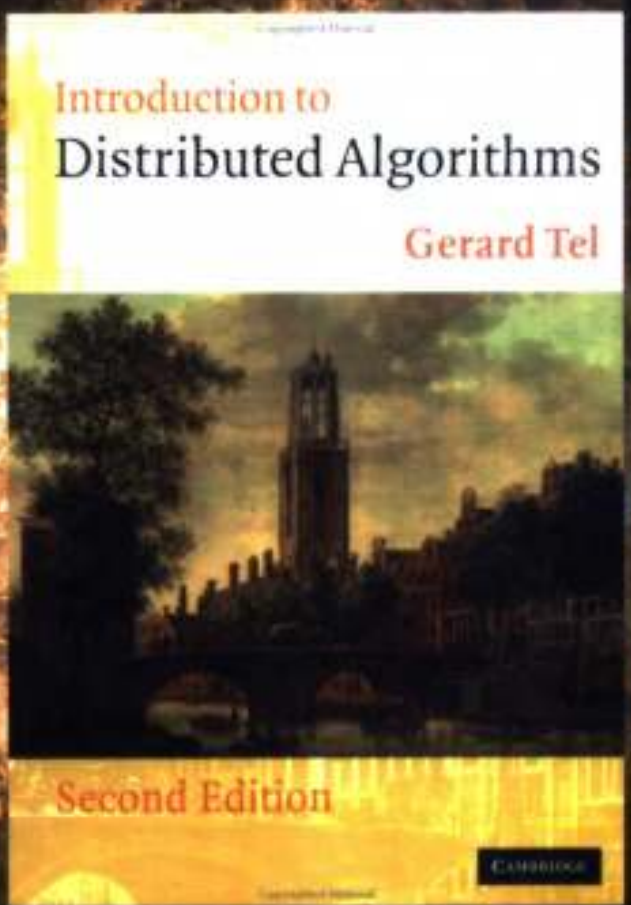
计 算 机 科 学 丛 书

原书第2版

分布式算法导论

（荷） Gerard Tel 著 霍红卫 译



Introduction to Distributed Algorithms
(Second Edition)

机械工业出版社
China Machine Press

在过去20年里，分布式算法一直是备受关注的研究课题，本书是这个领域的经典教科书，曾在国外多所大学被广泛使用。

本书重点讲解点到点消息传递模型上的算法，包括用于计算机通信网络的算法。还分析其它分布式应用的控制算法（波动算法、广播算法、选举算法、终止检测算法、匿名网络的随机算法、快照算法、死锁检测算法和同步系统算法），涉及利用分布式算法实现容错计算。本书反映了当今分布式算法最新技术的发展水平，对于进一步研究这个领域提供了帮助。

作者简介

Gerard Tel 在荷兰Utrecht大学获得博士学位，现任Utrecht大学计算与信息科学学院助理教授。其主要研究方向包括复杂性、压缩、密码学、通信和编码等。出版过多本广受好评的著作。



ISBN 7-111-14674-3

9 787111 146742

HZ BOOKS 华章图书

华章网站 http://www.hzbook.com

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

ISBN 7-111-14674-3/TP・3574

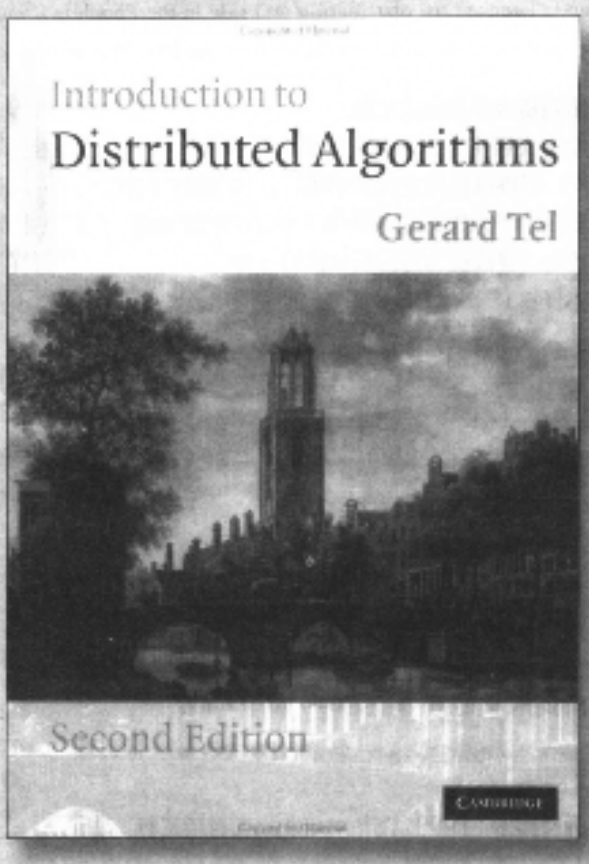
定价：39.00元

计 算 机 科 学 丛 书

原书第2版

分布式算法导论

(荷) Gerard Tel 著 霍红卫 译



Introduction to Distributed Algorithms
(Second Edition)

机械工业出版社
China Machine Press

本书详细介绍了分布式算法及其理论，结合大量定理、引理、命题等的证明，讨论了点到点消息传递模型上的算法，计算机通信网络中实现的算法，重点是分布式应用的控制算法（如波动算法、广播算法、选举算法、同步系统算法等），还涉及了利用分布式算法实现容错计算、方向侦听和故障检测器等方面的内容。本书条理清晰、深入浅出，适合作为大学本科高年级和研究生的分布式算法课程的教材和参考书，对于具有实践经验的专业人员也大有帮助。

Gerard Tel: Introduction to Distributed Algorithms, Second Edition(ISBN 0-521-79483-8).

Originally published by Cambridge University Press in 1994, 2000.

This Chinese edition is published with the permission of the Syndicate of the Press of the University of Cambridge, Cambridge, England.

Copyright © 2004 by Cambridge University Press.

This edition is licensed for distribution and sale in the People's Republic of China only, excluding Hong Kong, Taiwan and Macao and may not be distributed and sold elsewhere.

本书原版由剑桥大学出版社出版。

本书简体字中文版由英国剑桥大学出版社授权机械工业出版社独家出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

此版本仅限在中华人民共和国境内（不包括中国香港、台湾、澳门地区）销售发行，未经授权的本书出口将被视为违反版权法的行为。

版权所有，侵权必究。

本书版权登记号：图字：01-2002-1180

图书在版编目（CIP）数据

分布式算法导论（原书第2版）/（荷）泰尔（Tel, G.）著；霍红卫译. -北京：机械工业出版社，2004.9

（计算机科学丛书）

书名原文：Introduction to Distributed Algorithms, Second Edition

ISBN 7-111-14674-3

Ⅰ.分… Ⅱ.①泰… ②霍… 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（2004）第058300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：王铁元

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2004年9月第1版第1次印刷

787mm×1092mm 1/16·25印张

印数：0 001-3000 册

定价：39.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

前　言

在过去几年里，分布式系统和分布式信息处理得到广泛关注。几乎每一所大学都会开设至少一门关于分布式算法设计的课程。出现了许多关于分布式系统原理方面的书籍，例如Tanenbaum [Tan96] 和Sloman & Kramer [SK87]，但这些书籍主要是针对结构而不是算法。自从本书第1版问世以来，相继出版的分布式算法方面的著作有Barbosa [Bar96]、Lynch [Lyn96] 和Attiya & Welch [AW98]。

因为算法是计算机应用的基础，因此需要一本专门介绍分布式算法的书。本书的目的是展示过去20年来分布式算法方面的诸多理论。本书可作为分布式算法1~2学期的教学用书，一学期的课程可由教师从本书中选择若干专题来安排。

本书也可作为相关专业的工程师和从事分布式系统研究的科研人员的参考书。

练习。每章（除第1章和第13章）后面附有一些习题和项目。项目通常要求读者开发涉及该章内容的一些应用。大多数情况下，没有提供“答案”。

致谢。本书经以下人士的仔细校对。他们是：Erwin Bakker、Hans Bodlaender、Stefan Dobrev、Petra van Haaften、Ted Herman、Jan van Leeuwen、Patrick Lentfert、Friedemann Mattern、Pascale van der Put、Peter Ružička、Martin Rudalics、Anneke Schoone和Kaisa Sere。他们对手稿质量的改进提出了有益的意见。此外，在Utrecht大学选修秋季“分布式算法”课程的学生们也提供了有益的建议。计算机科学系为所需的文本处理和输出提供了技术支持。Susan Parkinson进行了文字编辑。

Gerard Tel

1994年4月/2000年2月

译 者 序

本书是关于分布式算法的最优秀的著作之一。它系统地阐述了分布式算法设计的理论、方法和应用实例。目前，国内尚缺少专门介绍分布式算法的著作。我们希望本书能对我国高等院校的计算机教育有所帮助。

在过去的二十年里，分布式算法一直是备受关注的研究课题。这本成功的教科书的第二版，介绍了分布式算法研究领域的最新进展，新增了两章关于方向侦听和故障检测器的内容，代表了当今该领域最新技术发展水平。

本书分四部分：协议（第2章－第5章）、基本算法（第6章－第12章）、容错（第13章－第17章）和附录（附录A、附录B）。书中内容全面阐述了过去20年来分布式算法方面的诸多理论。本书主要内容及特点如下：

- 第一部分介绍了分布式系统和通信网络的基本概念，讨论了平衡滑动窗口协议和基于计时器的协议，以严谨简明的形式对路由算法作了系统论述，最后讨论了缓冲区有限时无死锁的包交换问题。
- 第二部分讨论了基本算法，包括：波动算法、遍历算法、广播算法、选举算法、终止检测算法、匿名网络的随机算法、快照算法、方向侦听与定向算法、死锁检测算法和同步系统算法。
- 第三部分讨论了容错问题，引入了健壮算法和稳定算法的概念，证明了同步系统的健壮性要比异步系统更大，最后讨论了故障检测和稳定算法。
- 第四部分介绍了伪代码使用约定，图和网络中的一些基本概念和常用术语。

所有算法既给出严格的数学定义及类Pascal语言的形式描述，又以算法不变式作为手段给出算法正确性的形式证明，充分反映了作者在分布式算法方面的造诣。

本书适合作为高等院校分布式算法、分布式计算课程的本科生和研究生教材，同时可作为从事分布式系统设计与应用的专业人员的参考书。

由于时间较紧及译者水平有限，译文难免有错误及不妥之处，恳请读者批评指正。

霍红卫
西安电子科技大学计算机学院
2003年12月

译者简介



霍红卫，1963年8月出生，博士。现为西安电子科技大学计算机学院教授。主要研究方向：算法分析与设计、并行与分布式计算、遗传算法、生物信息学中的优化算法。著作有：《算法设计与分析》、《并行分类算法》和《Exercises & Solutions on Algorithms》。

作 者 序

With great pleasure I welcome this publication of the Chinese translation of my book——《Introduction to Distributed Algorithms》.

In recent years, the flourishing economy of China greatly promotes the development of science and technology. Cooperation between China and other countries in IT industry is increasingly strengthened. More and more Chinese researchers, along with their counterparts in other countries, participate in various distributed computing projcets both at home and abroad.

All these distributed computing projects need algorithms for cooperation, coordination, information exchange, overcoming failures, etc.

The topics discussed in this text will promote a fundamental style of thinking about algorithms, mathematical proofs, specifications and models They may not only be helpful in studying existing methods, but also lay a intellectual foundation for studying new problems. I hope that many Chinese readers will find the book useful.

Acknowledgements:

Thanks are due to the China Machine Press for planning and printing this edition of my book in the Chinese language.

Special thanks go to Prof. Hongwei Huo of Xidian University for she has undertaken the enormous work of translating this book into the Chinese language.

Gerard Tel, June 2004

欣闻我的著作——《分布式算法导论》中文版即将出版，非常高兴。

最近几年，繁荣的中国经济极大地促进了科学技术的发展。IT界的中外合作日益加强。越来越多的中国研究人员与国外同行一起参加到国内外各种分布式计算工程中。所有这些分布式计算工程都需要协调合作、信息交换和故障排除的算法。

本书讨论的课题将促进对算法、数学证明、说明以及模型的根本思考。它们不仅对研究已有的方法大有帮助，而且会为研究新的问题奠定知识基础。我希望广大中国读者会从本书中受益。

致谢:

非常感谢机械工业出版社策划和出版了本书的中文版。

还要特别感谢西安电子科技大学计算机学院的霍红卫教授承担了本书的翻译工作。

Gerard Tel

2004年6月

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
周立柱	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		

秘　书　组

武卫东　　温莉芳　　刘　江　　杨海玲

出版者的话

文艺复兴以来，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

目　录

第1章　导论: 分布式系统 ……1
1.1　分布式系统的定义 ……1
1.1.1　动机 ……1
1.1.2　计算机网络 ……3
1.1.3　广域网络 ……3
1.1.4　局域网 ……5
1.1.5　多处理器计算机 ……6
1.1.6　协同操作进程 ……8
1.2　体系结构和语言 ……10
1.2.1　结构 ……10
1.2.2　OSI参考模型 ……11
1.2.3　局域网络OSI模型: IEEE标准 ……13
1.2.4　语言支持 ……14
1.3　分布式算法 ……15
1.3.1　分布式算法与集中式算法 ……15
1.3.2　一个例子: 单消息通信 ……16
1.3.3　研究领域 ……20
1.4　本书概要 ……21

第一部分　协　议

第2章　模型 ……25
2.1　转移系统和算法 ……25
2.1.1　转移系统 ……26
2.1.2　异步消息传递系统 ……26
2.1.3　同步消息传递系统 ……27
2.1.4　公平性 ……28
2.2　转移系统性质的证明 ……29
2.2.1　安全性 ……29
2.2.2　活动性 ……30
2.3　事件的因果序和逻辑时钟 ……31
2.3.1　事件的独立性和相关性 ……32
2.3.2　执行的等价性: 计算 ……33
2.3.3　逻辑时钟 ……35
2.4　附加假设，复杂度 ……37
2.4.1　网络拓扑结构 ……37
2.4.2　信道性质 ……38
2.4.3　实时性假设 ……39
2.4.4　进程知识 ……40
2.4.5　分布式算法的复杂度 ……40
习题 ……41
第3章　通信协议 ……43
3.1　平衡滑动窗口协议 ……44
3.1.1　协议表示 ……44
3.1.2　协议的正确性证明 ……46
3.1.3　协议讨论 ……47
3.2　基于计时器的协议 ……49
3.2.1　协议表示 ……51
3.2.2　协议的正确性证明 ……54
3.2.3　协议讨论 ……57
习题 ……60
第4章　路由算法 ……61
4.1　基于目的节点的路由 ……62
4.2　所有点对之间的最短路径问题 ……65
4.2.1　Floyd-Warshall算法 ……65
4.2.2　Toueg最短路径算法 ……67
4.2.3　讨论以及更多算法 ……70
4.3　变更算法 ……73
4.3.1　算法描述 ……74
4.3.2　变更算法的正确性 ……78
4.3.3　算法讨论 ……79
4.4　带有压缩路由表的路由 ……80
4.4.1　树标号模式 ……80
4.4.2　区间路由 ……82
4.4.3　前缀路由 ……88

4.5 分级路由 ……90
习题 ……92
第5章 无死锁的包交换 ……93
5.1 引言 ……93
5.2 有结构的方法 ……94
5.2.1 缓冲图 ……95
5.2.2 图G的定向 ……97
5.3 无结构的方法 ……100
5.3.1 前向计数控制器和后向计数控制器 ……100
5.3.2 前向状态控制器和后向状态控制器 ……101
5.4 需进一步研究的问题 ……102
5.4.1 拓扑变化 ……102
5.4.2 其他类型的死锁 ……103
5.4.3 活锁 ……104
习题 ……105

第二部分 基本算法

第6章 波动算法与遍历算法 ……107
6.1 波动算法的定义和使用 ……107
6.1.1 波动算法定义 ……107
6.1.2 波动算法的一些基本结果 ……109
6.1.3 具有反馈的信息传播 ……110
6.1.4 同步 ……111
6.1.5 计算下确界函数 ……111
6.2 波动算法集 ……112
6.2.1 环网算法 ……112
6.2.2 树算法 ……113
6.2.3 回波算法 ……115
6.2.4 轮询算法 ……116
6.2.5 相位算法 ……117
6.2.6 Finn算法 ……118
6.3 遍历算法 ……120
6.3.1 遍历团 ……121
6.3.2 遍历圆环 ……121
6.3.3 遍历超立方体 ……122
6.3.4 遍历连通网络 ……123
6.4 深度优先搜索的时间复杂度 ……124
6.4.1 分布式深度优先搜索 ……125
6.4.2 线性时间的深度优先搜索算法 ……126
6.4.3 具有近邻知识的深度优先搜索 ……130
6.5 遗留问题 ……130
6.5.1 波动算法综述 ……130
6.5.2 计算和 ……130
6.5.3 时间复杂度的另一种定义 ……132
习题 ……134
第7章 选举算法 ……137
7.1 引言 ……137
7.1.1 本章所做假设 ……138
7.1.2 选举和波动 ……138
7.2 环网 ……140
7.2.1 LeLann和Chang-Roberts算法 ……140
7.2.2 Peterson/Dolev-Klawe-Rodeh算法 ……144
7.2.3 一个下界 ……146
7.3 任意网 ……148
7.3.1 废止和快速算法 ……149
7.3.2 Gallager-Humblet-Spira算法 ……151
7.3.3 GHS算法的全局描述 ……152
7.3.4 GHS算法的详细描述 ……153
7.3.5 GHS算法的讨论和变化 ……157
7.4 Korach-Kutten-Moran算法 ……158
7.4.1 模块构造 ……158
7.4.2 KKM算法的应用 ……161
习题 ……162
第8章 终止检测 ……165
8.1 预备知识 ……165
8.1.1 定义 ……165
8.1.2 两个下界 ……167
8.1.3 终止进程 ……169
8.2 计算树和森林 ……169
8.2.1 Dijkstra-Scholten算法 ……169
8.2.2 Shavit-Francez算法 ……172
8.3 基于波动的方法 ……175
8.3.1 Dijkstra-Feijen-Van Gasteren算法 ……175
8.3.2 基本消息的计数: Safra算法 ……178
8.3.3 利用确认 ……181
8.3.4 带波动的终止检测 ……183
8.4 其他方法 ……184
8.4.1 信用-恢复算法 ……184

8.4.2 基于时戳的终止检测方法 ……186
习题……188
第9章 匿名网络……191
9.1 预备知识……192
9.1.1 定义 ……192
9.1.2 概率算法的分类 ……194
9.1.3 本章考虑的问题 ……195
9.1.4 同步消息传递与异步消息传递 ……195
9.2 确定算法……196
9.2.1 确定性的选举：否定性的结果 ……196
9.2.2 环上函数计算 ……197
9.3 概率选举算法……200
9.4 网络规模计算……202
9.4.1 否定性结果 ……203
9.4.2 计算环规模的算法 ……204
习题……206
第10章 快照 ……209
10.1 预备知识 ……209
10.2 两个快照算法 ……212
10.2.1 Chandy-Lamport算法……212
10.2.2 Lai-Yang算法 ……213
10.3 使用快照算法 ……214
10.3.1 计算信道状态……214
10.3.2 快照的适时性……215
10.3.3 稳定性检测……216
10.4 应用：死锁检测 ……217
10.4.1 基本计算模型和问题阐述……217
10.4.2 全局-标记算法 ……219
10.4.3 受限模型的死锁检测……220
习题……221
第11章 方向侦听与定向 ……223
11.1 引言和定义 ……223
11.1.1 方向侦听的定义和特性……223
11.1.2 利用方向侦听……225
11.1.3 具有方向侦听的广播……226
11.2 环和弦环的选举算法 ……228
11.2.1 Franklin算法 ……228
11.2.2 Attiya改进 ……229
11.2.3 最小化弦数……230
11.2.4 1-弦线性算法 ……232
11.3 超立方体上的计算 ……234
11.3.1 基线：没有拓扑知识……235
11.3.2 进行比赛的算法……235
11.3.3 多路径流量算法……237
11.3.4 使用掩码的有效超立方体算法……240
11.3.5 无标号超立方体上的选举算法……241
11.4 与复杂度有关的问题 ……242
11.4.1 团或任意图的定向……242
11.4.2 位复杂度和多路径流量算法……243
11.4.3 Verweij随机漫步算法 ……244
11.5 结论和未解决的问题 ……246
11.5.1 利用方向侦听……246
11.5.2 复杂度归约……246
11.5.3 当前研究……247
习题……247
第12章 网络中的同步 ……249
12.1 预备知识 ……249
12.1.1 同步网络……249
12.1.2 通过同步提高效率……250
12.1.3 异步有限延迟网络……251
12.2 同步网络中的选举 ……254
12.2.1 网络规模已知……254
12.2.2 网络规模未知……255
12.2.3 补充结果……256
12.3 同步器算法 ……256
12.3.1 简单同步器……256
12.3.2 α、β和γ同步器 ……258
12.4 应用：广度优先搜索 ……260
12.4.1 同步BFS算法 ……261
12.4.2 与同步器组合……261
12.4.3 异步BFS算法 ……261
12.5 Archimedean假设 ……264
习题……265

第三部分 容 错

第13章 分布式系统中的容错……267
13.1 利用容错算法的原因 ……267
13.2 健壮算法 ……268
13.2.1 故障模型……268
13.2.2 判定问题……269

13.2.3 第14章到第16章综述……270
13.2.4 本书中没有涉及的主题……271
13.3 稳定算法……271
第14章 异步系统中的容错……273
14.1 一致性的不可能性……273
14.1.1 表示、定义及基本结果……273
14.1.2 不可能性证明……274
14.1.3 讨论……275
14.2 初始死进程……276
14.3 确定可实现实例……277
14.3.1 可解问题：重命名……278
14.3.2 扩展的不可能性结果……280
14.4 概率一致性算法……282
14.4.1 损毁-健壮一致协议……282
14.4.2 Byzantine-健壮一致性协议……285
14.5 弱终止性……288
习题……290
第15章 同步系统中的容错……293
15.1 同步判定协议……293
15.1.1 弹性界限……294
15.1.2 Byzantine广播算法……295
15.1.3 多项式级的广播算法……297
15.2 鉴别协议……300
15.2.1 高度弹性的协议……301
15.2.2 数字签名的实现……303
15.2.3 ElGamal签名模式……303
15.2.4 RSA签名模式……304
15.2.5 Fiat-Shamir签名模式……305
15.2.6 概述和讨论……306
15.3 时钟同步……308
15.3.1 读取远程时钟……308
15.3.2 分布式时钟同步……310
15.3.3 轮模型的实现……313
习题……314
第16章 故障检测……315
16.1 模型和定义……315
16.1.1 四种基本检测器类型……316
16.1.2 故障检测器的用途和缺陷……317
16.2 用弱精确检测器解一致性问题……318
16.3 最终弱精确检测器……319
16.3.1 弹性上界……319
16.3.2 一致算法……320
16.4 故障检测器的实现……321
16.4.1 同步系统：完美检测……321
16.4.2 部分同步系统：最终完美检测……321
16.4.3 小结……322
习题……323
第17章 稳定性……325
17.1 引言……325
17.1.1 定义……325
17.1.2 稳定系统中的通信……326
17.1.3 例子：Dijkstra令牌环……327
17.2 图论算法……329
17.2.1 环定向……329
17.2.2 最大匹配……331
17.2.3 选举和生成树构造……332
17.3 稳定方法学……334
17.3.1 协议组合……334
17.3.2 计算最小路径……338
17.3.3 结论和讨论……342
习题……342

第四部分 附 录

附录A 伪代码使用约定……345
附录B 图和网络……349
参考文献……359
主题词索引……375

第1章 导论: 分布式系统

本章简要介绍了研制分布式算法所依据的硬件和软件系统。阐述了研究分布式算法的原因。通过一个分布式系统表明若干台计算机或处理器相互协作完成计算机应用。分布式系统的定义不仅包括广域计算机通信网，还有局域网、多处理器计算机（每一台处理器都有自己的控制单元）及协同处理系统。

1.1节介绍了各种类型的分布式系统，讨论了使用分布式系统的原因，给出了现有分布式系统的例子。然而，本书的主题既不是这些系统的构成，也不是如何使用它们，而是它们的工作原理。同时，对分布式系统中所使用的算法做专门研究。

当然，仅研究分布式系统的算法是不能完全理解它的整体结构和操作的。为了充分理解这一系统，还必须研究硬件及软件的整个体系结构，即把功能划分成模块。同时还需要研究与程序设计语言性质有关的重要问题，而程序设计语言可用于构建分布式系统中的软件，1.2节将会讨论这些主题。

现有的一些关于分布式系统方面的优秀著作，比如，Tanenbaum [Tan96] 、Sloman and Kramer [SK87] 、Bal [Bal90] 、Coulouris and Dollimore [CD88] or Goscinski [Gos91] 侧重于系统的体系结构和语言方面。正如已经提到的那样，本书主要讨论分布式系统的算法。1.3节解释了分布式算法设计与集中式算法设计的差异。概略地描述了分布式算法的研究领域，并概括了本书的其余部分。

1.1 分布式系统的定义

本章中的“分布式系统”表示自主计算机、进程或者处理器互连的集合。计算机、进程或者处理器称为分布式系统中的节点（node）（在后续章节中，将使用更具技术性的表示方法，参见定义2.6）。“自主性”是指节点必须至少配备自己专用的控制单元。因此，单指令多数据（SIMD）模型的并行计算机并不是分布式系统。“互连性”是指节点之间必须能够交换信息。

由于（软件）进程能够起着系统中节点的作用，因此这个定义包括了作为通信进程集合而构造的软件系统，即使是运行在单一硬件装置上。然而，在大多数情况下，一个分布式系统至少包含几个处理器，这些处理器通过通信硬件互连。

在文献中可以看到对分布式系统更严格的定义。例如，Tanenbaum[Tan96]认为，仅当系统中自主性节点的存在对用户是透明的时候，一个系统才称为分布式系统。从这种意义上讲，分布式系统就像是一个虚拟的、单一的计算机系统，但要实现这种透明性，需要开发复杂的分布式控制算法。

1.1.1 动机

分布式计算机系统比顺序系统更可取，或者说，它们的使用是必然的。我们将对其原因进

行分析，下面的分析只是部分原因。分布式系统的选择受到下列诸多因素的影响，有时可能源于其他原因，但其优势随之显现。分布式系统的特征可能会随着其存在的形式不同而不同。

1.1.2节1.1.6节将会更详细地讨论这些问题。

（1）信息交换　在60年代，当多数大学和公司开始拥有自己的大型机时，产生了在不同
2 计算机间进行数据交换的需求。不同机构的人员通过这些机构的计算机交换数据，使合作变得更为便利。从而引发了对所谓的广域网（wide-area network，WAN）的开发。当今因特网的前身ARPANET于1969年12月投入使用。广域网（有时称远程网络，long-haul network）中所连接的计算机通常都配有用户所需的各种设备，如备份存储器、磁盘、各种应用程序及打印机等。

后来，计算机体积变得越来越小，价格越来越便宜，很快，每一个机构都有了许多计算机，现在，常常是人手一台计算机（个人计算机或工作站）。因此，机构人员间的信息（电子）交换要求自主性计算机互连。甚至有的个人或家庭将多台计算机连接成一小型个人家庭网络也是很常见的。

（2）资源共享　尽管由于计算机的价格便宜，机构可为每一位员工提供一台个人计算机，但对于外围设备（如打印机、存储设备和磁盘等）情况却并非如此。在较小规模内，每台计算机可以依靠专用服务器提供编译器和其他应用程序。此外，在所有计算机上进行应用程序和相关文件资源的复制，则效率低下；除了浪费磁盘空间，还会引起不必要的维护问题。因此，用户计算机要依靠专用节点进行打印输出和磁盘服务。在一个组织内部连接而成的计算机网络称为局域网（local-area network，LAN）。

对于机构而言，建立小型计算机组成的网络，而不是购买大型机，不仅是为了降低系统成本，而且是使系统具有可扩充性。首先，较之大型计算机，小型计算机系统有更好的性能价格比；虽然典型的大型机要比个人计算机快50倍，但其价格却是个人计算机的500倍之多。其次，如果一个小型计算机系统的容量不足，可根据机构需要在网络中增加机器（文件服务器、打印机和工作站）。而如果单机系统的容量不够，则只能更换。

（3）通过复制提高可靠性　分布式系统较之单机系统更具可靠性。这是因为它们有局部
3 故障（partial-failure）的性质。这表明，如果系统中的某些节点发生故障，其他节点仍可正常运行并且能够接管故障的部分。而单机系统的故障则会影响整个系统，在这种情况下，系统不可能继续运行。因此，在设计高可靠性计算机系统时，分布式体系结构常常更受到关注。

高可靠性系统一般有多个处理器，是运行一个应用程序的单处理器的二至四倍用程序并利用表决机制决定机器的输出。因此，当系统的某个部件发生故障时，要使分布式系统正常运行，需要相当复杂的算法支持。

（4）通过并行化提高性能　由于分布式系统中存在多处理器，人们可以把计算密集的作业进行划分，并分布到若干台处理器上分别处理，来减少作业的处理时间。

并行计算机的工作原理就是如此，但通过将任务分配到其他工作站上进行并行处理，局域网的用户也可受益。

（5）通过规范简化设计　计算机系统设计相当复杂，尤其是有相当多的功能要求时。将系统分成模块，可简化系统设计，每一模块实现部分功能，并可与其他模块通信。

通过定义抽象数据类型和不同任务过程，可以得到独立的程序模块。也可将一个大的系统定义为协同进程的集合。在这两种情况下，模块都可以在单一计算机上执行。但在一个局域网中可能会有不同类型的计算机，如，有一台配备专门进行密集数值计算（number crunching）的硬件的计算机，有一台带有制图硬件的计算机，还有一台配备了磁盘等设备。

1.1.2 计算机网络

计算机网络是由通信设备把多个计算机互连而成的计算机集合，通过这些设备实现信息交换。交换是通过接收和发送信息实现的。计算机网络与分布式系统的定义不谋而合。根据计算机之间的距离，计算机网络可分为*广域网*和*局域网*。

4

广域网通常连接不同机构（各行业、大学之间等）的计算机。节点间的物理距离不少于10公里。网络中的每一节点是一台完整的计算机，包括所有的外围设备和相当数量的应用程序。广域网的主要目标是实现不同节点间用户的信息交换。

局域网通常是连接某一机构内部的计算机。节点间的物理距离不大于10公里。网络中的节点可以是工作站、文件服务器或者打印服务器，即机构内部的用于特定功能的相对较小的专用工作站。局域网的主要目标是实现信息交换和资源共享。

这两种类型的网络的界限并不总能清晰界定。从算法观点来看，这种区分不是很重要。因为所有计算机网络中的算法类似。以下列出与算法开发有关的因素。

（1）*可靠性参数* 在广域网中，不能忽视消息传递过程中发生错误的可能性。通常，广域网的分布式算法就设计成要处理这种可能性。局域网就可靠得多。在算法设计时，通常假设通信是完全可靠的。然而，在这种情况下，出错的不可能事件可能会难以察觉地发生，并导致系统操作错误。

（2）*通信时间* 广域网中的消息传输时间要比局域网中的消息传输时间大若干数量级。在广域网中，较之消息的传输时间，消息的处理时间几乎可以忽略。

（3）*同一性* 即使在局域网中，所有的节点也未必相同。通常在一个机构内部，可能会赞同采用共同的软件和协议。在广域网中，有多种协议，这就使得在不同协议间转换和设计出能兼容不同标准的软件变得相当困难。

（4）*互信* 在同一机构内部，所有用户都互相信任。但在广域网中，决不是这种情况。广域网需要开发安全算法，防范其他节点非法用户的入侵。

5

1.1.3节和1.1.4节分别简要讨论了广域网络和局域网络。

1.1.3 广域网络

1. 发展历史

在开发广域网络的过程中，大量早期开创性的工作是在美国国防部的高级研究计划局（Advanced Research Projects Agency，ARPA）的一个项目中完成的。1969年，ARPANET投入运行。当时连接了4个节点。现在这个网络已增长到数百个节点。利用类似的一些技术（MILNET、CYPRESS等），还建立了其他一些网络。ARPANET包括一些特殊节点（称为接口信息处理器，IMP），其惟一目的是处理大量信息。

当UNIX ⊖ 系统广泛流行时，人们意识到需要在不同UNIX机器间进行信息交换。为了达

⊖ UNIX是AT&T贝尔实验室的注册商标。

到这一目的，编写了uucp（UNIX-to-UNIX CoPy）程序。使用此程序，通过电话线就可以进行文件交换，称为UUCP网络的UNIX用户网络迅速出现。由于ARPANET属于国防部，仅有某些机构可以与之相连，因此80年代，开发出另一个主要的网络，称为BITNET。

如今，所有这些网络都是互连的；有一些节点连接了两种网络（称为网关），允许在不同网络的节点之间进行信息交换。统一地址空间和通用协议的引入把网络变成一个单一的虚拟网络，通常称为因特网。不同于“单一”网络，因特网有许多用户，并且没有一个权威机构。但其组织上的多样性对于用户是仔细隐藏的。作者的邮件地址（gerard@cs.uu.nl）并没有提供其部门所连接的网络的信息。

2. 组织和算法上的问题

广域网总是被组成点对点网络。这表明，只有通过一种专门用于这两个节点的连接机制，
6 才会发生一对节点间的通信。这种装置可以是电话线、光纤或者卫星网等。点对点的互连结构可以用图直观地表示出来，用圆圈或方框表示网络中的节点，节点之间的边表示两个节点间的通信线路，如图1-1所示。用更技术化的语言，用图表示结构，其中边代表网络的通信线路。关于图论中的一些术语在附录B中给出。

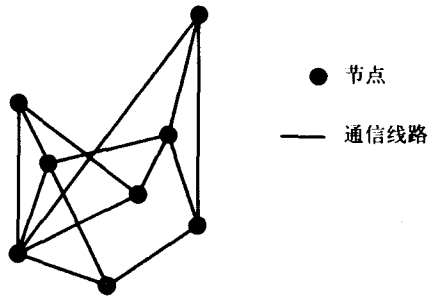


图1-1　点对点网络的一个例子

广域网的主要目的是进行信息交换，如电子邮件、电子公告和远程文件与数据库访问。通过一个应用程序：网页浏览器，可以使用大多数服务。要完成一个能实现这些目标的适合的通信系统，需要解决下列算法问题，其中一些问题在本书第一部分讨论。

（1）点对点数据交换的可靠性（第3章）　通过连接两节点间的线路进行数据交换，必须处理可能会发生的线路不可靠问题。由于大气噪声、动力受损以及其他的物质环境影响，通
7 过线路所发送的消息可能只接收到一部分，甚至丢失。因此必须辨别这些传输故障并纠正。

这个问题不仅发生在用通信线路直接相连的两点间，而且发生在借助中间节点通信的未直接相连的节点之间。在这种情况下，问题更复杂，因为消息到达的次序可能与发送次序不同，可能在很长时间以后到达或者被复制。

（2）通信路径的选择（第4章）　在点对点的网络中，由于费用昂贵，不可能在每一对节点之间都提供通信线路。因此，某些节点必须依靠其他节点进行通信。路由问题所关注的是如何在要通信的节点之间选择一条（或多条）路径。所用的路径选举算法与节点命名模式有关，例如，某节点用于向另一个节点发送消息的地址格式。中间节点的路径选择利用这一地址来进行，而如果能将拓扑信息“编码”在地址中，就能选择更有效的路径。

（3）拥塞控制　如果同时传送大量信息，通信网络的吞吐量就会急剧下降。必须控制各节点所产生的信息，并使其适合网络的可用容量。文献［Tan96，5.3节］讨论了几种避免拥

塞的方法。

（4）*死锁预防*（第5章） 点对点的网络有时称为*存储转发*（Store-and-forward）网络。因为信息发送所经过的中间节点必须将收到的信息存储起来，然后再转发到下一个节点。由于中间节点可用于这一目的的存储空间是有限的，因此必须仔细管理存储空间，预防死锁的发生。在死锁发生时，有许多消息不能被转发，因为下一节点的存储空间完全被其他消息所占据。

（5）*安全性* 网络中的计算机用户各种各样，其中有些人会试图滥用甚至破坏其他人的设备。因为，从世界的任何地方都可登录到网络中的某台计算机上，所以需要身份认证、密码方法以及扫描输入信息（例如，扫描病毒）的安全技术。密码方法（参见［Tan96，7.1节］）可用于数据加密以防未经授权的读操作，也可实现数字签名防止未经授权的写操作。

1.1.4 局域网

局域网用于连接机构内部的计算机。通常，这种连接的主要目的是实现资源共享（文件 8
以及外围设备），使得员工之间的信息交换变得容易。有时，网络也用于加速计算（将任务划分至其他节点）。某些节点可用作备用节点，以便故障发生时使用。

1. 例子和机构

20世纪70年代上半叶，Xerox公司开发出了以太网[⊖]（Ethernet）局域网。广域网的名字，诸如ARPANET、BITNET等都是指特定的网络，而局域网的名字通常是指产品名。例如，只有一个ARPANET、一个BITNET、一个UUCP网络和一个Internet，但是每个公司可能都会建立自己的专用以太网、*令牌环网*（Token Ring）或者*系统网络体系结构*（system network architecture，SNA）。

与广域网络不同，以太网采用*类似总线*（bus-like）的结构，例如，节点间通过连接节点的机制进行通信；参见图1-2。尽管连接机制和如何使用有所不同，类似总线的结构在局域网上非常普遍。

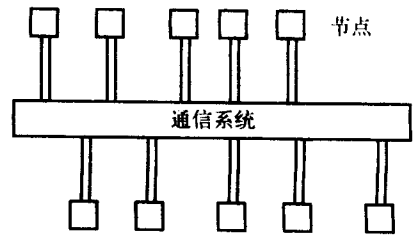


图1-2 总线连接的网络

以太网的设计一次仅允许传输一条消息；其他设计，例如，令牌网（token ring，由IBM Zürich 实验室开发）允许空间*重用*（spatial reuse），即通过通信机制可以同时传输数条消息。总线结构要求少量硬件，因此其价格低廉，但是这种结构也具有*可扩充性*（scalable）差的缺点。这表明一条总线所连接的节点的最大数量相当有限。拥有大量计算机的大公司必须用几条总线将它们连接起来，总线之间用*网桥*（bridge）相连。这就引出了一个分层次的整体网络结构问题。 9

并不是所有的局域网都采用总线结构。IBM设计了一个点对点的网络产品，称为SNA。SNA可将不同的IBM产品相连。SNA设计复杂，因为要与IBM许多现有网络产品兼容。

2. 算法上的问题

对于上面提到的广域网中的一些问题，局域网的实现需要其中一些问题的解决方案。由于总线通常可靠且速度快，因此，可靠的数据交换不成问题。在类似总线结构的网络中，也

⊖ Ethernet是Xerox公司的商标。

不会有路由问题，这是因为可以通过总线直接按地址编码确定每条消息的目的地。在环形网络中，所有消息沿着环向同一方向发送，并且由接收者或发送者将它们移去，这使路由问题变得无效。在总线中没有拥塞问题。因为每条消息在发送后可以很快地被接收到（从总线上取走）。但是限制在节点中等待进入总线的消息数量却是必要的。由于消息不在中间节点存储，就不会产生存储转发的死锁问题。如果所有计算机都是由家庭或信用其员工的某个公司所拥有，那么通常操作系统提供的保护即可解决安全问题。

在局域网上，运行一个分布式应用程序（进程集合，其分布在网络的节点上）需要解决以下分布式控制的问题，有些问题在第二部分讨论。

（1）*广播和同步*（第6章）　如果所有进程都必须得到信息或者只有当全局条件得到满足，所有进程才能继续进行，就需要一种消息传递模式，使消息到达所有进程。

（2）*选择*（第7章）　有些任务必须由一个集合中的某一进程来执行，例如，产生输出或初始化数据结构。如果有时值得或有必要，而又没有一个进程赋予这个任务优先级，就必须执行分布式算法，来选择一个进程执行这项任务。

（3）*终止检测*（第8章）　在分布式系统中，进程并不总是能够直接地观测到它们进行的分布式计算是否已经终止。这就需要进行终止检测，使计算结果更确切。

10

（4）*资源分配*　一个节点可能访问网络中某些资源，这些资源在网络的其他地方才可用，尽管可能不知道资源所处的位置。仅仅维持一张资源所处位置的列表是不够的，因为潜在资源的数量可能会很大，而且资源会从一个节点向另一节点迁移。在这种情况下，请求资源的节点可能会查询全部或者部分其他节点有关资源可用性的情况，例如，使用广播机制。在第6章中描述了关于此问题的算法所依据的波动机制，参见文献Baratz et al. [BGS87]。

（5）*互斥*　当进程必须依赖于一次只能被一个进程使用的公共资源时，就会引起互斥问题。这类资源可以是一台打印设备或要被写的文件。分布式算法有必要决定：当几个进程同时请求访问资源时，哪一个进程被允许先使用资源。同时保证只有当前一进程用完所需资源后，下一进程才能开始使用这一资源。

（6）*死锁检测和解决方法*　如果进程必须相互等待（当它们共享资源并且其计算要依赖于其他进程提供的数据时），周期性等待可能发生，此时不能进一步进行计算。因此必须检测出这些死锁的情形，并采取适当措施重新开始或继续计算过程。

（7）*分布式文件维护*　当节点对一远程文件进行读写请求时，这些请求的处理次序是任意的。必须做出规定来保证每一节点对文件读写次序的一致性。利用时戳（time stamp）以及文件中的信息可以做到这一点，并根据时戳，对不断到来的请求定义一个次序。参见文献[LL86]。

1.1.5 多处理器计算机

一台多处理器计算机是由几个规模较小的处理器组成的计算机，通常在一个大机盒内。它与局域网的区别在以下几个方面：它的处理器是同质的，即它们是由相同的硬件构成的。机器所处物理范围很小，通常在一米左右。在一个计算中使用多个处理器（要么提高速度，

要么提高可靠性）。如果多处理器计算机的主要设计目标是提高计算机的速度，常称它是并行 11
计算机。如果它的主要设计目标是提高可靠性，则称它是可复制的系统。并行计算机可以分成单指令、多数据（SIMD）机器和多指令、多数据（MIMD）机器。SIMD机器有一个指令解释器，但是指令是通过大量算术单元来实现。显然，这些单元缺乏关于分布式系统定义所要求的自主性，因此，本书将不考虑SIMD计算机。MIMD机由几个独立的处理器组成，它们归类为分布式系统。

处理器上通常配有与其他节点进行通信的专用硬件。处理器之间的通信可以通过总线或者点对点的链路。如果选择总线结构，其结构只能扩展到有限值。

Inmos开发的Transputer ⊖ 芯片曾经是用于设计多处理器计算机的广受欢迎的处理器。参见图1-3。Transputer芯片由中央处理单元（CPU），专用浮点运算单元（FPU），局部存储器和4个专用通信处理器组成。这种芯片非常适于构筑度为4的网络（每一节点与其他4个节点相连）。Inmos还生产专用通信芯片，被称为*路由器*。每台路由器能够同时处理32个Transputer链路的通信量。检查每一到来的消息，确定要转发的链路，然后经由那条链路发送。在90年前半叶，这些机器流行的程度达到巅峰。

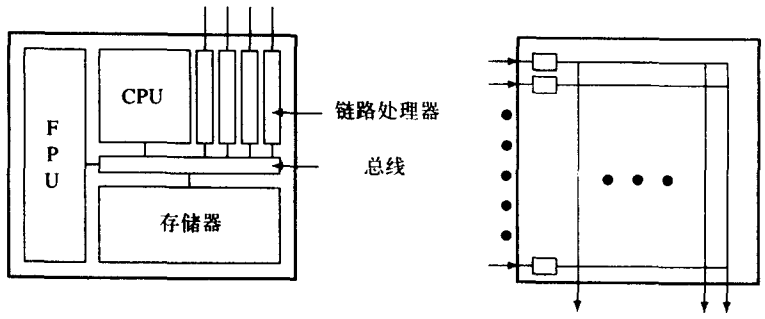


图1-3 TRANSPUTER和路由芯片

另一个并行计算机的例子是Thinking Machines公司［LAD+92］开发的Connection Machine CM-5 ⊖ 系统，并由Connection Machines Services 公司继续开发。该机器的每一个节点由快速处理器和向量处理单元组成。除了由许多节点提供并行性之外，还提供内部并行性。每一节点的潜在性能可以达到每秒1.28亿次操作，一个机器包含16 384个节点，整个机器每秒可执行10^{12}次以上的操作（由16 384个处理器组成的最大机器可占据900平方米的房间，也可能很昂贵）。CM-5的节点由三个点对点的通信网络连接而成。*数据网具有胖树*（fat tree）拓扑结构，用于在处理器间以点对点方式交换数据。*控制网络具有二叉树拓扑结构*，用于进行专用操作，如，全局同步和组合输入。*诊断网络*对于编程人员是不可见的，它用于传播失效组件的信息。该计算机支持SIMD和MIMD两种程序设计模式。

在并行计算机中，计算被分成若干子计算，每一节点执行一个子计算。在可复制的系统中，每一节点执行整个计算。计算之后，对结果进行比较并找出和改正错误。

多处理器计算机的构建，需要解决几个算法上的问题，有些类似于计算机网络上发生的

⊖ Transputer是Inmos公司的商标，现在属于SGS-Thomson Microelectronics。

⊖ Connection Machine是一注册商标，CM-5是Thinking Machines公司的商标。

问题。本书讨论了几个这样的问题。

（1）*消息传递系统的实现*　如果多处理器计算机所构成的是点对点的网络，则必须设计通信系统。这里所引出的问题类似于计算网络实现时所出现的问题，如，传输控制，死锁和拥塞预防。这些问题的解决办法常常要比一般计算机网络中要简单一些。例如，路由问题可以通过规则的网络拓扑结构（如环或网格）和可靠的结点来简化。

12
~
13

Inmos C104路由芯片利用非常简单的路由算法，称为区间路由（interval routing）算法，在4.4.2节中将对此进行讨论，这一算法不能有效地被应用在任意拓扑结构的网络中。这就提出了一个问题，是否其他一些问题的解决方法，例如死锁预防，可与此路由机制结合使用。

（2）*虚拟共享内存的实现*　许多并行算法是为所谓的并行随机访问存储器（PRAM）模型设计的，该模型中的每个处理器可以访问共享存储器。物理上共享的存储器的体系结构是不可扩展的。这就严格限制了一片存储器芯片上的处理器数目。

因而研究的方向应该是具有多个存储节点的体系结构，存储节点与处理器通过互联网连接。

（3）*负载平衡*　只有当计算的负载被均匀分布到各处理器上时，才能充分开拓并行计算机的计算能力。工作的负载集中在某个节点上，会降低节点的性能。如果在编译时能确定计算的所有步骤，就可能静态地分布计算负载。更困难的情况是在计算的过程中动态产生的任务；在这种情况下，需要复杂的技术。必须定期地对处理器的任务队列进行比较，之后将任务从一个处理器迁移到另一个处理器。有关负载平衡的某些技术和算法参见文献Goscinski[Gos91，第9章]或Harget和Johnson[HJ90]。

（4）*健壮性对不可测故障*（第三部分）在一个可复制的系统中，必须有一种机制来克服一个或多个处理器故障。当然，尽管网络中某一节点会发生故障，计算机网络还是要继续运行，在这种情形下，通常假设这种故障可以被其他节点检测出（参见4.3节的网络变更算法）。如果假设可复制的系统一定是正确的，则会带来更加严重的后果，这是因为处理器可能产生一个错误的响应，而按照协议就像运行正确的处理器。必须实现能对处理器结果进行筛选的选举机制，使得只有大多数处理器运行正确，才会发送正确的结果。

1.1.6　协同操作进程

14

通过将软件组织成（顺序）进程集可以简化复杂软件系统的设计。其中每一进程具有一个定义明确的简单任务。

描述这种简化的经典例子是Conway的*记录转换*（Conway's record conversion）。问题是读入80个字符的记录，并在125个字符的记录上写下同样数据。输入每条记录后，插入一个空格，每对星号（"**"）必须用一个感叹号（"！"）替代。每条输出记录后必须紧跟一个记录结束符（EOR）。可用程序实现这种转换，但是写这个程序还是很复杂的。所有功能，即用"！"替代"**"，空格的插入，以及EOR字符的插入，必须在一个循环内完成。

这个程序可以更好地结构化为两个协同操作的进程。第一个进程p_1读取输入卡，并把输入转换成可打印的字符流而不是分解成记录。第二个进程p_2接收字符流，每125个字符后插入一个EOR。通常假设将操作系统，电话交换中心，以及计算机网络中的通信软件设计成进程的

集合。参见1.2.1节。

设计成协同进程能够把应用变成逻辑上（logically）是分布式的，但也可能在同一台计算机上执行进程，在这种情况下，它不是物理上分布式的。对于逻辑上已经分布的系统，做到物理上分布要更容易些。计算机操作系统必须控制进程的并发执行，并且提供进程间通信和同步的方法。

同一处理器上执行的进程可以访问同一物理存储器，因此自然可以用存储器进行通信。一个进程写入存储器某一位置，另一进程从此位置读取。Dijkstra [Dij68] and Owicki and Gries [OG76] 采用了这种并发进程模型。本文中考虑的问题包含以下几方面。

（1）存储器操作的原子性 通常假设读写存储器的某一字是原子性的，即，一个进程读/写完之后，另一进程才可开始读/写。如果大于一个字的结构要被更新，就要仔细地进行同步 15
操作，以避免对部分更新的结构进行读取。例如，可以对结构实现互斥 [Dij68] 来达到同步：当某一进程访问结构时，不允许其他进程读/写结构。利用共享变量实现互斥是复杂的，因为几个进程可能同时寻找结构的入口。

互斥访问共享数据所强加的等待条件可能降低处理器的性能，例如，“快”的进程必须等待“慢”的进程当前正在访问的数据。近年来，人们关注原子共享变量实现的无等待性（wait-free），它表明一个进程的读/写无需等待其他进程。读／写操作可能会重叠，但经过仔细地设计读/写算法，原子性可以保持。关于无等待的原子共享变量算法可参见Kirousis与Kranakis[KK89]，或Attiya和Welch[AW98]。

（2）生产者消费者问题 两个进程，一个要写入共享缓冲区，另一个要从此缓冲区读出，二者要协作，以防止缓冲区满时第一个进程写入，缓冲区空时第二个进程读出的情况。当解决Conway转换问题时，就会引出生产者消费者问题：p_1产生中间字符流，p_2消费它们。

（3）无用存储单元的收集 利用动态数据结构进行应用程序设计时，可能会产生一些不用的存储单元，称为无用存储单元（garbage）。以前，当存储系统用完自由空间时，必须中断应用程序，调用无用单元收集程序（garbage collector），找出并回收那些不用的存储单元。Dijkstra et al.[DLM+78]提出了不工作（on-the-fly）无用单元收集程序，它可作为单独的进程，与应用程序并发执行。

由于应用可能修改存储区中的指针结构，而收集程序要决定哪一单元是无用的，因此需要在应用和收集程序之间进行复杂的合作。必须仔细分析算法，来表明所作的修改不会引起有用单元被错误回收。Ben-Ari[BA84]给出了on-the-fly无用单元收集算法的更加简单的正确性证明。

这里所列出的问题的解决方法表明，可以为通过共享存储器进行通信的进程解决进程交互的困难问题。然而，其解决方法常常是非常复杂的，有时一个非常微妙的不同进程的步骤 16
交错就会导致那些咋一看似乎是正确的解决方法的错误结果。因此，操作系统和程序设计语言为更结构化的组织进程间的通信提供原语。

（1）信号量（Semaphore） 信号量 [Dij68] 是一个非负的（整数）变量，在一次原子操作内其值可被读写。一次V操作使其值增加1，当它的值为正时，一次P操作使其值减小1（只要其值为0，则将执行该进程挂起）。

信号量是实现共享数据结构上互斥的合适工具：信号量初始化为1，在访问数据结构之

前进行一次P操作，之后跟着一次V操作。信号量赋予每一进程正确使用的责任，如果一个进程错误地操作了数据，或者没有执行所要求的P和V操作，则共享数据的完整性就要受到侵犯。

（2）管程（Monitor）　管程 [Hoa74] 由数据结构和过程集组成。它以互斥的方式调用进程执行这些过程。由于数据仅能经过管程中声明的过程进行存取，因此只要正确地声明管程，就能正确使用数据。管程限制了对数据的无限存取，并通过不同进程来使这些存取同步。

（3）管道（Pipe）　管道是一种移动数据流的机制，它使数据流从一个进程移动到另一个进程。并使两个通信的进程同步。它是生产者和消费者问题的预编程解决办法。

在UNIX操作系统中，管道是最基本的通信机制。如果程序p_1实现了Conway的问题转换进程p_1，程序p_2实现了Conway的进程p_2，那么UNIX命令$p_1 | p_2$调用两个程序，并通过管道将它们连接起来。p_1的输出被放入缓冲区，成为p_2的输入；当缓冲区满时，挂起p_1，当缓冲区空时，挂起p_2。

（4）消息传递（Message passing）　某些程序设计语言，如occam和Ada，为进程间通信
17 提供了消息传递机制。利用消息传递解决同步问题相对要容易一些；因为消息在发送之后才接收，消息交换导致了事件间的临时关系。

利用管程或者管道可以实现消息传递。自然地，消息传递是运行在分布式硬件（无共享存储器）上的系统的通信工具。occam和Ada语言实际上是考虑为具有物理上分布的应用程序而开发的。

1.2 体系结构和语言

实现计算机通信网的软件是非常复杂的。本节解释该软件的构成。它由一些称为层（1.2.1节）的相关模块组成。我们讨论两种网络体系结构标准，即，作为广域网标准的开放系统互连（open systems interconnection，OSI）模型和局域网的IEEE补充标准（1.2.2节和1.2.3节）。同时简要地讨论了分布式系统所用的程序设计语言（1.2.4节）。

1.2.1 结构

在分布式系统，由通信子系统执行的任务的复杂性，要求子系统的设计高度结构化。为此，将网络划分成模块，每一模块执行确定的功能，并依赖其他模块所提供的服务。在网络的组织中，模块之间，有严格的层次（hierarchy），每一模块只使用上一层模块所提供的服务。在网络实现中，称模块为层或者级。参见图1-4。

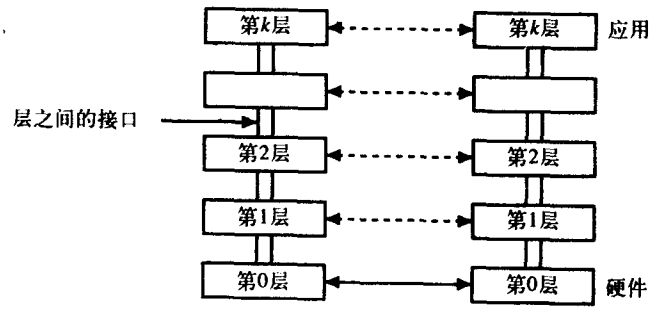


图1-4　分层网络体系结构

每一层实现网络功能一部分，并依赖于它的下一层。在第i层到第i+1层的接口（简记，i （i+1)接口）中，详细描述了第i层到第i+1层所提供的服务。设计网络时，第一件要做的事情就是定义层数和子层的接口。

每一层的功能由分布式算法实现。假设（i-1)/i接口已经定义，第i层的算法解决i/(i+1)接口所定义的“问题”。例如，(i-1)/i接口确定了消息从节点p向节点q传送，但有些消息可能丢失，而i/(i+1）接口确定了消息从节点p向节点q可靠地传送。第i层的算法问题就是利用不可靠的消息传递来实现可靠的消息传递，通过确认和重发丢失消息就可做到（参见1.3.1节和第3章)。问题的解决方法定义了通过第i层的进程所交换的消息的类型以及这些消息的含义，如，进程是如何对这些消息做出反应的。第i层进程之间 “谈话”所用的规则和约定称为*i-层协议*(layer-i protocol)。

最底层（图1-4中第0层）总是硬件层。0/1接口描述了一些过程，通过这些过程，第1层可以通过连接线路传送信息，并且层自身的定义确定了所用导线的类型，以及多少伏特电压表示一个1或多少伏特电压表示一个0等。重要的一点是在第0层实现的变化（连线用其他线或者卫星网络替换）并不要求0/1接口改变。同样规则对于高一层也适用：层接口用于将一个层的实现与其他层屏蔽开，并可以对实现进行改变而不影响其他的层。

通过网络的*体系结构*（architecture)，我们确定了层的集合以及所有接口及协议的定义。因为网络可能包括不同厂商所生产的节点，不同公司编写的软件，所以不同公司产品的兼容性是非常重要的。兼容性的重要性受到广泛认可，因此已经开发出标准的网络体系结构。下一节，讨论两个标准，因为它们被具有影响力的组织（国际标准化组织，ISO以及电气与电子工程师协会，IEEE）采用。

18 ~ 19

TCP/IP是因特网中所用的协议集。TCP/IP并不是官方标准，但它的广泛使用使其成为事实上的标准。TCP/IP协议组是按下节讨论的OSI模型进行结构化的，但协议在局域网和广域网中都可使用。较高一层包含了电子邮件（简单邮件传输协议，SMTP)、文件传输（文件传输协议，FTP）和远程登录双向通信的协议（Telnet)。

1.2.2 OSI参考模型

国际标准化组织（ISO）已经为广域网上使用的计算机网络产品制定了标准。网络体系结构的标准称为*开放系统互连*（OSI）参考模型，本节将对它作简单描述。由于这一标准并不完全适合局域网络，下一节讨论IEEE关于局域网的附加标准。

OSI参考模型由7层组成，即，*物理层*、*数据链路层*、*网络层*、*传输层*、*会话层*、*表示层*和*应用层*。该参考模型确定了每层的接口并为每一层提供了一到两种标准协议（实现该层的分布式算法)。

1. 物理层

物理层的目的是在通信信道上传输比特序列。正如本层的名称所示，通过两个节点间的物理连接，如，电话线、光纤、卫星就可达到目标。本层的设计纯粹是电气工程师的事。其中1/2接口确定了下层调用物理层服务的过程。物理层的服务并不可靠，在传输过程中，位流可能混在一起。

2. 数据链路层

数据链路层的作用是屏蔽物理层的不可靠性，即，为更高一层提供可靠的链路。数据链

20 路层仅仅实现由物理链路直接相连的节点间可靠连接，因为它是直接建在物理层上的（非相邻节点间的通信在网络层实现）。

为此，本层将位流分成固定长的片，称为*帧*（frame）。由帧的接收者验证它的*校验和*（checksum），并判定该帧是否被正确接收。校验和是加在每帧后的冗余信息。通过从接收方到发送方的反馈机制，通知发送方所发帧是否被正确接收，依靠*确认消息*（acknowledgement message）完成反馈。如果发送方接到接收不正确或者完全丢失的信息，就会重发该帧。

可以将上节中解释的一般性原理精化，使其适用于不同的数据链路协议。例如，可以在接收若干帧之后发送确认消息（正确认），或者在丢失接收的若干帧之后发送确认消息（负确认）。在正确地传输所有帧之后，正确传送所有帧的最终责任可以落在发送方也可以落在接收方。可以对一帧或多帧进行确认，每帧可以有序号或者没有。

3. 网络层

网络层的作用不仅仅只是为那些经物理信道连接的节点提供通信手段，而是为所有节点之间的通信提供手段。该层还要为那些不相邻节点的通信，选择通过网络的路由，并控制每一节点和信道的通信负载。

通常根据*路由表*中的网络拓扑结构信息选择路由，每一节点存有一张路由表。如果网络的拓扑结构发生变化（由于节点或信道的故障或恢复），网络层包含*更新路由表*（routing table）的算法。这样的故障或恢复由数据链路层检测。

尽管数据链路层为网络层提供了可靠的服务，但网络层提供的服务是不可靠的。从一个节点向另一节点发送的消息（在这一层称为*包*，packet）可能沿不同路径传送，从而导致一条消息赶上另一条消息的情况发生。由于节点的故障，消息可能会丢失（节点可能在还持有消息的时候，停止作业）；由于过量的重传，消息甚至可能被复制。该层可能会保证一个有限
21 的包生存期；例如，存在常数c，满足在c秒内，每一包或者被发送至目的节点，或者丢失。

4. 传输层

传输层的作用是屏蔽网络层所带来的不可靠性，为任意两节点之间提供可靠的*端对端*（end-to-end）通信。这个问题类似于数据链路层中解决的问题，但由于消息的多次复制及消息重排使问题复杂化。由于网络层不能保证包的生存期限，这使得不可能利用周期序列号。

传输层中传输控制所用的算法类似于数据链路层中的算法：序列号，经确认的反馈和重传。

5. 会话层

会话层的作用是为维持不同节点的进程之间的连接提供便利。一条链路可以打开也可关闭，且在链路打开与关闭之间，使用话路地址而不是复制带有每条消息的远程进程地址，链路可用于进行数据交换。会话层利用传输层提供的可靠的端到端通信。

会话可用于文件传输和远程登录。如果一个节点在会话过程中损毁，会话层能够提供恢复机制；如果在两端不能同时执行临界区操作，会话层还提供互斥机制。

6. 表示层

当节点上信息的表示互不相同，或者不适合于传输时，通过表示层进行数据转换。在本层的下面（即5/6接口处），数据是可传输的标准形式；而在本层之上（即6/7接口处），数据是用户或者计算机确定的形式。本层还进行*数据压缩*（compression）和*解压缩*（decompression），以减小交给较低一层的数据量。本层进行数据*加密*（encryption）与*解密*（decryption），来保

证数据的安全性和完整性，防止传输数据受到恶意破坏。

7. 应用层

应用层完成具体的用户要求，如文件传输、电子邮件、公告牌或虚拟终端。由于可能的应用的广泛性，因而不可能对这一层的所有功能标准化，但是对这里所列出的应用已经提出了标准。 22

1.2.3 局域网络OSI模型：IEEE标准

OSI参考模型的设计在某种程度上受到现有广域网络体系结构的影响。由于局域网中可能没有某些层，局域网中所用的技术提出了不同的软件需求。如果网络的组织依赖于所有节点共享的总线（参见1.1.4节），那么网络层几乎是空的，因为各对节点都直接通过总线相连。与中等规模点对点网络相比，通过限制由总线引入的不确定性因素，可以简化传输层的设计。与此相比，由于同一物理介质潜在地可能被大量节点访问，所以数据链路层更复杂。

为了解决这些问题，IEEE通过了附加标准。这个标准只覆盖OSI层次的较底层，可用于局域网（更准确地说，用于总线结构的网络，而不是点到点的网络）。因为没有一个标准能够适用于现在广泛使用的所有网络，IEEE通过了三个不同的、非兼容的标准，即CSMA/CD，令牌总线和令牌环。用两层子层替代数据链路层，即*介质访问控制*（medium access control）子层和*逻辑链路控制*（logical link control）子层。

1. 物理层

IEEE标准中的物理层的作用类似于最初ISO标准中的物理层，即传输位序列。实际的标准定义（线路的类型等）大不相同。这是由于所有通信是通过公共可访问介质，而不是通过点到点的链路。

2. 介质访问控制子层

这一子层的目的是解决使用共享通信介质的节点之间出现的冲突问题。静态方法是对每一节点允许访问介质的时间区间进行调度。然而这种方法浪费大量带宽，如果只有几个节点有数据传输，其他节点无事可做；在调度这些节点时，介质处于空闲状态。

在令牌总线和令牌环中，对介质的访问是基于循环方式的：节点循环地请求一个权力，称为令牌，只允许持有令牌的节点利用介质。如果持有令牌的节点没有数据传输，它就把令牌传给下一节点。在令牌环中，节点得到令牌的循环次序取决于物理链路的拓扑结构（即，环）。在令牌总线中，这个循环次序动态地取决于节点地址的顺序。 23

在具有冲突检测的载波侦听多路访问（CSMA/CD）标准中，当节点观测到介质空闲时，它们就被允许发送。如果两个或多个节点（大约）同时发送消息，就会产生*冲突*，当被检测到时，引起各节点中断其传输，稍候再发。

3. 逻辑链路控制子层

这一层的作用相当于OSI模型中数据链路层的作用，即控制两节点间数据的交换。本层提供差错控制和流控制，它所使用的技术类似于OSI协议中使用的技术，即，序列号和确认。

从较高层来看，逻辑链路控制子层就像是OSI模型中的网络层。事实上，任意一对节点间无需中间节点就可以通信，逻辑链路子层直接可对它们操作。在局域网中没有实现单独的网络层，而是将传输层直接地建在逻辑链路控制子层的上面。

1.2.4 语言支持

不论是通信网还是分布式应用，它们各软件层的实现都需要将该层或应用所使用的分布式算法通过一种程序设计语言进行编码。实际的编码主要受到语言的影响，尤其是受到它所提供的原语的影响。因为在本书中，我们主要集中在算法方面，并不关心把它编码成程序，进程的基本模型是基于进程的状态和状态转移（参见2.1.2节），而不基于执行从规定的集合中获取的指令。当然，对于我们所提出的算法，一定要求用形式化的表示方法。书中所用的程序设计语言表示参见附录A。

本节我们对分布式系统的程序设计语言的结构进行描述。这里仅对这些结构作简要描述；更多的细节和实际使用各种结构的语言的例子参见文献Bal[Bal90]。分布式应用的程序设
24 计语言必需提供表达并行性、进程交互作用以及非确定性的手段。并行性要求系统中的各节点能够并发地执行程序中的它们各自的那一部分。通过程序设计语言来支持节点间的通信。因为一个节点有时必需能够接收来自不同节点的信息或者能够发送/接收一条消息，这就对非确定性提出了要求。

1. 并行性

在一个分布式系统中，最合适的并行度取决于通信开销和计算开销的比例。较大的并行度可以快速地执行，但也要求更多通信。如果通信代价高，在计算速度方面所得收益将会在通信中损失。

通常定义几个进程（process）来表示并行性，每一进程是一个顺序的实体，具有自己的状态空间。语言可以静态定义进程集，也可以允许动态创建进程和终止进程。还可以用并行语句表示并行性。语言中的并行性并不总是显式的。将代码划分成并行进程由复杂编译器来进行。

2. 通信

对于分布式系统，进程间的通信是固有的。如果进程不进行通信，每一进程独立于其他进程，就可以单独对它进行研究，它就不是分布式系统的一部分。进程在计算过程中，如果需要另一进程产生的中间结果时，就会需要通信。同样进程之间也需要同步。这是因为进程在未得到它所需的结果时，必须被挂起。消息传递（message passing）可以完成通信和同步。而共享存储（shared memory）只能进行通信。此外还需关注利用共享存储器进行通信的进程的同步。

在提供消息传递的语言中，“send”和“receive”操作是可用的。在一个进程（这里称发送方进程）执行发送操作而另一进程（接收方进程）执行接收操作就会发生通信。发送操作的变量包括接收方的地址和附加数据，形成了消息的内容。当执行完接收语句时，对于接收
25 方才可用附加数据。此即实现了通信。只有当发送操作执行完之后，接收操作才能完成。这实现了同步。在一些语言中，接收操作并不是显式的，而是，当接到消息时，激活一个过程或操作。

语言可能提供同步（synchronous）消息传递机制，在这种情况下，仅当接收操作执行完后，发送操作才可完成。换句话说，在消息未接收完之前，发送方受到阻塞。导致发送方和接收方双向同步。

消息也可按点到点发送，从发送方向接收方，或者向其他节点广播，所有接收方接收同一消息。用组播来表示将消息发送到进程集（不必是所有进程）。远程过程调用（remote

procedure call，RPC）是稍微更结构化的通信原语。为了与进程b通信，进程a调用出现在进程b中的过程，并在消息中发送过程的参量。在另一消息中返回过程的结果之前，进程a被挂起。

另一种消息传递是利用共享存储器（shared memory）进行通信；一个进程将值写入变量，另一进程读取该值。难以达到进程间的同步，因为读取变量可以在写入变量之前。利用同步原语，如信号量（semaphore）[Dij68]或者管程[Hoa78]，在共享变量环境中实现消息传递是可能的。反之，在消息传递的环境中，实现一个（虚拟）共享存储器也是可能的，但是效率很低。

3. 非确定性

在进程执行中的许多地方，进程可能以各种不同方式继续运行。接收操作常常是非确定性的，这是因为它允许接收来自不同发送方的消息。表达非确定性的另一种方式是基于保护命令（guarded command）。最一般形式的保护命令是一张语句列表，每一语句之前有一个布尔表达式（它的保护）。只要语句的相应的保护求值为真，进程就能继续执行。保护可能包括一个接收操作，在这种情况下，如果接收到一条消息，其值就为真。

1.3 分布式算法

上节给出了利用分布式计算机系统的原因，并解释了这些系统的性质；其结果是需对这些系统进行程序设计。分布式系统的程序设计一定基于正确、灵活和有效的算法。本节，所 26
讨论的分布式算法的开发在本质上不同于开发集中式算法所使用的技术。分布式和集中式系统在许多方面有本质的不同，1.3.1节讨论这些不同点，并在1.3.2节给出例证。分布式算法研究作为独立的科学研究领域得到了发展，参见1.3.3节。本书旨在将此研究领域介绍给读者。本书的目标以及所选的内容在1.4节阐明。

1.3.1 分布式算法与集中式算法

分布式系统与集中式（一个处理机）计算机系统的不同，主要表现在以下三个方面：

（1）*缺乏全局状态知识* 在集中式算法中，基于对系统状态的观察做出控制决策。即使在一次单机操作过程中，通常不能访问整个状态，程序可以一个接一个地检查变量，考虑所有相关的信息后做出决策。在检查及决策的过程之间，不修改数据，从而保证了决策的完整性。

分布式系统中的节点只可以访问它们自己的状态，不能访问整个系统的全局状态。因此它不可能基于全局状态来做控制决策。一个节点可以接收其他节点的状态信息，并以此信息作为控制决策的基础。与集中式系统相比，所接收的陈旧信息可能导致信息失效。因为在发送信息的过程和基于此而做决策的过程之间，其他节点的状态可能已经改变。

节点从来不能直接地观察到通信子系统（即，在某时刻，哪些消息正在发送过程中）的状态。这个信息只能通过比较节点所发送和接收的消息间接推出。

（2）*缺乏全局时间帧* 构成集中式算法执行的事件，完全自然地按照事件发生的时间定序；对于每一对事件，或一个早于一个，或一个迟于一个。组成分布式算法执行的事件所导 27
致的时序关系并不是全序的，对于某些对事件，可以决定事件的先后，但对于其他一些事件，则不能确定时序[Lam78]。

在集中式系统中可以实现互斥，如果进程p对资源的访问在进程q的访问之后，那么进程p必须在进程q结束之后才可开始对资源的访问。的确，所有这类事件（进程p和q的开始时间和

结束时间）通过时序关系完全可以决定顺序；而在分布式系统中，却不能做到这一点。进程p和q可能开始访问资源，但是它们开始的先后次序不能确定。

（3）*非确定性*　集中式算法描述计算，是明确地从某一输入开始的；给定程序和输入，只有一种计算可能。相反，由于系统各组件在执行速度上的差异，分布式系统的执行通常具有不确定性。

考虑一下服务器进程可能接受许多未知客户进程请求时的情形。在所有请求没有接收完时，服务器进程不能挂起对请求的处理，因为不知道将会到达多少消息。因此，必须立即处理每一请求，处理的顺序就是请求到达的顺序。尽管可能知道客户发送请求的顺序，但由于传输延迟未知，请求到达的顺序可能与发送顺序不同。

缺乏全局状态、缺乏全局时间帧和非确定性，使分布式算法的设计技术极为复杂因为这三方面会以几种方式，进行干扰。

时间和状态的概念联系紧密；在集中式系统中，可将时间定义为系统执行过程中的状态序列。即使在分布式系统中，可以定义全局状态，并把执行看作全局状态的序列（参见定义2.2），这种观点也具有局限性，因为也可用其他全局状态序列描述系统的执行（定理2.21）。这些可选序列通常由不同的全局状态组成；这导致“在系统执行中，假设了这个或那个状态”
28 意义不明确。

如果能从要被执行的算法预测全局状态，可能会弥补全局状态知识的缺乏。令人遗憾的是，由于分布式系统固有的非确定性，这一点不可能做到。

1.3.2　一个例子：单消息通信

我们用一个例子，来说明由于缺乏全局状态知识和全局时间-帧，所引起的困难，即在文献[Bel76]中由Belsnes所讨论的在不可靠的介质上实现可靠的信息交换。考虑由数据网络连接的两个进程a和b，从一个进程向另一个进程传输消息。消息发出之后，可以在任意时间内接收，但是也可能在网络中丢失。*网络控制过程*（network control procedure，NCP）的使用增加了通信的可靠性，通过NCP，进程a和b访问网络。进程a给NCP A一个信息单元m来初始化通信。NCP之间的交互作用（经过数据网络，DN）必须保证信息m被发送到进程b（通过NCP B），之后，NCP A 通知进程a关于信息的发送。图1-5描述了它们的通信结构。

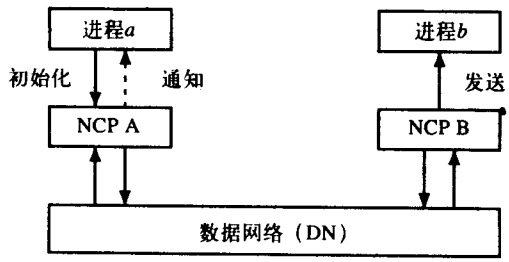


图1-5　简化的网络体系结构

即使进程a只传送一个信息单元给进程b，由于网络不可靠性，NCP A 与NCP B之间的对话由几条消息组成。它们要维持这个对话的状态信息，但由于与每一进程可能的对话对象数量很大，因此在消息交换完成之后，状态信息即被抛弃。状态信息的初始化称作*打开*，抛弃
29 状态信息称作*关闭*。可见，在关闭对话之后，NCP又完全回到打开之前的状态；这称为*关闭*

状态。

如果b通知a接收到消息，而该单元实际上从未发送到b，则称信息单元m是丢失的，如果它被发送两次，则称单元m为可复制的（duplicated）。可靠的通信机制不仅防止复制而且防止丢失。假设NCP可能失效（损毁），失效后，在关闭状态可以重新启动（有效地丢失所有关于当前打开对话的信息）。

1. 可靠的通信不存在

由此看来，不管NCP设计的如何复杂，也不可能达到完全可靠的通信。这一结论独立于数据网或者NCP的设计并仅仅取决于这样一个假设：NCP可能丢失有关一次激活对话的信息。

为了表明这一点，假设进程a进行通信初始化之后，NCP A 与NCP B开始对话，在接到来自NCP A的消息m之后，假定NCP B将消息m发送至b。考虑一下这样一种情况：NCP B损毁，并且在NCP A发送消息之后，在关闭状态重新启动。在这种情况下，当NCP B损毁时，NCP A和NCP B都不能断定是否m已经发送；这是因为NCP A不能观察到NCP B中的事件（缺乏全局状态知识），并且NCP B已经损毁，在关闭状态重新启动。不管NCP如何继续进行对话，还是会引入一个错误。如果NCP A再次给NCP B发送消息，NCP B传送这条消息，则消息复制可能出现。如果a所接到的通知是没有进行传送的，则可能出现消息丢失。

针对丢失或者复制消息的情形，我们对NCP的几种可能设计做一评价。我们试图设计这样一种协议，在任何情况下，它都可以避免丢失。

2. 一消息对话

在最简单的设计中，初始化之后，NCP A单独由网络发送不可更改的数据，并通知a，然后关闭。NCP B总是将所接收的数据传送给b，传送之后即关闭。

当网络不能发送消息时，这种协议就会引起消息丢失，但是它不可能引入消息复制。

3. 两消息对话

在协议中增加一个确认，就可对消息进行有限的保护以防丢失。在一个正常的对话中， 30
NCP A发送数据消息< **data**, m >，并等待来自NCP B的确认消息< **ack** >。当接到这个确认消息时，NCP A关闭对话。NCP B一旦接收到这一消息< **data**, m >，就把m传送到b，并用< **ack** >消息进行应答，然后关闭。总之，无差错的对话由三个事件组成。

1）NCP A 发送 < **data**, m >。

2）NCP B 接收 < **data**, m >, 传送m, 发送 < **ack** >, 关闭。

3）NCP A 接收 < **ack** >, 通知, 关闭。

如果在某段时间后，NCP A没有接到确认应答，数据丢失的可能性迫使NCP A再次发送数据< **data**, m >。（由于缺乏全局状态知识，在接收< **data**, m >和发送< **ack** >之间，NCP A不能观测到是否< **data**, m >已丢失、< **ack** >已丢失或者NCP B损毁。）因此，NCP A在有限时间内，等候接收确认消息，如果不能接收到这一消息，则计时器断定时间用完，就会产生超时（timeout）信息。显而易见，这种选择重传消息的机制引入了复制的可能性。即，不是原数据消息的丢失，而是确认消息的丢失，情形如下：

1）NCP A 发送 < **data**, m >。

2）NCP B 接收 < **data**, m >, 传送 m, 发送 < **ack** >, 关闭。

3）DN < **ack** > 丢失。

4）NCP A 超时, 发送 < **data**, m >。

5）NCP B　接收 < **data**, m >, 传送 m, 发送 < **ack** >, 关闭。

6）NCP A　接收 < **ack** >, 通知, 关闭。

确认不仅能引起复制的可能性，而且不能进行保护以免丢失，情形如下。进程a提供两个信息单元m_1和m_2进行传输。

1）NCP A　发送 < **data**, m_1 >。

2）NCP B　接收 < **data**, m_1 >, 传送 m_1, 发送 < **ack** >, 关闭。

3）NCP A　超时, 发送 < **data**, m_1 >。

4）NCP B　接收 < **data**, m_1 >, 传送 m_1, 发送 < **ack** >, 关闭。

5）NCP A　接收 < **ack** >, 通知, 关闭。

6）NCP A　发送 < **data**, m_2 >。

7）DN　　< **data**, m_2 > 丢失。

8）NCP A　接收 < **ack** > (step 2) , 通知, 关闭。

31 消息m_1，如前一情形，被复制。但第一次发送的确认消息没有丢失而是较慢。引起稍后信息单元的丢失。由于没有全局时间，不能检测出较慢的发送。

如果假设有一个较弱的全局时间，即，对于任何通过网络发送的消息的传输延迟存在上限T。就可更容易地解决可靠的进程间通信问题。这被认为是全局（global）定时假设，因为它导出了不同节点（NCP A发送和NCP B接受）中事件之间的时序关系。这个协议可以防止过早接收对话消息，因为在发送最近消息的2T时间后，才关闭NCP A中的对话。

4. 三消息对话

当一个确认应答丢失或者延迟，两消息协议就会丢失消息或者复制信息。一种解决的方法是在对话中加入第三个消息，通知NCP B，NCP A已经收到确认。一般对话由下列事件组成。

1）NCP A　发送 < **data**, m >。

2）NCP B　接收 < **data**, m >, 传送 m, 发送 < **ack** >。

3）NCP A　接收 < **ack** >, 通知, 发送 < **close** >, 关闭。

4）NCP B　接收 < **close** >, 关闭。

由于消息< **data**, m >丢失，引起NCP A超时，在这种情况下，NCP A重发消息。< **ack** >消息丢失也引起< **data**, m >消息重发，但这不会导致复制，因为NCP B存在一个打开的对话，并且能够识别它已经接收的消息。

令人遗憾的是，该协议仍然存在丢失和复制信息的可能。因为当< **close** >消息丢失，NCP B也必须关闭，如果NCP B没有接到< **close** >消息，它必须重传< **ack** >消息。在NCP B关闭之后，NCP A应答没有对话（< **nocon** >）。重传的< **ack** >可能会到下一次NCP A的对话中，被
32 当作那次对话中的确认消息，在丢失之前，引起下一次信息。情形如下。

1）NCP A　发送 < **data**, m_1 >。

2）NCP B　接收 < **data**, m_1 >, 传送 m_1, 发送 < **ack** >。

3）NCP A　接收< **ack** >, 通知，发送 < **close** >, 关闭。

4）DN　　< **close** > 丢失。

5）NCP A　发送 < **data**, m_2 >。

6）DN　　< **data**, m_2 > 丢失。

7）NCP B 重传 < **ask** >（第2步）。

8）NCP A 接收 < **ack** >，通知，发送 < **close** >，关闭。

9）NCP B 接收 < **close** > 关闭。

由于一个对话中的消息干扰了另一对话中的消息，因而再次引出问题。可以为每个新的对话选择一对新的对话识别号解决这一问题，NCP A，NCP B各一个。所选的识别号包括在所有对话的信息中，用于验证所接到的消息是否确实属于当前对话。一般的三消息协议如下。

1）NCP A 发送 < **data**, m, x >。

2）NCP B 接收 < **data**, m, x >，传送 m，发送 < **ack**, x, y >。

3）NCP A 接收 < **ack**, x, y >，通知，发送 < **close**, x, y >，关闭。

4）NCP B 接收 < **close**, x, y >，关闭。

对三消息协议修改排除了早先给出的错误对话，因为NCP A在第8步中所接到的消息并不被认为是对第5步所发送的数据消息的确认应答。然而，在传送m之前（第2步），NCP B并不验证一条消息< **data**, m, x >的有效性，这很容易导致信息的复制。如果第1步中所发的消息被延迟并重新发送，后到的< **data**, m, x >消息将引起NCP B再次传送信息m。

当然，如果NCP B在发送数据之前接到消息，也应该验证消息的有效性。可考虑对三消息对话做修改，使得NCP B在第4步而不是第2步发送数据。在NCP B发送之前，NCP A给出通知，但由于NCP B已经接到信息，这似乎是合理的。尽管必须保证，NCP B现在可以在任意情况下发送数据。尤其是，当< **close**, x, y >消息丢失时。NCP B复制< **ack**, x, y >消息，NCP A用< **nocon**, x, y >消息应答，这引起NCP B发送和关闭，情形如下。 33

1）NCP A 发送 < **data**, m, x >。

2）NCP B 接收 < **data**, m, x >，发送 < **ack**, x, y >。

3）NCP A 接收 < **ack**, x, y >，通知，发送 < **close**, x, y >，关闭。

4）DN < **close**, x, y > 丢失。

5）NCP B 超时，重发< **ack**, x, y >。

6）NCP A 接收 < **ack**, x, y >，应答 < **nocon**, x, y >。

7）NCP B 接收 < **nocon**, x, y >，发送m，关闭。

由此，为了避免信息丢失，即使NCP A不能确认与x和y有连接，NCP B也必须发送数据。这就致使有效性机制对于NCP B是无效的，导致复制信息的可能性。

1）NCP A 发送 < **data**, m, x >。

2）NCP A 超时，重发数据 < **data**, m, x >。

3）NCP B 接收 < **data**, m, x >（第2步中发送），发送 < **ack**, x, y_1 >。

4）NCP A 接收 < **ack**, x, y_1 >，通知，发送< **close**, x, y_1 >，关闭。

5）NCP B 接收 < **close**, x, y_1 >，发送m，关闭。

6）NCP B 接收 < **data**, m, x >（第1步中发送），发送 < **ack**, x, y_2 >。

7）NCP A 接收 < **ack** x, y_2 >，应答 < **nocon**, x, y_2 >。

8）NCP B 接收 < **nocon**, x, y_2 > 应答< **ack**, x, y_2 >，发送m，关闭。

5. 四消息对话

如果在发送数据之前，NCP之间认同其对话识别号，就可以避免以前对话的信息传送。

1）NCP A 发送 < **data**, m, x >。

2）NCP B　接收< **data**, m, x >, 发送 < **open**, x, y >。

3）NCP A　接收 < **open**, x, y >, 发送 < **agree**, x, y >。

4）NCP B　接收 <**agree**, x, y >, 传送m, 发送 < **ack**, x, y >, 关闭。

5）NCP A　接收 < **ack**, x, y >, 通知, 关闭。

NCP B损毁的可能性迫使差错处理满足，即使实际上NCP 没有损毁时，复制仍然会出现。当NCP B接收到消息< **agree**, x, y >并且没有对话打开时，它就发送错误消息< **nocon**, x, y >。假设NCP A没有接到消息< **ack**, x, y >，即使重传几次< **agree**, x, y >之后，也只能接到< **nocon**, x, y >消息。因为在接收到< **agree**, x, y >以前，NCP B已经损毁，这迫使NCP A打开
34 一个新的对话（通过发送< **data**, m, x >）以防止m丢失！但是NCP B可能已经传送m，并且消息< **ack**, x, y >已经丢失，在这种情况下，会引发复制。

可以用这样一种方法修改协议，这就是一旦接收到消息< **nocon**, x, y >，NCP A即通知并关闭；这可以防止复制，但可能引起消息丢失，这是不可取的。

6. 五消息对话和比较

Belsnes[Bel76]在文献中给出了一种五消息协议，不丢失信息而且只有在NCP实际损毁时才进行复制。因此，根据观察，既然不可能有可靠的通信，这可能是最好的协议。由于开销过大（传输一个信息单元，NCP要交换五次消息），是否五消息协议一定比简单的两消息协议更可取，这是令人怀疑的。事实上，因为即使五消息协议也可能引入复制（当NCP损毁时），进程级必须处理这种情形。两消息协议，可能带来复制，但如果在协议中加入对话识别（就像在三消息协议中所做的那样）就可避免消息丢失，因此也可以被采用。

1.3.3 研究领域

在过去20年里，对于分布式算法的研究一直都在继续着，尤其是在20世纪80年代，取得了相当大的进步这一领域逐渐走向成熟。在已讨论各节中，我们指出的某些技术上的进展激励着分布式算法的研究，即计算机网络（广域网和局域网）和多处理器计算机的设计。最初的研究目标着眼于算法在广域网中的应用，但是如今该领域已经开发出了精确数学模型，允许将研究结果和方法应用到更加宽广的分布式环境。然而，该项研究与通信技术中的工程发展保持着紧密的联系，这是因为算法的结果常常对网络模型的变化非常敏感。例如，廉价的微处理器使得构造由大量相同处理器组成的系统成为可能，并促进了“匿名网络”的研究（参见第9章）。

有几种期刊和每年一度的会议专门关注分布式算法及分布式计算的研究和进展。还有一
35 些刊物和会议虽然不是专攻该领域，但也包含许多该领域的出版物。自1982年以来，每年在北美都组织*分布式计算原理*（Principles of Distributed Computing，PoDC）的年度会议，它的会议论文由美国计算机学会出版。国际*分布式算法专题*（International Workshops on Distributed Algorithms，WDAG）曾于1985、1987、1989年分别在Ottawa，Amsterdam和Nice举行。自那时起，每年Springer-Verlag都将这些会议论文以*计算机报告*（Lecture Notes on Computer Science）的丛书形式出版。1998年，该会议的名字改为*分布式计算*（Distributed Computing，DISC）。每年在*计算理论专题*（Symposia on Theory of Computing，SToC）和*计算机科学基础*（Foundations of Computer Science，FoCS）的专题报告覆盖了计算机领域的所有主题，常常有分布式计算方面的论文。SToC会议的论文集由计算机学会出版，FoCS会议的

论文集由IEEE出版。*并行和分布式计算*（Journal of Parallel and Distributed Computing，JPDC）和*分布式计算*（Distributed Computing）期刊定期发表分布式算法方面的论文。*信息处理快报*（Information Processing Letters，IPL）也定期出版分布式算法方面的论文。

1.4 本书概要

本书以下列三个目标为宗旨。

（1）使读者熟悉研究给定分布式算法性质所用的技术、分析和解决分布式系统中出现的问题以及评价某一网络模型的价值所用的技术。

（2）提供对几种系统模型的固有可能性和不可能性的洞察力。在3.2节、第12章和第15章研究了全局时间帧可用性的效果。第9章研究了进程标识知识的影响。第8章讨论了进程终止条件的影响。第三部分研究了进程故障的影响。

（3）给出了最新分布式算法的研究进展，算法的复杂度证明和分析。

在不能详尽讨论某一主题时，可参见给定的相关科学文献。本书所收集资料分为三部分：协议，基本算法和容错。

36

1. 第一部分：协议

这部分讨论计算机通信网实现中所用的通信协议，引入后一部分所需的技术。

第2章引入了以后大部分章节中所用的模型。模型既有一般性又有严密性，即，不但适合于算法的设计和验证，而且也可用于证明不可能性的结果。它基于转移系统的表示，对于这些模型，能容易地给出安全性和活动性的证明规则。此外，还引入了将因果关系表示成一个计算中事件的偏序的方法，并且定义了逻辑时钟。

第3章讨论了两个节点间消息转移的问题。首先提出了单链上包交换协议组，并给出了正确性证明（源于Schoone）。同时，论述了Fletcher和Watson提出的协议，其正确性依赖于正确地使用计时器。这一协议的论述方法表明，验证方法是如何应用到基于定时器的协议中。

第4章讨论了计算机网络中的路由问题。首先给出关于路由的一般性理论和由Toueg提出的有关路由表计算的一种算法。其次，论述了Tajibnapis提出的变更算法（Netchange algorithm），给出了Lamport提出的该算法的正确性证明。本章以压缩路由算法结束，其中包括区间路由算法和前缀路由算法。这些算法统称为*压缩路由算法*（compact routing algorithm），因为它们在网络的每一节点只需少量的存储空间。

第5章讨论了包交换的计算机网络中避免存储转发死锁的一些策略，作为关于计算机网络协议讨论的结束。利用网络中节点的缓冲区，基于定义一些无回路的有向图来设计这些策略。可以看到，如何在各节点仅用适量的缓冲区来构造这样的一个图。

2. 第二部分：基本算法

这部分提出了许多算法上的“积木块”，它们可用作分布式应用中的过程。同时对不同网络的计算能力进行了理论研究。

第6章定义了“波动算法”的表示方法。这是访问网络中所有节点的一种模式。波动算法可用于通过网络发布信息、使节点同步或者计算一个函数，这个函数依赖于分布在所有节点上的信息。正如将在后续章节讨论的那样，许多分布式控制问题可用非常一般的算法模式求解，其中波动算法就是其中的一种。此外，本章定义了分布式算法的时间复杂度，并研究了许多分布式深度优先搜索算法的时间和消息复杂度。

37

分布式系统中的根本问题是选举问题：即选举在以后计算中起着重要作用的某个进程。第7章研究这个问题。在环网中首次研究了该问题，环网中该问题的消息复杂度为（在N个处理器的环上$\Theta(N \log N)$个消息）。一般的网络中也研究该问题并给出一些构造方法，通过这些构造可以由波动和遍历算法得到选举算法。本章还讨论了Gallager等人提出的生成树构造方法。

分布式系统中的第二个根本问题是终止检测（termination detection）问题：辨别一个分布式计算是否已经完成（由进程自己完成）。第8章中讨论这个问题，并证明了解决该问题复杂度的一个下界，且详尽讨论了其他几种算法。本章还包括一些经典算法（如，Dijkstra、Feijen和Van Gasteren提出的算法，Dijkstra和Scholten提出的算法）。并再次给出了由波动算法构造该问题的算法。

第9章研究了系统的计算能力，这类系统不能由惟一标识辨别进程。Angluin表明，在这种情况下，许多计算不能由确定性算法实现。本章还引入了概率算法，并且研究这类算法可解决哪些问题。

第10章解释了系统中的进程是如何计算一个全局“图片”的，即系统状态的一个快照。这样的一个快照可用于决定计算的性质，例如，是否发生了死锁、或计算进行到什么程度。在出现错误时，快照还用于重新启动一个计算。

第11章研究了网络中方向性知识的可用性，并给出了计算这种知识的几种算法。

第12章研究了全局时间概念。定义了几种程度的同步。通过相当详细的算法，完全异步的系统可以模拟完全同步的系统。可见，关于同步的假设并不影响分布式系统可计算的功能
38 的集合。但是，它对许多问题的通信复杂度有影响。网络的同步机制越好，问题的算法复杂度越低。

3. 第三部分：容错

在实际的分布式系统中，不能忽视系统中某一部分出现故障的可能性。因此，如果某一部分失效，研究算法的行为就非常重要。在本书的最后一部分将讨论该问题；第13章简要介绍了该主题。

第14章研究了异步系统的容错。Ficher等人给出了一种结果；表明确定性的异步算法不能处理甚至是非常适度类型的故障，单个进程的损毁。本章将要表明可以处理较弱类型的错误，并且尽管损毁故障发生，有些问题还是可以解决的。相反，Bracha和Toueg的算法表明，随机异步系统能够处理大量的故障。在这种情况下，较之确定性算法，随机算法能为可靠系统（参见第9章）提供更多的可能性。

第15章研究了同步算法的容错问题。Lamport等提出的算法表明，确定的同步算法可以容忍重大故障。显而易见，与可靠系统（第12章）遇到的情况不同，同步系统比异步系统提供了更多容错的可能性。如果进程能够“签署”与其他进程的通信，就能容忍更大量的故障。因此，在不可靠的系统中实现同步比在可靠系统中实现同步要更加复杂。第15章最后一节将专门进行该问题的研究。

第16章研究了抽象机制的性质，称为故障检测器（failure detector）。通过该检测器，进程可以获得对损毁及正确进程的一个估计。我们将描述这种机制的实现，以及如何利用它们在错环境中实现规范。

关于可靠性的另一种方法，即自稳定算法，在17章中讨论。称一个算法是稳定的，如果不管它的初始配置，它最终都收敛到想要的行为状态。我们将研究稳定算法的一些理论，并

提出大量的稳定算法。这些算法包括若干图论算法，例如，深度优先搜索树的计算·(参见6.4节)，路由表的计算（参见第4章）。同时，提出了数据传输的稳定算法（参见第3章），这表明， 39
稳定算法可以实现整个计算机网络。

4. 附录

附录A解释了本书中用于表示分布式算法的一些符号。附录B给出了图论及其术语的一些背景。以参考文献和索引结束本书。

第一部分 协 议

第2章 模 型

在分布式算法研究中，通常要用几种不同的分布式信息处理模型。选择哪一种模型取决于问题所属的分布式计算领域以及算法的类型，否则会给出不可能性的证明。在本书中，尽管包括了广泛的分布式算法及其理论，但是本章我们尽可能尝试描述一种通用模型。

为了容许不可能性（某些任务的算法不存在性证明）的结果，模型必须精确。一种不可能性的结果是阐述系统中所有可能的算法。因此，模型必须足够精确，能够描述所有确认无疑的算法的相关性质。然而，计算模型要比描述某一特定的计算机系统或者程序设计语言要更详尽。有许多不同的计算系统，我们期望模型可以应用到一类相关系统，这类系统具有分布式系统的本质特性。最后，模型必须合理而简洁，因为在证明中，必须考虑模型的所有方面。总之，模型既要精确又要简明，能够兼顾到一类计算系统的各个方面。

分布式计算通常被认为是离散事件的集合，每一事件在配置（整个系统的状态）中是一原子变量。在2.1节的转移系统（transition system）的定义中利用了这种概念，并导致了可达性配置的概念和对由算法引起的执行集合的定义。如果每次转移只受到配置的部分（part）影响，或转移只对配置产生部分影响，这就使系统成为"分布式"系统。部分主要是指某个进程的（局部）状态（或者交互进程子集的局部状态）。

2.2节及2.3节讨论了2.1节所描述的模型的性质和结果。2.2节论述了给定的分布式算法所期望的性质的证明问题。2.3节讨论了一种非常重要的表示法，即，一次执行中，事件之间的因果关系。这种关系引出对执行上的等价关系的定义；计算是此关系下的一个等价类。定义了时钟，所提出的逻辑时钟作为本书讨论的第一个分布式算法。2.4节讨论了基本模型中没有的假设和表示法。

2.1 转移系统和算法

状态按照离散步骤（转移或事件）变化的系统，通常可以用转移系统的概念描述。在分布式算法研究中，总体上可将转移系统用于分布式系统，也可用于算法中协同工作的进程之间。因此，对于研究分布式算法，转移系统是一个重要的概念，在2.1.1节中定义。

在分布式系统中，转移只影响到配置（系统的全局状态）的一部分。每一配置自身是一个元组，每一转移只涉及到元组中的一部分。配置的分量包括每一进程的状态。对于配置的精确描述，不同的分布式系统描述不同，取决于进程之间的通信模式。

分布式系统中的进程通信，既可通过访问共享变量（shared variable），又可通过消息传递（message passing）来进行。我们将采取更加严格的方式，只考虑进程通过信息交换进行通信的分布式系统。第17章讨论进程通过共享变量进行通信的分布式系统。对用共享变量进行通

信感兴趣的读者，可参阅Dijkstra的标志性论文[Dij68]，或者[OG76] Owicki和Gries论文；最近的论述参见Attiya和Welch [AW98]。

分布式系统中的消息可以通过同步（synchronously）或者异步（asynchronously）的方式传递。本书着重强调异步消息传递的分布式系统算法。为此，可将同步消息传递看作是异步消息传递的特例。正如Charron-Bost等人[CBMT96]表明的那样。2.1.2节精确地描述了异步消
44 息传递模型；2.1.3节中将此模型改造成适合于采用同步消息传递的系统。

2.1.1 转移系统

转移系统由系统中所有可能的状态集组成，系统可以从这个状态集以及允许系统开始的状态子集转移（“移动”）。为避免进程的状态和整个算法的状态（“全局状态”）相混淆，从现在开始，我们称后者为配置（configuration）。

定义2.1　转移系统是一个三元组（triple）$S = (C, \rightarrow, I)$，其中C是配置集（set of configurations），$\rightarrow$是C上的二元转移关系（binary transition relation），I是初始配置（initial configuration）C的一个子集（subset）。

转移关系是$C \times C$的一个子集。用更便利的表示法：$\gamma \rightarrow \delta$，而不用$(\gamma, \delta) \in \rightarrow$表示。

定义2.2　设$S=(C, \rightarrow, I)$是转移系统。S的一次执行是一个最大序列（maximal sequence）$E = (\gamma_0, \gamma_1, \gamma_2, \cdots)$，其中，$\gamma_0 \in I$，并且对所有$i \geqslant 0$，$\gamma_i \rightarrow \gamma_{i+1}$。

终止配置表示一个配置γ，在这个配置中，不存在δ满足条件：$\gamma \rightarrow \delta$。注意，对所有$i$，具有$\gamma_i \rightarrow \gamma_{i+1}$的序列$E = (\gamma_0, \gamma_1, \gamma_2, \cdots)$，如果它是无限的，或者以终止配置结束，则它是最大的。

定义2.3　如果对所有$0 \leqslant i < k$，存在序列$\gamma = \gamma_0, \gamma_1, \gamma_2, \cdots \gamma_k = \delta$且$\gamma_i \rightarrow \gamma_{i+1}$，则称配置$\delta$是由$\gamma$可达的（reachable），记为$\gamma \rightsquigarrow \delta$。如果配置$\delta$从初始配置是可达的，则称它是可达的。

2.1.2 异步消息传递系统

分布式系统由进程集和通信子系统组成。每一进程自身是一转移系统，且它能与通信子系统进行交互。为避免整个分布式系统的属性和单个进程的属性混淆，我们使用下列约定。术语“转移”和“配置”用于表示整个系统中的属性。术语“事件”和“状态”用于表示进
45 程的属性。为了与通信系统交互，进程不仅有普通事件（称为内部事件），而且有发送事件和接收事件，在其中消息被产生和消耗。设$\mathcal{M}$是可能消息的集合。用$\mathbb{M}(\mathcal{M})$表示来自$\mathcal{M}$中元素的多集的集合。

定义2.4　定义进程的局部算法为五元组$(Z, I, \vdash^{\mathrm{i}}, \vdash^{\mathrm{s}}, \vdash^{\mathrm{r}})$，其中$Z$表示状态集，$I$表示初始状态$Z$的子集，$\vdash^{\mathrm{i}}$表示$Z \times Z$上的关系，$\vdash^{\mathrm{s}}$和$\vdash^{\mathrm{r}}$表示$Z \times \mathcal{M} \times Z$上的关系，$Z$上的二元关系$\vdash$定义如下：

$$c \vdash d \iff (c, d) \in \vdash^{\mathrm{i}} \vee \exists m \in \mathcal{M}\,((c, m, d) \in \vdash^{\mathrm{s}} \cup \vdash^{\mathrm{r}}).$$

关系$\vdash^{\mathrm{i}}$，$\vdash^{\mathrm{s}}$和$\vdash^{\mathrm{r}}$分别表示关于内部事件、发送事件和接收事件的状态转移。以下，用p，q，r，p_1，p_2等表示进程。用$\mathbb{P}$表示系统中进程的集合。定义2.4作为进程的理论模型；当然，本书并不用列举进程的状态和事件来描述算法。而是用便利的伪随机代码（参见附录A）描述。进程的执行就是转移系统$(Z, \vdash, I)$的执行。

然而，我们对整个系统的执行感兴趣，在这样的执行中，进程的执行通过通信子系统来协调进行。为描述这种协调关系，把分布式系统定义为转移系统，其中的配置集、转移关系以及初始状态由进程的相应分量来构造。

定义2.5 定义进程集$\mathbb{P}=\{p_1, \ldots, p_N\}$的分布式算法为局部算法的集合，$\mathbb{P}$中的每一进程代表一个局部算法。

分布式算法的行为可用如下转移系统描述。配置由每一进程的状态和传输中的消息集组成；转移是进程中的事件，不仅影响到进程的状态，而且影响到消息集（或被消息集影响）；进程在初始状态且消息集为空时的配置为初始配置。

定义2.6 在异步通信条件下，由进程$p_1, \ldots, p_N$的分布式算法（进程p_i的局部算法为$(Z_{p_i}, I_{p_i}, \vdash^{i}_{p_i}, \vdash^{s}_{p_i}$, 和$\vdash^{r}_{p_i})$所导出的转移系统定义为$S=(\mathcal{C}, \rightarrow, \mathcal{I})$，其中

(1) $\mathcal{C}=\{(c_{p_1}, \ldots, c_{p_N}, M): (\forall p \in \mathbb{P}: c_p \in Z_p)$ 且$M \in \mathbb{M}(\mathcal{M})\}$。 46

(2) $\rightarrow=(\cup_{p\in\mathbb{P}} \rightarrow_p)$，其中$\rightarrow_p$表示与进程$p$的状态变化相对应的转移；$\rightarrow_{p_i}$表示序偶的集合

$$\left(c_{p_1}, \ldots, c_{p_i}, \ldots, c_{p_N}, M_1\right), \left(c_{p_1}, \ldots, c'_{p_i}, \ldots, c_{p_N}, M_2\right)$$

下列三个条件之一成立：

- $(c_{p_i}, c'_{p_i}) \in \vdash^{i}_{p_i}$ 且$M_1=M_2$；
- 对某些$m \in \mathcal{M}$，$(c_{p_i}, m, c'_{p_i}) \in \vdash^{s}_{p_i}$ 且$M_2=M_1 \cup \{m\}$；
- 对某些$m \in \mathcal{M}$，$(c_{p_i}, m, c'_{p_i}) \in \vdash^{r}_{p_i}$ 且$M_1=M_2 \cup \{m\}$；；

(3) $\mathcal{I}=\{(c_{p_1}, \ldots, c_{p_N}, M): (\forall p \in \mathbb{P}: c_p \in I_p) \wedge M=\varnothing\}$。

分布式算法的一次执行就是导出的转移系统的一次执行。一次执行中的事件对于下列记号是显式执行的。称序偶$(c, d) \in \vdash^{i}_{p}$为进程$p$的（可能的）内部事件。$\vdash^{s}_{p}$和$\vdash^{r}_{p}$中的三元组称为进程的发送和接收事件。

- 在配置$\gamma=(c_{p_1}, \ldots, c_p, \ldots, c_{p_N}, M)$中，如果$c_p=c$，则称$p$中的内部事件$e=(c, d)$是可应用的（applicable）。在这种情况下，定义$e(\gamma)$为$(c_{p_1}, \ldots, d, \ldots, c_{p_N}, M)$。
- 在配置$\gamma=(c_{p_1}, \ldots, c_p, \ldots, c_{p_N}, M)$中，如果$c_p=c$，则称$p$中的发送事件$e=(c, m, d)$是可应用的。在这种情况下，定义$e(\gamma)$为$(c_{p_1}, \ldots, d, \ldots, c_{p_N}, M \cup\{m\})$。
- 在配置$\gamma=(c_{p_1}, \ldots, c_p, \ldots, c_{p_N}, M)$中，如果$c_p=c$且$m \in M$，则称$p$中的接收事件$e=(c, m, d)$是可应用的。在这种情况下，定义$e(\gamma)$为$(c_{p_1}, \ldots, d, \ldots, c_{p_N}, M \backslash\{m\})$。

假设对于每一消息，存在惟一进程接收该消息。称此进程为消息的目的地。

2.1.3 同步消息传递系统

如果一个发送事件及对应的接收事件协同形成系统的一个转移，则称消息传递是同步的。即，在消息的目的地准备接收消息时，进程才能发送消息。因此，有两种类型的系统转移：一种是进程内部状态的变化，另一种是两个进程之间的组合通信事件。

定义2.7 在同步通信条件下，由进程$p_1, \ldots, p_N$的分布式算法所导出的转移系统定义为$S=(C, \rightarrow, \mathcal{I})$，其中 47

(1) $C=\{(c_{p_1}, \ldots, c_{p_N}): \forall p \in \mathbb{P}: c_p \in Z_p\}$。

(2) $\rightarrow=(\cup_{p\in\mathbb{P}} \rightarrow_p) \cup (\cup_{p,q\in\mathbb{P}: p\neq q} \rightarrow_{pq})$，其中

- $\rightarrow_{p_i}$表示序偶

  $\left(c_{p_1},\ldots,c_{p_i},\ldots,c_{p_N}\right),\left(c_{p_1},\ldots,c'_{p_i},\ldots,c_{p_N}\right)$ 集合，其中 $\left(c_{p_i},c'_{p_i}\right)\in\vdash^{i}_{p_i}$;

- $\rightarrow_{p_ip_j}$表示序偶

  $\left(\ldots,c_{p_i},\ldots,c_{p_j},\ldots\right),\left(\ldots,c'_{p_i},\ldots,c'_{p_j}\ldots\right)$ 集合，

  其中存在消息$m\in\mathcal{M}$，满足

  $(c_{p_i},\ m,\ c'_{p_i})\in\vdash^{s}_{p_i}$且 $(c_{p_j},\ m,\ c'_{p_j})\in\vdash^{r}_{p_j}$。

（3）$\mathcal{I}=\{(c_{p_1},\ldots,c_{p_N}):(\forall p\in\mathbb{P}:C_p\in I_p)\}$。

一些分布式系统允许混合形式的通信；系统中的进程以同步方式及异步方式采用通信原语进行消息传递。给定上述定义的两种模型，不难为这种类型分布式系统设计一种形式模型。这种系统的配置包括进程的状态和传输中的消息集（即，同步消息）。转移包括所有出现在定义2.6和定义2.7中的转移类型。

同步及其对算法的影响

如前所述，可以把多种用途的同步消息传递看作是异步消息传递的特例。对于那些每次发送事件又紧跟着对应的接收事件[CBMT96]的执行，执行集限制在同步消息传递的情况内。因此，我们认为异步消息传递是比同步消息传递更具通用性的模型，并将主要为这一通用情况开发算法。

然而，值得注意的是，为异步消息传递系统所开发的算法是否能在同步消息传递系统中执行。简化的通信子系统的不确定性必须通过增加进程的不确定性达到平衡，否则就会导致死锁。

用一个基本例子来说明，两个进程相互发送消息的情形。在异步情况下，每一进程必须首先发送一条消息，随后接收其他进程的消息。在进程的发送和接收过程之间，消息被暂时
48 放在通信子系统的缓冲区中。同步的情况，不可能存在这样的缓冲区。如果两个进程必须在接收消息之前发送它们各自的消息，就不会发生转移。在同步情况下，进程必须接到另一进程的消息，才能发送自己的消息。不用说，如果两个进程必须接到消息，才能发送它们各自的消息，也不会发生转移。

同步系统中，只有当满足下列两个条件之一，才能进行信息交换。

（1）预先可以确定，哪个进程先发送，哪个进程先接收。在许多情况下，不可能预先做出选择，因为这需要两个进程执行不同的局部算法。

（2）进程具有不确定的选择，要么先发送后接收，要么先接收后发送。在每次执行中，为每一进程选择一种可能的执行次序，即打破进程通信子系统的对称性。

当我们为异步消息传递设计一种算法，并阐明该算法同样可适用于同步消息传递，这种确定性（总是可能的）的增加被隐含地做了假设。

2.1.4 公平性

在某些情况下，必须对系统的行为限制到所谓的公平执行。公平性条件排除了某些事件总是（常常无限）可用但从发生转移的情况（因为它们会随其他可应用事件继续执行）。

定义2.8　在无限多连续配置中，且该配置没有发生在执行过程中，如果没有事件是可应

用的，则称执行是弱公平的（weakly fair）。在无限多配置中，且该配置没有发生在执行过程中，如果没有事件是可应用的，则称执行是强公平的（strongly fair）。

在明确的形式模型中，可能会将公平性条件结合进去。正如Manna和Pnueli所作的那样[MP88]。本书中的大多数算法不依赖于这些条件；因此我们的算法将不结合这些条件。但是在它们应用到某一特定算法和问题时，我们会明确地阐明这些条件。同时对在分布式模型中 49
包含公平性假设是否合理展开辩论。在算法中不做公平性假设，使得算法设计不依赖于这些假设。对有关公平性假设的某些复杂问题的讨论可参见[Fra86]。

2.2 转移系统性质的证明

给定某一问题的分布式算法，需要证明算法是问题的一个正确解。问题明确了所求算法必有的性质；问题的解必须具有这些性质。验证分布式算法的问题已经受到广泛的关注，并且大量的文献讨论形式化验证方法；参见[CM88, Fra86, Kel76, MP88]。本节中，我们用一些简单却常用的方法证明分布式算法的正确性。这些方法依赖于转移系统的定义。

可将分布式算法其有的性质分为两类：*安全性要求*和*活动性要求*（liveness requirement）。算法的安全性要求是指，对于系统的每一次执行，以及那次执行达到的每个配置，所应有的性质必须保持。算法的活动性要求是指，对于系统的每一次执行，以及那次执行达到的某些配置，所应有的性质必须保持。关于构成安全性和活动性的形式描述参见Alpern和Schneider[AS85]。文献表明，每种可能的规范（执行集的性质）都可以看成是安全性和活动性的结合。

这些要求也可能以弱化的形式出现。例如，它们必须满足可能执行集上的某种固定概率分布。其他对于算法的要求，可能包括某一给定知识的运用（参见2.4.4节），对某些进程的故障具有弹性（参见第三部分），以及进程是同等的（参见第9章）等。

本节中所描述的验证方法，基于一次执行中所达配置的*断言*（assertion）的真值。这种方法称为*断言式验证方法*。一个断言是配置集上的一元关系，即，对于配置的一个子集谓词为真，其他为假。 50

2.2.1 安全性

算法的安全性是指："在算法每次执行的每次配置中，*断言P为真*"。用非形式化的语言，可描述为"断言P总是为真（true）"。按照下列所给定义，证明断言P总是为真的基本技术，是要证明P是*不变式*（invariant）。记号$P(\gamma)$是布尔表达式，如果P在γ中成立，其值为真；否则为假，其中γ表示配置。

这些定义与给定的转移系统$S = (C, \rightarrow, \mathcal{I})$有关。$\{P\} \rightarrow \{Q\}$表示，对于$S$的每一次转移$\gamma \rightarrow \delta$，如果$P(\gamma)$，则$Q(\delta)$。$\{P\} \rightarrow \{Q\}$含义是，如果$P$在任一转移之前成立，那么$Q$在转移之后成立。

定义2.9 *断言P是S的一个不变式，如果*

（1）对所有$\gamma \in \mathcal{I}$，$P(\gamma)$成立，且

（2）$\{P\} \rightarrow \{P\}$。

定义表明，在每次初始配置中，不变式成立。并在每次转移过程中保持不变。由此可得，在每一可达的配置中，不变式成立。表述成如下定理。

定理2.10　如果P是S的一个不变式，那么对于S的每次执行的每一配置，P成立。

证明。设$E = (\gamma_0, \gamma_1, \gamma_2, \cdots)$是$S$的一次执行。用归纳法证明，对于每个$i$，$P(\gamma_i)$成立。首先，因为$\gamma_0 \in \mathcal{I}$，且由定义2.9的第一个条件，$P(\gamma_0)$成立。其次，假设$P(\gamma_i)$成立，且$\gamma_i \rightarrow \gamma_{i+1}$是出现在$E$中的一次转移。由定义2.9的第二个条件，$P(\gamma_{i+1})$成立。定理证毕。 □

相反地，并不是在每次执行的每一配置都成立的断言就是不变式（参见练习2.2）。因此，并不是每一安全性质都能利用定理2.10来证明。然而，不变式蕴含每次断言总是为真；可用以下定理证明断言总是真的。（需要注意的是，当用此定理时，要找一个合适的不变式Q常常是非常困难的。）

定理2.11　设Q是S的不变式，假设$Q \Rightarrow P$（对于每一$\gamma \in C$）。那么P在S的每次执行的每一配置中成立。

51

证明。由定理2.10，在每个配置中，Q成立。因为Q蕴含P，在每个配置中，P也成立。□

有时首先证明弱不变式是有用的，接着利用这一结果来证明强不变式。下列的定义和定理证明了如何使一个不变式成为强不变式。

定义2.12　设S是转移系统，P，Q是断言。称P为Q-导出的，如果

（1）对所有$\gamma \in \mathcal{I}$，$Q(\gamma) \Rightarrow P(\gamma)$；且

（2）$\{Q \wedge P\} \rightarrow \{Q \Rightarrow P\}$。

定理2.13　如果Q是不变式，P是Q-导出的，那么$Q \wedge P$是不变式。

证明。按照2.9的定义，要证

（1）对所有$\gamma \in \mathcal{I}$，$Q(\gamma) \wedge P(\gamma)$；且

（2）$\{Q \wedge P\} \rightarrow \{Q \wedge P\}$。

因为Q是不变式，对所有$\gamma \in \mathcal{I}$，$Q(\gamma)$成立，且对所有$\gamma \in \mathcal{I}$，$Q(\gamma) \Rightarrow P(\gamma)$，对所有$\gamma \in \mathcal{I}$，$P(\gamma)$成立。因此，对所有$\gamma \in \mathcal{I}$，$Q(\gamma) \wedge P(\gamma)$成立。

假设$\gamma \rightarrow \delta$且$Q(\gamma) \wedge P(\gamma)$。因为$Q$是不变式，$Q(\delta)$成立，又因为$\{Q \wedge P\} \rightarrow \{Q \Rightarrow P\}$，$Q(\delta) \Rightarrow P(\delta)$，由此，$P(\delta)$成立。于是，$Q(\delta) \wedge P(\delta)$成立。 □

在3.1节给出了本节安全性例子的证明。

2.2.2　活动性

算法的活动性是指："在算法每次执行的某些配置中，断言P为真"。用非形式化的语言，即为"断言P最终为真"。证明断言P最终为真的基本技术是*范函数*和*无死锁性*（deadlock-freeness）或者*正常终止*（proper termination）。算法中使用的简单技术是只允许固定有限的执行。

设S表示转移系统，P是一个谓词。定义**term**为谓词，它在所有最终配置中为真，在所有非最终配置中为假。我们首先考虑达到最终配置的执行。通常，人们不希望这样的配置，对于该配置，不能达到"目标"P，这就是人们所说的死锁情形。另一方面，如果目标P已经达到，则认为终止；这就是人们所说的正常终止的情形。

52

定义2.14　如果谓词（$\mathbf{term} \Rightarrow P$）在$S$中总是为真，则系统$S$正常终止（或者无死锁）。

范函数依靠*良基集*（well-founded）的数学概念。这是一个具有序$<$的集合，其中不存在无穷递减的序列。

定义2.15　称一个偏序集$(W, <)$是良基的，如果不存在无穷递减序列

$$w_1 > w_2 > w_3 \cdots$$

本书中所用良基集的例子是具有通常序的自然数和具有字典序的自然数的n-元组（参见4.3节）。良基集中不存在无穷递减序列的性质可用于证明断言P最终为真。要证明这一点，就要证明，存在一个从C到良基集W的函数f，满足在每次转移过程中，f的值递减，或者P变为真。

定义2.16 *给定转移系统S和断言P。称从C到良基集W的函数f为范函数（关于P），如果对于每次转移$\gamma \to \delta$，$f(\gamma) > f(\delta)$或者$P(\delta)$成立。*

定理2.17 *给定转移系统S和断言P。如果S正常终止，且范函数f（关于P）存在，那么在S每次执行的某些配置中P为真。*

证明。设$E = (\gamma_0, \gamma_1, \gamma_2, \cdots)$表示$S$的一次执行。如果$E$是有限的，它的最终配置是一个终止配置，并且在$S$中当谓词**term** $\Rightarrow P$总是为真时，P在此配置中成立。如果E是无限的，设E'是E的最长前缀，其中不存在P为真的配置，设s为所有出现在E'中，所有配置γ_i构成的序列$(f(\gamma_0), f(\gamma_1), \cdots)$。由$E'$的选择和$f$的性质，$s$是递减序列，因此由$W$的良基性，$s$是有限的。这也表明，$E'$是$E$的一个有限前缀$(\gamma_0, \gamma_1, \cdots, \gamma_k)$。由$E'$的选择，$P(\gamma_{k+1})$成立。 □

如果假定公平性质，由弱前提（定理2.17）就可得出P最终为真；范函数的值不必随每次转移递减。公平性假设可用于证明，无限次执行包含某种类型的转移是无限的。进而能够证明f永不递增，但却随着这种类型的每次转移递减。 53

在某些情况下，我们将利用下述结果，它是定理2.17的特例。

定理2.18 *如果S正常终止，且存在数K，满足每次执行至多包含K次转移，那么在每次执行的某些配置中，P为真。*

2.3 事件的因果序和逻辑时钟

将执行看作转移的序列很自然地引出执行中的时间表示。如果a在序列中出现在b之前，就称转移a出现在转移b之前。对于执行$E = (\gamma_0, \gamma_1, \cdots)$，定义相关的事件序列$\bar{E} = (e_0, e_1, \cdots)$，其中$e_i$表示事件，通过此事件，配置从$\gamma_i$变到$\gamma_{i+1}$。可见，用这种方法，每次执行定义了惟一的一个事件序列。可以用时空图（space-time diagram）来可视化执行过程。图2-1给出了一个例子。在图中，水平线表示进程，进程线上的一个点表示一个事件，它的位置表示事件发生的地方。如果消息m在事件s中发送，在事件r中接收，就从s到r画一个箭头，在这种情况下，称事件s和r是对应事件。

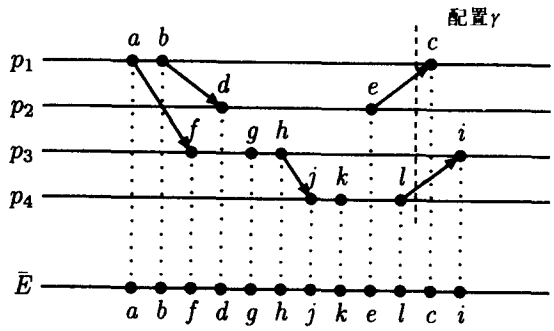


图2-1 时空图的一个例子

正如在2.3.1节可看到的，分布式执行中的事件有时可以相互交换，而不影响以后执行的
54 配置。因此，把时间作为执行中事件的全序，并不适合于分布式执行。因此引入了因果相关表示法。在事件的重新排列之下，2.3.2节研究了执行的等价性。在2.3.3节讨论了如何定义用于测量因果相关性的时钟（而不是时间），提出了Lamport的逻辑时钟，给出了这种时钟的一个例子。

2.3.1 事件的独立性和相关性

分布式系统的转移只影响到（或者受到影响）配置的一部分。由此可见，影响配置不同部分的两个连续事件，是独立的，并且可以以相反的次序出现。对于具有异步消息传递的系统，可以表述成下面的定理。

定理2.19 *设γ是分布式系统的一个配置（具有异步消息传递），e_p和e_q分别是不同进程p和q中的事件，它们在γ中是可应用的。那么，e_p在$e_q(\gamma)$中是可应用的，e_q在$e_p(\gamma)$中是可用的，并且$e_p(e_q(\gamma)) = e_q(e_p(\gamma))$。*

*证明。*为避免分析发送事件、接收事件或者内部事件的一种，将每一事件表示成一致的形式（c，x，y，d）。这里，c和d分别表示事件前、后的进程状态，x是事件所接收的消息集，y是事件所发送的消息集。因此，内部事件（c，d）可表示为（c，$\varnothing$，$\varnothing$，d）；发送事件（c，m，d）可表示为（c，$\varnothing$，$\{m\}$，d），接收事件（c，m，d）可表示为（c，$\{m\}$，$\varnothing$，d）。基于这种表示法，如果$c_p = c$且$x \subseteq M$，则进程p中的事件$e =(c, x, y, d)$在配置$\gamma =(c_{p_1}, \cdots, c_p, \cdots, c_{p_N}, M)$中是可应用的。在这种情况下

$$e(\gamma) = (c_{p_1},\ldots,d,\ldots,(M \setminus x)\cup y)。$$

现假设$e_p = (b_p, x_p, y_p, d_p)$和$e_q = (b_q, x_q, y_q, d_q)$在

$$\gamma = (\ldots,c_p,\ldots,c_q,\ldots,M)$$

中是可应用的。即，$c_p = b_p$，$c_q = b_q$，$x_p \subseteq M$，和$x_q \subseteq M$。重要的观察是x_p和x_q不相交；x_p
55 中的信息（如果有）目的地为p，而x_q中的信息（如果有）目的地为q。

记$\gamma_p = e_p(\gamma)$，注意，

$$\gamma_p = (\ldots,d_p,\ldots,c_q,\ldots,(M \setminus x_p)\cup y_p)。$$

因为$x_q \subseteq M$且$x_q \cap x_p = \varnothing$，由此可得，$x_q \subseteq (M \setminus x_p \cup y_p)$，因此$e_q$在$\gamma_p$中是可应用的。记$\gamma_{pq} = e_q(\gamma_p)$，且有

$$\gamma_{pq} = (\ldots,d_p,\ldots,d_q,\ldots,((M \setminus x_p\cup y_p) \setminus x_q)\cup y_q)。$$

由对称性，可以得出，e_p在$\gamma_q = e_q(\gamma)$中是可应用的。记$\gamma_{qp} = e_p(\gamma_q)$，注意

$$\gamma_{qp} = (\ldots,d_p,\ldots,d_q,\ldots,((M \setminus x_q\cup y_q) \setminus x_p\cup y_p))。$$

因为M是消息的多集，$x_p \subseteq M$，且$x_q \subseteq M$，　　$\subseteq$

$$((M \setminus x_p\cup y_p) \setminus x_q\cup y_q) = ((M \setminus x_q\cup y_q) \setminus x_p\cup y_p)，$$

因此$\gamma_{pq} = \gamma_{qp}$。　□

设e_p和e_q是出现在一次执行中的两个连续事件，即对于某些γ，执行包括子序列

$$\ldots,\gamma,e_p(\gamma),e_q(e_p(\gamma)),\ldots$$

把定理2.19的前提应用到这些事件，除去下列两种情况。

（1）$p=q$；或者

（2）e_p是发送事件，e_q是对应的接收事件。

的确，定理明确地表明p和q必须不同，且如果e_q接收到e_p中所发送的消息，接收事件就不能应用到e_p的初始配置中，正如所要求的。因此，如果这两种说法其一为真，那么事件就不能以相反次序出现；否则，若它们以相反的次序出现，就会导致同样配置。注意，从全局的角度来看，转移是不能交换的，这是因为（定理2.19中的表示）从γ_p到γ_{pq}的转移不同于从γ到γ_q的转移。然而，从进程的角度来看，这些事件是不能区别的。

某一对事件不能交换的事实表明，这两个事件之间存在因果关系。这个关系可被扩展成执行事件集上的偏序，称为执行的因果序。

定义2.20 设E是一次执行。称因果序的关系$\prec$是执行事件上满足下列条件的最小关系。 56

（1）如果e和f是同一进程中的不同事件，e在f之前出现，那么$e \prec f$。

（2）如果s是发送事件，r是对应的接收事件，那么，$s \prec r$。

（3）$\prec$是可传递的。

用$a \preceq b$来表示（$a \prec b \vee a=b$）。因为$\preceq$是偏（partial）序，则可能存在事件a，b，使得$a \preceq b$和$b \preceq a$都不成立。称这些事件是并发的，记为$a \parallel b$。在图2-1中，$b \parallel f$，$d \parallel i$等并发。Lamport[Lam78]首先定义了因果序。因果序在分布式算法的推理中起着重要作用。$\prec$的定义蕴含在因果相关的事件之间存在因果链。用此表示法，$a \prec b$表示存在序列$a=e_0, e_1, \cdots, e_k=b$，链中的每一连续事件对，或满足定义2.20中的（1），或满足定义中的（2）。可以选择因果链，使得满足条件（1）的每一对事件都是它们所出现的进程中的连续事件，即，它们之间没有其他事件。在图2-1中，事件a和l之间的因果链是序列a, f, g, h, j, k, l。

2.3.2 执行的等价性：计算

只要与因果序一致，执行中的事件可以任意重新排列而不影响执行的结果。事件的重排过程引出了不同的配置序列，但是，可以认为这个执行与原执行等价。

设$f=(f_0, f_1, f_2, \cdots)$表示事件序列。这个序列是与执行$F=(\delta_0, \delta_1, \delta_2, \cdots)$有关的事件序列，如果对于每一个$i$，$f_i$在$\delta_i$中是可应用的，并且$f_i(\delta_i)=\delta_{i+1}$）。如果情况的确如此，称$F$为序列$f$的隐执行（implied execution）。我们想让F由f惟一确定。但情况并非总是如此；如果对某些进程p，p中的所有事件都不在f中，那么，p的状态可以是任意初始状态。然而，如果f至少包含p中的一个事件，那么p中的第一个事件（c, x, y, d）定义了p的初始状态c。因而，如果f至少包含每一进程中的一个事件，且δ_0惟一定义，也就惟一地定义了整个执行。

设$E=(\gamma_0, \gamma_1, \cdots)$是与事件序列$\bar{E}=(e_0, e_1, \cdots)$有关的一次执行并假设$f$是$\bar{E}$的一个排列。这表明，存在自然数（或者，如果$E$是$k$个事件的有限执行，为集合$\{0, \cdots, k-1\}$）的 57
一个排列σ，满足$f_i=e_{\sigma(i)}$。如果$f_i \preceq f_j$蕴含$i \leqslant j$，则E中事件的排列$(f_0, f_1, f_2, \cdots)$与因果序一致，即因果上有前后关系的事件在序列中保持同样的顺序关系。

定理2.21 设$f=(f_0, f_1, f_2, \cdots)$表示$E$中的事件排列，它与$E$中因果序一致。那么$f$定义了惟一的执行$F$，它开始于$E$的初始配置。$F$，$E$中的事件同样多，并且如果$E$是有限的，则$F$的最终配置和$E$的最终配置相同。

证明。逐步构造F的配置，要构造δ_{i+1}，就要证明f_i在δ_i中是可应用的。取$\delta_0 = \gamma_0$。

假设对于所有$j < i$，f_j在配置δ_j中是可应用的，且$\delta_{j+1} = f_j(\delta_j)$。令$\delta_i = (c_{p_1}, \ldots, c_{p_N}, M)$，设$f_i = (c, x, y, d)$是进程$p$中的事件；如果$c_p = c$且$x \subseteq M$，则事件$f_i$在$\delta_i$中是可应用的。

为证明$c_p = c$，我们讨论两种情况。在这两种情况下，可注意到，E的因果序完全确定了进程p中事件的次序；这就意味着进程p中的事件出现的次序与f和$\bar{E}$中的事件次序完全相同。

情况1：f_i是f中进程p的第一个事件；则c_p是p的初始状态。然而f_i也是$\bar{E}$中进程p的第一个事件，这意味着c是p的初始状态。因此，$c = c_p$。

情况2：f_i不是f中进程p的第一个事件；令在f_i之前，f中p的最后一个事件为$f_i' = (c', x', y', d')$，那么$c_p = d'$；然而f_i'也是$\bar{E}$中进程p的最后一个事件（f_i之前），这意味着$c = d'$。因此，$c = c_p$。

为证明$x \subseteq M$，我们注意在f和$\bar{E}$中的发送和接收事件以同样次序出现。如果f_i不是接收事件，则$x = \varnothing$和$x \subseteq M$平凡成立。如果f_i是接收事件，设f_j是对应的发送事件。因为$f_j \prec f_i$，$j < i$成立，即发送事件在f中的f_i之前；因此，$x \subseteq M$。

我们已证明，对于每一i，f_i在δ_i中是可应用的，可取δ_{i+1}为$f_i(\delta_i)$。最后必须证明，如果E是有限的，F和E的最终配置相同。设γ_k是E中的最后配置。如果$\bar{E}$不含p中的事件，γ_k中p的状态等于它的初始状态。由于f也不含p中的事件，δ_k中p的状态也等于它的初始状态，因此，δ_k中p的状态等于它在γ_k中的状态。否则，γ_k中p的状态就是$\bar{E}$中进程p的最后一个事件之后的状态；这也是f中进程p中的最后事件，因此，这也是δ_k中进程p的状态。

58　在γ_k中，传输中的消息恰好是$\bar{E}$中的发送事件与对应的接收事件不相匹配的消息。但是$\bar{E}$和f包含同样的事件集，在F的最终配置中，传输的是相同消息。　□

执行F和E具有相同的事件集，且这些事件的因果序相同。因此，$\bar{E}$是F中事件的一个排列，且因果序相同。如果应用定理2.21中条件，则称E和F为**等价执行**，表示为$E \sim F$。

图2-2表示执行的时间图，该图与图2-1等价。用“橡皮圈转换”[Mat89c]可得到等价的时间图。假设进程的时间轴可被压缩和被拉伸，只要保持消息的箭头“向右”，就可以把图2-1变成图2-2。

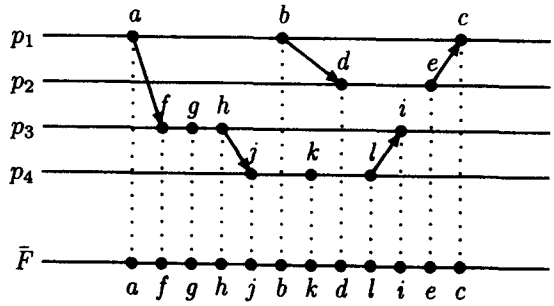


图2-2　与图2-1等价的时空图

尽管所描述的执行是等价的并包含同样多的事件，但是它们并不包含相同的配置集。图2-1中包含的一个配置（γ''），其中事件e中所发送的消息和事件l中所发送的消息同时传输。图2-2并不包含这样的配置，这是因为事件l中所发送的消息在事件e发生之前收到。

可以访问实际事件序列的全局观测器，能辨别出两个等价执行。即，能区别出是哪一个在执行。然而，进程不能辨别两个等价的执行，因为它们不可能区别出这两个等价的执行是

哪一个在执行。说明如下。假设必须决定，是否事件e中所发送的消息和事件l中所发送的消息同时传输。在其中的一个进程中有一布尔变量sim，如果消息同时传输，必须将该变量设为真；否则，该变量为假。在图2-1的最终配置中，sim为真。在图2-2的最终配置中，sim为假。 59
根据定理2.21，配置相等，这表明，所要求的对sim赋值是不可能的。

可以用事件集和这些事件的因果序刻画～下的等价类；称等价类为算法的计算。

定义2.22 *称分布式算法的一次计算为算法执行的等价类（～下）。*

谈论一次计算的配置毫无意义，这是因为不同的计算执行可能会产生不同的配置。因此讨论计算的事件集才有意义，因为计算的所有执行由相同事件集组成。也可以为计算定义事件因果序。如果执行是有限的，则称计算是有限的。如果计算是有限的，开始于同一配置（参见定理2.21）的一次计算的所有执行，则会终止于同一配置；这些配置称为计算的初始配置和终止配置。可以用属于它的事件偏序集来确定一次计算。

由偏序理论可知，对于一次计算中的一对并发事件，可能出现任意次序。

事实2.23 *设$(X, <)$是偏序，且$a, b \in X$，满足$b \not< a$。存在$<$的一个线性扩展$<_1$，满足$a <_1 b$。*

因此，如果a和b是计算C中的并发事件，则存在这次计算的执行E_a和E_b，满足a在E_a中早于b，b在E_b中早于a。执行中的进程没有任何手段来决定两个事件的哪一个先发生。

1. 同步消息传递

定理2.19也可用来说明具有同步消息传递的系统。在这样的系统中，如果两个连续事件影响不同的进程，则这两个事件是独立的。如以下定理所述。

定理2.24 *设γ是具有同步消息传递的分布式系统的配置，e_1是进程p和q的一次转移，e_2是进程r和s（不同于p、q）的一次转移，满足e_1、e_2在γ中都是可应用的，那么，e_1在$e_2(\gamma)$中* 60
是可应用的，e_2在$e_1(\gamma)$中是可应用的，并且$e_1(e_2(\gamma)) = e_2(e_1(\gamma))$。

定理的证明，与定理2.19的证明方法相同，留作习题2.9。同步系统因果关系的定义类似于定义2.20。感兴趣的读者可参见[CBMT96]。同步系统也有和定理2.21对应的定理。

2. 执行和计算

我们首先定义了执行（正如转移系统的运行），然后引入计算作为执行的等价类。这个次序反映了分布式系统中所发生的可操作的观点。许多其他的著作首先引入计算作为事件上的偏序集，然后引入执行，作为这些集的线性化。例如，Charron-Bost *et al*[CBMT96]在标准的消息序列图的形式化[RGG96]中就是这样做的。这两种方法是等价的。

2.3.3 逻辑时钟

与度量真正时间的物理时钟类似，在分布式计算中，可将时钟定义成表示因果关系。本节中，Θ表示有序集$(X, <)$上事件的函数。

定义2.25 *时钟是有序集$(X, <)$上事件集的函数Θ，满足*

$$a \prec b \Rightarrow \Theta(a) < \Theta(b)。$$

在本节的其余部分，讨论了关于时钟的一些例子。

（1）序列中的次序 在一次执行中，定义E为事件序列$(e_0, e_1, e_2, \cdots)$，设$\Theta_g(e_i) = i$。每个事件用它在事件序列中的位置进行标号。

可用此函数作为系统的全局观察器，按照事件出现的次序访问事件。然而，在系统内是不可能观察到这个次序的，换句话说，一个分布式算法不能计算出Θ_g。这是定理2.19的结
61 论；假设某些分布式算法存储了事件e_i的值$\Theta_g(e_i)=i$（满足定理的前提条件）。在一次等价执行中，这个事件与下一事件交换，就有了不同的Θ_g值，同一值i被存储在进程中。换句话说，Θ_g是定义在执行上，而不是定义在计算上。

（2）实时时钟　可对此模型进行扩充，为每一进程提供一个硬件时钟这是本章的主题。用这种方法，可记录每一事件发生的真实时间；所得的数值满足时钟定义。

具有实时时钟的分布式系统并不适合于定义2.6。因为时钟的物理性质与不同进程的改变同步。时间在所有进程中继续着，这会导致转移，这种转移改变所有进程的状态（即，时钟读数）。结果这些“全局转移”急剧地改变了模型的性质；事实上，如果假设实时时钟，定理2.19就不再成立。然而，具有实时时钟的分布式系统实际上已在使用，它们将在本书第3.2节，第12章和第15章中进行讨论。

（3）Lamport逻辑时钟　Lamport[Lam78]已提出一个时钟函数，它赋予事件a长度k，其中k表示最长事件序列的长度$(e_1,\cdots,e_k)$。满足下式:

$$e_1 \prec e_2 \prec \ldots \prec e_k = a \text{。}$$

的确，如果$a \prec b$，扩展此序列可得，$\Theta_L(a)<\Theta_L(b)$。可计算每一事件的Θ_L值，通过基于下列关系的分布式算法计算:

a）如果a是一个内部或者发送事件，a'是同一进程中的前一事件，那么$\Theta_L(a)=\Theta_L(a')+1$。

b）如果a是一个接收事件，a'是同一进程中的前一事件，b是与a对应的发送事件，那么$\Theta_L(a)=\max(\Theta_L(a'),\Theta_L(b))+1$。

在两种情况下，如果a是进程的第一个事件，则假定$\Theta_L(a')$为0。

为了用分布式算法计算时钟值，需要将进程p的最后一个事件的时钟值存储在变量Θ_p中
62 （初始化为0）。为了计算一个接收事件的时钟值，每条消息m包含事件e的时钟值Θ_m，且在事件e中发送该值。图2-3中的算法给出了Lamport的逻辑时钟。对于进程p中的事件e，$\Theta_L(e)$的值就是紧跟在e之后的Θ_p的值，即进程p发生状态变化的时刻。留作练习证明，这样定义的Θ_L是时钟。

还未明确在什么条件下，必须发送消息，进程的状态如何改变。时钟是加在分布式算法中用于对事件定序的一种辅助机制。

（4）向量时钟　时钟不仅能够表示因果序（定义2.25所要求的）而且能够表示并发性。如果并发事件用无法比较的时钟值标记，则用时钟来表示并发性。即定义2.25中的蕴含用等价替代，设

$$a \prec b \Leftrightarrow \Theta(a)<\Theta(b) \tag{2-1}$$

并发事件的存在表明，这样的时钟域（集X）是一个非全序集。

在Mattern的向量时钟[Mat89b]中，$X=\mathbb{N}^N$，即$\Theta_v(a)$是长为N的向量。长为n的向量自然地按照向量次序排列，定义如下:

$$(a_1,\ldots,a_n)<_v(b_1,\ldots,b_n) \Leftrightarrow \forall i(1<i<n): a_i < b_i \tag{2-2}$$

（向量序不同于习题2.5中定义的字典序；后者是全序。）定义时钟为$\Theta_v(a) = (a_1, \ldots, a_N)$，其中，$a_i$是进程$p_i$中满足$e \preceq a$的事件$e$的数目。正像Lamport时钟一样，分布式算法可以计算出这个函数。

Charron-Bost [CB89]已经证明不可能用更短的向量（如式(2-2)中所定义向量序）。如果将N个进程任意一次执行的事件映射到长为n的向量上，并满足式(2-1)，那么$n \geqslant N$。

```
var θ_p : integer      init 0 ;

(* An internal event *)
   θ_p := θ_p + 1 ;
   Change state

(* A send event *)
   θ_p := θ_p + 1 ;
   send ⟨ messg, θ_p ⟩ ; Change state

(* A receive event *)
   receive ⟨ messg, θ ⟩ ; θ_p := max(θ_p, θ) + 1 ;
   Change state
```

图2-3 LAMPORT的逻辑时钟算法

2.4 附加假设，复杂度

本章目前所作的定义为其余的章节打下了基础。所定义的模型为算法表示和验证、分布式问题解的不可能性证明提供了框架。在后续章节中，如果必要，可以附加假设和使用符号。本节讨论一些术语，它们是分布式算法文献中常用的。

2.4.1 网络拓扑结构

到目前为止，我们用当前传输中的消息，对分布式系统的通信子系统进行了模拟。此外，还假设每一消息只能被一个称为消息目的地的进程接收。一般而言，未必每一进程都能够发送消息到其他进程。而是，对于每一进程，定义它能发送消息的其他进程（称为进程的*近邻*）的一个子集。如果进程p能够发送消息到进程q，则称从p到q存在*信道*。除非说明，否则假设信道是*双向的*，即，在同一信道上，允许进程q向进程p发送消息。仅允许从p到q单向通信的信道称为*单向或有向信道*。

进程和通信子系统的集合也称为*网络*。通信子系统的结构常常用图$G = (V, E)$表示，图中，节点表示进程，当且仅当两进程之间存在信道，两进程之间存在边。可以把单向信道的系统类似地表示成有向图。分布式系统的图也称为*网络拓扑结构*。

表示成图可以让我们用图论的术语讨论通信系统；关于术语的简介参见附录B。因为网络拓扑结构对许多问题的分布式算法的存在性、外观、复杂度有着重要影响，以下简明讨论通常所用的拓扑结构；详情参见附录B。贯穿本书，除非有特别的说明，都假设拓扑结构是*连通的*，即任何两个节点之间存在路径。

63 ~ 64

（1）*环* N-节点的环（ring）是图，有节点v_0到v_{N-1}和边v_iv_{i+1}（下标模N）。由于其简单性，环常用于分布式控制的计算中。同时，一些物理网络，如令牌环(Token Ring)[Tan96，参见

4.3.3节]，也按照环排列节点。

（2）树　N-节点的树是有$N-1$条边，且不包含环的连通图。在分布式计算中使用树，是因为它允许以较低通信开销进行计算，此外，每一连通图中包含树作为生成子网。

（3）星　N-节点的星有一特殊节点（中心），$N-1$条边，将$N-1$个节点连向中心。在集中式计算中使用星形连接，其中一个进程作为控制器，其余进程只与该特殊进程通信。星型拓扑结构的缺点是在中心节点会出现通信瓶颈和系统易受中心节点发生的故障影响。

（4）团　团是一个网络，其中任意两节点之间存在边。图中“隐含地”假设，每一进程能够直接地与其他进程通信。参见第13章和第16章。

（5）超立方体　超立方体$HC_N=(V, E)$是一个有$N=2^n$个节点的图。这里V是长为n的二进制位串的集合：

$$V=\left\{(b_0,\ldots,b_{n-1}):b_i\in\{0,1\}\right\},$$

当且仅当位串b，c只在其中某一位不同，则这两个节点b，c之间有一条边相连。超立方体的名字指的是将网络图示成n-维的单位立方体，立方体的顶点即为节点。

这些网络的例子参见图2-4。网络的拓扑结构可以是*静态的*也可以是*动态的*。静态拓扑结构表示，在分布式计算过程中拓扑结构保持不变。动态拓扑结构表示，在分布式计算过程中，可以添加或去除通道（有时甚至是进程）。这些拓扑结构上的变化可以通过配置的转移来模拟，即，是否进程状态反映了进程的近邻集合（参见第4章）。 65

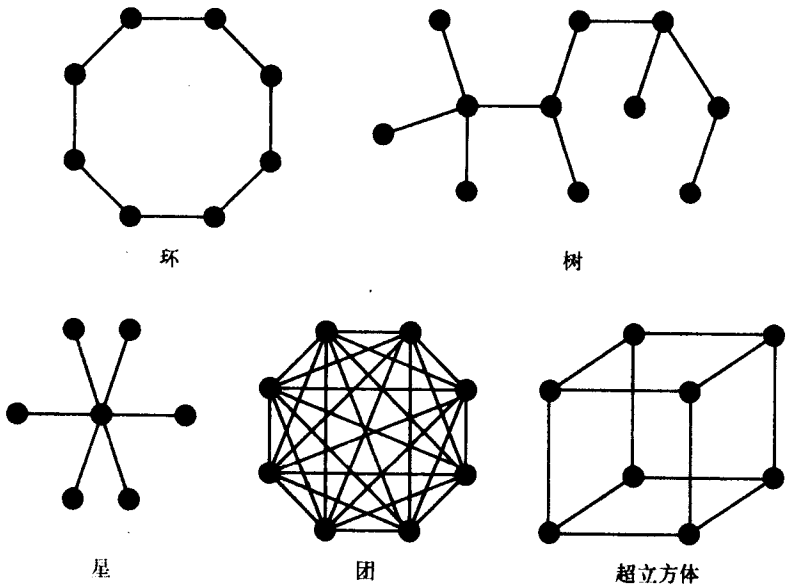


图2-4　通常所使用的拓扑结构示例

2.4.2　信道性质

可将2.1.2节所描述的模型精细化，将每一信道的内容分别表示在配置中，即对每一单向信道pq用集合集M_{pq}替代集合M。正像我们已经假定的那样，每一消息隐含地定义了它的目的

地，因而，这样的修改并不改变模型的重要性质。下面，讨论对发送和接收事件通信所做的假设。

（1）*可靠性* 称信道是*可靠的*，当在信道中所发送的每一消息只被明确地接收一次（假定目的地能够接收消息）时。除非特别声明，否则本书总是假定，信道是可靠的。这种假设实际上增加了（弱）公平性条件；的确，在消息发出之后，消息的接收方是可应用的。

信道的不可靠可能显现为*通信故障*，有几种类型的故障，例如，丢失、混淆、复制、创建。通过2.6节定义的模型，用转移代表这些故障。 66

消息的*丢失*发生在所发送的消息从未被接收；可用从M中去掉消息的一次转移进行模拟。消息的*混淆*（garble）发生在所接收的消息不同于发送的消息的时候。它可通过改变M中一条消息的转移加以模拟。消息的*复制*出现在接收的消息多于所发送的消息。它可通过复制M中一条消息的转移进行模拟。消息的*创建*出现在当消息被接收，而从未发送过该消息。它可通过在M中插入一条消息的转移加以模拟。

（2）*先进先出的性质* 称信道是*先进先出*（fifo）的，如果考虑所发送消息通过信道的次序。即，如果进程p向进程q发送两条消息m_1，m_2，并且在p中m_1的发送早于m_2的发送，那么在q中m_1的接收早于m_2的接收。除非特别声明，本书将不假设先进先出信道。

在定义2.6的模型中，将集合M用一组*队列*（queue）代替，表示先进先出信道，每个信道一个队列。发送时，将消息添加到此队列的末尾，接收事件从队列头删除一条消息。当假设先进先出信道时，引出新的类型的通信故障，即消息在信道中重排。它可通过交换队列中的两条消息的转移进行模拟。

有时分布式算法得益于信道的先进先出的性质；参见3.1节的通信协议的例子。利用消息接收的次序会降低每次消息中必须传递的信息量。然而，在许多情况下，即使消息可以在信道重排，也能设计算法使之正常（有效）运行。一般而言，在分布式系统中实现先进先出的性质，可以降低计算的固有并行性，因为在消息被处理之前，它需要消息缓冲。因此，本书并不隐含的假设先进先出的性质。

Ahuja[Ahu90]提出了弱假设；*清仓信道*（flush channel）仅考虑发送方所确定的消息次序。Schiper 等人[SES89]定义了*因果次序的消息传输*如下。如果p_1、p_2向事件e_1和e_2中的进程q发送 67
消息m_1和m_2，且$e_1 \prec e_2$，那么q在m_2之前接到m_1。传输假设的层次性由完全异步性、因果次序的传输、先进先出、同步通信构成，参见Charron-Bost等人[CBMT96]的讨论。

（3）*信道容量* 同时在信道中传输的消息数定义为*信道容量*（channel capacity）。当信道中包含的消息等于信道的容量时，信道是*满的*。仅当信道不满时，发送事件是可应用的。

定义2.6给出了无限容量信道的模型，即信道永远不会满。本书总是假设信道容量是无限的。

2.4.3 实时性假设

分布性是所提出模型的本质特征：不同进程中事件的完全独立性，如定理2.19中所表示的那样。当假设全局时间帧和进程观察物理时间的能力（物理时钟设备）时，就会失去这一性质。当某实时时间扫过时，这个时间会扫过所有进程，这个过程出现在每一进程的时钟中。

通过为每个进程配备实时时钟变量可以将实时时钟结合进来；通过取出进程时钟的转移

可以模拟实时扫过时间；参见3.2节。通常，结合实时时钟可用性，假设消息的传输时间（消息发送与接收之间的时间）有限。这个界限可以包含在通用转移系统模型中。

除非说明本书对于实时性不做一般性的假设；我们考虑完全异步的系统和算法。在3.2节、第12章和第15章会用到实时性的假设。

2.4.4 进程知识

初始进程的知识是指分布式系统中的信息，用进程的初始状态表示。如果某一算法声称
68 依赖于这种信息，则它假设了在系统开始执行之前，有关的信息已经正确地存储在进程中。具有这种知识的例子含有以下信息。

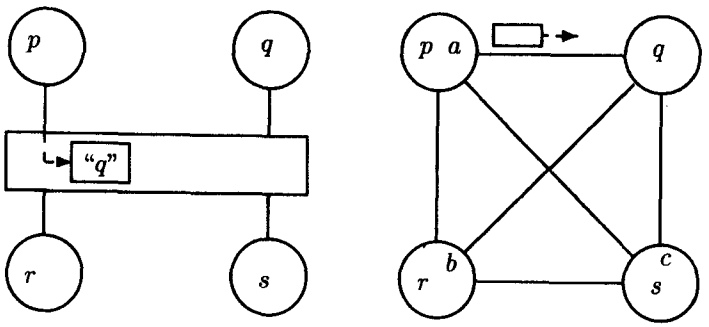
（1）*拓扑信息*　拓扑信息包括：进程数、网络图直径和图的拓扑结构。如果对于，进程图中带有方向的一致标签边是为进程所知的（参见附录B），则称网络*有方向侦听*（sense of direction）。

