。
672 然而，这个问题对分布式计算而言是根本性问题，找到绕过这一内在限制的方法非常重要。为获得绕过这一限制的一个算法，必须弱化正确性需求，强化模型，或两者同时进行。

在本节，同时弱化正确性需求和强化模型。我们将说明在随机异步网络中可以解决一致性问题。这个模型比普通的异步网络模型功能更强，因为它允许进程在计算过程中做随机的选择。另一方面，有效性条件比以前要略微弱化一些：尽管良构性条件、一致性条件和有效性条件仍需要保证，但终止条件是概率性的。即，所有无故障进程在全部输入都到达之后将在时间t做出决定的概率至少为$p(t)$，其中 p 是一个特殊的、单调非递减的无界限函数。这意味着最终终止的概率为1。

在接下来几节，讨论绕过由定理21.2所表达的内在限制的其他方法，包括使用故障检测器、允许一个以上的决定值、允许以近似一致性替代精确一致性。

下面的算法是Ben-Or提出的，该算法在$n > 3f$ 和$V = \{0,1\}$条件下有效。从形式上说，该算法是8.8节中所描述概率化模型的一个实例。

***BenOr* 算法:**

每个进程 P_i 有两个局部变量x 和 y，它们的初始值都为*null*。*init* $(v)_i$ 输入使进程P_i 设置 $x := v$。P_i 执行一组阶段，这些阶段的编号依次为1、2、……，每个阶段由两个回合组成。P_i 从一个$init_i$ 输入收到它的初始值后，开始阶段 1。它将永远不停地执行这个算法，即使在它做出决定之后。

对每个阶段 $s \geq 1$，P_i 执行如下操作：

Round 1: P_i 广播 (“*first*”, s, v)，其中v是x的当前值，然后等待获得$n-f$个形式为(“*first*”, s,*)的消息。如果所有消息都有相同的值v，那么P_i 设置$y := v$，否则设置$y := null$。

Round 2: P_i 广播 (“*second*”, s, v)，其中v是y的当前值，然后等待获得$n-f$个形式为(“*second*”, s,*)的消息。会有三种情况。第一种情况是：如果所有消息都有相同的值v，且$v \neq null$，那么，P_i 设置$x := v$，并且，如果P_i 还未执行任何$decide(v)_i$ 操作，则执行一个$decide(v)_i$。第二种情况是：如果不是所有的消息都有相同的值，但至少有$n-2f$个消息有相同值$v \neq null$，那么，P_i 设置 $x := v$ (但不做决定)。(根据假设$n > 3f$ 可知，不可能存在两个满足这种条件的不
673 同v值。) 否则，P_i 按相同概率随机设置x 为 0 或 1。

需要提醒读者注意的是，*BenOr* 算法的组织和6.3.3节 *TurpinCoan* 算法的组织之间存在相似性。

引理 21.3　*BenOr*算法保证良构性条件、一致性条件和有效性条件。

证明　良构性条件是显而易见的。对于有效性条件，假设在一个执行中所发生的全部*init*事件都包含同一个值v。那么显然，任何完成了阶段1 的进程必须在这个阶段决定值v 。这是因为任何进程在(“*first*”, 1,*) 消息中发送或接收的值全都是v，所以在(“*second*”, 1,*) 消息中发送

的值只能是v。

对于一致性条件，假设进程P_i在阶段s决定值v，并且没有进程在比编号s更前的阶段决定v。那么P_i一定接收了$n-f$个(“*second*”, s, v)消息。这意味着任何其他完成了阶段s的进程P_j至少收到了$n-2f$个(“*second*”, s, v)消息，因为P_i收到其消息的全部进程，P_i收到其消息的进程数不超过f个。这意味着在阶段s，P_j不能决定不同于v的值；而且P_j在阶段s设置$x := v$。因为这一情形对所有完成阶段s的进程P_j都成立，所以（像有效性条件的证明中一样）任何完成阶段$s+1$的进程一定在阶段$s+1$决定v。 □

下面考虑终止条件。首先，不难看出算法将在后续阶段中继续执行；这个事实并不依赖于概率。

引理 21.4 在*BenOr*算法的、且*init*事件在所有端口都发生的每个公平运行中，每个无故障进程完成无限多个阶段。而且，如果l是每个进程任务的时间上界，且d是最早消息处在从每个P_i到每个P_j传递途中的传递时间上界，那么在最后的*init*事件后，每个无故障进程在$O(s(d+l))$时间内完成阶段s。

然而，引理21.4并不意味着每个无故障进程最终做出决定。事实上，*BenOr*算法并不保证这个特征，这一特征只是可能成立。

例 21.3.1 一个无决定的运行

我们来描述$n=3f+1$的一个*BenOr*算法公平运行，该运行中没有进程做出决定。每个阶段s的过程也一样，如下所述：

m个进程从$x=0$开始，$f+1 \leqslant m \leqslant 2f$，其余进程从$x=1$开始。在第一回合后，所有进程有 674
$y=null$，且在第二回合，所有进程都随机选择自己的新x值。其中m'个进程的随机选择为0，$f+1 \leqslant m' \leqslant 2f$，其余为1，导致$m'$个进程以$x=0$开始阶段$s+1$，其余进程以$x=1$开始阶段$s+1$。

与11.4节相同，我们假设算法中的所有非确定性选择——这里指下一步发生哪个操作、什么时候执行、结果状态如何——都受一个对手的控制。我们强制这个对手来加强所有进程I/O自动机和广播通道自动机的公平性条件。我们同时还强制该对手遵从常用的时间限制：进程内任务的时间上界l，以及最早消息处于从每个P_i到每个P_j传递途中的传递时间上界d。最后，我们需要该对手允许在所有端口发生*init*事件。假设该对手知道全部的过去执行。任意一个这样的对手可决定算法执行的概率分布。

引理 21.5 对于任意一个对手和任意$s \geqslant 0$，所有无故障进程在$s+1$个阶段内做出决定的概率至少为$1-(1-\frac{1}{2^n})^s$。

证明概要 在$s=0$时比较简单。考虑$s \geqslant 1$的阶段。我们论述所有无故障进程在阶段s结束时选择同样值x的概率至少为$\frac{1}{2^n}$（不必考虑在其他阶段如何解决随机选择）。这种情况下，根据一致性，所有无故障进程在阶段$s+1$结束时做出选择。

对于阶段s，考虑任意一个最短有限运行α，某个无故障进程P_i在α中已经收到$n-f$个(“*first*”, s, $*$)消息。（因此，α在结束时发送其中的一个消息。）如果在这些消息中至少有$f+1$个消息包含一个特殊的值v,，那么在运行α后定义v为一个优选值，这样的优选值可能有一到两个。我们有下述结论：如果在运行α后只有一个优选值v，那么在α的任意扩展中所发送的每个消息(“*second*”, s, $*$)一定都包含值v或*null*。这是因为如果P_i收到v的$f+1$个副本，那么其他每个进程至少收到v的一个副本，因此不会发送消息(“*second*”, s, $\bar{v}$)。(这里我们用符号$\bar{v}$

表示值 $1-v$。）同样，如果在运行α后有两个优选值，那么在α的任意扩展中所发送的每个消息（“*second*”, s, $*$)中一定包含值*null*。

因此，如果只有一个优选值v，那么在α的任意扩展中，阶段s结束时，v是可以被“强制”作为任何进程的x值的惟一值，这一“强制”赋值是由一个随机赋值操作完成。同样，如果有
675 两个优选值，那么不能用这种方法进行强制赋值。因为没有进程为α中的阶段s做随机选择，因此在进行阶段s的随机选择前，所有能在阶段s中进行强制赋值的变量值就已经确定。

所以，如果只有一个优选值，那么所有随机选择其x值的进程将选择这个优选值的概率至少为 $\frac{1}{2^n}$，从而与那些非随机选择的进程一致。同样，如果有两个优选值，那么所有进程将(随机)选择相同x值的概率至少为 $\frac{1}{2^n}$。无论是那种情况，在阶段s结束时，所有无故障进程有相同x值的概率至少为 $\frac{1}{2^n}$。

至此，对每个阶段s的讨论仅与本阶段的随机选择有关，而与其他阶段的选择无关。因此我们可以将不同阶段的概率值综合起来，可以看出在阶段s'结束时，所有无故障进程获得相同x值的概率至少为 $1-(1-\frac{1}{2^n})^{s'}$，其中$1 \leqslant s' \leqslant s$。因此，所有无故障进程在阶段$s+1$内做出选择的概率至少为 $1-(1-\frac{1}{2^n})^{s}$。 □

下面定义一个从$\mathbb{N}^+$到$R^{>0}$的函数T，使得在最后一个*init*事件后，每个无故障进程在$T(s)$时间内完成阶段s。根据引理21.4，可以选择$O(s(d+l))$作为$T(s)$。同样，如果$t < T(1)$则定义$p(t)$为0，如果$s \geqslant 1$、$T(s) \leqslant t < T(s+1)$，则定义$p(t)$为 $1-(1-\frac{1}{2^n})^{s-1}$。那么根据引理21.5和引理21.4可导出：

引理 21.6　对任意一个对手和任意$t \geqslant 0$，在最后一个*init*事件后，所有无故障进程在时间t内做出选择的概率为$p(t)$。

主要的正确性结果如下：

定理 21.7　*Benor* 算法保证良构性条件、一致性条件和有效性条件。它同样保证所有无故障进程最终做出选择的概率为1。

证明　根据引理21.3、引理21.6和引理21.4。（为证明$p(t)$是无限的，需要引理21.4。） □

676 **随机一致性与非随机一致性**　*BenOr* 算法之所以重要的一个原因是，它证实随机异步网络模型和非随机异步网络模型间的一个本质差别。具体来说，在会出现进程故障的非随机模型中，根本不可能解决一致性问题，而这个问题在随机模型中却可以轻松解决(概率为1)。11.4节中的*LehmannRabin*算法给出一个类似的对比。

减少复杂性　*BenOr*算法并不实用，因为它的概率时间限制太高。通过提高不同进程在同一阶段有相同随机值的概率，可以降低*BenOr*算法的时间复杂性。但是，需要使用密码技术，对该技术的讨论超出这里所给模型的范围。

21.4 故障检测器

在容易发生故障的异步网络中，解决一致性问题的另一种方法是增加一种被称为故障检测器的新型系统组件，对模型进行强化。故障检测器是一个模块，用于向一个异步网络中的进程提供关于以前进程故障的信息。根据所提供的停止信息是否总是正确以及是否完整，可将故障

检测器分为多种类型。最简单的一类故障检测器是完美故障检测器，这种故障检测器只报告那些确实已经发生的故障、并保证最终将所有故障的信息报告给其他所有无故障进程。

为给出完美故障检测器的形式化表示，我们考虑一个系统A，A的结构与异步网络系统的结构相同，但对每个由端口i和j组成的端口对都分别增加一个附加输入操作$inform\text{-}stopped(j)_i$，其中$i \neq j$。系统$A$的一个完美故障检测器是一个有输入操作$stop_i$ $(1 \leqslant i \leqslant n)$和输出操作$inform\text{-}stopped(j)_i$ $(1 \leqslant i, j \leqslant n, i \neq j)$的I/O自动机。这一自动机的构造思想是使得故障检测器可以知道发生在网络任何地方的停止故障，并将这些故障通知给其他进程。每个$inform\text{-}stopped(j)_i$操作表示端口$i$处的一个关于进程$j$已经停止的断言。一个简单的三进程系统的结构如图21-1所示。下面的算法在与完美故障检测器一起使用时可解决一致性问题。

***PerfectFDAgreement*算法（非形式化）：**

每个进程P_i都尽量使下列两部分数据稳定不变：

1）向量val，索引为$\{1,\cdots\cdots,n\}$，其中的值属于$V \cup \{null\}$。当$val(j) = v \in V$时，则表明P_i已经知道P_j的初始值为v。

2）进程索引的集合$stopped$。当$j \in stopped$时，则表明P_i知道P_j已经停止。

进程P_i不断地广播当前的val和$stopped$数据，并从那些不属于$stopped$的进程接收数据，根据所接收的数据对val和$stopped$数据进行更新。但P_i忽略从那些已经属于$stopped$的进程发来 677
的消息。同时，P_i还保留那些“批准”其数据的进程的踪迹。当一个进程发送给P_i的$(val, stopped)$数据与P_i自己的$(val, stopped)$数据相同时，则表示该进程“批准”P_i的数据。当P_i达到一个其数据已经“稳定”的状态时，即当它从所有未停止的进程都收到对其当前数据的批准时，P_i将以与其val向量中的最小索引所对应的非空值作为选择。

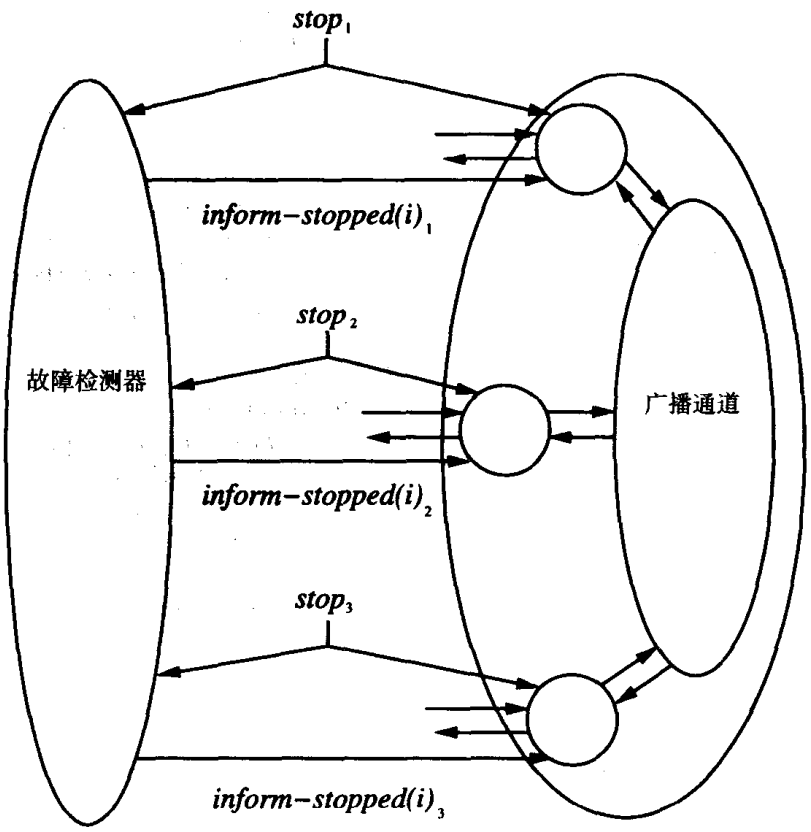


图21-1 带完美故障检测器的异步广播系统的结构图

具体代码如下。令W表示一组向量的集合，用$\{1,\cdots, n\}$对这些向量进行索引，向量的元素属于$V \cup \{null\}$。我们定义一个关于二元组(w, I)的偏序关系，其中$w \in W$、$I \subseteq \{1,\cdots, n\}$。具体说，当下述两个条件满足时，我们记$(w, I) \leq_d (w', I')$并称$(w', I')$支配$(w, I)$。

678

1）对所有k，如果$w(k) \in V$，那么$w(k) = w'(k)$。

2）$I \subseteq I'$。

这一偏序关系表达的意思是(w', I')至少包含(w, I)的所有信息。

为避免混淆，我们不具体描述发生一个$stop_i$事件后P_i的行为。就像通常一样，进程P_i因为$stop_i$事件而停止。

PerfectFDAgreement$_i$ automaton (formal):

Signature:

Input:
- $init(v)_i$, $v \in V$
- $receive(w, I)_{j,i}$, $w \in W$, $I \subseteq \{1, \ldots, n\}$, $1 \leq j \leq n$
- $inform\text{-}stopped(j)_i$, $j \neq i$

Output:
- $bcast(w, I)_i$, $w \in W$, $I \subseteq \{1, \ldots, n\}$
- $decide(v)_i$, $v \in V$

States:

$val \in W$, initially identically *null*
$stopped \subseteq \{1, \ldots, n\}$, initially $\emptyset$
$ratified \subseteq \{1, \ldots, n\}$, initially $\emptyset$
decided, a Boolean, initially *false*

Transitions:

$init(v)_i$
- Effect:
  - $val(i) := v$
  - $ratified := \{i\}$

$inform\text{-}stopped(j)_i$
- Effect:
  - $stopped := stopped \cup \{j\}$
  - $ratified := \{i\}$

$bcast(w, I)_i$
- Precondition:
  - $w = val$
  - $I = stopped$
  - $val(i) \neq null$
- Effect:
  - none

$receive(w, I)_{j,i}$
- Effect:
  - if $j \notin stopped$ then
    - if $(w, I) = (val, stopped)$ then
      - $ratified := ratified \cup \{j\}$
    - else if $(w, I) \not\leq_d (val, stopped)$ then
      - $stopped := stopped \cup I$
      - for all k, $1 \leq k \leq n$, do
        - if $val(k) = null$ then $val(k) := w(k)$
      - $ratified := \{i\}$

$decide(v)_i$
- Precondition:
  - $ratified \cup stopped = \{1, \ldots, n\}$
  - $v = val(j)$, where j is the smallest index with $v(j) \neq null$
  - $decided = false$
- Effect:
  - $decided := true$

Tasks:

$\{bcast(w, I)_i : w \in W, I \subseteq \{1, \ldots, n\}\}$
$\{decide(v)_i : v \in V\}$

679

定理 21.8　当与任意一个完美故障检测器一起使用时，*PerfectFDAgreement*算法解决一致性问题并保证无等待终止。

证明 良构性条件和有效性条件是显而易见的，无须证明。对于无等待终止条件，我们考虑一个在所有端口都发生*init* 事件的公平运行α，并令i表示某个无故障端口；我们要证明P_i 最终在运行α中做出选择。需要指出的是，P_i 的$(val_i, stopped_i)$数据每次在α中改变时，一定是新的(w, I)支配旧的(w, I)。由于所有可能二元组(w, I)的总量有限，因此$(val_i, stopped_i)$数据一定会达到最终值(w_{final}, I_{final})。 如果 P_i 在此之前做出选择，则定理得证，因此假设P_i 在此之前没有做出选择。于是我们给出一个命题，在此之后最终有$ratified_i \cup stopped_i = \{1,\cdots, n\}$，依据这一命题足以说明会发生一个 $decide_i$ 事件。只要证明每个没有故障的进程$j \neq i$最终都批准这个 (w_{final}, I_{final})对，就证明了该命题。

因此我们考虑任意一个无故障的进程$j \neq i$。最终，P_j 会收到一个由P_i广播的包含 (w_{final}, I_{final})的消息，在此之后P_j 的二元组(w, I)将总是支配 (w_{final}, I_{final})。但是 P_j 的二元组(w, I)不能永远严格支配(w_{final}, I_{final})，因为否则P_j 最终会将这个新信息传递给P_i。因此，P_j 的二元组(w, I)最终将与 (w_{final}, I_{final}) 相等并且不再改变。这样，P_i 最终会收到从P_j 广播的一个包含(w_{final}, I_{final})的消息。从而将j 放入$ratified_i$，这正是我们期望的结果。

最后我们证明一致性条件。假设P_i 是第一个做出选择的进程，令w和I分别是$decide_i$事件π发生时val_i和$stopped_i$ 的值。那么在运行α中，I中的所有进程在π之前都已经出现故障，因此都再也不可能做出选择。 令$J = \{1,\cdots\cdots, n\} - I$；我们要证明$J$中所有做出选择的进程必定与$P_i$选择相同的值。

在π发生时，J中的每个进程j必定也属于$ratified_i$，因此在π之前的某个点t_j一定已经以 (w, I)作为其局部数据。我们的命题是：在α中的t_j点后，J的每个进程j一定有$val = w$并始终保持这一结果，这就意味着如果它做出选择，则它与P_i保持一致。

我们假设这一命题不成立，并令j是J中第一个获得包含不属于w的信息的val向量（例如，向量的某些元素属于V，而对应的w元素却为$null$）。那么这种val向量的获得必定是因为在点t_j后发生一个$receive_{k,j}$事件，这一$receive_{k,j}$ 事件发生时，广播进程P_k 有一个包含不属于w的信息的val向量。由于在点t_j后，P_j忽略I中的所有进程，因此广播进程P_k一定属于J。但是这与进程j是J中第一个获得不属于w的信息的进程相矛盾。 □

复杂性 *PerfectFDAgreement*算法的通信复杂性和时间复杂性是无上界的。这并不可怕，因为在本章只讨论计算能力问题。但是，设计复杂性有限的类似一致性的可能性是存在的。我们将其留做一道习题。 680

21.5 *k*—致性

下面我们考虑对问题陈述进行弱化。像7.1节和12.5节分别针对同步网络环境和异步共享存储环境所描述的一样，*k*—致性问题是一致性问题的一个变种，在异步网络中只要限制故障的数量$(f < k)$就可以解决该问题。我们采用与12.5节中相同的问题定义：具体说，这个问题与普通的一致性问题有相同的良构性条件和终止条件，而一致性条件和有效性条件则用下面的条件定义代替，其中k是某个整数且$k \geqslant 1$。

一致性条件 在任何运行中，存在V的一个子集W，$|W| = k$，使得所有决定值都属于W。

有效性条件 在任何运行中，任何进程的任何决定值都是某个进程的初始值。

这个一致性条件比一般一致性问题的一致性条件要弱一些，表现在该一致性条件允许k个决定值，而不只是1个决定值。有效性条件比通常一致性的有效性条件略微有所加强。下面是一个用于解决异步广播网络中的k-集合一致性问题的简单算法，其中$f < k$。

TrivialKAgreement **算法：**

进程 $P_1, P_2, ..., P_k$（只）广播它们的初始值。每个进程P_i以它所接收的第一个值为决定值。

定理21.9　当$f < k$时，*TrivialKAgreement*算法解决k—致性问题并保证f故障终止。

基于稳定的向量，不难设计出一个类似于*PerfectFDAgreement* 的k—致性算法。我们将该问题留做一道习题。另外，用定理17.5把异步共享存储模型转化为异步网络模型，我们可以根据异步共享存储模型的算法获得异步网络模型的k—致性算法；但是，这个方法的缺点是必须在$n > 2f$时才有效，而 *TrivialKAgreement* 和基于稳定向量的算法在$n \leqslant 2f$时仍然有效。

681　事实上，故障数$\geqslant k$时不能解决k—致性问题。

定理 21.10　在异步广播模型中，k故障终止时k—致性问题是无解的。

证明　根据定理12.13和定理17.8。　□

21.6　近似一致性

我们再次弱化问题陈述。像7.2节和12.5节分别针对同步网络环境和异步共享存储环境所描述的一样，一致性问题的另一个变种是近似一致性问题。采用12.5节中同样的问题定义。具体说，值的集合V是实数的集合，并允许进程在消息中发送实数型数据。与一致性问题中要求完全一致不同，近似一致性问题的要求是各自之间的相差值不能大于一个绝对公差ε。这个问题与普通一致性问题有相同的良构条件和终止条件，而一致性条件和有效性条件则定义如下。

一致性条件　在任何运行中，任何两个决定值都在彼此的公差ε范围内。

有效性条件　在任何运行中，任何决定值都在初始值的范围内。

在会出现停止故障的异步环境中，存在一个类似于7.2节中*ConvergeApproxAgreement*算法的算法，该算法在$n > 3f$条件下有效。每个进程P_i执行一组阶段，在每个阶段它不需要等待接收来自全部n个进程的消息，而只是等待接收来自其中任意$n-f$个进程的消息。（它不能等待接收来自全部进程的消息，因为有多达f个的进程可能停止。）因为我们现在只考虑停止故障，所以P_i不需要丢弃极端值以“减小”其值的多集。下面描述中使用的*mean*函数和 *select* 函数以及像实数多集的*width*这样的符号，都是在7.2节中定义的。

AsynchApproxAgreement **算法：**

假设 $n > 3f$。每个P_i维护一个变量*val*，用于记录其最近的估算。该变量被初始化为在一个 $init(v)_i$输入中到达的值v。在每个阶段，P_i进行如下操作：首先，广播 *val* 值，并用阶段号s对这个值进行标记。然后，它将在阶段s所接收的前$n-f$个值放入多集W。最后，它将*val* 置为

682　*mean*(*select*(*W*))。

显然，任何进程在任何阶段s所选择的*val*在所有进程在阶段$s-1$所选择的全部*val*的范围内（当$s = 1$时，就在全部初始*val*值的范围内）。我们的命题是，在每个阶段，*val*多集的宽度按照一个至少为 $\left\lfloor \frac{n-f-1}{f} \right\rfloor + 1$ 的因子收缩。因为 $n > 3f$, 所以这个区域是不断收敛的。

引理 21.11　在一个*AsynchApproxAgreement*运行的阶段s，令v和v'分别是进程 P_i和 P_i'所选择的 val_i 和val_i'的值，那么

$$|v - v'| \leqslant \frac{d}{\left\lfloor \frac{n-f-1}{f} \right\rfloor + 1}$$

其中，当$s \geq 2$时，d是在阶段$s-1$所选择的val值的范围的宽度；当$s=1$时，d是初始值的宽度。

证明 与引理7.17的证明类似。 □

终止条件 到此为止，我们关于*AsynchApproxAgreement*所介绍的每个结论都只有在假定$n>2f$(而不是$n>3f$)满足时才有效。但还没有一个完整的算法，因为我们还没有说进程何时真正做出选择。我们使用额外的进程来帮助达到终止。

在*ConvergeApproxAgreement*中使用的简单终止策略在这里不能使用，因为一个进程不能在阶段1等待从所有进程收到消息，所以不能确定初始值多集的范围上界。然而，可以对这个策略做少许修改，在算法的开始增加一个特殊的初始化阶段0。在阶段0，每个进程P_i广播其val，搜集$n-f$个$vals$的多集，并选择多集的中间值作为阶段1所用的新val。因为$n>3f$，所以很容易验证这一点：在阶段0，任何进程P_i选择的任何val位于任意进程P_j在阶段0所搜集的多集范围内。因此，每个P_i都可以用其在阶段0所搜集多集的范围计算一个阶段号，根据这个阶段号可以确定在阶段s时任意两个进程的val值的公差最大为ε。这一策略的剩下部分与*ConvergeApproxAgreement*的相同。

*AsynchApproxAgreement*算法也不是最佳算法，理由是该问题实际上对任何$n>2f$都是可解的。但需要一个更复杂的算法。例如，根据一个基于单写/多读共享寄存器并保证无等待终止的共享存储近似一致性算法A，可获得一个能在$n>2f$时有效的算法。定理12.14断言存在这样的算法A（也可以在[24]中找到）。使用定理17.5，可以推导在$n>2f$时解决近似一致性问题并保证f故障终止的异步网络算法的存在性。[⊖] 另一方面，不难看出在$n \leq 2f$时是无法解决近似一致性问题的。 683

定理21.12 当$n \leq 2f$时，在会出现f故障终止的异步广播模型中，近似一致性问题无解。

证明概要 这一证明与定理17.6的证明相似。简单地讲，假设存在这样的一个算法，并令G_1为$1,\ldots,n-f$的集合，G_2为集合$n-f+1,\ldots,n$。考虑一个公平运行α_1，其中的所有进程以值v_1开始，并且所有索引属于G_2的进程在开始时即发生故障。根据f故障终止条件可知，G_1中的所有进程必定最终做出选择，而有效性条件表明它们必定选择v_1。与此对应，考虑另一个公平运行α_2，其中的所有进程以值v_2开始，这里$|v_1-v_2|>\varepsilon$，并且所有索引属于G_1的进程在开始时即发生故障。在α_2中，G_2中的所有进程必定最终选择v_2。

像在定理17.6的证明中一样，我们接下来将α_1和α_2结合来构造一个有限运行α。在α中，属于G_1的进程选择v_1，属于G_2的进程选择v_2，从而与一致性条件矛盾。 □

21.7 异步网络的计算能力*

采用与在定理17.6和定理21.12的证明中使用的相同构造方法，可以证明在异步网络中，如果半数进程可能发生故障，那么许多其他全局协调问题都是无解的。

同在12.5节中一样，我们可以考虑异步网络中任意的决定问题的可解性。普通一致性问题、k一致性和近似一致性问题都是决定问题的例子，并且我们已经给出这些问题在异步网络中计算能力的主要结果。至于读/写共享存储模型，我们给一个定理，该定理给出一些条件，根据

⊖ 为应用定理17.5，A必须满足17.1.1节中所给的“轮换”限制。可以构造共享存储近似一致性算法以满足这一条件。

这些条件可知有一个问题在1-故障终止的异步网络模型中是无解的。

定理 21.13　令D为一个决定映射，在异步广播模型中，它的决定问题在1-故障终止时是可解
684 的。那么一定存在一个决定映射D'，对所有w都有$D'(w) \subseteq D(w)$，使得下列两个结论都成立：

1）如果输入向量w和w'只在一个位置不同，那么存在$y \in D'(w)$和$y' \in D'(w')$，使得y和y'至多只在一个位置不同。

2）对每个w，$D'(w)$所定义的图是连通的。

证明　根据定理12.15和定理17.8。 □

通常，用定理17.8，可以将读/写共享存储环境中计算能力的不可能性结果转换为适用于异步网络的结论。用定理17.5也可以对算法进行相应的转换，但只能在定理17.5的限制条件下进行这种转换，其中包括$n > 2f$的要求。

21.8 参考文献注释

定理21.2首先是由Fischer、Lynch和Paterson[123]证明的，这一定理描述在会出现停止故障时的一致性不可能性[123]。他们三人的原始证明是直接根据异步广播模型给出的，没有借助转换。Loui 和 Abu-Amara[199]发现使用同样的证明方法，可以将定理21.2拓展到读/写共享存储模型。我们对定理12.8的证明使用的就是 Loui 和 Abu-Amara的发现。在习题21.2习题21.3和习题21.4中概述Fischer、Lynch和Paterson所提出的原始证明，但根据Bridgland 和 Watro[58]的建议重新进行一些调整。

Benor 算法是 Ben-Or[46]发明的。后来Rabin[248]和 Feldman[114]所做的工作得出其他具有更好时间复杂性（实际为常数）的随机算法。这些算法使用“机密共享”技术，以提高在同一阶段不同进程选择相同随机值的概率。

故障检测器的概念是由Chandra 和Toueg [66]以及由Chandra、Hadzilacos和Toueg [65]所定义和完善的。这些文章不仅描述这里讨论的完美故障检测器，而且讨论其他完美性略弱些的变种，包括错误地将进程识别为故障的故障检测器、以及未成功地向所有进程通报故障的故障检测器。这些相对弱一些的故障检测器也可以用于解决一致性问题，并且在实际的分布式系统，可以采用超时机制实现其中的某些故障检测器。Hadzilacos和Toueg[143]也对故障检测器进行讨论 。

在第7章和第12章的参考文献注释中，我们已经讨论了k一致性问题和近似一致性问题的起源。 Attiya、Bar-Noy、Dolev、Koller、Peleg和Reischuk[19,20,40]描述另外一些在有故障的异
685 步网络中可解的有意义问题，包括进程重命名问题和插入式排除问题。Bridgland 和 Watro[58]描述资源分配问题，该问题在带有故障的异步网络中是可解的。稳定向量算法的思想应归功于Attiya 等[20]。

定理 21.12的证明采用的是由Bracha 和Toueg [56]以及由Attiya、Bar-Noy、Dolev、Peleg和Reischuk[20]提出的证明方法。基于Moran和Wolfstahl[230]早先提出的不可能性结果，Biran、Moran和Zaks[51]描述在1-故障终止的异步网络中可解决的决定问题的特征。定理 21.13 就来自这两篇论文。

21.9 习题

21.1 证明定理21.1。

21.2 假设V={0, 1}。如果 A 是一个可解决一致性问题的异步广播系统，那么采用12.2.2节中同样的方法，定义A的有限运行的0价、1价、单价和双价，并定义A的初试化操作。

a）给出这样一个系统 A的一个例子，其中有一个双价初始化操作。

b）给出这样一个系统 A的一个例子，其中的全部初始化操作都是单价的。

c）证明：如果A 保证1-故障终止，那么就存在一个双价初始化操作。

21.3 令V、A同习题21.2。当一个有限无故障、输入优先的运行α对某个i满足下列条件时，则称运行α是一个*决定者*(*decider*)：

a）α是双价的。

b）存在α的一个 0价无故障扩展α_0，使得α_0中位于α之后的部分仅由进程i的步组成。

c）存在α的一个1价无故障扩展α_1，使得α_1中位于α之后的部分仅由进程i的步组成。即，单个进程i 可以用两种不同的方式工作（例如，用两种不同的方式交替局部控制的消息接收步，或接收两个不同的消息序列），以用两种不同的方式解决最终选择。 686

证明：如果 A 有一个双价初始化操作，那么 A 有一个决定者。需要提醒的是，我们只假设 A 解决一致性问题，并没有给出容错假设。(提示：参考引理12.7的证明。)

21.4 使用习题21.2 和习题21.3的结果证明定理21.2。

21.5 用广播模型重新考虑本章的一致性问题。考虑一个比通常停止故障更加受限制的故障模型，该模型中进程只能在计算开始时发生故障。(换句话说，所有的*stop* 事件都发生在所有其他事件之前。) 在这个模型中是否能解决一致性问题，以保证：

a）1 – 故障终止？

b）当$n>2f$时f故障终止？

c）无等待终止？

在每种情况下，给出实现算法或不可能性的证明。

21.6 设计*BenOr* 算法的一个变种，使得其中所有无故障进程都将最终停止。

21.7 针对下列情况，分别设计*BenOr*随机一致性算法的变种：

a）带停止故障的同步网络模型。

b）带Byzantine故障的同步网络模型。

c）带Byzantine故障的异步网络模型。（如14.1.1节所述，通过允许用一个与进程P_i有相同外部接口的任意I/O自动机代替P_i，可以模型化P_i的一个Byzantine故障。）

在每种情况下，尽可能设计一个能为尽可能少进程工作的算法，进程的数量是与所容忍故障数量f相比较而言的。

21.8 采用一个任意值的集合V代替{0, 1}，为带停止故障的一致性设计一个随机异步算法。请使用尽可能少的进程。（提示：将*TurpinCoan*算法和*BenOr*算法的思想结合起来。）

21.9 在Byzantine故障的情况下重复习题21.8。 687

21.10 给*PerfectFDAgreement*设计一个可替代的一致性，该一致性也使用完美故障检测器获得无等待一致性，但它的通信和时间复杂性“小”。请尽量获得尽可能小的通信和时间复杂性。

21.11 定义一个*非完美故障检测器*如下。它拥有与完美故障检测器相同的外部接口，并为每个j和i都增加一个*inform-not-stopped*$(j)_i$ 操作，其中$j \neq i$。这个操作用于纠正前面的*inform-stopped*$(j)_i$操作，即：通知进程P_i 应忽略前面的错误通知，P_j 实际上并没有停止。一个非完美故障检测器可以交替发生任意次数的*inform-stopped*$(j)_i$ 和 *inform-not-stopped*$(j)_i$ 事件。但在该故障检测器的任何公平运行α中，对任何i和j只能有有限多个这种事件，并且其中的最后事件必须包含正确的信息——说明在α中是否发生事件$stop_j$。

假设$n > 2f$。使用某个非完美故障检测器，设计一个算法，解决保证f故障终止的一致性问题。

21.12 证明：当$n \leq 2f$时，使用习题21.11中定义的任意一个非完美故障检测器，不存在算法能够解决保证f故障终止的一致性问题。

21.13 给出“稳定向量”算法一个类似于*PerfectFDAgreement*的前提-结果代码，以解决k一致性问题。证明：在$f < k$时该代码能正确工作。（提示：状态只包含组件*val*、*ratified*和*decided*，不包含*stopped*组件。当$|ratified| \geq n - f$时，可以做出一个决定。）

21.14 对于一个k一致性算法的一个有限运行α，如果出现在α扩展中不同决定值的数量正好为m，则称α是m价的，并像在12.2.2节中一样定义一个*initialization*。证明（不使用定理21.10）：在异步广播模型中，保证k故障终止的任何k一致性算法一定有一个（$k + 1$）价的初始运行。（提示：使用7.1节中的思想，包括Sperner引理。）

21.15 给出*AsynchApproxAgreement*算法的完整前提-结果代码，包括终止一致性。证明其正确性。

21.16 修改*AsynchApproxAgreement*算法及其证明，使其能够用于Byzantine故障的情况。需要多少进程？（提示：采用同步Byzantine环境的*ConvergeApproxAgreement*算法的思想。）

688

21.17 用定理21.12的证明中的构造方法，证明一个你能够证明的最通用的不可能性结果。

21.18 给出一个在1-故障终止的异步网络中可以解决的决定问题（见12.5节中定义）的全面描述。（警告：这非常困难。）

689
~
690

第22章 数据链路协议

本章考虑用不可靠通道实现可靠FIFO通信的问题。这是通信网络解决的最基本问题之一。这里所讲的“不可靠通道”包括两种类型：一种是会出现消息丢失或复制类故障的通道，另一类是对消息重排序的通道。还要考虑进程崩溃的情况，这种情况下会丢失进程的状态信息。我们只在一个有两个节点的特殊网络环境中考虑这个问题。

首先介绍两个简单而著名的算法：Stenning协议和位交换协议。在Stenning协议中，发送端进程将（无界）整数标志添加到用户提交的消息中；协议容许通道中的消息丢失、复制和重排序。而位交换协议不同，只使用有界标志，容许消息丢失和复制，但不能对消息重排序。接着，考虑在有界标志情况下，是否可以容许消息重排序。最后，考虑因进程崩溃而丢失状态信息的情况。

本章将从两个层次来讨论消息：通信系统的用户层和传递消息的通道层。为了区别这两类消息，分别将它们称作“高层消息”和“低层消息”。通常用M和M'分别表示高层消息和低层消息的字母表。另外，用大写字母（例如*SEND*和*RECEIVE*）表示用户接口的操作，仍然用小写字母例如（*send*和*receive*）表示通道接口的操作。

本章用于算法建模的技术(用I/O自动机、合成和模拟关系)也适合于分层通信结构（例如ISO分层结构）的建模。

22.1 问题阐述

考虑问题的环境是一个异步发送/接收网络，由节点1和2组成，用一条无向边连接。需要解决的问题是：在这两个节点端的用户U_1和U_2之间，实现可靠的FIFO通信。节点1上，U_1把消息提交给进程P_1，希望随后能将消息传递给U_2。每个消息只能传递一次，而且消息要按照被提交的顺序传递。

像14.1.2节和例8.1.1中定义的那样，用F表示字母表为M、从节点1到2的通用可靠FIFO发送/接收通道；这里将外部操作重命名为$SEND(m)_{1,2}$和$RECEIVE(m)_{1,2}$，$m \in M$。这样，协议的正确性需求就是它必须“实现”F，也就是说，当把它的任意一个公平运行α投影到F的外部操作上时，都要产生一个F的公平轨迹。采用第8章介绍的I/O自动机符号，这一需求可以更精确地表示为$\alpha|ext(F) \in fairtraces(F)$。

这个通用可靠FIFO通道F实质上是一个无界队列，因此F的任何实现都需要无界的存储空间。这个问题的另一个建模方法是用U_1与通道的显式握手，通过握手通道通知U_1何时可以提交下一个高层消息。这样可以避免需要无界的存储空间。但是，却增加了握手协议建模的复杂性。

运行实现F代码的两个进程用I/O自动机模拟。两个方向上连接它们的通道也都是I/O自动机，但这两个通道通常不是可靠FIFO通道。它们可能会丢失、复制或重排序低层消息。

我们不考虑其他特定类型的不可靠性，例如制造伪消息。另外，我们对消息丢失也进行限制——通常假定这样的活性属性：

如果无限数量的消息被发送，那么就有无限数量的消息被传输。

可以用两种方法对这一属性进行形式化——用14.1.2节定义的*强丢失限制(SLL)*和*弱丢失限制*

(*WLL*)条件。二者的区别在于SLL条件指明通道对每个特定的消息类型都是公平的。本章将根据需要使用这两个条件。并常对消息复制设定有限的范围。

692 本章所需大部分通道的被许可行为的形式化描述已经在14.1.2节中出现过。其中有些描述本身就是I/O自动机(并使用I/O自动机的公平性表达所需要的活性条件)。有些描述是公理性的，采用从*receive*事件到*send*事件的*cause*函数形式。还有些描述是一个自动机与一组附加活性条件的组合。本章将使用这三种类型的描述，哪种方便就用哪种。

本章中我们考虑的体系结构如图22-1所示。包括两个进程自动机P_1和P_2，两个通道自动机$C_{1,2}$和$C_{2,1}$，每个通道分别在一个方向。进程通过*SEND*和*RECEIVE*操作与用户交互，用*send*和*receive*操作与通道交互。在22.5节，还要引入新的操作来模拟进程崩溃。

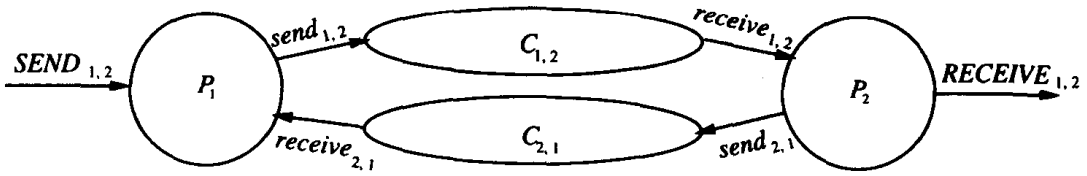


图22-1　数据链路协议结构图

22.2 Stenning协议

在不可靠的通道上保证可靠FIFO消息传输的最简单协议应是Stenning协议。该协议容许全部三种类型的通道不可靠性：(有限)丢失、(有限)复制和重排序。

*Stenning*协议(非形式化)：

进程P_1将用户U_1提交的高层消息放入缓冲区$buffer_1$，并对它们加上从1开始的连续整数标志。P_1不断将缓冲区$buffer_1$中的第一条消息及其标志发送给P_2。进程P_2首先从它所收到的消息中接收标志为1的消息。然后，P_2接收后续消息，并且每次所接收消息的标志正好比前一个被接收消息的标志大1。P_2把接收到的消息放入缓冲区$buffer_2$，并按照顺序将$buffer_2$中的消息传递给U_2。

P_2每接收一条高层消息，都将该消息的标志发回P_1作为确认。当P_1收到当前标志的确认后前移，开始处理下一条高层消息。

693 下面是*Stenning*协议中进程P_1的代码。

$Stenning_1$ automaton (formal):

Signature:

Input:
$SEND(m)_{1,2}$, $m \in M$
$receive(k)_{2,1}$, $k \in \mathbb{N}$

Output:
$send(m,k)_{1,2}$, $m \in M$, $k \in \mathbb{N}$

States:
buffer, a FIFO queue of elements of M, initially empty
$tag \in \mathbb{N}$, initially 1

Transitions:
$SEND(m)_{1,2}$
Effect:

$receive(k)_{2,1}$
Effect:

add m to *buffer*

$send(m,k)_{1,2}$
Precondition:
m is first on *buffer*
$k = tag$
Effect:
none

if $k = tag$ then
remove first element (if any) of *buffer*
$tag := tag + 1$

Tasks:
$\{send(m,k)_{1,2} : m \in M, k \in \mathbb{N}\}$

下面是进程P_2的代码。

$Stenning_2$ automaton (formal):

Signature:

Input:
$receive(m,k)_{1,2}, m \in M, k \in \mathbb{N}$

Output:
$RECEIVE(m)_{1,2}, m \in M$
$send(k)_{2,1}, k \in \mathbb{N}$

States:
buffer, a FIFO queue of elements of M, initially empty
$tag \in \mathbb{N}$, initially 0

694

Transitions:

$RECEIVE(m)_{1,2}$
Precondition:
m is first on *buffer*
Effect:
remove first element of *buffer*

$send(k)_{2,1}$
Precondition:
$k = tag$
Effect:
none

$receive(m,k)_{1,2}$
Effect:
if $k = tag + 1$ then
add m to *buffer*
$tag := tag + 1$

Tasks:
$\{RECEIVE(m)_{1,2} : m \in M\}$
$\{send(k)_{2,1} : k \in \mathbb{N}\}$

*Stenning*协议中的通道$C_{1,2}$和$C_{2,1}$是满足例14.1.2所给损耗式重排序通道规范的I/O自动机(操作被重命名)。也就是说，它们允许消息的有限丢失、有限复制和重排序。在例14.1.2中，用一个基本自动机和一组附加活性属性的组合给出所允许的通道行为。我们之所以采用这种形式的规范说明，是因为它能够产生我们可在不变式断言和模拟关系中使用的一种显式状态。组合P_1、P_2、$C_{1,2}$和$C_{2,1}$就得到完整的*Stenning*协议。

为了证明*Stenning*协议的正确性，我们先介绍几个不变式。但必须注意这样一个学术原则：

由于我们需要的不变式涉及通道状态，因此这些不变式必须以通道自动机的形式给出。因而，我们在陈述不变式时，采用例14.1.2中定义的基本I/O自动机$A_{1,2}$和$A_{2,1}$。其中*in-transit*变量是自动机$A_{1,2}$和$A_{2,1}$的状态组件。

引理22.1　对于使用通道$A_{1,2}$和$A_{2,1}$的*Stenning*协议，在它的每个可到达状态中，下列断言为真：

1）$tag_2 \leqslant tag_1 \leqslant tag_2+1$。

2）如果(m, k)在$in\text{-}transit_{1,2}$中，那么$k \leqslant tag_1$。

3）如果(m, tag_1)在$in\text{-}transit_{1,2}$中，那么m是$buffer_1$的第一个元素。

695　4）如果k在$in\text{-}transit_{2,1}$中，那么$k \leqslant tag_2$。

5）如果$tag_2=tag_1$，或者在$in\text{-}transit_{1,2}$或$in\text{-}transit_{2,1}$中存在一条标志值等于tag_1的消息，那么$buffer_1$非空。

证明　直接推导，留做一道习题。　□

我们的目标是要证明，使用任意合法的通道，*Stenning*协议都能够保证可靠的FIFO消息传递。首先介绍一个技术性的引理，该引理用通道规范说明的形式断言正确性，而没有采用任何合法通道。这一引理隐含任意合法通道的结果。

引理22.2的陈述使用稍微难懂一点的符号，但也不是真的很复杂。引理的条件是说对α映射，使节点进程P_1和P_2得到公平、且损耗式重排序通道规范所允许的运行。结论说α产生F的公平轨迹，也就是说它执行可靠的FIFO消息传递。

引理22.2　令α是*Stenning*协议的一个运行，且该协议有$A_{1,2}$和$A_{2,1}$。假设：

1）$\alpha|P_1$和$\alpha|P_2$是公平的。

2）$\alpha|A_{1,2}$和$\alpha|A_{2,1}$满足例14.1.2中的活性条件。

那么，$\alpha|ext(F) \in fairtraces(F)$。

证明概要　令$\beta=trace(\alpha)$。显然$\beta \in traces(F)$，也就是说，在β的*RECEIVE*事件中的高层消息序列是它的*SEND*事件中高层消息序列的前缀。证明方法是，先用一个从有$A_{1,2}$和$A_{2,1}$的*Stenning*协议到F的(单值的)模拟关系，然后调用定理8.13。用引理 22.1证明模拟关系。由于定理中的假设条件1和2只与活性有关，在这部分证明中甚至不需要使用这两条假设。我们把这个证明留做一道习题。

剩下的是要证明F的公平性条件，即提交给P_1的任何高层消息最终都能传输到P_2。(发送和传输事件的对应关系惟一由F的定义决定。)否则，假定m是第一个已经提交但未传输的高层消息，k表示m的标志。由于对P_2的*buffer*的增加，这个消息不能够被P_2接收，因为如果被接收，

696　P_2的公平性属性则意味着把这条消息传递给U_2。因此，tag_2总是$\leqslant k-1$。

我们声明消息m最终将到达$buffer_1$的前端。当$k=1$时，m是所发送的第一条消息，这个结论是显然的。如果$k \geqslant 1$，由于以前的消息已传递给U_2，那么也必定被P_2接收。这样，tag_2的值最终变成与$k-1$相等。而P_2的公平性属性意味着P_2不断重复发送$k-1$消息，$A_{2,1}$的弱丢失限制(WLL)条件则保证P_1能收到其中的一个消息。因此，$tag=k-1$的消息被从$buffer_1$中删除，m到达$buffer_1$的前端。

当消息m到达$buffer_1$的前端后，它将一直停留在这里(因为P_2没有接收它)。P_1的公平性意味着P_1将一直发送消息(m, k)，而$A_{1,2}$的WLL条件保证最终有一个(m, k)消息到达并被P_2接收。与假设矛盾。　□

用定理8.13和一个从*Stenning*到F的模拟，通过运行的对应关系也能证明这个公平性条件。这个证明留做一道习题。

引理22.2隐含*Stenning*协议的主要正确性结果，如下面的定理所述。这一定理说明，采用损耗式重排序通道的*Stenning*协议保证可靠FIFO消息传递。

定理22.3 采用任意的损耗式重排序通道(如例14.1.2中所定义)，*Stenning*协议实现F，即：对(该协议及其通道的)每个公平运行α，$\alpha|ext(F) \in fairtraces(F)$。

证明概要 根据引理22.2以及定理8.4和定理8.2中I/O自动机组合的基本属性，可得到本定理。具体证明留做一道习题。 □

22.3 位变换协议

*Stenning*协议的一个很有意思的变种是位变换协议(*Alternating Bit Protocol*)，简记为*ABP*。*ABP*的行为与*Stenning*协议非常相似，但*ABP*只使用{0,1}型值的标志，而不是整型值的标志。实际上，*ABP*可以看作是*Stenning*的一个优化版本，用低阶的位来替换其中的整型值标志。当然，这也就意味着*ABP*对低层的通道需要更严格的要求才能正常工作。 697

除了协议本身的意义外，多年来，*ABP*一直被作为演示多种协议证明技巧的标准例子之一。

ABP(非形式化)：

进程P_1将用户U_1提交的高层消息放入缓冲区$buffer_1$，并以交替方式为每个消息加上一个0或1的二进制值作为标志。P_1不断将它的缓冲区$buffer_1$中的第一条消息及其标志发送给P_2。进程P_2首先从它所收到的消息中接收第一个标志为1的消息。然后，P_2接收后续消息，并且每次所接收消息的标志与前一个被接收消息的标志不同。P_2把接收到的消息放入缓冲区$buffer_2$，并按照顺序将$buffer_2$中的消息传递给U_2。

P_2每接收一条高层消息，都将该消息的标志发回P_1作为确认。当P_1收到当前标志的确认后前移，开始处理下一条高层消息。

下面是进程P_1的代码。

ABP_1 automaton (formal):

Signature:

Input:
$SEND(m)_{1,2}$, $m \in M$
$receive(b)_{2,1}$, $b \in \{0,1\}$

Output:
$send(m,b)_{1,2}$, $m \in M$, $b \in \{0,1\}$

States:
buffer, a FIFO queue of elements of M, initially empty
$tag \in \{0,1\}$, initially 1.

