Lecture Notes in Networks and Systems 27

Ludwik Czaja

Introduction to Distributed Computer Systems

Principles and Features

Springer

Lecture Notes in Networks and Systems

Volume 27

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

The series "Lecture Notes in Networks and Systems" publishes the latest developments in Networks and Systems—quickly, informally and with high quality. Original research reported in proceedings and post-proceedings represents the core of LNNS.

Volumes published in LNNS embrace all aspects and subfields of, as well as new challenges in, Networks and Systems.

The series contains proceedings and edited volumes in systems and networks, spanning the areas of Cyber-Physical Systems, Autonomous Systems, Sensor Networks, Control Systems, Energy Systems, Automotive Systems, Biological Systems, Vehicular Networking and Connected Vehicles, Aerospace Systems, Automation, Manufacturing, Smart Grids, Nonlinear Systems, Power Systems, Robotics, Social Systems, Economic Systems and other. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution and exposure which enable both a wide and rapid dissemination of research output.

The series covers the theory, applications, and perspectives on the state of the art and future developments relevant to systems and networks, decision making, control, complex processes and related areas, as embedded in the fields of interdisciplinary and applied sciences, engineering, computer science, physics, economics, social, and life sciences, as well as the paradigms and methodologies behind them.

Advisory Board

More information about this series at http://www.springer.com/series/15179

Ludwik Czaja

Introduction to Distributed Computer Systems

Principles and Features

Springer

Ludwik Czaja
Vistula University
Warsaw
Poland

and

Institute of Informatics
University of Warsaw
Warsaw
Poland

ISSN 2367-3370 ISSN 2367-3389 (electronic)
Lecture Notes in Networks and Systems
ISBN 978-3-319-72022-7 ISBN 978-3-319-72023-4 (eBook)
https://doi.org/10.1007/978-3-319-72023-4

Library of Congress Control Number: 2017959902

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Nina and Mateusz

Acknowledgement

I am indebted to my colleagues for encouragement to collect notes of my 12 lectures into a book form, but my special thanks go to Prof. Andrzej Skowron and Prof. Janusz Kacprzyk for initiation and promoting idea of publication of the book in Springer-Verlag. Also, Mr. Viju Falgon, a project manager at Scientific Publishing Services of Springer who handled the editorial production of the book, deserves special thanks, being bothered with numerous corrections of my errors and misprints I noticed during the production process.

Ludwik Czaja

Contents

1 Instruction Execution Cycle and Cooperation of Processes 1
1.1 Instruction Execution Cycle of Sequential Processor 1
1.2 Concurrent Execution of Programs in the Language of Machine Instructions 4
1.3 Mutual Exclusion of Processes; Application of Semaphores 8
1.4 Synchronous Communication of Processes 19
1.5 Asynchronous Communication of Processes 37
1.6 Synchronous Vector Systems 43
1.7 Some Classifications of Computer Systems 47
References 48

2 Distributed Systems—Objectives, Features, Applications 49
2.1 What Systems Do We Consider as Distributed? 49
2.2 Most Important Objectives of Distributed Systems 53
2.2.1 Economy 53
2.2.2 Increase of Computing Power 53
2.2.3 Internal Distribution 54
2.2.4 Reliability 55
2.2.5 Independence from Changes of Environment 55
2.2.6 Flexibility 56
2.3 Main Features of Distributed Systems 56
2.3.1 Resource Sharing 57
2.3.2 Openness 59
2.3.3 Transparency 60
2.3.4 Scalability 61
2.4 Exemplary Memory Connection Structures in Centralized Multiprocessors 61
Reference 64

3 Concurrency . . . 65
3.1 Concurrent Execution of Programs . . . 65
3.2 Deadlock . . . 73
3.3 Starvation . . . 75
3.4 Mutual Exclusion by Supervisory Server . . . 78
3.5 Mutual Exclusion—Token-Ring Algorithm . . . 79
References . . . 84

4 Time, Coordination, Mutual Exclusion Without Supervisory Manager . . . 85
4.1 Physical Time . . . 85
4.1.1 The Cristian Method of Clock Synchronization (Cristian 1989) . . . 88
4.1.2 The Berkeley Method of Clock Synchronization (Gusella 1989) . . . 90
4.1.3 The Network Time Protocol (NTP) Method of Synchronization of Clocks (Mills 1991) . . . 94
4.2 Logical Time: Precedence of Events, Time Compensation, Timestamps, Logical Clock . . . 94
4.3 Distributed Mutual Exclusion Without External Service for Processes—A Method Based on Global Timestamps (Ricart 1981) . . . 101
4.4 Distributed Mutual Exclusion Without External Service for Processes—A Method Based on Vectors of Global Timestamps (Czaja 2012) . . . 102
References . . . 117

5 Interprocess Communication . . . 119
5.1 Basic Problems of Communication . . . 119
5.2 Tasks of Communication Protocols—Examples . . . 121
5.3 Dispatch and Reception . . . 124
5.4 Modes of Communication: Synchronous and Asynchronous, Connection-Oriented and Connectionless, Multicast and Broadcast, Group Communication . . . 127
5.5 Layered Structure of the Set of Communication Protocols: OSI/RM and ATM . . . 133
References . . . 139

6 Remote Procedure Call . . . 141
6.1 Motivations, Problems, Limitations . . . 141
6.1.1 Different Environments of the Client and Server . . . 142
6.1.2 Conflicts When Using Shared Resources . . . 143
6.1.3 The Stub . . . 144
6.1.4 The Binder—Finding a Server . . . 145
6.1.5 Exceptions . . . 146

6.1.6 Lost and Repeated Messages 146
6.2 Example of RPC Mechanism Activity 147
References 155

7 Failures and Damages in Distributed Systems 157
7.1 Chances and Kinds of Failure, Remedial Measures, Fault Tolerance 157
7.1.1 Probability of System's Defective Activity; Expected Time up to a Breakdown 158
7.1.2 Kinds of Failure and Some Mechanisms of the Fault-Tolerant Systems 159
7.2 Some Problems of Activity Coordination 162
7.2.1 Infinite Cycle of Confirmations—the "Two Army" Problem 162
7.2.2 Reaching Consensus with Fault Tolerance—the "Byzantine Generals" Problem 165
7.3 Election of a New Coordinator Following Detection of the Damaged 173
7.3.1 Bully Algorithm of Election 173
7.3.2 Ring Algorithm of Election 177
References 185

8 Distributed Shared Memory 187
8.1 Structure, Motivations, Problems, Advantages, Disadvantages . . . 187
8.2 Interleaving Model of System Activity 190
8.3 Concurrency of Access Operations to Distributed Shared Memory, Examples of Sequential and Strict Consistency, Informal Description 196
8.4 Events of Initiations and Terminations of Read/Write Operations 210
8.5 Formal Definitions of Sequential and Strict Consistency 215
8.6 Some Other Models of Memory Consistency 217
8.6.1 Causal Consistency (Hutto and Ahamad 1990) 217
8.6.2 PRAM (Pipelined Random Access Memory) Consistency (Lipton and Sandberg 1988) 219
8.7 Exemplary Algorithms Realizing Memory Consistency 221
8.7.1 Algorithms for Sequential Consistency 222
8.7.2 An Algorithm Implementing Causal Consistency for Computer of Number i 223
8.7.3 An Algorithm Implementing PRAM Consistency for Computer of Number i 225
References 225

9 The Control Flow During Execution of the Alternating Bit Protocol Specified by the Cause-Effect Structure 227
References 239

10 Some Mathematical Notions Used in the Previous Chapters 241
10.1 Binary Relations 241
10.2 Infinite Series of Real Numbers 242
10.2.1 D'Alambert (Sufficient) Criterion of Convergence 242
10.2.2 Mertens Theorem on Multiplication of Series 242
10.3 Probability, Independent Events, Random Variable and Its Expected Value 244
10.3.1 Basic Notions of Cause-Effect Structures 246
References 249

Final Remarks 251

Bibliography 255

Index 257

About the Author

Ludwik Czaja obtained his M.Sc., Ph.D., and habilitation degrees from the University of Warsaw, where he was a professor of informatics at the Faculty of Mathematics, Informatics, and Mechanics. He spent some years in other universities, like Carnegie-Mellon, Oxford, Ben-Gurion, Humboldt, as visiting professor or a research fellow. Now he is a full professor of the Vistula University and professor emeritus of the University of Warsaw. His work encompasses formal and programming languages, compilers, theory of computation, parallel, and distributed processing.

Introduction

In the most general meaning, a distributed computer system is identified with computer network. There are two problems with this identification. One concerns the word "computer". Even the origins of such systems encompassed not only computers but also devices far from being computers, like missile launchers and other military objects (a brief historical outline is in Chap. 2). Let alone today's networks that connect things of professional and everyday usage: one can hardly say that the so-called intelligent refrigerator is a computer. The other problem concerns the word "distributed". The main feature of such architectures is lack of shared physical memory, where processors intercommunicate by message passing through data links (channels) of arbitrary length and bandwidth. So, in this meaning, a motherboard of transputers (Sect. 5.3 in Chap. 5), processors interacting by data packet exchange inside one machine, all are distributed systems too. But on the other hand, a computer surrounded by simple terminals for data input and output, multi-access computer or workstations with direct access to memory of a mainframe, are systems categorized as not distributed in the aforesaid meaning, though distributed in the common parlance, because separated spatially. Nonetheless, the distinguishing characteristic of distributed computer systems as the research and engineering domain, is inter-computer communication by message passing based on networks. However, the design, technical solutions, and application of distributed systems and general ("generic") networks, are not identical. The conceptual difference lies in their destination. A distributed system is being often constructed as a computation environment for specific class of applications, for a company of certain activity profile, for a corporation or education center, whereas a network is a universal tool for data transmission not limited to particular applications. Thus, the constructional difference consists in software: in (distributed) operating system and programs on top of it—compilers of programming languages and diverse applications. But dividing functions of a system into hardware and software is not constitutive from the information processing point of view. For distributed systems, hardware and software of a network constitute a communication infrastructure. So, presentation of their principles takes into account some issues of network technology, because design and implementation of

distributed systems exploits solutions elaborated there. The principles concern, for instance, physical and logical time, coordination of processes (synchronization in particular), communication, exception handling, fault tolerance, and security of system's behavior. More specific are distributed transactions, scheduling joint actions of computers, remote procedure call and simulation of distributed shared memory, giving the user impression of having direct access to huge memory space. Presentation of these issues, limited to principles, in order not to overshadow them with technical details, is contained in successive chapters. The details change as the research and technology develops, the principles remain longer. Therefore, great many dedicated distributed systems and several of general usage, are not discussed here. Their presentation would exceed capacity of any book, a few of such projects are mentioned only in Chap. 2. Emphasis in this book is on objectives of distributed systems. The objectives determine properties. As the most important is regarded *transparency*, that is, hiding the joint usage of resources by multiple users—making every user imagine to be exclusive one. He/she is to be unaware how and where the resources and mechanisms of their usage are located: in hardware, operating system kernels, application libraries, compilers, or runtime systems of programming languages. From the principles point of view, this is immaterial. In existent distributed systems, the transparency has been accomplished only to a certain degree, as the problem of balance of user's convenience against system's efficiency becomes of prime importance. However, along with the development of communication infrastructure and increase of large data sets' transmission rate, attainment of full transparency seems realistic. To keep up with the progress in materializing such ideas, a look through professional writings is indispensable. Their short selection, quoted in the following chapters, is in the Bibliography.

A presentation of the wide subject matter in a small book requires making decision regarding its content. Apart from some issues belonging to the mainstream of network technology and distributed systems outlined here, such topics like construction of distributed operating systems (including resource naming and mapping names onto addresses) and security of communication (cryptography in particular), are beyond this book. They are broad and individual research and technology branches, enjoying rich and easily available literature. Chapter 9, however, contains a fairly well-known pattern-protocol "in action" (expressed by a cause/effect structure diagram; the calculus of cause/effect structures is outlined in Chap. 10) as a sequence of states which the protocol passes during execution. This is the so-called *alternating bit protocol*, which reveals basic features of a certain category of protocols (its enhanced versions like "*sliding window*", "*abracadabra*", became a basis for some existent protocols design).

The book is inspired by a 30-h lecture course on distributed computer systems, given on the elementary level by the author. It is a handbook, although containing some monographic elements not published elsewhere, like a protocol described in Sect. 4.4, formal proofs of some important facts and a method of presentation of algorithms in action: as sequences of states. Experience shows that students, especially not professional programmers, easier assimilate essence of algorithmic constructs presented in action rather than in the form of static descriptions as

programs. Hence, the method of presenting: a short explanation of a construct is followed by a sequence of states it passes. If these states are regarded as animated film frames, then projection of the film illustrates process of execution of the construct. In accordance with such metaphor, description of a construct (a program) is a screenplay, while the process—realization of this screenplay on the film tape. Furthermore, using computer for the animated show of the process gives rise to see it as a live scenery in a slow motion.

Understanding of the following chapters does not require advanced knowledge in informatics. Sufficient will be familiarity with topics provided in an introductory course to computer science accompanied by laboratory exercises, and to algorithms and data structures. The participants acquire there a certain skill in design and analysis of algorithms, to write them as programs and run on the computer. Beneficial will also be some knowledge in foundations of operating systems and computer networks, where pretty much time is devoted to standard communication protocols, being omitted here as technical realization of principles. Of great importance is experience in a laboratory teamwork—a practical contact with a kind of distributed system. However, a certain problem may be noticed. Despite having acquired these foundations, the students of computer science or engineering, who begin programming in a high-level language, often exhibit ignorance in knowledge of structure and activity of a processor on the level of internal machine language, what is the instruction execution cycle in particular. Yet, many functions of distributed system are found in ordinary, stand-alone machine, especially with multiprogram operating system. For instance, communication mechanisms in networks play a part similar to input/output channels (serviced by protocols too), management of concurrent transactions requires actions similar to synchronizing processes in a sequential multiprogrammed machine, etc. That is why, the first chapter contains an outline of sequential machines activity with internal control, where many mechanisms have counterparts, though significantly more sophisticated, in network systems. The activity of a machine model (also multiprogrammed) with a short set of typical instructions, as well as a collection of such machines interconnected by communication channels, has been demonstrated by means of abovementioned "film method". This visual demonstration should clearly explain principle of sequential processor's activity. In Chap. 1, the reader will also find example of a synchronous vector machine "in action". The presented material is, as usually based on the principle of von Neumann's stored program sequential machines. A few sentences about other principles of information processing, though not yet materialized in architectures for the public use, are in the final remarks of the book.

