

Michel Raynal

# Distributed Algorithms for Message-Passing Systems

 Springer

# Distributed Algorithms for Message-Passing Systems

Michel Raynal

# Distributed Algorithms for Message-Passing Systems

 Springer

Michel Raynal  
Institut Universitaire de France  
IRISA-ISTIC  
Université de Rennes 1  
Rennes Cedex  
France

ISBN 978-3-642-38122-5

ISBN 978-3-642-38123-2 (eBook)

DOI 10.1007/978-3-642-38123-2

Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013942973

ACM Computing Classification (1998): F.1, D.1, B.3

© Springer-Verlag Berlin Heidelberg 2013

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media ([www.springer.com](http://www.springer.com))

# Preface

*La profusion des choses cachait la rareté des idées et l'usure des croyances.  
[...] Retenir quelque chose du temps où l'on ne sera plus.  
In Les années (2008), Annie Ernaux*

*Nel mezzo del cammin di nostra vita  
Mi ritrovai per una selva oscura,  
Ché la diritta via era smarrita.  
In La divina commedia (1307–1321), Dante Alighieri (1265–1321)*

*Wir müssen nichts sein, sondern alles werden wollen.  
Johann Wolfgang von Goethe (1749–1832)*

*Chaque génération, sans doute, se croit vouée à refaire le monde.  
La mienne sait pourtant qu'elle ne le refera pas. Mais sa tâche est peut-être plus grande.  
Elle consiste à empêcher que le monde ne se défasse.  
Speech at the Nobel Banquet, Stockholm, December 10, 1957, Albert Camus (1913–1960)*

*Rien n'est précaire comme vivre  
Rien comme être n'est passager  
C'est un peu fondre pour le givre  
Ou pour le vent être léger.  
J'arrive où je suis étranger.  
In Le voyage de Hollande (1965), Louis Aragon (1897–1982)*

**What Is Distributed Computing?** Distributed computing was born in the late 1970s when researchers and practitioners started taking into account the intrinsic characteristic of physically distributed systems. The field then emerged as a specialized research area distinct from networking, operating systems, and parallel computing.

*Distributed computing* arises when one has to solve a problem in terms of distributed entities (usually called processors, nodes, processes, actors, agents, sensors, peers, etc.) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved. While parallel computing and real-time computing can be characterized, respectively, by the terms *efficiency* and *on-time computing*, distributed computing can be characterized by the term *uncertainty*. This uncertainty is created by asynchrony, multiplicity of control flows,

absence of shared memory and global time, failure, dynamicity, mobility, etc. Mastering one form or another of uncertainty is pervasive in all distributed computing problems. A main difficulty in designing distributed algorithms comes from the fact that each entity cooperating in the achievement of a common goal cannot have instantaneous knowledge of the current state of the other entities; it can only know their past local states.

Although distributed algorithms are often made up of a few lines, their behavior can be difficult to understand and their properties hard to state and prove. Hence, distributed computing is not only a fundamental topic but also a challenging topic where simplicity, elegance, and beauty are first-class citizens.

**Why This Book?** While there are a lot of books on sequential computing (both on basic data structures, or algorithms), this is not the case in distributed computing. Most books on distributed computing consider advanced topics where the uncertainty inherent to distributed computing is created by the net effect of asynchrony and failures. It follows that these books are more appropriate for graduate students than for undergraduate students.

The aim of this book is to present in a comprehensive way basic notions, concepts and algorithms of distributed computing when the distributed entities cooperate by sending and receiving messages on top of an underlying network. In this case, the main difficulty comes from the physical distribution of the entities and the asynchrony of the environment in which they evolve.

**Audience** This book has been written primarily for people who are not familiar with the topic and the concepts that are presented. These include mainly:

- Senior-level undergraduate students and graduate students in computer science or computer engineering, who are interested in the principles and foundations of distributed computing.
- Practitioners and engineers who want to be aware of the state-of-the-art concepts, basic principles, mechanisms, and techniques encountered in distributed computing.

Prerequisites for this book include undergraduate courses on algorithms, and basic knowledge on operating systems. Selections of chapters for undergraduate and graduate courses are suggested in the section titled “How to Use This Book” in the Afterword.

**Content** As already indicated, this book covers algorithms, basic principles, and foundations of message-passing programming, i.e., programs where the entities communicate by sending and receiving messages through a network. The world is distributed, and the algorithmic thinking suited to distributed applications and systems is not reducible to sequential computing. Knowledge of the bases of distributed computing is becoming more important than ever as more and more computer applications are now distributed. The book is composed of six parts.

- The aim of the first part, which is made up of six chapters, is to give a feel for the nature of distributed algorithms, i.e., what makes them different from sequential or parallel algorithms. To that end, it mainly considers distributed graph algorithms. In this context, each node of the graph is a process, which has to compute a result whose meaning depends on the whole graph.

Basic distributed algorithms such as network traversals, shortest-path algorithms, vertex coloring, knot detection, etc., are first presented. Then, a general framework for distributed graph algorithms is introduced. A chapter is devoted to leader election algorithms on a ring network, and another chapter focuses on the navigation of a network by mobile objects.

- The second part is on the nature of distributed executions. It is made up of four chapters. In some sense, this part is the core of the book. It explains what a distributed execution is, the fundamental notion of a consistent global state, and the impossibility—without freezing the computation—of knowing whether a computed consistent global state has been passed through by the execution or not.

Then, this part of the book addresses an important issue of distributed computations, namely the notion of logical time: scalar (linear) time, vector time, and matrix time. Each type of time is analyzed and examples of their uses are given. A chapter, which extends the notion of a global state, is then devoted to asynchronous distributed checkpointing. Finally, the last chapter of this part shows how to simulate a synchronous system on top of an asynchronous system (such simulators are called synchronizers).

- The third part of the book is made up of two chapters devoted to distributed mutual exclusion and distributed resource allocation. Different families of permission-based mutual exclusion algorithms are presented. The notion of an adaptive algorithm is also introduced. The notion of a critical section with multiple entries, and the case of resources with a single or several instances is also presented. Associated deadlock prevention techniques are introduced.
- The fourth part of the book is on the definition and the implementation of communication operations whose abstraction level is higher than the simple send/receive of messages. These communication abstractions impose order constraints on message deliveries. Causal message delivery and total order broadcast are first presented in one chapter. Then, another chapter considers synchronous communication (also called rendezvous or logically instantaneous communication).
- The fifth part of the book, which is made up of two chapters, is on the detection of stable properties encountered in distributed computing. A stable property is a property that, once true, remains true forever. The properties which are studied are the detection of the termination of a distributed computation, and the detection of distributed deadlock. This part of the book is strongly related to the second part (which is devoted to the notion of a global state).
- The sixth and last part of the book, which is also made up of two chapters, is devoted to the notion of a distributed shared memory. The aim is here to provide the entities (processes) with a set of objects that allow them to cooperate at

an abstraction level more appropriate than the use of messages. Two consistency conditions, which can be associated with these objects, are presented and investigated, namely, atomicity (also called linearizability) and sequential consistency. Several algorithms implementing these consistency conditions are described.

To have a more complete feeling of the spirit of this book, the reader is invited to consult the section “The Aim of This Book” in the Afterword, which describes what it is hoped has been learned from this book. Each chapter starts with a short presentation and a list of the main keywords, and terminates with a summary of its content. Each of the six parts of the book is also introduced by a brief description of its aim and its technical content.

**Acknowledgments** This book originates from lecture notes for undergraduate and graduate courses on distributed computing that I give at the University of Rennes (France) and, as an invited professor, at several universities all over the world. I would like to thank the students for their questions that, in one way or another, have contributed to this book. I want also to thank Ronan Nugent (Springer) for his support and his help in putting it all together.

Last but not least (and maybe most importantly), I also want to thank all the researchers whose results are presented in this book. Without their work, this book would not exist.

Michel Raynal

Professeur des Universités  
Institut Universitaire de France  
IRISA-ISTIC, Université de Rennes 1  
Campus de Beaulieu, 35042, Rennes, France

March–October 2012

Rennes, Saint-Grégoire, Tokyo, Fukuoka (AINA’12), Arequipa (LATIN’12),  
Reykjavik (SIROCCO’12), Palermo (CISIS’12), Madeira (PODC’12), Lisbon,  
Douelle, Saint-Philibert, Rhodes Island (Europar’12),  
Salvador de Bahia (DISC’12), Mexico City (Turing Year at UNAM)

# Contents

## Part I Distributed Graph Algorithms

<b>1</b>	<b>Basic Definitions and Network Traversal Algorithms</b>	<b>3</b>
1.1	Distributed Algorithms	3
1.1.1	Definition	3
1.1.2	An Introductory Example: Learning the Communication Graph	6
1.2	Parallel Traversal: Broadcast and Convergecast	9
1.2.1	Broadcast and Convergecast	9
1.2.2	A Flooding Algorithm	10
1.2.3	Broadcast/Convergecast Based on a Rooted Spanning Tree	10
1.2.4	Building a Spanning Tree	12
1.3	Breadth-First Spanning Tree	16
1.3.1	Breadth-First Spanning Tree Built Without Centralized Control	17
1.3.2	Breadth-First Spanning Tree Built with Centralized Control	20
1.4	Depth-First Traversal	24
1.4.1	A Simple Algorithm	24
1.4.2	Application: Construction of a Logical Ring	27
1.5	Summary	32
1.6	Bibliographic Notes	32
1.7	Exercises and Problems	33
<b>2</b>	<b>Distributed Graph Algorithms</b>	<b>35</b>
2.1	Distributed Shortest Path Algorithms	35
2.1.1	A Distributed Adaptation of Bellman–Ford’s Shortest Path Algorithm	35
2.1.2	A Distributed Adaptation of Floyd–Warshall’s Shortest Paths Algorithm	38
2.2	Vertex Coloring and Maximal Independent Set	42
2.2.1	On Sequential Vertex Coloring	42

2.2.2	Distributed $(\Delta + 1)$ -Coloring of Processes . . . . .	43
2.2.3	Computing a Maximal Independent Set . . . . .	46
2.3	Knot and Cycle Detection . . . . .	50
2.3.1	Directed Graph, Knot, and Cycle . . . . .	50
2.3.2	Communication Graph, Logical Directed Graph, and Reachability . . . . .	51
2.3.3	Specification of the Knot Detection Problem . . . . .	51
2.3.4	Principle of the Knot/Cycle Detection Algorithm . . . . .	52
2.3.5	Local Variables . . . . .	53
2.3.6	Behavior of a Process . . . . .	54
2.4	Summary . . . . .	57
2.5	Bibliographic Notes . . . . .	58
2.6	Exercises and Problems . . . . .	58
<b>3</b>	<b>An Algorithmic Framework</b>	
	<b>to Compute Global Functions on a Process Graph</b> . . . . .	59
3.1	Distributed Computation of Global Functions . . . . .	59
3.1.1	Type of Global Functions . . . . .	59
3.1.2	Constraints on the Computation . . . . .	60
3.2	An Algorithmic Framework . . . . .	61
3.2.1	A Round-Based Framework . . . . .	61
3.2.2	When the Diameter Is Not Known . . . . .	64
3.3	Distributed Determination of Cut Vertices . . . . .	66
3.3.1	Cut Vertices . . . . .	66
3.3.2	An Algorithm Determining Cut Vertices . . . . .	67
3.4	Improving the Framework . . . . .	69
3.4.1	Two Types of Filtering . . . . .	69
3.4.2	An Improved Algorithm . . . . .	70
3.5	The Case of Regular Communication Graphs . . . . .	72
3.5.1	Tradeoff Between Graph Topology and Number of Rounds . . . . .	72
3.5.2	De Bruijn Graphs . . . . .	73
3.6	Summary . . . . .	75
3.7	Bibliographic Notes . . . . .	76
3.8	Problem . . . . .	76
<b>4</b>	<b>Leader Election Algorithms</b> . . . . .	77
4.1	The Leader Election Problem . . . . .	77
4.1.1	Problem Definition . . . . .	77
4.1.2	Anonymous Systems: An Impossibility Result . . . . .	78
4.1.3	Basic Assumptions and Principles of the Election Algorithms . . . . .	79
4.2	A Simple $O(n^2)$ Leader Election Algorithm for Unidirectional Rings . . . . .	79
4.2.1	Context and Principle . . . . .	79
4.2.2	The Algorithm . . . . .	80
4.2.3	Time Cost of the Algorithm . . . . .	80

4.2.4	Message Cost of the Algorithm . . . . .	81
4.2.5	A Simple Variant . . . . .	82
4.3	An $O(n \log n)$ Leader Election Algorithm for Bidirectional Rings . . . . .	83
4.3.1	Context and Principle . . . . .	83
4.3.2	The Algorithm . . . . .	84
4.3.3	Time and Message Complexities . . . . .	85
4.4	An $O(n \log n)$ Election Algorithm for Unidirectional Rings . . . . .	86
4.4.1	Context and Principles . . . . .	86
4.4.2	The Algorithm . . . . .	88
4.4.3	Discussion: Message Complexity and FIFO Channels . . . . .	89
4.5	Two Particular Cases . . . . .	89
4.6	Summary . . . . .	90
4.7	Bibliographic Notes . . . . .	90
4.8	Exercises and Problems . . . . .	91
<b>5</b>	<b>Mobile Objects Navigating a Network . . . . .</b>	<b>93</b>
5.1	Mobile Object in a Process Graph . . . . .	93
5.1.1	Problem Definition . . . . .	93
5.1.2	Mobile Object Versus Mutual Exclusion . . . . .	94
5.1.3	A Centralized (Home-Based) Algorithm . . . . .	94
5.1.4	The Algorithms Presented in This Chapter . . . . .	95
5.2	A Navigation Algorithm for a Complete Network . . . . .	96
5.2.1	Underlying Principles . . . . .	96
5.2.2	The Algorithm . . . . .	97
5.3	A Navigation Algorithm Based on a Spanning Tree . . . . .	100
5.3.1	Principles of the Algorithm: Tree Invariant and Proxy Behavior . . . . .	101
5.3.2	The Algorithm . . . . .	102
5.3.3	Discussion and Properties . . . . .	104
5.3.4	Proof of the Algorithm . . . . .	106
5.4	An Adaptive Navigation Algorithm . . . . .	108
5.4.1	The Adaptivity Property . . . . .	109
5.4.2	Principle of the Implementation . . . . .	109
5.4.3	An Adaptive Algorithm Based on a Distributed Queue . . . . .	111
5.4.4	Properties . . . . .	113
5.4.5	Example of an Execution . . . . .	114
5.5	Summary . . . . .	115
5.6	Bibliographic Notes . . . . .	115
5.7	Exercises and Problems . . . . .	116

## Part II Logical Time and Global States in Distributed Systems

<b>6</b>	<b>Nature of Distributed Computations and the Concept of a Global State . . . . .</b>	<b>121</b>
6.1	A Distributed Execution Is a Partial Order on Local Events . . . . .	122
6.1.1	Basic Definitions . . . . .	122

6.1.2	A Distributed Execution Is a Partial Order on Local Events	122
6.1.3	Causal Past, Causal Future, Concurrency, Cut	123
6.1.4	Asynchronous Distributed Execution with Respect to Physical Time	125
6.2	A Distributed Execution Is a Partial Order on Local States	127
6.3	Global State and Lattice of Global States	129
6.3.1	The Concept of a Global State	129
6.3.2	Lattice of Global States	129
6.3.3	Sequential Observations	131
6.4	Global States Including Process States and Channel States	132
6.4.1	Global State Including Channel States	132
6.4.2	Consistent Global State Including Channel States	133
6.4.3	Consistent Global State Versus Consistent Cut	134
6.5	On-the-Fly Computation of Global States	135
6.5.1	Global State Computation Is an Observation Problem	135
6.5.2	Problem Definition	136
6.5.3	On the Meaning of the Computed Global State	136
6.5.4	Principles of Algorithms Computing a Global State	137
6.6	A Global State Algorithm Suited to FIFO Channels	138
6.6.1	Principle of the Algorithm	138
6.6.2	The Algorithm	140
6.6.3	Example of an Execution	141
6.7	A Global State Algorithm Suited to Non-FIFO Channels	143
6.7.1	The Algorithm and Its Principles	144
6.7.2	How to Compute the State of the Channels	144
6.8	Summary	146
6.9	Bibliographic Notes	146
6.10	Exercises and Problems	147
<b>7</b>	<b>Logical Time in Asynchronous Distributed Systems</b>	<b>149</b>
7.1	Linear Time	149
7.1.1	Scalar (or Linear) Time	150
7.1.2	From Partial Order to Total Order: The Notion of a Timestamp	151
7.1.3	Relating Logical Time and Timestamps with Observations	152
7.1.4	Timestamps in Action: Total Order Broadcast	153
7.2	Vector Time	159
7.2.1	Vector Time and Vector Clocks	159
7.2.2	Vector Clock Properties	162
7.2.3	On the Development of Vector Time	163
7.2.4	Relating Vector Time and Global States	165
7.2.5	Vector Clocks in Action: On-the-Fly Determination of a Global State Property	166
7.2.6	Vector Clocks in Action: On-the-Fly Determination of the Immediate Predecessors	170
7.3	On the Size of Vector Clocks	173

- 7.3.1 A Lower Bound on the Size of Vector Clocks . . . . . 174
- 7.3.2 An Efficient Implementation of Vector Clocks . . . . . 176
- 7.3.3  $k$ -Restricted Vector Clock . . . . . 181
- 7.4 Matrix Time . . . . . 182
  - 7.4.1 Matrix Clock: Definition and Algorithm . . . . . 182
  - 7.4.2 A Variant of Matrix Time in Action: Discard Old Data . . . 184
- 7.5 Summary . . . . . 186
- 7.6 Bibliographic Notes . . . . . 186
- 7.7 Exercises and Problems . . . . . 187
- 8 Asynchronous Distributed Checkpointing . . . . . 189**
  - 8.1 Definitions and Main Theorem . . . . . 189
    - 8.1.1 Local and Global Checkpoints . . . . . 189
    - 8.1.2 Z-Dependency, Zigzag Paths, and Z-Cycles . . . . . 190
    - 8.1.3 The Main Theorem . . . . . 192
  - 8.2 Consistent Checkpointing Abstractions . . . . . 196
    - 8.2.1 Z-Cycle-Freedom . . . . . 196
    - 8.2.2 Rollback-Dependency Trackability . . . . . 197
    - 8.2.3 On Distributed Checkpointing Algorithms . . . . . 198
  - 8.3 Checkpointing Algorithms Ensuring Z-Cycle Prevention . . . . . 199
    - 8.3.1 An Operational Characterization of Z-Cycle-Freedom . . . 199
    - 8.3.2 A Property of a Particular Dating System . . . . . 199
    - 8.3.3 Two Simple Algorithms Ensuring Z-Cycle Prevention . . . 201
    - 8.3.4 On the Notion of an Optimal Algorithm  
for Z-Cycle Prevention . . . . . 203
  - 8.4 Checkpointing Algorithms  
Ensuring Rollback-Dependency Trackability . . . . . 203
    - 8.4.1 Rollback-Dependency Trackability (RDT) . . . . . 203
    - 8.4.2 A Simple Brute Force RDT Checkpointing Algorithm . . . 205
    - 8.4.3 The Fixed Dependency After Send (FDAS)  
RDT Checkpointing Algorithm . . . . . 206
    - 8.4.4 Still Reducing the Number of Forced Local Checkpoints . . 207
  - 8.5 Message Logging for Uncoordinated Checkpointing . . . . . 211
    - 8.5.1 Uncoordinated Checkpointing . . . . . 211
    - 8.5.2 To Log or Not to Log Messages on Stable Storage . . . . . 211
    - 8.5.3 A Recovery Algorithm . . . . . 214
    - 8.5.4 A Few Improvements . . . . . 215
  - 8.6 Summary . . . . . 216
  - 8.7 Bibliographic Notes . . . . . 216
  - 8.8 Exercises and Problems . . . . . 217
- 9 Simulating Synchrony on Top of Asynchronous Systems . . . . . 219**
  - 9.1 Synchronous Systems, Asynchronous Systems, and Synchronizers 219
    - 9.1.1 Synchronous Systems . . . . . 219
    - 9.1.2 Asynchronous Systems and Synchronizers . . . . . 221
    - 9.1.3 On the Efficiency Side . . . . . 222

9.2	Basic Principle for a Synchronizer . . . . .	223
9.2.1	The Main Problem to Solve . . . . .	223
9.2.2	Principle of the Solutions . . . . .	224
9.3	Basic Synchronizers: $\alpha$ and $\beta$ . . . . .	224
9.3.1	Synchronizer $\alpha$ . . . . .	224
9.3.2	Synchronizer $\beta$ . . . . .	227
9.4	Advanced Synchronizers: $\gamma$ and $\delta$ . . . . .	230
9.4.1	Synchronizer $\gamma$ . . . . .	230
9.4.2	Synchronizer $\delta$ . . . . .	234
9.5	The Case of Networks with Bounded Delays . . . . .	236
9.5.1	Context and Hypotheses . . . . .	236
9.5.2	The Problem to Solve . . . . .	237
9.5.3	Synchronizer $\lambda$ . . . . .	238
9.5.4	Synchronizer $\mu$ . . . . .	239
9.5.5	When the Local Physical Clocks Drift . . . . .	240
9.6	Summary . . . . .	242
9.7	Bibliographic Notes . . . . .	243
9.8	Exercises and Problems . . . . .	244

### Part III Mutual Exclusion and Resource Allocation

<b>10</b>	<b>Permission-Based Mutual Exclusion Algorithms . . . . .</b>	<b>247</b>
10.1	The Mutual Exclusion Problem . . . . .	247
10.1.1	Definition . . . . .	247
10.1.2	Classes of Distributed Mutex Algorithms . . . . .	248
10.2	A Simple Algorithm Based on Individual Permissions . . . . .	249
10.2.1	Principle of the Algorithm . . . . .	249
10.2.2	The Algorithm . . . . .	251
10.2.3	Proof of the Algorithm . . . . .	252
10.2.4	From Simple Mutex to Mutex on Classes of Operations . . . . .	255
10.3	Adaptive Mutex Algorithms Based on Individual Permissions . . . . .	256
10.3.1	The Notion of an Adaptive Algorithm . . . . .	256
10.3.2	A Timestamp-Based Adaptive Algorithm . . . . .	257
10.3.3	A Bounded Adaptive Algorithm . . . . .	259
10.3.4	Proof of the Bounded Adaptive Mutex Algorithm . . . . .	262
10.4	An Algorithm Based on Arbiter Permissions . . . . .	264
10.4.1	Permissions Managed by Arbiters . . . . .	264
10.4.2	Permissions Versus Quorums . . . . .	265
10.4.3	Quorum Construction . . . . .	266
10.4.4	An Adaptive Mutex Algorithm Based on Arbiter Permissions . . . . .	268
10.5	Summary . . . . .	273
10.6	Bibliographic Notes . . . . .	273
10.7	Exercises and Problems . . . . .	274

- 11 Distributed Resource Allocation . . . . . 277**
  - 11.1 A Single Resource with Several Instances . . . . . 277
    - 11.1.1 The  $k$ -out-of- $M$  Problem . . . . . 277
    - 11.1.2 Mutual Exclusion with Multiple Entries:
      - The 1-out-of- $M$  Mutex Problem . . . . . 278
    - 11.1.3 An Algorithm for the  $k$ -out-of- $M$  Mutex Problem . . . . . 280
    - 11.1.4 Proof of the Algorithm . . . . . 283
    - 11.1.5 From Mutex Algorithms to  $k$ -out-of- $M$  Algorithms . . . . . 285
  - 11.2 Several Resources with a Single Instance . . . . . 285
    - 11.2.1 Several Resources with a Single Instance . . . . . 286
    - 11.2.2 Incremental Requests for Single Instance Resources:
      - Using a Total Order . . . . . 287
    - 11.2.3 Incremental Requests for Single Instance Resources:
      - Reducing Process Waiting Chains . . . . . 290
    - 11.2.4 Simultaneous Requests for Single Instance Resources
      - and Static Sessions . . . . . 292
    - 11.2.5 Simultaneous Requests for Single Instance Resources
      - and Dynamic Sessions . . . . . 293
  - 11.3 Several Resources with Multiple Instances . . . . . 295
  - 11.4 Summary . . . . . 297
  - 11.5 Bibliographic Notes . . . . . 298
  - 11.6 Exercises and Problems . . . . . 299

**Part IV High-Level Communication Abstractions**

- 12 Order Constraints on Message Delivery . . . . . 303**
  - 12.1 The Causal Message Delivery Abstraction . . . . . 303
    - 12.1.1 Definition of Causal Message Delivery . . . . . 304
    - 12.1.2 A Causality-Based Characterization
      - of Causal Message Delivery . . . . . 305
    - 12.1.3 Causal Order
      - with Respect to Other Message Ordering Constraints . . . . . 306
  - 12.2 A Basic Algorithm for Point-to-Point Causal Message Delivery . . 306
    - 12.2.1 A Simple Algorithm . . . . . 306
    - 12.2.2 Proof of the Algorithm . . . . . 309
    - 12.2.3 Reduce the Size of Control Information
      - Carried by Messages . . . . . 310
  - 12.3 Causal Broadcast . . . . . 313
    - 12.3.1 Definition and a Simple Algorithm . . . . . 313
    - 12.3.2 The Notion of a Causal Barrier . . . . . 315
    - 12.3.3 Causal Broadcast with Bounded Lifetime Messages . . . . . 317
  - 12.4 The Total Order Broadcast Abstraction . . . . . 320
    - 12.4.1 Strong Total Order Versus Weak Total Order . . . . . 320
    - 12.4.2 An Algorithm Based on a Coordinator Process
      - or a Circulating Token . . . . . 322

- 12.4.3 An Inquiry-Based Algorithm . . . . . 324
- 12.4.4 An Algorithm for Synchronous Systems . . . . . 326
- 12.5 Playing with a Single Channel . . . . . 328
  - 12.5.1 Four Order Properties on a Channel . . . . . 328
  - 12.5.2 A General Algorithm Implementing These Properties . . . 329
- 12.6 Summary . . . . . 332
- 12.7 Bibliographic Notes . . . . . 332
- 12.8 Exercises and Problems . . . . . 333
- 13 Rendezvous (Synchronous) Communication . . . . . 335**
  - 13.1 The Synchronous Communication Abstraction . . . . . 335
    - 13.1.1 Definition . . . . . 335
    - 13.1.2 An Example of Use . . . . . 337
    - 13.1.3 A Message Pattern-Based Characterization . . . . . 338
    - 13.1.4 Types of Algorithms  
 Implementing Synchronous Communications . . . . . 341
  - 13.2 Algorithms for Nondeterministic Planned Interactions . . . . . 341
    - 13.2.1 Deterministic and Nondeterministic Communication  
 Contexts . . . . . 341
    - 13.2.2 An Asymmetric (Static) Client–Server Implementation . . 342
    - 13.2.3 An Asymmetric Token-Based Implementation . . . . . 345
  - 13.3 An Algorithm for Nondeterministic Forced Interactions . . . . . 350
    - 13.3.1 Nondeterministic Forced Interactions . . . . . 350
    - 13.3.2 A Simple Algorithm . . . . . 350
    - 13.3.3 Proof of the Algorithm . . . . . 352
  - 13.4 Rendezvous with Deadlines in Synchronous Systems . . . . . 354
    - 13.4.1 Synchronous Systems and Rendezvous with Deadline . . . 354
    - 13.4.2 Rendezvous with Deadline Between Two Processes . . . . 355
    - 13.4.3 Introducing Nondeterministic Choice . . . . . 358
    - 13.4.4 *n*-Way Rendezvous with Deadline . . . . . 360
  - 13.5 Summary . . . . . 361
  - 13.6 Bibliographic Notes . . . . . 361
  - 13.7 Exercises and Problems . . . . . 362
- Part V Detection of Properties on Distributed Executions**
- 14 Distributed Termination Detection . . . . . 367**
  - 14.1 The Distributed Termination Detection Problem . . . . . 367
    - 14.1.1 Process and Channel States . . . . . 367
    - 14.1.2 Termination Predicate . . . . . 368
    - 14.1.3 The Termination Detection Problem . . . . . 369
    - 14.1.4 Types and Structure of Termination Detection Algorithms . 369
  - 14.2 Termination Detection in the Asynchronous Atomic Model . . . . 370
    - 14.2.1 The Atomic Model . . . . . 370

- 14.2.2 The Four-Counter Algorithm . . . . . 371
- 14.2.3 The Counting Vector Algorithm . . . . . 373
- 14.2.4 The Four-Counter Algorithm  
vs. the Counting Vector Algorithm . . . . . 376
- 14.3 Termination Detection in Diffusing Computations . . . . . 376
  - 14.3.1 The Notion of a Diffusing Computation . . . . . 376
  - 14.3.2 A Detection Algorithm Suited to Diffusing Computations . 377
- 14.4 A General Termination Detection Algorithm . . . . . 378
  - 14.4.1 Wave and Sequence of Waves . . . . . 379
  - 14.4.2 A Reasoned Construction . . . . . 381
- 14.5 Termination Detection in a Very General Distributed Model . . . . 385
  - 14.5.1 Model and Nondeterministic Atomic Receive Statement . . 385
  - 14.5.2 The Predicate *fulfilled()* . . . . . 387
  - 14.5.3 Static vs. Dynamic Termination: Definition . . . . . 388
  - 14.5.4 Detection of Static Termination . . . . . 390
  - 14.5.5 Detection of Dynamic Termination . . . . . 393
- 14.6 Summary . . . . . 396
- 14.7 Bibliographic Notes . . . . . 396
- 14.8 Exercises and Problems . . . . . 397
- 15 Distributed Deadlock Detection . . . . . 401**
  - 15.1 The Deadlock Detection Problem . . . . . 401
    - 15.1.1 Wait-For Graph (WFG) . . . . . 401
    - 15.1.2 AND and OR Models Associated with Deadlock . . . . . 403
    - 15.1.3 Deadlock in the AND Model . . . . . 403
    - 15.1.4 Deadlock in the OR Model . . . . . 404
    - 15.1.5 The Deadlock Detection Problem . . . . . 404
    - 15.1.6 Structure of Deadlock Detection Algorithms . . . . . 405
  - 15.2 Deadlock Detection in the One-at-a-Time Model . . . . . 405
    - 15.2.1 Principle and Local Variables . . . . . 406
    - 15.2.2 A Detection Algorithm . . . . . 406
    - 15.2.3 Proof of the Algorithm . . . . . 407
  - 15.3 Deadlock Detection in the AND Communication Model . . . . . 408
    - 15.3.1 Model and Principle of the Algorithm . . . . . 409
    - 15.3.2 A Detection Algorithm . . . . . 409
    - 15.3.3 Proof of the Algorithm . . . . . 411
  - 15.4 Deadlock Detection in the OR Communication Model . . . . . 413
    - 15.4.1 Principle . . . . . 413
    - 15.4.2 A Detection Algorithm . . . . . 416
    - 15.4.3 Proof of the Algorithm . . . . . 419
  - 15.5 Summary . . . . . 421
  - 15.6 Bibliographic Notes . . . . . 421
  - 15.7 Exercises and Problems . . . . . 422

**Part VI Distributed Shared Memory**

- 16 Atomic Consistency (Linearizability) . . . . . 427**
  - 16.1 The Concept of a Distributed Shared Memory . . . . . 427
  - 16.2 The Atomicity Consistency Condition . . . . . 429
    - 16.2.1 What Is the Issue? . . . . . 429
    - 16.2.2 An Execution Is a Partial Order on Operations . . . . . 429
    - 16.2.3 Atomicity: Formal Definition . . . . . 430
  - 16.3 Atomic Objects Compose for Free . . . . . 432
  - 16.4 Message-Passing Implementations of Atomicity . . . . . 435
    - 16.4.1 Atomicity Based on
      - a Total Order Broadcast Abstraction . . . . . 435
    - 16.4.2 Atomicity of Read/Write Objects Based on
      - Server Processes . . . . . 437
    - 16.4.3 Atomicity Based on
      - a Server Process and Copy Invalidation . . . . . 438
    - 16.4.4 Introducing the Notion of an Owner Process . . . . . 439
    - 16.4.5 Atomicity Based on a Server Process and Copy Update . . 443
  - 16.5 Summary . . . . . 444
  - 16.6 Bibliographic Notes . . . . . 444
  - 16.7 Exercises and Problems . . . . . 445
  
- 17 Sequential Consistency . . . . . 447**
  - 17.1 Sequential Consistency . . . . . 447
    - 17.1.1 Definition . . . . . 447
    - 17.1.2 Sequential Consistency Is Not a Local Property . . . . . 449
    - 17.1.3 Partial Order for Sequential Consistency . . . . . 450
    - 17.1.4 Two Theorems
      - for Sequentially Consistent Read/Write Registers . . . . . 451
    - 17.1.5 From Theorems to Algorithms . . . . . 453
  - 17.2 Sequential Consistency from Total Order Broadcast . . . . . 453
    - 17.2.1 A Fast Read Algorithm for Read/Write Objects . . . . . 453
    - 17.2.2 A Fast Write Algorithm for Read/Write Objects . . . . . 455
    - 17.2.3 A Fast Enqueue Algorithm for Queue Objects . . . . . 456
  - 17.3 Sequential Consistency from a Single Server . . . . . 456
    - 17.3.1 The Single Server Is a Process . . . . . 456
    - 17.3.2 The Single Server Is a Navigating Token . . . . . 459
  - 17.4 Sequential Consistency with a Server per Object . . . . . 460
    - 17.4.1 Structural View . . . . . 460
    - 17.4.2 The Object Managers Must Cooperate . . . . . 461
    - 17.4.3 An Algorithm Based on the OO Constraint . . . . . 462
  - 17.5 A Weaker Consistency Condition: Causal Consistency . . . . . 464
    - 17.5.1 Definition . . . . . 464
    - 17.5.2 A Simple Algorithm . . . . . 466
    - 17.5.3 The Case of a Single Object . . . . . 467
  - 17.6 A Hierarchy of Consistency Conditions . . . . . 468

- 17.7 Summary . . . . . 468
- 17.8 Bibliographic Notes . . . . . 469
- 17.9 Exercises and Problems . . . . . 470
- Afterword** . . . . . 471
  - The Aim of This Book . . . . . 471
  - Most Important Concepts, Notions, and Mechanisms Presented in This Book . . . . . 471
  - How to Use This Book . . . . . 473
  - From Failure-Free Systems to Failure-Prone Systems . . . . . 474
  - A Series of Books . . . . . 474
- References** . . . . . 477
- Index** . . . . . 495

# Notation

no-op	no operation
skip	empty statement
process	program in action
$n$	number of processes
$e$	number of edges in the process graph
$D$	diameter of the process graph
$p_i$	process whose index is $i$
$id_i$	identity of process $p_i$ (very often $id_i = i$ )
$\tau$	time instant (with respect to an external observer)
$\langle a, b \rangle$	pair with two elements $a$ and $b$
$\xrightarrow{ev}$	causal precedence relation on events
$\xrightarrow{\sigma}$	causal precedence relation on local states
$\xrightarrow{zz}$	z-precedence relation on local checkpoints
$\xrightarrow{\Sigma}$	precedence relation on global states
Mutex	mutual exclusion
ABCD	small capital letters: message type (message tag)
$abcd_i$	italics lower-case letters: local variable of process $p_i$
$\langle m_1; \dots; m_q \rangle$	sequence of messages
$a_i[1..s]$	array of size $s$ (local to process $p_i$ )
<b>for each</b> $i \in \{1, \dots, m\}$	order irrelevant
<b>for each</b> $i$ <b>from</b> 1 <b>to</b> $m$	order relevant
<b>wait</b> ( $P$ )	<b>while</b> $\neg P$ <b>do</b> no-op <b>end while</b>
return ( $v$ )	returns $v$ and terminates the operation invocation
% blablabla %	comments
;	sequentiality operator between two statements
$\neg(a R b)$	relation $R$ does not include the pair $\langle a, b \rangle$

# List of Figures and Algorithms

Fig. 1.1	Three graph types of particular interest . . . . .	4
Fig. 1.2	Synchronous execution ( <i>left</i> ) vs. asynchronous ( <i>right</i> ) execution	5
Fig. 1.3	Learning the communication graph (code for $p_i$ ) . . . . .	7
Fig. 1.4	A simple flooding algorithm (code for $p_i$ ) . . . . .	10
Fig. 1.5	A rooted spanning tree . . . . .	11
Fig. 1.6	Tree-based broadcast/convergecast (code for $p_i$ ) . . . . .	11
Fig. 1.7	Construction of a rooted spanning tree (code for $p_i$ ) . . . . .	13
Fig. 1.8	<i>Left</i> : Underlying communication graph; <i>Right</i> : Spanning tree . .	14
Fig. 1.9	An execution of the algorithm constructing a spanning tree . . .	14
Fig. 1.10	Two different spanning trees built from the same communication graph . . . . .	16
Fig. 1.11	Construction of a breadth-first spanning tree without centralized control (code for $p_i$ ) . . . . .	18
Fig. 1.12	An execution of the algorithm of Fig. 1.11 . . . . .	19
Fig. 1.13	Successive waves launched by the root process $p_a$ . . . . .	21
Fig. 1.14	Construction of a breadth-first spanning tree with centralized control (starting code) . . . . .	22
Fig. 1.15	Construction of a breadth-first spanning tree with centralized control (code for a process $p_i$ ) . . . . .	22
Fig. 1.16	Depth-first traversal of a communication graph (code for $p_i$ ) . .	25
Fig. 1.17	Time and message optimal depth-first traversal (code for $p_i$ ) . .	27
Fig. 1.18	Management of the token at process $p_i$ . . . . .	29
Fig. 1.19	From a depth-first traversal to a ring (code for $p_i$ ) . . . . .	29
Fig. 1.20	Sense of direction of the ring and computation of routing tables .	30
Fig. 1.21	An example of a logical ring construction . . . . .	31
Fig. 1.22	An anonymous network . . . . .	34
Fig. 2.1	Bellman–Ford’s dynamic programming principle . . . . .	36
Fig. 2.2	A distributed adaptation of Bellman–Ford’s shortest path algorithm (code for $p_i$ ) . . . . .	37
Fig. 2.3	A distributed synchronous shortest path algorithm (code for $p_i$ ) .	38
Fig. 2.4	Floyd–Warshall’s sequential shortest path algorithm . . . . .	39

Fig. 2.5	The principle that underlies Floyd–Warshall’s shortest paths algorithm . . . . .	39
Fig. 2.6	Distributed Floyd–Warshall’s shortest path algorithm . . . . .	41
Fig. 2.7	Sequential $(\Delta + 1)$ -coloring of the vertices of a graph . . . . .	42
Fig. 2.8	Distributed $(\Delta + 1)$ -coloring from an initial $m$ -coloring where $n \geq m \geq \Delta + 2$ . . . . .	43
Fig. 2.9	One bit of control information when the channels are not FIFO . . . . .	45
Fig. 2.10	Examples of maximal independent sets . . . . .	46
Fig. 2.11	From $m$ -coloring to a maximal independent set (code for $p_i$ ) . . . . .	47
Fig. 2.12	Luby’s synchronous random algorithm for a maximal independent set (code for $p_i$ ) . . . . .	48
Fig. 2.13	Messages exchanged during three consecutive rounds . . . . .	49
Fig. 2.14	A directed graph with a knot . . . . .	51
Fig. 2.15	Possible message pattern during a knot detection . . . . .	53
Fig. 2.16	Asynchronous knot detection (code of $p_i$ ) . . . . .	55
Fig. 2.17	Knot/cycle detection: example . . . . .	57
Fig. 3.1	Computation of routing tables defined from distances (code for $p_i$ ) . . . . .	63
Fig. 3.2	A diameter-independent generic algorithm (code for $p_i$ ) . . . . .	65
Fig. 3.3	A process graph with three cut vertices . . . . .	66
Fig. 3.4	Determining cut vertices: principle . . . . .	67
Fig. 3.5	An algorithm determining the cut vertices (code for $p_i$ ) . . . . .	68
Fig. 3.6	A general algorithm with filtering (code for $p_i$ ) . . . . .	71
Fig. 3.7	The De Bruijn’s directed networks $dB(2,1)$ , $dB(2,2)$ , and $dB(2,3)$ . . . . .	74
Fig. 3.8	A generic algorithm for a De Bruijn’s communication graph (code for $p_i$ ) . . . . .	75
Fig. 4.1	Chang and Robert’s election algorithm (code for $p_i$ ) . . . . .	80
Fig. 4.2	Worst identity distribution for message complexity . . . . .	81
Fig. 4.3	A variant of Chang and Robert’s election algorithm (code for $p_i$ ) . . . . .	83
Fig. 4.4	Neighborhood of a process $p_i$ competing during round $r$ . . . . .	84
Fig. 4.5	Competitors at the end of round $r$ are at distance greater than $2^r$ . . . . .	84
Fig. 4.6	Hirschberg and Sinclair’s election algorithm (code for $p_i$ ) . . . . .	85
Fig. 4.7	Neighbor processes on the unidirectional ring . . . . .	87
Fig. 4.8	From the first to the second round . . . . .	87
Fig. 4.9	Dolev, Klawe, and Rodeh’s election algorithm (code for $p_i$ ) . . . . .	88
Fig. 4.10	Index-based randomized election (code for $p_i$ ) . . . . .	90
Fig. 5.1	Home-based three-way handshake mechanism . . . . .	95
Fig. 5.2	Structural view of the navigation algorithm (module at process $p_i$ ) . . . . .	98
Fig. 5.3	A navigation algorithm for a complete network (code for $p_i$ ) . . . . .	99
Fig. 5.4	Asynchrony involving a mobile object and request messages . . . . .	100
Fig. 5.5	Navigation tree: initial state . . . . .	101
Fig. 5.6	Navigation tree: after the object has moved to $p_c$ . . . . .	102
Fig. 5.7	Navigation tree: proxy role of a process . . . . .	102

Fig. 5.8 A spanning tree-based navigation algorithm (code for  $p_i$ ) . . . . . 104

Fig. 5.9 The case of non-FIFO channels . . . . . 105

Fig. 5.10 The meaning of vector  
 $R = [d(i_1), d(i_2), \dots, d(i_{x-1}), d(i_x), 0, \dots, 0]$  . . . . . 108

Fig. 5.11 A dynamically evolving spanning tree . . . . . 110

Fig. 5.12 A navigation algorithm based on a distributed queue  
 (code for  $p_i$ ) . . . . . 112

Fig. 5.13 From the worst to the best case . . . . . 113

Fig. 5.14 Example of an execution . . . . . 114

Fig. 5.15 A hybrid navigation algorithm (code for  $p_i$ ) . . . . . 117

Fig. 6.1 A distributed execution as a partial order . . . . . 124

Fig. 6.2 Past, future, and concurrency sets associated with an event . . . . . 125

Fig. 6.3 Cut and consistent cut . . . . . 126

Fig. 6.4 Two instances of the same execution . . . . . 126

Fig. 6.5 Consecutive local states of a process  $p_i$  . . . . . 127

Fig. 6.6 From a relation on events to a relation on local states . . . . . 128

Fig. 6.7 A two-process distributed execution . . . . . 130

Fig. 6.8 Lattice of consistent global states . . . . . 130

Fig. 6.9 Sequential observations of a distributed computation . . . . . 131

Fig. 6.10 Illustrating the notations “ $e \in \sigma_i$ ” and “ $f \notin \sigma_i$ ” . . . . . 133

Fig. 6.11 In-transit and orphan messages . . . . . 133

Fig. 6.12 Cut versus global state . . . . . 135

Fig. 6.13 Global state computation: structural view . . . . . 136

Fig. 6.14 Recording of a local state . . . . . 139

Fig. 6.15 Reception of a MARKER() message: case 1 . . . . . 139

Fig. 6.16 Reception of a MARKER() message: case 2 . . . . . 139

Fig. 6.17 Global state computation (FIFO channels, code for  $cp_i$ ) . . . . . 140

Fig. 6.18 A simple automaton for process  $p_i$  ( $i = 1, 2$ ) . . . . . 141

Fig. 6.19 Prefix of a simple execution . . . . . 142

Fig. 6.20 Superimposing a global state computation  
 on a distributed execution . . . . . 142

Fig. 6.21 Consistent cut associated with the computed global state . . . . . 143

Fig. 6.22 A rubber band transformation . . . . . 143

Fig. 6.23 Global state computation (non-FIFO channels, code for  $cp_i$ ) . . . . . 145

Fig. 6.24 Example of a global state computation (non-FIFO channels) . . . . . 145

Fig. 6.25 Another global state computation  
 (non-FIFO channels, code for  $cp_i$ ) . . . . . 148

Fig. 7.1 Implementation of a linear clock (code for process  $p_i$ ) . . . . . 150

Fig. 7.2 A simple example of a linear clock system . . . . . 151

Fig. 7.3 A non-sequential observation obtained from linear time . . . . . 152

Fig. 7.4 A sequential observation obtained from timestamps . . . . . 153

Fig. 7.5 Total order broadcast: the problem that has to be solved . . . . . 155

Fig. 7.6 Structure of the total order broadcast implementation . . . . . 155

Fig. 7.7 Implementation of total order broadcast (code for process  $p_i$ ) . . . . . 157

Fig. 7.8 To\_delivery predicate of a message at process  $p_i$  . . . . . 157

Fig. 7.9	Implementation of a vector clock system (code for process $p_i$ ) . . . . .	160
Fig. 7.10	Time propagation in a vector clock system . . . . .	161
Fig. 7.11	On the development of time (1) . . . . .	164
Fig. 7.12	On the development of time (2) . . . . .	164
Fig. 7.13	Associating vector dates with global states . . . . .	165
Fig. 7.14	First global state satisfying a global predicate (1) . . . . .	167
Fig. 7.15	First global state satisfying a global predicate (2) . . . . .	168
Fig. 7.16	Detection the first global state satisfying $\bigwedge_i LP_i$ (code for process $p_i$ ) . . . . .	169
Fig. 7.17	Relevant events in a distributed computation . . . . .	171
Fig. 7.18	Vector clock system for relevant events (code for process $p_i$ ) . . . . .	171
Fig. 7.19	From relevant events to Hasse diagram . . . . .	171
Fig. 7.20	Determination of the immediate predecessors (code for process $p_i$ ) . . . . .	172
Fig. 7.21	Four possible cases when updating $imp_i[k]$ , while $vc_i[k] = vc[k]$ . . . . .	173
Fig. 7.22	A specific communication pattern . . . . .	175
Fig. 7.23	Specific communication pattern with $n = 3$ processes . . . . .	175
Fig. 7.24	Management of $vc_i[1..n]$ and $kprime_i[1..n, 1..n]$ (code for process $p_i$ ): Algorithm 1 . . . . .	178
Fig. 7.25	Management of $vc_i[1..n]$ and $kprime_i[1..n, 1..n]$ (code for process $p_i$ ): Algorithm 2 . . . . .	179
Fig. 7.26	An adaptive communication layer (code for process $p_i$ ) . . . . .	181
Fig. 7.27	Implementation of a $k$ -restricted vector clock system (code for process $p_i$ ) . . . . .	182
Fig. 7.28	Matrix time: an example . . . . .	183
Fig. 7.29	Implementation of matrix time (code for process $p_i$ ) . . . . .	184
Fig. 7.30	Discarding obsolete data: structural view (at a process $p_i$ ) . . . . .	185
Fig. 7.31	A buffer management algorithm (code for process $p_i$ ) . . . . .	185
Fig. 7.32	Yet another clock system (code for process $p_i$ ) . . . . .	188
Fig. 8.1	A checkpoint and communication pattern (with intervals) . . . . .	190
Fig. 8.2	A zigzag pattern . . . . .	192
Fig. 8.3	Proof of Theorem 9: a zigzag path joining two local checkpoints of $LC$ . . . . .	194
Fig. 8.4	Proof of Theorem 9: a zigzag path joining two local checkpoints . . . . .	195
Fig. 8.5	Domino effect (in a system of two processes) . . . . .	196
Fig. 8.6	Proof by contradiction of Theorem 11 . . . . .	200
Fig. 8.7	A very simple z-cycle-free checkpointing algorithm (code for $p_i$ ) . . . . .	201
Fig. 8.8	To take or not to take a forced local checkpoint . . . . .	202
Fig. 8.9	An example of z-cycle prevention . . . . .	202
Fig. 8.10	A vector clock system for rollback-dependency trackability (code for $p_i$ ) . . . . .	204
Fig. 8.11	Intervals and vector clocks for rollback-dependency trackability . . . . .	204

Fig. 8.12	Russell's pattern for ensuring the RDT consistency condition . . .	205
Fig. 8.13	Russell's checkpointing algorithm (code for $p_i$ ) . . . . .	205
Fig. 8.14	FDAS checkpointing algorithm (code for $p_i$ ) . . . . .	207
Fig. 8.15	Matrix $causal_i[1..n, 1..n]$ . . . . .	208
Fig. 8.16	Pure ( <i>left</i> ) vs. impure ( <i>right</i> ) causal paths from $p_j$ to $p_i$ . . . . .	208
Fig. 8.17	An impure causal path from $p_i$ to itself . . . . .	209
Fig. 8.18	An efficient checkpointing algorithm for RDT (code for $p_i$ ) . . . . .	210
Fig. 8.19	Sender-based optimistic message logging . . . . .	212
Fig. 8.20	To log or not to log a message? . . . . .	212
Fig. 8.21	An uncoordinated checkpointing algorithm (code for $p_i$ ) . . . . .	214
Fig. 8.22	Retrieving the messages which are in transit with respect to the pair $(c_i, c_j)$ . . . . .	215
Fig. 9.1	A space-time diagram of a synchronous execution . . . . .	220
Fig. 9.2	Synchronous breadth-first traversal algorithm (code for $p_i$ ) . . . . .	221
Fig. 9.3	Synchronizer: from asynchrony to logical synchrony . . . . .	222
Fig. 9.4	Synchronizer $\alpha$ (code for $p_i$ ) . . . . .	226
Fig. 9.5	Synchronizer $\alpha$ : possible message arrival at process $p_i$ . . . . .	227
Fig. 9.6	Synchronizer $\beta$ (code for $p_i$ ) . . . . .	229
Fig. 9.7	A message pattern which can occur with synchronizer $\beta$ (but not with $\alpha$ ): Case 1 . . . . .	229
Fig. 9.8	A message pattern which can occur with synchronizer $\beta$ (but not with $\alpha$ ): Case 2 . . . . .	229
Fig. 9.9	Synchronizer $\gamma$ : a communication graph . . . . .	230
Fig. 9.10	Synchronizer $\gamma$ : a partitioning . . . . .	231
Fig. 9.11	Synchronizer $\gamma$ (code for $p_i$ ) . . . . .	233
Fig. 9.12	Synchronizer $\delta$ (code for $p_i$ ) . . . . .	235
Fig. 9.13	Initialization of physical clocks (code for $p_i$ ) . . . . .	236
Fig. 9.14	The scenario to be prevented . . . . .	237
Fig. 9.15	Interval during which a process can receive pulse $r$ messages . . . . .	238
Fig. 9.16	Synchronizer $\lambda$ (code for $p_i$ ) . . . . .	239
Fig. 9.17	Synchronizer $\mu$ (code for $p_i$ ) . . . . .	240
Fig. 9.18	Clock drift with respect to reference time . . . . .	241
Fig. 10.1	A mutex invocation pattern and the three states of a process . . . . .	248
Fig. 10.2	Mutex module at a process $p_i$ : structural view . . . . .	250
Fig. 10.3	A mutex algorithm based on individual permissions (code for $p_i$ ) . . . . .	251
Fig. 10.4	Proof of the safety property of the algorithm of Fig. 10.3 . . . . .	253
Fig. 10.5	Proof of the liveness property of the algorithm of Fig. 10.3 . . . . .	254
Fig. 10.6	Generalized mutex based on individual permissions (code for $p_i$ ) . . . . .	256
Fig. 10.7	An adaptive mutex algorithm based on individual permissions (code for $p_i$ ) . . . . .	258
Fig. 10.8	Non-FIFO channel in the algorithm of Fig. 10.7 . . . . .	259
Fig. 10.9	States of the message PERMISSION( $\{i, j\}$ ) . . . . .	260

Fig. 10.10	A bounded adaptive algorithm based on individual permissions (code for $p_i$ ) . . . . .	261
Fig. 10.11	Arbiter permission-based mechanism . . . . .	265
Fig. 10.12	Values of $K$ and $D$ for symmetric optimal quorums . . . . .	266
Fig. 10.13	An order two projective plane . . . . .	267
Fig. 10.14	A safe (but not live) mutex algorithm based on arbiter permissions (code for $p_i$ ) . . . . .	269
Fig. 10.15	Permission preemption to prevent deadlock . . . . .	270
Fig. 10.16	A mutex algorithm based on arbiter permissions (code for $p_i$ ) . . . . .	272
Fig. 11.1	An algorithm for the multiple entries mutex problem (code for $p_i$ ) . . . . .	279
Fig. 11.2	Sending pattern of NOT_USED() messages: Case 1 . . . . .	281
Fig. 11.3	Sending pattern of NOT_USED() messages: Case 2 . . . . .	282
Fig. 11.4	An algorithm for the $k$ -out-of- $M$ mutex problem (code for $p_i$ ) . . . . .	282
Fig. 11.5	Examples of conflict graphs . . . . .	286
Fig. 11.6	Global conflict graph . . . . .	287
Fig. 11.7	A deadlock scenario involving two processes and two resources . . . . .	288
Fig. 11.8	No deadlock with ordered resources . . . . .	289
Fig. 11.9	A particular pattern in using resources . . . . .	289
Fig. 11.10	Conflict graph for six processes, each resource being shared by two processes . . . . .	290
Fig. 11.11	Optimal vertex-coloring of a resource graph . . . . .	291
Fig. 11.12	Conflict graph for static sessions ( $SS\_CG$ ) . . . . .	292
Fig. 11.13	Simultaneous requests in dynamic sessions (sketch of code for $p_i$ ) . . . . .	296
Fig. 11.14	Algorithms for generalized $k$ -out-of- $M$ (code for $p_i$ ) . . . . .	297
Fig. 11.15	Another algorithm for the $k$ -out-of- $M$ mutex problem (code for $p_i$ ) . . . . .	299
Fig. 12.1	The causal message delivery order property . . . . .	304
Fig. 12.2	The delivery pattern prevented by the empty interval property . . . . .	305
Fig. 12.3	Structure of a causal message delivery implementation . . . . .	307
Fig. 12.4	An implementation of causal message delivery (code for $p_i$ ) . . . . .	308
Fig. 12.5	Message pattern for the proof of the causal order delivery . . . . .	309
Fig. 12.6	An implementation reducing the size of control information (code for $p_i$ ) . . . . .	312
Fig. 12.7	Control information carried by consecutive messages sent by $p_j$ to $p_i$ . . . . .	312
Fig. 12.8	An adaptive sending procedure for causal message delivery . . . . .	313
Fig. 12.9	Illustration of causal broadcast . . . . .	313
Fig. 12.10	A simple algorithm for causal broadcast (code for $p_i$ ) . . . . .	314
Fig. 12.11	The causal broadcast algorithm in action . . . . .	315
Fig. 12.12	The graph of immediate predecessor messages . . . . .	316
Fig. 12.13	A causal broadcast algorithm based on causal barriers (code for $p_i$ ) . . . . .	317
Fig. 12.14	Message with bounded lifetime . . . . .	318

Fig. 12.15 On-time versus too late . . . . . 319

Fig. 12.16 A  $\Delta$ -causal broadcast algorithm (code for  $p_i$ ) . . . . . 320

Fig. 12.17 Implementation of total order message delivery requires  
coordination . . . . . 321

Fig. 12.18 Total order broadcast based on a coordinator process . . . . . 322

Fig. 12.19 Token-based total order broadcast . . . . . 323

Fig. 12.20 Clients and servers in total order broadcast . . . . . 324

Fig. 12.21 A total order algorithm from clients  $p_i$  to servers  $q_j$  . . . . . 326

Fig. 12.22 A total order algorithm from synchronous systems . . . . . 327

Fig. 12.23 Message  $m$  with type `ct_future` (cannot be bypassed) . . . . . 329

Fig. 12.24 Message  $m$  with type `ct_past`  
(cannot bypass other messages) . . . . . 329

Fig. 12.25 Message  $m$  with type `marker` . . . . . 329

Fig. 12.26 Building a *first in first out* channel . . . . . 330

Fig. 12.27 Message delivery according to message types . . . . . 331

Fig. 12.28 Delivery of messages typed `ordinary` and `ct_future` . . . . . 332

Fig. 13.1 Synchronous communication:  
messages as “points” instead of “intervals” . . . . . 336

Fig. 13.2 When communications are not synchronous . . . . . 338

Fig. 13.3 Accessing objects with synchronous communication . . . . . 339

Fig. 13.4 A crown of size  $k = 2$  (*left*) and a crown of size  $k = 3$  (*right*) . . . . . 339

Fig. 13.5 Four message patterns . . . . . 340

Fig. 13.6 Implementation of a rendezvous when the client is the sender . . . . . 344

Fig. 13.7 Implementation of a rendezvous when the client is the receiver . . . . . 345

Fig. 13.8 A token-based mechanism to implement an interaction . . . . . 346

Fig. 13.9 Deadlock and livelock prevention in interaction implementation . . . . . 347

Fig. 13.10 A general token-based implementation for planned interactions  
(rendezvous) . . . . . 348

Fig. 13.11 An algorithm for forced interactions (rendezvous) . . . . . 351

Fig. 13.12 Forced interaction: message pattern when  $i > j$  . . . . . 352

Fig. 13.13 Forced interaction: message pattern when  $i < j$  . . . . . 352

Fig. 13.14 When the rendezvous must be successful  
(two-process symmetric algorithm) . . . . . 356

Fig. 13.15 Real-time rendezvous between two processes  $p$  and  $q$  . . . . . 357

Fig. 13.16 When the rendezvous must be successful  
(asymmetric algorithm) . . . . . 358

Fig. 13.17 Nondeterministic rendezvous with deadline . . . . . 359

Fig. 13.18 Multirendezvous with deadline . . . . . 361

Fig. 13.19 Comparing two date patterns for rendezvous with deadline . . . . . 363

Fig. 14.1 Process states for termination detection . . . . . 368

Fig. 14.2 Global structure of the observation modules . . . . . 370

Fig. 14.3 An execution in the asynchronous atomic model . . . . . 371

Fig. 14.4 One visit is not sufficient . . . . . 371

Fig. 14.5 The four-counter algorithm for termination detection . . . . . 372

Fig. 14.6 Two consecutive inquiries . . . . . 373

Fig. 14.7	The counting vector algorithm for termination detection . . . . .	374
Fig. 14.8	The counting vector algorithm at work . . . . .	375
Fig. 14.9	Termination detection of a diffusing computation . . . . .	378
Fig. 14.10	Ring-based implementation of a wave . . . . .	380
Fig. 14.11	Spanning tree-based implementation of a wave . . . . .	381
Fig. 14.12	Why $(\bigwedge_{1 \leq i \leq n} idle_i^x) \Rightarrow TERM(\mathcal{C}, \tau_\alpha^x)$ is not true . . . . .	382
Fig. 14.13	A general algorithm for termination detection . . . . .	384
Fig. 14.14	Atomicity associated with $\tau_i^x$ . . . . .	384
Fig. 14.15	Structure of the channels to $p_i$ . . . . .	386
Fig. 14.16	An algorithm for static termination detection . . . . .	391
Fig. 14.17	Definition of time instants for the safety of static termination . . . . .	392
Fig. 14.18	Cooperation between local observers . . . . .	394
Fig. 14.19	An algorithm for dynamic termination detection . . . . .	395
Fig. 14.20	Example of a monotonous distributed computation . . . . .	398
Fig. 15.1	Examples of wait-for graphs . . . . .	402
Fig. 15.2	An algorithm for deadlock detection in the AND communication model . . . . .	410
Fig. 15.3	Determining in-transit messages . . . . .	411
Fig. 15.4	PROBE () messages sent along a cycle (with no application messages in transit) . . . . .	411
Fig. 15.5	Time instants in the proof of the safety property . . . . .	412
Fig. 15.6	A directed communication graph . . . . .	414
Fig. 15.7	Network traversal with feedback on a static graph . . . . .	414
Fig. 15.8	Modification in a wait-for graph . . . . .	415
Fig. 15.9	Inconsistent observation of a dynamic wait-for graph . . . . .	416
Fig. 15.10	An algorithm for deadlock detection in the OR communication model . . . . .	418
Fig. 15.11	Activation pattern for the safety proof . . . . .	420
Fig. 15.12	Another example of a wait-for graph . . . . .	423
Fig. 16.1	Structure of a distributed shared memory . . . . .	428
Fig. 16.2	Register: What values can be returned by read operations? . . . . .	429
Fig. 16.3	The relation $\xrightarrow{op}$ of the computation described in Fig. 16.2 . . . . .	430
Fig. 16.4	An execution of an atomic register . . . . .	432
Fig. 16.5	Another execution of an atomic register . . . . .	432
Fig. 16.6	Atomicity allows objects to compose for free . . . . .	435
Fig. 16.7	From total order broadcast to atomicity . . . . .	436
Fig. 16.8	Why read operations have to be to-broadcast . . . . .	437
Fig. 16.9	Invalidation-based implementation of atomicity: message flow . . . . .	438
Fig. 16.10	Invalidation-based implementation of atomicity: algorithm . . . . .	440
Fig. 16.11	Invalidation and owner-based implementation of atomicity (code of $p_i$ ) . . . . .	441
Fig. 16.12	Invalidation and owner-based implementation of atomicity (code of the manager $p_X$ ) . . . . .	442
Fig. 16.13	Update-based implementation of atomicity . . . . .	443

Fig. 16.14 Update-based algorithm implementing atomicity . . . . . 444

Fig. 17.1 A sequentially consistent computation (which is not atomic) . . . 448

Fig. 17.2 A computation which is not sequentially consistent . . . . . 449

Fig. 17.3 A sequentially consistent queue . . . . . 449

Fig. 17.4 Sequential consistency is not a local property . . . . . 450

Fig. 17.5 Part of the graph  $G$  used in the proof of Theorem 29 . . . . . 452

Fig. 17.6 Fast read algorithm implementing sequential consistency  
(code for  $p_i$ ) . . . . . 454

Fig. 17.7 Fast write algorithm implementing sequential consistency  
(code for  $p_i$ ) . . . . . 456

Fig. 17.8 Fast enqueue algorithm implementing a sequentially consistent  
queue (code for  $p_i$ ) . . . . . 457

Fig. 17.9 Read/write sequentially consistent registers  
from a central manager . . . . . 458

Fig. 17.10 Pattern of read/write accesses used in the proof of Theorem 33 . 459

Fig. 17.11 Token-based sequentially consistent shared memory  
(code for  $p_i$ ) . . . . . 460

Fig. 17.12 Architectural view associated with the OO constraint . . . . . 461

Fig. 17.13 Why the object managers must cooperate . . . . . 461

Fig. 17.14 Sequential consistency with a manager per object:  
process side . . . . . 462

Fig. 17.15 Sequential consistency with a manager per object:  
manager side . . . . . 463

Fig. 17.16 Cooperation between managers is required by the OO constraint 464

Fig. 17.17 An example of a causally consistent computation . . . . . 465

Fig. 17.18 Another example of a causally consistent computation . . . . . 466

Fig. 17.19 A simple algorithm implementing causal consistency . . . . . 467

Fig. 17.20 Causal consistency for a single object . . . . . 467

Fig. 17.21 Hierarchy of consistency conditions . . . . . 469

# Part I

## Distributed Graph Algorithms

This first part of the book is on distributed graph algorithms. These algorithms consider the distributed system as a connected graph whose vertices are the processes (nodes) and whose edges are the communication channels. It is made up of five chapters.

After having introduced base definitions, Chap. 1 addresses network traversals. It presents distributed algorithms that realize parallel, depth-first, and breadth-first network traversals. Chapter 2 is on distributed algorithms solving classical graph problems such as shortest paths, vertex coloring, maximal independent set, and knot detection. This chapter shows that the distributed techniques to solve graph problems are not obtained by a simple extension of their sequential counterparts.

Chapter 3 presents a general technique to compute a global function on a process graph, each process providing its own input parameter, and obtaining its own output (which depends on the whole set of inputs). Chapter 4 is on the leader election problems with a strong emphasis on uni/bidirectional rings. Finally, the last chapter of this part, Chap. 5, presents several algorithms that allow a mobile object to navigate a network.

In addition to the presentation of distributed graph algorithms, which can be used in distributed applications, an aim of this part of the book is to allow readers to have a better intuition of the term *distributed* when comparing distributed algorithms and sequential algorithms.

# Chapter 1

## Basic Definitions and Network Traversal Algorithms

This chapter first introduces basic definitions related to distributed algorithms. Then, considering a distributed system as a graph whose vertices are the processes and whose edges are the communication channels, it presents distributed algorithms for graph traversals, namely, parallel traversal, breadth-first traversal, and depth-first traversal. It also shows how spanning trees or rings can be constructed from these distributed graph traversal algorithms. These trees and rings can, in turn, be used to easily implement broadcast and convergecast algorithms.

As the reader will see, the distributed graph traversal techniques are different from their sequential counterparts in their underlying principles, behaviors, and complexities. This comes from the fact that, in a distributed context, the same type of traversal can usually be realized in distinct ways, each with its own tradeoff between its time complexity and message complexity.

**Keywords** Asynchronous/synchronous system · Breadth-first traversal · Broadcast · Convergecast · Depth-first traversal · Distributed algorithm · Forward/discard principle · Initial knowledge · Local algorithm · Parallel traversal · Spanning tree · Unidirectional logical ring

### 1.1 Distributed Algorithms

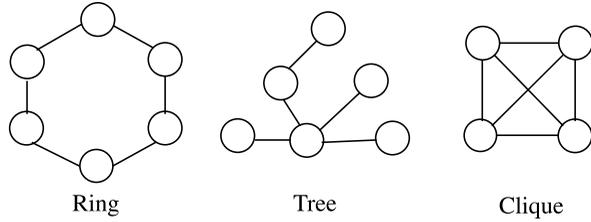
#### 1.1.1 Definition

**Processes** A distributed system is made up of a collection of computing units, each one abstracted through the notion of a *process*. The processes are assumed to cooperate on a common goal, which means that they exchange information in one way or another.

The set of processes is static. It is composed of  $n$  processes and denoted  $\Pi = \{p_1, \dots, p_n\}$ , where each  $p_i$ ,  $1 \leq i \leq n$ , represents a distinct process. Each process  $p_i$  is *sequential*, i.e., it executes one step at a time.

The integer  $i$  denotes the *index* of process  $p_i$ , i.e., the way an external observer can distinguish processes. It is nearly always assumed that each process  $p_i$  has its own identity, denoted  $id_i$ ; then  $p_i$  knows  $id_i$  (in a lot of cases—but not always— $id_i = i$ ).

**Fig. 1.1** Three graph types of particular interest



**Communication Medium** The processes communicate by sending and receiving *messages* through *channels*. Each channel is assumed to be reliable (it does not create, modify, or duplicate messages).

In some cases, we assume that channels are *first in first out* (FIFO) which means that the messages are received in the order in which they have been sent. Each channel is assumed to be bidirectional (can carry messages in both directions) and to have an infinite capacity (can contain any number of messages, each of any size). In some particular cases, we will consider channels which are unidirectional (such channels carry messages in one direction only).

Each process  $p_i$  has a set of neighbors, denoted  $neighbors_i$ . According to the context, this set contains either the local identities of the channels connecting  $p_i$  to its neighbor processes or the identities of these processes.

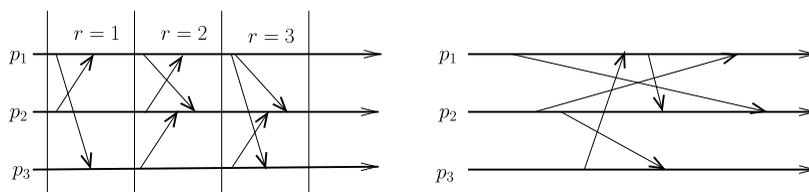
**Structural View** It follows from the previous definitions that, from a structural point of view, a distributed system can be represented by a connected undirected graph  $G = (I, C)$  (where  $C$  denotes the set of channels). Three types of graph are of particular interest (Fig. 1.1):

- A *ring* is a graph in which each process has exactly two neighbors with which it can communicate directly, a left neighbor and a right neighbor.
- A *tree* is a graph that has two noteworthy properties: it is acyclic and connected (which means that adding a new channel would create a cycle while suppressing a channel would disconnect it).
- A *fully connected* graph is a graph in which each process is directly connected to every other process. (In graph terminology, such a graph is called a clique.)

**Distributed Algorithm** A *distributed algorithm* is a collection of  $n$  automata, one per process. An automaton describes the sequence of steps executed by the corresponding process.

In addition to the power of a Turing machine, an automaton is enriched with two communication operations which allows it to send a message on a channel or receive a message on any channel. The operations are `send()` and `receive()`.

**Synchronous Algorithm** A distributed *synchronous* algorithm is an algorithm designed to be executed on a synchronous distributed system. The progress of such a system is governed by an external global clock, and the processes collectively execute a *sequence of rounds*, each round corresponding to a value of the global clock.



**Fig. 1.2** Synchronous execution (*left*) vs. asynchronous (*right*) execution

During a round, a process sends at most one message to each of its neighbors. The fundamental property of a *synchronous* system is that a message sent by a process during a round  $r$  is received by its destination process during the very same round  $r$ . Hence, when a process proceeds to the round  $r + 1$ , it has received (and processed) all the messages which have been sent to it during round  $r$ , and it knows that the same is true for any process.

**Space/time Diagram** A distributed execution can be graphically represented by what is called a *space/time diagram*. Each sequential progress is represented by an arrow from left to right, and a message is represented by an arrow from the sending process to the destination process. These notions will be made more precise in Chap. 6.

The space/time diagram on the left of Fig. 1.2 represents a synchronous execution. The vertical lines are used to separate the successive rounds. During the first round,  $p_1$  sends a message to  $p_3$ , and  $p_2$  sends a message to  $p_1$ , etc.

**Asynchronous Algorithm** A distributed *asynchronous* algorithm is an algorithm designed to be executed on an asynchronous distributed system. In such a system, there is no notion of an external time. That is why asynchronous systems are sometimes called *time-free* systems.

In an asynchronous algorithm, the progress of a process is ensured by its own computation and the messages it receives. When a process receives a message, it processes the message and, according to its local algorithm, possibly sends messages to its neighbors.

A process processes one message at a time. This means that the processing of a message cannot be interrupted by the arrival of another message. When a message arrives, it is added to the input buffer of the receiving process. It will be processed after all the messages that precede it in this buffer have been processed.

The space/time diagram of a simple asynchronous execution is depicted on the right of Fig. 1.2. One can see that, in this example, the messages from  $p_1$  to  $p_2$  are not received in their sending order. Hence, the channel from  $p_1$  to  $p_2$  is not a FIFO (first in first out) channel. It is easy to see from the figure that a synchronous execution is more structured than an asynchronous execution.

**Initial Knowledge of a Process** When solving a problem in a synchronous/asynchronous system, a process is characterized by its input parameters (which are related to the problem to solve) and its *initial knowledge* of its environment.

This knowledge concerns its identity, the total number  $n$  of processes, the identity of its neighbors, the structure of the communication graph, etc. As an example, a process  $p_i$  may only know that

- it is on a unidirectional ring,
- it has a left neighbor from which it can receive messages,
- it has a right neighbor to which it can send messages,
- its identity is  $id_i$ ,
- the fact that no two processes have the same identity, and
- the fact that the set of identities is totally ordered.

As we can see, with such an initial knowledge, no process initially knows the total number of processes  $n$ . Learning this number requires the processes to exchange information.

### ***1.1.2 An Introductory Example: Learning the Communication Graph***

As a simple example, this section presents an asynchronous algorithm that allows each process to learn the communication graph in which it evolves. It is assumed that the channels are bidirectional and that the communication graph is connected (there is a path from any process to any other process).

**Initial Knowledge** Each process  $p_i$  has identity  $id_i$ , and no process knows  $n$  (the total number of processes). Initially, a process  $p_i$  knows its identity and the identity  $id_j$  of each of its neighbors. Hence, each process  $p_i$  is initially provided with a set  $neighbors_i$  and, for each  $id_j \in neighbors_i$ , the pair  $\langle id_i, id_j \rangle$  denotes locally the channel connecting  $p_i$  to  $p_j$ . Let us observe that, as the channels are bidirectional, both  $\langle id_i, id_j \rangle$  and  $\langle id_j, id_i \rangle$  denote the same channel and are consequently considered as synonyms.

**The Forward/Discard Principle** The principle on which the algorithm relies is pretty simple: Each process initially sends its position in the graph to each of its neighbors. This position is represented by the pair  $(id_i, neighbors_i)$ .

Then, when a process  $p_i$  receives a pair  $(id_k, neighbors_k)$  for the first time, it updates its local representation of the communication graph and forwards the message it has received to all its neighbors (except the one that sends this message). This is the “when new, forward” principle. On the contrary, if it is not the first time that  $p_i$  receives the pair  $(id_k, neighbors_k)$ , it discards it. This is the “when not new, discard” principle.

When  $p_i$  has received a pair  $(id_k, neighbors_k)$ , we say that it “knows the position” of  $p_k$  in the graph. This means that it knows both the identity  $id_k$  and the channels connecting  $p_k$  to its neighbors.

```

operation start() is
(1) for each  $id_j \in neighbors_i$ ;
(2)   do send POSITION( $id_i, neighbors_i$ ) to the neighbor identified  $id_j$ 
(3) end for;
(4)  $part_i \leftarrow true$ 
end operation.

when START() is received do
(5) if ( $\neg part_i$ ) then start() end if.

when POSITION( $id, neighbors$ ) is received from neighbor identified  $id_x$  do
(6) if ( $\neg part_i$ ) then start() end if;
(7) if ( $id \notin proc\_known_i$ ) then
(8)    $proc\_known_i \leftarrow proc\_known_i \cup \{id\}$ ;
(9)    $channels\_known_i \leftarrow channels\_known_i \cup \{(id, id_k) \text{ such that } id_k \in neighbors_i\}$ ;
(10)  for each  $id_y \in neighbors_i \setminus \{id_x\}$ 
(11)    do send POSITION( $id, neighbors$ ) to the neighbor identified  $id_y$ 
(12)  end for;
(13)  if ( $\forall \langle id_j, id_k \rangle \in channels\_known_i : \{id_j, id_k\} \subseteq proc\_known_i$ )
(14)    then  $p_i$  knows the communication graph; return()
(15)  end if
(16) end if.

```

Fig. 1.3 Learning the communication graph (code for  $p_i$ )

**Local Representation of the Communication Graph** The graph is locally represented at each process  $p_i$  with two local variables.

- The local variable  $proc\_known_i$  is a set that contains all the processes whose position is known by  $p_i$ . Initially,  $proc\_known_i = \{id_i\}$ .
- The local variable  $channels\_known_i$  is a set that contains all the channels known by  $p_i$ . Initially,  $channels\_known_i = \{(id_i, id_j) \text{ such that } id_j \in neighbors_i\}$ .

Hence, after a process has received a message containing the pair  $(id_j, neighbors_j)$ , we have  $id_j \in proc\_known_i$  and  $\{(id_j, id_k) \text{ such that } id_k \in neighbors_j\} \subseteq channels\_known_i$ .

In addition to the local representation of the graph,  $p_i$  has a local Boolean variable  $part_i$ , initialized to *false*, which is set to *true* when  $p_i$  starts participating in the algorithm.

**Internal Versus External Messages** This participation of a process starts when it receives an external message START() or an internal message POSITION().

An *internal* message is a message generated by the algorithm, while an *external* message is a message coming from outside. External messages are used to launch the algorithm. It is assumed that at least one process receives such a message.

**Algorithm: Forward/Discard** The algorithm is described in Fig. 1.3. As previously indicated, when a process  $p_i$  receives a message START() or POSITION(), it starts participating in the algorithm if not yet done (line 5 or 6). To that end it sends

the message  $\text{POSITION}(id_i, neighbors_i)$  to each of its neighbors (line 2) and sets  $part_i$  to *true* (line 4).

When  $p_i$  receives a message  $\text{POSITION}(id, neighbors)$  from one of its neighbors  $p_x$  for the first time (line 7), it includes the position of the corresponding process  $p_j$  in the local data structures  $proc\_known_i$  and  $channels\_known_i$  (lines 8–9) and, as it has learned something new, it forwards this message  $\text{POSITION}()$  to all its neighbors, but the one that sent it this message (line 10). If it has already received the message  $\text{POSITION}(id, neighbors)$  (we have then  $j \in proc\_known_i$ ),  $p_i$  discards the message.

**Algorithm: Termination** As the communication graph is connected, it is easy to see that, as soon as a process receives a message  $\text{START}()$ , each process  $p_i$  will send a message  $\text{POSITION}(id_i, neighbors_i)$  which, from neighbor to neighbor, will be received by each process. Consequently, for any pair of processes  $(p_i, p_j)$ ,  $p_i$  will receive a message  $\text{POSITION}(id_j, neighbors_j)$ , from which it follows that any process  $p_i$  eventually learns the communication graph.

Moreover, as (a) there is a bounded number of processes  $n$ , (b) each process  $p_i$  is the only process to initiate the sending of the message  $\text{POSITION}(id_i, neighbors_i)$ , and (c) any process  $p_j$  forwards this message only once, it follows that there is a finite time after which no more message is sent. Consequently, the algorithm terminates at each process. While the algorithm always terminates, the important question is the following: When does a process know that it can stop participating in the algorithm? Trivially, a process can stop when it knows that it has learned the whole communication graph (due to the “forward” strategy, when a process knows the whole graph, it also knows that its neighbors eventually know it). This knowledge can be easily captured by a process  $p_i$  with the help of its local data structures  $proc\_known_i$  and  $channels\_known_i$ . More precisely, remembering that the pairs  $\langle id_i, id_j \rangle$  and  $\langle id_j, id_i \rangle$  are synonyms and using a classical graph closure property, a process  $p_i$  knows the whole graph when  $\forall \langle id_j, id_k \rangle \in channels\_known_i : \{id_j, id_k\} \subseteq proc\_known_i$ . This *local termination* predicate appears at line 13. When it becomes locally satisfied, a process  $p_i$  learns that it knows the whole graph and learns also that its neighbors eventually know it. That process can consequently stop its execution by invoking the statement  $\text{return}()$  line 14.

It is important to notice that the simplicity of the termination predicate comes from an appropriate choice of the local data structures ( $proc\_known_i$  and  $proc\_known_i$ ) used to represent the communication graph.

**Cost** Let  $e$  be the total number of channels and  $D$  be the diameter of the communication graph. The *diameter* of a graph is the longest among all the shortest distances connecting any pair of processes, where the shortest distance between  $p_i$  and  $p_j$  is the smallest number of channels to go from  $p_i$  to  $p_j$ . The diameter notion is a global notion that measures the “breadth” of the communication graph.

For any  $i$  and any channel, a message  $\text{POSITION}(id_i, -)$  is sent at least once and at most twice (once in each direction) on that channel. It follows that the message complexity is upper bounded by  $2ne$ .

As far as the time complexity is concerned, let us consider that each message takes one time unit and local processing has zero duration. In the worst case, a

single process  $p_k$  receives a message `START()` and there is a process  $p_\ell$  at distance  $D$  from  $p_k$ . In this case, it takes  $D$  time units for a message `POSITION( $id_k, -$ )` to arrive at  $p_\ell$ . This message wakes up  $p_\ell$ , and it takes then  $D$  time units for a message `POSITION( $id_\ell, -$ )` to arrive at  $p_k$ . It follows that the time complexity is upper bounded by  $2D$ .

Finally, let  $d$  be the *maximal degree* of the communication graph (i.e.,  $d = \max(\{|neighbors_i| \mid 1 \leq i \leq n\})$ , and  $b$  the number of bits required to encode any identity  $id_i$ . The maximal number of bits needed for a message `POSITION()` is  $b(d + 1)$ .

**When Initially the Channels Have Only Local Names** Let us consider a process  $p_i$  that has  $c_i$  neighbors to which it is point-to-point connected by  $c_i$  channels locally denoted  $channel_i[1..c_i]$ . When each process  $p_i$  is initially given only  $channel_i[1..c_i]$ , the processes can easily compute their sets  $neighbors_i$ . To that end, each process executes a preliminary communication phase during which it first sends a message `ID( $i$ )` on each  $channel_i[x]$ ,  $1 \leq x \leq c_i$ , and then waits until it has received the identities of the processes at the other end of its  $c_i$  channels. When  $p_i$  has received `ID( $id_k$ )` on channel  $channel_i[x]$ , it can associate its local address  $channel_i[x]$  with the identity  $id_k$  whose scope is the whole system.

**Port Name** When each channel  $channel_i[x]$  is defined by a local name, the index  $x$  is sometimes called a *port*. Hence, a process  $p_i$  has  $c_i$  communication ports.

## 1.2 Parallel Traversal: Broadcast and Convergecast

It is assumed that, while the identity of a process  $p_i$  is its index  $i$ , no process knows explicitly the value of  $n$  (i.e.,  $p_n$  knows that its identity is  $n$ , but does not know that its identity is also the number of processes).

### 1.2.1 Broadcast and Convergecast

Two frequent problems encountered in distributed computing are broadcast and convergecast. These two problems are defined with respect to a distinguished process  $p_a$ .

- The *broadcast* problem is a one-to-many communication problem. It consists in designing an algorithm that allows the distinguished process  $p_a$  to disseminate information to the whole set of processes.

A variant of the broadcast problem is the *multicast* problem. In that case, the distinguished process  $p_a$  has to disseminate information to a subset of the processes. This subset can be statically defined or dynamically defined at the time of the multicast invocation.

```

when GO(data) is received from  $p_k$  do
(1) if (first reception of go(data)) then
(2)   for each  $j \in \text{neighbors}_i \setminus \{k\}$  do send GO(data) to  $p_j$  end for
(3)   end if.

```

**Fig. 1.4** A simple flooding algorithm (code for  $p_i$ )

- The *convergecast* problem is a many-to-one communication problem. It consists in designing an algorithm that allows each process  $p_j$  to send information  $v_j$  to a distinguished process  $p_a$  for it to compute some function  $f()$ , which is on a vector  $[v_1, \dots, v_n]$  containing one value per process.

Broadcast and convergecast can be seen as dual communication operations. They are usually used as a pair:  $p_a$  broadcasts a query to obtain values, one from each process, from which it computes the resulting value  $f()$ . As a simple example,  $p_a$  is a process that queries sensors for temperature values, from which it computes output values (e.g., maximal, minimal and average temperature values).

## 1.2.2 A Flooding Algorithm

A simple way to implement a broadcast consists of what is called a *flooding* algorithm. Such an algorithm is described in Fig. 1.4. To simplify the description, the distinguished process  $p_a$  initially sends to itself a message denoted  $\text{GO}(\text{data})$ , which carries the information it wants to disseminate. Then, when a process  $p_i$  receives for the first time a copy of this message, it forwards the message to all its neighbors (except to the sender of the message).

Each message  $\text{GO}(\text{data})$  can be identified by a sequence number  $sn_a$ . Moreover, the flooding algorithm can be easily adapted to work with any number of distinguished processes by identifying each message broadcast by a distinguished process  $p_a$  with an identity pair  $(a, sn_a)$ .

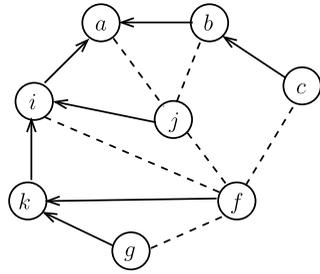
As the set of processes is assumed to be connected, it is easy to see that the algorithm described in Fig. 1.4 guarantees that the information sent by a distinguished process is eventually received exactly once by each process.

## 1.2.3 Broadcast/Convergecast Based on a Rooted Spanning Tree

The previous flooding algorithm may use up to  $2e - |\text{neighbors}_a|$  messages (where  $e$  is the number of channels), and is consequently not very efficient. A simple way to improve it consists of using an underlying spanning tree rooted at the distinguished process  $p_a$ .

**Fig. 1.5**

A rooted spanning tree



**Rooted Spanning Tree** A spanning tree rooted at  $p_a$  is a tree which contains  $n$  processes and whose channels (edges) are channels of the communication graph. Each process  $p_i$  has a single parent, locally denoted  $parent_i$ , and a (possibly empty) set of children, locally denoted  $children_i$ . To simplify the notation, the parent of the root is the root itself, i.e., the distinguished process  $p_a$  is the only process  $p_i$  such that  $parent_i = i$ . Moreover, if  $j \neq a$ , we have  $j \in children_i \Leftrightarrow parent_j = i$ , and the channel  $\langle i, j \rangle$  belongs to the communication graph.

An example of a rooted spanning tree is described in Fig. 1.5. The arrows (oriented toward the root) describe the channels of the communication graph that belong to the spanning tree. The dotted edges are the channels of the communication graph that do not belong to the spanning tree. This spanning tree rooted at  $p_a$  is such that, when considering the position of process  $p_i$  where  $neighbors_i = \{a, k, j, f\}$ , we have  $parent_i = a$ ,  $children_i = \{j, k\}$  and consequently  $parent_j = parent_k = i$ . Moreover,  $children_i \cup \{parent_i\} \subseteq neighbors_i = \{a, k, j, f\}$ .

**Algorithms** Given such a rooted spanning tree, the algorithms implementing a broadcast by  $p_a$  and the associated convergecast to  $p_a$  are described in Fig. 1.6. As far as the broadcast is concerned,  $p_a$  first sends the message  $GO(data)$  to itself, and then this message is forwarded along the channels of the spanning tree, and this restricted flooding stops at the leaves of the tree.

As far as the convergecast is concerned, each leaf  $p_i$  sends a message  $BACK(val\_set_i)$  to its parent (line 4) where  $val\_set_i = \{(i, v_i)\}$  (line 2), i.e.,  $val\_set_i$  contains a

```

===== Broadcast =====
when  $GO(data)$  is received from  $p_k$  do
(1) for each  $j \in children_i \setminus \{k\}$  do send  $GO(data)$  to  $p_j$  end for.

===== Convergecast =====
when  $BACK(val\_set_j)$  received from each  $p_j$  such that  $j \in children_i$  do
(2)  $val\_set_i \leftarrow (\bigcup_{j \in children_i} val\_set_j) \cup \{(i, v_i)\}$ ;
(3) let  $k = parent_i$ ;
(4) if  $(k \neq i)$  then send  $BACK(val\_set_i)$  to  $p_k$ 
(5) else the root  $p_i (= p_a)$  can compute  $f(val\_set_i)$ 
(6) end if.
    
```

**Fig. 1.6** Tree-based broadcast/convergecast (code for  $p_i$ )

single pair carrying the value  $v_i$  sent by  $p_i$  to the root. A non-leaf process  $p_i$  waits for the pairs  $(k, v_k)$  from all its children, adds its own pair  $(i, v_i)$ , and finally sends the resulting set  $val\_set_i$  to its parent (line 4). When the root has received a set of pairs from each of its children, it has a pair from each process and can compute the function  $f()$  (line 5).

### 1.2.4 Building a Spanning Tree

This section presents a simple algorithm that (a) implements broadcast and convergecast, and (b) builds a spanning tree. This algorithm is sometimes called *propagation of information with feedback*. Once a spanning tree has been constructed, it can be used for future broadcasts and convergecasts involving the same distinguished process  $p_a$ .

**Local Variables** As before, each process  $p_i$  is provided with a set  $neighbors_i$  which defines its position in the communication graph and, at the end of the execution, its local variables  $parent_i$  and  $children_i$  will define its position in the spanning tree rooted at  $p_a$ .

To compute its position in the spanning tree rooted at  $p_a$ , each process  $p_i$  uses an auxiliary integer local variable denoted  $expected\_msg_i$ . This variable contains the number of messages that  $p_i$  is waiting for from its children before sending a message `BACK()` to its parent.

**Algorithm** The broadcast/convergecast algorithm building a spanning tree is described in Fig. 1.7. To simplify the presentation, it is first assumed that the channels are FIFO (first in, first out). The distinguished process  $p_a$  is the only process which receives the external message `START()` (line 1). Upon its reception,  $p_a$  initializes  $parent_a$ ,  $children_a$  and  $expected\_msg_a$  and sends a message `GO(data)` to each of its neighbors (line 2).

When a process  $p_i$  receives a message `GO(data)` for the first time, it defines the sender  $p_j$  as its parent in the spanning tree, and initializes  $children_i$  to  $\emptyset$  and  $expected\_msg_i$  the number of its neighbors minus  $p_j$  (line 4). If its parent is its only neighbor, it sends back the pair  $(i, v_i)$  thereby indicating to  $p_j$  that it is one of its children (lines 5–6). Otherwise,  $p_i$  forwards the message `GO(data)` to all its neighbors but its parent  $p_j$  (line 7).

If  $parent_i \neq \perp$ , when  $p_i$  receives `GO(data)`, it has already determined its parent in the spanning tree and forwarded the message `GO(data)`. It consequently sends by return to  $p_j$  the message `BACK( $\emptyset$ )`, where  $\emptyset$  is used to indicate to  $p_j$  that  $p_i$  is not one of its children (line 9).

When a process  $p_i$  receives a message `BACK(res, val_set)` from a neighbor  $p_j$ , it decreases  $expected\_msg_i$  (line 11) and adds  $p_j$  to  $children_i$  if  $val\_set \neq \emptyset$  (line 12). Then, if  $p_i$  has received a message `BACK()` from all its neighbors (but its parent, line 13), it sends to its parent (lines 15–16) the set  $val\_set$  containing its own pair

```

when START() is received do   % only  $p_a$  receives this message %
(1)  $parent_i \leftarrow i$ ;  $children_i \leftarrow \emptyset$ ;  $expected\_msg_i \leftarrow |neighbors_i|$ ;
(2) for each  $j \in neighbors_i$  do send GO( $data$ ) to  $p_j$  end for.

when GO( $data$ ) is received from  $p_j$  do
(3) if ( $parent_i = \perp$ )
(4)   then  $parent_i \leftarrow j$ ;  $children_i \leftarrow \emptyset$ ;  $expected\_msg_i \leftarrow |neighbors_i| - 1$ ;
(5)     if ( $expected\_msg_i = 0$ )
(6)       then send BACK( $\{(i, v_i)\}$ ) to  $p_j$ 
(7)       else for each  $k \in neighbors_i \setminus \{j\}$  do send GO( $data$ ) to  $p_k$  end for
(8)     end if
(9)   else send BACK( $\emptyset$ ) to  $p_j$ 
(10) end if.

when BACK( $val\_set$ ) is received from  $p_j$  do
(11)  $expected\_msg_i \leftarrow expected\_msg_i - 1$ ;
(12) if ( $val\_set \neq \emptyset$ ) then  $children_i \leftarrow children_i \cup \{j\}$  end if;
(13) if ( $expected\_msg_i = 0$ ) then   % a set  $val\_set_x$  has been received from each child  $p_x$  %
(14)   let  $val\_set = (\bigcup_{x \in children_i} val\_set_x) \cup \{(i, v_i)\}$ ; let  $pr = parent_i$ ;
(15)   if ( $pr \neq i$ )
(16)     then send BACK( $val\_set$ ) to  $p_{pr}$            % local termination for  $p_i$  %
(17)     else  $p_i (= p_a)$  can compute  $f(val\_set)$  % global termination %
(18)   end if
(19) end if.

```

Fig. 1.7 Construction of a rooted spanning tree (code for  $p_i$ )

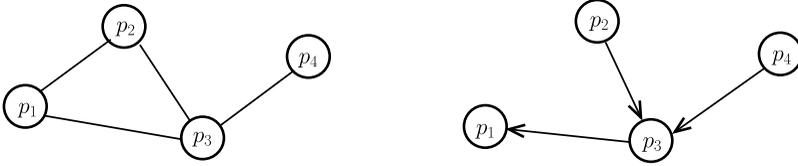
$(i, v_i)$  plus all the pairs  $(k, v_k)$  it has received from its children line 14). Then,  $p_i$  has terminated its participation in the algorithm (its local variable  $expected\_msg_i$  then becomes useless). If  $p_i$  is the distinguished process  $p_a$ , the set  $val\_set$  contains a pair  $(x, v_x)$  per process  $p_x$ , and  $p_a$  can accordingly compute  $f(val\_set)$  (where  $f()$  is the function whose result is the output of the computation).

Let us notice that, when the distinguished process  $p_a$  discovers that the algorithm has terminated, all the messages sent by the algorithm have been received and processed.

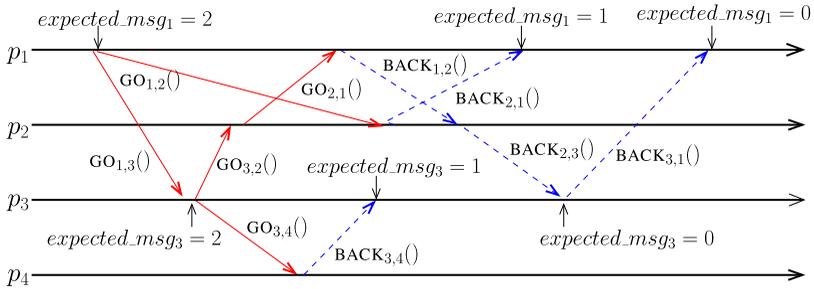
**Cost** Let us observe that a message BACK() is eventually sent as a response to each message GO(). Moreover, except on the channels of the spanning tree that is built, two messages GO() can be sent (one in each direction).

Let  $e$  be the number of channels of the underlying communication graph. It follows that the algorithm gives rise to  $2(n - 1)$  messages which travel on the channels of tree and  $4(e - (n - 1))$  messages which travel on the other channels, i.e.,  $2(2e - n + 1)$  messages. Then, once the tree is built, a broadcast/convergecast costs only  $2(n - 1)$  messages.

Assuming all messages take one time unit and local computations have zero duration, it is easy to see that the time complexity is  $2D$  where  $D$  is the diameter of the communication graph. Once the tree is built, the time complexity of a broad-



**Fig. 1.8** *Left*: Underlying communication graph; *Right*: Spanning tree



**Fig. 1.9** An execution of the algorithm constructing a spanning tree

cast/convergecast is  $2D_a$ , where  $D_a$  is the longest distance from  $p_a$  to any other process.

**An Example** An execution of the algorithm described in Fig. 1.7 for the communication graph depicted in the left part of Fig. 1.8 is described in Fig. 1.9.

Figure 1.9 is a space-time diagram. The execution of a process  $p_i$ ,  $1 \leq i \leq 4$ , is represented by an axis oriented from left to right. An arrow from one axis to another represents a message transfer. In this picture, an arrow labeled  $GO_{x,y}()$  represents a message  $GO()$  sent by  $p_x$  to  $p_y$ . Similarly, an arrow labeled  $BACK_{x,y}()$  represents a message  $BACK()$  sent by  $p_x$  to  $p_y$ .

The process  $p_1$  is the distinguished process that receives the external message  $START()$  and consequently will be the root of the tree. It sends a message  $GO()$  to its neighbors  $p_2$  and  $p_3$ . When  $p_3$  receives this message, it defines its parent as being  $p_1$  and forwards message  $GO()$  to its two other neighbors  $p_2$  and  $p_4$ .

Since the first message  $GO()$  received by  $p_2$  is the one sent by  $p_3$ ,  $p_2$  defines its parent as being  $p_3$  and forwards the message  $GO()$  to its other neighbor, namely  $p_1$ . When  $p_1$  receives a message  $GO()$  from  $p_2$ , it sends back a message  $BACK(\emptyset)$  to  $p_2$ . In contrast, when  $p_4$  receives the message  $GO()$  from  $p_3$ , it sends by return to  $p_3$  a message  $BACK()$  carrying the pair  $(4, v_4)$ . Moreover, when  $p_2$  has received a message  $BACK()$  from  $p_1$ , it sends to its parent  $p_3$  a message  $BACK()$  carrying the pair  $(2, v_2)$ .

Finally, when  $p_3$  receives the messages  $BACK()$  from  $p_2$  and  $p_4$ , it discovers that these processes are its children and sends a message  $BACK()$  carrying the set  $\{(2, v_2), (3, v_3), (4, v_4)\}$  to its parent  $p_1$ . When  $p_1$  receives this message, it discov-

ers that  $p_2$  is its only child. It can then compute  $f()$  on the vector  $[v_1, v_2, v_3, v_4]$ . The tree that has been built is represented at the right of Fig. 1.8.

**On the Parenthesized Structure of the Execution** It is important to notice that the spanning tree that has been built depends on the speed of the messages  $GO()$ . Another execution of the same algorithm on the same network with the same distinguished process could produce a different tree rooted at  $p_1$ .

It is also interesting to observe that each message  $GO()$  can be seen as an opening bracket that can be associated with a message  $BACK()$ , which is the corresponding closing bracket. This appears on the figure as follows:  $GO_{x,y}()$  is an opening bracket whose associated closing bracket is  $BACK_{y,x}()$ .

**The Case of Non-FIFO Channels** Assuming non-FIFO channels and taking into account Fig. 1.9, let us consider that the message  $GO_{1,2}()$  arrives at  $p_2$  after the message  $BACK_{1,2}()$ . It is easy to see that the algorithm remains correct (i.e., a spanning tree is built).

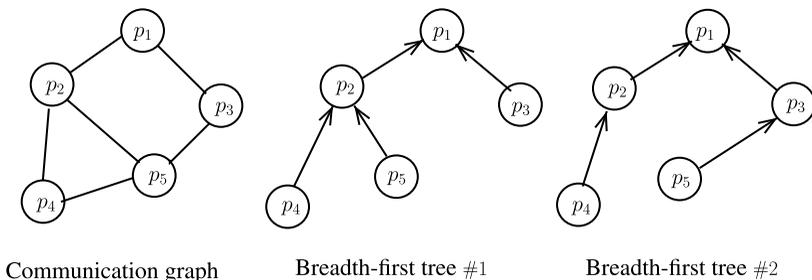
The only thing that changes is the meaning associated with line 16. When a process sends a message  $BACK()$  to its parent, it can no longer claim that its local computation is terminated. A process needs now to have received a message on each of its incident channels before claiming local termination.

**A Spanning Tree per Process** The algorithm of Fig. 1.7 can be easily generalized to build  $n$  trees, each one associated with a distinct process which is its distinguished process. Then, when any process  $p_i$  wants to execute an efficient broadcast/convergecast, it has to use its associated spanning tree.

To build a spanning tree per process, the local variables  $parent_i$ ,  $children_i$ , and  $expected\_msg_i$  of each process  $p_i$  have to be replaced by the arrays  $parent_i[1..n]$ ,  $children_i[1..n]$  and  $expected\_msg_i[1..n]$  and all messages have to carry the identity of the corresponding distinguished process. More precisely, when a process  $p_k$  receives a message  $START()$ , it uses its local variables  $parent_k[k]$ ,  $children_k[k]$ , and  $expected\_msg_k[k]$ . The corresponding messages will carry the identity  $k$ ,  $GO(k, -)$  and  $BACK(k, -)$ , and, when a process  $p_i$  receives such messages, it will use its local variables  $parent_i[k]$ ,  $children_i[k]$  and  $expected\_msg_i[k]$ .

**Concurrent Initiators for a Single Spanning Tree** The algorithm of Fig. 1.7 can be easily modified to build a single spanning tree while allowing several processes to independently start the execution of the algorithm, each receiving initially a message  $START()$ . To that end, each process manages an additional local variable  $max\_id_i$  initialized to 0, which contains the highest identity of a process competing to be the root of the spanning tree.

- If a process  $p_i$  receives a message  $START()$  while  $max\_id_i \neq 0$ ,  $p_i$  discards this message (in that case, it already participates in the algorithm but does not compete to be the root). Otherwise,  $p_i$  starts executing the algorithm and all the corresponding messages  $GO()$  or  $BACK()$  carry its identity.



**Fig. 1.10** Two different spanning trees built from the same communication graph

- Then, when a process  $p_i$  receives a message  $GO(j, -)$ ,  $p_i$  discards the message if  $j < \max\_id_i$ . Otherwise,  $p_i$  considers  $p_j$  as the process with the highest identity which is competing to be the root. It sets consequently  $\max\_id_i$  to  $j$  and continues executing the algorithm by using messages  $GO()$  and  $BACK()$  carrying the identity  $j$ .

It is easy to see that this simple application of the forward/discard strategy ensures that a single spanning tree will be constructed, namely the one rooted at  $p_j$  where  $j$  is such that, at the end of the execution, we have  $\max\_id_1 = \dots = \max\_id_n = j$ .

### 1.3 Breadth-First Spanning Tree

Let us remember that the *distance* between a process  $p_i$  and a process  $p_j$  is the length of the shortest path connecting these two processes, where the length of a path is measured by the number of channels it is made up of (this distance is also called the *hop distance*).

The spanning tree built by the algorithm of Fig. 1.7 does not necessarily build a *breadth-first tree*, i.e., a tree where the children of the root are its neighbors, and more generally the processes at distance  $d$  of the root in the tree are the processes at distance  $d$  of the root process in the communication graph. As we have seen in the example described in Fig. 1.8, the tree that is built depends on the speed of messages during the execution of the algorithm, and consequently distinct trees can be built by different executions.

Breadth-first traversal does not imply that the tree that is built is independent of the execution. It only means that two processes at distance  $d$  of the root in the communication graph are at distance  $d$  in the tree. According to the structure of the graph, two processes at distance  $d$  of the root do not necessarily have the same parent in different executions. A simple example is given in Fig. 1.10 where process  $p_5$ , which is at distance 2 of the root  $p_1$ , has different parents in the breadth-first trees described on the right part of the figure.

This section presents two algorithms that build breadth-first trees. Both are iterative algorithms, but the first one has no centralized control, while the second one is based on a centralized control governed by the distinguished root process.

### 1.3.1 Breadth-First Spanning Tree Built Without Centralized Control

**Principle of the Algorithm** This algorithm, which is due to T.-Y. Cheung (1983), is based on parallel traversals of the communication graph. These traversals are concurrent and some of them can stop others. In addition to the local variables  $parent_i$ ,  $children_i$ , and  $expexted\_msg_i$ , each process  $p_i$  manages a local variable, denoted  $level_i$ , which represents its current approximation of its distance to the root. Moreover, each message  $GO()$  carries now the current level of the sending process.

Then, when a process  $p_i$  receives a message  $GO(d)$ , there are two cases according to the current state of  $p_i$  and the value of  $d$ .

- The message  $GO(d)$  is the first message  $GO()$  received by  $p_i$ . In that case,  $p_i$  initializes  $level_i$  to  $d + 1$  and forwards the message  $GO(d + 1)$  to its neighbors (except the sender of the message  $GO(d)$ ).
- The message  $GO(d)$  is not the first message  $GO()$  received by  $p_i$  and  $level_i > d + 1$ . In that case,  $p_i$  (a) updates its variable  $level_i$  to  $d + 1$ , (b) defines the sender of the message  $GO(d)$  just received as its new parent, and (c) forwards a message  $GO(d + 1)$  to each of its other neighbors  $p_k$  in order that they recompute their distances to the root.

As we can see, these simple principles consist of a chaotic distributed iterative computation. They are used to extend the basic parallel network traversal algorithm of Fig. 1.7 with a forward/discard strategy that allows processes to converge to their final position in the breadth-first spanning tree.

**Description of the Algorithm** The algorithm is described in Fig. 1.11. As just indicated, it uses the parallel network traversal described in Fig. 1.7 as a skeleton on which are grafted appropriate statements to obtain a breadth-first rooted spanning tree. These new statements implement the convergence of the local variables  $parent_i$ ,  $children_i$ , and  $level_i$  to their final values. More precisely, we have the following.

Initially, a single process  $p_i$  receives an external message  $START()$ . This process, which will be the root of the tree, sends a message  $GO(-1)$  to itself (line 1). When it receives the message,  $p_i$  sets  $parent_i = i$  (hence it is the root) and its distance to itself is set to  $level_i = 0$ .

As previously indicated, there are two cases when a process  $p_i$  receives a message  $GO(d)$ . Let us remember that  $d$  represents the current approximation of the distance of the sender of the message  $GO()$  to the root. If  $parent_i = \perp$ , this message is the first message  $GO()$  received by  $p_i$  (line 2). In that case,  $p_i$  enters the tree at level  $d + 1$  (line 3) and propagates the network traversal to its other neighbors by sending them the message  $GO(d + 1)$  in order that they enter the tree or improve their position in the tree under construction (line 5). If the sender of the message  $GO(d)$  is its only neighbor,  $p_i$  sends by return the message  $BACK(yes, d + 1)$  to inform it that it is one of its children at level  $d + 1$  (line 6).

```

when START() is received do    % only the distinguished process receives this message %
(1)  send GO(-1) to itself.

when GO( $d$ ) is received from  $p_j$  do
(2)  if ( $parent_i = \perp$ )
(3)    then  $parent_i \leftarrow j$ ;  $children_i \leftarrow \emptyset$ ;  $level_i \leftarrow d + 1$ ;
(4)     $expected\_msg_i \leftarrow |neighbors_i \setminus \{j\}|$ ;
(5)    if ( $expected\_msg_i = 0$ )
(6)      then send BACK(yes,  $d + 1$ ) to  $p_{parent_i}$ 
(7)      else for each  $k \in neighbors_i \setminus \{j\}$  do send GO( $d + 1$ ) to  $p_k$  end for
(8)    end if
(9)  else if ( $level_i > d + 1$ )
(10)    then  $parent_i \leftarrow j$ ;  $children_i \leftarrow \emptyset$ ;  $level_i \leftarrow d + 1$ ;
(11)     $expected\_msg_i \leftarrow |neighbors_i \setminus \{j\}|$ ;
(12)    if ( $expected\_msg_i = 0$ )
(13)      then send BACK(yes,  $level_i$ ) to  $p_{parent_i}$ 
(14)      else for each  $k \in neighbors_i \setminus \{j\}$  do send GO( $d + 1$ ) to  $p_k$  end for
(15)    end if
(16)  else send BACK(no,  $d + 1$ ) to  $p_j$ 
(17)  end if
(18) end if.

when BACK( $resp, d$ ) is received from  $p_j$  do
(19) if ( $d = level_i + 1$ )
(20)  then if ( $resp = yes$ ) then  $children_i \leftarrow children_i \cup \{j\}$  end if;
(21)   $expected\_msg_i \leftarrow expected\_msg_i - 1$ ;
(22)  if ( $expected\_msg_i = 0$ )
(23)    then if ( $parent_i \neq i$ ) then send BACK(yes,  $level_i$ ) to  $p_{parent_i}$ 
(24)    else  $p_i$  learns that the breadth-first tree is built
(25)    end if
(26)  end if
(27) end if.

```

**Fig. 1.11** Construction of a breadth-first spanning tree without centralized control (code for  $p_i$ )

If the message GO( $d$ ) is not the first message GO() received by  $p_i$ , there are two cases. Let  $p_j$  be the sender of the message GO( $d$ ).

- If  $level_i \leq d + 1$ ,  $p_i$  cannot improve its position in the tree. It then sends by return the message BACK(no,  $d + 1$ ) to inform the sender of the message GO() that it cannot be its child at distance  $d + 1$  of the tree (line 16). Hence,  $p_i$  stops the network traversal associated with the message GO( $d$ ) it has received.
- If  $level_i > d + 1$ ,  $p_i$  has to improve its position in the tree under construction. To that end, it propagates the network traversal associated with the message GO( $d$ ) it has received in order to allow its other neighbors to improve their positions in the tree. Hence, it executes the same statements as those executed when it received its first message GO( $d$ ) (lines 10–15 are exactly the same as lines 3–8).

When a process  $p_i$  receives a message BACK( $resp, d$ ), it considers it only if  $level_i = d - 1$  (line 19). This is because this message is meaningful only if its sender  $p_j$  sent it when its level was  $level_j = d = level_i + 1$ . In the other cases, the message

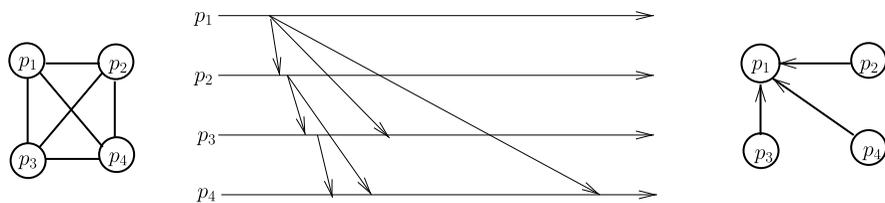


Fig. 1.12 An execution of the algorithm of Fig. 1.11

BACK() is discarded. If the message is meaningful and  $resp = yes$ ,  $p_i$  adds  $p_j$  to its children (and those are at level  $level_i + 1$ , line 20).

Finally, as in a simple parallel network traversal, if it has received a message  $BACK(-, level_i + 1)$  from each of its other neighbors,  $p_i$  sends a message  $BACK(yes, level_i)$  to its current parent. If  $p_i$  is the root, it learns that the breadth-first tree is built. Let us observe that, if  $p_i$  is not the root, it is possible that it later receives other messages  $GO()$  that will improve the current value of  $level_i$ .

**Termination** It follows from line 23 that the root learns the termination of the algorithm.

On the other hand, the local variable  $level_i$  of a process  $p_i$ , which is not the root, can be updated each time it receives a message  $GO()$ . Unfortunately, the number of such messages received by  $p_i$  is not limited to the number of its neighbors but depends on (a) the number of neighbors of its neighbors, etc. (i.e., the structure of the communication graph), and (b) the speed of messages (i.e., asynchrony). As its knowledge of the communication graph is local (it is restricted to its neighbors), a process cannot define a local predicate indicating that its local variables have converged to their final values. But, as the root can discover when the construction of the tree has terminated, it can send a message (which will be propagated along the tree) to inform the other processes that their local variables  $parent_i$ ,  $children_i$ , and  $level_i$  have their final values.

**A Simple Example** Let us consider the communication graph described on the left of Fig. 1.12. As the graph is a clique, a single breadth-first tree can be obtained, namely the star graph centered at the distinguished process (here  $p_1$ ). This star is depicted on the right of the figure.

A worst case execution of the algorithm of Fig. 1.11 is depicted in the middle of Fig. 1.12 (worst case with respect to the number of messages). Only the message  $GO()$  from a process  $p_i$  to a process  $p_j$  with  $i < j$  is represented in the space-time diagram. The worst case occurs when the message  $GO()$  sent by  $p_1$  to  $p_3$  arrives after the message  $GO()$  sent by  $p_2$ , and both the message  $GO()$  sent by  $p_1$  to  $p_4$  and the message  $GO()$  sent by  $p_2$  to  $p_4$  arrive after the message  $GO()$  sent by  $p_3$  to  $p_4$ . It follows that, with this pattern of message arrivals,  $p_2$  discovers it is at distance 1 when it receives its first message  $GO()$ ,  $p_3$  discovers that it is at distance 1 when it receives the message  $GO()$  sent by  $p_1$ , and similarly  $p_4$  discovers it when it receives the message  $GO()$  sent by  $p_1$ .

**Cost** There are two type of messages, and each message carries an integer whose value is bounded by the diameter  $D$  of the communication graph. Moreover, a message `BACK()` carries an additional Boolean value. It follows that the size of each message is upper bounded by  $2 + \log_2 D$  bits. It is easy to see that the time complexity is  $O(D)$ , i.e.,  $O(n)$  in the worst case.

As far as the message complexity is concerned, the worst case is a fully connected communication graph (i.e., any pair of processes is connected by a channel) and a process at distance  $d$  of the root updates *level* <sub>$i$</sub>   $d$  times (as in Fig. 1.12). This means that among the  $(n - 1)$  processes which are not the root, one updates its level once, another one updates it twice, etc., and one updates it  $(n - 1)$  times. Moreover, each time a process updates its level, a process forwards the messages `GO()` to  $(n - 2)$  of its neighbors (all processes but itself and the sender of the `GO()` that entailed the update of its own level). The root sends  $(n - 1)$  messages `GO()`. Hence the total number of messages `GO()` is

$$(n - 1) + (n - 2) \sum_{i=1}^{n-1} i = \frac{(n - 1)(n^2 - 2n + 2)}{2}.$$

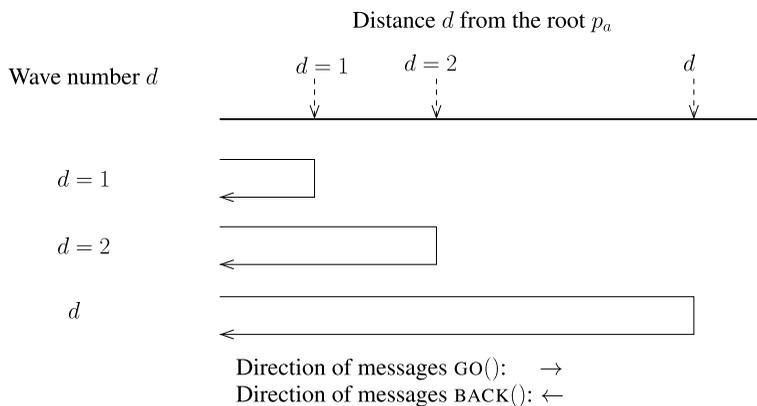
As at most one message `BACK()` is associated with each message `GO()`, it follows that, in a fully connected network, the message complexity is upper bounded by  $O(n^3)$ .

### 1.3.2 Breadth-First Spanning Tree Built with Centralized Control

This section presents a second distributed algorithm that builds a breadth-first spanning tree. Differently from the previous one, this algorithm—which is due to Y. Zhu and T.-Y. Cheung (1987)—is based on a centralized control that allows each process to locally learn when its participation to the algorithm has terminated. Moreover, both its time and message complexities are  $O(n^2)$ . This shows an interesting trade-off with the previous algorithm whose time complexity is  $O(n)$  while its message complexity is  $O(n^3)$ .

**Underlying Principle** This principle consists in a distributed iteration whose progress is handled by the distinguished process  $p_a$ . Let us call *wave* each new iteration launched by  $p_a$ . The first wave (first iteration) attains only the neighbors of the root  $p_a$ , which discover they are at distance 1 and enter the tree. The second wave directs the processes at distance 2 of the root to enter the tree. More generally, the wave number  $d$  directs the processes at distance  $d$  of the root to enter the tree. After the wave number  $d$  has attained the processes at distance  $d$ , it returns to the root in order to launch the next wave (see Fig. 1.13).

It follows that, when  $p_a$  launches the wave number  $d + 1$ , the processes at distance less than  $d + 1$  know their position in the tree, and the processes at distance  $d + 1$  will discover they are at distance  $d + 1$  by receiving a message from a process



**Fig. 1.13** Successive waves launched by the root process  $p_a$

at distance  $d$ . Hence, the messages implementing the wave number  $d + 1$  can use only the channels of the breadth-first spanning tree of depth  $d$  that has been built by the previous waves. This reduces consequently the number of messages needed to implement a wave.

From an implementation point of view, for any  $d$ , the wave number  $d$  going from the root up to processes at distance  $d$  is implemented with messages  $GO()$ , while its return to the root is implemented with messages  $BACK()$ , as depicted in Fig. 1.13.

**Algorithm: Local Variables** In addition to the constant set  $neighbors_i$  and its local variables  $parent_i$  (initialized to  $\perp$ ) and  $children_i$ , each process  $p_i$  manages the following local variables in order to implement the underlying principle previously described.

- $distance_i$  is a write-once local variable that keeps the distance of  $p_i$  to the root.
- $to\_send_i$  is a set that, once  $p_i$  has been inserted into the spanning tree, contains its neighbors to which it has to propagate the waves it receives from the root. If  $p_i$  is at distance  $d$ , these wave propagations will concern waves whose number is greater than  $d$ .
- $waiting\_from_i$  is a set used by  $p_i$  to manage the return of the current wave to its parent in the tree. (Its role is similar to that of the local variable  $expected\_msg_i$  used in the previous algorithms.)

**Algorithm: Launching by  $p_a$**  The algorithm assumes that the system is made up of at least two processes.

The initial code executed by the distinguished process is described in Fig. 1.14. This process defines itself as the root (as in previous algorithms, it will be the only process  $p_i$  such that  $parent_i = i$ ), initializes its other local variables and sends a message  $GO(0)$  to each of its neighbors. More generally, the value  $d$  in a message  $GO(d)$  sent by a process  $p_i$  means that  $d$  is the distance of  $p_i$  to the root of the tree.

The behavior of a process  $p_i$  when it receives a message  $GO()$  or  $BACK()$  is described in Fig. 1.15 (the root process receives only messages  $BACK()$ ).

```

when START() is received do % only the distinguished process receives this message %
   $parent_i \leftarrow i$ ;  $children_i \leftarrow \emptyset$ ;  $distance_i \leftarrow 0$ ;  $to\_send_i \leftarrow neighbors_i$ ;
  for each  $k \in to\_send_i$  do send GO(0) to  $p_k$  end for;
   $waiting\_from_i \leftarrow neighbors_i$ .

```

**Fig. 1.14** Construction of a breadth-first spanning tree with centralized control (starting code)

```

when GO( $d$ ) is received from  $p_j$  do
  (1) if ( $parent_i = \perp$ )
  (2)   then  $parent_i \leftarrow j$ ;  $children_i \leftarrow \emptyset$ ;  $distance_i \leftarrow d + 1$ ;  $to\_send_i \leftarrow neighbors_i \setminus \{j\}$ ;
  (3)     if ( $to\_send_i = \emptyset$ ) then send BACK(stop) to  $p_j$ 
  (4)       else send BACK(continue) to  $p_j$  end if
  (5)   else if ( $parent_i = j$ )
  (6)     then for each  $k \in to\_send_i$  do send GO( $distance_i$ ) to  $p_k$  end for;
  (7)        $waiting\_from_i \leftarrow to\_send_i$ 
  (8)     else send BACK(no) to  $p_j$ 
  (9)   end if
  (10) end if.

when BACK( $resp$ ) is received from  $p_j$  do
  (11)  $waiting\_from_i \leftarrow waiting\_from_i \setminus \{j\}$ ;
  (12) if  $resp \in \{continue, stop\}$  then  $children_i \leftarrow children_i \cup \{j\}$  end if;
  (13) if  $resp \in \{stop, no\}$  then  $to\_send_i \leftarrow to\_send_i \setminus \{j\}$  end if;
  (14) if ( $to\_send_i = \emptyset$ ) % we have then  $waiting\_from_i = \emptyset$  %
  (15)   then if ( $parent_i = i$ ) then the root learns that the tree is built
  (16)     else send BACK(stop) to  $p_{parent_i}$ 
  (17)   end if
  (18) else if ( $waiting\_from_i = \emptyset$ )
  (19)   then if ( $parent_i = i$ )
  (20)     then for each  $k \in to\_send_i$  do send GO( $distance_i$ ) to  $p_k$  end for;
  (21)      $waiting\_from_i \leftarrow to\_send_i$ 
  (22)     else send BACK(continue) to  $p_{parent_i}$ 
  (23)   end if
  (24) end if
  (25) end if.

```

**Fig. 1.15** Construction of a breadth-first spanning tree with centralized control (code for a process  $p_i$ )

**Reception of a Message GO()** When a process  $p_i$  receives for the first time a message GO( $d$ ), it discovers that it is at distance  $d + 1$  of the root (line 1). Let  $p_j$  be the sender of this message GO( $d$ ). The receiving process  $p_i$  consequently initializes its local variables  $parent_i$  to  $j$  and  $distance_i$  to  $d + 1$  (line 2). It also initializes its set  $children_i$  to  $\emptyset$  and  $to\_send_i$  to its set of neighbors except  $p_j$ .

Then,  $p_i$  returns the current wave to its parent  $p_j$  by returning to it a message BACK() (line 3). If  $to\_send_i$  is empty,  $p_j$  is the only neighbor of  $p_i$ , and consequently  $p_i$  returns the message BACK(stop) to indicate to  $p_j$  that (a) it is one of its children and (b) no more processes can be added to the tree as far it is concerned. Hence, if there are new waves, its parent  $p_j$  will not have to send it messages GO( $d$ )

in order to expand the tree. If  $to\_send_i$  is not empty,  $p_i$  is one of the children of  $p_j$  and the tree can possibly be expanded from  $p_i$ . In this case,  $p_i$  sends the message `BACK(continue)` to its parent  $p_j$  to inform it that it is one of its children and (b) possibly, during the next wave, new processes can be added to the tree with  $p_i$  as parent. These two cases are expressed at line 3.

If  $p_i$  already has a parent ( $parent_i \neq \perp$ ), i.e., it is already in the tree when it receives a message `GO(d)`, its behavior depends then on the sender  $p_j$  of the message `GO(d)` line 5. If  $p_j$  is its parent,  $p_i$  forwards the wave by sending the message `GO(d + 1)` to its other neighbors in the set  $to\_send_i$  (line 6) and resets accordingly  $waiting\_from_i$  to  $to\_send_i$  (line 7). If  $p_j$  is not its parent (line 8),  $p_i$  sends back the message `BACK(no)` to the process  $p_j$  to inform it that (a) it is not one of its children and consequently (b)  $p_j$  no longer has to forward waves to it.

**Reception of a Message `BACK()`** When a process  $p_i$  receives a message `BACK()` it has already determined its position in the breadth-first spanning tree. This message `BACK()` sent by a neighbor  $p_j$  is associated with a message `GO()` sent by  $p_i$  to  $p_j$ . It carries a value  $resp \in \{\text{stop}, \text{continue}, \text{no}\}$ .

Hence, when  $p_i$  receives `BACK(resp)` from  $p_j$ , it first suppresses  $p_j$  from the process from which it waits for messages (line 11). Then, if  $resp \in \{\text{stop}, \text{continue}\}$ ,  $p_j$  is one of its children (line 12) and if  $resp \in \{\text{stop}, \text{no}\}$ ,  $p_i$  discovers that it has no longer to send messages to  $p_j$  (line 13). The behavior of  $p_i$  depends then on the set  $to\_send_i$ .

- If  $to\_send_i$  is empty (line 14),  $p_i$  knows that its participation to the algorithm is terminated. (Let us notice that, due to lines 7, 11 and 13, we have then  $waiting\_from_i = \emptyset$ .) If  $p_i$  is the root, it also knows that the algorithm has terminated (line 15). If it is not the root, it sends the message `BACK(stop)` to its parent to inform it that (a) it has locally terminated, and (b) the tree can no longer be extended from it (line 16).
- If  $to\_send_i$  is not empty, it is possible that the tree can be expanded from  $p_i$ . In this case, if  $waiting\_from_i = \emptyset$ ,  $p_i$  returns the wave to its parent by sending it the message `BACK(continue)` (line 22). If it is the root,  $p_i$  starts a new wave by sending the message `GO(0)` to its neighbors from which the tree can possibly be expanded (line 20) and resets appropriately its local set  $to\_send_i$  (line 21).

**On Distributed Synchronization** As we can see, this algorithm exhibits two types of synchronization. The first, which is global, is realized by the root process that controls the sequence of waves. This appears at lines 20–21. The second is local at each process. It occurs at line 3, line 16, or line 22 when a process  $p_i$  sends back a message `BACK()` to its parent.

**Local Versus Global Termination** Differently from the algorithm described in Fig. 1.11 in which no process—except the root—knows when its participation to the algorithm has terminated, the previous algorithm allows each process to know that it has terminated.

For a process which is not the root, this occurs when it sends a message `BACK(stop)` to its parent at line 3 (if the parent of  $p_i$  is its only neighbor) or

at line 16 (if  $p_i$  has several neighbors). Of course, the fact that a process has locally terminated does not mean that the algorithm has terminated. Only the root can learn it. This occurs at line 15 when the root  $p_i$  receives a message `BACK(stop)` entailing the last update of  $to\_send_i$  which becomes empty.

**Cost** Let us first consider the number of messages. As in previous algorithms, each message `BACK()` is associated with a message `GO()`. Hence, we have only to determine the number of messages `GO()`.

Let  $e$  be the number of channels of the communication graph. At most two messages `GO()` are exchanged on each channel that will not belong to the tree. It follows that at most  $2e - (n - 1)$  messages `GO()` are exchanged on these channels. Since the tree will involve  $(n - 1)$  channels and, in the worst case, there are at most  $n$  waves, it follows that at most  $O(n^2)$  messages `GO()` travel on the channels of the tree. As  $e$  is upper bounded by  $O(n^2)$ , it follows that the total number of messages `GO()` and `BACK()` is upper bounded by  $O(n^2)$ .

As far as the time complexity is concerned we have the following. The first wave takes two time units (messages `GO()` from the root to its neighbors followed by messages `BACK()` from these processes to the root). The second wave takes 4 time units (two sequential sendings of messages `GO()` from the root to the processes at distance 2 from it and two sequential sendings of messages `BACK()` from these processes to the root), etc. As there are at most  $D$  waves (where  $D$  is the diameter of the communication graph), the time complexity is upper bounded by  $2(1 + 2 + \dots + D)$ , i.e.,  $O(D^2)$ .

It follows that both the message and the time complexities are bounded by  $O(n^2)$ . As already noticed, this shows an interesting tradeoff when comparing this algorithm with the algorithm without centralized synchronization described in Fig. 1.11 whose message and time complexities are  $O(n^3)$  and  $O(n)$ , respectively. The added synchronization allows for a reduction of the number of messages at the price of an increase in the time complexity.

## 1.4 Depth-First Traversal

Starting from a distinguished process  $p_a$ , a distributed depth-first network traversal visits one process at a time. This section presents first a simple distributed depth-first traversal algorithm which benefits from the traversal to construct a rooted tree. Two improvements of this algorithm are then presented. Finally, one of them is enriched to obtain a distributed algorithm that builds a logical ring on top of an arbitrary connected communication graph.

### 1.4.1 A Simple Algorithm

**Description of the Algorithm** The basic depth-first distributed algorithm is described in Fig. 1.16. The algorithm starts when a distinguished process received the

```

when START() is received do  % only  $p_a$  receives this message %
(1)  $parent_i \leftarrow i$ ;  $children_i \leftarrow \emptyset$ ;  $visited_i \leftarrow \emptyset$ ;
(2) let  $k \in neighbors_i$ ; send GO() to  $p_k$ .

when GO() is received from  $p_j$  do
(3) if ( $parent_i = \perp$ )
(4)   then  $parent_i \leftarrow j$ ;  $children_i \leftarrow \emptyset$ ;  $visited_i \leftarrow \{j\}$ ;
(5)     if ( $visited_i = neighbors_i$ )
(6)       then send BACK(yes) to  $p_j$ 
(7)       else let  $k \in neighbors_i \setminus visited_i$ ; send GO() to  $p_k$ 
(8)     end if
(9)   else send BACK(no) to  $p_j$ 
(10)  end if.

when BACK( $resp$ ) is received from  $p_j$  do
(11) if ( $resp = yes$ ) then  $children_i \leftarrow children_i \cup \{j\}$  end if;
(12)  $visited_i \leftarrow visited_i \cup \{j\}$ ;
(13) if ( $visited_i = neighbors_i$ )
(14)   then if ( $parent_i = i$ )
(15)     then the traversal is terminated  % global termination %
(16)     else send BACK(yes) to  $p_{parent_i}$  % local termination %
(17)   end if
(18)   else let  $k \in neighbors_i \setminus visited_i$ ; send GO() to  $p_k$ 
(19)   end if.

```

**Fig. 1.16** Depth-first traversal of a communication graph (code for  $p_i$ )

external message START(). The distinguished process  $p_a$  sends first a message GO () to one of its neighbors (line 2).

Then, when a process  $p_i$  receives a message GO (), it defines the message sender as its parent in the depth-first tree (lines 3–4). The local variable  $visited_i$  is a set containing the identities of its neighbors which have been already visited by the depth-first traversal (implemented by the progress of the message GO()). If  $p_j$  is its only neighbor,  $p_i$  sends back to  $p_j$  the message BACK(yes) to inform it that (a) it is one of its children and (b) it has to continue the depth-first traversal (lines 5–6). Otherwise ( $neighbors_i \neq visited_i$ ),  $p_i$  propagates the depth-first traversal to one of its neighbors that, from its point of view, has not yet been visited (line 7). Finally, if  $p_i$  has already been visited by the depth-first traversal ( $parent_i \neq \perp$ ), it sends back to  $p_j$  a message BACK(no) to inform it that it is not one of its children (line 9).

When a process  $p_i$  receives a message BACK( $resp$ ), it first adds its sender  $p_j$  to its set of children if  $resp = yes$  (line 11). Moreover, it also adds  $p_j$  to the set of its neighbors which have been visited by the depth-first traversal (line 12). Then, its behavior is similar to that of lines 5–8. If, from its point of view, not all of its neighbors have been visited, it sends a message GO() to one of them (line 18). If all of its neighbors have been visited (line 13), it claims the termination if it is the root (line 14). If it is not the root, it sends to its parent the message BACK(yes) to inform the parent that it is one of its children and it has to forward the depth-first traversal.

**On the Tree That Is Built** It is easy to see that, given a predefined root process, the depth-first spanning tree that is built does not depend on the speed of messages but depends on the way each process  $p_i$  selects its neighbor  $p_k$  to which it propagates the depth-first traversal (line 7 and line 18).

**Cost** As in previous algorithms, a message `BACK()` is associated with each message `GO()`. Moreover, at most one message `GO()` is sent in each direction on every channel. It follows that the message complexity is  $O(e)$ , where  $e$  is the number of channels of the communication graph. Hence, it is upper bounded by  $O(n^2)$ .

There are two types of messages, and message `BACK()` carries a binary value. Hence, the size of a message is one or two bits.

Finally, at most one message `GO()` or `BACK()` is traveling on a channel at any given time. It follows that the time complexity is the same as the message complexity, i.e.,  $O(e)$ .

**An Easy Improvement of the Basic Algorithm** A simple way to improve the time complexity of the previous distributed depth-first traversal algorithm consists in adding a local information exchange phase when the depth-first traversal visits a process for the first time. This additional communication phase, which is local (it involves only the process receiving a message `GO()` and its neighbors), allows for an  $O(n)$  time complexity (let us remember that  $n$  is the number of processes).

This additional exchange phase is realized as follows. When a process  $p_i$  receives a message `GO()` for the first (and, as we will see, the only) time, the following statements are executed before it forwards the depth-first traversal to one of its neighbors (line 7):

- $p_i$  sends an additional control message `VISITED()` to each of its neighbors  $p_j$ , and waits until it has received an answer message `KNOWN()` from each of them.
- When a process  $p_j$  receives a message `VISITED()` from one of its neighbors  $p_i$ , it adds  $i$  to its local set  $visited_j$  and sends by return the message `KNOWN()` to  $p_i$ .

It is easy to see that when a message `GO()` has been received by a process  $p_i$ , all its neighbors are informed that it has been visited by the depth-first traversal, and consequently none of them will forward the depth-first traversal to it. Moreover, thanks to this modification, no message `BACK()` carries the value `no` (line 9 disappears). Consequently, as all the messages `BACK()` carry the value `yes`, this value can be left implicit and line 11 can be shortened to “ $children_i \leftarrow children_i \cup \{j\}$ ”.

It follows that the modified algorithm sends  $(n - 1)$  messages `GO()`, the same number of messages `BACK()`,  $2e - (n - 1)$  messages `VISITED()`, and the same number of messages `KNOWN()`. Hence, the message complexity is  $O(e)$ .

As far as the time complexity is concerned, we have the following. There are  $(n - 1)$  messages `GO()`, the same number of messages `BACK()`, and no two of these messages are traveling concurrently. When it receives a message `GO()`, each process sends a message `VISITED()` to its neighbors (except its parent), and these messages travel concurrently. The same occurs for the answer messages `KNOWN()`. Let  $n_1$  be the number of processes that have a single neighbor in the communication graph. The time complexity is consequently  $2(n - 1) + 2n - 2n_1$ , i.e.,  $O(n)$ .

```

when START() is received do    % only  $p_a$  receives this message %
(1)   $parent_i \leftarrow i$ ;
(2)  let  $k \in neighbors_i$ ;
(3)  send GO( $\{i\}$ ) to  $p_k$ ;  $children_i \leftarrow \{k\}$ .

when GO( $visited$ ) is received from  $p_j$  do
(4)   $parent_i \leftarrow j$ ;
(5)  if ( $neighbors_i \subseteq visited$ )
(6)    then send BACK( $visited \cup \{i\}$ ) to  $p_j$ ;  $children_i \leftarrow \emptyset$ ;
(7)    else let  $k \in neighbors_i \setminus visited$ ;
(8)      send GO( $visited \cup \{i\}$ ) to  $p_k$ ;  $children_i \leftarrow \{k\}$ 
(9)  end if.

when BACK( $visited$ ) is received from  $p_j$  do
(10) if ( $neighbors_i \subseteq visited$ )
(11)  then if ( $parent_i = i$ )
(12)    then the traversal is terminated    % global termination %
(13)    else send BACK( $visited$ ) to  $p_{parent_i}$  % local termination %
(14)    end if
(15) else let  $k \in neighbors_i \setminus visited$ ;
(16)  send GO() to  $p_k$ ;  $children_i \leftarrow children_i \cup \{k\}$ 
(17) end if.

```

Fig. 1.17 Time and message optimal depth-first traversal (code for  $p_i$ )

## 1.4.2 Application: Construction of a Logical Ring

**The Notion of a Local Algorithm** Both the previous algorithm (Fig. 1.16) and its improvement are *local* in the sense that (a) each process has initially to know only its identity, that of its neighbors, and the fact no two processes have the same identity, and (b) the size of the information exchanged between any two neighbors is bounded.

**Another Improvement of the Basic Depth-First Traversal Algorithm** This part presents a depth-first traversal algorithm that is not local but whose message complexity and time complexity are  $O(n)$ . The idea is to replace the local synchronization used by the improved version of the previous algorithm (implemented with the messages VISITED() and KNOWN()) by a global control information, namely, the set (denoted *visited*) of the processes which have been already visited by the network traversal (i.e., by messages GO()). To that end, each message GO() and each message BACK() carry the current value of the set *visited*.

The corresponding depth-first traversal algorithm is described in Fig. 1.17. When the distinguished process  $p_a$  receives the external message START(), it defines itself as the root ( $parent_i = i$ , line 1), and launches the depth-traversal by sending a message GO( $\{i\}$ ) to one of its neighbors that it includes in its set of children (lines 2–3).

Then, when a process  $p_i$  receives a message GO(*visited*) from a neighbor process  $p_j$ , it defines  $p_j$  as its parent (line 4). If all of its neighbors have been visited,  $p_i$  sends back the message GO( $visited \cup \{i\}$ ) to its parent (line 6). Otherwise, it

propagates the depth-first traversal to one of its neighbors  $p_k$  that has not yet been visited and initializes  $children_i$  to  $\{k\}$  (lines 7–8).

Finally, when a process  $p_i$  receives a message  $BACK(visited)$  such that all its neighbors have been visited (line 10), it claims termination if it is the root (line 12). If it is not the root, it forwards the message  $BACK(visited)$  to its parent (line 13). If some of its neighbors have not yet been visited,  $p_i$  selects one of them, propagates the network traversal by sending to it the message  $GO(visited)$  and adds it to its set of children (lines 15–16).

It is easy to see that this algorithm builds a depth-first spanning tree, and requires  $(n - 1)$  messages  $GO()$  and  $(n - 1)$  messages  $BACK()$ . As no two messages are concurrent, the time complexity is  $2(n - 1)$ . As already indicated, this algorithm is not local: the set  $visited$  carried by each message grows until it contains all the process identities. Hence, the size of a message includes one bit for the message type and up to  $n \log_2 n$  bits for its content.

**A Token Traveling Along a Logical Ring** A token is a message that navigates a network. A simple way to exploit a token consists in using a logical unidirectional ring. The token progresses then from a process to the next process on the ring. Assuming that no process keeps the token forever, it follows that no process that wants to use the token will miss it.

The problem that then has to be solved consists in building a logical unidirectional ring on top of a connected arbitrary network. Such a construction is presented below.

In order that the ring can be exploited by the token, the algorithm building the ring computes the value of two local variables at each process  $p_i$ :

- The local variable  $succ_i$  denotes the identity of the successor of  $p_i$  on the unidirectional ring.
- The local variable  $routing_i[j]$ , where  $p_j$  is a neighbor of  $p_i$ , contains the identity of its neighbor to which  $p_i$  will have to forward the token when it receives it from  $p_j$ .

The set of local variables  $routing_i[j]$ ,  $1 \leq i \leq n$ , defines the appropriate routing (at the level of the underlying communication graph) which ensures the correct move of the token on the logical ring from any process to its logical successor.

Once these two local variables have been computed at each process  $p_i$ , the navigation of the token is easily ensured by the statements described in Fig. 1.18. When it leaves a process, the token carries the identity  $dest$  of its destination, i.e., the next process on the unidirectional ring. When it receives the token and  $dest = i$ , a process  $p_i$  uses it. Before releasing it, the process sets the token field  $dest$  to  $succ_i$ , the next process on the ring (i.e., the next process allowed to use the token). Finally, (whether it was its previous destination or not)  $p_i$  sends the token to its neighbor  $p_k$  such that  $k = routing_i[j]$  (where  $p_j$  is the process from which it received the token in the communication graph). Hence, when considering the underlying communication graph, the token progresses from  $p_i$  to  $p_k$  where  $k = routing_i[j]$ , then to  $p_\ell$  where  $\ell = routing_k[i]$ , then to  $p_m$  where  $m = routing_\ell[k]$ , etc., where the sequence

**Fig. 1.18** Management of the token at process  $p_i$

```

when TOKEN(dest) is received from  $p_j$  do
  if ( $dest = i$ ) then use the token;  $dest \leftarrow succ_i$  end if;
  let  $k = routing_i[j]$ ; send TOKEN(dest) to  $p_k$ .

```

of successive neighbor processes  $p_j, p_i, p_k, p_\ell, \dots, p_j$  constitutes the circuit implementing the logical unidirectional ring.

**A Distributed Algorithm Building a Logical Ring** The distributed depth-first traversal algorithm described in Fig. 1.17 constitutes the skeleton on which are grafted statements that build the logical ring, i.e., the statements that give their values to the local variables  $succ_i$  and  $routing_i[j]$  of each process  $p_i$ . The resulting algorithm, which is due to J.-M. H elary and M. Raynal (1988), is described in Fig. 1.19.

As we can see, this algorithm is nearly the same as that of Fig. 1.17. Its underlying principle follows: As a process receives a single message GO(), a process is added to the logical ring when it receives such a message. Moreover, in addition to the set of processes *visited*, each message is required to carry the identity (denoted *last*) of the last process that received a message GO().

In order to establish the sense of direction of the ring and compute the distributed routing tables, the algorithm uses the total order in which processes are added to the ring, i.e., the order in which the processes receive a message GO(). More precisely, the sense of direction of the ring is then defined as opposite to the order in which the processes are added to the ring during its construction. This is depicted in Fig. 1.20.

```

when START() is received do   % only  $p_a$  receives this message %
(1)   $parent_i \leftarrow i$ ;
(2)  let  $k \in neighbors_i$ ;
(3)  send GO( $\{i\}, i$ ) to  $p_k$ ;  $first_i \leftarrow k$ .

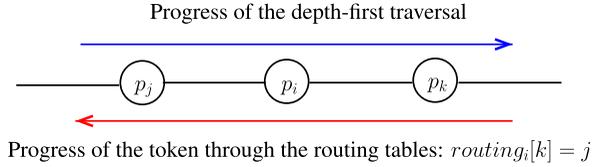
when GO(visited, last) is received from  $p_j$  do
(4)   $parent_i \leftarrow j$ ;  $succ_i \leftarrow last$ ;
(5)  if ( $neighbors_i \subseteq visited$ )
(6)    then send BACK( $visited \cup \{i\}, i$ ) to  $p_j$ ;  $routing_i[j] \leftarrow j$ 
(7)    else let  $k \in neighbors_i \setminus visited$ ;
(8)      send GO( $visited \cup \{i\}, i$ ) to  $p_k$ ;  $routing_i[k] \leftarrow j$ 
(9)    end if.

when BACK(visited, last) is received from  $p_j$  do
(10) if ( $neighbors_i \subseteq visited$ )
(11)   then if ( $parent_i = i$ )
(12)     then  $succ_i \leftarrow last$ ;  $routing_i[first_i] \leftarrow j$  % the ring is built %
(13)     else send BACK(visited, last) to  $p_{parent_i}$ ;  $routing_i[parent_i] \leftarrow j$ 
(14)   end if
(15) else let  $k \in neighbors_i \setminus visited$ ;
(16)   send GO(visited, last) to  $p_k$ ;  $routing_i[k] \leftarrow j$ 
(17) end if.

```

**Fig. 1.19** From a depth-first traversal to a ring (code for  $p_i$ )

**Fig. 1.20** Sense of direction of the ring and computation of routing tables



From an operational point of view, we have the following. When the distinguished process receives the message  $START()$  it defines itself as the starting process ( $parent_i = i$ ), selects one of its neighbors  $p_k$ , and sends to  $p_k$  the message  $GO(visited, i)$  where  $visited = \{i\}$  (lines 1–3). Moreover, the starting process records the identity  $k$  in order to be able to close the ring when it discovers that the depth-first traversal has terminated (line 12).

When a process  $p_i$  receives a message  $GO(visited, last)$  (let us remember that it receives exactly one message  $GO()$ ), it defines (a) its parent with respect to the depth-first traversal as the sender  $p_j$  of the message  $GO()$ , and (b) its successor on the ring as the last process (before it) that received a message  $GO()$ , i.e., the process  $p_{last}$  (line 4). Then, if all its neighbors have been visited by the depth-first traversal, it sends back to its parent  $p_j$  the message  $GO(visited \cup \{i\}, i)$  and defines the appropriate routing for the token, namely it sets  $routing_i[j] = j$  (lines 5–6). If there are neighbors of  $p_i$  that have not yet been visited,  $p_i$  selects one of them (say  $p_k$ ) and propagates the depth-first traversal by sending the message  $GO(visited \cup \{i\}, i)$  to  $p_k$ . As before, it also defines the appropriate routing for the token, namely it sets  $routing_i[k] = j$  (lines 7–8).

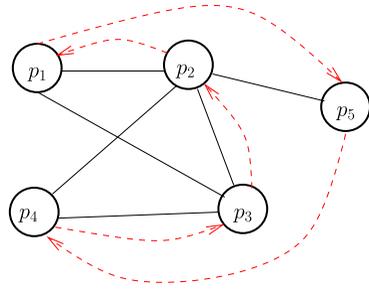
When a process  $p_i$  receives a message  $BACK(visited, last)$ , it does the same as previously if some of its neighbors have not yet been visited (lines 15–16 are similar to lines 7–8). If all its neighbors have been visited and  $p_i$  is the starting process, it closes the ring by assigning to  $routing_i[first_i]$  the identity  $j$  of the process that sent the message  $BACK(-, -)$  (lines 11–12). If  $p_i$  is not the starting process, it forwards the message  $BACK(visited, last)$  to its parent which will make the depth-first traversal progress. It also assigns the identity  $j$  to  $routing_i[parent_i]$  for the token to be correctly routed along the appropriate channel of the communication graph in order to attain its destination process on the logical ring.

**Cost** This ring construction requires always  $2(n - 1)$  messages:  $(n - 1)$  messages  $GO()$  and  $(n - 1)$  messages  $BACK()$ . Moreover the ring has  $n$  virtual channels  $vc_1, \dots, vc_n$  and the length  $\ell_x$  of  $vc_x$ ,  $1 \leq x \leq n$ , is such that  $1 \leq \ell_x \leq (n - 1)$  and  $\sum_{1 \leq x \leq n} \ell_x = 2(n - 1)$ . Hence, both the cost of building the ring and ensuring a full turn of the token on it are  $2(n - 1)$  messages sent one after the other.

**Remarks** The two logical neighbors on the ring of each process  $p_i$  depend on the way a process selects a non-visited neighbor when it has to propagate the depth-first traversal.

Allowing the messages  $GO()$  and  $BACK()$  to carry more information (on the structure of the network that has been visited—or has not been visited—by the depth-first

**Fig. 1.21** An example of a logical ring construction



traversal) allows the length of the ring at the communication graph level to be reduced to  $x$ , where  $x \in [n \dots 2(n - 1)]$ . This number  $x$  depends on the structure of the communication graph and the way neighbors are selected when a process propagates the network traversal.

**An Example** Let us consider the communication graph depicted Fig. 1.21 (without the dotted arrows). The dotted arrows represent the logical ring constructed by the execution described below.

In this example, when a process has to send a message  $GO()$  to one of its neighbors, it selects the neighbor with the smallest identity in the set  $neighbors_i \setminus visited$ .

1. The distinguished process sends the message  $GO(\{1\}, 1)$  to its neighbor  $p_2$  and saves the identity 2 into  $first_i$  to be able to close the ring at the end of the network traversal.
2. Then, when it receives this message,  $p_2$  defines  $succ_2 = 1$ , forwards the depth-first traversal by sending the message  $GO(\{1, 2\}, 2)$  to  $p_3$  and defines  $routing_2[3] = 1$ .
3. When  $p_3$  receives this message, it defines  $succ_3 = 2$ , forwards the depth-first traversal by sending the message  $GO(\{1, 2, 3\}, 3)$  to  $p_4$  and defines  $routing_3[4] = 2$ .
4. When  $p_4$  receives this message, it defines  $succ_4 = 3$ , and propagates the depth-first traversal by sending the message  $BACK(\{1, 2, 3, 4\}, 4)$  to its parent  $p_3$ . Moreover, it defines  $routing_4[3] = 3$ .
5. When  $p_3$  receives this message, as  $neighbors_3 \subset visited$  and it is not the starting process, it forwards  $BACK(\{1, 2, 3, 4\}, 4)$  to its parent  $p_2$  and defines  $routing_3[2] = 4$ .
6. When  $p_2$  receives this message, as  $neighbors_3$  is not included in  $visited$ , it selects its not yet visited neighbor  $p_5$  and sends it the message  $GO(\{1, 2, 3, 4\}, 4)$ . It also defines  $routing_2[5] = 3$ .
7. When  $p_5$  receives this message, it defines  $p_4$  as its successor on the logical ring ( $succ_5 = 4$ ), sends back to its parent  $p_2$  the message  $BACK(\{1, 2, 3, 4, 5\}, 5)$  and defines  $routing_5[2] = 2$ .
8. When  $p_2$  receives this message,  $p_2$  forwards it to its parent  $p_1$  and defines  $routing_2[1] = 5$ .

**Table 1.1** The paths implementing the virtual channels of the logical ring

Virtual channel of the ring	Implemented by the path
$2 \rightarrow 1$	$2 \rightarrow 1$
$1 \rightarrow 5$	$1 \rightarrow 2 \rightarrow 5$
$5 \rightarrow 4$	$5 \rightarrow 2 \rightarrow 3 \rightarrow 4$
$4 \rightarrow 3$	$4 \rightarrow 3$
$3 \rightarrow 2$	$3 \rightarrow 2$

9. Finally, when  $p_1$  receives the message  $\text{BACK}(\{1, 2, 3, 4, 5\}, 5)$ , all its neighbors have been visited. Hence, the depth-first traversal is terminated. Consequently,  $p_1$  closes the ring by assigning its value to  $\text{routing}_1[\text{first}_1]$ , i.e., it defines  $\text{routing}_1[2] = 2$ .

The routing tables at each process constitute a distributed implementation of the paths followed by the token to circulate on the ring from each process to its successor. These paths are summarized in Table 1.1 which describes the physical paths implementing the  $n$  virtual channels of the logical ring.

## 1.5 Summary

After defining the notion of a distributed algorithm, this chapter has presented several traversal network algorithms, namely, parallel, breadth-first, and depth-first traversal algorithms. It has also presented algorithms that construct spanning trees or rings on top of a communication graph. In addition to being interesting in their own right, these algorithms show that distributed traversal techniques are different from their sequential counterparts.

## 1.6 Bibliographic Notes

- Network traversal algorithms are presented in several books devoted to distributed computing, e.g., [24, 150, 219, 242, 335]. Several distributed traversal algorithms are described by A. Segall in [341].
- A distributed version of Ford and Fulkerson's algorithm for the maximum flow problem is described in [91]. This adaptation is based on a distributed depth-first search algorithm.
- The notion of a wave-based distributed iteration has been extensively studied in several books, e.g., [308, 365]. The book by Raynal and H elary [319] is entirely devoted to wave-based and phase-based distributed algorithms.
- The improved depth-first traversal algorithm with  $O(n)$  time complexity is due to B. Awerbuch [25]. The technique of controlling the progress of a network traversal with a set *visited* carried by messages is due to J.-M. H elary, A. Maddi and

M. Raynal [170]. Other depth-first traversal algorithms have been proposed in the literature, e.g., [95, 224].

- The breadth-first traversal algorithm without centralized control is due to C.-T. Cheung [91]. The one with centralized control is due to Y. Zhu and C.-T. Cheung [395].
- The algorithm building a logical ring on top of an arbitrary network is due to J.-M. Hélarly and M. Raynal [177]. It is shown in this paper how the basic algorithm described in Fig. 1.19 can be improved to obtain an implementation of the ring requiring a number of messages  $\leq 2n$  to implement a full turn of the token on the ring.
- The distributed construction of minimum weight spanning trees has been addressed in many papers, e.g., [143, 209, 231].
- Graph algorithms and algorithmic graph theory can be found in many textbooks (e.g., [122, 158]). The book by M. van Steen [359] constitutes an introduction to graph and complex networks for engineers and computer scientists. Recent advances on graph theory are presented in the collective book [164].

## 1.7 Exercises and Problems

1. Let us assume that a message `GO()` is allowed to carry the position of its sender in the communication graph. How can we improve a distributed graph traversal algorithm so that it can benefit from this information?

Solution in [319].

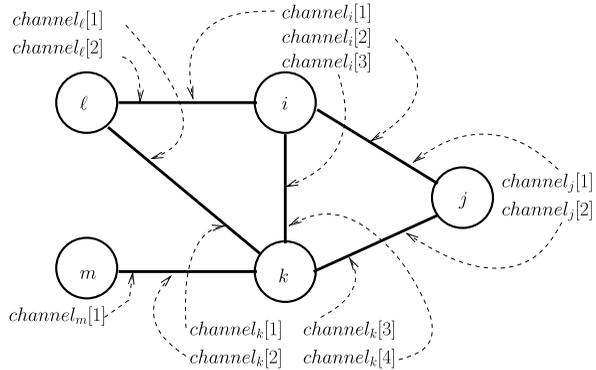
2. Write the full text of the depth-first traversal algorithm corresponding to the improvement presented in the part titled “An easy improvement of the basic algorithm” of Sect. 1.4.1. Depict then a run of it on the communication graph described in the left part of Fig. 1.8.
3. Let us consider the case of a directed communication graph where the meaning of “directed” is as follows. A channel from  $p_i$  to  $p_j$  allows (a)  $p_i$  to send only messages `GO()` to  $p_j$  and (b)  $p_j$  to send only messages `BACK()` to  $p_i$ . Two processes  $p_i$  and  $p_j$  are then such that either there is no communication channel connecting them, or there is one directed communication channel connecting one to the other, or there are two directed communication channels (one in each direction).

Design a distributed algorithm building a breadth-first spanning tree with a distinguished root  $p_a$  and compare it with the algorithm described in Fig. 1.11. It is assumed that there is a directed path from the distinguished root  $p_a$  to any other process.

Solution in [91].

4. Consider a communication graph in which the processes have no identity and each process  $p_i$  knows its position in the communication network with the help of a local array `channeli[1..ci]` (where  $c_i$  is the number of neighbors of  $p_i$ ). An example is given in Fig. 1.22. As we can see in this figure, the channel connecting

**Fig. 1.22**  
An anonymous network



$p_i$  and  $p_j$  is locally known by  $p_i$  as  $channel_i[2]$  and locally known by  $p_j$  as  $channel_j[1]$ .

Using a distributed depth-first traversal algorithm where a distinguished process receives a message  $START()$ , design an algorithm that associates with each process  $p_i$  an identity  $id_i$  and an identity table  $neighbor\_name_i[1..c_i]$  such that:

- $\forall i : id_i \in \{1, 2, \dots, n\}$ .
- $\forall i, j : id_i \neq id_j$ .
- If  $p_i$  and  $p_j$  are neighbors and the channel connecting them is denoted  $channel_i[k]$  at  $p_i$  and  $channel_j[\ell]$  at  $p_j$ , we have  $neighbor\_name_i[k] = id_j$  and  $neighbor\_name_j[\ell] = id_i$ .

5. Enrich the content of the messages  $GO()$  and  $BACK()$  of the algorithm described in Fig. 1.19 in order to obtain a distributed algorithm that builds a logical ring and uses as few messages as possible. This means that the number of messages  $x$  will be such that  $n \leq x \leq 2(n - 1)$ . Moreover, the length of the logical ring (counted as the number of channels of the underlying communication graph) has to be equal to  $x$ .

Solution in [177].

6. Design a distributed algorithm that builds a logical ring such that the length of the path in the communication graph of any two neighbors in the logical ring is at most 3.

Let us recall that, if  $G$  is a connected graph,  $G^3$  is Hamiltonian. Hence, the construction is possible. (Given a graph  $G$ ,  $G^k$  is a graph with the same vertices as  $G$  and any two vertices of  $G^k$  are neighbors if their distance in  $G$  is at most  $k$ .)

# Chapter 2

## Distributed Graph Algorithms

This chapter addresses three basic graph problems encountered in the context of distributed systems. These problems are (a) the computation of the shortest paths between a pair of processes where a positive length (or weight) is attached to each communication channel, (b) the coloring of the vertices (processes) of a graph in  $\Delta + 1$  colors (where  $\Delta$  is the maximal number of neighbors of a process, i.e., the maximal degree of a vertex when using the graph terminology), and (c) the detection of knots and cycles in a graph. As for the previous chapter devoted to graph traversal algorithms, an aim of this chapter is not only to present specific distributed graph algorithms, but also to show that their design is not always obtained from a simple extension of their sequential counterparts.

**Keywords** Distributed graph algorithm · Cycle detection · Graph coloring · Knot detection · Maximal independent set · Problem reduction · Shortest path computation

### 2.1 Distributed Shortest Path Algorithms

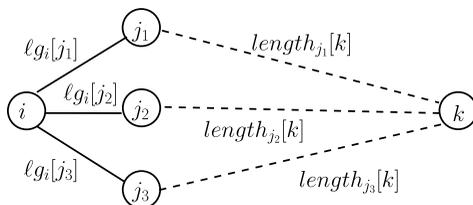
This section presents distributed algorithms which allow each process to compute its shortest paths to every other process in the system. These algorithms can be seen as “adaptations” of centralized algorithmic principles to the distributed context.

The notations are the same as in the previous chapter. Each process  $p_i$  has a set of neighbors denoted  $neighbors_i$ ; if it exists, the channel connecting  $p_i$  and  $p_j$  is denoted  $\langle i, j \rangle$ . The communication channels are bidirectional (hence  $\langle i, j \rangle$  and  $\langle j, i \rangle$  denote the same channel). Moreover, the communication graph is connected and each channel  $\langle i, j \rangle$  has a positive length (or weight) denoted  $\ell g_i[j]$  (as  $\langle i, j \rangle$  and  $\langle j, i \rangle$  are the same channel, we have  $\ell g_i[j] = \ell g_j[i]$ ).

#### 2.1.1 A Distributed Adaptation of Bellman–Ford’s Shortest Path Algorithm

Bellman–Ford’s sequential algorithm computes the shortest paths from one predetermined vertex of a graph to every other vertex. It is an iterative algorithm based

**Fig. 2.1** Bellman–Ford’s dynamic programming principle



on the dynamic programming principle. This principle and its adaptation to a distributed context are presented below.

**Initial Knowledge and Local Variables** Initially each process knows that there are  $n$  processes and the set of process identities is  $\{1, \dots, n\}$ . It also knows its position in the communication graph (which is captured by the set  $neighbors_i$ ). Interestingly, it will never learn more on the structure of this graph. From a local state point of view, each process  $p_i$  manages the following variables.

- As just indicated,  $lg_i[j]$ , for  $j \in neighbors_i$ , denotes the length associated with the channel  $\langle i, j \rangle$ .
- $length_i[1..n]$  is an array such that  $length_i[k]$  will contain the length of the shortest path from  $p_i$  to  $p_k$ . Initially,  $length_i[i] = 0$  (and keeps that value forever) while  $length_i[j] = +\infty$  for  $j \neq i$ .
- $routing\_to_i[1..n]$  is an array that is not used to compute the shortest paths from  $p_i$  to each other process. It constitutes the local result of the computation. More precisely, when the algorithm terminates, for any  $k$ ,  $1 \leq k \leq n$ ,  $routing\_to_i[k] = j$  means that  $p_j$  is a neighbor of  $p_i$  on a shortest path to  $p_k$ , i.e.,  $p_j$  is an optimal neighbor when  $p_i$  has to send information to  $p_k$  (where optimality is with respect to the length of the path from  $p_i$  to  $p_k$ ).

**Bellman–Ford Principle** The dynamic programming principle on which the algorithm relies is the following. The local inputs at each process  $p_i$  are the values of the set  $neighbors_i$  and the array  $lg_i[neighbors_i]$ . The output at each process  $p_i$  is the array  $length_i[1..n]$ . The algorithm has to solve the following set of equations (where the unknown variables are the arrays  $length_i[1..n]$ ):

$$\forall i, k \in \{1, \dots, n\}: \quad length_i[k] = \min_{j \in neighbors_i} (lg_i[j] + length_j[k]).$$

The meaning of this formula is depicted in Fig. 2.1 for a process  $p_i$  such that  $neighbors_i = \{j_1, j_2, j_3\}$ . Each dotted line from  $p_{j_x}$  to  $p_k$ ,  $1 \leq x \leq 3$ , represents the shortest path joining  $p_{j_x}$  to  $p_k$  and its length is  $length_{j_x}[k]$ . The solution of this set of equations is computed asynchronously and iteratively by the  $n$  processes, each process  $p_i$  computing successive approximate values of its local array  $length_i[1..n]$  until it stabilizes at its final value.

**The Algorithm** The algorithm is described in Fig. 2.2. At least one process  $p_i$  has to receive the external message `START()` in order to launch the algorithm. It

```

when START() is received do
(1) for each  $j \in \text{neighbors}_i$  do send UPDATE( $length_i$ ) to  $p_j$  end for.

when UPDATE( $length$ ) is received from  $p_j$  do
(2)  $updated_i \leftarrow false$ ;
(3) for each  $k \in \{1, \dots, n\} \setminus \{i\}$  do
(4)   if ( $length_i[k] > \ell_{g_i}[j] + length[k]$ )
(5)     then  $length_i[k] \leftarrow \ell_{g_i}[j] + length[k]$ ;
(6)        $routing\_to_i[k] \leftarrow j$ ;
(7)      $updated_i \leftarrow true$ 
(8)   end if
(9) end for;
(10) if ( $updated_i$ )
(11)   then for each  $j \in \text{neighbors}_i$  do send UPDATE( $length_i$ ) to  $p_j$  end for
(12) end if.

```

**Fig. 2.2** A distributed adaptation of Bellman–Ford’s shortest path algorithm (code for  $p_i$ )

sends then to each of its neighbors the message UPDATE( $length_i$ ) which describes its current local state as far as the computation of the length of its shortest paths to each other process is concerned.

When a process  $p_i$  receive a message UPDATE( $length$ ) from one of its neighbors  $p_j$ , it applies the forward/discard strategy introduced in Chap. 1. To that end,  $p_i$  first strives to improve its current approximation of its shortest paths to any destination process (lines 3–9). Then, if  $p_i$  has discovered shorter paths than the ones it knew before,  $p_i$  sends its new current local state to each of its neighbors (lines 10–12). If its local state (captured by the array  $length_i[1..n]$ ) has not been modified,  $p_i$  does not send a message to its neighbors.

**Termination** While there is a finite time after which the arrays  $length_i[1..n]$  and  $routing\_to_i[1..n]$ ,  $1 \leq i \leq n$ , have obtained their final values, no process ever learns when this time has occurred.

**Adding Synchronization in Order that Each Process Learns Termination** The algorithm described in Fig. 2.3 allows each process  $p_i$  not only to compute the shortest paths but also to learn that it knows them (i.e., learn that its local arrays  $length_i[1..n]$  and  $routing\_to_i[1..n]$  have converged to their final values).

This algorithm is synchronous: the processes execute a sequence of synchronous rounds, and rounds are given for free: they belong to the computation model. During each round  $r$ , in addition to local computation, each process sends a message to and receives a message from each of its neighbors. The important synchrony property lies in the fact that a message sent by a process  $p_i$  to a neighbor  $p_j$  at round  $r$  is received and processed by  $p_j$  during the very same round  $r$ . The progress of a round  $r$  to the round  $r + 1$  is governed by the underlying system. (A general technique to simulate a synchronous algorithm on top of an asynchronous system will be described in Chap. 9.)

```

when  $r = 1, 2, \dots, D$  do
begin synchronous round
(1) for each  $j \in neighbors_i$  do send UPDATE( $length_i$ ) to  $p_j$  end for;
(2) for each  $j \in neighbors_i$  do receive UPDATE( $length_j$ ) from  $p_j$  end for;
(3) for each  $k \in \{1, \dots, n\} \setminus \{i\}$  do
(4)   let  $length_{ik} \leftarrow \min_{j \in neighbors_i} (\ell_{g_i[j]} + length_j[k]);$ 
(5)   if ( $length_{ik} < length_i[k]$ ) then
(6)      $length_i[k] \leftarrow length_{ik};$ 
(7)      $routing\_to_i[k] \leftarrow$  a neighbor  $j$  that realizes the previous minimum
(8)   end if
(9) end for
end synchronous round.

```

**Fig. 2.3** A distributed synchronous shortest path algorithm (code for  $p_i$ )

The algorithm considers that the diameter  $D$  of the communication graph is known by the processes (let us remember that the diameter is the number of channels separating the two most distant processes). If  $D$  is not explicitly known, it can be replaced by an upper bound, namely the value  $(n - 1)$ .

The text of the algorithm is self-explanatory. There is a strong connection between the current round number and the number of channels composing the paths from which  $p_i$  learns information. Let us consider a process  $p_i$  at the end of a round  $r$ .

- When  $r < D$ ,  $p_i$  knows the shortest path from itself to any other process  $p_k$ , which is composed of at most  $r$  channels. Hence, this length to  $p_k$  is not necessarily the shortest one.
- Differently, when it terminates round  $r = D$ ,  $p_i$  has computed both (a) the shortest lengths from it to all the other processes and (b) the corresponding appropriate routing neighbors.

### 2.1.2 A Distributed Adaptation of Floyd–Warshall’s Shortest Paths Algorithm

Floyd–Warshall’s algorithm is a sequential algorithm that computes the shortest paths between any two vertices of a non-directed graph. This section presents an adaptation of this algorithm to the distributed context. This adaptation is due to S. Toueg (1980). As previously, to make the presentation easier, we consider that the graph (communication network) is connected and the length of each edge of the graph (communication channel) is a positive number. (Actually, the algorithm works when edges have negative lengths as long as no cycle has a negative length.)

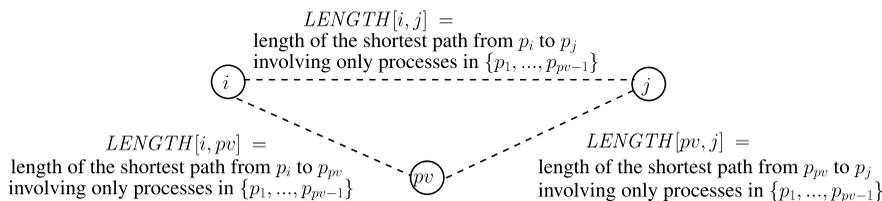
**Floyd–Warshall’s Sequential Algorithm** Let  $LENGTH[1..n, 1..n]$  be a matrix such that, when the algorithm terminates,  $LENGTH[i, j]$  represents the length of the shortest path from  $p_i$  to  $p_j$ . Initially, for any  $i$ ,  $LENGTH[i, i] = 0$ , for any pair  $(i, j)$

```

(1) for  $p_v$  from 1 to  $n$  do
(2)   for  $i$  from 1 to  $n$  do
(3)     for  $j$  from 1 to  $n$  do
(4)       if  $LENGTH[i, p_v] + LENGTH[p_v, j] < LENGTH[i, j]$ 
(5)         then  $LENGTH[i, j] \leftarrow LENGTH[i, p_v] + LENGTH[p_v, j]$ ;
(6)            $routing\_to_i[j] \leftarrow routing\_to_i[p_v]$ 
(7)       end if
(8)     end for
(9)   end for
(10) end for.

```

**Fig. 2.4** Floyd–Warshall’s sequential shortest path algorithm



**Fig. 2.5** The principle that underlies Floyd–Warshall’s shortest paths algorithm

such that  $j \in neighbors_i$ ,  $LENGTH[i, j]$  is initialized to the length of the channel from  $p_i$  to  $p_j$ , and  $LENGTH[i, j] = +\infty$  in the other cases. Moreover, for any  $i$ , the array  $routing\_to_i[1..n]$  is such that  $routing\_to_i[i] = i$ ,  $routing\_to_i[j] = j$  for each  $j \in neighbors_i$  and  $routing\_to_i[j]$  is initially undefined for the other values of  $j$ .

The principle of Floyd–Warshall’s algorithm is an iterative algorithm based on the following principle. For any process  $p_i$ , the algorithm computes first the shortest path from any process  $p_i$  to any process  $p_j$  that (if any) passes through process  $p_1$ . Then, it computes the shortest path from any process  $p_i$  to any process  $p_j$  among all the paths from  $p_i$  to  $p_j$  which pass only through processes in the set  $\{p_1, p_2\}$ . More generally, at the step  $p_v$  of the iteration, the algorithm computes the shortest path from any process  $p_i$  to any process  $p_j$  among all the paths from  $p_i$  to  $p_j$  which are allowed to pass through the set of processes  $\{p_1, \dots, p_{p_v}\}$ . The text of the algorithm is given in Fig. 2.4. As we can see, the algorithm is made up of three nested **for** loops. The external one defines the processes (namely  $p_1, \dots, p_{p_v}$ ) allowed to appear in current computation of the shortest from any process  $p_i$  to any process  $p_j$ . The process index  $p_v$  is usually called the *pivot*.

The important feature of this sequential algorithm lies in the fact that, when computing the shortest path from  $p_i$  to  $p_j$  involving the communication channels connecting the processes in the set  $\{p_1, \dots, p_{p_v}\}$ , the variable  $LENGTH[i, p_v]$  contains the length of the shortest path from  $p_i$  to  $p_{p_v}$  involving only the communication channels connecting the processes in the set  $\{p_1, \dots, p_{p_v-1}\}$  (and similarly for the variable  $LENGTH[p_v, j]$ ). This is described in Fig. 2.5 when considering the computation of the shortest from  $p_i$  to  $p_j$  involving the processes  $\{p_1, \dots, p_{p_v}\}$  (this constitutes the  $p_v$ th iteration step of the external loop).

**From a Sequential to a Distributed Algorithm** As a distributed system has no central memory and communication is by message passing between neighbor processes, two issues have to be resolved to obtain a distributed algorithm. The first concerns the distribution of the data structures; the second the synchronization of processes so that they can correctly compute the shortest paths and the associated routing tables.

The array  $LENGTH[1..n, 1..n]$  is split into  $n$  vectors such that each process  $p_i$  computes and maintains the value of  $LENGTH[i, 1..n]$  in its local array  $length_i[1..n]$ . Moreover, as seen before, each process  $p_i$  computes the value of its routing local array  $routing\_to_i[1..n]$ . On the synchronization side, there are two issues:

- When  $p_i$  computes the value of  $length_i[j]$  during the iteration step  $pv$ , process  $p_i$  locally knows the current values of  $length_i[j]$  and  $length_i[pv]$ , but it has to obtain the current value of  $length_{pv}[j]$  (see line 4 of Fig. 2.4).
- To obtain from  $p_{pv}$  a correct value for  $length_{pv}[j]$ , the processes must execute simultaneously the same iteration step  $pv$ . If a process  $p_i$  is executing an iteration step with the pivot value  $pv$  while another process  $p_k$  is simultaneously executing an iteration step with the pivot value  $pv' \neq pv$ , the values they obtain, respectively, from  $p_{pv}$  for  $length_{pv}[j]$  and from  $p_{pv'}$   $length_{pv'}[j]$  can be mutually inconsistent if these computations are done without an appropriate synchronization.

**The Distributed Algorithm** The algorithm is described in Fig. 2.6. The processes execute concurrently a loop where the index  $pv$  takes the successive values from 1 to  $n$  (line 1). If a process receives a message while it has not yet started executing its local algorithm, it locally starts the local algorithm before processing the message. As the communication graph is connected, it follows that, as soon as at least one process  $p_i$  starts its local algorithm, all the processes start theirs.

As indicated just previously, when the processes execute the iteration step  $pv$ , the process  $p_{pv}$  has to broadcast its local array  $length_{pv}[1..n]$  so that each process  $p_i$  to try to improve its shortest distance to any process  $p_j$  as indicated in Fig. 2.5.

To this end, let us observe that if, at the  $pv$ th iteration of the loop, there is path from  $p_i$  to  $p_{pv}$  involving only processes in the set  $\{p_1, \dots, p_{pv-1}\}$ , there is then a favorite neighbor to attain  $p_{pv}$ , namely the process whose index has been computed and saved in  $routing\_to_i[pv]$ . This means that, at the  $pv$ th iteration, the set of local variables  $routing\_to_x[pv]$  of the processes  $p_x$  such that  $length_x[pv] \neq +\infty$  define a tree rooted at  $p_{pv}$ .

The algorithm executed by the processes, which ensures a correct process coordination, follows from this observation. More precisely, a local algorithm is made up of three parts:

- Part 1: lines 1–6. A process  $p_i$  first sends a message to each of its neighbors  $p_k$  indicating if  $p_i$  is or not one of  $p_k$ 's children in the tree rooted at  $p_{pv}$ . It then waits until it has received such a message from each of its neighbors.

Then,  $p_i$  executes the rest of the code for the  $pv$ th iteration only if it has a chance to improve its shortest paths with the help of  $p_{pv}$ , i.e., if  $length_i[pv] \neq +\infty$ .

```

(1) for  $p_v$  from 1 to  $n$  do
(2)   for each  $k \in neighbors_i$  do
(3)     if ( $routing\_to_i[p_v] = k$ ) then  $child \leftarrow yes$  else  $child \leftarrow no$  end if;
(4)     send CHILD( $p_v, child$ ) to  $p_k$ 
(5)   end for;
(6)   wait (a message CHILD( $p_v, -$ ) received from each neighbor);
(7)   if ( $length_i[p_v] \neq +\infty$ ) then
(8)     if ( $p_v \neq i$ ) then
(9)       wait (message PV_LENGTH( $p_v, p_v\_length[1..n]$ ) from  $p_{routing\_to_i[p_v]}$ )
(10)    end if;
(11)   for each  $k \in neighbors_i$  do
(12)     if (CHILD( $p_v, yes$ ) received from  $p_k$ ) then
(13)       if ( $p_v = i$ ) then send PV_LENGTH( $p_v, length_i[1..n]$ ) to  $p_k$ 
(14)       else send PV_LENGTH( $p_v, p_v\_length[1..n]$ ) to  $p_k$ 
(15)     end if
(16)   end if
(17) end for;
(18) for  $j$  from 1 to  $n$  do
(19)   if  $length_i[p_v] + p_v\_length[j] < length_i[j]$ 
(20)     then  $length_i[j] \leftarrow length_i[p_v] + p_v\_length[j]$ ;
(21)      $routing\_to_i[j] \leftarrow routing\_to_i[p_v]$ 
(22)   end if
(23) end for
(24) end if
(25) end for.

```

**Fig. 2.6** Distributed Floyd–Warshall’s shortest path algorithm

- Part 2: lines 8–17. This part of the algorithm ensures that each process  $p_i$  such that  $length_i[p_v] \neq +\infty$  receives a copy of the array  $length_{p_v}[1..n]$  so that it can recompute the values of its shortest paths and the associated local routing table (which is done in Part 3).

The broadcast of  $length_{p_v}[1..n]$  by  $p_{p_v}$  is launched at line 13, where this process sends the message PV\_LENGTH( $p_v, length_{p_v}$ ) to all its children in the tree whose it is the root. When it receives such a message carrying the value  $p_v$  and the array  $p_v\_length[1..n]$  (line 9), a process  $p_i$  forwards it to its children in the tree rooted at  $p_{p_v}$  (lines 12 and 14).

- Part 3: lines 18–23. Finally, a process  $p_i$  uses the array  $p_v\_length[1..n]$  it has received in order to improve its shortest paths that pass through the processes  $p_1, \dots, p_{p_v}$ .

**Cost** Let  $e$  be the number of communication channels. It is easy to see that, during each iteration, (a) at most two messages CHILD() are sent on each channel (one in each direction) and (b) at most  $(n - 1)$  messages PV\_LENGTH() are sent. It follows that the number of messages is upper-bounded by  $n(2e + n)$ ; i.e., the message complexity is  $O(n^3)$ . As far the size of messages is concerned, a message CHILD() carries a bit, while PV\_LENGTH() carries  $n$  values whose size depends on the individual lengths associated with the communication channels.

```

(1) for  $i$  from 1 to  $n$  do
(2)    $c \leftarrow 1$ ;
(3)   while ( $COLOR[i] = \perp$ ) do
(4)     if ( $\bigwedge_{j \in neighbors_i} COLOR[j] \neq c$ ) then  $COLOR[i] \leftarrow c$  else  $c \leftarrow c + 1$  end if
(5)   end while
(6) end for.

```

**Fig. 2.7** Sequential  $(\Delta + 1)$ -coloring of the vertices of a graph

Finally, there are  $n$  iteration steps, and each has  $O(n)$  time complexity. Moreover, in the worst case, the processes starts the algorithm one after the other (a single process starts, which entails the start of another process, etc.). When summing up, it follows that the time complexity is upper-bounded by  $O(n^2)$ .

## 2.2 Vertex Coloring and Maximal Independent Set

### 2.2.1 On Sequential Vertex Coloring

**Vertex Coloring** An important graph problem, which is encountered when one has to model application-level problems, concerns vertex coloring. It consists in assigning a value (color) to each vertex such that (a) no two vertices which are neighbors have the same color, and (b) the number of colors is “reasonably small”. When the number of colors has to be the smallest possible one, the problem is NP-complete.

Let  $\Delta$  be the maximal degree of a graph (let us remember that, assuming a graph where any two vertices are connected by at most one edge, the degree of a vertex is the number of its neighbors). It is always possible to color the vertices of a graph in  $\Delta + 1$  colors. This follows from the following simple reasoning by induction. The assertion is trivially true for any graph with at most  $\Delta$  vertices. Then, assuming it is true for any graph made up of  $n \geq \Delta$  vertices and whose maximal degree is at most  $\Delta$ , let us add a new vertex to the graph. As (by assumption) the maximal degree of the graph is  $\Delta$ , it follows that this new vertex has at most  $\Delta$  neighbors. Hence, this vertex can be colored with the remaining color.

**A Simple Sequential Algorithm** A simple sequential algorithm that colors vertices in at most  $(\Delta + 1)$  colors is described in Fig. 2.7. The array variable  $COLOR[1..n]$ , which is initialized to  $[\perp, \dots, \perp]$ , is such that, when the algorithm terminates, for any  $i$ ,  $COLOR[i]$  will contain the color assigned to process  $p_i$ .

The colors are represented by the integers 1 to  $(\Delta + 1)$ . The algorithm considers sequentially each vertex  $i$  (process  $p_i$ ) and assigns to it the first color not assigned to its neighbors. (This algorithm is sensitive to the order in which the vertices and the colors are considered.)

```

(1) for each  $j \in neighbors_i$  do send INIT( $color_i[i]$ ) to  $p_j$  end for;
(2) for each  $j \in neighbors_i$ 
(3)     do wait (INIT( $col\_j$ ) received from  $p_j$ );  $color_i[j] \leftarrow col\_j$ 
(4) end for;
(5) for  $r_i$  from  $(\Delta + 2)$  to  $m$  do
    begin asynchronous round
(6)     if ( $color_i[i] = r_i$ )
(7)         then  $c \leftarrow$  smallest color in  $\{1, \dots, \Delta + 1\}$  such that  $\forall j \in neighbors_i : color_i[j] \neq c$ ;
(8)          $color_i[i] \leftarrow c$ 
(9)     end if;
(10)    for each  $j \in neighbors_i$  do send COLOR( $r_i, color_i[i]$ ) to  $p_j$  end for;
(11)    for each  $j \in neighbors_i$  do
(12)        wait (COLOR( $r, col\_j$ ) with  $r = r_i$  received from  $p_j$ );
(13)         $color_i[j] \leftarrow col\_j$ 
(14)    end for
    end asynchronous round
(15) end for.