（2）*进程标识*　很多算法要求进程有惟一的名字（标识），初始时，每一进程知道自己的名字。假定进程包含一个变量，初始化为进程名（即，每一进程有不同的名字）。可对选择名字的集合做进一步假设。例如，名字是线性有序，或者它们是（正）整数。

除非说明，本书总是假设进程可以访问它们的标识；在这种情况下，称系统为*命名网络*（named network）。非此种情况，称为*匿名网络*（anonymous network），将在第9章中讨论。

（3）*近邻标识*　如果用惟一名字区分进程，可假设每一进程初始时知道它近邻的名字。这种假设称为*近邻知识*（neighbor knowledge），一般不做此假设。进程名对消息定址有用；当用*直接定址*（direct addressing）法来发送消息时，给定了消息的目的地名。一个更强的假设是，每一进程知道整个进程名的集合。

弱的假设是，进程只知道其近邻存在，但并不知道名字。在这种情况下，不能利用直接定址法，当为消息定址时，进程利用信道的局部名字，这称为是*间接定址*（indirect addressing）法[SK87，P.54]。直接定址和间接定址法如图2-5所示。直接定址法利用进程标识作为地址，而在间接定址法中，进程p、r和s用不同的名字（分别为a、b和c）为带有目的地q的消息定址。



a）直接定址　　b）间接定址

图 2-5

2.4.5 分布式算法的复杂度

正确性是分布式算法最重要的性质：必须满足算法对要解决的问题所提出的要求。为比

较同一问题的不同算法，有必要对算法所消耗的资源进行度量。消耗越低，算法越“好”。可用以下方法度量分布式算法所消耗的资源。

（1）消息复杂度 算法中所交换的消息总数。

（2）位复杂度 因为集合$\mathcal{M}$通常包含不同的消息，每条消息必须传输的信息的数量用$\mathcal{M}$标识。对于利用集合$\mathcal{M}$的一个算法，若集合$\mathcal{M}$较小，每条消息只能用少量的位来标识，而使用许多不同消息的算法在每条消息中需要更多位来标识。因为传输“长”消息比“短”消息开销大，因此也可以计算消息中所包含的总位数。

本书中的大多数算法，使用包含$O(\log N)$位的消息（其中N是进程数）。因此，它们的位复杂度超过了消息复杂度一个对数因子。在大多数情况下，只分析算法的消息复杂度。只有在利用很长或很短的消息时，才分析算法的位复杂度。

（3）时间复杂度 如果分布式算法模型中没有包含时间概念，定义分布式算法的复杂度就不那么显而易见。在文献中（参见6.4节，作为比较），可以找到不同的复杂度的定义。本书的定义基于计算中事件的理想时间，按照以下假设。 69 ~ 70

（a）处理一个事件的时间为0时间单位。

（b）传输时间（即，发送和接收一条消息之间的时间）最多为一个时间单位。

在这些假设之下，算法的时间复杂度是计算所消耗的时间。注意此假设仅适合于定义算法的时间复杂度。异步算法正确性的证明必须独立于这些假设。

（4）空间复杂度 算法的空间复杂度等于执行算法的进程所需存储量的大小。进程所需的空间是进程状态数的对数。

由于分布式算法的操作是不确定性的，算法可能产生多种计算，对于这些计算，度量可能是不相等的。因此产生最坏情况复杂度和平均情况复杂度的区别。最坏情况的复杂度是算法的最大计算复杂度。平均情况复杂度是算法中所有可能计算的平均值，需定义所有计算上的概率分布。

习题

2.1节

2.1 定义一个分布式系统模型，其支持同步和异步消息传输。

2.2节

下列三个练习研究不变式和总是为真谓词的差异，研究不变式的析取与合取及其导出式，以及并行程序组合的不变式的行为。

2.2 给出一个转移系统S和断言P，满足P在S中总是为真，但不是S的不变式。

（提示：在文献[GT90]中，给出了具有三种配置的一个例子。能找出具有两种配置的一个例子吗？） 71

2.3 假设P_1、P_2是系统S的不变式，证明$(P_1 \vee P_2)$和$(P_1 \wedge P_2)$是不变式。

假设P_1、P_2是系统S的Q-导出的。证明$(P_1 \vee P_2)$和$(P_1 \wedge P_2)$是系统S的Q-导出的。

具有同一配置集和初始配置集的转移系统可通过将其转移关系组合而成。因此$S_1 = (C, \rightarrow_1, \mathcal{I})$与$S_2 = (C, \rightarrow_2, \mathcal{I})$的并行组合是系统$S = (C, \rightarrow, \mathcal{I})$，其中$\rightarrow = (\rightarrow_1 \cup \rightarrow_2)$。

2.4 设S是S_1和S_1的并行组合。证明，如果P是S_1和S_2的不变式，那么P是S的不变式。证

明，如果P是S_1和S_2的Q-导出的，那么P是S的Q-导出的。给出一个实例，使得P在S_1和S_2总是为真，在S中却不是。

下个练习关注良基集。即在集合中有一个元素，存在集合中的无限多个元素大于它，这个集合可为良基的。向量$a = (a_0, a_1, \cdots, a_n)$和$b = (b_0, b_1, \cdots, b_n)$字典序由以下关系定义。

$$a <_l b \Leftrightarrow a_1 < b_1 \vee \left(a_1 = b_1 \wedge \left(a_2,\ldots,a_n\right) <_l \left(b_2,\ldots,b_n\right)\right)$$

2.5　证明$(\mathbb{N}^n, <_l)$（其中$n \geqslant 2$）存在多个元素，小于集合中的无限多个元素。

证明$(\mathbb{N}^n, <_l)$是良基的。

2.3节

2.6　对带有时钟值的图2-1中的事件加标签，时钟值由Lamport逻辑时钟给出。

对带有时钟值的事件加标签，时钟值由Mattern向量时钟给出。

区分并发事件对，并检查赋给这些事件的标签是否为有序的。

2.7　给出一个分布式算法（与图2-3中的算法类似），它可对带有时钟值的事件加标签，时钟值由Mattern向量时钟给出。

2.8　你能反复将定理2.19应用到E的事件中，来证明定理2.21吗？

72　2.9　证明定理2.24。

2.10　定义具有同步通信的系统的转移因果序。并为此系统定义时钟。给出计算时钟的分

73　布式算法。

第3章 通信协议

本章讨论两个协议，它们可用于两计算站点间可靠的数据交换。在理想情况下，只需通过发送消息和接收消息即可交换数据。然而令人遗憾的是，不是总能够忽略通信出错的可能性；消息经过物理介质传输时，可能丢失、复制、重排或者混淆。必须检测出这些错误，并通过辅助机制加以纠正。在传统上，这些机制称为协议，在计算站点中实现。

这些协议的主要功能是*数据传输*，即在一个站点上接收信息，在另一个站点上发送信息。可靠的数据传输包括反复发送丢失的消息、拒绝或纠正混淆的消息和丢弃复制的消息。为此，协议维护状态信息，这些信息记录着哪些数据已发送，哪些数据已证实接收等等。利用状态信息的必要性引出了*连接管理*问题，即状态信息的初始化和丢弃。称初始化为*打开*（opening）一个连接，称丢弃为*关闭*（closing）一个连接。连接管理的困难在于当关闭连接时，消息仍然有可能留在信道中。当不存在连接或者在后一次连接的过程中，这样的消息仍可被接收，并且它的接收必须不影响当前连接的正常操作。

按照协议的层次，本章所讨论的协议适用于不同的层，例如，OSI参考模型（1.2.2节）。它们被包括在本书中是由于多种原因；第一个协议是完全异步的，第二个协议要靠正确地使 74
用计时器。在两种情形下，对协议的验证集中在所需要的*安全性*方面，即，接收方只发送正确的数据。

第一个协议（3.1节）是为两个站点之间进行数据交换而设计的，两站点之间有直接的物理连接（如，电话线），因此属于OSI模型的数据链路层。第二个协议（3.2节）是为必须通过中间网络（可能包含其他站点，并经过不同路径连接到结束站点）进行通信的两站点设计的。因此，这个协议属于OSI模型的传输层。这种差异影响着协议所要求的功能，体现在以下两方面。

（1）*差错考虑* 对于这两个协议，考虑不同类型的传输错误。在一个物理连接中，消息不能相互经过旁路，也不能被复制；在3.1节，只考虑消息丢失的情况（对于消息的混淆，参见下面）。在网络中，由于消息可沿不同路径行进，因此，相互可经过旁路；同时，由于中间站点的故障，消息不仅可能丢失而且可能复制。因此，在3.2节，讨论消息的丢失、复制和重定序问题。

（2）*连接管理* 在第一个协议中，将不考虑连接管理，但在第二个协议中要考虑。通常假设物理连接可持续操作很长一段时间，而不是反复打开和关闭连接。这不是与远程站点连接的情况。这样的连接可能是暂时地用于交换一些数据，但是由于开销太大，不能不确定地维持与每个远程站点的连接。因此，在第二个协议中，需要具有打开和关闭连接的能力。

第一个协议表明，要达到数据传输协议所要求的安全性，基于计时器的机制不是主要问题。根据2.2节描述的证明方法，3.1节给出了第一个大的安全性证明的例子。普遍认为[Wat81]，计时器的正确使用和对消息传输中的时间限制对安全连接管理是必要的。因此，为了证明连接管理协议的安全性，必须考虑计时器的作用。3.2节表明，分布式系统（定义2.6）的模型是如何被扩展到使用计时器的进程中，并提供了此扩展的一个例子。 75

消息混淆

考虑消息在传输过程中混淆的可能性是现实的。由于环境噪声、存储单元故障等因素，可能造成通过物理连接进行通信的消息遭到破坏。然而，假设接收进程可以检测出消息混淆，例如，通过奇偶校验或者更一般的校验和机制（参见[Tan96，第3章]）。对于收到的混淆消息，处理起来就好像没有接到消息，因而消息混淆事实上导致消息丢失。因此，并不明确处理消息的混淆；而总是考虑消息丢失的可能性。

3.1 平衡滑动窗口协议

本节研究了可靠的双向信息发送对称协议。这一协议取自[Sch91，第2章]。因为它用于有直接连线的节点的信息交换，因此假设信道有先进先出的性质。这个假设直到3.1.3节才用，在假设中，对协议所用的序列号进行了限制。3.1.1节给出了该协议，3.1.2节证明了它的正确性。

用p、q表示两个通信进程。假设、要求和协议关于p、q是完全对称的。p的输入是由必须发送给q的信息组成，用无限大的字数组in_p模型化。p的输出由接收q的信息组成，也用无限大的字数组out_p模型化。暂时假设p能对数组in_p随机读，对数组out_p随机写。初始时，$out_p[i]$值未定义，用*udef*表示。进程q的输入和输出分别用in_q、out_q进行模型化。这些数组的下标为自然数，从0开始。3.1.3节表明，对于数组的随机访问，可以限制到访问具有有限长度的“窗口”上，窗口在数组上滑动。这就是为什么称此协议为“滑动窗口”协议。

进程p包含一个变量s_p，表明p期望从q接收的最小的有限字。因此，在任意时刻，p已经写入$out_p[0]$到$out_p[s_p-1]$。s_p的值永不递减。类似地，进程q包含一个变量s_q。现在阐述协议所要求
76 的性质。安全性指的是，每一进程仅输出正确的数据；活动性指的是，所有数据最终被发送。

（1）*安全发送*　在协议的每一个可达配置中，

$$out_p\left[0..s_p-1\right]=in_q\left[0..s_p-1\right] \text{ 且 } out_q\left[0..s_q-1\right]=in_q\left[0..s_q-1\right]$$

（2）*最终发送*　对每一整数$k>0$，最终达到配置，达到最终配置时有$s_p>k$和$s_q>k$

3.1.1 协议表示

传输协议通常依赖于确认消息的使用。接收进程发送确认消息，通知发送方数据已经正确接收。如果数据的发送方没有接到确认，它就假定数据消息（或确认）丢失，并重发数据。然而，在本节的协议中，没有使用明确的确认消息。该协议中两个站点都可以向另一方发送消息；站点的消息也可用作对另一站点消息的确认。

进程交换的消息称作包，它们具有形式<**pack**，w，i>，其中w是数据字，i是自然数（称为包的序列号）。当p把该包向q发送时，传输字$w=in_p[i]$。同时正如前面所述，对q所接收包的序列号进行确认。如果假定进程p所发送的数据包<**pack**，w，i>确认q接收了$0..i-l_p$个字，进程p就可以是q“前”固定数目l_p个包。（进程q发送时，包的含义是类似的。）常数l_p和l_q是非负的，且为进程p、q所知。对于协议的转移，作为确认消息的数据包含有两方面的意义：

（1）仅当存储$out_p[0]$至$out_p[i-l_p]$的所有字之后，即，如果$i<s_p+l_p$，进程p才可以发送字$in_p[i]$（正如包<**pack**，$in_p[i]$，i>）。

（2）当p接收<**pack**，w，i>，不再需要重新传输从$in_p[0]$到$in_p[i-l_q]$的字。

1. 伪代码的解释

在做了这些设计选择之后，不难给出协议的代码；参见图3-1中的算法。将变量a_p、a_q分别引入进程p、q中，表示p、q分别还未收到确认时，编号最小的字。 77

在图3-1的算法中，行为$\mathbf{S}_p$表示p所发送的第i个字，行为$\mathbf{R}_p$表示p所接收的一个字，行为$\mathbf{L}_p$表示目的地为p的包丢失。进程p可以发送下标位于上述所列范围的任意字。当接收到消息时，首先做校验，是否接到相同的消息（以防是重传）。如果不是相同消息，则将所包含的字写入输出，并更新a_p和s_p。当p，q颠倒时，也会有行为$\mathbf{S}_q$、$\mathbf{R}_q$和$\mathbf{L}_q$。

2. 协议的不变式

可用两队列来表示通信子系统。Q_p表示目的地为p的包。Q_q表示目的地为q的包。可以看出$\mathbf{R}_p$中s_p的更新计算，从不会计算出比以前小的值，因此，s_p从不递减。为了证明算法满足早先给定的要求，首先要证明，断言P是不变式。（在本断言和其他断言中，i是自然数。）

$$
\begin{aligned}
P \equiv\ & \forall i < s_p:\ out_p[i] \neq udef && (0p)\\
\wedge\ & \forall i < s_q:\ out_q[i] \neq udef && (0q)\\
\wedge\ & \langle \mathbf{pack},\ w,\ i\rangle \in Q_p \Rightarrow w = in_q[i] \wedge (i < s_q + l_q) && (1p)\\
\wedge\ & \langle \mathbf{pack},\ w,\ i\rangle \in Q_q \Rightarrow w = in_p[i] \wedge (i < s_p + l_p) && (1q)\\
\wedge\ & out_p[i] \neq udef \Rightarrow out_p[i] = in_q[i] \wedge (a_p > i - l_q) && (2p)\\
\wedge\ & out_q[i] \neq udef \Rightarrow out_q[i] = in_p[i] \wedge (a_q > i - l_p) && (2q)\\
\wedge\ & a_p \lessdot s_q && (3p)\\
\wedge\ & a_q \lessdot s_p && (3q)
\end{aligned}
$$

引理3.1 *P是图3-1中的算法的不变式。*

证明。在每个初始配置中，Q_p、Q_q都为空，对所有i，$out_p[i]$和$out_q[i]$为$udef$，a_p、a_q、s_p、s_q置为0；这些蕴含着P为真。现在依次考虑协议的转移，以证明在转移中P保持性质。首先注意到，in_p和in_q的值不变。

$\mathbf{S}_p$：为证明$\mathbf{S}_p$保持式（0p），观察可得$\mathbf{S}_p$不增加s_p，也不会使$out_p[i]$等于$udef$。

为证明$\mathbf{S}_p$保持式（0q），观察可得$\mathbf{S}_p$不增加s_q，也不会使$out_q[i]$等于$udef$。

为证明$\mathbf{S}_p$保持式（1p），观察可得$\mathbf{S}_p$不增加Q_p的包，也不会使s_q下降。

为证明$\mathbf{S}_p$保持式（1q），观察可得$\mathbf{S}_p$为Q_q增加$\langle \mathbf{pack},\ w,\ i\rangle$，且$w = in_p[i]$和$i < s_p + l_p$，而保持$s_p$值不变。 78

为证明$\mathbf{S}_p$保持式（2p）和式（2q），观察可得$\mathbf{S}_p$不改变out_p、out_q、a_p或a_q的值。

为证明$\mathbf{S}_p$保持式（3p）和式（3q），观察可得$\mathbf{S}_p$不改变a_p、a_q、s_q或s_p的值。

$\mathbf{R}_p$：为证明$\mathbf{R}_p$保持式（0p），观察可得$\mathbf{R}_p$不会使$out_p[i]$等于$udef$，如果它重新计算s_p，也满足式（0p）。

为证明$\mathbf{R}_p$保持式（0q），观察可得$\mathbf{R}_p$不改变out_q或s_q。

为证明$\mathbf{R}_p$保持式（1p），观察可得$\mathbf{R}_p$不增加Q_p的包，也不会使s_q下降。

为证明$\mathbf{R}_p$保持式（1q），观察可得$\mathbf{R}_p$不增加Q_q的包，也不会使s_p下降。

为证明$\mathbf{R}_p$保持式（2p），观察可得$\mathbf{R}_p$一旦接到$\langle \mathbf{pack},\ w,\ i\rangle$，就将$out_p[i]$改为$w$。因在$\mathbf{R}_p$被应用之前，$Q_p$包含此包，式（1p）蕴含$w = in_q[i]$。赋值语句$a_p := \max(a_p,\ i - l_q + 1)$保证应用之后$a_p > i - l_q$成立。

为证明$\mathbf{R}_p$保持式（2q），观察可得$\mathbf{R}_p$不改变out_q或a_q的值。

为证明$\mathbf{R}_p$保持式（3p），观察可得当$\mathbf{R}_p$执行赋值语句$a_p := \max(a_p, i-l_q+1)$（一旦接到<**pack**, w, i>），式（1p）蕴含$i < s_q + l_q$。因此赋值语句执行后，$a_p \leqslant s_q$成立。$\mathbf{R}_p$不改变s_q。

为证明$\mathbf{R}_p$保持式（3q），观察可得，s_p只能在$\mathbf{R}_p$的应用中增加。

$\mathbf{L}_p$：为证明$\mathbf{L}_p$保持式（0p）、式（0q）、式（2p）、式（2q）、式（3p）和式（3q），观察可得$\mathbf{L}_p$不改变进程的任何状态就足够了。因为$\mathbf{L}_p$仅删除包（并不插入和混淆它们），因此它保持式（1p）和式（1q）。

按照对称性，$\mathbf{S}_q$、$\mathbf{R}_q$和$\mathbf{L}_q$保持P。 □

```
var s_p, a_p : integer                    init 0, 0 ;
    in_p     : array of word              (* Data to be sent *) ;
    out_p    : array of word              init udef, udef, ... ;

S_p: { a_p ⩽ i < s_p + l_p }
     begin send ⟨pack, in_p[i], i⟩ to q end

R_p: { ⟨pack, w, i⟩ ∈ Q_p }
     begin receive ⟨pack, w, i⟩ ;
           if out_p[i] = udef then
             begin out_p[i] := w ;
                   a_p := max(a_p, i − l_q + 1) ;
                   s_p := min{j | out_p[j] = udef}
             end
           (* else ignore, packet was retransmission *)
     end

L_p: { ⟨pack, w, i⟩ ∈ Q_p }
     begin Q_p := Q_p \ {⟨pack, w, i⟩} end
```

图3-1　平衡滑动窗口协议（进程p）算法

3.1.2　协议的正确性证明

现在证明图3-1所示的算法能确保安全性并最终发送。安全性隐含在不变式中，如定理3.2，但是活动性的证明有点难。

定理3.2　图3-1所示的算法满足安全发送的要求。

证明。式（0p）和式（2p）蕴含$out_p[0..s_p-1] = in_q[0..s_p-1]$，式（0q）和式（2q）蕴含$out_q[0..s_q-1] = in_p[0..s_q-1]$。 □

为证明协议的活动性，有必要做关于公平性的假设，以及关于l_p、l_q的假设。如果没有这些假设，协议不能满足活动性要求，正如在下面的证明中所见。由于非负常数l_p、l_q未定；如果它们为0，则在初始配置（每一初始配置是终止配置）中，协议死锁。因此假设$l_p+l_q > 0$。

协议的配置可表示为$\gamma = (c_p, c_q, Q_p, Q_q)$，其中$c_p$、$c_q$是$p$、$q$的状态。设$\gamma$是一个配置，其中$\mathbf{S}_p$是可应用的（对某些$i$）。设

$$\delta = \mathbf{S}_p(\gamma) = \left(c_p, c_q, Q_p, (Q_q \cup \{m\})\right),$$

注意，$\mathbf{L}_q$在δ中是可应用的。如果$\mathbf{L}_q$去除m，则$\mathbf{L}_q(\delta) = \gamma$。关系式$\mathbf{L}_q(\mathbf{S}_p(\gamma)) = \gamma$引起无限计算，既不增加$s_p$，也不增加$s_q$。

如果满足下面的两个公平性假设，则协议满足最终发送要求。

F1 对于无限长的时间，如果包的发送是可应用的，则可无限多次地发送包。

F2 如果同一包可发送无限多次，则它可无限多次被接收。

79 ~ 80

假设F1保证，如果没有接到确认，可反复的再次重发包；F2排除了重发从未接收的情况。

这两个进程都不能一个在另一个之前太多，s_p和s_q之间的差距应保持有限。这就是为什么协议称之为平衡协议，它表明，如果满足s_p最终发送的要求，那么它也满足s_q的要求。反之亦然。它还表明，该协议不能用于一个进程所发送的字多于另一个进程时的情况。

引理3.3 *P蕴含* $s_p - l_q \leqslant a_p \leqslant s_q \leqslant a_q + l_p \leqslant s_p + l_p$ 。

证明。由式（0p）和式（2p）可得，$s_p - l_q \leqslant a_p$ 成立。由式（3p)可得，$a_p \leqslant s_q$ 。由式（0q）和式（2q）可得，$s_q \leqslant a_q + l_p$ 成立。由式（3q）可得 $a_q + l_p \leqslant s_p + l_p$ 成立。 □

定理3.4 *图3-1所示的算法满足最终发送要求。*

证明。首先证明，协议不可能死锁。不变式蕴含着，两个进程之一能够发送包含最小编号字的包，该包还是另一进程所缺少的。 □

声明3.5 *P蕴含着，p的发送<pack，* $in_p[s_q]$，s_q *>，或者q的发送<pack，* $in_q[s_p]$，s_p *>都是可应用的。*

证明。由于 $l_p + l_q > 0$，在引理3.3中，至少有一个是严格不等式，也就是，

$$s_q < s_p + l_p \vee s_p < s_q + l_q \text{ 。}$$

P还蕴含，$a_p \leqslant s_q(3p)$ 且 $a_q \leqslant s_p(3q)$，由此得出，

$$\left(a_p \leqslant s_q < s_p + l_p\right) \vee \left(a_q \leqslant s_p < s_q + l_q\right),$$

即，对于 $i = s_q$，$\mathbf{S}_p$是可应用的，或者对于 $i = s_p$，$\mathbf{S}_q$是可应用的。 □

我们现在可以证明，在每一次计算中，s_p和s_q无限次增加。按照推论3.5，协议没有终止配置，因此，每次计算是无限的。假设C是一次计算，其中s_p和s_q只做有限次增加，设σ_p、σ_q是C中变量所取的最大值。根据声明，在a_p、a_q、s_p和s_q到达最终值之后，p的发送<pack，$in_p[\sigma_q]$，σ_q >，或者q的发送<pack，$in_q[\sigma_p]$，σ_p >都是永远可应用的。因此，按照F1，有一个包会无限多次的发送，按照F2，有一个包会无限多次的接收。但是，当p接到带有序列号s_p的包时，就会引起s_p增加（反之，对于q也是如此），这与s_p和s_q中没有一个会无限增加的假设相矛盾。因此定理3.4得证。 □

81

本节，对假设F1、F2做了简要的讨论。F2是连接P、q的信道必须满足的进行数据交换的最小质量要求。显然，如果某些字$in_p[i]$从不通过信道，这些字最终也不可能发送。在协议中，假设F1是利用超时条件实现的：如果在某一特定的时间内，a_p不增加，则再次传输$in_p[a_p]$。正如已在本章的引言中讨论的那样，如果不考虑时间问题，就可证明协议的安全性。

3.1.3 协议讨论

1. 限制进程中的存储空间

由于在图3-1所示的算法中，每一进程存有无限数量的信息量（数组*in*和*out*）和利用了无界的序列号，因此，在计算机网络中是不可能实现的。现在要证明，任何时刻，只须存储有限个字，就足够了。设 $L = l_p + l_q$ 。

引理3.6 *P蕴含着，p的发送<pack，w，i >仅对* $i < a_p + L$ *是可应用的。*

证明。$\mathbf{S}_p$的阈值要求$i < s_p + l_p$，由引理3.3可得，$i < a_p + L$。 □

引理3.7 *P蕴含着，如果$out_p[i] \neq udef$，那么，$i < s_p + L$。*

证明。由式（2p），$a_p > i - l_q$；因此，$i < a_p + l_q$，由引理3.3可得，$i < s_p + L$。 □

图3-2说明了这两个引理的结果。进程p仅需要存储字$in_p[a_p .. s_p + l_p - 1]$。因为这些是进程$p$能发送的字。称它们为$p$的发送窗（在图3-2中表示为**S**）。无论什么时候，a_p增加，p都丢弃不再落入发送窗内的字（在图3-2中表示为**A**）。无论什么时候s_p增加，p都从产生这些字的地
82 方读取发送窗内的下一字。按照引理3.6，p的发送窗至多包含L个字。

类似的证明给出了p中存储数组$out_p[i]$所需的空间范围。对于$i < s_p$，因为$out_p[i]$的值不再改变，可以假设p将不可逆地输出这些值，不再存储它们（在图3-2中表示为**W**）。对于$i \geq s_p + L$，由于$out_p[i] = udef$，$out_p[i]$中的这些值也不需存储在进程中，子数组$out_p[s_p .. s_p + L - 1]$称为$p$的接收窗。在图3-2中，接收窗用**u**表示*udef*，用**R**表示已经接收的字。进程中，仅存储落入该窗内的字。引理3.6和3.7表明，每个进程的任一时刻，至多存储$2L$个字。

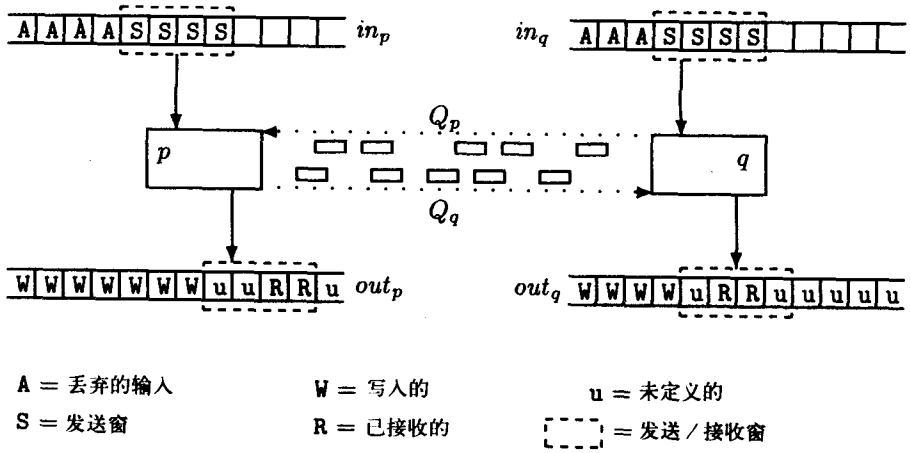


图3-2　协议的滑动窗口

2. 限制进程中的序列号

如果利用信道的先进先出性质，也可对进程中的序列号进行限制。利用先进先出的假设，可以证明，进程p所接收的包的序列号总是落入以s_p为中心的$2L$的范围内。注意，这是第一次
83 利用先进先出假设。

引理3.8 *定义断言P'为*

$$P' \equiv P$$
$$\wedge <\mathbf{pack}, w, i> \text{在} <\mathbf{pack}, w', i'> \text{之后}\ (Q_p \text{中}) \Rightarrow i > i' - L \qquad (4p)$$
$$\wedge <\mathbf{pack}, w, i> \text{在} <\mathbf{pack}, w', i'> \text{之后}\ (Q_q \text{中}) \Rightarrow i > i' - L \qquad (4q)$$
$$\wedge <\mathbf{pack}, w, i> \in Q_p \Rightarrow i \geq a_p - l_p \qquad (5p)$$
$$\wedge <\mathbf{pack}, w, i> \in Q_q \Rightarrow i \geq a_q - l_q \qquad (5q)$$

是图3-1所示的算法的不变式。

证明。因为已证明P为不变式，我们只需证明初始时式（4p）、式（4q）、式（5p）和式（5q）成立，并在每次转移中保持不变。因为在初始配置中，队列为空，因此式（4p）、式（4q）、式（5p）和式（5q）成立。以下证明，转移保持这些断言。

$\mathbf{S}_p$: 为证明S_p保持式（4p）和式（5p），观察可得$\mathbf{S}_p$不向Q_p添加包，也不改变a_p的值。
为证明$\mathbf{S}_p$保持式（5q），观察可得如果$\mathbf{S}_p$为Q_q增加包<**pack**，w，i>，那么，$i \geqslant a_p$，由引理3.3，这蕴含着$i > a_q - l_q$。
为证明$\mathbf{S}_p$保持式（4q），观察可得，如果<**pack**，w'，i'>在Q_q中，那么由式（1q）可得，$i' < s_p + l_p$，因此，如果$\mathbf{S}_p$增加包<**pack**，w，i>且$i \geqslant a_p$则由引理3.3可得，$i' < a_p + L \leqslant i + L$。

$\mathbf{R}_p$: 为证明$\mathbf{R}_p$保持式（4p）和式（4q），观察可得$\mathbf{R}_p$不向Q_p，或者Q_q添加包。
为证明$\mathbf{R}_p$保持式（5p），观察可得当a_p增加（一旦接到<**pack**，w'，i'>）到$i' - l_q + 1$时，则对Q_p中其余的包<**pack**，w，i>，由式（4p）可得，$i > i' - L$。因此，在a_p增加之后，$i \geqslant a_p - l_p$成立。

为证明$\mathbf{R}_p$保持式（5q），观察可得$\mathbf{R}_p$不改变Q_q或者a_q。

$\mathbf{L}_p$: $\mathbf{L}_p$不增加Q_p或者Q_q的包，也不改变a_p或a_q的值；因此保持式（4p）、式（4q）、式（5p）和式（5q）不变。

由协议的对称性，$\mathbf{S}_q$、$\mathbf{R}_q$和$\mathbf{L}_q$同样保持P'。 □

引理3.9 P'蕴含

$$<\mathbf{pack}, w, i> \in Q_p \Rightarrow s_p - L \leqslant i < s_p + L$$

和

$$<\mathbf{pack}, w, i> \in Q_q \Rightarrow s_q - L \leqslant i < s_q + L$$

84

证明。设<**pack**，w，i> $\in Q_p$。由（1p）可得，$i < s_q + l_q$，由引理3.3，得$i < s_p + L$。由式（5p）可得，$i \geqslant a_p - l_p$，由引理3.3，得$i \geqslant s_p - L$。对于属于Q_q中的包也有类似证明。 □

按照引理，用模k的序列号足以发送该包，这里$k \geqslant 2L$。确实，已知s_p和$i \bmod k$，进程p就可计算出i。

3. 参数的选择

常数l_p和l_q对协议的效率有着重要的影响。其对协议吞吐量的影响，在文献［Sch91，第2章］中进行了概率分析。它的最优值取决于与系统有关的参数量，例如，

（1）通信时间，即，进程的两次连续操作之间的时间。

（2）来回路由延迟，即，从p到q传输一个包，并得到从q到p的确认的平均时间。

（3）差错概率，某一个包丢失的概率。

4. 交替位协议

滑动窗口协议的一个有趣的特例是：当$L = 1$时，即，$l_p = 1$和$l_q = 0$时（或者反之亦然）。变量a_p和a_q初始化为$-l_p$和$-l_q$，而不是0。可以证明，$a_p + l_q = s_p$和$a_q + l_p = s_q$总是成立，因此，在协议中只需存储a_p和s_p（a_q和s_q）其中之一。如果对计时器的使用进一步做出限制，以确保站点依次发送消息，就得到著名的交替位协议（alternating-bit protocol）[Lyn68]。

3.2 基于计时器的协议

通过分析Fletcher和Watson提出的端到端消息通信的Δt-协议的简化形式，我们研究在通信协议的设计和验证中，计时器所起的作用。该协议由[FW78]提出，但本节取自[Tel91b，3.2节]，并作了简化。该协议不仅提供了数据传输机制（正如3.1节平衡滑动窗口协议），而且还

能打开连接和关闭连接。对于消息的丢失、复制以及重排具有弹性。

协议的状态信息（数据传输部分）保存在称为连接记录的数据结构中。（在3.2.1就会看到
85 连接记录中有些什么信息）。连接记录随着连接的打开和关闭，建立和撤销连接。因此，如果连接记录存在，则称连接是打开的（在某一站点）。

为了集中讨论协议的相关方面（即，连接管理机制以及计时器在其中的作用），考虑该协议的一个简化版本。更实际和有效的关于该协议扩展的讨论，参见[FW78]和[Tel91b,3.2节]。以下对这里所描述的协议作了4个方面的简化。

（1）一个方向　仅考虑一个方向的数据传输，即，从p到q的方向。有时称p为发送方，q为接收方。然而，需要注意协议对消息确认在相反方向进行，即，从q到p的方向。

通常数据需要在两个方向进行传输。为了处理这种情况，需要另外执行第二个协议，在这个协议中，p和q的作用相反。也可以引入组合的data/ack消息，在组合消息中，既包含（与序列号对应的）数据，也包含基于计时器协议的确认包中的信息。

（2）单字接收窗　接收方不存储任何序列号高于期望数的数据包。仅当下一个到达的包是期望虑的包时，才加以考虑，并立即进行发送。该协议的一个更复杂的版本，是存储早先时刻到达的具有较大序列号的包，并在所有具有较低序列号的包都到达后，才发送它们。

（3）简化的计时假设　协议用最少量的计时器表示。例如，假设只要接收方的连接是打开的，接收进程可以在任一时刻发送确认消息。另一种方法是，在较小的时间间隔内发送确认消息，但这会使协议更复杂。

同时，在对协议的描述中，如在3.1节，省略了用于引发数据包重发的计时器机制。仅仅
86 包括确保协议安全性所需的机制。

（4）单字包　发送方只能在每一数据包中放进一个字。如果数据包能够包含几块连续的字，则协议会更有效。

协议是基于计时器的，即，进程可访问物理时钟设备。对系统中的时间及计时器作以下假设。

（5）全局时间　时间的全局度量可扩展到系统中的所有进程中，即，每一事件都是在某一时刻发生。假设每一事件自身的持续时间为0，事件发生的时刻对进程来说是不可见的。

（6）有限的包生命期　包的生存期可由常数μ（最大包生存期，maximum packet lifetime）限定。因此，如果一个包在时刻σ发送，在时刻τ接收，那么

$$\sigma < \tau < \sigma + \mu$$

如果一个包在信道中被复制，则在初始包（或者丢失）发送后的μ时间内，必须接到每一复制。

（7）计时器　进程不能得知其行为的绝对时间，但是它们可以访问计时器。计时器是进程的一个实值变量，其值按照时间连续递减（当在其他地方没有明确赋值时）。更精确地，如果Xt是一个计时器，我们用$Xt^{(t)}$表示它在时刻t的值，且如果在t_1与t_2之间，Xt未赋给不同的值，那么

$$Xt^{(t_1)} - Xt^{(t_2)} = t_2 - t_1$$

注意，在时间δ内，它们按照δ准确递减，在这种意义上，这些计时器准确运行。在3.2.3

节，我们将讨论计时器遭受漂移的情况。

如3.1节，发送方的输入字可以通过一个无限大的数组in_p来模拟。同时，不在p中存储整个数组；在任一时刻，p只能访问它的一部分。当p得到产生该字的进程的下一字时，可扩展（在较高端）被进程p访问的部分in_p。发送方称此操作为字的*接受*。

在本节中，对接收方的发送字模拟不同于3.1节。接收方不是写入（无限大）数组，而是通过称为*发送字*的操作将字交给消费进程。在理想的情况下，in_p中的每一字只传输一次，并且顺序正确。 87

然而协议的规范，要弱于此。这是因为在有限的时间间隔内，协议只允许处理in_p中的每一个字。协议不能保证在有限的时间内能接收字，因为在这期间，有可能丢失所有的包。因此，在发送协议产生差错报告来表明该字可能丢失的地方，协议的规范允许*报告的丢失*的可能性存在。（因此，要是高级协议能够再次向进程p提供该字，那么复制就可能有效地出现；然而在此，我们并不关注这个问题。）3.2.2节中要证明的协议性质包括：

（1）*无损* 在p接受字后的有限时间内，in_p中的每个字由q发送，或者p进行报告（“可能丢失”）。

（2）*有序* q所发送的字按照严格递增次序出现在in_p中。

3.2.1 协议表示

当先前连接不存在时，协议中首先打开连接，然后发送方接收下一个字，或者包到达时，接收方能够发送它。在这个协议中，要打开一个连接，在数据包发送之前，无需交换任何控制消息。这使在每次连接中，对于仅传输几个字的应用，协议是相对有效的（小量快速传输）。当发送方（接收方）的连接是打开的，谓词cs（或者cr）为真。对于发送方（或接收方），这通常不是明确定义的一个布尔变量；而是通过连接记录的存在性定义的一个打开连接。进程通过查找自己的打开连接表的连接记录的存在性，测试连接是否打开。

当发送方打开一个新的连接时，它就开始对接收的字从0编号。连接中已经接收的字的编号用$High$表示，已经收到确认的字用Low表示。这表明（类似于3.1节的协议）发送方可能传输序列号从Low到$High-1$的包。反之，在发送方接到字的时刻开始的长为U的时间间隔里，发送方可能只发送一字。为此，将计时器$Ut\,[i]$与每一个字$in_p\,[i]$关联；在接受的那一刻，计时器设置为U，对于要传输的字而言，计时器必须为正。因此，p的发送窗由范围在Low到$High-1$的字组成，且相关的计时器为正。 88

```
网络常量:
        μ      : real ;        (* Maximum packet lifetime *)
协议常量:
        U      : real ;        (* Length of send interval *)
        R      : real ;        (* Receiver time-out value: R ≥ U + μ *)
        S      : real ;        (* Sender time-out value: S ≥ R + 2μ *)
发送方连接记录:
        Low    : integer ;     (* Acknowledged words of current connection *)
        High   : integer ;     (* Accepted words of current connection *)
        St     : timer ;       (* Connection timer *)
```

图3-3 基于计时器的协议变量

```
接收方连接记录:
        Exp   : integer ;      (* Next expected sequence number *)
        Rt    : timer ;        (* Connection timer *)
通信子系统:
        M_q   : channel ;      (* Data packets for q *)
        M_p   : channel ;      (* Acknowledgement packets for p *)
辅助变量:
        B     : integer init 0 ;          (* Words in previous connections *)
        cr    : boolean init false ;      (* Connection exists at receiver *)
        cs    : boolean init false ;      (* Connection exists at sender *)
```

图3-3　(续)

协议发送的数据包，由位（*序列开始位*，以下讨论）、序列号和字组成。为了分析协议，每一数据包还有第4个域，称为*剩余包生存期*。按照有限生存期的假设，在被接收或者丢失之前，它表示包仍能在信道中所花费的最大时间。当包发送时，总认为剩余包生存期为μ。协议中所发送的确认包仅仅由q所期望的下个序列号组成；同样，为了分析方便，每一确认包也含剩余包生存期。

89

```
A_p: (* Accept next word *)
    begin if not cs then
            begin (* Connection is opened first *)
                create (St, High, Low) ; (* cs := true *)
                Low := High := 0 ; St := S
            end ;
        Ut[B + High] := U ; High := High + 1
    end

S_p: (* Send ith word from current connection *)
    { cs  ∧  Low ≤ i < High  ∧  Ut[B + i] > 0 }
    begin send ⟨data, (i = Low), i, in_p[B + i], μ⟩ ;
        St := S
    end

R_p: (* Receive an acknowledgement *)
    { cs  ∧  ⟨ack, i, ρ⟩ ∈ M_p }
    begin receive ⟨ack, i, ρ⟩ ; Low := max (Low, i) end

E_p: (* Generate error report for possibly lost word *)
    { cs  ∧  Ut[B + Low] < −2μ − R }
    begin error[B + Low] := true ; Low := Low + 1 end

C_p: (* Close the connection *)
    { cs  ∧  St < 0  ∧  Low = High }
    begin B := B + High ; delete (St, High, Low) end
    (* cs := false *)
```

图3-4　发送方协议算法

连接的关闭由发送方的计时器St和接收方的计时器Rt控制。由于每个字的有限发送时间间隔和包的有限生存期，因此在信道中找到每个字的时间限制在$\mu+U$的时间间隔内，自接受字

90

的那一刻起。这使得接收方在接到字之后，要丢弃大约$\mu+U$的时间单位的信息；这个时间之后，不会有复制到达，因此，没有复制发送的危险。每次发送一个字，就将计时器Rt设为R，其中R是常数，满足选择$R > U + \mu$。如果在R时间单位内，要发送下一个字，则更新计时器Rt，否则就要关闭连接。选择发送方的计时器的值满足：当关闭连接时，永远不会接到确认消息；因此在发送包之后，至少在S时间间隔内应保持连接，S是常数，满足$S > R + 2\mu$。每次发送一个包，就将计时器St设为S，仅当$St < 0$时，才关闭连接。如果仍然剩余一些字（如，还未收到确认的字），那么在关闭连接之前，必须报告这些字。

```
R_q: (* Receive a data packet *)
     { ⟨data, s, i, w, ρ⟩ ∈ M_q }
     begin receive ⟨data, s, i, w, ρ⟩ ;
           if cr then
             if i = Exp then
               begin Rt := R ; Exp := i + 1 ; deliver w end
           else if s = true then
             begin create (Rt, Exp) ; (* cr := true *)
                   Rt := R ; Exp := i + 1 ; deliver w
             end
     end

S_q: (* Send an acknowledgement *)
     { cr }
     begin send ⟨ack, Exp, μ⟩ end

(* Close connection if Rt times out, see action Time *)
```

图3-5 接收方协议算法

在一个关闭的连接内，当接到包时，接收方利用序列开始位决定连接是否能被打开（以及包中发送的字）。当确认了以前的所有字或者报告可能丢失字后，发送方将开始位设置为真。当q接到处于打开连接状态的包时，发送所包含的字，当且仅当包的序列号等于下一个期望的序列号（存储在Exp中）。

91

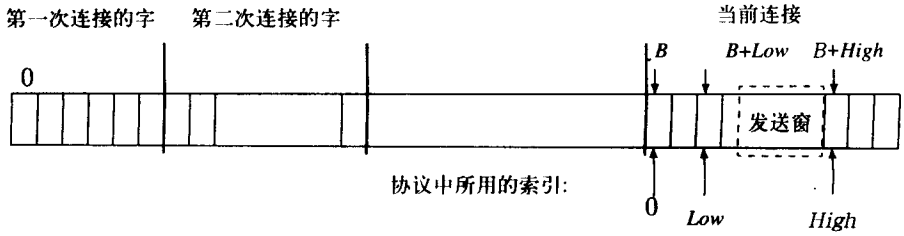
```
Loss: { m ∈ M }     (* M is either M_p or M_q *)
      begin remove m from M end

Dupl: { m ∈ M }     (* M is either M_p or M_q *)
      begin insert m in M end

Time: (* δ > 0 *)
      begin forall i do Ut[i] := Ut[i] − δ ;
            St := St − δ ;
            Rt := Rt − δ ;
            if Rt < 0 then delete (Rt, Exp) ; (* cr := false *)
            forall ⟨.., ρ⟩ ∈ M_p, M_q do
                  begin ρ := ρ − δ ;
                        if ρ < 0 then remove packet
                  end
      end
```

图3-6 协议转移补充算法

接下来讨论发送协议中变量B的作用。这是一个辅助变量，引入的目的是用于协议的正确性证明。发送方对每次连接中的字从0开始编号，为了区别不同连接中的字，在协议的分析中，对所有的字按照数字连续递增的方式编号。因此在发送方按i索引一个字时，所指的字的“绝对”数应为$B+i$，其中B是在以前连接中由p收到的包的总数。“内部”和“绝对”字号码的对应关系参见图3-7。在协议的实现中，并不存储B，发送方“忘记了”$in_p\,[0..B-1]$中的所有字。



92

图3-7　字的序列号

通信子系统用两个多集M_p和M_q表示，M_p和M_q分别表示目的地分别为p和q的包。发送方协议参见图3-4所示的算法，接收方协议参见图3-5所示的算法。图3-6所示的算法给出了补充的系统转移，它并不与进程中协议的步骤相对应。这些转移代表信道故障和时间的进行。在转移中，**Loss**和**Dupl** M代表M_p或者M_q。**Time**的作用使得系统中的所有计时器减小δ。它发生在两个离散事件应用的δ时间单位内。当接收方的计时器为0时，它的连接关闭。

3.2.2　协议的正确性证明

用一组引理和定理来证明所要求的协议性质。以下所定义的断言P_0表示：只要系统中有包存在，发送方的连接始终是打开的，并且在当前的连接中，这些包的序列号有正确的含义。

$$
\begin{aligned}
P_0 \equiv\ & cs \Rightarrow St \leqslant S && (1)\\
\wedge\ & cr \Rightarrow 0 < Rt \leqslant R && (2)\\
\wedge\ & \forall i < B + High:\ Ut\,[i] \leqslant U && (3)\\
\wedge\ & \forall <..,\ \rho> \in M_p,\ M_q:\ 0 < \rho \leqslant \mu && (4)\\
\wedge\ & <\mathbf{data},\ s,\ i,\ w,\ \rho> \in M_q \Rightarrow cs \wedge St \leqslant \rho + \mu + R && (5)\\
\wedge\ & cr \Rightarrow cs \wedge St \geqslant Rt + \mu && (6)\\
\wedge\ & <\mathbf{ack},\ i,\ \rho> \in M_p \Rightarrow cs \wedge St > \rho && (7)\\
\wedge\ & <\mathbf{data},\ s,\ i,\ w,\ \rho> \in M_q \Rightarrow (w = in_p\,[B+i] \wedge i < High) && (8)
\end{aligned}
$$

为解释式（3），在发送方不存在连接的地方，将所有配置中的$High$值设为0。

引理3.10　P_0是基于计时器协议的不变式。

证明。初始时，连接不存在，也没有包。$B=0$，这意味着P_0为真。

$\mathbf{A}_p$：式（1）保持，这是因为，对St的赋值总是使$St = S$。式（3）保持，是因为在增加$High$之前，$Ut\,[B+High]$赋值为U。式（5）、式（6）和式（7）保持，是因为只能增加St的值。式（8）保持，是因为$High$只能增加。

$\mathbf{S}_p$：式（1）保持，这是因为，St总设为S。式（4）保持，是因为在剩余包生存期（μ）的时间内发送包。式（5）保持，是因为发送包<.., μ>，St设为S且$S = R + 2\mu$。式（6）和式（7）保持，是因为St只能增加。式（8）保持，是因为新包满足$w = in_p\,[B+i]$且$i < High$。

$\mathbf{R}_p$: $\mathbf{R}_p$的行为并不改变P_0中提到的任何变量。包的去除保持式（4）和式（7）的性质。

$\mathbf{E}_p$: $\mathbf{E}_p$的行为并不改变P_0中提到的任何变量。

$\mathbf{C}_p$: $\mathbf{C}_p$的行为使式（5）、式（6）和式（7）的结论为假。但是（由式（2）、式（5）、式（6）和式（7））仅当前提为假时，$\mathbf{C}_p$是可应用的。同时$\mathbf{C}_p$改变B的值，但由式（5）和式（7），没有包传输，因此式（8）成立。 93

$\mathbf{R}_q$: Rt总被赋值R，因此式（2）保持。仅当收到包<**data**，s，i，w，ρ>时，将Rt设为R，因此式（6）保持。式（4）和式（5）蕴含$cs \wedge St \geqslant R + \mu$。

$\mathbf{S}_q$: 因为每个包在剩余包生存期μ内发送，因此式（4）保持。式（7）成立是因为，当cr为真时，且$\rho = \mu$时，发送包<**ack**，i，ρ>，因此由式（2）和式（6），$St > \mu$。

Loss: 删除包只能使前提为假，式（4）、式（5）、式（7）和式（8）保持。

Dupl: 式（4）、式（5）、式（7）和式（8）保持，仅当m在信道中时，包m的插入才是可应用的，这蕴含着，即使在插入之前，相应的子句为真。

Time: 式（1）、式（2）和式（3）保持，因为St、Rt和Ut [i]只能减小，且当Rt变到0时，接收方的连接关闭。式（4）保持，是因为ρ只能减小，且当它的ρ-域值为0时，删除包。观察**Time**将所有计时器（包括包的ρ-域）减小相同量，因此保持了所有形如$Xt \geqslant Yt + C$的断言。其中Xt和Yt是计时器，C是常数。这就表明式（5）、式（6）和式（7）保持。 □

协议的第一个要求是每个字最终被发送，或者被报告有丢失。定义谓词$Ok(i)$为$Ok(i) \Leftrightarrow error\,[i] = true \vee q$已发送$in_p\,[i]$。

可以证明，如果没有报告，协议并不损失任何字。定义断言P_1为

$$P_1 \equiv \quad P_0$$

$$\wedge\ \neg cs \Rightarrow \forall i < B : Ok(i) \tag{9}$$

$$\wedge\ cs \Rightarrow \forall i < B + Low:\ Ok(i) \tag{10}$$

$$\wedge\ \langle \mathbf{data}, true, I, w, \rho \rangle \in M_q \Rightarrow \forall i < B + I:\ Ok(i) \tag{11}$$

$$\wedge\ cr \Rightarrow \forall i < B + Exp:\ Ok(i) \tag{12}$$

$$\wedge\ \langle \mathbf{ack}, I, \rho \rangle \in M_p \Rightarrow \forall i < B + I:\ Ok(i) \tag{13}$$

引理3.11 P_1是基于计时器协议的不变式。

证明。首先观察，对某些i，一旦$Ok(i)$变为真，从那以后就不会变为假。初始时，无连接，没有包，且$B = 0$，这蕴含着P_1成立。 94

$\mathbf{A}_p$: $\mathbf{A}_p$的行为打开一个连接，因为在$Low = 0$和$\forall i < B$时，连接被打开，由式（9）可得，$Ok(i)$保持式（10）成立。

$\mathbf{S}_p$: $\mathbf{S}_p$的行为可能发送包<**data**，s，i，w，ρ>，但因为s为真，仅当$I = Low$时，由式（10）可得，式（11）保持。

$\mathbf{R}_p$: 如果接到包<**ack**，I，ρ>，Low的值可能增加。然而，式（10）保持是因为式（13），$\forall i < B + I$: 如果接到确认回答，$Ok(i)$成立。

$\mathbf{E}_p$: $\mathbf{E}_p$的应用可能使得Low的值增加，但是错误报告的产生确保式（10）被保持。

$\mathbf{C}_p$: $\mathbf{C}_p$的行为使cs为假。但是仅当$St < 0$且$Low = High$时，$\mathbf{C}_p$是可应用的。式（10）蕴含着: $\forall i < B + High$: 在$\mathbf{C}_p$的执行之前，$Ok(i)$成立，因此式（9）被保持。在执行中，式（10）的前提为假，式（5）、式（6）和式（7）蕴含着式（11）、式（12）和式

（13）的前提为假；因此式（10）、式（11）、式（12）和式（13）保持。

$\mathbf{R}_q$：首先考虑q接到包<**data**，*true*，I，w，ρ>且连接不存在（cr为假）的情况，那么，$\forall i < B + I$：由式（11），$Ok(i)$保持，且在行动中，w被发送。当$w = in_p\,[B + I]$时，由式（8），赋值语句$Exp := I + 1$保持式（12）。

现在考虑接到包<**data**，s，Exp，w，ρ>且连接打开的情况，此时Exp增加。由式（12），$\forall i < B + Exp$：在收到之前，$Ok(i)$成立，在行动中，字$w = in_p\,[B + Exp]$被发送。因此Exp的增加保持式（12）。

$\mathbf{S}_q$：由式（12），发送<**ack**，Exp，μ>保持式（13）。

Loss：**Loss**的应用只能使得子句的前提为假。

Dupl：即使在插入之前，仅当相应子句的前提为真时，包m的插入才是可应用的。

Time：式（9）~式（13）没有明确提到计时器。q所进行的包的删除和关闭只能使得式（11）、式（12）或者式（13）的前提为假。　□

在做出附加的假设后，可以证明协议规范的第一部分。没有这一假设，发送方不会报告可能丢失的字；在图3-4所示的算法中，在字的发送间隔结束后的$2\mu+R$时间内，不会出现报告，
95 但也并不是最终一定会出现。因此，我们做附加假设，在合理时间内，即在$Ut\,[B + Low] = -2\mu-R-\lambda$之前，$p$实际上会执行$\mathbf{E}_p$，

定理3.12（无损） *在p接受字之后的$U + 2\mu + R + \lambda$时间内，q发送in_p中的每个字，或者p报告。*

*证明。*在接受字$in_p\,[I]$之后，$B + High > I$成立，并且继续成立。如果在接受字$in_p\,[I]$之后的某个指定时间段内连接关闭，那么$B > I$，由式（9）可得结论成立。如果在这个时间段内，连接没有关闭且$B + Low \leqslant I$，那么，$in_p\,[I]$的发送间隔结束之后的$2\mu + R$时间内可以报告范围$B + Low\,..\,I$的所有字。这表明，发送间隔结束之后的$2\mu + R + \lambda$接收之后的$U + 2\mu + R + \lambda$报告已经发生。这也蕴含着，$I < B + Low$，由式（10），或者发送字，或者进行报告。　□

为建立协议的第二个正确性要求，必须证明所接受的每一个字都有高于以前接受的字的下标。用pr表示最新发送的字的下标（为方便起见，初始时$pr = -1$，$Ut[-1] = -\infty$）。定义断言P_2为：

$$
\begin{aligned}
P_2 \equiv \quad & P_1 \\
& \wedge \langle \mathbf{data}, s, i, w, \rho \rangle \in M_q \Rightarrow Ut\,[B + i] > \rho - \mu && (14)\\
& \wedge\ i_1 \leqslant i_2 < B + High \Rightarrow Ut\,[i_1] \leqslant Ut\,[i_2] && (15)\\
& \wedge\ cr \Rightarrow Rt \geqslant Ut\,[pr] + \mu && (16)\\
& \wedge\ pr < B + High \wedge (Ut\,[pr] > -\mu \Rightarrow cr) && (17)\\
& \wedge\ cr \Rightarrow B + Exp = pr + 1 && (18)
\end{aligned}
$$

引理3.13 *P_2是基于计时器协议的不变式。*

*证明。*初始时，M_q为空，$B + High$为0。$\neg cr$成立且$Ut\,[pr] < -\mu$，由此式(14)~式(18)成立。

$\mathbf{A}_p$：因为新接受的字得到计时器的值U，由式（3）可得，该值至少等于所有已接受字的值，因此式（15）保持。

$\mathbf{S}_p$：因为$Ut\,[B + i] > 0$，包发送时有$\rho = \mu$，因此式（14）保持。

$\mathbf{C}_p$：由式（5）、式（6）可得，当$\mathbf{C}_p$是可应用时，式（14）、式（16）和式（18）的前提为假，因此式（14）、式（16）和式（18）保持。因为B被赋值$B+High$且计时器不改

变，因此式（15）保持。因为B被赋值$B+High$且pr和cr不改变，因此式（17）保持。 96

$\mathbf{R}_q$: 当Rt设为R时（一旦接受字），$Ut\,[pr] \leqslant U$，由式（3）可得，$R \geqslant 2\mu + U$。因此式（16）保持。由式（8）可得，$pr < B + High$，cr为真，因此式（17）保持。因为Exp为$i+1$，pr为$B+i$，蕴含式（18）为真。

Time: 因为$Ut\,[B+i]$与ρ减小相同量（仅当包的删除使前提为假），因此式（14）成立。因为$Ut\,[i_1]$和$Ut\,[i_2]$减小相同值，因此式（15）成立。由于cr在执行中没变为真，Rt和$Ut\,[pr]$减小相同值，因此式（16）成立。式（17）成立是因为，如果Rt变为$\leqslant 0$，它的结论为假，由式（16），这蕴含着$Ut\,[pr]$变为$\leqslant -\mu$。因为如果cr不为假，B、Exp和pr都不改变，式（18）成立。

$\mathbf{R}_p$、$\mathbf{E}_p$和$\mathbf{S}_q$不改变式(14)～式(18)中的任何变量，同样，通过用与前面证明中所使用的相同变量，**Loss**和**Dupl**使得式(14)～式(18)成立。 □

引理3.14 由P_2可得，

$$\langle \mathbf{data}, s, i_1, w, \rho \rangle \in M_q \Rightarrow (cr \vee B + i_1 > pr)。$$

证明。由式（14）可得，$\langle \mathbf{data}, s, i_1, w, \rho \rangle \in M_q$蕴含$Ut\,[B+i_1] > \rho - \mu > -\mu$。如果$B + i_1 \leqslant pr$，由式（15），$pr < B + High$，则有$Ut\,[pr] > -\mu$，于是由式（17），$cr$为真。 □

定理3.15（有序） q所发送的字按照严格递增次序出现在in_p中。

证明。假设q接收包$\langle \mathbf{data}, s, i_1, w, \rho \rangle$并发送$w$。如果在接收之前连接不存在，$B + i_1 > pr$（由引理3.14），因此，在$in_p$中字$w$出现在位置$pr$之后。如果连接的确存在，$i_1 = Exp$，因此由式（18）可得，$B + i_1 = B + Exp = pr + 1$，这蕴含着$w = in_p\,[pr+1]$。 □

3.2.3 协议讨论

在本节的引言部分已经讨论了协议的一些扩展。通过对本节协议、引入和使用的技术做进一步讨论结束本节。 97

1. 协议的质量

无损和有序要求都是安全性质。如果只给出简单的解决方法，即协议并不发送或接收任何包，当丢失信息时，也不报告每个字。这样的协议不能完成任何形式的从发送方到接收方的数据传输。不能说它是一种“好的”方法。

好的解决问题的方法不仅要满足无损和有序要求，而且当丢失发生时，尽可能少地报告。为了做到这一点，在本节的协议中增加一种机制，使得它可以重复地（直到发送间隔结束）发送每个字直到收到确认消息。发送间隔必须足够长，才能使某一个字复制传输若干次，这样丢失一个字的概率就变得很小了。

接收方的机制是，在打开连接的状态，无论什么时候发送或者接收到一个包，就触发发送确认消息。

2. 有限的序列号

可以限制协议中所用的序列号，其证明类似于平衡滑动窗口协议[Tel91b，3.2节]中的引理3.9。为此，必须假设对p的接收速率进行限定，满足如果以前的第L个字至少过了$U + 2\mu + R$的时间，才能接收下一个字。将阈值

$$\{(High < L) \vee (Ut[B + High - L] < -R - 2\mu)\}$$

加到A_p中。在这一假设下，接收数据包的序列号在以Exp为中心的$2L$范围内，确认的序列号在以$High$为中心的L范围内。因此，序列号可以模$2L$传输。

3. 行为设计与不变式

利用断言方法，关于通信协议的推理可归约到公式的操作。公式操作是一种安全的技术，因为每一步都经过仔细地证明，因此，推理中产生错误的可能性很小。存在着失去对协议以及协议与所考虑公式的关系的总体看法的风险。应从实际和形式两方面来理解协议的设计问
98 题。Fletcher和Watson[FW78]认为，控制信息必须受到“保护”，在这种意义上，即它不能被包的丢失或者复制所改变，这是实际的观点。在断言方法中，对于控制信息“意义”的验证反映在选取某种断言作为不变式。这些不变式的选取以及转移的设计构成了一种形式的观点。根据公式的“形式”，可以重新阐述Fletcher和Watson的结果，即选取哪些公式作为协议的不变式以减少包的丢失和复制。

所有与包有关的P_2的不变式子句具有形式

$$\forall m \in M : A(m)$$

显而易见，这个子句通过包的复制或者丢失保持。在后面的章节中，我们会看到不变式具有更一般的形式，例如，

$$\sum_{m \in M} f(m) = K$$

或者

$$\text{条件} \Rightarrow \exists m \in M:\ A(m)。$$

由于包的丢失或者复制，具有这些形式的断言可以为假。因它不能用于必须容忍这些错误的算法的正确性证明中。

类似的观察可应用到**Time**行为的不变式形式中。这种行为保持所有具有形式

$$Xt > Yt + C$$

的断言。这里Xt和Yt是计时器，C是常数。

4. 不精确的计时器

被**Time**的行为模型理想化的计时器，它在δ时间单位正好减小δ。但实际上，计时器会遭受不精确之苦，称之为*漂移*（drift）。总是假设漂移是$\in$-有限的，对于已知的常数$\in$。它表明，在δ时间单位内，计时器减小量为δ'，满足$\delta/(1+\in) < \delta' < \delta \times (1+\in)$。（一般选取$\in$的阶为$10^{-5}$或者$10^{-6}$。）在图3-8所示的算法中，用**Time-**$\in$模拟计时器的行为。

可见，**Time**保持了特殊形式$Xt \geqslant Yt + C$的断言，因为计时器在不等式两边严格减小相同量，并且$Xt \geqslant Yt + C$蕴含$(Xt - \delta) \geqslant (Yt - \delta) + C$。对于**Time-**$\in$可做类似观察。对于实数$Xt$、$Yt$、$\delta$、$\delta'$、$\delta''$、$r$和$c$，满足$\delta > 0$和$r > 1$，

$$(Xt > r^2 Yt + c) \wedge \left(\frac{\delta}{r} < \delta' < \delta \times r\right) \wedge \left(\frac{\delta}{r} < \delta'' < \delta \times r\right)$$

蕴含

$$(Xt - \delta') > r^2 (Yt - \delta'') + c\ 。$$

因此，**Time-∈**保持形如

$$Xt \geq (1+\in)^2 Yt + c \text{ 。}$$

的断言。

通过修改不变式，现在协议适合于具有漂移计时器的情形。为使其他的行为也保持修改过的不变式，协议中的R和S必须满足

$$R \geq (1+\in)\left((1+\in)U + (1+\in)^2\mu\right) \text{ 且 } S \geq (1+\in)\left(2\mu + (1+\in)R\right) \text{ 。}$$

除了修改的常数，协议中其余地方保持不变，图3-9中给出了它的不变式。

Time-ϵ: $\{\ \delta > 0\ \}$
begin (* The timers in p decrease by δ' *)
$\delta' := \ldots$; (* $\frac{\delta}{(1+\epsilon)} \leq \delta' \leq \delta \times (1+\epsilon)$ *)
forall i **do** $Ut[i] := Ut[i] - \delta'$;
$St := St - \delta'$;
(* The timers in q decrease by δ'' *)
$\delta'' := \ldots$; (* $\frac{\delta}{(1+\epsilon)} \leq \delta'' \leq \delta \times (1+\epsilon)$ *)
$Rt := Rt - \delta''$;
if $Rt \leq 0$ **then** delete (Rt, Exp) ;
(* The ρ-fields run exactly *)
forall $\langle .., \rho\rangle \in M_p, M_q$ **do**
begin $\rho := \rho - \delta$;
if $\rho \leq 0$ **then** remove packet
end
end

图3-8 修改的Time行为算法

$$
\begin{aligned}
P_2' \equiv\ & cs \Rightarrow St \leq S && (1')\\
\wedge\ & cr \Rightarrow 0 < Rt \leq R && (2')\\
\wedge\ & \forall i < B + High:\ Ut[i] \leq U && (3')\\
\wedge\ & \forall \langle .., \rho\rangle \in M_p, M_q : 0 < \rho \leq \mu && (4')\\
\wedge\ & \langle \mathbf{data}, s, i, w, \rho\rangle \in M_q \Rightarrow cs \wedge St \geq (1+\epsilon)(\rho + \mu + (1+\epsilon)R) && (5')\\
\wedge\ & cr \Rightarrow cs \wedge St \geq (1+\epsilon)((1+\epsilon)Rt + \mu) && (6')\\
\wedge\ & \langle \mathbf{ack}, i, \rho\rangle \in M_p \Rightarrow cs \wedge St > (1+\epsilon) \times \rho && (7')\\
\wedge\ & \langle \mathbf{data}, s, i, w, \rho\rangle \in M_q \Rightarrow (w = in_p[B+i] \wedge i < High) && (8')\\
\wedge\ & \neg cs \Rightarrow \forall i < B:\ Ok(i) && (9')\\
\wedge\ & cs \Rightarrow \forall i < B + Low:\ Ok(i) && (10')\\
\wedge\ & \langle \mathbf{data}, true, I, w, \rho\rangle \in M_q \Rightarrow \forall i < B + I:\ Ok(i) && (11')\\
\wedge\ & cr \Rightarrow \forall i < B + Exp:\ Ok(i) && (12')\\
\wedge\ & \langle \mathbf{ack}, I, \rho\rangle \in M_p \Rightarrow \forall i < B + I:\ Ok(i) && (13')\\
\wedge\ & \langle \mathbf{data}, s, i, w, \rho\rangle \in M_q \Rightarrow Ut[B+i] > (1+\epsilon)(\rho - \mu) && (14')\\
\wedge\ & i_1 \leq i_2 < B + High \Rightarrow Ut[i_1] \leq Ut[i_2] && (15')\\
\wedge\ & cr \Rightarrow Rt \geq (1+\epsilon)((1+\epsilon)Ut[pr] + (1+\epsilon)^2\mu) && (16')\\
\wedge\ & pr < B + High \wedge Ut[pr] > -(1+\epsilon)\mu \Rightarrow cr && (17')\\
\wedge\ & cr \Rightarrow B + Exp = pr + 1 && (18')
\end{aligned}
$$

图3-9 带有计时器漂移协议的不变式

定理3.16 P'_2是基于计时器协议的不变式，具有∈-有限的计时器漂移。协议满足无损和

有序要求。

习题

3.1节

3.1　证明如果在公平性假设F1和F2中，只有F2成立，平衡滑动窗口协议并不满足最终发送要求。

3.2　证明在平衡滑动窗口协议中，如果$L = 1$，a_p和a_q分别初始化为$-l_q$和$-l_p$，那么，$a_p + l_q = s_p$和$a_q + l_p = s_q$总是成立。

3.2节

3.3　在基于计时器的协议中，发送方报告某个字可能丢失，而实际上接收方已经正确地发送了这个字。

（1）在这种现象出现的地方，描述协议的一次执行。

（2）是否能设计一种协议，使得发送方在限定的时间内产生错误报告，当且仅当接收方没有发送该字？

99 ~ 101

3.4　假设由于错误的时钟设备，接收方可能未能及时关闭它的连接。描述基于计时器协议的一次计算，其中发送方未能报告字的丢失。

3.5　描述基于计时器协议的一次计算，其中收到序列号大于0的包时，接收方打开一个连接。

3.6　**Time-∈**的行为不能模拟剩余包生存期中的漂移。为什么不能？

3.7　证明定理3.16。

3.8　网络工程师想要利用基于计时器的协议，但却要通过对E_p做下列更改以允许提早报告字的丢失。

E_p:　(* *Generate error report for possibly lost word* *)

{*Ut* [*B* + *Low*] < 0 }

begin *error* [*B* + *Low*] := *true*; *Low* := *Low* + 1 **end**

102

更改过的协议仍然满足无损和有序的要求吗？或者还必须再做其他修改？

给出这个更改的优缺点。

第4章 路 由 算 法

进程（计算机网络中的一个节点）一般不能与其他进程通过信道直接相连。节点只能将信息包直接发给节点的一个子集，这个子集称作节点的近邻（neighbor）。路由是用于描述判定过程的术语，通过路由，节点可以选择一个（或多个）近邻进行包的转发，最终到达目的节点。路由算法的设计目标是为每一个节点产生一个决策过程来完成这项功能，并保证每个包的传输。

很显然，必须将网络拓扑结构信息存储在节点中作为（局部）判定过程的工作基础。称此信息为*路由表*。有了这些路由表的指引，在算法上就可以将路由问题分为两部分，表结构的定义与算法设计有关。

（1）*路由表计算* 网络初始化时要计算路由表。如果网络的拓扑结构发生变化，就要及时更新路由表。

（2）*包转发* 当要在网络中发送包时，必须利用路由表进行包的转发。

评判一个“好的”路由方法应包括以下几个方面:

（1）*正确性* 算法必须将提供给网络的每个包传递至其最终目的节点。

（2）*效率* 算法必须通过“好的”路径发送包，例如，经历延迟小的路径，确保整个网络的高吞吐量。如果算法利用“最好的”路径，则称算法是*最优的*。 103

（3）*复杂度* 路由表的计算方法必须利用尽可能少的消息、时间及存储空间。复杂度的其他方面还有，如何进行快速路由决策，如何快速准备传输包等。但本章对这些方面关注较少。

（4）*健壮性* 在网络拓扑结构改变时（通道或节点的增加或删除），算法能够更新路由表。以便能够在改变的网络中执行路由功能。

（5）*适应性* 算法通过更改路由表，来平衡信道和节点的负载以避免那些通过繁忙信道和节点的路径，而选择某些负载较轻信道和节点。

（6）*公平性* 算法必须为每个用户提供相同程度的服务。

这些准则有时是相互矛盾的，如果取准则的子集，大多数算法会执行的很好。

通常用图表示网络，图中的顶点表示网络中的节点。如果两节点在网络中相邻（即，它们之间有通信信道），则两节点在图中有一条边相连。算法的最优性取决于图中最优路径；有几种表示最优的方法，每一种都有自己的路由算法:

（1）*最小跳数* 可以用路径中的跳数（节点之间通过的信道数或者步数）来度量使用路径的成本。最小跳数路由算法利用具有最少可能跳数的路径。

（2）*最短路径* 静态地为每一信道赋以权值（非负）。用路径中信道的权值之和度量路径的代价。最短路径算法利用具有最小代价的路径。

（3）*最小延迟* 动态地为每一信道赋以权值，这取决于信道中的信息量。最小延迟算法反复地修改路由表，使得总能选择（接近）最小总延迟的路径。信道的延迟与实际的信息量有关，并且网络中所传输的各种包相互影响，4.1节讨论信息量对路由算法的影响。 104

路径最优性的其他表示方法可能在特定应用中有用，这里不做讨论。

本章概述

本章中讨论下面一些内容。4.1节描述的结果表明，至少对于最小跳数路由和最短路径路由，可用一棵根向d的生成树将所有包最优地路由至同一目的节点d。因而，当进行路由决策时，包的源节点可以忽略。

4.2节描述了具有加权信道的静态网络路由表计算的算法。算法分布地计算每一对节点的最短路径，并在每个源节点中存储通往各目的节点的路径中的第一个近邻。这种算法的缺点是，当网络的拓扑结构发生变化时，需要重复所有计算。因此算法是不健壮性的。

4.3节讨论的变更算法没有这种缺点。通过部分再计算路由表，它能适合故障信道或者恢复信道。为使分析简单，将它表示成最小跳数路由算法，即，用跳数来度量路径代价。有可能修改变更算法，使其能够处理发生故障或进行恢复的加权信道。

4.2和4.3节的路由算法，使用包含所有可能的目的节点的路由表。对于节点较小的大网络来说，对存储空间的需求巨大。4.4节讨论一些路由策略，包括如何将拓扑信息编码在节点的地址中，以便利用更短的路由表或做更少的表查找。这些所谓的“压缩”路由算法通常并没有用到最优路径。此外，讨论了树模式路由算法、区间路由算法和前缀路由算法。

4.5节讨论了分层路由方法。在这些方法中，网络被分成一些（连通的）簇，对簇内路由和簇间路由作出区分。这种范型可以减少一个路径中必须做的路由决策的数量，或者减少各节点中存储路由表所需要的空间的大小。

4.1 基于目的节点的路由

转发包时所做的路由决策通常基于包的目的节点（和路由表中的内容），与包的原始发送
105 方（源节点）无关。路由可以忽视源节点，仍然可以利用最优路径，正如本节结果所蕴含的。虽然其结果并不依赖于为路径选取特定的优化准则，但是下述假设必须成立。（回忆，如果一条路径所包含的每个节点至多出现一次，称这条路径是简单路径。如果第一个节点等于最后一个节点，称路径是一个回路。）

（1）经过路径P发送包的代价与实际利用的路径无关。尤其是其他消息利用P的边时。

这种假设使得我们可以把利用路径P的代价看作是路径的函数。用$C(P) \in \mathbb{R}$表示P的代价。

（2）两条路径连接的代价等于被连接路径代价的和。即对所有$i = 0, \ldots, k$，

$$C(<u_0, u_1, \ldots, u_k>) = C(<u_0, \ldots, u_i>) + C(<u_i, \ldots, u_k>)。$$

因而，空路径$<u_0>$（从u_0到u_0的路径）满足$C(<u_0>) = 0$。

（3）图中不包含负代价的回路。

（最小跳数和最短路径代价准满足这些准则。）如果不存在从u到v具有更小代价的路径，称从u到v的路径是最优的。观察可得，最优路径并不总是惟一的，可能存在不同的路径都具有相同的（最小）代价。

引理4.1　设u、v位V中的顶点，如果G中存在一条从u到v的路径，那么存在一条最优简单路径。

证明。因为只有有限条简单路径，因此存在一条从u到v且具有最小代价的简单路径，记为S_0。即，对于从u到v的每一条简单路径P'，$C(S_0) \leqslant C(P')$。其余要证明，$C(S_0)$是每一条

(非简单)路径代价的下界。

设$V=\{v_1, \ldots, v_N\}$。通过不断地从P中去除包含v_1、v_2的回路，可以证明，对于从u到v的每一条路径P，存在一条满足$C(P')\leqslant C(P)$的简单路径P'。令$P_0=P$，对于$i=1, \ldots, N$，构造路径P_i如下。如果v_i在P_{i-1}中至多出现一次，那么，$P_i=P_{i-1}$。否则，记$P_{i-1}=<u_0, \ldots, u_k>$，设$u_{j_1}$和$u_{j_2}$分别是$v_i$在$P_{i-1}$中的第一次和最后一次的出现，令

$$P_i=<u_0,\ldots,u_{j_1}\left(=u_{j_2}\right),u_{j_2+1},\ldots,u_k>。$$

通过构造，P_i是从u到v的路径，包含所有节点$\{v_0, \ldots, v_i\}$至多一次，因此P_N是从u到v的简单路径。P_{i-1}由P_i和回路$Q=u_{j_1}, \ldots, u_{j_2}$组成，因此$C(P_{i-1})=C(P_i)+C(Q)$。又因为不存在 106
负权值的回路，这蕴含着，$C(P_i)\leqslant C(P_{i-1})$，因此$C(P_N)\leqslant C(P)$。

由S_0的选取，$C(S_0)\leqslant C(P_N)$，由此可得，$C(S_0)\leqslant C(P)$成立。 □

如果G包含负权值的回路，则最优路径未必存在。对于每一条路径，都存在另一条代价比它更小的包含负回路的路径。下面的定理，假设G是连通的(对于不连通图，可将定理单独用到它的每一连通分量上)。

定理4.2 *对于$d\in V$，存在一棵树$T_d=(V, E_d)$，满足$E_d\subseteq E$，且对每一节点$v\in V$，T_d中从v到d的路径是G中从v到d的一条最优路径。*

*证明。*设$V=\{v_1, \ldots, v_N\}$。我们用归纳法构造一组具有下列性质的树$T_i=(V_i, E_i)$(对于$i=0, \ldots, N$)

(1) 每一T_i是G的一棵子树，即，$V_i\subseteq V$，$E_i\subseteq E$，且T_i是一棵树。

(2) 每一T_i(对于$i<N$)是T_{i+1}的一棵子树。

(3) 对于所有$i>0$，$v_i\in V_i$，$d\in V_i$。

(4) 对于所有$w\in V_i$，T_i中从w到d的简单路径是G中从w到d的一条最优路径。

这些性质蕴含着，T_N满足T_d的要求。

为构造树的序列，设$V_0=\{d\}$和$E_0=\varnothing$。树T_{i+1}构造如下：选择从v_{i+1}到d的一条最优简单路径$P=<u_0, \ldots, u_k>$，l是满足$u_l\in T_i$的最小下标(这样的最小下标l存在，是因为$u_k=d\in T_i$；可能$l=0$)。现设

$$V_{i+1}=V_i\cup\left\{u_j: j<l\right\}\text{ 且 }E_{i+1}=E_i\cup\left\{(u_j,u_{j+1}): j<l\right\}。$$

(图4-1显示了这个构造过程。)易证，T_i是T_{i+1}的一棵子树，并且$v_{i+1}\in V_{i+1}$。为证T_{i+1}是一棵树，由构造可得，T_{i+1}是连通的，并且节点数比边数多一个。(T_0具有滞后性质，在每一阶段所加入的节点数和边数一样多。)

下面要证明，对于所有$w\in V_{i+1}$，T_{i+1}中从w到d的(惟一)路径是G中从w到d的一条最优路径。对于节点$w\in V_i\subset V_{i+1}$成立，这是因为，T_i是T_{i+1}的一棵子树。T_{i+1}中从w到d的路径与T_i中的路径相同，这是最优路径。现设$w=u_j$，$j<l$是$V_{i+1}\setminus V_i$中的节点。Q表示T_i中从u_i到d的路径，那么在T_{i+1}中，u_j通过与Q连接的路径$<u_j, \ldots, u_l>$连到d。下面证明这就是G中的最优路径。首先，P的后缀$P'=<u_l, \ldots, u_k>$是从u_l到d的最优路径，即，$C(P')=C(Q)$：Q的最优性蕴含着$C(P')\geqslant C(Q)$，而$C(Q)<C(P')$蕴含(由路径代价的增加性)路径$<u_0, \ldots, u_l>$与Q的连接要比P具有更小的代价，这与P的最优性矛盾。现假设从u_j到d的路径R要比$<u_j, \ldots, u_l>$与Q连接的路径代价要小。那么，由前面的观察，R的代价比P的后缀$<u_l, \ldots, u_k>$的代价小， 107

这蕴含着（由路径代价的增加性），< u_0，…，u_j>与R的连接要比P具有更小的代价，这与P的最优性矛盾。 □

根向d的生成树称为d的汇集树，如果一棵树具有定理4.2中给定的性质，则称该树为**最优汇集树**（optimal sink tree）。最优汇集树的存在性表明，如果只考虑路由算法，其中的转发机制如算法4-2所示，则不会是对最优性的折衷。在算法中，$table_lookup_u$是只有一个变量的局部过程，返回u的近邻（在参考路由表之后）。当所有目的节点为d的包，经过根向d的生成树最优路由时，如果对于所有$u \neq d$，$table_lookup_u$ (d)返回生成树T_d中u的父节点，则转发是最优的。

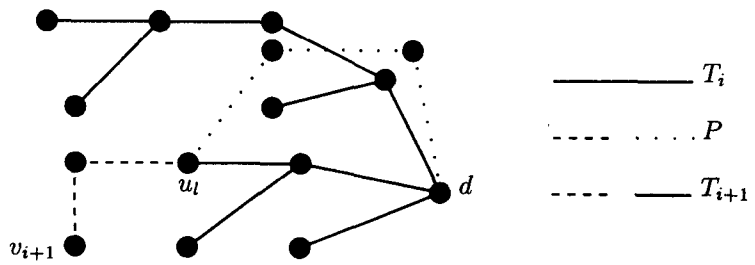


图4-1　T_{i+1}的构造

当转发机制具有这种形式，并且拓扑结构不再（或进一步）变化，使用下面结果就能验证路由表的正确性。称路由表**包含回路**（对目的节点d），如果存在节点u_1，…，u_k满足，对所有i，$u_i \neq d$，对所有$i < k$，$table_lookup_{u_i}(d) = u_{i+1}$，且$table_lookup_{u_k}(d) = u_1$。如果对任何目的节点$d$，路由表不包含回路，称它是**无回路的**（cycle-free）。

108

```
(* A packet with destination d was received or generated at node u *)
if d = u
  then deliver the packet locally
  else send the packet to table_lookup_u(d)
```

图4-2　基于目的节点的转发（节点u）算法

引理4.3　*当且仅当路由表无回路，转发机制将每一包传递至其目的节点。*

证明。如果路由表中含有到目的节点d的回路，如果它的源节点在回路中，则节点d的包永远不能被传递至目的地。

假设路由表无回路，目的节点为d（源节点为u_0）的包经由u_0，u_1，u_2，…转发。如果在序列中同一节点出现两次，记为$u_i = u_j$，那么，路由表包含回路，即< u_i，…，u_j>，这与路由表无回路的假设矛盾。因此，每一节点出现至多一次，蕴含着序列是有限的，设结束点为u_k（$k < N$）。按照转发程序，序列只能以d结束，即$u_k = d$，因此序列在至多$N-1$次跳跃后到达目的节点。 □

在某些路由算法，计算路由表的过程中，路由表并不是没有回路的，但是这种情况只能发生在表的计算阶段完成之后。当采用这种算法时，在路由表的计算过程中，包可能遍历回路，但如果拓扑结构不再变化后，表计算完成后，在至多$N-1$次跳跃后也可到达其目的节点。

如果拓扑结构不断变化，即，网络不断地受到拓扑上不断变化的无穷序列制约，即使在更新过程中表无回路，包也未必能够到达目的节点。参见习题4.1。

最小延迟分支路由算法

如果要求路由经过最小延迟路径，且信道的延迟取决于它的利用率（本节开始所作的假设（1）无效），则不能只用路径函数作为评价路径代价的惟一因素。此外，还必须考虑信道上的信息量。为避免路径上的拥塞（其结果是更高延迟），通常有必要将具有同一源节点-目的节点对的包沿不同的路径发送，将该对的信息量分裂成一个节点或者多个中间节点的说明参见图4-3。利用不同路径到达同一个目的节点的路由方法称为*多路径方法*或者*分支路由方法*。由于分支路由方法较复杂，本章暂时不予考虑。 109

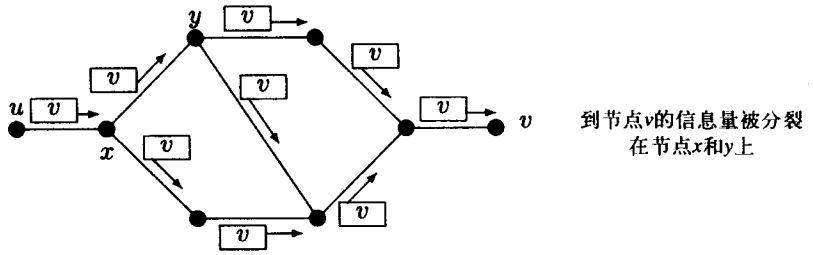


图4-3 分支路由例子

4.2 所有点对之间的最短路径问题

本节讨论Toueg[Tou80a]提出的同时计算网络中所有节点的路由表的算法。该算法计算每一对节点（u，v）从u到v的最短路径，并将该路径的第一个信道存储在u中。计算图中任意两节点之间最短路径的问题称为所有点对之间的最短路径。针对此问题的Toueg的分布式算法是基于Floyd-Warshall提出的集中式算法[CLR90，26.4节]。4.2.1节讨论Floyd-Warshall算法，4.2.2节接着讨论Toueg的分布式算法，4.2.3节简要讨论其他一些关于所有点对之间最短路径问题的算法。

4.2.1 Floyd-Warshall算法

给定加权图$G=(V, E)$，用ω_{uv}表示边uv的权值。不必假设$\omega_{uv}=\omega_{vu}$，但假设图中不包含总权值为负的回路。路径$<u_0, \ldots, u_k>$的权值定义为$\sum_{i=0}^{k-1}\omega_{u_iu_{i+1}}$。$d(u, v)$表示从$u$到$v$的距离，是从$u$到$v$的任意路径的最小权值（如果这样的路径不存在，则为∞）。所有点对最短路径问题 110
是为每个u和v计算$d(u, v)$。（在4.2.2节对此算法进行扩充，以便存储该路径的第一条边）。

为计算所有距离，Floyd-Warshall算法采用S-路径表示法，即存在路径，其上的所有中间节点属于V的子集S。

定义4.4 *设S是V的子集。称路径$<u_0, \ldots, u_k>$是S-路径，如果对于所有i，$0<i<k$，$u_i\in S$。定义从u到v的S-距离$d^S(u, v)$为从u到v的任一最小加权S-路径（如果这样的路径不存在，则为∞）。*

算法从所有$\varnothing$-路径开始，逐步计算更大子集S的S-路径，直到考虑完所有V-路径。有如下观察。

命题4.5　对于所有u和S，$d^S(u, u) = 0$。且对于$u \neq v$，S-路径满足下列规则。

（1）存在一条从u到v的$\varnothing$-路径，当且仅当$uv \in E$。

（2）如果$uv \in E$，那么，$d^{\varnothing}(u, v) = \omega_{uv}$，否则，$d^{\varnothing}(u, v) = \infty$。

（3）如果$S' = S \cup \{w\}$，那么从u到v的简单路径S'-路径是从u到v的S-路径，或者是u到w的S-路径与w到v的S-路径的连接。

（4）如果$S' = S \cup \{w\}$，那么$d^{S'}(u, v) = \min(d^S(u, v), d^S(u, w) + d^S(w, v))$。

（5）从u到v存在路径，当且仅当从u到v存在V-路径。

（6）$d(u, v) = d^V(u, v)$

证明。对于所有u和S，$d^S(u, u) \leqslant 0$，因为空路径（由0条边组成）是u到u且权值为0的S-路径。没有一条路径有更小的权值，因为G不包含负权值的回路，因此$d^S(u, u) = 0$。

对于（1）：一条$\varnothing$-路径没有中间节点，因此，从u到v的$\varnothing$-路径仅由信道uv组成。

对于（2）：由（1）直接可得。

对于（3）：从u到v的简单S'-路径将节点w做为中间节点包含一次或者0次。如果w不是S'-路径的中间节点，则它就是一条S-路径，否则它就是两条S-路径的连接，一条连到w，一条来自w。

对于（4）：由引理4.1可得，（如果从u到v存在S'-路径）就存在一条从u到v长为$d^{S'}(u, v)$的简单S'-路径，由（3）可得，这蕴含着$d^{S'}(u, v) = \min\{ d^S(u, v), d^S(u, w) + d^S(w, v) \}$。

对于（5）：每一条V-路径是一条路径。反之亦然。

对于（6）：每一条V-路径是一条路径。反之亦然。因此，一条最优的V-路径也是一条最
111 优路径。 □

利用命题4.5，不难设计所有点对之间最短路径问题的“动态规划算法”。参见图4-4所示的算法。算法首先考虑$\varnothing$-路径，然后逐步计算更大子集S的S-路径（通过“枢轴”循环增大S），直到考虑完所有路径。

```
begin (* Initialize S to ∅ and D to ∅-distance *)
      S := ∅ ;
      forall u, v do
            if u = v then D[u, v] := 0
            else if uv ∈ E then D[u, v] := ω_uv
                           else D[u, v] := ∞ ;
      (* Expand S by pivoting *)
      while S ≠ V do
         (* Loop invariant: ∀u, v : D[u, v] = d^S(u, v) *)
         begin pick w from V \ S ;
               (* Execute a global w-pivot *)
               forall u ∈ V do
                     (* Execute a local w-pivot at u *)
                     forall v ∈ V do
                           D[u, v] := min ( D[u, v], D[u, w] + D[w, v] ) ;
               S := S ∪ {w}
         end   (* ∀u, v : D[u, v] = d(u, v) *)
end
```

图4-4　Floyd-Warshall算法

定理4.6 图4-4所示的算法的计算复杂度为$\Theta(N^3)$。

证明。算法开始时，如果$u = v$，$D[u, v] = 0$；如果$uv \in E$，$D[u, v] = \omega_{uv}$；否则，$D[u, v] = \infty$，且$S = \varnothing$。由命题4.5的(1)和(2)部分，$\forall u, v$: $D[u, v] = d^S(u, v)$成立。在以w为枢轴节点（pivot node）的一轮轴循环中，用w扩展集合S，对$D[u, v]$的赋值（由命题4.5的(3)和(4)部分）确保断言$\forall u, v$: $D[u, v] = d^S(u, v)$仍然是循环不变式。当$S = V$时，程序终止。即，（由命题4.5的(5)、(6)部分和循环不变式），S-距离就是最终所求距离。

主循环执行N次，包含N^2个操作（执行可并行或串行），这蕴含定理中陈述的结论。 □

112

4.2.2 Toueg最短路径算法

基于上节描述的Floyd-Warshall算法，Toueg[Tou80a]给出了计算路由表的分布式算法。可以证明，Floyd-Warshall算法适合于作此用途，即，在分布式系统中，它的假设是现实的。算法最重要的假设是图中不包含负权值的回路。分布式系统中的假设的确如此，每一信道的代价为正值。以下作进一步假设。

A1 网络中每一回路具有正权值。

A2 网络中每一节点初始时知道所有节点的标识（集合V）。

A3 网络中每一节点知道它的近邻（对于节点u，存储在$Neigh_u$中）节点以及其传发信道的权值。

如果先讨论Toueg算法的简化版本，“简单算法”（图4-5所示的算法），再讨论Toueg算法的正确性（图4-6所示的算法），就会更容易理解。

```
var S_u  : set of nodes ;
    D_u  : array of weights ;
    Nb_u : array of nodes ;

begin S_u := ∅ ;
      forall v ∈ V do
            if v = u
              then begin D_u[v] := 0 ; Nb_u[v] := udef end
              else if v ∈ Neigh_u
                     then begin D_u[v] := ω_uv ; Nb_u[v] := v end
                     else begin D_u[v] := ∞ ; Nb_u[v] := udef end ;
      while S_u ≠ V do
        begin pick w from V \ S_u ;
              (* All nodes must pick the same node w here *)
              if u = w
                then "broadcast the table D_w"
                else "receive the table D_w" ;
              forall v ∈ V do
                    if D_u[w] + D_w[v] < D_u[v] then
                      begin D_u[v] := D_u[w] + D_w[v] ;
                            Nb_u[v] := Nb_u[w]
                      end ;
              S_u := S_u ∪ {w}
        end
end
```

图4-5 简单算法（节点u）

1. 简单算法

为将Floyd-Warshall算法变成分布式算法，需将算法中变量、操作划分到网络中的节点上。变量$D[u, v]$是属于节点u的变量；按照惯例，可用下标表示：从现在起，记作$D_u[v]$。对$D_u[v]$的赋值必须由节点u执行，在这个操作中，当需要节点w的一个变量值时，这个值必须发送给u。在Floyd-Warshall算法中，所有节点必须利用来自枢轴节点（循环体中的w）的信息，枢轴节点通过（广播）操作将此信息同时发送到所有节点中。最终，算法不仅保存最短S-路径的长度（在变量$D_u[v]$中），而且保存这条路径上的第一条信道（在变量$Nb_u[v]$中），通过这一操作使算法得以扩充。

网络中回路权值为正的假设可用于证明每轮循环后路由表中不存在回路。

引理4.7　给定S和w，假定

（1）对于所有u，$D_u[w]= d^S(u, w)$，且

（2）如果$d^S(u, w) <\infty$，且$u \neq w$，那么$Nb_u[w]$是到w的最短S-路径上的第一条信道。

那么，有向图$T_w = (V_w, E_w)$是一棵根向w的树，其中

$(u \in V_w \Leftrightarrow D_u[w] <\infty)$，且$(ux \in E_w \Leftrightarrow (u \neq w \wedge Nb_u[w] = x))$。

证明。对于$u \neq w$，如果$D_u[w]< \infty$，那么$Nb_u[w] \neq udef$且$D_{Nb_u[w]}[w] < \infty$。于是对于每个$u \in V_w$，$u \neq w$，存在一个节点x，使得$Nb_u[w] = x$，这个节点满足$x \in V_w$。

对于V_w中每个$u \neq w$的节点，E_w中存在一条边，T_w中的节点数比边数多1，这足以表明T_w中不存在回路。因为$ux \in E_w$蕴含，$d^S(u, w) =\omega_{ux} + d^S(x, w)$，$T_w$中回路$< u_0, u_1, \ldots, u_k >$的存在性蕴含

$$d^S(u_0,w) = \omega_{u_0u_1} + \omega_{u_1u_2} + \cdots + \omega_{u_{k-1}u_0} + d^S(u_0,w) \ ,$$

113
~
114

即，$\omega_{u_0u_1} + \omega_{u_1u_2} + \cdots + \omega_{u_{k-1}u_0} = 0$，这与每一回路有正权值的假设相矛盾。 □

现在可以直接将Floyd-Warshall算法转变成图4-5所示的算法。每一节点初始化它自己的变量，并执行N次主循环。算法到此并没有完结，因为还没有表明如何有效地广播枢轴节点表。但现在可以认为w执行操作“broadcast the table D_w”，其他节点执行操作“receive the table D_w”，每个节点可以访问表D_w。

为了保证节点以同一次序选择枢轴节点，还须注意操作“pick w from $V \setminus S$”。假设所有节点都预先知道V，可以简单地假设以某种规定次序选择节点（如节点名字的字母次序）。

下面定理表明简单算法的正确性。

定理4.8　在主循环迭代N次后，图4-5所示的算法在每一个节点上终止。当算法终止在节点u时，$D_u[v]= d(u, v)$，并且如果从u到v的路径存在，那么$Nb_u[v]$是从u到v的最短路径上的第一条信道，否则，$Nb_u[v] = udef$。

证明。由Floyd-Warshall算法（定理4.6）得出$D_u[v]$的终止性和正确性。由于每次对$D_u[v]$赋值时，都更新$Nb_u[v]$的值，因此上述关于$Nb_u[v]$说法成立。 □

2. 改进算法

为了提高图4-5所示的算法中广播的效率，Toueg观察到，在以w为枢轴的那一轮开始时，对于$D_u[w] = \infty$的节点u，在该轮的执行中，并不改变它的表。如果$D_u[w] = \infty$，则对每个v，$D_u[w] + D_w[v]< D_u[v]$为假。因此，只有属于T_w（在以w为中心的这轮循环的开始时）的节点需要接收w的表，可以只通过属于树T_w的信道发送路由表D_w来有效地进行广播，即，w将D_w发送给它在T_w中的子节点，T_w中接收该表（从其在T_w中的父节点）的每个节点，把表转发给它在

T_w中的子节点。

在以w为枢轴节点的那一轮的开始，满足$D_u[w] < \infty$的节点u知道它的父节点（T_w中）是谁， 115
但不知道它的子节点。因此，每一节点v必须向它的每个近邻u发送消息，告知u，是否v是u在T_w中的子节点。图4-6所示的算法给出了完整算法。当一个节点得知它的近邻中哪些是它在T_w中的子节点时，就可以参与转发w表的过程。算法中利用了三种类型的消息：

（1）如果x是u的父节点（T_w中），则在w为枢轴节点的那一轮的开始，u将消息<**ys**，w>（**ys**代表“你的子节点”）发送至x。

（2）如果x不是u的父节点（T_w中），则在w为枢轴节点的那一轮的开始，u将消息<**nys**，w>（**nys**代表“不是你的子节点”）发送至x。

（3）在以w为中心的那一轮中，发送消息<**dtab**，w，D>。经过T_w的每条边把D_w的值传输到需要该值的每一节点中。

```
var S_u   : set of nodes ;
    D_u   : array of weights ;
    Nb_u  : array of nodes ;

begin S_u := ∅ ;
      forall v ∈ V do
            if v = u
              then begin D_u[v] := 0 ; Nb_u[v] := udef end
              else if v ∈ Neigh_u
                     then begin D_u[v] := ω_uv ; Nb_u[v] := v end
                     else begin D_u[v] := ∞ ; Nb_u[v] := udef end ;
      while S_u ≠ V do
        begin pick w from V \ S_u ;
              (* Construct the tree T_w *)
              forall x ∈ Neigh_u do
                     if Nb_u[w] = x then send ⟨ys, w⟩ to x
                                    else send ⟨nys, w⟩ to x ;
              num_rec_u := 0 ; (* u must receive |Neigh_u| messages *)
              while num_rec_u < |Neigh_u| do
                     begin receive ⟨ys, w⟩ or ⟨nys, w⟩ message ;
                           num_rec_u := num_rec_u + 1
                     end ;
              if D_u[w] < ∞ then (* participate in pivot round *)
                begin if u ≠ w
                        then receive ⟨dtab, w, D⟩ fromthis Nb_u[w] ;
                      forall x ∈ Neigh_u do
                            if ⟨ys, w⟩ was received from x
                              then send ⟨dtab, w, D⟩ to x ;
                      forall v ∈ V do (* local w-pivot *)
                            if D_u[w] + D[v] < D_u[v] then
                              begin D_u[v] := D_u[w] + D[v] ;
                                    Nb_u[v] := Nb_u[w]
                              end
                end ;
              S_u := S_u ∪ {w}
        end
end
```

图4-6 Toueg算法（节点u）

假设权值（一条边的或一条路径的）和节点名可用W位表示，下面定理给出了算法的复杂度。

定理4.9　图4-6所示的算法计算每一对节点u、v从u到v的距离，如果距离是有限的，同时计算该长度路径上的第一个信道。算法中，每个信道交换消息为$O(N)$，总共为$O(N\cdot|E|)$条消息，每一信道有$O(N^2W)$位，共有$O(N^3W)$位，每个节点的存储空间为$O(NW)$。

证明。可从图4-5所示的算法导出图4-6所示的算法的正确性。

每一信道可携带两个消息<**ys**，w>（或者<**nys**，w>）（每个方向各有一个），在以w为枢轴节点的一轮循环中，至多有一个消息<**dtab**，w，D>，因此每个信道至多共有$3N$条消息。消息<**ys**，w>或者<**nys**，w>包含$O(W)$位，消息<**dtab**，w，D>包含$O(NW)$位，这给出了每个信道位数的界限。因此至多交换N^2条<**dtab**，w，D>消息、$2N\cdot|E|$条<**ys**，w>消息和<**nys**，w>消息，总共有$O(N^2\cdot NW + 2N\cdot|E|\cdot W) = O(N^3W)$位。在节点$u$保存的表$D_u$和$Nb_u$需$O(NW)$位存储空间。　□

在以w为枢轴节点的一轮循环中，只允许节点接收和处理那一轮循环中的消息，即，那些携带参数w的消息。如果信道满足fifo的性质，那么，<**ys**，w>和<**nys**，w>消息是那轮开始后首先到达的消息，每个信道一个消息，而<**dtab**，w，D>消息是下一个由$Nb_u[w]$到达的消息（如果节点在V_w中）。如果信道是fifo的，经过仔细程序设计，可以省略所有消息中的参数w。如果信道不是fifo的，也可能参数为w'的消息到达时，节点却期望w轮中的消息，这里w'是w之后的枢轴节点。在这种情况下，参数用于区别每一枢轴轮中的消息，因此必须使用局部缓冲区（或者在信道中或者在节点中），来推迟处理w'-消息。

116
~
117

Toueg给出了优化算法，参见下列结果。（节点u_2是u_1的后代，如果u_2属于u_1的子树。）

引理4.10　设$u_1 \neq w$，在以w为枢轴节点那一轮循环的开始时刻，u_2是T_w中u_1的后代。如果u_2在该轮中改变了它到v的距离，那么，在该轮中u_1也改变它到v的距离。

证明。由于u_2是u_1的后代（T_w中），

$$d^S(u_2,w) = d^S(u_2,u_1) + d^S(u_1,w) \tag{1}$$

又因为$u_1 \in S$，

$$d^S(u_2,v) \leqslant d^S(u_2,u_1) + d^S(u_1,v) \tag{2}$$

注意，在该轮计算中，u_2改变$D_{u_2}[v]$，当且仅当

$$d^S(u_2,w) + d^S(w,v) < d^S(u_2,v) \tag{3}$$

先利用式（2），然后利用式（1），再减去$d^S(u_2,u_1)$，可得，

$$d^S(u_1,w) + d^S(w,v) < d^S(u_1,v) \tag{4}$$

这表明，在本轮中u_1改变$D_{u_1}[v]$。　□

按照引理，图4-6所示的算法修改如下。在接到路由表D_w（消息<**dtab**，w，D>）之后，节点u首先执行局部以w为枢轴节点的操作，然后将路由表转发给它在T_w中的子节点。在转发表的过程中，只需发送以w为枢轴节点的计算中，使$D_u[v]$改变的那些$D[v]$，经过修改，在每一枢轴轮之间（如引理4.7所示），以及枢轴轮中，路由表是无回路的。

4.2.3　讨论以及更多算法

Toueg算法为从串行算法到分布式算法的转换提供了一种途径。因此，串行算法中变量可

以分布在进程中，拥有变量x的进程执行对变量x（在串行算法中）的赋值。当赋值表达式包含对于其他进程变量的引用时，进程之间就需通信，以便传输该变量的值，并同步化进程。可以充分利用串行算法的特定性质使得通信时间最小化。 118

Toueg算法简单、容易理解、具有较低复杂度并经过最优路径路由。它的主要缺点是健壮性不好。当网络拓扑结构发生变化时，整个计算需要重新进行。此外，从分布式算法工程角度来看，算法所具有的两个性质也使得它缺少吸引力。

第一，通过所有节点统一选择下一个枢轴节点（w）要求预先准确地知道参加节点的集合。由于一般预先不能得知，因此在执行Toueg算法之前，就需要执行另外一个分布式算法来计算这个集合（例如，Finn算法，图6-9所示的算法）。

第二，Toueg算法需要反复利用三角不等式$d(u, v) \leqslant d(u, w)+d(w, v)$。节点$u$对该不等式右端的计算需要关于$d(w, v)$的信息，这个信息一般是远程的，即，无论是$u$中还是它的近邻都不会有这个信息。对远程数据的依赖需要向远程节点传输信息，这一点可在Toueg算法中看到（广播部分）。

作为另一种选择，可在最短路径问题的算法中，利用以下所定义的$d(u, v)$方程：

$$d(u,v)=\begin{cases}0 & \text{if } u=v\\ \min\limits_{w\in Neigh_u} \omega_{uw}+d(w,v) & \text{否则}\end{cases} \tag{4-1}$$

该方程的两个性质使得基于它的算法不同于Toueg算法。

（1）数据局部性。为了计算式（4-1）的右边，节点u只需要局部信息（ω_{uw}）或近邻的信息（即，$d(w, v)$）。避免了远程节点之间的数据传输。

（2）目的节点独立性。计算从u到v的距离只需要到v的距离（即，u的近邻w的距离$d(w, v)$）。因此，到一个固定目的节点v_0的所有距离的计算，独立于到其他节点的距离的计算，因此距离的计算可以独立进行研究。

本节的其余部分，讨论了基于式（4-1）的两个算法。即，Merlin-Segall和Chandy-Misra算法。尽管这些算法具有数据局部性的优点，但是这些算法的通信复杂度并不比Toueg算法的通信复杂度有所改进。这是由于式（4-1）引入的目的节点独立性所导致。显然，利用其他目的节点的结果（正如在Toueg算法中所做的那样）要比引入数据局部性效果更好。 119

如果不能导致降低通信复杂度，那么数据局部性的重要性又在哪里？如果网络拓扑结构发生变化（信道和节点的故障或者修正），数据能够变化，则对远程数据的依赖要求后者不断地广播。达到这些广播的目标（根据广播中发生新的拓扑结构变化的可能性），产生了代价高昂的非-平凡问题的解决方法（参见文献[Gaf87]）。因此，基于式（4-1）的算法更易于处理拓扑结构易变的网络。在4.3节作为例子，深入讨论了这一算法。

1. Merlin-Segall算法

由Merlin-Segall [MS79]提出的计算每一目的节点路由表的算法，可以完全独立地运行；不同目的节点的计算互不影响。对于目的节点v，算法开始于根向v的树T_v，并不断地更新这棵树，最终使得该树成为目的节点为v的最优汇集树。

对于目的节点v，每一节点u保持到v的距离的估算值$D_u[v]$，以及包所转发的节点的近邻$Nb_u[v]$，这个近邻也是T_v中u的父节点。在更新一轮中，每个节点u将计算的距离$D_u[v]$发送到除$Nb_u[v]$的所有近邻中（在消息<**mydist**，v，$D_u[v]$>中）。如果节点u接到近邻w的消息<**mydist**，

v，d>，并且$d+\omega_{uw} < D_u[v]$，那么，u将$Nb_u[v]$改变为w，$D_u[v]$改变为$d+\omega_{uw}$。节点v控制更新一轮的计算，并要求每个信道交换两条W位的消息。

文献[MS79]表明，更新i轮后，所有跳数至多为i的最短路径已经正确计算。因此，至多N轮后，就能计算出到v的所有最短路径。可以为每个目的节点独立地运行算法，计算出到每一目的节点的最短路径。

定理4.11　Merlin-Segall计算最短路径路由表的算法复杂度为：每一信道交换消息的复杂度为$O(N^2)$，每一信道的位复杂度为$O(N^2 \cdot W)$，因此，算法的消息复杂度为$O(N^2 \cdot |E|)$，位复杂度为$O(N^2 \cdot |E| W)$。

算法也适合于网络拓扑结构和信道权值变化的情况。算法的一个重要性质是在更新一轮
120 的过程中，路由表是无回路的。

2. Chandy-Misra算法

由Chandy-Misra [CM82]所提出的计算同一目的节点的所有最短路径算法，利用了扩散计算（diffusing computation）。即，一种分布式计算，由一个节点初始化，其他节点一接到消息后就加入。

要计算所有节点到节点v_0的距离，每一节点u从$D_u[v_0] = \infty$开始并等候接收消息。注意节点v_0将消息<**mydist**，v_0，0>发送到所有近邻中。当节点u接到近邻w的消息<**mydist**，v_0，d>时，其中$d + \omega_{uw} < D_u[v_0]$时，$u$将$d+\omega_{uw}$赋给$D_u[v_0]$。并向所有近邻发送消息<**mydist**，$v_0$，$D_u[v_0]$>。参见图4-7所示的算法。

```
var D_u[v_0]  : weight   init ∞ ;
    Nb_u[v_0] : node     init udef ;

For node v_0 only:
    begin D_{v_0}[v_0] := 0 ;
          forall w ∈ Neigh_{v_0} do send ⟨mydist, v_0, 0⟩ to w
    end

Processing a ⟨mydist, v_0, d⟩ message from neighbor w by u:
    { ⟨mydist, v_0, d⟩ ∈ M_wu }
    begin receive ⟨mydist, v_0, d⟩ from w ;
          if d + ω_uw < D_u[v_0] then
            begin D_u[v_0] := d + ω_uw ; Nb_u[v_0] := w ;
                  forall x ∈ Neigh_u do send ⟨mydist, v_0, D_u[v_0]⟩ to x
            end
    end
```

图4-7　Chandy-Misra算法（节点u）

不难证明，$D_u[v_0]$总是$d(u, v_0)$的上界，即，由算法的不变式可得，$d(u, v_0) \leqslant D_u[v_0]$成立。参见习题4.3。为证明算法计算的距离正确，必须证明最终达到的一个配置，在这个配置中，对于每一个u，$D_u[v_0] \leqslant d(u, v_0)$也成立。在弱公平性假设的前提下，证明了这一性质。即，在每次计算中最终都会所收到发送的每条消息。如果不做这一假设，证明相当复杂。

定理4.12　在图4-7所示的算法的每次计算中，都达到一个配置，在该配置中，对于每一
121 节点u，满足$D_u[v_0] \leqslant d(u, v_0)$。

证明。固定v_0的一棵最优汇集树T，对除v_0之外的节点编号$v_1 \ldots v_{N-1}$，按照以下方式，如果v_i是v_j的父节点，那么$i < j$。设C是一次计算，可以对j用归纳法证明：对于每个$j \leqslant N-1$，达到一个配置，在配置中，对于每个$i \leqslant j$，$D_{v_i}[v_0] \leqslant d(v_i, v_0)$。由算法可见，$D_{v_i}[v_0]$从不增加。因此，如果$D_{v_i}[v_0] \leqslant d(v_i, v_0)$在某些配置中成立，则它在后面的所有配置中也成立。

情形 $j=0$：v_0执行完初始化后，$d(v_0, v_0)=0$；$D_{v_0}[v_0]=0$；因此，算法执行后，$D_{v_0}[v_0] \leqslant d(v_0, v_0)$成立。

情形 $j+1$：假设达到某一配置，其中对于每个$i \leqslant j$，$D_{v_i}[v_0] \leqslant d(v_i, v_0)$，并考虑节点$v_{j+1}$。从$v_{j+1}$到$v_0$，存在一条长为$d(v_{j+1}, v_0)$的最短路径$v_{j+1}, v_i, \ldots, v_0$，其中$v_i$是$T$中$v_{j+1}$的父节点，因此，$i \leqslant j$。由归纳假设，达到配置时$D_{v_i}[v_0] \leqslant d(v_i, v_0)$。无论何时$D_{v_i}[v_0]$下降，$v_i$都将消息**<mydist**，$v_0$，$D_{v_i}[v_0]$>发送至近邻，因此，$d \leqslant d(v_i, v_0)$的消息**<mydist**，$v_0$，$d$>被发送到$v_{j+1}$至少一次。

根据假设，在C中，由v_{j+1}接收消息。算法蕴含接到消息后，$D_{v_{j+1}}[v_0] \leqslant d + \omega_{v_{j+1}} v_i$成立。由$i$的选取，可得$d + \omega_{v_{j+1}} v_i \leqslant d(v_{j+1}, v_0)$。 □

与4.3.3节开始部分有关变更算法的评述相比，完整算法还包括一个机制，通过该机制，节点能够检测出计算是否已经完成。检测计算是否完成的机制在8.2.1节讨论，这是一种Dijkstra-Scholten算法的变形。

该算法与Merlin-Segall算法存在两点不同之处。第一，不存在节点u的父节点，该节点被期望发送类型**<mydist**，.，.>的消息。即使是在计算中和网络拓扑结构改变时，Merlin-Segall算法的这种性质也总是可以保证路由表是无回路的。第二，**<mydist**，.，.>消息交换不必按照轮数协调，完全任意发生。这对算法的复杂度是不利的。对于某一目的节点v_0，算法需要指数级的消息数计算到某个目的节点v_0的路径。如果假设所有信道代价相同（即，考虑最小跳数路由算法），到v_0的所有最短路径利用$O(N \cdot |E|)$条消息，每条消息$O(W)$位进行计算，则得以下结果。 122

定理4.13 *Chandy-Misra算法计算最小跳数路由表的算法复杂度为：每一信道交换消息复杂度为$O(N^2)$，每一信道位复杂度为$O(N^2 \cdot W)$，算法消息复杂度为$O(N^2 \cdot |E|)$，位复杂度为$O(N^2 \cdot |E| \cdot W)$。*

较之Merlin-Segall算法，Chandy-Misra算法的优点在于它的简明性，具有更小的空间复杂度和更低的时间复杂度。

4.3 变更算法

按照最小跳数度量，Tajibnapis提出的计算路由表的变更算法[Taj77]是最优的。算法可与Chandy-Misra算法做比较，它保存一些附加信息，使得在信道发生故障或者修复时，只做部分重新计算就能完成对路由表的更新。算法沿用文献Lamport[Lam82]中的表示。算法依赖以下假设。

N1 节点知道网络规模(N)。

N2 信道满足fifo假设。

N3 节点被通知关于其近邻节点的故障和修复情况。

N4 路径的代价等于路径上信道的数目。

算法能够处理信道故障、修复信道，还可以增加信道，但假设近邻信道发生故障或者正

在修复时，节点可以得知这种情况。因此不考虑节点故障或者恢复的情况，而是假设当连接节点的信道发生故障时，近邻可以得知节点的故障。算法在每个节点u中维持一张表$Nb_u[v]$，对于每个目的节点v，该表给出了u的近邻，v的包将向其转发。不能要求在任意情况下，这些表的计算，都能在有限步内终止。因为反复的信道故障和修复可能需要无限的重计算。算法要求如下。

R1　如果在有限次的网络拓扑结构变化之后，网络拓扑结构保持为常数，那么算法在有限步后终止。

R2　当算法终止时，表$Nb_u[v]$满足：

a）如果$v = u$，那么，$Nb_u[v] = local$;

b）如果从u到v（$\neq u$）的路径存在，那么，$Nb_u[v] = w$，其中w是从u到v的最短路径上，u的第一个近邻。

123　c）如果从u到v不存在路径，那么，$Nb_u[v] = udef$。

```
var Neigh_u      : set of nodes ;     (* The neighbors of u *)
    D_u          : array of 0.. N ;   (* D_u[v] estimates d(u, v) *)
    Nb_u         : array of nodes ;   (* Nb_u[v] is preferred neighbor for v *)
    ndis_u       : array of 0.. N ;   (* ndis_u[w, v] estimates d(w, v) *)

Initialization:
  begin forall w ∈ Neigh_u, v ∈ V do ndis_u[w, v] := N ;
        forall v ∈ V do
              begin D_u[v] := N ; Nb_u[v] := udef end ;
        D_u[u] := 0 ; Nb_u[u] := local ;
        forall w ∈ Neigh_u do send ⟨mydist, u, 0⟩ to w
  end

Procedure Recompute (v):
  begin if v = u
          then begin D_u[v] := 0 ; Nb_u[v] := local end
          else begin (* Estimate distance to v *)
                     d := 1 + min{ndis_u[w, v] : w ∈ Neigh_u} ;
                     if d < N then
                       begin D_u[v] := d ;
                             Nb_u[v] := w with 1 + ndis_u[w, v] = d
                       end
                     else begin D_u[v] := N ; Nb_u[v] := udef end
                end ;
        if D_u[v] has changed then
          forall x ∈ Neigh_u do send ⟨mydist, v, D_u[v]⟩ to x
  end
```

图4-8　变更算法（第一部分，节点u）

4.3.1　算法描述

图4-8和图4-9所示的算法给出了Tajibnapis的变更算法。通过非形式地描述算法中的一些操作，及随后严格证明算法的正确性。考虑到表述的直观，对拓扑结构变化的模拟可以简化如同[Lam82]，假设可以同时处理受影响的两节点中的变化的通知。4.3.3节讨论了如何处理这

些异步通知。

```
Processing a ⟨mydist, v, d⟩ message from neighbor w:
    { A ⟨mydist, v, d⟩ is at the head of Q_wv }
    begin receive ⟨mydist, v, d⟩ from w ;
          ndis_u[w, v] := d ; Recompute (v)
    end

Upon failure of channel uw:
    begin receive ⟨fail, w⟩ ; Neigh_u := Neigh_u \ {w} ;
          forall v ∈ V do Recompute (v)
    end

Upon repair of channel uw:
    begin receive ⟨repair, w⟩ ; Neigh_u := Neigh_u ∪ {w} ;
          forall v ∈ V do
                begin ndis_u[w, v] := N ;
                      send ⟨mydist, v, D_u[v]⟩ to w
                end
    end
```

图4-9 变更算法（第二部分，节点u）

基于对每个节点到目的节点v的距离的估算，选择一个近邻节点，目的节点为v的包将向其转发。总是选择具有最小估算距离的近邻节点。对于u的每个近邻w，节点u维持对$d(u, v)$的估算$D_u[v]$，并且，对于u的每个近邻w，估算$d(w, v)$的$ndis_u[w, v]$。由$ndis_u[w, v]$估算值计算出估算值$D_u[v]$，经过与近邻的通信可得$ndis_u[w, v]$估算值。

估算$D_u[v]$的计算过程如下。如果$v = u$，那么，$d(u, v) = 0$，因此，$D_u[v] = 0$。如果$u \neq v$，那么，从u到v的最短路径（如果路径存在）由从u到近邻与从近邻到v的最短路径连接而成，因此，

$$d(u,v) = 1 + \min_{w \in Neigh_u} d(w,v) \text{。}$$

由此方程，当节点$u \neq v$时，通过将本方程用于已经计算出的$d(w, v)$（表中为$ndis_u[w, v]$）来计算$d(u, v)$。因有N个节点，最小跳数路径长度为至多为$N-1$。如果计算的距离为N或者更大，则可怀疑该路径不存在。表中的N用于表示这种情况。

算法要求节点计算它的近邻到v的距离。因为它们在消息<**mydist** ,. , . > 中通信，所以这些信息可以从这些节点得到。如果节点u计算出它到v的距离$d(D_u[v] = d)$。就将这一信息放入消息<**mydist**，v，d >中，并发送到u的所有近邻中。节点u一旦接到近邻w的消息<**mydist**，v，d >，就将值d赋给$ndis_u[w, v]$。由于$ndis_u[w, v]$的值改变时，u的计算$d(u, v)$也要改变，于是，每当$ndis_u$表发生变化时，就要重新计算$d(u, v)$的值。如果该值确实改变，比如为d'，就要利用消息<**mydist**，v，d' >与近邻通信。 124 ~ 125

对于信道故障和修复，算法通过修改局部路由表对此做出反应，如果距离估算发生改变，就发送消息<**mydist**，.，.>。假设节点接到关于信道好坏（假设N3）的通知，通知形式为<**fail**，.>和<**repair**，.>的消息。用两队列模拟节点u_1和u_2之间的信道。$Q_{u_1u_2}$表示从u_1到u_2的消息，$Q_{u_2u_1}$表示从u_2到u_1的消息。当信道发生故障时，就从配置中删除这些队列（这引起在两队列中的所有消息丢失），且在信道两端的节点接收消息<**fail**，.>消息。如果u_1和u_2之间的信道

发生故障，则节点u_1接收消息<**fail**，u_2>，则节点u_2接收消息<**fail**，u_1>。当信道被修复时（或者向网络增加新信道），在配置中增加两个空队列，信道连接的两节点间接收消息<**repair**，.>。如果u_1和u_2之间出现信道，则u_1接收消息<**repair**，u_2>，而节点u_2接收消息<**repair**，u_1>。

算法对于故障和修复的响应如下。当u和w之间的信道发生故障时，从$Neigh_u$中删除w。到每个目的节点的距离需重新估算。如果距离改变，就要将结果发送到其他所有近邻中。如果以前的最佳路由经过故障信道，且没有其他近邻w'，满足$ndis_u[w', v] = ndis_u[w, v]$。当信道被修复时（或者增加新的信道），将$w$加入$Neigh_u$中，但是$u$还没有计算$d(w, v)$的距离（反之亦然）。新的近邻$w$立即被告知所有到目的节点$v$的距离$D_u[v]$（通过发送消息<**mydist**，$v$，$D_u$[v]>）。直到$u$接到来自$w$的类似消息。$u$利用$N$作为$d(w, v)$的估算值，即，设$ndis_u[w, v]$为$N$。

变更算法的不变式　我们要证明许多断言都是不变式。图4-10中给出了这些断言。断言$P(u, w, v)$阐明，如果u已经处理完来自w的消息<**mydist**，v，.>，那么，u对$d(w, v)$的估算等于w对$d(w, v)$的估算。设谓词$up(u, w)$为真，当且仅当u和w之间的（双向）信道存在，且正在工作。断言$L(u, v)$表明，u的估算$d(u, v)$总是与u的局部知识一致。因而，可以相应地设置$Nb_u[v]$。

$$
\begin{aligned}
&P(u, w, v) \equiv \\
&\qquad up(u, w) \iff w \in Neigh_u && (1)\\
&\quad \wedge\ up(u, w) \wedge Q_{wu} \text{ contains a } \langle \mathbf{mydist}, v, d\rangle \text{ message} \\
&\qquad \Rightarrow \text{the last such message satisfies } d = D_w[v] && (2)\\
&\quad \wedge\ up(u, w) \wedge Q_{wu} \text{ contains no } \langle \mathbf{mydist}, v, d\rangle \text{ message} \\
&\qquad \Rightarrow ndis_u[w, v] = D_w[v] && (3)\\
&L(u, v) \equiv \\
&\qquad u = v \Rightarrow (D_u[v] = 0 \wedge Nb_u[v] = local) && (4)\\
&\quad \wedge\ (u \neq v \wedge \exists w \in Neigh_u : ndis_u[w, v] < N - 1) \\
&\qquad \Rightarrow (D_u[v] = 1 + \min_{w \in Neigh_u} ndis_u[w, v] = 1 + ndis_u[Nb_u[v], v]) && (5)\\
&\quad \wedge\ (u \neq v \wedge \forall w \in Neigh_u : ndis_u[w, v] \geqslant N - 1) \\
&\qquad \Rightarrow (D_u[v] = N \wedge Nb_u[v] = udef) && (6)
\end{aligned}
$$

图4-10　不变式$P(u, w, v)$和$L(u, v)$

当信道中没有更多算法的消息在传输时，算法计算终止。对于整个系统，这些配置不是终止的，这是因为随着信道发生故障或者修复（对此系统必须做出响应），系统计算可能稍后继续。我们称几乎没有消息的配置是稳定的（stable），并定义谓词**stable**如下。

stable ≡ $\forall u, w$: $up(u, w) \Rightarrow Q_{wu}$没有包含消息<**mydist**，.，.>。

假设初始时，变量$Neigh_u$正确地反映了正在工作的通信信道的存在性，即，式（1）初始时成立。为证明断言的不变式，考虑三种类型的转移：

（1）收到消息<**mydist**，.，.>。假设整个代码段的具有执行原子性，并认为是一次转移。注意在这次转移中，只接到一个消息，却可能发送许多消息。

（2）当信道发生故障时，信道两端的节点处理消息<**fail**，.>。

（3）当信道发生修复时，两个连接的节点处理消息<**repair**，.>。

引理4.14　*对于所有u_0、w_0和v_0，$P(u_0, w_0, v_0)$是一个不变式。*

证明。初始时，在每一节点初始化过程执行之后，由假设，式（1）成立。初始时如果

$\neg up$（u_0，w_0）成立，则式（2）和式（3）都成立。如果初始时$up(u_0, w_0)$成立，则$ndis_{u_0}[w_0, v_0] = N$。如果$w_0 = v_0$，则$D_{w_0}[w_0] = 0$，但是消息<**mydist**，v_0，0>在队列$Q_{w_0u_0}$中，因此式（2）和式（3）为真。如果$w_0 \neq v_0$，则$D_{w_0}[v_0] = N$，但是队列中没有消息，这也蕴含着式（2）和式（3）成立。我们依次考虑上述的三种类型的状态转移。 126 ~ 127

类型（1） 假设u接收w的消息<**mydist**，v，d>。这种情况，不会引起拓扑结构变化，也不会引起$Neigh$集合的变化。因此式（1）依然成立。如果$v \neq v_0$，消息的接收不会改变$P(u_0, w_0, v_0)$。

如果$v = v_0$、$u = u_0$和$w = w_0$，$ndis_{u_0}[w_0, v_0]$的值可能发生变化。然而，如果信道中仍有另一消息<**mydist**，v_0，>，则消息的值仍然满足式（2），因此式（2）保持。式（3）保持是因为它的前提为假。如果所收到的消息是信道中这种类型的最后一条消息，那么由式（2）可得$d = D_{w_0}[v_0]$，这蕴含着式（3）的结论变为真，式（3）保持。此时式（2）的前提变为假，式（2）保持。

如果$v = v_0$、$u = w_0$（u_0是u的近邻），如果在w_0中执行$Recompute$（v）后，改变了$D_{w_0}[v_0]$的值，那么，式（2）和式（3）可能为假。这种情况下，具有新值的消息<**mydist**，v_0，.> 向u_0发送。这蕴含着式（3）的前提为假，式（2）的结论变为真，因此式（2）和式（3）保持。这是把消息<**mydist**，v_0，.>加入队列$Q_{w_0u_0}$中的惟一一种情况。总是满足$d = D_{w_0}[v_0]$。

如果$v = v_0$和$u \neq u_0$，w_0在$P(u_0, w_0, v_0)$中不会改变。

类型（2） 假设信道uw发生故障。

如果$u = u_0$和$w = w_0$，这个故障使得式(2)和式(3)的前提为假，因此式（2）和式（3）保持。式（1）保持是因为w_0被从$Neigh_{u_0}$中删除，反之亦然。如果$u = w_0$和$w = u_0$，同样情况发生。

如果$u = w_0$，但是$w \neq u_0$，由于$D_{w_0}[v_0]$的值发生改变，所以式（2）和式（3）的结论可能为假。这种情况下，由w_0所发送的消息<**mydist**，v_0，.>再次使得式（3）的前提为假。这使得式（2）的结论为真，因此式（2）和式（3）保持。

在所有其他情况下，$P(u_0, w_0, v_0)$未有任何变化。

类型（3） 假设增加信道uw。

如果$u = u_0$和$w = w_0$，这使得$up(u_0, w_0)$为真，但是将w_0加入$Neigh_{u_0}$（反之亦然）保持式（1）。

由w_0所发送的消息<**mydist**，v_0，$D_{w_0}[v_0]$>使得式（2）为真，式（3）的前提为假，因此$P(u_0, w_0, v_0)$被保持。

在所有其他情况下，$P(u_0, w_0, v_0)$没有发生变化。 □

引理4.15 *对于每一u_0和v_0，$L(u_0, v_0)$是一个不变式。* 128

证明。 初始时$D_{u_0}[u_0] = 0$和$Nb_{u_0}[u_0] = local$。对于$v_0 \neq u_0$，初始时，对于所有$w \in Neigh_u$，$ndis_{u_0}[w, v_0] = N$，并且$D_{u_0}[v_0] = N$和$Nb_{u_0}[v_0] = udef$。

类型（1） 假设u接收w的消息<**mydist**，v，d>。如果$u \neq u_0$，或者$v \neq v_0$，$L(u_0, v_0)$中没有变量改变。如果$u = u_0$、$v = v_0$，且$ndis_{u_0}[w, v_0]$的值改变，需重新准确计算$D_{u_0}[v_0]$和$Nb_{u_0}[v_0]$，满足$L(u_0, v_0)$。

类型（2） 假设信道uw发生故障。

如果$u = u_0$或$w = u_0$，那么，$Neigh_{u_0}$改变。需再次重新准确计算$D_{u_0}[v_0]$和$Nb_{u_0}[v_0]$，以满足$L(u_0, v_0)$。

类型（3） 假设增加信道uw。

如果$u = u_0$，那么，由于增加w使得$Neigh_{u_0}$改变，但由于u设置$ndis_{u_0}[w, v_0]$值为N，这使$L(u_0, v_0)$保持。 □

4.3.2 变更算法的正确性

现在将要证明算法的两个正确性要求。

定理4.16 *当达到稳定配置时，路由表$Nb_u[v]$满足：*

(1) 如果$u = v$，那么$Nb_u[v] = local$;

(2) 如果存在从u到$v \neq u$的路径，那么，$Nb_u[v] = w$，其中w是从u到v的最短路径上u的第一个近邻;

(3) 如果从u到v的路径不存在，那么，$Nb_u[v] = udef$。

证明。当算法终止时，对于所有u、v和w，谓词$P(u, w, v)$及谓词**stable**都成立。这蕴含着，对于所有u、v和w，

$$up(u,w) \Rightarrow ndis_u[w,v] = D_w[v] \ 。 \tag{4-2}$$

对于所有u和v，再次应用$L(u, v)$可得

$$D_u[v] = \begin{cases} 0 & \text{if} \quad u = v \\ 1 + \min\limits_{w \in Neigh_u} D_w[v] & \text{if} \quad u \neq v \wedge \exists w \in Neigh_u : D_w[v] < N-1 \\ N & \text{if} \quad u \neq v \wedge \forall w \in Neigh_u : D_w[v] \geqslant N-1 \end{cases} \tag{4-3}$$

如果u和v在网络的同一连通分量中，则证明$D_u[v] = d(u, v)$就足够了。如果u和v在网络的不同连通分量中，则需证明$D_u[v] = N$。

129 首先对$d(u, v)$用归纳法证明，如果u和v在同一连通分量中，则$D_u[v] \leqslant d(u, v)$。

情形$d(u, v) = 0$：这隐含着$u = v$，因此，$D_u[v] = 0$。

情形$d(u, v) = k+1$：这蕴含着存在节点$w \in Neigh_u$且$d(w, v) = k$，由归纳假设$D_w[v] \leqslant k$，由式(4-3)，这蕴含着$D_u[v] \leqslant k+1$。

现在对$D_u[v]$用归纳法证明，如果$D_u[v] < N$，那么，在u和v之间存在路径，且$d(u, v) \leqslant D_u[v]$。

情形$D_u[v] = 0$：式(4-3)蕴含仅当$u = v$时，$D_u[v] = 0$。这表明u和v之间的路径为空，且$d(u, v) = 0$。

情形$D_u[v] = k+1 < N$：式(4-3)蕴含，存在节点$w \in Neigh_u$且$D_w[v] = k$。由归纳假设，w和v之间存在路径，且$d(w, v) \leqslant k$，这蕴含着u和v之间存在路径，且$d(u, v) \leqslant k+1$。

由此可得，如果u和v在同一连通分量中，那么$D_u[v] = d(u, v)$，否则，$D_u[v] = N$。式(4-2)表明，$\forall u, v$：$L(u, v)$蕴含所阐述的关于$Nb_u[v]$的结果。 □

为证明如果拓扑结构不再变化，算法最终达到稳定情形，定义关于**stable**的范函数。对于算法的配置γ，定义

t_i =（消息<**mydist**, ..., i>的数目）+（满足$D_u[v] = i$有的序对u, v的个数。）

和函数f

$$f(\gamma) = (t_0, t_1, \ldots, t_N)$$

$f(\gamma)$是自然数的$(N+1)$元组，并假设字典有序（用$<_l$表示）。回忆$(\mathbb{N}^{N+1}, <_l)$是一

良基集（习题2.5）。

引理4.17 消息<**mydist**，.，.>的处理使得f递减。

证明。假设$D_u[v] = d_1$的节点u收到消息<**mydist**，v，d_2>，重新计算之后，$D_u[v]$的新值为d。算法蕴含$d \leqslant d_2+1$。

如果$d < d_1$：$d = d_2+1$，这蕴含着，t_{d_2}按照1递减（t_{d_1}也按照1递减），并且只有$d > d_2$时t_d增加。这表明f的值递减。

如果$d = d_1$：u不发送任何新的消息<**mydist**，.，.>，惟一对f产生影响的是t_{d_2}按照1递减，因此，f的值递减。

130

如果$d > d_1$：t_{d_1}按照1递减（t_{d_2}也按照1递减），只有当$d > d_1$时，t_d增加，这蕴含着f的值递减。 □

定理4.18 在经过有限次变化之后，如果网络的拓扑结构不再变化，那么算法在有限步后达到稳定配置。

证明。如果只有消息<**mydist**，.，.>的处理，而网络拓扑结构保持，由前述引理可得，每次转移，f的值递减。由f域的良基性，只能发生有限次的转移；因此算法在有限步后达到稳定配置。 □

4.3.3 算法讨论

算法的正确性保证，经过最终拓扑变化之后，在有限时间内算法收敛到正确的路由表。这并不是算法实际的行为。谓词**stable**可能在实际中的大部分时间为假（即，如果拓扑变化频繁），当**stable**为假时，我们对路由表一无所知。它们可能包含回路，或者给出关于目的节点可达性的错误信息。因此算法只适合于拓扑结构不常变化的应用，与拓扑结构突变之间的平均时间相比，算法的收敛时间较小。这是由于**stable**是全局性质，节点难以区别算法的稳定配置和非稳定配置。这意味着，节点不知道它的路由表是否正确地反映了网络的拓扑结构，并且，在达到稳定配置之前，不能推迟对于数据包的转发。

1. 异步处理通知

本节已经假设，拓扑结构变化的通知可随着一次转移中的变化自动处理。处理在被删除（或被增加）的信道的两边同时进行。Lamport[Lam82]更详细地进行了分析，并允许在处理这些通知时有一个延时。从w到u的通信信道用三个队列的连接进行模拟。

131

（1）OQ_{wu}，w的输出队列。

（2）TQ_{wu}，当前正被传输的消息队列(数据包)。

（3）IQ_{wu}，u的输入队列。

在信道的正常操作下，w通过向队列OQ_{wu}追加消息向u发送消息，然后消息从队列OQ_{wu}移到队列TQ_{wu}，再从队列TQ_{wu}移到队列IQ_{wu}，u通过从队列IQ_{wu}中删除消息来接收消息。当信道发生故障时，则丢弃队列TQ_{wu}中的消息，队列OQ_{wu}中的消息也随之丢弃，而不是将其追加到队列TQ_{wu}中。消息<**fail**，w>被放在队列IQ_{wu}尾，当恢复正常操作时，消息<**repair**，w>被放在队列IQ_{wu}尾部。谓词$P(u, w, v)$具有稍微复杂的形式，但算法不变。

2. 最短路径路由

可能为每一信道赋以权值，并修改算法来计算最短路径，而不是最小跳数路径。在变更算法的Recompute过程中，如果用w_{uw}代替常数1，在估算经过w的最短路径长度时，就要考虑

信道uw上的权值。用网络直径的上限代替算法中的常数N。

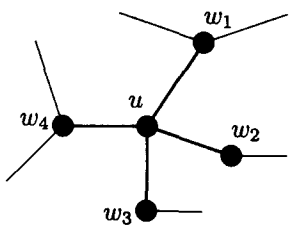
容易证明，当修改的算法达到稳定配置时，路由表是正确的，并给出最优路径（网络中的所有回路必须具有正权值）。算法最终达到这样配置的证明需要更复杂的范函数。

可以对算法进行扩充，使其可以处理变化的信道权值；节点u对信道权值变化的反应是重新计算所有v的$D_u[v]$。与收敛时间相比，信道代价变化之间的平均时间很大时，算法才有实际意义，然而，这是不切实际的假设。在这些情况下，希望算法在收敛过程中无回路，如Merlin-Segall算法。

4.4 带有压缩路由表的路由

到目前为止，所讨论的路由算法都要求各节点维持一张带有各自可能目的节点分开表项的路由表。当需要通过网络转发包时，就要访问路径上各节点的这些路由表（除目的节点外）。本节研究路由表的组织结构，以降低路由表的存储空间和查找开销。这里不考虑分布式算法如何计算这些路由表。为简化表示，在本节中，假设网络都是连通的。

本节所讨论的在三种路由机制中获得更小路由表的策略，可以解释如下。如果节点的路由表分开存储每一目的节点的输出信道，路由表表长必须为N，无论如何对输出信道进行压缩编码，路由表要求$\Omega(N)$位的空间。现在考虑路由表的重组，其中路由表包含节点信道的表项，告知哪些目的节点要经过此信道路由。参见图4-11。对于有deg个信道的节点，表长为deg，在存储空间上的实际节省，取决于如何能压缩地表示每个信道目的节点集合。为保证查找的效率，对于给定的目的节点，所构造的路由表要能快速检索出其输出信道。



dest.	chan.
v_1	w_2
u	-
v_j	w_3
v_N	w_1

chan.	destinations
w_1	..., v_N
w_2	v_1, ...
w_3	..., v_j, ...
w_4	

图4-11　减小路由表的大小

4.4.1 树标号模式

Santoro和Khatib提出了第一个压缩路由方法[SK85]。该方法基于对节点用0到$N-1$的整数进行标号，按照这种方法，每一信道的目的节点集合为一个区间。令$\mathbb{Z}_N$表示集合$\{0, 1, \ldots, N-1\}$。在本节中，集合中的所有计算以N为模。例如，$(N-1)+1 \equiv 0$，序号和在$\mathbb{Z}$中的一样。

定义4.19　循环区间是 $[a,b)$ 是$\mathbb{Z}_N$中如下定义的整数集合

$$[a,b) = \begin{cases} \{a,a+1,\cdots,b-1\} & \text{如果} a < b \\ \{0,\cdots,b-1,a,\cdots,N-1\} & \text{如果} a > b \end{cases}$$

易见，$[a, a)= \mathbb{Z}_N$，对于$a \neq b$，$[a, b)$ 的补集是 $[b, a)$。如果$a < b$，称循环区间 $[a, b)$ 为线性的。

定理4.20　树T的节点可以按照这样一种方式编号，使得对于每一节点的每条输出信道，

必须经过那个信道路由的目的节点集合是循环区间。

证明。任取节点v_0作为树根，对于每一节点w，设$T[w]$表示以w为根的T的子树。可能按照这样一种方式为节点编号，使得对于每个节点w，$T[w]$中节点的编号形成线性区间，如图4-12中按照树的前序遍历编号。按照这个次序，w是树$T[w]$中第一个被访问的节点；然后，在访问完树$T[w]$中的所有节点后，才能访问树$T[w]$之外节点。因此，$T[w]$中的节点可以按照线性区间 $[l_w, l_w+|T[w]|)$（l_w是w的标号）编号。

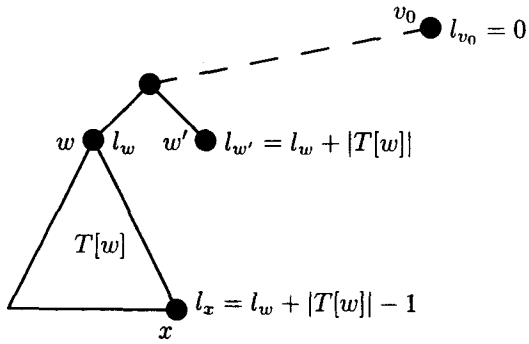


图4-12 前序遍历树

设 $[a_w, b_w)$ 表示赋给树$T[w]$中结点编号的区间。w的近邻或者是它的子节点或者是它的父节点。节点w向它的子节点u转发目的节点在$T[u]$中的包，即，编号在 $[a_u, b_u)$ 中的节点。节点w向它的父节点转发目的节点不在$T[w]$中的包，即，节点编号在$\mathbb{Z}_N \setminus [a_w, b_w) = [b_w, a_w)$中的节点。 □

给定起始点和结束点，只利用$2\log N$位就可以表示一个循环区间。由于在应用中，必须存储并集为$\mathbb{Z}_N$的不相交区间的集合（把u加入到节点u的区间之一），每个区间有$\log N$位就足够了。只要存储与每个信道对应的区间的起始点，同一节点对应区间的结束点等于下一区间的起始点。与节点u的信道uw对应的区间起始点如下

$$\alpha_{uw} = \begin{cases} l_w & \text{如果}w\text{是}u\text{的子节点} \\ l_u + |T[u]| & \text{如果}w\text{是}u\text{的父节点} \end{cases}$$

假设用α_1，....，α_{deg_u}对度为deg_u的节点u的信道进行编号，其中$\alpha_1 < .... < \alpha_{deg_u}$。图4-13所示的算法给出了转发过程。信道标号把集合$\mathbb{Z}_N$划分成deg_u段，每个段与一个信道对应。如图4-14所示。观察可见，至多有一个区间是非线性的。如果标号在节点处排序，利用折半查找用$O(\log deg_u)$步就能找到正确标号。下标i的计算为模deg_u，即，$\alpha_{deg_u}+1 = \alpha_1$。

```
(* A packet with address d was received or generated at node u *)
if d = l_u
  then deliver the packet locally
  else begin select α_i s.t. d ∈ [α_i, α_{i+1}) ;
             send packet via the channel labeled with α_i
       end
```

图4-13 区间转发算法（节点u）

树标号模式的路由是最优的，因为在树中每两个节点之间只存在一条简单路径。如果网

络不是一棵树，也可用这种方法。选择网络的一棵固定生成树T，将树标号模式应用于这棵生成树上。不属于这棵树的信道从不使用，在路由表中以特殊记号表示这些不用信道，以表明没有包经过此信道路由。

为比较用这种方法选出的路径长度与最优路径，设$d_T(u, v)$表示T中从u到v的距离，$d_G(u, v)$表示G中从u到v的距离。令D_G表示G的直径，定义为u和v上$d_G(u, v)$的最大值。

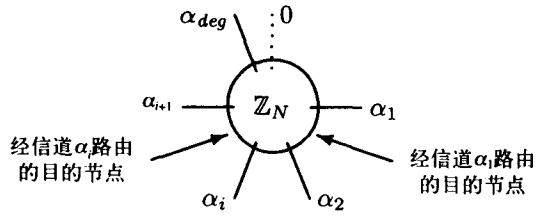


图4-14　一个节点的$\mathbb{Z}_N$划分

引理4.21　$d_T(u, v)$和$d_G(u, v)$之间的比率不存在一致界限。在以跳数作为路径度量的特殊情况下已经成立。

证明。选择G为N个节点的环，从G中去掉一条信道，如xy，则得G的一棵生成树。现在，$d_G(x, y) = 1$和$d_T(x, y) = N-1$，因此，比率为$N-1$。通过选取大环，这个比率可以是任意的。　□

以下引理依赖信道代价的对称性，即$\omega_{uv} = \omega_{vu}$。这蕴含着对于所有$u$和$v$，$d_G(u, v) = d_G(v, u)$。

引理4.22　可以按照这样一种方式选择T，使得对于所有u和v，满足$d_T(u, v) \leqslant 2D_G$。

证明。选择T是节点w_0的最优汇集树（如在定理4.2中所作的那样）。那么，

$$
\begin{aligned}
d_T(u, v) &\leqslant d_T(u, w_0) + d_T(w_0, v) \\
&= d_T(u, w_0) + d_T(v, w_0) && \text{由}\omega\text{的对称性} \\
&= d_G(u, w_0) + d_G(v, w_0) && \text{由}T\text{的选择} \\
&\leqslant D_G + D_G && \text{由}D_G\text{的定义}
\end{aligned}
$$

□

总之，与两个相同节点之间的最优路径相比较（引理4.21），用这种方法选择的路径会任意坏，它最多比系统中其他两节点间的路径差两倍。这表明，这种方法适合于通信节点之间的距离为$\Theta(D_G)$的情况。如果节点间的大多数通信为G中的较短距离，就不需使用这种方法。

132 ~ 136

除了所选路径长度的因素之外，树路由模式还有以下缺点：

（1）不能使用不属于T的信道，这是网络资源的一种浪费。

（2）信息量集中在树上，容易引起拥塞。

（3）T中的每一条信道故障，都会将网络分开。

4.4.2 区间路由

Van Leeuwen和Tan[LT87]扩充了树标号模式到非树的网络，用这种方法，（几乎）每一条信道都可用于包通信。

定义4.23　网络的区间标号模式（ILS）是

（1）将$\mathbb{Z}_N$中的不同标号赋给网络中的节点，且

（2）对于每一个节点，将$\mathbb{Z}_N$中的不同标号赋给那个节点的各个信道。

在区间路由中，假设给定ILS，包的转发如图4-13所示的算法。

定义4.24　如果按区间标号模式转发的所有包最终到达其目的节点，则区间标号模式是

有效的。

可以证明，对于每一连通网络G，存在有效的区间标号模式（定理4.25）。然而，对于任意连通网络，这种模式通常不是非常有效的。在证明存在性之后，研究通过区间路由模式选择路径的最优性。

定理4.25 *对于每一连通网络G，存在有效的区间标号模式。*

*证明。*通过扩展Santoro和Khatib提出的树标号模式，构造有效的区间标号模式，并应用于网络中的一棵生成树T。给定一棵生成树非树边（frond edge）表示不属于这棵生成树的边。v是u的祖先，当且仅当$u \in T[v]$。构造的主要问题是如何给非树边赋标号（树边的标号如同树标号模式）。用这样一种方法选择生成树，以致于所有非树边取一种限定的形式，正如将要表明的那样。 137

引理4.26 *存在生成树，满足所有非树边介于节点与那个节点的祖先之间。*

*证明。*用深度优先搜索对网络进行遍历，得到的生成树具有引理所述的性质。参见文献[Tar72]和6.4节。 □

设T是一棵G的深度优先搜索生成树。

定义4.27 *G的深度优先搜索ILS（与T有关）是满足以下规则的标号模式。*

（1）节点按照对T的前序遍历次序标号，即，子树$T[w]$中的节点用$[l_w, l_w+|T[w]|)$中的数标号。记$k_w = l_w + |T[w]|$。

（2）节点u的边uw标号为α_{uw}。

a）如果uw是非树边，那么$\alpha_{uw} = l_w$。

b）如果w是u的子节点（T中），那么$\alpha_{uw} = l_w$。

c）如果w是u的父节点，那么$\alpha_{uw} = k_u$。除非$k_u = N$且u有到根的非树边。

（在后一种情况下，由规则（a）对非树边标以0，如果标以k_u，则会违反条件，即u的所有边的标号不同。标号以N为模，因此$N \equiv 0$。）

（d）如果w是u的父节点，u有到根的非树边，且$k_u = N$，那么$\alpha_{uw} = l_w$。

图4-15给出了深度优先搜索ILS的例子。其中所有非树边按照规则（2a）标号，节点4、8和10的父节点的边按照规则（2c）标号，节点9的父节点的边按照规则（2d）标号。

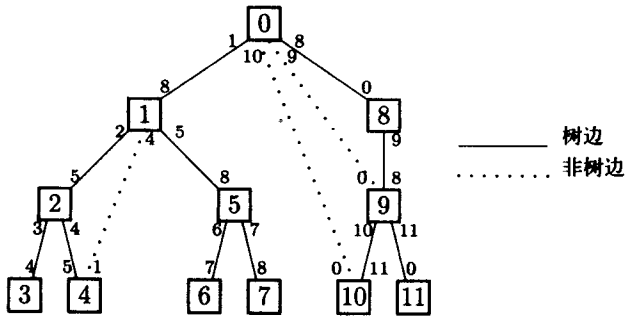


图4-15 深度优先搜索ILS 138

可以证明，深度优先搜索ILS是有效的模式。观察可见，$v \in T[u] \Leftrightarrow l_v \in [l_u, k_u)$。以下三个引理涉及节点$u$向节点$w$（$u$的近邻）转发带有目的节点$v$的包的过程。将目的节点为$v$的包转发给节点$w$。这蕴含着，对于某些$u$中的标号$\alpha$，$l_v \in [\alpha_{uw}, \alpha)$，且在节点$u$中，不存在标号$\alpha' \neq$

α_{uw}满足$\alpha' \in [\alpha_{uw}, l_v)$。

引理4.28　如果$l_u > l_v$，那么，$l_w < l_u$。

证明。首先考虑$\alpha_{uw} \leqslant l_v$的情形。节点$w$不是$u$的子节点，因为在那种情况下，$\alpha_{uw} = l_w > l_u > l_v$。如果$uw$是非树边，那么$l_w = \alpha_{uw} < l_v < l_u$。如果节点$w$是$u$的父节点，那么$l_w < l_u$在任何情况下都成立。其次，考虑$\alpha_{uw}$是$u$中具有最大标号的边，不存在标号$\alpha' \leqslant l_v$（即$l_v$是非线性区间的左边部分）。在这种情况下，与$u$的父节点相连的边标号不为0，而为$k_u$（因为$0 \leqslant l_v$，不存在标号$\alpha' \leqslant l_v$）。标号$k_u$是这种情况的最大标号；到子节点的边或者向下的非树边w'满足$\alpha_{uw'} = l_{w'} < k_u$，到祖先$w'$的非树边满足$\alpha_{uw'} = l_{w'} < l_u$。因此$w$是$u$的父节点，这蕴含着$l_w < l_u$。 □

下面两个引理涉及$l_u < l_v$的情形。推导出或者$v \in T[u]$或者$l_v \geqslant k_u$，后者的情形有$k_u < N$成立。因此到u的父节点的边用k_u标号。

引理4.29　如果$l_u < l_v$那么，$l_w \leqslant l_v$。

证明。首先考虑$v \in T[u]$的情形。设w'是u的子节点满足$v \in T[w']$，则有$\alpha_{uw'} = l_{w'} \leqslant l_v$。这蕴含着，$\alpha_{uw'} \leqslant \alpha_{uw} \leqslant l_v < k_{w'}$，我们推导出$w$不是$u$的父节点，因此，$l_w = \alpha_{uw}$，这蕴含着，$l_w \leqslant l_v$。

其次，考虑$l_v \geqslant k_u$的情形，在这种情况下w是u的父节点，可参见如下。到父节点的边标以k_u，$k_u \leqslant l_v$。到u的子节点的w'的边标以$l_{w'} < k_u$，向下的非树边w'标以$l_{w'} < l_u$。因为w是u的父节点，于是，$l_w < l_u < l_v$。 □

在v点传输的范函数定义如下。两个节点u和v的**最小公共祖先**（lowest common ancetor）
139 是树中的最小节点。设lca(u, v) 表示u和v的最小公共祖先的标号。定义

$$f_v(u) = (-\mathrm{lca}(u,v), l_u)。$$

引理4.30　如果$l_u < l_v$，那么，$f_v(w) < f_v(u)$。

证明。首先考虑$v \in T[u]$的情形，这蕴含着，lca(u, v) = l_u。如果w'是u的子节点，满足$v \in T[w']$，则有（由前述引理）$l_{w'} \leqslant l_w < k_{w'}$。因此$w \in T[w']$，这蕴含着lca($w$, v) $\geqslant l_{w'} > l_u$。于是，$f_v(w) < f_v(u)$。