The question of mathematical description of phenomena appearing in distributed systems remains. Excessive formalization has been avoided, yet clear presentation of some issues demanded a formal notation, especially in a few proofs of properties. To such issues belong, for instance, time (physical and logical), correctness of some algorithms of distributed mutual exclusion, chances of system failure, unattainable simultaneity of certain actions, definition of distributed memory consistency models and its consequences. Therefore, to express some concepts, facts and proofs, mathematical language has been used, though in a very modest extent: operations

on sets and relations, partial and linear order, functions, sequences and series, probability, elementary propositional and predicate calculus. Some of them are outlined in Chap. 10. Such topics are familiar to everybody who attended introductory course at mathematical or engineering studies.

Chapter 1
Instruction Execution Cycle and Cooperation of Processes

1.1 Instruction Execution Cycle of Sequential Processor

This chapter contains an outline of structure and functioning of stand-alone internally controlled computer with sequential processor, as well as its functioning in a collection of such machines. The internal control means that program instructions (commands) and data, encoded (commonly nowadays) as sequences of bits, are located in the internal memory (RAM) and are fetched by the processor to its specific register. A distinction between instructions and data depends on this register. If it is the instruction register (IR), the bit sequence is interpreted as instruction and taken to suitable electronic circuits where it is executed, if another register, e.g. accumulator (A), it is interpreted as data. This is the so-called stored-program architecture, devised by von Neumann (1945) with J. W. Mauchly and J. P. Eckert (in the 1940s), commonly known as the von Neumann's architecture. The details of a single instruction execution, that is actions of the electronic circuits, are on the lower description level than principles of distributed systems, so will not be considered. Why this book begins with such architecture (a brief note on alternative concepts of data processing is in the Final Remarks)? The following are some reasons:

- Multiprocessor and multicomputer distributed system is a set of such machines:
  - multiprocessor—a set of processors with common physical memory,
  - multicomputer—a set of stand-alone (autonomous) computers without common physical memory, possibly with diverse architectural details (e.g. different coding of numbers); the computers are connected by means of transmission channels and endowed with distributed operating system.
- The objective of the multiprocessor and multicomputer system is to give the users impression of work on a sequential, autonomous machine of enhanced performance of algorithms (due to concurrent processing of their parts) and endowed with mechanisms for joint resource usage, for interprocess

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_1

communication, for correct cooperation (in particular synchronizing some actions) between other machines, etc.

- Basic functions of distributed system are similar to those in stand-alone sequential machine (Fig. 1.1); for instance, communication mechanisms in a computer network plays similar part as input/output mechanisms in a sequential machine, but are significant extension of the latter. This extension creates many problems, absent in sequential machines, where communication proceeds only between its own internal and external components.
- The book begins with demonstrative execution of simple programs by a model of sequential von Neumann's machine and their parallel execution by a collection of such machines, connected by communication channels. Such presentation visualizes main problems of synchronization and communication in the real distributed systems.

Instruction execution cycle depicted in Fig. 1.2 is a hardware-implemented algorithm, that is, an automaton, which executes user programs being data for this algorithm. For this reason, the instruction execution cycle may be called—a „meta-program". It is defined by a structure of processor, that is, by its active components (like transistors) and their connections. The structure is changeless in processors not being microprogrammed. It can be modified in the microprogrammed processors, where useful (for some applications) instructions may be created, that

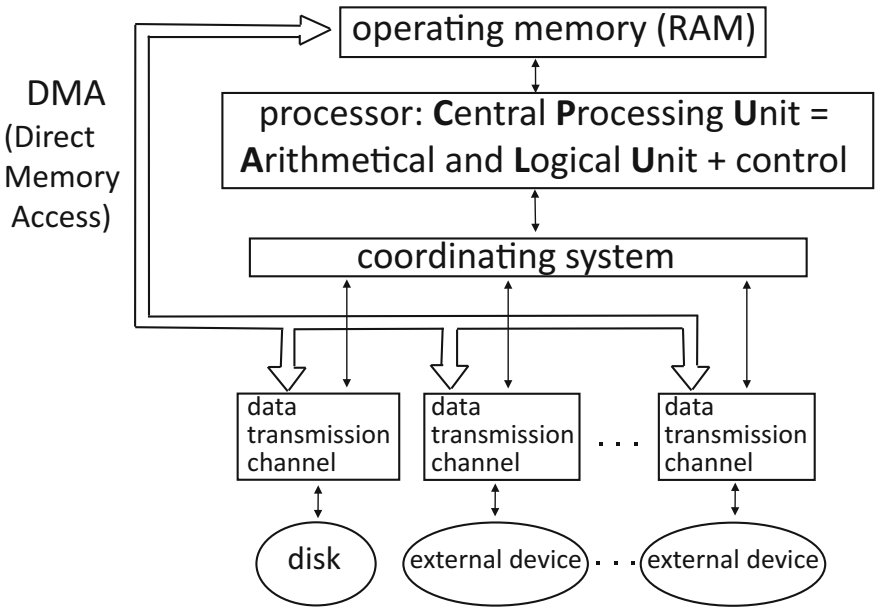


Fig. 1.1 Diagram of main components of a single-processor stand-alone computer; the memory arbiter, ensuring exclusive access to single memory cell by processor and various external devices, is usually integrated with DMA controller

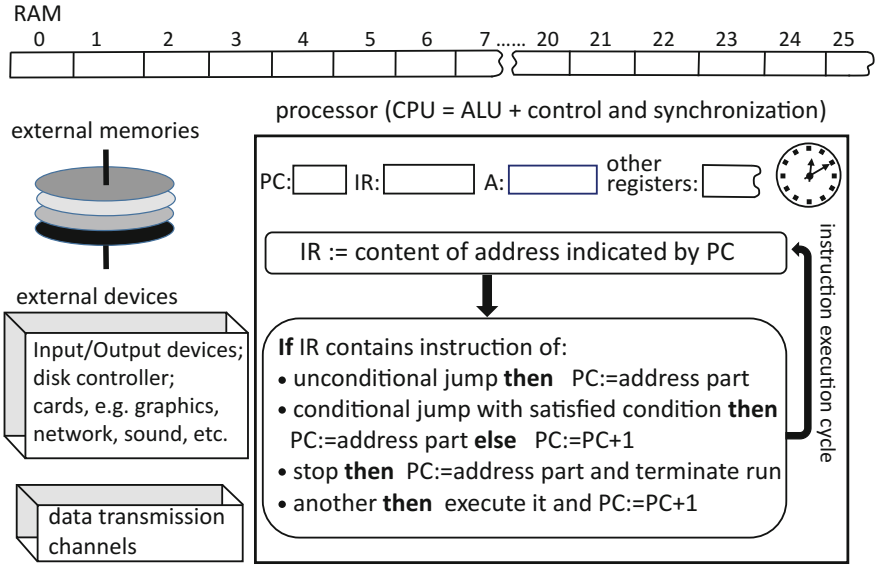


Fig. 1.2 Functional schema of internally controlled computer with instruction execution cycle; PC—Program counter, IR—Instruction register, A—Accumulator

means a modification of the processor's instruction set. To illustrate by examples, action of the schematic computer depicted in Fig. 1.2, let us fix the set of a few instructions, typical in their function for many commercial processors, though—in their form—freely adopted here for this purpose. Their names of operation part are shortcuts of their meaning description. For simplicity, they are confined to one-argument. Their set is in Fig. 1.3. Obviously, commercial processors are equipped with much broader instruction sets, of more complex functions as, for instance, fixed-point and floating-point arithmetic instructions and many more. The clock, which organizes execution of the instruction cycle, in some architectures is the processor's component, whereas in some others is outside the processor. This detail is immaterial for further exemplary presentation of instruction execution cycle.

Animated (when displayed on screen as consecutive transparencies) execution of assignment statement $x := y * z + u / v$ on schematic computer from Fig. 1.2 is shown in Table 1.1. This is a sequence of states of this computer, numbered 1–17, during execution of machine instructions stored in the memory cells of addresses 0–7 and variables *x, y, z, u, v, temp*, allocated in some further memory cells. This sequence is a process of computing the value of expression $y * z + u / v$ for $y = 3.14$, $z = 2.0$, $u = 15.9$, $v = 3.0$ and assigning this value to x. The small horizontal arrow points to current (in a given state) action of the instruction execution cycle. Considering states as an animated film's frames, projection of the film presents dynamics of execution of the assignment statement in machine instructions, that is a process of computation. Admitting this metaphor, one may say that the program is a screenplay, whereas its realization is the process, recorded on the film tape.

ope-ration	add-ress	meaning of operation
LA	*n*	**L**oad **A**ccumulator with the content of a cell *n* and go to the next instruction in program
SA	*n*	**S**tore the content of **A**ccumulator in a cell *n* and go to the next instruction in program
AD	*n*	**AD**d to the content of accumulator the content of a cell *n* and go to the next instruction in program
SU	*n*	**SU**btract from the content of accumulator the content of a cell *n* and go to the next instruction in program
MU	*n*	**MU**ltiply the content of accumulator by the content of a cell *n* and go to the next instruction in program
DI	*n*	**DI**vide the content of accumulator by the content of a cell *n* and go to the next instruction in program; if the content of *n* is 0 then report an error
JU	*n*	**JU**mp to the instruction stored in a cell *n* in program (unconditional jump)
JZ	*n*	**J**ump to the instruction stored in a cell *n* in program if accumulator contains **Z**ero; otherwise go to the next instruction in program (conditional jump)
JN	*n*	**J**ump to the instruction stored in a cell *n* in program if accumulator contains **N**egative value; otherwise go to the next instruction in program (conditional jump)
JΩ	*n*	**J**ump to the instruction stored in a cell *n* in program if the communication register (*chan*) contains **Ω**; otherwise go to the next instruction in program (conditional jump)
ST	*n*	go to the instruction stored in a cell *n* in program and **ST**op activity of program

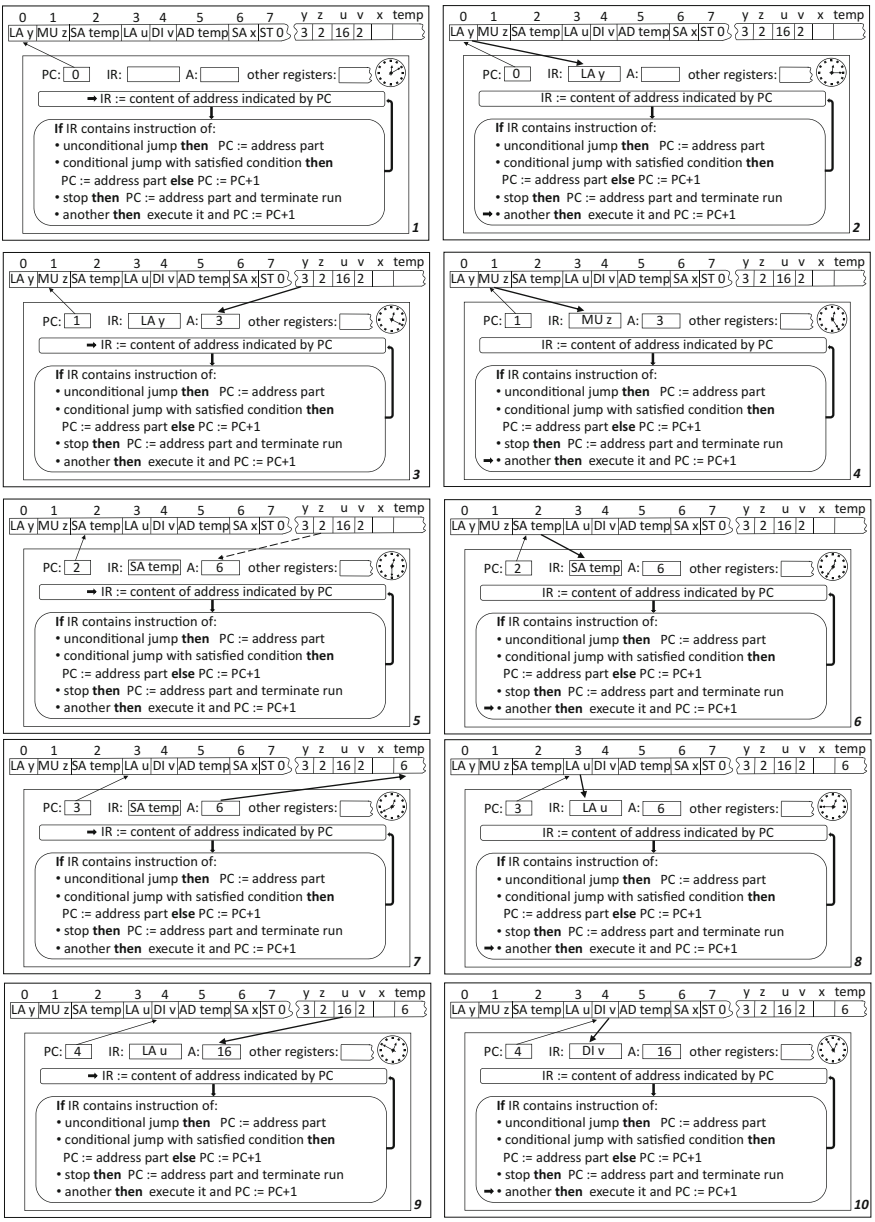
Fig. 1.3 Simple instruction set

Although the distributed systems, as understood here, are multicomputer decentralized systems, they may be implemented by a single computer, in the so-called multiprogramming mode, realized by the time-sharing of one processor. This is an *interleaving* of chunks of concurrent programs, the concept extensively exploited in Chap. 8 for the real multicomputer systems. Such simulation of multicomputers is illustrated by Table 1.2, where a single computer, when performing programs P1, P2, P3, P4, passes states 1–17. The computer is endowed with instructions of interrupt (suspension) and resumption of processes and a queue of processes suspended and may be in the following states: "before run", "run", "suspended", "terminated". The memory of this computer is shared between the four programs.