```

**Fig. 2.8** Distributed  $(\Delta + 1)$ -coloring from an initial  $m$ -coloring where  $n \geq m \geq \Delta + 2$

### 2.2.2 Distributed $(\Delta + 1)$ -Coloring of Processes

This section presents a distributed algorithm which colors the processes in at most  $(\Delta + 1)$  colors in such a way that no two neighbors have the same color. Distributed coloring is encountered in practical problems such as resource allocation or processor scheduling. More generally, distributed coloring algorithms are symmetry breaking algorithms in the sense that they partition the set of processes into subsets (a subset per color) such that no two processes in the same subset are neighbors.

**Initial Context of the Distributed Algorithm** Such a distributed algorithm is described in Fig. 2.8. This algorithm assumes that the processes are already colored in  $m \geq \Delta + 1$  colors in such a way that no two neighbors have the same color. Let us observe that, from a computability point of view, this is a “no-cost” assumption (because taking  $m = n$  and defining the color of a process  $p_i$  as its index  $i$  trivially satisfies this initial coloring assumption). Differently, taking  $m = \Delta + 1$  assumes that the problem is already solved. Hence, the assumption on the value of  $m$  is a complexity-related assumption.

**Local Variables** Each process  $p_i$  manages a local variable  $color_i[i]$  which initially contains its initial color, and will contain its final color at the end of the algorithm. A process  $p_i$  also manages a local variable  $color_i[j]$  for each of its neighbors  $p_j$ . As the algorithm is asynchronous and round-based, the local variable  $r_i$  managed by  $p_i$  denotes its current local round number.

**Behavior of a Process  $p_i$**  The processes proceed in consecutive asynchronous rounds and, at each round, each process synchronizes its progress with its neighbors. As the rounds are asynchronous, the round numbers are not given for free by

the computation model. They have to be explicitly managed by the processes themselves. Hence, each process  $p_i$  manages a local variable  $r_i$  that it increases when it starts a new asynchronous round (line 5).

The first round (lines 1–2) is an initial round during which the processes exchange their initial color in order to fill in their local array  $color_i[neighbors_i]$ . If the processes know the initial colors of their neighbors, this communication round can be suppressed. The processes then execute  $m - (\Delta + 1)$  asynchronous rounds (line 5).

The processes whose initial color belongs to the set of colors  $\{1, \dots, \Delta + 1\}$  keep their color forever. The other processes update their colors in order to obtain a color in  $\{1, \dots, \Delta + 1\}$ . To that end, all the processes execute sequentially the rounds  $\Delta + 2, \dots$ , until  $m$ , considering that each round number corresponds to a given distinct color. During round  $r$ ,  $\Delta + 2 \leq r \leq m$ , each process whose initial color is  $r$  looks for a new color in  $\{1, \dots, \Delta + 1\}$  which is not the color of its neighbors and adopts it as its new color (lines 6–8). Then, each process exchanges its color with its neighbors (lines 10–14) before proceeding to the next round. Hence, the round invariant is the following one: When a round  $r$  terminates, the processes whose initial colors were in  $\{1, \dots, r\}$  (a) have a color in the set  $\{1, \dots, \Delta + 1\}$ , and (b) have different colors if they are neighbors.

**Cost** The time complexity (counted in number of rounds) is  $m - \Delta$  rounds (an initial round plus  $m - (\Delta + 1)$  rounds). Each message carries a tag, a color, and possibly a round number which is also a color. As the initial colors are in  $\{1, \dots, m\}$ , the message bit complexity is  $O(\log_2 m)$ .

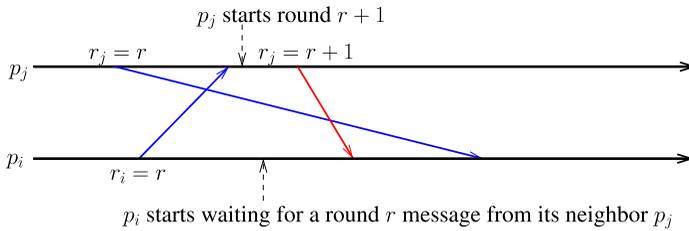
Finally, during each round, two messages are sent on each channel. The message complexity is consequently  $2e(m - \Delta)$ , where  $e$  denotes the number of channels.

It is easy to see that, the better the initial process coloring (i.e., the smaller the value of  $m$ ), the more efficient the algorithm.

**Theorem 1** *Let  $m \geq \Delta + 2$ . The algorithm described in Fig. 2.8 is a legal  $(\Delta + 1)$ -coloring of the processes (where legal means that no two neighbors have the same color).*

*Proof* Let us first observe that the processes whose initial color belongs to  $\{1, \dots, \Delta + 1\}$  never modify their color. Let us assume that, up to round  $r$ , the processes whose initial colors were in the set  $\{1, \dots, r\}$  have new colors in the set  $\{1, \dots, \Delta + 1\}$  and any two of them which are neighbors have different colors. Thanks to the initial  $m$ -coloring, this is initially true (i.e., for the fictitious round  $r = \Delta + 1$ ).

Let us assume that the previous assertion is true up to some round  $r \geq \Delta + 1$ . It follows from the algorithm that, during round  $r + 1$ , only the processes whose current color is  $r + 1$  update it. Moreover, each of them updates it (line 7) with a color that (a) belongs to the set  $\{1, \dots, \Delta + 1\}$  and (b) is not a color of its neighbors (we have seen in Sect. 2.2.1 that such a color does exist). Consequently, at the end of round  $r + 1$ , the processes whose initial colors were in the set  $\{1, \dots, r + 1\}$



**Fig. 2.9** One bit of control information when the channels are not FIFO

have new colors in the set  $\{1, \dots, \Delta + 1\}$  and no two of them have the same new color if they are neighbors. It follows that, as claimed, this property constitutes a round invariant from which we conclude that each process has a final color in the set  $\{1, \dots, \Delta + 1\}$  and no two neighbor processes have the same color.  $\square$

**Remark on the Behavior of the Communication Channels** Let us remember that the only assumption on channels is that they are reliable. No other behavioral assumption is made, hence the channels are implicitly non-FIFO channels.

Let us consider two neighbor processes that execute a round  $r$  as depicted in Fig. 2.9. Each of them sends its message  $\text{COLOR}(r, -)$  to its neighbors (line 10), and waits for a message  $\text{COLOR}()$  from each of them, carrying the very same round number (line 12).

In the figure,  $p_j$  has received the round  $r$  message from  $p_i$ , proceeded to the next round, and sent the message  $\text{COLOR}(r + 1, -)$  to  $p_i$  while  $p_i$  is still waiting for round  $r$  message from  $p_j$ . Moreover, as the channel is not FIFO, the figure depicts the case where the message  $\text{COLOR}(r + 1, -)$  sent by  $p_j$  to  $p_i$  arrives before the message  $\text{COLOR}(r, -)$  it sent previously. As indicated in line 12, the algorithm forces  $p_i$  to wait for the message  $\text{COLOR}(r, -)$  in order to terminate its round  $r$ .

As, in each round, each process sends a message to each of its neighbors, a closer analysis of the message exchange pattern shows that the following relation on round numbers is invariant. At any time we have:

$$\forall(i, j) : (p_i \text{ and } p_j \text{ are neighbors}) \Rightarrow (0 \leq |r_i - r_j| \leq 1).$$

It follows that the message  $\text{COLOR}()$  does not need to carry the value of  $r$  but only a bit, namely the parity of  $r$ . The algorithm can then be simplified as follows:

- At line 10, each process  $p_i$  sends the message  $\text{COLOR}(r_i \bmod 2, \text{color}_i[i])$  to each of its neighbors.
- At line 12, each process  $p_i$  waits for a message  $\text{COLOR}(b, \text{color}_i[i])$  from each of its neighbors where  $b = (r_i \bmod 2)$ .

Finally, it follows from previous discussion that, if the channels are FIFO, the messages  $\text{COLOR}()$  do not need to carry a control value (neither  $r$ , nor its parity bit).

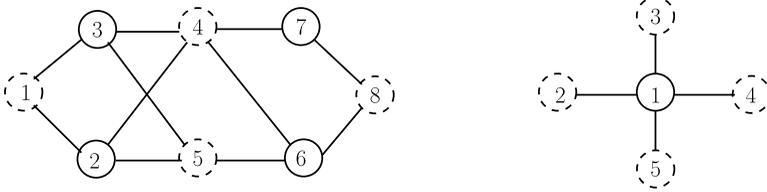


Fig. 2.10 Examples of maximal independent sets

### 2.2.3 Computing a Maximal Independent Set

**Maximal Independent Set: Definition** An *independent set* is a subset of the vertices of a graph such that no two of them are neighbors. An independent set  $M$  is *maximal* if none of its strict supersets  $M'$  (i.e.,  $M \subset M'$  and  $M \neq M'$ ) is an independent set. A graph can have several maximal independent sets.

The subset of vertices  $\{1, 4, 5, 8\}$  of the graph of depicted in the left part of Fig. 2.10 is a maximal independent set. The subsets  $\{1, 5, 7\}$  and  $\{2, 3, 6, 7\}$  are other examples of maximal independent sets of the same graph. The graph depicted on the right part has two maximal independent sets, the set  $\{1\}$  and the set  $\{2, 3, 4, 5\}$ .

There is a trivial greedy algorithm to compute a maximal independent set in a sequential context. Select a vertex, add it to the independent set, suppress it and its neighbors from the graph, and iterate until there are no more vertices. It follows that the problem of computing a maximal independent set belongs to the time complexity class P (the class of problems that can be solved by an algorithm whose time complexity is polynomial).

A *maximum independent set* is an independent set with maximal cardinality. When considering the graph at the left of Fig. 2.10, the maximal independent sets  $\{1, 4, 5, 8\}$  and  $\{2, 3, 6, 7\}$  are maximum independent sets. The graph on the right of the figure has a single maximum independent set, namely the set  $\{2, 3, 4, 5\}$ .

While, from a time complexity point of view, the computation of a maximal independent set is an *easy* problem, the computation of a maximum independent set is a *hard* problem: it belongs to the class of NP-complete problems.

**From  $m$ -Coloring to a Maximal Independent Set** An asynchronous distributed algorithm that computes a maximal independent set is presented in Fig. 2.11. Each process  $p_i$  manages a local array  $selected_i[j]$ ,  $j \in neighbors_i \cup \{i\}$ , initialized to  $[false, \dots, false]$ . At the end of the algorithm  $p_i$  belongs to the maximal independent set if and only if  $selected_i[i]$  is equal to *true*.

This algorithm assumes that there is an initial  $m$ -coloring of the processes (as we have just seen, this can be obtained from the algorithm of Fig. 2.8). Hence, the algorithm of Fig. 2.11 is a distributed reduction of the maximal independent set problem to the  $m$ -coloring problem. Its underlying principle is based on a simple observation and a greedy strategy. More precisely,

- Simple observation: the processes that have the same color define an independent set, but this set is not necessarily maximal.

```

(1) for  $r_i$  from 1 to  $m$  do
    begin asynchronous round
(2)   if ( $color_i = r_i$ ) then
(3)     if ( $\bigwedge_{j \in neighbors_i} (\neg selected_i[j])$ ) then  $selected_i[i] \leftarrow true$  end if;
(4)   end if;
(5)   for each  $j \in neighbors_i$  do send SELECTED( $r_i, selected_i[i]$ ) to  $p_j$  end for;
(6)   for each  $j \in neighbors_i$  do
(7)     wait (SELECTED( $r, selected\_j$ ) with  $r = r_i$  received from  $p_j$ );
(8)      $selected_i[j] \leftarrow selected\_j$ 
(9)   end for
    end asynchronous round
(10) end for.

```

Fig. 2.11 From  $m$ -coloring to a maximal independent set (code for  $p_i$ )

- Greedy strategy: as the previous set is not necessarily maximal, the algorithm starts with an initial independent set (defined by some color) and executes a sequence of rounds, each round  $r$  corresponding to a color, in which it strives to add to the independent set under construction as much possible processes whose color is  $r$ . The corresponding “addition” predicate for a process  $p_i$  with color  $r$  is that none of its neighbors is already in the set.

As previous algorithms, the algorithm described in Fig. 2.11 simulates a synchronous algorithm. The color of a process  $p_i$  is kept in its local variable denoted  $color_i$ . The messages carry a round number (color) which can be replaced by its parity. The processes execute  $m$  asynchronous rounds (a round per color). When it executes round  $r$ , if its color is  $r$  and none of its neighbors belongs to the set under construction, a process  $p_i$  adds itself to the set (line 3). Then, before starting the next round, the processes exchange their membership of the maximal independent set in order to update their local variables  $selected_i[j]$ . (As we can see, what is important is not the fact that the rounds are executed in the order  $1, \dots, m$ , but the fact that the processes execute the rounds in the same predefined order, e.g.,  $1, m, 2, (m - 1), \dots$ )

The size of the maximal independent set that is computed is very sensitive to the order in which the colors are visited by the algorithm. As an example, let us consider the graph at the right of Fig. 2.10 where the process  $p_1$  is colored  $a$  while the other processes are colored  $b$ . If  $a = 1$  and  $b = 2$ , the maximal independent set that is built is the set  $\{1\}$ . If  $a = 2$  and  $b = 1$ , the maximal independent set that is built is the set  $\{2, 3, 4, 5\}$ .

**A Simple Algorithm for Maximal Independent Set** This section presents an algorithm, due to M. Luby (1987), that builds a maximal independent set.

This algorithm uses a random function denoted `random()` which outputs a random value each time it is called (the benefit of using random values is motivated below). For ease of exposition, this algorithm, which is described in Fig. 2.12, is expressed in the synchronous model. Let us remember that the main property of the synchronous model lies in the fact that a message sent in a round is received by its

```

(1) repeat forever
    begin three synchronous rounds  $r, r + 1$  and  $r + 2$ 
    beginning of round  $r$ 
(2)    $random_i[i] \leftarrow random();$ 
(3)   for each  $j \in com\_with_i$  do send RANDOM( $random_i[i]$ ) to  $p_j$  end for;
(4)   for each  $j \in com\_with_i$  do
(5)     wait (RANDOM( $random\_j$ ) received from  $p_j$ );  $random_i[j] \leftarrow random\_j$ 
(6)   end for;
    end of round  $r$  and beginning of round  $r + 1$ 
(7)   if ( $\forall j \in com\_with_i : random_i[j] > random_i[i]$ )
(8)     then for each  $j \in com\_with_i$  do send SELECTED(yes) to  $p_j$  end for;
(9)      $state_i \leftarrow in;$  return( $in$ )
(10)    else for each  $j \in com\_with_i$  do send SELECTED(no) to  $p_j$  end for;
(11)    for each  $j \in com\_with_i$  do wait (SELECTED(-) received from  $p_j$ ) end for;
    end of round  $r + 1$  and beginning of round  $r + 2$ 
(12)    if ( $\exists k \in com\_with_i : SELECTED(yes)$  received from  $p_k$ )
(13)      then for each  $j \in com\_with_i : SELECTED(no)$  received from  $p_j$ 
(14)        do send ELIMINATED(yes) to  $p_j$ 
(15)      end for;
(16)       $state_i \leftarrow out;$  return( $out$ )
(17)    else for each  $j \in com\_with_i$  do send ELIMINATED(no) to  $p_j$  end for;
(18)    for each  $j \in com\_with_i$ 
(19)      do wait (ELIMINATED(-) received from  $p_j$ )
(20)    end for;
(21)    for each  $j \in com\_with_i : ELIMINATED(yes)$  received from  $p_j$ 
(22)       $com\_with_i \leftarrow com\_with_i \setminus \{j\}$ 
(23)    end for;
(24)    if ( $com\_with_i = \emptyset$ ) then  $state_i \leftarrow in;$  return( $in$ ) end if
(25)  end if
(26) end if;
    end three synchronous rounds
(27) end repeat.

```

**Fig. 2.12** Luby's synchronous random algorithm for a maximal independent set (code for  $p_i$ )

destination process in the very same round. (It is easy to extend this algorithm so that it works in the asynchronous model.)

Each process  $p_i$  manages the following local variables.

- The local variable  $state_i$ , whose initial value is arbitrary, is updated only once. Its final value ( $in$  or  $out$ ) indicates whether  $p_i$  belongs or not to the maximal independent set that is computed. When, it has updated  $state_i$  to its final value, a process  $p_i$  executes the statement return() which stops its participation to the algorithm. Let us notice that the processes do not necessarily terminate during the same round.
- The local variable  $com\_with_i$ , which is initialized to  $neighbors_i$ , is a set containing the processes with which  $p_i$  will continue to communicate during the next round.
- Each local variable  $random_i[j]$ , where  $j \in neighbors_i \cup \{i\}$ , represents the local knowledge of  $p_i$  about the last random number used by  $p_j$ .

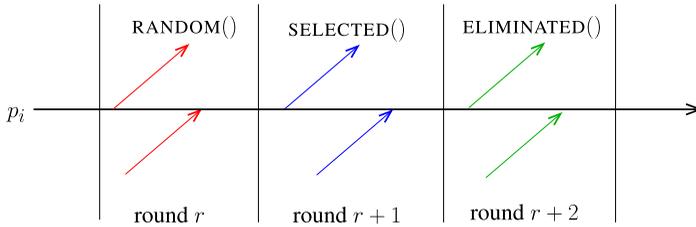


Fig. 2.13 Messages exchanged during three consecutive rounds

As indicated, the processes execute a sequence of synchronous rounds. The code of the algorithm consists in the description of three consecutive rounds, namely the rounds  $r$ ,  $r + 1$ , and  $r + 2$ , where  $r = 1, 4, 7, 10, \dots$ . The messages exchanged during these three consecutive rounds are depicted in Fig. 2.13.

The behavior of the synchronous algorithm during these three consecutive rounds is as follows:

- Round  $r$ : lines 2–6.
 

Each process  $p_i$  invokes first the function `random()` to obtain a random number (line 2) that it sends to all its neighbors it is still communicating with (line 3). Then, it stores all the random numbers it has received, each coming from a process in `com_withi`.
- Round  $r + 1$ : lines 7–11.
 

Then,  $p_i$  sends the message `SELECTED(yes)` to its neighbors in `com_withi` if its random number is smaller than theirs (line 8). In this case, it progresses to the local state `in` and stops (line 9).

Otherwise, its random number is not the smallest. In this case,  $p_i$  first sends the message `SELECTED(no)` to its neighbors in `com_withi` (line 10), and then waits for a message from each of these neighbors (line 11).
- Round  $r + 2$ : lines 12–26.
 

Finally, if  $p_i$  has not entered the maximal independent set under construction, it checks if one of its neighbors in `com_withi` has been added to this set (line 12).

  - If one of its neighbors has been added to the independent set,  $p_i$  cannot be added to this set in the future. It consequently sends the message `ELIMINATED(yes)` to its neighbors in `com_withi` to inform them that it no longer competes to enter the independent set (line 13). In that case, it also enters the local state `out` and returns it (line 16).
  - If none of its neighbors in `com_withi` has been added to the independent set,  $p_i$  sends them the message `ELIMINATED(no)` to inform them that it is still competing to enter the independent set (line 17). Then, it waits for a message `ELIMINATED(–)` from each of them (line 18) and suppresses from the set `com_withi` its neighbors that are no longer competing (those are the processes which sent it the message `ELIMINATED(yes)`, lines 21–23).

Finally,  $p_i$  checks if `com_withi = ∅`. If it is the case, it enters the independent set and returns (line 24). Otherwise, it proceeds to the next round.

The algorithm computes an independent set because when a process is added to the set, all its neighbors stop competing to be in the set (lines 12–15). This set is maximal because when a process enters the independent set, only its neighbors are eliminated from being candidates.

### Why to Use Random Numbers Instead of Initial Names or Precomputed Colors

As we have seen, the previous algorithm associates a new random number with each process when this process starts a new round triple. The reader can check that the algorithm works if each process uses its identity or a legal color instead of a new random number at each round. Hence, the question: Why use random numbers?

The instance of the algorithm using  $n$  distinct identities (or a legal process  $m$ -coloring) requires a number of round triples upper bounded by  $\lceil n/2 \rceil$  (or  $\lceil m/2 \rceil$ ). This is because, in each round triple, at least one process enters the maximal independent set and at least one process is eliminated. Taking random numbers does not reduce this upper bound (because always taking initial identities corresponds to particular random choices) but reduces it drastically in the average case (the expected number of round triples is then  $O(\log_2 n)$ ).

## 2.3 Knot and Cycle Detection

Knots and cycles are graph patterns encountered when one has to solve distributed computing problems such as deadlock detection. This section presents an asynchronous distributed algorithm that detects such graph patterns.

### 2.3.1 Directed Graph, Knot, and Cycle

A directed graph is a graph where every edge is oriented from one vertex to another vertex. A directed path in a directed graph is a sequence of vertices  $i_1, i_2, \dots, i_x$  such that for any  $y, 1 \leq y < x$ , there is an edge from the vertex  $i_y$  to the vertex  $i_{y+1}$ . A cycle is a directed path such that  $i_x = i_1$ .

A knot in a directed graph  $G$  is a subgraph  $G'$  such that (a) any pair of vertices in  $G'$  belongs to a cycle and (b) there is no directed path from a vertex in  $G'$  to a vertex which is not in  $G'$ . Hence, a vertex of a directed graph belongs to a knot if and only if it is reachable from all the vertices that are reachable from it. Intuitively, a knot is a “black hole”: once in a knot, there is no way to go outside of it.

An example is given in Fig. 2.14. The directed graph has 11 vertices. The set of vertices  $\{7, 10, 11\}$  defines a cycle which is not in a knot (this is because, when traveling on this cycle, it is possible to exit from it). The subgraph restricted to the vertices  $\{3, 5, 6, 8, 9\}$  is a knot (after entering this set of vertices, it is impossible to exit from it).

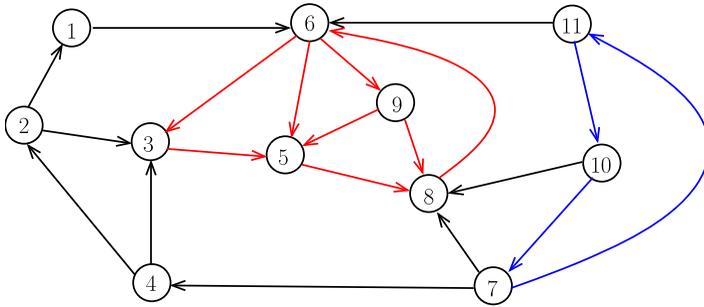


Fig. 2.14 A directed graph with a knot

### 2.3.2 Communication Graph, Logical Directed Graph, and Reachability

As previously, the underlying communication graph is not directed. Each channel is bidirectional which means that, if two processes are neighbors, either of them can send messages to the other.

It is assumed that a directed graph is defined on the communication graph. Its vertices are the processes, and if  $p_i$  and  $p_j$  are connected by a communication channel, there is (a) either a logical directed edge from  $p_i$  to  $p_j$ , or (b) a logical directed edge from  $p_j$  to  $p_i$ , or (c) two logical directed edges (one in each direction).

If there is a directed edge from  $p_i$  to  $p_j$ , we say “ $p_j$  is an immediate successor of  $p_i$ ” and “ $p_i$  is an immediate predecessor of  $p_j$ ”. A vertex  $p_j$  is said to be *reachable* from a vertex  $p_i$  if there is a directed path from  $p_i$  to  $p_j$ .

From an application point of view, a directed edge corresponds to a dependence relation linking a process  $p_i$  to its neighbor  $p_j$  (e.g.,  $p_i$  is waiting for “something” from  $p_j$ ).

### 2.3.3 Specification of the Knot Detection Problem

The problem consists in detecting if a given process belongs to a knot of a directed graph. For simplicity, it is assumed that only one process initiates the knot detection. Multiple instantiations can be distinguished by associating with each of them an identification pair made up of a process identity and a sequence number.

The knot detection problem is defined by the following properties, where  $p_a$  is the process that initiates the detection:

- Liveness (termination). If  $p_a$  starts the knot detection algorithm, it eventually obtains an answer.

- Safety (consistency).
  - If  $p_a$  obtains the answer “knot”, it belongs to a knot. Moreover, it knows the identity of all the processes involved in the knot.
  - If  $p_a$  obtains the answer “no knot”, it does not belong to a knot. Moreover, if it belongs to at least one cycle,  $p_a$  knows the identity of all the processes that are involved in a cycle with  $p_a$ .

As we can see, the safety property of the knot detection problem states what is a correct result while its liveness property states that eventually a result has to be computed.

### 2.3.4 Principle of the Knot/Cycle Detection Algorithm

The algorithm that is presented below relies on the construction of a spanning tree enriched with appropriate statements. It is due to D. Manivannan and M. Singhal (2003).

**Build a Directed Spanning Tree** To determine if it belongs to a knot, the initiator  $p_a$  needs to check that every process that is reachable from it is on a cycle which includes it ( $p_a$ ).

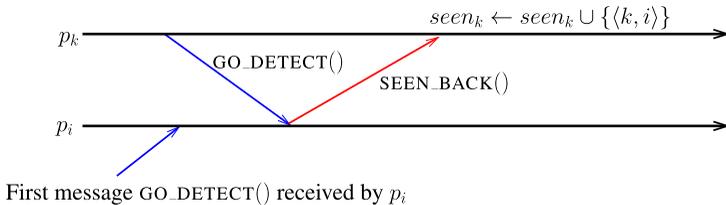
To that end,  $p_a$  sends a message `GO_DETECT()` to its immediate successors in the directed graph and these messages are propagated from immediate successors to immediate successors along directed edges to all the processes that are reachable from  $p_a$ . The first time it receives such a message from a process  $p_j$ , the receiver process  $p_i$  defines  $p_j$  as its parent in the directed spanning tree.

**Remark** The previous message `GO_DETECT()` and the messages `CYCLE_BACK()`, `SEEN_BACK()`, and `PARENT_BACK()` introduced below are nothing more than particular instances of the messages `GO()` and `BACK()` used in the graph traversal algorithms described in Chap. 1.

**How to Determine Efficiently that  $p_a$  Is on a Cycle** If the initiator  $p_a$  receives a message `GO_DETECT()` from a process  $p_j$ , it knows that it is on a cycle. The issue is then for  $p_a$  to know which are the processes involved in the cycles to which it belongs.

To that end,  $p_a$  sends a message `CYCLE_BACK()` to  $p_j$  and, more generally when a process  $p_i$  knows that it is on a cycle including  $p_a$ , it will send a message `CYCLE_BACK()` to each process from which it receives a message `GO_DETECT()` thereafter. Hence, these processes will learn that they are on a cycle including the initiator  $p_a$ .

But it is possible that, after it has received a first message `GO_DETECT()`, a process  $p_i$  receives more `GO_DETECT()` messages from other immediate predecessors (let  $p_k$  be one of them, see Fig. 2.15). If this message exchange pattern occurs,  $p_i$



**Fig. 2.15** Possible message pattern during a knot detection

sends back to  $p_k$  the message `SEEN_BACK()`, and when  $p_k$  receives this message it includes the ordered pair  $\langle k, i \rangle$  in a local set denoted  $seen_k$ . (Basically, the message `SEEN_BACK()` informs its receiver that its sender has already received a message `GO_DETECT()`.) In that way, if later  $p_i$  is found to be on a cycle including  $p_a$ , it can be concluded from the pair  $\langle k, i \rangle \in seen_k$  that  $p_k$  is also on a cycle including  $p_a$  (this is because, due to the messages `GO_DETECT()`, there is a directed path from  $p_a$  to  $p_k$  and  $p_i$ , and due to the cycle involving  $p_a$  and  $p_i$ , there is a directed path from  $p_i$  to  $p_a$ ).

Finally, as in graph traversal algorithms, when it has received an acknowledgment from each of its immediate successors, a process  $p_i$  sends a message `PARENT_BACK()` to its parent in the spanning tree. Such a message contains (a) the processes that, due to the messages `CYCLE_BACK()` received by  $p_i$  from immediate successors, are known by  $p_i$  to be on a cycle including  $p_a$ , and (b) the ordered pairs  $\langle i, \ell \rangle$  stored in  $seen_i$  as a result of the acknowledgment messages `SEEN_BACK()` and `PARENT_BACK()` it has received from its immediate successors in the logical directed graph. This information, which will be propagated in the tree to  $p_a$ , will allow  $p_a$  to determine if it is in a knot or a cycle.

### 2.3.5 Local Variables

**Local Variable at the Initiator  $p_a$  Only** The local variable  $candidates_a$ , which appears only at the initiator, is a set (initially empty) of process identities. If  $p_a$  is in a knot,  $candidates_a$  will contain the identities of all the processes that are in the knot including  $p_a$ , when the algorithm terminates. If  $p_a$  is not in a knot,  $candidates_a$  will contain all the processes that are in a cycle including  $p_a$  (if any). If  $candidates_a = \emptyset$  when the algorithm terminates,  $p_a$  belongs to neither a knot, nor a cycle.

**Local Variables at Each Process  $p_i$**  Each (initiator or not) process  $p_i$  manages the following four local variables.

- The local variable  $parent_i$  is initialized to  $\perp$ . If  $p_i$  is the initiator we will have  $parent_i = i$  when it starts the detection algorithm. If  $p_i$  is not the initiator,  $parent_i$  will contain the identity of the process from which the first message `GO_DETECT()` was received by  $p_i$ . When all the processes reachable from  $p_a$

have received a message `GO_DETECT()`, these local variables define a directed spanning tree rooted at  $p_a$  which will be used to transmit information back to this process.

- The local variable  $waiting\_from_i$  is a set of process identities. It is initialized to set of the immediate successors of  $p_i$  in the logical directed graph.
- The local variable  $in\_cycle_i$  is a set (initially empty) of process identities. It will contain processes that are on a cycle including  $p_i$ .
- The local variable  $seen_i$  is a set (initially empty) of ordered pairs of process identities. As we have seen,  $\langle k, j \rangle \in seen_i$  means that there is a directed path from  $p_a$  to  $p_k$  and a directed edge from  $p_k$  to  $p_j$  in the directed graph. It also means that both  $p_k$  and  $p_j$  have received a message `GO_DETECT()` and, when  $p_j$  received the message `GO_DETECT()` from  $p_k$ , it did not know whether it belongs to a cycle including  $p_a$  (see Fig. 2.15).

### 2.3.6 Behavior of a Process

The knot detection algorithm is described in Fig. 2.16.

**Launching the Algorithm** The only process  $p_i$  that receives the external message `START()` discovers that it is the initiator, i.e.,  $p_i$  is  $p_a$ . If it has no outgoing edges – predicate ( $waiting\_from_i \neq \emptyset$ ) at line 1 –,  $p_i$  returns the pair `(no knot,  $\emptyset$ )`, which indicates that  $p_i$  belongs neither to a cycle, nor to a knot (line 4). Otherwise, it sends the message `GO_DETECT()` to all its immediate successors in the directed graph (line 3).

**Reception of a Message `GO_DETECT()`** When a process  $p_i$  receives the message `GO_DETECT()` from  $p_j$ , it sends back to  $p_j$  the message `CYCLE_BACK()` if it is the initiator, i.e., if  $p_i = p_a$  (line 7). If it is not the initiator and this message is the first it receives, it first defines  $p_j$  as its parent in the spanning tree (line 9). Then, if  $waiting\_from_i \neq \emptyset$  (line 10),  $p_i$  propagates the detection to its immediate successors in the directed graph (line 11). If  $waiting\_from_i = \emptyset$ ,  $p_i$  has no successor in the directed graph. It then returns the message `PARENT_BACK( $seen_i, in\_cycle_i$ )` to its parent (both  $seen_i$  and  $in\_cycle_i$  are then equal to their initial value, i.e.,  $\emptyset$ ;  $seen_i = \emptyset$  means that  $p_i$  has not seen another detection message, while  $in\_cycle_i = \emptyset$  means that  $p_i$  is not involved in a cycle including the initiator).

If  $p_i$  is already in the detection tree, it sends back to  $p_j$  the message `SEEN_BACK()` or `CYCLE_BACK()` according to whether the local set  $in\_cycle_i$  is empty or not (line 14–15). Hence, if  $in\_cycle_i \neq \emptyset$ ,  $p_i$  is on a cycle including  $p_a$  and  $p_j$  will consequently learn that it is also on a cycle including  $p_a$ .

**Reception of a Message `XXX_BACK()`** When a process  $p_i$  receives a message `XXX_BACK()` (where XXX stands for SEEN, CYCLE, or PARENT), it first suppresses its sender  $p_j$  from  $waiting\_from_i$ .

```

when START() is received do
(1) if ( $waiting\_from_i \neq \emptyset$ )
(2)   then  $parent_i \leftarrow i$ ;
(3)     for each  $j \in waiting\_from_i$  do send GO_DETECT() to  $p_j$  end for
(4)   else return(no knot,  $\emptyset$ )
(5) end if.

when GO_DETECT() is received from  $p_j$  do
(6) if ( $parent_i = i$ )
(7)   then send CYCLE_BACK() to  $p_j$ 
(8)   else if ( $parent_i = \perp$ )
(9)     then  $parent_i \leftarrow j$ ;
(10)    if ( $waiting\_from_i \neq \emptyset$ )
(11)      then for each  $k \in waiting\_from_i$  do send GO_DETECT() to  $p_k$  end for
(12)      else send PARENT_BACK( $seen_i, in\_cycle_i$ ) to  $p_{parent_i}$ 
(13)      end if
(14)    else if ( $in\_cycle_i \neq \emptyset$ ) then send CYCLE_BACK() to  $p_j$ 
(15)      else send SEEN_BACK() to  $p_j$ 
(16)    end if
(17)  end if
(18) end if.

when SEEN_BACK() is received from  $p_j$  do
(19)  $waiting\_from_i \leftarrow waiting\_from_i \setminus \{j\}$ ;  $seen_i \leftarrow seen_i \cup \{(i, j)\}$ ; check_waiting_from().

when CYCLE_BACK() is received from  $p_j$  do
(20)  $waiting\_from_i \leftarrow waiting\_from_i \setminus \{j\}$ ;  $in\_cycle_i \leftarrow in\_cycle_i \cup \{j\}$ ;
(21) check_waiting_from().

when PARENT_BACK( $seen, in\_cycle$ ) is received from  $p_j$  do
(22)  $waiting\_from_i \leftarrow waiting\_from_i \setminus \{j\}$ ;  $seen_i \leftarrow seen_i \cup seen$ ;
(23) if ( $in\_cycle = \emptyset$ )
(24)   then  $seen_i \leftarrow seen_i \cup \{(i, j)\}$ 
(25)   else  $in\_cycle_i \leftarrow in\_cycle_i \cup in\_cycle$ 
(26) end if;
(27) check_waiting_from().

internal operation check_waiting_from() is
(28) if ( $waiting\_from_i = \emptyset$ ) then
(29)   if ( $parent_i = i$ )
(30)     then for each  $k \in in\_cycle_i$  do
(31)        $in\_cycle_i \leftarrow in\_cycle_i \setminus \{k\}$ ;  $candidates_i \leftarrow candidates_i \cup \{k\}$ ;
(32)       for each  $x \in \{1, \dots, n\}$  do
(33)         if ( $(x, k) \in seen_i$ )
(34)           then  $in\_cycle_i \leftarrow in\_cycle_i \cup \{x\}$ ;
(35)            $seen_i \leftarrow seen_i \setminus \{(x, k)\}$ 
(36)         end if
(37)       end for
(38)     end for;
(39)     if ( $seen_i = \emptyset$ ) then  $res \leftarrow$  knot else  $res \leftarrow$  no knot end if;
(40)     return( $res, candidates_i$ )
(41)   else if ( $in\_cycle_i \neq \emptyset$ ) then  $in\_cycle_i \leftarrow in\_cycle_i \cup \{i\}$  end if;
(42)   send PARENT_BACK( $seen_i, in\_cycle_i$ ) to  $p_{parent_i}$ ; return()
(43) end if
(44) end if.

```

Fig. 2.16 Asynchronous knot detection (code of  $p_i$ )

As we have seen, a message `SEEN_BACK()` informs its receiver  $p_i$  that its sender  $p_j$  has already been visited by the detection algorithm (see Fig. 2.15). Hence,  $p_i$  adds the ordered pair  $\langle i, j \rangle$  to  $seen_i$  (line 19). Therefore, if later  $p_j$  is found to be on a cycle involving the initiator, the initiator will be able to conclude from  $seen_i$  that  $p_i$  is also on a cycle involving  $p_a$ . The receiver  $p_i$  then invokes the internal operation `check_waiting_from()`.

If the message received by  $p_i$  from  $p_j$  is `CYCLE_BACK()`,  $p_i$  adds  $j$  to  $in\_cycle_i$  (line 20) before invoking `check_waiting_from()`. This is because there is a path from the initiator to  $p_i$  and a path from  $p_j$  to the initiator, hence  $p_i$  and  $p_j$  belong to a same cycle including  $p_a$ .

If the message received by  $p_i$  from  $p_j$  is `PARENT_BACK(seen, in_cycle)`,  $p_i$  adds the ordered pairs contained in  $seen$  sent by its child  $p_j$  to its set  $seen_i$  (line 22). Moreover, if  $in\_cycle$  is not empty,  $p_i$  merges it with  $in\_cycle_i$  (line 25). Otherwise  $p_i$  adds the ordered pair  $\langle i, j \rangle$  to  $seen_i$  (line 24). In this way, the information allowing  $p_a$  to know (a) if it is in a knot or (b) if it is only in a cycle involving  $p_i$  will be propagated from  $p_i$  first to its parent, and then propagated from its parent until  $p_a$ . Finally,  $p_i$  invokes `check_waiting_from()`.

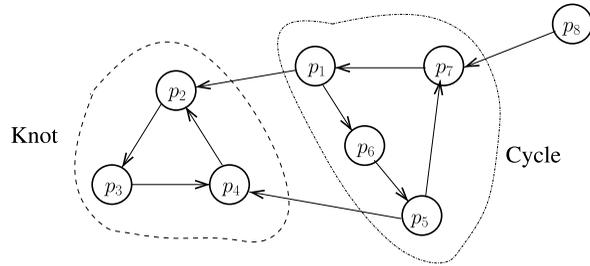
**The Internal Operation `check_waiting_from()`** As just seen, this operation is invoked each time  $p_i$  receives a message `XXX_BACK()`. Its body is executed only if  $p_i$  has received a message `XXX_BACK()` from each of its immediate successors (line 28). There are two cases.

If  $p_i$  is not the initiator, it first adds itself to  $in\_cycle_i$  if this set is not empty (line 41). This is because, if  $in\_cycle_i \neq \emptyset$ ,  $p_i$  knows that it is on a cycle involving the initiator (lines 20 and 25). Then,  $p_i$  sends to its parent (whose identity has been saved in  $parent_i$  at line 9) the information it knows on cycles involving the initiator. This information has been incrementally stored in its local variables  $seen_i$  and  $in\_cycle_i$  at lines 19–27. Finally,  $p_i$  invokes `return()`, which terminates its participation (line 42).

If  $p_i$  is the initiator  $p_a$ , it executes the statements of lines 30–39. First  $p_i$  cleans its local variables  $seen_i$  and  $in\_cycle_i$  (lines 30–38). For each  $k \in in\_cycle_i$ ,  $p_i$  first moves  $k$  from  $in\_cycle_i$  to  $candidates_i$ . This is because, if  $p_i$  is in a knot, so are all the processes which are on a cycle including  $p_i$ . Then, for each  $x$ , if the ordered pair  $\langle x, k \rangle \in seen_i$ ,  $p_i$  suppresses it from  $seen_i$  and adds  $p_x$  to  $in\_cycle_i$ . This is because, after  $p_a$  has received a message `XXX_BACK()` from each of its immediate successors, we have for each process  $p_k$  reachable from  $p_a$  either  $k \in in\_cycle_a$  or  $\langle x, k \rangle \in seen_a$  for some  $p_x$  reachable from  $p_a$ . Hence, if  $k \in in\_cycle_a$  and  $\langle x, k \rangle \in seen_a$ , then  $p_x$  is also in a cycle with  $p_a$ .

Therefore, after the execution of lines 30–38,  $candidates_a$  contains the identities of all the processes reachable from  $p_a$  which are on a cycle with  $p_a$ . It follows that, if  $seen_a$  becomes empty, all the processes reachable from  $p_a$  are on a cycle with  $p_a$ . The statement of line 39 is a direct consequence of this observation. If  $seen_a = \emptyset$ ,  $p_a$  belongs to a knot made up of the processes which belong to the set  $candidates_s$ . If  $seen_a \neq \emptyset$ ,  $candidates_a$  contains all the processes that are involved in a cycle including  $p_a$  (hence if  $candidates_a = \emptyset$ ,  $p_i$  is involved neither in a knot, nor in a cycle).

**Fig. 2.17**  
Knot/cycle detection:  
example



**An Example** Let us consider the directed graph depicted in Fig. 2.17. This graph has a knot composed of the processes  $p_2$ ,  $p_3$ , and  $p_4$ , a cycle involving the processes  $p_1$ ,  $p_6$ ,  $p_5$  and  $p_7$ , plus another process  $p_8$ . If the initiator process  $p_a$  belongs to the knot,  $p_a$  will discover that it is in a knot, and we will have  $candidates_a = \{2, 3, 5\}$  and  $seen_a = \emptyset$  when the algorithm terminates. If the initiator process  $p_a$  belongs to the cycle on the right of the figure (e.g.,  $p_a$  is  $p_1$ ), we will have  $candidates_a = \{1, 6, 5, 7\}$  and  $seen_a = \{\langle 4, 2 \rangle, \langle 3, 4 \rangle, \langle 2, 3 \rangle, \langle 1, 2 \rangle, \langle 5, 4 \rangle\}$  when the algorithm terminates (assuming that the messages `GO_DETECT()` propagate first along the process chain  $(p_1, p_2, p_3, p_4)$ , and only then from  $p_5$  to  $p_4$ ).

**Cost of the Algorithm** As in a graph traversal algorithm, each edge of the directed graph is traversed at most once by a message `GO_DETECT()` and a message `SEEN_BACK()`, `CYCLE_BACK()` or `PARENT_BACK()` is sent in the opposite direction. It follows that the number of message used by the algorithm is upper bounded by  $2e$ , where  $e$  is the number of edges of the logical directed graph.

Let  $D_T$  be the depth of the spanning tree rooted at  $p_a$  that is built. It is easy to see that the time complexity is  $2(D_T + 1)$  ( $D_T$  time units for the messages `GO_DETECT()` to go from the root  $p_a$  to the leaves,  $D_T$  time units for the messages `XXX_BACK()` to go back in the other direction and 2 more time units for the leaves to propagate the message `GO_DETECT()` to their immediate successors and obtain their acknowledgment messages `XXX_BACK()`).

## 2.4 Summary

Considering a distributed system as a graph whose vertices are the processes and edges are the communication channels, this chapter has presented several distributed graph algorithms. “Distributed” means here each process cooperates with its neighbors to solve a problem but never learns the whole graph structure it is part of.

The problems that have been addressed concern the computation of shortest paths, the coloring of the vertices of a graph in  $\Delta + 1$  colors (where  $\Delta$  is the maximal degree of the vertices), the computation of a maximal independent set, and the detection of knots and cycles.

As the reader has seen, the algorithmic techniques used to solve graph problems in a distributed context are different from their sequential counterparts.

## 2.5 Bibliographic Notes

- Graph notions and sequential graph algorithms are described in many textbooks, e.g., [122, 158]. Advanced results on graph theory can be found in [164]. Time complexity results of numerous graph problems are presented in [148].
- Distributed graph algorithms and associated time and message complexity analyses can be found in [219, 292].
- As indicated by its name, the sequential shortest path algorithm presented in Sect. 2.1.1 is due to R.L. Ford. It is based on Bellman’s dynamic programming principle [44]. Similarly, the sequential shortest path algorithm presented in Sect. 2.1.2 is due to R.W. Floyd and R. Warshall who introduced independently similar algorithms in [128] and [384], respectively. The adaptation of Floyd–Warshall’s shortest path algorithm is due to S. Toueg [373].  
Other distributed shortest path algorithms can be found in [77, 203].
- The random algorithm presented in Sect. 2.2.3, which computes a maximal independent set, is due to M. Luby [240]. The reader will find in this paper a proof that the expected number of rounds is  $O(\log_2 n)$ . Another complexity analysis of  $(\Delta + 1)$ -coloring is presented in [201].
- The knot detection algorithm described in Fig. 2.3.4 is due to D. Manivannan and M. Singhal [248] (this paper contains a correctness proof of the algorithm). Other asynchronous distributed knot detection algorithms can be found in [59, 96, 264].
- Distributed algorithms for finding centers and medians in networks can be found in [210].
- Deterministic distributed vertex coloring in polylogarithmic time, suited to synchronous systems, is addressed in [43].

## 2.6 Exercises and Problems

1. Adapt the algorithms described in Sect. 2.1.1 to the case where the communication channels are unidirectional in the sense that a channel transmits messages in one direction only.
2. Execute Luby’s maximal independent set described in Fig. 2.12 on both graphs described in Fig. 2.10 with various values output by the function `random()`.
3. Let us consider the instances of Luby’s algorithm where, for each process, the random numbers are statically replaced by its initial identity or its color (where no two neighbor processes have the same color).  
Compare these two instances. Do they always have the same time complexity?
4. Adapt Luby’s synchronous maximal independent set algorithm to an asynchronous message-passing system.
5. Considering the directed graph depicted in Fig. 2.14, execute the knot detection algorithm described in Sect. 2.3.4 (a) when  $p_5$  launches the algorithm, (b) when  $p_{10}$  launches the algorithm, and (c) when  $p_4$  launches the algorithm.

# Chapter 3

## An Algorithmic Framework to Compute Global Functions on a Process Graph

This chapter is devoted to distributed graph algorithms that compute a function or a predicate whose inputs are disseminated at the processes of a network. The function (or the predicate) is global because the output at each process depends on the inputs at all the processes. It follows that the processes have to communicate in order to compute their results.

A general algorithmic framework is presented which allows global functions to be computed. This distributed framework is (a) symmetric in the sense that all processes obey the same rules of behavior, and (b) does not require the processes to exchange more information than needed. The computation of shortest distances and the determination of a cut vertex in a graph are used to illustrate the framework. The framework is then improved to allow for a reduction of the size and the number of messages that are exchanged. Finally, the chapter analyzes the particular case of regular networks (networks in which all the processes have the same number of neighbors).

**Keywords** Cut vertex · De Bruijn’s graph · Determination of cut vertices · Global function · Message filtering · Regular communication graph · Round-based framework

### 3.1 Distributed Computation of Global Functions

#### 3.1.1 Type of Global Functions

The problems addressed in this chapter consist for each process  $p_i$  in computing a result  $out_i$  which involves the whole set of inputs  $in_1, in_2, \dots, in_n$ , where  $in_i$  is the input provided by process  $p_i$ . More precisely, let  $IN[1..n]$  be the vector that, from an external observer point of view, represents the inputs, i.e.,  $\forall i: IN[i] = in_i$ . Similarly, let  $OUT[1..n]$  be the vector such that  $\forall i: OUT[i] = out_i$ . Hence, the processes have to cooperate and coordinate so that they collectively compute

$$OUT = F(IN).$$

According to the function  $F()$  which is computed, all the processes may obtain the same result  $out$  (and we have then  $OUT[1] = \dots = OUT[n] = out$ ) or different

results (i.e., it is possible that there are processes  $p_i$  and  $p_j$  such that  $out_i \neq out_j$ ). Examples of such problems are the following.

**Routing Tables** In this case, the input of a process is its position in the network and its output is a routing table that, for each process  $p_j$ , defines the local channel that  $p_i$  has to use so that the messages to  $p_j$  travel along the path with the shortest distance (let us recall that the distance is the minimal number of channels separating  $p_i$  from  $p_j$ ).

**Eccentricity, Diameter, Radius, Center, and Peripheral Vertex** The eccentricity of a process  $p_i$  (vertex) in a communication graph is the longest distance from  $p_i$  to any other process. The eccentricity of  $p_i$  is denoted  $ecc_i$ .

The diameter  $D$  of a graph is its largest eccentricity:  $D = \max_{1 \leq i \leq n} (ecc_i)$ . The radius of a graph is its minimal eccentricity, a center of a graph is a vertex (process) whose eccentricity is equal to the radius, while a peripheral vertex (process) is a vertex (process) whose eccentricity is equal to the diameter.

The computation of the diameter (or the radius) of a communication graph corresponds to a function that provides the same parameter to all the processes. Differently, the computation of its eccentricity by each process corresponds to a function that does not necessarily provide the processes with the same result. The same occurs when each process has to know if it is a center (or a peripheral process) of the communication graph.

**Maximal or Minimal Input** Simple functions returning the same result to all the processes are the computation of the maximum (or minimum) of their inputs. This is typically the case in the election problem where, assuming that processes have distinct and comparable identities, the input of each process is its identity and their common output is the greatest (or smallest) of their identities, which defines the process that is elected.

**Cut Vertex** A cut vertex in a graph is a vertex (process) whose suppression disconnects the graph. Knowledge of such processes is important to analyze message bottleneck and fault-tolerance. The global function computed by each process is here a predicate indicating, at each process, if it is a cut vertex.

### 3.1.2 Constraints on the Computation

**On the Symmetry Side: No Centralized Control** A simple solution to compute a function  $F()$  would be to use a broadcast/convergecast as presented in Sect. 1.2.1. The process  $p_a$ , which is the root of the spanning tree, broadcasts a request message  $GO()$  and waits until it has received response messages  $BACK()$  from each of its children (each of these messages carries input values from a subset of processes, and they all collectively carry all the input values except the one of  $p_a$ ). The root process

$p_a$  then knows the input vector  $IN[1..n]$ . It can consequently compute  $OUT[1..n] = F(IN)$  with the help of a sequential algorithm, and returns its result to each process along the channels of the spanning tree.

While this solution is viable and worthwhile for some problems, we are interested in this chapter in a solution in which the control is distributed, that is to say in a solution in which no process plays a special role. This can be expressed in the form of the following constraint: the processes must obey the same rules of behavior.

**On the Efficiency Side: Do Not Learn More than What Is Necessary** A solution satisfying the previous constraint would be for each process to flood the system with its input so that all the processes learn all the inputs, i.e., the input vector  $IN[1..n]$ . Then, each process  $p_i$  can compute the output vector  $OUT[1..n]$  and extracts from it its local result  $out_i = OUT[i]$ .

As an example, if the processes have to compute the shortest distances, they could first use the algorithm described in Fig. 1.3 (Sect. 1.1.2) in order to learn the communication graph. They could then use any sequential shortest path algorithm to compute their shortest distances. This is not satisfactory when a process  $p_i$  learns more information than what is needed to compute its table of shortest distances. As we have seen in Sect. 2.1, it is not necessary for it to know the whole structure of the communication graph in which it is working.

The solutions in which we are interested are thus characterized by a second constraint: a process has not to learn information which is useless from the point of view of the local output it has to compute.

## 3.2 An Algorithmic Framework

This section presents an algorithmic framework which offers a solution for the class of distributed computations which have been described previously, while respecting the symmetry and efficiency constraints that have been stated. This framework is due to J.-Cl. Bermond, J.-Cl. König, and M. Raynal (1987).

### 3.2.1 A Round-Based Framework

The algorithmic framework is based on rounds. Moreover, to simplify the presentation, we consider that the communication channels are FIFO (if they are not—as we have seen in Fig. 2.9, Sect. 2.2.2—each message has to carry a parity bit defined from the current round of its sender).

Each process executes first an initialization part, followed by an asynchronous sequence of rounds. If a process receives a message before it started its participation in the algorithm, it executes its initialization part and starts its first round before processing the message.

**Asynchronous Round-Based Computation** Differently from synchronous systems, the rounds are not given for free in an asynchronous system. Each process  $p_i$  has to handle a local variable  $r_i$  which denotes its current round number.

We first consider that, in each round  $r$  it executes, a process sends a message to each of its neighbors, and receives a message from each of them. (This assumption on the communication pattern will be relaxed in the next section.) It then executes a local processing which depends on the particular problem that is solved.

**Local Variables at Each Process  $p_i$**  In addition to its local variable  $r_i$  and its identity  $id_i$ , a process manages the following local variables:

- As already indicated,  $in_i$  is the input parameter provided by  $p_i$  to solve the problem, while  $out_i$  is a local variable that will contain its local output.
- A process  $p_i$  has  $c_i$  ( $1 \leq c_i \leq n - 1$ ) bidirectional channels, which connect it to  $c_i$  distinct neighbors processes. The set  $channels_i = \{1, \dots, c_i\}$  is the set of local indexes that allow these neighbors to be addressed, and the local array  $channel_i[1..c_i]$  defines the local names of these channels (see Exercise 4, Chap. 1). This means that the local names of the channel (if any) connecting  $p_i$  and  $p_j$  are  $channel_i[x]$  (where  $x \in channels_i$ ) at  $p_i$  and  $channel_j[y]$  (where  $y \in channels_j$ ) at  $p_j$ .
- $new_i$  is a local set that, at the end of each round  $r$ , contains all the information that  $p_i$  learned during this round (i.e., it receives this information at round  $r$  for the first time).
- $inf_i$  is a local set that contains all the information that  $p_i$  has learned since the beginning of the execution.

**Principle** The underlying principle is nothing more than the forward/discard principle. During a round  $r$ , a process sends to its neighbors all the new information it has received during the previous round ( $r - 1$ ). It is assumed that it has received its input during the fictitious round  $r = 0$ .

It follows that, during the first round a process learns the inputs of the processes which are distance 1 from it, during the second round it learns the inputs of the processes at distance 2, and more generally it learns the inputs of the processes at distance  $d$  during the round  $r = d$ .

**Example: Computation of the Routing Tables** As a first example, let us consider the computation of the routing tables at each process  $p_i$ . Here  $in_i$  is the identity of  $p_i$  ( $id_i$ ), and it is assumed that no two processes have the same identity. As far as the output is concerned,  $out_i$  is here a routing table  $out_i[1..c_i]$ , where each  $out_i[x]$  is a set initially empty. At the end of the algorithm,  $out_i[x]$  will contain the identities of the processes  $p_j$  such that  $channel_i[x]$  is the channel that connects  $p_i$  to its neighbor on the shortest distance to  $p_j$ .

To simplify the presentation, the description of the algorithm assumes that the processes know the diameter  $D$  of the communication graph. It follows from the design principle of the algorithm that, when  $r_i = D$ , each process has learned all the

```

init  $inf_i \leftarrow \{id_i\}; new_i \leftarrow \{id_i\}; r_i \leftarrow 0.$ 

(1) for  $r_i$  from 1 to  $D$  do
    begin asynchronous round
(2)    $r_i \leftarrow r_i + 1;$ 
(3)   for each  $x \in channels_i$  do send MSG( $new_i$ ) on  $channel_i[x]$  end for;
(4)    $new_i \leftarrow \emptyset;$ 
(5)   for each  $x \in channels_i$  do
(6)     wait (MSG( $new$ ) received on  $channel_i[x]$ );
(7)     let  $aux = new \setminus (inf_i \cup new_i);$ 
(8)      $out_i[x] \leftarrow out_i[x] \cup aux;$ 
(9)      $new_i \leftarrow new_i \cup aux$ 
(10)  end for;
(11)   $inf_i \leftarrow inf_i \cup new_i$ 
    end asynchronous round
(12) end for;
(13) return( $out_i[1..c_i]$ ).

```

**Fig. 3.1** Computation of routing tables defined from distances (code for  $p_i$ )

information it can ever know. Consequently, if  $D$  is known,  $D$  rounds are necessary and sufficient to compute the routing tables.

The corresponding algorithm is described in Fig. 3.1. Each process  $p_i$  executes  $D$  rounds (line 1). During a round, it first sends to its neighbors what it has learned during the previous round (line 3), which has been saved in its local variable  $new_i$  (line 9; initially, it knows only its identity  $id_i$ ).

Then,  $p_i$  waits for a round  $r$  message from each of its neighbors. When it receives such a message on channel  $channel_i[x]$ , it first computes what it learns from this message (line 7). As already noticed, what it learns are identities of processes at distance  $r$  from it. It consequently adds these identities to the routing table  $out_i[x]$ . The update of  $out_i[x]$  (line 8) is such that  $channel_i[x]$  is the favorite channel to send messages to the processes whose identity belongs to  $out_i[x]$ . (If we want to use an array  $routing\_to_i[id]$ , as done in the previous chapter,  $id \in out_i[x]$  means that  $routing\_to_i[id] = channel_i[x]$ .) Then  $p_i$  updates appropriately its local variable  $new_i$  (line 9).

Finally, before proceeding to the next round,  $p_i$  updates  $inf_i$  which stores all the identities learned since the beginning (line 11). If it has executed the  $D$  rounds, it returns its local routing table  $out_i[1..n]$  (line 13).

**Cost** As previously, let one time unit be the maximal transit time for a message from a process to one of its neighbors. The time complexity is  $2D$ . The worst case is when a single process starts the algorithm. It then takes  $D$  messages sent sequentially from a process to another process before all processes have started their local algorithm. Then, all the processes execute  $D$  rounds.

Each process executes  $D$  rounds, and two messages are exchanged on each channel at each round. Hence, the total number of messages is  $2eD$ , where  $e$  is the number of channels of the communication graph.

**A General Algorithm** A general algorithm is easily obtained as follows. Let  $in_i$  the input of  $p_i$ . The initialization is then replaced by the statements “ $inf_i \leftarrow \{(i, in_i)\}; new \leftarrow \{(i, in_i)\}$ ”, and line 8 is modified according to the function or the predicate that one wants to compute.

### 3.2.2 When the Diameter Is Not Known

This section shows how to eliminate the a priori knowledge of the diameter.

**A Simple Predicate** Let us observe that, if a process does not learn any new information at a given round, it will learn no more during the next rounds.

This follows from the fact that, if  $p_i$  learns nothing new in round  $r$ , there is no process situated at distance  $r$  from  $p_i$ , and consequently no process at a distance greater than  $r$ . Hence, if it learns nothing new in round  $r$ ,  $p_i$  will learn nothing new during rounds  $r' > r$ .

Actually, the last round  $r$  during which  $p_i$  learns something new is the round  $r = ecc_i$  (its eccentricity), but, not knowing  $ecc_i$ , it does not know that it has learned everything. In contrast, at the end of round  $r = ecc_i + 1$ ,  $p_i$  will have  $new_i = \emptyset$  and it will then learn that it knows everything. (Let us observe that this predicate also allows  $p_i$  to easily compute  $ecc_i$ .)

**Not All Processes Terminate During the Same Round** Let us observe that, as not all the processes have necessarily the same eccentricity, they do not terminate at the same round. Let us consider two neighbor processes  $p_i$  and  $p_j$  such that  $p_i$  learns no new information during round  $r$  while  $p_j$  does. We have the following:

- As  $p_i$  and  $p_j$  are neighbors, we have  $0 \leq |ecc_i - ecc_j| \leq 1$ . Thus,  $p_j$  will have  $new_j = \emptyset$  at the end of round  $r + 1$ .
- In order that  $p_j$  does not block during round  $r + 1$  waiting for a message from  $p_i$ , this process has to execute round  $r + 1$ , sending it a message carrying the value  $new_i = \emptyset$ .

It follows that a process  $p_i$  now executes  $ecc_i + 2$  rounds. At round  $r = ecc_i$  it knows everything, at round  $r = ecc_i + 1$  it knows that it knows everything, and at round  $r = ecc_i + 2$  it informs its neighbors of this and terminates.

The corresponding generic algorithm is described in Fig. 3.2. Each process manages an additional local variable  $com\_with_i$ , which contains the channels on which it has to send and receive messages. The management of this variable is the main novelty with respect to the algorithm of Fig. 3.1.

If the empty set is received on a channel  $channel_i[x]$ , this channel is withdrawn from  $com\_with_i$  (line 7). Moreover, to prevent processes from being blocked waiting forever for a message, a process  $p_i$  (before it terminates) sends the message  $MSG(\emptyset)$  on each channel which is still open (line 15). It also empties the open channels by waiting for a message on each of them (line 16).

```

init  $inf_i \leftarrow \{(id_i, in_i)\}; new_i \leftarrow \{(id_i, in_i)\}; com\_with_i \leftarrow \{1, \dots, c_i\}; r_i \leftarrow 0.$ 

(1) while ( $new_i \neq \emptyset$ ) do
    begin asynchronous round
(2)    $r_i \leftarrow r_i + 1;$ 
(3)   for each  $x \in com\_with_i$  do send MSG( $new_i$ ) on  $channel_i[x]$  end for;
(4)    $new_i \leftarrow \emptyset;$ 
(5)   for each  $x \in com\_with_i$  do
(6)     wait (MSG( $new$ ) received on  $channel_i[x]$ );
(7)     if ( $new = \emptyset$ ) then  $com\_with_i \leftarrow com\_with_i \setminus \{x\}$ 
(8)       else let  $aux = new \setminus (inf_i \cup new_i);$ 
(9)         specific statements related to the function that is computed;
(10)         $new_i \leftarrow new_i \cup aux$ 
(11)     end if
(12)   end for;
(13)    $inf_i \leftarrow inf_i \cup new_i$ 
    end asynchronous round
(14) end while;
(15) for each  $x \in com\_with_i$  do send MSG( $new_i$ ) on  $channel_i[x]$  end for;
(16) for each  $x \in com\_with_i$  do wait (MSG( $new$ ) on  $channel_i[x]$ ) end for;
(17) specific statements which compute  $out_i$ ;
(18) return( $out_i$ ).

```

**Fig. 3.2** A diameter-independent generic algorithm (code for  $p_i$ )

**Cost** It is easy to see that the time complexity is upper-bounded by  $D + (D + 2) = 2D + 2$ , and the message complexity is  $O(2e(D + 2))$ .

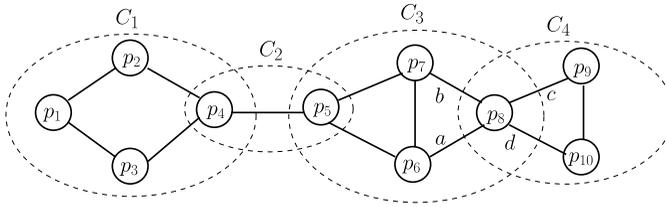
**A Round-Based Algorithm as an Iteration on a Set of Equations** The algorithm of Fig. 3.2 can be rephrased as a set of equations relating the local variables at round  $r - 1$  and round  $r$ . This makes the algorithm instantiated with a function  $F()$  easier to verify than an algorithm computing the same function  $F()$  which would not be based on rounds. These equations use the following notations:

- $sent_i[x]^r$  denotes the content of the message sent by  $p_i$  on  $channel_i[x]$  during round  $r$ ,
- $received_i[x]^r$  denotes the content of the message received by  $p_i$  on  $channel_i[x]$  during round  $r$ ; moreover,  $p_j$  is the process that sent this message on its local channel  $channel_j[y]$ ,
- $new_i^{r-1}$  is the content of  $new_i$  at the end of round  $r - 1$ ,
- $com\_with_i^{r-1}$  and  $inf_i^{r-1}$  are the values of  $com\_with_i$  and  $inf_i$  at the end of round  $r - 1$ , respectively.

Using these notations, the equations that characterize the generic round-based algorithm are the following ones for all  $i \in [1..n]$ :

$$\forall x \in com\_with_i^{r-1} : sent_i[x]^r = new_i^{r-1},$$

$$\forall x \in com\_with_i^{r-1} : received_i[x]^r = sent_j[y]^r,$$



**Fig. 3.3** A process graph with three cut vertices

$$\begin{aligned} \text{new}_i^r &= \left( \bigcup_{x \in \text{com\_with}_i^r} \text{received}_i[x]^r \right) \setminus \text{inf}_i^{r-1}, \\ \text{com\_with}_i^r &= \text{com\_with}_i^{r-1} \setminus \{x \mid \text{received}_i[x]^r = \emptyset\}. \end{aligned}$$

### 3.3 Distributed Determination of Cut Vertices

#### 3.3.1 Cut Vertices

**Definition** A *cut vertex* (or articulation point) of a graph is a vertex whose suppression disconnects the graph. A graph is *biconnected* if it remains connected after the deletion of any of its vertices. Thus, a connected communication graph is biconnected if and only if it has no cut vertex. A connected graph can be decomposed in a tree whose vertices are its biconnected components.

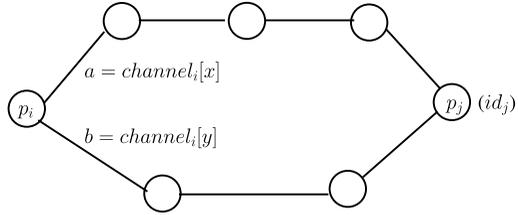
Given a vertex (process)  $p_i$  of a connected graph, let  $R_i$  be the local relation defined on the edges (channels) incident to  $p_i$  as follows. Let  $e_1$  and  $e_2$  be two edges incident to  $p_i$ ;  $e_1 R_i e_2$  if the edges  $e_1$  and  $e_2$  belong to the same biconnected component of the communication graph. It is easy to see that  $R_i$  is an equivalence relation; i.e., it is reflexive, symmetric, and transitive. Thus, if  $e_1 R_i e_2$  and  $e_2 R_i e_3$ , then, the three edges  $e_1$ ,  $e_2$ , and  $e_3$  incident to  $p_i$  belong to the same biconnected component.

**Example** As an example, let us consider the graph depicted in Fig. 3.3. The processes  $p_4$ ,  $p_5$ , and  $p_8$  are cut vertices (the deletion of any of them disconnects the graph).

There are four biconnected components (ellipsis on the figure) denoted  $C_1$ ,  $C_2$ ,  $C_3$ , and  $C_4$ . As an example, the deletion of any process (vertex) inside a component does not make it disconnected.

When looking at the four edges  $a$ ,  $b$ ,  $c$ , and  $d$  connecting process  $p_8$  to its neighbors, we have  $a R_8 b$  and  $c R_8 d$ , but we do not have  $a R_8 d$ . This is because the channels  $a$  and  $b$  belong to the biconnected component  $C_3$ , the channels  $c$  and  $d$  belong to the biconnected component  $C_4$ , and  $C_3 \cup C_4$  is not a biconnected component of the communication graph.

**Fig. 3.4**  
Determining cut vertices:  
principle



### 3.3.2 An Algorithm Determining Cut Vertices

**Principle of the Algorithm** The algorithm, which is due to J.-Cl. Bermond and J.-Cl. König (1991), is based on the following simple idea.

Given a process  $p_i$  which is on one or more cycles, let us consider an elementary cycle to which  $p_i$  belongs (an elementary cycle is a cycle that does not pass several times through the same vertex/process). Let  $a = channel_i[x]$  and  $b = channel_i[y]$  be the two distinct channels of  $p_i$  on this cycle (see an example in Fig. 3.4). Moreover, let  $p_j$  be the process on this cycle with the greatest distance to  $p_i$ , and let  $\ell$  be this distance. Hence, the length of the elementary cycle including  $p_i$  and  $p_j$  is  $2\ell$  or  $2\ell + 1$  (in the figure, it is  $2\ell + 1 = 3 + 4$ ).

The principle that underlies the design of the algorithm follows directly from (a) the message exchange pattern of the generic framework, and (b) the fact that, during each round  $r$ , a process sends only the new information it has learned during the previous round. More precisely, we have the following.

- It follows from the message exchange pattern that  $p_i$  receives  $id_j$  during round  $r = \ell$  (in the figure,  $id_j$  is learned from channel  $b$ ). Moreover,  $p_i$  also receives  $id_j$  from channel  $a$  during round  $r = \ell$  if the length of the cycle is even, or during round  $r = \ell + 1$  if the length of the cycle is odd. In the figure, as the length of the elementary cycle is odd,  $p_i$  receives  $id_j$  a first time on channel  $b$  during round  $\ell$  and a second time on channel  $a$  during round  $\ell + 1$ . The pair  $(a, b) \in R_i$ , i.e., the channels  $a$  and  $b$  (which are incident to  $p_i$ ) belong to the same biconnected component.

This observation provides a simple local predicate, which allows a process  $p_i$  to determine if two of its incident channels belong to the same biconnected component.

- Let us now consider two channels incident to the same process  $p_i$  that are not in the same biconnected component (as an example, this is the case of channels  $b$  and  $c$ , both incident to  $p_8$ ). As these two channels do not belong to a same elementary cycle including  $p_i$ , this process cannot receive two messages carrying the same process identity during the same round or two consecutive rounds. Thus, the previous predicate is an “if and only if” predicate.

**Description of the Algorithm** The algorithm is described in Fig. 3.5. This algorithm is an enrichment of the generic algorithm of Fig. 3.2: the only addition are the lines N.9\_1–N.9\_10, which replace line 9, and lines 17\_1–17\_3, which replace

```

init  $inf_i \leftarrow \{id_i\}$ ;  $new_i \leftarrow \{id_i\}$ ;  $com\_with_i \leftarrow \{1, \dots, c_i\}$ .

(1)   while ( $new_i \neq \emptyset$ ) do
      begin asynchronous round
(2)      $r_i \leftarrow r_i + 1$ ;
(3)     for each  $x \in com\_with_i$  do send MSG( $new_i$ ) on  $channel_i[x]$  end for;
(4)      $new_i \leftarrow \emptyset$ ;
(5)     for each  $x \in com\_with_i$  do
(6)       wait (MSG( $new$ ) received on  $channel_i[x]$ );
(7)       if ( $new = \emptyset$ ) then  $com\_with_i \leftarrow com\_with_i \setminus \{x\}$ 
(8)         else let  $aux = new \setminus (inf_i \cup new_i)$ ;
(N.9_1)         for each  $id \in new$  do
(N.9_2)           if ( $id \notin inf_i \cup new_i$ )
(N.9_3)             then  $routing\_to_i[id] \leftarrow x$ ;  $dist_i[id] \leftarrow r_i$ 
(N.9_4)             else if ( $r_i = dist_i[id] \vee r_i = dist_i[id] + 1$ )
(N.9_5)               then let  $y = routing\_to_i[id]$ ;
(N.9_6)                  $channel_i[y], channel_i[x]$ 
(N.9_7)                  $\in$  same biconnected component
(N.9_8)             end if
(N.9_9)           end if
(N.9_10)          end for;
(10)         $new_i \leftarrow new_i \cup aux$ 
(11)      end if
(12)    end for;
(13)     $inf_i \leftarrow inf_i \cup new_i$ 
      end asynchronous round
(14)  end while;
(15)  for each  $x \in com\_with_i$  do send MSG( $new_i$ ) on  $channel_i[x]$  end for;
(16)  for each  $x \in com\_with_i$  do wait (MSG( $new$ ) on  $channel_i[x]$ ) end for;
(N.17_1) Considering  $channel_i[1..c_i]$  and the pairs computed at line N.9-6, compute the
(N.17_2) transitive closure of the local relation  $R_i$  (belong to a same biconnected component);
(N.17_3) if (all channels of  $p_i$  belong to the same biconnected component)
(N.17_4)   then  $out_i \leftarrow no$  else  $out_i \leftarrow yes$ 
(N.17_5) end if;
(18)  return( $out_i$ ).

```

**Fig. 3.5** An algorithm determining the cut vertices (code for  $p_i$ )

line 17. Said differently, by suppressing all the lines whose number is prefixed by N, we obtain the algorithm of Fig. 3.2.

Each process  $p_i$  manages two array-like data structures, denoted  $routing\_to_i$  and  $dist_i$ . Given a process identity  $id$ ,  $dist_i[id]$  contains the distance from  $p_i$  to the process whose identity is  $id$ , and  $routing\_to_i[id]$  contains the local channel on which messages for this process have to be sent. We use an array-like structure to make the presentation clearer. Since initially a process  $p_i$  knows neither the number of processes, nor their identities, a dynamic list has to be used to implement  $routing\_to_i$  and  $dist_i$ .

Thus, let us consider a process  $p_i$  that, during a round  $r$ , receives on a channel  $channel_i[x]$  a message carrying the value  $new \neq \emptyset$ . Let  $id \in new$  (line N.9\_1) and let  $p_j$  be the corresponding process (i.e.,  $id = id_j$ ). There are two cases:

- If  $id$  is an identity not yet known by  $p_i$  (line N.9\_2),  $p_i$  creates the variables  $routing\_to_i[id]$  and  $dist_i[id]$  and assigns them  $x$  (the appropriate local channel to send message to  $p_j$ ) and  $r_i$  (the distance separating  $p_i$  and  $p_j$ ), respectively (line N.9\_3).
- If  $id$  is an identity known by  $p_i$ , it checks the biconnectivity local predicate, namely  $(r_i = dist_i[id]) \vee (r_i = dist_i[id] + 1)$  (line N.9\_4). If this predicate is true,  $p_i$  has received twice the same identity during the same round  $r_i$ , or during the rounds  $r_i$  and  $r_i - 1$ . As we have seen, this means that  $channel_i[x]$  (the channel on which the message  $MSG(new)$  has been received) and  $channel_i[y]$  (the channel on which  $id$  has been received for the first time, line N.9\_5), belong to the same biconnected component. Hence, we have  $(channel_i[x], channel_i[y]) \in R_i$  (line N.9\_6).

The second enrichment of the generic algorithm is the computation of the result at lines N.17\_1–N.17\_3. A process  $p_i$  computes the transitive closure of its relation  $R_i$ , which has been incrementally computed at line N.9\_6. Let  $R_i^*$  denote this transitive closure. If  $R_i^*$  is made up of a single class of equivalence, then  $p_i$  is not a cut vertex and the result is consequently the value `no`. If  $R_i^*$  contains two or more classes of equivalence, each class defines a distinct biconnected component to which  $p_i$  belongs. In this case,  $p_i$  is a cut vertex of the communication graph, and it consequently locally outputs `yes` (N.17\_3).

## 3.4 Improving the Framework

This section shows that the previous generic algorithm can be improved so as to reduce the size and the number of messages that are exchanged.

### 3.4.1 Two Types of Filtering

**Filtering That Affects the Content of Messages** A trivial way to reduce the size of messages exchanged during a round  $r$ , is for each process  $p_i$  to not send on a channel  $channel_i[x]$  the complete information  $new_i$  it has learned during the previous round, but to send only the content of  $new_i$  minus what has been received on this channel during the previous round.

Let  $received_i[x]$  be the information received by  $p_i$  on the channel  $channel_i[x]$  during a round  $r$ . Thus,  $p_i$  has to send only  $sent_i[x] = new_i \setminus received_i[x]$  on  $channel_i[x]$  during the round  $r + 1$ . This is the first filtering: it affects the content of messages themselves.

**Filtering That Affects the Channels That Are Used** The idea is here for a process  $p_i$  to manage its channels according to their potential for carrying new information.

To that end, let us consider two neighbor processes  $p_i$  and  $p_j$  such that we have  $new_i = new_j$  at the end of round  $r$ . According to the message exchange pattern and the round-based synchronization, this means that  $p_i$  and  $p_j$  have exactly the same set  $E$  of processes at distance  $r$ , and during that round both learn the input values of the processes in the set  $E$ . Hence, if  $p_i$  and  $p_j$  knew that  $new_i = new_j$ , they could stop exchanging messages. This is because the new information they will acquire in future rounds will be the inputs of the processes at distance  $r + 1$ ,  $r + 2$ , etc., which they will obtain independently from one another (by way of the processes in the set  $E$ ).

The problem of exploiting this property lies in the fact that, at the end of a round, a process does not know the value of the set  $new$  that it will receive from each of its neighbors. However, at the end of a round  $r$ , each process  $p_i$  knows, for each of its channels  $channel_i[x]$ , the value  $sent_i[x]$  it has sent on this channel, and the value  $received_i[x]$  it has received on this channel. Four cases may occur (let  $p_j$  denote the neighbor to which  $p_i$  is connected by  $channel_i[x]$ ).

1.  $sent_i[x] = received_i[x]$ . In this case,  $p_i$  and  $p_j$  sent to each other the same information during round  $r$ . They do not learn any new information from each other. What is learned by  $p_i$  is the fact that it has learned in round  $(r - 1)$  the information that  $p_j$  sent it in round  $r$ , and similarly for  $p_j$ . Hence, from now on, they will not learn anything more from each other.
2.  $sent_i[x] \subset received_i[x]$ . In this case,  $p_j$  does not learn anything new from  $p_i$  in the current round. Hence, it will not learn anything new from  $p_i$  in the future rounds.
3.  $received_i[x] \subset sent_i[x]$ . This case is the inverse of the previous one:  $p_i$  learns that it will never learn new information on the channel  $channel_i[x]$ , in all future rounds.
4. In the case where  $sent_i[x]$  and  $received_i[x]$  cannot be compared, both  $p_i$  and  $p_j$  learn new information from each other,  $received_i[x] \setminus sent_i[x]$  as far as  $p_i$  is concerned.

These items allow for the implementation of the second filtering. It is based on the values carried by the messages for managing the use of the communication channels.

### 3.4.2 An Improved Algorithm

**Two More Local Variables** To implement the previous filtering, each process  $p_i$  is provided with two local set variables, denoted  $c\_in_i$  and  $c\_out_i$ . They contain indexes of local channels, and are initialized to  $\{1, \dots, c_i\}$ . These variables are used as follows in order to implement the management of the communication channels:

- When  $sent_i[x] = received_i[x]$  (item 1),  $x$  is suppressed from both  $c\_in_i$  and  $c\_out_i$ .
- When  $sent_i[x] \subset received_i[x]$  (item 2),  $x$  is suppressed from  $c\_out_i$ .
- When  $received_i[x] \subset sent_i[x]$  (item 3),  $x$  is suppressed from  $c\_in_i$ .

```

init  $inf_i \leftarrow \{(id_i, in_i)\}; new_i \leftarrow \{(id_i, in_i)\}; r_i \leftarrow 0;$ 
 $c\_in_i \leftarrow \{1, \dots, c_i\}; c\_out_i \leftarrow \{1, \dots, c_i\};$ 
for each  $x \in \{1, \dots, c_i\}$  do  $received_i[x] \leftarrow \emptyset$  end for.

(1) while  $(c\_in_i \neq \emptyset) \vee (c\_out_i \neq \emptyset)$  do
    begin asynchronous round
(2)  $r_i \leftarrow r_i + 1;$ 
(3) for each  $x \in c\_out_i$  do
(4) if  $(x \in c\_in_i)$  then  $sent_i[x] \leftarrow new_i \setminus received_i[x]$  else  $sent_i[x] \leftarrow new_i$  end if;
(5) send  $MSG(sent_i[x])$  on  $channel_i[x];$ 
(6) if  $(sent_i[x] = \emptyset)$  then  $c\_out_i \leftarrow c\_out_i \setminus \{x\}$  end if
(7) end for;
(8)  $new_i \leftarrow \emptyset;$ 
(9) for each  $x \in c\_in_i$  do
(10) wait  $(MSG(received_i[x])$  received on  $channel_i[x]);$ 
(11) if  $(x \in c\_out_i)$ 
(12) then if  $(sent_i[x] \subseteq received_i[x])$  then  $c\_out_i \leftarrow c\_out_i \setminus \{x\}$  end if;
(13) if  $(received_i[x] \subseteq sent_i[x])$  then  $c\_in_i \leftarrow c\_in_i \setminus \{x\}$  end if
(14) else if  $(received_i[x] = \emptyset)$  then  $c\_in_i \leftarrow c\_in_i \setminus \{x\}$  end if
(15) end if;
(16)  $new_i \leftarrow new_i \cup (received_i[x] \setminus inf_i)$ 
(17) end for;
(18)  $inf_i \leftarrow inf_i \cup new_i$ 
end asynchronous round
(19) end while;
(20) return $(inf_i).$ 

```

**Fig. 3.6** A general algorithm with filtering (code for  $p_i$ )

**The Improved Algorithm** The final algorithm is described in Fig. 3.6. It is the algorithm of Fig. 3.2 modified according to the previous discussion. The local termination is now satisfied when a process  $p_i$  can no longer communicate; i.e., it is captured by the local predicate  $(c\_in_i = \emptyset) \wedge (c\_out_i = \emptyset)$ .

When it executes its “send” part of the algorithm (lines 3–7), a process  $p_i$  has now to compute the value  $sent_i[x]$  sent on each open output channel  $channel_i[x]$ . Moreover, if  $sent_i[x] = \emptyset$ , this channel is suppressed from the set  $c\_out_i$ , which contains the indexes of the output channels of  $p_i$  that are still open. In this case, the receiving process  $p_j$  will also suppress the corresponding channel from its set of open input channels  $c\_in_j$  (lines 14).

When it executes its “receive” part of the algorithm (lines 9–17), a process  $p_i$  updates its set of input channels  $c\_in_i$  and its set of output channels  $c\_out_i$  according to the value  $received_i[x]$  it has received on each input channel that is still open (i.e., on each  $channel_i[x]$  such that  $x \in c\_in_i$ ).

**Complexity** The algorithm terminates in  $D + 1$  rounds. This comes from the fact that when  $sent_i[x] = \emptyset$ , both the sender  $p_i$  and the receiver  $p_j$  withdraw the corresponding channel from  $c\_out_i$  and  $c\_in_j$ , respectively. The maximum number of messages is consequently  $2e(D + 1)$ . The time complexity is  $2D + 1$  in the worst case (which occurs when a single process starts, its first round message wakes up

other processes, etc., and the eccentricity of the starting process is equal to the diameter  $D$  of the communication graph).

## 3.5 The Case of Regular Communication Graphs

### 3.5.1 Tradeoff Between Graph Topology and Number of Rounds

A graph is characterized by several parameters, among which its number of vertices  $n$ , its number of edges  $e$ , its diameter  $D$ , and its maximal degree  $\Delta$ , are particularly important.

**On the Message Complexity** This appears clearly in the message complexity (denoted  $C$  in the following) of the previous algorithm for which  $C$  is upper bounded by  $2e(D + 1)$ . If  $D$  is known by the processes, one round is saved, and we have  $C = 2eD$ . This means that

- If the graph is fully connected we have  $D = 1$ ,  $e = n(n - 1)/2$ , and  $C = O(n^2)$ .
- If the graph is a tree we have  $e = (n - 1)$ , and  $C = O(nD)$ .

This shows that it can be interesting to first build a spanning tree of the communication graph and then use it repeatedly to compute global functions. However, for some problems, a tree is not satisfactory because the tree that is obtained can be strongly unbalanced in the sense that processes may have a distinct number of neighbors.

**The Notion of a Regular Graph** Hence, for some problems, we are interested in communication graphs in which the processes have the same number of neighbors (i.e., the same degree  $\Delta$ ). When, they exist, such graph are called *regular*. In such a graph we have  $e = (n\Delta)/2$ , and consequently we obtain

$$C = n\Delta D.$$

This relation exhibits a strong relation linking three of the main parameters associated with a regular graph.

**What Regular Graphs Can Be Built?** Given  $\Delta$  and  $D$ , Moore's bound (1958) is an upper bound on the maximal number of vertices (processes) that a regular graph with diameter  $D$  and degree  $\Delta$  can have. This number is denoted  $n(D, \Delta)$ , and we have  $n(D, \Delta) \leq 1 + \Delta + \Delta(\Delta - 1) + \dots + \Delta(\Delta - 1)^{D-1}$ , i.e.,

$$n(D, 2) \leq 2D + 1, \quad \text{and}$$

$$n(D, \Delta) \leq \frac{\Delta(\Delta - 1)^D - 2}{\Delta - 2} \quad \text{for } \Delta > 2.$$

This is an upper bound. Therefore (a) it does not mean that regular graphs for which  $n(D, \Delta)$  is equal to the bound exist for any pair  $(D, \Delta)$ , and (b) when such a graph exists, it does not state how to build it. However, this bound states that, in the regular graphs that can be built, we have  $\Delta \geq \sqrt[D]{n}$ . It follows that, in the regular networks that can be built, we have

$$C = D \sqrt[D]{n^{D+1}}.$$

This formula relates clearly the number of rounds  $D$  and the number of messages exchanged at each round, namely  $n \sqrt[D]{n}$ .

### 3.5.2 De Bruijn Graphs

The graphs known as *De Bruijn's graphs* are directed regular graphs, which can be easily built. This section presents them, and shows their interest in computing global functions. (These graphs can also be used as overlay structures in distributed applications.)

Let  $x$  be a vertex of a directed graph.  $\Delta^+(x)$  denotes its input degree (number of incoming edges), while  $\Delta^-(x)$  denotes its output degree (number of output edges). In a regular network, we have  $\forall x : \Delta^+(x) = \Delta^-(x) = \Delta$ , and the value  $\Delta$  defines the degree of the graph.

**De Bruijn's Graph** Let us consider a vocabulary  $V$  of  $\Delta$  letters (e.g.,  $\{0, 1, 2\}$  for  $\Delta = 3$ ).

- The vertices are all the words of length  $D$  that can be built on a vocabulary  $V$  of  $\Delta$  letters. Hence, there are  $n = \Delta^D$  vertices.
- Each vertex  $x = [x_1, \dots, x_{D-1}, x_D]$  has  $\Delta$  output edges that connect it to the vertices  $y = [x_2, \dots, x_D, \alpha]$ , where  $\alpha \in V$  (this is called the *shifting* property).

It follows from this definition that the input channels of a vertex  $x = [x_1, \dots, x_{D-1}, x_D]$  are the  $\Delta$  vertices labeled  $[\beta, x_1, \dots, x_{D-1}]$ , where  $\beta \in V$ .

Let us also observe that the definition of the directed edges implies that each vertex labeled  $[a, a, \dots, a]$ ,  $a \in V$ , has a channel to itself (this channel counts then as both an input channel and an output channel).

A De Bruijn's graph defined from a specific pair  $(\Delta, D)$  is denoted  $\text{dB}(\Delta, D)$ . Such a graph has  $n = \Delta^D$  vertices and  $e = n\Delta$  directed edges.

**Examples of De Bruijn's Graphs** Examples of directed De Bruijn's graphs are described Fig. 3.7.

- The graph at the top of the figure is  $\text{dB}(2,1)$ . We have  $\Delta = 2$ ,  $D = 1$ , and  $n = 2^1 = 2$  vertices.
- The graph in the middle of the figure is  $\text{dB}(2,2)$ . We have  $\Delta = 2$ ,  $D = 2$ , and  $n = 2^2 = 4$  vertices.
- The graph at the bottom of the figure is  $\text{dB}(2,3)$ . We have  $\Delta = 2$ ,  $D = 3$ , and  $n = 2^3 = 8$  vertices.

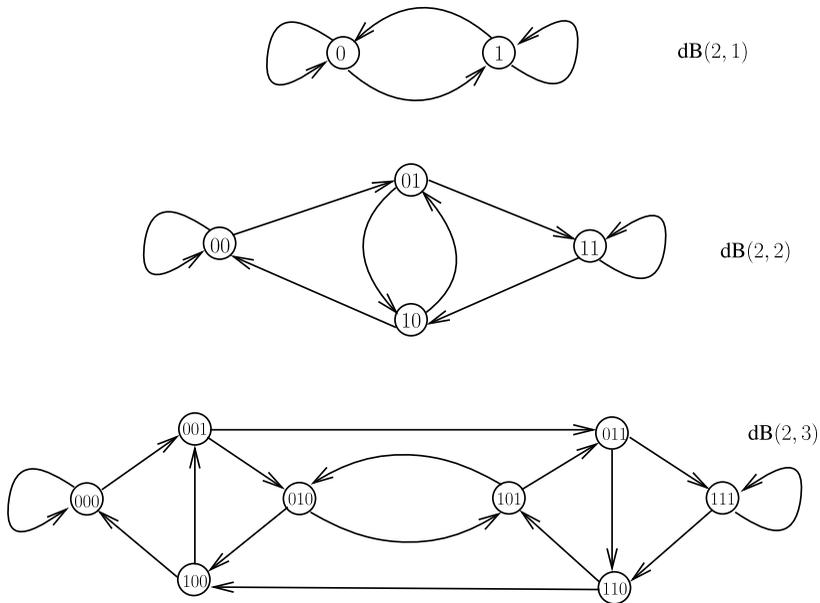


Fig. 3.7 The De Bruijn's directed networks  $dB(2,1)$ ,  $dB(2,2)$ , and  $dB(2,3)$

**A Fundamental Property of a De Bruijn's Graph** In addition to being easily built, De Bruijn's graphs possess a noteworthy property which makes them attractive for the computation of global functions, namely there is exactly one directed path of length  $D$  between any pair of vertices (including each pair of the form  $(x, x)$ ).

**Computing a Function on a De Bruijn's Graph** The algorithm described in Fig. 3.8 is tailored to compute a global function  $F()$  in a round-based synchronous (or asynchronous) distributed system whose communication graph is a De Bruijn's graph (as seen in Fig. 2.9, messages have to carry the parity bit of the current round number in the asynchronous case).

This algorithm is designed to benefit from the fundamental property of De Bruijn's graphs. Let  $c_{in_i}$  denote the set of  $\Delta$  input channels and  $c_{out_i}$  denote the set of  $\Delta$  output channels of  $p_i$ . At the end of a round, the local variable  $received_i$  contains all pairs  $(j, in_j)$  received by  $p_i$  in the current round (lines 3–7). Initially,  $received_i$  contains the input pair of  $p_i$ , i.e.  $(i, in_i)$ . When, it starts a round, a process  $p_i$  sends the value of  $received_i$  on each of its output channels (line 2). Hence, during a round  $r$ , a process sends on each of its output channels what it has learned from all its input channels during the previous round  $(r - 1)$ . There is neither filtering nor memorization of what has been received during all previous rounds  $r' < r$ .

It follows from the fundamental property of De Bruijn's graphs that, at the end of the last round  $r = D$ , the set  $received_i$  of each process  $p_i$  contains all the input values, each exactly once; i.e., we have then  $received_i = \{(1, in_1)$ ,

```

init  $received_i \leftarrow \{(i, in_i)\}$ .

(1) when  $r = 1, 2, \dots, D$  do
    begin synchronous round
(2)   for each  $x \in c\_out_i$  do send MSG( $received_i$ ) on  $channel_i[x]$  end for;
(3)    $received_i \leftarrow \emptyset$ ;
(4)   for each  $x \in c\_in_i$  do
(5)     wait (MSG( $rec$ ) received on  $channel_i[x]$ );
(6)      $received_i \leftarrow received_i \cup rec$ 
(7)   end for
    end synchronous round
(8) end when;
(9)  $out_i \leftarrow F(received_i)$ ;
(10) return( $out_i$ ).

```

**Fig. 3.8** A generic algorithm for a De Bruijn's communication graph (code for  $p_i$ )

$(2, in_2), \dots, (n, in_n)\}$ . Each process can consequently compute  $F(received_i)$  and returns the corresponding result.

**A Simple Example** Let us consider the communication graph DB(2,2) (the one described in the middle of Fig. 3.7). We have the following:

- At the end of the first round:
  - The process labeled 00 is such that  $received_{00} = \{(00, in_{00}), (10, in_{10})\}$ .
  - The process labeled 10 is such that  $received_{10} = \{(01, in_{01}), (11, in_{11})\}$ .
- At the end of the second round, the process labeled 01 is such that  $received_{01}$  contains the values of  $received_{00}$  and  $received_{10}$  computed at the previous round. We consequently have  $received_{01} = \{(00, in_{00}), (10, in_{10}), (01, in_{01}), (11, in_{11})\}$ .

If the function  $F()$  is associative and commutative, a process can compute  $to\_send = F(received_i)$  at the end of each round, and send this value instead of  $received_i$  during the next round (line 2). Merging of files,  $\max()$ ,  $\min()$ , and  $+$  are examples of such functions.

## 3.6 Summary

This chapter has presented a general framework to compute global functions on a set of processes which are the nodes of a network. The main features of this framework are that all processes execute the same local algorithm, and no process learns more than what it needs to compute  $F$ . Moreover, the knowledge of the diameter  $D$  is not necessary and the time complexity is  $2(D + 1)$ , while the total number of messages is upper bounded by  $2e(D + 2)$ , where  $e$  is the number of communication channels.

### 3.7 Bibliographic Notes

- The general round-based framework presented in Sect. 3.2 and its improvement presented in Sect. 3.4 are due to J.-Cl. Bermond, J.-Cl. König, and M. Raynal [48].
- The algorithm that computes the cut vertices of a communication graph is due to J.-Cl. Bermond and J.-Cl. König [47]. Other distributed algorithms determining cut vertices have been proposed (e.g., [187]).
- The tradeoff between the number of rounds and the number of messages is addressed in [223, 243] and in the books [292, 319].
- The use of the general framework in regular directed networks has been investigated in [46]. Properties of regular networks (such as hypercubes, De Bruijn's graphs, and Kautz's graphs) are presented in [45, 49].
- A general technique for network synchronization is presented [27].
- A graph problem is *local* if it can be solved by a distributed algorithm with time complexity smaller than  $D$  (the diameter of the corresponding graph). The interested reader will find in [236] a study on the locality of the graph coloring problem in rings, regular trees of radius  $r$ , and  $n$ -vertex graphs with largest degree  $\Delta$ .

### 3.8 Problem

1. Adapt the general framework presented in Sect. 3.2 to communication graphs in which the channels are unidirectional. It is, however, assumed that the graphs are strongly connected (there is a directed path from any process to any process).

# Chapter 4

## Leader Election Algorithms

This chapter is on the leader election problem. Electing a leader consists for the processes of a distributed system in selecting one of them. Usually, once elected, the leader process is required to play a special role for coordination or control purposes.

Leader election is a form of symmetry breaking in a distributed system. After showing that no leader can be elected in anonymous regular networks (such as rings), this chapter presents several leader election algorithms with a special focus on non-anonymous ring networks.

**Keywords** Anonymous network · Election · Message complexity · Process identity · Ring network · Time complexity · Unidirectional versus bidirectional ring

### 4.1 The Leader Election Problem

#### 4.1.1 Problem Definition

Let each process  $p_i$  be endowed with two local Boolean variables  $elect_i$  and  $done_i$ , both initialized to *false*. (Let us recall that  $i$  is the index of  $p_i$ , i.e., a notational convenience that allows us to distinguish processes. Indexes are not known by the processes.) The Boolean variables  $elect_i$  are such that eventually exactly one of them becomes true, while each Boolean variable  $done_i$  becomes true when the corresponding process  $p_i$  learns that a process has been elected. More formally, the election problem is defined by the following safety and liveness properties, where  $var_i^\tau$  denotes the value of the local variable  $var_i$  at time  $\tau$ .

- Safety property:
  - $(\forall i : elect_i^\tau \Rightarrow (\forall \tau' \geq \tau : elect_i^{\tau'})) \wedge (\forall i : done_i^\tau \Rightarrow (\forall \tau' \geq \tau : done_i^{\tau'}))$ .
  - $\forall i, j, \tau, \tau' : (i \neq j) \Rightarrow \neg(elect_i^\tau \wedge elect_j^{\tau'})$ .
  - $\forall i : done_i^\tau \Rightarrow (\exists j, \tau' \leq \tau : elect_j^{\tau'})$ .

The first property states that the local Boolean variables  $elect_i$  and  $done_i$  are stable (once true, they remain true forever). The second property states that at most one process is elected, while the third property states that a process cannot learn that the election has terminated while no process has yet been elected.

- Termination property:

- $\exists i, \tau : \text{elected}_i^\tau$ .
- $\forall i : \exists \tau : \text{done}_i^\tau$ .

This liveness property states that a process is eventually elected, and this fact is eventually known by all processes.

### 4.1.2 Anonymous Systems: An Impossibility Result

This section assumes that the processes have no identities. Said differently, there is no way to distinguish a process  $p_i$  from another process  $p_j$ . It follows that all the processes have the same number of neighbors, the same code, and the same initial state (otherwise these features could be considered as their identities). The next theorem shows that, in such an anonymity context, there is no solution to the leader election problem. For simplicity reasons, the theorem considers the case where the network is a ring.

**Theorem 2** *There is no deterministic election algorithm for leader election in a (bi/unidirectional) ring of  $n > 1$  processes.*

*Proof* The proof is by contradiction. Let us assume that there is a deterministic distributed algorithm  $A$  that solves the leader election problem in an anonymous ring.  $A$  is composed of  $n$  local deterministic algorithms  $A_1, \dots, A_i, \dots, A_n$ , where  $A_i$  is the local algorithm executed by  $p_i$ . Moreover, due to anonymity, we have  $A_1 = \dots = A_n$ . We show that  $A$  cannot satisfy both the safety and the termination properties that define the leader election problem.

For any  $i$ , let  $\sigma_i^0$  denote the initial state of  $p_i$ , and let  $\Sigma^0 = (\sigma_1^0, \dots, \sigma_n^0)$  denote the initial global state. As all the processes execute the same local deterministic algorithm  $A_i$ , there is a synchronous execution during which all the processes execute the same step of their local deterministic algorithm  $A_i$ , and each process  $p_i$  proceeds consequently from  $\sigma_i^0$  to  $\sigma_i^1$  (this step can be the execution of an internal statement or a communication operation). Moreover, as  $\sigma_1^0 = \dots = \sigma_n^0$  and  $A_i$  is deterministic and the same for all processes, it follows that the set of processes progress from  $\Sigma^0 = (\sigma_1^0, \dots, \sigma_n^0)$  to  $\Sigma^1 = (\sigma_1^1, \dots, \sigma_n^1)$ , where  $\sigma_1^1 = \dots = \sigma_n^1$ . The important point here is that the execution progresses from a symmetric global state  $\Sigma^0$  to another symmetric global state  $\Sigma^1$  (“symmetric” meaning here that all processes are in the same local state).

As all  $A_i$  are deterministic and identical, the previous synchronous execution can be continued and the set of processes progresses then from the symmetric global state  $\Sigma^1$  to a new symmetric global state  $\Sigma^2$ , etc. It follows that the synchronous execution can be extended forever from a symmetric global state  $\Sigma$  to another symmetric global state  $\Sigma'$ .

Consequently, the previous synchronous execution never terminates. This is because for it to terminate, it has to enter an asymmetric global state (due to its very definition, the global state in which a process is elected is asymmetric). Hence, the algorithm  $A$  fails to satisfy the termination property of the leader election problem, which concludes the proof of the theorem.  $\square$

### 4.1.3 Basic Assumptions and Principles of the Election Algorithms

**Basic Assumptions** Due to the previous theorem, the rest of this chapter assumes that each process  $p_i$  has an identity  $id_i$ , and that distinct processes have different identities (i.e.,  $i \neq j \Rightarrow id_i \neq id_j$ ). Moreover, it is assumed that identities can be compared with the help of the operators  $<$ ,  $=$ ,  $>$ .

**Basic Principles of the Election Algorithms** The basic idea of election algorithms consists in electing the process whose identity is an extremum (the greatest identity or the smallest one) among the set of all processes or a set of candidate processes. As no two processes have the same identity and all identities can be compared, there is a single extremum identity.

## 4.2 A Simple $O(n^2)$ Leader Election Algorithm for Unidirectional Rings

### 4.2.1 Context and Principle

**Context** This section considers a network structured as a unidirectional ring, i.e., each process has two neighbors, one from which it can receive messages and another one to which it can send messages. Moreover, a channel connecting two neighbor processes is not required to be FIFO (first in first out).

A process knows initially its identity and the fact that no two processes have the same identity. It does not know the total number of processes  $n$  that define the ring.

**Principle** This algorithm, which is due to E. Chang and R. Roberts (1979), is one of the very first election algorithms. The idea is the following. Each process  $p_i$  sends its identity on the ring, and stops the progress of any identity  $id_j$  it receives if  $id_j < id_i$ . As all the processes have distinct identities, it follows that the only identity whose progress on the ring cannot be stopped is the highest identity. This is the identity of the process that is eventually elected.

```

when START() is received do
(1) if ( $\neg part_i$ ) then  $part_i \leftarrow true$ ; send ELECTION( $id_i$ ) end if.

when ELECTION( $id$ ) is received do
(2) case ( $id > id_i$ ) then  $part_i \leftarrow true$ ; send ELECTION( $id$ )
(3)   ( $id < id_i$ ) then if ( $\neg part_i$ ) then  $part_i \leftarrow true$ ; send ELECTION( $id_i$ ) end if
(4)   ( $id = id_i$ ) then send ELECTED( $id$ );  $elected_i \leftarrow true$ 
(5) end case.

when ELECTED( $id$ ) is received do
(6)  $leader_i \leftarrow id$ ;  $done_i \leftarrow true$ ;
(7) if ( $id \neq id_i$ ) then  $elected_i \leftarrow false$ ; send ELECTED( $id$ ) end if.

```

**Fig. 4.1** Chang and Robert's election algorithm (code for  $p_i$ )

## 4.2.2 The Algorithm

The code of the algorithm is described in Fig. 4.1. In addition to its identity  $id_i$ , and the Boolean variables  $elected_i$  and  $done_i$ , each process  $p_i$  manages another Boolean denoted  $part_i$  (and initialized to *false*), and a variable  $leader_i$  that will contain the identity of the elected process.

The processes that launch the election are the processes  $p_i$  that receive the external message START() while  $part_i = false$ . When this occurs,  $p_i$  becomes a participant and sends the message ELECTION( $id_i$ ) on its single outgoing channel (line 1).

When a process  $p_i$  receives a message ELECTION( $id$ ) on its single input channel, there are three cases. If  $id > id_i$ ,  $p_i$  cannot be the winner, and it forwards the message it has received (line 2). If  $id < id_i$ , if not yet done,  $p_i$  sends the message ELECTION( $id_i$ ) (line 3). Finally, if  $id = id_i$ , its message ELECTION( $id_i$ ) has visited all the processes, and consequently  $id_i$  is the highest identity. So,  $p_i$  is the elected process. Hence, it informs the other processes by sending the message ELECTED( $id_i$ ) on its outgoing channel (line 4).

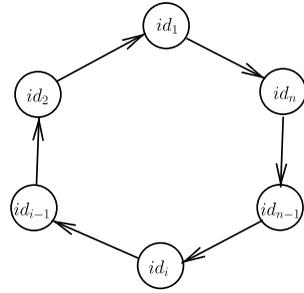
When a process  $p_i$  receives a message ELECTED( $id$ ), it learns both the identity of the leader and the fact that the election is terminated (line 5). Then, it forwards the message ELECTED( $id$ ) so that all processes learn the identity of the elected leader.

## 4.2.3 Time Cost of the Algorithm

In all cases, the algorithm sends  $n$  messages ELECTED(), which are sent one after the other. Hence, in the following we focus only on the cost due to the messages ELECTION(). To compute the time complexity, we assume that each message takes one time unit (i.e., all messages take the worst transfer delay which is defined as being equal to one time unit).

The best case occurs when only the process  $p_i$  with the highest identity receives a message START(). It is easy to see that the time complexity is  $n$  (the message

**Fig. 4.2**  
Worst identity distribution  
for message complexity



ELECTION( $id_i$ ) sent by  $p_i$  is passed from each process to its neighbor on the ring before returning to  $p_i$ ).

The worst case occurs when the process  $p_j$  with the second highest identity (a) is the only process that receives a message START(), and (b) follows on the ring the process  $p_i$  that has the highest identity. Hence,  $(n - 1)$  processes separate  $p_j$  and  $p_i$ . The message ELECTION( $id_j$ ) takes  $(n - 1)$  time units before attaining  $p_i$ , and then the message ELECTION( $id_i$ ) takes  $n$  times units to travel the ring. Hence,  $(2n - 1)$  time units are required. It follows that, whatever the case, an election requires between  $n$  and  $(2n - 1)$  times units.

### 4.2.4 Message Cost of the Algorithm

**Best Case and Worst Case** The best case for the number of messages is the same as for the time complexity, which happens when only the process with the highest identity receives a message START(). The algorithm then gives rise to exactly  $n$  messages ELECTION(). The worst case is when (a) each process  $p_i$  receives a message START() while  $part_i = false$ , (b) each message takes one time unit, and (c) the processes are ordered in the ring as follows: first, the process with the highest identity, then the one with the second highest identity, etc., until the process with the smallest identity (Fig. 4.2 where  $id_n$  is the highest identity, etc., until  $id_1$ , which is the smallest one).

It follows that the message START() received by the process with the smallest identity gives rise to one message ELECTION(), the one with the second smallest identity gives rise to two messages ELECTION() (one from itself to the process with the smallest identity, plus another one from this process to the process with the highest identity), and so on until the process with the highest identity whose message START() gives rise to  $n$  messages ELECTION(). It follows that the total number of messages is  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ , i.e.,  $O(n^2)$ .

**Average Case** To compute the message complexity in the average case, let  $P(i, k)$  be the probability that the message ELECTION() sent by the process  $p_x$  with the  $i$ th smallest identity is forwarded  $k$  times. Assuming that the direction of the ring

is clockwise (as in Fig. 4.2),  $P(i, k)$  is the probability that the  $k - 1$  clockwise neighbors of  $p_x$  (the processes that follow  $p_x$  on the ring) have an identity smaller than  $id_x$  and the  $k$ th clockwise neighbor of  $p_x$  has an identity greater than  $id_x$ . Let us recall that there are  $(i - 1)$  processes with an identity smaller than  $id_x$  and  $(n - i)$  processes with an identity greater than  $id_x$ .

Let  $\binom{a}{b}$  denote the number of ways of choosing  $b$  elements in a set of  $a$  elements. We have

$$P(i, k) = \frac{\binom{i-1}{k-1}}{\binom{n-1}{k-1}} \times \frac{n-i}{n-k}.$$

Since there is a single message that makes a full turn on the ring (the one carrying the highest identity), let us consider each of the  $(n - 1)$  other messages. The expected number of passes of the  $i$ th message (where  $i$  denotes the rank of the identity of the corresponding ELECTION() message) is then for  $i \neq n$ :

$$E_i(k) = \sum_{k=1}^{n-1} kP(i, k).$$

Hence, the expected number of transfers for all messages is

$$E = n + \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} kP(i, k),$$

which can be simplified to

$$E = n + \sum_{k=1}^{n-1} \frac{n}{k+1} = n \left( 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right).$$

As the harmonic series  $1 + 1/2 + \cdots + 1/n + \cdots$  is upper bounded by  $c + \log_e n$  (where  $c$  is a constant), the average number of ELECTION() messages  $E$  is upper bounded by  $O(n \log n)$ .

### 4.2.5 A Simple Variant

The previous algorithm always elects the process with the maximal identity. It is possible to modify this algorithm to obtain an algorithm that elects the process with the highest identity among the processes whose participation in the algorithm is due to the reception of a START() message (this means that, whatever its identity, a process that does not receive a START() message, or that receives such a message only after having received an ELECTION() message, cannot be elected).

The corresponding algorithm is depicted in Fig. 4.3. Each process  $p_i$  manages a local variable  $idmax_i$  which contains the greatest identity of a competing process seen by  $p_i$ . Initially,  $idmax_i$  is equal to 0.

```

when START() is received do
(1) if ( $idmax_i = 0$ ) then  $idmax_i \leftarrow id_i$ ; send ELECTION( $id_i$ ) end if.

when ELECTION( $id$ ) is received do
(2) case ( $id > idmax_i$ ) then  $idmax_i \leftarrow id$ ; send ELECTION( $id$ )
(3)     ( $id < idmax_i$ ) then skip
(4)     ( $id = id_i$ )   then send ELECTED( $id$ );  $elected_i \leftarrow true$ 
(5) end case.

when ELECTED( $id$ ) is received do
(6)  $leader_i \leftarrow id$ ;  $done_i \leftarrow true$ ;
(7) if ( $id \neq id_i$ ) then  $elected_i \leftarrow false$ ; send ELECTED( $id$ ) end if.

```

**Fig. 4.3** A variant of Chang and Robert's election algorithm (code for  $p_i$ )

When it receives a START() message (if it ever receives one), a process  $p_i$  considers and processes it only if  $idmax_i = 0$ . Moreover, when it receives a message ELECTION( $id$ ), a process  $p_i$  discards it if  $id < idmax_i$  (this is because  $p_i$  has seen a competing process with a higher identity). The rest of the algorithm is similar to the algorithm of Fig. 4.1.

## 4.3 An $O(n \log n)$ Leader Election Algorithm for Bidirectional Rings

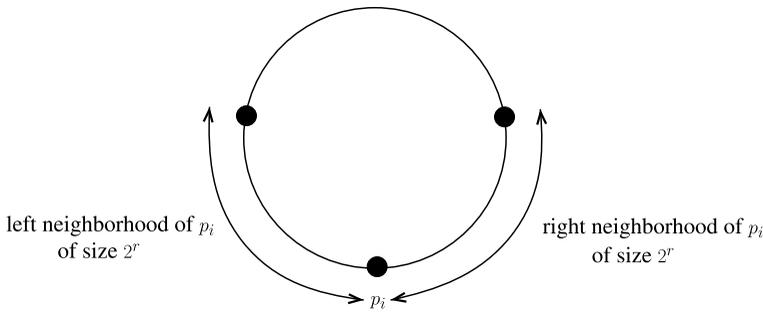
### 4.3.1 Context and Principle

**Context** This section considers a bidirectional ring. As before, each process has a left neighbor and a right neighbor, but it can now send a message to, and receive a message from, any of these neighbors. Given a process  $p_i$ , the notations  $left_i$  and  $right_i$  are used to denote the channels connecting  $p_i$  to its left neighbor and to its right neighbor, respectively.

The notions of left and right are global: they are the same for all processes, i.e., going only to left allows a message to visit all processes (and similarly when going only to right).

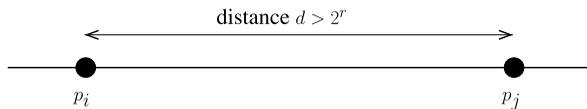
**Principle** The algorithm presented below is due to D.S. Hirschberg and J.B. Sinclair (1980). It is based on the following idea. The processes execute asynchronous rounds. During each round, processes compete, and only the processes that win in a round  $r$  are allowed to continue competing during round  $r + 1$ . During the first round (denoted round 0), all processes compete.

A process  $p_i$ , which is competing during round  $r$ , is a winner at the end of that round if it has the largest identifier on the part of the ring that spans the  $2^r$  processes on its left side and the  $2^r$  processes on its right side, i.e., in a continuous neighborhood of  $2^{r+1} + 1$  processes. This is illustrated in Fig. 4.4.



**Fig. 4.4** Neighborhood of a process  $p_i$  competing during round  $r$

**Fig. 4.5** Competitors at the end of round  $r$  are at distance greater than  $2^r$



If it has the highest identity,  $p_i$  proceeds to the next round as a competitor. Otherwise, it no longer competes to become leader. It follows that any two processes that remain competitors after round  $r$  are at a distance  $d > 2^r$  (see Fig. 4.5, where  $p_i$  and  $p_j$  are competitors at the end of round  $r$ ). Said differently, after each round, the number of processes that compete to become leader is divided at least by two.

### 4.3.2 The Algorithm

To simplify the presentation, it is assumed that each process receives a message `START()` before any message sent by the algorithm.

The algorithm is described in Fig. 4.6. When it starts participating in the algorithm, a process  $p_i$  sends a message `ELECTION( $id_i, r, d$ )` to its left and right neighbors (line 1), where  $id_i$  is its identity,  $r = 0$  is the round number associated with this message (let us recall that the first round is numbered 0), and  $d = 1$  is the number of processes that have already been visited by this message.

When a process  $p_i$  receives a message `ELECTION( $id, r, d$ )`, its behavior depends on the respective values of  $id_i$  and  $id$ . If  $id > id_i$ , there are two cases.

- If  $d < 2^r$ , the message has not yet visited the full left (or right) neighborhood of size  $2^r$  it has to visit. In this case,  $p_i$  forwards the message `ELECTION( $id_i, r, d + 1$ )` to the appropriate right or left neighbor (line 2).
- If  $d \geq 2^r$ , the message has visited all the process neighborhood it had to visit. As the progress of this message has not been stopped,  $p_i$  sends back (line 3) the message `REPLY( $id, r$ )` to inform the process  $p_x$  whose identity is  $id$  that the message `ELECTION( $id, r, -$ )` it sent has visited all the processes of its right (or left) neighborhood of size  $2^r$  without being stopped (this neighborhood starts at  $p_x$  and ends at  $p_i$ ).

```

when START() is received do
(1) send ELECTION( $id_i, 0, 1$ ) on both  $left_i$  and  $right_i$ .

when ELECTION( $id, r, d$ ) is received on  $left_i$  (resp.,  $right_i$ ) do
(2) case ( $id > id_i$ )  $\wedge$  ( $d < 2^r$ ) then send ELECTION( $id, r, d + 1$ ) to  $right_i$  (resp.,  $left_i$ )
(3)   ( $id > id_i$ )  $\wedge$  ( $d \geq 2^r$ ) then send REPLY( $id, r$ ) to  $left_i$  (resp.,  $right_i$ )
(4)   ( $id < id_i$ ) then skip
(5)   ( $id = id_i$ ) then send ELECTED( $id$ ) on  $left_i$ ;  $electe_d_i \leftarrow true$ 
(6) end case.

when REPLY( $id, r$ ) is received on  $left_i$  (resp.,  $right_i$ ) do
(7) if ( $id \neq id_i$ )
(8)   then send REPLY( $id, r$ ) on  $right_i$  (resp.,  $left_i$ )
(9)   else if (already received REPLY( $id, r$ ) from  $right_i$  (resp.,  $left_i$ ))
(10)    then send ELECTION( $id_i, r + 1, 1$ ) on both  $left_i$  and  $right_i$ 
(11)    end if
(12) end if.

when ELECTED( $id$ ) is received on  $right_i$  do
(13)  $leader_i \leftarrow id$ ;  $done_i \leftarrow true$ ;
(14) if ( $id \neq id_i$ ) then  $electe_d_i \leftarrow false$ ; send ELECTED( $id$ ) on  $left_i$  end if.

```

Fig. 4.6 Hirschberg and Sinclair's election algorithm (code for  $p_i$ )

If  $id_i > id$ , the process whose identity is  $id$  cannot be elected (this is because its identity is not the greatest one). Hence,  $p_i$  stops the progress of the message ELECTION( $id, -, -$ ) (line 4). Finally, if  $id_i = id$ , the message ELECTION( $id, r, -$ ) sent by  $p_i$  has visited all the processes without being stopped. It follows that  $id_i$  is the greatest identity, and  $p_i$  consequently sends a message ELECTED( $id_i$ ) (line 5), which will inform all processes that the election is over (lines 13–14).

When a process  $p_i$  receives a message REPLY( $id, r$ ),  $p_i$  forwards it to the appropriate (right or left) neighbor if it is not the final destination process of this message (line 8). If it is the final destination process ( $id_i = id$ ), and this is the second message REPLY( $id, r$ ) (i.e., the message coming from the other side of the ring),  $p_i$  learns that it has the highest identity in both its left and right neighborhoods of size  $2^r$  (line 9). It consequently proceeds to the next round by sending to its left and right neighbors the message ELECTION( $id_i, r + 1, 1$ ) (line 10), which starts its next round.

### 4.3.3 Time and Message Complexities

**Message Complexity** Let us first notice that a process  $p_i$  remains a competitor on its left and right neighborhoods, each of size  $2^r$ , only if it has not been defeated by a process within distance  $2^{r-1}$  on its left or its right neighborhood. Moreover, in a set of  $2^{r-1} + 1$  consecutive processes, at most one process can remain competitor for round  $r$ . It follows that we have the following:

- $n$  processes are competitors on paths of length  $2^0 = 1$  (round 0),
- At most  $\lceil \frac{n}{2} \rceil$  processes are competitors on paths of length  $2^1 = 2$  (round 1),
- At most  $\lceil \frac{n}{3} \rceil$  processes are competitors on paths of length  $2^2$  (round 2), etc.,
- At most  $\lceil \frac{n}{2^{r-1}+1} \rceil$  processes are competitors on paths of length  $2^r$  (round  $r$ ), etc.

Let us also observe that a process, which is competitor at round  $r$ , entails the transfer of at most  $4 \times 2^r$  messages (a message `ELECTION()` and a message `REPLY()`, one in each direction, both on the right path and the left path of size  $2^r$ ). It follows that the total number of `ELECTION()` and `REPLY()` messages is upper bounded by

$$4 \left( 1 \times n + 2 \left\lceil \frac{n}{2^0 + 1} \right\rceil + 2^2 \left\lceil \frac{n}{2^1 + 1} \right\rceil + 2^3 \left\lceil \frac{n}{2^2 + 1} \right\rceil + \cdots + 2^r \left\lceil \frac{n}{2^{r-1} + 1} \right\rceil + \cdots \right)$$

Each internal term is less than  $2n$ , and there are at most  $1 + \log_2 n$  terms. It follows that the total number of messages is upper bounded by  $8n(1 + \log_2 n)$ , i.e.,  $O(n \log_2 n)$ .

**Time Complexity** To simplify, let us assume that  $n = 2^k$ , and let us consider the best case, namely, all the processes start simultaneously. Moreover, as usual for time complexity, it is assumed that each message takes one time unit. The process with the highest identity will execute from round 0 until round  $k$ , and a round  $r$  will take  $2^r$  time units. By summing the time taken by all rounds, the process with the highest identity will be elected after at most  $2(1 + 2^1 + 2^2 + \cdots + 2^r + \cdots + 2^k)$  time units (the factor 2 is due to a message `ELECTION()` in one direction plus a message `REPLY()` in the other direction). It follows that, in the best case, the time complexity is upper bounded by  $2(\frac{2^{k+1}-1}{2-1}) = 4n - 2$ , i.e.,  $O(n)$ .

The worst case time complexity, which is also  $O(n)$ , is addressed in Exercise 3. This means that the time complexity is linear with respect to the size of the ring.

## 4.4 An $O(n \log n)$ Election Algorithm for Unidirectional Rings

### 4.4.1 Context and Principles

**Context** This section considers a unidirectional ring network in which the channels are FIFO (i.e., on each channel, messages are received in their sending order). As in the previous section, each process  $p_i$  knows only its identity  $id_i$  and the fact that no two processes have the same identity. No process knows the value  $n$ .

**Principle** The algorithm presented below is due to D. Dolev, M. Klawe, and M. Rodeh (1982). It is based on a very simple and elegant idea.

As in the previous election algorithms, initially all processes compete to be elected as a leader, and execute consecutive rounds to that end. During each round, at most half of the processes that are competitors remain competitors during the next round. Hence, there will be at most  $O(\log_2 n)$  rounds.



Fig. 4.7 Neighbor processes on the unidirectional ring

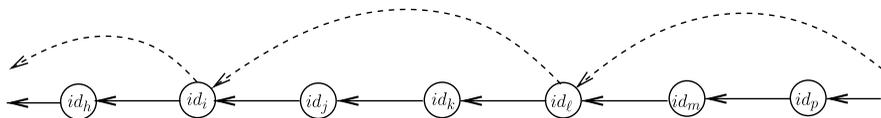


Fig. 4.8 From the first to the second round

The novel idea is the way the processes simulate a bidirectional ring. Let us consider three processes  $p_i$ ,  $p_j$ , and  $p_k$  such that  $p_i$  and  $p_k$  are the left and the right neighbor of  $p_j$  on the ring (Fig. 4.7). Moreover, let us assume that a process receives messages from its right neighbor and sends messages to its left neighbor.

During the first round, each process sends its identity to its left neighbor, and after it has received the identity  $id_x$  of its right neighbor  $p_x$ , it forwards  $id_x$  to its left neighbor. Hence, when considering Fig. 4.7,  $p_i$  receives first  $id_j$  and then  $id_k$ , which means that it knows three identities:  $id_i$ ,  $id_j$ , and  $id_k$ . It follows that it can play the role of  $p_j$ . More precisely

- If  $id_j > \max(id_i, id_k)$ ,  $p_i$  considers  $id_j$  as the greatest identity it has seen and progresses to the next round as a competitor on behalf of  $id_j$ .
- If  $id_j < \max(id_i, id_k)$ ,  $p_i$  stops being a competitor and its role during the next rounds will be to forward to the left the messages it receives from the right.

It is easy to see that, if  $p_i$  remains a competitor (on behalf of  $id_j$ ) during the second round, its left neighbor  $p_h$  and its right neighbor  $p_j$  can no longer remain competitors (on behalf of  $id_i$ , and on behalf of  $id_k$ , respectively). This is because

$$id_j > \max(id_i, id_k) \Rightarrow \neg(id_i > \max(id_h, id_j)) \wedge \neg(id_k > \max(id_j, id_l)).$$

It follows that, during the second round, at most half of the processes remain competitors. During that round, the processes that are no longer competitors only forward the messages they receive, while the processes that remain competitors do the same as during the first round, except that they consider the greatest identity they have seen so far instead of their initial identity. This is illustrated in Fig. 4.8, where it is assumed that  $id_j > \max(id_i, id_k)$ ,  $id_l < \max(id_k, id_m)$ , and  $id_m > \max(id_l, id_p)$ , and consequently  $p_i$  competes on behalf of  $id_j$ , and  $p_l$  competes on behalf of  $id_m$ . The processes that remain competitors during the second round define a logical ring with at most  $n/2$  processes. This ring is denoted with dashed arrows in the figure.

Finally, the competitor processes that are winners during the second round proceed to the third round, etc., until a round with a single competitor is attained, which occurs after at most  $1 + \lceil \log_2 n \rceil$  rounds.

```

when START() is received do
(1)  $competitor_i \leftarrow true; maxid_i \leftarrow id_i; \text{send ELECTION}(1, id_i).$ 

when ELECTION(1,  $id$ ) is received do
(2) if ( $\neg competitor_i$ )
(3)   then  $\text{send ELECTION}(1, id)$ 
(4)   else if ( $id \neq maxid_i$ )
(5)     then  $\text{send ELECTION}(2, id); proxy\_for_i \leftarrow id$ 
(6)     else  $\text{send ELECTED}(id, id_i)$ 
(7)   end if
(8) end if.

when ELECTION(2,  $id$ ) is received do
(9) if ( $\neg competitor_i$ )
(10)  then  $\text{send ELECTION}(2, id)$ 
(11)  else if ( $proxy\_for_i > \max(id, maxid_i)$ )
(12)    then  $maxid_i \leftarrow proxy\_for_i; \text{send ELECTION}(1, proxy\_for_i)$ 
(13)    else  $competitor_i \leftarrow false$ 
(14)  end if
(15) end if.

when ELECTED( $id1, id2$ ) is received do
(16)  $leader_i \leftarrow id1; done_i \leftarrow true; elected_i \leftarrow (id1 = id_i);$ 
(17) if ( $id2 \neq id_i$ ) then  $\text{send ELECTED}(id1, id2)$  end if.

```

Fig. 4.9 Dolev, Klawe, and Rodeh's election algorithm (code for  $p_i$ )

#### 4.4.2 The Algorithm

As in Sect. 4.3, it is assumed that each process receives an external message START() before any message sent by the algorithm. The algorithm is described in Fig. 4.9.

**Local Variables** In addition to  $done_i$ ,  $leader_i$ , and  $elected_i$ , each process  $p_i$  manages three local variables.

- $competitor_i$  is a Boolean which indicates if  $p_i$  is currently competing on behalf of some process identity or is only relaying messages. The two other local variables are meaningful only when  $competitor_i$  is equal to *true*.
- $maxid_i$  is the greatest identity know by  $p_i$ .
- $proxy\_for_i$  is the identity of the process for which  $p_i$  is competing.

**Process Behavior** When a process receives a message START(), it initializes  $competitor_i$  and  $maxid_i$  before sending a message ELECTION(1,  $id_i$ ) on its single outgoing channel (line 1). Let us observe that messages do not carry round numbers. Actually, round numbers are used only to explain the behavior of the algorithm and compute the total number of messages.

When a process  $p_i$  receives a message ELECTION(1,  $id$ ),  $p_i$  forwards it on its outgoing channel if it is no longer a competitor (lines 2–3). If  $p_i$  is a competitor, there are two cases.

- If the message  $\text{ELECTION}(1, id_i)$  is such that  $id = \text{maxid}_i$ , then it has made a full turn on the ring, and consequently  $\text{maxid}_i$  is the greatest identity. In this case,  $p_i$  sends the message  $\text{ELECTED}(\text{maxid}_i, id_i)$  (line 6), which is propagated on the ring to inform all the processes (lines 16–17).
- If message  $\text{ELECTION}(1, id)$  is such that  $id \neq \text{maxid}_i$ ,  $p_i$  copies  $id$  in  $\text{proxy\_for}_i$ , and forwards the message  $\text{ELECTION}(2, id)$  on its outgoing channel.

When a process  $p_i$  receives a message  $\text{ELECTION}(2, id)$ , it forwards it (as previously) on its outgoing channel if it is no longer a competitor (lines 9–10). If it is a competitor,  $p_i$  checks if  $\text{proxy\_for}_i > \max(id, \text{maxid}_i)$ , i.e., if the identity of the process it has to compete for (namely,  $\text{proxy\_for}_i$ ) is greater than both  $\text{maxid}_i$  (the identity of the process on behalf of which  $p_i$  was previously competing) and the identity  $id$  it has just received (line 11). If it is the case,  $p_i$  updates  $\text{maxid}_i$  and starts a new round (line 12). Otherwise,  $\text{proxy\_for}_i$  is not the highest identity. Consequently, as  $p_i$  should compete for an identity that cannot be elected, it stops competing (line 13).

### 4.4.3 Discussion: Message Complexity and FIFO Channels

**Message Complexity** During each round, except the last one, each process sends two messages: a message  $\text{ELECTION}(1, -)$  and a message  $\text{ELECTION}(2, -)$ . Moreover, there are only  $\text{ELECTION}(1, -)$  messages during the last round. As we have seen, there are at most  $\log_2 n + 1$  rounds. It follows that the number of messages  $\text{ELECTION}(1, -)$  and  $\text{ELECTION}(2, -)$  sent by the algorithm is at most  $2n \log n + n$ .

**Type of Channels** Each process receives alternately a message  $\text{ELECTION}(1, -)$  followed by a message  $\text{ELECTION}(2, -)$ . As the channels are FIFO, it follows that the numbers 1 and 2 used to tag the messages are useless: A message  $\text{ELECTION}()$  needs to carry only a process identifier, and consequently, there are only  $n$  different messages  $\text{ELECTION}()$ .

## 4.5 Two Particular Cases

**Leader Election in an Arbitrary Network** To elect a leader in a connected arbitrary network, one can use network traversal algorithms such as those described in Chap. 1.

Each process launches a network traversal with feedback, and all the messages related to the network traversal launched by a process  $p_i$  are tagged with its identity  $id_i$ . Moreover, each process continues participating only in the network traversal it has seen that has been launched with the greatest identity. It follows that a single network traversal will terminate, namely the one launched by the process with the greatest identity.

```

(1)  $rd_i \leftarrow \text{random}(1, n)$ ; broadcast  $\text{RANDOM}(rd_i)$ ;
(2) wait(a message  $\text{RANDOM}(rd_x)$  from each process  $p_x$ );
(3)  $\text{elected}_i \leftarrow (\sum_{x=1}^n rd_x) \bmod n + 1$ .

```

**Fig. 4.10** Index-based randomized election (code for  $p_i$ )

**When the Indexes Are the Identities** When the identity of a process is its index, and both this fact and  $n$  are known by all processes, the leader election problem is trivial. It is sufficient to statically select an index and define the corresponding process as the leader. While this works, it has a drawback, namely, the same process is always elected.

There is a simple way to solve this issue, as soon as the processes can use random numbers. Let  $\text{random}(1, n)$  be a function that returns a random number in  $\{1, \dots, n\}$  each time it is called.

The algorithm described in Fig. 4.10 is a very simple randomized election algorithm. Each process first obtains a random number and sends it to all. Then, it waits until it has received all random numbers. When this occurs, the processes can compute the same process identity, and consistently elect the corresponding process (line 3). The costs of the algorithm are  $O(1)$  time units and  $O(n^2)$  messages.

The probability that a given process  $p_x$  is elected can be computed from the specific probability law associated with the function  $\text{random}()$ .

## 4.6 Summary

This chapter was devoted to election algorithms on a ring. After a simple proof that no such algorithm exists in anonymous ring networks, the chapter presented three algorithms for non-anonymous rings. Non-anonymous means here that (a) each process  $p_i$  has an identity  $id_i$ , (b) no two processes have the same identity, (c) identities can be compared, (d) initially, a process knows only its identity, and (e) no process knows  $n$  (total number of processes).

Interestingly, this chapter has presented two  $O(n \log n)$  elections algorithms that are optimal. The first, due to Hirschberg and Sinclair, is suited to bidirectional rings, while the second, due to Dolev, Klawe, and Rodeh, is suited to both unidirectional rings and bidirectional rings (this is because a bidirectional ring can always be considered as if it was unidirectional). This algorithm shows that, contrary to what one could think, the fact that a ring is unidirectional or bidirectional has no impact on its optimal message complexity when considering a  $O()$  point of view.

## 4.7 Bibliographic Notes

- The impossibility of solving the leader election problem in anonymous rings is due to D. Angluin [19].

- The general problem of what can be computed in anonymous networks is addressed in [22, 392, 393].
- The distributed election problem and its first solution are due to G. Le Lann [232].
- The  $O(n^2)$  election algorithm presented in Sect. 4.2 is due to E.J.H. Chang and R. Roberts [83].
- The  $O(n \log n)$  election algorithm for bidirectional rings presented in Sect. 4.3 is due to D.S. Hirschberg and J.B. Sinclair [185].
- The  $O(n \log n)$  election algorithm for unidirectional rings presented in Sect. 4.4 is due to D. Dolev, M. Klawe, and M. Rodeh [117]. Simultaneously and independently, a similar algorithm was presented by G.L. Peterson [295]. Another leader election algorithm for rings is presented in [134].
- Leader election algorithms in complete networks are studied in [208]. Numerous leader election algorithms suited to various network topologies (e.g., trees) are presented by N. Santoro in [335].
- Several authors have shown that  $\Omega(n \log n)$  is a lower bound on the number of messages in both ring networks and complete networks (e.g., [24, 56, 288]).
- The best election algorithm on unidirectional ring networks known so far is due to L. Higham and T. Przytycka [184]. Its message complexity is  $1.271 n \log n + O(n)$ . However, it is not yet known what is the smallest constant  $c$  such that an election can be solved on unidirectional rings with message complexity  $c \times n \log n + O(n)$  (it is only known that  $c \geq 0.69$  [56]).

Higham and Przytycka's algorithm is based on rounds and assumes that all processes start with the same initial round number. An extension of this algorithm is presented in [20], which allows the processes to start with arbitrary round numbers. This extension, which is motivated by fault-tolerance with respect to initial values, guarantees that the message complexity is  $O(n \log n)$  when the processes start with the same round number, and may increase up to  $O(n^2)$  when each process starts with an arbitrary round number. This fault-tolerance property is called *graceful degradation* with respect to initial values.

- Leader election in dynamic networks and mobile ad hoc networks is addressed in [197, 244].

## 4.8 Exercises and Problems

1. Extend the proof of Theorem 2 so that it works for any anonymous regular network.
2. Considering the variant of Chang and Robert's election algorithm described in Fig. 4.3, and assuming that  $k$  processes send an ELECTION() message at line 1, what is the maximal number of ELECTION() messages sent (at lines 1 and 2) during an execution of the algorithm?

Answer:  $nk - \frac{k(k-1)}{2}$ .

3. Show that the worst case for the time complexity of Hirschberg and Sinclair's election algorithm is when  $n$  is 1 more than a power of 2. Show that, in this case, the time complexity is  $6n - 6$ .

Solution in [185].

# Chapter 5

## Mobile Objects Navigating a Network

A mobile object is an object that, according to requests issued by user processes, travels from process to process. This chapter is on algorithms that allow a mobile object to navigate a network. It presents three distributed navigation algorithms with different properties. All these algorithms ensure both that the object remains always consistent (i.e., it is never present simultaneously at several processes), and that any process that requires the object eventually obtains it.

**Keywords** Adaptive algorithm · Distributed queuing · Edge/link reversal · Mobile object · Mutual exclusion · Network navigation · Object consistency · Routing · Scalability · Spanning tree · Starvation-freedom · Token

### 5.1 Mobile Object in a Process Graph

#### 5.1.1 Problem Definition

A mobile object is an object (such as a file or a data structure) that can be accessed sequentially by different processes. Hence, a mobile object is a concurrent object that moves from process to process in a network of processes.

When a process momentarily owns a mobile object, the process can use the object as if it was its only user. It is assumed that, after using a mobile object, a process eventually releases it, in order that the object can move to another process that requires it. So what has to be defined is a navigation service that provides the processes with two operations denoted `acquire_object()` and `release_object()` such that any use of the object by a process  $p_i$  is bracketed by an invocation to each of these operations, namely

`acquire_object();`   use of the object by  $p_i$ ;   `release_object();`

As already noticed, in order that the state of the object remains always consistent, it is required that the object be accessed by a single process at a time, and the object has to be live in the sense that any process must be able to obtain the mobile object. This is captured by the two classical safety and liveness properties which instantiate as follows for this problem (where the sentences “the object belongs to a process”

or “a process owns the object” means that the object is currently located at this process):

- Safety: At any time, the object belongs to at most one process.
- Liveness: Any process that invokes `acquire_object()` eventually becomes the owner of the object.

Let us notice that a process  $p_i$  invokes `release_object()` after having obtained and used the object. Hence, it needs to invoke again `acquire_object()` if it wants to use the mobile object again.

### 5.1.2 Mobile Object Versus Mutual Exclusion

The particular case where the ownership of the mobile object gives its owner a particular right (e.g., access to a resource) is nothing more than an instance of the mutual exclusion problem. In that case, the mobile object is a stateless object, which is usually called a *token*. The process that has the token can access the resource, while the other processes cannot. Moreover, any process can require the token in order to be eventually granted the resource.

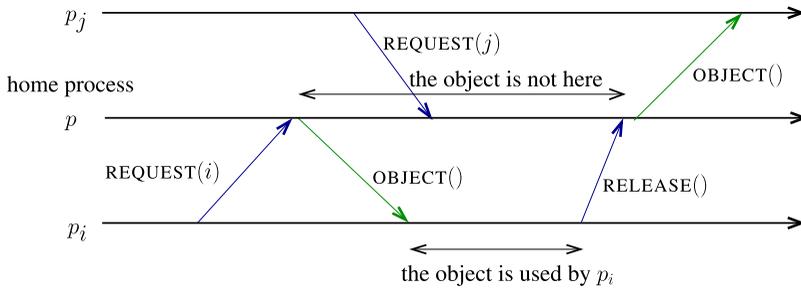
Token-based mutual exclusion algorithms define a family of distributed mutual exclusion algorithms. We will see in Chap. 10 another family of distributed mutual exclusion algorithms, which are called *permission-based* algorithms.

### 5.1.3 A Centralized (Home-Based) Algorithm

A simple solution to the navigation of a mobile object consists in using a *home-based* structure by statically associating a fixed process  $p$  with the mobile object. Such a home-based scheme is easy to implement. When it is not used, the object resides at its home process  $p$ , and a three-way handshake algorithm is used to ensure that any process that invokes `acquire_object()` eventually obtains the object.

**Three-Way Handshake Algorithm** The three-way handshake algorithm works as follows. Its name comes from the three messages used to satisfy an object request.

- When a process  $p_i$  invokes `acquire_object()`,  $p_i$  sends a message `REQUEST( $i$ )` to the home process  $p$ , and waits for the object.
- When the home process receives a message `REQUEST( $i$ )` from a process  $p_i$ , it adds this message in a local queue and sends back the object to  $p_i$  if this request is at the head of the queue.
- When  $p_i$  receives the object, it uses the object, and eventually invokes `release_object()`. This invocation entails sending a message `RELEASE_OBJECT( $i$ )` to the home process  $p$ . Moreover, if the object has been updated by  $p_i$ , the `RELEASE_OBJECT( $i$ )` message carries the last value of the object as modified by  $p_i$ .



**Fig. 5.1** Home-based three-way handshake mechanism

- When the home process receives a message  $RELEASE\_OBJECT(i)$ , it stores the new value of the object (if any), and suppresses the request of  $p_i$  from its local queue. Then, if the queue is not empty,  $p$  sends the object to the first process of the queue.

This three-way handshake algorithm is illustrated in Fig. 5.1, with two processes  $p_i$  and  $p_j$ . When the home process  $p$  receives the message  $REQUEST(j)$ , it adds it to its local queue, which already contains  $REQUEST(i)$ . Hence, the home process will answer this request when it receives the message  $RELEASE\_OBJECT(i)$ , which carries the last value of the object as modified by  $p_i$ .

**Discussion** Let us first observe that the home process  $p$  can manage its internal queue on a FIFO basis or use another priority discipline. This actually depends on the application.

While they may work well in small systems, the main issue of home-based algorithms lies in their poor ability to cope with scalability and locality. If the object is heavily used, the home process can become a bottleneck. Moreover, always returning the object to its home process can be inefficient (this is because when a process releases the object, it could be sent to its next user without passing through its home).

### 5.1.4 The Algorithms Presented in This Chapter

The algorithms that are described in this chapter are not home-based and do not suffer the previous drawback. They all have the following noteworthy feature: If, when a process  $p_i$  releases the object, no other process wants to acquire it, the object remains at its last user  $p_i$ . It follows that, if the next user is  $p_i$  again, it does not need to send messages to obtain the object. Consequently, no message is needed in this particular case.

The three algorithms that are presented implicitly consider that the home of the object is dynamically defined: the home of the mobile object is its last user. These

algorithms differ in the structure of the underlying network they assume. As we will see, their cost and their properties depend on this structure. They mainly differ in the way the mobile object and the request messages are routed in the network.

## 5.2 A Navigation Algorithm for a Complete Network

This section presents a navigation algorithm which assumes a complete network (any pair of processes is connected by an asynchronous channel). The channels are not required to be FIFO. This algorithm was proposed independently by G. Ricart and A.K. Agrawala (1983), and I. Suzuki and T. Kasami (1985). The system is made up of  $n$  processes,  $p_1, \dots, p_n$  and the index  $i$  is used as the name of  $p_i$ .

### 5.2.1 Underlying Principles

As there is no statically defined home notion, the main issue that has to be solved is the following: When a process  $p_i$  releases the object, to which process has it to send the mobile object?

**A Control Data Inside the Mobile Object** First, the mobile object is enriched with a control data, denoted  $obtained[1..n]$ , such that  $obtained[i]$  counts the number of times that  $p_i$  has received the object. Let us notice that it is easy to ensure that, for any  $i$ ,  $obtained[i]$  is modified only when  $p_i$  has the object. It follows that the array  $obtained[1..n]$  always contains exact values (and no approximate values). This array is initialized to  $[0, \dots, 0]$ .

**Local Data Structure** In order to know which processes are requesting the object, a process  $p_i$ , which does not have the object and wants to acquire it, sends a message  $REQUEST(i)$  to every other process to inform them that it is interested in the object.

Moreover, each process  $p_i$  manages a local array  $request\_by_i[1..n]$  such that  $request\_by_i[j]$  contains the number of  $REQUEST()$  messages sent by  $p_j$ , as know by  $p_i$ .

**Determining Requesting Processes** Let  $p_i$  be the process that has the mobile object. The set of processes that, from its point of view, are requesting the token can be locally computed from the arrays  $request\_by_i[1..n]$  and  $obtained[1..n]$ . It is the set  $S_i$  including the processes  $p_k$  such that (to  $p_i$ 's knowledge) the number of  $REQUEST()$  messages sent by  $p_k$  is higher than the number of times  $p_k$  has received the object (which is saved in  $obtained[k]$ ), i.e., it is the set

$$S_i = \{k \mid request\_by_i[k] > obtained[k]\}.$$

This provides  $p_i$  with a simple predicate ( $S_i \neq \emptyset$ ) that allows it to know if processes are requesting the mobile object. This predicate can consequently be used to

ensure that, if processes want to acquire the object, eventually there are processes that obtain it (deadlock-freedom property).

**Ensuring Starvation-Freedom** Let us consider the case where all processes have requested the object and  $p_1$  has the object. When,  $p_1$  releases the object, it sends it  $p_2$ , and just after sends a message REQUEST(1) to again obtain the object. It is possible that, when it releases the object,  $p_2$  sends it to  $p_1$ , and just after sends a message REQUEST(2) to again obtain the object. This scenario can repeat forever, and, while processes  $p_1$  and  $p_2$  repeatedly forever obtain the mobile object, the other processes never obtain it.

A simple way to solve this problem (and consequently obtain the starvation-freedom property) consists in establishing an order on the processes of  $S_i$  that  $p_i$  has to follow when it releases the object. The order, which depends on  $i$ , is the following one for  $p_i$ :

$$i + 1, i + 2, \dots, n, 1, \dots, i - 1.$$

The current owner of the object  $p_i$  sends the object to the first process of this list that belongs to  $S_i$ . This means that, if  $(i + 1) \in S_i$ ,  $p_i$  sends the object to  $p_{i+1}$ . Otherwise, if  $(i + 2) \in S_i$ ,  $p_i$  sends the object to  $p_{i+2}$ , etc. It is easy to see that, as no process can be favored, no process can be missed, and, consequently, any requesting process will eventually receive the mobile object.

## 5.2.2 The Algorithm

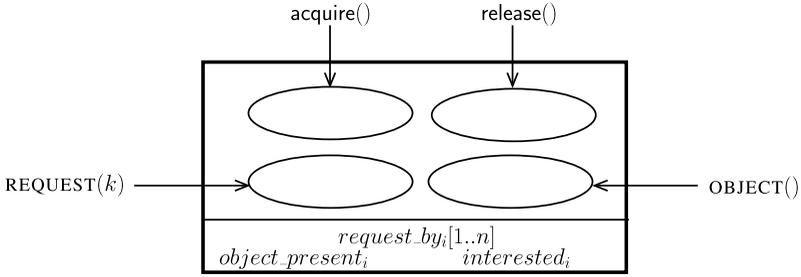
**Additional Local Variables and Initialization** In addition to the array  $request\_by_i[1..n]$  (which is initialized to  $[0, \dots, 0]$ ), each process  $p_i$  manages the following local variables:

- $interested_i$  is a Boolean initialized to *false*. It is set to *true* when  $p_i$  is interested in the object (it is waiting for it or is using it).
- $object\_present_i$  is a local Boolean variable, which is true if and only if the object is present at process  $p_i$ . Initially, all these Boolean variables are equal to *false*, except at the process where the object has been initially deposited.

It is important to notice that, in addition to the fact that processes have distinct names (known by all of them), the initialization is asymmetric: the mobile object is initially present at a single predetermined process  $p_j$ .

**Structural View** The structural view of the navigation algorithm at each process is described in Fig. 5.2. The module associated with each process  $p_i$  contains the previous local variables and four pieces of code, namely:

- The two pieces of code for the algorithms implementing the operations `acquire_object()` and `release_object()`, which constitute the interface with the application layer.



**Fig. 5.2** Structural view of the navigation algorithm (module at process  $p_i$ )

- Two additional pieces of code, each associated with the processing of a message, namely the message `REQUEST()` and the message `OBJECT()`. As we have seen, the latter contains the mobile object itself plus the associated control data *obtained*[1..n].

**The Algorithm** The navigation algorithm is described in Fig. 5.3. Each piece of code is executed atomically, except the `wait()` statement. This means that, at each process  $p_i$ , the execution of lines 1–4, lines 7–14, line 15, and lines 16–19, are mutually exclusive. Let us notice that these mutual exclusion rules do not prevent a process  $p_i$ , which is currently using the mobile object, from being interrupted to execute lines 16–19 when it receives a message `REQUEST()`.

When a process  $p_i$  invokes `acquire_object()`, it first sets *interested<sub>i</sub>* to true (line 1). If it is the last user of the object, (*object\_present<sub>i</sub>* is then true, line 2),  $p_i$  returns from the invocation and uses the object. If it does not have the object,  $p_i$  increases *request\_by<sub>i</sub>* (line 3), sends a `REQUEST(i)` message to each other process (line 4), and waits until it has received the object (lines 5 and 15).

When a process which was using the object invokes `release_object()`, it first indicates that it is no longer interested in the mobile object (line 7), and updates the global control variable *obtained*[*i*] to the value *request\_by<sub>i</sub>*[*i*] (line 8). Then, starting from  $p_{i+1}$  (line 9),  $p_i$  looks for the first process  $p_k$  that has requested the mobile object more often than the number of times it acquired the object (line 10) (let us notice that  $p_i$  is then such that  $\neg \textit{interested}_i \wedge \textit{object\_present}_i$ ).

If there is such a process,  $p_i$  sends it the object (lines 11–12) and returns from the invocation of `release_object()`. If, to its local knowledge, no process wants the object,  $p_i$  keeps it and returns from the invocation of `release_object()` (let us notice that  $p_i$  is then such that  $\neg \textit{interested}_i \wedge \textit{object\_present}_i$ ).

As already seen,  $p_i$  sets *object\_present<sub>i</sub>* to the value *true* when it receives the object. A process can receive the mobile object only if it has previously sent a request message to obtain it.

Finally, when a process  $p_i$  receives a message `REQUEST(k)`, it first increases accordingly *request\_by<sub>i</sub>*[*k*] (line 16). Then, if it has the object and is not using it,  $p_i$  sends it to  $p_k$  by return (lines 17–19).

```

operation acquire_object() is
(1) interestedi ← true;
(2) if ( $\neg$  object_presenti) then
(3)   request_byi[i] ← request_byi[i] + 1;
(4)   for  $k \in \{1, \dots, n\} \setminus \{i\}$  do send REQUEST(i) to  $p_k$  end for;
(5)   wait (object_presenti)
(6) end if.

operation release_object() is
(7) interestedi ← false;
(8) obtained[i] ← request_byi[i];
(9) for  $k$  from  $i + 1$  to  $n$  and then from 1 to  $i - 1$  do
(10)  if (request_byi[k] > obtained[k]) then
(11)    object_presenti ← false;
(12)    send OBJECT() to  $p_k$ ; exit loop
(13)  end if
(14) end for.

when OBJECT() is received do
(15) object_presenti ← true.

when REQUEST( $k$ ) is received do
(16) request_byi[k] ← request_byi[k] + 1;
(17) if (object_presenti  $\wedge$   $\neg$  interestedi) then
(18)  object_presenti ← false; send OBJECT() to  $p_k$ 
(19) end if.

```

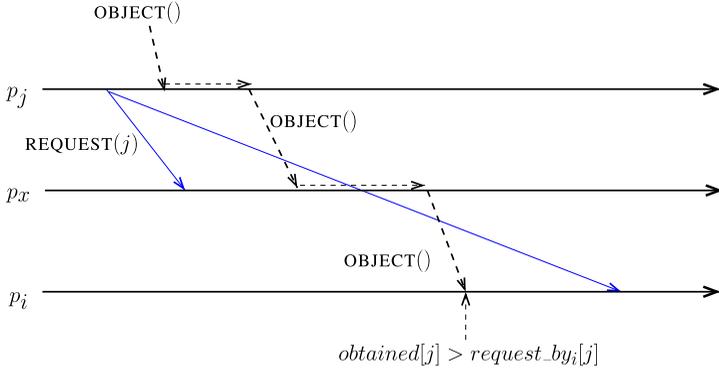
**Fig. 5.3** A navigation algorithm for a complete network (code for  $p_i$ )

**Cost of the Algorithm** The number of messages needed for one use of the mobile object is 0 when the object is already present at the process that wants to use it, or  $n - 1$  request messages plus the message carrying the object).

A message REQUEST() carries a process identity. Hence, its size is  $O(\log_2 n)$ .

The time complexity is 0 when the object is already at the requesting process. Otherwise, let us first observe that the transit time used by request messages that travel while the object is used has not to be counted. This is because, whatever their speed, as the object is used, these transit times cannot delay the transfer of the object. Assume that all messages take one time unit; it follows that the time complexity lies between 1 and 2. One time unit happens in heavy load: When a process  $p_i$  releases the object, the test of line 10 is satisfied, and the object takes one time unit before being received by its destination process. Two times units are needed in light load: One time unit is needed for the request message to attain the process that has the object, and one more time unit is needed for the object to travel to the requesting process (in that case, the object is sent at line 18).

**Are Early Updates Good?** When a process  $p_i$  receives the object (which carries the control data *obtained*[1.. $n$ ]), it is possible that some entries  $k$  are such that *obtained*[ $k$ ] > *request\_by<sub>i</sub>[k]*. This is due to asynchrony, and is independent of whether or not the channels are FIFO. This occurs when some request messages



**Fig. 5.4** Asynchrony involving a mobile object and request messages

are very slow, as depicted in Fig. 5.4, where the path followed by the mobile object is indicated with dashed arrows (the dashed arrow on a process is the period during which this process uses the mobile object). Moreover, as shown in the figure, the  $REQUEST()$  message from  $p_j$  to  $p_i$  is particularly slow with respect to the object.

Hence the question: when a process  $p_i$  receives the object, is it interesting for  $p_i$  to benefit from the array  $obtained[1..n]$  to update its local array  $request\_by_i[1..n]$ , i.e., to execute at line 15 the additional statement

**for**  $k \in \{1, \dots, n\}$  **do**  $request\_by_i[k] \leftarrow \max(request\_by_i[k], obtained[k])$  **end for**.

Due to the fact that each  $REQUEST()$  message has to be counted exactly once, this early update demands other modifications so that the algorithm remains correct. To that end, line 4 has to be replaced by

(4') **for**  $k \in \{1, \dots, n\} \setminus \{i\}$  **do** send  $REQUEST(i, request\_by_i[i])$  to  $p_k$  **end for**,

and, when a message  $REQUEST(k, rnb)$  is received, line 16 has to be replaced by

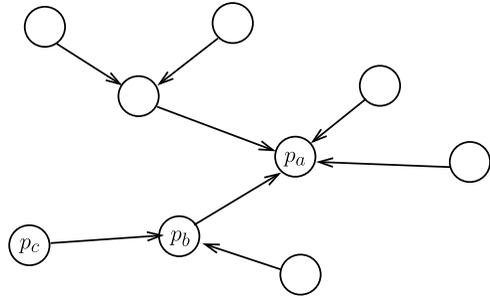
(16')  $request\_by_i[k] \leftarrow \max(request\_by_i[k], rnb)$ .

It follows that trying to benefit from the array  $obtained[1..n]$  carried by the mobile object to early update the local array  $request\_by_i[1..n]$  requires us to add (a) sequence numbers to  $REQUEST()$  messages, and (b) associated update statements. It follows that these modifications make the algorithm less efficient and a little bit more complicated. Hence, contrarily to what one could a priori hope, early updates are not good for this navigation algorithm.

### 5.3 A Navigation Algorithm Based on a Spanning Tree

The previous navigation algorithm is based on the broadcast of requests and uses sequence numbers, which can increase forever. In contrast, the algorithm presented

**Fig. 5.5** Navigation tree:  
initial state



in this section has only bounded variables. It is based on a statically defined spanning tree of the network, and each process communicates only with its neighbors in the tree, hence the algorithm is purely local. This algorithm is due to K. Raymond (1989).

### 5.3.1 Principles of the Algorithm: *Tree Invariant and Proxy Behavior*

As just indicated, the algorithm considers a statically defined spanning tree of the network. Only the channels of this spanning tree are used by the algorithm.

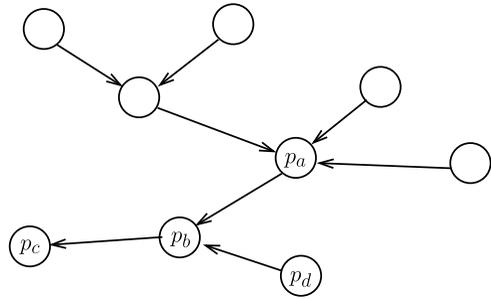
**Tree Invariant** Initially, the process that has the mobile object is the root of the tree, and each process has a pointer to its neighbor on its path to the root. This is depicted in Fig. 5.5, where the mobile object is located at process  $p_a$ .

This tree structure is the invariant maintained by the algorithm: A process always points to its neighbor in the subtree containing the object. In that way, a process always knows in which part of the tree the object is located.

Let us consider the process  $p_c$  that wants to acquire the object. Due to the tree orientation, it can send a request to its current parent in the tree, namely process  $p_b$ , which in turn can forward it to its parent, etc., until it attains the root of the tree. Hence, the tree orientation allows requests to be propagated to the appropriate part of the tree. When  $p_a$  receives the request, it can send the object to  $p_b$ , which forwards it to  $p_c$ . The object consequently follows the same path of the tree (in the reverse order) as the one the request. Moreover, in order to maintain the invariant associated with the tree orientation, the mobile object reverses the direction of the edges during its travel from the process where it was previously located ( $p_a$  in the figure) to its destination process ( $p_c$  in the figure). This technique, which is depicted in Fig. 5.6, is called *edge reversal*.

**The Notion of a Proxy** Let us consider process  $p_b$  that receives a request for the mobile object from its neighbor  $p_d$  after it has received a request from  $p_c$  (Fig. 5.6). Hence, it has already sent a request to  $p_a$  to obtain the object on behalf of  $p_c$ . Moreover, a process only knows its neighbors in the tree. Additionally, after having

**Fig. 5.6** Navigation tree:  
after the object has moved to  $p_c$



received requests from its tree neighbors  $p_c$  and  $p_d$ ,  $p_b$  may require the object for itself. How do solve these conflicts between the requests by  $p_c$ ,  $p_d$ , and  $p_b$ ?

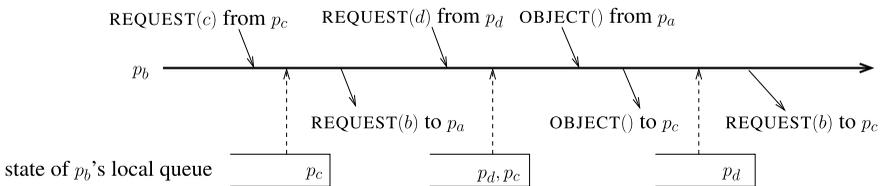
To that end,  $p_b$  manages a local FIFO queue, which is initially empty. When it receives the request from  $p_c$ , it adds the request to the queue, and as the queue contains a single request, it plays a proxy role, namely, it sends a request for itself to its parent  $p_a$  (but this request is actually for the process  $p_c$ , which is at the head of its local queue). This is a consequence of the fact that a process knows only its neighbors in the tree ( $p_a$  does not know  $p_c$ ).

Then, when it receives the request from  $p_d$ ,  $p_b$  adds to its queue, and, as this request is not the only one in the queue,  $p_b$  does not send another request to  $p_a$ . When later  $p_b$  receives the mobile object, it forwards it to the process whose request is at the head of its queue (i.e.,  $p_c$ ), and suppresses its request from the queue. Moreover, as the queue is not empty,  $p_b$  continues to play its proxy role: it sends to  $p_c$  a request (for the process at the head of its queue, which is now  $p_d$ ). This is depicted in Fig. 5.7, where the successive states of the local queue of  $p_b$  are explicitly described.

### 5.3.2 The Algorithm

The structure of the algorithm is the same as the one described in Fig. 5.2.

**Local Variables and Initialization** In addition to the Boolean variable *interested<sub>i</sub>*, whose meaning and behavior are the same as in the algorithm described in Fig. 5.3, each process  $p_i$  manages the following local variables.



**Fig. 5.7** Navigation tree: proxy role of a process

- $queue_i$  is the local queue in which  $p_i$  saves the requests it receives from its neighbors in the tree, or from itself. Initially,  $queue_i$  is empty. If  $p_i$  has  $d$  neighbors in the tree,  $queue_i$  will contain at most  $d$  process identities (one per incoming edge plus one for itself). It follows that the bit size of  $queue_i$  is bounded by  $d \log_2 n$ .

The following notations are used:

- $queue_i \leftarrow queue_i + \langle j \rangle$  means “add  $j$  at the end of  $queue_i$ ”.
- $queue_i \leftarrow queue_i - \langle j \rangle$  means “withdraw  $j$  from  $queue_i$  (which is at its head)”.
- $head(queue_i)$  denotes the first element of  $queue_i$ .
- $|queue_i|$  denotes the size of  $queue_i$ , while  $\emptyset$  is used to denote the empty queue.
- $parent_i$  contains the identity of the parent of  $p_i$  in the tree. The set of local variables  $\{parent_i\}_{1 \leq i \leq n}$  is initialized in such a way that they form a tree rooted at the process where the object is initially located. The root of the tree is the process  $p_k$  such that  $parent_k = k$ .

The Boolean  $object\_present_i$  used in the previous algorithm is no longer needed. This is because it is implicitly encoded in the local variable  $parent_i$ . More precisely, we have  $(parent_i = i) \equiv object\_present_i$ .

**Behavior of a Process** The algorithm is described in Fig. 5.8. When a process  $p_i$  invokes `acquire_object()`, it first sets  $interested_i$  to the value *true* (line 1), and checks if it has the object (predicate  $parent_i = i$ , line 2). If it has not the object, it adds its request to its local queue (line 3), and sends a request to its current parent in the spanning tree if  $queue_i$  contains only its own request (line 4). Then,  $p_i$  waits until it is at the head of its queue and receives the object (lines 5, 20, and 21).

When, after it has used the mobile object, a process  $p_i$  invokes `release_object()`, it first resets  $interested_i$  to the value *false* (line 7), and then checks the state of its local queue (line 8). If  $queue_i$  is not empty,  $p_i$  sends the object to its neighbor in the tree that is at the head of  $queue_i$ , and defines it as the new parent in the tree. Let us notice that, in that way,  $parent_i$  points to the neighbor of  $p_i$  in the subtree containing the object. Finally, if  $queue_i$  is not empty,  $p_i$  sends a request to its new parent (line 11) so that the object eventually returns to it to satisfy this request. (Let us remark that, if two messages are sent at lines 10 and 11, they can be sent as a single message carrying both the object and a request.)

When a process  $p_i$  receives a message `REQUEST()` from a neighbor  $p_k$  in the spanning tree, ( $p_i$  is then the parent of  $p_k$  in the tree), its behavior depends on whether or not it has the object. If  $p_i$  has the object and is using it, it enqueues the request (lines 13–14). If  $p_i$  has the object and it is not using it ( $p_i$  was then the last user of the object), it sends the object to  $p_k$ , and defines  $p_k$  as its new parent in the spanning tree (line 15). If  $p_i$  does not have the object when it receives a message `REQUEST( $k$ )`, it adds  $k$  to its request queue (line 17), and, as a proxy, sends a request message to its parent if the request it has received is the only request in its queue (line 18, which is the same as line 4).

Finally, let us observe that a process can receive the object only if it has previously sent a request. This means that, when a process  $p_i$  receives the mobile object, we necessarily have  $queue_i \neq \emptyset$ . Hence, when it receives the object, a process  $p_i$

```

operation acquire_object() is
(1)  interestedi  $\leftarrow$  true;
(2)  if (parenti  $\neq$  i) then
(3)    queuei  $\leftarrow$  queuei +  $\langle i \rangle$ ;
(4)    if ( $|queue_i| = 1$ ) then send REQUEST(i) to pparenti end if;
(5)    wait (parenti = i)
(6)  end if.

operation release_object() is
(7)  interestedi  $\leftarrow$  false;
(8)  if (queuei  $\neq$   $\emptyset$ ) then
(9)    let k = head(queuei); queuei  $\leftarrow$  queuei -  $\langle k \rangle$ ;
(10)   send OBJECT() to pk; parenti  $\leftarrow$  k;
(11)   if (queuei  $\neq$   $\emptyset$ ) then send REQUEST(i) to pparenti end if
(12) end if.

when REQUEST(k) is received do
(13) if (parenti = i)
(14)   then if (interestedi) then queuei  $\leftarrow$  queuei +  $\langle k \rangle$ 
(15)   else send OBJECT() to pk; parenti  $\leftarrow$  k
(16)   end if
(17) else queuei  $\leftarrow$  queuei +  $\langle k \rangle$ ;
(18)   if ( $|queue_i| = 1$ ) then send REQUEST(i) to pparenti end if
(19) end if.

when OBJECT() is received do
(20) let k = head(queuei); queuei  $\leftarrow$  queuei -  $\langle k \rangle$ ;
(21) if (i = k) then parenti  $\leftarrow$  i
(22)   else send OBJECT() to pk; parenti  $\leftarrow$  k;
(23)   if (queuei  $\neq$   $\emptyset$ ) then send REQUEST(i) to pparenti end if
(24) end if.

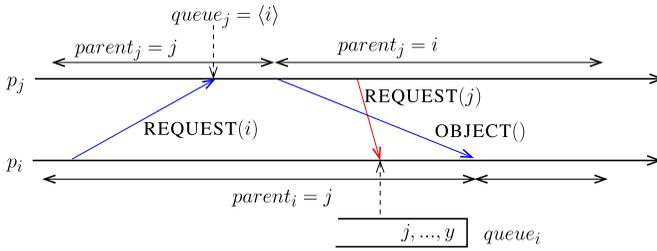
```

**Fig. 5.8** A spanning tree-based navigation algorithm (code for  $p_i$ )

considers the request at the head of  $queue_i$  (line 20). If it is its own request, the object is for it, and consequently  $p_i$  becomes the new root of the spanning tree and is the current user of the object (line 21). Otherwise, the object has to be forwarded to the neighbor  $p_k$  whose request was at the head of  $queue_i$  (line 22), and  $p_i$  has to send a request to its new parent  $p_k$  if  $queue_i \neq \emptyset$  (line 23). (Let us observe that lines 22–23 are the same as lines 10–11.)

### 5.3.3 Discussion and Properties

**On Messages** As shown in Figs. 5.5 and 5.6, it follows that, during the object’s travel in the tree to its destination process, the object reverses the direction of the tree edges that have been traversed (in the other direction) by the corresponding request messages.



**Fig. 5.9** The case of non-FIFO channels

Each message  $REQUEST()$  carries the identity of its sender (lines 4, 11, 18, 23). Thus, the control information carried by these messages is bounded. Moreover, all local variables are bounded.

**Non-FIFO Channels** As in the previous one, this algorithm does not require the channels to be FIFO. When channels are not FIFO, could a problem happen if a process  $p_i$ , which has requested the object for it or another process  $p_k$ , first receives a request and then the object, both from one of its neighbor  $p_j$  in the tree?

This scenario is depicted in Fig. 5.9, where we initially have  $parent_i = j$ , and  $p_i$  has sent the message  $REQUEST(i)$  to its parent on behalf of the process  $p_y$  at the head of  $queue_i$  (hence  $|queue_i| \geq 1$ ). Hence, when  $p_i$  receives  $REQUEST(j)$ , it is such that  $parent_i \neq i$  and  $|queue_i| \geq 1$ . It then follows from lines 13–19 that the only statement executed by  $p_i$  is line 17, namely,  $p_i$  adds  $j$  to its local queue  $queue_i$ , which is then such that  $|queue_i| \geq 2$ .

**On the Tree Invariant** The channels used by the algorithm constitute an undirected tree. When considering the orientation of these channels as defined by the local variable  $parent_i$ , we have the following.

Initially, these variables define a directed tree rooted at the process where the object is initially located. Then, according to requests and the move of the object, it is possible that two processes  $p_i$  and  $p_j$  are such that  $parent_i = j$  and  $parent_j = i$ . Such is the case depicted in Fig. 5.9, in which the message  $REQUEST(j)$  can be suppressed (let us observe that this is independent of whether or not the channel connecting  $p_i$  and  $p_j$  is FIFO). When this occurs we necessarily have:

- $parent_j = i$  and  $p_j$  has sent the object to  $p_i$ , and
- $parent_i = j$  and  $p_i$  has sent the message  $REQUEST(i)$  to  $p_j$ .

When the object arrives at  $p_i$ ,  $parent_i$  is modified (line 21 or line 22, according to the fact that  $p_i$  is or is not the destination of the object), and, from then on, the edge of the spanning tree is directed from  $p_j$  to  $p_i$ .

Hence, let us define an abstract spanning tree as follows. The orientation of all its edges, except the one connecting  $p_i$  and  $p_j$  when the previous scenario occurs, are defined by the local variables  $parent_k$  ( $k \neq i, j$ ). Moreover, when the previous scenario occurs, the edge of the abstract spanning tree is from  $p_j$  to  $p_i$ , i.e., the

direction in which the object is moving. (This means that the abstract spanning tree logically considers that the object has arrived at  $p_i$ .) The invariant preserved by the algorithm is then the following: At any time there is exactly one abstract directed spanning tree, whose edges are directed to the process that has the object, or to which the object has been sent.

**Cost of the Algorithm** As in the previous algorithm, the message cost is 0 messages in the most favorable case. The worst case depends on the diameter  $D$  of the tree, and occurs when the two most distant processes are such that one has the object and the other requests it. In this case,  $D$  messages and  $D$  time units are necessary for the requests to arrive at the owner of the object, and again  $D$  messages and  $D$  time units are necessary for the object to arrive at the requesting process. Hence, both message complexity and time complexity are  $O(D)$ .

**On Process Identities** As any process  $p_i$  communicates only with its neighbors in the spanning tree, the only identities that can appear in  $queue_i$  are the identities of these neighbors plus its own identity  $i$ . So that no two identities in  $queue_i$  can be confused, the algorithm requires them to be different.

As the identities of any two processes  $p_i$  and  $p_j$  that are at a distance greater than 2 in the tree can never appear in the same queue, it follows that any two processes at a distance greater than 2 in the tree can have the same name without making the algorithm incorrect. This noteworthy property is particularly interesting from a scalability and locality point of view.

**On Priority** As each local queue  $queue_i$  is managed according to the FIFO discipline, it follows that each request is eventually granted (the corresponding process obtains the mobile object). It is possible to apply other management rules to these queues, or particular rules to a subset of processes, in order to favor some processes, or obtain specific priority schemes. This versatility dimension of the algorithm can be exploited by some applications.

### 5.3.4 Proof of the Algorithm

**Theorem 3** *The algorithm described in Fig. 5.8 guarantees that the object is never located simultaneously at several sites (safety), and any process that wants the object eventually acquires it (liveness).*

*Proof* Proof of the safety property. Let us first observe that the object is initially located at a single process. The proof is by induction. Let us assume that the property is true up to the  $x$ th move of the object from a process to another process. Let  $p_i$  be the process that has the object after its  $x$ th move. If  $p_i$  keeps the object forever, the property is satisfied. Hence, let us assume that eventually  $p_i$  sends the object. There are three lines at which  $p_i$  can send the object. Let us consider each of them.

- $p_i$  invokes `release_object()`. In this case, we have  $i \notin queue_i$ . This is because when, after its last `acquire_object()`,  $p_i$  received the object, it was at the head of  $queue_i$  but it suppressed its name from  $queue_i$  before using the object (lines 20–21).

If  $queue_i = \emptyset$ ,  $p_i$  keeps the object and the property is satisfied. If  $queue_i \neq \emptyset$ ,  $p_i$  sends the object to the process  $p_k$  whose request is at the head of  $queue_i$  (and  $k \neq i$ , due to the previous observation). It also sets  $parent_i$  to  $k$  (lines 7–10). We have then  $parent_i = k \neq i$ , which means that the object is no longer at  $p_i$ , and the property is satisfied.

- $p_i$  executes line 15. In this case, we have  $k \neq i$  at line 15 (this is because only  $p_i$  can send messages `REQUEST(i)`, and a process does not send request messages to itself). It follows that, after it has executed line 15, we have  $parent_i \neq i$ , i.e.,  $p_i$  no longer has the object, and the safety property is satisfied.
- $p_i$  executes line 22. In this case, we have  $i \neq k$  (line 21). Hence,  $parent_i = k \neq i$  after  $p_i$  has forwarded the object, and the safety property is satisfied.

Proof of the liveness property. This proof is made up of two parts. It is first proved that, if processes want to acquire the object, one process obtains it (deadlock-freedom). It is then proved that any process that wants to acquire the object, eventually obtains it (starvation-freedom).

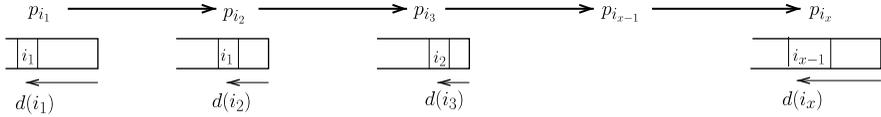
Proof of the deadlock-freedom property. Let us consider that the object is located at  $p_i$ , and at least one process  $p_j$  has sent a request. Hence,  $p_j$  has sent a message `REQUEST(j)` to  $parent_j$ , which in turn has sent a message `REQUEST(parent_j)` to its parent, etc. (these request messages are not necessarily due to the request of  $p_i$  if other processes have issued requests). The important point is that, due to orientation of the edges and the loop-freedom property of the abstract dynamic spanning tree,  $p_i$  receives a message `REQUEST(k)` (this message is not necessarily due to the  $p_j$ 's request). When this occurs, if  $interested_i$ ,  $p_i$  adds  $k$  to  $queue_i$  (the important point is here that  $queue_i \neq \emptyset$ , line 14), and will send the object to the head of  $queue_i$  when it will execute line 10 of the operation `release_object()`. If  $\neg interested_i$  when  $p_i$  receives `REQUEST(k)`, it sends by return the object to  $p_k$  (line 15). Hence, whatever the value of  $interested_i$ ,  $p_i$  eventually sends the object to  $p_k$ . If  $k$  is at the head of  $queue_k$ , it has requested the object and obtains it. Otherwise,  $p_k$  forwards the object to the process  $p_{k'}$  such that  $k'$  is at the head of  $queue_k$ . Iterating this reasoning, and considering the orientation property of the abstract spanning tree, it follows that the object attains a process that has a pending request. It follows that the algorithm is deadlock-free.

Proof of the starvation-freedom property. Let  $D$  be the diameter of the spanning tree,  $1 \leq D \leq n - 1$ . Let  $p_i$  be process that has sent a message `REQUEST(i)` and  $p_j$  the process that has the mobile object. Let us associate the following vector  $R$  with the request of  $p_i$  (see Fig. 5.10):

$$R = [d(i_1), d(i_2), \dots, d(i_{x-1}), d(i_x), 0, \dots, 0],$$

where  $i_1 = i$ ,  $i_x = j$ ,  $N(i_y)$  is the degree of  $p_{i_y}$  in the spanning tree, and

- $d(i_1)$  is the rank of  $i_1$  in  $queue_{i_1} \equiv queue_i$  ( $1 \leq d(i_1) \leq N(i_1)$ ),



**Fig. 5.10** The meaning of vector  $R = [d(i_1), d(i_2), \dots, d(i_{x-1}), d(i_x), 0, \dots, 0]$

- $d(i_2)$  is the rank of  $i_1$  in  $queue_{i_2}$  ( $1 \leq d(i_2) \leq N(i_2)$ ),
- $d(i_3)$  is the rank of  $i_2$  in  $queue_{i_3}$  ( $1 \leq d(i_3) \leq N(i_3)$ ), etc.,
- $d(i_x)$  is the rank of  $i_{x-1}$  in  $queue_{i_x} \equiv queue_j$  ( $1 \leq d(i_x) \leq N(i_x)$ ).

Let us observe that, as  $R$  has  $D$  entries and each entry has a bounded domain, it follows that the set of all possible vectors  $R$  is bounded and totally ordered (when considering lexicographical ordering).

Moreover, as each local queue is a FIFO queue and a process cannot issue a new request while its previous request is still pending, it follows that, when  $p_x$  sends the object (this necessarily occurs from the deadlock-freedom property), we proceed from the vector  $R$  to the vector

- $R' = [d(i_1), d(i_2), \dots, d(i_{x-1}) - 1, 0, \dots, 0]$  if  $d(i_x) = 1$ , or
- $R'' = [d(i_1), d(i_2), \dots, d(i_{x-1}), d(i_x) - 1, *, \dots, *]$  if  $d(i_x) > 1$ .

The first case ( $d(i_x) = 1$ ) is when  $p_x$  sends the object to  $p_{i_{x-1}}$  (which is at the head of its queue), while the second case ( $d(i_x) > 1$ ) is when a process different from  $p_{i_{x-1}}$  is at the head of  $queue_{i_x}$  (where the “\*” stands for appropriate values according to the current states the remaining  $(D - x)$  local queues).

The important point is that we proceed from vector  $R$  to a vector  $R'$  or  $R''$ , which is smaller than  $R$  according to the total order on the set of possible vectors. Hence, starting from  $R'$  or  $R''$ , we proceed to another vector  $R'''$  (smaller than  $R'$  or  $R''$ ), etc., until we attain the vector  $[1, 0, \dots, 0]$ . When this occurs  $p_i$  has the object, which concludes the proof of the starvation-freedom property.  $\square$

## 5.4 An Adaptive Navigation Algorithm

This section presents a distributed algorithm that implements a distributed FIFO queue. A process invokes the operation `enter()` to append itself at the end of the queue. Once it has become the head of the queue, it invokes the operation `exit()` to leave the queue.

This algorithm, which was proposed by M. Naimi and M. Trehel (1987), assumes both a complete asynchronous network, and an underlying spanning tree whose structure evolves according to the requests to enter the queue issued by the processes.

From a mobile object point of view, this means that a process invokes `enter()` (i.e., `acquire_object()`) to obtain the object, and this operation terminates when the invoking process is at the head of the queue. After using the object, a process invokes

exit() (i.e., release\_object()) to exit the queue and allows its successor in the queue to become the new head of the queue.

### 5.4.1 The Adaptivity Property

A distributed algorithm implementing a distributed object (or a distributed service) is *adaptive*, if, for each process that is not interested in using the object (or the service), there is a finite time after which these processes no longer have to participate in the algorithm (hence, they no longer receive messages sent by the algorithm).

As an example, let us consider the two algorithms described in Figs. 5.3 and 5.8, which implement a navigation service for a mobile object. None of them is adaptive. Let  $p_x$  be a process that, after some time, never invokes the operation acquire\_object(). In the algorithm of Fig. 5.3,  $p_x$  receives all the messages REQUEST() sent by the other processes. In the algorithm of Fig. 5.8, according to its position in the spanning tree,  $p_x$  has to play a proxy role for the requests issued by its neighbors in the tree, and, in the other direction, the object has to pass through  $p_x$  to attain its destination. Hence, in both cases,  $p_x$  has to forever participate in the navigation algorithm.

As we are about to see, the algorithm described in this section is adaptive: If after some time  $\tau$ , a process  $p_x$  never invokes enter(), there is a finite time  $\tau' \geq \tau$ , after which  $p_x$  it is not required to participate in the distributed algorithm.

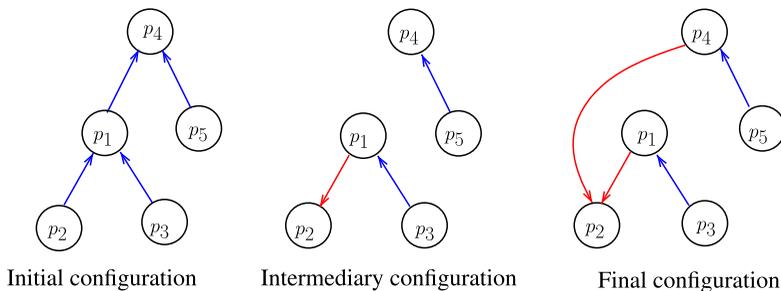
### 5.4.2 Principle of the Implementation

**A Distributed Queue** To implement the queue, each process  $p_i$  manages two local variables. The Boolean *interested<sub>i</sub>* is set to the value *true* when  $p_i$  starts entering the queue, and is reset to the value *false* when it exits the queue. The second local variable is denoted *next<sub>i</sub>*, and is initialized to  $\perp$ . It contains the successor of  $p_i$  in the queue. Moreover, *next<sub>i</sub>* =  $\perp$  if  $p_i$  is the last element of the queue.

Hence, starting from the process  $p_x$  that is the head of the queue, the queue is defined by the sequence of pointers *next<sub>x</sub>*, *next<sub>next<sub>x</sub></sub>*, etc., until the process  $p_y$  such that *next<sub>y</sub>* =  $\perp$ .

Due to its very definition, there is a single process at the head of the queue. Hence, the algorithm considers that this is the process at which the mobile object is currently located. When this process  $p_i$  exits the queue, it has to send the mobile object to  $p_{next_i}$  (if *next<sub>i</sub>*  $\neq \perp$ ).

**How to Enter the Queue: A Spanning Tree to Route Messages** Let  $p_\ell$  be the process that is currently the last process of the queue (hence, *interested<sub>ℓ</sub>*  $\wedge$  *next<sub>ℓ</sub>* =  $\perp$ ). The main issue that has to be solved consists in the definition of an addressing mechanism that allows a process  $p_i$ , that wants to enter the queue, to inform  $p_\ell$  that it is no longer the last process in the queue and it has a successor.