其次，考虑$l_v \geqslant k_u$的情形，由前述引理，w是u的父节点，由于$v \notin T[u]$，lca(w, v) = lca(u, v)。但是现在$l_w < l_u$，因此，$f_v(w) < f_v(u)$。 □

现在可以证明，每一个包到达它的目的节点。图4-16表明了流向v的包流。设到v的包在节点u中产生。由引理4.28可得，节点标号每次跳跃后减小，经过有限次跳跃后，该包被节点w接收到，且$l_w \leqslant l_v$。由引理4.29，w之后包所转发到的每个节点，其标号也满足$\leqslant l_v$。由引理4.30，节点每次跳跃后f_v减小，或者包到达节点v，因此经过有限次跳跃后，节点v接到该包。这就完成了定理4.25的证明。 □

1. 区间路由效率：一般情形

定理4.25表明，每个网络存在有效的ILS，但这并不蕴含着，这种模式所选择的路径是有效的。深度优先搜索ILS用来表明网络中模式的存在性，但不一定是最好的可能的模式。例如，如果深度优先搜索被用到N个节点的环上，存在节点u和v且$d(u, v) = 2$，这种模式使用$N-2$次跳跃将包从u传递到v（习题4.8）。由定理4.34可得，对于同样的环，存在ILS通过最小跳跃路径传递每个包。

为了分析路由方法在这方面的质量，首先做出以下定义。

定义4.31　如果ILS经过最优路径转发所有包，则它是最优的。

如果ILS在一次跳跃中，将包从一个节点传递到该节点的近邻，则称ILS是近邻的。

如果与每条边对应的区间是线性的，则称ILS是线性的。

如果关于最小跳数（或者最短路径）代价度量是最优的，则称ILS是最小跳数（或最短路径）的。易证明，如果模式是最小跳数的，那么它是近邻的。容易验证，ILS是线性的，当且仅当在$l_u \neq 0$的各节点u中，存在标号为0的边，且在标号为0的那个节点中，存在标号为0或1的边。由此得出，对于一般的网络，路由方法的质量是不佳的，但是对于几类特殊的网络拓扑结构，这种模式的质量非常好。这使得该方法适合于具有规则结构的处理器网络，例如，那些用于实现具有全局共享存储器的并行计算机。

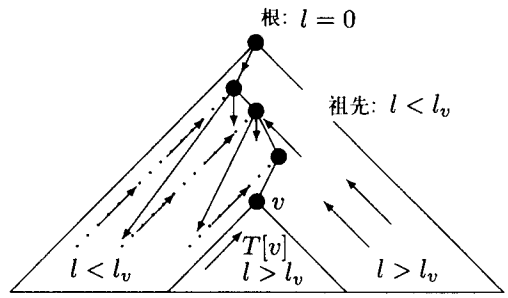


图4-16 深度优先搜索ILS中的包路由（节点v）

对于任意网络，还不清楚最好的区间标号模式与一个最优路由算法相比的差距。Ružička给出的路径长度下界，蕴含着最优的ILS并不总是存在。

定理4.32[Ruž88] *存在网络G，满足对于G的每一有效ILS，存在节点u和v，满足至少3/2 D_G次跳跃后，将包从u传输到v。*

目前还不清楚网络最好的深度优先搜索ILS与同网络的最优ILS比较的结果。习题4.7给出了一个较差的网络深度优先搜索ILS的例子，实际上，（由定理4.37）承认了它是最优的ILS。但对同一网络，也可能存在更好的深度优先搜索ILS。

在大多数通信是在近邻之间进行的情况下，成为近邻是ILS的充分条件。正如在图4.15中所见到的那样，深度优先搜索ILS不必是近邻的。节点4经过节点1转发包给节点2（近邻）。

140
~
141

2. 多路区间路由模式

如果允许对每一条边赋予一个以上的标号，可以改进路由算法的效率，就是我们所说的*多路区间路由模式*。这将边的目的节点集合定义为几个区间的并集。通过增加区间的数目，甚至对于任意网络，也可以达到最优路径。为了表明这一点，首先考虑最优路由表，例如，变更算法计算的最优路由表，通过某一边路由的目的节点的集合可以表示为循环区间的并集。利用这种方法，每一边可以找到至多$N/2$个标号，任一节点至多共有N个标号，这样一张表的存储空间，和存储整张路由表的存储空间相同。

可能以存储空间复杂度为代价，提高路由的效率。很自然地引出的问题是，要达到最优路由，网络实际需要多少标号。Flammini等人做了开创性的工作[FLMS95]，他们研究了一种寻找模式并提供较低界限的方法，这就是在一般情况下，每一条边有$\Theta(N)$个标号就能达到最优。但只允许在路径长度上进行小的折衷，就可以大大减小路由表的大小，有关这种权衡的详细结果参见Růzička所做的汇总[Ruž98]。

3. 线性区间路由模式

在区间路由方法中，考虑循环区间是重要的。尽管对某些网络的确有效，甚至是最优，线性区间标号模式不可能用线性区间对每一网络有效地标号。Bakker、Van Leeuwen和Tan研究了线性区间标号模式的可应用性[BLT91]。

定理4.33 *存在一种网络，不存在有效线性区间标号模式。*

证明。考虑长度为2的三条腿的蜘蛛图，如图4-17的描述。最小标号0和最大标号6赋给两个节点。因为有三条腿，所以至少有一条腿既不包含最小标号，也不包含最大标号。设x是这条腿上远离中心的第一个节点。节点x将地址为0和6的包转发到中心，包含0和6的惟一线性区间是整个集合$\mathbb{Z}_N$。因此，x同时也将包转发到朝向中心的它的其他近邻，这些包永远到不了目
142 的节点。 □

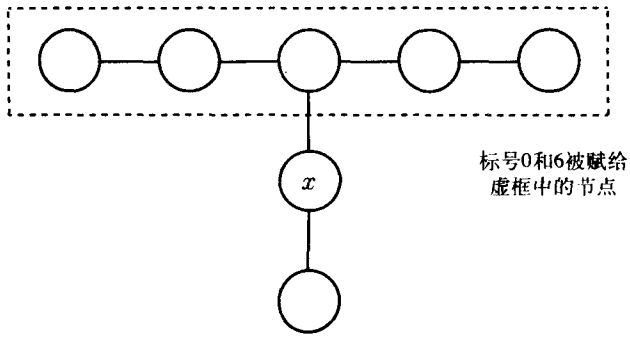


图4-17　三条腿的蜘蛛图

Bakker、Van Leeuwen和Tan全面地刻画了承认最短路径线性ILS的那类网络拓扑结构，提出了许多与承认自适应性和最小跳数线性ILS的拓扑结构相关的结果。Fragniaud和Gavoille[FG94]刻画了允许线性模式的网络。Eilam等人[EMZ96]证明了在这些网络中使用线性模式所得到的路径长度可与$O(D^2)$一样大。

4. 区间路由的最优性：特殊拓扑结构

对于几类规则结构的网络，存在最优区间标号模式。这些结构的网络可用于并行计算机的实现。

定理4.34[LT87] *对于N个节点的环网，存在最小跳数ILS。*

证明。对节点的标号按照顺时针方向从0到$N-1$赋值。对于节点i，顺时针信道被赋给标号$i+1$，逆时针信道被赋给标号$(i+\lceil N/2\rceil) \bmod N$；参见图4-18。用这种标号模式，标号为$i$的节点经顺时针信道将包发送到节点$i+1$，…，$(i+\lceil N/2\rceil)-1$，经逆时针信道将包发送到节点$(i+\lceil N/2\rceil)$，…，$i-1$，这是最优的。 □

143 由于定理4.34的证明，ILS是最优的，所以它是相邻的和非线性的。

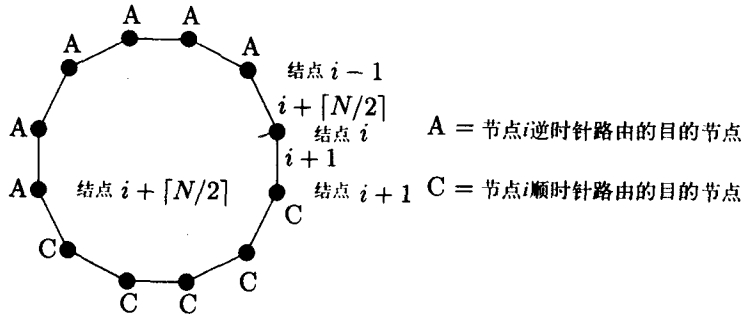


图4-18　环网的最优ILS

定理4.35[LT87] *对于$n\times n$的网格，存在最小跳数ILS。*

证明。以行为主序对节点的标号赋值。例如，第j行、第i个节点的标号为$(j-1)n+(i-1)$。节点的上信道标以0，左信道标以$(j-1)n$，右信道标以$(j-1)n+i$，下信道标以jn。参见图4-19。

容易验证，当节点u转发包给v时，

情形1：如果v所在的行高于u所在的行，那么u经过上信道发送包。

情形2：如果v所在的行低于u所在的行，那么u经过下信道发送包。

情形3：如果v与u在同一行，且在u的左边，那么u经过左信道发送包。

情形4：如果v与u在同一行，且在u的右边，那么u经过右信道发送包。

在任何情况下，u发送包到一个比v近的节点，这蕴含着所选路径是最优的。 □

因为定理4.35所证明的ILS是最优的，所以它是相邻的；标号模式也是线性的。

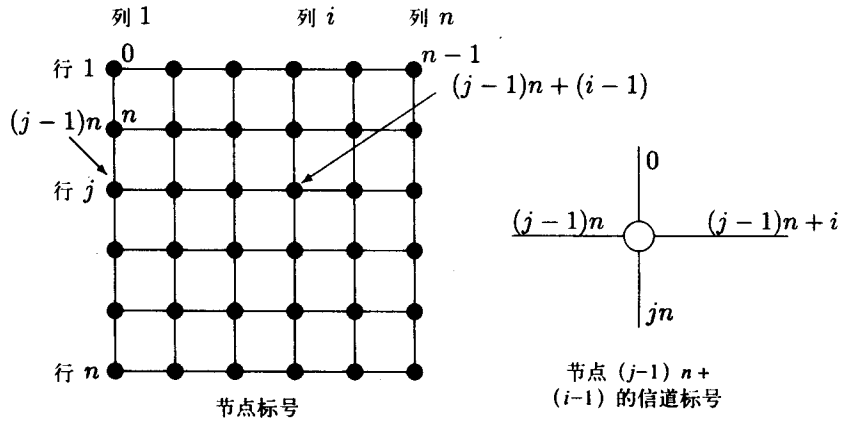


图4-19 $n\times n$网格的最优ILS

我们仅给出了以下两个结果，没有证明。定理中声称的标号模式的构造过程作为练习留给读者。

定理4.36 *对于超立方体网络，存在最小跳数线性ILS。*

定理4.37[FJ88] *对于具有任意信道权值的外平面网络，存在最短路径ILS。*

与分别存储每个目的节点输出信道的经典路由方法相比，区间路由有许多吸引人的优势。

（1）*低空间复杂度* 对于度为deg的节点，存储路由表的空间复杂度为$O(deg\cdot \log N)$位。

（2）*路由表计算效率高* 利用分布式深度优先搜索遍历网络，计算出深度优先搜索ILS的路由表。该计算利用了$O(E)$条消息，时间复杂度为$O(N)$。参见6.4节。

（3）*最优性* 对于几类特殊的网络，路由方法可以选择出最优路径。参见定理4.34到4.37。

这些优点使得方法适合于具有规则拓扑结构的处理器网络。例如，Transputers常常用于构造这样的拓扑结构。Inmos C104路由芯片（参见1.1.5小节）被设计使用区间路由。

令人遗憾的是，对于任意拓扑结构网络的应用，当使用深度优先搜索ILS时，就存在许多缺点。

144 ~ 145 （1）健壮性不好　信道或节点的加入网络或从网络中去除，都可能有点不适应深度优先搜索ILS。

ILS所基于的深度优先搜索树可能不再满足只在节点和它的祖先之间有非树边的要求。因此，网络拓扑结构的稍微改变就要求对路由表进行完全的重新计算，包括对每一节点重新赋给地址（标号）。

（2）非最优性　深度优先搜索ILS在进行路由时，经过的路径长度为$\Omega(N)$即使在网络直径较小时，参见习题4.7。

文献中讨论了许多区间路由的其他形式。以前讨论过的多路区间路由是一种相当实用的方法，因为随着现代存储技术的进步，拥有额外标号的代价将不再是问题。Flammini等人[FGNT98]描述了多维变量方法。多维空间中的节点名和标号是点和区间，而不是$\mathbb{Z}_N$。该方法为有多维结构，如产品图的网络带来了好处。

更灵活的一种利用网络结构的方法是由Flammini等人[FG S 93]提出的布尔路由算法(boolean routing)。其中节点名是位串，节点的标号是谓词；当且仅当$J(\lambda)$成立时，目的节点标号为λ的消息能够被通过标号为L的链路发送。该方法允许有效地使用相当复杂的网络结构，但对于标号的说明却相当复杂。

4.4.3 前缀路由

为克服区间路由算法的缺点，Bakker，Van Leeuwen和Tan[BLT93]设计了一种路由算法，利用任意生成树来计算路由表。非限制性生成树的使用既可以提高健壮性又可以提高效率。如果在两个现有节点间增加信道，生成树仍然是生成树，而新的信道是非树边。如果增加新的节点以及许多将它与现有节点连接的信道，使用其中的一个信道和这个新节点就可以扩展这棵生成树，其余信道是非树边。通过选择最小深度生成树，可以改进最优性，如引理4.22所示。

146 前缀路由中节点和信道所用的标号是串，而不是区间路由中使用的整数。设$\sum$是字母表，标号则是定义在字母表$\sum$上的串，$\in$表示空串，$\sum^*$表示$\sum$上串的集合。为了选择转发一个包所用的信道，算法考虑标号为目的节点地址前缀的所有信道。选择最长的标号，以及相应的用于转发包的信道。例如，假设有一节点，其信道标号为**aabb**、**abba**、**aab**、**aabc**和**aa**，要转发地址为**aabbc**的包。信道标号为**aabb**、**aab**，且**aa**是**aabbc**的前缀，这三种标号中最长的是**aabb**。因此节点经过信道**aabb**转发包。图4-20所示的算法给出了转发算法。我们用$\alpha \lhd \beta$表示α是β的前缀。

```
(* A packet with address d was received or generated at node u *)
if d = l_u
  then deliver the packet locally
  else begin α_i := the longest channel-label s.t. α_i ◁ d ;
              send packet via the channel labeled with α_i
       end
```

图4-20　前缀转发算法（节点u）

定义4.38　定义网络G的前缀标号模式（$\sum$上）为

（1）用$\sum^*$中不同字符串给G中节点赋值；且

（2）对于每一节点，用不同字符串给该节点的信道赋值。

前缀路由算法假设给定前缀标号模式（PLS），用图4-20所示的算法的转发包。

定义4.39 如果所有包按照这种方法最终转发到达目的节点，则称前缀标号模式是有效的。

定理4.40 对于每一个连通网络G，存在有效的PLS。

证明。定义一类前缀标号模式，并且在定理4.25中证明，这类模式是有效的。设T表示G的任意一棵有根生成树。

定义4.41 G的树PLS（关于树T）是一种前缀标号模式，满足以下规则。

（1）根节点的标号是$\in$。

（2）如果w是u的子节点，那么，用一个字符l_w可以扩展l_u，即，如果u_1，…，u_k是T中u的子节点，那么，$l_{u_i} = l_u \cdot a_i$，其中a_1，…，a_k是$\sum$中的k个不同字符。 147

（3）如果uw是非树边，那么，$\alpha_{uw} = l_w$。

（4）如果w是u的子节点，那么，$\alpha_{uw} = l_w$。

（5）如果w是u的父节点，那么，$\alpha_{uw} = \in$，除非u有到根的非树边，在这种情况下，$\alpha_{uw} = l_w$。

在树PLS中，除根节点之外，每一节点都有标以$\in$的信道，这个信道连接该节点与它的祖先（节点的父节点或树的根）。对于每一信道uw，或者$\alpha_{uw} = l_w$，或者$\alpha_{uw} = \in$。对于所有u和v，v是u的祖先，当且仅当$l_v \lhd l_u$。

首先必须证明，包不会“卡”在一个与其目的节点不同的节点处，即，与目的节点不同的每个节点都可利用图4-20所示的算法的转发包。

引理4.42 对于所有满足条件$u \neq v$的节点u和v，u中存在信道，以l_v的前缀标号。

证明。如果u不是T的根，那么u有标以$\in$的信道，它是l_v的前缀。如果u是T的根，那么v不是T的根，且$v \in T[u]$成立。如果w是u的子节点，满足$v \in T[w]$，那么由构造，$\alpha_{uw} \lhd l_v$。 □

利用图4-20所示的算法，以下三个引理考虑节点u为节点v向节点w（u的近邻）转发包的情况。

引理4.43 如果$u \in T[v]$，那么，w是u的祖先。

证明。如果$\alpha_{uw} = \in$，那么如前所述，w是u的祖先。如果$\alpha_{uw} = l_w$，那么因为$\alpha_{uw} \lhd l_v$且$l_w \lhd l_v$。这蕴含着w是v的祖先，也是u的祖先。 □

引理4.44 如果u是v的祖先，那么，w是v的祖先，比u更靠近v。

证明。设w'是u的子节点，满足$v \in T[w']$，则$\alpha_{uw'} = l_{w'}$是l_v的非空前缀。由于α_{uw}是l_v的最长前缀（在u中），由此可得$\alpha_{uw'} \lhd \alpha_{uw} \lhd l_v$，因此$w$是在$u$下面$v$的祖先。 □

引理4.45 如果$u \notin T[v]$，那么，w是v的祖先，或者$d_T(w, v) < d_T(u, v)$。

证明。如果$\alpha_{uw} = \in$，那么w是u的父节点或者根；因$u \notin T[v]$，所以u的父节点比u更靠近v，根是v的祖先。如果$\alpha_{uw} = l_w$，那么，因为$\alpha_{uw} \lhd l_v$，所以w是v的祖先。 □

设树T的深度为$depth$，即从根到任一叶节点最长简单路径的跳数。可见，目的节点为v的包到达其目的节点最多的跳数为$2 \cdot depth$。如果包是v的祖先产生的，那么由引理4.44，在 148
$depth$跳数内就可达到v。如果包是在子树$T[v]$内产生的，那么由引理4.43，在$depth$跳数内就可达到v的祖先，再经过另一个$depth$跳数达到v。（因为在这种情况下，路径只包含源节点的祖先，它的长度实际上也受到$depth$的限制。）由引理4.45，在所有其他情况下，在$depth$之内就可达到v的祖先，之后，在另一$depth$跳数之内就可达到v。（在这种情况下，路径长度受到

$2 \cdot depth$的限制。）这就完成了定理4.40的证明。 □

推论4.46　*对于直径为D_G的每个网络G（以跳数作为度量标准），存在前缀标号模式，至多在$2D_G$个跳数内传递所有包。*

证明。利用引理4.22所选择的树构造PLS即得。 □

最后简明分析树标号模式所需空间来结束本节。如前所述，设$depth$表示T的深度，k是树T中任一节点子节点的最大数。最长的标号由$depth$个字符组成，因为$\sum$至少包含k个字符，一个标号存储所需空间为$depth \cdot \log k$位。有deg条信道的节点的路由表的存储空间为$O(deg \cdot depth \cdot \log k)$位。

Bakker等人在文献[BLT93]中提出了其他一些前缀标号模式。他们的论文刻画了这类网络的拓扑结构，该拓扑结构使链路权值动态改变时，也有最优的前缀标号模式。

4.5　分级路由

一种减少路由中各种代价参量的方法是对网络进行分级划分以及使用相关分级路由方法。在多数情况下，目标是尽可能地利用这样一个事实，即，计算机网络中的很多通信都是局部的，如在距离相对小的节点间进行通信。路由方法的一些代价参数取决于整个网络的大小，而不是所选路径的长度，正如现在我们所解释的。

（1）*地址长度*　因为N个节点中，每个都有不同的地址，每个地址至少由$\log N$位组成。如
149 果信息按照地址进行编码，就需要更多位存储空间，如，前缀路由。

（2）*路由表大小*　在4.2节和4.3节中所描述的路由方法中，路由表中包含每一节点的条目，且路由表的大小是线性的。

（3）*查找路由表的代价*　对于大型路由表或者更大地址，单个路由表查找的代价可能较大。传输一条消息的查找总时间还取决于访问路由表的次数。

在分级路由方法中，网络被划分成*簇*，每一簇是节点的连通子集。如果包的源节点和目的节点在同一簇中，则转发消息的代价较低，因为在簇内路由时，把一簇看作相对较小独立的网络。对于4.5.1节中所描述的方法，每一簇中有一个固定的节点（簇的*中心*），如果需要将包发送到其他簇，它进行更复杂的路由决策。只在中心需要较长路由表和长地址的操作。每一簇又可分成若干子簇，以便获得节点的多级划分。

簇之间的每次通信不一定必须通过簇中心。这种设计类型也有缺点，如果簇中心发生故障，会影响整个簇。Lenfert等人[LUST89]描述了一种分级路由方法，其中每个节点都可以同等地向外簇发送消息。然而这种方法只利用了小规模的路由表，因为将整个簇（该簇中有不属于它的节点）看作了一个节点。Awerbuch等人[ABNL90]利用分级路由的范型构造了一类路由模式，允许在效率和空间复杂度之间进行权衡。

减少路由决策数　到目前为止所讨论的路由方法，要求在每一中间节点上做出路由决策。这意味着对于长为l的路由，要访问路由表l次。对于最小跳数路由策略，l由网络直径限定。但是对于一般无回路的路由策略（如区间路由），$N-1$是给出的最好界限。在本小节，我们讨论一种方法通过它，可以降低查找路由表的数目。

150 利用以下引理，讨论将网络划分成连通子簇的可能性。

引理4.47　*对于每个$s \leqslant N$，存在将网络分成簇C_1，…，C_m的划分方法，满足*

（1）每个簇是一连通子簇，

（2）每个簇至少包含s个节点，且

（3）每个簇的半径至多为$2s$。

证明。设D_1，…，D_m是不相交连通子图的最大集合，满足每一D_i的半径$\leqslant s$，至少包含s个节点。不在集合$\cup_{i=1}^{m}$中的每一节点，通过长度至多为s的路径与这些子集之一相连，否则增加路径作为独立簇。通过将不在集合$\cup_{i=1}^{m}$中的每一节点包含进与它最接近的簇，可以形成簇C_i。扩展的簇每个仍然包括至少s个节点，它们仍然是连通和不相交的。且半径至多为$2s$。 □

假设只有几种颜色可用，路由方法为每一个包赋予一种颜色。节点作用如下。依靠它的颜色，包或者立即通过固定信道转发（对应于颜色），或者调用更复杂的路由决策方法。它允许节点运用不同的协议处理包。

定理4.48[LT86] *对于N个节点的网络，存在路由方法，对于每个包至多要求$O(\sqrt{N})$次路由决策和使用三种颜色。*

证明。假设给定引理4.47所蕴含的划分。由于C_i的半径至多为$2s$，对于每一$v \in C_i$，每个C_i包含满足条件$d(v, c_i) \leqslant 2s$的节点c_i。设T是G的连接所有c_i的最小规模子树。因为T最小，它包含至多m个叶节点，因此它包含至多$m-2$个分支节点（度大于2的节点）；参见习题4.9。我们称T中的节点为中心节点（c_i）、分支节点和路径节点（其余节点）。

路由方法首先将包发至源节点所在簇的中心节点c_i（绿阶段）处，然后经过T到目的节点簇的中心节点c_j（蓝阶段），最后在C_j内到达目的节点（红阶段）。对于每簇的中心节点，绿阶段使用固定汇集树，不需路由决策。T的路径节点有两条依附树的信道，最终经过不接收包的树的信道将每一蓝包转发出去。分支节点和树T的中心节点必须经过路由决策。对于红阶段，在簇内利用最短路径路由方法，将该阶段路由决策数限制到$2s$。这使得路由决策数的界限为$2m-2+2s$，至多为$2N/s-2+2s$，选择$s \approx \sqrt{N}$，则得$O(\sqrt{N})$的界限。 □ 151

定理4.48建立了传输每个包所需的路由决策总数。但是在做决策时，这一结果不依赖于任何特定的算法。T中所用的路由方法可以是Santoro和Khatib的树路由模式。也可以将聚类原理进一步应用于T来减少路由决策数。

定理4.49[LT86] *对于N个节点的网络和每个正整数$f \leqslant \log N$，存在路由方法，对于每个包至多要求$O(f \cdot N^{1/f})$次路由决策和$2f+1$种颜色。*

证明。论证过程类似于定理4.48的证明。不是选择$s \approx \sqrt{N}$，而是反复地将构造过程用于树T（具有相同的簇大小s），树是一个具有少于$2m$个节点的连通网络。由于T的路径节点只将包从一个固定信道传输到另一个信道，因此可以忽略路径节点。

分簇重复f次。网络G有N个节点。一级分簇后所得的树至多有N/s个中心节点和N/s个分支节点，即，$N(2/s)$个关键节点。如果i级分簇后所得树有m_i个关键节点，那么$i+1$级分簇后所得树至多有m_i/s个中心节点和m_i/s个分支节点，即，$m_i(2/s)$个关键节点。f级分簇后所得树至多有$m_f = N(2/s)^f$个关键节点。

每一级的分簇增加两种颜色，因此，f级分簇要用$2f+1$种颜色。在最高级，至多需$2m_f$次决策，在目的节点簇中，每一级分簇需要s次决策。这使得路由决策数为$2m_f + fs$。选择$s \approx 2N^{1/f}$，则得$m_f = O(1)$。因此路由决策数$f \cdot s = O(f \cdot N^{1/f})$。 □

大约用$\log N$种颜色所导致的路由方法需做$O(\log N)$次路由决策。在这种情况下，对包颜色的检查也变成一种路由决策。但仅涉及规模较小的表（表长至多为$O(\log N)$）。实际上只需

152 要一小部分节点。

习题

4.1节

4.1　假设每次拓扑结构变化后，要用这样一种方式更新路由表，即，即使是在更新表的过程中，也是无回路的。当网络拓扑结构发生无限次变化时，这种方法能保证包总是被传输吗？

证明在拓扑结构不断变化的情况下，不存在路由算法能保证包被传输。

4.2节

4.2　有一名学生提出略去图4-6所示的算法中的<**nys**，w>消息的发送；他认为如果一个节点没有接到近邻的消息<**ys**，w>，那么节点知道这个近邻不是T_w中的子节点。

是否可以按照这种方法修改算法呢？此时算法的复杂度有何变化？

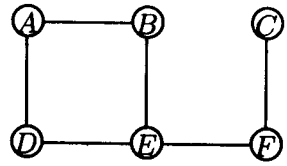
4.3　证明以下断言是Chandy-Misra算法计算到v_0的路径（图4-7所示的算法）的不变式。

$$\forall u,\ w:\ \langle \textbf{mydist},\ v_0,\ d\rangle \in M_{wu} \Rightarrow d(w,\ v_0) < d$$
$$\wedge\ \forall u:\ d(u,\ v_0) \leqslant D_u[v_0]$$

给出一个执行的例子，其中消息数与网络信道数成指数关系。

4.3节

4.4　如下图所示，变更算法用于下列拓扑结构的网络，给出算法的终止配置中所有变量的值。



在达到终止配置之后，在A和F之间加入一条信道。当处理通知<**repair**，A>时，F向A发
153 送什么消息？一旦接到F的这些消息，A发送什么消息？

4.4节

4.5　对于具有非对称信道代价的网络，给出一个引理4.22不成立的例子，并加以说明。

4.6　存在不需使用所有信道进行路由的ILS吗？存在有效的吗？存在最优的吗？

4.7　给出一个图G和G的深度优先搜索树T，满足G有$N = n^2$个节点，G的直径和树T的深度为$O(n)$，且存在节点u和v，满足在深度优先搜索ILS上的$N-1$次跳跃后，包从u被传递到v。

（可以用这种方法，选择图，即G是外平面图，（由定理4.37）蕴含着G实际存在最优ILS。）

4.8　给出N个节点环网的深度优先搜索ILS。找出节点u和v，满足$d(u,\ v) = 2$，且模式利用$N-2$次跳跃将包从节点u传输到节点v。

4.5节

4.9　证明：在定理4.48的证明中，树T的最小性蕴含着它至多有m个叶子。证明具有m个
154 叶子的树至多有$m-2$个分支节点。

第5章 无死锁的包交换

通过包交换通信网络进行遍历的消息(包)，在被转发至目的路径的下一节点时，必须存储在每一节点中。为此，网络中的节点预留一些缓冲区。 由于每个节点上的缓冲区数量有限，当下一节点的所有缓冲区已满，就会造成包不能够转发的情况，如图5-1中所示。四个节点都有B个缓冲区，每一缓冲区只能容纳一个包。节点s已将目的节点为v的B个包发送至节点t，节点v已将目的节点为s的B个包发送至节点u。现在u和v的所有缓冲区已满，导致存储在u和v中的包都不能被转发至其目的节点。

当一组包正在相互等待当前被组中的另一个包占据的缓冲区时，就会出现这组包永远不能到达目的节点的情况，我们称这种情况为*存储-转发死锁*(store-and-forward deadlock)。(其他类型的死锁将在本章最后讨论。) 在设计包交换网络过程中，重要问题是如何处理存储-转发死锁。本章论述几种称为*控制器*(controller)的方法，通过对包的产生或者转发进行限制来避免存储-转发死锁。在OSI参考模型的网络层可以找到避免存储-转发死锁的方法(1.2.2小节)。

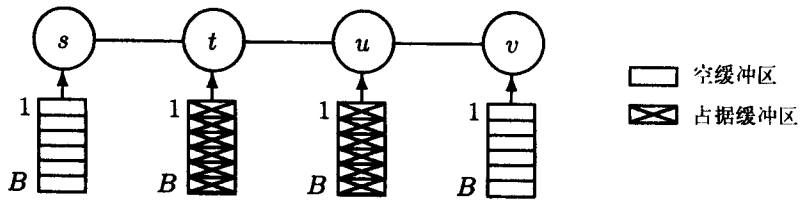


图5-1 存储-转发死锁示例

基于*有结构的*和*无结构的*缓冲池，本章主要讨论了两种方法。如果一个节点产生或接收数据包，利用有结构的缓冲池(5.2节)的方法为节点获取一个专门缓冲区。如果该缓冲区被占据，就不能接收包。在无结构的缓冲池(5.3节)方法中，所有缓冲区是平等的，方法仅指出是否能够接收包，并不决定要将包放在哪一个缓冲区中。5.1节引入了一些表示法和定义，5.4节对问题做了进一步的讨论，做为本章结束。

5.1 引言

通常，用图$G = (V, E)$模拟网络，用跳数度量节点间的距离。每一节点有B个缓冲区用来临时存放包。$\mathcal{B}$表示所有缓冲区的集合，符号b、c、b_u等用于表示缓冲区。

可用以下三种类型的迁移描述节点对包的处理。

(1) *产生* 节点u“创建”新包p(实际上是接收更高一层协议的包)，并把它放入u的空缓冲区中。节点u称为p的*源*。

(2) *转发* 包p从节点u被转发到路由的下一节点w的空缓冲区中(路由由所用的路由算法决定)。迁移的结果是，p以前所占据的缓冲区变空。尽管我们将要定义的控制器可能不允许迁移，但是假设网络总是允许这种迁移，即，如果控制器允许迁移，则它是可应用的。

在具有同步消息传递的系统中，很容易将这种迁移看作是定义2.7中的一次转移。在具有
155
~
156
异步消息传递的系统中，这种迁移并不是像定义2.6中的一次转移，但是可以实现它。例如，节点u不断地将p传输到w，只要没有接到确认，缓冲区就不丢弃该包。当节点w接收包时，它决定是否接收该数据包，并将它放入其中一个缓冲区。如果接收，将它放入缓冲区中，并向u发送确认消息，否则，简单地忽略该数据包。当然，可以设计更有效的协议实现迁移，例如，如果u知道w会接收p，u才将p传输给w。在这种情况下，迁移是由定义2.6中论及的几种转移组成的本章将把它们看作步骤之一。

(3) 消耗　占据目的节点缓冲区的包p被从缓冲区删除。假设网络总是允许这种迁移。

用$\mathcal{P}$表示数据包所通过的所有路径集合。这个集合由路由算法决定（参见第4章）。这里并不关注如何决定该集合。设k是$\mathcal{P}$中最长路径的跳数。并假设k不等于G的直径。如果路由算法没有选择最小跳数路径，k就可能超过网络直径。如果G中所有节点的通信是在有限距离的节点间，那么k可能小于该直径。

从本章开始所给出的例子来看，如果所有迁移不加限制的进行，就有可能出现死锁（如，阻止执行以下限制：如果u中产生包，u中必须有空缓冲区；如果将包转发到节点w，w中必须有空缓冲区）。我们定义控制器为这样一个算法：它允许或者禁止网络中的各种迁移，并要满足以下要求。

(1) 总是允许包的消耗（在目的节点）。

(2) 在所有缓冲区为空节点上，允许产生包。

(3) 控制器只利用局部信息，即，是否在节点u能接收一个包，只取决于u所获得的信息和包中的信息。

第二种情况排除了一些琐碎的方法，通过拒收网络中的任何包来避免死锁包（参见定义5.2）。如第2章所示，Z_u表示节点u的状态集，$\mathcal{M}$表示可能的消息(包)集。

定义5.1　网络$G=(V, E)$的控制器是序偶$\mathbf{con}=\{Gen_u, For_u\}_{u\in V}$的集合，其中$Gen_u \subseteq Z_u \times \mathcal{M}$和$For_u \subseteq Z_u \times \mathcal{M}$。如果$c_u \in Z_u$是$u$的一个状态，且$u$的所有缓冲区为空，那么对于所
157
有$p \in \mathcal{M}$，$(c_u, p) \in Gen_u$。

控制器**con**允许在节点u中产生包p，其中u的状态为c_u，当且仅当$(c_u, p) \in Gen_u$，允许从u向w转发包p，当且仅当$(c_w, p) \in For_w$。控制器的形式定义不包括包消耗的条件，因为(在目的节点)总是允许包消耗。在控制器**con**之下的网络迁移就是**con**所允许的网络迁移。

如果在网络的任何迁移序列中，都不能到达目的节点，称网络中的包是死锁的。

定义5.2　给定网络G、G的控制器**con**和G的配置γ，如果不存在**con**中的迁移序列，在γ中是可应用的，且p被消耗，称包p(出现在配置γ中)是死锁的。如果一个配置包含死锁包，称配置是死锁配置。

图5-1的例子表明，每个控制器都存在死锁配置。控制器的作用是避免网络进入死锁配置。网络的初始配置为空，即配置中不存在包。

定义5.3　如果从初始配置开始，在控制器作用下，不会达到死锁配置，则称控制器是无死锁的。

5.2 有结构的方法

我们将要讨论所谓的依赖缓冲图的一类控制器，这类控制器是由Merlin和Schweitzer

[MS80a]提出的。这些缓冲图的原理是基于这样一种观察，死锁是由于出现循环等待的情况（在无控制器情况下）。在循环等待的情况下，存在包的序列p_0，…，p_{s-1}，满足对于每一个i，p_i想要迁移到被p_{i+1}（下标计算为模s）所占据的缓冲区中。可以沿着非循环图（缓冲图）中的路径迁移包，来避免死锁发生。在5.2.1小节中，定义了缓冲图和有关的控制器。给出了两个简单的缓冲图的例子。在5.2.2小节中，给出了更复杂的缓冲图构造过程，同时给出了两个例子。

5.2.1 缓冲图

给定带有缓冲区集合$\mathcal{B}$的网络G。 158

定义5.4 缓冲图（对于G，$\mathcal{B}$）是定义在网络缓冲区上的有向图BG，即，$BG = (\mathcal{B}, \vec{BE})$，满足

（1）BG是无环的（不包含有向回路）。

（2）$bc \in \vec{BE}$，蕴含b和c在同一节点的缓冲区中，或者在G中信道连接两节点的缓冲区中，且

（3）对于每一路径$P \in \mathcal{P}$，BG中存在路径，它的映像是P。

由第二个要求可得，存在从BG中路径到G中路径的映射。如果b_0，b_1，…，b_s是BG中的一条路径，那么，如果u_i是缓冲区b_i所在的节点，则u_0，u_1，…，u_s是节点序列，满足对于每个$i < s$，或者$u_iu_{i+1} \in E$或者$u_i = u_{i+1}$。那么，由此序列去掉重复之后所得G中的路径称为BG中原始路径b_0，b_1，…，b_s的像。

包不能放入任意选择的缓冲区中；必须放在经过BG中的路径能够到达的目的节点的缓冲区中。即，按照如下定义，放在合适于包的缓冲区中。

定义5.5 设p是目的节点为v的节点u中的包。如果在BG中存在从b到v中缓冲区c的路径，则称u中的缓冲区b适合于p。它的像是G中p通过的一条路径。

BG中的这样一条路径被指定为保证路径，且$nb(p, b)$表示该保证路径上的下一个缓冲区。对于每一个u中新产生的包p，存在u中指定的合适缓冲区$fb(p)$。

这里fb和nb通过分别只取第一缓冲区(first buffer)和下一缓冲区(next buffer)的首字母而得。观察而得，缓冲区$nb(p, b)$总是适合于p。本节所用的所有缓冲图中，$nb(p, b)$所驻留的节点不同于b所驻留的节点。BG中“内部”边的使用，即，同一节点的两个缓冲区之间的边，将稍后讨论。

1. 缓冲图控制器

缓冲图BG可用于实现无死锁的控制器$\mathbf{bgc}_{BG}$，如果缓冲区$nb(p, b)$按照每一个包编码，或者按照p所驻留的节点状态编码。

定义5.6 控制器$\mathbf{bgc}_{BG}$定义如下。

（1）当且仅当缓冲区$fb(p)$为空时，u中可以产生包p。如果产生包，则放入此缓冲区中。

（2）当且仅当w中的$nb(p, b)$为空时，包p从u中的缓冲区转发到w中的缓冲区（可能$u = w$）。 159
如果发生转发，则将p放入$nb(p, b)$中。

定理5.7 控制器$\mathbf{bgc}_{BG}$是无死锁的控制器。

证明。如果节点u的缓冲区都为空，可允许u中产生包，这蕴含着，$\mathbf{bgc}_{BG}$是控制器。

对于$b \in \mathcal{B}$，按照BG中结束于b的最长路径长度定义b的缓冲级。观察可得，BG中路径上（尤其是在保证路径上）的缓冲区的缓冲级严格递增，即，$nb(p, b)$的缓冲级大于b的缓冲级。

由于控制器只允许将包放置在合适的缓冲区中，并且初始时没有包，$\mathbf{bgc}_{BG}$下，网络的每种可达配置只包含合适缓冲区中的包。通过对缓冲级运用向下归纳法，容易证明，在这种配置下，缓冲级r的缓冲区不包含死锁包。设R是最高的缓冲级。

情形 $r = R$: BG中具有最高缓冲级的节点u的缓冲区b没有输出边。因此，对于b为合适缓冲区的包，当它在b中时可被消耗，其目的节点为u。由此可得，缓冲级为R的缓冲区不含死锁包。

情形 $r < R$: 由归纳假设，对于每一满足条件$r < r' \leqslant R$的r'，缓冲级为r'的缓冲区不含死锁包（在可达配置中）。

设γ是可达配置，其中包p在节点u的缓冲级$r < R$的缓冲区b中，如果u是p的目的节点，那么p可被消耗，因此不会死锁。否则，设$nb(p, b) = c$是从b开始的保证路径上的下一缓冲区，观察可得，c的缓冲级r'超过r。由归纳假设，c不含死锁包，因此存在由γ可达的配置δ，其中c为空。在δ中，p可迁移到c，由归纳假设，在所得配置δ'中，p不会死锁。因此p在γ中不会死锁。 □

由证明过程可见，如果保证路径上包含缓冲图的“内部”边（同一节点两个缓冲区之间的边），那么，控制器一定允许另外的迁移，使得包可以被放在同一节点的其他缓冲区中。通常保证路径并不包含这样的边。它们仅被用作可选的迁移，以增加转发的效率。例如，下列
160 所述情形。包p驻留在节点u的缓冲区b_1中。节点w的缓冲区$nb(p, b_1)$被占据。然而，存在u中的空缓冲区b_2适合于p，而且节点w中的$nb(p, b_2)$为空。在这种情况下，包经过过b_2和$nb(p, b_2)$进行迁移。

下面讨论两个缓冲图应用的例子，即目的节点模式和当前跳数模式。

2. 目的节点模式

目的节点模式用每个节点u中的N个缓冲区，用缓冲区$b_u[v]$表示每个可能的目的节点v。称节点v为缓冲区$b_u[v]$的目标。假设这种模式的路由算法，经过根向v的有向树T_v转发目的节点为v的所有包。（实际上，这种假设可以放松。它满足向着v路由所用的信道形成G的无环子图。）

用$BG_d = (\mathcal{B}, \vec{BE})$定义缓冲图，其中，$b_u[v_1]\ b_w[v_2] \in \vec{BE}$，当且仅当$v_1 = v_2$且$uw$是$T_{v_1}$中的一条边。为表明$BG_d$是无环的，观察可见，不同目标的缓冲区之间不存在边。同一目标v的缓冲区之间形成与T_v同构的一棵树。终点为v的每一条路径$P \in \mathcal{P}$是T_v中的路径。通过构造，BG_d中存在目标为v的缓冲区路径，其映像为P。选此路径作为保证路径。这表明，对于目的节点为v的包p，在节点u中产生，$fb(p) = b_u[v]$。如果包必须被转发到w，则$nb(p, b) = b_w[v]$。

定义5.8 *控制器*dest*被定义为*$\mathbf{bgc}_{BG_d}$*，*fb*和*nb*如前段定义。*

定理5.9 *对于每个节点上使用N个缓冲区的任意连通网络，且允许包经任意选定的汇集树路由，则存在无死锁的控制器。*

证明。**dest**就是使用缓冲区数的无死锁的控制器。 □

如前所述，对于经过汇集树路由的要求可放松到，经过形成无环图的信道发送朝向某一目的节点的包。$\mathcal{P}$只包含简单路径是不足够的。正如图5-2所示的例子所表明的那样。这里从

u_1到v的包经过简单路径<u_1, w_1, u_2, ..., v>路由，从u_2到v的包经过简单路径<u_2, w_2, u_1, ..., v>路由。$\mathcal{P}$中每条路径是简单的，将包路由到v所用到的全部信道的集合包含回路<u_1, w_1, u_2, w_2, u_1>。

控制器**dest**简单易用，但缺点是要求每个节点有大量缓冲区N。可以定义双重源模式，其中以v_i为下标的缓冲区用来存储在v_i中产生的包。

3. 当前跳数模式（hops-so-far）

在当前跳数模式中，节点u包含$k+1$个缓冲区$b_u[0]$, ..., $b_u[k]$。假设每个包有一个跳数（hop count），用以表明包从源节点开始已经做了多少次跳跃。

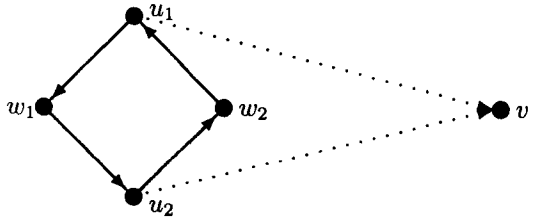


图5-2 控制器**dest**禁止的路由

用BG_h=（$\mathcal{B}$, $\vec{BE}$）定义缓冲图，其中，$b_u[i]$ $b_w[j] \in \vec{BE}$，当且仅当$i+1 = j$且$uw \in E$。为表明BG_h是无环的，观察可见，缓冲区的下标沿着BG_h中的每条边严格递增。因为$\mathcal{P}$中的每条路径至多有k个跳数长，因此，缓冲图中存在对应的路径；如果$P = u_0$, ..., u_l ($l < k$) 那么，

$$b_{u_0}[0], b_{u_1}[1], \ldots, b_{u_l}[l] 。$$

是BG_h中像为P的一条路径。用$fb(p) = b_u[0]$（p在u中产生）和$nb(p, b_u[i]) = b_w[i+1]$（包必须从u转发到w）描述这条保证路径。

定义5.10 *控制器***hsf***定义为***bgc***$_{BG_h}$，fb和nb如前段定义。*

定理5.11 *对于每个节点上利用$D+1$个缓冲区（D是网络直径）的任意连通网络，并且要求用最小跳数路径发送包，存在无死锁的控制器。* 161 ~ 162

证明。由最小跳数路径可得，$k = D$。**hsf**是每个节点上使用$D+1$个缓冲区的无死锁控制器。（如果距离较大的节点之间不需交换包，缓冲区数还可更小。） □

在当前跳数模式中，以i为下标的缓冲区用于存储当前经过i次跳跃的包。可以设计双重的下一跳数（hops-to-go）模式，其中以i为下标的缓冲区用于存储有i个以上朝向其目的节点的跳数的包。参见习题5.3。

5.2.2 图G的定向

本节考虑一种构造复杂缓冲图的方法，每个节点只需几个缓冲区。在当前跳数控制器中，存储包的缓冲区的下标随着每次跳跃增加。现在假设缓冲区下标（不要与缓冲级混淆）只随某些跳跃增加，不必随着所有跳跃而增加，这样可以减缓缓冲区下标的增长（也就减小了每个节点缓冲区的总量）。为避免缓冲图中的回路，那些遍历中不增加缓冲区下标的信道形成无环图。

定义5.12 *G的无环定向是对G中所有边进行定向所得的有向无环图。参见图5-3。*

G的无环定向序列G_1, ..., G_B是路径集合$\mathcal{P}$的大小为B的无环有向覆盖，如果对于每条路径$P \in \mathcal{P}$，都可以写成B条路径P_1, ..., P_B的连接，其中P_i是G_i中的一条路径。

当大小为B的无环有向覆盖可用时，可以构造每个节点上只使用B个缓冲区的控制器。包总是在节点u的缓冲区$b_u[1]$中产生。如果u和w之间的边是G_i中朝向w的有向边，缓冲区$b_u[i]$中

的包必须被转发到节点w，并放在缓冲区$b_w[i]$中。如果这条边是G_i中朝向u的有向边，缓冲区$b_u[i]$中的包必须被转发到节点w，并放在缓冲区$b_w[i+1]$中。

定理5.13　*如果大小为B的$\mathcal{P}$中存在无环有向覆盖，那么，存在无死锁的控制器，该控制器仅使用每个节点上的B个缓冲区。*

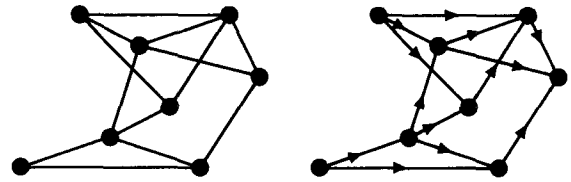


图5-3　图与无环方向图

证明。设G_1，…，G_B是这个无环有向覆盖，$b_u[1]$，…，$b_u[B]$是节点u的缓冲区。如果边uw是G_i中朝向w的有向边，则记为$uw \in \vec{E_i}$。如果边uw是G_i中朝向u的有向边，则记为$wu \in \vec{E_i}$。定义缓冲图为$BG_a = (\mathcal{B}, \vec{BE})$，其中，$b_u[i]\, b_w[j] \in \vec{BE}$，当且仅当$uw \in E$且$(i = j \wedge uw \in \vec{E_i})$或者$(i+1 = j \wedge wu \in \vec{E_i})$。为了表明图是无环的，注意不存在包含不同下标的缓冲区的回路，这是因为从给定缓冲区到另一个具有较小下标的缓冲区之间不存在边。具有同一下标i的缓冲区也不存在回路，因为这些缓冲区按照无环图G_i排列。

对于每一$P \in \mathcal{P}$，存在像为P的保证路径的证明留给读者（参见习题5.4）。用下列的定义描述这样一条路径:

$$fb(p) = b_u[1]$$

$$nb(p, b_u[i]) = \begin{cases} b_w[i] & \text{如果} uw \in \vec{E_i} \\ b_w[i+1] & \text{如果} wu \in \vec{E_i} \end{cases}$$

定义了每个节点上有B个缓冲区的缓冲图，我们可以得出结论，存在每个节点上使用B个缓冲区的控制器。定理证毕。　□

我们称所得到的控制器类为*无环有向覆盖控制器*，或者简称AOC控制器。以下继续表明，利用无环有向覆盖很容易设计控制器。

1. 环上的包交换

无环有向覆盖可用于设计几类网络的无死锁控制器。首先给出环上的控制器，每节点只需使用3个缓冲区。以下定理假设，信道的权值对称，即$\omega_{uw} = \omega_{wu}$。

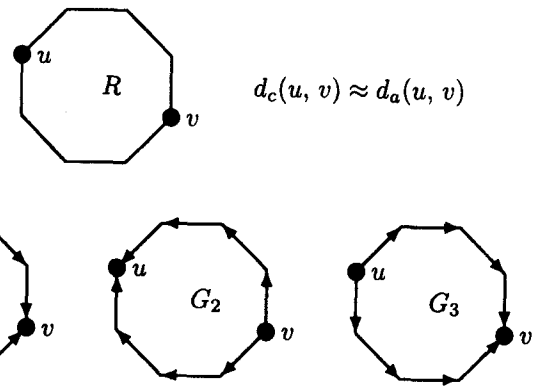


图5-4　环的无环方向覆盖

定理5.14　*对于每个节点有3个缓冲区并允许使用最短路径路由包的环网，存在无死锁的*

控制器。

证明。由定理5.13，对于每对节点间包含最短路径的路径集合，足以给出大小为3的无环有向覆盖。

利用以下表示。对于节点u和v，$d_c(u, v)$表示从u到v的顺时针路径长度，$d_a(u, v)$表示从u到v的逆时针路径长度，$d_c(v, u) = d_a(u, v)$且$d(u, v) = \min(d_c(u, v), d_a(u, v))$成立。$C$为所有信道权值总和（环的周长），显然，对于所有$u$和$v$，$d_c(u, v) + d_a(u, v) = C$，因此$d(u, v) \leqslant C/2$。

首先考虑简单情况。存在节点u和v，满足$d(u, v) = C/2$。G_1和G_3由所有指向v的边组成，G_2由所有指向u的边组成，参见图5-4。

从u到v的最短路径包含在G_1或者G_3中，从v到u的最短路径包含在G_2中。设x，y是不等于u、v的节点对。则当$d(x, y) < C/2$，存在节点x和y之间的最短路径P，不包含u和v。如果P既不包含u也不包含v，则该条路径要么完全在G_1中，要么完全在G_2中。如果P包含v，则该条路径是G_1中的一条路径与G_2中的一条路径的连接。如果P包含u，则它是G_2中的一条路径与G_3中的一条路径的连接。

如果不存在节点u和v，满足$d(u, v) = C/2$。选择路径$d(u, v)$与$C/2$尽可能接近的点对。可以表明，如果存在点对x，y，满足在方向覆盖中，不存在最短路径作为路径连接，那么较之$d(u, v)$，$d(x, y)$更接近$C/2$。 □ 163 ~ 165

2. 树上的包交换

无环有向覆盖可用于构造每节点有2个缓冲区的树网的控制器。

定理5.15 *对于每个节点有2个缓冲区的树网，存在无死锁的控制器。*

证明。由定理5.13，对于包含所有简单路径的树，足以给出无环有向图。任选节点r，通过将所有边指向r，得T_1，通过将所有边背离r，可得T_2，参见图5-5。从u到v的简单路径，是从u到最小公共祖先（在T_1中）的路径与从最小公共祖先到v（在T_2中）的路径的连接。 □

这里所描述的模式可用于任意网络的（最小深度）生成树中。实际上我们已经表明，对于每个节点上有2个缓冲区的网络，可能存在无死锁的路由。路由所用路径一般而言不是最优的。

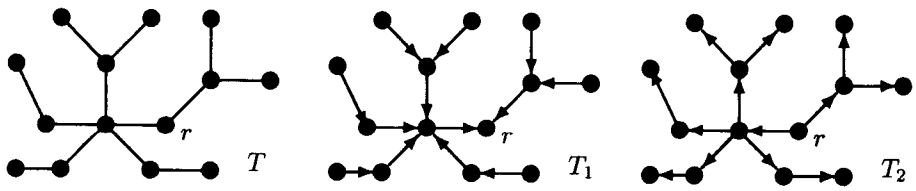


图5-5 树的无环方向覆盖

3. AOC控制器的应用

环的例子表明；使用AOC控制器，需要确定进行消息路由的具体路径集合。这样引出了几个问题，在过去的若干年里，人们一直在研究这些问题。

有可能只用最少量的缓冲区又通过最优路径同时路由包吗？如果不可能，可否在缓冲区和效率之间做出权衡呢？Štefankovič[Ste99]证明，AOC控制器在缓冲区的使用方面效率较低。他给出了一类网络，在每个节点上使用常数量的缓冲区，可以进行无死锁的最优路由。但是

最好的最优AOC控制器需要Ω(lg$N$/lglg$N$)个缓冲区。最好的AOC控制器和最好的控制器之
166 间的差距尚不得而知。

用压缩路由模式，如区间路由，可对AOC控制器中使用的路径编码吗？对于超立方体，可知（1）存在最优区间标号模式，（2）存在利用最优路径的控制器，每节点只需2个缓冲区，但是（3）对于只需2个缓冲区，并且所用最优路径利用区间标号模式编码的AOC控制器是不存在的。这个领域的大量研究仍然在进行。

5.3 无结构的方法

我们将要讨论由Toueg和Ullman[TU81]提出的一类控制器。这些控制器并不规定包必须被放进哪一个缓冲区，并且它们只用简单的局部信息，如，跳数或者节点所占据的缓冲区数。

5.3.1 前向计数控制器和后向计数控制器

1. 前向计数控制器

对于包p，设s_p是到目的节点仍需进行的跳数。显然，$0 \leq s_p \leq k$成立。在包中并不必总是维持s_p，因为许多路由算法将这一信息存储在每一节点中。如4.3节的变更算法。对于节点u，f_u表示u中空缓冲区数。$0 \leq f_u \leq B$总是成立。

定义5.16 *前向计数控制器***FC***在节点u中接受包p，当且仅当$s_p < f_u$。*

如果节点上包含的空缓冲区数比要进行跳跃的包多，则控制器接收包。

定理5.17 *如果$B > k$，那么，***FC***是无死锁的控制器。*

*证明。*为证明在空节点上可以产生包，观察可得，如果u中所有缓冲区为空，则$f_u = B$。新包至多要进行k次跳跃，因此$B > k$蕴含包被接收。

用反证法证明**FC**的无死锁性。假设γ是控制器的一种可达死锁配置。将γ用于最大转发和消耗迁移序列上，得到配置δ。在δ中包不能迁移。因为γ是死锁配置，在配置δ中，至少存在
167 一个包被留下。设p是δ中到目的节点最小距离的包，即，s_p是δ中任何包的最小值。

设u是包p所驻留的节点。由于u不是p的目的节点（否则，p在δ中被消耗），则必须将p转发至w，w是u的近邻。因为在**FC**中不允许这种迁移，

$$s_p - 1 \geq f_w$$

因为$s_p \leq k$，由假设，$k < B$，这蕴含着，$f_w < B$，这表明至少有一个包驻留在配置δ的w中。在w的所有包中，设q是w最新接收的包，f'_w表示w接收q之前的w中空缓冲区数。因为包q现在占据这些f'_w空缓冲区之一，（由q的选择）w中在q之后所接收的包都被去除，因此$f'_w \leq f_w + 1$。

w接收包q，蕴含$s_q < f'_w$，结合三个导出的不等式，可得

$$s_q < f'_w \leq f_w + 1 \leq s_p$$

这与p的选择矛盾。 □

2. 后向计数控制器

当按照包已做过的跳数来决定包的接收时，就得到对**FC**的“双重”控制器。设t_p是包p从源开始已进行的跳数。显然，$0 \leq t_p \leq k$总是成立。

定义5.18 *后向计数控制器***BC***在节点u中接受包p，当且仅当$t_p > k - f_u$。*

BC是无死锁控制器的证明（习题5.6）非常类似于定理5.17的证明。

5.3.2 前向状态控制器和后向状态控制器

通过利用驻留在节点中的更多信息，可以给出类似于前向计数控制器的控制器，但这样的控制器允许更多迁移。

1. 前向状态控制器

s_p的含义如前所述。定义节点u的状态向量为$<j_0, \ldots, j_k>$，其中j_s是u中包p的个数，且$s_p = s$。 168

定义5.19 前向状态控制器FS在节点u中接收状态向量为$<j_0, \ldots, jk>$的包p，当且仅当

$$\forall_i, 0 < i < s_p : i < B - \sum_{s=i}^{k} j_s$$

定理5.20 如果$B > k$，那么，FS是无死锁的。

证明。空节点接受每个包留给读者证明。假设存在可达的死锁配置γ，通过使用最大转发序列和消耗迁移，得到配置δ。在δ中包不能迁移，且至少存在一个包被留在δ中。选择具有最小s_p值的包p。设u是p所驻留的节点，w是p必须向其转发的节点。设$< j_0, \ldots, j_k >$是δ中w的状态向量。

如果w没有包含包，那么$\sum_{s=0}^{k} j_s = 0$，这蕴含着w能够接受p，然而，情况并非如此。因此，w至少包含一个包，在w的所有包中，设q是到目的节点最近距离的包，即$s_q = \min\{s: j_s > 0\}$。要证$s_q < s_p$，这是一个矛盾。

在w的所有包中，设r是w最新接收的包，则$s_q \leq s_r$成立。设$< j'_0, \ldots, j'_k >$是接收r之前w中的状态向量。r的接收蕴含着

$$\forall i, 0 < i < s_r : i < B - \sum_{s=i}^{k} j'_s$$

当$< j'_0, \ldots, j'_k >$是w的状态向量时，w接收r。其后，此包可从w中迁移，但是所有在r之后接收的包已经被删除（由r的选择），这蕴含着

$$j_s \leq j'_s \quad \text{对于} s \neq s_r$$
$$j_{s_r} \leq j'_{s_r} + 1$$

这蕴含

$$\forall_i, 0 < i < s_r : i < B - \sum_{s=i}^{k} j_s + 1$$

取$i = s_q$，

$$s_q < B - \sum_{s=s_q}^{k} j_s$$ 169

现在利用w没有接受p的事实，即，

$$\exists i_0, 0 < i_0 < s_p - 1 : i_0 \geq B - \sum_{s=i_0}^{k} j_s$$

由此得不等式

$$s_p > s_p - 1$$

$$
\begin{aligned}
&\geqslant i_0 && \text{如上所述}\\
&\geqslant B-\sum_{s=i_0}^{k} j_s && \text{如上所述}\\
&\geqslant B-\sum_{s=0}^{k} j_s && \text{因为}j_s \geqslant 0\ (\text{且}i_0 \geqslant 0)\\
&\geqslant B-\sum_{s=s_q}^{k} j_s && \text{因为}j_s = 0\text{对于}s < s_q\\
&\geqslant s_q
\end{aligned}
$$

是所要求的矛盾结果。 □

2. 后向状态控制器

与后向状态控制器比较，前向状态控制器利用更详细的信息并允许更多迁移。t_p的含义如前所述。定义节点u的状态向量为$<i_0, \ldots, i_k>$，其中i_r是u中已经做过t次跳跃的包的数目。

定义5.21 *后向状态控制器***BS***在节点u中接收状态向量为$<i_0, \ldots, i_k>$的包p，当且仅当*

$$\forall j, t_p < j < k : j > \sum_{t=0}^{j} i_t - B + k$$

BS是无死锁控制器的证明非常类似于定理5.20的证明。

3. 两种控制器的比较

在某种意义上，由于前向状态控制器允许更多迁移，因此前向状态控制器比前向计数控制器更宽松。

引理5.22 ***FC****所允许的每次迁移，****FS****也允许。*

证明。假设**FC**允许u接收包p。那么，$s_p < f_u = B - \sum_{s=0}^{k} j_s$，因此对于$i < s_p$，$i < B - \sum_{s=i}^{k} j_s$成立，这蕴含着**FS**允许迁移。 □

170 文献[TU81]中表明，**FC**比**BC**更宽松，**FS**比**BS**更宽松，**BS**比**BC**更宽松。经证明，在利用同样信息的控制器中，这4种控制器的每一种都可能是最宽松的控制器。

5.4 需进一步研究的问题

本章所述的结论，控制器所需的缓冲区数总是起着重要作用。通常存在这种情况，如果有更多缓冲区可用，则吞吐量增加。在无结构的方法中，对于缓冲区数目只有很低的限定，无需任何修改，即可使用更大数目的缓冲区，缓冲图中可静态或者动态地插入附加缓冲区。如果静态插入，每一缓冲区在缓冲图中有固定的位置，即，构造的缓冲图比5.2节提供的例子“更宽”。除了单一缓冲区*fb*(p) 或者*nb*(p, b) 之外，通常要确定几个缓冲区，以备路径开始或继续穿过缓冲图时使用。如果附加缓冲区是动态插入的，缓冲图的构造首先要包含尽可能少的缓冲区。我们称图中的缓冲区为*逻辑缓冲区*。在操作中，每个实际的缓冲区（称为*物理缓冲区*）可作为逻辑缓冲区用。其中必须保证，对于每个逻辑缓冲区，至少有一个物理缓冲区扮演它的角色。为避免死锁，用这种方法只需预留少量的缓冲区。其余缓冲区可灵活使用。

本章假设包的大小固定：缓冲区的大小相等，且只能包含一个包。也可假设包的大小不同来考虑此问题。5.3节的方法适合于Bodlaender提出的假设，参见文献[Bod86]。

5.4.1 拓扑变化

至今为止，我们还没有考虑，在包从源节点到目的节点路由的过程中，网络拓扑结构发

生变化的可能性。在出现这种变化后，每个节点的路由表将被更新，然后利用路由表中变化后的值对包进行转发。修改路由表的结果是，包行进的路径，可能是路由表变化之前从未有过的路径。也可能是这种情况，包的最终路由包含回路。 171

本章讨论的其对避免死锁的方法所产生的影响与直觉恰恰相反。**Dest**控制器，可以不加修改的应用，其正确性取决于$\mathcal{P}$所具有的只包含简单路径的性质。只对路径跳数上界作出假设的控制器，当用于这种情况时，需要关注其他一些问题。

1. 控制器Dest。

在最新拓扑结构变化后的有限时间内，路由表已经收敛到无回路的路由表。在路由表的计算过程中，即使存在回路等待的情形，当计算完成时，缓冲图再次无回路，所有包被存储在合适的缓冲区中。因此路由表收敛到不含死锁包的最终配置。

2. 跳数控制器

考虑一个控制器，它所依赖的假设是包至多做k次跳跃。可以选择足够大的k，即使在包从源节点到目的节点遍历的过程中，出现某些拓扑变化，也会以大概率保证每个包在k次跳跃内到达目的节点。对于在k个跳数之内到达目的节点的所有包，当前跳数、后向计数和后向状态控制器可以不加修改的应用。但是总有可能，由于拓扑变化和路由表的更新，使得一个包不能在k次跳跃之内到达目的节点。如果这种情况发生，包就被存储在一个不合适的缓冲区中，它将永远被阻塞在一个与目的节点不同的节点中。

利用有协议层次的高级协议，仍然可以解决这类问题。最简单的解决办法是丢弃这个包。从端到端传输协议的观点来看，包现在丢失了。但是传输层可以解决这种损失。参见3.2节。

为实现3.2节所做的假设，即包的最大生存期为μ，丢弃包也是需要的。如果转发包的时间至少为μ_0，对于包的端到端的生存期μ的限定，蕴含着对包所能进行的跳数的界定k为μ/μ_0。

5.4.2 其他类型的死锁

本章只考虑了一种存储转发的死锁。如果5.1节所做的假设是有效的，则存储转发死锁是惟一能够出现的死锁类型。然而，在实际的网络中，不能总是证明这些假设是正确的，正如Merlin和Schweitzer[MS80b]所指出的那样，这可能引起其他类型的死锁。Merlin和Schweitzer 172
考虑了4种类型的死锁，即，子代死锁、版本复制死锁、调步死锁和装配死锁。并表明通过扩展缓冲图的方法可以避免这些死锁。

1. 子代死锁

当网络中的包p能建立另一个包q时，可能出现子代死锁。例如，如果遇到信道发生故障，就向源节点报告。这就在新产生的包q和现有包p的转发或消耗之间，强加了一种因果关系。这就违背了5.1节的假设，即网络总是允许包的转发和消耗。

可以通过复制两个缓冲图来避免子代死锁。一个用来记录原始信息，另一个用来记录子代的次要信息。如果子代又建立了子代，也可以利用多级缓冲图。

2. 版本复制死锁

直到接到目的节点接收包的确认消息（端到端），源节点还持有包的副本，这时可能引起版本复制死锁。（比较3.2节基于定时器的协议，并假设序列in_p被存储在与被路由机制用作临时包的存储空间相同的存储空间中。）这就违背了5.1节的假设，即当占据缓冲区的包被转发

后，缓冲区应该为空。

我们给出了两个扩展的缓冲图法则。用这些法则可以避免版本死锁。第一种方法假设，只要循环等待原始消息和确认消息，就会出现版本复制死锁。解决办法是将确认消息作为子代，并在缓冲图中存储它们的单独副本。第二种方法，在大多数情况下，只需较少缓冲区。新产生的包被放在专用的源节点源缓冲区中，其中不能放置转发包。

3. 调步死锁

当网络包含的节点具有有限的内部存储区时，这些节点在其他一些消息被产生后，才消耗消息，这样会引发调步死锁。例如，电传打字机终端在接收下一些字符来显示之前，必须先要输出一些字符。这违背了假设：目的节点上的包总是被消耗。

173

我们可以通过区别可调步的（paceable）包和调步（pacing）响应，来避免调步死锁。例如，直到产生第二种类型的包后，前一种类型的包才能被消耗。用不同缓冲图的副本来表示这两种类型的消息。

4. 装配死锁

在网络中，将消息划分成小包进行传输，直到所有包达到目的节点时才能删除包，这会引起装配死锁。（比较3.1节的滑动窗口协议，其中，仅当具有较小下标的所有字被接收后，才可以从out_p中删除字。）这违背了包在目的节点总是可能被消耗的假设。

5.4.3 活锁

死锁包的定义（定义5.2）蕴含着，在无死锁的控制器下，每个包至少存在被消耗的一次计算。一般情况下，可能存在大量的不同计算，这并不蕴含着每个包最终都能够被传递到目的节点。即使有无限次计算，如图5-6所示。假设节点u有无限的包流要发送到节点v，每当w中的缓冲区空时，就接收u的下一个包。节点s中有到t的包，它不是死锁的，这是因为，每当w中的一个缓冲区变空时，就有继续计算的可能性。其中w接收包，并将包转发到t。尽管这种继续是可能的，但并不是必须这样做。因此包可能永远留在s中。这种情况称为活锁。

可以扩展本章所讨论的控制器，以避免活锁这种情况。

定义5.23　给定网络，控制器con和配置γ，如果存在一个无限的迁移序列，在γ中是可应用的，且p在其中不被消耗，称p是活锁的。

如果它包含活锁包，称配置γ为活锁配置。在控制器迁移下，如果不存在活锁配置，从不含包的配置可达，称控制器是无活锁的。

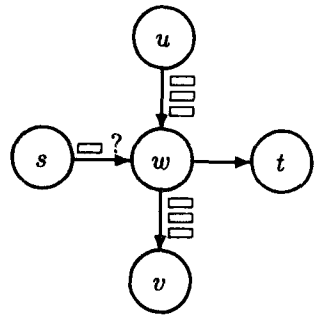


图5-6　活锁示例

本节其余部分，将证明缓冲图控制器的无活锁性。最后，简要提出了无结构方法的扩展。

1. 缓冲图控制器

可以证明，如果在一个无限序列中的迁移，满足许多公平性假设，5.2节的控制器不加修改就是无活锁的控制器。F1和F2是强公平性假设，F3是弱公平性假设。

F1　如果试图连续产生包p，那么每一无限计算都包含p的产生，在计算中$fb(p)$在无限多个配置中是空的。

F2 如果在配置γ中，包p必须经从u转发到w，那么，在γ中开始的每一无限计算都包含p的转发，在计算中$nb(p, b)$在无限多个配置中是空的。

F3 如果包p在配置γ中的目的节点中，那么，在γ中开始的每一无限计算都包含p的消耗。

引理5.24 如果满足公平性假设F2和F3，那么在**bgc**的每一次无限计算中，每一缓冲区为空的次数无限。

证明。对缓冲级用向下归纳法证明。正如在定理5.7中证明的那样，设R是最大的缓冲级。在**bgc**下可达的配置中，所有包驻留在合适的缓冲区中。

情形 $r = R$: 级R的缓冲区没有出边，因此，在这样的缓冲区中的包已经达到目的节点。由假设F3，每次配置后，这样的一个缓冲区被占据。存在配置，其缓冲区为空。这蕴含着在无限多次的配置中，缓冲区为空。

174
~
175

情形 $r < R$: 设γ是一种配置，其中级$r< R$的缓冲区b被包p占据。如果p已达其目的节点，将会有一配置，根据F3，其中的b为空。否则，p必被转发到缓冲级$r' > r$的缓冲区$nb(p, b)$中。由归纳得，在γ中开始的每一无限计算，该缓冲区为空的时间无限。由F2，这蕴含着，p将被转发，因此，γ之后将会有一配置，其中的缓冲区b为空。 □

定理5.25 如果满足公平假设F1、F2和F3，那么，在每次无限的计算中，出现在网络中的每个包将在目的节点被消耗。

证明。由引理5.24，在无限多次的配置中，所有缓冲区为空。由F1，这蕴含着，将为网络不断地提供包。由F2，消息到达目的节点就会被转发。由F3，消息在目的节点被消耗。 □

有人会假设，在分布式系统中，非确定性的选择机制保证了满足三条公平性假设。可供选择的是，通过向控制器加入一种机制来执行该假设，该机制保证，当缓冲区为空时，允许较老的包以较高优先级进入缓冲区。

2. 无结构的方法

必须修改5.3节的控制器，以便使其变为无活锁的。通过构造一个无限的计算可以对此加以证明，在这个无限的计算中，连续的包流导致控制器禁止某些包的转发。Toueg[Tou80b]给出了这样一个计算（对**FC**），并且对**FS**进行了修改（类似于在缓冲图控制器中提出的**FS**），得出它是无活锁的。

习题

5.1节

5.1 证明存在无活锁的控制器，每个节点只利用一个缓冲区，并且允许每个节点至少向其他节点发送一次包。

176

5.2节

5.2 如果包路由如图5-2所示，证明**dest**不是无死锁的。

5.3 （下一跳数模式）给出缓冲图和控制器fb与nb函数，控制器利用缓冲区$b_u[i]$存储那些距目的节点的跳数大于i的包。

$b_u[i]$的缓冲级是多少？每一个包都需要保存跳数吗？

5.4 完成图BG_a（在定理5.13的证明中定义）的确是一个缓冲图的证明，即，对于每一条

路径$P \in \mathcal{P}$，存在一条映像为P的保证路径。证明*fb*和*nb*确实可以描述为BG_a中的一条路径。

项目5.5　证明存在一个无死锁的控制器，对于超立方体上的包交换，每个节点只用两个缓冲区，并且允许包经过最小跳数路由。

可否利用区间路由算法（4.4.2小节）得到所用过的路径的集合?利用线性区间标号方法可以吗?

5.3节

5.6　证明**BC**和**BS**都是无死锁的控制器。

177　5.7　证明**BC**所允许的每次迁移，**FC**也允许。

第二部分 基本算法

第6章 波动算法与遍历算法

在设计各种应用的分布式算法中，进程网络中的几个非常一般性的问题经常作为子任务出现。这些基本任务包括：信息广播（例如一开始或者终止消息），达到进程间的全局同步，触发每个进程中某一事件的执行，或者计算一个函数，其中每一进程拥有部分输入。按照一些规定的，保证所有进程参与的拓扑结构模式，进行消息传递来执行这些任务。在以后的章节里，读者可以更加清晰地看到这些任务是如此基本，以至于对更复杂的问题如选举问题（第7章），终止检测（第8章）问题或者互斥问题的解决方法，其中进程之间的通信都经过这些消息传递模式完成。

消息传递模式的重要性，从现在起，被称为波动算法，将它们从模式被嵌入到特定应用中的算法分离出来，单独进行研究证明是正确的。本章形式地定义了波动算法（6.1.1小节），并证明了关于它们的一般结果（6.1.2小节）。观察可得，同一算法可用于上述所列的所有基本任务，即，广播、同步和计算全局函数（6.1.3小节到6.1.5小节）。6.2节提出了一些广泛使用的波动算法。6.3节讨论了遍历算法，它是具有另外一些性质的波动算法。算法中计算的事件具有因果全序关系。6.4节给出了分布式深度优先搜索的几种算法。

即使是波动算法经常用作其他算法的子程序，本章还是将它们作为独立的问题处理。原因有2个。首先，引入概念使得更多与算法有关的处理变得容易，因为子程序的性质已经得到研究。其次，分布式计算中的某些问题可通过一般的构造方法解决。当对某个波动算法参数化时，会产生特定算法。同一种构造可用于给出不同网络拓扑结构的算法，或者用于给出关于进程初始知识各种假设的算法。

178 ~ 181

6.1 波动算法的定义和使用

除非特别阐明，本章假设网络的拓扑结构固定（拓扑结构不发生变化），无向（每个信道可在两个方向传递消息）和连通（任何两个进程之间存在路径）。P表示所有进程的集合，E表示所有信道的集合。正如前面章节所述假设系统使用异步消息传递且没有全局时钟或者实时时钟的概念。本章算法也可用于具有同步消息传递（可能要做小的修改以避免死锁）的系统，或者如果可行，还可用于具有全局时钟的系统。然而，有些一般性的定理在这种情况下，不成立；参见习题6.1。

6.1.1 波动算法定义

正如第2章中所见的那样，由于进程以及通信子系统的非确定性，分布式算法通常允许进行大量可能计算。计算是事件的集合，是因果优先关系 $\preceq$ 的偏序，因果优先关系如同2.2.0中的定义。$|C|$表示计算C中事件的数量，C_p表示进程p中出现事件的子集。假设有一种特殊类型

的内部事件，称为判定事件。本章的算法中，这样的事件用语句*decide*简单表示。波动算法交换有限的信息，然后作出决策，这在因果关系上取决于进程中的一些事件。

定义6.1 *波动算法是满足如下三个要求的分布式算法。*

（1）**终止性** *每个计算是有限的:*

182

$$\forall C : |C| < \infty$$

（2）**判定性** *每次计算至少包括一个判定事件:*

$$\forall C : \exists e \in C : e \text{是一个判定事件}$$

（3）**相关性** *每次计算中，存在进程中的事件，该事件在因果关系上优先于判定事件。*

$$\forall C : \forall e \in C : (e \text{是一个判定事件} \Rightarrow \forall q \in P \exists f \in C_q : f \preceq e)$$

波动算法的一次计算称为一次波动。作为另一种表示，在算法的一次计算中，*初始进程*（initiator）也称*开始进程*（starter），*非初始进程*（non-initiator），也称*跟随进程*（follower）。如果进程自发地开始执行局部算法，就称它是初始进程，即，它的执行是由于进程内部的某些条件触发所致。只有当算法中的消息到达并引发进程算法的执行，非初始进程才执行算法。初始进程中的第一个事件是内部事件或者发送事件，非初始进程中的第一个事件是接收事件。

由于算法在诸多方面的不同，因而存在多种波动算法。作为处理本章许多算法的基本原理，以及作为选择特定问题的算法的助手，以下给出关于波动算法互不相同方面的一列表。也可参见表6-19。

（1）*集中*　如果在每次计算中，只有一个初始进程，则称算法是*集中式的*（centralized）。如果算法从进程的任意子集自发地开始执行，则称算法是*分散式的*（decentralized）。集中式算法也称为*单源*（single-source）算法，分散式算法称为*多源*（multi-source）算法。集中式算法对波动算法的复杂度具有重要影响。参见表6-20。

（2）*拓扑结构*　可能为某种特定的拓扑结构设计算法，例如环、树和团等。参见2.4.1小节和B.2节。

（3）*初始知识*　在进程中，假设算法可以利用各种初始知识，参见2.4.4小节。例子中的预备知识包括以下方面:

（a）*进程标识*　每个进程初始时知道自己的惟一名字。

（b）*近邻标识*　每个进程初始时知道它近邻的名字。

183

（c）*方向侦听*　参见B.3节。

（4）*判定次数*　本章所有的波动算法中，每个进程至多进行一次判定。执行一个判定事件的进程数可能不同。在某些算法中，只有一个进程判定，而在另外一些算法中，所有进程进行判定。树算法（6.2.2小节）只在两个进程中引起判定。

（5）*复杂度*　本章中所考虑的复杂度度量是交换消息的数量、交换位的数目和一次计算所需的时间（6.4节定义）。也参见2.4.5小节。

本章的每个波动算法中所用的变量一并给出，如有必要也要给出消息中所交换的信息量。在没有任何实际信息时（消息只带有因果关系，不是信息），大多数算法发送“空消息”。图6-9、图6-11、图6-12和图6-18所示的算法利用消息携带非平凡信息。图6-15、图6-16和图6-17所示的算法利用不同消息类型，这就要求每一消息包括一到二位用以区别不同类型的消息。

当使用波动算法时，消息中可能包含更多变量和其他一些信息。许多应用依靠几种波动同时或者顺序传播。在这种情况下，消息所属的那个波动信息必须包含在消息中。进程还可以保存另外的变量来管理波动，或者管理当前处于活动状态的那些波动。

波动算法的一个重要子类是由集中式波动算法构成的。具有下述的另外两个性质：初始进程是判定的惟一进程；所有事件是完全因果关系有序的。具有这些性质的波动算法称为*遍历算法*（traversal algorithm）。6.3节中讨论。

6.1.2 波动算法的一些基本结果

本节所给出的引理已被证明，这些引理可以使人们更深刻理解波动计算的结构。提出了波动算法消息复杂度的两个平凡下界。

1. 波动的结构性质

首先，对于计算中的每一事件，初始进程中的事件先于该事件。

引理6.2 *对于每一事件$e \in C$，存在初始进程p和C_p中的事件f，满足$f \preceq e$。* 184

*证明。*为f选择一个e的历史中的最小元素，即，$f \preceq e$且不存在事件$f' \prec f$。由于每个事件的历史是有限的，因此这样的事件f存在。仍然要证明，在f发生的地方，进程p是初始进程。首先，注意f是进程p中的第一个事件，否则就会有p中较早的事件领先于f。非初始进程的第一个事件是接收事件，会有一对应的发送事件先于该接收事件。这与f的最小性矛盾。因此，p是一初始进程。 □

一个有初始进程的波动定义了网络的一棵生成树。对于每一个非初始进程，选择信道，通过该信道接收第一条消息。

引理6.3 *设C是具有一个初始进程p的波动，且对于每一非初始进程q，设$father_q$是q的近邻，从该近邻，q可以在其第一个事件中接收消息。那么图$T = (\mathbb{P}, E_T)$，且$E_T = \{qr: q \neq p \wedge r = father_q\}$是一棵方向朝$p$的生成树。*

*证明。*由于T中的节点数比边数多一个，足以证明T不包含回路。这是因为，对于q中的第一个事件e_q，$qr \in E_T$蕴含着$e_r \preceq e_q$，且$\preceq$是偏序。 □

定义6.1中第三个子句中的事件f可以选择为所有q的发送事件，除了判定事件发生的那个进程之外。

引理6.4 *设C是一次波动，$d_p \in C$是进程p中的一个判定事件。那么*

$$\forall q \neq p : \exists f \in C_q : (f \preceq d_p \wedge f\text{是一个发送事件})$$

*证明。*由于C是一次波动，存在事件$f \in C_q$，在因果关系上先于d_p。选择f为C_q中先于d_p的最后事件。为证明f是一发送事件，观察因果关系定义（定义2.20），它的定义蕴含着存在序列（因果链）

$$f = e_0,\ e_1,\ \ldots,\ e_k = d_p,$$

满足对于每一个$i < k$，e_i和e_{i+1}或者是同一进程中的连续事件或者是对应的发送-接收事件对。由于f为q中先于d_p的最后一个事件。e_1出现在与进程q不同的一个进程中，因此f是一发送事件。 □

2. 波动复杂度的下界

由引理6.4立即可得，在一次波动中所交换的消息数的下界为$N-1$。如果判定事件出现在

185 波动的惟一初始进程中（在遍历算法中总是这种情况），这时下界为N条消息。任意网络的波动算法至少利用$|E|$条消息。

定理6.5　*设C是只有一个初始进程p的一次波动，满足判定事件d_p出现在p中。那么，C中至少交换N条消息。*

*证明。*由引理6.2，对于C中的每个事件，p中存在事件优先于它。利用因果关系序列，如，在引理6.4证明中的序列，容易证明p中至少出现一个发送事件。由引理6.4，发送事件也出现在其他各进程中。这使得发送事件的总数为N。□

定理6.6　*设A是任意网络的波动算法，没有近邻标识的初始知识。那么，每次计算中，A至少交换$|E|$条消息。*

*证明。*假设A有一个计算C。其中交换消息数少于$|E|$。存在信道xy，它不携带C中的消息。如图6-1所示。考虑在信道x和y之间插入节点z所得的网络G'，由于节点没有近邻标识知识，G'中x和y的初始状态如同在G中的初始状态。对于G中的所有其他节点同样成立。因此，C中所有的事件，从G'的初始配置开始，以同样的次序，都是可应用的。但是现在，存在z中的一个事件不先于判定事件发生。□

在第7章中，证明了无近邻知识的环网和任意网上，分散式波动算法消息复杂度的一个更好下界。参见推论7.14和7.16。

6.1.3 具有反馈的信息传播

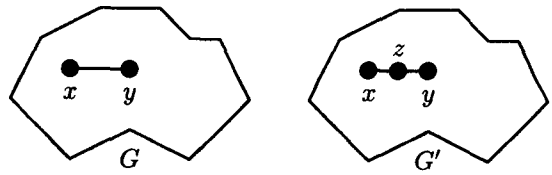


图6-1　未用信道中插入进程

本小节表明，当某些信息必须被广播到所有进程中，且完成广播后，某些进程必须接收通知消息，波动算法就是所需的算法。以下问题[Seg83]阐述了具有*反馈信息传播*（propagation of information with feedback，PIF）的要求。进程子集由那些具有消息M的那些进程组成（所有进程具有相同消息），且这些消息必须被广播，所有进程都接收并接受消息M。某些进程必须得到广播终止的*通知*，即，它们必须执行特殊的*通知*事件。要求当所有进程已经收到消息M，才能执行*通知*事件。算法必须利用有限个消息。

PIF算法的通知被认为是一个判定事件。

定理6.7　*每个PIF算法是一个波动算法。*

*证明。*设P是PIF算法。要求P中的每个计算是有限的。在每次计算中，出现一个通知事件（判定）。如果在P的某些计算中，通知d_p出现在进程q的任一事件之前，那么由定理2.21和事实2.23，存在P的一次执行，其中通知事件出现在q已经接受任何消息之前，这与要求发生矛盾。□

这里必须记住定理2.21仅对异步消息传递的系统成立。参见习题6.1。

定理6.8　*每个波动算法可用作一个PIF算法。*

*证明。*设A是波动算法。为把A作为PIF算法使用，初始时知道M的进程是A的初始进程。信息M被追加在A的每条消息后面。这是可能的，因为通过构造，A的初始进程开始时知道M，接收进程在接到消息之前，并不发送消息。因此接收进程也会知道M。波动中的判定事件在每个进程中的一个事件之后发生。当前者出现时，每个进程知道M，它被认为是PIF算法所要求的*通知*。

□

构造的PIF算法与算法A具有相同的消息复杂度。并且共享6.1.1节所提到的A的所有其他性质，除了位复杂度。可以只将M附加到通过每个信道发送的第一条消息中，来减少位复杂度。如果w是M的位数，则所得算法的位复杂度超过A的位复杂度一个因子$w \cdot |E|$。 186 ~ 187

6.1.4 同步

本小节表明，当要在进程间达到全局同步时，波动算法就是所需的算法。下述问题[Fin79]阐明了同步（Synchronization，SYN）的要求。在每个进程q中，必须执行事件a_q；在某些进程中，必须执行事件b_p，满足所有a_q事件的执行必须暂时地发生在所有b_p事件被执行之前。算法必须利用有限个消息。

在SYN算法中，将事件b_p看作判定事件。

定理6.9 *每一个SYN算法是一个波动算法。*

*证明。*设S是SYN算法。要求S中的每次计算是有限的，且在每次计算中事件b_p（判定）出现。如果在S的某些计算中，出现事件b_p，a_q不是因果关系上优先于事件b_p，那么，由定理2.21和事实2.23可得，存在S的一次执行，在这次执行中，b_p在a_q之前出现。 □

定理6.10 *每个波动算法可用作一个SYN算法。*

*证明。*设A是波动算法。为把A用作SYN算法，要求每个进程q在发送A的任何消息之前或者判定之前执行a_q。事件b_p发生在进程p中判定事件之后。由引理6.4，对于每个q，a_q在因果关系上出现在各判定事件之前。 □

构造的SYN算法与A具有相同的消息复杂度，并且共享在6.1.1节谈到的A的其他所有性质。

6.1.5 计算下确界函数

本节的波动算法就是这样一种算法，计算一个函数，它的值主要依靠每个进程的输入。作为这种函数的一个代表，我们考虑这样一种算法，由偏序集计算所有输入的下确界。

如果$(X, \prec)$是偏序，那么c称为a和b的下确界（infimum），如果$c \prec a$，$c \prec b$，且$\forall d$: $(d \prec a \wedge d \prec b \Rightarrow d \prec c)$。假设$X$具有性质，即它的下确界总是存在；在这种情况下，下确界是惟一的。用$a \curlywedge b$表示。现在$\curlywedge$是一个二元运算符，因为它是可交换（即，$a \curlywedge b = b \curlywedge a$）和可结合的（即，$a \curlywedge (b \curlywedge c) = (a \curlywedge b) \curlywedge c$），操作可以推广到有限集上： 188

$$\inf\{j_1, \ldots, j_k\} = j_1 \curlywedge \ldots \curlywedge j_k$$

下确界（INF）计算可以阐述为以下问题。每个进程q具有输入j_q，它是偏序集合X上的一个元素。要求某些进程计算下确界的值$\inf\{j_q: q \in \mathbb{P}\}$，并且这些进程知道计算什么时候终止。它们将计算的结果写入变量out中，以后不允许改变这个变量。

写事件赋值为out，被认为是INF中的一个判定事件。

定理6.11 *每一个INF算法是一个波动算法。*

*证明。*设I是INF算法。假设存在I中的计算C，具有初始配置γ，在计算中，p将值J写入out_p，q中任一事件都不在该写事件之前发生。考虑初始配置γ'，在这个配置中，除了事件q选择了不同输入j'_q并满足条件$J \not\preceq j'_q$之外，配置γ'与γ相等。因为没有一个使用q的输入的事件在

因果关系上先于C中p的写事件，在这个写事件之前的C中所有事件可被以从γ'开始的相同次序应用。因此，在最终计算中，如同在C中一样，p写入错误的结果J。 □

定理6.12 *每个波动算法可用来计算一个下确界。*

*证明。*设给定的波动算法A。每个进程q另设一个变量v_q，初始化为j_q。在一次波动中，该变量赋值如下。无论什么时候，进程q发送A的一条消息，v_q的当前值就包括在这条消息中，无论什么时候，进程q接到A的一条消息，消息中包括值v，就将v_q的值设为$v_q \curlywedge v$。当判定事件在p中执行时，v_p的当前值就写入out_p。

必须证明写入了正确值。称正确解为$J = \inf\{j_q\colon\ q \in \mathbb{P}\}$。对于进程$q$中的事件$a$，用$v^{(a)}$表示执行$a$后$v_q$的值。因为$v_q$被初始化为$j_q$，在波动过程中它的值只减小。对于$q$中的每个事件$a$，$v^{(a)} \leqslant j_q$成立。对$v$的赋值蕴含着，对于事件$a$和$b$，$a \preceq b \Rightarrow v^{(a)} \geqslant v^{(b)}$。同时由于$v$的值总是被计算
189 为两个已存在值的下确界，对于波动过程中的所有值，$J \leqslant v$成立。因此，如果d是p中的判定事件，那么值$v^{(d)}$满足$J \leqslant v^{(d)}$，并且每个进程q中至少有一个事件在d之前发生，对于所有q，$v^{(d)} \leqslant j_q$。这蕴含着，$J = v^{(d)}$。 □

所构造的INF算法共享了A的所有性质，除了位复杂度。因为X中的元素被附加到A的每条消息后面。下确界函数的表示看起来非常抽象，但是事实上许多函数可表示为下确界，参见[Tel91b，Theorem4.1.1.2]。

事实6.13（下确界定理） *如果$\star$是集合X上的二元算子，满足*

（1）*$\star$是可交换的，即$a \star b = b \star a$，*

（2）*$\star$是可结合的，即$(a \star b) \star c = a \star (b \star c)$，且*

（3）*$\star$是幂等的，即$a \star a = a$*

那么，存在X上的偏序$\leqslant$，满足$\star$是下确界函数。

满足这三个准则的运算符有：逻辑合取、逻辑析取、整数最小值或最大值，最大公因子、最小公倍数和集合的交或并。

推论6.14 *进程的局部运算符$\wedge$，$\vee$，min，max，gcd，lcm，$\cap$和$\cup$的值可在一次波动中计算出。*

具有可结合的和可交换的但不是幂等的运算符的计算在6.5.2节中讨论。

6.2 波动算法集

以下三节讨论波动算法和遍历算法的集。给出的所有算法都是指进程p的算法。

6.2.1 环网算法

本小节给出环网上的波动算法。同样的算法可用于哈密尔顿（Hamiltonian）网络，其中，一个确定的哈密尔顿回路被编码到进程中。假设对于每个进程p，给定它的一个专门的近邻$Next_p$，满足以这种方式选择的所有信道形成一个哈密尔顿回路。

算法是集中式的。初始进程沿着回路发送消息<tok>（称为令牌），每个进程传递消息，
190 当消息回到初始进程，初始进程进行判定。参见图6-2所示的算法。

定理6.15 *环网算法（图6-2所示的算法）是波动算法。*

*证明。*称初始进程为p_0。因为每个进程至多发送一条消息，算法总共至多交换N条消息。

在有限步内，算法达到终止配置。在这个配置中，p_0已经发送了令牌，即，在它的程序中

已经传递了发送语句。信道中没有<tok>消息的在传输中，否则消息会被接收，而且配置也不是终止。也没有除了p_0之外的进程“持有”令牌（即，已经收到令牌，但没有发送<tok>），否则这个进程就会发送<tok>消息，配置也不是终止的。总之，(1) p_0已经发送了令牌，(2) 对于每个发送了令牌的p，$Next_p$已经收到这个令牌，(3) 对于每个收到该令牌的进程$p \neq p_0$，p已经发送了令牌。由此以及*Next*的性质可得，每个进程已经发送和接收令牌。由于p_0已经收到令牌，配置是终止的，p_0执行了判定语句。

每个$p \neq p_0$的进程对于令牌<tok>的接收和发送，在p_0的接收之前，因此满足相关性条件。

□

```
For the initiator:
    begin send ⟨tok⟩ to Next_p ; receive ⟨tok⟩ ; decide end

For non-initiators:
    begin receive ⟨tok⟩ ; send ⟨tok⟩ to Next_p end
```

图6-2 环网算法

6.2.2 树算法

本小节给出树网上的波动算法。如果网络的生成树是可用的相同的算法可被用于任意网络。假设从树的叶节点开始执行算法。算法中，每个进程只发送一条消息。如果进程经过依附它的每一个信道接收一条消息，除了一种情况（初始时，这个条件对叶节点成立），则进程经过其余信道发送消息。如果进程经过所有依附的信道接收消息，则它进行判定。参见图6-3所示的算法。

```
var rec_p[q] for each q ∈ Neigh_p  : boolean init false ;
    (* rec_p[q] is true if p has received a message from q *)

begin while #{q : rec_p[q] is false} > 1 do
            begin receive ⟨tok⟩ from q ; rec_p[q] := true end ;
      (* Now there is one q_0 with rec_p[q_0] is false *)
      send ⟨tok⟩ to q_0 with rec_p[q_0] is false ;
   x: receive ⟨tok⟩ from q_0 ; rec_p[q_0] := true ;
      decide
      (* Inform other processes of decision:
         forall q ∈ Neigh_p, q ≠ q_0 do send ⟨tok⟩ to q *)
end
```

图6-3 树网算法

为证明此算法是波动算法，引入几种表示法。设f_{pq}是p向q发送消息的事件，设g_{pq}是q接收p的消息的事件。T_{pq}是从p不经过边pq可达的进程子集（在这条边上p这边的进程）。网络的连通性蕴含（参见图6-4）

$$T_{pq} = \bigcup_{r \in Neigh_p \setminus \{q\}} T_{rp} \cup \{p\} \text{ 和 } \mathbb{P} = \{p\} \cup \bigcup_{r \in Neigh_p} T_{rp}$$

图6-3所示的算法中注释符内的**forall**语句将在本小节最后讨论。下面定理是有关图6-3所

示的算法的，但除了这条语句。

定理6.16　*树网算法（图6-3所示的算法）是波动算法。*

*证明。*因为每个进程至多发送一条消息，算法中共利用至多N条消息。这蕴含着，在有限步后，算法到达终止配置γ。要证明在配置γ中，至少有一个进程执行了判定事件。

设F是γ中rec位值为假的个数，K是γ中已经发送了消息的进程数。因为在γ中，没有消息在传输中（否则γ将不会终止），因而$F = (2N-2) - K$；rec位的总数为$2N-2$，其中的K个为真。

假设γ中不存在进程已经进行判定，γ中还没有发送消息的$N-K$个进程中，每个至少有两个假的rec位；否则它们能够发送消息，这与γ是终止的相矛盾。在γ中已经发送一条消息的K个进程中，每个进程至少有一个假的rec位；否则它们就会判定，这与γ是终止的相矛盾。因此$F \geqslant 2(N-K) + K$，且$(2N-2) - K \geqslant 2(N-K) + K$，这表明，$-2 \geqslant 0$。矛盾。因此$\gamma$中至少发生一次判定。参见习题6.5。

191 ~ 192

最后，要证明每个进程中存在一个事件在判定事件之前发生。设f_{pq}是p向q发送消息的事件，设g_{pq}是q接收p的消息的事件。对接收事件用归纳法证明

$$\forall s \in T_{pq}\ \exists e \in Cs:\ e \preceq g_{pq}$$

假设上式对于g_{pq}之前的所有接收事件为真，现在f_{pq}先于g_{pq}（进程p中），p的程序蕴含对于所有$r \in Neigh_p$且$r \neq q$，g_{rp}先于f_{pq}。由归纳假设可得，对于所有这样的r，以及所有$s \in T_{rp}$，存在事件$e \in C_s$，且$e \preceq g_{rp}$，因此$e \preceq g_{pq}$。

对于所有$r \in Neigh_p$，g_{rp}在进程p中的判定事件d_p之前，这蕴含着$\forall s \in \mathbb{P}\exists e \in C_s:\ e \preceq d_p$。

□

读者可以在一棵较小的树上模拟算法的一次计算（例如，图6-4所示的子树），充分理解下列所讨论的事实。在图6-3所示的算法中，有两个进程经过它们的信道接收消息，并进行判定；其他所有进程仍在等待消息，它们的程序计数器指向终止配置中的x。如果将**forall**语句（图6-3所示算法的注释中）加入程序，所有进程都判定，在终止配置中，每个进程处于终止状态。修改后的程序总共利用$2N-2$条消息。

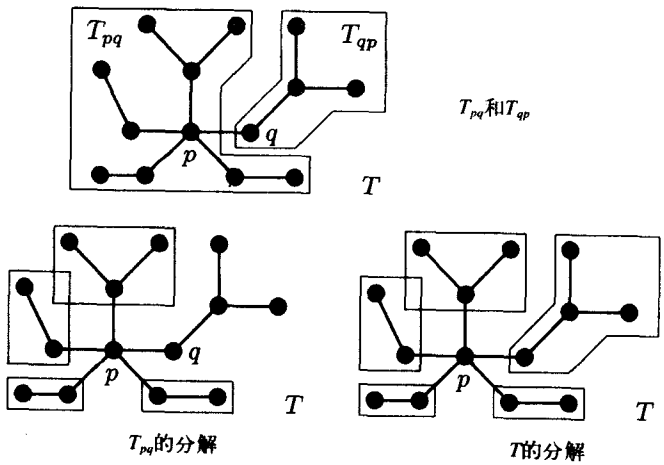


图6-4　子树T_{pq}

6.2.3 回波算法

回波算法是一种集中式的波动算法，适用于任何网络的拓扑结构。Chang[Cha82]首先独立提出了这个算法。因此有时称该算法为Chang的回波算法。Segall[Seg83]提出了一个稍微更有效的算法，本节给出这个算法。

算法将消息<tok>扩散到所有进程中，定义了一棵如同引理6.3中定义的生成树。令牌沿着这棵树边上的“回声”返回，就像树网算法中的消息流。图6-5所示的算法中给出这个算法。初始进程向所有它的近邻发送消息。一旦接到第一条消息，非初始进程就把消息转发给它的所有近邻，除了它收到消息的那个近邻。当非初始进程收到来自于所有近邻的消息时，就像它的父节点发送回波。当初始进程收到所有近邻的消息时，就进行判定。

```
var rec_p    : integer  init 0 ; (* Counts number of received messages *)
    father_p : ℙ        init udef;

For the initiator:
    begin forall q ∈ Neigh_p do send ⟨tok⟩ to q ;
          while rec_p < #Neigh_p do
                begin receive ⟨tok⟩ ; rec_p := rec_p + 1 end ;
          decide
    end

For non-initiators:
    begin receive ⟨tok⟩ from neighbor q ; father_p := q ; rec_p := rec_p + 1 ;
          forall q ∈ Neigh_p, q ≠ father_p do send ⟨tok⟩ to q ;
          while rec_p < #Neigh_p do
                begin receive ⟨tok⟩ ; rec_p := rec_p + 1 end ;
          send ⟨tok⟩ to father_p
    end
```

图6-5 回波算法

193
~
194

定理6.17 *回波算法（图6-5所示的算法）是波动算法。*

证明。 每个进程经过所依附的每条信道，至多只能发送一条消息，每次计算中所交换的消息数是有限的。设γ是初始进程为p_0的一次计算C中所达到的终止配置。

对于这个配置，定义（类似于定理6.3中的定义）图$T=(\mathbb{P}, E_T)$，$pq \in E_T \Leftrightarrow father_p = q$。为证明此图是一棵树，必须证明，图中的边数比节点数少1（引理6.3声明，T是一棵树，但假设算法是波动算法，这仍是我们这里要证明的）。观察可见，C中的每一进程发送消息到所有它的近邻中，除了（如果进程不是初始进程）它接收第一条消息的近邻。这蕴含着，每一个它的近邻至少接收C中的一条消息。并且加入到C中。由此可得，对于所有$p \neq p_0$的进程，$father_p \neq udef$。T不含回路的证明如同引理6.3中的证明。

树根朝向p_0。用T_p表示p的子树中的节点集合。网络中不属于T的边为非树边。在配置γ中，每个进程p至少会向它的所有近邻发送消息，除了它的父节点$father_p$，因此，在C中每条非树边都可以在两个方向携带消息。设f_p是p向其父节点发送消息的事件（如果它出现在C中），g_p是p的父节点接收来自p中消息的事件（如果它出现）。对这棵树中节点用归纳法，可以证明

（1）对于每个$p \neq p_0$的进程，C中包含事件f_p。

（2）对于所有$s \in T_p$，存在事件$e \in C_s$，满足$e \preceq g_p$。

我们考虑以下两种情况。

p是叶节点。p在C中已经接收来自父节点的消息，以及所有其他近邻的消息（因为所有其他信道都是非树边）。因此将令牌<tok>发送给p的父节点是可应用的，当γ终止时，它已经发生。T_p中只包含p，显然，$f_p \preceq g_p$。

p不是叶节点。在C中，p已经收到来自父节点的消息，并且经过所有非树边接收消息。由归纳假设，对于p的每个子节点p'，C包含$f_{p'}$。由于γ是终止的，C也包含$g_{p'}$。因此，令牌<tok>向p的父节点的发送是可应用的，当γ终止时，它已经发生。T_p是由$T_{p'}$、p的子节点及p自身的并集组成的集合。由归纳假设可以证明，对于这个集合中的每一进程，存在先于g_p的事件。

由此可得，p_0已经接收来自于每个近邻的一条消息，p_0已经执行了一次判定事件，每个进程中有一事件在这个判定事件之前。 □

195 图6-5所示的算法中的一次计算所构造的生成树有时用在随后执行的算法中。（例如，计算最短路径路由表的Merlin-Segall算法，假设初始时给出了根为v_0的生成树；参见4.2.3小节。可以利用回波算法计算初始的生成树。）在算法的最终配置中，每个进程（除了进程p_0）已经记录了树中哪一个近邻是它的父节点，但是没有记录哪个近邻是它的子节点。在算法中，经过非树边接收来自父节点的相同消息，并接收来自子节点的相同消息。如果要了解树中子节点的知识，就要稍微修改算法，以便能向父节点发送不同的消息（非初始进程进行的最后发送操作）。进程的子节点就是那些近邻节点，从这些近邻接收不同的消息。

6.2.4 轮询算法

在团网中，每一对进程之间存在信道。如果进程已经接收了每一近邻的消息，就进行判定。在图6-6所示的轮询算法中，初始进程要求每个近邻用一条消息进行应答，并在收到所有消息后进行判定。

定理6.18 *轮询算法（图6-6所示的算法）是波动算法。*

证明。经依附于初始进程的每条信道，算法发送两条消息。初始进程的每个近邻对于原始轮询只应答一次。因此，初始进程接到$N-1$个应答。这就是判定所需的准确数目，这蕴含着初始进程将做判定，且每一进程中有一个事件在该判定事件之前。 □

轮询算法可用于星型网络，在这种网络中，中心就是初始进程。

```
var rec_p  : integer    init 0 ;  (* for initiator only *)

For the initiator:
    begin forall q ∈ Neigh_p do send ⟨tok⟩ to qf ;
          while rec_p < #Neigh_p do
                begin receive ⟨tok⟩ ; rec_p := rec_p + 1 end ;
          decide
    end

For non-initiators:
    begin receive ⟨tok⟩ from q ; send ⟨tok⟩ to q end
```

196 图6-6 轮询算法

6.2.5 相位算法

本节所给出的相位算法，是一种分散式的算法，适合于任意网络的拓扑结构。算法在文献[Tel91b，4.2.3节]中论述。这个算法可用作有向网的波动算法。

算法要求进程知道网络直径，并在算法中将它作为常量D。如果进程利用大于网络直径的常量D'，算法仍然正确（尽管效率不是很高）。因此，为了应用这一算法，不需要精确地知道网络直径。如果已知网络直径的上界（例如，$N-1$）就足够了。所有进程必须利用同一个直径D'。Peleg[Pel90]扩充了算法，使得在执行中可以计算出网络直径。但是这种扩充要求有可用的惟一标识。

1. 一般情况

算法可用于任意的有向网中，信道只能在一个方向传递信息。在这种情况下，节点p的近邻有入近邻（in-neighbor，进程只能向p发送消息）和出近邻（out-neighbor，p只能向进程发送消息）。p的入近邻存储在集In_p中，p的出近邻存储在集Out_p中。

在相位算法中，每个进程向每个出近邻只发送D条消息。仅当收到来自每个入近邻的i条消息后，才将第（i+1）条消息发送给每个出近邻，参见图6-7所示的算法。

的确，由算法可见，通过每条信道至多发送D条消息（以下证明通过每条信道至少发送D条消息）。如果边pq存在，设$f_{pq}^{(i)}$是p向q发送消息的第i个事件，$g_{pq}^{(i)}$是q接收p的消息的第i个事件。如果p和q之间的信道满足fifo假设，这些事件通信，并且$f_{pq}^{(i)} \preceq g_{pq}^{(i)}$自然满足。如果信道不具有fifo性质，$f_{pq}^{(i)}$和$g_{pq}^{(i)}$之间的因果关系也成立，正如下列引理给出的证明那样。

引理6.19 *如果信道不具有先进先出性质，$f_{pq}^{(i)} \preceq g_{pq}^{(i)}$也成立。*

证明。 定义m_h满足$f_{pq}^{(m_h)}$是对应于$g_{pq}^{(h)}$的发送事件，即，在它的第h次接收事件中，q接收p的第m_h个消息。由因果关系的定义可得$f_{pq}^{(m_h)} \preceq g_{pq}^{(h)}$。 197

因为在C中，每条消息被接收一次，所有m_i不同。这蕴含着，在数m_1，…，m_i中，至少有一个大于或等于i。选择$j < i$满足$m_j > i$。现在$f_{pq}^{(i)} \preceq f_{pq}^{(m_j)} \preceq g_{pq}^{(j)} \preceq g_{pq}^{(i)}$。 □

定理6.20 *相位算法（图6-7所示的算法）是波动算法。*

证明。 因为每个进程通过信道至多发送D条消息，算法在有限步后终止。设γ是算法中计算C的终止配置，假设C中至少有一个初始进程（可能有多个）。

为表明在γ中每个进程已经做过判定，首先证明，每个进程至少已经发送一次消息。因为在γ的信道中没有消息，对于每个信道qp，就是$Rec_p[q] = Sent_q$的情况。同时，且当接到消息时，每个进程发送最新消息，$Rec_p[q] > 0 \Rightarrow Sent_p > 0$。假设存在有一个初始进程$p_0$，且$Sent_{p_0} > 0$，这蕴含着，对于每个$p$，$Sent_p > 0$。

随后证明每个进程已做判定。设p是γ中变量$Sent$为最小值的进程，即对于所有q，$Sent_q \geq Sent_p$。特别对于q是p的入近邻，成立。$Rec_p[q] = Sent_q$蕴含，$\min_q Rec_p[q] \geq Sent_p$。但是这蕴含着$Sent_p = D$；否则在最后一次接到消息的时候，$p$就会发送了附加消息。因而，对于所有$p$，$Sent_p = D$，且对于所有$qp$，$Rec_p[q] = D$，这表明每个进程确实已经作了判定。

接下来要证明，每个进程中存在一个事件在每次判定之前发生。如果$P = p_0, p_1, \ldots, p_l$，是网络中的一条路径，且$l < D$，那么由引理6.19，

$$f_{p_i p_{i+1}}^{(i+1)} \preceq g_{p_i p_{i+1}}^{(i+1)}$$

对于$0 \leqslant i < l$，由算法

$$g_{p_i p_{i+1}}^{(i+1)} \preceq f_{p_{i+1} p_{i+2}}^{(i+2)}$$

对于$0 \leqslant i < l-1$。从而，$f_{p_0 p_1}^{(1)} \preceq g_{p_{l-1} p_l}^{(l)}$。因为网络直径为$D$，对于每一$q$和$p$，存在长度至多为$D$的路径$q = p_0, p_1, \ldots, p_l = p$。因此对于每个$q$，存在$l \leqslant D$，且$p$的入近邻$r$满足$f_{qq'}^{(1)} \preceq g_{rp}^{(l)}$。由算法，$g_{rp}^{(l)}$在$d_p$之前。 □

```
cons D          : integer      = the network diameter ;
var  Rec_p[q]   : 0..D         init 0, for each q ∈ In_p ;
                 (* Number of messages received from q *)
     Sent_p     : 0..D         init 0 ;
                 (* Number of messages sent to each out-neighbor *)

begin if p is initiator then
        begin forall r ∈ Out_p do send ⟨tok⟩ to r ;
              Sent_p := Sent_p + 1
        end ;
      while min_q Rec_p[q] < D do
            begin receive ⟨tok⟩ (from neighbor q_0) ;
                  Rec_p[q_0] := Rec_p[q_0] + 1 ;
                  if min_q Rec_p[q] ⩾ Sent_p and Sent_p < D then
                    begin forall r ∈ Out_p do send ⟨tok⟩ to r ;
                          Sent_p := Sent_p + 1
                    end
            end ;
      decide
end
```

图6-7　相位算法

算法通过每个信道发送D条消息，使得消息复杂度为$|E| \cdot D$。应该注意，在这个表达式中，$|E|$表示有向信道的数量。如果将算法用于无向网络，每个信道相当两个有向信道。这使得消息的复杂度为$2|E| \cdot D$。

2. 团网中的相位算法

如果网络是直径为1的团。在这种情况下，只接收每个近邻的一条消息，每个进程只需计算它所接收消息的总数就足够了。而不需分别计算每个入近邻的消息数，参见图6-8所示的算法。消息复杂度为$N(N-1)$。算法仅用了$O(\log N)$位的内部存储空间。

6.2.6　Finn算法

Finn算法［Fin79］是另一种波动算法，可用于任意的有向网络中。它不要求预先知道网络直径，但是要求有可用的进程惟一标识。在消息中交换进程的标识集，这使得算法的位复杂度相当高。

198 ~ 199

进程p维持进程标识的两个集合，Inc_p和$NInc_p$。非形式地讲，Inc_p是进程q的集合，满足：q中的事件在p中最近事件之前发生，$NInc_p$是进程q的集合，满足对于所有q的近邻r，r中的事件在p中最近事件之前发生。关系维持如下。初始时，$Inc_p = \{p\}$和$NInc_p = \varnothing$。进程p发送包括Inc_p和$NInc_p$的消息，每次，其中一个集合增加。当p接收到包括Inc集和$NInc$集的消息时，所

收到的这些标识被插入到这些集合的p中。当p收到所有入近邻的一条消息时，p被插入到$NInc_p$中。当两个集合相等时，p进行判定。参见图6-9所示的算法。两个集合的非形式上的含义是，对于每个进程q，满足q中的事件先于d_p，对于q的每个近邻r，r中的事件也先于d_p，这蕴含算法的相关性。

在正确性证明中表明，对于每个p，两个集合相等蕴含着每个进程中存在事件在判定事件之前发生。

```
var Rec_p  : 0..N − 1  init 0 ;
           (* Number of messages received *)
    Sent_p : 0..1      init 0 ;
           (* Number of messages sent to each neighbor *)

begin if p is initiator then
        begin forall r ∈ Neigh_p do send ⟨tok⟩ to r ;
              Sent_p := Sent_p + 1
        end ;
      while Rec_p < #Neigh_p do
            begin receive ⟨tok⟩ ;
                  Rec_p := Rec_p + 1 ;
                  if Sent_p = 0 then
                    begin forall r ∈ Neigh_p do send ⟨tok⟩ to r ;
                          Sent_p := Sent_p + 1
                    end
            end ;
      decide
end
```

图6-8 团的相位算法

定理6.21 Finn算法（图6-9所示的算法）是波动算法。

证明。观察进程所保持的两个集合，只能增大。因为在第一条消息通过每个信道发送时，两个集合大小相加至少为1。在最后一条消息通过每个信道发送时，这两个集合大小相加至多为$2N$，因此消息总数界限为$2N \cdot |E|$。 200

设C是一次计算，其中至少有一个初始进程，设γ是最终配置，即终止配置。如同定理6.20的证明，可得如果进程p至少已经向每个近邻发送一次消息，q是p的出近邻，那么q也至少发送一次消息。这蕴含着每个进程至少已经发送一条消息（通过每个信道）。

要证在γ中每个进程已经判定。首先，如果存在边pq，那么在γ中$Inc_p \subseteq Inc_q$。在Inc_p的最后改变之后，进程p发送消息<**sets**，Inc_p，$NInc_q$>。在接收消息之后，进程q执行语句$Inc_q := Inc_q \cup Inc_p$。网络的强连通性表明，对于所有$p$和$q$，$Inc_p = Inc_q$。由于$p \in Inc_p$成立，每个$Inc$集合只包含进程标识，因此对于每个$p$，$Inc_p = \mathbb{P}$成立。

其次，对于每个p和q，类似地可以证明$NInc_p = NInc_q$。因为每个进程通过每个信道上至少发送一条消息，每个进程p满足$\forall q \in In_p$: $rec_p[q]$，因而，对于每个p，$p \in NInc_p$成立。同时，$NInc$集合只包含进程标识，这蕴含着对于每个p，$NInc_p = \mathbb{P}$成立。由$Inc_p = \mathbb{P}$和$NInc_p = \mathbb{P}$可得，$Inc_p = NInc_p$，因此每个进程p在γ中已经判定。

还需证明，每个进程中存在事件在进程p中的判定事件d_p之前发生。对于进程p中的事件e，设$Inc^{(e)}$（或$NInc^{(e)}$，分别地）表示执行事件e之后的Inc_p（或$NInc_p$，分别地）的值（参考定理

6.12的证明)。下面的两个声明形式化了本节开始时的非形式化的集合描述。

声明6.22　如果存在事件$e \in C_q$: $e \preceq f$，那么。$q \in Inc^{(f)}$。

证明。如同定理6.12的证明，由$e \preceq f \Rightarrow Inc^{(e)} \subseteq Inc^{(f)}$，且$e \in C_q \Rightarrow q \in Inc^{(e)}$。结果得证。 □

声明6.23　如果$q \in NInc^{(f)}$，那么，对于所有$r \in In_q$，存在事件$e \in C_r$: $e \preceq f$。

证明。设a_q是进程q第一次执行$NInc_q := NInc_q \cup \{q\}$的内部事件。事件$a_q$是$q \in NInc^{(a_q)}$中的惟一事件，并且对于满足$q \in NInc^{(a')}$的另一事件$a'$，$a'$不在事件$a_q$之前。于是$q \in NInc^{(f)} \Rightarrow a_q \preceq f$。

算法表明，对于每个$r \in In_q$，存在事件$e \in C_r$，在a_q之前。结果得证。 □

仅当$Inc_p = NInc_p$时，进程p才进行判定。记为$Inc^{(d_p)} = NInc^{(d_p)}$，它满足如下情形:

(1) $p \in Inc^{(d_p)}$; 且

(2) $q \in Inc^{(d_p)}$蕴含$q \in NInc^{(d_p)}$蕴含$In_q \subseteq Inc^{(d_p)}$。

由网络的强连通性可得，$Inc^{(d_p)} = \mathbb{P}$。

```
var Inc_p     : set of processes          init {p} ;
    NInc_p    : set of processes          init ∅ ;
    rec_p[q]  : boolean for q ∈ In_p       init false ;
                (* indicates whether p has already received from q *)

begin if p is initiator then
        forall r ∈ Out_p do send ⟨sets, Inc_p, NInc_p⟩ to r ;
      while Inc_p ≠ NInc_p do
            begin receive ⟨sets, Inc, NInc⟩ from q_0 ;
                  Inc_p := Inc_p ∪ Inc ; NInc_p := NInc_p ∪ NInc ;
                  rec_p[q_0] := true ;
                  if ∀q ∈ In_p : rec_p[q] then NInc_p := NInc_p ∪ {p} ;
                  if Inc_p or NInc_p has changed then
                    forall r ∈ Out_p do send ⟨sets, Inc_p, NInc_p⟩ to r
            end ;
      decide
end
```

图6-9　Finn算法

6.3　遍历算法

本节讨论一类特殊的波动算法。即，算法中的所有事件是因果关系上的全序。其中的最后一个事件与第一个事件出现在同一个进程中。

定义6.24　遍历算法是具有以下性质的算法。

(1) 每次计算只有一个初始进程，只发送一条消息开始执行算法。

201
~
202

(2) 进程一接到消息，就发送出一条消息，或者进行判定。

前两个性质表明，在每次有限计算中，只有一个进程判定。在一个进程判定后，称算法是终止的。

(3) 算法在初始进程终止，当这种情况发生时，每个进程至少已经发送一次消息。

在遍历算法的每个可达配置中，或者只有一条消息在传输，或者只有一个进程已经收到

消息并且还没有发送应答消息。从更抽象的观点来看，计算中的所有消息可被看作一个对象（令牌），被从一个进程传给另一个进程，因此能够“访问” 所有进程。在7.4节中，遍历算法用于构造选举算法。为此，不仅需要知道在一次波动中所传递的令牌总数，而且还要知道访问前x个进程需要传递的令牌次数。

定义6.25 *称算法是f-遍历算法（对于一类网络），如果*

（1）*算法是遍历算法（对于这类网络），且*

（2）*在每次计算中，传递$f(x)$次令牌之后，至少访问了$\min(N, x+1)$个进程。*

环网算法（图6-2所示的算法）是遍历算法，因为在x步后（$x < N$），$x+1$个进程已经处理了令牌，N步之后，所有进程已经处理了令牌。因此环网算法是x-遍历算法。

6.3.1 遍历团

用*顺序轮询*算法可以遍历团；算法非常像图6-6所示的算法。但是一次只对初始进程的一个近邻轮询。仅当接到一个近邻的应答之后，才轮询下一个近邻。参见图6-10所示的算法。

```
var rec_p : integer  init 0 ;  (* for initiator only *)

For the initiator:
    (* Write Neigh_p = {q_1, q_2, ..., q_{N-1}} *)
    begin while rec_p < #Neigh_p do
                begin send ⟨tok⟩ to q_{rec_p+1} ;
                      receive ⟨tok⟩; rec_p := rec_p + 1
                end ;
          decide
    end

For non-initiators:
    begin receive ⟨tok⟩ from q ; send ⟨tok⟩ to q end
```

图6-10 顺序轮询算法

定理6.26 *顺序轮询算法（图6-10所示的算法）是团的$2x$-遍历算法。*

*证明。*易见，当算法终止于初始进程时，每个进程都已经发送了应答。第（$2x-1$）条消息是对q_x的轮询，第$2x$条消息是它的应答。因此，当交换了$2x$条消息时，就已经访问了$x+1$个进程p，q_1，…，q_x。 □

6.3.2 遍历圆环

$n \times n$的圆环图是图$G = (V, E)$，其中

$$V = \mathbb{Z}_n \times \mathbb{Z}_n = \{ (i, j) : 0 \leq i, j < n \}$$

且

$$E = \{ (i, j)(i', j') : (i = i' \wedge j = j' \pm 1) \vee (i = i' \pm 1 \wedge j = j') \}$$

参见B.2.4节。（这里加减以n为模。）假设圆环可以进行方向侦听（参见B.3节），即，在节点(i, j)上到$(i, j+1)$的信道标以Up，到$(i, j-1)$的信道标以$Down$，到$(i+1, j)$的信道标以$Right$，到$(i-1, j)$的信道标以$Left$，坐标对(i, j)是定义网络拓扑结构和它的方

向侦听的便利工具但假设进程并不知道这些坐标。拓扑知识仅限于信道标号上。

圆环是哈密尔顿图，即在任意大小的圆环中存在哈密尔顿回路。利用图6-11所示的算法沿着这个回路发送令牌。在令牌的第k步后，如果$n \mid k$（n除k）为真，令牌向上发送，否则令牌向右发送。

```
For the initiator, execute once:
    send ⟨num, 1⟩ to Up

For each process, upon receipt of the token ⟨num, k⟩:
    begin if k = n^2 then decide
          else if n|k then send ⟨num, k + 1⟩ to Up
                      else send ⟨num, k + 1⟩ to Right
    end
```

图6-11　圆环遍历算法

定理6.27　*圆环算法（图6-11所示的算法）是圆环的x-遍历算法。*

*证明。*从算法中易见，在令牌传递n^2次之后，发生判定。如果令牌从p到q的移动是进行向上的U次移动和向右的R次移动，那么$p = q$当且仅当（$n|U \wedge n|R$）。设初始进程为p_0，p_k是接到令牌<**num**，k>的进程。

在传递令牌的n^2步中，有n步是向上的，其余的n^2-n步是向右的。因为n和n^2-n都是n的倍数，因此可得，$p_{n^2}= p_0$，因此算法在初始进程终止。

以下证明，进程p_0到p_{n^2-1}互不相同。因为有n^2个进程，这蕴含着，每个进程都被访问。假设对于所有$0 \leqslant a < b < n^2$，$p_a = p_b$。在p_a和p_b之间，令牌已经做了一些向上移动和一些向右移动，且由于$p_a = p_b$ 这两个方向的移动数都是n的倍数。由算法可见，

$$\#\{k : a \leqslant k < b \wedge n|k\}\text{是}n\text{的倍数}$$

和

$$\#\{k : a \leqslant k < b \wedge n \nmid k\}\text{是}n\text{的倍数}$$

这两个集合大小之和为$b-a$，由此可得，$n \mid (b-a)$。记$(b-a) = l \cdot n$，那么集合$\{k : a \leqslant k < b\}$包含$n$的$l$倍。这蕴含着，$n \mid l$。于是，$n^2 \mid (b-a)$，矛盾。

因为进程$p_0 .. p_{n^2-1}$互不相同。在令牌传递x次之后，访问了x+1个进程。　□

6.3.3　遍历超立方体

n维超立方体是图$G = (V, E)$，其中

$$V = \{(b_0, \ldots, b_{n-1}) : b_i = 0, 1\}$$

203
~
205

和

$E = \{(b_0, \ldots, b_{n-1}), (c_0, \ldots, c_{n-1}) : b\text{和}c\text{只在某一位上不同}\}$;

参见B.2.5小节。假设超立方体上可以进行方向侦听，（参见B.3节），即，在节点b和c之间的信道上标以i，其中b和c的二进制数在第i位不同。但假设进程并不知道这些节点标号。拓扑知识仅限于信道标号上。

就像圆环，超立方体是哈密尔顿回路，利用图6-12所示的算法可以遍历哈密尔顿回路。

算法的正确性证明几乎与图6-11所示的算法的正确性证明一样。

```
For the initiator, execute once:
    send ⟨num, 1⟩ through channel n − 1 .

For each process, upon receipt of the token ⟨num, k⟩:
    begin if k = 2^n then decide
          else begin let l the largest number s.t. 2^l | k ;
                     send ⟨num, k + 1⟩ through channel l
               end
    end
```

图6-12 超立方体遍历算法

定理6.28 *超立方体算法（图6-12所示的算法）是超立方体上的x-遍历算法。*

证明。 由算法可见，令牌在经过2^n次跳跃后，发生判定。设初始进程为p_0，p_k是收到令牌<**num**，k>的进程。对于每个$k < 2^n$，p_k和p_{k+1}的标号只有一位不同，用$l(k)$表示下标，满足

$$l(k) = \begin{cases} n-1 & \text{如果} k = 0 \\ \text{the largest } l \text{ with } 2^l \mid k & \text{如果} k > 0 \end{cases}$$

因为对于每个$i < n$，有偶数个$k \in \{0, \ldots, 2^n\}$且$l(k) = i$，$p_0 = p_{2^n}$，因此判定发生在初始进程中。类似于定理6.27的证明，可以证明$p_a = p_b$蕴含着$2^n \mid (b-a)$。这表明所有进程$p_0 \ldots p_{2^n-1}$互不相同。

由此可得，当终止时，所有进程已被访问过，在传递令牌x次之后，访问了x+1个进程。

□

6.3.4 遍历连通网络

Tarry[T1895]1895给出了连通网络的遍历算法。算法按照以下两个规则形成，参见算法6-13。

R1 进程从不在同一信道转发两次令牌。

R2 按照规则R1，仅当没有其他信道可用时，非初始进程才将令牌转发给其父节点（即非初始进程第一次从其处接到令牌的近邻节点）。

定理6.29 *Tarry算法（图6-13所示的算法）是遍历算法。*

证明。 因为令牌在各信道的每个方向至多发送一次，在算法终止之前，它至多被传递2|*E*|次。因为通过每个信道，每个进程至多发送一次令牌至多接收一次令牌。每次当非初始进程p持有令牌时，进程p的接收次数比发送次数多一次。这蕴含着依附于p的信道数超过p所用的信道数至少一个，于是p没有判定，而是转发令牌。由此可得，判定在初始进程发生。

我们分三步证明，当算法终止时，每个进程已经转发了令牌。

（1）*所有依附于初始进程的信道在每个方向已用一次。*

初始进程已经使用每个信道发送令牌，否则算法将不会终止。初始进程接收令牌的次数恰恰等于它发送令牌的次数。由于每次通过不同的信道接收，因此令牌通过每个信道被接收一次。

（2）*对于每个被访问的进程p，依附于p的所有信道在每个方向上已经被用一次。*

假设不是这种情况。选择p为第一个不满足这种性质的被访进程，由（1）可见，p不是初

始进程。由p的选择，所有依附于$father_p$的信道在每个方向上已被使用一次，这表明，p已经向其父节点发送了令牌。这蕴含着，p已经使用了所有依附的信道发送令牌；但由于令牌结束于初始进程。p收到的令牌数恰好和p发送的令牌数相等。因此p通过每个依附的信道接到一次令牌。矛盾。

```
var used_p[q]   : boolean   init false for each q ∈ Neigh_p ;
                            (* Indicates whether p has already sent to q *)
    father_p    : process   init udef ;

For the initiator only, execute once:
    begin father_p := p ; choose q ∈ Neigh_p ;
          used_p[q] := true ; send ⟨tok⟩ to q
    end

For each process, upon receipt of ⟨tok⟩ from q_0:
    begin if father_p = udef then father_p := q_0 ;
          if ∀q ∈ Neigh_p : used_p[q]
            then decide
          else if ∃q ∈ Neigh_p : (q ≠ father_p ∧ ¬used_p[q])
            then begin choose q ∈ Neigh_p \ {father_p}
                               with ¬used_p[q] ;
                       used_p[q] := true ; send ⟨tok⟩ to q
                 end
            else begin used_p[father_p] := true ;
                       send ⟨tok⟩ to father_p
                 end
    end
```

图6-13　Tarry的遍历算法

（3）所有进程已被访问，每个信道在两个方向上被用一次。如果存在未被访问的进程，则存在近邻p和q，满足p已被访问，q未被访问，这与p的各信道在两个方向已被使用发生矛盾。因此由（2）可得，所有进程被访问，所有信道在两个方向上只用一次。　□

Tarry算法的每次计算定义了引理6.3中所示的网络的一棵生成树。树根是初始进程，在计算的最后，每一个非初始进程p将其父节点存储在树中的变量$father_p$中。如果想要每个进程知道（在计算的最后）其近邻中哪些是其子节点，可以通过将特殊消息发给$father_p$实现。

6.4　深度优先搜索的时间复杂度

206
~
208

Tarry算法中的进程具有足够的自由度，在选择要向其转发令牌的近邻中，允许出现许多类生成树。本节要讨论的算法是如何计算具有其他性质的生成树，每一条非树边连接两个节点，其中一个是另一个的祖先。非树边是不属于生成树的边。给定G的一棵有根生成树T，$T[p]$表示p的子树中的进程集合，$A[p]$表示p的祖先，即，T中根与p之间的路径上的那些节点。观察可见，$q \in T[p] \Leftrightarrow p \in A[q]$。

定义6.30　对于每一非树边pq，$q \in T[p] \vee q \in A[q]$，$G$的有根生成树$T$是一棵深度优先搜索树。

深度优先搜索树可用于许多图的算法中，例如测试图的平面性，测试双连通性，区间标

号模式的构造（参见4.4.2节）。6.4.1节表明，对Tarry算法稍做修改（即，限制选择进程的自由度），就可以使算法用于计算深度优先搜索树。经修改的最终算法称为经典深度优先搜索算法（classical depth-first search algorithm）。6.4.2节讨论了两种深度优先搜索算法，它们能用比经典算法少的时间计算深度优先搜索树。因此，以下将定义分布式算法的时间复杂度。6.4.3节提出了一种具有初始近邻标识知识的网络的深度优先搜索算法。在这种情况下，不能应用定理6.6，实际上，给出了只用$2N-2$条消息的算法。

分布式算法的时间复杂度。异步网络的传输时间未知，我们的分析并不依赖于对系统有特定要求的参数。回忆一下在串行计算中，时间复杂度并没有表示成物理时间单元，而是表示成指令数。这里我们同样这样做。用执行中最长的消息延迟作为度量分布式执行期间的时间单位。用公理定义该时间单位，每个消息延迟都受到该时间单位的限定因此以下的T2并不真是关于计算的一种假设，而是关于时间单位的一种声明。

定义6.31 *分布式算法的时间复杂度定义为算法的一次计算的最大时间。满足下列假设：*

T1 *进程在0个时间内，可以执行任意有限个事件。*

T2 *消息的发送和接收之间的时间至多一个时间单位。*

本章所有波动算法的时间复杂度参见表6-19。表中没有证明的算法复杂度作为练习留给读者。6.5.3小节讨论另一种时间复杂度的定义。

引理6.32 *对于遍历算法，其时间复杂度等于消息复杂度。*

证明。消息的交换是顺序的，每次交换需要一个单位时间。 □

6.4.1 分布式深度优先搜索

如果用以下第3条规则限制Tarry算法中选择一个近邻来转发令牌的自由度，就得到经典深度优先算法，参见图6-14所示的算法。

R3 当进程收到所发送的令牌时，如果规则R1和R2允许，进程就通过相同信道将其发送回去。

定理6.33 *经典深度优先搜索算法（图6-14所示的算法）利用$2|E|$条消息和$2|E|$个时间单位，计算一棵深度优先搜索生成树。*

证明。作为实现Tarry算法的算法，它是遍历算法，并且计算一棵生成树T。已经证明，每个信道携带两条消息（每个方向一条消息），这就证明了消息的复杂度进而也可以证明时间复杂度这是。因为有$2|E|$条消息依次交换，每次交换至多花一个单位时间。接下来要证规则R3蕴含，其结果是一棵深度优先搜索树。

首先，规则R3蕴含，非树边的第一次遍历之后，相反方向紧跟着第二次遍历。假设pq是非树边，p是利用非树边的第一个进程。当q接到p的令牌时，q已被访问过（否则，q就会将$father_p$设置为p，边也不是非树边），且$used_q[p]$为假（因为由假设，p是这两个进程中的第一个利用该边的进程）。因此，由R3，q紧接着将令牌发回p。

可以证明，如果pq是首先被p使用的非树边，那么，$q \in A[p]$。考虑令牌所走过的路径，直到经pq将它发送。因为pq是非树边，在令牌经过pq这条边到达q之前，q已被访问过：

$$\ldots, q, \ldots, p, q$$

由此路径，通过用r_1替换所有模式r_1，r_2，r_1，就能得到更短的一条路经，其中r_1r_2是非树

边。根据以前的观察，所有非树边现在已被删除，这蕴含着所得路径是T中的一条路径，仅仅是由pq第一次使用之前所用的边组成。如果q不是p的祖先，这蕴含着从q到$father_p$的边在边qp被使用之前已经得到使用，这与算法中的规则2发生矛盾。 □

```
var used_p[q]   : boolean   init false for each q ∈ Neigh_p ;
                            (* Indicates whether p has already sent to q *)
    father_p    : process   init udef ;

For the initiator only, execute once:
    begin father_p := p ; choose q ∈ Neigh_p ;
          used_p[q] := true ; send ⟨tok⟩ to q
    end

For each process, upon receipt of ⟨tok⟩ from q_0:
    begin if father_p = udef then father_p := q_0 ;
          if ∀q ∈ Neigh_p : used_p[q]
            then decide
          else if ∃q ∈ Neigh_p : (q ≠ father_p ∧ ¬used_p[q])
            then begin if father_p ≠ q_0 ∧ ¬used_p[q_0]
                         then q := q_0
                         else choose q ∈ Neigh_p \ {father_p}
                                   with ¬used_p[q]) ;
                       used_p[q] := true ; send ⟨tok⟩ to q
                 end
            else begin used_p[father_p] := true ;
                       send ⟨tok⟩ to father_p
                 end
    end
```

图6-14　经典深度优先搜索算法

经典分布式优先搜索的消息复杂度为2|E|，如果近邻标识初始时未知，由定理6.6，它也是最优的（除了一个常数因子2）。算法的时间复杂度也是2|E|。由引理6.32，这可能是在这种情况下的遍历算法中最好的。Cheung[Che83]首先给出了深度优先搜索的分布式算法。

211

在6.4.2小节，在没有近邻标识知识的情况下，讨论了两种构造网络中深度优先搜索树的方法，其时间复杂度为O(N) 时间单位。因此这些算法不是遍历算法。在6.4.3小节，在利用近邻知识的情况下，可以得到算法的消息复杂度和时间复杂度为O(N) 的算法。

6.4.2　线性时间的深度优先搜索算法

经典深度优先搜索算法具有较高时间复杂度的原因是，所有边（树边和非树边）按照顺序遍历。令牌消息<tok>遍历所有非树边并且紧接着返回，正如定理6.33中证明的那样。具有较低时间复杂度的解决方案都是基于这样一个原理，即，令牌消息的遍历仅通过树边，显然遍历只需线性时间，因为只有$N-1$条树边。

1. Awerbuch解决方案

算法现在包括一种机制，禁止令牌通过非树边传输。在Awerbuch[Awe85b]算法中，保证当一个进程要转发令牌时，知道哪个近邻已被访问。然后进程选择一个未曾访问的近邻进行转发，如果近邻都已被访问过，就将令牌发给自己的父节点。

当令牌首次访问进程p时（对于初始进程，当算法开始时，就会出现这种情况），进程p通过向r发送消息<vis>，通知每个近邻r这次访问，除了它的父节点。令牌的转发将被挂起，直到p收到每个近邻的消息<ack>。这保证在进程p转发令牌的时刻p的每个近邻r知道p已被访问过。当稍后令牌到达r时，r不会将令牌转发给p，除非p是r的父节点。参见图6-15所示的算法。

在大多数情况下，消息<vis>的交换引起$used_p[father_p]$为真，即使是p还未将令牌发给其父节点时。因此，在算法中，必须进行详细的程序设计使得只有初始进程才能判定。对于所有近邻q，使得$used_p[q]$为真的非初始进程p将令牌转发给其父节点。

```
var used_p[q]   : boolean   init false for each q ∈ Neigh_p ;
                            (* Indicates whether p has already sent to q *)
    father_p    : process   init udef ;

For the initiator only, execute once:
    begin father_p := p ; choose q ∈ Neigh_p ;
          forall r ∈ Neigh_p do send ⟨vis⟩ to r ;
          forall r ∈ Neigh_p do receive ⟨ack⟩ from r ;
          used_p[q] := true ; send ⟨tok⟩ to q
    end

For each process, upon receipt of ⟨tok⟩ from q_0:
    begin if father_p = udef then
            begin father_p := q_0 ;
                  forall r ∈ Neigh_p \ {father_p} do send ⟨vis⟩ to r ;
                  forall r ∈ Neigh_p \ {father_p} do receive ⟨ack⟩ from r
            end ;
          if p is the initiator and ∀q ∈ Neigh_p : used_p[q]
            then decide
          else if ∃q ∈ Neigh_p : (q ≠ father_p ∧ ¬used_p[q])
            then begin if father_p ≠ q_0 ∧ ¬used_p[q_0]
                         then q := q_0
                         else choose q ∈ Neigh_p \ {father_p}
                                    with ¬used_p[q] ;
                       used_p[q] := true ; send ⟨tok⟩ to q
                 end
            else begin used_p[father_p] := true ;
                       send ⟨tok⟩ to father_p
                 end
    end

For each process, upon receipt of ⟨vis⟩ from q_0:
    begin used_p[q_0] := true ; send ⟨ack⟩ to q_0 end
```

图6-15 AWERBUCH深度优先搜索算法

定理6.34 *Awerbuch*算法（图6-15所示的算法）利用$4N-2$个时间单位和$4 \cdot |E|$条消息，计算一棵深度优先搜索生成树。

证明。如同图6-14所示算法，实际上通过相同信道发送令牌，除了通过非树边的发送以外。因为对任何进程p，通过非树边的传输并不影响$father_p$的最终值，其所得树总是图6-14所示的算法的一种可能结果。

令牌顺序遍历$N-1$条树边中的每条边两次，这需要$2N-2$个时间单位。在每个节点上，令牌被转发之前，至多等待一次。对于消息<vis>/<ack>的交换，在每个节点上引起至多两个时间单位的延迟。

每条非树边携带两条<vis>消息和两条<ack>消息。每条树边携带两条<tok>消息，一条<vis>消息（从父节点到子节点），一条<ack>消息（从子节点到父节点）。因此交换了$4 \cdot |E|$条消息。□

212 ~ 213

对于有进程向其转发令牌的近邻，消息<vis>的发送可以省略。这种改进（图6-15所示的算法中没有实现）可使每条树边节省两条消息，从而使得消息复杂度减小到$2N-2$。

2. Cidon的解决方案

通过不发送在Awerbuch算法中使用的＜**ack**＞消息，Cidon [cid88] 算法改进了Awerbuch算法的时间复杂度。在Cidon改进的算法中，令牌的转发不需等待，即没有在Awerbuch算法中因等待应答而引起的两个时间单位的延迟。Lakshmanan等人[LMT87]提出了相同算法。下面的情况可能出现在Cidon算法中。进程p已被令牌访问过，并且已将消息<vis>发给近邻r。令牌稍后访问r，但是r接到令牌时，p的消息<vis>可能还未到达r。在这种情况下，r可能将令牌转发给p，实际上，是通过非树边发送。（观察在Awerbuch算法中，消息<ack>是如何防止这种情况发生的。）

为了处理这种情况，进程p把它最新向近邻的令牌发送记录在变量mrs_p中。当令牌只对树边进行遍历时，下一次进程p从同一近邻mrs_p接收令牌。上面定义的这种情况中，进程p从不同的近邻接收<tok>消息，即从r接收。在这种情况下，忽略了令牌。但是进程p将边rp标记为使用过，就好像消息<vis>从r已经被接收。在向p发送令牌之后，进程r接收p的<vis>消息，即r接收近邻mrs_r的消息<vis>。在这种情况下，r的行为就好像它还没有向p发送令牌，r选择下一个近邻并转发令牌。参见图6-16或图6-17所示的算法。

```
var used_p[q]   : boolean   init false for each q ∈ Neigh_p ;
    father_p    : process   init udef ;
    mrs_p       : process   init udef ;

For the initiator only, execute once:
    begin father_p := p ; choose q ∈ Neigh_p ;
          forall r ∈ Neigh_p do send ⟨vis⟩ to r ;
          used_p[q] := true ; mrs_p := q ; send ⟨tok⟩ to q
    end

For each process, upon receipt of ⟨vis⟩ from q_0:
    begin used_p[q_0] := true ;
          if q_0 = mrs_p then (* Interpret as ⟨tok⟩ message *)
            forward ⟨tok⟩ message as upon receipt of ⟨tok⟩ message
    end
```

图6-16 CIDON深度优先搜索算法（PART 1）

定理6.35 Cidon的算法（图6-16或图6-17所示的算法）利用$2N-2$个时间单位和$4 \cdot |E|$个消息，计算一棵深度优先搜索生成树。

证明。令牌的遍历类似于图6-14所示的算法中的遍历。如在图6-15所示的算法中那样，或

者忽略通过非树边的传输，或者令牌携带消息<vis>穿过非树边。在后一种情况下，接收消息<vis>的进程继续转发令牌，这和令牌通过非树边被立刻发回的效果一样。

```
For each process, upon receipt of ⟨tok⟩ from q0:
    begin if mrs_p ≠ udef and mrs_p ≠ q0
            (* This is a frond edge, interpret as ⟨vis⟩ message *)
            then used_p[q0] := true
          else (* Act as in previous algorithm *)
            begin if father_p = udef then
                    begin father_p := q0 ;
                          forall r ∈ Neigh_p \ {father_p} do
                                send ⟨vis⟩ to r
                    end ;
                  if p is the initiator and ∀q ∈ Neigh_p : used_p[q]
                    then decide
                  else if ∃q ∈ Neigh_p : (q ≠ father_p ∧ ¬used_p[q])
                    then begin if father_p ≠ q0 ∧ ¬used_p[q0]
                                 then q := q0
                                 else choose q ∈ Neigh_p \ {father_p}
                                       with ¬used_p[q] ;
                               used_p[q] := true ; mrs_p := q ;
                               send ⟨tok⟩ to q
                         end
                    else begin used_p[father_p] := true ;
                               send ⟨tok⟩ to father_p
                         end
            end
    end
```

图6-17 CIDON深度优先搜索算法（PART 2）

令牌通过树边的两次连续传输的时间限定为一个时间单位。如果令牌沿着树边在时刻t被发送到p，那么在时刻t，所有以前访问过的p的近邻q的<vis>消息已被发送。因此这些消息在最近$t+1$时刻到达。即使在时刻$t+1$之前，p已通过非树边发送几次令牌，那么在时刻$t+1$，最近的p也已经从所有这些错误中恢复，并通过树边转发令牌。因为必须遍历$2N-2$条树边，因此，算法在$2N-2$个时间单位后终止。 214 ~ 215

每个信道至多传输两条<vis>消息和两条<tok>消息。这就证明，消息的复杂度为$4 \cdot |E|$。

□

在许多情况下，算法中发送的消息数要少于Awerbuch算法中的消息数。在分析Cidon算法的消息数时，假设最乐观的情况，即令牌消息通过非树边在两个方向发送。在避免许多不必要的传输方面，可以认为<vis>消息是成功的，在那些情况下，只有两到三条消息通过每个信道传输。

根据Cidon观察，即使算法可能发送令牌给以前访问过的节点，它也有比图6-15所示的算法更好的（通信）时间复杂度，因为它避免了不必要的传输。这表明从不必要行为中恢复过来所花费的时间和消息较少。Cidon将它作为一个公开问题，即，是否存在DFS算法，其消息复杂度达到经典算法的消息复杂度，即$2|E|$，而且其利用$O(N)$时间单位。

6.4.3 具有近邻知识的深度优先搜索

如果进程知道近邻的标识，通过在令牌中包括一张访问过的进程的表，就可避免令牌对非树边的遍历。进程p接到带有表L的令牌，并不将令牌转发给L中的进程。可以省略变量$used_p[q]$。因为如果p以前已经把令牌转发给q，那么，$q \in L$；参见图6-18所示的算法。

```
var father_p   : process   init udef ;

For the initiator only, execute once:
    begin father_p := p ; choose q ∈ Neigh_p ;
          send ⟨ tlist, {p} ⟩ to q
    end

For each process, upon receipt of ⟨ tlist, L ⟩ from q_0:
    begin if father_p = udef then father_p := q_0 ;
          if ∃q ∈ Neigh_p \ L
            then begin choose q ∈ Neigh_p \ L ;
                       send ⟨ tlist, L ∪ {p} ⟩ to q
                 end
          else if p is initiator
            then decide
            else send ⟨ tlist, L ∪ {p} ⟩ to father_p
    end
```

图6-18　具有近邻知识的深度优先搜索算法

定理6.36　*具有近邻知识的DFS算法是遍历算法，利用2N−2条消息和2N−2个时间单位，计算一棵深度优先搜索生成树。*

算法的位复杂度高。如果w是表示一个标识需要的位数，则表L可能需要$N \cdot w$位。参见习题6.14。

6.5 遗留问题

6.5.1 波动算法综述

表6-1给出了本章所考虑的波动算法。以Number为表头的列表示本章中的算法编号，以C/D为表头的列表示算法是集中式算法（C）还是分散式算法（D）。列T表示算法是否是遍历算法。M列表示消息的复杂度，Time为表头的列表示时间复杂度。在这些列中，N表示进程数，$|E|$是信道数，D是网络直径（按跳数）。

大多数拓扑结构网络中，波动的复杂度在相当程度上依赖于算法属于哪一种类型，是集中式算法还是分散式算法。表6-2列出了环、任意网和树的集中式和分散式算法的消息复杂度。以同样方式，可以研究复杂度对其他参数，如近邻知识、方向侦听等的依赖性（B.3节）。

6.5.2 计算和

216 ~ 217

6.1.5小节表明，一次波动可以计算出所有进程输入的下确界。下确界的计算可用于计算可交换的、可结合的以及幂等的算子，例如，最小值，最大值等（参见推论6.14）。用这种方

法，大量的函数是不可计算的，这些函数包括计算所有输入和，因为和算子不是幂等的。对输入的求和可用于对具有某一性质的进程计数（如果进程具有这种性质，则设置为1，否则为0），本节的结果也可用于其他可交换的和可结合的算子，如求整数的乘积和多集的并。

并不存在利用波动算法的计算和的一般方法。但在特殊情况下，计算和是可能的，例如，在算法是遍历算法时，进程有标识时或者算法包括的生成树可用时。

1. 一般构造的不可能性

利用任何一种任意波动算法，不可能给出计算和的一般构造方法，类似于定理6.12计算下确界所用的构造方法，如下所示。对于一类网络，存在一种波动算法。这类网络包括所有直径为2的无向匿名网络，即相位算法（参数$D=2$）。不存在一种算法，可以计算所有输入的

表6-1 本章的波动算法

算法	编号	拓扑结构	C/D	T	M	Time
			6.2节 一般算法			
环	6.2	环	C	yes	N	N
树	6.3	树	D	no	N	$O(D)$
回波	6.5	任意	C	no	$2\|E\|$	$O(N)$
轮询	6.6	团	C	no	$2N-2$	2
相位	6.7	任意	D	no	$2D\cdot\|E\|$	$2D$
团上相位	6.8	团	D	no	$N(N-1)$	2
Finn	6.9	任意	D	no	$\leq 4\cdot N\cdot\|E\|$	$O(D)$
			6.3节 遍历算法			
顺序轮询	6.10	团	C	yes	$2N-2$	$2N-2$
圆环	6.11	圆环	C	yes	N	N
超立方	6.12	超立方	C	yes	N	N
Tarry	6.13	任意	C	yes	$2\|E\|$	$2\|E\|$
			6.4节 深度优先搜索算法			
经典算法	6.14	任意	C	yes	$2\|E\|$	$2\|E\|$
Awerbuch	6.15	任意	C	no	$4\cdot\|E\|$	$4N-2$
Cidon	6.16/6.17	任意	C	no	$4\cdot\|E\|$	$2N-2$
近邻知识	6.18	任意	C	yes	$2N-2$	$2N-2$

注：相位算法（图6-7）和Finn算法（图6-9）适用于有向网络。

表6-2 集中式算法对消息复杂度的影响

拓扑结构	C/D	复杂度	参考
环	C	N	算法（图6-2）
	D	$O(N\log N)$	算法（图7-7）
任意	C	$2\|E\|$	算法（图6-5）
	D	$O(N\log N+\|E\|)$	7.3节
树	C	$2(N-1)$	算法（图6-5）
	D	$O(N)$	算法（图6-3）

和，并且对于所有直径为2的无向匿名网络都是正确的。这类网络包括两种网络，参见图

6-19所示。假设每个进程有输入为1，图中左边的网络的解为6，右边网络的解为8。利用第9章引入的技术，可以证明，对于任意算法，这两种网络都会输出同样的结果，因此在这两种网络中，它是不正确的。论证的详尽细节作为习题9.7留给读者。

2. 用遍历算法计算和

如果A是遍历算法，所有输入的和可如下计算。进程p有变量j_p，初始化为p的输入。令牌包括一个另外的域s。无论什么时候p转发令牌，p都执行

$$s := s + j_p;\ j_p := 0$$

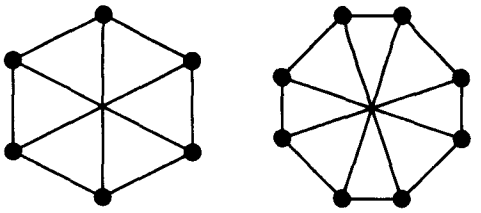


图6-19　直径为2和度为3的两个网络

进而可以证明，在任何时刻，对于每个曾经访问过的进程p，$j_p = 0$，且s等于所有曾经访问过的进程的输入和。因此，当算法终止时，s等于所有输入的和。

3. 用生成树计算和

对于进程p中的每一判定事件d_p，有些波动算法可用于根为p的生成树，经过这棵生成树，消息向着p发送。实际上，波动算法的每次计算包含这样的生成树。然而，也可能是这种情况，进程q发送几条消息，并不知道输出边中的哪一个属于这样的一棵生成树。如果进程知道哪一条输出边是此树中到父节点的边，那么这棵树可用于计算和。每个进程向树中的父节点发送消息，则所有输入的和在子树中。

这个原理可用到树算法（回波算法）中，也可用于团的相位算法中。从p到q所发送的消息，在树算法中很容易包括T_{pq}中的输入和。判定进程通过计算穿过一条边的两个消息中所包含的值，即可得到最终结果。同样对于团相位算法，它将q的输入在每个消息中从q发至p。进程p将所有接收的值和它自己的输入相加，当p判定时，其结果就是正确的答案。在回波算法的计算过程中，构造一棵生成树，来计算输入和。参见习题6.15。