Transitions:

$SEND(m)_{1,2}$
Effect:
add m to *buffer*

$send(m,b)_{1,2}$
Precondition:
if $b = tag$ then
m is first element of *buffer*
Effect:
remove first element of *buffer*

$receive(b)_{2,1}$
Effect:
if $b = tag$ then
remove first element (if any) of *buffer*
$tag := tag + 1 \bmod 2$

m is first on *buffer*
$b = tag$
Effect:
698 none
Tasks:
$\{send(m,b)_{1,2} : m \in M, b \in \{0,1\}\}$

现在给出进程P_2的代码。

ABP_2 automaton (formal):

Signature:

Input:
$receive(m,b)_{1,2}, m \in M, b \in \{0,1\}$

Output:
$RECEIVE(m)_{1,2}, m \in M$
$send(b)_{2,1}, b \in \{0,1\}$

States:
buffer, a FIFO queue of elements of M, initially empty
$tag \in \{0,1\}$, initially 0.

Transitions:

$RECEIVE(m)_{1,2}$
Precondition:
m is first on *buffer*
Effect:
remove first element of *buffer*

$send(b)_{2,1}$
Precondition:
$b = tag$
Effect:
none

$receive(m,b)_{1,2}$
Effect:
if $b \neq tag$ then
add m to *buffer*
$tag := tag + 1 \bmod 2$

Tasks:
$\{RECEIVE(m)_{1,2} : m \in M\}$
$\{send(b)_{2,1} : b \in \{0,1\}\}$

与我们在*Stenning*协议中假定的通道可靠性条件相比，*ABP*的要求更高：现在要假定通道不会对低层消息进行重排序，但仍然允许丢失和复制消息。这样，通道$C_{1,2}$和$C_{2,1}$是满足例14.1.1所给损耗式*FIFO*通道规范的I/O自动机(操作被重命名)。也就是说，它们允许消息的有限
699 丢失、有限复制，但不能有消息的重排序。与以前一样，用一个基本自动机和一组附加活性属性的组合描述所允许的通道行为。通过组合P_1、P_2、$C_{1,2}$和$C_{2,1}$，得到完整的*ABP*。

我们证明*ABP*正确性的策略是：用一个模拟关系，将*ABP*与*Stenning*协议联系起来。在这个模拟中，我们与损耗式FIFO通道结合来考虑*Stenning*进程，而不采用22.2节中考虑的更通用的损耗式重排序通道。

本节的余下部分，我们用$A_{1,2}$和$A_{2,1}$表示来源于例14.1.1、具有适合*ABP*外部接口的基本自动机。对于相同的自动机，如果提供的外部接口是适合于*Stenning*协议的，则用$A'_{1,2}$和$A'_{2,1}$表示。

最后，我们分别用P'_1和P'_2表示进程$Stenning_1$和$Stenning_2$，以区别于*ABP*的进程P_1和P_2。

这一模拟证明的关键是关于*Stenning*协议的一个新不变量，该协议使用通道$A'_{1,2}$和$A'_{2,1}$。

引理22.4 对于使用通道$A'_{1,2}$和$A'_{2,1}$的*Stenning*协议，在它的每个可到达状态中，下列断言为真：

记T为一整数序列，其组成依次是$queue_{2,1}$中标志（按照队列中从前到后的顺序）、tag_2、$queue_{1,2}$中元素的标志部分、tag_1。则T中的整数是非降序的，且T中第一个整数和最后一个整数的相差不超过1。

证明 该证明留做一道习题。 □

现在可以建立*ABP*与*Stenning*协议的联系。引理22.5告诉我们，对于采用损耗式FIFO通道*ABP*的每个运行α，存在一个*Stenning*协议的运行α'，该*Stenning*协议采用损耗式FIFO通道，使得α和α'在外部接口处看到的相同。

引理22.5 令α是*ABP*的一个运行，且该*ABP*有$A_{1,2}$和$A_{2,1}$。假设：

1）$\alpha|P_1$和$\alpha|P_2$是公平的。

2）$\alpha|A_{1,2}$和$\alpha|A_{2,1}$满足例14.1.1中的活性条件。

那么存在*Stenning*协议的一个运行α'，该*Stenning*协议采用$A'_{1,2}$和$A'_{2,1}$，使得

1）$\alpha|P'_1$和$\alpha|P'_2$是公平的。

2）$\alpha|A'_{1,2}$和$\alpha|A'_{2,1}$满足例14.1.1中的活性条件。 700

3）$\alpha|ext(F) = \alpha'|ext(F)$。

证明概要 首先创建一个从*ABP*到*Stenning*协议的模拟关系f，该*ABP*采用$A_{1,2}$和$A_{2,1}$，*Stenning*协议采用相应$A'_{1,2}$和$A'_{2,1}$。这一关系表达一个事实，即*ABP*中的二进制标志就直接是*Stenning*协议中整数标志的低端二进制位。具体来说，如果s和u分别是*ABP*和*Stenning*协议的状态，那么仅在下列情况下我们定义$(s, u) \in f$：

1）$s.buffer_1 = u.buffer_1$且$s.buffer_2 = u.buffer_2$

2）$s.tag_1 = u.tag_1 \bmod 2$且$s.tag_2 = u.tag_2 \bmod 2$

3）$s.queue_{1,2}$与$u.queue_{1,2}$拥有相同数量的元素。并且，对于任意的j，如果(m, k)是$u.queue_{1,2}$的第j个元素，那么$(m, k \bmod 2)$是$s.queue_{1,2}$的第j个元素。

4）$s.queue_{2,1}$与$u.queue_{2,1}$拥有相同数量的元素。并且，对于任意的j，如果k是$u.queue_{2,1}$的第j个元素，那么$(k \bmod 2)$是$s.queue_{2,1}$的第j个元素。

显然易证f是一个模拟关系。大部分我们需要证明的都是从f的定义以及*ABP*和*Stenning*的变换直接推导而来的。在证明模拟关系定义的条件2(步条件)时，引理22.4用于*receive*操作的证明。特别是，对于*ABP*接收消息的每个*receive*步，我们必须说明*Stenning*协议的相应*receive*步也将促使消息被接收。以*ABP*的一个步$(s, receive(m, b)_{1,2}, s')$为例，在这一步中$m$被$P_2$接收。促使$m$被接收的条件是$b \neq s.tag_2$。在*Stenning*协议的相应状态$u$中，模拟条件意味着所引入的低层消息带有一个与$u.tag_2$ modulo 2不同的标志k。但为了证明m在状态u中被接收，我们必须证明$k = tag_2 + 1$。引理22.4能够用来证明这一点。

然而，仅仅创建一个模拟关系并不足以证明活性条件。但实际上，它也表明f比通常的模拟关系更加严格：它将*ABP*的每个步映射到一个有相同类型操作的*Stenning*协议步。事实上，除*Stenning*操作包含一个整数k而相应*ABP*操作包含低阶二进制位b的情况外，这些操作都是相同的。

现在，将α固定为定理的假设。模拟f生成*Stenning*系统的一个“相应”运行α'。这种对应
701 关系保证下列两结论成立：

1）除上述的一个特例外，α和α'有相同的操作序列。

2）α和α'中同一个位置的两个状态由f关联。

这些条件已经足够让我们能够从α的条件推导出所需要的α'的条件。 □

请参考8.5.5节中关于这一运行对应关系的思想的另一个版本，以及10.9.4节和第16章中涉及运行对应关系的类似证明。

引理22.5隐含了下述关于*ABP*的技术性引理。该引理表明，对于通道行为是由损耗式FIFO通道的规范所认可的*ABP*协议，它的任何公平运行都展现可靠的FIFO消息传递。

引理22.6 令α是*ABP*协议的一个运行，且该协议有$A_{1,2}$和$A_{2,1}$。假设：

1）$\alpha|P_1$和$\alpha|P_2$是公平的。

2）$\alpha|A_{1,2}$和$\alpha|A_{2,1}$满足例14.1.1中的活性条件。

那么，$\alpha|ext(F) \in fairtraces(F)$。

证明概要 由引理22.5和引理22.2可证。 □

引理22.6则隐含下述定理中关于*ABP*的主要正确性结果。该定理所表达的是，采用损耗式FIFO通道的*ABP*保证可靠的FIFO消息传递。

定理22.7 采用任意的损耗式FIFO通道(如例14.1.1中所定义)，*ABP*实现F，即：对(该协议及其通道的)每个公平运行α，$\alpha|ext(F) \in fairtraces(F)$。

证明概要 根据引理22.6、以及定理8.4和定理8.2中组合自动机的基本属性，可得到本定理。具体证明留做一道习题。 □

无限复制 需要注意的是，采用那些允许无限复制的略微更通用些的通道，*ABP*仍然可正常工
702 作。这些通道也不对消息进行重排序，而WLL条件对消息丢失进行限制。这种通道与上述损耗式FIFO通道的惟一实质差别在于当所发送消息的数量有限时，新通道会对最后被发送的消息进行无限制的复制。我们之所以没有采用这种更加通用的通道来阐述引理22.5和引理22.6、定理22.7，是因为想在证明中使用不变量断言和模拟，描述这些更通用的通道时使用公理比使用自动机更容易。

22.4 可重排序的有界标志协议

至此，我们已经看到，使用带无界标志的*Stenning*协议，在低层消息有限丢失、有限复制和随机重排序的情况下，是可以获得可靠FIFO通信的。用*ABP*以及有界标志，可以容许有限丢失和有限复制，但不能对消息重排序。在本节，我们考虑是否可以设计容许低层消息重排序的有界标志协议。

首先考虑当*ABP*与能对低层消息重排序的通道一起使用时会出现什么问题：如果一个老的高层消息m恰好到达进程P_2、并且其标志与当前所期待的消息标志相同，那么P_2会错误地接收m。这种行为将导致同一个高层消息被重复传递给U_2，从而违反可靠通信的要求。

例如，图22-2中的发送/接收示意图描述的就是这样的一个运行，其中的P_2在接收完上一个消息m'后接收m的一个重复副本。

至此，我们看出*ABP*不能采用允许低层消息重排序的通道，但这并不意味着没有其他容许重排序的有界标志协议。

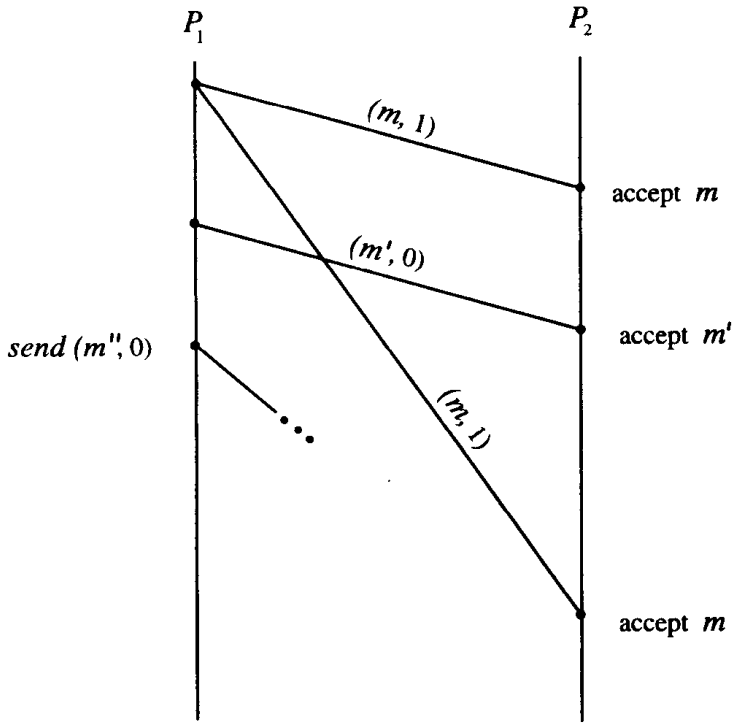


图22-2 P_2被骗而接收m

我们给出三个结果。首先，22.4.1节中将证明不存在同时允许重排序和复制的有界标志协议。接着在22.4.2节，给出一个容许丢失和重排序、但不容许复制的有界标志协议。不足之处是，该协议很复杂。最后，在22.4.3节中，证明不存在容许丢失和重排序的“高效”协议。这也意味着无法避免22.4.2节所给协议的高复杂性。

在本节中，我们假定高层消息字母表M和低层消息字母表M'都是有限的，以规范一个“有界标志”协议的概念。 703

22.4.1 关于重排序和复制的不可能性结论

本节将论述采用同时允许对低层消息进行重排序和复制的通道，没有协议能够解决可靠FIFO通信的问题。为了方便，我们的形式化陈述将基于合法通道行为的公理性说明规范。

我们需要某种通用的术语来描述进程自动机与通道轨迹属性间的交互。即，如果P_1和P_2组成一个协议，且$Q_{1,2}$和$Q_{2,1}$是关于P_1和P_2间两个通道的轨迹属性，当$\alpha|ext(Q_{1,2}) \in traces(Q_{1,2})$时我们称$P_1 \times P_2$的运行$\alpha$是与$Q_{1,2}$一致的。我们采用类似的方式定义与$Q_{2,1}$的一致性。同样，当$\alpha|ext(Q_{1,2})$是$traces(Q_{1,2})$中某个序列的有限前缀时，我们称$P_1 \times P_2$的运行$\alpha$与$Q_{1,2}$是有限一致的，关于$Q_{2,1}$的有限一致性定义与此类似。

例 22.4.1 一致性与I/O自动机

考虑一种特殊情况，其中$A_{1,2}$和$A_{2,1}$是具有合适通道外部接口的任意I/O自动机，$traces(Q_{1,2})$ 704
和$traces(Q_{2,1})$分别被定义为$A_{1,2}$和$A_{2,1}$的公平轨迹。

那么在$P_1 \times P_2$的公平轨迹中，与$Q_{1,2}$和$Q_{2,1}$一致的那部分轨迹恰好是组合$P_1 \times P_2 \times A_{1,2} \times A_{2,1}$的公平轨迹。同样，在$P_1 \times P_2$的有限运行中，与$Q_{1,2}$和$Q_{2,1}$有限一致的那部分运行恰好是组合$P_1 \times P_2 \times A_{1,2} \times A_{2,1}$的有限轨迹。

根据第8章中的合成性结论，包括定理8.1、定理8.3、定理8.4和定理8.6，可以证明这些事实。

在本小节的结论中，令$Q_{1,2}$为具有输入$send(m)_{1,2}$、输出$receive(m)_{1,2}$的轨迹属性，且它的全部轨迹都不包含消息丢失、只有有限的消息复制，其中$m \in M'$。这些轨迹中可以有随机重排序操作。(即，存在一个14.1.2节中的多到一的*cause*函数，而且每个*send*事件所对应的*receive*事件数是有限的。)令$Q_{2,1}$为相反通道方向的类似轨迹属性。当运行与$Q_{1,2}$和$Q_{2,1}$一致或有限一致时，简称该运行是*一致的或有限一致的*。

下面定理的意思是，采用允许消息重排序和复制的通道，任何有界标志协议都不能确保可靠的FIFO消息传递。

定理22.8　不存在采用重排序、复制通道$Q_{1,2}$和$Q_{2,1}$来实现F的有界标志协议(P_1, P_2)(即，如果α是$P_1 \times P_2$的一致公平运行，那么$\alpha|ext(F) \in fairtraces(F)$)。

证明　为了证明这样的一个实现(P_1, P_2)是矛盾的，我们构造一个有错误行为的运行。

首先，我们尽量运行该系统，直到P_1不能再发送任何具有新值的低层消息。方法是，我们构造一个$P_1 \times P_2$的有限一致运行α_1，使得当在α_1的任何有限一致扩展中发生某个$send(m)_{1,2}$事件时，α_1中也发生一个$send(m)_{1,2}$事件。通过连续扩展可以完成这一构造，在每个扩展中都发送一个新的低层消息，直到不能继续这一扩展为止；根据低层消息字母表M'的有限性可知，这一扩展过程一定会终止。假设在α_1中有n个(用户界面)*SEND*事件。

现在，令α_2为α_1的一个公平、一致的扩展，α_2中*SEND*事件的数量比α_1中*SEND*事件的数量
705 多1，正好有$n+1$个*SEND*事件。根据正确性条件，由U_1在α_2中提交的所有消息都将最终被传递给U_2，所以在α_2中恰好有$n+1$个*RECEIVE*事件。令α_3为α_2中包含最后一个*RECEIVE*事件的前缀。

现在构造一个具有下列属性的有限一致运行α_4：

1）α_4是α_1的扩展。

2）对P_1而言，α_4与α_1是无区别的。

3）对P_2而言，α_4与α_3是无区别的。

我们构造α_4的方法是，阻止紧跟α_1之后涉及P_1的全部事件、并允许P_2的全部事件像在α_3中一样继续执行。P_2的额外事件可包括*receive*事件、*send*事件、内部事件以及我们所需要的*RECEIVE*事件。在证明α_4是一个有限一致运行时，惟一困难在于*receive*事件：我们必须证明，尽管P_1未在α_1之后发送任何另外的低层消息，在α_1之后P_2能够接收像在α_3中一样的低层消息。但这种情况是可能发生的，因为在α_1之后，P_1在α_3中所发送的全部低层消息都包含P_1在α_1中已经发送的值。因此，P_2在α_1之后接收的任何低层消息都可以看作是在α_1中所发送某个低层消息的副本。

在α_4中，共有n个*SEND*事件和$n+1$个*RECEIVE*事件。为了完成这一矛盾的构造，我们在不引入任何新*SEND*事件的情况下，将α_4扩展成一个公平、一致的运行。最终运行中的*RECEIVE*事件比*SEND*事件多，与正确性条件矛盾。　□

因此，如果通道允许低层消息的有限复制和随机重排序，那么即使没有低层消息丢失，高层消息的可靠FIFO传输也是不可能的。

22.4.2　容许丢失和重排序的有界标志协议

尽管非常不明显，但事实证明有可能采用有界标志来容许消息的丢失和重排序(当然，不能容许消息的复制)。我们介绍一个实现这一目标的算法——*Probe*算法。*Probe*算法并不是一个
706 实用的算法；它只是一个反例算法，目的是为了说明不存在这一问题的不可能性证明。

***Probe*算法(非形式化)：**

与I/O自动机合成结合，这一算法常被简单地划分为两个层次。第1层用给定的通道实现中

间通道$I_{1,2}$和$I_{2,1}$，它们不对消息进行重排序，但可以丢失消息或对消息进行复制。更精确地讲，每个中间通道都满足一个14.1.2节所定义*cause*函数形式的公理性说明规范。这种情况下，要求*cause*函数不对消息进行重排序，但必需满足WLL丢失限制条件。无限复制是允许的。第2层用所得到的FIFO通道实现可靠FIFO通信。

第1层和第2层组合方式如图22-3所示，第2层协议所需要的每个通道分别由第1层协议的一个实例实现。把第2层协议的进程P_1与第1层中$I_{1,2}$实现的发送进程、第1层中$I_{2,1}$实现的接收进程组合起来，即得到整个算法的进程P_1。与此对称，把第2层协议的进程P_2与第1层中$I_{1,2}$实现的接收进程、第1层中$I_{2,1}$实现的发送进程组合起来，即得到整个算法的进程P_2。同样，为实现每个第1层实现的一个通道，必须对整个算法中的各个通道进行“复合”。 707

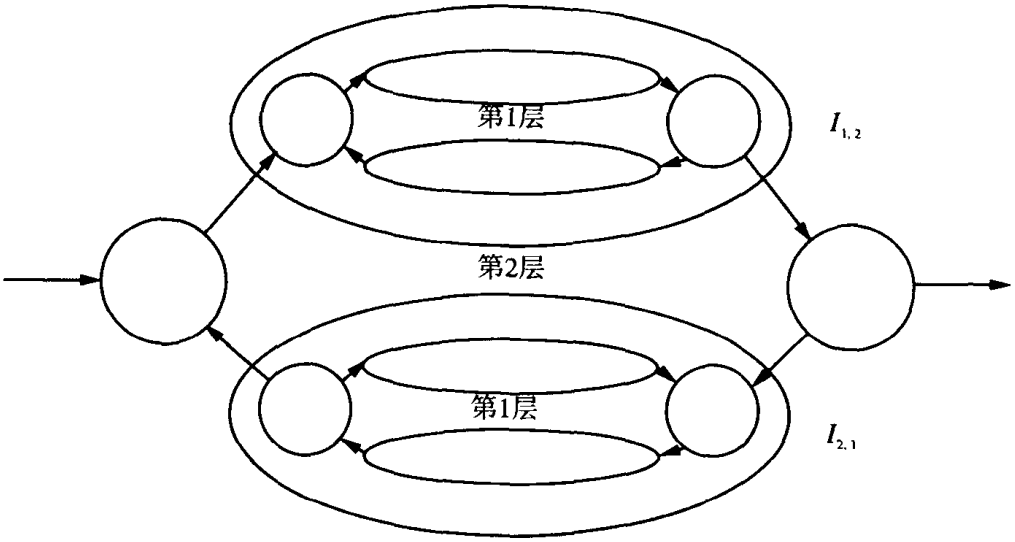


图22-3 *Probe*协议的分层结构

第2层是显而易见的——例如，可以用*ABP*。(注意，与我们在22.3节中考虑的损耗式FIFO通道相比，通道$I_{1,2}$和$I_{2,1}$要略微更通用些，具体表现在它们允许消息的无限复制；但是，22.3节末尾的注释表明采用这种更通用的通道，*ABP*仍然可以正常工作。)第1层涉及根据给定通道(它们可以丢失消息、对消息重排序，但不允许对消息进行复制)实现中间通道，比第2层要困难。

每个第1层的实现工作过程如下。仅当回应来自P_2的一个显式*probe*消息时，P_1才向P_2发送一个低层消息。P_1所发送的低层消息中通常包含它从U_1所接收的最近一个高层消息的值，P_1将这个高层消息的路径保留在*latest*中。因此，在这一协议中，P_1并不记住U_1所提交的所有消息，而只记住U_1最近一次提交的那个消息。(这样做的理由是，协议所要实现的中间通道被容许丢失部分高层消息。)为了保证P_1仅在回应*probe*消息时才发送消息，P_1中维持一个变量*unanswered*。P_1每接收一个*probe*消息时对该变量加1，每发送一个低层消息时对该变量减1。

P_2不断向P_1发送*probe*消息，并将它所发送*probe*消息的数量记录在*pending*中。同时，P_2在*count*(*m*)中统计自它最后一次向U_2传递高层消息以来(或者，如果还没有消息被传递给U_2，统计自运行开始以来)，所收到每个高层消息*m*的副本数。开始时，以及每次向U_2传递一个消息时，P_2将*pending*设为*old*。当*count*(*m*)超出*old*时，P_2可以输出*m*。

下面是第1层协议实现$I_{1,2}$的代码。当然，$I_{2,1}$的代码与该代码是对称的。在这一描述中，我们采用*M*代表第1层协议的高层消息字母表。

Probe Layer 1, process P_1:

Signature:

Input:
$SEND(m)_{1,2}$, $m \in M$
$receive(\text{“}probe\text{”})_{2,1}$

Output:
$send(m)_{1,2}$, $m \in M$

States:

$latest \in M \cup \{null\}$, initially $null$
$unanswered \in \mathbb{N}$, initially 0

708

Transitions:

$SEND(m)_{1,2}$
Effect:
$latest := m$

$receive(\text{“}probe\text{”})_{2,1}$
Effect:
$unanswered := unanswered + 1$

$send(m)_{1,2}$
Precondition:
$unanswered > 0$
$m = latest$
Effect:
$unanswered := unanswered - 1$

Tasks:

$\{send(m)_{1,2} : m \in M\}$

Probe Layer 1, process P_2:

Signature:

Input:
$receive(m)_{1,2}$, $m \in M$

Output:
$RECEIVE(m)_{1,2}$, $m \in M$
$send(\text{“}probe\text{”})_{2,1}$

States:

$pending \in \mathbb{N}$, initially 0
$old \in \mathbb{N}$, initially 0
for every $m \in M$:
$count(m) \in \mathbb{N}$, initially 0

Transitions:

$RECEIVE(m)_{1,2}$
Precondition:
$count(m) > old$
Effect:
for all $m' \in M$ do
$count(m') := 0$
$old := pending$

$send(\text{“}probe\text{”})_{2,1}$
Precondition:
$true$
Effect:
$pending := pending + 1$

$receive(m)_{1,2}$
Effect:
$count(m) := count(m) + 1$

Tasks:

$\{RECEIVE(m)_{1,2} : m \in M\}$
$\{send(\text{“}probe\text{”})_{2,1}\}$

709

第1层协议所使用的通道$C_{1,2}$和$C_{2,1}$不对消息进行复制，但可以对消息重排序、也可以丢失消息。也就是说，它们的规范说明是以*cause*函数的形式给出的，如14.1.2节所示。这种情况下，*cause*函数必须是一对一的，但不要求是单调的。然而，消息丢失是受WLL条件限制的。通道$C_{1,2}$和$C_{2,1}$是其公平轨迹满足这一说明规范的任意I/O自动机(具有合适的外部接口)。通过组合P_1、P_2以及这两个通道，即得到完整的第1层协议。

下面引理的意思是，采用给定的无复制通道，*Probe*协议的第1层实现中间通道$I_{1,2}$。

引理22.9 采用任意的无复制通道(如上述$C_{1,2}$和$C_{2,1}$所定义)，*Probe*协议的第1层实现中间通道$I_{1,2}$，即：对每个公平运行α，$\alpha|ext(I_{1,2}) \in traces(I_{1,2})$。

证明概要 首先证明$I_{1,2}$不会对消息进行重排序。为说明这一点，请注意P_2在执行第一个*RECEIVE*后，每执行一个*RECEIVE*时都要检查$count(m)>old$，其中m是被传递的高层消息。根据*old*变量的管理、再加上P_1只在回应*probe*消息时才发送消息和通道不复制消息的事实，可知在上一个*RECEIVE*事件处最多只有*old*个低层消息正处在从P_1到P_2的传递途中。因此，自上一个*RECEIVE*事件以来，至少有一个包含m的消息被P_1发送。这也就意味着m一定是$lastest_1$在上一个*RECEIVE*事件后某一点的值。从而表明没有发生重排序。

剩下的是要证明$I_{1,2}$保证WLL条件——当有无限数量的*SEND*事件时，其中一定有无限数量的*SEND*事件有相应*RECEIVE*事件。因此假定有无限数量的*SEND*事件。综合下列事实表明P_2执行了无限数量的*RECEIVE*事件：P_2一直在不断地发送*probe*消息、P_1一直在不断地回应所接收的*probe*消息、通道的活性假设、以及高层消息字母表M的有限性。但如在上一段所述，事实上在第一个消息之后，传递给U_2的任何消息一定是$latest_1$在上一个*RECEIVE*事件后某一点的值。这就足以表明*RECEIVE*事件一定对应无限个不同的*SEND*事件。 □

现在考虑完整的*Probe*协议。如前所述，每个进程是一个第2层进程和两个第1层进程的组合，如图22-3所示。每个通道都被“复合”，以实现这两个第1层协议的各自通道。 710

完整*Probe*协议所需要的通道与第1层实现所采用的通道$C_{1,2}$和$C_{2,1}$类似，都不允许对消息进行复制，但可以重排序和丢失消息。也就是说，通道的说明规范采用14.1.2节中*cause*函数的方式给出。像$C_{1,2}$和$C_{2,1}$中的一样，*cause*函数必须是一对一的，但不要求是单调的。

然后，事实表明我们需要一个比通道$C_{1,2}$和$C_{2,1}$中所采用的WLL条件更严格的丢失限制条件。确切地说，对完整*Probe*协议的通道所实现的两个通道中的任何一个而言，完整*Probe*协议的每个通道都必须满足WLL条件。我们采用一种更简单和更保守的策略——我们需要SLL条件。(这样确实能为所实现的两个通道分别保证SLL。) 关于通道多路复合技术参见习题14.7。

现在，通道已经是其公平轨迹满足这些新通道说明规范的(具有合适外部接口的)任意I/O自动机。通过组合P_1、P_2以及这两个通道，即得到完整的*Probe*协议。

下面的定理指出，采用给定的无复制SLL通道，完整的*Probe*协议确保可靠FIFO传递。

定理22.10 采用任意的无复制SLL通道(如上所述)，*Probe*协议实现可靠的FIFO通道F，即：对每个公平运行α，$\alpha|ext(F) \in fairtraces(F)$。

证明 这一结论是由第1层(参见引理22.9中的证明)和Layer 2的实现的正确性推导而来的。需要注意的是，根据每个给定通道的SLL条件，可得到它所实现的两个第1层通道的WLL条件。 □

复杂性分析 这里并不是试图对*Probe*协议进行一个正式规范的复杂性分析(也不对本章中其他协议进行正式规范的复杂性分析)。但必须注意，*Probe*协议有一个严重的复杂性问题：对越是后面的高层消息，完成传递需要的低层消息越多。更确切地讲，在第1层协议中，一旦k个低层

消息丢失，在传递每个后续高层消息时，即使再没有消息丢失，它将至少需要k+1个低层消息。
711 在下一小节，讨论是否可以避免这种开销。

22.4.3 不存在容许消息丢失和重排序的高效协议

我们刚描述了*Probe*协议，该协议采用容许消息丢失和重排序、但无消息复制的通道实现可靠FIFO通信。在这一节中，我们将证明任何实现这一功能的协议都需要与*Probe*协议类似的开销，在传递越是后面的高层消息时需要的低层消息越多。

像前面的关于不可能性结论的定理22.8一样，我们的陈述以定义合法通道行为的公理性轨迹属性描述为基础。我们采用22.4.1节中介绍的通用术语来描述进程自动机与通道轨迹属性间的交互，特别是与两个通道接口的轨迹属性均相一致的$P_1 \times P_2$的运行的定义、以及与某个轨迹属性有限一致的$P_1 \times P_2$的有限运行α的定义。

本节中，令$Q_{1,2}$为输入为$send(m)_{1,2}$、输出为$receive(m)_{1,2}$的轨迹属性，其中$m \in M'$，且它的全部轨迹都不包含消息复制、而消息丢失受SLL条件的限制。这些轨迹中可以有随机重排序操作。令$Q_{2,1}$为相反通道方向的类似轨迹属性。我们将不明确提到通道轨迹属性，只说运行是一致的或有限一致的。

采用任何满足这些说明规范的通道时，*Probe*协议就能够发挥作用(例如，实现可靠FIFO通信)。我们将证明，从传递稍后的高层消息所需要低层消息的数量这个角度来说，任何达到这一目的的协议都需要高昂的代价。为此，我们需要一个关于开销的精确定义。

首先，对于$P_1 \times P_2$的一个有限一致运行α，当其中的*SEND*事件与*RECEIVE*事件数量相等时，我们称α是完整的。这也就是说，该协议将U_1所提交的全部高层消息都成功地传递给U_2。

下面定义所表达的基本思想是，为了成功传递每个高层消息，在最好情况下，该协议只需要发送一个有限数量的低层消息。如果α是一个完整的运行、$k \in \mathbb{N}^+$、$m \in M$，那么当下列条件成立时我们称扩展α'是α的k扩展：

1）在α'中位于α之后的部分，只有$SEND(m)_{1,2}$与$RECEIVE(m)_{1,2}$两个用户接口事件。(这意味着在α'中位于α之后的部分，用户U_1只发送一个高层消息m，并且该消息也成功地传递
712 到U_2。这一条件表明α'也是一个完整运行。)

2）在α'中位于α之后部分，所有由P_2接收的低层消息都是在α之后发送的。(即，没有老的高层消息被接收。)

3）在α'中位于α之后的部分，$receive_{1,2}$事件的数量小于或等于k。

对于一个协议，如果任取完整运行α和$m \in M$，对m都存在α的一个k扩展，则该协议是k-消息限制的(*k-message-bounded*)。对于某个$k \in \mathbb{N}^+$，如果一个协议是k-消息限制的，则该协议是消息限制的(*message-bounded*)。

因此，一个消息限制的协议在其开销方面仅满足一个非常小的要求：在预期没有进一步低层消息丢失的最佳情况下，传递一个高层消息所需要低层消息的数量不能无限制增长。但是，尽管这一要求很弱，我们仍然可以证明：没有任何消息限制的协议能够采用允许消息丢失和复制的通道实现可靠FIFO通信。

定理22.11　不存在消息限制的有界标志协议(P_1, P_2)，使得(P_1, P_2)采用损耗式重排序通道$Q_{1,2}$和$Q_{2,1}$实现F(意思是，如果α是$P_1 \times P_2$的一个一致公平运行，那么$\alpha|ext(F) \in fairtraces(F)$)。

证明　用反证的方法，假定有这样的一个协议(P_1, P_2)，并取一个k，使得(P_1, P_2)是k-消息限制的。

假设我们能够生成一个M'元素的多集T、一个$P_1 \times P_2$的完整运行α、以及α的一个k扩展α'，

它们同时满足下列两个条件：

1）运行α后，T中的全部消息都处在从P_1到P_2的“传递途中”(例如，它们已经被发送出来但还未被接收)。[⊖]

2）由P_2在α'的位于α之后部分所接收的低层消息多集是T的一个子多集。

在这种情况下，就能够推导出一个悖论，方法如下。采用类似于定理22.8证明中的构造方法，构造另外一个有限一致运行α_1，使得下列结论成立：

1）α_1是α的一个扩展。

2）对P_1而言，α_1与α是不可区分的。

3）对P_2而言，α_1与α'是不可区分的。 713

我们的做法是，阻止紧跟α之后涉及P_1的所有事件，同时允许P_2的所有事件像在α'中一样继续执行。我们这样做是可行的，因为P_2的额外*receive*事件可以由α之后、正处在从P_1到P_2途中的低层消息产生。通过生成一个*RECEIVE*事件比*SEND*事件多的公平、一致运行，就得到一个如同定理22.8的证明中一样的悖论。

这样，就具备产生这种悖论的充分条件。下面的这个重要断言指出，如果一个低层消息多集T正处在从P_1到P_2的传递途中，那么这一悖论或者已经存在、或者我们能够将T扩大成一个更大的多集T'。

声明22.12 假设α是一个完整运行、T是一个位于α之后处于从P_1到P_2传递途中的低层消息多集，其中T包含任何元素的最多k个副本。那么下列两条件中至少有一个是成立的：

1）存在α的一个k扩展α'(对某个m)，使得P_2在α'中位于α之后所接收的低层消息多集是T的一个子多集。

2）存在α的一个完整运行α'以及一个在α'之后处于从P_1到P_2传递途中的新低层消息多集T'，其中T'包含任何元素的至多k个副本、$T \subset T'$。[⊖]

首先假定声明22.12为真。证明在这种情况下能够构造上述的悖论，我们已经说明这一悖论可充分证明该定理。为此，定义两个序列：一个完整运行的序列α_0、α_1、…；一个低层消息的多集序列T_0、T_1、…，其中每个多集最多包含任何元素的k个副本。每个α_i都是前一个完整运行的扩展，并且对每个i有$T_i \subset T_{i+1}$。而且，对每个i，在α_i之后多集T_i处于从P_1到P_2的传递途中。

我们从α_0和T_0开始，其中α_0由P_1和P_2的初始状态组成，T_0为空。如果声明22.12中的条件1为真，我们就已经生成悖论，证明工作完成。否则，声明22.12中的条件2必然成立。在这种情况下，令$\alpha_1=\alpha'$、$T_1=T'$。通常，如果在α_i和T_i情况下条件1成立，证明工作完成；否则我们可以根据条件2定义α_{i+1}和T_{i+1}。

现在证明条件1最终一定成立。因为如果不是这样，那么对每个i都有条件2成立，我们定义两个无限序列。具体讲，我们得到一个无限链$T_0 \subset T_1 \subset T_2 \subset \cdots$。但由于根据每个$T_i$的定义，$T_i$中最多有每个$M'$元素的$k$个副本，因此该链条中多集的数量不可能多余$k|M'|+1$。由于$|M'|$是有限的，因此该链条中多集的数量也是有限的。因此条件1最终一定成立。 714

剩下的只要证明声明22.12。

(声明22.12的)证明 任取$m \in M$，并为消息m获得α的一个k扩展α'；由于假定该协议是k-消息限制的，因此能够得到这一k扩展。如果P_2在α'的位于α之后部分所接收的低层消息多集是多集T

⊖ 已经被发送但还未被接收的消息多集是由运行α惟一决定的。

⊖ 这就是说，T是T'的一个真子多集，即至少有一个M'元素，在T'中至少比T中多一个它的副本。

的子集，那么条件1成立，证明结束。因此假定这种情况不成立。这样就存在$p \in M'$，使得α'中位于α之后的$receive(p)_{1,2}$事件的数量多于T中p副本的数量。令$T'=T \cup \{p\}$(用多集的并操作)。由于α'是一个k扩展，因此这一新$receive(p)_{1,2}$事件的数量不超过k，这就说明在T'中每个元素的副本数量最多为k。这样，我们就得到α的一个完整运行，该运行使得T'在传递途中。

我们知道，在α'的位于α之后部分至少有一个$send(p)_{1,2}$事件，因为P_2在α'之后所接收的全部低层消息都被假定是在α'之后发送的。令α_1是α'的前缀，并以第一个这样的$send(p)_{1,2}$事件为结尾；那么α_1是α的一个有限一致扩展。因此，在α_1之后低层消息多集T处于传递途中。如果α_1同时包括新$SEND(m)_{1,2}$事件与$RECEIVE(m)_{1,2}$事件、或者既不包括$SEND(m)_{1,2}$事件也不包括$RECEIVE(m)_{1,2}$事件，那么α_1是完整的，因而满足条件2。

剩下的情形是α_1只包括$SEND(m)_{1,2}$事件，而不包括$RECEIVE(m)_{1,2}$事件。这种情况下，我们将α_1扩展成一个有限一致的α_2，α_2包含一个附加的$RECEIVE(m)_{1,2}$事件，但其中没有任何来自T'的低层消息被P_2所接收。基于下列事实，我们是能够构造这样的一个α_2的：$P_1 \times P_2$的任意一个有限一致运行都能够扩展成一个公平一致运行，使得无新的$SEND_{1,2}$事件发生、且所有新$receive_{1,2}$事件都是由新$send_{1,2}$事件引起的。将这一事实用于α_1，我们就得到α_1的一个公平一致扩展α_3，根据正确性条件限制可知α_3必定包含一个与最后$SEND(m)_{1,2}$所对应的$RECEIVE(m)_{1,2}$事件。我们所需要的完整运行α_2就是α_3中以该$RECEIVE$事件结尾的前缀。 □

至此，完成定理22.11的证明。 □

22.5 容许进程崩溃

至此为止，本章中所给出的结果几乎解决了采用不可靠通道实现可靠FIFO通信所涉及的每
715 个问题，至少在假定进程是可靠的情况下解决了这种实现所涉及的各个问题。当只有两个节点时，考虑进程的停止故障或Byzantine故障的意义不大。然而，分析在进程崩溃以及后来恢复崩溃进程时所发生的事情是有作用的。当一个进程崩溃可简单地归结为停止故障、而随后的恢复操作也仅涉及从进程停止处的简单复原时，那么一个崩溃后又恢复的进程与一个中止了一段时间的正确进程并没有什么差别。但是，如果一个进程崩溃涉及状态信息的部分或全部丢失，则会引起新的问题。

在这一节，将在进程会出现崩溃、丢失信息、而后又恢复执行的情况下，考虑可靠FIFO通信问题。这种设置下的进程所模拟的是那种有易失性存储器、或同时有易失性存储器和稳定存储器的物理处理器。这样的一个处理器崩溃时，易失性存储器的全部内容将被丢失。进程恢复时，从稳定存储器的前一个状态继续执行，而对于易失性存储器则使用缺省状态。当一个处理器恢复时，通常所做的第一件事情是运行一个恢复协议，使用稳定存储器中的信息把易失性存储器恢复到某个合法的状态。在正式的模型中，我们将整个恢复协议看作是单个的恢复步。

在22.5.1节中，将阐述在有进程崩溃的情况下，实现可靠FIFO通信的不可能性，而这种可靠FIFO通信在可靠进程的情况下是能够实现的。这样就促使我们弱化新的环境下 的问题需求。在22.5.2节给出另一个不可能性结论，该结论适用于更加弱化的问题陈述。最后在22.5.3节中，介绍一个同时容许进程崩溃和不可靠通道的实用算法。

在本节中，我们假定进程P_i新增加输入操作$CRASH_i$和输出操作$RECOVER_i$，并考虑由后者组成一个新的任务。一个$CRASH_i$事件激发一个相应的$RECOVER_i$，同时暂停所有其他的局部控制操作直至一个$RECOVER_i$发生。这就要求在P_i的任何一个公平运行中，最终必须发生这样的一个$RECOVER_i$事件。我们假定，在$CRASH_i$事件及其之后的那个$RECOVER_i$之间，所发生的

任何事件(包括其他的$CRASH_i$事件)对状态不产生任何影响。

新的接口如图22-4所示。

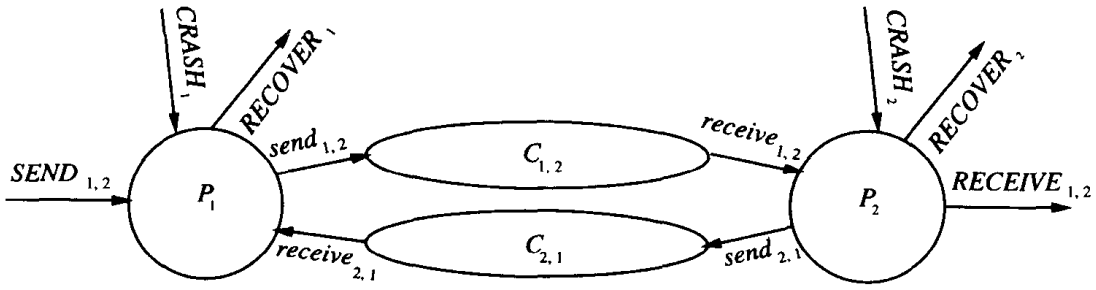


图22-4 容许崩溃的进程的外部接口

22.5.1 简单的不可能性结论

考虑$RECOVER_i$操作将进程P_i的状态设置回随机初始状态的情况。因此，这种情况下，$CRASH_i$以及随后的$RECOVER_i$将使得进程状态的全部信息都丢失。在这一模型中，显然无法解决可靠FIFO通信问题，即使支撑通道本身是可靠FIFO通道！ 716

定理22.13 在给定的崩溃故障模型中，不存在使用可靠FIFO通道实现F的协议(意思是，对任何一个公平运行α，都有$\alpha|ext(F) \in fairtraces(F)$)。

证明 证明的策略是获得这样一个协议的一个反例。基本思想是：在P_2的一个崩溃事件之后，该协议无法判断一个高层消息是否已经被传递给U_2。

令α_1为协议的任意一个公平运行，其中只有一个*SEND*事件而无*CRASH*事件发生。那么根据正确性可知，在*SEND*事件之后有一个相应的*RECEIVE*事件。令α_2和α_2'分别是α_1的两个前缀，α_2恰好在该*RECEIVE*事件之前结束，α_2'恰好在该*RECEIVE*事件之后结束。

同时令α_3为α_2的一个扩展，其中只有一个$CRASH_2$事件，相应的$RECOVER_2$事件紧随$CRASH_2$之后。那么α_3可扩展成一个不包含任何*SEND*事件或崩溃的公平运行α_4。由于α_4也必须满足正确性条件，因此α_4中必定包含一个与其中*SEND*事件对应的*RECEIVE*事件，且该*RECEIVE*事件要发生在*CRASH*事件和*RECOVER*事件之后。

现在构造另一个能够替换α_4的公平运行α_5。运行α_5以α_2'开始，接着是$CRASH_2$和$RECOVER_2$，最后以α_4中*RECOVER*之后部分结束。那么α_5也是一个公平运行。但α_5中包含两个*RECEIVE*事件而仅有一个*SEND*事件，从而与正确性条件矛盾。 □

需要指出的是，在一个*CRASH*和*RECOVER*事件总是连续发生、且仅有有限个崩溃事件发生的更强模型中，仍然可用上述的证明方法证明定理22.13。 717

22.5.2 更复杂的不可能性结论

定理 22.13表明，我们前面所使用的问题陈述对容许进程崩溃的环境而言太强硬。合理的做法是尽量弱化这一问题陈述，以得到一个在这一环境中也能够解决的协议问题版本。不幸的是，事实证明即使对问题陈述进行大量弱化，这一问题仍然无法解决。本节中，我们针对该问题的一个更加弱化版本，介绍一个不可能性结论。(当然，证明工作也更加困难。)

我们使用的崩溃模型与22.5.1节中的相同——具体说，当一个进程崩溃时，会丢失所有的状态信息。

我们减少对外部接口的要求，以弱化问题陈述。即，所要实现的通道不允许消息复制，但可以对消息重排序。至于消息丢失，我们现在只要求如果一个消息的*SEND*事件之后没有跟随

*RECOVER*事件，则该消息一定被传递。这就是说，对任何消息，当其*SEND*事件在某个*RECOVER*事件之前时，我们容许该消息的丢失。因此，当有无限数量的*CRASH*和*RECOVER*事件时，根本就没有任何消息被要求传递。但是，如果只有有限数量的这种事件，在最后一个*RECOVER*事件之后发送的任何消息都必须被传递。我们用B来表示这一说明规范(正式的说法是，轨迹属性)。

我们弱化问题陈述的另一途径是，对实现中采用的通道做更多的假设。所增加的具体假设包括：不允许消息复制或者重排序。通道所能做的全部事情只是丢失消息，并且这种丢失受SLL条件限制。我们分别用$Q_{1,2}$和$Q_{2,1}$来表示这两个通道的说明规范。我们现在说一个协议的某个运行是一致的或有限一致的时，意思是对通道$Q_{1,2}$和$Q_{2,1}$来说它都有这些属性。

对通道规范说明$Q_{1,2}$和$Q_{2,1}$中的任意一个而言，都可以说在一个有限一致运行的某个点，一个消息序列T是“在途的”。这句话的意思是，T是某个已经发送消息序列的子序列，且T是在(为某个*cause*函数所传递的)最后一个被传递的消息之后发送的。根据这一定义可得到的一个结论是，即使没有新的*send*事件，任何在途消息序列T都可能是下一个被通道传递的消息序列。

下面是我们在本节中阐述的不可能性结论。该结论表明，使用损耗式低层通道，任何协议都不能保证无消息复制、在全部进程崩溃和恢复都已停止后无消息丢失的通信。

718 **定理22.14**　在给定的崩溃故障模型中，不存在使用损耗式通道$Q_{1,2}$和$Q_{2,1}$实现B的协议(P_1, P_2)(意思是，如果α是$P_1 \times P_2$的一个公平一致运行，那么$\alpha|ext(B) \in traces(B)$)。

在这一定理的证明中，我们用符号 $\bar{i}$ 表示与进程i相对的进程，即$\bar{1}=2$，$\bar{2}=1$。同样，如果α是$P_1 \times P_2$的某个有限一致运行且$i \in \{1, 2\}$，则我们定义

- $in(\alpha, i)$为P_i在α期间所接收的低层消息序列
- $out(\alpha, i)$为P_i在α期间所发送的低层消息序列
- $state(\alpha, i)$为P_i在α之后的状态

证明　假定存在这样的一个协议，以得到一个矛盾。这一证明的关键是下述断言。该断言阐述，对任意一个无崩溃的有限一致运行α，都能够使用进程崩溃产生一种情形，使得两个进程的状态都与它们在α结束时的状态相同，而在其中一个通道中有一个在途消息序列，该序列中包含α中沿着该通道所发送的全部低层消息。

声明22.15　令α为某个无崩溃有限一致运行。令$i \in \{1, 2\}$。假设α不包含任何步或者α的最后一个步是P_i的一个步。那么存在$P_1 \times P_2$的一个有限一致运行α'，在α'结束时下列结论成立：

1）P_1的状态为$state(\alpha, i)$

2）$P_{\bar{i}}$的状态为$state(\alpha, \bar{i})$

3）序列$out(\alpha, i)$处在从P_i到$P_{\bar{i}}$的传递途中

运行α'可以包含*CRASH*和*RECOVER*事件，但不存在未匹配的*CRASH*事件——即每个*CRASH*都有一个后续的相应*RECOVER*。

(声明22.15的)证明　证明的方法是归纳α中步的数量。

基础：0步。那么$\alpha'=\alpha$，证明结束。

归纳步：k步，$k>0$。

如果α不包含$P_{\bar{i}}$的步，那么$\alpha'=\alpha$，证明结束。因此假设α至少包含$P_{\bar{i}}$的一个步。那么令α_1为α中以$P_{\bar{i}}$的一个步结尾的最长前缀。显然α_1是α的严格意义上的前缀，因为我们已经假定α的最后一
719 步是P_i的一个步。显然$state(\alpha, \bar{i}) = state(\alpha_1, \bar{i})$。同样，$in(\alpha, i)$是$out(\alpha_1, \bar{i})$的子序列。

那么根据归纳假设，存在一个有限一致运行α_1'，在α_1'结束时下列结论成立：

1）P_i的状态为$state(\alpha_1, i)$

2）$P_{\bar{i}}$的状态为$state(\alpha_1, \bar{i})$

3）序列$out(\alpha_1, \bar{i})$处在从$P_{\bar{i}}$到P_i的传递途中

并且，α_1'中不包含任何未匹配的*CRASH*事件。由于$in(\alpha, i)$是$out(\alpha_1, \bar{i})$的子序列，因此在α_1'之后$in(\alpha, i)$也处在从$P_{\bar{i}}$到P_i的传递途中。