**Fig. 5.11** A dynamically evolving spanning tree

To that end, we need a distributed routing structure that permits any process to send a message to the last process of the queue. Moreover, this distributed routing structure has to be able to evolve, as the last process of the queue changes according to the request issued by processes. The answer is simple, namely, the distributed routing structure we are looking for is a dynamically evolving spanning tree whose current root is the last process of the queue.

More specifically, let us consider Fig. 5.11. There are five processes,  $p_1, \dots, p_5$ . The local variable  $parent_i$  of each process  $p_i$  points to the process that is the current parent of  $p_i$  in the tree. As usual,  $parent_x = x$  means that  $p_x$  is the root of the tree. The process  $p_4$  is the current root (initial configuration). The process  $p_2$  sends a message REQUEST(2) to its parent  $p_1$  and defines itself as new root by setting  $parent_2$  to 2. When  $p_1$  receives this message, as it is not the root, it forwards this message to its parent  $p_4$  and redefines its new parent as being  $p_2$  (intermediary configuration). Finally, when  $p_4$  receives the message REQUEST(2) forwarded by  $p_1$ , it discovers that it has a successor in the queue (hence, it executes  $next_4 \leftarrow 2$ ), and it considers  $p_2$  as the new root of the spanning tree (update of  $parent_4$  to 2 in the final configuration).

As shown by the previous example, it is possible that, at some times, several trees coexist (each spanning a distinct partition of the network). The important point is that there is no creation of cycles, and there is a single new spanning tree when all control messages have arrived and been processed.

Differently from the algorithm described in Fig. 5.8, which is based on “edge reversal” on a statically defined tree, this algorithm benefits from the fact any channel can be part of the dynamically evolving spanning tree. The directed path  $p_2, p_1, p_4$  of the initial spanning tree, is replaced by two new directed edges (from  $p_1$  to  $p_2$ , and from  $p_4$  to  $p_2$ ). Hence, a path (in the initial configuration) of length  $d$  from the new root to the old root is replaced by  $d$  edges, each directly pointing to the new root (in the final configuration).

**Heuristic Used by the Algorithm** The previous discussion has shown that the important local variable for a process  $p_i$  to enter the queue is  $parent_i$ . It follows from the modification of the edges of the spanning tree, which are entailed by the

messages REQUEST(), that the variable  $parent_i$  points to the process  $p_k$  that has issued the last request seen by  $p_i$ .

Hence, when  $p_i$  wants later to enter the queue, it sends its request to the process  $p_{parent_i}$ , because this is the last process in the queue from  $p_i$ 's point of view.

**The Case of the Empty Queue** As in the diffusion-based algorithm of Fig. 5.3, each process  $p_i$  manages a local Boolean variable  $object\_present_i$ , whose value is *true* if and only if the mobile object is at  $p_i$  (if we are not interested in the navigation of a mobile object, this Boolean could be called  $first\_in\_queue_i$ ).

As far as the management of the queue is concerned, its implementation has to render an account of the case where the queue is empty. To that end, let us consider the case where the queue contains a single process  $p_i$ . This process is consequently the first and the last process of the queue, and we have then  $object\_present_i \wedge (parent_i = i)$ .

If  $p_i$  exits the queue, the queue becomes empty. The previous predicate remains satisfied, but we have then  $\neg interested_i$ . It follows that, if a process  $p_i$  is such that  $(\neg interested_i \wedge object\_present_i) \wedge (parent_i = i)$ , it knows that it was the last user of the queue, which is now empty.

### 5.4.3 An Adaptive Algorithm Based on a Distributed Queue

**Structure of the Algorithm and Initialization** The structure of the algorithm is the same as in both previous algorithms (see Fig. 5.2). In order to be as generic as possible, and allow for an easy comparison with the previous algorithms, we continue using the operation names `acquire_object()` and `release_object()`, instead of `enter()` and `exit()`, respectively.

Initially, the object is located at some predetermined process  $p_k$ . This is a process that is fictitiously placed at the head of the distributed empty queue (see the predicate associated with an empty queue). The local variables are then initialized as follows:

- $object\_present_k$  is initialized to *true*,
- $\forall i \neq k$ :  $object\_present_i$  is initialized to *false*,
- $\forall i$ :  $parent_i$  is initialized to  $k$ , and  $interested_i$  is initialized to *false*.

**Behavior of a Process** The algorithm is described in Fig. 5.12. As in the previous algorithms, the four parts of code (except the **wait** statement at line 4) are mutually exclusive.

When a process invokes `acquire_object()` (i.e., when it wants to enter the queue), it first sets  $interested_i$  to the value *true* (line 1). If  $object\_present_i$  is equal to *false*,  $p_i$  sends the message REQUEST( $i$ ) to its parent in order to be added at the end of the queue, and defines itself as the new end of the queue (line 3). Then, it waits until it has obtained the object, i.e., it is the head of the queue (lines 4 and 10).

If  $object\_present_i$  is equal to *true*,  $p_i$  is at the head of the queue (it has the object). This is due to the following observation. Just before invoking `acquire_object()` to enter the queue (obtain the object),  $p_i$  was such that  $(\neg interested_i) \wedge object\_present_i$ ,

```

operation acquire_object() is
(1)   $interested_i \leftarrow true$ ;
(2)  if ( $\neg object\_present_i$ ) then
(3)    send REQUEST( $i$ ) to  $p_{parent_i}$ ;  $parent_i \leftarrow i$ ;
(4)    wait( $object\_present_i$ )
(5)  end if.

operation release_object() is
(6)   $interested_i \leftarrow false$ ;
(7)  if ( $next_i \neq \perp$ ) then
(8)    send OBJECT() to  $p_{next_i}$ ;  $object\_present_i \leftarrow false$ ;  $next_i \leftarrow \perp$ 
(9)  end if.

when OBJECT() is received do
(10)  $object\_present_i \leftarrow true$ .

when REQUEST( $k$ ) is received do
(11) if ( $parent_i \neq i$ )
(12)   then send REQUEST( $k$ ) to  $p_{parent_i}$ 
(13)   else if ( $interested_i$ ) then  $next_i \leftarrow k$ 
(14)   else send OBJECT() to  $p_k$ ;  $object\_present_i \leftarrow false$ 
(15)   end if
(16) end if;
(17)  $parent_i \leftarrow k$ .

```

**Fig. 5.12** A navigation algorithm based on a distributed queue (code for  $p_i$ )

from which we conclude that, the last time  $p_i$  has invoked `release_object()`, it was such that  $next_i = \perp$  (line 7), and no other process has required the object (otherwise,  $p_i$  would have set  $object\_present_i$  to `false` at line 14). It follows that the object remained at  $p_i$  since the last time it used it (i.e., the queue was empty and  $p_i$  was its fictitious single element).

When  $p_i$  invokes `release_object()`, it first resets  $interested_i$  to the value `false`, i.e., it exits the queue (line 6). Then, if  $next_i = \perp$ ,  $p_i$  keeps the object. Otherwise, it first sends the object to  $p_{next_i}$  (i.e., it sends it the property “you are the head of the queue”), and then resets  $object\_present_i$  to `false`, and  $next_i$  to  $\perp$  (line 8).

When  $p_i$  receives a message `REQUEST( $k$ )`, its behavior depends on the fact that it is, or it is not, the last element of the queue. If it is not, it has only to forward the request message to its current parent in the spanning tree, and redefine  $parent_i$  as being  $p_k$  (lines 11–12 and 17). This is to ensure that  $parent_i$  always points to the last process that, to  $p_i$ 's knowledge, has issued a request. Let us remark that the message `REQUEST( $k$ )` that is forwarded is exactly the message received. This ensures that the variables  $parent_x$  of all the processes  $p_x$  visited by this message will point to  $p_k$ .

If  $p_i$  is such that  $parent_i = \perp$ , there are two cases. If  $interested_i$  is true,  $p_i$  is in the queue. It consequently adds  $p_k$  to the queue (line 13). If  $interested_i$  is false, we are in the case where the queue is empty ( $p_i$  is its fictitious single element). In this case,  $p_i$  has the object and sends it to  $p_k$  (line 14). Moreover, whatever the case, to preserve its correct meaning ( $p_{parent_i}$  is the last process that, to  $p_i$  knowledge, has issued a request),  $p_i$  updates  $parent_i$  to  $k$  (line 17).

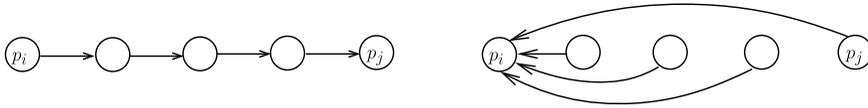


Fig. 5.13 From the worst to the best case

### 5.4.4 Properties

**Variable and Message Size, Message Complexity** A single bit is needed for each local variable  $object\_present_i$  and  $interested_i$ ;  $\lceil \log_2 n \rceil$  bits are needed for each variable  $parent_i$ , while  $\lceil \log_2(n + 1) \rceil$  bits are needed for each variable  $next_i$ . As each message  $REQUEST()$  carries a process identity, it needs to  $\lceil \log_2 n \rceil$  bits. If the algorithm is used only to implement a queue, the message  $OBJECT()$  does not have to carry application data, and a single bit is needed to distinguish the two message types.

In the best case, an invocation of  $acquire\_object()$  does not generate  $REQUEST()$  messages, while it gives rise to  $(n - 1)$   $REQUEST()$  messages plus one  $OBJECT()$  message in the worst case. This case occurs when the spanning tree is a chain.

The left side of Fig. 5.13 considers the worst case: The process denoted  $p_i$  issues a request, and the object is at the process denoted  $p_j$ . The right side of the figure shows the spanning tree after the message  $REQUEST(i)$  has visited all the processes. The new spanning tree is optimal in the sense that the next request by a process will generate only one request message. More generally, it has been shown that the average number of messages generated by an invocation of  $acquire\_object()$  is  $O(\log n)$ .

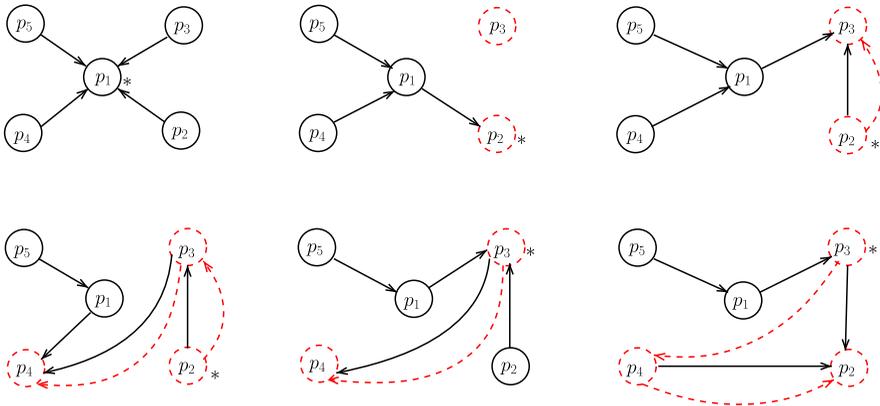
**Adaptivity** The adaptivity property concerns each process  $p_i$  taken individually. It has been defined as follows: If after some time  $p_i$  never invokes  $acquire\_object()$ , there is a finite time after which it does not receive  $REQUEST()$  messages, and it consequently stops participating in the algorithm.

Considering such a process  $p_i$ , let  $\tau$  be a time after which all the messages  $REQUEST(i)$  that have been sent have been received and processed. Moreover, let  $\{k_1, k_2, \dots\}$  be the set of identities of the processes whose parent is  $p_i$  at time  $\tau$  (this set of processes is bounded).

Let us first observe that, as  $p_i$  never sends a message  $REQUEST(i)$  at any time after  $\tau$ , it follows from line 17 that no process  $p_y$ ,  $y \in \{1, \dots, n\} \setminus \{k_1, k_2, \dots\}$  can be such that  $parent_y = i$  after time  $\tau$ .

Moreover, let us assume that after time  $\tau$ ,  $p_i$  forwards a message  $REQUEST(k)$  it has received from some process  $p_x$ ,  $x \in \{k_1, k_2, \dots\}$ . It follows from line 17 that we have  $parent_x = k$  after  $p_x$  has forwarded the message  $REQUEST(k)$  to  $p_i$ . Hence, in the future,  $p_i$  will no longer receive messages  $REQUEST()$  from  $p_x$ . As this is true for any process  $p_x$ ,  $x \in \{k_1, k_2, \dots\}$ , and this set is bounded, it follows that there is a time  $\tau' \geq \tau$  after which  $p_i$  will no longer receive  $REQUEST()$  messages.

This adaptivity property is particularly interesting from a scalability point of view. Intuitively, it means that if, during a long period, some processes do not invoke



**Fig. 5.14** Example of an execution

ACQUIRE\_OBJECT(), they are “ignored” by the algorithm whose cost is accordingly reduced.

### 5.4.5 Example of an Execution

An example of an execution of the algorithm in a system of five processes,  $p_1, \dots, p_5$  is described in Fig. 5.14. The initial configuration is depicted on the left of the top line of the figure. The plain arrows represent the spanning tree (pointer  $parent_i$ ), while the dashed arrows represent the queue (pointer  $next_i$ ). The processes that are in the queue, or are entering it, are represented with dashed circles. The star represents the object; it is placed close to the process at which it is currently located.

Both the processes  $p_2$  and  $p_3$  invoke `acquire_object()` and send to their parent  $p_1$  the message `REQUEST(2)` and `REQUEST(3)`, respectively. Let us assume that the message `REQUEST(2)` is the one that arrives first at  $p_1$ . This process consequently sends the object to  $p_2$ . We obtain the structure depicted in the middle of the top line of the figure (in which the message `REQUEST(3)` has not yet been processed by  $p_1$ ).

When  $p_1$  receives the message `REQUEST(3)`, it forwards it to  $p_2$ , and defines  $p_3$  as its new parent in the tree. Then, when  $p_2$  receives the message `REQUEST(3)`, it defines  $p_3$  as its new parent (for its next requests), and sets  $next_2$  to point to  $p_3$ . The queue including  $p_2$  and  $p_3$  is now formed (right of the top line of the figure).

Then,  $p_4$  invokes `acquire_object()`. It sends the message `REQUEST(4)` to its parent  $p_1$ , and considers itself the new root. When  $p_1$  receives this message, it forwards `REQUEST(4)` to its previous parent  $p_3$ , and then updates its pointer  $parent_1$  to 4. Finally, when  $p_3$  receives `REQUEST(4)`, it updates accordingly  $parent_3$  and  $next_3$ , which now point to  $p_4$  (left of the bottom line of the figure).

Let us assume that  $p_2$  invokes `release_object()`. It consequently leaves the queue by sending the object to the process pointed to by  $next_2$ , (i.e.,  $p_3$ ). This is depicted

in the middle of the bottom line of the figure. Finally, the last subfigure (right of the bottom line) depicts the value of the pointers  $parent_i$  and  $next_i$  after a new invocation of `acquire_object()` by process  $p_2$ .

As we can see, when all messages generated by the invocations of `acquire_object()` and `release_object()` have been received and processed, the sets of pointers  $parent_i$  and  $next_i$  define a single spanning tree and a single queue, respectively. Moreover, this example illustrates also the adaptivity property. When considering the last configuration, if  $p_5$  and  $p_1$  do not invoke the operation `acquire_object()`, they will never receive messages generated by the algorithm.

## 5.5 Summary

This chapter was on algorithms that allow a mobile object to navigate a network made up of distributed processes. The main issue that these algorithms have to solve lies in the routing of both the requests and the mobile object, so that any process that requires the object eventually obtains it. Three navigation algorithms have been presented. They differ in the way the requests are disseminated, and in the way the mobile object moves to its destination.

The first algorithm, which assumes a fully connected network, requires  $O(n)$  messages per use of the mobile object. The second one, which uses only the edges of a fixed spanning tree built on the process network, requires  $O(D)$  messages, where  $D$  is the diameter of the spanning tree. The last algorithm, which assumes a fully connected network, manages a spanning tree whose shape evolves according to the requests issued by the processes. Its average message cost is  $O(\log n)$ . This algorithm has the noteworthy feature of being adaptive, namely, if after some time a process is no longer interested in the mobile object, there is a finite time after which it is no longer required to participate in the algorithm.

When the mobile object is a (stateless) token, a mobile object algorithm is nothing more than a token-based mutual exclusion algorithm. Actually, the algorithms presented in this chapter were first introduced as token-based mutual exclusion algorithms.

Finally, as far as algorithmic principles are concerned, an important algorithmic notion presented in this chapter is the “edge reversal” notion.

## 5.6 Bibliographic Notes

- The diffusion-based algorithm presented in Sect. 5.2 was proposed independently by G. Ricart and A.K. Agrawala [328], and I. Suzuki and T. Kasami [361]. This algorithm is generalized to arbitrary (connected) networks in [176].
- The algorithm presented in Sect. 5.3 is due to K. Raymond [304]. This algorithm, which assumes a statically defined tree spanning the network of processes, is based on the “edge reversal” technique.

- The edge reversal technique was first proposed (as far as we know) by E. Gafni and D. Bertsekas [141]. This technique is used in [78] to solve resource allocation problems. A monograph entirely devoted to this technique has recently been published [388].
- The algorithm presented in Sect. 5.4 is due to M. Naimi and M. Trehel [275, 374]. A formal proof of it, and the determination of its average message complexity (namely,  $O(\log n)$ ), can be found in [276].
- A generic framework for mobile object navigation along trees is presented in [172]. Both the algorithms described in Sects. 5.3 and 5.4, and many other algorithms, are particular instances of this very general framework.
- Variants of the algorithms presented in Sects. 5.3 and 5.4 have been proposed. Among them are the algorithm presented by J.M. Bernabéu-Aubán and M. Ahamad [50], the algorithm proposed by M.L. Nielsen and M. Mizuno [279] (see Exercise 4), a protocol proposed by J.L.A. van de Snepscheut [355], and the *arrow* protocol proposed by M.J. Demmer and M. Herlihy [105]. (The message complexity of this last algorithm is investigated in [182].)
- The algorithm proposed by J.L.A. van de Snepscheut [355] extends a tree-based algorithm to work on any connected graph.
- Considering a mobile object which is a token, a dynamic heuristic-based navigation algorithm is described in [345]. Techniques to regenerate lost tokens are described in [261, 285, 321]. These techniques can be extended to more sophisticated objects.

## 5.7 Exercises and Problems

1. Modify the navigation algorithm described in Fig. 5.3 (Sect. 5.2), so that all the local variables  $request\_by_i[k]$  have a bounded domain  $[1..M]$ . (Hint: consider the process that is the current user of the object.)
2. The navigation algorithm described in Fig. 5.3 assumes that the underlying communication network is a complete point-to-point network. Generalize this algorithm so that it works on any connected network (i.e., a non-necessarily complete network).  
Solution in [176].
3. A *greedy* version of the spanning tree-based algorithm described in Fig. 5.8 (Sect. 5.3) can be defined as follows. When a process  $p_i$  invokes `acquire_object()` while  $parent_i \neq i$  (i.e., the mobile object is not currently located at the invoking process),  $p_i$  adds its identity  $i$  at the head of  $queue_i$  (and not at its tail as done in Fig. 5.8). The rest of the algorithm is left unchanged.

What is the impact of this modification on the order in which the processes obtain the mobile object? Does the liveness property remain satisfied? (Justify your answers.)

Solution in [304].

```

operation acquire_object() is
(1)  interestedi ← true;
(2)  if ( $\neg$  object_presenti) then
(3)    send REQUEST(i, i) to pparenti; parenti ← i;
(4)    wait(object_presenti)
(5)  end if.

when release_object() is
(6)  interestedi ← false;
(7)  if (nexti ≠  $\perp$ ) then
(8)    send OBJECT() to pnexti; object_presenti ← false; nexti ←  $\perp$ 
(9)  end if.

when OBJECT() is received do
(10) object_presenti ← true.

when REQUEST(j, k) is received do
(11) if (parenti ≠ i)
(12)   then send REQUEST(i, k) to pparenti
(13)   else if (interestedi) then nexti ← k
(14)   else send OBJECT() to pk; object_presenti ← false
(15)   end if
(16) end if;
(17) parenti ← j.

```

**Fig. 5.15** A hybrid navigation algorithm (code for  $p_i$ )

4. In the algorithm described in Fig. 5.8 (Sect. 5.3), the spanning tree is fixed, and both the requests and the object navigate its edges (in opposite direction). Differently, in the algorithm described in Fig. 5.12, the requests navigate a spanning tree that they dynamically modify, and the object is sent directly from its current owner to its next owner.

Hence the idea to design a variant of the algorithm of Fig. 5.12 in which the requests are sent along a fixed spanning (only the direction of its edges is modified according to requests), and the object is sent directly from its current user to its next user. Such an algorithm is described in Fig. 5.15. A main difference with both previous algorithms lies in the message REQUEST(). Such a message carries now two process identities  $j$  and  $k$ , where  $p_j$  is the identity of the process that sent the message, while  $p_k$  is the identity of the process from which originates this request. (In the algorithm of Fig. 5.8, which is based on the notion of a proxy process, a request message carries only the identity of its sender. Differently, in the algorithm of Fig. 5.12, a request message carries only the identity of the process from which the request originates.) The local variables have the same names (*interested<sub>i</sub>*, *object\_present<sub>i</sub>*, *parent<sub>i</sub>*, *next<sub>i</sub>*), and the same meaning as in the previous algorithms.

Is this algorithm correct? If it is not, find a counterexample. If it is, prove its correction and compute its message complexity.

Solution in [279].

## Part II

# Logical Time and Global States in Distributed Systems

This part of the book, which consists of four chapters (Chap. 6 to Chap. 9), is devoted to the concepts of event, local state, and global state of a distributed computation and associated notions of logical time. These are fundamental notions that provide application designers with sane foundations on the nature of asynchronous distributed computing in reliable distributed systems.

Chapter 6 shows how a distributed computation can be represented as a partial order on the set of events produced by the processes. It also introduces the notion of a consistent global state and presents two algorithms that compute such global states on the fly. The notion of a lattice of global states and its interest are also discussed.

Chapter 7 introduces distinct notions of logical time encountered in distributed systems, namely, linear time (also called scalar time, or Lamport's time), vector time, and matrix time. Each type of time is defined, its main properties are stated, and examples of its uses are given.

Chapter 8 addresses distributed checkpointing in asynchronous message-passing systems. It introduces the notion of communication and checkpoint pattern, and presents two consistency conditions which can be associated with such an abstraction of a distributed computation. The first one (called z-cycle-freedom) captures the absence of cycles among local checkpoints, while the second one (called rollback-dependency trackability) allows us to associate, without additional computation, a global checkpoint with each local checkpoint. Checkpointing algorithms that ensure these properties are presented.

Finally, Chap. 9, which is the last chapter of this part, presents general techniques (called synchronizers) to simulate a synchronous system on top of an asynchronous distributed system.

## Chapter 6

# Nature of Distributed Computations and the Concept of a Global State

A sequential execution can be represented by the sequence (trace) of consecutive local states it has produced, or, given its initial state, by the sequence of statements that have been executed. Hence, a question that comes naturally to mind is the following one: How do we model a distributed execution?

This chapter answers first this question. To that end, it gives basic definitions, and presents three ways to model a distributed execution, namely, a partial order on a set of events, a partial order on a set of local states, and a lattice of global states. While these three types of models are equivalent, it appears that each one is more appropriate than the others to analyze and understand specific features of distributed executions.

The chapter then focuses on a fundamental notion of distributed computing, namely, the notion of a *global state*. The chapter analyzes global states and presents several distributed algorithms, which compute on the fly global states of a distributed application. These algorithms are *observation* algorithms (they have to observe an execution without modifying its behavior). It is shown that the best that can be done is the computation of a global state in which a distributed execution has passed or could have passed. This means that no process can know if the distributed execution has passed or has not passed through the global state which is returned as a result. This noteworthy feature illustrates the relativistic nature of the observation of distributed computations. Despite this relativistic feature, the computation of such global states allows distributed computing problems to be solved (such as the detection of stable properties).

Both the terms “global state” and “snapshot” are used with the same meaning in the literature. They have to be considered as synonyms. This chapter uses the term global state.

**Keywords** Event · Causal dependence relation · Causal future · Causal path · Causal past · Concurrent (independent) events · Causal precedence relation · Consistent global state · Cut · Global state · Happened before relation · Lattice of global states · Observation · Marker message · Nondeterminism · Partial order on events · Partial order on local states · Process history · Process local state · Sequential observation

## 6.1 A Distributed Execution Is a Partial Order on Local Events

This chapter considers asynchronous systems made up of  $n$  processes  $p_1, \dots, p_n$ , where the identity of process  $p_i$  is its index  $i$ . The power of a process is that of a Turing machine enriched with point-to-point send and receive operations.

### 6.1.1 Basic Definitions

**Events** The execution of a statement by a process is called an *event*. Hence, being sequential, a process consists of a sequence of events. In a distributed system, three types of events can be distinguished.

- A communication event involves a process and a channel connecting this process to another process. There are two types of communication events:
  - A *send* event occurs when a process sends a message to another process.
  - A *receive* event occurs when a process receives a message from another process.
- The other events are called *internal* events. Such an event involves a single process and no communication channel. Its granularity depends on the observation level. It can be the execution of a subprogram, the execution of a statement expressed in some programming language, the execution of a single machine instruction, etc. The important point is that an internal event does not involve communication. Hence, any sequence of statements executed by a process between two communication events can be abstracted by a single internal event.

**Process History** As each process is sequential, the events executed by a process  $p_i$  are totally ordered. The corresponding sequence is sometimes called the *history* of  $p_i$ .

Let  $e_i^x$  and  $\widehat{h}_i$  denote the  $x$ th event produced by process  $p_i$ , and the history of  $p_i$ , respectively. We consequently have

$$\widehat{h}_i = e_i^1, e_i^2, \dots, e_i^x, e_i^{x+1}, \dots$$

From a notational point of view, we sometimes denote  $\widehat{h}_i$  as a pair  $(h_i, \rightarrow_i)$ , where  $h_i$  is the set of events produced by  $p_i$ , and  $\rightarrow_i$  is the (local) total order on the events of  $h_i$ .

### 6.1.2 A Distributed Execution Is a Partial Order on Local Events

Let  $H_i = \bigcup_{1 \leq i \leq n} h_i$  (i.e., the set of all the events produced by a distributed execution). Moreover (to simplify the presentation), let us assume that no process sends the same message twice.

**Message Relation** Let  $M$  be the set of all the messages exchanged during an execution. Considering the associated send and receive events, let us define a “message order” relation, denoted “ $\longrightarrow_{msg}$ ”, as follows. Given any message  $m \in M$ , let  $s(m)$  denote its send event, and  $r(m)$  denote its receive event. We have

$$s(m) \longrightarrow_{msg} r(m).$$

This relation expresses the fact that any message  $m$  is sent before being received.

**Distributed Computation** The flow of control of a distributed execution (computation) is captured by the smallest partial order relation, denoted  $\widehat{H}_i = (H_i, \xrightarrow{ev})$ , where  $e_i^x \xrightarrow{ev} e_j^y$  if:

- Process order:  $i = j \wedge x < y$ , or
- Message order:  $\exists m \in M: e_i^x = s(m)$  and  $e_j^y = r(m)$ , or
- Transitive closure:  $\exists e: e_i^x \xrightarrow{ev} e \wedge e \xrightarrow{ev} e_j^y$ .

“Process order” comes from the fact that each process is sequential; a process produces one event at a time. “Message order” comes from the fact that a message has first to be sent in order to be later received. Finally, “transitive closure” binds process order and message order.

The relation  $\xrightarrow{ev}$  is usually called *happened before* relation. It is also sometimes called the *causal precedence* relation. There is then a slight abuse of language as, while we can conclude from  $\neg(e_i^x \xrightarrow{ev} e_j^y)$  that the event  $e_i^x$  is not a cause of the event  $e_j^y$ , it is not possible to conclude from  $e_i^x \xrightarrow{ev} e_j^y$  that  $e_i^x$  is a cause of  $e_j^y$  (as a simple example, two consecutive events on a process are not necessarily “causally related”; they appear one after the other only because the process that issued them is sequential). Despite this approximation, we will use “causally precede” as a synonym of “happen before”. Hence,  $e_i^x \xrightarrow{ev} e_j^y$  has to be understood as “it is possible that event  $e_i^x$  causally affects event  $e_j^y$ ”.

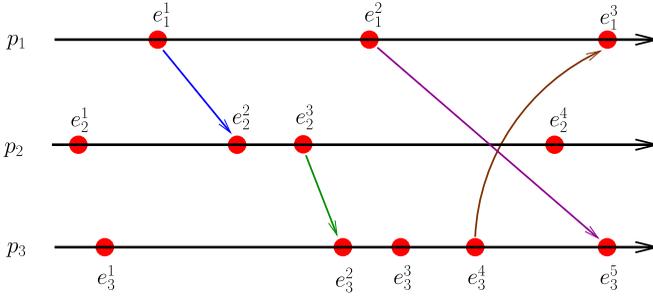
This approach to model a distributed computation as a partial order on a set of events is due to L. Lamport (1978). It is fundamental as, being free from physical time, it captures the essence of asynchronous distributed computations, providing thereby a simple way to abstract them so that we can analyze and reason on them.

An example of a distributed execution is depicted in the classical space-time diagram in Fig. 6.1. There are three processes, and each bullet represents an event;  $e_3^3$  is an internal event,  $e_1^3$  is a receive event, while  $e_2^3$  is a send event.

### 6.1.3 Causal Past, Causal Future, Concurrency, Cut

**Causal Path** A causal path is a sequence of events  $a(1), a(2), \dots, a(z)$  such that

$$\forall x : 1 \leq x < z : a(x) \xrightarrow{ev} a(x+1).$$



**Fig. 6.1** A distributed execution as a partial order

Hence, a causal path is a sequence of consecutive events related by  $\xrightarrow{ev}$ . Let us notice that each process history is trivially a causal path.

When considering Fig. 6.1, the sequence of events  $e_2^2, e_2^3, e_3^2, e_3^4, e_3^5, e_1^3$  is a causal path connecting the event  $e_2^2$  (sending by  $p_2$  of a message to  $p_3$ ) to the event  $e_1^3$  (reception by  $p_1$  of a message sent by  $p_3$ ). Let us also observe that this causal relates an event on  $p_2$  ( $e_2^3$ ) to an event on  $p_1$  ( $e_1^3$ ), despite the fact that  $p_2$  never sends a message to  $p_1$ .

**Concurrent Events, Causal Past, Causal Future, Concurrency Set** Two events  $a$  and  $b$  are *concurrent* (or *independent*) (notation  $a||b$ ), if they are not causally related, none of them belongs to the causes of the other, i.e.,

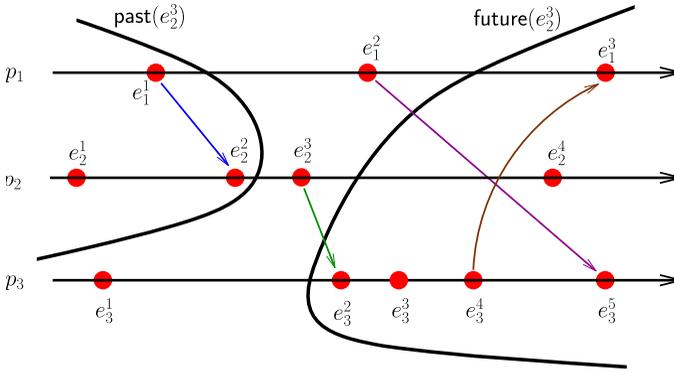
$$a||b \stackrel{def}{=} \neg(a \xrightarrow{ev} b) \wedge \neg(b \xrightarrow{ev} a).$$

Three more notions follow naturally from the causal precedence relation. Let  $e$  be an event.

- Causal past of an event:  $\text{past}(e) = \{f \mid f \xrightarrow{ev} e\}$ .  
This set includes all the events that causally precede the event  $e$ .
- Causal future of an event:  $\text{future}(e) = \{f \mid e \xrightarrow{ev} f\}$ . This set includes all the events that have  $e$  in their causal past.
- Concurrency set of an event:  $\text{concur}(e) = \{f \mid f \notin (\text{past}(e) \cup \text{future}(e))\}$ .  
This set includes all the events that are not causally related with the event  $e$ .

Examples of such sets are depicted in Fig. 6.2, where the event  $e_2^3$  is considered. The events of  $\text{past}(e_2^3)$  are the three events to the left of the left bold line, while the events of  $\text{future}(e_2^3)$  are the six events to the right of the bold line on the right side of the figure.

It is important to notice that while, with respect to physical time,  $e_3^1$  occurs “before”  $e_2^3$ , and  $e_1^2$  occurs “after”  $e_2^3$ , both are independent from  $e_2^3$  (i.e., they are logically concurrent with  $e_2^3$ ). Process  $p_1$  cannot learn that the event  $e_2^3$  has been produced before receiving the message from  $p_3$  (event  $e_1^3$ , which terminates the causal



**Fig. 6.2** Past, future, and concurrency sets associated with an event

path from starting at  $e_2^3$  on  $p_2$ ). A process can learn it only thanks to the flow of control created by the causal path  $e_2^2, e_3^2, e_3^3, e_3^4, e_3^5, e_1^3$ .

**Cut and Consistent Cut** A *cut*  $C$  is a set of events which define initial prefixes of process histories. Hence, a cut can be represented by a vector  $[\text{prefix}(\widehat{h}_1), \dots, \text{prefix}(\widehat{h}_n)]$ , where  $\text{prefix}(\widehat{h}_i)$  is the corresponding prefix for process  $p_i$ . A *consistent cut*  $C$  is a cut such that  $\forall e \in C: f \xrightarrow{ev} e \Rightarrow f \in C$ .

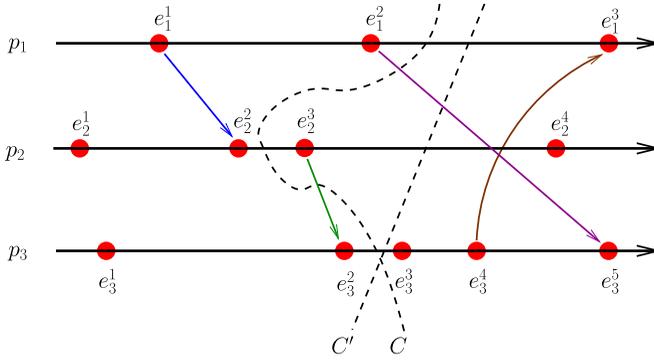
As an example,  $C = \{e_1^2, e_2^1, e_2^2, e_3^1\}$  is not a cut because the only event from  $p_1$  is  $e_1^2$ , which is not a prefix of the history of  $p_1$  (an initial prefix also has to include  $e_1^1$ , because it has been produced before  $e_1^2$ ).

$C = \{e_1^1, e_2^1, e_2^2, e_3^1, e_3^2\}$  is a cut because its events can be partitioned into initial prefix histories:  $e_1^1, e_1^2$  is an initial prefix of  $\widehat{h}_1$ ,  $e_2^1, e_2^2$  is an initial prefix of  $\widehat{h}_2$ , and  $e_3^1, e_3^2$  is an initial prefix of  $\widehat{h}_3$ . It is easy to see that the cut  $C$  is not consistent, because  $e_2^3 \notin C, e_2^3 \in C$ , and  $e_2^3 \xrightarrow{ev} e_2^2$ . Differently, the cut  $C' = C \cup \{e_2^3\}$  is consistent.

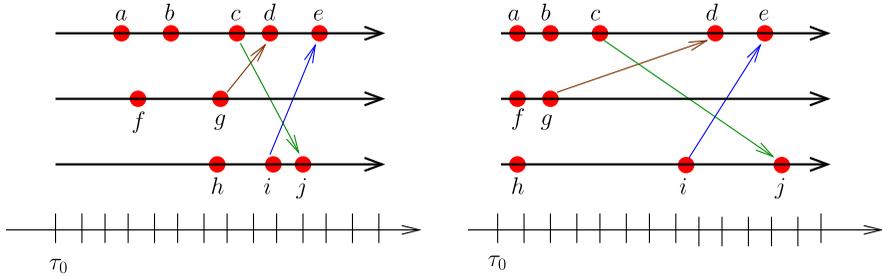
The term “cut” comes from the fact that a cut can be represented by a line separating events in a space-time diagram. The events at the left of the cut line are the events belonging to the cut. The cut  $C$  and the consistent cut  $C'$  are represented by dashed lines in Fig. 6.3.

### 6.1.4 Asynchronous Distributed Execution with Respect to Physical Time

As we have seen, the definition of a distributed execution does not refer to physical time. This is similar to the definition of a sequential execution, and is meaningful as long as we do not consider real-time distributed programs. Physical time is a resource needed to execute a distributed program, but is not a programming object



**Fig. 6.3** Cut and consistent cut



**Fig. 6.4** Two instances of the same execution

accessible to the processes. (Physical time can be known only by an omniscient global observer, i.e., an observer that is outside the computation.)

This means that the sets of distributed executions with the same set of events and the same partial order on these events are the very same execution, whatever the physical time at which the events have been produced. This is depicted in Fig. 6.4.

This means that equivalent executions are obtained when abstracting space/time diagrams (as introduced in Sect. 1.1.1) from the duration of both real-time intervals between events and message transfer delays.

The events, which are denoted  $a, \dots, j$ , are the same in both executions. As an example,  $a$  is the event abstracting the execution of the statement  $x_1 \leftarrow 0$  (where  $x_1$  is a local variable of the process  $p_1$ ),  $g$  is the sending of a given message by  $p_2$  to  $p_1$ , and  $d$  is the reception of this message by  $p_1$ .

Physical time is represented by a graduated arrow at the bottom of each execution, where  $\tau_0$  denotes the starting time of both executions. Each small segment of a graduated arrow represents one physical time unit.

This shows that the partial order on events produced by a distributed execution captures causal dependences between events, and only them. There is no notion of “duration” known by the processes. Moreover, this shows that the only way for a process to learn information from its environment is by receiving messages.

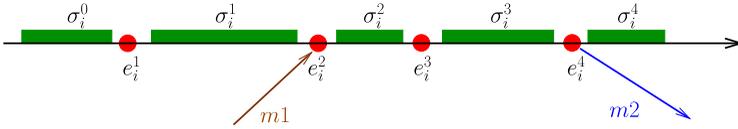


Fig. 6.5 Consecutive local states of a process  $p_i$

In an asynchronous system, the passage of time alone does not provide information to the processes. This is different in synchronous systems, where, when a process proceeds to the next synchronous round, it knows that all the processes do the same. (This point will be addressed in Chap. 9, which is devoted to synchronizers.)

## 6.2 A Distributed Execution Is a Partial Order on Local States

**From Events to Local States** Each process  $p_i$  starts from an initial local state denoted  $\sigma_i^0$ . Then, its first event  $e_i^1$  entails its move from  $\sigma_i^0$  to its next local state  $\sigma_i^1$ , and more generally, its  $x$ th event  $e_i^x$  entails its progress from  $\sigma_i^{x-1}$  to  $\sigma_i^x$ . This is depicted in Fig. 6.5, where the small rectangles denote the consecutive local states of process  $p_i$ .

We sometimes use the transition-like notation  $\sigma_i^x = \delta(\sigma_i^{x-1}, e_i^x)$  to state that the statement generating the event  $e_i^x$  makes  $p_i$  progress from the local state  $\sigma_i^{x-1}$  to the local state  $\sigma_i^x$ .

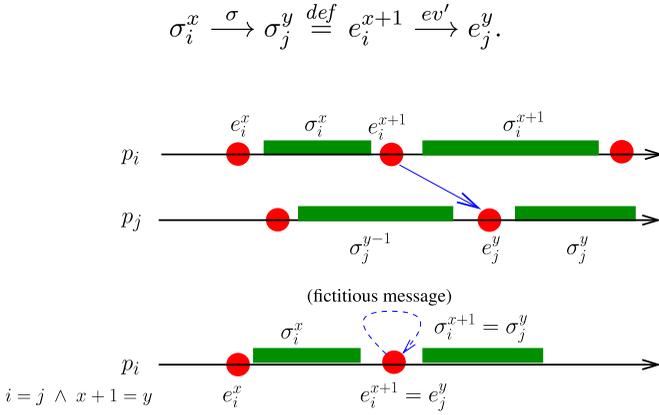
**A Slight Modification of the Relation  $\xrightarrow{ev}$**  In order to obtain a simple definition for a relation on local states, let us consider the relation on events denoted  $\xrightarrow{ev'}$ , which is  $\xrightarrow{ev}$  enriched with reflexivity. This means that the “process order” part of the definition of  $\xrightarrow{ev}$ , namely  $i = j \wedge x < y$ , is extended to  $i = j \wedge x \leq y$  (i.e., by definition, each event precedes itself).

**A Partial Order on Local States** Let  $S$  be the set of all local states produced by a distributed execution. Thanks to  $\xrightarrow{ev'}$ , we can define, in a simple way, a partial order on the elements of  $S$ . This relation, denoted  $\xrightarrow{\sigma}$ , is defined as follows

$$\sigma_i^x \xrightarrow{\sigma} \sigma_j^y \stackrel{def}{=} e_i^{x+1} \xrightarrow{ev'} e_j^y.$$

This definition is illustrated on Fig. 6.6. There are two cases.

- If  $e_i^{x+1}$  is the event associated with the sending of a message, and  $e_j^y$  the event associated with its reception, the local state  $\sigma_i^x$  preceding  $e_i^{x+1}$  “happens before” (causally precedes) the local state  $\sigma_j^y$  generated by  $e_j^y$  (top of the figure).



**Fig. 6.6** From a relation on events to a relation on local states

- If the events  $e_i^{x+1}$  and  $e_j^y$  have been produced by the same process ( $i = j$ ), and are consecutive ( $y = x + 1$ ), then the local state  $\sigma_i^x$  preceding  $e_i^{x+1} = e_j^y$  “happens before” (causally precedes) the local state  $\sigma_j^y = \sigma_i^{x+1}$  generated by  $e_j^y$  (bottom of the figure). In this case, the reflexivity ( $e_i^{x+1}$  is  $e_j^y$ ) can be interpreted as an “internal communication” event, where  $p_i$  sends to itself a fictitious message. And, the corresponding send event  $e_i^{x+1}$  and receive event  $e_j^y$  are then merged to define a single same internal event.

It follows from the definition of  $\xrightarrow{\sigma}$  that a distributed execution can be abstracted as a partial order  $\widehat{S}$  on the set of the process local states, namely,  $\widehat{S} = (S, \xrightarrow{\sigma})$ .

**Concurrent Local States** Two local states  $\sigma 1$  and  $\sigma 2$  are *concurrent* (or independent, denoted  $\sigma 1 || \sigma 2$ ) if none of them causally precedes the other one, i.e.,

$$\sigma 1 || \sigma 2 \stackrel{def}{=} \neg(\sigma 1 \xrightarrow{\sigma} \sigma 2) \wedge \neg(\sigma 2 \xrightarrow{\sigma} \sigma 1).$$

It is important to notice that two concurrent local states may coexist at the same physical time. This is, for example, the case of the local states  $\sigma_i^{x+1}$  and  $\sigma_j^{y-1}$  in the top of Fig. 6.6 (this coexistence lasts during the—unknown and arbitrary—transit time of the corresponding message). On the contrary, this can never occur for causally dependent local states (a “cause” local state no longer exists when any of its “effect” local states starts existing).

## 6.3 Global State and Lattice of Global States

### 6.3.1 The Concept of a Global State

**Global State and Consistent Global State** A global state  $\Sigma$  of a distributed execution is a vector of  $n$  local states, one per process:

$$\Sigma = [\sigma_1, \dots, \sigma_i, \dots, \sigma_n],$$

where, for each  $i$ ,  $\sigma_i$  is a local state of process  $p_i$ .

Intuitively, a *consistent global state* is a global state that could have been observed by an omniscient external observer. More formally, it is a global state  $\Sigma = [\sigma_1, \dots, \sigma_i, \dots, \sigma_n]$  such that

$$\forall i, j : i \neq j \Rightarrow \sigma_i \parallel \sigma_j.$$

This means no two local states of a consistent global state can be causally dependent.

**Global State Reachability** Let  $\Sigma_1 = [\sigma_1, \dots, \sigma_i, \dots, \sigma_n]$  be a consistent global state. The global state  $\Sigma_2 = [\sigma'_1, \dots, \sigma'_i, \dots, \sigma'_n]$  is *directly reachable* from  $\Sigma_1$  if there is a process  $p_i$  and an event  $e_i$  (representing a statement that  $p_i$  can execute in its local state  $\sigma_i$ ) such that

- $\forall j \neq i : \sigma'_j = \sigma_j$ , and
- $\sigma'_i = \delta(\sigma_i, e_i)$ .

Hence, direct reachability states that, the distributed execution being in the global state  $\Sigma_1$ , there is a process  $p_i$  that produces its next event, and this event directs the distributed execution to enter the global state  $\Sigma_2$ , which is the same as  $\Sigma_1$  except for its  $i$ th entry.

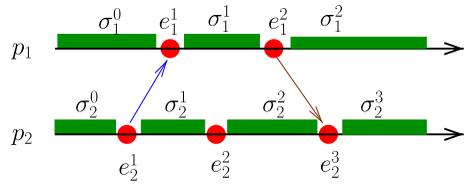
It follows that, given a consistent global state  $\Sigma_1 = [\sigma_1, \dots, \sigma_i, \dots, \sigma_n]$ , for any  $i$ ,  $1 \leq i \leq n$ , and assuming  $e_i$  exists, the consistent global state  $\Sigma_2 = [\sigma_1, \dots, \delta(\sigma_i, e_i), \dots, \sigma_n]$  is directly reachable from  $\Sigma_1$ . This is denoted  $\Sigma_2 = \delta(\Sigma_1, e_i)$ .

More generally,  $\Sigma_1$  and  $\Sigma_z$  being two consistent global states,  $\Sigma_z$  is *reachable* from  $\Sigma_1$  (denoted  $\Sigma_1 \xrightarrow{\Sigma} \Sigma_z$ ) if there is a sequence of consistent global states  $\Sigma_2, \Sigma_3$ , etc., such that, for any  $k$ ,  $1 < k \leq z$ , the global state  $\Sigma_k$  is directly reachable from the global state  $\Sigma_{k-1}$ . By definition, for any  $\Sigma_1$ ,  $\Sigma_1 \xrightarrow{\Sigma} \Sigma_1$ .

### 6.3.2 Lattice of Global States

Let us consider the very simple two-process distributed execution described in Fig. 6.7. While it is in its initial local state,  $p_1$  waits for a message from  $p_2$ . The reception of this message entails its progress from  $\sigma_1^0$  to  $\sigma_1^1$ . Then,  $p_1$  sends back an answer to  $p_2$  and moves to  $\sigma_1^2$ . The behavior of  $p_2$  can be easily deduced from the figure.

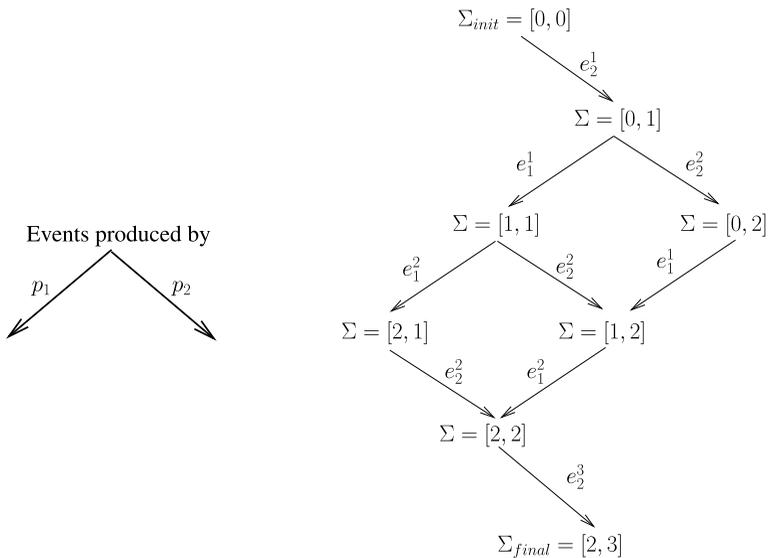
**Fig. 6.7** A two-process distributed execution



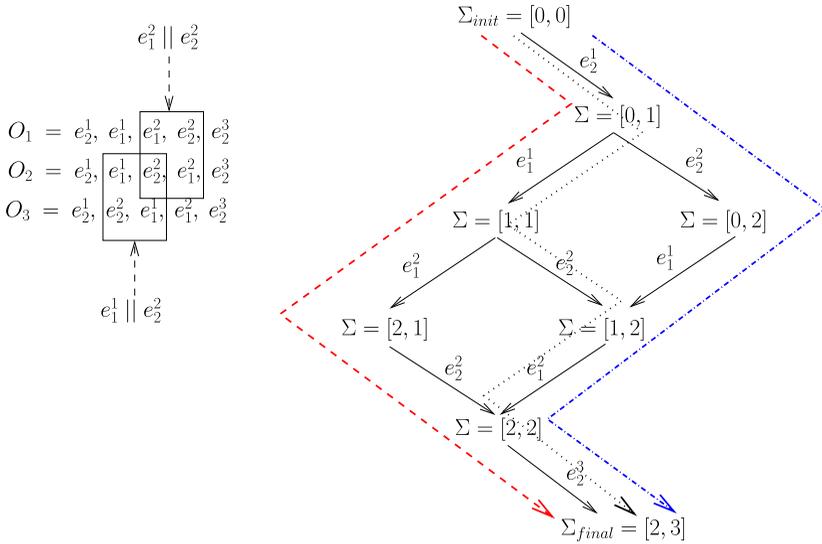
**Reachability Graph** Interestingly, it is possible to represent all the consistent global states in which a distributed execution can pass, as depicted in Fig. 6.8. To simplify the notation, the global state  $[\sigma_1^x, \sigma_2^y]$  is denoted  $[x, y]$ . Hence, the initial global state  $\Sigma_{init}$  is denoted  $[0, 0]$ , while the final global state  $\Sigma_{final}$  is denoted  $[2, 3]$ .

Starting from the initial global state, only  $p_2$  can produce an event, namely the sending of a message to  $p_1$ . It follows that a single consistent global state can be directly attained from the initial global state  $[0, 0]$ , namely the global state  $[0, 1]$ . Then, the execution being in the global state  $[0, 1]$ , both  $p_1$  and  $p_2$  can produce an event. If  $p_1$  produces  $e_1^1$ , the execution progresses from the global state  $[0, 1]$  to  $[1, 1]$ . Differently, if  $p_2$  produces its next event  $e_2^2$ , the execution progresses from the global state  $[0, 1]$  to the global state  $[0, 2]$ . Progressing this way, we can build the reachability graph including all the consistent global states that can be visited by the distributed execution depicted in Fig. 6.6.

**A Distributed Execution as a Lattice of Global States** The fact that the previous execution has two processes, generates a graph with “two dimensions”: one asso-



**Fig. 6.8** Lattice of consistent global states



**Fig. 6.9** Sequential observations of a distributed computation

ciated with the events issued by  $p_1$  (edges going from right to left in the figure), the other one associated with the events issued by  $p_2$  (edges going left to right in the figure). More generally, an  $n$ -process execution gives rise to a graph with “ $n$  dimensions”.

The reachability graph actually has a lattice structure. A *lattice* is a directed graph such that any two vertices have a unique greatest common predecessor and a unique lowest common successor. As an example, the consistent global states denoted  $[2, 1]$  and  $[0, 2]$  have several common predecessors, but have a unique greatest common predecessor, namely the consistent global state denoted  $[0, 1]$ . Similarly they have a unique lowest common successor, namely the consistent global state denoted  $[2, 2]$ . As we will see later, these lattice properties become particularly relevant when one is interested in determining if some global state property is satisfied by a distributed execution.

### 6.3.3 Sequential Observations

**Sequential Observation of a Distributed Execution** A *sequential observation* of a distributed execution is a sequence including all its events, respecting their partial ordering. The distributed execution of Fig. 6.6 has three sequential observations, which are depicted in Fig. 6.9. They are the following:

- $O_1 = e_2^1, e_1^1, e_1^2, e_2^2, e_2^3$ . This sequential observation is depicted by the dashed line at the left of Fig. 6.9.

- $O_2 = e_2^1, e_1^1, e_2^2, e_1^2, e_2^3$ . This sequential observation is depicted by the dotted line in the middle of Fig. 6.9.
- $O_3 = e_2^1, e_2^2, e_1^1, e_1^2, e_2^3$ . This sequential observation is depicted by the dashed/dotted line at the right of Fig. 6.9.

As each observation is a total order on all the events that respects their partial ordering, it follows that we can go from one observation to another one by permuting any pair of consecutive events that are concurrent. As an example,  $O_2$  can be obtained from  $O_1$  by permuting  $e_2^1$  and  $e_2^2$  (which are independent events). Similarly, as  $e_1^1$  and  $e_2^2$  are independent events, permuting them in  $O_2$  provides us with  $O_3$ .

Let us finally observe that the intersection of all the sequential observations (i.e., their common part) is nothing more than the partial order on the events associated with the computation.

**Remark 1** As each event makes the execution proceed from a consistent global state to another consistent global state, an observation can also be defined at a sequence of consistent global states, each global state being directly reachable from its immediate predecessor in the sequence. As an example,  $O_1$  corresponds to the sequence of consistent global states  $[0, 0]$ ,  $[0, 1]$ ,  $[1, 1]$ ,  $[2, 1]$ ,  $[2, 2]$ ,  $[2, 3]$ .

**Remark 2** Let us insist on the fact that the notion of global state reachability considers a single event at a time. During an execution, independent events can be executed in any order or even simultaneously. If, for example,  $e_1^1$  and  $e_2^2$  are executed “simultaneously”, the execution proceeds “directly” from the global state  $[0, 1]$  to the global state  $[1, 2]$ . Actually, whatever does really occur, it is not possible to know it. The advantage of the lattice approach, which considers one event at a time, lies in the fact that no global state in which the execution could have passed is missed.

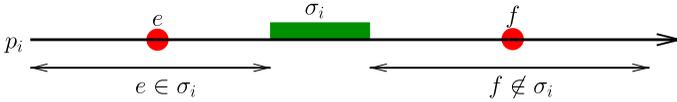
## 6.4 Global States Including Process States and Channel States

### 6.4.1 Global State Including Channel States

In some cases, we are not interested in global states consisting of only one local state per process, but in global states consisting both of process local states and channel states.

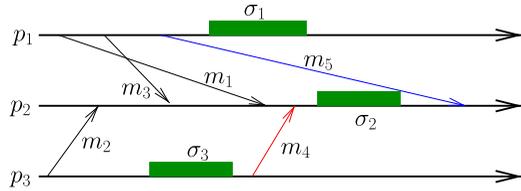
To that end we consider that processes are connected by unidirectional channels. This is without loss of generality as a bidirectional channel connecting  $p_i$  and  $p_j$  can be realized with two unidirectional channels, one from  $p_i$  to  $p_j$  and one from  $p_j$  to  $p_i$ . The state of the channel from  $p_i$  to  $p_j$  consists in the messages that have been sent by  $p_i$ , and have not yet been received by  $p_j$ . A global state is consequently made up of two parts:

- a vector  $\Sigma$  with a local state per process, plus
- a set  $\Sigma C$  whose each element represents the state of a given channel.



**Fig. 6.10** Illustrating the notations “ $e \in \sigma_i$ ” and “ $f \notin \sigma_i$ ”

**Fig. 6.11** In-transit and orphan messages



### 6.4.2 Consistent Global State Including Channel States

**Notation** Let  $\sigma_i$  be a local state of a process  $p_i$ , and  $e$  and  $f$  be two events produced by  $p_i$ . The notation  $e \in \sigma_i$  means that  $p_i$  has issued  $e$  before attaining the local state  $\sigma_i$ , while  $f \notin \sigma_i$  means that  $p_i$  has issued  $f$  after its local state  $\sigma_i$ . We then say “ $e$  belongs to the past of  $\sigma_i$ ”, and “ $f$  belongs to the future of  $\sigma_i$ ”. These notations are illustrated in Fig. 6.10.

**In-transit Messages and Orphan Messages** The notions of in-transit and orphan messages are with respect to an ordered pair of local states. Let  $m$  be a message sent by  $p_i$  to  $p_j$ , and  $\sigma_i$  and  $\sigma_j$  two local states of  $p_i$  and  $p_j$ , respectively. Let us recall that  $s(m)$  and  $r(m)$  are the send event (by  $p_i$ ) and the reception event (by  $p_j$ ) associated with  $m$ , respectively.

- A message  $m$  is *in-transit* with respect to the ordered pair of local states  $\langle \sigma_i, \sigma_j \rangle$  if

$$s(m) \in \sigma_i \wedge r(m) \notin \sigma_j.$$

- A message  $m$  is *orphan* with respect to the ordered pair of local states  $\langle \sigma_i, \sigma_j \rangle$  if

$$s(m) \notin \sigma_i \wedge r(m) \in \sigma_j.$$

These definitions are illustrated in Fig. 6.11, where there are three processes, and two channels, one from  $p_1$  to  $p_2$ , and one from  $p_3$  to  $p_2$ . As, on the one hand,  $s(m_1) \in \sigma_1$  and  $s(m_3) \in \sigma_1$ , and, on the other hand,  $r(m_1) \in \sigma_2$  and  $r(m_3) \in \sigma_2$ , both  $m_1$  and  $m_3$  are “in the past” of the directed pair  $\langle \sigma_1, \sigma_2 \rangle$ . Differently, as  $s(m_5) \in \sigma_1$  and  $r(m_5) \notin \sigma_2$ , the message  $m_5$  is in-transit with respect to the ordered pair  $\langle \sigma_1, \sigma_2 \rangle$ .

Similarly, the message  $m_2$  belongs to the “past” of the pair of local states  $\langle \sigma_3, \sigma_2 \rangle$ . Differently, with respect to this ordered pair, the message  $m_4$  has been received by  $p_2$  ( $r(m_4) \in \sigma_2$ ), but has not been sent by  $p_3$  ( $s(m_4) \notin \sigma_3$ ); hence it is an orphan message.

Considering Fig. 6.11, let us observe that message  $m_5$ , which is in-transit with respect to pair of local states  $\langle \sigma_1, \sigma_2 \rangle$ , does not make this directed pair inconsistent

(it does not create a causal path invalidating  $\sigma_1 \parallel \sigma_2$ ). The case of an orphan message is different. As we can see, the message  $m_4$  creates the dependence  $\sigma_3 \xrightarrow{\sigma} \sigma_2$ , and, consequently, we do not have  $\sigma_3 \parallel \sigma_2$ . Hence, orphan messages prevent local states from belonging to the same consistent global state.

**Consistent Global State** Let us define the state of a FIFO channel as a sequence of messages, and the state of a non-FIFO channel as a set of messages. The state of a channel from  $p_i$  to  $p_j$  with respect to a directed pair  $\langle \sigma_i, \sigma_j \rangle$  is denoted  $c\_state(i, j)$ . As already indicated, it is the sequence (or the set) of messages sent by  $p_i$  to  $p_j$  whose send events are in the past of  $\sigma_i$ , while their receive events are not in the past of  $\sigma_j$  (they are in its “future”).

Let  $C = \{(i, j) \mid \exists \text{ a directed channel from } p_i \text{ to } p_j\}$ . A global state  $(\Sigma, C\Sigma)$ , where  $\Sigma = [\sigma_1, \dots, \sigma_n]$  and  $C\Sigma = \{c\_state(i, j)\}_{(i,j) \in C}$ , is consistent if, for any message  $m$ , we have (where  $\oplus$  stands for exclusive or)

- C1:  $(s(m) \in \sigma_i) \Rightarrow (r(m) \in \sigma_j \oplus m \in c\_state(i, j))$ , and
- C2:  $(s(m) \notin \sigma_i) \Rightarrow (r(m) \notin \sigma_j \wedge m \notin c\_state(i, j))$ .

This definition states that, to be consistent, a global state  $(\Sigma, C\Sigma)$  has to be such that, with respect to the process local states defined by  $\Sigma$ , (C1) each in-transit message belongs to the state of the corresponding channel, and (C2) there is no message received and not sent (i.e., no orphan message).

**$\Sigma$  Versus  $(\Sigma, C\Sigma)$**  Let us observe that the knowledge of  $\Sigma$  contains implicitly the knowledge of  $C\Sigma$ . This is because, for any channel, the past of the local state  $\sigma_i$  contains implicitly the messages sent by  $p_i$  to  $p_j$  up to  $\sigma_i$ , while the past of the local state  $\sigma_j$  contains implicitly the messages sent by  $p_i$  and received by  $p_j$  up to  $\sigma_j$ . Hence, in the following we use without distinction  $\Sigma$  or  $(\Sigma, C\Sigma)$ .

### 6.4.3 Consistent Global State Versus Consistent Cut

The notion of a cut was introduced in Sect. 6.1.3. A cut  $C$  is a set of events defined from prefixes of process histories. Let  $\sigma_i$  be the local state of  $p_i$  obtained after executing the events in its prefix history  $\text{prefix}(\widehat{h}_i)$  as defined by the cut, i.e.,  $\sigma_i = \delta(s_i^0, \text{prefix}(\widehat{h}_i))$  using the transition function-based notation ( $\sigma_i^0$  being the initial state of  $p_i$ ). It follows that  $[\sigma_1, \dots, \sigma_i, \dots, \sigma_n]$  is a global state  $\Sigma$ . Moreover, the cut  $C$  is consistent if and only if  $\Sigma$  is consistent.

When considering  $(\Sigma, C\Sigma)$ , we have the following. If the cut giving rise to  $\Sigma$  is consistent, the state of the channels in  $C\Sigma$  correspond to the messages that cross the cut line (their send events belong to the cut, while their receive events do not). If the cut is not consistent, there is at least one message that crosses the cut line in the “bad direction” (its send event does not belong to the cut, while its receive event does, this message is an orphan message).

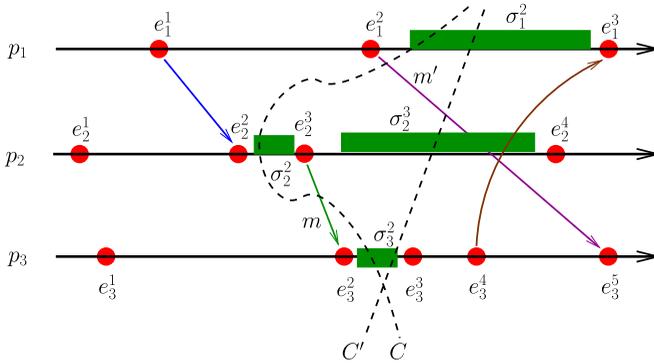


Fig. 6.12 Cut versus global state

Examples of a cut  $C'$ , which is consistent, and a cut  $C$ , which is not consistent, were depicted in Fig. 6.3. This execution is reproduced in Fig. 6.12, which is enriched with local states. When considering the consistent cut  $C'$ , we obtain the consistent global state  $\Sigma' = [\sigma_1^2, \sigma_2^3, \sigma_3^2]$ . Moreover,  $C\Sigma'$  is such that the state of each channel is empty, except for the channel from  $p_1$  to  $p_3$  for which we have  $c\_state(1, 3) = \{m'\}$ . When considering the inconsistent cut  $C$ , we obtain the (inconsistent) global state  $\Sigma = [\sigma_1^2, \sigma_2^2, \sigma_3^2]$ , whose inconsistency is due to the orphan message  $m$ .

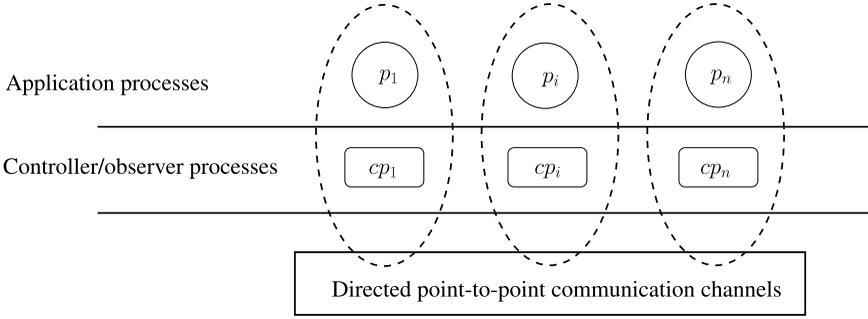
## 6.5 On-the-Fly Computation of Global States

### 6.5.1 Global State Computation Is an Observation Problem

The aim is to design algorithms that compute a consistent global state of a distributed application made up of  $n$  processes  $p_1, \dots, p_n$ . To that end, a controller (or observer) process, denoted  $cp_i$ , is associated with each application process  $p_i$ . The role of a controller is to observe the process it is associated with, in such a way that the set of controllers computes a consistent global state of the application.

Hence, the computation of a consistent global state of a distributed computation is an *observation* problem: The controllers have to observe the application processes without modifying their behavior. This is different from problems such as the navigation of a mobile object (addressed in the previous chapter), where the aim was to provide application processes with appropriate operations (such as `acquire()` and `release()`) that they can invoke.

The corresponding structural view is described in Fig. 6.13. Ideally, the addition/suppression of controllers/observers must not modify the execution of the distributed application.



**Fig. 6.13** Global state computation: structural view

### 6.5.2 Problem Definition

The computation of a global state is launched independently by any number  $x$  of controllers, where  $1 \leq x \leq n$ . The problem is defined by the following properties.

- **Liveness.** If at least one controller launches a global state computation, a global state is computed.
- **Safety.** Let  $(\Sigma, C\Sigma)$  be the global state that is computed.
  - **Validity.** Let  $\Sigma_{start}$  be the global state of the application when the global state computation starts and  $\Sigma_{end}$  its global state when it terminates.  $\Sigma$  is such that  $\Sigma_{start} \xrightarrow{\Sigma} \Sigma$  and  $\Sigma \xrightarrow{\Sigma} \Sigma_{end}$ .
  - **Channel consistency.**  $C\Sigma$  records all the messages (and only those) that are in-transit with respect to  $\Sigma$ .

The validity property states that the global state  $\Sigma$  that is computed is consistent and up to date (always returning the initial state would not solve the problem). Moreover, when considering the lattice of global states associated with the distributed execution, the computed global state  $\Sigma$  is reachable from  $\Sigma_{start}$ , and  $\Sigma_{end}$  is reachable from it. This defines the “freshness” of  $\Sigma$ . The channel consistency states that  $C\Sigma$  (recorded channel states) is in agreement with  $\Sigma$  (recorded process local states).

### 6.5.3 On the Meaning of the Computed Global State

**On the Nondeterminism of the Result** Considering the lattice of global states described in Fig. 6.8, let us assume that the computation starts when the application is in state  $\Sigma_{start} = [0, 1]$  and terminates when the application is in state  $\Sigma_{end} = [1, 2]$ . The validity property states that the computed global state  $\Sigma$  is one of the global states  $[0, 1]$ ,  $[1, 1]$ ,  $[0, 2]$ , or  $[1, 2]$ .

The global state that is computed depends actually on the execution and the interleaving of the events generated by the application processes and by the controllers in charge of the computation of the global state. The validity property states only that  $\Sigma$  is a consistent global state that the execution *might* have passed through. Maybe the execution passed through  $\Sigma$ , maybe it did not. Actually, there is no means to know if the distributed execution passed through  $\Sigma$  or not. This is due to fact that independent events can be perceived, by distinct sequential observers, as having been executed in different order (see Fig. 6.9). Hence, the validity property characterizes the best that can be done, i.e., (a)  $\Sigma$  is consistent, and (b) it can have been passed through by the execution. Nothing stronger can be claimed.

**The Case of Stable Properties** A stable property is a property that, once true, remains true forever. “The application has terminated”, “there is a deadlock”, or “a distributed cell has become inaccessible”, are typical examples of stable properties that can be defined on the global states of a distributed application. More precisely, let  $\mathcal{P}$  be a property on the global states of a distributed execution. If  $\mathcal{P}$  is a stable property, we have

$$\mathcal{P}(\Sigma) \Rightarrow (\forall \Sigma' : \Sigma \xrightarrow{\Sigma} \Sigma' : \mathcal{P}(\Sigma')).$$

Let  $\Sigma$  be a consistent global state that has been computed, and  $\Sigma_{start}$  and  $\Sigma_{end}$  as defined previously. We have the following:

- As  $\mathcal{P}$  is a stable property and  $\Sigma \xrightarrow{\Sigma} \Sigma_{end}$ , we have  $\mathcal{P}(\Sigma) \Rightarrow \mathcal{P}(\Sigma_{end})$ . Hence, if  $\mathcal{P}(\Sigma)$  is true, whether the distributed execution passed through  $\Sigma$  or not, we conclude that  $\mathcal{P}(\Sigma_{end})$  is satisfied, i.e., the property is satisfied in the global state  $\Sigma_{end}$ , which is attained by the distributed execution (and can never be explicitly known). Moreover, from then on, due to its stability, we know that  $\mathcal{P}$  is satisfied on all future global states of the distributed computation.
- If  $\neg\mathcal{P}(\Sigma)$ , taking the contrapositive of the definition of a stable property, we can conclude  $\neg\mathcal{P}(\Sigma_{start})$ , but nothing can be concluded on  $\mathcal{P}(\Sigma_{end})$ .

In this case, a new global state can be computed, until either a computed global state satisfies  $\mathcal{P}$ , or the computation has terminated. (Let us observe that, if  $\mathcal{P}$  is “the computation has terminated”, it is eventually satisfied. Special chapters of the book are devoted to the detection of stable properties such as distributed termination and deadlock detection.)

#### 6.5.4 Principles of Algorithms Computing a Global State

An algorithm that computes a global state of a distributed execution has to ensure that each controller  $cp_i$  records a local state  $\sigma_i$  of the process  $p_i$  it is associated with, and each pair of controllers  $(cp_j, cp_i)$  has to compute the value  $c\_state(j, i)$  (i.e., the state of the directed channel from  $p_j$  to  $p_i$ ).

In order that  $(\Sigma, C\Sigma)$  be consistent, the controllers have to cooperate to ensure the conditions C1 and C2 stated in Sect. 6.4.2. This cooperation involves both synchronization and message recording.

- **Synchronization.** In order that there is no orphan message with respect to a pair  $\langle \sigma_j, \sigma_i \rangle$ , when a message is received from  $p_j$ , the controller  $cp_i$  might be forced to record the local state of  $p_i$  before giving it the message.
- **Message recording.** In order that the in-transit messages appear in the state of the corresponding channels, controllers have to record them in one way or another.

While nearly all global state computation algorithms use the same synchronization technique to ensure  $\Sigma$  is consistent, they differ in (a) the technique they use to record in-transit messages, and (b) the FIFO or non-FIFO assumption they consider for the directed communication channels.

## 6.6 A Global State Algorithm Suited to FIFO Channels

The algorithm presented in this section is due to K.M. Chandy and L. Lamport (1985). It was the first global state computation algorithm proposed. It assumes that (a) the channels are FIFO, and (b) the communication graph is strongly connected (there is a directed communication path from any process to any other process).

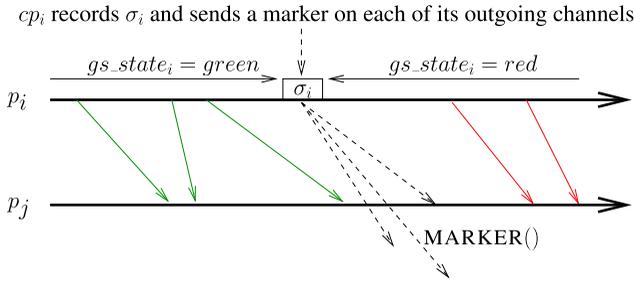
### 6.6.1 Principle of the Algorithm

**A Local Variable per Process Plus a Control Message** The local variable  $gs\_state_i$  contains the state of  $p_i$  with respect to the global state computation. Its value is *red* (its local state has been recorded) or *green* (its local state has not yet been recorded). Initially,  $gs\_state_i = green$ .

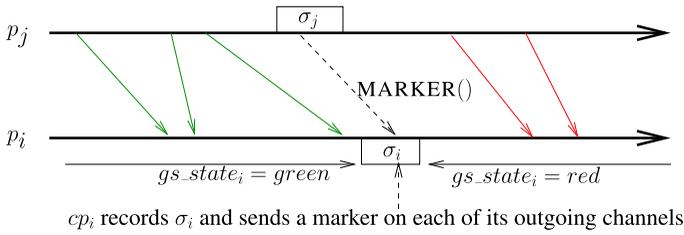
A controller  $cp_i$ , which has not yet recorded the local state of its associated process  $p_i$ , can do it at any time. When  $cp_i$  records the local state of  $p_i$ , it atomically (a) updates  $gs\_state_i$  to the value *red*, and (b) sends a control message (denoted `MARKER()`, and depicted with dashed arrows) on all its outgoing channels. As channels are FIFO, a message `MARKER()` is a synchronization message separating the application messages sent by  $p_i$  before it from the application messages sent after it. This is depicted in Fig. 6.14.

When a message is received by the pair  $(p_i, cp_i)$ , the behavior of  $cp_i$  depends on the value of  $gs\_state_i$ . There are two cases.

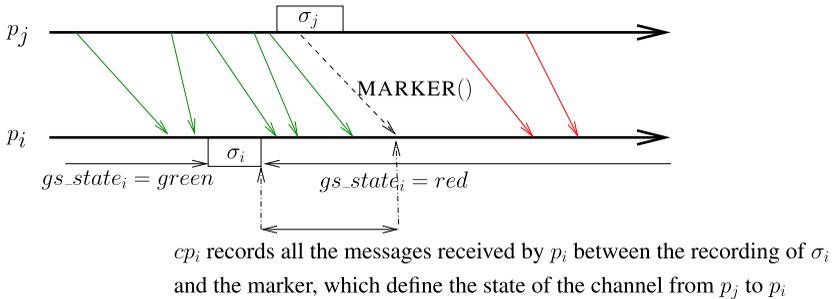
- $gs\_state_i = green$ . This case is depicted in Fig. 6.15. The controller  $cp_i$  discovers that a global state computation has been launched. It consequently participates in it by atomically recording  $\sigma_i$ , and sending `MARKER()` messages on its outgoing channels to inform their destination processes that a global state computation has been started.



**Fig. 6.14** Recording of a local state



**Fig. 6.15** Reception of a MARKER() message: case 1



**Fig. 6.16** Reception of a MARKER() message: case 2

Moreover, for the ordered pair of local states  $\langle \sigma_j, \sigma_i \rangle$  to be consistent,  $cp_i$  defines the state  $c\_state(j, i)$  of the incoming channel from  $p_j$  as being empty.

- $gs\_state_i = red$ . In this case, depicted in Fig. 6.16,  $cp_i$  has already recorded  $\sigma_i$ . Hence, it has only to ensure that the recorded channel state  $c\_state(j, i)$  is consistent with respect to the ordered pair  $\langle \sigma_j, \sigma_j \rangle$ . To that end,  $cp_i$  records the sequence of messages that are received on this channel between the recording of  $\sigma_i$  and the reception of the marker sent by  $cp_j$ .

**Properties** Assuming at least one controller process  $cp_i$  launches a global state computation (Fig. 6.14), it follows from the strong connectivity of the communi-

```

internal operation record_ls() is
(1)  $\sigma_i \leftarrow$  current local state of  $p_i$ ;
(2)  $gs\_state_i \leftarrow red$ ;
(3) for each  $k \in c\_in_i$  do  $c\_state(k, i) \leftarrow \emptyset$  end for;
(4) for each  $j \in c\_out_i$  do send MARKER() on  $out\_channel_i[j]$  end for.

when START() is received do
(5) if ( $gs\_state_i = green$ ) then record_ls() end if.

when MSG( $v$ ) is received on  $in\_channel_i[j]$  do
(6) case MSG = MARKER then if ( $gs\_state_i = green$ ) then record_ls() end if;
(7)  $closed_i[j] \leftarrow true$ 
(8) MSG = APPL then if ( $gs\_state_i = red$ )  $\wedge$  ( $\neg closed_i[j]$ )
(9) then add the message APPL() at the tail of  $c\_state(j, i)$ 
(10) end if;
(11) pass the message APPL( $v$ ) to  $p_i$ 
(12) end case.

```

**Fig. 6.17** Global state computation (FIFO channels, code for  $cp_i$ )

cation graph and the rules associated with the control messages MARKER() that exactly one local state per process is recorded, and exactly one marker is sent on each directed channel. It follows that a global state is eventually computed.

Let us color a process with the color currently in  $cs\_state_i$ . Similarly, let the color of an application message be the color of its sender when the message is sent. According to these colorings, an orphan message is a red message received by a green process. But, as the channel from  $p_j$  to  $p_i$  is FIFO, no red message can arrive before a marker, from which it follows that there is no orphan message. The fact that all in-transit messages are recorded follow the recording rules expressed in Figs. 6.15 and 6.16. The recorded global state is consequently consistent.

## 6.6.2 The Algorithm

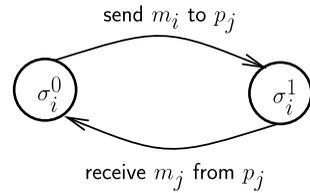
**Local Variables** Let  $c\_in_i$  denote the sets of process identities  $j$  such that there is a channel from  $p_j$  to  $p_i$ ; this channel is denoted  $in\_channel_i[j]$ . Similarly, let  $c\_out_i$  denote the sets of process identities  $j$  such that there is a channel from  $p_i$  to  $p_j$ ; this channel is denoted  $out\_channel_i[j]$ . These sets are known both by  $p_i$  and its controller  $cp_i$ . (As the network is strongly connected, no set  $c\_in_i$  or  $c\_out_i$  is empty.)

The local array  $closed_i[c\_in_i]$  is a Boolean array. Each local variable  $closed_i[j]$  is initialized to *false*, and is set to value *true* when  $cp_i$  receives a marker from  $cp_j$ .

**Algorithm Executed by  $cp_i$**  The algorithm executed by a controller process  $cp_i$  is described in Fig. 6.17. The execution of the internal operation record\_ls(), and the processing associated with the reception of a message START() or MSG() are

**Fig. 6.18**

A simple automaton  
for process  $p_i$  ( $i = 1, 2$ )



atomic. This means they exclude each other, and exclude also concurrent execution of the process  $p_i$ . (Among other issues, the current state of  $p_i$  has not to be modified when  $cp_i$  records  $\sigma_i$ ).

One or several controller processes launch the algorithm when they receive an external message `START()`, while their local variable  $gs\_state_i = green$  (line 4). Such a process then invokes the internal operation `record_ls()`. This operation records the current local state of  $p_i$  (line 1), switches  $gs\_state_i$  to *red* (line 2), initializes the channel states  $c\_state(j, i)$  to the empty sequence, denoted  $\emptyset$  (line 3), and sends markers on  $p_i$ 's outgoing channels (line 4).

The behavior of a process  $cp_i$  that receives a message depends on the message. If it is marker (control message),  $cp_i$  learns that the sender  $cp_j$  has recorded  $\sigma_j$ . Hence, if not yet done,  $cp_i$  executes the internal operation `record_ls()` (line 6). Moreover, in all cases, it sets  $closed_i[j]$  to *true* (line 7), to indicate that the state of the input channel from  $p_j$  has been computed, this channel state being consistent with respect to the pair of local states  $(\sigma_j, \sigma_i)$  (see Figs. 6.15 and 6.16).

If the message is an application message, and the computation of  $c\_state(j, i)$  has started and is not yet finished (predicate of line 8),  $cp_i$  adds it at the end of  $c\_state(j, i)$ . In all cases, the message is passed to the application process  $p_i$ .

### 6.6.3 Example of an Execution

This section illustrates the previous global state computation algorithm with a simple example. Its aim is to give the reader a deeper insight on the subtleties of global state computations.

**Application Program and a Simple Execution** The application is made up of two processes  $p_1$  and  $p_2$  and two channels, one from  $p_1$  to  $p_2$ , and one from  $p_2$  to  $p_1$ . Moreover, both processes have the same behavior, described by the automaton described in Fig. 6.18. Process  $p_i$  is in state  $\sigma_i^0$  and moves to state  $\sigma_i^1$  by sending the message  $m_i$  to the other process  $p_j$ . It then returns to its initial state when it receives the message  $m_j$  from process  $p_j$ , and this is repeated forever.

A prefix of an execution of  $p_1$  and  $p_2$  is described in Fig. 6.19.

**Superimposing a Global State Computation** Figure 6.20 describes a global state computation. This global state computation is superimposed on the distributed execution described in Fig. 6.19. The control processes  $cp_1$  and  $cp_2$  access the local

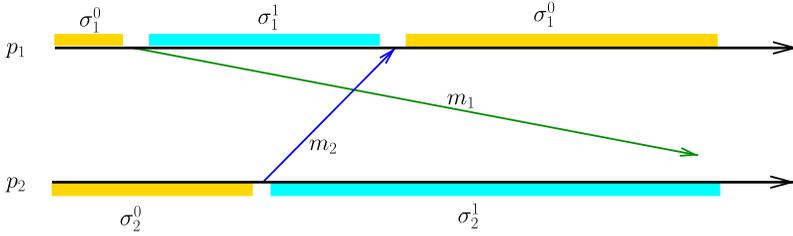


Fig. 6.19 Prefix of a simple execution

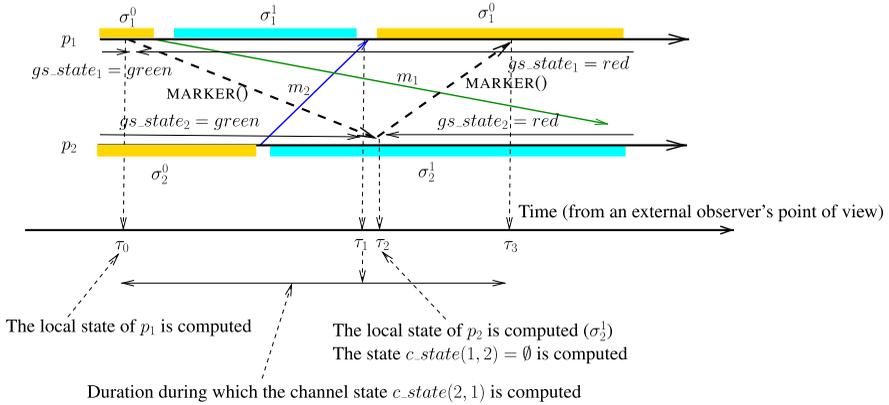


Fig. 6.20 Superimposing a global state computation on a distributed execution

control variables  $gs\_state_1$  and  $gs\_state_2$ , respectively, and send  $MARKER()$  messages. This global state computation involves four time instants (defined from an external omniscient observer's point of view).

- At time  $\tau_0$ , the controller  $cp_1$  receives a message  $START()$  (line 5). Hence, when the global state computation starts, the distributed execution is in the global state  $\Sigma_{start} = [\sigma_1^0, \sigma_2^0]$ . The process  $cp_1$  consequently invokes  $record\_ls()$ . It records the current state of  $p_1$ , which is then  $\sigma_1^0$ , and sends a marker on its only outgoing channel (lines 1–5).
- Then, at time  $\tau_1$ , the application message  $m_2$  arrives at  $p_1$  (lines 8–11). This message is first handled by  $cp_1$ , which adds a copy of it to the channel state  $c\_state(2, 1)$ . It is then received by  $p_1$  which moves from  $\sigma_1^1$  to  $\sigma_1^0$ .
- Then, at time  $\tau_2$ ,  $cp_2$  receives the marker sent by  $cp_1$  (lines 6–7). As  $gs\_state_2 = green$ ,  $cp_2$  invokes  $record\_ls()$ : It records the current state of  $p_2$ , which is  $\sigma_2^1$ , and sends a marker to  $cp_1$ .
- Finally, at time  $\tau_3$ , the marker sent by  $cp_2$  arrives at  $cp_1$ . As  $gs\_state_1 = red$ ,  $cp_1$  has only to stop recording the messages sent by  $p_2$  to  $p_1$ . Hence, when the global state computation terminates, the distributed execution is in the global state  $\Sigma_{end} = [\sigma_1^0, \sigma_2^1]$ .

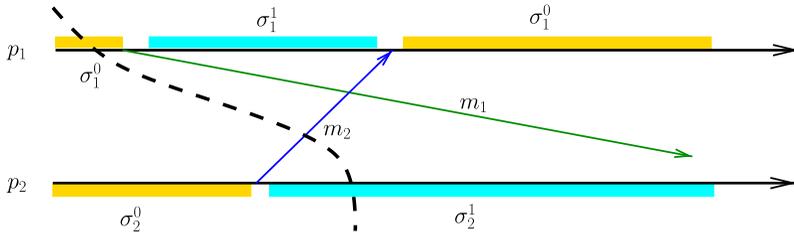


Fig. 6.21 Consistent cut associated with the computed global state

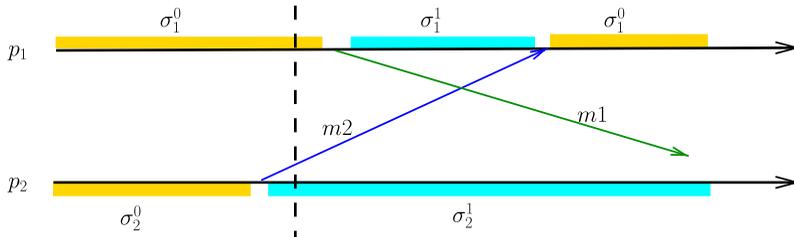


Fig. 6.22 A rubber band transformation

It follows that the global state that been cooperatively computed by  $cp_1$  and  $cp_2$  is the pair  $(\Sigma, C\Sigma)$ , where  $\Sigma = [\sigma_1^0, \sigma_2^1]$ , and  $C\Sigma = \{c\_state(1, 2), c\_state(2, 1)\}$  with  $c\_state(1, 2) = \emptyset$  and  $c\_state(2, 1) = \langle m_2 \rangle$ . It is important to see that this computation is not at all atomic: as illustrated in the space-time diagram, it is distributed both with respect to space (processes) and time.

**Consistent Cut and Rubber Band Transformation** Figure 6.21 is a copy of Fig. 6.19, which explicitly represent the consistent cut associated with the global state that has been computed. This figure shows that the distributed execution did not pass through the computed global state.

Actually, as suggested in Sect. 6.1.4, we can play with asynchrony and physical time to exhibit a distributed execution which has passed through the computed global state. This distributed is obtained by the “rubber band transformation”: in the space-time diagram, we consider process axes are rubber bands that can be stretched or shrunk as long as no message is going backwards. Such a rubber band transformation is described in Fig. 6.22. As the execution described in Fig. 6.21 and the execution described in Fig. 6.22 define the same partial order, they are actually the *same* execution.

## 6.7 A Global State Algorithm Suited to Non-FIFO Channels

This section presents a global state computation algorithm suited to systems with directed non-FIFO channels. This algorithm, which is due to T.H. Lai and T.H.

Yang (1987), differs from the previous one mainly in the way it records the state of the channels.

### 6.7.1 The Algorithm and Its Principles

The local variables  $c\_in_i$ ,  $c\_out_i$ ,  $in\_channel_i[k]$ , and  $out\_channel_i[k]$  have the same meaning as before. In addition to  $gs\_state_i$ , which is initialized to *green* and is eventually set to *red*, each controller  $cp_i$  manages two arrays of sets defined as follows:

- $rec\_msg_i[c\_in_i]$  is such that, for each  $k \in c\_in_i$ , we have

$$rec\_msg_i[k] = \{\text{all the messages received on } in\_channel_i[k] \text{ since the beginning}\}.$$

- $sent\_msg_i[c\_out_i]$  is such that, for each  $k \in c\_out_i$ , we have

$$sent\_msg_i[k] = \{\text{all the messages sent on } out\_channel_i[k] \text{ since the beginning}\}.$$

Each array is a log in which  $cp_i$  records all the messages it receives on each input channel, and all the messages it sends on each output channel. This means that (differently from the previous algorithm),  $cp_i$  has to continuously observe  $p_i$ . But now there is no control message: All the messages are application messages. Moreover, each message inherits the current color of its sender (hence, each message carries one control bit).

The basic rule is that, while a green message can always be consumed, a red message can be consumed only when its receiver is red. It follows that, when a green process  $p_i$  receives a red message,  $cp_i$  has first to record  $p_i$ 's local state so that  $p_i$  becomes red before being allowed to receive and consume this message.

The algorithm is described in Fig. 6.23. The text is self-explanatory.  $m.color$  denotes the color of the application message  $m$ . Line 7 ensures that a red message cannot be received by a green process, which means that there is no orphan message with respect to an ordered pair of process local states that have been recorded.

### 6.7.2 How to Compute the State of the Channels

Let  $cp$  be any controller process. After it has recorded  $\sigma_i$ ,  $rec\_msg_i[c\_in_i]$ , and  $sent\_msg_i[c\_out_i]$  (line 3), a controller  $cp_i$  sends this triple to  $cp$  (line 4).

When it has received such a triple from each controller  $cp_i$ , the controller  $cp$  pieces together all the local states to obtain  $\Sigma = [\sigma_1, \dots, \sigma_n]$ . As far as  $C\Sigma$  is concerned, it defines the state of the channel from  $p_j$  to  $p_i$  as follows:

$$c\_state(j, i) = sent\_msg_j[i] \setminus rec\_msg_i[j].$$

```

internal operation record_ls() is
(1)  $\sigma_i \leftarrow$  current local state of  $p_i$ ;
(2)  $gs\_state_i \leftarrow red$ ;
(3) record  $\sigma_i$ ,  $rec\_msg_i[c\_in_i]$ , and  $sent\_msg_i[c\_out_i]$ ;
(4) send the previous triple to  $cp$ .

when START() is received do
(5) if ( $gs\_state_i = green$ ) then record_ls() end if.

when MSG( $m$ ) is received on  $in\_channel_i[j]$  do
(6)  $rec\_msg_i[j] \leftarrow rec\_msg_i[j] \cup \{m\}$ ;
(7) if ( $m.color = red$ )  $\wedge$  ( $gs\_state_i = green$ ) then record_ls() end if;
(8) pass MSG( $m$ ) to  $p_i$ .

when MSG( $m$ ) is sent on  $out\_channel_i[j]$  do
(9)  $m.color \leftarrow gs\_state_i$ ;
(10)  $sent\_msg_i[k] \leftarrow sent\_msg_i[j] \cup \{m\}$ .
    
```

Fig. 6.23 Global state computation (non-FIFO channels, code for  $cp_i$ )

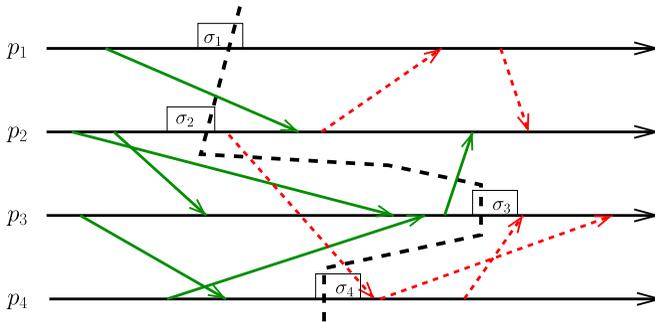


Fig. 6.24 Example of a global state computation (non-FIFO channels)

It follows from (a) this definition, (b) the fact that  $sent\_msg_j[i]$  is the set of messages sent by  $p_j$  to  $p_i$  before  $\sigma_j$ , and (c) the fact that  $rec\_msg_i[j]$  is the set of messages received by  $p_i$  from  $p_j$  before  $\sigma_i$ , that  $c\_state(j, i)$  records the messages that are in-transit with respect to the ordered pair  $\langle \sigma_j, \sigma_i \rangle$ . As there is no orphan message with respect to the recorded local states (see above), the computed global state is consistent.

**An Example** An example of execution of the previous algorithm is described in Fig. 6.24. Green messages are depicted with plain arrows, while red messages are depicted with dashed arrows.

In this example, independently from the other controller processes, each of  $cp_1$  and  $cp_2$  receives an external START() message, and consequently records the local states of  $p_1$  and  $p_2$ , respectively. After  $\sigma_2$  has been recorded,  $p_2$  sends a (red) message to  $p_4$ . When this message is received,  $\sigma_4$  is recorded by  $cp_4$  before being

passed to  $p_4$ . Then,  $p_4$  sends a (red) message to  $p_3$ . As this message is red, its reception entails the recording of  $\sigma_3$  by  $cp_3$ , after which the message is received and processed by  $p_3$ .

The cut corresponding to this global state is indicated by a bold dashed line. It is easy to see that it is consistent. The in-transit messages are the two green messages (plain arrows) that cross the cut line.

**Remark** The logs  $sent\_msg_i[c\_out_i]$  and  $rec\_msg_i[c\_in_i]$  may require a large memory, which can be implemented with a local secondary storage. In some applications, based on the computation of global states, what is relevant is not the exact value of these sets, but their cardinality. In this case, each set  $sent\_msg_i[k]$ , and each set  $rec\_msg_i[j]$ , can be replaced by a simple counter. As a simple example, this is sufficient when one wants to know which channels are empty in a computed global state.

## 6.8 Summary

This chapter was on the nature of a distributed execution and the associated notion of a global state (snapshot). It has defined basic notions related to the execution of a distributed program (event, process history, process local state, event concurrency/independence, cut, and global state). It has also introduced three approaches to model a distributed execution, namely, a distributed execution can be represented as a partial order on events, a partial order on process local states, or a lattice of global states.

The chapter then presented algorithms that compute on the fly consistent global states of a distributed execution. It has shown that the best that can be done is the computation of a global state that might have been passed through by the distributed computation. The global state that has been computed is consistent, but no process can know if it really occurred during the execution. This nondeterminism is inherent to the nature of asynchronous distributed computing. An aim of this chapter was to give the reader an intuition of this relativistic dimension, when the processes have to observe on the fly the distributed execution they generate.

## 6.9 Bibliographic Notes

- The capture of a distributed execution as a partial order on the events produced by processes is due to L. Lamport [226]. This is the first paper that formalized the notion of a distributed execution.
- The first paper that presented a precise definition of a global state of a distributed computation and an algorithm to compute a consistent global state is due to K.M. Chandy and L. Lamport [75]. Their algorithm is the one presented in Sect. 6.6. A formal proof can be found in [75].

- The algorithm by Chandy and Lamport computes a single global state. It has been extended in [57, 357] to allow for repeated global state computations.
- The algorithm for non-FIFO channels presented in Sect. 6.7 is due to Y.T. Lai and Y.T. Yang [222].
- Numerous algorithms that compute consistent global states in particular contexts have been proposed. As an example, algorithms suited to causally ordered communication are presented in [1, 14]. Other algorithms that compute (on the fly) consistent global states are presented in [169, 235, 253, 377]. A reasoned construction of an algorithm computing a consistent global state is presented in [80]. A global state computation algorithm suited to large-scale distributed systems is presented in [213]. An introductory survey on global state algorithms can be found in [214].
- The algorithms that have been presented are non-inhibitory, in the sense that they never freeze the distributed execution they observe. The role of inhibition for global state computation is investigated in [167, 363].
- The view of a distributed computation as a lattice of consistent global states is presented and investigated in [29, 98, 250]. Algorithms which determine which global states of a distributed execution satisfy some predefined properties are presented in [28, 98]. Those algorithms are on-the-fly algorithms.
- A global state  $\Sigma$  of a distributed execution is *inevitable* if, when considering the lattice associated with this execution, it belongs to all the sequential observations defined by this lattice. An algorithm that computes on the fly all the inevitable global states of a distributed execution is described in [138].
- The causal precedence (happened before) relation has been generalized in several papers (e.g., [174, 175, 297]).
- Snapshot computation in anonymous distributed systems is addressed in [220].

## 6.10 Exercises and Problems

1. Adapt the algorithm described in Sect. 6.6 so that the controller processes are able to compute several consistent global states, one after the other.  
Answer in [357].
2. Use the rubber band transformation to give a simple characterization of an orphan message.
3. Let us consider the algorithm described in Fig. 6.25 in which each controller process  $cp_i$  manages the local variable  $gs\_state_i$  (as in the algorithms described in this chapter), plus an integer  $count_i$ . Moreover, each message  $m$  carries the color of its sender (in its field  $m.color$ ). The local control variable  $count_i$  counts the number of green messages sent by  $p_i$  minus the number of green messages received by  $p_i$ . The controller  $cp$  is one of the controllers, known by all, that is in charge of the construction of the computed global state.

When  $cp$  has received a pair  $(\sigma_i, count_i)$  from each controller  $cp_i$ , it computes  $ct = \sum_{1 \leq i \leq n} count_i$ .

```

internal operation record_ls() is
(1)  $\sigma_i \leftarrow$  current local state of  $p_i$ ;
(2)  $gs\_state_i \leftarrow red$ ;
(3) record  $\sigma_i, count_i$ ; send the pair  $(\sigma_i, count_i)$  to  $cp$ .

when START() is received do
(4) if  $(gs\_state_i = green)$  then record_ls() end if.

when MSG( $m$ ) is received on  $in\_channel_i[j]$  do
(5) if  $(m.color = red) \wedge (gs\_state_i = green)$  then record_ls() end if;
(6) if  $(m.color = green)$  then
(7)   if  $(gs\_state_i = green)$  then  $count_i \leftarrow count_i - 1$  else send a copy of  $m$  to  $cp$  end if
(8) end if;
(9) pass MSG( $m$ ) to  $p_i$ .

when MSG( $m$ ) is sent on  $out\_channel_i[j]$  do
(10)  $m.color \leftarrow gs\_state_i$ ;
(11) if  $(m.color = green)$  then  $count_i \leftarrow count_i + 1$  end if.

```

**Fig. 6.25** Another global state computation (non-FIFO channels, code for  $cp_i$ )

- What does the counter  $ct$  represent?
- What can be concluded on a message  $m$  sent to  $cp$  by a controller  $cp_i$  at line 7?
- Show that the global state  $\Sigma = [\sigma_1, \dots, \sigma_n]$  obtained by  $cp$  is consistent.
- How can  $cp$  compute the state of the channels?
- Does this algorithm compute a consistent pair  $(\Sigma, C\Sigma)$ ? To answer “yes”, you need to show that each channel state  $c\_state(j, i)$  contains all the green messages sent by  $p_j$  to  $p_i$  that are received by  $p_i$  while it is red. To answer “no”, you need to show that there is no means for  $cp$  to always compute correct values for all channel states.

Answer in [250].

# Chapter 7

## Logical Time in Asynchronous Distributed Systems

This chapter is on the association of consistent dates with events, local states, or global states of a distributed computation. Consistency means that the dates generated by a dating system have to be in agreement with the “causality” generated by the considered distributed execution. According to the view of a distributed execution we are interested in, this causality is the causal precedence order on events (relation  $\xrightarrow{ev}$ ), the causal precedence order on local states (relation  $\xrightarrow{\sigma}$ ), or the reachability relation in the lattice of global states (relation  $\xrightarrow{\Sigma}$ ), all introduced in the previous chapter. In all cases, this means that the date of a “cause” has to be earlier than the date of any of its “effects”. As we consider time-free asynchronous distributed systems, these dates cannot be physical dates. (Moreover, even if processes were given access to a global physical clock, the clock granularity should be small enough to always allow for a consistent dating.)

Three types of logical time are presented, namely, scalar (or linear) time, vector time, and matrix time. Each type of time is defined, its properties are stated, and illustrations showing how to use it are presented.

**Keywords** Adaptive communication layer · Approximate causality relation · Causal precedence · Causality tracking · Conjunction of stable local predicates · Detection of a global state property · Discarding old data · Hasse diagram · Immediate predecessor · Linear (scalar) time (clock) · Logical time · Matrix time (clock) · Message stability · Partial (total) order · Relevant event ·  $k$ -Restricted vector clock · Sequential observation · Size of a vector clock · Timestamp · Time propagation · Total order broadcast · Vector time (clock)

### 7.1 Linear Time

Linear time considers events and the associated partial order relation  $\xrightarrow{ev}$  produced by a distributed execution. (The same could be done by considering local states, and the associated partial order relation  $\xrightarrow{\sigma}$ .) This notion of logical time is due to L. Lamport (1978), who introduced it together with the relation  $\xrightarrow{ev}$ .

```

when producing an internal event  $e$  do
(1)  $clock_i \leftarrow clock_i + 1$ . % date of the internal event
(2) Produce event  $e$ .

when sending  $MSG(m)$  to  $p_j$  do
(3)  $clock_i \leftarrow clock_i + 1$ ; % date of the send event
(4) send  $MSG(m, clock_i)$  to  $p_j$ .

when  $MSG(m, h)$  is received from  $p_j$  do
(5)  $clock_i \leftarrow \max(clock_i, h)$ ;
(6)  $clock_i \leftarrow clock_i + 1$ . % date of the receive event.

```

**Fig. 7.1** Implementation of a linear clock (code for process  $p_i$ )

### 7.1.1 Scalar (or Linear) Time

**Linear (Scalar) Clock** As just indicated, the aim is to associate a logical date with events of a distributed execution. Let  $date(e)$  be the date associated with event  $e$ . To be consistent the dating system has to be such that

$$\forall e_1, e_2 : (e_1 \xrightarrow{ev} e_2) \Rightarrow date(e_1) < date(e_2).$$

The simplest time domain which can respect event causality is the sequence of increasing integers (hence the name linear or scalar time): A date is an integer.

Hence, each process  $p_i$  manages an integer local variable  $clock_i$  (initialized to 0), which increases according to the relation  $\xrightarrow{ev}$ , as described by the local clock management algorithm of Fig. 7.1. Just before producing its next internal event, or sending a message, a process  $p_i$  increases its local clock  $clock_i$ , and this new clock value is the date of the corresponding internal or send event. Moreover, each message carries its sending date. When a process receives a message, it first updates its local clock and then increases it, so that the receive event has a date greater than both the date of the corresponding send event and the date of the last local event.

It follows trivially from these rules that the linear time increases along all causal paths, and, consequently, this linear time is consistent with the relation  $\xrightarrow{ev}$  (or  $\xrightarrow{\sigma}$ ). Any increment value  $d > 0$  could be used instead of 1. Considering  $d = 1$  allows for the smallest clock increase while keeping consistency.

**An Example** An illustration of the previous linear clock system is depicted in Fig. 7.2. The global linear clock is implemented by three integer local clocks,  $clock_1$ ,  $clock_2$ , and  $clock_3$ , and all the integers associated with events or messages represent clock values. Let us consider process  $p_2$ . Its first event is the sending of a message to  $p_3$ : Its date is 1. Its next event is an internal event, and its date is 2. Its third event is the reception of a message whose sending date is 3, hence this receive event is dated 4. Similarly, its next receive event is dated 5, etc.

This example shows that, due to resetting entailed by message receptions, a local clock  $clock_i$  may skip integer values. As an example,  $clock_1$  jumps from value 4

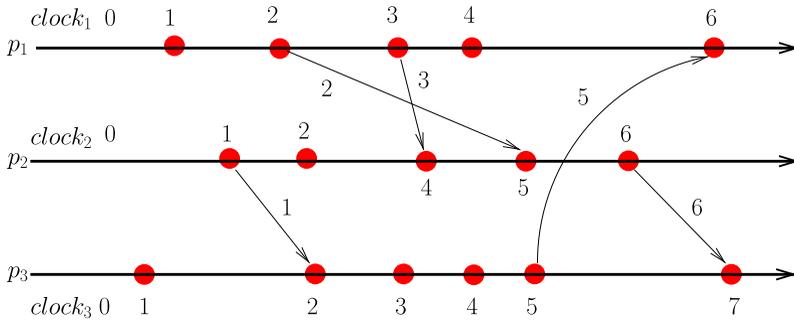


Fig. 7.2 A simple example of a linear clock system

to value 6. The fact that (a) logical time increases along causal paths, and (b) the increment value has been chosen equal to 1, provides us with the following property

$$(date(e) = x) \iff (\text{there are } x \text{ events on the longest causal path ending at } e).$$

**Properties** The following properties follow directly from clock consistency. Let  $e_1$  and  $e_2$  be two events.

- $(date(e_1) \leq date(e_2)) \Rightarrow \neg(e_2 \xrightarrow{ev} e_1).$
- $(date(e_1) = date(e_2)) \Rightarrow (e_1 || e_2).$

The first property is simply the contrapositive of the clock consistency property, while the second one is a restatement of  $(date(e_1) \leq date(e_2)) \wedge (date(e_2) \leq date(e_1)).$

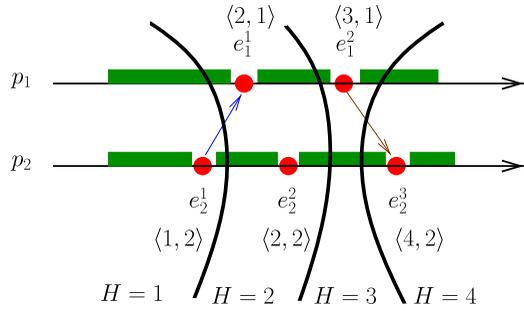
These properties are important from an operational point of view. The partial order  $\xrightarrow{ev}$  on events allows us to understand and reason on distributed executions, while the dates associated with events are operational and can consequently be used to design and write distributed algorithms. Hence, the aim is to extract information on events from their dates.

Unfortunately, it is possible to have  $(date(e_1) < date(e_2)) \wedge \neg(e_1 \xrightarrow{ev} e_2).$  With linear time, it is not because the date of an event  $e_1$  is earlier (smaller) than the date of an event  $e_2$ , that  $e_1$  is a cause of  $e_2$ .

### 7.1.2 From Partial Order to Total Order: The Notion of a Timestamp

The previous observation is the main limitation of any linear time system. But, fortunately, this is not a drawback if we have to totally order events while respecting the partial order  $\xrightarrow{ev}$ . To that end, in addition to its temporal coordinate (its date), let us associate with each event a spatial coordinate (namely the identity of the process  $p_i$  that issued this event).

**Fig. 7.3** A non-sequential observation obtained from linear time



**The Notion of a Timestamp** Hence let us define the *timestamp* of an event  $e$  as the pair  $\langle h, i \rangle$  such that:

- $p_i$  is the process that issued  $e$ , and
- $h$  is the date at which  $e$  has been issued.

Timestamps allow events to be totally ordered without violating causal precedence. Let  $e_1$  and  $e_2$  be two events whose timestamps are  $\langle h, i \rangle$  and  $\langle k, j \rangle$ , respectively. The causality-compliant total relation on the events, denoted  $\xrightarrow{to\_ev}$ , is defined as follows:

$$e_1 \xrightarrow{to\_ev} e_2 \stackrel{def}{=} (h < k) \vee ((h = k) \wedge (i < j)).$$

This total order is nothing more than the lexicographical order on the pairs made up of two integers, a date and a process identity. (It is of course assumed that no two processes have the same identity.)

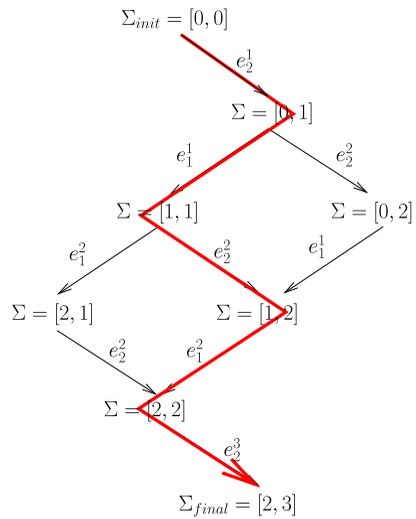
**Notation** In the following the notation  $\langle h, i \rangle < \langle k, j \rangle$  is used to denote  $(h < k) \vee ((h = k) \wedge (i < j))$ .

### 7.1.3 Relating Logical Time and Timestamps with Observations

The notion of a sequential observation was introduced in Chap. 6 devoted to global states, namely, it is a sequence (a) including the events generated by a distributed execution, and (b) respecting their partial ordering.

**Linear Time and Non-sequential Observation** Let us consider the distributed execution depicted in Fig. 7.3, in which the corresponding timestamp is associated with each event. Let us first consider all the events whose date is  $H = 1$ , then all the events whose date is  $H = 2$ , etc. This is indicated in the figure with bold lines, which partition the events according to their dates. This corresponds to a non-sequential observation, i.e., execution that could be observed by an observer able to see several events at the same time, namely all the events with the same logical date.

**Fig. 7.4**  
A sequential observation  
obtained from timestamps



**Timestamps and Sequential Observation** Figure 7.4 represents the lattice of global states of the distributed execution of Fig. 7.3. Let us now totally order all the events produced by this execution according to the relation  $\xrightarrow{to-ev}$ . We obtain the sequence of events  $e_2^1, e_1^1, e_2^2, e_1^2, e_2^3$ , which is denoted with a bold zigzag arrow on Fig. 7.4. It is easy to see that this sequence is nothing more than a particular sequential observation of the distributed execution depicted in Fig. 7.3.

More generally, the total order on timestamps defines a sequential observation of the corresponding distributed execution.

### 7.1.4 Timestamps in Action: Total Order Broadcast

Linear time and timestamps are particularly useful when one has to order operations or messages. The most typical case is the establishment of a total order on a set of requests, which have to be serviced one after the other. This use of timestamps will be addressed in Chap. 10, which is devoted to permission-based mutual exclusion.

This section considers another illustration of timestamps, namely, it presents a timestamp-based implementation of a high-level communication abstraction, called *total order broadcast*.

**The Total Order Broadcast Abstraction** Total order broadcast is a communication abstraction defined by two operations, denoted `to_broadcast()` and `to_deliver()`. Intuitively, `to_broadcast()` allows a process  $p_i$  to send a message  $m$  to all the processes (we then say “ $p_i$  to\_broadcasts  $m$ ”), while `to_deliver()` allows a process to receive such a message (we then say “ $p_i$  to\_delivers  $m$ ”). Moreover, all the messages that have been to\_broadcast must be to\_delivered in the same order at each

process, and this order has to respect the causal precedence order. To simplify the presentation, it is assumed that each message is unique (this can be easily realized by associating a pair  $\langle$ sequence number, sender identity $\rangle$  with each message).

**A Causality-Compliant Partial Order on Messages**  $M$  being the set of messages which are to\_broadcast during an execution, let  $\hat{M} = (M, \rightarrow_M)$  be the relation where  $\rightarrow_M$  is defined on  $M$  as follows. Given  $m, m' \in M$ ,  $m \rightarrow_M m'$  (and we say “ $m$  causally precedes  $m'$ ”) if:

- $m$  and  $m'$  have been to\_broadcast by the same process, and  $m$  has been to\_broadcast before  $m'$ , or
- $m$  has been to\_delivered by a process  $p_i$  before  $p_i$  to\_broadcasts  $m'$ , or
- there is a message  $m'' \in M$  such that  $m \rightarrow_M m''$  and  $m'' \rightarrow_M m'$ .

**Total Order Broadcast: Definition** The total order broadcast abstraction is formally defined by the following properties. Said differently, this means that, to be correct, an implementation of the total order broadcast abstraction has to ensure that these properties are always satisfied.

- Validity. If a process to\_delivers a message  $m$ , there is a process that has to\_broadcast  $m$ .
- Integrity. No message is to\_delivered twice.
- Total order. If a process to\_delivers  $m$  before  $m'$ , no process to\_delivers  $m'$  before  $m$ .
- Causal precedence order. If  $m \rightarrow_M m'$ , no process to\_delivers  $m'$  before  $m$ .
- Termination. If a process to\_broadcasts a message  $m$ , any process to\_delivers  $m$ .

The first four properties are safety properties. Validity relates the outputs to the inputs. It states that there is neither message corruption, nor message creation. Integrity states that there is no message duplication. Total order states that messages are to\_delivered in the same order at every process, while causal precedence states that this total order respects the message causality relation  $\rightarrow_M$ . Finally, the termination property is a liveness property stating that no message is lost.

**Principle of the Implementation** To simplify the description, we consider that the communication channels are FIFO. Moreover, each pair of processes is connected by a bidirectional channel.

The principle that underlies the implementation is very simple: it consists in associating a timestamp with each message, and to\_delivering the messages according to their timestamp order. As timestamps are totally ordered and this order respects causal precedence, we obtain both total order and causal precedence order properties.

To illustrate the main issue posed by associating appropriate timestamps with messages, and define accordingly a correct message delivery rule, let us consider Fig. 7.5. Independently one from the other,  $p_1$  and  $p_2$  to\_broadcast the messages  $m_1$  and  $m_2$ , respectively. Neither of  $p_1$  and  $p_2$  can immediately to\_deliver its message, otherwise the total order delivery property would be violated. The processes have to

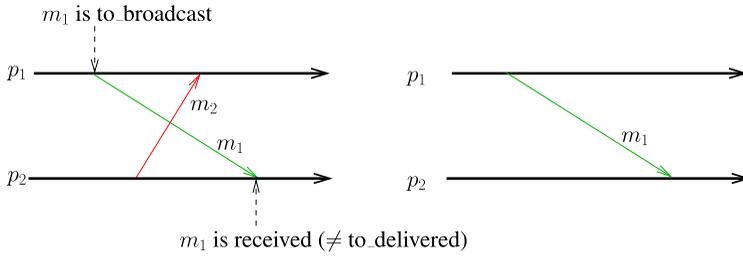


Fig. 7.5 Total order broadcast: the problem that has to be solved

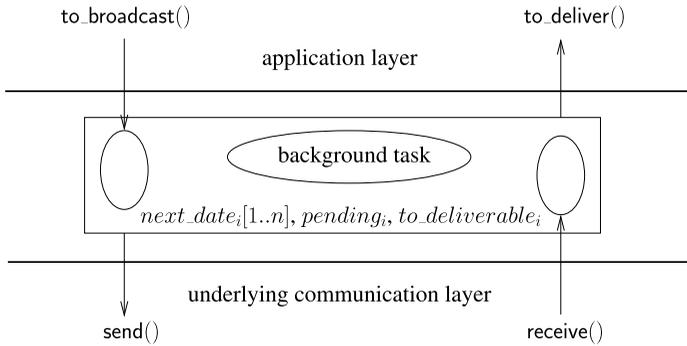


Fig. 7.6 Structure of the total order broadcast implementation

cooperate so that they to\_deliver  $m_1$  and  $m_2$  in the same order. This remains true if only  $p_1$  to\_broadcasts a message. This is because, when it issues to\_broadcast( $m_1$ ),  $p_1$  does not know whether  $p_2$  has independently issued to\_broadcast( $m_2$ ) or not.

It follows that a to\_broadcast message generates two distinct communication events at each process. The first one is associated with the reception of the message from the underlying communication network, while the second one is associated with its to\_delivery.

**Message Stability** A means to implement the same to\_delivery order at each process consists in providing each process  $p_i$  with information on the clock values of the whole set of processes. This local information can then be used by each process  $p_i$  to know which, among the to\_broadcast messages it has received and not yet to\_delivered, are *stable*, where message stability is defined as follows.

A message timestamped  $\langle k, j \rangle$  received by a process  $p_i$  is *stable* (at that process) if  $p_i$  knows that all the messages it will receive in the future will have a timestamp greater than  $\langle k, j \rangle$ . The main job of a timestamp-based implementation consists in ensuring the message stability at each process.

**Global Structure and Local Variables at a Process  $p_i$**  The structure of the implementation is described in Fig. 7.6. Each process  $p_i$  has a local module imple-

menting the operations to\_broadcast() and to\_deliver(). Each local module manages the following local variables.

- $clock_i[1..n]$  is an array of integers initialized to  $[0, \dots, 0]$ . The local variable  $clock_i[i]$  is the local clock of  $p_i$ , which implements the global linear time. Differently, for  $j \neq i$ ,  $clock_i[j]$  is the best approximation of the value of the local clock of  $p_j$ , as known by  $p_i$ . As the communication channels are FIFO,  $clock_i[j]$  contains the last value of  $clock_j[j]$  received by  $p_i$ . Hence, in addition to the fact that the set of local clocks  $\{clock_i\}_{1 \leq i \leq n}$  implement a global scalar clock, the local array  $clock_i[1..n]$  of each process  $p_i$  represents its current knowledge on the progress of the whole set of local logical clocks.
- $to\_deliverable_i$  is a sequence, initially empty (the empty sequence is denoted  $\epsilon$ ). This sequence contains the list of messages that (a) have been received by  $p_i$ , (b) have then been totally ordered, and (c) have not yet been to\_delivered. Hence,  $to\_deliverable_i$  is the list of messages that can be to\_delivered to the local upper layer application process.
- $pending_i$  is a set of pairs  $\langle m, \langle d, j \rangle \rangle$ , where  $m$  is a message whose timestamp is  $\langle d, j \rangle$ . Initially,  $pending_i = \emptyset$ .

**Description of the Implementation** The timestamp-based algorithm implementing the total order broadcast abstraction is described in Fig. 7.7. It is assumed that  $p_i$  does not interleave the execution of the statements at lines 1–4, lines 9–14, and lines 18–22. Let us recall that there is a bidirectional FIFO point-to-point communication channel connecting any pair of distinct processes. This algorithm works as described below.

When a process  $p_i$  invokes to\_broadcast( $m$ ), it first associates with  $m$  a new timestamp  $ts(m) = \langle clock_i[i], i \rangle$  (line 2). Then, it adds the pair  $\langle m, ts(m) \rangle$  to its local set  $pending_i$  (line 3), and sends to all the other processes the message TOBC( $m, ts(m)$ ) to inform them that a new message has been to\_broadcast (line 4). An invocation of to\_deliver() returns the first message in the local list  $to\_deliverable_i$  (line 5–8).

The behavior of a process  $p_i$  when it receives a message TOBC( $m, \langle sd\_date, j \rangle$ ) can be decomposed into two parts.

- The process  $p_i$  first modifies its local context according to content of the TOBC() message it has just received. It updates  $clock_i[j]$  (line 9), and adds the pair  $\langle m, \langle sd\_date, j \rangle \rangle$  to its local set  $pending_i$  (line 10).  
Let us notice that, as (a)  $clock_j[j]$  never decreases, (b)  $p_j$  increased it before sending the message TOBC( $m, -$ ), and (c) the channels are FIFO, it follows that we have  $clock_i[j] < sd\_date$  when the message TOBC( $m, \langle sd\_date, j \rangle$ ) is received. Hence the systematic update at line 9.
- The second set of statements executed by  $p_i$  (lines 11–14) is related to the update of its local clock  $clock_i[i]$ , and the dissemination of its new value to the other processes. This is to ensure both that (a) each message that has been to\_broadcast is eventually to\_delivered, and (b) message to\_deliveries satisfy total order.

If  $sd\_date \geq clock_i[i]$ ,  $p_i$  resets its local clock (line 12) to a value greater than  $sd\_date$  (as in the basic scalar clock algorithm of Fig. 7.1). Then  $p_i$  sends to each

```

operation to_broadcast( $m$ ) is
(1)  $clock_i[i] \leftarrow clock_i[i] + 1$ ;
(2) let  $ts(m) = \langle clock_i[i], i \rangle$ ;
(3)  $pending_i \leftarrow pending_i \cup \{ \langle m, ts(m) \rangle \}$ ;
(4) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do send TOBC( $m, ts(m)$ ) to  $p_j$  end for.

operation to_deliver( $m$ ) is
(5) wait ( $to\_deliverable_i \neq \epsilon$ );
(6) let  $m$  be the first message in the list  $to\_deliverable_i$ ;
(7) withdraw  $m$  from  $to\_deliverable_i$ ;
(8) return( $m$ ).

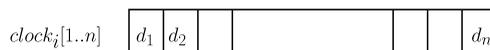
when TOBC( $m, \langle sd\_date, j \rangle$ ) is received do
(9)  $clock_i[j] \leftarrow sd\_date$ ;
(10)  $pending_i \leftarrow pending_i \cup \{ \langle m, \langle sd\_date, j \rangle \rangle \}$ ;
(11) if ( $sd\_date \geq clock_i[i]$ ) then
(12)    $clock_i[i] \leftarrow sd\_date + 1$ ;
(13)   for each  $k \in \{1, \dots, n\} \setminus \{i\}$  do send CATCH_UP( $clock_i[i], i$ ) to  $p_j$  end for
(14) end if.

when CATCH_UP( $last\_date, j$ ) is received do
(15)  $clock_i[j] \leftarrow last\_date$ .

background task  $T$  is
(16) repeat forever
(17) wait ( $pending_i \neq \emptyset$ );
(18) let  $\langle m, \langle d, k \rangle \rangle$  be the pair in  $pending_i$  with the smallest timestamp;
(19) if ( $\forall j \neq k : \langle d, k \rangle < \langle clock_i[j], j \rangle$ ) then
(20)   add  $m$  at the tail of  $to\_deliverable_i$ ;
(21)    $pending_i \leftarrow pending_i \setminus \{ \langle m, \langle d, k \rangle \rangle \}$ 
(22) end if
(23) end repeat.

```

Fig. 7.7 Implementation of total order broadcast (code for process  $p_i$ )



To\_delivery predicate for the message timestamped  $\langle d, k \rangle$ :  $\forall j \neq k : \langle d, k \rangle < \langle clock_i[j], j \rangle$

Fig. 7.8 To\_delivery predicate of a message at process  $p_i$

process a control message denoted CATCH\_UP(). This message carries the new clock value of  $p_i$  (line 13). As the channels are FIFO, it follows that when this message is received by  $p_k$ ,  $1 \leq k \neq i \leq n$ , this process has necessarily received all the messages TOBC() sent by  $p_i$  before this CATCH\_UP() message. This allows  $p_k$  to know if, among the to\_broadcast messages it has received, the one with the smallest timestamp is stable (Fig. 7.8).

Finally, a background task checks if some of the to\_broadcast messages which have been received by  $p_i$  are stable, and can consequently be moved from the set

$pending_i$  to the local sequence  $to\_partitionable_i$ . To that end, when  $pending_i \neq \emptyset$  (line 17),  $p_i$  looks for the message  $m$  with the smallest timestamp (line 18). Let  $ts(m) = \langle d, k \rangle$ . If, for any  $j \neq k$ ,  $ts(m)$  is smaller than  $\langle clock_i[j], j \rangle$ , it follows (from lines 11–14 executed by each process when it received the pair  $\langle m, ts(m) \rangle$ ) that any  $to\_broadcast$  message not yet received by  $p_i$  will have a timestamp greater than  $ts(m)$ . Hence, if the previous predicate is true, the message  $m$  is stable at  $p_i$  (Fig. 7.8). Consequently,  $p_i$  withdraws  $\langle m, ts(m) \rangle$  from the set  $pending_i$  (line 20) and adds  $m$  at the tail of  $to\_partitionable_i$  (line 21). As we will see in the proof, message stability at each process ensures that the  $to\_broadcast$  messages are added in the same order to all the sequences  $to\_partitionable_x, \leq x \leq n$ .

**Theorem 4** *The algorithm described in Fig. 7.7 implements the total order broadcast abstraction.*

*Proof* The validity property (neither corruption, nor creation of messages) follows directly from the reliability of the channels. The integrity property (no duplication) follows from the reliability of the underlying channels, the fact that no two  $to\_broadcast$  messages have the same timestamp, and the fact that when a message is added to  $to\_deliverable_i$ , it is suppressed from  $pending_i$ .

The proof of the termination property is by contradiction. Assuming that  $to\_broadcast$  messages are never  $to\_delivered$  by a process  $p_i$  (i.e., added to  $to\_deliverable_i$ ), let  $m$  be the one with the smallest timestamp, and let  $ts(m) = \langle d, k \rangle$ .

Let us observe that, each time a process  $p_j$  updates its local clock (to a greater value), it sends its new clock value to all processes. This occurs at lines 1 and 4, or at lines 12 and 13.

As each other process  $p_j$  receives the message  $TOBC(m, ts(m))$  sent by  $p_k$ , its local clock becomes greater than  $d$  (if it was not before). It then follows from the previous observation that a value of  $clock_j[j]$  greater than  $d$  becomes eventually known by each process, and we eventually have  $clock_i[j] > d$ . Hence, the  $to\_delivery$  predicate for  $m$  becomes eventually satisfied at  $p_i$ . The message  $m$  is consequently moved from  $pending_i$  to  $to\_deliverable_i$ , which contradicts the initial assumption and proves the termination property.

The proof of the total order property is also by contradiction. Let  $m_x$  and  $m_y$  be two messages timestamped  $ts(m_x) = \langle d_x, x \rangle$  and  $ts(m_y) = \langle d_y, y \rangle$ , respectively. Let us assume that  $ts(m_x) < ts(m_y)$ , and  $m_y$  is  $to\_delivered$  by a process  $p_i$  before  $m_x$  (i.e.,  $m_y$  is added to  $to\_deliverable_i$  before  $m_x$ ).

Just before  $m_y$  is added to  $to\_deliverable_i$ ,  $(m_y, ts(m_y))$  is the pair with the smallest timestamp in  $pending_i$ , and  $\forall j \neq y : \langle d_y, y \rangle < \langle clock_i[j], j \rangle$  (lines 18–19). It follows that we have then  $\langle d_x, x \rangle < \langle d_y, y \rangle < \langle clock_i[x], x \rangle$ . As (a)  $p_x$  sends only increasing values of its local clock (lines 1 and 4, and lines 12–13), (b)  $d_x < clock_i[x]$ , and (c) the channels are FIFO, it follows that  $p_i$  has received the message  $TOBC(m_x, ts(m_x))$  before the message carrying the value of  $clock_x[x]$  which entailed the update of  $clock_i[x]$  making true the predicate  $\langle d_y, y \rangle <$

$\langle clock_i[x], x \rangle$ . Consequently, as  $m_x$  is not yet to\_delivered (assumption), it follows that  $(m_y, ts(m_y))$  cannot be the pair with the smallest timestamp in  $pending_i$ . This contradicts the initial assumption. It follows that the messages that have been to\_broadcast and are to\_delivered, are to\_delivered at each process in the same total order (as defined by their timestamp).

The proof that the total order on timestamps respects the causal precedence order on to\_broadcast messages follows from the two following observations.

- A process  $p_i$  increases its local clock  $clock_i[i]$  each time it invokes to\_broadcast(). Hence, if  $p_i$  to\_broadcasts  $m$  before  $m'$ , we have  $ts(m) < ts(m')$ .
- Due to line 11, it follows that, after it has received a message TOBC( $m, \langle d, j \rangle$ ), we have  $clock_i[i] > d$ . It follows that, if later  $p_i$  to\_broadcasts a message  $m'$ , we necessarily have  $ts(m) < ts(m')$ .  $\square$

## 7.2 Vector Time

As we have seen, the domain of linear time is the set of non-negative integers, and a set of  $n$  logical clocks (one per process) allows us to associate dates with events, in such a way that the dates respect the causal precedence relation  $\xrightarrow{ev}$ . However, as we have seen, except for those events that have the same date, linear dates do not allow us to determine with certainty if one event belongs to the causes of another one. The aim of vector time (which is implemented by vector clocks) is to solve this issue. Vector time, and its capture by vector clocks, was simultaneously and independently proposed in 1988 by C.J. Fidge, F. Mattern, and F. Schmuck.

### 7.2.1 Vector Time and Vector Clocks

**Vector Time: Definition** A vector clock system is a mechanism that is able to associate dates with events (or local states) in such a way that the comparison of their dates allows us to determine if the corresponding events are causally related or not, and if they are which one belongs to the cause of the other. Vector time is the time notion captured by vector clocks.

More precisely, let  $date(e)$  be the date associated with an event  $e$ . The aim of a vector clock system is to provide us with a time domain in which any two dates either can be compared ( $<$ ,  $>$ ), or are incomparable (denoted  $||$ ), in such a way that the following properties are satisfied

- $\forall e_1, e_2: (e_1 \xrightarrow{ev} e_2) \Leftrightarrow date(e_1) < date(e_2)$ , and
- $\forall e_1, e_2: (e_1 || e_2) \Leftrightarrow date(e_1) || date(e_2)$  (the dates cannot be compared).

This means that the dating system has to be in perfect agreement with the causal precedence relation  $\xrightarrow{ev}$ . To attain this goal, vector time considers that the time domain is made up of all the integer vectors of size  $n$  (the number of processes).

```

when producing an internal event  $e$  do
(1)  $vc_i[i] \leftarrow vc_i[i] + 1$ ;
(2) Produce event  $e$ . % The date of  $e$  is  $vc_i[1..n]$ .

when sending  $MSG(m)$  to  $p_j$  do
(3)  $vc_i[i] \leftarrow vc_i[i] + 1$ ; %  $vc_i[1..n]$  is the sending date of the message.
(4) send  $MSG(m, vc_i[1..n])$  to  $p_j$ .

when  $MSG(m, vc)$  is received from  $p_j$  do
(5)  $vc_i[i] \leftarrow vc_i[i] + 1$ ;
(6)  $vc_i[1..n] \leftarrow \max(vc_i[1..n], vc[1..n])$ . %  $vc_i[1..n]$  is the date of the receive event.

```

**Fig. 7.9** Implementation of a vector clock system (code for process  $p_i$ )

**Vector Clock: Definition** To implement vector time, each process  $p_i$  manages a vector of non-negative integers  $vc_i[1..n]$ , initialized to  $[0, \dots, 0]$ . This vector is such that:

- $vc_i[i]$  counts the number of events produced by  $p_i$ , and
- $vc_i[j]$ ,  $j \neq i$ , counts the number of events produced by  $p_j$ , as known by  $p_i$ .

More formally, let  $e$  be an event produced by some process  $p_i$ . Just after  $p_i$  has produced  $e$  we have (where  $1(k, i) = 1$  if  $p_k$  is  $p_i$ , and  $1(k, i) = 0$  otherwise):

$$vc_i[k] = |\{f \mid (f \text{ has been produced by } p_k) \wedge (f \xrightarrow{ev} e)\}| + 1(k, i).$$

Hence,  $vc_i[k]$  is the number of events produced by  $p_k$  in the causal past of the event  $e$ . The term  $1(k, i)$  is to count the event  $e$ , which has been produced by  $p_i$ . The value of  $vc_i[1..n]$  is the vector date of the event  $e$ .

**Vector Clock: Algorithm** The algorithm implementing the vector clock system has exactly the same structure as the one for linear time (Fig. 7.1). The difference lies in the fact that only the entry  $i$  of the local vector clock of  $p_i$  (i.e.,  $vc_i[i]$ ) is increased each time it produces a new event, and each message  $m$  piggybacks the current value of the vector time, which defines the sending time of  $m$ . This value allows the receiver to update its local vector clock, so that the date of the receive event is after both the date of the sending event associated with  $m$  and the immediately preceding local event produced by the receiver process. The operator  $\max()$  on integers is extended to vectors as follows (line 5):

$$\begin{aligned} \max(v1, v2) \\ = [\max(v1[1], v2[1]), \dots, \max(v1[j], v2[j]), \dots, \max(v1[n], v2[n])]. \end{aligned}$$

Let us observe that, for any pair  $(i, k)$ , it follows directly from the vector clock algorithm that (a)  $vc_i[k]$  never decreases, and (b) at any time,  $vc_i[k] \leq vc_k[k]$ .

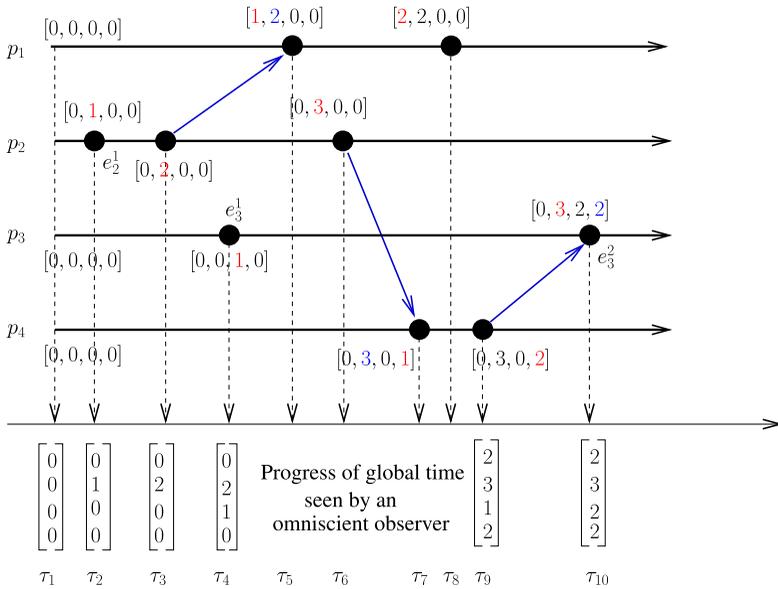


Fig. 7.10 Time propagation in a vector clock system

**Example of Time Propagation** An example of an execution of the previous algorithm is depicted in Fig. 7.10. The four local vector clocks are initialized to  $[0, 0, 0, 0]$ . Then,  $p_2$  produces an internal event dated  $[0, 1, 0, 0]$ . Its next event is the sending of a message to  $p_1$ , dated  $[0, 2, 0, 0]$ . The reception of this message by  $p_1$  is its first event, and consequently the reception of this message is dated  $[1, 2, 0, 0]$ , etc.

The bottom of the figure shows what could be seen by an omniscient external observer. The global time seen by this observer is defined as follows: At any time  $\tau_x$ , it sees how many events have been produced by each process up to  $\tau_x$ . Hence, the current value of this global time is  $[vc_1[1], vc_2[2], vc_3[3], vc_4[4]]$ . Initially, no process has yet produced events, and the observer sees the global vector time  $[0, 0, 0, 0]$  at external time  $\tau_1$ . Then, at  $\tau_3$ , it sees the sending by  $p_2$  of a message to  $p_1$  at the global time  $[0, 2, 0, 0]$ , etc. Some global time instants are indicated on the figure. The value of the last global time seen by the omniscient observer is  $[2, 3, 2, 2]$ .

The global time increases when events are produced. As the observer is omniscient, this is captured by the fact that the global time values it sees increase in the sense that, if we consider any two global time vectors  $GD1$  followed by  $GD2$ , we have  $(\forall k : GD1[k] \leq GD2[k]) \wedge (\exists k : GD1[k] < GD2[k])$ .

**Notation** Given two vectors  $vc1$  and  $vc2$ , both of size  $n$ , we have

- $vc1 \leq vc2 \stackrel{def}{=} (\forall k \in \{1, \dots, n\} : vc1[k] \leq vc2[k])$ .
- $vc1 < vc2 \stackrel{def}{=} (vc1 \leq vc2) \wedge (vc1 \neq vc2)$ .

- $vc1 || vc2 \stackrel{def}{=} \neg(vc1 \leq vc2) \wedge \neg(vc1 \leq vc2)$ .

When considering Fig. 7.10, we have  $[0, 2, 0, 0] < [0, 3, 2, 2]$ , and  $[0, 2, 0, 0] || [0, 0, 1, 0]$ .

## 7.2.2 Vector Clock Properties

The following theorem characterizes the power of vector clocks.

**Theorem 5** *Let  $e.vc$  be the vector date associated with event  $e$ , by the algorithm of Fig. 7.9. These dates are such that, for any two distinct events  $e_1$  and  $e_2$  we have (a)  $(e_1 \xrightarrow{ev} e_2) \Leftrightarrow (e_1.vc < e_2.vc)$ , and (b)  $(e_1 || e_2) \Leftrightarrow (e_1.vc || e_2.vc)$ .*

*Proof* The proof is made up of three cases.

- $(e_1 \xrightarrow{ev} e_2) \Rightarrow (e_1.vc < e_2.vc)$ . This case follows directly from the fact that the vector time increases along all causal paths (lines 1 and 3–6).
- $(e_1.vc < e_2.vc) \Rightarrow (e_1 \xrightarrow{ev} e_2)$ . Let  $p_i$  be the process that produced event  $e_1$ . We have  $e_1.vc < e_2.vc \Rightarrow e_1.vc[i] \leq e_2.vc[i]$ . Let us observe that only process  $p_i$  can entail an increase of the entry  $i$  of any vector (if  $p_i$  no longer increases  $vc_i[i] = a$  at line 1, no process  $p_j$  can be such that  $vc_j[i] > a$ ). We conclude from this observation, the code of the algorithm (vector time increases only along causal paths), and  $e_1.vc[i] \leq e_2.vc[i]$  that there is a causal path from  $e_1$  to  $e_2$ .
- $(e_1 || e_2) \Leftrightarrow (e_1.vc || e_2.vc)$ . By definition we have  $(e_1 || e_2) = (\neg(e_1 \xrightarrow{ev} e_2) \wedge \neg(e_2 \xrightarrow{ev} e_1))$ . It follows from the previous items that  $\neg(e_1 \xrightarrow{ev} e_2) \Leftrightarrow \neg(e_1.vc \leq e_2.vc)$ , i.e.,  $\neg(e_1 \xrightarrow{ev} e_2) \Leftrightarrow (e_1.vc > e_2.vc) \vee (e_1.vc || e_2.vc)$ . Similarly,  $\neg(e_2 \xrightarrow{ev} e_1) \Leftrightarrow (e_2.vc > e_1.vc) \vee (e_2.vc || e_1.vc)$ . Combining the previous observations, and observing that we cannot have simultaneously  $(e_1.vc > e_2.vc) \wedge (e_2.vc > e_1.vc)$ , we obtain  $(e_1 || e_2) \Leftrightarrow (e_1.vc || e_2.vc)$ .  $\square$

The next corollary follows directly from the previous theorem.

**Corollary 1** *Given the dates of two events, determining if these events are causally related or not can require up to  $n$  comparisons of integers.*

**Reducing the Cost of Comparing Two Vector Dates** As the cost of comparing two dates is  $O(n)$ , an important question is the following: Is it possible to add control information to a date in order to reduce the cost of their comparison? If the events are produced by the same process  $p_i$ , a simple comparison of the  $i$ th entry of their vector dates allows us to conclude. More generally, given an event  $e$  produced by a process  $p_i$ , let us associate with  $e$  a timestamp defined as the pair  $\langle e.vc, i \rangle$ . We have then the following theorem from which it follows that, thanks to the knowledge of the process that produced an event, the cost of deciding if two events are or not causally related is reduced to two comparisons of integers.

**Theorem 6** Let  $e_1$  and  $e_2$  be events timestamped  $\langle e_1.vc, i \rangle$  and  $\langle e_2.vc, j \rangle$ , respectively, with  $i \neq j$ . We have  $(e_1 \xrightarrow{ev} e_2) \Leftrightarrow (e_1.vc[i] \leq e_2.vc[i])$ , and  $(e_1 \parallel e_2) \Leftrightarrow ((e_1.vc[i] > e_2.vc[i]) \wedge (e_2.vc[j] > e_1.vc[j]))$ .

*Proof* Let us first observe that time increases only along causal paths, and only the process that produced an event entails an increase of the corresponding entry in a vector clock (Observation O). The proof considers each case separately.

- $(e_1 \xrightarrow{ev} e_2) \Leftrightarrow (e_1.vc[i] \leq e_2.vc[i])$ .  
 If  $e_1 \xrightarrow{ev} e_2$ , there is a causal path from  $e_1$  to  $e_2$ , and we have  $e_1.vc \leq e_2.vc$  (Theorem 6), from which  $e_1.vc[i] \leq e_2.vc[i]$  follows.  
 If  $e_1.vc[i] \leq e_2.vc[i]$ , it follows from observation O that there is a causal path from  $e_1$  to  $e_2$ .
- $(e_1 \parallel e_2) \Leftrightarrow ((e_1.vc[i] > e_2.vc[i]) \wedge (e_2.vc[j] > e_1.vc[j]))$ .  
 As, at any time, we have  $vc_j[i] \leq vc_i[i]$ ,  $p_i$  increases  $vc_i[i]$  when it produces  $e_1$ , it follows from the fact that there is no causal path from  $e_1$  to  $e_2$  and observation O that  $e_1.vc[i] > e_2.vc[i]$ . The same applies to  $e_2.vc[j]$  with respect to  $e_1.vc$ .  
 In the other direction, we conclude from  $e_1.vc[i] > e_2.vc[i]$  that there is no causal path from  $e_1$  to  $e_2$  (otherwise we would have  $e_1.vc[i] \leq e_2.vc[i]$ ). Similarly,  $e_2.vc[j] > e_1.vc[j]$  implies that there is no causal path from  $e_2$  to  $e_1$ .  $\square$

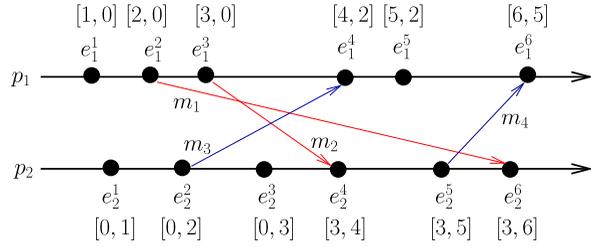
To illustrate this theorem, let us consider Fig. 7.10, where the first event of  $p_2$  is denoted  $e_2^1$ , first event of  $p_3$  is denoted  $e_3^1$ , and the second event of  $p_3$  is denoted  $e_3^2$ . Event  $e_2^1$  is timestamped  $\langle [0, 1, 0, 0], 2 \rangle$ ,  $e_3^1$  is timestamped  $\langle [0, 0, 1, 0], 2 \rangle$ , and  $e_3^2$  is timestamped  $\langle [0, 3, 2, 2], 2 \rangle$ . As  $e_2^1.vc[2] = 1 \leq e_3^2.vc[2] = 3$ , we conclude  $e_2^1 \xrightarrow{ev} e_3^2$ . As  $e_2^1.vc[2] = 1 > e_3^1.vc[2] = 0$  and  $e_3^1.vc[3] = 1 > e_2^1.vc[3] = 0$ , we conclude  $e_2^1 \parallel e_3^1$ .

### 7.2.3 On the Development of Vector Time

When considering vector time, messages put restrictions on the development of time. More precisely, let  $m$  be a message sent by a process  $p_i$  at local time  $vc_i[i] = a$  and received by a process  $p_j$  at local time  $vc_j[j] = b$ . This message induces the restriction that no event  $e$  can be dated  $e.vc$  such that  $e.vc[i] < a$  and  $e.vc[j] \geq b$ . This restriction simply states that there is no event such that there is a message whose reception belongs to its causal past while its sending does not.

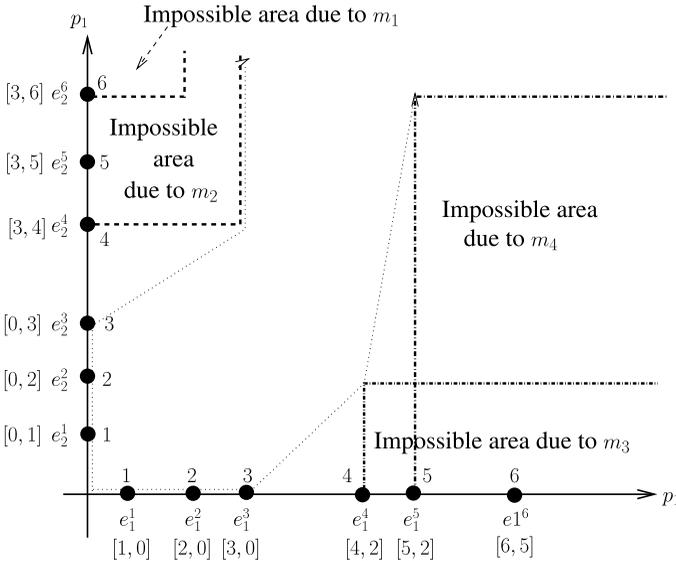
To illustrate this restriction created by messages on the vector time domain, let us consider the execution depicted in Fig. 7.11. There are two processes, and each of them produces six events. The vector time date of each event is indicated above or below the corresponding event. Let us notice that the channel from  $p_1$  to  $p_2$  is not FIFO ( $p_1$  sends  $m_1$  before  $m_2$ , but  $p_2$  receives  $m_2$  before  $m_1$ ).

**Fig. 7.11** On the development of time (1)



Let us consider the message  $m_2$ . As its sending time is  $[3, 0]$  and its receive time is  $[3, 4]$ , it follows that it is impossible for an external observer to see a global time  $GD$  such that  $GD[1] < 3$  and  $GD[2] \geq 4$ . The restrictions on the set of possible vector dates due to the four messages exchanged in the computation are depicted in Fig. 7.12. Each message prevents some vector clock values from being observed. As an example, the date  $[2, 5]$  cannot exist, while the vector date  $[3, 3]$  can be observed by an external observer.

The borders of the area including the vector dates that could be observed by external observers are indicated with dotted lines in the figure. They correspond to the history (sequence of events) of each process.



**Fig. 7.12** On the development of time (2)

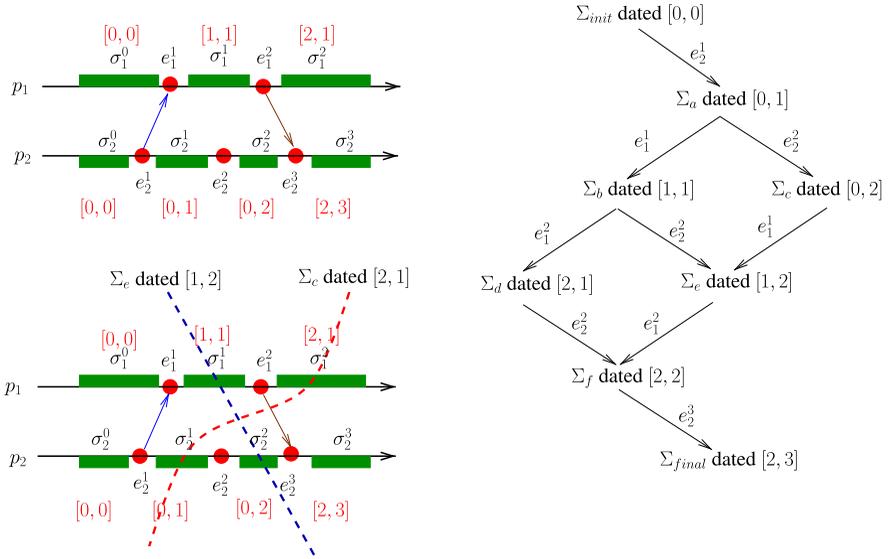


Fig. 7.13 Associating vector dates with global states

### 7.2.4 Relating Vector Time and Global States

Let us consider the distributed execution described on the left top of Fig. 7.13. Vector dates of events and local states are indicated in this figure. Both initial local states  $\sigma_1^0$  and  $\sigma_2^0$  are dated  $[0, 0]$ . Then, each  $\sigma_i^x$  inherits the vector date of the event  $e_i^x$  that generated it. As an example the date of the local state  $\sigma_2^2$  is  $[0, 2]$ .

The corresponding lattice of global states is described at the right of Fig. 7.13. In this lattice, a vector date is associated with each global state as follows: the  $i$ th entry of the vector is the number of events produced by  $p_i$ . This means that, when considering the figure, the vector date of the global state  $[\sigma_i^x, \sigma_j^y]$  is  $[x, y]$ . (This dating system for global states, which is evidently based on vector clocks, was implicitly introduced and used in Sect. 6.3.2, where the notion of a lattice of global states was introduced.) Trivially, the vector time associated with global dates increases along each path of the lattice.

Let us recall that the greatest lower bound (GLB) of a set of vertices of a lattice is their greatest common predecessor, while the least upper bound (LUB) is their least common successor. Due to the fact that the graph is a lattice, each of the GLB and the LUB of a set of vertices (global states) is unique.

An important consequence of the fact that the set of consistent global states is a lattice and the associated dating of global states is the following. Let us consider two global states  $\Sigma'$  and  $\Sigma''$  whose dates are  $[d'_1, \dots, d'_n]$  and  $[d''_1, \dots, d''_n]$ , respectively. Let  $\Sigma^- = \text{GLB}(\Sigma', \Sigma'')$  and  $\Sigma^+ = \text{LUB}(\Sigma', \Sigma'')$ . We have

- $date(\Sigma^-) = date(\text{GLB}(\Sigma', \Sigma'')) = [\min(d'_1, d''_1), \dots, \min(d'_n, d''_n)]$ , and

- $date(\Sigma^+) = date(\text{LUB}(\Sigma', \Sigma'')) = [\max(d'_1, d''_1), \dots, \max(d'_n, d''_n)]$ .

As an example, let us consider the global states denoted  $\Sigma_d$  and  $\Sigma_e$  in the lattice depicted at the left of Fig. 7.13. We have  $\Sigma_b = \text{GLB}(\Sigma_d, \Sigma_e)$  and  $\Sigma_f = \text{LUB}(\Sigma_d, \Sigma_e)$ . It follows that we have  $date(\Sigma_b) = [\min(2, 1), \min(1, 2)] = [1, 1]$ . Similarly, we have  $date(\Sigma_f) = [\max(2, 1), \max(1, 2)] = [2, 2]$ . The consistent cuts associated with  $\Sigma_d$  and  $\Sigma_e$  are depicted in the bottom of the left side of Fig. 7.13.

These properties of the vector dates associated with global states are particularly important when one has to detect properties on global states. It allows processes to obtain information on the lattice without having to build it explicitly. At the operational level, processes have to compute vector dates identifying consistent global states which are relevant with respect to the property of interest (see Sect. 7.2.5).

### 7.2.5 Vector Clocks in Action:

#### *On-the-Fly Determination of a Global State Property*

To illustrate the use of vector clocks, this section presents an algorithm that determines the first global state of a distributed computation that satisfies a conjunction of stable local predicates.

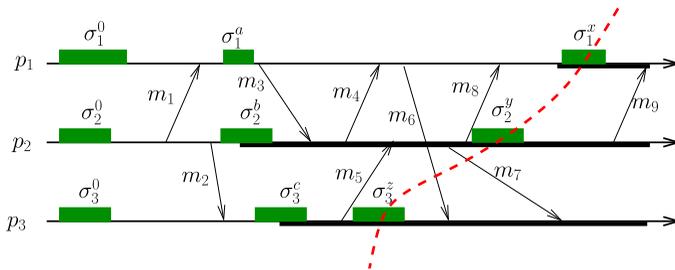
**Conjunction of Stable Local Predicates** A predicate is *local* to a process  $p_i$  if it is only on local variables of  $p_i$ . Let  $LP_i$  be a predicate local to process  $p_i$ . The fact that the local state  $\sigma_i$  of  $p_i$  satisfies the predicate  $LP_i$  is denoted  $\sigma_i \models LP_i$ .

Let  $\{LP_i\}_{1 \leq i \leq n}$  be a set of  $n$  local predicates, one per process. (If there is no local predicate for a process  $p_x$ , we can consider a fictitious local predicate  $LP_x = \text{true}$  satisfied by all local states of  $p_x$ .) The predicate  $\bigwedge_i LP_i$  is a global predicate, called *conjunction of local predicates*. Let  $\Sigma = [\sigma_1, \dots, \sigma_n]$  be a consistent global state. We say that  $\Sigma$  satisfies the global predicate  $\bigwedge_i LP_i$ , if  $\bigwedge_i (\sigma_i \models LP_i)$ . This is denoted  $\Sigma \models \bigwedge_i LP_i$ .

A predicate is *stable* if, once true, it remains true forever. Hence, if a local state  $\sigma_i$  satisfies a local stable predicate  $LP_i$ , all the local states that follow  $\sigma_i$  in  $p_i$ 's local history satisfy  $LP_i$ . Let us observe that, if each local predicate  $LP_i$  is stable, so is the global predicate  $\bigwedge_i LP_i$ .

This section presents a distributed algorithm that computes, on the fly and without using additional control messages, the first consistent global state that satisfies a conjunction of stable local predicates. The algorithm, which only adds control data to application messages, assumes that the application sends “enough” messages (the meaning of “enough” will appear clearly in the description of the algorithm).

**On the Notion of a “First” Global State** The consistent global state  $\Sigma$  defined by a process  $p_i$  from the vector date  $first_i[1..n]$  is the *first* global state satisfying  $\bigwedge_i LP_i$ , in the following sense: There is no global state  $\Sigma'$  such that  $(\Sigma' \neq \Sigma) \wedge (\Sigma' \models \bigwedge_i LP_i) \wedge (\Sigma' \xrightarrow{\Sigma} \Sigma)$ .



**Fig. 7.14** First global state satisfying a global predicate (1)

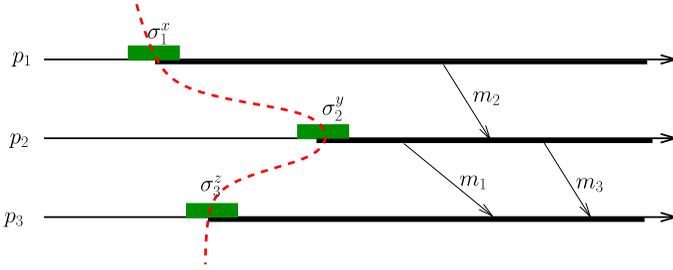
**Where Is the Difficulty** Let us consider the execution described in Fig. 7.14. The predicates  $LP_1$ ,  $LP_2$  and  $LP_3$  are satisfied from the local states  $\sigma_1^x$ ,  $\sigma_2^b$ , and  $\sigma_3^c$ , respectively. (The fact that they are stable is indicated by a bold line on the corresponding process axis.) Not all local states are represented; the important point is that  $\sigma_2^y$  is the state of  $p_2$  after it has sent the message  $m_8$  to  $p_1$ , and  $\sigma_3^z$  is the state of  $p_3$  after it has sent the message  $m_5$  to  $p_2$ . The first consistent global state satisfying  $LP_1 \wedge LP_2 \wedge LP_3$  is  $\Sigma = [\sigma_1^x, \sigma_2^y, \sigma_3^z]$ .

Using causality created by message exchanges and appropriate information piggybacked by messages, the processes can “learn” information related to the predicate detection. More explicitly, we have the following when looking at the figure.

- When  $p_1$  receives  $m_1$ , it learns nothing about the predicate detection (and similarly for  $p_3$  when it receives  $m_2$ ).
- When  $p_1$  receives  $m_4$  (sent by  $p_2$ ), it can learn that (a) the global state  $[\sigma_1^0, \sigma_2^b, \sigma_3^0]$  is consistent and (b) it “partially” satisfies the global predicate, namely,  $\sigma_2^b \models LP_2$ .
- When  $p_2$  receives the message  $m_3$  (from  $p_1$ ), it can learn that the global state  $[\sigma_1^a, \sigma_2^b, \sigma_3^0]$  is consistent and such  $\sigma_2^a \models LP_2$ . When it later receives the message  $m_5$  (from  $p_3$ ), it can learn that the global state  $[\sigma_1^a, \sigma_2^b, \sigma_3^c]$  is consistent and such  $(\sigma_2^b \models LP_2) \wedge (\sigma_3^c \models LP_3)$ . Process  $p_3$  can learn the same when it receives the message  $m_7$  (sent by  $p_2$ ).
- When  $p_1$  receives the message  $m_8$  (sent by  $p_2$ ), it can learn that, while  $LP_1$  is not yet locally satisfied, the global state  $[\sigma_1^a, \sigma_2^b, \sigma_3^c]$  is the first consistent global state that satisfies  $LP_2$  and  $LP_3$ .
- Finally, when  $p_1$  produces the internal event giving rise to  $\sigma_1^x$ , it can learn that the first consistent global state satisfying the three local predicates is  $[\sigma_1^x, \sigma_2^y, \sigma_3^z]$ . The corresponding consistent cut is indicated by a dotted line on the figure.

Let us recall that the vector date of the local state  $\sigma_2^y$  is the date of the preceding event, which is the sending of  $m_8$ , and this date is piggybacked by  $m_5$ . Similarly, the date of  $\sigma_3^z$  is the date of the sending of  $m_5$ .

Another example is given in Fig. 7.15. In this case, due to the flow of control created by the exchange of messages, it is only when  $p_3$  receives  $m_3$  that it can learn



**Fig. 7.15** First global state satisfying a global predicate (2)

that  $[\sigma_1^x, \sigma_2^y, \sigma_3^z]$  is the first consistent global state satisfying  $\bigwedge_i LP_i$ . As previously, the corresponding consistent cut is indicated by a dotted line on the figure.

As no control messages are allowed, it is easy to see that, in some scenarios, enough application messages have to be sent after  $\bigwedge_i LP_i$  is satisfied, in order to compute the vector date of the first consistent global state satisfying the global predicate  $\bigwedge_i LP_i$ .

**Local Control Variables** In order to implement the determination of the first global state satisfying a conjunction of stable local predicates, each process  $p_i$  manages the following local variables:

- $vc_i[1..n]$  is the local vector clock.
- $sat_i$  is a set, initially empty, of process identities. It has the following meaning:  $(j \in sat_i) \Leftrightarrow (p_i \text{ knows that } p_j \text{ has entered a local state satisfying } LP_j)$ .
- $done_i$  is Boolean, initialized to *false*, and then set to the value *true* by  $p_i$  when  $LP_i$  becomes satisfied for the first time. As  $LP_i$  is a stable predicate,  $done_i$  then keeps that value forever.
- $first_i$  is the vector date of the first consistent global state, known by  $p_i$ , for which all the processes in  $sat_i$  satisfy their local predicate. Initially,  $first_i = [0, \dots, 0]$ . Hence,  $[\sigma_1^{first[1]}, \dots, \sigma_n^{first[n]}]$  is this global state, and we have  $\forall j \in sat_i : \sigma_j^{first[j]} \models LP_j$ .

As an example, considering Fig. 7.14, after  $p_1$  has received the message  $m_4$ , we have  $sat_1 = \{2\}$  and  $first_1 = [0, b, 0]$ . After it has received the message  $m_8$ , we have  $sat_1 = \{2, 3\}$  and  $first_1 = [0, b, c]$ .

**The Algorithm** The algorithm is described in Fig. 7.16. It ensures that if consistent global states satisfy  $\bigwedge LP_i$ , at least one process will compute the vector date of the first of them. As already indicated, this is under the assumption that the processes send enough application messages.

Before producing a new event,  $p_i$  always increases its local clock  $vc_i[i]$  (lines 7, 10, and 13). If the event  $e$  is an internal event, and  $LP_i$  has not yet been satisfied (i.e.,  $done_i$  is false),  $p_i$  invokes the operation `check_lp()` (line 9). If its current local state  $\sigma$  satisfies  $LP_i$  (line 1),  $p_i$  adds its identity to  $sat_i$  and, as it is the first time that

```

internal operation check_lp( $\sigma$ ) is
(1) if ( $\sigma \models LP_i$ ) then
(2)    $sat_i \leftarrow sat_i \cup \{i\}; first_i[1..n] \leftarrow vc_i[1..n]; done_i \leftarrow true;$ 
(3)   if ( $sat_i = \{1, \dots, n\}$ ) then
(4)      $first_i[1..n]$  is the vector date of the first global state satisfying  $\bigwedge_j LP_j$ 
(5)   end if
(6) end if.

when producing an internal event  $e$  do
(7)  $vc_i[i] \leftarrow vc_i[i] + 1;$ 
(8) Produce event  $e$  and move to the next state  $\sigma$ ;
(9) if ( $\neg done_i$ ) then check_lp( $\sigma$ ) end if.

when sending MSG( $m$ ) to  $p_j$  do
(10)  $vc_i[i] \leftarrow vc_i[i] + 1;$  move to the next local state  $\sigma$ ;
(11) if ( $\neg done_i$ ) then check_lp( $\sigma$ ) end if;
(12) send MSG( $m, vc_i, sat_i, first_i$ ) to  $p_j$ .

when MSG( $m, vc, sat, first$ ) is received from  $p_j$  do
(13)  $vc_i[i] \leftarrow vc_i[i] + 1; vc_i \leftarrow \max(vc_i[1..n], vc[1..n]);$ 
(14) move to the next local state  $\sigma$ ;
(15) if ( $\neg done_i$ ) then check_lp( $\sigma$ ) end if;
(16) if ( $sat \not\subseteq sat_i$ ) then
(17)    $sat_i \leftarrow sat_i \cup sat; first_i \leftarrow \max(first_i[1..n], first[1..n]);$ 
(18)   if ( $sat_i = \{1, \dots, n\}$ ) then
(19)      $first_i[1..n]$  is the vector date of the first global state satisfying  $\bigwedge_j LP_j$ 
(20)   end if
(21) end if.

```

**Fig. 7.16** Detection the first global state satisfying  $\bigwedge_i LP_i$  (code for process  $p_i$ )

$LP_i$  is satisfied, it defines accordingly  $first_i$  as being  $vc_i$  (which is the vector date of the global state associated with the causal past of the event  $e$  currently produced by  $p_i$ , line 2). Process  $p_i$  then checks if the consistent global state defined by the vector date  $first_i (= vc_i)$  satisfies the whole global predicate  $\bigwedge_j LP_j$  (lines 3–4). If it is the case,  $p_i$  has determined the first global state satisfying the global predicate.

If the event  $e$  is the sending of a message by  $p_i$  to  $p_j$ , before sending the message, process  $p_i$  first moves to its next local state  $\sigma$  (line 10), and does the same as if  $e$  was an internal event. The important point is that, in addition to the application message,  $p_i$  sends to  $p_j$  (line 12) its full current state (from the global state determination point of view), which consists of  $vc_i$  (current vector time at  $p_i$ ),  $sat_i$  (processes whose local predicates are satisfied), and  $first_i$  (date of the consistent global state satisfying the local predicates of the processes in  $sat_i$ ).

If the event  $e$  is the reception by  $p_i$  of a message sent by  $p_j$ ,  $p_i$  updates first its vector clock (line 13), and moves to its next local state (line 14). As in the previous cases, if  $LP_i$  has not yet been satisfied,  $p_i$  then invokes check\_lp() (line 15). Finally, if  $p_i$  learns something new with respect to local predicates (test of line 16), it “adds” what it knew before ( $sat_i$  and  $first_i[1..n]$ ) with what it learns ( $sat$  and  $first[1..n]$ ). The new value of  $first_i[1..n]$  is the vector date of the first consistent global state

in which all the local states of the processes in  $sat_i \cup sat$  satisfy their local predicates. Finally,  $p_i$  checks if the global state defined by  $first_i[1..n]$  satisfies all local predicates (lines 18–20).

When looking at the executions described in Figs. 7.14 and 7.15, we have the following. In Fig. 7.14,  $p_1$  ends the detection at line 4 after it has produced the event that gave rise to  $\sigma_1^x$ . In Fig. 7.15,  $p_3$  ends the detection at line 19 after it has received the protocol message  $MSG(m_3, [-, -, -], \{1, 2\}, [x, y, 0])$ . Just before it receives this message we have  $sat_3 = \{2, 3\}$  and  $first_3 = [0, y, z]$ .

## 7.2.6 Vector Clocks in Action: On-the-Fly Determination of the Immediate Predecessors

**The Notion of a Relevant Event** At some abstraction level, only a subset of events are relevant. As an example, in some applications only the modification of some local variables, or the processing of specific messages, are relevant. Given a distributed computation  $\widehat{H} = (H, \xrightarrow{ev})$ , let  $R \subset H$  be the subset of its events that are defined as relevant.

Let us accordingly define the causal precedence relation on these events, denoted  $\xrightarrow{re}$ , as follows

$$\forall e_1, e_2 \in R : (e_1 \xrightarrow{re} e_2) \Leftrightarrow (e_2 \xrightarrow{ev} e_1).$$

This relation is nothing else than the projection of  $\xrightarrow{ev}$  on the elements of  $R$ . The pair  $\widehat{R} = (R, \xrightarrow{re})$  constitutes an abstraction of the distributed computation  $\widehat{H} = (H, \xrightarrow{ev})$ .

Without loss of generality, we consider that the set of relevant events consists only of internal events. Communication events are low-level events which, without being themselves relevant, participate in the establishment of causal precedence on relevant events. If a communication event has to be explicitly observed as relevant, an associated relevant internal event can be generated just before or after it.

An example of a distributed execution with both relevant events and non-relevant events is described in Fig. 7.17. The relevant events are represented by black dots, while the non-relevant events are represented by white dots.

The management of vector clocks restricted to relevant events (Fig. 7.18) is a simplified version of the one described in Fig. 7.9. As we can see, despite the fact that they are not relevant, communication events participate in the tracking of causal precedence. Given a relevant  $e$  timestamped  $\langle e.vc[1..n], i \rangle$ , the integer  $e.vc[j]$  is the number of relevant events produced by  $p_j$  and known by  $p_i$ .

**The Notion of an Immediate Predecessor and the Immediate Predecessor Tracking Problem** Given two events  $e_1, e_2 \in R$ , the relevant event  $e_1$  is an *immediate predecessor* of the relevant event  $e_2$  if

$$(e_1 \xrightarrow{re} e_2) \wedge (\nexists e \in R : (e_1 \xrightarrow{re} e) \wedge (e \xrightarrow{re} e_2)).$$

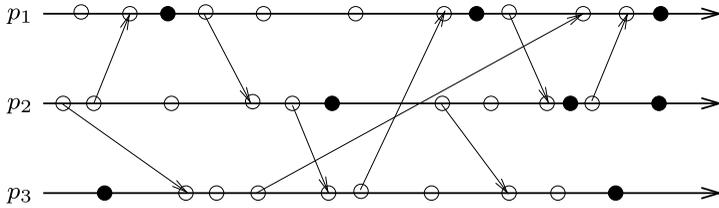


Fig. 7.17 Relevant events in a distributed computation

```

when producing a relevant internal event  $e$  do
  (1)  $vc_i[i] \leftarrow vc_i[i] + 1;$ 
  (2) Produce the relevant event  $e$ . % The date of  $e$  is  $vc_i[1..n]$ .

when sending MSG( $m$ ) to  $p_j$  do
  (3) send MSG( $m, vc_i[1..n]$ ) to  $p_j$ .

when MSG( $m, vc$ ) is received from  $p_j$  do
  (4)  $vc_i[1..n] \leftarrow \max(vc_i[1..n], vc[1..n]).$ 
    
```

Fig. 7.18 Vector clock system for relevant events (code for process  $p_i$ )

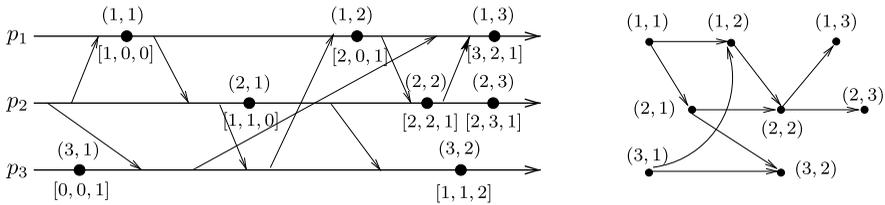


Fig. 7.19 From relevant events to Hasse diagram

The *immediate predecessor tracking* (IPT) problem consists in associating with each relevant event the set of relevant events that are its immediate predecessors. Moreover, this has been done on the fly and without adding control messages. The determination of the immediate predecessors consists in computing the transitive reduction (or Hasse diagram) of the partial order  $\widehat{R} = (R, \xrightarrow{r^e})$ . This reduction captures the essential causality of  $\widehat{R}$ .

The left of Fig. 7.19 represents the distributed computation of Fig. 7.17, in which only the relevant events are explicitly indicated, together with their vector dates and an identification pair (made up of a process identity plus a sequence number). The right of the figure shows the corresponding Hasse diagram.

**An Algorithm Solving the IPT Problem: Local Variables** The  $k$ th relevant event on process  $p_k$  is unambiguously identified by the pair  $(k, vc_k)$ , where  $vc_k$  is the value of  $vc_k[k]$  when  $p_k$  has produced this event. The aim of the algorithm is conse-

```

when producing a relevant internal event  $e$  do
(1)  $vc_i[i] \leftarrow vc_i[i] + 1$ ;
(2) Produce the relevant event  $e$ ; let  $e.imp = \{(k, vc_i[k]) \mid imp_i[k] = 1\}$ ;
(3)  $imp_i[i] \leftarrow 1$ ;
(4) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do  $imp_i[j] \leftarrow 0$  end for.

when sending MSG( $m$ ) to  $p_j$  do
(5) send  $MSG(m, vc_i[1..n], imp_i[1..n])$  to  $p_j$ .

when MSG( $m, vc, imp$ ) is received do
(6) for each  $k \in \{1, \dots, n\}$  do
(7)   case  $vc_i[k] < vc[k]$  then  $vc_i[k] \leftarrow vc[k]$ ;  $imp_i[k] \leftarrow imp[k]$ 
(8)      $vc_i[k] = vc[k]$  then  $imp_i[k] \leftarrow \min(imp_i[k], imp[k])$ 
(9)      $vc_i[k] > vc[k]$  then skip
(10)  end case
(11) end for.

```

**Fig. 7.20** Determination of the immediate predecessors (code for process  $p_i$ )

quently to associate with each relevant event  $e$  a set  $e.imp$  such that  $(k, vc[k]) \in e.imp$  if and only if the corresponding event is an immediate predecessor of  $e$ .

To that end, each process  $p_i$  manages the following local variables:

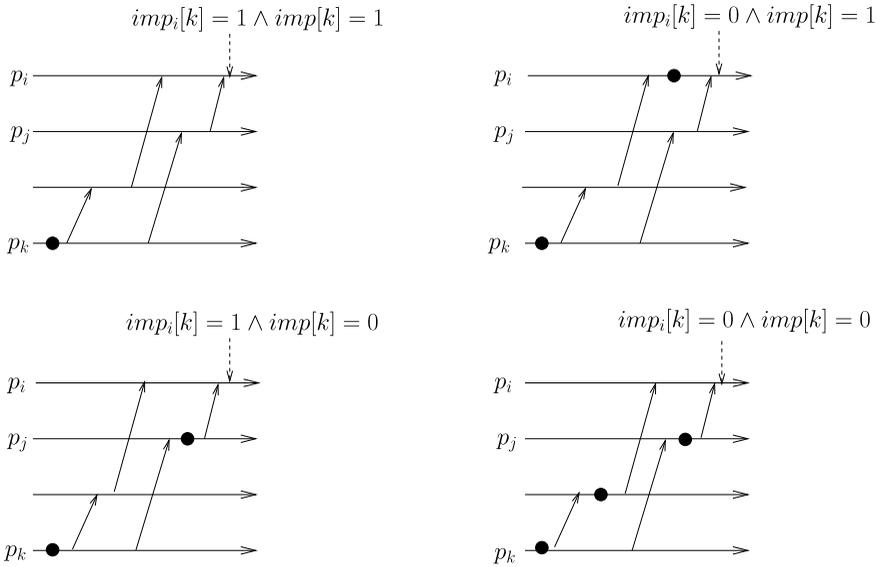
- $vc_i[1..n]$  is the local vector clock.
- $imp_i[1..n]$  is a vector initialized to  $[0, \dots, 0]$ . Each variable  $imp_i[j]$  contains 0 or 1. Its meaning is the following:  $imp_i[j] = 1$  means that the last relevant event produced by  $p_j$ , as known by  $p_i$ , is candidate to be an immediate predecessor of the next relevant event produced by  $p_i$ .

**An Algorithm Solving the IPT Problem: Process Behavior** The algorithm executed by each process  $p_i$  is a combined management of both the vectors  $vc_i$  and  $imp_i$ . It is described in Fig. 7.20.

When a process  $p_i$  produces a relevant event  $e$ , it increases its own vector clock (line 1). Then, it considers  $imp_i[1..n]$  to compute the immediate predecessors of  $e$  (line 2). According to the definition of each entry of  $imp_i$ , those are the events identified by the pairs  $(k, vc_i[k])$  such that  $imp_i[k] = 1$  (which indicates that the last relevant event produced by  $p_k$  and known by  $p_i$ , is still a candidate to be an immediate predecessor of  $e$ ).

Then, as it produced a new relevant event  $e$ ,  $p_i$  must reset its local array  $imp_i[1..n]$  (lines 3–4). It resets (a)  $imp_i[i]$  to 1 (because  $e$  is candidate to be an immediate predecessor of relevant events that will appear in its causal future) and (b) each  $imp_i[j]$  ( $j \neq i$ ) to 0 (because no event that will be produced in the future of  $e$  can have them as immediate predecessors).

When  $p_i$  sends a message, it attaches to this message all its local control information, namely,  $vc_i$  and  $imp_i$  (line 5). When it receives a message,  $p_i$  updates its local vector clock as in the basic algorithm so that the new value of  $vc_i$  is the component-wise maximum of  $vc$  and the previous value of  $vc_i$ .



**Fig. 7.21** Four possible cases when updating  $imp_i[k]$ , while  $vc_i[k] = vc[k]$

The update of the array  $imp_i$  depends on the value of each entry of  $vc_i$ .

- If  $vc_i[k] < vc[k]$ ,  $p_j$  (the sender of the message) has fresher information on  $p_k$  than  $p_i$ . Consequently,  $p_i$  adopts what is known by  $p_j$ , and sets  $imp_i[k]$  to  $imp[k]$  (line 7).
- If  $vc_i[k] = vc[k]$ , the last relevant event produced by  $p_k$  and known by  $p_i$  is the same as the one known by  $p_j$ . If this event is still candidate to be an immediate predecessor of the next event produced by  $p_i$  from both  $p_i$ 's point of view ( $(imp_i[k])$ ) and  $p_j$ 's point of view ( $(imp[k])$ ), then  $imp_i[k]$  remains equal to 1; otherwise,  $imp_i[k]$  is set to 0 (line 8). The four possible cases are depicted in Fig. 7.21.
- If  $vc_i[k] > vc[k]$ ,  $p_i$  knows more on  $p_k$  than  $p_j$ . Hence, it does not modify the value of  $imp_i[k]$  (line 9).

### 7.3 On the Size of Vector Clocks

This section first shows that vector time has an inherent price, namely, the size of vector clocks cannot be less than the number of processes. Then it introduces the notion of a *relevant event* and presents a general technique that allows us to reduce the number of vector entries that have to be transmitted in each message. Finally, it presents the notions of *approximation* of the causality relation and approximate vector clocks.

### 7.3.1 A Lower Bound on the Size of Vector Clocks

A vector clock has one entry per process, i.e.,  $n$  entries. It follows from the algorithm of Fig. 7.9 and Theorem 5 that vectors of size  $n$  are sufficient to capture causality and independence among events. Hence the question: Are vectors of size  $n$  necessary to capture causality and concurrency among the events produced by  $n$  asynchronous processes communicating by sending and receiving messages through a reliable asynchronous network?

This section shows that the answer to this question is “yes”. This means that there are distributed executions in which causality and concurrency cannot be captured by vector clocks of size smaller than  $n$ . The proof of this result, which is due to B. Charron-Bost (1991), consists in building such a specific execution and showing a contradiction if vector clocks of size smaller than  $n$  are used to capture causality and independence of events.

**The Basic Execution** Let us consider an execution of  $n$  processes in which each process  $p_i$ ,  $1 \leq i \leq n$ , executes a sending phase followed by a reception phase. There is a communication channel between any pair of distinct processes, and the communication pattern is based on a logical ring defined as follows on the process identities:  $i, i + 1, i + 2, \dots, n, 1, \dots, i - 1, i$ . The notations  $i + x$  and  $i - y$  are used to denote the  $x$ th successor and the  $y$ th predecessor of the identity  $i$  on the ring, respectively. More precisely, the behavior of each process  $p_i$  is as follows.

- A process  $p_i$  first sends, one after the other, a message to each process of the following “increasing” list of  $(n - 2)$  processes:  $p_{i+1}, p_{i+2}, \dots, p_n, p_1, \dots, p_{i-2}$ . The important points are the “increasing” order in the list of processes and the fact that  $p_i$  does not send a message to  $p_{i-1}$ .
- Then,  $p_i$  receives, one after the other, the  $(n - 2)$  messages sent to it, in the “decreasing” order on the identity of their senders, namely,  $p_i$  receives first the message from  $p_{i-1}$ , then the one from  $p_{i-2}, \dots, p_1, p_n, \dots, p_{i+2}$ . As before, the important points are the “decreasing” order in the list of processes and the fact that  $p_i$  does not receive a message from  $p_{i+1}$ .