4. 利用标识计算和

假设每个进程有一个惟一标识。如下计算所有输入的和。每个进程通过形成点对(p, j_p)用它的标识为它的输入做出标记。可知，不存在两个进程具有相同的点对。算法保证，当进程判定时，它能识别出每个输入。$S = \{(p, j_p):\ p \in \mathbb{P}\}$表示$p$的所有集合$S_p = \{(p, j_p)\}$的并集，它可以在一次波动中计算出来。由此集合，用局部操作即可计算出所要的结果。

218 ~ 220

解决方法要求每个进程具有惟一标识，这大大增加了位复杂度。波动算法的每条消息包括S的一个子集，如果需要w位来表示一个标识和一个输入，则总共需要$N \cdot w$位，参见习题6.16。

6.5.3　时间复杂度的另一种定义

分布式算法的时间复杂度可以按照几种方式定义。本书中，在讨论时间复杂度时，总是使用6.31中的定义。这里讨论其他一些可能的定义。

1. 基于更严格时间假设的定义

可以用对系统中的事件计时的更严格的假设来估算分布式计算所消耗的时间。

定义6.37　*算法一次计算的最大时间定义为算法的单位时间复杂度。满足以下假设。*

O1 进程在0个时间内，可以执行任意有限个事件。

O2 消息的发送和接收之间的时间恰好为一个时间单位。

与定义6.31比较，可知，假设O1与T1相同。因为T2假设的传输时间至多等于O2假设的时间，这就使人们可能认为单位时间复杂度总是至少等于时间复杂度。因此，人们可能会得出在T2中假设的每次计算至少会与在O2中假设的一样快。因此，具有最大时间的计算在T2 下并不比O2下更差。这种论证的缺点是，按照T2，所允许的传输时间上的变化可能会产生更大的计算，也许包括时间性能较差的计算行为。回忆一下T2的定义，它并不是对要度量的执行进行限制，而是一种时间单位定义。O2实际上将执行集合限制到所有具有相同时间延迟的消息上。我们用回波算法说明这一点。

事实上，反过来也是真的：算法的时间复杂度至少等于那个算法的单位时间复杂度。在假设O1和O2下，所允许的每次计算同样在假设T1和T2下也允许。在后者假设下，计算需要相同时间。因此，按照O1和O2，算法在最坏情况下的行为包括在定义6.31中，是时间复杂度的下界。

定理6.38 *回波算法的单位时间复杂度为*$O(D)$。*回波算法的时间复杂度为*$\Theta(N)$，*甚至在直径为*1*的网络中。* 221

证明。为了分析单位时间复杂度，假设O1和O2。距离初始进程跳数距离为d的进程，在计算开始后，恰好经过d个时间单位接收第一条消息<tok>。在所得的树T中的深度为d。（可以对d用归纳法证明这一点。）设D_T表示树T的深度，那么，$D_T \leqslant D$。并且树T中深度为d的进程，最迟在计算开始后的$(2D_T+1)-d$时间单位向其父节点发送消息<tok>。（可对d用向后归纳证明。）由此可得，初始进程最迟在计算开始后的$2D_T+1$时间单位进行判定。

为了分析时间复杂度，假设T1和T2。与初始进程相距跳数为d的进程，在计算开始后，最迟经过d个时间单位接收第一条消息<tok>。（可以对d用归纳法证明这一点。）假设计算开始后，F个时间单位完成生成树的构造，那么$F \leqslant D$。生成树的深度D_T不必受网络直径所限（在以下的计算中将要证明）。但是因为有N个进程，深度至多为$N-1$。对于树T中深度为d的进程，最迟在计算开始后的$(F+1)+(D_T-d)$时间单位向其父节点发送消息<tok>。（可对d用向后归纳证明。）由此可得，初始进程最迟在计算开始后的$(F+1)+D_T$个时间单位判定，即为$O(N)$。

为证明$\Omega(N)$为时间复杂度的下界，利用时间N，构造了N个进程的团上的一次计算。在团中，决定一棵深度为$N-1$的生成树（实际上是节点的线性链）。假设经过树边向下发送的所有消息<tok>在发送的$1/N$时间单位后被收到，经过非树边的消息<tok>在一个时间单位后收到。按照假设T2，允许这些延迟。在这个计算中，在一个时间单位内构造完全树，但其深度为$N-1$。现在假设经过树边向上发送的所有消息也有一个单位时间的延迟，在这种情况下，计算开始后恰好N个单位时间后发生判定。 □

也许有人会问，在讨论分布式算法的时间复杂度时，应该用哪一个定义。单位时间复杂度具有一些缺点，没有考虑某些计算，尽管这些计算在算法中是可能的。在所忽略的某些计算中，可能有相当一些计算是极其耗时的。定义6.31中所做的假设并不排除个别计算。在定义中，仅仅定义了每次计算的时间度量。时间复杂度仍然受到由计算所决定结果的不利影响（*例如*定理6.38的证明），尽管可能这些计算被认为是极不可能出现的。实际上，在这个计算中，某个消息可能被$N-1$个顺序传输的消息“绕过”。 222

为了协调这两种定义，还可以考虑α-时间复杂度，它是根据每个消息的延迟介于α和1之间（α是$\leqslant 1$的常数）这样一个假设确定的。令人遗憾的是，这种折衷具有这两种算法的缺点。读者可能自己要证明，回波算法的α-时间复杂度为$O(\min(N, D/\alpha))$。

如果假设了消息延迟的概率分布，就能获得最准确的时间复杂度的度量。由此假设，可以计算出算法的期望计算时间。这种方法有两个缺点。第一，算法分析太依赖于系统，因为在每个分布式系统中，消息延迟的分布不同。第二，分析过于复杂，在大多数情况下，不能实现。

2. 基于消息链的定义

可以利用计算的结构性质，而不是理想化的定时假设，来定义分布式计算的时间消耗。设C是一次计算。

定义6.39　*C中的消息链是一个消息序列m_1，m_2，…，m_k，满足对于每个i，$0<i<k$，m_i的接收在因果关系上优先于m_{i+1}的发送。*

分布式算法的链时复杂度定义为算法中任何一次计算的最长消息链长度。

如同定义6.31，这个定义考虑了算法执行的所有可能性，来定义它的时间复杂度，但是为计算赋予不同的度量。考虑这种情况（出现在定理6.38的证明中的计算），k个消息的一条链绕过了某个消息。这个(子)计算的时间复杂度为1。而同样的(子)计算，链时复杂度为k。在消
223 息延迟上界有保证的(就好像在时间复杂度定义中的假设)系统中，时间复杂度是合适的度量。而在“平均”延迟后大多数消息才被传递的系统中，只有一小部分消息可能有较大的延迟，此时，链时复杂度是一种更好的选择。

习题

6.1节

6.1　给出具有同步消息传递系统的一个PIF算法的例子，但该系统不能计算下确界（与定理6.7和6.12比较）。你的例子可能只适合于某种特定的拓扑结构。

6.2　在偏序$(X, \leqslant)$中，称元素b为底，如果对于所有$c \in X$，$b \leqslant c$。

它被用于定理6.11的证明中，偏序$(X, \leqslant)$不包含底。底在哪儿？

你能给出一个算法，用具有底的偏序来计算下确界吗？且该算法不是波动算法。

6.3　给出两个自然数集上的偏序，(1) 其中一个其下确界函数是最大公因子，(2) 另一个其下确界函数是最小公共祖先。

给出全集U的子集集合上的偏序，(1) 其中一个其下确界函数是集合的交，(2) 另一个其下确界函数是集合的并。

6.4　证明下确界定理（定理6.13）。

6.2节

6.5　证明在树网算法（图6-3所示的算法）的每次计算中，只有两个进程判定。

6.6　利用回波算法（图6-5所示的算法），写一个任意网上计算前缀标号模式（4.4.3小节）
224 的算法，利用$2|E|$条消息和$O(N)$个时间单位。

你能给出一个用$O(D)$的时间计算标号模式的算法吗？（D是网络的直径。）

6.7 证明，如果在信道pq中消息丢失，引理6.19中的关系仍成立，但是如果消息可被复制，引理6.19中的关系不成立。如果消息可复制，证明中的那一步是失效的？

6.8 把定理6.12的构造方法应用到相位算法中，得到的算法可以计算所有进程输入（整数）的最大值。你所设计的算法的消息复杂度、时间复杂度和位复杂度是多少？

6.9 假设你想将波动算法用于其中可能出现消息复制的网络中。

（1）对回波算法要做哪些修改？

（2）对Finn算法要做哪些修改？

6.3节

6.10 完全二分图是图$G=(V, E)$，其中$V=V_1 \cup V_2$且$V_1 \cap V_2=\varnothing$，$E=V_1 \times V_2$。

给出一个完全二分网络的$2x$-遍历算法。

6.11 （项目）文献[DRT98]表明，在没有方向侦听的超立方体上进行遍历和广播也是可能的。算法是f-遍历的f值为多少？

6.4节

6.12 给出Tarry算法计算的一个例子，所得的树T不是一棵DFS树。

6.13 对于一个任意连通网络，给出计算深度优先搜索区间标号模式（4.4.2小节）的算法。

它可在$O(N)$时间单位完成吗？可用$O(N)$条消息来完成吗？

6.14 假设具有近邻知识的深度优先搜索算法用于系统中，系统中每个进程不仅知道近邻标识，而且知道所有进程标识集（$\mathbb{P}$）。证明每个进程只需N位消息即可。 225

6.5节

6.15 修改回波算法（图6-5所示的算法），使其能够计算进程输入和。

6.16 在图6-21说明的网络中，假设进程具有惟一标识，每个进程有一个整数输入。在两个网络上，模拟相位算法的一次计算，计算集合$S=\{(p, j_p): p \in \mathbb{P}\}$，以及对输入求和。

6.17 团的相位算法（图6-8所示的算法）的链时复杂度是多少？ 226

第7章 选举算法

本章讨论选举问题，也称找领导人问题。LeLann[LeL77]首先提出了选举问题，参见7.2.1小节。问题是从具有同一状态的进程配置开始，最后达到一种配置状态，其中只有一个进程处于*leader*状态，而其他所有进程处于*lost*状态。

如果要执行集中式算法，且没有一个优先的候选人作为算法的初始进程，就要举行进程的选举。例如，可能存在这样一种情况，初始进程必须在初始时执行或者在系统损毁后执行。因为预先并不知道活动进程集合。它不可能对一个进程指派一次，也不能对所有进程都指派领导人的角色。

现有大量关于选举问题（算法以及更一般的理论）的成果存在。本章的内容按照以下准则选取。

（1）同步系统、匿名进程和容错算法在其他章讨论。本章总是假设，进程和信道是可靠的，系统是完全异步的，以惟一标识区别进程。

（2）比较不同计算模型的效率，选举问题起着“基准问题”（benchmarking problem）的作用。我们只涉及比较中所需的一些结果，在后面的章节（9、12、11）中还会经常讨论到这个问题。

（3）本章集中讨论消息复杂度、有改进的时间复杂度的算法。没有讨论在时间复杂度和消息复杂度之间权衡后所得到的结果。

7.1 引言

选举问题要求，从具有相同状态的进程的配置开始，达到一种配置状态，其中只有一个进程处于领导人状态，而其他所有进程处于失败状态。在计算的最后，处于领导人状态的进程称为领导人，它由算法选出。

定义7.1 *选举算法满足以下性质的。*

（1）*每个进程有相同的局部算法。*

（2）*算法是分散式的，即，进程的任意非空子集都能开始一次计算。*

（3）*在每次计算中，算法达到终止配置。在每一个可达的终止配置中，只有一个进程处于领导人状态，而其他所有进程处于失败状态。*

最后一个性质有时可以弱化，只要求有一个进程处于领导人状态。然而也有这种情况，被选进程得知它已赢得选举，而失败进程还未意识到它们的失败。如果一个算法满足这个给定的弱要求条件，就很容易由领导人开始一个扩散过程，并通知所有进程选举结果。本章的一些算法省略了这个通知。

本章的所有算法中，进程p有变量$state_p$，其可能的值包括*leader*和*lost*。有时假设在p执行算法的任何步之前，$state_p$的值为*sleep*。如果p已经参加计算，但还不知道它是否赢得选举，则设$state_p$的值为*cand*。有些算法还用另外一些状态，如*活动的*（active）、*被动的*（passive）等状态。这些将会在算法中表明。

7.1.1 本章所做假设

我们现在回顾一下本章研究的选举问题。

（1）*系统是完全异步的*　假设进程不能访问公共时钟，消息传输时间可任意长短。

228 由此可得这样一个假设：同步消息传递（即，认为消息的发送和接收是一次转移）几乎不会影响选举问题所获得的结果。读者自己可以确信，本章给出的算法可用于具有同步消息传递的系统，所得结果下界也适用于这种情况。

全局定时假设对选举问题的解决方法具有重要影响。例如，假设进程可观察到真实时间而且消息延迟有限。

（2）*每个进程具有惟一标识其标识初始时对于进程是可知的*　为简单起见，假设进程p的标识就是p，标识可从全序集$\mathcal{P}$中取出，即，标识上的关系$<$可用。w表示一个标识的位数。

在选举问题中，标识的惟一性是非常重要的。不仅可用来对消息定址，也可用于打破进程之间的对称性。但设计选举算法时，可能假定具有最小（或最大）标识的进程赢得选举。那么，问题就变成了用分散式算法找最小标识的问题。在这种情况下，称选举问题为*找极值*（extrema-finding）问题。

尽管本章所讨论的某些算法，最初是为选出最大进程而进行形式化的，但实际上，我们将大多数这样的算法形式化为选取最小进程的问题。我们可以通过将两个标识之间的比较次序颠倒过来，得到最大进程的选举算法。

（3）*本章中与比较算法有关的一些结果*　比较算法就是利用比较作为对标识的惟一操作的算法。通过对算法进行检查，可以看出，本章所有算法都是比较算法。当给出一个下界结果时，我们会详细地阐明它是否与比较算法有关。

可以证明（例如，按Bodlaender [Bod91b]针对环网络的情况），异步网络中，任意算法不会比比较算法有更好的复杂度。与第12章中将要讨论的同步系统情况不同。在这些同步系统中，任意算法会达到比比较算法更好的复杂度。

229 （4）*每条消息可能包含$O(w)$位*　每条消息最多只可包含固定数目的进程标识。这样做的目的是，可以公平地比较不同算法的通信复杂度。

7.1.2 选举和波动

如前所述，进程标识可以用来打破进程之间的对称性。可以这样设计算法，使得具有最小标识的进程就是所选举的进程。按照6.1.5小节的结果，在一次波动中，进程可以计算出最小标识。这表明可以执行计算最小标识的一次波动来举行选举。执行之后，具有这个标识的进程就成为领导人。由于选举算法必须是分散式的，这个原理只能用于分散式的波动算法（参见表6-1）。

1. 基于树算法的选举

如果网络的拓扑结构是树，或者网络的生成树可用，可以使用树算法进行选举（6.2.2节）。在树算法中，要求至少所有叶子节点是算法的初始进程。为使算法进行，以免算法中只有部分进程是初始进程，我们加入*唤醒*（wake-up）阶段。想要开始选举的进程将消息**<wakeup>**扩散到所有进程中。布尔变量ws用于使每个进程至多发送一次消息。变量wr用于对进程接收的消息**<wakeup>**计数。当进程通过每个信道接收到消息**<wakeup>**时，就开始执行图6-3所示

的算法，可以扩充图6-3所示的算法（正如定理6.12中所作的那样），使其进行最小标识的计算。并使得每个进程判定。当进程进行判定时，就知道了领导人的标识。如果该标识与进程标识相等，它就称为领导人，否则就成为失败进程（参见图7-1所示的算法）。

```
var ws_p     : boolean                          init false ;
    wr_p     : integer                          init 0 ;
    rec_p[q] : boolean for each q ∈ Neigh_p     init false ;
    v_p      : P                                init p ;
    state_p  : (sleep, leader, lost)            init sleep ;

begin if p is initiator then
        begin ws_p := true ;
              forall q ∈ Neigh_p do send ⟨wakeup⟩ to q
        end ;
      while wr_p < #Neigh_p do
        begin receive ⟨wakeup⟩ ; wr_p := wr_p + 1 ;
              if not ws_p then
                begin ws_p := true ;
                      forall q ∈ Neigh_p do send ⟨wakeup⟩ to q
                end
        end ;
      (* Now start the tree algorithm *)
      while #{q : ¬rec_p[q]} > 1 do
        begin receive ⟨tok, r⟩ from q ; rec_p[q] := true ;
              v_p := min(v_p, r)
        end ;
      send ⟨tok, v_p⟩ to q_0 with ¬rec_p[q_0] ;
      receive ⟨tok, r⟩ from q_0 ;
      v_p := min(v_p, r) ; (* decide with answer v_p *)
      if v_p = p then state_p := leader else state_p := lost ;
      forall q ∈ Neigh_p, q ≠ q_0 do send ⟨tok, v_p⟩ to q
end
```

图7-1 基于树的选举算法

定理7.2 基于树的选举(图7-1所示的算法)的消息复杂度为$O(N)$，时间复杂度为$O(D)$。

证明。至少有一个进程开始执行算法，所有进程向各自所有近邻发送消息<**wakeup**>，每个进程在接到来自每个近邻的消息<**wakeup**>之后，开始执行树网选举算法。所有进程以同一个值v终止该算法。即，进程的最小标识。具有这个标识的(惟一)进程结束时处于领导人状态。其他进程结束时处于失败状态。

经过每个信道发送两条消息<**wakeup**>和两条消息<**tok**，r>，这使得消息复杂度为$4N-4$。第一个进程开始执行算法后的D个时间单位内，每个进程发送了消息<**wakeup**>，因此，在$D+1$个时间单位内，每个进程开始波动。显而易见，第一次判定发生在波动开始后的D个时间单位内，最后一次判定发生在第一次判定后的D个时间单位内，这使得总时间为$3D+1$。更细致的算法分析揭示算法总是在$2D$个时间单位内终止，这留给读者去论证，参见习题7.2。

□

如果消息在信道中重新排列序列(即信道不是fifo)，进程可能从近邻接收消息<**tok**，r>，

230 ~ 231 而此时它还没有接到这个近邻进程的消息<**wakeup**>。在这种情况下，可将消息<**tok**，r>暂时存储，或者将它作为随后到达的消息<**tok**，r>处理。

通过两个修正可以减少消息数。第一，可以排列，使得非初始进程并不向它第一次接到消息<**wakeup**>的进程发送消息<**wakeup**>。第二，可将叶节点发送的消息<**wakeup**>与该叶节点发送的消息<**tok**，r>组合。做了这些修正，算法中所要求的消息数可减少到$3N-4+k$，其中k是非叶初始进程个数[Tel91b，p.139]。

2. 基于相位的选举算法

相位算法可用来解决选举问题，方法是通过在一次波动中计算最小标识，见定理6.12。

定理7.3　*在任意网上用相位算法（图6-7所示的算法）解选举问题，消息复杂度为* $O(D \cdot |E|)$，*利用* $O(D)$ *个时间单位。*

Peleg算法[Pel90]基于相位算法。利用$O(D \cdot |E|)$条消息和$O(D)$个时间单位。但不需要知识D，因为算法中包括直径的在线计算。

3. 基于Finn算法的选举

Finn算法（图6-9所示的算法）并不要求预先知道网络直径。Finn算法所用的$O(N \cdot |E|)$条消息要比本章所做的假设要长，因此，Finn算法的每条消息总计有$O(N)$条消息，这使得消息复杂度达到$O(N^2 \cdot |E|)$。

7.2 环网

本节考虑单向环上的选举算法。LeLann[LeL77]首先提出了环网上的选举问题，同时给出了消息复杂度为$O(N^2)$的解决方法。Chang和Roberts[CR79]改进了这个算法。给出了算法最坏情况下的复杂度$O(N^2)$。但算法的平均复杂度仅为$O(N \log N)$。在7.2.1讨论了LeLann和Chang-Roberts的方法。直到1980年，Hirschberg和Sinclair[HS80]给出了最坏情况下复杂度为$O(N \log N)$的选举算法。与早先的一些算法不同，Hirschberg-Sinclair算法要求信道是双向的。暂且推测，$\Omega(N^2)$的消息复杂度是单向环的下界。但是Petersen[Pet82]和Dolev、Klawe及Rodeh[DKR82]各自独立地提出了单向环上复杂度为$O(N \log N)$的解决方法。这些方法在7.2.2小节讨论。

通过匹配大约相同时间的下界，可以对算法进行补充。Bodlaender[Bod88]证明了双向环上最坏情况下消息复杂度的一个下界，约为$0.34\ N \log N$。Pachl、Korach和Rotem[PKR84]证明了平均情况下消息复杂度的一个下界$\Omega(N \log N)$，对于双向和单向环都成立。在7.2.3节将讨论下界的结果。

7.2.1 LeLann和Chang-Roberts算法

在LeLann[LeL77]算法中，每个初始进程计算所有初始进程的标识表。计算之后，具有最小标识的进程被选中。通过环网，每个初始进程发送一个包括它自己标识的令牌，这个令牌被所有进程转发。假设信道是fifo的，某个初始进程必须在其他初始进程接到令牌之前，产生它的令牌。（当进程接到令牌时，它以后再不会初始算法。）当初始进程p接到它自己的返回令牌，所有初始进程的令牌都已经过p，p当选，当且仅当p是初始进程中的最小的。参见图7-2所示的算法。

定理7.4　*LeLann选举算法(图7-2所示的算法)解决环网的选举问题，消息复杂度为*

$O(N^2)$，利用$O(N)$个时间单位。

证明。因为在环上令牌的次序保持不变（由fifo假设），初始进程q在q接到消息<**tok**，p>之前，发送出消息<**tok**，q>。初始进程p在p接到返回消息<**tok**，p>之前接到消息<**tok**，q>。由此可得，每个初始进程p以$List_p$结束，$List_p$等于所有初始进程的集合。具有最小标识的初始进程是当选的惟一进程。至多有N个不同令牌，每个进行N步，这使得消息复杂度达到$O(N^2)$。在第一个初始进程发送出令牌后最迟$N-1$个时间单位，每个初始进程也已经发送完成，且在那个令牌产生后的N个时间单位内，每个初始进程接到返回的令牌。这蕴含着，算法在$2N-1$个时间单位内终止。 □

```
var List_p      : set of P     init {p} ;
    state_p ;

begin if p is initiator then
        begin state_p := cand ; send ⟨tok, p⟩ to Next_p ; receive ⟨tok, q⟩ ;
              while q ≠ p do
                    begin List_p := List_p ∪ {q} ;
                          send ⟨tok, q⟩ to Next_p ; receive ⟨tok, q⟩
                    end ;
              if p = min(List_p) then state_p := leader
                                 else state_p := lost
        end
      else while true do
                begin receive ⟨tok, q⟩ ; send ⟨tok, q⟩ to Next_p ;
                      if state_p = sleep then state_p := lost
                end
end
```

图7-2 Lelann选举算法

非初始进程都进入失败状态，但是仍然等待更多消息<**tok**，r>。如果领导人沿着环发出一个特殊令牌，宣告选举已经结束，那么等待才能中止。

Chang-Roberts算法[CR79]改进了LeLann算法，从环中删除了所有那些看起来将失去选举的进程的令牌。即，如果$q > p$，则初始进程p从环中删除令牌<**tok**，q>。当初始进程p接到令牌q，且其标识满足$q < p$时，初始进程p变成失败进程。当进程接到标识为p的令牌时，就变成领导人。参见图7-3所示的算法。

定理7.5 Chang-Roberts算法（图7-3所示的算法）解决了环网的选举问题，最坏情况下的消息复杂度为$\Theta(N^2)$，利用$O(N)$个时间单位。

证明。设p_0是具有最小标识的初始进程。每个进程或者是非初始进程，或者是标识大于p_0的初始进程，所有进程转发由p_0所发送的令牌<**tok**，p_0>。因此，p_0接到返回的令牌，并当选。

非初始进程没有当选，而是都进入失败状态，最迟在p_0的令牌被转发时。$p > p_0$的初始进程p也没有当选，p_0并不转发令牌<**tok**，p>，因此，进程p永远接收不到自己的令牌。这样的初始进程p，最迟在<**tok**，p_0>被转发时进入失败状态。这就证明了算法解决了选举问题。

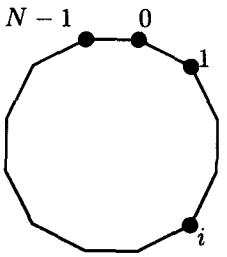
至多使用N个不同的令牌，每个令牌至多被转发N个跳数，这就证明了消息复杂度的界限为$O(N^2)$。为证明至少利用$\Omega(N^2)$条消息，考虑初始配置，其中标识沿环上的递增次序排列（参考图7-4）。每个进程都是一个初始进程。进程0可以从环上删除每个进程的令牌。因此进

232 ~ 234

程i的令牌被转发$N-i$个跳数。这使得消息传递数达到$\sum_{i=0}^{N-1}(N-i)=1/2N(N+1)$。 □

```
var state_p ;
begin if p is initiator then
        begin state_p := cand ; send ⟨tok, p⟩ to Next_p ;
              while state_p ≠ leader do
                    begin receive ⟨tok, q⟩ ;
                          if q = p then state_p := leader
                          else if q < p then
                                begin if state_p = cand then state_p := lost ;
                                      send ⟨tok, q⟩ to Next_p
                                end
                    end
        end
      else while true do
                begin receive ⟨tok, q⟩ ; send ⟨tok, q⟩ to Next_p ;
                      if state_p = sleep then state_p := lost
                end
end
(* 只有领导人终止这个程序。它将消息分散到所有进程中，告知它们领导人的标识，并终止。*)
```

图7-3　Chang-Roberts选举算法

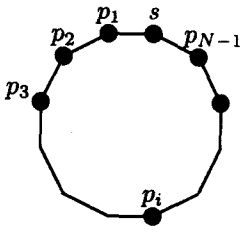


顺时针方向传递消息

235

图7-4　Chang-Roberts算法的最坏情况

当考虑时间复杂度或最坏情况消息复杂度时，Chang-Roberts算法不再是对LeLann算法的改进。而如果考虑平均情况，则是一种改进。平均是指标识在环上所有可能的排列。



顺时针方向传递消息

图7-5　环上标识排列

定理7.6　*当所有进程都是初始进程时，Chang-Roberts算法平均情况下只要求*$O(N \log N)$*次消息传递。*

证明。(这个证明基于Friedemann Mattern的提示。)

假设所有进程都是初始进程，我们计算令牌在N个不同标识组成的所有环形排列上，令牌传递的平均数。考虑N个标识的固定集合，设s是最小标识。有$(N-1)!$个标识的不同环形排列。在给定的环形排列中，设p_i是在s之前出现i步的标识，参见图7-5。

要计算所有排列上所传递的令牌总数，首先计算在所有排列上传递令牌<**tok**，p_i>的总次数，然后对i求和。在每个排列中，传递令牌<**tok**，s> N次，因此共传递$N(N-1)!$次。至多传递令牌<**tok**，p_i> i次。因为如果它到达s，就会被从环上删除。设$A_{i,k}$是令牌<**tok**，p_i>只被传递k次的环形排列数。则令牌<**tok**，p_i>被传递的总次数为 $\sum_{k=1}^{i}(k \cdot A_{i,k})$ 。

如果p_i是标识p_1到p_i的最小标识，则令牌<**tok**，p_i>只被传递i次，这是排列数为$(1/i) \cdot (N-1)!$次的情况。因此，

$$A_{i,i} = \frac{1}{i}(N-1)!$$

如果进程p_i后有$k-1$个进程的标识大于p_i，则令牌<**tok**，p_i>至少被传递k次 $(k \leqslant i)$。当然， 236
p_i是k个标识p_{i-k+1}，…，p_i中最小标识的排列数为所有排列的$1/k$，即，$(1/k) \cdot (N-1)$!。现在，对于$k < i$，如果令牌<**tok**，p_i>恰恰传递k次，且不多于k次，即，至少传递k次但不是至少传递$k+1$次，则令牌恰好传递k次。因此这种情况发生时，排列数为

$$\frac{1}{k}(N-1)! - \frac{1}{k+1}(N-1)!$$

即，

$$A_{i,k} = \frac{1}{k(k+1)}(N-1)! \quad (\text{对于} k < i)$$

在所有排列中，传递令牌<**tok**，p_i>的总次数为

$$\sum_{k=1}^{i-1} k\left(\frac{1}{k(k+1)}(N-1)!\right) + i\frac{1}{i}(N-1)!$$

它等于 $\left(\sum_{k=1}^{i} 1/k\right)(N-1)!$。称和 $\left(\sum_{k=1}^{i} 1/k\right)$ 为第i个**调和数** (harmonic number)，用$\mathbf{H}_i$表示。以下等式的证明，留做习题7.3。

$$\sum_{i=1}^{m} \mathbf{H}_i = (m+1)\mathbf{H}_m - m$$

接下来对i求和计算所传递的令牌数，得到所有排列中令牌传递的总数（除了那些<**tok**，s>的传递)。总数为

$$\sum_{i=1}^{N-1} [\mathbf{H}_i (N-1)!] = (N\mathbf{H}_{N-1} - (N-1))(N-1)!$$

加上传递令牌<**tok**，s>的次数$N \cdot (N-1)!$，可得令牌传递的总数为

$$(N\mathbf{H}_{N-1} + 1)(N-1)! = (N\mathbf{H}_N) \cdot (N-1)!$$

因为这是对于$(N-1)!$个不同排列，因此平均传递次数显然为$N\mathbf{H}_N$，约为$0.69N \log N$（参见习题7.4)。

□

7.2.2 Peterson/Dolev-Klawe-Rodeh算法

Chang-Roberts算法平均情况下的消息复杂度为O($N\log N$)。但不是最坏情况。
237 Franklin[Fra82]给出了最坏情况下的消息复杂度为O($N\log N$)的算法。在算法中，要求信道是双向的。Peterson[Pet82]和Dolev、Klawe和Rodeh[DKR82]分别独立地研究出了类似算法，解决了问题，单向环上最坏情况下消息复杂度为O($N\log N$)。算法要求信道是fifo的。

算法首先计算最小标识，并使它为每一进程所知。具有该标识的那个进程变成领导人，其他进程都被击败。如果把算法的执行看作是标识的执行，而不是进程的执行，算法就会更容易理解。初始时，每个标识是*活动的*，在每一轮以后，有一些标识变成*被动的*，如稍后所示。在一轮中，一个*活动*标识和与它相邻的两个活动标识（一个在顺时针方向，一个在逆时针方向）进行比较，如果它是局部最小，则它在那一轮中生存下来，否则，就变成*被动的*。因为所有标识不同，与局部最小相邻的标识不是局部最小，这蕴含着，一轮中至少有一半的标识不会生存下来。因此，至多log N轮后，只剩下一个*活动*标识。它就是获胜者。

在双向网络中，可用直接方式详细描述这个原理，正像Franklin算法[Fra82]中所做的那样。在有向环中，消息只能按照顺时针方向发送，这使得它在顺时针方向难以获得下一个活动标识，参见图7-6。标识q必须与r和p进行比较，标识r可被发送到q，但是标识p必须沿信道的反方向遍历，才能到达q。为了能与r和p进行比较，标识q被发送（在环的方向）到持有标识p的进程中，r不仅被转发到持有q的进程中，而且进一步被转发到持有p的进程中。如果q是在那轮开始的惟一*活动*进程，则q在遍历中遇到的第一个标识等于q（即，在这种情况下，$p = q$）。当出现这种情况时，该标识就是选举中的获胜者。

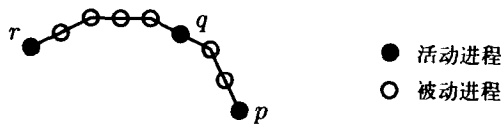


图7-6　进程p获得当前q和r的标识

图7-7所示的算法给出了单向环上的进程算法。称进程p在一轮中是*活动*的，如果在该轮
238 的一开始，它持有*活动*标识ci_p。否则，p是*被动*的，并简单地传递所有它所接收的消息。*活动*进程把它的当前标识发送到下一个*活动*进程，并利用消息**<one, >**获得前一个*活动*进程的当前标识。将接收的标识存储在变量acn_p中。如果标识在这轮中幸存下来，它将是下一轮中p的当前标识。为决定是否标识acn_p在这轮中幸存下来，就要将它与ci_p和消息**<two, >**中所得到的活动标识比较。进程p发送消息**<two,** acn_p**>**，使得在下一个活动进程中做出决策成为可能。当$acn_p = ci_p$时，出现例外。在这种情况下，该标识是惟一剩下的活动标识，并在消息**<smal,** acn_p**>**中，向所有进程发布。

定理7.7　*图7-7所示的算法解决单向环上的选举问题，消息复杂度为*O($N\log N$)。

证明。如果进程第i次执行主循环，则称它是在第i轮中。轮并不是全局同步的；在环的不同部分，一个进程可能在另一个进程的前几轮。但是，在每一轮中，由于每个进程只能发送和接收两条消息，且信道是fifo的，因此消息的接收和发送总是在同一轮中。在第一轮中，所有初始进程都是*活动*的，每个*活动*进程持有不同的“当前标识”。

```
var ci_p   : P   init p ;        (* Current identity of p *)
    acn_p  : P   init udef ;     (* Id of anticlockwise active neighbor *)
    win_p  : P   init udef ;     (* Id of winner *)
    state_p: (active, passive, leader, lost) init active ;

begin if p is initiator then state_p := active else state_p := passive ;
      while win_p = udef do
        begin if state_p = active then
                begin send ⟨one, ci_p⟩ ; receive ⟨one, q⟩ ; acn_p := q ;
                      if acn_p = ci_p then (* acn_p is the minimum *)
                        begin send ⟨smal, acn_p⟩ ; win_p := acn_p ;
                              receive ⟨smal, q⟩
                        end
                      else (* acn_p is current id of neighbor *)
                        begin send ⟨two, acn_p⟩ ; receive ⟨two, q⟩ ;
                              if acn_p < ci_p and acn_p < q
                                then ci_p := acn_p
                                else state_p := passive
                        end
                end
              else (* state_p = passive *)
                begin receive ⟨one, q⟩ ; send ⟨one, q⟩ ;
                      receive m ; send m ;
                      (* m is either ⟨two, q⟩ or ⟨smal, q⟩ *)
                      if m is a ⟨smal, q⟩ message then win_p := q
                end
        end ;
      if p = win_p then state_p := leader else state_p := lost
end
```

图7-7 Peterson/Dolev-Klawe-Rodeh算法

声明7.8 *如果在第i轮开始，有$k > 1$个活动进程，每个进程持有不同的ci，那么，在那一轮中，至少有一个进程幸存，至多有$k/2$个进程幸存。并且在那一轮执行完后，所有活动进程的当前标识不同，且这些标识中包括最小标识。*

*证明。通过交换由被动进程进行传递的消息<**one**，q>。每个活动进程得到它逆时针方向的第一个活动近邻的当前标识，这个标识在任何情况下都不会和自身的标识相同。因此，每个活动进程通过消息<**two**，q>的交换继续那一轮的执行，通过交换，每个活动进程还得到它逆时针方向第二个活动近邻的当前标识。每个活动进程现在持有不同的acn值，这蕴含，那一轮中的所有幸存者，在那一轮结束时具有不同标识。至少在那一轮开始时具有最小标识的进程会幸存下来，因此至少有一个幸存者。与局部最小相邻的标识不是局部最小标识，这蕴含着幸存者的数目至多为$k/2$。* □

声明7.8表明，将会存在一轮，轮数$\leqslant \lfloor \log N \rfloor + 1$，开始时只有一个活动标识，即初始进程的最小标识。

声明7.9 *如果某一轮开始时，只有一个活动进程p，且当前标识为ci_p。在那一轮以后，算法终止，且对于每个进程q，$win_q = ci_p$。*

*证明。所有进程传递的p的消息<**one**，ci_p>，最终由p接收。进程p获得$acn_p = ci_p$，且沿着环发送消息<**smal**，acn_p>，这引起每个进程q在$win_q = acn_p$时，退出主循环。* □

239 ~ 240 每个进程中算法终止，所有进程对领导人标识（在变量win_q中）取得一致，这个进程处于领导人状态，所有其他进程处于失败状态。

至多有$\lfloor \log N \rfloor + 1$轮，每一轮只被$2N$个消息交换，这就证明消息复杂度的界限为$2N \log N + O(N)$。这就证明了定理7.7。 □

Dolev等人将该算法改进到$1.5\ N \log N$。此后Peterson发现了只用$1.44\ N \log N$条消息的算法，这个结果再次被Dolev等人改进到$1.356\ N \log N$。环上选举问题的上限$1.356\ N \log N$，保持了10年以上的最好结果，在1993年被Higham和Przytycka[HP93]改进到$1.271\ N \log N$。

7.2.3 一个下界

在本小节中，证明了单向环上选举复杂度的一个下界。因为可以在分散式波动算法的一次执行中举行选举，因而，作为推论，可得环上分散式波动算法复杂度的下界。

在下列假设下，Pachl、Korach和Rotem[PKR84]给出了上述结果。

（1）证明了计算最小标识算法的下界。如果领导人可用，那么在N条消息中，可以计算出最小标识。如果至少一个进程知道最小标识，那么具有这个标识的进程在N条消息中当选。因此，选举问题和计算最小标识问题在复杂度上至多相差N条消息。

（2）环是单向的。

（3）进程不知道环的大小。

（4）假设信道满足fifo。这种假设并没有弱化结果。因为非fifo的算法也不比fifo算法有更好的复杂度。

（5）假设所有进程都是初始进程。这种假设并没有弱化结果。因为对于每个分散式算法，它描述了一种可能情形。

（6）假设算法是消息驱动的。即，算法开始执行，并发送一些消息之后，进程仅当接到一条消息时，才进一步发送消息。当考虑异步系统时，一般算法不会比消息驱动的算法有更
241 好的复杂度。事实上，如果A是一个异步算法，那么消息驱动的算法B可构造如下。开始执行并接到消息之后，B发送A所允许发送的最大消息序列而没有接收消息。只有在那时，才能接收下一条消息。算法B不仅是消息驱动的，而且B的每次计算是A的一次可能计算（可能在相当悲观的消息传输延迟分布之下）。

后三条假设消除了系统的非确定性。在这些假设之下，从给定初始配置开始的每次计算，包含相同的事件集合。

在本小节中，$s = (s_1, \ldots, s_N)$、t等代表不同进程标识序列。D表示所有这些序列的集合，即

$$D = \{ (s_1, \ldots, s_k) : s_i \in \mathcal{P} \text{且} i \neq j \Rightarrow s_i \neq s_j \}$$

$len(s)$表示序列s的长度。st表示序列s和t的连接。s的*循环移位*（cyclic shift）是一个序列$s's''$，其中$s = s''s'$，具有形式$s_i, \ldots, s_N, s_1, \ldots, s_{i-1}$。$CS(s)$表示$s$的循环移位集合。当然，$|CS(s)| = len(s)$。

称环以序列$(s_1, \ldots, s_N)$为*标号*，如果进程标识s_1到s_N以此顺序出现在环上(规模为N的环)。标号为s的环也称为s-环。如果t是s的循环移位，那么t-环和s-环相同。

对于算法中发送的每一条消息，将会有一个进程标识序列与此相关，称为消息的*轨迹*(trace)。如果m是进程p接到消息之前所发送的一条消息，那么，m的轨迹就是(p)。如果m是

进程p接到轨迹为$s=(s_1, \ldots, s_k)$的消息之后所发送的消息，那么，m的轨迹就是$(s_1, \ldots, s_k, p)$。具有轨迹s的消息称为s-消息。可从算法发送所有消息的轨迹集合的性质导出算法的下界。

设E是D的子集，集合E是穷举的(exhaustive)，如果

(1) E是闭前缀的，即$tu \in E \Rightarrow t \in E$；且

(2) E循环覆盖D，即，$\forall s \in D$: $CS(s) \cap E \neq \varnothing$。

以下将要表明，算法中的所有轨迹集合是穷举的，为了从这个事实导出算法复杂度的下界，需要定义集合E的两种度量。如果序列t是$r \in CS(s)$的前缀，则在s-环中，序列t出现为标识的连续序列。$M(s, E)$是s-环中以这种方式出现的E的序列的个数，$M_k(s, E)$表示长为k的这些串的个数:

$$M(s, E) = |\{t \in E: t\text{是某些}r \in CS(s)\text{的一个前缀}\}|$$

且

$$M_k(s, E) = |\{t \in E: t\text{是某些}r \in CS(s)\text{的一个前缀且}len(t)=k\}|$$

设A是计算最小标识的算法，定义E_A为序列s的集合，满足当算法A在s-环上执行时，发送s-消息。

引理7.10 *如果s是t和u的子串，当算法A在t-环上执行时，发送s-消息，那么当算法A在u-环上执行时，也发送s-消息。*

证明。进程s_k发送的s-消息，其中$s=(s_1, \ldots, s_k)$，只在因果关系上依赖于进程s_1到进程s_k。它们在u-环上的初始状态与在t-环上的初始状态相同（这里，我们记住环大小是未知的），因此进行消息发送的事件的集合在u-环上也是可应用的。 □

引理7.11 *E_A是穷举集。*

证明。为证明E_A是闭前缀的，观察可见，如果A在s-环上执行时，发送一条s-消息，那么，对于s的每个前缀t，A首先在s-环上发送一条t-消息。由引理7.10，A在t-环上发送一条t-消息，因此$t \in E_A$。

为证明E_A循环覆盖D，考虑s-环上A的一次计算。至少有一个进程对最小标识判定。这蕴含着（类似于定理6.11所用的方法），这个进程已经接到轨迹长度为$len(s)$的消息。这个轨迹是s的循环移位，且在E中。 □

引理7.12 *在s-环上的一次计算中，算法A至少发送$M(s, E_A)$条消息。*

证明。设$t \in E_A$是s的循环移位r的前缀。由E_A的定义，A在t-环上的一次计算中，发送一个t-消息，因此，在r-环上也发送一个t-消息，这个r环等于s-环。因此，对于每个t

$$\{t \in E: t\text{是某些}r \in CS(s)\text{的一个前缀}\}$$

在s-环上的一次计算中，至少发送一条t-消息。这就证明了那次计算中的消息数至少是$M(s, E)$。 □

242 ~ 243

对于进程标识的有限集合I，设$Per(I)$表示I的所有排列的集合。$ave_A(I)$表示A所用的消息的平均数，其中算法A在所有以I中的标识作标记的环上，$wor_A(I)$表示最坏情况下的消息数。前述引理表明，如果I有N个元素，那么

(1) $ave_A(I) > \frac{1}{N!}\sum_{s \in Per(I)} M(s, E_A)$，且

(2) $wor_A(I) \geq \max_{s \in Per(I)} M(s, E_A)$。

通过分析任意穷举集合证明下界。

定理7.13 *单向寻找最小标识的算法平均情况复杂度至少为$N \cdot \mathbf{H}_N$。*

证明。对所有标以集合I的初始配置求平均数，可得

$$\begin{aligned} ave_A(I) &\geq \frac{1}{N!} \sum_{s \in Per(I)} M(s, E_A) \\ &= \frac{1}{N!} \sum_{s \in Per(I)} \sum_{k=1}^{N} M_k(s, E_A) \\ &= \frac{1}{N!} \sum_{k=1}^{N} \sum_{s \in Per(I)} M_k(s, E_A) \end{aligned}$$

我们固定k，对于每个$s \in Per(I)$，有N个长为k的s的循环移位前缀。$Per(I)$中有$N!$个排列，产生$N \cdot N!$个这类前缀，可以分成$N \cdot N!/k$个组，每组包含一个序列的k个循环移位。因为，E_A循环地覆盖D，E_A与每个小组相交。因此，

$$\sum_{s \in Per(I)} M_k(s, E_A) \geq \frac{N \cdot N!}{k}$$

这蕴含着

$$ave_A(I) \geq \frac{1}{N!} \sum_{k=1}^{N} \frac{N \cdot N!}{k} = N \cdot \mathbf{H}_N。$$ □

这个结果表明，如果考虑平均情况，Chang-Roberts算法是最优的。最坏情况下的复杂度至少等于平均情况，这蕴含着最坏情况下可达的最好的复杂度介于$N \cdot \mathbf{H}_N \approx 0.69N \log N$和$\approx 1.356N \log N$之间。

244 本节的证明结果，主要依赖于环是单向的和环大小未知这两个假设。在假设双向(bidirectional)环和环大小未知的条件下，Bodlaender[Bod88]证明了选举算法的平均情况复杂度下界为$1/2\ N \cdot \mathbf{H}_N$。为了从双向环上消除非确定性，考虑这样一种计算，其中每个进程同时开始和每条消息具有相同传输延迟。对于已知环大小的情况，Bodlaender[Bod91a]证明了单向环上的一个下界$1/2\ N \cdot \log N$和双向环上的下界$(1/4-\varepsilon)N \cdot \mathbf{H}_N$(两种都为平均情况)。

总之，由上可得，环上选举算法的复杂度对几乎所有能够做的假设不敏感。无论环大小已知还是未知，双向还是单向，也无论是考虑最坏情况复杂度还是平均情况复杂度，其复杂度在所有情况下都为$\Theta(N \log N)$。关键是环是异步的。对于全局时间可用的网络，消息复杂度较低，参见第12章。

当在分散式波动算法的一次计算中，可以选出领导人时，选举的下界也蕴含了波动算法的下界。

推论7.14 *任何环网上的分散式波动算法至少交换$\Omega(N \log N)$条消息，对于平均情况和最坏情况均如此。*

7.3 任意网

现在研究任意网的选举问题。未知的网络拓扑结构和没有近邻知识可用。以下将给出消息复杂度的一个下界$\Omega(N \log N + |E|)$。证明过程结合了定理6.6的思想和前面小节中的一些结

果。在7.3.1小节中，给出了最坏情况下，具有低时间复杂度、高消息复杂度的一个简单算法。7.3.2节提出了最坏情况下的一个最优算法。

定理7.15 任意网络上基于比较的选举算法至少具有$\Omega(|E| + N \log N)$的消息复杂度（最坏情况和平均情况）。

证明。$\Omega(N \log N)$项是一个下界，因为任意网络包括环，而在环上此下界成立。为了证明$|E|$个消息也是一个下界，即使在所有计算中最好情况。假设选举算法A在网络G上有一 245
次计算C，在计算中需要交换的消息数小于$|E|$，参见图7-8。在C中不用的一条边上插入节点，用这两节点间的一条边连接G的两个拷贝来构造G'。网络中两部分标识与G中的标识具有相同的相对次序。在G'的两部分中可以同时模拟计算C，产生一次计算，在计算中两个进程当选。 □

推论7.16 在没有近邻知识的任意网络上，分散式波动算法消息复杂度至少为$\Omega(|E| + N \log N)$。

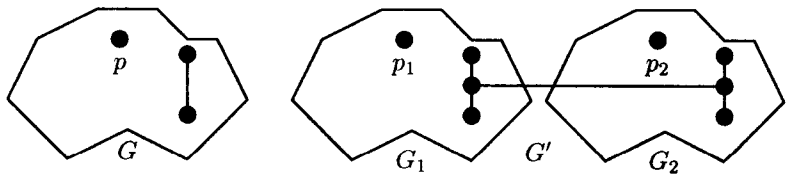


图7-8 有两个领导人的计算

7.3.1 废止和快速算法

将称为*废止*（extinction）的构造过程应用于任意集中式波动算法，就得到领导人选举算法。在所得的选举算法中，每个初始进程分别开始波动，由进程p开始执行的波动消息必须加上标号p，以便将它们与来自不同波动的消息区别出来。算法保证，不管开始多少次波动，只有一个波动进行判定，即，具有最小初始进程的波动进行判定。在发生判定之前，所有其他波动将会中止。

对于波动算法A，选举算法$Ex(A)$如下。每个进程每次至多在一个波动中是活动的；这个波动是它的*当前活动波动*（currently active wave），用caw表示，其初始值为$udef$。选举中的初始进程作用开始执行一次波动并caw设为自己的标识。如果由q开始执行的某些波动的消息到达p，p处理这些消息如下。如果$q > caw_p$，那么简单忽略消息，有效地引起q的波动失败。如果$q = caw_p$，那么完全按照波动算法处理消息。如果$q < caw_p$，或者caw_p为$udef$，那么通过重设变量到初始值，且设$caw_p := q$，将P加入到q的波动执行。当q开始的波动执行一个判定事 246
件时（在大多数波动算法中，这种判定总是发生在q中），q将当选。如果波动算法满足判定进程不必是初始进程，判定进程经过引理6.3中定义的生成树通知初始进程。这至多需要花费$N-1$条消息。在下个定理中，我们将忽略这些。

定理7.17 如果A是集中式波动算法，每次波动利用M条消息，那么算法$Ex(A)$至多利用NM条消息选出领导人。

证明。设p_0是最小初始进程，由p_0初始的波动立刻就被接收该波动消息的进程加入。因为没有更小标识的波动，会异常终止p_0波动的执行，因此每个进程都会完成这次波动。因此p_0的

波动运行到完成，并发生判定，最终p_0成为领导人。

如果p是非初始进程，标识为p的波动未曾被开始执行，因此p不会变成领导人。如果$p \neq p_0$是初始进程，标识为p的波动将开始，但是p_0中的发送事件（对于这次波动）将在这个波动的判定之前发生，或者这个波动中的判定在p_0中发生（引理6.4）。由于p_0从不执行标识为p的波动中的发送事件，或者内部事件，因此这样的判定并不发生，p不会被选中。

至多开始N次波动，每个波动至多利用M条消息，使得总复杂度达到NM。 □

估算时间复杂度$Ex(A)$是一个更细致的问题。在许多情况下，它的时间复杂度和A的时间复杂度具有相同的数量级。然而令人遗憾的是，可能具有最小标识的初始进程很迟才开始它的波动。在一般情况下，其时间复杂度为$O(N \cdot t)$（其中t是波动算法的时间复杂度）。因为在初始进程p开始波动后的t个时间单位内，p的波动判定，或者开始另一次波动。

如果将废止方法应用于环网算法，就得到Chang-Roberts算法。参见习题7.9。图7-9所示的算法给出了由回波算法所得的选举算法。为了简化描述，假设对于所有$q \in \mathcal{P}$，$udef > q$。由图7-9所示算法得知，一旦接收带有$r < caw_p$的消息<**tok**，r>，由于稍早对caw_p的赋值，就会执行条件为$r = caw_p$的**if**语句。当进程p当选时（已经接到来自每个近邻的消息<**tok**，p>），p将消息<**ldr**，p>分散到所有进程中，通知它们p是领导人，使得它们终止算法的执行。

```
var caw_p      : P        init udef ;  (* Currently active wave *)
    rec_p      : integer  init 0 ;     (* Number of ⟨tok, caw_p⟩ received *)
    father_p   : P        init udef ;  (* Father in wave caw_p *)
    lrec_p     : integer  init 0 ;     (* Number of ⟨ldr, .⟩ received *)
    win_p      : P        init udef ;  (* Identity of leader *)

begin if p is initiator then
        begin caw_p := p ;
              forall q ∈ Neigh_p do send ⟨tok, p⟩ to q
        end ;
      while lrec_p < #Neigh_p do
        begin receive msg from q ;
              if msg = ⟨ldr, r⟩ then
                begin if lrec_p = 0 then
                            forall q ∈ Neigh_p do send ⟨ldr, r⟩ to q ;
                      lrec_p := lrec_p + 1 ; win_p := r
                end
              else (* a ⟨tok, r⟩ message *)
                begin if r < caw_p then (* Reinitialize algorithm *)
                            begin caw_p := r ; rec_p := 0 ; father_p := q ;
                                  forall s ∈ Neigh_p, s ≠ q
                                        do send ⟨tok, r⟩ to s
                            end ;
                      if r = caw_p then
                        begin rec_p := rec_p + 1 ;
                              if rec_p = #Neigh_p then
                                if caw_p = p
                                  then forall s ∈ Neigh_p
                                                do send ⟨ldr, p⟩ to s
                                  else send ⟨tok, caw_p⟩ to father_p
```

图7-9　废止方法应用于回波算法

```
                    end
                (* If r > caw_p, the message is ignored *)
            end
        end ;
    if win_p = p then state_p := leader else state_p := lost
end
```

图7-9 （续）

7.3.2 Gallager-Humblet-Spira算法

任意网络上的选举问题与用分散式算法计算生成树的问题密切相关。设C_E是选举问题的消息复杂度，C_T是计算生成树的消息复杂度。定理7.2蕴含$C_E \leqslant C_T + O(N)$。如果领导人可用，利用带回波算法的$2|E|$条消息就可以计算生成树。这蕴含着，$C_T \leqslant C_E + 2|E|$。C_E（定理7.15）的下界蕴含着这两个问题复杂度具有相同的数量级，即它们至少要求$\Omega(N \log N + |E|)$条消息。 247 ~ 248

本小节给出计算（最小）生成树的Gallager-Humblet-Spira（GHS）算法，消息复杂度为$2|E| + 5N \log N$。这表明C_E和C_T为$\Theta(N \log N + |E|)$。这个算法发表在文献[GHS83]上。可以很容易地修改这个算法（本小节最后简要说明）使得在计算的过程中选出领导人，上述论证中所指出的单独选举没有必要。

GHS算法依赖如下假设。

（1）每条边e有惟一的权值$\omega(e)$，这里假设$\omega(e)$是实数，但是整数也有可能作为边的权值。

如果惟一的边权值预先不可用，可将由这条边所连接节点的标识作为权值赋给每条边。两者中较小者优先。由于计算边的权值要求节点知道近邻标识，因此在算法开始执行阶段，这需要另外$2|E|$条消息。

（2）所有节点尽管初始时处于睡眠态，但它们在算法开始执行时苏醒。有些节点自然地被唤醒（如果由于这些节点中出现的情况变化，触发了算法执行），其他节点可能仍在睡眠态时，就接到算法的消息。在后一种情况下，接收节点首先开始执行局部过程，然后处理消息。

最小生成树 设$G = (V, E)$是带权图，其中$\omega(e)$表示边e的权值。G的生成树T的权值等于T中所包含的$N-1$条边的权值之和。如果没有一棵树其权值比T的权值小，则称T为最小生成树（minimal spanning tree），或MST，（有时称为最小权值生成树，minimal-weight spanning tree）。本小节中，假设每条边具有惟一权值，即，不同边具有不同权值。这是一个众所周知的事实，在这种情况下，有惟一的一棵最小生成树。

命题7.18 如果所有边权值不同，则有惟一一棵MST。 249

证明。反证法。假设T_1和T_2（其中$T_1 \neq T_2$）都是最小生成树。设e是出现在其中一棵树中权值最小的边，但并不在两棵树中都出现。由于$T_1 \neq T_2$，因此这样的边存在。不失一般性，假设e在T_1中，不在T_2中。图$T_2 \cup \{e\}$包含回路，因为T_1不含回路，因此这条回路中至少有一条边e'不在T_1中。e的选择表明，$\omega(e) < \omega(e')$，但是树$T_2 \cup \{e\} \backslash \{e'\}$具有比$T_2$更小的权值。这与$T_2$是MST矛盾。 □

命题7.18使得分布式构造最小生成树变得相当容易。因为不需从合法解的集合做出选择（分布地）。正相反，每个节点只需局部地选择属于任一最小生成树的边，加入到全局惟一最

小生成树MST的构造中。

计算最小生成树的所有算法基于片断（fragment）表示，它是MST的一棵子树。称边e是片断F的出边（outgoing edge），如果e的一个端点在F中，另一个端点不在F中。算法从由一个节点构成的片断开始，逐步增大片断直到完成MST的构造。

命题7.19 *如果F是片断，e是F的最小带权出边，那么$F \cup \{e\}$是片断。*

*证明。*假设$F \cup \{e\}$不是MST的一部分，那么e与MST中某些边形成回路，且在这个回路中，MST中的一条边f是F的一条出边。由e的选择，$\omega(e) < \omega(f)$，但接着从MST中删除f，并插入e，则得到一棵比MST具有更小权值的树，这与MST是最小生成树矛盾。 □

著名的计算MST的顺序算法是Prim算法和Kruskal算法。Prim算法[CLR90，24.2节]从单一片断开始，在每步中，用当前片断中具有最小权值的出边逐渐扩大片断。Kruskal算法[CLR90，24.2节]从单节点片断集合开始，在每步中，用每个片断中最小带权出边逐渐扩大片
250 断，并对产生的片断进行归并。因为kruskal算法可以独立地处理片断，因此它更适合于在分布式算法中实现。

7.3.3 GHS算法的全局描述

从片断的观点来看，我们首先描述算法是如何以全局方式运转的。然后描述每个节点必须执行的局部算法，通过该局部算法获得片断的全局操作。

GHS算法的计算按以下几步进行：

1）保持片断集合，满足所有片断的并集包括所有节点。

2）初始时，集合中包含的每个节点是由一个节点组成的片断。

3）通过片断中的节点找出片断的最小带权出边。

4）当已知片断的最小带权出边时，通过增加出边，将片断之间连接过来。

5）当只剩一个片断时，算法终止。

为了有效地实现这些步骤，需要引入一些表示和机制。

（1）*片断名。*为了决定最小加权输出边，必须能够确定是否一条边是出边或者是通向同一片断中的节点。因此，每个片断将有一个名字，且该名字要为那个片断中的进程所知。进程通过比较它们的片断名，就可断定一条边是内部边还是出边。

（2）*大小片断的组合。*当组合两个片断时，至少有一个片断中的进程片断名改变，这就要求两个片断中至少有一个片断，对其中每个节点进行更新。为了保持更新效率，组合策略采取把*两个片断中的较小者*组合进*两个片断中的较大者*中，并沿用较大片断名作为新片断名。

（3）*片断级。*稍微思考可以看出，对两个片断中的哪一个是较大者的判定，并不是基于两个片断中的节点数。这就需要在大小子片断的每个进程中，更新片断大小，因此破坏了所
251 需要的性质，即，更新只需发生在较小片断中的性质。因此我们为每个片断赋予一个级，0表示初始的只含一个节点的片断。当片断F_1和更高一级的片断F_2组合后，新片断$F_1 \cup F_2$的级与F_2的级相同。新片断和F_2同名。因此无需对F_2中的节点更新。也可能是两个同级片断进行组合，在这种情况下，新片断有一个新名字，它的级别比当前被组合的片断高一级。片断的新名字为组合两个片断的边上的权值。这条边称为新片断的*核心边*（core edge）。核心边所连接的两个节点称为*核心节点*（core node）。

引理7.20 如果满足这些组合条件，则进程改变它的片断名或者片断级的次数至多为$N \log N$。

证明。进程级永远不会减小，仅当进程级增加时，进程改变片断名。级L的片断中至少包含2^L个进程。因此最大级为$\log N$。这蕴含着每个进程级至多增加$\log N$次。因此，片断名和片断级改变的总数为$N \log N$。 □

组合策略概述 $F = (FN, L)$表示名为FN和级为L的片断F。e_F表示F的最小加权出边。

规则A 如果e_F通向$L < L'$的片断$F' = (FN', L')$，F组合成F'，新片断名为FN'，片断级为L'，这些新值被发送到F中的所有进程中。

规则B 如果e_F通向$L = L'$的片断$F' = (FN', L')$，且$e_{F'} = e_F$，这两个片断组合成新片断，片断级为$L+1$，片断名为$\omega(e_F)$。这些新值被发送到F和F'中的所有进程中。

规则C 在所有其他情况下（如，$L > L'$或者$L = L'$且$e_{F'} \neq e_F$），片断F必须等待直到规则A或者规则B被应用。

7.3.4 GHS算法的详细描述

1. 节点和链路状态

节点p保持变量，如图7-10所示的算法所述。其中包括每个信道pq的状态$stach_p[q]$。该状态是分支的（branch），如果得知它是MST中的一条边；该状态是废弃的（reject），如果它不是MST中的一条边，该状态是基本的（basic），如果这条边仍未用。决定片断中最小带权出边 252
的通信，在片断的分支边进行。对于片断中的进程p，$father_p$就是通向片断中核心边的边。如果节点p当前正在查找片断中的最小带权输出边，节点p的状态$state_p$为查找状态；否则处于已找到状态。如同图7-10、图7-11和图7-12所示的算法，给出了该算法。有时局部条件满足以后，才能处理消息。在这种情况下，假设消息被存储，以后再找回并处理，就好像在那个时刻已经接收到一样。如果进程接收一条消息，而它仍在睡眠状态，那么在处理消息之前，算法在那个节点开始执行（通过执行行为(1)）。

```
var state_p                          : (sleep, find, found) ;
    stach_p[q]                       : (basic, branch, reject)   for each q ∈ Neigh_p ;
    name_p, bestwt_p                 : real ;
    level_p                          : integer ;
    testch_p, bestch_p, father_p     : Neigh_p ;
    rec_p                            : integer ;

(1) As the first action of each process, the algorithm must be initialized:
    begin let pq be the channel of p with smallest weight ;
          stach_p[q] := branch ; level_p := 0 ;
          state_p := found ; rec_p := 0 ;
          send ⟨connect, 0⟩ to q
    end

(2) Upon receipt of ⟨connect, L⟩ from q:
    begin if L < level_p then (* Combine with Rule A *)
              begin stach_p[q] := branch ;
```

图7-10 Gallager-Humblet-Spira算法(PART 1)

```
                send ⟨ initiate, level_p, name_p, state_p ⟩ to q
          end
      else if stach_p[q] = basic
              then (* Rule C *) process the message later
              else (* Rule B *) send ⟨ initiate, level_p + 1, ω(pq), find ⟩ to q
end

(3) Upon receipt of ⟨ initiate, L, F, S ⟩ from q:
begin level_p := L ; name_p := F ; state_p := S ; father_p := q ;
      bestch_p := udef ; bestwt_p := ∞ ;
      forall r ∈ Neigh_p : stach_p[r] = branch ∧ r ≠ q do
            send ⟨ initiate, L, F, S ⟩ to r ;
      if state_p = find then begin rec_p := 0 ; test end
end
```

图7-10　(续)

```
(4) procedure test:
    begin if ∃q ∈ Neigh_p : stach_p[q] = basic then
              begin testch_p := q with stach_p[q] = basic and ω(pq) minimal ;
                    send ⟨ test, level_p, name_p ⟩ to testch_p
              end
          else begin testch_p := udef ; report end
    end

(5) Upon receipt of ⟨ test, L, F ⟩ from q:
    begin if L > level_p then     (* Answer must wait! *)
              process the message later
          else if F = name_p then (* internal edge *)
                  begin if stach_p[q] = basic then stach_p[q] := reject ;
                        if q ≠ testch_p
                          then send ⟨ reject ⟩ to q
                          else test
                  end
               else send ⟨ accept ⟩ to q
    end

(6) Upon receipt of ⟨ accept ⟩ from q:
    begin testch_p := udef ;
          if ω(pq) < bestwt_p
            then begin bestwt_p := ω(pq) ; bestch_p := q end ;
          report
    end

(7) Upon receipt of ⟨ reject ⟩ from q:
    begin if stach_p[q] = basic then stach_p[q] := reject ;
          test
    end
```

图7-11　GALLAGER-HUMBLET-SPIRA算法（PART 2）

2. 找最小带权出边

片断中的节点相互协作以找出片断中的最小带权出边。如果找到，则通过这条边发送消息<**connect**，L>，L是片断级。如果片断只由一个节点组成，就像是这个节点初始化后的情形，

所求边即为这个节点的最小权值近邻边，参见行为（1）。经过这条边发送消息<**connect**，0>。

```
(8) procedure report:
    begin if rec_p = #{q : stach_p[q] = branch ∧ q ≠ father_p}
                and testch_p = udef then
              begin state_p := found ; send ⟨report, bestwt_p⟩ to father_p end
    end

(9) Upon receipt of ⟨report, ω⟩ from q:
    begin if q ≠ father_p
          then (* reply for initiate message *)
               begin if ω < bestwt_p then
                          begin bestwt_p := ω ; bestch_p := q end ;
                     rec_p := rec_p + 1 ; report
               end
          else (* pq is the core edge *)
               if state_p = find
                 then process this message later
                 else if ω > bestwt_p
                           then changeroot
                           else if ω = bestwt_p = ∞ then stop
    end

(10) procedure changeroot:
     begin if stach_p[bestch_p] = branch
           then send ⟨changeroot⟩ to bestch_p
           else begin send ⟨connect, level_p⟩ to bestch_p ;
                      stach_p[bestch_p] := branch
                end
     end

(11) Upon receipt of ⟨changeroot⟩:
     begin changeroot end
```

图7-12 GALLAGER-HUMBLET-SPIRA算法（PART 3）

接下来考虑通过组合两个片断形成新片断的情形，通过边$e = pq$连接。如果两个被组合的片断具有相同级别L，则p和q都将经过e发送消息<**connect**，L>，且当e的状态为分支状态时，则反过来接收消息<**connect**，L>，参见行为（2）。边pq成为片断的核心边。p和q交换消息<**initiate**，$L+1$，N，S>，给出片断的新级别和片断名。片断名为$\omega(pq)$，状态*find*引起每个进程开始查找最小带权出边，参见行为（3）。消息<**initiate**，$L+1$，N，S>被扩散到新片断中的每一个节点。如果p的片断级小于q的片断级，p将经过e发送消息<**connect**，L>，反过来接收q所发送的消息<**initiate**，L'，N，S>，参见行为（2）。在这种情况下，L'和N是当前的片断级和q的名字，该边上q这一边的节点名和级别并不改变。在这条边上p这一边，初始消息被扩散到所有节点中（参见行为（3）），使得每个进程更新它的片断名和级别。如果q当前正在查找最小带权出边（$S = find$），通过调用*test*，p的片断中的进程加入查找过程。

253 ~ 255

片断中的每个进程通过它的边（如果边存在，参见行为（4）、（5）、（6）和（7））查找是否存在直通片断外的边，如果有直通片断外的边，选择一条权值最小的边。利用消息<**report**，ω>向每棵子树报告这条最小带权出边，参见行为（8）。节点p利用变量rec_p，对它所接到的消息<**report**，ω>进行计数，在查找开始时，它的值设为0（参见行为（3））。随着每次接到消息，

它的值增加（参见行为(9)）。当进程从其子节点已接到这样的消息时，并完成局部查找出边后，便向其父节点发送消息<**report**，ω>。

每个进程沿着核心边的方向发送消息<**report**，ω>，两个核心节点的消息在这条边上相遇；都接收来自*father*的消息，参见行为(9)。在核心节点处理其他进程的消息之前，每个核心节点等待发送自己的消息<**report**，ω>。当核心节点的两条消息<**report**，ω>相遇时，核心节点知道最小带权出边的权值。如果在该点上根本没有出边报告（两个消息都报告值∞），则算法终止。

如果报告一条出边，从报告最好边的核心节点开始，沿着每个节点中*bestch*指针找到最好的边。消息<**connect**，L>必须通过这条边发送，片断中所有*father*指针必须指向这个方向，可以通过发送消息<**changeroot**>完成。在最小带权出边所定位的那一边的核心节点发送消息<**changeroot**>，经过通向最小带权出边的树完成发送，参见行为(10)和行为(11)。当消息<**changeroot**>到达依附于最小带权出边的节点时，这个节点通过最小带权出边发送消息<**connect**，L>。

3. 测试边

为了找到最小带权出边，节点p按照权值增加的次序逐个检查它的基本边，参见行为(4)。当没有边剩余时(所有边都是废弃边或者分支边)，参见行为(4)，或者找到一条出边时，参见行为(6)，局部查找边结束。因为p按照次序检查这些边，因此如果p识别出一条出边，那么它必然是来自p的最小带权出边。

为检查边pq，p向q发送消息<**test**，$level_p$，$name_p$>，并等待应答，这种应答可以是<**reject**>，<**accept**>或者<**test**，L，F>消息。如同在测试消息中，如果进程q发现p的片断名与
256 q的片断名一致，进程q发送消息<**reject**>（参见行为(5)）。在这种情况下，节点q也丢弃这条边。p一旦接到消息<**reject**>，p丢弃边pq，并继续进行局部查找，参见行为(7)。如果q正好利用边pq发送<**test**，L，F>消息，可以省略<**reject**>消息。在这种情况下，q的消息<**test**，L，F>作为对p的消息的应答，参见行为(5)。如果q的片断名与p的片断名不同，就发送<**accept**>消息。p一旦接到这个消息，就终止对出边的局部查找，并选择边pq作为局部查找的最好结果，参见行为(6)。

如果$L > level_p$，p则延期处理消息<**test**，L，F>。原因是p和q可能属于同一个片断，但是消息<**initiate**，L，F，S>还未到达进程p。节点p可能用一条消息<**accept**>，错误应答q。

4. 组合片断

确定片断F = ($name$，$level$) 的最小带权出边后，经过这条边发送消息<**connect**，$level$>，由属于片断F' = ($name'$，$level'$) 中的节点接收这条消息。发送消息<**connect**，$level$>的进程为p，接收这条消息的进程是q。节点q较早发送消息<**accept**>给p，作为对消息<**test**，$level$，$name$>的应答，因为这时在p的片断中已经终止查找最好出边。在回答测试消息（参见行为(5)）之前引入的等待，蕴含着$level' > level$。

按照早先讨论的组合规则，在两种情况下，用消息<**initiate**，L，F，S>回答消息<**connect**，$level$>。

情形A：如果$level' > level$，p的片断被吸收，通过消息<**initiate**，$level'$，$name'$，S>通知片断中的节点新的片断名和片断级，它将这条消息扩散到片断F的所有节点中。在片断F'的生成树中，整个被吸收的片断F变成q的一棵子树。如果q当前在片断F'中查找最好出边，那么F中的所有进程必须参加查找。这就是为什么q在它的消息<**initiate**，$level'$，$name'$，S>中包含它

的状态（*find*或者*found*）的原因。

情形B：如果两个片断具有相同的级别，则片断F'的最好出边也是pq，形成的新片断高1级，片断名为边pq上的权值，参见行为(2)。如果两个片断级相等，并且经过分支边接收连接消息；就会出现这种情况。观察可见，如果一条连接消息通过一条边发送，那么这条边的状态就变成分支状态。

如果这两种情况都不出现，片断F必须等待，直到q发送消息<**connect**，L>消息，或者q的片断级足够增加到使情形A是可应用的。 257

5. 正确性和复杂度

从算法的详细描述可得，片断发送消息<**connect**，L>所通过的那条边，就是片断的最小带权出边。结合命题7.19可得，尽管算法导致的等待存在，但如果每个片断确实发送这样一条消息，并加入其他片断中，则MST的计算是正确的。最复杂的消息包括一个边权值、一个级别（可达到$\log N$）以及表示消息类型和节点状态的固定数目的位。

定理7.21 Gallager-Humblet-Spira计算最小生成树的算法（图7-10、图7-11、图7-12所示的算法），至多利用$5N\log N+2|E|$条消息。

证明。死锁本质上是由于节点或者片断等待另一节点或者片断中的条件出现所引起的。核心边上引入的对消息<**report**，ω>的等待不会导致死锁，这是因为每个核心节点最终都会接到来自所有子节点的报告（除非整个片断都在等待另一片断），然后处理消息。

考虑片断$F_1=(level_1, name_1)$中的消息到达片断$F_2=(level_2, name_2)$中一个节点的情形。如果$level_1 > level_2$，并且通过同一条边，片断F_2还没有发送消息<**connect**，$level_2$>，则消息<**connect**，$level_1$>必须等待，参见行为(2)。如果$level_1 > level_2$，则消息<**test**，$level_1$，$name_1$>必须等待。参见行为(5)。在F_1等待F_2的所有情况中，以下情形之一成立。

(1) $level_1 > level_2$;

(2) $level_1 = level_2 \wedge \omega(e_{F_1}) > \omega(e_{F_2})$；

(3) $level_1 = level_2 \wedge \omega(e_{F_1}) = \omega(e_{F_2})$；并且$F_2$仍在查找最小带权出边。（因为$e_{F_1}$是$F_2$的出边，所以$\omega(e_{F_2}) > \omega(e_{F_1})$的情形不可能出现。）

因此，不会出现死锁回路。

每条边至多被废弃一次，这需要两条消息，其限定了废弃消息数加上导致废弃的测试消息数共为$2|E|$。在任何一级上，节点至多接收一条初始消息、一条接收消息，发送至多一条报告消息、一条改变根的消息，或一条连接消息、以及一条不会导致废弃的测试消息。在0级，没有接收消息，也不发送报告消息或者测试消息。在最高级上，每个节点只发送报告消息和 258
接收一次初始消息。因此消息的总数至多为$5N\log N+2|E|$。 □

7.3.5 GHS算法的讨论和变化

Gallager-Humblet-Spira算法是最复杂的波动算法之一，只需要局部知识，就能得到最优消息复杂度。很容易扩展该算法，选出领导人，而这些只需增加两条消息。算法在两个节点上终止，即，最后片断（生成整个网络）的核心节点上。不用执行**stop**，核心节点交换标识，其中较小者成为领导人。

已有大量变化和相关算法发表。如果某些节点开始算法很晚，GHS算法就会要求$\Omega(N^2)$的时间。如果使用另外的唤醒过程（至多再需要$2|E|$条消息），算法的时间复杂度为$5N\log N$，参见习题7.11。Awerbuch[Awe87]表明，算法的时间复杂度可以改进到$O(N)$，同时保持算法

的消息复杂度最优，即$O(|E| + N \log N)$。

Afek等人[ALSY90]改进了算法，所计算的生成森林具有很好性质。即，每棵树的直径和树的个数为$O(\sqrt{N})$。他们的算法可以分布地计算网络的簇，如引理4.47所表明的那样。并且计算生成树和簇的中心。

也许有人会问，任意生成树的构造是否比最小生成树的构造更有效。定理7.15给出了构造任意生成树的复杂度下界$\Omega(N\log N + |E|)$。在网络是稀疏的条件下，Johansen等人[JJN+87] 给出了计算任意生成树的算法，该算法利用$3N \log N + 2 \cdot |E|+O(N)$条消息，因此通过一个常数因子改进了GHS算法。Bar-Ilan和Zernik[BIZ89]提出了计算随机生成树的算法，其中以等概率选择每棵可能的生成树。算法随机化，利用的期望消息数介于$O(N \log N +|E|)$和$O(N^3)$之间，这取决于网络的拓扑结构。

在任意网上，构造任意生成树和最小生成树具有相同的难度。但在团上却不是这样。
259 Korach、Moran和Zaks[KMZ85]已经证明，在带权的团上，构造最小生成树需要交换$\Omega(N^2)$条消息。结果表明，拓扑结构知识并没有降低在定理7.15给出的下界下寻找MST的复杂度。团网上任意生成树的构造可用$O(N \log N)$条消息。下一节中将要讨论。也可参见[KMZ84]。

7.4 Korach-Kutten-Moran算法

至今已得到许多选举问题的算法。不仅有环网、任意网，还有其他一些特殊的拓扑结构，如团网等。在几种情况下，最好的已知算法，其消息复杂度为$O(N \log N)$。在某些情况下，这个结果还可与下界$\Omega(N \log N)$相匹配。Korach、Kutten and Moran[KKM90]表明，在选举问题和网络的遍历之间有着紧密的关系。主要结果是给出一类网络的遍历算法，为该类网络构造一个有效的选举算法。

这种构造为许多网络产生$O(N \log N)$的选举算法。因为$O(x)$（线性）的遍历是可能的最好结果，用这项技术不可能获得更好的算法。相反，许多具有方向侦听的网络允许有更好的选举算法，在第11章中将会看到。同时，Korach-Kutten-Moran算法的时间复杂度与消息复杂度相同。在某些情况下，具有同样消息复杂度的其他算法具有更低的时间复杂度。构造具有一般性，在解决更高一级的问题时，如选举问题，遍历之间所形成的关系作为“模块”调用。

7.4.1 模块构造

Korach-Kutten-Moran算法利用废止构造（7.3.1小节）和Peterson/Dolev-Klawe-Rodeh算法（7.2.2小节）的思想。与废止构造的类似之处在于，选举的初始进程通过以它们的标识所标记的令牌，开始遍历网络。如果遍历完成（判定），那次遍历的初始进程当选。算法表明，只在一次遍历中进行选举。本小节描述的算法假设信道满足fifo，但是通过在每个令牌中和每个进程中维持更多的信息，可以修改算法使其适合于非fifo的情况，参见[KKM90]。

260 为了处理多于一个初始进程的情形，算法按照级（level）操作，与Peterson/Dolev-Klawe-Rodeh算法中的轮相对应。如果至少开始两次遍历，那么令牌将要到达已经被其他令牌访问过的进程。如果出现这种情形，通过所到达的令牌将中止这次遍历。算法的目标现在变成将两个令牌放在一个进程中，并在进程中杀死它们，开始新的遍历。与Peterson/Dolev等人的算法相比，在一轮中，每两个标识中至多有一个幸存，并继续到下一轮次。在Korach-Kutten-

Moran算法中用级的概念代替轮的概念；仅当两个令牌具有相同的级时，它们将引起新的遍历。新产生的令牌高出一级。如果令牌遇到高一级的令牌，或者到达已被高一级令牌访问的节点，到达的令牌则简单地被杀死，而不影响更高一级的令牌。

图7-13所示的算法描述了这个算法。为了将同级令牌放在一个进程中，每个令牌处于三种模式之一：合并（annexing）、追赶（chasing）或等待（waiting）。用（q，l）表示令牌，其中q表示令牌的初始进程，l表示它的级别。变量lev_p表示进程p的级，变量cat_p表示p所转发的最后合并令牌的初始进程（p的当前活动遍历）。如果没有令牌等待p，变量$wait_p$值为$udef$；如果令牌（q，lev_p）等待p，变量$wait_p$值为q。变量$last_p$用于处于追赶模式的令牌：如果追赶令牌不在它之后发送，p将级为lev_p的合并令牌转发给它的近邻；在这种情况下，$last_p = udef$。算法将遍历算法作用函数$trav$调用。函数$trav$返回到令牌必须被转发至的近邻，或者如果遍历终止，则返回判定。

```
var lev_p          : integer     init −1 ;
    cat_p, wait_p  : P           init udef ;
    last_p         : Neigh_p     init udef ;

begin if p is initiator then
        begin lev_p := lev_p + 1 ; last_p := trav(p, lev_p) ;
              cat_p := p ; send ⟨annex, p, lev_p⟩ to last_p
        end ;
      while ... (* Termination condition, see text *) do
        begin receive token (q, l) ;
              if token is annexing then t := A else t := C ;
              if l > lev_p then (* Case I *)
                begin lev_p := l ; cat_p := q ;
                      wait_p := udef ; last_p := trav(q, l) ;
                      send ⟨annex, q, l⟩ to last_p
                end
              else if l = lev_p and wait_p ≠ udef then (* Case II *)
                begin wait_p := udef ; lev_p := lev_p + 1 ;
                      last_p := trav(p, lev_p) ; cat_p := p ;
                      send ⟨annex, p, lev_p⟩ to last_p
                end
              else if l = lev_p and last_p = udef then (* Case III *)
                wait_p := q
              else if l = lev_p and t = A and q = cat_p then (* Case IV *)
                begin last_p := trav(q, l) ;
                      if last_p = decide
                        then p announces itself leader
                        else send ⟨annex, q, l⟩ to last_p
                end
              else if l = lev_p and ((t = A and
                   q > cat_p) or t = C) then (* Case V *)
                begin send ⟨chase, q, l⟩ to last_p ; last_p := udef end
              else if l = lev_p then (* Case VI *)
                wait_p := q
        end
end
```

图7-13 Korach-Kutten-Moran算法

在合并模式中令牌（q，l）开始执行，在这种模式中，该令牌服从遍历算法（如同图7-13所示的算法中的Case IV），直到出现下列情形之一。

（1）遍历算法终止：在这种情况下，q当选为领导人（参见图7-13所示的算法中的Case IV）。

（2）令牌到达级为$lev_p > l$的节点p：在这种情况下，令牌被杀死。（这种情况蕴含在图7-13中所示的算法，算法中的条件要求，$l > lev_p$或者$l = lev_p$。）

（3）令牌到达一个节点，其中级为l的令牌正在等待：在这种情况下，两个令牌被杀死。从那个节点开始新的遍历（参见图7-13所示的算法中Case II）。

261 ~ 262

（4）令牌到达级为l的节点，而且该节点最近已被标识为$cat_p > q$的令牌访问过（参见Case VI），或者被追赶令牌访问过（参见Case III）：令牌在那个节点变成等待状态。

（5）令牌到达级为l的节点，而且该节点最近已被标识为$cat_p < q$的合并令牌访问过：在这种情况下，令牌变成追赶状态，并用和前一令牌相同的信道发送（参见Case V）。

每个节点经过最近刚发送过令牌的信道，转发追赶令牌（q，l），直到出现以下情形之一。

（1）令牌到达级为$lev_p > l$的进程：在这种情况下，令牌被杀死。

（2）令牌到达一个进程，它有一个级为l的等待令牌：这两个令牌被删除，通过这个进程开始新的遍历（参见Case II）。

（3）令牌到达级为l的进程，其中最近传递的令牌处于追赶状态：令牌变成等待状态（参见Case III）。

等待令牌驻留在进程中，直到出现以下情形之一。

（1）高一级的令牌到达同一个进程：等待令牌被杀死（参见Case I）。

（2）相同级的令牌到达：两个令牌被删除，开始高一级的遍历（参见Case II）。

在图7-13所示的算法中，忽略了遍历算法中所用的变量和令牌信息。观察可见，如果p接到比lev_p更高级别的令牌，这是一个p不为初始进程的合并令牌。如果遍历终止在p中，p成为领导人，并把消息扩散到所有进程中，引起它们终止。

1. 正确性和复杂度

为了证明Korach-Kutten-Moran算法的正确性，要表明每一级中所产生的令牌数递减，直到在某一级，只有一个令牌产生，其中初始者当选。

引理7.22　*如果在级l所产生的令牌数$k > 1$，那么在级$l+1$至少产生一个令牌，至多产生$k/2$个令牌。*

*证明。*在级$l+1$至多产生$k/2$个令牌，这是因为，每产生一个令牌，同时将级l的两个令牌杀死。

假设存在级l，满足在级l产生$k > 1$个令牌，但在$l+1$级不产生令牌。设q是在级l产生的具有最大标识的进程令牌。令牌（q，l）没有完成遍历，因为它将被已转发级l另一个令牌的进程p接收。当这种情形首次发生时，（q，l）变成追赶模式，或者如果p已经转发追赶令牌，（q，l）变成等待状态。不管是哪一种情形，级l都存在追赶令牌。

263

设（r，l）是且具有最小标识的令牌，其后发送追赶令牌。令牌（r，l）自身并不是追赶令牌，因为令牌仅仅追赶具有最小标识的令牌。当它首次到达进程p'，并且该进程已经转发了级l的另一令牌，这时可以假定（r，l）变成等待状态。设p''是（r，l）路径上所接收的最后一个进程，在它转发（r，l）之后，合并令牌变成追赶r的令牌。这个追赶令牌或者遇见p'中的（r，l），或者放弃追赶，如果在令牌到达p'之前，发现一个等待令牌。在两种情况下，产生级

为$l+1$的一个令牌。这是一个矛盾。 □

定理7.23 Korach-Kutten-Moran算法（图7-13所示的算法）是选举算法。

证明。假设至少有一个进程开始执行算法。由前述引理，在每一级中所产生的令牌数递减，则存在级为l，其中只产生一个令牌，即，(q, l)。没有级为$l'< l$的令牌来完成遍历，因此，这些令牌中没有一个会使得进程当选。在级l的惟一令牌只会遇见级小于l的进程，或者$cat = q$（如果到达已经访问过的一个进程），在这两种情况下，惟一令牌被转发。因此，令牌的遍历完成，q当选。 □

称函数f为凸的（convex），如果$f(a) + f(b) \leqslant f(a+b)$。在分析算法时，假设利用$f(x)$-遍历算法（参见6.3节），且$f$为凸函数。

定理7.24 如果利用$f(x)$-遍历算法，且f为凸函数，那么KKM选举算法利用至多（1+ log k）$[f(N) + N]$条消息，假设开始时有k个进程执行算法。

证明。如果有k个进程开始执行算法。在级l至多产生$2^{-l}k$个令牌，这蕴含着，最高一级至多为$\lfloor \log k \rfloor$。

在每一级上，每个进程至多发送一个标识的合并令牌，如果在某些级l上，有一些标识为$p_1, \ldots, p_j$的令牌，并且有N_i个进程已经转发了合并令牌(p_i, l)，那么，由此可得，$\sum_{i=1}^{j} N_i \leqslant N$。因为遍历算法是$f(x)$-遍历算法，至多发送$f(N_i)$次合并令牌$(p_i, l)$，这使得在第$l$级携带合并令牌的消息总数至多达到$\sum_{i=1}^{j} f(N_i)$，因为$f$是凸函数，所以消息总数至多为$f(N)$。每个进程每级至多发送一次追赶令牌，这使得每级追赶令牌数至多为N。 264

因此，至多有（1+ log k）个不同级，每级至多发送$f(N) + N$条消息，结果得证。 □

2. Attiya的构造

Attiya从遍历算法构造了另一种选举算法[Att87]。在所构造的算法中，一旦标识为q的令牌到达一个已被另一进程p的令牌访问过的进程r，它的遍历不被中止。合并令牌不是在r处等待，而是发出“搜寻进程”令牌，来追赶p的令牌，如果成功地击败了p，则返回到r。如果搜寻进程返回，就不必要开始新的遍历，而是，q中令牌的当前遍历继续进行，因此潜在地节省了消息复杂度。为了使搜寻进程返回到进程r处，必须假设网络是双向的。如果利用$f(x)$遍历算法，所得选举算法的消息复杂度近似为$3 \cdot \sum_{i=1}^{N} f(N/i)$。

7.4.2 KKM算法的应用

如果存在一类网络的$f(x)$-遍历算法，称这类网络为$f(x)$-可遍历的。因为许多网络是$O(x)$-可遍历的，构造给出了消息复杂度为$O(N \log N)$的选举算法，这些是已知的最好结果。在某些情况下，当存在方向侦听时，可以构造更好的算法（参见第11章）。

1. 环上的选举

环是x-可遍历的，因此，KKM算法在环上选举领导人利用$2N \log N$条消息。由定理7.13可知，这可能是最好的结果。

2. 团上的选举

团是$2x$-可遍历的，在没有方向侦听的情况下，按照定理7.24，KKM算法利用$3N \log N$条消息在团上选举领导人。对算法的仔细分析，揭示出算法实际上的消息复杂度为$2N \log N + O(N)$。至多利用三次追赶消息就可以追上每个令牌。因此一次计算中追赶消息的总数为

$3\cdot\sum_{i=0}^{\log N+1}2^{-i}N=O(N)$。到目前为止，不存在比复杂度为$2N\log N+O(N)$更好的团上选举
265 算法。Korach、Moran and Zaks[KMZ84]证明了它的下界为$\Omega(N\log N)$。

这些结果仅适合于没有方向侦听的团。如果有方向侦听，则存在团上的线性算法。

3. 圆环上的选举

具有方向侦听的圆环网络是x-可遍历的（参见图6.11所示的算法），因此KKM算法在圆环上利用$2N\log N$条消息选举领导人。如果没有方向侦听，可以利用Tarry算法进行遍历，并且在圆环上该算法是线性操作的。Peterson[Pet85]已经给出了网格和圆环上的选举算法，消息复杂度为$O(N)$，并且不要求对边进行标号。

习题

7.1节

7.1　如果把进程变成领导人的事件看作是判定事件，证明任意网上基于比较的选举算法是波动算法。

7.2　证明算法7-1的时间复杂度为$2D$。

7.2节

7.3　证明7.2.1小节中所用的等式：$\sum_{i=1}^{m}\mathbf{H}_i=(m+1)\mathbf{H}_m-m$。

7.4　证明$\ln(N+1)<\mathbf{H}_N<\ln(N)+1$。（ln表示自然对数）

7.5　考虑Chang-Roberts算法，假设每个进程都是初始进程。如何在环上分布标识使得消息复杂度达到最小？在这种情况下，需要多少次消息交换？

7.6　如果只有S个初始进程，并且S个进程中的每一个进程等概率成为初始进程集，那么，Chang-Roberts算法在平均情况下的复杂度是多少？

7.7　给出算法7-7的一个初始配置，且算法实际需要$\lfloor\log N\rfloor+1$轮。另外，给出算法只需
266 要2轮的一个初始配置，不管有多少个初始进程。算法在一轮中可能终止吗？

7.8　确定Chang-Roberts算法的集合E_{CR}（正如在引理7.10之前所定义的）。

7.3节

7.9　把废止原理应用到环算法，并与Chang-Roberts算法比较。差别是什么？差别的影响有多大？

7.10　对于在Gallager/Humblet/Spira算法中所用的7种消息的每一种，确定是否这种类型的消息可发送给处于睡眠状态的节点。

7.11　假设GHS算法利用另外一种唤醒过程，来保证每个节点在N个时间单位内开始执行算法。

用归纳法证明至多$5Nl-3N$个时间单位后，每个节点处在级别l。

证明算法在$5N\log N$个时间单位内终止。

7.4节

7.12　证明平面网络上存在$O(N\log N)$的选举算法。

7.13　证明没有方向侦听的圆环网络上存在$O(N\log N)$的选举算法。（**提示**：分析Tarry

算法在圆环上的性能。）

7.14 证明没有方向侦听的超立方体网络上存在$O(N \log N)$的选举算法。

7.15 证明节点度界限为k的网络上存在$O(N(\log N + k))$的选举算法（即，网络中每个节点至多有k个近邻）。 267

第8章 终止检测

当算法达到*终止配置*时，分布式算法的计算终止。即配置中不存在进一步可应用的算法步。在终止配置中，并不总是存在这样一种情形，即，每个进程处于*终止状态*，也就是说，进程所处的状态没有事件可应用。考虑一种配置，其中每个进程处于允许接收的状态，且所有信道为空。这样的配置是终止的，但是进程状态可能是计算过程的中间状态。在这种情况下，进程并没有意识到计算已经终止，称计算的终止是*隐式的*。如果终止配置中进程的状态，就是进程的终止状态，则称计算的终止是*显式的*。计算的隐式终止也称为*消息终止*，因为在达到终止配置时，没有更多的消息要交换。显式终止也称为*进程终止*，因为如果算法显式终止，进程已经终止。

通常，设计隐式终止算法要比显式终止算法容易。确实，在算法设计过程中，忽略了有关进程正确终止的各个方面，算法设计主要关注如何限制事件发生的总数。另一方面，算法的应用可能要求进程显式终止。仅当显式终止之后，才认为计算结果是最终的，并将计算中的变量丢弃。同时，分布式算法的*死锁*导致终止配置，在这种情况下，达到终止配置时，必须重新开始计算。

本章研究将消息-终止的算法变成进程-终止的算法的一般方法。给定这样一个方法，在算 268
法设计时，只需考虑消息终止（即，保证算法只允许有限次的计算）。利用其中一种方法，可将消息-终止的算法变成进程-终止的算法。方法由另外两个算法组成，这两个算法与给定的消息-终止的算法相结合，其中一个算法负责观察计算，并检查计算是否已经达到了算法的终止状态。然后，它调用第二个算法，把终止消息扩散到所有进程中，使得它们进入终止状态。

转换的最难部分是检测算法终止性的算法。扩散的过程相当繁琐，将在8.1.3节中简要讨论。可以证明，对于已知其上波动算法的所有网络，终止检测是可能的。这些网络包括：领导人可用的网络、具有进程标识的网络、树网，和拓扑结构信息已知的网络，如已知网络直径和进程个数。另一方面，在第9章中，将要表明，对于未知大小的匿名网络，存在隐式终止算法，但是不存在计算进程输入最大值的显式终止算法。因此，当匿名网络的大小未知时，终止检测是不可能的。

对于可以进行终止检测的情况，我们将建立终止检测算法中消息交换数的下界。可以证明，存在与这个下界匹配的算法。8.1节通过给出分布式计算行为模型，形式化地引入问题，并给出下界和扩散过程。8.2节提出了基于维持进程树（或森林）的解决方法，至少包括仍在计算的所有进程。本节的解决方法并不是很难，并与8.1节的下界匹配。前两节包括所有涉及终止检测算法存在性和复杂度的一些基本结果。由于种种原因，一种终止检测算法可能比另一种终止检测算法更适合于某种特定应用。因此，8.3节和8.4节提出了许多其他解决方法。 269

8.1 预备知识

8.1.1 定义

本节将定义分布式计算模型，它处理分布式计算的终止问题。该模型由第2章的模型导出。

但是忽略了与终止问题不相关的所有问题。

把进程p的状态集合Z_p分成两个子集合，*活动状态*的进程集合与*被动状态*的进程集合。如果p的内部事件或者发送事件在c_p中是可应用的，则称p的状态c_p是活动的。否则称p的状态c_p是被动的。在处于被动状态c_p时，仅有接收事件是可应用的，或者根本没有事件是可应用的，在这种情况下，c_p是p的终止状态。如果进程p处于活动状态，则称它是活动的，否则称为被动的。显然，消息只能由活动进程发送。仅当被动进程接收消息之后，才能变成活动进程。当活动进程进入被动状态时，它就变成被动的。为了简化本章的算法描述，我们做出一些假设。

（1）*仅当在内部事件内，活动进程才变成被动的*。可以容易地修改进程满足这个假设。设（c，m，d）是p的发送（或接收）事件，其中d是被动状态。用（c，m，d'）代替p中的这个事件。其中d'是新的状态，在d'中惟一可应用的事件是内部事件（d'，d）。因为d'是处于活动状态，在内部事件（d'，d）中，p变成被动的。

（2）*当接到消息时，进程总是变成活动的*。可以容易地修改进程来满足这个假设。设（c，m，d）是p中的接收事件，其中d处于被动动态。用（c，m，d'）代替p中的这个事件。其中d'是新的状态，在d'中惟一可应用的事件是内部事件（d'，d）。因为d'是一个活动状态，p在接收事件后，变成活动状态；在它的下一个事件（d'，d）后，p变成被动状态。

（3）*p变成被动的内部事件是p的惟一内部事件*。内部事件中，忽略了p从一种活动状态到另一种活动状态的转移。因为终止检测算法一定是忘记了，不允许利用进程的局部计算是如何进行的信息，因而对于终止检测算法而言，所有活动进程都是相同的。

```
var state_p   : (active, passive) ;

S_p: { state_p = active }
     begin send ⟨mes⟩ end

R_p: { A message ⟨mes⟩ has arrived at p }
     begin receive ⟨mes⟩ ; state_p := active end

I_p: { state_p = active }
     begin state_p := passive end
```

图8-1　基本的分布式算法

在p的状态中，惟一有关的是进程的活动状态和被动状态。用变量$state_p$表示。图8-1所示的算法中给出了计算中的所有转移。通常，假设初始配置没有消息传输。初始时，进程可以是活动的也可以是被动的。

为了区别这个算法与终止检测算法，常常称这个算法为*基本算法*，称它的计算为*基本计算*或者*基础计算*。称终止检测算法为*控制算法*或者*叠加算法*。称它的计算为*控制计算*或者*叠加计算*。同样，相应的消息称为*基本消息*或者*控制消息*。

如果在每个配置中，不存在基本计算事件是可应用的，则定义谓词**term**为真。按照下列定理，如果所有进程状态是被动的，并且没有基本消息在传输中，指的就是这种情况。

定理8.1　**term** $\Leftrightarrow$ （$\forall p \in \mathbb{P}$: $state_p = passive$） $\wedge$ （$\forall pq \in E$: M_{pq}不包含<mes>消息）。

证明。如果所有进程处于被动状态，则不存在内部事件或者发送事件是可应用的。此外，

如果没有一个信道包含<mes>消息，就不会有接收事件是可应用的，因此根本没有基本事件是可应用的。

如果某些进程是活动的，则在那个进程中可能存在发送事件或者内部事件。如果某些信道包含<mes>消息，则消息的接收是可应用的。 □

在基本算法的终止配置中，每个进程等待接收消息，并一直保持等待状态。本章中讨论的问题是把控制算法加入到系统中的问题，在基本计算到达终止配置后，系统使进程进入终止状态。对于组合算法（基本算法加上控制算法），满足**term**的配置未必是终止的。总之，控制算法中会有可应用的事件。控制算法交换（控制）消息，这些消息可由被动进程发送，当它们被接收时，不会使得被动进程变成活动进程。 270 ~ 271

控制算法由*终止检测算法*和*终止发布算法*组成。终止检测算法调用Announce，这个发布算法使得进程进入终止状态。检测算法必须满足以下三个要求：

（1）**非干扰性** 检测算法不能影响基本算法的计算。

（2）**活动性** 如果**term**成立，必须在有限步内调用Announce。

（3）**安全性** 如果调用Announce，配置必须满足**term**。

终止检测问题首先是由Francez[Fra80]提出的。Francez提出了一种解决方法，它不满足非干扰性。方法通过阻塞所有事件，“冷冻”基本计算，然后检查配置，看它是否终止。如果终止，则调用Announce；否则，“解冻”基本计算，稍后重复这一过程。上述提到的要求不包括这种解决方法，即，执行基本计算时，检测算法不工作。在本章的终止检测正确性证明中，并不涉及非干扰性，这是因为从算法的描述中，显然满足这一要求。

如果在每一个初始状态，只有一个活动进程，称基本计算是集中式的。如果在每一个初始配置，有任意多个活动进程，则称基本计算是分散式的。通常在关于终止检测的文献中，称集中式基本计算为*扩散计算*（diffusing computation）。如果在控制计算中有一个特殊的进程，如果进程都只执行同一个控制算法，称控制计算是分散式的。

8.1.2 两个下界

终止检测算法的复杂度可用基本计算中所用的参数N、$|E|$和交换的消息数M表示。终止检测的复杂度还与执行波动算法的开销有关。设W表示最好波动算法的复杂度。W与所考虑的网络特征有关，例如，是否领导人可用，网络拓扑结构，以及假设的进程初始知识。 272

可以表明，对于集中式计算和分散式计算，它们的终止检测在最坏情况下的复杂度相同，都为下界M。可以证明，对于集中式基本计算，其终止检测的复杂度具有下界W。在本小节最后，讨论Chandrasekaran和Venkatesan给出的$|E|$条消息的下界。

定理8.2 *对于每个终止检测算法，存在一个基本计算，交换M条基本消息，并且在这个基本计算中，检测算法至少交换M条控制消息。*

*证明。*如果系统能够达到一个配置状态，在这个配置中，控制算法可能交换无限多的控制消息，而不出现一个基本事件。结果平凡成立。在证明的其余部分，假设控制算法只与每个基本事件中的有限条消息作用。

设γ是一个配置，p、q是其中的两个活动进程，且没有消息在传输中。如果基本算法是集中式的，从初始配置交换一条基本消息，就可达到这个配置；否则，这个配置中包含初始配置。

首先考虑从配置γ开始的计算，两个进程同时变成被动的，即系统达到配置$\delta = \mathbf{I}_p(\mathbf{I}_q(\gamma))$。在有限步内必须检测出终止性，但是$p$或$q$在没有接到来自其他进程的第一条消息之前，都不能调用Announce。否则，就会在配置中错误地检测出终止，而此时其他进程仍处于活动状态。（如果第三个进程检测出终止性，则至少需要两条消息。）因此，在检测出终止之前，在配置δ中至少交换一条控制消息。

不失一般性，假设p在配置δ中将发送控制消息。考虑从γ开始的计算，只有p变成被动的，即系统到达配置$\gamma_p = \mathbf{I}_p(\gamma)$。$p$的状态与配置$\gamma_p$和$\delta$中的状态相同。因此，$p$在配置$\gamma_p$中也发送控制消息。接着控制算法会有更多的行为，但这并不导致检测，因为q仍然是活动的。控制算法
273 停止交换消息之后，q向p发送一条基本消息，并返回到p、q都为活动的配置。接着又有更多的控制活动，但在经过有限步后，再次达到p、q都为活动的配置。并且没有消息在传输中。总之，

（1）从初始配置开始，至多交换一条基本消息，就可达到p、q都为活动，且没有消息在传输中的配置；

（2）基本算法通过交换一条消息，并迫使控制算法至少交换一条控制消息，可以从一个这样的配置转移到另一个配置；且

（3）如果基本计算在这个配置后终止，为了检测出这个终止性，至少交换一条控制消息。这就蕴含着结果。 □

定理8.3 *检测分散式基本计算的终止性在最坏情况下，至少需要交换W条控制消息。*

*证明。*考虑不交换任何消息的基本计算。其中每个活动进程在它的第一次事件中变成被动状态。如果认为检测（调用Announce）就是判定，这个基本计算要求检测算法就是波动算法。的确，必须在有限步内调用Announce，这证明了检测算法可以自身终止和判定。如果判定不能由某些进程q的一个事件进行，就要考虑另一个基本计算，其中q还没有变成被动状态。判定在因果关系上并不依赖于q中的任一事件，因此在q仍然是活动时，判定算法可能错误地调用Announce。因为检测算法是一次波动，因此它至少交换W条消息。 □

检测算法的开始　Chandrasekaran和Venkatesan[CV90]在以下两个假设条件下，得出了$|E|$条控制消息的下界。

C1　信道是fifo。

C2　基本计算开始后的任一时间，终止检测算法可能开始它的执行，即在基本计算的任一配置中。

在这些假设条件下，如果检测算法并没有通过某一条特定的边，如，pq发送控制消息，
274 就可能进行不正确的检测。正好在检测算法开始之前，基本计算经由信道pq发送了一条附加消息；而控制算法没有看到这条消息，由此导致不正确检测。Chandrasekaran和Venkatesan的算法经由每条信道发送控制消息，因此，他们的算法使得在控制算法开始之前发送的所有消息，在检测发生之前都被接收。

类似于[CV90]中的讨论，如果假设C2有效，而C1无效，可以证明该问题可能根本无解。本章假设控制算法在基本计算的初始配置中开始，即，在控制算法开始之前，基本计算不执行未得到通知的事件。如果用假设C2代替这个假设，问题有解，当且仅当信道是fifo的，在这种情形下，[CV90]给出了问题的解。

8.1.3 终止进程

为了把终止消息发布到所有进程中，需要将<stop>消息扩散到所有进程中。每个进程局部调用Announce，或一旦接到第一条<stop>消息，就向其所有近邻发送这条消息，但至多发送一次。如果进程接到来自每个近邻的<stop>消息，就执行语句**stop**，引起进程进入终止状态。图8-2所示的算法给出了发布过程。

图8-2所示的算法可用于任意连通的拓扑结构，包括有向网，不需要领导人，不需要进程标识，根本不需要网络拓扑结构知识。

```
var SentStop_p   : boolean   init false ;
    RecStop_p    : integer   init 0 ;

Procedure Announce:
    begin if not SentStop_p then
              begin SentStop_p := true ;
                    forall q ∈ Out_p do send ⟨stop⟩ to q
              end
    end

{ A ⟨stop⟩ message has arrived at p }
    begin receive ⟨stop⟩ ; RecStop_p := RecStop_p + 1 ;
          Announce ;
          if RecStop_p = #In_p then halt
    end
```

图8-2 发布算法

8.2 计算树和森林

本节描述的问题解，是基于动态地维持有向图。这个有向图也称为计算图（computation graph）。在这个图中的节点，包括了所有活动进程，以及所有传输中的基本消息。当计算图变为空时，进行终止检测。本节的解决方法网络要求是无向的，即，消息可以通过每个信道在两个方向发送。8.2.1小节描述了集中式基本计算的一种方法，其中计算图是一棵以初始进程为根的树。8.2.2小节概述了分散式基本计算的一种方法，其中计算图是森林，其中基本计算的每个初始进程为树的根。

8.2.1 Dijkstra-Scholten算法

Dijkstra-Scholten算法[DS80]检测集中式基本计算的终止性（在DS[80]中称为扩散计算）。基本计算的初始进程p_0（在DS[80]中称为环境）在检测算法中起着特殊的作用。

检测算法动态地维持一棵计算树$T = (V_T, E_T)$，具有如下两个性质。

（1）或者T为空，或者T是一棵根为p_0的有向树。

（2）集合V_T包括所有活动进程，以及传输中的所有基本消息。

当$p_0 \notin V_T$时，初始进程p_0调用Announce。由第一条性质，在这种情形下，T为空。由第二条性质，**term**成立。

在基本计算进行时，为了保持计算树的性质，当发送一条基本消息，或者不在树中的进

程变成活动时，必须对T进行扩展。当p发送基本消息<mes>时，则<mes>被插入到树中，且<mes>的父节点为p。当不在树中的进程p，由于接到来自进程q的消息变成活动时，q成为p的父节点。为了显式地表示消息的发送者，用<mes，q>表示q所发送的基本消息<mes>。

因为两个原因，需要从T中删除节点。首先，当接到一条基本消息时，将它删除。其次，
275
~
276
为了保证检测算法的进行，在终止后的有限步内，树必须解体。消息是T中的叶节点。进程维持一个变量，该变量对进程在T中的子节点数进行计数。对进程p的子节点的删除出现在另一进程q中；它或者是对子节点消息的接收，或者是对子进程q的删除。为了防止对p的已删除子节点的计数，当删除p的子节点时，可向p发送信号消息<sig，p>。这个消息代替了p中已被删除的子节点，它的删除，即，它的接收出现在进程p中，当进程p接到该信号消息时，会减少子节点的计数。

```
var state_p   : (active, passive)   init if p = p_0 then active else passive ;
    sc_p      : integer             init 0 ;
    father_p  : ℙ                   init if p = p_0 then p else udef ;

S_p: { state_p = active }
    begin send ⟨mes, p⟩ ; sc_p := sc_p + 1 end

R_p: { A message ⟨mes, q⟩ has arrived at p }
    begin receive ⟨mes, q⟩ ; state_p := active ;
          if father_p = udef then father_p := q else send ⟨sig, q⟩ to q
    end

I_p: { state_p = active }
    begin state_p := passive ;
          if sc_p = 0 then (* Delete p from T *)
            begin if father_p = p
                    then Announce
                    else send ⟨sig, father_p⟩ to father_p ;
                  father_p := udef
            end
    end

A_p: { A signal ⟨sig, p⟩ arrives at p }
    begin receive ⟨sig, p⟩ ; sc_p := sc_p − 1 ;
          if sc_p = 0 and state_p = passive then
            begin if father_p = p
                    then Announce
                    else send ⟨sig, father_p⟩ to father_p ;
                  father_p := udef
            end
    end
```

图8-3　DIJKSTRA-SCHOLTEN算法

277　图8-3所示的算法给出了这个算法。每个进程p有变量$father_p$，如果$p \notin V_T$，则$father_p$为*udef*；如果p是根，则$father_p$为p；如果p是T中非根节点，则$father_p$为p的父节点。变量sc_p给出了T中p的子节点数。

严格的正确性证明表明，图T是一棵树，仅当基本计算终止后，这棵树变为空。对于图8-3所示的算法的任一配置γ，定义

$$V_T = \{ p : father_p \neq udef \} \cup \{\langle \mathbf{mes}, p\rangle \text{ 在传输中}\} \cup \{\langle \mathbf{sig}, p\rangle \text{ 在传输中}\}$$

且

$$\begin{aligned} E_T = \quad & \{ (p, father_p) : father_p \neq udef \wedge father_p \neq p \} \\ & \cup \{ (\langle \mathbf{mes}, p\rangle, p) : \langle \mathbf{mes}, p\rangle \text{在传输中}\} \\ & \cup \{ (\langle \mathbf{sig}, p\rangle, p) : \langle \mathbf{sig}, p\rangle \text{ 在传输中}\} \end{aligned}$$

由断言P得出算法的安全性，如下定义

$$\begin{aligned} P \equiv \quad & state_p = active \Rightarrow p \in V_T && (1) \\ & \wedge (u, v) \in E_T \Rightarrow u \in V_T \wedge v \in V_T \cap \mathbb{P} && (2) \\ & \wedge sc_p = \#\{v : (v, p) \in E_T\} && (3) \\ & \wedge V_T \neq \varnothing \Rightarrow T\text{是根为}p_0\text{的树} && (4) \\ & \wedge (state_p = passive \wedge sc_p = 0) \Rightarrow p \notin V_T && (5) \end{aligned}$$

这个不变式的意义如下所述。由定义，T的节点集包括所有消息（基本消息及控制消息），且由式（1），它也包括所有活动进程。式（2）具有相当技术性，它阐明T的确是图，所有边都指向进程。式（3）表示，对每个进程子节点的计数是正确的。式（4）阐述了T是一棵根为p_0的树。式（5）用于表明，如果基本计算终止，树的确解体。对于正确性证明，注意，P蕴含着，$father_p = p$成立，仅当$p = p_0$时。

引理8.4 P是Dijkstra-Scholten算法的一个不变式。

证明。初始时，对于所有$p \neq p_0$，$father_{p0} \neq udef$，有$state_p = passive$，这就证明了式（1）。同时，$E_T = \varnothing$，证明了式（2）。对于每个p，由于$sc_p = 0$，这表明，满足式（3）。$V_T = \{p_0\}$，$E_T = \varnothing$，因此T是一棵根为p_0的树，这就证明了式（4）。V_T中的惟一进程是p_0，p_0是活动的。

$\mathbf{S}_p$: 在$\mathbf{S}_p$中没有进程变成活动的，也没有进程从V_T中被删除，因此式（1）保持。

可应用性的行为蕴含着，新节点$\langle \mathbf{mes}, p\rangle$的父节点$p$在$V_T$中。这证明了式（2）也保持。

作为行为的结果，用$\langle \mathbf{mes}, p\rangle$扩展$V_T$，用$(\langle \mathbf{mes}, p\rangle, p)$扩展$E_T$，这表明$T$仍然是一棵树，$sc_p$正确增加，反映$p$的新增子节点。因此式（3）和式（4）保持。 278

在V_T中，没有进程变成被动叶节点，也没有进程的插入。因此式（5）保持。

$\mathbf{R}_p$: 或者p已经在V_T中（$father_p \neq udef$），或者p是在行为中被插入V_T中的，因此式（1）保持。

如果对$father_p$进行赋值，则它的新值为q，如果p发送一个信号，则它的父节点也是q，q在V_T中，因此式（2）保持。

因为q的子节点$\langle \mathbf{mes}, q\rangle$或者被子节点$p$代替，或者被子节点$\langle \mathbf{sig}, q\rangle$代替，因此$q$的子节点数没有变化。因此，$sc_p$依然正确，式（3）保持。

图的结构没有变化，因此式（4）保持。

任何情形的行为后，进程p在V_T中。因此式（5）保持。

$\mathbf{I}_p$: 使得p变成被动状态，保持式（1）、式（2）、式（3）和式（4）。p以前是活动的，蕴含着p在V_T中。如果$sc_p = 0$，p被从V_T中删除，因此式（5）保持。

下面考虑，如果从T中删除p，会发生什么情况。即，如果p成为T中的被动叶节点。

如果p发送一个信号，则信号的父节点是p中最近父节点，它在V_T中。因此式（2）保持。在这种情形下，信号代替p作为$father_p$的子节点，因此sc_{father_p}保持正确，式（3）保持。图的结构未变。因此式（4）保持。

否则，如果$father_p = p$为真，$p = p_0$且p是叶节点，这蕴含着p是T中惟一的节点，它的删除使得T为空，因此式（4）保持。

$\mathbf{A}_p$：信号的接收使得p的子节点数减1，对sc_p的赋值使得式（3）保持。那个p是信号的父节点蕴含着，p在V_T中。如果$state_p = passive$且在接收后$sc_p = 0$成立，则p被删除，因此式（5）保持。

如果从V_T中删除p，不变式保持，证明同行为$\mathbf{I}_p$。 □

定理8.5　Dijkstra-Scholten算法（图8-3所示的算法）是正确的终止检测算法，利用M条控制消息。

证明。定义S为所有子节点计数的和，即$S = \sum_{p \in P} sc_p$。初始时，S为0。当发送基本消息时
279 （在$\mathbf{S}_p$中），S增加。当接到控制消息时（在$\mathbf{A}_p$中），S减小。由式（3），S从不为负。这蕴含着在任何计算中，控制消息数从不会超过基本消息数。

为了证明算法的活动性，假设基本计算已经终止。终止后，只有行为$\mathbf{A}_p$能够发生，因为在每次这样的迁移中，S减1，算法达到终止状态。观察这样的配置中，V_T不含消息。且由式（5），V_T不含被动叶节点。因此，T不含叶节点，这蕴含着T为空。当p_0删除自身时，树变为空。在这一步中，程序满足，p_0调用Announce。

为了证明安全性，注意只有p_0调用Announce。当它从V_T中删除自身时，才调用Announce。由式（4），当发生这种情形时，T为空，这蕴含着**term**成立。 □

Dijkstra-Scholten算法在控制通信和基本通信方面，达到了很好的平衡。对于p向q发送的每一个基本消息，算法中q只向p发送一次控制消息。控制通信等于定理8.2中给出的下界，因此对于集中式计算的终止检测，算法达到了最坏情况下的最优算法。

在本小节的描述中，所有消息显式地携带父节点。但是没有必要这样做，因为基本（或者控制）消息的父节点总是它的发送者（或者目的地）。

8.2.2　Shavit-Francez算法

Shavit-Francez将Dijkstra-Scholten算法推广到分散式基本计算上[SF86]。在它们的算法中，计算图是森林，其中每棵树以基本计算的初始进程为根。以p为根的树用T_p表示。

算法维持图$F = (V_F, E_F)$，满足（1）或者F为空，或者F是森林，其中的每棵树以初始进程为根；（2）V_F包括所有活动进程和所有基本消息。正如在Dijkstra-Scholten算法中所作的那样，当图变空时，进行终止检测。令人遗憾的是，在森林的情形下，不容易看出图是否为空。确实，以p_0为根的计算树的性质蕴含着，p_0观测树是否为空，当树为空时，它调用Announce。在森林的情形，每个初始进程只观察自己所属树的是否为空，但是这并不蕴含森
280 林是否为空。

可用一次波动来验证所有树已经解体。维持森林具有另一个性质，就是如果树T_p变为空，此后都为空。注意这样并不能防止p变成活动状态，但如果在它的树解体之后，p变成活动的，它被插入到另一个初始进程的树中。仅当它的树解体时，每个进程参加波动。当波动判定时，调用Announce。（如果所选的波动算法在每个进程中产生一次判定，调用Announce是多余的；在这种情况下，在判定和完成波动算法之后，进程简单地终止。）

图8-4给出了这个算法，其中没有显式地给出波动算法。每个进程p有一个变量$father_p$，如果$p \notin V_F$，则$father_p$为$udef$；如果p是根，则$father_p$为p；如果p是F中非根节点，则$father_p$为

p的父节点。变量sc_p给出F中p的子节点数。布尔变量$empty_p$为真，当且仅当p的树为空。

```
var state_p   : (active, passive)   init if p is initiator then active else passive ;
    sc_p      : integer             init 0 ;
    father_p  : ℙ                   init if p is initiator then p else udef ;
    empty_p   : boolean             init if p is initiator then false else true ;

S_p: { state_p = active }
    begin send ⟨mes, p⟩ ; sc_p := sc_p + 1 end

R_p: { A message ⟨mes, q⟩ has arrived at p }
    begin receive ⟨mes, q⟩ ; state_p := active ;
          if father_p = udef then father_p := q else send ⟨sig, q⟩ to q
    end

I_p: { state_p = active }
    begin state_p := passive ;
          if sc_p = 0 then (* Delete p from F *)
            begin if father_p = p
                    then empty_p := true
                    else send ⟨sig, father_p⟩ to father_p ;
                  father_p := udef
            end
    end

A_p: { A signal ⟨sig, p⟩ arrives at p }
    begin receive ⟨sig, p⟩ ; sc_p := sc_p - 1 ;
          if sc_p = 0 and state_p = passive then
            begin if father_p = p
                    then empty_p := true
                    else send ⟨sig, father_p⟩ to father_p ;
                  father_p := udef
            end
    end
```

进程并发地执行波动算法，其中，仅当$empty_p$为true时，p发送或者判定；*decide*调用Announce。

图8-4 Shavit-Francez算法

算法的正确性证明类似于Dijkstra-Scholten算法的证明。对于图8-4所示的算法的任一配置γ，定义

$$V_F = \{\, p : father_p \neq udef \,\} \cup \{\text{<mes, } p\text{> 在传输中}\} \cup \{\text{<sig, } p\text{> 在传输中}\}$$

且

$$\begin{aligned} E_F = \ & \{\, (p, father_p) : father_p \neq udef \wedge father_p \neq p \,\} \\ & \cup \{\, (\text{<mes, } p\text{>}, p) : \text{<mes, } p\text{>在传输中}\} \\ & \cup \{\, (\text{<sig, } p\text{>}, p) : \text{<sig, } p\text{> 在传输中}\} \end{aligned}$$

由断言Q得出算法的安全性，如下定义。它是算法的不变式。不变式的证明类似于引理8.4的证明。Q的式（1）到式（5）的意义与Dijkstra-Scholten算法中的不变式相同。式（6）表示，每个进程正确地记录了是否它仍然是森林中树的根。当然，如果没有一个进程为树的根，则森林为空。

$$
\begin{aligned}
Q \Leftrightarrow \quad & state_p = active \Rightarrow p \in V_F && (1)\\
\wedge\ & (u,\ v) \in E_F \Rightarrow u \in V_F \wedge v \in V_F \cap \mathbb{P} && (2)\\
\wedge\ & sc_p = \#\{v:\ (v,\ p) \in E_F\} && (3)\\
\wedge\ & V_F \neq \varnothing \Rightarrow F\text{是森林} && (4)\\
\wedge\ & (state_p = passive \wedge sc_p = 0) \Rightarrow p \notin V_F && (5)\\
\wedge\ & empty_p \Leftrightarrow T_p\text{为空} && (6)
\end{aligned}
$$

引理8.6　Q是Shavit-Francez算法的一个不变式。

证明。初始时，对于每个非初始进程p，$state_p = passive$，对于每个初始进程p，$father_p = p$，这证明了式（1）。并且$E_F = \varnothing$，证明了式（2）。对于每个p，$sc_p = 0$，这表明，式（3）被满足。$V_F = \{p:\ p$是初始进程$\}$，$E_F = \varnothing$，因此F是由一个节点的树组成的森林，每棵树由初始进程构成。这就证明了式（4）。V_F中的进程是初始进程，它们是活动的，这就证明了式（5）。初始时，$empty_p$相等（p是非初始进程），且T_p为空，当且仅当p不是初始进程，这就证明了式（6）。

$\mathbf{S}_p$: 在$\mathbf{S}_p$中没有进程变成活动的，也没有进程从V_F中被删除，因此式（1）保持。

行为的可应用性蕴含着，新节点的父节点p在V_F中。这证明了式（2）也保持。

作为行为的结果，用<**mes**，p>扩展V_F，用（<**mes**，p>，p）扩展E_F，这表明F仍然是森林，sc_p正确地增加，反映p的新增子节点。因此式（3）和式（4）保持。

在V_F中，没有进程变成被动叶节点，也没有进程的插入。因此式（5）保持。

当新叶节点被加入到非空树中，不存在树变成非空树，因为$empty$变量不变，因此式（6）保持。

$\mathbf{R}_p$: 或p已经在V_F中（$father_p \neq udef$），或在行为中p被插入，因此式（1）保持。

如果对$father_p$进行赋值，则它的新值为q，如果p发送一个信号，则它的父节点也是q，q在V_T中，因此式（2）保持。

因为q的子节点数<**mes**，q>或者被子节点p代替，或者被子节点<**sig**，q>代替，因此q的子节点没有变化。因此sc_p依然保持式（3）正确。

图的结构没有变化，因此式（4）保持。

任何情形下，不存在进程变成被动叶结点，也没有进程插入V_F中。因此式（5）保持。

没有树变成空或者变成非空，因此式（6）保持。

$\mathbf{I}_p$: 使得p被动保持式（1）、式（2）、式（3）和式（4）。p曾经是活动的，蕴含着p在V_F中。

如果$sc_p = 0$，从V_F中删除p，因此式（5）保持。

下面考虑，如果将p从F中删除，会发生什么？即，如果p是F中的被动叶结点。

如果发送一个信号，则信号的父节点是p中最后的父节点，它在V_F中。因此式（2）保持。在这种情形下，信号代替p作为$father_p$的子节点，因此sc_{father_p}保持正确，式（3）保持。图的结构未变。因此式（4）保持。否则，如果$father_p = p$为真，p是根且p是叶节点，这蕴含着p是T_p中惟一的节点，它的删除使得T_p为空，对$empty_p$的赋值使得式（6）保持。

$\mathbf{A}_p$: 信号的接收使得p的子节点数减1，对sc_p的赋值使得式（3）保持。那个p是信号的父节点蕴含着，p在V_F中。如果$state_p = passive$且在信号接收后$sc_p = 0$成立，则p被删除，因此式（5）保持。

281
~
283

如果从V_F中删除p，不变式保持，证明同$\mathbf{I}_p$。 □

定理8.7　Shavit-Francez算法（图8-4所示的算法）是正确的终止检测算法，利用$M + W$条

控制消息。

证明。如同定理8.5的证明，信号的数目从不会超过基本消息的数目。除了信号消息，控制算法只为一次波动发送消息，由此可得，至多发送$M + W$条控制消息。

为了证明算法的活动性，假设基本计算已经终止。在有限步后，终止检测算法达到终止配置。如同定理8.5的证明，在这个配置中，F为空。因此，波动中的所有事件可在每个进程中执行。配置是终止的，蕴含着波动的所有事件都被执行，至少包括一次判定，它引起Announce的调用。

为了证明安全性，注意只有在判定出现在波动算法中时，才调用Announce。这蕴含着每个进程p已经发送了一次波动消息或者已经判定。当p这样做时，算法蕴含着$empty_p$为真。没有行为再次使得$empty_p$为假。因此（对于每个p）当调用Announce时，$empty_p$为真。由式（6），V_F为空，这蕴含着**term**成立。 □

Shavit-Francez算法中交换的控制消息数，与定理8.2和8.3中证明的下界在数量级上相匹配。因此对于分散式计算的终止检测（如果利用最优的波动算法），算法达到了最坏情况下的最优算法。

本节中考虑的算法要求通信信道是双向的。对于p向q发送的每一条基本消息，必须发送一条从q到p的信号。平均情况下的复杂度等于最坏情况下的复杂度。每次执行中，每条基本消息需要一条信号消息。在Shavit-Francez算法的情形，仅执行一次波动算法。

8.3 基于波动的方法

除了8.2节介绍的两个终止检测算法，有两个原因需要考虑其他终止检测算法。首先，这 284
里给出的算法只能用于双向信道。其次，尽管这些消息的复杂度在最坏情况下是最优的，还有一些其他算法有更好的平均情况复杂度。每次执行中，前面小节所述算法利用最坏情况下的消息数。

本节研究一些基于反复执行波动算法的算法，在每次波动以后，或者检测终止性，或者开始新的波动。在每个进程中，如果满足局部条件，就检测终止性。

首先考虑算法的具体实例。在所有情形中，采用环算法作为波动算法。并假定环可以嵌入到网络中，作为网络的子拓扑结构，但是对于基本消息的交换并不限于环中的信道。进程从p_0到p_{N-1}编号。通过p_0向p_{N-1}发送令牌完成环的初始化。进程p_{i+1}（对于$i < N-1$）把令牌转发给进程p_i。当进程p_0接收返回的令牌时，令牌遍历结束。

第一个这类方法是Dijkstra、Feijen和Van Gasteren提出的算法（8.3.1小节），该算法检测具有同步消息传递的计算的终止性。还有一些作者将这个算法推广到具有异步消息传递的计算上，其中的主要问题是验证通信信道为空。我们讨论了Safra提出的方法（8.3.2小节），它在每个进程中计算收发的消息数，通过比较消息数，来确认信道为空。它也可能利用确认机制（8.3.3小节），但是这种方法再次要求信道是双向的，控制消息数至少等于Shavit-Francez算法中所用的消息数。

在8.3.4小节，将检测的一般原理推广到任意波动算法的使用上。

8.3.1 Dijkstra-Feijen-Van Gasteren算法

Dijkstra-Feijen-Van Gasteren算法[DFG83]利用同步消息传递（synchronous message

passing）检测基本计算的终止性。这种计算的行为在算法8-5中给出。在这些计算中，终止用以下方式描述

285

$$\textbf{term} \Leftrightarrow \forall p:\ state_p = passive$$

将该算法和**term**与图8-1所示的算法和定理8.2进行比较。

```
var state_p    : (active, passive) ;

C_pq: { state_p = active }
      begin (* p sends a basic message, which is received by q *)
            state_q := active
      end

I_p:  { state_p = active }
      begin state_p := passive end
```

图8-5　具有同步消息的基本算法

算法由一系列步骤组成。每步易于理解。由不变式可得算法的正确性。这里的处理取自文献[DFG83]。以下，t表示持有令牌的进程数，如果令牌在传输中，则传输令牌的进程数是动态的。令牌只能由进程p_i转发，每次t减1。当$t = 0$时，波动结束。因此，选择不变式P，满足从P、$t = 0$和p_0中的其他信息可推出终止性。当p_0开始波动时，不变式必须成立，即$t = N-1$。

作为首次尝试，设$P = P_0$，这里

$$P_0 \equiv \forall i\ (N > i > t):\ state_{p_i} = passive$$

当$t = N-1$时，P_0为真。如果$t = 0$且$state_{p_0} = passive$，由此断言，可推出终止性。仅当只有被动进程转发令牌时，令牌的转发才保持P_0，因此我们采用以下规则。

规则1　只有进程处于被动状态时，它才处理令牌。

在这种规则下，通过转发令牌和内部行为，保持P。令人遗憾的是，通信行为并不能保持P。当进程p_i激活进程p_j时，其中$j > t$和$i \leqslant t$，谓词P_0为假；参见习题8.4。因为P_0可为假，可用弱断言（$P_0 \vee P_1$）代替P，选择P_1使得每次P_0值为假时，P_1为真。我们为每个进程提供一种颜色，或为白色或为黑色。设$P = (P_0 \vee P_1)$，其中，

286

$$P_1 \equiv \exists j\ (t \geqslant j \geqslant 0):\ color_{p_j} = black$$

如果发送进程对自己着黑色，则P_0的每个假值，保证P_1为真或者变为真。

规则2　发送进程变成黑色。

因为（$P \wedge color_{p_0} = white \wedge t = 0$）$\Rightarrow \neg P_1$，因此用新不变式仍然可能检测终止性，即处理令牌时，通过P_0是否是白色（且被动），判定终止性。

弱化P可以防止通信使得谓词为假的情况。但是令牌的转发可使得弱断言为假，即，如果进程t是惟一的黑色进程，并转发令牌。可以进一步弱化P解决这种情况。假定令牌也具有颜色（白色或者黑色），P弱化为（$P_0 \vee P_1 \vee P_2$），其中

$$P_2 \equiv \text{令牌为黑色}$$

如果黑色进程转发令牌，令牌变黑，则令牌转发保持P_2。

规则3 当黑色进程转发令牌时（除去进程p_0），令牌变成黑色。

因为（令牌为白色）$\Rightarrow \neg P_2$，p_0通过它是否接到白色令牌（且自己也是白色，并处于被动状态），仍然可检测出终止性。

现在可验证，内部行为、基本通信以及令牌转发保持P。变黑的令牌引起*不成功波动*（unsuccessful wave）现象；如果返回令牌为黑色，p_0不能判定终止性；如果波动不成功结束，则必须开始新的波动。

规则4 当波动不成功结束时，p_0开始新一次波动。

如果没有一种方式使得黑色进程再次变成白色进程，下次波动一定会像上次一样不会成功。当转发令牌时，黑色进程将使得令牌为黑色，同样引起下一次波动失败。

观察可见，如果$i > t$，变成白色的进程p_i并不使P为假，当p_0向p_{N-1}发送令牌时，开始波动，P总是变为真。这蕴含着，一旦转发令牌，变成白色可以安全地发生。

规则5 在发送令牌之后，每个进程接着变成白色。

在基本计算终止后，变成白色可保证波动最终成功。如图8-6所示的算法。 287

```
var state_p  : (active, passive) ;
    color_p  : (white, black) ;

C_pq: { state_p = active }
      begin (* p sends a basic message, which is received by q *)
            color_p := black ;   (* Rule 2 *)
            state_q := active
      end

I_p:  { state_p = active }
      begin state_p := passive end

Start the detection, executed once by p_0:
      begin send ⟨tok, white⟩ to p_{N-1} end

T_p:  (* Process p handles the token ⟨tok, c⟩ *)
      { state_p = passive }   (* Rule 1 *)
      begin if p = p_0
                then if (c = white ∧ color_p = white)
                        then Announce
                        else send ⟨tok, white⟩ to p_{N-1}   (* Rule 4 *)
                else if (color_p = white)   (* Rule 3 *)
                        then send ⟨tok, c⟩ to Next_p
                        else send ⟨tok, black⟩ to Next_p ;
            color_p := white   (* Rule 5 *)
      end
```

图8-6 Dijkstra-Feijen-Van Gasteren算法

定理8.8 *对于利用同步消息传递的基本计算，Dijkstra-Feijen-Van Gasteren算法（图8-6所示的算法）是正确的终止检测算法。*

证明。谓词$P = (P_0 \vee P_1 \vee P_2)$，设计算法满足，$P$是算法的不变式。当处于被动状态的白色$p_0$处理白色令牌时，检测终止性。实际上，当这种情况发生时，令牌的颜色蕴含着$\neg P_2$，p_0的颜色和$t = 0$蕴含$\neg P_1$，则P_0和p_0的状态蕴含**term**，因此算法是安全的。

288　为了证明活动性，假设基本计算终止。从那时起，所有进程无延迟地转发收到的令牌。当令牌完成它的首次完全巡回，该巡回在终止后开始，所有进程为白色；当令牌完成它的下次巡回，检测终止性。□

现在估算算法中所用的控制消息数。对于每两条基本消息，定理8.2证明中所用的基本计算使得算法至少利用令牌的一轮巡回。因此算法的最坏情况复杂度为$1/2N \cdot M$条消息。参见习题8.5。

对于“平均”基本计算，算法利用少得多的消息。假设基本计算的时间复杂度为T，因为令牌总是顺序转发，假设令牌在基本计算终止之前，转发大约T次是合理的。（甚至这种估算可能是太悲观，因为令牌的转发在活动进程中被挂起。）由于终止后，令牌转发小于$3N$次。这种情况下，算法交换$T+3N$条控制消息。基本计算的复杂度至少为T（即，它的时间复杂度），但是如果计算包含足够的并行性，它的消息复杂度可高达$\Omega(N \cdot T)$。如果并行性允许每个进程每个时间单位发送α条消息，基本计算的消息复杂度为$N \cdot T \cdot \alpha$，即，$\Omega(N \cdot T)$。控制消息数r为$O(N+T)$，这要比终止检测问题期望的最坏情况复杂度好的多。

8.3.2 基本消息的计数：Safra算法

在Dijkstra-Feijen-Van Gasteren算法的基本计算中，假设同步消息传递，对于它的可应用性具有一般的局限性。几个作者将这一算法推广到具有异步消息传递的计算中（参见图8-1所示的算法）。本小节将讨论Safra算法[Dij87]，它的平均情况复杂度可与Dijkstra-Feijen-Van Gasteren算法的复杂度相比。

对于每个配置，定义传输中的消息数为B。现在**term**等价于

$$(\forall p:\ state_p = passive)\ \wedge\ B = 0$$

289　设计不变式P，满足从P，t =0及p_0中的其他信息可得出算法的终止性。当p_0开始波动时，即当$t = N-1$时，不变式成立。为了使得B中的信息在进程（虽然是分布式方式）中可用，进程p中有消息计数器mc_p，且进程将P_m作为不变式，其中

$$P_m \equiv B = \sum_{p \in P} mc_p$$

初始时，对于每个p，$mc_p = 0$，可得不变式P_m。进程服从以下规则。

规则M　当发送消息时，进程p增加它的消息计数器增加1。当接收一条消息时，进程p的消息计数器减少1。

当进程p_0接到令牌（$t = 0$）时，不变式必须允许P_0判定**term**成立。因为**term**包括对B值的限制，可使令牌携带整数q来计算已经转发给它的进程中消息计数器的总和。作为首次尝试，可设$P = P_m \wedge P_0$，其中

$$P_0 \equiv \left(\forall i (N > i > t): state_{p_i} = passive\right) \wedge \left(q = \sum_{N>i>t} mc_{p_i}\right)$$

当$t = N-1$，$q = 0$时，P_0为真，如果$t = 0$，那么P蕴含着

$$\left(\forall i > 0: state_{p_i} = passive\right) \wedge \left(mc_{p_0} + q = B\right)$$

因此，如果$state_{p_0} = passive$且$mc_{p_0} + q = 0$。p_0可推出终止性。

当p_0通过发送令牌p_{N-1}且$q = 0$时开始波动，建立断言P_0。如果只有被动进程转发令牌并增加消息计数器的值，则令牌的转发保持断言P_0。因此我们采用以下规则。

规则1 当进程处于被动状态时，只处理令牌。当进程转发令牌时，向q增加消息计数器的值。在这种规则下，令牌的转发和内部行为使得P保持。遗憾的是，当进程p_i接到消息的$i > t$时，并不能保持P。在接收状态，发生P_0为假。即，仅当$B > 0$，它是可应用的。因为在为假之前，P成立。这蕴含着P_1成立，其中

$$P_1 \equiv \left(\sum_{i<t} mc_{p_i} + q\right) > 0$$

290

当进程p_i接到$i > t$的消息时，断言P_1为真。因此，当进程p_i接到消息且$i > t$时，用$P_m \wedge (P_0 \vee P_1)$来定义弱断言$P$。

修改过的断言仍然允许由p_0进行终止检测，而条件不变。因为如果$t = 0$，P_1读到$mc_{p_0} + q > 0$，因此，由于$mc_{p_0} + q = 0$（这是检测所要求的），$\neg P_1$成立。令牌转发保持P_1，基本消息的发送同样保持P_1。但是，当进程p_i接到$i < t$的消息时，P_1为假。如同在8.3.1小节，可以对每个进程着色，修改本规则。

规则2 接收进程变成黑色。

用$P_m \wedge (P_0 \vee P_1 \vee P_2)$代替$P$，其中

$$P_2 \equiv \exists j(t > j \geq 0) : color_{p_j} = black$$

使P_1为假的每次接收，使P_2为真。因此任何一次基本行为都不会使得P为假。因为$(P \wedge color_{p_0} = white \wedge t = 0) \Rightarrow \neg P_2$，因此用新断言仍然可能检测终止性，即，处理令牌时，通过p_0是否是白色（且被动），判定终止性。

弱化P可以防止基本事件使谓词为假的情况。但是令牌的转发可使得弱断言为假，即，如果进程t是惟一的黑色进程，并转发令牌。可以进一步弱化P来解决这种情况。再次假定令牌也具有颜色（白色或者黑色），P被弱化为$P_m \wedge (P_0 \vee P_1 \vee P_2 \vee P_3)$，其中

$$P_3 \equiv \text{令牌为黑色}$$

如果黑色进程转发令牌，令牌变黑，则令牌转发保持P_3。

规则3 当黑色进程转发令牌时，令牌变成黑色。

因为（令牌为白色）$\Rightarrow \neg P_3$，p_0通过它是否接到白色令牌（且自己也是白色，处于被动状态），仍然可检测出终止性。

现在可验证，内部行为、基本通信以及令牌转发保持P。如果令牌返回p_0，且p_0为黑色，或者$mc_{p_0} + q \neq 0$，那么波动就不能成功结束。如果波动不成功的结束，则必须开始新的波动。

规则4 当波动不成功结束时，p_0开始新一次波动。

如果没有一种方式使得黑色进程再次变成白色进程，下次波动还是不会成功。当转发令牌时，黑色进程将使得令牌为黑色，同时引起下一次波动失败。

观察可见，如果$i > t$，变成白色的进程p_i并不使得P为假，当p_0通过向p_{N-1}发送令牌开始波动时，P总是变为真。这蕴含着，一旦转发令牌，令牌可以安全地变成白色。

规则5 在发送令牌之后，每个进程立即变成白色。

在基本计算终止后，变成白色足以保证波动最终成功。如图8-7所示的算法所示。

```
var state_p   : (active, passive) ;
    color_p   : (white, black) ;
    mc_p      : integer init 0 ;

S_p: { state_p = active }
    begin send ⟨mes⟩ ;
          mc_p := mc_p + 1    (* Rule M *)
    end

R_p: { A message ⟨mes⟩ has arrived at p }
    begin receive ⟨mes⟩ ; state_p := active ;
          mc_p := mc_p − 1 ;    (* Rule M *)
          color_p := black    (* Rule 2 *)
    end

I_p: { state_p = active }
    begin state_p := passive end

Start the detection, executed once by p_0:
    begin send ⟨tok, white, 0⟩ to p_{N−1} end

T_p: (* Process p handles the token ⟨tok, c, q⟩ *)
    { state_p = passive }    (* Rule 1 *)
    begin if p = p_0
            then if (c = white) ∧ (color_p = white) ∧ (mc_p + q = 0)
                    then Announce
                    else send ⟨tok, white, 0⟩ to p_{N−1}    (* Rule 4 *)
            else if (color_p = white)    (* Rules 1 and 3 *)
                    then send ⟨tok, c, q + mc_p⟩ to Next_p
                    else send ⟨tok, black, q + mc_p⟩ to Next_p ;
          color_p := white    (* Rule 5 *)
    end
```

图8-7　Safra算法

定理8.9　对于带异步消息传递的计算，Safra算法（图8-7所示的算法）是正确的终止检测算法。

证明。谓词$P = P_m \wedge (P_0 \vee P_1 \vee P_2 \vee P_3)$，设计算法满足，$P$是算法的不变式。

为了证明安全性，观察当$t = 0$，$state_{p_0} = passive$，$color_{p_0} = white$，且$mc_{p_0} + q = 0$时，检测终止性。这些条件蕴含着$\neg P_3$、$\neg P_2$及$\neg P_1$，因此$P_m \wedge P_0$。此结论与$state_{p_0} = passive$和$mc_{p_0} + q = 0$，蕴含**term**。

为了证明活动性，基本计算终止后。消息计数器为常数，其和为0。在这种配置中开始的波动以$mc_{p_0} + q = 0$且所有进程变成白色为结束。这样保证了下一次波动是成功的。　□

不像Dijkstra-Feijen-Van Gasteren算法，Safra算法没有限定的最坏情况复杂度。当所有令牌处于被动状态，令牌可被无限次的传递，但是有些基本消息仍在传输相当长的时间。至于Dijkstra-Feijen-Van Gasteren算法，对于时间复杂度为T的基本计算，消息复杂度期望值为$\Theta(T + N)$。

向量计数算法

Mattern[Mat87]提出了可与Safra算法相比的算法，该算法保存对每个目的节点计数的专门消息。对于进程p，用数组$mc_p[\mathbb{P}]$存放消息的计数。当p向q发送消息时，p用$mc_p[q] = mc_p[q]+1$

更新计数，当p接收消息时，p用$mc_p[p] = mc_p[p]-1$更新计数。令牌也携带计数数组，当p处理令牌时，mc_p被加到令牌中，重设为$\vec{0}$（数组的每个元素为0），令牌等于$\vec{0}$时，检测终止性。

向量计数算法优于Safra算法。但是也有严重的不足。该算法的优点是终止检测快，即，在终止出现后令牌的一轮循环内，进行检测。第二个优点是只要计算没有终止，令牌时常被挂起。这可能减少算法中交换的控制消息数。在Safra算法中，每个被动进程转发令牌，在向量计数算法中，如果令牌中的信息蕴含着消息仍在到p的路上，则进程p将不转发令牌。

向量计数算法的主要缺点是令牌包含大量信息（即，每个进程一个整数），在每条控制消息中都被传递。如果进程数较小，这种缺点可能不是太严重。另一个缺点是，需要有关其他进程标识的知识。如果用数组表示向量，每个进程必须预先知道全体标识的集合。如果用进程-整数对的集合表示向量，则可放宽这个要求。初始时，每个进程必须至少知道它的近邻标识（以便正确地增加计数），其他的标识在计算的过程中得知。

291 ~ 293

8.3.3 利用确认

Safra算法对发送和接收的基本消息进行计数，以便确定没有基本消息在传输中，也可能使用确认来确保这样，几位作者扩充了Dijkstra-Feijen-Van Gasteren算法，参见Naimi[Nai88]。这里仅对改变的原理作简要讨论，因为所得算法并没有对Shavit-Francez算法有所改进，因此已经过时。

首先，*没有消息在传输中*等价于：*对于所有p，不存在p所发送的消息在传输中*。每个进程对其所发送的消息负责，即，在没有肯定那个进程所发送的所有基本消息都被接收之前，不能调用Announce。检测方法按照以下方式，为每个进程定义了一个局部条件$quiet(p)$。

$$quiet(p) \Rightarrow (state_p = passive \wedge \text{传输中没有}p\text{所发送的基本消息})$$

则 $(\forall p:\ quiet(p)) \Rightarrow \mathbf{term}$。

为了确定没有p所发送的消息在传输中。每条消息都要被确认，每个进程保存对必须接收的确认消息的计数。形式地，断言P_a定义如下。

$$P_a \equiv \forall p:\ (unack_p = \#(\text{传输发送方为}p\text{的消息}) + \#(\text{传输到}p\text{的确认消息}))$$

在下述规则下，它是不变式。

规则A 当发送消息时，p使得$unack_p$增加；当接收来自q的消息时，p向q发送确认消息；当接到确认消息时，p使得$unack_p$减小。

上述对于*quiet*（即，$quiet(p)$蕴含着p是被动的且没有p所发送的基本消息在传输中）所做的要求可以得到满足，如果*quiet*如下定义

$$quiet(p) \equiv (state_p = passive \wedge unack_p = 0)$$

检测算法的推导与Dijkstra-Feijen-Van Gasteren算法的步骤近似。并通过考虑断言P_0来开始，P_0定义为

$$P_0 \equiv \forall i (N > i > t):\ quiet(p)$$

引入P_1需要仔细。因为当进程p_i发送消息时，进程p_i（$i \leqslant t$）对进程p_j（$j > t$）的激活不会与由p_i发送消息发生在同一事件中。然而，当p_j被激活时（因此使得P_0为假），$unack_{p_i} > 0$。

如果断言P_b如下定义

$$P_b \equiv \forall p:\ (unack_p > 0 \Rightarrow color_p = black)$$

在下述规则下，它是不变式。

规则B　当进程发送时，它变成黑色；仅当进程静止时，才变成白色。

结论正好证明了P_0为假时，P_1成立，因此（$P_0 \vee P_1$）不为假。

图8-8所示的算法给出了所得算法。它的不变式为$P_a \wedge P_b \wedge (P_0 \vee P_1 \vee P_2)$，其中

$P_a \equiv \forall p:\ (unack_p =$ #（传输发送方为p的消息）
+ #（传输目的为p的确认消息））

$P_b \equiv \forall p:\ (unack_p > 0 \Rightarrow color_p = black)$

$P_0 \equiv \forall i(N > i > t):\ quiet\ (p)$

$P_1 \equiv \exists i(t \geqslant i \geqslant 0):\ color_{p_i} = black$

$P_2 \equiv$ 令牌为黑色

```
var state_p   : (active, passive) ;
    color_p   : (white, black) ;
    unack_p   : integer init 0 ;

S_p: { state_p = active }
    begin send ⟨mes⟩ ; unack_p := unack_p + 1 ;    (* Rule A *)
          color_p := black    (* Rule B *)
    end

R_p: { A message ⟨mes⟩ from q has arrived at p }
    begin receive ⟨mes⟩ ; state_p := active ;
          send ⟨ack⟩ to q    (* Rule A *)
    end

I_p: { state_p = active }
    begin state_p := passive end

A_p: { An acknowledgement ⟨ack⟩ has arrived at p }
    begin receive ⟨ack⟩ ; unack_p := unack_p − 1 end    (* Rule A *)

Start the detection, executed once by p_0:
    begin send ⟨tok, white⟩ to p_{N−1} end

T_p: (* Process p handles the token ⟨tok, c⟩ *)
    { state_p = passive ∧ unack_p = 0 }
    begin if p = p_0
              then if c = white ∧ color_p = white
                       then Announce
                       else send ⟨tok, white⟩ to p_{N−1}
              else if (color_p = white)
                       then send ⟨tok, c⟩ to Next_p
                       else send ⟨tok, black⟩ to Next_p ;
          color_p := white    (* Rule B *)
    end
```

图8-8　利用确认的终止检测

定理8.10　对于具有异步消息传递的计算，图8-8所示的算法是正确的终止检测算法。

证明。当进程p_0静止且进程为白色时，发布终止消息。这些条件蕴涵$\neg P_2$和$\neg P_1$，因此$P_a \wedge P_b \wedge P_0$成立。这蕴含着对于$quiet(p_0)$，所有进程静止，因此**term**成立。

当基本计算终止时，经过若干时间，收到所有确认消息，且所有进程变成静止状态。所有进程为静止时开始的第一次波动，在终止时变成白色，在下一次波动结束时发布终止消息。 □

基于有限消息延迟的方法 在消息延迟受常数μ限定（参见3.2节）的假设下，文献[Tel91b，4.1.3小节]描述了一类解决终止检测（和其他问题的）方法。在这些方法中，在最近的消息发送后，进程在μ个时间间隔保持非静止状态，同时只要进程处于非静止状态，就保持黑色。在解决方法中，利用上面描述的确认方法。进程p变成静止，如果（1）p最近发送的消息至少在μ个时间单位前，（2）p是被动的。算法的完整形式推导留给读者。 294 ~ 296

8.3.4 带波动的终止检测

到目前为止所讨论的终止检测算法，利用环形拓扑结构进行控制通信。所有算法基于环上的波动算法。类似的方法也涉及其他拓扑结构上的一些算法。例如Francez & Rodeh[FR82]和Topor[Top84]提出了用有根生成树进行控制通信的算法。Tan和Van Leeuwen[TL86]给出了环网、树网和任意网的分散式解决方法。纵观这些方法，它们大多数算法几乎相似，除了所依赖的波动算法不同。

本小节基于任意波动算法，概述了终止检测算法（和它的不变式）的导出结果，而非某种具体的算法（如，环网算法）。对于每次波动，进程发送消息，或者判定的第一个事件，称为对那个进程的访问。如果需要，假设进程在满足局部条件之前，能够挂起访问。那个进程中同一波动的后一事件永远不会被挂起。

本小节所导出的结论仅适合于基本计算中同步消息传递的情况（就像在8.3.1小节中的那样）。这个导出结果可以推广到异步消息传递的情况，类似于8.3.2节和8.3.3节的情况。

当波动进行判定时，算法的不变式必须允许进行终止检测。因此，作为首次尝试，设$P = P_0$，其中

$$P_0 \equiv \text{所有被访问过的进程是被动的}$$

确实，当判定发生时，所有进程已被访问过，这个断言使得当波动判定时，进行终止检测。此外，开始波动时（没有一个进程被访问过），P_0被确定。如果观察以下的规则1，可得波动算法的活动性保持P_0。

规则1 波动只能访问被动进程。

遗憾的是，当未被访问过的进程激活已被访问过的进程时，P_0为假。因此必须为每个进程提供一种颜色，P被弱化为$(P_0 \vee P_1)$，其中

$$P_1 \equiv \text{存在一个未被访问过的黑色进程}$$

在规则2之下，弱化的不变式得到保持。

规则2 发送进程变成黑色。 297

如果只能对未访问的黑色进程访问，波动使得弱断言为假。通过进一步弱化P，可以改进这种情况。每个进程提交一种颜色，或者为白色或者为黑色，作为波动的输入。修改波动使

其能够计算所提交颜色中的最黑的。回忆一下，波动可以计算最小值，“最黑的”就是指最小值。当波动进行判定时，计算所有提交颜色中的最黑的。如果所有进程提交白色则最小值为白色，如果至少有一个进程提交黑色，则为黑色。在波动中，如果还没有进程提交黑色，则调用波动为白色。如果至少一个进程已经提交黑色，则调用波动为黑色。因此，一个进程，当它被访问时，或者提交白色，保持波动的颜色不变；或者提交黑色，对波动着黑色。

P被弱化为 $(P_0 \vee P_1 \vee P_2)$，其中

$$P_2 \equiv \text{波动为黑色}$$

在规则3之下，断言被保持。

规则3　当进程被访问时，向波动提交当前的颜色。

所有基本通信以及波动活动保持这个断言，因此它是不变式。如果进程判定结果为黑色，则波动不成功结束。在这种情况下，开始新一轮的波动。如果进程变成白色，新的波动才能成功，这种情况在访问波动后立即发生。

规则4　黑色波动中的判定进程开始新一轮波动。

规则5　进程每访问一次波动后变成白色。

这些规则保证，在基本计算终止后，波动最终成功。如果基本计算已经终止，则终止后开始的第一次波动使得所有进程为白色，下一次波动成功结束。

在算法中，任何时刻只能运行一个波动。如果有两个波动A和B并发运行，访问B后，进程变白可能侵犯波动A的不变式。因此，如果检测算法一定是分散式的，就必须用分散式的波动算法，才能做到检测算法的所有初始进程在同一个波动中合作。也可能利用另一种检测原理，在这种原理中，不同的波动可以并发地计算而不违反检测算法的正确操作，参见8.4.2小节。

8.4　其他方法

本节讨论终止检测问题的另外两种方法：信用-恢复算法和时戳算法。

8.4.1　信用-恢复算法

Mattern[Mat89a]提出了快速检测终止性的算法，即在发生后的一个单位时间内进行检测（在定义6.31的理想的定时假设下）。算法检测集中式计算的终止性，并假设每个进程可以直接向计算的初始进程发送消息（即，网络是星型的，以初始进程为中心）。

在算法中，赋给每条消息和每个进程一个信用（credit）值，该值在0和1之间（包括0和1），算法用以下断言作为不变式。

S1　所有信用值（在消息和进程中）的和为1。

S2　基本消息的信用值为正值。

S3　活动进程的信用值为正值。

按照规则（例如，被动进程），当没有指定时，持有正信用值的进程向初始进程发送它们的信用值。初始进程的作用就像一个银行，收集所有发送给它的信用，并存放在变量*ret*（表示*returned*）中。

当初始进程拥有所有信用时，对于要求正信用值的活动进程和基本消息，蕴含着不存在

这样的进程和消息，因此**term**成立。

规则1 当ret =1时，初始进程调用Announce。

为了满足活动性要求，必须保证，如果出现终止，所有信用值最终被传输到初始进程。如果基本计算已经终止，也没有任何基本消息，我们仅需关注进程持有的信用值。

规则2 当进程变成被动时，就将信用值发送给初始进程。

在初始配置中，只有初始进程是活动的，且具有正的信用值，1和$ret = 0$，这蕴含着满足S1到S3。在计算过程中，必须保持不变式。可按以下规则做到这一点。首先，当发送每条基本消息时，必须给定每条基本消息正信用值；幸而，它的发送进程是活动的，因此信用值为正。

规则3 当活动进程p发送一条消息时，它的信用值被划分在进程p和消息中。

当进程被激活时，必须给定一个正信用值。幸而，它所接收的消息包含正信用值。

规则4 当进程被激活时，它的信用值为激活它的消息的信用。

有一种情况没有包含在这些规则中，就是已经处于活动状态的进程接收基本消息。进程已有正信用值，因此并不需要消息的信用值去满足S3。然而，信用值不能被毁灭，否则这将导致对S1的侵犯。接收进程可用两种不同方式处理信用，两种方法都可得到正确算法。

规则5a 当活动进程接到一条基本消息时，就将那条消息的信用发送给初始进程。

规则5b 当活动进程接到一条基本消息时，就将那条消息的信用加到进程的信用中。

如图8-9所示的算法所示。在算法中，假设每个进程知道初始进程的名字（至少在它首次变成被动时），算法实现了规则5b。当初始进程变成被动时，它向自己发送消息。

```
var state_p  : (active, passive) init if p = p0 then active else passive ;
    cred_p   : fraction          init if p = p0 then 1 else 0 ;
    ret      : fraction          init 0 ;   for p0 only

S_p: { state_p = active }   (* Rule 3 *)
     begin send ⟨mes, cred_p/2⟩ ; cred_p := cred_p/2 end

R_p: { A message ⟨mes, c⟩ has arrived at p }
     begin receive ⟨mes, c⟩ ; state_p := active ;
           cred_p := cred_p + c   (* Rules 4 and 5b *)
     end

I_p: { state_p = active }
     begin state_p := passive ;
           send ⟨ret, cred_p⟩ to p0 ; cred_p := 0   (* Rule 2 *)
     end

A_p0: { A ⟨ret, c⟩ message has arrived at p0 }
     begin receive ⟨ret, c⟩ ; ret := ret + c ;
           if ret = 1 then Announce   (* Rule 1 *)
     end
```

图8-9 信用-恢复算法

定理8.11 信用-恢复算法（图8-9所示的算法）是正确的终止检测算法。

证明。算法实现规则1至5，这蕴含着$S_1 \wedge S_2 \wedge S_3$是不变式，其中

$$S_1 \equiv 1 = \left(\sum_{<\mathbf{mes},c>} c\right) + \left(\sum_{p \in \mathbf{P}} cred_p\right) + \left(\sum_{<\mathbf{ret},c>} c\right) + ret$$

$$S_2 \equiv \forall <\mathbf{mes}, c> \text{在传输中}: \ c > 0$$

$$S_3 \equiv \forall p \in \mathbb{P}: \ (state_p = passive \Rightarrow cred_p = 0) \wedge (state_p = active \Rightarrow cred_p > 0)$$

当$ret = 1$时，检测终止性，由此及不变式，蕴含着**term**成立。

为了证明活动性，由于终止后，没有基本行为出现，因此只会出现接收消息<**ret**，c>，每次接收使得传输中的消息数减1。因此算法达到终止配置。在这个配置中，没有基本消息（由**term**），对于所有p，$cred_p = 0$（由**term**和S_3），也没有<**ret**，c>消息（配置是终止的）。因此，$ret = 1$（由S_1），终止性得到检测。 □

如果实现规则5a，控制消息数比基本消息数多1。（这里，我们也对p_0一旦变成被动而发给自己的消息计数。）如果实现规则5b，控制消息数等于基本计算中的内部事件数加1，它至多比基本消息数多1。规则5b明显更倾向于从控制算法的消息复杂度方面考虑问题。当考虑位复杂度时，情形有所不同。在规则5a下，除ret之外，系统中每个信用的大小为2的负幂次方（即，对于某些自然数，为2^{-i}）。用它的负对数减去所要传输的位数表示信用。

信用-恢复算法是本章惟一要求在基本消息中包含附加信息的算法（即，信用）。向基本消息中增加信息称为*捎带*（piggybacking）。如果不想捎带，消息的信用可在控制消息中传输，并在基本消息之后立即发送。（下一小节，如果利用Lamport逻辑时钟实现算法，也要求捎带。）

如果（消息和进程的）信用按照固定的位数存储，就会引出问题。在这种情形下，存在最小的正信用值，不可能再对这个值一分为二。当必须分割具有最小可能值的信用时，基本计算就会被挂起，同时进程获得来自初始进程的附加信用。初始进程从ret中减去这个量（结果ret可能变成负数），并将这个量传输到请求的进程。一旦接到，进程再恢复基本计算。信用的上升引起基本计算的阻塞，这与终止检测算法不影响基本计算的要求相矛盾。幸而，这种操作非常少见。

8.4.2 基于时戳的终止检测方法

本小节基于时戳方法，讨论终止检测算法。假设进程装有时钟（2.3.3小节），为此，可用硬件时钟和Lamport逻辑时钟（2.3.3小节）。Rana[Ran83]提出了检测原理。

就像8.3.3小节中的解决方法，对于每个进程p，Rana方法基于局部逻辑谓词$quiet(p)$，其中

$$quiet(p) \Rightarrow state_p = passive \wedge \text{传输中没有进程}p\text{发送的基本消息}$$

这蕴含着，$(\forall p \ quiet(p)) \Rightarrow$ **term**。如前，$quiet$如下定义

$$quiet(p) \equiv (state_p = passive \wedge unack_p = 0)$$

算法的目标是，对于时刻t中的某一点，检查是否所有进程在时刻t处于静止状态。由此得t时刻的终止性。这可以通过一次波动完成。要求每个进程确认，在那个时刻或以后，进程是静止的。不静止的进程对于波动中的消息不做响应。这就有效地停止了波动。

与8.3节的方法不同，波动对于进程p所做的访问，并不影响进程p中用于终止检测的变量，（波动的访问可能影响波动算法中的变量，并且，如果使用Lamport逻辑时钟，也会影响进程时钟。）因此，算法的正确操作不会受到几个并发执行波动的影响。

Rana算法是集中式的。每个进程只执行同一个检测算法。如果在8.3.4小节的算法中利用分散式波动算法，也可得到分散式算法。在Rana算法中，进程可以开始各自的波动，这些波动并发执行。

当变成静止时，进程p存储静止时刻的时间qt_p，并开始一次波动，检查自qt_p时刻以来，所有进程是否已经静止。如果是，则检测出终止。否则，将会有一个稍后会变成静止的进程，并开始新一轮的波动。图8-10所示的算法利用Lamport时钟，并将环算法作为波动，实现了这个原理。

```
var state_p   : (active, passive) ;
    θ_p       : integer init 0 ;   (* Logical clock *)
    unack_p   : integer init 0 ;   (* Number of unacknowledged messages *)
    qt_p      : integer init 0 ;   (* Time of most recent transition to quiet *)

S_p: { state_p = active }
    begin θ_p := θ_p + 1 ; send ⟨mes, θ_p⟩ ; unack_p := unack_p + 1 end

R_p: { A message ⟨mes, θ⟩ from q has arrived at p }
    begin receive ⟨mes, θ⟩ ; θ_p := max(θ_p, θ) + 1 ;
          send ⟨ack, θ_p⟩ to q ; state_p := active
    end

I_p: { state_p = active }
    begin θ_p := θ_p + 1 ; state_p := passive ;
          if unack_p = 0 then (* p becomes quiet *)
            begin qt_p := θ_p ; send ⟨tok, θ_p, qt_p, p⟩ to Next_p end
    end

A_p: { An acknowledgement ⟨ack, θ⟩ has arrived at p }
    begin receive ⟨ack, θ⟩ ; θ_p := max(θ_p, θ) + 1 ;
          unack_p := unack_p − 1 ;
          if unack_p = 0 and state_p = passive then (* p becomes quiet *)
            begin qt_p := θ_p ; send ⟨tok, θ_p, qt_p, p⟩ to Next_p end
    end

T_p: { A token ⟨tok, θ, qt, q⟩ has arrived at p }
    begin receive ⟨tok, θ, qt, q⟩ ; θ_p := max(θ_p, θ) + 1 ;
          if quiet(p) then
            if p = q then Announce
            else if qt ⩾ qt_p then send ⟨tok, θ_p, qt, q⟩ to Next_p
    end
```

图8-10 Rana算法

定理8.12 Rana算法（图8-10所示的算法）是正确的终止检测算法。

证明。为了证明算法的活动性，假设**term**在配置γ中成立，其中a个确认仍在传输中。从那时起，只有行为$\mathbf{A}_p$和$\mathbf{T}_p$出现。因为每个行为$\mathbf{A}_p$使传输中的$<\mathbf{ack}, \theta>$消息数减1。这种行为只能出现有限步。每个进程变成静止至多一次。因此，一个令牌至多产生N次，每个令牌至多被传送N次。所以在$a + N^2$步内，终止检测算法达到终止配置δ。在配置中，**term**仍然成立。

设p_0是δ中qt值最大的进程，即，在终止配置中，对于每个进程p，有$qt_{p_0} \geqslant qt_p$。当p_0在最后qt_{p_0}时刻变成静止时，它发出令牌<**tok**，qt_{p_0}，qt_{p_0}，p_0>。这个令牌沿着环一路上被传递，最后返回到p_0。当它接到这个令牌时，每个进程p一直保持静止，且满足$qt_p < qt_{p_0}$。如果不是这样，一旦接到令牌并且迟于p_0变成静止，p就会将它的时钟设置到大于qt_{p_0}的一个值，这与p_0的选择矛盾。当令牌回到p_0时，p_0仍然静止，因此可以调用Announce。

为了证明算法的安全性，假设进程p_0调用Announce。当p_0为静止，且收回它的令牌<**tok**，θ，qt，p_0>时，就会出现这种调用情况。这个令牌已被所有进程转发。接着用反证法证明。假设当p_0检测终止性时，**term**并不成立，这蕴含着，存在一个进程p是不静止的。在这种情况下，在转发p_0的令牌之后，p变成不静止的。事实上，当p转发令牌时，它还是静止的。设q是转发令牌<**tok**，θ，qt，p_0 >后，第一个变成不静止的进程。这蕴含着q一旦接到某个进程，如，r的消息，就被激活，而进程r还未转发p_0的令牌。（否则r在转发令牌后，就会变成不静止的，但r的这个状态在q变成不静止状态之前，这与假设矛盾。）

298 ~ 304

转发令牌之后，$\theta_q > qt_{p_0}$依然成立。这蕴含着，对q为不静止的消息的确认，携带时戳$\theta_0 > qt_{p_0}$，而此消息发送给r。因此，当r接到这个确认消息，变成静止时，$\theta_r > qt_{p_0}$成立。于是，当r接到令牌时，$qt_r > qt_{p_0}$成立。按照算法，r并不转发令牌。与假设矛盾。

Van Wezel[WT94]利用不变式和范函数，给出了算法正确性证明。Huang[Hua88]给出了不依赖于环拓扑结构的算法。

习题

8.1节

8.1　刻画附录A-2所示的算法的活动状态和被动状态。这些状态在附录A-1所示的算法中的什么地方可以找到？

8.2节

定义终止-检测算法的时间复杂度（time complexity）为基本计算终止和调用Announce之间的最差情况单位时间数（定义6.31的理想假设之下）。

8.2　试求Dijkstra-Scholten算法的时间复杂度。

8.3　将Shavit-Francez算法应用到具有惟一标识的任意网上，并保持较低的控制消息开销，且用Gallager-Humblet-Spira算法作为波动算法。检测的时间复杂度为$\Omega(N \log N)$。

你能以交换$O(N)$条额外控制消息作为代价，将算法的时间复杂度改进到$O(N)$吗？

8.3节

8.4　如果p_j被p_i激活，且$j < t$，或者$i > t$，在Dijkstra-Feijen-Van Gasteren算法的结果中，为什么谓词P_0不会为假？

305

8.5　证明，对于每个m，存在交换m条消息的基本计算，使得Dijkstra-Feijen-Van Gasteren算法交换$m \cdot (N-1)$条控制消息。

8.4节

8.6　在图8-9所示的算法中，需做那些修改，实现信用-恢复算法中的规则5a，而非规则

5b?