现在我们构造证明需要的运行α'。运行α'以α_1'开始。α'的其余部分只涉及P_i（因为$P_{\bar{i}}$已经处于证明需要的最终状态）。首先，发生一个$CRASH_i$和一个$RECOVER_i$，使得P_i回到它在α中的初始状态。接着P_i像在α中一样开始自己的运行，必要时从流入通道提取低层消息。因为在α_1'之后$in(\alpha, i)$处在从$P_{\bar{i}}$到P_i的传递途中，所以P_i提取消息是可能的。从而使得P_i进入状态$state(\alpha, i)$，并把所需要的、位于$out(\alpha, i)$中的低层消息放入流出通道。同样，α'中不包含未匹配的*CRASH*事件。

图22-5所示的是P_i在α_1'之后运行时的系统状态变化示意图。 □

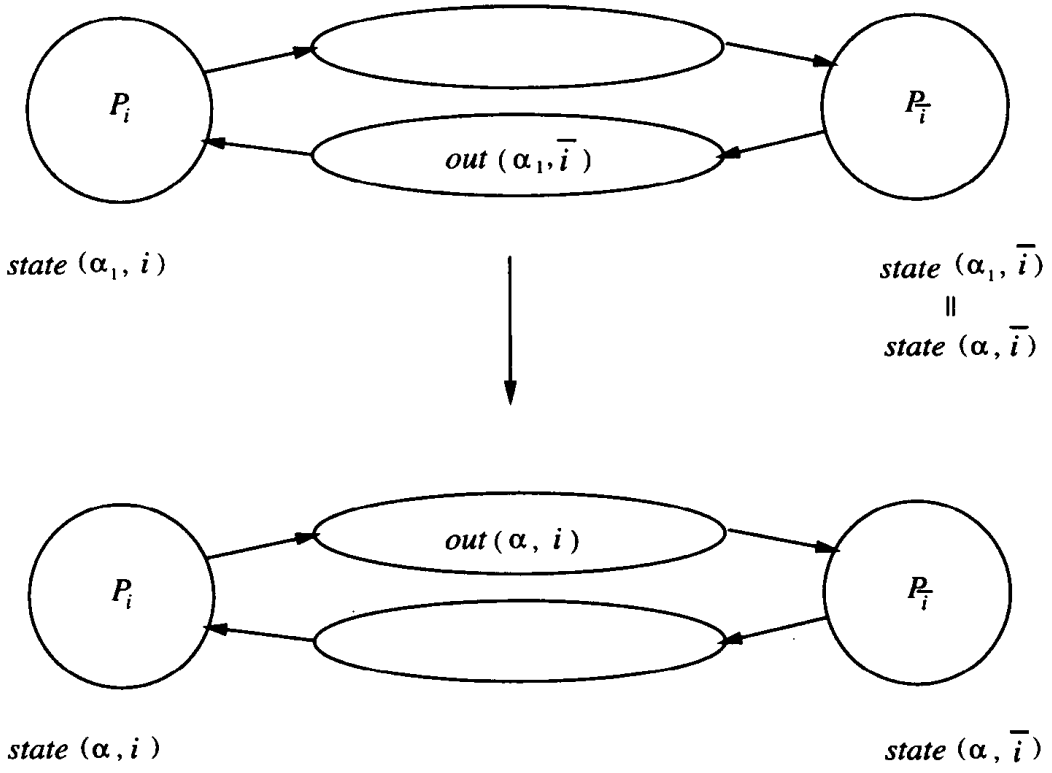


图22-5 从α_1'到α'的系统状态变化

现在我们用声明 22.15来完成定理22.14的证明。令α为某个无崩溃的有限一致运行，且α只包含一个*SEND*事件及相应的*RECEIVE*事件，同时为了保持一般性，假定α以该*RECEIVE*事件结束。

我们构造一个运行α_1，其进程最终状态与α的进程最终状态相同，但α_1包含一个作为其最后外部接口事件的*SEND*。换句更精确的话说，就是没有后续的*SEND*、*RECEIVE*、*CRASH*或*RECOVER*事件，也没有未匹配的*CRASH*事件。首先，依据声明22.15派生一个有限一致运行α'，α'以分别等价于$state(\alpha, 1)$和$state(\alpha, 2)$的进程状态结束，且$out(\alpha, 2)$处在从P_2到P_1的传递途

中，同时α'中没有未匹配的*CRASH*事件。接着构造α_1，先采用声明22.15中归纳步的方法扩展α'，然后对P_1执行进程崩溃和恢复，接着像在α中一样运行P_1。(同样，所需要的输入序列$in(\alpha, 1)$正在流入通道中处于在途状态。) 这样就使得P_1再次到达$state(\alpha, 1)$。同时，在α_1的位于α'之后部分，有一个*SEND*步，但没有其他用户接口步。从而得到预期的α_1属性。

至此可以得到一个矛盾。令α_2为α_1的一个扩展，是一个未增加任何新*SEND*、*CRASH*或
720 *RECOVER*事件的公平一致运行，其中每个在α_1之后接收的低层消息都是在α_1之后发送的。(也就是说，所有老的低层消息都丢失。) 根据正确性要求，在后缀中至少有一个*RECEIVE*事件与α_1中的最后*SEND*事件对应。同时，α_2中位于α_1之后部分也可以添加于α之后，同样得到一个公平一致运行；这是因为在α和α_1之后这两个进程都处在相同的状态，并且所有老的低层消息都丢失。但这就违背了正确性要求，因为α中已经有相同数量的*SEND*和*RECEIVE*事件、而该后缀中还包含至少一个*RECEIVE*而没有*SEND*。 □

定理22.14表明，如果要允许丢失全部状态信息的进程崩溃，无论如何弱化容许崩溃的条件，都不可能解决可靠FIFO的消息传递问题。

22.5.3 实用的协议

尽管存在上两小节中给出的不可能性结论，在进程崩溃情况下能够确保某种可靠FIFO消息
721 传递的消息传递协议在实际中仍是非常重要的。在这一节，介绍一个重要的协议——*FivePacketHandshake*协议。该协议是建立网络连接的标准方法，被TCP、ISO TP-4以及许多其他重要传输协议采用。在本小节中，使用“包”作为“低层消息”的同义词。

*FivePacketHandshake*协议满足22.5.2节中的正确性说明规范*B*，该规范不允许消息复制、并要求在最后一个*RECOVER*之后提交的消息不被丢失。事实上，该协议还不对消息重排序。它不但容许进程崩溃，还容许范围广泛的通道故障。这一属性与定理22.14的不可能性结论之所以不矛盾，其原因在于*FivePacketHandshake*协议依赖于系统为消息提供惟一标识符(UID)的能力，这种标识符可看作是(以及模型化为)某种形式的稳定存储器。

为什么能用稳定存储器来模型化UID呢？UID的关键属性是任何UID最多只会被生成一次，即使发生强制性崩溃也是如此。在形式化模型中，我们可抽象地将这一属性表示成允许该协议即使在一个进程崩溃之后也能记录前面已经产生的每个UID，并检查这一记录以确保不会第二次生成同一个UID。换句更确切的话说，我们可以在协议状态中保留一个组件*used*，用来保存所生成的每个UID。当协议选择一个新UID时，挑选一个不在*used*集合中的UID。我们假定*used*集合不会由于进程崩溃而被破坏，即它存储在稳定存储器中。

在实际中，有许多产生UID的不同方法——例如，使用随机数发生器或者实时时钟。但是，事实证明以在稳定存储器中保留已使用UID的方式来模型化这些技巧既简单又易懂。

我们允许所使用的通道丢失、复制以及重排序消息。但是，我们只允许有限复制，且消息丢失也受SLL条件限制。

***FivePacketHandshake*协议(非形式化)：**

像在*Stenning*协议和*ABP*协议中一样，P_1维护一个由U_1所提交高层消息的缓冲区，并不断将这些消息传递给P_2，每次传递一个消息。

对于P_1试图发送的每个消息，在P_1和P_2之间有一个初始的双向包(低层消息)交换，以建立一个共同接受的消息标识符。在这一交换过程中，首先由P_1在(“*needuid*”，v)包中把一个新的UID v发送给P_2。P_2将这个UID v与另一个新UID u组成二元组(u, v)，并在(“*accept*”，u，v)包
722 中把二元组(u, v)发回给P_1。P_1可识别出这个包是最新的，因为其中包含P_1的最后UID v。然后P_1

选择u作为它所试图发送高层消息的UID。

现在P_1向P_2发送最近的高层消息m，把m和u组成一组作为(“*send*”，m，u)包，一同发送给P_2。P_2可识别出这个包是最新的，因为其中包含P_2的最后UID u。接收一个消息之后，P_2发送一个(“*ack*”，u)形式的确认包。

第5个包的形式为(“*cleanup*”，u)，由P_1用来通知P_2何时丢弃当前的UID。

*FivePacketHandshake*协议的5个包如图22-6所示。

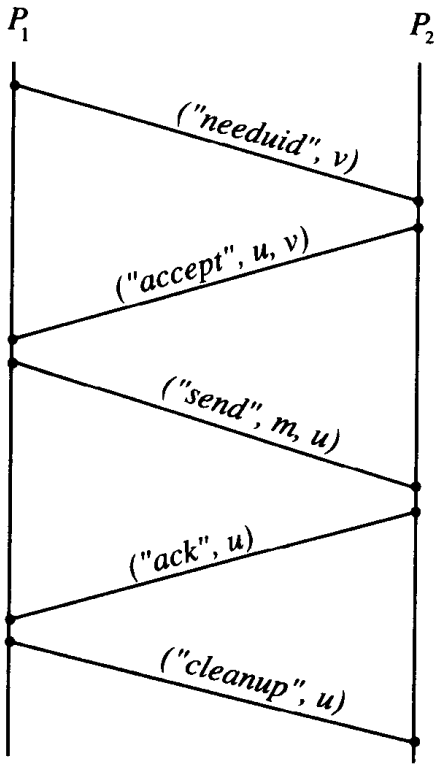


图22-6 *FivePacketHandshake*协议的五个包

协议代码如下。为了方便，在每个进程状态中都包括一个组件*used*，以记录已经被该进程生成的全部UID。*used*组件是惟一不受进程崩溃破坏的组件。 723

***FivePacketHandshake*, process P_1:**

Signature:

Input:

$SEND(m)_{1,2}$, $m \in M$

$receive(p)_{2,1}$, $p \in \{(\text{“}accept\text{”}, u, v) : u, v \text{ UIDs}\} \cup \{(\text{“}ack\text{”}, u) : u \text{ a UID}\}$

$CRASH_1$

Output:

$send(p)_{1,2}$, $p \in \{(\text{“}needuid\text{”}, v) : v \text{ a UID}\} \cup \{(\text{“}send\text{”}, m, u) : m \in M, u \text{ a UID}\}$
$\cup \{(\text{“}cleanup\text{”}, u) : u \text{ a UID}\}$

$RECOVER_1$

Internal:

$choose(v)_i$, v a UID

States:

$status \in \{idle, needuid, send, crashed\}$, initially $idle$

buffer, a FIFO queue of M, initially empty
uid-v, a UID or *null*, initially *null*
uid-u, a UID or *null*, initially *null*
used, a set of UIDs, initially empty
send-buffer, a FIFO queue of packets, initially empty

Transitions:

$SEND(m)_{1,2}$
Effect:
if $status \neq crashed$ then
add m to *buffer*

$choose(v)_1$
Precondition:
$status = idle$
buffer is nonempty
$v \notin used$
Effect:
$uid\text{-}v := v$
$used := used \cup \{v\}$
$status := needuid$

$send(\text{"needuid"}, v)_{1,2}$
Precondition:
$status = needuid$
$v = uid\text{-}v$
Effect:
none

$receive(\text{"ack"}, u)_{2,1}$
Effect:
if $status \neq crashed$ then
if $status = send$
and $u = uid\text{-}u$ then
remove first element of *buffer*
$uid\text{-}v := null$
$uid\text{-}u := null$
$status := idle$
add $(\text{"cleanup"}, u)$ to *send-buffer*

$send(\text{"cleanup"}, u)_{1,2}$
Precondition:
$status \neq crashed$
$(\text{"cleanup"}, u)$ is first on *send-buffer*
Effect:
remove first element of *send-buffer*

$receive(\text{"accept"}, u, v)_{2,1}$
Effect:
if $status \neq crashed$ then
if $status = needuid$
and $uid\text{-}v = v$ then
$uid\text{-}u := u$
$status := send$
else if $uid\text{-}u \neq u$ then
add $(\text{"cleanup"}, u)$ to *send-buffer*

$send(\text{"send"}, m, u)_{1,2}$
Precondition:
$status = send$
m is first on *buffer*
$u = uid\text{-}u$
Effect:
none

$CRASH_1$
Effect:
$status := crashed$

$RECOVER_1$
Precondition:
$status = crashed$
Effect:
buffer := empty sequence
$uid\text{-}v := null$
$uid\text{-}u := null$
send-buffer := empty sequence
$status := idle$

Tasks:
$\{send(\text{"needuid"}, v)_{1,2} : v \text{ a UID}\}$
$\{send(\text{"send"}, m, u)_{1,2} : m \in M, u \text{ a UID}\}$
$\{send(\text{"cleanup"}, u)_{1,2} : u \text{ a UID}\}$
$\{RECOVER_1\}$
$\{choose(v)_1 : v \text{ a UID}\}$

FivePacketHandshake, process P_2:

Signature:

Input:

$receive(p)_{1,2}$, $p \in \{(\text{"needuid"}, v) : v \text{ a UID}\} \cup \{(\text{"send"}, m, u) : m \in M, u \text{ a UID}\}$
$\cup \{(\text{"cleanup"}, u) : u \text{ a UID}\}$

$CRASH_2$

Output:

$RECEIVE(m)_{1,2}$, $m \in M$

$send(p)_{2,1}$, $p \in \{(\text{"accept"}, u, v) : u, v \text{ UIDs}\} \cup \{(\text{"ack"}, u) : u \text{ a UID}\}$

$RECOVER_1$

States:

status $\in \{idle, accept, rcvd, ack, crashed\}$, initially *idle*
buffer, a FIFO queue of M, initially empty
uid-v, a UID or *null*, initially *null*
uid-u, a UID or *null*, initially *null*
last, a UID or *null*, initially *null*
used, a set of UIDs, initially empty
send-buffer, a FIFO queue of packets, initially empty

725

Transitions:

$receive(\text{"needuid"}, v)_{1,2}$
Effect:
if *status* = *idle* then
$u :=$ any UID $\notin$ *used*
used := *used* $\cup \{u\}$
uid-v := v
uid-u := u
status := *accept*

$send(\text{"accept"}, u, v)_{2,1}$
Precondition:
status = *accept*
u = *uid-u*
v = *uid-v*
Effect:
none

$receive(\text{"send"}, m, u)_{1,2}$
Effect:
if *status* $\neq$ *crashed* then
if *status* = *accept*
and u = *uid-u* then
add m to *buffer*
last := u
status := *rcvd*
else if $u \neq$ *last* then
add ("ack", u) to *send-buffer*

$RECEIVE(m)_{1,2}$
Precondition:
status = *rcvd*
m is first on *buffer*
Effect:

$send(\text{"ack"}, u)_{2,1}$
Precondition:
status $\neq$ *crashed*
(*status* = *ack* and *last* = u)
or ("ack", u) is first on *send-buffer*
Effect:
if ("ack", u) is first on *send-buffer* then
remove first element of *send-buffer*

$receive(\text{"cleanup"}, u)_{1,2}$
Effect:
if *status* = *accept* and u = *uid-u*
or if *status* = *ack* and u = *last* then
uid-v := *null*
uid-u := *null*
last := *null*
status := *idle*

$CRASH_2$
Effect:
status := *crashed*

$RECOVER_2$
Precondition:
status = *crashed*
Effect:
buffer := empty sequence
uid-v := *null*
uid-u := *null*
last := *null*
send-buffer := empty sequence
status := *idle*

remove first element of *buffer*
status := *ack*

Tasks:
$\{RECEIVE(m)_{1,2} : m \in M\}$
$\{send(\text{“}accept\text{”}, u, v)_{2,1} : u, v \text{ UIDs}\}$
$\{send(\text{“}ack\text{”}, u)_{2,1} : u \text{ a UID}\}$
$\{RECOVER_2\}$

726 该代码非常巧妙。例如，有两种情形都使得P_1将一个*cleanup*包添加到它的*send-buffer*中。一是如上所述的“正常”情形，这种情况下P_1刚从P_2收到一个*ack*包。另一种情况是P_1刚收到一个UID u的(“*accept*”，u，v)包，而u不是P_1的当前*uid-u*；这种情况下P_2可能仍然以u作为其当前状态，需要一个*cleanup*包将其清除。

类似地，也有两种情况使得P_2生成*ack*包。一是P_2处在*ack*模式的“正常”情况，另一种情况是P_2刚收到某个“老”u的(“*send*”，m，u)包。后一种情况下，P_1的状态中可能仍然是*status=send*、*uid-u=u*，此时需要一个*ack*包以清除u。

下述定理说明，*FivePacketHandshake*协议采用非常弱的通道假设确保规范说明B。具体来说，通道可以丢失、重排序以及复制消息，惟一的限制是必须满足SLL条件和对复制的有限性限制。

定理22.16　采用任意的有限复制SLL通道，*FivePacketHandshake*协议实现规范说明B，即：对每个公平运行α，都有$\alpha|ext(B) \in traces(B)$。

证明概要　安全属性是显然的——协议不对消息做复制或重排序。证明的难点在于活性的论述。从该算法本身并不能看出它会持续不断地将后续的消息递送给U_2。

论述活性的关键部分在于对下述这样一种情况的证明：P_1的*status=needuid*，而P_2的*status=accept*且*uid-v*等于一个不同于P_1的当前*uid-v*值的v。这一情况意味着P_2将忽略来自P_1的任何当前*needuid*包。我们必需证明值v最终会被从P_2中剔除，从而使得当前*needuid*包有机会达到P_2。因此假设v不会被剔除。这样P_2的公平性属性意味着P_2将发送无限数量的(“*accept*”，u，v)包。而通道活性条件意味着这些包中有无限个包要达到P_1，且根据P_1的代码，每个到达P_1的包都会引起一个(“*cleanup*”，u)的发送。同样是根据通道活性属性，最终有一个(“*cleanup*”，u)包必定到达P_2。从而使得v被丢弃。

但是事情会变得比这更复杂。我们需要说明的是，在v被丢弃之后，P_2有可能获得另一个同样与P_1当前*uid-v*值不同的v。如果P_1收到一个旧的*needuid*包，这种情况就会发生。但根据上述的证明，这个*uid-v*值同样也会被丢弃。这种情况可以发生任意次，但由于我们已经假设通道只允
727 许有限次的复制，因此在一个当前*needuid*包最终到达P_2前，这种情况发生的次数是有限的。 □

终态静止性　*FivePacketHandshake*协议在实际应用中的一个重要属性是当只有有限数量的*SEND*、*CRASH*以及*RECOVER*事件时，两个进程最终都会到达各自的终态、并将一直保持在这个终态，每个进程的终态与其初始状态除*used*集合之外完全相同。简单地说，就是该协议最终将“忘掉”所发生的任何事情。因此，当协议不再处理从U_1到U_2的消息时，没有必要为其保留内存。事实上，这就使得网络中的同一个进程对可以并行地为大量不同的用户对(U_1，U_2)模拟*FivePacketHandshake*协议。(用每个进程的I/O自动机组合来合成协议的各实例。)如果在某个时刻只有少数几个二元组(U_1，U_2)正在通信，那么*FivePacketHandshake*协议的全部运行所需要

的总内存也就比较小。

有限UID集合 实际上，可用UID的数量是非常大的，但也不像我们想像的那样是无限制的。例如，可以选择一组递增的连续整数对n取模作为UID，其中n是一个非常大的整数。只要在UID“绕回来”重用某个值u之前，系统中所有携带同一个UID u的旧包都已经删除，则协议的这一“有限版本”就可以正常工作(事实上，可以用一个模拟关系将这个“有限版本”与普通*FivePacketHandshake*协议关联起来，以证明其正确性)。根据对消息传递时间、局部处理时间以及高层消息提交频率的已知限制，或者根据丢弃旧包的显式策略，在实际环境中这一断言是成立的。

22.6 参考文献注释

在[54，290，273]中有关于ISO分层通信体系结构的详细描述。*Stenning*协议出自Stenning[270]。*ABP*最早是由Bartlett、Scantlebury和Wilkenson[42]提出来的。*ABP*不但是一个有趣而有用的协议，而且还经常用作协议证明技巧的测试案例。例如，在[177，59，229，38，146，260，280]中均有*ABP*的正确性证明。

关于重排序和复制的简单不可能性结论是从Wang和Zuck的工作[284]推导而来的。*Probe*协议是由Afek、Attiya、Fekete、Fischer、Lynch、Mansour、Wang和Zuck[4]所开发的，所采用的是由Afek和Gafni[5]所开发的早期*Probe*协议的思想。解决这一问题的一个早期协议是由Attiya、Fischer、Wang和Zuck[23]所开发的，但不具备本书中所提供协议的模块性属性。Afek等[4]也 728
证明了定理22.11的不可能性结论。Mansour和Schieber[220]、Wang和Zuck[284]以及Tempero和Ladner[278，277]证明了相关的不可能性结论。

关于有进程崩溃环境的重要不可能性结论，定理22.14，出自Lynch、Mansour和Fekete[206]，还出自Spinelli[268]。这两个结论被综合到一篇文章[112]中。Spinelli[268]还证明了许多其他关于可靠通信不可能性的结论。Baratz和Segall[41]论述如何用很少量的稳定数据来容忍进程崩溃，并提出对没有稳定数据情况下的不可能性结论的猜测。Attiya、Dolev和Welch[21]证明了相关的结论。

*FivePacketHandshake*协议是Belsnes[44]所设计的一系列协议之一；这一组协议确保更严格的正确性条件，同时容忍当增加额外的包交换时更复杂的通道故障行为类型。*FivePacketHandshake*协议是建立网络连接的标准协议，在TCP、ISO TP-4以及许多其他传输协议中被采用。Lampson、Lynch和Sϕgaard-Andersen[188，190，264]给出该协议的一个泛化版本的完全正确性证明，同时还给出另一个采用定时机制的协议的证明。

22.7 习题

22.1 证明引理22.1。

22.2 证明存在从*Stenning*协议到F的模拟关系，其中*Stenning*协议采用例14.1.2中的通道$A_{1,2}$和$A_{2,1}$，F是相应的通用可靠FIFO通道。即，如果s和u分别是*Stenning*协议和F的状态，那么仅当下列条件全部为真时我们定义$(s, u) \in f$：

a）如果$s.tag_1 = s.tag_2$，那么首先移去$s.buffer_1$的第一个元素[⊖]，然后将这个被“简化”的$buffer_1$添加到$s.buffer_2$的尾部，就得到$u.queue$。

⊖ 根据引理22.1可知，这种情况下$s.buffer_1$非空。

b）否则，将$s.buffer_1$添加到$s.buffer_2$的尾部，就得到$u.queue$。

证明f是一个模拟关系。

22.3 用习题22.2的结果证明引理22.2。确切地说，就是用这个模拟以及一个相应运行证明公平
729 性属性。

22.4 证明定理22.3。

22.5 考虑采用这样一种通道的*Stenning*协议，这种通道与22.2节中的通道类似但允许消息的无限复制。即对例14.1.2中自动机A的代码进行修改，删除其中对*send*操作效果的有限性限制。

a）证明该协议不能再正常工作，换句话说，该协议违背它所要实现的可靠FIFO通道的公平性属性。

b）论述为恢复*Stenning*协议的正确性，需要如何对通道的活性条件进行少量强化。

22.6 证明引理22.4。

22.7 证明引理22.5证明中的f是一个模拟关系。

22.8 证明定理22.7。

22.9 设计一个*ABP*的泛化版本，该版本采用的标志是整数mod k，代替原来的整数mod 2，其中$k \geq 2$。新的协议必须采用与*ABP*同样的(FIFO)通道，也必须实现F。但是，在新协议中，当P_1在等待第一个消息的确认时，必须能够发送其*buffer*前端的前p个消息。对于k，p的值最大可取多少？

22.10 论述当允许通道无限制复制消息时，*ABP*仍然能正常工作。通道仍然不能够对消息进行重排序，并且消息丢失受WLL条件限制。

22.11 对于某个已知的范围k，如果限制通道最多只能复制每个低层消息k次，定理22.8还成立吗？我们仍然假设通道不丢失消息，但允许对消息进行任意重排序。如果不成立，给出证明；如果成立，给出相应的算法。

22.12 给出引理22.9的证明细节。换句话说，就是要给出*cause*函数的详细定义，并证明该函数满足$I_{1,2}$说明规范所要求的属性。

22.13 假设对*Probe*协议的Layer 1进行修改，在$receive(m)_{1,2}$操作的作用部分添加一行
730 “*pending*:=*pending* − 1”。结果协议还正确吗？证明其正确性或给出一个反例。

22.14 证明在声明22.12的证明中使用的事实，即：$P_1 \times P_2$的任何有限一致运行都可以被扩展成一个公平一致运行，且在扩展后，没有新的$SEND_{1,2}$事件发生、所有$receive_{1,2}$事件都是由新$send_{1,2}$事件引起的。

22.15 强化定理22.11的结论，在结论中给出最好情况下传输后续高层消息所需要低层消息数量的增长率的下限。

22.16 假设22.5.2节中的说明规范B被弱化，以允许丢失在最后*RECOVER*之后所发送的某些消息。然而，如果只有有限数量的*CRASH*和*RECOVER*事件，那么在这个*RECOVER*之后发送的前k个消息之后的全部消息都必须被递送。像前面一样，不允许消息复制，但可以进行重排序。

采用22.5.2节中所使用的同样的通道说明规范$Q_{1,2}$和$Q_{2,1}$，将定理22.14中的不可能性结论扩展至这一更加弱化的规范说明、或者设计一个解决这一问题的算法。

22.17 研究问题：回答习题22.16中的相同问题，但这一次是采用下述方法进一步弱化说明规范B。如果只有有限数量的*CRASH*和*RECOVER*事件，那么所发送的所有消息都必须最终被成功递送。

22.18 证明*FivePacketHandshake*协议不会对消息进行重排序或复制。

22.19 构造*FivePacketHandshake*协议的这样一个运行，其中需要用第二类*ack*包来剔除来自P_1的一个UID，这个*ack*包是在P_2收到一个旧的*send*包时产生的。

22.20 给出*FivePacketHandshake*协议所需要活性属性的详细证明。即，证明在最后一个*RECOVER*事件后发送的任何消息都最终一定被传递给U_2。

22.21 在只有有限数量的*SEND*、*CRASH*以及*RECOVER*事件情况下考虑*FivePacketHandshake*协议。证明在这种情况下，两个进程最终都会到达各自的终态、并将一直保持在这个终态，每个进程的终态与其初始状态除*used*集合之外完全相同。

22.22 在一个基于全连通无向图的网络中，设计一个高效算法，实现两个用户间的可靠FIFO通信。像通常一样，假设在该图的每个节点都有一个进程，在每条边上都有一个可靠FIFO发送/接收通道。

731

第三部分　部分同步算法

本书的最后部分由第23~25章组成。这些章中介绍关于部分同步模型的算法和下界结果。在部分同步模型中，系统组件中有一些关于计时的信息，只是这些信息不像同步模型中的那样完整。这部分信息可以提供一个在真实分布式系统中可获得的计时知识的现实模型。

同前面一样，在开始部分——第23章，介绍一个形式化的模型。随后的两章介绍两个算法：第24章讨论部分同步共享存储器系统的互斥性，第25章介绍部分同步网络系统的一致性。这几章所介绍的是分布式算法理论的前沿，这些内容有可能成为这一领域新的研究热点。

第23章 建模V: 部分同步系统模型

本书的最后三章是关于部分同步分布式算法（或称基于定时分布式算法）研究情况的简要介绍。我们回顾一下，第一部分 (第2~7章)分析同步分布式算法，而第二部分 (第8~22章)则讨论异步分布式算法。但是在这两者之间，还存在着一类有趣的模型和算法，我们将这类系统称作部分同步系统。在部分同步系统中，系统部件中有一些关于时间的信息，尽管这些信息可能并不精确。例如，在部分同步网络中，进程也许可访问接近同步的时钟，或可以知道进程步时间或消息递送时间的大概界限。

部分同步模型可能比完全同步模型或完全异步模型有更现实的意义，因为现实的系统通常会使用一些时间信息。然而，与同步系统和同步系统几近完善的理论相比，部分同步系统的理论还有待进一步的研究发展。我们在此所介绍的想法只是一个开始，我们认为在基于计时计算的基础上，今后将有大量这方面的研究工作。

在本章，介绍关于基于计时分布式算法的模型及其证明方法。首先在23.1节介绍一个称作*MMT*模型的定时自动机模型，该模型是根据其发现者Merritt、Modugno和Tuttle所命名的。MMT模型是一个I/O 自动机模型的简单变种，该I/O自动机模型可用于大多数基于计时的算法
735 的建模。为了在此模型中使用一些特定的基本证明方法——特别是不变式方法和模拟方法，我们发现将每个MMT自动机转换成我们称作通用定时自动机(*General Timed Automaton*, *GTA*) 的另一类自动机是很有用处的。我们将在23.2节介绍GTA模型，同时介绍如何将MMT自动机转换为GTA自动机。在23.3节，讨论可以用于这些模型的证明技巧。

在第24章和第25章，给出在部分同步环境中关于互斥和一致性的一些基本结论。

23.1 MMT 定时自动机

将一个I/O自动机模型的公平性条件简单地替换为时间的下界和上界，即得到一个MMT定时自动机模型。需要提醒读者的是，只用时间上界替代公平性条件不能给模型增加任何让我们感兴趣的功能，因为单独的上界并不会对一个I/O自动机产生的运行集合进行限制。(实际上，在本书全部有关“异步”的章节，为分析时间复杂性，我们已经将时间上界和算法的任务联系起来。这种分析之所以有用，是由于实际上这些时间的界限并不限制算法的行为。)然而，引入时间上界和下界的确可以产生更强的功能，因为这样使得我们能够对运行的集合进行限制。事实上，对于许多基于计时的算法，正确性主要依赖于由时间界限所引起的、对运行的限制。

23.1.1 基本定义

我们从只有有限任务的I/O自动机A开始。A的界限图 b 是一对映射$lower$和$upper$ ，它们分别对每个任务给出其时间的下界和上界。对每个任务C， 我们要求$lower(C)$ 和 $upper(C)$满足条件 $0 \leq lower < \infty$、$0 < upper(C) \leq \infty$以及$lower(C) \leq upper(C)$。即，下界不允许为∞，上界不允许为0，下界不能大于上界。一个*MMT*自动机由一个I/O自动机A以及A的一个界限图组成。

下面定义MMT自动机是如何执行的。定义MMT自动机$B = (A, b)$的一个定时运行就是一个有限序列$\alpha = s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \ldots, (\pi_r, t_r), s_r$或一个无限序列$\alpha = s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \ldots, (\pi_r, t_r), s_r, \ldots$，其中$s$是I/O自动机$A$的状态、$\pi$是$A$的操作、$t$是$R^{\geq 0}$中的时间。我们要求

序列s_0，π_1，s_1，...——即忽略时间后的序列α——是I/O 自动机A的一个常规运行。我们同样要 736
求在α中的连续时间t_r是非递减的，并且满足由界限图b所描述的时间下界和上界的要求。

满足时间上界和下界的要求意味着什么？为精确地说明这个问题，当任务C在s_r中被使能并且下列条件之一为真时，我们定义r为任务C的一个*初始索引*：

1）$r=0$。

2）C未在s_{r-1}中被使能。

3）$\pi_r \in C$。

初始索引集代表开始测量时间界限的那些点。因此，对任务C的每个初始索引r，我们要求下列条件成立。(令$t_0=0$。)

上界条件 如果存在$k>r$且$t_k > t_r + upper(C)$，那么存在$k'>r$且$t_{k'} \leq t_r + upper(C)$，使得$\pi_{k'} \in C$或$C$未在$s_{k'}$中被使能。

下界条件 不存在$k>r$使得$t_k < t_r + lower(C)$且$\pi_k \in C$。

上界条件表明，从任务C的任何初始索引开始，如果时间超过C的约定时间上界，那么在运行过程中，一定发生C的某操作，或者C一定变成非使能的。下界条件表明，对于C的任何初始索引，C的任何操作都不会在约定的下界之前发生。

我们用*texecs*(B)表示B的定时运行集合。当一个状态是B的某个有限时间运行的终态时，则称该状态在B中是*可到达的*。

上界属性和下界属性都是安全性属性。我们同样对下面的这个基本活性属性感兴趣：当满足下面的条件时，我们称一个定时运行是*可容许的*：

许可性条件 如果定时运行α是一个无限序列，那么操作的时间接近于∞。如果α是一个有限序列，那么在α的终态，当任务C是被使能时$upper(C)=\infty$。

许可性条件表明，时间消耗是正常的，并且如果自动机被调度执行更多操作，则处理不会停顿。我们用*atexecs*(B)表示B的可容许定时运行的集合。本书中，我们将重点放在可容许定时运行上。

需要提醒的是，在一个可容许定时运行中，任务C的时间上界∞并不强制需求不断地发生任务C中的操作。这一点与我们在关于异步的章节中所做的有所不同：在8.6节，定义关于定时 737
运行的另一个概念，这一概念要求所有任务满足公平性条件，同时其中的部分任务必须满足时间上界条件。前面我们用这个组合概念来分析时间复杂性。现在我们完全去掉公平性条件而仅考虑时间界限。定义一个部分任务有时间界限、部分任务有公平性条件的另一版本的MMT模型也是可以的，但是在本书中不给出这样的正式定义。相反，仅当在特定算法中需要组合时间界限和公平性条件时，我们才以非形式化的方式讨论这一问题。

在基于计时计算的任何有用模型中，为了排除某些实际上不会出现的异常行为（例如一个自动机在有限时间内执行无限多的输出[⊖]），需要某种类型的许可性条件。尽管这样的执行有一些形式上的意义，但它们对现实是无意义的，也很难理解。一个好的定时系统模型应该避免考虑这种问题。

我们定义*定时轨迹*来描述MMT自动机的外部行为。定时运行α的定时轨迹用*ttrace*(α)表示，是由α中的全部外部操作及对应的时间组成的一个子序列，其中α是B的一个定时运行。B的可容许定时轨迹是B的可容许定时运行的定时轨迹，我们用*attraces*(B)来表示这些定时轨迹的集合。

⊖ 这种行为有时称作*Zeno行为*，指Zeno悖论。在Zeno悖论中，奔跑者Achilles跑了无限多步，后面的每一步都比前面的短，因此越来越接近目标（一个龟）但永远无法达到目标。

在基于计时的系统中，MMT自动机可以用于其中多种类型组件的描述。特别适合于在低层次上模型化计算机系统，因为任务结构和相应的时间界限为模型化物理系统组件及其速度提供了自然方法。然而，它们不太适合在高层次上描述系统，也不适合于提供正确性规范说明。这是因为它们关于任务和时间界限的相对格式化约定不能为表达预定的行为提供最好的“语言”。

例 23.1.1　通道MMT 自动机

我们基于例8.1.1的通用可靠FIFO发送/接收通道自动机$C_{i,j}$，定义一个MMT自动机 $D_{i,j} = (C_{i,j}, b)$。$D_{i,j}$的界限图b 给出通道中最早消息的传递时间上界d, 其中d 是某个固定的正实数。而对传
738 递时间的下界则不作要求。在异步算法的有关章节中，为实现时间性能分析，我们常用到一个通道，而$D_{i,j}$则是对该通道的一种形式化描述。

因此，如果rec 表示$C_{i,j}$ 的惟一任务，那么对于某个固定的 $d \in R^+$,定义b 为二元组$(lower, upper)$，其中$lower(rec) = 0$、$upper(rec) = d$。下列都是$D_{i,j}$的可容许定时轨迹：

$$(send(1)_{i,j}, 0), (send(2)_{i,j}, 0), (receive(1)_{i,j}, d), (receive(2)_{i,j}, 2d)$$

$$(send(1)_{i,j}, 0), (send(2)_{i,j}, 0), (receive(1)_{i,j}, 0), (receive(2)_{i,j}, 0)$$

$$(send(1)_{i,j}, 0), (receive(1)_{i,j}, d), (send(2)_{i,j}, d), (receive(2)_{i,j}, 2d), (send(3)_{i,j}, 2d), (receive(3)_{i,j}, 3d), \ldots$$

另一方面，下面的则不是$D_{i,j}$的可容许定时轨迹：

$$(send(1)_{i,j}, 0), (send(2)_{i,j}, 0), (receive(1)_{i,j}, d)$$

$$(send(1)_{i,j}, 0), (receive(1)_{i,j}, 2d)$$

$$(send(1)_{i,j}, 0), (receive(1)_{i,j}, d), (send(2)_{i,j}, d), (receive(2)_{i,j}, d), (send(3)_{i,j}, d), (receive(3)_{i,j}, d), \ldots$$

这三个序列中，第一个序列不能成为一个可容许定时轨迹的原因在于它是有限的，而rec任务直到最后才使能。通常，任何包括至少 k个$send$ 输入的可容许定时轨迹也必须包括至少k个相应的$receive$ 输出，因为上界条件和许可性条件意味着rec任务的常规公平性条件。第二个序列不能成为一个可容许定时轨迹的原因在于它破坏了上界条件。第三个序列也不是一个可容许定时轨迹，因为它破坏了许可性条件——即使发生无限多个行为时，它也不允许时间的增加超过d。

例 23.1.2　超时 MMT自动机

我们定义一个MMT自动机P_2，该自动机等待从另一个进程P_1接收一条消息，如果在一定的时间内消息没有到达则执行一个$timeout$操作。P_2 测量所等待时间的方法是统计自己的$k \geq 1$个步，其中k是一个固定的数，并假定这些步都满足时间下界l_1和上界l_2，$0 < l_1 \leq l_2 < \infty$。$count$到达0后最多等到时间$l$，$timeout$就会被执行。需要说明的是，对每个任务我们用封闭时间间隔的
739 形式书写下界和上界——我们会经常使用这个约定。

P_2 automaton:

Signature:

Input:
　$receive(m)_{1,2}$, $m \in M$

Internal:
　$decrement$

Output:
　$timeout$

States:
count $\in \mathbb{N}$, initially k
status $\in \{active, done, disabled\}$, initially *active*

Transitions:

decrement
Precondition:
status = *active*
count > 0
Effect:
count := *count* − 1

timeout
Precondition:
status = *active*
count = 0
Effect:
status := *done*

$receive(m)_{1,2}$
Effect:
if *status* = *active* then
status := *disabled*

Tasks and bounds:
$\{decrement\}$, bounds $[\ell_1, \ell_2]$
$\{timeout\}$, bounds $[0, \ell]$

在一个可容许定时运行中，P_2 递减*count* 直到 *count* = 0 为止、或直到产生一个*receive*(*m*)来阻止超时为止。在*count* 为0后，P_2 执行一个 *timeout* 事件（条件是在此之前没有*receive*事件产生。）不难看出，在P_2的任何定时运行中，如果有*timeout*事件发生，则必定发生在$[kl_1, kl_2 + l]$时间间隔内的某个时间。而且，如果有*timeout*事件发生，则其前面没有*receive*事件发生。最后，在P_2的一个可容许定时运行中，如果没有 *receive* 事件发生，那么实际上一定会发生一个*timeout*事件。

例 23.1.3 双任务轨迹

我们定义一个有两个任务的简单MMT 自动机*Race*，这两个任务分别为*main* 和 *int* (中断)。任务*main*的作用是当布尔变量 *flag* 为 *false*时就不断的给计数器*count*加1。任务*int*的功能只是设置 *flag* := *true*。当*flag* = *true*时， 任务*main* 减少 *count* 直到其为0为止，然后报告结束操作。 740
任务*main* 与界限 l_1 和 l_2 关联，$0 < l_1 \leq l_2 < \infty$ ，而任务 *int* 只有上界 l。

***Race* automaton:**

Signature:

Input:	Internal:
none	*increment*
Output:	*decrement*
report	*set*

States:
count $\in \mathbb{N}$, initially 0
flag, a Boolean, initially *false*
reported, a Boolean, initially *false*

Transitions:

increment
Precondition:

set
Precondition:

flag = *false*
Effect:
count := *count* + 1

decrement
Precondition:
flag = *true*
count > 0
Effect:
count := *count* − 1

flag = *false*
Effect:
flag := *true*

report
Precondition:
flag = *true*
count = 0
reported = *false*
Effect:
reported := *true*

Tasks and bounds:
main = {*increment*, *decrement*, *report*}, bounds $[\ell_1, \ell_2]$
int = {*set*}, bounds $[0, \ell]$

在*Race*的每个可容许定时运行中，最后都会发生一个*report*事件。在23.3.3节，将略证这个*report*一定在$l + l_2 + Ll$时间内发生，其中$L = l_2 / l_1$。（L可以看作是对系统中计时不确定性的度量。）

23.1.2 操作

741 我们为MMT自动机集合定义合成和隐藏操作，这些操作与I/O自动机的相关操作类似。

合成 通过识别在不同自动机中的同名操作可以合成MMT自动机，这一方法基本与普通I/O自动机的合成方法完全相同。然而，不同于I/O自动机的是，我们只定义MMT自动机的有限集合的合成。这是因为一个MMT自动机只允许存在有限数量的任务。

按照8.2.1节的兼容性定义，如果在一个MMT自动机的有限集合中，各MMT自动机所采用的I/O自动机是兼容的，我们称这个MMT自动机有限集合是兼容的。因此，对于一组 MMT自动机$\{(A_i, b_i)\}_{i \in I}$的有限兼容集合，*composition* $(A, b) = \prod_{i \in I}(A_i, b_i)$是一个如下定义的MMT自动机：

- $A = \prod_{i \in I} A_i$，即，A是一个组合自动机，所有组件均采用I/O自动机A_i。
- 对于A的每个任务C，b对于C的下界和上界与b_i的相同，其中A_i是具有任务C的惟一I/O自动机组件。

对于一组 I/O 自动机，有时使用中缀操作符 × 来表示合成操作。例如，如果 $I = \{1, \ldots, n\}$，那么有时将$\prod_{i \in I} A_i$写作 $A_1 \times \cdots \times A_n$。

例 23.1.4 MMT自动机的合成

我们考虑三个MMT自动机的合成。第一个自动机是进程P_1，P_1可能是活动的也可能已经死掉（具体是哪种状态是不确定的，由初始状态决定）。如果它是活动的，则周期性地向流出通道发送消息，所发送消息取自一个固定的消息字母表M，并且两次发送之间的最大时间间隔为$l > 0$。

P_1 automaton:

Signature:
Input: none
Output: $send(m)_{1,2}$, $m \in M$

States:
status ∈ {*alive*, *dead*}, initially arbitrary

Transitions:

$send(m)_{1,2}$
Precondition:

status = *alive*
Effect:
none
Tasks and bounds:
$\{send(m)_{1,2} : m \in M\}$, bounds $[0, \ell]$

742

另外两个自动机分别是例23.1.1中定义的通道$D_{1,2}$、和在例23.1.2中定义的超时进程P_2。如果$kl_1 > l + d$，那么在任何可容许定时运行中，仅当P_1已经死亡时合成自动机才执行一个*timeout*。而且，这个*timeout*在不晚于$kl_2 + l$的时间执行。

我们用三个类似于定理8.1~8.3的基本结论来结束本小节。这些结论将一个合成自动机的可容许定时运行和可容许定时轨迹与MMT自动机组件的可容许定时运行和可容许定时轨迹联系起来。第一个结论表示，根据一个合成MMT自动机的可容许定时运行或可容许定时轨迹，可投影产生其中自动机组件的可容许定时运行或可容许定时轨迹。

令$\{(A_i, b_i)\}_{i\in I}$为一组MMT自动机的兼容集合，同时令$(A, b) = \prod_{i\in I} (A_i, b_i)$。对每个$i$，用$B_i$表示MMT自动机$(A_i, b_i)$，且用$B$表示$(A, b)$。对于$B$中的某个定时运行$\alpha = s_0, (\pi_1, t_1), s_1, \ldots$，删除其中$\pi_i$不是$A_i$的操作的每个二元组$(\pi_r, t_r)$，$s_r$，并将剩下的每个$s_r$用$(s_r)_i$代替，即用自动机$A_i$的状态$s_r$片段代替，所得到的结果序列用$\alpha|B_i$表示。同样，对于$B$的任何定时轨迹$\beta$（或者更通用一些，任何带有时间的操作序列），用$\beta|B_i$表示$\beta$中由包含$A_i$的操作的所有二元组所组成的子序列。

定理23.1 令$\{B_i\}_{i\in I}$为一组MMT自动机的兼容集合，$B = \prod_{i\in I} B_i$。

1）如果$\alpha \in atexecs(B)$，那么对每个$i \in I$，有$\alpha|B_i \in atexecs(B_i)$。

2）如果$\beta \in attraces(B)$，那么对每个$i \in I$，有$\beta|B_i \in attraces(B_i)$。

证明 此证明留做一道习题。 □

另外两个定理与定理23.1相反。下一个定理阐述的是在一定条件下，一组组件MMT自动机的可容许定时运行可以粘合在一起，组成合成MMT自动机的一个可容许定时运行。

定理23.2 令$\{B_i\}_{i\in I}$为一组MMT自动机的兼容集合，$B = \prod_{i\in I} B_i$。假设对每个$i \in I$，α_i是B_i的一个可容许定时运行，β是一个二元组*(action, time)*的序列，其中β的所有操作在*ext*(*A*)中，使得对于每个$i \in I$，有$\beta|B_i \in ttrace(\alpha_i)$。那么存在$B$的一个可容许定时运行$\alpha$，使得$\beta = ttrace(\alpha)$、并且对每个$i \in I$有$\alpha_i = \alpha|B_i$。

743

证明 此证明留做一道习题。 □

最后一个定理讲述的是把一组MMT自动机的可容许定时轨迹组件粘合在一起，形成一个合成可容许定时轨迹。

定理23.3 令$\{B_i\}_{i\in I}$为一组MMT自动机的兼容集合，$B = \prod_{i\in I} B_i$。假设β是一个二元组*(action, time)*的序列，其中β的所有操作在*ext*(*A*)中。如果对每个$i \in I$有$\beta|B_i \in attraces(B_i)$，那么$\beta \in attraces(B)$。

证明 此证明留做一道习题。 □

隐藏 对一组MMT自动机的隐藏操作是按照8.2.2节中所给普通I/O 自动机的隐藏操作进行定义的。即，如果$B = (A, b)$是一个MMT 自动机并且$\Phi \subseteq out(A)$，那么$hide_\Phi(B)$是一个 MMT自动机$(hide_\Phi(A), b)$。像I/O 自动机的隐藏操作一样，此操作只是简单地将输出操作重新归类为内部操作。

23.2 通用定时自动机

MMT自动机中，对时间的限制是通过增加运行的上界和下界条件给出的。这种时间限制的另一种表达方法是将时间限制直接编码到自动机的状态和转移中。这个方法的好处在于它支持一些基于状态的重要证明方法，例如用于时间系统的时间属性和正确性推论的不变式断言和模拟关系。

我们在本节描述另一种定时自动机模型，将其称作通用定时自动机(*GTA*)模型。通用定时自动机没有“外部”时间限制——它们的所有时间限制都被显式地编码到状态和转移中。我们将说明，MMT自动机可以看作是对时间限制进行编码的一种通用定时自动机特例。然而，并不
744 是所有的GTA自动机都是MMT自动机，事实上某些GTA具有任何MMT自动机都不具备的行为。

23.2.1 基本定义

我们假设一个通用操作的集合，其中包括特殊的时间通过操作$\nu(t)$, $t \in R^+$。时间通过操作$\nu(t)$用数值t表示已经过去的时间。时间签名S是一个由四个操作集合组成的四元组，其中每个集合与其他三个集合的交集都为空，这四个集合分别是：输入操作$in(S)$,输出操作$out(S)$，内部操作$int(S)$和时间通过操作。我们定义

- 可见操作$vis(S)$为输入操作和输出操作的并集$in(S) \cup out(S)$
- 外部操作$ext(S)$为可见操作和时间通过操作的并集$vis(S) \cup \{ \nu(t): t \in R^+\}$
- 离散操作$disc(S)$为可见操作和内部操作的并集$vis(S) \cup int(S)$
- 局部控制操作$local(S)$为输出操作和输入操作的并集$out(S) \cup int(S)$
- $acts(S)$是S的所有操作

一个GTA A由下列四个部分组成：

- $sig(A)$，时间签名
- $states(A)$，状态集合
- $start(A)$，$states(A)$的一个非空子集，也称作输入状态或起始状态。
- $trans(A)$，状态转移关系，其中$trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$

与I/O自动机和MMT自动机不同，GTA自动机没有$tasks(A)$组件。同前面一样，我们使用$acts(A)$作为$acts(sig(A))$的速记符号，类似的包括$in(A)$等。A必须满足下面两个简单的公理：

A1: 如果$(s, \nu(t), s')$和$(s', \nu(t'), s'')$都属于$trans(A)$，那么$(s, \nu(t+t'), s'')$属于$trans(A)$。

A2: 如果$(s, \nu(t), s') \in trans(A)$并且$0 < t' < t$，那么存在状态$s''$，使得$(s, \nu(t'), s'')$和$(s'', \nu(t-t'), s')$
745 都属于$trans(A)$。

公理A1允许将重复的时间通过步组合成一步，而公理A2与公理A1相反，是允许将一个时间通过步分解成两个时间通过步。

GTA的一个定时运行片断A是一个有限序列$\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$或者无限序列$\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r, \ldots$，其中$s$表示$A$的状态，$\pi$是$A$的操作（可以是输入操作、输出操作、内部操作或时间通过操作），且对每个k，$(s_k, \pi_{k+1}, s_{k+1})$是$A$的转移。需要说明的是，如果这个序列是有限的，它必须以一个状态结尾。当一个定时运行片断以一个起始状态开始时，则将其称为定时运行。

如果α是某个定时运行片断，且π_r是α中的某个离散操作，则将α中位于π_r之前的所有时间通过操作的实数之和称作π_r的发生时间。如果在定时运行片断α中所有时间通过操作的实数之和为∞，则称α为可容许的。我们用$atexecs(A)$表示A中的可容许定时运行集合。虽然我们有时也会考虑有限定时运行，即那些有限序列，但我们将主要考虑可容许定时运行。当一个状态是