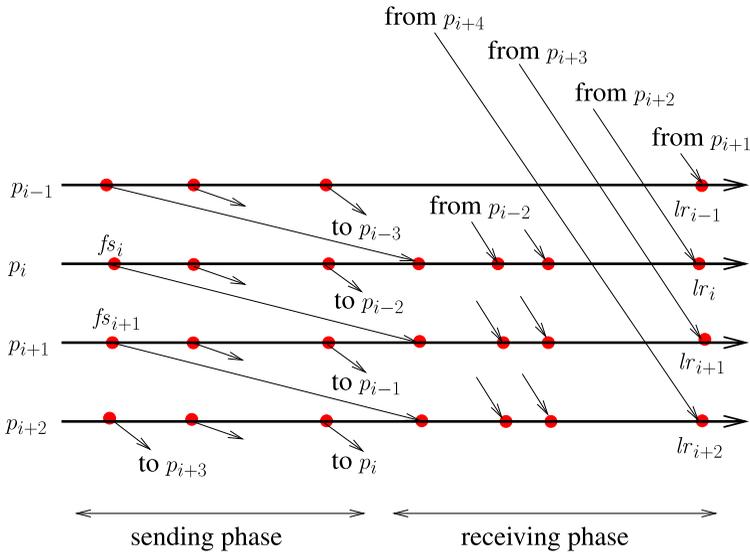
This communication pattern is described in Fig. 7.22. (A simpler figure with only three processes is described in Fig. 7.23.)

Considering a process  $p_i$ , let  $fs_i$  denote its first send event (which is the sending of a message to  $p_{i+1}$ ), and  $lr_i$  denote its last receive event (which is the reception of a message from  $p_{i+2}$ ).

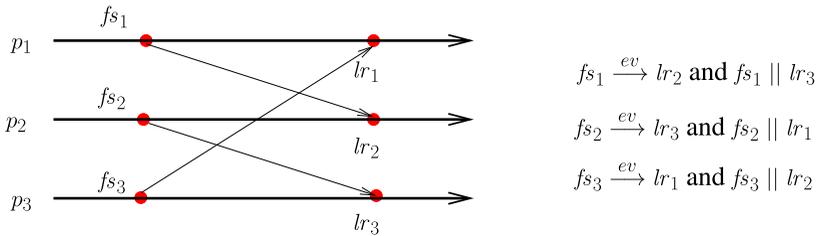
**Lemma 1**  $\forall i \in \{1, \dots, n\} : lr_i \parallel fs_{i+1}$ .

*Proof* As any process first sends messages before receiving any message, there is no causal chain involving more than one message. The lemma follows from this observation and the fact  $p_{i+1}$  does not send message to  $p_i$ .  $\square$

**Lemma 2**  $\forall i, j : 1 \leq i \neq j \leq n : fs_{i+1} \xrightarrow{ev} lr_j$ .



**Fig. 7.22** A specific communication pattern



**Fig. 7.23** Specific communication pattern with  $n = 3$  processes

*Proof* Let us first consider the case  $j = i + 1$ . We have then  $lr_j = lr_{i+1}$ . As both  $fs_{i+1}$  and  $lr_{i+1}$  are produced by  $p_{i+1}$ , and the send phase precedes the receive phase, the lemma follows.

Let us now consider the case  $j \neq i + 1$ . Due to the assumption, we have then  $j \notin \{i, i + 1\}$ . It then follows from the communication pattern that  $p_{i+1}$  sends a message to  $p_j$ . Let  $s(i, j)$  and  $r(i, j)$  be the corresponding send event and receive event, respectively. We have (a)  $fs_{i+1} = s(i, j)$  or  $fs_{i+1} \xrightarrow{ev} s(i, j)$ , (b)  $s(i, j) \xrightarrow{ev} r(i, j)$ , and (c)  $r(i, j) = lr_j$  or  $r(i, j) \xrightarrow{ev} lr_j$ . Combining the previous relations, we obtain  $fs_{i+1} \xrightarrow{ev} lr_j$ . □

**Theorem 7** *Let  $n \geq 3$  be the number of processes. To capture causality and independence of events produced by asynchronous computations, the dimension of vector time has to be at least  $n$ .*

*Proof* Let  $e.vc[1..k]$  ( $e.vc$ ) be the vector date associated with event  $e$ . Let us consider a process  $sp_i$ . It follows from Lemma 1 that, for each  $i \in \{1, \dots, n\}$ , we have  $lr_i \parallel fs_{i+1}$ , which means that the vector dates  $lr_i.vc$  and  $fs_{i+1}.vc$  have to be incomparable. If  $lr_i.vc[j] \geq fs_{i+1}.vc[j]$  for any  $j$ , we would have  $lr_i.vc \geq fs_{i+1}.vc$ , which is impossible since the events are independent. There exist consequently indexes  $x$  such that  $lr_i.vc[x] < fs_{i+1}.vc[x]$ . Let  $\ell(i)$  be one of these indexes. As this is true for any  $i \in \{1, \dots, n\}$ , we have defined a function

$$\ell : \{1, \dots, n\} \rightarrow \{1, \dots, k\}.$$

The rest of the proof shows that  $k \geq n$ . This is done by showing that the function  $\ell()$  is one-to-one.

Let us assume by contradiction that  $\ell()$  is such that there are two distinct indexes  $i$  and  $j$  such that  $\ell(i) = \ell(j) = x$ . Due to the definition of  $\ell()$ , we have  $lr_i.vc[x] < fs_{i+1}.vc[x]$  (A1), and  $lr_j.vc[x] < fs_{j+1}.vc[x]$  (A2). On another side, it follows from Lemma 2 that  $fs_{i+1} \xrightarrow{ev} lr_j$ . Since vector clocks are assumed to capture causality, we have  $fs_{i+1}.vc \leq lr_j.vc$  (A3). Combining (A1), (A2), and (A3), we obtain  $lr_i.vc[x] < fs_{i+1}.vc[x] \leq lr_j.vc[x] < fs_{j+1}.vc[x]$ , from which we conclude  $\neg(fs_{j+1} \xrightarrow{ev} lr_i)$ , which contradicts Lemma 2 and concludes the proof of the theorem.  $\square$

The reader can notice that the previous proof relies only on the fact that vector entries are comparable. Moreover, from a theoretical point of view, it does not require the value domain of each entry to be restricted to the set of integers (even if integers are easier to handle).

### 7.3.2 An Efficient Implementation of Vector Clocks

Theorem 7 shows that there are distributed executions in which the dimension of vector time must be  $n$  if one wants to capture causality and independence (concurrency) among the events generated by  $n$  processes.

Considering an abstraction level defined by relevant events (as defined in Sect. 7.2.6), this section presents an abstract condition, and two of its implementations, that allows each message to carry only a subset of the vector clock of its sender. Of course, in the worst case, this subset is the whole vector clock of the sender. This condition is on a “per message” basis. This means that the part of a vector clock carried by a message  $m$  depends on what is known by its sender about the values of all vector clocks, when it sends this message. This control information is consequently determined just before a message is sent.

Let us recall that communication events cannot be relevant events. Only a subset of the internal events are relevant.

**To Transmit or Not to Transmit Control Information: A Necessary and Sufficient Condition** Given a message  $m$  sent by a process  $p_i$  to a process  $p_j$ , let

$s(m, i, j)$  denote its send event and  $r(m, i, j)$  denote its receive event. Moreover, let  $pre(m, i, j)$  denote the last relevant event (if any) produced by  $p_j$  before  $r(m, i, j)$ . Moreover,  $e$  being any (relevant or not) event produced by a process  $p_x$ , let  $e.vc_x[k]$  be the value of  $vc_x[k]$  when  $p_x$  produces  $e$ . Let  $K(m, i, j, k)$  be the following predicate:

$$K(m, i, j, k) \stackrel{def}{=} s(m, i, j).vc_i[k] \leq pre(m, i, j).vc_j[k].$$

When true,  $K(m, i, j, k)$  means that  $vc_i[k]$  is smaller or equal to  $vc_j[k]$ , when  $p_i$  sends  $m$  to  $p_j$ ; consequently, it is not necessary for  $p_i$  to transmit the value of  $vc_i[k]$  to  $p_j$ . The next theorem captures the full power of this predicate.

**Theorem 8** *The predicate  $\neg K(m, i, j, k)$  is a necessary and sufficient condition for  $p_i$  to transmit the pair  $(k, vc_i[k])$  when it sends a message  $m$  to  $p_j$ .*

*Proof* Necessity. Let us assume that  $\neg K(m, i, j, k)$  is satisfied, i.e.,  $s(m, i, j).vc_i[k] > pre(m, i, j).vc_j[k]$ . According to the definition of vector clocks we must have  $s(m, i, j).vc_i[k] \leq r(m, i, j).vc_j[k]$ . If the pair  $(k, vc_i[k])$  is not attached to  $m$ ,  $p_j$  cannot update  $vc_j[k]$  to its correct value, which proves the necessity part.

Sufficiency. Let us consider a message  $m$  sent by  $p_i$  to  $p_j$  such that  $K(m, i, j, k)$  is satisfied. Hence, we have  $s(m, i, j).vc_i[k] \leq pre(m, i, j).vc_j[k]$ . As  $r(m, i, j).vc_j[k] = pre(m, i, j).vc_j[k]$ , we have  $s(m, i, j).vc_i[k] \leq r(m, i, j).vc_j[k]$ , from which it follows that, if the pair  $(k, vc_i[k])$  is attached to  $m$ , it is useless as  $vc_j[k]$  does not need to be updated.  $\square$

**From an Abstract Predicate to an Operational Predicate** Unfortunately,  $p_i$  cannot evaluate the predicates  $K(m, i, j, k)$ ,  $1 \leq k \leq n$ , before sending a message  $m$  to  $p_j$ . The aim is consequently to find an operational predicate  $K'(m, i, j, k)$ , such that  $K'(m, i, j, k) \Rightarrow K(m, i, j, k)$  (operational means that the corresponding predicate can be locally computed by  $p_i$ ). This means that if  $K'(m, i, j, k)$  indicates that it is useless to transmit the pair  $(k, vc_i[k])$ , then this decision does not entail an incorrect management of vector clocks.

Of course, always taking  $K'(m, i, j, k) = false$  works, but would force each message to carry the full vector clock. In order to attach to each message  $m$  as few pairs  $(k, vc_i[k])$  as possible, the aim is to find an operational predicate  $K'(m, i, j, k)$  that is the best possible approximation of  $K(m, i, j, k)$ , which can be locally computed.

To implement such a local predicate  $K'(m, i, j, k)$ , each process  $p_i$  manages an additional control data, namely a matrix denoted  $kprime_i[1..n, 1..n]$ . The value of each entry  $kprime_i[\ell, k]$  is 0 or 1, and its initial value is 1. It is managed in such a way that, to  $p_i$ 's knowledge,

$$(kprime_i[\ell, k] = 1) \Rightarrow (vc_i[k] \leq vc_\ell[k]).$$

Hence,

$$K'(m, i, j, k) \stackrel{def}{=} (s(m, i, j).kprime_i[j, k] = 1).$$

The algorithms that follow are due to J.-M. H elary, M. Raynal, G. Melideo, and R. Baldoni (2003).

```

when producing a relevant internal event  $e$  do
(1)  $vc_i[i] \leftarrow vc_i[i] + 1$ ; %  $e.vc_i[1..n]$  is the vector date of  $e$ 
(2) for each  $\ell \in \{1, \dots, n\} \setminus \{i\}$  do  $kprime_i[\ell, i] \leftarrow 0$  end for.

when sending MSG( $m$ ) to  $p_j$  do
(3) let  $vc\_set = \{(k, vck) \text{ such that } kprime_i[j, k] = 0\}$ ;
(4) sendMSG( $m, vc\_set$ ) to  $p_j$ .

when MSG( $m, vc\_set$ ) is received do
(5) for each  $(k, vck) \in vc\_set$  do
(6)   case  $vc_i[k] < vck$  then  $vc_i[k] \leftarrow vck$ ;
(7)     for each  $\ell \in \{1, \dots, n\} \setminus \{i, j, k\}$ 
(8)       do  $kprime_i[\ell, k] \leftarrow 0$ 
(9)     end for;
(10)     $kprime_i[j, k] \leftarrow 1$ 
(11)     $vc_i[k] = vck$  then  $kprime_i[j, k] \leftarrow 1$ 
(12)     $vc_i[k] > vck$  then skip
(13)  end case
(14) end for.

```

**Fig. 7.24** Management of  $vc_i[1..n]$  and  $kprime_i[1..n, 1..n]$  (code for process  $p_i$ ): Algorithm 1

**A First Algorithm** The algorithm of Fig. 7.24 describes the way each process  $p_i$  has to manage its vector clock  $vc_i[1..n]$  and its matrix  $kprime_i[1..n, 1..n]$  so that the previous relation is satisfied. Let us recall that  $vc_i[1..n]$  is initialized to  $[0, \dots, 0]$ , while  $kprime_i[1..n, 1..n]$  is initialized to  $[[1, \dots, 1], \dots, [1, \dots, 1]]$ .

When it produces a relevant event,  $p_i$  increases  $vc_i[i]$  (line 1) and resets to 0 (line 2) all entries of the column  $kprime_i[1..n, i]$  (except its own entry). This is because,  $p_i$  knows then that  $vc_i[i] > vc_\ell[i]$  for  $\ell \neq i$ .

When it sends a message to a process  $p_j$ ,  $p_i$  adds to it the set  $vc\_set$  containing the pairs  $(k, vck)$  such that, to its knowledge  $vc_j[k] < vc_i[k]$  (line 3). According to the definition of  $kprime_i[1..n, 1..n]$ , those are the pairs  $(k, -)$  such that  $kprime_i[j, k] = 0$ .

When a process  $p_i$  receives a message  $m$  with an associated set of pairs  $vc\_set$ , it considers separately each pair  $(k, vck) \in vc\_set$ . This is in order to preserve the property associated with  $K'(m, i, j, k)$  for each  $k$ , i.e.,  $(kprime_i[\ell, k] = 1) \Rightarrow (vc_i[k] \leq vc_\ell[k])$ . The behavior of  $p_i$  depends on the values of the pair  $(vc_i[k], vck)$ . More precisely, we have the following.

- If  $vc_i[k] < vck$ ,  $p_i$  is less informed on  $p_k$  than the sender  $p_j$  of the message. It consequently updates  $vc_i[k]$  to a more recent value (line 6), and sets (a)  $kprime_i[\ell, k]$  to 0 for  $\ell \neq i, j, k$  (this is because  $p_i$  does not know if  $vc_\ell[k] \geq vc_i[k]$ , lines 7–9), and (b)  $kprime_i[j, k]$  to 1 (because now it knows that  $vc_j[k] \geq vc_i[k]$ , line 10).
- If  $vc_i[k] = vck$ ,  $p_i$  sets accordingly  $kprime_i[j, k]$  to 1 (line 11).
- If  $vc_i[k] > vck$ ,  $p_i$  is more informed on  $p_k$  than the sender  $p_j$  of the message. It consequently does not modify the array  $kprime_i$  (line 12).

```

when producing a relevant internal event  $e$  do
(1)   $vc_i[i] \leftarrow vc_i[i] + 1$ ;  %  $e.vc_i[1..n]$  is the vector date of  $e$ 
(2)  for each  $\ell \in \{1, \dots, n\} \setminus \{i\}$  do  $kprime_i[\ell, i] \leftarrow 0$  end for.

when sending MSG( $m$ ) to  $p_j$  do
(3') let  $vc\_set = \{(k, vc_i[k], kprime_i[1..n, k]) \text{ such that } kprime_i[j, k] = 0\}$ ;
(4)  sendMSG( $m, vc\_set$ ) to  $p_j$ .

when MSG( $m, vc\_set$ ) is received do
(5') for each  $(k, vck, kpk[1..n]) \in vc\_set$  do
(6)   case  $vc_i[k] < vck$  then  $vc_i[k] \leftarrow vck$ ;
(7-10') for each  $\ell \in \{1, \dots, n\} \setminus \{i\}$ 
(7-10') do  $kprime_i[\ell, k] \leftarrow kpk[\ell]$ 
(7-10') end for;
(11')  $vc_i[k] = vck$  then for each  $\ell \in \{1, \dots, n\} \setminus \{i\}$ 
(11') do  $kprime_i[\ell, k] \leftarrow \max(kprime_i[\ell, k], kpk[\ell])$ 
(11') end for;
(12)  $vc_i[k] > vck$  then skip
(13) end case
(14) end for.

```

**Fig. 7.25** Management of  $vc_i[1..n]$  and  $kprime_i[1..n, 1..n]$  (code for process  $p_i$ ): Algorithm 2

**Remark** When considering a process  $p_i$ , the values in the column  $kprime_i[1..n, i]$  (but  $kprime_i[i, i]$ ) remain equal to 0 after its first update (line 2).

**The Case of FIFO Channels** If the channels are FIFO, when a process  $p_i$  sends a message  $m$  to another process  $p_j$ , the following line can be added after (line 3):

**for each**  $(k, -) \in vc\_set$  **do**  $kprime_i[j, k] \leftarrow 1$  **end for.**

These updates save future sendings of pairs to  $p_j$  as long as  $p_i$  does not produce a new relevant event (i.e., until  $vc_i[i]$  is modified). In particular, with this enhancement, a process  $p_i$  sends the pair  $(i, vc_i[i])$  to a process  $p_j$  only if, since its last relevant event, this sending is the first sending to  $p_j$ .

**A Modified Algorithm** When  $p_i$  sends a message to  $p_j$ , the more entries of  $kprime_i[j, k]$  are equal to 1, the fewer pairs  $(k, cvk)$  have to be transmitted (line 3). Hence, the idea is to design an algorithm that increases the number of entries of the local arrays  $kprime_i[1..n, 1..n]$  equal to 1, in order to decrease the size of  $vc\_set$ .

To that end, let us replace each pair  $(k, cvk)$  transmitted in  $vc\_set$  by a triplet  $(k, cvk, kprime_i[1..n, k])$ , and modify the statements associated with a message reception to benefit from the values of the binary vectors which have been received. The corresponding algorithm is described in Fig. 7.25.

When  $p_i$  receives a message sent by a process  $p_j$ , it can update  $kprime_i[\ell, k]$  to  $kprime_j[\ell, k]$  if  $p_j$  was more informed on  $p_k$  than  $p_i$  (case  $vc_i[k] < vc_j[k]$ ). In this case, for every  $\ell$ ,  $p_i$  updates it to the received value  $kprime_j[\ell, k]$  (lines 7–10'), which replaces lines 7 and 10 of the algorithm in Fig. 7.24). If  $p_i$  and  $p_j$  are such that  $vc_i[k] = vc_j[k]$ ,  $p_i$  updates each  $kprime_j[\ell, k]$  to  $\max(kprime_i[\ell, k], kprime_j[\ell, k])$  (line 11') which replaces line (line 11). This is

because, if  $vc_i[k] = vc_j[k]$  and  $kprime_j[\ell, k] = 1$ ,  $p_i$  knows that  $vc_\ell[k] \geq vc_i[k]$ , if it did not know it before. There is of course a tradeoff between the number of pairs whose sending is saved and the number of binary vectors which have now to be sent (see below).

**An Adaptive Communication Layer** Let  $s$  be the number of bits required to encode the sequence numbers of the relevant events of each process. Let P0 be the basic vector clock algorithm suited to relevant event (Fig. 7.18), P1 the algorithm in which messages carry pairs (Fig. 7.24), and P2 the algorithm in which messages carry triplets (Fig. 7.25).

As vectors have a canonical representation, the bit size of the control information associated with a message  $m$  is  $n \times s$  bits in P0. Given some point of an execution of algorithm P0, let  $m$  be a message sent by  $p_i$  to  $p_j$ . If, instead of the full vector clock,  $m$  had to piggyback the set  $vc\_set$  defined at line 3 of P1, the bit size of the control information associated with  $m$  would be  $\alpha_1 = set\_size(s + \log_2 n)$  bits. Similarly, it would be  $\alpha_2 = set\_size(n + s + \log_2 n)$  bits if  $m$  had to piggyback the set  $vc\_set$  defined at line 3' of P2.

Let us observe that, while  $\alpha_1 < \alpha_2$ , the algorithm P2 has the ability to make more entries of the matrices  $kprime_i[1..n, 1..n]$  equal to 1, which has a direct impact on the size of the control information associated with messages that will be sent in the future. Hence, choosing between P1 and P2 has to depend on some heuristic function based on the structure of the computation. It is nevertheless possible to define an adaptive communication layer which selects, dynamically for each message  $m$ , the “best” algorithm among P0, P1, and P2, to send and receive  $m$ .

These observations direct us to the adaptive algorithm described in Fig. 7.26, in which the statements associated with the sending and the reception of each message  $m$  are dynamically selected. If sending the full vector clock is cheaper, the sending and receiving rules of P0 are chosen (lines 2–3). Otherwise, a heuristic function is used to select either the rules of P1, or the rules of P2. This is expressed with the predicate *heuristic()* (line 4), which may depend on the communication graph or the application structure. A simple example of such a predicate is the following one:

**predicate *heuristic()* is**  $return((n - \sum_{1 \leq x \leq n} kprime_i[x, k]) > c)$  **end predicate.**

The value  $c$  is a threshold: if the number of triplets to transmit is not greater than  $c$ , then algorithm P2 is used, otherwise algorithm P1 is used.

Of course, plenty of possibilities are offered to the user. As a toy example, the messages sent to processes with an even identity could be sent and received with P1, while the other would be sent and received with P2. A more interesting strategy is the following. Let  $p_i$  and  $p_j$  be any pair of processes, where  $p_i$  is the sender and  $p_j$  the receiver. When P0 is not more efficient than P1 or P2 (line 2),  $p_i$  alternates in using P1 and P2 for its successive messages to  $p_j$ . Another strategy would consist to draw (at line 4) a random number in  $\{1, 2\}$ , which would be used to direct a process to use P1 or P2.

```

when sending MSG( $m$ ) to  $p_j$  do
(1) let  $set\_size = |\{k \text{ such that } kprime_i[j, k] = 0\}|$ ;
(2) if  $(s \times n) < set\_size(s + \log_2 n)$ 
(3)   then attach tag “0” to the message and use algorithm P0 to send it
(4)   else if (heuristic) then attach tag “1” to the message and use algorithm P1 to send it
(5)     else attach the “2” to the message and use algorithm P2 to send it
(6)   end if
(7) end if.

when MSG( $tag, m, vc\_set$ ) is received do
(8) According to the tag of the message, use the reception statements of P0, P1, or P2.

```

Fig. 7.26 An adaptive communication layer (code for process  $p_i$ )

### 7.3.3 $k$ -Restricted Vector Clock

**An Approximation of the Causal Precedence Relation** A relation  $\xrightarrow{app}$  on the set of events is an *approximation* of the causal precedence relation  $\xrightarrow{ev}$  if

$$\forall e_1, e_2 : (e_1 \xrightarrow{ev} e_2) \Rightarrow (e_1 \xrightarrow{app} e_2).$$

This means that the relation  $\xrightarrow{app}$  orders correctly any pair of causally related events. Hence, when  $e_1 \parallel e_2$ , we have either  $e_1 \xrightarrow{app} e_2$ , or  $e_2 \xrightarrow{app} e_1$ , or  $e_1$  and  $e_2$  are not ordered by the relation  $\xrightarrow{app}$ . The important point is that any approximation has to respect causal precedence. As a simple example, the order on events defined by linear time (see Sect. 7.1.1) is an approximation of causal precedence.

**$k$ -Restricted Vector Clocks** The notion of *restricted vector clocks* was introduced by F. Torres-Rojas and M. Ahamad (1999). It imposes a bound  $k$ ,  $1 \leq k \leq n$ , on the size of vector clocks (i.e., the dimension of vector time). The vector clock of each process  $p_i$  has only  $k$  entries, namely,  $vc_i[1..k]$ . These  $k$ -restricted vector clocks are managed by the algorithm described in Fig. 7.27 (which is the same as the vector clock algorithm described in Fig. 7.9, except for the way the vector entries are used).

Let  $f_k()$  be a deterministic surjective function from  $\{1, \dots, n\}$  (the set of process identities) to  $\{1, \dots, k\}$  (the set of vector clock entries). As a simple example,  $f_k(i)$  can be  $(i \bmod k) + 1$ . The function  $f_k()$  defines the set of processes that share the same entry of the restricted vector clocks.

Let  $e_1$  and  $e_2$  be two events timestamped  $\langle e_1.vc[1..k], i \rangle$  and  $\langle e_2.vc[1..k], j \rangle$ , respectively. The set of timestamps defines an approximation relation as follows:

- $((i = j) \wedge (e_1.vc[i] < e_2.vc[i])) \Rightarrow (e_1 \xrightarrow{ev} e_2).$
- $((i = j) \wedge (e_1.vc[i] > e_2.vc[i])) \Rightarrow (e_2 \xrightarrow{ev} e_1).$
- $(e_1.vc \parallel e_2.vc) \Rightarrow (e_1 \parallel e_2).$
- $((i \neq j) \wedge (e_1.vc < e_2.vc)) \Rightarrow (e_1 \xrightarrow{app} e_2).$  (We have then  $e_1 \xrightarrow{ev} e_2$  or  $e_1 \parallel e_2$ .)
- $((i \neq j) \wedge (e_1.vc > e_2.vc)) \Rightarrow (e_2 \xrightarrow{app} e_1).$  (We have then  $e_2 \xrightarrow{ev} e_1$  or  $e_1 \parallel e_2$ .)

```

when producing an internal event  $e$  do
(1)  $vc_i[f_k(i)] \leftarrow vc_i[f_k(i)] + 1;$ 
(2) Produce event  $e$ . % The date of  $e$  is  $vc_i[1..k]$ .

when sending  $MSG(m)$  to  $p_j$  do
(3)  $vc_i[f_k(i)] \leftarrow vc_i[f_k(i)] + 1;$  %  $vc_i[1..k]$  is the sending date of the message.
(4) send  $MSG(m, vc_i[1..k])$  to  $p_j$ .

when  $MSG(m, vc)$  is received from  $p_j$  do
(5)  $vc_i[f_k(i)] \leftarrow vc_i[f_k(i)] + 1;$ 
(6)  $vc_i[1..k] \leftarrow \max(vc_i[1..k], vc[1..k]).$  %  $vc_i[1..k]$  is the date of the receive event.

```

**Fig. 7.27** Implementation of a  $k$ -restricted vector clock system (code for process  $p_i$ )

When  $e_1 \xrightarrow{app} e_2$ , while  $e_1 \parallel e_2$ , we say that  $\xrightarrow{app}$  adds *false causality*.

If  $k = 1$ , we have then  $f_k(i) = 1$  for any  $i$ , and the  $k$ -restricted vector clock system boils down to linear time. If  $k = n$  and  $f_k(i) = i$ , the  $k$ -restricted vector clock system implements the vector time of dimension  $n$ . The approximate relation  $\xrightarrow{app}$  then boils down to the (exact) causal precedence relation  $\xrightarrow{ev}$ . When  $1 < k < n$ ,  $\xrightarrow{app}$  adds false causality. Experimental results have shown that for  $n = 100$  and  $1 < k \leq 5$ , the percentage of false causality (with respect to all the pairs of causally related events) added by  $\xrightarrow{app}$  remains smaller than 10%. This shows that approximations of the causality relation giving few false positives can be obtained with a  $k$ -restricted vector clock system working with a very small time dimension  $k$ . This makes  $k$ -restricted vector clock systems attractive when one has to simultaneously keep track of causality and cope with scaling problems.

## 7.4 Matrix Time

Linear time and vector time can be generalized to matrix time. While vector time captures “first-order” knowledge (a process knows that another process has issued some number of events), matrix time captures “second-order” knowledge (a process knows that another process knows that ...). This section introduces matrix time and illustrates its use with a simple example. Matrix time is due to M.J. Fischer and A. Michael (1982).

### 7.4.1 Matrix Clock: Definition and Algorithm

The matrix clock of a process  $p_i$  is a two-dimensional array, denoted  $mc_i[1..n, 1..n]$ , such that:

- $mc_i[i, i]$  counts the number of events produced by  $p_i$ .
  - $mc_i[i, k]$  counts the number of events produced by  $p_k$ , as known by  $p_i$ .
- It follows that  $mc_i[i, 1..n]$  is nothing else than the vector clock of  $p_i$ .

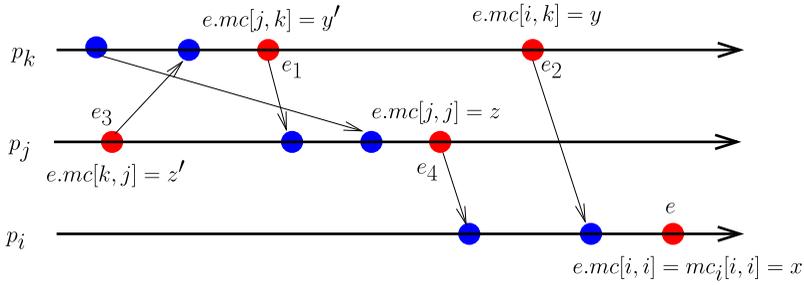


Fig. 7.28 Matrix time: an example

- $mc_i[j, k]$  counts the number of events produced by  $p_k$  and known by  $p_j$ , as known by  $p_i$ .

Hence, for any  $i, j, k$ ,  $mc_i[j, k] = x$  means “ $p_i$  knows that “ $p_j$  knows that “ $p_k$  has produced  $x$  events”””.

Development of matrix time is illustrated in Fig. 7.28. Let  $e$  be the last event produced by  $p_i$ . The values of  $mc_i$  considered are the values of that matrix clock just after  $e$  has been produced. It follows that, if  $e$  is the  $x$ th event produced by  $p_i$ , we have  $e.mc[i, i] = mc_i[i, i] = x$  (these numbers are indicated in the figure on top/bottom of the corresponding event).

- It is easy to see that  $e_2$  is the last event produced by  $p_k$  and known by  $p_i$  (according to the relation  $\xrightarrow{ev}$ ). If it is the  $y$ th event produced by  $p_k$ , we have  $e.mc[i, k] = mc_i[i, k] = mc_i[k, k] = y$ .
- Similarly, if  $e_4$  (last event produced by  $p_j$  and known by  $p_i$ ) is the  $z$ th event produced by  $p_j$ , we have  $e.mc[j, j] = mc_i[i, j] = mc_i[j, j] = y$ .
- The figure shows that  $e_1$  is the last event of  $p_k$  which is known by  $p_j$ , and this is known by  $p_i$ . Let this event be the  $y'$ th event produced by  $p_k$ . We consequently have  $e.mc[j, k] = mc_i[j, k] = y'$ .
- Finally,  $e_3$  is the last event produced by  $p_j$  and known by  $p_k$ , and this is known by  $p_i$  when it produces event  $e$ . Let this event be the  $z'$ th event produced by  $p_j$ . We have  $e.mc[k, j] = mc_i[k, j] = z'$ .

**Matrix Time Algorithm** The algorithm implementing matrix time is described in Fig. 7.29. A process  $p_i$  increases its event counter each time it produces an event (lines 1, 3, and 5). Each message piggybacks the current value of the matrix clock of its sender.

When a process  $p_i$  receives a message it updates its matrix clock as follows.

- The vector clock of  $p_i$ , namely  $mc_i[i, 1..n]$ , is first updated as in the basic vector clock algorithm (line 6). The value of the vector clock sent by  $p_j$  is in the vector  $mc[j, 1..n]$ .
- Then,  $p_i$  updates each entry of its matrix clock so that the matrix contains everything that can be known according to the relation  $\xrightarrow{ev}$  (line 7).

```

when producing an internal event  $e$  do
(1)  $mc_i[i, i] \leftarrow mc_i[i, i] + 1$ ;
(2) Produce event  $e$ . % The matrix date of  $e$  is  $mc_i[1..n][1..n]$ .

when sending MSG( $m$ ) to  $p_j$  do
(3)  $mc_i[i, i] \leftarrow mc_i[i, i] + 1$ ; %  $mc_i[1..n][1..n]$  is the sending date of the message.
(4) send MSG( $m, mc_i[1..n][1..n]$ ) to  $p_j$ .

when MSG( $m, mc$ ) is received from  $p_j$  do
(5)  $mc_i[i, i] \leftarrow mc_i[i, i] + 1$ ;
(6)  $mc_i[i, 1..n] \leftarrow \max(mc_i[i, 1..n], mc[j, 1..n])$ ;
(7) for each  $(k, \ell) \in [1..n, 1..n]$  do  $mc_i[k, \ell] \leftarrow \max(mc_i[k, \ell], mc[k, \ell])$  end for.
    %  $mc_i[1..n, 1..n]$  is the matrix date of the receive event.

```

**Fig. 7.29** Implementation of matrix time (code for process  $p_i$ )

**Property of Matrix Clocks** Let us consider the two following cases:

- Let  $\min(mc_i[1, k], \dots, mc_i[n, k]) = x$ . This means that, to  $p_i$ 's knowledge, all the processes know that  $p_k$  has produced  $x$  events. This can be used by  $p_i$  to forget events produced by  $p_k$  which are older than the  $(x + 1)$ th one.
- Let  $\min(mc_i[k, i], \dots, mc_i[n, i]) = x$ . This means that, to  $p_i$ 's knowledge, all the processes know that it has produced  $x$  events. This can be used by  $p_i$  to forget parts of its past older than its  $(x + 1)$ th event.

This means, in some applications, the two previous observations can be used by a process to discard old data, as soon as it knows that these data are known by all the processes.

### 7.4.2 A Variant of Matrix Time in Action: Discard Old Data

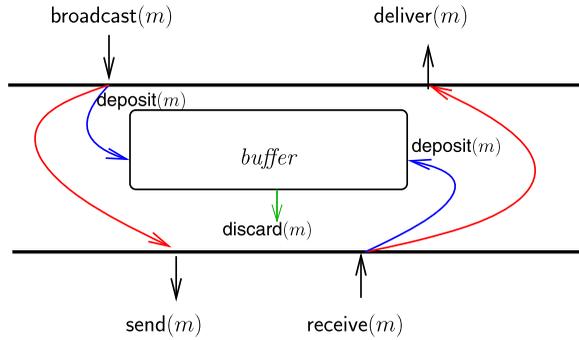
This example is related to the previous property of matrix clocks. It concerns the management of a message buffer.

**A Buffer Management Problem** A process can invoke two operations. The operation `broadcast( $m$ )` allows it to send a message to all processes, while the operation `deliver()` returns to it a message that has been broadcast.

For some reasons (fault-tolerance, archive recording, etc.) each process keeps in a private space (e.g., local disk) called *buffer*, all the messages it has broadcast or delivered. A condition imposed to a process which wants to destroy messages to free buffer space is that a message has to be known by all other processes before being destroyed. A structural view of this buffer management problem is described in Fig. 7.30.

**The Buffer Management Algorithm** A simple adaptation of matrix clocks solves this problem. As only broadcast messages are relevant, the local time of a process is

**Fig. 7.30** Discarding obsolete data: structural view (at a process  $p_i$ )



represented here by the number of messages it has broadcast. The definition of the content of the vector entry  $mc_i[j, k]$  has consequently to be interpreted as follows:  $mc_i[j, k] = x$  means that, to  $p_i$ 's knowledge, the  $x$  first messages broadcast by  $p_k$  have been delivered by  $p_j$ .

The corresponding algorithm (which assumes FIFO channels) is described in Fig. 7.31. The FIFO assumption simplifies the design of the algorithm. It guarantees that, if a process  $p_i$  delivers a message  $m$  broadcast by a process  $p_k$ , it has previously delivered all the messages broadcast by  $p_k$  before  $m$ .

When  $p_i$  broadcasts a message  $m$  it increases  $mc_i[i, i]$ , which is the sequence number of  $m$  (line 1). Then, after it has associated the timestamp  $\langle mc_i[i, 1..n], i \rangle$  with  $m$ ,  $p_i$  sends  $m$  and its timestamp to all the other process, and deposits the pair  $(m, \langle mc_i[i], i \rangle)$  in its local buffer (lines 2–3).

When it receives a message  $m$  with its timestamp  $\langle vc, j \rangle$ , a process  $p_i$  first deposits the pair  $(m, \langle vc[j], j \rangle)$ , in its buffer and delivers  $m$  to the local application process (line 4). Then,  $p_i$  increases  $mc_i[i, j]$  (it has delivered one more message from  $p_j$ ), and updates its local view of the vector clock of  $p_j$ , namely,  $mc_i[j, 1..n]$ , to  $vc$  (line 5). The fact that a direct assignment replaces the usual vector clock update

```

operation broadcast(m) is
(1)  $mc_i[i, i] \leftarrow mc_i[i, i] + 1;$ 
(2) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do  $\text{send}(m, \langle mc_i[i, 1..n], i \rangle)$  end for;
(3)  $\text{deposit}(m, \langle mc_i[i], i \rangle)$  into the buffer and deliver  $m$  locally.

when  $(m, \langle vc, j \rangle)$  is received do
(4)  $\text{deposit}(m, \langle vc[j], j \rangle)$  into the buffer;  $\text{deliver}(m)$  to the upper layer;
(5)  $mc_i[i, j] \leftarrow mc_i[i, j] + 1; mc_i[j, 1..n] \leftarrow vc[1..n].$ 

background task T is
(6) repeat forever
(7)   if  $(\exists(m, \langle sn, k \rangle) \in \text{buffer such that } sn \leq \min(mc_i[1, k], \dots, mc_i[n, k]))$ 
(8)     then  $(m, \langle sn, k \rangle)$  can be discarded from the buffer
(9)   end if
(10) end repeat.
    
```

**Fig. 7.31** A buffer management algorithm (code for process  $p_i$ )

statement  $mc_i[j, 1..n] \leftarrow \max(mc_i[j, 1..n], vc[1..n])$  is due to the FIFO property of the channels.

Finally, a local task  $T$  runs forever in the background. This task is devoted to the management of the buffer. If there is message  $m$  in the buffer whose tag  $\langle sn, k \rangle$  is such that  $sn \leq \min(mc_i[1, k], \dots, mc_i[n, k])$ ,  $p_i$  can conclude that all the processes have delivered this message. As a particular case,  $p_i$  knows that  $p_j$  has delivered  $m$  (which was broadcast by  $p_k$ ) because (a)  $mc_i[j, k] \geq sn$  and (b) the channels are FIFO ( $mc_i[j, k]$  being a local variable that  $p_i$  modifies only when it receives a message from  $p_j$ , it follows that  $p_i$  has previously received from  $p_j$  a message carrying a vector  $vc_j$  such that  $vc_j[k] \geq sn$ ).

*Remark* The reader can observe that this simple algorithm includes three notions of logical time: local time with sequence numbers, vector time which allows a process to know how many messages with a specific sender have been delivered by the other processes, and matrix time which encapsulates the previous notions of time. Matrix clocks are local, messages carry only vector clocks, and each buffer registers only sequence numbers.

## 7.5 Summary

This chapter has addressed the concept of *logical time* in distributed computations. It has introduced three types of logical time: linear (or scalar) time, vector time, and matrix time. An appropriate notion of virtual clock is associated with each of them. The meaning of these time notions has been investigated, and examples of their use have been presented. Basically, linear time is fundamental when one has to establish a total order on events that respects causal precedence, vector time captures exactly causal precedence, while matrix time provides processes with a “second order” knowledge on the progress of the whole set of processes.

## 7.6 Bibliographic Notes

- The notion of linear (scalar) time was introduced in 1978 by L. Lamport in [226]. This is a fundamental paper in which Lamport also introduced the *happened before* relation (causal precedence), which captures the essence of a distributed computation.
- The timestamp-based total order broadcast algorithm presented in Sect. 7.1.4 is a variant of algorithms described in [23, 226].
- The notions of vector time and vector clocks were introduced in 1988 (ten years after linear clocks) simultaneously and independently by C.J. Fidge [124], F. Mattern [250], and F. Schmuck [338]. The underlying theory is described in [125, 250, 340].

Preliminary intuitions of vector clocks appear in several papers (e.g., [53, 238, 290, 307, 338, 360]). Surveys on vector clock systems appear in [40, 149, 325].

The power and limitations of vector clocks are investigated in [126, 312].

- Tracking of causality in specific contexts is the subject of numerous papers. The case of synchronous systems is addressed in [6, 152], while the case of mobile distributed systems is addressed in [300].
- The algorithm which detects the first global state satisfying a conjunction of stable local predicates is due to M. Raynal [312].
- The proof of the lower bound showing that the size of vector clocks has to be at least  $n$  (the number of processes) if one wants to capture causality and independence of events is due to B. Charron-Bost [86].
- The notion of an efficient implementation of vector clocks was first introduced in [350]. The efficient algorithms implementing vector clocks presented in Sect. 7.3.2 are due J.-M. Hélary, M. Raynal, G. Melideo, and R. Baldoni [181]. An algorithm to reset vector clocks is presented in [394].
- The notion of a dependency vector was introduced in [129]. Such a vector is a weakened vector clock. This notion is generalized in [37] to the notion of  $k$ -dependency vector clock ( $k = n$  provides us with vector clocks).
- The notion of immediate predecessors of relevant events was introduced in [108, 198]. The corresponding tracking algorithm presented in Sect. 7.2.6 is due to E. Anceaume, J.-M. Hélary, and M. Raynal [17, 18].
- The notion of  $k$ -restricted vector clocks is due to F. Torres-Rojas and M. Ahamad [370], who also introduced a more general notion of *plausible* clock systems.
- Matrix time and matrix clocks were informally introduced in [127] and used in [334, 390] to discard obsolete data (see also [8]).
- Another notion of virtual time, suited to distributed simulation, is presented and studied in [199, 263].

## 7.7 Exercises and Problems

1. Let  $\mathcal{D}$  be the set of all the linear clock systems that are consistent. Given any  $D \in \mathcal{D}$ , let  $date_D(e)$  be the date associated by  $D$  with the event  $e$ . Let  $E$  be a distributed execution. As any  $D \in \mathcal{D}$  is consistent, we have  $(e_1 \xrightarrow{ev} e_2) \Rightarrow date_D(e_1) < date_D(e_2)$ , for any pair  $(e_1, e_2)$  of events of  $E$ .

Given a distributed execution  $E$ , show that, for any pair  $(e_1, e_2)$  of events of  $E$ , we have

- $(e_1 || e_2) \Rightarrow (\exists D' \in \mathcal{D} : date_{D'}(e_1) = date_{D'}(e_2))$ .
- $(e_1 || e_2) \Rightarrow$   
 $[\exists D', D'' \in \mathcal{D} : (date_{D'}(e_1) \leq date_{D'}(e_2)) \wedge (date_{D''}(e_1) \geq date_{D''}(e_2))]$ .

2. Considering the algorithm implementing the total order broadcast abstraction described in Fig. 7.7, let us replace the predicate of line 11,  $sd\_date \geq clock_i[i]$  by the following predicate  $\langle sd\_date, j \rangle < \langle clock_i[i], i \rangle$ .

```

when producing an internal event  $e$  do
(1)  $g1_i \leftarrow g1_i + 1$ ;
(2) Produce event  $e$ . % The date of  $e$  is the pair  $(g1_i, g2_i)$ .

when sending MSG( $m$ ) to  $p_j$  do
(3)  $g1_i \leftarrow g1_i + 1$ ; %  $(g1_i, g2_i)$  is the sending date of the message.
(4) send MSG( $m, g1_i$ ) to  $p_j$ .

when MSG( $m, g1$ ) is received from  $p_j$  do
(5)  $g1_i \leftarrow \max(g1_i, g1) + 1$ ;
(6)  $g2_i \leftarrow \max(g2_i, g1)$ . %  $(g1_i, g2_i)$  is the date of the receive event.

```

**Fig. 7.32** Yet another clock system (code for process  $p_i$ )

- Is the algorithm still correct? (Either a proof or a counterexample is needed.)
  - What is the impact of this new predicate on the increase of local clock values?
3. Show that the determination of the *first* global state satisfying a conjunction of stable local predicates cannot be done by repeated global state computations (which were introduced in Sect. 6.6 for the detection of stable properties).
  4. When considering  $k$ -restricted vector time, prove that the statements relating the timestamps with the relations  $\xrightarrow{app}$  are correct.
  5. Let us consider the dating system whose clock management is described in Fig. 7.32.

The clock of each process  $p_i$  is composed of two integers  $g1_i$  and  $g2_i$ , both initialized to 0. The date of an event is the current value of the pair  $(g1_i, g2_i)$ , and its timestamp is the pair  $\langle (g1_i, g2_i), i \rangle$ .

- What relation between  $g1_i$  and  $g2_i$  is kept invariant?
- What do  $g1_i$  and  $g2_i$  mean, respectively?
- Let  $e_1$  and  $e_2$  be two events timestamped  $\langle (g1, g2), i \rangle$  and  $\langle (h1, h2), i \rangle$ , respectively. What can be concluded on the causality relation linking the events  $e_1$  and  $e_2$  when  $(g1 > h2) \wedge (h1 > g2)$ ?

More generally, define from their timestamps  $\langle (g1, g2), i \rangle$  and  $\langle (h1, h2), i \rangle$  an approximation of the causal precedence relation on  $e_1$  and  $e_2$ .

- Compare this system with a  $k$ -restricted vector clock system,  $1 < k < n$ . Is one more powerful than the other from an approximation of the causal precedence relation point of view?

Solution in [370].

6. Design an algorithm that associates with each relevant event its immediate predecessors and uses the matrices  $kprime_i[1..n, 1..n]$  (introduced in Sect. 7.3.2) to reduce the bit size of the control information piggybacked by messages.

Solution in [18].

# Chapter 8

## Asynchronous Distributed Checkpointing

This chapter is devoted to checkpointing in asynchronous message-passing systems. It first presents the notions of local and global checkpoints and a theorem stating a necessary and sufficient condition for a set of local checkpoints to belong to the same consistent global checkpoint.

Then, the chapter considers two consistency conditions, which can be associated with a distributed computation enriched with local checkpoints (the corresponding execution is called a communication and checkpoint pattern). The first consistency condition (called z-cycle-freedom) ensures that any local checkpoint, which has been taken by a process, belongs to a consistent global checkpoint. The second consistency condition (called rollback-dependency trackability) is stronger. It states that a consistent global checkpoint can be associated on the fly with each local checkpoint (i.e., without additional communication).

The chapter discusses these consistency conditions and presents algorithms that, once superimposed on a distributed execution, ensure that the corresponding consistency condition is satisfied. It also presents a message logging algorithm suited to uncoordinated checkpointing.

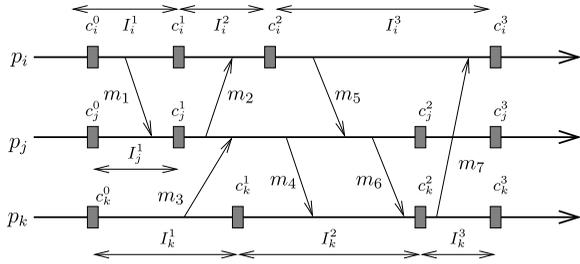
**Keywords** Causal path · Causal precedence · Communication-induced checkpointing · Interval (of events) · Local checkpoint · Forced local checkpoint · Global checkpoint · Hidden dependency · Recovery · Rollback-dependency trackability · Scalar clock · Spontaneous local checkpoint · Uncoordinated checkpoint · Useless checkpoint · Vector clock · Z-dependence · Zigzag cycle · Zigzag pattern · Zigzag path · Zigzag prevention

### 8.1 Definitions and Main Theorem

#### 8.1.1 Local and Global Checkpoints

It was shown in Chap. 6 that a distributed computation can be represented by a partial order  $\widehat{S} = (S, \xrightarrow{\sigma})$ , where  $S$  is the set of all the local states produced by the processes and  $\sigma$  is the causal precedence relation on these local states. This chapter has also defined the notion of a consistent global state, namely  $\Sigma = [\sigma_1, \dots, \sigma_n]$  is

**Fig. 8.1** A checkpoint and communication pattern (with intervals)



consistent if, for any pair of distinct local states  $\sigma_i$  and  $\sigma_j$ , we have  $\sigma_i \parallel \sigma_j$  (none of them depends on the other).

In many applications, we are not interested in all the local states, but only in a subset of them. Each process defines which of its local states are relevant. This is, for example, the case for the detection of properties on global states (a local checkpoint being then a local state satisfying some property), or for the definition of local states for consistent recovery. Such local states are called *local checkpoints*, and a set of  $n$  local checkpoints, one per process, is a global state called the *global checkpoint*.

A local checkpoint is denoted  $c_i^x$ , where  $i$  is the index (identity) of the corresponding process and  $x$  is its sequence number (among the local checkpoints of the same process). In the following,  $C$  will denote the set of all the local checkpoints.

An example of a distributed execution with local checkpoints is represented in Fig. 8.1, which is called a *checkpoint and communication pattern*. The local checkpoints are depicted with grey rectangular boxes. As they are irrelevant from a checkpointing point of view, the other local checkpoints are not represented. It is usually assumed that the initial local state and the final local state of every process are local checkpoints.

It is easy to see that the global checkpoint  $[c_i^1, c_j^1, c_k^1]$  is consistent, while the global checkpoint  $[c_i^2, c_j^2, c_k^1]$  is not consistent.

### 8.1.2 Z-Dependency, Zigzag Paths, and Z-Cycles

The sequence of internal and communication events occurring at a process  $p_i$  between two consecutive local checkpoints  $c_i^{x-1}$  and  $c_i^x$ ,  $x > 0$ , is called *interval*  $I_i^x$ . Some intervals are represented in Fig. 8.1.

**Zigzag Dependency Relation and Zigzag Path** These notions, which are due to R.H.B. Netzer and J. Xu (1995), are an extension of the relation  $\xrightarrow{\sigma}$  defined on local states. A relation on local checkpoints, called z-dependency, is defined as follows. A checkpoint  $c_i^x$  z-dependes on a local checkpoint  $c_j^y$  (denoted  $c_i^x \xrightarrow{z} c_j^y$ ), if:

- $c_i^x$  and  $c_j^y$  are in the same process ( $i = j$ ) and  $c_i^x$  appears before  $c_j^y$  ( $x < y$ ), or

- there a sequence of messages  $\langle m_1; \dots; m_q \rangle$ ,  $q \geq 1$ , such that (let us recall that  $s(m)$  and  $r(m)$  are the sending and receiving events associated with message  $m$ ):
  - $s(m_1) \in I_i^{x+1}$  (i.e.,  $m_1$  has been sent in the interval that starts just after  $c_i^x$ ),
  - $r(m_q) \in I_j^y$  (i.e.,  $m_q$  has been received in the interval finishing just before  $c_j^y$ ),
  - if  $q > 1$ ,  $\forall \ell : 1 \leq \ell < q$ , let  $I_k^t$  be the interval in which  $r(m_\ell)$  occurs (i.e.,  $m_\ell$  is received during  $I_k^t$ ). Then  $s(m_{\ell+1}) \in I_k^{t'}$  where  $t' \geq t$  (i.e.,  $m_{\ell+1}$  is sent by  $p_k$  in the interval in which  $m_\ell$  has been received, or in a later interval). Let us observe that it is possible that  $m_{\ell+1}$  has been sent before  $m_\ell$  is received.  
Such a sequence of messages is called a *zigzag path*.
- it exists  $c$  such that  $c_i^x \xrightarrow{zz} c$  and  $c \xrightarrow{zz} c_j^y$ .

As an example, due to the sequence of message  $\langle m_3; m_2 \rangle$ , we have  $c_k^0 \xrightarrow{zz} c_i^2$ . Similarly, due to sequence of messages  $\langle m_5; m_4 \rangle$ , (or the sequence  $\langle m_5; m_6 \rangle$ ), we have  $c_i^2 \xrightarrow{zz} c_k^2$ . Let us observe that we have  $c_i^2 \xrightarrow{\sigma} c_k^2$ , while we do not have  $c_k^0 \xrightarrow{\sigma} c_i^2$ .

A local checkpoint  $c_i^x$  belongs to a zigzag path  $\langle m_1; \dots; m_q \rangle$  if this path contains two consecutive messages  $m_\ell$  and  $m_{\ell+1}$  such that  $m_\ell$  is received by  $p_i$  before  $c_i^x$  and  $m_{\ell+1}$  is sent by  $p_i$  after  $c_i^x$ . As an example, in the figure, the local checkpoint  $c_i^2$  belongs to the zigzag path  $\langle m_3; m_2; m_5 \rangle$ .

A local checkpoint  $c$  belongs to a zigzag cycle if  $c \xrightarrow{zz} c$ . As an example, in Fig. 8.1, we have  $c_k^2 \xrightarrow{zz} c_k^2$ . This is because the sequence  $\langle m_7; m_5; m_6 \rangle$  is a zigzag cycle in which the reception of  $m_7$  and the sending of  $m_5$  (both by  $p_i$ ) belong to the same interval  $I_i^3$ .

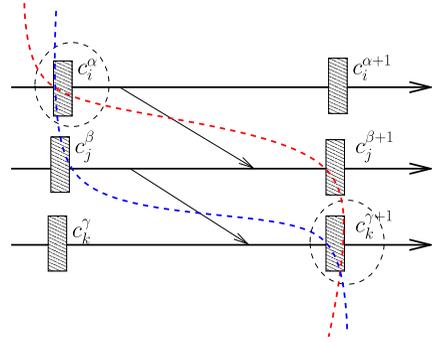
**Zigzag Pattern** Let us consider a zigzag path  $\langle m_1; \dots; m_q \rangle$ . Two consecutive messages  $m_\ell$  and  $m_{\ell+1}$  define a *zigzag pattern* (also denoted *zz-pattern*) if the reception of  $m_\ell$  and the sending of  $m_{\ell+1}$  occurs in the same interval, and the sending of  $m_{\ell+1}$  occur before the reception of  $m_\ell$ .

As we have just seen, the sequence of messages  $\langle m_7; m_5 \rangle$  is a zigzag pattern. Other zigzag patterns are the sequences of messages  $\langle m_3; m_2 \rangle$  and  $\langle m_5; m_4 \rangle$ .

It is easy to notice that zigzag patterns are what make zigzag paths different from causal paths. Every causal path is a zigzag path, but a zigzag path that includes a zigzag pattern is not necessarily a causal path. Hence,  $c_1$  and  $c_2$  being any two distinct local checkpoints, we always have

$$(c_1 \xrightarrow{\sigma} c_2) \Rightarrow (c_1 \xrightarrow{zz} c_2),$$

while it is possible that  $(c_1 \xrightarrow{zz} c_2) \wedge \neg(c_1 \xrightarrow{\sigma} c_2)$ . As an example, in Fig. 8.1, we have  $(c_k^0 \xrightarrow{zz} c_i^2) \wedge \neg(c_k^0 \xrightarrow{\sigma} c_i^2)$ . In that sense, the Z-dependency relation  $\xrightarrow{zz}$  is weaker (i.e., includes more pairs) than the causal precedence relations on events  $\xrightarrow{\sigma}$ .

**Fig. 8.2** A zigzag pattern

### 8.1.3 The Main Theorem

A fundamental question associated with local and global checkpoints is the following one: Given a checkpoint and communication pattern, and a set  $LC$  of local checkpoints (with at most one local checkpoint per process), is it possible to extend  $LC$  (with local checkpoints of the missing processes, if any) in order to obtain a consistent global checkpoint?

**What Is the Difficulty** To illustrate this question let us again consider Fig. 8.1.

- Let us first consider  $LC1 = \{c_j^1\}$ . Is it possible to extend this set with a local checkpoint from  $p_i$ , and another one from  $p_k$ , such that, once pieced together, these three local checkpoints define a consistent global checkpoint? The figure shows that the answer is “yes”: the global checkpoint  $[c_i^1, c_j^1, c_k^0]$  answers the question (as does also the global checkpoint  $[c_i^1, c_j^1, c_k^1]$ ).
- Let us now consider the question with  $LC2 = \{c_i^2, c_k^0\}$ . It is easy to see that neither  $c_j^t$  with  $t \leq 1$ , nor  $c_j^t$  with  $t \geq 2$ , can be added to  $LC2$  to obtain a consistent global checkpoint. Hence, the answer is “no” for  $LC2$ .
- Let us finally consider the case  $LC3 = \{c_k^2\}$ . The figure shows that neither  $c_i^2$  nor  $c_i^3$  can be consistent with  $c_k^2$  (this is due to the causality precedence relating these local checkpoints to  $c_k^2$ ). Hence, the answer is “no” for  $LC3$ .

If there is a causal path relating two local checkpoints, we know (from Chap. 6) that they cannot belong to the same consistent global checkpoint. Hence, to better appreciate the difficulty of the problem (and the following theorem), let us consider Fig. 8.2. Let  $LC = \{c_i^\alpha, c_k^{\gamma+1}\}$ . It is easy to see that adding either  $c_j^\beta$  or  $c_j^{\beta+1}$  to  $LC$  does not work. As depicted, each cut line (dotted line in the figure) defines an inconsistent global checkpoint. Consequently, there is no mean to extend  $LC$  with a local checkpoint of  $p_j$  in order to obtain a consistent global checkpoint. This observation shows that, absence of causal dependences among local checkpoints is a necessary condition to have a consistent global checkpoint, but is not a sufficient condition.

This example shows that, in addition to causal precedence, there are hidden dependences among local checkpoints that prevent them to belong to the same consistent global checkpoint. These hidden dependences are the ones created by zigzag patterns. These patterns, together with causal precedence, are formally captured by the relation  $\xrightarrow{zz}$ . The following theorem, which characterizes the set of local checkpoints that can be extended to form a consistent global checkpoint, is due to R.H.B. Netzer and J. Xu (1995).

**Theorem 9** *Let  $LC$  be a set of local checkpoints.  $LC$  can be extended to a consistent global checkpoint if and only if  $\forall c_1, c_2 \in LC$ , we have  $\neg(c_1 \xrightarrow{zz} c_2)$ .*

*Proof* Proof of the “if” part. Let us assume that,  $\forall c_1, c_2 \in LC$ , we have  $\neg(c_1 \xrightarrow{zz} c_2)$ . The proof consists in (a) first constructing a global checkpoint  $\Sigma$  that includes the local checkpoints of  $LC$ , plus one local checkpoint per process not in  $LC$ , and (b) then proving that  $\Sigma$  is consistent.

The global checkpoint  $\Sigma$  is built as follows. Let  $p_j$  be a process that has no local checkpoint in  $LC$ .

- Case 1:  $p_j$  has a local checkpoint  $c(j)$  with a zigzag path to a local checkpoint in  $LC$  (i.e.,  $\exists c \in LC : c(j) \xrightarrow{zz} c$ ).

The local checkpoint  $c'(j)$  is added to  $\Sigma$ , where  $c'(j)$  is the first checkpoint of  $p_j$  that has no zigzag path to a local checkpoint in  $LC$ . Such a local checkpoint  $c'(j)$  exists because it is assumed that the last local state of a process is its last local checkpoint (and such a local checkpoint cannot be the starting point of a zigzag path to any other local checkpoint).

- Case 2:  $p_j$  has no local checkpoint  $c(j)$  with a zigzag path to a local checkpoint in  $LC$  (i.e.,  $\nexists c \in LC : c(j) \xrightarrow{zz} c$ ).

The local checkpoint  $c'(j)$  is added to  $\Sigma$ , where  $c'(j)$  is the first checkpoint of  $p_j$ . Such a local checkpoint  $c'(j)$  exists because it is assumed that the first local state of a process is its first local checkpoint.

Let us recall that the definition of the consistency of a global checkpoint (global state)  $\Sigma$  is based on the relation  $\xrightarrow{\sigma}$ . More precisely, to prove that  $\Sigma$  is consistent, we have to show that, for any  $c_1, c_2 \in \Sigma$ , we cannot have  $c_1 \xrightarrow{\sigma} c_2$ . Let  $\overline{LC}$  be the set of local checkpoints which are in  $\Sigma$  and not in  $LC$ . The proof is by contradiction: It assumes that there are  $c_1, c_2 \in \Sigma$  such that  $c_1 \xrightarrow{\sigma} c_2$  and shows that this assumption is impossible. According to the fact that  $c_1$  and  $c_2$  belong or not to  $LC$  or  $\overline{LC}$  there are four possible cases to analyze:

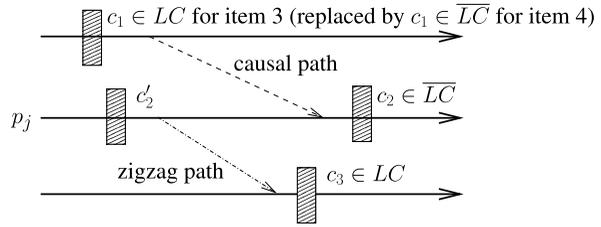
1.  $c_1, c_2 \in LC$  and  $c_1 \xrightarrow{\sigma} c_2$ .

This case is clearly impossible, as  $\xrightarrow{\sigma} \subseteq \xrightarrow{zz}$ ,  $c_1 \xrightarrow{\sigma} c_2$  contradicts  $\neg(c_1 \xrightarrow{zz} c_2)$ .

2.  $c_1 \in \overline{LC}$ ,  $c_2 \in LC$ , and  $c_1 \xrightarrow{\sigma} c_2$ .

This implies that  $c_1 \xrightarrow{zz} c_2$ , which is impossible because it contradicts the way the local checkpoint  $c_1 \in \overline{LC}$  is defined (according to the definition of the

**Fig. 8.3** Proof of Theorem 9: a zigzag path joining two local checkpoints of  $LC$



local checkpoints of  $\overline{LC}$ , none of them can be the starting local checkpoint of a zigzag path to a local checkpoint in  $LC$ ).

3.  $c_1 \in LC$ ,  $c_2 \in \overline{LC}$ , and  $c_1 \xrightarrow{\sigma} c_2$ .

In this case,  $c_2$  cannot be an initial local checkpoint. This is because no local checkpoint can causally precede (relation  $\xrightarrow{\sigma}$ ) an initial local checkpoint. Hence, as  $c_2 \in \overline{LC}$ ,  $c_2$  is a local checkpoint defined by Case 1, i.e.,  $c_2$  is the first local checkpoint (of some process  $p_j$ ) that has no zigzag path to a local checkpoint in  $LC$ .

Let  $c'_2$  the local checkpoint of  $p_j$  immediately preceding  $c_2$ . This local checkpoint  $c'_2$  must have a zigzag path to a local checkpoint  $c_3 \in LC$  (otherwise,  $c_2$  would not be the first local checkpoint of  $p_j$  that has no zigzag path to a local checkpoint in  $LC$ ).

This zigzag path from  $c'_2$  to  $c_3$ , plus the messages giving rise to  $c_1 \xrightarrow{\sigma} c_2$ , establish a zigzag path from  $c_1$  to  $c_3$  (Fig. 8.3). But this contradicts the fact that we have (assumption)  $\neg(c_1 \xrightarrow{zz} c_3)$  for any pair  $c_1, c_3 \in LC$ . This contradiction concludes the proof of the case.

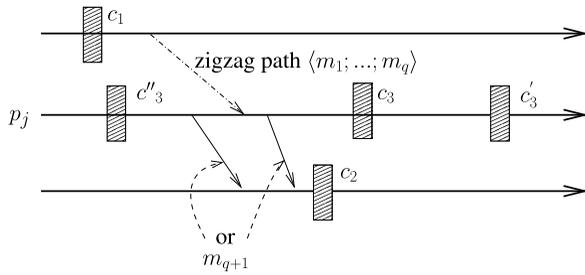
4.  $c_1, c_2 \in \overline{LC}$  and  $c_1 \xrightarrow{\sigma} c_2$ .

It follows from the argument of the previous item that there is a zigzag path from  $c_1$  to a local checkpoint in  $LC$  (see the figure where  $c_1 \in LC$  is replaced by  $c_1 \in \overline{LC}$ ). But this contradicts the definition of  $c_1$  (which is the first local checkpoint—of some process  $p_j$ —with no zigzag path to any local checkpoints of  $LC$ ).

Proof of the “only if” part. We have to show that, if there are two (not necessarily distinct) local checkpoints  $c_1, c_2 \in LC$  such that  $c_1 \xrightarrow{zz} c_2$ , there is no consistent global checkpoint  $\Sigma$  including  $c_1$  and  $c_2$ . Hence, let us assume that  $c_1 \xrightarrow{zz} c_2$ . If  $c_1$  and  $c_2$  are from the same process, the proof follows directly from the definition of global checkpoint consistency. Hence, let us assume that  $c_1$  and  $c_2$  are from different processes. There is consequently a zigzag path  $\langle m_1; \dots; m_q \rangle$  starting after  $c_1$  and finishing before  $c_2$ . The proof is by induction on the number of messages in this path.

- Base case:  $q = 1$ . If  $c_1 \xrightarrow{zz} c_2$  and the zigzag path contains a single message, we necessarily have  $c_1 \xrightarrow{\sigma} c_2$  (a zigzag path made up of a single message is necessarily a causal path), and the proof follows.

**Fig. 8.4** Proof of Theorem 9: a zigzag path joining two local checkpoints



- Induction case:  $q > 1$ . Let us assume that, if a zigzag path made up of at most  $q$  messages joins two local checkpoints, these local checkpoints cannot belong to the same consistent global checkpoint. We have to prove that if two local checkpoints  $c_1$  and  $c_2$  are connected by a zigzag path of  $q + 1$  messages  $\langle m_1; \dots; m_q; m_{q+1} \rangle$ , they cannot belong to the same consistent global checkpoint.

The proof is by contradiction. Let us assume that there is a consistent global checkpoint  $\Sigma$  including  $c_1$  and  $c_2$  such that these two local checkpoints are connected by a zigzag path of  $q + 1$  messages (if  $c_1 = c_2$ , the zigzag path is a zigzag cycle).

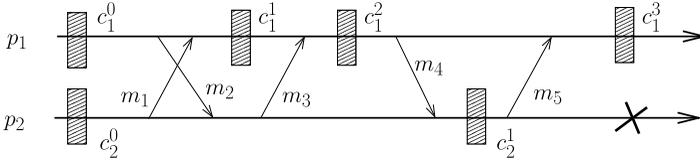
Let  $c_3$  be the local checkpoint preceding the reception of the message  $m_q$ , and  $p_j$  be the corresponding receiving process. This means that  $c_1$  is connected to  $c_3$  by the zigzag path  $\langle m_1; \dots; m_q \rangle$  (Fig. 8.4). It follows from the induction assumption that  $c_1$  and  $c_3$  cannot belong to the same consistent global checkpoint. More generally, given any  $c'_3$  that appears after  $c_3$  on the same process  $p_j$ ,  $c_1$  and  $c'_3$  cannot belong to the same consistent global checkpoint.

It follows from this observation that, for both  $c_1$  and  $c_2$  to belong to the same consistent global checkpoint  $\Sigma$ , this global checkpoint must include a local checkpoint of  $p_j$  that precedes  $c_3$ . But, due the definition of “zigzag path”, the message  $m_{q+1}$  has necessarily been sent after the local checkpoint of  $p_j$  that immediately precedes  $c_3$ . This local checkpoint is denoted  $c''_3$  in Fig. 8.4. It follows that any local checkpoint of  $p_j$  that causally precedes  $c_3$ , causally precedes  $c_2$ .

Thus,  $c_2$  cannot be combined with any local checkpoint preceding  $c_3$  to form a consistent global checkpoint. The fact that no local checkpoint of  $p_j$  can be combined with both  $c_1$  and  $c_2$  to form a consistent global checkpoint concludes the proof of the theorem. □

It follows that a global checkpoint is consistent if and only if there is no z-dependence among its local checkpoints.

**Useless Checkpoint** A local checkpoint is *useless* if it cannot belong to any consistent global checkpoint. It follows from the previous theorem that a local checkpoint  $c$  is useless if and only if  $c \xrightarrow{zz} c$  (i.e., it belongs to a cycle of the z-precedence relation). Said the opposite way, a local checkpoint belongs to a consistent global checkpoint if and only if it does not belong to a z-cycle.



**Fig. 8.5** Domino effect (in a system of two processes)

As an example, let us consider again Fig. 8.1. The local checkpoint  $c_k^2$  is useless because it belongs to the zigzag path  $\langle m_7; m_5; m_6 \rangle$  which includes the zigzag pattern  $\langle m_7; m_5 \rangle$ .

## 8.2 Consistent Checkpointing Abstractions

Let us recall that  $C$  denotes the set of all the local checkpoints defined during a distributed computation  $\widehat{S} = (S, \xrightarrow{\sigma})$ . The pair  $(C, \xrightarrow{zz})$  constitutes a checkpointing abstraction of this distributed computation. Hence the fundamental question of the asynchronous checkpointing problem: Is  $(C, \xrightarrow{zz})$  a consistent checkpointing abstraction of the distributed computation  $\widehat{S}$ ?

Two different consistency conditions can be envisaged to answer this question.

### 8.2.1 Z-Cycle-Freedom

**Definition** An abstraction  $(C, \xrightarrow{zz})$  is *z-cycle-free* if none of its local checkpoints belongs to a z-cycle:  $\forall c \in C : \neg(c \xrightarrow{zz} c)$ .

This means that z-cycle-freedom guarantees that no local checkpoint is useless, or equivalently each local checkpoint belongs to at least one consistent global checkpoint.

**Domino Effect** The domino effect is a phenomenon that may occur when looking for a consistent global checkpoint. As a simple example, let us consider processes that define local checkpoints in order to be able to recover after a local failure. In order that the computation be correct, the restart of a process from one of its previous checkpoints may entail the restart of other processes (possibly all) from one of their previous checkpoints. This is depicted in Fig. 8.5. After its local checkpoint  $c_2^1$ , process  $p_2$  experiences a failure and has to restart from one of its previous local state  $c_2$ , which is such that there is a local state  $c_1$  of process  $p_1$  and the global checkpoint  $[c_1, c_2]$  is consistent. Such a global checkpoint cannot be  $[c_1^3, c_2^1]$  because it is not consistent. Neither can it be  $[c_1^2, c_2^1]$  for the same reason. The reader can check that, in this example, the only consistent global checkpoint is the initial

one, namely  $[c_1^0, c_2^0]$ . This backtracking to find a consistent global checkpoint is called the *domino effect*.

It is easy to see that, if the local checkpoints satisfy the z-cycle-freedom property, no domino effect can occur. In the example, there would be a local checkpoint  $c_1$  on  $p_1$ , such that  $\Sigma = [c_1, c_2^1]$  would be consistent, and the computation could be restarted from this consistent global checkpoint. Moreover, this consistent global checkpoint  $\Sigma$  has the property to be “as fresh as possible” in the sense that, any other consistent global checkpoint  $\Sigma'$  from which the computation could be restarted is such that  $\Sigma' \xrightarrow{\Sigma} \Sigma$  (where  $\xrightarrow{\Sigma}$  is the reachability relation on global states defined in Chap. 6).

## 8.2.2 Rollback-Dependency Trackability

**Definition** Rollback-dependency trackability (RDT) is a consistency condition stronger than the absence of z-cycles. It was introduced by Y.-M. Wang (1997). An abstraction  $(C, \xrightarrow{zz})$  can be z-cycle-free, while having pairs of local checkpoints which are related only by zigzag paths which are not causal paths (hence, each of these zigzag paths includes a zigzag pattern). This means that,  $c_1$  and  $c_2$  being two such local checkpoints, we have  $c_1 \xrightarrow{zz} c_2$  and  $\neg(c_1 \xrightarrow{\sigma} c_2)$ .

RDT states that any hidden dependency (created by a zigzag pattern) is “doubled” by a causal path. Formally, an abstraction  $(C, \xrightarrow{zz})$  satisfies the RDT consistency condition if

$$\forall c_1, c_2 \in C : (c_1 \xrightarrow{zz} c_2) \Rightarrow (c_1 \xrightarrow{\sigma} c_2).$$

It follows that a checkpoint-based abstraction  $(C, \xrightarrow{zz})$  of a distributed execution satisfies the RDT consistency condition if it is such that, when considering only the local states which are local checkpoints, we have  $\xrightarrow{zz} \equiv \xrightarrow{\sigma}$ . This does not mean that there are no zigzag patterns. It means that, if there is a zigzag pattern on a zigzag path connecting two local checkpoints, there is also a causal path connecting these local checkpoints.

An example of such a “doubling” is given in Fig. 8.1. We have  $c_i^2 \xrightarrow{zz} c_k^2$ , and the zigzag pattern  $[m_5; m_4]$  is “doubled” by the causal path  $[m_5; m_6]$ . Differently, while we have  $c_k^0 \xrightarrow{zz} c_i^2$ , the zigzag pattern  $[m_3; m_2]$  is not “doubled” by a causal path.

**Why RDT Is Important** When a checkpoint-based abstraction of a distributed execution satisfies the RDT consistency condition, we know that (if any) hidden dependencies among local checkpoints are doubled by causal paths. It follows that, in such a context, the statement of Theorem 9 simplifies and becomes: Any set  $LC$  of local checkpoints which are not pairwise causally related can be extended to form a consistent global checkpoint.

As causal dependences can be easily tracked by vector clocks, the RDT consistency condition can benefit and simplify the design of many checkpoint-based applications such as the detection of global properties, or the definition of global checkpoints for failure recovery. A noteworthy property of RDT is the following one: it allows us to associate with each local checkpoint  $c$  (on the fly and without additional cooperation among processes) the first consistent global checkpoint including  $c$ . (The notion of “first” considered here is with respect to the sublattice of consistent global states obtained by eliminating the global states which are not global checkpoints, see Chap. 6.) This property is particularly interesting when one has to track software errors or recover after the detection of a deadlock.

### 8.2.3 On Distributed Checkpointing Algorithms

**Spontaneous vs. Forced Local Checkpoints** It is assumed that, for application-dependent reasons, each process  $p_i$  defines some of its local states as local checkpoints. Such checkpoints are called *spontaneous* checkpoints. However, there is no guarantee that the resulting abstraction  $(C, \xrightarrow{zz})$  satisfies any of the previous consistency conditions. To that end, checkpointing algorithms have to be superimposed to the distributed execution in order additional local checkpoints be automatically defined. These are called *forced* checkpoints.

When a local state of a process  $p_i$  is defined as a local checkpoint, it is said that “ $p_i$  takes a local checkpoint”. According to the application, the corresponding local checkpoint can be kept locally in volatile memory, saved in a local stable storage, or sent to a predetermined process.

**Classes of Checkpointing Algorithms** According to the underlying coordination mechanism they use, two classes of distributed checkpointing algorithms can be distinguished.

- In addition to the application messages that they can overload with control information, the algorithms of the first class use specific control messages (which do not belong to the application). Hence, this class of algorithms is based on *explicit synchronization*. Such an algorithm (suited to FIFO channels) was presented in Sect. 6.6. In the literature, these algorithms are called *coordinated checkpointing algorithms*.
- The second family of checkpointing algorithms is based on *implicit synchronization*. The processes can only add control information to application messages (they are not allowed to add control messages). This control information is used by the receiver process to decide if this message reception has to entail a forced local checkpoint. In the literature, these algorithms are called *communication-induced checkpointing algorithms*.

### 8.3 Checkpointing Algorithms Ensuring Z-Cycle Prevention

This section presents checkpointing algorithms that (a) ensure the z-cycle-freedom property, and (b) associate with each local checkpoint a consistent global checkpoint to which it belongs. It is important to notice that (b) consists in associating a global identity with each local checkpoint, namely the identity of the corresponding consistent global checkpoint

#### 8.3.1 An Operational Characterization of Z-Cycle-Freedom

Let us associate with each local checkpoint  $c$  a integer denoted  $c.date$  (a logical date from an operational point of view). Let  $(C, \xrightarrow{zz})$  be a checkpointing abstraction.

**Theorem 10**  $(\forall c_1, c_2 \in C : (c_1 \xrightarrow{zz} c_2) \Rightarrow (c_1.date < c_2.date)) \Leftrightarrow ((C, \xrightarrow{zz}) \text{ is } z\text{-cycle-free}).$

*Proof* Direction  $\Rightarrow$ . Let us assume by contradiction that there is a z-cycle  $c \xrightarrow{zz} c$ . It follows from the assumption that  $c.date < c.date$ , which is clearly impossible.

Direction  $\Leftarrow$ . Let us consider that  $(C, \xrightarrow{zz})$  is acyclic. There is consequently a topological sort of its vertices. It follows that each vertex  $c$  (local checkpoint) can be labeled with an integer  $c.date$  such that  $c_1 \xrightarrow{zz} c_2 \Rightarrow c_1.date < c_2.date$ .  $\square$

This theorem shows that all the algorithms ensuring the z-cycle-freedom property implement (in an explicit or implicit way) a consistent logical dating of local checkpoints (the time notion being linear time). It follows that, when considering the algorithms that implement explicitly such a consistent dating system, the local checkpoints that have the same date belong to the same consistent global checkpoint.

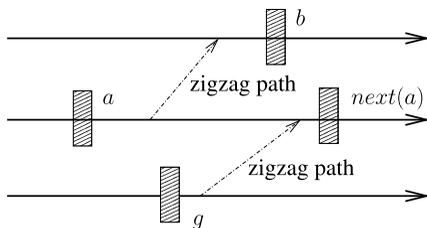
#### 8.3.2 A Property of a Particular Dating System

Given a checkpointing abstraction  $(C, \xrightarrow{zz})$  that satisfies the z-cycle-freedom property, let us consider the dating system that associates with each local checkpoint  $c$  the lowest possible consistent date, i.e.,  $c.date = \min\{c'.date \mid c' \xrightarrow{zz} c\} + 1$ .

This dating system has an interesting property, namely, it allows us to systematically associate with each local checkpoint  $c$  a consistent global checkpoint to which  $c$  belongs. It follows that any communication-induced checkpointing algorithm that implements this dating system has this property. The following theorem, which captures this dating system, is due to J.-M. H elary, A. Most efaoui, R.H.B. Netzer, and M. Raynal (2000).

**Fig. 8.6**

Proof by contradiction  
of Theorem 11



**Theorem 11** Let  $(C, \xrightarrow{zz})$  be a  $z$ -cycle-free checkpointing and communication pattern abstraction, in which the first local checkpoint of each process is dated 0, and the date of each other local checkpoint  $c$  is such that  $c.date = \max\{c'.date \mid c' \xrightarrow{zz} c\} + 1$ . Let us associate with each local checkpoint  $c$  the global checkpoint  $\Sigma(c) = [c_1^{x_1}, \dots, c_1^{x_n}]$  where  $c_i^{x_i}$  is the last local checkpoint of  $p_i$ , such that  $c_i^{x_i}.date \leq c.date$ . Then,  $\Sigma(c)$  includes  $c$  and is consistent.

*Proof* By construction of  $\Sigma(c)$ ,  $c$  belongs to  $\Sigma(c)$ . The proof that  $\Sigma(c)$  is consistent is by contradiction;  $\alpha$  being the date of  $c$  (i.e.,  $c.date = \alpha$ ), let us assume that  $\Sigma(c)$  is not consistent. Let  $next(x)$  be the local checkpoint (if any) that appears immediately after  $x$  on the same process.

- R1 Due to the theorem assumption, we have  $y.date \leq \alpha$  for any  $y \in \Sigma(c)$ .
- R2 If  $y \in \Sigma(c)$  and  $next(y)$  exist, due to the definition of  $y.date$ , we have  $next(y).date > \alpha$ .
- R3 As  $\Sigma(c)$  is not consistent (assumption), there are local checkpoints  $a, b \in \Sigma(c)$  such that  $a \xrightarrow{zz} b$  (Fig. 8.6).
- R4 It follows from R3, the way dates are defined, and R1 that  $a.date < b.date \leq \alpha$ .
- R5 As the last local state of any process is a local checkpoint, and this local checkpoint does not  $z$ -precede any other local checkpoint, it follows from R3 that  $next(a)$  exists.
- R6 It follows from R5 and R2 applied to  $next(a)$ , that  $next(a).date > \alpha$ . Moreover, due to the property of the underlying dating system (stated in the theorem) we also have  $next(a).date = \max\{c'.date \mid c' \xrightarrow{zz} next(a)\} + 1 > \alpha$ . Hence,  $\max\{c'.date \mid c' \xrightarrow{zz} next(a)\} \geq \alpha$ .
- R7 From  $a \xrightarrow{zz} next(a)$  and R4 (namely,  $a.date < \alpha$ ), we conclude that the maximal date used in R6 is associated with a local checkpoint  $g$  produced on a process different from the one on which  $a$  has been produced, and we have  $g.date = \max\{c'.date \mid c' \xrightarrow{zz} next(a)\} \geq \alpha$ .
- R8 It follows from R7 that  $g \xrightarrow{zz} next(a)$ . As  $a \xrightarrow{zz} b$ , we have  $g \xrightarrow{zz} b$  and, due to the dating system,  $date(g) < date(b)$ .
- R9 It follows from R7 and R8 that  $date(b) > \alpha$ . But, as  $b \in \Sigma(c)$ , this contradicts R1 which states that  $date(b) \leq \alpha$ , which concludes the proof of the theorem.  $\square$

```

internal operation take_local_checkpoint() is
(1)  $c \leftarrow$  copy of current local state;  $c.date \leftarrow clock_i$ ;
(2) save  $c$  and its date  $c.date$ .

when  $p_i$  decides to take a spontaneous checkpoint do
(3)  $clock_i \leftarrow clock_i + 1$ ; take_local_checkpoint().

when sending  $MSG(m)$  to  $p_j$  do
(4) send  $MSG(m, clock_i)$  to  $p_j$ .

when receiving  $MSG(m, sd)$  from  $p_j$  do
(5) if ( $clock_i < sd$ ) then
(6)    $clock_i \leftarrow sd$ ; take_local_checkpoint()   % forced local checkpoint
(7) end if;
(8) Deliver the message  $m$  to the application process.

```

**Fig. 8.7** A very simple z-cycle-free checkpointing algorithm (code for  $p_i$ )

### 8.3.3 Two Simple Algorithms Ensuring Z-Cycle Prevention

Both the algorithms that follow are based on Theorems 10 and 11: They both ensure z-cycle-freedom and associate dates with local checkpoints as described in Theorem 11. Hence, given any local checkpoint  $c$ , the processes are able to determine a consistent global checkpoint to which  $c$  belongs.

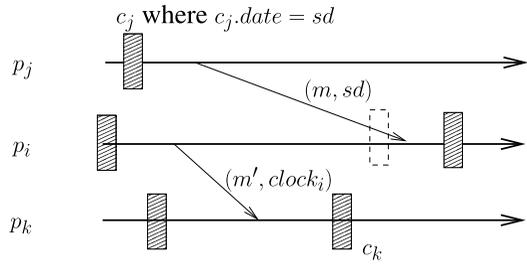
To that end, each process  $p_i$  has a scalar local variable  $clock_i$  that it manages and uses to associate a date with its local checkpoints. Moreover, each process defines its initial local state as its first local checkpoint (e.g., with date 1), and all local clocks are initialized to that value.

**A Simple Algorithm** A fairly simple checkpointing algorithm, which ensures the z-cycle-freedom property, is described on Fig. 8.7. This algorithm is due to D. Manivannan and M. Singhal (1996). Before taking a spontaneous local checkpoint a process  $p_i$  increases its local clock (line 3) whose next value is associated with this checkpoint (line 1). Each message carries the current value of the clock of its sender (line 4). Finally, when a process  $p_i$  receives a pair  $(m, sd)$  (where  $sd$  is the date of the last local checkpoint taken by the sender, before it sent this message), it compares  $sd$  with  $clock_i$  (which is the date of its last local checkpoint). If  $sd > clock_i$ ,  $p_i$  resets its clock to  $sd$  and takes a forced local checkpoint (lines 5–6) whose date is  $sd$ , and passes the message to the application process only after the local checkpoint has been taken.

The behavior associated with a message reception is motivated by the following observation. The message  $m$  received by a process  $p_i$  might create a zz-pattern (from  $c_j$  to  $c_k$ , as depicted in Fig. 8.8). However, if, since its last checkpoint,  $p_i$  has sent messages with dates  $clock_i \geq sd$ , the zz-pattern created by  $m$  does not prevent local checkpoint dates from increasing in agreement with the z-precedence relation. When this occurs,  $p_i$  has nothing specific to do in order to guarantee the property

**Fig. 8.8**

To take or not to take a forced local checkpoint



$\forall c_1, c_2: (c_1 \xrightarrow{zz} c_2) \Rightarrow (c_1.date < c_2.date)$  (used in Theorem 10 to obtain z-cycle-freedom). On the contrary, if  $clock_i < sd$ ,  $p_i$  has to prevent the possible formation of a zz-pattern with nonincreasing dates. A simple way to solve this issue (and be in agreement with the assumption of Theorem 10) consists in directing  $p_i$  to update  $clock_i$  and take a forced local checkpoint.

**A Simple Improvement** Let us observe that, whatever the values of  $clock_i$  and  $sd$ , no zz-pattern can be formed if  $p_i$  has not sent messages since its last local checkpoint. Let us introduce a Boolean variable  $sent_i$  to capture this “no-send” pattern. The previous algorithm is consequently modified as follows:

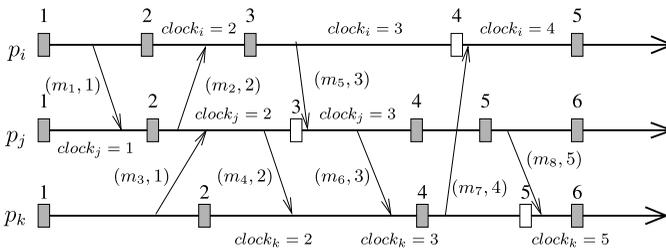
- $sent_i$  is set to *false* at line 1 and set to *true* at line 4.
- Moreover, line 6 becomes

$clock_i \leftarrow sd$ ; **if** ( $sent_i$ ) **then** take\_local\_checkpoint() **end if**.

Figure 8.9 represents an execution of the checkpointing algorithm of Fig. 8.7 enriched with the local Boolean variables  $sent_i$ . The added forced local checkpoints are depicted with white rectangles (the base execution is the one described in Fig. 8.1 with one more spontaneous local checkpoint—the one on  $p_j$  whose date is 5—and one more message, namely  $m_8$ ).

Let  $c(x, y)$  be the checkpoint of  $p_x$  whose date is  $y$ . The forced local checkpoint  $c(j, 3)$  is to prevent the zz-pattern  $\langle m_5, m_4 \rangle$  from forming. If, when  $p_j$  receives  $m_5$ ,  $clock_j$  was greater than or equal to 3 (the date of the last local checkpoint of the sender of  $m_5$ ), this forced local checkpoint would not have been taken.

Differently,  $p_j$  does not have to prevent the zz-pattern  $\langle m_3, m_2 \rangle$  from forming because it has previously taken a local checkpoint dated 2, and the message  $m_3$  it



**Fig. 8.9** An example of z-cycle prevention

receives has a smaller date. This means that  $c(j, 2)$  can be combined with  $c(k, 1)$  to form a consistent global checkpoint (let us notice that, as shown by the figure, it can also be combined with  $c(k, 2)$ , and that is the combination imposed by Theorem 11). The forced local checkpoint  $c(i, 4)$  is taken to prevent the formation of a z-cycle including  $c(k, 4)$ , while  $c(k, 5)$  is taken to prevent the possible formation of a z-cycle (that does not exist). Additionally,  $c(k, 5)$  is needed to associate a consistent global checkpoint with  $c(i, 5)$  as defined by Theorem 11). If  $c(k, 5)$  is not taken but the clock of  $p_k$  is updated when  $m_8$  is received,  $c(i, 5)$  would be associated with  $c(j, 5)$  and  $c(k, 4)$  which (due to message  $m_7$ ) defines an inconsistent global checkpoint. If  $c(k, 5)$  is not taken and the clock of  $p_k$  is not updated when  $m_8$  is received,  $c(k, 6)$  would be dated 5 and consequently denoted  $c'(k, 5)$ ;  $c(i, 5)$  would then be associated with  $c(j, 5)$  and  $c'(k, 5)$ , which (due to message  $m_8$ ) defines an inconsistent global checkpoint.

### 8.3.4 On the Notion of an Optimal Algorithm for Z-Cycle Prevention

An important issue concerns the design of an optimal communication-induced checkpointing algorithm which ensures z-cycle prevention. “Optimal” means here that the algorithm has to take as few forced local checkpoints as possible.

As it has been seen in Chaps. 6 and 7, the knowledge which is accessible to a process (and from which it can then benefit) is restricted to its causal past. This is because a process learns new information only when it receives messages. It is possible to design a communication-induced z-cycle-preventing algorithm which is optimal with respect to the communication and message pattern included in its causal past (see the bibliographic notes at the end of the chapter). But such an algorithm is not necessarily optimal with respect to the whole computation. This is due to the fact that, based on the information extracted from its causal past, a process  $p_i$  may not be forced to take a local checkpoint when it receives some message  $m$ . But, due to the (unknown for the moment) pattern of messages exchanged in the future, taking a local checkpoint when  $m$  is received could save the taking of local checkpoints in the future of  $p_i$  or other processes. In that sense, there is no optimal algorithm for z-cycle prevention.

## 8.4 Checkpointing Algorithms Ensuring Rollback-Dependency Trackability

### 8.4.1 Rollback-Dependency Trackability (RDT)

**Definition Reminder** As seen in Sect. 8.2.2, rollback-dependency trackability (RDT) is a consistency condition for communication and checkpoint patterns, which

```

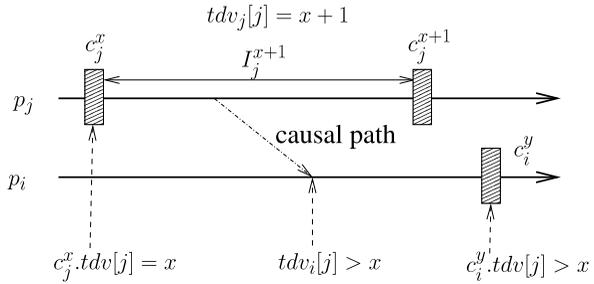
just after  $p_i$  has taken a spontaneous/forced local checkpoint do
(1)  $tdv_i[i] \leftarrow tdv_i[i] + 1$ .

when sending  $MSG(m)$  to  $p_j$  do
(2) send  $MSG(m, tdv_i[1..n])$  to  $p_j$ .

when receiving  $MSG(m, tdv)$  from  $p_j$  do
(3) for each  $k \in \{1, \dots, n\}$  do  $tdv_i[k] \leftarrow \max(tdvi[k], tdv[k])$  end for.
    
```

**Fig. 8.10** A vector clock system for rollback-dependency trackability (code for  $p_i$ )

**Fig. 8.11** Intervals and vector clocks for rollback-dependency trackability

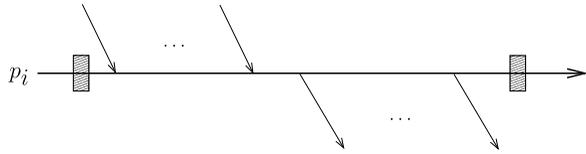


is stronger than z-cycle-freedom:  $c_1$  and  $c_2$  being any pair of local checkpoints, RDT states that  $(c_1 \xrightarrow{zz} c_2) \Rightarrow (c_1 \xrightarrow{\sigma} c_2)$ . This means that there is no z-dependence relation among local checkpoints which remains hidden from a causal precedence point of view. As we have seen, given any local checkpoint  $c$ , a noteworthy property of RDT is the possibility to associate with  $c$  a global checkpoint to which it belongs, and this can be done on the fly and without additional communication among processes.

**Vector Clock for RDT** As causality tracking is involved in RDT, at the operational level, a vector clock system suited to checkpointing can be defined as follows. Each process  $p_i$  has a vector clock, denoted  $tdv_i[1..n]$  (transitive dependence vector), managed as described in Fig. 8.10. The entry  $tdv_i[i]$  is initialized to 1, while all other entries are initialized to 0. Process  $p_i$  increases  $tdv_i[i]$  after it has taken a new local checkpoint. Hence,  $tdv_i[i]$  is the sequence number of the current interval of  $p_i$  (see Fig. 8.1), which means that  $tdv_i[i]$  is the sequence number of its next local checkpoint.

A vector date is associated with each local checkpoint. Its value is the current value of the vector clock  $tdv_i[1..n]$  of the process  $p_i$  that takes the corresponding local checkpoint. Let us consider Fig. 8.11 in which  $I_j^{x+1}$  is the interval separating  $c_j^x$  and  $c_j^{x+1}$ , which means that  $tdv_j[j] = x + 1$  just after  $c_j^x$ . It follows that the messages forming the causal path from  $I_j^{x+1}$  to  $p_i$  carry a value  $tdv[j] > x$ , and we have consequently  $c_i^y.tdv[j] > x$ .

**Fig. 8.12** Russell’s pattern for ensuring the RDT consistency condition



The RDT property can consequently be re-stated in an operational way as follows, where  $c_1$  and  $c_2$  are any two distinct local checkpoints and  $c_1$  belongs to  $p_j$ :

$$(c_1 \xrightarrow{zz} c_2) \Leftrightarrow (c_1.tdv[j] < c_2.tdv[j]).$$

If the communication and checkpoint pattern satisfies the RDT consistency condition, the previous vector clock system allows a process  $p_i$  to associate on the fly with each of its local checkpoint  $c$  the first consistent global checkpoint including  $c$ . This global checkpoint  $\Sigma = [c_1, \dots, c_n]$  is defined as follows: (a)  $c_i$  is  $c$  (the  $(c.tdv[i])$ th local checkpoint on  $p_i$ ) and (b) for any  $j \neq i$ ,  $c_j$  is the  $(c.tdv[j])$ th local checkpoint of  $p_j$ .

### 8.4.2 A Simple Brute Force RDT Checkpointing Algorithm

To ensure the RDT property, a very simple algorithm consists in preventing any  $zz$ -pattern from forming. To that end, the algorithm forces the communication and checkpoint pattern to be such that, at any process, there is no message sending followed by a message reception without a local checkpoint separating them. The only pattern allowed is the one described in Fig. 8.12. This pattern (called Russell’s pattern) states that the only message pattern that can appear at a process between two consecutive local checkpoints is a (possibly empty) sequence of message receptions, followed by a (possibly empty) sequence of message sendings.

The corresponding algorithm (due to Russell, 1980), is described in Fig. 8.13. The interest of this algorithm lies in its simplicity and in the fact that it needs only one Boolean per process.

```

internal operation take_local_checkpoint() is
(1)  $c \leftarrow$  copy of current local state; save  $c$ ;
(2)  $sent_i \leftarrow false$ .

when  $p_i$  decides to take a spontaneous checkpoint do
(3) take_local_checkpoint().

when sending MSG( $m$ ) to  $p_j$  do
(4) send MSG( $m$ ) to  $p_j$ ;  $sent_i \leftarrow true$ .

when receiving MSG( $m$ ) from  $p_j$  do
(5) if ( $sent_i$ ) then take_local_checkpoint() end if; % forced local checkpoint
(6) Deliver the message  $m$  to the application process.
    
```

**Fig. 8.13** Russell’s checkpointing algorithm (code for  $p_i$ )

It is possible to use the vector clock algorithm of Fig. 8.10 to associate a vector date with each checkpoint  $c$ . In this way, we obtain on the fly the first consistent global checkpoint to which  $c$  belongs.

### 8.4.3 The Fixed Dependency After Send (FDAS) RDT Checkpointing Algorithm

**On Predicates for Forced Local Checkpoints** According to the control data managed by processes and carried by application messages, stronger predicates governing forced checkpoints can be designed, where the meaning of “stronger” is the following. Let  $P1$  and  $P2$  be two predicates used to take forced checkpoints (when true, a checkpoint has to be taken).  $P1$  is stronger than  $P2$  if, when evaluated in the same context, we always have  $P1 \Rightarrow P2$ . This means that when  $P2$  is true while  $P1$  is false, no forced local checkpoint is taken if  $P1$  is used, while  $P2$  might force a local checkpoint to be taken. Hence  $P1$  is stronger in the sense that it allows for less forced checkpoints.

Let us recall that taking a local checkpoint can be expensive, mainly when it has to be saved on a disk. Hence, finding strong predicates, i.e., predicates that allows for “as few as possible” forced checkpoints is important.

This section presents a predicate which is stronger than the one used in the algorithm of Fig. 8.13. This predicate, which is called “fixed dependency set after send” (FDAS) was introduced by Y.-M. Wang (1997).

**The FDAS Predicate and the FDAS Algorithm** The predicate is on the local Boolean variable  $sent_i$  used in Russell’s checkpointing algorithm (Fig. 8.13), the vector clock  $tdv_i[1..n]$  of the process  $p_i$ , and the vector date  $tdv[1..n]$  carried by the message received by  $p_i$ . The corresponding checkpointing algorithm, with the management of vector clocks, is described in Fig. 8.14. The predicate controlling forced checkpoints appears at line 5. It is the following

$$sent_i \wedge (\exists k : tdv[k] > tdv_i[k]).$$

A process takes a forced local checkpoint if, when a message  $m$  arrives, it has sent a message since its last local checkpoint and, because of  $m$ , its vector  $tdv_i[1..n]$  is about to change.

As we have already seen, no zz-pattern can be created by the reception of a message  $m$  if the receiver  $p_i$  has not sent messages since its last checkpoint. Consequently, if  $sent_i$  is false, no local forced checkpoint is needed. As far as the second part of the predicate is concerned, we have the following. If  $\forall k : tdv[k] \leq tdv_i[k]$ , from a local checkpoint point of view,  $p_i$  knows everything that was known by the sender of  $m$  (when it sent  $m$ ). As this message does not provide  $p_i$  with new information on dependencies among local checkpoints, it cannot create local checkpoint

```

internal operation take_local_checkpoint() is
(1)  $c \leftarrow$  copy of current local state; save  $c$  and its vector date  $c.tdv = tdv_i[1..n]$ ;
(2)  $sent_i \leftarrow false$ ;  $tdv_i[i] \leftarrow tdv_i[i] + 1$ .

when  $p_i$  decides to take a spontaneous checkpoint do
(3) take_local_checkpoint().

when sending MSG( $m$ ) to  $p_j$  do
(4) send MSG( $m, tdv_i$ ) to  $p_j$ ;  $sent_i \leftarrow true$ .

when receiving MSG( $m, tdv$ ) from  $p_j$  do
(5) if ( $sent_i \wedge (\exists k : tdv[k] > tdv_i[k])$ )
(6)   then take_local_checkpoint() % forced local checkpoint
(7)   end if;
(8)   for each  $k \in \{1, \dots, n\}$  do  $tdv_i[k] \leftarrow \max(tdv_i[k], tdv[k])$  end for;
(9)   Deliver the message  $m$  to the application process.

```

**Fig. 8.14** FDAS checkpointing algorithm (code for  $p_i$ )

dependencies that would remain unknown to  $p_i$ . Hence, the name FDAS comes from the fact that, at any process, after the first message sending in any interval, the transitive dependency vector remains unchanged until the next local checkpoint.

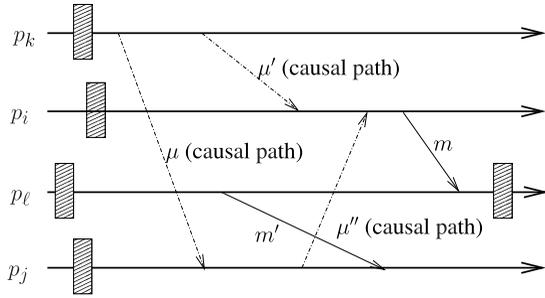
#### 8.4.4 Still Reducing the Number of Forced Local Checkpoints

The fact that a process takes a forced local checkpoint at some point of its execution has trivially a direct impact on the communication and checkpoint pattern defined by the application program: The communication pattern is not modified but the checkpoint pattern is. This means that it might not be possible to determine on the fly if taking now an additional forced local checkpoint would decrease the number of forced local checkpoints taken in the future. The best that can be done is to find predicates (such as FDAS) that, according to the control information at their disposal, strive to take as few as possible forced local checkpoints. This section presents such a predicate (called BHMR) and the associated checkpointing algorithm. This predicate, which is stronger than FDAS, and the associated algorithm are due to R. Baldoni, J.-M. H elary, A. Most efaoui, and M. Raynal (1997).

**Additional Control Variables** The idea that underlies the design of this predicate and the associated checkpointing algorithm is based on additional control variables that capture the interplay of the causal precedence relation and the last local checkpoints taken by each process. To that end, in addition to the vector clock  $tdv_i[1..n]$ , each process manages the following local variables, which are all Boolean arrays.

- $sent\_to_i[1..n]$  is a Boolean array such that  $sent\_to_i[j]$  is true if and only if  $p_i$  sent a message to  $p_j$  since its last local checkpoint. (This array replaces the Boolean variable  $sent_i$  used in the FDAS algorithm, Fig. 8.14.) Initially, for any  $j$ , we have  $sent\_to_i[j] = false$ .

**Fig. 8.15** Matrix  $causal_i[1..n, 1..n]$

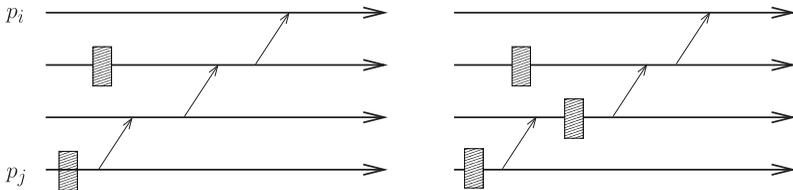


- $causal_i[1..n, 1..n]$  is a two-dimensional Boolean array such that  $causal_i[k, j]$  is true if and only if, to  $p_i$ 's knowledge, there is a causal path from the last local checkpoint taken by  $p_k$  (as known by  $p_i$ ) to the next local checkpoint that will be taken by  $p_j$  (this is the local checkpoint of  $p_j$  that follows its last local checkpoint known by  $p_i$ ). Initially,  $causal_i[1..n, 1..n]$  is equal to *true* on its diagonal, and equal to *false* everywhere else.

As an example, let us consider Fig. 8.15 where  $\mu$ ,  $\mu'$  and  $\mu''$  are causal paths. When  $p_i$  sends the message  $m$ , we have  $causal_i[k, j] = true$  ( $p_i$  learned the causal path  $\mu$  from  $p_k$  to  $p_j$  thanks to the causal path  $\mu''$ ),  $causal_i[k, i] = true$  (this is due to both the causal paths  $\mu'$  and  $\langle \mu; \mu'' \rangle$ ), and  $causal_i[j, i] = true$  (this is due to the causal path  $\mu''$ ).

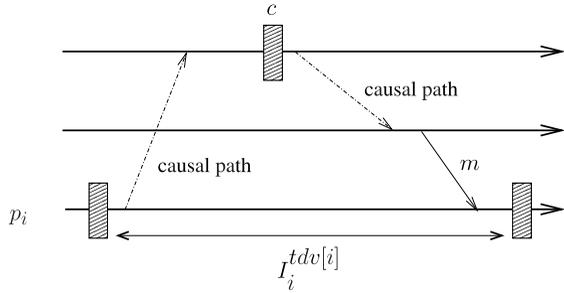
- $pure_i[1..n]$  is a Boolean array such that  $pure_i[j]$  is true if and only if, to  $p_i$ 's knowledge, no causal path starting at the last local checkpoint of  $p_j$  (known by  $p_i$ ) and ending at  $p_i$ , contains a local checkpoint. An example is given in Fig. 8.16. A causal path from a process to itself without local checkpoints is called *pure*. The entry  $pure_i[i]$  is initialized to *true* and keeps that value forever; for any  $j \neq i$ ,  $pure_i[j]$  is initialized to *false*.

**The BHM Predicate to Take Forced Local Checkpoints** Let  $MSG(m, tdv, pure, causal)$  be a message received by  $p_i$ . Hence, if  $p_j$  is the sender of this message,  $tdv[1..n]$ ,  $pure[1..n]$ , and  $causal[1..n]$  are the values of  $tdv_j$ ,  $pure_j$ , and  $causal_j$ , respectively, when it sent the message. The predicate is made up of two parts.



**Fig. 8.16** Pure (left) vs. impure (right) causal paths from  $p_j$  to  $p_i$

**Fig. 8.17** An impure causal path from  $p_i$  to itself



- The first part of the predicate, which concerns the causal paths from any process  $p_j$  ( $j \neq i$ ) to  $p_i$ , is

$$\exists(k, \ell): \text{sent\_to}_i[k] \wedge ((tdv[k] > tdv_i[k]) \wedge \neg\text{causal}[k, \ell]).$$

As we can see, the sub-predicate  $\exists k : \text{sent\_to}_i[k] \wedge (tdv[k] > tdv_i[k])$  is just the FDAS predicate expressed on a process basis. If it is true,  $p_i$  sent a message  $m'$  to  $p_k$  since its last checkpoint and the sender  $p_j$  knows more local checkpoints of  $p_k$  than  $p_i$ .

But, if  $\text{causal}[k, \ell]$  is true,  $p_j$  knows also that there is a causal path from the last local checkpoint it knows of  $p_k$  to  $p_\ell$ . Hence, there is no need for  $p_i$  to take a forced local checkpoint as the zz-pattern created by the message  $m$  just received, and the message  $m'$  previously sent by  $p_i$  to  $p_k$ , is doubled by a causal path (see the zigzag path  $\langle \mu' ; m ; m' \rangle$  in Fig. 8.15 which is doubled by the causal path  $\mu$ ). Consequently,  $p_i$  takes conservatively a local checkpoint only if  $\neg\text{causal}[k, \ell]$ .

- The second part of the predicate concerns the causal paths from  $p_i$  to itself, which start after its last local checkpoint. It is

$$(tdv[i] = tdv_i[i]) \wedge \neg\text{pure}[i].$$

In this case (see Fig. 8.17), if the causal path whose last message is  $m$  is not pure, a local checkpoint  $c$  has been taken along this causal path starting and ending in the same interval  $I_i^{tdv[i]}$  of  $p_i$ . In order this checkpoint  $c$  belongs to a consistent global checkpoint,  $p_i$  takes a forced local checkpoint (otherwise, a z-cycle would form).

**The BHMR Checkpointing Algorithm** The algorithm based on the previous control data structures and predicates is described in Fig. 8.18. When a process  $p_i$  takes a local checkpoint, it defines its vector date as the current value of its vector clock  $tdv_i[1..n]$  (line 1), and updates accordingly the appropriate entries of its arrays  $\text{sent\_to}_i$ ,  $\text{pure}_i$  and  $\text{causal}_i$  (lines 2–3). Finally, it increases  $tdv_i[i]$  to the sequence number of its new interval.

When it sends a message  $m$ , a process  $p_i$  adds to it the current values of its three control data structures (line 6). When it receives a message  $\text{MSG}(m, tdv_j[1..n], \text{pure}[1..n], \text{causal}[1..n, 1..n])$  from a process  $p_j$ ,  $p_i$  check first the predicate BHMR

```

internal operation take_local_checkpoint() is
(1)  $c \leftarrow$  copy of current local state; save  $c$  and its vector date  $c.tdv = tdv_i[1..n]$ ;
(2) for each  $k \in \{1, \dots, n\}$  do  $sent\_to_i[k] \leftarrow false$  end for;
(3) for each  $k \in \{1, \dots, n\} \setminus \{i\}$  do  $pure_i[k] \leftarrow false$ ;  $causal_i[i, k] \leftarrow false$  end for;
(4)  $tdv_i[i] \leftarrow tdv_i[i] + 1$ .

when  $p_i$  decides to take a spontaneous checkpoint do
(5) take_local_checkpoint().

when sending MSG( $m$ ) to  $p_j$  do
(6) send MSG( $m, tdv_i[1..n], pure_i[1..n], causal_i[1..n, 1..n]$ ) to  $p_j$ ;
(7)  $sent\_to_i[j] \leftarrow true$ .

when receiving MSG( $m, tdv_j[1..n], pure_j[1..n], causal_j[1..n, 1..n]$ ) from  $p_j$  do
(8) if  $[\exists(k, \ell) : sent\_to_i[k] \wedge ((tdv[k] > tdv_i[k]) \wedge \neg causal[k, \ell])$ 
(9)  $\vee [(tdv_i[i] = tdv_j[i]) \wedge \neg pure_i[i]]]$ 
(10) then take_local_checkpoint() % forced local checkpoint
(11) end if;
(12) for each  $k \in \{1, \dots, n\}$  do
(13) case  $(tdv[k] > tdv_i[k])$  then
(14)  $tdv_i[k] \leftarrow tdv[k]$ ;  $pure_i[k] \leftarrow pure[k]$ ;
(15) for each  $\ell \in \{1, \dots, n\}$  do  $causal_i[k, \ell] \leftarrow causal[k, \ell]$  end for
(16)  $(tdv[k] = tdv_i[k])$  then
(17)  $pure_i[k] \leftarrow pure_i[k] \wedge pure[k]$ ;
(18) for each  $\ell \in \{1, \dots, n\}$ 
(19) do  $causal_i[k, \ell] \leftarrow causal_i[k, \ell] \vee causal[k, \ell]$ 
(20) end for
(21)  $(tdv[k] < tdv_i[k])$  then skip
(22) end case
(23) end for;
(24) for each  $\ell \in \{1, \dots, n\}$  do  $causal_i[\ell, i] \leftarrow causal_i[\ell, i] \vee causal[\ell, j]$  end for;
(25) Deliver the message  $m$  to the application process.

```

**Fig. 8.18** An efficient checkpointing algorithm for RDT (code for  $p_i$ )

that has been presented previously (lines 8–9). Then, before delivering the message (line 25),  $p_i$  updates its control data structures so that their current values correspond to their definition (lines 12–24).

For each process  $p_k$ ,  $p_i$  compares  $tdv[k]$  (the value of  $tdv_j[k]$  when  $p_j$  sent the message) with its own value  $tdv_i[k]$ .

- If  $tdv[k] > tdv_i[k]$ ,  $p_j$  knows more local checkpoints of  $p_k$  than  $p_i$ . In this case,  $p_i$  resets  $tdv_i[k]$ ,  $pure_i[k]$ , and  $causal_i[k, \ell]$  for every  $\ell$ , to the corresponding more up to date values sent by  $p_j$  (lines 13–15).
- If  $tdv[k] = tdv_i[k]$ ,  $p_i$  and  $p_j$  know the same last local checkpoint of  $p_k$ . As they possibly know it through different causal paths,  $p_i$  adds what is known by  $p_j$  ( $pure[k]$  and  $causal[k, \ell]$ ) to what it already knows (lines 16–18).
- If  $p_i$  knows more on  $p_k$  than  $p_j$ , it is more up to date and consequently does nothing (line 21).

Finally, as the message  $m$  extends causal paths ending at  $p_j$ , process  $p_i$  updates accordingly each Boolean  $causal[\ell, i]$ ,  $1 \leq \ell \leq n$  (line 24).

This algorithm reduces the number of forced local checkpoints at the price of more control data and more information carried by each application message, which has to carry  $n^2 + n$  bits and  $n$  integers (logical dates). Nevertheless, as we have seen in Sect. 8.3.4 for z-cycle prevention, there is no optimal communication-induced checkpointing algorithm that ensures the RDT property.

## 8.5 Message Logging for Uncoordinated Checkpointing

### 8.5.1 Uncoordinated Checkpointing

In uncoordinated checkpointing, each process defines independently its local checkpoints and there is no notion of a forced checkpoint. While this approach is prone to the domino effect, it can be interesting for some applications (for example, *backward recovery* after a failure) where processes take few local checkpoints and do it periodically. A consistent global checkpoint has then to be computed by a *recovery* algorithm. Moreover, according to the aim of the computed global checkpoint, this algorithm might also have to compute channel states, which have to be consistent with the computed global checkpoint. It follows that, if channel states have to be computed, the processes have to *log messages on stable storage* during their execution.

**Pessimistic vs. Optimistic Message Logging** The message logging technique is called *sender-based* (resp., *receiver-based*) if messages are logged by their senders (resp., receivers). Two logging techniques are possible.

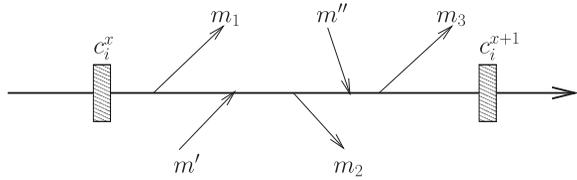
- In the case of *pessimistic logging*, a message is saved on stable storage by its sender (receiver) at the time it is sent (received). This can incur high overhead in failure-free executions, as each message entails an additional input/output.
- In the case of *optimistic logging*, a message is first saved in a *volatile log*, and this log is then saved on stable storage when a process takes a local checkpoint. When this occurs, the corresponding process saves on stable storage both its local state and the messages which are in its volatile log.

**Content of This Section** Considering an uncoordinated checkpointing algorithm, this section presents an optimistic sender-based message logging algorithm. The main feature of this algorithm is that only a subset of messages logged on a volatile log have to be saved on stable storage. To simplify the presentation, the channels are assumed to be FIFO.

### 8.5.2 To Log or Not to Log Messages on Stable Storage

**Basic Principle** Let  $c_i^x$  denote the  $x$ th local checkpoint taken by a process  $p_i$ . When  $p_i$  sends a message  $m$ , it saves it in a local volatile log. When later  $p_i$  takes

**Fig. 8.19** Sender-based optimistic message logging



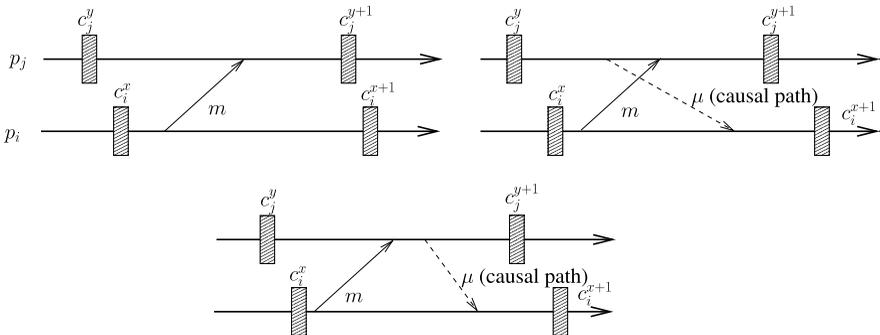
its next local checkpoint  $c_i^{x+1}$ , it (a) saves on stable storage  $c_i^{x+1}$  and the content of its volatile log, and (b) re-initializes to empty its volatile log. Hence, the messages are saved on stable storage by batch, and not individually. This decreases the number of input/output and consequently the overhead associated with message logging.

A simple example is depicted in Fig. 8.19. After it has taken its local checkpoint  $c_i^x$ , the volatile log of  $p_i$  is empty. Then, when it sends  $m_1$ ,  $m_2$ , and  $m_3$ , it saves them in its volatile log. Finally, when it takes  $c_i^{x+1}$ ,  $p_i$  writes  $c_i^{x+1}$  and the current content of its volatile message log on stable storage before emptying its volatile log.

**To Log or Not to Log: That Is the Question** The question is then the following one: Which messages saved in its volatile log,  $p_i$  has to save in stable storage when it take its next local checkpoint  $c_i^{x+1}$ ?

To answer this question, let us consider Fig. 8.20. When looking at the execution of the left side, the local checkpoints  $c_j^y$  and  $c_i^{x+1}$  are concurrent and can consequently belong to the same consistent global checkpoint. The corresponding state of the channel from  $p_i$  to  $p_j$  then comprises the message  $m$  (this message is *in transit* with respect to the ordered pair  $(c_i^{x+1}, c_j^y)$ ).

When looking at the execution of the right side, there is an additional causal path starting after  $c_j^y$  and ending before  $c_i^{x+1}$ . It follows that, in this case,  $c_j^y$  and  $c_i^{x+1}$  are no longer concurrent (independent) and it is not necessary to save the message  $m$  on stable storage as it cannot appear in a consistent global checkpoint. Hence, if  $p_i$  knows this fact, it does not have to save  $m$  from its volatile storage to stable storage. Process  $p_i$  can learn it if there is causal path starting from  $p_j$  after it has



**Fig. 8.20** To log or not to log a message?

received  $m$ , and arriving at  $p_i$  before it takes  $c_i^{x+1}$  (this is depicted in the execution at the bottom of the figure).

**Checkpointing Algorithm: Local Data Structures** Hence, a process  $p_i$  has to log on stable storage the messages it sends only if (from its point of view) they might belong to a consistent global checkpoint. To implement this idea, the checkpointing algorithm is enriched with the following control data structures.

- $volatile\_log_i$  is the volatile log of  $p_i$ . It is initially empty.
- $sn_i[1..n]$  is an array of sequence numbers.  $sn_i[j] = \alpha$  means that  $p_i$  has sent  $\alpha$  messages to  $p_j$  (as the channels are FIFO, those are the first  $\alpha$  messages sent by  $p_i$  to  $p_j$ ).
- $rec\_known_i[1..n, 1..n]$  is an array of sequence numbers which captures the knowledge of  $p_i$  on the messages that have been exchanged by each pair of processes.  $rec\_known_i[j, k] = \beta$  means that  $p_i$  knows (due to causal paths) that  $p_j$  has received  $\beta$  messages from  $p_k$ .

(If processes do not send messages to themselves, all entries  $rec\_known_i[j, j]$  remain equal to 0. From an implementation point of view, they can be used to store the array  $sn_i[1..n]$ .)

- $ckpt\_vc_i[1..n]$  is a vector clock associated with local checkpoints.  $ckpt\_vc_i[j] = \gamma$  means that  $p_i$  knows that  $p_j$  has taken  $\gamma$  local checkpoints. This control data is managed by the checkpointing algorithm, but used by the recovery algorithm to compute a consistent global checkpoint.

**Checkpointing Algorithm: Process Behavior** The behavior of a process  $p_i$  is described in Fig. 8.21.

When a process  $p_i$  sends a message  $m$  to a process  $p_j$ , it first adds the triple  $\langle m, sn_i[j], j \rangle$  to its volatile log (the pair  $(sn_i[j], j)$  is the identity of  $m$ , lines 1–2). Then it sends  $m$  to  $p_j$  with its vector date  $ckpt\_vc_i[1..n]$  and its current knowledge on the messages that have been received (line 3).

When it receives a message,  $p_i$  first updates its vector clock  $ckpt\_vc_i[1..n]$  (line 4), and its knowledge on which messages have been received (lines 5–6). Then it delivers the message (line 9).

When  $p_i$  takes an uncoordinated local checkpoint, it first updates its entry of its vector clock (line 10). Then it withdraws from its volatile log every message  $m$  it has sent and, to its current knowledge, has been received by its destination process  $p_{dest}$  (lines 11–13). Finally, before emptying its volatile log (line 15),  $p_i$  saves in stable storage the current state  $\sigma_i$  of the application process plus the control data that will allow for a computation of a consistent global checkpoint and the corresponding channel states (line 14). These control data comprise the volatile log plus three vectors of size  $n$ .

```

when sending MSG( $m$ ) to  $p_j$  do
(1)  $sn_i[j] \leftarrow sn_i[j] + 1$ ;
(2) add  $\langle m, sn_i[j], j \rangle$  to  $volatile\_log_i$ ;
(3) send MSG( $m, rec\_known_i[1..n, 1..n], ckpt\_vc_i[1..n]$ ) to  $p_j$ .

when receiving MSG( $m, rec\_known[1..n, 1..n], ckpt\_vc[1..n]$ ) from  $p_j$  do
(4) for each  $k \in \{1, \dots, n\}$  do  $ckpt\_vc_i[k] \leftarrow \max(ckpt\_vc_i[k], ckpt\_vc[k])$  end for;
(5)  $rec\_known_i[i, j] \leftarrow rec\_known_i[i, j] + 1$ ;
(6) for each  $k, \ell \in \{1, \dots, n\}$ 
(7)   do  $rec\_known_i[k, \ell] \leftarrow \max(rec\_known_i[k, \ell], rec\_known_i[k, \ell])$ 
(8) end for;
(9) Deliver the message  $m$  to the application process.

when  $p_i$  decides to take a (spontaneous) local checkpoint do
(10)  $ckpt\_vc_i[i] \leftarrow ckpt\_vc_i[i] + 1$ ;
(11) for each  $\langle m, sn, dest \rangle \in volatile\_log_i$  do
(12)   if  $(sn \leq rec\_known_i[dest, i])$  then withdraw  $\langle m, sn, dest \rangle$  from  $volatile\_log_i$  end if
(13) end for;
(14) save on stable storage the current local state plus
       $volatile\_log_i, sn_i[1..n], ckpt\_vc_i[1..n]$ , and  $rec\_known_i[i, 1..n]$ ;
(15) empty  $volatile\_log_i$ .

```

Fig. 8.21 An uncoordinated checkpointing algorithm (code for  $p_i$ )

### 8.5.3 A Recovery Algorithm

The following recovery algorithm is associated with the previous uncoordinated checkpointing algorithm. When a failure occurs, the recovery algorithm executes the following sequence of steps.

1. The non-faulty processes are first required to take a local checkpoint.
2. The most recent set of  $n$  concurrent local checkpoints it then computed iteratively.

Let  $\Sigma = [c_1, \dots, c_i, \dots, c_n]$ , where  $c_i$  is last local checkpoint taken by  $p_i$ ;

```

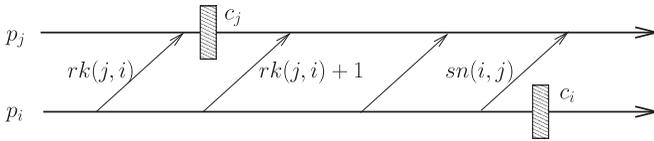
while  $(\exists (i, j)$  such that  $c_i \xrightarrow{\sigma} c_j)$  do
  let  $c'_j =$  first predecessor of  $c_j$  such that  $\neg(c_i \xrightarrow{\sigma} c'_j)$ ;
   $\Sigma \leftarrow [c_1, \dots, c_i, \dots, c'_j, \dots, c_n]$ 
end while.

```

Thanks to Theorem 6, the values of  $c_i \xrightarrow{\sigma} c_j$  and  $\neg(c_i \xrightarrow{\sigma} c'_j)$  can be efficiently determined from the vector date  $c_i.chpt\_vc$  associated with each local checkpoint.

Assuming that the initial local state of each process is a local checkpoint, the previous loop terminates and the resulting global checkpoint  $\Sigma$  is consistent.

3. The state of the channel connecting  $p_i$  to  $p_j$  (i.e., the sequence of messages which are in transit with respect to the ordered pair  $(c_i, c_j)$ ) is extracted from the stable storage of  $p_i$  as follows (let us note that only the sender  $p_i$  is involved in this computation).



**Fig. 8.22** Retrieving the messages which are in transit with respect to the pair  $(c_i, c_j)$

Let  $sn(i, j)$  be the value of the sequence number  $sn_i[j]$  which has been saved by  $p_i$  on its stable storage together with  $c_i$ ; this is the number of messages sent by  $p_i$  to  $p_j$  before taking its local checkpoint  $c_i$ . It follows that the messages sent by  $p_i$  to  $p_j$  (after  $c_i$ ) have a sequence number greater than  $sn(i, j)$ . Similarly, let  $rk(j, i)$  be the value of  $rec\_known_j[j, i]$  which has been saved by  $p_j$  on its stable storage together with  $c_j$ ; this is the number of messages received (from  $p_i$ ) by  $p_j$  before it took its local checkpoint  $c_j$ .

It follows that the messages from  $p_i$  received by  $p_j$  (before  $c_j$ ) have a sequence number smaller than or equal to  $rk(j, i)$ . As the channel is FIFO, the messages whose sequence number  $s_qnb$  is such that  $rk(j, i) < s_qnb \leq sn(i, j)$ , define the sequence of messages which are in transit with respect to ordered pair  $(c_i, c_j)$ . This is depicted in Fig. 8.22, which represents the sequence numbers attached to messages.

Due to the predicate used at line 12 of the checkpointing algorithm, these messages have not been withdrawn from the volatile log of  $p_i$  and are consequently with  $c_i$  in  $p_i$ 's stable storage.

### 8.5.4 A Few Improvements

**Adding Forced Checkpoints** Let us note that, even if the basic checkpointing algorithm is uncoordinated, a few forced checkpoints can be periodically taken to reduce the impact of the domino effect.

**Space Reclamation** As soon as a consistent global checkpoint  $\Sigma = [c_1, \dots, c_i, \dots, c_n]$  has been determined, the control data, kept in stable storage, which concern the local checkpoints which precede  $c_i$ ,  $1 \leq i \leq n$ , become useless and can be discarded.

Moreover, a background task can periodically compute the most recent consistent global checkpoint in order to save space in the stable storage of each process.

**Cost** The fact that each message sent by a process  $p_i$  has to carry the current value of the integer matrix  $rec\_known_i[1..n, 1..n]$  can reduce efficiency and penalize the application program. Actually, only the entries of the matrix corresponding to the (directed) channels of the application program have to be considered. When the communication graph is a directed ring, the matrix shrinks to a vector. A similar gain is obtained when the communication graph is a tree with bidirectional channels.

Moreover, as channels are FIFO, for any two consecutive messages  $m_1$  and  $m_2$  sent by  $p_i$  to  $p_j$ , for each entry  $rec\_known_i[k, \ell]$ ,  $m_2$  has only to carry the difference between its current value and its previous value (which was communicated to  $p_j$  by  $m_1$ ).

**The Case of Synchronous Systems** Let us finally note that synchronous systems can easily benefit from uncoordinated checkpointing without suffering the domino effect. To that end, it is sufficient for the processes to take local checkpoints at the end of each round of a predefined sequence of rounds.

## 8.6 Summary

This chapter was on asynchronous distributed checkpointing. Assuming that processes take local checkpoints independently from another, it has presented two consistency conditions which can be associated with local checkpoints, namely, z-cycle-freedom and rollback-dependency trackability. The chapter then presented distributed algorithms that force processes to take additional local checkpoints so that the resulting communication and checkpoint pattern satisfies the consistency condition we are interested in. The chapter also introduced important notions such as the notion of a zigzag path, and the z-dependence relation among local checkpoints. Finally, a message-logging algorithm suited to uncoordinated checkpointing was presented.

## 8.7 Bibliographic Notes

- The checkpointing problem is addressed in many textbooks devoted to operating systems. One of the very first papers that formalized this problem is due to B. Randell [303]. The *domino effect* notion was also introduced in this paper.
- The fundamental theorem on the necessary and sufficient condition for a set of local checkpoints to belong to a same consistent global checkpoint is due to R.H.B. Netzer and J. Xu [283]. This theorem was generalized to more general communication models in [175], and to the read/write shared memory model in [35].
- The notions of zigzag path and z-cycle-freedom were introduced R.H.B. Netzer and J. Xu [283].
- The notion of an *interval* and associated consistency theorems are presented in [174].
- The rollback-dependency trackability consistency condition was introduced by Y.-M. Wang [381]. An associated theory is presented and investigated in [33, 36, 145, 375].
- Theorems 10 and 11 on linear dating to prevent z-cycle-freedom are due to J.-M. HéLary, A. Mostéfaoui, R.H.B. Netzer, and M. Raynal [171].

- The algorithm that ensures z-cycle-freedom presented in Fig. 8.7 is due to D. Manivannan and M. Singhal [247].

A more sophisticated algorithm which takes fewer forced checkpoints is presented [171]. This algorithm manages additional control data and uses a sophisticated predicate (whose spirit is similar to that of Fig. 8.18) in order to decrease the number of forced checkpoints. This algorithm is optimal with respect to the causal past of each process.

An evaluation of z-cycle-free checkpointing algorithms is presented in [376].

- The FDAS algorithm that ensures rollback-dependency trackability described in Fig. 8.14 is due Y.-M. Wang [381]. Russell’s algorithm is described in [332].
- The algorithm that ensures rollback-dependency trackability described in Fig. 8.18 is due to R. Baldoni, J.-M., H elary, A. Most efaoui, and M. Raynal [34].
- Another algorithm that ensures rollback-dependency trackability is presented in [146]. A garbage collection algorithm suited to RDT checkpointing protocols is presented in [337].
- Numerous checkpointing algorithms are presented in the literature, e.g., [2, 52, 62, 75, 103, 129, 144, 207, 246, 299, 344] to cite a few. The textbooks [149, 219] contain specific chapters devoted to checkpointing.
- The sender-based message logging algorithm presented in Sect. 8.5 is due to A. Most efaoui and M. Raynal [271]. The matrix of sequence numbers used in this algorithm is actually a matrix time (as defined in Chap. 7).

Other message logging algorithms have been proposed (e.g., [16, 104, 202, 383]). Recovery is the topic of many papers (e.g., [161, 284, 352] to cite a few). Space reclamation in uncoordinated checkpointing is addressed in [382]. A nice survey on rollback-recovery protocols for message-passing systems is presented in [120].

## 8.8 Exercises and Problems

1. Let us consider the z-dependency relation  $\xrightarrow{zz}$  introduced in Sect. 8.1.

- Show that:

$$((c_i^x \xrightarrow{zz} c_k^{t+1}) \wedge (c_k^t \xrightarrow{zz} c_j^y)) \Rightarrow (c_i^x \xrightarrow{zz} c_j^y).$$

- Let  $c_1$  and  $c_2$  be two local checkpoints (from distinct processes). Show that they can belong to the same consistent global checkpoint if
  - no z-cycle involving  $c_1$  or  $c_2$  exists, and
  - no zigzag path exists connecting  $c_1$  and  $c_2$ .

Solution in [283].

2. Let  $c$  be a local checkpoint of a process  $p_i$ , which is not its initial checkpoint. The notation  $pred(c)$  is used to denote the local checkpoint that precedes immediately

$c$  on  $p_i$ . Let us define a new relation  $\xrightarrow{zz'}$  as follows.  $c_1 \xrightarrow{zz'} c_2$  if:

$$(c_1 \xrightarrow{\sigma} c_2) \wedge (\exists c \in C : (c_1 \xrightarrow{\sigma} c) \wedge (\text{pred}(e) \xrightarrow{zz'} c_2)).$$

Show first that  $\xrightarrow{zz'} \equiv \xrightarrow{zz}$ . Then give a proof of Theorem 9 based on the relations  $\xrightarrow{\sigma}$  and  $\xrightarrow{zz'}$  (instead of the relations  $\xrightarrow{\sigma}$  and  $\xrightarrow{zz}$ ).

Solution in [149] (Chap. 29).

3. Prove formally that the predicate BHMR used to take a forced checkpoint in Sect. 8.4.4 is stronger than FDAS.

Solution in [34].

4. The communication-induced checkpointing algorithms presented in Sects. 8.3 and 8.4 do not record the messages which are in transit with respect to the corresponding pairs of local checkpoints. Enrich one of the checkpointing algorithms presented in these sections so that in-transit messages are recorded.

Solution in [173].

5. Modify the uncoordinated checkpointing algorithm described in Sect. 8.5 so that only causal paths made up of a single message are considered (i.e., when considering that the causal path  $\mu$  of Fig. 8.20 has a single message, and this message is from  $p_j$ ).

When comparing this algorithm with the one described in Fig. 8.21, does this constraint reduce the size of the control information carried by messages? Does it increase or reduce the number of messages that are logged on stable storage? Which algorithm do you prefer (motivate your choice)?

# Chapter 9

## Simulating Synchrony on Top of Asynchronous Systems

Synchronous distributed algorithms are easier to design and analyze than their asynchronous counterparts. Unfortunately, they do not work when executed in an asynchronous system. Hence, the idea to simulate synchronous systems on top of an asynchronous one. Such a simulation algorithm is called a *synchronizer*. First, this chapter presents several synchronizers in the context of fully asynchronous systems. It is important to notice that, as the underlying system is asynchronous, the synchronous algorithms simulated on top of it cannot consider physical time as a programming object they could use (e.g., to measure physical duration). The only notion of time they can manipulate is a logical time associated with the concept of a round. Then, the chapter presents synchronizers suited to partially synchronous systems. Partial synchrony means here that message delays are bounded but the clocks of the processes (processors) are not synchronized (some private local area networks have such characteristics).

**Keywords** Asynchronous system · Bounded delay network · Complexity · Graph covering structure · Physical clock drift · Pulse-based programming · Synchronizer · Synchronous algorithm

### 9.1 Synchronous Systems, Asynchronous Systems, and Synchronizers

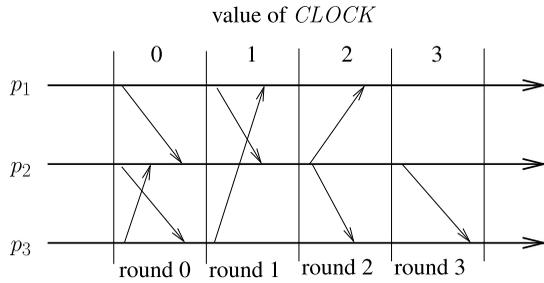
#### 9.1.1 Synchronous Systems

Synchronous systems were introduced in Chap. 1, and several synchronous algorithms have been presented in Chap. 2; namely the computation of shortest paths in Fig. 2.3, and the construction of a maximal independent set in Fig. 2.12. In the first of these algorithms, a process sends a message to each of its neighbors at every round, while in the second algorithm the set of neighbors to which it sends a message monotonically decreases as rounds progress.

In a general synchronous setting, a process may send messages to some subset of neighbors during a round, and to a different subset during another round. It can also send no message at all during a round. In the following, the terms “pulse” and “round” are considered synonyms.

**Fig. 9.1**

A space-time diagram  
of a synchronous execution



**A Pulse-Based Synchronous Model** We consider here another way to define the synchronous model. There is a logical global clock (denoted *CLOCK*) that can be read by all the processes. This clock produces pulses at instants defined by the integers 0, 1, 2, etc. The behavior of the processes and the channels is governed by the clock. They behave as follows.

- A message sent by a process  $p_i$  to its neighbors  $p_j$  at the beginning of a pulse  $r$  is received and processed by  $p_j$  before the pulse  $r + 1$  starts. Hence, in terms of pulses, the transfer delays are bounded.
- It is assumed that local processing times are negligible with respect to transfer delays. Hence, they are assumed to have a null duration. (Actually, the processing time associated with the reception of a message can be seen as being “absorbed” in the message transfer time.)
- A process sends to a given neighbor at most one message per pulse. Hence, when a new pulse  $r + 1$  is generated, a process  $p_i$  knows that (a) all the messages sent during pulse  $r$  have been received and processed, and (b) all the other processes are at pulse  $r + 1$ .

As we can see, this definition captures the same behavior as the definition given in Sect. 1.1. Only the terminology changes: a pulse is simply a round number. The only difference lies in the writing of algorithms. The clock/pulse-based notation allows synchronous algorithms to be easily expressed with the pattern “**when ... do ...**” (instead of the round-based pattern used in Chap. 2).

A synchronous execution is described in Fig. 9.1, where time flows from left to right and space is from top to bottom. In this space-time diagram, there are three processes  $p_1$ ,  $p_2$ , and  $p_3$ . During the first pulse (round),  $p_1$  sends a message only to  $p_2$ , and  $p_2$  and  $p_3$  send a message to each other. Each of the other rounds has a specific message exchange pattern.

**A Synchronous Breadth-First Traversal Algorithm** To illustrate the synchronous model expressed with pulses, let us consider the breadth-first traversal problem. The channels are bidirectional, and the communication graph is connected.

Let  $p_a$  be the process that launches the traversal. As in the previous chapter,  $channel_i[1..c_i]$  denotes the array that defines the local addresses of the  $c_i$  channels of a process  $p_i$ . Moreover, each process has two local variables, denoted  $level_i$  and  $parent_i$ .

```

when CLOCK is increased do
% a new pulse (round) is generated by the synchronous system %
  if (leveli = CLOCK) then
    for each x ∈ {1, . . . , ci} do send LEVEL(leveli) on channeli[x] end for
  end if.

when LEVEL(ℓ) is received on channeli[x] do
  if (fatheri = ⊥) then leveli ← ℓ + 1; parenti ← x end if.
    
```

Fig. 9.2 Synchronous breadth-first traversal algorithm (code for  $p_i$ )

- The initial values of the variables  $level_i$ ,  $1 \leq i \leq n$ , are such that  $level_a = 0$ , and  $level_i = +\infty$  for  $i \neq a$ . At the end of the algorithm,  $level_i$  contains the level of  $p_i$  (i.e., its distance to  $p_a$ ).
- The initial values of the variables  $parent_i$ ,  $1 \leq i \leq n$ , are such that  $parent_a = a$ , and  $parent_i = \perp$  for  $i \neq a$ . At the end of the algorithm,  $parent_i$  contains the index of the channel connecting  $p_i$  to its parent in the tree rooted at  $p_a$ .

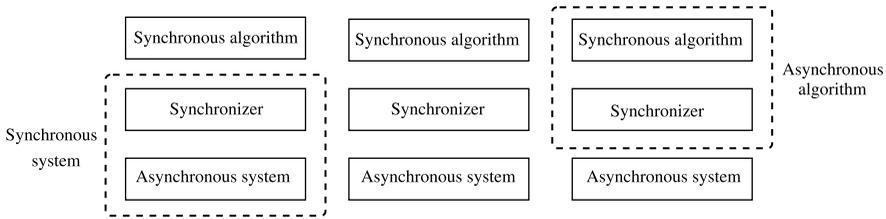
The corresponding synchronous algorithm is described in Fig. 9.2. It is particularly simple. When the clock is set to 0, the algorithm starts, and  $p_a$  sends the message LEVEL(0) to all its neighbors which discover they are at distance 1 from  $p_a$  by the end of the first pulse. Then, when the next pulse is generated ( $CLOCK = 1$ ), each of them sends the message LEVEL(1) to all its neighbors, etc. Moreover, the first message received by a process defines its parent in the breadth-first tree rooted at  $p_a$ . The global synchronization provided by the model ensures that the first LEVEL() message received by a process has followed a path whose number of channels is the distance from the root to that process.

It is easy to see that the time complexity is equal to the eccentricity of  $p_a$  (maximal distance from  $p_a$  to any other process), and the message complexity is equal to  $O(e)$ , the number of communication channels. (Let us observe that  $(n - 1)$  messages can be saved by preventing a process from sending a message to its parent.)

### 9.1.2 Asynchronous Systems and Synchronizers

**Asynchronous Systems** As we saw in Chap. 1, an asynchronous distributed system is a time-free system in the sense that there is no notion of an *external* time that would be given a priori, and could be used in the algorithms executed by the processes. In an asynchronous system, the progress of time is related only to the sequence of operations executed by each process and the flow of messages that they exchange.

To simplify the presentation, it is assumed that all the channels are FIFO channels. (If the channels are not FIFO, sequence numbers can be used to provide a communication layer where channels are FIFO.)



**Fig. 9.3** Synchronizer: from asynchrony to logical synchrony

**Synchronizer** A synchronizer is a distributed asynchronous algorithm that simulates a synchronous system on top of an asynchronous system, as shown in Fig. 9.3. This concept was introduced by B. Awerbuch (1985).

Hence, if we consider a synchronizer and an asynchronous system, we obtain a simulator of a synchronous system on which synchronous algorithms can be executed (left part of Fig. 9.3). Conversely, if we consider a synchronous algorithm and a synchronizer, we obtain an asynchronous algorithm that can be executed on top of an asynchronous system (as depicted on the right part of Fig. 9.3). It is important to see that the simulation has to be general in the sense it has not to depend on the specific synchronous algorithms that will be executed on top it. A synchronizer is consequently an interpreter for synchronous algorithms.

Thus, the design of a synchronizer consists of developing a simulation technique enabling users to design distributed algorithms as they were intended to be executed on a synchronous system (as defined previously). To that end, a synchronizer has to ensure the following:

- Implement a local variable  $clock_i$  on each process  $p_i$  such that the set of logical clocks  $clock_1, \dots,$  and  $clock_n,$  behave as if there was a single read-only clock  $CLOCK$ .
- Ensure that each message sent at pulse (round)  $r$  is received and processed at the very same pulse  $r$  by its destination process.

### 9.1.3 On the Efficiency Side

**Time and Message Costs** An important question concerns the cost added by a synchronizer to simulate a given synchronous algorithm.  $A_s$  being a synchronous algorithm, let  $T_s(A_s)$  and  $M_s(A_s)$  be its complexities in time and in number of messages (when executed in a synchronous system).

As we have just seen, a synchronizer  $\Sigma$  generates a sequence of pulses on each process in such a way that all the processes are simultaneously (with respect to a logical time framework) at the same pulse at the same time.

- The simulation of a pulse by  $\Sigma$  requires  $M_\Sigma^{pulse}$  messages and  $T_\Sigma^{pulse}$  time units.
- Moreover, the initialization of  $\Sigma$  requires  $M_\Sigma^{init}$  messages and  $T_\Sigma^{init}$  time units.

These values allow for the computation of the time and message complexities, denoted  $T_{as}(A_s)$  and  $M_{as}(A_s)$ , of the asynchronous algorithm resulting from the execution of  $A_s$  by  $\Sigma$ . More precisely, as  $A_s$  requires  $T_s(A_s)$  pulses, and each pulse costs  $M_\Sigma^{pulse}$  messages and  $T_\Sigma^{pulse}$  time units, we have

$$M_{as}(A_s) = M_s(A_s) + M_\Sigma^{init} + (T_s(A_s) \times M_\Sigma^{pulse}), \quad \text{and}$$

$$T_{as}(A_s) = T_\Sigma^{init} + (T_s(A_s) \times T_\Sigma^{pulse}).$$

A synchronizer  $\Sigma$  will be efficient if  $M_\Sigma^{init}$ ,  $T_\Sigma^{init}$ ,  $M_\Sigma^{pulse}$ , and  $T_\Sigma^{pulse}$  are “reasonably” small. Moreover, these four numbers, which characterize every synchronizer, allow synchronizers to be compared. Of course, given a synchronizer  $\Sigma$ , there is a compromise between these four values and they cannot be improved simultaneously.

**Design Cost** As indicated on the right of Fig. 9.1, combining a synchronous algorithm with a synchronizer gives an asynchronous algorithm. In some cases, such an asynchronous algorithm can “compete” with ad hoc asynchronous algorithms designed to solve the same problem.

Another interest of the synchronizer concept lies in its implementations. Those are based on distributed synchronization techniques, which are general and can be used to solve other distributed computing problems.

## 9.2 Basic Principle for a Synchronizer

### 9.2.1 The Main Problem to Solve

As stated previously, the role of a synchronizer is to

- generate pulses at each process of an asynchronous distributed system as if these pulses were issued by a global clock whose changes of value would be known instantaneously by all the processes, and
- ensure that a message sent at the beginning of a pulse  $r$  is received by its destination process before this process starts pulse  $r + 1$ .

It follows that a sequence of pulses has to satisfy the following property, denoted  $\mathcal{P}$ : A new pulse  $r + 1$  can be generated at a process only after this process has received all the pulse  $r$  messages (of the synchronous algorithm) sent to it by its neighbors.

The crucial issue is that, during a round  $r$ , a process does not know which of its neighbors sent it a message (see Fig. 9.1). At the level of the underlying asynchronous system, transit delays are finite but not bounded, and consequently the solution for a process to wait during a “long enough” period of time cannot work.

## 9.2.2 Principle of the Solutions

Solving the previous issue requires adding synchronization among neighboring processes. To that end, let us introduce the notion of a safe process.

**Notion of a Safe Process** A process  $p_i$  is *safe* with respect to a pulse  $r$  if all the messages it has sent to its neighbors at the beginning of this pulse have been received by their destination process.

Let us observe that, using acknowledgment messages, it is easy for a process  $p_i$  to learn when it becomes safe with respect to a given pulse  $r$ . This occurs when it has received the acknowledgments associated with the messages it sent during the pulse  $r$ . With respect to the synchronous algorithm, this doubles the number of messages but does not change its order of complexity. Moreover, a process that sends no messages during a pulse is safe (at no cost) since the beginning of the corresponding round.

**From Safe Processes to the Property  $\mathcal{P}$**  The notion of a safe process can be used to ensure the property  $\mathcal{P}$  as follows: the local module implementing the synchronizer at a process  $p_i$  can generate a new pulse ( $r + 1$ ) at that process when (a)  $p_i$  has carried on all its processing of pulse  $r$  and (b) learned that each of its neighbors is safe with respect to the pulse  $r$ .

The synchronizers, which are presented below, differ in the way they deliver to each process  $p_i$  the information “your neighbor  $p_j$  is safe with respect to the current pulse  $r$ ”.

## 9.3 Basic Synchronizers: $\alpha$ and $\beta$

Both the synchronizers  $\alpha$  and  $\beta$  are due to B. Awerbuch (1985).

### 9.3.1 Synchronizer $\alpha$

**Principle of the Synchronizer  $\alpha$**  The underlying principle is very simple. When, thanks to the acknowledgment messages, a process  $p_i$  learns that it is safe with respect to its current pulse, it indicates this to its neighbors by sending them a control message denoted SAFE(). So, when a process (a) has terminated its actions with respect to a pulse  $r$  and (b) learned that its neighbors are safe with respect this pulse  $r$ , the local synchronizer module can generate the pulse  $r + 1$  at  $p_i$ . This is because there is no longer any message related to a pulse  $r' \leq r$ , which is in transit on a channel incident to  $p_i$ .

**Complexities** For the complexities  $M_\alpha^{pulse}$  and  $T_\alpha^{pulse}$ , we have the following, where  $e$  is the number of channels of the communication graph. At each pulse, every application message is acknowledged, and each process  $p_i$  informs its  $c_i$  neighbors that it is safe. Hence  $M_\alpha^{pulse} = O(e)$ , i.e.  $M_\alpha^{pulse} \leq O(n^2)$ . As far the time complexity is concerned, let us observe that control messages are sent only between neighbors, hence  $T_\alpha^{pulse} = O(1)$ .

For the complexities  $M_\alpha^{init}$  and  $T_\alpha^{init}$ , it follows from the fact that there is no specific initialization part that we have  $M_\alpha^{init} = T_\alpha^{init} = 0$ .

**Messages Used by the Synchronizer  $\alpha$**  The local modules implementing the synchronizer  $\alpha$  exchange three types of message.

- An acknowledgment message is denoted ACK().
- A control message denoted SAFE() is sent by a process  $p_i$  to indicate that it is safe with respect to its last pulse.
- Each application message  $m$  sent by a synchronous application process to one of its neighbors is encapsulated in a simulation message denoted MSG( $m$ ).

**Local Variables of the Synchronizer  $\alpha$**  The local variable  $clock_i$  is built by the synchronizer and its value can only be read by the local synchronous application process.

The local variables  $channels_i$  and  $channel_i[1..c_i]$  are defined by the communication graph of the synchronous algorithm. As in previous algorithms,  $channels_i$  is the set  $\{1, \dots, c_i\}$  (which consists of the indexes of the  $c_i$  channels of  $p_i$ ), and for each  $x \in channels_i$ ,  $channel_i[x]$  denotes locally at  $p_i$  the corresponding channel.

The other two local variables, which are hidden to the upper layer, are used only to implement the required synchronization. These variables are the following:

- $expected\_ack_i$  contains the number of acknowledgments that  $p_i$  has still to receive before becoming safe with respect to the current round.
- $neighbors\_safe_i$  is a multiset (also called a bag) that captures the current perception of  $p_i$  on which of its neighbors are safe. Initially,  $neighbors\_safe_i$  is empty.

A multiset is a set that can contain several times some of its elements. As we about to see, it is possible that a process  $p_i$  receives several messages SAFE() (each corresponding to a distinct pulse  $r, r + 1$ , etc.), from a neighbor  $p_j$  while it  $p_i$  still at pulse  $r$ . Hence, the use of a multiset.

**On the Wait Statement** It is assumed that the code of all the synchronizers cannot be interrupted except in a **wait** statement. This means that a process receives and processes a message (MSG(), ACK(), or SAFE()) only when it executes line 5 or line 7 of Fig. 9.4 when considering the synchronizer  $\alpha$ . The same holds for the other synchronizers.

**Algorithm of the Synchronizer  $\alpha$**  The algorithm associated with the local module implementing  $\alpha$  at a process  $p_i$  is described in Fig. 9.4. When a process starts its next pulse (line 1), it first sends the messages MSG( $m$ ) (if any), which correspond to

```

repeat
(1)  $clock_i \leftarrow clock_i + 1$ ; % next pulse is generated %
(2) Send the messages MSG() of the current pulse of the local synchronous algorithm;
(3)  $expected\_ack_i \leftarrow$  number of MSG( $m$ ) sent during the current pulse;
(4)  $neighbors\_safe_i \leftarrow neighbors\_safe_i \setminus channels_i$ ;
(5) wait ( $expected\_ack_i = 0$ ); %  $p_i$  is safe respect to the current pulse %
(6) for each  $x \in channels_i$  do send SAFE() on  $channel_i[x]$  end for;
(7) wait ( $channels_i \subseteq neighbors\_safe_i$ ) % The neighbors of  $p_i$  are safe %
    %  $p_i$  has received all the messages MSG() sent to it during pulse  $clock_i$  %
until the last local pulse has been executed end repeat.

when MSG( $m$ ) is received on  $channel_i[x]$  do
(8) send ACK() on  $channel_i[x]$ ;
(9) if ( $x \notin neighbors\_safe_i$ )
(10) then  $m$  belongs to the current pulse; deliver it to the synchronous algorithm
(11) else  $m$  belongs to the next pulse; keep it to deliver it at the next pulse
(12) end if.

when ACK() is received on  $channel_i[x]$  do
(13)  $expected\_ack_i \leftarrow expected\_ack_i - 1$ .

when SAFE() is received on  $channel_i[x]$  do
(14)  $neighbors\_safe_i \leftarrow neighbors\_safe_i \cup \{x\}$ .

```

**Fig. 9.4** Synchronizer  $\alpha$  (code for  $p_i$ )

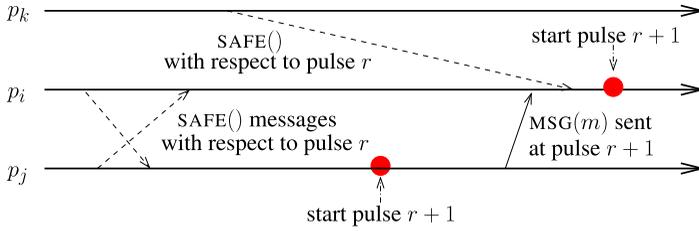
the application messages  $m$  that the local synchronous algorithm must send at pulse  $clock_i$  (line 2).

Then  $p_i$  initializes appropriately  $expected\_ack_i$  (line 3), and  $neighbors\_safe_i$  (line 4). As  $neighbors\_safe_i$  is a multiset, its update consists in suppressing one copy of each channel index. Process  $p_i$  then waits until it has become safe (line 5), and when this happens, it sends a message SAFE() to each of its neighbors to inform them (line 6). Finally, it waits until all its neighbors are safe before proceeding to the next pulse (line 7).

When, it receives a message ACK() or SAFE(), a process  $p_i$  updates the corresponding local variable  $expected\_ack_i$  (line 13), or  $neighbors\_safe_i$  (line 14).

Let us remark that, due to the control messages SAFE() exchanged by neighbor processes, a message MSG( $m$ ) sent at pulse  $r$  by a process  $p_j$  to a process  $p_i$  will arrive before  $p_i$  starts pulse  $(r + 1)$ . This is because  $p_i$  must learn that  $p_j$  is safe with respect to pulse  $r$  before being allowed to proceed to pulse  $(r + 1)$ . But a message MSG( $m$ ) sent at pulse  $r'$  to a process  $p_i$  can arrive before  $p_i$  starts pulse  $r'$ . This is depicted on Fig. 9.5, where  $p_j$  and  $p_k$  are two neighbors of  $p_i$ , and where  $r' = r + 1$  and  $p_i$  receives a pulse  $(r + 1)$  message while it is still at pulse  $r$ .

Moreover, let us benefit from this figure to consider the case where  $p_j$  does not send application messages during pulse  $(r + 1)$ . In Fig. 9.5, the message MSG() sent by  $p_j$  is consequently replaced by the message SAFE(). In that case,  $neighbors\_safe_i$  contains twice the local index of the channel connecting  $p_j$  to  $p_i$ . This explains why  $neighbors\_safe_i$  has to be a multiset.



**Fig. 9.5** Synchronizer  $\alpha$ : possible message arrival at process  $p_i$

When it receives a message  $MSG(m)$  on a channel  $channel_i[x]$  (which connects it to its neighbor  $p_j$ ), a process  $p_i$  first sends back an  $ACK()$  message (line 8). According to the previous observation (Fig. 9.5), the behavior of  $p_i$  depends then on the current value of  $neighbors\_safe_i$  (line 9). There are two cases.

- The channel index  $x \notin neighbors\_safe_i$  (line 10). In that case,  $p_i$  has not received from  $p_j$  the  $SAFE()$  message that closes pulse  $r$ . Hence, the message  $MSG(m)$  is associated with pulse  $r$ , and consequently,  $p_i$  delivers  $m$  it to the upper layer local synchronous algorithm.
- The channel index  $x \in neighbors\_safe_i$  (line 11). This means that  $p_i$  has already received the message  $SAFE()$  from  $p_j$  concerning the current pulse  $r$ . Hence,  $m$  is a message sent at pulse  $r + 1$ . Consequently,  $p_i$  has to store the message  $m$ , and delivers it during pulse  $r + 1$  (after it has sent its messages associated with pulse  $r + 1$ ).

### 9.3.2 Synchronizer $\beta$

**Principle of the Synchronizer  $\beta$**  The synchronizer  $\beta$  is based on a spanning tree rooted at some process  $p_a$ . The construction of this tree has to be done in an initialization stage. This tree is used to convey the control messages  $SAFE()$  from the leaves to the root, and (new) control messages  $PULSE()$  from the root to the leaves.

As soon as a process that is a leaf in the control tree is safe, it indicates this to its parent in the tree (messages  $SAFE()$ ). A non-leaf process waits until it and its children are safe before informing its parent of this. When the root learns that all the processes are safe with respect to the current round, it sends a message  $PULSE()$ , which is propagated to all the processes along the channels of the tree.

Let us notice that, when the root learns that all the processes are safe, no message  $MSG(m)$  is in transit in the system.

**Complexities** Every message  $MSG(m)$  gives rise to an acknowledgment message, and at every pulse, two control messages ( $SAFE()$  and  $PULSE()$ ) are sent on each channel of the tree. Moreover, the height of the tree rooted at a process  $p_a$  is equal to its eccentricity  $ecc_a$ , which is at most  $(n - 1)$ . Hence, we have  $M_\beta^{pulse} = T_\beta^{pulse} = O(n)$ .

If a rooted tree pre-exists, we have  $M_\beta^{init} = T_\beta^{init} = 0$ . If the tree has to be built,  $M_\beta^{init}$  and  $T_\beta^{init}$  are the costs of building a spanning tree (see Chap. 1).

**Local Variables of the Synchronizer  $\beta$**  As before, each process  $p_i$  has  $c_i$  neighbors with which it can communicate through the channels  $channel_i[1..c_i]$ . The spanning tree is implemented with the following local variables at each process  $p_i$ .

- $channel_i[parent_i]$  denotes the channel connecting  $p_i$  to its parent in the tree. Moreover, the root  $p_a$  has an additional channel index  $parent_a$  such that  $channel_a[parent_a] = \perp$ .
- $children_i$  is a set containing the indexes of the channels connecting process  $p_i$  to its children. If  $children_i = \emptyset$ ,  $p_i$  is a leaf of the tree.

Finally, as the messages `SAFE()` are sent only from a process to its parents, the multiset  $neighbors\_safe_i$  (used in the synchronizer  $\alpha$ ) is replaced by a local variable denoted  $children\_safe_i$ . Since a process  $p_i$  waits for messages `SAFE()` only from its children in the control tree, this variable is no longer required to be a multiset. The set  $children\_safe_i$  is initially empty.

**Algorithm of the Synchronizer  $\beta$**  The behavior of the synchronizer  $\beta$  is described in Fig. 9.6. The root sends a message `PULSE()` to its neighbors, and this message is propagated along the tree to all the processes (lines 1–3). Then,  $p_i$  sends (if any) its messages related to pulse  $clock_i$  of its local synchronous algorithm (line 4), and resets  $expected\_ack_i$  to the number of these messages (line 6).

After it has sent the messages of the current pulse (if any),  $p_i$  waits until both itself and its children are safe (line 7). When this happens, it sends a message `SAFE()` to its parent to inform it that the subtree it controls is safe (line 8). Process  $p_i$  also resets  $children\_safe_i$  to  $\emptyset$  (line 9). In this way, all the local variables  $children\_safe_i$  are equal to their initial value ( $\emptyset$ ) when the root triggers the next pulse. If, while  $p_i$  is waiting at line 7, or after it has sent the message `SAFE()` to its parent,  $p_i$  receives messages `MSG(m)` from some neighbor,  $p_i$  processes them and this lasts until it enters the next pulse at line 2.

This last point is one where synchronizer  $\beta$  differs from synchronizer  $\alpha$ . In  $\alpha$ , a process  $p_i$  learns locally when its neighbors are safe. In the synchronizer  $\beta$ , a process learns locally this only from its children (which are a subset of its neighbors). It learns that all its neighbors are safe only when it receives a message `PULSE()` which carries the global information that all the processes are safe with respect to pulse  $clock_i$ . This means that the message pattern depicted in Fig. 9.7 can happen with synchronizer  $\beta$ , while it cannot with synchronizer  $\alpha$ .

Moreover, a message `MSG(m)` sent at pulse  $r$  can arrive at a process  $p_i$  while it is still at pulse  $clock_i = r - 1$ . This is due to the fact that messages `PULSE()` are not sent between each pair of neighbors, but only along the channel spanning tree. This pattern is described in Fig. 9.8. A simple way to allow the destination process  $p_i$  to know if a given message `MSG(m)` is related to pulse  $clock_i$  or pulse  $clock_i + 1$  consists in associating a sequence number to these messages. Actually, as the channels are FIFO, a simple parity bit (denoted  $pb$  in Fig. 9.6) is sufficient to disambiguate messages.

```

repeat
(1) if ( $channel_i[parent_i] \neq \perp$ ) then wait (PULSE() received on  $channel_i[parent_i]$ ) end if;
    %  $p_i$  and all its neighbors are safe with respect to the pulse  $clock_i$ : %
    % it has received all the messages MSG() sent to it during pulse  $clock_i$  %
(2)  $clock_i \leftarrow clock_i + 1$ ; % next pulse is generated %
(3) for each  $x \in children_i$  do send PULSE() on  $channel_i[x]$  end for;
(4) Send the messages MSG( $-, pb$ ) of the local synchronous algorithm
(5)                               where  $pb = (clock_i \bmod 2)$ ;
(6)  $expected\_ack_i \leftarrow$  number of MSG( $m$ ) sent during the current pulse;
(7) wait ( $(expected\_ack_i = 0) \wedge (children\_safe_i = children_i)$ );
    %  $p_i$  and all its children are safe respect to the current pulse %
(8) if ( $channel_i[parent_i] \neq \perp$ ) then send SAFE() on  $channel_i[parent_i]$  end if;
(9)  $children\_safe_i \leftarrow \emptyset$ 
until the last local pulse has been executed end repeat.

when MSG( $m, pb$ ) is received on  $channel_i[x]$  do
(10) send ACK() on  $channel_i[x]$ ;
(11) if ( $pb = (clock_i \bmod 2)$ )
(12)   then  $m$  belongs to the current pulse; deliver it to the synchronous algorithm
(13)   else  $m$  belongs to the next pulse; keep it to deliver it at the next pulse
(14) end if.

when ACK() is received on  $channel_i[x]$  do
(15)  $expected\_ack_i \leftarrow expected\_ack_i - 1$ .

when SAFE() is received on  $channel_i[x]$  do % we have then  $x \in children_i$  %
(16)  $children\_safe_i \leftarrow children\_safe_i \cup \{x\}$ .
    
```

Fig. 9.6 Synchronizer  $\beta$  (code for  $p_i$ )

Fig. 9.7 A message pattern which can occur with synchronizer  $\beta$  (but not with  $\alpha$ ): Case 1

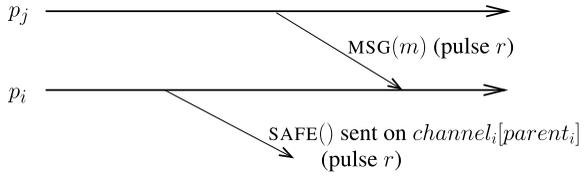
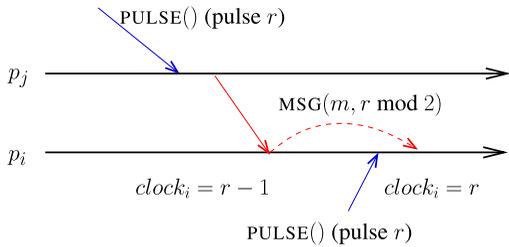
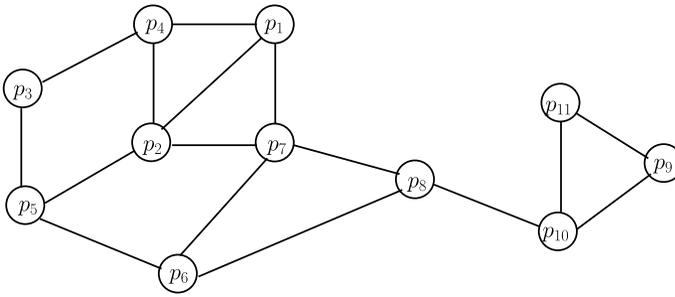


Fig. 9.8 A message pattern which can occur with synchronizer  $\beta$  (but not with  $\alpha$ ): Case 2





**Fig. 9.9** Synchronizer  $\gamma$ : a communication graph

## 9.4 Advanced Synchronizers: $\gamma$ and $\delta$

Both the synchronizers  $\gamma$  and  $\delta$  are generalizations of  $\alpha$  and  $\beta$ . Synchronizer  $\gamma$  is due to B. Awerbuch (1985), while synchronizer  $\delta$  is due to D. Peleg and J.D. Ullman (1989). These synchronizers differ in the totally different ways that they generalize  $\alpha$  and  $\beta$ .

### 9.4.1 Synchronizer $\gamma$

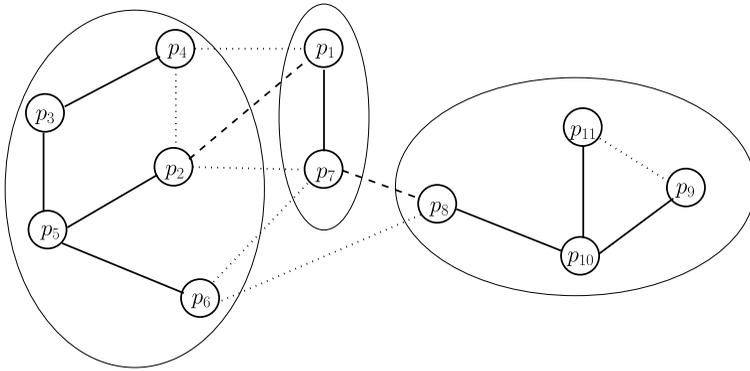
#### Looking for an Overlay Structure: Partitioning the Communication Graph

When looking at complexity, the synchronizers  $\alpha$  and  $\beta$  have opposite performances, namely  $\alpha$  is better than  $\beta$  for time but worse for the number of messages. Synchronizer  $\alpha$  is good for communication graphs with limited degree, while synchronizer  $\beta$  is good for communication graphs with small diameter.

The principle that underlies the design of synchronizer  $\gamma$  is to combine  $\alpha$  and  $\beta$  in order to obtain better time complexity than  $\beta$  and better message complexity than  $\alpha$ . Such a combination relies on a partitioning of the system (realized during the initialization part of  $\gamma$ ).

To fix the idea, let us consider Fig. 9.9 which describes a communication graph with eleven processes. A partitioning of this graph is described in Fig. 9.10, where the communication graph is decomposed into three spanning trees which are interconnected. Each tree is surrounded by an ellipsis, and its corresponding edges (communication channels) are denoted by bold segments. The trees are connected by communication channels denoted by dashed segments. The communication channels which are neither in a tree nor interconnecting two trees are denoted by dotted segments.

While, messages of the synchronous application algorithm can be sent on any communication channel, control messages sent by the local modules of the synchronizer  $\gamma$  are sent only on the (bold) edges of a tree, or on the (dashed) edges that interconnect trees.



**Fig. 9.10** Synchronizer  $\gamma$ : a partitioning

A tree is called a *group*, and we say “intragroup channel” or “intergroup channel”. Hence, when considering Fig. 9.10, the dashed segments are intergroup channels.

**Principle of Synchronizer  $\gamma$**  Synchronizer  $\gamma$  is based on a two-level synchronization mechanism.

- Inside each group, the tree is used (as in  $\beta$ ) to allow the processes of the group to learn that the group is safe (i.e., all its processes are safe).
- Then, each group behaves as if it was a single process and the synchronizer  $\alpha$  is used among the groups to allow each of them to learn that its neighbor groups are safe. When this occurs, a new pulse can be generated inside the corresponding group.

**Local Variables Used by Synchronizer  $\gamma$**  The local variables of a process  $p_i$  related to communication are the following ones.

- As previously,  $channels_i = \{1, \dots, c_i\}$  is the set of local indexes such that, for each  $x \in channels_i$ ,  $channels_i[x]$  locally denotes a communication channel of  $p_i$ . These are all the channels connecting process  $p_i$  to its neighbor processes.
- As for  $\beta$ ,  $channel_i[parent_i]$  denotes the channel connecting  $p_i$  to its parent in the tree associated with its group, and  $children_i$  denotes the set of its children. The root  $p_a$  of a group is the process  $p_a$  such that  $channel_a[parent_a] = \perp$ . These channels are depicted as bold segments in Fig. 9.10.
- The set  $it\_group\_channels_i \subseteq \{1, \dots, c_i\}$ , contains indexes of channels connecting  $p_i$  to processes of other groups. From a control point of view, two neighbor groups are connected by a single channel on which travel (application messages and) control messages. These channels are called *inter-group* channels. These are the channels depicted by dashed segments in Fig. 9.10.

The local variable  $expected\_ack_i$  has the same meaning as in  $\alpha$  or  $\beta$ , and  $children\_safe_i$  has the same meaning as in  $\beta$ , while  $neighbors\_safe_i$  is a multiset (initialized to  $\emptyset$ ) which has the same meaning as in  $\alpha$ .

**Messages Used by Synchronizer  $\gamma$**  In addition to the messages `MSG()`, `ACK()` used as in  $\alpha$  or  $\beta$ , and the messages `SAFE()` used as in  $\beta$  (i.e., used inside a group and sent along the tree to inform the root), processes exchange messages `GROUP_SAFE()` and `ALL_GROUP_SAFE()`.

A message `GROUP_SAFE()` is sent by a process  $p_i$  to its children, and to its neighbors which belong to other groups. These messages are used to inform their destination processes that all the processes of the group to which the sender belongs are safe. A message `ALL_GROUP_SAFE()` is used inside a group (from the leaves of the spanning tree until its root), to inform its processes that its neighbor groups are safe.

**Algorithm of the Synchronizer  $\gamma$**  The algorithm implementing synchronizer  $\gamma$  at a process  $p_i$  is described in Fig. 9.11.

Inside each group, the root process first sends a message `PULSE()` that is propagated along the channels of the tree associated to this group (lines 1–3). After it has received a message `PULSE()`, a process  $p_i$  starts interpreting the current pulse of the synchronous algorithm by sending the corresponding application messages (line 4). As we have seen with the tree-based synchronizer  $\beta$ , these messages carry the parity of the corresponding pulse so that they are processed at the correct pulse (lines 23–26). Then,  $p_i$  initializes `expected_acki` to its new value (line 6), and `neighborsi_safei` similarly to what is done in the synchronizer  $\alpha$  (line 7).

After these statements,  $p_i$  waits until it and its children are safe (hence, all the processes of its group in the subtree for which it is the root are safe line 8). When this occurs,  $p_i$  informs its parent by sending it the message `SAFE()` (line 10). If it is root,  $p_i$  learns that all the processes in its group are safe. It consequently sends a message `GROUP_SAFE()` (line 12–14) to its children and the processes of the other groups to which it is connected with a channel in `it_group_channelsi`. These messages `GROUP_SAFE()` are propagated to all the processes of the group, and along the channels in `it_group_channelsi` (lines 12–14 and 29–31). Moreover, when a process  $p_i$  receives a message `GROUP_SAFE()` from a process in another group (this occurs on a channel in `it_group_channelsi`), it updates `neighbors_safei` accordingly (line 32).

Then,  $p_i$  waits until it knows that all its neighbor groups are safe with respect to the current pulse `clocki` (line 16). When this occurs, the corresponding information will propagate inside a group from the leaves of the spanning tree to its root (lines 17 and 18–20). (Let us observe that the waiting predicate of line 17 is trivially satisfied at a leaf of the tree.) When the root learns that the processes in all groups are safe with respect the current round, it starts the next pulse inside its groups.

**Two Particular (Extreme) Cases** If each process constitutes a group, the spanning trees inside each group, the tree-related variables (`parenti`, `childreni`), and the tree-related messages (`GROUP_SAFE()`) disappear. Moreover, we have then `it_group_channelsi = channelsi`. The parity can also be suppressed. It follows that we can suppress lines 1, 3, 9–11, 15, 17–21, 28, and 29–31. As the reader can check, we then obtain the synchronizer  $\alpha$  (where the messages `SAFE()` are replaced by `GROUP_SAFE()`).

```

repeat
(1) if ( $channel_i[parent_i] \neq \perp$ ) then wait (PULSE() received on  $channel_i[parent_i]$ ) end if;
    % all processes are safe with respect to pulse  $clock_i$  %
(2)  $clock_i \leftarrow clock_i + 1$ ; % next pulse is generated %
(3) for each  $x \in children_i$  do send PULSE() on  $channel_i[x]$  end for;
(4) Send the messages MSG( $-, pb$ ) of the local synchronous algorithm
(5) where  $pb = (clock_i \bmod 2)$ ;
(6)  $expected\_ack_i \leftarrow$  number of MSG( $m$ ) sent during the current pulse;
(7)  $neighbors\_safe_i \leftarrow neighbors\_safe_i \setminus (it\_group\_channels_i \cup children_i)$ ;
(8) wait ( $(expected\_ack_i = 0) \wedge (children\_safe_i = children_i)$ );
    %  $p_i$  and all its children are safe respect to the current pulse %
(9) if ( $channel_i[parent_i] \neq \perp$ )
(10) then send SAFE() on  $channel_i[parent_i]$ 
(11) else %  $p_i$  is the root of its group %
(12) for each  $x \in children_i \cup it\_group\_channels_i$ 
(13) do send GROUP_SAFE() on  $channel_i[x]$ 
(14) end for
(15) end if;
(16) wait ( $it\_group\_channels_i \subseteq neighbors\_safe_i$ );
    %  $p_i$ 's group and its neighbor groups are safe with respect to pulse  $clock_i$  %
(17) wait ( $\forall x \in children_i$ : ALL_GROUPS_SAFE() received on  $channels_i[x]$ );
(18) if ( $channel_i[parent_i] \neq \perp$ )
(19) then send ALL_GROUPS_SAFE() on  $channel_i[parent_i]$ 
(20) end if;
(21)  $children\_safe_i \leftarrow \emptyset$ 
until the last local pulse has been executed end repeat.

when MSG( $m, pb$ ) is received on  $channel_i[x]$  do
(22) send ACK() on  $channel_i[x]$ ;
(23) if ( $pb = (clock_i \bmod 2)$ )
(24) then  $m$  belongs to the current pulse; deliver it to the synchronous algorithm
(25) else  $m$  belongs to the next pulse; keep it to deliver it at the next pulse
(26) end if.

when ACK() is received on  $channel_i[x]$  do
(27)  $expected\_ack_i \leftarrow expected\_ack_i - 1$ .

when SAFE() is received on  $children_i[x]$  do % we have then  $x \in children_i$  %
(28)  $children\_safe_i \leftarrow children\_safe_i \cup \{x\}$ .

when GROUP_SAFE() is received on  $channel_i[parent_i]$  do
(29) for each  $x \in children_i \cup it\_group\_channels_i$ 
(30) do send GROUP_SAFE() on  $channel_i[x]$ 
(31) end for.

when GROUP_SAFE() is received on  $channel_i[x]$  where  $x \in it\_group\_channels_i$  do
(32)  $neighbors\_safe_i \leftarrow neighbors\_safe_i \cup \{x\}$ .

```

**Fig. 9.11** Synchronizer  $\gamma$  (code for  $p_i$ )

In the other extreme case, there is a single group to which all the processes belong. In this case, both  $it\_group\_channels_i$ , which is now equal to  $\emptyset$ , and  $neighbors\_safe_i$  become useless and disappear. Similarly, the messages GROUP\_SAFE() and ALL\_GROUP\_SAFE() become useless. It follows that we can suppress lines 7, 11–14, 16–20, 29–31, and 32, and synchronizer  $\beta$ .

**Complexities** Let  $E_p$  be the channels on which control messages other than  $\text{ACK}()$  travel (those are the channels of the spanning trees and the intergroup channels). We do not consider the messages  $\text{ACK}()$  because they are used in all synchronizers. Let  $H_p$  be the maximum height of a spanning tree.

At most four message are sent on each channel of  $E_p$ , namely  $\text{PULSE}()$ ,  $\text{SAFE}()$ ,  $\text{GROUP\_SAFE}()$ , and  $\text{ALL\_GROUP\_SAFE}()$ . It follows that we have  $C_Y^{\text{pulse}} = O(|E_p|)$  and  $T_Y^{\text{pulse}} = O(|H_p|)$ . It is possible to find partitions such that  $C_Y^{\text{pulse}} \leq kn$  and  $T_Y^{\text{pulse}} \leq \log_k n$ , where  $1 \leq k \leq n$ .

More generally, according to the partition that is initially built, we have  $O(n) \leq C_Y^{\text{pulse}} \leq O(e) \leq O(n^2)$  (where  $e$  is the number of communication channels) and  $O(1) \leq T_Y^{\text{pulse}} \leq O(n)$ .

## 9.4.2 Synchronizer $\delta$

**Graph Spanner** Let us first recall that a *partial* graph is obtained by suppressing edges, while a *subgraph* is obtained by suppressing vertices and their incident edges.

Given a connected undirected graph  $G = (V, E)$  ( $V$  is the set of vertices and  $E$  the set of edges), a partial subgraph  $G' = (V, E')$  is a *t-spanner* if, for any edge  $(x, y) \in E$ , there is path (in  $G'$ ) from the vertex  $x$  to the vertex  $y$  whose distance (number of channels) is at most  $t$ .

The notion of a graph spanner generalizes the notion of a spanning tree. It is used in distributed message-passing distributed systems to define overlay structures with appropriate distance properties.

**Principle of Synchronizer  $\delta$**  This synchronizer assumes that a *t-spanner* has been built on the communication graph defined by the synchronous algorithm. Its principle is simple. When a process becomes safe, it executes  $t$  communication phases with its neighbors in the *t-spanner*, at the end of which it will know that all its neighbors in the communication graph are safe.

From an operational point of view, let us consider a process  $p_i$  that becomes safe. It sets a local variable  $ph_i$  to 0, sends a message  $\text{SAFE}()$  to its neighbors in the *t-spanner*, and waits such a message from each of its neighbors in the *t-spanner*. When, it has received these messages, it increases  $ph_i$  to  $ph_i + 1$  and re-executes the previous communication pattern. After this has been repeated  $t$  times,  $p_i$  locally generates its next pulse.

**Theorem 12** For all  $k \in [0..t]$  and every process  $p_i$ , when  $ph_i$  is set to  $k$ , the processes at distance  $d \leq k$  from  $p_i$  in the communication graph are safe.

*Proof* The proof is by induction. Let us observe that the invariant is true for  $k = 0$  ( $p_i$  is safe when it sets  $ph_i$  to 0). Let us assume that the invariant is satisfied up to  $k$ . When  $p_i$  increases its counter to  $k + 1$ , it has received  $(k + 1)$  messages  $\text{SAFE}()$

```

repeat
  %  $p_i$  and all its neighbors are safe with respect to the pulse  $clock_i$  %
  (1)  $clock_i \leftarrow clock_i + 1$ ; % next pulse is generated %
  (2) Send the messages  $MSG(-, pb)$  of the local synchronous algorithm
  (3)                               where  $pb = (clock_i \bmod 2)$ ;
  (4)  $expected\_ack_i \leftarrow$  number of  $MSG(m)$  sent during the current pulse;
  (5) wait ( $expected\_ack_i = 0$ );
  %  $p_i$  and all its children are safe respect to the current pulse %
  (6)  $ph_i \leftarrow 0$ ;
  (7) repeat  $t\_neighbors\_safe_i \leftarrow t\_neighbors\_safe_i \setminus spanner\_channels_i$ ;
  (8)   for each  $x \in spanner\_channels_i$  do send  $SAFE()$  on  $channel_i[x]$  end for;
  (9)   wait ( $spanner\_channels_i \subseteq t\_neighbors\_safe_i$ );
  (10)   $ph_i \leftarrow ph_i + 1$ 
  (11) until ( $ph_i = t$ ) end repeat
until the last local pulse has been executed end repeat.

when  $MSG(m, pb)$  is received on  $channel_i[x]$  do
  (12) send  $ACK()$  on  $channel_i[x]$ ;
  (13) if ( $pb = (clock_i \bmod 2)$ )
  (14)   then  $m$  belongs to the current pulse; deliver it to the synchronous algorithm
  (15)   else  $m$  belongs to the next pulse; keep it to deliver it at the next pulse
  (16) end if.

when  $ACK()$  is received on  $channel_i[x]$  do
  (17)  $expected\_ack_i \leftarrow expected\_ack_i - 1$ .

when  $SAFE()$  is received on  $channel_i[x]$  do % we have then  $x \in spanner\_channels_i$  %
  (18)  $t\_neighbors\_safe_i \leftarrow t\_neighbors\_safe_i \cup \{x\}$ .