8.7 在*Rana*算法中，假设进程具有标识。现在假设进程匿名，但是进程具有给环中后继发送消息的方法，且已知进程个数。修改图8-10所示的算法，使其在新的假设下有效。

8.8 从算法的一个不变式，证明*Rana*算法（图8-10所示的算法）的正确性。 306

8.9 重写*Rana*算法（图8-10所示的算法），使其适合用任意波动算法进行通信，而不用环网算法进行通信。

第9章 匿名网络

在前面各章中，为了区别分布式系统中的进程，假设进程具有惟一标识。在大多数现有系统中，这种假设是有效的。进程的标识常常作为地址，用于向进程发送消息。但是为以后所用，识别进程也可经由间接定址（indirect addressing，参见2.4.4节），每个进程用已知局部信道名识别它的近邻。

第7章中讨论了标识的另一种用途，即用进程标识打破进程之间的对称性。可在简单算法中，像Chang和Robert算法（图7-3所示的算法），当两个进程p和q初始化算法时，找到这种情形的典型例子。在这个计算中，p接收q的令牌，q接收p的令牌，如果对进程标识排序不可用，这种情况就是完全对称的。通过排序，就可以打破这种对称性，如两个进程中较小进程生存，较大者被击败。在第7章的其他一些算法中，可以找到更多依靠标识计算打破对称性的复杂例子。

全局惟一标识还可用于检测终止性，例如在Finn算法（图6-9所示的算法）中。进程对它间接地收到的所有进程信息进行检测，判断是否两个名字集一致，其中的名字集合*Inc*包括*NInc*中（参见6.2.6节）进程的所有近邻名字。在Gallager-Humblet-Spira算法中，以更复杂的方式类拟地利用了进程的标识。

本章对匿名网络的能力与命名网络的能力作了比较。在命名网络上可解的哪一类问题也可以在匿名网络上求解？因为即使大多数分布式系统的确为进程提供惟一名字，从理论上来看，这个问题仍然主要是一个相对的问题。但实际上，当把价格低廉（例如：嵌入）的设备组合成网络时，就可能遇到匿名网络。这里我们用Lego MindStorm玩具的供应作为一个例子：多个控制器可被装配成Lego模型，并进行通信，但是控制器是系列生产的芯片，都相同。首先，本章中的结果表明，如果实际中不用进程名，许多问题得不到解决。也许，出乎意料的例子是同步消息传递的实现（参见9.1.4节）。第二，结果蕴含着，如果要用相同的处理器，如Transputer或者Lego MindStorm，构造网络，就必须满足一定的条件。组件必须带有随机数产生器，以便执行概率算法，并且进程必须知道网络规模，或者必须应用集中式初始网络的过程。确实，MindStorm控制器提供随机数。

本章表明，可能打破匿名网络的对称性，但却不可能检测出终止性，除非知道网络规模。利用*概率算法*（probabilistic algorithm）打破对称性是可能的。在这种算法中，进程反复地“投掷硬币”，直到产生不同结果。当发生这种情况时，就打破了对称性。不用说，比较和重复投掷硬币的分布式算法证明是比对原理（参见9.3节）的简洁陈述更复杂。如果网络规模已知，且进程数与计数结果相一致，就可能进行终止检测，并把结果与网络规模进行比较，如果相等，就可进行终止检测。通过投掷硬币不可能进行终止检测，这是因为当投掷时，进程以很小概率事件得出相等结果，其后就会错误地结束终止（参见9.4节）。

如前所述，概率算法可以被广泛地应用于匿名网络以打破对称性，9.1节形式化地引入概率算法。由此产生了几种概率上的正确性表示法，该正确性表示法也在9.1节定义。除了在匿名网络中的重要性，概率算法在分布式计算的其他领域也令人感兴趣。它们在获取分布式系

统中的故障弹性方面也起着重要作用，正如将在第14章看到的。对于某些由确定算法解决的
308 问题，概率解决方法具有较低的复杂度，但在本书中对这个问题不做进一步讨论。

9.1 预备知识

匿名网络是这样一种网络，其中进程的惟一标识不能用于区分进程。同时假设领导人不是优先可用的。因此，具有相同度的所有进程相同。Angluin[Ang80]提出了处理器线（line of processor）这一术语，用以表示进程P_1，P_2,…的无限序列。其中P_i的度为i（即，与i个信道相连）。术语对称系统（symmetric system）也用来表示所有进程都相同（即对称性）的网络。

9.1.1 定义

本小节，对第2章中提出的分布式系统模型进行了扩充，使其包括随机算法。随机化是指利用“电子硬币投掷”和随机数生成器获得进程的随机行为，其中每种行为以已知的概率发生。随机化不同于非确定性，非确定性是由于相对速度的不可预见性，以及进程通过不同事件继续执行的可能性所导致的。它是一种在系统计算集合上引入了概率分布的编程工具，允许我们以一定的概率来阐述算法的性质。固有的非确定性是不可控制的，因此，算法设计者总是要处理由于固有的非确定性引起的所有选择的最坏情况。

1. 概率算法

概率进程可模型化为一个进程，该进程以投掷硬币决定执行每一步。可应用步的集合不仅依赖于进程状态，而且还依赖于进程中前一步的投掷硬币的结果。进程的第一步依赖于初始状态中所作的投掷。

常常称非概率进程（或算法）为确定性进程（deterministic process）（或算法）。但是这个术语并没有排除这种可能性，即非概率进程网络以非确定性方式执行。按照第2章的模型，这样的网络行为具有非确定性，但是其中的选择取决于计算中事件的调度安排，对于这些选择，也不可能进行概率分析。通过随机化引入的选择可进行概率分析，将概率分布引入系统的计
309 算中。为使分布显式，在形式模型中，假设进程执行中产生的整个投掷序列在执行之前给出。

为了理解形式定义，我们首先考虑仅有内部事件的进程。用4-元组$p = (Z, I, \vdash^0, \vdash^1)$定义概率进程（probabilistic process），其中Z和I的定义同定义2.4，$\vdash^0$和$\vdash^1$表示Z上的二元关系，如果第i步硬币投掷为0，则进程第i步必须按照$\vdash^0$执行，否则按照$\vdash^1$执行。设$\rho = (\rho_0, \rho_1, \ldots)$是一个由0和1组成的硬币投掷的无限序列。$p$的$\rho$-执行（$\rho$-execution）是一个状态$c_0, c_1, \ldots$的最大序列，满足$c_0 \in I$，$(c_i, c_{i+1}) \in \vdash \rho^i$。观察可见，固有的非确定性依然存在，因为考虑到硬币投掷结果，可能产生不同事件序列。然而，对于以后的选择不做概率上的假设。关于概率选择，可以做这样的假设，对于每个$k > 0$，每个k位的序列以等概率（即，2^{-k}）出现作为ρ的前缀。给定序列ρ，概率进程的行为就像一个确定进程，可看作概率进程的一个特例。

为了引入进程之间的通信，定义进程为状态集、初始状态集和六个事件关系组成的8-元组，其中六个事件关系为发送事件、接收事件和内部事件，每种事件表示硬币投掷为0，硬币投掷1的两个事件。分布式概率算法（probabilistic distributed algorithm）可表示为概率进程的集合$\mathbb{P}$。这里并没有给出它的完整形式定义，而是指出它和2.1.2节中的定义的区别。如

果算法利用异步消息传递，分布式概率算法的配置每个进程的状态和一个整数以及消息集组成。整数用于对每个进程执行的事件数计数。系统的转移如2.1.2节中所述，另外与p有关的每次转移都增加一个p的步数的计数。

如果ρ是对进程p的由0和1组成的无限序列ρ_p的赋值。系统的ρ-计算（ρ-computation）是一次计算，在这个计算中，涉及进程p的第i次转移就在由ρ_p中的第i个元素所确定的事件关系中。给定序列的集合ρ，概率算法的行为就像一个确定的分布式算法，可看作分布式概率算法 310
的一个特例。对于每个$k>0$，假设k位的N个序列以等概率（即，2^{-kN}）出现作为序列ρ的前缀。由假设可得，在概率算法的一个实例中，每个进程以概率1执行一个不同的局部算法。

2. 概率算法的正确性和复杂度

本章我们用*后置条件*（postcondition ψ）获得问题的正确性。算法的目标是达到终止配置，在这个配置中，后置条件ψ得到满足。后置条件与所考虑的问题有关。对于选举问题，ψ就是“一个进程处于*领导人状态*，其他进程处于*失败状态*”，而对于计算网络规模的算法，ψ就是“对于每个p，$size_p$等于N”。终止配置或者是消息终止（进程都在等待接收消息），或者是进程终止（进程处于终止状态）。

对于确定算法，它的正确性表示如同2.2节。如果在每次计算中，算法达到终止配置，则称确定算法是*终止的*。如果确定算法的每个终止配置满足ψ，则称确定算法是*部分正确的*（对于后置条件ψ）。如果一个确定算法既是终止的又是部分正确的，则称算法是*正确的*。

对于概率算法，通过对ρ赋值所得实例的正确性可以定义算法的正确性。称概率算法是*ρ-终止的*（ρ-terminating），如果在每次ρ-计算中，概率算法达到终止配置。称概率算法是*ρ-部分正确的*（ρ-partially correct，对于后置条件ψ），如果ρ-计算的每个终止配置满足ψ。称概率算法是*ρ-正确的*（ρ-correct），如果一个概率算法既是ρ-终止的又是ρ-部分正确的。

如果对于每个ρ，概率算法是ρ-终止的，则概率算法是*终止的*。如果对于每个ρ，概率算法是ρ-部分正确的，则概率算法是*部分正确的*。如果对于每个ρ，概率算法是ρ-正确的，则概率算法是*正确的*。

如果算法是ρ-终止的概率至少为P，概率算法*以概率P终止*（terminating with probability P）。如果算法是ρ-部分正确的概率至少为P，概率算法*以概率P部分正确*（partially correct with probability P）。如果算法是ρ-正确的概率至少为P，*概率算法以概率P正确*（correct with probability P）。

以概率1终止（部分正确，或正确）的算法并不等价于终止性（部分正确，或正确）。对于非空赋值集，其中ρ在那个集合中的概率为0，概率算法则不适用。

概率算法的*ρ-消息复杂度*（ρ-message complexity）是指任何一次ρ-计算中，交换消息的 311
最大数。概率算法*平均消息复杂度*（average message complexity）是指ρ-消息复杂度的期望值。类似地可以定义概率算法的时间复杂度和位复杂度。

3. 证明否定性结果

本章证明一些否定性结果，即证明不存在达到某一后置条件的算法。对于确定算法，容易证明这样的结果。我们可以构造一个计算，它不终止，但配置满足ψ。定理9.5中可以找到这类证明的典型例子。

某一个无限计算的构造并不能证明概率算法的不存在性。这样一个算法可能有无限次计算，仍然以概率1终止。为了证明概率解的不可能性，假定有一个从不同初始配置开始的正

确计算，由该计算可以构造一个不正确的计算。定理9.12和定理9.13中可以找到这种证明的例子。

9.1.2 概率算法的分类

按照算法能够确保的正确性程度，概率算法可分为（拉斯维加斯）Las Vegas算法、（蒙特卡罗）Monte Carlo算法和（舍伍德）Sherwood算法。概率算法可能是正确的（即，部分正确和终止的），或者按一定概率是正确的。如在9.1.1小节中的解释。

1. Sherwood算法

正确的概率算法称为Sherwood算法[BB88]。因为我们只关心问题的可计算性（主要不是关注它的复杂度），Sherwood算法是令人最少感兴趣的算法，可由下面看出。

定理9.1　*如果存在达到后置条件ψ的正确概率算法，那么存在一个达到ψ的正确确定算法。*

*证明。*设*PA*是正确的概率算法。对于每个ρ，*PA*是ρ-正确的。尤其是对于每个进程被赋值为0的情况；设$\rho^{(0)}$是这个赋值。

312 考虑算法*DA*，当每次硬币投掷产生值0时，它的行为如同*PA*（即在程序中，用0代替每次对硬币投掷的调用）。算法是确定性的，而且由于*PA*是$\rho^{(0)}$-正确的，所以算法是正确的。 □

这个结果并不蕴含着Sherwood算法是无用的。它们的平均复杂度可与最好的确定算法的（最坏情况）复杂度相比。作为一个例子，考虑Chang-Roberts选举算法（图7-3所示的算法），它在最坏情况下的消息复杂度为$\Theta(N^2)$，但在平均情况消息复杂度为$\Theta(N \log N)$。在随机版本的算法中，每个进程通过随机选出的位串扩展它的名字，这个随机选出的部分在比较中是最重要的。不管初始标识的分布是什么，改进算法的期望复杂度为原始算法平均情况复杂度，即$\Theta(N \log N)$条消息。

本书将不讨论Sherwood算法。

2. Las Vegas算法和Monte Carlo算法

作为定理9.1的结论，对于不能用确定算法解决的问题，最好的算法是按概率$P(0 < P < 1)$是正确的概率算法。通常，每次执行中，或者满足终止性，或者满足部分正确性。

定义9.2　Las Vegas算法是概率算法

（1）该算法以正概率终止，且

（2）是部分正确的。

Monte Carlo算法是概率算法

（1）该算法是终止的，且

（2）以正概率部分正确。

Las Vegas算法的终止概率常常为1。只有当无限次选择随机位的不利模式时，才存在无限次执行的情况。实际上，这意味着，Las Vegas算法总是终止的，尽管只能给出期望的算法执行步数。却不能给出算法执行步数的上界。一个Las Vegas算法的例子是由Itai和Rodeh（图9-4所示的算法）提出的选举算法。

Monte Carlo算法（如果它不是Sherwood算法）的失败概率总是小于1。在一个错误的有限次的执行中，只能选举有限多个随机位，因此执行的概率是正的。通常可以给出Monte
313 Carlo算法执行步数的上界。一个Monte Carlo算法的例子是由Itai和Rodeh提出的计算环规模的

算法（图9-5所示的算法）。

在某些附加约束条件下，可能从Monte Carlo算法得到Las Vegas算法，反之亦然。参见习题9.1和9.2。

3. 相关方面的概述

本章的算法按照以下4个准则分类；并简要说明，哪一类算法被认为更具吸引力。

（1）网络规模已知或未知。不依赖于知识N的算法更具吸引力，因为它们更具一般性。

（2）进程-终止或消息-终止。进程终止算法更具吸引力，因为进程显式终止，因此进程可以利用计算所得的结果。

（3）确定性的或概率的。确定性算法更具吸引力，因为，不需要随机数生成器，其正确性性质比Las Vegas算法和Monte Carlo算法所满足的正确性性质更强。

（4）Las Vegas算法或Monte Carlo算法。通常认为Las Vegas算法更具吸引力，因为在实际中，它们的概率终止性与终止性几乎难以区分。

9.1.3 本章考虑的问题

提出匿名网络上计算的完整理论，包括从匿名网络中得出的大量结果，不是我们的目的。当考虑与选举、函数计算有关的问题，包括计算网络规模时，我们的意图是研究这些网络的计算能力。

这些问题是最基本的。如果能选出领导人，匿名网络就和命名网络具有相同能力。因为用集中式算法对名字进行赋值，可如下进行。在9.3节中，如果网络规模已知，可以通过概率算法选出领导人，这表明计算网络规模的重要性。因为用集中式算法很容易计算出网络规模（正如以下所示），选举和计算网络规模就成了等价的问题。我们将要看到，网络规模不能按概率计算出来，并且，如果网络规模未知，选举问题也是不可解的。 314

集中式计算 如果惟一初始进程（领导人）可用，通过回波算法的一次计算，即可容易地计算出网络规模。通过计算中导出的生成树，每条向上发送的消息报告发送进程子树中的进程数。参见图9-1所示的算法。

为了赋惟一名字，首先用图9-1所示的算法计算网络规模，然后在进程之间划分区间$\{0, \cdots, N-1\}$。每棵子树接收一个区间，区间长度对应于子树规模。

定理9.3 *存在计算网络规模的确定性集中式算法，交换$2|E|$条消息，时间复杂度为$O(N)$。存在赋值惟一名字的确定性集中式算法，交换$2|E| + (N-1)$条消息，时间复杂度为$O(N)$。*

存在更有效的命名赋值算法。参见习题9.3。定理9.3的结果表明，只用很少的开销，领导人的可用性就能补偿缺少惟一命名的问题。因此，在匿名网络的研究中，假设除了没有名字以外，领导人也不可用。

9.1.4 同步消息传递与异步消息传递

匿名团上与选举相关的结果对于通信方式是同步还是异步消息传递很敏感。在同步消息传递下，系统的一次转移中的两个进程合作，其中两个进程间的对称性被打破。

定理9.4 *对于通过同步消息传递进行通信的两个进程组成的网络，存在确定性的、进程*

终止的选举算法。

证明。每个进程有三种状态，即*sleep*状态、*leader*状态和*lost*状态。初始状态是*sleep*状态。每个进程或者发送并进入终止状态*leader*，或者接收并进入终止状态*lost*。

```
var rec_p      : integer  init 0 ; (* Counts number of received messages *)
    father_p   : ℙ        init udef ;
    size_p     : integer  init 1 ; (* Size of subtree of p *)

For the initiator:
    begin forall q ∈ Neigh_p do send ⟨tok, 0⟩ to q ;
          while rec_p < #Neigh_p do
                begin receive ⟨tok, s⟩ ; rec_p := rec_p + 1 ;
                      size_p := size_p + s
                end    (* The network size is size_p *)
    end

For non-initiators:
    begin receive ⟨tok, s⟩ from neighbor q ; father_p := q ; rec_p := rec_p + 1 ;
          forall r ∈ Neigh_p, r ≠ father_p do send ⟨tok, 0⟩ to r ;
          while rec_p < #Neigh_p do
                begin receive ⟨tok, s⟩ ; rec_p := rec_p + 1 ;
                      size_p := size_p + s
                end ;
          send ⟨tok, size_p⟩ to father_p
    end
```

图9-1　网络规模的集中式计算

因此，系统有九种配置，但只有三种是可达的。从初始配置（*sleep*，*sleep*）开始，系统可转移到配置（*leader*，*lost*）和配置（*lost*，*leader*）。这两个配置都是进程终止的，在状态*leader*和状态*lost*，都只包含一个进程。 □

算法来自Angluin[Ang80]，概述了任意规模团上的算法。如果通信是异步消息传递，则该问题不存在确定性算法。可用与定理9.5中将使用的类似方法证明。我们不考虑对称性被通信机制打破的情况，只考虑通信是同步消息传递的系统。

Angluin提供的方法，是特定用于拓扑结构的，例如，不适合于环。本章不可能性的结果将推广到具有任意或环的拓扑结构的同步消息传递的系统上。

如果消息传递是同步的，就一定能打破两个进程之间的对称性。但是如果消息传递是异步的，就不能打破两个进程之间的对称性。结果是不存在确定算法，在异步消息传递的匿名网络中实现同步消息传递。

315 ~ 316

9.2 确定算法

本节介绍一些与确定算法相关的结果。最重要的结果是否定性的，不可能确定性地选出领导人。甚至在环规模已知时也不能选出。同时证明，如果已知环规模，就可进行函数计算；如果环规模未知，就不能计算函数。

9.2.1 确定性的选举：否定性的结果

本小节将要证明，在匿名网上，不存在确定性选举算法。证明利用了对称配置（symme-

tric configuration）的概念。可以证明，存在一次计算，它在对称配置中开始，每N步之后，达到一次对称配置。确定性算法的不可能性，可由以下观察导出：正确算法不能终止于对称配置。

定理9.5 *在已知规模的匿名环上，不存在确定性选举算法。*

证明。用单向环证明这个结果。当理解了这个证明过程时，很容易证明双向环上的情况。设进程p_0至p_{N-1}按照如下方式在环上排列，对于$i < N-1$，p_i只能向p_{i+1}发送消息，p_{N-1}向p_0发送消息。对于环上的一个配置，设c_i表示p_i的状态，M_i表示目的节点为p_i的消息集。

称配置是对称的，如果所有状态相同，且传输到每个进程的消息集相同。即，

$$\forall i, j: (c_i = c_j \wedge M_i = M_j)$$

对于环上的任何算法，构造计算$C = (\gamma_0, \gamma_1, \cdots)$，满足在$C$中的每个配置$\gamma_{kN}$是对称的。设$\gamma_0$是对称初始配置。存在这样的配置，因为每个进程具有相同的初始状态集，初始时，所有M_i都为$\varnothing$。

如果配置γ_{kN}是终止的，则构造完成。计算C终止于对称配置。否则，γ_{kN}是对称配置，其中至少有一个事件是可应用的，称这个事件为p_i中的事件e_i。配置的对称性现在蕴含着，同一个事件在每个进程中是可应用的。定理2.19蕴含着，它可以被所有进程并发地执行，执行N次转移之后，产生对称配置。所有进程从相同状态转移到相同状态。如果在事件中接收到一条 317
消息，就从每个相同消息集M_i中删除该消息；如果发送一条消息，就将它加到每个相同消息集M_i中。因此计算被扩展了N步。此后，导致对称配置$\gamma_{(k+1)N}$。

由此可得，匿名环的每个算法的计算，或者无限，或者结束于对称终止配置。在对称配置中，或者所有进程处于状态*leader*，或者一个也没有。因此在对称配置中，不存在只有一个进程处于*leader*状态。对于确定性选举算法，所有计算结束于终止配置，但只有一个进程处于*leader*状态，因此这样的算法不存在。 □

定理9.5蕴含对于规模未知的任意网络或者环中的更一般的选举问题，不存在确定算法解。

9.2.2 环上函数计算

选举的不可能性提出了这样一个问题，当选举不可能时，是否存在更“容易的”可解问题。在本小节中，考虑有向匿名环上的函数计算问题。假设进程p_i输入为$x_i \in X$，设SX表示X中所有元素序列集。SX上的函数f是周期的(cyclic)，如果在输入的周期移动中它是不变的。即，对于所有x的周期移动y，$f(y) = f(x)$。周期函数的例子有：求和、求整数的最小值或最大值、布尔函数的合取、析取和不可兼或、输入的长度等等。假设进程p有变量$result_p$，计算f的算法的后置条件为“$result_p = f(x_0, \ldots, x_{N-1})$”。

本小节的结果表明了环规模知识以及区别消息终止和进程终止的重要性。

1. 已知环规模

如果已知环规模，可以收集每个进程中的所有输入，用确定算法就可以计算出每个周期函数。它的解称为输入集（input collection）。输入集的代价为$N(N-1)$条消息，这也是用确定算法计算某些函数的下界。

定理9.6 *如果已知环规模，用进程-终止的确定算法计算周期函数的消息复杂度为$N(N-1)$。* 318

证明。为使叙述简明，假设信道是fifo的。算法按$N-1$步发送消息。在第一步中，每个进

程向其顺时针方向的近邻发送自己的输入，并从其逆时针方向的近邻接收输入。在下一步中，每个进程将它在前一轮中所接收的值发送给其顺时针方向的近邻，并从其逆时针方向的近邻处接收新值。由于在每一轮中，每个输入被进一步传递一个进程，在第i步中，进程接收来自它的第i个逆时针方向近邻的输入，$N-1$步后，收集完所有输入（按照它们在环上出现的次序）。通信轮完成之后，每个进程局部计算f的值，并终止。 □

定理9.7　*任何计算AND、OR和SUM的确定进程-终止算法，在最坏情况下至少交换$N(N-1)$条消息。*

证明。设给定解决这些问题之一的确定的进程-终止算法A。类似于7.2.3小节中的定义，消息的轨迹（trace）定义如下。如果p在接收消息之前发送消息，则这条消息的轨迹为(p)。如果到目前为止p已经接收的消息的最长轨迹为$(p_1, \cdots, p_k)$，则发送消息m，m的轨迹为$(p_1, \cdots, p_k, p)$。

定理中提到的三个问题具有共性。至少对于一个对称初始配置，在终止之前，进程必须接收长度为$N-1$的轨迹。对于SUM问题，每一种初始配置，就是这种情况。对于AND问题，存在这样一种情况，即每个进程输入为*true*。而对于OR问题，存在每个进程输入为*false*的情况。为证明必须接收长度为$N-1$的轨迹，考虑AND的计算，其中每个进程输入为*true*，在这种情况下，解为*true*。假设当进程所接到的最长消息轨迹为$k < N-1$时，进程以输出*true*终止。其中的进程因某种原因而终止的事件只取决于$k+1$个进程中的事件，因此它也可以出现在从初始配置开始的计算中，在初始配置中，这k个进程输入为*true*，其他输入为*false*。

如果每个进程处于同一状态且每个信道包含同样的消息集，则称配置是对称的。正如在定理9.5中证明的那样，构造一个计算，使其每N步后，达到对称配置状态，且在计算中，每
319 个进程执行相同的事件集。选举初始配置，使得算法至少发送一条轨迹长为$N-1$的消息。在计算中，对于每个$i \leqslant N-1$，算法至少发送一条轨迹长为i的消息。但是，因为每个进程执行相同的事件集，所发送的轨迹长为i的消息数是N的倍数。因此至少为N。于是消息数至少为$N \cdot (N-1)$。 □

2. 环规模未知

如果环规模未知，利用进程-终止的确定算法就不能计算出非-常函数。结果的证明利用了重现算法在环上各个部分进行（正确）计算的技术。如果f是常函数（即对于每个x，$f(x) = c$），它的计算可以利用一种算法简单地进行，该算法可以立即地终止于每个进程p，且$result_p = c$，否定性的结果只关心非-常函数。

定理9.8　*如果环规模未知，不存在计算非-常函数的确定进程-终止算法。*

证明。因为f是非-常函数，存在输入$x = (x_0, \ldots, x_{k-1})$和$y = (y_0, \ldots, y_{l-1})$，满足$f(x) \neq f(y)$，不妨设为$f(x) = a$，$f(y) = b$。设$A$是一个确定的、进程-终止的算法，满足在输入为$x$的环上，存在计算$C_x$，在计算中存在进程$p$以$result_p = a$终止，且在输入为$y$的环上，存在计算$C_y$，在计算中存在进程$q$以$result_q = b$终止。

设在C_x中，p所接收的消息的最长轨迹为K，可能$K > k$，因为在p终止之前，A可能沿着环循环几次消息。考虑更大的环，该环包括一个有$K+1$个进程的段S，具有以下性质。已知S的最后一个进程（即在段的顺时针方向的最后进程）与计算C_x中进程p的输入相同，称此进程为p'，已知p'的第i个逆时针近邻的输入与计算C_x中进程p的第i个逆时针近邻的输入相同。S中的进程与环中输入为x的进程的对应，如图9-2所示。

当在包含段S的环上执行时，算法A的计算具有以下性质。对于每个$i \leqslant K$，进程p'的第i个逆时针近邻发送所有轨迹长$\leqslant K+1-i$的消息，这些消息也是由计算C_x中进程p的第i个逆时针近邻发送的。这可以对K用向下归纳法证明。

情形$i=K$：进程p'的第K个逆时针近邻与C_x中进程p的第K个逆时针近邻具有相同初始状态，因此存在计算，在该计算中，发送相同的消息集，消息轨迹长度为1。 320

归纳步：假设在某些计算中，进程p'的第（$i+1$）个逆时针近邻发送轨迹长$\leqslant K-i$的所有消息，这些消息是由计算C_x中进程p的第i个逆时针近邻发送的。在这个计算中，进程p'的第i个逆时针近邻与C_x中进程p的第i个逆时针近邻具有相同初始状态，并且接收由轨迹长$\leqslant K-i$的消息组成的相同消息集。因此存在计算，其中所发送的由轨迹长度$\leqslant K+1-i$的消息组成的消息集与C_x中进程p的第i个逆时针近邻发送的消息集相同。

在所构造的计算中，p'接收的消息集与C_x中进程p接收的消息集相同，且p'与p有相同初始状态。因此存在计算，在这个计算中，p'以$result_{p'}=a$终止。

所构造的计算“欺骗”进程p'以结果$result_{p'}=f(x)$进入终止状态，而实际上，输入是一个不同于x的更长序列x'。如果是$f(x')=f(x)$的情况，计算仍会给出正确结果。为了产生矛盾结果，考虑包含该段的环，在该段上，进程模拟计算C_y，进程q'以$result_{q'}=b$终止。对于这个环，存在计算，其中的两个进程终止时，具有不同$result$值，因为至多一个结果是正确的，因此矛盾。 □

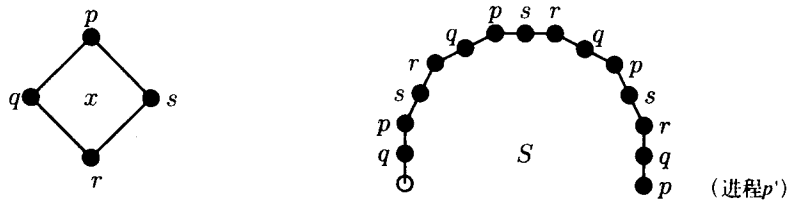


图9-2 不正确计算的构造

定理9.9 *在规模未知的环上，存在计算OR和AND的确定消息-终止算法，至多利用N条消息。*

证明。如果至少一个进程的输入为*true*，则OR的值为*true*。如果所有进程的输入都为*false*，则它的值为*false*。一旦初始化，进程p进行赋值$result_p := x_p$，且如果x_p为*true*，则向其顺时针近邻发送<yes>消息。当进程p接到这个消息时，如果$result_p$的值为*true*，则忽略这条消息；否则，将$result_p$的值设为*true*，并转发这条消息，参见图9-3所示的算法。 321

首先考虑每个进程输入为*false*的情况。每个进程对$result_p$赋值为*false*，并开始等待一条消息。由于没有进程发送消息，其中的每个进程等待的配置是终止的，在这种情况下，$result_p$包含正确解（*false*）。

其次考虑有一些进程其输入为*true*的情形，输入为*true*的进程p为$result_p$赋值*true*，并把<yes>消息发送到下一进程。由于每个进程至多发送一条消息（参见图9-3所示的算法），在这种情况下，也达到终止配置。如果在终止配置中，$result_p=true$，那么$result_{Next_p}=ture$，因为p所发送的消息被$Next_p$接收。由于至少存在一个进程p，使得$result_p=true$，因此，对于每个进程p，$result_p=true$，在这种情况下，算法在每个进程中终止，且有正确解。

AND的计算类似，但当所有进程为*true*时，才为真，否则为*false*。 □

```
var result_p  : boolean ;   (* Result of the computation *)
    x_p       : boolean ;   (* Input of p *)

begin result_p := x_p ;
      if x_p is true then send ⟨yes⟩ to Next_p ;
      receive ⟨yes⟩ ;
      if result_p is false then
        begin result_p := true ; send ⟨yes⟩ to Next_p end
end
```

图9-3　确定性的OR计算

3. 已知环规模的部分知识

作为在已知环规模和未知环规模之间的折衷，我们考虑环规模N未知，但是已知它在一定界限之间的情况。通过修改本节所做证明，可以很容易地证明以下结果。

（1）不存在计算SUM的确定算法，对于两个不同规模的环都是正确的。

322 （2）不存在计算XOR的确定算法，对于偶数环和基数环都是正确的。

（3）如果已知环规模的上界S，利用$N\cdot(S-1)$条消息，就可确定地计算出AND和OR。

4. 其他拓扑结构

对于其他类型的网络拓扑结构，也可给出重现证明技术。但要求能在较大的图上几次“展开”（unrolled）较小的图。从一个图到另一个图所要求的映射类型称为一个*覆盖*（cover）。

我们也可给出其他拓扑结构的匿名网络上函数计算的算法。Beame和Bodlaender[BB89]已经证明，在$n\times n$的圆环上进行确定输入收集的消息复杂度为$O(n^3)$。对于某些函数计算，包括AND和OR，计算的下界是$\Omega(N^3)$。Kranakis和Krizanc证明，在n-维超立方体上进行输入收集的位复杂度为$O(2^n\cdot n^4)$ [KK90]。

9.3 概率选举算法

如果已知网络规模，可以给出以概率1终止的部分正确的选举算法。即算法属于Las Vegas类型。由于不存在确定性的算法（参见定理9.5），Las Vegas算法是最好的选举算法。Itai和Rodeh[IR81]提出了已知规模的匿名环网络上的选举算法。算法依赖于Chang-Roberts算法（图7.3所示的算法）中使用的原理，但要使用该算法，必需对算法作某些修改。

首先，因为Chang-Roberts算法依赖标识信息的可用性，任何起着初始进程作用的（匿名）进程p，随机地从集合$\{1,\cdots,N\}$选择一个标识id_p。这使得比较不同进程的令牌成为可能，但是也带来一些新的困难，这是由于几个进程可能具有同一标识。在Chang-Roberts算法中，当令牌中的标识与进程的标识一致时，进程知道它已经接到自己的令牌，这是进程变成*leader*的充分条件。

因为几个进程可能选择同一标识，进程利用令牌中的跳数计数器，识别对自己令牌的接收。当跳数等于N时，进程p接到自己的令牌。令牌的接收蕴含，没有进程已经选择更小标识。

323 但这不是变成*leader*的充分条件。因为可能存在另一个进程也具有同样的标识。为了检测出这种情况，每个令牌还有一个布尔值un，当令牌产生时，它的值为真；如果它被一个具有相同

标识的进程转发时，设它的值为假。当进程接到自己的令牌且$un = true$时，它变成*leader*。

当进程接到自己的标识且$un = false$时，出现困难问题。这表明进程的确有最小标识，但是存在另一具有相同标识的进程。具有最小标识的所有进程可能接到它们自己的令牌，并在Chang-Roberts算法中“取胜”。如果出现这种情况，Chang-Roberts算法中的每个“获胜者”，会再次初始化Chang-Roberts算法，在更高一级开始新一轮计算。为了防止另一次的不分胜负，在新一级，初始化下一级的每个进程必须选择新的标识。令牌中也标明这个级别，某些级的令牌可以中止所有更小一级的活动。

总之，如果进程p接收到自己令牌时，仍处于*cand*状态，则它在chang-Roberts算法中获胜。如果p是惟一的获胜者，则它变成*leader*，否则p在下一级初始化Chang-Roberts算法。如果进程p接到具有较小标识的更高一级的令牌或同级令牌，则进程p变成*lost*；令牌被转发。具有相同级别和标识的令牌被转发，但设置un为假。较低级或同级更大标识的令牌，被清除(即中止它的转发)。参见图9-4所示的算法。我们将继续证明算法是部分正确的，且以概率1终止。

```
var state_p  : (sleep, cand, leader, lost)   init sleep ;
    level_p  : integer                       init 0 ;
    id_p     : integer ;
    stop_p   : boolean                       init false ;

begin if p is initiator then
        begin level_p := 1 ; state_p := cand ; id_p := rand({1, ..., N}) ;
              send ⟨tok, level_p, id_p, 1, true⟩ to Next_p
        end ;
      while not stop_p do
        begin receive a message ; (* Either a token or a ready message *)
              if it is a token ⟨tok, level, id, hops, un⟩ then
                if hops = N and state_p = cand then
                  if un then (* Become elected *)
                    begin state_p := leader ; send ⟨ready⟩ to Next_p ;
                          receive ⟨ready⟩ ; stop_p := true
                    end
                  else (* There are more winners for this level *)
                    begin level_p := level_p + 1 ; id_p := rand({1, ..., N}) ;
                          send ⟨tok, level_p, id_p, 1, true⟩ to Next_p
                    end
                else if level > level_p or (level = level_p and id < id_p) then
                       begin level_p := level ; state_p := lost ;
                             send ⟨tok, level, id, hops + 1, un⟩ to Next_p
                       end
                     else if level = level_p and id = id_p then
                            send ⟨tok, level, id, hops + 1, false⟩ to Next_p
                          else skip (* Purge the token *)
              else (* The message is ⟨ready⟩ *)
                  begin send ⟨ready⟩ to Next_p ; stop_p := true end
        end
end
```

图9-4 ITAI-RODEH选举算法

引理9.10 如果在l级，至少有一个进程产生一个令牌，则或者有一个进程在那一级被选

中，且不存在进程在$l+1$级产生令牌；或者至少有一个进程在$l+1$级产生一个令牌。

证明。如果进程p接到自己的令牌（l级），则进程p可以产生$l+1$级的令牌。

假设进程p产生l级的令牌，但它的标识在l级所选的标识中不是最小的。在这种情况下，令牌并不返回到p。如果令牌没有被更高一级的进程清除，它就会被同级但标识较小的进程接收，在那里它会被清除。（观察可见在接到l级的p的令牌之前，所有在l级产生令牌的进程都是这样做的。）由此可得，只有在l级选择最小标识的进程才能在那一级当选，或者在级$l+1$产生令牌。

假设进程p产生l级的令牌，且它的标识在l级所选的标识中是最小的。如果令牌没有被更高一级的进程清除（在这种情况下，至少有一个进程产生$l+1$级的令牌），令牌就会返回到p。出现两种情况：或者p是l级选择最小标识的惟一进程，或者有更多进程选择这个标识。在第一种情况下，令牌返回且$un = true$，p当选。在第二种情况下，令牌返回且$un = false$，产生$l+1$级的一个令牌。 □

324 l 325

定理9.11　Itai-Rodeh算法（图9-4所示的算法）是已知规模环上的选举算法，它是部分正确的，且以概率1终止。

证明。如果算法终止，且只有一个领导人，利用引理9.10可证明它的部分正确性。设l是产生令牌的最高一级。由假设，定义l是因为只考虑有限计算。没有一个进程会在小于l的级当选。在$l+1$级不产生令牌蕴含着，在l级只有一个进程当选。

接着要证明，算法以概率1终止。如果对于某级l，存在$k > 1$的进程，进程在l级产生令牌，存在概率$P_k > 0$，在该级上只有一个进程选择最小标识，在这种情况下，算法终止。设P是P_k中大于k的最小值。如果令牌在l级产生，则令牌也将在$l+1$级产生的概率至多为$(1-P)$。因此，经过l级后，算法不终止的概率为$(1-P)^l$，当l趋于无穷大时，它的极限值为0。 □

文献[IR81]中所作的概率分析，揭示出级的期望值界限为$e \cdot N(N-1)$。每级消息期望值的界限为$O(N \log N)$（可用与定理7.6中使用的类似证明给出），这蕴含着算法的期望消息复杂度为$O(N \log N)$。

任意网　利用非常类似的想法，可给出任意网上的选举算法。该算法是基于图7-9所示的算法，而不是基于Chang-Roberts算法。如果进程已经初始化回波算法，并接到来自近邻的回波，那么进程在算法的某一级“取胜”。在Itai-Rodeh算法中，这不是变成领导人的充分条件，因为有可能具有相同标识的几个进程已经开始选举。为了检测出这种情况，由回波算法所构造的树的规模可以如同图9-1所示的算法计算。如果有几个初始进程具有相同标识，每次对回波算法的调用，都会构造一棵只包含进程子集的树。当进程执行完回波算法（通过接收来自每个近邻的回波），就可获知它所产生的树的规模。如果规模等于N，则进程当选；否则回波算法在更高一级被初始化。

所得算法是部分正确的，且以概率1终止。它的完整描述祥见文献[Tel94]。其他的选举算法参见文献[SS89]和[MA89]。

326

9.4　网络规模计算

本章中讨论的结果表明了网络规模知识的重要性。本节考虑计算匿名环规模的问题。可以用消息-终止算法以及按得到错误解的非零概率来计算环规模。

9.4.1 否定性结果

本小节包含两个与计算环规模相关的不可能性的结果。证明思想与定理9.8的证明相同。即，将给定（假定是正确的）计算在更大环的不同部分重放。对于概率算法，必须证明这样的重放具有非零概率，这可由在一个有限执行中，进程只利用它随机位序列的有限前缀这样一个事实而得出。

定理9.12 *不存在进程-终止算法，以概率r>0正确计算环规模。*

证明。假设A是规模为N的环上的概率算法。有ρ-计算C_N。在C_N中，每个进程p以$result_p = N$终止。选择一个进程p，令C_N中p所接收的消息的最长轨迹长度为K。每个进程在C_N中只能执行有限步。设L是计算C_N中任意进程所执行的最大步数。

考虑一个更大的环，它包含一个由$K+1$个进程形成的段。设p_0是最后一个进程，p_i是p_0的第i个逆时针方向近邻。ρ'是环上进程位序列的赋值，满足以下性质。ρ'_{p_0}的前L位与ρ_p的前L位相同。对于$i < k$，ρ'_{p_i}的前L位与ρ_q的前L位相同。q是在N环上p的第i个逆时针方向近邻。当将段长固定为$K+1$时，这样赋值的概率很小，即$2^{-(K+1)L}$。

在这个更大的环上，存在A的ρ'-计算C，在这个计算中，进程p_i只发送与C_N中p的第i个逆时针方向近邻相同的消息，且消息轨迹长度$\leq K+1-i$。证明同定理9.8。可见，对于任意长为$K+1$的固定段，存在至少为$2^{-(K+1)L}$的概率，在该段中的进程重放计算C_N。在这种情况下，进程 327
p_0以$result_{p_0} = N$终止，这是不正确的。

在一个固定段上出现这种情况的概率是极小的。但是如果环规模很大，在环上某个地方发生的概率也可任意大。设给定概率$r > 0$，选择自然数R，满足$(1-2^{-(K+1)\cdot L})^R < r$且考虑环规模为$R \cdot (K+1)$。这个环上有$R$个不相交且长为$K+1$的段，对于每个这样的段，概率至多为$1-2^{-(K+1)\cdot L}$，它的最后一个进程不能以错误的解终止。由此可得，没有一个进程能够以错误解终止的概率小于r。 □

利用同样的论证可得，事实上，不存在非-常函数，可由消息-终止算法以概率$r > 0$正确计算出。下一个不可能性结果不能推广到任意的非-常函数。结果表明，不存在消息-终止的Las Vegas算法，因此我们所能达到的最好算法是消息-终止的Monte Carlo算法。

定理9.13 *不存在计算环规模的消息-终止算法，算法正确的概率为1。*

证明。假设A是规模为N的环上的概率算法，有ρ-计算C_N。对某些进程p，它以$result_p = N$消息-终止。每个进程在C_N中只能执行有限步。设L是计算C_N中任一进程所执行的最大步数。在规模为N的环上，进程的编号从p_0到p_{N-1}。

考虑规模为$2N$的环，进程编号为从q_0到q_{2N-1}。对位串的赋值ρ'，满足对于所有$i < N$，ρ'_{q_i}和$\rho'_{q_{i+N}}$的前L位与ρ_{p_i}的前L位相同。构造规模为$2N$的环上A的ρ'-计算如下。q_i和q_{i+N}并发地执行C_N中p_i的每一步。因为C_N中的最后一个配置是终止的，因此在C_{2N}中的最后一个计算也是终止的，在这个配置中，至少存在以$result_q = N$的两个进程q，这是不正确的结果。

这样赋值ρ'的概率为Z^{-2NL}，它是正的，这表明A不能以大于$1-2^{-2NL}$的概率正确计算环规模。 □

进程-终止或者消息-终止算法正确性的概率之间的差异非形式地解释如下。如果某一不 328
幸的随机选择出现在环上的某处，则进程-终止算法给出不正确的结果。当环规模变大时，发生这种情况的概率增加。在消息-终止算法中，造成那种不幸选择的进程总是有从它们的不正确解恢复过来的可能性，如果它们接到来自其他进程的信息。因此，如果某一不幸的随机选

择出现在环上的每处，消息-终止的算法会给出不正确的结果。尽管不能永远排除这种情况，但是发生这种情况的概率是有界的，实际上，当环规模变大时，发生这种情况的概率减小。

9.4.2 计算环规模的算法

本小节讨论计算环规模的消息-终止Monte Carlo算法。可以证明，当算法终止时，对于所有p，算法以某一概率满足$est_p = N$，且可以通过选择算法中的参数R，使得该概率任意大。算法与Itai和Rodeh [IR81]给出的算法类似，但稍微有些简化。

进程p维持局部对环规模的估算（estimate），est_p，保证任何时刻这个估算值是守恒的，即$est_p \leq N$。初始时，$est_p = 2$。当p接收的信息蕴含着，est_p不等于环规模时，就增加该估算值。

为了获得正确估算的可靠性，进程p产生一个令牌，并在环上发送est_p次。如果该估算值是正确的，p接收自己的令牌。如果发生这种情况，p将暂时得到满足，并不采取进一步的行动。在N实际上大于est_p时，进程p接收类似的令牌，但是p的第est_p个逆时针近邻具有相同的估算值，并且也产生一个令牌。

可以以大概率检测出这种情况（即进程p接收的不是自己的令牌），每个进程从{1，…，R}中随机选择一个标号，并将它包含在令牌中。只要p的估算值不增加，p的标号保持不变。如果p接到来自另一个进程的令牌，则令牌中所包含的标号不同于p的标号的概率为1-1/R。如果进程接到自己的令牌，则这两个标号一定完全一样。

进程p按照以下两种情况增加它的估算值。

（1）进程接到一个令牌，其所含的$est > est_p$，因为保证所有估算值都是守恒的，该令牌的
329 接收蕴含着$N > est_p$。

（2）当接到估算值$est = est_p$的令牌时，这个令牌已经做过est次跳跃，但是它所包含的标号不同于p的标号。p的第（est）个逆时针近邻产生这个令牌。其后的进程标号不同于p的进程标号。从而，p的第（est）个逆时针近邻不同于p。因此，$N \neq est$。

具有$est < est_p$的令牌被清除（即中止它的传播），参见图9-5。

引理9.14　算法在交换$O(N^3)$条消息后，消息-终止。在终止配置中，所有est_p都相等，它们的共同值的上限为N，即对于所有p和q，$est_p = est_q \leq N$。

证明。证明终止性的关键步骤是证明计算值是守恒的。观察可见，est_p从不减少，仅当est_p增加时，lbl_p才改变。这蕴含着只要p的令牌<**test**，est，lbl，h>循环传递，且$est = est_p$，则$lbl_p = lbl$。利用这一点，可用归纳法证明，计算的所有值是守恒的。

因为令牌中所找到的估算值就是进程的计算值，只需考虑由进程对估算值的计算，它发生在以下三种情形。

（1）进程p一旦接到带有估算值est的令牌，可能将est_p增加到est。但因这个值被较早进行计算，按归纳假设，它是守恒的。

（2）进程p一旦接到带有估算值est，且跳数为est的令牌，可能将est_p增加到est+1。在这种情况下，由归纳法可得，est是守恒的，且$N \neq est$，因为令牌的初始进程跳数为est，不等于p。

（3）最后，进程p一旦接收带有$est = h = est_p$，但是$lbl \neq lbl_p$的令牌，可能将est_p增加到est+1。令牌的初始进程跳数为est步远，在这种情形，$lbl \neq lbl_p$蕴含初始进程不等于p。再次蕴含着$N \neq est$。est的守恒性蕴含着$N > est$。

```
cons R      : integer     ;  (* Determines correctness probability *)

var  est_p  : integer ;
     lbl_p  : integer ;

begin est_p := 2 ; lbl_p := rand({1, ..., R}) ;
      send ⟨test, est_p, lbl_p, 1⟩ to Next_p ;
      while true do
        begin receive ⟨test, est, lbl, h⟩ ;
              if est > est_p then     (* p's estimate must increase *)
                if est = h then     (* est is also too low *)
                  begin est_p := est + 1 ; lbl_p := rand({1, ..., R}) ;
                        send ⟨test, est_p, lbl_p, 1⟩ to Next_p
                  end
                else    (* forward token and increase est_p *)
                  begin send ⟨test, est, lbl, h + 1⟩ to Next_p ;
                        est_p := est ; lbl_p := rand({1, ..., R}) ;
                        send ⟨test, est_p, lbl_p, 1⟩ to Next_p
                  end
              else if est = est_p then
                     if h < est then send ⟨test, est, lbl, h + 1⟩ to Next_p
                     else (* This token has made est hops *)
                          if lbl ≠ lbl_p then
                            begin est_p := est + 1 ;
                                  lbl_p := rand({1, ..., R}) ;
                                  send ⟨test, est_p, lbl_p, 1⟩ to Next_p
                            end
                          else skip     (* Possibly p's token returned *)
                   else (* est < est_p *) skip
        end
end
```

图9-5 概率计算环规模

每个进程产生少于N个不同令牌，即值从2到N。估算值为e的令牌至多被转发e个跳数。这蕴含着消息复杂度为$O(N^3)$。如果进程p使est_p增加到e，p向$Next_p$发送包含值e的令牌。一旦接到这个令牌，est_{Next_p}值至少为e，因此在终止配置中，$est_{Next_p} \geqslant est_p$。这对于所有$p$成立。这蕴含着所有估算值相同。前面已经证明，它们的值小于N。 □

定理9.15 *对于所有p，图9-5所示的算法至少以概率$1-(N-2)\cdot(1/R)^{N/2}$终止，一旦终止，$est_p = N$。*

证明。按照引理9.14，算法终止于一个配置，在这个配置中，所有估算值相同，称为e，假设$e < N$。按照顺时针方向对进程命名，从p_0到p_{N-1}。

当设置est_{p_i}的值为e时，进程p_i选择标号lbl_{p_i}，并产生令牌<**test**，e，lbl_{p_i}，h>。这个令牌被转发e个跳数，到达p_i的第e个顺时针近邻p_{i+e}，p_{i+e}接收这个令牌，并不进一步增加它的值。这蕴含着对于所有i，$lbl_{p_i} = lbl_{p_{i+e}}$。设$f = \gcd(N, e)$，利用等式$lbl_{p_i} = lbl_{p_{i+e}}$可得，当$i$和$j$相差$f$的倍数时，$p_i$和$p_j$所选的标号是相同的，即，

330 ~ 331

$$f \mid (j-i) \Rightarrow lbl_{p_i} = lbl_{p_j}$$

进程p_0到p_{N-1}被划分成f组，每组有N/f个进程。同组中的所有进程具有相同标号。

但由于标号可从{1，…，R}中随机选取，同组所有进程选择同一标号的概率是$(1/R)^{N/f-1}$，在所有f组中发生这种情况的概率仅为$(1/R)^{f(N/f-1)}=(1/R)^{N-f}$。对于$e$的所有可能值，$f$至多为$N/2$，对所有可能的不正确解的概率求和（从2至$N-1$），得到正确解的概率至少是$1-(N-2)(1/R)^{N/2}$。□

习题

9.1节

9.1　设ψ是后置条件，假定

（1）终止于Monte Carlo算法A的进程确定ψ，且

（2）一个确定、进程终止的**验证算法**B测试ψ是否成立。

如何构造Las Vegas算法C确定ψ？

9.2　设ψ是后置条件，假设ψ由Las Vegas算法A确定。已知由A交换的消息的期望数K。构造确定ψ的Monte Carlo算法，该算法以错误概率ε为参数（即算法必须终止且正确，概率为$1-\varepsilon$）。

9.3　给出一个确定、集中式的命名赋值算法，算法利用$2|E|$条消息和$O(D)$个时间单位。

9.4　[Ang80]给出团上的确定选举算法，其中的通信方式采用同步消息传递。

9.5　给出环上计算MAX的确定、消息-终止的算法。给出规模未知的任意网上计算MAX
332 的确定、消息-终止的算法。

9.6　考虑未知规模的匿名树上的选举问题，其中的通信方式采用异步消息传递。

（1）给出以概率1、部分正确的进程终止的随机算法。且算法的期望消息复杂度为$O(N)$。（实际上，期望消息复杂度可达到$N+1$。）

（2）在这种情况下，存在确定算法吗？

9.2节

9.7　证明不存在计算进程数的确定算法，对于所有直径至多为2的匿名网络都是正确的。

提示：利用图6.21中网络的对称性。

9.8　证明不存在已知环（即使规模已知）上的确定选举算法，其中的通信方式采用同步消息传递。

概述证明过程，证明对于所有组合环规模上的选举问题，选举算法的不可能性。

9.9　定义整数串$x=(x_1,\cdots,x_k)$上的函数f，如果所有x_i相等，则$f(x)$为真；否则$f(x)$为假。对于未知规模的环，f可以被进程终止算法确定地计算吗？对于规模未知的环，f可以被消息终止算法确定地计算吗？

9.10　问题同习题9.9，但是函数不同，如果x有非降循环移动，则$g(x)$为真即存在x的循环移动z，满足$i<j\Rightarrow z_i\leqslant z_j$。

9.3节

9.11　给出已知规模的匿名环上的进程-终止的Monte Carlo选举算法，你所设计的算法的消息复杂度是多少？算法成功的概率是多大？

可能达到任意接近1的成功概率吗？ 333

伪-匿名网络（pseudo-anonymous network）是这样的网络：进程具有标识，但标识不必都不相同。伪-匿名环（如，进程p_0到p_{N-1}）是周期的，如果存在数$k < N$，满足对于所有i，$id_i = id_{i+k}$。

9.12 证明在已知规模的非周期伪匿名环上，存在确定、进程终止的选举算法。

9.13 证明在未知规模的非周期伪匿名环上，不存在随机、进程终止且以正概率正确的选举算法。

9.4节

9.14 函数g的定义同习题9.10。给出未知规模的环上计算函数g的随机、消息终止的算法，且该算法以概率$1-\varepsilon$正确。

9.15 利用Safra的终止-检测算法（图8-7所示的算法）能够检测出Itai-Rodeh算法（图9-5所示的算法）的终止性吗？ 334

你能想出曾经讨论过的用于此目的的其他终止检测算法吗？

第10章 快　照

到目前为止我们考虑的算法（第8章中的算法除外）所执行的任务，都与网络结构或者全局网络功能有关。本章我们要讨论的算法，其任务是分析通常由其他算法所引起的计算性质。然而，令人吃惊的是，从分布式系统内部，我们很难了解这个系统的计算。在进行系统计算的算法设计时，计算和存储该计算的某个配置的过程，即所谓的快照（snapshot），是重要的基本任务。

按照定义2.22，可以用不同配置序列描述分布式计算，这个配置序列包括不同配置集。因此考虑哪个配置作为计算的配置并不是显而易见的。在10.1节中，将论述这个问题，其中把割、一致割及快照的概念放在一起。这些概念与第2章中定义的因果关系有紧密的联系。

一些应用促进了快照的出现。这里我们列举三个。

第一，可以对在某个配置内部所反映的计算性质，用一个观察（固定的）快照而不是观察（变化的）实际进程状态的算法，做离线分析。这些性质包括稳定性质。配置的一个性质P是稳定的，如果

$$P(\gamma) \wedge \gamma \rightsquigarrow \delta \Rightarrow P(\delta)$$

简单地说，如果计算到达使P成立的配置γ，从那时起，在每个配置δ中，P保持为真。因而，如果在计算的一次快照中，发现P为真，那么对于从那时起的所有配置，可以得出P都为 335
真。稳定性质的例子有终止、死锁、令牌丢失以及动态存储结构中对象的不可达性。

第二，如果由于进程故障计算必须重新开始，可以使用快照而不是初始配置。这样，在快照中所捕获的进程p的局部状态c_p在那个进程中恢复。此后算法继续运行。

第三，在调试分布式程序时，快照是有用的工具。对于取自错误执行的配置的离线分析可能揭示出程序的行为为什么不像期望的那样。

10.1 预备知识

设C是分布式系统的计算，由进程集合$\mathbb{P}$组成。Ev表示计算的事件集。我们做出弱公平性假设，每条消息将在有限时间内被接收。假设网络是（强）连通的。进程p的局部计算由进程状态$c_p^{(0)}$，$c_p^{(1)}$，…序列组成，其中$c_p^{(0)}$是进程p的初始状态。用进程p中的事件$e_p^{(i)}$的发生来标记从状态$c_p^{(i-1)}$到状态$c_p^{(i)}$的转移。参见图10-1。因此，$Ev = \cup_{p \in \mathbb{P}} \{ e_p^{(1)}, e_p^{(2)}, \cdots \}$。

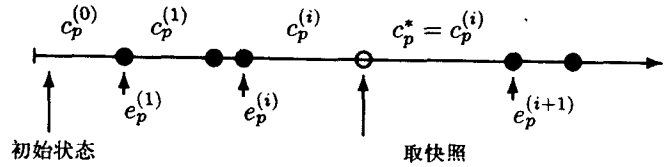


图10-1 进程p的计算

在进程p的事件中，定义局部因果序（local causal order）如下

$$e_p^{(i)} \preceq_p e_p^{(j)} \Leftrightarrow i \leqslant j$$

每个事件是发送事件、接收事件或内部事件之一。参见第2章中的定义。为了简化本章中算法和定理的表示，我们暂时假设进程的整个通信历史被反映在它的状态上。即如果存在从p到q的信道，那么，p的状态$c_p^{(i)}$包括在事件$e_p^{(1)}$至事件$e_p^{(i)}$中p已向q发送的消息列表$sent_{pq}^{(i)}$。同时，q的状态$c_q^{(i)}$包括在事件$e_q^{(1)}$至事件$c_q^{(i)}$中q已接收p的消息列表$rcvd_{pq}^{(i)}$。为了避免在算法的应用中对通信历史的明确存储，在10.3.1小节中，表明了如何撤消这个假设。

快照算法的目标是通过组合每个进程的局部状态（快照状态），明确地构造系统配置。进程p通过存储一个局部状态c_p^*，取得局部快照（local snapshot），称c_p^*为p的快照状态（snapshot state）。如果快照状态是$c_p^{(i)}$，即p取得事件$e_p^{(i)}$和事件$c_p^{(i+1)}$之间的快照，则事件$e_p^{(j)}$，$j \leqslant i$，称为p的前照事件（preshot event）；事件$e_p^{(j)}$，$j > i$，称为p的后照事件（postshot event）。在时间图中，我们在进程状态变成快照状态的位置用一个空心圆说明局部快照的获取，如图10-1和10-2所示。

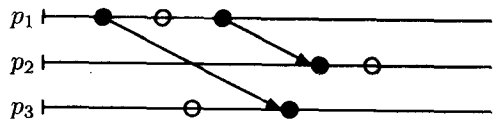


图10-2　不一致的快照

（全局）快照S^*由集合$\mathbb{P}$中每个进程p的快照状态c_p^*组成，记为$S^* = (c_{p_1}^*, \cdots, c_{p_N}^*)$。因为局部状态包括通信历史，如果信道$pq$的状态定义为$p$所发送（根据$c_p^*$）但不由$q$接收（根据$c_q^*$）的消息集，那么快照$S^*$定义了一个配置$\gamma^*$。换句话说，信道$pq$在快照$S^*$中的状态定义为消息$sent_{pq}^* \setminus rcvd_{pq}^*$集。用$\gamma^*$表示由快照状态和定义的信道状态组成的配置。

在构造配置的过程中，如果$rcvd_{pq}^*$不是$sent_{pq}^*$的子集，就会引起异常情况。参见图10-2。按照所收集的快照状态$c_{p_1}^*$，p_1向p_3发送消息，但是按照状态$c_{p_3}^*$，没有来自p_1的消息被接收。因此，信道p_1p_3包含快照中的一条信息，称这条消息在快照中“在传输中”。现在考虑从p_1发向p_2的消息。这条消息的发送是一个后照事件，而它的接收是一个前照事件。因此，按照状态$c_{p_1}^*$，在信道p_1p_2上没有消息发送。但是根据状态$c_{p_2}^*$，经此信道有消息被接收。因为$rcvd_{p_1p_2}^* \not\subseteq sent_{p_1p_2}^*$，对于信道$p_1p_2$的状态不能做出有意义的选择。受此启发，产生以下定义。

336
~
337

定义10.1　*如果对于每两个（近邻）进程p和q，有$rcvd_{pq}^* \subseteq sent_{pq}^*$，那么称快照$S^*$是可行的。*

快照的可行性意味着，在所蕴含的配置的构造过程中，“保留”在$rcvd_{pq}^*$中的消息都能从$sent_{pq}^*$中被删除。如果在前照事件中发送一条消息，那么我们称这条消息为前照消息（preshot message）；如果在后照事件中发送一条消息，那么我们称这条消息为后照消息（postshot message）。

在计算的事件集合中，快照和有限割（cut）存在一一对应关系。割是事件集，对于局部因果关系，它是左闭合的。

定义10.2　*Ev的割是集合$L \subseteq Ev$，满足*

$$e \in L \wedge e' \preceq_p e \Rightarrow e' \in L$$

如果$L_1 \subseteq L_2$，则称割L_2在割L_1之后。

一方面，对于每个快照S^*，很容易把前照事件的集合L看作一个有限割。另一方面，设L是有限割，对于每个进程p，或者p中没有一个事件包含在L中（在这种情况下，设$m_p = 0$），或者有一个最大事件$e_p^{(m_p)}$包含在L中，并且所有满足$e \preceq_p e_p^{(m_p)}$的事件也包含在L中。于是，L恰好是由$S^* = (c_{p_1}^{(m_{p_1})}, \ldots, c_{p_N}^{(m_{p_N})})$所定义的快照的前照事件集。

快照可用于导出关于计算的信息，由此计算得到快照，但是任意获取的一个快照几乎不能提供关于这个计算的信息。作为例子，考虑基于单个令牌传递的算法，如6.3节的遍历算法。假设进程p持有令牌时获取它的快照，一段时间后，当进程q持有令牌时也获取它的快照。在所构造的配置中，两个进程都持有令牌，这正是在算法的任何计算中，从未发生的情况。

凭直觉，我们希望快照算法能够计算出“实际出现”的一个配置。然而，令人遗憾的是，所出现的配置集合并不是等价不变式，如在2.3.2小节中证明的那样，因而它不能被定义来用于计算。因此我们会将计算中可能的任何配置（例如，发生在计算的某些执行中），作为算法有意义的输出。 338

定义10.3 如果存在一次执行$E \in C$，满足γ^*是E的一个配置，那么称快照S^*在计算C中是有意义的。

我们要求快照算法可以协调局部快照配置，使得全局快照是有意义的。在10.3.2小节讨论快照的适时性。

快照的可行性只强加了近邻的局部快照之间的关系，而更有意义的是快照的全局性质。以下还可证明（定理10.5），具有可行性的快照是有意义的，反之亦然。同时这些快照与一致割相对应。称割是一致的，如果它关于因果关系是左闭合的。

定义10.4 Ev的一致割是一个集合$L \subseteq Ev$，满足

$$e \in L \wedge e' \preceq e \Rightarrow e' \in L$$

定理10.5 设S^*是快照，L是S^*所蕴含的割。以下三种声明是等价的。

（1）S^*是可行的。

（2）L是一致割。

（3）S^*是有意义的。

证明。我们证明（1）蕴含着（2）、（2）蕴含着（3）和（3）蕴含着（1）。

先证明（1）蕴含着（2）。假设S^*是可行的，为了证明L是一致的，取$e \in L$且$e' \preceq e$。由$\preceq$的定义，只需证明$e' \in L$在以下两种情况下成立

（1）$e' \preceq_p e$（其中p是e'和e发生时的进程）：在这种情况下，由于L是割，可得$e' \in L$。

（2）e'是发送事件，e是相应的接收事件：设p是事件e'发生时的进程，q是事件e发生时的进程，m是在这些事件中交换的消息。现在

$$\begin{aligned} e \in L &\Rightarrow m \in rcvd^*_{pq} \text{因为} e \text{是前照事件} \\ &\Rightarrow m \in sent^*_{pq} \text{因为} S^* \text{是可行的} \\ &\Rightarrow e' \in L \end{aligned}$$

（2）蕴含着（3）。证明的关键是构造一次执行，在这次执行中，所有前照事件在所有后

照事件之前发生。

设$f=(f_0, f_1, \cdots)$是Ev的枚举，定义如下。首先，f按照与$\preceq$一致的次序列出了Ev中的
339 所有前照事件；然后，f按照与$\preceq$一致的次序列出了后照事件，如图10-3所示。为了应用定理2.21，必须证明，整个序列f与$\preceq$一致。设$f_i \preceq f_j$，如果f_i和f_j都是前照事件，因为f按照与$\preceq$一致的次序枚举了前照事件，由此可得，$i < j$。如果f_i和f_j都是后照事件，同样成立。如果f_i是前照事件而f_j是后照事件，因为f包含的所有前照事件在所有后照事件之前，由此可得，$i < j$。通过假设割L是一致的，可以排除f_j是前照事件而f_i是后照事件的情况；即如果$f_j \in L$且$f_i \preceq f_j$，那么$f_i \in L$，即f_i是前照事件。我们可以得出结论，f与$\preceq$是一致的。

由定理2.21，存在执行F，由Ev中的事件组成，按照次序f出现。在所有前照事件执行后，执行F立即包含配置γ^*。

（3）蕴含着（1）。如果S^*是有意义的，γ^*出现在C的一次执行中。在每次执行中，消息在接收之前被发送，这蕴含着，对于每个p和q，$rcvd^*_{pq} \subseteq sent^*_{pq}$。因此$S^*$是可行的。 □

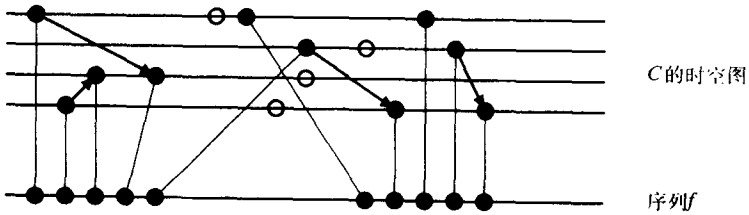


图10-3　执行F

10.2　两个快照算法

由定理10.5，足以协调局部快照的配置，来确保所得快照是可行的。这可以把对快照算法的要求简化为以下两个性质。

（1）在每个进程中必须触发对局部快照的获取。

（2）在前照事件中不接收后照消息。

在所有快照算法中，保证在接收后照消息之前，进程获取其快照。本节的两个算法，在
340 如何识别这些消息以及如何保证每个进程获取快照这两方面有所不同。

为了正确区别快照算法中的消息和计算中的消息，称快照算法中的消息为控制消息，计算中的消息为基本消息。

10.2.1　Chandy-Lamport算法

在本小节中，假设信道是fifo的，即经过某一信道发送的消息，以与发送顺序相同的次序被接收。在Chandy-Lamport [CL85]算法中，进程通过每个信道发送特殊的消息**<mkr>**（标记消息（marker)），相互通知所构造的快照。当进程获取它的局部快照时，经过各近邻信道，每个进程只发送一次标记消息，标记消息是控制消息。还未获取快照的进程接到**<mkr>**消息，会引起该进程取得快照，同时发送**<mkr>**消息。参见图10-4所示的算法。算法在计算C上并发执行。

```
var taken_p : boolean  init false ;

To initiate the algorithm:
    begin record the local state ; taken_p := true ;
          forall q ∈ Neigh_p do send ⟨mkr⟩ to q
    end

If a marker has arrived:
    begin receive ⟨mkr⟩ ;
          if not taken_p then
            begin record the local state ; taken_p := true ;
                  forall q ∈ Neigh_p do send ⟨mkr⟩ to q
            end
    end
```

图10-4 Chandy-Lamport快照算法

引理10.6 *如果至少有一个进程开始执行算法，那么在有限时间内，所有进程取得局部快照。*

证明。 因为每个进程取得快照和发送消息<**mkr**>至多一次。快照算法的活动在有限时间内停止。如果p是那时已经取得快照的一个进程，且q是p的近邻，那么q也已经取得快照。这是由于p所发送的消息<**mkr**>已经被q接收到，如果q还未取得快照的话，会导致它取得快照。 341
因为至少有一个进程初始化算法，则至少有一个进程已经取得快照。网络的连通性蕴含着所有进程都已取得快照。 □

观察可见，算法必须被至少一个进程初始化，但是如果算法由任意非空进程集初始化，它也能正确工作。

定理10.7 *在至少有一个进程初始化算法之后，*Chandy-Lamport*算法（图10-4所示的算法）在有限时间内计算有意义的快照。*

证明。 由前述引理，算法在有限时间内计算快照。接下来要证明所得快照是可行的，即在后照事件中接收每个后照（基本）消息。设m是p发送给q的后照消息。在发送m之前，p取得局部快照，并把消息<**mkr**>发送到所有它的近邻中，包括q。因为信道是fifo的，q在m之前接到这个消息<**mkr**>，按照算法，q一旦接到这个消息，就取得它的快照。因此，m的接收是后照事件。 □

每个进程中，Chandy-Lamport算法需要交换$2 \cdot |E|$条<**mkr**>消息，需要一位存储空间（当然，加上所记录的快照状态）。它的时间复杂度为$O(D)$。

10.2.2 Lai-Yang算法

Lai-Yang [LY87]算法并不依赖信道的fifo性质。因此，该算法不可能像在Chandy-Lamport算法中所作的那样，利用标记消息就能将前照消息和后照消息“分开”。而是在每个基本消息中用表明它是前照事件还是后照事件的信息做出标记。因此，发送C中一条消息的进程p，将$taken_p$的值追加在其后。由于在这里不关心的消息c的内容，我们把这些消息简单的表示为<**mes**，c>，其中c是发送进程中所包含的$taken$的值。快照算法在接到第一条快照消息之前，检查进来的消息，并记录它的局部状态。参见图10-5所示的算法。

```
var taken_p : boolean  init false ;

To initiate the algorithm:
    begin record the local state ; taken := true end

To send a message of C:
    send ⟨mes, taken_p⟩

If a message ⟨mes, c⟩ has arrived:
    begin receive ⟨mes, c⟩ ;
          if c and not taken_p then
            begin record the local state ; taken := true end ;
          change state as in the receive event of C
    end
```

图10-5　Lai-Yang快照算法

图10-5不交换控制消息，但并不确保每个进程最终记录它的状态，可能确实做不到这一点。考虑进程p，它不是快照算法的初始进程，假设p的近邻在取得它们的局部快照之后，并没有向p发送消息。在这种情况下，p从不记录它的状态。快照算法以不完整的快照终止。

解决这个问题依赖于对计算C的了解。如果与每个进程的最终通信得到保证，将总能获得完整的快照。否则，如果初始时需交换所有进程中的控制消息，就要对算法进行扩充，正如图10-4的算法所示。因而这些消息仅保证，每个进程最终记录它的状态（如引理10.6中所证明的），但在证明所得快照的可行性方面不起作用。不论是哪种方法，都可以保证完整快照的计算。

定理10.8　*Lai-Yang算法（图10-5所示的算法）只能计算有意义的快照。*

证明。考虑由算法计算的一个快照。设m = <**mes**, c>是从p发送到q的后照消息。这蕴含着，$c = true$，因此q最迟一旦接到m，就取得快照。因此，q所记录的状态发生在m的接收之前，m的接收是一个后照事件。（回忆一下，这仅仅影响哪个局部状态被记录，与何时记录无关；在这种情况下，记录可能与第一个后照事件同时发生。）　□

禁用　Chandy-Lamport算法（图10-4所示的算法）利用标记消息区别前照消息和后照消息，并要求信道是fifo的。Lai-Yang算法（图10-5所示的算法）把它们是前照事件还是后照事件明确地包含在基本消息中，因此要求算法有捎带（piggyback）基本消息中信息的能力。Taylor[Tay89]已经证明，如果信道不是fifo并且不能使用捎带，快照问题的解决方法一定会受到限制，即，暂时挂起基本计算。在关于通信的各种假设下，对不同限制类型的分类和对必须的限制的特征描述，可在Critchlow和Taylor的文献[CT90]中找到。

342 ~ 343

10.3　使用快照算法

10.3.1　计算信道状态

到目前为止，假设每个进程的状态包含进程的通信历史，因此可从近邻进程的状态计算出快照中信道的状态。然而，在大多数情况下，由于开销太大，不能把进程发送和接收的所有消息明确地存储起来。现在我们将要阐述，如何更有效地构造信道状态。

1. 简化有关信息

取决于快照的目的，可以只存储有限数量的关于通信历史的有关信息就足够了。例如，我们可能希望构造令牌-交换算法（如Korach-Kutten-Moran算法）的快照，确定有多少令牌遍历仍处在活动状态。为了确定每个信道中的令牌数，每个进程存储它所发送和接收的令牌数就足够了。作为第二个例子，构造快照来测试是否计算已经终止（参见第8章）。这仅仅需要确定信道是否为空，只需对消息计数，而无需确切地存储它们。（在这两个例子中，每个进程所记录的状态也可以限制到少量有关信息上，即，驻留在进程中的令牌数，或者进程是处在活动状态还是被动状态。）

2. 明确构造信道状态

借助以下引理，可以明确地构造信道状态。

引理10.9 *在可行的快照中，$sent^*_{pq}\backslash rcvd^*_{pq}$等于p在前照事件中所发送的和q在后照事件中所接收的消息集。*

证明。一方面，易于证明，一条属于$sent^*_{pq}\backslash rcvd^*_{pq}$中的消息$m$，在前照中被发送和在后照中被接收。另一方面，如果消息m在前照中发送和后照中接收，根据定义，它被包括在$sent^*_{pq}$中，但不在$rcvd^*_{pq}$中，因此它在$sent^*_{pq}\backslash rcvd^*_{pq}$。 □ 344

通过记录所有在后照事件中接收的前照消息，进程q构造信道pq的状态。由于所有这些消息在q取得它的快照之后才被接收，q在记录它的状态之后才开始记录消息，并且所有前照消息到达之后，停止记录。

Chandy-Lamport算法。在从p到q发送消息<**mkr**>之前，接收所有从p到q的前照消息，只有前照消息在标记消息之前被接收。在这种情况下，信道状态的构造极其简单：信道pq的状态是消息的集合，这些消息在记录完状态后，以及接收p的<**mkr**>消息之前被q接收。Lai-Yang算法。如果信道不是fifo的，进程q可能交替地接收前照消息和后照消息。后照消息的接收并不蕴含着，所有前照消息已被接收。因此，尽管很清楚，q应该记录哪些消息（即，标记为*false*的消息是前照消息），但是不清楚q何时构造完整的信道状态，并能够停止记录。

Mattern [Mat89c]提出了一种解决这个问题的方法，p可以计算发送给q的前照消息的总数，并把这个数通知给q（或者在某个单独的消息中，或者在后照消息中捎带这个数）。进程q计算所接到的前照消息的数目（包括前照事件中接到的消息和后照事件中接到的消息），当接到足够多的前照消息时，终止信道状态的构造。

10.3.2 快照的适时性

由本章算法所构造的快照被保证在其被取得的计算中是有意义的。除此之外，我们希望快照是最近的，即由快照得到的信息不是很旧。由于需要一种时间表示方法，能够表达出信息新近的程度，因此我们将快照与分布式执行中出现的配置联系起来。在一次执行中，所计算的快照“位于”记录开始的配置和记录结束的配置之间。如下列定理所表达的。

定理10.10 *设E是一次执行，S^*是这次执行中取得的可行快照，γ^*是蕴含的系统配置。设γ_s是这样一个配置，其中第一个进程记录它的状态，设γ_c是这样一个配置，其中最后一个进程记录它的状态。* 345

那么，$\gamma_s \rightsquigarrow \gamma^* \rightsquigarrow \gamma_c$。

证明。我们强调定理10.5证明中使用的论证方法。通过首先按照它们在E中发生的次序枚举E中的前照事件，然后按照它们在E中的出现次序，枚举E中的后照事件，来构造事件序列f。

这个枚举与E的因果序一致，因此定义了一次执行F。

F按照这种次序包含配置γ_s、γ^*和γ_e。这显示了所需的结果。F包含γ_s，是因为E中γ_s之前的所有事件都是前照事件，因此f恰好以相同次序从这些事件开始。F包含γ_e，是因为执行E中γ_e之前的事件集合包含所有前照事件，因此f由这些事件开始（尽管可能以不同次序）。根据定理2.21，F包含γ_e。 □

10.3.3 稳定性检测

设P是配置的稳定性质，一旦执行达到了使P成立的配置，P会永远成立。稳定性质包括：

（1）*终止性* 如果γ是终止配置，且$\gamma \rightsquigarrow \delta$，那么，$\gamma = \delta$，因此$\delta$是终止配置。所以，终止配置问题（第8章），可通过计算快照并为其中的活动进程和基本消息检查此快照，加以解决。然而第8章中的特殊解决方法通常更有效。

（2）*死锁* 如果在配置γ中，因为S中的所有进程等待S中的其他进程，使得进程的一个子集S被阻塞，即使S外的进程状态可能已经改变，在以后的配置中这种阻塞情况同样成立。检测死锁的问题在10.4节中讨论。

（3）*令牌丢失* 考虑在进程间循环传递令牌的算法。进程可能消耗令牌。性质“至多有k个令牌”是稳定的，因为令牌可能被消耗，但是不会产生。

（4）*无用信息* 在面向对象程序设计环境中，创建对象集合，每个对象可能保持对其他对象的引用。如果按照引用，能够找到一条从某个指定对象到该对象的路径，则称该对象
346 是*可达的*，否则称该对象是*无用信息*。引用可以被增加和删除。但是对无用信息的引用不能增加。因此，一旦一个对象变成无用信息，它将永远是无用信息。（有许多关于无用信息收集问题的文献，本书中不做论述。感兴趣的读者可以参考[TM93]以及文章中的参考文献。）

为了在P变为真时，采取适当的行动，可以通过另外一个算法，检测P的真值，该算法用于观察计算，并且一旦发现P为真时，则触发行动。

一个非常通用的检测机制作为图10-6所示的算法给出。算法要求对于快照配置，检测P是否成立。但是通常可以通过一个相对简单的分布式过程做到这一点。检测算法的任务是观察分布式计算的动态行为，而这个过程只对静态数据进行分析（过程的发生，表示一个配置）。*Pholds*的计算既可通过集中式，又可通过分布式方式。在集中式计算中，每个进程向一个特定进程发送它的快照状态，这个特定进程构造完整配置，并计算*Pholds*。在分布式计算中，计算是由所有进程合作进行的。

```
var detect    : boolean    init false ;

while not Pholds do
      begin compute a global snapshot γ* ;
            compute Pholds := P(γ*)
      end ;
detect := true
```

图10-6 稳定-性质-检测算法

图10-6所示的算法满足安全性和活动性，如以下两个定理所表述的。这些性质就是检测算法中所要求的性质。假设图10-6所示的算法观察产生执行E的分布式算法。

定理10.11 （安全性）如果图10-6所示的算法将*ture*赋给变量*detect*，而系统此时配置为δ，那么$P(\delta)$成立。

证明。设γ^*是所计算的最后快照，γ_e是E中的配置，最后局部快照γ^*取自该配置。由算法， 347
$P(\gamma^*)$成立。由定理10.10，$\gamma^* \rightsquigarrow \gamma_e$。因为在$\gamma^*$完成之后，才对*detect*赋值，$\gamma_e \rightsquigarrow \delta$。由$P$的稳定性，蕴含$P(\delta)$成立。 □

定理10.12 （活动性）如果E达到使P成立的配置，那么图10-6所示的算法在有限时间内，将*true*值赋给变量*detect*。

证明。假设δ是E的满足P的第一个配置。设γ_s是δ后（或者等于它）的第一个配置，在这个配置中，图10-6所示的算法开始计算快照。γ^*是计算在γ_s中开始的快照。由于$\delta \rightsquigarrow \gamma_s \rightsquigarrow \gamma^*$成立，所以$P(\gamma^*)$成立。这一轮之后，检测算法终止。 □

图10-6所示的算法可用于解决所有稳定-检测问题，但是在使用时，注意以下几点：

（1）对于某个特定性质P，$P(\gamma^*)$的（分布式）求值可能需要相当复杂的算法。这是针对死锁-检测问题的某些模型的情况；参见10.4节，测试给定配置中死锁的一个算法。

（2）在某些情况下，由于空间限制，方法是不可行的。这就是无用信息收集问题的情况。存储所有调用和快照消息会使算法对空间的要求增加两倍以上，代价太大。

当评估无用信息的谓词时，早先的无用信息收集算法挂起基本计算，这种方法相当基础，但由于挂起基本计算感觉是不可接受的，这种方法通常遭到拒绝。更多最新算法是“不工作”，而不是对与某个配置操作。这些算法与基本计算协作，而且不属于依赖快照计算的算法。参见文献[Tel91b，第5章]和[TM93]。

（3）即使在某些情形下，其中基于图10-6所示算法的解决方法是简单和有效的，这些算法也不可能是最简单和最有效的。这种情况出现在终止-检测问题中，在第8章中提出了相当基本的解决方法。

快照可被用于对系统配置的*单调函数*（monotonic function）求值。称函数f是单调的，如果

$$\gamma \rightsquigarrow \delta \Rightarrow f(\gamma) \leqslant f(\delta)$$

如果计算快照γ^*，并对$f^* = f(\gamma^*)$求值，那么对于其后的每个配置δ，$f(\delta) \geqslant f^*$成立。分 348
布式离散-事件模拟器的*全局虚拟时间*（global virtual time，GVT）是单调函数的例子（参见[Tel91b，第4章]或者[CBT97]）。

10.4 应用：死锁检测

在某些分布式程序中，当进程等待来自其他进程的消息时，暂时被阻塞。这样的例子包括数据库系统中的事务处理，以及竞争几种资源的进程等。如果出现死锁，而且得不到恰当处理，就会严重危及分布式系统的性能和使用。首先，死锁中涉及的进程将永远不能完成它们的任务，其次，这些进程所占据的资源也不能为其他进程所用。显然，在分布式系统的实现中，恰当地处理死锁是最重要的。

10.4.1 基本计算模型和问题阐述

考虑执行基本计算（basic computation）的进程的集合$\mathbb{P}$，这里不关心它的目的和工作方

式。进程由于等待其他进程的消息可能不得不挂起它的局部计算。这可以一个用阻塞的状态代替进程的活动状态来模型化。

当进程p变成阻塞时（图10-7所示的算法中的行为$\mathbf{B}_p$），它将请求消息发给用$Reqs_p$表示的进程集。为了再次变成活动状态（行为$\mathbf{F}_p$），p必须接收$Reqs_p$中进程的准许消息，但不必是从全部进程。$Reqs_p$的子集，就是那些满足谓词$Free_p$的子集其中来自$Reqs_p$的准许消息足以使p再次成为活动状态。p所接收的请求（行为$\mathbf{R}_p$）存储在集合$Pend_p$中，当p自身处于活动状态时，行为$\mathbf{G}_p$对这些请求作出应答。进程p发出的请求数，附在每次请求和准许消息的后面，使得p可以抛弃为对陈旧消息的应答而发送的准许消息（参见$\mathbf{F}_p$）。

```
var state_p  : (active, blocked)   init active ;
    v_p      : integer             init 0 ;     (* Request number *)
    Reqs_p   : set of processes ;                (* Request set *)
    Grant_p  : set of processes ;                (* Received grants *)
    Pend_p   : set of requests     init ∅ ;      (* Pending requests *)
    Free_p   : predicate ;                       (* Freeing constraints *)

B_p: { state_p = active }
    begin determine Reqs_p and Free_p ; v_p := v_p + 1 ;
          forall q ∈ Reqs_p do send ⟨req, v_p⟩ to q ;
          Grant_p := ∅ ; state_p := blocked
    end

R_p: { A message ⟨req, v⟩ arrives from q }
    begin receive ⟨req, v⟩ from q ; Pend_p := Pend_p ∪ {(q, v)} end

G_p: { state_p = active ∧ (q, v) ∈ Pend_p }
    begin send ⟨grant, v⟩ to q ; Pend_p := Pend_p \ {(q, v)} end

F_p: { A message ⟨grant, v⟩ arrives from q }
    begin receive ⟨grant, v⟩ from q ;
          if state_p = blocked and v = v_p then
            begin Grant_p := Grant_p ∪ {q} ;
                  if Free_p(Grant_p) then state_p := active
            end
    end
```

图10-7　基本算法

我们用几个例子说明$Free_p$的使用。首先，如果p需要应答所有请求消息，$Free_p$仅对$Reqs_p$为真。其次，如果一条准许消息对于p是足够的，那么对于每一非空集合，$Free_p$为真。第三也是最后，假设p需要1000M-字节自由存储空间，并向磁盘A（900M字节自由存储空间）、B（600M字节自由存储空间）、C（300M字节自由存储空间）和D（100M字节自由存储空间）发出请求，如果A和其他的任何磁盘可用，p能继续。同样，如果B、C和D可用，p也能继续。这里$Free_p(S)$可被表示成

$$(\#S \geq 2 \wedge A \in S) \vee (\#S \geq 3)$$

每次p受到阻塞时，谓词$Free_p$可能不同。但总是满足以下两个假设。

D1。$Free_p(\varnothing)$为假，$Free_p(Reqs_p)$为真。

D2。$Free_p(A)$和$A \subseteq B$蕴含$Free_p(B)$。

按照D1，在接到任何准许消息之前，p不会变成活动的。当接到来自$Reqs_p$中所有进程的准许消息时，p才变成活动的。按照D2，如果p收到多于最小准许消息集合的消息，p仍然可能变成活动的。

根据基本算法，我们现在定义死锁进程或者死锁配置。

定义10.13 *称进程p在配置γ中是活着的，如果存在由γ可达的配置δ，在这个配置中p是活动的；如果它不是活着的，则进程p是死锁的。* 349 ~ 350

如果在γ中，存在死锁进程，则称配置γ是死锁配置。

显然，死锁具有稳定性质。如果p在γ_1中是死锁的，且$\gamma_1 \rightsquigarrow \gamma_2$，那么$p$在$\gamma_2$中是死锁的。将系统从配置$\gamma$带到配置$\delta$的基本计算的事件序列，称为基本计算从$\gamma$到$\delta$的一个连续序列。

死锁检测问题是设计一个控制算法，能够被强加于基本计算之上，并满足以下三个要求。

（1）非-干涉性 控制算法不会影响基本计算。

（2）活动性 如果出现死锁，控制算法可以检测到。

（3）安全性 仅当存在死锁时，算法才进行死锁检测。

控制算法可能要求交换附加信息，称为控制消息，活动进程或者阻塞进程都可发送控制消息。除了报告发生死锁之外，通常还要求控制算法识别一个或者多个将被用于死锁解除的死锁进程。

10.4.2 全局-标记算法

本节中，我们提出一个称为全局-标记（global-marking）的算法，用于死锁检测和所有死锁进程的识别。该算法不断计算一个快照，并将算法10-8应用于所构造的快照配置。$state^*_p$，v^*_p，$Reqs^*_p$，$Grant^*_p$，$Pend^*_p$，$Free^*_p$和（$sent^*_{qp} \backslash rcvd^*_{qp}$）的值作为常量（由快照算法计算出它们的值）。算法模拟准许消息的交换和来自γ^*的一个连续序列中的进程的释放过程，以确定哪个进程是活着的。

变量$alive_p$初始值为假，但当p检测出它在快照中是活着的，则设它为真。这就是当p在快照中处于活动状态的情况。但是如果到目前为止所接收的**<grant，v^*_p>**和**<Alive>**消息集合满足$Free^*_p$，同样设$alive_p$为真。变量$GrRec_p$包含快照中已经收到的或者传输中的准许消息（针对当前请求）、以及p所收到的**<Alive>**消息组成的集合。

定理10.14 *全局-标记算法终止，并且一旦终止，$alive_p$为真，当且仅当p在γ^*中是活着的。* 351

证明。 进程p至多将$alive_p$设为*true*一次（并发送相关的**<Alive>**消息），因此只交换有限多的消息。

标记算法模拟γ^*中基本算法的一种可能的连续序列，其中变成活动的进程就是那些设置*alive*为*true*，但准许消息是**<Alive>**消息的进程。在模拟的一个连续序列中，当进程变成活动时，就立即准许所有挂起的请求。执行$\mathbf{M}_p$的进程p在γ^*中是活动的，或者由于接到γ^*中正在传输的消息就变成活动的，因此进程p在γ^*中是活着的。在行为$\mathbf{P}_p$中设置$alive_p$为*true*的进程p，因为接到连续序列所发送的准许消息，而在模拟的连续序列中变成活动的。这也蕴含着进程在γ^*中是活着的。因此当标记算法终止时，所有$alive_p = true$的进程p在γ^*中都是活着的。

接下来要证明，如果进程p在γ^*中是活着的，那么$alive_p$的值为*ture*。设p在γ^*中是活着的，且（f_1，…，f_l）为γ^*中的一个连续序列，该连续序列导致生成一个其中的p处于活动状态的

配置δ。按照这样的顺序（γ_0，γ_1，…，γ_l）调用配置序列，其中$\gamma_0 = \gamma^*$，$\gamma_l = \delta$。

```
var alive   : boolean      init false;
    GrRec   : set of ℙ     init Grant*_p ∪ {q : ⟨grant, v*_p⟩ ∈ sent*_qp \ rcvd*_qp} ;

M_p: (* Start marking in active process *)
    { ¬alive_p  ∧  (state*_p = active  ∨  Free*_p(GrRec_p)) }
    begin alive_p := true ;
          forall q ∈ Pend*_p ∪ {q : ⟨req, v⟩ ∈ (sent*_qp \ rcvd*_qp)}
                do send ⟨Alive⟩ to q
    end

P_p: (* Propagate marking in freed processes *)
    { An ⟨Alive⟩ message arrives from q }
    begin receive ⟨Alive⟩ from q ; GrRec_p := GrRec_p ∪ {q} ;
          if not alive_p and Free*_p(GrRec_p) then
            begin alive_p := true ;
                  forall q ∈ Pend*_p ∪ {q : ⟨req, v⟩ ∈ (sent*_qp \ rcvd*_qp)}
                        do send ⟨Alive⟩ to q
            end
    end
```

图10-8　全局-标记算法

352　我们对i用归纳法证明，如果γ_i是进程q为活动的第一个配置（位于γ_0到γ_l），那么在标记过程中，$alive_p$值设置为*true*。

情形$i = 0$：由于q在γ^*中是活动的，那么行为$\mathbf{M}_q$是可应用的，将$alive_q$值设置为*true*。

情形$i > 0$：进程q在γ_0到γ_{i-1}中受到阻塞，在事件f_i中变成活动的。显然，f_i是对消息<**grant**，v^*_q>的接收，并把$Grant_q$加入到满足$Free^*_q(G)$的集合G中。我们证明，对于每个$r \in G$，r最终被包括在标记算法的$GrRec_q$中。考虑两种情况。

（1）在γ^*中，消息<**grant**，v^*_q>在从r到q的传输中。在这种情况下，在初始化为$Grant^*_q \cup \{r : \langle \mathbf{grant}, v^*_q\rangle \in sent^*_{rq} \setminus rcvd^*_{rq}\}$时，$r$已经被包括在$GrRec_q$中。

（2）在事件f_1至f_{i-1}中，有一个事件发送了消息<**grant**，v_q>。在这种情况下，$(q, v_q) \in Pend_r$或者消息<**req**，v_q>在配置γ中到r的传输中，r在配置γ_0至配置γ_{i-2}之间的一个配置中是活动的。由归纳假设，标记算法将$alive_r$的值设为*true*，当这种情况发生时，r向q发送<**Alive**>消息，接到消息之后，$r \in GrRec_q$成立。

如果在初始化后，$GrRec_q$包含G，则$\mathbf{M}_q$是可应用的，并设置$alive_q$值为*true*；否则，在接到完成G的消息<**Alive**>后，设$alive_q$值为*true*。

特别地，$alive_p$被设为*true*。　□

图10-8所示的算法的终止性是隐含的，但是可通过使用终止检测算法（第8章）来检测。当检测出全局标记算法的终止性时，那些$alive_p = false$的进程p在γ^*中处于死锁状态。

10.4.3　受限模型的死锁检测

本节中所用的基本计算模型是用于研究死锁-检测问题的最通用模型。用这种模型检测死锁的算法并不是很多。

Brzezinski、Hélary和Raynal[BHR92]提出了一种算法，该算法也是基于模拟基本算法的

计算。他们的算法将所有控制通信集中在一个嵌入式的环上，循环传递包含着不知是否还活着的进程集的令牌。保持所有信息集中于令牌中，便于进行标记的终止检测。当令牌环行一周，集合还未改变时，检测出终止。 353

常常在与进程激活有关的更严格的假设下研究终止检测问题，允许利用图论来刻画死锁特性。在Knapp的文章[Kna87]中，综述了几种模型和算法。有两种最常用的模型，一种是进程必须获得请求集合中所有进程的准许，才能变成活动状态（AND模型）。另一种是进程只需获得请求集合中某个进程的准许，才能变成活动状态（OR模型）。

1. 资源死锁：AND模型

分布式数据库由分散在许多站点（计算机）上的文件集合组成。数据库管理系统允许用户访问这些数据，或者只读取数据，或者修改数据。通过数据库事务处理，以结构化的方式访问数据，事务处理通常将数据定址在不同点上。由于数据项的分散性，在不同事务处理时，必须特别仔细以避免由不同事务采取的步骤发生交错。通常采用对事务所操作的数据项加锁的方法来保证数据库操作的正确性。当然，如果所要求的数据已经被另一事务加锁，则事务必须等待。

为了变成活动状态，事物必须获得它所请求的全部锁。即$Free_p\ (G) \equiv (G = Reqs_p)$。当进程所有请求得到准许后，进程才能变成活动状态，这种死锁的受限模型称作是AND模型。

可以证明，在AND模型中，死锁与等待图（wait-for graph）中的循环等价。等价图是这样一种图，如果进积p受到阻塞，带边pq的所有进程要等待q。例如，Chandy、Misra和Haas[CMH83]、Menasce和Muntz[MM79]以及Mitchell和Merritt[MM84]提出了检查图中循环的算法。

2. 通信死锁：OR模型

在分布式算法中，与其他进程合作的进程可能进入死锁状态，在这种状态下，惟一可能的事件是与其他进程的通信。任何通信事件的执行都会使得进程进入另一种状态，从这个状态，进程可能继续它的计算。因此要变成活动状态，进程的某个请求得到准许就足够了，即，对于$Reqs_p$的每个非空子集，$Free_p$为$true$。某个请求得到准许就足够使得进程变成活动状态，这种死锁的受限模型称为OR模型。

在OR模型中，死锁等价于在等待图中出现结。例如，Chandy、Misra和Haas[CMH83]以及Natarajan[Nat86]提出了针对这种模型的算法。 354

习题

10.1节

10.1 将p的局部快照配置作为附加的内部事件a_p，证明

S^*是有意义的 $\Leftrightarrow \forall_p, q: a_p \parallel a_q$

10.3节

10.2 给出Lai-Yang算法的完整描述，包括快照完成和信道状态构造的实施机制。

10.3 Przlwytszkowsky教授写道：

“阅读第10章，提高了我对第8章算法的理解。在Safra算法（图8-7所示的算法）中，例如，

p对令牌的处理应该被看作是定义p的快照状态。在所构造的快照中，所有进程是被动的，因为令牌只能由被动进程处理。因此*Pholds*的计算只要求一次检查，以确定是否所有信道为空，因此，令牌收集消息计数的和。

然而，我不明白，黑白颜色的作用，以及如何保证快照是有意义的”。

355　你能够帮助教授吗？

第11章 方向侦听与定向

规则结构的网络，如圆环、超立方体，通常在它们的链路上标以方向，我们现在讨论一些最新评价这种标注方法好处的研究成果，称之为方向侦听（sense of direction），或简记为SoD。方向侦听的可用性增强了模型，使得处理器之间的通信效率更高，并且在算法中利用可以网络拓扑结构的性质。

已经证明，方向侦听可以降低某些问题的复杂度，但是可应用方向倾听的问题数量非常少。确实，对于许多问题，如果充分利用边路标号，就能得到更有效和更容易的算法。然而，在许多非平凡的例子中，对于未做标号的例子，即使有好的下界，也不可用。此外，在未标号的图中，存在令人惊奇的有效算法。

本章试图按照由Santoro[San84]开始的科学研究，建立在两个领域支持方向侦听的例子：广播和选举。

我们给出几类具体网络的方向侦听定义，并且表明对于环网络如果弦和其上的方向侦听可用，可以更有效的解决环上的选举问题；如果SoD可用，可以用线性复杂度解决超立方体和团上的选举问题；但是没有SoD，利用随机算法也可达到同样的复杂度。同时讨论了在无任何已知条件的网络上，计算网络中SoD的算法。

本章综述。11.1节介绍了一些基本的表示方法，以及本章后面要讨论的一些问题。11.1.1节引入了方向侦听的形式定义。11.1.2节说明了它的用处。11.1.3节描述了如何利用一致方向侦听，有效地广播消息。11.2节提出了具有方向侦听的环和弦环上的选举方法，包括借助方向侦听降低团上选举复杂度的示例。 356

11.3节研究超立方体上方向侦听的优点。近年来，人们已经知道在标号超立方体上存在非常有效的算法。而在未标号的超立方体上，算法却不太有效。研究的重点放在证明相匹配的下界上。最近，人们提出了未标号超立方体上的更有效的算法，这个算法表明，有效性的获得是拓扑结构的固有特性，而与边路标号无关。

11.4节讨论了各种与复杂度有关的主题，包括计算未标号的网络上的方向侦听的算法。最后以讨论和一些需继续研究的问题结束本章（11.5节）。

11.1 引言和定义

和往常一样，我们用有N个节点、m条边的图将处理器网络模型化。每个处理器的边都被局部地命名，因此一个处理器可以辨别出它已经从哪一条边接到信息，并且能够选择通过哪一条边来发送信息。处理器有不同的标识，但是这些标识是不能解释的数字，也没有拓扑结构的意义。我们假设了一个异步网络。

11.1.1 方向侦听的定义和特性

尽管近年来方向侦听引起相当关注，但是关于它的概念还没有统一定义。文献[Tel94]定义了几个例子。方向侦听的类别可在[FMS89]中找到。

1. 群定义

我们按照群论方法求解问题。首先回忆群论的一些概念。对于某些学生，由于不熟悉这个理论，有些定义似乎略显神秘，但是在可能的地方，我们将给出具体结构的明确例子。

可交换群或者阿贝尔（abelian）群是集合G，有一个特殊的零元素0，和一个二元算子+，满足以下要求。

（1）闭性　对于所有x，$y \in G$，$(x+y) \in G$。

（2）恒等性　对于所有$x \in G$，$0+x=x+0=x$。

（3）可逆性　对于所有$x \in G$，存在$y \in G$，满足$x+y=y+x=0$。

（4）可结合性　对于所有x，y，$z \in G$，$(x+y)+z=x+(y+z)$。

357 （5）可交换性　对于所有x，y，$x+y=y+x$。

由于它的可结合性，我们在求和中将省略括号。用$-x$表示x的逆，如果$s \in \mathbb{Z}$，那么$s \cdot x$表示s个x的和。

G中元素的个数称为它的阶（order），用ord(G)表示。在方向侦听的应用中，假设ord(G)有限。在有限群中，对于每个x，存在正数k，满足$k \cdot x = 0$。最小的这种正数是x的阶。它总是对群的阶进行划分。

对于元素g_1，…，g_k，考虑能表示成g_i的和的元素集合：

$$S=\left\{x:\exists s_1,\cdots,s_k:x=\sum_{i=1}^{k}s_i\cdot g_i\right\}$$

这个集合本身是一个群，称为由g_1至g_k生成的子群，用$< g_1$，…，$g_k >$表示。对$y \in G$，集合$T=S+y=\{y+x:\ x \in S\}$被称为S的轨迹（orbit）或者g_1到g_k的轨迹。S的所有轨迹大小相等，轨迹$S+y_1$和$S+y_2$要么相等，要么不相交，因此S的轨迹划分G。

称群是循环的（cyclic），如果它由一个元素生成，即存在一个元素g，满足$G=<g>$。生成器不是惟一的（除了阶为2的群的情况）。但是我们通常固定生成器，并称它为1。用i表示$i \cdot 1$，用$\mathbb{Z}_k$表示阶为k的循环群。

因此，$\mathbb{Z}_k$简单的表示模k数的群。它的元素为0到$k-1$的数。另一个值得特别关注的群是群$(\mathbb{Z}_k)^d$；它的元素是长为d的向量，群的操作是逐点增加（模k）。

2. 网络和标号

设G表示可交换群，对于近邻p和q，设$\mathcal{L}_p(q)$表示在p处链路pq的名字。群方向侦听的概念就是网络的拓扑结构与群的结构相匹配。

定义11.1　*如果边标号是G中的元素，称对边标号$\mathcal{L}$为方向侦听（基于G），并且存在从节点到G的单射N，满足对于所有近邻p和q，$\mathcal{N}_q=\mathcal{N}_p+\mathcal{L}_p(q)$。*

给定方向侦听$\mathcal{L}$，按照定义中的规定，对节点标号称为$\mathcal{L}$的证据标号（witnessing labeling）或者证据（witness）。处理器知道链路标号$\mathcal{L}$，但是证据节点标号不要求或者不假设为处理器所知。且证据节点标号不是方向侦听的一部分。容易证明[Tel95]，证据节点标号不是惟一的，任何给定的证据节点标号都可以通过把一个固定元素$s \in G$添加到每个标号上来
358 修改。因此方向侦听的证据标号数恰好为ord(G)。

无需参考对节点的标号，也能描述方向侦听的特点，即，用封闭路径性质进行刻画，参见习题11.1。

观察可见，SoD并不把标号0作为边的标号（因为近邻节点标号不同），并且SoD满足反对

称（anti-symmetry）性质 $\mathcal{L}_p(q)+\mathcal{L}_q(p)=0$，或等价于，$\mathcal{L}_p(q)=-\mathcal{L}_q(p)$。通常，群的阶等于节点数，因此证据标号是双射的。对于我们所给出的大多数算法，这个性质是不重要的。但我们常常隐含地作出这个假设。

给定N个处理器的网络和阶为N的群G，可以很容易地构造方向侦听。如果先用G中的不同元素任意对节点标号，然后对于每条边pq，取 $\mathcal{L}_p(q)$ 为 $\mathcal{N}_q-\mathcal{N}_p$，就得到一个方向侦听。

对于某些任务，正如后面将要表明的那样，加上标号可以大大降低复杂度。但是大多数结果是非常有效的标号都满足另外一个性质：一致性。

3. 一致性

通常并行计算机中的处理器结构是对称且已知的（即，从所有处理器来看，图“看起来是一样的”）。对称性和拓扑结构知识隐含在一致方向侦听中。

定义11.2 *如果每个处理器具有相同的局部标号集，那么方向侦听是一致的。*

设L是链路标号的公共集合，直接可得一致方向侦听的一些性质。

（1）如果$g\in L$，那么$-g\in L$：标号为g的链路在另一端的标号为$-g$。

（2）L生成G，因为对于每个$g\in G$，有满足$SUM_{\mathcal{L}}(P)=g$的路径P存在。

（3）对于$g\in G$，由*钱币兑换问题*的一般情况，就可局部地计算出满足$SUM_{\mathcal{L}}(P)=g$的一条最短路径P。并可计算出L中和Hg的最小标号序列，且因为每个处理器都有这些标号，这个序列定义了网络中的一条路径。

我们用一个处理器上的标号集合描述具有一致方向侦听的网络，常常省略相反情况。 359

4. 例子

文献中所找到的方向侦听的例子，其中大多数是我们所做定义的实例。Flocchini等人[FMS98]把*弦*（chordal）方向侦听定义为$\mathbb{Z}_N$上的方向侦听。特别令人感兴趣的是一致弦SoD，其中1（或者任何其他生成器）在标号中。大小为N的*弦环*（chordal ring）就像环，但在每个节点上，另有一个“短割”指向环中一定步数之外的节点。

定义11.3 *弦环$C_N(c_1,\cdots,c_k)$是一个网络。对于这个网络，存在一致弦方向侦听，且*

$$L=\{-c_k,\cdots,-c_1,-1,1,c_1,\cdots,c_k\};$$

这里k称为弦数（number of chords）。

要求群的生成器作为标号出现是至关紧要的。例如，对于有环绕且k、l互质的$k\times l$圆环，它具有一致的弦方向侦听，但不是一个弦环。在群的框架下，我们现在重新定义一些已知的拓扑结构。

定义11.4 *一个$n\times n$的圆环是一个网络，对于这个网络，存在$(\mathbf{Z}_n)^2$上的一致SoD，且*$L=\{(0,1),(0,-1),(1,0),(-1,0)\}$。*（这些标号方便地称为**N**，**S**，**E**，**W**。）*

定义11.5 *一个n-维的超立方体是一个网络，对于这个网络，存在基为$(\mathbf{Z}_2)^n$的一致SoD，且对于$i=0,\cdots,n-1$，用$i=(0,\cdots,1,\cdots,0)$表示n个标号。*

定义11.6 *一个团是一个网络，对于这个网络，存在基为$\mathbb{Z}_N$的一致SoD，且$L=\mathbb{Z}_N\setminus\{0\}$。*

11.1.2 利用方向侦听

我们现在给出几种关键技术，这些技术作为利用方向侦听的基础。在本章的最后，将会给出完整的算法，用以说明和阐述这些技术。

1. 路径比较

给定两条路径π_1和π_2，起点相同。从路径上的标号可以判定，是否它们结束于同一点。实际上，所有我们需要做的就是，把路径上的标号加起来，当且仅当标号之和相等，路径的终点相同。

这给了方向侦听一个等于近邻知识的优点，并且如6.4.3小节，可进行深度优先搜索遍历，
360 消息复杂度为$O(N)$。要使用这项技术，方向侦听无需是一致的。

路径比较的可能性对于方向侦听是如此基本，以至于Flocchini等人[FMS98]曾提议把它作为方向侦听的定义。在他们的方法中，恰当的标号伴随着明确的平移函数，该函数用来计算一条路径上终点的相对位置。一方面，这个定义更具一般性，因为它描述了更大一类标号。然而，它的优点是有限的，更大一类的标号包括了近邻知识（在我们的定义中，这并不是方向侦听的例子），以及一些在分布式算法的设计中几乎不用的标号。另一方面，群组方法获得了简明的平移函数（如，群元素的相加），因此使得人们可以更专注于方向侦听的更高一级使用上。

2. 路径压缩

如果方向侦听是一致的，它也蕴含着网络拓扑结构的完整知识。实际上，在这种情况下，考虑方向侦听的可用性不能独立于对拓扑结构的全面了解。利用前面提到的通用钱币交换问题，一致方向侦听可使到达网络中相对位置已知的任一节点的路由更有效。

3. 网络结构

一致方向侦听可以充分利用网络的群组结构，来指导算法的整个进程。我们以广播算法以及超立方体上的选举算法（11.3.2小节）为例加以说明（图11-2所示算法）。

11.1.3 具有方向侦听的广播

因为我们的主要目标之一是比较具有方向侦听和没有方向侦听时广播算法的复杂度。现
361 在将要证明，方向侦听可以使广播算法的消息复杂度为$O(N)$。如果没有拓扑信息可用，广播要求至少通过每个信道交换一条消息（定理6.6），因此消息复杂度为$\Omega(|E|)$。

1. 利用方向侦听进行深度优先搜索

Mans和Santoro观察到，利用近邻知识的深度优先搜索算法（6.4.3小节）可以很容易地推广到具有方向侦听的网络上。回忆一下图6-18所示算法，它循环传递令牌，令牌中包含已访问的处理器的名单。可以完全避免通过后向边发送令牌，因为转发令牌的处理器会检查这张表，以确定是否近邻名字就在令牌中的这张表里。

图11-1所示算法表明了如何利用方向侦听，而不是近邻知识进行深度优先搜索。在令牌中循环的这张表不再包括处理器名（并不假设这些处理器名是可用的），但却标明了相对于消息持有者的处理器的位置。算法利用了“路径比较”能力：处理器q将自身加入列表，并总是将数0插入列表。在标号和为σ的路径π上将这张表转发给处理器p后，0已经被变成$-\sigma$。此时，$\mathcal{L}_q(p)=\sigma$，且$\mathcal{L}_p(q)=-\sigma$。因此在表中可以找到边pq上的标号。

节点一旦收到这张列表，则将接收消息所经过的链路标号相加，完成相对位置的更新。这是对图6-18所示的算法惟一所作的增加。

这个算法不仅可用于广播算法，而且可用于Kutten等人（7.4节）提出的选举算法。其中所产生的算法的消息复杂度为$O(N \log N)$。结果，我们得到了第一个方向侦听的成功算法。

因为对于没有SoD的广播和选举，我们已有好的下界（定理6.6和定理7.15）。我们强调方向侦听不必是一致的，因此利用路径比较能力足以获得这个结果。

推论11.7 在没有拓扑信息的网络中，方向侦听的可用性使得广播算法的复杂度从$\Theta(m)$下降到$\Theta(N)$，选举算法的复杂度从$\Theta(N \log N + m)$降低到$\Theta(N \log N)$。

```
var father_p : process  init udef ;

For the initiator only, execute once:
    begin father_p := p ; choose q ∈ Neigh_p ;
          send ⟨ tlist, {0} ⟩ to q
    end

For each process, upon receipt of ⟨ tlist, L ⟩ through link λ:
    begin forall x ∈ L do x := x + λ ;
          if father_p = udef then father_p := λ ;
          if ∃q ∈ Neigh_p \ L
            then begin choose q ∈ Neigh_p \ L ;
                       send ⟨ tlist, L ∪ {0} ⟩ to q
                 end
            else if p is initiator
              then decide
              else send ⟨ tlist, L ∪ {0} ⟩ to father_p
    end
```

图11-1 具有SoD的深度优先搜索算法

2. 一致SoD的结构算法

一致方向侦听可使处理器决定处于“就绪”状态的网络的一棵生成树，并且通过这棵生成树发送广播消息，只用$N-1$条消息就实现广播。这个数是最优的，因为与广播的初始进程不同的$N-1$个处理器，其中的每一个必须通过接收一条消息获悉信息。

我们首先给出关于该算法的一个基本例子：假设网络为$n \times n$的圆环，其中边被标号为North、East等。广播如下进行。初始处理器先把消息发向East，然后消息在东的方向被转发$n-1$个跳数。随后从West方向接收消息的初始处理器和每个节点向North方向转发消息，消息在那个方向只进行$n-1$步传播。

显而易见，这种策略可行，且只利用$N-1$条消息。通过增加另外$N-1$条确认消息，算法实现了具有反馈（PIF）机制的广播。假如我们进一步把消息向East方向转发一步，初始进程就会是下一个接收者，这一步是多余的。假如消息被进一步向North方向转发一步，就会被初始时向North方向发送它的处理器接到，这一步也是多余的。

例子表明，一致方向侦听是如何局部地提供关于整个网络结构的信息。这个信息可以被算法使用。我们用本算法更一般的形式继续讨论。其中基于网络的群可以是任意的。可以按任意次序调用标号g_1到g_k（省略相反次序），n_1到n_k定义如下。首先，设n_1为g_1的阶；接着，设n_2为g_2在$G/\langle g_1 \rangle$中的阶，n_2是满足$n_2 \cdot g_2$是g_1倍数的最小数。进而，设n_i为g_i在$G/\langle g_1, \cdots, g_{i-1} \rangle$中的阶，$n_i$为使$g_i$表示成$\{g_1, \cdots, g_{i+1}\}$中元素和的最小数。$n_i$具有以下性质：它们的乘积等于$N$，$G$中的每个$g$惟一地记为

362
~
363

$$g = \sum_{i=1}^{k} s_i \cdot g_i \quad \text{且} 0 \leqslant s_i \leqslant n_i$$

图11-2所示的算法将信息沿着g_1方向发送，经过n_1-1次跳跃。为所有与初始处理器成g_1倍数的处理器提供消息。接到$-g_1$方向消息的初始处理器和所有处理器向g_2方向转发该消息，经过n_2-1次跳跃。为那些与初始处理器成g_1加g_2倍数的处理器提供消息。总之，初始理器和所有处理器从方向$-g_1$到$-g_{i-1}$接到消息，并沿g_i方向转发该消息，经过n_{i-1}次跳跃。因为不同于初始处理器的每个处理器只接收一次消息，因此图11-2所示的算法进行$N-1$次消息交换。

如果需要，加上$N-1$次反馈消息就可将这个算法变成PIF算法。

如果需要，非-初始处理器也能存储链路，通过这条链路，它们接收消息，来建立以初始处理器为根的生成树。这棵树可用于将信息发送回初始处理器的信息，参见11.3.2小节。算法中生成树的深度为$\sum_{i=1}^{k}(n_i-1)$。如果需要，这个深度可也受到这样一个次序的影响，按照这个次序g_1，…，g_k列出L中标号。在此将不进一步讨论这个问题（它与广播的时间复杂度有关）。

```
For the initiator:
    for i = 1 to k
        do if n_i > 1 then send ⟨info, n_i − 1⟩ via link g_i

Upon receiving ⟨info, s⟩ via link −g_j:
    if s > 1 then send ⟨info, s − 1⟩ via link g_j ;
    for i = j + 1 to k
        do if s_i > 1 then send ⟨info, s_i − 1⟩ via link g_i
```

图11-2　具有一致方向侦听的广播算法

11.2　环和弦环的选举算法

环和弦环上的选举问题有很长的历史，参见7.2节。11.2.1节提出了Franklin方法[Fra82]，
364 后来Attiya等人[ALSZ89]通过使用被加到环上（一致地）的弦，改进了该算法。他们的算法利用了路径压缩技术，并且只要少量的弦就可达到线性复杂度，参见11.2.2节和11.2.3节。然而，这个算法只利用了路径压缩技术，而没有利用网络结构。在11.2.4节中表明，如果利用了网络结构，还可以使用更少的弦，同样可以得到线性算法。

11.2.1　Franklin算法

Franklin算法将处理器分为*活动*处理器和*转播*（relay）处理器，并且利用一连串的轮。在第一轮循环之前，每个处理器都是活动的。开始轮时活动处理器数多于一个，在一轮循环中，至少有一半、至多有总数-1个处理器要变成转播处理器，接着进行下一轮。只有一个活动处理器开始的这轮，会选择这个处理器，这也是最后一轮。因此，轮次数至多为$\lfloor \log N \rfloor + 1$，因为每一轮只有$2N$条消息，最坏情况下消息复杂度大约是$2N \log N$。

在每一轮中，活动处理器与左边的最近活动处理器和右边最近活动处理器交换名字。如果一个处理器是它自己的“最近活动处理器”，则它就是惟一的活动处理器，并且当选。否则，仅当它的名字大于它的左右“活动近邻”的名字时，它在该轮循环中存活。至少有一个处理器（最大活动处理器）和至多有半数处理器（由于在一个“生存者”的两个近邻中失败）是为真的条件。具有较大最近活动近邻的活动处理器变成转播处理器。转播处理器转发它们在相反方向所接收的所有消息。这一点在图11-3中没有表明。

```
state_p := active ;
while state_p = active do
      begin send ⟨name, p⟩ to left ; send ⟨name, p⟩ to right ;
            receive ⟨name, x⟩ from right ; receive ⟨name, y⟩ from left ;
            if x = p and y = p then state_p := elected ;
            if x > p or y > p then state_p := relay
      end
```

图11-3 FRANKLIN环选举算法

365

11.2.2 Attiya改进

在Franklin算法中，*活动*进程发送的消息数是线性的，图11-4所示的算法通过转播处理器消除尽可能多的消息。

随着每轮中活动处理器数减半，对lg$N$轮中的活动处理器数*求和*，其值限定为$2N$。因为在一轮中，一个活动处理器发送两条消息，活动处理器发送的消息数是线性的：$4N$。然而，消息被连续转播较大的距离，这是因为在第i轮之后，连续的活动处理器至少相隔2^i个位置。总之，由被动处理器转播的消息几乎占据了整个消息复杂度的主要部分。

在环网上，这是不可避免的。即使活动处理器知道距下一处理器的距离，消息也必须通过环中的节点转发，因为这是最短路径。Attiya算法保留了Franklin算法的基本控制结构，但是保证活动处理器知道距下一处理器的距离，并且利用弦和路径压缩来快速转发消息<**name**, p>。弦不被用于修改算法的基本竞争结构。

```
state_p := active ; Left_p := −1 ; Right_p := +1 ;
while state_p = active do
   begin Send(⟨name, p⟩, Left) ; Send(⟨name, p⟩, Right) ;
         Receive(⟨name, x⟩, Right) ; Receive(⟨name, y⟩, Left) ;
         if p ⩾ x and p ⩾ y then
           begin Send(⟨pos, 0⟩, Left) ; Send(⟨pos, 0⟩, Right) ;
                 Receive(⟨pos, r⟩, Right) ; Right := Right + r ;
                 Receive(⟨pos, l⟩, Left) ; Left := Left + l ;
                 if Left_p = 0 then state_p := elected
           end
         else
           begin (* relay the ⟨pos, .⟩ messages *)
                 Receive(⟨pos, r⟩, Right) ; Send(⟨pos, r + Right⟩, Left) ;
                 Receive(⟨pos, l⟩, Left) ; Send(⟨pos, l + Left⟩, Right)
           end
   end
```

图11-4 ATTIYA弦环选举算法

我们现在考虑具有方向侦听的弦环网络。在每轮的一开始，一个活动处理器知道它左边的最近活动处理器和右边最近活动处理器的相对位置，并利用弦发送它的名字。假设用通信原语*Send*(*message*, g)和*Receive*(*message*, g)实现路径压缩技术。第一个原语经过$SUM_c = g$的路径转发消息，第二条原语接收这个消息。一种可能的实现是*贪婪路由*（greedy routing）方法，它总是通过不超过目的处理器的最大弦来发送消息。参见图11-5所示的算法。当然，也可利用更复杂的算法选择边，但其结果，对选举算法复杂度所产生的影响，微乎

其微。

```
procedure Send(m, g):
  begin if g = 0
          then deliver message m to waiting Receive
          else begin ch := the largest of g_i s.t. ch ⩽ g ;
                       send ⟨m, g − g_i⟩ through link ch
               end
  end

Upon receipt of ⟨m, g⟩:
  Send(m, g)
```

图11-5　贪婪路由策略

初始时，左右活动近邻的相对位置分别为-1和+1 。在一轮结束时，生存下来的活动处理器通过在该轮中变成转播的一连串的处理器，传递<**pos**，.>消息，寻找最近活动处理器的相对位置。如果在某一轮循环结束时，只有一个处理器存活，则处理器发现它是自己的近邻（$Left = 0$），于是找出领导人。

新算法与Franklin算法的不同之处在于，它们的活动处理器的通信方式不同。对于活动处理器数所经过的次数求和，仍为$2N$。现在，每一轮中，活动处理器所发送的消息数是4。因此，在整个算法中，*Send*操作的数目为$8N$。

我们首先讨论团的特例。这种情形下的算法最早由Loui等人提出[LMW86]。因为网络是完全链接的，*Send*操作直接路由消息。这只花费一次消息传递，整个消息复杂度为$8N$。这是够幸运的，因为Korach等人[KMZ84]已经给出了没有方向侦听的团上的选举算法，其消息复杂度下限为$\Omega(N \log N)$。

推论11.8　*在一个团上，方向侦听的可用性使得选举算法的复杂度从$\Theta(N \log N)$降低到$\Theta(N)$。*

对于一般的弦环，Attiya将*消息代价函数*F（message cost function F）定义为最小单调凸函数，对于这个函数，*Send*(m, g)要求至多交换$F(g)$条消息。（非形式地，F表示向相隔跳数为g的一个处理器发送一条消息的代价。）在最坏情况下，第i轮开始时，有$N/(2^{i-1})$个活动处理器，相隔2^{i-1}个跳数，因此利用$\frac{N}{2^{i-1}} \times 4 \times F(2^{i-1})$条消息。

引理11.9　*图11-4所示的算法在最坏情况下，利用$4N \cdot \Sigma_{i=0}^{n-1} \frac{F(2^i)}{2^i}$条消息。*

引理表明，选举算法的整体复杂度取决于环上发送消息的代价。作为特例，环（$F(g) = g$）上的消息复杂度和团（$F(g) = 1$）上的消息复杂度分别为$O(N \log N)$和$O(N)$。Attiya等人已经证明，只需对数条弦，就可使得和为常数。例如在弦环上，$C_N(2, 3, 4, \cdots, \lg N)$。这里不再重复计算。

11.2.3　最小化弦数

Attiya算法确定了弦环上的选举算法复杂度为线性，现在我们把注意力放到获得该复杂度所必需的最小弦数。本节中，我们利用数学公式的“向后图工程”，确定为Attiya算法渐近地需要的弦数。结果表明了算法研究的一种趋势：就是变得越来越不“算法化”，而更多关注分析。一旦处理了选举的算法方面的问题，弦和复杂度的关系已“封装”在公式中（引理11.9），

我们的工作集中在对公式的处理方面。

1. 和是几何级数

由于几何级数（增长率小于1）和有界，首先研究多少条弦才能使 $\frac{F(2^i)}{2^i}$ 的界限为c^i，其中c < 1。

考虑弦的序列L = {2，4，16，256，…}，其中每后一条的弦是前一条弦的平方；当下一个数超过N时，序列结束。弦具有2^{2^i}的形式，因此，共有为log log N条弦。 366 ~ 368

引理11.10 $F(d) \leqslant 2\cdot\sqrt{d}$ 。

证明。对d用归纳法证明。证明使用贪婪路由。和为d的路径从与d相匹配的最大弦l开始，在长为$d-l$的路径上继续下去。首先，$F(1) = 1 \leqslant 2\sqrt{1}$ 。

现在假设$d > 0$ 但对于$d' < d$，结果为真。弦的构造蕴含着，与d匹配的最长的弦长度为l，满足 $\sqrt{d} < l \leqslant d$，可得，$F(d) \leqslant 1 + F(d-l) \leqslant 1 + 2\sqrt{d-l} < 1 + 2\sqrt{d-\sqrt{d}} < 2\sqrt{d}$ 。 □

因此，$\frac{F(2^i)}{2^i}$ 界限为 $2\cdot\left(\frac{1}{\sqrt{2}}\right)^i$ 。引理11.9中的和变成一个几何级数，而且和有界。我们已经证明，用O(log log N) 条弦，进行线性选举是可能的。由此可得，要使和为几何级数，必需使用Ω(log log N) 条弦（如果使用贪婪路由）。

引理11.11 如果存在常数$c < 1$，满足利用贪婪路由 $\frac{F(2^i)}{2^i} < c^i$，那么弦数为Ω(log log N)。

证明。对于每个i < log N，$F(2^i) < c^i \cdot 2^i$，这蕴含着，要用弦长至少为 $(1/c)^i$的弦，而且由贪婪路由策略，这条弦至多为2^i。因此，对于每个i，存在一条长度位于区间[$(1/c)^i\ i \cdots 2^i$] 内的弦。观察可见，如果$i' = i\cdot(^{1/c}\log 2)$，则$i'$和$i$的区间是不相交的。因此，对于$(^{1/c}\log 2)\log(^2\log N)$ 个不同的值$i_r = (^{1/c}\log 2)^r$ （这里$r < (^{1/c}\log 2)\log(^2\log N)$），我们得到一个不相交区间的集合[$(1/c)^{i_r} \ldots 2^{i_r}$]，每个区间包含一个弦长。 □

采用其他的路由策略，不会对结果做出超过一个常数因子的改进。对此的证明仍然是一个未解决的问题。从L中确定F与钱币交换问题有关，该问题要求使用给定的由不同币值硬币组成的集合中的硬币，来支付一定数额。在钱币交换问题中，一般的路由策略意味着，允许利用钱币返回。

2. 更慢的递减求和

在引理11.9中的几何级数给出了线性复杂度，但却要求Θ(log log N) 条弦才能实现这个结果。因此，我们可以选择一个下降更慢的序列，但是这个序列和仍然有界。调和平方级数Σ $(1/i^2)$ 就是所选序列。

假设弦长集合L = {36，64，256，65536，…}，其中$g_{i+1} = 2^{\sqrt{g_i}}$。36是该公式所给出2的更大次幂的最小数。由于$g_i = \log^2(g_{i+1})$，对于每个d，存在一条弦长介于$\log^2 d$和d之间的弦。

引理11.12 L中的弦数少于2 log* N。

证明。因为 $g_{i+2} = 2^{\sqrt{2^{\sqrt{g_i}}}} = 2^{(\sqrt{2}\sqrt{g_i})} > 2^{g_i}$ ，序列g_0，g_2，g_4，g_6，…要比序列2，2^2，2^{2^2}，$2^{2^{2^2}}$，…增长更快，超过N的log*N项。 □ 369

为了在短距离（$d < 36$）上有效地进行消息路由，需要添加一些短弦，但这仅是一个常数。

引理11.13 $F(d) \leqslant 2\cdot\frac{d}{\log^2 d}$ 。

证明。对于较大d，贪婪路由算法首先选择弦长大于$\log^2 d$的弦l，且实现

$$F(d)=1+F(d-l)<1+2\cdot\frac{d-l}{\log^2(d-l)}<1+2\cdot\frac{d-\log^2 d}{\log^2(d-\log^2 d)}<2\cdot\frac{d}{\log^2 d}$$ □

计算表明，要获得选举问题的线性复杂度，$\Theta(\log^* N)$条弦就足够了。我们将给出与之相匹配的弦数下界。证明是基于求和过程，该求和过程下降更慢，且有无限和：调和级数$\Sigma(1/i)$。这个序列的前$\log n$项的总和为$\Theta(\log\log N)$。然而，要达到这个结果，需要$\Omega(\log^* N)$条弦。

引理11.14 *采用贪婪路由，如果* $\frac{F(2^i)}{2^i}<1/i$，*那么至少存在*$\log^* N$*条弦。*

证明。类似于前一个关于下界的证明。对于每个i，一定存在介于i和2^i之间的弦。 □

可得：

（1）$O(\log^* N)$条弦是保证调和平方级数收敛（引理11.3）的充分条件。

（2）$\Omega(\log^* N)$条弦是使得调和级数（引理11.4）发散的必要条件。

为了得到线性复杂度，必须对引理11.9中的和加以限制。因此如果要比调和级数下降更快，至少利用$\Omega(\log^* N)$条弦。这蕴含着，$\Theta(\log^* N)$条弦是使Attiya算法具有线性复杂度的充
370 分必要条件。

11.2.4 1-弦线性算法

本小节提出了一个线性弦环选举算法，该算法仅需一条弦。我们已经看到，这在Attiya算法中是不可能的。因为在Attiya算法中，只用了路径压缩方法，没有利用网络拓扑结构。

现在假设弦环只有一条长为t的弦，且t近似为$\sqrt{N}$。只有一条弦长近似为$\sqrt{N}$的弦环在拓扑结构上像圆环。我们的算法是由Peterson[Pet85]的圆环算法改进而来。

当新一轮开始时，活动处理器试图寻找同一轮中的另一个活动处理器。一个处理器是最多可看见一个其他处理器，但是可能被一个以上的处理器看见。有两种方式可使活动处理器被提升到下一轮循环中。

第一种方法类似于Franklin算法中的“局部最大”提升：如果处理器看见一个更小的处理器，并且也被一个更小的处理器看见，则该处理器能被提升。按照这个规则，至多有一半的处理器可以被提升。因为如果一个处理器被提升，至少一个看见它的较小处理器将不能被提升。

每个活动处理器搜索网络，直到找到另一个活动处理器，代价太大。因此，活动处理器只搜索网络的一个受限的子网络，没有找到其他的活动处理器，搜索可能终止。在这种情况下，第二种提升规则阐明，正在搜索的处理器被提升到下一轮循环中！

算法的关键成就在于以这样一种方法设计搜索的过程。（1）搜索区域足够大，以至于按照第二个规则，只有几个处理器被提升；（2）搜索区域足够有效，可得到较好的整体复杂度。现在想象一下在一个无限的网格上画一个越来越大的方形，可以观察到在边界上的点数要比内部的点数增长更慢。

1. 方形和边界，搜索过程

对于$g\in\mathbb{Z}_N$，定义g的l-方形为集合$S_{g,l}=\{g+i\cdot 1+j\cdot t:\ 0\leqslant i\leqslant l,\ 0\leqslant j\leqslant l\}$。$S_{g,l}$是通过交叉至多$l$条1-边和至多$l$条$t$-边，由$g$可达的（至多）$(l+1)^2$个处理器组成的集合。一个活动

处理器在l-方形内搜索其他活动处理器，但是并不向方形内的所有处理器发送消息，因为代价太大。

定义边界（boundary）$B_{g,l}$为点的集合，其中至少有一个不等式中是等式，即，$B_{g,l}=\{g+i\cdot 1,\ g+i\cdot 1+l\cdot t,\ g+i\cdot t,\ g+l\cdot 1+i\cdot t:\ 0\leqslant i\leqslant l\}$。定义内部（internal）点的集合$I_{g,l}=S_{g,l}\setminus B_{g,l}$。边界$B_{g,l}$包含（至多）$4l$个处理器。如果$S_{g,l}$和$S_{h,l}$相交，那么$B_{g,l}$和$B_{h,l}$相交。 371
图11-6，以弦长为8的弦环为例说明3-方形。

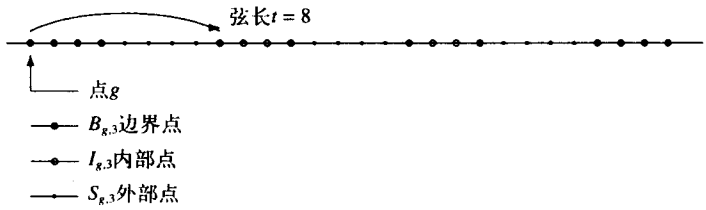


图11-6 方形和边界

在第i轮中的处理器，通过沿着l-方形的边界，发送一个搜索令牌，开始寻找另外的处理器。令牌试图沿方向1行进l步，沿方向t行进l步，沿方向-1行进l步，沿方向$-t$行进l步。方形的长度依赖于轮次数。在第i轮中，长度选为$l_i=\alpha^i$（其中α是比1稍大的常数）。遍历整个边界需要花费$4l$条消息，但是有几个原因可以中断这个遍历。

如果p的令牌进入一个已被一个具有更大轮数的令牌访问过的处理器，就中止遍历，p就永远不能接到搜索令牌的返回消息，也就永远不能进入下一轮。如果p的令牌第一次进入一个已被同一轮循环的不同令牌（如，q）访问过的处理器，p“看见”q。如果q小于p，处理器p必须变成已知。因为看见q对于p是至关重要的，因为按照规则1，它可以被提升。如果q大于p，p就失去了被提升的机会，但是因为被p看见对于q按照规则1被提升是至关重要的，因此q必须被告知。因此或者令牌返回p，或者通过q的令牌所遍历的边界，将追逐令牌发送到q以通知q它已被更小处理器看见。现在，当q被告知，一个更小的处理器看到了它时，已经转发追逐令牌给q的处理器将不再转发p的追逐令牌。

描述令牌处理的过程，蕴含着如下提升p到下一轮的局部条件。首先，p的搜索令牌返回时，已经看到一个较小处理器，并且追逐令牌到达，报告p被一个较小的处理器看见。其次，p的搜索令牌返回，没有看见另一个处理器。

我们要证明，每一轮中至少提升一个处理器。如果没有处理器看见另一个处理器，所有 372
处理器都被提升。如果至少有一个处理器看见另一个处理器，设r是被看见的最大处理器。或者r没看见处理器，按照第二个规则，它被提升，或者看见一个较小的处理器，按照第一个规则，它也被提升。

2. 一些计算

我们的算法非常近似于Peterson的算法[Pet85]，并且计算也相同。设A_i是第i轮开始时的活动处理器数，则有$A_0\leqslant N$。我们首先对于所有轮中活动处理器数A_i进行限制，方形大小$<N$，即，循环轮数$i<{}^{\alpha}\log\sqrt{N}$。

引理11.15 对于所有轮，如果$i<{}^{\alpha}\log\sqrt{N}$，则有$A_i<\dfrac{N}{\alpha^{2i}(2-\alpha^2)}$

证明。我们已经证明，每个按照第一个规则被提升的处理器被另一个未被提升的处理器看见。因此，最多一半的看见另一个处理器的处理器，按照这个规则被提升。

某些处理器可能按照第二个规则被提升（没有看到另外的处理器），但是如果p和q都按此规则得到提升，则$S_{p,l}$和$S_{q,l}$不相交。如果方形重叠，p和q的令牌遍历的边界也重叠。当其中的一个令牌首先到达交叉点，使得另一个令牌由于没有看到另一个处理器而不可能完成遍历。因此按照这个规则被提升的处理器数少于N/l^2_i。

由上所述及以前的选择$l_i = \alpha^i$，可得

$$A_{i+1} \leqslant \frac{A_i + N/\alpha^{2i}}{2}$$

利用对i的归纳法，可以验证，上式蕴含着$A \leqslant \frac{N}{\alpha^{2i}(2-\alpha^2)}$。 □

引理蕴含消息数为线性界，以及达到轮$^{\alpha}\log\sqrt{N}$时，处理器数常数。

推论11.16 *在第0轮到第$^{\alpha}\log\sqrt{N}$轮的中，交换的消息数少于$\frac{8\alpha}{(\alpha-1)(2-\alpha^2)}N$。*

证明。引理11.15已经确定了第i轮中活动处理器的数目。消息计算如下。活动处理器p的搜索可以进行$4\alpha^i$步，归于p。向p路由的追逐令牌也归于p。由p的搜索所访问的每个处理器至多转发一次追逐令牌，因此至多$4\alpha^i$个追逐步归于p。

373 因此在第i轮中，$\frac{N}{\alpha^{2i}(2-\alpha^2)}$中的每个处理器至多交换$8\alpha^i$条消息。因此第$i$轮中的消息总数界限为$\frac{8N}{2-\alpha^2}\cdot(1/\alpha)^i$。对轮数求和，得到一个和为$\frac{8N}{2-\alpha^2}\cdot\frac{\alpha}{\alpha-1}$的几何级数。 □

3. 算法的终止性

引理11.15蕴含着，在第$^{\alpha}\log\sqrt{N}$轮中，只有少于常数，即，$1/(2-\alpha^2)$个处理器能够存活下来。为了在这些处理器之间进行选举，算法在那轮之后继续进行第二步，如同Chang-Robert算法（7.2.1小节）对其进行组织。每个在第$^{\alpha}\log\sqrt{N}$轮中存活下来的处理器沿着整个环发送令牌，即，在+1的方向进行N步。令牌可被具有更大标识且在那轮中存活下来的令牌删除，并且在算法的第一步中，中止处理器的活动。在第二步中，保证只有在第$^{\alpha}\log\sqrt{N}$轮中存活的最大处理器接收它的返回令牌，则该处理器被选中。

定理11.17 *具有一条长为$\sqrt{N}$的弦的弦环上算法选举一个领导人的消息复杂度为$O(N)$。*

证明。在第一步中，没有一轮会杀死所有活动处理器，至少有一个处理器存活到第二步。第二步保证这些处理器中只有一个被选，这就确定了算法的正确性。

由推论11.16可得，第一步利用$O(N)$条消息，第二步只利用$O(N)$条消息，这是因为在那一轮中，至多常数个处理器初始一个令牌。 □

11.3 超立方体上的计算

这一节，我们研究是否方向侦听会减小在超立体方体上广播和选举的复杂度。答案是肯定的，因为已知的利用方向侦听的算法已经胜过无标号超立方体上的已知的最好算法。然而，因为对于未标号的情形，还没有相匹配的下界，因此还没有得出肯定的结论。

近年来，人们提出了未标号超立方体上的有效算法。弥补了有标号和无标号情形下已知复杂度之间的差距。11.3.1小节考虑的是没有方向侦听、没有拓扑结构知识的网络。即，我们

考虑在每个网络上都正确的算法。并研究它们在超立方体上的复杂度。在11.3.2节，我们的研 374
究是建立在假设有方向侦听和拓扑结构知识的网络上，将会看到，那里存在精致、简单、有效的算法。

研究的难点是有拓扑结构知识而没有方向侦听的情形。即，算法只需在超立方体上正确运行，但对有意义的边标号不做假设。可以证明，利用线性消息，就能进行广播。11.3.3节提出超立方体上的定向算法，11.3.4节引入了掩码技术和广播算法。11.3.5节讨论了对于无标号的超立方体，最近的一些选举算法中使用的技术。我们没有考虑有方向侦听但是没有拓扑结构知识的情况。

11.3.1 基线：没有拓扑知识

在没有拓扑结构知识和方向侦听的情况下，广播算法和选举算法要求网络的每一条边至少携带一条消息。事实上，算法要在每个图中都正确运行，这一点允许使用定理6.6和定理7.15中的“节点插入”技术。因此，当在超立方体上执行时，算法至少利用$\Theta(N \log N)$条消息。

另一方面，通过扩散式路由或者回波算法（图6-5所示的算法）也可进行广播。利用Gallager等人的算法（图7-10、图7-11以及图7-12所示的算法）可以进行选举。因此，两个任务都需要$O(N \log N)$条消息。

定理11.18 *在没有拓扑结构知识的超立方体上进行选举和广播，消息复杂度为$\Theta(N \log N)$。*

11.3.2 进行比赛的算法

在具有方向侦听的超立方体上，进行比赛的选举算法使用超立方体图的递归结构。为了选出d+1维超立方体上的领导人，算法首先在几乎是最后一个生成器生成的d维超立方体的每个面上选出一个领导人，然后选出两个领导人中的一个。为了避免算法的不同阶段所产生的领导人之间相互混淆，如果它赢得一个d维面上的选举，我们称一个节点是*d-领导人*。

本算法的基础情况是在第0维面上的选举，这很简单。网络只由一个节点组成，它就是0-领导人。

1. 锦标赛 375

成为d-领导人（对于$d < n$）的节点与在方向d上的近邻面上的d-领导人进行锦标赛。如果两节点能直接通信，那么它们之间的比赛是容易组织的。每个节点向另一节点发送包含自己名字的消息，接到名字比自己的大的节点成为*非-领导人*，接到名字比自己的小的节点成为*领导人*。

按照同样的方法进行两个d-领导人之间的比赛。但是存在这样的困难，就是节点并不知道如何到达另一个面上（即使它们自己的面）的领导人。作为第一步，成为d-领导人的节点p通过方向d上的边发送一条比赛消息<**tour**，p，d>，在消息中包含自己的名字和步数d。这条消息被另一个d-维面上的节点接收。这个我们称为*入口*（entry）节点的接收节点，其职责是转发消息给它的d-领导人，参见图11-7。

把消息转发给d-领导人的困难是，入口节点并不知道d-领导人的相对位置。如果d-领导人把它的位置通知d-立方体上的所有节点，使得每个入口节点都能以d步转发消息，代价太大。这种通知要求2^d-1条消息，从而导致选举的整个复杂度为$O(N \log N)$。类似地，如果入口节点通过d-立方体广播锦标赛消息，这也需要2^d-1条消息，代价还是太大。

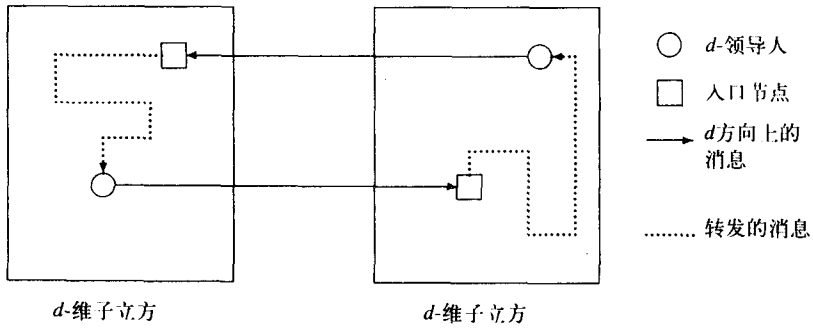


图11-7　锦标赛中的消息转发

我们的解决方法使用了这两个基本策略的组合。d-领导人向在$\lfloor d/2 \rfloor$-维的面上的所有节点发布它是领导人的消息，称$\lfloor d/2 \rfloor$-维的面为领导人的行。入口节点通过$\lceil d/2 \rceil$-维的面广播锦标赛消息，称此$\lceil d/2 \rceil$-维的面为它的列。由于每行每列只相交一点（如下所示），存在一个节点，称它为比赛节点（match node）。它既接收来自d-领导人发布的消息，也接收锦标赛消息。比赛节点经过发布消息所产生的生成树，进一步将锦标赛消息转发给d-领导人。

376

定义11.19　考虑d-维超立方体。

一行是关于生成器0到$\lfloor d/2 \rfloor$-1的一个面。

一列是关于生成器$\lfloor d/2 \rfloor$到d-1的一个面。

两个d-领导人之间的锦标赛组织如下。

（1）一个d-领导人p经过链路d发送消息<**tour**，p，d>。

（2）一个d-领导人利用图11-2所示的算法在它的行上广播它的领导人消息。行中的节点将接收广播消息所通过的链路存储起来，并计算行的一棵生成树。（广播算法采取简单的形式，因为生成器是独立的，每个n_i等于1。跳数计数器可以查禁。）

（3）一个入口节点（即，经过链路d接收<**tour**，p，d>的节点）利用图11-2所示的算法通过它的列广播消息。

（4）在领导人所在行中，接收<**tour**，p，d>消息的非d-领导人q（即，q已经接到<**ann**，d>消息）把消息发送给它的父节点。

（5）接收到<**tour**，p，d>消息的d-领导人q与p进行比较，如果$q > p$，则q变为（d+1）-领导人，否则q变成非-领导人。

2. 通信复杂度

设$T(d)$是在锦标赛中两个d-领导人交换的消息数，$E(n)$是选举算法（在维数为d的超立方体上）中所使用的消息数。由于产生一次选举，要求两个在较小的立方体上的选举和一次锦标赛，我们可得到递归方程。

$$E(n)=\begin{cases}0 & \text{如果} n=0\\ 2\cdot E(n-1)+T(n-1) & \text{否则}\end{cases}$$

在分析消息复杂度时，我们需要知道无穷级数和（$\alpha < 1$），以及关于α的导出结果。

377

$$\sum_{c=0}^{\infty}\alpha^c=\frac{1}{1-\alpha}\text{和}\sum_{c=0}^{\infty}c\cdot\alpha^{c-1}=\frac{1}{(1-\alpha)^2}$$

对两个d-领导人之间的锦标赛，计算其中每一步交换的消息数。可得

步骤1：2条消息。

步骤2：$2\cdot(2^{\lfloor d/2\rfloor}-1)$条消息。

步骤3：$2\cdot(2^{\lceil d/2\rceil}-1)$条消息。

步骤4：至多$2\cdot\lfloor d/2\rfloor$条消息。

求和，可得，$T(d)<2\cdot(2^{\lfloor d/2\rfloor}+2^{\lceil d/2\rceil}+\lfloor d/2\rfloor-1)$。

对于偶数d，$\lfloor d/2\rfloor=\lceil d/2\rceil=d/2$；对于奇数$d$，$\lfloor d/2\rfloor=(d/2)-1/2$且$\lceil d/2\rceil=(d/2)+1/2$；由此可得，

$$\begin{aligned}T(2c)&=2\cdot(2^c+2^c+c-1)=4\cdot 2^c+2c-2,\\ T(2c+1)&=2\cdot(2^c+2^{c+1}+c-1)=6\cdot 2^c+2c-2。\end{aligned}$$

随后，记$F(\mathrm{n})=E(\mathrm{n})/2^n$，利用$E$的递归关系，可得

$$F(n)=\begin{cases}0 & \text{如果}n=0\\ F(n-1)+\dfrac{T(n-1)}{2^n}, & \text{否则}\end{cases}$$

我们可以把$F(n)$记为和，并且以无穷级数为界。

$$F(n)=\sum_{d=0}^{n-1}\frac{T(d)}{2^{d+1}}<\sum_{d=0}^{\infty}\frac{T(d)}{2^{d+1}}$$

现在我们利用上述对于奇数d和偶数d的T的表达式，并将条件分类为指数、线性和常数numerator，可得

$$\begin{aligned}\sum_{d=0}^{\infty}\frac{T(d)}{2^{d+1}}&=\sum_{c=0}^{\infty}\frac{T(2c)}{2^{2c+1}}+\sum_{c=0}^{\infty}\frac{T(2c+1)}{2^{2c+2}}\\ &=\sum_{c=0}^{\infty}\frac{4\cdot 2^c+2c-2}{2^{2c+1}}+\sum_{c=0}^{\infty}\frac{6\cdot 2^c+2c-2}{2^{2c+2}}\\ &=\sum_{c=0}^{\infty}\left(\frac{4\cdot 2^c}{2^{2c+1}}+\frac{6\cdot 2^c}{2^{2c+2}}\right)+\sum_{c=0}^{\infty}\left(\frac{2c}{2^{2c+1}}+\frac{2c}{2^{2c+2}}\right)-\sum_{c=0}^{\infty}\left(\frac{2}{2^{2c+1}}+\frac{2}{2^{2c+2}}\right)\\ &=3\frac{1}{2}\cdot\left(\sum_{c=0}^{\infty}\frac{1}{2^c}\right)+\frac{3}{8}\cdot\left(\sum_{c=0}^{\infty}\frac{c}{4^{c-1}}\right)-\frac{3}{2}\cdot\left(\sum_{c=0}^{\infty}\frac{1}{4^c}\right)\\ &=3\frac{1}{2}\cdot 2+\frac{3}{8}\cdot\frac{16}{9}-\frac{3}{2}\cdot\frac{4}{3}\\ &=7+\frac{2}{3}-2=5\frac{2}{3}\end{aligned}$$

378

由此可得，$E(n)<5\frac{2}{3}\cdot N$。

11.3.3 多路径流量算法

作为研究线性广播算法的基础，我们现在提出一种计算n-维超立方体上方向侦听的算法。该算法，假设一个处理器被指派为算法的初始处理器（领导人），领导人的链路可任意标上0，…，$n-1$中的数。初始处理器的标号惟一地定义了方向侦听，如以下定理所示。

定理11.20 *设w是节点，$\mathcal{P}$是对w的边以0，…，$n-1$中的数标号。那么存在惟一定向$\mathcal{L}$，对于w的每个近邻，满足$\mathcal{L}_w(v)=\mathcal{P}(v)$。*

算法准确地计算这个定向，并且惟一的计算证据节点标号，在节点标号中，领导人被标

以（0，…，0）。算法利用三种类型的消息。领导人在消息<**dmn**，i>中，向它的每个近邻发送连接链路的标号，非-领导人在消息<**iam**，nla>中，向其他处理器发送它们的节点标号。非-领导人在消息<**labl**，i>中，通知它们的近邻连接链路的标号。

```
begin num_rec_p := 0 ; dis_p := 0 ; lbl_p := (0, ···, 0) ;
      for l = 0 to n − 1 do (* Send for phase 1 *)
          begin send ⟨dmn, l⟩ via link l ;
                π_p[l] := l
          end ;
      while num_rec_p < n do (* Receive for phase 2 *)
          begin receive ⟨labl, l⟩ ;      (* necessarily via link l *)
                num_rec_p := num_rec_p + 1
          end
end
```

图11-8　超立方体上的定向（初始处理器）算法

```
begin num_rec_p := 0 ; dis_p := n + 1 ; lbl_p := (0, ··· ,0) ;
      forall l do nei_p[l] := nil ;
      while num_rec_p < dis_p do (* Receive for phase 1 *)
        begin receive msg via link l ; num_rec_p := num_rec_p + 1 ;
              (* msg is a ⟨dmn, i⟩ or ⟨iam, nla⟩ message *)
              if msg is ⟨dmn, i⟩ then
                begin dis_p := 1 ;
                      nei_p[l] := (0, ··· ,0) ; lbl_p[i] := 1
                      (* So now lbl_p = (0, ··· ,1, ··· ,0), with one 1 *)
                end
              else
                begin dis_p := 1 + # of 1's in nla ;
                      lbl_p := (lbl_p or nla) ;
                      nei_p[l] := nla
                end
        end ;
      (* Send for phase 1 *)
      forall l with nei_p[l] = nil do
          send ⟨iam, lbl_p⟩ via link l ;
      while num_rec_p < n do (* Receive for phase 2 *)
        begin receive ⟨labl, i⟩ via link l ;
              num_rec_p := num_rec_p + 1 ; π_p[l] := i
        end ;
      (* Send for phase 2 *)
      forall l with nei_p[l] ≠ nil do
        begin π_p[l] := bit in which lbl_p and nei_p[l] differ ;
              send ⟨labl, π_p[l]⟩ via link l
        end
end
```

图11-9　超立方体上的定向（非-初始处理器）算法

如图11-8所示的算法，给出了领导人的算法，如图11-9所示的算法，出了非-领导人的算法。它由两步骤组成。在步骤1中，消息流远离领导人，在步骤2中，消息流朝向领导人。节点p的前驱（predecessor）是p的近邻q，其中$d(q, w) < d(p, w)$。节点p的后继（successor）

是p的近邻q，其中$d(q, w) > d(p, w)$。在超立方体上，对于$d(q, w) = d(p, w)$，q不是节点p的近邻。距离领导人为d的节点有d个前驱和$n-d$个后继。

领导人首先通过在标号为i的链路上发送消息<**dmn**，i>来初始化算法。当非-领导人处理器p得知它距领导人的距离dis_p，并且已经收到来自它前驱的消息时，p就能够计算它的节点标号nla_p。处理器p通过消息<**iam**，nla_p>向它的后继转发这个标号。为了表明p确实能够这样做，首先考虑p通过链路l接收到消息<**dmn**，i>的情形。当领导人发送消息时，$dis_p = 1$，所有其他的近邻都是后继。p的节点标号在第i位为1，在其他位为0。（在这种情况下，链路l的标号变为i。）然后，p经过所有$k \neq l$的链路转发消息<**iam**，nla_p>。

接下来，考虑p接收消息<**iam**，*label*>的情形。这条消息的发送者和领导人间的距离d是由此消息导出的（*label*中的1的数目）。消息<**iam**，*label*>仅被发送到后继中，因此，发送者是p的前驱，且$dis_p = d + 1$。这也揭示出前驱的数目，p一直等到接收到dis_p条消息<**iam**，*label*>。然后，p计算它的节点标号作为所接到的节点标号的逻辑析取，并把它转发到没有接到消息<**iam**，*label*>的那些近邻中，因为它们是后继。

379 ~ 381

在步骤1中，每个非-领导人处理器p计算它的节点标号。在步骤2中，每个非-领导人处理器p从它的后继得知指向后继的链路的定向，并计算指向前驱的链路的定向。在消息<**labl**，i>中经过这条链路发送这个信息。只要从所有后继接到这些消息，处理器就向其前驱发送消息<**labl**，i>，然后终止。当从所有近邻接到消息<**labl**，i>时，领导人终止。

处理器p的变量为：num_rec_p，表示已经接到的消息数；dis_p，到领导人的距离（当第一条消息到达时计算，初始化为$n+1$）；lbl_p，p所计算的节点标号；$nei_p[0..n-1]$，存储p的前驱节点标号的数组；$\pi_p[0..n-1]$，用于存储方向侦听。

引理11.21 *算法在每个处理器中终止。*

*证明。*对d进行归纳，容易证明，距离至多为d的所有处理器最终在步骤1中发送消息。对于$d = 0$，只有领导人自身距离领导人为d，它才可能在没有首先接收到其他消息时，就发送消息。假设在步骤1中，距离领导人为d的所有处理器发送所有消息，并考虑距离领导人为$d+1$的处理器p。当p的所有前驱最终向p发送步骤1中的消息后，p接到这些消息中的一条，并设置$dis_p := d+1$。当p接收到它的所有$d+1$个前驱的消息时，p自己发送步骤1中的消息（向它的后继）。

类似地，可以证明，所有处理器发送步骤2中的消息，并且终止。 □

引理11.22 *终止后，π是一个定向。对于经过链路l连接的近邻p和q，lbl_p和lbl_q仅在第$\pi_p[l]$（等于$\pi_q[l]$）位不同。*

*证明。*按照定理11.20，只存在一个定向$\mathcal{L}$和一个证据节点标号N，满足$\mathcal{L}_w(p) = \mathcal{P}_w(p)$，且$\mathcal{N}(w) = (0, \cdots, 0)$。

在步骤1中，处理器计算节点标号N。正如对与领导人的距离进行归纳所证明的那样。节点w设置lbl_w为$(0, \cdots, 0)$，它就是$\mathcal{N}(w)$。如果从w到p的链路在w被标号为i，则w的近邻p用b_i为1设置lbl_p，否则用0设置。因此，$lbl_p = \mathcal{N}(p)$。

现在假设距w为d的所有节点q计算$lbl_q = \mathcal{N}(q)$，并考虑距w为$d+1$的节点p。$\mathcal{N}(p)$是一个串，由$d+1$个1和$n-d-1$个0组成。节点p有$d+1$个前驱，对于前驱q而言，只需把$\mathcal{N}(p)$中的一个1变成0就找到了$\mathcal{N}(q)$。因此，把$d+1$个标号$\mathcal{N}(q)$的析取就是$\mathcal{N}(p)$。

步骤1之后，对于经过链路l连接的p的前驱q，$nei_p[l] = lbl_q$。在步骤2中，当lbl_p和$nei_p[l]$在某一位不同时，p才计算$\pi_p[l]$。因此lbl_p和lbl_q只在位$\pi_p[l]$不同。当q接到p的消息<**labl**，$\pi_p[l]$>后，

q在这条链路上也使用同样标号。 □

总之，图11-8和图11-9所示的算法定向于具有领导人的超立方体，需要交换$2 \cdot |E|$条消息。消息复杂度渐进地最优，参见推论11.27。

11.3.4 使用掩码的有效超立方体算法

图11-8和图11-9所示的算法可被用于在未标号的超立方体上执行其他的任务[DRT98]。人们甚至可以通过从发送消息中排除一些处理器，而使用更少的消息。这可以通过定义一个*掩码*（mask）来实现，掩码是长度为n的位串的子集。一旦计算出它的节点标号，处理器就检查
382 是否这个标号在掩码中（掩码被预先定义，且为全部处理器所知）。*当且仅当*标号在掩码中，处理器将发送消息，就像在图11-8和图11-9所示的算法中所规定的那样。

1. 掩码算法的解释

M是$\mathbb{Z}^n_2$的子集，即长度为n的位串的集合。用M_k表示只包含k个1的位串的集合M。可以理解，掩码必须满足一些条件，并且要对算法作出一些修改，我们现在对此作出解释。首先，位串00...00总是在M中，初始处理器的近邻一旦接到消息<**dmn**，i>，就计算它们的标号。而掩码中的其他节点必须接收到至少两个前驱的消息才能开始计算它们标号。因此，对于串$v \in M_k$（其中$k \geqslant 2$），一定至少有两个前驱x和y在M_{k-1}中。

现在解释技术上的细节。由于x和y是v的前驱，v是x和y的共同后继，但是x和y也有共同前驱，记为w。确实有两个接收到x和y的消息的处理器，它们是处理器v和w。我们要求，通过假设w也在掩码中，*x和y发送之前*，x和y的共同前驱已经计算了它的标号。

定义11.23 *掩码是子集$M \subseteq \mathbb{Z}^n_2$，满足*

（1）$0^n \in M$。

（2）*对于$v \in M_k$，其中$k \geqslant 2$，存在x，$y \in M_{k-1}$是v的前驱。*

（3）*如果v和它的两个前驱x和y都在M中，那么，x和y的共同前驱在M中。*

给定（共同已知的）掩码M，算法操作如下。

（1）初始处理器在链路i上发送消息<**dmn**，i>，其中i的二进制表示为00...1...00 $\in M$（其中1在第i个位置）

（2）处理器一旦收到这个消息，就计算它的标号lbl_p = 00...1...00，并且（因为它在M中）通过所有其他边发送消息<**iam**，lbl_p>。

（3）处理器一旦收到两条消息<**iam**，l>，且这两条消息的串中包含相同个数的1，一个节点就可计算标号v，它是这两个标号的共同后继。

如果这个标号不在M中：忽略所有进一步的消息。

如果这个标号在M中：等待接收来自位于M中的v的所有前驱的消息<**iam**，v>，然后通过
383 所有链路发送消息<**iam**，v>。

观察可见，该算法计算的信息较之图11-8和图11-9所示的算法要少。对于那些节点标号在M中的处理器，才计算它的节点标号，而只有两端都在M中的链路，才计算它的链路标号。但是它也使用较少的消息：其消息数可用$n \cdot |M|$限定。因为只有标号在M中的节点才发送消息，每次至多发送n条消息。

2. 发送和广播的小掩码

我们现在能观察到类似于11.2.3小节的一个现象。对算法的微调并不集中在算法方面的问

题，而是一个关于掩码性质的组合练习。

我们考虑的第一个任务是*FarSend*，其中初始处理器向距离为d的某个节点发送消息，假设它的标号是1^d0^{n-d}。这个地址可以解释为初始处理器的“私有”方向侦听，即初始处理器的局部链路标号，它当然无需与其他节点处的定向一致。回忆开始时，除了初始处理器，其他节点都没有任何它的节点和链路标号的知识!