A的某个有限定时运行的终态时，称该状态在A中是可到达的。

定时运行片断α的定时轨迹是α中可见事件的序列，其中每个事件都有自己的发生时间。A的可容许定时轨迹是A的可容许定时运行的定时轨迹，用$attraces(A)$表示。需要说明的是，一个可容许定时轨迹可以是有限的，即使它是从一个（无限的）可容许定时运行所导出的。

从公理A1 和公理A2可以知道，对于那些仅在时间通过步的分开和合并方面存在分别的定时运行片断之间没有太大的差别。因此，我们为定时运行片断定义一个等价关系，以说明它们除时间通过外都是相同的。也就是说，当定时运行片断α和α'有相同的初始状态、最终状态和总的时间通过量，除了在α中将α'的部分时间通过步用时间通过步的有限序列代替外，α和α'完全相同，则称α是α'的时间通过细化。如果定时运行片断α和α'有一个公共的时间通过细化，我们就说它们是时间通过等价的。

例 23.2.1 一个通用定时自动机

我们描述一个类似于例23.1.1中MMT自动机$D_{i,j}$的通用定时自动机$D'_{i,j}$。具体说，它们有同样的可容许定时轨迹。$D'_{i,j}$只是简单地将$D_{i,j}$的时间限制——通道中最早消息的传递时间上界d——编码到它的状态和转移中。实现编码的方法是：在变量now中保存当前时间的显式轨迹，同时在变量$last$中保存可能发生下一个消息传递的最晚时间的轨迹。需要说明的是，$last$ 的值代表的是绝对时间，而不是增加的时间。 746

我们用描述其他自动机时所采用的同类因果符号来描述$D'_{i,j}$，只是此时既需要描述时间通过操作的代码，也需要描述离散操作的代码。

当发生一个$send$事件时，像以前一样修改$queue$，但是除此而外，如果没有已被调度的消息递送，则需要将变量$last$设置成$now + d$，以反映在d时间内必须发生下一个消息递送的需求。当发生一个$receive$事件时，如果在此事件后$queue$仍然非空，则$last$界限被重新设置为$now + d$，以反映对下一个消息递送的时间需求；另一方面，如果$queue$为空，则把$last$设置成∞，以表示所有被调度要递送的消息都已经递送完毕。

时间通过操作$\nu(t)$的代码的书写方法与其他操作的代码的书写方法非常类似。$\nu(t)$的作用就是简单地对当前时间now加t。需要提醒的是，$\nu(t)$ 也有一个重要的前提条件：$now + t \leq last$。也就是说，时间不允许超过下一个消息递送的规定期限。这一条件初看起来好像有些奇怪——毕竟，一个程序或一台机器怎么能阻止时间的通过？但是，这种时间通过操作的规范说明方式仅仅是一种形式化的说明方法，目的是为了说明在指定的时间流逝前自动机保证执行某一操作。

$D'_{i,j}$ automaton:

Timed Signature:

Input:	Internal:
$send(m)_{i,j}$, $m \in M$	none
Output:	Time-passage:
$receive(m)_{i,j}$, $m \in M$	$\nu(t)$, $t \in R^+$

States:

$queue$, a FIFO queue of elements of M, initially empty
$now \in R^{\geq 0}$, initially 0
$last \in R^+ \cup \{\infty\}$, initially ∞ 747

Transitions:

$send(m)_{i,j}$	$\nu(t)$

Effect:
add m to *queue*
if $|queue| = 1$ then
$last := now + d$

Precondition:
$now + t \leq last$
Effect:
$now := now + t$

$receive(m)_{i,j}$
Precondition:
m is first on *queue*
Effect:
remove first element of *queue*
if *queue* is nonempty then
$last := now + d$
else $last := \infty$

不难看出，$D'_{i,j}$与$D_{i,j}$有同样的可容许定时轨迹集合。我们把其作为一道习题。

根据例23.2.1，我们应该有一个关于为何可以把MMT自动机看作是通用定时自动机的一个特例的概念：由MMT自动机(A, b)的界限图b所表达的时间需求可以被编码到相应GTA的状态和转移中。编码的实现方法是：用*last*状态组件保存上界需求的轨迹，并用附加的*first*状态组件保存下界需求的轨迹。在23.2.2节给出详细的编码构造方法。

然而GTA模型比MMT自动机模型通用得多。下一个例子是用GTA表达的另一个通道，但不能用MMT自动机表达这个通道。

例 23.2.2 一个非MMT的通用定时自动机

下面描述另一个GTA，$D''_{i,j}$，它代表一个可靠FIFO通道，但在这个GTA中，通道中的每个消息都需要时间界限d，而不仅仅是最早消息需要时间界限d。此时将消息递送的最后期限与消息
748 一起存储在*queue* 中，而不是存储在独立的*last* 组件中。当然，对最后期限的处理也是类似的。

$D''_{i,j}$ automaton:

Timed Signature:

Input:
$send(m)_{i,j}$, $m \in M$
Output:
$receive(m)_{i,j}$, $m \in M$
Internal:
none
Time-passage:
$\nu(t)$, $t \in R^+$

States:
queue, a FIFO queue of elements of $M \times R^+$, initially empty
$now \in R^{\geq 0}$, initially 0

Transitions:

$send(m)_{i,j}$
Effect:
add $(m, now + d)$ to *queue*

$receive(m)_{i,j}$
Precondition:
(m, t) is first on *queue*, for some t
Effect:
remove first element of *queue*

$\nu(t)$
Precondition:
if *queue* is nonempty then
$now + t \leq t'$, where t' is the time
in the first pair of *queue*
Effect:
$now := now + t$

我们声明不存在与$D''_{i,j}$有相同可容许定时轨迹集合的MMT自动机(这一断言的证明留做一道习题)。这一结论也可以理解为$D''_{i,j}$在物理上是无法实现的。但是，正如我们在前面章节所见，当我们不希望考虑通道中堆积的消息时，$D''_{i,j}$可以作为一个方便的抽象模型，用于算法的时间复杂性分析。

下面的例子说明一种异常情况：一个没有可容许定时运行的GTA。尽管这是一种异常的情况，但在通用模型中并没有任何东西来避免这种情况的发生。对于MMT自动机这种特别情况（以及在23.2.2节中描述的、与MMT自动机对应的GTA），这种异常不会发生。（见习题23-1。）可以在GTA模型中增加额外的限制，以排除这种异常，但是因为本书将主要讨论能用MMT自动机表达的算法，所以在此不描述这些限制。 749

例 23.2.3 一个没有可容许定时运行的通用定时自动机

考虑一个无限次发送同一个消息m的“过程自动机”A。但是连续发送的时间越来越接近，接近时间极限1。

A automaton:

Timed Signature:

Input:	Internal:
none	none
Output:	Time-passage:
$send(m)$	$\nu(t)$, $t \in R^+$

States:

$now \in R^{\geq 0}$, initially 0
$last \in R^{\geq 0} \cup \{\infty\}$, initially 0

Transitions:

$send(m)$
Precondition:
$now = last$
Effect:
$last := now + \frac{1}{2}$

$\nu(t)$
Precondition:
$now + t \leq last$
Effect:
$now := now + t$

事实上，事情会变得更糟糕——GTA的定义甚至允许定时自动机完全没有时间通过步！

GTA模型并不是基于计时计算的最通用模型。例如，它没有表达活性属性的功能（许可性除外）。在定时环境中，活性属性并不像在非定时环境中那么重要，因为许多活性属性（例如，说明某些事情最终会发生的条件）可以被相应的时间上界条件取代（例如，说明事件在时间t内发生的条件）。然而，有时它是有用的，可以同时表示一个系统的时间界限和活性属性。

GTA模型也不足以通用到可以为混合系统——由类似的物理组件和离散计算机组件组成的系统——提供详细描述的程度。但该模型对本书的目标已经足够。 750

23.2.2 将MMT自动机转化为通用定时自动机

我们已经说过，通用定时自动机是对MMT定时自动机的扩展。但是这一说法是不精确的，因为它们说明时间限制的方式不同：MMT自动机模型使用界限图，而GTA模型将时间限制编码成状态和转移。为了将MMT模型看作是GTA模型的一个特例，还需要做一些工作。在本节，阐述如何把MMT自动机(A, b)转化成一个自然对应的通用定时自动机$A' = gen\ (A, b)$。

这个构造方法与例23.2.1中根据$D_{i,j}$构造$D'_{i,j}$所使用的方法类似。包括将最后时间期限构造到状态中，并且在它们仍然有效时阻止时间超过这些期限。我们也对无时间通过的操作增加新的限制，以表示时间下界条件。

具体地说，在所采用I/O自动机的状态中增加一个*now*组件，并为每个任务C增加*first*(C)和*last*(C)组件。*first*(C) 和 *last*(C) 组件分别表示任务C中下一个操作所能发生的最早时间和最迟时间。*now*、*first*和 *last* 组件都以绝对时间而不是增量时间为值。还增加时间通过操作 $\nu(t)$ 。

根据界限图b所规定的*lower*和*upper*界限，*first* 和 *last* 组件被不同的步所更新。时间通过操作$\nu(t)$ 有一个显式的前提条件，该条件要求时间不能超过任何*last*(C)值；这是因为这些*last*(C)值代表不同任务的最后期限。对各个任务C的操作也增加限制，要求当前时间*now*必须大于下界*first*(C)。

更具体地说，除增加时间通过操作$\nu(t)$外，$t \in R^+$，$A' = gen\ (A, b)$ 的时间签名与A的时间签名相同。A'的每个状态由下列组件组成：

basic $\in$ *states*(A), initially a start state of A
now $\in R^{\geq 0}$, initially 0
for each task C of A:
　first(C) $\in R^{\geq 0}$, initially *lower*(C) if C is enabled in state *basic*, otherwise 0
　last(C) $\in R^+ \cup \{\infty\}$, initially *upper*(C) if C is enabled in *basic*, otherwise ∞

其转移的定义如下：

751 如果$\pi \in acts(A)$，那么仅当下列条件全部成立时有$(s,\pi, s') \in trans(A')$：

1. $(s.basic, \pi, s'.basic) \in trans(A)$.
2. $s'.now = s.now$.
3. For each $C \in tasks(A)$,
   a) If $\pi \in C$, then $s.first(C) \leq s.now$.
   b) If C is enabled in both $s.basic$ and $s'.basic$ and $\pi \notin C$, then $s.first(C) = s'.first(C)$ and $s.last(C) = s'.last(C)$.
   c) If C is enabled in $s'.basic$ and either C is not enabled in $s.basic$ or $\pi \in C$, then $s'.first(C) = s.now + lower(C)$ and $s'.last(C) = s.now + upper(C)$.
   d) If C is not enabled in $s'.basic$, then $s'.first(C) = 0$ and $s'.last(C) = \infty$.

如果$\pi = \nu(t)$，那么仅当下列条件全部成立时有$(s,\pi, s') \in trans(A')$：

1. $s'.basic = s.basic$.
2. $s'.now = s.now + t$.
3. For each $C \in tasks(A)$,
   a) $s'.now \leq s.last(C)$.
   b) $s'.first(C) = s.first(C)$ and $s'.last(C) = s.last(C)$.

定理 23.4　如果(A, b)是任意一个MMT定时自动机，那么$gen(A, b)$是一个通用定时自动机。而且，$attraces(A, b) = attraces\ (gen(A, b))$。

引理 23.5　在$gen(A, b)$的任意可到达状态中，以及对A的任意任务C，下列结论成立：

1）$now \leq last(C)$。

2）如果C被使能，那么 $last(C) \leq now + upper(C)$。

3）$first(C) \leqslant now + lower(C)$。

4）$first(C) \leqslant last(C)$。

删除不重要组件 如果b所规定的某些时间需求是不重要的——即，如果某些下界为0，或某些上界为∞——那么可以省略对这些组件的描述，以简化自动机$gen(A, b)$。我们将在例子中做这种简化。

例 23.2.4 经过转化的MMT自动机

令(A, b)为例23.1.4所描述的合成MMT自动机，由P_1、P_2和通道$D_{1,2}$组成。我们给出转化后的MMT自动机$A' = gen\ (A, b)$的详细代码。如上所述，删除其中不重要的界限。因此，在该代码中需要保留的界限仅包括各任务的上界以及P_2的递减任务的下界。我们用下面的名字表示其中的任务：P_1的惟一任务用*send*表示，通道$D_{1,2}$的惟一任务用*rec*表示，P_2的两个任务分别用*dec*和*timeout*表示。 752

A' automaton:

Timed Signature:

Input:
none
Output:
$send(m)_{1,2}$, $m \in M$
$receive(m)_{1,2}$, $m \in M$
timeout

Internal:
decrement
Time-passage:
$\nu(t)$, $t \in R^+$

States:

$status_1 \in \{alive, dead\}$, initially arbitrary
queue, a FIFO queue of elements of M, initially empty
$count_2 \in \mathbb{N}$, initially k
$status_2 \in \{active, done, disabled\}$, initially *active*

$now \in R^{\geq 0}$, initially 0
$last(send) \in R^+ \cup \{\infty\}$, initially ℓ if $status = alive$, otherwise ∞
$last(rec) \in R^+ \cup \{\infty\}$, initially ∞
$first(dec) \in R^{\geq 0}$, initially ℓ_1
$last(dec) \in R^+ \cup \{\infty\}$, initially ℓ_2
$last(timeout) \in R^+ \cup \{\infty\}$, initially ∞

Transitions:

$send(m)_{1,2}$
Precondition:
$status_1 = alive$
Effect:
add m to *queue*
$last(send) := now + \ell$
if $|queue| = 1$ then
$last(rec) := now + d$

$receive(m)_{1,2}$
Precondition:
m is first on *queue*
Effect:
remove first element of *queue*
if $status_2 = active$ then
$status_2 := disabled$
if *queue* is nonempty then
$last(rec) := now + d$
else $last(rec) := \infty$
$first(dec) := 0$
$last(dec) := \infty$
$last(timeout) := \infty$ 753

decrement

timeout

Precondition:
　$status_2 = active$
　$count_2 > 0$
　$now \geq first(dec)$
Effect:
　$count_2 := count_2 - 1$
　if $count_2 > 0$ then
　　$first(dec) := now + \ell_1$
　　$last(dec) := now + \ell_2$
　else
　　$first(dec) := 0$
　　$last(dec) := \infty$
　　$last(timeout) := now + \ell$

Precondition:
　$status_2 = active$
　$count_2 = 0$
Effect:
　$status_2 := done$
　$last(timeout) := \infty$

$\nu(t)$
Precondition:
　$now + t \leq last(send)$
　$now + t \leq last(rec)$
　$now + t \leq last(dec)$
　$now + t \leq last(timeout)$
Effect:
　$now := now + t$

23.2.3 操作

合成　我们为通用定时自动机定义一个合成操作，该操作是对已定义的MMT自动机合成操作的扩展。首先，对于一组时间签名$\{S_i\}_{i\in I}$，如果对每个i、$j \in I$且$i \neq j$，下列等式都成立时，我们则称这些时间签名是兼容的。

1）$int(S_i) \cap acts(S_j) = \phi$

2）$out(S_i) \cap out(S_j) = \phi$

对于一组GTA，当其中各GTA的时间签名都是兼容的，我们则称这一组GTA是兼容的。

我们定义有限兼容时间签名集合$\{S_i\}_{i\in I}$的合成$S = \prod_{i\in I} S_i$是满足下列条件的一个时间签名：

- $out(S) = \cup_{i\in I} out(S_i)$
- $int(S) = \cup_{i\in I} int(S_i)$
- $in(S) = \cup_{i\in I} in(S_i) - \cup_{i\in I} out(S_i)$

754　有限兼容GTA集合$\{A_i\}_{i\in I}$的合成$A = \prod_{i\in I} A_i$定义如下[⊖]：

- $sig(A) = \prod_{i\in I} sig(A_i)$
- $states(A) = \prod_{i\in I} states(A_i)$
- $start(A) = \prod_{i\in I} start(A_i)$
- $trans(A)$为三元组(s, π, s')的集合，使得对所有$i \in I$，如果$\pi \in acts(A_i)$，那么有$(s_i, \pi, s'_i) \in trans(A_i)$，否则有$s_i = s'_i$。

在一个合成中，允许那些在签名中有某个特定操作π的各组件同时参与涉及π的步，而其他所有组件不做任何事，则得到该合成的转移。需要指出的是，这表明所有组件参与时间通过步，对所有组件都有相同的时间总量通过。同样，我们有时也用中缀操作符×来表示合成。

定理23.6　一组兼容通用定时自动机的合成是一个通用定时自动机。

合成操作与*gen*　对于一组给定的兼容MMT自动机，合成操作和*gen*转换的执行顺序对结果没有影响，既可以是先进行合成操作然后对每个组件执行*gen*转换，也可以先对每个组件执行*gen*转换后再合成。最终的GTA都是一样的，直至（机器可接触部分的）同构（isomorphism）。

⊖　在$start(A)$和$states(A)$定义中使用的符号$\prod$就是指通常的笛卡儿乘积，而$sig(A)$定义中使用的符号$\prod$则指上面定义的时间签名合成操作。同样，我们在此用符号s_i表示状态向量s的第i个元素。

同样，我们可以得到类似于定理8.1~8.3的投影和粘贴定理。令$\{B_i\}_{i\in I}$为一组兼容的GTA，$B=\prod_{i\in I}B_i$。对B的某个定时运行$\alpha=s_0,\pi_1,s_1,\ldots$，删除其中$\pi_r$不是$B_i$的操作的每个二元组$\pi_r,s_r$，并将剩下的每个$s_r$用$(s_r)_i$代替，即用自动机$A_i$的状态$s_r$片段代替，所得到的结果序列用$\alpha|B_i$表示。同样，对于$B$的某个定时轨迹$\beta$（或者更通用一些，某个带有时间的操作序列），用$\beta|B_i$表示$\beta$中由包含$B_i$的操作的所有二元组所组成的子序列。

定理23.7 令$\{B_i\}_{i\in I}$为一组兼容的通用定时自动机，$B=\prod_{i\in I}B_i$。

1）如果$\alpha\in atexecs(B)$，那么对每个$i\in I$，有$\alpha|B_i\in atexecs(B_i)$。

2）如果$\beta\in attraces(B)$，那么对每个$i\in I$，有$\beta|B_i\in attraces(B_i)$。

证明 此证明留做一道习题。 □ 755

由于GTA模型允许在一个运行中出现连续的时间通过步，因此第一个粘贴定理（定理23.8）存在一个小的技术问题。具体来说，把单个的可容许定时运行α_i“粘贴在一起”得到可容许定时运行α，但对α投影可能会得不到原始的α_i，而是得到与原始α_i时间通过等价的可容许定时运行。

定理23.8 令$\{B_i\}_{i\in I}$为一组兼容的通用定时自动机，$B=\prod_{i\in I}B_i$。假设对每个$i\in I$，α_i都是B_i的一个可容许定时运行，并假设β是一个由二元组$(action, time)$组成的序列，其中的所有操作都属于$vis(B)$，使得对每个$i\in I$都有$\beta|B_i=ttrace(\alpha_i)$。那么存在$B$的一个可容许定时运行$\alpha$，使得$\beta=ttrace(\alpha)$、并且对每个$i\in I$都有$\alpha_i$与$\alpha|B_i$是时间通过等价的。

证明 此证明留做一道习题。 □

定理23.9 令$\{B_i\}_{i\in I}$为一组兼容的通用定时自动机，$B=\prod_{i\in I}B_i$。假设β是由二元组$(action, time)$所组成的一个序列，其中β的所有操作都属于$vis(A)$。如果对每个$i\in I$都有$\beta|B_i\in attraces(B_i)$，那么$\beta\in attraces(B)$。

证明 此证明留做一道习题。 □

隐藏 如果A是一个GTA并且$\Phi\subseteq out(A)$，那么$hide_\Phi(A)$是一个GAT，$hide_\Phi(A)$除了将Φ中的操作重新归类为内部操作外，与A完全相同。

23.3 属性和证明方法

对于基于计时的算法和系统，其正确性和性能常常严重依赖于对时间的假设。与在异步环境中不同，对计时假设的很小改动，也可能对基于计时算法的行为产生重大影响。然而，分析这种时间依赖性的原因却格外困难，即使是对于本章所给例子中的那些非常简单的“算法”也是如此。在这种环境中，系统的证明方法会有很大的帮助。

在本节，讲述两个用于基于计时算法的重要证明方法：不变式断言法和模拟关系法。由于这些方法已经成功地用于同步和异步环境，因此很自然地要试图在基于计时的环境中采用它们。我们还定义一个定时轨迹属性的概念，类似于8.5.2节所介绍的轨迹属性的概念。 756

23.3.1 不变式

我们将通用定时自动机A的一个不变式断言定义为一个对A的全部可到达状态都为真的属性。

从形式上看，这个定义与我们在异步环境中使用那个定义相同。但它们之间是有差别的：在一个异步系统中，状态通常由普通数据（如局部变量和共享变量值、通道中的在途消息序列）组成。但是在一个基于计时的系统中，状态通常还包括一些与时间相关的信息，如当前时间以及未来事件的预定期限等。例如，如果一个消息正在通道中传输，状态可能包括相关的时间信

息，以给出完成该消息传递的未来时间范围。这说明在计时环境中，不变式断言除包括普通数据外，还可能涉及与时间有关的信息。

虽然在计时环境中，状态中包含的信息类型更丰富，但是不变式的证明方法还和以前一样——归纳法。这里是对导致问题中状态的那个定时运行的步数量进行归纳。

需要提醒的是，我们是按照通用定时自动机来介绍不变式断言法的。如果我们想将此证明方法用于MMT自动机，则必须先将MMT自动机转化成GTA。

例 23.3.1　超时系统的不变式

考虑例23.2.4的超时系统A'，并假定 $kl_1 > l + d$ 。最好能够证明在所包含的进程P_1确实已经死亡的情况下，系统只执行一个*timeout* 。下面的不变式可以证明这一点。

断言 23.3.1　在A'的任何可到达状态中,如果$status_1 = alive$, 那么 $count_2 > 0$。

不幸的是，像通常一样，仅用归纳法还无法证明断言23.3.1——需要辅助断言。在这种情况下，首先（通过一个简单的归纳）证明下面的断言。

断言 23.3.2　在A'的任何可到达状态中,如果$status_2 = done$，那么$count_2 = 0$。

然后我们用一个稍微复杂一点的归纳，证明下面这个断言23.3.1的加强版本。这个断言涉及关于状态的时间组件*first*和*last*的陈述。

断言 23.3.3　在A'的任何可到达状态中,如果 $status_1 = alive$, 那么下面结论成立：

757

1）$count_2 > 0$

2）或者 $last(send) + d < first(dec) + (count_2 - 1)\, l_1$、*queue*非空，或者$status_2 = disabled$。

3）如果 *queue*非空，那么或者$last(rec) < first(dec) + (count_2 - 1)\, l_1$，或者$status_2 = disabled$。

条件1只是断言23.3.1的重述。条件2和条件3分别在一个不等式中使用表达式$first(dec) + (count_2 - 1)\, l_1$ 。这个表达式描述$count_2$可能到达0的最早时间，且假设$count_2$当前为正。换句话说，$first(dec)$ 是下一个*decrement*的最早时间，并且需要另外$count_2 - 1$次递减操作才能使得$count_2$为0，而l_1 是每次递减操作的最小时间。条件 2 的意思是，要么按计划要发送一个消息，并在$count_2$达到0前有充足的时间到达，要么一个消息已经正在传送途中，要么就是一个消息已经到达（从而阻止超时操作）。条件3的意思是，如果一个消息正在传送途中，那么或者在$count_2$达到0之前某个消息将会到达，或者消息已经到达。因此，用状态的*first*和*last*期限组件，一些关于计时事件的断言可以简洁地表示成不变式。

通过对一个定时运行中的操作数量进行归纳，可以证明断言 23.3.3。这一证明比较简单（实际上是琐碎的），我们之所以在这里阐述它，是因为它为其他类似证明提供了一个好的样板。

基础：初始状态时，$count_2 = k > 0$, *queue*为空，且$first(dec) = l_1$ 。根据这些事实知在初始状态下条件1 和 3成立。此外，如果$status_1 = alive$，那么 $last(send) = l$。因此：

$$last(send) + d = l + d < k\, l_1 = count_2\, l_1 = first(dec) + (count_2 - 1)\, l_1$$

这说明条件2在初始状态下也成立。

归纳步：像通常一样，我们基于不同的操作类型进行分类分析，只是在这一证明中，还需要有对计时通过操作$\nu(t)$的分析。假设$(s, \pi, s') \in trans(A')$ 并且s 满足不变式条件。假设$s'.status_1 = alive$；那么$s.status_1 = alive$也成立。

1）$\pi = send(m)_{1,2}$

那么 $s.first(dec) = s'.first(dec)$、$s.count_2 = s'.count_2$、$s.status_2 = s'.status_2$。此步对条件1无影响

并使条件2为真。我们考虑条件3。如果$s.queue$非空，那么$s.last(rec) = s'.last(rec)$，因此只要$s$的条件3成立，$s'$的条件3 必定成立。 758

假设$s.queue$ 为空。那么根据归纳假设（条件2），或者$s.last(send) + d < s.first(dec) + (s.count_2 - 1)\ l_1$、或者$s.status_2 = disabled$ 。 若是后一种情况，则证明结束，因此假设是前一种情况。那么根据引理23.5，$s'.last(rec) = s.now + d \leqslant s.last(send) + d$， $< s.first(dec) + (s.count_2 - 1)\ l_1 = s'.first(dec) + (s'.count_2 - 1)\ l_1$，结论得证。

2）$\pi = receive(m)_{1,2}$

根据归纳假设，$s.count_2 > 0$。此步对该陈述无影响，因此条件1成立。同样，根据断言23.3.2 知$s.status_2 \neq done$。因此，$s'.status_2 = disabled$，从而表明条件2和条件3成立。

3）$\pi = timeout$

根据归纳假设，$s.count_2 > 0$。因此$timeout$在s中没有被使能，这说明此种情况不可能发生。

4）$\pi = decrement$

我们用反证法证明条件1。如果$s'.count_2 = 0$，那么$s.count_2 = 1$，因此根据归纳假设（条件2和条件3），或者$s.last(send) + d < s.first(dec)$、$s.last(rec) < s.first(dec)$，或者$s.status_2 = disabled$。根据$decrement$的前提条件知后一种情况是不可能的。因此我们有：

$\min(s.last(send), s.last(rec)) < s.first(dec) \leqslant s.now$

但是根据引理23.5知$s.now \leqslant s.last(send)$、$s.now \leqslant s.last(rec)$，因此$s.now \leqslant \min(s.last(send), s.last(rec))$。这是一个矛盾。

对于条件2和条件3，只要证明$first(dec) + (count_2 - 1)\ l_1$的值在此步不减少。由于其中的第二个谓词正好减少$l_1$，而第一个谓词至少增加$l_1$，因此条件2和条件3得证。（这是因为$s.first(dec) \leqslant s.now$且$s'.first(dec) = s.now + l_1$。）

5）$\pi = \nu(t)$

此步对三个条件中的任何一个都无影响，因为只有now被改变，而三个条件都未涉及now。

23.3.2 定时轨迹属性

请大家回忆一下，为异步系统所证明的许多属性，都可以自然地延伸为这些系统的轨迹属性或公平轨迹属性。与此类似，定时系统的许多令我们感兴趣的属性，也可以延伸为它们的可容许定时轨迹的属性。可用这种方法描述的属性包括性能属性和普通正确性属性。 759

定义一个*定时轨迹属性*P 的组成如下：

- $sig(P)$，一个不包含内部操作的时间签名
- $ttraces(P)$，一组由二元组(action, time)所组成的序列的集合，每个序列中的时间组件必须是单调非递减的，并且，当该序列是无限的时候，时间组件也必须是无界限的。

我们常将这一阐述解释为，如果说GTA A满足轨迹属性P，则意味这 $in(A) = in(P)$、$out(A) = out(P)$以及$attraces(A) \subseteq ttraces(P)$。

例 23.3.2 定时轨迹属性

令P 为如下定义的定时轨迹属性。签名$sig(P)$为：

Input:	Internal:
$receive(m)_{1,2}$, $m \in M$	none
Output:	Time-passage:
$timeout$	$\nu(t)$, $t \in R^+$

定时轨迹集合$ttraces(P)$是一组由二元组(action, time)所组成序列β的集合，β满足单调性和有界限性条件、并且使得下列结论成立

1）如果在β中有一个二元组$(timeout, t)$，那么$kl_1 \leqslant t \leqslant kl_2 + l$。

2）如果在β中有一个*timeout*对，那么其前面没有 *receive* 对。

3）如果在β中没有*receive* 对，那么在β中有一个*timeout*对。

那么$gen(P_2)$满足定时轨迹属性P，使得$attraces(gen(P_2)) \subseteq ttraces(P)$，其中$P_2$是例23.1.2中的MMT自动机。

23.3.3 模拟

模拟方法既可用于基于计时的系统的推理，也可用于同步和异步系统的推理。为此，我们定义两个通用定时自动机的状态之间的“定时模拟关系” 的概念。该定义与8.5.5节中I/O自动
760 机的模拟关系定义非常类似。

令A和B为有同样输入和输出操作的两个通用定时自动机。假设f是在$states(A)$和$states(B)$上的一个二元关系；我们以符号$u \in f(s)$作为$(s, u) \in f$的另一种表示法。那么如果下面两条件为真，则f是一个从A到B的定时模拟关系：

1）如果$s \in start(A)$，那么$f(s) \cap start(B) \neq \phi$

2）如果s是A的一个可到达状态，$u \in f(s)$是B的一个可到达状态，并且$(s, \pi, s') \in trans(A)$，那么存在一个以$u$开始、以某个$u' \in f(s')$结束的定时运行片断$\alpha$，使得

a）$ttrace(\alpha) = ttrace(s, \pi, s')$。

b）α中的时间通过总量与(s, π, s')中的时间通过总量相同。

因而，该起始条件与I/O自动机模拟关系的起始条件相同。步条件略有不同——我们现在要求对方保存定时轨迹，即可见操作的序列，每个操作都带有自己的发生时间，再加上时间通过的总量。需要指出的是，在步条件中，π既可以是一个时间通过操作也可以是一个离散操作。如果π是一个可见操作，那么α一定有一个π步，也可能还有一些在前驱的和/或在后继的内部步。如果π是一个内部操作，那么α必定只由内部步组成。如果$\pi = \nu(t)$，那么α中必定有t个时间通过步，而且这些时间通过步被内部步两两隔开。

同前面一样，由于假设步条件的状态s和u是可到达的，所以在证明f是一个定时模拟关系时，可以使用关于状态A和B的不变式断言。

下面的定理给出定时模拟关系的关键属性。

定理 23.10　如果存在一个从A到B的定时模拟关系，那么$attraces(A) \subseteq attraces(B)$。

证明　该证明留做一道习题。　□

在本节的剩余部分，我们用例子说明如何用定时模拟来证明定时系统的属性。这种模拟的一个有趣应用是为有时间假设的系统证明时间界限。方法是将时间说明规范形式化为一个GTA B，
761 并用 *last* 和 *first* 期限组件来表达所要求的定时行为 (分别为上界和下界)。也将时间说明规范的实现形式化为一个GTA A，并用 *last* 和 *first*组件表示时间假设。从A到B的定时模拟的存在意味着A满足时间需求。

由于在定时环境中可以用模拟来证明定时属性，因此模拟方法在定时环境中比在异步环境中更有效。在异步环境中，我们经常关注*活性属性*，在定时环境中，我们关注更多的是*时间界限*。在对活性属性进行严格证明时，除采用模拟方法外，还采用像时间逻辑这样的特定机器，但时间界限只要用模拟方法就可以证明。

例 23.3.3　对超时进程的时间界限模拟证明

我们将说明，如果例23.1.2中的超时MMT自动机P_2没有收到消息，则必定在$[kl_1, kl_2 + l]$间隔内执行一个*timeout* 。为简化问题，我们定义P_2的一个变量A，在该变量的签名内甚至没有一

个*receive* 操作。*A*的代码如下。

A automaton:

Signature:

Input:
none
Internal:
decrement
Output:
timeout

States:

$count \in \mathbb{N}$, initially k
$status \in \{active, done\}$, initially *active*

Transitions:

decrement
Precondition:
$status = active$
$count > 0$
Effect:
$count := count - 1$

timeout
Precondition:
$status = active$
$count = 0$
Effect:
$status := done$

Tasks and bounds:

$dec = \{decrement\}$, bounds $[\ell_1, \ell_2]$
$timeout = \{timeout\}$, bounds $[0, \ell]$

762

那么 *gen*(*A*) 的代码如下。

---

gen(*A*) automaton:

Timed Signature:

Input:
none
Internal:
decrement
Output:
timeout
Time-passage:
$\nu(t)$, $t \in R^+$

States:

$count \in \mathbb{N}$, initially k
$status \in \{active, done\}$, initially *active*

$now \in R^{\geq 0}$, initially 0
$first(dec) \in R^{\geq 0}$, initially ℓ_1
$last(dec) \in R^+ \cup \{\infty\}$, initially ℓ_2
$last(timeout) \in R^+ \cup \{\infty\}$, initially ∞

Transitions:

decrement
Precondition:
$status = active$
$count > 0$
$now \geq first(dec)$
Effect:

timeout
Precondition:
$status = active$
$count = 0$
Effect:
$status := done$

$count := count - 1$
if $count > 0$ then
　$first(dec) := now + \ell_1$
　$last(dec) := now + \ell_2$
else
　$first(dec) := 0$
　$last(dec) := \infty$
　$last(timeout) := now + \ell$

　$last(timeout) := \infty$

$\nu(t)$
　Precondition:
　　$now + t \leq last(dec)$
　　$now + t \leq last(timeout)$
　Effect:
　　$now := now + t$

自动机 A 只简单地从k到0进行递减计数，然后执行一个 *timeout*。显然，在所断言的时间间隔$[kl_1, kl_2 + l]$内会发生惟一的一个*timeout*。为形式化地证明这一点，我们使用一个简单的高级GTA来表达这些定时需求。这个GTA的形式为$gen(B)$，其中B是下面的简单MMT自动机。

763

B automaton:

Signature:

Input:　Output:
　none　　*timeout*

States:

$status \in \{active, done\}$, initially *active*

Transitions:

timeout
　Precondition:
　　$status = active$
　Effect:
　　$status := done$

Tasks and bounds:

$timeout = \{timeout\}$, bounds $[k\ell_1, k\ell_2 + \ell]$

现在我们构造一个从$gen(A)$到$gen(B)$的定时模拟关系f，从而说明 A 满足定时需求。如果s和u分别是$gen(A)$和 $gen(B)$的状态，那么当下列条件成立时，我们称$(s, u) \in f$：

1. $s.now = u.now$.
2. $s.status = u.status$.
3. $u.last(timeout) \geq$
$$\begin{cases} s.last(dec) + (s.count - 1) \cdot \ell_2 + \ell & \text{if } s.count > 0, \\ s.last(timeout) & \text{otherwise.} \end{cases}$$
4. $u.first(timeout) \leq$
$$\begin{cases} s.first(dec) + (s.count - 1) \cdot \ell_1 & \text{if } s.count > 0, \\ s.first(timeout) & \text{otherwise.} \end{cases}$$

涉及*now* 和 *status* 值的那些关系是很明显的。我们所感兴趣的是涉及*last* 和*first* 期限的关系。要求$u.last(timeout)$ 值(在$gen(B)$中)不小于一个特定量，该特定量是根据$gen(A)$的状态（包括期限组件）计算的。该特定量是$gen(A)$最后一次可能执行*timeout*的计算时间上界。有两种情况：

如果 $count > 0$，那么这个时间的界限为可发生第一个*decrement* 的最后时间，加上执行$count - 1$个*decrement*步以及随后的一个*timeout*步所需要的附加时间；由于每个 *count* 步所需要的时间不超过l_2而*timeout* 需要的时间不超过l，因此整个附加时间至多为 $(count - 1) \cdot l_2 + l$。另一方面，如果$count = 0$，那么这个时间由*timeout* 可能发生的最后时间限定。这个不等式表达的事实是，所计算出的到*timeout*为止的实际时间界限最多等于所要证明的上界。 764

对*first*(*timeout*) 不等式的解释是对称的——*first*(*timeout*)的值应该不大于到*gen*(*A*)执行一个*timeout*操作为止的最早计算时间下界。

为了证明 f 是一个定时模拟，我们首先证明一个简单的不变式。

断言23.3.4 在 *gen*(*A*)的任何可到达状态中，如果$count > 0$，那么$status = active$。

下面按照模拟的通常方式继续证明，证明起始条件和步条件。对不等式的处理方式与状态间其他任意类型关系的处理方式相同。像在例23.3.1中一样，为给其他类似证明树立样板，我们在证明中讲述部分细节；其余的细节留做习题。

对于起始条件，令s和u 分别为*gen*(*A*) 和 *gen*(*B*)的惟一起始状态。我们必须证明$u \in f(s)$。f定义的条件1和条件2显然成立。考虑条件3。根据*gen*(*B*) 的定义可知，$u.last(timeout) = kl_2 + l$，根据*gen*(*A*)的定义知$s.count > 0$ 和 $s.last(dec) + (s.count - 1) \cdot l_2 + l = l_2 + (k - 1)l_2 + l = kl_2 + l$。因此 $u.last(timeout) = s.last(dec) + (s.count - 1) \cdot l_2 + l$，从而证明条件3。条件4与条件3类似。

对于步条件，我们假设 $(s, \pi, s') \in trans(gen(A))$，$s$ 是可到达的，且 u 是$f(s)$的可到达状态。我们基于操作类型考虑不同的情况，包括时间通过操作。

例如，考虑π= *decrement*的情况。根据*decrement*的前提条件可知 $s.count > 0$。根据 $u \in f(s)$ 这一事实可知$s.now = u.now$、$s.status = u.status$、$u.last(timeout) \geq s.last(dec) + (s.count - 1) \cdot l_2 + l$、以及$u.first(timeout) \leq s.first(dec) + (s.count - 1) \cdot l_1$。从而可证明$u \in f(s')$。

对条件1和条件2无需进一步证明。假设$s'.count > 0$。对于条件3，该步未改变不等式的左端*last*(*timeout*)，而不等式的右端也没有增加。后一属性之所以为真，原因在于*last*(*dec*)最多增加l_2，而第二个谓词正好减少l_2、第三个谓词不变。*last*(*dec*)最多增加l_2的原因在于$s.now \leq s.last(dec)$ 和$s'.last(dec) = s.now + l_2$。）这表明在这个步之后不等式仍然成立。可以对条件4和$s'.count = 0$的情况做类似证明。 765

对其他类型的操作，可以采用同样的风格进行相应的证明。对于π= *timeout*的情况，需要证明的是状态 u满足前提条件$first(timeout) \leq now$。这个不等式之所以成立，原因在于：由*gen*(*A*)中π的前提条件知 $s.first(timeout) \leq s.now = u.now$，由条件4知$u.first(timeout) \leq s.first(timeout)$。

对于π是时间通过操作的情况，需要证明的是前提条件$u'.now \leq u.last(timeout)$。这个不等式之所以成立，原因在于：由*gen*(*A*)中π 的前提条件知$u'.(now) = s'.now \leq \min(s.last(dec), s.last(timeout))$，由条件3知$\min(s.last(dec), s.last(timeout)) \leq u.last(timeout)$。时间通过步不改变$f$定义中除*now*外的任何事情，因此时间通过步显然保留f中的所有关系。

由于f是一个定时模拟，因此定理23.10 表明 $attraces(gen(A)) \subseteq attraces(gen(B))$，定理23.4表明$attraces(A) \subseteq attraces(B)$。这说明 A 满足定时需求。

当然，除了使用定时模拟外，还有其他方式可以证明基于定时系统的时间界限。例如，习题23.13中基于不变式的可操作参数，可以用来证明到*timeout*发生为止的时间的上界$kl_2 + l$。

例 23.3.4 双任务轨迹

本例采用模拟法概要地证明下列结论：$l + l_2 + Ll$是到例23.1.3的*Race*自动机执行一个*report*

输出为止的时间的上界。规范说明为$gen(B')$，其中 B' 是一个类似于例23.3.3中说明自动机B的
766 MMT自动机。

B' automaton:

Signature:

Input: none　　Output: *report*

States:

reported, a Boolean, initially *false*

Transitions:

report
　Precondition:
　　reported = *false*
　Effect:
　　reported := *true*

Tasks and bounds:

report = {*report*}, bounds $[0, \ell + \ell_2 + L\ell]$

凭直觉，$l + l_2 + Ll$ 是一个正确上界的理由如下。在时间l内，任务 *int* 设置 *flag* 为 *true*。在这期间，*count*可到达的最大值为 $\frac{l}{l_1}$。因此，*main*任务将*count*递减为0所需要的最大时间量为 $\frac{l}{l_1} l_2 = Ll$，另外最多还需要时间l_2来执行*report*。

现在，定义一个从$gen(Race)$到$gen(B')$的定时模拟关系g。如果s 和u 分别是$gen(Race)$ 和 $gen(B')$ 的状态，那么如果下面的条件成立，我们称$(s, u) \in g$：

1. $s.now = u.now$.
2. $s.reported = u.reported$.
3. $u.last(report) \geq$
$$\begin{cases} s.last(int) + (s.count + 2)\ell_2 + L(s.last(int) - s.first(main)) \\ \quad \text{if } s.flag = false \text{ and } s.first(main) \leq s.last(int). \\ s.last(main) + (s.count)\ell_2 \\ \quad \text{otherwise.} \end{cases}$$

第三个条件的思想如下。如果*flag* = *true*，那么到*report*为止的剩余时间刚好是任务*main* 执行剩
767 余的*decrement* 步以及随后的最终*report*的时间。如果*flag* 仍为 *false*，推理同样成立，但是要求 *flag*在另一个*increment*发生前变为*true*，即$s.first(main) > s.last(int)$。否则，$s.flag = false$和 $s.first(main) \leq s.last(int)$成立，这意味着剩下的时间足够至少一个*increment*发生。那么就变成 *last*(*report*)不等式的第一种情况。

在这种情况下，在*set*之后，任务*main*将目前的*count* 倒计数、然后*report*所需要的全部时间可能长达$(count + 1)\, l_2$。但是在*set*之前，当前的*count*可能由于某些额外的*increment*事件而增

加。这些可能发生的*increment*事件的最大数量为$1 + [last(int) - first(main)]/ l_1$。将这个数乘以$l_2$，即得出递减这些额外记数需要的最大时间。

证明 g 是一个定时模拟关系的方法与在例23.3.3使用的通用证明方法相同。我们将其留做一道习题。

23.4 构造共享存储器和网络系统的模型

在本章的结尾，简要说明如何使用MMT自动机和GTA自动机来构造部分同步共享存储器系统和部分同步网络系统的模型。我们将在第24章和第25章使用这些模型。

23.4.1 共享存储器系统

我们将一个部分同步共享存储器系统模型化为一个MMT自动机(A, b)。在此按照第9章的定义，假设I/O自动机A是一个异步共享存储器系统；惟一新增的约束是A只有有限个任务。界限图 b 为每个任务增加时间界限。

大多数时候，我们将假设每个进程只有一个任务，界限图为每个任务指派一个下界l_1和一个上界l_2，其中$0 < l_1 \leqslant l_2 < \infty$。在这种情况下，我们将记$L = l_2/ l_1$；同前面一样，$L$ 是衡量系统中时间不确定性的依据。

23.4.2 网络

在部分同步环境中，我们将只考虑发送/接受网络，而不考虑广播或多播系统。假设采用有向图$G = (V, E)$。我们将一个部分同步发送/接收网络系统模型化为一个进程自动机集合和一个通道自动机集合，其中每个进程自动机对应一个顶点、每个通道自动机对应一条边。 768

与每个顶点i所对应的进程自动机是一个MMT自动机P_i。P_i有输入和输出操作，通过这些操作与外部用户进行通信，还有形式为$send(m)_{i,j}$的输出，其中m是一个消息，j是流出邻接节点，P_i还有形式为$receive(m)_{j,i}$的输入，其中j是流入邻接节点。为抽象表示进程停止失败，我们在P_i中定义一个$stop_i$输入操作。这个操作的结果是永久停止P_i的所有任务。通常，我们将假设每个进程P_i都有其中各任务（任务的总数是有限的）的时间界限l_1和l_2，其中$0 < l_1 \leqslant l_2 < \infty$。

与每条有向边(i, j)所对应的通道自动机是一个GTA $C_{i,j}$。它的“可见接口”由$send(m)_{i,j}$形式的输入和$receive(m)_{i,j}$形式的输出组成。对通道外部行为的限制用定时轨迹属性P表示，P所定义的通道是那些具有与P相同的可见操作、且其可容许定时轨迹属于$ttraces(P)$的GTA。有两种常见情况：

1）每个$C_{i,j}$是具有例23.2.1所描述的合适时间签名的GTA $D'_{i,j}$，即，一个对最早消息的递送时间有上界d的可靠FIFO通道。

2）每个$C_{i,j}$是具有例23.2.2所描述的合适时间签名的GTA $D''_{i,j}$，即，一个对每个消息的递送时间都有上界d的可靠FIFO通道。

同样，我们记$L = l_2/ l_1$并将L用作衡量系统中时间不确定性的依据。

23.5 参考文献注释

MMT定时自动机模型是由 Merritt、Modugno和 Tuttle[227]设计的。他们的模型要比我们在本书中使用的模型略微通用一些，具体表现在他们的模型中既可以有最终上界也可以有基于实时值的上界。我们所使用模型的变量与Lynch 和 Attiya[215]所定义的比较接近。双任务轨迹例子是作为定时系统证明方法的测试实例由Pnueli[243]提出的。

通用定时自动机模型基于Lynch 和 Vaandrager[210, 212, 211]的定时自动机模型；它类似于Alur 和 Dill[9]的定时自动机模型。与可容许定时运行的存在性有关的问题是由Gawlick、

Segala、Sφgaard-Andersen和Lynch[136]研究的。从MMT自动机到通用定时自动机的转换方法
769 是Lynch和Attiya[215]开发的。GTA的操作取自[212]；该文章除描述合成和隐藏操作外，还描述GTA的许多其他操作，包括连续合成和多种形式的选择、中断以及超时。

包括计时期限在内的不变式已经被Tel[275]、Lewis [194]、Shankar [259]、Abadi与Lamport [1]、Lynch [204]和其他人使用过。证明时间属性的模拟方法是首先由 Lynch 和Attiya[215]使用的。在例23.3.3和例23.3.4中使用的时间界限的模拟证明方法源自[215]以及综述文章 [204, 205]。GTA的其他模拟类型是由Lynch 和 Vaandrager[210, 211]定义的。

还有一些其他的定时模拟证明方法，分别是由下述人员开发的：Sφgaard-Andersen、Lampson和 Lynch [264, 190]、Heitmeyer 和Lynch [148]以及Luchangco [201]。在使用自动定理证明器来帮助检验和执行定时模拟证明方面，Luchangco、 Söylemez、Garland和Lynch做了一些初步工作[202]。该工作使用的是Larch 证明器[134]。

23.6 习题

23.1 令(A, b)为任意的 MMT自动机，α为(A, b)的某个有限定时运行。证明下列结论：

a）存在(A, b)的一个（可容许）定时运行，该定时运行从α开始。

b）令β为某个由带有时间的输入操作组成的有限序列，其中时间是非递减的，并且不大于α中发生的最大时间。那么存在 (A, b)的一个（可容许）定时运行α'，使得 α'以α开始，并使得 β是α'中在α后发生的输入操作及相关时间的子序列。

c）设β为某个由带有时间的输入操作组成的无限序列，其中时间是非递减的、无界限的，并且至少与α中的最大发生时间同样大。那么存在(A, b)的一个（可容许）定时运行α'，使得α'开始于α，并使得β是由α'中在α后发生的输入操作及相关时间所组成的子序列。

23.2 假设对MMT自动机的定义进行弱化，将其中的有限多个任务用可数数量的任务代替。证明存在一个满足这个新定义的自动机(A, b)，其中(A, b)没有（可容许）定时运行。

770 23.3 在$kl_1 \leq l + d$的情况下，详细描述例23.1.4中的合成MMT自动机的行为。

23.4 证明定理23.1、定理23.2和定理23.3。

23.5 考虑15.4节中*AsynchBellmanFord*算法的时间界限变量，其中：

- 每个进程自动机是由给定I/O自动机和每个任务的界限$[l_1, l_2]$所组成的MMT自动机，其中$0 < l_1 \leq l_2 < \infty$。
- 每个通道自动机是例23.1.1中的相应MMT自动机$D_{i,j}$。

分析最终算法的通信和时间复杂性。

23.6 证明例23.2.1中的GTA $D'_{i,j}$和例23.1.1中的MMT自动机$D_{i,j}$有同样的可容许定时轨迹集合。

23.7 证明不存在与例23.2.2中GTA $D''_{i,j}$有同样的可容许定时轨迹集合的MMT自动机。

23.8 给出具有下列行为的GTA A的前提－结果代码。在任意一个可容许定时运行中，A 只执行两个输出a 和 b，且都在时间1时执行，执行顺序为先a后b。而且，只要符合给定的限制，在不同的可容许定时运行中，A 应该允许a和b 在任何时间发生。证明不存在与A有同样可容许定时轨迹集合的MMT自动机。

23.9 给出变换自动机*gen*(*Race*)的详细前提－结果代码，其中*Race*是例23.1.3中定义的MMT自动机。代码要采用例23.2.4中代码的类似格式。

23.10 证明定理23.7、定理23.8 和定理23.9。

23.11 对定理23.8进行简化改写，以断言$\alpha_i = \alpha | B_i$代替它们是时间通过等价的，证明这一简化后的版本不成立。

23.12 证明定理23.10。

23.13 对例23.2.4中的系统A'，证明下面的多部分不变式。如果$status_1 = dead$，那么

a）*queue* 为空。

b）$status_2 \neq disabled$。

c）如果$count_2 > 0$，那么 $last(dec) + (count_2 - 1)\, l_2 \leq kl_2$。 771

d）如果$count_2 = 0$，那么$last(timeout) \leq kl_2 + l$。

23.14 使用23.3.3节的模拟方法证明引理11.3：*RightLeftDP* 算法的时间界限。

23.15 在例23.3.3中关于f是一个定时模拟关系的证明中，添加证明的具体细节。

23.16 证明：例23.3.4中定义的关系g 是一个定时模拟关系。 772

第24章 部分同步的互斥

本章中，将第三次讨论互斥问题，这次是在部分同步共享存储器环境下讨论。我们只给出最基本的结果：简单的基于定时的算法及其分析和简单的不可能性结果。

24.1 问题

这一环境与第10章的基本相同——一个有n个端口、与用户U_1、...、U_n交互的共享存储器系统。由try_i、$crit_i$、$exit_i$和rem_i操作组成的外部接口与前面的完全一样。但这里的用户和共享存储器系统都被模型化为23.1节中所定义的MMT 自动机，而不是I/O 自动机。图10-4仍然可用来表示本章中我们所关心的体系结构。