```

**Fig. 9.12** Synchronizer  $\delta$  (code for  $p_i$ )

from each of its neighbors in the  $t$ -spanner. Let  $p_j$  be one of these neighbors. When  $p_j$  sent its  $(k + 1)$ th message  $SAFE()$  to  $p_i$ , we had  $ph_j = k$ . It follows from the induction assumption that the processes at distance  $d \leq k$  from  $p_j$  (in the communication graph) are safe, and this applies to every neighbor of  $p_i$  in the  $t$ -spanner. So, when  $p_i$  increases its counter to  $k + 1$ , all its neighbors in the communication graph at a distance  $d \leq k + 1$  are safe.  $\square$

**Algorithm of Synchronizer  $\delta$**  The behavior of a process  $p_i$  is described in Fig. 9.12. The local set  $spanner\_channels_i$  contains the indexes of the channels of  $p_i$  that belong to the  $t$ -spanner. The local variable  $t\_neighbors\_safe_i$  (which is initially empty) is a multiset whose role is to contain indexes of  $t$ -spanner channels on which messages  $SAFE()$  have been received.

**Complexities** Let  $m$  be the number of channels in the  $t$ -spanner. It is easy to see that  $C_\gamma^{pulse} = O(t)$  and  $T_\gamma^{pulse} = O(mt)$ .

It is easy to see that the case  $t = 1$  corresponds to the synchronizer  $\alpha$ . If the  $t$ -spanner is a spanning tree, we have  $m = n - 1$  and  $t \leq D$  (let us recall that  $D$

```

when INIT() received do
  if ( $\neg done_i$ ) then
     $done_i \leftarrow true$ ; set  $timer_i$  to 0;
    for each  $x \in neighbors_i$  do send INIT() on  $channel_i[x]$  end for
  end if.

```

**Fig. 9.13** Initialization of physical clocks (code for  $p_i$ )

is the diameter of the communication graph), and we have  $C_Y^{pulse} = O(D)$  and  $T_Y^{pulse} = O(nD)$ .

## 9.5 The Case of Networks with Bounded Delays

While synchronous and asynchronous algorithms are two extreme points of the synchronous behavior spectrum, there are distributed systems that are neither fully synchronous nor entirely asynchronous. This part of this chapter is devoted to such a type of system, and the construction of synchronizers, which benefit from its specific properties, is used to illustrate their noteworthy properties. Both synchronizers presented here are due to C. Y. Chou, I. Cidon, I. Gopal, and S. Zaks (1987).

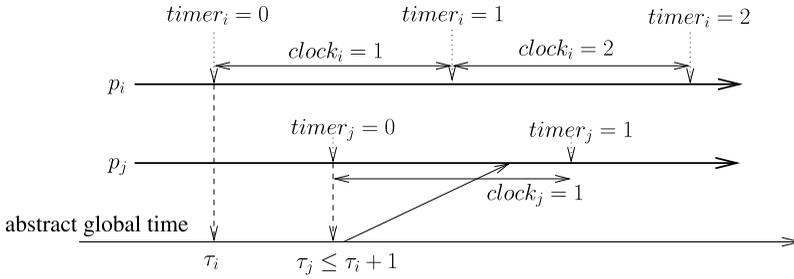
### 9.5.1 Context and Hypotheses

**Bounded Delay Networks** These networks are systems in which (a) communication delays are bounded (by one time unit), (b) each process (processor) has a physical clock, (c) the clocks progress to same speed but are not synchronized, and (d) processing times are negligible when compared to message delay and are consequently assumed to be equal to 0.

Thus, the local clocks do not necessarily show the same time at the same moment but advance by equal amounts in equal time intervals. If the clocks could be started simultaneously, we would obtain a perfectly synchronous system. The fact that the system is not perfectly synchronous requires the addition of a synchronizer when one wants to execute synchronous algorithms on top of such systems.

**Initialization of the Local Clocks** The initialization (reset to 0) of the local clocks can be easily realized by a network traversal algorithm as described in Fig. 9.13. Each process has a Boolean  $done_i$  initialized to *false*. Its local clock is denoted  $timer_i$ .

At least one process (possibly more) receives an external message INIT(). The first time it receives such a message, a process sets its timer and propagates the message to its neighbors. The local variables  $neighbors_i$  and  $channel_i[neighbors_i]$  have the same meaning as before.



**Fig. 9.14** The scenario to be prevented

Let us consider an abstract global time, and let  $\tau_i$  be the time at which  $p_i$  set  $timer_i$  to 0. This global time, which is not accessible to the processes, can be seen as the time of an omniscient external observer. Its unit is assumed to be the same as the one of the local clocks. The previous initialization provides us with the following relation:

$$\forall(i, j): |\tau_i - \tau_j| \leq d(i, j) \quad (R1),$$

where  $d(i, j)$  is the distance separating  $p_i$  and  $p_j$  (minimal number of channels between  $p_i$  and  $p_j$ ). Let us notice that if  $p_i$  and  $p_j$  are neighbors, we have  $|\tau_i - \tau_j| \leq 1$ .

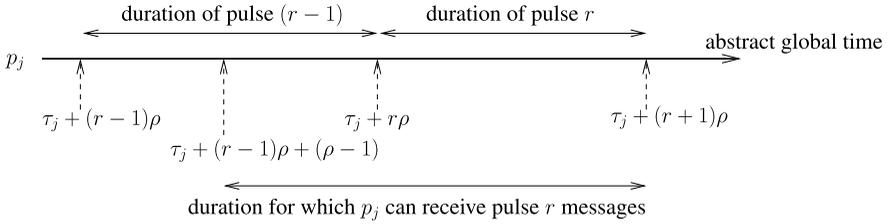
### 9.5.2 The Problem to Solve

After a process  $p_i$  has received a message  $INIT()$ ,  $timer_i$  accurately measures the passage of time, one unit of time being always the maximum transit time for a message.

Let us consider a scenario where, after the initialization of the physical clocks, there is no additional synchronization. This scenario is depicted in Fig. 9.14 where there are two neighbor processes  $p_i$  and  $p_j$ , and a logical pulse takes one physical time unit. At the beginning of its first pulse,  $p_j$  sends a message to  $p_i$ , but this message arrives while  $p_i$  is at its second pulse. Hence, this message arrives too late, and violates the fundamental property of synchronous message-passing.

Hence, synchronizers are not given for free in bounded delay networks. Implementing a synchronizer requires to compute an appropriate duration of  $\rho$  physical time units for a logical pulse (the previous scenario shows that we have necessarily  $\rho > 1$ ). The  $r$ th pulse of  $p_i$  will then start when  $timer_i = r\rho$ , and will terminate when  $timer_i$  reaches the value  $(r + 1)\rho$ .

Several synchronizers can be defined. They differ in the value of  $\rho$  and the instants at which they send messages of the synchronous algorithm they interpret. The next sections present two of them, denoted  $\lambda$  and  $\mu$ .



**Fig. 9.15** Interval during which a process can receive pulse  $r$  messages

### 9.5.3 Synchronizer $\lambda$

In the synchronizer  $\lambda$ , a process  $p_i$  sends a message  $m$  of the synchronous algorithm relative to pulse  $r$  when  $timer_i = \rho r$ . It remains to compute the value of  $\rho$  so that no message is received too late.

**Pulse Duration** Let  $p_j$  be a neighbor of a  $p_i$ , and let  $\tau_j(r)$  be the global instant time at which  $p_j$  receives a pulse  $r$  message  $m$  from  $p_i$ . This message must be received and processed before  $p_j$  enters pulse  $r + 1$ , i.e., we must have

$$\tau_j(r) < \tau_j + (r + 1)\rho \quad (\text{R2}),$$

the left-hand side of this inequality is the abstract global time at which  $p_j$  starts pulse  $r + 1$ . As transfer delays are at most one time unit, we have  $\tau_j(r) < (\tau_i + r\rho) + 1$ . Combined with (R1), namely  $\tau_i < \tau_j + 1$ , we obtain  $\tau_j(r) < \tau_j + r\rho + 2$ , that is to say

$$\tau_j(r) < \tau_j + (r + 1)\rho + (2 - \rho) \quad (\text{R3}).$$

It follows that we have

$$[(\rho \geq 2) \wedge (\text{R3})] \Rightarrow (\text{R2}).$$

This means that the property (R2) required for the correct implementation of a synchronizer (namely, no message arrive too late) is satisfied as soon as  $\rho \geq 2$ .

(R3) gives an upper bound on the global time instant at which a process can receive pulse  $r$  messages sent by its neighbors. In the same way, it is possible to find a lower bound. Let  $p_i$  be the sender of a pulse  $r$  message received by process  $p_j$  at time  $\tau_j(r)$ . We have  $\tau_j(r) \geq \tau_i + r\rho$ . Combining this inequality with (R1) ( $\tau_i \geq \tau_j + 1$ ), we obtain

$$\tau_j(r) \geq \tau_j + r\rho - 1 = \tau_j + (r - 1)\rho + (\rho - 1) \quad (\text{R4}).$$

Hence, the condition  $\rho \geq 2$  ensures that a message sent at pulse  $r$  will be received by its destination process  $p_i$  (a) before  $p_i$  progresses to the pulse  $(r + 1)$ , and (b) after  $p_i$  has started its pulse  $(r - 1)$ . This is illustrated in Fig. 9.15.

```

when  $timer_i = \rho r$  do
     $clock_i \leftarrow r$ ; % next pulse is generated %
    Send the messages  $MSG(-, pb)$  of the local synchronous algorithm where
     $pb = (clock_i \bmod 2)$ ;
    process the pulse  $r$  messages.

when  $MSG(m, pb)$  is received on  $channel_i[x]$  do
    if  $(clock_i \times \rho \leq timer_i < (clock_i + 1)\rho) \wedge (pb = (clock_i \bmod 2))$ 
        then  $m$  belongs to the current pulse; deliver it to the synchronous algorithm
        else  $m$  belongs to the next pulse; keep it to deliver it at the next pulse
    end if.

```

**Fig. 9.16** Synchronizer  $\lambda$  (code for  $p_i$ )

It follows that the pulse  $r$  of a process  $p_i$  spans the global time interval

$$[\tau_i + \rho r, \tau_i + (\rho + 1)r),$$

and during this time interval,  $p_i$  can receive from its neighbors only messages sent at pulse  $r$  or  $r + 1$ . It follows that messages have to carry the parity bit of the pulse at which they are sent so that the receiver be able to know if the received message is for the current round  $r$  or the next one ( $r + 1$ ).

**Algorithm of the Synchronizer  $\lambda$**  The algorithm defining the behavior of a process  $p_i$  is described in Fig. 9.16. This description follows the previous explanation.

**Complexities** The time complexity of the synchronizer  $\lambda$  is a function of  $\rho$ . Taking  $\rho = 2$ , the time of the synchronous algorithm is multiplied by 2 and we have  $T_\lambda^{pulse} = O(1)$ . Moreover, as every message  $m$  has to carry one control bit and there are no additional control messages, the message complexity is  $C_\lambda^{pulse} = O(1)$ . Finally, the initialization part consists of the initialization of the local physical clocks of the processes.

### 9.5.4 Synchronizer $\mu$

**Aim of Synchronizer  $\mu$**  The aim of synchronizer  $\mu$  is to ensure that each message sent at pulse  $r$  is received by its destination process  $p_j$  while it executes the pulse  $r$ . So that no message arrives too early, we need to have

$$\tau_j + \rho r \leq \tau_j(r) < \tau_j + (r + 1)\rho \quad (\text{R5}).$$

From an operational point of view, this means that the parity bits used in  $\lambda$  have to be eliminated.

```

when  $timer_i = \rho r$  do
     $clock_i \leftarrow r$ . % next pulse is generated %

when  $timer_i = \rho \times clock_i + \eta$  do
    Send the messages  $MSG(-)$  of the local synchronous algorithm;
    When received, process pulse  $clock_i$  messages.

when  $MSG(m)$  is received on  $channel_i[x]$  do
     $m$  belongs to the current pulse; deliver it to the synchronous algorithm.

```

**Fig. 9.17** Synchronizer  $\mu$  (code for  $p_i$ )

**Determining the Appropriate Timing Parameters** One way of achieving the previous requirement consists in delaying the sending of messages by an appropriate amount  $\eta$  of time units (in such a way that  $\eta = 0$  would correspond to the synchronizer  $\lambda$ ).

As the transit time is upper bounded by 1, and the sender  $p_i$  sends a pulse  $r$  message at time  $\tau_i + r\rho + \eta$ , we have on the one hand  $\tau_j(r) \leq \tau_i + r\rho + \eta + 1$ , and on the other hand  $\tau_j(r) \geq \tau_i + r\rho + \eta$  (as transit times are not negative). Hence, to satisfy (R5), it is sufficient to ensure

$$\tau_j(r) \leq \tau_i + r\rho + \eta + 1 \leq \tau_j + (r + 1)\rho \quad (\text{R6}),$$

and

$$\tau_j(r) \geq \tau_i + r\rho + \eta \geq \tau_j + r\rho \quad (\text{R7})$$

or again

$$\rho \geq \eta + 1 + (\tau_i - \tau_j) \quad \text{and} \quad \eta \geq \tau_j - \tau_i.$$

But we have from (R1):  $\tau_j - \tau_i < 1$  and  $\tau_i - \tau_j < 1$ . Hence, conditions (R6) and (R7) are satisfied when

$$\rho \geq \eta + 2 \quad \text{and} \quad \eta \geq 1.$$

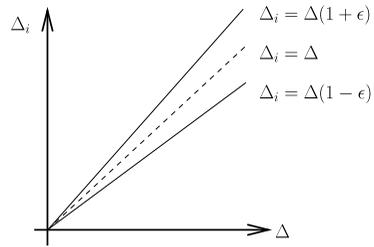
It follows that any pair of values  $(\eta, \rho)$  satisfying the previous condition ensures that any message is received at the same pulse at which it has been sent. The smallest values for  $\eta$  and  $\rho$  are thus 1 and 3, respectively.

**Algorithm of Synchronizer  $\mu$**  The algorithm defining the behavior of a process  $p_i$  is described in Fig. 9.17. This synchronizer adds neither control messages nor control information to application messages, and we have  $C_\mu^{pulse} = T_\mu^{pulse} = O(1)$ .

### 9.5.5 When the Local Physical Clocks Drift

**When the Clocks Drift** The previous section assumed that, while local clocks of the processes (processors) do not output the same value at the same reference

**Fig. 9.18** Clock drift with respect to reference time



time, they progress at the same speed. This simplifies the design of synchronizers  $\alpha$  and  $\beta$ .

Unfortunately, physical clocks drift, but fortunately their drift with respect to the abstract global time perceived by an omniscient external observer (also called *reference time*) is usually bounded and captured by a parameter denoted  $\epsilon$  (called the *clock drift*).

Hence, we consider that the faster clock counts one unit for  $(1 - \epsilon)$  of reference time, while the slowest clock counts one unit for  $(1 + \epsilon)$  of reference time (Fig. 9.18). Let  $\Delta$  denote a time duration measured with respect to the reference time, and  $\Delta_i$  be this time duration as measured by the clock of  $p_i$ . We have

$$\Delta(1 - \epsilon) \leq \Delta_i \leq \Delta(1 + \epsilon).$$

This formula is depicted in Fig. 9.18. As  $\epsilon$  is usually very small,  $\epsilon^2$  is assumed to be equal to 0 (hence  $(1 - \epsilon)(1 + \epsilon) \simeq 1$ ). The upper and lower lines define what is called the *linear time envelope* of a clock. The upper line in the figure considers the case where the clock of  $p_i$  always counts 1 for  $(1 - \epsilon)$  of reference time, while the lower line considers the case where its clock always counts 1 for  $(1 + \epsilon)$  of reference time. The dotted line in the middle considers the case where the clock does not drift. Of course, the drift of a clock is not constant, and a given clock can have a positive drift at some time and a negative one at another time.

It follows that, advancing at different rates, the clocks can measure differently a given interval of the reference time. It is consequently important to know how many pulses can be generated before such a confusion occurs, and makes a message arrive too late with respect to its pulse number, thereby making the corresponding synchronizer incorrect.

**Conditions for Synchronizer  $\mu$**  We consider here the case of synchronizer  $\mu$ . Let  $p_i$  and  $p_j$  be two neighbor processes, and let us assume that their clocks (*timer<sub>i</sub>* and *timer<sub>j</sub>*) have maximum and opposite drifts (i.e., of  $\epsilon$  with respect to the reference time).

As we have seen, the conditions (R6) and (R7) stated in Sect. 9.5.4 ensure the respect of condition (R7), which defines the correct behavior of  $\mu$ , namely a message sent by  $p_i$  at pulse  $r$  arrives at  $p_j$  during its pulse  $r$ .

When considering (R6), the worst case is produced when the sender  $p_i$  has the slower clock (*timer<sub>i</sub>* counts 1 for  $1 + \epsilon$  of reference time), while the receiver process

$p_j$  has the faster clock ( $timer_i$  counts 1 for  $1 - \epsilon$  of reference time). In such a context, the condition (R6) becomes

$$\tau_i + (r\rho + \eta)(1 + \epsilon) + 1 \leq \tau_j + (r + 1)\rho(1 - \epsilon) \quad (\text{R6}'),$$

where everything is expressed within the abstract reference time.

When considering (R7), the worst case is produced when the sender  $p_i$  has the faster clock, while the receiver  $p_j$  has the slower clock. In such a context, the condition (R7) becomes

$$\tau_i + (r\rho + \eta)(1 - \epsilon) \geq \tau_j + r\rho(1 + \epsilon) \quad (\text{R7}').$$

Let us now consider (R1) to eliminate  $\tau_i$  and  $\tau_j$  and obtain conditions on  $\rho$ ,  $\eta$ , and  $\epsilon$ , which implies simultaneously (R6') and (R7').

- Taking  $\tau_i - \tau_j < 1$ , and considering (R7'), we obtain the following condition (C1) which implies (R6')

$$\text{C1} = [2r\rho\epsilon \leq \rho(1 - \epsilon) - 2 - \eta(1 + \epsilon)].$$

- Taking  $\tau_j - \tau_i < 1$ , and considering (R7'), we obtain the following condition (C2), which implies (R7')

$$\text{C2} = (2r\rho\epsilon \leq \eta(1 - \epsilon) - 1).$$

It follows from these conditions that the greatest pulse number  $r_{max}$  that can be attained without problem is such that  $2r_{max}\rho\epsilon = \rho((1 - \epsilon) - 2 - \eta(1 + \epsilon)) = \eta(1 - \epsilon) - 1$ , which is obtained for

$$\eta = \frac{\rho(1 - \epsilon) - 1}{2},$$

and we have then

$$r_{max} = \frac{\rho(1 - \epsilon)^2 - 3 + \epsilon}{4\rho\epsilon}.$$

Let us remark that, when there is no drift, we have  $\epsilon = 0$ , and we obtain  $r_{max} = +\infty$  and  $2\eta = \rho - 1$ . As already seen in Sect. 9.5.4,  $\rho = 3$  and  $\eta = 1$  are the smallest values satisfying this equation.

Considering physical clocks whose drift is  $10^{-1}$  seconds a day (i.e.,  $\epsilon = 1/864000$ ), Table 9.1 shows a few numerical results for three increasing values of  $\rho$ .

## 9.6 Summary

This chapter has presented the concept of a synchronizer which encapsulates a general methodology to simulate (non-real-time) distributed synchronous algorithms on

**Table 9.1** Value of  $r_{max}$  as a function of  $\rho$

Value of $\rho$	Value of $r_{max}$
4	54 000
8	135 000
12	162 000

top of asynchronous distributed systems. It has presented several synchronizers for both pure asynchronous systems and for bounded delay networks. The chapter has also shown that graph covering structures (spanning trees and  $t$ -spanners) are important concepts when one has to define appropriate overlay structures on which are sent control messages.

### 9.7 Bibliographic Notes

- The interpretation of synchronous distributed algorithms on asynchronous distributed systems was introduced by B. Awerbuch [27] who in 1985 introduced the concept of a synchronizer and defined synchronizers  $\alpha$ ,  $\beta$ , and  $\gamma$ . This paper also presents a distributed algorithm that builds partitions for  $\gamma$  where the spanning trees and the intergroups channels are such that, given  $k \in [1..n]$ , we obtain a message complexity  $C_\gamma^{pulse} \leq kn$  and a time complexity  $T_\gamma^{pulse} \leq \log_k n$ . The work of B. Awerbuch is mainly focused on the design of synchronizers from the point of view of their time and message complexities.
- Synchronizer  $\delta$  is due to D. Peleg and J.D. Ullman [293]. They applied it to networks whose  $t$ -spanner is a hypercube structure. The notion of a graph  $t$ -spanner is due to D. Peleg. An in-depth study of these graph covering structures can be found in [292].
- The synchronization-related concepts used in the design of synchronizers have their origin in the work of R.G. Gallager [142] (1982), A. Segall [341] (1983), and B. Awerbuch [26] (1985).

The idea of a safe state of a process (i.e., it knows that all its messages have been received) has been used in many distributed algorithms. One of the very first works using this idea is a paper by E.W. Dijkstra and C.S. Scholten [115], who use it to control termination of what they called a “diffusing computation”.

- The study of synchronizers for bounded delay networks is due to C.T. Chou, I. Cidon, I. Gopal, and S. Zaks [93], who defined the synchronizers  $\lambda$  and  $\mu$ . Their bounded delay assumption is valuable for many local area networks and real-time embedded systems. Adaptation to the case where clocks can drift is addressed in [319].
- The interested reader will find in [385] a method of simulating a system composed of synchronous processes and asynchronous channels on top of a system where both processes and channels are asynchronous. This type of simulation is investigated in the presence of different types of faults.

- On the implementation side, K.B. Lakshmanan and K. Thulisaraman studied the problems posed by the management of waiting queues in which messages are stored, when one has to implement a synchronizer on top of a fully asynchronous system [225]. On the theoretical side, A. Fekete, N.A. Lynch, and L. Shrira presented a general methodology, based on communicating automata, for the design of modular proofs of synchronizers [123].
- Numerous works have addressed the synchronization of distributed physical clocks (e.g., [101, 116, 230, 291, 358, 386]).
- Optimal synchronization for asynchronous bounded delay networks with drifting local clocks is addressed in [211].
- The distributed unison problem requires that (a) no process starts its round  $r + 1$  before all processes have executed their round  $r$ , and (b) no process remains blocked forever in a given round. Self-stabilizing algorithms are algorithms that can cope with transient faults such as the corruption of values [112, 118]. An abundant literature has investigated distributed unison in the context of self-stabilizing algorithms. The interested reader can consult [60, 100, 162] to cite a few.

## 9.8 Exercises and Problems

1. The simulation of a synchronous system implemented by the synchronizer  $\alpha$  presented in Sect. 9.3.1 is more synchronized than what is needed. More precisely, it allows a process to proceed to next pulse  $r + 1$  only when (a) its neighbors are safe with respect to pulse  $r$ , and (b) it is safe itself with respect to pulse  $r$ . But, as the reader can observe, item (b) is not required by the property  $\mathcal{P}$ , which has to be satisfied for a process to locally progress from its current pulse to the next one (property  $\mathcal{P}$  is stated in Sect. 9.2.1).  
 Modify the synchronizer  $\alpha$  in order to obtain a less synchronized synchronizer  $\alpha'$  in which only item (a) is used to synchronize neighbors' processes.  
 Solutions in [27, 319].
2. Write versions of synchronizers  $\lambda$  and  $\mu$  in which the drift of the physical clocks of processors is upper bounded by  $\epsilon$ .
3. Prove that it is impossible to implement a synchronizer in an asynchronous distributed system in which processes (even a single one) may crash.

## Part III

# Mutual Exclusion and Resource Allocation

This part of the book and the following one are on the enrichment of the distributed message-passing system in order to offer high-level operations to processes. This part, which is on resource allocation, is composed of two chapters. (The next part will be on high-level communication operations.)

Chapter 10 introduces the mutual exclusion (mutex) problem, which is the most basic problem encountered in resource allocation. An algorithm solving the mutex problem has to ensure that a given hardware or software object (resource) is accessed by at most one process at a time, and that any process that wants to access it will be able to do so. Two families of mutex algorithms are presented. The first is the family of algorithms based on individual permissions, while the second is the family of algorithms based on arbiter permissions. A third family of mutex algorithms is the family of token-based algorithms. Such a family was already presented in Chap. 5, devoted to mobile objects navigating a network (a token is a dataless mobile object).

Chapter 11 considers first the problem posed by a single resource with several instances, and then the problem posed by several resources, each with multiple instances. It assumes that a process is allowed to acquire several instances of several resources. The main issues are then to prevent deadlocks from occurring and to provide processes with efficient allocation algorithms (i.e., algorithms which reduce process waiting chains).

# Chapter 10

## Permission-Based Mutual Exclusion Algorithms

This chapter is on one of the most important synchronization problems, namely mutual exclusion. This problem (whose name is usually shortened to “mutex”) consists of ensuring that at most one process at a time is allowed to access some resource (which can be a physical or a virtual resource).

After having defined the problem, the chapter presents two approaches which allow us to solve it. Both are based on permissions given by processes to other processes. The algorithms of the first approach are based on individual permissions, while the algorithms of the second approach are based on arbiter permissions (arbiter-based algorithms are also called quorum-based algorithms).

**Keywords** Adaptive algorithm · Arbiter permission · Bounded algorithm · Deadlock-freedom · Directed acyclic graph · Extended mutex · Adaptive algorithm · Grid quorum · Individual permission · Liveness property · Mutual exclusion (mutex) · Preemption · Quorum · Readers/writers problem · Safety property · Starvation-freedom · Timestamp · Vote

### 10.1 The Mutual Exclusion Problem

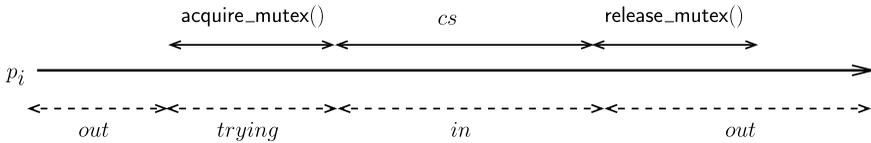
#### 10.1.1 Definition

**The Operations** `acquire_mutex()` and `release_mutex()` The mutual exclusion problem consists in enriching the underlying system with two operations denoted `acquire_mutex()` and `release_mutex()`, which are used as “control statements” to encapsulate a set of application statements usually called a *critical section*. Let  $cs$  denote such a set of application statements. A process  $p_i$  that wants to execute  $cs$  issues the following sequence of statements:

`acquire_mutex(); cs; release_mutex()`.

The operations `acquire_mutex()` and `release_mutex()` can be seen as “control brackets” which open and close a particular execution context, respectively.

It is assumed that the processes are well-formed, i.e., they always execute the previous pattern when they want to execute the statements  $cs$ . Moreover, it is assumed that, when executed by a single process, the code denoted  $cs$  always terminates.



**Fig. 10.1** A mutex invocation pattern and the three states of a process

**The Three States of a Process** Given a process  $p_i$ , let  $cs\_state_i$  be a local variable denoting its current local state from the critical section point of view. We have  $cs\_state_i \in \{out, trying, in\}$ , where

- $cs\_state_i = out$ , means that  $p_i$  is not interested in executing the statement  $cs$ .
- $cs\_state_i = trying$ , means that  $p_i$  is executing the operation `acquire_mutex()`.
- $cs\_state_i = in$ , means that  $p_i$  is executing the statement  $cs$ .

An invocation pattern of `acquire_mutex()` and `release_mutex()`, together with the corresponding values of  $cs\_state_i$ , is represented in Fig. 10.1.

**Problem Definition** The mutual exclusion problem consists in designing an algorithm that implements the operations `acquire_mutex()` and `release_mutex()` in such a way that the following properties are satisfied:

- **Safety.** At any time, at most one process  $p_i$  is such that  $cs\_state_i = in$  (we say that at most one process is inside the critical section).
- **Liveness.** If a process  $p_i$  invokes `acquire_mutex()`, then we eventually have  $cs\_state_i = in$  (i.e., if  $p_i$  wants to enter the critical section, it eventually enters it).

This liveness property is sometimes called *starvation-freedom*. It is important to notice that it is a property stronger than the absence of deadlock. Deadlock-freedom states that if processes want to enter the critical section (i.e., invoke `acquire_mutex()`), at least one process will enter it. Hence, a solution that would allow some processes to repeatedly enter the critical section, while preventing other processes from entering, would ensure deadlock-freedom but would not ensure starvation-freedom.

### 10.1.2 Classes of Distributed Mutex Algorithms

**Mutex Versus Election** The election problem was studied in Chap. 4. The aim of both an election algorithm and a mutex algorithm is to create some *asymmetry* among the processes. But these problems are deeply different. In the election problem, any process can be elected and, once elected, a process remains elected forever. In the mutex problem, each process that wants to enter the critical section must eventually be allowed to enter it. In this case, the asymmetry pattern evolves dynamically according to the requests issued by the processes.

**Token-Based Algorithms** One way to implement mutual exclusion in a message-passing system consists in using a token that is never duplicated. Only the process that currently owns the token can enter the critical section. Hence, the safety property follows immediately from the fact that there is a single token. A token-based mutex algorithm has only to ensure that any process that wants the token will eventually obtain it.

Such algorithms were presented in Chap. 5, which is devoted to mobile objects navigating a network. In this case, the mobile object is a pure control object (i.e., the token carries no application-specific data) which moves from process to process according to process requests.

**Algorithms Based on Individual/Arbiter Permissions** This chapter is on the class of mutex algorithms which are based on permissions. In this case, a process that wants to enter the critical section must ask for permissions from other processes. It can enter the critical section only when it has received all the permissions it has requested. Two subclasses of permission-based algorithms can be distinguished.

- In the case of individual permissions, the permission given by a process  $p_i$  to a process  $p_j$  engages only  $p_i$ . Mutex algorithms based on individual permissions are studied in Sects. 10.2 and 10.3.
- In the case of arbiter permissions, the permission given by a process  $p_i$  to a process  $p_j$  engages all the processes that need  $p_i$ 's permission to enter the critical section. Mutex algorithms based on arbiter permissions are studied in Sect. 10.4.

In the following  $R_i$  represents the set of processes to which  $p_i$  needs to ask permission in order to enter the critical section.

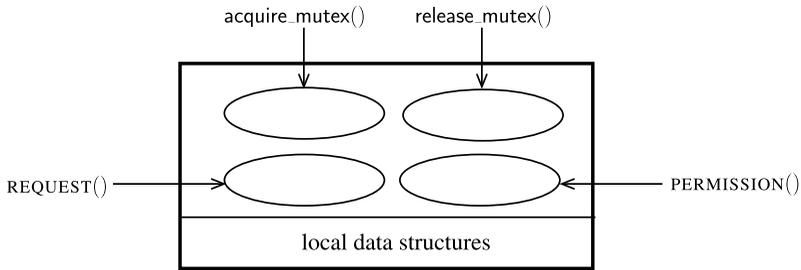
**Remark on the Underlying Network** In all the algorithms that are presented in Sects. 10.2, 10.3, and 10.4, the communication network is fully connected (there is a bidirectional channel connecting any pair of distinct processes). Moreover, the channels are not required to be FIFO.

## 10.2 A Simple Algorithm Based on Individual Permissions

### 10.2.1 Principle of the Algorithm

The algorithm presented in this section is due to G. Ricart and A.K. Agrawala (1981).

**Principle: Permissions and Timestamps** The principle that underlies this algorithm is very simple. We have  $R_i = \{1, \dots, n\} \setminus \{i\}$ , i.e., a process  $p_i$  needs the permission of each of the  $(n - 1)$  other processes in order to be allowed to enter the critical section. As already indicated, the intuitive meaning of the permission sent by  $p_j$  to  $p_i$  is the following “as far as  $p_i$  (only) is concerned,  $p_i$  allows  $p_j$  to enter the critical section”.



**Fig. 10.2** Mutex module at a process  $p_i$ : structural view

Hence, when a process  $p_i$  wants to enter the critical section, it sends a `REQUEST()` message to each other process  $p_j$ , and waits until it has received the  $(n - 1)$  corresponding permissions. The core of the algorithm is the predicate used by a process  $p_j$  to send its permission to a process  $p_i$  when it receives a request from this process. There are two cases. The behavior of  $p_j$  depends on the fact that it is currently interested or not in the critical section.

- If  $p_j$  is not interested (i.e.,  $cs\_state_j = out$ ), it sends by return its permission to  $p_i$ .
- If  $p_j$  is interested (i.e.,  $cs\_state_j \neq out$ ), it is either waiting to enter the critical section or is inside the critical section. In both cases,  $p_j$  has issued a request, and the requests from  $p_j$  and  $p_i$  are conflicting. One process has to give its permission to the other one (otherwise, they will deadlock each other), and only one has to give its permission by return (otherwise, the safety property would be violated). This issue can be solved by associating a priority with each request.

Priorities can easily be implemented with timestamps. As we have seen in Chap. 7, a timestamp is a pair  $\langle h, j \rangle$ , where  $h$  is a logical clock value and  $i$  a process identity. As we have seen in Sect. 7.1.2, any set of timestamps can be totally ordered by using a topological sort, namely,  $\langle h1, i \rangle$  and  $\langle h2, j \rangle$  being two timestamps, we have

$$\langle h1, i \rangle < \langle h2, j \rangle \stackrel{def}{=} (h1 < h2) \vee ((h1 = h2) \wedge (i < j)).$$

It follows that this algorithm uses timestamps to ensure both the safety property and the liveness property defining the mutex problem.

**Structural View** The structure of the local module implementing the mutual exclusion service at a process  $p_i$  is described in Fig. 10.2. (As the reader can check, this structure is the same as the one described in Fig. 5.2.)

```

operation acquire_mutex() is
(1)  $cs\_state_i \leftarrow \text{trying}$ ;
(2)  $\ell rd_i \leftarrow \text{clock}_i + 1$ ;
(3)  $\text{waiting\_from}_i \leftarrow R_i$ ; %  $R_i = \{1, \dots, n\} \setminus \{i\}$ 
(4) for each  $j \in R_i$  do send REQUEST( $\ell rd_i, i$ ) to  $p_j$  end for;
(5) wait ( $\text{waiting\_from}_i = \emptyset$ );
(6)  $cs\_state_i \leftarrow \text{in}$ .

operation release_mutex() is
(7)  $cs\_state_i \leftarrow \text{out}$ ;
(8) for each  $j \in \text{perm\_delayed}_i$  do send PERMISSION( $i$ ) to  $p_j$  end for;
(9)  $\text{perm\_delayed}_i \leftarrow \emptyset$ .

when REQUEST( $k, j$ ) is received do
(10)  $\text{clock}_i \leftarrow \max(\text{clock}_i, k)$ ;
(11)  $\text{prio}_i \leftarrow (cs\_state_i \neq \text{out}) \wedge (\langle \ell rd_i, i \rangle < \langle k, j \rangle)$ ;
(12) if ( $\text{prio}_i$ ) then  $\text{perm\_delayed}_i \leftarrow \text{perm\_delayed}_i \cup \{j\}$ 
(13) else send PERMISSION( $i$ ) to  $p_j$ 
(14) end if.

when PERMISSION( $j$ ) is received do
(15)  $\text{waiting\_from}_i \leftarrow \text{waiting\_from}_i \setminus \{j\}$ .

```

Fig. 10.3 A mutex algorithm based on individual permissions (code for  $p_i$ )

## 10.2.2 The Algorithm

**Description of the Algorithm: Local Variables** In addition to the constant set  $R_i$  and the local variable  $cs\_state_i$  (initialized to *out*), each process  $p_i$  manages the following variables:

- $\text{clock}_i$  is a scalar clock initialized to 0. Its scope is the whole execution;  $\ell rd_i$  (for *last request date*) is a local variable used by  $p_i$  to save the logical date of its last invocation of `acquire_mutex()`.
- $\text{waiting\_from}_i$  is a set used to contain the identities of the processes from which  $p_i$  is waiting for a permission.
- $\text{perm\_delayed}_i$  is a set used by  $p_i$  to contain the identities of the processes to which it will have to send its permission when it exits the critical section.
- $\text{prio}_i$  is an auxiliary Boolean variable where  $p_i$  computes its priority when it receives a request message.

**Description of the Algorithm: Behavior of a Process** The algorithm implementing mutual exclusion at a process  $p_i$  is described in Fig. 10.3. Except for the **wait** statement at line 5, the four sets of statements are locally executed in mutual exclusion.

When it invokes `acquire_mutex()`, a process  $p_i$  first updates  $cs\_state_i$  (line 1), computes a clock value for its current request (line 2), and sets  $\text{waiting\_from}_i$  to  $R_i$  (line 3). Then, it sends a timestamped request message to each other process (line 4),

and waits until it has received the corresponding permissions (line 5). When this occurs, it enters the critical section (line 6).

When it receives a permission,  $p_i$  updates accordingly its set  $waiting\_from_i$  (line 15).

When it invokes `release_mutex()`,  $p_i$  proceeds to the local state *out* (line 7), and sends its permission to all the processes of the set  $perm\_delayed_i$ . This is the set of processes whose requests were competing with  $p_i$ 's request, but  $p_i$  delayed the corresponding permission-sending because its own request has priority over them (lines 8–9).

When  $p_i$  receives a message `REQUEST( $k, j$ )`, it first updates its local clock (line 10). It then computes if it has priority (line 11), which occurs if  $cs\_state_i \neq out$  (it is then interested in the critical section) and the timestamp of its current request is smaller than the timestamp of the request it has just received. If  $p_i$  has priority, it adds the identity  $j$  to the set  $perm\_delayed_i$  (line 12). If  $p_i$  does not have priority, it sends by return its permission to  $p_j$  (line 13).

**A Remark on the Management of  $clock_i$**  Let us observe that  $clock_i$  is not increased when  $p_i$  invokes `acquire_mutex()`: The date associated with the current request of  $p_i$  is the value of  $clock_i$  plus 1 (line 2). Moreover, when  $p_i$  receives a request message, it updates  $clock_i$  to  $\max(clock_i, k)$ , where  $k$  is the date of the request just received by  $p_i$  (line 10), and this update is the only update of  $clock_i$ .

As a very particular case, let us consider a scenario in which only  $p_i$  wants to enter the critical section. It is easy to see that, not only  $\langle 1, i \rangle$  is the timestamp of its first request, but  $\langle 1, i \rangle$  is the timestamp of all its requests (this is because, as line 10 is never executed,  $clock_i$  remains forever equal to 0).

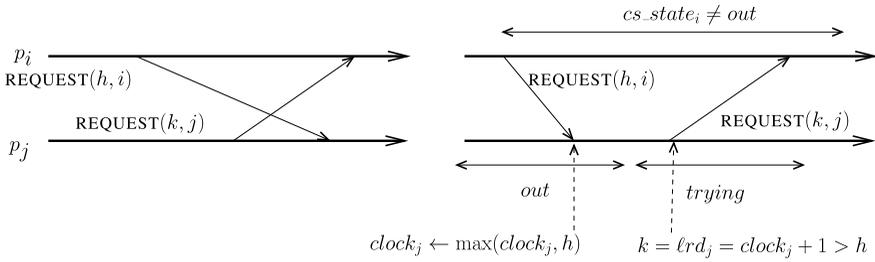
As we are about to see in the following proof, the algorithm is correct. It actually considers that, when a process  $p_i$  enters several times the critical section while the other processes are not interested, it is not necessary to increase the clock of  $p_i$ . This is because, in this case and from a clock point of view, the successive invocations of `acquire_mutex()` issued by  $p_i$  can appear as a single invocation. Hence, the algorithm increases the local clocks as slowly as possible.

**Message Cost** It is easy to see that each use of the critical section by a process requires  $2(n - 1)$  messages:  $(n - 1)$  request messages and  $(n - 1)$  permission messages.

### 10.2.3 Proof of the Algorithm

**Lemma 3** *The algorithm described in Fig. 10.3 satisfies the mutex safety property.*

*Proof* The proof is by contradiction. Let us assume that two processes  $p_i$  and  $p_j$  are simultaneously in the critical section, i.e., from an external omniscient observer point of view, we have  $cs\_state_i = in$  and  $cs\_state_j = in$ . It follows from the code of



**Fig. 10.4** Proof of the safety property of the algorithm of Fig. 10.3

`acquire_mutex()` that each of them has sent a request message to the other process and has received its permission. A priori two scenarios are possible for  $p_i$  and  $p_j$  to be simultaneously inside the critical section. Let  $\langle h, i \rangle$  and  $\langle k, j \rangle$  be the timestamps of the request messages sent by  $p_i$  and  $p_j$ , respectively.

- Each process has sent its request message before receiving the request from the other process (left side of Fig. 10.4).

As  $i \neq j$ , we have either  $\langle h, i \rangle < \langle k, j \rangle$  or  $\langle k, j \rangle < \langle h, i \rangle$ . Let us assume (without loss of generality) that  $\langle h, i \rangle < \langle k, j \rangle$ . In this case,  $p_j$  is such that  $\neg prio_j$  when it received  $\text{REQUEST}(h, i)$  (line 11) and, consequently, it sent its permission to  $p_i$  (line 13). Differently, when  $p_i$  received  $\text{REQUEST}(h, i)$ , we had  $prio_i$ , and consequently  $p_i$  did not send its permission to  $p_j$  (line 12; it will send the permission only when it will execute (line 8 of `release_mutex()`)).

It follows that, when this scenario occurs,  $p_j$  cannot enter the critical section while  $p_i$  is inside the critical section, which contradicts the initial assumption.

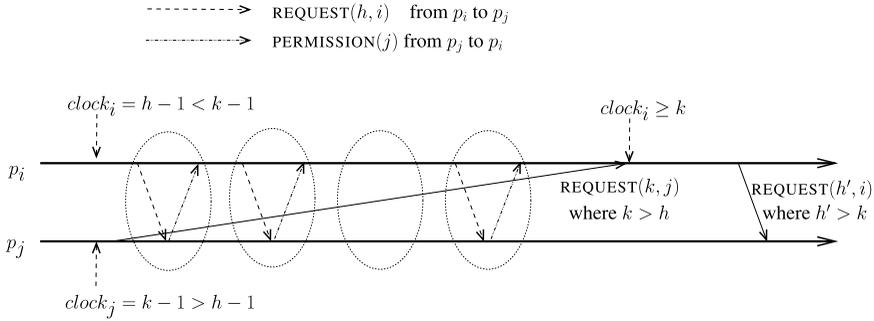
- One process (e.,  $p_j$ ) has sent its permission to the other process ( $p_i$ ) before sending its own request (right side of Fig. 10.4).

When  $p_j$  receives  $\text{REQUEST}(h, i)$ , it executes  $clock_j \leftarrow \max(clock_j, h)$  (line 10), hence we have  $clock_j \geq h$ . Then, when later  $p_j$  invokes `acquire_mutex()`, it executes  $lrd_j \leftarrow clock_j + 1$  (line 2). Hence,  $lrd_j > h$ , and the message  $\text{REQUEST}(k, j)$  sends by  $p_j$  to  $p_i$  is such that  $k = lrd_j > h$ . It follows that, when  $p_i$  receives this message we have  $cs\_state_i \neq out$  (assumption), and  $\langle h, i \rangle < \langle k, j \rangle$ . Consequently  $prio_i$  is true (line 11), and  $p_i$  does not send its permission to  $p_j$  (line 12). It follows that, as in the previous case,  $p_j$  cannot enter the critical section while  $p_i$  is inside the critical section, which contradicts the initial assumption.  $\square$

It can be easily checked that the previous proof remains valid if messages are lost.

**Lemma 4** *The algorithm described in Fig. 10.3 satisfies the mutex liveness property.*

*Proof* The proof of the liveness property is done in two parts. The first part shows that the algorithm is deadlock-free. The second part shows the algorithm



**Fig. 10.5** Proof of the liveness property of the algorithm of Fig. 10.3

is starvation-free. Let us first observe that as the clocks cannot decrease, the timestamps cannot decrease.

**Proof of the deadlock-freedom property.** Let us assume, by contradiction, that processes have invoked `acquire_mutex()` and none of them enters the local state *in*. Among all these processes, let  $p_i$  be the process that sent the request message with the smallest timestamp  $\langle h, i \rangle$ , and let  $p_j$  any other process. When  $p_i$  receives `REQUEST( $h, i$ )` it sends by return the message `PERMISSION( $j$ )` to  $p_i$  if  $cs\_state_j = out$ . If  $cs\_state_j \neq out$ , let  $\langle k, j \rangle$  the timestamp of its request. Due to definition of  $\langle h, i \rangle$ , we have  $\langle h, i \rangle < \langle k, j \rangle$ . Consequently,  $p_j$  sends `PERMISSION( $j$ )` to  $p_i$ . It follows that  $p_i$  receives a permission message from each other process (line 15). Consequently,  $p_i$  stops waiting (5) and enters the critical section (line 6). Hence, the algorithm is deadlock-free.

**Proof of the starvation-freedom property.** To show that the algorithm is starvation-free, let us consider two processes  $p_i$  and  $p_j$  which are competing to enter the critical section. Moreover, let us assume that  $p_i$  repeatedly invokes `acquire_mutex()` and enters the critical section, while  $p_j$  remains blocked at line 5 waiting for the permission from  $p_i$ . The proof consists in showing that this cannot last forever.

Let  $\langle h, i \rangle$  and  $\langle k, j \rangle$  be the timestamps of the requests of  $p_i$  and  $p_j$ , respectively, with  $\langle h, i \rangle < \langle k, j \rangle$ . The proof shows that there is a finite time after which the timestamp of a future request of  $p_i$  will be  $\langle h', i \rangle > \langle k, j \rangle$ . When this will occur, the request of  $p_j$  will have priority with respect to that of  $p_i$ . The worst case scenario is described in Fig. 10.5:  $clock_i = h - 1 < clock_j = k - 1$ , the request messages from  $p_i$  to  $p_j$  and the permission messages from  $p_j$  to  $p_i$  are very fast, while the request message from  $p_j$  to  $p_i$  is very slow. When it receives the message `REQUEST( $h, i$ )`,  $p_j$  sends by return its permission to  $p_i$ . This message pattern, which is surrounded by an ellipsis, can occur repeatedly an unbounded number of times, but the important point is that it cannot appear an infinite number of times. This is because, when  $p_i$  receives the message `REQUEST( $k, j$ )`, it updates  $clock_i$  to  $k$  and, consequently, its next request message (if any) will carry a date greater than  $k$  and will have a smaller priority than  $p_j$ 's current request. Hence, no process can prevent another process from entering the critical section. □

The following property of the algorithm follows from the proof of the previous lemma. The invocations of `acquire_mutex()` direct the processes to enter the critical section according to the total order on the timestamps generated by these invocations.

**Theorem 13** *The algorithm described in Fig. 10.3 solves the mutex problem.*

*Proof* The proof is a direct consequence of Lemmas 3 and 4. □

### 10.2.4 From Simple Mutex to Mutex on Classes of Operations

**The Readers/Writers Problem** The most known generalization of the mutex problem is the readers/writers problem. This problem is defined by two operations, denoted `read()` and `write()`, whose invocations are constrained by the following synchronization rule.

- Any execution of the operation `write()` is mutually exclusive with the simultaneous execution of any (`read()` or `write()`) operation.

It follows from this rule that simultaneous executions of the operation `read()` are possible, as long as there is no concurrent execution of the operation `write()`.

**Generalized Mutex** More generally, it is possible to consider several types of operations and associated exclusion rules defined by a concurrency matrix. Let `op1()` and `op2()` be two operations whose synchronization types are *st1* and *st2*, respectively. These synchronization types are used to state concurrency constraints on the execution of the corresponding operations. To that end, a Boolean symmetric matrix denoted *exclude* is used (symmetric means that  $exclude[st1, st2] = exclude[st2, st1]$ ). Its meaning is the following: An operation whose type is *st1* and an operation whose type is *st2* cannot be executed simultaneously if  $exclude[st1, st2]$  is true.

As an example, let us consider the readers/writers problem. There are two operations and a synchronization type per operation, namely *r* is the type of the operation `read()` and *w* is the type associated with the operation `write()`. The concurrency matrix is such that  $exclude[r, r] = false$ ,  $exclude[w, r] = exclude[w, w] = false$ .

**An Extended Mutex Algorithm** Let us associate with each operation `op()` two control operations denoted `begin_op()` and `end_op()`. These control operations are used to bracket each invocation of `op()` as follows:

`begin_op(); op(); end_op()`.

Let *op\_type* be the synchronization type associated with the operation `op()` (let us observe that several operations can be associated with the same synchronization type). The algorithm described in Fig. 10.6 is a trivial extension of the mutex

```

operation begin_op() is
(1)   $cs\_state_i \leftarrow \text{trying};$ 
(2)   $\ell rd_i \leftarrow \text{clock}_i + 1;$ 
(3)   $\text{waiting\_from}_i \leftarrow R_i; \quad \% R_i = \{1, \dots, n\} \setminus \{i\}$ 
(4') for each  $j \in R_i$  do send REQUEST( $\ell rd_i, i, op\_type$ ) to  $p_j$  end for;
(5)  wait ( $\text{waiting\_from}_i = \emptyset$ );
(6)   $cs\_state_i \leftarrow \text{in}.$ 

operation end_op() is
(7)   $cs\_state_i \leftarrow \text{out};$ 
(8)  for each  $j \in \text{perm\_delayed}_i$  do send PERMISSION( $i$ ) to  $p_j$  end for;
(9)   $\text{perm\_delayed}_i \leftarrow \emptyset.$ 

when REQUEST( $k, j, op\_t$ ) is received do
(10)  $\text{clock}_i \leftarrow \max(\text{clock}_i, k);$ 
(11')  $\text{prio}_i \leftarrow (cs\_state_i \neq \text{out}) \wedge (\langle \ell rd_i, i \rangle < \langle k, j \rangle) \wedge \text{exclude}(op\_type, op\_t);$ 
(12) if ( $\text{prio}_i$ ) then  $\text{perm\_delayed}_i \leftarrow \text{perm\_delayed}_i \cup \{j\}$ 
(13)   else send PERMISSION( $i$ ) to  $p_j$ 
(14) end if.

when PERMISSION( $j$ ) is received do
(15)  $\text{waiting\_from}_i \leftarrow \text{waiting\_from}_i \setminus \{j\}.$ 

```

**Fig. 10.6** Generalized mutex based on individual permissions (code for  $p_i$ )

algorithm described in Fig. 10.3. It ensures that (a) the concurrency constraints expressed by the exclusion matrix are respected, and (b) the invocations which are not executed concurrently are executed according to their timestamp order.

Only two lines need to be modified to take into account the synchronization type of the operation (their number is postfixed by  $'$ ). At line 4', the request message has to carry the type of the corresponding operation. Then, at line 11', the Boolean value of  $\text{exclude}(op\_type, op\_t)$  (where  $op\_type$  is the type of the current operation of  $p_i$  and  $op\_t$  is the type of the operation that  $p_j$  wants to execute) is used to compute the value of  $\text{prio}_i$ , which determines if  $p_i$  has to send by return its permission to  $p_j$ .

It is easy to see that, if there is a single operation  $\text{op}()$ , and this operation excludes itself (i.e., its type  $op\_type$  is such that  $\text{exclude}(op\_type, op\_type) = \text{true}$ ), the algorithm boils down to that of Fig. 10.3.

## 10.3 Adaptive Mutex Algorithms Based on Individual Permissions

### 10.3.1 The Notion of an Adaptive Algorithm

The notion of an adaptive message-passing algorithm was introduced in Sect. 5.4.1, in the context of a mobile object navigating a network. In the context of mutual exclusion, it translates as follows. If after some time  $\tau$  a process  $p_i$  is no longer

interested in accessing the critical section, then there is a time  $\tau' \geq \tau$  after which this process is no longer required to participate in the mutual exclusion algorithm. It is easy to see that the mutex algorithm described in Fig. 10.3 is not adaptive: Each time a process  $p_i$  wants to enter the critical section, every other process  $p_j$  has to send it a permission, even if we always have  $cs\_state_j = out$ .

This section presents two adaptive mutex algorithms. The first one is obtained from a simple modification of the algorithm of Fig. 10.3. The second one has the noteworthy property of being both adaptive and bounded.

### 10.3.2 A Timestamp-Based Adaptive Algorithm

This algorithm is due to O. Carvalho and G. Roucairol (1983).

**Underlying Principle: Shared Permissions** Let us consider two processes  $p_i$  and  $p_j$  such that  $p_i$  wants to enter the critical section several times, while  $p_j$  is not interested in the critical section. The idea is the following:  $p_i$  and  $p_j$  share a permission and, once  $p_i$  has this permission, it keeps it until  $p_j$  asks for it. Hence, if  $p_j$  is not interested in the critical section,  $p_j$  will not reclaim it, and  $p_i$  will not have to ask for it again to  $p_j$ .

As an example, let consider the case where only  $p_i$  wants to enter the critical section and, due to its previous invocation of `acquire_mutex()`, it has the  $(n - 1)$  permissions that it shares with every other process. In this scenario,  $p_i$  can enter the critical section without sending request messages, and its use of the critical section then costs no message.

The previous idea can be easily implemented with a message `PERMISSION({i, j})` shared by each pair of processes  $p_i$  and  $p_j$  (`PERMISSION({j, i})` is not another permission but a synonym of `PERMISSION({i, j})`). Initially, this message is placed either on  $p_i$  or  $p_j$ , and the set  $R_i$  containing the identities of the processes to which  $p_i$  has to ask the permission is initialized as follows

$$R_i = \{j \mid \text{PERMISSION}(\{j, i\}) \text{ is initially on } p_j\}.$$

Then,  $p_i$  adds  $j$  to  $R_i$  when it sends `PERMISSION({i, j})` to  $p_j$ , and suppresses  $k$  from  $R_i$  when it receives `PERMISSION({k, i})` from  $p_k$ .

**Timestamp-Based Adaptive Mutex Algorithm** The corresponding algorithm is described in Fig. 10.7.

The code implementing `acquire_mutex()` is nearly the same as in Fig. 10.3. The only difference lies in the fact that the set  $R_i$  is no longer a constant. A process  $p_i$  sends a request message only to the processes from which it does not have shared permission (line 3), and then waits until it has the  $(n - 1)$  permissions it individually shares with each other process (line 4). When it receives such a permission it withdraws the corresponding process from  $R_i$  (line 17).

The code implementing `nearly_mutex()` is the same as in Fig. 10.3, with an additional statement related to the management of  $R_i$  (line 8). This set takes the value

```

operation acquire_mutex() is
(1)  $cs\_state_i \leftarrow trying$ ;
(2)  $\ell rd_i \leftarrow clock_i + 1$ ;
(3) for each  $j \in R_i$  do send REQUEST( $\ell rd_i, i$ ) to  $p_j$  end for;
(4) wait ( $R_i = \emptyset$ );
(5)  $cs\_state_i \leftarrow in$ .

operation release_mutex() is
(6)  $cs\_state_i \leftarrow out$ ;
(7) for each  $j \in perm\_delayed_i$  do send PERMISSION( $\{i, j\}$ ) to  $p_j$  end for;
(8)  $R_i \leftarrow perm\_delayed_i$ ;
(9)  $perm\_delayed_i \leftarrow \emptyset$ .

when REQUEST( $k, j$ ) is received from  $p_j$  do
(10)  $clock_i \leftarrow \max(clock_i, k)$ ;
(11)  $prio_i \leftarrow (cs\_state_i = in) \vee ((cs\_state_i = trying) \wedge (\ell rd_i, i) < (k, j))$ ;
(12) if ( $prio_i$ ) then  $perm\_delayed_i \leftarrow perm\_delayed_i \cup \{j\}$ 
(13)     else send PERMISSION( $\{i, j\}$ ) to  $p_j$ 
(14)          $R_i \leftarrow R_i \cup \{j\}$ ;
(15)         if ( $cs\_state_i = trying$ ) then send REQUEST( $\ell rd_i, i$ ) to  $p_j$  end if
(16) end if.

when PERMISSION( $\{i, j\}$ ) is received from  $p_j$  do
(17)  $R_i \leftarrow R_i \setminus \{j\}$ .

```

**Fig. 10.7** An adaptive mutex algorithm based on individual permissions (code for  $p_i$ )

of  $perm\_delayed_i$ , which contains the set of processes to which  $p_i$  has just sent the permission it shares with each of them.

Finally, when  $p_i$  receives a message REQUEST( $k, j$ ), it has the priority if it is currently inside the critical section or it is waiting to enter it (line 11). If its current request has priority with respect to the request it has just received, it delays the sending of its permission (line 12). If it does not have priority, it sends by return to  $p_j$  the message PERMISSION( $\{i, j\}$ ) (line 13) and adds  $j$  to  $R_i$  (line 14). Moreover, if  $p_i$  is competing for the critical section ( $cs\_state_i = trying$ ), it sends a request message to  $p_j$  so that  $p_j$  eventually returns to it the shared message PERMISSION( $\{i, j\}$ ) (line 15) so that it will be allowed to enter the critical section.

**Adaptivity, Message Cost, and the Management of Local Clocks** It is easy to see that the algorithm is adaptive. If after some time a process  $p_i$  does not invoke acquire\_mutex(), while other processes  $p_j$  do invoke this operation,  $p_i$  sends to each of them the permission it shares with them, after which it will no longer receive request messages.

It follows from the adaptivity property that the number of messages involved in one use of the critical section is  $2|R_i|$  (requests plus the associated permissions), where  $0 \leq |R_i| \leq n - 1$ . The exact number depends on the current state of the processes with respect to their invocations of acquire\_mutex() and release\_mutex().

When considering the basic algorithm of Fig. 10.3, a process that invokes acquire\_mutex() sends a timestamped request message to each other process. This

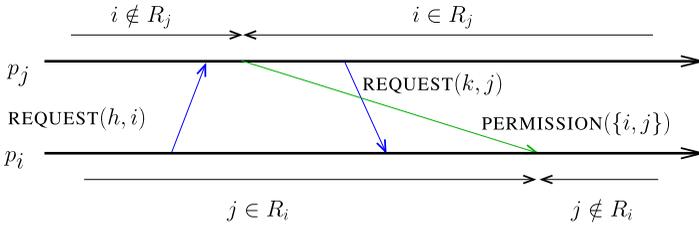


Fig. 10.8 Non-FIFO channel in the algorithm of Fig. 10.7

allows the local clocks to be synchronized in the sense that, as it has been already noticed, we always have  $|clock_i - clock_j| \leq n - 1$ . Due to the adaptivity feature of the previous algorithm, this local clock synchronization is no longer ensured and the difference between any two local clocks cannot be bounded.

**Non-FIFO Channels and Management of the  $R_i$  Sets** It is possible that a process  $p_i$  sends to a process  $p_j$  a message  $PERMISSION(\{i, j\})$  followed by a message  $REQUEST(\ell rd_i, i)$  (lines 13 and 15). As the channel is not required to be FIFO, it is possible that  $p_j$  receives and processes first the message  $REQUEST(\ell rd_i, i)$  and then the message  $PERMISSION(\{i, j\})$ . Does this create a problem?

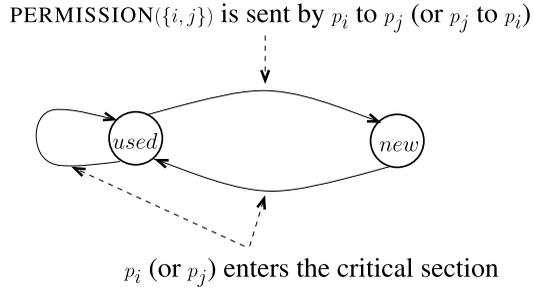
To see why the answer is “no”, let us consider Fig. 10.8, where are depicted a request message from  $p_i$  to  $p_j$ , the corresponding permission message from  $p_j$  to  $p_i$ , and a request message from  $p_j$  to  $p_i$ , which arrives at  $p_i$  before the request message. Predicates on the values of  $R_i$  and  $R_j$  are indicated on the corresponding process axis. Due to line 10, which is executed when  $p_j$  receives the message  $REQUEST(h, i)$ , and line 2, which is executed before  $p_j$  sends the message  $REQUEST(k, j)$ , we have  $k > h$ . It follows that, when  $p_i$  receives  $REQUEST(k, j)$ , we have  $\langle h, i \rangle < \langle k, j \rangle$ , i.e.,  $prio_i$  is true, and  $p_i$  delays the sending of the permission to  $p_j$  (line 12). Hence,  $p_i$  and  $p_j$  do not enter a livelock in which they would repeatedly send to each other the permission message without ever entering the critical section. The fact that the message  $REQUEST(k, j)$  arrives before or after the message  $PERMISSION(\{i, j\})$  has no impact on the way it is processed.

### 10.3.3 A Bounded Adaptive Algorithm

The mutex algorithm that is presented in this section has two main properties: It is adaptive and has only bounded variables. Moreover, process identities (whose scope is global) can be replaced by channel identities whose scopes are local to each process.

This algorithm is due to K.M. Chandy and J. Misra (1984). It is derived here from the permission-based algorithms which were presented previously. To simplify the presentation, we consider a version of the algorithm which uses process identities.

**Fig. 10.9** States of the message  $\text{PERMISSION}(\{i, j\})$



**Principle of the Algorithm: State of a Permission** As before, each pair of distinct processes  $p_i$  and  $p_j$  share a permission message denoted  $\text{PERMISSION}(\{i, j\})$ , and, to enter the critical section, a process needs each of the  $(n - 1)$  permissions it shares with each other process.

The main issue consists in ensuring, without using (unbounded) timestamps, that each invocation of `acquire_mutex()` terminates. To that end, a state is associated with each permission message; the value of such a state is *used* or *new*.

When a process  $p_i$  receives the message  $\text{PERMISSION}(\{i, j\})$  from  $p_j$ , the state of the permission is *new*. After it has benefited from this message to enter the critical section, the state of the permission becomes *used*. The automaton associated with the permission shared by  $p_i$  and  $p_j$  is described in Fig. 10.9.

**Principle of the Algorithm: Establish a Priority on Requests** The core of the algorithm is the way it ensures that each invocation of `require_mutex()` terminates. The state of a permission is used to establish a priority among conflicting requests.

Let  $\text{perm\_state}_i[j]$  be a local variable of  $p_i$ , which stores the current state of the permission shared by  $p_i$  and  $p_j$  when this permission is at process  $p_i$ .

When, while it has issued a request, process  $p_i$  receives a request from  $p_j$ , it has priority on  $p_j$  if one of the following cases occurs:

- $\text{cs\_state}_i = \text{in}$ . (Similarly to the corresponding case in the algorithm of Fig. 10.7,  $p_i$  has trivially priority because it is inside the critical section.)
- If  $\text{cs\_state}_i = \text{trying}$ , there are two subcases.
  - Case  $j \notin R_i$ . In this case, the permission shared by  $p_i$  and  $p_j$  is located at  $p_i$ , and  $p_i$  does not have to ask for it. The priority depends then on the state of this permission. If  $\text{perm\_state}_i[j] = \text{new}$ ,  $p_i$  has not yet used the permission and consequently it has priority. If  $\text{perm\_state}_i[j] = \text{used}$ ,  $p_i$  does not have priority with respect to  $p_j$ .
  - Case  $j \in R_i$ . As  $p_i$  receives a request from  $p_j$ , process  $p_j$  was such that  $i \in R_j$  when it sent the request, which means that  $p_j$  did not have the message  $\text{PERMISSION}(\{i, j\})$ . As  $j \in R_i$ , this permission message is not at process  $p_i$  either. It follows that this permission is still in transit from  $p_j$  to  $p_i$  (it has been sent by  $p_j$  before its request, but will be received by  $p_i$  after  $p_j$ 's request). When this permission message  $\text{PERMISSION}(\{i, j\})$  will be received by  $p_i$ , its

```

operation acquire_mutex() is
(1)  $cs\_state_i \leftarrow trying$ ;
(2) for each  $j \in R_i$  do send REQUEST() to  $p_j$  end for;
(3) wait ( $R_i = \emptyset$ );
(4)  $cs\_state_i \leftarrow in$ ;
(5) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do  $perm\_state_i[j] \leftarrow used$  end for.

operation release_mutex() is
(6)  $cs\_state_i \leftarrow out$ ;
(7) for each  $j \in perm\_delayed_i$  do send PERMISSION( $\{i, j\}$ ) to  $p_j$  end for;
(8)  $R_i \leftarrow perm\_delayed_i$ ;
(9)  $perm\_delayed_i \leftarrow \emptyset$ .

when REQUEST() is received from  $p_j$  do
(10)  $prio_i \leftarrow (cs\_state_i = in) \vee$ 
       $((cs\_state_i = trying) \wedge [(perm\_state_i[j] = new) \vee (j \in R_i)])$ ;
(11) if ( $prio_i$ ) then  $perm\_delayed_i \leftarrow perm\_delayed_i \cup \{j\}$ 
(12)   else send PERMISSION( $\{i, j\}$ ) to  $p_j$ 
(13)    $R_i \leftarrow R_i \cup \{j\}$ ;
(14)   if ( $cs\_state_i = trying$ ) then send REQUEST() to  $p_j$  end if
(15) end if.

when PERMISSION( $\{i, j\}$ ) is received from  $p_j$  do
(16)  $R_i \leftarrow R_i \setminus \{j\}$ ;
(17)  $perm\_state_i[j] \leftarrow new$ .

```

**Fig. 10.10** A bounded adaptive algorithm based on individual permissions (code for  $p_i$ )

state will be *new*, which means that the current request of  $p_i$  has priority on the request of  $p_j$ .

This scenario, which is due to the fact that channels are not FIFO, is exactly the scenario which has been depicted in Fig. 10.8 for the timestamp-based adaptive mutex algorithm (in Fig. 10.8, as  $p_i$  knows both the timestamp of its last request and the timestamp of  $p_j$ 's request, a timestamp comparison is used instead of the predicate  $j \in R_i$ ).

It follows that, when  $p_i$  receives a request from  $p_j$ , it has priority if

$$(cs\_state_i = in) \vee ((cs\_state_i = trying) \wedge [(perm\_state_i[j] = new) \vee (j \in R_i)]).$$

**Initialization** As in both previous algorithms, the local variables  $cs\_state_i$  and  $perm\_delayed_i$  of each process  $p_i$  are initialized to *out* and  $\emptyset$ , respectively. Moreover, for each pair of processes  $p_i$  and  $p_j$  such that  $i < j$ , the message PERMISSION( $\{i, j\}$ ) is initially at  $p_i$  (hence,  $i \in R_j$  and  $j \notin R_i$ ), and its initial state is *used* (i.e.,  $perm\_state_i[j] = used$ ).

**Bounded Adaptive Mutex Algorithm** The corresponding algorithm is described in Fig. 10.10.

The code is nearly the same as in Fig. 10.7. The only modifications are the suppression of the management of the local clocks, the addition of the management of the permission states, and the appropriate modification of the priority predicate.

- Just before entering the critical section, a process  $p_i$  indicates that the permission it shares with any other process  $p_j$  is now used (line 5).
- When  $p_i$  receives (from  $p_j$ ) the message PERMISSION( $\{i, j\}$ ),  $p_i$  sets to *new* the state of this permission (line 17).
- The computation of the value of the Boolean  $prio_i$  is done as explained previously (the timestamp-based comparison  $\langle lrd_i, i \rangle < \langle k, j \rangle$  used in Fig. 10.7 is replaced by the predicate  $perm\_state_i[j] = new \vee j \in R_i$ , which is on bounded local variables).

**Adaptivity and Cost** As for the algorithm of Fig. 10.7, it is easy to see that the algorithm is adaptive and each use of the critical section costs  $2|R_i|$  messages, where the value of  $|R_i|$  is such that  $0 \leq |R_i| \leq n - 1$  and depends on the current state of the system. Moreover, the number of distinct messages is bounded.

As far as the local memory of a process  $p_i$  is concerned, we have the following: As in the previous algorithms,  $cs\_state_i$ ,  $R_i$ , and  $perm\_delayed_i$  are bounded, and so is  $perm\_state_i[1..n]$ , which is an array of one-bit values.

### 10.3.4 Proof of the Bounded Adaptive Mutex Algorithm

**An Acyclic Directed Graph** The following directed graph  $G$  (which evolves according to the requests issued by the processes) is central to the proof of the liveness property of the bounded adaptive algorithm. The vertices of  $G$  are the  $n$  processes. There is a directed edge from  $p_i$  to  $p_j$  (meaning that  $p_j$  has priority over  $p_i$ ) if:

- the message PERMISSION( $\{i, j\}$ ) is located at process  $p_i$  and  $perm\_state_i[j] = used$ , or
- the message PERMISSION( $\{i, j\}$ ) is in transit from  $p_i$  to  $p_j$ , or
- the message PERMISSION( $\{i, j\}$ ) is located at process  $p_j$  and  $perm\_state_j[i] = new$ .

It is easy to see that the initial values are such that there is a directed edge from  $p_i$  to  $p_j$  if and only if  $i < j$ . Hence, this graph is initially acyclic.

**Lemma 5** *The graph  $G$  always remains acyclic.*

*Proof* Let us observe that the only statement that can change the direction of an edge is (a) when a process uses the corresponding permission and (b) the previous state of this permission was *new*. This occurs at line 5. If  $perm\_state_i[j] = new$  before executing  $perm\_state_i[j] \leftarrow used$ , the execution of this statement changes the priority edge from  $p_j$  to  $p_i$  into an edge from  $p_i$  to  $p_j$  (i.e.,  $p_j$  then has priority with respect to  $p_i$ ).

It follows that, whatever the values of the local variables  $perm\_state_i[x]$  before  $p_i$  executes line 5, after it has executed this line, the edges adjacent to  $p_i$  are only outgoing edges. Consequently, no cycle involving  $p_i$  can be created by this statement. It follows that, if the graph  $G$  was acyclic before the execution of line 5, it remains acyclic. The fact that the graph is initially acyclic concludes the proof of the lemma.  $\square$

**Theorem 14** *The algorithm described in Fig. 10.10 solves the mutex problem.*

*Proof* Proof of the safety property. It follows from the initialization that, for any pair of processes  $p_i$  and  $p_j$ , we initially have either  $(i \in R_j) \wedge (j \notin R_i)$  or  $(j \in R_i) \wedge (i \notin R_j)$ . Then, when a process  $p_i$  sends a permission it adds the corresponding destination process to  $R_i$  (lines 7–8, or lines 12–13). Moreover, when it receives a permission from a process  $p_j$ ,  $p_i$  suppresses this process from  $R_i$ . It follows that there is always a single copy of each permission message.

Due to the waiting predicate of line 3, a process  $p_i$  has all the permissions it shares with each other process when it is allowed to enter the critical section. From then on, and until it executes `release_mutex()`, we have  $cs\_state_i = in$ . Hence, during this period, the Boolean variable  $prio_i$  cannot be false, and consequently,  $p_i$  does not send permissions. As permissions are not duplicated, and there is a single permission shared by any pair of processes, it follows that, while  $cs\_state_i = in$ , no process  $p_j$  has the message `PERMISSION({i, j})` that it needs to enter the critical section, which proves the safety property of the bounded adaptive mutex algorithm.

Proof of the liveness property. Considering a process  $p_i$  in the acyclic graph  $G$ , let  $height(i)$  be the maximal distance of a path from  $p_i$  to the process without outgoing edges. Let  $p_i$  be a process such that  $cs\_state_i = trying$ , and  $k = height(i)$ . The proof consists in showing, by induction on  $k$ , that eventually  $p_i$  is such that  $cs\_state_i = in$ .

Base case:  $k = 0$ . In this case,  $p_i$  has only incoming edges in  $G$ . Let us consider any other process  $p_j$ .

- It follows from the directed edge from  $p_j$  to  $p_i$  in  $G$  that, if the message `PERMISSION({i, j})` is at  $p_i$  (or in transit from  $p_j$  to  $p_i$ ), its state is (or will be) *new*. It then follows from the priority computed at line 10 that, even if it receives a request from  $p_j$ ,  $p_i$  keeps the permission until it invokes `release_mutex()`.
- If the message `PERMISSION({i, j})` is at  $p_j$  and  $p_j$  is such that  $cs\_state_j \neq in$ , it follows from the directed edge from  $p_j$  to  $p_i$  in  $G$  that  $perm\_state_i[j] = used$ . Hence,  $p_j$  sends the message `PERMISSION({i, j})` to  $p_i$ . As the state of this message is set to *new* when it arrives,  $p_i$  keeps it until it invokes `release_mutex()`.
- If the message `PERMISSION({i, j})` is at  $p_j$  and  $p_j$  is such that  $cs\_state_j = in$ , the previous item applies after  $p_j$  invoked `release_mutex()`.

It follows that  $p_i$  eventually obtains and keeps the  $(n - 1)$  messages, which allows it to enter the critical section (lines 3–4).

Induction case:  $k > 0$ . Let us assume that all the processes that are at a height  $\leq k - 1$ , eventually enter the critical section. Let  $p_j$  be a process whose height is  $k$ .

This process has incoming edges and outgoing edges. As far as the incoming edges are concerned, the situation is the same as in the base case, and  $p_j$  will eventually obtain and keep the corresponding permissions.

Let us now consider an outgoing edge from  $p_i$  to some process  $p_j$ . The height of  $p_j$  is  $\leq k - 1$ . Moreover, (a) the message  $\text{PERMISSION}(\{i, j\})$  is at  $p_i$  in the state *used*, or (b) is in transit from  $p_i$  to  $p_j$ , or (c) is at  $p_j$  in the state *new*. As the height of  $p_j$  is  $\leq k - 1$ , it follows from the induction assumption, that  $p_j$  eventually enters the critical section. When  $p_j$  does, the state of the permission becomes *used*, and the direction of the edge connecting  $p_i$  and  $p_j$  is then from  $p_j$  to  $p_i$ . Hence, the height of  $p_i$  becomes eventually  $\leq k - 1$ , and the theorem follows.  $\square$

## 10.4 An Algorithm Based on Arbiter Permissions

### 10.4.1 Permissions Managed by Arbiters

**Meaning of an Arbiter Permission** In an algorithm based on arbiter permissions (similarly to the algorithms based on individual permissions), each process  $p_i$  must ask the permission of each process in its request set  $R_i$ . As indicated in Sect. 10.1.2, the important difference lies in the meaning of a permission whose scope is no longer restricted to a pair of processes.

A permission has now a global scope. More precisely, let  $\text{PERMISSION}(i)$  be the permission managed by  $p_i$ . When it gives its permission to  $p_j$ ,  $p_i$  gives this permission, not on behalf of itself, but on behalf of all the processes that need this permission to enter the critical section. It follows that, when a process  $p_j$  exits the critical section, it has to send back to each process  $p_i$  such that  $i \in R_j$  the permission it has previously obtained from it. This is needed to allow  $p_i$  to later give the permission  $\text{PERMISSION}(i)$  it manages to another requesting process.

**Mutex Safety from Intersecting Sets** As a process gives its permission to only one process at a time, the safety property of the mutual exclusion problem is ensured if we have

$$\forall i, j: R_i \cap R_j \neq \emptyset.$$

This is because, as there is at least one process  $p_k$  such that  $k \in R_i \cap R_j$ , this process cannot give  $\text{PERMISSION}(k)$  to  $p_j$ , if it has sent it to  $p_i$  and  $p_i$  has not yet returned it. Such a process  $p_k$  is an *arbiter* for the conflicts between  $p_i$  and  $p_j$ . According to the definition of the sets  $R_i$  and  $R_j$ , the conflicts between the processes  $p_i$  and  $p_j$  can be handled by one or more arbiters.

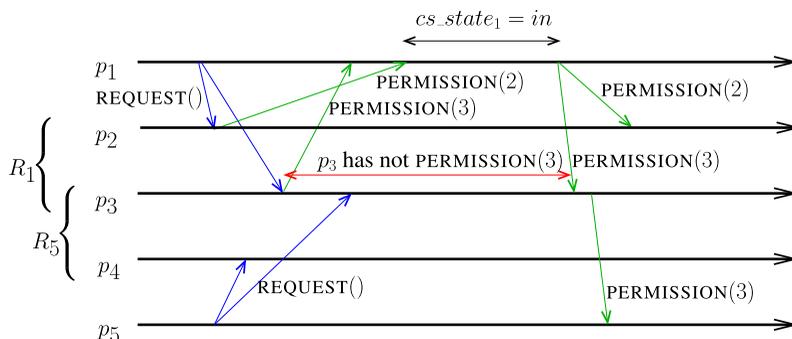


Fig. 10.11 Arbiter permission-based mechanism

**Example** This arbiter-based mechanism is depicted in Fig. 10.11, where there are five processes,  $p_1$ ,  $p_2$ ,  $p_3$ ,  $p_4$ , and  $p_5$ ,  $R_1 = \{2, 3\}$  and  $R_5 = \{3, 4\}$ . As  $3 \in R_1 \cap R_5$ , process  $p_3$  is an arbiter for solving the conflicts involving  $p_1$  and  $p_5$ . Hence, as soon as  $p_3$  has sent `PERMISSION(3)` to  $p_1$ , it cannot honor the request from  $p_5$ . Consequently,  $p_3$  enqueues this request in a local queue in order to be able to satisfy it after  $p_1$  has returned the message `PERMISSION(3)` to it. (A similar three-way handshake mechanism was described in Fig. 5.1, where the home process of a mobile object acts as an arbiter process.)

### 10.4.2 Permissions Versus Quorums

**Quorums** A *quorum system* is a set of pairwise intersecting sets. Each intersecting set is called a *quorum*. Hence, in mutual exclusion algorithms based on arbiter permissions, each set  $R_i$  is a quorum and the sets  $R_1, \dots, R_n$  define a quorum system.

**The Case of a Centralized System** An extreme case consists in defining a quorum system made up of a single quorum containing a single process, i.e.,

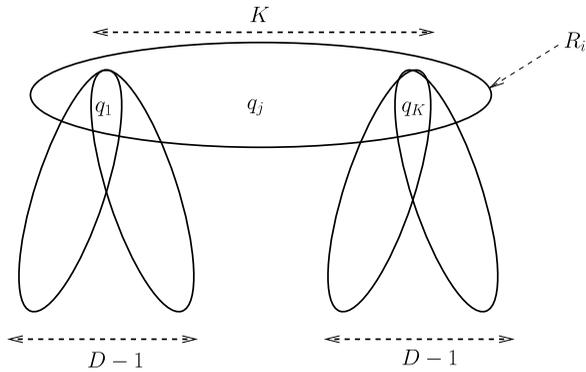
$$\forall i: R_i = \{k\}.$$

In this case,  $p_k$  arbitrates all the processes, and the control is consequently centralized. (This corresponds to the home-based solution for a mobile object—token—navigating a network; see Chap. 5.)

**Constraints on the Definition of Quorums** Ideally, a quorum system should be symmetric and optimal, i.e., such that

- All quorums have the size  $K$ , i.e.,  $\forall i: |R_i| = K$ . This is the “equal effort rule”: All the processes need the same number of permissions to enter the critical section.

**Fig. 10.12** Values of  $K$  and  $D$  for symmetric optimal quorums



- Each process  $p_i$  belongs to the same number  $D$  of quorums i.e.,  $\forall i: |\{j \mid i \in R_j\}| = D$ . This is the “equal responsibility rule”: All the processes are engaged in the same number of quorums.
- $K$  and  $D$  have to be as small as possible. (Of course, a solution in which  $\forall i: R_i = \{1, \dots, n\}$  works, but we would then have  $K = D = n$ , which is far from being optimal.)

The two first constraints are related to symmetry, while the third one is on the optimality of the quorum system.

### 10.4.3 Quorum Construction

**Optimal Values of  $K$  and  $D$**  Let us observe that the previous symmetry and optimality constraints on  $K$  and  $D$  link these values. More precisely, as both  $nK$  and  $nD$  represent the total number of possible arbitrations, the relation  $K = D$  follows.

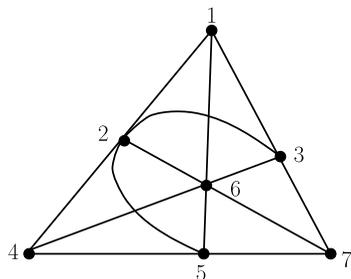
To compute their smallest value, let us count the greatest possible number of different sets  $R_i$  that can be built. Let us consider a set  $R_i = \{q_1, \dots, q_K\}$  (all  $q_j$  are distinct and  $q_j$  is not necessarily  $p_j$ ). Due to the definition of  $D$ , each  $q_j$  belongs to  $R_i$  and  $(D - 1)$  other distinct sets. Hence, an upper bound on the total number of distinct quorums that can be built is  $1 + K(D - 1)$ . (The value 1 comes from the initial set  $R_i$ , the value  $K$  comes from the size of  $R_i$ , and the value  $D - 1$  is the number of quorums to which each  $q_j$  belongs—in addition to  $R_i$ —see Fig. 10.12.) As there is a quorum per process and there are  $n$  processes, we have

$$n = K(K - 1) + 1.$$

It follows that the lower bound on  $K$  and  $D$ , which, satisfies both the symmetry and optimality constraints, is  $K = D \simeq \sqrt{n}$ .

**Finite Projective Planes** Find  $n$  sets  $R_i$  satisfying  $K = D \simeq \sqrt{n}$  amounts to find a finite projective plane of  $n$  points. There exists such planes of order  $k$  when  $k$  is

**Fig. 10.13** An order two projective plane



**Table 10.1** Defining quorums from a  $\sqrt{n} \times \sqrt{n}$  grid

$$\begin{pmatrix} 12 & 8 & 5 & 9 \\ 6 & 2 & 13 & 1 \\ 10 & 3 & 4 & 7 \\ 14 & 11 & 8 & 6 \end{pmatrix}$$

power of a prime number. Such a plane has  $n = k(k + 1) + 1$  points and the same number of lines. Each point belongs to  $(k + 1)$  distinct lines, and each line is made up of  $(k + 1)$  points. Two distinct points share a single line, and two distinct lines meet a single point. A projective plane with  $n = 7$  points (i.e.,  $k = 2$ ) is depicted in Fig. 10.13. (The points are marked with a black bullet. As an example, the lines “1, 6, 5” and “3, 2, 5” meet only at the point denoted “5”.) A line defines a quorum.

Being optimal, any two quorums (lines) defined from finite projective planes have a single process (point) in common. Unfortunately, there are not finite projective planes for any value of  $n$ .

**Grid Quorums** A simple way to obtain quorums of size  $O(\sqrt{n})$ , consists in arbitrarily placing the processes in a square grid. If  $n$  is not a square,  $(\lceil \sqrt{n} \rceil)^2 - n$  arbitrary processes can be used several times to complete the grid. An example with  $n = 14$  processes is given in Table 10.1. As  $(\lceil \sqrt{14} \rceil)^2 - 14 = 2$ , two processes are used twice to fill the grid (namely,  $p_5$  and  $p_8$  appear twice in the grid).

A quorum  $R_i$  consists then of all the processes in a line plus one process per column. As an example the set  $\{6, 2, 13, 1, 8, 4, 14\}$  constitutes a quorum. As any quorum includes a line of the grid, it follows from their construction rule that any two quorums intersect. Moreover, due to the grid structure, and according to the value of  $n$ , we have  $\lceil \sqrt{n} \rceil \leq |R_i| \leq 2\lceil \sqrt{n} \rceil - 1$ .

**Quorums and Antiquorums** Let  $\mathcal{R} = \{R_1, \dots, R_n\}$  be a set of quorums. An *antiquorum* (with respect to  $\mathcal{R}$ ) is a set  $R'$  such that  $\forall i: R' \cap R_i \neq \emptyset$ . Let us observe that two antiquorums are not required to intersect.

An arbiter permission-based system that solves the readers/writers problem can be easily designed as follows. A quorum  $R_i$  and an antiquorum  $R'_i$  are associated with each process  $p_i$ . The quorum  $R_i$  defines the set of processes from which  $p_i$  must obtain the permission in order to write, while the antiquorum  $R'_i$  defines the set of processes from which it must obtain the permission in order to read.

When considering grid-based quorums, an antiquorum can be defined as composed of all the process in a column. The size of such an antiquorum is consequently  $\lceil \sqrt{n} \rceil$ . It is easy to see that we have then for all pairs  $(i, j)$ :

- $R_i \cup R_j \neq \emptyset$ , which ensures mutual exclusion among each pair of concurrent write operations.
- $R'_i \cup R_j \neq \emptyset$ , which ensures mutual exclusion among each pair of concurrent read and write operations.

It is also easy to see that the square grid structure can be replaced by a rectangle. When this rectangle becomes a vector (one line and  $n$  columns) a write operation needs all permissions, while a read operation needs a single permission. This extreme case is called ROWA (Read One, Write All).

**Crumbling Walls** In a *crumbling wall*, the processes are arranged in several lines of possibly different lengths (hence, all quorums will not have the same size). A quorum is then defined as a full line, plus a process from every line below this full line.

A triangular quorum system is a crumbling wall in which the processes are arranged in such a way the  $\ell$ th line has  $\ell$  processes (except possibly the last line).

**Vote-Based Quorums** Quorums can also be defined from weighted votes assigned to processes, which means that each process has a weighted permission. A vote is nothing more than a permission. Let  $S$  be the sum of the weight of all votes. A quorum is then a set of processes whose sum of weighted votes is greater than  $S/2$ . This vote system is called *majority voting*.

#### 10.4.4 An Adaptive Mutex Algorithm Based on Arbiter Permissions

The algorithm that is presented in this section is a variant of an algorithm proposed by M. Maekawa (1985). The introduction of quorums of size  $\sqrt{n}$  (defined from projective planes and grids) is also due to M. Maekawa.

**Two Simplifying Assumptions** To simplify the presentation, the channels are assumed to be FIFO. Moreover, while in practice it is interesting to have  $i \in R_i$  (to save messages), we assume here that  $i \notin R_i$ . This simplification allows for a simpler explanation of the behavior of  $p_i$  as a client (which wants to enter the critical section and, to that end, asks for and releases permissions), and its behavior as a server (which implements an arbiter by managing and granting a permission).

**On the Safety Side** This section describes a first sketch of an algorithm where the focus is only on the safety property. This safe algorithm is described in Fig. 10.14. The meaning of the local variables is the same as in the previous permission-based algorithms. An empty queue is denoted  $\emptyset$ .

```

operation acquire_mutex() is
(1)  $cs\_state_i \leftarrow trying;$ 
(2)  $waiting\_from_i \leftarrow R_i;$ 
(3) for each  $j \in R_i$  do send REQUEST() to  $p_j$  end for;
(4) wait ( $waiting\_from_i = \emptyset$ );
(5)  $cs\_state_i \leftarrow in.$ 

when PERMISSION( $j$ ) is received from  $p_j$  do %  $j \in R_i$  %
(6)  $waiting\_from_i \leftarrow waiting\_from_i \setminus \{j\}.$ 

operation release_mutex() is
(7)  $cs\_state_i \leftarrow out;$ 
(8) for each  $j \in R_i$  do send PERMISSION( $j$ ) to  $p_j$  end for.

when REQUEST() is received from  $p_j$  do %  $i \in R_j$  %
(9) if ( $perm\_here_i$ ) then send PERMISSION( $i$ ) to  $p_j$ ;
(10)  $perm\_here_i \leftarrow false$ 
(11) else append  $p_j$ 's request to  $queue_i$ ;
(12) end if.

when PERMISSION( $i$ ) is received from  $p_j$  do %  $i \in R_j$  %
(13) withdraw  $p_j$ 's request from  $queue_i$ ;
(14) if ( $queue_i \neq \emptyset$ ) then let  $p_k$  be the process at the head of  $queue_i$ ;
(15) send PERMISSION( $i$ ) to  $p_k$ 
(16) end if.

```

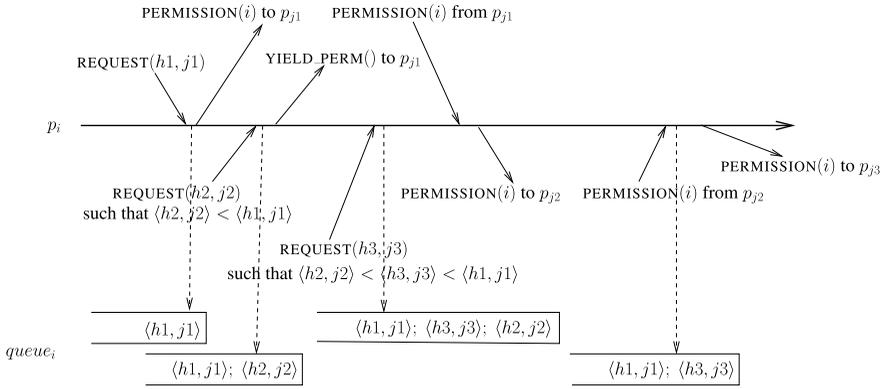
**Fig. 10.14** A safe (but not live) mutex algorithm based on arbiter permissions (code for  $p_i$ )

On the client side we have the following (lines 1–8). When a process invokes `acquire_mutex()`, it sends a request to each process  $p_j$  of its request set  $R_i$  to obtain the permission `PERMISSION( $j$ )` managed by  $p_j$  (line 3). Then, when it has received all the permissions (line 6 and line 4),  $p_i$  enters the critical section (line 5). When, it releases the critical section,  $p_i$  returns each permission to the corresponding arbiter process (line 8).

On the arbiter side, the behavior of  $p_i$  is as follows (lines 9–16). The local Boolean variable  $perm\_here_i$  is equal to *true* if and only if  $p_i$  has the permission it manages (namely, `PERMISSION( $i$ )`).

- When it receives a request from a process  $p_j$  (we have then  $i \in R_j$ ),  $p_i$  sends it the permission it manages (`PERMISSION( $i$ )`) if it has this permission (lines 9–10). Otherwise, it adds the request of  $p_j$  to a local queue (denoted  $queue_i$ ) in order to serve it later.
- When  $p_i$  is returned its permission from a process  $p_j$ , it suppresses  $p_j$ 's request from its queue  $queue_i$ , and, if this queue is not empty, it sends `PERMISSION( $i$ )` to the first process of this queue. Otherwise, it keeps its permission until a process  $p_j$  (such that  $j \in R_j$ ) requests it.

**The Liveness Issue** Due to the quorum intersection property ( $\forall i, j: R_i \cap R_j \neq \emptyset$ ) and the management of permissions (which are returned to their managers after being used), the previous algorithm satisfies the safety property of the mutex problem.



**Fig. 10.15** Permission preemption to prevent deadlock

Unfortunately, this algorithm is deadlock-prone. As a simple case, let us consider that both  $p_{j_1}$  and  $p_{j_2}$  are such that  $i_1, i_2 \in R_{j_1} \cap R_{j_2}$ . If  $p_{i_1}$  gives its permission to  $p_{j_1}$  while concurrently  $p_{i_2}$  gives its permission to  $p_{j_2}$ , none of  $p_{j_1}$  and  $p_{j_2}$  will obtain both permissions and will wait forever for the other permission.

Such a deadlock scenario may occur even if all the intersections of the sets  $R_i$  contain a single process. To that, it is sufficient to consider a system of three processes such that  $R_1 = \{2, 3\}$ ,  $R_2 = \{1, 3\}$ ,  $R_3 = \{1, 2\}$ , and an execution in which  $p_1$  gives its permission to  $p_2$ ,  $p_2$  gives its permission to  $p_3$ , and  $p_3$  gives its permission to  $p_1$ . When this scenario occurs, no process can progress.

**Solving the Liveness Issue, Part 1: Using Timestamps** Two additional mechanisms are introduced to solve the liveness issue. The first is a simple scalar clock system that permits us to associate a timestamp with each request issued by a process. The local clock of a process  $p_i$ , denoted  $clock_i$ , is initialized to 0. As we saw in Chap. 7 and Sect. 10.2.1, this allows requests to be totally ordered, and consequently provides us with a simple way to establish priority among conflicting requests.

**Solving the Liveness Issue, Part 2: Permission Preemption** As a process is an arbiter for a subset of processes and not for all processes, it is possible that a process  $p_i$  gives its permission to a process  $p_{j_1}$  (whose request its timestamped  $\langle h1, j1 \rangle$ ), and while it has not yet been returned its permission, it receives a request timestamped  $\langle h2, j2 \rangle$ , such that  $\langle h2, j2 \rangle < \langle h1, j1 \rangle$ . As the request from  $p_{j_2}$  has a smaller timestamp, it has priority over the request of  $p_{j_1}$ .

In that case,  $p_i$  asks  $p_{j_1}$  to yield the permission in order to prevent a possible deadlock. This is described in Fig. 10.15.

- $p_i$  receives first a request message from  $p_{j_1}$ , timestamped  $\langle h1, j1 \rangle$ . As  $queue_i = \emptyset$ ,  $p_i$  sends its permission to  $p_{j_1}$  and adds  $\langle h1, j1 \rangle$  to  $queue_i$ .
- Then  $p_i$  receives a request message from  $p_{j_2}$ , timestamped  $\langle h2, j2 \rangle$  and such that  $\langle h2, j2 \rangle < \langle h1, j1 \rangle$ .  $p_i$  then adds  $\langle h2, j2 \rangle$  to  $queue_i$  (which is always ordered

according to timestamp order, with the smallest timestamp at its head), and sends the message `YIELD_PERM()` to  $p_{j1}$  in order to be able to serve the request with the highest priority (here the request from  $p_{j2}$ , which has the smallest timestamp).

- Then  $p_i$  receives a request message from  $p_{j3}$ . The timestamp of this message  $\langle h3, j3 \rangle$  is such that  $\langle h2, j2 \rangle < \langle h3, j3 \rangle < \langle h1, j1 \rangle$ . As this timestamp is not smaller than the timestamp at the head of the queue,  $p_i$  only inserts it in  $queue_i$ .
- When  $p_i$  receives `PERMISSION( $i$ )` from  $p_{j1}$ ,  $p_i$  forwards it to the process whose request is at the head of its queue (namely  $p_{j2}$ ).
- Finally, when  $p_{j2}$  returns its permission to  $p_i$ ,  $p_i$  forwards it to the process which is at the head of its queue, namely  $p_{j3}$ .

The resulting algorithm, which is an extension of the safe algorithm of Fig. 10.14, is described in Fig. 10.16. On the client side we have the following:

- When a process  $p_i$  invokes `acquire_mutex()`, it increases its local clock (line 3), and sends a message timestamped  $\langle clock_i, i \rangle$  (line 4).
- When  $p_i$  receives a permission with a date  $d$ ,  $p_i$  updates its local clock (line 7) and its waiting set  $waiting\_from_i$  (line 8).
- When a process  $p_i$  invokes `release_mutex()`, it returns its permission to each process of the set  $R_i$  (line 10).
- When  $p_i$  receives a message `YIELD_PERM()` from one of its arbiters  $p_j$ , it returns its permission to  $p_j$  only if its local state is such that  $cs\_state_i = trying$  (lines 21–23). In this case, as it has no longer the permission of  $p_j$ , it also adds  $j$  to  $waiting\_from_i$ .

If  $cs\_state_i = in$  when  $p_i$  receives a message `YIELD_PERM()` from one of its arbiter  $p_j$ , it does nothing. This is because, as it is in the critical section, it will return  $p_j$ 's permission when it will invoke `release_mutex()`. Let us finally notice that it is not possible to have  $cs\_state_i = out$  when  $p_i$  receives a message `YIELD_PERM()`. This is because, if  $p_i$  receives such a message, it is because it has previously received the corresponding permission (let us recall that the channels are FIFO).

On the arbiter side, the behavior of  $p_i$  is defined by the statements it executes when it receives a message `REQUEST()` or `RETURNED_PERM()`, or when a process sends it back its the message `PERMISSION( $i$ )`.

- When  $p_i$  receives a message `REQUEST( $d$ )` from a process  $p_j$ , it first updates its clock (line 11) and adds the corresponding timestamp  $\langle d, j \rangle$  to  $queue_i$  (line 12).

Then, if `PERMISSION( $i$ )` is here,  $p_i$  sent it by return to  $p_j$  (lines 13–14). In this case, the permission message carries the current value of  $clock_i$ . This is to allow the client  $p_j$  to update its local clock (line 7). In this way, thanks to the intersection property of the sets  $R_i$  and  $R_j$ , the local clocks of the processes are forced to progress (and this global progress of local clocks allows each request to eventually have the smallest timestamp).

If `PERMISSION( $i$ )` is not here,  $p_i$  has sent this permission message to some process  $p_k$  (whose identity has been saved in  $sent\_to_i$ ). If  $\langle d, j \rangle$  is the smallest timestamp in  $queue_i$  and  $p_i$  has not reclaimed its permission to  $p_k$ ,  $p_i$  reclaims it (lines 15–18).

```

operation acquire_mutex() is
(1)  $cs\_state_i \leftarrow \text{trying}$ ;
(2)  $waiting\_from_i \leftarrow R_i$ ;
(3)  $clock_i \leftarrow clock_i + 1$ ;
(4) for each  $j \in R_i$  do send REQUEST( $clock_i$ ) to  $p_j$  end for;
(5) wait ( $waiting\_from_i = \emptyset$ );
(6)  $cs\_state_i \leftarrow in$ .

when PERMISSION( $j, d$ ) is received from  $p_j$  do %  $j \in R_i$  %
(7)  $clock_i \leftarrow \max(clock_i, d)$ ;
(8)  $waiting\_from_i \leftarrow waiting\_from_i \setminus \{j\}$ .

operation release_mutex() is
(9)  $cs\_state_i \leftarrow out$ ;
(10) for each  $j \in R_i$  do send PERMISSION( $j$ ) to  $p_j$  end for.

when REQUEST( $d$ ) is received from  $p_j$  do
(11)  $clock_i \leftarrow \max(clock_i, d)$ ;
(12) add the timestamp  $\langle d, j \rangle$  to  $queue_i$  and sort it according to timestamp order;
(13) if ( $perm\_here_i$ ) then send PERMISSION( $i, clock_i$ ) to  $p_j$ ;
(14)  $sent\_to_i \leftarrow j$ ;  $perm\_here_i \leftarrow false$ 
(15) else if ( $((d, j) = \text{head of } queue_i) \wedge (\neg perm\_asked_i)$ )
(16) then let  $k = sent\_to_i$ ;
(17) send YIELD_PERM() to  $p_k$ ;
(18)  $perm\_asked_i \leftarrow true$ 
(19) end if
(20) end if.

when YIELD_PERM() is received from  $p_j$  do %  $j \in R_i$  %
(21) if ( $cs\_state_i = \text{trying}$ )
(22) then send RETURNED_PERM( $j$ ) to  $p_j$ ;  $wait\_from_i \leftarrow wait\_from_i \cup \{j\}$ 
(23) end if.

when RETURNED_PERM( $i$ ) is received from  $p_j$  do %  $j \in R_i$  %
(24) let  $\langle -, k \rangle = \text{head of } queue_i$ ;
(25) send PERMISSION( $i$ ) to  $p_k$ ;
(26)  $sent\_to_i \leftarrow k$ ;  $perm\_asked_i \leftarrow false$ .

when PERMISSION( $i$ ) is received from  $p_j$  do %  $j \in R_i$  %
(27) withdraw  $\langle -, j \rangle$  from  $queue_i$ ;
(28) if ( $queue_i \neq \emptyset$ ) then let  $\langle -, k \rangle = \text{head of } queue_i$ ;
(29) send PERMISSION( $i, clock_i$ ) to  $p_k$ ;
(30)  $sent\_to_i \leftarrow j$ ;  $perm\_asked_i \leftarrow false$ 
(31) end if.

```

**Fig. 10.16** A mutex algorithm based on arbiter permissions (code for  $p_i$ )

- When  $p_i$  receives RETURNED\_PERM() from  $p_j$ , it forwards the message PERMISSION( $i$ ) to the process at the head of the queue, which is the requesting process with the smallest timestamp (lines 24–26). As the request of  $p_j$  remains pending,  $p_i$  does not suppress  $p_j$ 's timestamp from  $queue_i$ .

- When  $p_i$  receives from  $p_j$  the message `PERMISSION( $i$ )` it manages,  $p_i$  first suppresses  $p_j$ 's timestamp from  $queue_i$  (lines 27). Then, if  $queue_i$  is not empty,  $p_i$  sends its permission (with its local clock value) to the first process in  $queue_i$  (lines 27–31).

**Message Cost of the Algorithm** The number of messages generated by one use of the critical section depends on the current system state. In the best case, a single process  $p_i$  wants to use the critical section, and consequently  $3|R_i|$  messages are used. When several processes are competing, the *average number* of messages per use of the critical sections can be up to  $6|R_i|$ . This is due to the following observations. First, a `YIELD_PERM()` message can entail the sending of a `RETURNED_PERM()` message and, in that case, the permission message will have to be sent again to the yielding process. Second, it is possible that several `YIELD_PERM()` messages be (sequentially) sent to the same process  $p_i$  by the same arbiter  $p_j$  (each other process  $p_k$  such that  $k \in R_j$  can entail such a scenario once).

## 10.5 Summary

The aim of this chapter was to present the mutual exclusion problem and the class of permission-based algorithms that solve it. Two types of permission-based algorithms have been presented, algorithms based on individual permissions and algorithms based on arbiter permissions. They differ in the meaning of a permission. In the first case a permission engages only its sender, while it engages a set of processes in the second case. Arbiter-based algorithms are also known under the name quorum-based algorithms. An aim of the chapter was to show that the concept of a permission is a fundamental concept when one has to solve exclusion problems. The notions of an adaptive algorithm and a bounded algorithm have also been introduced and illustrated.

The algorithms presented in Chap. 5 (devoted to mobile objects navigating a network) can also be used to solve the mutex problem. In this case, the mobile object is usually called a token, and the corresponding algorithms are called token-based mutex algorithms.

## 10.6 Bibliographic Notes

- The mutual exclusion problem was introduced in the context of shared memory systems by E.W. Dijkstra, who presented its first solution [109].
- One of the very first solutions to the mutex problem in message-passing systems is due to L. Lamport [226]. This algorithm is based on a general state-machine replication technique. It requires  $3n$  messages per use of the critical section.

- The mutex problem has received many solutions both in shared memory systems (e.g., see the books [317, 362]) and in message-passing systems [306]. Surveys and taxonomies of message-passing mutex algorithms are presented in [310, 333, 349].
- The algorithm presented in Sect. 10.2.1 is due to R. Ricart and A.K. Agrawala [327].
- The unbounded adaptive algorithm presented in Sect. 10.3.2 is due to O. Carvalho and G. Roucairol [71]. This algorithm was obtained from a Galois field-based systematic distribution of an assertion [70].
- The bounded adaptive algorithm presented in Sect. 10.3.3 is due to K.M. Chandy and J. Misra [78]. This is one of the very first algorithms that used the notion of edge reversal to maintain a dynamic cycle-free directed graph. A novelty of this paper was the implementation of edge reversals with the notion of a *new/used* permission.
- The algorithm based on arbiter permissions is due to M. Maekawa [243], who was the first to define optimal quorums from finite projective planes.
- The notion of a quorum was implicitly introduced by R.H. Thomas and D.K. Gifford in 1979, who were the first to introduce the notion of a vote to solve resource allocation problems [159, 368].

The mathematics which underly quorum and vote systems are studied in [9, 42, 147, 195]. The notion of an anti-quorum is from [41, 147]. Tree-based quorums were introduced in [7]. Properties of crumbling walls are investigated in [294]. Availability of quorum systems is addressed in [277]. A general method to define quorums is presented in [281]. A monograph on quorum systems was recently published [380].

- Numerous algorithms for message-passing mutual exclusion have been proposed. See [7, 68, 226, 239, 282, 348] to cite a few. A mutex algorithm combining individual permissions and arbiter permissions is presented in [347]. An algorithm for arbitrary networks is presented in [176].

## 10.7 Exercises and Problems

1. Let us consider the mutex algorithm based on individual permission described in Sect. 10.2. The aim is here to prevent the phenomenon depicted in Fig. 10.5. More precisely, if  $p_i$  and  $p_j$  are conflicting and  $p_i$  wins (i.e., the timestamp associated with the request of  $p_i$  is smaller than the timestamp associated with the request of  $p_j$ ), then  $p_j$  has to enter the critical section before the next invocation of  $p_i$ .
  - Modify the algorithm so that it satisfies the previous sequencing property.
  - Obtain the sequencing property by adding a simple requirement on the behavior of the channels (and without modifying the algorithm).
  - Which of the previous solutions is the most interesting? Why?

2. When considering the mutex algorithm based on individual permission described in Sect. 10.2, each use of the critical section by a process requires  $2(n - 1)$  messages. Modify this algorithm in such a way that each use of the critical section requires exactly  $n$  messages when the fully connected network is replaced by a unidirectional ring network.  
Solution in [327].
3. When considering the mutex algorithm based on individual permission described in Sect. 10.2, show that the relation  $|clock_i - clock_j| \leq n - 1$  is invariant. Exploit this property to obtain an improved algorithm in which the clock value domain is bounded by  $2n - 1$ .  
Solution in [327].
4. Rewrite the bounded adaptive mutex algorithm described in Fig. 10.10, in such a way that the identities of the processes (which have a global meaning) are replaced by channel identities (which have a local meaning).
5. When considering the bounded adaptive mutex algorithm described in Fig. 10.10, can the predicate “ $j \in R_i$ ” (used to compute the priority at line 10) be suppressed when the channels are FIFO?
6. When considering the mutex algorithm based on arbiter permissions described in Fig. 10.16, is the process identity  $j$  or  $i$  necessary in the following message received by  $p_i$ : PERMISSION( $j, d$ ), RETURN\_PERM( $i$ ), and PERMISSION( $i$ ).
7. To simplify its presentation, the algorithm described in Fig. 10.16 assumes that  $i \notin R_i$ . To save messages, it is interesting to always have  $i \in R_i$ . Modify the algorithm so that it works when, for any  $i$ , we have  $i \in R_i$ .

# Chapter 11

## Distributed Resource Allocation

This chapter is on resource allocation in distributed systems. It first considers the case where there are  $M$  instances of the same resource, and a process may request several instances of it. The corresponding resource allocation problem is called *k-out-of-M* problem (where  $k$ ,  $1 \leq k \leq M$ , stands for the—dynamically defined—number of instances requested by a process). Then, the chapter addresses the case where there are several resources, each with a single or several instances.

The multiplicity of resources may generate deadlocks if resources are arbitrarily allocated to processes. Hence, the chapter visits deadlock prevention techniques suited to resource allocation. It also introduces the notion of a conflict graph among processes. Such a graph is a conceptual tool, which captures the possible conflicts among processes, when each resource can be accessed by a subset of processes. Finally, the chapter considers two distinct cases, according to the fact that the subset of resources required by a process is always the same, or may vary from one resource session to another one.

**Keywords** Conflict graph · Deadlock prevention · Graph coloring · Incremental requests · *k-out-of-M* problem · Permission · Resource allocation · Resource graph · Resource type · Resource instance · Simultaneous requests · Static/dynamic (resource) session · Timestamp · Total order · Waiting chain · Wait-for graph

### 11.1 A Single Resource with Several Instances

#### 11.1.1 The *k-out-of-M* Problem

The mutual exclusion problem considers the case of a single resource with a single instance. A simple generalization is when there are  $M$  instances of the same resource, and a process may require several instances of it. More precisely, (a) each instance can be used by a single process at a time (mutual exclusion), and (b) each time it issues a request, a process specifies the number  $k$  of instances that it needs (this number is specific to each request).

Solving the *k-out-of-M* problem consists in ensuring that, at any time:

- no resource instance is accessed by more than one process,

- each process is eventually granted the number of resource instances it has requested, and
- when possible, resource instances have to be granted simultaneously. This last requirement is related to efficiency (if  $M = 5$  and two processes asks for  $k = 2$  and  $k' = 3$  resource instances, respectively, and no other process is using or requesting resource instances, these two processes must be allowed to access them simultaneously).

### 11.1.2 Mutual Exclusion with Multiple Entries: The 1-out-of- $M$ Mutex Problem

The 1-out-of- $M$  problem is when every request of a process is always for a single resource instance. This problem is also called *mutual exclusion with multiple entries*.

This section presents an algorithm that solves the 1-out-of- $M$  problem. This algorithm, which is due to K. Raymond (1989), is a straightforward adaptation of the mutex algorithm based on individual permission described in Sect. 10.2.

**A Permission-Based Predicate** As in Sect. 10.2, a process  $p_i$  that wants to acquire an instance of the resource, asks for the individual permission of each other process  $p_j$ . As there are  $M$  instances of the resource,  $p_i$  has to wait until it knows that at most  $(M - 1)$  other processes are using a resource instance. This translates into the waiting statement: Wait until  $(n - 1) - (M - 1) = n - M$  permissions have been received.

**Management of Individual Permissions** When a process has received  $(n - M)$  permissions, it is allowed to use an instance of the resource. The  $(M - 1)$  permissions that have not yet arrived, will be received later, i.e., while  $p_i$  is using the resource instance it has obtained, after it has used it, or even during a new use of a resource instance.

Moreover, let us notice that a process  $p_i$  may delay the sending of several permissions with respect to another process  $p_j$ . As an example, let us consider three processes,  $p_1$ ,  $p_2$ , and  $p_3$ , and two instances of a resource ( $M = 2$ ). Process  $p_1$  obtains  $n - M = 1$  permission (from  $p_2$  or  $p_3$ ) and uses a resource instance for a very long period of time. During this period of time, process  $p_2$  may use the other resource instance several times due to the permission sent by  $p_3$  (which is not interested in the resource). In this scenario, process  $p_1$  receives several requests from  $p_2$  that it will answer only when it will release its instance of the resource.

**Local Variables of a Process  $p_i$**  In addition to the local variables  $cs\_state_i$ ,  $clock_i$ ,  $lrd_i$ , and the constant set  $R_i = \{1, \dots, n\} \setminus \{i\}$ , and according to the previous discussion on the permissions sent and received by the processes, each process  $p_i$  manages two local arrays of non-negative integers (both initialized to  $[0, \dots, 0]$ ), and a non-negative integer variable.

```

operation acquire_resource() is
(1)  $cs\_state_i \leftarrow trying$ ;
(2)  $\ell rd_i \leftarrow clock_i + 1$ ;
(3)  $nb\_perm_i \leftarrow 0$ ;
(4) for each  $j \in R_i$  do send REQUEST( $\ell rd_i, i$ ) to  $p_j$ ;
(5)  $wait\_perm_i[j] \leftarrow wait\_perm_i[j] + 1$ 
(6) end for;
(7) wait ( $nb\_perm_i \geq n - M$ );
(8)  $cs\_state_i \leftarrow in$ .

operation release_resource() is
(9)  $cs\_state_i \leftarrow out$ ;
(10) foreach  $j$  such that  $perm\_delayed_i[j] \neq 0$  do
(11) send PERMISSION( $i, perm\_delayed_i[j]$ );  $perm\_delayed_i[j] \leftarrow 0$ 
(12) end for.

when REQUEST( $k, j$ ) is received do
(13)  $clock_i \leftarrow \max(clock_i, k)$ ;
(14)  $prio_i \leftarrow (cs\_state_i = in) \vee [(cs\_state_i = trying) \wedge (\langle \ell rd_i, i \rangle < \langle k, j \rangle)]$ ;
(15) if ( $prio_i$ ) then  $perm\_delayed_i[j] \leftarrow perm\_delayed_i[j] + 1$ 
(16) else send PERMISSION( $i, 1$ ) to  $p_j$ 
(17) end if.

when PERMISSION( $j, x$ ) is received do
(18)  $wait\_perm_i[j] \leftarrow wait\_perm_i[j] - x$ ;
(19) if ( $(cs\_state_i = trying) \wedge (wait\_perm_i[j] = 0)$ ) then  $nb\_perm_i \leftarrow nb\_perm_i + 1$  end if.

```

**Fig. 11.1** An algorithm for the multiple entries mutex problem (code for  $p_i$ )

- $wait\_perm_i[1..n]$  is an array such that  $wait\_perm_i[j]$  contains the number of permissions that  $p_i$  is waiting on from  $p_j$ .
- $perm\_delayed_i[1..n]$  is an array such that  $perm\_delayed_i[j]$  contains the number of permissions that  $p_i$  has to send to  $p_j$  when it will release the resource instance it is currently using.
- $nb\_perm_i$ , which is meaningful only when  $p_i$  is waiting for permission, contains the number of permissions received so far by  $p_i$  (i.e.,  $nb\_perm_i = |\{j \neq i \text{ such that } wait\_perm_i[j] = 0\}|$ ).

**Behavior of a Process  $p_i$**  The algorithm based on individual permission that solves the multiple entries mutual exclusion problem is described in Fig. 11.1. As already indicated, this algorithm is a simple extension of the mutex algorithm described in Fig. 10.3. The following comments are only on the modified parts of this algorithm.

When a process  $p_i$  invokes `acquire_resource()`, it sends a timestamped request message to each other process  $p_j$  in order to obtain its permission, and updates accordingly  $wait\_perm_i[j]$  (lines 4–5). Then,  $p_i$  waits until it has received  $(n - M)$  permissions (line 7).

When a process  $p_i$  invokes `release_resource()`, it sends to each process  $p_j$  all the permissions whose sending has been delayed. If several permissions have been delayed with respect to  $p_j$ , they are sent in a single message (line 11).

When  $p_i$  receives a message `REQUEST( $k, j$ )`,  $p_i$  has priority if it is currently using an instance of the resource ( $cs\_state_i = in$ ), or it is waiting and its current request has a smaller timestamp than the one it just received (line 14). If  $p_i$  has priority, it increases accordingly  $perm\_delayed_i[j]$  (line 15), otherwise it sends its permission by return (line 16).

Finally, when  $p_i$  receives a message `PERMISSION( $j, x$ )`, it updates accordingly  $wait\_perm_i[j]$  (line 18). Moreover, if  $p_i$  is waiting to obtain a resource instance and it is no longer waiting permissions from  $p_j$ ,  $p_i$  increases  $nb\_perm_i$  (line 19), which may allow it to use an instance of the resource if  $nb\_perm_i \geq n - M$  (line 7).

**Message Cost of the Algorithm** There are always  $(n - 1)$  request messages per use of a resource instance, while the number of permission messages depends on the system state and varies between  $(n - 1)$  and  $(n - M)$ . The total number of messages per use of a resource instance is consequently at most  $2(n - 1)$  and at least  $2n - (M + 1)$ .

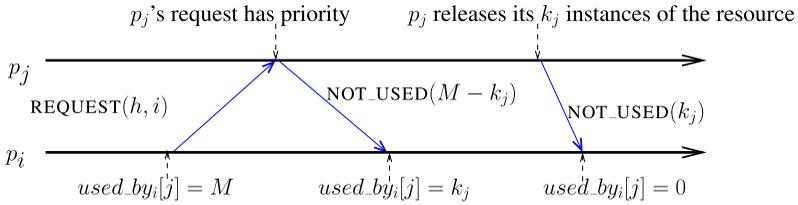
### 11.1.3 An Algorithm for the $k$ -out-of- $M$ Mutex Problem

**From 1-out-of- $M$  to  $k$ -out-of- $M$**  As previously indicated, in the  $k$ -out-of- $M$  problem, there are  $M$  instances of the same resource type and, each time a process  $p_i$  invokes `acquire_resource()`,  $p_i$  passes as an input parameter the number  $k$  of instances it wants to acquire. The parameter  $k$ ,  $1 \leq k \leq M$ , is specific to each instance (which means that there is no relation linking the parameters  $k$  and  $k'$  of any two invocations `acquire_resource( $k$ )` and `acquire_resource( $k'$ )`).

An algorithm solving the  $k$ -out-of- $M$  problem can be obtained from an appropriate modification of a mutex algorithm. As for the 1-out-of- $M$  problem studied in the previous section, we consider here the mutex algorithm based on individual permissions described in Sect. 10.2. The resulting algorithm, which is described below, is due to M. Raynal (1991).

**Algorithmic Principles: Guaranteeing Safety and Liveness** In order to never violate the safety property (each resource instance is accessed in mutual exclusion and no more than  $M$  instances are simultaneously allocated to processes), a process  $p_i$  first computes an upper bound on the number of instances of the resource which are currently allocated to the other processes. To that end, each process  $p_i$  manages a local array  $used\_by_i[1..n]$  such that  $used\_by_i[j]$  is an upper bound on the number of instances currently used by  $p_j$ . Hence, when a process  $p_i$  invokes `acquire_resource( $k_i$ )`, it first computes the values of  $used\_by_i[1..n]$ , and then waits until the predicate

$$\sum_{1 \leq j \leq n} used\_by_i[j] \leq M$$



**Fig. 11.2** Sending pattern of NOT\_USED() messages: Case 1

becomes satisfied. When this occurs,  $p_i$  is allowed to access  $k_i$  resource instances. As far as the liveness property is concerned, each request message carries a timestamp, and the total order on timestamps is used to establish a priority among the requests.

**Basic Message Pattern (1)** When a process  $p_i$  sends a timestamped request message to  $p_j$ , it conservatively considers that  $p_j$  uses the  $M$  resource instances. When it receives the request sent by  $p_i$ ,  $p_j$  sends by return a message NOT\_USED( $M$ ) if it is not interested, or if its current request has a greater timestamp.

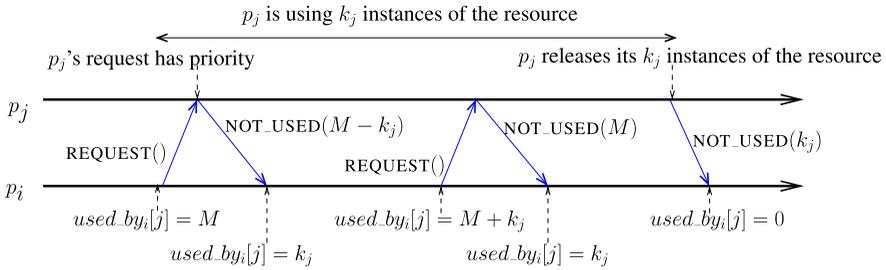
If its current request has priority over  $p_i$ 's request,  $p_j$  sends it by return a message NOT\_USED( $M - k_j$ ) (where  $k_j$  is the number of instances it asked for). This is depicted on Fig. 11.2. This message allows  $p_i$  to know how many instances of the resource are currently used or requested by  $p_j$ . Then, when it will later invoke `release_resource(k_j)`,  $p_j$  will send to  $p_i$  the message NOT\_USED( $k_j$ ) to indicate that it has finished using its  $k_j$  instances of the resource.

When considering the permission-based terminology, a message PERMISSION() sent by a process  $p_j$  to a process  $p_i$  in the mutex algorithm based on individual permissions is replaced here by two messages, namely a message NOT\_USED( $M - k_j$ ) and a message NOT\_USED( $k_j$ ). Said in another way, a message NOT\_USED( $x$ ) represents a fraction of a whole permission, namely,  $\frac{x}{M}\%$  of it.

**Basic Message Pattern (2)** It is possible that a process  $p_j$  uses  $k_j$  instances of a resource during a long period, and during this period  $p_i$  invokes several times first `acquire_resource(k_i)` (where  $k_i + k_j \leq M$ ), and later invokes `acquire_resource(k'_i)` (where  $k'_i + k_j \leq M$ ), etc. This scenario is depicted in Fig. 11.3.

When  $p_i$  issues its second invocation, it adds  $M$  to `used_by_i[j]`, which becomes then equal to  $M + k_j$ . In order that this value does not prevent  $p_i$  from progressing,  $p_j$  is required to send to  $p_i$  a message NOT\_USED( $M$ ) when it receives the second request of  $p_i$ . The corresponding message exchange pattern is described in Fig. 11.3.

**Behavior of a Process  $p_i$**  The corresponding  $k$ -out-of- $M$  algorithm executed by each process  $p_i$  is described in Fig. 11.4. The local variables `cs_state_i`, `clock_i`, `lrd_i`, and `perm_delayed_i` are the same as in the basic-mutex algorithm of Fig. 10.3. The meaning of the additional array variable `used_by_i[1..n]` has been explained above. The message pattern described in Fig. 11.3 is generated by the execution of line 15.



**Fig. 11.3** Sending pattern of `NOT_USED()` messages: Case 2

```

operation acquire_resource( $k_i$ ) is
(1)  $cs\_state_i \leftarrow trying$ ;
(2)  $\ell rd_i \leftarrow clock_i + 1$ ;
(3) for each  $j \in R_i$  do send REQUEST( $\ell rd_i, i$ ) to  $p_j$ ;
(4)  $used\_by_i[j] \leftarrow used\_by_i[j] + M$ 
(5) end for;
(6)  $used\_by_i[i] \leftarrow k_i$ ;
(7) wait ( $\sum_{1 \leq j \leq n} used\_by_i[j] \leq M$ );
(8)  $cs\_state_i \leftarrow in$ .

operation release_resource( $k_i$ ) is
(9)  $cs\_state_i \leftarrow out$ ;
(10) foreach  $j \in perm\_delayed_i$  do send NOT_USED( $k_i$ ) end for;
(11)  $perm\_delayed_i \leftarrow \emptyset$ .

when REQUEST( $k, j$ ) is received do
(12)  $clock_i \leftarrow \max(clock_i, k)$ ;
(13)  $prio_i \leftarrow ((cs\_state_i \neq out) \wedge (\ell rd_i, i) < (k, j))$ ;
(14) if ( $\neg prio_i \vee [prio_i \wedge (j \in perm\_delayed_i)]$ )
(15) then send NOT_USED( $M$ ) to  $p_j$ 
(16) else if ( $k_i \neq M$ ) then send NOT_USED( $M - k_i$ ) to  $p_j$  end if;
(17)  $perm\_delayed_i \leftarrow perm\_delayed_i \cup \{j\}$ 
(18) end if.

when NOT_USED( $x$ ) is received from  $p_j$  do
(19)  $used\_by_i[j] \leftarrow used\_by_i[j] - x$ .
% this line can render true the waiting predicate of line 7 %.

```

**Fig. 11.4** An algorithm for the  $k$ -out-of- $M$  mutex problem (code for  $p_i$ )

The particular case where a process asks for all instances of the resource is addressed at line 16.

Let us remember that, except for the **wait** statement of line 7, the code of each operation and the code associated with message receptions are locally executed in mutual exclusion.

The requests are served in their timestamp order. As an example, let us consider that  $M = 8$  and let us assume that the first five invocations (according to their times-

tamp) are for 3, 1, 2, 5, 1 instances of the resource, respectively. Then, the first three invocations are served concurrently. The fourth invocation will be served as soon as any three instances of the resource will have been released.

Let us finally observe that it is easy to modify the algorithm to allow a process  $p_i$  to release separately its  $k_i$  instances of the resource.

**Message Cost of the Algorithm** There are always  $(n - 1)$  request messages per use of a resource instance. The number of NOT\_USED() messages depends on the system state and varies between  $(n - 1)$  and  $2(n - 1)$ . Hence, the total number of messages per use of a set of instances of the resource is at least  $2(n - 1)$  and at most  $3(n - 1)$ .

### 11.1.4 Proof of the Algorithm

This section proves that the previous algorithm solves the  $k$ -out-of- $M$  problem. It assumes that (a) each invocation `acquire_resource( $k$ )` is such that  $1 \leq k \leq M$ , and (b) the periods during which a process uses the instances of the resource that it has obtained are finite.

**Lemma 6** *Let  $x^\tau$  be the number of instances of the resource accessed by the processes at time  $\tau$ . The algorithm described in Fig. 11.4 guarantees that  $\forall \tau: x^\tau \leq M$ .*

*Proof* The proof of the safety property is based on three claims.

*Claim C1*  $\forall i, j: used\_by_i[j] \geq 0$ .

*Proof of Claim C1* As all the local variables `used_by_i[j]` are initialized to 0, the claim is initially true.

A process  $p_i$  increases all its local counters `used_by_i[j]` at line 4 when it invokes `acquire_resource( $k_i$ )` (and this is the only place where  $p_i$  increases them). Moreover, it increases each of them by  $M$ . After these increases, the claim remains trivially true.

The reception of a message `REQUEST()` sent by  $p_i$  to another process  $p_j$  gives rise to either one message `NOT_USED( $M$ )` or at most one message `NOT_USED( $M - k_j$ )` followed by at most one message `NOT_USED( $k_j$ )`. It follows from line 19 that, when these messages are received by  $p_i$ , `used_by_i[j]` is decreased and can be set to 0. Hence, `used_by_i[j]` never becomes negative and the claim follows. (End of the proof of the claim.)

*Claim C2*  $\forall i: \forall \tau: (cs\_state_i^\tau = in) \Rightarrow (\sum_{1 \leq j \leq n} used\_by_i^\tau[j] \leq M)$ .

*Proof of Claim C2* It follows from line 7 that the claim is true when  $p_i$  sets `cs_state_i` to the value `in`. As  $p_i$  does not invoke `acquire_resource()` while `cs_state_i = in`, no local variable `used_by_i[j]` is increased when `cs_state_i = in`, and the claim follows. (End of the proof of the claim.)

*Claim C3* Let  $IN^\tau = \{i \mid cs\_state_i^\tau = in\}$ . Moreover, let  $nb\_alloc_j$  be such that  $nb\_alloc_j = k_j$  if  $j \in IN^\tau$ , and  $nb\_alloc_j = 0$ , otherwise ( $k_j$  is the number of instances requested by  $p_j$ ).

Considering  $IN^\tau \neq \emptyset$ , let  $p_m$  be the process such that  $m \in IN^\tau$ , and its request has the greatest timestamp among all the requests issued by the processes in  $IN^\tau$  (i.e.,  $\forall k \in IN^\tau \wedge k \neq m : \langle d_m, m \rangle > \langle d_k, k \rangle$ , where  $\langle d_x, x \rangle$  is the timestamp associated with the request of  $p_x$ ). We claim  $\forall j \neq m : used\_by_m^\tau[j] \geq nb\_alloc_j$ .

*Proof of Claim C3* If  $j \notin IN^\tau$ , we have (from Claim C1)  $used\_by_m^\tau[j] \geq 0 = nb\_alloc_j$ , and the claim follows. Hence, let us assume  $j \in IN^\tau$  and  $j \neq m$ . We have then  $nb\_alloc_j = k_j$ . Due to the definition of  $p_m$ , we have  $\langle d_m, m \rangle > \langle d_j, j \rangle$ . Consequently, when  $p_j$  receives REQUEST( $d_m, m$ ), we have  $prio_j = true$  and  $p_j$  sends to  $p_m$  the message NOT\_USED( $M - k_j$ ) (line 16). It follows that at time  $\tau$ ,  $p_j$  has not yet received NOT\_USED( $k_j$ ) and consequently  $used\_by_m^\tau[j] \geq k_j = nb\_alloc_j$ . (End of the proof of the claim.)

Let  $U^\tau$  be the number of instances of the resource which are used at time  $\tau$ . If  $IN^\tau = \emptyset$ , no instance is used and consequently  $U^\tau = 0 \leq M$ . So, let us assume  $IN^\tau \neq \emptyset$ . Let  $p_m$  be defined as in Claim C3. We have the following:

- $U^\tau = \sum_{j \in IN^\tau} k_j$  (definition of  $U^\tau$ ),
- $\sum_{j \in IN^\tau} k_j \leq k_m + \sum_{j \in IN^\tau \setminus \{m\}} used\_by_m^\tau[j]$  (from Claim C3),
- $k_m + \sum_{j \in IN^\tau \setminus \{m\}} used\_by_m^\tau[j] \leq \sum_{1 \leq j \leq n} used\_by_m^\tau[j]$  (from Claim C1),
- $\sum_{1 \leq j \leq n} used\_by_m^\tau[j] \leq M$  (from Claim C2).

It follows by transitivity that  $U^\tau \leq M$ , which concludes the proof of the safety property.  $\square$

**Lemma 7** *The algorithm described in Fig. 11.4 ensures that any invocation of acquire\_resource() eventually terminates.*

*Proof* Let us notice that the only statement at which a process may block is the **wait** statement at line 7. Let  $WAIT^\tau = \{i \mid (cs\_state_i = trying) \wedge (\sum_{1 \leq j \leq n} used\_by_i^\tau[j] > M)\}$ . The proof is based on the following claim.

*Claim C4* Considering  $WAIT^\tau \neq \emptyset$ , let  $p_m$  be the process such that  $m \in WAIT^\tau$ , and its request has the smallest timestamp among all the requests issued by the processes in  $WAIT^\tau$ .

Then, the quantity  $\sum_{1 \leq j \leq n} used\_by_m[j]$  decreases when  $\tau$  increases, and eventually either  $cs\_state_m = in$  or  $\sum_{1 \leq j \leq n} used\_by_m[j] = k_m + \sum_{z \in PRIO} k_z > M$ , where  $PRIO = \{z \mid (\langle d_z, z \rangle < \langle d_m, m \rangle) \wedge (cs\_state_z = in)\}$ .

*Proof Claim C4* Let  $j \neq m$ . There are two cases.

1. Case  $j \in WAIT^\tau$ . Due to the definition of  $m$  we have  $\langle d_j, j \rangle > \langle d_m, m \rangle$ . It follows that, when  $p_j$  receives the message REQUEST( $d_m, m$ ), it sends NOT\_USED( $M$ ) by return to  $p_m$  (lines 13–15). It follows that  $j \notin PRIO$  and we eventually have  $used\_by_m[j] = 0$ .

2. Case  $j \notin WAIT^\tau$ . If  $cs\_state_j \neq in$  or  $\langle d_j, j \rangle > \langle d_m, m \rangle$ , we are in the same case as previously,  $j \notin PRIO$  and eventually we have  $used\_by_m[j] = 0$ . If  $cs\_state_j = in$  and  $\langle d_j, j \rangle < \langle d_m, m \rangle$ , we have  $j \in PRIO$  and  $p_j$  sends `NOT_USED( $M - k_j$ )` by return to  $p_m$  (line 16). Hence, we eventually have  $used\_by_m[j] = k_j$ .

While  $p_m$  is such that  $cs\_state_m = trying$ , it does send new requests, and consequently no  $used\_by_m[x]$  can increase. It follows that, after a finite time, we have  $cs\_state_m = in$  or  $\sum_{1 \leq j \leq n} used\_by_m[j] = k_m + \sum_{z \in PRIO} k_z > M$ , which concludes the proof of the claim. (End of the proof of the claim.)

We now prove that, for any  $\tau$ , if a process belongs to  $WAIT^\tau$ , it will be such that  $cs\_state_i = in$ . Let us consider the process  $p_m$  of  $WAIT^\tau$  that has the smallest timestamp. It follows from Claim C4 that, after some finite time, we have

- Either  $cs\_state_m = in$ . In this case,  $p_m$  returns from its invocation of `acquire_resource()`.
- Or  $(cs\_state_m = trying) \wedge (k_m + \sum_{z \in PRIO} k_z > M)$ . In this case, as the resource instances are eventually released (assumption), the set  $PRIO$  decreases and the quantity  $\sum_{z \in PRIO} k_z$  decreases accordingly allowing the predicate  $cs\_state_m = in$  to eventually become true.

Hence,  $p_m$  eventually returns from its invocation of `acquire_resource()`.

The proof that any process  $p_x$  returns from `acquire_resource()` follows from the fact that the new invocations will eventually have timestamps greater than the one of  $p_x$ . The proof is the same as for the mutex algorithm based on individual permissions (see the proof of its liveness property in Lemma 4, Sect. 10.2.3).  $\square$

**Theorem 15** *The algorithm described in Fig. 11.4 solves the  $k$ -out-of- $M$  problem.*

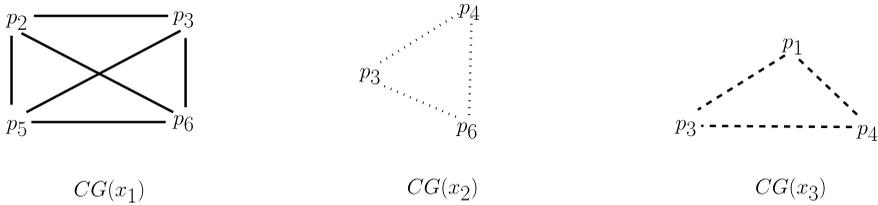
*Proof* The theorem follows from Lemmas 6 and 7.  $\square$

### 11.1.5 From Mutex Algorithms to $k$ -out-of- $M$ Algorithms

As noted when it was presented, the previous algorithm, which solves the  $k$ -out-of- $M$  mutex problem, is an adaptation of the mutex algorithm described in Sect. 10.2. More generally, it is possible to extend other mutex algorithms to solve the  $k$ -out-of- $M$  mutex problem. Problem 1 at the end of this chapter considers such an extension of the adaptive mutex algorithm described in Sect. 10.3.

## 11.2 Several Resources with a Single Instance

This section considers the case where there are  $X$  resource types, and there is one instance of each resource type  $x$ ,  $1 \leq x \leq X$ .



**Fig. 11.5** Examples of conflict graphs

The pattern formed by (a) the acquisition of resources by a process, (b) their use, and (c) finally their release, is called a (resource) *session*. During its execution, a process usually executes several sessions.

### 11.2.1 Several Resources with a Single Instance

**The Notion of a Conflict Graph** A graph, called *conflict graph*, is associated with each resource. Let  $CG(x)$  be the conflict graph associated with the resource type  $x$ . This graph is an undirected fully connected graph whose vertices are the processes allowed to access the resource  $x$ . An edge means a possible conflict between the two processes it connects: their accesses to the resource must be executed in mutual exclusion.

As an example, let us consider six processes  $p_1, \dots, p_6$ , and three resource types  $x_1, x_2$ , and  $x_3$ . The three corresponding conflict graphs are depicted in Fig. 11.5. The edges of  $CG(x_1)$  are depicted with plain segments, the edges of  $CG(x_2)$  with dotted segments, and edges of  $CG(x_3)$  with dashed segments. As we can see, the resource  $x_1$  can be accessed by the four processes:  $p_2, p_3, p_5$ , and  $p_6$ ;  $p_1$  accesses only  $x_3$ , while both  $p_2$  and  $p_4$  are interested in the resources  $x_1$  and  $x_3$ .

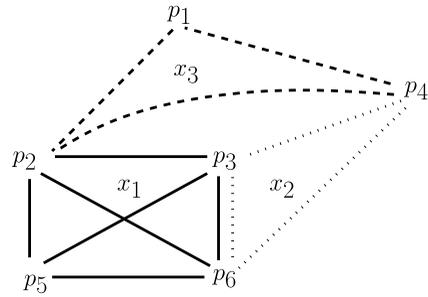
The union of these three conflict graphs defines a graph in which each edge is labeled by the resource from which it originates (the label is here the fact that the edge is plain/dotted/dashed). This graph, called *global conflict graph*, gives a global view on the conflicts on the whole set of resources.

The global graph associated with the previous three resources is depicted in Fig. 11.6. As we can see, this global graph allows  $p_1$  to access  $x_3$  while  $p_6$  is accessing the pair resources  $(x_1, x_2)$ . Differently,  $p_3$  and  $p_6$  conflict on both the resources  $x_1$  and  $x_2$ .

**Requests Are on All or a Subset of the Resources** As already indicated, two types of requests can be defined.

- In the first case, each resource session of a process  $p_i$  is *static* in the sense that it is always on all the resources that  $p_i$  is allowed to access. This set of resources is the set  $\{x \mid i \in CG(x)\}$ .

**Fig. 11.6**  
Global conflict graph



When considering the global conflict graph of Fig. 11.6, this means that each session of  $p_6$  is always on both  $x_1$  and  $x_2$ , while each resource session of  $p_4$  is always on  $x_2$  and  $x_3$ , etc.

- In the second case, each session of a process  $p_i$  is *dynamic* in the sense that it is on a dynamically defined subset of the resources that  $p_i$  is allowed to access.

In this case, some sessions of  $p_6$  can be only on  $x_1$ , others on  $x_2$ , and others on both  $x_1$  and  $x_2$ . According to the request pattern, dynamic allocation may allow for more concurrency than static allocation. As a simple example, if  $p_6$  wants to access  $x_1$  while  $p_3$  wants to access  $x_2$ , and no other process wants to access these resources, then  $p_6$  and  $p_3$  can access them concurrently.

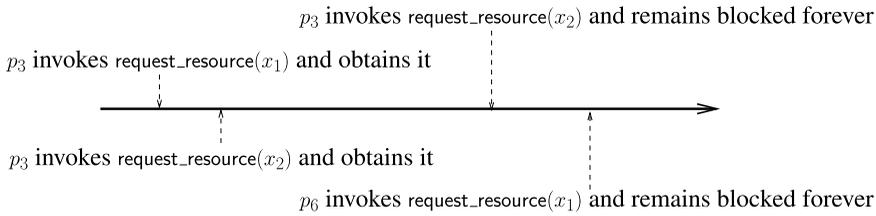
The first case is sometimes called in the literature the *dining philosophers problem*, while the second case is sometimes called the *drinking philosophers problem*.

### 11.2.2 Incremental Requests for Single Instance Resources: Using a Total Order

**Access Pattern** We consider here the case where a process  $p_i$  requires the resources it needs for its session one after the other, hence the name *incremental requests*. Moreover, a session can be static or dynamic.

**Possibility of Deadlock** The main issue that has to be solved is the prevention of deadlocks. Let us consider the processes  $p_3$  and  $p_6$  in Fig. 11.6. When both processes want to acquire both the resources  $x_1$  and  $x_2$ , the following can happen with incremental requests.

- $p_3$  invokes `acquire_resource( $x_1$ )` and obtains the resource  $x_1$ .
- $p_6$  invokes `acquire_resource( $x_2$ )` and obtains the resource  $x_2$ .
- Then  $p_3$  and  $p_6$  invoke `acquire_resource( $x_2$ )` and `acquire_resource( $x_1$ )`, respectively. As the resource asked by a process is currently owned by the other process, each of  $p_3$  and  $p_6$  starts waiting, and the waiting period of each of them will terminate when the resource it is waiting on is released. But, as each process releases resources only after it has obtained all the resources it needs for the current session, this will never occur (Fig. 11.7).



**Fig. 11.7** A deadlock scenario involving two processes and two resources

This is a classical deadlock scenario. (Deadlocks involving more than two processes accessing several resources can easily be constructed.)

**Deadlock Prevention: A Total Order on Resources** Let  $\{x_1, x_2, \dots, x_X\}$  be the whole set of resources accessed by the processes, each process accessing possibly only a subset of them (as an example, see Fig. 11.6). Let  $<$  be a total order on this set of resources, e.,g.,  $x_1 < \dots < x_m$ . The processes are required to obey the following rule:

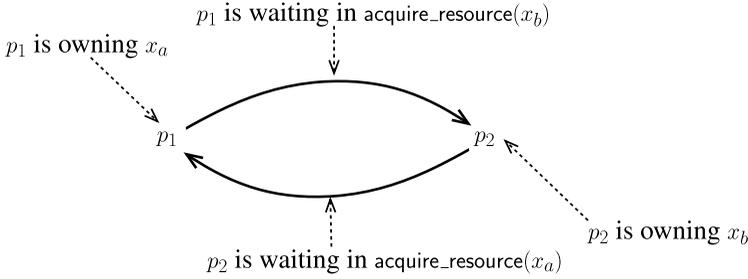
- During a session, a process may invoke `acquire_resource( $x_k$ )` only if it has already obtained all the resources  $x_j$  it needs (during this session) which are such that  $x_j < x_k$ .

**Theorem 16** *If, during each of its sessions, every process asks for the resources (it needs for that session) according to the total order  $<$ , no deadlock can occur.*

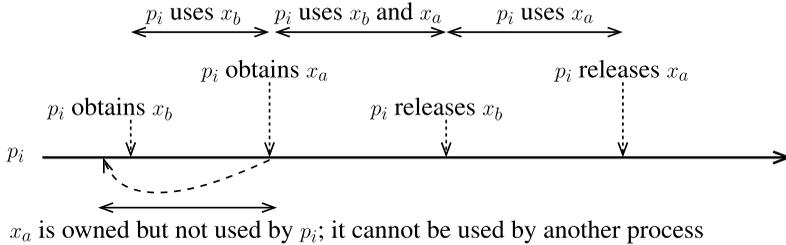
*Proof* Let us first introduce the notion of a *wait-for* graph (WFG). Such a graph is a directed graph defined as follows (see also Sect. 15.1.1). Its vertices are the processes and its edges evolve dynamically. There is an edge from  $p_i$  to  $p_j$  if (a)  $p_j$  is currently owning a resource  $x$  (i.e.,  $p_j$  has obtained and not yet released the resource  $x$ ) and (b)  $p_i$  wants to acquire it (i.e.,  $p_i$  has invoked `acquire_resource( $x$ )`). Let us observe that, when there is a (directed) cycle, this cycle never disappears (for it to disappear, a process in the cycle should release a resource, but this process is blocked waiting for another resource). A cycle in the WFG means that there is deadlock: each process  $p_k$  in the cycle is waiting for a resource owned by a process  $p_{k'}$  that belongs to the cycle, which in turn is waiting for a resource owned by a process  $p_{k''}$  that belongs to the cycle, etc., without ever exiting from the cycle. The progress of each process  $p_k$  on the directed cycle depends on a statement (releasing a resource) that it cannot execute.

The proof is by contradiction. Let us suppose that there is a directed cycle in the WFG. To simplify the proof, and without loss of generality, let us assume that the cycle involves only two processes  $p_1$  and  $p_2$ :  $p_1$  wants to acquire the resource  $x_a$  that is currently owned by  $p_2$ , and  $p_2$  wants to acquire the resource  $x_b$  that is currently owned by  $p_1$  (Fig. 11.8).

The following information can be extracted from the graph:



**Fig. 11.8** No deadlock with ordered resources



**Fig. 11.9** A particular pattern in using resources

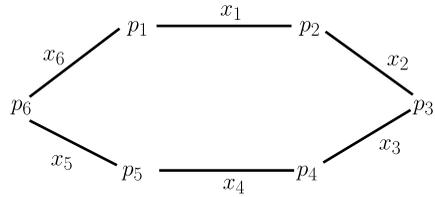
- As  $p_1$  is owning the resource  $x_a$  and waiting for the resource  $x_b$ , it invoked first `acquire_resource( $x_a$ )` and then `acquire_resource( $x_b$ )`.
- As  $p_2$  is owning the resource  $x_b$  and waiting for the resource  $x_a$ , it invoked first `acquire_resource( $x_b$ )` and then `acquire_resource( $x_a$ )`.

As there is an imposed total order to acquire resources, it follows that either  $p_1$  or  $p_2$  does not follow the rule. Taking the contrapositive, we have that “each process follows the rule”  $\Rightarrow$  “there is no cycle in WFG”. Hence, no deadlock can occur if every process follows the rule (during all its sessions of resource use).  $\square$

**Drawback of the Previous Approach** Accessing resources according to a predefined total order agreed upon by all processes prevents deadlock from occurring. But deadlock prevention always has a price. Let us consider a process that, during a session, wants to use first a resource  $x_b$  alone, then both the resources  $x_a$  and  $x_b$ , and finally the resource  $x_a$  alone. The corresponding time pattern is described (above the process axis) in Fig. 11.9.

- If  $x_b < x_a$ , the order in which  $p_i$  needs the resources is the same as the imposed total order. In this case, the allocation is optimal in the sense that, during the session,  $x_b$  and  $x_a$  are obtained just when  $p_i$  needs them. This is the case depicted above the process axis in Fig. 11.9.
- If  $x_a < x_b$ , to obey the rule,  $p_i$  is forced to require  $x_a$  before  $x_b$ . This case is depicted below the process axis in Fig. 11.9. The backward arrow means that  $p_i$  has

**Fig. 11.10** Conflict graph for six processes, each resource being shared by two processes



to invoke `acquire_resource( $x_a$ )` before invoking `acquire_resource( $x_b$ )`. The total order on  $x_a$  and  $x_b$  does not fit the order in which  $p_i$  needs them. Hence, during some duration,  $x_a$  is owned but not used by  $p_i$ . If, during that period of time, another process wants to use the resource  $x_a$ , it cannot. This is the inescapable price that has to be paid when a total order on resources is used to prevent deadlocks from occurring.

### 11.2.3 Incremental Requests for Single Instance Resources: Reducing Process Waiting Chains

**A Worst-Case Scenario** Let us consider six processes  $p_1, \dots, p_6$ , and six resources  $x_1, \dots, x_6$ , such that, in each session, each process needs two resources, namely

$p_i$  requires  $x_{i-1}$  and  $x_i$ , for  $1 < i \leq 6$ , and  
 $p_1$  requires  $x_1$  and  $x_6$ .

The corresponding global conflict graph is described in Fig. 11.10. The label associated with an edge indicates the resource on which its two processes (vertices) conflict. (This example is the conflict graph of the resource allocation problem known as the *dining philosophers problem*. Each process is a philosopher who needs both the fork at his left and the fork at its right—the forks are the resources—to eat spaghetti.)

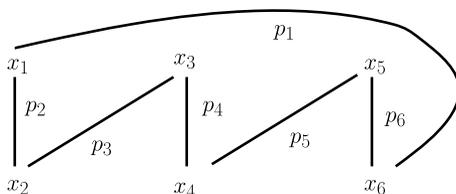
Let us consider an execution in which

First  $p_6$  obtains  $x_5$  and  $x_6$ ,  
 Then  $p_5$  obtains  $x_4$  and waits for  $x_5$  (blocked by  $p_6$ ),  
 Then  $p_4$  obtains  $x_3$  and waits for  $x_4$  (blocked by  $p_5$ ),  
 Then  $p_3$  obtains  $x_2$  and waits for  $x_3$  (blocked by  $p_4$ ),  
 Then  $p_2$  obtains  $x_1$  and waits for  $x_2$  (blocked by  $p_3$ ),  
 Then  $p_1$  waits for  $x_2$  (blocked by  $p_2$ ).

There is no deadlock (process  $p_6$  is active, using  $x_5$  and  $x_6$ ). However, all the processes, except  $p_6$ , are passive (waiting for a resource instance). Moreover,  $p_1$ ,  $p_2$ ,  $p_3$ , and  $p_4$  are waiting for a resource currently owned by a passive process. The corresponding WFG is as follows (let us recall that a directed edge means “blocked by”)

$$p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow p_5 \rightarrow p_6.$$

**Fig. 11.11** Optimal vertex-coloring of a resource graph



Hence, the length of the waiting chain (longest path in the WFG) is  $n - 1 = 5$ , which is the worst case: only one process is active.

**Reduce the Length of Waiting Chains: Coloring a Resource Graph** An approach to reduce the length of waiting chains consists in imposing on each process separately a total order on the resources it wants to acquire. Moreover, the resulting partial order obtained when considering all the resources has to contain as few edges as possible. This approach is due to N.A. Lynch (1981).

To that end, let us define a *resource graph* as follows. Its vertices are the resources and there is an edge connecting two resources if these two resources can be requested by the same process during one of its sessions. The resource graph associated with the previous example is described in Fig. 11.11 (the label on an edge denotes the processes that request the corresponding resources). The meaning of a (non-directed) edge is that the corresponding resources are requested in some order by some process(es). As there is no process accessing both  $x_1$  and  $x_4$ , there is no edge connecting these vertices.

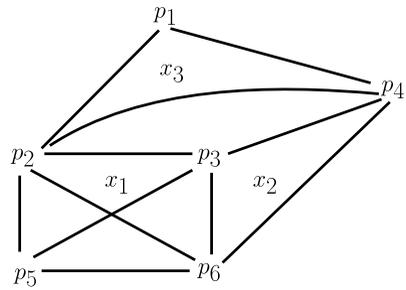
Hence, the aim is to find a partial order on the resources such that (a) the resources requested by each process (during a session) are totally ordered and (b) there are as few edges as possible. Such a partial order can be obtained as follows.

- First, find an optimal vertex-coloring of the resource graph (no two resources connected by an edge have the same color and the number of color is minimal).
- Then, define a total order on the colors.
- Finally, each process acquires the resources it needs during a session according to the order on their colors. (Let us observe that, due to its construction, the resources accessed by a process are totally ordered.)

**Example** When considering Fig. 11.11, the minimal number of colors is trivially two. Let us color the non-neighbor resources  $x_1, x_3,$  and  $x_5$  with color  $a$  and the non-neighbor resources  $x_2, x_4,$  and  $x_6$  with color  $b$ . Moreover, let  $a < b$  be the total order imposed on these colors. Hence, the edges of the corresponding resource graph are directed from top to bottom, which means that  $p_4$  has to acquire first  $x_3$  and then  $x_4$ , while  $p_5$  has to acquire first  $x_5$  and then  $x_4$ .

The worst case scenario described in Sect. 11.2.2 cannot occur. When  $p_6$  has obtained  $x_5$  and  $x_6$ ,  $p_5$  is blocked waiting for  $x_5$  (which it has to require before  $x_4$ ). Hence,  $p_4$  can obtain  $x_3$  and  $x_4$ , and  $p_2$  can obtain  $x_1$  and  $x_2$ . More generally, the maximal length of a waiting chain is the number of colors minus 1.

**Fig. 11.12** Conflict graph for static sessions ( $SS\_CG$ )



**Remark** Let us remember that the optimal coloring of the vertices of a graph is an NP-complete problem (of course, a more efficient near-optimal coloring may be used). Let us observe that the non-optimal coloring in which all the resources are colored with different colors corresponds to the “total order” strategy presented in Sect. 11.2.2.

### 11.2.4 Simultaneous Requests for Single Instance Resources and Static Sessions

Differently from the *incremental request* approach, where a process requires, one after the other, the resources it needs during a session, the *simultaneous request* approach directs each process to require simultaneously (i.e., with a single operation invocation) all the resources it needs during a session. To that end, the processes are provided with an operation

`acquire_resource(res_seti),`

whose input parameter  $res\_set_i$  is the set of resources that the invoking process  $p_i$  needs for its session.

As we consider static sessions, every session of a process  $p_i$  involve all the resources that this process is allowed to access (i.e.,  $res\_set_i = \{x \mid i \in CG(x)\}$ ).

**Conflict Graph for Static Sessions** As, during a session, the requests of a process are on all the resources it is allowed to access, the global conflict graph (see Fig. 11.6) can be simplified. Namely, when two processes conflict on several resources, e.g.,  $x_a$  and  $x_b$ , these resources can be considered as a single virtual resource  $x_{a,b}$ . This is because, the sessions being static, each session of these processes involves both  $x_a$  and  $x_b$ .

The corresponding conflict graph for static session is denoted  $SS\_CG$ . Its vertices are the vertices in the conflict graphs  $\bigcup_{1 \leq x \leq M} CG(x)$ . There is an edge  $(p_y, p_z)$  in  $SS\_CG$  if there is a conflict graph  $CG(x)$  containing such an edge. The graph  $SS\_CG$  associated with the global conflict graph of Fig. 11.6 is described in Fig. 11.12. Let

us observe that, when  $p_6$  accesses  $x_1$  and  $x_2$ , nothing prevents  $p_1$  from accessing  $x_3$ .

Due to the fact that, in each of its sessions, a process  $p_i$  requires all the resources  $x$  such that  $i \in CG(x)$ , it follows that each process is in mutual exclusion with all its neighbors in the static session conflict graph. Hence, a simple solution consists in adapting a mutex algorithm so that mutual exclusion is required only between processes which are neighbors in the static conflict graph.

**Mutual Exclusion with Neighbor Processes in the Conflict Graph** Let us consider the mutex algorithms based on individual permissions described in Chap. 10. These algorithms are modified as follows:

- Each site manages an additional local variable  $neighbors_i$  which contains the identities of its neighbors in the conflict graph. (Let us notice that if  $j \in neighbors_i$ , then  $i \in neighbors_j$ .)
- When considering the non-adaptive algorithm of Fig. 10.3, the set  $R_i$  (which was a constant equal to  $\{1, \dots, n\} \setminus \{i\}$ ) remains a constant which is now equal to  $neighbors_i$ .
- When considering the adaptive algorithm of Fig. 10.7, let  $p_i$  and  $p_j$  be two processes which are neighbors in the conflict graph. The associated message PERMISSION( $\{i, j\}$ ) is initially placed at one these processes, e.g.,  $p_i$ , and the initial values of  $R_i$  and  $R_j$  are then such that  $j \notin R_i$  and  $i \in R_j$ .
- When considering the bounded adaptive algorithm of Fig. 10.10, the initialization is as follows. For any two neighbors  $p_i$  and  $p_j$  in the conflict graph such that  $i > j$ , the message PERMISSION( $\{i, j\}$ ) is initially at  $p_i$  (hence,  $i \in R_j$  and  $j \notin R_i$ ), and its initial state is *used* (i.e.,  $perm\_state_i[j] = used$ ).

In the last two cases,  $R_i$  evolves according to requests, but we always have  $R_i \subseteq neighbors_i$ . Moreover, when the priority on requests (liveness) is determined from timestamps (the first two cases), as any two processes  $p_i$  and  $p_j$  which are not neighbors in the conflict graph never compete for resources, it follows that  $p_i$  and  $p_j$  can have the same identity if  $neighbors_i \cap neighbors_j = \emptyset$  (this is because their common neighbors, if any, must be able to distinguish them). Hence, any two processes at a distance greater than 2 in the conflict graph can share the same identity. This can help reduce the length of waiting chains.

### 11.2.5 Simultaneous Requests for Single Instance Resources and Dynamic Sessions

**The Deadlock Issue** In this case, each session of a process  $p_i$  involves a dynamically defined subset of the resources for which  $p_i$  is competing with other processes, i.e., a subset of  $\{x \mid i \in CG(x)\}$ .

Let us consider the global conflict graph depicted in Fig. 11.6. If, during a session, the process  $p_6$  wants to access the resource  $x_1$  only, it needs a mutual exclusion

algorithm in order to prevent  $p_2$ ,  $p_3$ , or  $p_5$  from simultaneously accessing this resource. To that end, a mutex algorithm per resource is used. Let  $M(x)$  denote the mutex algorithm associated with  $x$ .

As each of these algorithms involves only the processes accessing the corresponding resource, this allows processes whose current sessions require different sets of resources to access them concurrently. As an example, if the current session of  $p_6$  requires only  $x_1$  while, simultaneously, the current session of  $p_3$  requires only  $x_2$ , being managed by distinct mutex algorithms, these resources can be accessed concurrently by  $p_6$  and  $p_3$ .

Unfortunately, as the mutex algorithms  $M(x)$  are independent one from the others, there is a risk of deadlock when both the current sessions of  $p_6$  and  $p_3$  require both the resources  $x_1$  and  $x_2$ . This is because the mutex algorithm  $M(x_1)$  can allocate  $x_1$  to  $p_3$ , while the algorithm  $M(x_2)$  allocates  $x_2$  to  $p_6$ .

**Establish a Priority to Prevent Deadlock** A way to prevent deadlock from occurring consists in the stacking of an additional mutex algorithm on top of the  $M(x)$  algorithms. More precisely, let us consider an additional mutex algorithm, denoted  $GM$ , which ensures mutual exclusion on neighbor processes in the conflict graph for static session  $SS\_CG$  introduced in the previous section (Fig. 11.12).

This algorithm is used to solve conflicts between neighbor processes in  $SS\_CG$  when they want to use several resources. As an example, when both  $p_6$  and  $p_3$  want to use the resources  $x_1$  and  $x_2$ , their conflict is solved by  $GM$ : If  $p_6$  has priority over  $p_3$  in  $GM$ , it has priority to access both resources. This has a side effect, namely, if  $M(x_2)$  has already granted the resource  $x_2$  to  $p_3$ , this process has to release it in order  $p_6$  can obtain it. (If  $p_3$  has obtained both resources, whatever the priority given by  $GM$ , it keeps them until the end of its session).

**Cooperation Between  $GM$  and the Mutex Algorithms Associated with Each Resource** Mutex algorithms between neighbor processes in a graph were introduced in the last part of Sect. 11.2.4. Let us consider that the mutex algorithm  $GM$  and all the mutex algorithms  $M(x)$  are implemented by the one described in the last item of Sect. 11.2.4, in which all variables are bounded. (As we have seen, this algorithm was obtained from a simple modification of the adaptive mutex algorithm of Fig. 10.10.)

Let us recall that such a mutex algorithm is fair: Any process  $p_i$  that invokes `acquire_mutex()` eventually enters the critical section state, and none of its neighbors  $p_j$  will enter it simultaneously. Moreover, two processes  $p_i$  and  $p_j$  which are not neighbors can be simultaneously in critical section.

Let  $cs\_state_i$  be the local variable of  $p_i$  that describes its current mutex exclusion state with respect to the general algorithm  $GM$ . As we have seen, its value belongs to  $\{out, trying, in\}$ . Similarly, let  $cs\_state_i[x]$  be the local variable of  $p_i$  that describes its current state with respect to the mutex algorithm  $M(x)$ .

Let us consider the algorithm  $GM$ . Given a process  $p_i$ , we have the following with regard the transitions of its local variable  $cs\_state_i$ . Let us recall that the transition of  $cs\_state_i$  from *trying* to *in* is managed by the mutex algorithm itself. Differently, the transitions from *out* to *trying*, and from *in* to *out*, are managed by the

invoking process  $p_i$ . Hence, as far as  $GM$  is concerned, rules that force a process to proceed from *out* to *trying* and from *in* to *out* have to be defined. These rules are as follows:

- R1. If  $(cs\_state_i = out) \wedge (\exists x : cs\_state_i[x] = trying)$ ,  $p_i$  must invoke the operation `acquire_resource()` of  $GM$  to acquire the mutual exclusion with respect to its neighbors in  $SS\_CG$ , so that eventually  $cs\_state_i = trying$ .
- R2. If  $(cs\_state_i = in) \wedge (\forall x : i \in CG(x) : cs\_state_i[x] \neq trying)$ ,  $p_i$  must invoke the operation `release_resource()` of  $GM$  so that eventually  $cs\_state_i = out$ .

Finally, when a process  $p_i$  receives a request for a resource  $x$  from its neighbor  $p_j$ , the priority rule followed by  $p_i$  is the following:

- R3. If (a)  $p_i$  is not interested in the resource  $x$  ( $cs\_state_i[x] = out$ ), or (b)  $p_i$  is neither using the resource  $x$  nor having priority with respect to  $p_j$  in  $GM$ , then  $p_i$  allows  $p_j$  to use the resource  $x$  (i.e.,  $p_i$  sends its permission to  $p_j$ ).

**Sketch of a Bounded Resource Allocation Algorithm for Dynamic Sessions** At each process  $p_i$  and for each resource  $x$ ,  $cs\_state_i[x]$  is initialized to *out*. Moreover, each process  $p_i$  manages the following additional local variables.

- $need_i[x]$  is a Boolean which is true when  $p_i$  is interested in the resource  $x$ .
- $req_i[x, j]$  is a Boolean which is true when  $p_i$  is allowed to request (to  $p_j$ ) the permission to access the resource  $x$ . This permission is represented by the message `X_PERMISSION( $x(i, j)$ )`.
- $here_i[x, j]$  is a Boolean which is true when  $p_i$  has obtained (from  $p_j$ ) the permission to access the resource  $x$ .

Initially, for any pair of neighbors  $p_i$  and  $p_j$  the message `X_PERMISSION( $x(i, j)$ )` is placed at one process, e.g.,  $p_i$ . We have then  $here_i[x, j] = true$ ,  $req_i[x, j] = false$ ,  $here_j[x, i] = false$ , and  $req_j[x, i] = true$ .

As already indicated, the transitions of the mutex algorithm  $GM$  are governed by the rules  $R1$  and  $R2$ . A sketch of a description of the resource allocation algorithm is presented in Fig. 11.13. It considers a single pair of neighbor processes  $p_i$  and  $p_j$  and a single of the possibly many resources  $x$  that they share. When it receives from  $p_j$  a request for the resource  $x$ ,  $p_i$  computes its priority (line 5); as explained above, this computation involves the state of  $p_i$  with respect to  $GM$ . (This is expressed by the presence of the message `PERMISSION( $\{i, j\}$ )` at  $p_i$ . Let us recall that this message is a message of the algorithm  $GM$ .) The writing of the whole detailed algorithm is the topic of Problem 5 at the end of this chapter.

## 11.3 Several Resources with Multiple Instances

**The Generalized  $k$ -out-of- $M$  Problem** This section considers the case where there are  $X$  resource types, and for each  $x$ ,  $1 \leq x \leq X$ , there are  $M[x]$  instances of the resource type  $x$ . A request of a process  $p_i$  may concern several instances of

```

to acquire  $x$  with respect to  $p_j$  do
(1) if  $((cs\_state_i[x] = trying) \wedge need_i[x] \wedge req_i[x, j] \wedge (\neg here_i[x, j]))$ 
(2)   then send REQUEST( $x$ ) to  $p_j$ ;  $req_i[x, j] \leftarrow false$ 
(3) end if.

to release  $x$  with respect to  $p_j$  do
(4) if  $(req_i[x, j] \wedge here_i[x, j])$ 
(5)   then if  $(\neg need_i[x, j]) \vee \neg[(cs\_state_i[x] = in) \wedge (PERMISSION(\{i, j\}) \text{ is at } p_i)]$ 
(6)     then send X_PERMISSION( $x$ ) to  $p_j$ ;  $here_i[x, j] \leftarrow false$ 
(7)     end if
(8) end if.

when REQUEST( $x$ ) is received from  $p_j$  do
(9)    $req_i[x, j] \leftarrow true.$ 

when X_PERMISSION( $x$ ) is received from  $p_j$  do
(10)   $here_i[x, j] \leftarrow true.$ 

```

**Fig. 11.13** Simultaneous requests in dynamic sessions (sketch of code for  $p_i$ )

each resource. As an example, for a given session,  $p_i$  may request  $k_i[x]$  instances of resource type  $x$  and  $k_i[y]$  instances of the resource type  $y$ .

This section presents solutions for dynamic sessions (as we have seen this means that, in each session, a process defines the specific subset of resources it needs from the set of resources it is allowed to access). Let us observe that, as these solutions work for dynamic sessions, they trivially work for static sessions.

**The Case of Dynamic Sessions with Incremental Requests** In this case, the same techniques as the ones described in Sect. 11.2 (which was on resources with a single instance) can be used to prevent deadlocks from occurring (total order on the whole set of resources, or partial order defined from a vertex-coloring of the resource graph).

As far as algorithms are concerned, a  $k$ -out-of- $M$  mutex algorithm is associated with each resource type  $x$ . A process invokes then `acquire_resource( $x, k_i$ )`, where  $x$  is the resource type and  $k_i$  the number of its instances requested by  $p_i$ , with  $1 \leq k_i \leq M[x]$ .

**The Case of Dynamic Sessions with Simultaneous Requests** Let  $RX_i \subset \{x \mid i \in CG(x)\}$  denote the dynamically defined set of resource types that the invoking process  $p_i$  wants to simultaneously acquire during a session, and, for each  $x \in RX_i$ , let  $k_i^x$  denote the number of  $x$ 's instances that it needs.

A generalization of the  $k$ -out-of- $M$  algorithm presented in Fig. 11.4, which implements the operations `acquire_resource( $\{(x, k_i^x)\}_{x \in RX_i}$ )` and `release_resource( $\{(x, k_i^x)\}_{x \in RX_i}$ )`, is presented in Fig. 11.14. It assumes a partial instance of the basic-algorithm of Fig. 11.4 for each resource type  $x \in RX_i$ . Let  $cs\_tate_i^x$ ,  $used\_by_i^x$ , and  $perm\_delayed_i^x$  be the local variables of  $p_i$  associated with the resource type  $x$ ;  $perm\_delayed_i^x$  contains only identities of the processes in  $CG(x)$ ,

```

operation acquire_resource( $\{(x, k_i^x)\}_{x \in RX_i}\}$  is
(1) for each  $x \in RX_i$  do  $cs\_state_i^x \leftarrow$  trying end for;
(2)  $\ell rd_i \leftarrow$   $clock_i + 1$ ;
(3) for each  $x \in RX_i$  do
(4)   for each  $j \in CF(x)$  do send REQUEST( $x, k_i^x, (\ell rd_i, i)$ ) to  $p_j$ ;
(5)    $used\_by_i^x[j] \leftarrow used\_by_i^x[j] + M[x]$ 
(6)   end for;
(7)    $used\_by_i^x[i] \leftarrow k_i^x$ ;
(8) end for;
(9) wait ( $\bigcap_{x \in RX_i} (\sum_{1 \leq j \leq n} used\_by_i^x[j] \leq M[x])$ );
(10) for each  $x \in RX_i$  do  $cs\_state_i^x \leftarrow$  in end for.

operation release_resource( $\{(x, k_i^x)\}_{x \in RX_i}\}$  is
(11) for each  $x \in RX_i$  do
(12)    $cs\_state_i^x \leftarrow$  out;
(13)   for each  $j \in perm\_delayed_i^x$  do send NOT_USED( $x, k_i^x$ ) to  $p_j$  end for;
(14)    $perm\_delayed_i^x \leftarrow \emptyset$ 
(15) end for.

```

**Fig. 11.14** Algorithms for generalized  $k$ -out-of- $M$  (code for  $p_i$ )

and  $used\_by_i^x$  is an array with one entry per process in  $CG(x)$ . Moreover, each message is tagged with the corresponding resource type. Figure 11.14 is a simple extension of the basic- $k$ -out-of- $M$ .

When  $p_i$  invokes  $acquire\_resource(\{(x, k_i^x)\}_{x \in RX_i})$ , it first proceeds to the state *trying* for each resource  $x$  (line 1). Then, it computes a date for its request (line 2). Let us observe that this date is independent of the set  $RX_i$ . Process  $p_i$  then sends a timestamped request to all the processes with which it competes for the resources in  $RX_i$ , and computes an upper bound of the number of instances of the resources in which it is interested (lines 3–8). When there are enough available instances of the resources it needs,  $p_i$  is allowed to use them (line 9). It then proceeds to the state *in* with respect to each of these resources (line 10).

When  $p_i$  invokes  $release\_resource(\{(x, k_i^x)\}_{x \in RX_i})$ , it executes the same code as in Fig. 11.4 for each resource of  $RX_i$  (lines 11–15).

The “server” role of a process (management of the message reception) is the same as in Fig. 11.4. The messages for a resource type  $x$  are processed by the corresponding instance of the basic-algorithm. The important point is that all these instances share the same logical clock  $clock_i$ . The key of the solution lies in the fact that a single timestamp is associated with all the request messages sent during an invocation, and all conflicting invocations on one or several resources are totally ordered by their timestamps.

## 11.4 Summary

This chapter was devoted to the generalized  $k$ -out-of- $M$  problem. This problem captures and abstracts resource allocation problems where (a) there are one or several

types of resource, (b) each resource has one or several instances, and (c) each process may request several instances of each resource type. Several algorithms solving this problem have been presented. Due to the multiplicity of resources, one of the main issues that these algorithms have to solve is deadlock prevention. Approaches that address this issue have been discussed.

The chapter has also introduced the notion of a conflict graph, which is an important conceptual tool used to capture conflicts among processes. It has also shown how the length of process waiting chains can be reduced. Both incremental versus simultaneous requests on the one side, and static versus dynamic sessions of resource allocation on the other side, have been addressed in detail.

## 11.5 Bibliographic Notes

- Resource allocation and the associated deadlock prevention problem originated in the design and the implementation of the very first operating systems (e.g., [110, 165]).
- The dining philosophers problem was introduced by E.W. Dijkstra in [111]. It abstracts the case where, in each of its sessions, each process requires always the same set of resources. The drinking philosophers problem was introduced by K.M. Chandy and J. Misra in [78]. It generalizes the dining philosophers problem in the sense that the set of resources required by a process is defined dynamically in each session.
- The deadlock prevention technique based on a total ordering of resources is due to J.W. Havender [166]. The technique based on vertex coloring of a resource graph is due to N.A. Lynch [241].
- The algorithm solving the multiple entries mutex problem (1-out-of- $M$ ) presented in Fig. 11.1 is due to K. Raymond [305].

The general  $k$ -out-of- $M$  resource allocation algorithm presented in Fig. 11.4 is due to M. Raynal [311].

The basic mutex algorithm from which these two algorithms have been derived is due to G. Ricart and A.K. Agrawala [327] (this algorithm was presented in Chap. 10).

- The bounded resource allocation algorithm for single instance resources in dynamic sessions presented in Sect. 11.2.5 is due to K.M. Chandy and J. Misra [78]. This algorithm is known under the name *drinking philosophers* algorithm (the resources are bottles shared by neighbor processes). Presentation of this algorithm can also be found in [149, 242, 387].
- Quorum (arbiter)-based algorithms that solve the  $k$ -out-of- $M$  are presented in [245, 280].

```

operation acquire_resource( $k_i$ ) is
(1)  $cs\_state_i \leftarrow trying$ ;
(2)  $\ell rd_i \leftarrow clock_i + 1$ ;
(3) for each  $j \in R_i$  do
(4)   if ( $used\_by_i[j] = 0$ ) then send REQUEST( $\ell rd_i, i$ ) to  $p_j$ ;
(5)      $sent\_to_i[j] \leftarrow true$ ;  $used\_by_i[j] \leftarrow M$ 
(6)     else  $sent\_to_i[j] \leftarrow false$ 
(7)   end if
(8) end for;
(9)  $used\_by_i[i] \leftarrow k_i$ ;
(10) wait ( $\sum_{1 \leq j \leq n} used\_by_i[j] \leq M$ );
(11)  $cs\_state_i \leftarrow in$ .

operation release_resource( $k_i$ ) is
(12)  $cs\_state_i \leftarrow out$ ;
(13) foreach  $j \in perm\_delayed_i$  do send NOT_USED( $k_i$ ) end for;
(14)  $perm\_delayed_i \leftarrow \emptyset$ .

when REQUEST( $k, j$ ) is received do
(15)  $clock_i \leftarrow \max(clock_i, k)$ ;
(16)  $prio_i \leftarrow ((cs\_state_i = in) \vee [(cs\_state_i = trying) \wedge ((\ell rd_i, i) < (k, j))])$ ;
(17) if ( $\neg prio_i$ ) then send NOT_USED( $M$ ) to  $p_j$ 
(18)   else if ( $k_i \neq M$ ) then send NOT_USED( $M - k_i$ ) to  $p_j$  end if;
(19)    $perm\_delayed_i \leftarrow perm\_delayed_i \cup \{j\}$ 
(20) end if.

when NOT_USED( $x$ ) is received from  $p_j$  do
(21)  $used\_by_i[j] \leftarrow used\_by_i[j] - x$ ;
(22) if ( $(cs\_state_i = trying) \wedge (used\_by_i[j] = 0) \wedge (\neg sent\_to_i[j])$ )
(23)   then send REQUEST( $\ell rd_i, i$ ) to  $p_j$ 
(24)      $sent\_to_i[j] \leftarrow true$ ;  $used\_by_i[j] \leftarrow M$ 
(25) end if.

```

Fig. 11.15 Another algorithm for the  $k$ -out-of- $M$  mutex problem (code for  $p_i$ )

## 11.6 Exercises and Problems

1. The  $k$ -out-of- $M$  algorithm described in Fig. 11.4 requires between  $2(n - 1)$  and  $3(n - 1)$  messages per use of a set of instances of the resource. The following algorithm is proposed to reduce this number of messages. When  $used\_by_i[j] \neq 0$ , the process  $p_i$  knows an upper bound on the number of instances of the resource used by  $p_j$ . In that case, it is not needed for  $p_i$  to send a request message to  $p_j$ . Consequently,  $p_i$  sends a request to  $p_j$  only when  $used\_by_i[j] = 0$ ; when this occurs,  $p_i$  records it by setting a flag  $sent\_to_i[j]$  to the value *true*.

The corresponding algorithm is described in Fig. 11.15 (where  $R_i = \{1, \dots, n\} \setminus \{i\}$ ). Let us recall that the quantity  $\sum_{1 \leq j \leq n} used\_by_i[j]$ , which appears in the **wait** statement (line 10) is always computed in local mutual exclusion with the code associated with the reception of a message NOT\_USED().

- Show that the algorithm is correct when the channels are FIFO.

- Show that the algorithm is no longer correct when the channels are not FIFO. To that end construct a counterexample.
- Is the statement  $sent\_to_i[j] \leftarrow true$  at line 24 necessary? Justify your answer.
- Is it possible to replace the static set  $R_i$  by a dynamic set (as done in the mutex algorithm of Fig. 10.7)?
- Can the message exchange pattern described in Fig. 11.3 occur?
- What are the lower and upper bounds on the number of messages per use of a set of  $k$  instances of the resource?
- Is the algorithm adaptive?
- Let the waiting time be the time spent in the **wait** statement. Compare the waiting time of the previous algorithm with the waiting time of the algorithm described in Fig. 11.4. From a waiting time point of view, is one algorithm better than the other?

Solution in [311].

2. Write the server code (i.e., the code associated with message receptions) of the generalized  $k$ -out-of- $M$  algorithm for simultaneous requests whose client code is described in Fig. 11.14.
3. The generalized  $k$ -out-of- $M$  algorithm for simultaneous requests in dynamic sessions described in Fig. 11.14 uses timestamp.

Assuming the conflict graph of Fig. 11.6 in which each resource has a single instance (hence  $k = 1$ ), let us consider an execution in which concurrently

- $p_2$  issues `acquire_resource(1)` (i.e.,  $p_2$  requests resource  $x_1$ ),
- $p_6$  issues `acquire_resource(1, 3)` (i.e.,  $p_6$  requests each of the resources  $x_1$  and  $x_3$ ),
- $p_4$  issues `acquire_resource(3)` (i.e.,  $p_4$  requests resource  $x_3$ ).

Moreover, let  $\langle h_i, i \rangle$  be the timestamp of the request of  $p_i$ . How are these requests served if

- $\langle h_2, 2 \rangle < \langle h_6, 6 \rangle < \langle h_4, 4 \rangle$ ?
- $\langle h_2, 2 \rangle < \langle h_4, 4 \rangle < \langle h_6, 6 \rangle$ ?

What can be concluded about the order in which the requests are served?

4. The algorithm of Fig. 11.14 solves the generalized  $k$ -out-of- $M$  problem for simultaneous requests in dynamic sessions. This algorithm uses timestamp and is consequently unbounded. Design a bounded algorithm for the same problem.
5. Write the full code of the algorithm for simultaneous requests in dynamic sessions (see Sect. 11.2.5), i.e., the code of (a) the corresponding operations `acquire_resource( $RX_i$ )` and `release_resource( $RX_i$ )`, and (b) the code associated with the corresponding message receptions. (As in the algorithm described in Fig. 11.14,  $RX_i$  denotes the dynamically defined set of resources that  $p_i$  needs for its current session.)

Elements for a solution in [242, 387].

## Part IV

# High-Level Communication Abstractions

This part of the book is on the enrichment of a base send/receive distributed message-passing system with high-level communication abstractions. Chapter 12 focuses on abstractions that ensure specific order properties on message delivery. The most important of such communication abstractions is causal message delivery (also called causal order). Chapter 13 is on the rendezvous communication abstraction and logically instantaneous communication (also called synchronous communication).

These communication abstractions reduce the asynchrony of the system and consequently can facilitate the design of distributed applications. Their aim is to hide “basic machinery” to users and offer them high-level communication operations so that they can concentrate only on the essence of the problem they have to solve.

# Chapter 12

## Order Constraints on Message Delivery

High-level communication abstractions offer communication operations which ensure order properties on message delivery. The simplest (and best known) order property is the *first in first out* (FIFO) property, which ensures that, on each channel, the messages are received in their sending order. Another order property on message delivery is captured by the total order broadcast abstraction, which was presented in Sect. 7.1.4. This communication abstraction ensures that all the messages are delivered in the same order at each process, and this order complies with their causal sending order.

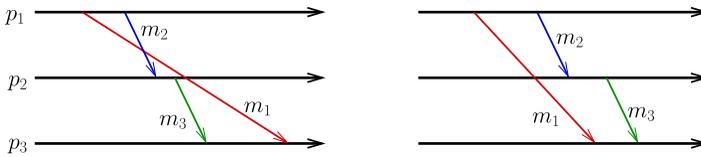
This chapter focuses first on causal message delivery. It defines the corresponding message delivery property and presents several algorithms that implement it, both for point-to-point communication and broadcast communication. Then the chapter presents new algorithms that implement the total order broadcast abstraction. Finally, the chapter plays with a channel by considering four order properties which can be associated with each channel taken individually.

When discussing a communication abstraction, it is assumed that all the messages sent at the application level are sent with the communication operation provided by this communication abstraction. Hence, there is no hidden relation on messages that will be unknown by the algorithms implementing these abstractions.