1.2 Concurrent Execution of Programs in the Language of Machine Instructions

If a number of processes are running simultaneously, either in a single processor system in the time-sharing mode or by a real multiprocessor system, some conflicts between the processes utilizing shared resources may arise. Such case is illustrated by Table 1.3—execution of erroneously programmed assignment statement $x := y * z + u / v$ by the two-computer system with a shared memory for data. The computers intercommunicate by a shared variable *chan* (pretending a communication channel). Computer 1 sends value $y * z$ to *chan*, whereas computer 2—value u / v. The conflict arises if both computers are sending these values to variable *chan* at the same moment or very close to each other moments. This case occurs in the

Table 1.1 Process of execution of assignment statement $x := y * z + u / v$. States 1–17

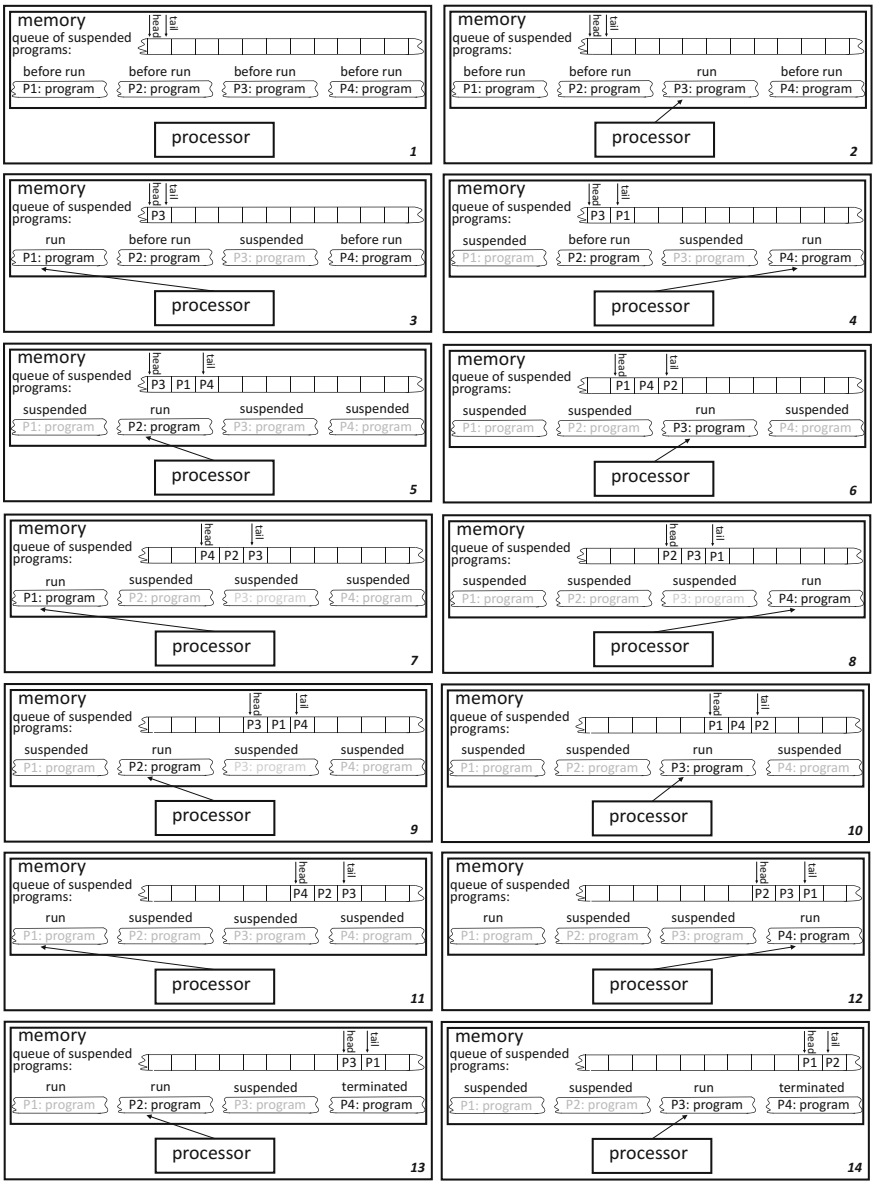


(continued)

Table 1.1 (continued)

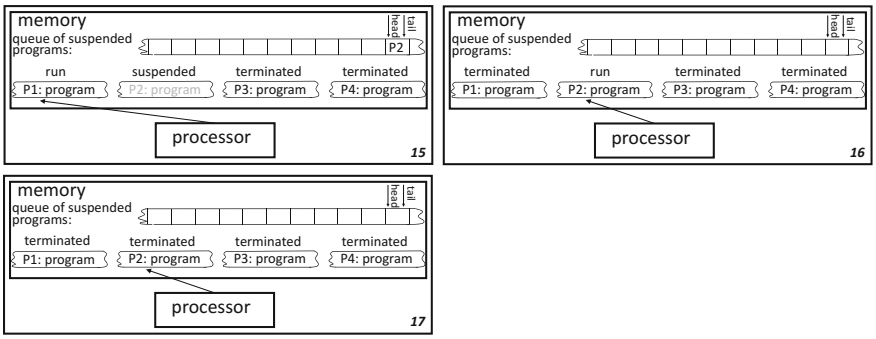
state 9 of the execution. It is not known which value will appear in the „channel" *chan*: undesirable non-determinism occurs. Notice that if the two values are sent to variable *chan* at sufficiently distant moments, the result will be correct, however the programmer may not programm this way, not knowing the temporal relationships of processes. Usually, the programmer does not know the pace of processes, thus has to programm so that result of the program execution is independent of their speed. This is the case of ordinary paradigms, languages and techniques of programming.

Table 1.2 Activity of a single multiprogrammed computer executing 4 programs in the timesharing mode. States 1–17



(continued)

Table 1.2 (continued)



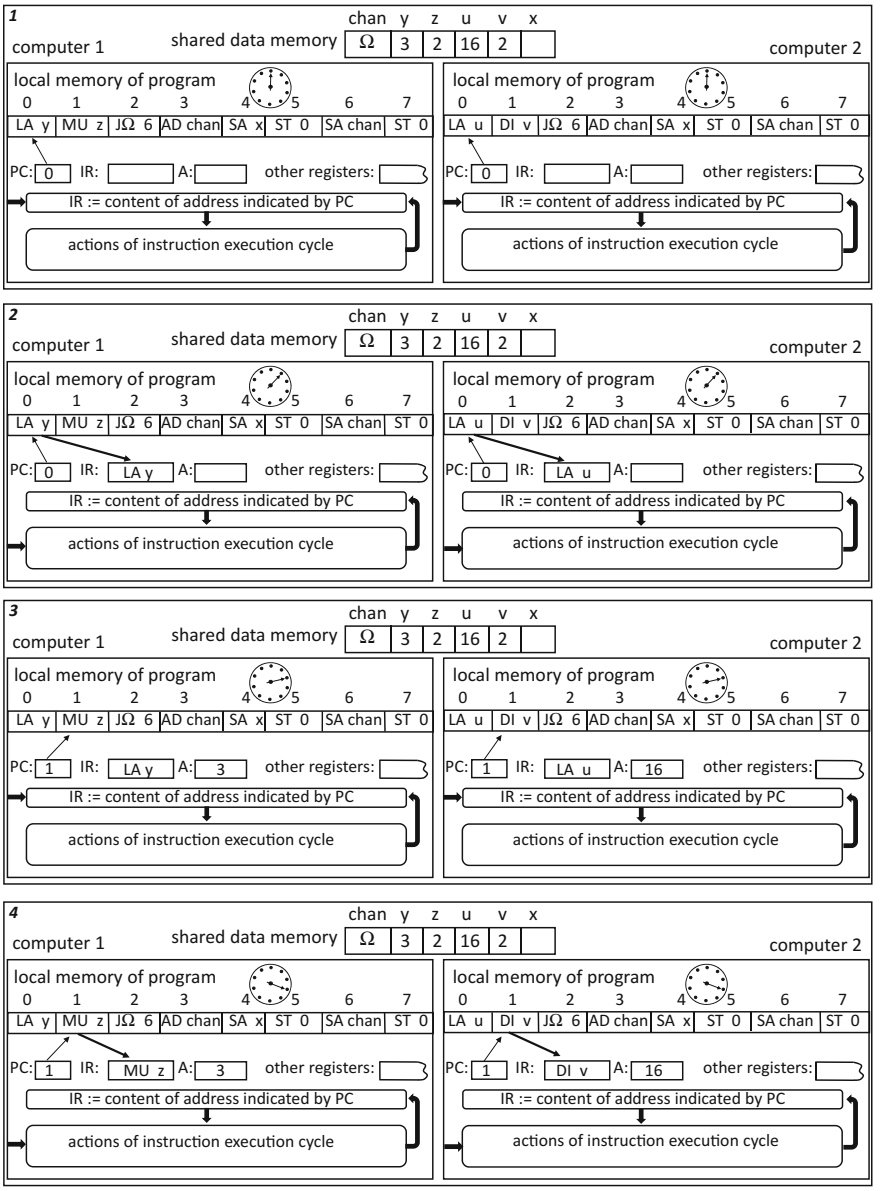
1.3 Mutual Exclusion of Processes; Application of Semaphores

In order to avoid conflicts between processes that use shared resources, some mechanisms are applied ensuring access to a resource for at most one process at a time and, after its usage, giving access to this resource for another process. Such mechanisms implement mutual exclusion of processes, that is determine the so-called critical section, being a program fragment. In exclusive execution of this fragment, a protected resource is being used. Let us consider the semaphores as such mechanism, leaving out others, like TEST&SET, as well as algorithms of mutual exclusion, for instance the Dekker's algorithm (published in Dijkstra 1965), probably the first correct solution to this problem. The semaphore, introduced by the Dutch scientist Dijkstra (1965, 1968), is a variable, that assumes integer values and is shared by cooperating programs. For further considerations, the semaphores will be limited to binary ones, that is, they will assume only values 0 and 1. Permissible operations on semaphores are named P and V, and are of the following meaning, where *sem* is a semaphore:

- P(*sem*): if $sem > 0$ then $sem := sem - 1$, and if $sem = 0$ then suspend the process
- V(*sem*): $sem := 1$ and resume a certain suspended process (for instance, suspended by the longest time)

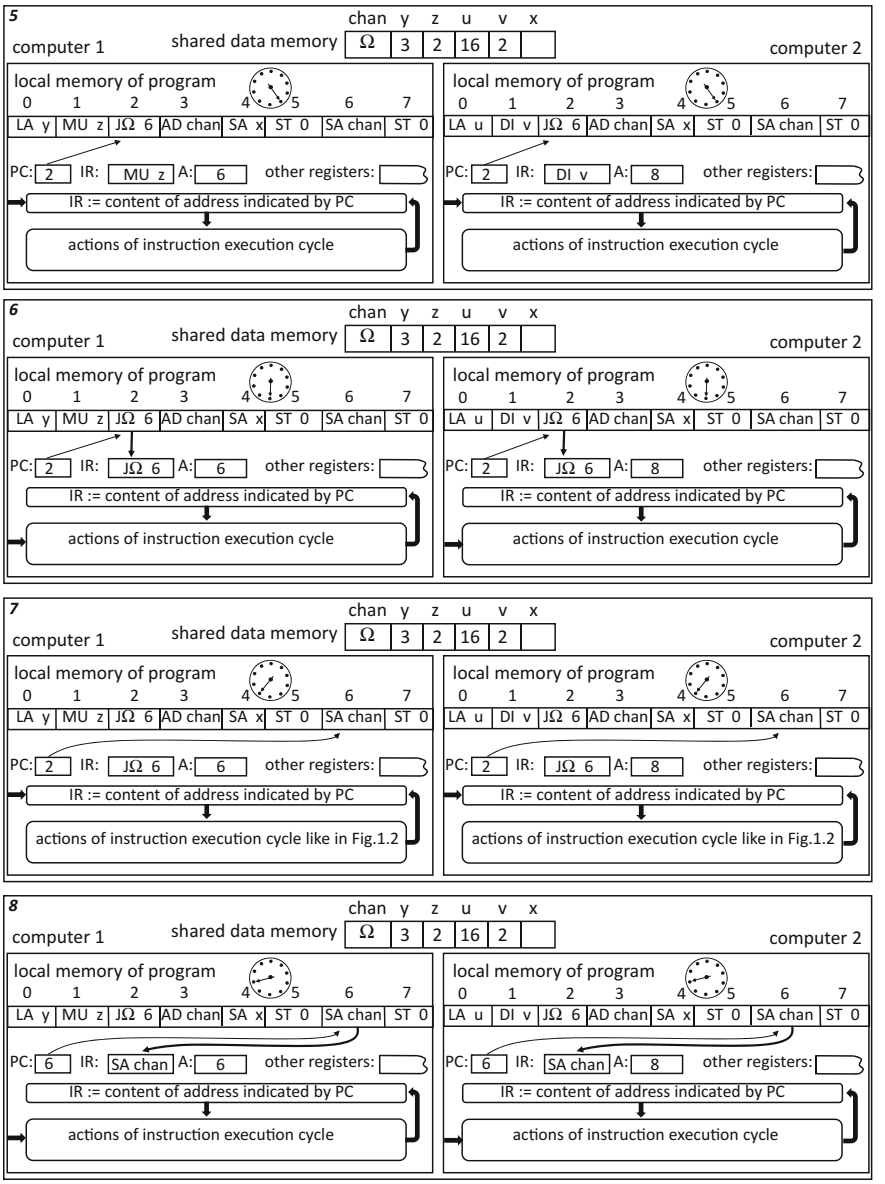
The operations are indivisible (atomic): when a certain program performs one of them, another program cannot perform none of them at the same time. Thus, they are critical sections themselves, but performed at the lower level than the user's programs (for instance by computer hardware or in the kernel of operating system). In some programming languages, the P operation is named „*wait*" and V—„*signal*" (P and V are the first letters of Dutch words "passeren" and "vrijmaken", meaning