引理11.24 *存在包含标号1^d0^{n-d}，且大小为$1+\frac{d(d+1)}{2}$的掩码。*

*证明。*集合$M=\{0^i1^k0^{n-i-k}:\ i+k<d\}$满足所要求的性质。 □

因此，在无标号的超立方体上，*FarSend*仅花费$O(d^2n)$条消息。文献[DRT98]中证明，这个掩码的大小是最优的。我们用F_v表示，*FarSending*将一条消息发送给标号为v的节点的掩码。

可用覆盖整个网络的掩码解决广播任务。即每个节点都包含在掩码中，或者与掩码中的一个节点近邻。然而，为了得到线性的复杂度，掩码应该至多包含$O(N/n)$个节点，还不知道是否存在同样大小的覆盖掩码。因而，广播算法利用*FarSend*把消息传输到所选节点的一个更稀疏的集合中，然后从这个集合，经过有限次的跳跃将其进行扩散。因为*FarSend*每次将$O(n^3)$条消息传输到一个节点上，因此算法的主节点数为N/n^3。

命题11.25 *存在集合$C\subseteq\mathbb{Z}_2^n$，满足*

(1) $|C|=O\left(\frac{N}{n^3}\right)$

(2) *对于每个$x\in\mathbb{Z}_2^n$，$d(x,C)<3$。*

广播算法使用命题中确定的集合，它的成员称为主节点（chief）。步骤1中，每个主节点 384
通过*FarSend*接到初始处理器的消息。步骤2中，每个主节点以3为距离扩散消息，即把消息发送到每个近邻中，近邻再把消息转发到它们的近邻中，后者再转发一步到它们的近邻中。

现在观察，因为C的第二个性质，每个节点在步骤2中接到消息。下面，考虑复杂度。因为每个主节点以$O(n^3)$条消息可达，对C的大小的限定蕴含着，步骤1的消息复杂度是线性的。因为主节点有n个近邻，且有（少于）n^2个近邻的近邻，在步骤2中这些节点中的每个节点发送n条消息，因此步骤2的消息复杂度也是线性的。

3. 另一种应用：远程定向（FarOrient）

掩码算法可以用于一个节点相对于另一个节点的定向。假设节点v想要对距离为d的某个节点定向，即这个节点必须对所有它的边重新标号，且要与在v处的边标号一致。节点u从包含v的所有近邻的掩码M开始运行掩码算法。由于v将通过每一条链路接收消息**<iam**，l>，它将能够计算每条链路的方向。可以证明[DRT98]，存在大小约为$n\cdot d$的掩码，并且这个掩码是最优的，*FarOrient*的消息复杂度为$O(n^3)$。

11.3.5 无标号超立方体上的选举算法

最有效的超立方体上的选举算法基于归并树原理。如在Galleger-Humblet-Spira算法（7.3.2小节）中所用的那样。超立方体的拓扑性质可用于降低复杂度。本节试图描述最重要的技术，但这里没有给出算法的完整描述，参见文献[Dob99]。

GHS算法的消息复杂度为$\Theta(N\log N+m)$。这是最优的算法（无拓扑结构知识）。这两项代表不同含义。算法的一个组成部分表示搜索离开当前子树的一条边：GHS算法测试每条边，

寻找它与另一棵树可能的归并。只有线性次数的这种尝试取得了成功，在大多数情况下，找到这条边，就可返回到同一棵子树的一个成员。而那些不成功的搜索，其复杂度为m，代表消息复杂度中的第二项。算法的另一部分是对两棵大小分别为n_1和n_2的子树进行归并，得到大小为n_1+n_2的一棵子树。GHS算法对较小的一棵子树重新标号，因此以$\min(n_1, n_2)$条消息实现了归并。这使得归并的总代价为$\Theta(N \log N)$。

385 为使选举的代价降低，我们需要对未能成功尝试的边的数目进行限制，并且必须减少归并子树的代价。在Dobrev算法的步骤1中，节点开始按照广度优先搜索（breadth-first search）方法，构造一棵深度为常数（5）的子树。超立方体的拓扑结构蕴含着对不成功搜索尝试的次数的限定。离根的距离为5的节点在树的第4层只有5个近邻，因此成功搜索只用5次比较，这是一个常数。因此，用可接受的较低的代价可以构造深度为5的BFS树。

如果一棵树包含离根的距离为5，而距某一中心节点的距离少于5的全部节点，就要使用另一种扩展技术。因为至多还有$O(N/n^5)$棵树，每次归并的n^5条消息的代价是可接受的。在一棵树的中心，可用相对位置的列表表示这棵树。但在其他地方用“专有”方向侦听表示。即如果把中心的局部边标号扩展到整个图上，就会产生方向侦听。

有了这种格式的整棵树之后，中心节点可以确定出对树进行扩展的位置：一个离这棵树距离为5的点。如果不存在这样的点，中心就变成领导人；条件蕴含着，在超立方体上，不存在属于另一棵子树的半径为5的球。用*FarOrient*方法可以靠近这个所选择的点，从该点扩展这棵树，或者用BFS扩展5层，或者通过与另一棵子树的归并扩展。

为了归并两棵树，一个中心的位置列表被转化为另一个中心的方向侦听。这可以通过*FarOrient*完成，由它来确定两个节点定向的相互一致性。所得算法完全不简单，对于它的分析是当前研究的课题。

11.4 与复杂度有关的问题

在前面的几节中，主要研究了方向侦听和拓扑结构知识的主题。我们现在继续研究另外一些没有多少相关性的问题。它们中的一些是当前研究的中心。

11.4.1 团或任意图的定向

了解到方向侦听可以减少某些任务的复杂度，我们的问题是，是否在没有给定方向侦听的网络中，可以计算出方向侦听。为了与有方向图中边操作不相混淆，我们称计算方向侦听
386 的问题为网络定向。本节中，我们证明了对图定向的消息复杂度的下界，提出了网络定向的几种算法。

1. 定向算法的下界

因为要对网络定向，需要在每一条边上设置标号，如同第一个定理证明中的那样，需要利用每一条边。

定理11.26 *每次执行中，任何定向算法至少交换$m-1/2N$条消息。*

*证明。*对于链路标号$\mathcal{P}$，设$\mathcal{P}^{u,v,w}$（v，w是u的近邻）是由$\mathcal{P}_u^{u,v,w}(v)=\mathcal{P}_u(w)$，$\mathcal{P}_u^{u,v,w}(w)=\mathcal{P}_u(v)$，以及$\mathcal{P}^{u,v,w}$在$\mathcal{P}$中的所有其他标号所定义的标号。（交换$\mathcal{P}_u(w)$和$\mathcal{P}_u(v)$可得$\mathcal{P}^{u,v,w}$）如果$\mathcal{L}$是定向，则$\mathcal{L}_u[v]=-\mathcal{L}_v[u]$，但$\mathcal{L}_u^{u,v,w}[v]$不同，因此，$\mathcal{L}^{u,v,w}$不是定向。

考虑定向算法的一次执行，初始标号为$\mathcal{P}$。算法终止时，每个节点上有一个排列π_v（$\mathcal{L}=\pi$

($\mathcal{P}$)是一个定向)。此外假设在这次执行中，某些节点u并没有向（从）它的两个近邻v和w发送（接收）消息。由于u没有与v，或者w通信，如果网络初始时标号为$\mathcal{P}^{u,v,w}$，且所有处理器终止于同一个排列，那么执行有可能相同。然而，$\mathcal{L}' = \pi(\mathcal{P}^{u,v,w}) = \mathcal{L}^{u,v,w}$不是一个定向，算法是不正确的。

由此可得，在每次执行中，每个节点必须至少与除了一个以外，所有它的近邻进行通信。 □

推论11.27 *N个顶点团的定向需要交换$\Omega(N^2)$条消息，n-维超立方体的定向需要交换$\Omega(n2^n)$条消息，$n \times n$圆环的定向需要交换$\Omega(n^2)$条消息。*

2. 团的定向

团的定向算法（图11-10所示的算法）赋给每个处理器惟一的编号，号码范围为0，…，$N-1$。在接到每个近邻的名字之后，每个处理器计算它在由所有名字组成的集合中的序号。每个近邻的序号可以被类似地计算，近邻序号和它自己的序号的差，为链路标号。

算法依赖节点中不同标识的可用性，但是很容易修改，而变成使用领导人的算法。在这种情况下，领导人开始向它的近邻发送1至$N-1$的数，然后这些数被用作标识。与寻找链路标号相比，分发这些不同的数的代价较小（N条消息），因为按照定理11.26，寻找链路标号的代价为消息数的平方。 387

```
begin for l = 1 to N − 1 do send ⟨name, p⟩ via l ;
      rec_p := 0 ;
      while rec_p < N − 1 do
         begin receive ⟨name, n⟩ via link l ;
               rec_p := rec_p + 1 ; nei_p[l] := n
         end ;
      (* Compute node label *)
      lbl_p := #{k : nei_p[k] < p} ;
      for l = 1 to N − 1 do
         begin (* Compute neighbor's node label and link label *)
               ll := #{k : nei_p[k] < nei_p[l]} ;
               if p < nei_p[l] then ll := ll + 1 ;
               π_p[l] := (ll − lbl_p) mod N
         end
end
```

图11-10 团的定向算法

3. 任意网上弦方向侦听

在任意网络上，假如用一个节点作为起始节点，可以用一个遍历网络的令牌，如，深度优先搜索遍历，对节点进行编号。遍历用$2m$条消息（每条消息在每个方向上通过每条边），节点按照被访问的次序编号。在遍历过程中，处理器得知自己的编号，也得知近邻的编号，然后就能够计算弦方向侦听。（必须已知节点数。如果预先不知道节点数，就要在遍历的过程中计数，并从初始处理器利用$N-1$条消息将这条消息广播出去。）

11.4.2 位复杂度和多路径流量算法

我们现在通过考虑超立方体上定向算法的位复杂度，来更详细地评估复杂度。定理11.26已经蕴含着消息数的界限。但是对其扩展，就能对算法的位复杂度进行界定。

388 **引理11.28** *在n-维超立方体的定向算法中，每个节点通信的位复杂度至少为*$\Omega(n \log n)$。

证明。已经解释，节点至少通信$n-1$条消息才能区分它的n条链路。为了真正区分这些链路，这些消息必须不同，这蕴含着它们至少包括$\Omega(\log n)$位。 □

多路径流量算法（11.3.3小节）中使用的消息包含节点的标号，其占用n位。现在证明，只用$O(\log n)$位的消息就可以实现算法。

算法不需要包含在<**iam**，*label*>中的所有信息。有许多冗余。只要传输1的个数、串中为1的最小下标，以及对应1的下标和（模n）。对于一个节点，标号*label*定义1的个数、最小下标以及下标和为$wgt(label) = \#\{i : b_i = 1\}$；$low(label) = \min\{i : b_i = 1\}$；$ixs(label) = (\sum_{b_i=1} i) \bmod n$。定义综合（*summary*）为三元组$smy(label) = (wgt(label), low(label), ixs(label))$。节点的综合就是其节点标号的综合。

引理11.29 *设p是距离w为*$d+1 \geqslant 2$*的节点。*

（1）*由p的一个前驱的综合可导出*$dis_p = d+1$。

（2）*从p的*$d+1$*个前驱的综合，可计算出p的综合。*

（3）*从p的综合以及p的*$d+1$*个前驱的综合，可计算出p的节点标号。*

（4）*由p的节点标号和q的综合，可计算出p的前驱q的节点标号。*

证明。（1）由$wgt(\mathcal{N}(q))$与$d(q, w)$相等，dis_p的计算平凡。

（2）现在设给定p的$d+1$个前驱的综合。那么，$d+1$个综合的d，其*low*值等于$low(\mathcal{N}(p))$，而一个综合有更高的*low*值（通过将$\mathcal{N}(p)$中第一个1变为0而得其标号的前驱）。这给出$low(\mathcal{N}(p))$值，同时也确定节点标号的下标和ixs_0，该节点标号与$\mathcal{N}(p)$在位置*low*不同。因此，$ixs(\mathcal{N}(p)) = (ixs_0 + low(\mathcal{N}(p))) \bmod n$。这就完成了$smy(\mathcal{N}(p))$的计算。

（3）对q进行$d+1$次选择作为p的前驱，计算$ixs(\mathcal{N}(p)) - ixs(\mathcal{N}(q)) \bmod n$，得到$\mathcal{N}(p)$中1的$d+1$个位置。

（4）对于p的前驱q，在$\mathcal{N}(p)$中，把下标$(ixs(\mathcal{N}(p)) - ixs(\mathcal{N}(q))) \bmod n$对应的位从1变到0就得到$\mathcal{N}(q)$。 □

389 由引理11.29可得，在定向算法中，只发送节点标号的综合，无需发送整个标号，因此，用$O(\log n)$位就可实现算法。

11.4.3 Verweij随机漫步算法

我们现在给出Verweij[VT95]提出的随机Monte Carlo选举算法。它适合于任意的拓扑结构，也没有利用方向侦听，因为它是基于竞争的随机漫步的设计理念。

初始时，每个处理器处于睡眠状态，一旦处理器苏醒（自然地出现这种状态，或者在睡眠状态中接到消息），它就产生包含自己标识和跳数计数器的令牌。在网络中按照随机漫步算法转发令牌，并且该令牌可被具有更大标识的令牌所访问过的处理器从网络中删除。当令牌完成K步，还没有被删除时，持有该令牌的处理器被选中，参见图11-11所示的。

1. 算法如何工作

被初始化的最大令牌不会被杀死，因此，这个令牌会完成K步，并使一个进程变成领导人，这被称为正确的领导人，这个性质蕴含着，算法仅在选出多个领导人的失败。而不是在根本选不出领导人时，失败。

如果任何一个非最大值的令牌成功地完成了K步，而没有进入已被更大令牌访问过的处

理器，算法就会失败（即选出多于一个领导人）。所选的领导人为假领导人。可以通过增加K值，来减少出现这种假声明的可能性，但由于令牌要进行K步之多，增加K值也增加了消息的复杂度。

```
cons K                                          (* Safety parameter *)

var   state : (sleep, awake, leader) init sleep ;
      maxid_p ;                                 (* Highest identity seen *)

To start an election:
      begin i := p ; s := 0 ; maxid_p := i ;
            if s < K
              then send ⟨walk, i, s + 1⟩ through a random edge
              else state_p := leader
      end

Upon arrival of token ⟨walk, i, s⟩:
      begin if state_p = sleep then start an election ;
            receive ⟨walk, i, s⟩ ; maxid_p := max(maxid_p, i) ;
            if i = maxid_p
              then if s < K
                     then send ⟨walk, i, s + 1⟩ through a random edge
                     else state_p := leader
              else (* Kill token *) skip
      end
```

图11-11 选举算法（对于处理器p）

对于$K = 0$，每个令牌产生假的声明结果。因此，错误的概率是非常大的（1），但是消息的复杂度低（0）。另一方面，如果K趋于∞，错误的概率就会收敛到0。因为不存在最大令牌可以进行无限多步，仍然以正概率避免最大处理器。

幸运的是，对于算法的正确性或者算法的复杂性质，K的选择并不是非常关键的。首先，即使K值很小，消息的复杂度至少是关于N的线性关系，因为每个处理器可能产生一个令牌。其次，选择比获得期望的可靠性所必须的更大的K值，会产生额外的令牌步，但是（以更大概率）仅对最大令牌适合。讨论可得，只要K值超出最小值一个线性量，渐进的期望的消息数就不会受到K的影响。

目前对于复杂度的数学分析和所需的K值还一无所知。这是一个尚待解决的问题。随后，我们会描述它为什么是一个困难的问题，并结合定时假设，给出一个完整、有竞争力的分析。

2. 定时假设

K的最优值和算法性能受到令牌相对速度的影响。下面的讨论针对网络是团的情况。即令牌的每个连续步可以导致一个完全随机的节点。

一种极端的情形是，与次最大令牌相比，最大令牌无限的慢。这里，仅当次最大令牌进入最大处理器时才被杀死，例如，在团上，K必须为线性值，才能以接近于1的概率保证做到这一点。

考虑消息的复杂度，最坏的定时出现在，仅当所有具有较小标识的令牌都已被杀死时，才接到令牌的时候。然后，仅当被一个具有更大标识的处理器接收时，令牌才被杀死，对于第（i+1）个最大的标识，仅当期望的$\Omega(N/i)$步后，这种情况才会发生。讨论蕴含着，在一

390 ~ 391

个完全异步的团上，消息的复杂度为$\Omega(N \log N)$。

另一方面，考虑同步团上的例子，在任何时刻，每个幸存的令牌前进步数相等。在$O(\sqrt{N})$步后，最大的令牌已经访问了$O(\sqrt{N})$处理器。在经过另一个$O(\sqrt{N})$步后，第二个令牌以接近于1的概率进入这些处理器之一。因此，$K = O(\sqrt{N})$足以以大概率杀死第二个最大的令牌。由较小令牌做出假领导人的声明的概率是很小的。

3. 复杂度的模拟研究

文献[VT95]在团和超立方体上对算法进行了模拟，来实验性地确定它的复杂度。消息复杂度约为$3N$。可见这种度量远远好于完全异步情形下所能证明的复杂度。这蕴含着，为了导出复杂度的解析式，必须考虑定时假设（参见Archimedean假设）。

11.5 结论和未解决的问题

我们已经看到，方向侦听可以有效地利用网络的拓扑结构，使得处理器之间的通信更直接。研究方向侦听的性质是每年一度的SIROCCO（Structural InfoRmatiOn and Communication COmplexity）会议的主题。SIROCCO会议文集由Carleton大学出版社出版。

11.5.1 利用方向侦听

在11.1.2小节中，指出方向侦听具有诸多方面的用途。路径比较的能力使它有可能回避广播和选举算法的下界$\Omega(m)$，因为不再需要通过边发送消息来验证，按照算法是否已经到达近邻。路径比较展现了任意网上线性广播和$O(N \log N)$选举的可能性。

路径压缩使通过最短路经到达已知位置的节点成为可能。11.2节表明了它在弦环上的选举
392 算法中的用图。通过为弦选择各种设置，可以精细化调整路径压缩的数量。可以证明，即使只利用适量的弦数，路径压缩仍然可以导致线性选举算法。路径压缩的使用要求方向侦听是一致的。

11.2节把路径压缩比作方向侦听的第三种能力：充分利用蕴含在一致方向侦听中的拓扑知识。已经证明，如果在算法中利用这种知识，只需更少的弦数就能够获得线性的选举算法。

11.5.2 复杂度归约

图11-12概述了本章中所报告的许多成果。给出了广播和选举算法在各种网络结构上的消息复杂度，考虑了三种假设：第一种假设没有拓扑结构知识和方向侦听因为算法不“知道”它处在哪种拓扑结构中，拓扑结构知识和方向侦听均要求算法通过每条边发送消息；第二种假设只有拓扑结构知识可用，这种情况使得算法可以利用网络图的性质，但边的方向未知；最后一种假设给定拓扑结构知识和方向侦听。

任务	网络	无SoD 无TA	无SoD TA	SoD 且 TA	改进 由于
BC	任意网	m	(1)	N	Yes:SD
	弦环（2）	$c \cdot N$	未知	N	Yes（3）
	圆环	N	N	N	No
	超立方体	$N \cdot n$	N	N	Yes:TA
	团	N^2	N	N	Yes:TA

图11-12　结构化知识和复杂度

任务	网络	无SoD 无TA	无SoD TA	SoD 且 TA	改进 由于
EI	任意网	$N \log N + m$	(1)	$N \log N$	Yes:SD
	弦环（2）	$N \log N$	N	N	Yes:TA
	圆环	$N \log N$	N	N	Yes:TA
	超立方体	$N \log N$	??	N	Yes:TA or SD
	团	N^2	$N \log N$	N	Yes:TA+SD

图11-12 （续）

注释：（1）拓扑结构知识与这类网络无关。

（2）c是弦数，假设$c \leqslant \log N$。

（3）是否只用TA就能达到改进，未知。

最右列报告了是否结构化信息有助于降低任务的复杂度。除了在圆环上的广播之外（即 393
使不知道网络是圆环，线性消息复杂度也可以广播。）结构信息可以降低消息复杂度。然而，在大多数情况下，这种复杂度的降低是由于利用了拓扑知识。不包括任意网络上的消息复杂度的降低，团上的选举一直是惟一的任务，对于这个任务，有向边降低了消息复杂度。

如果按照渐进消息复杂度这不是优点的话，方向侦听还有其他一些好处：算法简单、隐含的常量更小，以及更好的时间复杂度。

11.5.3 当前研究

当前的研究方向包含如下几个方面：

（1）针对一般路由策略，证明引理11.11和引理11.14。

（2）弦长t取何值时，1-弦算法可被修改，使它仍然具有线性的消息复杂度？

（3）计算随机漫步选举算法的期望复杂度（对于团、超立方体和圆环）。

（4）基于不可交换群，概述本章的方向侦听的结果。这样的概述将会是有趣的。例如，利用排列群定义的立方连接的环网。

习题

11.1节

11.1 设$\mathcal{L}$是用群G中的元素对边的标号。用$SUM(P)$表示路径P上找到的标号的和。证明以下两种说法是等价的。

（1）$\mathcal{L}$是方向侦听。

（2）路径P是闭的，当且仅当$SUM(P) = 0$。

11.2节 394

11.2 （项目）对于任意路由策略，证明$\Omega(\text{loglog}N)$条弦，是使引理11.9中的求和为几何级数的必要条件。

11.3节

11.3 进行比赛的算法的时间复杂度是多少（11.3.2小节）？ 395

第12章 网络中的同步

本章研究分布式计算的理论是如何受到假设影响的。假设存在进程可以访问的全局时间。全局时间帧是我们周围物理现实的一部分，这个现实包括分布式系统的进程，而且分布式算法的设计会得益于对时间的充分利用。本章结果表明，如果处理时间和通信时间有界，就可以充分利用时间来减少分布式算法的通信复杂度。

然而，不利用时间就不可解的问题，在引入时间后，仍然是不可解的。因此，引入同步机制改进了复杂度，但是不能提高分布式系统的计算能力。这种情况与遭受故障的网络中的同步机制影响不同。在第14章和第15章中，可以看到，如果充分利用同步机制，就可以确定处理许多类故障。在3.2节中已经利用同步机制来克服转移故障。

本章主要研究理想的情况下的全局时间问题，即，整个网络按照离散步骤操作。每一步骤称为*脉冲*（pulse），在12.1.1小节中会进行解释。在每个脉冲中，每个进程进行局部计算，并保证在一个脉冲中发送的消息在下一个脉冲之前被接收到。这个全局同步操作也称为*锁步*（lockstep）操作。如果时钟可用，并且假设转移延迟有上界，锁步操作的实现相当容易。参见12.1.3小节。

锁步中的网络操作通常被当作*同步网络*（synchronous network）。同步网络一定不能与2.1.3节所定义的同步传递消息的网络相混淆。在后者中没有全局同步机制。仅当两个进程通 396
信时，它们才在消息交换中暂时同步，但并不与所有进程同步，消息交换之后，同步很快就被打破。

全局同步的网络在解决分布式问题时，比完全异步的网络通信量小。可以用同步算法（对于选举问题）证明这一点，对于同一问题，同步算法具有比异步算法更低的复杂度。参见12.2节。我们用所谓的*同步器*（synchronizer）算法证明，同步算法不能比异步算法解决更多的问题。利用同步器算法，异步网络可以模拟同步网络。参见12.3节。

在同步网络上，进行分布式算法的设计常常要容易些。因为不需考虑太多的非-确定性。对于异步算法的设计，我们可以首先给出它的同步算法，然后将它与12.3节的模拟技术结合起来，产生异步算法。尽管引入同步器会带来一些开销，所得算法仍然优于直接为异步网络所设计的算法。作为同步器的应用例子，广度优先搜索树的构造将在12.4节进行考虑。

即使时钟不可用，做一些弱的定时假设也是合理的。例如，假设系统中最快的和最慢的组件的执行速度之间的比率是有界的。这个假设被称为Archimedean假设[Vit85]。在12.5节作了简要讨论。已经证明，由于几种原因，一般来说，完全依赖异步算法而不利用定时假设会更好。在Archimedean框架下，所设计的算法，它们的正确性不依赖于定时假设，而它们的复杂度却得益于这些假设。

12.1 预备知识

12.1.1 同步网络

在同步的分布式系统中，每个进程的操作发生在一个离散步的序列（可能无限）中。这

个序列中的每一步称为脉冲（*pulse*）。在一个脉冲中，进程首先发送（零或者多个）消息，然后接收（零或者多个）消息，最后进行局部计算（改变状态）。消息的接收与发送在同一脉冲中，即如果p在第i个脉冲向q发送一条消息，那么q在它的第i个脉冲接收这个消息，且q的接收
397 在那个脉冲中q的计算开始之前发生。

可以把脉冲数看作全局时钟的滴答（“脉冲”）数。计算在时钟脉冲中进行，在一个脉冲中发送的消息保证在下一个脉冲开始之前接收到。系统在第i个脉冲之后的配置，定义为第i个脉冲之后每个进程状态组成的元组。

为了形式地定义同步系统，如第2章，设$\mathcal{M}$表示消息的集合，$\mathcal{PM}$表示消息多集的集合。在同步系统中，进程的定义与异步系统中进程的定义不同，因为在系统的一次转移中就可进行消息集的接收和发送。为了避免混淆，我们称同步系统的进程为同步进程（*synchronous process*）。

定义12.1　同步进程是4-元组$p=(Z, I, \mathcal{MG}, \vdash)$。其中

（1）Z是状态集合，

（2）I是初始状态集，Z的子集。

（3）$\mathcal{MG}: Z \to \mathcal{PM}$是消息生成函数，且

（4）$\vdash$是$Z\times\mathcal{PM}\times Z$上的关系。我们用$(c, M)\vdash d$表示$(c, M, d)\in\vdash$。

进程在状态$c_0\in I$开始它的计算。当进程在状态$c\in Z$开始一个脉冲时，它就发送消息集$\mathcal{MG}(c)$。在下一个脉冲之前，它接收到在这个脉冲中发送给它的消息集，如，M，并进入状态d，满足$(c, M)\vdash d$。

在这种定义之下，进程发送的消息集完全由脉冲开始时的状态决定。进程的所有内部非-确定性都在关系$\vdash$中。非确定性也可能在$\mathcal{MG}$中（此时，$\mathcal{MG}$是关系而不是函数）。但是非确定性选择在关系$\vdash$中，同样能够被很好地模型化。

同步系统是同步进程的集合$\mathbb{P}$。同步系统的配置是由每个进程的状态组成的元组。初始配置是这样一种配置，其中每个进程都处于初始状态。所有进程的状态改变同步于系统的一次全局转移，其中每个进程状态的改变受到在当前配置中发送给它的消息集的影响。

定义12.2　由（同步）进程组成的同步系统$\mathbb{P}=(p_1, \ldots, p_N)$（其中$p_i$是进程$(Z_i, I_i, \mathcal{MG}_i, \vdash_i)$）是一个转移系统$S=(C, \to, \mathcal{I})$，其中

398 （1）$C=\{(c_1, c_2, \ldots, c_N): \forall i, c_i\in Z_i\}$，

（2）如果对于每个$p_i\in\mathbb{P}$；$(c_i, \{m\in(\bigcup_{p_j\in\mathbb{P}}\mathcal{MG}_j(c_j)): m$的目的节点为$p_i\})\vdash_i d_i$；则$\to$是由$(c_1, \ldots, c_N)\to(d_1, \ldots, d_N)$所定义的关系，

（3）$\mathcal{I}=\{(c_1, c_2, \ldots, c_N): \forall i, c_i\in I_i\}$。

系统的同步计算是配置的最大序列$\gamma_0, \gamma_1, \cdots$，满足$\gamma_0$是初始配置，对于所有$i\geqslant 0$，$\gamma_i\to\gamma_{i+1}$。这个计算的消息复杂度就是所交换的消息数，时间复杂度等于最后配置的下标（假设计算是有限的）。

12.1.2　通过同步提高效率

在需要的地方，我们将在本章的算法中隐含地假设，进程状态中包含一个脉冲计数器，用以标明进程执行的脉冲数。本小节中，我们将讨论许多范型，这些范型可用于同步网络，目的是降低分布式算法的通信复杂度。

1. 按照时间编码

只用两位来发送任意消息，可以节省大量位交换数。为此，必须对消息的内容按照两位发送之间的时间（脉冲数）“编码”。

假设进程p必须向进程q发送消息。假设消息的内容可表示为整数m。为了传输值m，需要利用两条消息<start>和<stop>。进程p在脉冲i中发送<start>消息，该脉冲也就是传输开始时的脉冲，在脉冲$i+m$中发送<stop>消息。进程q在脉冲a和b分别接收消息<start>和<stop>，并在脉冲b接收消息$b-a$。按照时间编码方法致使消息传输很耗时。q接到消息m的脉冲要比进程p初始化传输的脉冲晚。可以发送更多的位，来减少消息传输对时间的消耗。参见习题12.1。

2. 隐式消息

在异步系统中，仅当进程接到来自近邻的消息时，才可以获得来自这个近邻的信息。在同步系统中，如果进程在某些脉冲内没有接到从已知近邻而来的消息，那么这个近邻在这个脉冲内就不会发送消息，表明进程已经获得关于其近邻的状态的信息。

例如，考虑计算所有点对之间最短路径问题的Toueg算法（图4-6所示的算法），在计算枢轴元素w的这一枢轴轮中，每个进程必须知道它的哪些近邻是它在树T_w中的子节点，为此，经过每个信道经交换消息<**ys**，w>或者消息<**nys**，w>。在同步网络中，只需发送消息<**ys**，w>。 399
如果不能从一个近邻接收到这样的消息（在发生这些消息交换的脉冲中），那么这个近邻就不是子节点。生成消息<**nys**，w>隐含着减少计算每个进程子节点的复杂度，从$2|E|$降到至多$N-1$条消息。（在异步系统中，同样可以避免发送消息<**nys**，w>，但却以设计更复杂算法和增加空间复杂度为代价。）

3. 选择性延迟

我们已经碰到一些开始几个子计算的分布式算法，其中一个子计算保证产生所需的结果，而所有其他的子计算的结果被忽略。这类算法的例子包括把废止原理（7.3.1小节）应用到集中式的波动算法，所获得的选举算法。

为了减少消息复杂度，可以用不同条件减慢每个特定的子计算。在异步情况下，这是不可避免的（至少在最坏情形下），不成功的子计算运行到完成，使得通信复杂度为子计算数与每个子计算的计算复杂度的乘积。在同步情况下，用成功子计算的完成时间除以本次计算的减慢因子所得的值对不成功子计算所需的步数做了界定。在12.2.2小节，分析了由仔细选择的减慢因子所造成的选择性延迟。

12.1.3 异步有限延迟网络

同步网络所基于的假设是一个很强的假设，即所有进程状态的改变是全局同步的。更适度的假设是每个进程都有一个物理时钟（不必同步），且消息传输时间有上界。这一模型在3.2节中使用过，在Chou等人[CCGZ90]之后，这类网络被称之为异步有限-延迟网络（asynchronous bounded-delay network）或者简称为ABD网络。

定义12.3 异步有限-延迟网络是分布式系统，在这个系统中，以下假设成立。

（1）在一个进程内执行一个事件需要0个时间单位。

（2）每个进程有一个时钟，用于测量物理时间；如果$CLOCK_p(t)$表示p的时钟在t时刻的值，并且在t_1和t_2之间，p并不对它的时钟赋值，那么 400

$$CLOCK_p^{(t_2)} - CLOCK_p^{(t_1)} = t_2 - t_1$$

（3）消息发送和接收之间的物理时间被界定为μ。即，如果消息在时间σ发送，在时间τ接收，那么，

$$\sigma < \tau < \sigma + \mu$$

在ABD网络上实现同步网络并不难，因为所需要的消息和时间开销较小。实现同步网络的机制称为同步器（synchronizer），见12.3节。在设计同步器时遇到的主要困难是要保证进程在改变状态并开始脉冲$i+1$之前，接收到脉冲i的所有消息。

1. ABD同步器

ABD网络同步器的主要原理是进程的时钟尽可能精确同步，如，具有精度δ，每个进程在其时钟$\delta + \mu$个时间单位后才执行下一个脉冲。15.3节讨论了时钟同步算法。我们现在给出Tel等人[TKZ94]提出的一些基本算法。

Tel等人提出的ABD同步器，由时钟-同步（clock-synchronization phase）阶段和模拟（simulation phase）阶段两部分组成，作为图12-1所示的算法中给出。在时钟同步阶段中，每个进程将它的时钟重新设置为0，并向它的每个近邻发送消息<start>。由ABD网络所基于的假设，重新设置时钟和发送消息<start>需要的时间为0。进程执行这些步，或者来初始化同步算法（自然地），或者在接到第一条消息<start>的时候。作为同步阶段的结果，近邻进程的时钟以精度μ同步。

```
var CLOCK_p   : clock ;
    state_p   : Z_p ;
    pulse_p   : integer ;

procedure init:          (* Can be executed spontaneously *)
    begin if not started_p then
            begin CLOCK_p := 0 ;
                  forall q ∈ Neigh_p do send ⟨start⟩ to q
            end
    end

I_p: { A ⟨start⟩ message has arrived at p }
    begin receive ⟨start⟩ ; init end

P_p^(i):{ The start of the ith pulse at p }
    when CLOCK_p = 2iμ do
        begin (* End previous pulse, compute state_p^(i-1) *)
              if i = 1
                then state_p := initial state
                else (* M is the collection of (i-1)-messages received *)
                     state_p := d_p satisfying (state_p, M) ⊢ d_p ;
              pulse_p := i ;
              (* send the messages of pulse i *)
              M' = MG(state_p) ; send all messages of M'
        end

M_p:{ A basic i-message has arrived *)
    begin receive and store the message end
```

图12-1　ABD同步器算法

定理12.4 *近邻p和q执行初始化过程init后，在每个时间t，p和q的时钟满足*

$$|CLOCK_p^{(t)} - CLOCK_q^{(t)}| < \mu$$

证明。设w_p（或w_q）表示p（或q）执行过程*init*的时刻。当p在时间w_p向q发送消息<start>，q在接到消息<start>的最近时刻执行*init*，$w_q < w_p + \mu$。反之，$w_p < w_q + \mu$，由此可得$|w_p - w_q| < \mu$。因为p和q在执行*init*之后，并没有对时钟赋值，对于$t \geqslant \max(w_p, w_q)$，则有

$$\begin{aligned}
&|CLOCK_p^{(t)} - CLOCK_q^{(t)}| \\
&= |[CLOCK_p^{(w_p)} + (t - w_p)] - [CLOCK_q^{(w_p)} + (t - w_q)]| \\
&= |[0 + (t - w_p)] - [0 + (t - w_q)]| \\
&= |w_q - w_p| < \mu
\end{aligned}$$

□

在模拟阶段中，进程执行同步算法指定的脉冲（状态改变）。同步算法在脉冲i中所交换的消息，称为i-消息。利用到目前为止接到的（$i-1$）-消息，改变状态后，当局部时钟读数为$2i\mu$时，开始执行第i个脉冲。近似同步以及消息延迟的上界表明，一个脉冲中的所有消息在进程开始下一个脉冲之前已经到达进程，并且系统的行为就像是一个同步系统。设$state_p^{(i)}$表示第$i+1$个状态改变后进程p的状态，即第（$i+1$）个脉冲开始后的状态，记γ_i表示元组（$state_{p_1}^{(i)}, \ldots, state_{p_N}^{(i)}$）。读者应该注意到，这个元组不必作为ABD网络的一个配置出现。因为它可能就是这样一种情况，即，进程从来不会同时出现在这些状态中。 401 ~ 402

定理12.5 *序列$C = (\gamma_0, \gamma_1, \ldots)$是同步算法的一次计算。*

证明。对i用归纳法证明，序列（$\gamma_0, \gamma_1, \ldots, \gamma_i$）是计算的前缀。对于$i = 0$，观察可见，$\gamma_0$是同步算法的初始配置，因为对于每个$p$，$state_p^{(0)}$是同步算法中$p$的初始状态。

假设（$\gamma_0, \gamma_1, \ldots, \gamma_i$）是同步算法计算的前缀。进程$p$利用到目前为止接收的（$i+1$）-消息的集合$M_p^{(i+1)}$，当它的时钟读数为$2(i+2)\mu$时，进程$p$结束第$i+1$个脉冲，并将它的状态变为$state_p^{(i+1)}$。为了证明由（$state_{p_1}^{(i+1)}, \ldots, state_{p_N}^{(i+1)}$）定义的配置扩展了计算，只要证明，$M_p^{(i+1)}$等于在第$i+1$个脉冲发送给$p$的消息集合就足够了。

假设q在第$i+1$个脉冲发送给p一条消息，即，当q的时钟读数为$2(i+1)\mu$时。设σ是这个消息被发送时的（全局）时间，τ是p接收这个消息的（全局）时间。由于$CLOCK_q^{(\sigma)} = 2(i+1)\mu$和$q$是$p$的近邻，$CLOCK_p^{(\sigma)} < 2(i+1)\mu + \mu$（由定理12.4）。因为$\tau - \sigma < \mu$，$CLOCK_p^{(\tau)} < 2(i+1)\mu + \mu + \mu = 2(i+2)\mu$，即$p$在结束第$i+1$个脉冲之前接收消息。 □

进程也可能在开始自身的脉冲之前，接收脉冲i的一条消息。然后存储此消息，仅在下次脉冲中处理该消息。对于每条所接收的消息，进程可能要决定在哪一个脉冲中处理该消息。如果模2的脉冲数随着每条消息发送，是可能出现这种情况的。参见习题12.3。如果每个进程能把接收每一近邻的消息<start>时的时钟时间存储起来，就不需要附加信息。参见习题12.4或者[TKZ94]。

2. 同步复杂度

同步机制的复杂度可用通信和时间来度量，同时还要考虑同步器的初始化和每个模拟脉冲的开销，参见12.3节。

初始化阶段要求交换$2|E|$条消息，需要$O(D)$时间单位。模拟阶段不需要附加信息（仅仅 403

是模拟的同步算法的消息被发送）。进程每2μ个时间单位完成一次脉冲。脉冲之间的时间称为脉冲时间。

Tel等人[TKZ94]研究了ABD网络的所有同步器的类别，这些网络在初始化阶段利用$2|E|$条消息，在模拟过程中不需要附加消息。他们已经证明，对于任意网、环网和超立方体，脉冲时间2μ是最优的。星型网的脉冲时间为$(3/2)\mu$，团网的脉冲时间为$(2-1/N)\mu$。所有这些脉冲时间都是最优的。

3. 时钟漂移和不可忽略的处理时间

为了在实际情况下利用ABD网络的结果，必须放宽定义12.3中的假设。在实际中，处理器的时钟会有漂移，处理一条消息的时间不为0。

在模拟阶段中，我们可以通过逐步增加脉冲时间来处理时钟漂移问题以补偿近邻进程的时钟读取之间可能不断增加的差别。如果脉冲时间因此变得太大（在模拟多个脉冲之后），在一个预先确定的脉冲数之后，有可能重新执行同步阶段。参见[Tel91b，第2章]。

如果处理一个脉冲的消息以及开始下一个脉冲所需的时间是不可忽略的，但又是由π界定的，那么可以通过将脉冲时间增加π，来轻易地修改同步器。

12.2　同步网络中的选举

本节目标是证明，对于某些特定问题的解决方法，同步网络严格意义上比异步网络要求更小的消息复杂度。在第7章中，证明了异步网络中选举领导人所需交换的消息数的下界。本节给出的同步算法，其消息复杂度比这些下界要低。时间复杂度（即，选举领导人的脉冲的数目）主要取决于网络大小是否已知。

算法假设进程标识是不同的整数，如果这些标识取自任意可数集合，那么利用对这个集合的固定的枚举，就可以把它们转换成整数。同时还假设，算法的所有初始进程在同一脉冲
404 （脉冲0）开始选举。初始进程不必在同一个脉冲中开始的情形，将在12.2.3小节中简要讨论。

12.2.1　网络规模已知

本小节考虑的算法需要关于网络直径上界D的知识。如果已知进程数N，那么值$N-1$总是作为直径的上界。

1. 算法

进程p通过将消息发送到它的每一个近邻中，而将消息扩散，在这之后，除了进程p之外，第一次接到消息的每个进程在下一个脉冲中，将接收到的消息发送到它的每个近邻中。按照这种方法，扩散一条消息需要$2|E|$条消息的代价，且如果进程在脉冲i_0，初始化消息的扩散过程，那么每个进程最迟在脉冲$i_0+(D-1)$接收到这条消息。这蕴含着，如果一个进程在脉冲$i+(D-1)$或更早一些还没有接到扩散的消息，那么在脉冲i或者更早一些的脉冲，没有进程已经初始化了扩散过程。

在选举算法中，只有一条消息被具有最小标识的进程p_0进行扩散，在脉冲p_0D中初始化这一扩散过程。如果进程p_0在脉冲p_0D-1或者之前还没有接到扩散的消息，那么它知道没有进程的标识比它更小。不初始化消息扩散过程，所有进程，蕴含着已经扩散了这样一条消息“我的标识超过p_0-1”。

总之，进程p等待，直至或者一条消息到达，或者在p接到消息之前，脉冲数达到pD。在

后一种情况下，进程p初始化消息扩散过程，并成为领导人。算法首先由Santoro和Rotem [SR85]提出。消息复杂度为$O(|E|)$，时间复杂度为$p_0D+(D-1)$，其中p_0是进程的最小标识。观察可见，时间复杂度并不限定为N的函数。

2. 扩展和变化

可以容易地扩展算法来计算以领导人为根的一棵生成树。每个进程选择在最早脉冲中接到消息的一个进程作为它的父节点。

由于消息中不需要进行信息的交换，这些算法的位复杂度等于它们的消息复杂度。

如果拓扑知识可用，就可以更有效地进行扩散过程。当然，如果拓扑结构任意和未知，那么通过每个信道至少要发送一条消息，类似于定理7.15中的所使用的论证。如果拓扑结构已知为团、圆环或者超立方体，利用更有效的扩散算法可以减少消息数，参见习题12.6。

Van Leeuwen等人[LSUZ87]已经证明，可以通过交换更多消息，减少时间复杂度。进行k 405
次扩散过程（使消息复杂度达到$O(k\cdot|E|)$，在$O(p_0^{1/k}D)$个脉冲内就可以找到最小标识。

12.2.2 网络规模未知

在网络大小和直径未知时，利用选择性延迟，把废止技术应用在遍历算法中（7.3.1小节），可得到有效的选举算法。

如果进程p是选举算法的初始进程，p初始化遍历算法，且使令牌的标号为p。按照遍历算法转发令牌，直到令牌被一个进程清除，该进程已经看到一个更小标识，或者在进程p中终止它的遍历。如果出现后一种情况，p成为领导人。在每次跳跃中，初始进程p的令牌被延迟2^p个脉冲，即，如果进程在脉冲i接收到p的令牌，它就在脉冲$i+2^p$转发该令牌。

所得算法的正确性证明类似于定理7.17。设p_0是具有最小标识的初始进程，进程p_0接收到返回的令牌后，则被选中。不存在其他初始进程的令牌终止这个遍历，因为它已经被p_0或者只一个初始进程清除。

接下来要证明，选择性延迟是如何把消息复杂度限制到$O(W)$的，其中W是遍历算法的复杂度。进程p_0在$2^{p_0}W$个脉冲后被选，因为令牌进行W次跳跃，每次跳跃需要花费2^{p_0}个脉冲。在$2^{p_0}W$个脉冲后，每个进程已经看到标识p_0，因此在那个脉冲或者其后的脉冲，不会转发其他的令牌。初始进程$p\neq p_0$的令牌至多被转发$(2^{p_0}W)/2^p$次，因为它在$2^{p_0}W$脉冲后不再转发这些令牌，在脉冲P之前，至多被转发$P/2^p$次。设S表示初始进程的集合。传递的消息总数界限为W（p_0的令牌的跳数）加上除p_0的令牌以外的令牌的跳数：

$$\begin{aligned}\text{复杂度} &= W+\sum_{p\in S\setminus\{p_0\}}(p\text{的令牌跳数})\\ &< W+\sum_{p\in S\setminus\{p_0\}}\frac{2^{p_0}W}{2^p}\\ &< W+\sum_{i\geq 0}\frac{W}{2^i}=2W\end{aligned}$$

消息的内容可按时间编码，可用两条消息传输每个令牌，每条消息占一位。这就证明了 406
选举算法的位复杂度为$O(W)$。在这种情况下，按时间编码并不会进一步增加时间复杂度。因为令牌在每个节点上所遭受到的延迟超过了它的传输所需的时间。算法的时间复杂度也不是N的函数，而是进程的最小标识p_0的指数（exponential）函数。

12.2.3 补充结果

1. 比较算法

本节中描述的算法不是比较算法。在比较算法中，对进程标识允许的惟一操作是比较。直到依赖标识的值到达，才能对脉冲计数。

Frederickson和Lynch[FL84]已经证明，在同步环上，选举的比较算法至少交换$\Omega(N \log N)$条消息。即，异步算法的下界对于同步比较算法成立。同时，Frederickson和Lynch证明，如果时间复杂度为N的函数，可能的标识的集合大小不限，每个算法的行为就像是最坏情况下的比较算法，这蕴含着消息复杂度至少为$\Omega(N \log N)$。

这表明消息复杂度的降低，是以将时间复杂度增加到一个不是N的函数的值为代价。对于网络大小未知，是否存在同步选举算法，具有上述给定算法的消息复杂度，且时间复杂度为最小标识的多项式（polynomial）函数，还仍是一个尚待解决的问题。

2. 不同时开始

如果不假设（如我们隐含地所做的）每个进程同时初始化算法（所有进程在脉冲0开始计数），选举问题就会变得更难一些。通过扩散唤醒消息，当某些进程开始时，能够保证所有初始进程在D个脉冲之内开始执行算法，对于本节的算法（改编版）就足够了。然而，扩散消息的代价为$O(|E|)$。

3. 团上的选举

利用本节的算法，在团上选举领导人的消息复杂度为$O(N)$。然而，这些算法要求，所有进程在同一个脉冲开始执行算法。增加一个初始化阶段，在该阶段中扩散一条唤醒消息，使消息复杂度增加了$2|E|$，达到$\Omega(N^2)$。Afek和Gafni[AG91]分析了团上选举算法的复杂度，证
407 明如果不假设同时开始，需要的消息数为$\Theta(N \log N)$，并且这是一个充分条件。他们提出了消息复杂度为$\Theta(N \log N)$、时间复杂度为$O(\log N)$的算法。

12.3 同步器算法

本节考虑完全异步网络上的几个同步器算法。这些同步器算法的复杂度由以下4个参数度量。

M_{init}，初始化阶段的消息复杂度。

T_{init}，初始化阶段的时间复杂度。

M_{pulse}，模拟每个脉冲的消息复杂度。

T_{pulse}，模拟每个脉冲的时间复杂度。

当进程发送某次脉冲的消息时，称它模拟那次脉冲。可用任何进程的第一步和第一次脉冲的最后一次模拟之间的最大时间，度量初始化阶段的时间复杂度。一次脉冲的时间复杂度定义为一次脉冲的最后一次模拟和下一次脉冲的最后一次模拟之间的最大时间。在这两种情况下，时间复杂度的度量都是在定义6.31的理想计时假设下进行的。脉冲的消息复杂度只对附加消息计数，不对模拟算法中的消息计数。

消息复杂度为M、时间复杂度为T的同步算法，可用$M_{\text{init}} + M + M_{\text{pulse}}T$条消息、$T_{\text{init}} + T_{\text{pulse}}T$个时间单位模拟。表12-3给出了本章同步器算法的概览。

12.3.1 简单同步器

在本章的所有同步器中，进程保证当前脉冲的所有消息都已被接收到，此时正是所有消

息得到处理，下次脉冲开始的时刻。在ABD同步器中，这种保证基于时钟可用性，但是对于完全异步系统的同步器，就不可能保证这一点。因为所有异步算法可以用消息驱动的形式写出，每个进程必须接收一条消息，以触发下一次脉冲，这蕴含着模拟每个脉冲至少需要N条消息。

在最简单的同步器中，每个脉冲中的每个进程只向每个近邻发送一条消息。如果模拟的算法在某些脉冲中并不发送消息，同步器就会加上一条“空”消息。如果模拟的算法发送多于一条的消息，这些消息就会被打包成同步器中的一条消息。每个脉冲中的每个进程只能接收每个近邻的一条消息，当接收到每个近邻的脉冲i的消息时，开始处理所有消息，然后开始第$i+1$个脉冲，参见图12-2所示的算法。一旦跟随者接到第一条消息，初始者自发地执行初始化过程。

408

```
var pulse_p   : integer ;
    state_p   : Z_p ;
    wait_p    : integer     init |Neigh_p| ;
    nwait_p   : integer     init |Neigh_p| ;

To initiate the algorithm:
    begin (* start first pulse spontaneously *)
          state_p := initial state ; pulse_p := 1 ;
          forall q ∈ Neigh_p do
                begin M_p[q] :={m ∈ MG_p(state_p) :
                                m has destination q} ;
                      send ⟨pack, M_p[q], pulse_p⟩ to q
                end
    end

{ A message ⟨pack, (m_1, ..), i⟩ has arrived }
    begin receive and store message ;
          if i = pulse_p then wait_p := wait_p − 1
                         else nwait_p := nwait_p − 1 ;
          while wait_p = 0 do
             begin (* M is the set of received messages of this pulse *)
                   state_p := d_p with (state_p, M) ⊢ d_p ;
                   pulse_p := pulse_p + 1 ;
                   wait_p := nwait_p ; nwait_p := |Neigh_p| ;
                   forall q ∈ Neigh_p do
                      begin M_p[q] :={m ∈ MG_p(state_p) :
                                      m has destination q} ;
                            send ⟨pack, M_p[q], pulse_p⟩ to q
                      end
             end
    end
```

图12-2 简单同步器算法

$pulse_p = i$的进程p只能接收第i个脉冲和第$i+1$个脉冲的消息。早先脉冲的所有消息已经接收到。在p发送它的第$i+1$个脉冲的消息之前，没有它的近邻能够发送脉冲数大于$i+1$的消息。因此p只需保持当前（$wait_p$）脉冲和下一次（$nwait_p$）脉冲所期望的消息数的信息。因此传输脉冲数模2就够了。

409

初始化模拟过程不需要附加消息。每个进程或者自发地，或者当接到一条消息的时候，才发送初始脉冲的消息。当至少有一个进程开始模拟时，所有进程在D个时间单位内都会加入模拟过程。因此，$M_{init} = 0$和$T_{init} = D$。

每个脉冲只需要交换$2|E|$条消息，在最坏情况下，所有这些消息都是附加消息。因此$M_{pulse} = 2|E|$。如果脉冲i的最后模拟出现在时间t，在时间$i+1$收到脉冲i的所有消息。因此，脉冲$i+1$的最后模拟最迟出现在时间$t+1$。因此，$T_{pulse} = 1$。

简单同步器算法的应用不需要领导人，不需要网络规模知识或者其他拓扑信息，也不需要对网络命名。简单同步器算法的通用性表明，所有同步算法可用异步网络模拟。因此，能用同步算法求解的所有问题也可在异步系统中求解。

12.3.2　α、β和γ同步器

在简单同步器中，每次脉冲中的每个进程与它的每个近邻通信，使得模拟每个脉冲的消息复杂度达到$\Theta(|E|)$。如果经过子拓扑结构进行通信，则可降低消息复杂度（以增加时间复杂度为代价）。同步器导致的通信的目标是让进程得知，它的近邻在当前脉冲中发送的所有消息已经收到。

如果在某个脉冲进程p所发送的所有消息被接收后，则进程p在当前脉冲变成安全的。对于模拟算法中的每条消息，都发送一条确认消息，当进程p收到每条消息的确认消息时，它变成安全的。现在，同步器必须提供进程之间的通信，通过通信，进程得知它的所有近邻是安全的。当进程p已经验证了对于脉冲i它的所有近邻都是安全的之后，p就处理脉冲i的消息，改变状态，并开始第$i+1$个脉冲。

下面提出的α和β同步器是以后所要给出的γ同步器的两种特例。这三种同步器是由
410 Awerbuch[Awe85a]提出的。

1. α同步器

第一种同步器，称为α同步器，非常类似于简单同步器。在同步器中，经过每个信道进行通信：每个安全进程向它的每个近邻发送消息**<iamsafe>**。当进程接收到每个近邻的消息**<iamsafe>**时，进程知道它的所有近邻是安全的。

初始化过程不是必要的，很显然，对于每个脉冲，同步器发送$O(|E|)$条消息。如果脉冲i的最后一次模拟发生在时间t，那么到$t+1$，这个脉冲的所有消息已被接收，到$t+2$，所有确认消息已被接收。因此，到$t+2$，每个进程是安全的，到$t+3$，消息**<iamsafe>**已被接收，并允许所有进程模拟第$i+1$个脉冲，因此$T_{pulse} = 3$。

2. β同步器

在β同步器中，经过生成树进行通信。当子树中的所有进程安全时，进程向它的父节点发送消息**<ts>**（“树是安全的”），这就是当进程自身是安全的，并从它在树中的每个子节点接收到消息**<ts>**时出现的情况。当根节点是安全的，并已经接收每个子节点的消息**<ts>**，则所有进程是安全的。根节点经过生成树发送消息**<pulse>**，让每个进程知道在当前脉冲中的所有进程是安全的。接收消息**<pulse>**表明，所有进程是安全的。这蕴含着，接收进程的所有近邻是安全的，因此进程进行到下一个脉冲。

同步器需要初始化阶段，在这一阶段中计算生成树。7.3节表明，假如惟一标识可用，利用$\Theta(N \log N + |E|)$条消息，可以计算出生成树。利用$O(N)$个时间单位的方法是可能的。集

中式算法利用O(|E|)条消息，在O(N)时间单位内，可以计算出生成树。

除了确认消息，每个脉冲要求经过树的每条边，发送消息<ts>和<pulse>，共有O(N)条控制消息。所花费的时间与树的深度成正比。在最坏情况下时间为Ω(N)。

3. γ同步器

γ同步器要求初始化阶段，在这一阶段中将网络划分成簇。在每一簇内，算法操作如同β同步器。在簇之间，算法操作如同α同步器。

每一簇包含一个簇生成树和一个簇领导人，它是这棵簇生成树的根。对于每两个近邻簇（称两个簇是近邻，如果在这两个簇中的节点之间，存在一条边），选择两近邻簇之间的一条首选边（preferred edge）。同步过程由5步组成。 411

（1）如果子树中的所有进程是安全的，即，如果进程是安全的，并且已经收到每个子节点的消息<ts>，则进程向簇树中的父节点发送消息<ts>。

（2）如果簇的领导人是安全的，并且已经收到每个子节点的消息<ts>，那么，簇中所有进程是安全的，经过簇生成树发送消息<cs>（“cluster safe”，簇是安全的），让簇中每个进程知道，簇中所有进程是安全的。

（3）依附于首选边（该首选边通向另一簇）的簇中的进程，向其他簇发送消息，表明这个簇是安全的。为此，当接到消息<cs>时，就通过首选边发送消息<ocs>（“our cluster safe”，我们的簇是安全的）。

（4）在每个簇树中，向上发送消息，报告所有近邻簇是安全的。当由它的生成树中的首选边连接的所有近邻簇被报告是安全的，在簇树中，进程就向父节点发送消息<ncs>近邻簇是安全的）。

（5）当簇领导人接收到每个子节点的消息<ncs>和每条依附首选边的消息<ocs>时，那么所有近邻簇是安全的。当这种情况发生时，簇自身是安全的，经过簇树发送消息<pulse>，触发簇中每个进程开始下一次脉冲。

当进程接收到消息<pulse>时，它的簇和近邻簇是安全的。这蕴含着，所有它的近邻是安全的。

为了确定同步器的复杂度，对于簇c，设E_c是c中树边和首选边的数目，H_c是c中树的最大高度。因为同步器经过每个树边发送四条消息（<ts>，<cs>，<ncs>和<pulse>），经过每条首选边发送两条消息（两条消息<ocs>），模拟每次脉冲，M_{pulse}为O(E_c)。设t是脉冲i最后模拟的时间。在时间t+2，所有进程是安全的，在时间t+2 + H_c，领导人知道所有簇是安全的，而在时间t +2 + 2H_c，所有进程已经收到消息<cs>，并经过每条首选边发送消息<ocs>。在时间t + 3 + 2H_c，所有消息<ocs>已被接收，在时间t +3 + 3H_c，每簇的根知道所有近邻簇是安全的，且在时间t +3 + 4H_c，所有进程接收到消息<pulse>，模拟下一次脉冲。于是，T_{pulse}为O(H_c)。 412

如果把整个网络看作一个簇，那么E_c = N−1和H_c = O(N)，此时本同步器与同步器β相同。如果将每个节点看作一个簇，那么E_c = E和H_c = O(1)，此时同步器与同步器α相同。

4. 计算合适的簇

γ同步器的复杂度取决于算法中所用的簇。通过簇，我们特别进行如下规范：

（1）网络按照簇划分（连通子图）。

（2）每簇一个中心（领导人）和一棵有根生成树。

（3）每一对近邻簇之间有一条首选边。

定理12.6　对于每个k，$2 \leq k < N$，存在一个簇c满足$E_c < k \cdot N$且$H_c \leq \log N / \log k$。

证明。通过选择簇C_1，C_2，...逐步构造簇C_1，C_2，...。假设已经选择簇C_1，...，C_r，那么网络中存在剩余不属于任何C_1至C_r的簇的节点。

为了构造C_{r+1}，在剩下的节点中，选择一个节点p作为簇的领导人，并向簇中一层一层地增加节点。一个节点$\{p\}$是簇的第一层。在C_{r+1}的第i层构造完之后，令n表示C_{r+1}中的结点数，考虑不属于任何簇的第i层的节点的近邻集合S。如果S的大小至少为$(k-1)n$，则将S中的节点被加到C_{r+1}中作为下一层。否则，如果S中包含的节点数少于$(k-1)n$，那么C_{r+1}的构造完成。

簇的构造过程表明，一个有i层的簇（即，深度为$i-1$的生成树）至少包含k^{i-1}个节点，因此，H_c被$\log_k N = \log N / \log k$限定。

簇中树边数不超过N，因为树边形成网络的生成森林。对于每两个近邻簇，选择一条首选边，并使这条边属于这两个簇中第一个被建立的簇。当剩下的近邻节点数比簇的规模小$k-1$倍时，完成簇的构造。这蕴含着，对于大小为n的簇，以后至多要建立$(k-1)n$个近邻簇。因此，
413 根据这个方针，大小为n的簇至多有$(k-1)n$条首选边，首选边数不会超过$(k-1)N$。这就证明了$E_c < k \cdot N$。□

同步	M_{init}	T_{init}	M_{pulse}	T_{pulse}	Remarks
ABD	$2\|E\|$	$D\mu$	0	2μ	
简单	0	D	$2\|E\|$	1	
α	0	D	$2\|E\|$	3	确认
β	$\Theta(N \log N + \|E\|)$	$O(N)$	$2N-2$	$2N$	确认
γ	$O(k \cdot N^2)$	$O(N \frac{\log N}{\log k})$	$O(k \cdot N)$	$O(\frac{\log N}{\log k})$	$2 \leq k < N$确认

图12-3　同步器算法一览

Awerbuch[Awe85a]给出了实现这个构造的分布式算法。算法利用$O(k \cdot N^2)$条消息。时间复杂度为$O(N \log N / \log k)$个时间单位。

12.4　应用：广度优先搜索

作为同步器的应用，本节提出几种计算网络广度优先搜索（BFS）生成树的算法。网络G的一棵生成树T是*广度优先搜索树*（breadth-first search），如果对于每个节点，到树根的路径是G中的最小跳数路径。即，对于它的根，广度优先搜索树是一棵最优汇集树，以最小跳数距离度量（参见定理4.2）。

广度优先搜索树在设计消息和时间有效算法方面起着重要的作用，因为可以用最少的时间进行信息的最优消息广播。利用BFS树，可使Santoro和Khatib（4.4.2小节）的树标号模式更有效地工作。Frederickson[Fre85]利用BFS树，有效地解决了最短路径问题。

集中式算法可进行BFS树的计算，该集中式算法从树根开始（假设已定义了优先级）。根据参数N和$|E|$，本节分析了广度优先搜索算法的复杂度。观察可见，树的深度不超过N。经过更仔细的分析，可以建立关于参数N、$|E|$和D的复杂度。树的深度受D所限。

414 在12.4.1小节，首先证明，BFS树的计算在同步网络中是非常简单的。事实上，回波算法每次执行都会产生一棵BFS树。在12.4.2小节中，同步算法将与同步器算法组合，并通过列表

给出了几种方法，复杂度之间的比较。12.4.2小节介绍的异步算法容易得到，且易于理解，但是较之其他算法，它们不是很有效。我们最后讨论了由Frederickson[Fre85]（12.4.3小节）提出的两个异步算法。

12.4.1 同步BFS算法

一种建立BFS树的自然方法就是按照层次计算树。初始时，根形成第0层，在完成第i层之后，通过添加第i层节点的近邻，就可将树扩展到第$i+1$层。

在计算广度优先搜索树的同步算法中，每个进程只在一个脉冲中，向它的每个近邻发送消息。初始进程在第一个脉冲中这样做。如果i是进程接到一至多条消息的第一个脉冲，那么其他每个进程在脉冲$i+1$发送消息。在脉冲i中第一次接收到消息的进程，将其自身赋给第i层，并从脉冲i中，那些可以从其处接收到消息的那些进程中，选择一个进程作为自己的父节点。

定理12.7 *根据上述算法，与初始进程相距为i的进程，在脉冲$i+1$发送消息，并且如果$i>0$，它选择距初始进程为$i-1$的一个进程作为它的父节点。*

父节点的选择蕴含着，所得的树是一棵广度优先搜索树，因为每个进程只发送一次消息，消息复杂度为$2|E|$。由于脉冲N之后，每个进程已经决定了它的父节点，则时间复杂度为$O(N)$。

12.4.2 与同步器组合

我们把同步算法与早先本章前面讨论的同步器之一进行结合，可得ABD网络或异步网络的广度优先搜索算法。图12-4给出了所得异步算法的复杂度。用这种方法得到的算法易于理解，但不是最著名的异步算法。

415

算 法	消息复杂度	时间复杂度
同步	$2\|E\|$	$O(N)$
同步网+ ABD	$O(\|E\|)$	$O(N)$
同步网+ α	$O(N\cdot\|E\|)$	$O(N)$
同步网+ β	$O(N^2)$	$O(N^2)$
同步网+ γ	$O(k\cdot N^2)$	$O\left(N\frac{\log N}{\log k}\right)$
简单	$O(N^2+\|E\|)$	$O(N^2)$
高级	$O(N\sqrt{\|E\|})$	$O(N\sqrt{\|E\|})$
[AG85]	$O(\|E\|\cdot 2^{\sqrt{\log N\log\log N}})$	$O(N\cdot 2^{\sqrt{\log N\log\log N}})$

图12-4 广度优先搜索算法的复杂度

12.4.3 异步BFS算法

本节讨论三个广度优先搜索异步算法。与前面小节所讨论的算法相比，这些算法具有较低的复杂度。在异步算法设计中，读者不应认为，在异步算法的设计中，采用同步器的概念是毫无用处的。在以下算法的设计中，隐含地利用了同步器算法。

广度优先搜索树通常是按照层次建立的。在完成第i层之后，在那层的（此时，它们还不是树中的节点）节点的近邻被加入到树中作为第$i+1$层。考虑这样一种情况，在第$i+1$层的构造完成之前，第$i+2$层的构造就已开始。在层$i+1$中节点的近邻在第$i+2$层被加入到树中，但是，

随后，这样的节点也可能成为第i层中节点的近邻，实际上，节点的层应该是i+1。

以下给出的简单算法通过对每两层的树之间进行同步扩展避免了这种情况。按照同步器β进行同步，并应用于已构造的部分树中。在更高级的算法中，每l层后，就会发生一次同步。这降低了由于同步带来的开销。但要求一个节点在计算过程中，可能位于不同层的树中。

在最终BFS树中的节点的层次等于在整个网络中它与根的距离。层为f的节点只有层为
416 $f-1$、f、$f+1$的近邻，参见习题12.9。

1. 简单算法

为了开始算法，初始进程为自己赋值为0层，此后BFS树的第0层构造完成。树按照层次构造，当第f层构造完成时，在第f层的每个节点知道自己在第f层，且知道它近邻中的哪些节点在$f-1$层。

假设第f层构造完成，但是算法还未终止。为了开始第$f+1$层的构造，初始进程将消息<**forward**，f>沿树向下广播。在第f层的节点p，一旦收到消息<**forward**，f>，就把消息<**explore**，$f+1$>发送到p还不知道是在第$f-1$层的那些近邻中。消息<**explore**，$f+1$>的发送过程见图12-5。

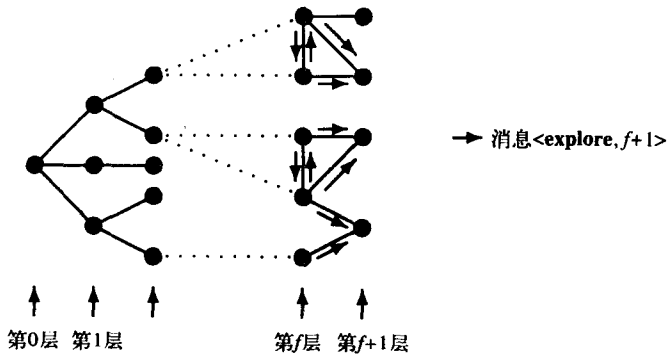


图12-5　第$f+1$层的构造

第f层的所有节点发送消息<**explore**，$f+1$>后，则距离初始进程为$f+1$的节点从第f层的每个近邻接收到这样一条消息。节点接收到的第一条消息<**explore**，f>定义了第f层以及BFS树中节点的父节点，并用消息<**reverse**，*true*>回答。随后到达的消息<**explore**，f>用消息<**reverse**，*false*>回答。已知这些消息的所有发送者在第$f-1$层。观察可见，当第$f+1$层构造完成时，这蕴含着，所有<**explore**，$f+1$>消息都得到回答，在第$f+1$层的每个进程知道它的哪些近邻处在第f层。

417 在第f层的节点也可能收到消息<**explore**，$f+1$>，即收到层也为f的近邻的消息。这些消息不用消息<**reverse**，b>回答，因为发送给这个近邻的消息<**explore**，$f+1$>被解释为<**reverse**，*false*>消息。

在第f层的节点等待对于它们所发送的所有消息<**explore**，$f+1$>的应答。应答或是消息<**reverse**，b>（由第$f+1$层的新节点发送，它是节点的子节点，当且仅当$b = true$），或是消息<**explore**，$f+1$>（由层也为f的近邻发送）。层至多为f的每个节点向它在树中的父节点转发消息<**reverse**，b>，报告由其子树中的节点所发送的所有消息<**explore**，$f+1$>已经得到应答。b的值为*true*，当且仅当新节点已经被加入到子树中。在第f层的节点在收到自己发送的对每个

消息<**explore**，f +1>的应答时，发送消息<**reverse**，b>，如果在这些应答消息中，至少有一条消息为<**reverse**，*true*>，则b = *true*。如果收到它的每个子节点的消息<**reverse**，b'>，在小于f的层上的节点，就向它的父节点发送消息<**reverse**，b>。如果在所接收到的消息中，至少已有一条消息为<**reverse**，*true*>，则b = *true*。

当根节点收到每个子节点的消息<**reverse**，b>时，第f +1层的构造终止。如果没有节点被加入树中（所有接收到的消息包含值*false*），则树的构造终止。否则继续构造第f +2层。

定理12.8 *利用集中式异步算法构造广度优先搜索树，消息复杂度为*$O(N^2)$，*时间复杂度为*$O(N^2)$ *个时间单位。*

*证明。*至多N层后，构造终止，每一层的每条树边至多传送一次转发或者**explore**消息，并发送一条**reverse**消息来应答。而连接第f层的两节点的非树边，在第f +1层的构造过程中，传送两条消息<**explore**，f +1>。而连接两节点（一个在第f层，一个在第f +1层）的非树边，在第f +1层的构造过程中，传送一条消息<**explore**，f +1>和一条消息<**reverse**，*false*>。在其中的任何一种情况下，非树边为整个算法只传送两条消息。因此，消息的复杂度为$2N(N-1)+2(|E|-(N-1)) \leqslant 2N^2+2|E| = O(N^2)$。层$f$ +1的计算需要$2(f+1)$ 时间单位。这蕴含着算法至多在$O(N^2)$ 个时间单位后终止。 □

2. 高级算法

在简单算法中，每一层的探测由根进行同步（通过逆转和转发消息）。从复杂度分析可证明，同步消息控制着算法的复杂度。如果网络是*稠密的*（即$|E|$几乎为$\Omega(N^2)$），由于$|E|$是消息数的下界，所以不能改进消息复杂度。如果网络是*稀疏的*（即$|E|$为$O(N^2)$），可以在同步的轮数之间，探索几层，来减少同步的开销。 418

假设完成了树的第f层的构造。为了构造下一个l层，*初始进程沿树向下广播消息*<**forward**，f>。一旦接收到这个消息，第f层的节点向那些近邻发送消息<**explore**，f +1，l>，这些近邻还不为第f−1层所知。**explore**消息的第一个参数表示接收者的新层数，如果接收者变成发送者的一个子节点。第二个参数表示从消息的发送者开始必须被探索的层数。

初始时，节点在层∞。如果一条消息<**explore**，f，l>到达节点p，就将f与p的当前层$level_p$比较。如果$f > level_p$，就发送消息<**reverse**，*false*>应答。如果$f < level_p$，p将消息的发送者作为它的父节点，并把层变为f，并且如果$l > 1$，就向那些近邻发送消息<**explore**，f +1，l−1>，这些近邻还不为第f +1层所知（后者就是这样一种情况，即，如果以前就从这个近邻接收消息<**explore**，f+2，l>）。第f+1层至$f + l$层构造的终止性，可通过将**reverse**消息扩散以应答**explore**消息和转发消息来进行检测。就像在简单算法中那样。

定理12.9 *如果*$|E|$*和*N^2*之间的比率（近似地）已知，可用集中式异步算法构造一棵广度优先搜索树，消息复杂度为*$O(N\sqrt{E})$，*时间复杂度为*$O(N\sqrt{E})$。

*证明。*经过大约N/l轮同步后，构造过程终止，每一条树边至多传送$2N/l$条转发消息和对转发消息的应答，共需$O(N^2/l)$。每条边至多传送l条探索消息和l条对它们的应答，共有$O(l|E|)$ 条消息，这使得消息复杂度达到$O(N^2/l + l|E|)$。因为第f +1层至f +l层的计算需要$O(f+l)$ 个时间单位，算法的时间复杂度为$O(N^2/l)$。

通过选择l为$\sqrt{N^2/E}$，可得定理中所规定的复杂度的界限。 □

3. Awerbuch-Gallager算法

通过引入更多的同步机制，Awerbuch-Gallager[AG85]可以进一步减少广度优先搜索的复 419

杂度。在各轮同步之间所探索的l层称为带（strip）。在Awerbuch-Gallager算法中，不仅在每两条带的探索之间存在全局同步，而且通过探索过程中建立的生成森林，每条带自身的探索也是同步的。算法过于复杂，在这里不作详细的解释。但为了完整性，在表12-4中列出了它的复杂度。

12.5 Archimedean 假设

取决于物理时钟的可用性，进程可能看到时间，也可能看不到。即使看不到时间，所谓的Archimedean假设也可能被证明是合理的。按照这一假设，进程的某一步或者消息传输的某一步所消耗的最大可能时间和最小可能时间的比率有界。

即使系统没有装备测量实际时间的设备，也可以获得时钟的粗略表示，例如，通过计算每个处理器执行的指令数，以及定义每1000条或者10 000条指令为时钟的一次嘀嗒。进程中所“扫过的时间”为那个进程中所计算的时钟嘀嗒数。接下来，用t_1和t_2表示物理时间的实例。时间t_1和t_2可从上下文中理解，设m表示t_1和t_2之间任意进程的两次连续时钟嘀嗒之间的最小时间，设u表示t_1和t_2之间任意消息的最大传输延迟与t_1和t_2之间任意进程的两个连续时钟嘀嗒之间的最大时间之和。称比率u/m为间隔（t_1，t_2）的Archimedean比率。

定义12.10 *如果比率u/m有界s，则系统是*Archimedean*的（有*Archimedean*比率s）。*

给定t_1和t_2，在t_1和t_2之间时钟嘀嗒和消息出现的次数是有限的。这蕴含着，存在s，满足系统是Archimedean的，且具有比率s。对于许多系统，这可能是一个合理的假设，即，如果存在s_0，满足系统是Archimedean的，且具有位于由系统执行的算法的开始和终止之间的比率s。这蕴含着，在系统中，（相当弱的）同步假设被验证，但是否这种形式的弱同步可用于设计有效的分布式算法，仍然是一个问题。

420 已经证明，利用完全异步算法且不依赖于定时假设会更好。有如下几个原因：

（1）系统中最慢组件（与最快组件一起）决定了Archimedean比率值。如果算法的时间复杂度受到比率的影响（通常如此），那么最慢的组件决定了整个系统的速度。

（2）对延时假设的分析相当困难，并且会导致不正确的算法。

（3）算法执行中任意延迟的引入（例如，在被用于Dijkstra、Feijen和Van Gasteren的终止检测算法的环算中）可能使得Archimedean假设无效。因此，如果算法依赖于Archimedean假设，就不能使用这项算法技术。

（4）用更快或更慢的组件替换系统中的组件，可能改变Archimedean比率，也可能使得算法传输错误的结果。这种类型的错误是极其难以跟踪和修复的，因为错误原因是用功能上等价的组件代替系统中的组件所至。

根据这些假设，有两种利用Archimedean假设的方式。称算法是*健壮的*（robust），如果它的正确性不依赖于假设，即，算法在一个完全异步系统中是正确的。Archimedean假设用于算法的复杂度分析，它可能揭示出，如果这个比率接近于1，算法就表现得更好。称算法是*临界的*（critical），如果它的正确性依赖于Archimedean假设，即，在执行过程中，如果Archimedean比率超过设计参数s_0，则算法可能产生错误的结果。“按时间编码”和发送“隐消息”技术都是关于Archimedean假设临界的算法，而“选择性延迟”是健壮算法。

Vitányi[Vit85]表明，利用选择性延迟，在具有一条消息时间复杂度的Archimedean系统中，可以健壮地进行选举，参见习题12.11。Spirakis和Tampakas[ST89]已经提出了几种用于不同任

务的健壮和临界的分布式算法。

习题

12.1节

12.1 改进通过“按时间编码”发送信息的协议，使得传输消息m的时间为$O(\sqrt{m})$个时间单位，且利用常数个位。 421

12.2 定理9.8阐述：

如果环规模未知，不存在计算非-常函数f的确定进程-终止算法。

这个定理对于同步网络也成立吗？

12.3 证明图12-1所示的算法中，在第i个脉冲（局部时钟（$2i\mu$，$2(i+1)\mu$）的间隔）执行中，进程只能接收第i个脉冲和第$i+1$个脉冲的消息。

（因而，消息的脉冲数可由接收时间和脉冲数的基偶性确定）

12.4 假设按照这样一种方式扩展图12-1中的算法的同步阶段，进程p在时间τ记录时钟时间$\delta_{pq} = CLOCK_p^{(\tau)}$，在此时，进程$p$接收到$q$的消息<start>。（$\delta_{pq} = 0$如果这个消息引起$init$在$p$中执行。）证明当$p$接收到$q$的脉冲$i$的消息时，$p$的时钟值介于$\delta_{pq}+2i\mu-\mu$和$\delta_{pq}+2i\mu+\mu$之间。

（因而，在时钟时间c接收到q的消息的脉冲数为$\lfloor (c-\delta_{pq}+\mu)/2 \rfloor$，因此，与消息脉冲数有关的信息不需包含在消息中）。

12.5 修改图12-1所示的算法使其能处理内部处理时间非0的情况。而是假设执行$\mathbf{P}_p^{(i)}$的程序体所需时间为λ。

12.2节

12.6 给出团、圆环以及超立方体上扩散消息的算法，它们需要$N-1$条消息以及$O(D)$个时间单位。（必须假设圆环和超立方体是有标号的。）

12.7 给出规模已知的网络同步选举算法，在这种情况下，进程无需都在同一个脉冲中开始选举，而是可以在不同的脉冲中初始化算法。 422

12.3节

12.8 把12.2节的同步算法和12.3节的同步器结合起来得到一个新的异步选举算法，确定这个算法的消息复杂度。

12.4节

12.9 证明在BFS树中，第f层一个节点的近邻，层为$f-1$、f或者$f+1$。

12.10 根据N、$|E|$和D（网络直径）分析广度优先搜索算法的复杂度。

12.5节

12.11 [Vit85]本习题的目的是研究选择性延迟在Archimedean网络中的效果。设给定Archimedean比率为s的网络。

把废止技术应用于消息复杂度为W的遍历算法进行选举。进程p的每条消息在每个进程中延迟$f(p)-1$个时钟嘀嗒。有一个单独的唤醒过程保证，在算法开始后的$D \cdot u$个时间单位内，每个进程开始遍历。

（1）证明算法在$D \cdot u + W \cdot u \cdot f(p_0)$时间单位内终止，其中$p_0$是具有最小标识的进程。

（2）证明在长为t的时间间隔内，进程p的令牌至多发送$1 + t/(f(p)-2)$次。

（3）导出算法的最坏情况下的消息复杂度公式。

423　（4）证明通过改变f，可以得到线性消息复杂度。

第三部分 容 错

第13章 分布式系统中的容错

本书前面研究的分布式系统中所达到的协同行为中，进程是可靠的（reliable）。由于几种原因，在假设进程可能发生故障的情况下，研究进程可能具有的协同行为是非常有吸引力的。这项研究是本书最后一部分的主题。容错问题的许多解决方案是特定的（*ad hoc*），同时，在收集的大量不可能性中，有时难以发现证明结构和基础的理论。另一方面，有些问题有着一流而完备的理论以及简单的解决办法。对它们的陈述在这本导论性的教科书中非常适合。

本章作为后续章节的导引，我们说明了利用容错算法的原因（13.1节）。随后，介绍了两种主要的容错算法，即健壮算法（13.2节）和稳定算法（13.3节）。

13.1 利用容错算法的原因

分布式系统中越来越多的组件意味着，在分布式算法的执行中，某些组件将发生故障。网络中的计算机可能发生故障，系统中的进程可能由于断开工作站而被错误地杀死，或者由于存储器发生故障，机器可能产生不正确的结果。现代计算机变得越来越可靠，使得单个计算机发生这类故障的概率降低。尽管如此，当它的组件数增加时，分布式系统中某处出现故障的机会可能任意地增大。每当发生故障时，为了避免重新起动算法，算法应被设计成能够恰当地处理这些故障。

424
↓
423

当然，在串行计算机中，在安全为主的应用中，或者计算运行了很长时间，产生了不可验证的结果，易于受到故障影响的问题是值得关注的。可进行某些内部检查，防止某些类型的错误（例如，模块检查以免于计算错误）。但是没有一种保护机制可以达到使程序完全无损或使代码免于错误更改。因此，用顺序算法和单一处理器的计算机系统进行容错计算的可能性非常有限。

因为在分布式系统中，由于处理资源的传播，这些系统具有部分故障（partial-failure）性质。无论出现哪种故障，通常只影响整个系统的一部分。组件数的增加极有可能使故障就发生在某些组件上，而且故障极不可能发生在所有组件上。这就使得当发生故障时，可由其他组件接管发生故障的进程的任务，从而导致性能下降，而不是全面的故障。确实，正如在下面章节中所看到的那样，有可能为这样的系统设计分布式算法，系统中的进程是有缺陷的，同时也可实现协调工作。

对于本质上不是分布式却要求高可靠性的计算机应用设计，部分故障性质使得利用分布式（"复制的"）体系结构成为有吸引力的选择。航天飞机的主计算机系统就是一个例子。Spector和Gifford[SG84]描述了它的研制。飞机主要由不需定制的微处理器控制，在设计中主要关注的是它在航行过程中处理器发生故障的可能性。最终控制系统由4个相同处理器组成，

每个处理器只进行同样的计算[⊖]，激励者号对结果进行投票，即使一个处理器发生故障，也能完全控制系统。(激励者号的物理实现表明，即使第二个处理器后来发生故障，系统也能继续运行。) 尽管复制是增加可靠性的有吸引力的一种选择，但是需要协调一群（不可靠）的处理器的算法设计远非那么简单。由于精确负责任务的软件中的一个错误，致使航天飞机首航延期（从1981年4月8日延期到1981年4月10日）正说明了这一点。

428 幸运的是，自从1981年以来，容错算法的研究取得了相当大的进展。基于复制技术的可靠应用也已成熟。

文献中出现了两种很不同的容错方法。本书研究了这两种方法。在*健壮算法*中，每个进程中的每一步经过仔细设计，以确保尽管发生故障，正确进程只执行正确的步骤。在*稳定算法*中，正确进程可能受到故障的影响，但是算法保证当进程恢复正确行为时，可以从任意一个配置中进行恢复。在13.2节和13.3节，我们分别简要地介绍了健壮算法和稳定算法，并在本章最后，对它们作了简明比较。

13.2 健壮算法

在算法的执行中尽管故障出现在其他进程中，健壮算法被设计成可以保证运行正确的进程其行为的连续性和正确性。这些算法依靠某种策略，如表决机制，通过该机制，当足够多的其他进程宣称已经接收到这个信息的时候，一个进程将只接受确切的消息。然而，如果进程已经损毁，就可能出现死锁，一个进程可能永远不能接收所有其他进程的信息。

13.2.1 故障模型

为了决定运行正确的进程如何保护自己，免受故障进程的影响，必须作出这样的假设，即进程*怎样*才可能发生故障。在以下的章节中，总是假设只有*进程*才可能发生故障，而信道是可靠的。因此，如果正确的进程向另一正确的进程发送消息，就可以保证在有限的时间内接收到消息。(故障信道可用某个所依附的进程中的故障来模型化，例如，省略故障。) 作为附加假设，我们总是假设每个进程可以向其他每个进程发送消息。

本书中利用下面的*故障模型*（fault model)。

(1) *初始死进程* 如果它在局部算法中没执行过一步，则称进程为初始死进程。

(2) *损毁模型*（crash model) 如果进程正确地执行局部算法到某一时刻，此后并不进一步执行，则称它是损毁的。

429 (3) Byzantine行为（Byzantine behavior) 如果它执行了与局部算法不一致的任意步，则称进程是Byzantine的。尤其是Byzantine进程发送的消息可能包含任意内容。

健壮算法提出的正确性要求总是指*正确*进程的局部状态（或者输出)。初始死进程从不会产生输出，且它的状态总是等于它的初始状态。如果损毁进程有输出，则它的输出是正确的，因为到损毁发生时，进程的行为是正确的。不用说，一个Byzantine进程的局部状态或者输出，可以是任意的，任何算法都不满足Byzantine进程的非平凡性要求。

1. 错误模型的层次

可将错误模型分为三个层次。首先，初始死进程可看作损毁进程的特例，即损毁出现在

⊖ 见[SG84]对于第5个（“hot spare”）处理器和第6个（“cold spare”）处理器的解释。

进程的第一个事件之前。第二，损毁进程可看作Byzantine进程的特例，因为对Byzantine进程假设的任意行为包括根本不执行任何一步。因此，容忍损毁要比容忍初始死进程要困难，而容忍Byzantine进程则更困难。由于规定不同，健壮Byzantine算法也是健壮损毁算法，而健壮损毁算法也是健壮初始死进程算法。另一方面，健壮初始死进程算法的不可能性蕴含着健壮损毁算法的不可能性，而健壮损毁算法的不可能性蕴含着健壮Byzantine算法的不可能性。

*省略故障*包括跳过算法中的某些步（例如消息的发送或接收），此后算法继续执行。省略故障是Byzantine行为的特殊情形。而损毁故障是省略故障的特殊情形（即，某一时刻后，所有步都被省略）。因此，省略故障的层次在损毁和Byzantine故障之间。

2. 混合故障和定时错误

初始死进程和损毁故障称为*良性*（benign）故障类型，而Byzantine故障称为*恶性*（malign）故障类型。对于几种分布式问题，如果$2t < N$，则N个进程的集合可以容忍t个良性故障，而对于能容忍t个恶性故障的健壮算法，要求$3t < N$（*异步计算模型*）。因为在实际中，常常不能排除恶性故障，但是与良性故障相比，恶性故障非常少。Garay and Perry[GP92]将一些结果扩展到混合故障模型，其中t个进程可能发生故障，当中的b个是恶性故障。在这种模型中，如果$2b + t < N$（*同步计算模型*），就可以实现其余进程的正确行为。 430

在同步分布式系统中，还有另一种故障模型，即进程在错误时间执行正确步（由于进程时钟的快或慢）。这种类型的错误进程行为称为*定时错误*（timing error）。

13.2.2 判定问题

健壮算法的研究是以所谓的*判定问题*为中心的，要求每个正确进程必须把“判定”值写入其输出中。所要求的判定值通常以十分平凡的方式依赖于进程输入值，这使得在无故障（可靠）的环境中解决这些问题相当容易。

对于判定的要求通常有三种类型，即*终止性*、*一致性*和*非平凡性*。

终止性要求说明，所有正确进程都将进行判定，即最终向输出写值。在确定性算法中，所有计算中都要求终止性，在概率算法中，只要求算法以概率1发生终止（参见第9章）。所有正确进程向其输出写值的要求排出了这样一种方法，就是进程必须等待接收来自超过$N-t$个进程的信息。t个进程的停止引起某个正确进程的不确定的等待，违反了终止性要求。

一致性要求在不同进程所进行的判定之间强加了一种关系。最简单的情形是，要求所有判定是相同的，*一致性问题*（consensus problem）就属于这种情况。在更复杂的问题中，定义一类输出向量，正确进程的判定形成该类中的一个向量。在选举问题中（第7章），例如，要求一个进程判定它*被选*（“1”），所有其他进程判定*失败*（“0”）。

要使一个分布式算法有用，要求终止性和一致性就足够了。在表明任务不可能求解的情形中，就需要有另外的要求。非平凡性要求排除了基于问题的固定输出的算法，其中每个进程不经通信就可判定。例如，一致性问题可用一个算法求解，在这个算法中，每个进程直接向输出写“0”。非平凡性要求表明，在算法的不同执行中存在两种本质上不同的输出（即在一致性的情形下，算法要进行写“0”的执行及写“1”的执行，当然，在一次执行内，所有判定是一致的）。 431

判定问题对大量分布式计算中的一般情况进行了抽象，正如我们现在所要讨论的。

（1）*提交-中止*　在一个分布式数据库中，涉及多个站点的事务处理，必须在所涉及的所有站点上或者不在任何这类站点上执行。因此在宣布更新这些站点的数据之后，每个站点要决定是否执行局部更新，并进行“yes”或“no”的表决。随后，所有正确站点必须进行判定，是否提交（commit）事务，若提交，意味着它将在各处执行，若中止，意味着它将不能执行。如果所有进程投票“yes”，进程必须判定提交；如果一些进程投票“no”，则判定结果一定是中止。一致性的含义是所有判定是相等的，问题是非平凡的，因为依赖于输入且对提交和中止都有要求。

（2）*分布式计算输入的一致性*　在系统中，当计算被复制时，就提高了可靠性，仅当计算是基于相同输入时，各个处理器的结果才会相等。有缺陷的向各个处理器发送不同值的传感器的影响，必须通过在处理器间执行一致性算法来消除。每个处理器的输入就是从传感器接收的值，每个（正确的）处理器必须对将在后续计算中使用的相同的值进行判定。通常要求输出是大部分输入的值，或者更少一些，可以是至少作为输入出现过一次的值。在这两种情况下，问题是非平凡的。

（3）*选举*　在选举问题中（参见第7章），要求一个进程判定成为领导人，而其他所有（正确）进程判定为非领导人。如果我们要求潜在的不同的进程能够成为领导人，问题就变成非平凡的。

（4）*近似一致*　当输入的一致性达不到时，对于某些应用，更弱的一致性就足够了。在
432 近似一致的问题中，每个进程输入为一个整数，取自给定的有限范围，如，1，…，k。要求进程的判定至多相差1，并且判定结果位于实际上作为输入出现的那些值中。也可参见14.3节关于本问题的继续阐述（ε-近似一致）。

13.2.3　第14章到第16章综述

由于存在大量关于健壮算法方面的成果，本书中，不可能完全讨论该领域的最新成果。第14章到第16章所呈现的材料是按照如下标准进行选择的。

（1）一些基本结果应该被包括：在异步系统中，不存在确定的一致性，在多达一半的进程中，概率算法容忍良性故障，或者在多达三分之一的进程中，容忍恶性故障。

（2）给出一些用于获得健壮性或证明不可能性的技术示例。

（3）示例应该包括，同步系统可以达到比异步系统更大的健壮性。这与第12章所提出的结果形成对比，第12章中的结果表明可靠的同步系统并不比可靠的异步系统更强大。

（4）故障检测器是有希望的新范型，它可以很快地成为应用技术，也应该被包括进来。

第14章研究了*异步系统*能够达到的健壮性。Fischer、Lynch and Paterson[FLP85]展示了一些基本的结果，对于一致性问题，不存在甚至能容忍一个损毁故障的确定算法。Moran and Wolfstahl[MW87]把这个结果推广到更加广泛的一类判定问题（任务）上，例如，包括选举问题，但不包括近似一致问题。这些工作的结论是损毁仅能被概率算法（或者同步算法）容忍。这个结果不能被增强以应用到弱初始死进程故障模型中，可以通过确定一致性和初始死进程的选举算法的存在性来证明这一点。随机化也有所帮助。Bracha and Toueg[BT85]提出的随机一致性算法是异步算法，能容忍$t < N/2$个损毁进程，或者$t < N/3$个Byzantine进程。

第15章研究了*同步系统*中能够达到的健壮性。不同于在异步模型中，进程的损毁可以通

过其余进程检测，因为正确进程的有限响应时间可以得到保证。从而可以达到更高一级的健壮性，这可以通过按照脉冲运行的系统的理想模型来证明（参见第12章）。确定性算法可以容忍$t < N/3$个Byzantine进程，$t < N$个损毁进程，如果对消息的鉴定是可能的，则可以容忍$t < N$个Byzantine进程。这个结果和第14章的不可能性的结果表明，对于完全异步网络，不存在健壮同步器。在有限同步（有物理时钟和有限的消息-延迟时间）的系统中实现脉冲模型，要求时钟精确同步。这是第15章其余部分所要讨论的主题。 433

第16章讨论故障检测，这是加强计算模型的另一种方法，就像同步，通常在分布式系统中以某种形式可用。正如人们可以比较各种同步模型，从中发现解决问题的所必要的需求。为了解决一致性问题，我们可以研究故障检测模型，从而了解输出必须如何精确反映实际的故障。

13.2.4 本书中没有涉及的主题

第14章和第15章中给出的结果表明，在分布式系统中可实现的健壮程度。许多问题和结果仍然有待进一步研究，以下我们给出其中的一些，并给出一些参考文献的线索。

（1）*同步假设的精细化* 本书中，只考虑了完全异步和完全同步的系统，这些系统在它们可实现的健壮性方面差异相当大。Dolev、Dwork、Lynth and Stockmeyer[DDS87]，[DLS88]在对同步作了中等假设之下，研究了系统的容错性。

（2）*可解任务的确定性* 本书中，展示了某些任务的可解性，以及其他任务的不可解性。Biran、Moran and Zaks[BMZ90]给出了可解问题和不可解问题的精确特征描述。

（3）*容错的复杂度* 除了判定哪些任务是可解的，也可能要研究对于可解任务，协议所消耗的计算资源量。考虑了几种复杂度的度量：消息复杂度，位复杂度和时间复杂度（本书中常常称为循环复杂度）。在文献[BGP92]中，可以找到一些结果的综述和同时满足几个下界的一致性协议。 434

（4）*动态系统和组员资格* 本书中，假设进程的集合是固定（静态）和已知（尽管某些进程可能失效）的。进程的静态集合适合于必须可靠地运行固定的、有限的时间的应用。例如，航天器控制系统，其中修复和重新配置离线进行。对于一段不确定的持续时间（例如，在操作系统中，或者在空中交通控制系统中），系统的容错操作要求进程能被修复，且进程集能被在线重新配置，即，不停止应用。通过执行组员资格协议，重新配置进程集合，在协议中，活动进程与系统中的进程集合达成一致。关于这些协议及其参考文献参见[RBS92]。

（5）*利用共享变量进行通信* 本书中，考虑通过消息传递进行的进程间的通信。有些作者研究了分布式系统的容错问题，系统中的通信是基于共享变量，参见Taubenfeld and Moran[TM89]。

（6）*无等待的同步* 一种更复杂的通信模型是进程可以共享任意对象，而不是寄存器。在这种模型中，进程之间所期望的交互可以根据对象的方法形式化地表示，算法的问题被形式化，作为对象的实现。对于哪个对象能够被实现，哪个对象不能被实现存在广泛的理论，参见[AW98]。

13.3 稳定算法

即使发生了故障，但故障数量有限，且通常必须精确知道故障模型，健壮算法也一直显

示出正确的协同行为。容错的第二种方法，稳定算法却不同。它允许任何数目的任意类型的故障，但是算法的正确行为被挂起，直到故障修复后的某一时间。稳定算法（有时也称自-稳定）可从任何系统配置开始，最终达到允许的状态，并按照从那时起的规范运行。因此，临时性故障影响逐渐消失，无需始终如一地初始化系统。

435 第17章研究了稳定算法。早在1974年[Dij74]就引入了稳定化的概念。并把它应用于算法中，以实现在由处理器组成的环上的互斥。讨论这些算法的原因是由于它们的历史价值以及它们是一流的和见解深刻的。本书中前面所解决的许多问题存在稳定解，例如，数据传输、选举、路由表计算和深度优先查找树。

健壮算法和稳定算法

稳定算法可以提供保护以避免所谓的瞬时故障（transient failure），即系统组件中的暂时错误行为。当物理条件暂时达到极限值时，这些故障可能在分布式系统的许多部分发生，包括存储器和处理器的错误行为。例如，当大量宇宙射线击中太空船时，它的控制系统就会受到影响，同时系统中的许多组件就会受到灾难性的破坏。当条件恢复到正常时，进程按照程序恢复运行，但由于它们的暂时错误行为，全局状态（配置）可能成为任意一种状态（配置）。稳定性质保证了收敛到所要求的行为。

健壮算法提供保护以避免有限数量组件的永久性故障（permanent failure）。在系统修复和重新配置过程中，幸存的进程维持着正确（尽管可能不是十分有效）的行为。因此，当服务的暂时中断是不可接受的时候，必须利用健壮算法。

Gopal and Perry[GP93]和Anagnostou and Hadzilacos[AH93]所发表的文章，研究了健壮且稳定的算法。Gopal和Perry的研究表明，如何将健壮协议进行自动修改（编译），使它变成既健壮又稳定的协议。Anagnostou和Hadzilacos说明不存在选举和计算环规模的健壮且稳定的算
436 法，并提供了赋给不同名字的一种（随机的）协议。

第14章 异步系统中的容错

本章研究异步分布式系统中判定问题的可解性。研究结果是以14.1节Fischer、Lynth and Paterson[FLP85]提出的基本结果为顺序排列的。并对一类判定问题不可能性的证明做了系统阐述，结果也被看作是一些假设，这些假设将判定问题的解排除在外。放宽假设，使得获得各种问题的实际解成为可能，正如在后续的几节中表明的那样。也可在14.1.3小节看到进一步的讨论。

14.1 一致性的不可能性

在本节，给出了Fischer、Lynth and Paterson[FLP85]提出的基本定理的证明，该定理阐明不存在异步、确定的、1-损毁、健壮的一致性协议。通过对算法的公平执行序列的推理来证明这个结果。我们首先引入一些表示方法（除了2.1节引入的表示方法之外），并给出在后续几节中可用的一些基本结果。

14.1.1 表示、定义及基本结果

事件中的序列$\sigma = (e_1, \ldots, e_k)$在配置$\gamma$中是可应用的，如果$e_1$在配置$\gamma$中是可应用的，$e_2$在配置$e_1(\gamma)$中是可应用的等等。如果最终得到的配置为$\delta$，则我们将它记为$\gamma \rightsquigarrow \delta$或者$\sigma(\gamma) = \delta$，使得从$\gamma$到$\delta$中的事件是显式的。如果$S \subseteq \mathbb{P}$且$\sigma$只包含$S$的进程中的事件，我们也记作$\gamma \rightsquigarrow_S \delta$。 437

命题14.1 *设序列σ_1和σ_2在配置γ中是可应用的，且不存在进程，同时出现在序列σ_1和σ_2中。那么，σ_2在配置$\sigma_1(\gamma)$中是可应用的，σ_1在配置$\sigma_2(\gamma)$中是可应用的，且$\sigma_2(\sigma_1(\gamma)) = \sigma_1(\sigma_2(\gamma))$。*

证明。不断应用定理2.19可得结论。 □

进程p有一个只读的输入变量x_p，一个只写一次的输出寄存器y_p，它的初始值为b。对于每个进程p，输入配置完全由x_p的值决定。通过将值写入y_p中，进程p可以对值进行判定（通常为0或1），初始值b不是一个判定值。假设正确进程在公平执行中，执行无限多的事件。为此，一个进程总是能够执行一个内部事件（可能为void）。

定义14.2 *在t-损毁的公平执行中，至少有$N-t$个进程执行无限多的事件，且发送给正确进程的每条消息被接收。（进程是正确的，如果它执行无限多的事件。）*

算法能够处理的错误进程的最大数称为算法的弹性（resilience），总是用t表示。本节中，证明了不存在弹性为1的异步、确定算法。

定义14.3 *一个1-损毁-健壮的一致算法，满足以下三个要求。*

（1）*终止性* *在每个1-损毁的公平执行中，所有正确进程进行判定。*

（2）*一致性* *在一个可达的配置中，如果对于正确进程p和q，$y_p \neq b$且$y_q \neq b$，那么，$y_p = y_q$。*

（3）*非平凡性* *对于$v = 0$和$v = 1$，存在可达配置，在这个配置中，对于某些p，$y_p = v$。*

对于$v = 0$、1，称配置是v-判定的（v-decided），如果对于某些p，$y_p = v$。称配置为判定

的，如果它是0-判定的或者1-判定的。在一个v-判定的配置中，某些进程已对v判定。称配置是v-价的（v-valent），如果从该配置可达的所有判定配置是v-判定的。称配置是双价的（bivalent），如果从该配置可达0-判定和1-判定配置。称配置是单价的（univalent），如果该配置是1-价的或者0-价的。在单价配置中，尽管进程还没有必要进行判定，但是最终判定已经隐含地确定。

438

称t-健壮协议的配置γ是分支点（fork），如果存在至多t个进程的集合T，以及配置γ_0和γ_1，满足$\gamma \rightsquigarrow_T \gamma_0$，$\gamma \rightsquigarrow_T \gamma_1$，且$\gamma_v$是$v$-价的。非形式地，称$\gamma$是分支点，如果$t$个进程的子集能够强行执行一次1-判定以及一次0-判定。以下命题阐明，在任何时刻，至多t个进程的损毁一定被其余进程救活。

命题14.4　*对于t-健壮算法的每一可达配置，以及至少$N-t$个进程组成的子集S，存在一个判定的配置δ，满足$\gamma \rightsquigarrow_S \delta$。*

*证明。*设γ和S如上所述，考虑达到配置γ的一次执行，该执行包含S中每个进程的无限多个事件，（此后进程中每一步都在S中）。这个执行是t-损毁公平的，并且S中的进程是正确的，因此它们可达判定。 □

引理14.5　*不存在可达的分支点。*

*证明。*设γ是可达的配置，T是至多t个进程的一个子集。

设S是T的补集，即$S = \mathbb{P} \backslash T$。$S$有至少$N-t$个进程，因此存在一个判定的配置$\delta$，满足$\gamma \rightsquigarrow_S \delta$（命题14.4）。配置$\delta$或者是0-判定的，或者是1-判定的。不妨假设它是0-判定的。

现在证明，对于非1-价的配置γ'，$\gamma \rightsquigarrow_T \gamma'$。设$\gamma'$是任一配置，满足$\gamma \rightsquigarrow_T \gamma'$。因为在$S$和$T$中的步是可交换的（命题14.1），存在配置$\delta'$，从$\delta$和$\gamma'$都可达。由于$\delta$是0-判定的，因此$\delta'$也是0-判定的，这表明$\gamma'$不是1-价的。 □

14.1.2　不可能性证明

首先我们利用问题的非平凡性证明，存在双价的初始配置（引理14.6）。随后，将要证明，从双价配置开始的每一个激活步不需迁移到单价配置中就可以执行（引理14.7）。这足以证明一致性算法的不可能性（定理14.8）。接下来，设A是1-损毁-健壮的一致性算法。

439

引理14.6　*存在A的双价初始配置。*

*证明。*因为A是非平凡的（定义14.3），存在可达的0-判定和1-判定配置。设δ_0和δ_1是初始配置，满足v-判定的配置从δ_v可达。

如果$\delta_0 = \delta_1$，则初始配置是双价的，结论成立。否则，存在初始配置γ_0和γ_1，满足从γ_v可达v-判定配置，且γ_0和γ_1在某个进程的输入中不同。考虑从δ_0开始，δ_1结束的初始配置序列，在这些配置中，一个进程中的下一个初始配置与上一个初始配置不同。（把输入位一个接一个颠倒过来可得该序列。）从序列中的第一个配置δ_0开始，可达0-判定的配置，从最后一个配置δ_1，可达1-判定的配置。因为从每个初始配置，可达一个判定的配置，可以将所描述的γ_0和γ_1作为序列中的两个后续配置。设p是γ_0和γ_1不同的进程。

考虑在γ_0中开始的公平执行，在执行中，p没有执行任何步。这个执行是1-损毁公平的，并且达到判定的配置β。如果β是1-判定的，则γ_0是双价的。如果β是0-判定的，观察可得，γ_0和γ_1仅在p中不同。在执行中，p没有执行。因此β是从γ_1可达的，这证明了γ_1是双价的。（更精确地，从γ_1是可达配置β'，β'和β仅在进程p的状态中不同。因此β'是0-判定的。） □

为了构造非-判定的公平执行，我们必须证明，每个进程可以执行一步，每条消息可被接收，且不需强加判定。设一步s表示接收和处理一条特定消息，或者由某一特定进程进行的自然迁移（内部或发送）。根据该步发生时进程的状态，可能导致不同事件。如果在传输中，消息的接收是可应用的。自然迁移步总是可应用的。

引理14.7 *设γ是可达的双价配置，s是γ中进程p的可应用步。存在事件序列σ，满足s在$\sigma(\gamma)$中是可应用的，且$s(\sigma(\gamma))$是双价的。*

证明。设C是没有应用s、从γ可达的配置的集合，即$C = \{\sigma(\gamma)$: s不在σ中出现$\}$。s在C的每个配置中是可应用的（回忆s只是一步，不是某个特定事件）。

存在C中的配置α_0和α_1，满足v-判定的配置是从s（α_v）可达的。为了证明这一点，按照γ的双价性，对于$v = 0$，1，从γ可达v-判定的配置β_v。如果$\beta_v \in C$（即没有应用s达到一个判定的配置），观察可见，s（β_v）仍然是v-判定的，因此，选择$\alpha_v = \beta_v$。如果$\beta_v \notin C$（即应用s达到一个判定的配置），选择α_v作为配置，从这个配置，s被应用。 440

如果$\alpha_0 = \alpha_1$，$s(\alpha_0)$是所要求的双价配置。进一步假设$\alpha_0 \neq \alpha_1$，考虑从γ到α_0和α_1的路径上的配置。称这些路径上的两个配置为近邻，如果其中一个配置经过一步达到另一个配置。因为0-判定的配置从$s(\alpha_0)$可达，1-判定的配置从$s(\alpha_1)$可达，由此可得

（1）存在路径上的配置γ'，满足$s(\gamma')$是双价的；或

（2）存在近邻γ_0和γ_1，满足$s(\gamma_0)$是0-价的，$s(\gamma_1)$是1-价的。

在第一种情况下，$s(\gamma')$是所要求的双价配置，结果得证。在第二种情况下，γ_0和γ_1中有一个是分支点，矛盾。不妨假设从γ_0经一步而得γ_1，即对于进程q中的事件e；$\gamma_1 = e(\gamma_0)$。现在$s(e(\gamma_0))$是$s(\gamma_1)$，因此它是1-价的，但这不是$e(s(\gamma_0))$为1-价的情形，因为$s(\gamma_0)$已经是0-价的。因此，e和s不进行交换，这蕴含着（定理2.19）$p = q$，但可达的配置γ_0满足$\gamma_0 \rightsquigarrow_p s(\gamma_0)$且$\gamma_0 \rightsquigarrow_p s(e(\gamma_0))$。由于前者是0-价的，后者是1-价的，$\gamma_0$是分支点，这是一个矛盾。□

定理14.8 *不存在异步、确定性的、1-损毁-健壮的一致性算法。*

证明。假设存在这样的算法，从双价初始配置γ_0开始，可以构造非-判定的公平执行。

直到配置γ_i，构造才完成。此时为s_i选择一个可应用步，它已经被保持可用了最长的可能的步数。由前述引理，以这样一种方式，即，s_i被执行且可达双价配置γ_{i+1}，对执行进行扩展。

构造给出了一个无限公平执行，在这个执行中，所有进程都是正确的，但是从未进行判定。 □

14.1.3 讨论

结果说明，对于一致性问题，不存在异步、确定性的、1-损毁-健壮判定算法。这不包含一类非-平凡问题的算法（参见13.2.2节）。 441

幸运的是，Fischer、Lynch和Paterson的结果所基于的某些隐含假设可以显式化，其结果对于任何假设的弱化是十分敏感的。尽管结果具有不可能性，但甚至在异步系统中和进程可能发生故障的地方，许多非平凡问题仍的确有解。

（1）*弱故障模型* 14.2节考虑初始死进程的故障模型，它比损毁模型要弱。在该模型中，一致性问题和选举问题确定可实现。

（2）*弱协同* 与一致性问题相比，14.3节考虑的问题，对于进程之间的协同要求较弱。并

且通过实例证明，这些问题中的某些，包括重命名问题，在损毁模型中是可解的。

（3）随机化　14.4节考虑了随机协议，协议中对终止性要求充分放松，使得即使在出现Byzantine故障时，问题也是可解的。

（4）弱终止性　14.5节考虑了另一种对终止性要求的放宽，即仅当给定进程正确时，才要求它的终止性。同时，Byzantine健壮解也是可能的。

（5）同步　在第15章进一步研究了同步的影响。

如果略去定义14.3中的三个要求之一，也可能得到相当平凡解。参见习题14.1。在习题14.2中，研究了省略所有输入组合都是可能的这样一个假设（引理14.6的证明中隐含地使用了该假设）。

14.2　初始死进程

在初始死进程模型中，在执行一个事件之后，进程不能发生故障。因此，在一个公平的执行中，每个进程或者执行0个事件，或者执行无限多个事件。

定义14.9　*在t-初始死进程的一次公平执行中，至少有$N-t$个进程是活动的，每一个活动进程执行无限多个事件，发给正确进程的每条消息被接收到。*

在t-初始死进程健壮的一致算法中，每个正确进程在每个t-初始死进程公平执行中进行判
442 定。它的一致性和非平凡性的定义如同损毁模型。

因为在发送一条消息之后，进程不会发生故障。对于等待接收进程p的消息的进程，如果它知道进程p已经发送了至少一条消息，则它是安全的。将要证明，只要发生故障的进程占据少数（$t < N/2$），在初始死进程模型中，一致算法和选举算法都是可解的。大量的初始死进程不能被容忍（参见习题14.3）。

1. 正确进程子集上的一致性

在Fischer、Lynch and Paterson[FLP85]提出的第一个算法中，每个正确进程都计算正确进程的同一个集合。算法的弹性为$\lfloor (N-1)/2 \rfloor$。设L代表$\lceil (N+1)/2 \rceil$，可见，至少有L个正确进程。算法分两个阶段执行，参见图14-1所示的算法。

```
var Succ_p, Alive_p, Rcvd_p : sets of processes      init ∅ ;

begin shout ⟨name, p⟩ ;
      (* that is: forall q ∈ ℙ do send ⟨name, p⟩ to q *)
      while #Succ_p < L
            do begin receive ⟨name, q⟩ ; Succ_p := Succ_p ∪ {q} end ;
      shout ⟨pre, p, Succ_p⟩ ;
      Alive_p := Succ_p ;
      while Alive_p ⊈ Rcvd_p
            do begin receive ⟨pre, q, Succ⟩ ;
                     Alive_p := Alive_p ∪ Succ ∪ {q} ;
                     Rcvd_p := Rcvd_p ∪ {q}
               end ;
      Compute a knot in G
end
```

图14-1　结的计算算法

可见，进程向它们自己发送消息，在许多健壮性算法中就是这样做的。在这里以及其余

部分，设操作“shout <**mes**>”代表

forall $q \in \mathbb{P}$ **do** 发送<**mes**> 到q

进程通过发出它们的标识（在消息<**name**，p>中），并等待接收L条消息，来构造有向图G。由于至少有L个正确进程，每个正确进程接收足够多的消息来完成这部分计算。在图G中p的后继是节点q，从该节点进程p接到消息<**name**，q>。

初始死进程即没有发送也没有接收任何消息，因此形成G中的一个孤立节点。正确进程有 443
L个后继，因此不是孤立的。结是一个没有输出边的强连通分量，至少包含两个节点。G中存在包含正确进程的结。因为每个正确进程出度为L，这个结的大小至少为L。因此，当$2L > N$时，只存在一个结，称它为K。最后，当正确进程p有L个后继时，至少会有一个后继属于K。这蕴含着K中的所有进程是p的后代。

因此，在算法的第二个阶段中，进程构造了G的一个导出子图，这个导出子图至少包含了它们的后代，并从它们认为是正确的每个进程，接收后继的集合。因为进程发送一条消息后，不会发生故障，在这一阶段，不会出现死锁。进程p等待接收q的消息，仅当在第一阶段中，某些进程已经接收到了表明q是正确的<**name**，q>消息。

一旦图14-1所示的算法终止，每个正确进程已经接收到它的每个后代的后继集合，并允许计算G中的惟一结。

2. 一致性和选举问题

当所有正确进程正确进程的结达成一致后，选择进程则是平凡的。K中具有最大标识的进程被选上。同时也容易取得一致性。每个进程同它的后继、输入(x)一起进行广播，计算K之后，进程对某个值进行判定，该值是K中输入集的一个函数（例如，更经常出现的一个值，在平局的情况下为0）。

结协议算法，一致算法以及选举算法要交换$O(N^2)$条消息，其中一条消息可能包含有L个进程名的一张表。更有效的选举算法已被提出。Itai等人[IKWZ90]给出了消息复杂度为$O(N(t+\log N))$的一个算法，并证明了$O(N(t+\log N))$是一个下界。Masuzawa等人[MNHT89]考虑了具有方向侦听的团上的问题，并提出了消息复杂度为$O(Nt)$的算法，这是一个最优算法。

选择正确进程作为领导人的任何选举算法也可以解一致性问题。领导人将它的输入进行广播，所有正确进程对其进行判定。因此，上述提到的上界，对于初始死进程的一致性问题同样成立。然而，领导人的可用性并不有助于解决一致性问题。在广播它的输入之前，领导人自身也可能损毁。此外，正如下一节中所表明的，损毁模型并不能解决选举问题。 444

14.3 确定可实现实例

到目前为止研究的一致性问题，要求每个进程对同一个值判定。本节研究的任务可解性，对进程之间的密切协同要求较少。在14.3.1小节，给出了一个实际问题的解决方法，即，用较小的命名空间，对进程集合进行重命名。在14.3.2小节，将之前给出的不可能性的结果扩展到更大一类的判定问题上。

用可能的输入集合X和输出集合D描述分布式任务，一个（可能是部分）映射

$$T: X^N \to \mathcal{P}(D^N)$$

映射T表示，如果向量$\vec{x} = (x_1, \ldots, x_N)$代表进程的输入，那么$T(\vec{x})$是算法的合法输出集

合，用判定向量 $\vec{d} = (d_1, \ldots, d_N)$ 表示。如果T是部分函数，那么并不是每个输入值的组合都是允许的。

定义14.10 *算法是任务T的t-损毁健壮解，如果它满足以下条件。*

（1）**终止性** 在每次t-损毁的公平执行中，所有正确进程进行判定。

（2）**一致性** 如果所有进程是正确的，判定向量 $\vec{d}$ 在 $T(\vec{x})$ 中。

一致性条件蕴含着，进程的子集进行判定的那些执行中，判定的部分向量总是可以扩展为 $T(\vec{x})$ 中的向量。集合D_T表示所有输出向量的集合，即T的值域。

（1）例子：一致性问题。一致性问题要求所有判定相等，即

$$D_{\text{cons}} = \{(0,0,\ldots,0),(1,1,\ldots,1)\}$$

（2）例子：选举问题。选举问题要求一个进程对1判定，其他进程对0判定，即

$$D_{\text{elec}} = \{(1,0,\ldots,0),(0,1,\ldots,0),\ldots,(0,0,\ldots,1)\}$$

（3）例子：近似一致性问题。在ε-近似一致性问题中，每个进程有一个真实的输入值，
445 并对真实的输出值进行判定。要求任意两个输出值之间的最大差至多为ε。

$$D_{\text{approx}} = \{(d_1,\ldots,d_N) : \max(d_i) - \min(d_i) < \varepsilon\}$$

（4）例子：重命名问题。在重命名问题中，每个进程具有不同标识，该标识取自一个任意大的域。每个进程必须对一个新名字进行判定。新名字的定义域为1，…，K，满足所有新名字不同。

$$D_{\text{rename}} = \{(d_1,\ldots,d_N) : i \neq j \Rightarrow d_i \neq d_j\}$$

在保持顺序（order-preserving）的重命名问题中，要求新名字保持旧名字的顺序，即，$x_i < x_j \Rightarrow d_i < d_j$。

14.3.1 可解问题：重命名

本小节给出Attiya *et al.*[ABND+90]提出的重命名算法。算法可忍受达到$t < N/2$次的损毁（t是算法中的参数）且在大小为$K = (N-t/2)(t+1)$的空间中命名。

1. t 的上界

首先表明，没有一个重命名算法能够忍受$N/2$次或更多的损毁。事实上，几乎所有损毁-健壮算法的故障数极限为$t < N/2$，以下的证明也适合于其他的问题。

定理14.11 *不存在$t \geq N/2$的重命名算法。*

证明。 如果$t \geq N/2$，可以形成两个不相交的大小为$N-t$的进程组S和T。由于t个进程可能发生故障，一个组必须能够对“自己的组”进行判定，即不与组外的进程进行交互（参见命题14.4）。因此，在一次执行中，组可以独立地完成判定。证明的关键是说明，这些判定是相互不一致的。我们继续证明重命名问题的例子。

由命题14.4，对于每一个初始配置γ，存在配置δ_S，满足S中的所有进程已经判定，且$\gamma \rightsquigarrow_S \delta_S$，对于$T$，类似的性质成立。在$N-t$个进程的组内的算法操作，定义了从$N-t$个初始标识组成的向量到$N-t$个新名字组成的向量的一个关系。因为初始的命名空间是无限的，新名字来自有
446 限域，那么存在不相交被映射到重叠的输出上的输入。即存在输入向量（长为$N-t$）$\vec{u}$ 和 $\vec{v}$，

对于所有i和j，满足$u_i \neq v_j$，但是存在相应的输出向量 $\vec{d}$ 和 $\vec{e}$，对于某些i和j，满足$d_i = e_j$。

现在，不正确的执行构造如下。初始配置γ在组S中的输入为 $\vec{u}$，在组T中的输入为 $\vec{v}$。观察可见，所有初始名字不同（在这两个组之外的初始名字可任意选择）。设σ_T是步骤的序列，通过这些步骤，组T由γ可达，在配置δ_T中，T中的进程已经对名字 $\vec{e}$ 进行判定。由命题14.1，这个序列在配置γ_S中仍然是可应用的，在这个配置中，S中的进程已经对名字 $\vec{d}$ 进行判定。在$\sigma_T(\gamma_S)$中，两个进程已对同一名字（因为$d_i = e_j$）进行判定，这表明算法不是一致的。 □

在以下情况下，假设$t < N/2$。

2. 重命名算法

在重命名算法中（图14-2所示的算法），进程p维持它已经看到的进程输入的集合V_p，初始时，V_p正好包含x_p。每次p接收的输入集合中有新的输入时，就对V_p进行扩展。一旦开始，且每次V_p被扩展后，p向所有近邻发送它的集合。由此，集合V_p仅在执行中增长，即V_p的后续值完全由所包含的输入来定序，此外V_p包含至多N个名字。因此，进程p向所有近邻发送它的集合至多N次，表明算法终止，且消息复杂度为$O(N^3)$。

```
var V_p    : set of identities ;
    c_p    : integer ;

begin V_p := {x_p} ; c_p := 0 ; shout ⟨set, V_p⟩ ;
      while true
      do begin receive ⟨set, V⟩ ;
               if V = V_p then
                 begin c_p := c_p + 1 ;
                       if c_p = N − t and y_p = b then
                         (* V_p is stable for the first time: decide *)
                         y_p := ( # V_p , rank(V_p, x_p) )
                 end
               else if V ⊆ V_p then
                 skip    (* Ignore "old" information *)
               else  (* new input; update V_p and restart counting *)
                 begin if V_p ⊂ V then c_p := 1 else c_p := 0 ;
                       V_p := V_p ∪ V ; shout ⟨set, V_p⟩
                 end
          end
end
```

图14-2 简单重命名算法

此外，进程p还对已经接收到对当前集合V_p的拷贝的次数进行计数（在变量c_p中）。初始时，c_p为0，每当接到包含V_p的一条消息时，c_p就增加。消息<**set**，V>的接收可能引起V_p增长，须重新设置c_p。如果V_p的新值与V相等（即如果V是旧V_p的严格超集），则将c_p设为1，否则为0。

称进程p达到**稳定集**V，如果当V_p的值是V时，c_p变成$N-t$。换句话说，p已经第$N-t$次接到V_p的当前值V。

引理14.12 稳定集是完全有序的，即如果q达到稳定集V_1和r达到稳定集V_2，那么$V_1 \subseteq V_2$，或者$V_2 \subseteq V_1$。

证明。假设q到达稳定集V_1，r到达稳定集V_2。这蕴含着q已经从$N-t$个进程接收到消息<**set**，V_1>，r已经从$N-t$个进程接收到消息<**set**，V_2>。因为$2(N-t) > N$，至少存在一个进程p，q和r

从它分别接收消息<set，V_1>和<set，V_2>。因此，V_1和V_2都是V_p中的值，蕴含着一个被包含在另一个中。□

引理14.13 *在每次公平t-损毁的执行中，每个正确进程至少到达一个稳定集合。*

*证明。*设p是正确进程，集合V_p只能被扩展，至多包含N个输入名。因此对于V_p可达最大值V_0。进程p向所有近邻发送这个值，每个正确进程接到消息<set，V_0>。这表明，最终每个正确进程持有V_0的超集。

然而，这个超集不是严格的，否则，一个正确进程就会向p发送V_0的严格超集。这与V_0的选择矛盾（因为最大集合总是被p保存）。因此，每个正确进程q在执行中至少有一次有值$V_q = V_0$，因此每个正确进程在执行中向p发送消息<set，V_0>。所有这些消息在执行中都被接收，且V_p的增加从不会超过V_0，它们都被计数，使得V_0在p中变成稳定的。□

447
~
448

一旦首次到达稳定集合V，进程p就对（s，r）进行判定，其中s是V的大小，r是x_p在V中的次序。从$N-t$个进程已经接收到稳定集合，因此这个稳定集合至少包含$N-t$个输入名字，表明$N-t \leqslant s \leqslant N$。在大小为$s$的集合中的次序满足$1 \leqslant r \leqslant s$。于是，可能的判定数为$K = \sum_{s=N-t}^{N} s$，它的值为$(N-t/2)(t+1)$。如果需要，也可用从数对到范围为1，...，$k$的整数的固定映射（参见习题14.5）。

定理14.14 *图14-2所示的算法解决了输出名空间大小为$K = (N-t/2)(t+1)$的重命名问题。*

*证明。*因为在任何公平t-损毁的执行中，每个正确进程到达一个稳定集合，每个正确进程对新名字进行判定。为了证明所有新名字都不同，考虑进程q和r分别可达的稳定集合V_1和V_2。如果集合大小不同，q和r的判定也不同，因为这个大小包含在判定中。如果集合大小相等，那么，由引理14.12，它们是相等的。因此q和r在集合中有不同的次序，再次表明，它们的判定是不同的。□

3. 讨论

观察可见，进程在对自己的名字进行判定之后，并不能使图14-2所示的算法终止。算法继续被执行，来“帮助”其他进程进行判定。Attiya[ABND+90]表明，这是必须的，因为算法必须处理这样一种情况，即，某些进程执行很慢，其他一些进程已经执行判定后，这些慢进程才执行它们的第一步。

根据命名空间的大小，这里提出的简单算法并不是最好的。Attiya *at al.*[ABND+90]给出了一个更复杂的算法，所赋名字的值域为1，...，$N+t$。下一小节蕴含着损毁健壮重命名问题，它的新名字空间大小的下界为$N+1$。

Attiya等人同时提出了保持次序的重命名算法。重命名的整数的域为1，...，$K = 2^t \cdot (N-t+1)-1$。同时证明了这是t-损毁、健壮、保持次序重命名算法的可能的最小命名空间。

14.3.2 扩展的不可能性结果

Moran and Wolfstahl[MW87]将一致性问题的不可能性结果推广到更一般的判定问题上。任务T的判定图是图$G_T = (V, E)$，其中$V = D_T$且

449

$$E = \{ (\vec{d}_1, \vec{d}_2) : \ \vec{d}_1 \text{和} \vec{d}_2 \text{只有一个分量不同} \}$$

如果G_T是连通图，则称任务T是连通的，否则称它是不连通的。Moran和Wolfstahl假设T的输入图（定义类似于判定图）是连通的，即，如同在引理14.6证明的中，可以通过逐一改

变进程的输入，在任意两个输入配置之间转移。此外，证明了非-平凡算法的不可能性结果，非-平凡性算法是这样的算法，它除了满足（1）终止性（2）一致性，还要满足（3）**非-平凡性**。对于每个 $\vec{d} \in D_T$，存在可达配置，在这个配置中，进程已经对 $\vec{d}$ 进行判定。

定理14.15 *对于不连通的任务T，不存在非-平凡、1-损毁-健壮判定算法。*

证明。用反证法。假设存在这样的算法A，从它可导出一致性算法A'，由定理14.8，这构成矛盾。为了简化证明过程，我们假设G_T包含两个连同分量，“0”和“1”。

算法A'首先模拟算法A，而不是对值d进行判定，一个进程向所有近邻发送消息<**vote**，d>，并等待接收$N-1$条投票消息。不会出现死锁，因为所有正确进程在A中判定，因此至少有$N-1$个进程向所有近邻发送投票消息。

接到消息之后，进程p持有D^N中向量的$N-1$个分量。用没有接到投票的进程的值对这个向量进行扩展，按照这种方式，整个向量在D_T中。（事实上，这个进程进行一致性判定，或者这个进程仍然只是可能进行一致性判定。）

观察可见，不同进程可能计算不同扩展，但是这些扩展属于G_T的同一个连通分量。接到$N-1$个投票的每个进程，对扩展向量所属的连通分量的每个名字进行判定。这仍然可以表明，A'是一致性算法。

终止性 上述已经证明，每个正确进程至少接到$N-1$次投票。

一致性 我们首先证明，存在向量 $\vec{d}_0 \in D_T$，满足每个正确进程得到 $\vec{d}_0$ 的$N-1$个分量。

情形1：*所有进程在A中都可找到一次判定。*设 $\vec{d}_0$ 是所达判定的向量，每个进程得到 $\vec{d}_0$ 的$N-1$个分量，尽管每个进程“丢失的”分量可能不同。

情形2：*除了一个进程r之外，所有进程在A中都可找到一次判定。*所有正确进程接到相同的$N-1$个判定，即除了r以外的所有那些进程。r可能损毁，但也可能由于r只是慢，因此r仍然 450
可能达到一个判定，即，存在向量 $\vec{d}_0 \in D_T$，对到目前为止的判定进行扩展。

由 $\vec{d}_0$ 的存在性，可得每个进程对向量 $\vec{d}_0$ 的连通分量进行判定。

非-平凡性 由A的非平性，可达分量0和分量中1的判定向量。按照A'的构造，两种判定都是可能的。

因此，A'是一个异步、确定性的、1-损毁-健壮的一致性算法。由定理14.8，可得不存在算法A。 □

讨论

非-平凡性的要求，规定D_T中的每一个判定向量是可达的，这是相当强的要求。人们可能会问，在这种意义上平凡的算法是否仍然值得考虑。作为一个例子，考虑重命名图14-2所示的算法，它的非-平凡性并不明显，即，具有不同名字的每个向量是可达的。甚至在这种情况为什么对非-平凡性感兴趣也不太清楚。

对定理14.15的证明的考察揭示出，在证明中可用较弱的非平凡性要求，即，判定向量至少在G_T的两个不同连通分量中是可达的。这种弱的非平凡性有时可从问题的陈述导出。

当有一个处理器出现故障时，Biran、Moran and Zaks[BMZ90]做了与可解和不可解的判定任务相关的一些基础工作。他们给出了可解判定任务的完整组合特性。

14.4 概率一致性算法

451 定理14.8的证明表明，每个异步一致性算法有无限次执行，在这些执行中，不发生判定。幸运的是，对于精心选择的算法，这样的执行足以少到概率为0，这使得算法在概率意义上非常实际，参见第9章。在本节中，我们提出两个概率一致性算法，一个用于损毁模型，另一个用于Byzantine模型。算法是由Bracha和Toueg[BT85]提出的。在两种情况下，首先假设了弹性（分别为$t < N/2$和$t < N/3$）的上界，然后给出了上界相匹的两个算法。

在这些概率一致性算法的正确性要求中，终止性要求被概率化，即用弱的收敛要求代替终止性要求。

收敛性 对于每个初始配置，

$$\lim_{k\to\infty} \mathbf{Pr}[\text{一个正确进程在}k\text{步之后，还未判定的概率}] = 0$$

在每次执行中，必须满足部分正确性（一致性）。所得概率算法就是Las Vegas算法[9.1.2小节]。

从给定初始配置开始的所有执行都以概率进行。为了使得概率具有意义，必须给出这些执行的概率分布。可以在进程中利用随机化做到这一点（如在第9章）。但这里定义了消息到达的概率分布。

通过假设公平调度（fair scheduling）来定义从给定初始配置开始执行的概率分布。这两个算法按轮操作。在一轮循环中，一个进程向所有近邻发送一条消息，并等待接收$N-t$条消息。定义$R(q, p, k)$为第k轮循环的事件，进程p接收前$N-t$条消息中q的（第k轮）消息。公平调度意味着

（1）$\exists \varepsilon > 0 \forall p,q,k : \mathbf{Pr}[R(p,q,k)] > \varepsilon$ 。