**Keywords** Asynchronous system · Bounded lifetime message · Causal barrier · Causal broadcast · Causal message delivery order · Circulating token · Client/server broadcast · Coordinator process · Delivery condition · First in first out (FIFO) channel · Order properties on a channel · Size of control information · Synchronous system

### 12.1 The Causal Message Delivery Abstraction

The notion of causal message delivery was introduced by K.P. Birman and T.A. Joseph (1987).



**Fig. 12.1** The causal message delivery order property

### 12.1.1 Definition of Causal Message Delivery

**The Problem** Let us consider the communication pattern described at the left of Fig. 12.1. Process  $p_1$  sends first the message  $m_1$  to  $p_3$  and then the message  $m_2$  to  $p_2$ . Moreover, after it has received  $m_2$ ,  $p_2$  sends the message  $m_3$  to  $p_3$ . Hence, when considering the partial order relation  $\xrightarrow{ev}$  on events (defined in Chap. 6), the sending of  $m_1$  belongs to the causal past of the sending of  $m_2$ , and (by the transitivity created by  $m_2$ ) belongs to the causal past of the sending of  $m_3$ . We consequently have  $s(m_1) \xrightarrow{ev} s(m_3)$  (where  $s(m)$  denotes the “sending of  $m$ ” event). But we do not have  $r(m_1) \xrightarrow{ev} r(m_3)$  (where  $r(m)$  denotes the “reception of  $m$ ” event).

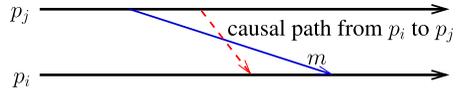
While the messages  $m_1$  and  $m_2$  are sent to the same destination process, and their sending are causally related, their reception order does not comply with their sending order. The causal message delivery order property (formally defined below) is not ensured. Differently, the reception order in the communication pattern described at the right in Fig. 12.1 is such that  $r(m_1) \xrightarrow{ev} r(m_3)$  and satisfies the causal message delivery order property.

**Definition** The *causal message delivery* (also called *causal order* or *causal message ordering*) abstraction provides the processes with two operations denoted `co_send()` and `co_deliver()`. When a process invokes them, we say that it `co_sends` or `co_delivers` a message. The abstraction is defined by the following properties. It is assumed that all messages are different (which can be easily realized by associating with each message a pair made up of a sequence number plus the identity of the sender process). Let  $co\_s(m)$  and  $co\_del(m)$  be the events associated with the `co_sending` of  $m$  and its `co_delivery`, respectively.

- **Validity.** If a process  $p_i$  `co_delivers` a message  $m$  from a process  $p_j$ , then  $m$  was `co_sent` by  $p_j$ .
- **Integrity.** No message is `co_delivered` more than once.
- **Causal delivery order.** For any pair of messages  $m$  and  $m'$ , if  $co\_s(m) \xrightarrow{ev} co\_s(m')$  and  $m$  and  $m'$  have the same destination process, we have  $co\_del(m) \xrightarrow{ev} co\_del(m')$ .
- **Termination.** Any message that was `co_sent` is `co_delivered` by its destination process.

This definition is similar to the one of the total order broadcast abstraction given in Sect. 7.1.4, from which the “total order” requirement is suppressed. The first

**Fig. 12.2**  
The delivery pattern prevented  
by the empty interval property



three requirements define the safety property of causal message delivery. Validity states that no message is created from thin air or is corrupted. Integrity states that there is no message duplication. Causal order states the added value provided by the abstraction. The last requirement (termination) is a liveness property stating that no message is lost.

While Fig. 12.1 considers a causal chain involving only two messages ( $m_2$  and  $m_3$ ), the length of such a chain in the third requirement can be arbitrary.

**A Geometrical Remark** As suggested by the right side of Fig. 12.1, causal order on message delivery is nothing more than the application of the famous “triangle inequality” to messages.

### 12.1.2 A Causality-Based Characterization of Causal Message Delivery

Let us recall the following definitions associated with each event  $e$  (Sect. 6.1.3):

- Causal past of  $e$ :  $\text{past}(e) = \{f \mid f \xrightarrow{ev} e\}$ ,
- Causal future of  $e$ :  $\text{future}(e) = \{f \mid e \xrightarrow{ev} f\}$ .

Let us consider an execution in which the messages are sent with the operations  $\text{send}()$  and  $\text{receive}()$ , respectively. Moreover, let  $M$  be the set of messages that have been sent. The message exchange pattern of this execution satisfies the causal message delivery property, if and only if we have

$$\forall m \in M: \text{future}(s(m)) \cap \text{past}(r(m)) = \emptyset,$$

or equivalently,

$$\forall m \in M: \{e \mid (s(m) \xrightarrow{ev} e) \wedge (e \xrightarrow{ev} r(m))\} = \emptyset.$$

This formula (illustrated in Fig. 12.2) states that, for any message  $m$ , there is a single causal path from  $s(m)$  to  $r(m)$  (namely the path made up of the two events  $s(m)$  followed by  $r(m)$ ). Considered as a predicate, this formula is called the *empty interval* property.

**Table 12.1** Hierarchies of communication abstractions

Point-to-point	Asynchronous $\prec$ FIFO channels $\prec$ Causal message delivery
Broadcast	Asynchronous $\prec$ FIFO channels $\prec$ Causal message delivery $\prec$ Total order

### 12.1.3 Causal Order with Respect to Other Message Ordering Constraints

It is important to notice that two messages  $m$  and  $m'$ , which have been co\_sent to the same destination process and whose co\_sendings are not causally related, can be received in any order. Concerning the abstraction power of causal message delivery with respect to other ordering constraints, we have the following:

- The message delivery constraint guaranteed by FIFO channels is simply causal order on each channel taken separately. Consequently, FIFO channels define an ordering property weaker than causal delivery.
- Let us extend causal message delivery from point-to-point communication to broadcast communication. This is addressed in Sect. 12.3. Let us recall that the total order broadcast abstraction presented in Sect. 7.1.4 is stronger than causal broadcast. It is actually “causal broadcast” plus “same message delivery order at each process” (even the messages whose co\_broadcasts are not causally related must be co\_delivered in the same order at any process).

We consequently have the hierarchies of communication abstractions described in Table 12.1, where “asynchronous” means no constraint on message delivery, and  $\prec$  means “strictly weaker than”.

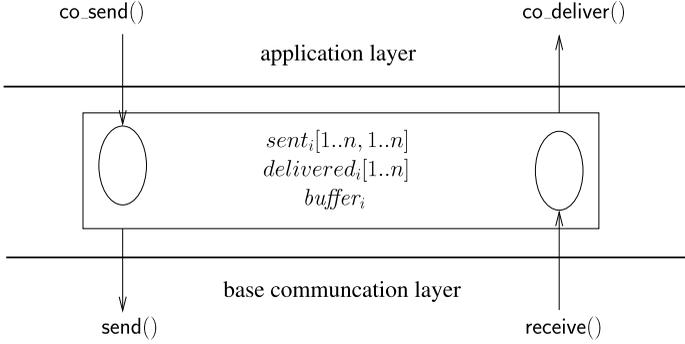
## 12.2 A Basic Algorithm for Point-to-Point Causal Message Delivery

### 12.2.1 A Simple Algorithm

The algorithm described in this section is due to M. Raynal, A. Schiper, and S. Toueg (1991). It assumes that no process sends messages to itself.

**Underlying Principle** As done when implementing the FIFO property on top of a non-FIFO channel, a simple solution consists in storing in a local buffer the messages whose delivery at reception time would violate causal delivery order. The corresponding structure of an implementation at a process  $p_i$  is described in Fig. 12.3.

The key of the algorithm consists in devising a delivery condition that allows us (a) to delay the delivery of messages that arrive “too early” (to ensure the safety property of message deliveries), and (b) to eventually deliver all the messages (to ensure the liveness property). To that end, each process manages the following local data structures.



**Fig. 12.3** Structure of a causal message delivery implementation

- $sent_i[1..n, 1..n]$  is an array of integers, each initialized to 0. The entry  $sent_i[k, \ell]$  represents the number of messages `co_send` by  $p_k$  to  $p_\ell$ , as known by  $p_i$ . (Let us recall that a causal message chain ending at a process  $p_i$  is the only way for  $p_i$  to “learn” new information.)
- $delivered_i[1..n]$  is an array of integers, each initialized to 0. The entry  $delivered_i[j]$  represents the number of messages `co_delivered` by  $p_i$ , which have been `co_send` by  $p_j$  to  $p_i$ .
- $buffer_i$  is the local buffer where  $p_i$  stores the messages that have been received and cannot yet be `co_delivered`. The algorithm is expressed at an abstraction level at which the use of this buffer is implicit.

**Delivery Condition** When a process  $p_j$  `co_sends` a message  $m$  to a process  $p_i$ , it associates with  $m$  its current knowledge of which messages have been `co_send` in the system, i.e., the current value of its array  $sent_j[1..n, 1..n]$ . Let  $CO(m, sent_j)$  denote the corresponding message sent to  $p_j$ .

When  $p_i$  receives, at the underlying network level, the message  $CO(m, sent)$  from  $p_j$ , it is allowed to `co_deliver`  $m$  only if it has already `co_delivered` all the messages  $m'$  which have been sent to it and are such that  $co\_s(m') \xrightarrow{ev} co\_s(m)$ . This delivery condition is captured by the following predicate, which can be locally evaluated by  $p_i$ :

$$DC(m) \equiv (\forall k : delivered_i[k] \geq sent[k, i]).$$

Due to its definition,  $sent[k, i]$  is the number of messages sent by  $p_k$  to  $p_i$ , to  $m$ 's knowledge (i.e., as known by  $p_j$  when it sent  $m$ ). Hence, if  $delivered_i[k] \geq sent[k, i]$ ,  $p_i$  has already `co_delivered` all the messages  $m'$  whose sending is in the causal past of the event  $co\_s(m)$ . If this is true for all  $k \in \{1, \dots, n\}$ ,  $p_i$  can safely `co_deliver`  $m$ . If there exists a process  $p_k$  such that  $delivered_i[k] < sent[k, i]$ , there is at least one message sent by  $p_k$  to  $p_i$ , whose sending belongs to the causal past of  $m$ , which has not yet been `co_delivered` by  $p_i$ . In this case,  $m$  is stored in the local buffer  $buffer_i$ , and remains in this buffer until its delivery condition becomes true.

```

operation co_send( $m$ ) to  $p_j$  is
(1)  send  $\text{CO}(m, \text{sent}_i)$  to  $p_j$ ;
(2)   $\text{sent}_i[i, j] \leftarrow \text{sent}_i[i, j] + 1$ .

when  $\text{CO}(m, \text{sent})$  is received from  $p_j$  do
(3)  wait ( $\forall k : \text{delivered}_i[k] \geq \text{sent}[k, i]$ );
(4)  co_delivery of  $m$  to the application layer;
(5)   $\text{sent}_i[j, i] \leftarrow \text{sent}_i[j, i] + 1$ ;
(6)   $\text{delivered}_i[j] \leftarrow \text{delivered}_i[j] + 1$ ;
(7)  for each  $(x, y) \in \{1, \dots, n\}^2$ 
(8)    do  $\text{sent}_i[x, y] \leftarrow \max(\text{sent}_i[x, y], \text{sent}[x, y])$ 
(9)  end for.

```

**Fig. 12.4** An implementation of causal message delivery (code for  $p_i$ )

**The Algorithm** The algorithm based on the previous data structures is described in Fig. 12.4. When a process  $p_i$  invokes  $\text{co\_send}(m)$ , it first sends the message  $\text{CO}(m, \text{sent}_i)$  to the destination process  $p_j$  (line 1), and then increases the sequence number  $\text{sent}_i[i, j]$ , which counts the number of messages  $\text{co\_sent}$  by  $p_i$  to  $p_j$  (line 2). Let us notice that the sequence number of an application message  $m$  is carried by the algorithm message  $\text{CO}(m, \text{sent})$  (this sequence number is equal to  $\text{sent}[i, j] + 1$ ).

When  $p_i$  receives from the network a message  $\text{CO}(m, \text{sent})$  from a process  $p_j$ , it stores the message in  $\text{buffer}_i$  until its delivery condition  $\text{DC}(m)$  becomes satisfied (line 3). When this occurs,  $m$  is  $\text{co\_delivered}$  (line 4), and the control variables are updated to take this  $\text{co\_delivery}$  into account. First  $\text{sent}_i[j, i]$  and  $\text{delivered}_i[j]$  are increased (lines 5–6) to record the fact that  $m$  has been  $\text{co\_delivered}$ . Moreover, the knowledge on the causal past of  $m$  (which is captured in  $\text{sent}[1..n, 1..n]$ ) is added to current knowledge of  $p_i$  (namely, every local variable  $\text{sent}_i[x, y]$  is updated to  $\max(\text{sent}_i[x, y], \text{sent}[x, y])$ , lines 7–9).

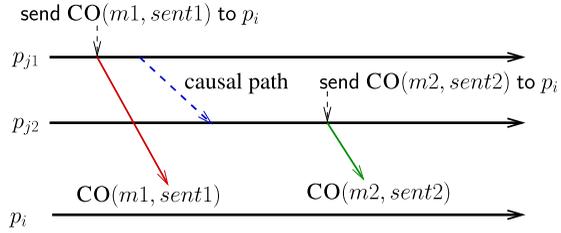
It is assumed that a message is  $\text{co\_delivered}$  as soon as its delivery condition becomes true. If, due to the  $\text{co\_delivery}$  of some message  $m$ , the conditions associated with several messages become true simultaneously, these messages are  $\text{co\_delivered}$  in any order, one after the other.

**Remark** Let us observe that, for any pair  $(i, j)$ , both  $\text{sent}_i[j, i]$  and  $\text{delivered}_i[j]$  are initialized to 0 and are updated the same way at the same time (lines 5–6). It follows that the vector  $\text{delivered}_i[1..n]$  can be replaced by the vector  $\text{sent}_i[j, 1..n]$ . Line 6 can then be suppressed and the delivery condition becomes

$$\text{DC}(m) \equiv (\forall k : \text{sent}_i[k, i] \geq \text{sent}[k, i]).$$

In the following we nevertheless consider the algorithm as written in Fig. 12.4 as it is easier to understand.

**Fig. 12.5** Message pattern for the proof of the causal order delivery



### 12.2.2 Proof of the Algorithm

**Lemma 8** Let  $m$  be an application message sent by  $p_j$  to  $p_i$ , and  $sent[1..n, 1..n]$  the control information attached to this message. When considering all the messages  $co\_sent$  by  $p_j$  to  $p_i$ , and assuming that sequence numbers start at value 1,  $sent[j, i] + 1$  is the sequence number of  $m$ .

*Proof* As the process  $p_j$  does not  $co\_send$  messages to itself, the only line at which  $sent_j[j, i]$  is modified is line 2. The proof trivially follows from the initialization of  $sent_j[j, i]$  to 0 and the sequentiality of the lines 1 and 2.  $\square$

**Lemma 9** Message  $co\_delivery$  respects causal delivery order.

*Proof* Let  $m1$  and  $m2$  be two application messages such that (a)  $co\_send(m1)$  causally precedes  $co\_send(m2)$ , (b) both are  $co\_sent$  to the same process  $p_i$ , (c)  $m1$  is  $co\_sent$  by  $p_{j1}$ , and (d)  $m2$  is  $co\_sent$  by  $p_{j2}$ . Moreover, let  $CO(m1, sent1)$  and  $CO(m2, sent2)$  be the corresponding messages sent by the algorithm (see Fig. 12.5).

As there is a causal path from  $co\_send(m1)$  to  $co\_send(m2)$ , it follows from the fact that line 2 is executed between  $send(m1, sent1)$  and  $send(m2, sent2)$  that we have  $sent1[j1, i] < sent2[j1, i]$ . We consider two cases.

- $j1 = j2$  ( $m1$  and  $m2$  are  $co\_sent$  by the same sender). As  $delivered_i[j1]$  is the sequence number of the last message  $co\_delivered$  from  $p_{j1}$ , it follows from (a) the predicate  $delivered_i[j1] \geq sent[j1, i]$ , and (b) the fact that  $sent[j1, i]$  is the sequence number of the previous message  $co\_sent$  by  $p_{j1}$  to  $p_i$  (Lemma 8), that the messages  $co\_sent$  by  $p_{j1}$  to  $p_i$  are  $co\_delivered$  according to their sequence number. This proves the lemma for the messages  $co\_sent$  by the same process.
- $j1 \neq j2$  ( $m1$  and  $m2$  are  $co\_sent$  by distinct senders). If the delivery condition of  $m2$  is satisfied we have  $delivered_i[j1] \geq sent2[j1, i]$ , which means that, as  $sent2[j1, i] > sent1[j1, i]$ , we have then  $delivered_i[j1] > sent1[j1, i]$ . But, it follows from the previous item that the messages from  $p_{j1}$  are received according their sequence numbers, from which we conclude that  $m1$  has been previously  $co\_delivered$ , which concludes the proof of the lemma.  $\square$

**Lemma 10** Any message that is  $co\_sent$  by a process is  $co\_delivered$  by its destination process.

*Proof* Let  $\prec$  denote the following relation on the application messages. Let  $m$  and  $m'$  be any pair of application messages;  $m \prec m'$  if  $\text{co\_send}(m)$  causally precedes  $\text{co\_send}(m')$ . As the causal precedence relation  $\xrightarrow{ev}$  on events is a partial order, so is the relation  $\prec$ .

Given a process  $p_i$ , let  $\text{pending}_i$  be the set of messages which have been  $\text{co\_sent}$  to it and are never  $\text{co\_delivered}$ . Assuming  $\text{pending}_i \neq \emptyset$ , let  $m \in \text{pending}_i$  a message that is minimal with respect  $\prec$  (i.e., a message which has no predecessor—according to  $\prec$ —in  $\text{pending}_i$ ).

As  $m$  cannot be  $\text{co\_delivered}$  by  $p_i$  there is (at least) one process  $p_k$  such that  $\text{delivered}_i[k] < \text{sent}[k, i]$  (line 3). It then follows from Lemma 9 that there is a message  $m'$  such that (a)  $m'$  has been  $\text{co\_sent}$  by  $p_k$  to  $p_i$ ,  $\text{co\_send}(m')$  causally precedes  $\text{co\_send}(m)$ , and  $m'$  is not  $\text{co\_delivered}$  by  $p_i$  (Fig. 12.5 can still be considered after replacing  $m_1$  by  $m'$  and  $m_2$  by  $m$ ). Hence this message belongs to  $\text{pending}_i$  and is such that  $m' \prec m$ . But this contradicts the fact that  $m$  is minimal in  $\text{pending}_i$ , which concludes the proof of the liveness property.  $\square$

**Theorem 17** *The algorithm described in Fig. 12.4 implements the causal message delivery abstraction.*

*Proof* The proof of the validity and integrity properties are trivial and are left to the reader. The proof of the causal message delivery follows from Lemma 9, and the proof of the termination property follows from Lemma 10.  $\square$

### 12.2.3 Reduce the Size of Control Information Carried by Messages

The main drawback of the previous algorithm lies in the size of the control information that has to be attached to each application message  $m$ . Let  $b$  be the number of bits used for each entry of a matrix  $\text{sent}_i[1..n, 1..n]$ . As a process does not send message to itself, the diagonal  $\text{sent}_i[j, j]$ ,  $1 \leq j \leq n$ , can be saved. The size of the control information that is transmitted with each application message is consequently  $(n^2 - n)b$  bits. This section shows how this number can be reduced.

**Basic Principle** Let us consider a process  $p_i$  that sends a message  $\text{CO}(m, \text{sent}_i)$  to a process  $p_j$ . The idea consists in sending to  $p_j$  only the set of values of the entries of the matrix  $\text{sent}_i$  that have been modified since the last message  $\text{CO}()$  sent to  $p_j$ . The set of values that has to be transmitted by  $p_i$  to  $p_j$  is the set of 3-tuples

$$\{(k, \ell, \text{sent}_i[k, \ell]) \mid \text{sent}_i[k, \ell] \text{ has been modified since the last } \text{co\_send} \text{ of } p_i \text{ to } p_j\}.$$

Let us remember that a similar approach was used in Sect. 7.3.2 to reduce the size of the vector dates carried by messages.

**A First Solution** An easy solution consists in directing each process to  $p_i$  manage  $(n - 1)$  additional matrices, one per process  $p_j$ ,  $j \neq i$ , in such a way that the matrix  $last\_sent_i[j]$  contains the value of the matrix  $sent_i$  when  $p_i$  sent its last message to  $p_j$ . Each such matrix is initialized as  $sent_i$  (i.e., 0 everywhere). The code of the operation  $co\_send(m)$  then becomes:

```

operation  $co\_send(m)$  to  $p_j$  is
  let  $set_i = \{\langle k, \ell, sent_i[k, \ell] \rangle \mid sent_i[k, \ell] \neq last\_sent_i[j][k, \ell]\}$ ;
  send  $CO(m, set_i)$  to  $p_j$ ;
   $last\_sent_i[j] \leftarrow sent_i$ ;
   $sent_i[i, j] \leftarrow sent_i[i, j] + 1$ .

```

The code associated with the reception of a message  $CO(m, set)$  can be easily modified to reconstruct the matrix  $sent_i$  used in the algorithm of Fig. 12.4.

As far as local memory is concerned, this approach costs  $n - 1$  additional matrices (without their diagonal) per process, i.e.,  $(n - 1)(n^2 - n)b \in O(n^3b)$  bits per process. It is consequently worthwhile only when  $n$  is small.

**A Better Solution: Data Structures** This section presents a solution for which the additional data structures at each process need only  $(n^2 + 1)b$  bits. These data structures are the following.

- $clock_i$  is a local logical clock which measures the progress of  $p_i$ , counted as the number of messages it has  $co\_sent$  (i.e.,  $clock_i$  counts the number of relevant events—here they are the invocations of  $co\_send()$ —issued by  $p_i$ ). Initially,  $clock_i = 0$ .
- $last\_send\_to_i[1..n]$  is a vector, initialized to  $[0, \dots, 0]$ , such that  $last\_send\_to_i[j]$  records the local date of the last  $co\_send$  to  $p_j$ .
- $last\_mod_i[1..n, 1..n]$  is an array such that  $last\_mod_i[k, \ell]$  is the local date of the last modification of  $sent_i[k, \ell]$ . The initial value of each  $last\_mod_i[k, \ell]$  is  $-1$ .

As the diagonal of  $last\_mod_i$  is useless, it can be used to store the vector  $last\_send\_to_i[1..n]$ .

**A Better Solution: Algorithm** The corresponding algorithm is described in Fig. 12.6. When a process wants to  $co\_send$  a message  $m$  to a process  $p_j$ , it first computes the set of entries of  $sent_i[1..n, 1..n]$  that have been modified since the last message it sent to  $p_j$  (line 1). Then, it attaches the corresponding tuples to  $m$ , sends them to  $p_j$  (line 2), and increases the local clock (line 3). Finally,  $p_i$  updates the other control variables:  $sent_i[i, j]$  (as in the basic algorithm) and  $last\_i[i, j]$  (line 4); and  $last\_sent\_i[j]$  (line 5).

When  $p_i$  receives a message  $CO(m, set)$  from  $p_j$ , it first checks the delivery condition (line 6). Let us notice that the delivery condition is now only on the pairs  $(k, i)$  such that  $\langle k, i, - \rangle \in set$ . This is because, for each pair  $(k', i)$  such that  $\langle k', i, - \rangle \notin set$ , the local variable  $sent_j[k', i]$  of the sender  $p_j$  has not been modified since its last sending to  $p_i$ . Consequently the test  $delivered_i[k'] \geq sent_j[k', i]$  was done when the message  $m'$  carrying  $\langle k', i, - \rangle$  (previously sent by  $p_j$  to  $p_i$ ) is

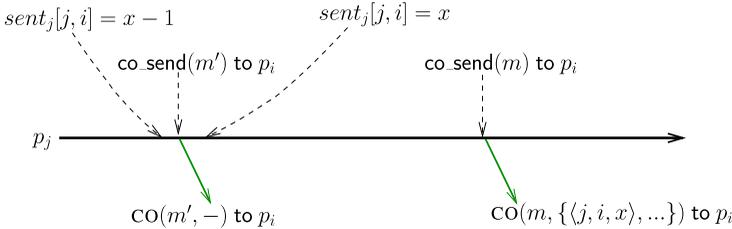
```

operation co_send( $m$ ) to  $p_j$  is
(1) let  $set_i = \{(k, \ell, sent_i[k, \ell]) \mid last\_mod_i[k, \ell] \geq last\_sent\_to_i[j]\}$ ;
(2) send CO( $m, set_i$ ) to  $p_j$ ;
(3)  $clock_i \leftarrow clock_i + 1$ ;
(4)  $sent_i[i, j] \leftarrow sent_i[i, j] + 1$ ;  $last\_mod_i[i, j] \leftarrow clock_i$ ;
(5)  $last\_sent\_to_i[j] \leftarrow clock_i$ .

when CO( $m, set$ ) is received from  $p_j$  do
(6) wait ( $\forall (k, \ell, x) \in set : delivered_i[k] \geq x$ );
(7) co_delivery of  $m$  to the application layer;
(8)  $delivered_i[j] \leftarrow delivered_i[j] + 1$ ;
(9)  $sent_i[j, i] \leftarrow sent_i[j, i] + 1$ ;  $last\_mod_i[i, j] \leftarrow clock_i$ ;
(10) for each  $(k, \ell, x) \in set_i$  do
(11)   if ( $sent_i[k, \ell] < x$ ) then  $sent_i[k, \ell] \leftarrow x$ ;  $last\_mod_i[k, \ell] \leftarrow clock_i$  end if
(12) end for.

```

**Fig. 12.6** An implementation reducing the size of control information (code for  $p_i$ )



**Fig. 12.7** Control information carried by consecutive messages sent by  $p_j$  to  $p_i$

received by  $p_i$ . Let us notice that—except possibly for the first message  $co\_send$  by  $p_j$ —the set  $set$  cannot be empty because there is at least the triple  $\langle j, i, x \rangle$ , where  $x$  is the sequence number of the previous message  $co\_send$  by  $p_j$  to  $p_i$ . (See Fig. 12.7.)

After  $m$  has been  $co\_delivered$ ,  $p_i$  updates its local control variables  $delivered_i[j]$ ,  $sent_i[j, i]$  as in the basic algorithm (line 8–9). It also updates entries of  $sent_i[1..n, 1..n]$  according to the values it has received in the set  $set_i$  (line 10–12).

**An Adaptive Solution** As done in Sect. 7.3.2 for vector clocks, it is possible to combine the basic algorithm of Fig. 12.4 with the algorithm of Fig. 12.6 to obtain an adaptive algorithm. The resulting sending procedure to be used at line 2 of Fig. 12.6 is described in Fig. 12.8.

As we can see, the test at line 2 is a simple switch that directs  $p_i$  to attach the least control information to the message  $m$ . The associated delivery condition is then that of Fig. 12.4 or that of Fig. 12.6 according to the tag of the message.

```

before sending CO(m, -) to pj is
(1) let s = |seti|;
(2) if ((n2 - n)b < s(2 log2 n + b))
(3)   then tag the message to be sent with "0" and send the full matrix senti
(4)   else tag the message to be sent with "1" and send the set seti
(5) end if.
    
```

Fig. 12.8 An adaptive sending procedure for causal message delivery

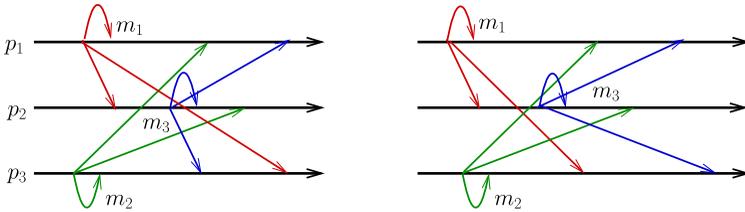


Fig. 12.9 Illustration of causal broadcast

## 12.3 Causal Broadcast

### 12.3.1 Definition and a Simple Algorithm

**Definition** Causal broadcast is a communication abstraction that ensures causal message delivery in the context of broadcast communication. It provides processes with two operations denoted `co_broadcast()` and `co_deliver()`. The only difference from point-to-point communication is that each message has to be `co_delivered` by all the processes (including its sender).

An example is depicted in Fig. 12.9. The message pattern on the left side does not satisfy the causal message delivery property. This is due to the messages  $m_1$  and  $m_3$ : while the broadcast of  $m_1$  causally precedes the broadcast of  $m_3$ , their delivery order at  $p_3$  does not comply with their broadcast order. Differently, as there is no causal relation linking the broadcast of  $m_1$  and  $m_2$ , they can be delivered in any other at each process (and similarly for  $m_2$  and  $m_3$ ). The message pattern on the right side of the figure satisfies the causal message delivery property.

**A Simple Causal Broadcast Algorithm** A simple algorithm can be derived from the point-to-point algorithm described in Fig. 12.4. As the sending of a message  $m$  to a process is replaced by the sending of  $m$  to all processes, the matrix `senti[1..n, 1..n]` can be shrunk into a vector `broadcasti[j]` such that

$$broadcast_i[j] = sent_i[j, 1] = \dots = sent_i[j, n],$$

which means that `broadcasti[j]` represents the number of messages that, to  $p_i$ 's knowledge, have been broadcast by  $p_j$ . (The initial value of `broadcasti[j]` is 0.)

```

operation co_broadcast( $m$ ) is
(1) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do send CO_BR( $m, broadcast_i[1..n]$ ) to  $p_j$  end for;
(2)  $broadcast_i[i] \leftarrow broadcast_i[i] + 1$ ;
(3) co_delivery of  $m$  to the application layer.

when CO_BR( $m, broadcast[1..n]$ ) is received from  $p_j$  do
(4) wait ( $\forall k : broadcast_i[k] \geq broadcast[k]$ );
(5) co_delivery of  $m$  to the application layer;
(6)  $broadcast_i[j] \leftarrow broadcast_i[j] + 1$ .

```

**Fig. 12.10** A simple algorithm for causal broadcast (code for  $p_i$ )

In addition to the previous observation, let us remark that when a process  $p_i$  which co\_broadcasts a message, can locally co\_deliver it at the very same time. This is because the local co\_delivery of such a message  $m$  cannot depend on messages not yet co\_delivered by  $p_i$  (it causally depends only on the messages previously co\_broadcast or co\_delivered by  $p_i$ ).

The corresponding algorithm is described in Fig. 12.10. When a process  $p_i$  invokes co\_broadcast( $m$ ), it sends the message CO\_BR( $m, broadcast_i$ ) to each other process (line 1), increases accordingly  $broadcast_i[i]$  (line 2), and co\_delivers  $m$  to itself (line 3).

When it receives a message CO\_BR( $m, broadcast$ ), a process  $p_i$  first checks the delivery condition (line 4). As the sequence numbers of all the messages  $m'$  that have been co\_broadcast in the causal past of  $m$  are registered in the vector  $broadcast[1..n]$ , the delivery condition is

$$(\forall k : broadcast_i[k] \geq broadcast[k]).$$

When this condition becomes true,  $m$  is locally co\_delivered (line 5), and  $broadcast_i[j]$  is increased to register this co\_delivery of a message co\_broadcast by  $p_j$  (line 6).

**Remark on the Vectors  $broadcast_i$**  Let us notice that each array  $broadcast_i[1..n]$  is nothing more than a vector clock, where the local progress of each process is measured by the number of messages it has co\_broadcast. Due to the delivery condition, the update at line 6 is equivalent to the vector clock update

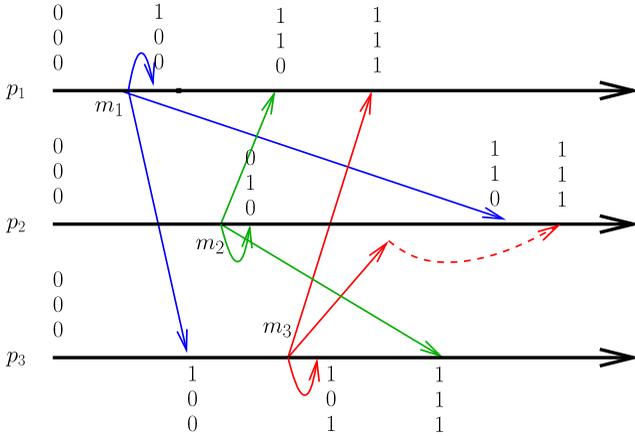
```

for each  $k \in \{1, \dots, n\}$ 
  do  $broadcast_i[k] \leftarrow \max(broadcast_i[k], broadcast[m])$ 
end for.

```

It follows that the technique presented in Sect. 7.3.2 to reduce the size of control information carried by application messages can be used.

**An Example** An example of an execution of the previous algorithm is described in Fig. 12.11. The co\_broadcast of the application messages  $m_1$  and  $m_2$  are independent (not causally related). The co\_broadcast of  $m_1$  generates the algorithm message CO\_BR( $m_1, [0, 0, 0]$ ). As it has an empty causal past (from the co\_broadcast



**Fig. 12.11** The causal broadcast algorithm in action

point of view), this message can be co\_delivered as soon as it arrives at a process. We have the same for the application message  $m_2$ , which gives rise to the algorithm message  $CO\_BR(m_2, [0, 0, 0])$ . When this messages arrives at  $p_1$  we have  $broadcast_1 = [1, 0, 0] > [0, 0, 0]$ , and after  $p_1$  has co\_delivered  $m_2$ , we have  $broadcast_1 = [1, 1, 0]$ , witnessing that the co\_broadcast of  $m_1$  and  $m_2$  are in the causal past of the next message that will co\_broadcast by  $p_1$ .

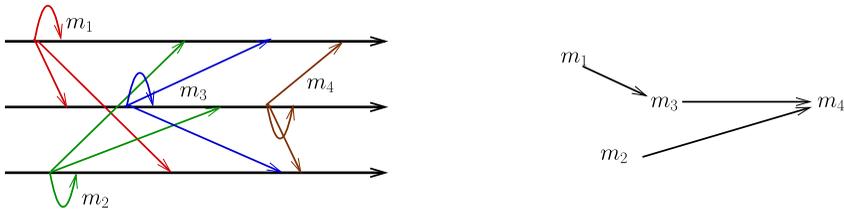
Differently, the co\_broadcast of  $m_1$  belongs to the causal past of  $m_3$ , and  $p_3$  consequently sends the algorithm message  $CO\_BR(m_3, [1, 0, 0])$ . As  $broadcast_1 = [1, 1, 0]$  when  $p_1$  receives  $CO\_BR(m_3, [1, 0, 0])$ ,  $p_1$  can immediately co\_deliver  $m_3$  when it receives  $CO\_BR(m_3, [1, 0, 0])$ . Differently,  $broadcast_2 = [0, 1, 0]$  when  $p_2$  receives  $CO\_BR(m_3, [1, 0, 0])$ . As  $broadcast_2[1] = 0 < 1$ ,  $p_2$  is forced to delay the co\_delivery of  $m_3$  until it has co\_delivered  $m_1$ .

### 12.3.2 The Notion of a Causal Barrier

This section presents a simple causal broadcast algorithm that reduces the size of the control information carried by messages.

**Message Causality Graph** Let  $M$  be the set of all the messages which are co\_broadcast during the execution of an application. Let us define a partial order relation, denoted  $<_{im}$ , on application messages as follows. Let  $m$  and  $m'$  be any two application messages. We have  $m <_{im} m'$  (read:  $m$  precedes immediately  $m'$  in the message graph) if:

- The co\_broadcast( $m$ ) causally precedes co\_broadcast( $m'$ ).
- There is no message  $m''$  such that (a) co\_broadcast( $m$ ) causally precedes co\_broadcast( $m''$ ), and (b) co\_broadcast( $m''$ ) causally precedes co\_broadcast( $m'$ ).



**Fig. 12.12** The graph of immediate predecessor messages

An example is depicted on Fig. 12.12. The execution is on the left side, and the corresponding message graph is on the right side. This graph, which captures the immediate causal precedence on the co\_broadcast of messages, states that the co\_delivery of a message depends only on the co\_delivery of its immediate predecessors in the graph. As an example, the co\_delivery of  $m_4$  is constrained by the co\_delivery of  $m_3$  and  $m_2$ , and the co\_delivery of  $m_3$  is constrained by the co\_delivery of  $m_1$ . Hence, the co\_delivery of  $m_4$  is not (directly) constrained by the co\_delivery of  $m_1$ . It follows that a message has to carry control information only on its immediate predecessors in the graph defined by the relation  $<_{im}$ .

**Causal Barrier and Local Data Structures** Let us associate with each application message an identity (a pair made up of a sequence number plus a process identity). The *causal barrier* associated with an application message is the set of identities of the messages that are its immediate predecessors in the message causality graph. Each process manages accordingly the following data structures.

- $causal\_barrier_i$  is the set of identities of the messages that are immediate predecessors of the next message that will be co\_broadcast by  $p_i$ . This set is initially empty.
- $delivered_i[1..n]$  has the same meaning as in previous algorithms. It is initialized to  $[0, \dots, 0]$ , and  $delivered_i[j]$  contains the sequence number of the last message from  $p_j$  that has been co\_delivered by  $p_i$ .
- $sn_i$  is a local integer variable initialized to 0. It counts the number of messages that have been co\_broadcast by  $p_i$ .

**Algorithm** The corresponding algorithm, in which the causal barrier of a message  $m$  constitutes the control information attached to it, is described in Fig. 12.13.

When a process  $p_i$  invokes  $co\_broadcast(m)$ , it first builds the message  $CO\_BR(m, causal\_barrier_i)$  and sends it to all the other processes (line 1). According to the definition of  $causal\_barrier_i$ , a destination process will be able to co\_deliver  $m$  only after having co\_delivered all the messages whose identities belong to  $causal\_barrier_i$ . Then,  $p_i$  co\_delivers locally  $m$  (line 2), increases  $sn_i$  (line 3), and resets its causal barrier to  $\{(sn_i, i)\}$  (line 4). This is because the co\_delivery of the next message co\_broadcast by  $p_i$  will be constrained by  $m$  (whose identity is the pair  $(sn_i, i)$ ).

```

operation co_broadcast( $m$ ) is
(1) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do send CO_BR( $m, causal\_barrier_i$ ) to  $p_j$  end for;
(2) co_delivery of  $m$  to the application layer;
(3)  $sn_i[i] \leftarrow sn_i + 1$ ;
(4)  $causal\_barrier_i \leftarrow \{sn_i, i\}$ .

when CO_BR( $m, causal\_barrier$ ) is received from  $p_j$  do
(5) wait ( $\forall (sn, k) \in causal\_barrier : delivered_i[k] \geq sn$ );
(6) co_delivery of  $m$  to the application layer;
(7)  $delivered_i[j] \leftarrow delivered_i[j] + 1$ ;
(8)  $causal\_barrier_i \leftarrow (causal\_barrier_i \setminus causal\_barrier) \cup \{(delivered_i[j], j)\}$ .

```

**Fig. 12.13** A causal broadcast algorithm based on causal barriers (code for  $p_i$ )

When it receives an algorithm message  $(m, causal\_barrier)$  from  $p_j$ ,  $p_i$  delays the co\_delivery of  $m$  until it has co\_delivered all the messages whose identity belongs to  $causal\_barrier$ , i.e., until it has co\_delivered all the immediate predecessors of  $m$  (lines 5–6). Then,  $p_i$  updates  $delivered_i[j]$  (line 7). Finally,  $p_i$  updates its causal barrier (line 8). To that end,  $p_i$  first suppresses from it the message identities which are in  $causal\_barrier$  (this is because, due to delivery condition, it has already co\_delivered the corresponding messages). Then,  $p_i$  adds to  $causal\_barrier_i$  the identity of the message  $m$  it has just co\_delivered, namely the pair  $\langle delivered_i[j], j \rangle$  (this is because, the message  $m$  will be an immediate predecessor of the next message that  $p_i$  will co\_broadcast).

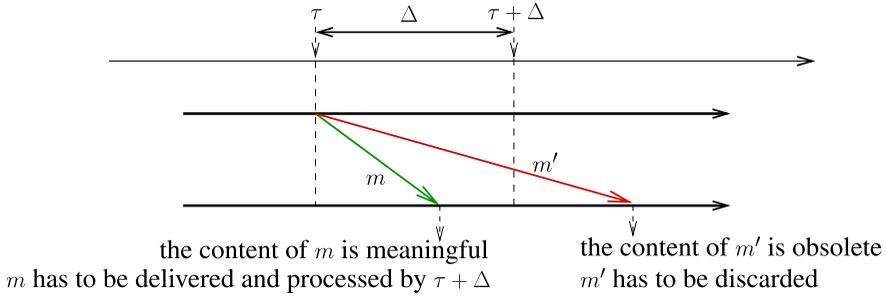
Let us finally observe that the initial size of a set  $causal\_barrier$  is 0. Then, as soon as  $p_i$  has co\_broadcast or co\_delivered a message, we have  $1 \leq |causal\_barrier_i| \leq n$ .

### 12.3.3 Causal Broadcast with Bounded Lifetime Messages

This section has two aims: show the versatility of the causal barrier notion and introduce the notion of messages with bounded lifetime. The algorithm presented in this section is due to R. Baldoni, R. Prakash, M. Raynal, and M. Singhal (1998).

**Asynchronous System with a Global Clock** Up to now we have considered fully asynchronous systems in which there is no notion of physical time accessible to processes. Hence, no notion of expiry date can be associated with messages in such systems.

We now consider that, while the speed of each process and the transit time of each message remains arbitrary (asynchrony assumption), the processes have access to a common physical clock that they can only read. This global clock is denoted *CLOCK*. (Such a common clock can be implemented when the distributed application covers a restricted geographical area.)



**Fig. 12.14** Message with bounded lifetime

**Bounded Lifetime Message** The *lifetime* of a message is the physical time duration during which, after the message has been sent, its content is meaningful and can consequently be used by its destination process(es). A message that arrives at its destination process after its lifetime has elapsed becomes useless and must be discarded. For the destination process, it is as if the message was lost. A message that arrives at a destination process before its lifetime has elapsed must be delivered by the expiration of its lifetime.

For simplicity, we assume that all the messages have the same lifetime  $\Delta$  and processing times are negligible when compared to message transit times.

Let  $\tau$  be the sending time of a message. The physical date  $\tau + \Delta$  is the *deadline* after which this message is useless for its destination process. This is illustrated in Fig. 12.14. The message  $m$  arrives by its deadline and must be processed by its deadline. On the opposite,  $m'$  arrives after its deadline and must be discarded.

It is also assumed that the lifetime  $\Delta$  is such that, in practice, a great percentage of messages arrives by their deadline, as is usually the case in distributed multimedia applications.

**$\Delta$ -Causal Broadcast**  $\Delta$ -causal broadcast is a causal broadcast in a context in which the application messages have a bounded lifetime  $\Delta$ . It is defined by the operations `co_broadcast()` and `co_deliver()` which satisfy the following properties.

- **Validity.** If a process  $p_i$  `co_delivers` a message  $m$  from a process  $p_j$ ,  $m$  was `co_broadcast` by  $p_j$ .
- **Integrity.** No message is `co_delivered` more than once.
- **Causal delivery order.** For any pair of messages  $m$  and  $m'$  that arrives at a process  $p_j$  by their deadlines,  $p_j$  `co_delivers`  $m$  before  $m'$  if `co_broadcast`( $m$ ) causally precedes `co_broadcast`( $m'$ ).
- **Expiry constraint.** No message that arrives after its deadline at a process is `co_delivered` by this process.
- **Termination.** Any message that arrives at a destination process  $p_j$  by its deadline is processed by  $p_j$  by its deadline.

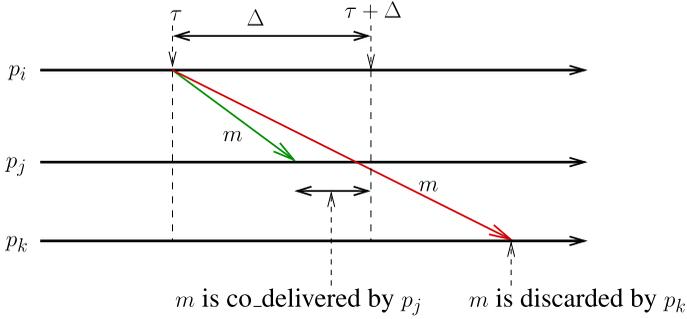


Fig. 12.15 On-time versus too late

As shown in Fig. 12.15, it is possible for a message  $m$  to be co\_delivered by its deadline at a process  $p_j$  (termination property), and discarded at another process  $p_k$  (expiry constraint).

**Message Delivery Condition** Let us replace the sequence numbers associated with messages in the time-free algorithm of Fig. 12.13 by their sending dates, as defined by the common clock  $CLOCK$ . Consequently, the identity of a message  $m$  is now a pair made up of a physical date (denoted)  $sdt$  plus a process identity.

An algorithm message sent by a process  $p_j$  carries now three parts: the concerned application message  $m$ , its sending date  $st$ , and the current value of  $causal\_barrier_j$ .

When,  $p_i$  receives such a message  $CO\_BR(m, st, causal\_barrier)$  from  $p_j$ ,  $p_i$  discards the message if it arrives too late, i.e., if  $CLOCK - st > \Delta$ . In the other case ( $CLOCK - st \leq \Delta$ ), the message has to be co\_delivered by time  $st + \Delta$ . If  $\Delta$  was equal to  $+\infty$ , all the messages would arrive by their deadlines, and they all would have to be co\_delivered. The delivery condition would then be that of the time-free algorithm, where sequence numbers are replaced by physical sending dates, i.e., we would have

$$DC'(m) \equiv (\forall \langle sdt, k \rangle \in causal\_barrier : delivered_i[k] \geq sdt).$$

When  $\Delta \neq +\infty$ , it is possible that there is a process  $p_k$  that, in the causal past of  $co\_broadcast(m)$ , has co\_broadcast a message  $m'$  (identified  $\langle sdt, k \rangle$ ) such that (a)  $\langle sdt, k \rangle \in causal\_barrier$ , (b)  $delivered_i[k] > sdt$ , and (c)  $m'$  will be discarded because it will arrive after its deadline. The fact that (i) the co\_broadcast of  $m'$  causally precedes the co\_broadcast of  $m$ , and (ii)  $m'$  is not co\_delivered, has not to prevent  $p_i$  from co\_delivering  $m$ . To solve this issue,  $p_i$  delays the co\_delivery of  $m$  until the deadline of  $m'$  (namely  $sdt + \Delta$ ), but no more. The final delivery condition is consequently

$$DC'(m) \equiv \forall \langle sdt, k \rangle \in causal\_barrier : \begin{cases} (delivered_i[k] \geq sdt) \\ \vee (CLOCK - sdt > \Delta). \end{cases}$$

```

operation co_broadcast( $m$ ) is
(1)  $st \leftarrow \text{CLOCK}$ ;
(2) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do send CO_BR( $m, st, \text{causal\_barrier}_i$ ) to  $p_j$  end for;
(3) co_delivery of  $m$  to the application layer;
(4)  $\text{causal\_barrier}_i \leftarrow \{(st, i)\}$ .

when CO_BR( $m, st, \text{causal\_barrier}$ ) is received from  $p_j$  do
(5)  $\text{current\_time} \leftarrow \text{CLOCK}$ ;
(6) if  $\text{current\_time} > st + \Delta$ 
(7)   then discard  $m$ 
(8)   else wait ( $\forall (std, k) \in \text{causal\_barrier} :$ 
(9)      $(\text{delivered}_i[k] \geq sdt) \vee (\text{current\_time} > std + \Delta)$ );
(10)   co_delivery of  $m$  to the application layer;
(11)    $\text{delivered}_i[j] \leftarrow sdt$ ;
(12)    $\text{causal\_barrier}_i \leftarrow (\text{causal\_barrier}_i \setminus \text{causal\_barrier}) \cup \{(sdt, j)\}$ ;
(13) end if.

```

**Fig. 12.16** A  $\Delta$ -causal broadcast algorithm (code for  $p_i$ )

**Algorithm** The corresponding  $\Delta$ -causal broadcast algorithm is described in Fig. 12.16. It is a simple adaptation of the previous time-free causal broadcast algorithm to (a) physical time and (b) messages with a bounded lifetime  $\Delta$ . (Taking  $\Delta = +\infty$  provides us with a deadline-free causal order algorithm based on physical clocks.)

As already said, it is assumed that the duration of local processing is 0. Moreover, the granularity of the common clock is assumed to be such that any two consecutive invocations of `co_broadcast()` by a process have different dates.

A message `CO_BR( $m, st, \text{causal\_barrier}_i$ )` sent at line 2 carries its sending date  $st$ , so that the receiver can discard it as soon as it arrives, if it arrives too late (lines 6–7).

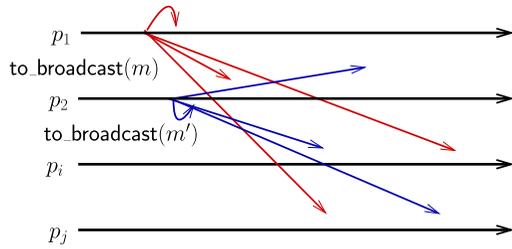
## 12.4 The Total Order Broadcast Abstraction

### 12.4.1 Strong Total Order Versus Weak Total Order

**Strong Total Order Broadcast Abstraction** The total order broadcast abstraction was introduced in Sect. 7.1.4 to illustrate the use of scalar clocks. For the self-completeness of this chapter, its definition is repeated here. This abstraction provides the processes with two operations denoted `to_broadcast()` and `to_deliver()` which satisfy the following properties. Let us recall that it is assumed that all the messages which are `co_broadcast` are different.

- **Validity.** If a process `to_delivers` a message  $m$ , there is a process that has `to_broadcast`  $m$ .
- **Integrity.** No message is `to_delivered` more than once.

**Fig. 12.17** Implementation of total order message delivery requires coordination



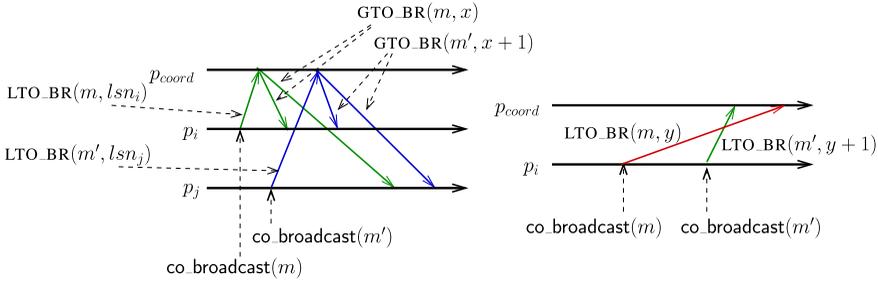
- Total order. If a process `to_delivers`  $m$  before  $m'$ , no process `to_delivers`  $m'$  before  $m$ .
- Causal precedence order. If the `to_broadcast` of  $m$  causally precedes the `to_broadcast` of  $m'$ , no process `to_delivers`  $m'$  before  $m$ .
- Termination. If a process `to_broadcasts` a message  $m$ , every process `to_delivers`  $m$ .

In the following this communication abstraction is called *strong* total order abstraction.

**Weak Total Order Broadcast Abstraction** Some applications do not require that message delivery complies with causal order. For these applications, the important point is the fact that the messages are delivered in the same order at each process, the causality among their broadcast being irrelevant. This defines a *weak* total order abstraction, which is total order broadcast without the “causal precedence order” requirement.

**Causal Order Versus Total Order** As has been seen in the algorithms ensuring causal message delivery, when a process  $p_i$  receives an algorithm message carrying an application message  $m$ , it can be forced to delay the local delivery of  $m$  (until the associated delivery condition becomes true), but it is not required to coordinate with other processes.

This is the fundamental difference with the algorithms that implement total order message delivery. Let us consider Fig. 12.17 in which, independently,  $p_1$  invokes `to_broadcast( $m$ )` and  $p_2$  invokes `to_broadcast( $m'$ )`. Let us consider any two distinct processes  $p_i$  and  $p_j$ . As the `co_broadcasts` of  $m$  and  $m'$  are not causally related, it is possible that  $p_i$  receives first the algorithm message carrying  $m$  and then the one carrying  $m'$ , while  $p_j$  receives them in the opposite order. If the message delivery requirement was causal order, there will be no problem, but this is no longer the case for total order:  $p_i$  and  $p_j$  have to coordinate in one way or another, directly or indirectly, to agree on the same delivery order. This explains why the algorithms implementing the total order message delivery requirement are inherently more costly (in terms of time and additional exchanged messages) than the algorithms implementing only causal message delivery.



**Fig. 12.18** Total order broadcast based on a coordinator process

### 12.4.2 An Algorithm Based on a Coordinator Process or a Circulating Token

**Using a Coordinator Process** A simple solution to implement a total order on message delivery consists in using a coordinator process  $p_{coord}$ , as depicted in the example on the left of Fig. 12.18. More precisely, we have the following.

- Each process  $p_i$  manages a local variable  $lsn_i$ , which is used to associate a sequence number with each application message it to\_broadcasts. Then, when a process  $p_i$  invokes to\_broadcast( $m$ ), it increases  $lsn_i$  and sends the algorithm message  $LTO\_BR(m, lsn_i)$  to  $p_{coord}$ .
- The coordinator process  $p_{coord}$  manages a local variable  $gsn_i$ , used to associate a global sequence number (i.e., a sequence number whose scope is the whole system) with each message  $m$  it receives.

For each process  $p_i$ , the coordinator processes the messages  $LTO\_BR(m, lsn)$  it receives from  $p_i$  according to their local sequence numbers. Hence, when looking at the arrival pattern described on the right of Fig. 12.18,  $p_{coord}$  is able to recover the sending order of  $m$  and  $m'$ .

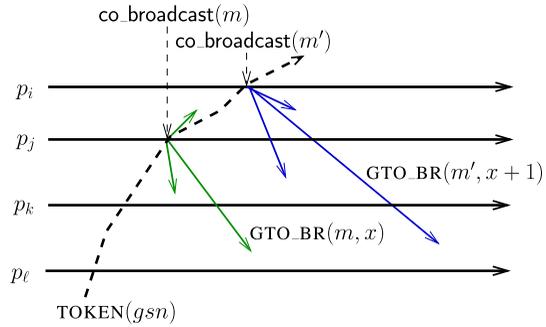
The processing of a message  $m$  is as follows. First  $p_{coord}$  increases  $gsn_i$ , and associates the new value with  $m$ . It then sends the message  $GTO\_BR(m, gsn_i)$  to all the processes.

- A process  $p_i$  to\_delivers the application messages  $m$ , that it receives in algorithm messages  $GTO\_BR(m, gsn)$ , according to their global sequence numbers.

This simple algorithm implements the strong total order broadcast abstraction (i.e., total order on the delivery of application messages which complies with message causal precedence). It is a two-phase algorithm: For each application message  $m$ , a process sends first an algorithm message to the coordinator process, and then the coordinator sends an algorithm message to all.

**Replacing the Coordinator Process by a Token** The coordinator process can be replaced by a mobile token which acts as a moving coordinator. The token message, denoted  $TOKEN(gsn)$ , carries a sequence number generator  $gsn$ , initialized

**Fig. 12.19** Token-based total order broadcast



to 0. When a process  $p_i$  invokes `co_broadcast(m)`, it waits for the token and, when it receives the token, it increases  $gsn$  by 1, whose new value becomes the global sequence number associated with  $m$ . Then,  $p_i$  sends the message `GTO_BR(m, gsn)` to all the processes. Finally, a process  $p_j$  delivers the application messages it receives according to their sequence numbers. An example is depicted in Fig. 12.19.

As far as the moves of the token are concerned, two approaches are possible.

- As the token is a mobile object, any of the algorithms presented in Chap. 5 can be used. A process that needs a global sequence number invokes the corresponding operations `acquire_object()` and `release_object()`.

This approach can be interesting when the underlying navigation algorithm is adaptive as, in this case, only the processes that want to `to_broadcast` messages are required to participate in the navigation algorithm.

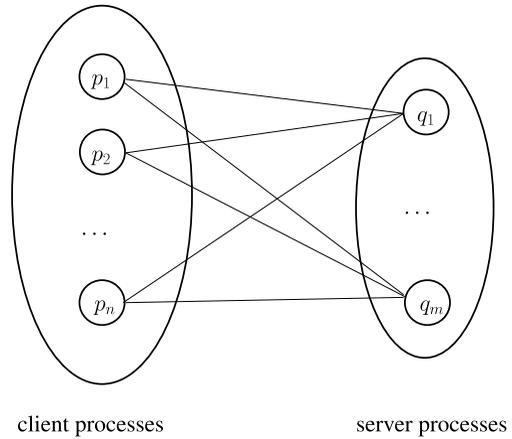
- The processes are arranged along a unidirectional logical ring, and the token moves perpetually along the ring. When a process receives the token, it computes the next sequence number (if needed), and then forwards the token to the next process on the ring.

This solution has the drawback that all the processes are required to participate in the move of the token, even if they do not want to `co_broadcast` messages. This approach is consequently more interesting in applications where all the processes very frequently want to `to_broadcast` messages.

**Mutual Exclusion Versus Total Order** As shown by the previous algorithms, there is a strong relation between mutual exclusion and total order broadcast. In both cases, a total order has to be implemented: on the accesses to a critical section, or on the delivery of messages.

Nevertheless, mutual exclusion and total order broadcast differ in the sense that, in addition to a total order, mutual exclusion prevents two or more processes from being simultaneously in the critical section (hence the operation `release_resource()`, which needs to be invoked to allow the next process to enter the critical section; such a “release” operation does not exist in total order broadcast).

**Fig. 12.20**  
Clients and servers  
in total order broadcast



### 12.4.3 An Inquiry-Based Algorithm

**Structural Decomposition: Clients, Servers, and State Machine** Let us consider that the processes are decomposed into two groups: the client processes  $p_1, \dots, p_n$ , and the server processes  $q_1, \dots, q_m$  (see Fig. 12.20). Of course several processes can be both client and server, but, to simplify the presentation and without loss of generality, we consider here clients and servers as distinct entities. This algorithm is due to D. Skeen (1982).

The clients broadcast messages to the servers, and all servers have to deliver these messages in the same order. Hence, this structure is particularly well suited to the duplication of a deterministic state machine (e.g., a queue, a stack, etc.), where each server maintains a copy of the state machine. When a process wants to issue an operation (or command) on the state machine, it builds a message describing this operation and broadcasts it to all the servers using a total order broadcast communication abstraction. The same sequence of operations (command) will consequently be applied on all the copies of the state machine.

**Underlying Principle** The idea is to associate a timestamp with each application message and to deliver messages according to their timestamp order. The dates used in timestamps, which can be considered as delivery dates, are computed as follows from logical scalar clocks associated with each server.

When a process  $p_i$  wants to broadcast an application message  $m$ , it sends an inquiry message to each server  $q_j$ , which sends back to it the current value of its clock. This value is a proposed delivery date for  $m$ . When  $p_i$  has received all the dates proposed for the delivery of  $m$ , it computes the maximal one, which becomes the final delivery date of  $m$ , and sends it to all the servers. As a same delivery date can be associated with different messages, (as already announced) the control data associated with  $m$  is actually a timestamp, i.e., a pair made up of the delivery date of  $m$  and the identity of the client that sent it. Finally, each server delivers the application messages according to their timestamp order.

**Communication Graph** Let us remark that in the communication pattern generated by the previous process and server behavior, the clients (resp., the servers) do not communicate among themselves: A client communicates only with servers and a server communicates only with clients. The communication graph is a bipartite graph with a channel connecting each client and each server as depicted in Fig. 12.20.

**Local Variable at a Client Process** A client  $p_i$  manages a single local variable denoted  $sn_i$ . Initialized to 0, this variable allows  $p_i$  to associate a sequence number with every application message that it to\_broadcasts. Hence, each pair  $(sn_i, i)$  is the identity of a distinct message.

**Local Variables at a Server Process** A server process  $q_j$  manages three local variables.

- $clock_j$ , is the local scalar clock of  $q_j$ . It is initialized to 0.
- $pending_i$  is a set, initialized to  $\emptyset$ , which contains the application messages (and associated control data) received and not yet to\_delivered by  $q_j$ .
- $to\_deliverable_i$  is a queue, initially empty, at the tail of which  $q_j$  deposits the next to\_delivered message.

**Delivery Condition** The set  $pending_i$  contains tuples of the form  $\langle m, date, i, tag \rangle$ , where  $m$  is an application message,  $i$  the identity of its sender,  $date$  the tentative delivery date currently associated with  $m$ , and  $tag$  is a control bit. If  $tag = no\_del$ , the message  $m$  has not yet been assigned its final delivery date and cannot consequently be to\_delivered (added to  $to\_deliverable_j$ ). Let us notice that its final delivery date will be greater than or equal to its current tentative delivery date. If  $tag = del$ ,  $m$  has been assigned its final delivery date and can be to\_delivered if it is stable. Stability means that no message in  $pending_i$  (and by transitivity, no message to\_broadcast in the future) can have a timestamp smaller than the one associated with  $m$ .

The delivery condition  $DC(m)$  for a message  $m$  such that  $\langle m, date, i, del \rangle \in pending_i$  is consequently the following (let us recall that all application messages are assumed to be different):

$$\forall \langle m', date', i', - \rangle \in pending_i: \\ (m' \neq m) \Rightarrow [(date_i < date_{i'}) \vee (date_i = date_{i'} \wedge i < i')].$$

**The Total Order Broadcast Algorithm** The corresponding total order broadcast algorithm is described in Fig. 12.21. A client process  $p_i$  is allowed to invoke again to\_broadcast() only when it has completed its previous invocation (this means that, while it is waiting at line 3, a process remains blocked until it has received the appropriate messages).

When it invokes to\_broadcast( $m$ ),  $p_i$  first computes the sequence number that will identify  $m$  (line 1). It then sends the message INQUIRY( $m, sn_i$ ) to each server (line 2) and waits for their delivery date proposals (line 3). When  $p_i$  has received

```

===== on client side =====
operation co_broadcast( $m$ ) by a client  $p_i$  is
(1)  $sn_i \leftarrow sn_i + 1$ ;
(2) for each  $j \in \{1, \dots, m\}$  do send INQUIRY( $m, sn_i$ ) to  $q_j$  end for;
(3) wait (a message PROP_DATE( $sn_i, d_j$ ) has been received from each  $q_j$ );
(4) Let  $date \leftarrow \max(\{d_j\}_{1 \leq j \leq m})$ ;
(5) for each  $j \in \{1, \dots, m\}$  do send FINAL_DATE( $sn_i, date$ ) to  $q_j$  end for.

===== on server side =====
when INQUIRY( $m, sn$ ) is received from  $p_i$  by a server  $q_j$  do
(6)  $msg\_sn \leftarrow sn$ ;
(7)  $clock_j \leftarrow clock_j + 1$ ;
(8)  $pending_j \leftarrow pending_j \cup \{ \langle m, clock_j, i, no\_del \rangle \}$ ;
(9) send PROP_DATE( $sn, clock_i$ ) to  $p_i$ ;
(10) wait (FINAL_DATE( $sn, date$ ) received from  $p_i$  where  $sn = msg\_sn$ );
(11) replace  $\langle m, -, i, no\_del \rangle$  in  $pending_j$  by  $\langle m, date, i, del \rangle$ ;
(12)  $clock_i \leftarrow \max(clock_i, date)$ .

back ground task  $T$  is
(13) repeat forever
(14) wait (  $\exists \langle m, date, i, del \rangle \in pending_i$  such that  $\forall \langle m', date', i', - \rangle \in pending_i$ :
            $(m' \neq m) \Rightarrow [(date_i < date_{i'}) \vee (date_i = date_{i'} \wedge i < i')]$ );
(15) withdraw  $\langle m, date, i, del \rangle$  from  $pending_i$ ;
(16) add  $m$  at the tail of  $to\_deliverable_j$ 
(17) end repeat.

```

**Fig. 12.21** A total order algorithm from clients  $p_i$  to servers  $q_j$

them,  $p_i$  computes the final delivery date of  $m$  (line 4) and sends it to the servers (line 5). (As we can see, this is nothing more than a classical handshake coordination mechanism which has been already encountered in other chapters.)

The behavior of a server  $q_j$  is made up of two parts. When  $q_j$  receives a message TO\_1( $m, sn$ ) from a client  $p_i$ ,  $q_j$  stores the sequence of the message (line 6), and increases its local clock (line 7), adds  $\langle m, clock_j, i, no\_del \rangle$  to  $pending_j$  (line 8), and sends to  $p_i$  a proposed delivery date for  $m$  (line 9). Then, as far as  $m$  is concerned,  $q_j$  waits until it has received the final date associated with  $m$  (line 10). When this occurs, it replaces in  $pending_j$  the proposed date by the final delivery date and marks the message as deliverable (line 11), and updates its local scalar clock (line 12).

The second processing part associated with a server is a background task which suppresses messages from  $pending_j$  and adds them at the tail of the queue  $to\_deliverable_j$ . The core of this task, described at lines 13–17, is the delivery condition  $DC(m)$ , which has been previously introduced.

#### 12.4.4 An Algorithm for Synchronous Systems

This section considers total order broadcast in a synchronous system. The algorithm which is described is a simplified version of a fault-tolerant algorithm due to F. Cristian, H. Aghili, R. Strong, and D. Dolev (1995).

```

operation co_broadcast( $m$ ) is
(1)  $sdt \leftarrow CLOCK$ ;
(2) for each  $j \in \{1, \dots, m\}$  do send TO_BR( $m, sdt$ ) to  $p_j$  end for.

when ( $m, sdt$ ) is received from  $p_j$  do
(3)  $pending_j \leftarrow pending_j \cup \{(m, sdt + \Delta)\}$ .

back ground task  $T$  is
(4) repeat forever
(5)   let  $dmin =$  smallest date in  $pending_i$ ;
(6)   wait( $CLOCK = dmin$ );
(7)   withdraw from  $pending_i$  the messages whose delivery date is  $dmin$ ,
(8)   and add them at the tail of  $to\_deliverable_j$  in increasing timestamp order
(9)   end repeat.

```

**Fig. 12.22** A total order algorithm for synchronous systems

**Synchronous System** The synchrony provided to the processes by the system is defined as follows:

- There is an upper bound, denoted  $\Delta$ , on message transit duration.
- There is a global physical clock, denoted  $CLOCK$ , that all the processes can read. (The case where  $CLOCK$  is implemented with local physical clocks—which can drift—is considered in Problem 8.)

The granularity of this clock is such that no two messages sent by the same process are sent at the same physical date.

**The Algorithm** The algorithm is described in Fig. 12.22. When a process invokes  $co\_broadcast(m)$  it sends the message  $TO\_BR(m, sdt)$  to all the processes (including itself), where  $sdt$  is the message sending date.

When a process  $p_i$  receives a message  $TO\_BR(m, sdt)$ ,  $p_i$  delays its delivery until time  $sdt + \Delta$ . If several messages have the same delivery date, they are  $to\_delivered$  according their timestamp order (i.e., according to the identity of their senders). The local variables  $pending_i$  and  $to\_deliverable_i$  have the same meaning as in the previous algorithms.

As we can see, this algorithm is based on the following principle: It systematically delays the delivery of each message as if its transit duration was equal to the upper bound  $\Delta$ . Hence, this algorithm reduces all cases to the worst-case scenario. It is easy to see that the total order on message delivery, which is the total order on their sending times (with process identities used to order the messages sent at the same time), is the same at all the processes.

**Total Order in Synchronous Versus Asynchronous Systems** Whether the system is synchronous or asynchronous, let  $\hat{\Delta}$  be the average message transit time. Assuming that processing times are negligible with respect to transit times, this means

that, in the average, the `to_delivery` of a message takes  $2\widehat{\Delta}$  in the coordinator-based algorithm of Sect. 12.4.2, and  $3\widehat{\Delta}$  in the client–server algorithm of Sect. 12.4.3. Differently, the `to_delivery` of a message takes always  $\Delta$  in the synchronous algorithm of Fig. 12.22. It follows that, when considering algorithms executed on top of a synchronous system, an asynchronous algorithm can be more efficient than a synchronous algorithm when  $2\widehat{\Delta} < \Delta$  (or  $3\widehat{\Delta} < \Delta$ ).

## 12.5 Playing with a Single Channel

Considering a channel, this section considers four ordering properties that can be associated with each message sent of this channel, and algorithms which implement them. This section has to be considered as an exercise in two-process communication.

### 12.5.1 Four Order Properties on a Channel

**Definitions** Let us consider a message  $m$  sent by a process  $p_i$  to a process  $p_j$ , and  $m'$  any other message sent by  $p_i$  to  $p_j$ . Four types of delivery constraints can be associated with the message  $m$ . Let  $s(m)$  and  $del(m)$  be the events “sending of  $m$ ” and delivery of  $m$ , respectively (and similarly for  $m'$ ). The four types of delivery constraints which can be imposed on  $m$  are denoted `ct_future`, `ct_past`, `marker`, and `ordinary`. They are defined as follows, where `type(m)` denotes the constraint associated with  $m$ .

- `type(m) = ct_future`. In this case,  $m$  cannot be bypassed by messages  $m'$  sent after it ( $m$  controls its future). Moreover, nothing prevents  $m$  from bypassing messages  $m''$  sent before it. This constraint is depicted in Fig. 12.23: all the messages  $m'$  sent after  $m$  are delivered by  $p_j$  after  $m$ .

Formally, if `type(m) = ct_future`, we have for any other message  $m'$  sent by  $p_i$  to  $p_j$ :

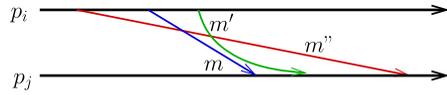
$$(s(m) \xrightarrow{ev} s(m')) \Rightarrow (del(m) \xrightarrow{ev} del(m')).$$

- `type(m) = ct_past`. In this case,  $m$  cannot bypass the messages  $m'$  sent before it ( $m$  is controlled by its past). Moreover, nothing prevents  $m$  from being bypassed by messages  $m''$  sent after it. This constraint is depicted in Fig. 12.24: all the messages  $m'$  sent before  $m$  are delivered by  $p_j$  before  $m$ .

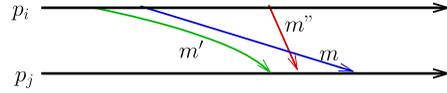
Formally, if `type(m) = ct_past`, we have for any other message  $m'$  sent by  $p_i$  to  $p_j$ :

$$(s(m') \xrightarrow{ev} s(m)) \Rightarrow (del(m') \xrightarrow{ev} del(m)).$$

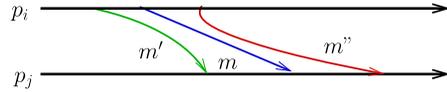
**Fig. 12.23** Message  $m$  with type `ct_future` (cannot be bypassed)



**Fig. 12.24** Message  $m$  with type `ct_past` (cannot bypass other messages)



**Fig. 12.25** Message  $m$  with type `marker`



- $\text{type}(m) = \text{marker}$ . In this case,  $m$  can neither bypass messages  $m'$  sent after it, nor be bypassed by messages  $m''$  sent before it. This constraint is depicted in Fig. 12.25. (This type of message has been implicitly used in Sect. 6.6, devoted to the determination of a consistent global state of a distributed computation.)

Formally, if  $\text{type}(m) = \text{marker}$ , we have for any other message  $m'$  sent by  $p_i$  to  $p_j$ :

$$[(s(m) \xrightarrow{ev} s(m')) \Rightarrow (del(m) \xrightarrow{ev} del(m'))] \\ \wedge [(s(m') \xrightarrow{ev} s(m)) \Rightarrow (del(m') \xrightarrow{ev} del(m))].$$

- $\text{type}(m) = \text{ordinary}$ . In this case,  $m$  imposes no delivery constraints on the other messages sent by  $p_i$  to  $p_j$ .

It is easy to see that if the type of all the messages sent by  $p_i$  to  $p_j$  is `marker`, the channel behaves as a FIFO channel. The same is true if the type of all the messages is either `ct_future` or `ct_past`.

### 12.5.2 A General Algorithm Implementing These Properties

This section presents an algorithm which ensures that the messages sent by  $p_i$  to  $p_j$  are delivered according to the constraints defined by their types.

**A Simple Algorithm** An algorithm that builds a FIFO channel on top of a non-FIFO channel can be used. Considering that (whatever their actual types) all the messages are typed `ordinary`, such an algorithm ensures that each message  $m$  behaves as if it was typed `marker`. As this type is stronger than the other types, all the constraints defined by the types of all the messages are trivially satisfied.

```

operation send( $m$ , type( $m$ )) by  $p_i$  to  $p_j$  is
(1)   $sn_i \leftarrow sn_i + 1$ ;
(2)  send ( $m$ ,  $sn_i$ ) to  $p_j$ .

when ( $m$ ,  $sn$ ) is received by  $p_j$  from  $p_i$  do
(3)  if ( $last\_sn_j + 1 \neq sn$ )
(4)    then  $pending_j \leftarrow pending_j \cup \{(m, sn)\}$ 
(5)    else deliver  $m$ ;  $last\_sn_j \leftarrow last\_sn_j + 1$ 
(6)      while ( $\exists (m', sn') \in pending_j : sn' = last\_sn_j + 1$ )
(7)        do deliver  $m'$ ;  $last\_sn_j \leftarrow last\_sn_j + 1$ 
(8)      end while
(9)  end if.

```

**Fig. 12.26** Building a *first in first out* channel

Such an algorithm is describe in Fig. 12.26. The sender  $p_i$  manages a local variable  $sn_i$  (initialized to 0), that it uses to associate a sequence number with each message. The receiver process  $p_j$  manages two local variables:  $last\_sn_j$  (initialized to 0) contains the sequence number of the last message of  $p_i$  that it has delivered;  $pending_j$  is a set (initially empty) which contains the messages (with their sequence numbers) received and not year delivered by  $p_j$ . The sequence numbers allows  $p_j$  to deliver the messages in their sending order.

**A Genuine Algorithm** The previous algorithm is stronger than necessary: Whatever the type of each message, it forces all of them to behave as if they were typed marker. This section presents a genuine algorithm, i.e., an algorithm which imposes to messages only the constraints defined by their type.

To that end, in addition to  $sn_i$  (managed by  $p_i$ ) and  $pending_j$  (managed by  $p_j$ ), the sender  $p_i$  and the receiver  $p_j$  manage the following local variables:

- $no\_bypass_i$  is a local variable of  $p_i$  that contains the sequence number of the last message which has been sent and cannot be bypassed by the messages that will be sent in the future. It is initialized to 0, and updated when  $p_i$  sends a message whose type is `ct_future` or `marker` (this is because, due to the definition of these types, such messages cannot be bypassed by messages sent in the future).
- $deliv\_sn_j$  is a set of  $p_j$  that contains the sequence numbers of all the messages that  $p_j$  has already delivered. Its initial value is  $\{0\}$  (the sequence numbers start at 1). (As the messages are not necessarily delivered in their sending order, a simple counter such as  $last\_sn_j$  is no longer sufficient to register which messages have been delivered.)

The genuine algorithm is described in Fig. 12.27. When  $p_i$  wants to send a message  $m$  to  $p_j$ , it associates the next sequence number  $sn_i$  with  $m$ , and sends the message  $(m, sn_i, type(m), barrier)$  to  $p_j$ . The value  $barrier$  is a control data, which is the sequence number of the last message  $m'$  that  $m$  cannot bypass. If `type(m)` is `ct_past` or `marker`, due to the definition of these types,  $m'$  is the last message that  $p_i$  sent before  $m$  (lines 3 and 4). If `type(m)` is `ct_future` or `ordinary`,  $barrier$  is set to the value of  $no\_bypass_i$  (lines 2 and 5). Moreover, if `type(m)` is

```

operation send( $m, \text{type}(m)$ ) by  $p_i$  to  $p_j$  is
(1)  $sn_i \leftarrow sn_i + 1$ ;
(2) case ( $\text{type}(m) = \text{ct\_future}$ ) then  $\text{barrier} \leftarrow \text{no\_bypass}_i$ ;  $\text{no\_bypass}_i \leftarrow sn_i$ 
(3)   ( $\text{type}(m) = \text{marker}$ )   then  $\text{barrier} \leftarrow ns_i - 1$ ;  $\text{no\_bypass}_i \leftarrow sn_i$ 
(4)   ( $\text{type}(m) = \text{ct\_past}$ )   then  $\text{barrier} \leftarrow ns_i - 1$ 
(5)   ( $\text{type}(m) = \text{ordinary}$ ) then  $\text{barrier} \leftarrow \text{no\_bypass}_i$ ;
(6) end case;
(7) send ( $m, sn_i, \text{type}(m), \text{barrier}$ ) to  $p_j$ .

when ( $m, sn, \text{type}, \text{barrier}$ ) is received from  $p_i$  do
(8) if [ $(\text{type}(m) \in \{\text{ct\_past}, \text{marker}\}) \wedge (\{1, \dots, \text{barrier}\} \subseteq \text{deliv\_sn}_j)$ 
     $\vee [(\text{type}(m) \in \{\text{ordinary}, \text{ct\_future}\}) \wedge (\text{barrier} \in \text{deliv\_sn}_j)]$ ]
(9) then deliver  $m$ ;  $\text{deliv\_sn}_j \leftarrow \text{deliv\_sn}_j \cup \{sn\}$ 
(10)   while ( $\exists (m', sn', \text{type}', \text{barrier}') \in \text{pending}_i$  such that  $DC(m')$ )
(11)     do deliver  $m'$ ;  $\text{deliv\_sn}_j \leftarrow \text{deliv\_sn}_j \cup \{sn'\}$ ;
(12)     withdraw ( $m', sn', \text{type}', \text{barrier}'$ ) from  $\text{pending}_i$ 
(13)   end while
(14) else add ( $m, sn, \text{type}, \text{barrier}$ ) to  $\text{pending}_i$ 
(15) end if.

```

Fig. 12.27 Message delivery according to message types

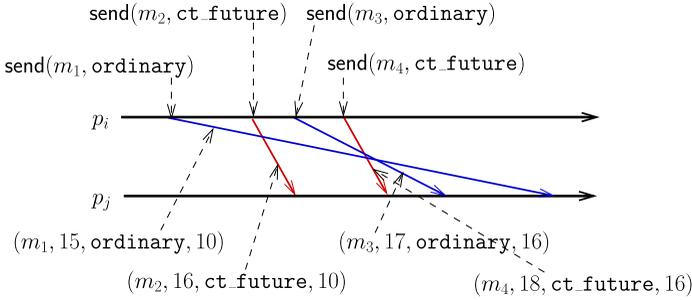
$\text{ct\_future}$  or  $\text{marker}$ , the local variable  $\text{no\_bypass}_i$  is set to  $sn_i$ , as no message sent in the future will be allowed to bypass  $m$  (whose sequence number is  $sn_i$ ).

When  $p_j$  receives an application message  $(m, sn, \text{type}, \text{barrier})$ , it checks the delivery condition (line 8). If the condition is false,  $p_j$  stores  $(m, sn, \text{type}, \text{barrier})$  in  $\text{pending}_j$  (line 14). If the condition is true,  $p_j$  delivers the application message  $m$ , and adds its sequence number to  $\text{deliv\_sn}_j$  (line 9). Moreover, it also delivers the messages in  $\text{pending}_j$  whose delivery condition has become true due to the delivery of  $m$ , or—transitively—the delivery of other messages (lines 10–13).

**The Delivery Condition** Let  $(m, sn, \text{type}, \text{barrier})$  be an algorithm message received by  $p_j$ . The delivery condition  $DC(m)$  associated with  $m$  depends on the type of  $m$ .

- If  $\text{type}(m)$  is  $\text{ct\_past}$  or  $\text{marker}$ ,  $m$  has to be delivered after all the messages sent before it, which means that we need to have  $\{1, \dots, \text{barrier}\} \subseteq \text{deliv\_sn}_j$ .
- If  $\text{type}(m)$  is  $\text{ct\_future}$  or  $\text{ordinary}$ ,  $m$  can be delivered before messages sent before it. Hence, the only constraint is that its delivery must not violate the requirements imposed by the type of other messages. But these requirements are captured by the message parameter  $\text{barrier}$ , which states that the delivery of the message whose sequence number is  $\text{barrier}$  has to occur before the delivery of  $m$ . Hence, for these message types, the delivery condition is  $\text{barrier} \in \text{deliv\_sn}_j$ .

An example is depicted in Fig. 12.28. The sequence numbers of  $m_1, m_2, m_3$ , and  $m_4$ , are 15, 16, 17, and 18, respectively. Before the sending of  $m_1$ ,  $\text{no\_bypass}_i = 10$ . These messages can be correctly delivered in the order  $m_2, m_4, m_3, m_1$ .



**Fig. 12.28** Delivery of messages typed ordinary and ct\_future

To summarize, the delivery condition  $DC(m)$  is:

$$DC(m) \equiv [(\text{type}(m) \in \{\text{ct\_past}, \text{marker}\}) \wedge (\{1, \dots, \text{barrier}\} \subseteq \text{deliv\_sn}_j)] \\ \vee [(\text{type}(m) \in \{\text{ordinary}, \text{ct\_future}\}) \wedge (\text{barrier} \in \text{deliv\_sn}_j)].$$

## 12.6 Summary

The aim of this chapter was to present two communication abstractions, namely, the causal message delivery abstraction and the total order broadcast abstraction. These abstractions have been defined and algorithms implementing them have been described. Variants suited to bounded lifetime messages and synchronous systems have also been presented. Finally, as an exercise, the chapter has investigated ordering properties which can be imposed on messages sent on a channel.

## 12.7 Bibliographic Notes

- The notion of message causal ordering is due to K.P. Birman and T.A. Joseph [53]. This notion was then extended to overlapping groups of processes and multicast communication in [55].  
Developments on the notion of causal order can be found in [88, 90, 217, 254, 340].
- The point-to-point causal message ordering described in Sect. 12.2 is due to M. Raynal, A. Schiper, and S. Toueg [324].  
Other causal order algorithms can be found in [255, 296, 336].
- Causal order-based algorithms which compute a consistent snapshot of a distributed computation are described in [1, 14].
- The technique to reduce the size of control information presented in Sect. 12.2.3 is due to F. Mattern.
- The causal broadcast algorithm of Sect. 12.3 is from [55, 324].

- The notion of a causal barrier and the associated causal broadcast algorithm of Sect. 12.3.2 are due to R. Baldoni, R. Prakash, M. Raynal, and M. Singhal [39]. This algorithm was extended to mobile environments in [298].
- The notion of bounded lifetime messages and the associated causal order algorithm presented in Sect. 12.3.3 are due to R. Baldoni, A. Mostéfaoui, and M. Raynal [38].
- The notion of total order broadcast was introduced in many systems (e.g., [84] for an early reference).
- The client/server algorithm presented in Sect. 12.4.3 is due to D. Skeen [353].
- The total order algorithm for synchronous systems presented in Sect. 12.4.4 is a simplified version of a fault-tolerant algorithm due to F. Cristian, H. Aghili, R. Strong, and D. Dolev [102].
- State machine replication was introduced by L. Lamport in [226]. A general presentation of state machine replication can be found in [339].
- The notions of the message types ordinary, marker, controlling the past, and controlling the future, are due to M. Ahuja [12]. The genuine algorithm presented in Sect. 12.5.2 is due to M. Ahuja and M. Raynal [13].
- A characterization of message ordering specifications and algorithms can be found in [274]. The interconnection of systems with different message delivery guarantees is addressed in [15].
- While this book is devoted to algorithms in reliable message-passing systems, the reader will find algorithms that implement causal message broadcast in asynchronous message-passing systems in [316], and algorithms that implement total order broadcast in these systems in [24, 67, 242, 316].

## 12.8 Exercises and Problems

1. Prove that the empty interval predicate stated in Sect. 12.1.2 is a characterization of causal message delivery.
2. Prove that the causal order broadcast algorithm described in Fig. 12.10 is correct. Solution in [39].
3. When considering the notion of a causal barrier introduced in Sect. 12.3.2, show that any two messages that belong simultaneously to  $causal\_barrier_i$  are independent (i.e., the corresponding invocations of `co_broadcast()` are not causally related).
4. Modify the physical time-free algorithm of Fig. 12.4 so that it implements causal message delivery in a system where the application messages have a bounded lifetime  $\Delta$ . Prove then that the resulting algorithm is correct. Solution in [38].
5. Let us consider an asynchronous system where the processes are structured into (possibly overlapping) groups. As an example, when considering five processes, a possible structuring into four groups is  $G_1 = \{p_1, p_2, p_4, p_5\}$ ,  $G_2 = \{p_2, p_3, p_4\}$ ,  $G_3 = \{p_3, p_3, p_4\}$ , and  $G_4 = \{p_1, p_5\}$ .

When a process sends a message  $m$ , it sends  $m$  to all the processes of a group to which it belongs. As an example, when  $p_2$  sends a message  $m$ , it sends  $m$  to the group  $G_1$  or to the group  $G_2$ . The sending of a message to a group is called *multicast*.

A simple causal multicast algorithm consists in using an underlying causal broadcast algorithm in which each process discards all the messages that have been multicast in a group to which it does not belong. While this solution works, it is not *genuine* in the sense that each process receives all messages.

Design a causal multicast algorithm in which a message sent to a group is sent only to the processes of this group.

Solution in [55].

6. As in the previous problem, let us consider an asynchronous system in which the processes are structured into (possibly overlapping) groups. Design a genuine total order multicast algorithm (i.e., an algorithm in which a message sent to a group is sent only to the processes of this group).

Solution in [55, 135]. (The algorithms described in these papers consider systems in which processes may crash.)

7. Let us consider the coordinator-based total order broadcast algorithm presented in Sect. 12.4.2.

- (a) Prove that this algorithm implements the strong total order broadcast abstraction.

- (b) Does this algorithm implement the strong total order broadcast abstraction when all sequence numbers are suppressed but channels to and from the coordinator process are FIFO?

- (c) Let us suppress the local variable  $lsn_i$  of each process  $p_i$  and replace each algorithm message  $LTO\_BR(m, lsn_i)$  by  $LTO\_BR(m)$ . Hence, the only sequence numbers used in this modified algorithm are the global sequence numbers generated by the coordinator process  $p_{coord}$ . Show that this modified algorithm implements the weak total order broadcast abstraction.

8. Let us consider a synchronous system where the global clock  $CLOCK$  is implemented with a physical clock  $ph\_clock_i$  per process  $p_i$ . Moreover, the synchronization of these physical clocks is such that their drift is bounded by  $\epsilon$ . This means that we always have  $|ph\_clock_i - ph\_clock_j| \leq \epsilon$  for any pair of processes  $p_i$  and  $p_j$ .

Modify the total order algorithm described in Fig. 12.22 so that it works with the previous physical clocks.

9. Let us consider a channel connecting the processes  $p_i$  and  $p_j$ , and the message types  $ct\_future$ ,  $ct\_past$ , and  $ct\_marker$ , introduced in Sect. 12.5. Show that it is impossible to implement the message type  $ct\_marker$  from messages typed  $ct\_future$  and  $ct\_past$ . (At the application level,  $p_i$  can send an arbitrary number of messages typed  $ct\_future$  or  $ct\_past$ .)

# Chapter 13

## Rendezvous (Synchronous) Communication

While the previous chapter was devoted to communication abstractions on message ordering, this chapter is on *synchronous* communication (also called *logically instantaneous* communication, or *rendezvous*, or *interaction*). This abstraction adds synchronization to communication. More precisely, it requires that, for a message to be sent by a process, the receiver has to be ready to receive it. From an external observer point view, the message transmission looks instantaneous: The sending and the reception of a message appear as a single event (and the sense of the communication could have been in the other direction). From an operational point of view, we have the following: For each pair of processes, the first process that wants to communicate—be it the sender or the receiver—has to wait until the other process is ready to communicate.

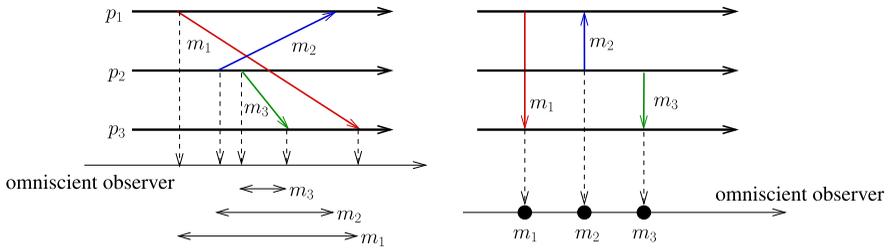
This chapter first defines synchronous communication and introduces a characterization based on a specific message pattern called a crown. It then presents several implementations of this communication abstraction, each suited to a specific context. It also describes implementations for real-time rendezvous in the context of synchronous message-passing systems. In this case, each process is required to associate a deadline with each of its rendezvous.

**Keywords** Asynchronous system · Client–server hierarchy · Communication initiative · Communicating sequential processes · Crown · Deadline-constrained interaction · Deterministic vs. nondeterministic context · Logically instantaneous communication · Planned vs. forced interaction · Rendezvous · Multiparty interaction · Synchronous communication · Synchronous system · Token

### 13.1 The Synchronous Communication Abstraction

#### 13.1.1 Definition

**Underlying Intuition** When considering an asynchronous message-passing system, two events are associated with each message, namely its send event and its receive event. This captures the fact that a message takes time to transit from its sender to its destination process, and this duration can be arbitrary.



**Fig. 13.1** Synchronous communication: messages as “points” instead of “intervals”

The intuition that underlies the definition of the synchronous communication abstraction is to consider messages as “points” instead of “time intervals” whose length is arbitrary. When considering messages as points (the send and receive events of each message being pieced together into a single point), the set of messages is structured as a partial order. As we will see below, this allows us to reason on distributed objects (each object being managed by a distinct process) as if they were all kept in a shared common memory. Hence, while it reduces asynchrony, synchronous communication makes reasoning and program analysis easier.

A simple example that shows how synchronous communication reduces messages as time intervals into messages as points is described in the right part of Fig. 13.1. From a space-time diagram point of view, reducing messages to points amounts to considering the transit time of each message as an arrow of zero duration (from an omniscient observer’s point of view), i.e., as a vertical arrow. Messages considered as intervals are described on the left part of Fig. 13.1. It is easy to see that this communication pattern does not comply with synchronous communication: if the transfer of  $m_2$  and  $m_3$  can be depicted as vertical arrows, that of  $m_1$  cannot. This is because the sending of  $m_1$  has to appear before the point  $m_2$ , which appears before the point  $m_3$ , which in turn appears before the reception of  $m_1$ . Such a schedule of events, which prevents a message pattern from being synchronous, is formalized below in Sect. 13.1.3.

**Sense of Message Transfer** As shown by the execution on the right of Fig. 13.1, it is easy to see that, when considering synchronous communication, the sense of direction for message transfer is irrelevant.

The fact that any of the messages  $m_1$ ,  $m_2$ , or  $m_3$  would be transmitted in the other direction, does not change the fact that message transfers remain points when considering synchronous communication. We can even assume such a point abstracts the fact that two messages are simultaneously exchanged, one in each direction. Hence, the notions of sender and receiver are not central in synchronous communication.

**Definition** The synchronous communication abstraction provides the processes with two operations, denoted `synchr_send()` and `synchr_deliver()`, which allow them to send and receive messages, respectively. As in the previous chapter, we say

“a process `synchr_sends` or `synchr_delivers` a message”, and assume (without loss of generality) that all the application messages are different. These operations satisfy the following properties, where the set  $H$  of the events that are considered are the events at program level. The following notations are used:  $sy\_s(m)$  and  $sy\_del(m)$  denote the events associated with the synchronous send and synchronous delivery of  $m$ , respectively.

- **Validity.** If a process  $p_i$  `synchr_delivers` a message from a process  $p_j$ , then  $p_j$  has `synchr_sent` this message.
- **Integrity.** No message is `synchr_delivered` more than once.
- **Synchrony.** Let  $\mathcal{D}$  be a dating function from the set of events  $H$  (program level) into the scalar time domain (set of natural numbers). We have:
  - For any two events  $e_1$  and  $e_2$  produced by the same process:
 
$$(e_1 \xrightarrow{ev} e_2) \Rightarrow (\mathcal{D}(e_1) < \mathcal{D}(e_2)).$$
  - For any message  $m$ :  $(\mathcal{D}(sy\_s(m)) = \mathcal{D}(sy\_del(m)))$ .
- **Termination.** Assuming that each process executes `synchr_deliver()` enough times, every message that was `synchr_sent` is `synchr_delivered`.

The validity property (neither creation nor corruption of messages), integrity property (no duplication), and termination property are the classical properties associated with message communication abstractions. The synchrony property states that there is time notion that increases inside each process taken individually, and, for each message  $m$ , associates a same integer date with its `synchr_send` and `synchr_deliver` events. This date, which is the “time point” associated with  $m$ , expresses the fact that  $m$  has been transmitted instantaneously when considering the time frame defined by the dating function  $\mathcal{D}()$ .

### 13.1.2 An Example of Use

Let us consider a simple application in which two processes  $p_1$  and  $p_2$  share two objects, each implemented by a server process. These objects are two FIFO queues  $Q_1$  and  $Q_2$ , implemented by the processes  $q_1$  and  $q_2$ , respectively.

The operations on a queue object  $Q$  are denoted  $Q.enqueue(x)$  (where  $x$  is the item to enqueue), and  $Q.dequeue()$ , which returns the item at the head of the queue or  $\perp$  if the queue is empty. The invocation of  $Q_j.enqueue(x)$  and  $Q_j.dequeue()$  by a process are translated as described in Table 13.1. (Due to the fact that communications are synchronous, the value returned by  $Q_j.dequeue()$  can be returned by the corresponding synchronous invocation.)

Let us consider the following sequences of invocations by  $p_1$  and  $p_2$ :

$p_1$ : ... `synchr_send(enqueue, a)` to  $q_1$ ;    `synchr_send(enqueue, b)` to  $q_2$ ;    ...  
 $p_2$ : ... `synchr_send(enqueue, c)` to  $q_2$ ;    `synchr_send(enqueue, d)` to  $q_1$ ;    ...

**Table 13.1** Operations as messages

Queue operation	Translated into
$Q_j.enqueue(x)$	<code>synchr_send(enqueue, x) to <math>q_j</math></code>
$Q_j.dequeue()$	<code>synchr_send(dequeue) to <math>q_j</math></code>

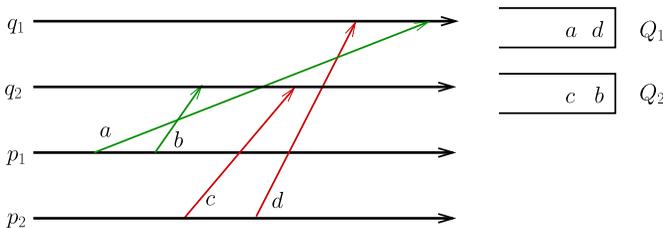
In a context where the communications are asynchronous (i.e., when `synchr_send()` and `synchr_deliver()` are replaced by `send()` and `receive()`, respectively), the message pattern depicted in Fig. 13.2 can occur, where the label associated with a message is the value it carries.

When all messages have been received,  $Q_1$  contains  $d$  followed by  $a$ , while  $Q_2$  contains  $b$  followed by  $c$ , as shown on the right part of the figure. But this is inconsistent with the order of the invocations in  $p_1$  and  $p_2$ . As  $Q_2$  contains  $b$  followed by  $c$ , it follows that the invocation of `send(enqueue, a)` by  $p_1$  occurred before the invocation of `send(enqueue, d)` by  $p_2$ . But, due to the asynchrony of the channels, the server  $q_1$  receives first  $d$  and then  $a$ .

Synchronous communication solves this inconsistency problem. Several scenarios can happen, but any scenario which happens guarantees that the queues are consistent with the order in which the processes have sent the messages. Figure 13.3 describes one of the three possible consistent scenarios. The three scenarios differ according to the speed of processes  $p_1$  and  $p_2$ . In the scenario of the figure,  $p_1$  and  $p_2$  first enqueue  $a$  in  $Q_1$  and  $c$  in  $Q_2$ , respectively. Then  $p_1$  enqueues  $b$  in  $Q_2$  while  $p_2$  enqueues  $d$  in  $Q_1$ . (In another possible scenario  $Q_1$  contains  $a$  followed by  $d$ , while  $Q_2$  contains  $b$  followed by  $c$ . This scenario is the case where  $p_1$  accesses each object before  $p_2$ . In the last possible scenario  $Q_1$  contains  $d$  followed by  $a$ , while  $Q_2$  contains  $c$  followed by  $d$ . This scenario is the case where  $p_2$  accesses each object before  $p_1$ .)

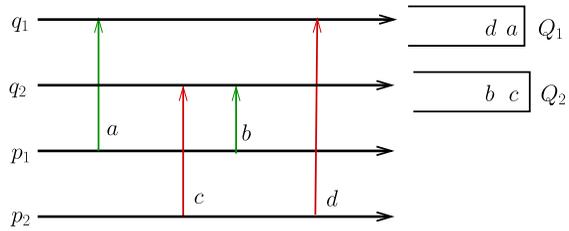
### 13.1.3 A Message Pattern-Based Characterization

**Synchronous Communication Is Strictly Stronger than Causal Order** To show that synchronous communication is stronger than causal order, let us consider



**Fig. 13.2** When communications are not synchronous

**Fig. 13.3** Accessing objects with synchronous communication



two messages  $m_1$  and  $m_2$ , sent to the same process, which are such that the event  $sy\_s(m_1)$  causally precedes the event  $sy\_s(m_2)$ . We have the following.

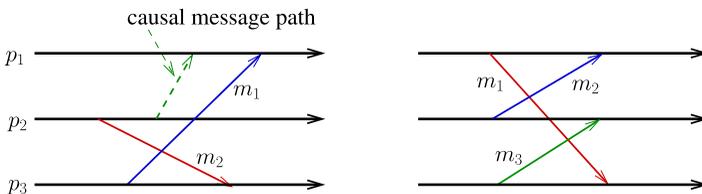
- $\mathcal{D}(sy\_s(m_1)) = \mathcal{D}(sy\_del(m_1))$  (from communication synchrony).
- $\mathcal{D}(sy\_s(m_2)) = \mathcal{D}(sy\_del(m_2))$  (from communication synchrony).
- $\mathcal{D}(sy\_s(m_1)) < \mathcal{D}(sy\_s(m_2))$  (due to  $sy\_s(m_1) \xrightarrow{ev} sy\_s(m_2)$ ).
- $\mathcal{D}(sy\_del(m_1)) < \mathcal{D}(sy\_del(m_2))$  follows from the previous items. Consequently, it follows from the fact that the function  $\mathcal{D}()$  is strictly increasing inside a process, and any two events of a process are ordered, that we necessarily have  $sy\_del(m_1) \xrightarrow{ev} sy\_del(m_2)$ .

To show that synchronous communication is strictly stronger than causal order, let us observe that the message pattern described in the left part of Fig. 13.1 satisfies the causal message delivery order, while it does not satisfy the synchronous communication property.

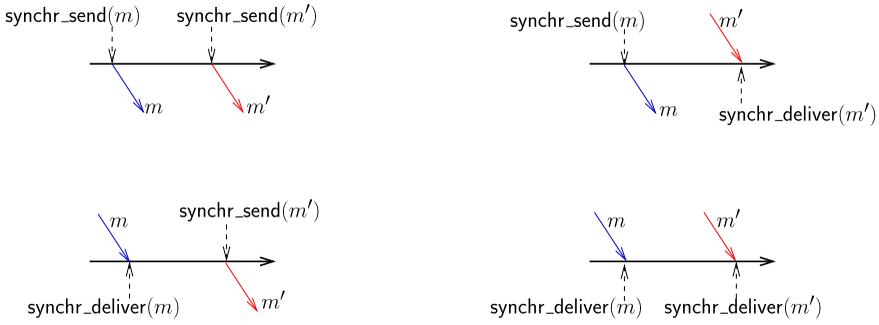
**The Crown Structure** As we are about to see, the notion of a crown allows for a simple characterization of synchronous communication. A *crown* (of size  $k \geq 2$ ) is a sequence of messages  $m_1, m_2, \dots, m_k$ , such that we have

$$\begin{aligned}
 &sy\_s(m_1) \xrightarrow{ev} sy\_del(m_2), \\
 &sy\_s(m_2) \xrightarrow{ev} sy\_del(m_3), \\
 &\dots, \\
 &sy\_s(m_{k-1}) \xrightarrow{ev} sy\_del(m_k), \\
 &sy\_s(m_k) \xrightarrow{ev} sy\_del(m_1).
 \end{aligned}$$

Examples of crowns are depicted in Fig. 13.4. On the left side, the event  $sy\_s(m_1)$  causally precedes the event  $sy\_del(m_2)$  and the event  $sy\_s(m_2)$  causally precedes



**Fig. 13.4** A crown of size  $k = 2$  (left) and a crown of size  $k = 3$  (right)



**Fig. 13.5** Four message patterns

the event  $sy\_del(m_1)$ , creating a crown of size  $k = 2$  made up of the messages  $m_1$  and  $m_2$ . The right side of the figure describes a crown of size  $k = 3$ . Hence, a crown is a specific message pattern whose interest lies in the theorem that follows.

**Theorem 18** *The communication pattern of a distributed execution satisfies the synchronous communication property if and only if there is no crown.*

*Proof* Let us first show that if the communication pattern satisfies the synchronous communication property, there is no crown. To that end, let us assume (by contradiction) that the communications are synchronous and there is a crown. Hence:

- There is a sequence of  $k \geq 2$  messages  $m_1, m_2, \dots, m_k$  such that  $sy\_s(m_1) \xrightarrow{ev} sy\_del(m_2), \dots, sy\_s(m_k) \xrightarrow{ev} sy\_del(m_1)$ . It follows that  $\forall x \in \{1, \dots, k-1\}$ , we have  $\mathcal{D}(sy\_s(m_x)) < \mathcal{D}(sy\_del(m_{x+1}))$ , and  $\mathcal{D}(sy\_s(m_k)) < \mathcal{D}(sy\_del(m_1))$ .
- As the communications are synchronous, we have  $\forall x \in \{1, \dots, k\}$ :  $\mathcal{D}(sy\_s(m_x)) = \mathcal{D}(sy\_del(m_x))$ .

Combining the two previous items, we obtain

$$\begin{aligned} \mathcal{D}(sy\_s(m_1)) &< \mathcal{D}(sy\_del(m_2)) = \mathcal{D}(sy\_s(m_2)), \\ \mathcal{D}(sy\_s(m_2)) &< \mathcal{D}(sy\_del(m_3)) = \mathcal{D}(sy\_s(m_3)), \quad \text{etc., until} \\ \mathcal{D}(sy\_del(m_k)) &= \mathcal{D}(sy\_s(m_k)) < \mathcal{D}(sy\_del(m_1)) = \mathcal{D}(sy\_s(m_1)), \end{aligned}$$

i.e.,  $\mathcal{D}(sy\_s(m_1)) < \mathcal{D}(sy\_s(m_1))$ , which is a contradiction.

To show that the communication pattern is synchronous if there is no crown let us consider the following directed graph  $G$ . Its vertices are the application messages exchanged by the processes and there is an edge from a message  $m$  to a message  $m'$  if one of the following patterns occurs (Fig. 13.5).

- $sy\_s(m) \xrightarrow{ev} sy\_s(m')$ , or
- $sy\_s(m) \xrightarrow{ev} sy\_del(m')$ , or
- $sy\_del(m) \xrightarrow{ev} sy\_s(m')$ , or
- $sy\_del(m) \xrightarrow{ev} sy\_del(m')$ .

It is easy to see that each case implies  $sy\_s(m) \xrightarrow{ev} sy\_del(m')$ , which means that a directed edge  $(m, m')$  belongs to  $G$  if and only if  $sy\_s(m) \xrightarrow{ev} sy\_del(m')$ . As there is no crown, it follows that  $G$  is acyclic.  $G$  can consequently be topologically sorted, and such a topological sort defines a dating function  $\mathcal{D}()$  trivially satisfying the properties defining synchronous communication.  $\square$

### 13.1.4 Types of Algorithms Implementing Synchronous Communications

An algorithm implementing synchronous communications has to prevent crowns from forming. This chapter considers three distinct approaches to implement such a prevention. The terms *interaction* or *rendezvous* are used as synonyms of synchronous communication.

- A first approach consists in assuming that the application programs are well written (there is no deadlock, etc.). To that end, a nondeterministic construct is offered to users. Such a construct allows a process to list several synchronous communication statements such that one of them will be selected at run time according to the synchronous communications invoked by the other processes. This approach, called *nondeterministic planned interaction* is addressed in Sect. 13.2.
- A second approach consists in considering that a process is always ready to `synchron_deliver` a message. Hence, the nondeterministic construct is implicit. This approach, called *nondeterministic forced interaction*, is addressed in Sect. 13.3.
- A third approach consists in adding a deadline to each invocation of a synchronous communication operation in such a way that the communication occurs before the deadline or not at all. Of course, this approach is meaningful only in synchronous systems, i.e., systems where the processes have a common notion of physical time and the message transfer duration is bounded. This approach, which is called *deadline-constrained interaction*, is addressed in Sect. 13.4.

## 13.2 Algorithms for Nondeterministic Planned Interactions

This section considers planned interactions. The term *planned* refers to the fact that each invocation of a communication operation mentions the identity of the other process. While this is always the case for a send operation, here each invocation of `synchron_del()` states the identity of the process from which a message has to be received.

### 13.2.1 Deterministic and Nondeterministic Communication Contexts

**Deterministic Context** The invocation of a synchronous communication operation by a process  $p_i$  can appear in a classical deterministic context such as

`statements_1; synchron_send(m) to p_x; statements_2,`

which means that the process  $p_i$  executes first the set of statements defined by `statements_1`, and then invokes the synchronous operation `synchr_send( $m$ )` to  $p_x$ . It remains blocked until  $p_x$  invokes the matching synchronous operation `synchr_deliver( $v$ )` from  $p_i$ . When this occurs the value of  $m$  is copied into the local variable  $v$  of  $p_x$ , and then both  $p_i$  and  $p_x$  continue their sequential execution. As far as  $p_i$  is concerned, it executes the set of statements defined by `statements_2`.

**Nondeterministic Construct** A process can also invoke communication operations in a nondeterministic context. A nondeterministic construct is used to that end. An example of such a construct is as follows:

```

begin nondeterministic context
    synchr_send( $m_1$ ) to  $p_x$    then statements_1
    or synchr_send( $m_2$ ) to  $p_y$    then statements_2
    or synchr_deliver( $m$ ) from  $p_z$  then statements_3
end nondeterministic context.

```

The meaning of this construct is the following. It states that the process  $p_i$  wants to execute one of the three synchronous operations. The choice is nondeterministic (it will actually depend on the speed of processes and implementation messages). Once one of these invocations has succeeded,  $p_i$  executes the sequence of statements associated with the corresponding invocation.

**Associated Properties** The fact that several synchronous communications may appear in a nondeterministic construct requires the statement of properties associated with this construct. These properties are the following.

- **Safety.** If a process enters a nondeterministic construct, it executes at most one of the synchronous communications listed in the construct. Moreover, if this synchronous communication is with process  $p_j$ , this process has invoked a matching synchronous operation.
- **Liveness.** If both  $p_i$  and  $p_j$  have invoked matching synchronous communication operations, and none of them succeeds in another synchronous communication, then the synchronous communication between  $p_i$  and  $p_j$  succeeds.

The safety property is a mutual exclusion property: A process can be engaged in at most one synchronous communication at a time. The liveness property states that, despite nondeterminism, if synchronous communications are possible, then one of them will occur (hence, nondeterminism cannot remain pending forever: it has to be solved).

### 13.2.2 An Asymmetric (Static) Client–Server Implementation

The algorithm presented in this section is due to A. Silberschatz (1979).

**Basic Idea: An Underlying Client–Server Hierarchy** Given a synchronous communication (rendezvous) between two processes  $p_i$  and  $p_j$ , the idea is to associate a client behavior with one of them and a server behavior with the other one. These associations are independent of the sense of the transfer. A client behavior means that the corresponding process has the initiative of the rendezvous, while a server behavior means that the corresponding process is waiting for the rendezvous.

The idea is to associate a client behavior with all the rendezvous invocations appearing in a deterministic context and a server behavior with all the invocations appearing in a nondeterministic context. Let us observe that, if needed, it is always possible to transform a deterministic context into a nondeterministic one, (while the opposite is not possible). As an example, the deterministic invocation “`synchron_send(m) to  $p_x$` ” can be replaced by

```

begin nondeterministic context
    synchron_send(m) to  $p_x$  then skip
end nondeterministic context,

```

to force it to be implemented with a server behavior. This can be done at compile time.

The limit of this implementation of a rendezvous lies consequently in the fact that it does not accept that both the matching invocations of a rendezvous appear in a nondeterministic context.

Let us define the implementation rendezvous graph of a distributed program as follows. This graph is a directed graph which captures the client–server relation imposed on the processes by their rendezvous invocations. Its vertices are the processes and there is an edge from a process  $p_i$  to a process  $p_j$  if (a) there a rendezvous involving  $p_i$  and  $p_j$ , and (b) the client behavior is associated with  $p_i$  while the server behavior is associated with  $p_j$ . To prevent deadlock and ensure liveness, the algorithm implementing rendezvous that follows requires that this graph be acyclic. (Let us observe that this graph can be computed at compile time.)

**Local Variables** Each process  $p_i$  manages the following local variables:

- $my\_client_i$  is a set containing the identities of the processes for which  $p_i$  behave as a server.
- $buffer_i$  is a variable in which  $p_i$  deposits the message it wants to send, or retrieves the message it is about to `synchron_deliver`.
- $may\_read_i[client_i]$  is an array of Booleans, initialized to  $[false, \dots, false]$ ;  $may\_read_i[j] = true$  (where  $j \in my\_client_i$ ) means that the “client”  $p_j$  allows the “server”  $p_i$  to read the content of its buffer  $buffer_j$ . This variable is used in the rendezvous where the client  $p_j$  is the sender and the server  $p_i$  the receiver.
- $may\_write_i[client_i]$  is an array of Booleans, initialized to  $[false, \dots, false]$ ;  $may\_write_i[j] = true$  means that the “client”  $p_j$  allows the “server”  $p_i$  to write into its local buffer  $buffer_j$ . This variable is used in the rendezvous where the client  $p_j$  is the receiver and the server  $p_i$  the sender.
- $end\_rdv_i$  is a Boolean (initialized to  $false$ ), which is set to the value  $true$  to signal the end of the rendezvous.

Process $p_j$ (server)	Process $p_i$ (client)
<b>operation</b> <code>synch_del(x)</code> from $p_i$ is	<b>operation</b> <code>synch_send(m)</code> to $p_j$ is
(S1) <code>wait(may_read_j[i]);</code>	(C1) <code>buffer_i ← m;</code>
(S2) <code>x ← obtain(i);</code>	(C2) <code>signal(j, may_read[i]);</code>
(S3) <code>may_read_j[i] ← false;</code>	(C3) <code>wait(end_rdv_i).</code>
(S4) <code>signal(i, end_rdv).</code>	(C4) <code>end_rdv_i ← false.</code>

**Fig. 13.6** Implementation of a rendezvous when the client is the sender

**Underlying Communication** The processes send and receive messages through a fully connected underlying asynchronous network. To have a modular presentation, the following intermediate communication operations are defined:

- **deposit( $i, a$ )** allows the invoking process  $p_j$  to deposit the content of  $a$  in  $buffer_i$ .  
This operation can easily be realized with a single message:  $p_j$  sends the message `STORE( $a$ )` to  $p_i$ , and, when it receives,  $p_i$  deposits  $a$  into  $buffer_i$ .
- **obtain( $i$ )** allows the invoking process  $p_j$  to obtain the content of  $buffer_i$ .  
This operation can be easily be realized with two messages. Process  $p_j$  sends a request message `REQ()` to  $p_i$ , which sends by return to  $p_j$  the message `ANSWER( $buffer_i$ )`.
- **signal( $i, x$ )** allows the invoking process  $p_j$  to set to `true` the Boolean local variable  $x_i$  of process  $p_i$ .  
This operation can be easily be realized with a single message:  $p_j$  sends the message `SIGNAL( $x$ )` to  $p_i$ , and when it receives it,  $p_i$  assigns the value `true` to its local Boolean identified by  $x$ .

As the messages `STORE()` and `SIGNAL()` sent by a process  $p_j$  to a process  $p_i$  have to be received in their sending order, the channels are required to be FIFO. If they are not, sequence numbers have to be associated with messages.

**Implementation When the Sender  $p_i$  Is the Client** The corresponding implementation for a pair of processes  $(p_i, p_j)$  is described in Fig. 13.6. As  $i \in my\_client_j$ , the invocations of “`synch_send( $m$ )` to  $p_j$ ” by  $p_i$  always occur in a deterministic context, while the matching invocations “`synch_del( $x$ )` from  $p_i$ ” issued by  $p_j$  always occur in a nondeterministic context.

When  $p_i$  invokes “`synch_send( $m$ )` to  $p_j$ ”,  $p_i$  deposits the message  $m$  in its local buffer (line C1), and sends a signal to  $p_j$  (line C2) indicating that  $p_j$  is allowed to read its local buffer. Finally,  $p_i$  awaits a signal indicating that the rendezvous is terminated (line C3).

When  $p_j$  invokes “`synch_del( $x$ )` from  $p_i$ ”,  $p_j$  waits until  $p_i$  allows it to read the content of its local buffer (line R1). Then, when  $p_j$  receives the corresponding signal, it reads the value saved in  $buffer_i$  (line S2). Finally it resets `may_read_j[i]` to its initial value (line S3), and sends a signal to  $p_i$  indicating the end of the rendezvous (line S4).

It is easy to see that, due to the signal/wait pattern used by the client  $p_i$  and the opposite wait/signal pattern used by the server  $p_j$ , no deadlock can occur and

Process $p_j$ (server)	Process $p_i$ (client)
<b>operation</b> <code>synch_send(m)</code> from $p_i$ <b>is</b>	<b>operation</b> <code>synch_del(x)</code> to $p_j$ <b>is</b>
(S1) <code>wait(may_write_j[i]);</code>	(C1) <code>signal(j, may_write[i]);</code>
(S2) <code>deposit(i, m);</code>	(C2) <code>wait(end_rdv_i);</code>
(S3) <code>may_write_j[i] ← false;</code>	(C3) <code>x ← buffer_i;</code>
(S4) <code>signal(i, end_rdv).</code>	(C4) <code>end_rdv_i ← false.</code>

**Fig. 13.7** Implementation of a rendezvous when the client is the receiver

there is a physical time at which both processes are executing their synchronous communication operations.

**Implementation When the Sender  $p_i$  Is the Server** This implementation, described in Fig. 13.7, is similar to the previous one. The message control pattern is exactly the same as the previous one. The only thing that differs is the sense of the transfer at the synchronous communication level.

**Solving Nondeterminism** To complete the description of the implementation, the way the nondeterministic construct is implemented has to be described. To that end, let us consider the nondeterministic construct described in Sect. 13.2.1, which involves the following three invocations: “`synchr_send( $m_1$ ) to  $p_x$` ”, “`synchr_send( $m_2$ ) to  $p_y$` ”, and “`synchr_deliver( $x$ ) from  $p_z$` ”.

Let  $p_j$  be the corresponding invoking process. This process is a server for  $p_x$ ,  $p_y$ , and  $p_z$ , and we have consequently  $x, y, z \in my\_client_j$ . Moreover, each of the matching invocations issued by these three processes is in deterministic context. When,  $p_j$  enters this nondeterministic construct, it executes the following sequence of statements:

```

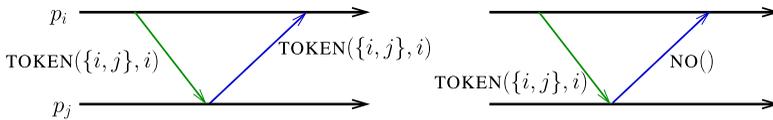
wait(may_write_j[x] ∨ may_write_j[y] ∨ may_read_j[z]);
among these Booleans, select one equal to true;
let  $i \in \{x, y, z\}$  be the corresponding process and
let may_xxx_j[i] the corresponding Boolean;
if (may_xxx_j[i] is may_read_j[i])
    then execute the lines S2, S3, and S4 of Fig. 13.6
    else execute the lines S2, S3, and S4 of Fig. 13.7
end if.

```

Thanks to the modular decomposition of the implementation, the previous code solving nondeterministic choices is particularly simple.

### 13.2.3 An Asymmetric Token-Based Implementation

The implementation of synchronous communication described in this section allows any combination of deterministic and nondeterministic matching invocations of the



**Fig. 13.8** A token-based mechanism to implement an interaction

operations `synchr_send()` and `synchr_del()`. More precisely, any two matching invocations are allowed to appear both in a deterministic context, or in a nondeterministic context, or (as in the previous section) one in a deterministic context and the other one in a nondeterministic context. Hence, this section focuses only on the control needed to realize an interaction and not on the fact that a single message, or a message in each direction, is transmitted during the rendezvous. It is even possible that no message at all is transmitted. In this case, the interaction boils down to a pure rendezvous synchronization mechanism. The corresponding algorithm is due to R. Bagrodia (1989).

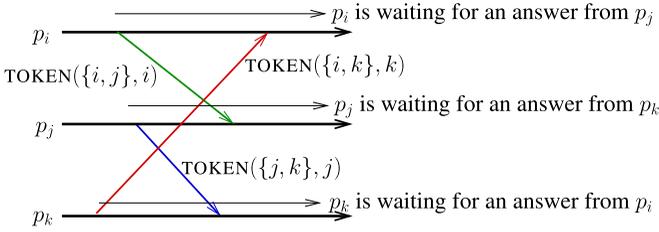
This section uses the term *interaction*, which has to be considered as a synonym of *rendezvous* or *synchronous communication*.

**Underlying Idea: Associate a Token with Each Interaction** A token is associated with each possible interaction. Considering any pair of processes  $p_i$  and  $p_j$ , the token  $\text{TOKEN}(\{i, j\})$  is introduced to allow them to realize a rendezvous.

The process that currently has the token (e.g.,  $p_i$ ) has the initiative to ask the other process (e.g.,  $p_j$ ) if  $p_j$  agrees to perform an interaction with it. To that end,  $p_i$  sends the token to  $p_j$ . When  $p_j$  receives the token, several cases are possible.

- If  $p_j$  agrees with this interaction request, it sends the token back to  $p_i$  and the interaction succeeds. This scenario is depicted on the left side of Fig. 13.8 (in which the second parameter in the token is the identity of the process that initiated the interaction).
- If  $p_j$  is not interested in an interaction with  $p_i$ , it sends back to  $p_i$  a message  $\text{NO}()$  and keeps the token. This means that the future request for an interaction involving  $p_i$  and  $p_j$  will be initiated by  $p_j$ . This scenario is depicted on the right side of Fig. 13.8.
- If  $p_j$  is interested in an interaction with  $p_i$ , but is waiting for an answer from a process  $p_k$  (to which it previously sent  $\text{TOKEN}(\{j, k\})$ ), it will send an answer to  $p_i$  according to the answer it receives from  $p_k$ . The implementation has to ensure that there is neither a deadlock (with a process waiting forever for an answer), nor a livelock (with processes always sending  $\text{NO}()$  while an interaction is possible).

**Preventing Deadlock and Livelock** Let us consider the scenario described in Fig. 13.9. Process  $p_i$ , which has the token for initiating interactions with  $p_j$ , sends it this token, and starts waiting for an answer. The same occurs for  $p_j$  that sent  $\text{TOKEN}(\{j, k\}, j)$  to  $p_k$ , and  $p_k$  that sent  $\text{TOKEN}(\{i, k\}, k)$  to  $p_i$ . Typically, if none of them answers, there is a deadlock, and if all of them send the answer  $\text{NO}()$ , the liveness property is compromised as one interaction is possible.



**Fig. 13.9** Deadlock and livelock prevention in interaction implementation

Solving this issue requires us to break the symmetry among the three messages  $\text{TOKEN}()$ . As seen in Sect. 11.2.2 when discussing resource allocation, one way to do that consists in defining a total order on the request messages  $\text{TOKEN}()$ . Such a total order can be obtained from the identity of the processes. As an example, assuming  $j < i < k$ , let us consider the process  $p_i$  when it receives the message  $\text{TOKEN}(\{i, k\}, k)$ . Its behavior is governed by the following rule:

```

if ( $i < k$ ) then delay the answer to  $p_k$ 
                else send  $\text{NO}()$  to  $p_k$ 
end if,
    
```

i.e.,  $p_i$  has priority (in the sense that it does not answer by return) if its identity is smaller than that of the process that sent the request.

When considering the execution of Fig. 13.9,  $p_i$  delays its answer to  $p_k$ ,  $p_j$  delays its answer to  $p_i$ , but  $p_k$  sends the answer  $\text{NO}()$  to  $p_j$ . When  $p_j$  receives this message, it can tell  $p_i$  that it accepts its interaction, and finally when  $p_i$  learns it,  $p_i$  sends  $\text{NO}()$  to  $p_k$ . (As seen in Sect. 11.2.2 this identity-based “priority” rule prevents deadlocks from occurring and ensures interaction liveness.) Considering examples with more than three processes would show that this priority scheme allows independent interactions to be executed concurrently.

When considering Fig. 13.9, the crown that appears involves implementation messages. The priority rule prevents the application level from inheriting this implementation level crown.

**Local Variable at a Process** To implement the previous principles, each process manages the following local variables:

- $state_i$  describes the local interaction state of  $p_i$ . Its value domain is  $\{out, interested, engaged\}$ .  $state_i = out$  means that  $p_i$  is not currently interested in a rendezvous;  $state_i = engaged$  means that  $p_i$  is interested and waiting for an answer;  $state_i = interested$  means that, while  $p_i$  is interested in an interaction, it is not currently waiting for an answer.
- $interaction_i$  is a set containing the identities of the processes currently proposed by  $p_i$  to have an interaction. When it invokes an interaction with a process  $p_j$  in a deterministic context,  $p_i$  sets  $interaction_i$  to  $\{j\}$ . When it enters a nondeterministic construct containing invocations of rendezvous with  $p_j, p_k, p_\ell$ , etc., it sets  $interaction_i$  to  $\{j, k, \ell, \dots\}$ .

```

when  $p_i$  enters a (deterministic or nondeterministic) rendezvous context do
(1)  $interactions_i \leftarrow \{\text{ids of the candidate proc. defined in the rendezvous context}\};$ 
(2)  $state_i \leftarrow interested;$ 
(3) if  $(\exists j \in interactions_i \text{ such that } \text{TOKEN}(\{i, j\}) \in tokens_i)$ 
(4)   then withdraw  $\text{TOKEN}(\{i, j\})$  from  $tokens_i;$ 
(5)     send  $\text{TOKEN}(\{i, j, i\})$  to  $p_j;$ 
(6)      $state_i \leftarrow engaged;$ 
(7)      $delayed_i \leftarrow \perp$ 
(8)   end if.

when  $\text{TOKEN}(\{i, j, x\})$  is received from  $p_j$  do %  $x \in \{i, j\}$  %
(9) if  $(state_i = out) \vee (j \notin interaction_i)$ 
(10)  then send  $\text{NO}()$  to  $p_j;$  add  $\text{TOKEN}(\{i, j\})$  to  $token_i$ 
(11)  else if  $(state_i = interested)$ 
(12)    then send  $\text{TOKEN}(\{i, j, x\})$  to  $p_j;$   $state_i \leftarrow out$ 
(13)    else %  $state_i = engaged$  %
(14)      case  $(x > i) \wedge (delayed_i = \perp)$ 
(15)        then  $delayed_i \leftarrow \text{TOKEN}(\{i, j, x\})$ 
(16)         $(x < i) \vee [(x > i) \wedge (delayed_i \neq \perp)]$ 
(17)        then send  $\text{NO}()$  to  $p_j;$  add  $\text{TOKEN}(\{i, j\})$  to  $token_i$ 
(18)         $(x = i)$  then add  $\text{TOKEN}(\{i, j\})$  to  $token_i;$ 
(19)          if  $(delayed_i \neq \perp)$ 
(20)            then let  $delayed_i = \text{TOKEN}(\{i, k, k\});$ 
(21)              send  $\text{NO}()$  to  $p_k;$ 
(22)              add  $\text{TOKEN}(\{i, k\})$  to  $token_i$ 
(23)            end if
(24)             $state_i \leftarrow out$ 
(25)          end case
(26)        end if
(27) end if.

when  $\text{NO}()$  is received from  $p_j$  do %  $state_i = engaged$  %
(28) if  $(delayed_i \neq \perp)$  then let  $delayed_i = \text{TOKEN}(\{i, k, k\});$ 
(29)   send  $\text{TOKEN}(\{i, k, k\})$  to  $p_k;$   $state_i \leftarrow out$ 
(30)   else same as lines 2–8
(31) end if.

```

**Fig. 13.10** A general token-based implementation for planned interactions (rendezvous)

- $tokens_i$  is a set containing the tokens currently owned by  $p_i$  (let us recall that the token  $\text{TOKEN}(\{i, j\})$ , allows its current owner— $p_i$  or  $p_j$ —to send a request for an interaction with the other process).

Initially, each interaction token is placed at one of the processes associated with it.

- $delayed_i$  is a variable which contains the token for which  $p_i$  has delayed sending an answer;  $delayed_i = \perp$  if no answer is delayed.

**Behavior of a Process** The algorithm executed by a process  $p_i$  is described in Fig. 13.10. When it enters a deterministic or nondeterministic communication context (as defined in Sect. 13.2.1), a process  $p_i$  becomes *interested* (line 2), and initializes its set  $interactions_i$  to the identities of the processes which it has defined

as candidates for a synchronous communication (line 1). Then, if it has the token for one of these interactions (line 3), it selects one of them (line 4), and sends the associated token to the corresponding process  $p_j$  (line 5). It becomes then *engaged* (line 5), and sets  $delayed_i$  to  $\perp$  (line 7). If the predicate of line 3 is false,  $p_i$  remains in the local state *interested*.

The core of the algorithm is the reception of a message  $TOKEN(\{i, j\}, x)$  that a process  $p_i$  receives from a process  $p_j$ . As  $x$  is the identity of the process that initiated the interaction, we necessarily have  $x = i$  or  $x = j$ . When it receives such a message, the behavior of  $p_i$  depends on its state.

- If it is not interested in interactions or, while interested,  $j \notin interactions_i$ ,  $p_i$  sends by return the message  $NO()$  to  $p_j$  and saves  $TOKEN(\{i, j\})$  in  $tokens_i$  (lines 9–10). In that way,  $p_i$  will have the initiative for the next interaction involving  $p_i$  and  $p_j$ , and during that time  $p_j$  will no longer try to establish a rendezvous with it.
- If  $j \in interactions_i$ , and  $p_i$  has no pending request (i.e.,  $state_i = interested$ ), it commits the interaction with  $p_j$  by sending back the message  $TOKEN(\{i, j\}, j)$  (lines 11–12).
- Otherwise, we have  $j \in interactions_i$  and  $state_i = engaged$ . Hence,  $p_i$  has sent a message  $TOKEN(\{i, k\}, i)$  for an interaction with a process  $p_k$  and has not yet received an answer. According to the previous discussion on deadlock prevention, there are three cases.
  - If  $x > i$  and  $p_i$  has not yet delayed an answer, it delays the answer to  $p_j$  (lines 14–15).
  - If  $x < i$ , or  $x > i$  and  $p_i$  has already delayed an answer, it sends the answer  $NO()$  to  $p_j$ . In that way  $p_j$  will be able to try other interactions (lines 16–17).
 

Let us remark that, if  $delayed_i \neq \perp$ ,  $p_i$  will commit an interaction: either the interaction with the process  $p_k$  from which  $p_i$  is waiting for an answer to the message  $TOKEN(\{i, k\}, i)$  that  $p_i$  previously sent to  $p_j$ , or with the process  $p_\ell$  waiting for its answer ( $p_\ell$  is such that  $delayed_i = TOKEN(\{\ell, i\}, \ell)$ ).
  - If  $x = i$ , by returning the message  $TOKEN(\{i, j\}, i)$  to  $p_i$ , its sender  $p_j$  commits the interaction. In that case, before moving to state *out* (line 24),  $p_i$  stores  $TOKEN(\{i, j\})$  in  $tokens_i$  (hence,  $p_i$  will the initiative for the next interaction with  $p_j$ ), and sends the answer  $NO()$  to the delayed process  $p_k$ , if any (lines 18–24).

Finally, when  $p_i$  receives the answer  $NO()$  from a process  $p_j$  (which means that it previously became engaged by sending the request  $TOKEN(\{i, j\}, i)$  to  $p_j$ ), the behavior of  $p_i$  depends on the value of  $delayed_i$ . If  $delayed_i \neq \perp$ ,  $p_i$  has delayed its answer to the process  $p_k$  such that  $delayed_i = TOKEN(\{k, i\}, k)$ . In that case, it commits the interaction with  $p_k$  (lines 28–29). Otherwise, it moves to the local state *interested* and tries to establish an interaction with another process in  $interactions_i$  for which it has the token (lines 2–8).

**Properties** Due to the round-trip of a message  $TOKEN(\{i, j\}, i)$ , it is easy to see that a process can simultaneously participate in at most one interaction at a time.

Each request (sending of a message  $\text{TOKEN}(\{i, j\}, i)$ ) gives rise to exactly one answer (the same message echoed by the receiver or a message  $\text{NO}()$ ). Moreover, due to the total order on process identities, no process can indefinitely delay another one. Let us also notice that, if a process  $p_i$  is such that  $k \in \text{interactions}_i$  and  $\text{TOKEN}(\{i, k\}) \in \text{tokens}_i$ ,  $p_i$  will send the request  $\text{TOKEN}(\{i, k\}, k)$  to  $p_k$  (if it does not commit another interaction before).

Let us consider the following directed graph. Its vertices are the processes, and there is an edge from  $p_i$  to  $p_j$  if  $\text{delayed}_j = \text{TOKEN}(\{j, i\}, i)$  (i.e.,  $p_j$  is delaying the sending of an answer to  $p_i$ ). This graph, whose structure evolves dynamically, is always acyclic. This follows from the fact that an edge can go from a process  $p_i$  to a process  $p_j$  only if  $i > j$ . As the processes that are sink nodes of the graph eventually send an answer to the processes they delay, it follows (by induction) that no edge can last forever.

The liveness property follows from the previous observations, namely, if (a) two processes  $p_i$  and  $p_j$  are such that  $i \in \text{interactions}_j$  and  $j \in \text{interactions}_i$ , and (b) none of them commits another interaction, then  $p_i$  and  $p_j$  will commit their common interaction.

## 13.3 An Algorithm for Nondeterministic Forced Interactions

### 13.3.1 Nondeterministic Forced Interactions

To prevent crown patterns from forming (at the level of application messages), processes have sometimes to deliver a message instead of sending one. In the context of the previous section, this is planned at the programming level and, to that end, processes are allowed to use explicitly a nondeterministic communication construct. As we have seen, this construct allows the communication operation which will prevent crown formation to be selected at run time.

This section considers a different approach in which there is no nondeterministic choice planned by the programmer, and all the invocations of `synchron_send()` appear in a deterministic context. To prevent crowns from forming, a process can be forced at any time to deliver a message or to delay the sending of a message, hence the name *forced interaction*.

### 13.3.2 A Simple Algorithm

**Principle** As no algorithm implementing a rendezvous can be completely symmetric, a way to solve conflicts is to rely on the identities of the processes (as done in Sect. 13.2.3). An interaction (rendezvous) involving  $p_i$  and  $p_j$  is conceptually controlled by the process  $p_{\max(i,j)}$ . If the process  $p_{\min(i,j)}$  wants to `synchron_send` a message to  $p_{\max(i,j)}$ , it has to ask  $p_{\max(i,j)}$  to manage the interaction. This algorithm is due to V.V. Murty and V.K. Garg (1997).

```

operation synchr_send( $m$ ) to  $p_j$  is
(1) if ( $i > j$ ) then wait ( $\neg engaged_i$ );
(2)           send MSG( $m$ ) to  $p_j$ ;  $engaged_i \leftarrow true$ 
(3)           else  $buffer_i[j] \leftarrow m$ ; send REQUEST() to  $p_j$ 
(4) end if.

when MSG( $m$ ) is received from  $p_j$  do
(5) if ( $i > j$ ) then synchr_delivery of  $m$ ;  $engaged_i \leftarrow false$ 
(6)           else wait ( $\neg engaged_i$ );
(7)           synchr_delivery of  $m$ ; send ACK() to  $p_j$ 
(8) end if.

when ACK() is received from  $p_j$  do
(9)  $engaged_i \leftarrow false.$ 

when REQUEST() is received from  $p_j$  do %  $j < i$  %
(10) wait ( $\neg engaged_i$ );
(11) send PROCEED() to  $p_j$ ;  $engaged_i \leftarrow true.$ 

when PROCEED() is received from  $p_j$  do %  $j > i$  %
(12) wait ( $\neg engaged_i$ );
(13) send MSG( $buffer_i[j]$ ) to  $p_j.$ 

```

**Fig. 13.11** An algorithm for forced interactions (rendezvous)

**The Algorithm** Each process  $p_i$  manages a local Boolean variable  $engaged_i$ , initially equal to *false*. This variable is set to *true*, when  $p_i$  is managing a synchronous communication with a process  $p_j$  such that  $i > j$ . The aim of the variables  $engaged_i$  is to prevent the processes from being involved in a cycle that would prevent liveness.

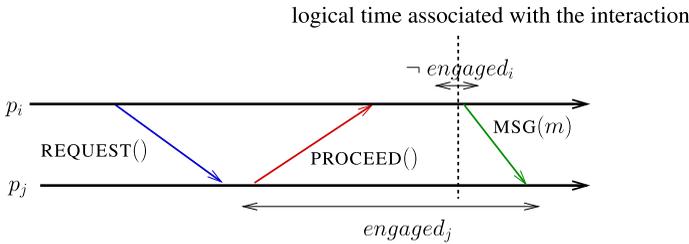
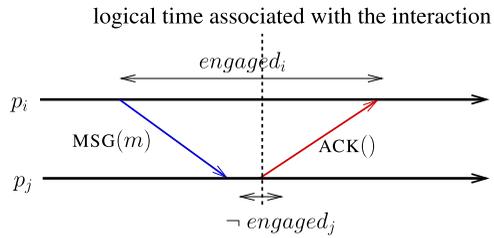
To present the algorithm we consider two cases according to the values of  $i$  and  $j$ , where  $p_i$  is the sender and  $p_j$  the receiver.

- $i > j$ . In that case the sender has the initiative of the interaction. It can send the message  $m$  only if it is not currently engaged with another process. It sends then MSG( $m$ ) to  $p_j$  and becomes engaged (lines 1–2).

When  $p_j$  receives MSG( $m$ ) from  $p_i$ , it waits until it is no longer engaged in another rendezvous (line 6). Then it synchr\_delivers the message  $m$ , and sends the message ACK() to  $p_i$  (line 7). When  $p_i$  receives this message, it learns that the synchronous communication has terminated and consequently resets  $engaged_i$  to *false* (line 9). The corresponding pattern of messages exchanged at the implementation level is described in Fig. 13.12.

- $i < j$ . In this case,  $p_i$  has to ask  $p_j$  to manage the interaction. To that end, it sends the message REQUEST() to  $p_j$  (line 3). Let us notice that  $p_i$  is not yet engaged in a synchronous communication. When  $p_j$  receives this message, it waits until it is no longer engaged in another interaction (line 10). When this occurs,  $p_j$  sends to  $p_i$  a message PROCEED() and becomes engaged with  $p_i$  (line 11). When it receives this message,  $p_i$  sends the application message (which has been previously saved in  $buffer_i[j]$ ) and terminates locally the interaction (line 12). Fi-

**Fig. 13.12**  
 Forced interaction:  
 message pattern when  $i > j$



**Fig. 13.13** Forced interaction: message pattern when  $i < j$

nally, the reception of this message  $MSG(m)$  by  $p_j$  entails the `synchr_delivery` of  $m$  and the end of the interaction on  $p_j$ 's side (line 5). The corresponding pattern of messages exchanged at the implementation level is described in Fig. 13.13. As in the previous figure, this figure indicates the logical time that can be associated with the interaction (from an application level point of view).

### 13.3.3 Proof of the Algorithm

This section shows that, when controlled by the previous algorithm, the pattern of application messages satisfies the synchrony property, and all application messages are `synchr_delivered`.

**Lemma 11** *The communication pattern of application messages generated by the algorithm satisfies the synchrony property.*

*Proof* To show that the algorithm ensures that the application messages `synchr_sent` and `synchr_delivered` satisfy the synchrony property, we show that there is no crown at the application level. It follows then from Theorem 18 that the synchrony property is satisfied.

The proof is made up of two parts. We first show that, if the computation has a crown of size  $k > 2$ , it has also a crown of size 2 (Part I). Then, we show that there is no crown of size 2 (Part II). As in Sect. 13.1, let  $sy_s(m)$  and  $sy_{del}(m)$  be the events associated with the synchronous send and the synchronous delivery of the application message  $m$ , respectively.

Proof of Part I. Let  $m_1, m_2, \dots, m_k$  be the sequence of messages involved in a crown of size  $k > 2$ , and  $m_x$  be a message of this crown `synchron_sent` by a process  $p_i$  to a process  $p_j$ . We consider two cases.

- $i > j$ . Due to the crown, we have  $sy\_s(m_x) \xrightarrow{ev} sy\_del(m_{x+1})$ . As, after it has sent the implementation message `MSG(mx)` to  $p_j$  (line 2),  $p_i$  remains engaged with  $p_j$  until it receives an implementation message `ACK()` from it (line 9), it follows that we have  $sy\_del(m_x) \xrightarrow{ev} sy\_del(m_{x+1})$ . As (due to the crown) we also have  $sy\_s(m_{x-1}) \xrightarrow{ev} sy\_del(m_x)$ , it follows that  $sy\_s(m_{x-1}) \xrightarrow{ev} sy\_del(m_{x+1})$ , and there consequently a crown of size  $k - 1$ .
- $i < j$ . The reasoning is similar to the previous case. Due to the crown, we have  $sy\_s(m_{x-1}) \xrightarrow{ev} sy\_del(m_x)$ . As, after it has sent the message `PROCEED()` to  $p_i$  (line 11),  $p_j$  remains engaged with  $p_i$  until it receives `MSG(mx)` from it (line 5), it follows that we have  $sy\_s(m_{x-1}) \xrightarrow{ev} sy\_s(m_x)$ . Combining this with  $sy\_s(m_x) \xrightarrow{ev} sy\_del(m_{x+1})$ , we obtain  $sy\_s(m_{x-1}) \xrightarrow{ev} sy\_del(m_{x+1})$ , and obtain a crown whose size is  $k - 1$ .

Proof of Part II. Let assume by contradiction that there is crown of size 2. Hence, there are two messages  $m_1$  and  $m_2$  such that  $sy\_s(m_1) \xrightarrow{ev} sy\_del(m_2)$  and  $sy\_s(m_2) \xrightarrow{ev} sy\_del(m_1)$ . (Let us recall that these relations can involve causal paths as shown in the crown of size 2 depicted at the left side of Fig. 13.4.) Let  $p_i$  be the process that `synchron_sent`  $m_1$  to  $p_j$ , and  $p_{i'}$  be the process that `synchron_sent`  $m_2$  to  $p_{j'}$ . There are two cases.

- $m_1$  is such that  $i > j$  or  $m_2$  is such that  $i' > j'$ . Without loss of generality we consider  $i > j$ . It follows from the previous reasoning (first item of Part I) that  $sy\_del(m_1) \xrightarrow{ev} sy\_del(m_2)$ .
  - If  $i' > j'$ , we obtain with the same reasoning  $sy\_del(m_2) \xrightarrow{ev} sy\_del(m_1)$ ,
  - If  $i' < j'$ , we obtain with a similar reasoning  $\neg(sy\_del(m_1) \xrightarrow{ev} sy\_del(m_1))$ .

Both cases contradict  $sy\_del(m_1) \xrightarrow{ev} sy\_del(m_2)$ .

- $m_1$  and  $m_2$  are such that  $i < j$  and  $i' < j'$ . By reasoning similar to that used in the second item of Part I, we obtain  $sy\_s(m_1) \xrightarrow{ev} sy\_s(m_2)$  and  $sy\_s(m_2) \xrightarrow{ev} sy\_s(m_1)$ , a contradiction which concludes the proof of the lemma.  $\square$

**Lemma 12** *Each invocation of `synchron_send()` terminates and the corresponding message is `synchron_delivered`.*

*Proof* To prove this lemma we show that if a process  $p_i$  invokes `synchron_send()`, it proceeds eventually to the state  $\neg engaged_i$ .

Let us observe that process  $p_1$  is never engaged (due to its smallest identity, it never executes line 2 or line 11, and consequently we always have  $\neg engaged_1$ ). It follows that  $p_1$  never receives a message `REQUEST()` or `ACK()`. Moreover, when it receives a message `MSG(m)` or `PROCEED()`, it always sends back an answer (lines 6–7 and lines 12–13).

The rest of the proof is by induction. Let us assume that each process  $p_1, p_2, \dots, p_i, \dots, p_k$ , eventually moves eventually to the state  $\neg engaged_i$  and answers the messages it receives. Let us observe that  $p_{k+1}$  can become engaged only when it sends a message `MSG()` to a process with a smaller identity (line 2), or when it sends a message `PROCEED()` to a process with a smaller identity (line 11). As these processes will answer the message from  $p_{k+1}$  (induction assumption), it follows that  $p_{k+1}$  is eventually such that  $\neg engaged_{k+1}$ , and will consequently answer the message it receives. Hence, no process will remain blocked forever, and all the invocations of `synchr_send()` terminate.

The fact that the corresponding messages are `synchr_delivered` follows trivially from line 5 and line 7.  $\square$

**Theorem 19** *The algorithm of Fig. 13.11 ensures that the message communication pattern satisfies the synchrony property, no process blocks forever, and each message that is `synchr_sent` is `synchr_delivered`.*

*Proof* The proof follows from Lemmas 11 and 12.  $\square$

## 13.4 Rendezvous with Deadlines in Synchronous Systems

This section introduces rendezvous with deadlines. As noticed in Sect. 13.1.4, this is possible only in synchronous distributed systems. This is due to the fact that, in asynchronous distributed systems, there is no notion of physical time accessible to the processes. This section presents first definitions, and then algorithms implementing rendezvous with deadline. These algorithms are due to I. Lee and S.B. Davidson (1987).

### 13.4.1 Synchronous Systems and Rendezvous with Deadline

**Synchronous Systems** As processing times are negligible when compared to message transfer durations, they are considered as having a zero duration. Moreover, there is an upper bound, denoted  $\delta$ , on the transit time of all messages that are sent through the underlying network, where the transit time of a message is measured as the duration that elapses between its sending and its reception for processing by the destination process. (The time spent by a message in input/output buffers belongs to its transit time.) This bound is such that it is an upper bound, whatever the process that measures it.

Each process has a local physical clock denoted  $ph\_clock_i$  that it can read to obtain the current date. These clocks are synchronized in such a way that, at any time, the difference between two clocks is upper bounded by  $\theta$ .

The previous assumptions on  $\delta$  and  $\theta$  means that if, at time  $\tau$  measured on  $p$ 's local clock, process  $p$  sends a message  $m$  to a process  $q$ ,  $m$  will be received by  $q$  by time  $\tau + \delta + \theta$  measured on  $q$ 's local clock.

If there is a common physical clock that can be read by all processes, the local clocks can be replaced by this common clock, and then  $\theta = 0$ .

**Rendezvous with Deadline** The *rendezvous with deadline* abstraction provides the processes with two operations denoted `timed_send()` and `timed_receive()`. Both the sender  $p$ , when it invokes `timed_send( $m$ ,  $deadline_1$ )`, and the receiver  $q$ , when it invokes `timed_receive( $x$ ,  $deadline_2$ )`, specify a deadline for the rendezvous ( $x$  is the local variable of  $q$  where the received message is deposited if the rendezvous succeeds).

If the rendezvous cannot happen before their deadlines, both obtain the control value `timeout`. If the rendezvous succeeds before their deadlines, both obtain the value `commit`. It is important to notice that both processes receive the same result: The rendezvous is successful for both (and then  $m$  has been transmitted), or is unsuccessful for both (and then  $m$  has not been transmitted).

**Temporal Scope Construct** The following language construct, called *temporal scope*, is used to ease the presentation of the algorithms. Its simplest form is the following:

```
within deadline do
    when a message is received do statements
at deadline occurrence return(timeout)
end within.
```

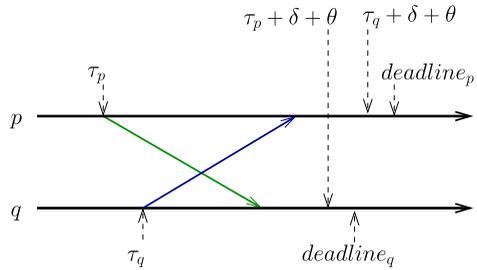
Its meaning is the following: Let  $\tau$  be the time (measured by  $ph\_clock_i$ ) at which a process  $p_i$  enters a **within ... end within** construct. The duration spent in this construct is bounded by  $\tau + deadline$ . During this period of time, if  $p_i$  receives a message, it executes the associated code denoted “statements”, which can contain invocations of `return(commit)` or `return(timeout)`. If it executes such a `return()`,  $p_i$  exits the construct with the corresponding result. If `return()` has not been invoked by time  $\tau + deadline$ ,  $p_i$  exits the construct with the result `timeout`.

### 13.4.2 Rendezvous with Deadline Between Two Processes

Let us consider the base case of two processes  $p$  and  $q$  such that  $p$  invokes `timed_send( $m$ ,  $deadline_p$ )` and  $q$  invokes `timed_receive( $x$ ,  $deadline_q$ )`. Moreover, let  $\tau_p$  be the time, measured at its local clock, at which  $p$  starts its invocation of `timed_send( $m$ ,  $deadline_p$ )`. Similarly, let  $\tau_q$  be the time, measured at its local clock, at which  $q$  starts its invocation of `timed_receive( $x$ ,  $deadline_q$ )`.

**A Predicate to Commit or Abort a Rendezvous** Let us observe that if  $p$  sends a message to  $q$  at a time  $\tau_p$  (measured on its clock) such that  $\tau_p + \delta + \theta \leq deadline_q$ , then  $q$  will receive this message before its deadline. Similarly, if  $q$  sends a message

**Fig. 13.14** When the rendezvous must be successful (two-process symmetric algorithm)



to  $p$  at a time  $\tau_q$  (measured on its clock) such that  $\tau_q + \delta + \theta \leq \text{deadline}_p$ , then  $p$  will receive this message before its deadline. Hence, when true, the predicate

$$(\tau_p + \delta + \theta \leq \text{deadline}_q) \wedge (\tau_q + \delta + \theta \leq \text{deadline}_p)$$

states that, whatever the actual speed on the messages, the rendezvous can be successful (see Fig. 13.14). If the predicate is false, the rendezvous can succeed or fail, which depends on the speed of the messages and on the deadline values. If  $\tau_p + \delta + \theta > \text{deadline}_q$ , the message sent by  $p$  to  $q$  may arrive before  $\text{deadline}_q$  but it may also arrive later. There is no way to know this from the values of  $\tau_p$  and  $\text{deadline}_q$ . (And similarly, in the other direction.) Hence, the previous predicate is the weakest predicate (based only on  $\tau_p$ ,  $\tau_q$ ,  $\text{deadline}_p$ , and  $\text{deadline}_q$ ) which (when true) ensures safely that it is impossible for the rendezvous not to occur.

In order that both  $p$  and  $q$  compute this predicate, the message sent by  $p$  to  $q$  has to contain  $\tau_p$  and  $\text{deadline}_p$ , and the message sent by  $q$  to  $p$  has to contain  $\tau_q$  and  $\text{deadline}_q$ . This is the principle on which the rendezvous with deadlines between two processes is based.

**The Rendezvous Algorithm** The corresponding algorithms implementing `timed_send()` and `timed_receive()` are described in Fig. 13.15. Their code is a simple translation of the previous discussion in terms of the **within** temporal scope construct.

In addition to the application message  $m$ , the sender  $p$  sends to  $q$  the date of its starting time  $\tau_p$  and its deadline  $\text{deadline}_p$  (line 2). Process  $p$  then enters the temporal scope (line 3), and waits for a message from  $q$ . If no message is received by  $p$ 's deadline, the rendezvous failed and  $p$  returns `timeout` (line 9). If  $p$  receives a message from  $q$  (line 4) before its deadline, it computes the value of the success predicate (line 5), and returns the value `commit` if the rendezvous cannot “not occur”, (line 6), or `timeout` otherwise.

The code of the operation `timed_receive()` is the same as that of the operation `timed_send()` (with an additional store of  $m$  into the local variable  $x$  if the rendezvous succeeds, line 16).

**Theorem 20** *The algorithm described in Fig. 13.15 ensures that both processes return the same value. Moreover this value is `commit` if and only the predicate  $(\tau_p + \delta + \theta \leq \text{deadline}_q) \wedge (\tau_q + \delta + \theta \leq \text{deadline}_p)$  is satisfied, and then the message  $m$  is delivered by  $q$ .*

```

operation timed_send( $m, deadline_p$ ) to  $q$  is
(1)   $\tau_p \leftarrow ph\_clock_p$ ;
(2)  send MSG( $m, \tau_p, deadline_p$ ) to  $q$ ;
(3)  within  $deadline_p$  do
(4)    when READY( $\tau_q, deadline_q$ ) is received from  $q$  do
(5)      if ( $\tau_p + \delta + \theta \leq deadline_q$ )  $\wedge$  ( $\tau_q + \delta + \theta \leq deadline_p$ )
(6)        then return(commit)
(7)        else return(timeout)
(8)      end if
(9)  at deadline occurrence return(timeout)
(10) end within.

operation timed_receive( $x, deadline_q$ ) from  $p$  is
(11)  $\tau_q \leftarrow ph\_clock_q$ ;
(12) send READY( $m, \tau_q, deadline_q$ ) to  $p$ ;
(13) within  $deadline_q$  do
(14)  when MSG( $\tau_p, deadline_p$ ) is received from  $q$  do
(15)    if ( $\tau_p + \delta + \theta \leq deadline_q$ )  $\wedge$  ( $\tau_q + \delta + \theta \leq deadline_p$ )
(16)      then  $x \leftarrow m$ ; return(commit)
(17)      else return(timeout)
(18)    end if
(19)  at deadline occurrence return(timeout)
(20) end within.

```

**Fig. 13.15** Real-time rendezvous between two processes  $p$  and  $q$

*Proof* Let us observe that if each process receives a control message from the other process, they necessarily return the same result (commit or timeout) because they compute the same predicate on the same values. Moreover, if no process receives a message by its deadline, it follows from the temporal scope construct that they both return timeout.

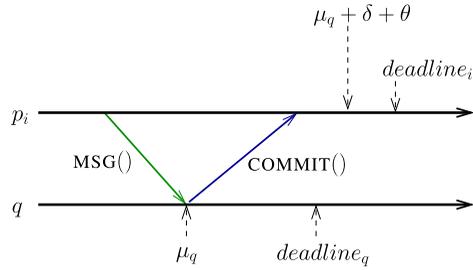
If a process receives a control message by its deadline, while the other does not, we have the following where, without loss of generality,  $p$  be the process that does not receive a message by its deadline. Hence  $p$  returns timeout. As it has not received the message READY( $\tau_q, deadline_q$ ) by its deadline, we have  $\tau_q + \delta + \theta > deadline_p$ . It follows that, when evaluated by  $q$ , the predicate will be false, and  $q$  will return timeout (lines 17).

Finally, it follows from lines 15–16 that  $q$  delivers the application  $m$  if and only if the predicate is satisfied.  $\square$

**A Simple Improvement** If the message READY() (resp., MSG()) has already arrived when the sender (resp., receiver) invokes its operation, this process can immediately evaluate the predicate and return timeout if the predicate is false. In that case, it may also save the sending of the message MSG() (resp., READY()). If the predicate is true, it executes its algorithm as described in Fig. 13.15.

It follows that, whatever its fate (commit or timeout), a real-time rendezvous between two processes requires one or two implementation messages.

**Fig. 13.16** When the rendezvous must be successful (asymmetric algorithm)



### 13.4.3 Introducing Nondeterministic Choice

In some applications, a sender process may want to propose a rendezvous with deadline to any receiver process of a predefined set of processes, or a receiver may want to propose a rendezvous to any sender of a predefined set of processes. The first rendezvous that is possible is then committed. We consider here the second case (multiple senders and one receiver). Each of the senders and the receiver define their own deadline. Let  $p_1, p_2, \dots, p_n$  denote the senders and  $q$  denote the receiver.

**Principle of the Solution: A Predicate for the Receiver** As the invocation of  $timed\_send(m, deadline_i)$  appears in a deterministic context, the implementation of this operation is close to that of Fig. 13.15, namely, the sender  $p$  sends the message  $send\ MSG(m, deadline_i)$  to  $q$ .

On the other hand, differently from the two-process case, the receiver process  $q$  plays now a particular role, namely, it has to select a single sender from the  $n$  senders. Moreover, the rendezvous is allowed to fail only if no rendezvous is possible, which means that  $q$  has to select one of them if several rendezvous are possible.

Let  $\mu_q$  be the date (measured with its physical clock) at which the receiver  $q$  receives the message  $MSG(m, deadline_i)$  from the sender  $p_i$ . If the rendezvous is possible with  $p_i$ ,  $q$  sends back the message  $COMMIT()$ . Otherwise it sends no message to  $p_i$ , which will obtain the value `timeout` when its deadline will be attained.

Hence, as before, the core of the algorithm is the predicate used by the receiver to answer (positively) or not answer a sender. Let us observe that  $\mu_q \leq deadline_q$  is a necessary requirement to answer, but this is not sufficient to ensure that both  $q$  and  $p_i$  take the same decision concerning their rendezvous (`commit` or `timeout()`). To that end, similarly to the two-process case and assuming that the message  $COMMIT()$  sent by  $q$  to  $p_i$  is sent at time  $\mu_q$ , it is required that this message  $COMMIT()$  is received by  $p_i$  before its deadline, i.e.,

$$\mu_q + \delta + \theta \leq deadline_i.$$

This is illustrated in Fig. 13.16. If this predicate is false, there is no certainty that the control message it is about to send to  $p_i$  will arrive at  $p_i$  before its deadline. Hence, if the predicate is false,  $q$  does not send the message  $COMMIT()$  to  $p_i$ .

```

operation timed_send( $m, deadline_i$ ) to  $q$  is    % invoked by  $p_i$  %
(1)  send MSG( $m, deadline_i$ ) to  $q$ ;
(2)  within  $deadline_i$  do
(3)    when COMMIT() is received from  $q$  do return(commit)
(4)  at deadline occurrence return(timeout)
(5)  end within.

operation timed_receive( $x, deadline_q$ ) is    % invoked by  $q$  %
(6)  within  $deadline_q$  do
(7)    when MSG( $m, deadline_i$ ) is received from  $p_i$  do
(8)      if ( $ph\_clock_q + \delta + \theta \leq deadline_i$ )
(9)        then send COMMIT() to  $p_i$ ;
(10)        $x \leftarrow m$ ; return(commit)
(11)    end if
(12)  at deadline occurrence return(timeout)
(13)  end within.

```

**Fig. 13.17** Nondeterministic rendezvous with deadline

As we can see, differently from the two-process case where the algorithms executed by the sender and the receiver are symmetric, the principle is now based on an asymmetric relation: The sender (which is in a deterministic context) acts as a client, while the receiver (which is in a nondeterministic context) acts as a server. Despite this asymmetry, the aim is the same as before: allow the processes to take a consistent decision concerning their rendezvous.

**The Asymmetric Algorithm for Nondeterministic Choice** The algorithm is described in Fig. 13.17. When a process  $p_i$  invokes  $\text{timed\_send}(m, \text{deadline}_i)$ , it sends the implementation message  $\text{MSG}(m, \text{deadline}_i)$  to  $q$  (line 1), and enters a temporal scope construct whose ending date is upper bounded by  $\text{deadline}_i$  (line 2). If, while it is in its temporal scope,  $p_i$  receives a message from  $q$ , its rendezvous with  $q$  is committed and it returns the value `commit` (line 3). If it receives no message from  $q$  by its deadline, the rendezvous failed and  $p_i$  returns the value `timeout` (line 4).

When it invokes  $\text{timed\_receive}(x, \text{deadline}_q)$ , the receiver  $q$  enters a temporal scope construct upper bounded by its deadline (line 6). Then it starts waiting. If during this period it receives a message  $\text{MSG}(m, \text{deadline}_i)$  from a sender  $p_i$  such that the predicate  $h\_clock_q + \delta + \theta \leq \text{deadline}_i$  is satisfied, it commits the rendezvous with  $p_i$  by sending back to  $p_i$  the message `COMMIT()` (lines 7–11). If no message satisfying this predicate is received by its deadline, no rendezvous can be committed and, consequently,  $q$  returns the value `timeout` (line 12).

**Remark** This asymmetric algorithm can be used when there is a single sender. We then obtain an asymmetric algorithm for a rendezvous between two processes (see Problem 5).

### 13.4.4 *n*-Way Rendezvous with Deadline

**The Problem** In some real-time applications, processes need to have a rendezvous with deadline involving all of them. This can be seen as a decision problem in which all the processes must agree on the same output (`commit` or `timeout`). Moreover, if the output is `commit`, each process must have received the messages sent by each other process. This section presents an algorithm solving *n*-way rendezvous with deadline (where *n* is the number of processes). This algorithm can be modified to work with predefined subsets of processes.

The operation offered to processes for such a multirendezvous is denoted `multi_rdv(m, deadline)` where *m* is the message sent by the invoking process *p<sub>i</sub>*,  $1 \leq i \leq n$ , and *deadline* is its deadline for the global rendezvous.

**The Predicate** As each process is now simultaneously a sender and a receiver, the algorithm is a symmetric algorithm, in the sense that all the processes execute the same code. Actually, this algorithm is a simple extension of the symmetric algorithm for two processes, described in Fig. 13.15.

Hence, when a process *p<sub>i</sub>* invokes `multi_rdv(mi, deadlinei)`, it sends the message `MSG(mi,  $\tau_i$ , deadlinei)` to each other process (where, as before,  $\tau_i$  is the sending date of this message). Let us consider a process *p<sub>i</sub>* that has received such a message from each other process. As the rendezvous is global, the two-process predicate  $(\tau_p + \delta + \theta \leq \textit{deadline}_q) \wedge (\tau_q + \delta + \theta \leq \textit{deadline}_p)$  has to be replaced by a predicate involving all pairs of processes. We consequently obtain the generalized predicate

$$\forall(k, \ell): \quad \tau_k + \delta + \theta \leq \textit{deadline}_\ell.$$

Instead of considering all pairs  $(\tau_k, \textit{deadline}_\ell)$ , this predicate can be refined as follows:

- The *n* starting dates  $\tau_1, \dots, \tau_n$ , are replaced by a single one, namely the worst one from the predicate point of view, i.e., the latest sending time  $\tau = \max(\{\tau_x\}_{1 \leq x \leq n})$ .
- Similarly, the *n* starting deadlines can be replaced by the worst one from the predicate point of view, i.e., the earliest deadline  $\textit{deadline} = \min(\{\textit{deadline}_x\}_{1 \leq x \leq n})$ .

The resulting predicate is consequently:

$$\tau + \delta + \theta \leq \textit{deadline}.$$

**The Algorithm** As indicated, the resulting algorithm is a simple extension of the two-process symmetric algorithm of Fig. 13.15. It is described in Fig. 13.18.

The local variable *rec<sub>i</sub>* is used to count the number of messages received by *p<sub>i</sub>*. If the deadline occurs before *rec<sub>i</sub>* = *n* - 1, one or more processes are late with respect to *p<sub>i</sub>*'s deadline and, consequently, the multirendezvous cannot occur. In this case, *p<sub>i</sub>* returns the value `timeout` (line 13). Otherwise, *p<sub>i</sub>* has received a message from each other process before its deadline. In this case, *p<sub>i</sub>* returns `commit` or `timeout` according to the value of the predicate (line 9–12). The proof that this algorithm is correct is a straightforward generalization of the proof of Theorem 20.

```

operation multi_rdv( $m, deadline_i$ ) is % invoked by  $p_i$  %
(1)  $rec_i \leftarrow 0$ ;  $\tau_i \leftarrow ph\_clock_i$ ;
(2) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do send MSG( $m, \tau_i, deadline_i$ ) to  $p_j$  end for;
(3) within  $deadline_i$  do
(4)   while ( $rec_i < n - 1$ ) do
(5)     when MSG( $m_j, \tau_j, deadline_j$ ) is received from  $p_j$  do  $rec_i \leftarrow rec_i + 1$ 
(6)   end while;
(7)   let  $\tau = \max(\{\tau_x\}_{1 \leq x \leq n})$ ;
(8)   let  $deadline = \max(\{deadline_x\}_{1 \leq x \leq n})$ ;
(9)   if ( $\tau + \delta + \theta \leq deadline$ )
(10)    then return(commit)
(11)    else return(timeout)
(12)  end if
(13) at deadline occurrence return(timeout)
(14) end within.

```

**Fig. 13.18** Multirendezvous with deadline

Observing that for any  $i$ , it is not necessary that the constraint  $\tau_i + \delta + \theta \leq deadline_i$  be satisfied, an improved predicate, which allows for more rendezvous to succeed, can be designed. This is the topic of Problem 7.

## 13.5 Summary

This chapter was on synchronous communication. This type of communication synchronizes the sender and the receiver, and is consequently also called rendezvous or interaction. The chapter has first given a precise meaning to what is called *synchronous communication*. It has also presented a characterization based on a specific message pattern called a crown.

The chapter then presented several algorithms implementing synchronous communication, each appropriate to a specific context: planned interactions or forced interaction in fully asynchronous systems, and rendezvous with deadline suited to synchronous systems.

## 13.6 Bibliographic Notes

- The notion of a synchronous communication originated in the work of P. Brinch Hansen [63] (who introduced the notion of distributed processes), and C.A.R. Hoare [186] (who introduced the notion of communicating sequential processes, thereafter known under the name CSP).

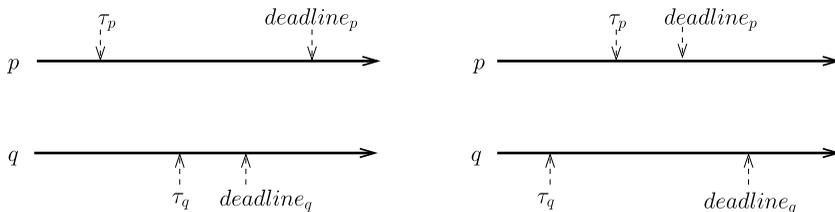
The CSP language constructs relate intimately synchronous communication with the notion of guarded command and nondeterminism introduced by E.W. Dijkstra [113].

The notion of remote procedure call developed in some distributed systems [54] is strongly related to synchronous communication.

- The definition of synchronous communication based on logical scalar clocks (as used in this book) is due to V.K. Garg [149, 274]. The example used in Sect. 13.1.2 is from [149].
  - The characterization of synchronous communication based on the absence of the message pattern called a crown is due to B. Charron-Bost, G. Tel, and F. Mattern [88].
- Another characterization based on the acyclicity of a message graph is given by T. Soneoka and T. Ibaraki in [356].
- The asymmetric algorithm based on an acyclic client–server relation among the processes, presented in Sect. 13.2.2, is due to A. Silberschatz [343].
  - The asymmetric token-based algorithm presented in Sect. 13.2.3 is due to R. Bagrodia [31].
  - The algorithm for forced (message delivery) interactions presented in Sect. 13.3 is due to V.V. Murty and V.K. Garg [273].
  - Other algorithms implementing synchronous rendezvous in asynchronous systems, and analyses of the concepts of synchrony and nondeterminism associated with synchronous communications, can be found in many papers (e.g., [30, 51, 57, 58, 66, 80, 94, 131, 133, 354]).
  - The notion of multiparty interaction in asynchronous systems is addressed in [85, 121].
  - The notion of rendezvous with deadline and the algorithms presented in Sect. 13.4 are due to I. Lee and S.B. Davidson [233].
  - Other notions of synchronous communication have been introduced, mainly in the domain of distributed simulation [140, 199, 263, 329].

## 13.7 Exercises and Problems

1. Let us recall that a sequential observation  $\widehat{S} = (H, \xrightarrow{to})$  of a distributed execution  $\widehat{H} = (H, \xrightarrow{ev})$  is a topological sort of the partial order  $\widehat{H}$ , i.e., the total order  $\xrightarrow{to}$  respects the partial order  $\xrightarrow{ev}$  (see Chap. 6).  
 Show that the communication pattern of a distributed execution  $\widehat{H}$  satisfies the synchronous communication property if and only if  $\widehat{H}$  has a sequential observation  $\widehat{S} = (H, \xrightarrow{to})$  in which, for each message  $m$ , the receive event immediately follows the send event (i.e., both events can be packed to form a single event).
2. Enrich the algorithm presented in Sect. 13.2.2 so that:
  - During each interaction, a message is transmitted in each direction.
  - Some interactions allow a message in each direction to be transmitted, while other interactions allow only one message to be transmitted (as in Sect. 13.2.2).
3. Let us consider the token-based implementation for planned interactions described in Fig. 13.10. Given two processes  $p_i$  and  $p_j$ , determine an upper bound on the number of messages exchanged by these processes to commit a rendezvous or discover that it cannot be realized.  
 Solution in [31].



**Fig. 13.19** Comparing two date patterns for rendezvous with deadline

- Design an algorithm implementing a rendezvous in which each message is sent to a set of destination processes. The communication operations offered to processes are “`synchr_send(m) to dest(m)” and “synchr_del()”, where dest(m) is the set of processes to which m is sent, and synchr_del() allows the invoking process to deliver a message sent by any process. The set dest(m) can be different for each message m. Moreover, the communication context is that of forced interactions (see Sect. 13.3).`

Everything has to appear in such a way that, for each message *m*, the same logical date can be associated with its send event and its delivery events by the processes of *dest(m)* (synchrony property stated in Sect. 13.1.1).

Solution in [272].

- Let us consider rendezvous with deadline between two processes. Compare the respective advantages of the symmetric algorithm of Fig. 13.15 and the asymmetric algorithm of Fig. 13.17 (instantiated with a single sender). To that end, it is interesting to compare them in the two date patterns described in Fig. 13.19, where *p* is the sender and *q* the receiver.
- Let us consider rendezvous with deadline in a synchronous system where the maximal drift of a local clock with respect to real-time is bounded by  $\epsilon$ . This physical time setting was introduced in Sect. 9.5.5. Let  $\Delta$  be a real-time duration. This quantity of time is measured by a process  $p_i$ , with its physical clock, as a value  $\Delta_i$  such that

$$(1 - \epsilon)\Delta \leq \Delta_i \leq (1 + \epsilon)\Delta.$$

The left side corresponds to the case where the physical clock of  $p_i$  is slow, while the right side corresponds to the case where it is rapid. Hence, if a clock of a process is always rapid while the clock of another process is always slow, the difference between these two clocks can become arbitrarily large.

Modify the algorithms described in Figs. 13.15, 13.17, and 13.18, so that they are based on the assumption  $\epsilon$  (instead of the assumption  $\theta$ , which assume an underlying clock synchronization algorithm). What do you notice for the modified algorithm from Fig. 13.17?

Solution in [233].

- Let us consider the algorithm for multirendezvous with deadline presented in Sect. 13.4.4. As noticed at the end of this section, it is not necessary that, for any *i*, the constraint  $\tau_i + \delta + \theta \leq \text{deadline}_i$  be satisfied.

To show this, in addition to  $\tau$  and  $deadline$ , let us define two new values as follows:  $\tau'$  is the second greatest sending time of a message  $MSG(m, \tau_i, deadline_i)$  sent at line 2, and  $deadline'$  is the second smallest deadline value. These values can be computed by a process at line 7 and line 8. Moreover, let  $same\_proc(\tau, deadline)$  be a predicate whose value is true if and only if the values  $\tau$  and  $deadline$  come from the same process. Show that the predicate  $(\tau + \delta + \theta \leq deadline)$  used at line 9 can be replaced by the following weaker predicate:

$$\begin{aligned} & [same\_proc(\tau, deadline) \wedge (\tau + \delta + \theta \leq deadline') \wedge (\tau' + \delta + \theta \leq deadline)] \\ & \vee [\neg same\_proc(\tau, deadline) \wedge (\tau + \delta + \theta \leq deadline)]. \end{aligned}$$

# Part V

## Detection of Properties on Distributed Executions

The two previous parts of the book were on the enrichment of the system to provide processes with high-level operations. Part III was on the definition and the implementation of operations suited to the consistent use of shared resources, while Part IV introduced communication abstractions with specific ordering properties. In both cases, the aim is to allow application programmers to concentrate on their problems and not on the way some operations have to be implemented.

This part of the book is devoted to the observation of distributed computations. Solving an observation problem consists in superimposing a distributed algorithm on a computation, which records appropriate information on this computation in order to be able to detect if it satisfies some property. The specificity of the information is, of course, related to the property one is interested in detecting.

Two detection problems are investigated in this part of the book: the detection of the termination of a distributed execution (Chap. 14), and the detection of deadlocks (Chap. 15). Both properties “the computation has terminated” and “there is deadlock” are stable properties, i.e., once satisfied they remain satisfied in any future state of the computation.

**Remark** Other property detection problems concern the detection of unstable properties such as the conjunction of local predicates or the detection of properties on execution flows. Their detection is a more advanced topic not covered in this book. The interested reader is invited to consult the following (non-exhaustive) list of references [73, 89, 98, 137, 139, 153, 154, 192, 193, 378].

# Chapter 14

## Distributed Termination Detection

This chapter is on the detection of the termination of a distributed computation. This problem was posed and solved for the first time in the early 1980s independently by E.W. Dijkstra and C.S. Scholten (1980) and N. Francez (1980). This is a non-trivial problem. While, in sequential computing, the termination of the only process indicates that the computation has terminated, this is no longer true in distributed computing. Even if we were able to observe simultaneously all the processes, observing all of them passive could not allow us to conclude that the distributed execution has terminated. This is because some messages can still be in transit, which will reactivate their destination processes when they arrive, and these re-activations will, in turn, entail the sending of new messages, etc.

This chapter presents several models of asynchronous computations and observation/detection algorithms suited to termination detection in each of them. As in other chapters, the underlying channels are not required to be FIFO. Moreover, while channels are bidirectional, the term “output” channels (resp., “input” channels) is used when considering message send (resp., message reception).

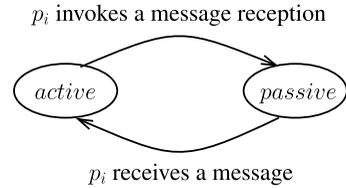
**Keywords** AND receive · Asynchronous system · Atomic model · Counting · Diffusing computation · Distributed iteration · Global state ·  $k$ -out-of- $n$  receive statement · Loop invariant · Message arrival vs. message reception · Network traversal · Non-deterministic statement · OR receive statement · Reasoned construction · Receive statement · Ring · Spanning tree · Stable property · Termination detection · Wave

### 14.1 The Distributed Termination Detection Problem

#### 14.1.1 Process and Channel States

As in previous chapters,  $p_1, \dots, p_n$  denote the set of asynchronous processes. Moreover, we assume that the underlying asynchronous communication network is fully connected.

**Fig. 14.1** Process states for termination detection



**Process States** Each process  $p_i$  has a local variable denoted  $state_i$ , the value of which is *active* or *passive*. Initially, some processes are in the active state, while the others are in the passive state. Moreover, at least one process is assumed to be in the active state. As far as the behavior of a process  $p_i$  is concerned, we have the following (Fig. 14.1):

- When  $state_i = active$ :
  - $p_i$  can execute local computations and send messages to the other processes.
  - $p_i$  can invoke a message reception. In that case, the value of  $state_i$  automatically changes from *active* to *passive*, and  $p_i$  starts waiting for a message.
- When,  $p_i$  receives a message (we have then  $state_i = passive$ ), the value of  $state_i$  automatically changed from *passive* to *active*.

It is assumed that there is an invocation of a message reception for every message that is sent. (This assumption will be removed in Sect. 14.5.)

**Channel States** While each channel is bidirectional at the network level, it is assumed that at the application level the channels are unidirectional. Given two processes  $p_i$  and  $p_j$ , this allows us, from an observer’s point of view, to distinguish the state of the channel from  $p_i$  to  $p_j$  from the state of the channel from  $p_j$  to  $p_i$ .

The abstract variable  $empty_{(i,j)}$  is used to describe the state of the channel from  $p_i$  to  $p_j$ . It is a Boolean variable whose value is *true* if and only if there is no application message in transit  $p_i$  to  $p_j$ . Differently from the local variables  $state_i$ , the variables  $empty_{(i,j)}$  are “abstract” because no process can instantaneously know their value (the sender does not know when messages arrive, and the receiver does not know when they are sent).

### 14.1.2 Termination Predicate

**The Predicate** Intuitively, a distributed computation is terminated if there is a time at which, simultaneously, all the processes are passive and all the channels are empty.

More formally, let us consider an external time reference (real-time, which is not known by the processes);  $\tau$  being any date in this time framework, let  $state_i^\tau$  and

$empty_{(i,j)}^\tau$  denote the value of  $state_i$  and  $empty_{(i,j)}$  at time  $\tau$ , respectively.  $\mathcal{C}$  being a distributed execution, let us define the predicate  $TERM(\mathcal{C}, \tau)$  as follows:

$$TERM(\mathcal{C}, \tau) \equiv [(\forall i: state_i^\tau = passive) \wedge (\forall i, j: empty_{(i,j)}^\tau)].$$

This predicate captures the fact that, at time  $\tau$ ,  $\mathcal{C}$  has terminated. Finally, the predicate that captures the termination of  $\mathcal{C}$ , denoted  $TERM(\mathcal{C})$ , is defined as follows:

$$TERM(\mathcal{C}) \equiv (\exists \tau: TERM(\mathcal{C}, \tau)).$$

As already seen, a stable property is a property that, once true, remains true forever.

**Theorem 21**  *$TERM(\mathcal{C})$  is a stable property.*

*Proof* Assuming that  $TERM(\mathcal{C}, \tau)$  is satisfied, let  $\tau' \geq \tau$ . As, at time  $\tau$ , no process is active and all channels are empty, it follows from the rules governing the behavior of the processes that no process can be re-activated by a message reception. Consequently, as only active processes can send messages, no new message will be sent after  $\tau$ . Hence  $TERM(\mathcal{C}, \tau) \Rightarrow TERM(\mathcal{C}, \tau')$ .  $\square$

### 14.1.3 The Termination Detection Problem

Given a distributed computation  $\mathcal{C}$ , the termination detection problem consists in designing an algorithm  $A$  that satisfies the two following properties:

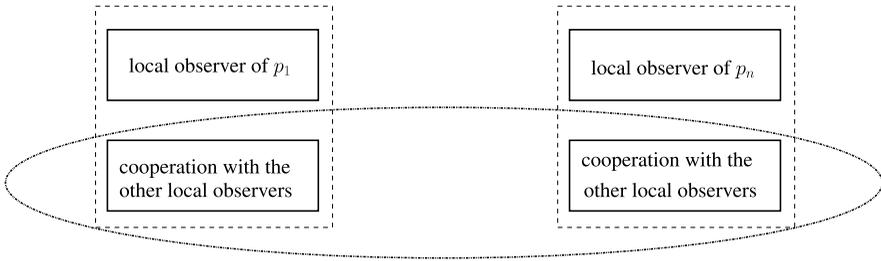
- **Safety.** If, at time  $\tau$ , the algorithm  $A$  claims that  $\mathcal{C}$  has terminated, then  $TERM(\mathcal{C}, \tau)$  is true (i.e., the computation has terminated at some time  $\tau' \leq \tau$ ).
- **Liveness.** If  $\mathcal{C}$  terminates at time  $\tau$  (i.e.,  $TERM(\mathcal{C}, \tau) \wedge (\forall \tau' < \tau: \neg TERM(\mathcal{C}, \tau'))$ ), then there is a time  $\tau''$  at which  $A$  will claim that the computation has terminated.

As usual, the safety property is on consistency: An algorithm is not allowed to claim termination before it occurs. Safety alone is not sufficient as, if termination occurs, it allows an algorithm to never claim termination. This is prevented by the liveness property.