同前面一样，要求每个用户 U_i 必需有良构性。我们允许对用户的任意时间约束。具体说，每个MMT 自动机 U_i 形式为(A_i, b_i)，其中A_i是在10.2节中所允许的某个I/O自动机、并且只有有限多个任务，b_i 是一个任意的界限图。这些可用界限图中所包含的是简单界限图，以给出简单低端界限0和简单高端界限∞ 。

系统的余下部分由一个代表该共享存储器系统的MMT 自动机 $B = (A, b)$组成。B所采用的底层I/O自动机A 的形式同第10章中我们为解决异步共享存储器模型的互斥问题所考虑的自动
773 机。具体说，它由n个进程组成，每个进程位于一个端口。本章中我们假定每个进程只有一个任务。界限图 b 为每个任务配一个下界限l_1 和一个上界限l_2，其中$0 < l_1 \leq l_2 < \infty$ 。同前面一样，我们记$L = l_2 / l_1$；L是一种对系统中计时不确定性的度量。

在本章还做了其他三个约束。首先，用与第10章同样的方法对进程的活性进行限制：在共享存储器系统中，只有当U_i在尝试区或退出区时，每个进程i惟一的任务才能被激活。第二，假设当U_i 在尝试区或退出区时，进程i 惟一的任务事实上总是处于被激活状态。（然而，我们允许这样的可能性存在，即这个惟一被激活的操作可以是不引起任何状态改变的虚操作。）第三，只考虑有读/写共享变量的共享存储器系统。

我们所需要的正确性条件与第10章中的正确性条件基本相同。针对定时自动机重述这些条件，可得到：

良构性条件 在组合系统的任何定时运行中，对任意的i，描述U_i 和 $B = (A, b)$ 之间交互的子序列对i都是合式的。

互斥性条件 不存在超过一个用户位于临界区C的可达到系统状态。

演进性条件 在一个可容许定时运行的任何点[⊖]，

1）（尝试区的演进） 如果在T中至少有一个用户，而在C中没有用户，那么在其后的某一点会有用户进入C。

2）（退出区的演进）如果E中至少有一个用户，那么在其后的某一点会有用户进入R。

如果B为每个用户集都解决了互斥问题（即保证良构性条件、互斥性条件和演进性条件），我们则说B解决了互斥问题。同23.3.2节中定义的一样，这些正确性条件也可以用定时轨迹属性P的形式来表达。

⊖ 同23.1节，这一定义的目的是为了表示时间正常过去，以及如果有更多的工作要做，处理进度就不停。

24.2 单寄存器算法

在本节，介绍一个部分同步互斥算法，*FischerME* 算法，该算法只用一个读/写寄存器。这个简单的算法已表明部分同步模型与异步模型有很大不同，因为正如我们在定理10.33中阐述的那样，任何异步读/写共享存储器的互斥算法至少需要n个共享寄存器。 774

我们以下述的不正确异步算法作为阐述该算法的起点。

IncorrectFischerME 算法（非形式化的）：

算法使用单个读/写共享变量*turn*，该变量对所有进程都是可读、可写的。每个希望获得该资源的进程i重复测试*turn*，直到发现其值等于0。当进程i发现$turn = 0$后，它设置*turn*为自己的索引i。然后它检验*turn*是否仍等于i。如果是，进程i进入临界区，否则它回到开始，继续测试是否$turn = 0$。当一个进程i退出时，它将*turn*重置为0。

采用第10章中用于共享存储器程序的格式，该算法书写如下：

***IncorrectFischerME* algorithm (formal):**

Shared variables:
$turn \in \{0, 1, \ldots, n\}$, initially 0, writable and readable by all processes

Process i:

```
        ** Remainder region **

        try_i
L:      if turn ≠ 0 then goto L
        turn := i
        if turn ≠ i then goto L
        crit_i

        ** Critical region **

        exit_i
        turn := 0
        rem_i
```

IncorrectFischerME 算法是不正确的，因为它没有保证互斥。（我们知道它一定不正确，因为否则就会违反定理10.33。） 775

例 24.2.1 *IncorrectFischerME* 的有害运行

考虑一个运行，其中有两个进程1和2都测试*turn* 并发现$turn = 0$。接着，进程1设置$turn := 1$ 并立刻检验发现 $turn = 1$。然后进程2 设置 $turn := 2$，并立刻检验发现$turn = 2$。于是进程1和进程2都进入临界区。这一运行如图24-1所示。

为避免这种有害的事件交叉，我们可以增加一个简单的定时限制。具体做法是，任何设置$turn := i$的进程i 可以延迟其校验*turn*的时间，所延迟的时间大于假定的进程步时间上界l_2。所有其他步以正常速度执行，同一进程中连续步间隔时间的范围为$[l_1, l_2]$。这一限制防止图24-1中出现的有害交叉：任何设置$turn := i$的进程 i 在检验前要等足够的时间，以确保其他任何在i设置*turn*之前测试*turn*的进程j（因而随后能设置*turn*为自己的索引）已经将*turn*设置为其索引。

也就是说，当i最终校验$turn$时，没有进程还在执行设置$turn$的操作。

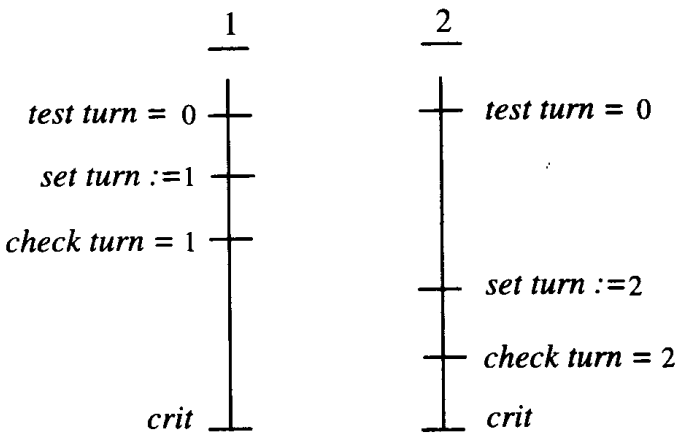


图24-1　*IncorrectFischerME*算法的有害执行

这里是前提－结果代码。在这一代码中，我们假设a_1和a_2是两个正实数，且$l_2 < a_1 \leq a_2$。需要指出的是，这个代码为每个进程i定义两个任务：界限为$[l_1, l_2]$的任务 $main_i$和界限为$[a_1, a_2]$的任务$check_i$。从技术角度看，这种做法是该模型所不许可的，模型只允许每个进程有一个界限为$[l_1, l_2]$的任务。但是，我们可以对这个算法做简单的修改：在任意*check*前插入$k-1$个显式的
776 *delay*步，其中$kl_1 > l_2$，并将每个进程的所有操作放入一个界限为$[l_1, l_2]$的任务中。该算法的这一修改版本“行为上类似于” $a_1 = kl_1$ 及$a_2 = kl_2$ 的*FischerME* 算法。我们忽略形式上的细节。

***FischerME* algorithm:**

Shared variables:
$turn \in \{0, 1, \ldots, n\}$, initially 0

Actions of i:

Input:	Internal:
try_i	$test_i$
$exit_i$	set_i
Output:	$check_i$
$crit_i$	$reset_i$
rem_i	

States of i:
$pc \in \{rem, test, set, check, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially *rem*

Transitions of i:

try_i
　Effect:
　　$pc := test$

$test_i$
　Precondition:
　　$pc = test$
　Effect:
　　if $turn = 0$ then $pc := set$

set_i

$crit_i$
　Precondition:
　　$pc = leave\text{-}try$
　Effect:
　　$pc := crit$

$exit_i$
　Effect:
　　$pc := reset$

$reset_i$

Precondition:
$pc = set$
Effect:
$turn := i$
$pc := check$

Precondition:
$pc = reset$
Effect:
$turn := 0$
$pc := leave\text{-}exit$

$check_i$
Precondition:
$pc = check$
Effect:
if $turn = i$ then $pc := leave\text{-}try$
else $pc := test$

rem_i
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

777

Tasks and bounds:
$main_i = \{test_i, set_i, crit_i, reset_i, rem_i\}$, bounds $[\ell_1, \ell_2]$
$check_i = \{check_i\}$, bounds $[a_1, a_2]$

定理24.1 $l_2 < a_1$的*FischerME*算法解决互斥问题。

证明 我们将*FischerME*算法与任意用户集一起考虑。良构性条件是显而易见的。对于互斥属性，我们希望证明这一组合系统（算法加用户）的如下不变式。

断言 24.2.1 在任何可达状态中，不存在i和j且$i \neq j$，使得$pc_i = pc_j = crit$。

像通常一样，用归纳法证明这个断言需要辅助不变式。但是，此时需要的辅助不变式涉及时间信息和普通的程序变量。

因此，将该系统转换为一个23.2.2节中所描述的通用定时自动机(GTA)。这个转换将所有的定时约束编码成系统的状态和转移，而不是将它们表达成对定时运行的“外部”约束。具体方法是，在状态中包括组件$first(check_i)$和$last(main_i)$，分别代表下一个$check_i$操作可能发生的最早时间和$main_i$中下一个操作可能发生的最迟时间。我们把断言24.2.1和下面的其他断言看作是通过这种转换所得GTA的状态的属性。

下面是一个关键的断言，用归纳法可证明这一断言。该断言的意思是说，一个成功$check_i$所能发生的最早时间是在任意其他进程j的set_j之后，其中进程j是某个已经通过测试$test_j$的进程。该引理被用于排除例24.2.1中的有害交叉。

断言24.2.2 在任何可达状态中，如果$pc_i = check$、$turn = i$且$pc_j = set$，那么$first(check_i) > last(main_j)$。

对导致问题中状态的定时运行中的步的数量进行简单归纳，即可证明断言24.2.2。这里的步既包括时间通过步，也包括普通的输入、输出和内部控制步。参见例23.3.1中如何进行这类证明的模板。对于断言24.2.2，我们关心的是那些涉及(s, π, s')形式步的论述，其中π是一个set_i或者一个成功的$test_j$，$j \neq i$。（这里的索引i和j与断言中出现的相同。）

778

1）$\pi = set_i$。

在这种情况下，$s'.first(check_i) = s.now + a_1$。同样，如果$s'.pc_j = set$，那么根据引理23.5有$s'.last(main_j) \leq s.now + l_2$。因为$l_2 < a_1$，所以不等式成立。

2）$\pi = test_j$且$s.turn = 0$（即测试成功）。

在这种情况下，$s'.turn = 0$，断言无意义。

断言24.2.2可用于证明下面的断言。这个断言阐述的是，如果一个进程i在临界区内（或刚好在其前或其后），那么$turn = i$并且没有其他进程可以设置$turn$。需要指出的是，与断言

24.2.2不同，断言24.2.3 未提及任何时间信息。但在它的归纳证明中用到时间信息。

断言24.2.3　在任何可达状态中，如果$pc_i \in \{leave\text{-}try, crit, reset\}$，那么$turn = i$，并对所有的$j$有$pc_j \neq set$。

这一证明用的还是归纳法。我们现在关心的是那些涉及步(s, π, s')的论述，其中π或者是一个成功的$check_i$、一个set_j或者一个$reset_j$ $(j \neq i)$、或者是一个成功的$test_j$ $(j \neq i)$。

1）$\pi = check_i$且$s.turn = i$ (即检验是成功的)。

那么$s'.turn = i$。假设存在某个j使得$s'.pc_j = set$。那么也有$s.pc_j = set$。这样根据断言24.2.2知$s.first(check_i) > s.last(main_j)$。但是由引理23.5得$s.now \leqslant s.last(main_j)$，因此$s.first(check_i) > s.now$。这与GTA的定时约束相矛盾。也就是不存在$j$使得$s'.pc_j = set$。

2）$\pi = set_j$，$j \neq i$。

假设$s'.pc_i \in \{leave\text{-}try, crit, reset\}$。那么$s.pc_i \in \{leave\text{-}try, crit, reset\}$。根据归纳假设知不存在$j$使得$s.pc_j = set$。但是这样在$s$中$\pi$就不能被激活，这是一个矛盾。

3）$\pi = reset_j$，$j \neq i$。

假设$s'.pc_i \in \{leave\text{-}try, crit, reset\}$。那么$s.pc_i \in \{leave\text{-}try, crit, reset\}$，并且根据归纳假设知$s.turn = i$。但是，根据$\pi$在$s$中被激活的事实知$s.pc_j = reset$，这样由归纳假设得$s.turn = j$。这是一个矛盾。

4）$\pi = test_j$、$j \neq i$且$s.turn = 0$ (即测试是成功的)。

那么根据归纳假设知$s.pc_i \notin \{leave - try, crit, reset\}$，因此$s'.pc_i \notin \{leave - try, crit, reset\}$，
779 这意味着该条件无意义。

根据断言24.2.3可以直接得到互斥属性，即断言24.2.1。

最后，讨论演进性条件。还有一个不变式对于该条件也是有用的，用一个简单的归纳即可证明该不变式。

断言24.2.4　在任何可达状态中，如果$turn = i$，那么$pc_i \in \{check, leave\text{-}try, crit, reset\}$。

使用断言24.2.4，按照引理10.4的证明方法可以有效地证明演进性，引理10.4是用于*DijkstraME*算法的演进性引理。具体说，就是要考虑一个达到某一点的可容许定时运行α，这个点必须满足条件：至少有一个用户位于T中、没有用户在C中，同时为导出矛盾，假设在此点之后没有用户进入C。那么我们可以证明，最终在α中不会发生新的区域改变，每个进程要么在T中要么在R中，并且某些进程是在T中。这样(用断言24.2.4)我们可以证明，最终*turn* 将获得一个竞争者的索引（如，T中的一个进程）。从而可知，*turn* 必定等于某个竞争者的索引，尽管它可能变成不同竞争者的索引。但是*turn*最后会稳定为某个最终（竞争者的）索引，称之为i。再次使用断言24.2.4，我们可证明进程i进入C。

这样就完成了定理24.1的证明。　□

现在来考虑*FischerME*的时间复杂度。

定理24.2　在*FischerME*算法的任何定时运行中，

1）从任意进程i位于尝试区的任意点开始，直到某个进程位于临界区的间隔时间至多为$2a_2 + 5\,l_2$。

2）从任意进程i位于退出区的任意点开始，直到某个进程位于剩余区的间隔时间至多为$2l_2$。

证明　退出区的时间界限是显而易见的。对于尝试区的时间界限，我们可用操作证明的方式来证明它的时间界限，但为了介绍不同的证明方法，我们给出一个采用23.3.3节中所描述的定时模

拟的证明。需要指出的是，*FischerME*演进性（在定理24.1的证明中）的证明是基于到达一定“里程碑”——例如，某竞争者“夺取”*turn*变量、将*turn*变量“稳定为”某特定竞争者的索引——的运行。我们把这些里程碑及其时间界限合并到一个“抽象互斥算法” *B*中。然后用从*FischerME*到*B*的模拟证明*FischerME*的时间界限。这一证明方法与23.3.3节中的证明方法相同。

抽象算法 B 是下述的 MMT自动机。 780

B automaton:

Signature:

Input:
- try_i, $1 \le i \le n$
- $exit_i$, $1 \le i \le n$

Internal:
- *seize*
- *stabilize*

Output:
- $crit_i$, $1 \le i \le n$
- rem_i, $1 \le i \le n$

States:

status, an element of $\{start, seized, stab\}$, initially *start*
for every i, $1 \le i \le n$:
- $region_i$, an element of $\{R, T, C, E\}$, initially R

Transitions:

try_i
- Effect:
  - $region_i := T$

seize
- Precondition:
  - $\exists i$, $region_i = T$
  - $\forall i$, $region_i \neq C$
  - $status = start$
- Effect:
  - $status := seized$

stabilize
- Precondition:
  - $status = seized$
- Effect:
  - $status := stab$

$crit_i$
- Precondition:
  - $region_i = T$
  - $status = stab$
- Effect:
  - $region_i := C$
  - $status := start$

$exit_i$
- Effect:
  - $region_i := E$

rem_i
- Precondition:
  - $region_i = E$
- Effect:
  - $region_i := R$

Tasks and bounds:

$seize = \{seize\}$, bounds $[0, a_2 + 3\ell_2]$
$stab = \{stabilize\}$, bounds $[0, \ell_2]$
$crit = \{crit_i : 1 \le i \le n\}$, bounds $[0, a_2 + \ell_2]$
for every i, $1 \le i \le n$:
- $rem_i = \{rem_i\}$, $1 \le i \le n$, bounds $[0, 2\ell_2]$

算法B 非常抽象——它只表达了良构性条件和互斥性条件、加上位于尝试区的全局里程碑（包括时间界限）、加上退出区中单个进程的时间界限。由于里程碑的时间界限合计达$[2a_2 +$ 781

5 l_2]，这正是我们所要为尝试区证明的时间界限，因此不难看出B解决了互斥问题并具有所要求的时间界限。现在给出从*FischerME*系统（算法加用户）到B系统（与*FischerME*系统有相同的用户）的一个定时模拟f。因为f是一个定时模拟关系，因此(见定理23.10) *FischerME* 算法也遵从所要求的时间界限。

如果下面的条件成立，则定义$(s, u) \in f$。（我们假设进程索引的全部自由使用都是隐含统一定量的。）

1. $s.now = u.now$.
2. All user states are identical in s and u.
3. $$u.region_i = \begin{cases} R & \text{if } s.pc_i = rem, \\ T & \text{if } s.pc_i \in \{test, set, check, leave\text{-}try\}, \\ C & \text{if } s.pc_i = crit, \\ E & \text{if } s.pc_i \in \{reset, leave\text{-}exit\}. \end{cases}$$
4. $$u.status = \begin{cases} start & \text{if } s.turn = 0 \text{ or } \exists i : s.pc_i \in \{crit, reset\}, \\ seized & \text{if } s.turn \neq 0,\ \nexists i : s.pc_i \in \{crit, reset\}, \text{ and } \exists i : s.pc_i = set, \\ stab & \text{if } s.turn \neq 0 \text{ and } \nexists i : s.pc_i \in \{crit, reset, set\}. \end{cases}$$
5. $u.last(seize) \geq s.last(main_i) + a_2 + 2\ell_2$ if $s.pc_i = reset$.
6. $u.last(seize) \geq \min_i\{g(i)\}$ if $s.turn = 0$, where
$$g(i) = \begin{cases} s.last(check_i) + 2\ell_2 & \text{if } s.pc_i = check, \\ s.last(main_i) + \ell_2 & \text{if } s.pc_i = test, \\ s.last(main_i) & \text{if } s.pc_i = set, \\ \infty & \text{otherwise.} \end{cases}$$
7. $u.last(stab) \geq s.last(main_i)$ if $s.pc_i = set$.
8. $$u.last(crit) \geq \begin{cases} s.last(check_i) + \ell_2 & \text{if } s.pc_i = check \wedge s.turn = i, \\ s.last(main_i) & \text{if } s.pc_i = leave\text{-}try. \end{cases}$$
9. $$u.last(rem_i) \geq \begin{cases} s.last(main_i) + \ell_2 & \text{if } s.pc_i = reset, \\ s.last(main_i) & \text{if } s.pc_i = leave\text{-}exit. \end{cases}$$

其中*now*、用户和区域的对应关系都是显而易见的。对应的*status*给出*FischerME*算法中竞争的*status*的定义：如果 $turn = 0$ 或某个进程正位于（或刚过）临界区，那么竞争的状态为*start*。如果*turn* 等于某个竞争进程的索引（即*turn*非零并且不等于正位于临界区或刚过临界区的进程的索引），并且如果某个进程仍能修改*turn*，那么竞争的状态为*seized*。如果*turn* 等于一个竞争者的索引并且没有进程能够设置 *turn*，那么竞争的状态为 *stab*。

782

last(*seize*)的第一个不等式说明：如果某进程要执行*reset*操作，那么在*reset*操作发生后，直到*turn*变量被夺取前的时间至多为$a_2 + 2l_2$。*seize*的第二个不等式说明：如果 $turn = 0$ (这意味着没有进程处于*crit* 或 *reset*状态)，那么直到*turn*变量被夺取前的时间是由一个可能时间集合中的最小值决定的，集合中的每个时间对应一个可能设置*turn*的候选进程。例如，如果$pc_i = set$——即如果进程i要设置 *turn*——那么相应的时间就是它可以进行下一步操作的最迟时间，但是如果$pc_i = test$——即如果进程i 要测试该变量——那么相应的时间就是在*test*发生后的一个额外时间l_2。余下不等式的解释与此类似。

这样就不难证明f 是一个定时模拟关系；证明沿用例23.3.3和例23.3.4中的论述风格。此证

明用到断言24.2.3 和断言24.2.4；断言24.2.3用于*set*、*crit*和*reset*情况的证明，而断言24.2.4则用于时间通过情况的证明。在这个模拟中，*FischerME* 系统的每个外部步模拟*B*系统的一个相应外部步。（将*turn*从0改变为一个进程索引的）*set*步模拟*seize*，（不留其他进程在*set*步的）*set*步模拟*stabilize*。同时满足这两个条件的*set* 步模拟，既模拟*seize*也模拟*stabilize*（按先*seize*后*stabilize*的顺序模拟）。其他每个步都模拟一个没有任何操作的简单定时运行片段。我们将这些细节留做习题。

从定理23.10可以看出，*FischerME* 系统的可容许定时运行轨迹包含在*B*系统的可容许定时轨迹中。这就证明了我们所需要的时间界限。 □

当对*FischerME* 算法进行修改以适应我们的模型时，尝试区的时间界限为$2kl_2 + 5l_2$，包括在此代码之前我们所讨论的$k-1$个显式*delay*步，其中$kl_1 > l_2$。选择尽可能小的k，即$k = \lfloor L \rfloor + 1$，其中$L = l_2 / l_1$，可得到时间界限$2Ll_2 + O(l_2)$。

放大时间复杂度 时间界限$2Ll_2 + O(l_2)$说明时间不确定性能够如何“放大”一个算法的时间复杂度。当$L = 1$时，即如果$l_1 = l_2$，那么系统中不存在时间不确定性。在这种情况下，时间界限仅为$O(l_2)$——仅取决于每个进程步之间的实际时间上界l_2。但如果 L不等于1，时间界限就会相应增长。实际上，时间界限的实际时间l_2要被乘以时间不确定性L。 783

FischerME 算法中出现谓词Ll_2的原因如下。为了确定一定数量的实际时间t已经过去，进程对自己的步进行统计。它必须统计足够的步，使得即使各步使用的时间为最小消耗量l_1，实际时间t 一定已经过去；因此，步的总量必须至少为t/l_1。但是这些步实际上使用的时间可能是最大的时间消耗量l_2，这时实际过去的总时间至少为$(t/l_1)l_2 = Lt$。

粗略地说，在时间不确定性为L的系统中，为确定实际时间t 已经过去，进程需要的实际时间为Lt。在这一意义下，时间复杂度被一个等于时间不确定性的因子所“放大”。这个时间复杂度放大现象在例23.1.4中的超时例子已经出现过。在那里，为实现正确的超时，需要不等式$kl_1 > l + d$——超时进程实际上检验的是大于$l + d$的实际时间已经过去。然后最晚到时间$kl_2 + l > L(l + d) + l$时可能发生一个超时。

混合时间界限和公平性 我们可以考虑*FischerME* 算法的一个变种，在这一变种中仅有的时间约束是从一个*check*操作被激活直到其发生的时间下界和*set* 操作的时间上界。除了*check*操作外，任何被激活的、局部控制的操作只要求最终会发生。不难看到，这一变种也解决了互斥问题。该变种不能用我们在本书中介绍的MMT自动机来表示；它需要这个模型的另一个版本，这个版本必须支持某些任务的时间界限和其他任务的公平性条件。当然，对于该算法的这一版本，不能证明任何时间界限。

24.3 对时间故障的回复性

FischerME 算法的正确性严重依赖于时间约束。在重要时间限制——*check*步的下界a_1和*set*步的上界l_2——被破坏的定时运行中，即使是互斥这样最基本的正确性条件也会不成立。对这一算法进行改进，使得无论在定时方面发生什么情况都能够满足互斥条件，将是非常有意义的。作为一个通用的设计原则，无论定时如何变化，基于定时的算法都能保证它们最关键的安全属性，这是非常重要的。

用这种方式改进*FischerME*算法的一个想法是：以另一个算法*S*的尝试区、临界区和退出区替代它的临界区。算法*S* 应该总能保证其临界区的互斥性条件，而不受步定时的影响。但是当 784
满足定时约束时，*S* 也不能阻止*FischerME*算法的演进性。我们可以令 *S* 为解决互斥问题（满

足良构性条件、互斥性条件和演进性条件）的任意一个异步算法，但是不幸的是，根据定理10.33知这样的算法需要至少n个共享寄存器。幸运的是，对S我们不需要这样强的演进性条件，相反，我们可以使用如下所示弱一点的演进性条件。

1并发演进性　在每次最多只有一个用户位于R之外的任意可容许定时运行中，

1）（尝试区的1并发演进性）如果U_i在T中，那么在其后的某个点它进入C。

2）（退出区的1并发演进性）如果U_i在E中，那么在其后的某个点它进入R。

当然，这里用户和区域指的是算法S中的用户和区域。

一个满足所需条件的异步算法S的例子如下。需要指出的是，这个算法只使用两个共享寄存器。

S:

Shared variables:
x, a process index, initially arbitrary, writable and readable by all processes
$y \in \{0,1\}$, initially 0, writable and readable by all processes

Process *i*:

```
    ** Remainder region **

    try_i
M:  x := i
    if y ≠ 0 then goto M
    y := 1
    if x ≠ i then goto M
    crit_i

    ** Critical region **

    exit_i
    y := 0
    rem_i
```

785

定理24.3　异步共享存储器算法S保证良构性条件、互斥性条件和1并发演进性条件。

证明　该证明留做一道习题。该证明与本书中其他许多关于互斥算法的证明类似。

FischerME 算法和S的结合可以用下面的代码描述。

***FischerS* algorithm:**

Shared variables:
$turn \in \{0,1,\ldots,n\}$, initially 0, writable and readable by all processes
x, a process index, initially arbitrary, writable and readable by all processes
$y \in \{0,1\}$, initially 0, writable and readable by all processes

Process *i*:

```
    ** Remainder region **
```

```
     try_i
L:   if turn ≠ 0 then goto L
     turn := i
     if turn ≠ i then goto L

M:   x := i
     if y ≠ 0 then goto M
     y := 1
     if x ≠ i then goto M
     crit_i

     ** Critical region **

     exit_i
     y := 0

     turn := 0
     rem_i
```

FischerS 代码既可以被看作是表示一个异步算法，也可以被看作是表示一个部分同步算法。当它表示异步算法时，我们假设所有进程的公平性条件成立。得到定理22.4。 786

定理24.4 被看作是一个异步算法的*FischerS*算法保证良构性条件和互斥性条件。

证明 保证良构性条件是显然的，根据定理24.3所阐述的事实，*S*保证互斥，知互斥性条件成立。 □

我们将*FischerS*演进性属性的确定留做一道习题。

另一方面，当 *FischerS* 代码表示部分同步算法时，像*FischerME*一样，我们假设每个进程i有两个任务，一个任务的时间界限为 $[a_1, a_2]$，另一个任务的时间界限为$[l_1, l_2]$，其中$l_2 < a_1$。第一个任务只包括进程i检验*turn*值的步，第二个任务包括其余的所有步。

定理 24.5 被看作是一个部分同步算法的*FischerS* 算法解决互斥问题，即它保证良构性条件、互斥性条件和演进性条件。

证明 根据定理24.4可以得到良构性条件和互斥性条件。证明退出区的演进性条件也很容易。我们讨论尝试区的演进性条件。在这一证明中，*R*、*T*、*C*和 *E* 表示*FischerS* 算法的区域。我们还定义*FischerME* 的尝试区为*T* 中位于标签*M*之前的部分，并定义*S*的尝试区为*T*的剩余部分。同样，我们定义*S*的退出区为*E*中位于赋值运算$y := 0$之前的部分，*FischerME* 的退出区为*E*的剩余部分。我们同样定义*FischerME*的临界区为*S* 的尝试区、*C*和 *S*的退出区的组合体。

假设在一个可容许定时运行的某个点，至少有一个用户在*T*中，而且没有用户在*C*中。如果在后续的一个点，某进程位于*S* 的尝试区中，那么（使用*FischerME*保证互斥这一事实）由*S*的1并发演进性条件知某个进程最终会进入*C*，定理得证。

另一方面，假设没有进程随后到达*S* 的尝试区。那么由*S* 的1并发演进性条件知*S*的退出区最终将变为空。这意味着*FischerME* 的临界区为空，因此根据*FischerME*算法的演进性条件知某进程最终将进入*FischerME* 的临界区。但是这意味着它进入*S* 的尝试区，这是一个矛盾。 □ 787

24.4 不可能性结果

我们以两个不可能性结果来结束本章。第一个是在部分同步模型中解决互斥问题所需要时

间的下界。第二个是关于要求时间界限最终成立的情况的一个不可能性结果。

24.4.1 时间下界

FischerME 算法解决部分同步共享存储器模型中的互斥问题，并得到尝试区中演进性的最坏时间界限$2Ll_2 + O(l_2)$。有可能对这一情形进行改进以获得时间界限$Ll_2 + O(l_2)$。（我们将这个改进留做一道习题。）但是是否可以做得更好？也就是说，在此模型中，是否存在一个解决互斥问题的更快算法，并且仍然只需要固定数量的变量？我们针对只有一个变量的特殊情况，给出一个简单的结果，这个陈述与定理10.34的关系很密切。

定理 24.6　在部分同步读/写共享存储器模型中，不存在这样的算法：只使用一个读/写共享变量解决了两个进程的互斥问题，并且尝试区中演进性的时间上界为Ll_2。

定理24.6的证明使用一个有趣的证明方法，该方法涉及“放大”和“收缩”定时运行，同时还要遵从时间约束。这一证明主要是基于定理10.34的证明方法。

证明　假设存在这样的一个算法A，A使用单个共享寄存器x，以导出矛盾。我们构造A的一个违反互斥性条件的定时运行。

考虑A的一个可容许定时运行α_1，其中进程1独自运行，并以最慢的速度运行步——也就是说，它的连续步间的时间间隔为l_2。根据时间界限假设，在α_1中，在时间Ll_2内进程1必定到达C。像10.8节中的论证一样，在进入C前，进程1必定写共享变量。令α_2为α_1的前缀，该前缀正好在进程1第一次写x前结束。

类似地，考虑一个慢的可容许定时运行α_3，α_3只涉及进程2并从与α_1相同的起始状态开始，α_3中进程2在Ll_2时间内到达C。令α_4为α_3的一个前缀，当进程2进入C时α_4结束。令α_5为A的另一个有限定时运行，除对其中的每个步都以因子$L = l_2 / l_1$加速（“收缩”）外，α_5与α_4完全相同。因
788 此在α_5中，进程2在l_2时间内进入C。

反例定时运行α以α_2开始，因此将使进程1到达写x的点。在这一点，我们让进程1暂停。现在让进程2像在快速定时运行α_5中一样执行步。（由于在α_2中进程1没有写x，因此进程2无法知道进程1是活动的，从而可以像只有自己一个进程一样执行自己的步。）这样，进程2在开始操作后的时间l_2内到达C。我们让进程1暂停的时间正好为l_2，这个时间足够让进程2达到C。接着我们再继续进程1，让它像在α_1中一样继续执行。它所做的第一件事是写x，从而覆盖进程2在到达C的过程中可能写入的任何东西。这一操作删除进程2的运行的所有证据，因而使得进程1像在α_1中一样运行，最终到达C。但是这就导致C中同时有两个进程，与互斥的要求相矛盾。　□

可以把定理24.6中的下界扩展到有更多共享变量的情况，但是当前关于这些情况的已知结果都不太严密。10.8节的方法可导出一些局部结果。

24.4.2 最终时间界限的不可能性结果*

FischerS 算法在部分同步运行时解决了互斥问题（包括演进性条件），并且在异步运行时至少保证互斥属性。在弱一点的条件下，例如算法异步运行一段时间但最终开始满足时间约束，能保证演进性吗？不难看出，*FischerS* 算法不能做出这样的保证；我们将其留做一道习题。我们将说明实际上没有算法能做到这一点。

定理24.7　对于$n \geq 2$个进程，没有算法能使下列结论全部成立：

1）异步运行时保证良构性条件和互斥性条件

2）当以每个进程的步界限最终都位于范围$[l_1, l_2]$内的方式运行时保证演进性条件[⊖]

3）使用的共享读/写寄存器总数少于n

证明概要 下面的证明与定理10.33的证明方法非常接近。具体来说，其中的主要引理与引理10.37类似——它断言k-可达系统状态的存在性，在这一系统状态中，k 个不同的变量被k个进程"覆盖"。这个引理的陈述中没有出现时间约束。 789

用归纳法证明这个主要引理，采用的构造方法与证明引理10.37的构造方法相同。惟一的不同之处在于，早先证明使用通用演进性条件，而我们现在必须采用略微弱一点的"最终时间有限的"演进性条件。现在，当我们想让进程前进时，所要做的仅仅是按照这样一种方式开始运行它们：从这一点开始，满足它们的时间约束。

该构造中有一个小的技巧：当我们将进程$k+1$的计算接入涉及进程1、...、k的主计算时，必须"收缩"$k+1$的计算，以在其他进程开始它们后面的步前将其接入主计算，并且要让进程$k+1$暂停足够长的时间，以使其他进程完成它们的计算。这些时间调整可能导致时间约束被破坏。但这不是问题——该引理不要求所构造的运行满足任何特定的时间约束。 □

24.5 参考文献注释

FischerME 算法由 *Fischer*[116]设计。近来，这个算法已经被用作测试示例，以演示形式化方法用于推理有关基于定时的系统的能力。*FischerME* 满足互斥属性的证明方法来源于Abadi 和 Lamport [1]及 Luchangco[201]所给的证明。 *FischerME*的时间界限证明参见Luchangco 和 Lynch[201, 204, 205]。改进的时间界限证明同样见[201]。*FischerME* 算法的所有证明都已经由计算机检验过，使用的是Larch 定理证明器[202]。*DijkstraME* 算法的时间界限证明概要见[204]。

FischerS 算法以及24.4节中的不可能性结果参见Lynch和Shavit[209]。Alur和 Taubenfeld [10] 设计了一个部分同步互斥算法，当并发请求的数量有限时该算法有好的时间复杂度；他们的模型和方法与这里使用的模型和方法有所不同。Attiya 和Lynch[25]得到部分同步网络中互斥时间复杂度的一些上界和下界结果。他们的问题与这里所考虑的问题不同，表现在系统不对何时完成临界区给出明确通知。 790

24.6 习题

24.1 证明断言24.2.4。

24.2 为定理24.1补充演进性的操作式证明细节。

24.3 论述*FischerME* 算法允许一个进程被锁定。

24.4 *IncorrectFischerME* 算法是否满足演进性条件？给出一个证明或一个反例。

24.5 补充定理24.2证明中模拟论述部分的细节。

24.6 证明*FischerME*算法的改进后时间界限$2a_2+5l_2-a_1$。

24.7 描述 *FischerME*算法的一个定时运行，使得该运行从某个进程在T中时开始到某个进程在C中为止所需要的时间尽量长。a_2之前的系数2是如何产生的？

24.8 使用一个读/写共享变量，为部分同步共享存储器模型设计一个替代的互斥算法。该算法

⊖ 严格来讲，我们应该定义MMT自动机的最终定时运行，并采用最终定时运行的概念来陈述这个条件。我们略去这一严格的陈述方式。

的时间界限必须是$Ll_2 + O(l_2)$形式，在Ll_2之前没有系数2。

24.9 令P是一个没有输入操作、只有一个输出操作a的MMT自动机。假设P只有一个任务，相应的界限为$[l_1, l_2]$，其中$0 < l_1 \leqslant l_2 < \infty$，并且该任务始终处于被激活状态。假设在每个可容许定时运行中，P只在实际时间大于或等于d时执行一个输出a。证明：存在P的某个可容许定时运行，其输出α的实际时间大于或等于Ld，其中$L = l_2 / l_1$。

24.10 重新考虑10.3节的*DijkstraME*算法。证明：从某个进程在尝试区时开始直到某个进程在临界区为止的时间界限为$(3n + 11)l$，假设l是进程步的时间上界。证明时，将该算法看作是一个MMT自动机，并使用类似于在定理24.2的证明中使用的定时模拟。

24.11 证明定理24.3。（提示：令I_1为进程i的集合，使得$x = i$且i将设置y。令I_2为进程i的集合，使得$x = i$且i将测试x。令I_3是所有在C中、刚好在C前或刚好在C后的进程的集合。证明中或许会用到下列不变式：

791

a）$|I_1 \cup I_2 \cup I_3| \leqslant 1$。

b）如果$|I_2 \cup I_3| > 0$，那么$y = 1$。

c）如果所有进程都属于R，那么$y = 0$。）

24.12 说明算法S不保证演进性（在有并发需求的时候）。要求给出一个其中演进性条件被破坏的运行。

24.13 当*FischerS*算法被看作是异步算法时，它满足1并发演进性条件吗？证明或给出一个反例。

24.14 给出*FischerS*算法作为异步算法的一个显式运行，在该运行中演进性条件被破坏。

24.15 给出一个算法，该算法具有我们为*FischerS*算法所给的所有正确性属性(即它异步运行时保证良构性条件和互斥性条件、部分同步运行时保证演进性条件)，但只使用两个读/写共享寄存器而不是三个读/写共享寄存器。

24.16 证明：不存在这样的一个算法，该算法具有我们为*FischerS*算法所给的所有正确性属性，且只使用一个读/写共享变量而不是三个读/写共享变量。

24.17 *研究问题*：考虑习题10.32中定义的k并发演进性条件。设计一个算法，使其满足良构性条件和互斥性条件，并且异步运行时满足k并发演进性条件、部分同步运行时也满足演进性条件。请使用尽量少的共享寄存器。

24.18 *研究问题*：设计一个基于定时的算法来解决互斥问题（保证良构性条件、互斥性条件和演进性条件）。而且，它应该满足下面的所有时间界限要求：

a）从某用户在T中时开始直到某用户在C中为止的最坏情况的时间是$O(Ll_2)$。

b）从某用户i在T中并且其他所有用户在R中时开始直到某用户i进入到C中或某其他用户进入T中为止的最坏情况的时间是$O(l_2)$。

792

c）从任意用户在E中时开始直到该用户到达R为止的最坏情况的时间是$O(l_2)$。

并且，对该问题设计另一个算法，将你的结论进行泛化，当至多有k个用户同时在R外面时将第二个要求泛化为对尝试区中演进性的好的上界的断言。（在此，k是固定的，$1 \leqslant k \leqslant n$。）

24.19 在部分同步模型中，有两个共享读/写变量的情况下，求尝试区中演进性的时间下界。（提示：参考10.8节的证明。下界的形式为cLl_2，其中c是一个小常量。）

24.20 *研究问题*：对每个k，$1 \leqslant k \leqslant n$，在有$k$个共享变量的部分同步读/写共享存储器模型中，求互斥算法的演进性在最坏情况下的紧时间下界和紧时间上界。

24.21 给出一个特殊的运行，以说明*FischerS*算法不满足定理24.7说明中列出的需求。

24.22 给出定理24.7的一个更详细的证明。

24.23 在一个未知时间界限的模型中考虑互斥问题的可解性。在该模型中，假设过程步时间的上界和下界分别为l_1和l_2，但是这些界限对进程来说是“未知”的。(即，在不同的运行中它们可以不同，虽然每个运行中的界限在整个运行过程中是固定的。)

24.24 研究问题：为更通用的资源分配问题开发一个部分同步算法的定理。

793

第25章 部分同步的一致性

在本书的最后一章，第四次讨论一致性问题，这次是在部分同步网络环境中。我们只考虑停止故障。部分同步环境中的一致性结论与同步和异步环境中的一致性结论有很大差别。首先介绍一个基本的算法和一个基本的下界，它们都是从同步环境中的相应结果推导而来的；基于时间不确定性，在这两个结果的时间复杂度之间有很大差距。然后介绍一个更复杂的算法和一个更复杂的下界结果，它们极大地缩小了这个差距。在本章结尾，给出若干针对更弱的时间模型的结果，并展望一些将来可能进行的工作。

25.1 问题

我们使用与12.1节和21.2节相同的方法定义一致性问题。具体讲，系统A的外部接口包括：输入操作$init(v)_i$和输出操作$decide(v)_i$，其中$1 \leqslant i \leqslant n$且$v \in V$，再加上输入操作$stop_i$。每个用户$U_i$有输出操作$init(v)_i$和输入操作$decide(v)_i$，$v \in V$。现在，$U_i$是一个在任何定时运行中至多执行一次$init_i$操作的MMT自动机。

对于一个由$init_i$操作和$decide_i$操作组成的序列，如果它是一个由$init(v)_i, decide(w)_i$形式所组成的序列的某个前缀，那么该序列对i来说是良构的。对于由A和用户U_i组成的组合系统，我们考虑如下条件：

良构性条件 在组合系统的任意定时运行中，对任意端口i，U_i和A之间的交互对i来说是良构的。

795

一致性条件 在任意定时运行中，所有决定值是惟一的。

有效性条件 在任意定时运行中，如果发生的所有*init*操作都包含相同值v，那么v是惟一可选的决定值。

无故障终止条件 在所有端口都发生*init*事件的任何可容许无故障定时运行中，每个端口都发生一个*decide*事件。

f故障终止条件，$0 \leqslant f \leqslant n$ 在所有端口都发生*init*事件的任何可容许定时运行中，如果在最多f个端口有*stop*事件，那么在每个无故障端口都发生一个*decide*事件。

无等待终止被定义成$f = n$时f终止的特殊情况。

我们假设A是一个23.4.2节所描述的部分同步发送/接收网络系统。每个进程P_i是一个MMT自动机，它的每个任务（任务的总数是有限的）有时间界限l_1和l_2，其中$0 < l_1 \leqslant l_2 < \infty$；令$L = l_2 / l_1$。这些进程都有可能发生停止故障。通道被假设为23.4.2节中所定义的第二种类型，也就是对每个消息的递送时间有上界d的可靠FIFO通道。

如果A对每个用户集都保证良构性条件、一致性条件、有效性条件和无故障终止条件，我们就说它解决了一致性问题。针对f的不同值，我们考虑保证f故障终止的算法。我们考虑的问题是：在所有输入到达后，所有无故障进程做出决定需要多长时间。我们以不确定性参数L在时间复杂度中的作用为中心。

在整个这一章中，我们讨论该问题的一个特例。具体讲，我们假设：$V = \{0, 1\}$、网络图是全连通的。我们假设l_1和l_2比d小得多，实际上，我们假设与d相比nl_2和Ll_2都很小。

我们还需要一个技术假设：一个无故障进程的每个进程任务总是处于使能状态的（虽然任

务惟一使能的操作可能是一个不会引起任何状态改变的虚操作）。该假设使我们在下界证明时可以只考虑步时间的简单模式。

25.2 故障检测器

本章中算法的一个有用构造块是“完美故障检测器”F。在21.4节中定义异步环境的故障检测器。重新回忆一下，故障检测器有输入操作$stop_i$和输出操作$inform\text{-}stopped(j)_i$，其中$j \neq i$。 796
一个$inform\text{-}stopped(j)_i$操作表示端口$i$处的一个关于进程$j$已经停止的断言。一个完美故障检测器保证只报告那些已经实际发生的故障，并保证最终将所有这些故障报告给其他所有无故障进程。与21.4节的惟一不同之处在于我们不再假设故障检测器是I/O自动机，而假设其为通用计时自动机(GTA)。

我们给出一个实现完美故障检测器的部分同步网络系统（在本章假设的模型中）。这个想法与在例23.1.2的超时MMT 自动机中的想法类似。

PSynchFD **算法：**

每个进程P_i不断向所有其他进程P_j发送消息，每个进程使用一个任务。如果一个进程P_i执行m个步而没有收到P_j的任何消息，其中m足够大，它则记录P_j已经停止并输出$inform\text{-}stopped(j)_i$。

步的数量m取严格大于$(d + l_2)/l_1 + 1$的最小整数值。

定理 25.1 *PSynchFD* 是一个完美故障检测器。

证明 显然，所有故障最终都会被所有其他无故障的进程检测到。我们必须证明只有真正的故障被检测到。因此假设P_i输出 $inform\text{-}stopped(j)_i$。那么在此之前，$P_i$执行了总量大于 $(d + l_2)/l_1 + 1$的步而没有从P_j收到任何消息。这说明P_i没有收到P_j任何消息的时间严格大于$d + l_2$。但由于P_i向P_j连续发送消息的时间间隔至多为 l_2，并且每个消息至多经过时间d就会到达，因此两个连续的成功接收事件间的时间间隔至多为$d + l_2$。因此，P_j一定已经停止。 □

我们也将需要*PSynchFD* 的两个时间属性。第一个时间属性的意思是：从故障发生开始，需要经过大于d的时间后才会发生故障通知。第二个时间属性提供直到发生故障通知为止的时间上界。

定理25.2

1）在同时包括$stop_j$事件和$inform - stopped(j)_i$事件的*PSynchFD*任意定时运行中，从$stop_j$事件开始直到$inform - stopped(j)_i$事件为止的时间大于d。

2）在发生一个$stop_j$事件的*PSynchFD*任意定时运行中，在该$stop_j$事件后的$Ld + d + O(Ll_2)$时间内，或者发生一个$inform - stopped(j)_i$事件，或者发生一个$stop_i$事件。 797

证明：

1）像在定理25.1的证明中一样，在$inform - stopped(j)_i$发生时的那一点，P_i已经有$a > d + l_2$的时间没有收到来自P_j的消息。假设在时间t时发生$inform - stopped(j)_i$事件。那么没有消息从P_j到达P_i的时间间隔为$(t - a, t)$。这样在时间间隔 $(t - a, t - a + l_2]$内，一定没有从P_j发送到P_i的消息，否则它将在时间间隔$(t - a, t - a + l_2 + d]$内到达P_i，而这个时间间隔包含在时间间隔$(t - a, t)$内。这就说明在时间 $t - a + l_2 < t - d$时P_j一定已经停止，定理的第一部分得证。

2）考虑*PSynchFD*的一个可容许定时运行，它在时间t时发生一个$stop_j$。那么在时间t后，

就没有从 P_j 发给P_i的消息，因此在时间$t + d$后，P_i就没有收到来自P_j的消息。在收到最后一个消息后，P_i计算m个步的时间至多为ml_2。由于 m恰好大于$(d + l_2)/l_1 + 1$，故$ml_2 = Ld + O(Ll_2)$。因此，如果在此期间P_i没有发生故障，那么从$stop_j$到 $inform - stopped(j)_i$ 的总时间为 $Ld + d + O(Ll_2)$，定理的第二部分得证。 □

定理25.2的第一部分有一个重要推论。当某个进程P_i与另一个进程P_j终止连接时，它知道在P_j发生故障前发送的全部消息都已经到达目的地。

由于我们假设与d相比 Ll_2 比较小，所以可以认为故障通知的时间界限大约为$Ld + d$。

25.3 基本结论

我们首先根据前面章节的结论，讨论对于一致性问题我们都知道什么，并尽量将这些结果推广到部分同步环境。相关的主要结论是为同步模型中f故障一致性匹配$f + 1$轮的上界和下界。这些结论分别在6.2节和6.7节。

25.3.1 上界

6.2节介绍几个用于解决有停止故障的同步网络模型中一致性问题的算法。大部分可容许f停止故障的算法只需要$f + 1$ 轮。也许能够对其中的部分算法进行转换，使之适用于部分同步环
798 境。转换方式如下。

设为A是一个全连通图网络的某个同步网络算法。回想一下同步模型的约定，可知输入出现在初始状态中、而输出被写入只可写一次的局部变量。我们根据A来描述一个适用于部分同步网络模型的算法A'。

A' 算法：

每个进程 P_i是两个MMT自动机的合成：节点i的$PSynchFD$算法部分Q_i、以及主自动机R_i。R_i 有输入操作$inform - stopped_i$。R_i维护一个变量$stopped$，其中记录它已收到输入$inform - stopped(j)_i$ 的进程j的集合，即那些它已经知道的发生故障的进程。R_i 还维护一个记录A中进程i的被模拟状态的变量。

为模拟每个第r轮，进程R_i 首先确定并发出其第r轮所有来自算法A的消息。（每个目的进程使用用一个任务）。这种决定是用A中的函数$msgs_i$ 完成的。接着R_i 等待每个$j \neq i$，直到它或者收到一个从R_j发出的第r轮消息、或者发现$j \in stopped$。然后R_i 使用收到的消息（对于那些R_j没有收到其第r轮消息的进程，使用一个$null$消息），从旧的模拟状态确定A的新的被模拟状态。

现在固定f 并假设A 是解决同步网络模型中的一致性问题的某个f容错、$f + 1$轮算法。如上所述，我们构造A的一个部分同步版本A'。这个算法几乎、但不完全就是我们所需要的算法——惟一的不同之处在于A'使用的输入/输出约定与本章中使用的输入/输出约定不同。因此我们按照下述方法修改A'，得到算法B：首先，在B中，R_i 直到收到输入$init(v)_i$ 时才开始A的模拟。在收到输入$init(v)_i$ 时，它将值v放入模拟输入变量中，并开始第1轮的模拟。（然而，Q_i 在定时运行的起点处开始它的超时活动。）其次，在B中，当R_i 模拟将v 值写入它的输出变量时，在其后立即执行一个$decide(v)_i$ 输出操作。

定理25.3 B解决了部分同步网络模型中的一致性问题，并保证f故障终止。而且，在任意一个所有端口都有输入到达且最多发生有f个故障的可容许定时运行中，从最后的$init$事件开始直到所有无故障进程做出决定为止的时间至多为 $f(Ld + d) + d + O(fLl_2)$。