（2）对于所有k，以及不同进程p，q，r，事件$R(q, p, k)$和事件$R(q, r, k)$是独立的。

观察可见，当要求收敛性（以概率1终止）时，命题14.4对于概率算法也成立。因为能够以正概率到达可达配置。从每一个可达配置，一定可以到达一个判定的配置（尽管不必在每次执行中都达到）。

14.4.1 损毁-健壮一致协议

本节假设在损毁故障模型中，研究一致性问题。首先证明了关于弹性的一个上界$t < N/2$，随后给出了弹性为$t < N/2$的算法。

定理14.16 *不存在$t \geq N/2$的t-损毁-健壮一致协议。*

证明。如果存在这样的协议P，那么下述的三个声明成立。

声明14.17 *P有双价初始配置。*

证明。类似于引理14.6的证明，详细证明留给读者。 □

对于进程的子集S，称配置γ为S-双价的，如果仅执行S中的步，从配置γ可达0-判定配置和
452 1-判定配置。称配置γ为S-0-价的，如果仅执行S中的步，从配置γ可达0-判定配置但不能达到1-判定配置。类似地可定义S-1-价的配置。

将进程分为两组，S和T，大小分别为$\lfloor N/2 \rfloor$和$\lceil N/2 \rceil$。

声明14.18 *对于可达配置γ，γ或者是S-0-价的和T-0-价的，或者是S-1-价的和T-1-价的。*

证明。协议的最大弹性蕴含，S和T都可独立地到达判定状态。如果允许进行不同判定，那么通过组合调度可达到不一致的配置。 □

声明14.19 *P没有可达的双价配置。*

证明。设给定可达的双价配置γ，不妨假设γ是S-1-价的和T-1-价的（利用声明4.18）。然而，γ是双价的，于是从γ也可达0-判定配置δ_0（明确地以组间合作的方式）。在从γ到δ_0的配置序列中，存在两个后续配置γ_1和γ_0，其中γ_v即是S-v-价的又是T-v-价的。设p是引起从γ_1到γ_0的转移的进程。现在$p \in S$是不可能的，因为γ_1是S-1-价的，γ_0是S-0-价的；类似地，$p \in T$是不可能的。这是一个矛盾。 □

由声明14.17和声明14.19，可引起与协议P的存在性的矛盾。因此定理14.16得证。 □

Bracha和Toueg损毁-健壮一致算法 Bracha和Toueg提出的损毁-健壮一致算法[BT85]按照轮执行：在第k轮中，进程向所有进程（包括自己）发送消息，并等待接收第k轮的$N-t$条消息。等待这个数目的消息并不会引入死锁的可能性（参见习题14.10）。

在每一轮中，进程p向所有近邻发送一次对0或者1的投票，及一个权值。权值是上一轮中所接到的对的那个值的投票数（在第一轮中为1）。权值超过$N/2$的投票称为证据。尽管不同进 453
程在一轮中可能进行不同投票，然而在一轮中，不会存在针对不同值的证据，正如以下将要表明的。如果进程p在第k轮中接到一个证据，那么进程p在第$k+1$轮中对它的值投赞成票；否则进程p对接到的大部分投票投赞成票。如果在一轮中，收到多于t个证据，就进行判定。经过判定的进程退出主循环，并在下两轮中向所有近邻发送证据，以便使得其他进程能够判定。协议如图14-3所示的算法。

对于到达稍后轮的投票必须在适当的轮中处理；在算法中，进程将这个消息发向自己，以备以后处理，来对以加以模型化。观察可得，在任何一轮中，进程至多从每个进程接收一次投票，因此总数可达$N-t$次投票。因为向所有近邻发送投票的进程数可能多于$N-t$，进程可能会考虑向所有近邻发送投票的不同子集。我们随后显示算法的几个性质，它们一起蕴含了算法是一个概率损毁-健壮一致性协议（定理14.24）。

引理14.20 *在任何一轮中，不存在两个进程为不同值作证。*

证明。假设在第k轮中，对于$k > 1$，进程p为v作证，进程q为w作证。因为在第1轮中，没有进程作证。假设蕴含着，在第$k-1$轮中，进程p接到赞成v的投票数超过$N/2$；进程q接到赞成w的投票数超过$N/2$。票数加起来超过N。因此，p和q接到的投票的进程有些是重叠的，即存在进程r，向p发送v-投票，向q发送w-投票。这蕴含着，$v = w$。 □

引理14.21 *如果一个进程判定，那么至多两轮之后，所有正确进程对同一个值判定。*

证明。设k表示第一次发生判定的轮数，第k轮发生判定的进程为p，v是p的判定值。判定蕴含着在第k轮中有v-个证据；因此由引理14.20可得，不存在其他值的证据。因此在第k轮中，不存在不同判定。

在第k轮中，为v作证的证据多于t个（由p的判定而得），因此，在第k轮中，所有正确进程至少接到一个v-证据。因此，在第$k+1$轮投票的所有进程赞成v（同时p仍然在第$k+1$轮向所有近邻发送一张票）。这蕴含着，如果在第$k+1$轮的确进行判定，那么这个判定就是对v的判定。

在第$k+1$轮中，仅有v-投票被提交，因此在第$k+2$轮投票的所有进程在那一轮为v作证（进 454
程p也如此）。因此，在第$k+2$轮中，在此之前各轮中没有判定的所有正确进程接到$N-t$条v-证据，并对v判定。 □

引理14.22 $\lim_{k\to\infty} \mathbf{Pr}[\text{在} < k \text{的轮中，未进行判定}] = 0$。

证明。设S是$N-t$个正确进程的集合（存在这样的集合），假设直到第k_0轮，才会发生判定。

公平调度假设蕴含，对于某些$\rho > 0$，在任何一轮中，S中的每个进程恰好接到S中$N-t$个进程投票的概率至少为ρ。这种情况至少以概率$\psi = \rho^3$发生在三个后续轮中，k_0，k_0+1和k_0+2。

```
var value_p          : (0, 1)    init x_p   (* p's vote *)
    round_p          : integer   init 0     (* Round number *)
    weight_p         : integer   init 1     (* Weight of p's vote *)
    msgs_p[0..1]     : integer   init 0     (* Count received votes *)
    witness_p[0..1]  : integer   init 0     (* Count received witnesses *)

begin
  while y_p = b do
    begin msgs_p[0], msgs_p[1], witness_p[0], witness_p[1] :=
                                          0, 0, 0, 0 ;   (* Reset counts *)
          shout ⟨vote, round_p, value_p, weight_p⟩ ;
          while msgs_p[0] + msgs_p[1] < N − t do
                begin receive ⟨vote, r, v, w⟩ ;
                      if r > round_p then      (* Future round ... *)
                        send ⟨vote, r, v, w⟩ to p   (* ... process later *)
                      else if r = round_p then
                        begin msgs_p[v] := msgs_p[v] + 1 ;
                              if w > N/2 then     (* Witness *)
                                witness_p[v] := witness_p[v] + 1
                        end
                      else (* r < round_p, ignore *) skip
                end ;
          (* Choose new value: vote and weight in next round *)
          if witness_p[0] > 0 then value_p := 0
          else if witness_p[1] > 0 then value_p := 1
          else if msgs_p[0] > msgs_p[1] then value_p := 0
          else value_p := 1 ;
          weight_p := msgs_p[value_p] ;
          (* Decide if more than t witnesses *)
          if witness_p[value_p] > t then y_p := value_p ;
          round_p := round_p + 1
    end ;
  (* Help other processes decide *)
  shout ⟨vote, round_p, value_p, N − t⟩ ;
  shout ⟨vote, round_p + 1, value_p, N − t⟩
end
```

图14-3　损毁-健壮一致算法

如果发生这种情形，S中的进程接收第k_0轮中的相同投票，因此在第k_0轮中选择同一个值v_0。S中的所有进程在第k_0+1轮投票赞成v_0，蕴含着S中的每个进程在第k_0+1轮接到赞成v_0的$N-t$次投票。这意味着，S中的进程在第k+2轮为v_0作证。因此它们在第k_0+2轮都接到为v_0作证的$N-t > t$条证据，并在那一轮中对v_0进行判定。由此可得，

Pr[S中的进程在第k+2轮还未进行判定的概率] $\leqslant \psi \times$ **Pr**[S中的进程在第k轮之前还未进行判定的概率]

引理得证。　□

引理14.23　*如果所有进程初始算法时输入为v，那么在第2轮后，所有进程对v判定。*

*证明。*在第1轮中，所有进程只收到对v的赞成投票，因此所有进程在第2轮中为v作证。

这蕴含着，它们在那一轮中对v进行判定。 □

定理14.24 *对于$t < N/2$，图14-3所示的算法是概率的、t-损毁-健壮的一致性协议。*

*证明。*引理14.22证明了收敛性。引理14.21证明了一致性。引理14.23蕴含了非-平凡性。 □

在习题14.11中，进一步分析了判定对于输入值的相关性。

14.4.2 Byzantine-健壮一致性协议

Byzantine故障模型比损毁模型更恶意，因为Byzantine进程可能会执行任意的状态转移，并且可能发送与算法不一致的消息。我们用$\gamma \rightsquigarrow \delta$（或$\gamma \rightsquigarrow_S \delta$）表示，存在*正确步*的序列，即协议的转移（在$S$的进程中），引导系统从配置$\gamma$到$\delta$。类似地，如果存在正确步的序列，从初始配置通到$\gamma$，则称$\gamma$是可达的。Byzantine模型的恶意蕴含着，它有比损毁模型更低的弹性最大值。 455 ~ 456

定理14.25 *不存在$t \geqslant N/3$的t-Byzantine-健壮一致性协议。*

*证明。*反证法。假设存在这样的协议。对于这样协议的双价初始配置的存在性留给读者证明（如往常一样，利用非-平凡性）。

协议的高弹性蕴含着，可以这样选择进程的两个集合S和T，满足$|S| \geqslant N-t$，$|T| \geqslant N-t$，且$|S \cap T| \leqslant t$。S和T足够大可以独立存在，但是它们的交集可能是完全恶意的。这一点可用来证明不存在可达的双价配置。

声明14.26 *可达配置γ或者是S-0-价的和T-0-价的，或者是S-1-价的和T-1-价的。*

*证明。*由于γ可由正确步序列达到，选择t个故障进程集合的所有可能性仍然悬而未定。相反，假设S和T可达不同判定，即$\gamma \rightsquigarrow_S \delta_{\bar{v}}$和$\gamma \rightsquigarrow_T \delta_{\bar{v}}$，其中$\delta_v$（$\delta_{\bar{v}}$）是$S$中（$T$中）的所有进程已经对$v$（$\bar{v}$）进行判定的配置。通过假设在$S \cap T$中的进程是恶意的，并组合调度如下，就可以达到不一致的状态。从配置γ开始，$S \cap T$中的进程与S中的其他进程协同，就如同在一个导致S中的v-判定的序列中一样。当S中的进程进行这个判定时，恶意进程将它们的状态恢复到如同在γ中的一样，随后与T中的进程协同，就如同在一个导致T中的$\bar{v}$判定的序列中一样。这导致一个配置，其中的正确进程已进行不同判定，与一致性要求矛盾。 □

声明14.27 *不存在可达的双价配置。*

*证明。*设给定可达的双价配置γ。不妨假设。γ是S-1-价的和T-1-价的（声明14.26）。然而，γ是双价的，因此0-判定的配置δ_0也是由γ可达的（显然是以S和T之间的协同），在从γ到δ_0的配置序列中，存在两个后续配置γ_1和γ_0，其中γ_v是S-v-价的和T-v-价的。设p是引起从γ_1到γ_0转移的进程。现在，$p \in S$是不可能的，因为γ_1是S-1-价的，γ_0是S-0-价的。类似地，$p \in T$是不可能的。这是矛盾。 □ 457

最后声明与双价初始配置的存在性矛盾。因此定理14.25得证。 □

Bracha和Toueg的Byzantine-健壮一致算法 对于$t < N/3$，存在t-Byzantine-健壮的一致性协议。通信系统允许进程决定所接收的消息由哪个进程来发送是十分必要的。如果Byzantine进程p可以向正确进程r发送消息，并成功地伪装成如像r接收这条*来自q*的消息（参见图14-4），那么问题就变得不可解。确实，进程p能够模拟足够多的正确进程，来实施在进程r中的不正确判定。

如同损毁-健壮协议，Byzantine-健壮协议（图14-5所示的算法）按照轮执行。在每一轮中，

每个进程都能提交投票，当有足够多的进程投票同一个值时，就进行判定。低弹性（$t < N/3$）消除了区别证据和非证据的必要性，一旦接到多于（$N+t$）/2个对同一个值的投票，就进行判定。

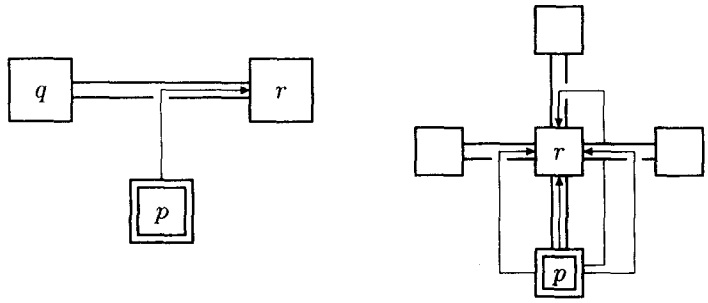


图14-4　模拟其他进程的Byzantine进程

故障模型的恶意性使得引入投票验证机制的成为必要。这是协议的关键所在。没有这样一个机制，Byzantine进程可能向各个正确进程发送不同的投票，在正确进程之间干扰投票过程。而在损毁模型中，这样的恶意行为是不可能的。验证机制保证，尽管Byzantine进程r可以向正确进程p和q发送不同的投票，它不能愚弄进程p和q，使它们接受对r的不同投票（在某些轮中）。

验证机制基于对消息的响应。进程向所有近邻发送它的投票（如*initial*，**in**），在某些轮中，每个进程一旦接收到对某些进程的第一次投票，就对投票做出响应（如*echo*，**ec**）。如果接到多于（$N+t$）/2个对它已经接收到的消息的响应，进程就将接受投票。验证机制保持正确
458 进程之间通信的（部分）正确性（引理14.28），正确进程从不接收对同一进程的不同投票（引理14.29），也不会引起死锁（引理14.30）。

我们称进程p在第k轮中接受一个对进程r的v-投票，如果p一旦接到投票消息<**vote**，**ec**，r，v，k>，使得$msgs_p[v]$增加。算法保证仅当接到$N-t$个投票后，p经过第k轮，同时保证p在每一轮中至多接收对每个进程的一次投票。

引理14.28　*如果正确进程p在第k轮接受对于正确进程r的投票v，那么r已经在第k轮中对v投票。*

证明。进程p一旦从超过（$N+t$）/2个不同进程接收消息<**vote**，**ec**，r，v，k>，就接收投票，至少有一个正确进程s已经向p发送这样的消息。进程s一旦接收到来自r的消息<**vote**，**in**，r，v，k>，就向p发送响应，由于r是正确的，这蕴含着r在第k轮中向v投票。　□

引理14.29　*如果正确进程p和进程q在第k轮接受对于进程r的一张投票，那么它们接收的是同一投票。*

证明。假设在第k轮中，进程p接收对于r的v-投票，进程q接收一次w-投票。因此，进程p已接收超过（$N+t$）/2个进程的消息<**vote**，**ec**，r，v，k>，进程q已从超过（$N+t$）/2个进程接收消息<**vote**，**ec**，r，w，k>。因为只有N个进程，超过t个的进程已经向p发送消息<**vote**，**ec**，r，v，k>并向q发送消息<**vote**，**ec**，r，w，k>。这蕴含着至少有一个正确进程这样做，因此$v = w$。　□

引理14.30　*如果所有正确进程都开始第k轮，那么所有正确进程在那一轮中接受足够多的投票来完成那一轮。*

证明。开始$value_r = v$的第k轮的正确进程r，在那一轮中向所有近邻发送初始投票，所有

正确进程都对此发出响应。因此，对于正确进程p和r，至少有$N-t$个进程向p发送消息<**vote**, **ec**, r, v, k >，并允许p在第k轮中接受对r的v-投票，除非$N-t$个其他投票之前已被接受。由此可得，进程p在这一轮中，接受$N-t$个投票。 □

```
var value_p          : (0, 1)    init x_p ;
    round_p          : integer   init 0 ;
    msgs_p[0..1]     : integer   init 0 ;
    echos_p[ℙ, 0..1] : integer   init 0 ;

while true do
    begin forall v, q do begin msgs_p[v] := 0 ; echos_p[q, v] := 0 end ;
        shout ⟨ vote, in, p, value_p, round_p ⟩ ;
        (* Now accept N − t votes for the current round *)
        while msgs_p[0] + msgs_p[1] < N − t do
            begin receive ⟨ vote, t, r, v, rn ⟩ from q ;
                if ⟨ vote, t, r, *, rn ⟩ has been received from q already
                  then skip    (* q repeats, must be Byzantine *)
                else if t = in and q ≠ r
                  then skip    (* q lies, must be Byzantine *)
                else if rn > round_p
                  then (* Process message in later round *)
                       send ⟨ vote, t, r, v, rn ⟩ to p
                else (* Process or echo vote message *)
                  case t of
                      in : shout ⟨ vote, ec, r, v, rn ⟩
                      ec : if rn = round_p then
                               begin echos_p[r, v] := echos_p[r, v] + 1 ;
                                     if echos_p[r, v] = ⌊(N + t)/2⌋ + 1
                                       then msgs_p[v] := msgs_p[v] + 1
                               end
                           else skip (* Old message *)
                  esac
            end ;
        (* Choose value for next round *)
        if msgs_p[0] > msgs_p[1] then value_p := 0 else value_p := 1 ;
        if msgs_p[value_p] > (N + t)/2 then y_p := value_p ;
        round_p := round_p + 1
    end
```

图14-5 Byzantine-健壮一致算法

由此可得协议的正确性证明，证明过程类似于损毁-健壮协议的正确性证明。

引理14.31 *如果一个正确进程在第k轮对v判定，那么所有正确进程在第k轮和以后的所有轮中选择v。*

*证明。*设S是至少 $(N+t)/2$个进程组成的集合，p在第k轮中接受对它的v-投票。正确进程q在第k轮中，接受$N-t$次投票，其中包含至少$|S|-t$ >$(N-t)/2$次对于S中的进程的投票。由引理14.29，q接受超过 $(N-t)/2$次的v-投票，这蕴含着q在第k轮中选择v。

为了证明所有正确进程在以后的轮中都选择v，假设所有正确进程在某些轮l中选择v。因此，所有正确进程在第l +1轮中对v投票。在第l +1轮中，每个正确进程接受$N-t$次投票，包含对于正确进程的 $(N-t)/2$次投票。由引理14.28，正确进程接受至少 $(N-t)/2$次v-投票，因此在第l +1轮中再次选择v。 □

459 ~ 460

引理14.32　$\lim_{k\to\infty}$**Pr**[正确进程p在第k轮之前还未进行判定的概率] = 0。

证明。设S是至少$N-t$个正确进程组成的集合，假设进程p在第k轮之前还未进行判定。以概率$\psi > 0$，S中的进程在第k轮接受对于$N-t$个进程组成的同一集合的投票，并且在第$k+1$轮，只接受对S中的进程的投票。如果发生这种情况，S中的进程在第$k+1$轮平等地投票，并在第$k+1$轮进行判定。由此可得

Pr[在第$k+2$轮之前正确进程还未进行判定的概率] $\leqslant \psi \times$ **Pr**[正确进程在第k轮之前还未进行判定的概率]

引理得证。 □

引理14.33　如果所有正确进程以输入v开始算法，那么最终要进行对v的判定。

证明。如同引理14.31的证明，可以证明，所有正确进程在每一轮中会再次选择v。 □

引理14.34　对于$t < N/3$，图14-5所示的算法是概率的、t-Byzantine-健壮一致性协议。

证明。引理14.32证明了收敛性。引理14.31证明了一致性。引理14.33蕴含了非-平凡性。□

在习题14.12中，进一步分析了判定对于输入值的相关性。算法14-5被描述为无限循环以便于陈述。我们最终描述如何修改算法，使得在每个判定进程中能够终止。在第k轮对v判定之后，进程p退出循环，并向所有近邻发送“多张”投票<**vote**, **in**, p, k^+, v>，以及响应消息<**vote**, **ec**, *, k^+, v>。这些消息被解释成对于k以后的所有轮的初始和响应投票。的确，进程p在以后所有轮中对v投票，所有正确进程也将会这样做（由引理14.31可得）。因此，多消息
461 就是算法继续执行时，进程p要发送的那些消息。只有一种可能的例外，就是对于恶意初始投票的响应。

14.5 弱终止性

在本节，研究了异步Byzantine广播问题。广播的目标是，产生一个出现在进程g中的值，称为将军（general），对于所有进程，该值是可知的。形式地，如果将军是正确的，则把判定值作为将军的输入，使得一致协议的非平凡性要求被加强。

相关性　如果将军是正确的，所有正确进程对它的输入进行判定。

根据这种规定，将军变成了一个独立的故障点，这蕴含着这个问题是不可解的。如以下定理所述。

定理14.35　不存在满足收敛性、一致性和相关性的1-Byzantine健壮算法，即使仅当将军已经发送至少一条消息，对收敛性有要求时。

证明。我们考虑两种情形。第一种情形中，假设将军是Byzantine的。这种情形定义了可达的配置γ。因此，在第二种情形，产生矛盾。

（1）假设将军是Byzantine的，并向进程p_0发送初始化“0”的广播的消息，向进程p_1发送初始化“1”的广播的消息。然后，将军停止。我们称得到的配置为γ。

根据收敛性，即使将军损毁，也可达判定的配置。设$S = \mathbb{P}\backslash\{g\}$，并假设$\gamma \rightsquigarrow_S \delta_0$，其中$\delta_0$是0-判定的。

（2）对于第二种情形，假设将军是正确的，且输入为1，并向进程p_0和进程p_1发送初始化“1”的广播的消息，在此之后，它的消息被延迟很长一段时间。现在假设p_0是Byzantine的，在接到消息之后，根据γ中的状态，改变其状态，即假装已经接到来自将军的0-消息。由于$\gamma \rightsquigarrow_S \delta_0$，不用与将军交互，就可以达到0-判定，这是不允许的，因为将军是正确的，且输入为1。□

不可能性是由这样一种情况所导致的，即，没有提供足够多的输入信息（如在第二种情形中所使用的），将军就初始化广播，并停止（第一种情形）。现在已证明，如果只在将军是正确的情形下要求终止性，则一种（确定性的）解是可能的。 462

定义14.36 *t*-Byzantine-健壮广播算法是满足以下三个要求的算法：

（1）**弱终止性** 所有正确进程判定，或者没有正确进程判定。如果将军是正确的，所有正确进程判定。

（2）**一致性** 如果正确进程判定，那么它们对同一个值进行判定。

（3）**相关性** 如果将军是正确的，所有正确进程对它的输入进行判定。

异步Byzantine广播算法的弹性能被证明有界$t < N/3$，证明过程类似于定理14.25中的证明。图14-6所示的算法给出了Bracha和Toueg广播算法[BT85]，其中使用了三种类型的投票消息：*initial*消息（类型**in**）、*echo*消息（类型**ec**）和*ready*消息（类型**re**）。每个进程对每种类型和值，以及接收了多少消息进行计数，对于每个进程所接收的最多一条消息计数。

将军向所有近邻发送一个初始投票，初始化广播。进程一旦接到来自将军的初始投票，就向所有近邻发送包含同一个值的响应投票。当接到多于$(N+t)/2$条带值v的响应消息时，就向所有近邻发送一条就绪消息。所要求的响应数要足够大，以保证没有正确进程发送不同值的就绪消息（引理14.37）。接到多于t条同一值的就绪消息（蕴含着至少有一个正确进程已经发送了这个消息），也会触发向所有近邻发送就绪消息。接到多于$2t$个同一值的就绪消息（蕴含着多于t个正确进程已经发送了这个消息），则会引起对那个值的判定。在图14-6所示的算法中，没有采取任何措施，阻止一个正确进程发出两次就绪消息，但是该消息被正确进程忽略。

```
var msgs_p[(in, ec, re), 0..1]    : integer    init 0 ;

For the general only:
    shout ⟨vote, in, x_p⟩

For all processes:
while y_p = b do
    begin receive ⟨vote, t, v⟩ from q;
        if a ⟨vote, t, *⟩ message has been received from q already
          then skip    (* q repeats, ignore *)
        else if t = in and q ≠ g
          then skip    (* q mimics g, must be Byzantine *)
        else begin msgs_p[t, v] := msgs_p[t, v] + 1 ;
                case t of
                    in : if msgs_p[in, v] = 1
                           then shout ⟨vote, ec, v⟩
                    ec : if msgs_p[ec, v] = ⌈(N + t)/2⌉ + 1
                           then shout ⟨vote, re, v⟩
                    re : if msgs_p[re, v] = t + 1
                           then shout ⟨vote, re, v⟩ ;
                         if msgs_p[re, v] = 2t + 1
                           then y_p := v
                esac
            end
    end
```

图14-6 Byzantine-健壮广播算法

引理14.37　不存在两个正确进程发送不同值的就绪消息。

证明。正确进程至多接受一条初始消息（从将军处），因此为至多一个值发送响应。

设p是发送值v的就绪消息的第一个正确进程，q是发送值w的就绪消息的第一个正确进程。尽管一旦接收足够多的就绪消息，就发送一条就绪消息。但这不是发送就绪消息的第一个正确进程的情形。之所以这样，是由于在发送一条就需消息之前，一定已经接收了$t+1$条就绪消息。这蕴含着来自至少一个正确进程的就绪消息已被接收。因此，进程p已经接收来自超过$(N+t)/2$个进程的v-响应消息，进程q已经接收来自超过$(N+t)/2$个进程的w-响应消息。

因为只有N个进程，且$t < N/3$，存在多于t个的进程，包含至少一个正确进程r，从这个进程，进程p已经接收一条v-响应消息，进程q已经接收一条w-响应消息。因为r是正确的，蕴含$v = w$。□

463 ~ 464

引理14.38　如果一个正确进程判定，那么所有正确进程判定，且对同一个值判定。

证明。必须接收多于$2t$条的v的就绪消息，才能对v判定，其中包括来自正确进程的超过t条的就绪消息。引理14.37蕴含可以进行判定。

假设正确进程p对v判定，p已经接收$2t$条就绪消息，包括来自正确进程的多于t条的就绪消息。向p发送就绪消息的正确进程把这条消息发向所有进程，这蕴含，所有正确进程接到多于t条的就绪消息，这反过来蕴含所有正确进程发送就绪消息，因此每个正确进程最终接收$N-t > 2t$条就绪消息，并进行判定。□

引理14.39　如果将军是正确的，所有正确进程对它的输入进行判定。

证明。如果将军是正确的，也没有发送带有与输入不同值的初始消息。因此，没有正确进程会发送带有与将军的输入不同值的响应消息，这蕴含着，至多t个进程发送这些坏响应消息。这些坏响应消息的数量，不足以使正确进程发送不同于将军输入的就绪消息。这蕴含着没有一个正确进程发送坏响应消息或者进行不正确的判定。

如果将军是正确的，它将带有它的输入的初始投票发送到所有正确进程中，而所有正确进程向所有近邻发送带有该值的响应。从而，所有正确进程将接收至少$N-t > (N+t)/2$条正确响应消息，并向所有近邻发送带有正确值的就绪消息。因此，所有正确进程接收至少$N-t > 2t$条正确就绪消息，并正确判定。□

465

引理14.40　对于$t < N/3$，图14-6所示的算法是异步、t-Byzantine-健壮的广播算法。

证明。由引理14.39和14.38可得弱终止性。由引理14.38可得一致性。由引理14.39可得相关性。□

习题

14.1节

14.1　对于定义14.3中关于一致性问题的三个要求（终止性、一致性和非-平凡性），忽略其中的任何一个，可得到非常简单的解。给出三种简单解证明这一点。

14.2　在引理14.6的证明中，假设对N个进程2^N个位的每一种赋值都会产生一种可能的输入配置。

针对下列每一个对输入值的限制条件，给出确定性的、1-损毁健壮一致性协议。

（1）在每个初始配置中，输入的奇偶性为偶数（即，存在输入为1的偶数个进程）。

（2）存在两个已知的进程r_1和r_2，每个初始配置满足$x_{r_1} = x_{r_2}$。

(3) 在每个初始配置中，至少存在⌈(N/2)+1⌉个进程具有相同输入值。

14.2节

14.3 证明对于$t \geqslant N/2$，不存在t-初始死进程-健壮选举算法。

14.3节

14.4 证明对于∈-近似一致性问题，不存在能容忍$t \geqslant N/2$个损毁的算法。

14.5 给出从以下集合

$$\{(s, r): N-t \leqslant s \leqslant N \text{且} 1 \leqslant r \leqslant s\}$$

到值域为整数域[1, ..., K]上的一个双射函数。

14.6 (项目) 图14-2所示的算法是非平凡的吗?

14.7 修改定理14.15的证明，使其适用于这样一种情形，即，G_T由k个连通分量组成。 466

14.8 在这个练习中，我们考虑[k, l]-选择问题，它是通常选举问题的一般化。问题要求所有正确进程对0("被击败")或1("被选上")进行判定，并且判定1的进程数介于k和l(包括边界)之间。

(1) [k, l]-选举问题有什么用处?

(2) 对于[k, k]-选举问题，用示例证明不存在确定、1-损毁健壮算法(如果$0 < k < N$)。

(3) 对于[k, k+2t]-选举问题，给出一个确定、t-损毁健壮算法。

14.4节

14.9 收敛性要求蕴含期望执行步数有界吗?

本节中所有算法的期望执行步数有界吗?

14.10 证明，如果所有正确进程开始执行损毁-健壮一致性算法的第k轮(图14.3所示的算法)，那么所有正确进程也将完成第k轮的执行。

14.11

(1) 证明，如果多于$(N+t)/2$个进程开始执行损毁-健壮一致性算法(图14-3所示的算法)，且算法输入为v，那么在三轮之后，对v进行判定。

(2) 证明，如果多于$(N-t)/2$个进程开始执行算法，且算法输入为v，那么对v的判定是可能的。

(3) 如果只有$(N-t)/2$个进程开始执行算法，且算法输入为v，那么可能对v进行判定吗?

(4) 算法的双价输入配置是什么?

14.12

(1) 证明，如果多于$(N+t)/2$个正确进程开始执行算法14-5，且算法输入为v，那么v判定最终被执行。

(2) 证明，如果多于$(N+t)/2$个正确进程开始执行算法14-5，且算法输入为v，$t < N/5$，那么在两轮之后，执行对v的判定。 467

14.5节

14.13 证明不存在$t \geqslant N/3$的异步、t-Byzantine-健壮广播算法。

14.14 证明在图14-6所示的算法的执行中，正确进程至多发送$N(3N+1)$条消息。 468

第15章 同步系统中的容错

前一章研究了完全异步系统中可实现的容错程度。尽管能达到合理的健壮性，但在实际中，由于使用计时器和消息传输时间有界，可靠系统总是同步的。在这些系统中，可以达到更高的健壮性，而算法更简单，在大多数情况下，算法保证了响应时间具有上界。

系统中的同步使故障进程不可能通过不发送信息而干扰正确进程。如果一个进程在预期的时刻没有接收到消息，就会用默认值，这样，发送者就会被怀疑发生了故障。因此，损毁进程立即被检测出来，不会在同步系统中造成任何难题。本章我们集中研究Byzantine故障。

在15.1节，研究在同步网络中进行广播的问题。我们提出具有最优弹性的两个算法，以及弹性的一个上界（$t < N/3$）。假设所有进程知道广播何时开始，算法是确定的，并可达到一致性。因为在异步系统中一致性不是确定可达的（定理14.8）。由此可得，在出现故障时（即使是一个损毁故障），同步系统的确展示了比异步系统更强的计算能力。

在同步系统中，因为损毁以及不能发送信息会被检测出来（因此是“无害的”），Byzantine进程只能通过发送错误信息扰乱计算，这些错误信息或者是有关其自身状态的，或者是由于不正确地转发信息所致。在15.2节中，通过实例表明，提供信息鉴别方法可以增强同步系统的健壮性。有了这些机制，一个恶意进程要对从其他进程接收的信息“撒谎”是不可能的。但是仍有可能发送与进程自身状态不一致的信息。同时表明，在实际中利用密码技术，可以实现鉴别方法。

15.1节和15.2节的算法假设了一种理想的同步系统模型，在这种系统中计算按照脉冲（也称为轮）进行，参见第12章。同步系统的弹性基本上高于异步系统的弹性，这表明在异步模型中，任何一种1-损毁-健壮确定的脉冲模型是不可能实现的。（在可靠网络中可能实现这种模型，称为同步器）。

然而，在异步有限延迟网络（12.1.3小节）中实现脉冲模型是可能的，其中进程拥有时钟，并且已知消息延迟的上界。即使时钟漂移，并且多达三分之一的进程可能恶意地产生故障，实现也是可能的。实现中最困难的部分是使进程时钟可靠地同步，这个问题将在15.3节中讨论。

15.1 同步判定协议

在本节，我们给出同步（脉冲）网络中Byzantine-健壮广播算法。首先简要回顾一下12.1.1节定义的脉冲网络模型。在同步网络中，进程按照被标号为1、2、3等的脉冲来运行。只要进程的局部算法没有终止，每个进程都可执行无限多次的脉冲。用进程的初始状态描述初始配置γ_0，以及第i个脉冲（用γ_i表示）后的配置。在脉冲i，每个进程首先发送有限的消息集，该消息集取决于它在γ_{i-1}中的状态。随后，每个进程在这个脉冲中，接收发送给它的所有消息，并由以前的状态和本次脉冲中接收到的消息集，计算新的状态。

脉冲模型是同步计算的理想模型。同步性反映在

（1）进程中，明显地同时出现状态转移；且

470 （2）保证在那个脉冲的状态转移之前，接收到一个脉冲中的消息。

这些理想假设可以被弱化成更符合现实的假设。即（1）硬件时钟的可用性，（2）消息传输时间有上界。*异步有限延迟网络*可以有效地实现脉冲模型（参见12.1.3节）。正如在第12章中表明的那样，状态转移的同时性仅仅是表面上的。在模型的实现中，状态转移可能发生在不同时间，只要保证所有消息及时接收。此外，实现允许进程执行无限次的脉冲。后一要求不包括第12章的实现在容错中的应用，因为它们都遭受死锁，其中的大多数甚至会在丢失一条消息的情形下，即发生死锁。如前所述，在15.3节中将讨论脉冲模型健壮性的实现。

因为脉冲模型保证了在同一个脉冲中的传输消息，进程能够确定一个近邻没有向它发送消息。这个特性在异步系统中是没有的，并且这个特性可用来解决一致性问题，甚至可用来解决在同步系统中可能会出现的可靠广播的问题，正如我们很快就会看到。

在Byzantine-广播问题中，给定一个独特的称为*将军*（general）的进程g，和取自集合V（通常为{0,1}）的输入x_g。不同于将军的进程称为*中尉*（lieutenant）。必须满足以下三个要求。

（1）**终止性**　每个正确进程p对值$y_p \in V$进行判定。

（2）**一致性**　所有正确进程对同一个值判定。

（3）**相关性**　如果将军是正确的，所有正确进程对x_g判定。

此外，可能还要求**同时性**，即所有正确进程在同一个脉冲中判定。本节和下一节讨论的所有算法满足同时性。也可参见15.2.6节。

15.1.1 弹性界限

同步网络对于Byzantine故障的弹性与异步网络（定理14.25）的情形一样，即弹性上界为$t < N/3$。Pease、Shostak和Lamport[PSL80]通过给出算法在$N/3$或更多Byzantine进程中的几种情形，首先证明了这个上界。与定理14.25证明中所用的情形不同，这里正确进程接收*矛盾*信息，
471 允许某些进程是故障进程。然而，不可能决定哪个进程是不可靠的，并且不正确进程能够进行不正确判定。

定理15.1　*不存在$t \geq N/3$的t-Byzantine-健壮广播协议。*

*证明。*如同前面的证明，$N/3$或更高的弹性，允许把进程划分为三组（S，T和U），其中每一组都可能整个发生故障。包含将军的那组称为S。考虑三种情形，推导出矛盾，如图15-1所示，故障组用双方框表示。

情形0中，将军广播值0，组U中的进程是故障进程。情形1中，将军广播值1，组T中的进程是故障进程。在情形0的第i个脉冲，组U中的进程只向组T中的进程发送已经在情形1中发送过的那些消息（按照协议）（即，作为对在情形1的第$i-1$个脉冲中所接收的消息的响应的那些消息）。对于S中的进程，它们发送由协议所定向的消息。S和T中的进程在所有脉冲中发送正确消息。在这种情形下，只有组U中的进程发送不正确的消息，协议的规范规定，所有正确进程，包括组T都对0进行判定。

情形1可类似地定义。但是T中的进程是故障进程，并发送它们已在情形0中发送过的消息。在这种情形下，U中的进程对1进行判定。

最后，考虑情形2，S中的进程是故障进程，表现如下。对于T中的进程，它们发送情形0中的消息，而对于U中的进程，它们发送情形1中的消息。用归纳法对脉冲数证明可得，T所发送给U（或由U发送给T）的消息就是那些在情形0（或情形1）中所发送的消息。因此，对

于T中的进程，情形2与情形0没有分别。对于U中的进程，情形2与情形1没有分别。由此可得，T中的进程对0判定，U中的进程对1判定，这是矛盾。 □

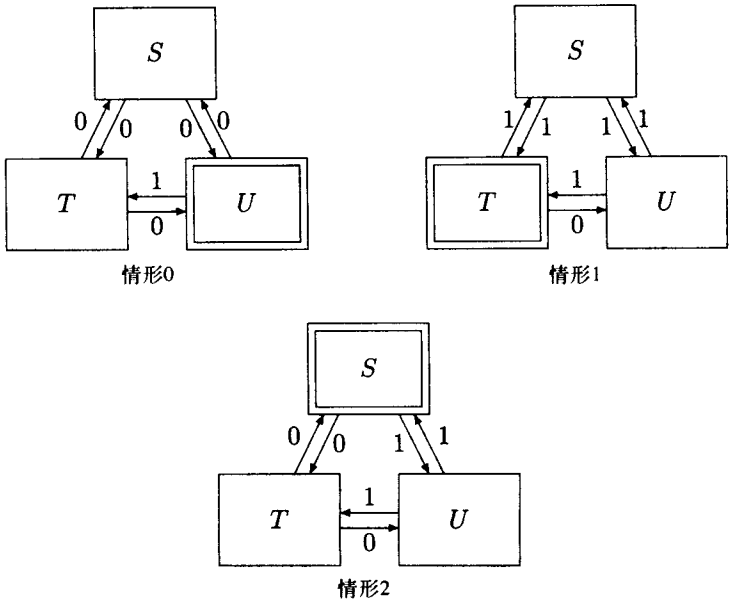


图15-1 定理15.1证明的情形

在证明中，即使Byzantine进程只接到0-情形中的消息，它们也可以发送1-情形中的消息。即进程不仅能对自己的状态“撒谎”，也能对它们接收到的消息“撒谎”。利用鉴别机制能够消除这种可能性，正如在15.2节中将要讨论的。这会导致弹性为$N-1$的结果。

15.1.2 Byzantine广播算法

本小节中，将证明前面小节中所证明的弹性上界。而且与异步网络的情况相反，利用确定算法就可达到最大弹性。我们给出Pease *et al.*[PSL80]提出的递归算法。该算法能容忍t个Byzantine故障，其中$t < N/3$。弹性是算法的参数。

图15-2所示的算法表示广播算法*Broadcast* $(N, 0)$。该算法可以容忍没有故障 $(t = 0)$，并且如果没有出现故障，所有进程就对脉冲1中将军输入进行判定。如果出现故障，一致性可能受到侵犯，但是终止性（和同时性）仍然会得到保证。

```
Pulse

1:      The general sends ⟨ value, x_g ⟩ to all processes,
        the lieutenants do not send.
        receive messages of pulse 1.
        The general decides on x_g.
        Lieutenants decide as follows:
            if a message ⟨ value, x ⟩ was received from g in pulse 1
              then decide on x
              else decide on udef
```

图15-2 *Broadcast* $(N, 0)$ 算法

弹性$t>0$的协议（图15-3所示的算法）递归调用弹性为$t-1$的过程。将军在脉冲1中将它的输入发送给所有中尉，在下一个脉冲中，每个中尉将所接收的值广播给其他中尉，但是这个广播的弹性为$t-1$。这个减小的弹性是算法中的微妙之处，因为（如果将军是正确的）在中尉中，有可能找出所有t个Byzantine进程，因此故障的实际数可能超过嵌套调用*Broadcast*的弹性。有必要利用弹性t和实际故障进程数f进行推理来证明所得算法的正确性（参见引理15.3）。在脉冲$t+1$中，嵌套调用产生一次判定，因此中尉p在$N-1$个嵌套广播中进行判定。这$N-1$个判定被存储在数组W_p中，由这个数组，通过多数投票（这里忽略了直接从将军处所接收的值！），得到p的判定。为此，一个确定性的函数*major*被定义在数组上，该函数具有这样的性质，即，如果v有W中的多数投票（即多于一半的输入等于v），那么$major(W)=v$。

```
Pulse

1:        The general sends ⟨ value, x_g ⟩ to all processes,
          the lieutenants do not send.
          receive messages of pulse 1.
          Lieutenant p acts as follows.
          if a message ⟨ value, x ⟩ was received from g in pulse 1
            then x_p := x else x_p := udef ;
            Announce x_p to the other lieutenants by acting as a general
                     in Broadcast_p(N − 1, t − 1) in the next pulse

(t + 1):  receive messages of pulse t + 1.
          The general decides on x_g.
          For lieutenant p:
          a decision occurs in Broadcast_q(N − 1, t − 1) for each lieutenant q.
            W_p[q] := decision in Broadcast_q(N − 1, t − 1) ;
            y_p := major(W_p)
```

图15-3　*Broadcast*（N，t）（对于$t>0$）算法

引理15.2（终止性）如果*Broadcast*(N，t）在脉冲1中开始执行，那么每个进程在脉冲$t+1$中判定。

证明。因为协议是递归的，利用对t的递归可证明它的性质。

在算法*Broadcast*(N，0）（图15-2中所示的算法），每个进程在脉冲1中判定。

在算法*Broadcast*(N，t）中，中尉在脉冲2中开始执行算法的递归调用，*Broadcast*($N-1$，$t-1$）。当在脉冲1开始执行时，算法在脉冲t判定（这是归纳假设），因此当在脉冲2中开始时，所有嵌套调用在脉冲$t+1$判定。在同一个脉冲中，在*Broadcast*(N，t）过程中进行判定。　□

为了证明相关性（也用归纳法），假设将军是正确的，因此可以在$N-1$个中尉中，找到所有t个故障进程。由于$t<(N-1)/3$不必为真，不能利用简单归纳法。我们利用故障的实际数f进行推理。

引理15.3（相关性）如果将军是正确的，且有f个故障进程，并且$N>2f+t$，那么所有正确进程对将军的输入判定。

证明。在算法*Broadcast*(N，0）中，如果将军是正确的，所有正确进程对将军的输入值进行判定。

现在假设引理对于*Broadcast*($N-1$，$t-1$）成立。因为将军是正确的，将军在脉冲1中向所有中尉发送它的输入，因此，每个正确中尉q选择$x_q=x_g$。现在$N>2f+t$蕴含着$(N-1)>2f+$

($t-1$)，因此可以把归纳假设应用到嵌套调用上，现在甚至可以在中尉中找到所有f个故障。在下一个脉冲中，每个中尉将所接收的值广播到其他中尉中。因此，对于正确中尉p和q，$Broadcast_q(N-1, t-1)$中p的判定等于x_q，即x_g。但是严格大多数中尉是正确的（$N > 2f+t$），进程p以W_p值结束，其中大多数值等于x_g。因此，p应用*major*产生出期望值x_g。 □

引理15.4 （一致性）所有正确进程对同一个值进行判定。

证明。因为相关性蕴含着执行中的一致性，在执行中，将军是正确的，我们现在集中考虑将军发生故障的情形。但另一方面，至多有$t-1$个中尉是故障的，这蕴含着嵌套调用在它们的弹性界限之内运行。 472 ~ 475

$t < N/3$蕴含着$t-1 < (N-1)/3$，因此嵌套调用满足一致性。于是，对于嵌套调用$Broadcast_q(N-1, t-1)$中的每个中尉q，所有正确中尉都会对同一个x_q的值进行判定。因此，在脉冲$t+1$中，每个正确中尉都只计算同一个向量W，这蕴含着，*major*的应用在每个正确进程中给出同样的结果。 □

定理15.5 对于$t < N/3$，$Broadcast(N, t)$协议（图15-2和图15-3所示的算法）是一个t-Byzantine-健壮广播协议。

证明。引理15.2证明了终止性。引理15.3证明了相关性。引理15.4证明了一致性。 □

广播协议在第（$t+1$）个脉冲中判定，哪一个是最优的，参见15.2.6小节。遗憾的是，这个协议的消息复杂度是指数级的，参见习题15.1。

15.1.3 多项式级的广播算法

本节给出Dolev *et al.*[DFF+82]提出的Byzantine广播算法，它的消息复杂度和位复杂度为多项式级的。时间复杂度要比前面协议的要高。算法要求$2t+3$个脉冲来实现判定。在下面的描述中，假设$N = 3t+1$，而$N > 3t+1$的情形将在以后讨论。

算法利用两个阈值，$L = t+1$和$H = 2t+1$。这些数的选取应满足（1）L个进程组成的每个集合至少包含一个正确进程，（2）H个进程组成的每个集合至少包含L个正确进程，（3）至少存在H个正确进程。注意假设$N \geqslant 3t+1$是L和H满足上述三个性质的充分必要条件。

算法交换<**bm**, v>型的消息，其中v值或为1，或为进程名。（**bm**代表“广播消息”）。进程p维持一个二维布尔表R，其中$R_p[q, v]$为真，当且仅当p已经接收到来自进程q的消息<**bm**, v>。初始时，表中的所有输入值为假，并且假设在每个脉冲的接收阶段更新这张表（算法15-4中并没有证明）。观察可见，R_p在脉冲中是单调的，即在某个脉冲中，如果$R_p[q, v]$变为真，则在以后的脉冲中它仍然为真。此外，由于正确进程只向所有近邻发送消息，在每个脉冲结束时，对于正确进程p、q和r，有$R_p[r, v] = R_q[r, v]$。 476

与前一节的广播协议不同，Dolev等人提出的协议关于值0和1是不对称的。对0的判定是默认值，如果没有足够多的消息被交换，就选择对0的判定。如果将军的输入为1，它将向所有近邻发送消息<**bm**, 1>，并且一旦收到足够多的回应<**bm**, q>型的消息，就引起进程对1判定。

算法中三种类型的活动是相关的：开始执行（initiating）、支持（supporting）和确认（confirming）。

（1）支持。如果p在q发送消息<**bm**, 1>的较早脉冲中已经接收到足够的证据，那么进程p在脉冲i中支持进程q。如果是这种情况，p将在脉冲i中发送消息<**bm**, q>。如果p已经接到来自q的消息<**bm**, 1>，则称进程p是q的直接支持者。如果p已经接收至少L个进程的消息<**bm**,

$q>$，则称进程p是q的间接支持者。p所支持的进程的集合S_p可由R_p隐含地定义为

$$DS_p = \{q : R_p[q,1]\}$$
$$IS_p = \{q : \#\{r : R_p[r,q]\} \geq L\}$$
$$S_p = DS_p \cup IS_p$$

成为间接支持者的阈值蕴含着，如果一个正确进程支持进程q，那么q已经至少发送了一条消息<**bm**，1>。实际上，假设某些正确进程支持q，并设i是发生这种情况的第一个脉冲。因为支持q间接地要求，至少接收到较早脉冲中正确进程的一条消息<**bm**，q>，正确进程对q的第一次支持是直接的。正确进程的直接支持蕴含着，这个进程已经接到来自q的消息<**bm**，1>。

（2）确认。进程p一旦接到H个进程的消息<**bm**，q>，则确认进程q，即

$$C_p = \{q : \#\{r : R_p[r,q]\} \geq H\}$$

阈值的选择蕴含着，如果一个正确进程p确认了q，那么所有正确进程至多在一个脉冲之后确认q。假设p在脉冲i之后确认q。进程p已经接收到H个进程的消息<**bm**，q>，（由阈值的选择）至少包括L个q的正确支持者。q的正确支持者向所有进程发送消息<**bm**，q>，蕴含着在第i个脉冲，所有正确进程至少接收L个消息<**bm**，q>，并在第$i+1$个脉冲中支持q。因此，在第$i+1$个脉冲中，所有正确进程发送消息<**bm**，q>，由于正确进程数至少为H，那么每个正确进程接收足够多的支持来确认q。

477

（3）开始执行。当进程p有足够多证据，证明最终判定值为1时，进程p开始执行。执行开始后，进程p发送消息<**bm**，1>。三种类型的证据可以引起初始执行，即（1）p是将军，且$x_p = 1$，（2）p在脉冲1中接收来自将军的消息<**bm**，1>，（3）在后一轮脉冲的最后，p已经确认足够多中尉。尤其需要关注最后一种可能性，因为“足够的”确认的中尉数在执行过程中会增加，确认的将军并不对此计数。在前三个脉冲中，L个中尉被确认来开始执行，但是从第四个脉冲开始，每两个脉冲阈值增加一次。因此，按照规则（3）的要求，到第i个脉冲结束时，需要确认$Th(i) = L + \max(0, \lfloor i/2 \rfloor - 1)$个中尉，才开始执行。在算法中，$C_p^L$表示确认的中尉的集合，即$C_p \setminus \{g\}$。布尔变量$ini_p$表示$p$的初始执行。

如果正确中尉r在第i个脉冲的最后开始执行，所有正确进程在第$i+2$个脉冲的最后确认r。实际上，r在第$i+1$个脉冲向所有近邻发送消息<**bm**，1>，因此在第$i+2$个脉冲中，所有正确进程（直接）支持r，在那个脉冲中，每个进程至少接收H个消息<**bm**，q>。

算法继续$2t+3$个脉冲，此时，如果进程p已经确认至少H个进程（这里将军没有计数），p对1进行判定，否则对0判定。参见图15-4所示的算法。

将军成为能够强制自己开始执行（在其他进程中）的惟一进程，在算法中占有重要地位。显而易见，如果将军正确开始执行，就会引起消息雪崩，使得所有正确进程确认H个进程，并对值1进行判定。同时，如果它没有开始执行，就不存在消息的“危险性聚集”，这些消息能使得任何正确进程进行初始执行。

引理15.6　图15-4所示的算法满足终止性（同时性）以及相关性。

证明。从算法可以看出，所有正确进程在$2t+3$个脉冲结束时进行判定，这证明了终止性和同时性。为了证明相关性，我们假设将军是正确的。

478

如果将军是正确的，且输入为1，在脉冲1中，它向所有近邻发送消息<**bm**，1>，使得每个正确进程q开始执行。因此，每个正确进程q在脉冲2中向所有近邻发送消息<**bm**，1>，到第

2个脉冲结束时，每个正确进程p支持所有其他正确进程。这蕴含着在脉冲3中，每个正确进程p向所有近邻发送支持每个正确进程q的消息<**bm**, 1>，因此，在第3个脉冲结束时，每个正确进程都从其他各正确进程接收到消息<**bm**, q>，并导致其对q的确认。因此在第3轮结束后，每个正确进程已经确认了H个进程，表明最终的判定将会是1。(所有正确进程比其他进程早一个脉冲支持并确认将军。)

```
var R_p[..., ...]  : boolean init false ;
    ini_p          : boolean init if p = g ∧ x_p = 1 then true else false ;

Pulse i: (* Sending phase *)
         if ini_p then shout ⟨bm, 1⟩ ;
         forall q ∈ S_p do shout ⟨bm, q⟩ ;
         receive all messages of pulse i ;
         (* State update *)
         if i = 1 and R_p[g, 1] then ini_p := true ;
         if #C_p^L ⩾ Th(i) then ini_p := true ;
         if i = 2t + 3 then   (* Decide *)
           if #C_p ⩾ H then y_p := 1 else y_p := 0
```

图15-4 可靠-广播协议算法

如果将军是正确的，且输入为0，在脉冲1中，它并不向所有近邻发送消息<**bm**, 1>，其他正确进程也不向所有近邻发送消息。假设在脉冲1到$i-1$中，没有正确进程已经开始执行，那么在脉冲i中没有正确进程发送消息<**bm**, 1>。在第i个脉冲结束后，没有正确进程支持或者确认任何正确进程，因为正如前面所看到的，这蕴含着以后的进程已经发送了消息<**bm**, 1>。因此，在第i个脉冲结束时，没有正确进程开始执行。由此可得，根本没有正确进程开始执行。这蕴含着没有正确进程曾经确认一个正确进程，因此没有正确进程确认多于t个进程，且在最终脉冲对0进行判定。 □

我们继续证明一致性，并且在下面的引理中，假设将军是有故障的。当至少有四个脉冲 479
要进行时，足够多会不可避免地导致1-判定的消息的“危险性聚集”，由L个正确进程的开始执行来创立。

引理15.7 *如果到第i个脉冲结束时，有L个正确进程开始执行，其中$i < 2t$，那么所有正确进程对值1判定。*

*证明。*设i是在该脉冲后至少L个正确进程开始执行的第一个脉冲，设A表示第i个脉冲结束时已经开始执行的正确进程的集合。A中所有正确进程都是中尉，因为将军是有故障的。在第$i+2$个脉冲结束时，所有正确进程已经对A中的中尉进行确认。证明在那个时刻，所有正确进程开始执行。

如果$i = 1$：到第3个脉冲结束时，所有正确进程已经确认了A中的中尉，并且开始执行，因为$\#A \geqslant L = Th(3)$。

如果$i > 2$：至少A中的一个进程，如，r在脉冲i开始执行，因为它已经确认了$Th(i)$个中尉（在第1个脉冲中，只可能通过接收将军的消息<**bm**, 1>才能开始执行）。在第$i+1$个脉冲结束时，所有正确进程确认了$Th(i)$个中尉，但是r并不属于这$Th(i)$个确认的中尉，因为在第$i+1$个脉冲r第一次发送消息<**bm**, 1>。然而，所有正确进程在第$i+2$个脉冲结束时，都将已经确认r。因此，在第$i+2$轮结束时，它们至少已经确认了$Th(i)+1$个中尉。最后，由于$Th(i$

+2) = $Th(i)$ +1，所有正确进程开始执行。

现在，由于到第i +2个脉冲结束时，所有正确进程开始执行，它们在第i +4个脉冲结束时（被所有正确进程）确认，因此所有正确进程至少已经确认了H个中尉。由于假设$i < 2t$，$i + 4 \leq 2t + 3$，因此，所有正确进程对值1进行判定。 □

对于任何对1进行判定的正确进程，由L个正确进程所进行的初始执行组成的一次雪崩是必须的。一个1-判定至少要求确认H个进程，包括L个正确进程，且这些正确进程已经开始执行。问题是是否一个Byzantine的共谋能够使雪崩的发生被推迟足够长时间，从而在某些正确进程中引发1-判定，而不用按照引理15.7，在所有进程中执行1-判定。当然，回答是否定的，因为一个共谋推迟雪崩的时间是有限的，我们精确地选择脉冲数$2t$ +3来阻止这种情形发生。原因是所要求的确认的进程数的阈值在增加。在以后的脉冲中，它变得如此之高，以至于要使下一个正确进程开始执行，L个已经开始执行的正确中尉是必须。

引理15.8　*假设在算法执行过程中至少有L个正确进程开始执行，并且设i是第一个脉冲，*
480 *在这个脉冲结束时，L个正确进程开始执行。那么$i < 2t$。*

证明。一个正确进程要在$2t$或更高的脉冲开始，至少需要确认$Th(2t) = L + (t-1)$个中尉。由于将军是有故障的，至多有$t-1$个故障中尉，因此至少有L个正确中尉必须已被确认，这表明在前面的脉冲中，L个正确进程必定已开始执行。 □

定理15.9　Dolev等人提出的广播算法（图15-4所示的算法）是一个t-Byzantine-健壮广播协议。

证明。引理15.6证明了终止性（以及同时性）和相关性。为了证明一致性，假设存在正确进程对值1进行判定。这蕴含着至少有L个正确进程已经开始执行。由引理15.8可得，这种情形在脉冲$i < 2t$中第一次发生。而另一方面，由引理15.7，所有正确进程对值1进行判定。 □

为了便于算法的表达，一直假设，进程在每一轮中重复它们在前几轮中已经发送的消息。由于正确进程记录了在前几轮中接收的消息，因此这个假设是不必要的，每条消息只发一次就够了。按照这种方式，每个正确进程向其他各进程，发送N +1条可能消息中的每一条至多一次，这使得消息的复杂度为$\Theta(N^3)$。由于只有N +1条不同的消息，每条消息只需包含$O(\log N)$位。

如果进程数超过$3t$ +1，就选择$3t$个活动中尉执行算法。（选择是静态进行的，例如，按照进程名的顺序，取名字位于g之后的$3t$个进程。）由将军和活动中尉通知被动中尉它们的判定，并且被动中尉对于从超过t个进程所接收的一个值进行判定。这种分层方法的消息复杂度为$\Theta(t^3 + t \cdot N)$，而位复杂度为$\Theta(t^3 \log t + tN)$。

15.2　鉴别协议

到目前为止所考虑的恶意行为，除了发送关于进程自身状态的不正确信息，还包括不正确的信息转发。幸运的是，利用密码手段，可以限制Byzantine进程的这种极端恶意的行为，这会致使定理15.1失效。在证明中所用的情形，即当故障进程只收到情形0的消息时，会像在
481 情形1中那样发送消息。

本节中，假设使用数字签名和鉴别（signing and authenticating）消息方法。发送消息M的进程p，在消息中增加一些额外信息$S_p(M)$，称之为p对消息M的数字签名（digital signature）。不像手写的签名，数字签名取决于M，它把拷贝签名变成其他无用消息。签名模式满足以下

性质。

（1）如果p是正确的，只有p才能可行性地计算$S_p(M)$，称这个计算为对消息M的签名。

（2）每个进程都能有效地验证（给定p、M和S），是否$S = S_p(M)$。称这个验证为对消息M的鉴别。

签名模式是基于私钥和公钥（private and public key）系统。第一条假设并没有排除Byzantine进程通过相互暴露密钥进行共谋，允许Byzantine进程伪造另一个进程的签名。假设只有正确进程才对它们的私钥保密。

我们将在15.2.2小节到15.2.5小节研究签名模式的实现。在下一小节中，进程p对消息<**msg**>的签名用序偶<**msg**>：p表示，由消息<**msg**>和S_p(<**msg**>)组成。

15.2.1 高度弹性的协议

Dolev and Strong[DS83]提出一种利用多项式的消息复杂度和$t+1$个脉冲的有效Byzantine广播算法。协议中所用的鉴别机制允许无限弹性。尽管不会有超过N个进程发生故障，且如果有N个进程发生故障，满足所有要求的集合为空，因此，设$t < N$。它们的协议是基于由Lamport、Shostak和Pease[LSP82]提出的更早的一个协议，具有指数级的消息复杂度。我们首先给出后一种协议。

在脉冲1中，将军向所有近邻发送消息<**value**，x_g>：g，包含它的（签过名的）输入。

在脉冲2至脉冲$t+1$中，进程签名并转发它们在前一次脉冲中接收到的消息。因此，在第i个脉冲中所交换的消息包含i个签名。对于接收进程p，称消息<**value**，v>：g：p_2：…：p_i为有效的，如果满足以下所有条件。

（1）所有i个签名是正确的。

（2）所有这i个签名来自i个不同进程。

（3）p并不出现在签名列表中。

在算法执行中，进程p维持值的集合W_p，这些值包含在p所接收的有效消息中。初始时，这个集合为空，然后将每个有效消息的值插入其中。

在第i个脉冲中所转发的消息，就是前一个脉冲中接收到的有效消息。在第$t+1$个脉冲结束时，进程p基于W_p进行判定。如果W_p是孤集$\{v\}$，则p对v判定；否则p对默认值（如，0）判定。为了保存消息数，p只向那些未出现在列表g，p_2，…，p_i中的进程转发消息<**value**，v>：g：p_2：…：p_i：p。这个修改对算法的行为没有影响，因为对于列表上的进程，消息不是有效的。

定理15.10 *对于$t<N$，Lamport、Shostak和Pease提出的算法是正确的Byzantine广播算法，$t<n$利用$t+1$个脉冲。*

证明。所有进程在第$t+1$个脉冲判定，蕴含着算法的终止性和同时性。

如果将军是正确的，且输入为v，所有进程在脉冲1接收它的消息<**value**，x_g>：g，于是所有正确进程在W中包含v，W中没有插入其他值，因为将军没有签过其他值。因此，在第$t+1$个脉冲，所有进程都有$W = \{v\}$，并对v判定。这蕴含着相关性。

为了证明一致性，对于正确进程p和q，在第$t+1$个脉冲结束时，我们要证明$W_p = W_q$。假设在第$t+1$个脉冲结束时，$v \in W_p$，设i是进程p一旦接收消息<**value**，v>：g：p_2：…：p_i，就将v插入W_p的那个脉冲。

情形1：如果q出现在g，p_2，…，p_i中，那么q自己已经看见v，并将它插入到W_q中。

482

情形2：如果q没有出现在序列g，p_2，…，p_i中，且$i \leqslant t$，那么p在第$i+1$个脉冲中向q转发消息<**value**，v>：g：p_2：…：p_i：p，因此q最迟在第$i+1$个脉冲中使v有效。

情形3：如果q没有出现在列表g，p_2，…，p_i中，且$i = t+1$，观察可见，p所接收的消息被$t+1$个连续进程签名，至少包括一个正确进程。这个正确进程向所有其他进程转发消息，包括q，因此q能看见v。

到第$t+1$个脉冲结束，由于$W_p = W_q$，p和q判定相同。 □

不可能在比脉冲$t+1$更早的脉冲中终止算法。在直到t的所有脉冲中，一个正确进程会接
483 收仅仅由故障处理器创建和转发的消息，并不发送给其他正确进程，这样可能导致不一致的判定。

前一个算法的中间结果，即所有正确进程中值集的一致性强于在一个值上所必须达到的一致性。Dolev and Strong[DS83]观察到这一点，并提出一个更有效的修改。事实上，在第$t+1$个脉冲结束时，或者（a）对于每个正确进程p，集合W_p是同一孤集，或者（b）对于不正确的进程p，集合W_p是孤集。在第一种情形中，所有进程对v判定，在后一种情形中，所有进程对0判定。（或者如果想要这样修改算法，则对“将军故障”进行判定）。

Dolev和Strong的算法达到了对集合W的弱要求。并非传输每个有效的消息，进程p至多转发两条消息，即一条消息带有p所接收的第一个值，另一条消息带有p所接收的第二个值。算法的完整描述留给读者。

定理15.11　*上面描述的Dolev和Strong算法是Byzantine广播协议，利用$t+1$个脉冲和至多$2N^2$条消息。*

证明。因为每个正确进程在第$t+1$个脉冲结束时判定，因此终止性和同时性和以前协议相同。相关性也可由以前协议而得。如果g在第一个脉冲中正确地向所有近邻发送v，那么在那个脉冲中，所有正确进程接受v，而不接收其他值。因此所有正确进程对v判定。由每个（正确）进程向所有近邻至多发送两条消息这样一个事实，可得所声称的消息复杂度。

为了证明一致性，我们应证明，对于正确进程p和q，在第$t+1$个脉冲结束时，W_p和W_q满足以下条件。

（1）如果$W_p = \{v\}$，那么$v \in W_q$。

（2）如果$\#W_p > 1$，那么$\#W_q > 1$。

对于（1）：假设p在脉冲i中一旦接到消息<**value**，v>：g：p_2：…：p_i，就接受值v，推理过程如同定理15.10的证明。

情形1：如果q出现在g，…，p_i中，那么q明确地接受v。

情形2：如果q没有出现在序列g，…，p_i中，且$i \leqslant t$，那么p向q转发值，在这种情形下，q将接受它。

情形3：如果q没有出现，且$i = t+1$，至少对消息签名的一个进程r是正确的。进程r已经
484 向q转发值v，蕴含着v在W_q中。

对于（2）：假设算法结束时，$\#W_p > 1$，并设w是p接收的第二个值。由类似推理可以证明，$v \in W_q$。这蕴含着$\#W_p > 1$。（不能导出W_p和W_q的相等性，因为进程p不会转发所接收的第三个值或其后所接收的值。）

已经证明了（1）和（2），假设正确进程p对$v \in W_p$判定。即$W_p = \{v\}$。那么由（1），v被包含在正确进程q的所有W_q中，但是由此可得W_q不会大于孤集$\{v\}$；否则由（2），W_p不是孤集。

因此，每个正确进程q也对v判定。进一步假设正确进程p对默认值进行判定，是因为W_p不是孤集。如果W_p是空集，由（1），每个正确进程q使W_q为空，并且由（2），如果$\#W_p > 1$，那么$\#W_q > 1$，因此，q也对默认值进行判定。 □

Dolev和Strong进一步改进了算法，并得到解决Byzantine-广播问题的一个算法，具有相同的脉冲数，只用$O(Nt)$条消息。

15.2.2 数字签名的实现

因为p的签名$S_p(M)$应该构成足够多的证据，可证明p是消息的初始者，签名必须由某种形式的信息组成，这种信息

（1）可由p进行有效的计算（签名）。

（2）除了p之外，其他进程不能对它进行有效的计算（伪造）。

我们立刻会注意到，对于今天所用的大多数签名模式，在某种程度上，第二个要求没有被证明，就是伪造问题具有指数级难度。通常伪造问题被证明与某些计算问题有关（有时等价），而这些计算问题已经被研究了很长时间，还没有得到多项式解。例如，在Fiat-Shamir模式中，伪造签名可以对大的整数进行因子分解；因为后者（假定）计算上是有难度的，前者一定也有计算上的难度。

基于各种假定的难解问题，提出了签名模式，例如计算离散对数、大数因子分解以及背包算法。要求（1）和（2）蕴含着，与其他进程相比，进程p必须有计算上的“优势”。这个优势是一些秘密信息，由p保留，称为p的*密钥*或者*私钥*。于是当密钥已知时，$S_p(M)$的计算是有效的。而没有这个信息，计算就（假设）是困难的。显然，如果p成功地保守了它的秘密，只有p能够顺利地计算$S_p(M)$。 485

所有进程必须能够验证签名，即，给定消息M和签名S，必须能够有效地验证，S确实是利用p的密钥对消息M计算而得。这种验证要求揭示关于p的某些密钥的信息。称这种信息为p的公钥（public key）。公钥允许对签名进行验证，但是可能的或至少计算上是有难度的，利用公钥计算p的密钥或者伪造签名。

到目前为止，最成功的签名模式是基于在算术环中模大数的数论计算。在这些环中可以执行基本的算术操作，如，加、乘和求幂运算，时间复杂度为（按位）模长度的多项式时间。如果分母与模互素（即，没有公共素因子），做除法也是可能的，并且也可用多项式时间完成。因为签名和验证要求对消息进行计算，因此可将M解释为一个大数。

15.2.3 ElGamal签名模式

ElGamal签名模式[ElG85]是基于称为*离散对数*（discrete logarithm）的数论函数。对于大素数P，用$\mathbb{Z}^*_p$表示的模P的乘法群，包含$P-1$个元素，且是*循环的*。后者的含义是，选择元素$g \in \mathbb{Z}^*_p$，满足这$P-1$个数，

$$g^0 = 1, g^1, g^2, \ldots, g^{P-3}, g^{P-2}$$

互不相同，因此枚举$\mathbb{Z}^*_p$中的所有元素。称这样的g为$\mathbb{Z}^*_p$的*生成元*（generator）。也称模P的*原根*（primitive root）。生成元不是惟一的，通常有多个生成元。给定固定数P和生成元g，对于每一个$x \in \mathbb{Z}^*_p$，存在模$P-1$的惟一整数i，满足$g^i = x$（等式在$\mathbb{Z}^*_p$中）。称这个i为x的*离散对数*

(有时称下标)。与上述提到的基本算术操作不同，这些离散对数的计算是不容易的。这是一个到目前为止，经过研究，仍未找到有效通用解决方法的问题，但是这个问题也没有被证明是难解的。文献[Odl84]有对结果的概述。

ElGamal[ELG85]签名模式是基于计算离散对数的难度。进程共享大素数P和$\mathbb{Z}^*_p$的一个原
486 根g。进程p在1和$P-2$之间随机地选择一个数d作为它的密钥，p的公钥为$e = g^d$；观察可见，d是e的离散对数。已知e的对数，可以有效地计算p的签名。于是形成一个隐含证明，即，签名者知道d。

对于消息M的有效签名模式是数对(r, s)，满足$g^M = e^r r^s$。利用密钥d，p很容易找到这样的数对。进程p选择一个随机数a，且该数与$P-1$互素，计算

$$r = g^a \pmod{P}$$

且

$$s = (M - dr)a^{-1} \pmod{(P-1)}$$

这些数确实满足

$$\begin{aligned} e^r \cdot r^s &= e^r (g^a)^{(M-dr)a^{-1}} \\ &= g^{dr} g^{M-dr} = g^M \end{aligned}$$

(所有等式在$\mathbb{Z}^*_p$中。)通过检查是否$g^M = e^r \cdot r^s$，很容易验证对消息M签名$S = (r, s)$的有效性。

1. 离散对数算法

因为p的密钥d等于它的公钥e的离散对数，如果模P的离散对数能被有效地计算，则很容易破解这种模式。到目前为止，一般情况下，还没有有效的算法能做到这一点，或者用任何其他方法伪造签名。

计算离散对数的一般算法是由Odlyzko[Odl84]提出的。它的复杂度和最著名的整数P的因子分解具有相同的数量级。算法首先只利用P和g来计算几张表，在第二步，计算给定数的对数。如果Q是$P-1$中的最大素因子，那么第一步的计算时间及表的大小与Q具有相同的数量级。因此，应选择P，满足$P-1$中有大的素因子。第二步计算对数，甚至在很小的计算机上只需几秒就可完成。因此需要经常充分地改变P和g，比如每个月，使得在它们的计算完成之前，针对某个特定的P的表已经过时。

2. 随机签名

签名过程的随机化，可以使得对于给定的消息，$\varnothing(P-1)$个不同签名[⊖]出现的概率相同。
487 因此，对于签过两次名的同一个文档，几乎肯定会产生两个不同的有效签名。在签名过程中，随机化是至关重要的。如果p利用同一个值a为两个消息签名，那么由签名可以计算出p的密钥。参见习题15.6。

15.2.4 RSA签名模式

如果n是个大整数，是两个素数P和Q乘积，则计算模n的平方根和更高阶根是相当难的，

⊖ 称函数$\varnothing$为*Euler phi*函数；(n)为$\mathbb{Z}^*_n$的大小。

除非已知其中的因子分解。可以利用计算平方根的能力找出n的因子（参见习题15.7），这证明了平方根的计算和因子分解一样困难。

在Rivest、Shamir and Adleman[RSA78]提出的签名模式中，p的公钥是一个大数n，且p知道其中的因子分解和一个指数e。p对消息M的签名是M的第e个根（模n），利用求幂计算对此很容易验证。p也能利用求幂计算找出高阶根。当生成它的钥匙时，p计算一个数d，满足$de = 1(\mathrm{mod}(\varnothing(n))$，这蕴含着$(M^d)^e = M$，即$M^d$是$M$的第$e$个根。$P$的密钥只由数$d$组成，即$p$不需要记住$n$的因子分解。

在RSA模式中，p通过计算模n的根表明它的标识，这（隐含）要求知道关于n的因子分解。只有p被假定具有这个知识。在这种模式中，每个进程利用不同的模。

15.2.5 Fiat-Shamir签名模式

Fiat和Shamir[FS86]提出的模式中更微妙地用到了求（平方）根的难度。在RSA模式中，进程的签名，是通过证明它能够计算根（模公钥）来完成的。假定计算根的能力要求因子分解知识。在Fiat-Shamir签名模式中，进程利用了公共模数n，其中的因子分解只对可信的中心而言是可知的。已知某些特定数（依赖p的标识）的平方根为进程p所知，p对M的签名提供了签名者知道这些平方根的证据，但并未揭示这些方根是什么。

Fiat-Shamir签名模式优于RSA模式的优势在于，它具有较低的算术复杂度，并且每个进程没有单独公钥。缺点是需要一个可信权威机构来分发这些密钥。正如以前所提到的，模式利用了大整数n，是仅为中心所知的两个大素数的乘积。另外，存在一个单向伪-随机函 488
数f，将串映射到$\mathbb{Z}_n$上。这个函数是已知的，每个进程都可计算它。但是它的逆函数是不能计算出的。

1. 密钥和公钥

作为密钥，给定k个模n的数s_1到s_k作为p的方根，即$s_j = \sqrt{v_j^{-1}}$，其中，$v_j = f(p, j)$。可以把v_j作为p的公钥，但是因为它们可由p的标识计算而得，因此不需要存储它们。为了避免技术上的麻烦，我们假设这k个数都是模n的二次剩余。平方根可由中心进行计算，它知道n的因子。

2. 签名消息：首次尝试

p的签名隐含地证明了签名者知道v_j的根，即它可提供一个数s满足$s^2v_j = 1$。这个数就是s_j，但是发送s_j自身可能暴露密钥。为了避免暴露密钥，模式利用了以下的思想。进程p选择随机数r，并计算$x = r^2$。现在p是惟一能够提供满足$y^2\ v_j = x$的数y的进程，即$y = rs_j$。因此，通过发送满足$y^2v_j = x$的数对(x, y)，p可以证明它对s_j有所知，而不会暴露它。由于p没有发送数r，不计算平方根，而从这个数对计算s_j是不可能的。

但是由这样的数对组成的签名，存在两个问题。第一，任何进程都可以按照以下方法通过欺骗产生这样的数对：首先选择y，然后计算$x = y^2v$。第二，签名并不依靠消息，因此接收来自p的签过名的消息的进程可以把这个签名拷贝到任何伪造消息中。签名模式的关键是使p表明，它知道v_j的一个子集乘积的根，而子集依靠消息和随机数。消息的不规则性以及通过f的随机数可以防止伪造者首先选择y。

为了对消息M签名，p进行如下计算。

（1）p选择随机数r，并计算$x = r^2$。

（2）p计算$f(M, x)$；前k位是e_1到e_k。

（3）p计算$y = r\prod_{e_j=1}s_j$

签名$S_p(M)$由元组（e_1，…，e_k，y）组成。

489 为了验证p对消息M的签名（e_1，…，e_k，y），进行如下计算。

（1）计算v_j且$z = y^2\prod_{e_j=1}v_j$。

（2）计算$f(M, z)$，并且验证前k位是e_1到e_k。

如果签名是真实的，那么在验证的第一步中所计算的z值等于签名中所用的x值。因此，$f(M, z)$的前k位等于e_1，…，e_k。

3. 伪造和最终解

我们现在考虑一种策略，一个伪造者在不知道s_j的情况下，按照上述的模式获得一个签名。

（1）选择k个随机位e_1，…，e_k。

（2）选择随机数y，并计算$x = y^2\prod_{e_j=1}v_j$。

（3）计算$f(M, x)$，并且验证它的前k位是否等于之前选择的e_1，…，e_k。如果相等，那么（e_1，…，e_k，y）是对消息M的伪造签名。

由于可以假设第（3）步中相等的概率为2^{-k}，那么经过预期的2^k次试验后，伪造成功。

如果$k = 72$，并假设需要10^{-9}秒来试验对e_j的一次选择，伪造所需的期望时间为$2^{72} \cdot 10^{-9}$秒，或者一百五十万年，这使得模式很安全。然而，每个进程必须存储k个根，并且由于空间的限制，如果k值必须是有限的，预期的2^k伪造时间可能是不能令人满意的。我们现在证明如何修改这个模式，以便利用k个根，对于所选择的整数t，获得期望值为2^{kt}的伪造时间。设想利用f-结果的前kt位，定义v_j的t个子集合，使p表明它有t个这些子集的乘积知识。为了对消息M签名，p进行如下计算。

（1）p选择随机数r_1，…，r_t，并计算$x_i = r^2_i$。

（2）p计算$f(M, x_1, \ldots, x_t)$，它的前kt位为e_{ij}（$1 \leqslant i \leqslant t$且$1 \leqslant j \leqslant k$）。

（3）对于$1 \leqslant i \leqslant t$，$p$计算$y_i = r_i\prod_{e_{ij}=1}s_j$。签名$S_p(M)$由元组（$e_{11}$，…，$e_{tk}$，$y_1$，…，$y_t$）组成。

为了验证p对消息M的签名（e_{11}，…，e_{tk}，y_1，…，y_t），进行如下计算。

（1）计算v_j且$z_i = y_i^2\prod_{e_{ij}=1}v_j$。

（2）计算$f(M, z_1, \ldots, z_t)$，并且验证前kt位是e_{11}，…，e_{tk}。

在第三步中，试图用如上所述的策略，产生一个有效签名的伪造者，其成功概率为2^{-kt}，
490 这蕴含预期的试验数为2^{kt}。Fiat和Shamir在他们的论文中证明，除非对n的因子分解变得容易，否则不存在本质上更好的伪造算法，因此只要选择k和t足够大，就可以使模式是安全的。

15.2.6 概述和讨论

本节以及前一节已经表明，对于Byzantine广播问题，在同步系统中存在确定解。如果不用鉴别（15.1节），这个解的最大弹性为$t < N/3$，而如果使用消息鉴别（本节），那么这个弹性是无限的。这里所给出的所有解中，我们一直用脉冲模型假设来模拟同步过程。在15.3节讨论了脉冲模型的容错实现。

1. 触发组问题

除了假设脉冲模型，第二个目前所给出的所有解决方法所基于的假设是，所有进程知道在哪一个脉冲中开始广播（为方便起见，标号为1）。如果这不是预先的情况，那么在一个或

者多个进程（自发地）开始执行广播算法的一个执行请求之后，会引起同时开始执行算法的问题。请求可能来自将军（计算出一个必须向所有进程宣布的结果之后），或者来自中尉（认识到它们都需要存储在将军中的信息）。文献中这个问题是作为触发组问题加以研究的。在这个问题中，一个或者多个进程开始执行（请求），但不必是在同一个脉冲中，进程可能被触发(fire)。要求是:

（1）**有效性** 除非某些进程已经开始执行，否则没有正确进程触发。

（2）**同时性** 如果任一正确进程触发，那么所有正确进程在同一个脉冲中触发。

（3）**终止性** 如果一个正确进程开始执行，那么所有正确进程在有限个脉冲内触发。

实际上，给定触发组问题的解，广播的第一个脉冲无需预先达成一致；请求广播的进程开始执行触发组算法，并且广播在触发之后的脉冲中开始。可以把在Byzantine-广播问题和触发组问题解中所用的技术组合，得到更有时效的协议，即，在没有关于第一个脉冲的优先一致性的情况下，可以直接地求解广播问题。 491

2. 时间复杂度和提前停止协议

本章中，我们已经给出了利用$t+1$或$2t+3$个脉冲或进行数轮通信的协议。Fischer and Lynch[FL82]已经证明，对于t-健壮一致协议，$t+1$轮的通信是最优的，并对这一结果作了扩展，涵盖了Dolev and Strong[DS83]提出的鉴别协议。

在这些证明中，细微之处在于，在所使用的情形中进程一定会在脉冲1至脉冲t中发生故障。因此下界是执行过程中实际故障数最坏情况。因为在大多数执行中，实际故障数要比弹性更低，已经研究了是否存在在只有少量故障的那些执行中能够较早达到一致性的协议。具有这种性质的广播协议称为*提前停止*（early stopping）协议。对于有f个故障的一次执行中的协议，Dolev、Reischuk and Strong[DRS82]证明了一个$f+2$轮的下界。 在文献[BGP92]中，可以找到几个关于提前停止广播性协议和一致性协议的讨论。

在正确进程断定已经有某个脉冲没有出现新故障后的几个脉冲内，早停协议进行判定。然而，不能保证，所有正确进程在同一个脉冲都得出这个结论。（除非它们在第$t+1$个脉冲中进行判定，因为至多有t个进程发生故障，则在前$t+1$轮中，存在没有出现新故障的一轮。）因此，早停协议并不满足同时性。Coan and Dwork[CD91]证明，为了达到同时性，在执行中，$t+1$轮不出现新故障是必要条件，甚至对于随机协议和在（非常弱的）损毁模型中也是这样。这蕴含着经鉴别的协议也需要$t+1$个脉冲来达到同时性。

3. 判定问题和交互一致性

利用广播协议作为子例程，实际上，在同步系统中，通过达到*交互一致性*（interactive consistency），可以解决所有判定问题，即，输入集上的一致性。在交互一致性问题中，进程对输入的向量进行判定，系统中的每个进程一个输入。形式上，有如下要求:

（1）**终止性** 每个正确进程p对向量V_p进行判定，每个进程一个输入。

（2）**一致性** 正确进程的判定向量是相等的。

（3）**相关性** 如果q是正确的，那么对于正确进程p，$V_p[q]=x_q$。

通过多次广播，可以实现交互一致性: 每个进程广播它的输入，并且进程p把它在q的广播中的判定放在$V_p[q]$中。从广播算法的相应性质，终止性、一致性和相关性直接被继承。 492

因为每个正确进程计算同一个向量（一致性），利用关于判定向量（该向量可以立即保证一致性）的确定函数，可以容易地解决大多数的判定问题。例如，可以抽取判定向量的大多

数值，即可解决一致性问题。在判定向量中选择最小惟一标识可以解决选举问题（注意，所选的进程可能是有故障的）。

15.3 时钟同步

前节证明，（当考虑确定算法时）同步系统比异步系统具有更高的弹性。对于理想同步模型，其中进程按照脉冲执行，得出了这一结果。脉冲模型具有更高弹性蕴含着，不可能用健壮方法确定地同步完全异步网络。本节将证明，在异步有限延迟网络（ABD网络）中实现健壮的脉冲模型是可能的。

ABD模型被为局部时钟可用和消息延迟有上界。在算法描述和分析中，我们利用实时帧（real-time frame），它是对每个事件赋给一个发生时间$t\in\mathbb{R}$。按照相对物理学，不存在标准的或者特定的方法来进行这个赋值。下面我们假设，已经选择了具有物理意义的赋值。系统中的进程不能看到实时帧，但是进程可以利用它们的时钟间接地观察到时间。时钟的值与实时时间有关。用C_p表示进程p的时钟，进程p可以对它进行读写（向时钟写需要同步）。当时钟没有被赋值时，时钟值按照时间连续变化。$C_p(t) = T$表示在实时时间t，读出时钟值为T。

大写字母（C，T）用于表示时钟时间，小写字母（c，t）用于表示实时时间。时钟可用于控制事件的发生，如在

when $C_p = T$ **then** send message

493 时，引起在时间$C_p^{-1}(T)$发送消息。用c_p表示函数C_p^{-1}。

理想时钟值按照Δ个时间单位每次增加Δ。即它满足$C(t+\Delta) = C(T=t)+\Delta$。理想时钟，一旦同步再不需要调整，但遗憾的是，它们只是一种（有用的）数学抽象。分布式系统中所用的时钟会遭受漂移（drift），但漂移限定在一个小的已知常数ρ的范围内（典型情况下，阶为10^{-5}或10^{-6}）。时钟C的漂移是ρ-有界的，如果对于t_1和t_2，满足对C的赋值不会出现在t_1和t_2之间。

$$(t_2-t_1)(1+\rho)^{-1} \leqslant C(t_2) - C(t_1) \leqslant (t_2-t_1)(1+\rho) \tag{15-1}$$

在任何给定的实时时间，分布式系统中的各种时钟并不能显示相同的时钟时间。即$C_p(t) = C_q(t)$不必成立。如果$|C_p(t) - C_q(t)| < \delta$，则在实时时间$t$时钟是$\delta$-同步的。如果$|c_p(T)-c_q(T)| < \delta$，则在时钟时间$T$时钟是$\delta$-同步的。我们将把这些表示视为等价的。参见习题15.8。时钟同步算法的目标是达到和维持全局δ-同步。即在每一对时钟之间δ-同步。参数δ是同步的精度。

δ_{min}表示消息延迟下界，δ_{max}表示消息延迟上界，其中$0 \leqslant \delta_{min} < \delta_{max}$，形式地，如果在实时时间$\sigma$发送一条消息，在实时时间$\tau$接收这条消息，那么

$$\delta_{min} \leqslant \tau-\sigma \leqslant \delta_{max} \tag{15-2}$$

因为实时时间帧的选择是任意的，假设式（15-1）和式（15-2）与时间帧以及时钟和通信系统相关。

15.3.1 读取远程时钟

本节研究进程p利用精度，将自身理想时钟调整成可靠服务器s的理想。利用确定协议，

能达到的最好精度是$1/2(\delta_{max}-\delta_{min})$。这个精度可以通过只交换一条消息的简单协议达到。概率协议可以达到任意精度，但消息复杂度依赖于所要求的精度和消息传输时间的分布。

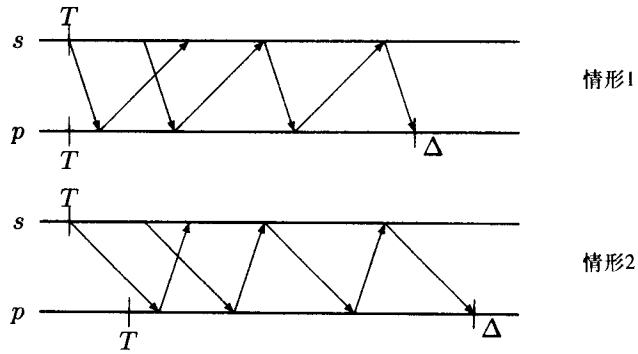


图15-5 确定协议的情形

定理15.12 *存在以精度$1/2(\delta_{max}-\delta_{min})$同步$C_p$到$C_s$的确定协议，只交换一条消息。不存在达到更高精度的确定协议。*

证明。我们首先提出一个简单协议，并证明它达到了定理中所声称的精度。为了使C_p同步，服务器发送一条消息<**time**，C_s>。当进程p接收到消息<**time**，T>时，就将它的时钟调整到$T+1/2(\delta_{max}+\delta_{min})$。

为了证明所声称的精度，称发送和接收消息<**time**，T>的实时时间分别为σ和τ。现在，$T=C_s(\sigma)$。因为时钟是理想的，$C_s(\tau)=T+(\tau-\sigma)$。在时间$\tau$，$p$将它的时钟调整到$C_p(\tau)=T+1/2(\delta_{max}+\delta_{min})$，因此，$C_s(\tau)-C_p(\tau)=(\tau-\sigma)-1/2(\delta_{max}+\delta_{min})$。现在$\delta_{min}\leqslant\tau-\sigma\leqslant\delta_{max}$，蕴含着$|C_s(\tau)-C_p(\tau)|\leqslant 1/2(\delta_{max}-\delta_{min})$。

为了证明精度的下界，设给定确定协议。在这个协议中，p和s交换某些消息，交换之后，p调整它的时钟。考虑协议的两种情形，如图15-5所示。在第一种情形中，执行之前，时钟是相等的，从s到p的所有消息在δ_{min}之后传输，而从p到s的所有消息在δ_{max}之后传输。如果在这种情形中所做的调整是Δ_1，那么同步之后，p的时钟比C_s恰好提前Δ_1。

在第二种情形中，执行之前，C_p比C_s落后$\delta_{max}-\delta_{min}$。从$p$到$s$的所有消息在$\delta_{min}$之后传输，而从$s$到$p$的所有消息在$\delta_{max}$之后传输。称在这种情形下所做的调整是$\Delta_2$，那么同步之后，我们发现，同步之后$p$的时钟恰好比$C_s$落后$\delta_{max}-\delta_{min}-\Delta_2$。

然而，p或者r都不能发现这两种情形之间的差异，因为消息延迟的不确定性掩盖了这种差异，因此，$\Delta_1=\Delta_2$。这蕴含着最坏情况下的最好精度为

$$\min_{\Delta}\max(|\Delta|,\ |\delta_{max}-\delta_{min}-\Delta|)$$

这个最小值等于$1/2(\delta_{max}-\delta_{min})$（当$\Delta=1/2(\delta_{max}-\delta_{min})$时出现最小值）。 □

494 ~ 495

如果两个进程p和q都以这个精度将它们的时钟与服务器同步，就可达到全局（$\delta_{max}-\delta_{min}$）同步，对于大多数应用，这已足够。

Cristian[Cri89]提出了一种概率同步协议，可以达到更好的精度。对于这个协议，假设消息延迟是一个随机变量，按照函数（不必已知）F: $[\delta_{min},\ \delta_{max}]\rightarrow[0,\ 1]$分布。任何消息到达$x$内的概率为$F(x)$，且不同消息的延迟是独立的。如果下界$\delta_{min}$是紧的，则可达到任意的精度，

即，对于所有$x>\delta_{min}$，$F(x)>0$。

协议很简单，p要求s发送时间，而s立即用消息<**time**，T>进行响应。进程p测量发送请求和接收响应之间的时间I，$I\geqslant 2\delta_{min}$成立。响应消息的延迟至少为δ_{min}，至多为$I-\delta_{min}$，因此至多与$(1/2)I$相差$(1/2)(I-2\delta_{min})$。于是p可以将它的时钟设置到$T+(1/2)I$，并达到精度$(1/2)(I-2\delta_{min})$。假设所需的精度是ε，如果$(1/2)(I-2\delta_{min})>\varepsilon$，进程$p$发送一个新的请求；否则算法终止。

引理15.13　*概率时钟-同步协议以至多为$F(\delta_{min}+\varepsilon)^{-2}$的期望消息数，达到精度$\varepsilon$。*

证明。进程p的请求在$\delta_{min}+\varepsilon$内到达的概率为$F(\delta_{min}+\varepsilon)$，响应在$\delta_{min}+\varepsilon$内到达的概率也为$F(\delta_{min}+\varepsilon)$。因此，进程$p$在$2\delta_{min}+2\varepsilon$内接收这个响应的概率至少为$F(\delta_{min}+\varepsilon)^2$，这蕴含在进行成功消息交换之前，所期望的试验数界限为$F(\delta_{min}+\varepsilon)^{-2}$。□

如果p发送新的请求，发送之后，如果在$2\delta_{min}+2\varepsilon$内没有收到响应，则协议的时间复杂度减少。期望时间独立于期望延迟或最大延迟，即$2(\delta_{min}+\varepsilon)\ F(\delta_{min}+\varepsilon)^{-2}$，协议在抗消息损失方面是健壮的。（必须用序列号区别过时的响应。）

15.3.2　分布式时钟同步

本节给出了Mahany and Schneider[MS85]提出的t-Byzantine-健壮（对于$t<N/3$）分布式时钟-同步算法。Dolev、Halpern and Strong[DHS84]证明，如果$t\geqslant N/3$，如果不用鉴别，是不可能达到同步。

同步算法的核心是一个协议，这个协议达到时钟平均值的*不精确一致性*（inexact
496 agreement）。进程调整它们的时钟，达到高度同步。由于漂移，不久，精度就会降低，在经过一定的间隔之后，需要新一轮的同步。假设在实时时间t_0，时钟是δ_0-同步的，那么直到时间t_0+R，时钟是$(\delta_0+2\rho R)$-同步的。因此，如果期望精度为δ，并且某个同步轮达到精度δ_0，那么每$(\delta-\delta_0)/2\rho$时间单位就要重复轮。与R相比，执行一次同步轮的时间S通常是非常小的。因此，在同步中，可以忽略漂移，即，时钟是理想的，这一简化的假设是合理的。

1. 不精确一致性：快速收敛算法

在由Mahany和Schneider[MS85]所使用的用于使时钟同步的不精确一致性问题中。进程p的输入值为实数x_p，对于正确进程p和q，$|x_p-x_p|\leqslant\delta$。进程p的输出为实数值y_p，输出的精度定义为$\max_{p,q}|y_p-y_q|$。算法的目标是获得很小的精度值。

Mahany和Schneider提出的快速收敛算法参见图15-6所示的算法。对于有限集合$A\subset\mathbb{R}$，定义两个函数$intvl(A)=[\min(A),\max(A)]$和$width(A)=\max(A)-\min(A)$。算法由输入收集阶段和计算阶段组成。在第一阶段中，进程p请求其他每个进程发送它的输入（通过向所有近邻发送<**ask**>消息），并等待$2\delta_{max}$时间单位。在这个时间过后，进程p已经收到所有正确进程的输入，以及来自故障进程子集的应答。对于不能应答的进程，上加上值∞（无意义）。

然后进程对接收的值进行过滤，这确保了所有正确进程的值以及那些与正确值足够接近的错误值通过。当正确值仅差δ时，至少有$N-t$个正确值，每个正确值至少有$N-t$个值，这些值与它至多相差δ。A_p存储具有这个性质的所接收的值。

然后，通过对这些值求平均值来计算输出，所有被拒绝的值用一个估算值替代，估算值是通过将确定函数$estimator$应用于生存下来的值计算而得。这个函数满足$estimator(A)\in intvl(A)$，但另外也可以是任意的。它可以是最小、最大、平均或者$(1/2)[\max(A)+\min(A)]$。

定理15.14　*快速收敛算法达到精度$2t\delta/N$。*

证明。当进程p超时（即v_{pr}或为x_r或为∞），设v_{pr}是包含在V_p中的进程r的值。当进程p计算

y_p时（即，a_{pr}或为v_{pr}或为$esti_p$），设a_{pr}是包含在A_p中进程r的值。把判定计算求和分成两部分，一部分为正确进程（C）上的求和，另一部分为不正确进程（B）上的求和，就可以对精度加以限定。对于正确进程p和q，如果$r \in C$，这个差$|a_{pr} - a_{qr}|$界限为0；如果$r \in B$，这个差$|a_{pr} - a_{qr}|$界限为2δ。

```
var x_p, y_p, esti_p   : real ;      (* Input, output, estimator of V *)
    V_p, A_p           : multiset of real ;

begin (* Input collection phase *)
      V_p := ∅ ;
      forall q ∈ ℙ do send ⟨ask⟩ to q ;
      wait 2δ_max ; (* Process ⟨ask⟩ and ⟨val, x⟩ messages *)
      while #V_p < N do insert(V_p, ∞) ;
      (* Now compute acceptable values *)
      A_p := {x ∈ V_p : #{y ∈ V_p : |y − x| ⩽ δ} ⩾ N − t} ;
      esti_p := estimator(A_p) ;
      while #A_p < N do insert(A_p, esti_p) ;
      y_p := (∑ A_p)/N
end

Upon receipt of ⟨ask⟩ from q:
      send ⟨val, x_p⟩ to q

Upon receipt of ⟨val, x⟩ from q:
      if no such message was received from q before
        then insert(V_p, x)
```

图15-6 快速收敛算法

因为如果p和r是正确进程，则$a_{pr} = x_r$，第一个界限由此而得。确实，由于r对p的消息<ask>迅速响应，$v_{pr} = x_r$。类似地，对于所有正确进程r'，$v_{pr'} = x_{r'}$，而对于输入的假设蕴含着，p过滤后，r的值存活下来，因此$a_{pr} = v_{pr}$。

第二个界限成立是由于，对于正确进程p和q，当进程p和q计算它们的判定时，$width(A_p \cup A_q) \leqslant 2\delta$。因为相加的估算值位于可接受的值之间，因此只需分别考虑通过p和q过滤的那些值a_p和a_q之间的最大差就足够了。对于$|v_{pr} - a_p| \leqslant \delta$，至少有$N - t$个进程$r$，而对于$|v_{qr} - a_p| \leqslant \delta$，至少有$N - t$个进程$r$。这蕴含着存在正确进程$r$，满足$|v_{pr} - a_p| \leqslant \delta$和$|v_{qr} - a_p| \leqslant \delta$，但因$r$是正确的，$v_{pr} = v_{qr}$，因此$|a_p - a_q| \leqslant 2\delta$。

由此可得，对于正确进程p和q，

$$
\begin{aligned}
|y_p - y_q| &= |(\Sigma A_p)/N - (\Sigma A_q)/N| \\
&= \frac{1}{N} \cdot \left| \left(\sum_{r \in C} a_{pr} + \sum_{r \in B} a_{pr} \right) - \left(\sum_{r \in C} a_{qr} + \sum_{r \in B} a_{qr} \right) \right| \\
&= \frac{1}{N} \cdot \left| \left(\sum_{r \in C} a_{pr} - \sum_{r \in C} a_{qr} \right) + \left(\sum_{r \in B} a_{pr} - \sum_{r \in B} a_{qr} \right) \right| \\
&\leqslant \frac{1}{N} \cdot \left[\left(\sum_{r \in C} |a_{pr} - a_{qr}| \right) + \left(\sum_{r \in B} |a_{pr} - a_{qr}| \right) \right] \\
&\leqslant \frac{1}{N} \cdot \left[(0) + \left(\sum_{r \in B} 2\delta \right) \right] \leqslant 2t\delta / N
\end{aligned}
$$

□

重复该算法可以达到任意的精度，当迭代i次后，精度变成$\left(\frac{2}{3}\right)^i\delta$。如果进程的一小部分（少于三分之一）是故障进程，这个精度还会更好。在导出这个精度的过程中，可以将t看作故障进程的实际数。即使选择合适的函数*estimator*，也不能提高算法的（最坏情况）输出精度。事实上，通过简单地将这个值发送给p，Byzantine进程r可以使p取$a_{pr} \in intvl\ (A_p)$中的任何值。如果知道错误进程的最可能的行为，则可以选择合适的函数，达到一个良好的平均精度。

2. 时钟同步

为同步时钟，利用快速收敛算法来达到对时钟新值的不精确的一致性。假设初始时，时钟是δ-同步的。必须修改算法，因为

（1）由于不能确切得知消息延迟时间，因此一个进程不知道另一个进程的精确值，且

497
~
499

（2）算法执行过程时间流逝，因此时钟没有常数值，而是随时间增加。

为了补偿未知延迟，进程对接收的时钟值增加$(1/2)(\delta_{\max}+\delta_{\min})$（如定理15.12的确定性协议所做的），在输出精度中引入增量项$\delta_{\max}-\delta_{\min}$。为了把接收的值表示成时钟值，而不是一个常数，$p$将接收的时钟值（加上$(1/2)(\delta_{\max}+\delta_{\min})$）与它自己值的差存为$\Delta_{pr}$。在时间$t$，$p$对于$r$的时钟的近似值为$C_p(t)+\Delta_{pr}$。修改的算法为图15-7所示的算法。

```
var C_p, Δ_p, esti_p   : real ;      (* Clock, adaptation, estimator of V *)
    D_p, A_p           : multiset of real ;

begin (* Input collection phase *)
      D_p := ∅ ;
      forall q ∈ ℙ do send ⟨ask⟩ to q ;
      wait 2δ_max ; (* Process ⟨ask⟩ and ⟨val, x⟩ messages *)
      while #D_p < N do insert(D_p, ∞) ;
      (* Now compute acceptable values *)
      A_p := {x ∈ D_p : #{y ∈ D_p : |y − x| ⩽ δ + (δ_max − δ_min)} ⩾ N − t} ;
      esti_p := estimator(A_p) ;
      while #A_p < N do insert(A_p, esti_p) ;
      Δ_p := (∑ A_p)/N ;
      C_p := C_p + Δ_p
end

Upon receipt of ⟨ask⟩ from q:
      send ⟨val, C_p⟩ to q

Upon receipt of ⟨val, C⟩ from q:
      if no such message was received from q before
        then insert(D_p, (C + ½(δ_max + δ_min)) − C_p)
```

图15-7　时钟的快速收敛算法

图15-7所示的算法中的过滤器，与图15-6所示的算法中的相比，有更广的界限，即$\delta+(\delta_{\max}-\delta_{\min})$，而在图15-6所示的算法中，这个界限为$\delta$。更广的界限补偿了未知消息的延迟，并由以下命题得出阈值。设d_{pr}表示p的第一阶段后，插入到D_p中的值（与上一个算法中的值v_{pr}相比较）。

命题15.15　对于正确进程p、q和r，在进程p超时后，$|d_{pr}-d_{qr}| \leqslant \delta+(\delta_{\max}-\delta_{\min})$成立。

证明。从q到p的消息<**val**, C>交换，实现了定理15.12中时钟读取的确定算法。当进程p接收到该消息时，$|C_p - [C + 1/2(\delta_{max} + \delta_{min})]|$界限为$(1/2)(\delta_{max} - \delta_{min})$，因此$d_{pq}$与$C_q - C_p$至多相差$1/2(\delta_{max} + \delta_{min})$。类似地，$d_{pr}$与$C_r - C_p$至多相差$(1/2)(\delta_{max} + \delta_{min})$。因为$C_q$与$C_r$至多相差$\delta$，因此结论成立。 □ 500

定理15.16 图15-7所示算法执行后，时钟是同步的，且精度为

$$(\delta_{max} - \delta_{min}) + \frac{2t}{N}[\delta + (\delta_{max} - \delta_{min})]$$

证明。在证明中，C_p表示未调整的时钟，C_p'表示经调整的时钟，即$C_p'(t) = C_p(t) + \Delta_p$。为了限制所调整时钟的精度，固定实时时间$t$，该时间迟于所有正确进程的超时，并设$w_{pr} = C_p + d_{pr}$。由命题的证明可以得出（对于正确进程$p$、$q$和$r$）$|w_{pr} - C_r(t)| \leq 1/2(\delta_{max} - \delta_{min})$，这蕴含着$|w_{pr} - w_{qr}| \leq (\delta_{max} - \delta_{min})$。对于不正确的$r$，差$|w_{pr} - w_{qr}|$的界限为$2\delta + 3(\delta_{max} - \delta_{min})$，它的证明类似于定理15.14中相应步的证明。

最后，由于$C_p'(t) = C_p(t) + \Delta_p = (\sum_r wPr)/N$，如同定理15.14的证明，将平均值分为正确进程上的平均值与不正确进程上的平均值，则可导出精度。 □

算法描述蕴含所有消息<**val**, C>与未调整的时钟值一起发送，要做到这一点可以通过延迟时钟调整，直到向所有<**ask**>消息都已发送应答。

15.3.3 轮模型的实现

可用具有弱同步假设的系统，模拟同步计算的脉冲模型。假定

（1）假设存在消息延迟上界δ_{max}。

（2）在一个脉冲（状态改变的时间加上发送消息的时间）中，局部计算所需时间具有上界γ；且

（3）在每个时钟时间T，进程的时钟是δ-同步的，并且漂移为ρ-有界的。

每个脉冲，模拟需要$(1 + \rho)(\delta_{max} + \delta + \gamma)$时钟时间。

模拟算法非常基本，假设模拟的算法在时钟时间0开始执行（通过向所有时钟时间加上一个固定项T_0，可以稍后开始执行算法）。当进程的时钟读到0时，它发送在脉冲1的消息。当进程的时钟读到$i(1 + \rho)(\delta_{max} + \delta + \gamma)$，进程执行脉冲$i$的状态变化，并随后向脉冲$(i + 1)$发送这些消息。当时钟增加到任意大的值时，用这个策略，每个正确进程执行无限多次的脉冲。 501

接下来要证明，在正确进程p执行那个脉冲的状态改变之前，要接收在脉冲i中（由正确进程）发送给它的所有消息。假设进程q在脉冲i向p发送消息。进程q在它的时钟读到时间$(i - 1)(1 + \rho)(\delta_{max} + \delta + \gamma)$，即，实时时间为$c_q[(i - 1)(1 + \rho)(\delta_{max} + \delta + \gamma)]$时，开始执行前一个脉冲的状态改变，并发送脉冲$i$的消息。关于局部处理时间界限的假设，蕴含着脉冲-$i$的消息最迟在实时时间

$$c_q[(i-1)(1+\rho)(\delta_{max} + \delta + \gamma)] + \gamma$$

被发送，这蕴含着（根据消息延迟界限）它最迟在实时时间

$$c_q[(i-1)(1+\rho)(\delta_{max} + \delta + \gamma)] + \gamma + \delta_{max}$$

被接收。进程p在局部时钟时间$i(1 + \rho)(\delta_{max} + \delta + \gamma)$开始执行状态改变，这意味着实时

时间为

$$c_p[i(1+\rho)(\delta_{\max}+\delta+\gamma)]$$

当p和q的时钟在时钟时间$(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)$是$\delta$-同步时，

$$c_q[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)] \leqslant c_p[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)]$$

p的时钟漂移是ρ有界的，这蕴含着

$$c_p[i(1+\rho)(\delta_{\max}+\delta+\gamma)] \geqslant c_p[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)]+(\delta_{\max}+\delta+\gamma)$$

组合这些方程，可得

$$c_p[i(1+\rho)(\delta_{\max}+\delta+\gamma)] \geqslant c_q[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)]+\gamma+\delta_{\max}$$

这蕴含着p在执行状态改变之前，接收消息。

习题

15.1节

502

15.1　在*Broadcast*(N, t)协议（图15-2和图15-3所示的算法）中，交换了多少条消息？

15.2　图15-4所示的算法中执行对0的判定中，正确进程发送的最大消息数是多少？针对两种情况做出回答，一种情况，将军是正确的；另一种情况，将军是有故障的。

15.2节

15.3　对于15.2.1小节中描述的Lamport、Shostak和Pease广播算法，其中交换了多少条消息？

15.4　给出15.2.1小节中描述的Dolev和Strong协议的一次执行，其中正确进程p和q以$W_p \neq W_q$结束。

15.5　证明当实现交互一致性时，可以解决值为1到N的保序重命名问题。

15.6　假设p用ElGamal签名模式（15.2.3小节），使用同一个值a，对消息M_1和M_2签名。说明如何从这两个签过名的消息，找出P的密钥。

15.7　设n是两个大素数的乘积，并假设给定计算平方根的黑盒子。即，给定二次剩余y，则盒子会输出x，且$x^2 = y$（方程模n）。证明如何用盒子进行对n的因子分解。

15.3节

15.8　证明如果时钟C_p和C_q的漂移是ρ-有界的，且在实时时间t是δ-同步的，那么在时钟时间$T = C_p(t)$，它们是$\delta(1+\rho)$-同步的。

15.9　如果多条消息的延迟接近$\delta_{\min}$，（即，甚至对于很小的ε，$F(\delta_{\min}+\varepsilon)$也会从0快速地离开）那么概率时钟同步协议是有效的。

如果多条消息的延迟接近$\delta_{\max}$（即，甚至对于很小的ε，$F(\delta_{\max}-\varepsilon)$也会离1足够远），给出一个能有效工作的“双重”协议。证明期望的消息复杂度为$2\cdot[1-F(\delta_{\max}-\varepsilon)]^{-2}$，并证明期望的运行时间界。

503
~
504

15.10　对于所有正确进程p，图15-6所示的算法中的集合A_p满足$width(A_p) < \delta$吗？

第16章 故障检测

在异步系统中（14.1节），一致性问题是不可解的，这导致了对问题的更弱表述和更强模型。前者的例子有弱协同和随机化，后者的例子包括引入同步。现在普遍认为故障检测器是增强计算模型的可选方法之一。

大多数分布式编程环境确实以某种方式提供时钟和计时器，这实际上引发了对同步模型的研究。理论研究揭示了什么任务需要利用这些原语，必须用到什么程度。有了故障检测器，情况是类似的：一旦企图与损毁进程通信，运行时间支持系统常常会返回错误消息。然而，这些错误消息并不总是绝对可靠。因此研究它们必须多么可靠，才能解决一致性问题（或者其他问题），是非常有用的。

与同步（利用物理时钟实现）形成对照，故障检测器没有直截了当的直观实现。这表明必须找到非平凡解，以便在运行系统中实现它们或者将其作为应用程序中的模块。实现它们时常依赖使用计时器（参见16.4节）。这已经导致故障检测器方法受到一些严厉批评。并认为这样的“故障检测器”并不解决任何问题，因为实现它们所要求的资源（时间），完全可被用来直接求解一致性问题。

可以肯定的一点是，过去半个世纪里，同样具有争议的结果实际上已应用到计算机科学中的每一项发明。如果程序必须被编译成机器语言才能执行，为什么还要用高级编程语言？我们何不直接用机器语言书写整个程序！为什么要利用设备驱动程序从应用程序中访问磁盘和打印机，如果这些驱动程序本身仅仅是程序而已？我们何不将驱动程序的代码写入应用程序中？这样没有驱动程序也能运行。当然，我们知道，利用高级语言和驱动程序的理由是：它们使得编程更简单，更易于理解和可移植，并且提高了研究某些概念的能力。我们也将这些理由用在故障检测中。 505

16.1 模型和定义

故障检测很久以前就被应用于许多实际系统，并在大约1991年，首先由Cornell大学Sam Toueg研究组表述为一种抽象机制。他们花了约5年时间，形成一篇基础性的论文并发表于[CT96]，我们的大部分定义和结果均由此而来。

故障检测器是一种模块，它为每个进程提供一种*可疑*（suspected）进程集。如果进程j在求值时刻受到怀疑，则程序中的测试$j \in D$返回。本章中所考虑的故障类型是损毁类型。不同进程中的模块无须一致（即，正确进程p可能怀疑r，而正确进程q不怀疑），在任何时刻，进程也不必是公平的（即，正确进程p可能怀疑正确进程r，也可能不怀疑故障进程r）。当然，为了能够用故障检测器推理程序，我们首先必须表达检测器的性质，特别值得注意的是检测器输出和实际故障之间的关系。

我们首先定义*故障模式*（failure pattern）来表示实际故障。为了避免使用时序逻辑表达式，在我们的表达式中，采用明确的时间变量t。这个时间变量t的值域为所有时间实例的集合T，例如，可以是自然数集。请注意，进程不知道时间t、故障模式F、损毁进程集合$Crash(F)$

和正确进程集合$Corr(F)$，或者故障检测器的历史！当进程q在时刻t访问它的故障检测器时，仅能“看见”值$H(q, t)$。

16.1.1 四种基本检测器类型

执行中出现的损毁可用故障模式模型化，它是一个函数F: $T \to \mathcal{P}(\mathbb{P})$。这里，$F(t)$是在时刻$t$损毁的进程的集合，因为我们假设不能重新开始，$t_1 < t_2$蕴含着$F(t_1) \subseteq F(t_2)$。用$Crash(F) = \cup_{t \in T} F(t)$和$Corr(F) = \mathbb{P} \setminus Crash(F)$分别表示损毁进程集合和正确进程集合。

506

怀疑因时间和进程的不同而有所差异。用函数H: $\mathbb{P} \times T \to \mathcal{P}(\mathbb{P})$模型化。这里$H(q, t)$是$q$在时刻$t$怀疑的进程的集合。

为使故障检测器成为非确定性的（即，已知一种故障模式，可能产生不同响应），我们将检测器$\mathcal{D}$模型化为一种映射，将故障模式映射到故障检测器历史的集合。即，对于故障模式F，元素$H \in \mathcal{D}(F)$是故障检测器的历史。

历史性质

可以理解的是，为使故障检测器可用，在它的输出（故障检测器历史）与输入（故障模式）之间必定存在一种关系。要求总是具有两种类型，即完全性（检测器将怀疑损毁进程）和精确性（检测器将不怀疑正确进程）。完全性限制了受到怀疑的进程集合的下界，精确性限制了受到怀疑的进程集合的上界，但是，两种情形中的任何一种都不能精确匹配所要求的损毁进程集（实际构想的系统永远不会满足这样的要求）。

我们只考虑一种形式完全性。

定义16.1 如果每个损毁进程最终受到各正确进程的怀疑，则称故障检测器$\mathcal{D}$是完全的：

$$\forall F: \forall H \in \mathcal{D}(F): \exists t: \forall p \in Crash(F): \forall q \in Corr(F): \forall t' > t: p \in H(q,t')$$

（在文献中，这个性质也称为强完全性。）

我们考虑四种不同形式的精确性。记住精确性规定，正确进程是不受怀疑的。在强完全性形式中，上述规定对所有正确进程成立；在弱完全性形式中，上述规定至少对一个正确进程成立。对于“最终”形式，上述规定不需要一开始就成立，而是从某时刻起成立。

定义16.2 如果进程没有损毁，也没有进程曾经受到怀疑，则称故障检测器$\mathcal{D}$是强精确的（strongly accurate）：

$$\forall F: \forall H \in \mathcal{D}(F): \forall t: \forall p,q \notin F(t): p \notin H(q,t)$$

如果存在一个正确进程从未受到怀疑，则称故障检测器$\mathcal{D}$是弱精确的（weakly accurate）：

$$\forall F: \forall H \in \mathcal{D}(F): \exists p \in Corr(F): \forall t: \forall q \notin F(t): p \notin H(q,t)$$

507

如果存在一个时刻，其后，没有正确进程受到怀疑，则称故障检测器$\mathcal{D}$是最终强精确的（eventually strongly accurate）：

$$\forall F: \forall H \in \mathcal{D}(F): \exists t: \forall t' > t: \forall p,q \in Corr(F): p \notin H(q,t')$$

如果存在一个时刻，其后，有一个正确进程不受怀疑，则称故障检测器$\mathcal{D}$是最终弱精确的（eventually weakly accurate）：

$$\forall F: \forall H \in \mathcal{D}(F): \exists t: \exists p \in Corr(F): \forall t' > t: \forall q \in Corr(F): p \notin H(q,t')$$

可知，定义仅仅限制非损毁进程q的怀疑。与损毁进程的怀疑无关。进而，强精确性定义明确地排除了损毁进程在损毁之前受到怀疑。对于其他三种形式的精确性，不包含这一点也没有任何关系。

非常容易获得一个完全的（检测器怀疑每个进程：$H(q, t) = \mathbb{P}$）故障检测器，同时，也非常容易得到一个精确的（检测器不怀疑任何进程：$H(q, t) = \varnothing$）故障检测器。有趣又有用的检测器总是把完全性和精确性结合起来。下面考虑四种类型。

定义16.3 如果故障检测器是完全且强精确的，则称它是**完美的**（perfect）；用$\mathcal{P}$表示理想检测器类。

如果故障检测器是完全且弱精确的，则称它是**强的**（strong）；用$\mathcal{S}$表示强检测器类。

如果故障检测器是完全且最终强精确的，则称它是**最终完美的**（eventually perfect）；用$\Diamond\mathcal{P}$表示最终理想检测器类。

如果故障检测器是完全且最终弱精确的，则称它是**最终强的**（eventually strong）；用$\Diamond\mathcal{S}$表示最终强检测器类。

16.1.2 故障检测器的用途和缺陷

不难想象在设计分布式应用程序时如何使用故障检测器。

回忆一下我们在异步分布式算法（第14章）中经常遇到的通信操作：

（1）每个节点进行一次向所有近邻发送消息（shout）的操作，即，向每个节点发送一条消息。

（2）每个节点收集所发出的$N - t$条消息。

进程永远不会等待多于$N - t$条消息的到达，因为在损毁情况下，这样做会有死锁的风险。构造不会引起永久阻塞（参见引理14.30），因为至少$N - t$个进程是活的。缺陷是即使没有损毁，收集的消息集也由于进程的不同而不同。收集的消息集大小总是为（$N - t$）。如果没有检测器，绝对禁止等待来自某个特定进程的消息的到达，因为如果希望的某个发送进程已经损毁，它可能导致阻塞。 508

利用故障检测器进行通信的标准方式是：

（1）每个节点向各节点发送一条消息。

（2）对于每个进程q，每个节点等待，直到来自q的消息到达，或者q受到怀疑。

故障检测器的完全性使得这种构造免于永久阻塞。事实上，由于已经损毁，不发送消息的每个进程最终受到怀疑。正如前面的构造，这样做有一个缺陷：存在各种情况，会导致由不同正确进程收集的消息所组成的不同集合。

（1）进程q损毁，但是在它损毁之前，发送了部分或者全部所请求的消息。进程p_1在怀疑q之前，接收这些消息，而进程p_2没有接收到这些消息，在结束它的接收阶段时没有q的消息。

（2）进程q是正确的，并且发送消息。但是进程p_1怀疑q，且未收集q的消息。而p_2不怀疑q，并且收集q的消息。

因此，所收集的消息的集合，可能包括来自损毁进程的消息，以及正确进程错过的消息。集合大小可能不同，即使损毁数存在界t，由于错误的怀疑，收集的消息数也可能少于$N-t$。只有限制受到怀疑的进程数，才能保证这个集合中消息数的下界。这主要是通过弱精确检测器完成的。

有了故障检测器，就有可能接收来自特别提到过的进程的消息：当产生对进程的怀疑时，

就会中断对这个消息的等待。由此可见，用这种方法，从一个正确进程到另一个正确进程的消息也可能不能被接收。

所勾画出的消息接收模式是如此普通，以至于我们为其引入编程简写方式：语句“collect <**message**, *par*> from *q*”。指令等待，直到接收*q*的消息<**message**, *par*>，或者*q*受到怀疑。
509 返回表明接收成功的布尔值。

```
x_i := input ;
for r := 1 to N
    do begin if i = r
                then forall j do send ⟨ value, x_i, r ⟩ to j ;
             if collect ⟨ value, x', r ⟩ from p_r
                then x_i := x'
       end ;
decide x_i
```

图16-1　旋转协调器算法，利用强故障检测器 $\mathcal{S}$（对于p_i）

16.2　用弱精确检测器解一致性问题

本节给出一种利用故障检测器的相对简单的一致算法。正确性规范是：

（1）**终止性**　每个正确进程判定一次。

（2）**一致性**　所有判定（在一次执行中）是相同的。

（3）**有效性**　一个判定至少等于一个进程的输入。

易证明，有效性蕴含着非-平凡性。因此，证明，如果没有故障检测我们会得到一个不可能的结果。（定理14.8）

在图16-1所示的算法中，利用弱精确故障检测器解决了这个问题。存在一个进程，从未受到怀疑，但是进程却不知道是哪一个进程！只有当某些进程收集了这个未知进程的一条消息后，才能保证它们全部接收这条消息，因此，也才能保证它们全部接收相同信息。在算法中，通过给每个进程一个机会，避免由于不知道哪个节点未受怀疑所引发的困难。称这种方法为*旋转协调器*（rotating coordinator）模式。算法由*N*轮循环组成，每个进程与它们中的一个进程协调。

我们称进程为p_1到p_N，且p_j协调第*j*轮。在一轮中，协调器向所有近邻发送它的值，所有节点收集这个协调器的消息。如果某个节点成功，则用从协调器接收的值替代它的值。在此只对正确性证明进行概述，因为详细情况是相当明显的。

定理16.4　*图16-1所示的算法求解一致性问题。*

证明。因为在一轮中，没有一个正确进程永远受到阻塞，算法满足终止性。协调器是正确的，并且发送消息，或者最终受到怀疑（由于检测器的完全性）。我们可以对*i*用归纳法证
510 明，每个正确进程可达第*i*轮循环。

因为进程只能保存它的值，或者用接收协调器的值来替代它，有效性得到满足，这使我们可以证明（再次用归纳法），各进程进入每一轮时，带有一个值，该值是某些进程输入。

为了证明一致性，设p_j是从不受到怀疑的进程。在第*j*轮中，每个进程（完成该轮循环的执行）接收来自p_j的值，因此，所有进程完成*j*轮时，具有相同值。以一个值进入那轮，完成时也要有一个值（相同的），这蕴含着，整个算法完成时，确实只有同一个值。　□

观察可知，在程序中，并没有出现弹性t。实际上，弹性为$N-1$。如果弹性受到限制，目前还不知道如何减少轮数。原因是正确协调器可能受到怀疑。在本算法中，如果进程是正确的，并没有多大的帮助，所需要的是它们不受怀疑。如果增强检测器的精确度，可以减少的轮数。参见习题16.5。

16.3 最终弱精确检测器

在前一节中，对弱精确性所做的假设，蕴含着在算法的前N轮中，存在一轮具有不受怀疑的协调器。因此，可以设计算法在N轮后终止。然而，最终弱精确性，尽管蕴含在将来某一时刻，会出现一轮，其中有不受怀疑的协调器，但它并未提供关于这是那一轮的一个界定范围。

因此，除了实际上所达到的一致性，解中还必须检测（detect）何时达到一致性。除了故障检测器之外，这个检测要求，存在界限为$t < N/2$的弹性。这个结果将在16.3.1小节证明。我们在16.3.2节提出的算法并没有提供最优弹性，而是假设弹性为$t < N/3$。

16.3.1 弹性上界

在任意长（但有限）的初始阶段，具有最终精确性质的故障检测器可能展示任意的行为。这意味着在防止有限次错误执行方面，它们是无用的。按照不可能性的证明：通过构造有限次错误执行所得的结果，可以继续应用到最终精确故障检测器的模型上。例如，定理14.16就 511
是这种情形。这里，我们将定理扩展到具有故障检测的模型上。定理是为最终完美检测器而制定的。其结果对最终强检测器也成立，因为这类检测器满足弱规范。

定理16.5 *不存在使用允许$t \geqslant N/2$次损毁的$\Diamond\mathcal{P}$的一致算法。*

证明。反证法。假设存在这样的算法。在$\mathbb{P}$中，可以构成两个大小都为$N-t$的不相交进程集S和T。考虑两种情形。

（1）在情形0中，S中的进程输入都为0，那些在S外面的进程立刻发生故障，这种情形一开始，就被S中的所有进程检测到。因为S中包含$N-t$个进程，S中的进程在有限时间内判定。又由于所有活动进程输入为0，因而它们的判定为0。

（2）在情形1中，T中的进程输入都为1，那些在T外面的进程立即发生故障，这种情形一开始，就被T中的所有进程检测到。因为T中包含$N-t$个进程，T中的进程在有限时间内判定。又由于所有活动进程输入为1，因而它们的判定为1。

由两种情形可知，故障检测器的行为与对$\Diamond\mathcal{P}$的要求一致。我们并没有详述，在情形0（或情形1）中为什么对0（或1）的判定是不可避免的。

因为S和T是不相交的，可以将这两种情形组合成一种新的情形，称它为情形2。假设只有$S \cup T$外面的进程发生故障，S中的进程输入都为0，T中的进程输入都为1。$S \cup T$外面的进程故障很快地被那些S和T中的进程检测出。而且，S中的进程从一开始就错误地怀疑那些T中的进程，而T中的进程从一开始就错误地怀疑那些S中的进程。最终完美性要求使得这种情形持续任意有限的时间。从S到T所发送的消息传输非常慢，反之亦然。

对于S中的进程情形2就如同情形0，而对于T中的进程情形2就像情形1，因此这些进程开始执行一系列相同步。在有限（finite time）的时间内，这些进程对同一个值判定，就如同之前所述的情形，即，S和T中的进程对0和1判定。在每组中至少发生一次判定之后，错误的怀

疑停止，从S向T发送的消息到达，反之亦然。

512 因此我们构造一种情形，其中故障进程数少于t个。故障检测器按照最终完美性的定义来运行，从正确进程到正确进程的所有消息都到达。然而，不同进程进行不同的判定，这表明一致性得不到满足。 □

一致性根本不可能性的证明（定理14.8），构造了一个无限不判定执行，在下一节将会看到，通过最终正确检测器可以避免这种无限行为。

16.3.2 一致算法

图16-1所示的算法从一开始就假设故障检测器是弱精确的。这蕴含着，存在不受怀疑的协调器，因此在前N轮的循环内，达到了值的一致性。在图16-2所示的算法中，协调器必须扩展它的活动，来发现是否一致性已经出现。它首先收集所有进程的当前值，并检查它们是否是一致的（参见步骤2）。这里有一个缺陷，因为由于错误的怀疑，协调器可能只有效地收集不到一半的活动值，然后就宣布一个值作为最终结果，而该结果实际上只得到少数进程的支持。这里，利用对故障进程数所做的限制：在收集阶段中，协调器并不使用它的检测器，而是像在经典方法中那样，等待$N-t$个进程的投票。图16-2所示的算法的健壮性被限定为$t < N/3$。现在，协调器所观测到的一致性至少蕴含着，绝大多数进程已经计算这个值，不必所有正确进程都这样做，由于协调器仅仅等待固定的票数，有些不一致的投票可能漏掉。

如果协调器宣布，它从一致的投票中选择值v，正如在宣布第r轮结果的消息<**outcome**, d, v, r>中位d所表明的那样，则允许判定。判定之后，进程继续它们的活动，并帮助其他进程进行判定。

引理16.6 *对于每轮循环r和正确进程p，p完成第r轮执行。*

*证明。*假设所有正确进程开始执行第r轮。正确协调器将完成步骤1，这是因为发送了足够多的投票。正确进程将完成步骤3，因为如果它是正确的，协调器将发送消息；如果它不是正确的，
513 最终将会受到怀疑。因此所有正确进程完成整个一轮的循环。可用归纳法完成整个证明。 □

我们现在证明大到足以导致判定的绝对多数是不变的。

引理16.7 *如果在第$k > N - t$轮开始，进程有值v，那么在该论的结束，至少有k个进程有值v。*

*证明。*进程可以怀疑协调器，并保持同一个值。改变值的惟一方法是接收具有该值的协调器的消息。协调器取出所接收的$N-t$个值中的大多数。至多有t个进程的值不同于v，因此，至少接收$N-2t$张对v的投票，且$N-2t > t$。这表明如果协调器在步骤2发送一值，它发送了值v，并且具有值v的进程数只能增加，不会下降。 □

引理16.8 *每个正确进程进行判定。*

*证明。*最终精确性蕴含着，最终存在某一轮r，它的协调器c在第r轮或者其后轮中不受怀疑。在那一轮中，所有正确进程接收协调器的消息，并对同一个值达成一致意见，并继续这样做（引理16.7）。因而，第r轮后，协调器所发送的全部消息，宣布v是一致的（$d = true$），因此只要进程在步骤3中收集一条消息，就开始对v进行判定。但c不再受到怀疑，在第$r + N$轮中，再次成为协调器，于是在该轮结束时，所有进程已经进行判定。 □

定理16.9 *图16-2所示的算法是具有$\Diamond S$的一致算法。*

证明。对r用归纳法相当容易证明（证明留给读者），如果进程持有第r轮中的一个值，这个值在输入中。这就证明了有效性。引理16.7蕴含着一致性，因为从一个判定轮可知，对其他值的判定不存在足够的支持。最后，在前述的引理中表达了终止性。 □

```
x_i := input ;
r := 0 ;
while true do
      begin (* Start new round and compute coordinator*)
            r := r + 1 ; c := (r mod N) + 1 ;
            (* Phase 1: all processes send value to coordinator *)
            send ⟨value, x_i, r⟩ to p_c ;
            (* Phase 2: coordinator evaluates outcome *)
            if i = c then
              begin wait until N − t mesgs. ⟨value, v_j, r⟩ have been received;
                    v := majority of received values;
                    d := (∀j : v_j = v) ;      (* range over received messages *)
                    forall j do send ⟨outcome, d, v, r⟩ to p_j
              end ;
            (* Phase 3: evaluate the round *)
            if collect ⟨outcome, d, v, r⟩ from p_c then
              begin x_i := v ;
                    if (d ∧ (y_i = @))
                      then decide(v)
              end
      end
```

图16-2 利用$\Diamond S$的旋转协调器算法

16.4 故障检测器的实现

故障检测器方法的明确优势是，只须处理检测器的性质，而不管它如何实现。然而，我们要给出一些可能的实现，来证明其后发生了什么，同时也是对定义的一种说明。在大多数情况下，使用定时器后，检测器只是异步接口的一种形式。

16.4.1 同步系统：完美检测

完美故障检测图16-3所示的算法假设通信延迟具有上界μ。活动进程以时间间隔σ发送消息<alive>，如果在$\sigma+\mu$时间还未接收到一个进程的消息，表明进程已经损毁。

16.4.2 部分同步系统：最终完美检测

如果消息延迟上界未知，或者由于某种原因，不愿将其包括在程序中（例如，因为可移植性）。具有稍微弱性质的检测器也是可能的。检测器（图16-4所示的算法）从一个较小的估算值μ开始。也可能发生早停，如，进程超时或受到怀疑，而实际上来自进程的消息<alive>仍然在传输中。如果从受到怀疑的进程发来的消息到达，就立即将它从D中删除，但另一方面，μ的估算值会增加。经过固定次数这样的修正之后，估算值超过了μ的实际值，此后，不会出现更多的错误怀疑。由于错误怀疑数有限，它们都会出现在执行的一个有限初始阶段，这证明检测器是最终强精确的。

var D_p set of processes ;
$r_p[\mathbb{P}]$ A timer for each process

Initialization of the detector:
$D_p := \varnothing$;
forall q **do** $r_p[q] := \sigma + \mu$

The sending process, at each multiple of σ:
forall q **do** send $\langle$ **alive** $\rangle$ to q

The receiving process, upon receipt of $\langle$ **alive** $\rangle$ from q:
$r_p[q] := \sigma + \mu$

Failure detection, when $r_p[q]$ elapses:
$D_p := D_p \cup \{q\}$

图16-3 完美故障检测器算法

var D_p set of processes ;
$r_p[\mathbb{P}]$ A timer for each process ;
me_p μ estimate, initially 1

Initialization of the detector:
$D_p := \varnothing$;
forall q **do** $r_p[q] := \sigma + me_p$

The sending process, at each multiple of σ:
forall q **do** send $\langle$ **alive** $\rangle$ to q

The receiving process, upon receipt of $\langle$ **alive** $\rangle$ from q:
if $j \in D_p$ **then**
begin $D_p := D_p \setminus \{q\}$;
$me_p := me_p + 1$
end ;
$r_p[q] := \sigma + me_p$

Failure detection, when $r[q]$ elapses:
$D_p := D_p \cup \{q\}$

图16-4 最终完美故障检测器

16.4.3 小结

本章通过给出一些易于理解的基本结果，引入了故障检测器的概念，并在课堂上加以介绍。由于这个研究课题相对较新，并且引起了国际研究界的注意。因此不可能只选出一些结果，就能表明未来的研究和发展方向。我们通过一些关于故障检测器结果的例子来结束本章。

按照故障检测器模拟的概念[CT96]，建立故障检测器的分级结构。如果存在利用 $\mathcal{A}$ 提供的信息并实现了检测器 $\mathcal{B}$ 的分布式算法，检测器 $\mathcal{A}$ 会模拟检测器 $\mathcal{B}$。在这种情况下，称检测器 $\mathcal{B}$ 比检测器 $\mathcal{A}$ 弱。

定义一种更弱形式的完全性：如果每个损毁进程最终永远受到至少一个正确进程的怀疑，则称检测器是弱完全的[CT96]。然而，容易证明，弱完全的检测器可以模拟强完全的检测器

(参见定义16.1)，而不减弱定义16.2中的精确性性质。因此，在文献中（或者在本书中），常常不考虑弱完全性。

还有一个问题就是，能否用一个比◇$\mathcal{S}$更弱的检测器，来解一致性问题（如16.3节所做的）。Chandra等人[CHT96]证明，任何一个允许实现一致性的检测器可以模拟◇$\mathcal{S}$。这个工作产生了许多围绕故障检测的数学机器和证明技术。

并不是所有故障检测器在实现过程中都用到超时：心跳（heartbeat）检测器[ACT97]只交换异步消息。然而，这个检测器需要与本章用到的进程进行各种交互，用它所解的问题需要稍微做一些设计。一些有关问题，如能异步地实现什么，异步实现的检测器能够解决什么问题等等，它们太复杂太新颖，在此不进行讨论。 514 ~ 517

习题

16.1节

16.1 第509页列出了正确进程在接收阶段，收集不同消息集的各种可能性。对于四种类型的检测器（定义16.3）的每一种，讨论是否这种可能性能够出现。

16.2 设F是故障模式，证明存在t满足$F(t) = Crash(F)$。

在什么地方，你的证明用到了$\mathbb{P}$是有限的这一条件？证明如果$\mathbb{P}$是无限进程集，结论不成立。

16.3 强精确性要求比不存在正确进程曾经受到怀疑这一要求**更强**：

$$\forall F : \forall H \in \mathcal{D}(F) : \forall t : \forall p \in Corr(F), q \notin F(t) : p \notin H(q,t)$$

给出一个满足这个性质而在强精确检测器中不允许的故障模式和故障检测器历史的例子，不满足。

16.4 进程p先向q发送消息，然后向r发送。q和r收集p的消息。用一个例子证明，即使检测是完美的，也可能出现r接收到消息，q没有接收到消息的情形。在你的例子中，q和r必须是正确的。

16.2节

16.5 如果在每一轮中，至少有l个正确进程不会受到怀疑，则称故障检测器是l-精确的。(这个性质蕴含着，弹性的界限为$N-l$，弱精确是1-精确的。)

修改图16-1所示的算法，使它在$N-l+1$轮中完成。

16.6 把图16-2所示的算法修改成图14-5所示的算法，以便获得一个有限算法，或者用最终精确检测器证明这是不可能的。

16.3节

16.7 给出图16-2所示的算法执行的一个例子，其中判定在不同轮中出现。 518

16.4节

16.8 在图16-3所示的算法的一次损毁和它的检测之间经过多长时间？

16.9 证明图16-4所示的算法的最终完美性。在每次执行中，最终都满足关系$me_p \geq \mu$吗？ 519

第17章 稳 定 性

本章所考虑的稳定算法所实现的容错行为，完全不同于前几章中所研究的健壮算法的容错行为。健壮算法沿用一种悲观方法，怀疑所有接收的信息，并通过足够的检查，来进行所有步骤以保证正确进程的全部步骤的有效性。在出现故障进程时，必须保证有效性。这就需要限制故障和故障模型的数目。

稳定算法是乐观算法，它可能引起正确进程表现不一致，但能保证在所有故障行为停止后的有限时间内，返回到正确行为。即，稳定算法能保护免于瞬时（transient）故障，假设故障最终能被修复。这种假设使得我们可以放弃故障模型，以及对故障数量的限制。不考虑要出错的进程，假设所有进程运行正确，但在瞬时故障期间，配置可能遭到破坏。忽略故障期间计算的历史，把算法分析开始的配置作为（正确运行）算法的初始配置。因此，如果算法最终开始正确执行（即，按照算法的规范），而不管它的初始配置，则称算法是稳定的。

Dijkstra[Dij74]首先提出了稳定性概念，但是直到20世纪80年代后期，才开始有较多这方面的研究。因此，这个学科相对较新。然而，在此后的若干年直至本书面世时，出现了大量稳定算法和相关结果。本章给出了经过选择的一些研究成果。 520

术语“稳定”贯穿始终，而在文献中常会碰到“自稳定”这一术语。

17.1 引言

17.1.1 定义

可以把稳定算法模型化为没有初始配置的转移系统（将此定义与定义2.1进行比较）。

定义17.1 *系统是序偶$S=(C, \rightarrow)$，其中C是配置集，$\rightarrow$是C上的二元转移关系。S的一次执行是一个最大序列$E=(\gamma_0, \gamma_1, \gamma_2, \cdots)$，满足对于所有$i \geqslant 0$，$\gamma_i \rightarrow \gamma_{i+1}$。*

与定义2.2不同，计算的每个（非空）后缀现在也是一次计算。算法的正确性，即，所期望的“进程一致行为”，表示为规范，它通常是配置序列上的一个谓词（通常用P表示）。

定义17.2 *系统S稳定到规范P，如果存在合法配置的一个子集$\mathcal{L} \subseteq C$，具有以下性质。*

（1）**正确性** *从$\mathcal{L}$中一个配置开始的每次执行满足P。*

（2）**收敛性** *每次执行包含$\mathcal{L}$中的一个配置。*

合法配置集通常是闭的，即，如果$\gamma \in \mathcal{L}$且$\gamma \rightarrow \delta$，那么$\delta \in \mathcal{L}$。但由于这不用于以下结果的证明中，因此在定义中没有包括闭性。

定理17.3 *如果系统S稳定到规范P，那么S的每次执行，都有一个满足P的非空后缀。*

证明。由收敛性质，每次执行包含一个合法配置；由正确性，在这个合法配置开始的一个后缀满足P。 □

1. 证明稳定性

合法配置的使用可使我们利用标准的验证技术，证明算法的稳定性。用范函数证明收敛性。

引理17.4　假设

521　（1）所有终止配置属于$\mathcal{L}$；

（2）存在函数f: $\mathcal{C} \to W$，其中W是良基集，且对于每次转移$\gamma \to \delta$，$f(\gamma) > f(\delta)$或者$\delta \in \mathcal{L}$成立。

那么$S = (\mathcal{C}, \to)$满足收敛性。

为了证明正确性，人们可以进行经典算法分析，把$\mathcal{L}$看作初始配置集。实际上，只考虑在$\mathcal{L}$中开始的执行。

2. 稳定算法的性质

较之经典算法，稳定算法具有以下三个基本优势。

（1）*容错*　正如在本章引言中所看到的那样，稳定算法为所有瞬时的进程故障，提供了完全和自动的保护。因为算法可从任何配置中恢复，而不管有多少数据由于故障遭受破坏。

（2）*初始化*　取消了对算法初始化的适合性和一致性要求，因为进程可在任意状态开始执行，并且最终协调行为得到保证。

（3）*动态拓扑结构*　在拓扑结构发生变化后，计算依赖于拓扑结构的函数（路由表、生成树）的稳定算法收敛于新解。

第四个优点是，在17.3.1小节中讨论了“顺序”组合的可能性，不需要终止检测。最后对于同一网络问题，研究表明，到目前为止的许多稳定算法都比相应的经典算法简单。然而，部分原因是由于这些算法的复杂度没有进行优化，因此，当对稳定算法进一步研究时，这个“优势”可能会消失。

另一方面，稳定算法有以下三个缺点。

（1）*初始不一致性*　在达到合法配置之前，算法可能显示不一致的输出。

（2）*高复杂度*　到目前为止已知的稳定算法，通常要比针对同一问题的相应的经典算法具有更低的效率。

（3）*不能检测出稳定性*　从系统的内部，不可能知道合法配置已经达到。因此进程从来
522　不会意识到，什么时候它们的行为已经变得可靠。

3. 伪稳定性

定理17.3中所证明的稳定算法性质，可作为稳定性的另一种定义：只要求每次执行有一个满足规范的后缀。然而，这种表示并不等价于我们所做的定义。因此，Burns等人[BGM93]将它称作*伪稳定性*（pseudo-stabilization）。

为了证明这种差异，考虑具有配置a和b，且有$a \to a$，$a \to b$，$b \to b$的转移的系统S。规范P规定“所有配置是相同的”。因为S可从a转移到b至多一次，容易证明，每次执行都有一个由相同配置组成的后缀。配置a不能被选为合法的，因为在这个配置开始的执行$(a, b, b, \ldots)$并不满足P。因此，S-执行$(a, a, a, \ldots)$（尽管满足P自身）并不包含合法配置。

Burns等人证明，弱伪稳定性要求，可以使没有稳定性解的问题有解。一个例子就是数据序列传输。另一方面，在伪稳定解中（没有稳定解），在满足规范P之前，系统可执行的步数没有上界，而对于稳定算法，却可以给出这样的上界。本章中，我们只对稳定算法进行研究。

17.1.2　稳定系统中的通信

在本书前面的大多数章节中，假设了一个通过消息传递进行通信的异步模型，但是这个

模型对于研究稳定算法并不令人满意。

首先考虑一个异步消息传递的算法，其中有一个进程p，对于该进程，每个状态是一个发送状态（即，p在每个局部状态可以发送消息）。系统中有一个仅由进程p的发送行为组成的执行，而其他所有进程在它们的初始状态保持冻结，并且这种行为并不满足任何有意义的非平凡规范。

其次，考虑一个算法，其中对于每个进程，存在一些状态，在这些状态进程并不发送消息（但可以只接收消息或者执行内部步骤）。如果一个配置中，每个进程都处于这种状态，且所有信道为空，则这个配置是终止配置，并一定满足规范。而且，所有这样的配置都不满足非平凡规范。

文献[Tel91a]中，给出了网络变更算法的稳定版本，每个进程中，利用计时器且消息传输时间有上界。在实际中，这个算法非常好。但是算法分析主要是针对定时技术细节方面。因 523
此，我们假设通过共享变量进行通信的更通用的模型，在这种模型中，一个进程可以对一个变量进行写操作，而另一些进程可对同一变量进行读操作。

在状态模型中，进程可以（自动地）读近邻的整个状态。在这样的模型中，p的每个近邻读取同一个状态，因此，进程不可能将不同信息传输到它的各个近邻。在*链接-寄存器*（link-register）模型中，进程间的通信是通过两个进程间共享的两个寄存器实现的。每个进程可从这两个寄存器之一读取值而向另一个写入值。通过在它的链接寄存器写不同的值，进程可以向各个近邻传输不同的信息。

我们进一步区分*读一个*（read-one）模型和*读所有*（read-all）模型。在读一个模型中，进程在一个原子步只能读一个近邻的状态（或链接寄存器）；在读所有模型中，进程在一个原子步可以读所有近邻状态。

17.1.3 例子：Dijkstra令牌环

Dijkstra[Dij74]首次提出了稳定算法。这些算法在一个进程环中实现*互斥*（mutual exclusion）。对于这个问题，假设进程有时必须执行代码的*临界区*（critical section），但要求在任何给定时间，至多只有一个进程执行临界区。每个进程被给定一个局部谓词，当该谓词为真时，表明进程有*特权*（privilege）执行临界区。则问题的规范为：

（1）在每个配置中，至多一个进程有此特权。

（2）每个进程可以无限次拥有特权。

我们可以引入一个令牌，在环上循环传递，并赋予持有令牌的进程特权，来解决这个问题。Dijkstra解决方法的优点是，可以在系统中没有令牌或者有多于一个令牌的情形中自动恢复（进程步为$O(N^2)$）。

1. 解的描述

进程i的状态表示为σ_i，是值域为$0,\ldots,K-1$中的一个整数。其中K是大于N的整数。进程i可以读σ_{i-1}（以及σ_i），进程0可以读σ_{N-1}。除进程0之外，所有进程都是平等的。如果$\sigma_i \neq \sigma_{i-1}$，则进程$i \neq 0$有特权；如果$\sigma_0 = \sigma_{N-1}$，则进程0有特权。具有特权的进程也能改变它的状态（假定它在完成临界区的操作之后，或者根本不再需要执行后者时，就可以改变状态）。 524

进程状态改变总是引起它的特权丢失。进程$i \neq 0$通过赋值$\sigma_i := \sigma_{i-1}$，将σ_i设为σ_{i-1}（当能够这样做时，即$\sigma_i \neq \sigma_{i-1}$）。进程0通过赋值$\sigma_0 := (\sigma_{N-1} + 1) \bmod K$，将$\sigma_0$设成不等于$\sigma_{N-1}$（当

前相等时)。

2. 稳定性证明

特权的定义蕴含着，在每个配置中，至少一个进程有特权（即，不存在终止配置)。如果没有除0以外的进程具有特权，那么，对于每个$i>0$，$\sigma_i=\sigma_{i-1}$。这蕴含着，$\sigma_0=\sigma_{N-1}$，且进程0具有特权。然而，没有步骤会增加特权进程的数目，因为改变状态的进程，同时失去了特权，在这一步中，惟一能够获得特权的进程是它的后继。

为了证明稳定性，定义L为配置集，且配置中只有一个进程具有特权。

引理17.5 *在合法配置中开始的执行满足P。*

证明。在一个合法配置中，只有一步是可能的，即，通过惟一具有特权的进程。在这一步中，该进程失去了它的特权，但是因为没有不带特权的配置，它的后继续必须获得特权。因此，特权在环上循环传递，在每N个连续配置中，每个进程拥有一次特权。 □

引理17.6 Dijkstra令牌环算法收敛到$\mathcal{L}$。

证明。由于不存在终止配置，系统的每次执行是无限的。

一次执行包含进程0的无限多步，因为如果没有进程0的一步，至多能发生$(1/2)N(N-1)$步。而进程i的一步去掉了i的特权，只可能将特权交给进程$i+1$，范函数为

$$F=\sum_{i\in S}(N-i)$$

其中$S=\{i:\ i>1$且进程i有特权$\}$，它是关于进程每一步的递减函数（进程0除外)。

在初始配置γ_0中，至多出现N个不同状态，因此，至少有$K-N$个状态没有出现在γ_0中。因为进程0在每一步中，状态（模K）增加，至多（进程0的）N步之后，进程0达到一个未在γ_0中出现的状态。除进程0之外的所有进程复制状态，因此进程0首次计算这样的状态，它的状态
525 在环中是惟一的。由于进程0的状态是惟一的，因此在配置满足$\sigma_1=\sigma_2=\cdots=\sigma_{N-1}=\sigma_0$之前，它不会再次得到特权，这描述了一个合法配置。

由于系统在进程0的第$(N+1)$步之前达到一个合法配置，并且其他进程(至多)出现$(1/2)N(N-1)$个连续步，因此，(至多)经过$(N+1)1/2N(N-1)+N=O(N^3)$步，达到合法配置。

利用更精确的分析，可以证明步数界限为$O(N^2)$，参见习题17.2。 □

推论17.7 Dijkstra令牌环算法稳定到互斥。

3. 一致解

称稳定算法是*一致的*，如果所有进程相同，且没有标识区别它们。找互斥以及其他问题的*一致解*近年来备受关注，这种兴趣主要是由于它们在理论上的重要性所致。然而，在实际中，一致解也具有吸引力，因为进程标识像其他进程变量一样，通常存储在存储器中，因此一个毁坏的配置可能侵犯所要求的标识惟一性。

一致稳定算法也是同一问题的一个匿名网络算法，因此不可解的匿名网络（第9章）问题不存在一致稳定解。通过假设对称配置，并证明可以不确定地保持对称性，Dijkstra[Dij82]证明，如果环规模不是素数，则不存在一致（确定的）令牌环。假设已知环规模为素数，Burns和Pachl[BP88]提出了一致令牌环算法。

可用类似方法证明，选举问题、生成树和其他一些破坏对称的问题，不可能存在确定一致稳定算法，但是如果可以利用随机化，就可能存在确定一致稳定解。由于本章关注的是稳

定性，而不关注匿名，我们在后续章节中，假设进程具有惟一名字，或领导人可用。

17.2 图论算法

如果算法的目的是达到后置条件ψ，可将它的规范设为“每个配置满足ψ”。我们称S随着ψ稳定，如果存在集合$\mathcal{L}$，满足 526

（1）$\mathcal{L}$是闭的，并且$\gamma \in \mathcal{L}$蕴含$\psi(\gamma)$；且

（2）每次计算达到$\mathcal{L}$的一个配置。

算法可能终止，但并不总是出现这种情形。17.2.1节给出一种环定向算法，它最终可以经历不同（但是有向的）配置的无限序列。17.2.2小节给出找最大匹配问题的一个算法，其中的合法配置是终止的。

17.2.1 环定向

在环定向问题中，我们考虑N个进程的无向环，其中每个进程将它的一个链路标以*succ*（后继），另一个标以*pred*（前驱）。（如果两个标号是相同的，进程立即改变其中一个标号）。称链路pq上进程p的标号为l_{pq}，在初始时，不做标号全局一致性的假设。定向算法的目标是计算环上一致的方向侦听。

定向问题要求系统达到后置条件ψ，该条件定义为“对于每条边pq，l_{pq}为*succ*当且仅当l_{qp}为*pred*”。Israeli和Jalfon[IJ90]证明，在读状态模型中，可能不存在确定解。我们给出了链接寄存器模型的Israeli-Jalfon算法。所给出的算法是一致算法。

var s_p : (S, R, I) ;

(1) $\{state_p = I \wedge s_q = S \wedge l_{qp} = succ\}$
$s_p := R$; **if** $l_{pq} = succ$ **then** *flip*

(2) $\{s_p = S \wedge l_{pq} = succ \wedge s_q = R \wedge l_{qp} = pred\}$
$s_p := I$

(3) $\{s_p = R \wedge l_{pq} = pred \wedge \neg(s_q = S \wedge l_{qp} = succ)\}$
$s_p := S$

(4) $\{s_p = s_q = S \wedge l_{pq} = l_{qp} = succ\}$
$s_p := R$; *flip*

(5) $\{s_p = s_q = I \wedge l_{pq} = l_{qp} = succ\}$
$s_p := S$

procedure *flip*: reverse *succ* and *pred* for p

图17-1 环定向算法

如图17-1所示的算法在算法中，除了利用保存*pred*或*succ*的链接寄存器之外，还利用变量s，其值为S、R和I。在一步中，进程p考虑它的状态、近邻q和链接寄存器的状态，如果这5个阈值中有一个值为*true*，则进程改变状态。进程循环传递令牌，每个进程将它的后继设置到最后转发的令牌的方向；如果两个令牌相遇，其中一个就要被删除。最终，所有剩下的令牌将在同一个方向遍历，这使得令牌被定向，剩余令牌继续传递。