#### 14.1.4 Types and Structure of Termination Detection Algorithms

The algorithms detecting the termination of distributed computations are structured in two parts (see Fig. 14.2):

- At each process, a local observer module that observes the corresponding application process. Basically, the local observer at a process  $p_i$  manages the local variable  $state_i$  and a few other control variables related to the state and the communications of  $p_i$  (e.g., number of messages sent and received by  $p_i$ ).



**Fig. 14.2** Global structure of the observation modules

- A distributed algorithm (with a module per process) which allows the local observers to cooperate in order to detect the termination.

The termination detection algorithms differ on two points: (a) the assumptions they do on the behavior of the computation, and (b) the way they cooperate.

As far as notations are concerned, the local observer of a process  $p_i$  is denoted  $obs_i$ .

**Remark** As termination is a stable property, it is possible to use a stable property detection algorithm based on global states (such as the one presented in Sect. 6.5.3) to detect termination.

The design of specific termination detection algorithms is motivated by efficiency and by the fact that termination detection algorithms are restricted to use only consistent global states. They may rely on both consistent and inconsistent global states.

## 14.2 Termination Detection in the Asynchronous Atomic Model

### 14.2.1 The Atomic Model

The atomic model is a simplified model in which only messages take time to travel from their senders to their destination processes. Message processing is done *atomically* in the sense that it appears as taking no time. Hence, when it can be observed a process is always passive.

A time-diagram of an execution in this model is represented in Fig. 14.3. Message processing is denoted by black dots. Initially,  $p_2$  sends a message  $m_1$  to  $p_1$ , while  $p_1$  and  $p_3$  are waiting for messages. When it receives  $m_1$ ,  $p_1$  processes it and sends two messages,  $m_2$  to  $p_3$  and  $m_3$  to  $p_2$ , etc. The processing of the messages  $m_4$ ,  $m_7$ , and  $m_8$  entails no message sending, and consequently (from a global observer point of view), the distributed computation has terminated after the reception and the processing of  $m_8$  by  $p_2$ .

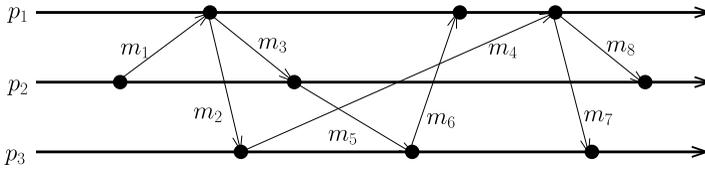


Fig. 14.3 An execution in the asynchronous atomic model

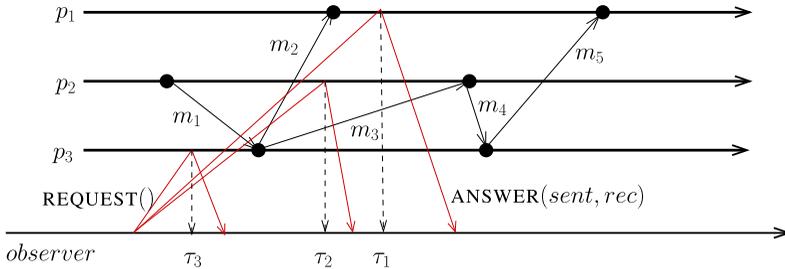


Fig. 14.4 One visit is not sufficient

### 14.2.2 The Four-Counter Algorithm

**An Inquiry-Based Principle** The idea that underlies this algorithm is fairly simple. On the observation side, each process  $p_i$  is required to count the number of messages it has received  $rec_i$ , and the number of messages it has sent  $sent_i$ .

To simplify the presentation we consider that there is an additional control process denoted *observer*. This process is in charge of the detection of the termination. It sends an inquiry message to each process  $p_i$ , which answers by returning its pair of values  $(sent_i, rec_i)$ . When the observer has all the pairs, it computes the total number of messages which have been sent  $S$ , and the total number of messages which have been received  $R$ . If  $S \neq R$  it starts its next inquiry.

Unfortunately, as shown by the counterexample presented in Fig. 14.4, the observer cannot conclude from  $S = R$  that the computation has terminated. This is due to the asynchrony of communication. The inquiry messages `REQUEST()` are not received at the same time by the application processes, and  $p_1$  sends back the message `ANSWER(0, 1)`,  $p_2$  sends back the message `ANSWER(1, 0)`, and  $p_3$  sends back the message `ANSWER(0, 0)`. We then have  $S = 1$  and  $R = 1$ , while the computation is not yet terminated. Due to the asynchrony among the reception of the messages `REQUEST()` at different processes, the final counting erroneously associates the sending of  $m_1$  with the reception of  $m_2$ . Of course, it could be possible to replace the counting mechanism by recording the identity of all the messages sent and received and then compare sets instead of counters, but this would be too costly and, as we are about to see, there is a simpler counter-based solution.

```

% local observation at each process  $p_i$  %
when a message  $m$  is received from a process  $p_j$  do
(1)  $rec_i \leftarrow rec_i + 1$ ;
(2) let  $x$  = number of messages sent due to the processing of  $m$ ;
(3)  $sent_i \leftarrow sent_i + x$ .

% interaction of  $p_i$ 's local observer with the centralized observer %
when a message REQUEST() is received from observer do
(4) send ANSWER( $sent_i, rec_i$ ) to observer.

detection task observer is % behavior of the centralized observer %
(5)  $S1 \leftarrow 0$ ;  $R1 \leftarrow 0$ ;
(6) repeat forever
(7) for each  $i \in \{1, \dots, n\}$  do send REQUEST() to  $p_i$  end for;
(8) wait (an answer message ANSWER( $sent_i, req_i$ ) received from each  $p_i$ );
(9)  $S2 \leftarrow \sum_{1 \leq i \leq n} sent_i$ ;  $R2 \leftarrow \sum_{1 \leq i \leq n} rec_i$ ;
(10) if ( $R1 = S2$ ) then claim termination; exit loop
(11) else  $S1 \leftarrow S2$ ;  $R1 \leftarrow R2$ 
(12) end if
(13) end repeat.

```

**Fig. 14.5** The four-counter algorithm for termination detection

**An Algorithm Based on Sequential Inquiries** A solution, due to F. Mattern (1987), consists in replacing an inquiry by two consecutive inquiries. The resulting algorithm is described in Fig. 14.5.

When an application process  $p_i$  receives a message it processes the message and increases accordingly  $rec_i$  and  $sent_i$  (lines 1–3). Moreover, it sends back the message ANSWER( $sent_i, req_i$ ) (line 4) when it receives a request from the observer.

The observer sends a request message to each application process (line 7) and waits for the corresponding answers (line 8). It then computes the results of its inquiry and stores them in  $S2$  and  $R2$  (line 9).  $S1$  and  $R1$  denote the values of its previous inquiry (lines 5 and 11). If  $S2 = R1$ , it claims termination. Otherwise, it starts a new inquiry. When considering two consecutive inquiries, let us observe that, when true, the termination predicate states that the number of messages perceived as sent by the second inquiry is equal to the number of messages perceived as received by the first inquiry. As we are about to see in the proof of the next theorem, the important point lies in the fact that inquiries are sequential, and this allows us to cope with fact that each inquiry is asynchronous.

**Theorem 22** *Assuming that the application executes in the asynchronous atomic model, the 4-counter algorithm satisfies the safety and liveness properties defining the termination detection problem.*

*Proof* Let us first prove the liveness property. If the computation terminates, there is a time  $\tau$  after which no message is sent. Hence, after  $\tau$ , no message is sent and no message is received. Let us consider the first two inquiries launched by the observer

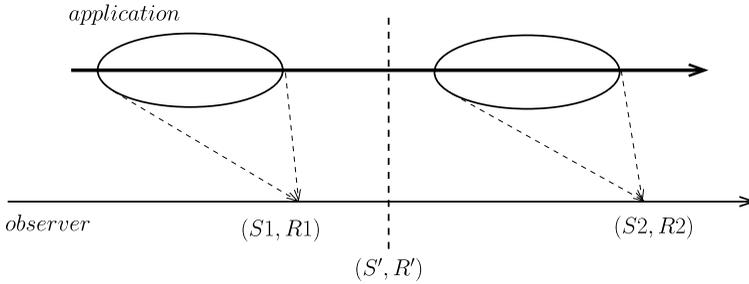


Fig. 14.6 Two consecutive inquiries

after  $\tau$ . It follows from the previous observation that  $E1 = R1 = E2 = R2$ , and the algorithm detects termination.

Proving the safety property consists in showing that, if termination is claimed, then the computation has terminated. To that end, let  $sent_i^\tau$  be the value of  $sent_i$  at time  $\tau$ . As counters are not decreasing, we trivially have  $(\tau \leq \tau') \Rightarrow (sent_i^\tau \leq sent_i^{\tau'})$ , and the same holds for the counters  $rec_i$ .

Let us consider Fig. 14.6, which abstracts two consecutive inquiries launched by the observer. The first inquiry obtains the pair of values  $(S1, R1)$ , and the second inquiry obtains the pair  $(S2, R2)$ . As the second inquiry is launched after the results of the previous one have been computed, they are sequential. Hence, there is a time, between the end of the first inquiry and the beginning of the send one at which the number of messages sent is  $S'$ , and the number of messages received is  $R'$  (these values are known neither by the processes, nor the observer). We have the following.

- $S1 \leq S' \leq S2$  and  $R1 \leq R' \leq R2$  (from the previous observation on the nondecreasing property of the counters  $sent_i$  and  $rec_i$ ).
- If  $S2 = R1$ , we obtain (from the previous item)  $S' \leq S2 = R1 \leq R'$ .
- $S' \geq R'$  (due to the computation itself).
- It follows from the two previous items that  $(S2 = R1) \Rightarrow (S' = R')$ .

Hence, if the predicate is satisfied, the computation was terminated before the second inquiry was launched, which concludes the proof of the theorem.  $\square$

### 14.2.3 The Counting Vector Algorithm

**Principle** The principle that underlies this detection algorithm, which is due to F. Mattern (1987), consists in using a token that navigates the network of processes. This token carries a vector  $msg\_count[1..n]$  such that, from its point of view,  $msg\_count[i]$  is the number of messages sent to  $p_i$  minus the number of messages received by  $p_i$ . When a process receives the token and the token is such that  $msg\_count = [0, \dots, 0]$ , it claims termination.

```

% local observation at each process  $p_i$  %
when a message  $m$  is received from a process  $p_j$  do
(1)  $msg_i[i] \leftarrow msg_i[i] - 1$ ;
(2) let  $sent_i[k]$  = number of messages sent to  $p_k$  due to the processing of  $m$ ;
(3) for each  $k \in \{1, \dots, n\} \setminus \{i\}$  do  $msg_i[k] \leftarrow msg_i[k] + sent_i[k]$  end for.

local detection task at process  $p_i$  % cooperation among local observers %
(4) repeat forever
(5)   wait (TOKEN( $msg\_count$ ));
(6)   for each  $k \in \{1, \dots, n\}$  do
(7)      $msg\_count[k] \leftarrow msg\_count[k] + msg_i[k]$ ;
(8)      $msg_i[k] \leftarrow 0$ 
(9)   end for;
(10)  if (all processes have been visited at least once)  $\wedge$  ( $msg\_count = [0, \dots, 0]$ )
(11)    then claim termination; exit loop
(12)    else send TOKEN( $msg\_count[1..n]$  to) next process on the ring
(13)  end if
(14) end repeat.

```

**Fig. 14.7** The counting vector algorithm for termination detection

**Algorithm** The algorithm is described in Fig. 14.7. It assumes that a process does not send messages to itself. Moreover, in order that all processes are visited by the token, it also assumes that the token moves along a ring including all the processes. (According to Fig. 14.2, this is the way the local observers cooperate to detect termination.)

Each process  $p_i$  manages a local vector  $msg_i[1..n]$  (initialized to  $[0, \dots, 0]$ ) such that  $msg_i[i]$  is decreased by 1 each time  $p_i$  receives a message, and  $msg_i[j]$  is increased by 1 each time  $p_i$  sends a message to  $p_j$  (lines 1–3). Moreover, when the token is at  $p_i$ ,  $msg_i[1..n]$  is added to the vector  $msg\_count[1..n]$ , and reset to its initial value (lines 6–9).

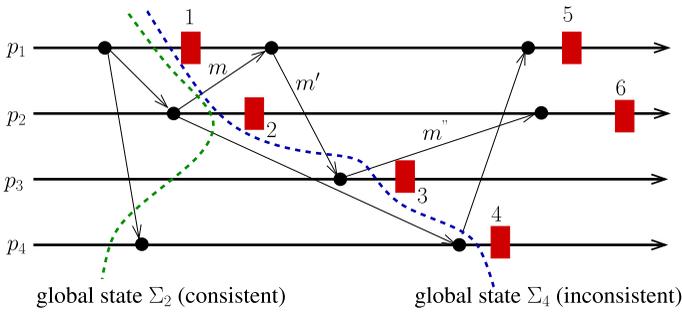
As previously, let  $xx^\tau$  be the value of  $x$  at time  $\tau$ . Let  $nb\_msg_i^\tau$  denote the number of messages in transit to the process  $p_i$  at time  $\tau$ . The algorithm maintains the following invariant:

$$\forall i: \left( msg\_count^\tau[i] + \sum_{1 \leq k \leq n} msg_k^\tau[i] \right) = nb\_msg_i^\tau.$$

After a process  $p_i$  has received the token and updated its value, it checks the predicate ( $msg\_count = [0, \dots, 0]$ ) if the token has visited each process at least once (line 10). If the predicate is satisfied,  $p_i$  claims termination (line 11). Otherwise, it propagates the token to the next process (line 12).

**Example** An example illustrating the algorithm is described in Fig. 14.8. The visit of the token at each process is indicated with a rectangle. The order in which the token visits the processes is indicated by integers representing “real time”.

We have the following. The value of the array  $msg\_count[1..n]$  is its value after its has been updated by its current owner (i.e., after line 9).



**Fig. 14.8** The counting vector algorithm at work

- At time  $\tau = 1$ :  $msg\_count^1[1..n] = [0, 1, 0, 1]$ .
- At time  $\tau = 2$ :  $msg\_count^2[1..n] = [1, 0, 0, 2]$ .
- At time  $\tau = 3$ :  $msg\_count^3[1..n] = [1, 1, -1, 2]$ .
- At time  $\tau = 4$ :  $msg\_count^4[1..n] = [2, 1, -1, 0]$ .
- At time  $\tau = 5$ :  $msg\_count^5[1..n] = [0, 1, 0, 0]$ .
- At time  $\tau = 6$ :  $msg\_count^6[1..n] = [0, 0, 0, 0]$ .

$msg\_count^2[1..n] = [1, 0, 0, 2]$  because, to the token’s knowledge, a message (denoted  $m$ ) is in transit to  $p_1$ , no messages are in transit to  $p_2$  and  $p_3$ , and a message is in transit to  $p_4$ .  $msg\_count^4[1..n] = [2, 1, -1, 0]$  because, to the token’s knowledge, two messages are recorded as being in transit to  $p_1$ , a message is in transit to  $p_2$ , a message (namely  $m'$ ) has been received by  $p_3$  but this message is not yet recorded as sent (hence the value  $-1$ ), and no messages are in transit to  $p_4$ .

While the computation has terminated when the token visits  $p_1$  at time  $\tau = 5$ , the content of  $msg\_count^5$  does not allow  $p_1$  to claim termination (the token has not yet seen the reception of the message  $m''$  by  $p_2$ ). Differently, at time  $\tau = 6$ , the content of  $msg\_count^6$  allows  $p_2$  to detect termination.

**Termination Detection, Global States, and Cuts** The notions of a *cut* and of a *global state* were introduced in Sects. 6.1.3, and 6.3.1, respectively. Moreover, we have seen that these notions are equivalent.

Actually, the value of the vector  $msg\_count$  defines a global state. The global state  $\Sigma_2$  associated with  $msg\_count^2$  and the global state  $\Sigma_4$  associated with  $msg\_count^4$  are represented on the figure.

As we can see, the global states associated with vectors all of whose values are not negative are consistent, while the ones associated with vectors in which some values are negative are inconsistent. This comes from the fact that a negative value in  $msg\_count[i]$  means that  $p_i$  is seen by the token as having received messages, but their corresponding sendings have not yet been recorded by the token (as is the case for  $msg\_count^3$  and  $msg\_count^4$ , which do not record the sending of  $m'$ ). The positive entries of a vector with only non-negative values indicate how many messages have been sent and are not yet received by the corresponding process (as

an example, in the consistent global state associated with  $msg\_count^5$ , the message  $m''$  sent to  $p_2$  is in transit).

#### 14.2.4 *The Four-Counter Algorithm vs. the Counting Vector Algorithm*

**Similarity** In both detection algorithms, no application message is overloaded with control information. Observation messages and application messages are independent.

**Difference** The four-counter algorithm requires an additional observer process. Moreover, after the computation has terminated, the additional observer process may have to issue up to two consecutive inquiries before claiming termination.

The counting vector algorithm does not require an additional process, but needs an underlying structure that allows the token to visit all the processes. This structure is not required to be a ring. Between two visits to a given process  $p_i$ , the token may visit any other process a finite number of times. Moreover, after termination has occurred, the token can claim it during the first visit (of all the processes), which starts after the computation has terminated. This is, for example, the case in Fig. 14.8. Other features of this algorithm are addressed in Problem 1.

### 14.3 Termination Detection in Diffusing Computations

This section and the following ones consider the base asynchronous model in which a process can be active or passive as defined in Sect. 14.1.1.

#### 14.3.1 *The Notion of a Diffusing Computation*

Some applications are structured in such a way that initially a single process is active. Without loss of generality, let  $p_1$  be this process (which is sometimes called the environment).

Process  $p_1$  can send messages to other processes, which then become active. Let  $p_i$  such a process. It can, in turn, send messages to other processes, etc. (hence the name *diffusing computation*). After a process has executed some local computation and possibly sent messages to other processes, it becomes passive. It can become active again if it receives a new message. The aim is here to design a termination detection algorithm suited to diffusing computations.

### 14.3.2 A Detection Algorithm Suited to Diffusing Computations

**Principle: Use a Spanning Tree** As processes become active when they receive messages, the idea is to capture the activity of the application with a spanning tree which evolves dynamically. A process that is not in the tree enters the tree when it becomes active, i.e., when it receives a message. A process leaves the tree when there is no more activity in the application that depends on it. This principle and the corresponding detection algorithm are due to E.W. Dijkstra and C.S. Scholten (1980).

**How to Implement the Previous Idea** Initially, only  $p_1$  is in the tree. Then, we have the following. Let us first consider a process  $p_i$  that receives a message from a process  $p_j$ . As we will see, a process that sends a message necessarily belongs to the tree. Moreover, due to the rules governing the behavior of a process, a process is always passive when it receives a message (see Fig. 14.1).

- If  $p_i$  is not in the tree, it becomes active and enters the tree. To that end, it defines the sender  $p_j$  of the message as its parent in the tree. Hence, each process  $p_i$  manages a local variable  $parent_i$ . Initially,  $parent_i = \perp$  at any process  $p_i$ , except for  $p_1$ , for which we have  $parent_1 = 1$ .
- If  $p_i$  is in the tree when it receives the message from  $p_j$ , it becomes active and, as it does not need to enter the tree, it sends by return to  $p_j$  a message  $ACK()$  so that  $p_j$  does not consider it as one of its children.

Let us now introduce a predicate that allows a process to leave the tree. First, the process has to be passive. But this is not sufficient. If a process  $p_i$  sent messages, these messages created activity, and it is possible that, while  $p_i$  is passive, the activated processes are still active (and may have activated other processes, etc.). To solve this issue, each process  $p_i$  manages a local variable, called  $deficit_i$ , which counts the number of messages sent by  $p_i$  minus the number of acknowledgment messages it has received.

- If  $deficit_i > 0$ , all the messages sent by  $p_i$  have not yet been acknowledged. Hence, there is possibly some activity in the subtree rooted at  $p_i$ . Consequently,  $p_i$  conservatively remains in the tree.
- If  $deficit_i = 0$ , all the messages that  $p_i$  has sent have been acknowledged, and consequently no more activity depends on  $p_i$ . In this case, it can leave the tree, and does this by sending the message  $ACK()$  to, its parent.

Hence, when considering a process  $p_i$  which is in the tree (i.e., such that  $parent_i \neq \perp$ ), the local predicate that allows it to leave the tree is

$$(state_i = passive) \wedge (deficit_i = 0).$$

Finally, the environment process  $p_1$  concludes that the diffusing computation has terminated when its local predicate  $deficit_1 = 0$  becomes satisfied.

**Fig. 14.9**

Termination detection  
of a diffusing computation

```

when  $p_i$  starts waiting for a message do
(1)  $state_i \leftarrow passive;$ 
(2) let  $k = parent_i;$ 
(3) if ( $deficit_i = 0$ )
(4)   then send ACK() to  $p_k$ ;  $parent_i \leftarrow \perp$ 
(5) end if.

when  $p_i$  executes send( $m$ ) to  $p_j$  do
  % ( $state_i = active$ )  $\wedge$  ( $parent_i \neq \perp$ ) %
(6)  $deficit_i \leftarrow deficit_i + 1.$ 

when  $p_i$  receives a message  $m$  from  $p_j$  do
(7)  $state_i \leftarrow active;$ 
(8) if ( $parent_i = \perp$ ) then  $parent_i \leftarrow j$ 
(9)   else send ACK() to  $p_j$ 
(10) end if.

when ACK() is received from  $p_j$  do
(11)  $deficit_i \leftarrow deficit_i - 1;$ 
(12) let  $k = parent_i;$ 
(13) if ( $deficit_i = 0$ )  $\wedge$  ( $state_i = passive$ )
(14)   then send ACK() to  $p_k$ ;  $parent_i \leftarrow \perp$ 
(15) end if.

```

**The Algorithm: Local Observation and Cooperation Between Observers** The algorithm is described in Fig. 14.9. In this algorithm the local observation of a process and the cooperation among the local observers are intimately related.

When a process  $p_i$  starts waiting for a message, it becomes passive and leaves the tree if all its messages have been acknowledged, i.e., it is not the root of a subtree in which some processes are possibly still active (lines 1–5). On the observation side, the local deficit is increased each time  $p_i$  sends a message (line 6). The process  $p_i$  is then necessarily active and belongs to the tree.

When a process  $p_i$  receives a message  $m$ , it becomes active (line 7), and enters the tree if it was not there (line 8). If  $p_i$  was already in the tree, it sends by return a message ACK() to the sender of  $m$  (line 9) to inform the sender that its was already in the tree and its activity does not depends on it.

Finally, when a process receives a message ACK(), it leaves the tree (line 14) if it is passive and is not the root of a subtree in which some processes are active (line 13).

## 14.4 A General Termination Detection Algorithm

This section presents a reasoned construction of a general termination detection algorithm. This algorithm is general in the sense that it assumes neither that processing times have a zero duration, nor that the observed computation is a diffusing computation (it allows any number of processes to be initially active). This algorithm is

also generic in that sense that it uses an abstraction called a *wave*, which can be implemented in many different ways, each providing a specific instance of the general algorithm. This reasoned construction is due to J.-M. Hélyar and M. Raynal (1991).

### 14.4.1 Wave and Sequence of Waves

**Definition** A *wave* is a control flow that is launched by a single process, visits each process once, and returns to the process that activated it. Hence, the notion of a *wave initiator* is associated with a wave. As a wave starts from a process and returns to that process, it can be used to disseminate information to each process and returns information from each process to the wave initiator.

As an example, each inquiry launched by the observer process in the four-counter algorithm (Sect. 14.2.2) is a wave. More generally, any distributed algorithm implementing a network traversal with feedback (such as the ones presented in Chap. 1) implements a wave.

A wave provides the processes with four operations, denoted `start_wave()`, `end_wave()`, `forward_wave()`, and `return_wave()`. Assuming the initiator is  $p_\alpha$ , let  $sw_\alpha^x$  and  $ew_\alpha^x$  denote the events corresponding to the execution of its  $x$ th invocation of `start_wave()` and `end_wave()`, respectively. Similarly, given any process  $p_i$ ,  $i \neq \alpha$ , let  $fw_i^x$  and  $rw_i^x$  denote the events corresponding to the execution of the  $x$ th invocation of `forward_wave()` and `return_wave()` by  $p_i$ , respectively.

The control flow associated with a sequence of waves initiated by  $p_\alpha$  is defined by the following properties (where “ $\xrightarrow{ev}$ ” is the causal order relation on events defined in Sect. 6.1.2).

- Process visit.  $\forall x: \forall i \neq \alpha: sw_\alpha^x \xrightarrow{ev} fw_i^x \xrightarrow{ev} rw_i^x \xrightarrow{ev} ew_\alpha^x$ . This property states that each process is visited by a wave before this wave returns to its initiator.
- Sequence of waves.  $\forall x: ew_\alpha^x \xrightarrow{ev} sw_\alpha^{x+1}$ . This property states that an initiator cannot launch the next wave before the previous has returned to it.

It is easy to see that the sequence of inquiries used in the four-counter algorithm (Fig. 14.5) is a sequence of waves.

**Implementing a Wave** In the following, two implementations of a wave are presented. They are canonical in the sense that they are based on the canonical structures which are a ring and a spanning tree. It is assumed that, in each wave, (a) the values returned to the initiator by the processes are Boolean values, and (b) the initiator is interested in the “and” of these values. (Adapting the implementation to other types of values, and operations more sophisticated than the “and” can be easily done.)

These implementations consider that the role of  $p_\alpha$  is only to control the waves, i.e.,  $p_\alpha$  is an additional process which is not observed. Hence, an application process

**Fig. 14.10** Ring-based implementation of a wave

```

operation start_wave() is % invoked by  $p_\alpha$  %
(1) send TOKEN (true) to  $next_\alpha$ .

operation forward_wave() is % invoked by  $p_i, i \neq \alpha$  %
(2) wait (TOKEN (r));  $x_i \leftarrow r$ .

operation return_wave(b) is % invoked by  $p_i, i \neq \alpha$  %
(3) send TOKEN ( $x_i \wedge b$ ) to  $next_i$ .

operation end_wave(res) is % invoked by  $p_\alpha$  %
(4) wait (TOKEN (r));  $res \leftarrow r$ .

```

is denoted  $p_i$ , where  $i \in \{1, \dots, n\}$  and  $\alpha \notin \{1, \dots, n\}$ . Modifying the wave implementations so that  $p_\alpha$  is also an application process (i.e., a process that is observed) is easy. (This is the topic of Problem 3.)

Let us observe that it is not required that all the waves of a sequence of waves are implemented the same way. It is possible that some waves are implemented in one way (e.g., ring-based implementation), while the other waves are implemented differently (e.g., tree-based implementation).

**Implementing a Wave on Top of a Ring** Let us assume an underlying unidirectional ring including each process exactly once (the construction of such a ring has been addressed in Sect. 1.4.2). This communication structure allows for a very simple implementation of a wave. The successor on the ring of a process  $p_i$  is denoted  $next_i$ .

The control flow is represented by the progress of a message on the ring. This message, denoted TOKEN(), carries a Boolean whose value is the “and” of the values supplied by the process it has already visited. The corresponding implementation (which is trivial) is described in Fig. 14.10.

When it launches a new wave, the initiator sends the message TOKEN (*true*) to its successor (line 1). Then, when a process  $p_i$  is visited by the wave, it stores the current value carried by the token  $r$  in a local variable  $x_i$  (line 2). To forward the wave, it sends the message TOKEN ( $x_i \wedge b$ ) to its successor, where  $b$  is its local contribution to the wave, and  $x_i$  is the contribution of the processes which have been already visited by the wave (line 3). Finally, when the initiator invokes end\_wave(*res*), it waits for the token and deposits its value in a local variable *res*.

**Implementing a Wave with a Spanning Tree** Let us assume a spanning tree rooted at the initiator process  $p_\alpha$  (several algorithms building spanning trees were presented in Chap. 1). Let  $parent_i$  and  $children_i$  denote the parent of  $p_i$  and the set of its children in this tree, respectively (if  $p_i$  is a leaf,  $children_i = \emptyset$ ).

If the spanning tree is a star centered at  $p_\alpha$ , the implementation reduces to the use of REQUEST() and ANSWER() messages, similarly to what is done in Fig. 14.5.

The tree-based implementation of a wave is described in Fig. 14.11. When the initiator  $p_\alpha$  invokes start\_wave(), it sends a message GO() to each of its children

```

operation start_wave() is % invoked by  $p_\alpha$  %
(1) for each  $j \in \text{children}_\alpha$  do send GO () to  $p_j$  end for.

operation forward_wave() is % invoked by  $p_i, i \neq \alpha$  %
(2) wait (GO () from  $p_{\text{parent}_i}$ );
(3) for each  $j \in \text{children}_i$  do send GO () to  $p_j$  end for.

operation return_wave( $b$ ) is % invoked by  $p_i, i \neq \alpha$  %
(4) if ( $\text{children}_i \neq \emptyset$ )
(5)   then wait (BACK( $v_j$ ) has been received from each  $p_j, j \in \text{children}_i$ );
(6)      $x_i \leftarrow (\bigwedge_{j \in \text{children}_i} v_j)$ 
(7)   else  $x_i \leftarrow \text{true}$ 
(8) end if;
(9) send BACK ( $x_i \wedge b$ ) to  $p_{\text{parent}_i}$ .

operation end_wave( $res$ ) is % invoked by  $p_\alpha$  %
(10) wait (BACK( $v_j$ ) has been received from each  $p_j, j \in \text{children}_\alpha$ );
(11)  $res \leftarrow (\bigwedge_{j \in \text{children}_\alpha} v_j)$ .

```

**Fig. 14.11** Spanning tree-based implementation of a wave

(line 1) and, when a process  $p_i$  is visited by the wave (i.e., receives a message GO()), it forwards the message GO() to each of its own children (lines 2–3).

Then, when a process  $p_i$  invokes return\_wave( $b$ ), where  $b$  is its contribution to the final result, it waits for the contributions from the processes belonging to the subtree for which it is the root (lines 5–9). When, it has received all of these contributions, it sends back to its parent the whole contribution of this subtree (line 10). Finally, when  $p_\alpha$  (which is the root of the spanning tree) has received the contribution of all its children, it deposits the Boolean result in its local variable  $res$ .

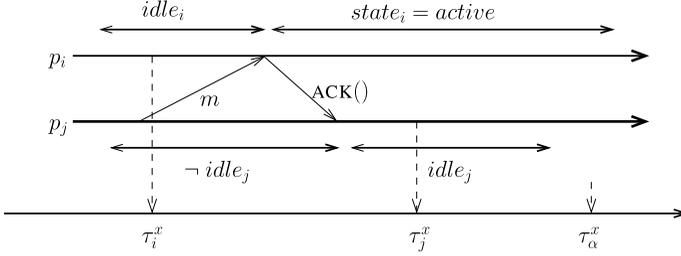
## 14.4.2 A Reasoned Construction

**A Reasoned Construction: Step 1 (Definition of a Locally Evaluable Unstable Predicate)** Let us recall that (as seen in Sect. 14.1.2), a computation  $\mathcal{C}$  is terminated at time  $\tau$  if the predicate  $TERM(\mathcal{C}, \tau)$  is true, where  $TERM(\mathcal{C}, \tau) \equiv (\forall i: \text{state}_i^\tau = \text{passive}) \wedge (\forall i, j: \text{empty}_{(i,j)}^\tau)$ .

While no process can instantaneously know the value of the predicate  $\bigwedge_{j \neq i} \text{empty}_{(i,j)}$ , it is possible to replace it by a stronger predicate, by associating an acknowledgment with each application message. (Differently from the acknowledgments used in the specific algorithm designed for diffusing computations, such acknowledgment messages are systematically sent by return when application messages are received.)

Let  $\text{deficit}_i$  be a local control variable of  $p_i$  which counts the number of application messages sent by  $p_i$  and for which  $p_i$  has not yet received the corresponding acknowledgment. Hence,  $p_i$  can locally evaluate the following unstable predicate

$$\text{idle}_i \equiv (\text{state}_i = \text{passive}) \wedge (\text{deficit}_i = 0).$$



**Fig. 14.12** Why  $(\bigwedge_{1 \leq i \leq n} idle_i^x) \Rightarrow TERM(\mathcal{C}, \tau_\alpha^x)$  is not true

This predicate is safe in the sense that  $idle_i \Rightarrow (state_i = passive) \wedge (\bigwedge_{j \neq i} empty_{(i,j)})$ . It allows consequently each process  $p_i$  to know the state of the channels from it to the other processes. Let us nevertheless observe that this predicate is unstable. This is because, if  $p_i$  receives a message while  $idle_i$  is satisfied,  $state_i$  becomes equal to  $active$ , and  $idle_i$  becomes consequently false.

**A Reasoned Construction: Step 2 (Strengthening the Predicate)** Let  $\tau_i^x$  be the time (from an external observer point of view) at which a process  $p_i$ , which is visited by the  $x$ th wave, starts the invocation of `return_wave()`, and  $\tau_\alpha^x$  be the time at which  $p_\alpha$  terminates its  $x$ th invocation of `end_wave()`. Let  $idle_i^x$  be the value of the predicate  $idle_i$  at time  $\tau_i^x$ .

Unfortunately, we do not have  $(\bigwedge_{1 \leq i \leq n} idle_i^x) \Rightarrow TERM(\mathcal{C}, \tau_\alpha^x)$ . This is due to the fact that the instants at which the predicates  $idle_i$  are evaluated and the corresponding process  $p_i$  invokes `return_wave()` are independent from each other. This is depicted in Fig. 14.12. Assuming the current wave is the  $x$ th, it is easy to see that, despite the fact that both  $idle_i^x$  and  $idle_j^x$  are true,  $p_\alpha$  cannot conclude at time  $\tau_\alpha^x$  that the computation has terminated.

**Continuously Passive Process** This counterexample shows that termination detection cannot boil down to a simple collection of values. The visits of the wave to processes have to be coordinated, and the activity period of each process has to be recorded in one way or another. To that end, let us strengthen the local predicate  $idle_i^x$  by adding the following predicate  $ct\_pass_i^x$  defined as follows:

$$ct\_pass_i^x \equiv p_i \text{ remained continuously passive between } \tau_i^x \text{ and } \tau_\alpha^x.$$

When satisfied, this predicate means that  $p_i$  has not been reactivated after the  $x$ th wave left it and returned at  $p_\alpha$ . The important point is the fact that we have the following:

$$\bigwedge_{1 \leq i \leq n} (idle_i^x \wedge ct\_pass_i^x) \Rightarrow TERM(\mathcal{C}, \tau_\alpha^x).$$

This follows from the following observation. It follows from  $idle_i^x \wedge ct\_pass_i^x$  that  $p_i$  remained passive during the time interval  $[\tau_i^x, \tau_\alpha^x]$ , and consequently its outgoing channels remained empty in this time interval. As this is true for all the processes,

we conclude that, at time  $\tau_\alpha^x$ , all the processes are passive and all the channels are empty.

Unfortunately, as a process  $p_i$  does not know the time  $\tau_\alpha^x$ , it cannot evaluate the predicate  $ct\_pass_i^x$ . As previously, a simple way to solve this problem consists in strengthening the predicate  $ct\_pass_i^x$  in order to obtain a predicate locally evaluable by  $p_i$ . As the waves are issued sequentially by  $p_\alpha$ , such a strengthening is easy. As  $\tau_\alpha^x < \tau_i^{x+1}$ ,  $\tau_\alpha^x$  (which is not known by  $p_i$ ) is replaced by  $\tau_i^{x+1}$  (which is known by  $p_i$ ). Hence, let us replace  $ct\_pass_i^x$  by

$$sct\_pass_i^{[x,x+1]} \equiv p_i \text{ remained continuously passive between } \tau_i^x \text{ and } \tau_i^{x+1}.$$

We trivially have

$$\bigwedge_{1 \leq i \leq n} (idle_i^x \wedge sct\_pass_i^{[x,x+1]}) \Rightarrow TERM(\mathcal{C}, \tau_\alpha^x).$$

### A Reasoned Construction: Step 3 (Loop Invariant and Progress Condition)

The predicate  $\bigwedge_{1 \leq i \leq n} (idle_i^x \wedge sct\_pass_i^{[x,x+1]})$  involves two consecutive waves, and each process is involved in the evaluation of two local predicates, namely,  $idle_i^x$  and  $sct\_pass_i^{[x,x+1]}$ . As a sequence of waves is nothing else than a distributed iteration controlled by the initiator process  $p_\alpha$ , and, due to the application, there is no guarantee that a process remains continuously passive between two consecutive waves, this directs us to consider

- $\bigwedge_{1 \leq i \leq n} idle_i^x$  as the loop invariant, and
- $\bigwedge_{1 \leq i \leq n} sct\_pass_i^{[x,x+1]}$  as the progress condition associated with the loop.

**A Reasoned Construction: Step 4 (Structure the Algorithm)** A local observer is associated with each process  $p_i$ . In addition to the local variables  $state_i$  and  $deficit_i$ , it manages the Boolean variable  $cont\_passive_i$ . As far as initial values are concerned, we have the following:  $state_i$  is initialized to *active* or *passive*, according to the fact that  $p_i$  is initially active or passive;  $deficit_i$  is initialed to 0;  $cont\_passive_i$  is initialized to *true* if and only if  $p_i$  is initially passive.

The algorithm is described in Fig. 14.13. The local observation of a process  $p_i$  is described at lines 1–6. The new point with respect to the algorithms that have been previously presented is line 4. When  $p_i$  receives a message, in addition of becoming active, its Boolean variable  $cont\_passive_i$  is set to *false*.

**A Reasoned Construction: Step 5 (Guaranteeing the Loop Invariant)** The loop invariant and the progress condition associated with the sequence of waves are implemented at lines 7–16, which describe the cooperation among the local observers.

After it has been visited by a wave (invocation of `forward_wave()` at line 12), a process waits until its local predicate  $idle_i$  is satisfied (line 13). When this occurs it invokes `return_wave(b)` (line 15) where  $b$  is the current value of  $cont\_passive_i$ .

**Fig. 14.13**

A general algorithm  
for termination detection

```

when  $p_i$  starts waiting for a message do
(1)  $state_i \leftarrow passive$ .

when  $p_i$  executes  $send(m)$  to  $p_j$  do
(2)  $deficit_i \leftarrow deficit_i + 1$ .

when  $p_i$  receives a message from  $p_j$  do
(3)  $state_i \leftarrow active$ ;
(4)  $cont\_passive_i \leftarrow false$ ;
(5)  $send\ ACK()$  to  $obs_j$ .

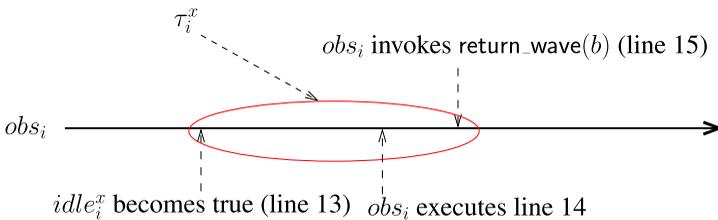
when a message  $ACK()$  is received do
(6)  $deficit_i \leftarrow deficit_i - 1$ .

local task at  $p_i$  for the cooperation between observers is
(7) if ( $i = \alpha$ )
(8)   then repeat
(9)      $start\_wave(); end\_wave(res)$ 
(10)    until ( $res$ ) end repeat;
(11)     $claim\ termination$ 
(12)  else  $forward\_wave();$ 
(13)    wait ( $(state_i = passive) \wedge (deficit_i = 0)$ );
(14)     $b \leftarrow cont\_passive_i; cont\_passive_i \leftarrow true$ ;
(15)     $return\_wave(b)$ 
(16) end if.

```

The important point is that, thanks to the **wait** statement, the predicate  $idle_i$  is true at time  $\tau_i^x$ . As this is true at any process, we have  $\bigwedge_{1 \leq i \leq n} idle_i^x$ , i.e., the loop invariant is satisfied. For  $idle_i$  to be true at time  $\tau_i^x$ , it is assumed that the application process is frozen (i.e., does not execute) from the time  $idle_i$  becomes true (line 13) until  $\tau_i^x$ , i.e., until  $return\_wave(b)$  starts to be executed (line 15). This is depicted in Fig. 14.14. This freezing is introduced to simplify the presentation, it is not mandatory (as shown in Problem 4).

**A Reasoned Construction: Step 6 (Ensuring Progress)** At time  $\tau_i^x$ ,  $p_i$  resets  $cont\_passive_i$  to *true*. (line 14), and will transmit its value  $b$  to  $(x + 1)$ th wave when it will invoke  $return\_wave(b)$  during this wave. Thanks to the “and” on the values collected by the  $(x + 1)$ th wave, the Boolean value  $res$  obtained by the initiator  $p_\alpha$

**Fig. 14.14** Atomicity associated with  $\tau_i^x$

at the end of the  $(x + 1)$ th wave is such that  $res = \bigwedge_{1 \leq i \leq n} sct\_pass_i^{[x,x+1]}$ . If this Boolean is true,  $p_\alpha$  claims termination (lines 10–11).

As we have then  $\bigwedge_{1 \leq i \leq n} idle_i^x$  (due to the loop invariant) and  $res = \bigwedge_{1 \leq i \leq n} sct\_pass_i^{[x,x+1]}$ , it follows that  $TERM(\mathcal{C}, \tau_\alpha^x)$  is true, and consequently the claim is true. Hence, if the computation terminates, its termination is detected. Moreover, the fact that there is no false claim follows from the use of the Boolean variables  $cont\_passive_i$  which are used to implement the predicates  $sct\_pass_i^{[x,x+1]}$ .

## 14.5 Termination Detection in a Very General Distributed Model

### 14.5.1 Model and Nondeterministic Atomic Receive Statement

**Motivation** In the classical asynchronous model (considered previously) each receive statement involves a single message, the sender of which is any process.

This basic receive statement is not versatile enough to render an account of specific cases where, for example, a process  $p_i$  wants to atomically receive either two messages, one from a given process  $p_j$  and the other from a given process  $p_k$ , or a single message from a given process  $p_\ell$ . This means that, when such a receive statement terminates,  $p_i$  has consumed atomically either two messages (one from  $p_j$  and one from  $p_k$ ), or a single message (from  $p_\ell$ ).

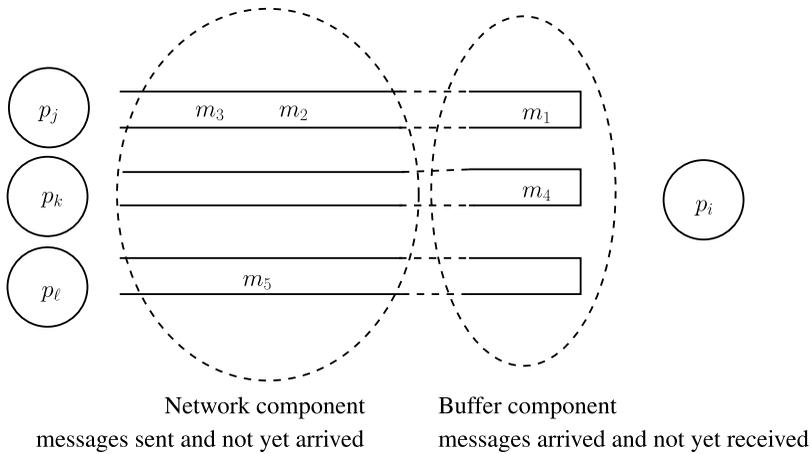
**Message Arrival vs. Message Reception** As suggested by the previous example, the definition of powerful nondeterministic reception statements needs the introduction of the notion of an *arrived and not yet consumed* message. Hence, a channel is now made up of two parts: a *network* component (that a message uses to attain its destination) and a *buffer* component (where a message that has arrived is stored until its destination process receives and consumes it).

There is consequently a notion of *message arrival* which is different from the notion of a *message reception* (these notions are merged in the classical asynchronous communication model). It follows that the underlying communication model is composed of a communication network plus an input buffer for each pair of processes  $(p_j, p_i)$ , where the message arrived from  $p_j$  and not yet received by  $p_i$  are saved.

This means that there is a part of each input channel, namely the buffer component, which is now visible by the observer  $obs_i$  associated with each process  $p_i$ . Differently, the content of the network component of the channels remains locally invisible.

A simple example is depicted in Fig. 14.15. The messages  $m_1$  (sent by  $p_j$ ) and  $m_4$  (sent by  $p_k$ ) have arrived, but have not yet been received. They are stored in input buffers. The messages  $m_2$ ,  $m_3$ , and  $m_5$  are still on their way to  $p_i$ .

**Dependency Set** Providing versatile message reception requires us to define a generalized nondeterministic reception statement. To that end, the notion of a dependency set is first introduced.



**Fig. 14.15** Structure of the channels to  $p_i$

When it enters a receive statement, a process  $p_i$  specifies a set of process identities, denoted  $dep\_set_i$ , composed of the processes from which it starts waiting for messages. Let us remark that this allows a message, which has been sent, to never be consumed by its destination process (because the sender never appears in the dependency set of its destination process). Several patterns of dependency sets can be defined, each one defining a specific message reception model.

**AND Model** In this reception pattern, a receive statement has the following structure:

receive message from ( $p_a$  and ... and  $p_x$ ).

We have then  $dep\_set_i = \{a, \dots, x\}$ . When invoked by  $p_i$ , this statement terminates when a message from each process  $p_j$ , such that  $j \in dep\_set_i$ , has arrived at  $p_i$ . The reception statement then withdraws these messages from their input buffers and returns them to  $p_i$ . Hence, this message reception statement allows for the atomic reception of several messages from distinct senders.

**OR Model** The receive statement of the OR pattern has the following structure:

receive message from ( $p_a$  or ... or  $p_x$ ).

As previously, the dependency set is  $dep\_set_i = \{a, \dots, x\}$ , but its meaning is different. This receive statement terminates when a message from a process  $p_j$ , such that  $j \in dep\_set_i$ , has arrived at  $p_i$ . If messages from several processes in  $dep\_set_i$  have arrived, a single one is withdrawn from its input buffer and received and consumed by  $p_i$ . Hence, the “or” is an exclusive “or”. This is consequently a simple nondeterministic receive statement.

**OR/AND Model** This reception pattern is a combination of the two previous ones. The receive statement has the following form.

receive message from  $dp_1$  or  $\dots$  or  $dp_x$ ,

where each  $dp_y$ ,  $1 \leq y \leq x$ , is a set of processes, and (as before) the “or” is an exclusive “or”. Moreover,  $dep\_set_i = \bigcup_{1 \leq y \leq x} dp_y$ .

This statement terminates as soon as there is a set  $dp_y$  such that a message from each process  $p_j$ , with  $j \in dp_y$ , has arrived at  $p_i$ . This receive statement consumes only the messages sent by these processes. It is a straightforward generalization of the AND and OR reception models.

**Basic  $k$ -out-of- $m$  Model** The receive statement of the  $k$ -out-of- $m$  pattern has the following structure:

receive  $k_i$  messages from  $dep\_set_i$ ,

where  $dep\_set_i$  (a set of process identities). It is assumed that  $1 \leq k_i \leq |dep\_set_i|$ . This statement terminates when a message from  $k_i$  distinct processes belonging to  $dep\_set_i$  have arrived at  $p_i$ .

**Disjunctive  $k$ -out-of- $m$  Model** This receive statement generalizes the previous one.

receive ( $k_i^1$  messages from  $dp_1$ ) or  
 $(k_i^2$  messages from  $dp_2$ ) or  $\dots$  or  
 $(k_i^x$  messages from  $dp_x$ ).

We have  $dep\_set_i = \bigcup_{1 \leq y \leq x} dp_y$ . This statement terminates as soon as there is a set  $dp_y$  such that  $k_i^y$  messages have arrived from distinct processes belonging to  $dp_y$ .

The disjunctive  $k$ -out-of- $m$  model is fairly general. When  $x = 1$ , we obtain the basic  $k$ -out-of- $m$  model. When  $x = 1$  and  $k_i^1 = |dp_1| = |dep\_set_i|$ , we obtain the AND model. When  $x \geq 1$  and  $|dp_y| = 1$ ,  $1 \leq y \leq x$ , we obtain the OR model.

**Unspecified Message Reception** It is possible that a message that has arrived at a process might never be consumed. This depends on the message pattern specified by processes when they invoke a reception statement. Maybe, in some cases, this can be due to an erroneous distributed program, but termination detection also has to work for the executions generated by such programs.

Interestingly, the algorithms developed in this section not only detect the termination of such executions, but allow also for the reporting of messages arrived and not received.

### 14.5.2 The Predicate *fulfilled()*

A predicate, denoted *fulfilled()*, is introduced to abstract the activation condition of a process that has invoked a receive statement. This predicate is used to know if the corresponding process can be re-activated.

Let  $A$  be a set of process identities such that a message has arrived from each of these processes and these messages have not yet been consumed (hence, they are still in their input buffers).  $fulfilled(A)$  is satisfied if and only if these messages are sufficient to reactivate the invoking process  $p_i$ .

By definition  $fulfilled(\emptyset)$  is equal to *false*. It is moreover assumed that the predicate  $fulfilled()$  is monotonous, i.e.,

$$(A \subseteq A') \Rightarrow [fulfilled(A) \Rightarrow fulfilled(A')].$$

(Monotonicity states only that, if a process can be reactivated with the messages from a set of processes  $A$ , it can also be reactivated with the messages from a bigger set of processes  $A'$ , i.e., such that  $A \subseteq A'$ .)

The result of an invocation of  $fulfilled(A)$  depends on the type of receive statement issued by the corresponding process  $p_i$ . As a few examples, we have the following:

- In the AND model:  $fulfilled(A) \equiv (dep\_set_i \subseteq A)$ .
- In the OR model:  $fulfilled(A) \equiv (A \cap dep\_set_i \neq \emptyset)$ .
- In the OR/AND model:  $fulfilled(A) \equiv (\exists dp_y : dp_y \subseteq A)$ .
- In the  $k$ -out-of- $n$  model:  $fulfilled(A) \equiv (|A \cap dep\_set_i| \geq k_i)$ .
- In the disjunctive  $k$ -out-of- $n$  model:  $fulfilled(A) \equiv (\exists y: |A \cap dp_i^y| \geq k_i^y)$ .

A process  $p_i$  stops executing because it is blocked on a receive statement, or because it has attained its “end” statement. This case is modeled as a receive statement in which  $dep\_set_i = \emptyset$ . As processes can invoke distinct receive statements at different times, the indexed predicate  $fulfilled_i()$  is used to denote the corresponding predicate as far as  $p_i$  is concerned.

### 14.5.3 Static vs. Dynamic Termination: Definition

**Channel Predicates and Process Sets** Let us define the following predicates and abstract variables that will be used to define two notions of termination of a computation in the very general distributed model.

- $empty(j, i)$  is a Boolean predicate (already introduced) which is true if and only if the network component of the channel from  $p_j$  to  $p_i$  is empty.
- $arrived_i(j)$  is a Boolean predicate which is true if and only if the buffer component of the channel from  $p_j$  to  $p_i$  is not empty.
- $arr\_from_i = \{j \mid arrived_i(j)\}$  (the set of processes from which messages have arrived at  $p_i$  but have not yet been received—i.e., consumed—by  $p_i$ ).
- $NE_i = \{j \mid \neg empty(j, i)\}$  (the set of processes that sent messages to  $p_i$ , and these messages have not yet arrived at  $p_i$ ).

When looking at Fig. 14.15, we have  $arr\_from_i = \{j, k\}$ , and  $NE_i = \{j, \ell\}$ .

**Static Termination** A distributed computation  $\mathcal{C}$  is *statically terminated* at some time  $\tau$  if the following predicate, denoted  $S\_TERM(\mathcal{C}, \tau)$ , is satisfied. The variables  $state_i$  have the same meaning as before (and  $xx^\tau$  is the value of  $xx$  at time  $\tau$ ).

$$S\_TERM(\mathcal{C}, \tau) \equiv [\forall i: (state_i^\tau = passive) \wedge (NE_i^\tau = \emptyset) \wedge (\neg fulfilled_i(arr\_from_i^\tau))].$$

This predicate captures the fact that, at time  $\tau$ , (a) all the processes are passive, (b) no application message is in transit in the network component, and (c) for each process  $p_i$  and according to its receive statement, the messages arrived and not yet consumed (if any) cannot reactivate it. Similarly to the definition of  $TERM(\mathcal{C})$  introduced in Sect. 14.1.2,  $S\_TERM(\mathcal{C})$  is defined as follows:

$$S\_TERM(\mathcal{C}) \equiv (\exists \tau: S\_TERM(\mathcal{C}, \tau)).$$

This definition of termination is focused on the states of processes and channels. It is easy to show that  $S\_TERM(\mathcal{C})$  is a stable property.

**Dynamic Termination** A distributed computation  $\mathcal{C}$  is *dynamically terminated* at some time  $\tau$  if the following predicate is satisfied, denoted  $D\_TERM(\mathcal{C}, \tau)$ , is satisfied.

$$D\_TERM(\mathcal{C}, \tau) \equiv [\forall i: (state_i^\tau = passive) \wedge (\neg fulfilled_i(NE_i^\tau \cup arr\_from_i^\tau))].$$

This predicate captures the fact that, at time  $\tau$ , (a) all the processes are passive, and (b) no process  $p_i$  can be reactivated when considering both the messages arrived and not yet consumed and the messages in transit to it. The important difference with the predicate  $S\_TERM(\mathcal{C}, \tau)$ , lies in the fact that it allows for early detection, namely, termination can occur and be detected even if some messages have not yet arrived at their destination processes. This definition is based on the real ( $state_i = passive$ ) or potential ( $fulfilled_i(NE_i \cup arr\_from_i)$ ) activity of processes and not on the fact that channels are or are not empty. As previously, let us define  $D\_TERM(\mathcal{C})$  as follows:

$$D\_TERM(\mathcal{C}) \equiv (\exists \tau: D\_TERM(\mathcal{C}, \tau)).$$

As all messages eventually arrive at their destination, it is easy to see that

$$D\_TERM(\mathcal{C}, \tau) \Rightarrow [\exists \tau' \geq \tau: S\_TERM(\mathcal{C}, \tau')].$$

In that sense, dynamic termination allows for “early” detection, while static termination allows only for “late” detection.

**Static/Dynamic Termination vs. Classical Termination** In the classical model for termination detection, each receive statement is for a single message from any process (hence there is no unspecified reception), and message arrival and message reception are merged (hence there is no notion of input buffer). Moreover, we have  $fulfilled(A) \equiv (A \neq \emptyset)$ . It follows that  $TERM(\mathcal{C}, \tau)$  can be rewritten as follows:

$$TERM(\mathcal{C}, \tau) \equiv [\forall i: (state_i^\tau = passive) \wedge (NE_i^\tau = \emptyset)].$$

The static and dynamic algorithms that follow are due to J. Brzezinsky, J.-M. H elary, and M. Raynal (1993).

### 14.5.4 Detection of Static Termination

The structure of the algorithm detecting the static termination of a computation  $\mathcal{C}$  (predicate  $S\_TERM(\mathcal{C})$ ) is the same as that of the algorithm in Fig. 14.13. A difference lies in the fact that the local observer  $obs_i$  of process  $p_i$  has now to consider the arrival of application messages instead of their reception.

**Local Variables and Locally Evaluable Predicate** As the set  $NE_i$  appears in  $S\_TERM(\mathcal{C}, \tau)$ , a value of it must consequently be computed or approximated. As a local observer  $obs_i$  cannot compute the state of its input channels without freezing (momentarily blocking) sender processes, it instead computes the state of its output channels. More precisely, as done in previous algorithms, a local observer can learn the state of the network component of its output channels by associating an acknowledgment with each application message. When a message from a process  $p_i$  arrives at a process  $p_j$ , the local observer  $obs_j$  sends by return an acknowledgment to the local observer  $obs_i$  associated with  $p_i$ . As previously, let  $deficit_i$  be the number of messages sent by  $p_i$  for which  $obs_i$  has not yet received an acknowledgment. When satisfied, the predicate  $deficit_i = 0$  allows the local observer  $obs_i$  to know that the all the messages by  $p_i$  have arrived to their destination processes. As we will see in the proof of the algorithm, replacing  $NE_i$  by  $deficit_i = 0$  allows for a safe detection of static termination.

To know the state of  $p_i$ 's input buffers, the local observer  $obs_i$  can use the set  $arr\_from_i$  (which contains the identities of the processes  $p_j$  such that there is a message from  $p_j$  in the input buffer of  $p_i$ ). The content of such a set  $arr\_from_i$  can be locally computed by the local observer  $obs_i$ . This is because this observer (a) sends back an acknowledgment each time a message arrives, and (b) observes the reception of messages by  $p_i$  (i.e., when messages are withdrawn from their input buffers to be consumed  $p_i$ ).

Finally, the predicate (which is not locally evaluable)

$$(state_i = passive) \wedge (NE_i = \emptyset) \wedge (\neg fulfilled_i(arr\_from_i))$$

is replaced by the locally evaluable predicate

$$(state_i = passive) \wedge (deficit_i = 0) \wedge (\neg fulfilled_i(arr\_from_i)).$$

In addition to the previous local variables, the algorithm uses also the Boolean variable  $cont\_passive_i$ , the role of which is exactly the same as in Fig. 14.13.

**Static Termination Detection Algorithm** The algorithm (which is close to the algorithm of Fig. 14.13) is described in Fig. 14.16. Lines 1–7 describe the observation part of the observer  $obs_i$  (at process  $p_i$ ). When a process enters a receive statement, the dependency set and the reactivation condition ( $fulfilled_i()$ ) associated with this receive statement are defined. When, according to messages arrived and not yet consumed, the predicate  $fulfilled_i(arr\_from_i)$  becomes true (lines 5–6), the process  $p_i$  becomes active and (as in Fig. 14.13) the Boolean  $cont\_passive_i$  is set to *false*. As already indicated, the role of  $cont\_passive_i$  is the same as in Fig. 14.13.

```

when  $p_i$  enters a receive statement do
(1)  compute  $dep\_set_i$  and  $fulfilled_i()$  from the receive statement;
(2)   $state_i \leftarrow passive$ .

when  $p_i$  executes  $send(m)$  to  $p_j$  do
(3)   $deficit_i \leftarrow deficit_i + 1$ .

when a message from  $p_j$  arrives do
(4)  send  $ACK()$  to  $obs_j$ .

when  $fulfilled_i(arr\_from_i)$  becomes satisfied do
(5)   $state_i \leftarrow active$ ;
(6)   $cont\_passive_i \leftarrow false$ .

when a message  $ACK()$  is received do
(7)   $deficit_i \leftarrow deficit_i - 1$ .

local task at  $p_i$  for the cooperation between observers is
(8)  if ( $i = \alpha$ )
(9)    then repeat
(10)     for each  $j \in \{1, \dots, n\}$  do send  $REQUEST()$  to  $p_j$  end for;
(11)     wait ( $ANSWER(b_j)$  received from each  $obs_j, j \in \{1, \dots, n\}$ );
(12)      $res \leftarrow \bigwedge_{1 \leq j \leq n} b_j$ 
(13)     until ( $res$ ) end repeat;
(14)     claim termination
(15)  else wait (message  $REQUEST()$  from  $obs_\alpha$ );
(16)     wait ( $(state_i = passive) \wedge (deficit_i = 0) \wedge (\neg fulfilled_i(arr\_from_i))$ );
(17)      $b_i \leftarrow cont\_passive_i$ ;  $cont\_passive_i \leftarrow true$ ;
(18)     send  $ANSWER(b_i)$  to  $obs_\alpha$ 
(19)  end if.

```

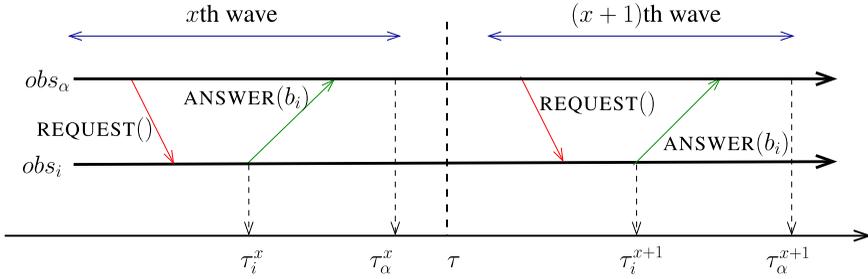
**Fig. 14.16** An algorithm for static termination detection

The cooperation among the local observers is described at lines 8–19. To simplify the presentation, we consider that each wave is implemented by a star which is (a) centered at a process  $obs_\alpha$  (which does not participate in the computation), and (b) implemented by messages  $REQUEST()$  and  $ANSWER()$  (as done in Fig. 14.5).

When  $obs_\alpha$  launches a new wave it sends a message  $REQUEST()$  to each process observer  $obs_i$  (line 10). Then, it waits for a message  $ANSWER(b_i)$  from each of them (line 11), and claims termination if the conjunction  $b_1 \wedge \dots \wedge b_n$  is true (lines 12–14).

When a local observer  $obs_i$  is visited by a new wave (line 15), it waits until its local predicate  $(state_i = passive) \wedge (deficit_i = 0) \wedge (\neg fulfilled_i(arr\_from_i))$  becomes true. When this occurs, it sends the current value of  $cont\_passive_i$  to  $obs_\alpha$  and resets this variable to the value  $true$  (lines 17–18).

**Theorem 23** *Assuming generalized receive statements, the algorithm described in Fig. 14.16 satisfies the safety and liveness properties defining the static termination detection problem.*



**Fig. 14.17** Definition of time instants for the safety of static termination

*Proof* Proof of the liveness property. We have to show that, if the computation  $\mathcal{C}$  is statically terminated, the algorithm eventually claims it.

If the computation is statically terminated at time  $\tau$ , we have  $S\_TERM(\mathcal{C}, \tau)$  (definition). As static termination is a stable property, it follows that all the processes  $p_i$  are continuously passive, their re-activation conditions are not satisfied (i.e.,  $\neg fulfilled_i(arr\_from_i)$  at each  $p_i$ ), and the network component of each channel is empty. Moreover, it follows from the acknowledgment mechanism that there is a time  $\tau' \geq \tau$  such that, from  $\tau'$ , the predicate  $deficit_i = 0$  is always satisfied at each process  $p_i$ .

Let the wave launched by  $obs_\alpha$  after  $\tau'$  be the  $x$ th wave. It follows from the predicate at line 16 that neither the  $x$ th wave, nor any future wave, will be delayed at this line. Moreover, while the value  $b_i$  returned by  $obs_i$  to the  $x$ th wave can be *true* or *false*, the value  $b_i$  returned to the  $(x + 1)$ th wave is necessarily the value *true* (this is because the  $x$ th wave set the local variables  $cont\_passive_i$  to *true*, and none of them has been modified thereafter).

It follows that (if not yet satisfied for the  $x$ th wave) the  $(x + 1)$ th wave will be such that  $b_i = true$  for each  $i$ . The observer  $obs_\alpha$  will consequently claims termination as soon as it has received all the answers for the  $(x + 1)$ th wave.

*Proof of the safety property.* We have to show that, if the algorithm claims termination at time  $\tau'$ , there is a time  $\tau \leq \tau'$  such that  $S\_TERM(\mathcal{C}, \tau)$ .

To that end, let  $\tau$  be a time such that, for each  $i$ , we have  $\tau_i^x < \tau_\alpha^x < \tau < \tau_i^{x+1} < \tau_\alpha^{x+1}$ , where  $\tau_i^x$  is the time at which  $obs_i$  sends its answer and locally terminates the visit of the  $x$ th wave, and  $\tau_\alpha^x$  be the time at which  $obs_\alpha$  terminates its **wait** statement of the  $x$ th wave (line 10). The time instants  $\tau_i^{x+1}$  and  $\tau_\alpha^{x+1}$  are defined similarly for  $(x + 1)$ th wave. These time instants are depicted in Fig. 14.17.

The proof consists in showing that, if the algorithm claims termination at the end of the  $(x + 1)$ th wave (i.e., just after time  $\tau_\alpha^{x+1}$ ), then computation was statically terminated at time  $\tau$  (i.e.,  $S\_TERM(\mathcal{C}, \tau)$  is true). The proof is decomposed into three parts.

- Proof that, for each  $i$ ,  $state_i^\tau = passive$ .

It follows from the management of the local variables  $cont\_passive_i$  (lines 10 and 17) that, if  $obs_\alpha$  claims termination just after time  $\tau_\alpha^{x+1}$ , each process  $p_i$  has

been continuously passive between  $\tau_i^x$  and  $\tau_i^{x+1}$ . As  $\tau_i^x < \tau < \tau_i^{x+1}$ , we conclude that we have  $state_i^\tau = passive$ .

- Proof that, for each  $i$ ,  $NE_i^\tau = \emptyset$ .

Due to the algorithm, each wave is delayed at an observer  $obs_i$  (at line 16, before the send of an answer message) until all messages sent by  $p_i$  have been acknowledged (predicate  $deficit_i = 0$ ). On the other hand, no process  $p_i$  sent messages between  $\tau_i^x$  and  $\tau_i^{x+1}$  (because it was continuously passive during the time interval  $[\tau_i^x, \tau_i^{x+1}]$ , see the previous item). It follows from these two observations that, at time  $\tau$ , there is no message sent by  $p_i$  and not yet arrived at its destination. As this is true for all the processes, it follows that we have  $NE_i^\tau = \emptyset$ .

- Proof that, for each  $i$ ,  $\neg fulfilled_i(arr\_from_i^\tau)$ .

As previously, a wave is delayed at an observer  $obs_i$  until  $\neg fulfilled_i(arr\_from_i)$ . Hence, as the algorithm claims termination just after  $\tau_\alpha^{x+1}$ , it follows that, for any  $i$ , we have  $\neg fulfilled_i(arr\_from_i^{x+1})$ , where  $arr\_from_i^{x+1}$  denotes the value of  $arr\_from_i$  at time  $\tau_i^{x+1}$  (line 16).

As any process  $p_i$  has been continuously passive in the time interval  $[\tau_i^x, \tau_i^{x+1}]$ , it consumed no message during that period. Consequently the set  $arr\_from_i$  can only increase during this period, i.e., we have  $arr\_from_i^\tau \subseteq arr\_from_i^{x+1}$ . It follows from the monotonicity of the predicate  $fulfilled_i()$  that  $(\neg fulfilled_i(arr\_from_i^{x+1}) \Rightarrow (\neg fulfilled_i(arr\_from_i^\tau)))$ . Hence, for any  $i$  we have  $\neg fulfilled_i(arr\_from_i^\tau)$ , which concludes the proof of the theorem.  $\square$

**Cost of the Algorithm** Each application message entails the sending of an acknowledgment. A wave requires two types of messages,  $n$  request messages which carry no value, and  $n$  answers which carry a Boolean. Moreover, after the computation has terminated, one or two waves are necessary to detect termination. Hence,  $4n$  control messages are necessary in the worst case. If, instead of a star, a ring is used to implement waves, this number reduces to  $2n$ .

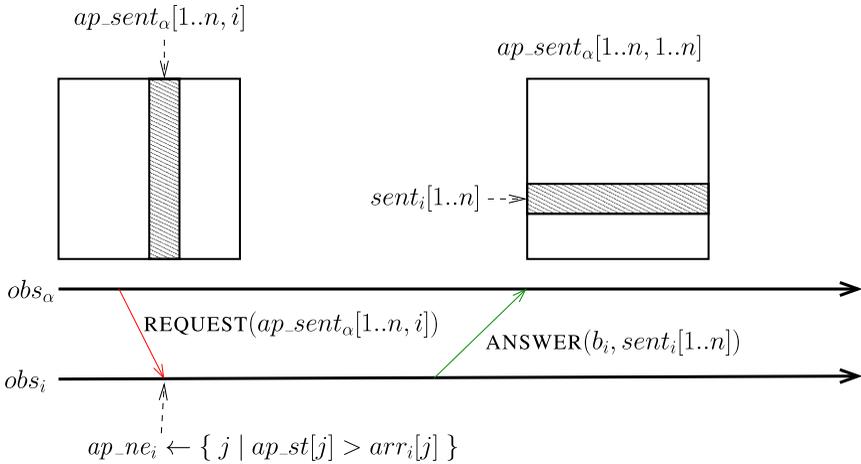
### 14.5.5 Detection of Dynamic Termination

**Local Data Structures** To detect dynamic termination of a distributed computation  $\mathcal{C}$ , i.e., the existence of a time  $\tau$  such that the predicate

$$D\_TERM(\mathcal{C}, \tau) \equiv [\forall i: (state_i^\tau = passive) \wedge (\neg fulfilled_i(NE_i^\tau \cup arr\_from_i^\tau))]$$

is satisfied, for each process  $p_i$ , some knowledge on which messages are currently in transit to  $p_i$  is required. To that end, each observer  $obs_i$  (in addition of  $state_i$  and  $arr\_from_i$ ) and the observer  $obs_\alpha$  are endowed with the following local variables.

- $sent_i[1..n]$  is an array, managed by  $obs_i$ , such that  $sent_i[j]$  contains the number of messages which, since the beginning of the execution, have been sent by  $p_i$  to  $p_j$ .



**Fig. 14.18** Cooperation between local observers

- $arr_i[1..n]$  is a second array, managed by  $obs_i$ , such that  $arr_i[j]$  contains the number of processes which, since the beginning of the execution, have been sent by  $p_j$  and have arrived at  $p_i$ .
- $ap\_sent_\alpha[1..n, 1..n]$  is a matrix, managed by  $obs_\alpha$  such that  $ap\_sent_\alpha[i, j]$  contains the number of messages that, from  $obs_\alpha$ 's point of view, have been sent by  $p_i$  to  $p_j$ . As  $obs_\alpha$  can learn now information only when it receives control messages from the local observers  $obs_i$ ,  $ap\_sent_\alpha[i, j]$  represents an approximate knowledge (hence the identifier prefix  $ap$ ). The way this array is used is described in Fig. 14.18.

**Dynamic Termination Detection Algorithm** The algorithm is described in Fig. 14.19. The local observation part of  $obs_i$  (lines 1–6) is close to that of the static termination detection algorithm. The difference is the fact that the acknowledgment messages are replaced by the counting of messages sent and arrived, on a per-process basis.

The cooperation among  $obs_\alpha$  and the local observers  $obs_i$  is different from that used in static termination detection (Fig. 14.18).

- First, the waves are used to provide  $obs_\alpha$  with the most recent values of the number of messages which have been sent by each process  $p_i$ .  
 More precisely, when a local observer answers a wave (line 19), it sends a control message including the current value of  $sent_i[1..n]$ , so that, when  $obs_\alpha$  receives this message, it can update accordingly the column  $i$  of its local matrix  $ap\_sent_\alpha$  to a more up-to-date value (lines 10–11). In that way,  $obs_\alpha$  is able to learn more recent information on the communication generated by the observed computation.
- The waves are also used by  $obs_\alpha$  to provide each local observer  $obs_i$  with more recent information on the number of messages which have been sent to process  $p_i$  (line 9 on  $obs_\alpha$ 's side, and lines 15–16 on the side of each observer  $obs_i$ ).

```

when  $p_i$  enters a receive statement do
(1) compute  $dep\_set_i$  and  $fulfilled_i()$  from the receive statement;
(2)  $state_i \leftarrow passive$ .

when  $p_i$  executes  $send(m)$  to  $p_j$  do
(3)  $sent_i[j] \leftarrow sent_i[j] + 1$ .

when a message from  $p_j$  arrives do
(4)  $arr_i[j] \leftarrow arr_i[j] + 1$ .

when  $fulfilled_i(arr\_from_i)$  becomes satisfied do
(5)  $state_i \leftarrow active$ ;
(6)  $cont\_passive_i \leftarrow false$ .

local task at  $p_i$  for the cooperation between observers is
(7) if ( $i = \alpha$ )
(8)   then repeat
(9)     for each  $j \in \{1, \dots, n\}$  do  $send\ REQUEST(ap\_sent_\alpha[1..n, j])$  to  $p_j$  end for;
(10)    wait ( $ANSWER(b_j, st_j[1..n])$  received from each  $obs_j, j \in \{1, \dots, n\}$ );
(11)    for each  $j \in \{1, \dots, n\}$  do  $ap\_sent_\alpha[1..n, j] \leftarrow st_j[1..n]$  end for;
(12)     $res \leftarrow \bigwedge_{1 \leq j \leq n} b_j$ 
(13)    until ( $res$ ) end repeat;
(14)    claim termination
(15)  else wait (message  $REQUEST(ap\_st[1..n])$  from  $obs_\alpha$ );
(16)     $ap\_ne_i \leftarrow \{j \mid ap\_st[j] > arr_i[j]\}$ ;
(17)     $b_i \leftarrow cont\_passive_i \wedge (\neg fulfilled_i(arr\_from_i \cup ap\_ne_i))$ ;
(18)     $cont\_passive_i \leftarrow (state_i = passive)$ ;
(19)    send  $ANSWER(b_i, sent_i[1..n])$  to  $obs_\alpha$ 
(20) end if.

```

**Fig. 14.19** An algorithm for dynamic termination detection

- As a local observer  $obs_i$  does not know the value of  $NE_i$ , it cannot compute the value of the predicate  $\neg fulfilled_i(NE_i \cup arr\_from_i)$  (which characterizes dynamic termination). To cope with this issue,  $obs_i$  computes the set  $ap\_ne_i = \{j \mid ap\_st[j] > arr_i[j]\}$  (line 16), which is an approximation of  $NE_i$  (maybe more messages have been sent to  $p_i$  than the ones whose sending is recorded in the array  $ap\_st[1..n]$  sent by  $obs_\alpha$ ).

The observer  $obs_i$  can now compute the value of the local predicate  $\neg fulfilled_i(ap\_ne_i \cup arr\_from_i)$  (line 17). Then,  $obs_i$  sends to  $obs_\alpha$  (line 19) a Boolean  $b_i$  which is true (line 17) if and only if (a)  $p_i$  has been continuously passive since the previous wave (as in static detection termination), and (b) the predicate  $fulfilled_i(ap\_ne_i \cup arr\_from_i)$  is false (i.e.,  $p_i$  cannot be reactivated with the messages that have arrived and are not yet consumed, and a subset of the messages which are potentially in transit to it).

Moreover, after  $b_i$  has been computed,  $obs_i$  resets its local variable  $cont\_passive_i$  to the Boolean value  $(state_i = passive)$ . This is due to the fact that, differently from the previous termination detection algorithms, the waves are not

delayed by the local observers in dynamic termination detection. This is required to ensure “as early as possible” termination detection.

Finally, when there is a wave such that the Boolean value

$$\bigwedge_{1 \leq i \leq n} (\neg \text{fulfilled}_i(\text{arr\_from}_i \cup \text{ap\_ne}_i))$$

is true,  $\text{obs}_\alpha$  claims dynamic termination. If the value is false, it starts a new wave.

The proof of this algorithm is close to that of static termination. It uses both the monotonicity of the predicate  $\text{fulfilled}_i()$  and the monotonicity of the counters (see Problem 6).

**Cost of the Algorithm** After dynamic termination has occurred, two waves are necessary to detect it, in the worst case. No acknowledgment is used, but a wave requires  $2n$  messages, each carrying an array of unbounded integers (freezing the observed computation would allow us to reset the counters). The algorithm does not require the channels to be FIFO (neither for application, nor for control messages). But as waves are sequential, the channels from  $\text{obs}_\alpha$  to each  $\text{obs}_i$ , and the channels from every  $\text{obs}_i$  to  $\text{obs}_\alpha$ , behave as FIFO channels for the control messages  $\text{REQUEST}()$  and control messages  $\text{ANSWER}()$ , respectively. It follows that, instead of transmitting the value of a counter, it is possible to transmit only the difference between its current value and the sum of the differences already sent.

## 14.6 Summary

A distributed computation has terminated when all processes are passive and all channels are empty. This defines a stable property (once terminated, a computation remains terminated forever) which has to be detected, in order that the system can reallocate local and global resources used by the processes (e.g., local memory space).

This chapter presented several distributed algorithms which detect the termination of a distributed computation. Algorithms suited to specific models (such as the asynchronous atomic model), and algorithms suited to specific types of computations (such as diffusing computations) were first presented. Then the chapter has considered more general algorithms. In this context it presented a reasoned construction of a very general termination detection algorithm. It also introduced a very general distributed model which allows for very versatile receive statements, and presented two termination detection algorithms suited to this distributed computation model.

## 14.7 Bibliographic Notes

- The distributed termination detection problem was simultaneously introduced by E.W. Dijkstra and C.S. Scholten [115], and N. Francez [130].

- The asynchronous atomic model was introduced by F. Mattern in [249], who proposed the four-counter algorithm (Fig. 14.5), and the counting vector algorithm (Fig. 14.7) to detect termination in this distributed computing model.
- The notion of diffusing computations was introduced by E.W. Dijkstra and C.S. Scholten [115], who proposed the algorithm described in Fig. 14.9 to detect the termination of such computations.
- The reasoned construction presented in Sect. 14.4, and the associated general termination detection algorithm presented in Fig. 14.13 are due to J.-M. Hélarý and M. Raynal [178, 180].
- The wave concept is investigated in [82, 180, 319, 365].
- The notion of freezing in termination detection algorithms is studied in [132].
- The problem of detecting the termination of a distributed computation in a very general asynchronous model, and the associated detection algorithms, are due to J. Brzezinski, J.-M. Hélarý, and M. Raynal [64].
- The termination detection problem has given rise to an abundant literature and many algorithms. Some are designed for the synchronous communication model (e.g., [114, 265]); some are based on snapshots [190]; some are based on roughly synchronized clocks [257]; some are based on prime numbers [309] (where the unicity of the prime number factorization of an integer is used to ensure consistent observations of a global state); some are obtained from a methodological construction (e.g., [79, 342]); and some are based on the notion of credit distribution [251]. Other investigations of the termination detection problem and algorithms can be found in [74, 169, 191, 221, 252, 261, 302, 330] to cite a few.
- Termination detection in asynchronous systems where processes may crash is addressed in [168].
- Relations (in both directions) between termination detection and garbage collection are investigated in [366, 367].

## 14.8 Exercises and Problems

1. Let us consider the counting vector algorithm described in Fig. 14.7.
  - Why does the termination predicate of line 10 need to include the fact that each process has been visited at least once?
  - When the local observer at process  $p_i$  receives the token and is such that  $msg\_count[i] > 0$ , it knows that messages sent to  $p_i$  have not yet arrived. Modify the algorithm so that the progression of the token is blocked at  $p_i$  until  $msg\_count[i] = 0$ .
2. Let us assume that the underlying system satisfies the following additional synchrony property: the transmission delay of each message is upper bounded by a constant  $\delta$ . Moreover, each process has a local clock that allows it to measure durations. These clocks are not synchronized but measure correctly the duration  $\delta$ .  
 Modify the termination detection algorithm suited to diffusing computations described in Fig. 14.9 so that it benefits from this synchrony assumption.

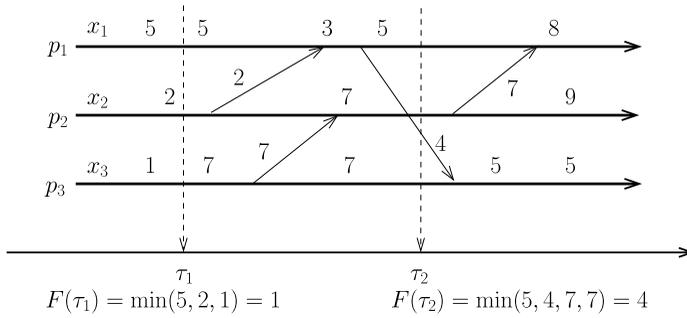


Fig. 14.20 Example of a monotonous distributed computation

Hint. When a process enters the tree, it sends a message  $ACK(in)$  to its parent and send no message if it is already in the tree. It will send to its parent a message  $ACK(out)$  when it leaves the tree.

Solution in [308].

3. Modify the algorithms implementing a wave and the generic termination detection algorithm described in Figs. 14.10, 14.11, and 14.13, respectively, so that the initiator  $p_\alpha$  is also an application process (which has consequently to be observed).
4. The termination detection algorithm presented in Fig. 14.13 assumes that the application process  $p_i$  is frozen (Fig. 14.14) from the time the predicate  $idle_i$  becomes true (line 13) until  $obs_i$  starts the invocation of  $return\_wave(b)$  (line 15).

Considering the  $x$ th wave, let us define  $\tau_i^x$  as the time at which  $idle_i$  becomes true at line 13, and let us replace line 14 by

$$b \leftarrow cont\_passive_i; \quad cont\_passive_i \leftarrow (state_i = passive).$$

Do these modifications allow for the suppression of the freezing of the application process  $p_i$ ?

5. Let us consider the family of distributed applications in which each process has a local variable  $x_i$  whose value domain is a (finite or infinite) ordered set, and each process follows the following rules:
  - Each message  $m$  sent by  $p_i$  carries a value  $c_m \geq x_i$ .
  - When it modifies the value of  $x_i$ , a process  $p_i$  can only increase it.
  - When it receives a message  $m$ , which carries the value  $c_m$ ,  $p_i$  can modify  $x_i$  to a value equal to or greater than  $\min(x_i, c_m)$ .

These computations are called *monotonous computations*. An example of a computation following these rules is given in Fig. 14.20, where the value domain is the set of positive integers. The value  $c_m$  associated with a message  $m$  is indicated with the corresponding arrow, and a value  $v$  on the axis of a process  $p_i$  indicates that its local variable  $x_i$  is updated to that value, e.g.,  $x_3 = 1$  a time  $\tau_1$ , and  $x_1 = 5$  at time  $\tau_2$ .

At some time  $\tau$ , the values of the local variables  $x_i$  and the values  $c$  carried by the messages define the current global state of the application. Let  $F(\tau)$  be the smallest of these values. Two examples are given in Fig. 14.20.

Distributed simulation programs are an example of programs where the processes follow the previous rules (introductory surveys on distributed simulation can be found in [140, 263]). The variables  $x_i$  are the local simulation times at each process, and the function  $F(\tau)$  defines the global simulation time.

- Show that  $(\tau_1 \leq \tau_2) \Rightarrow [F(\tau_1) \leq F(\tau_2)]$ .
- An observation algorithm for such a computation is an algorithm that computes approximations of the value of  $F(\tau)$  such that
  - Safety. If at time  $\tau$  the algorithm returns the value  $z$ , then  $z \leq F(\tau)$ .
  - Liveness. For any  $\tau$ , if observations are launched after  $\tau$ , there is a time after which all the observations return values greater than or equal to  $z = F(\tau)$ .

When considering distributed simulation programs, the safety property states a correct lower bound on the global simulation time, while the liveness property states that if the global simulation progresses then this progress can eventually be observed.

Adapt the algorithm of Fig. 14.13 so that it computes  $F(\tau)$  and satisfies the previous safety and liveness properties.

- Considering the value domain  $\{active, passive\}$  with the total order  $active < passive$ , show that the problem of detecting the termination of a distributed computation consists in repeatedly computing  $F(\tau)$  until  $F(\tau) = passive$ .

Solutions to all the previous questions in [87, 179].

6. Prove the dynamic termination detection algorithm described in Fig. 14.19. (This proof is close to that of the static termination detection algorithm presented in Fig. 14.16. In addition to the monotonicity of the predicate  $fulfilled_i()$ , the fact that the counters  $sent_i[j]$ ,  $rec_i[j]$  and  $ap\_sent_\alpha[i, j]$  are monotonically increasing has to be used.)

Solution in [64].

# Chapter 15

## Distributed Deadlock Detection

This chapter addresses the deadlock detection problem. After having introduced the AND deadlock model and the OR deadlock model, it presents distributed algorithms that detect their occurrence. Let us recall that the property “there is a deadlock” is a stable property (once deadlocked, a set of processes remain deadlocked until an external agent—the underlying system—resolves it). Hence, as seen in Sect. 6.5, algorithms computing global states of a computation can be used to detect deadlocks. Differently, the algorithms presented in this chapter are specific to deadlock detection. For simplicity, they all assume FIFO channels.

**Keywords** AND communication model · Cycle · Deadlock · Deadlock detection · Knot · One-at-a-time model · OR communication model · Probe-based algorithm · Resource vs. message · Stable property · Wait-for graph

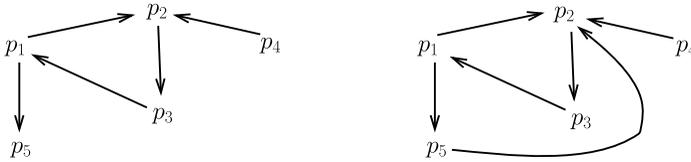
### 15.1 The Deadlock Detection Problem

#### 15.1.1 Wait-For Graph (WFG)

**Waiting for Resources** A process  $p_i$  becomes blocked when it starts waiting for a resource currently owned by another process  $p_j$ . This introduces a waiting relation between  $p_i$  and  $p_j$ . It is possible that a process  $p_i$  needs several resources simultaneously. If these resources are currently owned by several processes  $p_j, p_k, \text{etc.}$ , the progress of  $p_i$  depends on each of these processes.

**Waiting for Messages** As seen in the previous chapter (Sect. 14.5), a receive statement can be on a message from a specific sender, or on a message from any sender of a specific set of senders. The specific sender, or set of possible senders, is defined in the receive statement. (While more general receive statements were introduced in Sect. 14.5, this chapter considers only the case of a receive statement with a specific sender or a specific set of senders.)

The important point is that, when a process  $p_i$  enters a receive statement, it starts waiting for a message from a predefined process  $p_j$ , or from any process from a predefined set of processes.



**Fig. 15.1** Examples of wait-for graphs

**Resource vs. Message** After it has been used, a resource has to be released so that it can be used by another process. Let us observe that a message can be seen as a resource that is dynamically created by its sender and consumed by its destination processes. A receiver has to wait until a message is received. In that sense, a message can be seen as a consumable resource (the notion of release being then meaningless).

While a resource is shared among several processes and a message involves only its sender and its receiver, there is no conceptual difference between a process which is waiting for a resource and a process which is waiting for a message. In both cases, another process has to produce an action (release a resource, or send a message) in order that the waiting process be allowed to progress.

It follows that there is no fundamental difference between detection algorithms for deadlocks due to resources and detection algorithms for deadlocks due to messages.

**Wait-For Graph** The waiting relation between processes can be represented by a directed graph called a *wait-for* graph (WFG). Its vertices are the processes, and an edge from  $p_i$  to  $p_j$  means that  $p_i$  is blocked by  $p_j$  (or, equivalently, that  $p_j$  blocks  $p_i$ ). This graph is not a static graph. Its structure evolves according to the behavior of the processes (that acquire and release resources, or send and receive messages).

A wait-for graph is a conceptual tool that allows us to capture blocking relations and reason on them. Deadlock detection algorithms are not required to explicitly build this graph.

Two examples of a wait-for graph are depicted in Fig. 15.1. (The graph on the right side is the same as the graph on the left side with an additional edge from  $p_5$  to  $p_2$ .) A given graph can be considered as capturing the blocking relations either in the resource model or in the communication model.

The arrow from  $p_3$  to  $p_1$  means that  $p_3$  is blocked by  $p_1$ . If the graph is a resource wait-for graph,  $p_3$  is waiting for a resource owned by  $p_1$ , and if the graph is a communication wait-for graph, it means that  $p_3$  is waiting for a message from  $p_1$ . A process can be blocked by several other processes. This is the case of  $p_1$ , which is blocked by  $p_2$  and  $p_5$ . The meaning of the corresponding edges is explained below in Sect. 15.1.2.

**Dependency Set** Given a wait-for graph at some time  $\tau$ , the *dependency set* of a process  $p_i$ , denoted  $dep\_set_i$  (as in the previous chapter), is the set of all the processes  $p_j$  such that the directed edge  $(p_i, p_j)$  belongs to the graph at time  $\tau$ . If

a process is not waiting for a resource (or a message, according to the model), its dependency set is empty. As the wait-for graph evolves with time, the dependency sets are not static sets.

When looking at Fig. 15.1, we have  $dep\_set_1 = \{2, 5\}$  in both graphs,  $dep\_set_5 = \emptyset$  in the graph on the left and  $dep\_set_5 = \{2\}$  in the graph on the right.

### 15.1.2 AND and OR Models Associated with Deadlock

**AND (Resource or Communication) Model** In the AND waiting model a process  $p_i$  which is blocked will be allowed to progress only when

- AND communication model:  
It has received a message from each process that belongs to its current dependency set.
- AND resource model:  
It has obtained the resources which are currently owned by the processes in its dependency set.

Let us observe that, while it is waiting for resources, the dependency set of a process  $p_i$  might change with time. This is due to the fact that, while  $p_i$  is waiting, resources that  $p_i$  wants to acquire can be granted to other processes. The wait-for graph and the dependency set of  $p_i$  are then modified accordingly.

**OR (Resource or Communication) Model** In the OR waiting model, a process is allowed to progress when

- OR communication model:  
It has received a message from a process in its dependency set.
- OR resource model:  
It has obtained a resource from a process in its dependency set.

**One-at-a-Time (Resource or Communication) Model** The *one-at-a-time* model captures the case where at any time a process is allowed to wait for at most one resource (resource model), or one message from a predetermined process specified in the receive statement (communication model).

Let us observe that both the AND model and the OR model include the one-at-a-time model as their simplest instance. It is actually their intersection.

### 15.1.3 Deadlock in the AND Model

In the AND model, the progress of a process is stopped until it has received a message (or been granted a resource) by each process in its current dependency set. This means that a process is deadlocked when it belongs to a cycle of the current wait-for graph, or belongs to a path leading to a cycle of this graph.

Let us consider the wait-for graph on the left of Fig. 15.1. The processes  $p_1$ ,  $p_2$ , and  $p_3$  are deadlocked because, transitively, each of them depends on itself:  $p_2$  is

blocked by  $p_3$ , which is blocked by  $p_1$ , which in turn is blocked  $p_2$ , hence each of  $p_3$ ,  $p_1$ , and  $p_2$  is transitively blocked by itself. Process  $p_4$  is blocked by  $p_2$  (which is deadlocked) but, as it does not depend transitively on itself, it is not deadlocked according to the definition of “deadlocked process” (killing  $p_4$  will not suppress the deadlock, while killing any of  $p_3$ ,  $p_1$ , or  $p_2$ , suppresses the deadlock). Finally,  $p_5$  and the other processes (if any) are not blocked.

Hence, when considering the wait-for graph in the AND model, the occurrence of a deadlock is characterized by the presence of a cycle. Consequently, for an external observer that would always have an instantaneous view of the current wait-for graph, detecting a deadlock in the AND model, would consist in detecting a cycle in this graph.

### ***15.1.4 Deadlock in the OR Model***

In the OR model, a process is re-activated as soon as it receives a message from (or is granted a resource previously owned by) one of the processes in its dependency set. It follows that the existence of a cycle no longer reveals a deadlock.

Let us consider again the process  $p_1$  in the wait-for graph on the left of Fig. 15.1. As it is waiting for a message (resource) either from  $p_2$  or from  $p_5$ , and  $p_5$  is not blocked, it is possible that  $p_5$  (before terminating its local computation), unblocks it in the future. Hence, despite the presence of a cycle in the wait-for graph, there is no deadlock. Let us now consider the wait-for graph on the right of the figure. Each of  $p_1$ ,  $p_2$ ,  $p_3$ , and  $p_5$ , is blocked, and none of them can allow another one to progress in the future. This is because the progress of any of these processes depends on the progress of the others (moreover,  $p_4$  is also deadlocked by transitivity). As we can see, such a graph structure is a knot. Let us recall (see Sect. 2.3) that a knot in a directed graph is a set  $S$  of vertices such that (a) there is a directed path from any vertex of  $S$  to any other vertex of  $S$ , and (b) there is no outgoing edge from a vertex in  $S$  to a vertex which is not in  $S$  (see Fig. 2.14).

Hence, when considering the wait-for graph in the OR model, the occurrence of a deadlock is characterized by the presence of a knot. Consequently, for an external observer that would always have an instantaneous view of the current wait-for graph, detecting a deadlock in the OR model would consist in detecting a knot in this graph.

### ***15.1.5 The Deadlock Detection Problem***

As for the other problems (e.g., resource allocation, communication, termination detection), the deadlock detection problem is defined by safety and liveness properties that any of its solutions has to satisfy. These properties are the following:

- **Safety.** If, at time  $\tau$ , an observer claims that there is a deadlock, the process it is associated with is deadlocked at time  $\tau$ .

- **Liveness.** If a deadlock occurs at time  $\tau$ , there is a finite time  $\tau' \geq \tau$  such that, at time  $\tau'$ , the observer of at least one of the deadlocked processes claims that its associated process is deadlocked.

The safety property states that the detection is consistent if there is no “false” deadlock detection (i.e., claim of a deadlock while there is none). The liveness property states that a deadlock will be discovered by at least one observer associated with a deadlocked process.

**Remark** When comparing deadlock detection algorithms with algorithms that allow a process to know if it belongs to a cycle, or a knot, of a given communication graph (see the algorithms presented in Sect. 2.3), the additional difficulty lies in the fact that the graphs explored by deadlock detection algorithms consider are dynamic. The waiting relation depends on the computation itself, and consequently the wait-for graph evolves with time.

Moreover, while the abstract wait-for graph is modified instantaneously for an external observer’s point of view (i.e., when reasoning to obtain a characterization of deadlocks in terms of properties on a graph), at the operational level messages take time to inform processes about changes in the waiting relation. This creates an uncertainty on the view of the waiting relation as perceived by each process. The main issue of deadlock detection algorithms is to provide processes with an approximate view that allows them to correctly detect deadlocks that occur.

### 15.1.6 Structure of Deadlock Detection Algorithms

To detect the possible occurrence of deadlocks, an observer  $obs_i$  is associated with each application process  $p_i$ . As in termination detection, each observer has to locally observe the behavior of the process it is associated with, and observers have to cooperate among themselves to detect deadlocks. The global structure is the same as that described in Fig. 14.2.

A main difference between termination detection and deadlock detection lies in the fact that termination detection involves all the processes (and consequently all the observers have to permanently cooperate), while deadlock detection involves only a set of processes which is not known in advance (and consequently only the observers associated with blocked processes are required to cooperate).

## 15.2 Deadlock Detection in the One-at-a-Time Model

This section presents a simple algorithm that detects deadlock in the one-at-a-time model. Let us recall that, in this model, a process becomes blocked because it is waiting for a resource currently owned by a process  $p_k$  (resource model), or because it is waiting for a message from a predetermined process  $p_k$  (communication model). This algorithm is due to D.P. Mitchell and M.J. Merritt (1984).

### 15.2.1 Principle and Local Variables

**Principle** As we have seen, a deadlock in the one-at-a-time model corresponds to a cycle in the wait-for graph. The idea of the algorithm is to allow exactly one process in a cycle to detect the corresponding deadlock (if needed this process can then inform the other processes).

To that end, when a process observer suspects a deadlock, it suspects a cycle has formed in the wait-for graph. It then launches an election on this “assumed” cycle. (The election problem and election algorithms have been presented in Chap. 4.) If there is a cycle, one process in the cycle will win the election, proving thereby a deadlock occurrence.

The control messages used by the observers to discover a deadlock occurrence are sent along the edges of the wait-for graph but in the reverse order, i.e., if a process  $p_i$  is blocked by a process  $p_j$ , the observer  $obs_j$  sends messages to the observer  $obs_i$ .

**Local Variables** Two local control variables, denoted  $pub_i$  (for public) and  $priv_i$  (for private) are associated with each process  $p_i$ . These variables are integers which can only increase. Moreover, the values of the private local variables are always distinct from one another, and the local public variables can be read by the other processes.

A simple way to ensure that no two private variables ( $priv_i$  and  $priv_j$ ) have the same value consists in considering that the integer implementing a private variable  $priv_i$  is obtained from the concatenation of an integer with the identity of the corresponding process  $p_i$ . The reading of the local variable  $pub_j$  by an observer  $obs_i$  can be easily realized by a query/response mechanism involving  $obs_i$  and  $obs_j$ .

Hence, the behavior of each observer  $obs_i$  consists in an appropriate management of the pair  $(priv_i, pub_i)$  so that, if a cycle appears in the abstract wait-for graph, a single process in a cycle detects the cycle, and the processes which are not involved in a cycle never claim a deadlock.

### 15.2.2 A Detection Algorithm

When a process  $p_i$  becomes blocked because of a process  $p_j$ , its local observer  $obs_i$  (a), redefines the pair of its local variables  $(priv_i, pub_i)$  so that both variables become equal ( $priv_i = pub_i$ ) and greater than  $pub_j$ , and (b) sends its new value  $pub_i$  to the observers of the processes blocked by  $p_i$ .

The value  $pub_i$  is then propagated from observer to observer along the reverse edges of the wait-for graph. If  $pub_i$  returns to  $obs_i$ , there is a cycle in the wait-for graph. In order that a single observer on a cycle detects the cycle, the greatest value  $pub_i$  that is propagated along a cycle succeeds in completing a full turn on the cycle (as done in ring-based election algorithms).

The behavior of an observer  $obs_i$  is consequently defined by the four following rules, where  $\text{greater}(a, b)$  returns a value that is greater than both  $a$  and  $b$ .

- R1 (Blocking rule).

When  $p_i$  becomes blocked due to  $p_j$ ,  $obs_i$  resets the values of its local variables  $priv_i$  and  $pub_i$  such that

$$priv_i = pub_i = v, \quad \text{where } v = \text{greater}(pub_i, pub_j).$$

- R2 (Propagation rule).

When  $p_i$  is blocked by  $p_j$ ,  $obs_i$  repeatedly reads the value of  $pub_j$ , and executes

**if** ( $pub_i < pub_j$ ) **then**  $pub_i \leftarrow pub_j$  **end if.**

This means that, while  $p_i$  is blocked by  $p_j$ ,  $obs_i$  discovers that the public value  $pub_j$  is greater than its own public value  $pub_i$ , it propagates the value of  $pub_j$  by assigning it to  $pub_i$ . Assuming that  $pub_k$  is the greatest public value in a path of the wait-for graph, this rule ensures that  $pub_k$  is propagated from  $p_k$  to the process  $p_\ell$  blocked by  $p_k$ , then from  $p_\ell$  to the process blocked by  $p_\ell$ , etc.

- R3 (Activation rule).

Let  $p_j$ ,  $p_k$ , etc., be the processes blocked by a process  $p_i$ . When  $p_i$  unblocks one of them (e.g.,  $p_j$ ),  $obs_i$  informs the observers of the other processes so that they re-execute rule R1. (This is due to the fact that these processes are now blocked by  $p_j$ .)

- R4 (Detect rule).

If after it has read the value of  $pub_j$  (where  $p_j$  is the process that blocks  $p_i$ ),  $obs_i$  discovers that

$$priv_i = pub_j,$$

it claims that there is a cycle in the wait-for graph, and it is consequently involved in a deadlock.

### 15.2.3 Proof of the Algorithm

**Theorem 24** *If a deadlock occurs, eventually a process involved in the associated cycle detects it. Moreover, no observer claims a deadlock if its associated process is not deadlocked.*

*Proof* Proof of the liveness property. We have to prove that, if a deadlock occurs, a process involved in the associated cycle will detect it. Let us assume that there is a cycle in the wait-for graph. Let us observe that, as a deadlock defines a stable property, this cycle lasts forever.

It follows from rule R1 that, when it becomes blocked, each process  $p_x$  belonging to this cycle sets  $priv_x$  and  $pub_x$  to a value greater than its previous value of  $pub_x$  and greater than the value  $pub_y$  of the process  $p_y$  that blocks it. Moreover, due to the definition of private values, no other process  $p_z$  sets  $priv_z$  and  $pub_z$  to the same

value as  $p_x$ . It follows that there is a process (say  $p_i$ ) whose value  $pub_i = v$  is the greatest among the values  $pub_x$  computed by the processes belonging to the cycle.

When the processes of the cycle execute rule R2, the value of  $v$  is propagated, in the opposite direction, along the edges of the wait-for graph. Hence, the process  $p_j$  that blocks  $p_i$  is eventually such that  $pub_j = v$ . There is consequently a finite time after which  $p_i$  reads  $v$  from  $pub_j$  and concludes that there is a cycle (rule R4). Moreover, as  $v = priv_i$ ,  $v$  is a value that has necessarily been forged by  $p_i$  (no other process can have forged it). It follows that deadlock is claimed by a single process, which is a process involved in the corresponding cycle of the wait-for graph.

Proof of the safety property. Let us first observe that  $\forall x : priv_x \leq pub_x$  (this is initially true, and is kept invariant by the rules R1 and R2 executed thereafter). It follows from this invariant and R1 that, if  $p_x$  is blocked,  $(priv_x < pub_x) \Rightarrow p_x$  has executed R2.

Let us assume that a process  $p_i$ , blocked by process  $p_{j_1}$ , is such that  $priv_i = pub_{j_1} = v$ . We have to show that  $p_i$  belongs to a cycle of the wait-for graph. This is proved in three steps.

- It follows from the rule R2 that the value  $v$  (which has been forged by  $obs_i$ ) has been propagated from  $p_i$  to a process blocked by  $p_i$ , etc., until  $p_{j_1}$ . Hence, there is a set of processes  $p_i, p_{j_k}, p_{j_{k-1}}, \dots, p_{j_1}$ , such that the edges  $(p_{j_k}, p_i), (p_{j_{k-1}}, p_{j_k}), \dots, (p_{j_1}, p_i)$  have been, at some time, edges of the wait-for graph. The next two items show that all these edges exist simultaneously when  $p_i$  claims deadlock.
- The process  $p_i$  remained continuously blocked since the time it computed the value  $v$ . If it had become active and then blocked again, it would have executed again R1, and (due to the invariant  $priv_i \leq pub_i$  and the function  $greater()$ ) we would consequently have  $priv_i = v' > v$ .
- All the other process  $p_{j_k}, p_{j_{k-1}}, \dots, p_{j_1}$  remained continuously blocked since the time they have forwarded  $v$ . This follows from the following observation. Let us assume (by contradiction) that one of these processes, say  $p_{j_y}$ , became active after having transmitted  $v$ . This process has been unblocked by  $p_{j_{y-1}}$ , which has transmitted  $v$  before being unblocked by  $p_{j_{y-2}}$ , which in turn, etc., until  $p_{j_1}$  which has transmitted  $v$  to  $p_i$  before being unblocked by  $p_i$ . It follows that  $p_i$  has not been continuously passive since the time it computed the value  $v$ , which contradicts the previous item, and completes the proof. □

### 15.3 Deadlock Detection in the AND Communication Model

As already indicated, in the AND model, a process is blocked by several other processes and each of them has to release a resource or send it a message in order to allow it to progress. This section presents a relatively simple deadlock detection algorithm for the communication AND model.

### 15.3.1 Model and Principle of the Algorithm

**Model with Input Buffers** As in the previous chapter, let  $state_i$  be a control variable whose value domain is  $\{active, passive\}$ ; this variable is such that  $state_i = passive$  when  $p_i$  is blocked waiting for messages from a predefined set of processes whose identities define the set  $dep\_set_i$ .

As a process  $p_i$  consumes simultaneously a message from each process in  $dep\_set_i$  (and proceeds then to the state  $state_i = active$ ), it is possible that messages from processes in  $dep\_set_i$  have arrived and cannot be received and consumed by  $p_i$  because there is not yet a message from each process in  $dep\_set_i$ . To take this into account, the communication model described in Fig. 14.15 is considered. This model allows a process  $p_i$  (or more precisely its observer  $obs_i$ ) to look into its input buffers to know if messages have arrived and are ready to be consumed.

**Principle of the Algorithm** Let  $p_i$  be a process that is blocked (hence,  $state_i = passive$ ), and  $arr\_from_i$  be a set containing the identities of the processes from which messages have arrived at  $p_i$  and have not yet been consumed. This means that these messages can help unblock  $p_i$ .

When it suspects that  $p_i$  is deadlocked, its observer  $obs_i$  sends a control message PROBE() to each process  $p_j$  in  $dep\_set_i \setminus arr\_from_i$ . When such a process  $p_j$  receives this message, it discards the message if it is active. If it is passive, its observer  $obs_j$  forwards the message PROBE() to the observers  $obs_k$  such that  $k \in dep\_set_j \setminus arr\_from_j$ , and so on. If the observer  $obs_i$  that initiated the detection receives a message PROBE(), there is a cycle in the wait-for graph, and  $p_i$  belongs to a set of deadlocked processes.

### 15.3.2 A Detection Algorithm

**Local Variables at a Process  $p_i$**  In addition to the control variables  $state_i$ ,  $dep\_set_i$ , and  $arr\_from_i$  (which is managed as in Sect. 14.5), a local observer manages the three following arrays, all initialized to  $[0, \dots, 0]$ :

- $sn_i[1..n]$  is an array of sequence numbers;  $sn_i[i]$  is used by  $obs_i$  to identify its successive messages PROBE(), while  $sn_i[k]$  contains the highest sequence number received by  $obs_i$  in a message PROBE() whose sending has been initiated by  $p_k$ .
- $sent_i[1..n]$  is an array of integers such that  $sent_i[j]$  counts the number of messages sent by  $p_i$  to  $p_j$ .
- $arr_i[1..n]$  is an array of integers such that  $arr_i[j]$  counts the number of messages sent by  $p_j$  that have arrived to  $p_i$  (it is possible that, while they have arrived at  $p_i$ , some of these messages have not yet been consumed by  $p_i$ ).

```

when  $p_i$  enters a receive statement do
(1) compute  $dep\_set_i$  from the receive statement;
(2)  $state_i \leftarrow passive$ .

when  $p_i$  executes  $send(m)$  to  $p_j$  do
(3)  $sent_i[j] \leftarrow sent_i[j] + 1$ .

when a message from  $p_j$  arrives do
(4)  $arr_i[j] \leftarrow arr_i[j] + 1$ .

when  $arr\_from_i \subseteq dep\_set_i$  do
(5) messages are withdrawn from their input buffers and given to  $p_i$ ;
(6)  $state_i \leftarrow active$ .

when  $obs_i$  suspects that  $p_i$  is deadlocked do
    % we have then  $state_i = passive$  and  $arr\_from_i \not\subseteq dep\_set_i$  %
(7)  $sn_i[i] \leftarrow sn_i[i] + 1$ ;
(8) for each  $j \in dep\_set_i \setminus arr\_from_i$ 
(9)     do send  $PROBE(i, sn_i[i], i, arr_i[j])$  to  $obs_j$ 
(10)  end for.

when  $PROBE(k, seqnb, j, arrived)$  is received from  $obs_j$  do
(11) if  $(state_i = passive) \wedge (sent_i[j] = arrived)$  then
(12)   if  $(k = i)$  then claim  $p_i$  is deadlocked
(13)   else if  $(seqnb > sn_i[k])$  then
(14)      $sn_i[k] \leftarrow seqnb$ ;
(15)     for each  $\ell \in dep\_set_i \setminus arr\_from_i$ 
(16)       do send  $PROBE(k, seqnb, i, arr_i[\ell])$  to  $obs_\ell$ 
(17)     end for
(18)   end if
(19) end if
(20) end if.

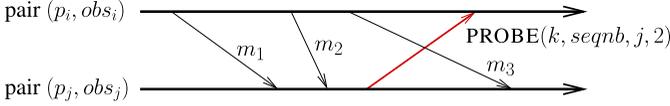
```

**Fig. 15.2** An algorithm for deadlock detection in the AND communication model

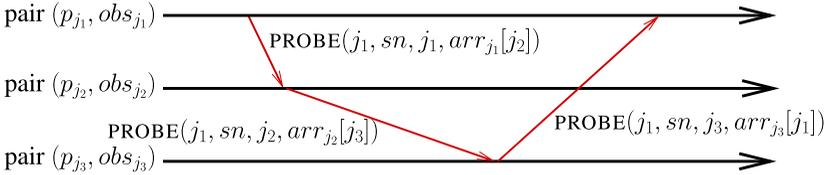
**The Algorithm** The algorithm is described in Fig. 15.2. All the sequences of statements prefixed by **when** are executed atomically. Lines 1–6 describe the behavior of  $obs_i$  as far as the observation of  $p_i$  is concerned.

When  $obs_i$  suspects  $p_i$  to be deadlocked, we have  $state_i = passive$  and  $arr\_from_i \not\subseteq dep\_set_i$ . This suspicion can be activated by an internal timer, or any predicate internal to  $obs_i$ . When this occurs,  $obs_i$  increases  $sn_i[i]$  (line 7), and sends a message  $PROBE()$  to each observer  $obs_j$  associated with a process  $p_j$  from which it is waiting for a message (lines 8–10).

A probe message is as follows:  $PROBE(k, seqnb, j, arrived)$ . Let us consider an observer  $obs_i$  that receives such a message. The pair  $(k, seqnb)$  means that this probe has been initiated by  $obs_k$  and it is its the  $seqnb$ th probe launched by  $p_k$ ;  $j$  is the identity of the sender of the message (this parameter could be saved, as a receiver knows which observer sent this message; it appears as a message parameter for clarity). Finally, the integer  $arrived$  is the number of messages that have been sent by  $p_i$  to  $p_j$  and have arrived at  $p_j$ .



**Fig. 15.3** Determining in-transit messages



**Fig. 15.4** PROBE () messages sent along a cycle (with no application messages in transit)

When  $obs_i$  receives a message  $PROBE(k, seqnb, j, arrived)$ , it discards it (line 11), if  $p_i$  is active or there were messages in transit from  $p_i$  to  $p_j$  when  $obs_i$  sent this probe message. This is captured by the predicate  $(send_i[j] \neq arrived)$  (let us recall that  $arrived = arr_j[i]$  when  $obs_j$  sent this message, and channels are assumed to be FIFO, see Fig. 15.3).

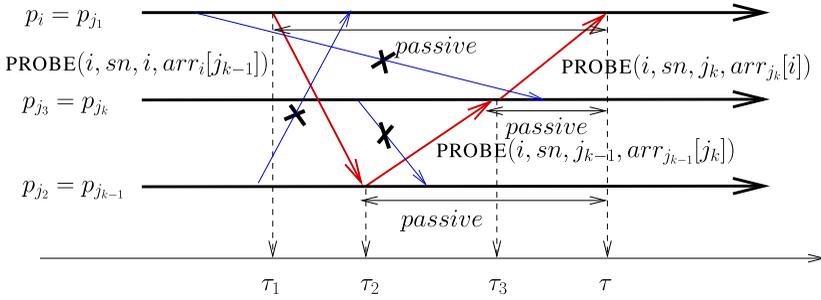
If  $p_i$  is passive and there is no message in transit from  $p_i$  to  $p_j$ ,  $obs_i$  checks first if it is the initiator of this probe (predicate  $k = i$ ). If it is, it declares that  $p_i$  is deadlocked (line 12). If  $k \neq i$  (i.e., the initiator of the probe identified  $(k, seqnb)$  is not  $obs_i$ ), and this is a new probe launched by  $obs_k$  (line 13),  $obs_i$  updates  $sn_i[k]$  (line 14) and propagates the probe by sending a control message  $PROBE(k, seqnb, i, arr_i[l])$  to each observer  $obs_\ell$  such that  $p_i$  is waiting for a message from  $p_\ell$ .

### 15.3.3 Proof of the Algorithm

**Theorem 25** *If a deadlock occurs, eventually a process involved in the associated cycle detects it. Moreover, no observer claims a deadlock if its associated process is not deadlocked.*

*Proof* Proof of the liveness property. Let us assume that a process  $p_i$  is deadlocked. This means that (a) there is a cycle of processes  $p_{j_1}, p_{j_2}, \dots, p_{j_k}, p_{j_1}$ , where  $i = j_1, j_2 \in dep\_set_{j_1}, j_3 \in dep\_set_{j_2}$ , etc.,  $j_1 \in dep\_set_{j_k}$ , (b) there is no application message in transit from  $p_{j_2}$  to  $p_{j_1}$ , from  $p_{j_3}$  to  $p_{j_2}$ , etc., and from  $p_{j_1}$  to  $p_{j_k}$ , and (c) none of these processes can be re-activated from the content of its input buffer.

Let us consider the observer  $obs_i$ , which launches a probe after the cycle has been formed (see Fig. 15.4, where  $k = 3$ ). The observer  $obs_i$  sends the message  $PROBE(i, sn, i, arr_i[j_2])$  to  $obs_{j_2}$  (line 9). As the channel from  $p_{j_2}$  to



**Fig. 15.5** Time instants in the proof of the safety property

$p_{j_1}$  is empty of application messages, the first time  $obs_{j_2}$  receives the message  $PROBE(i, sn, i, arr_i[j_2])$ , we have  $sent_{j_2}[i] = arr_i[j_2]$ , and  $obs_{j_2}$  executes lines 14–17. Consequently  $obs_{j_2}$  sends the message  $PROBE(i, sn, j_2, arr_{j_2}[j_3])$  to  $obs_{j_3}$ . And so on, until  $obs_{j_k}$  which receives the message  $PROBE(i, sn, j_{k-1}, arr_i[j_k])$  and sends the message  $PROBE(i, sn, j_k, arr_{j_k}[i])$  to  $obs_i$ . As the channel from  $p_i$  to  $p_{j_k}$  is empty, it follows from line 12 that, when it receives this message,  $obs_i$  is such that  $sent_i[j_k] = arr_{j_k}[i]$ . Hence,  $obs_i$  claims that  $p_i$  is deadlocked, which proves the liveness property.

**Proof of the safety property.** Let us consider an observer  $obs_i$  that claims that  $p_i$  is deadlocked. We have to show that  $p_i$  belongs to a cycle  $p_i = p_{j_1}, p_{j_2}, \dots, p_{j_k}, p_{j_1}$ , such that there is a time at which simultaneously (a)  $j_2 \in dep\_set_{j_1}$  and there is no message in transit from  $p_{j_2}$  to  $p_{j_1}$ , (b)  $j_3 \in dep\_set_{j_2}$  and there is no message in transit from  $p_{j_3}$  to  $p_{j_2}$ , (c) etc. until process  $p_{j_1}$  such that  $j_1 \in dep\_set_{j_k}$  and there is no message in transit from  $p_{j_1}$  to  $p_{j_k}$ .

As  $obs_i$  claims that  $p_{j_1}$  is deadlocked (line 12), there is a time  $\tau$  at which  $obs_{j_1}$  received a message  $PROBE(j_1, sn, j_k, arr_{j_k}[j_1])$  from some observer  $obs_{j_k}$ . Process  $p_{j_k}$  was passive when  $obs_{j_k}$  sent this message at some time  $\tau_k < \tau$ . Moreover, as  $obs_{j_1}$  did not discard this message when it received it (predicate  $sent_{j_1}[j_k] = arrived = arr_{j_k}[j_1]$ , line 11), the channel from  $p_{j_1}$  to  $p_{j_k}$  did not contain application messages between  $\tau_k$  and  $\tau$ . It follows that  $p_{j_k}$  remained continuously passive from the time  $\tau_k$  to time  $\tau$ . (See Fig. 15.5. The fact that there is no message in transit from one process to another is indicated by a crossed-out arrow.)

The same observation applies to  $obs_{j_k}$ . This local observer received at time  $\tau_k$  a control message  $PROBE(j_1, sn, j_{k-1}, arr_{j_{k-1}}[j_k])$ , which was sent by an observer  $obs_{j_{k-1}}$  at time  $\tau_{k-1}$ . As  $obs_{j_k}$  did not discard this message, we have  $sent_{j_k}[j_{k-1}] = arr_{j_{k-1}}[j_k]$  from which it follows that the channel from  $p_{j_k}$  to  $p_{j_{k-1}}$  did not contain application messages between  $\tau_{k-1}$  and  $\tau_k$ . Moreover, as  $j_k \in dep\_set_{j_{k-1}} \setminus arr\_from_{j_{k-1}}$ , and  $p_k$  remained continuously passive from time  $\tau_k$  to time  $\tau$ , it follows that  $p_{k-1}$  remained continuously passive from time  $\tau_{k-1}$  to time  $\tau$ . This reasoning can be repeated until the sending by  $p_{j_1}$  of the message  $PROBE(j_1, sn, j_1, arr_{j_1}[j_2])$ , at time  $\tau_1$ , from which we conclude that  $p_{j_1}$  remained continuously passive from time  $\tau_1$  to time  $\tau$ .

It follows that (a) the processes  $p_{j_1}, \dots, p_{j_k}$  are passive at time  $\tau$ , (b) the channel from  $p_{j_1}$  to  $p_{j_k}$ , the channel from  $p_{j_k}$  to  $p_{j_{k-1}}$ , etc., until the channel from  $p_{j_2}$  to  $p_{j_1}$  are empty at time  $\tau$ , and (c) none of these processes can be re-activated from the messages in its input buffer. Consequently these processes are deadlocked (which means that the cycle is a cycle of the wait-for graph), which concludes the proof of the safety property.  $\square$

## 15.4 Deadlock Detection in the OR Communication Model

As seen in Sect. 15.1, in the OR communication model, each receive statement specifies a set of processes, and the invoking process  $p_i$  waits for a message from any of these processes. This set of processes is the current dependence set of  $p_i$ , denoted  $dep\_set_i$ . As soon as a message from a process of  $dep\_set_i$  has arrived,  $p_i$  stops waiting and consumes it. This section presents an algorithm which detects deadlocks in this model. This algorithm is due to K.M. Chandy, J. Misra, and L.M. Haas (1983).

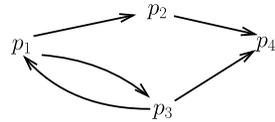
This algorithm considers the following slightly modified definition for a set of deadlocked processes, namely, a set  $D$  of processes is deadlocked if (a) all the processes of  $D$  are passive, (b) the dependency set of each of them is a subset of  $D$ , and (c) for each pair of processes  $\{p_i, p_j\} \in D$  such that  $j \in dep\_set_i$ , there is no message in transit from  $p_j$  to  $p_i$ . When compared to the definition given in Sect. 15.1.4, this definition includes the processes blocked by processes belonging to a knot of deadlocked processes. When considering the wait-for graph on the left of Fig. 15.1, this definition considers that  $p_4$  is deadlocked, while the definition of Sect. 15.1.4 considers it is blocked by a deadlocked process ( $p_2$ ).

This algorithm assumes also that a process  $p_i$  is passive only when it is waiting for a message from a process belonging to its current dependency set  $dep\_set_i$ , which means that processes do not terminate. The case where a process locally terminates (i.e., attains an “end” statement, after which it remains forever passive) is addressed in Problem 4.

### 15.4.1 Principle

**Network Traversal with Feedback** When the observer  $obs_i$  associated with a process  $p_i$  suspects that  $p_i$  is involved in a deadlock, it launches a parallel network traversal with feedback on the edges of the wait-for graph whose it is the origin, i.e., on the channels from  $p_i$  to  $p_j$  such that  $j \in dep\_set_i$ . (A network traversal algorithm with feedback that builds a spanning tree has been presented in Sect. 1.2.4.) If such a process  $p_j$  is active, it discards the message and, consequently, stops the network traversal. If it is itself blocked, it propagates the network traversal to the processes which currently define its set  $dep\_set_j$ . And so on. These control messages are used to build a spanning tree rooted at  $p_i$ .

**Fig. 15.6** A directed communication graph



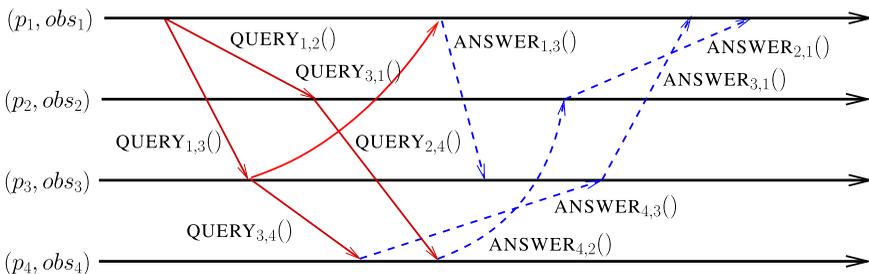
If the network traversal with feedback terminates (i.e.,  $obs_i$  receives an answer from each process  $p_j$  in  $dep\_set_i$ ), the aim is to allow  $obs_i$  to conclude that  $p_i$  is deadlocked. If an observer  $obs_j$  has locally stopped the progress of the network traversal launched by  $obs_i$ ,  $p_j$  is active and may send in the future a message to the process  $p_k$  that sent it a network traversal message. If re-activated, this process  $p_k$  may in turn re-activate a process  $p_\ell$  such that  $\ell \in dep\_set_k$ , etc. This chain of process re-activations can end in the re-activation of  $p_i$ .

**The Difficulty in the Observation of a Distributed Computation** The network traversal algorithms presented in Chap. 1 consider an underlying static network. Differently, the network defined by the wait relation (wait-for graph) is not static: it is modified by the computation. Hence the following problem: Can a network traversal algorithm designed for a static graph be used to do a consistent distributed observation of a graph that can be dynamically modified during its observation?

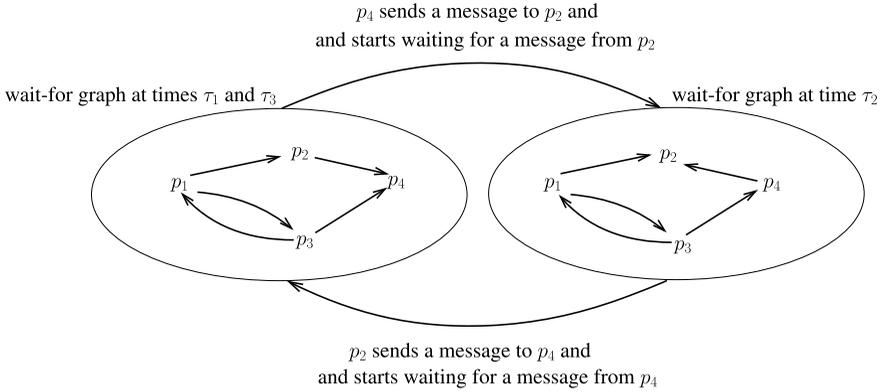
**Network Traversal with Feedback on a Directed Static Communication Graph**

Let QUERY be the type of messages that implement the propagation of a network traversal on the wait-for graph, and ANSWER the type of the messages that implement the associated feedback. (These messages are typed GO and BACK, respectively, in the network traversal with feedback algorithm described in Fig. 1.7).

Let us consider a four-process computation such that, at the application level, the communication graph is the directed graph described in Fig. 15.6. At the underlying level, the channels are bidirectional for the control messages exchanged by the process observers. Let us consider the case where the local observer  $obs_1$  launches a network traversal with feedback. This network traversal is described in Fig. 15.7, where the subscript  $(x, y)$  attached to a message means that this message is sent by  $obs_x$  to  $obs_y$ . First  $obs_1$  sends a message QUERY() to both  $obs_2$  and  $obs_3$  (messages QUERY<sub>1,2</sub>() and QUERY<sub>1,3</sub>()), then  $obs_2$  forwards the network traversal



**Fig. 15.7** Network traversal with feedback on a static graph



**Fig. 15.8** Modification in a wait-for graph

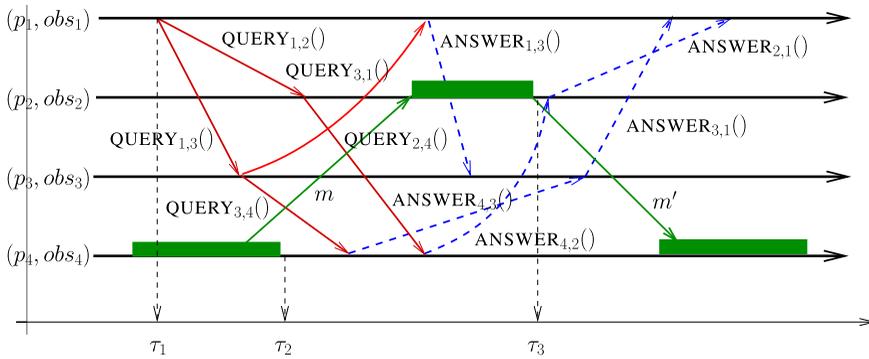
to  $obs_4$  (message  $QUERY_{2,4}()$ ), while  $obs_3$  forwards it to  $obs_1$  and  $obs_4$  (messages  $QUERY_{3,1}()$  and  $QUERY_{3,4}()$ ). As it cannot extend the traversal,  $obs_4$  sends back a message  $ANSWER()$  each time it receives a query (messages  $ANSWER_{4,2}()$  and  $ANSWER_{4,3}()$ ). As it has already been visited by the network traversal,  $obs_1$  sends by return the message  $ANSWER_{1,3}()$  when it receives  $QUERY_{3,1}()$ . When each of  $obs_2$  and  $obs_3$  has received all the answers matching its queries, it sends a message  $ANSWER()$  to  $obs_1$ . Finally, when  $obs_1$  has received the answers from  $obs_2$  and  $obs_3$ , the network traversal with feedback terminates.

**Network Traversal with Feedback on a Directed Dynamic Communication Graph** Let us now consider that at time  $\tau_1$ , the directed graph on the left of Fig. 15.8 is the current wait-for graph of the corresponding four-process computation. This means that, at  $\tau_1$ ,  $dep\_set_1 = \{2, 3\}$ ,  $dep\_set_2 = \{4\}$ ,  $dep\_set_3 = \{1, 4\}$ , and  $dep\_set_4 = \emptyset$  (hence  $p_1$ ,  $p_2$  and  $p_3$  are blocked, while  $p_4$  is active). Moreover, the channels are empty of application messages.

Let us consider the following scenario. Process  $p_4$  first sends a message  $m$  to  $p_2$  and then starts waiting for a message from  $p_2$ . The message  $m$  will re-activate  $p_2$ , and the wait-for graph (which is a conceptual tool) is instantaneously modified. This is depicted in Fig. 15.8, where the directed graph on the left side is wait-for graph at time  $\tau_1$ , i.e., just before  $p_4$  sends a message to  $p_2$  and starts waiting for a message from this process, while the graph on the right side is the wait-for graph at time  $\tau_2$ , just after  $p_4$  has executed these statements.

After  $p_2$  has been re-activated by the application message  $m$ , it sends a message  $m'$  to  $p_4$  and starts waiting for a message from this process. Let  $\tau_3$  be the time instant just after this has been done. The corresponding modification of the wait-for graph is indicated at the bottom of Fig. 15.8.

Let us finally consider that  $obs_1$  launches a network traversal with feedback at time  $\tau_1$ . This network traversal, which is depicted in Fig. 15.9, is exactly the same as that described in Fig. 15.7. Let us recall that, as indicated previously, an observer



**Fig. 15.9** Inconsistent observation of a dynamic wait-for graph

$obs_i$  propagates a network traversal only if  $p_i$  is passive. In the scenario which is described, despite the application messages  $m$  and  $m'$  exchanged by  $p_4$  and  $p_2$ , the control messages are received by any observer  $obs_i$  while the process  $p_i$  it is associated with is passive. Hence, the network traversal with feedback terminates, and  $obs_1$  concludes erroneously that  $p_1$  is involved in a deadlock.

**Observe if Processes Have Been Continuously Passive** The erroneous observation comes from the fact that there is process activity in the back of the network traversal, and the network traversal misses it. Let us observe that the FIFO property of the channels is not sufficient to solve the problem (in Fig. 15.9 all channels behave as FIFO channels).

A simple way to solve this problem consists in requiring that each observer  $obs_i$  observes if its process  $p_i$  remained continuously passive between the time it was visited by the network traversal (reception of the first message  $QUERY()$  from an observer  $obs_j$ ) and the time it sent a message  $ANSWER()$  to this observer  $obs_j$ . As demonstrated by the algorithm that follows, this observation Boolean value plus the fact that channels are FIFO allows for a consistent distributed observation of the computation.

As far as channels are concerned, let us notice that, if channels were not FIFO, erroneous observation could occur, despite the previous Boolean values. To see this, it is sufficient to consider the case where, in Fig. 15.9, the message  $m$  sent by  $p_4$  to  $p_2$  arrives after the control message  $ANSWER()$  sent by  $obs_4$  to  $obs_2$ .

### 15.4.2 A Detection Algorithm

**Local Variables** As in previous algorithms, each observer  $obs_i$  manages a local variable  $state_i$  that describes the current state of  $p_i$  (*active* or *passive*). The network whose traversal is launched by  $obs_i$  is dynamically defined according to the current

values of the sets  $dep\_set_j$ . (By definition  $dep\_set_j = \emptyset$  if  $p_j$  is active.) The vertices of the corresponding directed graph are the observers in the set  $DP_i$ , which is recursively defined as follows

$$DP_i = \{i\} \bigcup_{x \in DP_i} dep\_set_x,$$

and there is an edge from  $obs_x$  to  $obs_y$  if  $y \in dep\_set_x$ .

To implement network traversals while ensuring a correct observation, each observer  $obs_i$  manages the following local arrays whose entry  $k$  concerns the last network traversal initiated by  $obs_k$ .

- $sn_i[1..n]$  is an array of sequence numbers, whose meaning is the same as in the algorithm of Fig. 15.2, namely,  $sn_i[j]$  is the sequence number associated with the last network traversal initiated by  $obs_j$  and known by  $obs_i$ . Initially,  $sn_i[1..n] = [0, \dots, 0]$ .

The last network traversal initiated by an observer  $obs_i$  is consequently identified by the pair  $(i, sn_i[i])$ .

- $parent_i[1..n]$  is array such that  $parent_i[j]$  contains the parent of  $obs_i$  with respect to the last network traversal launched by  $obs_j$  and known by  $obs_i$ .
- $expected\_aswr_i[1..n]$  is array of non-negative integers such that  $expected\_aswr_i[j]$  is the number of messages ANSWER() that  $obs_i$  has still to receive before sending an answer to its parent, when considering the network traversal identified  $(j, sn_i[j])$ .

The pair of local variables  $(parent_i[j], expected\_aswr_i[j])$  is related to the last network traversal initiated by  $obs_j$ . The set of variables  $parent_i[j]$  implements a spanning tree rooted at  $obs_j$ , which is rebuilt during each network traversal launched by  $obs_j$ .

- $cont\_passive_i[1..n]$  is an array of Boolean values. For any  $j$ ,  $cont\_passive_i[j]$  is used to register the fact that during the last visit of a network traversal initiated by  $obs_j$ ,  $p_i$  did or did not remain continuously passive. This last visit started at the last modification of  $sn_i[j]$ .

**The Algorithm** The deadlock detection algorithm for the OR communication model is described in Fig. 15.10. As in the previous algorithm, the sequences of statements prefixed by **when** are executed atomically.

When a process  $p_i$  enters a receive statement,  $obs_i$  computes the associated set  $dep\_set_i$  and  $p_i$  becomes passive if there is no message from a process that belongs to  $dep\_set_i$ , which has already arrived and can be consumed (lines 1–3). If there is such a message,  $p_i$  consumes it and continues its execution.

When a message arrives from a process  $p_j$  and  $j \in dep\_set_i$ ,  $p_j$  is re-activated, and the Boolean array  $cont\_passive_i[1..n]$  is reset to  $[false, \dots, false]$  (lines 6–10).

When  $obs_i$  suspects that  $p_i$  is deadlocked (we have then necessarily  $state_i = passive$ ), it increases  $sn_i[i]$  and sends a query—identified as  $(i, sn_i[i])$ —to the observers associated with the processes  $p_j$  such that  $j \in dep\_set_i$  (because only a message from one of these processes can re-activate  $p_i$ ). It also assigns the value

```

when  $p_i$  enters a receive statement do
(1) compute  $dep\_set_i$  from the receive statement;
(2) if ( $\forall p_j$  such that  $j \in dep\_set_i$ :
      there is no message arrived from  $p_j$  and not yet consumed)
(3)   then  $state_i \leftarrow passive$ 
(4)   else consume a message received a process in  $dep\_set_i$ 
(5)   end if.

when a message arrives from  $p_j$  do
(6) if ( $state_i = passive$ )  $\wedge$  ( $j \in dep\_set_i$ )
(7)   then for each  $j \in \{1, \dots, n\}$  do  $cont\_passive_i[j] \leftarrow false$  end for;
(8)    $state_i \leftarrow active$ 
(9)   else keep the message in input buffer
(10)  end if.

when  $obs_i$  suspects that  $p_i$  is deadlocked do % we have then  $state_i = passive$  %
(11)  $sn_i[i] \leftarrow sn_i[i] + 1$ ;
(12) for each  $j \in dep\_set_i$  do send QUERY( $i, sn_i[i]$ ) to  $obs_j$  end for;
(13)  $expected\_aswr_i[i] \leftarrow |dep\_set_i|$ ;
(14)  $cont\_passive_i[i] \leftarrow true$ .

when QUERY( $k, seqnb$ ) is received from  $obs_j$  do
(15) if ( $state_i = passive$ )
(16)   then if ( $seqnb > sn_i[k]$ )
(17)      $sn_i[k] \leftarrow seqnb$ ;  $parent_i[k] \leftarrow j$ ;
(18)     for each  $j \in dep\_set_i$  do send QUERY( $k, seqnb$ ) to  $obs_j$  end for;
(19)      $expected\_aswr_i[k] \leftarrow |dep\_set_i|$ ;
(20)      $cont\_passive_i[k] \leftarrow true$ 
(21)   else if ( $cont\_passive_i[k] \wedge (seqnb = sn_i[k])$ )
(22)     then send ANSWER( $k, seqnb$ ) to  $obs_j$ 
(23)     end if
(24)   end if
(25) end if.

when ANSWER( $k, seqnb$ ) is received from  $obs_j$  do
(26) if ( $(seqnb = sn_i[k]) \wedge cont\_passive_i[k]$ )
(27)    $expected\_aswr_i[k] \leftarrow expected\_aswr_i[k] - 1$ ;
(28)   if ( $expected\_aswr_i[k] = 0$ )
(29)     if ( $k = i$ ) then claim  $p_i$  is deadlocked
(30)     else let  $x = parent_i[k]$ ; send ANSWER( $k, seqnb$ ) to  $obs_x$ 
(31)   end if
(32)   end if
(33) end if.

```

**Fig. 15.10** An algorithm for deadlock detection in the OR communication model

$|dep\_set_i|$  to  $expected\_aswr_i[i]$ , and (because it starts a new observation period of  $p_i$ ) it sets  $cont\_passive_i[i]$  to true (lines 11–14).

When  $obs_i$  receives a message QUERY( $k, seqnb$ ) from  $obs_j$  it discards the message if  $p_i$  is active (line 15). Hence, the network traversal launched by  $obs_k$  and

identified  $(k, seqnb)$  will not terminate, and consequently this network traversal will not allow  $obs_k$  to claim that  $p_i$  is deadlocked. If  $p_i$  is passive, there are two cases:

- If  $seqnb > sn_i[k]$ ,  $obs_i$  discovers that this query concerns a new network traversal launched by  $obs_k$ . Consequently, it updates  $sn_i[k]$  and defines  $obs_j$  as its parent in this network traversal (line 17). It then extends the network traversal by propagating the query message it has received to each observer of  $dep\_set_i$  (line 18), updates accordingly  $expected\_aswr_i[i]$  (line 19), and starts a new observation period of  $p_i$  (with respect to this network traversal) by setting  $cont\_passive_i[k]$  to true (line 20).
- If  $seqnb \neq sn_i[k]$ ,  $obs_i$  stops the network traversal if it is an old one ( $seqnb < sn_i[k]$ ), or if  $p_i$  has been re-activated since the beginning of the observation period (which started at the first reception of a message  $QUERY(k, seqnb)$ ).

If the query concerns the last network traversal launched by  $p_k$  and  $p_i$  has not been re-activated since the start of the local observation period associated with this network traversal,  $obs_i$  sends by return the message  $ANSWER(k, seqnb)$  to  $obs_j$  (line 22). This is needed to allow the network traversal to return to its initiator (if no other observer stops it).

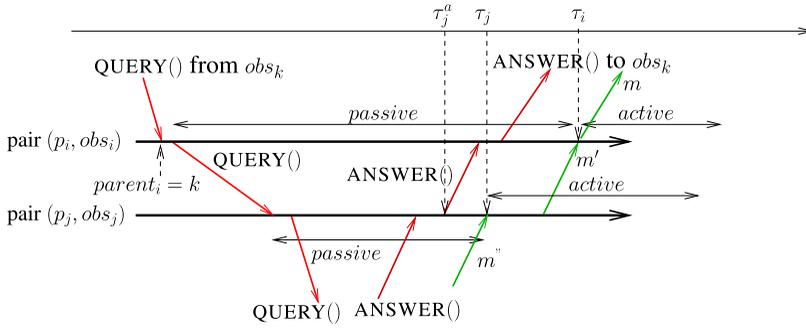
Finally, when an observer  $obs_i$  receives a message  $ANSWER(k, seqnb)$  it discards the message if this message is related to an old traversal, or if  $p_i$  did not remained continuously passive since the beginning of this network traversal. Hence, if  $(seqnb = sn_i[k]) \wedge cont\_passive_i[k]$ ,  $obs_i$  first decreases  $expected\_aswr_i[k]$  (line 27). Then, if  $expected\_aswr_i[k] = 0$ , the network traversal with feedback can leave  $obs_i$ . To that end, if  $k \neq i$ ,  $obs_i$  forwards the message  $ANSWER(k, seqnb)$  to its parent in the tree built by the first message  $QUERY(k, seqnb)$  it has received. If  $k = i$ , the network traversal has returned to  $obs_i$ , which claims that  $p_i$  is deadlocked.

### 15.4.3 Proof of the Algorithm

**Theorem 26** *If a deadlock occurs, eventually the observer of a deadlocked process detects it. Moreover, no observer claims a deadlock if its associated process is not deadlocked.*

*Proof* Proof of the liveness property. Let  $D$  be a set of deadlocked processes, and  $p_i$  be one of these processes. Let us assume that, after the deadlock has occurred,  $obs_i$  launches a deadlock detection. We have to show that  $obs_i$  eventually claims that  $p_i$  is deadlocked.

As there is deadlock, there is no message in transit among the pairs of processes of  $D$  such that  $i, j \in D$  and  $j \in dep\_set_i$ . Let  $(i, sn)$  be the identity of the network traversal with feedback launched by  $obs_i$  after the deadlock occurred. The observer  $obs_i$  sends a message  $QUERY(i, sn)$  to each observer  $obs_j$ , such that  $j \in dep\_set_i$ . When such an observer receives this message it sets  $cont\_passive_j[i]$  to true, and forwards  $QUERY(i, sn)$  to all the observers  $obs_k$  such that  $k \in dep\_set_j$ , etc. As there



**Fig. 15.11** Activation pattern for the safety proof

is no message in transit among the processes of  $D$ , and all the processes in  $D$  are passive, all the Boolean variables  $cont\_passive_x[i]$  of the processes in  $D$  are set to true and thereafter remain true forever. It follows that no message  $QUERY(i, sn)$  or  $ANSWER(i, sn)$  is discarded. Consequently, no observer stops the progress of the network traversal which returns at  $obs_i$  and terminates, which concludes the proof of the liveness of the detection algorithm.

Proof of the safety property. We have to show that no observer  $obs_i$  claims that  $p_i$  is deadlocked while it is not. Figure 15.11 is used to illustrate the proof.

*Claim C* Let  $obs_i$  be an observer that sends  $ANSWER()$  to  $obs_k$  where  $parent_i = k$ . Let us assume that, after this answer has been sent,  $p_i$  sends a message  $m$  to  $p_k$  at time  $\tau_i$ . Then, there is an observer  $obs_j$  such that  $j \in dep\_set_i$ , which received  $QUERY()$  from  $obs_i$  and, subsequently,  $obs_j$  sent an answer to  $obs_i$  at some time  $\tau_j^a$  and  $p_j$  became active at some time  $\tau_j$  such that  $\tau_j^a < \tau_j < \tau_i$ .

*Proof of the Claim C* In order for  $obs_i$  to send  $ANSWER()$  to its parent, it needs to have received an answer from each observer  $obs_j$  such that  $j \in dep\_set_i$  (lines 19, 27–28, and 30). Moreover, for  $p_i$  to become active and send a message  $m$  to  $p_k$ , it needs to have received an application message  $m'$  from a process belonging to  $dep\_set_i$  (lines 6 and 8). Let  $p_j$  be this process.

As the channels are FIFO, and  $obs_j$  sent a message  $ANSWER()$  to  $obs_i$ ,  $m'$  is not an old message still in transit; it has necessarily been sent after  $obs_j$  sent the answer message to  $obs_i$ . It follows that  $m'$  is received after the answer message and we have consequently,  $\tau_j^a < \tau_j$ . Finally, as the sending of  $m'$  by  $p_j$  occurs before its reception by  $p_i$  (which occurs after  $obs_i$  sent an answer to  $obs_k$ ), it follows that  $p_j$  becomes active at some time  $\tau_j$  such that  $\tau_j < \tau_i$ , and we obtain  $\tau_j^a < \tau_j < \tau_i$ . (End of the proof of the claim.)

Let  $D$  be the set of processes involved in the network traversal with feedback initiated by an observer  $obs_z$  which executes line 29 and claims that  $p_z$  is deadlocked. As the network traversal returns to its initiator  $obs_z$ , it follows that every observer

in  $D$  has received a matching answer for each query it has sent. Hence, every observer sent an answer message to its parent in the spanning tree built by the network traversal. Moreover, there is at least one cycle in the set  $D$  (otherwise, the network traversal will not have returned to  $obs_z$ ).

Let us consider any of these cycles after  $obs_z$  has claimed a deadlock at some time  $\tau$ . This cycle includes necessarily a process  $p_i$  that, when it receives a query from a process  $p_k$  of  $D$  defines  $p_k$  as its parent (line 17). Let us assume that such a process  $p_i$  is re-activated at some time  $\tau_i > \tau$ . It follows from the Claim C that there is a process  $p_j$ , such that  $j \in dep\_set_i$  and  $p_j$  sent to  $p_i$  a message  $m'$  such that  $\tau_j^a < \tau_j < \tau_i$ . When considering the pair  $(p_j, obs_j)$ , and applying again Claim C, it follows that there a process  $p_\ell$  such that  $\ell \in dep\_set_j$  and  $p_\ell$  sent to  $p_j$  a message  $m''$  such that  $\tau_\ell^a < \tau_\ell < \tau_j$ . Hence,  $\tau_\ell < \tau_j$ .

Let  $p_i, p_j, p_\ell, \dots, p_x, p_i$  be the cycle. Using inductively the previous argument, it follows that  $i \in dep\_set_x$  and  $p_i$  sent to  $p_x$  a message  $m^x$  at some time  $\tau_i'$  such that  $\tau_i' < \tau_x$ . Hence, we obtain  $\tau_i' < \tau_x < \dots < \tau_\ell < \tau_j < \tau_i$ . But as (a)  $p_i$  remained continuously passive between the reception of QUERY() from  $p_x$  and the sending of the matching message ANSWER(), and (b) the channel from  $p_i$  to  $p_x$  is FIFO, it follows that  $\tau_i' > \tau_i$ , a contradiction.

As the previous reasoning is for any cycle in the set  $D$ , it follows that, for any pair  $\{p_y, p_{y'}\} \in D$  such that  $y \in dep\_set_{y'}$ , there is no message in transit from  $p_y$  to  $p_{y'}$  at time  $\tau$ , which concludes the proof of the safety property.  $\square$

## 15.5 Summary

A process becomes deadlocked when, while blocked, its progress depends transitively on itself. This chapter has presented two types of communication (or resource) models, which are the most often encountered. In the AND model, a process waits for messages from several processes (or for resources currently owned by other processes). In the OR model, it waits for one message (or resource) from several possible senders (several distinct resources). After an analysis of the deadlock phenomenon and its capture by the notion of a wait-for graph, the chapter presented three deadlock detection algorithms. The first one is suited to the one-at-a-time model, which is the simplest instance of both the AND and OR models, namely, a process waits for a single message from a given sender (or a single resource) at a time. The second algorithm allows for the detection of deadlocks in the AND communication model, while the third one allows for the detection of deadlocks in the OR communication model.

## 15.6 Bibliographic Notes

- The deadlock problem originated in resource allocation when the very first operating systems were designed and implemented in the 1960s and early 1970s

(e.g., [97, 160, 165, 166, 188]). The deadlock problem has also been intensively studied in database systems (e.g., [218, 259, 287, 331]).

- The advent of distributed systems, distributed computing, and graph-based algorithms has given rise to a new impetus for deadlock detection, and many new algorithms have been designed (e.g., [61, 169, 216, 278] to cite a few).
- Algorithms to detect cycles and knots in static graphs are presented in [248, 264].
- The deadlock detection algorithm for the one-at-a-time model presented in Sect. 15.2 is due D.L. Mitchell and M. Merritt [266].
- The deadlock detection algorithm for the AND communication model presented in Sect. 15.3 is new. Another detection algorithm suited to this model is presented in [260], and a deadlock avoidance algorithm for the AND model is described in [389].
- The deadlock detection algorithm for the OR communication model presented in Sect. 15.4 is due to K.M. Chandy, J. Misra, and L.M. Haas [81]. This algorithm is based on the notion of a diffusing computation introduced by E.W. Dijkstra and C.S. Scholten in [115]. Another algorithm for the OR model is described in [379].
- A deadlock detection algorithm for a very general communication model including the AND model, the OR model, the  $k$ -out-of- $n$  model, and their combination is described and proved correct in [65].
- Proof techniques for deadlock absence in networks of processes are addressed in [76] and an invariant-based verification method for a deadlock detection algorithm is investigated in [215].
- Deadlock detection algorithms for synchronous systems are described in [308, 391].
- Deadlock detection in transaction systems is addressed in [163].
- Introductory surveys on distributed deadlock detection can be found in [206, 346].

## 15.7 Exercises and Problems

1. Considering the deadlock detection algorithm presented in Sect. 15.2, let us replace the detection predicate  $priv_i = pub_j$  by  $pub_i = pub_j$ , in order to eliminate the local control variable  $priv_i$ .

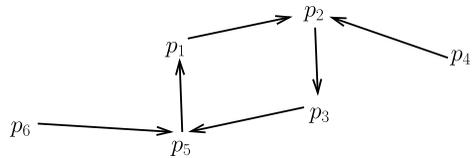
Is this predicate correct to ensure that each deadlock is detected, and is it detected by a single process, which is a process belonging to a cycle? To show the answer is “yes”, a proof has to be designed. To show the answer is “no”, a counterexample has to be produced. (To answer this question, one can investigate the particular case of the wait-for graph described in Fig. 15.12.)

2. Adapt the deadlock detection algorithm suited to the AND communication model presented in Sect. 15.3 to obtain an algorithm that works for the AND resource model.

Solution in [81].

3. Let us consider the general OR/AND receive statement defined in the previous chapter devoted to termination detection (Sect. 14.5). Using the predicate

**Fig. 15.12** Another example of a wait-for graph



*fulfilled()*, design an algorithm which detects communication deadlocks in this very general communication model.

(Hint: the solution consists in an appropriate generalization of the termination detection algorithm presented in Fig. 14.16.)

Solution in [65].

4. Considering the deadlock detection algorithm for the OR communication model described in Fig. 15.10, let us assume that, when a process  $p_i$  attains its “end” statement,  $obs_i$  (a) sets  $dep\_set_i$  to  $\emptyset$ , and (b) for any pair  $(k, sn)$ , sends systematically by return ANSWER( $k, sn$ ) each time it receives a message QUERY( $k, sn$ ). In that way, a locally terminated process never stops a network traversal with feedback.

Does this extension leave the detection algorithm correct? When a deadlock is detected by an observer  $obs_k$ , is it possible for  $obs_k$  to know which are the locally terminated processes involved in the deadlock?

## Part VI

# Distributed Shared Memory

A distributed shared memory is an abstraction that hides the details of communicating by sending and receiving messages through a network. The processes cooperate to a common goal by using shared objects (also called concurrent objects). The most famous of these objects is the read/write register, which gives the illusion that the processes access a classical shared memory. Other concurrent objects are the usual objects such as queues, stacks, files, etc.

This part of the book is devoted to the implementation of a shared memory on top of a message-passing system. To that end it investigates two consistency conditions which can be associated with shared objects, namely, atomicity (also called linearizability), and sequential consistency. For a given object, or a set of objects, a consistency condition states which of its executions are the correct ones. As an example, for a read/write shared register, it states which are the values that must be returned by the invocations of the read operation.

This part of the book is made up of two chapters. After having presented the general problem of building a shared memory on top of a message-passing system, Chap. 16 addresses the atomicity (linearizability) consistency condition. It defines it, presents its main composability property, and describes distributed algorithms that implement it. Then, Chap. 17 considers the sequential consistency condition, explains its fundamental difference with respect to atomicity, and presents several implementations of it.

# Chapter 16

## Atomic Consistency (Linearizability)

This chapter is on the strongest consistency condition for concurrent objects. This condition is called *atomicity* when considering shared registers, and *linearizability* when considering more sophisticated types of objects. In the following, these two terms are considered as synonyms.

The chapter first introduces the notion of a distributed shared memory. It then defines formally the atomicity concept, and presents its main *composability* property, and several implementations on top of a message-passing system.

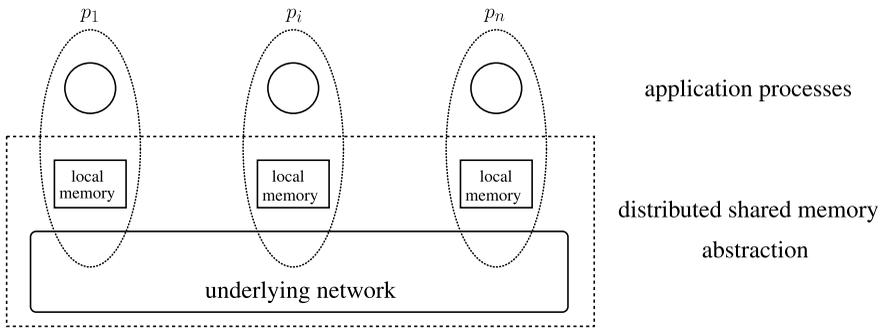
**Keywords** Atomicity · Composability · Concurrent object · Consistency condition · Distributed shared memory · Invalidation vs. update · Linearizability · Linearization point · Local property · Manager process · Object operation · Partial order on operations · Read/write register · Real time · Sequential specification · Server process · Shared memory abstraction · Total order broadcast abstraction

### 16.1 The Concept of a Distributed Shared Memory

**Concurrent Objects and Sequential Specification** An object is defined by a set of operations and a specification that defines the meaning of these operations. A concurrent object is an object which can be accessed (possibly concurrently) by several processes.

As an example, an unbounded stack  $S$  is defined by two operations denoted  $S.push()$  and  $S.pop()$ , and the following specification. An invocation of  $S.push(v)$  adds the value  $v$  to the stack object, and an invocation of  $S.pop()$  withdraws from the stack and returns the last value that has been added to the stack. If the stack is empty,  $S.pop()$  returns a predefined default value  $\perp$  (this default value is a value that cannot be added to the stack).

As we can see, this specification uses the term “last” and consequently assumes that the invocations of  $S.push()$  and  $S.pop()$  are totally ordered. Hence, it implicitly refers to some notion of time. More precisely, it is a sequential specification, i.e., a specification which defines the correct behaviors of an object by describing all the sequences of operation executions which are allowed.



**Fig. 16.1** Structure of a distributed shared memory

All the concurrent objects considered in the following are assumed to be defined by a sequential specification.

**Operations of a Register** A register  $R$  is an object which can be accessed by two operations denoted  $R.read()$  and  $R.write()$ . Intuitively, a register is atomic if (a) each operation appears as if it has been executed instantaneously between its start event and its end event, (b) no two writes appear as being executed simultaneously, and (c) each read returns the value written by the last write which precedes it.

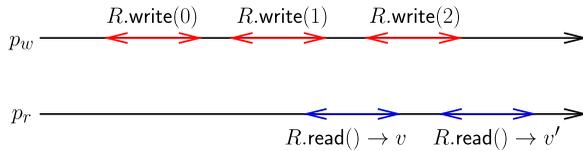
A formal specification of an atomic register is given below. A formal specification of a sequentially consistent register will be given in the next chapter. These definitions differ mainly in the underlying notion of time they use. Atomicity involves real time, while sequential consistency involves a logical time.

**Shared Memory** A shared memory is a set of concurrent objects. At the basic level of a centralized system these objects are the primitive read/write registers provided by the hardware. At a higher abstraction level, a shared memory can be made up of more sophisticated objects such as shared queues, stacks, sets, etc.

**Distributed Shared Memory System** A distributed shared memory system is a distributed algorithm that implements a shared memory abstraction on top of a message-passing system. To that end, the algorithm uses the local memories of the processes and the underlying message-passing system. The local memories are used to store the physical representation of the objects, and messages are used by the processes to cooperate in order to implement the operations on the objects so that their specification is never violated.

The structure of a distributed shared memory is represented in Fig. 16.1. As already indicated, according to the abstraction level which is realized, the shared memory can be composed of read/write registers, or objects of  $n$  higher abstraction level.

**Fig. 16.2** Register:  
What values can be returned  
by read operations?



## 16.2 The Atomicity Consistency Condition

### 16.2.1 What Is the Issue?

Let us consider Fig. 16.2, which represents a computation involving two processes accessing a shared register  $R$ . The process  $p_w$  issues write operations, while the process  $p_r$  issues read operations (the notation  $R.read() \leftarrow v$  means that the value returned by the corresponding read is  $v$ ).

The question is: Which values  $v$  and  $v'$  can be returned for this register execution to be correct? As an example do  $v = 0$  and  $v' = 2$  define a correct execution? Or do  $v = 2$  and  $v' = 1$  define a correct execution? Are several correct executions possible or is a single one possible?

The aim of a consistency condition is to answer this question. Whatever the object (register, queue, etc.), there are several meaningful answers to this question, and atomicity is one of them.

### 16.2.2 An Execution Is a Partial Order on Operations

By a slight abuse of language, we use the term “operation” also for “execution of an operation on an object”. Let  $OP$  be the set of all the operations issued by the processes.

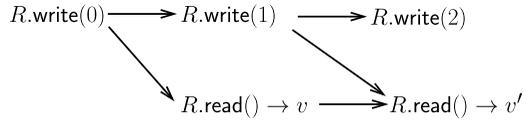
A computation of a set of processes accessing a set of concurrent objects is a partial order on the set of operations issued by the processes. This partial order, denoted  $\widehat{OP} = (OP, \xrightarrow{op})$ , is defined as follows. Let  $op_1$  be any operation issued by a process  $p_i$ , and  $op_2$  be any operation issued by a process  $p_j$ ;  $op_1$  is on object  $X$ , while  $op_2$  is on object  $Y$  (possibly  $i = j$  or  $X = Y$ ).  $op_1 \xrightarrow{op} op_2$  if  $op_1$  terminated before  $op_2$  started.

The projection of  $\xrightarrow{op}$  on the operations issued by a process is called *process order* relation. As each process  $p_i$  is sequential, the *process order* relation defines  $n$  total orders (one per process). When  $op_1$  and  $op_2$  are operations on the same object  $X$ , the projection of  $\xrightarrow{op}$  on the operations on  $X$  is called the *object order* relation.

Two operations which are not ordered by  $\xrightarrow{op}$  are said to be *concurrent* or *overlapping*. Otherwise they are *non-overlapping*.

The relation  $\xrightarrow{op}$  associated with the computation described in Fig. 16.2 is depicted in Fig. 16.3. The first read operation issued by  $p_r$  is concurrent with the two

**Fig. 16.3** The relation  $\xrightarrow{op}$  of the computation described in Fig. 16.2



last write operations issued by  $p_w$ , while its last read operation is concurrent only with the last write operation.

The relation  $\xrightarrow{op}$  associated with the computation described in Fig. 16.2 is depicted in Fig. 16.3.

**Remark** Let us notice that this definition generalizes the definition of a message-passing execution given in Sect. 6.1.2. A message-passing system is a system where any directed pair of processes  $(p_i, p_j)$  communicate by  $p_i$  depositing (sending) values (messages) in an object (the channel from  $p_i$  to  $p_j$ ), and  $p_j$  withdrawing (receiving) values from this object. Moreover, the inescapable transit time of each message is captured by the fact that a value is always withdrawn after it has been deposited.

**Sequential Computation, Equivalent Computations** A computation  $\widehat{OP}$  is *sequential* if “ $\xrightarrow{op}$ ” is a total order.

Let  $\alpha$  be any object  $X$  or any process  $p_i$ .  $\widehat{OP}|_\alpha$  ( $OP$  at  $\alpha$ ) denotes the projection of  $\widehat{OP}$  on  $\alpha$  (i.e., the partial order involving only the operations accessing the object  $\alpha$  if  $\alpha$  is an object, or issued by  $\alpha$  if  $\alpha$  is a process). As each process  $p_i$  is sequential, let us observe that  $\widehat{OP}|_{p_i}$  is a total order (the trace of the operations issued by  $p_i$ ).

Two computations  $\widehat{OP}_1$  and  $\widehat{OP}_2$  are *equivalent* if for any process  $p_i$ ,  $\widehat{OP}_1|_{p_i}$  is the same as  $\widehat{OP}_2|_{p_i}$ . This means that, when they are equivalent, no process can distinguish  $\widehat{OP}_1$  from  $\widehat{OP}_2$ .

**Legality** A sequential history is *legal* if it meets the sequential specification of all its objects. This means that for any object  $X$ ,  $\widehat{OP}|_X$  (i.e., the sequence of all the operations accessing  $X$ ) is a sequence that belongs to the specification of  $X$ .

If  $X$  is a register, this means that no read returns an overwritten value. If  $X$  is a stack this means that each invocation of `pop()` returns the last value that has been added to the stack (“last” with respect to the order defined by the sequence  $\widehat{OP}|_X$ ), etc.

### 16.2.3 Atomicity: Formal Definition

**Atomic Computation** A computation  $\widehat{OP}$  is *atomic* (or *linearizable*) if there is a sequential computation  $\widehat{S}$  such that

- $\widehat{OP}$  and  $\widehat{S}$  are equivalent (i.e., no process can distinguish  $\widehat{OP}$  and  $\widehat{S}$ ),
- $\widehat{S}$  is legal (i.e., the specification of each object is respected), and

- The total order defined by  $\widehat{S}$  respects the partial order defined by  $\widehat{OP}$  (i.e., whatever the processes that invoke them and the objects they access, any two operations ordered in  $\widehat{OP}$  are ordered the same way in  $\widehat{S}$ ; let us recall the order of non-overlapping operations in  $\widehat{OP}$  captures their real-time order).

This definition means that, for  $\widehat{OP}$  to be atomic, everything has to appear as if the computation was (a) sequential and legal with respect to each object  $X$  ( $X$  behaves as described by its sequential specification), (b) and in agreement with “real-time” (if an operation  $op_1$  terminates before another operation  $op_2$  starts, then  $op_1$  has to appear before  $op_2$  in  $\widehat{S}$ ).

Such an  $\widehat{S}$  is a computation that could have been obtained by executing all the operations, one after the other, while respecting the occurrence order of all the non-overlapping operations.

**Atomic Register** A register  $X$  is *atomic* (or behaves atomically) in a computation  $\widehat{OP}$  if the computation  $\widehat{OP}|X$  is atomic.

It follows from the definition of an atomic computation  $\widehat{OP}$ , and the definition of legality, that the sequential execution  $\widehat{S}$  is such that, for any object  $X$ ,  $\widehat{S}|X$  is legal. Hence, if a computation  $\widehat{OP}$  is atomic, so are all the objects involved in this computation.

**Linearization and Linearization Point** If computation  $\widehat{OP}$  is linearizable, a sequential execution such as the previous sequence  $\widehat{S}$ , is called a *linearization* of the computation  $\widehat{OP}$ .

The very existence of a linearization  $\widehat{S}$  means that, from an external observer point of view, each operation could have been executed at a point of the time line that lies between the time this operation starts and the time it ends. Such a point is called the *linearization point* of the corresponding operation.

Let us notice that proving that an algorithm implements atomic consistency amounts to identifying a linearization point for each of its operations, i.e., time instants that respect the occurrence order of non-overlapping operations and are in agreement with the sequential specification of each object.

**An Example** An execution of a read/write register accessed by three processes is described in Fig. 16.4. As we can see, the executions of the second read by  $p_i$ , the second write by  $p_j$ , and the write by  $p_k$  are overlapping. The line at the bottom of the figure represents the time line of an omniscient external observer. Two dotted arrows are associated with each operation. They meet at a point of the omniscient external observer’s time line at which the corresponding operation could have been instantaneously executed.

The linearization points represented by bullets on the observer’s time line describes a sequence  $\widehat{S}$  that (a) respects the occurrence order of non-overlapping operations, and (b) belongs to the sequential specification of a register. It follows that, in this execution, the register behaves as an atomic register.

Another computation is described in Fig. 16.5. It differs from the previous one in the fact that the concurrent (overlapping) operations  $R.write(2)$  by  $p_j$  and

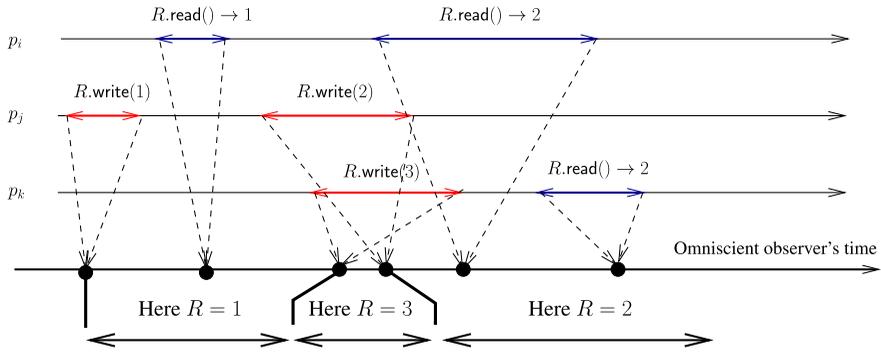


Fig. 16.4 An execution of an atomic register

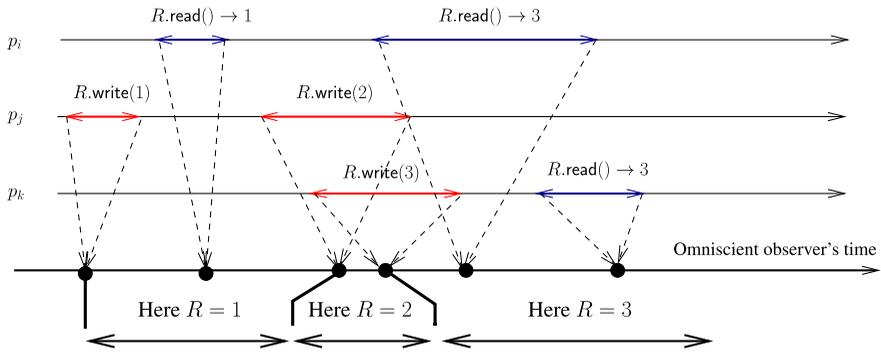


Fig. 16.5 Another execution of an atomic register

$R.write(3)$  by  $p_k$  are ordered in the reverse order. If the read operations issued by  $p_i$  and  $p_j$  return the value 3, the register is atomic. If one of them returns another value, it is not.

Let us finally notice that, as the second read by  $p_i$ , the operation  $R.write(2)$  by  $p_j$ , and the operation  $R.write(3)$  by  $p_k$  are all concurrent, it is possible that the execution be such that the second read by  $p_i$  appears as being linearized between  $R.write(2)$  by  $p_j$  and  $R.write(3)$  by  $p_k$ . In that case, for  $R$  to be atomic, the second read by  $p_i$  has to return the value 2.

### 16.3 Atomic Objects Compose for Free

**The Notion of a Local Property** Let  $P$  be any property defined on a set of objects. The property  $P$  is said to be *local* if the set of objects as a whole satisfies  $P$  whenever each object taken separately satisfies  $P$ .

Locality is an important concept that promotes modularity. Let us consider some local property  $P$ . To prove that an entire set of objects satisfy  $P$ , we have only to ensure that each object, independently from the others, satisfies  $P$ . As a consequence, the property  $P$  can be implemented for each object independently of the way it implemented for the other objects. At one extreme, it is even possible to design an implementation where each object has its own algorithm implementing  $P$ . At another extreme, all the objects (whatever their type) might use the same algorithm to implement  $P$  (each object using its own instance of the algorithm).

**Atomicity Is a Local Property** The following theorem, which is due to M. Herlihy and J. Wing (1990), shows that atomicity is a local property. Intuitively, the fact that atomicity is local comes from the fact that it involves the real-time occurrence order on non-overlapping operations whatever the objects and the processes concerned by these operations. This point appears clearly in the proof of the theorem.

**Theorem 27** *A computation  $\widehat{OP}$  is atomic (linearizable) if and only if each object  $X$  involved in  $\widehat{OP}$  is atomic (i.e.,  $\widehat{OP}|X$  is atomic/linearizable).*

*Proof* The “ $\Rightarrow$ ” direction (only if) is an immediate consequence of the definition of atomicity: If  $\widehat{OP}$  is linearizable then, for each object  $X$  involved in  $\widehat{OP}$ ,  $\widehat{OP}|X$  is linearizable. So, the rest of the proof is restricted to the “ $\Leftarrow$ ” direction.

Given an object  $X$ , let  $\widehat{S}_X$  be a linearization of  $\widehat{OP}|X$ . It follows from the definition of atomicity that  $\widehat{S}_X$  defines a total order on the operations involving  $X$ . Let  $\rightarrow_X$  denote this total order. We construct an order relation  $\rightarrow$  defined on the whole set of operations of  $\widehat{OP}$  as follows:

1. For each object  $X$ :  $\rightarrow_X \subseteq \rightarrow$ ,
2.  $\xrightarrow{op} \subseteq \rightarrow$ .

Basically, “ $\rightarrow$ ” totally orders all operations on the same object  $X$ , according to  $\rightarrow_X$  (first item), while preserving  $\xrightarrow{op}$ , i.e., the real-time occurrence order on the operations (second item).

*Claim* “ $\rightarrow$  is acyclic”. This claim means that  $\rightarrow$  defines a partial order on the set of all the operations of  $\widehat{OP}$ .

Assuming this claim (see its proof below), it is thus possible to construct a sequential history  $\widehat{S}$  including all operations of  $\widehat{OP}$  and respecting  $\rightarrow$ . We trivially have  $\rightarrow \subseteq \rightarrow_S$ , where  $\rightarrow_S$  is the total order (on the operations) defined from  $\widehat{S}$ . We have the three following conditions: (1)  $\widehat{OP}$  and  $\widehat{S}$  are equivalent (they contain the same operations, and the operations of each process are ordered the same way  $\widehat{OP}$  and  $\widehat{S}$ ), (2)  $\widehat{S}$  is sequential (by construction) and legal (due to the first item stated above), and (3)  $\xrightarrow{op} \subseteq \rightarrow_S$  (due to the second item stated above and the relation inclusion  $\rightarrow \subseteq \rightarrow_S$ ). It follows that  $\widehat{OP}$  is linearizable.

*Proof of the Claim* We show (by contradiction) that  $\rightarrow$  is acyclic. Assume first that  $\rightarrow$  induces a cycle involving the operations on a single object  $X$ . Indeed, as  $\rightarrow_X$  is a total order, in particular transitive, there must be two operations  $op_i$  and  $op_j$  on  $X$  such that  $op_i \rightarrow_X op_j$  and  $op_j \xrightarrow{op} op_i$ .

- As  $op_i \rightarrow_X op_j$  and  $X$  is linearizable (respects object order, i.e., respects real-time order), it follows that  $op_i$  started before  $op_j$  terminated (otherwise, we will have  $op_j \rightarrow_X op_i$ ). Let us denote this as  $start[op_i] < term[op_j]$ .
- Similarly, it follows from the definition of  $\xrightarrow{op}$  that  $op_j \xrightarrow{op} op_i \Rightarrow term[op_j] < start[op_i]$ .

These two items contradict each other, from which we conclude that, if there is a cycle in  $\rightarrow$ , it cannot come from two operations  $op_i$  and  $op_j$ , and an object  $X$  such that  $op_i \rightarrow_X op_j$  and  $op_j \xrightarrow{op} op_i$ .

It follows that any cycle must involve at least two objects. To obtain a contradiction we show that, in this case, a cycle in  $\rightarrow$  implies a cycle in  $\rightarrow_H$  (which is acyclic). Let us examine the way the cycle could be obtained. If two consecutive edges of the cycle are due to just some  $\rightarrow_X$  or just  $\xrightarrow{op}$ , then the cycle can be shortened, as any of these relations is transitive. Moreover,  $op_i \rightarrow_X op_j \rightarrow_Y op_k$  is not possible for  $X \neq Y$ , as each operation is on only one object ( $op_i \rightarrow_X op_j \rightarrow_Y op_k$  would imply that  $op_j$  is on both  $X$  and  $Y$ ). So let us consider any sequence of edges of the cycle such that:  $op_1 \xrightarrow{op} op_2 \rightarrow_X op_3 \xrightarrow{op} op_4$ . We have:

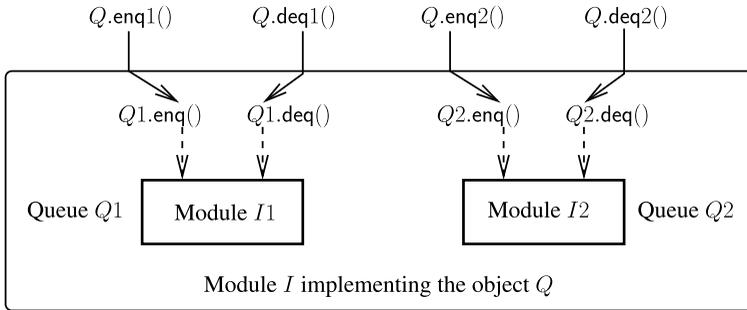
- $op_1 \xrightarrow{op} op_2 \Rightarrow term[op_1] < start[op_2]$  (definition of  $\xrightarrow{op}$ ).
- $op_2 \rightarrow_X op_3 \Rightarrow start[op_2] < term[op_3]$  (as  $X$  is linearizable).
- $op_3 \xrightarrow{op} op_4 \Rightarrow term[op_3] < start[op_4]$  (definition of  $\xrightarrow{op}$ ).

Combining these statements, we obtain  $term[op_1] < start[op_4]$ , from which we can conclude that  $op_1 \xrightarrow{op} op_4$ . It follows that any cycle in  $\rightarrow$  can be reduced to a cycle in  $\xrightarrow{op}$ , which is a contradiction as  $\xrightarrow{op}$  is an irreflexive partial order. (End of the proof of the claim.)  $\square$

**The Benefit of Locality** Considering an execution of a set of processes that access concurrently a set of objects, atomicity allows the programmer to reason as if all the operations issued by the processes on the objects were executed one after the other. The previous theorem is fundamental. It states that, to reason about sequential processes that access concurrent atomic objects, one can reason on each object independently, without losing the atomicity property of the whole computation.

**An Example** Locality means that atomic objects compose for free. As an example, let us consider two atomic queue objects  $Q1$  and  $Q2$ , each with its own implementation  $I1$  and  $I2$ , respectively (hence, the implementations can use different algorithms).

Let us define the object  $Q$  as the composition of  $Q1$  and  $Q2$  defined as follows (Fig. 16.6).  $Q$  provides processes with the four following operations  $Q.enq1()$ ,



**Fig. 16.6** Atomicity allows objects to compose for free

$Q.deq1()$ ,  $Q.enq2()$ , and  $Q.deq2()$ , whose effect is the same as  $Q1.enq()$ ,  $Q1.deq()$ ,  $Q2.enq()$  and  $Q2.deq()$ , respectively.

Thanks to locality, an implementation of  $Q$  consists simply in piecing together  $I1$  and  $I2$  *without any modification* to their code. As we will see in the next chapter, sequential consistency is not a local property. Hence, the previous object composition property is no longer true for sequential consistency.

## 16.4 Message-Passing Implementations of Atomicity

As atomicity requires that non-overlapping operations be executed in their “real-time” occurrence order, its implementations require a strong cooperation among processes for the execution of each operation. This cooperation can be realized with an appropriate underlying communication abstraction (such total order broadcast), or the association of a “server” process with each atomic object. Such implementations are described in this section.

### 16.4.1 Atomicity Based on a Total Order Broadcast Abstraction

**Principle** The total order broadcast abstraction has been introduced in Sect. 7.1.4 (where we also presented an implementation of it based on scalar clocks), and in Sect. 12.4 (where coordinator-based and token-based implementations of it have been described and proved correct).

This abstraction provides the processes with two operations, denoted  $to\_broadcast()$  and  $to\_deliver()$ , which allow them to broadcast messages and deliver these messages in the very same order. Let us recall that we then say that a process  $to$ -broadcasts and  $to$ -delivers a message.

An algorithm based on this communication abstraction, that implements an atomic object  $X$ , can be easily designed. Each process  $p_i$  maintains a copy  $x_i$  of the object  $X$ , and each time  $p_i$  invokes an operation  $X.oper()$ , it  $to$ -broadcasts a message describing this operation and waits until it  $to$ -delivers this message.

```

operation  $X.\text{oper}_\ell(\text{param})$  is
(1)  $\text{result}_i \leftarrow \perp$ ;
(2) to_broadcast OPERATION( $i, \text{oper}_\ell, \text{param}$ );
(3) wait ( $\text{result}_i \neq \perp$ );
(4) return( $\text{result}_i$ ).

when a message OPERATION( $j, \text{oper}_\ell, \text{param}$ ) is to-delivered do
(5)  $r \leftarrow x_i.\text{oper}_\ell(\text{param})$ ;
(6) if ( $i = j$ ) then  $\text{result}_i \leftarrow r$  end if.

```

**Fig. 16.7** From total order broadcast to atomicity

**The Algorithm** The corresponding algorithm is described in Fig. 16.7. Let  $\text{oper}_1(), \dots, \text{oper}_m()$  be the operations associated with the object  $X$ . If  $X$  is a read/write register, those are  $\text{read}()$  and  $\text{write}()$ . If  $X$  is a stack, they are  $\text{push}()$  and  $\text{pop}()$ , etc. It is assumed that each operation returns a result (which can be a default value for operations which have no data result such as  $\text{write}()$  or  $\text{push}()$ ). Moreover,  $\perp$  is a default control value which cannot be returned by an operation.

When a process  $p_i$  invokes  $X.\text{oper}_\ell(\text{param})$ , it to-broadcasts a message OPERATION() carrying the name of the operation, its input parameter, and the identity of the invoking process (line 2). Then,  $p_i$  is blocked until its operation has been applied to its local copy  $x_i$  of the object  $X$  (line 3). When this occurs, we have  $\text{result}_i \neq \perp$ , and  $p_i$  returns this value (line 4).

When  $p_i$  receives a message OPERATION(), it first applies the corresponding operation to its local copy  $x_i$  (line 5). Moreover, if it is the process that invoked this operation, it saves its result in  $\text{result}_i$ .

**Theorem 28** *The object  $X$  implemented by the algorithm described in Fig. 16.7 is atomic.*

*Proof* Let  $\widehat{OP}$  be an execution involving the object  $X$ . Let us first observe that, due the to-order broadcast abstraction defines a total order on all the operations invoked by the processes. This total order defines a sequence  $\widehat{S}$  which is trivially (a) equivalent to  $\widehat{OP}$ , (b) respects its partial order on operation, and (c) is legal. As each copy  $x_i$  of  $X$  is applied this sequence  $\widehat{S}$  of operations, each is a correct implementation of  $X$ .  $\square$

**Remark on the Locality Property of Atomicity** While the previous implementation has considered a single object  $X$ , it works for any number of objects. This is due to the fact that the atomicity consistency condition is a *local* property. Hence, if atomicity is implemented

- with the to-broadcast abstraction for some objects (possibly, each object being implemented with a specific implementation of to-broadcast),
- different approaches (such as those presented below) for the other objects,

then each object behaves atomically, and consequently the whole execution is atomic (Theorem 27).

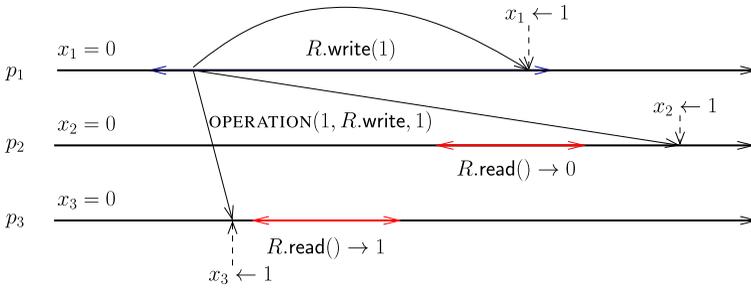


Fig. 16.8 Why read operations have to be to-broadcast

**The Case of Operations Which Do Not Modify the Objects** Let us consider the particular case where the object is a read/write register  $R$ . One could wonder why, as the read operations do not modify the object and each process  $p_i$  has a local copy  $x_i$  of  $R$ , it is necessary to to-broadcast the invocations of  $read()$ .