Table 1.3 Beginning of execution of incorrectly programmed assignment statement $x := y * z + u / v$ by 2 computers with a shared memory for data. States 1–11



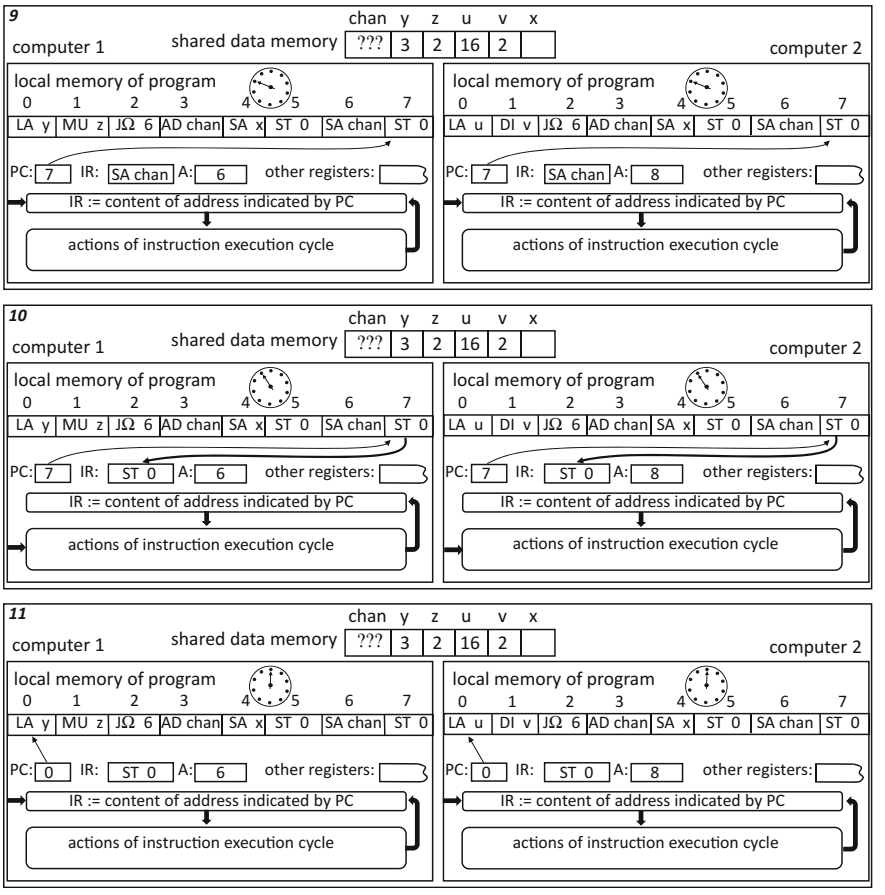
(continued)

Table 1.3 (continued)



(continued)

Table 1.3 (continued)



Undetermined value is stored in chan

"to pass" and "release"). They will be treated as computer's instructions. The operations encompass arbitrarily long piece of program where exclusive access to a resource is performed, in contrast to a hardware device, called a memory arbiter, which assures exclusive read/write of a single memory cell only. Notice that the name „semaphore" is taken from the railway terminology, because the semaphore closes entrance of program (train) to its critical section (occupied track) if another program is executing critical section which protects the same resource. This is illustrated in Table 1.4 (with somewhat outdated steam locomotive).

Table 1.4 The semaphore precludes to stay both vehicles under the water pump, at the same time

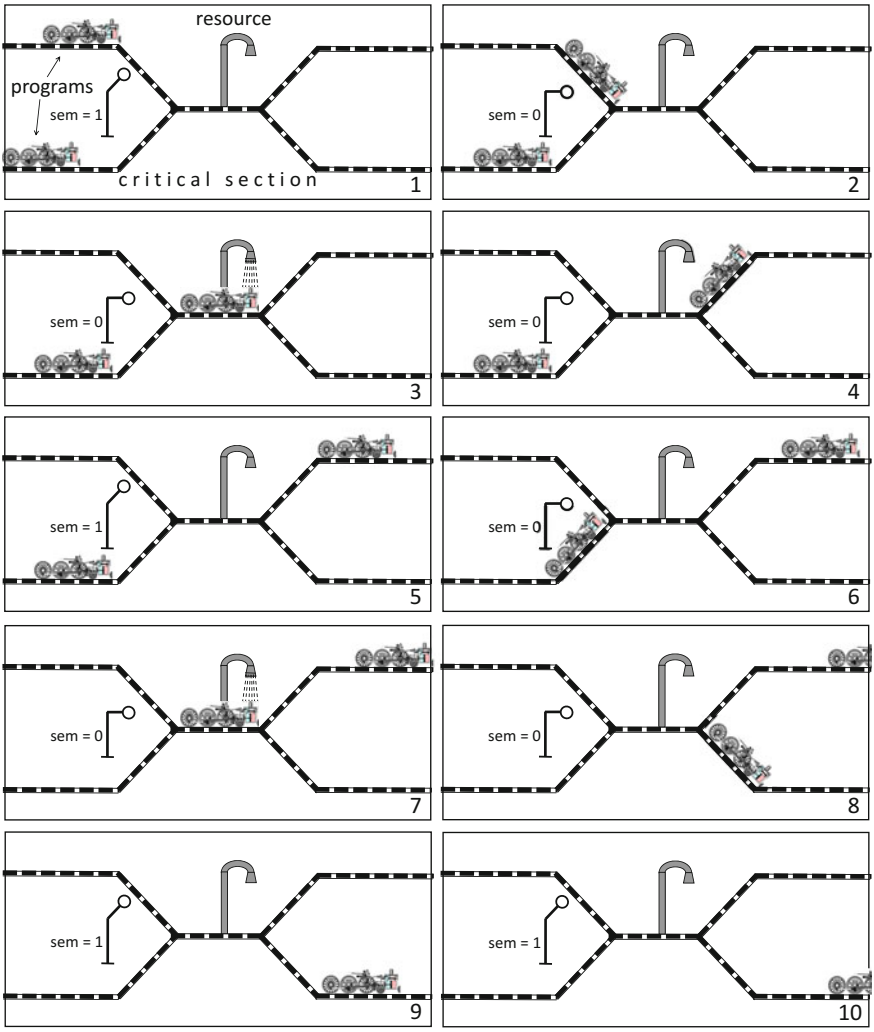
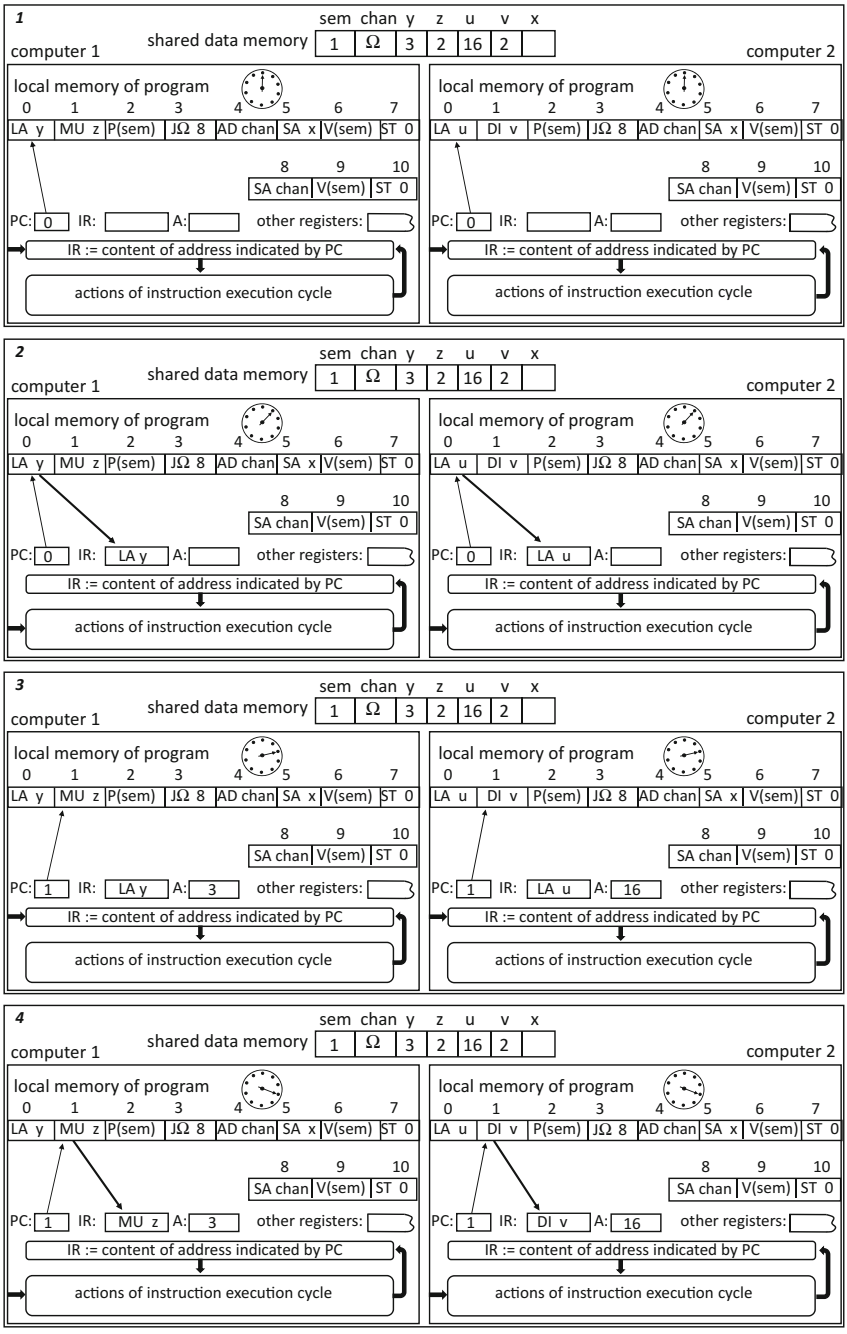


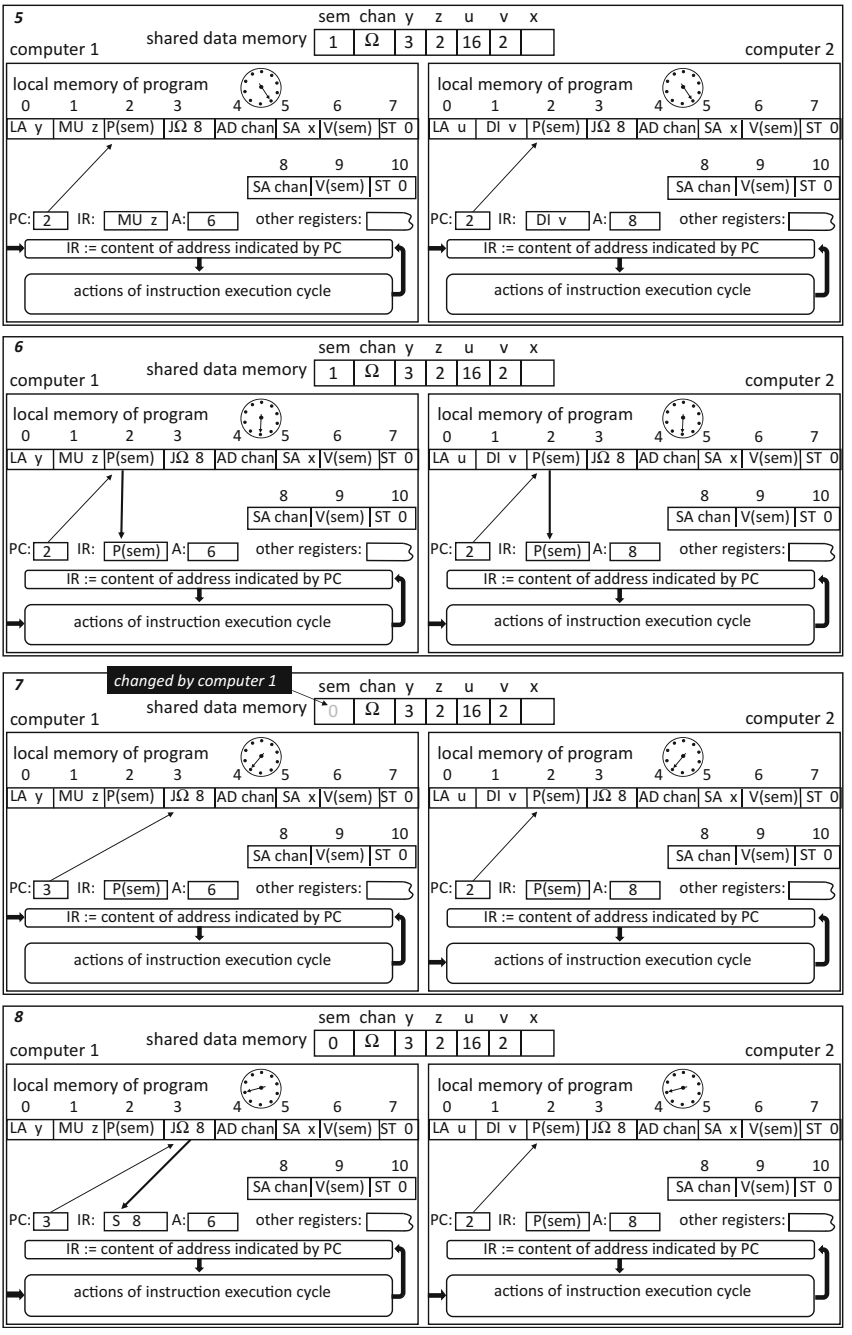
Table 1.5 illustrates an example of a process generated during execution of correctly programmed assignment statement $x := y * z + u / v$, performed by a system of two computers with shared memory for data. The conflict of access to the „communication channel", *chan*, is avoided by using instructions P(*sem*) and V(*sem*). The *sem* variable is a semaphore protecting critical section between these instructions. Though the pace (clock frequency) of both computers is similar, notice that the result of computation is independent of their pace.