799 **证明** 不难看出，B 正确地模拟了A，这表明B 解决了一致性问题。对于时间界限，我们给出一个可操作的证明。固定B的一个可容许定时运行α。令S为$PSynchFD$算法的一个上界，其中$S =$

$Ld + d + O(Ll_2)$。根据定理25.2知这样的S存在。我们定义一组时间里程碑$T(0)$、$T(1)$、$T(2)$、....。将证明里程碑$T(r)$是所有还未发生故障的进程完成第r轮模拟的时间上界。

首先，定义$T(0)$为α中最后一个*init*发生的时间。其次，定义：

$$T(1)=\begin{cases} T(0)+l_2+S, & \text{如果某进程在}T(0)+l_2\text{时间内发生故障,} \\ T(0)+l_2+d, & \text{否则。} \end{cases}$$

最后，当$r\geq 2$时，定义：

$$T(r)=\begin{cases} T(r-1)+l_2+S, & \text{如果某进程在时间间隔}(T(r-2)+l_2,T(r-1)+l_2]\text{内发生故障,} \\ T(r-1)+l_2+d, & \text{否则。} \end{cases}$$

因为S是检测故障的时间上界，所以该式是显而易见的。

声明25.4 令$r\geq 0$、j为某个进程索引。如果进程j在$T(r)+l_2$时间前发生故障，那么在$T(r+1)$时间前所有还未发生故障的进程将检测出j已经发生故障。

下面给出主要的声明。

声明25.5 对于所有的$r\geq 0$，$T(r)$是所有还未发生故障的进程完成A的第r轮模拟需要的时间上界。

(声明25.5的)证明 对r进行归纳。

基础：$r=0$。无需说明。

归纳步：$r\geq 1$。如果进程j在$T(r-1)+l_2$时间前发生故障，那么根据声明25.4知在$T(r)$时间前它被所有还未发生故障的进程终止连接。另一方面，如果进程j在$T(r-1)+l_2$时间前未发生故障，那么它在时间$T(r-1)+l_2$前成功发出所有第r轮消息。这些消息在时间$T(r-1)+l_2+d\leq T(r)$前到达各自的目的地。因而，所有进程在$T(r)$时间内完成第r轮。 □

现在我们证明所需要的时间界限，从而完成定理25.3的证明。根据声明25.5，$T(f+1)$是所有无故障进程完成$f+1$轮模拟的时间上界，因此$T(f+1)+O(l_2)$是所有无故障进程执行输出操作*decide*的时间上界。但是里程碑的定义和至多有f个故障的事实表明： 800

$$T(f+1)\leq T(0)+f(l_2+S)+(l_2+d)$$

插入S的界限可得到：

$$T(f+1)\leq T(0)+f(Ld+d)+d+O(fLl_2)$$

由此可得到所需要的界限。 □

25.3.2 下界

在定理6.33中，给出在f故障进程的同步网络模型中解决一致性问题所需要轮数量下界$f+1$。稍微做一点工作，可以将这个时间界限扩展到部分同步模型中，给出时间下界$(f+1)d$。需要指出的是，这个界限没有涉及时间不确定性L。

定理25.6 假设$n\geq f+2$，那么部分同步网络模型不存在保证f故障终止的n进程一致性算法，使得所有无故障进程总是在$(f+1)d$时间前做出决定。

证明概要 假设存在这样一个算法A，以导出矛盾。我们将A转换成一个f轮的同步算法A'，从而与定理6.33矛盾。

当然，当我们仅考虑部分同步模型的一种特殊情况时算法A必须工作正常，在这种特殊情况下，定时运行满足一定的交错和定时约束：

1）所有输入在开始时间0时到达。

2）只要不违背上界条件l_2，所有任务尽可能放慢进度；因而，进程的所有局部控制步都在l_2的整数倍时间时发生[⊖]。而且，每个进程的任务步是按照预定的顺序依次发生的。

3）对于每个$r \in \mathbb{N}$，所有在时间间隔$[rd, (r+1)d]$内发出的消息在$(r+1)d$时被递送。而且，
801 在同一时间发送给单个进程i的消息按照发送者的索引顺序递送。

4）在一个既是l_2的整数倍又是d的整数倍的时间，所有消息的递送都优先于所有局部控制进程步。

满足这些约束的部分同步模型被称作强定时模型。我们将A看作是强定时模型的一个算法。不失一般性，我们可以假设A是“确定性的，”意思是说：每个进程任务在任何状态中至多只有一个局部控制的操作处于使能状态，且对每个状态和每个操作最多可能有一个新状态。同时，由于所有消息都在时间为d的倍数时递送、进程在$(f+1)d$时间前做出决定，因此不失一般性，我们可以假设在消息递送时间fd后的第一步进程做出决定。

在强定时模型中，算法A的行为非常接近于一个f轮同步网络算法的行为。具体讲，对于每个$r \geq 1$，由于在时间$(r-1)d$和rd之间没有消息到达，在时间间隔$[(r-1)d, rd)$内发出的消息都由消息递送时间$(r-1)d$后的进程状态决定。因此我们可能希望将所有这些消息看作一个同步算法的第r轮消息。

但是，二者之间存在显著的技术差别。在同步模型中，如果进程i在第r轮发生故障，那么对于每个$j \neq i$，进程i或者成功地将其所有的第r轮信息发送到进程j，或者一个都没发送成功。如果它成功地发送了第r轮所有信息到进程j，并且没有把任何信息成功发送到进程j'，那么算法A中与这一情形对应的情形是：将其所有在时间间隔$[(r-1)d, rd)$内消息发送到j，但是未将在时间间隔$[(r-1)d, rd)$内的任何消息发送到j'。但是在强定时模型中，当i在该间隔内同时向j和j'发出若干消息时，这种行为是不可能的。

为了将A转换成一个同步算法，有必要对同步模型进行略微扩展。具体方法是，与允许每个进程i在每个r轮只向其他每个进程发送一个消息不同，允许它发送一个有限的消息序列，每个消息发送给任意一个确定的目的地。我们允许i的一个故障可在任何前缀之后中断该序列。不难看出，定理6.33的证明可扩展到这个稍微更通用的模型。只需要在定理6.33证明所构造的链中增加额外的步，以增加和删除该序列中的消息，每次增加或删除一个消息。

现在，可以把给定的一致性算法A转换成这个更强同步模型中的一致性算法A'，其中A'的每个运行与A的一个定时运行对应。进程i在A'的第r轮所发送的消息序列由它在A中$[(r-1)d, rd)$间隔
802 内发出的所有消息组成，顺序与在A中步的顺序相同。A'中由故障引起的行为对应A的可能行为。当$n \geq f+2$时，所得算法A'是该更强同步模型的一个f轮一致性算法。这与定理6.33相矛盾。 □

证明定理25.6的另一种方法是：采用与定理6.33的证明类似的链证明，但直接以强定时模型为条件。同样，在链中要增加额外的步，以增加和删除在“中间”轮所发送的消息。

25.4 有效算法

25.3节中描述的两个结果留下了有趣的时间复杂度差距。下界约为$(f+1)d$而上界约为$fLd+$

⊖ 请大家回顾一下，我们已经假定每个任务总有一个步处于使能的状态。

$(f+1)d$。最显著的差别是在上界中出现时间不确定性L，而在下界中却没有出现。我们希望了解这个问题的内在复杂度是怎样依赖于时间不确定性的。

理解L对时间复杂度影响的实际重要性依赖于L的大小。如果算法A的每个进程P_i都在一个独占处理器上运行，这样P_i步的速度就是由一个高度准确的处理器时钟决定的，那么L通常会非常小、而A的复杂度对L的依赖也可以忽略。另一方面，如果进程的速度由像进程交换这样的其他因素所决定，那么L可能很大而这种依赖就会非常重要。在任何情况下，这个问题都很有理论意义。

我们最初的想法常常是猜想有可能改进定理25.6中的下界结果，以合并乘法因子L。但是实际上是办不到的：实际上存在一个运行时间约为$Ld+(2f+2)d$的聪明算法。粗略地说，这表明只有一个消息的递送被时间不确定性L所“放大”。还有一个可导致下界$Ld+(f-1)d$的更复杂下界证明。在此节中介绍这个算法，在25.5节中介绍这个下界。

25.4.1 算法

我们描述一个部分同步算法*PSynchAgreement*，该算法保证无等待终止，且在最多f故障时该算法有时间界限$Ld+(2f+2)d+O(fl_2+Ll_2)$。*PSynchAgreement*的描述非常简单，但它的行为非常难理解。我们建议读者在读这个算法前，自己尝试设计一个解决方案。 803

在*PSynchAgreement*算法中，我们规定一个进程给“所有进程”发送特定的消息，包括发送给发送者自己。这个模型并不真的允许这样做，但像通常一样，可以使用内部步模拟这个操作。

PSynchAgreement算法：

该算法像25.3.1节中算法B一样使用*PSynchFD*故障检测器。也就是说，*PSynchAgreement*的每个进程P_i是两个MMT自动机的合成：节点i的*PSynchFD*算法部分Q_i、以及主自动机R_i。R_i有输入操作$inform\text{-}stopped_i$。R_i维护一个变量*stopped*，其中记录它已收到输入$inform-stopped(j)_i$的进程j的集合——即那些它已经知道发生故障的进程。

算法依次执行编号为0、1、...的各“轮”。在每一轮，每个R_i试图达到一个决定；但是，在一个偶数轮只允许以0作为决定，在奇数轮，只允许以1作为决定。R_i只在收到输入后才开始第0轮。R_i维护一个变量*decided*，以记录那些它从其收到*decided*消息的进程的轨迹。

第0轮：如果R_i的输入为1，那么R_i按顺序进行下列操作：

向所有进程发送 $goto(1)$

转到第1轮

如果R_i的输入为0，那么R_i按顺序进行下列操作：

向所有进程发送$goto(2)$

输出$decide(0)_i$

向所有进程发送*decided*

第r轮（$r>0$）：R_i等待，直到它或者收到一个$goto(r+1)$消息，或者从不属于$stopped \cup decided$集合的每个进程收到一个$goto(r)$消息为止。在该点，如果R_i收到一个$goto(r+1)$消息，那么它按顺序进行下列操作：

向所有进程发送$goto(r+1)$

转到第$r+1$轮

否则——即，如果R_i没有收到任何$goto(r+1)$消息，但是从不属于$stopped \cup decided$集合的每个进程都收到一个$goto(r)$消息——R_i按顺序进行下列操作：

向所有进程发送$goto(r+2)$

输出 $decide(r \bmod 2)_i$

804 向所有进程发送*decided*

因而，R_i 以检验其初始值开始。如果初始值为1，在告诉其他进程进入第1轮后，R_i 进入第1轮。另一方面，如果初始值为0，那么在告诉其他进程进入到第2轮后，R_i 就以0作为决定。这样就可以防止其他进程在第1轮（以导致矛盾的方式）做出决定。（注意：在开始时，算法支持以0为决定。）

在其后的任意第r轮，如果R_i 被告知进入到第$r+1$轮，那么在通知其他进程进入到第$r+1$轮后，就进入到第$r+1$轮。另一方面，如果R_i 没被告知进入到第$r+1$轮，却发现每个既未发生故障也未做出决定的进程到达第r轮，则它可以以$r \bmod 2$作为决定。

25.4.2 安全属性

首先说明安全属性：良构性、一致性和有效性。这些安全属性都是基于两个引理的。如果一个进程i在$r \geq 0$轮为准备*decide*事件至少发送了一个$goto(r+2)$消息，我们就称进程i在第r轮试图做出决定。

引理25.7 在*PSynchAgreement*的任何定时运行中，对任意的$r \geq 0$，下列结论为真：

1）如果有某个进程发送了一个$goto(r+2)$消息，那么就有进程试图在第r轮做出决定。

2）如果有某个进程到达第$r+2$轮，那么就有进程试图在第r轮做出决定。

证明 在这种方式下，一定会生成第一个 $goto(r+2)$ 消息。一个进程只有在收到$goto(r+2)$消息后，才进入到第$r+2$轮。 □

引理25.8 在*PSynchAgreement*的任何定时运行中，对任意的$r \geq 0$，如果一个进程i在第r轮做出决定，那么下列结论为真：

1）R_i 未发送$goto(r+1)$ 消息。

2）R_i 向每个进程发送一个$goto(r+2)$消息。

3）没有进程试图在第$r+1$轮做出决定。

805 **证明** 根据算法的描述，前两部分是显而易见的。对于第3部分，假设R_j 试图在第$r+1$轮做出决定，以导出矛盾。这意味在第$r+1$轮的某一点，进程R_j 没有收到$goto(r+2)$消息，而是从其他不属于$stopped_j \cup decided_j$的每个进程都收到$goto(r+1)$消息。因为R_i 不发送 $goto(r+1)$消息，所以在指定点一定有$i \in stopped_j \cup decided_j$。

如果在此点$i \in stopped_j$ 成立，那么根据定理25.2知，在R_i 发生故障前，R_j 必定已经收到从R_i 发出的所有消息。但是根据定理的第2部分知其中包括一个$goto(r+2)$ 消息，这是一个矛盾。

另一方面，如果在此点$i \in decided_j$ 成立，那么R_j 一定已经收到一个来自R_i 的*decided*消息。但是 R_i只有在将$goto(r+2)$消息送给R_j 后，才会发送这样的一个消息。那么根据通道的FIFO属性，在这个指定点，R_j 一定已经收到$goto(r+2)$消息，这又是一个矛盾。 □

现在说明安全属性。

定理25.9 *PSynchAgreement*算法保证良构性、一致性和有效性。

证明 良构性是显而易见的。对于有效性，如果所有进程以0开始，那么没有进程能够离开第0轮。因为只有在奇数轮才可以以1作为决定 ，所以没有进程可以以1作为决定。另一方面，如果所有进程以1开始，那么在第0轮就没有进程试图以0作为决定。根据引理25.7知没有进程可以达到第2轮。因此没有进程以0作为决定。

对于一致性，假设R_i在第r轮做出决定且在此之前没有任何进程做出决定。那么根据引理25.8知没有进程试图在第$r+1$轮做出决定。因此根据引理25.7，没有进程可以达到第$r+3$轮。故进程只可能在第r轮和第$r+2$轮做出决定。因为它们有相同的奇偶性系统，所以所有的决定必定相同。 □

25.4.3 活性和复杂度

下面证明无等待终止和时间界限。我们以可容许定时运行的一个活性声明开始。

引理 25.10 在*PSynchAgreement*的任意可容许定时运行中，每个进程不断从一轮前进到另一轮，直到它发生故障或者做出决定为止。

证明 如果不是这样，那么令r为第一个有进程进入停滞状态的轮，注意r必须至少为1。令i为在第r轮被停滞的某进程的索引。对于在此之间发生故障的任意其他进程R_j，R_i必定最终发现 806
这个故障并将j放入$stopped_i$中。同时，对于在此之前做出决定而未发生故障的任意进程R_j，R_i最终必定发现j已经做出决定并将j放入$decided_i$。令I为剩余进程的集合——即，除了那些已经发生故障或做出决定的进程的所有进程。

那么I中的所有进程最终一定到达第r轮，因为r是第一个有进程被停滞的轮。因为$r \geq 1$，这意味着每个进程R_j ($j \in I$)一定会发送一个 $goto(r)$ 消息到R_i，而R_i最终也会收到这个消息。但这样就满足R_i做出决定的条件，因此R_i一定做出决定或者前进到第$r+1$轮。这与开始时R_i在第r轮被停滞的假设矛盾。 □

现在定义一个用于活性和复杂度证明的概念，并证明它的一些属性。在*PSynchAgreement*的一个给定可容许定时运行中，如果某进程在第r轮没有从其他任何进程收到$goto(r+1)$消息，我们称第r轮是*安静的*。将这个新的定义和先前的一些引理相结合，我们得到引理25.11。

引理 25.11 在*PSynchAgreement*的任意可容许定时运行中，对任意$r \geq 0$，下列结论为真：

1）如果没有进程试图在第r轮做出决定，那么第$r+1$轮是安静的。

2）如果某进程在第r轮做出决定，那么第$r+2$轮是安静的。

证明 从引理25.7可以直接得出定理的第1部分。对于第2部分，如果某进程在第r轮做出决定，那么由引理25.8知没有进程在第$r+1$试图做出决定，故由定理的第1部分知第$r+2$轮是安静的。 □

安静轮的概念之所以重要，是因为没有进程可以越过一个安静轮而继续前进。

引理25.12 在*PSynchAgreement*的任意可容许定时运行中，如果第r轮是安静的，则没有进程会进入到第$r+1$轮。

证明 如果进程R_i进入到第$r+1$轮，那么它首先给所有进程发送一个$goto(r+1)$消息。这些消息最终都会被收到，这意味第r轮不是安静的。 □

现在说明一定会产生安静轮。

引理25.13 在*PSynchAgreement*的至多有f个故障的任意可容许定时运行中，存在一个编号不超过$f+2$的安静轮。 807

证明 如果任意进程在f轮内做出决定，依据引理25.11知定理成立。于是假设到第f轮为止没有进程做出决定。因为最多只有f个故障，所以一定存在某个第r轮，$0 \leq r \leq f$，在该轮没有进程发生故障。

我们断言没有进程试图在第r轮做出决定。为导出矛盾，假设某个进程i试图在第r轮做出决

定。那么由于在第r轮进程 i没有发生故障，根据容许性知进程i一定在第r轮做出决定。但这与假设到第f轮时没有进程做出决定矛盾。

因为没有进程试图在第r轮做出决定，根据引理25.11知第$r+1$轮是安静的。 □

现在可以来证明无等待终止。

定理 25.14 *PSynchAgreement*算法保证无等待终止。

证明 考虑一个所有端口发生*init*事件的可容许定时运行。设 i为某个未发生故障端口。证明R_i最终做出决定。

根据引理25.10，R_i 不断从一轮前进到另一轮直到它做出决定为止。但是引理25.13表明存在某个安静的第r轮，而根据引理25.12知R_i 不可能进入到第$r+1$轮。故R_i 必须做出决定。 □

最后，证明时间复杂度界限。在此，固定f为某个故障数，$0 \leqslant f \leqslant n$。

定理25.15 在*PSynchAgreement*任意可容许定时运行中，其中所有端口都有输入到达并且至多只有f个故障，从最后的*init*事件开始到所有无故障进程都做出决定为止的时间至多为 $Ld + (2f + 2)d + O(fl_2 + Ll_2)$。

证明概要 定理25.14的证明及其支持引理说明，该运行一定由一组不安静的轮以及随后的一个安静的轮组成，这组不安静的轮的数量最多为$f+1$，记最后那个安静的轮为第r轮。所有无故障的进程一定在没有经过第r轮之前做出决定。

令S是*PSynchFD*算法的一个上界，其中$S = Ld + d + O(Ll_2)$。定义一组时间里程碑T'、$T(0)$、$T(1)$、…、$T(r)$。T'是最后的*init*发生的时间。对每个k，$0 \leqslant k \leqslant r$，$T(k)$ 是每个进程发生故障、做出决定或者前进到下一轮$k+1$轮的最早时间。因此，到时间$T(r)$时所有无故障进程做出决定。不难看出，第0轮的时间$T(0) - T'$为$O(l_2)$。同样，对于$k \geqslant 1$，第k 轮的时间$T(k) - T(k-1)$至多
808 为$S + O(l_2)$，即比发现故障所需要的时间略微多一点。因此，$T(k) - T(k-1) \leqslant Ld + d + O(Ll_2)$。

一个更有意义的事实是任何不安静轮k的时间$T(k) - T(k-1)$不依赖于时间不确定性L，其中$1 \leqslant k \leqslant r - 1$。为说明这一点，考虑某个特定的进程$R_i$。由于第$k$轮不是安静的，因此$R_i$ 收到一个$goto(k+1)$ 消息，我们给出发生这一事件的时间范围。

该消息必定来源于一个进程R_j所发送的原始$goto(k+1)$消息，其中进程R_j 试图在$k-1$轮做出决定，从原始$goto(k+1)$消息到该消息也可能要经过一系列中间传递。见图25-1所示。

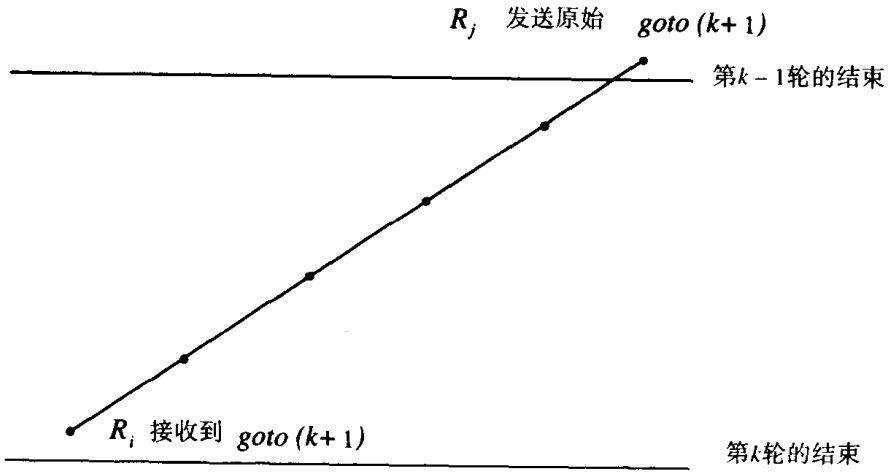


图25-1 从R_j传送到Ri 的$goto(k+1)$消息

声明25.16 令 f_k 表示在发送 $goto(k+1)$ 消息过程中发生故障的进程的总量。那么从 R_j 发送原始消息 $goto(k+1)$ 开始直到 R_i 接收到消息 $goto(k+1)$ 为止，总计需要的时间不超过 $(f_k+1)d + O(f_k l_2)$。

证明 R_j 发送 $goto(k+1)$ 消息，这一操作是它将这样的消息发送给所有进程的一个部分，所发送的目的进程也包括 R_i。如果 R_j 在这一过程中不发生故障，那么 R_j 就成功地将这个消息发送到 R_i，而 R_i 在 R_j 发送后的 d 时间内收到消息。即使 R_j 在这一过程中发生了故障，它所成功发送的全部消息在发送后的 d 时间内也都会到达。 809

同样，将该消息从 R_j 传递到 R_i 过程中涉及的每个进程 R_j 都发送它的消息 $goto(k+1)$，这一操作是它将这样的消息发送给所有进程的一个部分，所发送的目的进程也包括 R_i。而且，如果 R_j 在这一过程中不发生故障，那么 R_j 就成功地将这个消息发送到 R_i，并且 R_i 在 R_j 发送后 d 时间内收到消息。即使 R_j 在这一过程中发生了故障，它所成功发送的全部消息在发送后的 d 时间内也都会到达。

因此，从原始消息 $goto(k+1)$ 被 R_j 发送开始直到 i 收到某个消息 $goto(k+1)$ 为止的总计时间至多为 $(f_k+1)d + O(f_k l_2)$。（l_2 代表从一个传递进程收到 $goto(k+1)$ 消息开始直到它发送自己的 $goto(k+1)$ 消息为止的时间。） □

由于原始消息 $goto(k+1)$ 是由 R_j 在第 $k-1$ 轮时发送的，因此它是在 $T(k-1)$ 时间前发送的。由于所有进程在 $(f_k+1)d + O(f_k l_2)$ 时间内收到 $goto(k+1)$ 消息，因此在 $T(k-1) + (f_k+1)d + O(f_k l_2) + O(l_2) = T(k-1) + (f_k+1)(d + O(l_2))$ 时间前，每个进程前进到第 $k+1$ 轮、发生故障或者做出决定。这说明 $T(k) - T(k-1) \leq (f_k+1)(d + O(l_2))$。正如我们前面所说，这与时间不确定性 L 无关。

由于 $T(0) - T'$ 为 $O(l_2)$、对于所有 k 有 $1 \leq k \leq r-1$、$T(k) - T(k-1) \leq (f_k+1)(d + O(l_2))$ 以及 $T(r) - T(r-1) \leq Ld + d + O(Ll_2)$，因此：

$$T(r) - T' \leq \sum_{k=1}^{r-1} (fk+1)(d + O(l_2)) + Ld + d + O(Ll_2)$$

由于 $\sum_{k=1}^{r-1} f_k \leq f$ 和 $r \leq f+2$，因此得到

$$T(r) - T' \leq Ld + (2f+2)d + O(fl_2 + Ll_2)$$

这正是所需要的时间复杂度界限。

25.5 涉及时间不确定性的下界*

在25.4节中，给出了一个部分同步一致性算法 *PSynchAgreement*，其时间复杂度约为 $Ld + (2f+2)d$。沿着缩小25.3节中已经证明的简单大致上界 $fLd + (f+1)d$ 和简单大致下界 $(f+1)d$ 的差距的方向，*PSynchAgreement* 算法已经前进了很多。具体来说，*PSynchAgreement* 算法证实了不可能证明一个包括 fLd 的下界。在本节，证明一个依赖于 L 的下界 $Ld + (f-1)d$。虽然我们至少已经清楚时间复杂度对时间不确定性 L 的依赖形式，但在上下界之间仍然存在差距。

定理 25.17 假设 $n \geq f+1$。那么在部分同步模型中，不存在保证 f 故障终止的 n 进程一致性算法，其中所有无故障进程总是在时间 $Ld + (f-1)d$ 前做出决定。 810

定理25.17的证明是非常有趣的，因为它使用前面章节中多种技术的组合，其中包括第6章的链证明、第12章中基于各种决定值的可达性的证明和第24章中关于放大和收缩定时运行的证明。

在本节的剩余部分，为导出矛盾，我们假设 A 是部分同步网络模型的一个 n 进程一致性算法，可保证 f 故障终止，其中所有无故障进程总是在时间 $Ld + (f-1)d$ 前做出决定。不失一般性，我们像在定理25.6的证明中一样假设 A 是“确定性的”。我们证明一组引理，根据这些引理可导

出A不存在的结论。

首先，在引理25.18中，我们说明如果算法A是正确的，就不会出现定时运行的一个特定“坏组合”。这个坏组合包括一个“0价”定时运行α_0和一个“1价”定时运行α_1，二者可到达的时间都至少为$(f-1)d$，也几乎没有发生故障，最多与一个无故障进程是可以区别的。引理25.18是用放大和收缩方法证明的。第二，在引理25.19中，我们说明确实存在的一个相关组合——除不是要求0价和1价而是要求0是从α_0可达的且1是从α_1可达的外，该组合与上面坏组合有同样的条件。引理25.19是用链方法证明的。第三，在引理25.20中，我们生成一个“双价”定时运行α，可到达的时间都至少为$(f-1)d$并且几乎没有发生故障。引理25.20可直接从引理25.18和引理25.19得到。第四，在引理25.21中，我们对引理25.20进行强化，以包括一个“极大性”属性，该属性可导致α的两个直接扩展，一个0价扩展α_0和一个1价扩展α_1。但是α_0和α_1组成一个“坏组合”，导致一个矛盾。

现在给出这些细节。首先从A的所有定时运行中区分出一个称为“同步”定时运行的子集。一个同步无限定时运行是一个具备无限时间序列$t_0=0$，t_1，t_2，...的无限定时运行，其中对于$k\geq 0$有$l_1\leq t_{k+1}-t_k\leq l_2$，并满足下列条件：

1）所有输入刚好在开始时间t_0时到达。

2）无故障进程的所有任务都在时间t_1，t_2，...时执行步；我们称这些时间为*活动时间*[⊖]。而且，每个进程的任务步按照预定的顺序依次发生。

811　3）在同一时间递送给i的所有消息按照发送者索引的顺序被依次递送。

4）在每个活动时间，任何消息递送的发生都优先于所有局部控制进程步的发生。

这些条件与定理25.6的证明概要中使用的条件有些类似。一个同步无限定时运行可以划分为无限数量的“块”B_0，B_1，B_2，...，其中每个B_k包括在时间t_k时的全部输入和消息递送步，但是不包括在时间t_k时的局部控制进程步。因此，块B_0只包括输入事件，而每个$k\geq 1$的块B_k以时间t_{k-1}时的局部控制步开始、以时间t_k时的消息递送步结束。一个同步有限定时运行是一个同步无限定时运行的前缀，由有限数量的完整块组成。

如果α和α'是同步定时运行，其中α是α'的一个有限前缀，当α正好由α'的完整块B_0，B_1，...，B_k组成时，我们称α是α'的一个k块前缀，其中$k\geq 0$。（特别是，一个0块前缀包含一个块B_0。）如果对于某个$k\geq 0$，α是α'的一个k块前缀，我们就说α是α'的一个*块前缀*，并说α'是α的一个*块扩展*。

我们将对一些特定类型的块扩展特别感兴趣。具体来说，如果α是一个同步有限运行，α'是一个同步（有限或无限）运行，并且α是α'的一个k块前缀，$k\geq 0$，那么我们说α'是：

1）α的一个*快速扩展*，条件是α'中位于α之后的所有步只需要最小时间l_1，即对所有$i\geq k$有$t_{i+1}-t_i=l_1$。

2）α的一个*慢速扩展*，条件是α'中位于α之后的所有步都需要最大时间l_2，即对所有$i\geq k$有$t_{i+1}-t_i=l_2$。

3）α的一个*无故障扩展*，条件是α'中位于α之后没有*stop*事件。

4）α的一个*fff扩展*，条件是α'是α的一个快速无故障扩展。

我们强调所有这些类型的扩展都只是完整块的块扩展。需要指出的是，名称“快”和“慢”所指的只是进程步时间，而与消息递送时间无关，消息递送时间仍旧为$[0, d]$范围内的任意数。

现在定义一些概念，这些概念类似于在第12章中异步模型的一致性的不可能性结论中使用的概念。对于某个值$v\in\{0, 1\}$，如果存在同步有限定时运行α的某个*fff*扩展α'，在α'有某个进

⊖　请再次回顾一下，我们已经假定每个任务总有一个步处于使能的状态。

程以v作为决定，我们则称值v是从α中 *fff*可到达的。（这个决定可以发生在α中，也可以发生在α'位于α之后的部分。）对于一个同步有限定时运行α，当只有值0是从α中 *fff*可到达的时我们定义α为 0价的，当只有值1是从α中 *fff*可到达的时我们定义α为1价的，当二者都是从α中*fff*可到达的时我们定义α为双价的。当定时运行α是0价的或者1价的时则称它是单价的。 812

我们还需要一个概念——两个有限定时运行对一个特定进程i的“不可分辨性”概念。类似的概念已经在本书的有关同步和异步章节中被使用过。这里所需要的概念比前面的概念要略微复杂一些，因为它考虑了在运行的末尾还处于到i的传输途中的那些消息。具体来说，如果α和α'是有相同活动时间的两个同步有限定时运行，那么当下列条件成立时，我们说对i而言α和α'是不可区分的。

1）α和α'在i上的投影$\alpha|P_i$和$\alpha'|P_i$是时间通过等价的[⊖]。

2）在α和α'中，同样的消息由同样的进程、以同样的顺序、在同样的时间送给P_i。

下面的引理描述了定时运行的一个特定坏组合，当算法A正确时该坏组合不可能发生。

引理25.18 不存在两个k块同步定时运行α_0和α_1，使得下面的所有条件都成立：

1）α_0和α_1有同样的活动时间t_1，…，t_k，其中$t_k \geq (f-1)d$。

2）α_0是 0价。

3）α_1是 1价。

4）$|F| \leq f-1$，其中F是在α_0或α_1中发生故障的进程的集合。

5）α_0和α_1对于至多一个不在F中的进程是可区分的。

图25-2描述这个坏组合。

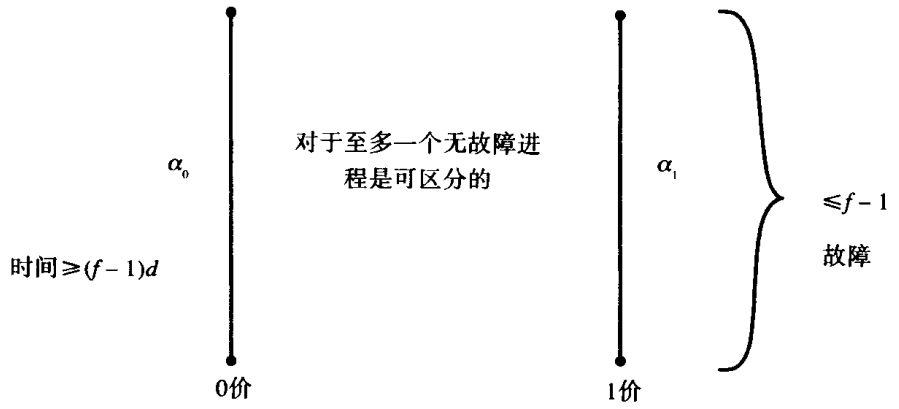


图25-2 引理25.18定时运行的坏组合

证明 为导出矛盾，假设存在这样α_0和α_1。我们将分别构造α_0和α_1的扩展β_0和β_1，这两个扩展都导致同样的决定，比如说0。然后我们将对β_1进行加速并删除其中的某些故障，以获得一个*fff*扩展β'_1，β'_1的决定也是0。这样就与α_1的1价矛盾。

更详细地讲，令G为F再加上两个定时运行α_0和α_1对其是可区别的那个进程，因而$|G| \leq f$。我们分别构造α_0和α_1的慢速扩展β_0和β_1如下。

首先，在时间t_k，我们为G中所有还无故障的进程提供*stop*事件。 然后没有额外故障的慢速扩展以同样的方式扩展α_0和α_1。用同样的方式对它们进行扩展是有可能的，因为α_0和α_1对于 813

⊖ 投影操作|是在23.2.3节中定义的，时间通过等价的概念是在23.2.1节中定义的。

除了G中进程外的所有进程都是不可区分的。根据所假定的A的上界，β_0和β_1中的所有无故障进程必定在时间$Ld + (f - 1)d \leqslant t_k + Ld$之前做出决定。因此，在决定发生之前，两个定时运行的新增部分经过的时间小于Ld。而且，因为α_0和α_1以同样的方式扩展，所以在β_0和β_1中达到的决定也相同。假设不损失一般性，这个共同的决定是0。见图25-3。

现在考虑将α_1扩展为另一个同步定时运行β'_1；与β_1不同，β'_1是α_1的一个*fff*扩展。除了α_1后面的部分被因子L“加速”而变快外，定时运行β'_1与β_1相同。而且，没有进程在β'_1中位于α_1之后的部分发生故障；但是在β'_1中位于α_1之后的部分，G中进程所发送的任何消息都需要经过最
814 大时间量d后才能到达。因此在时间$t_k + d$之前，β'_1的行为完全像β_1的一个加速版本。（需要说明的是，一旦G中进程所发送的消息到达，在β_1和β'_1中，事情看起来差别很大，但这没有关系。）由于在β_1中，所有不在G中的进程在时间$t_k + Ld$前决定0，所以在β'_1中，它们将在时间$t_k + d$前决定0。见图25-4。

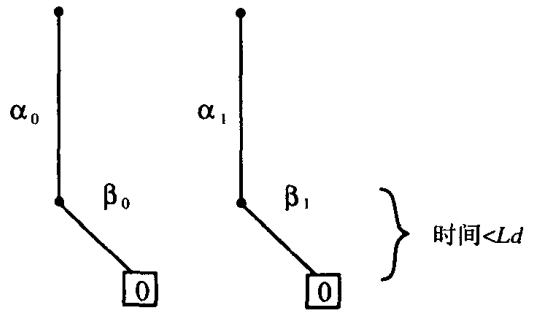


图25-3　引理25.18证明中β_0和β_1的扩展

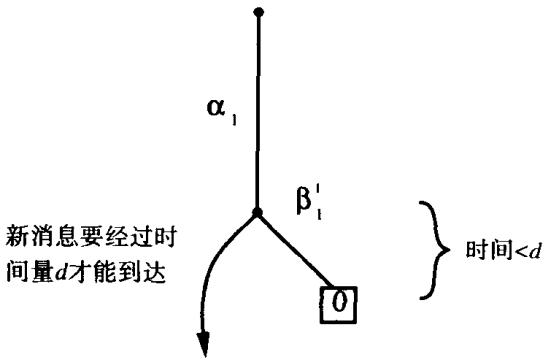


图25-4　引理25.18证明中的扩展β_1'

但由于β'_1是α_1的一个*fff*扩展，因此这与α_1的1价矛盾。 □

我们将说明引理25.18所描述的定时运行的坏组合一定会在实际中发生，从而得到一个矛盾。我们首先得到一个相关组合。

引理 25.19　对于某个k，存在两个k块同步定时运行α_0和α_1，使得下面的所有条件都成立：

1）α_0和α_1有同样的活动时间t_1，…，t_k，其中$t_k \geqslant (f - 1)d$。

2）0是从α_0 *fff*可达的。

3）1是从α_1 *fff*可达的。

4）$|F| \leqslant f - 1$，其中F是在α_0或α_1中发生故障的进程的集合。

5）α_0和α_1对于至多一个不在F中的进程是可区分的。

需要说明的是，这些条件和坏组合中条件的惟一不同是，条件2和条件3只要求0和1是*fff*可达的，而不是要求α_0是0价的以及α_1是1价的。见图25-5。

815 **证明概要**　使用类似定理6.33证明中的链证明方法，可以证明该引理。该证明留做一道习题。 □

综合引理25.18和引理25.19可直接得到引理25.20。

引理25.20　存在一个同步有限定时运行α，使得下面所有条件都成立：

1）α的最后活动时间t_k至少为$(f - 1)d$。

2）α是双价的。

3）在α中，至多有$f - 1$个进程发生故障。

见图25-6。

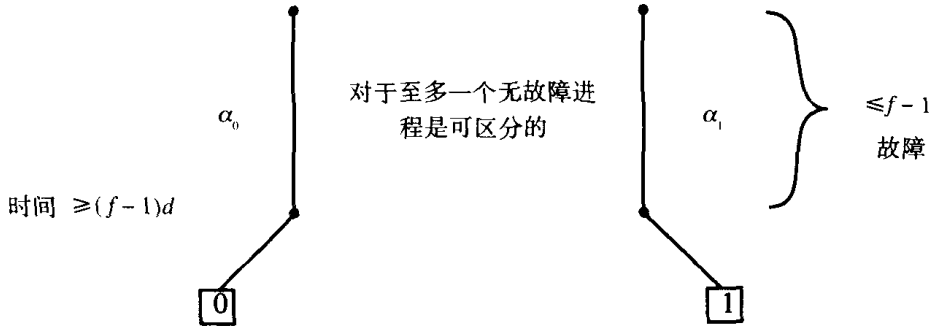


图25-5 引理25.19证明中的定时运行α_0和α_1

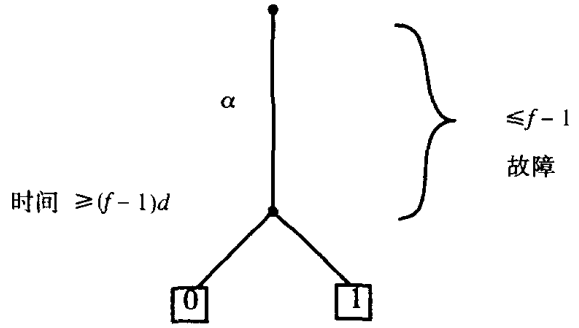


图25-6 引理25.20的定时运行α

证明 令α_0和α_1是两个同步定时运行，它们的存在性由引理25.19所断言。根据引理25.18，不可能出现α_0是0价同时α_1是1价的情况。因此，α_0和α_1中至少有一个是双价的，故它满足所有必要的条件。 □

下面对引理25.20进行强化，以包括一个“极大性”属性。 816

引理 25.21 存在一个同步有限定时运行α，使得下面的所有条件都成立：

1）α的最后活动时间t_k至少为$(f-1)d$。

2）α是双价的。

3）在α中，至多有$f-1$个进程发生故障。

4）存在α的两个*fff*扩展β_0和β_1，每个由一个块扩展，使得：

a）β_0是0价的。

b）β_1是1价的。

c）β_0和β_1对至多一个进程是可区分的。

见图25-7。（请注意，该引理所断言存在的构造与定理12.6证明中的*decider*概念之间的相似性。）

证明概要 令α为由引理25.20断言其存在性的同步有限定时运行。然后执行下面的“程序”，对α进行扩展：

```
当存在α的一个适当的双价fff扩展时，
则α：=任意这样的扩展
```

我们知道该程序最终会终止，因为在时间$Ld+(f-1)d$之前α的全部无故障扩展都需要决定。考

虑这一程序所产生的最终α。

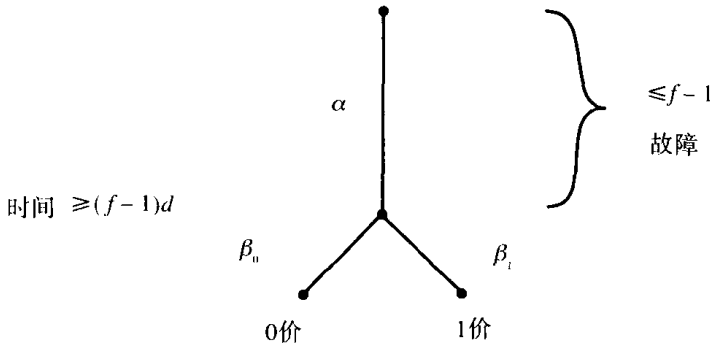


图25-7　引理25.21的α、β_0和β_1

817　我们断言该α有我们需要的所有属性。它满足所需要的时间界限、双价和故障条件。而且，因为它是双价的，但又不能被扩展成一个更长的定时运行，所以一定存在α的两个*fff*扩展γ_0和γ_1，每个由一个块扩展，使得：

1）γ_0是0价的。

2）γ_1是1价的。

但这并不就是我们想要的结果，因为γ_0和γ_1可能对一个以上进程是可区分的。

因此，我们在γ_0和γ_1之间再进行一次链构造，生成所需要的β_0和β_1。从γ_0开始，我们在链中的每一步只是对到一个进程P_i的所有消息递送进行简单地修改，使得这些到P_i的消息递送与在γ_1中的相同。链中的每两个连续的定时运行都只对一个进程是可区分的。由于所有这些定时运行都必定是单价的，因此在链中存在两个连续的运行β_0和β_1，使得β_0是0价的而β_1是1价的。从而给出所需要的全部属性。□

现在可以得到矛盾。

引理25.22　*A*不存在。

证明　由引理25.21断言其存在性的两个同步定时运行β_0和β_1满足引理25.18中所列的一个坏组合的所有需求。这是一个矛盾。□

这就证明了定理25.17。

25.6　其他结果*

在本节，考虑如果使用几种方式弱化定时模型，那么对一致性问题的结论会产生什么样的影响。我们这里的讨论是非形式化的。

25.6.1　同步进程、异步通道*

假设我们用第14章中定义的可靠FIFO通道对模型进行弱化，这种通道对消息递送时间没有上界，只保证消息的最终递送。但进程仍旧遵守$[l_1, l_2]$的时间范围约束。在这种情况下，不难看出，即使是一个停止故障情况下也不能解决一致性问题。即使$l_1 = l_2$时，即进程步时间完
818　全可以预计时，也不能解决一致性问题。

定理25.23　在同步进程和异步通道的模型中，不存在解决一致性问题并保证1故障终止的算法。

证明概要　为导出矛盾，假设*A*是这样的一个算法。那么使用第18章所定义的逻辑时间的实现，

可以在异步模型中“模拟”A。采用这种方法，可以在异步网络模型中生成一个保证1故障终止的一致性算法，与定理21.2矛盾。我们将证明细节留做一道习题。 □

25.6.2 异步进程、同步通道*

现在假设我们弱化部分同步模型，这次保留所有消息的递送时间上界d，但对进程只有公平性要求，没有给出时间界限。同样，不难看出，即使对一个停止故障，一致性问题也不能解决。

定理25.24 在异步进程和d界限通道的模型中，不存在解决一致性问题并保证1故障终止的算法。

证明 为导出矛盾，假设A是这样的一个算法。在异步模型中运行同样的算法A 。那么在这个异步模型中，A的任何公平运行α都能被“定时”，方法是让所有消息遵守上界d。这就意味着该运行满足1故障终止的一致性问题所要求的所有条件。因为所有这些条件都不依赖于时间，所以同样的条件对给定的公平运行α也都成立。因为这种做法可用于A的任何公平运行α，所以A解决了异步模型中1故障终止的一致性问题。同样与定理21.2相矛盾。 □

25.6.3 最终时间界限*

作为本书的最后结论，我们像在24.4.2节中一样考虑最终时间界限的情况。具体地说，考虑这样的一个模型，算法在其中异步运行一段时间但最终满足时间约束。一致性问题在这个模型中是可解的。然而，与在时间界限始终成立的部分同步模型中不同，解决方案要求$n > 2f$。使用类似于定理17.6证明中的论述方法，不难说明：当$n \leqslant 2f$时该问题在这个模型中是无解的。我们将其留做一道习题。 819

定理 25.25 只要$n > 2f$，在进程任务时间界限为$[l_1, l_2]$和所有消息的时间界限为d都最终成立的模型中，f故障终止的一致性问题是可解的。

在这个模型中为这个问题设计一个解决方案并不容易。用像在*PSynchAgreement*一致性中使用的、基于暂停与故障进程连接的这类策略解决不了问题，因为在时间界限成立前，进程能得到其他进程已经发生故障的错误结论。我们基于一个不同的策略来勾画一个算法。

该算法的核心是针对第2章同步模型的一个变种的A算法，在该算法中，除了有最多f个进程发生故障外，还可能有一些消息丢失。我们假设任何消息都可能丢失，但只能在总量有限的轮丢失消息。在某点之后，所有消息的递送都是有保证的。但是进程不知道该点是在什么时候。

A算法 的工作方法如下。再次假设进程既向其他进程发送消息，也向自己发送消息。

***A*算法：**

轮被组织成“运行节” 1、2、...，其中每个运行节s由4个连续的轮$4s-3$、$4s-2$、$4s-1$和$4s$组成。运行节s由进程$owner(s)$所“拥有”；这是一个其索引等于s对n取模的进程。

在不同的时间，一个进程可以锁定一个值$v \in \{0, 1\}$以及一个相关的运行节号s。如果进程i锁定(v, s)，则表示进程i认为$owner(s)$可能在运行节s决定v。当进程i继续认为$owner(s)$可能在运行节s决定v时，它就继续占有对v的某个锁。当进程i没有对v的锁时，则称值v对进程i是可接受的。开始时，没有值是被锁定的。

在任意运行节s中，拥有者i的处理过程如下。

第$4s-3$轮：所有进程向进程i发送它们的所有可接受值。进程i于是试图从中选择一个用来建议的值。为使进程i建议v，它必须听到至少$n-f$个进程（可能包括它自己）发现值v在运行节s是可接受的。适合i建议的值可能不止一个，这种情况下i选择它自己的初始值。

第$4s-2$轮：如果进程i已经决定建议一个值v，那么它向所有进程发送一个$(“lock”, v)$消息。任何收到这样一个消息的进程都锁住(v, s)、并释放先前对同一个值v的任何锁。 820

第$4s-1$轮：在第$4s-2$轮收到消息(“*lock*”, v)的任何进程发送一个*ack*消息给进程i。如果进程i从至少$f+1$个进程收到*ack*消息，那么进程i决定它的被建议值v。

第$4s$轮：每个进程向每个其他进程发送包括其当前锁的消息。任何锁住(v, s')并收到消息$(\bar{v}, s'')$的进程都释放先前的锁，其中$s''>s'$。

引理25.26　如果$n>2f$，算法A解决一致性问题并保证f故障终止。

证明　首先注意：

声明25.27　对每个运行节s，至多有一个值v在运行节s被建议，因此至多有一个值v使得某个进程持有对(v, s)的一个锁定。

那么根据对轮的数量的一个简单归纳（使用进程偏爱其初始值的事实）可知：

声明25.28　如果所有进程以初始值v开始，那么$\bar{v}$从不会被建议或锁定。

由于一个进程只决定一个它所建议的值，因此得到有效性。接着我们说明：

声明25.29　如果进程i在运行节s决定值v，那么在每个运行节$\geq s$的末尾，至少有$f+1$个进程有对v及相应运行节号$\geq s$的锁。

证明　该算法保证在第$4s-2$轮至少有$f+1$个进程锁定(v, s)。我们断言：如果没有立即获得对v的另一个锁，则这些进程都不会释放对v的一个锁。

为导出矛盾，假设其中某个进程i在没有立即获得对v的另一个锁的情况下释放对v的一个锁。那么进程i必定释放这个锁定，因为对某个$s'>s$它知道一个对$(\bar{v}, s')$的锁，这意味着*owner*(s')在运行节s'建议$\bar{v}$。取s'为$\bar{v}$被建议且$s'>s$成立的第一个运行节。

但是，正好位于运行节s'之前，对v必定有至少$f+1$个锁，这些锁将阻止*owner*(s')在第$4s'-3$轮从$n-f$个所必要的进程处获得对$\bar{v}$的认可。这是一个矛盾。 □

821

现在继续引理25.26的证明。我们给出一致性。假设进程i在运行节s决定v。那么没有进程可以在同一运行节决定$\bar{v}$。而且，根据声明25.29，从运行节s开始，对v总是有至少$f+1$个锁。这样就阻止了任何进程从它建议$\bar{v}$所必需的$n-f$个进程处获得对$\bar{v}$的认可。因此$\bar{v}$从不会被建议，也没有进程会决定$\bar{v}$。

为说明终止，考虑到达我们所假定的所有消息都被可靠传递的那一点后，会发生什么情况。在任何后续的运行节s，不难看出：在系统的所有无故障进程中至多只有一个锁定值。这是因为声明25.27和第$4s$轮中的锁定-释放规则。一旦如此，任何运行节的拥有者都将会成功地获得允许其做出决定所需要的全部认可和确认（如果它不发生故障）。 □

（定理25.25的）证明概要　我们只针对有最终时间界限的模型，给出算法B的构造的一般思想。B的每个进程P_i维护一个非负整型局部变量*clock*，其初始值为0。每个*clock*变量是单调非递减的。令$C=\max\{clock_i, 1\leq i\leq n\}$。那么可以把$C$看作是由系统维护的一类“全局时钟”。由一个包括重复发送和更新*clock*值的协议（在此我们忽略该协议），进程可以保证，从时间界限开始被满足的那一点p之后不久开始：

1）按照实际时间，C的增长速度是由已知的常数界限来限定上界和下界的。

2）每个*clock*在C的一个已知（额外）常量范围内。

因此，这些进程最终将到达相当同步的*clock*。

B的每个进程P_i除了维护其*clock*外，还模拟它在算法A中的副本，用局部*clock*决定所要模

拟的轮。模拟每个轮r所用到的时钟值的数量相当大（但是是可预见的）——足以保证在运行的点p后，在一个进程P_i的模拟轮r开始处由P_i所发送的任何消息都在进程P_j的模拟轮r结束前被真正传递到每个P_j。

需要说明的是，在点p之前，在某P_i的*clock*前进到太远之前，它可能还未完成某轮r的模拟。在这种情况下，如果P_i简单地忽略发送额外消息的操作，那么也不会产生不良后果——毕竟在A中这些额外消息可能被丢失。但是，P_i必须模拟轮r的转换状态。可以在轮r的模拟被中断后的第一步进行轮r的状态转换模拟。

822

采用这种方式，B模拟算法A并完成同样的正确性条件。 □

25.7 小结

在本章和前面一章，我们已经介绍部分同步模型中两个分布式计算基础问题的几个基本结论——互斥和一致性。这几个结果已经表明，部分同步分布式计算的理论与同步或异步分布式计算的理论有很大不同。

然而，在这个领域还有许多工作需要做。在分布式计算中有许多其他可在部分同步环境中考虑的重要问题。这些问题包括本书所描述的许多问题，例如，网络搜索问题、生成树构造问题、资源分配问题、快照问题和稳定属性检测问题。它们还包括在实际通信系统、分布式操作系统和实时进程控制系统中出现的许多其他问题。

用通用的描述结果来准确描述在部分同步系统中能够计算什么以及所需要的时间复杂度是非常有用的。建立部分同步模型的功能与同步和异步模型的功能之间关系的转换结果也很有用。

25.8 参考文献注释

本章大部分构造方法和结论都是由Attiya、Dwork、Lynch和Stockmeyer[22]证明的，这些构造方法和结果包括*PSynchFD*故障检测器、定理25.3和定理25.6中的简单上界和下界结果、以及在定理25.15和定理25.17中更复杂的上界和下界结果。Ponzio[247, 245]将25.4.1节中的算法扩展到更强的“发送 - 忽略”故障模型情况并针对Byzantine故障给出一个效率略低一些的算法。Berman和Bharali[48]改进了Ponzio的发送 - 忽略算法的复杂度。在一个两节点的系统中，Ponzio还获得了好的故障检测时间复杂度上界和下界[246]。同步进程和异步通道的不可能性结果，定理25.23，是由Dolev、Dwork,和Stockmeyer[95]首先证明的。这里所概述的证明基于*WelchTime*，出自Welch [287]。

823

定理25.25针对有最终时间界限的模型，是由Dwork、Lynch和Stockmeyer[104]证明的。该文章还包括未知时间界限模型的一个类似结果以及其他故障模型的结果。Lamport的Paxos算法（见[183]）与[104]中的算法非常类似。

部分同步模型的其他结果包括：由Attiya和Mavronicolas[17]提出的、解决16.6节中会话问题所需要时间的上界和下界；由Wang和Zuck[284]给出的、可靠高层消息传输所需要低层消息字母表的大小的界限；以及由Kleinberg、Attiya和Lynch[167]给出的、在连接管理协议中消息递送时间和系统静止时间的折衷上界和下界。

25.9 习题

25.1 给出*PSynchFD*算法中进程P_i的前提 - 结果代码。

25.2 假设不使用保证所有消息在时间d内递送的通道，而是使用只保证在时间d内递送最早消息

的通道。

a）修改*PSynchFD*算法，使其可用于这一模型，要尽可能减小修改后算法的时间复杂度。

b）证明这一环境中故障检测器的时间复杂度下界。

25.3 研究问题：为在部分同步模型中模拟有停止故障的同步网络算法，设计一个你认为最高效的算法。你能够达到模拟r轮所需要时间的上界为$Ld + rd$(加低阶谓词)吗?