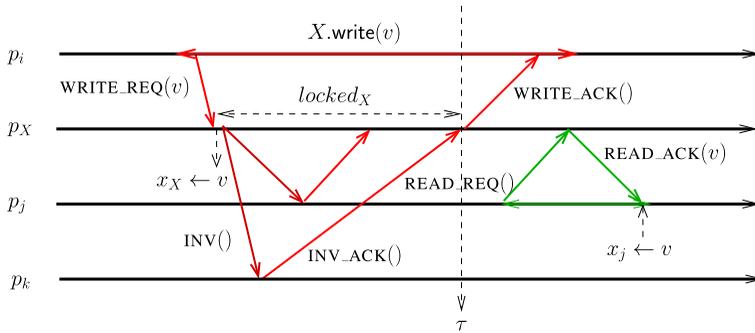
To answer this question, let us consider Fig. 16.8. The register  $R$  is initialized to the value 0. Process  $p_1$  invokes  $R.write(1)$ , and consequently issues an underlying to-broadcast of  $OPERATION(1, write_\ell, 1)$ . When a process  $p_i$  to-delivers this message, it assigns the new value 1 to  $x_i$ . After it has to-delivered the message  $OPERATION(1, write_\ell, 1)$ ,  $p_3$  invokes  $R.read()$  and, as  $x_i = 1$ , it returns the value 1. Differently, the message  $OPERATION(1, write_\ell, 1)$  sent by  $p_1$  to  $p_2$  is slow, and  $p_2$  invokes  $R.read()$  before this message has been to-delivered. This read consequently returns the current value of  $x_2$ , i.e., the initial value 0.

As the invocation of  $R.read()$  by  $p_3$  terminates before the invocation of  $R.read()$  by  $p_2$  starts we have  $(R.read()$  by  $p_3) \xrightarrow{op} (R.read()$  by  $p_2)$ , and consequently, the read by  $p_3$  has to be ordered (in  $\widehat{S}$ ) before the read by  $p_2$ . But then, the sequence  $\widehat{S}$  cannot be legal. This is because the read by  $p_3$  obtains the new value, while the read by  $p_2$  (which occurs later with respect to real time as formally captured by  $\xrightarrow{op}$ ) obtains the initial value, which has been overwritten (as witnessed by the read of  $p_3$ ).

Preventing such incorrect executions requires all operations to be totally ordered with the to-broadcast abstraction. Hence, when implementing atomicity with to-broadcast, even the operations which do not modify the object have to participate in the to-broadcast.

### 16.4.2 Atomicity of Read/Write Objects Based on Server Processes

A way to implement atomic read/write registers without using an underlying total order broadcast consists in associating a server process (also called manager) with a set of registers. At one extreme, it is possible that a single server manages all the registers, and at the other extreme, it is possible to have an independent server per



**Fig. 16.9** Invalidation-based implementation of atomicity: message flow

register. This is a consequence of the fact that atomicity is a local property: If  $X$  and  $Y$  are atomic, their implementations can be independent (i.e., the manager of  $X$  and the manager of  $Y$  never need to cooperate).

Hence, without loss of generality, we consider in the following that there is a single register  $X$  managed by a single server, denoted  $p_X$ . In addition to  $p_X$ , each process  $p_i$  has a local copy  $x_i$  of  $X$ . The local copy at  $p_X$  is sometimes called *primary copy*. To simplify the presentation (and again without loss of generality) we consider that the role of the server  $p_X$  is only to manage  $X$  (i.e., it does not involve operations on  $X$ ).

The role of  $p_X$  is to ensure atomicity by managing the local copies so that (a) each read operation returns a correct value, and (b)—when possible—read operations are local, i.e., do not need to send or receive messages. To attain this goal, two approaches are possible.

- **Invalidation.** In this case, at each write of  $X$ , the manager  $p_X$  invalidates all the local copies of  $X$ .
- **Update.** In this case, at each write of  $X$ , the manager  $p_X$  updates all the local copies of  $X$ .

The next two sections develop each of these approaches.

### 16.4.3 Atomicity Based on a Server Process and Copy Invalidation

**Principle and Flow of Messages** The principle of an invalidation-based implementation of atomicity is described in Fig. 16.9.

When a process  $p_i$  invokes  $X.write(v)$ , it sends a request message ( $WRITE\_REQ(v)$ ) to the manager of  $X$ , which updates its local copy and forwards an invalidation message to the processes which have a copy of  $X$ . When a process receives this message, it invalidates its copy and sends by return to  $p_X$  an acknowledgment (message  $INV\_ACK()$ ). When it has received all the acknowledgments,  $p_X$  informs

$p_i$  that there are no more copies of the previous value and consequently the write operation can terminate (message `WRITE_ACK()`).

Moreover, during the period starting at the reception of the message `WRITE_REQ()` and finishing at the sending of the corresponding `WRITE_ACK()`,  $p_X$  becomes locked with respect to  $X$ , which means that it delays the processing of all the message `WRITE_REQ()` and `READ_REQ()`. These messages can be processed only when  $p_X$  is not locked.

Finally, when the manager  $p_X$  receives a message `READ_REQ()` from a process  $p_j$ , it sends by return the current copy of  $X$  to  $p_j$ . When  $p_j$  receives it, it writes it in its copy  $x_j$  and, if it reads again  $X$ , it uses this value until it is invalidated. Hence, some read operations needs two messages, while others are purely local.

It is easy to see that the resulting register object  $X$  is atomic. All the write operations are totally ordered by  $p_X$ , and each read operation obtains the last written value. The fact that the meaning of “last” is with respect to real time (as captured by  $\xrightarrow{op}$ ) follows from the following observation: When  $p_X$  stops being locked (time  $\tau$  in Fig. 16.9), only the current writer and itself have the last value of  $X$ .

**The Algorithm** The invalidation-based algorithm is described in Fig. 16.10. This code is the direct operational translation of the previous principle. The manager  $p_X$  is such that  $x_X$  is initialized to the initial value of  $X$ .

The local array  $hlv_X[1..n]$  (for *hold last value*) is such that  $hlv_X[i]$  is true if and only if  $p_i$  has a copy of the last value that has been written. The initialization of this Boolean array is such that  $hlv_X[i]$  is true if  $x_i$  is initially equal to  $x_X$ . If  $hlv_X[i]$  is initially false,  $x_i = \perp$ , which locally means that  $p_i$  does not have the last value of  $X$ .

The messages `WRITE_REQ()` and `READ_REQ()` remain in their input buffer until they can be processed. It is assumed that every message is eventually processed (as  $p_X$  is the only process which receives these messages, this is easy to ensure).

### 16.4.4 Introducing the Notion of an Owner Process

**The Notion of an Owner Process** It is possible to enrich the previous algorithm by introducing the notion of a process owner. A process starts being the *owner* of a register when it terminates a write on this register and stops being the owner when this object is written or read by another process. This means that, when a process  $p_i$  owns a register  $X$ , it can issue several writes on  $X$  without contacting the manager of  $X$ . Only its last write is “meaningful” in the sense that this write overwrites the previous writes it has issued since it started being the current owner.

As we can see, as it is related to the pattern of read and write operations issued by the processes, this ownership notion is essentially dynamic. As soon as a process reads the object, there is no more owner until the next write operation. It is assumed that, initially, there is no owner process and only the manager  $p_X$  has the initial value of  $X$ .

```

===== on the side of process  $p_i$ ,  $1 \leq i \leq n$  =====
operation  $X.write(v)$  is
(1)  $x_i \leftarrow v$ ;
(2) send  $WRITE\_REQ(v)$  to  $p_X$ ;
(3) wait  $WRITE\_ACK()$  from  $p_X$ .

operation  $X.read()$  is
(4) if  $(x_i = \perp)$ 
(5)   then send  $READ\_REQ()$  to  $p_X$ ;
(6)     wait  $READ\_ACK(v)$  from  $p_X$ ;
(7)      $x_i \leftarrow v$ 
(8)   end if;
(9)   return $(x_i)$ .

when a message  $INV(X)$  is received from  $p_X$  do
(10)  $x_i \leftarrow \perp$ ;
(11) send  $ACK\_INV()$  to  $p_X$ .

===== on the side of the server  $p_X$  =====
when a message  $READ\_REQ()$  is received from  $p_i$  do
(12) wait  $(\neg locked_X)$ ;
(13) send  $READ\_ACK(x_X)$  to  $p_i$ ;
(14)  $hlv_X[i] \leftarrow true$ .

when a message  $WRITE\_REQ(v)$  is received from  $p_i$  do
(15) wait  $(\neg locked_X)$ ;
(16)  $locked_X \leftarrow true$ ;  $x_X \leftarrow v$ ;
(17) for each  $j$  such that  $hlv_X[j]$  do send  $INV()$  to  $p_j$  end for;
(18) wait  $(ACK\_INV)$  received from each  $j$  such that  $hlv_X[j]$ ;
(19)  $hlv_X[1..n] \leftarrow [false, \dots, false]$ ;  $hlv_X[i] \leftarrow true$ ;
(20) send  $WRITE\_ACK()$  to  $p_i$ ;
(21)  $locked_X \leftarrow false$ .

```

**Fig. 16.10** Invalidation-based implementation of atomicity: algorithm

**The Basic Pattern** The basic pattern associated with the ownership notion is the following one:

- A process  $p_i$  writes  $X$ : It becomes the owner, which entails the invalidation of all the copies of the previous value of  $X$ . Moreover,  $p_i$  can continue to update its local copy of  $X$ , without informing  $p_X$ , until another process  $p_j$  invokes a read or a write operation.
- If the operation issued by  $p_j$  is a write, it becomes the new owner, and the situation is as in the previous item.
- If the operation issued by  $p_j$  is a read, the current owner  $p_i$  is demoted, and it is no longer the owner of  $X$  and is downgraded from the writing/reading mode to the reading mode only. It can continue reading its copy  $x_i$  (without passing through the manager  $p_X$ ), but its next write operation will have to be managed by  $p_X$ . Moreover,  $p_i$  has to send the current value of  $x_i$  to  $p_X$ , and from now on  $p_X$  knows the last value of  $X$ .

```

operation  $X.write(v)$  is
(1) if ( $\neg owner_i$ )
(2)   then send WRITE_REQ( $v$ ) to  $p_X$ ;
(3)     wait WRITE_ACK() from  $p_X$ ;
(4)      $owner_i \leftarrow true$ 
(5)   end if;
(6)  $x_i \leftarrow v$ .

operation  $X.read()$  is
(7) if ( $x_i = \perp$ )
(8)   then send READ_REQ() to  $p_X$ ;
(9)     wait READ_ACK( $v$ ) from  $p_X$ ;
(10)     $x_i \leftarrow v$ 
(11)  end if;
(12) return( $x_i$ ).

when a message DOWNGRADE_REQ( $type$ ) is received from  $p_X$  do
(13)  $owner_i \leftarrow false$ ;
(14) if ( $type = w$ ) then  $x_i \leftarrow \perp$  end if;
(15) send DOWNGRADE_ACK( $x_i$ ) to  $p_X$ .

```

**Fig. 16.11** Invalidation and owner-based implementation of atomicity (code of  $p_i$ )

If another process  $p_k$  wants to read  $X$ , it can then obtain from  $p_X$  the last value of  $X$ , and keeps reading its new copy until a new write operation invalidates all copies.

**The Read and Write Operations** These operations are described in Fig. 16.11. Each process  $p_i$  manages an additional control variable, denoted  $owner_i$ , which is true if and only if  $p_i$  is the current owner of  $X$ .

If, when  $p_i$  invokes  $X.write(v)$ ,  $p_i$  is the current owner of  $X$ , it has only to update its local copy  $x_i$  (lines 1 and 6). Otherwise it becomes the new owner (line 4) and sends a message WRITE\_REQ( $v$ ) to the manager  $p_X$  so that it downgrades the previous owner, if any (line 2). When this downgrading has been done (line 3),  $p_i$  writes  $v$  in its local copy  $x_i$ , and the write terminates.

The algorithm implementing the read operation is similar to the one implementing the write operation. If there is a local copy of  $X$  ( $x_i \neq \perp$ ),  $p_i$  returns it. Otherwise, it sends a message READ\_REQ() to  $p_X$  in order to obtain the last value written into  $X$ . This behavior is described by lines 7–12.

The lines 13–15 are related to the management of the ownership of  $X$ . When the manager  $p_X$  discovers that  $p_i$  is no longer the owner of  $X$ ,  $p_X$  sent to  $p_i$  a message DOWNGRADE\_REQ( $type$ ), where  $type = w$  if the downgrading is due to a write operation, and  $type = r$  if it is due to a read operation. Hence, when it receives DOWNGRADE\_REQ( $type$ ),  $p_i$  first sets  $owner_i$  to false. Then it sends by return to  $p_X$  an acknowledgment carrying its value of  $x_i$  (which is the last value that has been written if  $type = r$ ).

```

when a message WRITE_REQ( $v$ ) is received from  $p_i$  do
(16) wait ( $\neg locked_X$ );
(17)  $locked_X \leftarrow true$ ;
(18) let  $send\_to = \{j \text{ such that } hlv_X[j]\} \setminus \{i\}$ ;
(19) for each  $j \in send\_to$  do send DOWNGRADE_REQ( $w$ ) to  $p_j$  end for;
(20) wait (DOWNGRADE_ACK() received from each  $p_j$  such that  $j \in send\_to$ );
(21)  $x_X \leftarrow v$ ;  $owner_X \leftarrow i$ ;
(22)  $hlv_X[1..n] \leftarrow [false, \dots, false]$ ;  $hlv_X[i] \leftarrow true$ ;
(23) send WRITE_ACK() to  $p_i$ ;
(24)  $locked_X \leftarrow false$ .

when a message READ_REQ() is received from  $p_i$  do
(25) wait ( $\neg locked_X$ );
(26)  $locked_X \leftarrow true$ ;
(27) if ( $owner_X \neq \perp$ )
(28)   let  $k = owner_X$ ;
(29)   send DOWNGRADE_REQ( $r$ ) to  $p_k$ ;
(30)   wait (DOWNGRADE_ACK( $v$ ) received from  $p_k$ );
(31)    $x_X \leftarrow v$ ;  $owner_X \leftarrow \perp$ ;
(32) end if;
(33)  $hlv_X[i] \leftarrow true$ ;
(34) send READ_ACK( $x_X$ ) to  $p_i$ ;
(35)  $locked_X \leftarrow false$ .

```

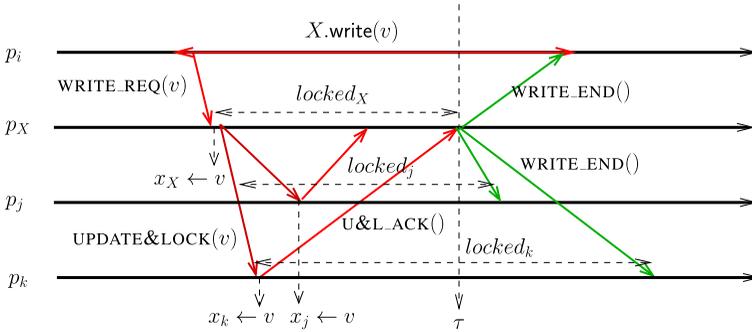
**Fig. 16.12** Invalidation and owner-based implementation of atomicity (code of the manager  $p_X$ )

**Behavior of the Manager  $p_X$**  The process  $p_X$  manages an additional variable  $owner_X$ , which contains the identity of the current owner of  $X$ . If there is no current owner, then  $owner_X = \perp$ . The behavior of  $p_X$  is described in Fig. 16.12. As in the previous algorithm, the Boolean  $locked_X$  is used to ensure that  $p_X$  processes one write or read operation at a time.

When  $p_X$  receives a message WRITE\_REQ( $v$ ) from a process  $p_i$ , it first sends a message DOWNGRADE\_REQ( $w$ ) to all the processes that have a copy of  $X$ , so that they invalidate their copies of  $X$  (lines 19–20). Moreover, as the invalidation is due to a write operation, the parameter *type* is set to  $w$  (because, in this case, no value has to be returned). Then,  $p_X$  updates its local context, namely,  $x_X$ ,  $owner_X$ , and  $hlv_X[1..n]$  (lines 21–22), before returning an acknowledgment to  $p_i$ , indicating the write has terminated (line 23).

Let us notice that, when  $owner_X = \perp$ , the local variable  $x_X$  of the manager  $p_X$  has the last value of  $X$ . Hence, when  $p_X$  receives a message READ\_REQ() from a process  $p_i$ , it sends by return to  $p_i$  the value of  $x_X$  if  $owner_X = \perp$ , and updates  $hlv_X[i]$  (lines 27 and 33–34). Differently, if  $owner_X \neq \perp$ ,  $p_X$  has first to downgrade the current owner from the writing mode to the reading mode and obtains from it the last value of  $X$  (lines 27–32).

It is easy to see that the number of control messages (involved in read and write operations), which are sent consecutively, varies from 0 (the last value is in the local variable  $x_i$  of the invoking process) up to 4 (namely, the sequence



**Fig. 16.13** Update-based implementation of atomicity

$WRITE\_REQ()$  – or  $READ\_REQ()$  –,  $DOWNGRADE\_REQ()$ ,  $DOWNGRADE\_ACK()$ , and  $WRITE\_ACK()$  – or  $READ\_ACK()$  –).

### 16.4.5 Atomicity Based on a Server Process and Copy Update

**Principle** The idea is very similar to the one used in the invalidation approach. It differs in the fact that, when the manager  $p_X$  learns a new value, instead of invalidating copies, it forwards the new value to the other processes.

To illustrate this principle, let us consider Fig. 16.13. When  $p_X$  receives a message  $WRITE\_REQ(v)$ , it forwards the value  $v$  to all the other processes, and (as previously) becomes locked until it has received all the corresponding acknowledgments, which means that it processes sequentially all write requests. When a process  $p_i$  receives a message  $UPDATE\&LOCK(v)$ , it updates  $x_i$  to  $v$ , sends an acknowledgment to  $p_X$ , and becomes locked. When  $p_X$  has received all the acknowledgments, it knows that all processes have the new value  $v$ . This time is denoted  $\tau$  on the figure. When this occurs,  $p_X$  sends a message to all the processes to inform them that the write has terminated. When a process  $p_i$  receives this message, it becomes unlocked, which means that it can again issue read or write operations.

It follows that, differently from the invalidation approach, all reads are purely local in the update approach (they send and receive messages).

**The Algorithm** The code of the algorithm, described in Fig. 16.14, is similar to the previous one. This algorithm considers that there are several atomic objects  $X$ ,  $Y$ , etc., each managed by its own server  $p_X$ ,  $p_Y$ , etc. The local Boolean variable  $locked_i[X]$  is used by  $p_i$  to control the accesses of  $p_i$  to  $X$ .

For each object  $X$ , the cooperation between the server  $p_X$  and the application processes  $p_i$ ,  $p_j$ , etc., is locally managed by the Boolean variables  $locked_i[X]$ ,  $locked_j[X]$ , etc. Thanks to this cooperation, the server  $p_X$  of each object  $X$  guarantees that  $X$  behaves atomically. As proved in Theorem 27, the set of servers  $p_X$ ,  $p_Y$ , etc., do not have to coordinate to ensure that the whole execution is atomic.

```

===== on the side of process  $p_i$ ,  $1 \leq i \leq n$  =====
operation  $X.write(v)$  is
(1) wait ( $\neg locked_i[X]$ );
(2)  $x_i \leftarrow v$ ;
(3) send WRITE_REQ( $v$ ) to  $p_X$ ;
(4) wait WRITE_END() from  $p_X$ .

operation  $X.read()$  is
(5) wait ( $\neg locked_i[X]$ );
(6) return( $x_i$ ).

when a message UPDATE&LOCK( $v$ ) is received from  $p_X$  do
(7)  $x_i \leftarrow v$ ;
(8)  $locked_i[X] \leftarrow true$ ;
(9) send U&L_ACK() to  $p_X$ .

when a message WRITE_END() is received from  $p_X$  do
(10)  $locked_i[X] \leftarrow false$ .

===== on the side of the server  $p_X$  =====
when a message WRITE_REQ( $v$ ) is received from  $p_i$  do
(11) wait ( $\neg locked_X$ );
(12)  $locked_X \leftarrow true$ ;
(13) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do send UPDATE&LOCK( $v$ ) to  $p_j$  end for;
(14) wait (U&L_ACK() received from each  $j \in \{1, \dots, n\} \setminus \{i\}$ );
(15) for each  $j \in \{1, \dots, n\}$  do send WRITE_END() to  $p_j$  end for;
(16)  $locked_X \leftarrow false$ .

```

Fig. 16.14 Update-based algorithm implementing atomicity

It is easy to see that, as in the invalidation-based algorithm,  $p_X$  serializes all write operations, and that no read operation obtains an overwritten value.

## 16.5 Summary

This chapter first introduced the concept of a distributed shared memory. It has then presented, both from intuitive and formal points of view, the atomicity consistency condition (also called linearizability). It then showed that atomicity is a consistency condition that allows objects to be composed for free (a set of objects is atomic if and only if each object is atomic). Then the chapter presented three message-passing algorithms that implement atomic objects: a first one based on a total order broadcast abstraction, a second one based on an invalidation technique, and a third one based on an update technique.

## 16.6 Bibliographic Notes

- Atomic consistency is the implicit consistency condition used for von Neumann machines. A formalization and an associated theory of atomicity are given by L.

- Lamport in [228, 229]. An axiomatic presentation of atomicity for asynchronous hardware is given by J. Misra in [262].
- The notion of linearizability, which generalizes atomicity to any type of object defined by a sequential specification, is due to M. Herlihy and J. Wing [183]. The theorem stating that atomic objects compose for free (i.e., atomicity is a local property, Theorem 27) is due to them.
  - The invalidation-based algorithm that implements atomic consistency presented in Sect. 16.4.3 is a simplified version of an algorithm managing a distributed shared memory, which is due to K. Li and P. Hudak [234]. An early introductory survey on virtual memory systems can be found in [106].
  - The update-based algorithm presented in Sect. 16.4.5, which implements atomic consistency, is from [32] (where it is used to implement data consistency in a programming language dedicated to distributed systems).
  - Consistency conditions for objects have given rise to many proposals. They are connected to cache consistency protocols [5, 72, 119]. The interested reader will find introductory surveys on consistency conditions in [3, 4, 286, 301, 323].
  - Numerous consistency conditions weaker than atomicity have been investigated: serializability [289], causal consistency [10], hybrid consistency [136], tuples space [69], normality [151], non-sequential consistency conditions [21], slow memory [194], lazy release consistency [204], timed consistency [371, 372], pipelined RAM (PRAM) consistency [237], to cite a few.
  - Generic algorithms which implement several consistency conditions are described in [200, 212].
  - Sequential consistency, which was introduced by L. Lamport [227], is (with atomicity) one of the most important consistency conditions. It is addressed in the next chapter. Its connection with linearizability is investigated in [23, 313].
  - Analysis of strong consistency conditions and their implementations can be found in [157, 256].
  - An object operation is *polyadic* if it is on several objects at the same time. As an example, the queue operation  $(Q, Q').\text{add}()$ , which adds  $Q'$  at the end of  $Q$ , is polyadic. Investigation of consistency conditions for objects whose operations are polyadic can be found in [267, 326].

## 16.7 Exercises and Problems

1. Design an algorithm that implements read/write atomic objects, and uses update (instead of invalidation) and the process ownership notion.
2. Extend the algorithm described in Figs. 16.11 and 16.12, which implements atomicity with invalidation and the ownership notion, to obtain an algorithm that implements a distributed shared memory in which each atomic object is a page of a classical shared virtual memory.  
Solution in [234].

3. When considering the update-based implementation of atomicity described in Fig. 16.14, describe a scenario where a process  $p_i$ , which has issued a write operation, becomes locked because of a write operation issued by another process  $p_j$ . Which write operation is ordered first?
4. Let us consider a shared memory computation  $\widehat{OP} = (OP, \xrightarrow{op})$ . Does the problem of checking if  $\widehat{OP}$  is atomically consistent belong to the complexity class P or NP?

# Chapter 17

## Sequential Consistency

This chapter is on sequential consistency, a consistency condition for distributed shared memory, which is weaker than atomicity (linearizability). After having defined sequential consistency, this chapter shows that it is not a local property. Then, it presents two theorems which are of fundamental importance when one has to implement sequential consistency on top of asynchronous message-passing systems. Finally, the chapter presents and proves correct several distributed algorithms that implement sequential consistency. Sequential consistency was introduced by L. Lamport (1979).

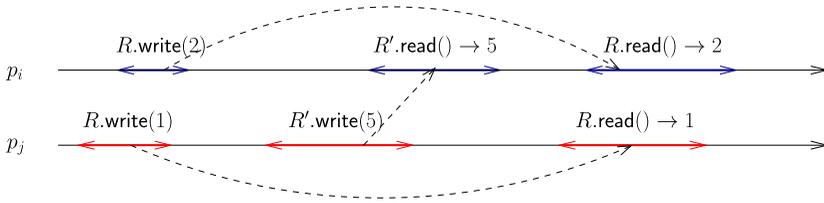
**Keywords** Causal consistency · Concurrent object · Consistency condition · Distributed shared memory · Invalidation · Logical time · Manager process · OO constraint · Partial order on operations · Read/write register · Sequential consistency · Server processes · Shared memory abstraction · Total order broadcast abstraction · WW constraint

### 17.1 Sequential Consistency

#### 17.1.1 Definition

**Intuitive Definition** Let us consider an execution of a set of processes accessing a set of concurrent objects. The resulting computation  $\widehat{OP}$  is sequentially consistent if it could have been produced (with the help of a scheduler) by executing it on a monoprocessor system. This means that, in a sequentially consistent execution, the operations of all the processes appear as if they have been executed in some sequential order, and the operations of each process appear in this total ordering in the order specified by its program.

**Formal Definition: Sequentially Consistent Computation** The formalism used below is the one that was introduced in Sect. 16.2.2, where it was shown that an execution of processes accessing concurrent objects is a partial order on the operations issued by these processes. The definition of sequential consistency is based on the “process order” relation that provides us with a formal statement that, each process being sequential, the operations it issued are totally ordered.



**Fig. 17.1** A sequentially consistent computation (which is not atomic)

A computation  $\widehat{OP}$  is sequentially consistent if there exists a sequential computation  $\widehat{S}$  such that:

- $\widehat{OP}$  and  $\widehat{S}$  are equivalent (i.e., no process can distinguish  $\widehat{OP}$  and  $\widehat{S}$ ), and
- $\widehat{S}$  is legal (the specification of each object is respected).

These two items are the same as for atomicity (which includes a third one related to real-time order). Atomicity is sequential consistency plus the fact that the operations that do not overlap (whatever the processes that issued them and the objects they access) are ordered in  $\widehat{S}$  according to their real-time occurrence order. Trivially, any computation that is atomic is also sequentially consistent, while the reverse is not true.

This is illustrated in the example depicted in Fig. 17.1. The computation described in this figure involves two processes and two registers,  $R$  and  $R'$ . It is sequentially consistent because there is a sequence  $\widehat{S}$  that respects (a) process order for each process, (b) the sequential specification of each register, and (c) is composed of the operations (parameter and result) as the “real” computation. The sequence  $\widehat{S}$  consists of the execution of  $p_j$  followed by that of  $p_i$ , namely

$$R.write_j(1), R'.write_j(5), R.read_j() \rightarrow 1, R.write_i(2), R'.read_i() \rightarrow 5, R.read_i() \rightarrow 2.$$

As we can see, this “witness” execution  $\widehat{S}$  does not respect the real time occurrence order of the operations from different processes. Hence, it is not atomic.

The dotted arrow depicts what is called the *read-from* relation when the shared objects are read/write registers. It indicates, for each read operation, the write operation that wrote the value read. The read-from relation is the analog of the send/receive relation in message-passing systems, with two main differences: (1) not all values written are read, and (2) a value written can be read by several processes and several times by the same process.

An example of a computation which is not sequentially consistent is described in Fig. 17.2. Despite the fact that each value that is returned by a read operation has been previously written, it is not possible to build there a sequence  $\widehat{S}$  which respects both process order and the sequential specification of each register.

Actually, determining whether a computation made up of processes accessing concurrent read/write registers is sequentially consistent is an NP-complete problem (Taylor, 1983). This result rules out the design of efficient algorithms that would

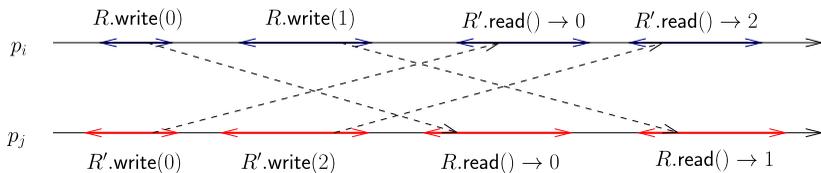
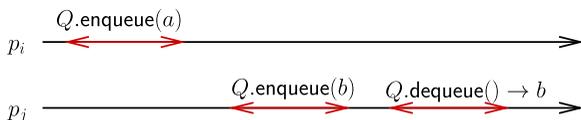


Fig. 17.2 A computation which is not sequentially consistent

Fig. 17.3 A sequentially consistent queue



implement sequential consistency and no more. As we will see, efficient algorithms for sequential consistency implement more than this condition. (As shown by the simple algorithms implementing atomicity, which is stronger than sequential consistency.)

**Formal Definition: Sequentially Consistent Object** Given a computation  $\widehat{OP}$  and an object  $X$ , this object is sequentially consistent in  $\widehat{OP}$  if the computation  $\widehat{OP}|X$  is sequentially consistent. (Let us recall that  $\widehat{OP}|X$  is  $\widehat{OP}$  from which have been suppressed all the operations which are not on  $X$ .)

### 17.1.2 Sequential Consistency Is Not a Local Property

While atomic consistency is a local property, and consequently atomic objects compose for free (see Sect. 16.3), this is no longer the case for sequentially consistent objects. The following counter-example proves this claim.

Let us consider a computation with two processes accessing queues. Each queue is accessed by the usual operations denoted  $enqueue()$ , which adds an item at the head of the queue, and  $dequeue()$ , which withdraws from the queue the item at its tail and returns it ( $\perp$  is returned if the queue is empty).

The computation described in Fig. 17.3, which involves one queue denoted  $Q$ , is sequentially consistent. This follows from the fact that the sequence

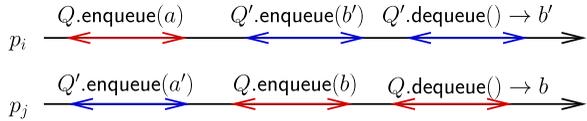
$$\widehat{S} = Q.enqueue_j(b), \quad Q.dequeue_j() \rightarrow b, \quad Q.enqueue_i(b)$$

is legal and respects the order of the operations in each process. As this computation involves a single object, trivially  $Q$  is sequentially consistent.

Let us now consider the computation described in Fig. 17.4, which involves two queues,  $Q$  and  $Q'$ . It is easy to see that each queue is sequentially consistent. The existence of the previous sequence  $S$  proves it for  $Q$ , and the existence of the following sequence proves it for  $Q'$ ,

$$\widehat{S}' = Q'.enqueue_i(a'), \quad Q'.dequeue_i() \rightarrow a', \quad Q'.enqueue_j(b').$$

**Fig. 17.4** Sequential consistency is not a local property



But the computation as a whole is not sequentially consistent: it is not possible to order all the operations in such a way that the corresponding sequence  $\widehat{SEQ}$  would be such that (a) the operations appear in their process order, and (b) both  $Q$  and  $Q'$  satisfy their sequential specification. This is because, to produce such a sequence, we have to order first either  $Q.enqueue_i(a)$  or  $Q'.enqueue_i(a')$ , e.g.,  $Q.enqueue_i(a)$ . Then, whatever the second operation placed in the sequence, we need to have  $Q.dequeue_j(a)$ , instead of  $Q.dequeue_j(b)$ , for  $Q$  to behave correctly.

It follows that, even if each object is sequentially consistent, the whole computation is not, i.e., sequential consistency is not a local property. From an implementation point of view, this will clearly appear in the implementation where a specific manager is associated with each object (see Sect. 17.4). The managers are required to cooperate to ensure that the computation is sequentially consistent. (As we have seen in the previous chapter, atomic consistency does not need this cooperation.)

### 17.1.3 Partial Order for Sequential Consistency

**Partial Order on Read and Write Operations** As we have seen, differently from atomicity (linearizability), sequential consistency does not refer to the real-time order on the operations issued by processes.

Let us consider the case where the objects are read/write registers. Moreover, to simplify the presentation, and without loss of generality, it is assumed that there is an initial write on each register, and all the values written into a register are different.

Let  $w_j(X, a)$  denotes the write of  $a$  into  $X$  by  $p_j$ , and  $r_i(X)a$  denotes a read of  $X$  by  $p_i$  which returns the value  $a$ . When  $X$  and  $a$  are not relevant, these operations are denoted  $w_i()$  and  $r_i()$ . As already indicated, this means that  $r_i(X)a$  reads from  $w_i(X, a)$ . The set of all the “read from” pairs is captured by the relation denoted  $\xrightarrow{rf}$ , i.e., if  $r_i(X)a$  reads from  $w_i(X, a)$ , we have  $w_i(X, a) \xrightarrow{rf} r_i(X)a$ .

In such a context, the relation  $\xrightarrow{po}$  defining a read/write computation is defined as follows:  $op_1$  and  $op_2$  being two operations on registers,  $op_1 \xrightarrow{po} op_2$  if

- $op_1$  and  $op_2$  have been issued by the same process with  $op_1$  first (process order relation), or
- $op_1 = w_i(X, a)$  and  $op_2 = r_i(X)a$  (read from relation), or
- $\exists op$  such that  $op_1 \xrightarrow{po} op$  and  $op \xrightarrow{po} op_2$  (transitivity).

Let us recall that two operations  $op_1$  and  $op_2$  such that  $\neg(op_1 \xrightarrow{po} op_2) \wedge \neg(op_2 \xrightarrow{po} op_1)$  are said to be *concurrent* or *independent*.

**A Definition of Legality Customized for Read/Write Registers** The notion of legality was introduced in the previous chapter to capture the fact a sequential computation satisfies the sequential specification of the objects it accesses. As we consider here read/write objects (registers), the notion of legality is extended to any computation (not only sequential computations) and reformulated in terms of read/write registers.

A read/write-based computation  $\widehat{OP}$  is *legal* if, for each read operation  $r_i(X)a$ , we have:

- $\exists w_j(X, a)$ , and
- $\nexists w_k(X, b)$  such that  $[w_j(X, a) \xrightarrow{op} w_k(X, b)] \wedge [w_k(X, b) \xrightarrow{op} r_i(X)a]$ .

The first item states that any value returned by a read operation has previously been written, while the second item states that no read operation returns an overwritten value.

### 17.1.4 Two Theorems for Sequentially Consistent Read/Write Registers

Considering processes accessing concurrent objects which are read/write registers, this section states two theorems which are important when designing algorithms which implement sequential consistency. These theorems are due to M. Mizuno, M. Raynal, and J.Z. Zhou (1994).

**Two Additional Constraints** Since checking sequential consistency is an NP-complete problem, the idea is to find additional constraints which can be (relatively easily) implemented and simplify the design of algorithms ensuring sequential consistency. Two such constraints have been identified.

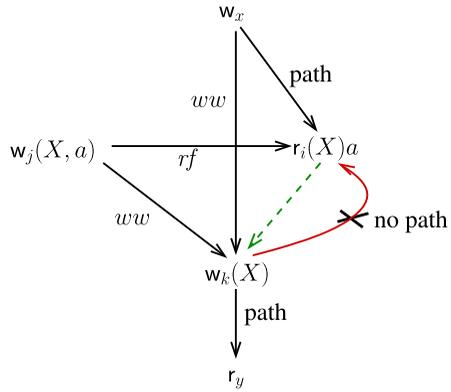
- **Write/write (WW) constraint.** All write operations are totally ordered.
- **Object-ordered (OO) constraint.** Conflicting operations on each object are totally ordered. (A write operation conflicts with any read operation and any other write operation, while a read operation conflicts with any write operation.)

It is important to see that the constraint WW imposes an order on all write operations whatever the objects they are on, while the constraint OO on each object taken individually.

When considering an execution that satisfies the WW constraint, the total order relation created by WW is denoted  $\xrightarrow{ww}$ . Similarly, the order relation created by the constraint OO on an object  $X$  is denoted  $\xrightarrow{oo(X)}$ .

**Theorem 29** *Let  $\widehat{OP}$  be a computation that satisfies the WW constraint. If  $\widehat{OP}$  is legal, it is sequentially consistent.*

**Fig. 17.5** Part of the graph  $G$  used in the proof of Theorem 29



*Proof* Assuming that  $\widehat{OP}$  is legal and satisfies the constraint WW, let us consider the directed graph denoted  $G$  and defined as follows. Its vertices are the operations, and there is an edge from  $op_1$  to  $op_2$  if (a)  $op_1 \xrightarrow{ww} op_2$ , or (b)  $op_1 \xrightarrow{rf} op_2$ , or (c)  $op_1$  and  $op_2$  have been issued by the same process with  $op_1$  first (process order at each  $p_i$ ). As  $\widehat{OP}$  is acyclic, so is  $G$ . The proof consists of two steps.

First step. For each object  $X$ , we add edges to  $G$  so that all (conflicting) operations on  $X$  are ordered, and these additions preserve the acyclicity and legality of the modified graph  $G$ .

Let us consider a register  $X$ . As all write operations on  $X$  are totally ordered, we have only to order the read operations on  $X$  with respect to the write operations on  $X$ . Let  $r_i(X)a$  be a read operation. As  $r_i(X)a$  is legal (assumption), there exists  $w_j(X)a$  such that  $w_j(X)a \xrightarrow{rf} r_i(X)a$ . (See Fig. 17.5, where the label associated with an edge explains it.)

Let  $w_k(X)$  be any write operation such that  $w_j(X)a \xrightarrow{ww} w_k(X)$ . For any such  $w_k(X)$ , let us add an edge from  $r_i(X)a$  to  $w_k(X)$  (dotted edge in the figure). This addition cannot create a cycle because, due to the legality of  $G$ , there is no path from  $w_k(X)$  to  $r_i(X)a$ . Let us now show that this addition preserves the legality of  $G$ .

Legality can be violated if adding an edge creates a path from some write operation  $w_x$  to some read operation  $r_y$ . Hence, let us assume that the addition of the edge from  $r_i(X)a$  to  $w_k(X)$  creates a new path from  $w_x$  to  $r_y$ . It follows that, before adding the edge from  $r_i(X)a$  to  $w_k(X)$ , there was a path from  $w_x$  to  $r_i(X)a$  and a path from  $w_k(X)$  to  $r_y$ . Due to the relation  $\xrightarrow{ww}$ , two cases have to be analyzed.

- $w_x = w_k(X)$  or  $w_x \xrightarrow{ww} w_k(X)$ . In this case, there is a path from  $w_x$  to  $r_y$  which does not use the edge from  $r_i(X)a$  to  $w_k(X)$ . This path goes from  $w_x$  to  $w_k(X)$  and then from  $w_k(X)$  to  $r_y$ , and (by induction on the previous edges added to  $G$ ) none of these paths violate legality.
- $w_k(X) \xrightarrow{ww} w_x$ . This case implies that there is path from  $w_k(X)$  to  $r_i(X)a$  (this path goes from  $w_k(X)$  to  $w_x$  and then from  $w_x$  to  $r_i(X)a$ ). But this path violates the assumption stating the graph  $G$  built before adding the edge from  $r_i(X)a$  to  $w_k(X)$  is legal. Hence, this case cannot occur.

It follows that the addition of the edge from  $r_i(X)a$  to  $w_k(X)$  does not violate the legality of the updated graph  $G$ .

Second step. The previous step is repeated until, for each object  $X$ , all read operations on  $X$  are ordered with respect to all write operations. When this is done, let us consider any topological sort  $\widehat{S}$  of the resulting acyclic graph. It is easy to see that  $\widehat{S}$  preserves process order and legality (i.e., no process reads an overwritten value). (Reminder: a topological sort of a directed acyclic graph is a sequence of all its vertices, which preserves their partial ordering.)  $\square$

**Theorem 30** *Let  $\widehat{OP}$  be a computation that satisfies the OO constraint. If  $\widehat{OP}$  is legal, it is sequentially consistent.*

The proof of this theorem is similar to the proof of the previous theorem. It is left to the reader.

### 17.1.5 From Theorems to Algorithms

Theorems 29 and 30 provide us with a simple methodology and two approaches to design algorithms implementing sequential consistency.

On the approach side, an algorithm is based on the WW constraint or the OO constraint. On the methodology side, an algorithm has first to ensure the WW constraint or the OO constraint, and then to only guarantee the legality of the read operations. This modularity favors the understanding of what we are doing, and both the design and the proof of algorithms.

The algorithms implementing sequential consistency, which are described in the rest of this chapter, follow this modular approach.

## 17.2 Sequential Consistency from Total Order Broadcast

The common principle the algorithms described in this section rely on, which is due to H. Attiya and J.L. Welch (1994), consists in using the total order broadcast abstraction to implement the WW constraint. Due to Theorem 29, these algorithms have then to ensure only that the computation is legal.

### 17.2.1 A Fast Read Algorithm for Read/Write Objects

**Total Order on Write Operations** This algorithm uses the total order broadcast abstraction to order all write operations. Moreover, as we are about to see, the legality of the read operations is obtained for free with a reading of the local copy of the register  $X$  at  $p_i$ , hence the name *fast read* algorithm.

**Fig. 17.6** Fast read algorithm implementing sequential consistency (code for  $p_i$ )

```

operation  $X.write(v)$  is
(1) to_broadcast SEQ_CONS( $i, X, v$ );
(2)  $received_i \leftarrow false$ ; wait ( $received_i$ );
(3) return().

operation  $X.read()$  is
(4) return( $x_i$ ).

when SEQ_CONS( $j, Y, v$ ) is to-delivered do
(5)  $y_i \leftarrow v$ ;
(6) if ( $j = i$ ) then  $received_i \leftarrow true$  end if.

```

**The Fast Read Algorithm** Each process  $p_i$  maintains a copy  $x_i$  of each read/write register  $X$ . When process  $p_i$  invokes  $X.write(v)$  it uses an underlying to-broadcast algorithm to send the value  $v$  to all the processes (including itself). It then waits until it is to-delivered its own message. When process  $p_i$  invokes  $X.read()$ , it returns the current value of its local copy  $x_i$  of  $X$ . The text of the algorithm is described in Fig. 17.6.

**Theorem 31** *The fast read algorithm implements sequential consistency.*

*Proof* Let us first observe that, thanks to the to-broadcast of all the values which are written, the algorithm satisfies the WW constraint. Hence, due to Theorem 29, it only remains to show that no read operation obtains an overwritten value.

To that end, let us construct a sequence  $\hat{S}$  by enriching the total order on write operations ( $\xrightarrow{ww}$ ) as follows. Let SEQ\_CONS( $j, X, v$ ) and SEQ\_CONS( $k, Y, v'$ ) be the messages associated with any two write operations which are consecutive in  $\xrightarrow{ww}$ . Due to the to-broadcast abstraction, any process to-delivers first SEQ\_CONS( $j, X, v$ ) and then SEQ\_CONS( $k, Y, v'$ ). For any process  $p_i$  let us add (while respecting the process order defined by  $p_i$ ) all the read operations issued by  $p_i$  between the time it has been to-delivered SEQ\_CONS( $j, X, v$ ) and the time it has been to-delivered SEQ\_CONS( $k, Y, v'$ ). It follows from the algorithms that all these read operations obtain the last value written in each register  $X, Y$ , etc., where the meaning of last is with respect to the total order  $\xrightarrow{ww}$ . It follows that, with respect to this total order, no read operation obtains an overwritten value, which concludes the proof of the theorem.  $\square$

**Remark** This implementation of sequential consistency shows an important difference between this consistency condition and atomicity. Both consistency conditions rely on a time notion to order the read and write operations. This time notion has to be in agreement with both process order and object order for atomicity, while it has to be in agreement only with each process order (taken individually) for sequential consistency.

The previous algorithm, which is based on a total order on all the write operations, does more than required. it would be sufficient to respect only process order

and ensure legality. Ordering all write operations allows for a simpler algorithm, which ensures more than sequential consistency but less than atomicity.

### 17.2.2 A Fast Write Algorithm for Read/Write Objects

**Non-blocking Write Operations** It appears that, instead of forcing a write operation issued by a process  $p_i$  to locally terminate only when  $p_i$  has to-delivered the corresponding SEQ\_CONS() message, it is possible to have a fast write implementation in which write operations are never blocked. The synchronization price to obtain sequential consistency has then to be paid by the read operations.

The corresponding fast write algorithm and the previous fast read algorithm are dual algorithms. This duality offers a choice when one has to implement sequentially consistent applications. The fast write algorithm is more appropriate for write-intensive applications, while the fast read algorithm is more appropriate for read-intensive applications.

**The Fast Write Algorithm** As previously, each process  $p_i$  maintains a copy  $x_i$  of each read/write register  $X$ . Moreover, each process  $p_i$  maintains a count of the number of messages SEQ\_CONS() it has to-broadcast and that are not yet to-delivered. This value is kept in the local variable  $nb\_write_i$  (which is initialized to 0). A read invoked by  $p_i$  is allowed to terminate only when  $p_i$  has to-delivered all the messages it has sent. When this occurs, the values written by  $p_i$  are in its past, and consequently (as in the fast read algorithm)  $p_i$  sees all its writes.

**Theorem 32** *The fast write algorithm implements sequential consistency.*

*Proof* As before, due to Theorem 29, we have only to show that no read operation obtains an overwritten value. To that end, let  $r_i(X)a$  be a read operation and  $w_j(X, a)$  be the corresponding write operation. We have to show that

$$\nexists w_k(X, b) \quad \text{such that } [w_j(X, a) \xrightarrow{op} w_k(X, b)] \wedge [w_k(X, b) \xrightarrow{op} r_i(X)a].$$

Let us assume by contradiction that such an operation  $w_k(X, b)$  exists. There are two cases.

- $k = i$ , i.e.,  $w_k(X, b)$  and  $r_i(X)a$  have been issued by the same process  $p_i$ . It follows from the read algorithm that  $nb\_write_i = 0$  when  $r_i(X)a$  is executed. As  $w_k(X, b) \xrightarrow{op} r_i(X)a$ , this means that  $x_i$  has been updated to the value  $b$  and due to total order broadcast, this occurs after  $x_i$  has been assigned the value  $a$ . It follows that  $p_i$  cannot return the value  $a$ , a contradiction.
- $k \neq i$ , i.e.,  $w_k(X, b)$  and  $r_i(X)a$  have been issued by different processes.
 

Since  $w_k(X, b) \xrightarrow{op} r_i(X)a$  and  $a \neq b$ , there is an operation  $oper_i()$  (issued by  $p_i$ ) such that  $w_k(X, b) \xrightarrow{op} oper_i() \xrightarrow{op} r_i(X)a$  (otherwise,  $p_i$  would have read  $b$ ). There are two subcases according to the fact that  $oper_i()$  is a read or a write operation.

**Fig. 17.7** Fast write algorithm implementing sequential consistency (code for  $p_i$ )

```

operation  $X.write(v)$  is
(1)  $nb\_write_i \leftarrow nb\_write_i + 1$ ;
(2)  $to\_broadcast\ SEQ\_CONS(i, X, v)$ ;
(3)  $return()$ .

operation  $X.read()$  is
(4) wait ( $nb\_write_i = 0$ );
(5)  $return(x_i)$ .

when  $SEQ\_CONS(j, Y, v)$  is to-delivered do
(6)  $y_i \leftarrow v$ ;
(7) if ( $j = i$ ) then  $nb\_write_i \leftarrow nb\_write_i - 1$  end if.

```

- $oper_i()$  is a write operation. In this case, due to the variable  $nb\_write_i$ ,  $p_i$  has to wait for this write to locally terminate before executing  $r_i(X)a$ . As  $w_k(X, b) \xrightarrow{op} oper_i$ , it follows from the total order broadcast that  $x_i$  has been assigned the value  $b$  before  $oper_i$  terminates, and due to  $w_j(X, a) \xrightarrow{op} w_k(X, b)$ , this value overwrites  $a$ . Hence,  $r_i(X)$  cannot return  $a$ , which is a contradiction.
- $oper_i()$  is a read operation. As  $w_k(X, b) \xrightarrow{op} oper_i$ , if  $oper_i$  is a read of  $X$ , it cannot return  $a$ , a contradiction. Otherwise  $oper_i$  is a read of another object  $Y$ . But, this implies again that  $x_i$  has been updated to  $b$  before being read by  $p_i$ , a contradiction as  $p_i$  returns  $a$ .  $\square$

### 17.2.3 A Fast Enqueue Algorithm for Queue Objects

An interesting property of the previous total order-based fast read and fast write algorithms lies in the fact that their skeleton (namely, total order broadcast and a fast operation) can be used to design an algorithm implementing a fast enqueue sequentially consistent queue. Such an implementation is presented in Fig. 17.8.

The algorithm implementing the operation  $Q.enqueue(v)$  is similar to that of the fast write algorithm of Fig. 17.7, while the algorithm implementing the operation  $Q.dequeue()$  is similar to that of the corresponding read algorithm. The algorithm assumes that the default value  $\perp$  can neither be enqueued, nor represent the empty stack.

## 17.3 Sequential Consistency from a Single Server

### 17.3.1 The Single Server Is a Process

**Principle** Another way to ensure the WW constraint consists in using a dedicated server process which plays the role of the whole shared memory. Let  $p_{sm}$  be this

**Fig. 17.8**

Fast enqueue algorithm  
implementing  
a sequentially consistent queue  
(code for  $p_i$ )

```

operation  $Q.enqueue(v)$  is
(1)  to_broadcast SEQ_CONS( $i, Q, enq, v$ );
(2)  return().

operation  $Q.dequeue()$  is
(3)   $result_i \leftarrow \perp$ ;
(4)  to_broadcast SEQ_CONS( $i, Q, deq, -$ );
(5)  wait ( $result_i \neq \perp$ );
(6)  return( $result_i$ ).

when SEQ_CONS( $j, Y, op, v$ ) is to-delivered do
(7)  if ( $op = enq$ )
(8)    then enqueue  $v$  at the head of  $q_i$ 
(9)    else  $r \leftarrow$  value dequeued from the tail  $q_i$ 
(10)   if ( $i = j$ ) then  $result_i \leftarrow r$  end if
(11) end if.

```

process. It manages all the objects. Moreover, each process  $p_i$  manages a copy of each object, and its object copies behave as a cache memory. As previously, for any object  $Y$  (capital letter),  $y_i$  (lowercase letter) denotes its local copy at  $p_i$ .

When it invokes  $X.write(a)$ , a process contact  $p_{sm}$ , which thereby defines a total order on all the write operations. It also contacts  $p_{sm}$  when it invokes a read operation  $Y.read()$  and its local copy of  $Y$  is not up to date (i.e.,  $y_i = \perp$ ).

The object manager process  $p_{sm}$  keeps track of which processes have the last value of each object. In that way, when  $p_{sm}$  answers a request from a process  $p_i$ ,  $p_{sm}$  can inform  $p_i$  on which of its local copies are no longer up to date.

**Local Variables Managed by  $p_{sm}$**  In addition of a copy  $x_{sm}$  of each read/write register object  $X$ , the manager  $p_{sm}$  maintains a Boolean array  $hlv_{sm}[1..n, [X, Y, \dots]]$  whose meaning is the following:

$$hlv_{sm}[i, X] \equiv (p_i \text{ has the last value of } X).$$

**The Algorithm** The algorithm is described in Fig. 17.9. When  $p_i$  invokes  $X.write(v)$  it sends a request message to  $p_{sm}$  and waits for an acknowledgment (lines 1–2). The acknowledgment message WRITE\_ACK() informs  $p_i$  on which of its object copies are obsolete. Consequently, when it receives this acknowledgment,  $p_i$  invalidates them (line 3), updates  $x_i$  (line 4), and terminates the write operation.

When  $p_i$  invokes  $X.read()$ , it returns the value of its current copy (line 11) if this copy is up to date, which is captured by the predicate  $x_i \neq \perp$  (line 15). If this copy is not up to date ( $x_i = \perp$ ),  $p_i$  behaves as in the write operation (lines 6–9 are nearly the same as lines 1–4). It contacts the manager  $p_{sm}$  to obtain the last value of  $X$  and, as already indicated,  $p_{sm}$  benefits from the acknowledgment message to inform it on its object copies which do not contain the last written value.

The behavior of  $p_{sm}$  is simple. When it receives a message WRITE\_REQ( $X, v$ ) from a process  $p_i$ , it saves the value  $v$  in  $x_i$  (line 12), and updates accordingly the entries of the array  $hlv_{sm}$  (lines 13–14). It then computes the set ( $inval$ ) of objects for

**Fig. 17.9** Read/write sequentially consistent registers from a central manager

```

===== code of  $p_i$  =====
operation  $X.write(v)$  is
(1) send WRITE_REQ( $X, v$ ) to  $p_{sm}$ ;
(2) wait (WRITE_ACK( $inval$ ) from  $p_{sm}$ );
(3) for each  $Y \in inval$  do  $y_i \leftarrow \perp$  end for;
(4)  $x_i \leftarrow v$ .

operation  $X.read()$  is
(5) if ( $x_i = \perp$ )
(6)   then send READ_REQ( $X$ ) to  $p_{sm}$ ;
(7)     wait (READ_ACK( $inval, v$ ) from  $p_{sm}$ );
(8)     for each  $Y \in inval$  do  $y_i \leftarrow \perp$  end for;
(9)      $x_i \leftarrow v$ 
(10) end if;
(11) return( $x_i$ ).

===== code of  $p_{sm}$  =====
when WRITE_REQ( $X, v$ ) is received from  $p_i$  do
(12)  $x_{sm} \leftarrow v$ ;
(13)  $hlw_{sm}[1..n, X] \leftarrow [false, \dots, false]$ ;
(14)  $hlw_{sm}[i, X] \leftarrow true$ ;
(15)  $inval \leftarrow \{Y \mid \neg hlw_{sm}[i, Y]\}$ ;
(16) send WRITE_ACK( $inval$ ) to  $p_i$ .

when READ_REQ( $X$ ) is received from  $p_i$  do
(17)  $hlw_{sm}[i, X] \leftarrow true$ ;
(18)  $inval \leftarrow \{Y \mid \neg hlw_{sm}[i, Y]\}$ ;
(19) send READ_ACK( $inval, x_{sm}$ ) to  $p_i$ .

```

which  $p_i$  has not the last value, and sends this set to  $p_i$  (lines 15–16). The behavior of  $p_{sm}$  when it receives READ\_REQ( $X$ ) is similar.

**Theorem 33** *The algorithm described in Fig. 17.9 implements a sequentially consistent shared memory made up of read/write registers.*

*Proof* Let us consider a computation of a set of processes cooperating through read/write registers whose accesses are all under the control of the algorithm of Fig. 17.9. We have to show that there is an equivalent sequential execution which is legal.

As the manager process  $p_{sm}$  trivially orders all the write operations, the constraint WW is satisfied. It follows from Theorem 29 that we have only to show that the computation is legal, i.e., no read operation of a process obtains an overwritten value.

To that end, considering a process  $p_i$ , let  $op_1$  be one of its (read or write) operations which entails communication with  $p_{sm}$ , and  $op_2$  its next (read or write) operation entailing again communication with  $p_{sm}$ . It follows from the algorithm that, between  $op_1$  and  $op_2$ ,  $p_i$  has executed only read operations, and these operations were local (they entailed no communication with  $p_{sm}$ ). Moreover, the value of a register  $X$  read by  $p_i$  between  $op_1$  and  $op_2$  is the value of  $x_i$  when  $op_1$  terminates,

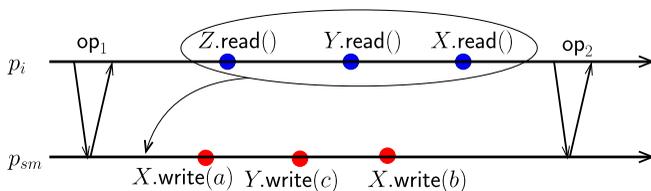


Fig. 17.10 Pattern of read/write accesses used in the proof of Theorem 33

which is the value of  $x_{sm}$  when  $p_{sm}$  sent an acknowledgment to  $p_i$  to terminate  $op_1$ . This is illustrated in Fig. 17.10.

It follows that all the read operations of  $p_i$  that occurred between  $op_1$  and  $op_2$  appear as having occurred after  $op_1$  and before the first operation processed by  $p_{sm}$  after  $op_1$  (these read operations are inside an ellipsis and an arrow shows the point at which—from the “read from” relation point of view—these operations seem to have logically occurred). Hence, no read operation returns an overwritten value, and the computation is legal, which concludes the proof of the theorem.  $\square$

### 17.3.2 The Single Server Is a Navigating Token

**Ensuring the WW Constraint and the Legality of Read Operations** The process  $p_{sm}$  used in the previous algorithm is static and each  $p_i$  sends (receives) messages to (from) it. The key point is that  $p_{sm}$  orders all write operations. Another way to order all write operations consists in using a dynamic approach, namely a navigating token (see Chap. 5). To write a value in a register, a process has first to acquire the token. As there is a single token, this generates a total order on all the write operations.

The legality of read operations can be ensured by requiring the token to carry the same Boolean array  $hlv[1..n, [X, Y, \dots]]$  as the one used in the algorithm of Fig. 17.9. Moreover, as the current owner of the token is the only process that can read and write this array, the current owner can use it to provide the process  $p_j$  to which it sends the token with up to date values.

**The Token-Based Algorithm** The resulting algorithm is described in Fig. 17.11. As the fast read algorithm (Fig. 17.6) and the fast write algorithm (Fig. 17.7) based on the total order broadcast abstraction, it never invalidates object copies at a process. As only the write operations require the token, it is consequently a fast read algorithm.

To execute a write operation on a register, the invoking process  $p_i$  needs the token (line 1). When  $p_i$  has obtained the token,  $op_i$  first updates its object copies which have been modified since the last visit of the token. The corresponding pairs  $(Y, w)$  are recorded in the set  $new\_values$  carried by the token (lines 2–3). Then,  $p_i$

```

operation  $X.write(v)$  is
(1)  acquire_token();
(2)  let  $(hlv, new\_values)$  be the pair carried by the token;
(3)  for each  $(Y, w) \in new\_values$  do  $y_i \leftarrow w; hlv[i, Y] \leftarrow true$  end for;
(4)   $hlw[1..n, X] \leftarrow [false, \dots, false]$ ;
(5)   $x_i \leftarrow v; hlv[i, X] \leftarrow true$ ;
(6)  let  $p_j$  be the process to which the token is sent;
(7)  let  $new\_values = \{(Y, y_i) \mid \neg hlv[j, Y]\}$ ;
(8)  add the pair  $(hlv, new\_values)$  to the token;
(9)  release_token() % for  $p_j$  %.

operation  $X.read()$  is
(10) return( $x_i$ ).

```

**Fig. 17.11** Token-based sequentially consistent shared memory (code for  $p_i$ )

writes the new values of  $X$  into  $x_i$  and updates accordingly  $hlv[i, X]$  and the other entries  $hlv[j, X]$  such that  $j \neq i$  (lines 4–5).

On the token side,  $p_i$  then computes, with the help of the vector  $hlv[i, [X, Y, \dots]]$ , the set of pairs  $(Y, y_i)$  for which the next process  $p_j$  that will have the token does not have the last values (lines 6–7). After it has added this set of pairs to the token (line 8),  $p_i$  releases the token, which is sent to  $p_j$  (line 9). Finally the read operation is purely local (line 10).

Let us notice that, when it has the token, a process can issue several write operations on the same or several registers. The only requirement is that a process that wants to write eventually must acquire the token.

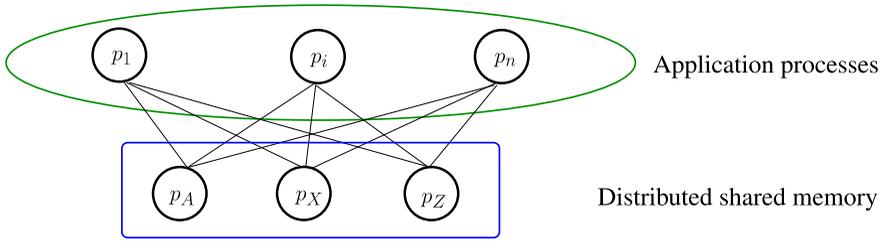
**On the Moves of the Token** The algorithm assumes that when a process releases the token, it knows the next user of the token. This can be easily implemented by having the token move on a ring. In this case, the token acts as an object used to update the local memories of the processes with the last values written. Between two consecutive visits of the token, a process reads its local copies. The consistency argument is the same as the one depicted in Fig. 17.10.

## 17.4 Sequential Consistency with a Server per Object

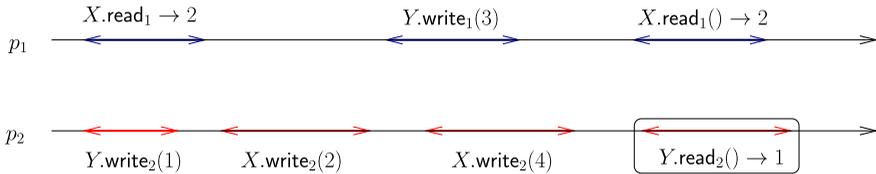
While the previous section presented algorithms based on the WW constraint, this section presents an algorithm based on the OO constraint.

### 17.4.1 Structural View

The structural view is described in Fig. 17.12. There is a manager process  $p_X$  per object  $X$ , and the set of managers  $\{p_A, \dots, p_Z\}$  implements, in a distributed way, the whole shared memory.



**Fig. 17.12** Architectural view associated with the OO constraint



**Fig. 17.13** Why the object managers must cooperate

There is a bidirectional channel for every pair made up of an application process and a manager process. This is because we are interested in a genuine OO-based implementation of sequential consistency: at the application level, there is no notion of a channel and the application processes are assumed to communicate only by reading and writing shared registers.

Moreover, there is a channel connecting any pair of manager processes. As we will see, this is required because, differently from atomicity, sequential consistency is not a local property. The object managers have to cooperate to ensure that there is a legal sequential sequence of operations that is equivalent to the real computation. Such a cooperation exists but does not appear explicitly in the algorithm based on a single process  $p_{sm}$  managing all the objects (such as the one presented in Sect. 17.3.1).

### 17.4.2 The Object Managers Must Cooperate

Let us consider the computation described in Fig. 17.13, where  $X$  and  $Y$  are initialized to 0, in which the last read by  $p_2$  is missing. As it is equivalent to the following legal sequence  $\widehat{S}$ ,

$$Y.write_2(1), X.write_2(2), X.read_1() \rightarrow 2, Y.write_1(3), X.read_1() \rightarrow 2, X.write_2(4),$$

this computation is sequentially consistent.

**Fig. 17.14**

Sequential consistency  
with a manager per object:  
process side

```

operation  $X.write(v)$  is
(1)  $valid \leftarrow \{Y \mid y_i \neq \perp\}$ ;
(2) send WRITE_REQ( $X, v, valid$ ) to  $p_X$ ;
(3) wait (WRITE_ACK( $inval$ ) from  $p_X$ );
(4) for each  $Y \in inval$  do  $y_i \leftarrow \perp$  end for;
(5)  $x_i \leftarrow v$ .

operation  $X.read()$  is
(6) if ( $x_i = \perp$ )
(7)   then  $valid \leftarrow \{Y \mid y_i \neq \perp\}$ ;
(8)     send READ_REQ( $X, valid$ ) to  $p_X$ ;
(9)     wait (READ_ACK( $inval, v$ ) from  $p_X$ );
(10)    for each  $Y \in inval$  do  $y_i \leftarrow \perp$  end for;
(11)     $x_i \leftarrow v$ 
(12) end if;
(13) return( $x_i$ ).

```

The full computation (including  $Y.read_2() \rightarrow 1$ ), is not sequentially consistent: There is no way to order all the operations while respecting both the process order relation and the sequential specification of both  $X$  and  $Y$ . This is due to the fact that, as the second read of  $p_1$  returns 2, the operation  $X.write_2(4)$  has to appear after it in a sequence (otherwise  $p_1$  could not have obtained the value 2). But, then this write has to appear after the operation  $X.write_2(4)$  issued by  $p_1$ , and consequently the read by  $p_2$  should return 3 for the computation to be sequentially consistent.

This means that when  $p_2$  executes  $X.write_2(4)$ , its local copy of  $Y$ , namely  $y_2 = 0$ , no longer ensures the legality of its next read of this object  $Y$ . The manager of  $X$  has to cooperate with the manager of  $X$  to discover it, so that  $p_4$  invalidate its local copy of  $Y$ .

### 17.4.3 An Algorithm Based on the OO Constraint

An algorithm based on the OO constraint is described in Fig. 17.14 (application process part) and Fig. 17.15 (object manager part).

**On an Application Process Side** The algorithms implementing the operations  $X.write(v)$  and  $X.read()$  are nearly the same as those of Fig. 17.9 which are for a single object (or a system where all the objects are managed by the same process  $p_{sm}$ ).

The only difference lies in the fact that a process  $p_i$  adds to its request message (WRITE\_REQ()) sent at line 2, and READ\_REQ() sent at line 8), the set of objects for which its local copies are valid (i.e., reading them before the current read or write operation would not violate sequential consistency). This information will help the manager of  $X$  (the object currently read or written by  $p_i$ ) to help it invalidate its local copies whose future read would not be legal with their previous values.

```

when WRITE_REQ( $X, v, valid$ ) is received from  $p_i$  do
(14)  $x_X \leftarrow v$ ;
(15)  $hlw_X[1..n] \leftarrow [false, \dots, false]$ ;
(16)  $hlw_X[i] \leftarrow true$ ;
(17)  $inval \leftarrow check\_legality(X, i, valid)$ ;
(18) send WRITE_ACK( $inval$ ) to  $p_i$ .

when READ_REQ( $X, valid$ ) is received from  $p_i$  do
(19)  $hlw_X[i] \leftarrow true$ ;
(20)  $inval \leftarrow check\_legality(X, i, valid)$ ;
(21) send READ_ACK( $inval, x_X$ ) to  $p_i$ .

procedure check_legality( $X, i, valid$ ) is
(22)  $inval\_set \leftarrow \emptyset$ ;
(23) for each  $Y \in valid \setminus \{X\}$  do send VALID_REQ( $i$ ) to  $p_Y$  end for;
(24) for each  $Y \in valid \setminus \{X\}$  do
(25)   receive VALID_ACK( $rep$ ) from  $p_Y$ ;
(26)   if  $rep = no$  then  $inval\_set \leftarrow inval\_set \cup \{Y\}$  end if
(27) end for;
(28) return( $inval\_set$ ).

when VALID_REQ( $j$ ) is received from  $p_Y$  do
(29) if ( $(p_X$  is processing a write request)  $\vee \neg hlw_X[j]$ )
(30)   then  $r \leftarrow no$  else  $r \leftarrow yes$ 
(31) end if;
(32) send VALID_ACK( $r$ ) to  $p_Y$ .

```

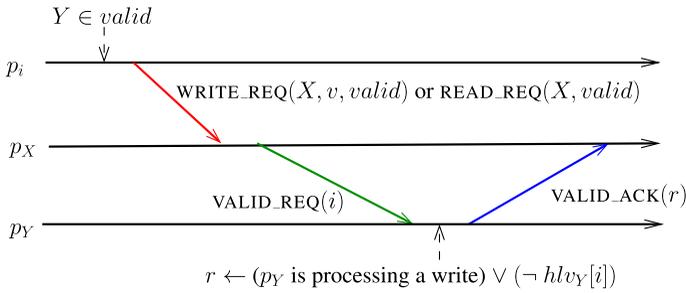
**Fig. 17.15** Sequential consistency with a manager per object: manager side

**On an Object Manager Side** The behavior of the process  $p_X$  managing the object  $X$  is described in Fig. 17.15. This manager maintains a Boolean vector  $hlw_X[1..n]$  such that  $hlw_X[i]$  is true if and only if  $p_i$  has the last value of  $X$ .

The code of the algorithms associated with the reception of a message WRITE\_REQ( $X, v, valid$ ) or the reception of a message READ\_REQ( $X, valid$ ) is similar to its counterpart of Fig. 17.9 (where all objects are managed by the same process  $p_{sm}$ ). The difference lies in the way the legality of the objects is ensured. The matrix  $hlv_{sm}[1..n, [X, Y, \dots]]$  used by the centralized manager  $p_{sm}$  is now distributed on all the object managers as follows: The vector  $hlv_{sm}[1..n, X]$  is now managed only by  $p_X$ , which records it in its local vector  $hlw_X[1..n]$ .

The read or write operation currently invoked by  $p_i$  may entail the invalidation of its local copy of some objects, e.g., the object  $Y$ . This means that the computation, by  $p_X$ , of the set  $inval$  associated with this read or write operation requires that  $p_X$  cooperates with other object managers. This cooperation is realized by the local procedure denoted  $check\_legality(X, i, valid)$ .

As only the objects belonging to the current set  $valid$  transmitted by  $p_i$  can be invalidated,  $p_X$  sends a message VALID\_REQ( $i$ ) to each manager  $p_Y$  such that  $Y \in valid$  (line 23). It then waits for an answer for each of these messages (line 24), each carrying the answer *no* or *yes*. The answer *yes* means that  $p_Y$  claims that the current



**Fig. 17.16** Cooperation between managers is required by the OO constraint

copy of  $Y$  at  $p_i$  is still valid (its future reads by  $p_i$  will still be legal). The answer *no* means that the copy of  $Y$  at  $p_i$  must be invalidated.

The behavior of a manager  $p_Y$ , when it receives a message  $\text{VALID\_REQ}(i)$  from a manager  $p_X$ , is described in Fig. 17.16. When this occurs,  $p_Y$  (conservatively) asks  $p_X$  to invalidate the copy of  $Y$  at  $p_i$  if it knows that this copy is no longer up to date. This is the case if  $\text{hlw}_Y[i]$ , or if  $p_Y$  is currently processing a write (which, by construction is a write of  $Y$ ). When this occurs,  $p_Y$  answers *no*. Otherwise, the copy of  $Y$  at  $p_i$  is still valid, and  $p_Y$  answers *yes* to  $p_X$ .

**Sequential Consistency vs. Atomicity/Linearizability** The previous OO-based implementation shows clearly the price that has to be paid to implement sequential consistency, namely, when each object is managed by a single process, the object managers must cooperate to guarantee a correct implementation.

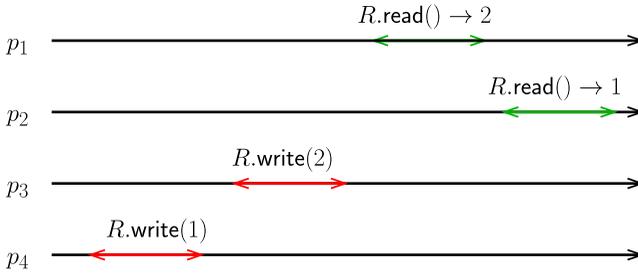
As seen in Sect. 16.4.3, this was not the case in the algorithm implementing atomicity (see the algorithm in Fig. 16.10). This has a very simple explanation: atomicity (linearizability) is a local property (Theorem 27), while sequential consistency is not (see Sect. 17.1.2).

## 17.5 A Weaker Consistency Condition: Causal Consistency

### 17.5.1 Definition

**Underlying Intuition** Causal consistency is a consistency condition for read/write objects which is strictly weaker than sequential consistency. Its essence is to capture only the causality relation defined by the read-from relation. To that end, it does not require that all the processes agree on the very same legal sequential history  $\widehat{S}$ .

Two processes are not required to agree on the write operations which are concurrent; they can see them in different orders. Read operations are, however, required to be legal.



**Fig. 17.17** An example of a causally consistent computation

**Remark** The notion of a *sequential observation* of a distributed message-passing computation was introduced in Sect. 6.3.3. A sequential observation of a distributed execution is a sequence including all its events, respecting their partial ordering. Hence, distinct observations differ in the way they order concurrent events.

The notion of causal consistency is a similar notion applied to read/write objects.

**Definition** The set of operations that may affect a process  $p_i$  are its read and write operations, plus the write operations issued by the other processes. Given a computation  $\widehat{OP}$ , let  $\widehat{OP}_i$  be the computation from which all the read operations not issued by  $p_i$  have been removed.

A computation  $\widehat{OP}$  is *causally consistent* if, for any process  $p_i$ , there is a legal sequential computation  $\widehat{H}_i$  that is equivalent to  $\widehat{OP}_i$ .

While this means that, for each process  $p_i$ ,  $\widehat{OP}_i$  is sequentially consistent, it does not mean that  $\widehat{OP}$  is sequentially consistent. (This type of consistency condition is oriented toward cooperative work.)

**Example 1** Let us first consider the computation described in Fig. 17.2. The following sequential computation  $\widehat{H}_i$  and  $\widehat{H}_j$  can be associated with  $p_i$  and  $p_j$ , respectively:

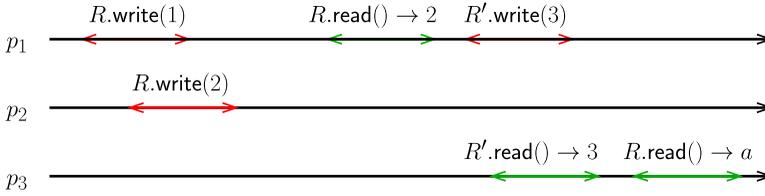
$$\begin{aligned} \widehat{H}_i &\equiv R'.write_j(0), \quad R'.read_i(0), \quad R'.write_j(2), \quad R'.read_i(2), \\ \widehat{H}_j &\equiv R.write_i(0), \quad R.read_i(0), \quad R.write_j(1), \quad R.read_i(1). \end{aligned}$$

As both  $\widehat{H}_i$  and  $\widehat{H}_j$  are legal, the computation is causally consistent.

**Example 2** A second example is described in Fig. 17.17, where it is assumed that the write operations are independent. It follows that  $p_1$  and  $p_2$  can order these write operations in any order, and consequently as the sequences

$$\begin{aligned} \widehat{H}_1 &\equiv R.write_4(1), \quad R.write_3(2), \quad R.read_1() \rightarrow 2, \quad \text{and} \\ \widehat{H}_2 &\equiv R.write_4(2), \quad R.write_3(1), \quad R.read_2() \rightarrow 1 \end{aligned}$$

are legal, the computation is causally consistent.



**Fig. 17.18** Another example of a causally consistent computation

Let us now assume that  $R.write_4(1) \xrightarrow{op} R.write_3(2)$ . In this case, both  $\widehat{H}_1$  and  $\widehat{H}_2$  must order  $R.write_4(1)$  before  $R.write_3(2)$ , and for the computation to be causally consistent, both read operations have to return the value 2. Let us observe that, in this case, this computation is also atomic.

**Example 3** A second example is described in Fig. 17.18, where the computation is made up of three processes accessing the registers  $R$  and  $R'$ . Moreover, as before, the operations  $R.write_1(1)$  and  $R.write_2(2)$  are concurrent.

If  $a = 2$  (value returned from the read of  $R$  by  $p_3$ ), the computation is causally consistent. This is due to the two following sequences which (a) respect the relation  $\xrightarrow{op}$  and (b) are legal:

$$\begin{aligned} \widehat{H}_1 &\equiv R.write_1(1), R.write_2(2), R.read_1() \rightarrow 2, R'.write_1(3), \\ \widehat{H}_2 &\equiv R.write_2(2), \quad \text{and} \\ \widehat{H}_3 &\equiv R.write_1(1), R.write_2(2), R'.write_1(3), R'.read_3() \rightarrow 3, R.read_3() \rightarrow 2. \end{aligned}$$

Let us notice that, in this case, the computation is also sequentially consistent.

If  $a = 1$  the computation is still causally consistent, but is no longer sequentially consistent. In this case  $p_3$  orders the concurrent write operations of  $R$  in the reverse order, i.e., we have

$$\widehat{H}_3 \equiv R.write_2(2), R.write_1(1), R'.write_1(3), R'.read_3() \rightarrow 3, R.read_3() \rightarrow 2.$$

### 17.5.2 A Simple Algorithm

**Causal Consistency and Causal Message Delivery** It is easy to see that, for read/write registers, causal consistency is analogous to causal broadcast on message deliveries. The notion of causal message delivery was introduced in Sect. 12.1, and its broadcast instance was developed in Sect. 12.3. It states that no message  $m$  can be delivered to a process before the messages broadcast in the causal past of  $m$ .

**Fig. 17.19**

A simple algorithm  
implementing causal consistency

```

operation  $X.write(v)$  is
(1) co_broadcast CAUSAL_CONS( $X, x_i$ );
(2)  $x_i \leftarrow v$ ;

operation  $X.read()$  is
(3) return( $x_i$ );

when CAUSAL_CONS( $Y, v$ ) is co-delivered do
(4)  $y_i \leftarrow v$ ;

```

For causal consistency, the causal past is defined with respect to the relation  $\xrightarrow{op}$ , a write operation corresponds to the broadcast of a message, while a read operation corresponds to a message reception (with the difference that a value written can never be read, and the same value can be read several times).

**A Simple Algorithm** It follows from the previous discussion that a simple way to implement causal consistency lies in using an underlying causal broadcast algorithm. Several algorithms were presented in Sect. 12.3; they provide the processes with the operations `co_broadcast()` and `co_deliver()`. We consider here the causal broadcast algorithm presented in Fig. 12.10.

The algorithm is described in Fig. 17.19. Each process manages a copy  $x_i$  of every register object  $X$ . It is easy to see that both the read operation and the write operation are fast. This is due to the fact that, as causality involves only the causal past, no process coordination is required.

### 17.5.3 The Case of a Single Object

**A Simple Algorithm** This section presents a very simple algorithm that implements causal consistency when there is a single object  $X$ . This algorithm is based on scalar clocks (these logical clocks were introduced in Sect. 7.1.1).

The algorithm is described in Fig. 17.20. The scalar clock of  $p_i$  is denoted  $clock_i$  and is initialized to 0.

```

operation  $X.write(v)$  is
(1)  $clock_i \leftarrow clock_i + 1$ ;
(2)  $x_i \leftarrow v$ ;
(3) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do send CAUSAL_CONS( $v, clock_i$ ) to  $p_j$  end for.

operation  $X.read()$  is
(4) return( $x_i$ );

when CAUSAL_CONS( $v, lw\_date$ ) is delivered do
(5) if ( $lw\_date > clock_i$ ) then  $clock_i \leftarrow lw\_date$ ;  $x_i \leftarrow v$  end if.

```

**Fig. 17.20** Causal consistency for a single object

As before, the read and write operations are fast. When a process invokes a write operation it increases its local clock, associates the corresponding date with its write, and sends the message  $\text{CAUSAL\_CONS}(v, \text{clock}_i)$  to all the other processes. The scalar clocks establish a total order on the write operations which are causally dependent.

Write operations with the same date  $w\_date$  are concurrent. Only the first of them that is received by a process  $p_i$  is taken into account by  $p_i$ . The algorithm considers that, from the receiver  $p_i$ 's point of view, the other ones are overwritten by the first one. Hence, two processes  $p_i$  and  $p_j$  can order differently concurrent write operations.

**From Causal Consistency to Sequential Consistency** This algorithm can be easily enriched if one wants to obtain sequential consistency instead of causal consistency. To that end, logical clocks have to be replaced by timestamps (see Sect. 7.1.2). Moreover, each process  $p_i$  manages two additional local variables, denoted  $\text{last\_writer}_i$  and  $\text{lw\_date}_i$ . To  $p_i$ 's knowledge,  $\text{last\_writer}_i$  contains the identity of the last process that has written into  $X$ , and  $\text{lw\_date}_i$  contains the date associated with this write. The pair  $\langle \text{lw\_date}_i, \text{last\_writer}_i \rangle$  constitutes the timestamp of the last write of  $X$  known by  $p_i$ . The modifications of the algorithm are as follows:

- Instead of sending the message  $\text{CAUSAL\_CONS}(v, \text{clock}_i)$  (line 3), a process  $p_i$  sends now the message  $\text{CAUSAL\_CONS}(v, \langle \text{clock}_i, i \rangle)$ .
- When it receives a message  $\text{CAUSAL\_CONS}(v, \langle \text{lw\_date}, j \rangle)$ , a process  $p_i$  takes it into account only if  $\langle \text{lw\_date}, j \rangle > \langle \text{lw\_date}_i, \text{last\_writer}_i \rangle$ . It then updates  $x_i$  to  $v$  and  $\langle \text{lw\_date}_i, \text{last\_writer}_i \rangle$  to  $\langle \text{lw\_date}, j \rangle$ .

In that way, their timestamps define a total order on all the write operations, and no two processes order two write operations differently. The writes discarded by a process correspond to values that it sees as overwritten values.

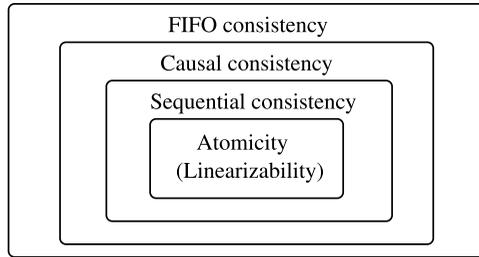
## 17.6 A Hierarchy of Consistency Conditions

Atomicity (linearizability), sequential consistency, causal consistency, and FIFO consistency (presented in Problem 5), define a strict hierarchy of consistency conditions. This hierarchy is described in Fig. 17.21.

## 17.7 Summary

This chapter introduced sequential consistency and causal consistency, and presented several algorithms which implement them. As far as sequential consistency is concerned, it presented two properties (denoted WW and OO) which simplify the design of algorithms implementing this consistency condition.

**Fig. 17.21** Hierarchy of consistency conditions



## 17.8 Bibliographic Notes

- Sequential consistency was introduced by L. Lamport [227].
- Analysis of sequential consistency and its relation with atomicity can be found in [23, 313]. It is shown in [364] that checking if a computation is sequentially consistent is an NP-complete problem.
- Detection of violation of sequential consistency is addressed in [156].
- A lower bound on the time cost to ensure sequential consistency of read/write registers is proved in [23]. Considering that the transit time of each message is upper bounded by  $\delta$ , it is shown that, whatever the algorithm, the sum of the delays for a read operation and a write operation is at most  $\delta$ . The fast read (resp., write) algorithm corresponds to the case where the read (resp., write) operation cost 0 time units, while the write (resp., read) operation can cost up to  $\delta$  time units.
- Both the WW constraint and the OO constraint, Theorems 29 and 30, are due to M. Mizuno, M. Raynal, and J.Z. Zhou [269]. The constraint-based approach is due to T. Ibaraki, T. Kameda, and T. Minoura [196].
- The three algorithms based on the total order broadcast abstraction (fast read algorithm, Sect. 17.2.1, fast write algorithm, Sect. 17.2.2, and fast enqueue algorithm, Sect. 17.2.3) are due to H. Attiya and J.L. Welch [23]. Formal proofs of these algorithms can be found in that paper.
- The algorithm described in Sect. 17.3.1 (sequential consistency with a single server for all the objects) and the algorithm described in Sect. 17.4 (sequential consistency with a server per object) are due to M. Mizuno, M. Raynal, and J.Z. Zhou [269].
- The token-based algorithm implementing sequential consistency is due to M. Raynal [314].
- An optimistic algorithm for sequential consistency is described in [268]. Other algorithms implementing sequential consistency can be found in [320, 322].
- The implementation of sequential consistency when objects are accessed by multiobject operations is addressed in [326].
- Causal consistency was introduced by M. Ahamad, G. Neiger, J.E. Burns, P.W. Hutto, and P. Kohli [10].
- Algorithms implementing causal consistency can be found in [10, 11, 92, 318]. An algorithm which allows processes to dynamically switch from sequential consistency to causal consistency, and vice-versa, is described in [369].

- Other consistency conditions weaker than sequential consistency have been proposed, e.g., slow memory [194], lazy release consistency [204], and PRAM consistency [237] (to cite a few). See [323] for a short introductory survey.
- Normality is a consistency condition which considers process order and object order [151]. When operations are on a single object, it is equivalent to sequential consistency.
- Very general algorithms, which can be instantiated with one of many consistency conditions, are described in [200, 212].

## 17.9 Exercises and Problems

1. Considering a concurrent stack defined by the classical operations `push()` and `pop()`, describe an execution where the stack behaves atomically. Design then an execution where the stack is sequentially consistent but not atomic.
2. Give a proof of Theorem 30.
3. Prove that the fast enqueue algorithm described in Fig. 17.8 implements a sequentially consistent queue.
4. Considering the fast write algorithm described in Sect. 17.2.2, let us replace the variable  $nb\_write_i$  by an array  $nb\_write_i[X, Y, Z, \dots]$ , where  $nb\_write_i[X]$  counts the number write into  $X$  issued by  $p_i$  and not yet to-delivered to  $p_i$ .

When  $p_i$  invokes  $X.read()$ , is it possible to replace the predicate ( $nb\_write_i = 0$ ) used at line 4 by the predicate  $nb\_write_i[X] = 0$ ? If the answer is “yes”, prove the corresponding algorithm. If the answer is “no”, give a counterexample.

5. FIFO consistency requires that the write operations issued by each process  $p_i$  are seen by each process in the order in which they have been issued by  $p_i$ . There is no constraint on the write operations issued by different processes.

Describe an algorithm implementing FIFO consistency. Show that this consistency condition is weaker than causal consistency.

6. Show that causal consistency plus the WW constraint provide sequential consistency.

Solution in [322].

7. Extend the algorithm described in Sect. 17.3.1 so that it benefits from the notion of an owner process introduced in Sect. 16.4.4. (A process owns an object  $X$  from the time it writes it until the next read or write operation that will be applied to  $X$  by another process. As during this period no other process accesses  $X$ ,  $p_i$  can write it several times without having to inform the manager process.)
8. Extend the algorithms described Fig. 17.14 (application process algorithm) and Fig. 17.15 (manager algorithm) so that they benefit from the notion of an owner process.

Solution in [313].

9. Design an algorithm that implements causal consistency, where each process manages copies of only a subset of the objects.

Solution in [318].

# Afterword

## The Aim of This Book

The practice of sequential computing has greatly benefited from the results of the theory of sequential computing that were captured in the study of formal languages and automata theory. Everyone knows what can be computed (computability) and what can be computed efficiently (complexity). All these results constitute the foundations of sequential computing, which, thanks to them, has become a *science*. These theoretical results and algorithmic principles have been described in many books from which students can learn basic results, algorithms, and principles of sequential computing (e.g., [99, 107, 148, 189, 205, 219, 258, 270, 351] to cite a few).

Since Lamport’s seminal paper “*Time, clocks, and the ordering of events in a distributed system*”, which appeared in 1978 [226], distributed computing is no longer a set of tricks or recipes, but a domain of computing science with its own concepts, methods, and applications. The world is distributed, and today the major part of applications are distributed. This means that message-passing algorithms are now an important part of any computing science or computing engineering curriculum.

Thanks to appropriate curricula—and good associated books—students have a good background in the theory and practice of sequential computing. In the same spirit, an aim of this book is to try to provide them with an appropriate background when they have to solve distributed computing problems.

Technology is what makes everyday life easier. Science is what allows us to transcend it, and capture the deep nature of the objects we are manipulating. To that end, it provides us with the right concepts to master and understand what we are doing. Considering failure-free asynchronous distributed computing, an ambition of this book is to be a step in this direction.

## **Most Important Concepts, Notions, and Mechanisms Presented in This Book**

Chapter 1: Asynchronous/synchronous system, breadth-first traversal, broadcast, convergecast, depth-first traversal, distributed algorithm, forward/discard principle, initial knowledge, local algorithm, parallel traversal, spanning tree, unidirectional logical ring.

Chapter 2: Distributed graph algorithm, cycle detection, graph coloring, knot detection, maximal independent set, problem reduction, shortest path computation.

Chapter 3: Cut vertex, de Bruijn's graph, determination of cut vertices, global function, message filtering, regular communication graph, round-based framework.

Chapter 4: Anonymous network, election, message complexity, process identity, ring network, time complexity, unidirectional versus bidirectional ring.

Chapter 5: Adaptive algorithm, distributed queuing, edge/link reversal, mobile object, mutual exclusion, network navigation, object consistency, routing, scalability, spanning tree, starvation-freedom, token.

Chapter 6: Event, causal dependence relation, causal future, causal path, causal past, concurrent (independent) events, causal precedence relation, consistent global state, cut, global state, happened before relation, lattice of global states, observation, marker message, nondeterminism, partial order on events, partial order on local states, process history, process local state, sequential observation.

Chapter 7: Adaptive communication layer, approximate causality relation, causal precedence, causality tracking, conjunction of stable local predicates, detection of a global state property, discarding old data, Hasse diagram, immediate predecessor, linear (scalar) time (clock), logical time, matrix time (clock), message stability, partial (total) order, relevant event,  $k$ -restricted vector clock, sequential observation, size of a vector clock, timestamp, time propagation, total order broadcast, vector time (clock).

Chapter 8: Causal path, causal precedence, communication-induced checkpointing, interval (of events), local checkpoint, forced local checkpoint, global checkpoint, hidden dependency, recovery, rollback-dependency trackability, scalar clock, spontaneous local checkpoint, uncoordinated checkpoint, useless checkpoint, vector clock,  $Z$ -dependence, zigzag cycle, zigzag pattern, zigzag path, zigzag prevention.

Chapter 9: Asynchronous system, bounded delay network, complexity, graph covering structure, physical clock drift, pulse-based programming, synchronizer, synchronous algorithm.

Chapter 10: Adaptive algorithm, arbiter permission, bounded algorithm, deadlock-freedom, directed acyclic graph, extended mutex, adaptive algorithm, grid quorum, individual permission, liveness property, mutual exclusion (mutex), preemption, quorum, readers/writers problem, safety property, starvation-freedom, timestamp, vote.

Chapter 11: Conflict graph, deadlock prevention, graph coloring, incremental requests,  $k$ -out-of- $M$  problem, permission, resource allocation, resource graph, resource type, resource instance, simultaneous requests, static/dynamic (resource) session, timestamp, total order, waiting chain, wait-for graph.

Chapter 12: Asynchronous system, bounded lifetime message, causal barrier, causal broadcast, causal message delivery order, circulating token, client/server broadcast, coordinator process, delivery condition, first in first out (FIFO) channel, order properties on a channel, size of control information, synchronous system.

Chapter 13: Asynchronous system, client-server hierarchy, communication initiative, communicating sequential processes, crown, deadline-constrained interaction, deterministic vs. nondeterministic context, logically instantaneous communication, planned vs. forced interaction, rendezvous, multiparty interaction, synchronous communication, synchronous system, token.

Chapter 14: AND receive, asynchronous system, atomic model, counting, diffusing computation, distributed iteration, global state,  $k$ -out-of- $n$  receive statement, loop invariant, message arrival vs. message reception, network traversal, nondeterministic statement, OR receive statement, reasoned construction, receive statement, ring, spanning tree, stable property, termination detection, wave.

Chapter 15: AND communication model, cycle, deadlock, deadlock detection, knot, one-at-a-time model, OR communication model, probe-based algorithm, resource vs. message, stable property, wait-for graph.

Chapter 16: Atomicity, composability, concurrent object, consistency condition, distributed shared memory, invalidation vs. update, linearizability, linearization point, local property, manager process, object operation, partial order on operations, read/write register, real time, sequential specification, server process, shared memory abstraction, total order broadcast abstraction.

Chapter 17: Causal consistency, concurrent object, consistency condition, distributed shared memory, invalidation, logical time, manager process, OO constraint, partial order on operations, read/write register, sequential consistency, server processes, shared memory abstraction, total order broadcast abstraction, WW constraint.

## How to Use This Book

This section presents two courses on distributed computing which can benefit from the concepts, algorithms and principles presented in this book. Each course is a one-semester course, and they are designed to be sequential (a full year at the undergraduate level, or split, with the first course at the undergraduate level and the second at the beginning of the graduate level).

- A first one-semester course on distributed computing could first focus on Part I, which is devoted to graph algorithms. Then, the course could address (a) distributed mutual exclusion (Chap. 10), (b) causal message delivery and total order

broadcast (Chap. 12), and (c) distributed termination detection (Chap. 14), if time permits.

The spirit of this course is to be an introductory course, giving students a correct intuition of what distributed algorithms are (they are not simple “extensions” of sequential algorithms), and show them that there are problems which are specific to distributed computing.

- A second one-semester course on distributed computing could first address the concept of a global state (Chap. 6). The aim is here to give the student a precise view of what a distributed execution is and introduce the notion of a global state. Then, the course could develop and illustrate the different notions of logical times (Chap. 7).

Distributed checkpointing (Chap. 8), synchronizers (Chap. 9), resource allocation (Chap. 11), rendezvous communication (Chap. 13), and deadlock detection (Chap. 15), can be used to illustrate the previous notions.

Finally, the meaning and the implementation of a distributed shared memory (Part VI) could be presented to introduce the notion of a consistency condition, which is a fundamental notion of distributed computing.

Of course, this book can also be used by engineers and researchers who work on distributed applications to better understand the concepts and mechanisms that underlie their work.

## From Failure-Free Systems to Failure-Prone Systems

This book was devoted to algorithms for failure-free asynchronous distributed applications and systems. Once the fundamental notions, concepts, and algorithms of failure-free distributed computing are mastered, one can focus on more specific topics of failure-prone distributed systems. In such a context, the combined effect of asynchrony and failures create *uncertainty* that algorithms have to cope with. The reader interested in the net effect of asynchrony and failure on the design of distributed algorithms is invited to consult the following books: [24, 67, 150, 155, 219, 242, 315, 316] (to cite a few).

## A Series of Books

This book completes a series of four books, written by the author, devoted to concurrent and distributed computing [315–317]. More precisely, we have the following.

- As has been seen, this book is on elementary distributed computing for *failure-free asynchronous* systems.
- The book [317] is on algorithms in *asynchronous* shared memory systems where processes can commit *crash failures*. It focuses on the construction of reliable concurrent objects in the presence of process crashes.

- The book [316] is on *asynchronous message-passing systems* where processes are prone to *crash failures*. It presents communication and agreement abstractions for fault-tolerant asynchronous distributed systems. Failure detectors are used to circumvent impossibility results encountered in pure asynchronous systems.
- The book [315] is on *synchronous* message-passing systems, where the processes are prone to *crash failures, omission failures, or Byzantine failures*. It focuses on the following distributed agreement problems: consensus, interactive consistency, and non-blocking atomic commit.

*Enseigner, c'est réfléchir à voix haute devant les étudiants.*  
Henri-Léon Lebesgue (1875–1941)

*Make everything as simple as possible, but not simpler.*  
Albert Einstein (1879–1955)

# References

1. A. Acharya, B.R. Badrinath, Recording distributed snapshot based on causal order of message delivery. *Inf. Process. Lett.* **44**, 317–321 (1992)
2. A. Acharya, B.R. Badrinath, Checkpointing distributed application on mobile computers, in *3rd Int'l Conference on Parallel and Distributed Information Systems* (IEEE Press, New York, 1994), pp. 73–80
3. S. Adve, K. Gharachorloo, Shared memory consistency models. *IEEE Comput.* **29**(12), 66–76 (1996)
4. S. Adve, M. Mill, A unified formalization of four shared memory models. *IEEE Trans. Parallel Distrib. Syst.* **4**(6), 613–624 (1993)
5. Y. Afek, G.M. Brown, M. Merritt, Lazy caching. *ACM Trans. Program. Lang. Syst.* **15**(1), 182–205 (1993)
6. A. Agarwal, V.K. Garg, Efficient dependency tracking for relevant events in concurrent systems. *Distrib. Comput.* **19**(3), 163–183 (2007)
7. D. Agrawal, A. El Abbadi, An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.* **9**(1), 1–20 (1991)
8. D. Agrawal, A. Malpini, Efficient dissemination of information in computer networks. *Comput. J.* **34**(6), 534–541 (1991)
9. M. Ahamad, M.H. Ammar, S.Y. Cheung, Multidimensional voting. *ACM Trans. Comput. Syst.* **9**(4), 399–431 (1991)
10. M. Ahamad, G. Neiger, J.E. Burns, P.W. Hutto, P. Kohli, Causal memory: definitions, implementation and programming. *Distrib. Comput.* **9**, 37–49 (1995)
11. M. Ahamad, M. Raynal, G. Thia-Kime, An adaptive protocol for implementing causally consistent distributed services, in *Proc. 18th Int'l Conference on Distributed Computing Systems (ICDCS'98)* (IEEE Press, New York, 1998), pp. 86–93
12. M. Ahuja, Flush primitives for asynchronous distributed systems. *Inf. Process. Lett.* **34**, 5–12 (1990)
13. M. Ahuja, M. Raynal, An implementation of global flush primitives using counters. *Parallel Process. Lett.* **5**(2), 171–178 (1995)
14. S. Alagar, S. Venkatesan, An optimal algorithm for distributed snapshots with message causal ordering. *Inf. Process. Lett.* **50**, 310–316 (1994)
15. A. Alvarez, S. Arévalo, V. Cholvi, A. Fernández, E. Jiménez, On the interconnection of message passing systems. *Inf. Process. Lett.* **105**(6), 249–254 (2008)
16. L. Alvisi, K. Marzullo, Message logging: pessimistic, optimistic, and causal. *IEEE Trans. Softw. Eng.* **24**(2), 149–159 (1998)
17. E. Anceaume, J.-M. Hélary, M. Raynal, Tracking immediate predecessors in distributed computations, in *Proc. 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'002)* (ACM Press, New York, 2002), pp. 210–219

18. E. Anceaume, J.-M. Hélary, M. Raynal, A note on the determination of the immediate predecessors in a distributed computation. *Int. J. Found. Comput. Sci.* **13**(6), 865–872 (2002)
19. D. Angluin, Local and global properties in networks of processors, in *Proc. 12th ACM Symposium on Theory of Computation (STOC'81)* (ACM Press, New York, 1981), pp. 82–93
20. I. Arrieta, F. Fariña, J.-R. Mendívil, M. Raynal, Leader election: from Higham-Przytycka's algorithm to a gracefully degrading algorithm, in *Proc. 6th Int'l Conference on Complex, Intelligent, and Software Intensive Systems (CISIS'12)* (IEEE Press, New York, 2012), pp. 225–232
21. H. Attiya, S. Chaudhuri, R. Friedman, J.L. Welch, Non-sequential consistency conditions for shared memory, in *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures (SPAA'93)* (ACM Press, New York, 1993), pp. 241–250
22. H. Attiya, M. Snir, M. Warmuth, Computing on an anonymous ring. *J. ACM* **35**(4), 845–876 (1988)
23. H. Attiya, J.L. Welch, Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.* **12**(2), 91–122 (1994)
24. H. Attiya, J.L. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, 2nd edn. (Wiley-Interscience, New York, 2004). 414 pages. ISBN 0-471-45324-2
25. B. Awerbuch, A new distributed depth-first search algorithm. *Inf. Process. Lett.* **20**(3), 147–150 (1985)
26. B. Awerbuch, Reducing complexities of the distributed max-flow and breadth-first algorithms by means of network synchronization. *Networks* **15**, 425–437 (1985)
27. B. Awerbuch, Complexity of network synchronization. *J. ACM* **4**, 804–823 (1985)
28. O. Babaoğlu, E. Fromentin, M. Raynal, A unified framework for the specification and the run-time detection of dynamic properties in distributed executions. *J. Syst. Softw.* **33**, 287–298 (1996)
29. O. Babaoğlu, K. Marzullo, Consistent global states of distributed systems: fundamental concepts and mechanisms, in *Distributed Systems* (ACM/Addison-Wesley Press, New York, 1993), pp. 55–93. Chap. 4
30. R. Bagrodia, Process synchronization: design and performance evaluation for distributed algorithms. *IEEE Trans. Softw. Eng.* **SE15**(9), 1053–1065 (1989)
31. R. Bagrodia, Synchronization of asynchronous processes in CSP. *ACM Trans. Program. Lang. Syst.* **11**(4), 585–597 (1989)
32. H.E. Bal, F. Kaashoek, A. Tanenbaum, Orca: a language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.* **18**(3), 180–205 (1992)
33. R. Baldoni, J.M. Hélary, A. Mostéfaoui, M. Raynal, Impossibility of scalar clock-based communication-induced checkpointing protocols ensuring the RDT property. *Inf. Process. Lett.* **80**(2), 105–111 (2001)
34. R. Baldoni, J.M. Hélary, A. Mostéfaoui, M. Raynal, A communication-induced checkpointing protocol that ensures rollback-dependency trackability, in *Proc. 27th IEEE Symposium on Fault-Tolerant Computing (FTCS-27)* (IEEE Press, New York, 1997), pp. 68–77
35. R. Baldoni, J.-M. Hélary, M. Raynal, Consistent records in asynchronous computations. *Acta Inform.* **35**(6), 441–455 (1998)
36. R. Baldoni, J.M. Hélary, M. Raynal, Rollback-dependency trackability: a minimal characterization and its protocol. *Inf. Comput.* **165**(2), 144–173 (2001)
37. R. Baldoni, G. Melideo,  $k$ -dependency vectors: a scalable causality-tracking protocol, in *Proc. 11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP'03)* (2003), pp. 219–226
38. R. Baldoni, A. Mostéfaoui, M. Raynal, Causal delivery of messages with real-time data in unreliable networks. *Real-Time Syst.* **10**(3), 245–262 (1996)
39. R. Baldoni, R. Prakash, M. Raynal, M. Singhal, Efficient delta-causal broadcasting. *Comput. Syst. Sci. Eng.* **13**(5), 263–270 (1998)
40. R. Baldoni, M. Raynal, Fundamentals of distributed computing: a practical tour of vector clock systems. *IEEE Distrib. Syst. Online* **3**(2), 1–18 (2002)

41. D. Barbara, H. Garcia Molina, Mutual exclusion in partitioned distributed systems. *Distrib. Comput.* **1**(2), 119–132 (1986)
42. D. Barbara, H. Garcia Molina, A. Spauster, Increasing availability under mutual exclusion constraints with dynamic vote assignments. *ACM Trans. Comput. Syst.* **7**(7), 394–426 (1989)
43. L. Barenboim, M. Elkin, Deterministic distributed vertex coloring in polylogarithmic time. *J. ACM* **58**(5), 23 (2011), 25 pages
44. R. Bellman, *Dynamic Programming* (Princeton University Press, Princeton, 1957)
45. J.-C. Bermond, C. Delorme, J.-J. Quisquater, Strategies for interconnection networks: some methods from graph theory. *J. Parallel Distrib. Comput.* **3**(4), 433–449 (1986)
46. J.-C. Bermond, J.-C. König, General and efficient decentralized consensus protocols II, in *Proc. Int'l Workshop on Parallel and Distributed Algorithms*, ed. by M. Cosnard, P. Quinton, M. Raynal, Y. Robert (North-Holland, Amsterdam, 1989), pp. 199–210
47. J.-C. Bermond, J.-C. König, Un protocole distribué pour la 2-connexité. *TSI. Tech. Sci. Inform.* **10**(4), 269–274 (1991)
48. J.-C. Bermond, J.-C. König, M. Raynal, General and efficient decentralized consensus protocols, in *Proc. 2nd Int'l Workshop on Distributed Algorithms (WDAG'87)*. LNCS, vol. 312 (Springer, Berlin, 1987), pp. 41–56
49. J.-C. Bermond, C. Peyrat, de Bruijn and Kautz networks: a competitor for the hypercube? in *Proc. Int'l Conference on Hypercube and Distributed Computers* (North-Holland, Amsterdam, 1989), pp. 279–284
50. J.M. Bernabéu-Aubán, M. Ahamad, Applying a path-compression technique to obtain an effective distributed mutual exclusion algorithm, in *Proc. 3rd Int'l Workshop on Distributed Algorithms (WDAG'89)*. LNCS, vol. 392 (Springer, Berlin, 1989), pp. 33–44
51. A.J. Bernstein, Output guards and non-determinism in “communicating sequential processes”. *ACM Trans. Program. Lang. Syst.* **2**(2), 234–238 (1980)
52. B.K. Bhargava, S.-R. Lian, Independent checkpointing and concurrent rollback for recovery in distributed systems: an optimistic approach, in *Proc. 7th IEEE Symposium on Reliable Distributed Systems (SRDS'88)* (IEEE Press, New York, 1988), pp. 3–12
53. K. Birman, T. Joseph, Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.* **5**(1), 47–76 (1987)
54. A.D. Birell, B.J. Nelson, Implementing remote procedure calls. *ACM Trans. Comput. Syst.* **3**, 39–59 (1984)
55. K.P. Birman, A. Schiper, P. Stephenson, Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.* **9**(3), 272–314 (1991)
56. H.L. Bodlaender, Some lower bound results for decentralized extrema finding in ring of processors. *J. Comput. Syst. Sci.* **42**, 97–118 (1991)
57. L. Bougé, Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP. *Theor. Comput. Sci.* **49**, 145–169 (1987)
58. L. Bougé, N. Francez, A compositional approach to super-imposition, in *Proc. 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88)* (ACM Press, New York, 1988), pp. 240–249
59. A. Boukerche, C. Tropper, A distributed graph algorithm for the detection of local cycles and knots. *IEEE Trans. Parallel Distrib. Syst.* **9**(8), 748–757 (1998)
60. Ch. Boulmier, F. Petit, V. Villain, Synchronous vs asynchronous unison. *Algorithmica* **51**(1), 61–80 (2008)
61. G. Bracha, S. Toueg, Distributed deadlock detection. *Distrib. Comput.* **2**(3), 127–138 (1987)
62. D. Briatico, A. Ciuffoletti, L.A. Simoncini, Distributed domino-effect free recovery algorithm, in *4th IEEE Symposium on Reliability in Distributed Software and Database Systems* (IEEE Press, New York, 1984), pp. 207–215
63. P. Brinch Hansen, Distributed processes: a concurrent programming concept. *Commun. ACM* **21**(11), 934–941 (1978)

64. J. Brzezinski, J.-M. Hélary, M. Raynal, Termination detection in a very general distributed computing model, in *Proc. 13th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'93)* (IEEE Press, New York, 1993), pp. 374–381
65. J. Brzezinski, J.-M. Hélary, M. Raynal, M. Singhal, Deadlock models and a general algorithm for distributed deadlock detection. *J. Parallel Distrib. Comput.* **31**(2), 112–125 (1995) (Erratum printed in *Journal of Parallel and Distributed Computing*, **32**(2), 232 (1996))
66. G.N. Buckley, A. Silberschatz, An effective implementation for the generalized input-output construct of CSP. *ACM Trans. Program. Lang. Syst.* **5**(2), 223–235 (1983)
67. C. Cachin, R. Guerraoui, L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd edn. (Springer, Berlin, 2012), 367 pages. ISBN 978-3-642-15259-7
68. G. Cao, M. Singhal, A delay-optimal quorum-based mutual exclusion algorithm for distributed systems. *IEEE Trans. Parallel Distrib. Syst.* **12**(12), 1256–1268 (1991)
69. N. Carriero, D. Gelernter, T.G. Mattson, A.H. Sherman, The Linda alternative to message-passing systems. *Parallel Comput.* **20**(4), 633–655 (1994)
70. O. Carvalho, G. Roucairol, On the distribution of an assertion, in *Proc. First ACM Symposium on Principles of Distributed Computing (PODC'1982)* (ACM Press, New York, 1982), pp. 18–20
71. O. Carvalho, G. Roucairol, On mutual exclusion in computer networks. *Commun. ACM* **26**(2), 146–147 (1983)
72. L.M. Censier, P. Feautrier, A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.* **C-27**(12), 112–118 (1978)
73. P. Chandra, A.K. Kshemkalyani, Causality-based predicate detection across space and time. *IEEE Trans. Comput.* **54**(11), 1438–1453 (2005)
74. S. Chandrasekaran, S. Venkatesan, A message-optimal algorithm for distributed termination detection. *J. Parallel Distrib. Comput.* **8**(3), 245–252 (1990)
75. K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* **3**(1), 63–75 (1985)
76. K.M. Chandy, J. Misra, Deadlock absence proof for networks of communicating processes. *Inf. Process. Lett.* **9**(4), 185–189 (1979)
77. K.M. Chandy, J. Misra, Distributed computation on graphs: shortest path algorithms. *Commun. ACM* **25**(11), 833–837 (1982)
78. K.M. Chandy, J. Misra, The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.* **6**(4), 632–646 (1984)
79. K.M. Chandy, J. Misra, An example of stepwise refinement of distributed programs: quiescence detection. *ACM Trans. Program. Lang. Syst.* **8**(3), 326–343 (1986)
80. K.M. Chandy, J. Misra, *Parallel Program Design* (Addison-Wesley, Reading, 1988), 516 pages
81. K.M. Chandy, J. Misra, L.M. Haas, Distributed deadlock detection. *ACM Trans. Comput. Syst.* **1**(2), 144–156 (1983)
82. E.J.H. Chang, Echo algorithms: depth-first algorithms on graphs. *IEEE Trans. Softw. Eng.* **SE-8**(4), 391–402 (1982)
83. E.J.H. Chang, R. Roberts, An improved algorithm for decentralized extrema finding in circular configurations of processes. *Commun. ACM* **22**(5), 281–283 (1979)
84. J.-M. Chang, N.F. Maxemchuck, Reliable broadcast protocols. *ACM Trans. Comput. Syst.* **2**(3), 251–273 (1984)
85. A. Charlesworth, The multiway rendezvous. *ACM Trans. Program. Lang. Syst.* **9**, 350–366 (1987)
86. B. Charron-Bost, Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.* **39**, 11–16 (1991)
87. B. Charron-Bost, G. Tel, Calcul approché de la borne inférieure de valeurs réparties. *Inform. Théor. Appl.* **31**(4), 305–330 (1997)
88. B. Charron-Bost, G. Tel, F. Mattern, Synchronous, asynchronous, and causally ordered communications. *Distrib. Comput.* **9**(4), 173–191 (1996)

89. C. Chase, V.K. Garg, Detection of global predicates: techniques and their limitations. *Distrib. Comput.* **11**(4), 191–201 (1998)
90. D.R. Cheriton, D. Skeen, Understanding the limitations of causally and totally ordered communication, in *Proc. 14th ACM Symposium on Operating System Principles (SOSP'93)* (ACM Press, New York, 1993), pp. 44–57
91. T.-Y. Cheung, Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Trans. Softw. Eng.* **SE-9**(4), 504–512 (1983)
92. V. Cholvi, A. Fernández, E. Jiménez, P. Manzano, M. Raynal, A methodological construction of an efficient sequentially consistent distributed shared memory. *Comput. J.* **53**(9), 1523–1534 (2010)
93. C.T. Chou, I. Cidon, I. Gopal, S. Zaks, Synchronizing asynchronous bounded delays networks, in *Proc. 2nd Int'l Workshop on Distributed Algorithms (WDAG'87)*. LNCS, vol. 312 (Springer, Berlin, 1987), pp. 212–218
94. M. Choy, A.K. Singh, Efficient implementation of synchronous communication over asynchronous networks. *J. Parallel Distrib. Comput.* **26**, 166–180 (1995)
95. I. Cidon, Yet another distributed depth-first search algorithm. *Inf. Process. Lett.* **26**(6), 301–305 (1988)
96. I. Cidon, An efficient knot detection algorithm. *IEEE Trans. Softw. Eng.* **15**(5), 644–649 (1989)
97. E.G. Coffman Jr., M.J. Elphick, A. Shoshani, System deadlocks. *ACM Comput. Surv.* **3**(2), 67–78 (1971)
98. R. Cooper, K. Marzullo, Consistent detection of global predicates, in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging* (ACM Press, New York, 1991), pp. 163–173
99. Th.H. Cormen, Ch.E. Leiserson, R.L. Rivest, *Introduction to Algorithms* (The MIT Press, Cambridge, 1998), 1028 pages
100. J.-M. Couvreur, N. Francez, M. Gouda, Asynchronous unison, in *Proc. 12th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'92)* (IEEE Press, New York, 1992), pp. 486–493
101. F. Cristian, Probabilistic clock synchronization. *Distrib. Comput.* **3**(3), 146–158 (1989)
102. F. Cristian, H. Aghili, R. Strong, D. Dolev, Atomic broadcast: from simple message diffusion to Byzantine agreement. *Inf. Comput.* **118**(1), 158–179 (1995)
103. F. Cristian, F. Jahanian, A timestamping-based checkpointing protocol for long-lived distributed computations, in *Proc. 10th IEEE Symposium on Reliable Distributed Systems (SRDS'91)* (IEEE Press, New York, 1991), pp. 12–20
104. O.P. Damani, Y.-M. Wang, V.K. Garg, Distributed recovery with  $k$ -optimistic logging. *J. Parallel Distrib. Comput.* **63**(12), 1193–1218 (2003)
105. M.J. Demmer, M. Herlihy, The arrow distributed directory protocol, in *Proc. 12th Int'l Symposium on Distributed Computing (DISC'98)*. LNCS, vol. 1499 (Springer, Berlin, 1998), pp. 119–133
106. P.J. Denning, Virtual memory. *ACM Comput. Surv.* **2**(3), 153–189 (1970)
107. P.J. Denning, J.B. Dennis, J.E. Qualitz, *Machines, Languages and Computation* (Prentice Hall, New York, 1978), 612 pages
108. Cl. Diehl, Cl. Jard, Interval approximations of message causality in distributed executions, in *Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS'92)*. LNCS, vol. 577 (Springer, Berlin, 1992), pp. 363–374
109. E.W. Dijkstra, Solution of a problem in concurrent programming control. *Commun. ACM* **8**(9), 569 (1965)
110. E.W. Dijkstra, The structure of “THE” multiprogramming system. *Commun. ACM* **11**(5), 341–346 (1968)
111. E.W. Dijkstra, Hierarchical ordering of sequential processes. *Acta Inform.* **1**, 115–138 (1971)
112. E.W. Dijkstra, Self stabilizing systems in spite of distributed control. *Commun. ACM* **17**, 643–644 (1974)

113. E.W. Dijkstra, Guarded commands, nondeterminacy, and formal derivation of programs. *Commun. ACM* **18**(8), 453–457 (1979)
114. E.W. Dijkstra, W.H.J. Feijen, A.J.M. van Gasteren, Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett.* **16**(5), 217–219 (1983)
115. E.W.D. Dijkstra, C.S. Scholten, Termination detection for diffusing computations. *Inf. Process. Lett.* **11**(1), 1–4 (1980)
116. D. Dolev, J.Y. Halpern, H.R. Strong, On the possibility and impossibility of achieving clock synchronization. *J. Comput. Syst. Sci.* **33**(2), 230–250 (1986)
117. D. Dolev, M. Klawe, M. Rodeh, An  $O(n \log n)$  unidirectional distributed algorithm for extrema finding in a circle. *J. Algorithms* **3**, 245–260 (1982)
118. S. Dolev, *Self-Stabilization* (The MIT Press, Cambridge, 2000), 197 pages
119. M. Dubois, C. Scheurich, Memory access dependencies in shared memory multiprocessors. *IEEE Trans. Softw. Eng.* **16**(6), 660–673 (1990)
120. E.N. Elnozahy, L. Alvisi, Y.-M. Wang, D.B. Johnson, A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* **34**(3), 375–408 (2002)
121. E. Evangelist, N. Francez, S. Katz, Multiparty interactions for interprocess communication and synchronization. *IEEE Trans. Softw. Eng.* **15**(11), 1417–1426 (1989)
122. S. Even, *Graph Algorithms*, 2nd edn. (Cambridge University Press, Cambridge, 2011), 202 pages (edited by G. Even)
123. A. Fekete, N.A. Lynch, L. Shrira, A modular proof of correctness for a network synchronizer, in *Proc. 2nd Int'l Workshop on Distributed Algorithms (WDAG'87)*. LNCS, vol. 312 (Springer, Berlin, 1987), pp. 219–256
124. C.J. Fidge, Timestamp in message passing systems that preserves partial ordering, in *Proc. 11th Australian Computing Conference* (1988), pp. 56–66
125. C.J. Fidge, Logical time in distributed computing systems. *IEEE Comput.* **24**(8), 28–33 (1991)
126. C.J. Fidge, Limitation of vector timestamps for reconstructing distributed computations. *Inf. Process. Lett.* **68**, 87–91 (1998)
127. M.J. Fischer, A. Michael, Sacrificing serializability to attain high availability of data, in *Proc. First ACM Symposium on Principles of Database Systems (PODS'82)* (ACM Press, New York, 1982), pp. 70–75
128. R.W. Floyd, Algorithm 97: shortest path. *Commun. ACM* **5**(6), 345 (1962)
129. J. Fowler, W. Zwaenepoel, Causal distributed breakpoints, in *Proc. 10th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'90)* (IEEE Press, New York, 1990), pp. 134–141
130. N. Francez, Distributed termination. *ACM Trans. Program. Lang. Syst.* **2**(1), 42–55 (1980)
131. N. Francez, B. Halpern, G. Taubenfeld, Script: a communication abstraction mechanism. *Sci. Comput. Program.* **6**(1), 35–88 (1986)
132. N. Francez, M. Rodeh, Achieving distributed termination without freezing. *IEEE Trans. Softw. Eng.* **8**(3), 287–292 (1982)
133. N. Francez, S. Yemini, Symmetric intertask communication. *ACM Trans. Program. Lang. Syst.* **7**(4), 622–636 (1985)
134. W.R. Franklin, On an improved algorithm for decentralized extrema-finding in circular configurations of processors. *Commun. ACM* **25**(5), 336–337 (1982)
135. U. Fridzke, P. Ingels, A. Mostéfaoui, M. Raynal, Fault-tolerant consensus-based total order multicast. *IEEE Trans. Parallel Distrib. Syst.* **13**(2), 147–157 (2001)
136. R. Friedman, Implementing hybrid consistency with high-level synchronization operations, in *Proc. 12th Annual ACM Symposium on Principles of Distributed Computing (PODC'93)* (ACM Press, New York, 1993), pp. 229–240
137. E. Fromentin, Cl. Jard, G.-V. Jourdan, M. Raynal, On-the-fly analysis of distributed computations. *Inf. Process. Lett.* **54**(5), 267–274 (1995)
138. E. Fromentin, M. Raynal, Shared global states in distributed computations. *J. Comput. Syst. Sci.* **55**(3), 522–528 (1997)

139. E. Fromentin, M. Raynal, V.K. Garg, A.I. Tomlinson, On the fly testing of regular patterns in distributed computations, in *Proc. Int'l Conference on Parallel Processing (ICPP'94)* (1994), pp. 73–76
140. R. Fujimoto, Parallel discrete event simulation. *Commun. ACM* **33**(10), 31–53 (1990)
141. E. Gafni, D. Bertsekas, Distributed algorithms for generating loop-free routes in networks with frequently changing topologies. *IEEE Trans. Commun.* **C-29**(1), 11–18 (1981)
142. R.G. Gallager, Distributed minimum hop algorithms. Tech Report LIDS 1175, MIT, 1982
143. R.G. Gallager, P.A. Humblet, P.M. Spira, A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.* **5**(1), 66–77 (1983)
144. I.C. Garcia, E. Buzato, Progressive construction of consistent global checkpoints, in *Proc. 19th Int'l Conference on Distributed Computing Systems (ICDCS'99)* (IEEE Press, New York, 1999), pp. 55–62
145. I.C. Garcia, L.E. Buzato, On the minimal characterization of the rollback-dependency trackability property, in *Proc. 21st Int'l Conference on Distributed Computing Systems (ICDCS'01)* (IEEE Press, New York, 2001), pp. 342–349
146. I.C. Garcia, L.E. Buzato, An efficient checkpointing protocol for the minimal characterization of operational rollback-dependency trackability, in *Proc. 23rd Int'l Symposium on Reliable Distributed Systems (SRDS'04)* (IEEE Press, New York, 2004), pp. 126–135
147. H. Garcia Molina, D. Barbara, How to assign votes in a distributed system. *J. ACM* **32**(4), 841–860 (1985)
148. M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, New York, 1979), 340 pages
149. V.K. Garg, *Principles of Distributed Systems* (Kluwer Academic, Dordrecht, 1996), 274 pages
150. V.K. Garg, *Elements of Distributed Computing* (Wiley-Interscience, New York, 2002), 423 pages
151. V.K. Garg, M. Raynal, Normality: a consistency condition for concurrent objects. *Parallel Process. Lett.* **9**(1), 123–134 (1999)
152. V.K. Garg, S. Skawratananond, N. Mittal, Timestamping messages and events in a distributed system using synchronous communication. *Distrib. Comput.* **19**(5–6), 387–402 (2007)
153. V.K. Garg, B. Waldecker, Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.* **5**(3), 299–307 (1994)
154. V.K. Garg, B. Waldecker, Detection of strong unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.* **7**(12), 1323–1333 (1996)
155. Ch. Georgiou, A. Shvartsman, *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity* (Springer, Berlin, 2008), 219 pages. ISBN 978-0-387-69045-2
156. K. Gharachorloo, P. Gibbons, Detecting violations of sequential consistency, in *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures (SPAA'91)* (ACM Press, New York, 1991), pp. 316–326
157. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J.L. Hennessy, Memory consistency and event ordering in scalable shared memory multiprocessors, in *Proc. 17th ACM Int'l Symposium on Computer Architecture (ISCA'90)* (1990), pp. 15–26
158. A. Gibbons, *Algorithmic Graph Theory* (Cambridge University Press, Cambridge, 1985), 260 pages
159. D.K. Gifford, Weighted voting for replicated data, in *Proc. 7th ACM Symposium on Operating System Principles (SOSP'79)* (ACM Press, New York, 1979), pp. 150–172
160. V.D. Gligor, S.H. Shattuck, Deadlock detection in distributed systems. *IEEE Trans. Softw. Eng.* **SE-6**(5), 435–440 (1980)
161. A.P. Goldberg, A. Gopal, A. Lowry, R. Strom, Restoring consistent global states of distributed computations, in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging* (ACM Press, New York, 1991), pp. 144–156
162. M. Gouda, T. Herman, Stabilizing unison. *Inf. Process. Lett.* **35**(4), 171–175 (1990)
163. D. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques* (Morgan Kaufmann, San Mateo, 1993), 1060 pages. ISBN 1-55860-190-2

164. J.L. Gross, J. Yellen (eds.), *Graph Theory* (CRC Press, Boca Raton, 2004), 1167 pages
165. H.N. Haberman, Prevention of system deadlocks. *Commun. ACM* **12**(7), 373–377 (1969)
166. J.W. Havender, Avoiding deadlocks in multitasking systems. *IBM Syst. J.* **13**(3), 168–192 (1971)
167. J.-M. Hélarý, Observing global states of asynchronous distributed applications, in *Proc. 3rd Int'l Workshop on Distributed Algorithms (WDAG'87)*. LNCS, vol. 392 (Springer, Berlin, 1987), pp. 124–135
168. J.-M. Hélarý, M. Hurfin, A. Mostéfaoui, M. Raynal, F. Tronel, Computing global functions in asynchronous distributed systems with perfect failure detectors. *IEEE Trans. Parallel Distrib. Syst.* **11**(9), 897–909 (2000)
169. J.-M. Hélarý, C. Jard, N. Plouzeau, M. Raynal, Detection of stable properties in distributed systems, in *Proc. 6th ACM Symposium on Principles of Distributed Computing (PODC'87)* (ACM Press, New York, 1987), pp. 125–136
170. J.-M. Hélarý, A. Maddi, M. Raynal, Controlling information transfers in distributed applications, application to deadlock detection, in *Proc. Int'l IFIP WG 10.3 Conference on Parallel Processing* (North-Holland, Amsterdam, 1987), pp. 85–92
171. J.-M. Hélarý, A. Mostéfaoui, R.H.B. Netzer, M. Raynal, Communication-based prevention of useless checkpoints in distributed computations. *Distrib. Comput.* **13**(1), 29–43 (2000)
172. J.-M. Hélarý, A. Mostéfaoui, M. Raynal, A general scheme for token and tree-based distributed mutual exclusion algorithms. *IEEE Trans. Parallel Distrib. Syst.* **5**(11), 1185–1196 (1994)
173. J.-M. Hélarý, A. Mostéfaoui, M. Raynal, Communication-induced determination of consistent snapshots. *IEEE Trans. Parallel Distrib. Syst.* **10**(9), 865–877 (1999)
174. J.-M. Hélarý, A. Mostéfaoui, M. Raynal, Interval consistency of asynchronous distributed computations. *J. Comput. Syst. Sci.* **64**(2), 329–349 (2002)
175. J.-M. Hélarý, R.H.B. Netzer, M. Raynal, Consistency criteria for distributed checkpoints. *IEEE Trans. Softw. Eng.* **2**(2), 274–281 (1999)
176. J.-M. Hélarý, N. Plouzeau, M. Raynal, A distributed algorithm for mutual exclusion in arbitrary networks. *Comput. J.* **31**(4), 289–295 (1988)
177. J.-M. Hélarý, M. Raynal, Depth-first traversal and virtual ring construction in distributed systems, in *Proc. IFIP WG 10.3 Conference on Parallel Processing* (North-Holland, Amsterdam, 1988), pp. 333–346
178. J.-M. Hélarý, M. Raynal, Vers la construction raisonnée d'algorithmes répartis: le cas de la terminaison. *TSI. Tech. Sci. Inform.* **10**(3), 203–209 (1991)
179. J.-M. Hélarý, M. Raynal, *Synchronization and Control of Distributed Systems and Programs* (Wiley, New York, 1991), 160 pages
180. J.-M. Hélarý, M. Raynal, Towards the construction of distributed detection programs with an application to distributed termination. *Distrib. Comput.* **7**(3), 137–147 (1994)
181. J.-M. Hélarý, M. Raynal, G. Melideo, R. Baldoni, Efficient causality-tracking timestamping. *IEEE Trans. Knowl. Data Eng.* **15**(5), 1239–1250 (2003)
182. M. Herlihy, F. Kuhn, S. Tirthapura, R. Wattenhofer, Dynamic analysis of the arrow distributed protocol. *Theory Comput. Syst.* **39**(6), 875–901 (2006)
183. M. Herlihy, J. Wing, Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3), 463–492 (1990)
184. L. Higham, T. Przytycka, A simple efficient algorithm for maximum finding on rings. *Inf. Process. Lett.* **58**(6), 319–324 (1996)
185. D.S. Hirschberg, J.B. Sinclair, Decentralized extrema finding in circular configuration of processors. *Commun. ACM* **23**, 627–628 (1980)
186. C.A.R. Hoare, Communicating sequential processes. *Commun. ACM* **21**(8), 666–677 (1978)
187. W. Hohberg, How to find biconnected components in distributed networks. *J. Parallel Distrib. Comput.* **9**(4), 374–386 (1990)
188. R.C. Holt, Comments on prevention of system deadlocks. *Commun. ACM* **14**(1), 36–38 (1871)

189. J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 2nd edn. (Addison-Wesley, Reading, 2001), 521 pages
190. S.-T. Huang, Termination detection by using distributed snapshots. *Inf. Process. Lett.* **32**(3), 113–119 (1989)
191. S.-T. Huang, Detecting termination of distributed computations by external agents, in *Proc. 9th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'89)* (IEEE Press, New York, 1989), pp. 79–84
192. M. Hurfin, N. Plouzeau, M. Raynal, Detecting atomic sequences of predicates in distributed computations. *SIGPLAN Not.* **28**(12), 32–42 (1993). *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*
193. M. Hurfin, M. Mizuno, M. Raynal, S. Singhal, Efficient distributed detection of conjunctions of local predicates. *IEEE Trans. Softw. Eng.* **24**(8), 664–677 (1998)
194. P. Hutto, M. Ahamad, Slow memory: weakening consistency to enhance concurrency in distributed shared memories, in *Proc. 10th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'90)* (IEEE Press, New York, 1990), pp. 302–311
195. T. Ibaraki, T. Kameda, A theory of coterries: mutual exclusion in distributed systems. *J. Parallel Distrib. Comput.* **4**(7), 779–794 (1993)
196. T. Ibaraki, T. Kameda, T. Minoura, Serializability with constraints. *ACM Trans. Database Syst.* **12**(3), 429–452 (1987)
197. R. Ingram, P. Shields, J.E. Walter, J.L. Welch, An asynchronous leader election algorithm for dynamic networks, in *Proc. 23rd Int'l IEEE Parallel and Distributed Processing Symposium (IPDPS'09)* (IEEE Press, New York, 2009), pp. 1–12
198. Cl. Jard, G.-V. Jourdan, Incremental transitive dependency tracking in distributed computations. *Parallel Process. Lett.* **6**(3), 427–435 (1996)
199. J. Jefferson, Virtual time. *ACM Trans. Program. Lang. Syst.* **7**(3), 404–425 (1985)
200. E. Jiménez, A. Fernández, V. Cholvi, A parameterized algorithm that implements sequential, causal, and cache memory consistencies. *J. Syst. Softw.* **81**(1), 120–131 (2008)
201. Ö. Johansson, Simple distributed  $(\Delta + 1)$ -coloring of graphs. *Inf. Process. Lett.* **70**(5), 229–232 (1999)
202. D.B. Johnson, W. Zwaenepoel, Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms* **11**(3), 462–491 (1990)
203. S. Kanchi, D. Vineyard, An optimal distributed algorithm for all-pairs shortest-path. *Int. J. Inf. Theories Appl.* **11**(2), 141–146 (2004)
204. P. Keleher, A.L. Cox, W. Zwaenepoel, Lazy release consistency for software distributed shared memory, in *Proc. 19th ACM Int'l Symposium on Computer Architecture (ISCA'92)*, (1992), pp. 13–21
205. J. Kleinberg, E. Tardos, *Algorithm Design* (Addison-Wesley, Reading, 2005), 838 pages
206. P. Knapp, Deadlock detection in distributed databases. *ACM Comput. Surv.* **19**(4), 303–328 (1987)
207. R. Koo, S. Toueg, Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.* **13**(1), 23–31 (1987)
208. E. Korach, S. Moran, S. Zaks, Tight lower and upper bounds for some distributed algorithms for a complete network of processors, in *Proc. 4th ACM Symposium on Principles of Distributed Computing (PODC'84)* (ACM Press, New York, 1984), pp. 199–207
209. E. Korach, S. Moran, S. Zaks, The optimality of distributive constructions of minimum weight and degree restricted spanning tree in complete networks of processes. *SIAM J. Comput.* **16**(2), 231–236 (1987)
210. E. Korach, D. Rotem, N. Santoro, Distributed algorithms for finding centers and medians in networks. *ACM Trans. Program. Lang. Syst.* **6**(3), 380–401 (1984)
211. E. Korach, G. Tel, S. Zaks, Optimal synchronization of ABD networks, in *Proc. Int'l Conference on Concurrency*. LNCS, vol. 335 (Springer, Berlin, 1988), pp. 353–367
212. R. Kordale, M. Ahamad, A scalable technique for implementing multiple consistency levels for distributed objects, in *Proc. 16th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'96)* (IEEE Press, New York, 1996), pp. 369–376

213. A.D. Kshemkalyani, Fast and message-efficient global snapshot algorithms for large-scale distributed systems. *IEEE Trans. Parallel Distrib. Syst.* **21**(9), 1281–1289 (2010)
214. A.D. Kshemkalyani, M. Raynal, M. Singhal, Global snapshots of a distributed systems. *Distrib. Syst. Eng.* **2**(4), 224–233 (1995)
215. A.D. Kshemkalyani, M. Singhal, Invariant-based verification of a distributed deadlock detection algorithm. *IEEE Trans. Softw. Eng.* **17**(8), 789–799 (1991)
216. A.D. Kshemkalyani, M. Singhal, Efficient detection and resolution of generalized distributed deadlocks. *IEEE Trans. Softw. Eng.* **20**(1), 43–54 (1994)
217. A.D. Kshemkalyani, M. Singhal, Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distrib. Comput.* **11**(2), 91–111 (1998)
218. A.D. Kshemkalyani, M. Singhal, A one-phase algorithm to detect distributed deadlocks in replicated databases. *IEEE Trans. Knowl. Data Eng.* **11**(6), 880–895 (1999)
219. A.D. Kshemkalyani, M. Singhal, *Distributed Computing: Principles, Algorithms and Systems* (Cambridge University Press, Cambridge, 2008), 736 pages
220. A.D. Kshemkalyani, M. Singhal, Efficient distributed snapshots in an anonymous asynchronous message-passing system. *J. Parallel Distrib. Comput.* **73**, 621–629 (2013)
221. T.-H. Lai, Termination detection for dynamically distributed systems with non-first-in-first-out communication. *J. Parallel Distrib. Comput.* **3**(4), 577–599 (1986)
222. T.H. Lai, T.H. Yang, On distributed snapshots. *Inf. Process. Lett.* **25**, 153–158 (1987)
223. T.V. Lakshman, A.K. Agrawala, Efficient decentralized consensus protocols. *IEEE Trans. Softw. Eng.* **SE-12**(5), 600–607 (1986)
224. K.B. Lakshmanan, N. Meenakshi, K. Thulisaraman, A time-optimal message-efficient distributed algorithm for depth-first search. *Inf. Process. Lett.* **25**, 103–109 (1987)
225. K.B. Lakshmanan, K. Thulisaraman, On the use of synchronizers for asynchronous communication networks, in *Proc. 2nd Int'l Workshop on Distributed Algorithms (WDAG'87)*. LNCS, vol. 312 (Springer, Berlin, 1987), pp. 257–267
226. L. Lamport, Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978)
227. L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* **C-28**(9), 690–691 (1979)
228. L. Lamport, On inter-process communications, part I: basic formalism. *Distrib. Comput.* **1**(2), 77–85 (1986)
229. L. Lamport, On inter-process communications, part II: algorithms. *Distrib. Comput.* **1**(2), 86–101 (1986)
230. L. Lamport, P.M. Melliar-Smith, Synchronizing clocks in the presence of faults. *J. ACM* **32**(1), 52–78 (1985)
231. Y. Lavallée, G. Roucairol, A fully distributed minimal spanning tree algorithm. *Inf. Process. Lett.* **23**(2), 55–62 (1986)
232. G. Le Lann, Distributed systems: towards a formal approach, in *IFIP World Congress*, (1977), pp. 155–160
233. I. Lee, S.B. Davidson, Adding time to synchronous processes. *IEEE Trans. Comput.* **C-36**(8), 941–948 (1987)
234. K. Li, K.P. Huda, Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.* **7**(4), 321–359 (1989)
235. T.F. Li, Th. Radhakrishnan, K. Venkatesh, Global state detection in non-FIFO networks, in *Proc. 7th Int'l Conference on Distributed Computing Systems (ICDCS'87)* (IEEE Press, New York, 1987), pp. 364–370
236. N. Linial, Locality in distributed graph algorithms. *SIAM J. Comput.* **21**(1), 193–201 (1992)
237. R.J. Lipton, J.S. Sandberg, PRAM: a scalable shared memory. Tech Report CS-TR-180-88, Princeton University, 1988
238. B. Liskov, R. Ladin, Highly available distributed services and fault-tolerant distributed garbage collection, in *Proc. 5th ACM Symposium on Principles of Distributed Computing (PODC'86)* (ACM Press, New York, 1986), pp. 29–39

239. S. Lodha, A.D. Ksemkalyani, A fair distributed mutual exclusion algorithm. *IEEE Trans. Parallel Distrib. Syst.* **11**(6), 537–549 (2000)
240. M. Luby, A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.* **15**(4), 1036–1053 (1987)
241. N.A. Lynch, Upper bounds for static resource allocation in a distributed system. *J. Comput. Syst. Sci.* **23**(2), 254–278 (1981)
242. N.A. Lynch, *Distributed Algorithms* (Morgan Kaufmann, San Francisco, 1996), 872 pages
243. M. Maekawa, A  $\sqrt{n}$  algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.* **3**(2), 145–159 (1985)
244. N. Malpani, J.L. Welch, N. Vaidya, Leader election algorithms for mobile ad hoc networks, in *Proc. 4th Int'l ACM Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M'00)* (ACM Press, New York, 2000), pp. 96–103
245. Y. Manabe, R. Baldoni, M. Raynal, S. Aoyagi,  $k$ -arbiter: a safe and general scheme for  $h$ -out-of- $k$  mutual exclusion. *Theor. Comput. Sci.* **193**(1–2), 97–112 (1998)
246. D. Manivannan, R.H.B. Netzer, M. Singhal, Finding consistent global checkpoints in a distributed computation. *IEEE Trans. Parallel Distrib. Syst.* **8**(6), 623–627 (1997)
247. D. Manivannan, M. Singhal, A low overhead recovery technique using quasi-synchronous checkpointing, in *Proc. 16th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'96)* (IEEE Press, New York, 1996), pp. 100–107
248. D. Manivannan, M. Singhal, An efficient distributed algorithm for detection of knots and cycles in a distributed graph. *IEEE Trans. Parallel Distrib. Syst.* **14**(10), 961–972 (2003)
249. F. Mattern, Algorithms for distributed termination detection. *Distrib. Comput.* **2**(3), 161–175 (1987)
250. F. Mattern, Virtual time and global states of distributed systems, in *Proc. Parallel and Distributed Algorithms Conference*, ed. by M. Cosnard, P. Quinton, M. Raynal, Y. Robert (North-Holland, Amsterdam, 1988), pp. 215–226
251. F. Mattern, Global quiescence detection based on credit distribution and recovery. *Inf. Process. Lett.* **30**(4), 195–200 (1989)
252. F. Mattern, An efficient distributed termination test. *Inf. Process. Lett.* **31**(4), 203–208 (1989)
253. F. Mattern, Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.* **18**, 423–434 (1993)
254. F. Mattern, Distributed algorithms and causally consistent observations, in *Proc. 16th Int'l Conference on Application and Theory of Petri Nets, (Invited Paper)*. LNCS, vol. 935 (Springer, Berlin, 1995), pp. 21–22
255. F. Mattern, S. Fünfrocken, A non-blocking lightweight implementation of causal order message delivery, in *Proc. Int'l Dagstuhl Workshop on Theory and Practice in Distributed Systems*. LNCS, vol. 938 (Springer, Berlin, 1995), pp. 197–213
256. M. Mavronicolas, D. Roth, Efficient, strong consistent implementations of shared memory, in *Proc. 6th Int'l Workshop on Distributed Algorithms (WDAG'92)*. LNCS, vol. 647 (Springer, Berlin, 1992), pp. 346–361
257. J. Mayo, Ph. Kearns, Efficient distributed termination detection with roughly synchronized clocks. *Inf. Process. Lett.* **52**(2), 105–108 (1994)
258. K. Mehlhorn, P. Sanders, *Algorithms and Data Structures* (Springer, Berlin, 2008), 300 pages
259. D. Menasce, R. Muntz, Locking and deadlock detection in distributed database. *IEEE Trans. Softw. Eng.* **SE-5**(3), 195–202 (1979)
260. J.R. Mendívil, F. Fariña, C.F. Garitagoitia, C.F. Alastruey, J.M. Barnabeu-Auban, A distributed deadlock resolution algorithm for the AND model. *IEEE Trans. Parallel Distrib. Syst.* **10**(5), 433–447 (1999)
261. J. Misra, Detecting termination of distributed computations using markers, in *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)* (ACM Press, New York, 1983), pp. 290–294
262. J. Misra, Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst.* **8**(1), 142–153 (1986)

263. J. Misra, Distributed discrete event simulation. *ACM Comput. Surv.* **18**(1), 39–65 (1986)
264. J. Misra, K.M. Chandy, A distributed graph algorithm: knot detection. *ACM Trans. Program. Lang. Syst.* **4**(4), 678–686 (1982)
265. J. Misra, K.M. Chandy, Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst.* **4**(1), 37–43 (1982)
266. D.P. Mitchell, M. Merritt, A distributed algorithm for deadlock detection and resolution, in *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC'84)* (ACM Press, New York, 1984), pp. 282–284
267. N. Mittal, V.K. Garg, Consistency conditions for multi-objects operations, in *Proc. 18th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'98)* (IEEE Press, New York, 1998), pp. 582–589
268. M. Mizuno, M.L. Nielsen, M. Raynal, An optimistic protocol for a linearizable distributed shared memory service. *Parallel Process. Lett.* **6**(2), 265–278 (1996)
269. M. Mizuno, M. Raynal, J.Z. Zhou, Sequential consistency in distributed systems, in *Int'l Dagstuhl Workshop on the Theory and Practice in Distributed Systems*. LNCS, vol. 938 (Springer, Berlin, 1994), pp. 224–241
270. B. Moret, *The Theory of Computation* (Addison-Wesley, Reading, 1998), 453 pages
271. A. Mostéfaoui, M. Raynal, Efficient message logging for uncoordinated checkpointing protocols, in *Proc. 2nd European Dependable Computing Conference (EDCC'96)*. LNCS, vol. 1150 (Springer, Berlin, 1996), pp. 353–364
272. A. Mostéfaoui, M. Raynal, P. Verissimo, Logically instantaneous communication on top of distributed memory parallel machines, in *Proc. 5th Int'l Conference on Parallel Computing Technologies (PACT'99)*. LNCS, vol. 1662 (Springer, Berlin, 1999), pp. 258–270
273. V.V. Murty, V.K. Garg, An algorithm to guarantee synchronous ordering of messages, in *Proc. 2nd Int'l IEEE Symposium on Autonomous Decentralized Systems* (IEEE Press, New York, 1995), pp. 208–214
274. V.V. Murty, V.K. Garg, Characterization of message ordering specifications and protocols, in *Proc. 7th Int'l Conference on Distributed Computer Systems (ICDCS'97)* (IEEE Press, New York, 1997), pp. 492–499
275. M. Naimi, M. Trehel, An improvement of the  $\log n$  distributed algorithm for mutual exclusion, in *Proc. 7th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'87)* (IEEE Press, New York, 1987), pp. 371–375
276. M. Naimi, M. Trehel, A. Arnold, A  $\log(n)$  distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.* **34**(1), 1–13 (1996)
277. M. Naor, A. Wool, The load, capacity and availability of quorums systems. *SIAM J. Comput.* **27**(2), 423–447 (2008)
278. N. Nararajan, A distributed scheme for detecting communication deadlocks. *IEEE Trans. Softw. Eng.* **12**(4), 531–537 (1986)
279. M.L. Neilsen, M. Mizuno, A DAG-based algorithm for distributed mutual exclusion, in *Proc. 11th IEEE Int'l IEEE Conference on Distributed Computing Systems (ICDCS'91)* (IEEE Press, New York, 1991), pp. 354–360
280. M.L. Neilsen, M. Masaaki, Nondominated  $k$ -coterics for multiple mutual exclusion. *Inf. Process. Lett.* **50**(5), 247–252 (1994)
281. M.L. Neilsen, M. Masaaki, M. Raynal, A general method to define quorums, in *Proc. 12th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'92)* (IEEE Press, New York, 1992), pp. 657–664
282. M. Nesterenko, M. Mizuno, A quorum-based self-stabilizing distributed mutual exclusion algorithm. *J. Parallel Distrib. Comput.* **62**(2), 284–305 (2002)
283. R.H.B. Netzer, J. Xu, Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.* **6**(2), 165–169 (1995)
284. N. Neves, W.K. Fuchs, Adaptive recovery for mobile environments. *Commun. ACM* **40**(1), 68–74 (1997)
285. S. Nishio, K.F. Li, F.G. Manning, A resilient distributed mutual exclusion algorithm for computer networks. *IEEE Trans. Parallel Distrib. Syst.* **1**(3), 344–356 (1990)

286. B. Nitzberg, V. Lo, Distributed shared memory: a survey of issues and algorithms. *IEEE Comput.* **24**(8), 52–60 (1991)
287. R. Obermarck, Distributed deadlock detection algorithm. *ACM Trans. Database Syst.* **7**(2), 197–208 (1982)
288. J.K. Pachi, E. Korach, D. Rotem, Lower bounds for distributed maximum-finding algorithms. *J. ACM* **31**(4), 905–918 (1984)
289. Ch.H. Papadimitriou, The serializability of concurrent database updates. *J. ACM* **26**(4), 631–653 (1979)
290. D.S. Parker, G.L. Popek, G. Rudisin, L. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D.A. Edwards, S. Kiser, C.S. Kline, Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.* **SE9**(3), 240–246 (1983)
291. B. Patt-Shamir, S. Rajsbaum, A theory of clock synchronization, in *Proc. 26th Annual ACM Symposium on Theory of Computing (STOC'94)* (ACM Press, New York, 1994), pp. 810–819
292. D. Peleg, *Distributed Computing: A Locally-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications (2000), 343 pages
293. D. Peleg, J.D. Ullman, An optimal synchronizer for the hypercube. *SIAM J. Comput.* **18**, 740–747 (1989)
294. D. Peleg, A. Wool, Crumbling walls: a class of practical and efficient quorum systems. *Distrib. Comput.* **10**(2), 87–97 (1997)
295. G.L. Peterson, An  $O(n \log n)$  unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.* **4**(4), 758–762 (1982)
296. L.L. Peterson, N.C. Bucholz, R.D. Schlichting, Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.* **7**(3), 217–246 (1989)
297. S.E. Pomares Hernandez, J.R. Perez Cruz, M. Raynal, From the happened before relation to the causal ordered set abstraction. *J. Parallel Distrib. Comput.* **72**, 791–795 (2012)
298. R. Prakash, M. Raynal, M. Singhal, An adaptive causal ordering algorithm suited to mobile computing environments. *J. Parallel Distrib. Comput.* **41**(1), 190–204 (1997)
299. R. Prakash, M. Singhal, Low-cost checkpointing and failure recovery in mobile computing systems. *IEEE Trans. Parallel Distrib. Syst.* **7**(10), 1035–1048 (1996)
300. R. Prakash, M. Singhal, Dependency sequences and hierarchical clocks: efficient alternatives to vector clocks for mobile computing systems. *Wirel. Netw.* **3**(5), 349–360 (1997)
301. J. Protic, M. Tomasevic, Distributed shared memory: concepts and systems. *IEEE Concurr.* **4**(2), 63–79 (1996)
302. S.P. Rana, A distributed solution of the distributed termination problem. *Inf. Process. Lett.* **17**(1), 43–46 (1983)
303. B. Randell, System structure for software fault-tolerance. *IEEE Trans. Softw. Eng.* **SE1**(2), 220–232 (1975)
304. K. Raymond, A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.* **7**(1), 61–77 (1989)
305. K. Raymond, A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.* **30**(4), 189–193 (1989)
306. M. Raynal, *Algorithms for Mutual Exclusion* (The MIT Press, Cambridge, 1986), 107 pages. ISBN 0-262-18119-3
307. M. Raynal, A distributed algorithm to prevent mutual drift between  $n$  logical clocks. *Inf. Process. Lett.* **24**, 199–202 (1987)
308. M. Raynal, *Networks and Distributed Computation: Concepts, Tools and Algorithms* (The MIT Press, Cambridge, 1987), 168 pages. ISBN 0-262-18130-4
309. M. Raynal, Prime numbers as a tool to design distributed algorithms. *Inf. Process. Lett.* **33**, 53–58 (1989)
310. M. Raynal, A simple taxonomy of distributed mutual exclusion algorithms. *Oper. Syst. Rev.* **25**(2), 47–50 (1991)

311. M. Raynal, A distributed solution to the  $k$ -out-of- $M$  resource allocation problem, in *Proc. Int'l Conference on Computing and Information*. LNCS, vol. 497 (Springer, Berlin, 1991), pp. 509–518
312. M. Raynal, Illustrating the use of vector clocks in property detection: an example and a counter-example, in *Proc. 5th European Conference on Parallelism (EUROPAR'99)*. LNCS, vol. 1685 (Springer, Berlin, 1999), pp. 806–814
313. M. Raynal, Sequential consistency as lazy linearizability, in *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)* (ACM Press, New York, 2002), pp. 151–152
314. M. Raynal, Token-based sequential consistency. *Comput. Syst. Sci. Eng.* **17**(6), 359–365 (2002)
315. M. Raynal, *Fault-Tolerant Agreement in Synchronous Distributed Systems* (Morgan & Claypool, San Francisco, 2010), 167 pages. ISBN 9781608455256
316. M. Raynal, *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems* (Morgan & Claypool, San Francisco, 2010), 251 pages. ISBN 9781608452934
317. M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations* (Springer, Berlin, 2012), 500 pages. ISBN 978-3-642-32026-2
318. M. Raynal, M. Ahamad, Exploiting write semantics in implementing partially replicated causal objects, in *Proc. 6th EUROMICRO Conference on Parallel and Distributed Processing (PDP'98)* (IEEE Press, New York, 1998), pp. 157–163
319. M. Raynal, J.-M. H elary, *Synchronization and Control of Distributed Systems and Programs*. Wiley Series in Parallel Computing (1991), 126 pages. ISBN 0-471-92453-9
320. M. Raynal, M. Roy, C. Tutu, A simple protocol offering both atomic consistent read operations and sequentially consistent read operations, in *Proc. 19th Int'l Conference on Advanced Information Networking and Applications (AINA'05)* (IEEE Press, New York, 2005), pp. 961–966
321. M. Raynal, G. Rubino, An algorithm to detect token loss on a logical ring and to regenerate lost tokens, in *Int'l Conference on Parallel Processing and Applications* (North-Holland, Amsterdam, 1987), pp. 457–467
322. M. Raynal, A. Schiper, From causal consistency to sequential consistency in shared memory systems, in *Proc. 15th Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'95)*. LNCS, vol. 1026 (Springer, Berlin, 1995), pp. 180–194
323. M. Raynal, A. Schiper, A suite of formal definitions for consistency criteria in distributed shared memories, in *Proc. 9th Int'l IEEE Conference on Parallel and Distributed Computing Systems (PDCS'96)* (IEEE Press, New York, 1996), pp. 125–131
324. M. Raynal, A. Schiper, S. Toueg, The causal ordering abstraction and a simple way to implement. *Inf. Process. Lett.* **39**(6), 343–350 (1991)
325. M. Raynal, S. Singhal, Logical time: capturing causality in distributed systems. *IEEE Comput.* **29**(2), 49–57 (1996)
326. M. Raynal, K. Vidyasankar, A distributed implementation of sequential consistency with multi-object operations, in *Proc. 24th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'04)* (IEEE Press, New York, 2004), pp. 544–551
327. G. Ricart, A.K. Agrawala, An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM* **24**(1), 9–17 (1981)
328. G. Ricart, A.K. Agrawala, Author response to “on mutual exclusion in computer networks” by Carvalho and Roucairol. *Commun. ACM* **26**(2), 147–148 (1983)
329. R. Righter, J.C. Walrand, Distributed simulation of discrete event systems. *Proc. IEEE* **77**(1), 99–113 (1989)
330. S. Ronn, H. Saikkonen, Distributed termination detection with counters. *Inf. Process. Lett.* **34**(5), 223–227 (1990)
331. D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis, System level concurrency control in distributed databases. *ACM Trans. Database Syst.* **3**(2), 178–198 (1978)

332. D.L. Russell, State restoration in systems of communicating processes. *IEEE Trans. Softw. Eng.* **SE6**(2), 183–194 (1980)
333. B. Sanders, The information structure of distributed mutual exclusion algorithms. *ACM Trans. Comput. Syst.* **5**(3), 284–299 (1987)
334. S.K. Sarin, N.A. Lynch, Discarding obsolete information in a replicated database system. *IEEE Trans. Softw. Eng.* **13**(1), 39–46 (1987)
335. N. Santoro, *Design and Analysis of Distributed Algorithms* (Wiley, New York, 2007), 589 pages
336. A. Schiper, J. Eggli, A. Sandoz, A new algorithm to implement causal ordering, in *Proc. 3rd Int'l Workshop on Distributed Algorithms (WDAG'89)*. LNCS, vol. 392 (Springer, Berlin, 1989), pp. 219–232
337. R. Schmid, I.C. Garcia, F. Pedone, L.E. Buzato, Optimal asynchronous garbage collection for RDT checkpointing protocols, in *Proc. 25th Int'l Conference on Distributed Computing Systems (ICDCS'01)* (IEEE Press, New York, 2005), pp. 167–176
338. F. Schmuck, The use of efficient broadcast in asynchronous distributed systems. Doctoral Dissertation, Tech. Report TR88-928, Dept of Computer Science, Cornell University, 124 pages, 1988
339. F.B. Schneider, Implementing fault-tolerant services using the state machine approach. *ACM Comput. Surv.* **22**(4), 299–319 (1990)
340. R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: in search of the Holy Grail. *Distrib. Comput.* **7**, 149–174 (1994)
341. A. Segall, Distributed network protocols. *IEEE Trans. Inf. Theory* **29**(1), 23–35 (1983)
342. N. Shavit, N. Francez, A new approach to detection of locally indicative stability, in *13th Int'l Colloquium on Automata, Languages and Programming (ICALP'86)*. LNCS, vol. 226 (Springer, Berlin, 1986), pp. 344–358
343. A. Silberschatz, Synchronization and communication in distributed systems. *IEEE Trans. Softw. Eng.* **SE5**(6), 542–546 (1979)
344. L.M. Silva, J.G. Silva, Global checkpoints for distributed programs, in *Proc. 11th Symposium on Reliable Distributed Systems (SRDS'92)* (IEEE Press, New York, 1992), pp. 155–162
345. M. Singhal, A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Trans. Comput.* **38**(5), 651–662 (1989)
346. M. Singhal, Deadlock detection in distributed systems. *IEEE Comput.* **22**(11), 37–48 (1989)
347. M. Singhal, A class of deadlock-free Maekawa-type algorithms for mutual exclusion in distributed systems. *Distrib. Comput.* **4**(3), 131–138 (1991)
348. M. Singhal, A dynamic information-structure mutual exclusion algorithm for distributed systems. *IEEE Trans. Parallel Distrib. Syst.* **3**(1), 121–125 (1992)
349. M. Singhal, A taxonomy of distributed mutual exclusion. *J. Parallel Distrib. Comput.* **18**(1), 94–101 (1993)
350. M. Singhal, A.D. Kshemkalyani, An efficient implementation of vector clocks. *Inf. Process. Lett.* **43**, 47–52 (1992)
351. M. Sipser, *Introduction to the Theory of Computation* (PWS, Boston, 1996), 396 pages
352. A.P. Sistla, J.L. Welch, Efficient distributed recovery using message logging, in *Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC'89)* (ACM Press, New York, 1989), pp. 223–238
353. D. Skeen, A quorum-based commit protocol, in *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (1982), pp. 69–80
354. J.L.A. van de Snepscheut, Synchronous communication between asynchronous components. *Inf. Process. Lett.* **13**(3), 127–130 (1981)
355. J.L.A. van de Snepscheut, Fair mutual exclusion on a graph of processes. *Distrib. Comput.* **2**(2), 113–115 (1987)
356. T. Soneoka, T. Ibaraki, Logically instantaneous message passing in asynchronous distributed systems. *IEEE Trans. Comput.* **43**(5), 513–527 (1994)
357. M. Spezialetti, Ph. Kearns, Efficient distributed snapshots, in *Proc. 6th Int'l Conference on Distributed Computing Systems (ICDCS'86)* (IEEE Press, New York, 1986), pp. 382–388

358. T.K. Srikanth, S. Toueg, Optimal clock synchronization. *J. ACM* **34**(3), 626–645 (1987)
359. M. van Steen, *Graph Theory and Complex Networks: An Introduction* (2011), 285 pages. ISBN 978-90-815406-1-2
360. R.E. Strom, S. Yemini, Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.* **3**(3), 204–226 (1985)
361. I. Suzuki, T. Kasami, A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.* **3**(4), 344–349 (1985)
362. G. Taubenfeld, *Synchronization Algorithms and Concurrent Programming* (Pearson Prentice-Hall, Upper Saddle River, 2006), 423 pages. ISBN 0-131-97259-6
363. K. Taylor, The role of inhibition in asynchronous consistent-cut protocols, in *Proc. 3rd Int'l Workshop on Distributed Algorithms (WDAG'87)*. LNCS, vol. 392 (Springer, Berlin, 1987), pp. 280–291
364. R.N. Taylor, Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inform.* **19**(1), 57–84 (1983)
365. G. Tel, *Introduction to Distributed Algorithms*, 2nd edn. (Cambridge University Press, Cambridge, 2000), 596 pages. ISBN 0-521-79483-8
366. G. Tel, F. Mattern, The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Trans. Program. Lang. Syst.* **15**(1), 1–35 (1993)
367. G. Tel, R.B. Tan, J. van Leeuwen, The derivation of graph-marking algorithms from distributed termination detection protocols. *Sci. Comput. Program.* **10**(1), 107–137 (1988)
368. R.H. Thomas, A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.* **4**(2), 180–209 (1979)
369. O. Theel, M. Raynal, Static and dynamic adaptation of transactional consistency, in *Proc. 30th Hawai'i, Int'l Conference on Systems Sciences (HICSS-30)*, vol. I (1997), pp. 533–542
370. F.J. Torres-Rojas, M. Ahamad, Plausible clocks: constant size logical clocks for distributed systems. *Distrib. Comput.* **12**(4), 179–195 (1999)
371. F. Torres-Rojas, M. Ahamad, M. Raynal, Lifetime-based consistency protocols for distributed objects, in *Proc. 12th Int'l Symposium on Distributed Computing (DISC'98)*. LNCS, vol. 1499 (Springer, Berlin, 1998), pp. 378–392
372. F. Torres-Rojas, M. Ahamad, M. Raynal, Timed consistency for shared distributed objects, in *Proc. 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)* (ACM Press, New York, 1999), pp. 163–172
373. S. Toueg, An all-pairs shortest paths distributed algorithm. IBM Technical Report RC 8327, 1980
374. M. Trehel, M. Naimi, Un algorithme distribué d'exclusion mutuelle en  $\log n$ . *TSI. Tech. Sci. Inform.* **6**(2), 141–150 (1987)
375. J. Tsai, S.-Y. Kuo, Y.-M. Wang, Theoretical analysis for communication-induced checkpointing protocols with rollback-dependency trackability. *IEEE Trans. Parallel Distrib. Syst.* **9**(10), 963–971 (1998)
376. J. Tsai, Y.-M. Wang, S.-Y. Kuo, Evaluations of domino-free communication-induced checkpointing protocols. *Inf. Process. Lett.* **69**(1), 31–37 (1999)
377. S. Venkatesan, Message optimal incremental snapshots, in *Proc. 9th Int'l Conference on Distributed Computing Systems (ICDCS'89)* (IEEE Press, New York, 1989), pp. 53–60
378. S. Venkatesan, B. Dathan, Testing and debugging distributed programs using global predicates. *IEEE Trans. Softw. Eng.* **21**(2), 163–177 (1995)
379. J. Villadangos, F. Fariña, J.R. Mendivil, C.F. Garitagoitia, A. Cordoba, A safe algorithm for resolving OR deadlocks. *IEEE Trans. Softw. Eng.* **29**(7), 608–622 (2003)
380. M. Vukolić, *Quorum Systems with Applications to Storage and Consensus* (Morgan & Claypool, San Francisco, 2012), 130 pages. ISBN 798-1-60845-683-3
381. Y.-M. Wang, Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Trans. Comput.* **46**(4), 456–468 (1997)
382. Y.-M. Wang, P.Y. Chung, I.J. Lin, W.K. Fuchs, Checkpointing space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Trans. Parallel Distrib. Syst.* **6**(5), 546–554 (1995)

383. Y.-M. Wang, W.K. Fuchs, Optimistic message logging for independent checkpointing in message-passing systems, in *Proc. 11th Symposium on Reliable Distributed Systems (SRDS'92)* (IEEE Press, New York, 1992), pp. 147–154
384. S. Warshall, A theorem on Boolean matrices. *J. ACM* **9**(1), 11–12 (1962)
385. J.L. Welch, Simulating synchronous processors. *Inf. Comput.* **74**, 159–171 (1987)
386. J.L. Welch, N.A. Lynch, A new fault-tolerance algorithm for clock synchronization. *Inf. Comput.* **77**(1), 1–36 (1988)
387. J.L. Welch, N.A. Lynch, A modular drinking philosophers algorithm. *Distrib. Comput.* **6**(4), 233–244 (1993)
388. J.L. Welch, J.E. Walter, *Link Reversal Algorithms* (Morgan & Claypool, San Francisco, 2011), 93 pages. ISBN 9781608450411
389. H. Wu, W.-N. Chin, J. Jaffar, An efficient distributed deadlock avoidance algorithm for the AND model. *IEEE Trans. Softw. Eng.* **28**(1), 18–29 (2002)
390. G.T.J. Wu, A.J. Bernstein, Efficient solutions to the replicated log and dictionary problems, in *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)* (ACM Press, New York, 1984), pp. 233–242
391. G.T.J. Wu, A.J. Bernstein, False deadlock detection in distributed systems. *IEEE Trans. Softw. Eng.* **SE-11**(8), 820–821 (1985)
392. M. Yamashita, T. Kameda, Computing on anonymous networks, part I: characterizing the solvable cases. *IEEE Trans. Parallel Distrib. Syst.* **7**(1), 69–89 (1996)
393. M. Yamashita, T. Kameda, Computing on anonymous networks, part II: decision and membership problems. *IEEE Trans. Parallel Distrib. Syst.* **7**(1), 90–96 (1996)
394. L.-Y. Yen, T.-L. Huang, Resetting vector clocks in distributed systems. *J. Parallel Distrib. Comput.* **43**, 15–20 (1997)
395. Y. Zhu, C.-T. Cheung, A new distributed breadth-first search algorithm. *Inf. Process. Lett.* **25**(5), 329–333 (1987)

# Index

## A

- Abstraction
  - Checkpointing, 196
- Adaptive algorithm, 108
  - Bounded mutual exclusion, 259
  - Causal order, 312
  - Mutual exclusion, 256
- AND model
  - Deadlock detection, 403
  - Receive statement, 386
- Anonymous systems, 78
- Antiquorum, 267
- Arbiter process, 264
- Asynchronous atomic model, 370
- Asynchronous system, 5
- Atomicity
  - Definition, 430
  - From copy invalidation, 438
  - From copy update, 443
  - From server processes, 437
  - From total order broadcast, 435
  - Is a local property, 433
  - Linearization point, 431

## B

- Bounded delay network
  - Definition, 236
  - Local clock drift, 240
- Breadth-first spanning-tree
  - Built with centralized control, 20
  - Built without centralized control, 17
- Broadcast
  - Definition, 9
  - On a rooted spanning tree, 10

## C

- Causal future, 124

## Causal order

- Bounded lifetime message, 317
  - Broadcast, 313
  - Broadcast causal barrier, 315
  - Causality-based characterization, 305
  - Definition, 304
  - Point-to-point, 306
  - Point-to-point delivery condition, 307
  - Reduce the size of control information, 310
- Causal past, 124
- Causal path, 123
- Causal precedence, 123
- Center of graph, 60
- Channel, 4
  - FIFO, 4
  - FIFO in leader election, 89
  - Four delivery properties, 328
  - State, 132, 368
- Checkpoint and communication pattern, 189
- Checkpointing
  - Classes of algorithms, 198
  - Consistent abstractions, 196
  - Domino effect, 196
  - Recovery algorithm, 214
  - Rollback-dependency trackability, 197
  - Stable storage, 211
  - Uncoordinated, 211
  - Z-cycle-freedom, 196
- Communication
  - Deterministic context, 341
  - Nondeterministic context, 342
- Communication graph, 6
- Concurrency set, 124
- Conflict graph, 286
- Consistency condition, 425
  - Atomicity, 430
  - Causal consistency, 464

- Consistency condition (*cont.*)
  - FIFO consistency, 470
  - Hierarchy, 468
  - Sequential consistency, 447
  - What is the issue, 429
- Continuously passive process
  - In deadlock detection, 416
  - In termination detection, 382
- Convergecast
  - Definition, 9
  - On a rooted spanning tree, 10
- Crown structure, 339
- Crumbling wall, 268
- Cut
  - Consistent, 125
  - Definition, 125
- Cut vertex of graph
  - Definition, 60, 66
  - Determination, 67
- D**
- De Bruijn graph
  - Computing, 74
  - Definition, 73
- Deadlock detection
  - AND model, 403
  - Definition, 404
  - Dependency set, 402
  - One-at-a-time model, 403
  - OR model, 404
  - Resource vs. message, 402
  - Structure of algorithms, 405
  - Wait-for graph, 402
  - What is the difficulty, 414
- Deadlock detection in the AND model, 408
- Deadlock detection in the one-at-a-time model, 405
- Deadlock detection in the OR model, 413
- Degree of a vertex, 42
- Delivery condition
  - Associated with a channel, 331
  - Causal barrier, 315
  - For bounded lifetime messages, 319
  - For message causal order, 307
- Dependency set
  - Deadlock detection, 402
  - Termination detection, 385
- Depth-first network traversal, 24
- Diameter of a graph, 60
- Diffusing computation
  - Definition, 376
  - Termination detection, 377
- Dining philosophers, 290
- Distributed algorithm, 4
  - Asynchronous, 5
  - Synchronous, 4
- Distributed computation: definition, 123
- Distributed cycle detection, 50
- Distributed knot detection, 50
- Distributed shared memory, 427
- Distributed shortest paths
  - Bellman–Ford, 35
  - Floyd–Warshall, 38
- Domino effect, 196
- Drinking philosophers, 295
- Dynamic programming principle, 36
- Dynamic termination
  - Detection algorithm, 394
  - Locally evaluable predicate, 393
- E**
- Eccentricity of a vertex, 60
- Event
  - Definition, 122
  - Immediate predecessor, 170
  - Partial order, 123
  - Relevant event, 170
- F**
- Fast read algorithm (sequential consistency), 453, 456, 459
- Fast write algorithm (sequential consistency), 455
- Finite projective planes, 266
- Flooding algorithm, 10
- Forward/discard principle, 6
- Fully connected network, 4
- G**
- Global checkpoint, 189, *see* Global state
- Global function, 59
- Global state
  - Consistency, 129, 134
  - Consistency wrt. channel states, 133
  - Definition, 129
  - Detection of a conjunction of local predicates, 166
  - In termination detection, 375
  - Including channel states, 132
  - Lattice structure, 129
  - On-the-fly determination, 135
  - Reachability, 129
  - vs. cut, 134
- Global state computation
  - Definition, 136
  - Meaning of the result, 136

- Graph algorithms
  - Distributed cycle detection, 50
  - Distributed knot detection, 50
  - Distributed shortest paths, 35
  - Distributed vertex coloring, 42
  - Maximal independent set, 46
- Graph topology vs. round numbers, 72
- Grid quorum, 267
- H**
- Happened before relation, 123
- I**
- Immediate predecessor
  - Event, 170
  - Tracking problem, 170
- Incremental requests, 287
- Interaction, *see* Rendezvous communication
- Invariance wrt. real time, 125
- K**
- $k$ -out-of- $M$  problem
  - Case  $k = 1$ , 278
  - Definition, 277
  - General case, 280
- $k$ -out-of- $M$  receive statement, 387
- L**
- Lattice of global states, 129
- Leader election
  - Impossibility in anonymous rings, 78
  - In bidirectional rings, 83
  - In unidirectional rings, 79
  - Optimality in unidirectional rings, 86
  - Problem definition, 77
- Linear time
  - Basic algorithm, 150
  - Definition, 150
  - Properties, 151
  - wrt. sequential observation, 152
- Linearizability, *see* Atomicity
- Linearization point, 431
- Liveness property
  - Deadlock detection, 405
  - Mutual exclusion, 248
  - Navigating object, 94
  - Observation of a monotonous computation, 399
  - Rendezvous communication, 337
  - Termination detection, 369
  - Total order broadcast, 321
- Local checkpoint, 189, *see* Local state
  - Forced vs. spontaneous, 198
  - Useless, 195
- Local clock drift, 240
- Local property, 432
  - Sequential consistency is not a, 449
- Local state
  - Definition, 127
- Logical instantaneity, *see* Rendezvous
- Logical ring construction, 27
- Loop invariant and progress condition in
  - termination detection, 383
- M**
- Matrix clock
  - Basic algorithm, 183
  - Definition, 182
  - Properties, 184
- Matrix time, 182
- Maximal independent set, 46
- Message, 4
  - Arrival vs. reception, 385
  - As a point, 336
  - As an interval, 336
  - Bounded lifetime, 317
  - Causal order, 304
  - Crown, 339
  - Delivery condition, 307
  - Filtering, 69
  - In transit, 133
  - Internal vs. external, 7
  - Logging, 211
  - Logical instantaneity, 335
  - Marker, 138
  - Orphan, 133
  - Relation, 123
  - Rendezvous, 336
  - Sense of transfer, 336
  - Stability, 155
  - Unspecified reception, 387
- Message complexity
  - Leader election, 81, 85, 89
- Message delivery: hierarchy, 306
- Message logging, 211
- Mobile object, 93
- Monotonous computation
  - Definition, 398
  - Observation, 399
- Multicast, 9
- Mutex, 247
- Mutual exclusion
  - Adaptive algorithm, 256
  - Based on a token, 94, 249
  - Based on arbiter permissions, 264
  - Based on individual permissions, 249
  - Bounded adaptive algorithm, 259
  - Definition, 247

Mutual exclusion (*cont.*)

- Process states, 248
- To readers/writers, 255
- vs. election, 248
- vs. total order broadcast, 323
- With multiples entries, 278
- wrt. neighbors, 293

**N**

Navigating token for sequential consistency, 459

## Navigation algorithm

- Adaptive, 108
- Based on a complete network, 96
- Based on a spanning tree, 100

## Network

- Bounded delay, 236
- Fully connected, 4
- Object navigation, 93
- Ring, 4
- Tree, 4

## Network traversal

- Breadth-first, 16, 17, 20
- Depth-first, 24
- In deadlock detection, 413
- Synchronous breadth-first, 220

## Nondeterminism, 136

- Communication, 342
- Communication choice, 358
- Nondeterministic receive statement, 385
- Solving, 345

**O**

## Object computation

- Equivalent computations, 430
- Legal computation, 430
- Partial order, 429

## OO constraint, 451

## OR model

- Deadlock detection, 403
- Receive statement, 386

## Owner process, 439

**P**

## Partial order

- On events, 123
- On local states, 127
- On object operations, 429

## Peripheral vertex of graph, 60

## Permission-based algorithms, 249

- Arbiter permission, 249
- Individual permission, 249
- Preemption, 270
- Quorum-based permission, 268

## Port, 9

Preemption (permission), 270

## Process, 3

- Arbiter, 264
- Continuously passive, 382, 416
- History, 122
- Initial knowledge, 5
- Notion of a proxy, 101
- Owner, 439
- Safe, 224
- State, 368

Proxy process, 101

Pulse model, 220

**Q**

## Quorum

- Antiquorum, 267
- Construction, 266
- Crumbling wall, 268
- Definition, 265
- Grid, 267
- Vote-based, 268

**R**

Radius of a graph, 60

## Receive statement

- AND model, 386
- Disjunctive  $k$ -out-of- $m$  model, 387
- $k$ -out-of- $m$  model, 387
- OR model, 386

## Regular graph, 72

- Computing on a De Bruijn graph, 74
- De Bruijn graph, 73

## Relevant event, 170

## Rendezvous communication

- Client-server algorithm, 342
- Crown-based characterization, 339
- Definition, 336
- $n$ -way with deadlines, 360
- Nondeterministic forced interactions, 350
- Nondeterministic planned, 341
- Token-based algorithm, 345
- With deadline, 354
- wrt. causal order, 338

## Resource allocation, 277

- Conflict graph, 286
- Dynamic session, 293
- Graph coloring, 291
- Incremental requests, 287
- Reduce waiting chains, 290
- Resources with several instances, 295
- Static session, 292

Resources with a single instance, 286

Restricted vector clock, 181

- Ring network, 4
  - Rollback-dependency trackability
    - Algorithms, 203
    - BHMR predicate, 208
    - Definition, 197
    - FDAS strategy, 206
  - Rooted spanning tree
    - Breadth-first construction, 16
    - Construction, 12
    - Definition, 11
    - For broadcast, 10
    - For convergecast, 10
  - Round numbers vs. graph topology, 72
  - Round-based algorithm
    - As a distributed iteration, 65
    - Global function computation, 61
    - Maximal independent set, 46
    - Shortest paths, 35
    - Vertex coloring, 43
  - Routing tables
    - From a global function, 62
    - From Bellman–Ford, 35
    - From Floyd–Warshall, 38
  - Rubber band transformation, 143
- S**
- Safety property
    - Deadlock detection, 404
    - Mutual exclusion, 248
    - Navigating object, 94
    - Observation of a monotonous computation, 399
    - Rendezvous communication, 337
    - Termination detection, 369
    - Total order broadcast, 320
  - Scalar time, *see* Linear time
  - Sequential consistency
    - Based on the OO-constraint, 462
    - Definition, 447
    - Fast enqueue, 456
    - Fast read algorithm, 453
    - Fast write algorithm, 455
    - From a navigating token, 459
    - From a server per object, 460
    - From a single server, 456
    - From total order broadcast, 453
    - Is not a local property, 449
    - Object managers must cooperate, 461
    - OO constraint, 451
    - Partial order for read/write objects, 450
    - Two theorems, 451
    - WW constraint, 451
  - Sequential observation
    - Definition, 131
  - Snapshot, 121
  - Space/time diagram
    - Equivalent executions, 125
    - Synchronous vs. asynchronous, 5
  - Spanner, 234
  - Stable property
    - Computation, 137
    - Definition, 137
  - Static termination
    - Detection algorithm, 390
    - Locally evaluable predicate, 390
  - Superimposition, 141
  - Synchronizer
    - Basic principle, 223
    - Definition, 222
    - Notion of a safe process, 224
  - Synchronizer  $\alpha$ , 224
  - Synchronizer  $\beta$  (tree-based), 227
  - Synchronizer  $\delta$  (spanner-based), 234
  - Synchronizer  $\gamma$  (overlay-based), 230
  - Synchronizer  $\mu$ , 239
  - Synchronizer  $\lambda$ , 238
  - Synchronous breadth-first traversal, 220
  - Synchronous communication, *see* Rendezvous communication
  - Synchronous system, 4, 219
    - For rendezvous with deadlines, 354
    - Pulse model, 220
- T**
- Termination
    - Graph computation algorithm, 8
    - Local vs. global, 23
    - Predicate, 368
    - Shortest path algorithm, 37
  - Termination detection
    - Atomic model, 370
    - Dynamic termination, 389
    - Four-counter algorithm, 371
    - General model, 378
    - General predicate *fulfilled()*, 387
    - Global states and cuts, 375
    - In diffusing computation, 376
    - Loop invariant and progress condition, 383
    - Problem, 369
    - Reasoned detection in a general model, 381
    - Static termination, 389
    - Static vs. dynamic termination, 388
    - Static/dynamic vs. classical termination, 389
    - Type of algorithms, 369
    - Vector counting algorithm, 373
    - Very general model, 385

## Timestamp

- Definition, 152
- wrt. sequential observation, 152

## Token, 94

## Total order broadcast

- Based on inquiries, 324
- Circulating token, 322
- Coordinator-based, 322
- Definition, 154, 320
- For sequential consistency, 453
- In a synchronous system, 326
- Informal definition, 153
- Strong, 320
- Timestamp-based implementation, 156
- To implement atomicity, 435
- vs. mutual exclusion, 323
- Weak, 321

## Tree invariant, 101

## Tree network, 4

## U

## Uncoordinated checkpointing, 211

## Unspecified message reception, 387

## V

## Vector clock

- Adaptive communication layer, 180
- Approximation, 181
- Basic algorithm, 160
- Definition, 160
- Efficient implementation, 176
- $k$ -restricted, 181
- Lower bound on the size, 174
- Properties, 162

## Vector time

- Definition, 159
- Detection of a conjunction of local predicates, 166
- Development, 163
- wrt. global states, 165

## Vertex coloring, 42

## Vote, 268

## W

## Wait-for graph, 402

## Waiting

- Due to messages, 401
- Due to resources, 401

## Wave and sequence of waves

- Definition, 379
- Ring-based implementation, 380
- Tree-based implementation, 380

## Wave-based algorithm

- Spanning tree construction, 17
- Termination detection, 381, 390, 394

## WW constraint, 451

## Z

## Z-cycle-freedom

- Algorithms, 201
- Dating system, 199
- Definition, 196
- Notion of an optimal algorithm, 203
- Operational characterization, 199

## Z-dependency relation, 190

## Zigzag path, 190

## Zigzag pattern, 191