Table 1.5 Parallel computation of assignment statement $x := y * z + u / v$, using semaphores



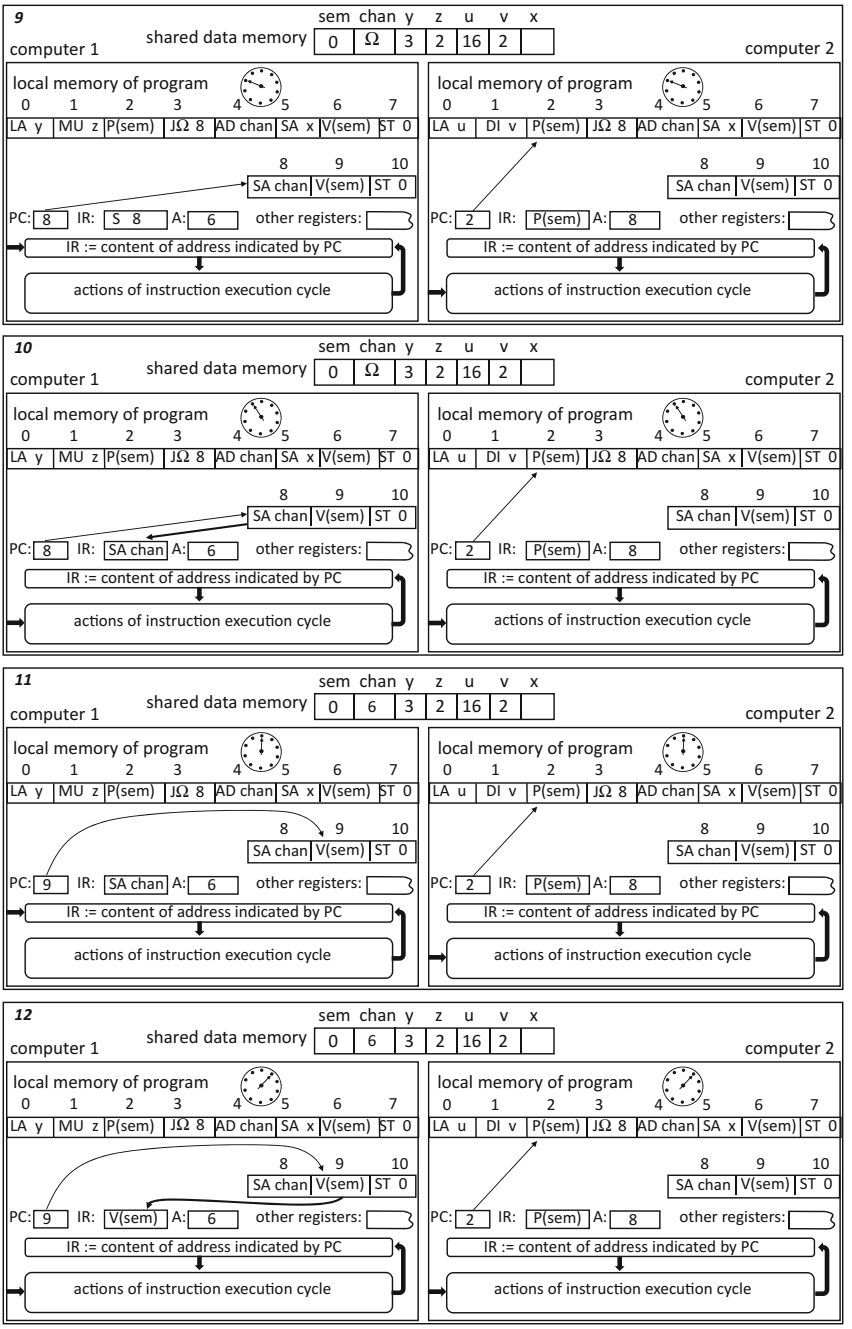
(continued)

Table 1.5 (continued)



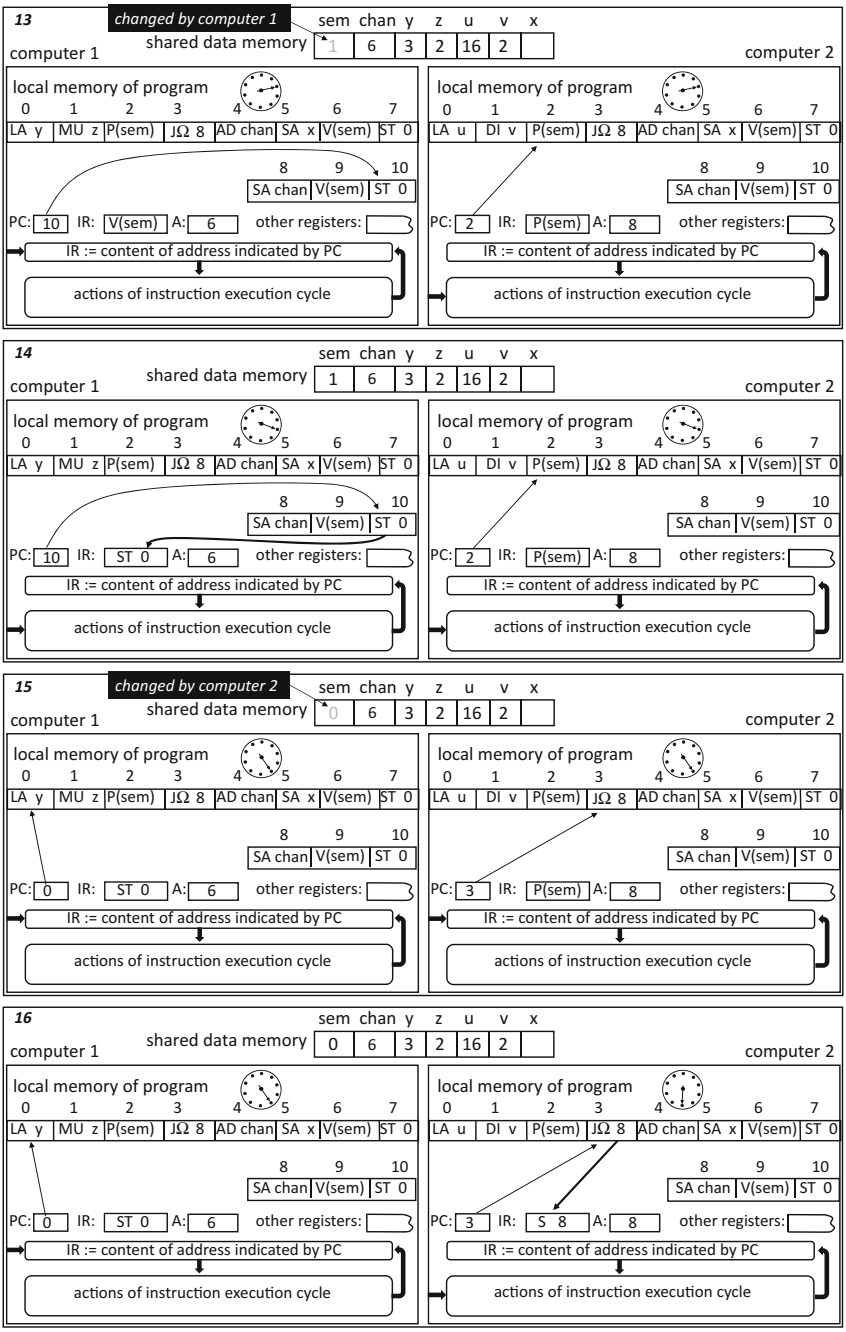
(continued)

Table 1.5 (continued)



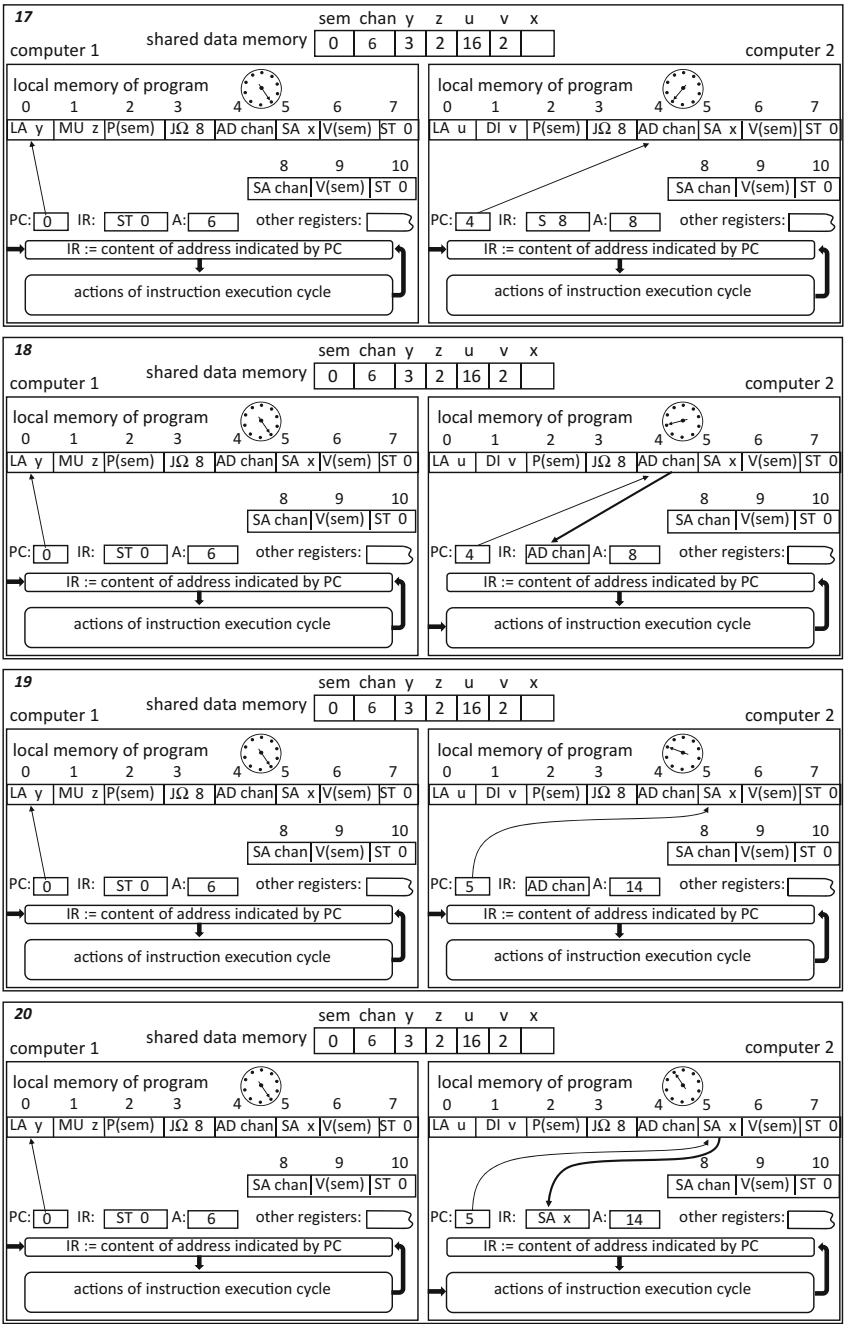
(continued)

Table 1.5 (continued)



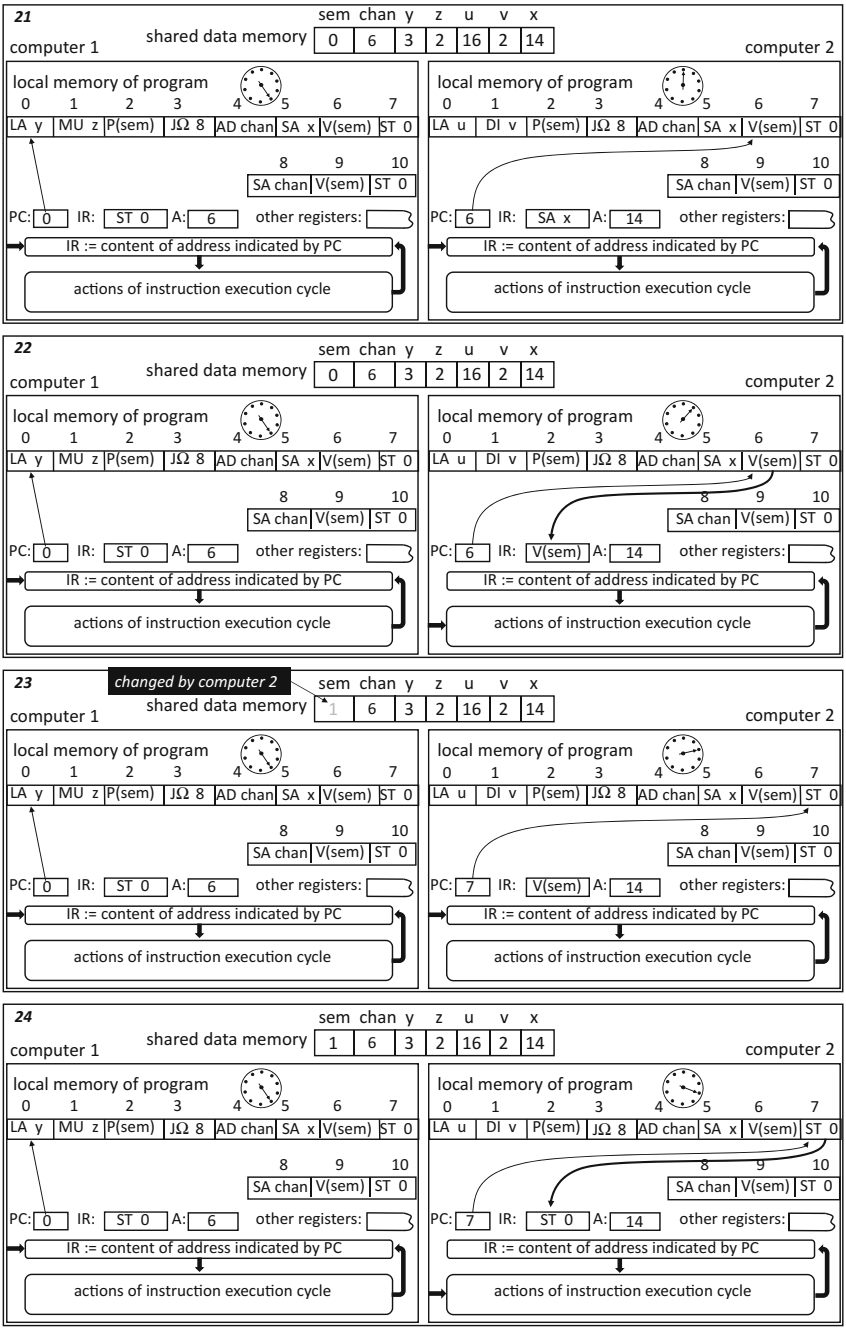
(continued)