25.4 在所有输入都假定在时间0时到达的特殊情况下，下面的策略可以解决部分同步网络中的一致性问题。

进程模拟6.2.3节中的*EIGStop*算法，模拟的方法是：在它们一收到信息时就将所接收的信息转播出去，并像前面一样在*EIG*树中记录这些值。每个进程必定在完成把值记录到自己树中时做出决定。方法是确保至少$(f + 1)(d + l)$的时间已经过去。

824 给出这样一个算法的详细代码，证明它工作正确，并分析它的时间复杂度。

25.5 针对在定理25.6证明中定义和使用的通用同步模型，证明与定理6.33类似的结论。(提示：该证明非常类似于定理6.33的证明。)

25.6 补充定理25.15的证明细节。具体而言，证明：第0轮的时间$T(0) - T'$为$O(l_2)$；对于$k \geq 1$，第k轮的时间$T(k) - T(k-1)$最多为$S + O(l_2)$。

25.7 对于任意的f，$0 \leq f \leq n$，描述*PSynchAgreement*算法的一个特定可容许定时运行，其中输入到达所有端口、有至多f个故障，而直到其中每个进程都发生故障或决定的时间由你自己确定。

25.8 假设不使用保证所有消息在时间d内递送的通道，而使用只保证在时间d内递送最早消息的通道。针对这一模型修改*PSynchAgreement*算法，请尽量降低所得算法的时间复杂度。

25.9 研究问题：为部分同步模型设计一个比*PSynchAgreement*算法更有效的一致性算法。你能使时间上界达到$Ld + fd$ (加低阶谓词) 吗?

25.10 证明引理25.19。(提示：使用类似于定理6.33证明中使用的链证明。证明以满足下面时间约束的一个同步定时运行子集为基础：对于每个$r \in \mathbb{N}$，在时间间隔$[rd, (r + 1)d)$内发送的所有消息都正好在时间$(r + 1)d$时被递送。这个子集中的定时运行类似于同步模型中的运行，可以使用同样类型的链证明。)

25.11 研究问题：证明一个比定理25.17所给的更好的部分同步模型中到达一致性的时间下界。你能到达下界$Ld + fd$吗?还能更好吗?

25.12 研究问题：求一个你所能得到的部分同步模型中Byzantine一致性问题的最好上界和下界。

25.13 研究问题：在有f停止故障的部分同步网络模型中，考虑21.5节所定义的k-一致性问题。对所有无故障进程都做出决定的时间给出一个好的上界和下界。你能达到大约为$Ld + \frac{f}{k} + d$

825 的界限吗?(该界限是根据*FloodMin*算法和定理7.14针对同步网络环境推测的。)

25.14 证明定理25.23，同步进程和异步通道的发送/接收网络中一致性问题的不可能性结论。(提示：用逻辑时间的*WelchTime*实现，说明如何用异步模型模拟这个模型的算法。*WelchTime*算法使用的*clock*值可以通过统计步来维护。)

25.15 证明：如果$n \leq 2f$，在时间界限最终成立的模型中一致性问题是无解的。

25.16 完成定理25.25的证明。即，

a）精确定义*clock*管理策略。

b）认真陈述所需要的同步度声明和增长率声明。

c）精确描述基于时钟的模拟，以完成算法B的描述。

d）证明：B保证有f故障终止的一致性问题的正确性条件。

25.17 分析在习题25.16所构造算法B的时间复杂度。

25.18 考虑在未知时间界限模型中一致性问题的可解性。在这个模型中，我们假设进程步时间的下界l_1和上界l_2，$0 < l_1 \leq l_2 < \infty$，并且假设每个消息的递送时间上界$d$，但是这些界限对进程而言是“未知的”。（即，在不同的运行中它们可以不同，尽管每个运行在整个运行过程中有固定的界限。）

参照定理12.12，证明针对未知时间界限模型的一个类似结果。

25.19 研究问题：使用23.3.3节中的模拟方法，重新证明*PSynchFD*和*PSynchAgreement*的时间界限结果。

25.20 在部分同步网络模型中，求16.6节中会话问题的时间复杂度的好的上界和下界。

25.21 在部分同步共享存储器模型中，求实现13.3节所定义原子对象快照问题的时间复杂度的好的上界和下界。（一定要认真描述所要测量的是什么。） 826

25.22 研究问题：在部分同步环境中，给出分布式计算中其他感兴趣问题的时间复杂度上界和下界。除本书中提到的问题外，考察发生在实际通信系统、分布式操作系统和实时进程控制系统中的问题。你也可以抛开本书所采用的部分同步环境的形式化表达方法。

25.23 研究问题：给出一个通用描述结果，精确描述在部分同步系统中什么是可以计算的、时间复杂度是多少、建立部分同步模型的功能与同步和异步模型的功能之间关系的转换结果。 827～828

参考文献

[1] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. In J. W. de Bakker et al., editors, *Real-Time: Theory in Practice* (REX Workshop, Mook, The Netherlands, June 1991), volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, New York, 1992.

[2] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 291–302, Toronto, Ontario, Canada, August 1988.

[3] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[4] Yehuda Afek, Hagit Attiya, Alan Fekete, Michael Fischer, Nancy Lynch, Yishay Mansour, Da-Wei Wang, and Lenore Zuck. Reliable communication over unreliable channels. *Journal of the ACM*, 41(6):1267–1297, November 1994.

[5] Yehuda Afek and Eli Gafni. End-to-end communication in unreliable networks. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 131–148, Toronto, Ontario, Canada, August 1988. ACM, New York.

[6] Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM Journal on Computing*, 20(2):376–394, April 1991.

[7] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, 1986.

[8] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

829

[9] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[10] Rajeev Alur and Gadi Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the Real-Time Systems Symposium*, pages 12–21, Phoenix, December 1992. IEEE, Los Alamitos, Calif.

[11] James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, April 1993.

[12] James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, May 1994.

[13] Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 82–93, Los Angeles, April 1980.

[14] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[15] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110–135, February 1975.

[16] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

[17] H. Attiya and M. Mavronicolas. Efficiency of semisynchronous versus asynchronous networks. *Mathematical Systems Theory*, 27(6):547–571, November/December 1994.

[18] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[19] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment. In *28th Annual Symposium on Foundations of Computer Science*, pages 337–346. IEEE, Los Alamitos, Calif., October 1987.

[20] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, July 1990.

[21] Hagit Attiya, Shlomi Dolev, and Jennifer L. Welch. Connection management without retaining information. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, volume II (Software Technology), pages 622–631, Wailea, Hawaii, January 1995. IEEE, Los Alamitos, Calif.

[22] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, January 1994.

[23] Hagit Attiya, Michael Fischer, Da-Wei Wang, and Lenore Zuck. Reliable communication using unreliable channels. Manuscript, 1989.

[24] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast?

830

Journal of the ACM, 41(4):725–763, July 1994.

[25] Hagit Attiya and Nancy A. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. *Information and Computation*, 110(1):183–232, April 1994.

[26] Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 29–40, Ithaca, N.Y., August 1993.

[27] Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing in an anonymous ring. *Journal of the ACM*, 35(4):845–875, October 1988.

[28] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[29] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[30] Baruch Awerbuch. Reducing complexities of the distributed max-flow and breadth-first search algorithms by means of network synchronization. *Networks*, 15(4):425–437, winter 1985.

[31] Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 230–240, New York, May 1987.

831

[32] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear cost sequential and distributed constructions of sparse neighborhood covers. In *34th Annual Symposium on Foundations of Computer Science*, pages 638–647, Palo Alto, Calif., November 1993. IEEE, Los Alamitos, Calif.

[33] Baruch Awerbuch and Robert G. Gallager. Distributed BFS algorithms. In *26th Annual Symposium on Foundations of Computer Science*, pages 250–256, Portland, Ore., October 1985. IEEE, Los Alamitos, Calif.

[34] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A tradeoff between information and communication in broadcast protocols. *Journal of the ACM*, 37(2):238–256, April 1990.

[35] Baruch Awerbuch and David Peleg. Sparse partitions. In *31st Annual Symposium on Foundations of Computer Science*, volume II, pages 503–513, St. Louis, October 1990. IEEE, Los Alamitos, Calif.

[36] Baruch Awerbuch and David Peleg. Routing with polynomial

communication-space trade-off. *SIAM Journal of Discrete Mathematics*, 5(2):151–162, 1992.

[37] Baruch Awerbuch and Michael Saks. A Dining Philosophers algorithm with polynomial response time. In *31st Annual Symposium on Foundations of Computer Science*, volume I, pages 65–74, St. Louis, October 1990. IEEE, Los Alamitos, Calif.

[38] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, Cambridge, U.K., 1990.

[39] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 42–51, Vancouver, British Columbia, Canada, August 1987.

[40] Amotz Bar-Noy, Danny Dolev, Daphne Koller, and David Peleg. Fault-tolerant critical section management in asynchronous environments. *Information and Computation*, 95(1):1–20, November 1991.

[41] Alan E. Baratz and Adrian Segall. Reliable link initialization procedures. *IEEE Transactions on Communications*, 36(2):144–152, February 1988. 832

[42] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.

[43] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[44] Dag Belsnes. Single-message communication. *IEEE Transactions on Communications*, COM-24(2):190–194, February 1976.

[45] M. Ben-Ari. *Principles of concurrent programming.* Prentice Hall, Englewood Cliffs, N.J., 1982.

[46] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, Montreal, Quebec, Canada, August 1983.

[47] Claude Berge. *Graphs and Hypergraph.* North-Holland, Amsterdam, 1973.

[48] Piotr Berman and Anupam A. Bharali. Distributed consensus in semi-synchronous systems. In *Proceedings of the Sixth International Parallel Processing Symposium*, pages 632–635, Beverly Hills, March 1992. IEEE, Los Alamitos, Calif.

[49] Piotr Berman and Juan A. Garay. Cloture voting: $n/4$-resilient distributed consensus in $t+1$ rounds. *Mathematical Systems Theory—An International Journal on Mathematical Computing Theory*, 26(1):3–20, 1993. Special issue on Fault-Tolerant Distributed Algorithms.

[50] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.

[51] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *Journal of Algorithms*, 11(3):420–440, September 1990.

[52] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

833 [53] Bard Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Communications*, 37(12):1506–1514, December 1988.

[54] Gregor Bochmann and Jan Gecsei. A unified method for the specification and verification of protocols. In B. Gilchrist, editor, *Information Processing 77* (Toronto, August 1977), volume 7 of *Proceedings of IFIP Congress*, pages 229–234. North-Holland, Amsterdam, 1977.

[55] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 91–100, San Diego, May 1993.

[56] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[57] Gabriel Bracha and Sam Toueg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, December 1987.

[58] Michael F. Bridgland and Ronald J. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 52–63, Vancouver, British Columbia, Canada, August 1987.

[59] Manfred Broy. Functional specification of time sensitive communicating systems. In W. P. de Roever, J. W. de Bakker, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (REX Workshop, Mook, The Netherlands, May/June 1989), volume 430 of *Lecture Notes in Computer Science*, pages 153–179. Springer-Verlag, New York, 1990.

[60] James E. Burns. Mutual exclusion with linear waiting using binary shared

variables. *ACM SIGACT News*, 10(2):42–47, summer 1978.

[61] James E. Burns. A formal model for message passing systems. Technical Report TR-91, Computer Science Department, Indiana University, Bloomington, September 1980.

[62] James E. Burns, Paul Jackson, Nancy A. Lynch, Michael J. Fischer, and Gary L. Peterson. Data requirements for implementation of N-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, January 1982.

[63] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993. 834

[64] O. S. F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–148, February 1983.

[65] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[66] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[67] K. M. Chandy and J. Misra. The Drinking Philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[68] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[69] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Mass., 1988.

[70] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[71] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, May 1979.

[72] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8(4):391–401,

July 1982.

[73] Soma Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.

[74] Soma Chaudhuri, Rainer Gawlick, and Nancy Lynch. Designing algorithms for distributed systems with partially synchronized clocks. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 121–132, Ithaca, N.Y., August 1993.

[75] Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight bounds for k-set agreement. Technical Report 95/4, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, Mass. To appear.

[76] Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. A tight lower bound for k-set agreement. In *34th Annual Symposium on Foundations of Computer Science*, pages 206–215, Palo Alto, Calif., November 1993. IEEE, Los Alamitos, Calif.

[77] Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. A tight lower bound for processor coordination. In Donald S. Fussell and Miroslaw Malek, editors, *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, chapter 1, pages 1–18. Kluwer Academic, Boston, 1995. (Selected papers from *Second International Workshop on Responsive Computer Systems*, Lincoln, N.H., September 1993.)

[78] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 86–97, Vancouver, British Columbia, Canada, 1987.

[79] Ching-Tsun Chou and Eli Gafni. Understanding and verifying distributed algorithms using stratified decomposition. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 44–65, Toronto, Ontario, Canada, August 1988.

[80] Manhoi Choy and Ambuj K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 593–602, Victoria, British Columbia, Canada, May 1992.

[81] William Douglas Clinger. *Foundations of Actor Semantics.* Ph.D. thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, June 1981. University Microfilms, Ann Arbor, Mich.

[82] Brian A. Coan. *Achieving Consensus in Fault-Tolerant Distributed Computer Systems: Protocols, Lower Bounds, and Simulations.* Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, June 1987.

[83] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, Cambridge, Mass./New York, 1990.

[84] Armin B. Cremers and Thomas N. Hibbard. Mutual exclusion of N processors using an $O(N)$-valued message variable. In G. Ausiello and C. Böhm, editors, *Automata, Languages and Programming: Fifth Colloquium* (5th ICALP, Udine, Italy, July 1978), volume 62 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, New York, 1978.

[85] Armin B. Cremers and Thomas N. Hibbard. Arbitration and queueing under limited shared storage requirements. Technical Report 83, Department of Informatics, University of Dortmund, March 1979.

[86] N. G. de Bruijn. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 10(3):137–138, March 1967.

[87] W. P. de Roever and F. A. Stomp. A correctness proof of a distributed minimum-weight spanning tree algorithm. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 440–447, Berlin, September 1987. IEEE, Los Alamitos, Calif.

[88] Richard A. DeMillo, Nancy A. Lynch, and Michael J. Merritt. Cryptographic protocols. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 383–400, San Francisco, May 1982.

[89] Harish Devarajan. A correctness proof for a network synchronizer. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, May 1993. Technical Report MIT/LCS/TR-588.

[90] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[91] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[92] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[93] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal of Computing*, 12(4):656–666, November 1983.

[94] Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, March 1982.

[95] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[96] Danny Dolev, Michael J. Fischer, Rob Fowler, Nancy A. Lynch, and H. Raymond Strong. An efficient algorithm for Byzantine agreement without authentication. *Information and Control*, 52(3):257–274, March 1982.

[97] Danny Dolev, Maria Klawe, and Michael Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, September 1982.

[98] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[99] Danny Dolev, Rudiger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[100] Danny Dolev and Nir Shavit. Bounded concurrent time-stamp systems are constructible. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 454–466, Seattle, May 1989. To appear in *SIAM Journal of Computing*.

[101] Danny Dolev and H. Raymond Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 401–407, San Francisco, May 1982.

[102] Cynthia Dwork, Maurice Herlihy, Serge A. Plotkin, and Orli Waarts. Time-lapse snapshots. In D. Dolev, Z. Galil, and M. Rodeh, editors, *Theory of Computing and Systems* (ISTCS '92, Israel Symposium, Haifa, May 1992), volume 601 of *Lecture Notes in Computer Science*, pages 154–170. Springer-Verlag, New York, 1992.

[103] Cynthia Dwork, Maurice P. Herlihy, and Orli Waarts. Contention in shared memory algorithms. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 174–183, San Diego, May 1993. Expanded version in Technical Report CRL 93/12, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, Mass.

838

[104] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[105] Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in

a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October 1990.

[106] Cynthia Dwork and Dale Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, Montreal, Quebec, Canada, August 1983.

[107] Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 655–666, Victoria, British Columbia, Canada, May 1992. Preliminary. Final version to appear in *Journal of the ACM*.

[108] Murray A. Eisenberg and Michael R. McGuire. Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM*, 15(11):999, November 1972.

[109] A. Fekete, N. Lynch, and L. Shrira. A modular proof of correctness for a network synchronizer. In J. van Leeuwen, editor, *Distributed Algorithms* (2nd International Workshop, Amsterdam, July 1987), volume 312 of *Lecture Notes in Computer Science*, pages 219–256. Springer-Verlag, New York, 1988.

[110] A. D. Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4(1):9–29, March 1990.

[111] A. D. Fekete. Asynchronous approximate agreement. *Information and Computation*, 115(1):95–124, November 15, 1994.

[112] Alan Fekete, Nancy Lynch, Yishay Mansour, and John Spinelli. The impossibility of implementing reliable communication in the face of crashes. *Journal of the ACM*, 40(5):1087–1107, November 1993.

[113] Paul Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous Byzantine agreement. To appear in *SIAM Journal on Computing*. Preliminary version appeared as Technical Report MIT/LCS/TM-425.b, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, December 1992. 839

[114] Paul Neil Feldman. *Optimal Algorithms for Byzantine Agreement*. Ph.D. thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, June 1988.

[115] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, Brisbane, Australia, February 1988.

[116] Michael Fischer. Re: Where are you? E-mail message to Leslie Lam-

port. Arpanet message number 8506252257.AA07636@YALE-BULLDOG .YALE.ARPA (47 lines), June 25, 1985.

[117] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Research Report YALEU/DCS/RR-273, Yale University, Department of Computer Science, New Haven, Conn., June 1983.

[118] Michael J. Fischer, Nancy D. Griffeth, and Nancy A. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, May 1982.

[119] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

[120] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure. In *20th Annual Symposium on Foundations of Computer Science*, pages 234–254, San Juan, Puerto Rico, October 1979. IEEE, Los Alamitos, Calif.

[121] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, January 1989.

[122] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, January 1986.

[123] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 840 32(2):374–382, April 1985.

[124] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science* (New York, April 1966), volume 19 of *Proceedings of the Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, 1967.

[125] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, N.J., 1962.

[126] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.

[127] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34(1):98–115, January 1987.

[128] Harold N. Gabow. Scaling algorithms for network problems. *Journal of*

Computer and System Sciences, 31(2):148–168, October 1985.

[129] Eli Gafni. Personal communication, April 1994.

[130] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[131] Robert G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, January 1982.

[132] Juan A. Garay, Shay Kutten, and David Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *34th Annual Symposium on Foundations of Computer Science*, pages 659–668, Palo Alto, Calif., November 1993. IEEE, Los Alamitos, Calif.

[133] Juan A. Garay and Yoram Moses. Fully polynomial Byzantine agreement in $t + 1$ rounds. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 31–41, San Diego, May 1993.

[134] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, Palo Alto, Calif., December 1991.

[135] Rainer Gawlick, Nancy Lynch, and Nir Shavit. Concurrent time-stamping made simple. In D. Dolev, Z. Galil, and M. Rodeh, editors, *Theory of Computing and Systems* (ISTCS '92, Israel Symposium, Haifa, May 1992), volume 601 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, New York, 1992.

841

[136] Rainer Gawlick, Roberto Segala, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. In Serge Abiteboul and Eli Shamir, editors, *Automata, Languages and Programming* (21st International Colloquium, ICALP '94, Jerusalem, July 1994), volume 820 of *Lecture Notes in Computer Science*, pages 166–177. Springer-Verlag, New York, 1994.

[137] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, Calif., December 1979. ACM, New York.

[138] Virgil D. Gligor and Susan H. Shattuck. On deadlock detention in distributed systems. *IEEE Transactions on Software Engineering*, SE-6(5):435–440, September 1980.

[139] Kenneth Goldman and Kathy Yelick. A unified model for shared-memory and message-passing systems. Technical Report WUCS–93–35, Washington University, St. Louis, June 1993.

[140] Kenneth J. Goldman and Nancy Lynch. Quorum consensus in nested transaction systems. *ACM Transactions on Database Systems*, 19(4):537–585, December 1994.

[141] Kenneth J. Goldman and Nancy A. Lynch. Modelling shared state in a shared action model. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 450–463, Philadelphia, June 1990.

[142] J. N. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3.F, page 465. Springer-Verlag, New York, 1978.

[143] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, second edition, chapter 5, pages 97–145. ACM Press/Addison-Wesley, New York/Reading, Mass., 1993.

[144] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *Journal of the ACM*, 42(1):186–203, January 1995.

[145] Joseph Y. Halpern, Yoram Moses, and Orli Waarts. A characterization of
842 eventual byzantine agreement. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 333–346, Quebec City, Quebec, Canada, August 1990.

[146] Joseph Y. Halpern and Lenore D. Zuck. A little knowledge goes a long way: Knowledge-based proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, July 1992.

[147] Frank Harary. *Graph Theory*. Addison-Wesley, Reading, Mass., 1972.

[148] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. Technical Memo MIT/LCS/TM-511, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, November 1994. Abbreviated version in *Proceedings of the Real-Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994. IEEE, Los Alamitos, Calif. Later version to appear in C. Heitmeyer and D. Mandrioli, editors *Formal Methods for Real-time Computing*, chapter 4, *Trends in Software* series, John Wiley & Sons, New York.

[149] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[150] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Pro-*

gramming Languages and Systems, 13(1):124–149, January 1991.

[151] Maurice Herlihy and Nir Shavit. A simple constructive computatability theorem for wait-free computation. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 243–262, Montreal, Quebec, Canada, May 1994.

[152] Maurice P. Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 111–120, San Diego, May 1993.

[153] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[154] Maurice Peter Herlihy. *Replication Methods for Abstract Data Types.* Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, May 1984. Technical Report MIT/LCS/TR-319.

[155] Lisa Higham and Teresa Przytycka. A simple, efficient algorithm for maximum finding on rings. In André Schiper, editor, *Distributed Algorithms* (7th International Workshop, WDAG '93, Lausanne, Switzerland, September 1993), volume 725 of *Lecture Notes in Computer Science*, pages 249–263. Springer-Verlag, New York, 1993.

[156] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 23(11):627–628, November 1980.

[157] Gary S. Ho and C. V. Ramamoorthy. Protocols for deadlock detection in distributed database systems. *IEEE Transactions on Software Engineering*, SE-8(6):554–557, November 1982.

[158] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[159] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, United Kingdom, 1985.

[160] Pierre A. Humblet. A distributed algorithm for minimum weight directed spanning trees. *IEEE Transactions on Communications*, COM-31(6):756–762, June 1983.

[161] Sreekaanth S. Isloor and T. Anthony Marsland. An effective "on-line" deadlock detection technique for distributed database management systems. In *Proceedings of COMPSAC 78: IEEE Computer Society's Second International Computer Software and Applications Conference*, pages 283–288, Chicago, November 1978.

843

[162] Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, July 1993.

[163] Amos Israeli and Meir Pinhasov. A concurrent time-stamp scheme which is linear in time and space. In A. Segall and S. Zaks, editors, *Distributed Algorithms: Sixth International Workshop* (WDAG '92, Haifa, Israel, November 1992), volume 647 of *Lecture Notes in Computer Science*, pages 95–109. Springer-Verlag, New York, 1992.

[164] Wil Janssen and Job Zwiers. From sequential layers to distributed processes: Deriving a distributed minimum weight spanning tree algorithm. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 215–227, Vancouver, British Columbia, Canada, August 1992.

844

[165] Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[166] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In Jan Van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, chapter 17, pages 869–942. Elsevier/MIT Press, New York/Cambridge, 1990.

[167] Jon Kleinberg, Hagit Attiya, and Nancy Lynch. Trade-offs between message delivery and quiesce times in connection management protocols. In *Proceedings of ISTCS 1995: The Third Israel Symposium on Theory of Computing and Systems*, pages 258–267, Tel Aviv, January 1995. IEEE, Los Alamitos, Calif.

[168] Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, May 1966.

[169] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, second edition. Addison-Wesley, Reading, Mass., 1973.

[170] Dénes König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 8:114–134, 1926.

[171] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 218–228, Calgary, Alberta, Canada, August 1986.

[172] Jaynarayan H. Lala. A Byzantine resilient fault-tolerant computer for nuclear power plant applications. In *FTCS: 16th Annual International Symposium on Fault-Tolerant Computing Systems*, pages 338–343, Vienna, July 1986. IEEE, Los Alamitos, Calif.

[173] Jaynarayan H. Lala, Richard. E. Harper, and Linda S. Alger. A design approach for ultrareliable real-time systems. *Computer*, 24(5):12–22, May 1991. Issue on Real-time Systems.

[174] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[175] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. 845

[176] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[177] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[178] Leslie Lamport. The weak Byzantine generals problem. *Journal of the ACM*, 30(3):669–676, July 1983.

[179] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

[180] Leslie Lamport. The mutual exclusion problem. Part II: Statement and solutions. *Journal of the ACM*, 33(2):327–348, April 1986.

[181] Leslie Lamport. On interprocess communication, Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, April 1986.

[182] Leslie Lamport. On interprocess communication, Part II: Algorithms. *Distributed Computing*, 1(2):86–101, April 1986.

[183] Leslie Lamport. The part-time parliament. Research Report 49, Digital Systems Research Center, Palo Alto, Calif., September 1989.

[184] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[185] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In Jan Van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 18, pages 1157–1199. Elsevier/MIT Press, New York/Cambridge, 1990.

[186] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Research Report 44, Digital Equipment Corporation, Systems Research Center, Palo Alto, Calif., May 1989.

[187] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[188] Butler Lampson, Nancy Lynch, and Jørgen Søgaard-Andersen. At-most-once message delivery: A case study in algorithm verification. In W. R. Cleaveland, editor, *CONCUR'92* (Third International Conference on Concurrency Theory, Stony Brook, N.Y., August 1992), volume 630 of *Lecture Notes in Computer Science*, pages 317–324. Springer-Verlag, New York, 1992.

[189] Butler Lampson, William Weihl, and Umesh Maheshwari. Principles of Computer Systems: Lecture Notes for 6.826, Fall 1992. Research Seminar Series MIT/LCS/RSS 22, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, July 1993.

[190] Butler W. Lampson, Nancy A. Lynch, and Jørgen F. Søgaard-Andersen. Correctness of at-most-once message delivery protocols. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyan, editors, *Formal Description Techniques VI* (Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques, FORTE '93, Boston, October, 1993) *IFIP Transactions C*, pages 385–400. North-Holland, Amsterdam, 1994.

[191] Gérard Le Lann. Distributed systems—towards a formal approach. In Bruce Gilchrist, editor, *Information Processing 77* (Toronto, August 1977), volume 7 of *Proceedings of IFIP Congress*, pages 155–160. North-Holland, Amsterdam, 1977.

[192] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the Dining Philosophers problem. In *Proceedings of Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 133–138, Williamsburg, Va., January 1981.

[193] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, Calif., 1992.

[194] Harry R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical Report TR-15-89, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, Mass., 1989.

[195] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, Englewood Cliffs, N.J., 1981.

[196] Ming Li and Paul M. B. Vitányi. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, July 1996.

[197] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 29–39, Calgary, Alberta, Canada, August 1986.

[198] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[199] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, editor, *Parallel and Distributed Computing*, volume 4 of *Advances in Computing Research*, pages 163–183. JAI Press, Greenwich, Conn., 1987.

[200] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing*, 15(4):1036–1053, November 1986.

[201] Victor Luchangco. Using simulation techniques to prove timing properties. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, June 1995.

[202] Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques* (FORTE '94, Berne, Switzerland, October 1994), pages 259–273. Chapman and Hall, New York, 1995.

[203] N. Lynch. Concurrency control for resilient nested transactions. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 166–181, Atlanta, March 1983.

[204] Nancy Lynch. Simulation techniques for proving properties of real-time systems. In W. P. de Roever, J. W. de Bakker, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives* (REX School/Symposium, Noordwijkerhout, The Netherlands, June 1993), volume 803 of *Lecture Notes in Computer Science*, pages 375–424. Springer-Verlag, New York, 1994.

[205] Nancy Lynch. Simulation techniques for proving properties of real-time systems. In Sang H. Son, editor, *Advances in Real-Time Systems*, chapter 13, pages 299–332. Prentice Hall, Englewood Cliffs, N.J., 1995.

[206] Nancy Lynch, Yishay Mansour, and Alan Fekete. The data link layer: Two impossibility results. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 149–170, Toronto, Ontario, Canada, August 1988.

[207] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, Calif., 1994.

[208] Nancy Lynch, Isaac Saias, and Roberto Segala. Proving time bounds for randomized distributed algorithms. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 314–323, Los Angeles, August 1994.

[209] Nancy Lynch and Nir Shavit. Timing-based mutual exclusion. In *Proceedings of the Real-Time Systems Symposium*, pages 2–11, Phoenix, December 1992. IEEE, Los Alamitos, Calif.

[210] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In J. W. de Bakker et al., editors, *Real-Time: Theory in Practice* (REX Workshop, Mook, The Netherlands, June 1991), volume 600 of *Lecture Notes in Computer Science*, pages 397–446. Springer-Verlag, New York, 1992.

[211] Nancy Lynch and Frits Vaandrager. Forward and backward simulations—Part II: Timing-based systems. *Information and Computation*, 128(1): 1–25, July 1996.

[212] Nancy Lynch and Frits Vaandrager. Action transducers and timed automata. Technical Memo MIT/LCS/TM-480.b, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, October 1994.

[213] Nancy A. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal of Computer and System Sciences*, 23(2):254–278, October 1981.

[214] Nancy A. Lynch. Multivalued possibilities mappings. In W. P. de Roever, J. W. de Bakker, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (REX Workshop, Mook, The Netherlands, May/June 1989), volume 430 of *Lecture Notes in Computer Science*, pages 519–543. Springer-Verlag, New York, 1990.

[215] Nancy A. Lynch and Hagit Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6(2):121–139, September 1992.

[216] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*,

849

13(1):17–43, 1981. Special issue on Semantics of Concurrent Computation.

[217] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, April 1987. Technical Report MIT/LCS/TR-387. Abbreviated version in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, British Columbia, Canada, August, 1987.

[218] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, November 1988.

[219] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[220] Yishay Mansour and Baruch Schieber. The intractability of bounded protocols for on-line sequence transmission over non-FIFO channels. *Journal of the ACM*, 39(4):783–799, October 1992.

[221] John C. Martin. *Introduction to Languages and the Theory of Computation.* McGraw-Hill, New York, 1991.

[222] Friedemann Mattern. Virtual time and global states of distributed systems. In Michel Cosnard et al., editors, *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms* (Chateau de Bonas, Gers, France, October, 1988), pages 215–226. North-Holland, Amsterdam, 1989.

[223] John M. McQuillan, Gilbert Falk, and Ira Richer. A review of the development and performance of the ARPANET routing algorithm. *IEEE Transactions on Communications*, COM-26(12):1802–1811, December 1978.

[224] Daniel A. Menasce and Richard R. Muntz. Locking and deadlock detection in distributed data bases. *IEEE Transactions on Software Engineering*, SE-5(3):195–202, May 1979.

[225] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

[226] Michael Merritt, 1985. Unpublished Notes.

[227] Michael Merritt, Francesmary Modugno, and Mark R. Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, editors, *CONCUR '91: 2nd International Conference on Concurrency Theory* (Amsterdam, August 1991), volume 527 of *Lecture Notes in Computer Science*, pages

850

408–423. Springer-Verlag, New York, 1991.

[228] Robin Milner. An algebraic definition of simulation between programs. In *2nd International Joint Conference on Artificial Intelligence*, pages 481–489, Imperial College, London, September 1971. British Computer Society, London.

[229] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, United Kingdom, 1989.

[230] Shlomo Moran and Yaron Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, November 1987.

[231] Yoram Moses and Orli Waarts. Coordinated traversal: $(t + 1)$-round Byzantine agreement in polynomial time. *Journal of Algorithms*, 17(1):110–156, July 1994.

[232] Gil Neiger and Sam Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *Journal of the ACM*, 40(2):334–367, April 1993.

[233] Tobias Nipkow. Formal verification of data type refinement: Theory and practice. In W. P. de Roever, J. W. de Bakker, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (REX Workshop, Mook, The Netherlands, May/June 1989), volume 430 of *Lecture Notes in Computer Science*, pages 561–591. Springer-Verlag, New York, 1990.

851 [234] Ron Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, June 1982.

[235] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs, I. *Acta Informatica*, 6(4):319–340, 1976.

[236] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science* (5th GI Conference, Karlsruhe, Germany, March 1981), volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, New York, 1981.

[237] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[238] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[239] Gary L. Peterson. An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, October 1982.

[240] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[241] Gary L. Peterson and James E. Burns. Concurrent reading while writing II: The multi-writer case. In *28th Annual Symposium on Foundations of Computer Science*, pages 383–392, Los Angeles, October 1987. IEEE, Los Alamitos, Calif.

[242] Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 91–97, Boulder, Colo., May 1977.

[243] Amir Pnueli. Personal communication, 1988.

[244] Amir Pnueli and Lenore Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, January 1986.

[245] Stephen Ponzio. Consensus in the presence of timing uncertainty: Omission and Byzantine failures. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 125–138, Montreal, Quebec, Canada, August 1991.

[246] Stephen Ponzio. Bounds on the time to detect failures using bounded-capacity message links. In *Proceedings of the Real-time Systems Symposium*, pages 236–245, Phoenix, December 1992. IEEE, Los Alamitos, Calif. 852

[247] Stephen J. Ponzio. The real-time cost of timing uncertainty: Consensus and failure detection. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, June 1991. Technical Report MIT/LCS/TR-518.

[248] Michael O. Rabin. Randomized Byzantine generals. In *24th Annual Symposium on Foundations of Computer Science*, pages 403–409, Tucson, November 1983. IEEE, Los Alamitos, Calif.

[249] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, 1986.

[250] Michel Raynal. *Networks and Distributed Computation: Concepts, Tools, and Algorithms.* MIT Press, Cambridge, 1988.

[251] Michel Raynal and Jean-Michel Helary. *Synchronization and Control of Distributed Systems and Programs.* John Wiley & Sons, Ltd., Chichester, U.K., 1990.

[252] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981. Corrigendum in *Communications of the ACM*, 24(9):578,

September 1981.

[253] Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing*, pages 101–110, San Diego, May 1993.

[254] Russell Schaffer. On the correctness of atomic multi-writer registers. Technical Memo MIT/LCS/TM-364, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, June 1988.

[255] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[256] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, March 1994.

[257] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, August 1995.
853 Special issue on selected papers from CONCUR'94.

[258] Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, January 1983.

[259] A. Udaya Shankar. A simple assertional proof system for real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 167–176, Phoenix, December 1992. IEEE, Los Alamitos, Calif.

[260] A. Udaya Shankar and Simon S. Lam. A stepwise refinement heuristic for protocol construction. *ACM Transactions on Programming Languages and Systems*, 14(3):417–461, July 1992.

[261] Nir Shavit. *Concurrent Time Stamping.* Ph.D. thesis, Department of Computer Science, Hebrew University, Jerusalem, Israel, January 1990.

[262] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts*, third edition. Addison-Wesley, Reading, Mass., 1992.

[263] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, March 1994.

[264] Jørgen Søgaard-Andersen. *Correctness of Protocols in Distributed Systems.* Ph.D. thesis, Department of Computer Science, Technical University of Denmark, Lyngby, December 1993. ID-TR: 1993-131.

[265] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation

proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification* (5th International Conference, CAV '93, Elounda, Greece, June/July 1993), volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, New York, 1993.

[266] Edwin H. Spanier. *Algebraic Topology.* McGraw-Hill, New York, 1966.

[267] E. Sperner. Neuer beweis für die invarianz der dimensionszahl und des gebietes. *Abhandlungen Aus Dem Mathematischen Seminar Der Hamburgischen Universität*, 6:265–272, 1928.

[268] John M. Spinelli. Reliable communication on data links. Technical Report LIDS-P-1844, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, December 1988.

[269] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, August 1987.

[270] N. V. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, September 1976.

[271] Tom Stoppard. *Rosencrantz & Guildenstern Are Dead.* Grove Press, New York, 1968.

[272] Eugene Styer and Gary L. Peterson. Improved algorithms for distributed resource allocation. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 105–116, Toronto, Ontario, Canada, August 1988.

[273] Andrew S. Tanenbaum. *Computer Networks*, second edition. Prentice Hall, Englewood Cliffs, N.J., 1988.

[274] Y. C. Tay and W. Tim Loke. On deadlocks of exclusive AND-requests for resources. *Distributed Computing*, 9(2):77–94, October 1995.

[275] Gerard Tel. Assertional verification of a timer based protocol. In Timo Lepistö and Arto Salomaa, editors, *Automata, Languages and Programming* (15th International Colloquium, ICALP '88, Tempere, Finland, July 1988), volume 317 of *Lecture Notes in Computer Science*, pages 600–614. Springer-Verlag, New York, 1988.

[276] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge, U.K., 1994.

[277] Ewan Tempero and Richard Ladner. Recoverable sequence transmission protocols. *Journal of the ACM*, 42(5):1059–1090, September 1995.

[278] Ewan Tempero and Richard E. Ladner. Tight bounds for weakly-bounded protocols. In *Proceedings of the Ninth Annual ACM Symposium on Prin-*

854

ciples of Distributed Computing, pages 205–218, Quebec City, Quebec, Canada, August 1990.

[279] Russell Turpin and Brian A. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18(2):73–76, February 1984.

[280] Jan L. A. van de Snepscheut. The sliding-window protocol revisited. *Formal Aspects of Computing*, 7(1):3–17, 1995.

[281] George Varghese and Nancy A. Lynch. A tradeoff between safety and liveness for randomized coordinated attack protocols. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, 855 pages 241–250, Vancouver, British Columbia, Canada, August 1992.

[282] Paul M. B. Vitányi. Distributed elections in an Archimedean ring of processors. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 542–547, Washington, D.C., April/May 1984.

[283] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science*, pages 233–243, Toronto, Ontario, Canada, October 1986. IEEE, Los Alamitos, Calif. Corrigendum in *28th Annual Symposium on Foundations of Computer Science*, page 487, Los Angeles, October 1987.

[284] Da-Wei Wang and Lenore D. Zuck. Tight bounds for the sequence transmission problem. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 73–83, Edmonton, Alberta, Canada, August 1989.

[285] Jennifer Welch and Nancy Lynch. A modular Drinking Philosophers algorithm. *Distributed Computing*, 6(4):233–244, July 1993.

[286] Jennifer Lundelius Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–171, August 1987.

[287] Jennifer Lundelius Welch. *Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms.* Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, March 1988.

[288] Jennifer Lundelius Welch, Leslie Lamport, and Nancy Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 28–43, Toronto, Ontario, Canada, August 1988.

[289] John H. Wensley et al. SIFT: Design and analysis of a fault-tolerant

computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[290] Hubert Zimmerman. OSI reference model—the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, April 1980.

856

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

≈, 148
., 206
$\leq_\gamma$, 88, 88, 190
$\xrightarrow[p]{t}$, 358
$\overset{i}{\sim}$, 21,84, 148, 244,300
$\rightarrow_\beta$, 464, 468, 536
~, 148
|, 14, 211
Abadi, M., 770, 790
ABD, 575-580, 580, 587
ABDObject, 575, 576, 577, 575-580, 588
abort（放弃）, 182, 371, 669
ABP, 698, 699, 697-703, 708, 722, 728, 730
Abrahamson, K.,392
abstraction（抽象）, 6
Abu-Amara, H., 391, 685
accepting state（接受状态）, 19
action（动作）, 201, 256, 745
active process（活动进程）, 104, 162
active round（活动轮）, 40, 40-44
acts, 201, 202, 745
AcyclicDigraphRA, 656, 656-659, 664, 666, 667
add（加）, 170
admissible timed execution（合法定时运行）, 737, 774
admissible timed execution fragment（合法定时运行片段）, 746
admissible timed trace（合法定时轨迹）, 738, 746
adversary（对手）, 23, 87, 87-97, 144, 230, 358, 358, 675
Afek, Y., 76, 230, 449, 471, 524, 728, 729
affects（影响）, 190
agent（代理）, 260
Agrawala, A., 664
agreement,（一致性）686-688
  approximate（近似）, 见 approximate agreement
agreement condition（一致性条件）, 81,83, 85-87, 97, 100, 101, 160, 162, 177, 182, 374, 390, 671, 681, 682, 796
agreement problem（一致性问题）, 99-162, 177, 192, 374, 371-395, 586, 669, 672, 671-681, 684-685, 688, 796, 795-827
aircraft（飞行器）, 8, 81, 100, 152
airline reservation system（机票预订系统）, xix
algebraic topology（代数拓扑结构）, 165-177, 192, 392
algorithm（算法）
  asynchronous（异步）, 见 asynchronous algorithm
  asynchronous network（异步网络）, 见asynchronous network algorithm
  asynchronous shared memory（异步共享存储器）, 见 asynchronous shared memory algorithm
  comparison-based（基于比较的）,见 comparison-based algorithm
  counterexample（反例）, 见 coutnterexample algorithm
  partially synchronous（部分同步）, 见 partially synchronous algorithm
  partially synchronous network（部分同步网络）,见 partially synchronous network algorithm
  partially synchronous shared memory（部分同步共享存储）, 见 partially synchronous shared memory algorithm
  randomized（随机）, 见 randomized algorithm
  synchronous network（同步网络）,见 synchronous network algorithm
algorithm transformation（算法变换）, 6, 11, 531-563, 565-589, 596-600, 611, 614, 617, 643, 664, 670, 672, 685, 798-801
Alpern, B., 230
Alpha, 546, 546-549, 551-555,560, 561
Alternating Bit protocol（交替位协议）, 12, 462, 691
altimeter（高度计）, 8, 81, 100
Alur, R., 769, 790
Anderson, J., 449, 450
Angluin, D., 46
approximate agreement（近似一致性）, 9, 125, 161, 177, 177-182, 192-194, 390, 390-392, 394, 395, 669, 673, 682, 682-685

Arjomandi, E., 560
ARPANET, 506
Ashcroft, E., 23
Aspnes, J., 392
assertion（断言）
  invariant（不变式），见 invariant assertion
AsynchApproxAgreement, 682, 682-684, 688
AsynchBcastAck, 499, 499-500, 503, 504, 526,527, 555,620, 637
AsynchBellmanFord, 506, 506-509, 524, 528, 554, 634, 637, 771
AsynchBFS, 502, 502-508, 524, 527, 619, 625, 634, 637, 638
AsynchLCR, 477, 476-482, 523-525
asynchronous algorithm（异步算法），197, 255-454, 731, 735, 792
  network（网络），见asynchronous network algorithm
  shared memory（共享存储），见 asynchronous shared memory algorithm
asynchronous model（异步模型），xx, 5, 7, 196-234, 735, 795, 823, 827
  network,（网络），见asynchronous network model
  shared memory（共享存储器），见 asynchronous shared memory model
asynchronous network algorithm（异步网络算法），xxii, 10-12, 197, 455, 475-731
asynchronous network model（异步网络模型），7, 51, 178, 199, 255, 327, 367, 371, 391, 392, 455-473, 475
asynchronous shared memory algorithm（异步共享存储器算法），9-10, 197, 203, 235, 255-454, 643, 654, 672, 685
asynchronous shared memory model（异步共享存储器模型），7, 11, 199, 235-253, 256-259, 372, 455, 475, 531, 556, 565-589, 609, 641, 664, 669, 670, 681, 682, 684
asynchronous shared memory system（异步共享存储器系统），237-253
asynchronous system（异步系统），2
AsynchSpanningTree, 496, 496-499, 501, 502, 527, 619, 620, 633
atexecs, 737, 746
atomic object（原子对象），10, 235, 406, 397-454, 568, 574, 575, 587, 606, 607, 613, 614, 646
atomic transaction（原子事务），81, 82, 573, 586
Atomic Transactions（原子事务），13, 230, 231, 573, 586, 612
atomicity condition（原子性条件），404
atomicity property（原子性属性），402
Attiya, H., 47, 192, 392, 449, 450, 471, 560, 575, 587, 685, 686, 728, 729, 769, 770, 790, 823, 824
attraces, 738, 746
augmented run（强化运动），169
authenticated Byzantine failure model（带鉴别的Byzantine故障模型），108, 115, 115-116, 129, 152, 153, 155, 156
automaton state（自动机状态），256
automaton transformation（自动机变换），735, 751-754, 769
Awerbuch, B., 367, 450,524, 560, 664
axiomatic specification（公理式规格说明），459, 693, 707

B

backup（备份），604, 617
Bakery, 296, 296-300, 302, 326, 330, 332, 433, 452, 453, 572, 588, 643, 664
banking system（银行系统），xix, 530, 600-604, 609, 611-613, 626-627, 630, 632-633, 637
Bar-Noy, A., 152, 575, 587, 685, 686
Baratz, A., 729
Bartlett, K., 728
Bellman, R., 62, 76, 524
BellmanFord, 62, 62-63, 76, 79, 506-508, 554, 555
Belsnes, D., 729
Ben-Ari, M., 327
Ben-Or, M., 673, 685
BenOr, 673, 673-677, 685, 687
Berman, P., 152
Bermuda Triangle（Bermuda三角），168, 165-177, 193
Bernstein, P., 13, 192, 586
Beta, 547, 546-549, 551, 553, 554, 560, 561
BFS，见breadth-first search
Bharali, A., 823
binary Byzantine agreement（二元Byzantine一致），123-125
Biran, O., 392, 686
bit-reversal ring（位反转环），42, 42, 49
bivalence（双价），377, 686
blocking commit algorithm（阻塞提交算法），183, 183, 184-185, 190
Bloom, 441, 440-447, 450, 453, 454
Bloom, B., 230, 450
Borodin, A., 326, 327
Borowsky, E., 392
bottle（瓶），654
bounded bypass condition（有界绕过条件），277, 311-319, 326
bounded tag（有界标志），428-434, 691, 703-715

BoundedSnapshot, 430, 428-434, 449, 452
boundmap（边界地图）, 736
Bracha, G., 587, 636, 686
breadth-first search（广度优先搜索）, 8, 11, 51, 57, 57-61, 77-78, 475, 496, 501-506, 527, 528, 554, 560, 625, 634-635, 637, 638
breadth-first spanning tree（广度优先生成树）, 57, 57, 58, 62, 496
Bridgland, M., 685, 686
broadcast（广播）, 1, 11, 57-58, 60-61, 63, 78, 457, 470, 475, 496, 498-500, 524, 526, 527, 555
broadcast channel（广播通道）, 466-468, 471
broadcast network（广播网络）, 641, 646
broadcast system（广播系统）, 457, 468, 466-469, 580, 585, 591, 594-595, 598, 606-610, 612-614, 669-689
BufferMainME, 316, 316-320, 326, 332, 333
Burns, J., 326, 327, 450, 524
BurnsME, 294, 294-296, 302, 326, 329, 330
busy-waiting（忙等待）, 572
Byzantine agreement（Byzantine一致）, 23, 101, 100-102, 116-161, 178
  binary（二元）, 见 binary Byzantine agreement
Byzantine failure（Byzantine故障）, 2, 9, 20, 99-102, 116-160, 177-182, 371, 458, 687, 688, 716, 823, 825
Byzantine Firing Squad（Byzantine射击队）, 159
Byzantine generals（Byzantine将军）, 99, 152
ByzApproxAgreement, 178, 178