在S状态，进程等待转发一个令牌（向当前它的后继）；而在R状态，进程等待接收一个令牌；在I状态，进程处于空闲状态。如果满足下述条件之一，则进程持有令牌，这些条件是：进程的状态在S中，或者在R中，且它的前驱不在状态S中。在算法的第（2）步中，令牌从p向q移动；在算法的第（4）或（5）步中，令牌在p中分别遭受破坏或者被创建。这些步称为令牌步（token step）。而算法的（1）和（3）步，对令牌的位置不产生影响，称这些步为无声步（silent step）。在执行第（1）步之后，同一进程的下一步是第（3）步，在第（3）步后，下一步是第（4）步或者第（2）步，这限定了无声步数与令牌步数成线性关系。

在行为（1）中，如果q想要将令牌转发给空闲进程p，p同意接收令牌（尽管它还没有移动），并且使q成为它的前驱。在行为（2）中，p观察到，它的后继同意接收令牌，它想要传输，并变成空闲，于是移动令牌。在行为（3）中，p接收来自q的令牌，并观察到，q变成空闲，则开始向它的后继传输令牌。最后两种行为关注对称情形。在（4）中，在不同方向遍历的两个令牌相遇，在（5）中，两个空闲进程定向不一致。在这些情形中，两个进程都被激活，第一个进程进行一步，打破了对称性[⊖]。在行为（4）中，进程p接收q的令牌，隐含地破坏了它自己的状态。在行为（5）中，p产生一个令牌，它将决定q的方向。

每个行为的效果如图17-2所示，其中每个进程的状态和定向可用其后继方向上的箭头表示。

行为号	q	p	p		
(1)	$\vec{S}$	I	→	$\vec{R}$	无论p的初始方向如何，行为（1）都是可应用的。
(2)	$\vec{R}$	$\vec{S}$	→	$\vec{I}$	
(3)	$\neg\vec{S}$	$\vec{R}$	→	$\vec{S}$	
(4)	$\vec{S}$	$\vec{S}$	→	$\vec{R}$	
(5)	$\vec{I}$	$\vec{I}$	→	$\vec{S}$	

图17-2　算法17-1中行为的效果

称配置是合法的，当且仅当它是定向的，即，所有箭头指向同一个方向。

引理17.8　*$\mathcal{L}$是闭的，并且$\gamma\in\mathcal{L}$蕴含$\psi(\gamma)$。*

证明。由$\mathcal{L}$的选择，合法配置被定向，这蕴含着引理的第二部分。此外，只有类型（1）、（2）和（3）中的步骤会出现。因为当存在不同定向的近邻时，步骤（4）和（5）才被激活。步骤（2）和（3）从不会翻转进程的定向。如果p与q定向在同一方向，（1）也不会改变进程的方向。这就证明了$\mathcal{L}$的闭性。　□

命题17.9　*终止配置是合法的。*

证明。设γ是终止的，γ中没有进程处于状态R，因为如果$s_p = R$，那么由行为（3），或者进程p被激活，或由行为（2），它的前驱被激活。

如果γ没有被定向，则存在两个进程相互指向对方。即，对于进程p和q，$l_{pq} = l_{qp} = succ$。如果p和q中有一个处于空闲，则行为（1）被激活，如果p和q都处于空闲，则行为（5）被激活，如果p和q都在发送，则行为（4）被激活。由此可得，没有接收进程的一个终止配置被定向，因此所有终止配置被定向。　□

定理17.10　*协议收敛到合法状态。*

⊖ Israeli和Jalfon考虑了一种更精细粒度的行为可分性，需要明确地打破对称性。根据［IJ90］，算法17.1对于中心守护程序有效。

证明。令牌创建以对的形式 $\overrightarrow{II}$ 出现，并且对于5种行为的任何一种，都不会形成这样的对。因此仅在初始空闲的进程产生一个令牌，并作为那个进程的第一步。这蕴含着一次执行中所存在的令牌总数不超过N。

如果在执行期间，配置中存在一个令牌，这个令牌已经移动k次，那么k+1个进程序列（以持有令牌的进程作为结束）被一致定向。所有这些进程已经转发了令牌，且采用了它的方向，没有可能会再次打乱这个方向的令牌在此令牌“之后”被创建。因此如果一个令牌已经移动$N-1$次，所有进程同样被定向，则配置是合法的。

527 ~ 529

因为在执行期间，不存在多于N个的不同令牌，在至多N^2个令牌步后，达到合法配置。□

17.2.2 最大匹配

图的一个匹配是边的集合，满足图中没有一个节点会依附于多于一条的边。如果这个集合不能用图中更多的边扩充，则称匹配是最大的。可用边数的线性时间来构造图的最大匹配。依次考虑每条边，如果这条边没有依附于匹配中已有的任何边，就把这条边加入到匹配中。然而，这个算法本质上是顺序的，并不适合于分布式执行。我们将会给出Hsu and Huang [HH92]提出的构造最大匹配的一个稳定算法。

1. 匹配算法

进程p中有一个变量$pref_p$，它的值属于$Neigh_p \cup \{nil\}$，表示p的首选近邻（preferred neighbor）。如果$pref_p = q$，p选择了它的近邻q，与之匹配，即，将边pq包含在匹配中。如果p没有选择相匹配的同伴，则有$pref_p = nil$。根据p的选择和其近邻的选择，我们区分5种情形。如果p已经选择q，且如果q还未做出选择，那么p等待q；如果q已经选择p，那么p与q匹配；如果q已经选择了一个不是p的近邻，那么p进行链接。当p还没有做出选择时，如果p的所有近邻都已匹配，那么p是死锁的；如果存在一个未匹配的近邻，那么p是自由的。表示为

$$
\begin{aligned}
&wait(p) && \equiv pref_p = q \in Neigh_p \wedge pref_q = nil\\
&match(p) && \equiv pref_p = q \in Neigh_p \wedge pref_q = p\\
&chain(p) && \equiv pref_p = q \in Neigh_p \wedge pref_q = r \in Neigh_q \wedge r \neq p\\
&dead(p) && \equiv pref_p = nil \wedge \forall q \in Neigh_p : match(q)\\
&free(p) && \equiv pref_p = nil \wedge \exists q \in Neigh_p : \neg match(q)
\end{aligned}
$$

算法要求的后置条件为

$$\psi \equiv \forall p : (match(p) \vee dead(p))$$

var $pref_p$: $Neigh_p \cup \{nil\}$;

$\mathbf{M}_p$: $\{pref_p = nil \wedge pref_q = p\}$
$pref_p := q$

$\mathbf{S}_p$: $\{pref_p = nil \wedge \forall r \in Neigh_p : pref_r \neq p \wedge pref_q = nil\}$
$pref_p := q$

$\mathbf{U}_p$: $\{pref_p = q \wedge pref_q \neq p \wedge pref_q \neq nil\}$
$pref_p := nil$

图17-3 最大匹配算法

命题17.11　如果ψ成立，集合$M=\{(p, pref_p): pref_p \neq nil\}$是最大匹配。

证明。如果存在一条边$pq \in M$，那么由M的定义，$pref_p = q$或者$pref_q = p$，但由于q不等待或链接，后者还蕴含$pref_p = q$，由此可得，至多一条依附于p的边属于M，因此M是一个匹配。

为了证明M是最大匹配，假设$M \cup \{pq\}$也是匹配，其中$pq \notin M$，那么M没有包含依附于p的边，这表明$dead(p) \vee free(p)$成立，因此，由ψ可得，$dead(p)$成立。但是$dead(p)$蕴含$match(q)$，因此M包含依附于q的一条边，这与用pq对M进行的可能的扩展相矛盾。　□

算法的描述基于读-所有状态模型，由每个进程p的三种行为组成。参见图17-3所示的算法。如果进程p是自由的，且进程q已经选择了p，则进程p与近邻q匹配（行为$\mathbf{M}_p$）。如果p是自由的，但还未匹配，则它选择一个自由近邻，如果这样做是可能的（行为$\mathbf{S}_p$）。如果p正在链接，则它处于释放状态（行为$\mathbf{U}_p$）。

2. 算法分析

引理17.12　配置γ是终止的，当且仅当$\psi(\gamma)$成立。

证明。行为$\mathbf{M}_p$得到p允许，仅当近邻q正处于等待状态，行为$\mathbf{S}_p$要求p是自由的，行为$\mathbf{U}_p$要求p正在链接，因此$\psi(\gamma)$蕴含在γ中没有行为得到允许。

如果ψ不成立，则存在p满足，p正在链接、等待或是自由的。在一次链接p中，$\mathbf{U}_p$得到允许，并且如果p正等待q，则$\mathbf{M}_q$得到允许。最后，假设p是自由的，且q是一个未匹配的近邻。因为p没有匹配，q不是死锁的。如果q正在等待，匹配行为在它的一个近邻中得到允许，且如果q正在链接，则$\mathbf{U}_q$得到允许。最终，如果q是自由的，p和q可以相互选择。因此，如果ψ不成立，配置不是终止的。　□

530
~
531

引理17.13　图17-3所示的算法以$O(N^2)$步达到终止配置。

证明。用数对$(c+f+w, 2c+f)$定义范函数F，其中c、f和w分别表示链数、自由进程数和等待进程数。我们将证明，F是步数的递减函数（按照词典次序）。首先可知，一个已匹配的进程会永远保持匹配，而一个死锁进程永远保持死锁状态。因此$c+f+w$从不递增。

$\mathbf{M}_p$: 当p是自由的，并且它的近邻q正在等待，应用该行为，使得两者相匹配。同时使$c+f+w$减少2。（如果p和q的近邻变成死锁，这个和可能还会减少。）

$\mathbf{S}_p$: 当p是自由的，可应用该行为，使得p变为等待状态，同时$2c+f$减少1。没有等待进程变成自由的，或是链接的，因为仅当没有进程等待p，且没有自由进程变成链接的时候，才能应用该行为。

$\mathbf{U}_p$: 当p是链接的，可应用该行为，使得p变成自由的（如果有未匹配的近邻），或p变成死锁的（如果所有进程都已匹配），因此，至少使$2c+f$减少1。链接p的近邻可能变成等待，因此会进一步减少c。

由于$c+f+w$上限为N，$2c+f$上限为$2N$。因而，F的不同值的数目至多为$(N+1)(2N+1)$，因此，每次执行在$2N^2+3N$步后终止。　□

易证明，取所有满足ψ的配置作为合法配置，图17-3所示的算法稳定到ψ。

17.2.3　选举和生成树构造

Afek、Kutten and Yung[AKY90]提出一个稳定算法，用于计算网络上的一棵生成树，其中最大进程是根。我们描述了读-所有状态的模型上的算法（尽管该算法也可用于读-一个状态的模型），并假设网络是连通的。在本小节中，我们利用词首大写表示进程变量，词首小写

表示谓词。

进程p保持变量$Root_p$、Par_p及Dis_p，用以描述树结构。我们引入以下谓词。 532

$$
\begin{aligned}
root(p) &\equiv Root_p = p \wedge Dis_p = 0\\
child(p,q) &\equiv Root_p = Root_q > p \wedge Par_p = q \in Neigh_p \wedge Dis_p = Dis_q + 1\\
tree(p) &\equiv root_{(p)} \vee \exists q : child(p,q)\\
lmax(p) &\equiv \forall q \in Neigh_p : Root_p \geqslant Root_q\\
sat(p) &\equiv tree(p) \wedge lmax(p)
\end{aligned}
$$

算法所需的后置条件为ψ，用$\forall p$：$sat(p)$定义。

引理17.14 ψ蕴含着，边集$\{(p, q): child(p, q)\}$形成一棵以最大进程为根的生成树。

证明。由于对于所有p，$lmax(p)$得到满足，且网络是连通的，所有进程具有相同$Root$值。具有最小Dis值的进程p_0没有$Dis_p = Dis_q + 1$的近邻q，因此$root(p_0)$成立，$Root$变量的公共值为p_0。但对于所有$p \neq p_0$，$child(p)$成立，并且，因为$child(p, q) \Rightarrow (Dis_p > Dis_q)$，因此孩子关系是无环的。引理得证。 □

1. 算法描述

为了建立ψ，$sat(p)$为真的进程p从不改变它的那些描述树结构的变量（参见图17-4所示的算法）。$sat(p)$为假的进程p，通过变成它的具有最大$Root$值（行为$\mathbf{J}_p$）的近邻的一个子节点，试图建立$sat(p)$。

称$Root_q$的值为假根（false root），如果在网络中，不存在具有那个标识的进程；初始时可能存在假根，但执行中不会创建假根。算法设计中的主要问题是，防止进程无限频繁地变成带假根进程的孩子节点。为此，可用三步完成将进程p加入q的树中的过程。首先，p成为根（行为$\mathbf{B}_p$），然后，请求允许加入（行为$\mathbf{A}_p$），最后当q对加入认可时，将p加入（$\mathbf{J}_p$）。

其余4种行为（$\mathbf{C}_p$，$\mathbf{F}_p$，$\mathbf{G}_p$和$\mathbf{R}_p$）实现请求/应答机制，而且仅被满足$sat(p)$的进程p执行。变量Req_p包含进程，它的加入请求p正在处理。$From_p$是进程p的近邻，p读取来自该近邻的请求To_p是进程p向其进行转发的近邻，Dir_p（值Ask和$Grant$）表示请求是否得到同意。我们定义 533
以下谓词。

$$
\begin{aligned}
idle(p) &\equiv Req_p = From_p = To_p = Dir_p = udef\\
asks(p,q) &\equiv ((root(p) \wedge Req_p = p) \vee child(p,q)) \wedge To_p = q \wedge Dir_p = Ask)\\
forw(p,q) &\equiv Req_p = Req_q \wedge From_p = q \wedge To_q = p \wedge To_p = Par_p\\
grant(p,q) &\equiv forw(p,q) \wedge Dir_p = Grant
\end{aligned}
$$

进程p清除请求处理的变量（行为$\mathbf{C}_p$），如果它当前没有处理请求，并且变量并非未定义。然后，如果p处于空闲状态，但有一个近邻q，向p发出请求（想要加入的根，或转发请求的p的子节点），进程p可能开始转发这个请求（行为$\mathbf{F}_p$），如果p是根节点并转发请求，它就会同意请求（行为$\mathbf{G}_p$），如果p将这个请求转发给它的父节点，且父节点同意请求，那么p就传递这个请求许可（行为$\mathbf{R}_p$）。

2. 算法正确性

Afek *et al.*[AKY90]利用行为推理，通过证明执行中的某些性质，证明了算法的正确性。

请求/应答机制确保，进程加入到一棵具有假根的树中的次数是有限的。因为不存在的根的请求不会得到许可，初始时，只存在有限多的假的许可。因此，没有一个$Root$变量会永远包含一个假根。包含假根且具有最小Dis值的进程不满足$tree$，并会重新将$Root$设置成自己的

标识。由此可得，最终不存在假根，并且最终具有最大标识的进程将会成为根，因为在没有更大*Root*值时，这是满足*sat*的惟一途径。

```
var Root_p, Par_p , Dis_p ;          (* Describe tree structure *)
    Req_p, From_p, To_p, Dir_p ;     (* Request forwarding *)

B_p: (* Become root *)
    { ¬tree(p) }
    Root_p := p ; Dis_p := 0 ;
    Req_p := p ; To_p := q ; Dir_p := Ask

A_p: (* Ask permission to join *)
    { tree(p) ∧ ¬lmax(p) }
    Select q ∈ Neigh_p with maximal value of Root_q ;
    Req_p := p ; From_p := p ; To_p := q ; Dir_p := Ask

J_p: (* Join tree *)
    { tree(p) ∧ ¬lmax(p) ∧ grant(To_p, p) }
    Par_p := q ; Root_p := Root_q ; Dis_p := Dis_q + 1 ;
    Req_p := From_p := To_p := Dir_p := udef

C_p: (* Clear request variables *)
    { sat(p) ∧ ¬∃q : forw(p, q) ∧ ¬idle(p) }
    Req_p := From_p := To_p := Dir_p := udef

F_p: (* Forward request *)
    { sat(p) ∧ idle(p) ∧ asks(q, p) }
    Req_p := Req_q ; From_p := q ; To_p := Par_p

G_p: (* Grant join request *)
    { sat(p) ∧ root(p) ∧ forw(p, q) ∧ Dir_p = Ask }
    Dir_p := Grant

R_p: (* Relay grant *)
    { sat(p) ∧ grant(Par_p, p) ∧ Dir_p = Ask }
    Dir_p := Grant
```

图17-4　生成树算法

如果一个节点不属于以最大节点为根的树，但在那棵树中确有一个近邻，则节点不满足*sat*谓词，并尝试加入，而树中的节点从不遗弃它。由于所有不正确的请求最终都会从树中消失，节点继续向树中增加，直到这棵树扩散至整个网络。

17.3　稳定方法学

稳定性要求，使得分布式算法设计相当复杂。但可利用一般范型。在17.3.1小节中，第一阶段讨论了既使不能检测出稳定性如何进行稳定计算。我们集中在Herman的研究[Her91]上，但是所用技术已知，并且前面已经用过。在17.3.2小节中，证明了凡是能够选择恰当的度量，表示成找“最小路径”的问题，都存在稳定算法。

17.3.1　协议组合

在许多情形中，达到期望的后置条件ψ的算法由两个阶段组成。第一阶段达到后置条件θ，

第二阶段前置条件为θ，并达到后置条件ψ。有无数组合算法的例子，包括本书中涉及的一些算法。 534 ~ 535

（1）*路由* 大多数路由方法从计算每个节点的路由表开始，计算之后，利用这些路由表转发包。

（2）*选举* 在网络上选举领导人原因通常在于，期望在网络中接着执行集中式算法。

（3）*死锁检测* 死锁检测的全局标记算法（图10-8所示的算法）首先计算基本计算的快照，然后对所得到的配置进行分析。

（4）*图的着色* 对于平面图的6-着色问题，通过计算图的一种合适无环定向（参见定义5.12），然后按照与这个方向一致的次序对图中节点着色。

利用终止检测协议，可以指导在两个经典的分布式算法之间进行控制转换。在第一阶段中添加这样的协议（参见第8章），当它表明第一阶段已经完成时（即，已经建立θ后），进程开始执行第二阶段。当所设计的算法一定要是稳定的，此时就不能使用类似的方法。因为不可能知道（按照稳定的方式）第一阶段的终止性。

幸运的是，稳定算法的类似组合更简单：第二阶段的执行可在任意时刻开始，即使第一阶段不能完成时。如果一个经典算法（对于第二阶段）在它的前置条件被建立之前开始，可能达到不可恢复的故障状态。这就需要检测第一阶段的终止性。然而，如果第二阶段是稳定的，尽管在前置条件达到值*true*之前，有一些错误步，它最终还是建立了它的前置条件。

1. 并-组合规则

在组合算法算子的定义中，Herman[Her91]将这些观察形式化。假设程序是由变量集以及作用在这些变量上的原子步组成。

定义17.15 *设S_1和S_2是程序，满足由S_2写入的变量不会出现在S_1中，符号$S_1 \boxdot S_2$表示S_1和S_2的并组合，它表示由S_1和S_2中的所有变量及行为组成的程序。* 536

在定义中，S_1表示实现计算第一阶段的程序，S_2表示实现第二阶段的程序。对S_1和S_2的限制意味着S_2的结果不被S_1所用，即，不存在从S_2到S_1的“反馈”。S_1中的变量在S_2中作为常量。

现在设θ是作用在S_1中变量上的谓词，ψ是作用在S_2中变量上的谓词。在组合算法中，θ由S_1确定，随后ψ由S_2确定。这两个技术条件必须铭记在心。首先，组合的执行对这两个程序都是公平的。这样做是排除一些执行，在这些执行中所有步出现在一个阶段，而阻止另一阶段继续执行。

定义17.16 *$S_1 \boxdot S_2$的执行对于S_i是公平的，如果它包含S_i中的无限多步，或者包含一个无限后缀，在这个后缀中，S_i中没有一步可以进行。*

其二，当条件θ已经被确定时，假设S_1不能改变S_2所读的变量。这就排除了通过为它提供不同的满足θ的变量设置，S_1来阻止S_2中程序执行的情形。

定理17.17 *如果以下四个条件成立：*

（1）程序S_1稳定到θ；

（2）程序S_2稳定到ψ，如果θ成立；

（3）一旦θ成立，程序S_1不能改变S_2所读的变量；且

（4）所有执行对于S_1和S_2都是公平的。

那么，$S_1 \boxdot S_2$稳定到ψ。

证明。我们通过涉及$S_1 \boxdot S_2$执行序列的推理，来确立这个结果。对于$S_1 \boxdot S_2$的配置γ，设

$\gamma^{(i)}$表示γ在S_i的变量上的投影。对于$S_1 \boxdot S_2$的一次执行$E = (\gamma_0, \gamma_1, \ldots)$，设$E^{(i)}$表示重复序列已被删除的序列$(\gamma_0^{(i)}, \gamma_1^{(i)}, \ldots)$。

设$E = (\gamma_0, \gamma_1, \ldots)$是$S_1 \boxdot S_2$的一次执行，并考虑序列$E^{(1)}$。因为$S_2$不会写入$S_1$的变量，$\gamma^{(1)}$中的所有改变都是按$S_1$中的步骤来进行，这蕴含着$E^{(1)}$是$S_1$的一次执行。$E$对于$S_1$的公平性蕴含着，如果$E^{(1)}$是有限的，那么最后配置对于$S_1$是终止的。因为$S_1$稳定到$\theta$，由此可得，$\theta$被确定，即存在$i$，满足对于每个$j \geqslant i$，$\theta(\gamma_j)$为真。

我们下面考虑在γ_i中开始的E的后缀E_i，即序列$(\gamma_i, \gamma_{i+1}, \ldots)$。在这个后缀中，对于每个配置，$\theta$为真，这蕴含着（由定理的第三个假设）$S_1$没有改变$S_2$所读变量。因此，序列$(\gamma_i^{(2)},$
537 $\gamma_{i+1}^{(2)}, \ldots)$是$S_2$的一次执行，如果有限，该序列结束于$S_2$的终止配置中。如果$\theta$成立，$S_2$稳定到$\psi$，由此可得，$\psi$被建立，并保持为真。 □

如果通过提供足够的范函数f_1和f_2（值域分别为W_1和W_2），可证明S_1和S_2的稳定性，就能给出$S_1 \boxdot S_2$的范函数。定义函数$f(\gamma) = (f_1(\gamma^{(1)}), f_2(\gamma^{(2)}))$。当$S_1$的每一步使$f_1$减少，$S_2$的每一步使$f_2$减少，并剩下$f_1$未变时，那么按照$S_1 \boxdot S_2$中的每一步使得$f$按照字典次序减少。然而，如果有一个程序确定了它的后置条件，而没有终止（例如，环定向程序），仍然需要公平性条件保证两个程序的执行。参见文献[Fra86]关于公平性问题的讨论。

2. 应用：平面图的6-着色问题

作为协议组合原理的一种应用，我们提出一种算法，它可以稳定到平面图的6-着色问题。假设进程p有变量c_p，取值为{1，2，3，4，5，6}；如果图中近邻着不同颜色，则称图是6-着色的。即如果如下定义的谓词ψ为真，

$$\psi \equiv (\forall pq \in E : c_p \neq c_q)$$

算法的开发过程接近证明过程。这样的着色算法存在，并基于以下事实。

事实17.18　一个平面图至少有一个度小于等于5的节点。

引理17.19　一个平面图有一个无环定向，其中的每个节点出度至多为5。

证明。对节点数用归纳法证明。单个节点的情形是平凡的。因此考虑图中有两个或者多个节点的情形。至少有一个节点v的度小于等于5。图$G-\{v\}$是平面图，并且只少一个节点。因此，由归纳法，它有无环方向图，在这个图中，每个节点的出度至多为5。在G的定向中，图$G-\{v\}$中所有边按照这个方向定向，而依附于v的边则远离v定向。显然，所得方向图是无环的，并且不存在5条以上的出边节点。 □

定理17.20（6-色定理）　平面图是6-可着色的。

证明。设G是平面图。考虑G的无环方向图。其中不存在有5条以上出边的节点。因为方
538 向图是无环的，可把节点编号为v_1到v_n，满足如果存在一条从v_j到v_i的有向边，那么$j > i$。按照v_1到v_n的次序对节点着色。编号性质意味着，在节点v之前，只有v的出近邻被着色。这一点以及方向图的性质蕴含着，在v之前，至多有v的5个近邻被着色，因此，v的着色可不同于它的所有（已着色的）近邻。 □

6-色定理的证明过程，表明了计算6-着色问题的算法设计过程。第一阶段计算引理17.19中表示的无环方向图。第二阶段按照与定向一致的次序对节点着色。为了表示无环方向图，给定进程p一个整数变量x_p，我们定义边pq为从p到q的有向边，用$\vec{pq}$表示。如果

$$x_p < x_q \vee (x_p = x_q \wedge p < q)$$

显然，只有一种方式对每条边确切定向（即，$\vec{pq}$和$\vec{qp}$中只有一种成立），且方向图是无环的。设$out(p)$表示p的出边的数目，即

$$out(p) = \#\{q \in Neigh_p : \vec{pq}\}$$

第一阶段希望的后置条件是θ，定义为

$$\theta \equiv (\forall p : out(p) < 5)$$

为了确定θ，程序S_1由每个进程的一次操作组成，如果p有多于5条的出边：

$\mathbf{D}_p$: $\{out(p) > 5\}$
$x_p := \max\{x_q : q \in Neigh_p\} + 1$

则将p的所有边定向p。

定理17.21 *程序S_1稳定到θ，并且如果θ成立，那么x_p值保持不变。*

证明。配置是终止的，当且仅当θ为真。因此如果程序停止，θ就被确定，而如果θ被确定，变量x_p保持为常数。

为了证明S_1的终止性，我们“引用”引理17.19中的无环有向图，然而，这个引用的有向图不必是由S_1计算出来的。引理17.19蕴含，存在节点v_1，v_2，...的枚举，满足每个节点至多有5个具有更大下标的近邻。称边v_iv_j为*错误的*，如果它被定向为$\vec{v_iv_j}$，而$i > j$。设$f(\gamma)$是表$(n_1, n_2, \ldots)$，这里n_i是依附于v_i的错误边数。行为$\mathbf{D}_p$的每次应用使f减少（按字典次序）。考虑它在节点v_j中的应用，它的具有最小下标的出近邻是v_i。由于v_j至少有6个出近邻，但至多有5个更 539
大下标的近邻，$i < j$，因此边v_iv_j是错误的，但不再是这样一条边，因此使n_i减少，而对于$k < i$，n_k不受影响。 □

要确定ψ，程序S_2也由每个进程的一次操作组成。在这个操作中，如果c_p等于这些颜色之一，并且存在未用颜色，则p采用一种不同于它后继的颜色b：

$\mathbf{C}_p$: $\{(\exists q: \vec{pq} \wedge c_p = c_q) \wedge (\forall r \text{ s.t. } \vec{pr}: c_r \neq b)\}$
$c_p := b$

定理17.22 *程序S_2终止，并且如果θ为真，那么最终配置满足ψ。*

证明。为了证明终止性，利用被定向在无环方向图中的那些边，因此存在可数个节点，v_1，v_2，...，满足，每个节点的后继具有更小下标。定义$g(\gamma) = (m_1, m_2, \ldots)$，其中$m_i$为0，如果$c_i$与每个$v_i$的后继着色不同；否则$m_i$为1。节点$v_i$应用行为$\mathbf{C}_p$，将$m_i$值从1变为0，对于$j > i$，也可能只将$m_i$值从0变为1。因此$g$的值按字典次序减少。

最后，假设θ成立，并考虑不应用行为$\mathbf{C}_p$的配置。进程p至多有5个后继，因此存在一种颜色$b \in \{1, 2, 3, 4, 5, 6\}$，不为它的任一后继所用。由于$\mathbf{C}_p$的阈值为假，由此$c_p$与每个后继的颜色不同。

最后，对于每条边qr，r是q的后继，或者q是r的后继，这蕴含着$c_r \neq c_q$。 □

S_1和S_2的并组合如图17-5所示的算法。

定理17.23 *图17-5所示的算法稳定到ψ。*

证明。可以构造并组合。因为S_2中的变量（c_p）并不出现在S_1中。上述表明，S_1稳定到θ，在确定它之后，并不改变x_p的值（定理17.21），且如果θ成立，S_2稳定到ψ（定理17.22）。

为了证明由定理17.17，$S_1 \boxdot S_2$稳定到ψ，仍需证明这种组合是公平的。由定理17.22，在

S_1的每两步之间，只有S_2中的有限步。这证明了关于S_1的公平性。由定理17.21，S_1终止，这证明关于S_2的公平性。□

var x_p : integer ;
　　c_p : $\{1, 2, 3, 4, 5, 6\}$;

　　$p\vec{q} \equiv x_p < x_q \ \vee \ (x_p = x_q \wedge p < q)$;
　　$out(p) \equiv \#\{q \in Neigh_p : p\vec{q}\}$;

$\mathbf{D}_p$: { $out(p) > 5$ }
　　$x_p := \max\{x_q : q \in Neigh_p\} + 1$

$\mathbf{C}_p$: { $(\exists q : p\vec{q} \wedge c_p = c_q) \wedge (\forall r$ s.t. $p\vec{r} : c_r \neq b)$ }
　　$c_p := b$

图17-5　平面图的6-可着色算法

17.3.2　计算最小路径

本小节给出解决一类问题的稳定算法，这类问题可以变成通过恰当选择路径的代价函数寻找最小路径。例子不仅包括计算路由表（直接），而且包括计算深度优先搜索树。更令人惊讶的是，还包括选举问题（参见习题17.12）。我们首先给出一个代数设置中的最小路径问题，然后提出修正算法（图17-6所示的算法），这个算法稳定到最小路径。在本节最后，给出了它在实际中的应用。

var K_p : D ;
　　L_p : $Neigh_p \cup \{nil\}$;

$\mathbf{C}_p$: $K_p := \min(c_p, \ \min\{f_{pq}(K_q) : q \in Neigh_p\})$;
　　if $K_p = c_p$ **then** $L_p := nil$
　　　　else $L_p := q$ s.t. $K_p = f_{pq}(K_q)$

图17-6　最小路径更新算法

1. 代价函数和最小路径的性质

假设给定从路径到全序域U的函数D。对于路径π，值$D(\pi)$称为π的代价。从p到p的长度为0的空路径（p）的代价，用c_p表示，并假设空路径为p所知。我们考虑代价函数，对于这个代价函数，可以递增地计算路径代价。这意味着，非空路径的代价可被计算为

$$D(p_0,\ldots,p_k,p_{k+1}) = f_{p_{k+1}p_k}(D(p_0,\ldots,p_k))$$

f_{pq}是函数f_{pq}: $U \to U$，称为**边缘函数**（edge function），每一条边pq有一个边缘函数。路径π = (p_0, p_1, …, p_{k-1}, p_k) 的代价可以递增地计算为$f_{p_kp_{k-1}}(\ldots(f_{p_1p_0}(c_{p_0}))\ldots)$。定义$f_{p_kp_{k-1}}(\ldots(f_{p_1p_0}(x))\ldots)$的值为$f_\pi(x)$，记$D(\pi)$为$f_\pi(c_{p_0})$。假定代价函数（和边缘函数）满足以下公理。

（1）**单调性**　对于每条边pq和$x \in U$，如果π是一条到q且不含p的简单路径，那么$D(\pi) < x \Rightarrow f_{pq}(D(\pi)) < f_{pq}(x)$

（2）**周期增加**　如果π是回路，那么对于所有$x \in U$，$f_\pi(x) > x$。

(3) **长度增加** 存在数B，满足对于长度大于等于B的每条路径ρ，每个$x \in U$，以及长度小于等于$N-1$的每条简单路径π，有$f_\rho(x) > D(\pi)$。

对于每个p，最小路径问题是计算任何以p为结束点的路径最小代价，以及最小代价路径上它的前驱q（如果最小路径为空，则为nil）。用$\kappa(p)$表示最小路径的代价，$\phi(p)$表示它的前驱。以下定理表述了最小路径的存在性，极其具有的重要性质。

定理17.24

(1) 对于每个p，存在一条结束于p的简单路径$\pi(p)$，满足所有结束于p的路径代价至少为$D(\pi(p))$。

(2) 存在生成森林（称为最小路径森林），满足对于每个p，从根到p的（惟一）路径有代价$D(\pi(p))$。

证明。对于(1)：因为只有有限多条简单路径结束于p，我们可以选择$\pi(p)$作为结束于p的最小简单路经。由$\pi(p)$的选择，所有结束于p的简单路径代价至少为$D(\pi(p))$。含有回路的路径代价（由回路增加和单调性可得）大于去掉回路得到的简单路径的代价，因此这些路径代价有界$D(\pi(p))$。这就表明$\pi(p)$证明了定理的第一部分。

对于(2)：作为最小路径，不必是惟一的。要求对定理的第二部分特别关注。取出现在最小路径中的所有边所得到的图不一定是森林。像定理4.2中考虑的最优汇集树，最小路径森林不是惟一的，但是可能构造这样一个森林。如下所示。

对于每个节点p，选择它的父节点$\phi(p)$如下。如果空路径(p)是最优的，则$\phi(p) = nil$；否则$\phi(p)$是q的近邻，满足到p的最小路径存在，且在这条最小路径上有倒数第二的节点q。考虑边p，$\phi(p)$，对于$\phi(p) \neq nil$的情形。

由单调性，对于$\phi(p) \neq nil$的每个p，$\kappa(p) \geq f_{p\phi(p)}(\kappa(\phi(p)))$。于是，如果$C$是所形成的图中的一条回路（从$p$到$p$），那么$\kappa(p) \geq f_C(\kappa(p))$，这与回路增加性质矛盾，因此所选的边定义了一个森林。

由对根的归纳证明了这个森林所要求的性质。 □

观察可见，在证明中，无需用到长度增加性质。仅需证明，初始配置中的错误信息，最终在计算最小路径的稳定算法中被去掉。

现在规定一个最小路径问题算法所要求的后置条件：

$$\psi \equiv (\forall p : K_p = \kappa(p) \wedge L_p = \phi(p))$$

确定ψ的经典（非稳定的）算法由每边上的一个操作组成，称为**推送**（pushing）这条边。

$\mathbf{P}_{pq}$: $\{f_{pq}(K_q) < K_p\}$

$K_p := f_{pq}(K_q)$; $L_p := q$

变量K_p初始化为c_p。可以证明断言$K_p \geq \kappa(p)$是不变式。为了证明活动性，我们必须假设推送对于每一条边是公平的，即，在一个无限执行中，每条边会被无限频繁地推送。我们对通向p的最小路径上的节点用归纳法，证明$K_p \leq \kappa(p)$的建立过程。读者可以比较Chandy-Misra计算最小路径的算法（图4-7所示的算法）。

2. 更新算法

为了以稳定的方式建立ψ，通过沿着边推动，有关现有路径的信息可能在图中流动。然而，

仍然需要从系统中去掉出现在初始配置中的错误信息，即，去掉那些不存在通向p的相应路径的K_p的值。这可以通过在越来越长的路径上移动该信息做到，路径满足按照长度增加的性质，最终错误信息将被拒绝，以支持与现有路径相关的信息。

更新算法由每个节点的一个操作组成，在算法中，计算空路径和路径的最小代价，路径
540 的代价被存储在它的近邻中，参见图17-6所示的算法。在算法分析中，正如在推动算法中所
l 做的那样，假设执行对于每个进程都是公平的，即，在每次执行中，各进程可计算无限多次。
543

定理17.25　*更新算法稳定到最小路径。*

证明。因为$\mathbf{C}_p$总是活动的，每个执行是无限的，公平性假设允许我们将执行划分成如下的轮（round）。第1轮结束时，每个进程至少已经计算一次。第$i+1$轮结束时，每个进程在第i轮结束后至少已经计算一次。

首先证明，最终K_p是$\kappa(p)$的下界。定义$\kappa_i(p)$为到p的，长度小于等于i的路径上的最小代价，我们认为，从第i轮结束起的任何时刻，有$K_p \leq k_{i-1}(p)$。对i进行归纳，来确定这一声明。

当$i=1$时：空路径代价为c_p，是长为0的到p惟一路径，由代码可证明，在第一个$\mathbf{C}_p$步之后，$K_p \leq c_p$。

当$i+1$时：设π是到p长度至多为i的最小路径，如果π的长度为0，而p执行每步后，$K_p \leq D(\pi)$，那么在$i+1$轮结束后同样成立。假设π是非空路径，即，路径ρ和边qp的连接，ρ的长度至多为$i-1$。由归纳得，在第i轮结束后，$K_p \leq D(\rho)$。由此可得，在i轮结束后p执行完每一步，$K_p \leq f_{pq}(D(\rho)) = D(\pi)$。在$i+1$轮结束时，$p$至少执行一个这样的步，自那一轮结束起，表明$K_p \leq D(\pi)$。

由于最小路径不含回路，它的长度受$N-1$所限，这蕴含着$\kappa(p) = \kappa_{N-1}(p)$，因此从第$N$轮结束起，$K_p \leq \kappa(p)$。

为了证明最终K_p是$\kappa(p)$的上界。我们将要证明，在第i轮之后，K_p值与一条现有的路径对应，或者与经过至少i次跳数的已经遍历过的初始信息对应。定义K_r的初始值为K_r^*。我们声明，如果$K_p = K$，那么存在到p的路径π，且$D(\pi) = K$，或者存在从r到p的路径ρ，满足$K = f_\rho(K_r^*)$。然而，如果K在第i轮被计算，那么ρ的长度至少为i。对一次执行中的步骤用归纳法，可得这个声明。

归纳基础：初始时，$K_r = f_{(r)}(K_r^*)$，其中(r)是长度为0的路径。因此K_r可能对应一条现存路径，也可能不与现存路径对应。

归纳步：考虑p在一步中计算的K值。如果$K = c_p$，它就是一条到p的现存路径（长度为0）的代价，结论成立。否则，$K = f_{pq}(K_q)$。由归纳假设，K_q是（1）到q的路径π的代价，或（2）存在从r到q的路径ρ，满足$K_q = f_\rho(K_r^*)$。在第一种情形，$\pi \cdot qp$是一条代价为$f_{pq}(D(\pi)) = f_{pq}(K_q) = K$的现存路径，结论成立。在第二种情形，$K = f_{pq}(f_\rho(K_r^*)) = f_{(\rho \cdot qp)}(K_r^*)$。然而，如果$K$在第$i$轮被计算，那么$K_q$就在第$i-1$轮或稍后被计算，通过归纳，这蕴含着$\rho$的长度至少为
544 $i-1$，因此$\rho \cdot qp$的长度至少为i。

结果是长度增加公理的一个应用。在第B轮的结束时，$f_\rho(K_r^*)$超过了任何现存简单路径的代价，这表明某些到p的现存路径的代价是K_p的下界。因此，在第B轮的结束时，对于每个p，$K_p = \kappa(p)$。容易验证，L_p的值满足为$\phi(p)$所设置的要求，这蕴含着，边(p, L_p)定义了一

个最小路径森林。 □

如果网络规模已知，可以在不满足它的路径代价函数上实现长度增加性质。修改路径代价，使它也包括长度。当比较代价时，长度大于等于N的路径总是大于长度小于等于$N-1$的路径。为了形式地描述这种修改，设给定代价函数D。用$D'(\pi) = (D(\pi), |\pi|)$定义代价函数$D'$，$p$的空路径的代价为$c'_p = (c_p, 0)$。用$f'_{pq}(C, k) = (f_{pq}(C), k+1)$表示边缘函数。路径代价比较如下

$$(C_1,k_1) <' (C_2,k_2) \Leftrightarrow (k_1 < N \wedge k_2 \geqslant N) \vee (C_1 < C_2)$$

所有简单路径可以按照D'定序，就像按照D定序一样，这表明保持了单调性。同时修改后，还保持了回路增加性质。因为长度至少为N的路径的代价大于长度小于N的路径代价，因此满足长度增加的性质。

3. 应用：路由

更新算法的直接应用出现在路由表的计算中。像变更算法一样，每个目的节点分别进行计算。固定目的节点v，如果π在v开始，定义$D(\pi)$为π的权（即，π中边上的权值之和）；否则为∞。如果$p = v$，则设$c_p = 0$；否则设为∞，并通过设置$f_{pq}(C) = C + w_{pq}$（其中w_{pq}为边pq上的权值），可以递增地计算$D(\pi)$。

由边缘函数的另外一些性质，可以得出它的单调性。由网络中的回路有正的代价这一假设可得回路增加性质。为了证明长度增加性质，注意只有有限多个简单回路，设δ是一个简单回路的最小权值，w是任何一条边上的最大权值。长度小于等于$N-1$的路径，它的权值小于N_w，而长度为$B = N(Nw)/\delta$的一条路径包含多于$(Nw)/\delta$条简单回路，并且权值大于Nw。

因此，用更新算法可以计算出到v的距离（对于v选定的近邻，到v的最短路径上的第一个近邻）。为了建立完整的路由表，对于每个目的节点，可以并行地执行算法。 545

4. 应用：深度优先搜索树

可以证明，按照字典次序选择到每个进程的最小简单路径，定义了一棵深度优先搜索生成树，它的根是网络中的最小进程。因此，用更新算法可以构造一棵深度优先搜索树。

定义出现在路径中的节点表为π的代价，即，$c_p = (p)$和$f_{pq}(\sigma) = \sigma \bullet p$。比较按照字典次序进行，但是简单路径总是小于含有回路的路径：

$$\sigma < \tau \Leftrightarrow (\sigma \text{是简单路径而} \tau \text{不是}) \vee (\sigma <_L \tau)$$

字典序性质蕴含着单调性。同样回路增加性质成立，这是因为包含回路的路径大于简单路径。长度增加性质成立，是因为长度大于等于N的路径包含回路，这使它们大于简单路径。因此，更新算法可以计算到每个节点的最小简单路径。接着要分析所得最小路径森林的性质。

首先，讨论由一棵树组成的森林，以最小节点为根。称这个最小节点为r。对于每个节点p，存在从r到p的简单路径，来自r的一条简单路径小于从其他节点到p的路径，因此到p的最小路径在r中开始。

其次，对于近邻p和q，$\kappa_p \lhd q$或$\kappa_q \lhd p$（$\lhd$定义了前缀关系）。事实上，考虑$\kappa(p)$和$\kappa(q)$，不妨假设$\kappa(p) < \kappa(q)$。因为最小路径是简单路径，可得$\kappa(p) <_L \kappa(q)$，同时它还蕴涵着，q没有出现在$\kappa(p)$中，否则它到q的前缀是到q的一条路径，且按字典次序小于$\kappa(q)$，这与$\kappa(q)$的定义矛盾。但是，$\kappa(p) \bullet q$是到q的简单路径的代价，这蕴含着$\kappa(q)$

$<_L \kappa(p) \cdot q$。因此，$\kappa(p) <_L \kappa(q) <_L \kappa(p) \cdot q$。这蕴含着，$\kappa(p) \lhd \kappa(q)$，或等价地，在最小路径树中，$p$是$q$的祖先。

这两个性质，即

(1) 森林由一棵树组成，且

(2) 对于近邻p和q，$\kappa_p \lhd q$或$\kappa_q \lhd p$。

蕴含着更新算法所计算的最小路径森林是一棵深度优先搜索树。更新算法的应用由
546 Herman提出[Her91]，他扩展了该算法，采用的方法是计算网络中的分离节点（如果去掉分离节点，就会使网络断开）。

17.3.3 结论和讨论

本节研究的两个策略，在稳定算法的设计中是非常有用的。并组合可用于设计由两个（或多个）时间上有序步组成的算法，更新算法可被用于求解能够被表示为寻找每个进程的最小路径的问题。

存在一些方法，可自动地将任意算法转换成稳定算法（满足同一规范）。为了达到这个目标，Katz and Perry[KP93]提出以下机制。并发地执行给定的经典算法P，系统反复计算P的全局状态的快照，并验证所构造的配置是否是P的可达配置。如果情况不是这样，复原协议会重新初始化系统到P的一个初始配置。然而，这个想法的实现，由于需要快照计算和复原的稳定算法，因此面临许多技术上的问题。

习题

17.1节

17.1　证明引理17.4。

17.2　证明Dijkstra令牌环在$O(N^2)$步达到合法配置。给出一个界限为N的2次的范函数，它随着算法的每一步递减，来缩短分析过程。

17.3　如果$K \geqslant N$（而不是像在课文中假设的那样$K > N$），证明Dijkstra令牌环已经稳定。（见文献[Hoe99].）

17.2节

547 17.4　修改环定向算法（图17-1所示的算法），使它在建立后置条件后终止。

17.5　（项目）设计一个确定圆环方向的稳定算法，算法最好能终止。

17.6　证明用链接寄存器读-所有模型，可以一致地实现最大匹配算法。

17.7　证明最大匹配算法在$N^2 + O(N)$步稳定，并证明在最坏情况下，该算法有为$\Omega(N^2)$的步数的下界。

17.8　设计构造最大独立集的稳定算法，并计算稳定之前的最大步数。

17.9　给出一个范函数，用来证明图17-4所示算法的终止性。

17.3节

17.10　证明外平面图是3-可着色的。设计一个计算外平面图的3-着色问题的稳定算法。

17.11　设计一个计算平面图的5-着色问题的稳定算法。（见文献[McH90]或关于图算法的

其他课本）

17.12 证明通过给出合适的路径代价函数，更新算法可用于选举和广度优先搜索生成树的计算。

17.13 给出一个计算网络规模的稳定算法。

17.14 证明如何用更新算法计算一棵树的深度。

548

第四部分 附 录

附录A 伪代码使用约定

本书中的算法没有用现有程序设计语言描述，而是用伪代码描述。伪代码简洁，且用户友好，而真正的程序往往因为一些涉及严格语法的细节，而变得晦涩难懂。

类Pascal伪码。本书中所用的伪码和Pascal语言比较类似。给出的算法通常作为系统进程的局部算法。系统名字用作程序中变量的下标。进程名可被用作数组下标，也可使用集合变量。变量$Neigh_p$表示与p相连的进程的集合（p的近邻）。

赋值语句（"$a := expression$"），条件语句（"**if** *condition* **then** *statement*"，还可以包括"**else** *statement*"部分），循环语句（"**while** *condition* **do** *statement*"）和Pascal中对应的语句意义相同。**forall**语句具有形式"**forall** $x \in X$ **do** *statement*"，在这里x表示形参，X表示集合。这条语句的含义是：对于集合X中每个元素x，依次执行*statement*语句。在上面的语句中未确定X集合中元素的执行次序，但是这条语句只能用在次序不重要的地方。例如，从p的每个近邻接收的消息数必须初始化为0，我们可以写做：

forall $q \in Neigh_p$ **do** $R_p[q] := 0$

在所有情况下，没有使用一条语句一个分号分隔的语句列表，而是将语句列表放在**begin**和**end**之间。语句中的行分隔和缩进本身没有特殊的意义，只是为了反映程序的分块结构。

消息传递。伪码中常用"发送"操作和"接受"操作来描述消息传递。消息包括消息类型、0或更多"具有特定类型"的数据域。例如，假设每个进程要将自己的进程标识（进程名）和度（近邻数）发送给它的近邻，为此，**info**类型的消息要用两个数据域，即进程标识和发送方的度。这条消息表示为<**info**，a，b>。

发送操作中必须确定要发送的消息（包括消息的类型和数据域的值）和它的目的地（发送方的近邻是哪一个）。数据域的值是一个能够由发送方求值的表达式。进程p为了向近程q发送它的进程标识和度，可能执行下面的操作：

send <**info**，id_p，$|Neigh_p|$> to q

简化符号"shout"用于向所有的近邻发送信息，即，语句"shout<**info**，id_p，$|Neigh_p|$>"可理解成：

forall $q \in Neigh_p$ **do** send <**info**，id_p，$|Neigh_p|$> to q

接收信息时，不但消息类型，而且消息数据域的形参和消息的发送方也要确定。接收一条与近邻节点标识和度有关的信息，进程p可以执行如下的操作：

receive <**info**，a，b> from q

这个操作删除通信系统中一条类型为**info**的消息。这次操作后，该消息的第一、第二个数据域，以及发送方分别是a，b，q。用这种形式，接收操作不能确定必须从哪一个近邻节点接收消息，但在接收操作之后，所接收的消息的发送方变成已知。

如果进程要从一个特定的进程，如，q_0接收消息，可以执行如下的操作

receive <**info**, a, b> fromthis q_0

有时我们可能要表示：会接收到不同类型的消息，并且随后的行为取决于实际接收到的消息的类型。这可以表示为（例如，在图7-7所示的算法和图7-9所示的算法中）：

receive m from q;　(* or: fromthis q *)
if m is <**info**, a, b> **then** *statement* **else** *statement*

p所使用的参数q是用来描述p的近邻，达到发送/接收称为近邻地址的消息。在*直接定址*（direct addressing）的情况下，地址与进程q的标识相等；这表明进程q的所有近邻利用q的同一个地址。在*间接定址*（indirect addressing）的情况下，地址为到进程p的局部名（信道名）；这种情况下，进程q的近邻可以使用q的不同地址，如图2-5所示。

```
var rec_p[q] for each q ∈ Neigh_p : boolean init false ;

begin while #{q : rec_p[q] is false} > 1 do
            begin receive ⟨tok⟩ from q ; rec_p[q] := true end ;
      send ⟨tok⟩ to q_0 with rec_p[q_0] is false ;
      receive ⟨tok⟩ from q_0 ; rec_p[q_0] := true ;
      decide
end
```

附录A-1　面向控制表示法（树算法）

进程间的通信可以是*异步方式*，也可以是同步方式（见2.1节）。如果采用异步方式（如果不另行规定，总是假设采用这种方式）进行通信，发送操作总是能立即被执行。在被执行相应的接收操作来删除之前，消息一直存储在通信子系统中。在某一确定类型的消息可被接收之前，接收操作将执行进程挂起。如果采用同步方式进行通信，不会临时存储消息。在这种情况下，发送和接收操作都将执行进程挂起，这是因为发送操作和相应的接收操作的执行是同步的。在两个进程都准备就绪即将运行之前，无论是发送进程还是接收进程中哪一个首先开始操作，都会被挂起。接着消息交换发生，两个进程继续执行各自的算法。

面向控制表示法和事件驱动表示法。算法的总体控制结构可以用两种方式来表示：即，明确地以*面向控制*（control-oriented）的描述法，或是隐含地以*事件驱动*（event-driven）描述法。在算法中可以任意选用两种结构之一，但是在很多情况下，两种描述方法中，一种比另一种更方便。

一个算法的面向控制表示法包括变量的声明，**begin**和**end**之间所包括的以分号隔开的一系列语句。在这种情况下，局部算法的执行由这个语句列表中的单个执行组成。变量可以在语句列表的开始被初始化，我们也可以在声明该变量的时候给它赋初值。

作为例子，考虑树附录A-1的算法。它是树算法（图6-3所示的算法）的一个面向控制表示法。由于语句列表描述了局部算法的整个控制流，非确定性必须在程序中明确指明。在附录A-1的算法的接收语句中存在非确定性，因为没有相应的发送语句被明确指明。很难在发送

和接收消息之间做出非确定性选择，但是如果接收信道能检测消息的可用性，那么这个问题可以得到解决。

var $rec_p[q]$ for each $q \in Neigh_p$: boolean **init** false ;
$sent_p$, dec_p : boolean **init** false ;

$\mathbf{R}_p$: { A message ⟨**tok**⟩ has arrived }
begin receive ⟨**tok**⟩ from q ; $rec_p[q] := true$ **end**

$\mathbf{S}_p$: { $\#\{q : rec_p[q]$ is false$\} = 1$ and $sent_p = false$ }
begin send ⟨**tok**⟩ to q_0 with $rec_p[q_0]$ is false ;
$sent_p := true$
end

$\mathbf{D}_p$: { $\#\{q : rec_p[q]$ is false$\} = 0$ and $dec_p = false$ }
begin *decide* ; $dec_p := true$ **end**

附录A-2 事件驱动表示法（树算法）

if a message **<info**, a, b**>** is available on channel q
then begin receive **<info**, a, b**>**; ... **end**
else begin send ...**end**

用算法的事件驱动表示方法更容易表示非确定性（因为它是隐式的）。这种表示法包括变量的声明以及这些变量的初始化，以及紧接着的一系列行为。每个行为由布尔表达式（行为阈值，guard）和一条语句（行为体，body）组成。一个行为是可行的（或是可应用的）仅当行为阈的值为真。行为体可以被以任何次序执行任意次，但是仅当行为是可行的，才被选择执行。

附录A-2的算法是同一树算法事件驱动表示法的一个例子。因为每条语句都能被选择执行任意次（假定行为阈值为真），必须显式地进行编程，使得发送操作和判定操作仅被执行一次。在附录A-2的算法使用$sent_p$和dec_p标志位来做到这些。

我们没有做出与事件驱动程序执行相关的公平性假设；执行模型没有要求，一个无限长时间可行的行为最终要被执行。因此，程序员有责任确保没有实质性行为会遭受饥饿（因为程序的其他行为总是被选择，而它从未被选择过）。

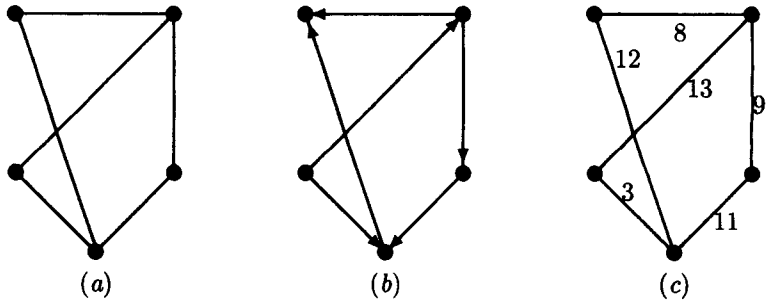
附录B 图 和 网 络

因为分布式系统的拓扑结构通常用图来表示，所以图论知识及其一些术语对于学习分布式算法是很有帮助的。而且，分布式算法的证明和数学论证有时也要用到图（参见8.2节）。

B.1节介绍了一些常用的定义和术语，B.2节论述了几类特定的图。

B.1 定义和术语

可以认为图是点（称为图的节点）的集合，其中一些点由线（称为边）连接，参见附录B-1。在有向图中，边是有方向的，用箭头表示；在加权图中，赋予每条边一个数值。



附录B-1 （a）无向图，（b）有向图，（c）加权图

B.1.1 无向图

无向图G是一个二元组$G = (V, E)$，其中：V称为G的节点集合；E是V中的无序二元组集，即，E中元素为二元组$\{u, v\}$，其中，$u, v \in V$，可简写成$uv \in E$，而不是$\{u, v\} \in E$。因为此二元组是无序的，所以$uv \in E$等价于$vu \in E$。边uv称为u（和v）的依附边。如果$uv \in E$，就称节点u和v是相邻的，或者称为近邻。节点的度定义为依附它的边数，或者是它的近邻的数目。如果所有节点都有相同的度（δ），则称该图为正则的。

节点v_0和v_k之间的长度为k的路径是节点的一个序列$P = <v_0, \ldots, v_k>$，其中，对于每一个$i < k$，满足$v_iv_{i+1} \in E$。在这条路径上，v_0为起点，v_k为终点。回路是一条起点等于终点的路径。如果一条路径中从v_0到v_k的所有节点都是不同的，则称该路径是简单的。如果一条回路中从v_0到v_k的所有节点都是不同的，则称该回路是简单的。

u和v之间的距离定义为u和v之间最短路径的长度，记为$d(u, v)$。图G的直径为图中任意两个节点之间的最大距离。如果一个无向图中任意两个节点间都存在一条路径，则称该无向图是连通的。如果一个无向图中不包含长度大于等于3的简单回路，则称该无向图是无环的。

一个图$G' = (V', E')$，如果$V' \subseteq V$，$E' \subseteq E$，则称其为G的子图（subgraph）。如果$V' = V$，则称G'为生成子图。如果$E' = \{uv \in E \mid u \in V' \wedge v \in V'\}$，则称$G'$为导出子图。$G$的连通分量（connected component）是G的最大连通导出子图G'，即，G'是一个连通的导出子图，但如果

增加节点来扩展V'，它的导出子图就不连通了。连通分量形成了图的一个划分，当且仅当整个图连通时，划分是由一个分量组成的。

如果一个图可以画在一个平面上，而不存在交叉边，则该图称为*平面图*。如果一个图可以画在一个平面上，而不存在交叉边，且所有节点都在图形边缘，则该图称为*外平面图*(outerplanar)。下面是平面图和外平面图的一些比较重要的性质:

(1) N个节点的平面图至多有$3N-6$条边。

(2) 一个平面图至少有一个度不大于5的节点。

(3) 一个外平面图至少有一个度不大于2的节点。

B.1.2 有向图

一个有向图G是一个二元组$G = (V, E)$，其中: V称为节点集合; E是V中元素的有序二元组的集合，即，E中元素为数对(u, v)，其中，$u, v \in V$。可简写成$uv \in E$，而不是$(u, v) \in E$。因为此二元组是有序的，所以$uv \in E$不等价于$vu \in E$。无向图的大多数定义稍做修改就变成了有向图的定义。

边uv称为u的*出边*，v的*入边*，如果$uv \in E$，u称为v的*出近邻*(out-neighbor)，v称为u的*入近邻*(in-neighbor)。节点的*入度*(in-degree)为其进入边的数目，*出度*(out-degree)为其输出边的数目，*度*为入度与出度之和。

节点v_0和v_k之间的长度为k的路径，为节点序列$P = \langle v_0, \ldots, v_k \rangle$，其中，对于每一个$i < k$，满足$v_i v_{i+1} \in E$。在这条路径上，$v_0$为起点，$v_k$为终点。*回路*(cycle)是一个起点等于终点的路径。和无向图一样，如果一条路径中从v_0到v_k的所有节点都是不同的，则称该路径是简单的。如果一条回路中从v_0到v_k的所有节点都是不同的，则称该回路是简单的。

从u到v之间的距离定义为u和v之间最短路径的长度，记为$d(u, v)$。图G的直径为图中任意两个节点间的最大距离。如果一个有向图中任意两个节点间都存在一条路径，则称该有向图为*强连通的*。如果一个有向图中不包含长度不小于2的简单回路，则称该有向图为*无环的*。

有向图中，子图、生成子图和导出子图的定义与无向图相同。

B.1.3 带权图

一个(有向或无向)图，如果对于每个$uv \in E$的二元组，u，v，都定义一个数值ω_{uv}，则称该图是*带权的*(weighted)。如果对于每个二元组，$\omega_{uv} = \omega_{vu}$，则称权值赋值是*对称的*。在带权图中，路径的权值定义为路径上所有边的权值总和。

如果假设一个图没有赋给权值，则称其为*不带权的*(unweighted)。

B.2 常用图

本节定义和讨论几类在分布式算法研究中经常遇到的图，即，环、树、森林、网格、圆环、团和超立方体。正如从本书中很多结果可以证明的，分布式系统中的计算在具有受限拓扑结构的网络上，能够比在任意(但不连通)拓扑网络上更有效地进行。

最终，设存在图$G = (V, E)$，N为其节点数，D为其直径。

B.2.1 环

环拓扑是节点的一个圆形排列，经常用于分布式计算的控制拓扑。一些网络（如，令牌环网[Tan96，4.3节]）将节点按环形进行物理排列，但在其他情况下，环主要用作嵌入控制结构。为了以循环方式给每个进程它的轮次（执行某些功能），进程在它们之间循环传递消息，v_0发送消息到v_1，然后v_1到v_2，等等，直到v_{N-1}，v_{N-1}再将消息发送到v_0。

定义B.1 环是一个无向、连通且度为2的正则图。

环的可以用几种方式来表述特征。

定理B.2 对于一个无向图G，以下是等价的：

（1）G是一个环；

（2）从V到$\{0, \ldots, N-1\}$存在一对一映射，满足节点i的近邻为节点$i-1$和$i+1 \pmod N$。

为了在其他网络拓扑中利用环拓扑的优点，有时一个生成环可以被定义为一个图$G = (V, E)$，也就是说，选择一个集合$E' \subseteq E$，满足(V, E')为一个环。但是，令人遗憾的是，并不是每个图都包含一个生成环（哈密尔顿回路（Hamiltonian cycle）），判定一个给定图是否包含生成环是一个NP-完全问题[GJ79]。然而，可以定义一个虚拟环（virtual ring），在这种情况下，环中连续节点在原图中不必是近邻。Bakker和Van leeuwen[BL93]描述了一个构造虚拟环的分布式算法，其中环的连续节点间的距离至多为3。

B.2.2 树、森林和星

树是一个包含最少的连接其所有节点的边的图（见定理B.4），所以，树形网络上的计算可以有非常低的通信复杂度。对于分散式计算，造成对树的计算的复杂度低的另一个因素是存在一个针对树的有效分散波动算法（图6-3所示的算法）。

定义B.3 树是一个无向、连通的无环图。

可以用下列方法描述树的特点[CLR90]。

定理B.4 对于一个无向图G，以下是等价的：

（1）G是一棵树。

（2）任意两节点间存在惟一一条简单路径。

（3）G是连通的，但删除任何一条边就变成不连通的。

（4）G是连通的，且$|E| = N-1$。

（5）G是无环的，且$|E| = N-1$。

（6）G是无环的，但任意增加一条边就变成有环的。

一棵树$T = (V, E)$，如果存在惟一指定的根节点r，则称该树是有根的。如果u是v和r之间的（惟一）简单路径上的节点，则u称为v的祖先，v称为u的后代。如果u和v是近邻，v称为u的子节点，u称为v的父节点。u的后代所导出的子图是一棵有根树（根为u），称为u的子树。树的深度定义为从根到任一节点的最大简单路径长度。

每个连通图$G = (V, E)$都包含一棵生成树，即，选择一个集合$E' \subseteq E$，满足(V, E')是一棵树，而且，如果G本身不是一棵树，可以按不同方法选择生成树。E'中的边称为树边，$E \setminus E'$中的边称为非树边。

下面是几类在分布式计算中比较重要的生成树:

(1) 小直径生成树　如果必须选择生成树来最小化计算它所必须的总的时间，那么就希望其直径尽可能小。定理4.2证明中构造的最优汇集树的直径至多是整个网络直径的2倍，可以用第4章和12.4节描述的算法进行构造。

(2) 最小加权生成树　生成树中边的数目总是等于$N-1$（参见定理B.4 (4)），但在加权图中，树的权值（树中每条边的权值的和）通常对于每个可能的生成树并不相同。如果一个生成树计算的总通信成本必须很低，则必须选择子树以使树的权值最小。如果所有边权值都不同，则最小加权生成树是惟一的（命题7.18）。计算这种惟一的树的分布式算法由Gallager等人[GHS83]给出，并在7.3.2节进行了讨论。

(3) 度受限的生成树　如果每个节点的计算开销必须很小，则必须选择节点的度较小的生成树。这种树的分布式构造算法由Korach等人[KMZ85]给出。

(4) 深度优先搜索树　如果一棵生成树的每条非树边都连接一个节点和该节点的一个后代，则称其为深度优先搜索树。深度优先搜索树用于很多对图操作的算法中，例如，检测平面性或双连通性的算法，也用于压缩路由的区间标号模式的构造算法，参见4.4.2节。构造深度优先搜索树的分布式算法在6.4节给出。

森林是对树的推广，星是特殊类型的树。由一些孤立树组成的图称为森林。

定义B.5　*森林是无向无环图。*

定理B.6　*对于一个无向图G，以下是等价的:*

(1) G是一个森林;

(2) 任何两个节点间至多存在一条简单路径;

(3) G的每个连通分量都是一棵树;

(4) 如果删除任意一条边，G的连通分量的数目就减小1;

(5) G的连通分量的数目等于$|V|-|E|$。

如果森林的每棵树都指定一个根节点，则称该森林是有根的，每个图都有一个生成森林，由图的每个连通分量的生成树组成。

星是一个带有中心的图，所有其他节点都只和中心相连接。

定义B.7　*星是一个深度为1的有根树，星的根称为中心。*

星并不是一种常用的分布式系统的物理连接拓扑，而是在由中心进程控制的计算中用作虚拟拓扑。在这种计算中，所有进程只与中心节点通信，也就是说只使用整个网络中为星的（生成）子图，见图6-6所示的算法。

定理B.8　*对于一个无向图G，以下是等价的:*

(1) G是一个星;

(2) G是连通的，至多包含一个度大于1的节点。

B.2.3 团

团，又称为完全图，其中每两个节点间都由一条边直接相连。

定义B.9　*团是直径为1的图。*

和星一样，团很少用作物理连接拓扑，之所以这样是因为图中边的数目大，需要大量的硬件。在大部分网络中，并不是每对节点都直接相连，不连通的节点间的通信需要这些节点

间的路径上的所有节点共同参与。但是，如果这种参与在系统较低层上不可见地实现，则网络在较高层上可以被认为是团。

定理B.10 对于一个无向图G，以下是等价的：

（1）G是一个团；

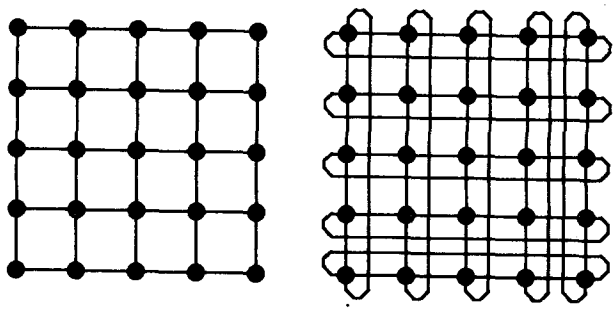
（2）$E=\{uv:\ u,\ v\in V,\ u\neq v\}$；

（3）$|E|=(1/2)N(N\text{-}1)$；

（4）每个节点的度为$N-1$。

B.2.4 网格和圆环

一个$n\times n$的网格，有$N=n^2$个节点，排列成n行，每行有n个节点，每个节点和它上、下、左、右的节点相连，见附录B-2。$n\times n$的圆环（torus）与网格相似，但除了每行最左边和最右边的节点是相连的，每列最上边和最下边的节点是相连的，见图附录B-2。网格和圆环是设计多处理器计算机时最常用的拓扑，因为每个节点的度至多为4，可以在Transputer芯片上物理地实现这些拓扑。这种拓扑结构非常适合进行矩阵计算。



附录B-2 网格和圆环

为了对这些拓扑进行形式定义，记$\mathbb{Z}_n$为模n的整数的集合。

定义B.11 $n\times n$网格是一个有$N=n^2$个节点的图，每个节点都可以用$\{(i,j):\ 0\leqslant i,j<n\}$中的元素惟一地标号，用这种方法，即，节点$(i,j)$和$(i',j')$是相邻的，当且仅当$(i=i')\wedge(j=j'+1)$，或$(i=i')\wedge(j=j'-1)$，或$(i=i'+1)\wedge(j=j')$，或$(i=i'-1)\wedge(j=j')$。

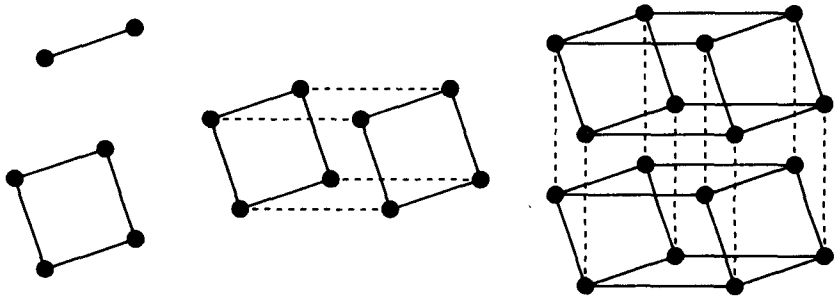
$n\times n$圆环是一个有$N=n^2$个节点的图，每个节点都可以用$\{(i,j):\ i,j\in\mathbb{Z}_n\}$中的元素惟一地标号，用这样一种方法，即，节点$(i,j)$和$(i',j')$是相连的，当且仅当$(i=i')\wedge(j=j'+1)$，或$(i=i')\wedge(j=j'-1)$，或$(i=i'+1)\wedge(j=j')$，或$(i=i'-1)\wedge(j=j')$。（这里的运算是取模$n$）

$n\times n$网格有$2n(n-1)$条边，包含度为2、3和4的节点，它的直径为$2(n-1)$。$n\times n$圆环有$2n^2$条边，每个节点的度为4，具有正则性，直径为$2\lfloor n/2\rfloor$。

B.2.5 超立方体

与网格和圆环类似，超立方体多用来设计多处理器计算机。它们结合了合理小的直径和合理小的度，直径和度都等于$\log N$，这里N为节点数。Saad和Schultz[SS88]研究了这类图的拓扑性质。

定义B.12 *n*-维超立方体是一个有$N=2^n$个节点的图，每个节点都可以赋给集合$\{(b_0, \ldots, b_{n-1}): b_i=0, 1\}$中的一个惟一元素，用这样一种方法，即，$E=\{uv: u和v的标号只有一位不同\}$。



附录B-3　1，2，3和4维超立方体

超立方体是度为*n*的正则图，直径为*n*。可以从不同的角度观察超立方体，一些有趣的图论结果可参见[SS88]。首先，定义B.12中的谈及的标号远不止一种，如果*G*是一个*n*维超立方体，则有$2^n n!$种这类标号。

可以通过取两个*n*维超立方体并连接对应的节点来构造（*n*+1）-维超立方体，在附录B-3中，对应节点间的边用虚线表示。

如果固定一个超立方体的节点的一个标号，不难证明下面的结论：如果*u*和*v*是相邻的，则*u*的近邻和*v*的近邻以一对一的方式连接，也就是说，*v*的每个近邻都只和*u*的一个近邻相邻，反之亦然。但是比较困难的是证明，（利用一些附加的，相当平凡的条件），相邻节点的近邻之间的一对一连接是否足以使一个图成为一个超立方体。

定理B.13 对于一个无向图*G*，以下是等价的：

（1）*G*是一个*n*维超立方体；

（2）$N=2^n$，*G*是正则的，度为*n*，*G*是连通的，对于每对相邻节点*u*，*v*，它们的近邻都以一对一的方式连接。

总结。附录B-4总结了本节讨论的各类图的主要特点。

图	度	|E|	D	注
环	2	N	$\lfloor \frac{N}{2} \rfloor$	
树	多种	$N-1$	$<N$	
森林	多种	$<N$		
星	1或$N-1$	$N-1$	2或1	$D=1$当且仅当$N=2$
团	$N-1$	$\frac{1}{2}N(N-1)$	1	
网格（$n\times n$）	2，3，4	$2n(n-1)$	$2(n-1)$	$N=n^2$
圆环（$n\times n$）	4	$2n^2$	$2\lfloor \frac{n}{2} \rfloor$	$N=n^2$
超立方体（*n*维）	n	$2^{n-1}n$	n	$N=2^n$

附录B-4　各类图

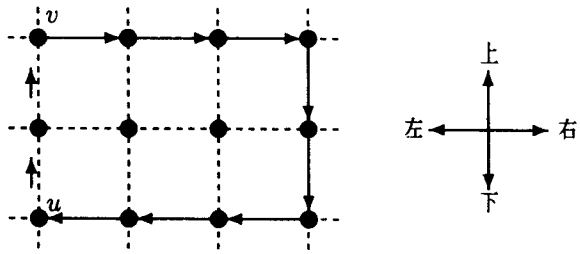
B.3 方向侦听

Santoro[San84]发现，分布式计算的通信复杂度除了受实际拓扑的影响外，还受以下因素影响：

（1）*拓扑意识*：为了利用某种特定拓扑的优点，有时进程需要“知道”它们被连接到哪种拓扑中。例如，选择问题（第7章）可以用团网络解决，只需$O(N\log N)$条消息。然而，如果在团中采用任意连接网络的最优算法，要交换$\Omega(N^2)$条消息。因此，为了利用全连接的优点，进程必须知道它们处于团中。

（2）*方向侦听*：如果依附于每个节点的边都用它们在网络中所指的“方向”进行标号，则网络中信息的路由就会更加有效率。有了这种信息，对于那些只知道存在长路径的节点，我们可以找到更短（通常是最优的）路径通向它。

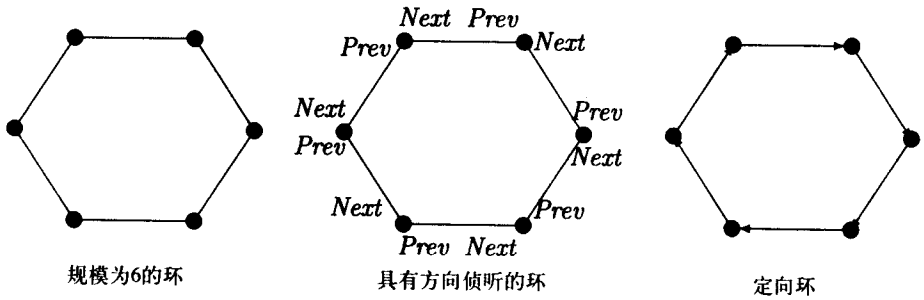
附录B-5描述了网格网络的一部分，从中可以看出方向侦听的优点。节点v已经发出了对某些资源的请求，该请求被几个节点转发（这些节点没有所请求的资源），最后到达拥有资源的u，因为u发现请求信息已经向右转发3次，向下2次，向左3次，于是u向上传送应答。应答被通过长度为2的路径发送，而不是沿着请求信息经过的路径（8条边）向回转发。显然，只有节点“知道”它们的每条依附边所指的方向，才能发现一个更短的返回路径。



附录B-5 探索方向侦听

当然，对于一个图，拓扑不同，“方向”的意义也不同。在这种情况下，假设对于每个节点，那个节点的边都标以对于该类图“有意义的方向”。如果S表示标号，对于节点v中的边uv，可分配的标号记作$S_v(u)$，同样的边在节点u中，则标号为$S_u(v)$。现有标号的集合对于每个节点来说是相同的，集合的大小$\mathcal{L}$等于网络的度，因为对于每个节点，边的标号是惟一的。为了进行方向侦听，还必须验证另外一个全局一致性条件。下面分别针对四种不同的图解释其各自方向侦听的意义。

环：如同度为2的正则环，环上只有两个方向，我们称之为“*Prev*”（前一个）和“*Next*”。如果u是v后的“*Next*”节点，则v是u之前的“*Prev*”节点。如果每对相邻节点都满足这种关系，则定向是全局一致的，见附录B-6。



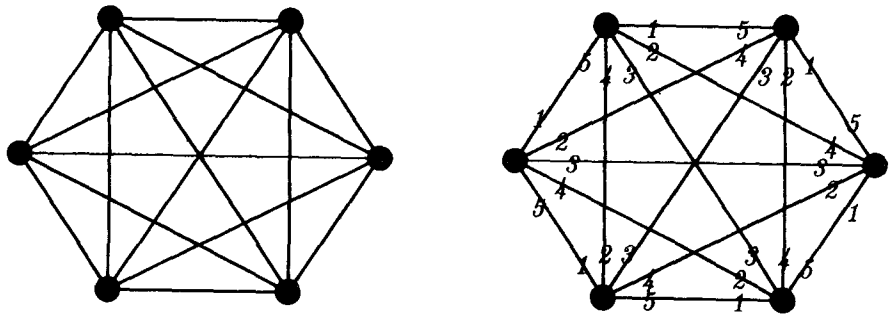
规模为6的环　　具有方向侦听的环　　定向环

附录B-6 环的方向侦听

定义B.14　环的标号就是在每个具有集合{*Prev*, *Next*}中的不同标号的节点处，对依附于那个节点的边赋值。如果对于所有的近邻节点*u*和*v*，有

$$\mathcal{S}_u(v) = Prev <=> \mathcal{S}_v(u) = Next$$

则环上的标号$\mathcal{S}$形成方向侦听。



附录B-7　团的方向侦听

环中的方向侦听不是惟一的，有两种边标号方式满足这些限制。假定环中总存在方向侦听。显然，如果一个节点通过它的*Next*边发送一个消息，则接收到该消息的每个节点再通过各自的*Next*边继续转发该消息，在消息返回第一个节点之前，所有的节点都会收到该消息。通常，这是通信信道被使用的惟一方向。通常假设环是有向的，如附录B-6所示。

团：带有*N*个节点的团的度为*N*−1，它的方向被标号为从1到*N*−1，即，$\mathcal{L} = \{1, \ldots, N-1\}$。如果通过将边*uv*标以（在*u*中）沿着环从*u*到*v*的距离，从而确定一个团中的有向哈密尔顿回路（Hamiltonian Cycle），则这种标号就形成了方向侦听。

定义B.15：团的标号就是在每个具有集合{1, …, *N*−1}中的不同标号的节点处，对依附于那个节点的边进行赋值。如果团的节点可以被标号为从0到*N*−1，标号采用这样的方法，即，对于每个*i*和*j*，$S_i(_j) = (j - i) \bmod N$，则团上的标号*S*形成方向侦听。

附录B−7中表示的是一个带有方向侦听的团。团的方向侦听也不是惟一的，实际上有(*N*−1)!种不同的标号方法可以形成团的方向侦听。Loui等人[LMW86]证明，如果方向侦听可用，团的选择问题用O(*N*)条消息就可以解决。

圆环：圆环的四个方向分别为上、下、左和右，缩写为*U*、*D*、*R*、*L*。

定义B.16　圆环的标号就是在每个具有集合{*U*、*D*、*R*、*L*}中的不同标号的节点处，对依附于那个节点的边进行赋值。如果节点可以按下面方式用集合{ (*i*, *j*): *i*, *j* ∈ $\mathbb{Z}_n$}的元素重命名，

(1) $S_{(i,j)}((i', j')) = U \Rightarrow (i' = i) \wedge (j' = j+1)$

(2) $S_{(i,j)}((i', j')) = D \Rightarrow (i' = i) \wedge (j' = j-1)$

(3) $S_{(i,j)}((i', j')) = L \Rightarrow (i' = i-1) \wedge (j' = j)$

(4) $S_{(i,j)}((i', j')) = R \Rightarrow (i' = i+1) \wedge (j' = j)$

则圆环上的标号*S*形成方向侦听。

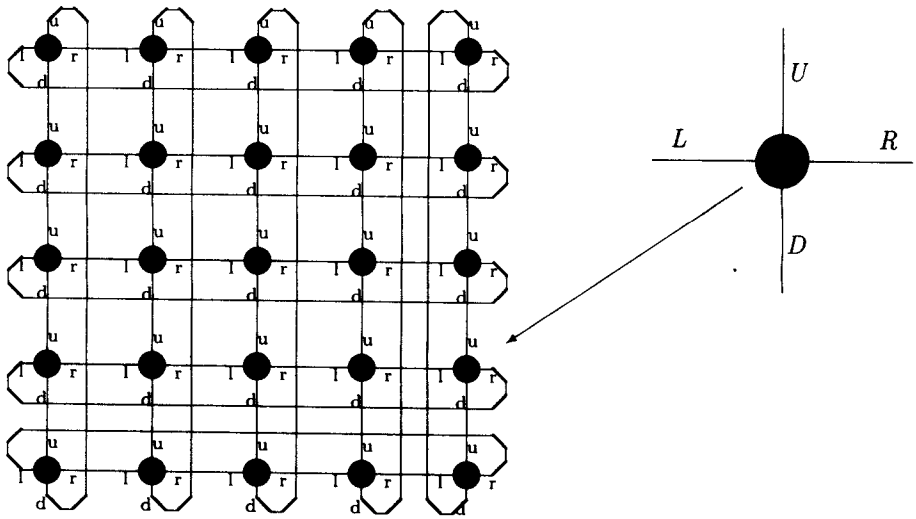
圆环的方向侦听不是惟一的，如果$n > 4$，则存在8种不同的能进行圆环方向侦听的标号，

附录B-8所示就是一个带方向侦听的圆环。

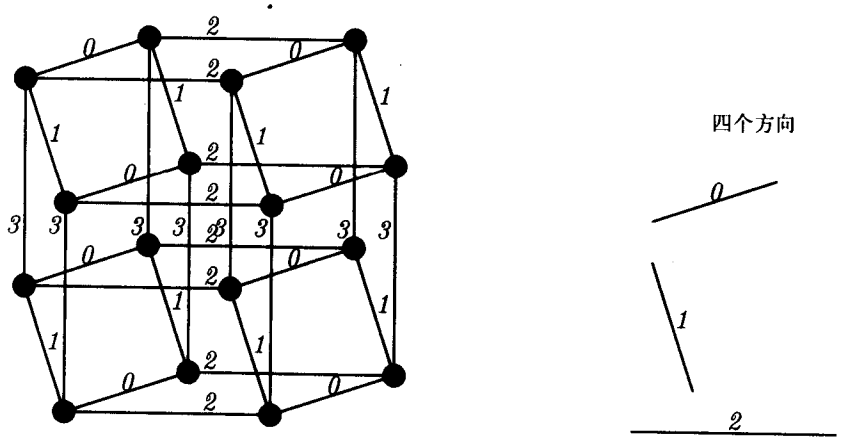
超立方体：n-维超立方体的维数从0到$n-1$进行标号，如果所有平行边被标号为相同的数字，则存在一个方向侦听。

定义B.17 *n-维超立方体的标号就是在每个具有集合$\{0, \ldots, n-1\}$中的不同标号的节点处，对依附于那个节点的边进行赋值。如果节点能用集合$\{(b_0, \ldots, b_{n-1}): b_i = 0, 1\}$的元素重命名，并满足如果标号$S_{(b_0, \ldots, b_{n-1})}((c_0, \ldots, c_{n-1})) = i$，则$(b_0, \ldots, b_{n-1})$和$(c_0, \ldots, c_{n-1})$只有第$i$位不同，此时，超立方体的标号$S$形成方向侦听。*

超立方体的方向侦听不是惟一的，对于n-维超立方体有$n!$种不同的给超立方体方向侦听的边标号。定义表明对于连通的节点u和v，$S_u(v) = S_v(u)$，因此，附录B-9中，每条边只有一个标号。



附录B-8 圆环上的方向侦听



附录B-9 超立方体上的方向侦听

图的类型	度	标号集 $\mathcal{L}$	方向侦听数	给出一致性的定义
环	2	{*Prev*、*Next*}	2	定义B.14
团	$N-1$	{1, ..., $N-1$}	$(N-1)!$	定义B.15
圆环	4	{*U*、*D*、*L*、*R*}	8（如果 $n>4$）	定义B.16
超立方体	n	{0, ...,$n-1$}	$n!$	定义B.17

附录B-10　图的方向侦听

附录B-10总结了四类图的方向侦听的性质，Flocchini *et al*[FMS98]给出了方向侦听的一般定义及其性质的正式描述。对于本节中所描述的以及其他更多的图，方向侦听都可以用第11章的群的概念适当地加以描述。计算超立方体和圆环上的方向侦听的分布式算法参见[FMS98]和[Tel94]。

习题

B.1节

B.1　证明无向图的以下结果：

n个节点的连通至少有$n-1$条边。

n个节点的无环图至多有$n-1$条边。

称无向图为二分图（bipartite），如果图中的节点可用两种颜色（红和黑）着色，使得相邻节点着不同颜色。

B.2　证明无向图为二分图，当且仅当图中不含奇数长度的回路。

B.2节

B.3　证明（1）规模为偶数的环，（2）树，和（3）超立方体都是二分图。

B.4　证明定理B.4和定理B.6。（提示：利用B.1）。

B.5　（项目）定理B.13（2）中的四个条件，其中任意一个可以删除吗？

B.3节

B.6　证明团上只有$(N-1)!$种不同的方向侦听。

证明n-维超立方体团上只有$n!$种不同的方向侦听。

B.7　设S是团上的标号。对于路径$P=\langle v_0,\cdots,v_k\rangle$，定义$Sum(P)=\mathcal{S}_{v_0}(v_1)+\cdots+\mathcal{S}_{v_{k-1}}(v_k)$。证明以下两种阐述等价。

（1）S是方向侦听。

（2）对于每条路径P，P是回路，当且仅当$Sum(P)\equiv 0 \pmod N$

参考文献

[ABND+90] ATTIYA, H., BAR-NOY, A., DOLEV, D., PELEG, D., AND REISCHUK, R. Renaming in an asynchronous environment. *J. ACM* **37** (1990), 524–548.

[ABNLP90] AWERBUCH, B., BAR-NOY, A., LINIAL, N., AND PELEG, D. Improved routing strategies with succinct tables. *J. Algorithms* **11** (1990), 307–341.

[ACT97] AGUILERA, M. K., CHEN, W., AND TOUEG, S. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In proc. *11th Int. Workshop on Distributed Algorithms* (1997), M. Mavronicolas and P. Tsigas (eds.), vol. 1320 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 126–140.

[AG85] AWERBUCH, B. AND GALLAGER, R. G. Distributed BFS algorithms. In proc. *Foundations of Computation Theory* (1985), pp. 250–256.

[AG91] AFEK, Y. AND GAFNI, E. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM J. Comput.* **20**, 2 (1991), 376–394.

[AH93] ANAGNOSTOU, E. AND HADZILACOS, V. Tolerating transient and permanent failures. In proc. *Int. Workshop on Distributed Algorithms* (Lausanne, 1993), A. Schiper (ed.), vol. 725 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 174–188.

[Ahu90] AHUJA, M. Flush primitives for asynchronous distributed systems. *Inf. Proc. Lett.* **34** (1990), 5–12.

[AKY90] AFEK, Y., KUTTEN, S., AND YUNG, M. Memory-efficient self stabilizing protocols for general networks. In proc. *4th Int. Workshop on Distributed Algorithms* (Bari, 1990), J. van Leeuwen and N. Santoro (eds.), vol. 486 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 15–28.

[ALSY90] AFEK, Y., LANDAU, G. M., SCHIEBER, B., AND YUNG, M. The power of multimedia: Combining point-to-point and multiaccess networks. *Inf. Comput.* **84**, 1 (1990), 97–118.

[ALSZ89] ATTIYA, H., LEEUWEN, J. VAN, SANTORO, N., AND ZAKS, S. Efficient elections in chordal ring networks. *Algorithmica* **4** (1989), 437–466.

[Ang80] ANGLUIN, D. Local and global properties in networks of processors. In proc. *Symp. on Theory of Computing* (1980), pp. 82–93.

[AS85] ALPERN, B. AND SCHNEIDER, F. B. Defining liveness. *Inf. Proc. Lett.* **21** (1985), 181–185.

[Att87] ATTIYA, H. Constructing efficient election algorithms from efficient traversal algorithms. In proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), vol. 312 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 377–344.

[AW98] ATTIYA, H. AND WELCH, J. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[Awe85a] AWERBUCH, B. Complexity of network synchronization. *J. ACM* **32** (1985), 804–823.

[Awe85b] AWERBUCH, B. A new distributed depth first search algorithm. *Inf. Proc. Lett.* **20** (1985), 147–150.

[Awe87] AWERBUCH, B. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In proc. *Symp. on Theory of Computing* (1987), pp. 230–240.

[BA84] BEN-ARI, M. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst.* **6**, 3 (1984), 333–344.

[Bal90] BAL, H. *Programming Distributed Systems.* Prentice-Hall, 1990.

[Bar96] BARBOSA, V. C. *An Introduction to Distributed Algorithms.* MIT Press, 1996 (365 pp.).

[BB88] BRASSARD, G. AND BRATLEY, P. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.

[BB89] BEAME, P. W. AND BODLAENDER, H. L. Distributed computing on transitive networks: The torus. In proc. *Symp. on Theoretical Aspects of Computer Science* (1989), B. Monien and R. Cori (eds.), vol. 349 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 294–303.

[Bel76] BELSNES, D. Single-message communication. *IEEE Trans. Comput.* **COM–24**, 2 (1976), 190–194.

[BGM93] BURNS, J. E., GOUDA, M. G., AND MILLER, R. E. Stabilization and pseudo-stabilization. *Distributed Comput.* **7**, 1 (1993), 35–42.

[BGP92] BERMAN, P., GARAY, J. A., AND PERRY, K. J. Optimal early stopping in distributed consensus. In proc. *6th Int. Workshop on Distributed Algorithms* (Haifa, 1992), A. Segall and S. Zaks (eds.), vol. 647 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 221–237.

[BGS87] BARATZ, A., GOPAL, I., AND SEGALL, A. Fault tolerant queries in computer networks. In proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), vol. 312 of *Lecture Notes in*

Computer Science, Springer-Verlag, pp. 30–40.

[BHR92] BRZEZINSKI, J., HÉLARY, J.-M., AND RAYNAL, M. Deadlock models and general algorithm for distributed deadlock detection. Research report, IRISA, Rennes, 1992. Submitted for publication.

[BIZ89] BAR-ILAN, J. AND ZERNIK, D. Random leaders and random spanning trees. In proc. *3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), vol. 392 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–12.

[BL93] BAKKER, E. M. AND LEEUWEN, J. VAN. Uniform d–emulations of rings, with an application to distributed virtual ring construction. *Networks* **23** (1993), 237–248.

[BLT91] BAKKER, E. M., LEEUWEN, J. VAN, AND TAN, R. B. Linear interval routing. *Algorithms Review* **2**, 2 (1991), 45–61.

[BLT93] BAKKER, E. M., LEEUWEN, J. VAN, AND TAN, R. B. Prefix routing schemes in dynamic networks. *Computer Networks and ISDN Systems* **26** (1993), 403–421.

[BMZ90] BIRAN, O., MORAN, S., AND ZAKS, S. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *J. Algorithms* **11** (1990), 420–440.

[Bod86] BODLAENDER, H. L. Distributed computing: Structure and complexity. Ph.D. thesis, Dept Computer Science, Utrecht University, The Netherlands, 1986 (297 pp.).

[Bod88] BODLAENDER, H. L. A better lower bound for distributed leader finding in bidirectional asynchronous rings of processors. *Inf. Proc. Lett.* **27** (1988), 287–290.

[Bod91a] BODLAENDER, H. L. New lower bound techniques for distributed leader finding and other problems on rings of processors. *Theor. Comput. Sci.* **81** (1991), 237–256.

[Bod91b] BODLAENDER, H. L. Some lower bound results for decentralized extrema-finding in rings of processors. *J. Comput. Syst. Sci.* **42**, 1 (1991), 97–118.

[Bou83] BOURNE, S. R. *The Unix System.* Addison-Wesley, 1983 (351 pp.).

[BP88] BURNS, J. E. AND PACHL, J. Uniform self-stabilizing rings. In proc. *Aegean Workshop on Computing* (1988), J. H. Reif (ed.), vol. 319 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 391–400.

[BT85] BRACHA, G. AND TOUEG, S. Asynchronous consensus and broadcast protocols. *J. ACM* **32** (1985), 824–840.

[CB89] CHARRON-BOST, B. Mesures de la concurrence et du parallélisme des calculs répartis. Ph.D. thesis, Université Paris VII, 1989 (129 pp.).

[CBMT96] CHARRON-BOST, B., MATTERN, F., AND TEL, G. Synchronous, asynchronous, and causally ordered communication. *Distributed Comput.* **9** (1996), 173–191.

[CBT97] CHARRON-BOST, B. AND TEL, G. Calculs approchés de la borne inférieure de valeurs réparties. *Inf. Théor. Appl.* **31**, 4 (1997), 305–330.

[CCGZ90] CHOU, C. T., CIDON, I., GOPAL, I. S., AND ZAKS, S. Synchronizing asynchronous bounded delay networks. *IEEE Trans. Commun.* **38**, 2 (1990), 144–147.

[CD88] COULOURIS, G. F. AND DOLLIMORE, J. *Distributed Systems: Concepts and Design.* Addison-Wesley, 1988 (366 pp.).

[CD91] COAN, B. A. AND DWORK, C. Simultaneity is harder than agreement. *Information and Computation* **91**, 2 (1991), 205–231.

[Cha82] CHANG, E. J.-H. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Softw. Eng.* **SE–8** (1982), 391–401.

[Che83] CHEUNG, T.-Y. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Trans. Softw. Eng.* **SE–9** (1983), 504–512.

[CHT96] CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. The weakest failure detector for solving consensus. *J. ACM* **43**, 4 (1996), 685–722.

[Cid88] CIDON, I. Yet another distributed depth-first-search algorithm. *Inf. Proc. Lett.* **26** (1988), 301–305.

[CL85] CHANDY, K. M. AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* **3**, 1 (1985), 63–75.

[CLR90] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms.* McGraw-Hill/MIT Press, 1990 (1028 pp.).

[CM82] CHANDY, K. M. AND MISRA, J. Distributed computation on graphs: Shortest path algorithms. *Commun. ACM* **25**, 11 (1982), 833–838.

[CM88] CHANDY, K. M. AND MISRA, J. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988 (516 pp.).

[CMH83] CHANDY, K. M., MISRA, J., AND HAAS, L. M. Distributed deadlock detection. *ACM Trans. Comput. Syst.* **1**, 2 (1983), 144–156.

[CR79] CHANG, E. J.-H. AND ROBERTS, R. An improved algorithm for decentralized extrema finding in circular arrangements of processes. *Commun. ACM* **22** (1979), 281–283.

[Cri89] CRISTIAN, F. Probabilistic clock synchronization. *Distributed Computing* **3** (1989), 146–158.

[CT90] CRITCHLOW, C. AND TAYLOR, K. The inhibition spectrum and the

achievement of causal consistency. Tech. rep. 90-1101, Dept Computer Science, Cornell Univ., 1990.

[CT96] CHANDRA, T. D. AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM* **43**, 2 (1996), 225–267.

[CV90] CHANDRASEKARAN, S. AND VENKATESAN, S. A message-optimal algorithm for distributed termination detection. *J. Parallel and Distributed Computing* **8**, 3 (1990), 245–252.

[Dav88] DAVIDSON, J. *An Introduction to TCP/IP.* Springer-Verlag, 1988.

[DDS87] DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *J. ACM* **34** (1987), 77–97.

[DFF+82] DOLEV, D., FISCHER, M. J., FOWLER, R., LYNCH, N. A., AND STRONG, H. R. An efficient algorithm for Byzantine agreement without authenication. *Information and Control* **52** (1982), 257–274.

[DFG83] DIJKSTRA, E. W., FEIJEN, W. H. J., AND GASTEREN, A. J. M. VAN. Derivation of a termination detection algorithm for distributed computations. *Inf. Proc. Lett.* **16**, 5 (1983), 217–219.

[DHS84] DOLEV, D., HALPERN, J. Y., AND STRONG, H. R. On the possibility and impossibility of achieving clock synchronization. In proc. *Symp. on Theory of Computing* (1984), pp. 504–511.

[Dij68] DIJKSTRA, E. W. Co-operating sequential processes. In *Programming Languages*, F. Genuys (ed.), Academic Press, 1968, pp. 43–112.

[Dij74] DIJKSTRA, E. W. Self-stabilizing systems in spite of distributed control. *Commun. ACM* **17** (1974), 643–644.

[Dij82] DIJKSTRA, E. W. Self-stabilization in spite of distributed control. In *Selected Writing on Computing: A Personal Perspective*, Springer-Verlag, 1982, pp. 41–46.

[Dij87] DIJKSTRA, E. W. Shmuel Safra's version of termination detection. EWD-Note 998, 1987.

[DKR82] DOLEV, D., KLAWE, M., AND RODEH, M. An $O(N \log N)$ unidirectional distributed algorithm for extrema-finding in a circle. *J. Algorithms* **3** (1982), 245–260.

[DLM+78] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* **21**, 11 (1978), 966–975.

[DLS88] DWORK, C., LYNCH, N. A., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *J. ACM* **35**, 2 (1988), 288–323.

[Dob99] DOBREV, S. Linear election on unoriented hypercubes and graphs of constant diameter with any sense of direction. Tech. rep., Inst. of Informatics, Comenius Univ., Bratislava, 1999.

[DRS82] DOLEV, D., REISCHUK, R., AND STRONG, H. R. Eventual is earlier than immediate. In proc. *23rd Foundations of Computation Theory* (1982), pp. 196–203.

[DRT98] DOBREV, S., RUŽIČKA, P., AND TEL, G. Time and bit optimal broadcasting on anonymous unoriented hypercubes. In proc. *5th Colloq. on Structural Information and Communication Complexity* (1998), L. Gargano and D. Peleg (eds.), Carleton Scientific Press, pp. 173–187.

[DS80] DIJKSTRA, E. W. AND SCHOLTEN, C. S. Termination detection for diffusing computations. *Inf. Proc. Lett.* **11**, 1 (1980), 1–4.

[DS83] DOLEV, D. AND STRONG, H. R. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.* **12**, 4 (1983), 656–666.

[ElG85] ELGAMAL, T. A public key cryptosystem and a signature scheme based on discrete logrithms. *IEEE Trans. Inf. Theory* **IT-31**, 4 (1985), 469–472.

[EMZ96] EILAM, T., MORAN, S., AND ZAKS, S. A lower bound for linear interval routing. In proc. *10th Int. Workshop on Distributed Algorithms* (1996), O. Babaoğlu and K. Marzullo (eds.), vol. 1151 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 191–205.

[FG94] FREIGNIAUD, P. AND GAVOILE, C. Interval routing schemes. Research Report 94-04, LIPS-ENS, Lyon, 1994.

[FGNT98] FLAMMINI, M., GAMBOSI, G., NANNI, U., AND TAN, R. B. Multidimensional interval routing schemes. *Theor. Comput. Sci.* **205** (1998), 115–133.

[FGS93] FLAMMINI, M., GAMBOSI, G., AND SALOMONE, S. Boolean routing. In proc. *Int. Workshop on Distributed Algorithms* (Lausanne, 1993), A. Schiper (ed.), vol. 725 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 219–233.

[Fin79] FINN, S. G. Resynch procedures and a fail-safe network protocol. *IEEE Trans. Commun.* **COM–27** (1979), 840–845.

[FJ88] FREDERICKSON, G. N. AND JANARDAN, R. Designing networks with compact routing tables. *Algorithmica* **3** (1988), 171–190.

[FL82] FISCHER, M. J. AND LYNCH, N. A. A lower bound for the time to assure interactive consistency. *Inf. Proc. Lett.* **14** (1982), 183–186.

[FL84] FREDERICKSON, G. N. AND LYNCH, N. A. The impact of synchronous communication on the problem of electing a leader in a ring. In proc. *Symp. on Theory of Computing* (Washington DC, 1984), ACM, pp. 493–503.

[FLMS95] FLAMMINI, M., LEEUWEN, J. VAN, AND MARCHETTI-SPACCAMELA, A. The complexity of interval routing on random graphs. Tech. rep. UU-CS-1995-16, Dept Computer Science, Utrecht University, The Netherlands, 1995.

[FLP85] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM* **32** (1985), 374–382.

[FMS98] FLOCCHINI, P., MANS, B., AND SANTORO, N. Sense of direction: Definitions, properties, and classes. *Networks* **32** (1998), 165–180.

[FR82] FRANCEZ, N. AND RODEH, M. Achieving distributed termination without freezing. *IEEE Trans. Softw. Eng.* **SE–8**, 3 (1982), 287–292.

[Fra80] FRANCEZ, N. Distributed termination. *ACM Trans. Program. Lang. Syst.* **2**, 1 (1980), 42–55.

[Fra82] FRANKLIN, W. R. On an improved algorithm for decentralized extrema finding in circular configurations of processors. *Commun. ACM* **25**, 5 (1982), 336–337.

[Fra86] FRANCEZ, N. *Fairness.* Springer-Verlag, 1986 (295 pp.).

[Fre85] FREDERICKSON, G. N. A single source shortest path algorithm for a planar distributed network. In proc. *Symp. on Theoretical Aspects of Computer Science* (Saarbrücken, 1985), K. Mehlhorn (ed.), vol. 182 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 143–150.

[FS86] FIAT, A. AND SHAMIR, A. How to prove yourself: Practical solutions to identification and signature problems. In proc. *Advances in Cryptology* (1986), A. M. Odlyzko (ed.), vol. 263 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 186–194.

[FW78] FLETCHER, J. G. AND WATSON, R. W. Mechanisms for a reliable timer-based protocol. *Computer Networks* **2** (1978), 271–290.

[Gaf87] GAFNI, E. Generalized scheme for topology-update in dynamic networks. In proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), Springer-Verlag, pp. 187–196.

[GHS83] GALLAGER, R. G., HUMBLET, P. A., AND SPIRA, P. M. A distributed algorithm for minimum weight spanning trees. *ACM Trans. Program. Lang. Syst.* **5** (1983), 67–77.

[GJ79] GAREY, M. R. AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[Gos91] GOSCINSKI, A. *Distributed Operating Systems: The Logical Design.* Addison-Wesley, 1991 (913 pp.).

[GP92] GARAY, J. A. AND PERRY, K. J. A continuum of failure models for distributed computing. In proc. *6th Int. Workshop on Distributed Algorithms* (Haifa, 1992), S. Zaks and A. Segall (eds.), vol. 647 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 153–165.

[GP93] GOPAL, A. S. AND PERRY, K. J. Unifying self-stabilization and fault-tolerance. In proc. *12th Symp. on Principles of Distributed Computing*

(Ithaca, 1993), pp. 195–206.

[GT90] GASTEREN, A. J. M. VAN AND TEL, G. Comments on "On the proof of a distributed algorithm": Always-true is not invariant. *Inf. Proc. Lett.* **35** (1990), 277–279.

[Her91] HERMAN, T. Adaptivity through distributed convergence. Ph.D. thesis, Dept Computer Science, University of Texas at Austin, 1991.

[HH92] HSU, S.-C. AND HUANG, S.-T. A self-stabilizing algorithm for maximal matching. *Inf. Proc. Lett.* **43**, 2 (1992), 77–81.

[HJ90] HARGET, A. J. AND JOHNSON, I. D. Load balancing algorithms in loosely-coupled distributed systems: A survey. In *Distributed Computer Systems*, H. S. M. Zedan (ed.), Butterworths, 1990, pp. 85–108.

[Hoa74] HOARE, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM* **17** (1974), 549–557.

[Hoa78] HOARE, C. A. R. Communicating sequential processes. *Commun. ACM* **21** (1978), 666–677.

[Hoe99] HOEPMAN, J.-H. Self-stabilizing mutual exclusion on a ring, even if $K = N$. Submitted for publication, 1999.

[HP93] HIGHAM, L. AND PRZYTYCKA, T. A simple, efficient algorithm for maximum finding on rings. In proc. *7th Int. Workshop on Distributed Algorithms* (1993), A. Schiper (ed.), vol. 725 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 249–263.

[HS80] HIRSCHBERG, D. S. AND SINCLAIR, J. B. Decentralized extrema-finding in circular configurations of processes. *Commun. ACM* **23** (1980), 627–628.

[Hua88] HUANG, S.-T. A fully distributed termination detection scheme. *Inf. Proc. Lett.* **29**, 1 (1988), 13–18.

[IJ90] ISRAELI, A. AND JALFON, M. Self-stabilizing ring orientation. In proc. *4th Int. Workshop on Distributed Algorithms* (Bari, 1990), J. van Leeuwen and N. Santoro (eds.), vol. 486 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–14.

[IKWZ90] ITAI, A., KUTTEN, S., WOLFSTAHL, Y., AND ZAKS, S. Optimal distributed t-resilient election in complete networks. *IEEE Trans. Softw. Eng.* **SE–8**, 4 (1990), 415–420.

[IR81] ITAI, A. AND RODEH, M. Symmetry breaking in distributive networks. In proc. *Symp. on Theory of Computing* (1981), pp. 150–158.

[JJN+87] JOHANSEN, K. E., JØRGENSEN, U. L., NIELSEN, S. H., NIELSEN, S. E., AND SKYUM, S. A distributed spanning tree algorithm. In proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), vol. 312 of *Lecture Notes in Computer Science*, Springer-

Verlag, pp. 1–12.

[Kel76] KELLER, R. M. Formal verification of parallel programs. *Commun. ACM* **19**, 7 (1976), 371–384.

[KK89] KIROUSIS, L. M. AND KRANAKIS, E. A brief survey of concurrent readers and writers. *CWI Quarterly* **2**, 4 (1989), 307–330.

[KK90] KRANAKIS, E. AND KRIZANC, D. Computing boolean functions on anonymous hypercube networks. Report CS–R9040, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[KKM90] KORACH, E., KUTTEN, S., AND MORAN, S. A modular technique for the design of efficient leader finding algorithms. *ACM Trans. Program. Lang. Syst.* **12** (1990), 84–101.

[KMZ84] KORACH, E., MORAN, S., AND ZAKS, S. Tight upper and lower bounds for some distributed algorithms for a complete network of processors. In proc. *Symp. on Principles of Distributed Computing* (1984), pp. 199–207.

[KMZ85] KORACH, E., MORAN, S., AND ZAKS, S. The optimality of distributive constructions of minimum weight and degree restricted spanning trees in a complete network of processors. In proc. *4th Symp. on Principles of Distributed Computing* (1985).

[Kna87] KNAPP, E. Deadlock detection in distributed databases. *Computing Surveys* **19**, 4 (1987), 303–328.

[KP93] KATZ, S. AND PERRY, K. J. Self-stabilizing extensions for message-passing systems. *Distributed Comput.* **7**, 1 (1993), 17–26.

[LAD+92] LEISERSON, C. E., ABUHAMDEH, Z. S., DOUGLAS, D. C., FEYNMAN, C. R., GANMUKHI, M. N., HILL, J. V., HILLIS, W. D., KUSZMAUL, B. C., ST. PIERRE, M. A., WELLS, D. S., WONG, M. C., YANG, S.-W., AND ZAK, R. The network architecture of the Connection Machine CM-5. In proc. *4th Symp. on Parallel Algorithms and Architectures* (San Diego, 1992), L. Snyder (ed.), pp. 272–285.

[Lam78] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21** (1978), 558–564.

[Lam82] LAMPORT, L. An assertional correctness proof of a distributed algorithm. *Sci. Computer Programming* **2** (1982), 175–206.

[LeL77] LELANN, G. Distributed systems: Towards a formal approach. In proc. *Information Processing '77* (1977), B. Gilchrist (ed.), North-Holland, pp. 155–160.

[LL86] LISKOV, B. AND LADIN, R. Highly-available distributed services and fault-tolerant distributed garbage collection. In proc. *5th Symp. on Principles of Distributed Computing* (Vancouver, 1986), pp. 29–39.

[LMT87] LAKSHMANAN, K. B., MEENAKSHI, N., AND THULASIRAMAN, K.

A time-optimal message-efficient distributed algorithm for depth-first-search. *Inf. Proc. Lett.* **25** (1987), 103–109.

[LMW86] LOUI, M. C., MATSUSHITA, T. A., AND WEST, D. B. Election in a complete network with a sense of direction. *Inf. Proc. Lett.* **22** (1986), 185–187. Addendum: *Inf. Proc. Lett.* **28** (1988), 327.

[LSP82] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.* **4** (1982), 382–401.

[LSUZ87] LEEUWEN, J. VAN, SANTORO, N., URRUTIA, J., AND ZAKS, S. Guessing games and distributed computations in synchronous networks. In proc. *Int. Colloq. Automata, Languages, and Programming* (Karlsruhe, 1987), vol. 267 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 347–356.

[LT86] LEEUWEN, J. VAN AND TAN, R. B. Computer networks with compact routing tables. In *The Book of L*, G. Rozenberg and A. Salomaa (eds.), Springer-Verlag, 1986, pp. 259–273.

[LT87] LEEUWEN, J. VAN AND TAN, R. B. Interval routing. *Computer J.* **30** (1987), 298–307.

[LUST89] LENTFERT, P. J. A., UITTENBOGAARD, A. H., SWIERSTRA, S. D., AND TEL, G. Distributed hierarchical routing. Tech. rep. RUU–CS–89–5, Dept Computer Science, Utrecht University, The Netherlands, 1989.

[LY87] LAI, T. H. AND YANG, T. H. On distributed snapshots. *Inf. Proc. Lett.* **25** (1987), 153–158.

[Lyn68] LYNCH, W. C. Reliable full-duplex file transmission over half-duplex telephone lines. *Commun. ACM* **11**, 6 (1968), 407–410.

[Lyn96] LYNCH, N. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[MA89] MATIAS, Y. AND AFEK, Y. Simple and efficient election algorithms for anonymous networks. In proc. *3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), vol. 392 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 183–194.

[Mat87] MATTERN, F. Algorithms for distributed termination detection. *Distributed Computing* **2**, 3 (1987), 161–175.

[Mat89a] MATTERN, F. Global quiescence detection based on credit distribution and recovery. *Inf. Proc. Lett.* **30**, 4 (1989), 195–200.

[Mat89b] MATTERN, F. Message complexity of simple ring-based election algorithms: an empirical analysis. In proc. *Proc. 9th Int. Conf. on Distributed Computer Systems* (1989), pp. 94–100.

[Mat89c] MATTERN, F. Virtual time and global states of distributed systems. In proc. *Parallel and Distributed Algorithms* (1989), M. Cosnard et al. (eds.), Elsevier Science Publishers, pp. 215–226.

[McH90] McHugh, J. A. *Algorithmic Graph Theory.* Prentice-Hall, 1990.

[MM79] Menasce, D. and Muntz, R. Locking and deadlock detection in distributed databases. *Commun. ACM* **21** (1979).

[MM84] Mitchell, D. P. and Merrit, M. J. A distributed algorithm for deadlock detection and resolution. In proc. *3rd Symp. on Principles of Distributed Computing* (1984), pp. 282–284.

[MNHT89] Masuzawa, T., Nishikawa, N., Hagihara, K., and Tokura, N. Optimal fault tolerant distributed algorithms for election in complete networks with a global sense of direction. In proc. *3rd Int. Workshop on Distributed Algorithms* (1989), J.-C. Bermond and M. Raynal (eds.), vol. 329 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 171–182.

[MP88] Manna, Z. and Pnueli, A. The anchored version of the temporal framework. In proc. *Symp. on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency* (Noordwijkerhout, 1988), J. W. de Bakker, W.-P. de Roever, and G. Rozenberg (eds.), vol. 354 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 201–284.

[MS79] Merlin, P. M. and Segall, A. A failsafe distributed routing protocol. *IEEE Trans. Commun.* **COM–27**, 9 (1979), 1280–1287.

[MS80a] Merlin, P. M. and Schweitzer, P. J. Deadlock avoidance in store-and-forward networks I: Store-and-forward deadlock. *IEEE Trans. Commun.* **COM–28**, 3 (1980), 345–354.

[MS80b] Merlin, P. P. and Schweitzer, P. J. Deadlock avoidance in store-and-forward networks II: Other deadlock types. *IEEE Trans. Commun.* **COM–28**, 3 (1980), 355–360.

[MS85] Mahany, S. R. and Schneider, F. B. Inexact agreement: Accuracy, precision, and graceful degradation. In proc. *4th Symp. on Principles of Distributed Computing* (1985), pp. 237–249.

[MW87] Moran, S. and Wolfstahl, Y. Extended impossibility results for asynchronous complete networks. *Inf. Proc. Lett.* **26** (1987), 145–151.

[Nai88] Naimi, M. Global stability detection in the asynchronous distributed computations. In proc. *Future trends of Distributed Computing Systems in the Nineties* (Hong Kong, 1988), pp. 87–92.

[Nat86] Natarajan, N. A distributed scheme for detecting communication deadlocks. *IEEE Trans. Softw. Eng.* **SE–12**, 4 (1986), 531–537.

[Odl84] Odlyzko, A. M. Discrete logarithms in finite fields and their cryptographic significancce. In proc. *Advances in Cryptology* (Paris, 1984), vol. 209 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 224–314.

[OG76] Owicki, S. and Gries, D. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM* **19**, 5 (1976), 279–285.

[Pel90] PELEG, D. Time-optimal leader election in general networks. *J. Parallel and Distributed Computing* **8**, 1 (1990), 96–99.

[Pet82] PETERSON, G. L. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.* **4** (1982), 758–762.

[Pet85] PETERSON, G. L. Efficient algorithms for elections in meshes and complete networks. Tech. rep. TR 140, Dept Computer Science, University of Rochester, Rochester NY 14627, 1985.

[PKR84] PACHL, J., KORACH, E., AND ROTEM, D. Lower bounds for distributed maximum finding algorithms. *J. ACM* **31** (1984), 905–918.

[PSL80] PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM* **27** (1980), 228–234.

[Ran83] RANA, S. P. A distributed solution of the distributed termination problem. *Inf. Proc. Lett.* **17** (1983), 43–46.

[RBS92] RICCARDI, A., BIRMAN, K., AND STEPHENSON, P. The cost of order in asynchronous systems. In proc. *6th Int. Workshop on Distributed Algorithms* (Haifa, 1992), A. Segall and S. Zaks (eds.), vol. 647 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 329–345.

[RGG96] RUDOLPH, E., GRAUBMANN, P., AND GRABOWSKI, J. Tutorial on message charts. *Comput. Networks ISDN Syst.* **28** (1996), 1629–1641.

[RSA78] RIVEST, R., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **21** (1978), 120–126.

[Ruž88] RUŽIČKA, P. On efficiency of interval routing algorithms. In proc. *Mathematical Foundations of Computer Science* (1988), M. P. Chytil, L. Janiga, and V. Koubek (eds.), vol. 324 of *Lecture Notes in Computer Science*, pp. 492–500.

[Ruž98] RUŽIČKA, P. Efficient communication schemes. In proc. *SOFSEM98: Theory and Practice of Informatics* (1998), B. Rovan (ed.), vol. 1521 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 244–263.

[San84] SANTORO, N. Sense of direction, topological awareness, and communication complexity. *ACM SIGACT News* **16** (1984), 50–56.

[Sch91] SCHOONE, A. A. Assertional verification in distributed computing. Ph.D. thesis, Dept Computer Science, Utrecht University, The Netherlands, 1991 (191 pp.).

[Seg83] SEGALL, A. Distributed network protocols. *IEEE Trans. Inf. Theory* **IT–29** (1983), 23–35.

[SES89] SCHIPER, A., EGGLI, J., AND SANDOZ, A. A new algorithm to implement causal ordering. In proc. *3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), vol. 392 of

Lecture Notes in Computer Science, Springer-Verlag, pp. 219–232.

[SF86] SHAVIT, N. AND FRANCEZ, N. A new approach to the detection of locally indicative stability. In proc. *Int. Colloq. Automata, Languages, and Programming* (1986), L. Kott (ed.), vol. 226 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 344–358.

[SG84] SPECTOR, A. AND GIFFORD, D. The Space Shuttle computer system. *Commun. ACM* **27** (1984), 874–900.

[SK85] SANTORO, N. AND KHATIB, R. Labelling and implicit routing in networks. *Computer J.* **28** (1985), 5–8.

[SK87] SLOMAN, M. AND KRAMER, J. *Distributed Systems and Computer Networks.* Prentice-Hall International, 1987 (336 pp.).

[SR85] SANTORO, N. AND ROTEM, D. On the complexity of distributed elections in synchronous graphs. In proc. *Int. Workshop on Graph-Theoretic Concepts in Computer Science* (1985), H. Noltemeier (ed.), Trauner Verlag, pp. 337–346.

[SS88] SAAD, Y. AND SCHULTZ, M. H. Topological properties of hypercubes. *IEEE Trans. Comput.* **C–37**, 7 (1988), 867–872.

[SS89] SCHIEBER, B. AND SNIR, M. Calling names on nameless networks. In proc. *8th Symp. on Principles of Distributed Computing* (Edmonton, 1989), pp. 319–329.

[ST89] SPIRAKIS, P. AND TAMPAKAS, B. Efficient distributed algorithms by using the Archimedean time assumption. *Informatique Théorique et Applications* **23**, 1 (1989), 113–128.

[Ste99] ŠTEFANKOVIČ, D. Acyclic orientations do not lead to optimal deadlock-free packet routing algorithms. *Inf. Proc. Lett.* (2000).

[T1895] TARRY, G. Le problème des labyrinthes. *Nouvelles Annales de Mathématique* **14** (1895).

[Taj77] TAJIBNAPIS, W. D. A correctness proof of a topology information maintenance protocol for a distributed computer network. *Commun. ACM* **20**, 7 (1977), 477–485.

[Tan96] TANENBAUM, A. S. *Computer Networks*, 3rd edition. Prentice Hall, 1996.

[Tar72] TARJAN, R. E. Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1** (1972), 146–160.

[Tay89] TAYLOR, K. The role of inhibition in asynchronous consistent-cut protocols. In proc. *3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), vol. 392 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 280–291.

[Tel91a] TEL, G. Fouttolerantie in gedistribueerde algoritmen. Lecture Note INF/DOC–91–02, Dept Computer Science, Utrecht University, The

Netherlands, 1991. In Dutch.

[Tel91b] TEL, G. *Topics in Distributed Algorithms*, vol. 1 of *Cambridge Int. Series on Parallel Computation.* Cambridge University Press, 1991 (240 pp.).

[Tel94] TEL, G. Network orientation. *Int. J. Foundations Comput. Sci.* **5**, 1 (1994), 23–57.

[Tel95] TEL, G. Sense of direction in processor networks. In proc. *SOFSEM'95: Theory and Practice of Informatics* (Milovy (Czech Rep.), 1995), M. Bartošek, J. Staudek, and J. Wiedermann (eds.), vol. 1012 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 50–82.

[TKZ94] TEL, G., KORACH, E., AND ZAKS, S. Synchronizing ABD networks. *IEEE Trans. Networking* **2**, 1 (1994), 66–69.

[TL86] TAN, R. B. AND LEEUWEN, J. VAN. General symmetric distributed termination detection. Tech. rep. RUU–CS–86–2, Dept Computer Science, Utrecht University, The Netherlands, 1986.

[TM89] TAUBENFELD, G. AND MORAN, S. Possibility and impossibility results in a shared memory environment. In proc. *3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), vol. 392 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 254–267.

[TM93] TEL, G. AND MATTERN, F. The derivation of termination detection algorithms from garbage collection schemes. *ACM Trans. Program. Lang. Syst.* **15**, 1 (1993), 1–35.

[Top84] TOPOR, R. W. Termination detection for distributed computation. *Inf. Proc. Lett.* **18** (1984), 33–36.

[Tou80a] TOUEG, S. An all-pairs shortest-path distributed algorithm. Tech. rep. RC 8327, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, 1980.

[Tou80b] TOUEG, S. Deadlock- and livelock-free packet switching networks. In proc. *Symp. on Theory of Computing* (1980), pp. 94–99.

[TU81] TOUEG, S. AND ULLMAN, J. D. Deadlock-free packet switching networks. *SIAM J. Comput.* **10**, 3 (1981), 594–611.

[Vit85] VITÁNYI, P. M. B. Time-driven algorithms for distributed control. Tech. rep. CS–R8510, Centre for Mathematics and Computer Science, Amsterdam, 1985.

[VT95] VERWEIJ, A. M. AND TEL, G. A Monte Carlo algorithm for election. In proc. *2nd Colloq. on Structural Information and Communication Complexity* (Olympia (Greece), 1995), L. M. Kirousis and E. Kranakis (eds.), Carleton University Press, pp. 77–88.

[Wat81] WATSON, R. W. Timer-based mechanisms in reliable transport protocol connection management. *Computer Networks* **5** (1981), 47–56.

[WT94] WEZEL, M. C. VAN AND TEL, G. An assertional proof of Rana's algorithm. *Inf. Proc. Lett.* **49**, 5 (1994), 227–233.

主题词索引

索引中的页码为英文原书页码，与书中边栏页码一致。

|| (concurrent)（并发），57
~ (equivalent executions)（等价执行），59
Hn (harmonic number)（调和数），237
M (A)（multisets)（多集），46
≺（因果序），56, 182
◁ (prefix)（前缀），147
Æ (transition relation)（转移关系），45
0/1-decided configuration（0/1-判定配置），438
0/1-valent configuration（0/1-价配置），438

ABD network,（ABD网络），400
accuracy （failure det.)（精确度，故障检测），507
acknowledgement（确认），77, 86
action（行为），554
acyclic orientation cover（无环有向覆盖），163
addressing（定址），69, 552
adjacent nodes（相邻节点），557
agreement（一致）
  for broadcast algorithm（广播算法），471
  for consensus algorithm（一致性算法），438
  for interactive consistency（交互一致性），492
algorithm（算法）
  Attiya et al. (renaming)（重命名），447
  Awerbuch, DFS（深度优先），212
  Awerbuch-Gallager, 419
  Bracha-Toueg (Byzantine broadcast)，(Byzantine广播），463
Bracha-Toueg (consensus)（一致性），453, 458
Chandy-Lamport, 341
Chandy-Misra, 121, 153
Chang, echo（回波），194
Chang-Roberts, 234, 323
Cidon, depth-first search（深度优先搜索），214
credit-recovery（信用恢复），299
deterministic（确定性的），309
Dijkstra-Feijen-VanGasteren, 285
approximate agreement（近似一致），432, 445
Archimedean assumption（Archimedean假设），420
architecture（体系结构），19
ARPANET network（ARPANET网络），6
assertion（断言），50
Dijkstra-Scholten, 276
distributed (definition)（分布式定义），46
Dolev-Klawe-Rodeh, 238
fast-convergence（快速收敛），497
Finn, 199, 232
Fischer et al. (knot agreement)（结一致），443
Floyd-Warshall, 110
Gallager-Humblet-Spira, 249
global-marking （全局-标记），351
Itai-Rodeh (election)（选举），323
Itai-Rodeh (ring size)（环规模），329
Korach-Kutten-Moran, 260
Lai-Yang, 342
LeLann, 233
local (definition)（局部定义），46
Merlin-Segall, 120
Netchange（网络变更），123, 153, 523, 545
Peleg (election)（选举），232
Peterson, 238, 266
Rana, 302
Safra, 289
Santoro-Khatib, 133, 414
Shavit-Francez, 280
Tarry, traversal （遍历），207
Toueg, 113, 399
Update（更新），543
vector-counting （向量计数），293
alive (of process)（活的进程），350
alpha synchronizer（α同步器），411
alpha-time complexity（α时间复杂度）,223
alternating-bit protocol（交替位协议），85
ancestor（祖先），560
AND model (deadlock)（AND模型）(死锁）,354
anonymous network（匿名网络），69, 307, 309, 526
applicable action（应用行为），554

application layer（应用层），22
asynchronous bounded-delay（ABD）network,（异步延迟受限网络），400, 493
authentication（of a message）（消息鉴别），482
Awerbuch' s *DFS algorithm*（Awerbuch的深度优先搜索算法），212
Awerbuch-Gallager BFS algorithm（Awerbuch-Gallager的广度优先搜索算法），419

backward-count controller（后向计数控制器），168
backward-state controller（后向状态控制器），170
balanced sliding-window protocol（平衡滑动窗口协议），79
basic algorithm（基本算法），271, 349
basic computation（基本计算），271, 349
basic message（基本消息），271, 341
benign failure（良性故障），430
beta synchronizer（b同步器），411
BFS tree（广度优先搜索树），414
bifurcated routing（分支路由），109
bipartite（双向），570
bivalent configuration（双价配置），438
body（of an action）（行为体），554
boolean routing（布尔路由），146
bottom of a partial order（偏序底），224
bounded drift（有限漂移），494
breadth-first search tree（广度优先搜索），414
broadcast message（广播消息），26
broadcasting（广播），10
browser（浏览器），7
buffer graph（缓冲图），159
buffer-graph controller（缓冲图控制器），159
bus（总线），9
Byzantine broadcast（Byzantine广播），462, 471
Byzantine process, 429
Byzantine-robust consensus （Byzantine健壮一致性），456

capacity（of a channel）（信道容量），68
causal order（$\prec$, $\preceq$）（因果序），56
  local（$\preceq_p$）（局部），336
causality chain（因果链）57, 185
centralized algorithm（集中式算法），183, 272
chain-time complexity（链时间复杂度），223
Chandy-Lamport algorithm（Chandy-Lamport算法），341
Chandy-Misra algorithm（Chandy-Misra算法），121,153
Chang's echo algorithm （Chang回波算法），194
Chang-Roberts algorithm（Chang-Roberts算法），234,323
channel（信道），64
child（子节点），560
chordal ring（弦环），360
Cidon's DFS algorithm （Cidon的DFS算法），214
clique（团），65, 562
  election in（选进），265, 422
  sense of direction in（方向侦听），567
  wave algorithm for（波动算法），196
clock （时钟），61, 302, 493
clock synchronization（时钟同步），494
deterministic（确定性的），494
probabilistic（概率性的），496
clustering（簇），150
coding in time（按时间编码），399
collateral composition（并组合），536
collect ....from （从....收集），509
coloring（of a planar graph）（平面图的着色），538
commit-abort（交付异常中断），432
communication deadlock （通信死锁），354
communication failure（通信故障），66
compact routing（压缩路由），132
  interval routing（区间路由），137
  prefix routing（前缀路由），146
  tree-labeling scheme（树标号模式），133
comparison algorithm（比较算法），229, 407
complete graph（完全图），562
completeness（failure det.）（完全性）（故障检测），507
computation（计算），60
  synchronous（同步），399
computation forest（计算森林），280
computation graph（计算图），276
computation tree（计算树），276
concurrent events（||）（并发事件（||）），57
configuration（配置），45
  0/1-decided（0/1-判定的），438
  0/1-valent（0/1-价的），438
  bivalent（双价的），438
  deadlock（死锁），158, 351
  decided（判定的），438
  initial（初始的），45

legitimate（合法的），521
livelock（活锁），174
of a probabilistic algorithm（概率算法），310
possible（可能的），338
cycle-increase（周期递增），541
symmetric（对称的），317
synchronous system（同步系统），398
terminal（终止），45
univalent（单价的），438
congestion（拥塞），8
connected component（连通分量），557
connected graph（连通图），557
connection management（连接管理），74, 90
consensus（一致性），432
consensus algorithm（一致性算法），438
agreement（一致），438
convergence（收敛性），452
dependence（相关性），462
non-triviality（非平凡性），438
termination（终止），438
consensus problem（一致性问题），431
consistency for decision task（判定任务的一致性），431, 445
consistent cut（一致割），339
consumption of a packet（包消耗），157
control algorithm（控制算法），271
control message（控制消息），271
control-oriented notation（面向控制的表示法），553
controller（控制器），157
acyclic orientation cover（无环有向覆盖），164
backward-count（后向计数），168
backward-state（后向状态），170
buffer-graph（缓冲图），159
deadlock-free（无死锁），158
destination scheme（目的地模式），161
forward-count（前向计数），167
forward-state（前向状态），168
hops-so-far scheme（当前跳数模式），162
hops-to-go scheme（下一跳数模式），163, 177
livelock-free（无活锁的），174
convergence（consensus alg.）（收敛,一致性算法），452
copy-release deadlock（无复制的死锁），173
crash model（损毁模型），429
crash-robust consensus（损毁-健壮一致性），452
creation of a message（消息创建），67
credit-recovery algorithm（信用恢复算法），299
critical section（临界区），524
CSMA/CD（载波侦听多路访问/冲突检测），24
cut（割），338
consistent（一致性），339
cycle（回路），557
cycle-free routing tables（无回路路由表），108
cyclic function（周期函数），318
cyclic interval（周期间隔），134
cyclic wait（周期等待），11

data-link layer（数据链路层），20, 75
deadlock（死锁），158, 268, 351
AND model（AND模型），354
communication deadlock（通信死锁），354
copy-release（无复制的），173
OR model（OR模型），354
pacing（步），173
progeny（后代），173
reassembly（重组），174
resource（资源），354
store-and-forward（存储转发），155
deadlock detection（死锁检测），11, 346, 349
deadlock freeness（死锁自由度），52
debugging（调试），336
decentralized algorithm（分散式算法），183, 272
decided configuration（判定的配置），438
decision event（判定事件），183
decision graph（判定图），449
decision problem（判定问题），431
decision task（判定任务），445
degree（度），557
dependence（相关性）
for broadcast algorithm（广播算法），471
for consensus algorithm（一致性算法），462
for interactive consistency（交互一致性），492
depth（深度），560
depth-first search（深度优先搜索），209
Awerbuch's algorithm（Awerbuch算法），212
Cidon's algorithm（Cidon算法），214
DFS tree（深度优先搜索树），209, 561
interval labeling scheme（ILS）（区间标号模式），138

Update algorithm（更新算法），546
with sense of direction（带方向侦听），362
derivate（导出），52，72
descendant（后代），560
destination（目的地），47
deterministic algorithm（确定性算法），309
diffusing computation（扩散计算），272，276
digital signature（数字签名），482
Dijkstra-Feijen-Van Gasteren algorithm
（Dijkstra-Feijen-Van Gasteren算法），285
decision（判定），183
Dijkstra-Scholten algorithm
（Dijkstra-Scholten算法），276
direct addressing（直接定址），69，553
directed graph（有向图），558
directed network（有向网络）
wave algorithm for（波动算法），197
direction，sense of（方向，侦听），565
discrete logarithm（离散对数），486
ElGamal's scheme（ElGamal模式），486
distance（距离），557
distributed algorithm（definition）（分布式算法定义），46
distributed database（分布式数据库），7，354
commit-abort（交付异常中断），432
deadlock（死锁），354
distributed system（分布式系统），2
Dolev-Klawe-Rodeh algorithm（Dolev-Klawe-Rodeh算法），238
drift（漂移），100，404，494
duplication of a message（消息复制），67，225
dynamic topology（动态拓扑结构），65，522

early stopping（提前停止），492
echo algorithm（回波算法），225，248
edge（边），556
edge function（边缘函数），541
election（选举），10，227，432
[k，l]-election（[k，l]-选举），467
Chang-Roberts alg.（Chang-Roberts算法），234
fault-tolerant（容错），444
Gallager-Humblet-Spira alg.
Gallager-Humblet-Spira算法），249
in hypercube（超立方体），267
in planar graph（平面图），267
in torus（圆环），267
Korach-Kutten-Moran alg.，（Korach-Kutten-Moran算法），260
LeLann' s alg.，（LeLann算法），233
match-making（hypercube）（进行比赛（超立方体）），375
Peterson's alg.，（Peterson算法），238
using traversal（利用遍历），260
ElGamal's signature scheme（ElGamal 签名模式），486
empty path（空路径），541
emulation（failure det.）（竞争、故障检测），517
enabled action（能行行为），554
environment（环境），276
equivalent of executions（~）（执行等价），59
Ethernet（以太网），9
Euler's phi function（Euler's F 函数），487
event（事件），45，47
corresponding（对应），54
internal（内部），45
postshot（后照），337
receive（接收），46
send（发送），46
event-driven notation（事件驱动表示法），554
eventual strong accuracy（最终强精确性），507
eventual week accuracy（最终弱精确性），508
eventually perfect failure det.（最终完美故障检测），508
eventually strong failure det.（最终强故障检测），508
execution（执行），45
exhaustive set（穷举集），242
explicit termination（显式终止），268
extinction（停止），246，267
extrema-finding problem（极值搜索问题），229

failure detector（故障检测器），505
history（历史纪录），507
failure pattern（故障模式），506
fair collateral composition（公平并组合），537
fair execution（t-crash）（公平执行，t-损毁），438
fair execution（t-initially-dead）（公平执行，t-初始-死亡），442
fair scheduling（公平调度），452
fairness assumption（公平性假设），49，80
fast-convergence algorithm（快速收敛算法），497

father（父节点），560
fault model（故障模型），429
  Byzantine, 429
  crash（损毁），429
  hierarchy（分层），430
  initially dead（初始死亡的），429
  omission（省略），430
  timing（定时），431
fault tolerance（容错），437
  stabilizing algorithm（稳定算法），522
feasible snapshot（可行快照），338
Fiat-Shamir signature scheme（Fiat-Shamir签名模式），488
fifo channel（先进先出信道），67
file-transfer protocol（FTP）（文件传输协议），20
Finn's algorithm（Finn算法），225, 232
firing-squad problem（触发组问题），491
Floyd-Warshall algorithm（Floyd-Warshall算法），110
flush channel（清仓信道），67
follower（追随者），183
forall statement（forall语句），551
forest（森林），561
fork（分支点），439
forward-count controller（前向计数控制器），167
forward-state controller（前向状态控制器），168
forwarding of a packet（包转发），156
fragment of an MST（MST碎片），250
frame（帧），21
fromthis（*in receive statement*）（从这个进程，在接收语句中），552
frond edge（非树边），137, 560
full channel（全信道），68

Gallager-Humblet-Spira MST algorithm（Gallager-Humblet-Spira MST算法），249
gamma synchronizer（g同步器），411
garbage collection（废料收集），16, 346
garbling of a message（消息碎片），67, 76
gateway（网关），6
general（通用），462, 471
generation of a packet（包生成），156
generator（of a group）（组生成器），486
global state（全局状态），45
  lack of（缺少），27
global time（全局时间），396
lack of（缺少），27
global virtual time（全局虚拟时间），349
global-marking algorithm（全局标记算法），351
graceful degradation（体面降级），428
graph（图），64, 557
greedy routing（贪婪路由），366
grid（网格），563
  routing（路由），144
group（algebra）（群，代数），357
group membership（组成员资格），435
group sense of direction（组方向侦听），358
guaranteed path（保障路径），159
guard（of an action）（行为阈值），554

Hamiltonian cycle（哈密尔顿回路），559
harmonic number（调和数），237, 266
heartbeat failure detector（心跳故障检测器），517
hierarchical routing（分级路由），149
hops-so-far scheme（当前跳数模式），162
hops-to-go scheme（下一跳数模式），163, 177
hypercube（超立方体），65, 563
  anonymous（匿名），323
  controller for（控制器），177
  election in（选举），267, 422
  interval routing in（区间路由），145
  match-making（进行比赛的），375
  sense of direction in（方向侦听），206, 568
  traversal algorithm for,（遍历算法），205

identity（标识），69
IEEE standards（IEEE标准），20, 23
ILS, interval labeling scheme（区间标号模式ILS），137
image（图像），159
IMP（interface message processor）（接口信息处理机IMP），6
implicit message（隐式消息），399, 405
implicit termination（隐式终止），268
in-neighbor（入近邻），558
incident edge（依附边），557
incoming edge（入边），558
index（索引），486
indirect addressing（间接定址），69, 553
induced subgraph（导出子图），557

inexact agreement（非精确一致性），497
infimum（下确界），188
infimum computation（下确界计算），189
infimum theorem（下确界定理），190
initial configuration（初始配置），45, 46
initially dead process（启动死进程），429
initiator（初始者），183
input collection（输入集合），318
interactive consistency（交互一致性），492
interface message processor（IMP）（接口消息处理机（IMP）），6
internal events（内部事件），45
Internet（因特网），6
interval labeling scheme（区间标号模式），137, 225
  depth-first search（深度优先搜索），138
  for a grid（网格），144
  for a hypercube（超立方体），145
  for a ring（环），143
  for an outerplanar network（外平面网络），145
  linear（线性），142
  valid（有效），137
interval routing（区间路由），137
  efficiency of（效率），140
  linear（线性），142
  multi-dimensional（多维），146
  neighborly（近邻的），141
  optimal（最优的），143
invariant（不变式），51, 72
ISO standard（ISO标准），20
Itai-Rodeh algorithm（election）（Itai-Rodeh算法（选择）），323
Itai-Rodeh algorithm（ring size）（Itai-Rodeh算法（环长度）），329

key（for signature）（钥匙，签名），485
knot（结），444
Korach-Kutten-Moran algorithm（Korach-Kutten-Moran算法），260
Kruskal's MST algorithm（Kruskal MST算法），250

Lai-Yang algorithm（Lai-Yang算法），342
Lamport's logical clock（Lamport逻辑时钟），62, 304
LAN（local-area network），（局域网）3
Las Vegas algorithm（Las Vegas算法），313, 452
layer（层），18
leader（领导人），228
legitimate configuration（合法配置），521
LeLann's algorithm（LeLann算法），233
length（长度），557
length-increase（长度增加），542
lexicographic order（字典序），72
lieutenant（中尉），471
line of processors（处理器线），309
linear ILS（线性ILS），141
linear-interval routing（线性区间路由），142
link register（链接寄存器），524
livelock（活锁），174
liveness（活动性），50, 52
load balancing（负载平衡），14
local algorithm（局部算法），46
local causal order（局部因果序），336
local snapshot（局部快照），336
local-area network（局域网），4, 8
local-area network（LAN）（局域网（LAN）），3
locking（in database）（数据库中加锁），354
lockstep operation（锁步操作），396
logical link control sublayer（逻辑链路控制子层），24
long-haul network（远程网络），3
loss of a message（消息损失），67
low-diameter spanning tree（小直径生成树），560

malign failure（恶性故障），430
mask（掩码），382
match-making algorithm（进行比赛的算法），375
matching（in graph）（匹配），530
Mattern's vector clock（Mattern向量时钟），63
Mattern, credit-recovery algorithm, 299
maximal matching（最大匹配），530
maximum packet lifetime（最大包生存期），87, 172
meaningful snapshot（有意义的快照），339
medium-access-control sublayer（介质访问控制子层），23
Merlin-Segall algorithm（Merlin-Segall算法），120
message（消息），46, 551
  broadcast（广播），26
  implicit（隐式），399
  multicast（组播），26
  point-to-point（点到点），26

postshot（后照），338
preshot（前照），338
message chain（消息链），223
message passing（消息传递），44，523
message termination（消息终止），268
message-generation function（消息生成函数），398
MIMD（多指令多数据），12
minimal spanning tree（MST）（最小生成树（MST）），249
minimal-path forest（最短路径森林），542
minimal-path problem（最短路径问题），541
minimal-weight spanning tree（最小加权生成树），249，560
minimal-delay routing（最小延迟路由），104
minimal-hop routing（最小跳数路由），104
mixed-failure model（混合故障模型），430
monitor（管程），17
monotonicity（单调性），541
Monte Carlo algorithm（Monte Carlo算法），313
MST（minimal spanning tree）（MST（最小生成树）），249
multi-dimensional interval routing（多维区间路由），146
multi-source algorithm（多源算法），183
multicast message（组播消息），26
multiple interval algorithm（多路区间算法），142
multiple-instruction multiple-data（MIMD），（多指令多数据（MIMD）），12
multiple-path routing（多路径路由），109
multiprocessor computer（多处理器计算机），11
multisets（M（A））（多集（M(A)）），46
mutual exclusion（互斥），11，524

named network（命名网络），69
neigh（相邻），551
neighbor（近邻），64，551，557
neighbor knowledge（近邻知识），69
depth-first search using（用深度优先搜索），216
neighborly ILS（近邻ILS），141
neighborly interval routing（近邻区间路由），141
Netchange algorithm（变更算法），123，153，523，545
network（网络），64
anonymous（匿名），69，307
asynchronous bounded-delay（ABD）（异步有限延迟），400
dynamic（动态），522
named（命名的），69
synchronous（同步的），398
node（节点），2，556
network layer（网络层），21，156
non-determinism（非确定性），28，309
non-triviality（非平凡），431
for consensus algorithm（一致算法），438
for decision task（确定任务），450
norm function（范函数），52，130，139

omission failure（省略故障），430
one-time complexity（单位时间复杂度），221
open-systems interconnection（OSI）（开放系统互连），20
optimal ILS（最优ILS），141
optimal sink tree（最优汇集树），108，136，560
optimistic fault tolerance（乐观容错），520
OR model（deadlock）（OR模型，死锁），354
order-preserving renaming（保持顺序的重命名），446，449
orientation（定向），163
OSI（开放系统互连），20
out-neighbor（出近邻），558
outerplanar graph（外平面图），557
interval routing in（区间路由），145
outgoing edge（出边），558

pacing deadlock（步死锁），173
packet（包），77
parallel composition（并行组合），72
parallel random-access memory（PRAM）（并行随机访问存储器），14
parent（父节点），560
partial failure（部分故障），3，428
path（路径），557
Peleg's algorithm（election）（Peleg选举算法），232
perfect failure detector（完美故障检测器），508
permanent failure（永久故障），436
pessimistic fault tolerance（悲观容错），520
Peterson's election algorithm（Peterson选举算法），238，266
physical layer（物理层），20，23
piggybacking（捎带），301，343

pipe（管道），17
planar graph（平面图），557
  coloring of（着色），538
  election in（选举中），267
point-to-point message（点到点消息），26
point-to-point network（点到点网络），6
polling algorithm（轮询算法），196
possible configuration（可能配置），338
postshot event（后照事件），337
postshot message（后照消息），338
PRAM, 14
precision（精度），494
prefix（$\lhd$）（前缀），147
prefix labeling scheme（前缀标号模式），147, 224
  tree（树），147
  valid（有效的），147
prefix routing（前缀路由），146
presentation layer（表示层），22
preshot event（前照事件），337
preshot message（前照消息），338
Prim's MST algorithm（Prim MST算法），250
primitive root（本原根），486
private key（私钥），485
probabilistic algorithm（概率算法），308
probabilistic consensus algorithm（概率一致性算法），453, 458
probabilistic process（随机过程），309
process（进程），46
  deterministic（确定性的），309
  probabilistic（概率性的），309
  synchronous（同步），398
process identity（进程标识），69
process termination（进程终止），268
producer-consumer problem（生产者消费者问题），16
progeny deadlock（子代死锁），173
propagation of information with feedback（具有反馈的信息传播），187
protocol（协议），19
pseudo-anonymous network（伪-匿名网络），333
pseudo-stabilization（伪-稳定性），523
pseudocode（伪代码），551
public key（公钥），486
pulse（脉冲），397, 470
pulse complexity（脉冲复杂度），492
pulse time（脉冲时间），404

Rana's algorithm（Rana算法），302
randomization（随机化），309
reachable（可达的），45
  in Byzantine model, 456
read-all（读所有），524
read-one（读一个），524
real time（实时），493
real-time clock（实时时钟），62
reassembly deadlock（重组死锁），174
receive events（接收事件），46
receive statement（接收语句），552
receive-fromthis statement（fromthis接收语句），552
regular graph（正则图），557
reliable channel（可靠信道），66
remote procedure call（远程过程调用），26
renaming problem（重命名问题），446
  order-preserving（保序的），446, 449
replay argument（重放证明），320, 323
resilience（弹回），438
resource allocation（资源分配），11
resource deadlock（资源死锁），354
restricted-degree spanning tree（度受限的生成树），561
ring（环），65, 559
  controller for（控制器），164
  election in（选举），232, 265
  orientation（定向），527
  routing for（路由），143
  sense of direction in（方向侦听），566
  termination detection in（终止检测），285
  wave algorithm for（波动算法），190
Rivest-Shamir-Adleman scheme（Rivest-Shamir-Adleman模式），488
robust algorithm（健壮算法），429
rooted tree（有根树），560
rotating coordinator（旋转协调器），510
round（循环），470
round complexity（循环复杂度），492
round-robin（循环），23
routing（路由），8, 104
  compact（压缩），132
  hierarchical（分级的），149
  minimum-delay（最小延迟），104

minimum-hop（最小跳数），104
shortest-path（最短路径），104
Update algorithm（更新算法），545
routing tables（路由表），103
RSA signature scheme（RSA签名模式），488
rubber band transformation（橡皮圈转换），59

safety（安全），50
Safra's algorithm（Safra算法），289
Santoro-Khatib tree-labeling（Santoro-Khatib树标号），133, 414
scalable organization（可扩展的组成），9
secret key（密钥），485
selective delay（选择性延迟），400, 406
self-stabilization（自稳定），521
self-stabilization algorithm（自稳定算法），435
semaphore（信号量），17
send event（发送事件），46
send statement（发送语句），552
sending window（发送窗口），82
sense of direction（方向侦听），69, 565
group（分组），358
in clique（团），567
in hypercube（超立方体），568
in ring（环），566
in torus（圆环），568
ring（环），527
sequence number（序号），77
session layer（会话层），22
shared variables（共享变量），44, 524
Shavit-Francez algorithm（Shavit-Francez算法），280
Sherwood algorithm（Sherwood算法），312
shortest-path routing（最短路径路由），104
with Netchange algorithm（变更算法），132
signature scheme（签名模式），482
ElGamal，486
Fiat-Shamir, 488
Rivest-Shamir-Adleman, 488
signing（of a message）（签名（消息）），482
SIMD（单指令多数据），12
simple cycle（简单回路），557
simple mail-transfer protocol（SMTP）（简单邮件传输协议），20
simple path（简单路径），557
simultaneity（同时）
for broadcast algorithm（广播算法），471
for firing-squad algorithm（触发组算法），491
single-instruction, multiple-data（SIMD）（单指令多数据），12
single-source algorithm（单数据源算法），183
sink tree（汇集树），108
optimal（最优），108, 136, 560
six-color theorem（6-色定理），538
size of routing table（路由表大小），150
SMTP, 20
snapshot（快照），335
feasible（可行的），338
meaningful（有意义的），339
snapshot state（快照状态），336
son（子），560
source of a packet（包源），156
Space Shuttle（航天飞机），428
spanning subgraph（生成子图），557
spanning tree（生成树），560
low diameter（小直径），560
specification（规范），521
square root（平方根），488
stabilization（稳定性），521
stabilizing algorithm（稳定算法），435
stable property（稳定性质），335, 346, 351
star（星），65, 561
starter（启动器），183
state model（状态模型），524
state of a process（进程的状态），46
static topology（静态拓扑结构），65
step（步），440
store-and-forward deadlock（存储转发死锁），155
store-and-forward network（存储转发网络），8
strong accuracy（强精确性），507
strong completeness（强完全性），507
strong failure detector（强故障检测器），508
strong fairness（强公平性），49
strongly connected graph（强连通图），558
subgraph（子图），557
subtree（子树），560
suitable（buffer）（适合的（缓冲区）），159
superimposed algorithm（叠加算法），271
symmetric configuration（对称配置），317

symmetric system（对称系统），309
synchronization（同步），10, 188, 408
  ABD network（ABD网络），401
  for fault tolerance（容错），434
synchronizer（同步器），397, 401, 434, 470
synchronous computation（同步计算），399
synchronous message passing（同步消息传递），47
synchronous process（同步进程），398
synchronous system（同步系统），397
decision graph of（判定图），449
Tarry's traversal algorithm（Tarry遍历（算法）），207
task（任务），433, 445
  connected（连通的），450

TCP/IP protocol（TCP/IP协议），20
terminal configuration（终端配置），45, 268
termination（终止），431
  for broadcast algorithm（广播算法），471
  for consensus algorithm（一致性算法），438
  for decision task（判定任务），445
  for firing-squad problem（触发组问题），491
  for interactive consistency（交互一致性），492
termination detection（终止检测），10, 268, 346
  with token algorithm（令牌算法），285
  with wave algorithm（波动算法），273
time complexity（时间复杂度），209
  alpha-time（α时间），223
  chain-time（链时间），223
  one-time（单位时间），221
timer（计时器），87
timing error（定时错误），431
token algorithm（令牌算法）
  for election（选举），233, 260
  for ring（环），190
  for termination detection（终止检测），285
  traversal（遍历），203
token bus（令牌总线），23
token ring（令牌环），9, 23
topological awareness（拓扑意识），565
topological information（拓扑信息），69
topology（拓扑学），64, 556
torus（圆环），563
  anonymous（匿名），323
  election in（选举），266, 267, 422
  sense of direction in（方向侦听），204, 568
  traversal algorithm for（遍历算法），204
Toueg's algorithm（Toueg算法），113, 399
trace（跟踪），242, 319
transaction, database（事务，数据库），354
transient failures（瞬时故障），520
transition（转移），45, 46
  for probabilistic algorithm（概率算法），310
  for stabilizing system（稳定系统），521
  for synchronous system（同步系统），399
transport layer（传输层），21, 75
Transputer, 12, 145, 308
traversal algorithm（遍历算法），181, 184, 202
  election with（选举），260
tree（树），65, 560
  controller for（控制器），166
  election for（选举），230
  routing for（路由），133
  wave algorithm for（波动算法），191
tree edge（树边），560
tree-labeling scheme（树标号模式），133, 414

underlying computation（潜在计算），271
undirected graph（无向图），557
uniform stabilizing algorithm（一致稳定算法），526
univalent configuration（单价配置），438
UNIX system（UNIX系统），6
Update algorithm（更新算法），543
UUCP network（UUCP网络），6

validity for firing-squad problem（触发组问题的有效性），491
vector clock（向量时钟），63
vector order（向量顺序），63
vector-counting algorithm（向量计数算法），293
virtual ring（虚拟环），559
visit（of a wave）（波动访问），297

wait-free atomic variables（无等待的原子变量），16
WAN（wide-area network）（广域网），3
wave algorithm（波动算法），181
  centralized versus decentralized（集中式与分散式），218
  definition（定义），182

echo algorithm（回波算法），194
for arbitrary network（任意网），246
for clique（团），196
for directed network（有向网络），197
for hypercube（超立方体），205
for ring（环），190，245
for torus（圆环），204
for tree（树），191
overview（综述），218
phase algorithm（相位算法），197
polling algorithm（轮询算法），196
use in election（选举中使用），230
use in termination detection（终止检测中使用），273，281，297
weak accuracy（弱精确性），507
weak completeness（failure det.）（弱完全性、故障检测），517
weak fairness（弱公平性），49
weighted graph（带权图），558
well-founded set（良基集），53，72
wide-area network（广域网），4，6
wide-area network（WAN）（广域网），3
window（窗口），82
witness（证据），453