Table 1.5 (continued)



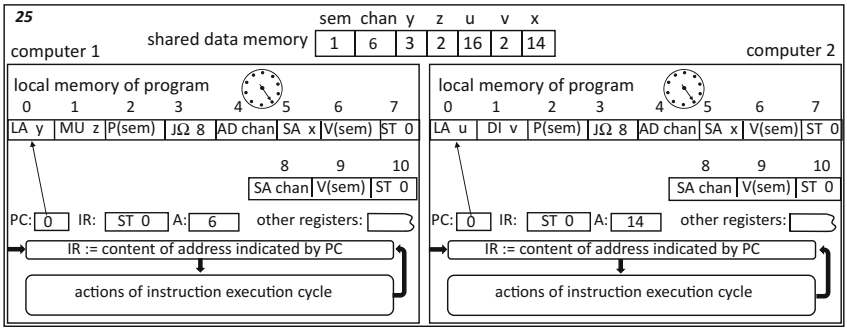
(continued)

Table 1.5 (continued)



(continued)

Table 1.5 (continued)



So far, synchronizing concurrent processes (occurring in not disjoint periods) has been considered. The problem was outlined by examples, using machine instructions of our computer model and a group of such cooperating computers. The main problem was ensuring mutual exclusion in access to shared resources, realized by means of operations on semaphores. Synchronization problems belong to a broader class of issues appearing in concurrent execution of programs. This is their coordination, which includes also scheduling of processes, like sequencing (serialization), that is, enforcing their desirable ordering or simultaneity of some actions (Sect. 7.2). Notice that mutual exclusion, in contrast to scheduling, does not require knowing names of processes that compete for resources, but only names of some shared variables—the semaphores. The sequencing problems constitute a vast domain of research and application, skipped here. One of the most important problems of distributed systems is communication between cooperating processes. In this chapter they will be outlined again by examples, on the low level—of machine instructions. To this end, names of processes and special instructions (commands) „send" and „receive", denoted concisely by symbols „!" and „?", will be introduced for synchronous communication, as well as symbols „S" and „R"—for asynchronous. Shortcuts „!" and „?" are borrowed from a theoretical model of communication between processes, called CSP (Communicating Sequential Processes), developed by Hoare in (1978, 1985).

1.4 Synchronous Communication of Processes

The instruction set in Fig. 1.3 is extended with the following instructions:

As shown in Fig. 1.4, process P1 sends the value of its variable n to process P2, which assigns this value to its variable m (in the original CSP notation, n is an expression, but m must be a variable; so the joined action of P1 and P2 is, in fact a distributed assignment statement $m := n$). The transmission of the value of variable

P2! *n* (instruction in program P1)	If program P2 is ready to receive a data from program P1, then send the content of cell with address *n* to program P2 (through communication channel) and go to execution of the next instruction; otherwise wait until P2 is ready.
P1? *m* (instruction in program P2)	If program P1 is ready to send a message to program P2 then receive this message from program P1, store it in a cell with address *m* and go to execution of the next instruction; otherwise wait until P1 is ready.

Fig. 1.4 Instructions of synchronous communication

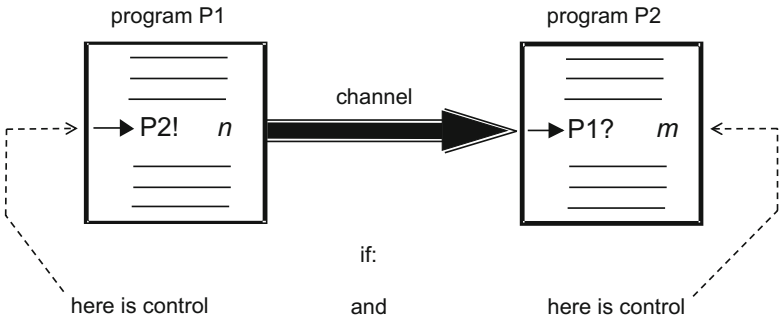


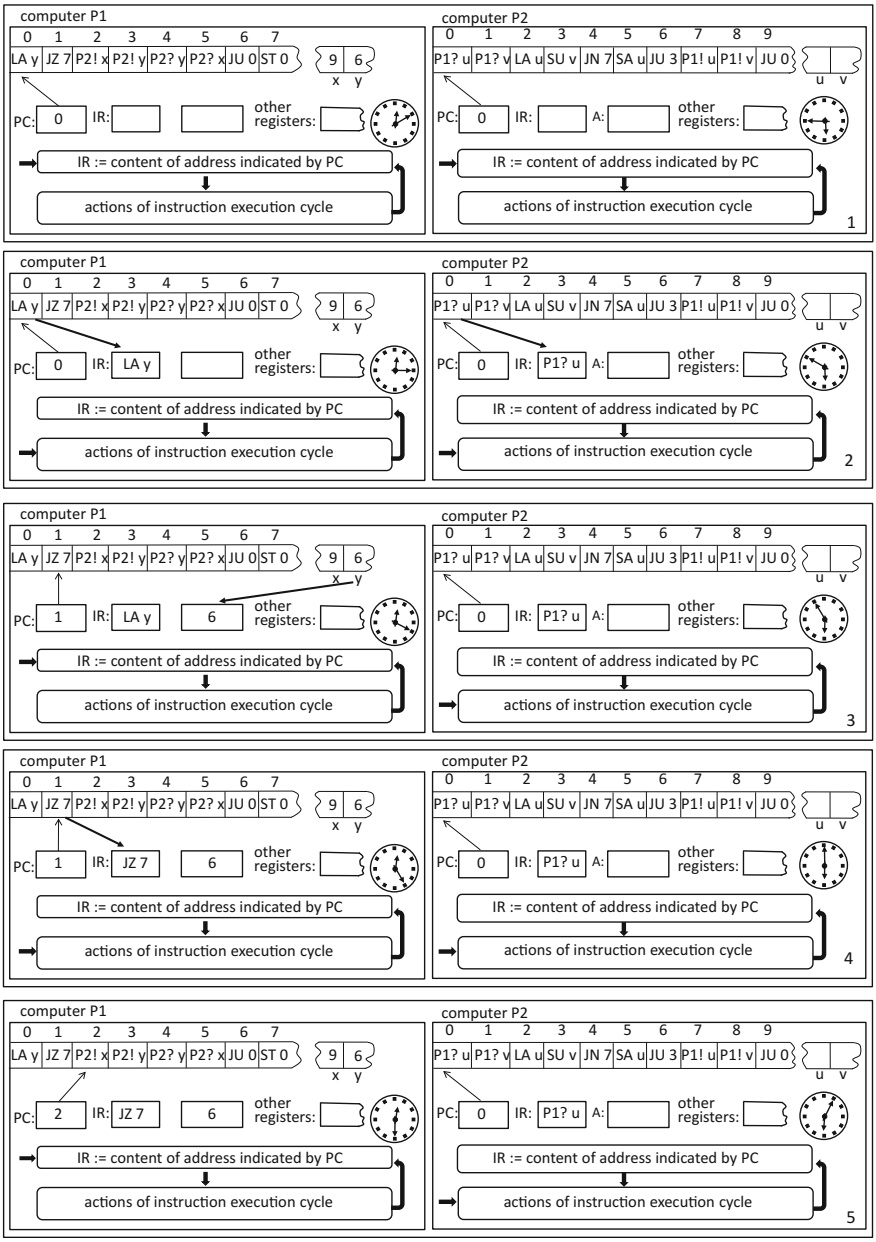
Fig. 1.5 Illustration of the principle of synchronous communication; the channel links ports or sockets in the processes (depending on a phraseology of concrete hardware or software implementation solutions)

n from P1 to P2 takes place, when instructions P2! *n* and P1? *m* are being executed jointly, that is when control of both processes reached these actions. Such „meeting" of the processes is called their *handshaking*, or *randezvous*: when one program is ready to communicate and its partner is not, then the former waits until the latter be ready. This is called a synchronous communication whose principle is depicted in Fig. 1.5. Such mechanism has been implemented in many programming languages, for instance in ADA, OCCAM and a number of more commonly used like Java, C++, etc.

Notice that such communication is *speed-insensitive* (computing result is independent of relative computation speed of computers) and resembles a phone call: the caller waits until the callee picks up the telephone receiver.

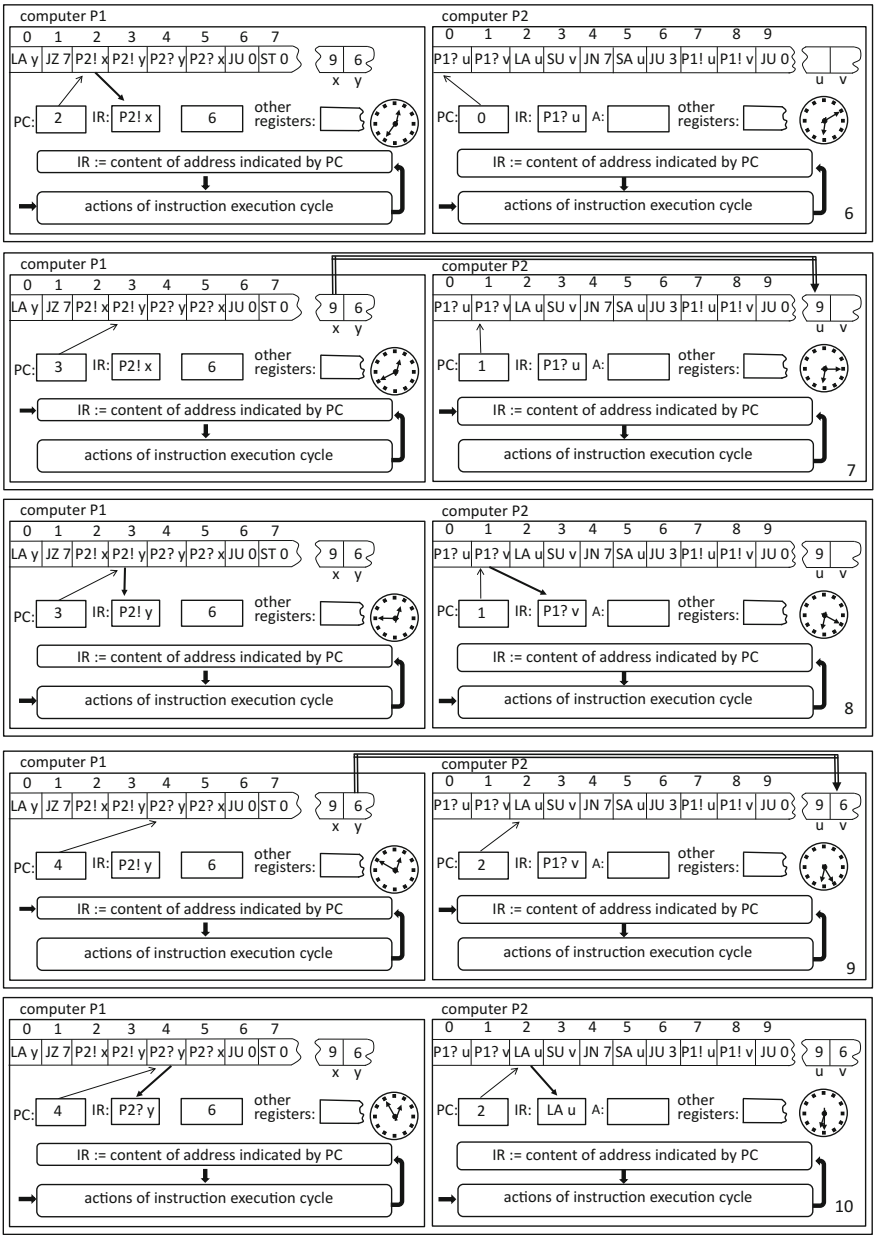
As an example consider a system of two-computers, which computes the greatest common divisor (gcd) of two integer numbers, where at least one is not 0. Its execution as consecutive states, is shown in Table 1.6. Since the task is not so trivial as the previous exemplary parallel computing of a simple assignment statement and more instructive for its algorithmic and program parallelization aspects, let us devote a little more attention to it.