C

c-connected graph（*c*连通图）, 135
c-symmetric ring（*c*对称环）, 42, 42-44, 47, 49
caching（缓存）, 572
canonical wait-free atomic object（规范无等待原子对象）, 398, 408-411, 449, 451, 452
Carvalho, O., 664
cause, 460, 471, 693
chain argument（链证明）, 145-151, 160, 165, 194, 378, 825
Chandra, T., 685
Chandy, K. M., 13, 367, 449, 612, 636, 664
ChandyLamport, 627, 627-632, 635-638
Chang, E., 27, 47, 523, 524
change, 170
channel（通道）, 18, 457, 476
channel failure（通道故障）, 19
channel I/O automaton（通道I/O自动机）, 201, 204, 204-205
channel MMT automaton（通道MMT自动机）, 738, 738-739
channel multiplexing（通道复用）, 472, 545, 568, 707, 710, 711
Chaudhuri, S., 192, 392, 612
child pointer（孩子指针）, 59, 58-60, 498
Chor, B., 392
Chou, C. T., 524
Choy, M., 367, 664
CirculatingToken, 643, 644, 643-646, 649, 664, 666
Clock, 213
clock（时钟）, 8, 735
  real-time（实时）, 见 real-time clock
clock synchronization（时钟同步）, 178
closed automaton（封闭自动机）, 202, 222
CLU, 23
cluster decomposition（簇分解）, 548, 560, 562
ClusterForest, 550, 550-551
ClusterSynch, 549, 549-552, 561
Coan, B., 152
Coloring, 351, 349-354, 367-369, 654, 654-655, 666, 667
coloring（着色）, 45, 350
commit（提交）, 9, 81, 82, 85, 93, 95, 161, 182, 182-192, 194-195, 371, 669
common node（公共节点）, 121, 156
common set of nodes（公共节点集）, 121
communication（通信）, xix, 2, 823, 827
  interprocess（进程间）, 见 interprocess communication
  reliable（可靠）,见 reliable communication
communication complexity（通信复杂度）, 21, 21, 466, 469
communication failure（通信故障）, 2, 3, 9, 12, 19, 81-97, 182, 183, 691-731
communication pattern（通信模式）, 87, 87, 89,92, 143, 143, 168,190
good（良好）, 见good communication pattern
compare-and-swap operation（比较-交换操作）, 248, 388, 393
comparison-based algorithm（基于比较的算法）, 27-35, 39, 38-45, 47, 476-486
compatible（兼容的）
  general timed automata（兼容通用定时自动机）, 754
  I/O automata（兼容I/O自动机）, 208, 232
  MMT automata（兼容MMT自动机）, 742
  signatures（兼容签名）, 207
  timed signatures（兼容定时签名）, 754
  trace properties（兼容轨迹属性）, 217
  variable types（兼容变量类型）, 249, 411
complete finitely consistent execution（完全有限一致运

行），712
complexity（复杂度）
  communication（通信），见 communication complexity
  time（时间），见 time complexity
complexity measure（复杂度衡量），21-22
composition（合成），6
  atomic objects（原子对象合成），398, 411, 451
  general timed automata（通用定时自动机合成），754, 754-756, 769
  I/O automata（I/O自动机合成），199, 208, 206-211, 231, 237, 238
  I/O automaton（I/O自动机合成），464, 487, 521
  MMT automata（MMT自动机合成），742, 741-744, 770
  parallel（并行），见 parallel composition
  signatures（签名合成）208
  timed signatures（定时签名合成），754
  trace properties（轨迹属性合成），218
  variable types（变量类型合成），249, 411
compositional reasoning（合成推理），221-224
computer-assisted verification（机器辅助证明），231, 770
concurrency control（并发控制），2, 230, 573, 586
conflicting users（冲突用户），336
congestion（拥塞），22, 32
conn, 135
connected graph（连通图），391, 395
connection management（连接管理），824
connectivity（连通性），135
consensus（一致性），2, 4, 8-10, 12, 13, 15, 31, 81-195, 235, 371-395, 413, 641, 733, 736, 795-827
consistent（一致的），705, 712
consistent broadcast（一致广播），125, 125-129, 152, 157
consistent global snapshot（一致全局快照），12, 449, 496, 612, 617, 626, 625-639
consistent with（与……一致），704, 712
ConsistentBroadcast, 126, 126-127, 152, 157
ConvergeApproxAgreement, 179, 178-182, 193, 194, 682, 683, 689
convergecast（敛播），11, 60, 70, 79, 45, 496, 498-501, 503-504, 524, 526, 527, 555, 620, 633, 637
coordinated attack（协同攻击），82, 83, 81-97, 99, 161, 183
corresponding executions（对应运行），228, 325, 540, 697, 701
corresponding messages（对应消息），40
corresponding states（对应状态），40, 40
counterexample algorithm（反例算法），36, 36, 315, 706
CountMoney, 601, 600-606, 611-613
cover（覆盖），303
crash（崩溃），691, 693, 715-728, 731
Cremers, A., 326, 327
critical region（临界区），9, 255, 259, 336

D

data abstraction（数据抽象），23
data link（数据链路），12, 228, 455, 691-731
database（数据库），192, 207, 230, 231, 259, 371, 609, 612, 637, 669
de Bruijn, J., 326
de, Roever, W., 524
deadlock（死锁），12, 344, 349, 617, 636
deadlock detection（死锁检测），2, 617, 635, 635-636, 639
debugging（调试），604, 617, 633
dec, 151
decider（决定者），378, 383, 393, 686
decision mapping（决定映射），389
decision problem（决定问题），389, 389-391, 394, 395, 684, 689
default value（默认值），90, 103, 179
Dekker, T., 265, 326
DeMillo, R., 153
depends on（依赖于），465, 468, 536, 592, 593
deterministic automaton（确定自动机），233, 376
Devarajan, H., 560
diam, 17, 457, 495
diameter（直径），17, 52, 61, 78, 495
diffusing algorithm（发散算法），618, 618-625
diffusing computation（发散计算），524
digital signature（数字签名），115, 129
Dijkstra, E., 9, 65, 255, 265, 326, 367, 524, 636
DijkstraME, 265, 267, 265-276, 279, 283, 295, 302, 326-328, 780, 790, 791
DijkstraScholten, 619, 622, 619-625, 635-638
Dill, D., 769
Dining Philosophers（哲学家用餐），10, 12, 335, 340, 339-349, 354-367
directed graph（有向图），17
directed spanning tree（有向生成树），57
disc, 745
discrete action（离散动作），745
disjoint lines（互不相交的线），488
distance（距离），17, 457
distributed consensus（分布式一致性），见 consensus

distributed database（分布式数据库），81, 82, 85, 93, 95, 182
distributed information processing（分布式信息处理），1
Dolev, D., 47, 152, 153, 192, 392, 449, 450, 523, 575, 587, 685, 686, 823
Dolev, S., 729
Draper Laboratories, 153
Drinking Philosophers（哲学家饮水），367, 654, 654, 658-664
dummy step（虚拟步/dummy步），376
duplication（复制）
  message（消息），见 message duplication
Dwork, C., 152, 153, 192, 252, 450, 823, 824
dynamic component creation（动态组成部分创建），207
dynamic process creation（动态进程创建），230

E

early stopping（提前终止），152, 157
EIG, 见exponential information gathering
EIG tree（EIG树），108, 108-116, 119, 152, 156
EIGByz, 120, 119-123, 129, 152, 156
EIGStop, 110, 109-115, 119, 120, 152, 154-156, 824
Eisenberg, M., 326
enabled action（允许行动），202
enabled task（允许任务），203
environment（环境/外界），241-243
environment node（环境节点/外界节点），19
environment process（环境进程/外界进程），19
event（事件），212
eventual time bound（最终时间界限），789, 819-823
exclusion condition（互斥条件），338, 653, 654
exclusion specification（互斥规格说明），336, 336-338, 367,653
exec, 144, 169, 358
execs, 206
execution（运行）
  asynchronous shared memory system（异步共享存储器系统运行），258
  I/O automaton（I/O自动机运行），199, 206, 206
  randomized synchronous system（随机同步系统运行），22, 22
  synchronous system（同步系统运行），18, 20, 20-21
  variable type（变量类型运行），249, 398
execution correspondence（运行对应关系），702
execution fragment（运行片段）
I/O automaton（I/O自动机运行片断），206
Executive, 319, 319-320, 333
exit protocol（退出协议），259
exit region（退出区），259
explicit resource specification（显式资源规格说明），336, 336-337, 349, 367, 653,667
exponential information gathering（指数信息收集），103, 108-116, 119-123
ext, 201, 745
extension（扩展），377
external action（外部动作），201, 745
external interface（外部接口），201
external signature（外部签名），201, 260
extsig, 201

F

f-failure termination condition（*f*故障终止条件），375, 407, 672, 796
f-simulation（*f*模拟），567, 583
failure（故障），2, 5, 6, 9, 10, 12, 19-20, 81-195, 251, 255, 371-454, 458, 461-464, 575-586, 669-689, 691-731, 769, 795-827
  Byzantine,见Byzantine failure
  channel（通道），见 channel failure
  communication（通信），见communication failure
  link（链路），见 link failure
  process（进程），见 process failure
  processor（处理器），见 processor failure
  stopping（停止），见 stopping failure
  transient（瞬时），见transient failure
failure detector（故障检测器），12, 669, 673, 677-681, 685, 796-798, 823
failure-free execution（无故障运行），373, 406
failure-free *l*-run（无故障*l*运动），169
failure-free run（无故障运动），148
failure-free termination condition（无故障终止条件），374, 406, 672, 796
fair execution fragment（公平运行片段），212, 232
fair trace（公平轨迹），212
fairexecs, 212
fairness（公平性），199, 203, 212-215, 240, 255,258-259, 261-262, 738, 784
fairtraces, 212
fault diagnosis（故障诊断），9, 81, 100, 153, 371, 669
fault-tolerance（容错），397, 407, 471, 531, 553, 575, 645, 672, 784-790
fault-tolerant multiprocessor（容错多处理器），100, 153
faulty process（故障进程 / 出错进程），143
Fekete, A., 13, 192, 230, 471, 560, 573, 586, 612, 728, 729
Feldman, P., 685

fetch-and-add operation（取-加操作）, 249, 388, 394
ff, 148
Fidge, C., 612
FIFO after a wait-free doorway（无等待门廊后的FIFO）, 296, 300
financial database（财务数据库）, 565
finite duplication（有限复制）, 462, 693, 695, 699, 722
finite nondeterminism（有限不确定性）, 451
finite-state automaton（有限状态自动机）, 19, 23
finitely consistent（有限一致）, 704, 712
first, 751
Fischer, M., 23, 152, 153, 251, 326, 327, 391, 471, 560, 587, 636, 685, 728, 729, 790
FischerME, 777, 774-788, 790, 791
FischerS, 786, 784-787,789,790, 792, 793
FivePacketHandshake, 722,724, 725, 721-729, 731
fixed-connection network（固定连通网络）, 2
Flaky Computer Corporation, 157, 331, 369, 562, 612, 667
Flatten, 174
FloodMax, 52, 52-56, 58, 61, 76, 77, 495, 531, 554
FloodMin, 163, 162-164, 192, 825
FloodSet, 103, 103-108, 152-155, 224, 562
Floyd, R., 23
FLP, 391, 685
Ford, L., 62, 76, 524
forest（森林）, 63
ForestSynch, 549-550, 550, 551, 552, 561
formal framework（形式化框架/正式框架）, 4
formal model（形式化模型/正式模型）, 6-8
Fowler, R., 152
Francez, N., 636
Frederickson, G., 47
FrontEnd, 542, 541-547, 550, 552, 553

G

Gabow, H., 524
Gafni, E., 76, 392, 449, 524, 728
Gallager, R., 66, 76, 509, 524
Gamma, 546, 548, 548-554, 560-562
Garay, J., 152, 524
garbage collection（垃圾收集）, 500, 526, 637
Garland, S., 231, 770
Gawlick, R., 450, 612, 769
general timed automaton（通用计时自动机）, 736, 745, 744-772
GHS, 66, 511, 509-524, 528, 529
Gifford, D., 586
Gligor, V., 636
global computation（全局计算）, 60-61, 78
global information system（全球信息系统）
global process labelling（全局进程标记）, 174
global snapshot（全局快照）, 2, 4, 604-606, 617
  consistent（一致）,见 consistent global snapshot
global synchronizer（全局同步器）, 533-535, 560
GlobSynch, 534, 533-536, 540, 545-547, 552, 560-562
Goldman, K., 326, 449, 586
Goldreich, O., 524
good communication pattern（良好通信模式）, 87, 87, 89, 92, 143, 168
Goodman, N., 13, 192, 586
Gouda, M., 450
granularity（粒度）, 216
Gray, J., 95
grid point（格点）, 166, 168
Gries, D., 231
Griffeth, N., 636
GTA,见 general timed automaton
Guttag, J., 231

H

Haas, L., 636
Hadzilacos, V., 13, 192, 471, 586, 685
Haldar, S., 450
Halpern, J., 152
halting（停止）, 19, 30-31, 47, 181
halting state（停止状态）, 19
handshake（握手）, 428-430, 449
Heitmeyer, C., 770
Helary, J. M., 13
Herlihy, M., 192, 252, 391, 392, 449, 450, 586
hexagon network（蜂窝网络）, 130, 158
Hibbard, T., 326, 327
hiding（隐藏）
  general timed automata（通用定时自动机隐藏）, 756, 770
  I/O automata（I/O自动机隐藏）, 206, 209, 212, 212, 568
  MMT automata（MMT自动机隐藏）, 741, 744, 744
  signatures（签名隐藏）, 212
hierarchical construction of atomic objects（原子对象的分级构造）, 398, 417-418, 449
hierarchical proof（层次式证明）, 224-228
hierarchical resource allocation（分级资源分配）, 349
high-level fairness（高层次公平性）, 276, 276
high-level message（高层次消息）, 691

high-level-fairness（高层次公平性），278, 339, 368
Higham, L., 47, 523
Hirschberg, D., 32, 47，523
history variable（历史记录变量），231
Ho, G., 636
Hoare, C. A. R., 23
hop count（跃迁数），33
HS, 32, 33, 31-35, 37, 38, 47, 48, 482, 485, 486, 523, 525, 553
Humblet, P., 66, 76, 509, 524
hybrid algorithm（混合算法），573
hybrid system（混合系统），750
HybridBFS, 505, 505-506, 527

I

I-failure termination condition（*I*故障终止条件），407
I-simulation（*I*模拟），566, 583
I/O automaton（I/O自动机），201, 199-234, 237, 251, 260, 372, 458, 476, 611, 735, 736
I/O automaton model（I/O自动机模型），7
idle system state（空闲系统状态），300
illogical time（不合理时间），612, 613, 665
imperfect failure detector（不完美故障检测器），688
impossibility result（不可能性结果），27, 43, 45, 71, 84, 93, 130, 133, 136, 141, 145, 147, 148, 177, 190, 302, 303, 314, 320, 342, 383, 386, 388, 390, 420, 486, 487, 557, 580, 586, 672, 682, 684, 705, 713, 717, 718, 788, 789, 801, 810, 819
in, 201, 745
in transit（在传递途中的），713, 718
in-nbrs, 457
incomplete operation（不完整操作），401, 419, 427, 431, 435, 438, 443, 450, 573, 574, 579
IncorrectFischerME, 775, 775-776, 791
independence of activities（活动的独立），2
independent progress condition（独立演进条件），338, 368, 653, 654
independent set（独立集），71, 71
indistinguishable executions（不可区分的运行 / 不可分运行），21, 84, 229, 229, 380, 415, 465, 469, 535, 567, 593, 626, 670, 706
indistinguishable states（不可区分的状态 / 不可分状态），244, 244, 300
infinite duplication（无限复制），702, 708
InfiniteTicketME, 322, 322-521
information flow（信息流），88
information level（信息层），88, 89, 92, 94
information processing（信息处理）
  distributed（分布式），见 distributed information processing
initial, 239
initial index（初始索引），737
initial state（初始状态），18，202，745
initial value（初始值 / 开始值），239
initialization（初始化），376, 392, 686
in-nbrs, 17
input（输入），20, 20
input action（输入动作），200, 201, 201, 256, 745
input transition（输入转移），202
input variable（输入变量），20
input-enabled automaton（输入使能自动机），202, 202-203, 217, 232, 257
input-first execution（输入起始运行），376
input/output automaton（输入/输出自动机），237, 见I/O automaton
input/output automaton model（输入/输出自动机模型），197, 见 I/O automaton model
instantaneous global snapshot（瞬时全局快照），602, 604, 605
int, 201, 745
IntegerBloom, 442, 442-447, 453, 579
interfering variable type（冲突变量类型），394
internal action（内部行为），200, 201, 201, 256, 745
interprocess communication（进程间通信），xx, 1, 5, 6
invariant（不变式），见invariant assertion
invariant assertion（不变式断言），7, 10, 21, 23, 216, 216-217, 219, 230, 256, 478, 521, 617, 633, 735, 757, 756-759, 770
invocation（调用），244
IPC, 见 interprocess communication
Isabelle, 231
Isis, 612
Isloor, S., 636
ISO, 691, 722, 728, 729
isolated request（孤立调用），643, 645, 649, 654
Israeli, A., 392, 450, 587

J

Jackson, P., 326
Janssen, W., 524
join, 488
Jonsson, B., 23, 231

K

k-agreement（*k*—致性），9, 102, 162, 161-177, 192-193,

390, 390-392, 394, 669, 673, 681-682, 684-685, 825
k-concurrent-progress condition（*k*并发演进条件）, 332, 792
k-exclusion（*k*互斥）, 327, 329, 331, 337
k-extension（*k*扩展）, 712
k-message-bounded（*k*消息限制）, 713, 731
k-neighborhood（*k*邻居）, 40, 40
Karp, R., 13
Klawe, M., 47, 523
Kleinberg, J., 824
knowledge（知识）, 88, 358, 487
Knuth, D., 326, 449
Koller, D., 685
König, D., 449
König's Lemma（König引理）, 404, 405, 405, 406, 449
Kruskal, J., 65
Kutten, S., 524

L

λ, 14
l-run（*l*运动）, 169, 193
Ladin, R., 612
Ladner, R., 729
Lamport, L., 13, 23, 99, 152, 153, 230, 231, 326, 449, 524, 597, 612, 636, 664, 770, 790, 824
LamportTime, 597, 597-600, 612, 649, 655
Lampson, B., 729, 770
Larch Prover, 231, 770
last, 751
LayeredBFS, 504, 504-506, 524, 527, 554
LCR, 28, 27-32, 35, 37, 38, 46-48, 54, 476-482, 553
Le Lann, G., 27, 46, 664
leader election（领导者选举）, 4, 8, 11, 15, 25, 25-49, 51-57, 61, 70, 77-79, 342, 475-496, 500-501, 523-527, 531, 553-554, 563, 635
Lehmann, D., 367
LehmannRabin, 355, 354-367, 369, 587, 676
Leighton, F.T., 13
LeLann, G., 523
level（层 / 级）, 96, 510, 522
$level_\gamma$, 88
Lewis, H., 23, 770
Li, M., 392, 450, 587
limit-closure（极限封闭）, 218
line graph（线图）, 526, 529
line of processes（进程线）, 487
linearizable object（可线性化对象）, 397, 449
link（链路）, 18
link failure（链路故障）, 9, 12, 15, 19, 81-97, 182, 183, 461-464, 691-731
Liskov, B., 23, 612
LISP, 45
live, 174
liveness（活性）, 5, 7, 197, 406-407, 459
liveness property（活性属性）, 219, 218-221, 230, 234, 253, 262
local, 201, 745
local area network（局域网）, 1, 8, 25
local process labelling（局部进程标记）,173
local synchronizer（局部同步器）, 535-537, 560, 561, 594
locality restriction（局部性限制）, 239
locally controlled action（局部控制动作）, 201, 745
lockout（锁定）, 263, 276, 369, 791
lockout-freedom condition（锁定权条件）, 277, 319-322, 326, 328, 610, 643, 654
LocSynch, 535, 535-537, 539, 541, 545, 560
logical time（逻辑时间）, 11, 455, 475, 566, 591-615, 646-653, 655-656
logical-time assignment（逻辑时间指派）, 592, 594
LogicalTimeME, 646, 647, 646-650, 664, 665
LogicalTimeSnapshot, 605, 605-606, 612, 613, 627
Loke, W.T., 636
loss（丢失）
  message（消息）, 见 message loss
lossy FIFO send/receive channel（损耗式FIFO发收通道）, 462, 462-463, 699
lossy reordering send/receive channel（损耗式重排序发收通道）, 463, 463-464, 695, 700
Loui, M., 391, 685
low-level fairness（低层次公平性）, 276
low-level message（低层次消息）, 691
lower, 736
Luby, M., 71, 74, 76
LubyMIS, 73, 71-76, 530, 555, 562
Luchangco, V., 770, 790
Lynch, N., 13, 23, 47, 95, 152, 153, 192, 230, 231, 251, 252, 326, 327, 367, 391, 392, 450, 471, 524, 560, 573, 586, 587, 612, 636, 664, 685, 728, 729, 769, 770, 790, 823, 824

M

majority voting（大多数投票）, 116
MajorityVoting, 574, 573-575, 577, 580, 586
MajorityVotingObject, 573, 573-575, 587, 588

Manna, Z., 230, 327
Mansour, Y., 471, 728, 729
Marsland, T., 636
Martin, J., 23
matrix computation（矩阵计算）, 556
Mattern, F., 612
Mavronicolas, M., 560, 824
maximal independent set（最大独立集）, 8, 51, 71, 71-76,80, 530, 555, 562
maximum flow（最大流）, 560
McGuire, M., 326
mean, 179, 682
Menasce, D., 636
Menger, K.,153
Menger's Theorem（Menger定理）, 135, 136, 153
merge（归并、合并）, 172, 193
Merritt, M., 13, 153, 230, 449, 573, 586, 612, 735, 769
message bounded（有界消息）, 713
message duplication（消息复制）, 2, 457, 460, 461, 691-693, 704-706
message loss（消息丢失）, 2, 457, 460, 461, 646, 691-693, 695, 699, 706-715
message passing（消息传递）, 6
message reordering（消息重排序）, 460, 691-693, 695, 703-715
message-generation function（消息生成函数）, 18
milestone（里程碑）, 780
Milner, R., 23
minimum spanning tree（最小生成树）, 8, 11, 51, 63, 63-70, 79, 475, 509, 509-524, 528-529, 531, 563
MIS, 见 maximal independent set
Misra, J., 13, 367, 636, 664
mixed specification strategy（混合规格说明策略）, 459
MMT automaton（MMT自动机）, 736, 735-744, 769-771
model（模型）
  asynchronous（异步）, 见asynchronous model
  asynchronous network（异步网络）, 见 asynchronous network model
  asynchronous shared memory（异步共享存储器）, 见 asynchronous shared memory model
  formal（形式化）, 见 formal model
  partially synchronous（部分同步）, 见 partially synchronous model
  partially synchronous network（部分同步网络）, 见 partially synchronous network model
  partially synchronous shared memory（部分同步共享存储器）, 见partially synchronous shared memory model
  randomized（随机）, 见randomized model
  shared memory（共享存储器）, 见shared memory model
  synchronous（同步）, 见 synchronous model
  synchronous network（同步网络）, 见 synchronous network model
modified *VariableSpeeds*（已修改的*VariableSpeeds*算法）, 37, 37-38
Modugno, F., 735, 769
modular decomposition（模块分解）, 532
ModularDP, 659, 660, 658-664, 667
modularity（模块性）, 6
monotone Boolean formula（单调布尔公式）, 367
Moran, S., 392, 686
Moses, Y., 152, 153
move, 170
msgs, 18, 22, 101
MST, 见minimum spanning tree
multi-copy replicated data scheme（多副本复制数据模式）, 567, 572-575
multicast（多播 / 组播）, 457, 655
multicast channel（多播通道）, 469-471, 641
multicast system（多播系统）, 457, 471, 469-471
multiprocessor（多处理机）, 1, 397, 454
  fault-tolerant（容错）,见 fault-tolerant multiprocessor
  shared memory（共享存储器）, 见 shared memory multiprocessor
Muntz, R., 636
mutual exclusion（互斥）, 4, 9-10, 12, 13, 228, 235, 261, 255-333, 335, 372, 433, 451, 453, 521, 588, 609-610, 614, 643, 641-653, 664-665, 733, 736, 774, 773-793, 823
mutual exclusion condition（互斥条件）, 261, 642, 774
MWOE, 66, 510

N

$\mathcal{N}$, 230
N, 14
N$^+$, 14
n-subset（*n*子集）, 45
Neiger, G., 612
network（网络）
  local area（局域）, 见local area network
Nipkow, T., 231
non-blocking commit algorithm（非阻塞提交算法）, 183, 183, 185-189
non-blocking condition（非阻塞条件）, 183
non-leader（非领导者）, 30, 47, 52, 525

nondeterminism（不确定性），3, 257
nondeterministic version（不确定版本），230
normal form algorithm（范式算法），45
now, 751
nuclear power plant（核电站），153
null, 18
null message（空消息），18, 28, 33, 35-37, 40, 45, 106, 115, 159, 185

O

Obermarck, R., 636
object（对象），2, 4
1-concurrent-progress condition（1并发进程条件），785, 786, 792
1-valence（1价），377, 686
operating system（操作系统），9,255,823, 827
operational proof（操作式证明），269
OptEIGStop, 114, 155
OptFloodMax, 54, 54-57, 76-78, 495, 526, 554, 635
OptFloodSet, 105, 105-108, 115, 129, 152, 154, 155, 224, 322
optimistic algorithm（最优算法），573
optimization（最优化），47, 54, 56, 105, 114, 495, 614, 649, 652, 665, 697
order-equivalent sequences（次序等价序列），39, 40, 42
orientation（定向/定位），25
out, 201, 745
out-nbrs, 17, 457
output（输出），20, 20
output action（输出动作），200, 201, 201, 256, 745
output transition（输出转移），202
output variable（输出变量），20
Owicki, S., 231

P

packet（包），722
Papadimitriou, C., 23
parallel composition（并行合成），7
parallel random access machine（并行随机存取机），2, 3
Park, D., 23, 231
partial order proof（偏序证明），536
partially synchronous algorithm（部分同步算法），12-13, 733, 773-827
partially synchronous model（部分同步模型），5, 7-8, 612, 732-772
partially synchronous network model（部分同步网络模型），768-769
partially synchronous network system（部分同步网络系统），733
partially synchronous shared memory model（部分同步共享存储器模型），768
partially synchronous shared memory system（部分同步共享存储器系统），733
partially synchronous system（部分同步系统），2, 560
pasting（粘贴），211, 214, 232, 743, 756, 771
Paterson, M., 391, 587, 685
path covering（路径覆盖），121
patt, 190
Paxos, 824
Pease, M., 99, 152
Peleg, D., 524, 587, 685, 686
perfect failure detector（完全故障检测器），677, 677, 685, 688, 796
PerfectFDAgreement, 677, 679, 677-681, 688
Peterson, G., 47, 278, 326, 449, 450, 523, 664
Peterson, J., 327
Peterson2P, 279, 280, 278-328
PetersonLeader, 482, 483, 482-486, 523, 525
PetersonNP, 284, 285, 283-289, 291, 292, 326, 328, 329, 587, 664
Pigeonhole Principle（鸽巢原理），45, 321
piggybacking（借道），58, 90, 498, 613, 619
pileup（堆积），466
Pinchasov, M., 450
Pinter, S., 192, 392
Plotkin, S., 450
Pnueli, A., 230, 327, 367, 769
Pogosyants, A., 231
point-to-point（点对点），1, 457, 470
PolyByz, 127, 127-129, 152, 157
polynomial communication（多项式通信），123, 125-129, 152, 153, 157, 158
Ponzio, S., 823
port（端口），237, 239, 372, 399, 566, 671, 773, 795
possibilities mapping（可能性映射），231
PRAM, 见parallel random access machine
precondition-effect code（前提-结果代码），203
prefix-closure（前缀封闭），218
preserving a safety property（保持安全属性），222, 222-224, 231, 234, 261, 374
Prim, R., 65
probabilistic condition（概率条件），22-23, 49, 71, 74-76, 85-87, 92, 94-96,230, 372, 674
probabilistic I/O automaton（概率I/O自动机），229, 251, 357

probabilistic shared memory model（概率共享存储器模型），251-252
probabilistic time bound（概率时间界限），357
probability distribution（概率分布），22, 72, 87, 358, 675
Probe, 706, 708, 709, 706-712, 728, 730
process（进程），17, 18, 202, 256, 457-458, 466-467, 469-470, 476
process automaton（进程自动机），200, 205, 205-206
process control（进程控制）
  real-time（实时），见real-time process control
process failure（进程故障），19, 99-195, 371-454, 566, 575-586, 641, 645, 669-689, 795-827
processor（处理器），17, 716
processor failure（处理器故障），2, 3, 9, 15
product trace property（乘积轨迹属性），218, 221, 472
program counter（程序计数器），267
programming language（编程语言），29
progress condition（演进条件），261, 328, 338, 642, 653, 654, 774, 791, 792
progress function（演进函数），219
projection（投影），211, 214, 232, 743, 755, 771
propagation（传播），438, 453
prune, 93, 93, 97
Przytycka, T., 47, 523
PSynchAgreement, 804, 803-810, 820, 825, 826
PSynchFD, 797, 797-800, 804, 808, 823, 824, 826

Q

QueueME, 311, 311-314, 322, 332, 610
quiescent state（安静状态/静态），19, 202, 618

R

$R^{\geq 0}$, 14
R^{+}, 14
Rabin, M., 367, 685
Race, 741, 740-741, 766-769
Rachman, O., 449
Ramachandran, V., 13
Ramamoorthy, C., 636
Ramsey's Theorem（Ramsey定理），45, 45, 46, 47
rand, 22, 73, 91
RandomAttack, 90, 90-93, 96, 97
randomization（随机化），22-23, 229-230, 251
randomized algorithm（随机算法），xx, 12, 49, 71-76, 85, 88-93, 95, 344, 354-367, 369, 530, 587, 669, 672-677, 685
randomized model（随机模型），22-23, 229-230, 251
range（范围/域），179
Raynal, M., 13, 327, 560, 664
reachable state（可到达状态），206, 737, 746
read quorum（读法定票数），588
read-modify-write atomic object（读-改-写原子对象），420-421, 449
read-modify-write shared memory model（读-改-写共享存储器模型），10, 309-326
read-modify-write shared variable（读-改-写共享变量），247, 247-248, 258, 387-388, 393
read/increment atomic object（读/递增原子对象），407-408
read/update/snap atomic object（读/改/快照原子对象），453
read/update/snap shared memory model（读/改/快照共享存储器模型），453
read/update/snap variable type（读/改/快照变量类型），433
read/write atomic object（读/写原子对象），399, 402-404, 421, 434-450, 573, 575, 580
read/write locking（读/写锁定），573, 575, 587, 588
read/write register（读/写寄存器），245, 245-247, 253, 258, 264, 376-387
read/write shared memory model（读/写共享存储器模型），10, 670, 684
read/write shared variable（读/写共享变量），245, 245-247, 253, 258, 264, 376-387, 571-575
real time（实时），591, 592, 610-611, 614
real-time clock（实时时钟），11
real-time process control（实时进程控制），1, 823, 827
RealTime, 611
recovery（恢复），183, 192, 716
recovery protocol（恢复协议），716
recurrence（递推），288-289, 292-293, 352-353, 800
reduce, 178, 193
refinement mapping（精化映射），231
region designation（区域指定），267
register（寄存器），245, 245-247, 253, 264, 376-387
regular communication pattern（正则通信模式），148
regular execution（正则运行），148
regular run（正则运动），148, 160
Reischuk, R., 152, 587, 685, 686
relaxation（松弛），62
reliable broadcast channel（可靠广播通道），467, 467-469, 586, 670
reliable communication（可靠通信），4, 641, 691-731
reliable FIFO send/receive channel（可靠FIFO发送/接收

通道），459, 459-460, 471, 475, 476, 535, 538, 557, 568, 670
reliable multicast channel（可靠多播通道），470, 470
reliable reordering send/receive channel（可靠重排序发送/接收通道），460, 460-461, 472
remainder region（剩余区），259
remote procedure call（远程过程调用），1
remove, 170
renaming（重命名），686
reordering（重排序）
  message（消息），见message reordering
replicated state machine（复制状态机），606-610, 612, 646
ReplicatedStateMachine, 607, 606-610, 612-614, 646
resource（资源），336
resource allocation（资源分配），xix, 2, 4, 10, 12, 31, 161, 202, 235, 255, 259, 338, 335-369, 455, 635, 641-667, 686, 793, 823
resource graph（资源图），350
response（回应），244
restricting process activity（受限进程活动），263, 339, 643
Ricart, G., 664
RicartAgrawalaME, 649, 649-653, 655, 656, 664, 665
RicartAgrawalaRA, 655-656, 666
RightLeftDP, 346, 344-354, 367, 368, 572, 772
rigor（严格），4
ring, 488
ring network（环网），8, 10, 25-49, 51, 54, 339-349, 354-367, 475-495, 643-646
RMWAgreement, 388, 387-388, 412, 420
RMWfromRW, 420, 451
Roberts, R., 27, 47, 523
Rodeh, M., 47, 523
Roucairol, G., 664
run, 144
runtime monitoring（运行时监控器），12, 455, 475, 617, 618

S

S-expression（*S*表达式），45-46
safe register（安全寄存器），296, 330
safe synchronizer（安全同步器），541-545, 560, 561
SafeSynch, 521, 541, 544, 544-547, 549-551, 561
safety property（安全属性），218, 218-224, 230, 233, 234, 253, 262, 404, 410
Saias, I., 252, 367
Saks, M., 367, 392, 664
Scantlebury, R., 728
Schaffer, R., 450
schedule module（调度模块），230
Schieber, B., 729
Schneider, F., 230, 449, 612
Scholten, C., 524, 636
scientific computing（科学计算），1
scientific programming（科学程序设计），565
search（查找），4, 823
Segala, R., 231, 252, 367, 769
Segall, A., 524, 729
select, 179, 193, 682
semaphore（信号量），367
send/receive channel（发收通道、发送/接收通道），458-464
send/receive diagram（发收图、发送/接收图），593, 594
send/receive network（发收网络、发送/接收网络），641
send/receive system（发收系统、发送/接收系统），464, 457-466, 583-585, 591-594, 597-600, 610, 669-671, 686
sending omission failure（发送遗漏故障），823
seq, 170, 193
serialization point（串行化点），402, 573
session（会话），556
session problem（会话问题），556, 556-559, 563, 824, 826
Shankar, A. U., 770
shared memory（共享存储器），6
shared memory model（共享存储器模型），6
shared memory multiprocessor（共享存储器多处理机），1
shared variable（共享变量），237-253, 256
shared variable type（共享变量类型），244-250
Shattuck, S., 636
Shavit, N., 192, 327, 392, 449, 450, 636, 790
shortest paths（最短路径），8, 11, 51, 61-63, 79, 475, 501, 506-509, 524, 528, 554-555, 634-635, 637
shortest paths tree（最短路径树），61, 62, 62
Shostak, R., 99, 152
shrinking（收缩），402, 788
Shrira, L., 560
SIFT, 152
sig, 201, 217, 745
signature（签名），201, 201
Silberschatz, A., 327
silent process（安静进程），168
silent state（安静状态），489, 526
simple synchronizer（简单同步器），560, 561
SimpleBcastSim, 585, 585

SimpleMST, 522-524, 529, 531, 532
SimpleShVarSim, 567, 568, 567-572, 574, 576, 580, 587, 588, 606, 609
SimpleSRSim, 583, 584, 583-585, 589
SimpleSynch, 521, 537, 537-541, 545, 547, 555, 560, 561
simplex（单工 / 单向）, 166, 166
simulation（模似）, 见 simulation relation
simulation proof（模拟证明）, 21, 23, 55-56, 106-108, 115, 155, 224-228, 231, 234, 256, 322-326, 445-447, 450, 521, 536, 539, 545, 700, 729, 735, 756, 760-768, 770, 780, 791
simulation relation（模拟关系）, 7, 10, 21, 56, 106, 115, 155, 224, 231, 323, 442, 445, 472, 521, 522, 532, 545, 696, 700, 701, 729, 730, 756
Sinclair, J., 32, 47, 523
Singh, A., 367, 450, 664
single-copy replicated data scheme（单副本复制数据模式）, 567-572
single-failure termination condition（单故障终止条件）, 375, 383-387, 391, 395, 684
Skeen, D., 192
SLL, 见strong loss limitation
slotted exclusion（分槽式互斥）, 686
snapshot（快照）, 823
  consistent global（一致全局）, 见 consistent global snapshot
  global（全局）, 见 global snapshot
snapshot atomic object（快照原子对象）, 10, 421-434, 449, 452, 589, 826
snapshot shared memory model（快照共享存储器模型）, 447-450, 452, 454
snapshot variable type（快照变量模型）, 422
SnapshotRegister, 448, 447-450, 454
Snir, M, 47
Søgaard-Andersen, J., 231, 729, 769, 770
sorting（排序）, 79, 529
source node（源节点）, 57, 496, 502, 619
source process（源进程）, 63
Söylemez, E., 770
spanning forest（生成森林）, 63
spanning tree（生成树）, 4, 63, 70, 79, 475, 496-524, 526, 561, 619, 638, 823
  breadth-first（广度优先）, 见 breadth-first spanning tree
  directed（有向）, 见 directed spanning tree
  minimum（最小）, 见 minimum spanning tree
Sperner, E., 192
Sperner coloring（Sperner着色）, 168, 176, 193
Sperner's Lemma（Sperner引理）, 168, 176, 176-177, 192
Spinelli J., 471, 729
Spira, P., 66, 76, 509, 524
splicing execution fragments（连接运行片断）, 303-308, 321, 322
splicing timed execution fragments（连接定时运行片断）, 788-789
Srikanth, T., 152
stable memory（稳定存储器）, 716
stable property（稳定属性）, 12, 617, 633, 633-639, 823
stable vector algorithm（稳定向量算法）, 677, 681, 686, 688
Stark, E., 192, 392
start, 18, 202, 239, 256, 745
start state（开始状态）, 18, 20, 202, 239, 745
starvation（饥饿 / 饿死）, 276
state（状态）, 18, 201, 239, 745
state transition relation（状态转移关系）, 202, 745
state-transition function（状态转移函数）, 18
states, 18, 201, 239, 256, 745
Stenning, 228, 463, 693, 694, 693-697, 699-703, 722, 728-730
Stenning, N., 691, 728
Stenning's protocol（Stenning协议）, 691
step（步）, 202
Stockmeyer, L., 823, 824
Stomp, F., 524
stop, 251, 372, 399, 458, 566
stopping agreement（停止一致性）, 101, 100-116, 142-162, 165, 183, 185, 189, 195
stopping failure（停止故障）, 2, 9, 12, 19, 99-116, 125, 142-160, 162-177, 182-192, 194, 251, 251, 371-395, 397-454, 458, 458, 566, 575-586, 646, 669-689, 716, 769, 795-827
stretching（伸展）, 783, 788
strong connectivity（强连通性）, 51
strong loss limitation condition（强丢失限制条件）, 461, 472, 692, 711, 712, 722
strong termination condition（强终止条件）, 183
Strong, R., 152, 153
strongly connected digraph（强连通有向图）, 526, 528, 592, 630, 665
strongly isolated request, 649, 664（强孤立请求）
STtoLeader, 501, 501, 520, 523
Styer, E., 664
submarine（潜水艇）, 153
successive refinement（逐步求精）, 224

superposition（叠加），636
supervisor（主管），316, 319, 332
swap operation（交换操作），248, 388, 394
symmetry（对称），31, 39-44, 76, 341-344, 658, 666
SynchBFS, 58, 58-62, 76-78, 496, 498, 502, 554, 562
SynchGHS, 66, 66-70, 76, 79, 509-522, 528
synchronization（同步），2, 4
synchronizer（同步器），11, 455, 475, 496, 521, 531-563, 565
  global（全局），见 global synchronizer
  local（局部），见local synchronizer
  safe（安全），见 safe synchronizer
  simple（简单），见 simple synchronizer
synchronous algorithm（同步算法），255, 496, 735
synchronous model（同步模型），5, 7, 15-23, 733, 735, 795, 823, 827
  network（网络），见 synchronous network model
synchronous network algorithm（同步网络算法），8-9, 15, 25-195, 496, 522, 523, 824
synchronous network model（同步网络模型），7-8, 11, 15-23, 51, 239, 322, 371, 390, 457, 475, 495, 506, 509, 531-563, 565, 681, 682
synchronous network system（同步网络系统），17-19
synchronous system（同步系统），2, 258
system（系统）
  asynchronous（异步），见 asynchronous system
  partially synchronous（部分同步），见 partially synchronous system
  synchronous（同步），见 synchronous system
  synchronous network（同步网络），见synchronous network system

T

task（任务），203, 203, 212, 240
task partition（任务划分），202, 203
tasks, 202, 745
Taubenfeld, G., 790
Tay, Y. C., 636
TCP, 722, 729
Tel, G., 770
telecommunications（电信），1
telephone system（电话系统）
Tempero, E., 729
temporal logic（时序逻辑），219, 230, 231, 327, 333
Temporal Logic of Actions, 231（动作时序逻辑）
temporary variable,（临时变量），266, 267
termination（终止），12, 181, 503, 524, 562, 617, 683
termination condition（终止条件），83, 88, 101, 162, 177, 183
termination detection（终止检测），496, 524, 604, 617-625, 634, 634-638
termination protocol（终止协议），188, 194
test-and-set operation（测试和设置操作），248,388, 394
texecs, 737
thread of control（控制线程），203
ThreePhaseCommit, 183, 186, 188, 185-189, 192, 194
TicketME, 228, 313, 313-314, 322-326, 332, 536
time bound condition（时间界限条件），277
time complexity（时间复杂度），21, 21, 228, 250, 256, 466, 469
time of occurrence（发生时间），746
time-passage action（时间通过动作），745, 745, 751
time-passage equivalent（时间通过等价），746, 756, 771
time-passage refinement（时间通过精化），746
timed execution（计时运行），557, 736, 746
  I/O automaton（计时I/O自动机运行），229
timed execution fragment,（计时运行片断） 746
timed signature（计时签名），745
timed simulation relation（计时模拟关系），760, 761, 764, 767, 771, 772, 780, 782
timed trace（计时轨迹），738, 746
timed trace property（计时轨迹属性），756, 760, 759-760, 774
timeout（超时），8, 12, 685
timeout MMT automaton（超时MMT自动机），739-740, 762
TimeSlice, 35, 35-38, 47, 48
timestamp-based algorithm（基于时戳的算法），573
timing anomaly（定时异常），498, 500, 527
timing failure（定时故障），784-790
timing model（实时模型）
timing uncertainty（定时不确定性），741, 768, 769, 774, 791, 795, 803, 810
timing-based algorithm（基于定时的算法），735
tiny simplex（微型单工），168
TLA, 见Temporal Logic of Actions
toggle bit（翻转位），428, 430, 452
token（令牌），8, 25, 32, 35-37, 48, 169, 169, 643-646
token ring（令牌环），8, 25, 47
topological sort（拓扑排序），351
topology（拓扑结构）
  algebraic（代数），见 algebraic topology
Toueg, S., 13, 152, 471, 587, 612, 636, 685, 686
Tournament, 290, 289-294, 326, 329
tournament tree（锦标赛树），289

trace（轨迹），206, 249, 399
trace, 206
trace liveness property（轨迹活性属性），219
trace property（轨迹属性），216, 216-756
trace safety property（轨迹安全属性），218
traces, 206, 217
trade-off（折衷），152, 505, 824
Trans, 413, 411-418, 420-421, 451, 568, 571
trans, 18, 22, 101, 202, 256, 745
transaction（事务），637
  atomic（原子），见 atomic transaction
transaction-consistent snapshot（事务一致快照），638
transformation（变换），31
  algorithm（算法），见 algorithm transformation
  automaton（自动机），见automaton transformation
transient failure（过渡故障/中间故障），2
transition（转移），202
transition function（转移函数）
  state（状态），见 state-transition function
transition relation（转移关系）
  state（状态），见state-transition relation
transitive closure（转移闭包），556, 562
triangulation（三角化），166, 168
triple-modular redundancy（三模块冗余），116
TrivialKAgreement, 681, 681
TrivialME, 310, 310-311
Tromp, J., 450
trying protocol（尝试协议），259
trying region（尝试区），259
ttrace, 738
TurpinCoan, 122, 123-125, 152, 156, 157, 674, 687
Tuttle, M., 23, 192, 230, 231, 735, 769
two-phase locking（两阶段锁定），573, 587, 588, 614
two-task race（双任务轨迹），740-741, 766, 769, 771
TwoPhaseCommit（算法），184, 183-185, 189, 192

U

UID, 见unique identifier
UnboundedSnapshot, 425, 423-428, 440, 449, 452
uncertainty（不确定性），2-3, 5, 8, 9, 11, 12, 255
undersea vehicle（海底运输工具），153
undirected graph（无向图），19
unique identifier（惟一标识符），8, 26, 27, 52, 344, 476
Unity（合一），636
univalence（单价），377, 686
universal reliable broadcast channel（通用可靠广播通道），468, 591, 594
universal reliable FIFO send/receive channel（通用可靠广播通道），459, 471, 472, 478, 486, 557, 591, 592, 692, 738
universal reliable multicast channel（通用可靠多播通道），470
unknown time bound（不可知时间界限），793, 824, 826
unreliable channel（不可靠通道），691-731
upper, 228, 736
user automaton（用户自动机），241, 241, 259, 331, 372, 399, 464, 532, 641, 670, 773, 795
user interface（用户界面 / 用户接口），458

V

Vaandrager, F., 769, 770
Vainish, R., 524
validity condition（有效性条件），81, 83, 85, 86, 88, 92, 96, 97, 100-102, 116, 139, 151, 153, 160, 162, 177, 183, 374, 389, 390, 672, 681, 682, 796
value（值），239
values, 239
Varghese, G., 95
variable start times（变量开始时间），19, 21, 31, 35, 47, 48, 159
variable type（变量类型），244, 244-606
VariableSpeeds, 36, 35-38, 47, 48
vector clock（向量时钟），612, 614
Vidyasankar, K., 450
virtual ring（虚拟环），645, 665
vis, 745
visible action（可见动作），745
Vitanyi, P., 47, 450
VitanyiAwerbuch, 436, 436-440, 448, 450, 453, 575, 577, 579
volatile memory（易失性存储器），691, 716, 716

W

Waarts, O., 152, 252, 450
wait-free termination condition（无等待终止条件），374, 378-383, 392, 397, 407, 672, 796
wait-freedom（无等待），10, 374
wait-freedom condition（无等待条件），300
waiting chain（等待链），344, 344-346, 666
wakeup, 481
wakeup message（唤醒消息），19, 21
Wang, D.W., 471, 728, 729, 824
Warmuth, M., 47
Watro, R., 685, 686
weak Byzantine agreement（弱Byzantine一致），139,

139-142, 153, 159
weak connectivity（弱连通性）, 68
weak logical time（弱逻辑时间）, 614
weak loss limitation condition（弱丢失限制条件）, 462, 463, 464, 692, 702, 707, 710, 711, 730
weak termination condition（弱终止条件）, 183
weather prediction system（天气预报系统）, xix
weight（权 / 权重）, 63, 502
weight, 61, 64, 502
Weihl, W., 13, 192, 230, 392, 573, 586, 612
Welch, J., 450, 524, 598, 612, 664, 729, 823
WelchTime, 599, 598-600, 612, 826
well-formedness（良构性）, 260, 338, 373, 401, 641, 671, 773, 795
well-formedness condition（良构性条件）,261, 338, 374, 401, 533, 642, 653, 671, 774, 795
width（宽度）, 179, 682
Wilkenson, P., 728
Wing, J., 449
WLL, 见weak loss limitation
Wolfstahl, Y., 392, 686
write quorum（写法定票数）, 588
write-once variable（"一次写"变量）, 20
WrongDP, 342, 342-344

Y

Yelick, K., 449

Z

Zaharoglou, F., 392
Zaks, S., 392, 686
Zeno behavior（Zeno行为）, 738
0-valence（0价）, 376, 686
Zuck, L., 367, 471, 728, 729, 824
Zwiers, J., 524

在本书中，作者给出设计、实现和分析分布式算法的蓝图。本书适合学生、程序员、系统分析员和研究人员等不同类型的读者。本书包括这个领域最重要的算法和不可能解，而且都采用简单的自动机理论进行论述。对所有算法的正确性都给予证明，并且根据精确定义的复杂度标准分析算法的复杂度。其中涉及的问题包括资源分配、通信、分布式处理器之间的一致性、数据一致性、死锁检测、领导者进程的选取、全局快照等。

本书的内容按照系统模型组织，首先是根据定时模型，然后在定时模型内再根据进程间的通信机制。不同系统的材料分别独立成章，便于查阅。

本书论述十分严谨，但又很直观，便于读者迅速理解。本书也为读者提供设计新的算法和证明新的不可能解的基本数学工具。而且，它教给读者怎样对分布式系统进行严格的推理——包括形式化建模，为它们所需的行为设计精确的指标，证明它们的正确性，并且用实际的度量标准来评价它们的性能。

作者简介

Nancy A. Lynch 是麻省理工学院电子工程和计算机科学系的教授，领导麻省理工学院的分布式系统理论研究组。在分布式算法和不可能解以及分布式系统的形式化建模和证明方面，她编写了大量的著作。

ELSEVIER

ISBN 7-111-13127-4

9 787111 131274

封面设计 陈子平

HZ BOOKS

华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线：(010)68995259，68995264

读者服务信箱：hzedu@hzbook.com

http://www.hzbook.com

ISBN 7-111-13127-4/TP · 2938

定价：59.00 元