Table 1.6 Computing gcd (9, 6) = 3 by two computers communicating synchronously



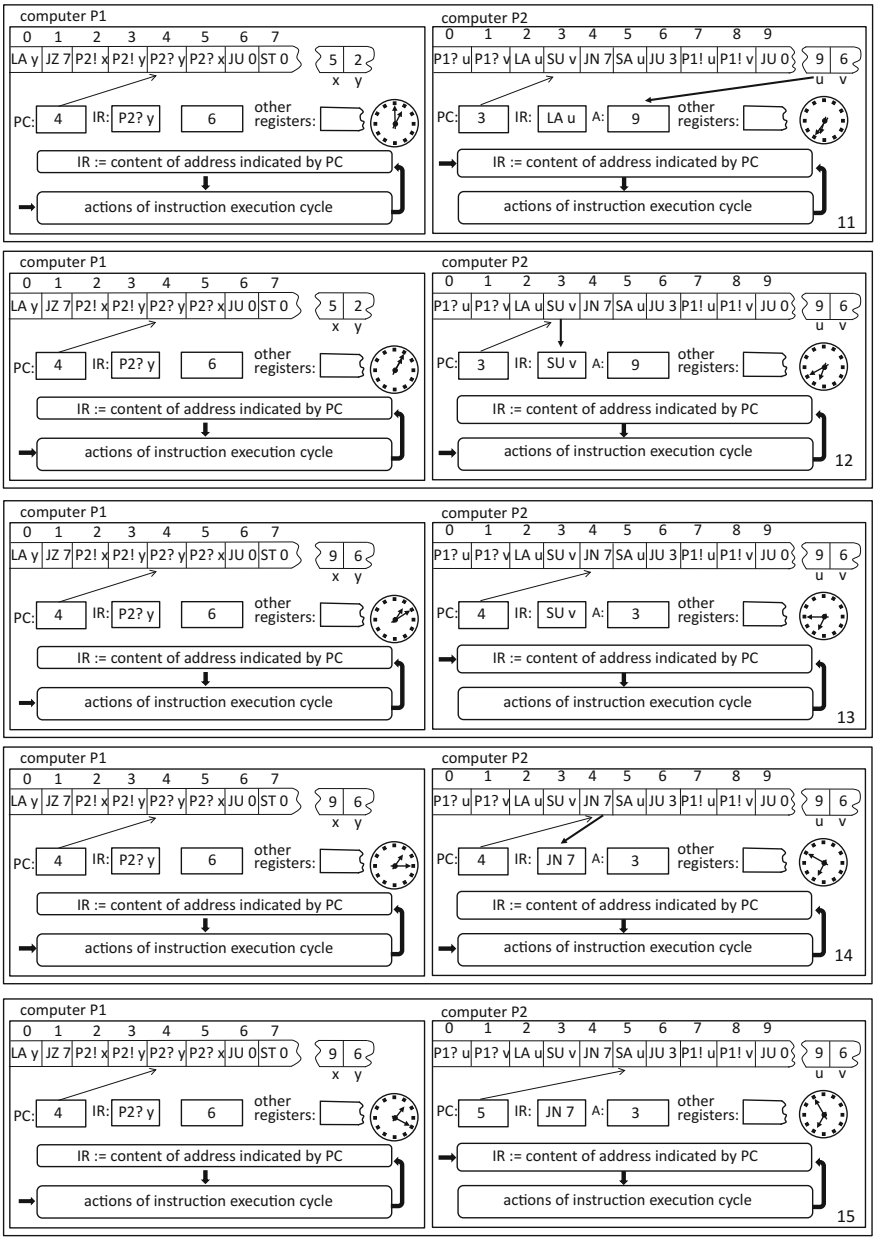
(continued)

Table 1.6 (continued)



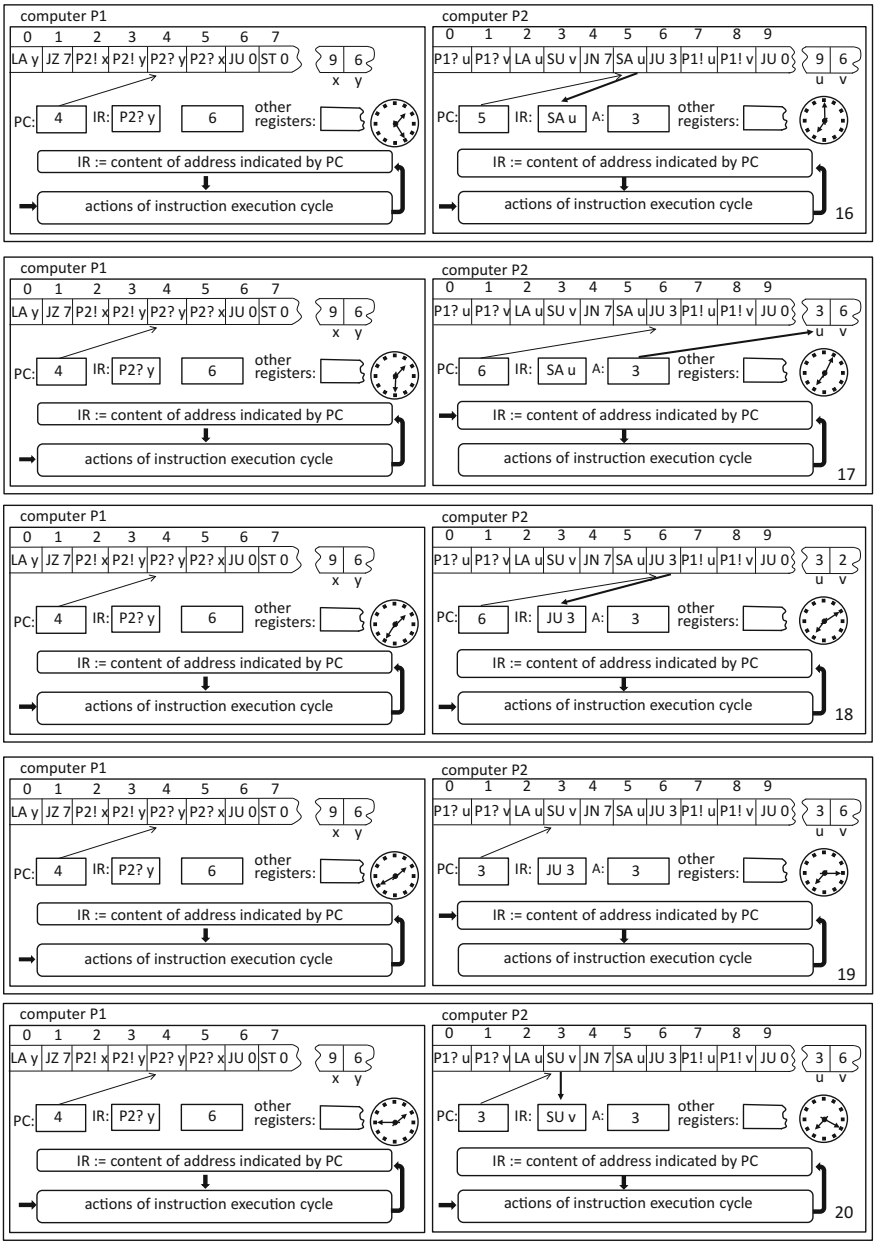
(continued)

Table 1.6 (continued)



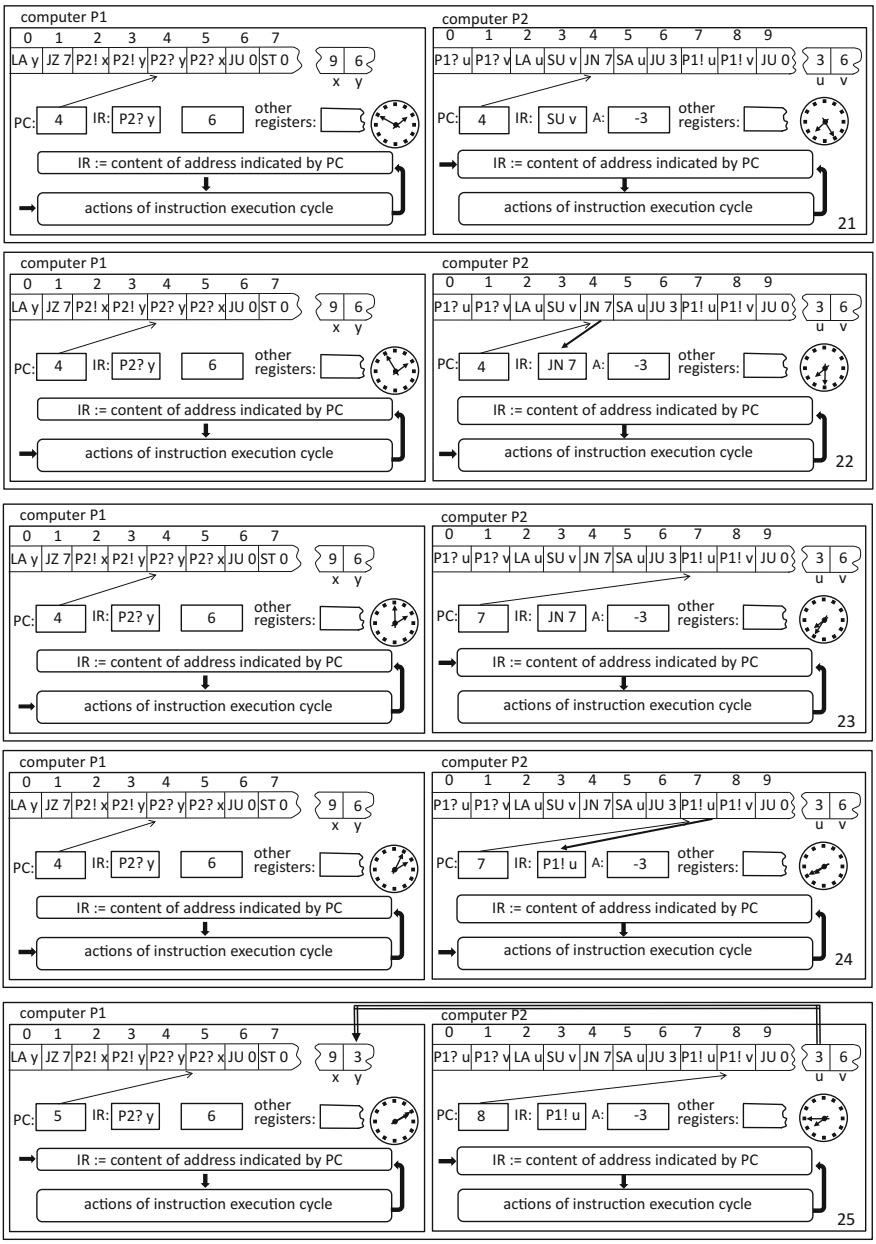
(continued)

Table 1.6 (continued)



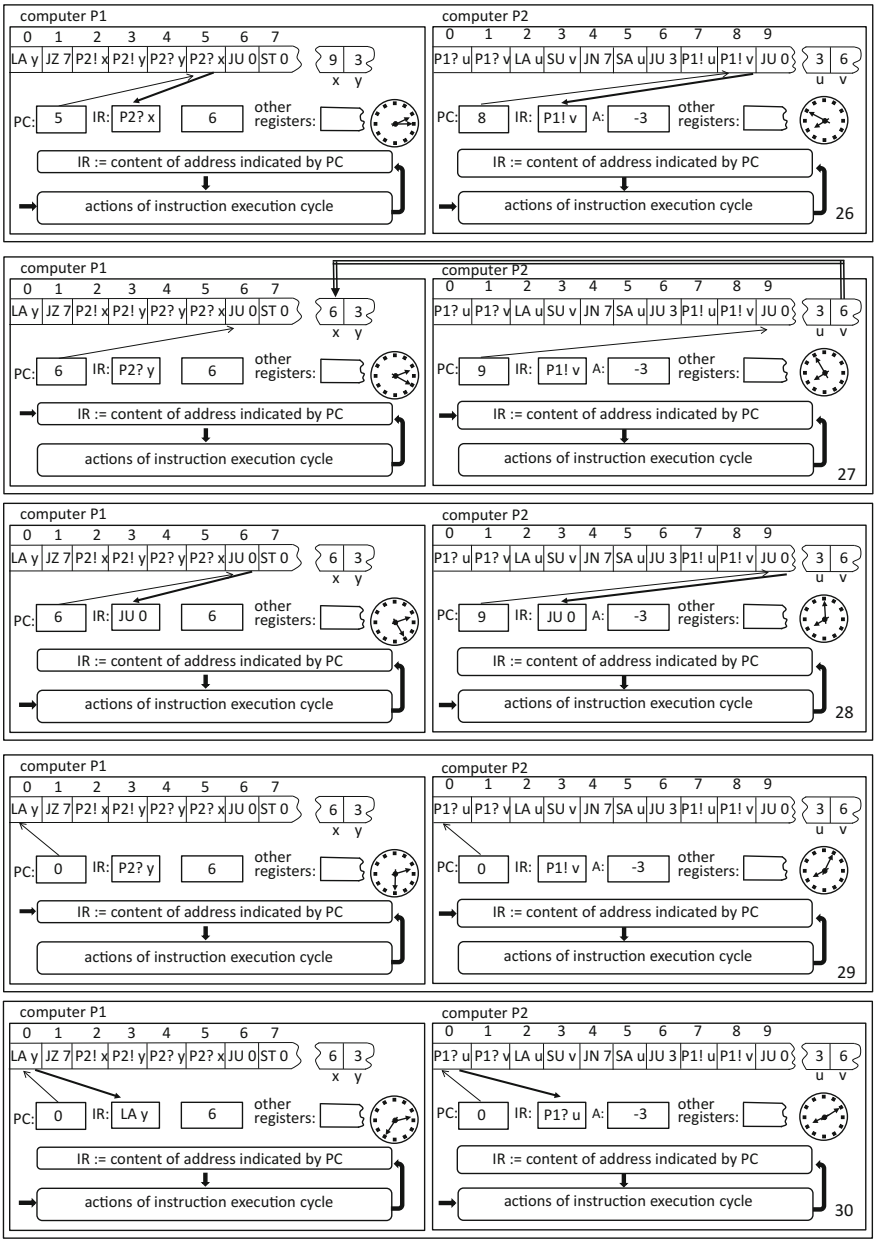
(continued)

Table 1.6 (continued)



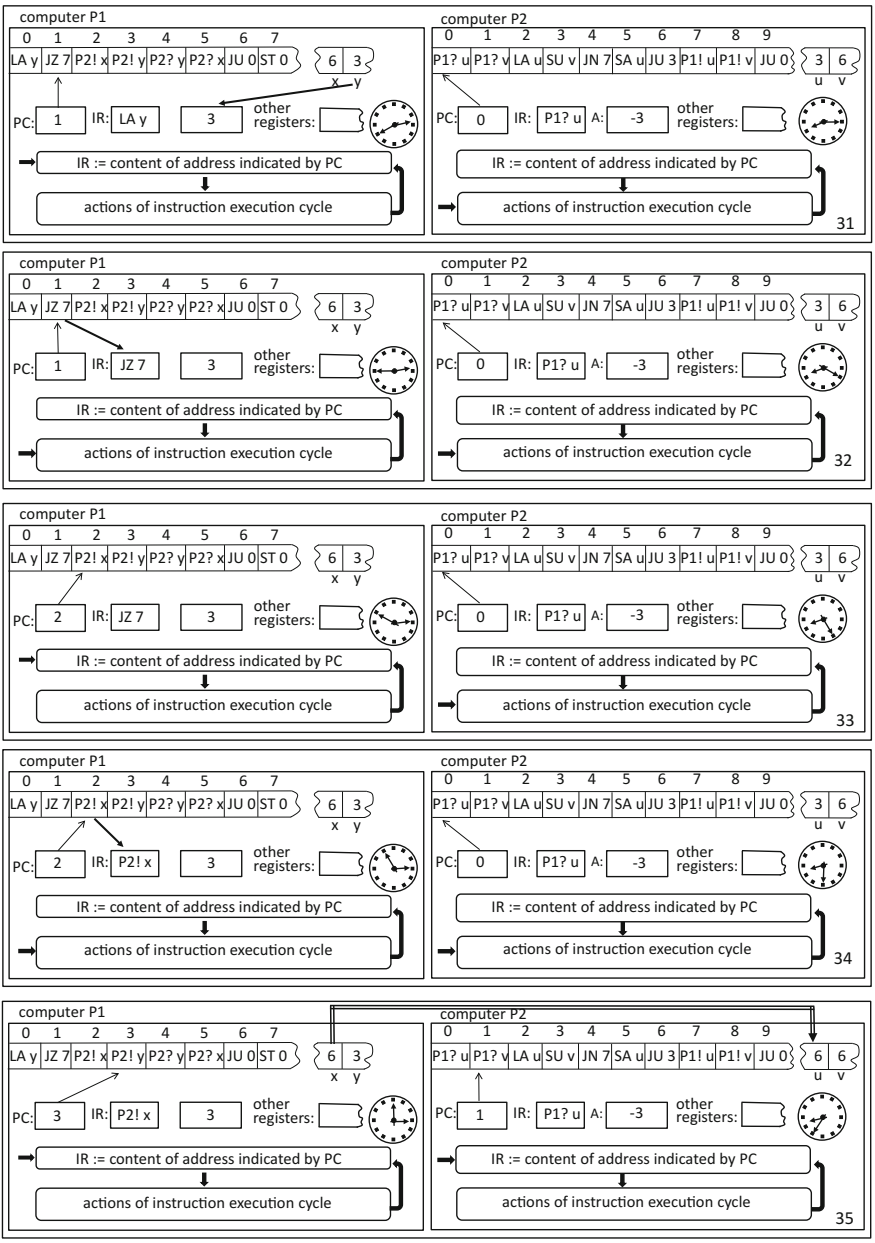
(continued)

Table 1.6 (continued)



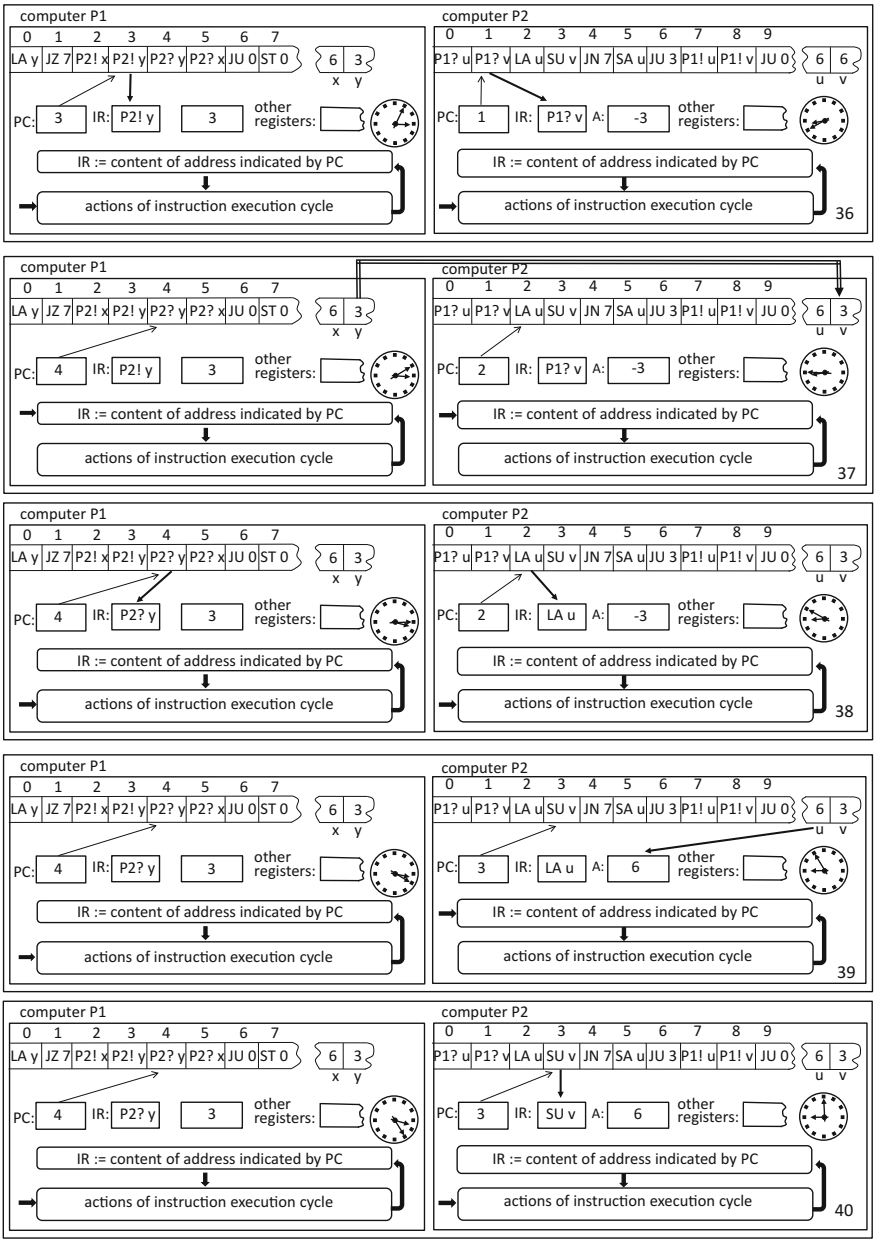
(continued)

Table 1.6 (continued)



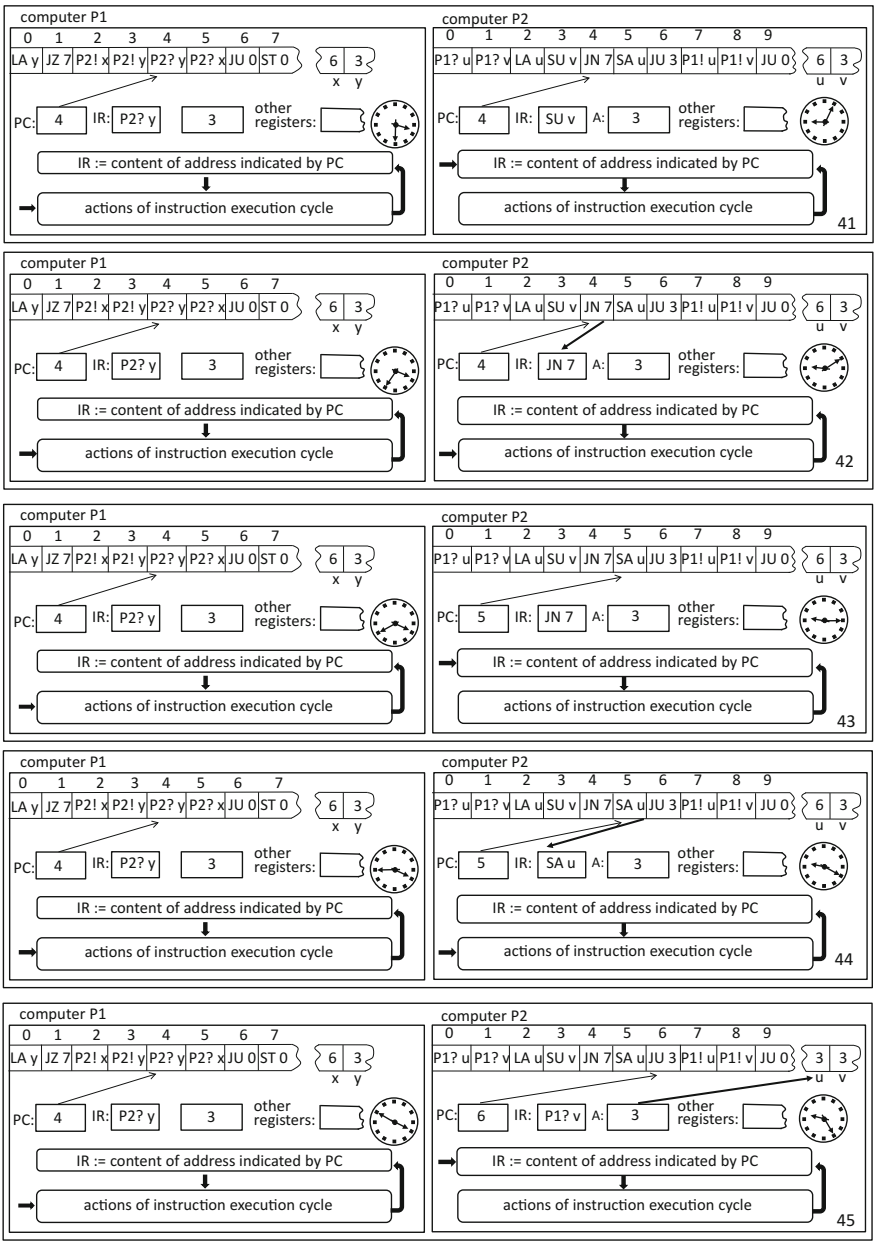
(continued)

Table 1.6 (continued)



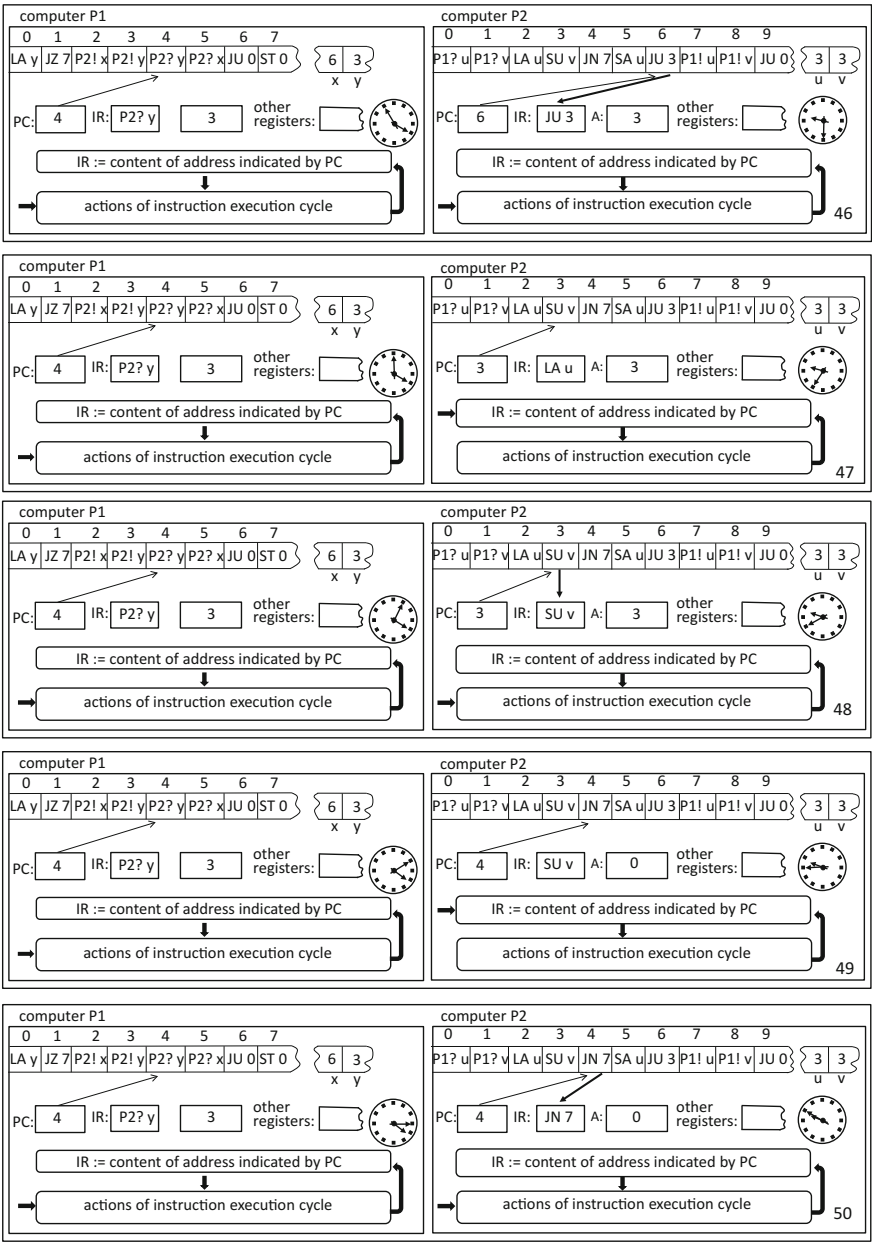
(continued)

Table 1.6 (continued)



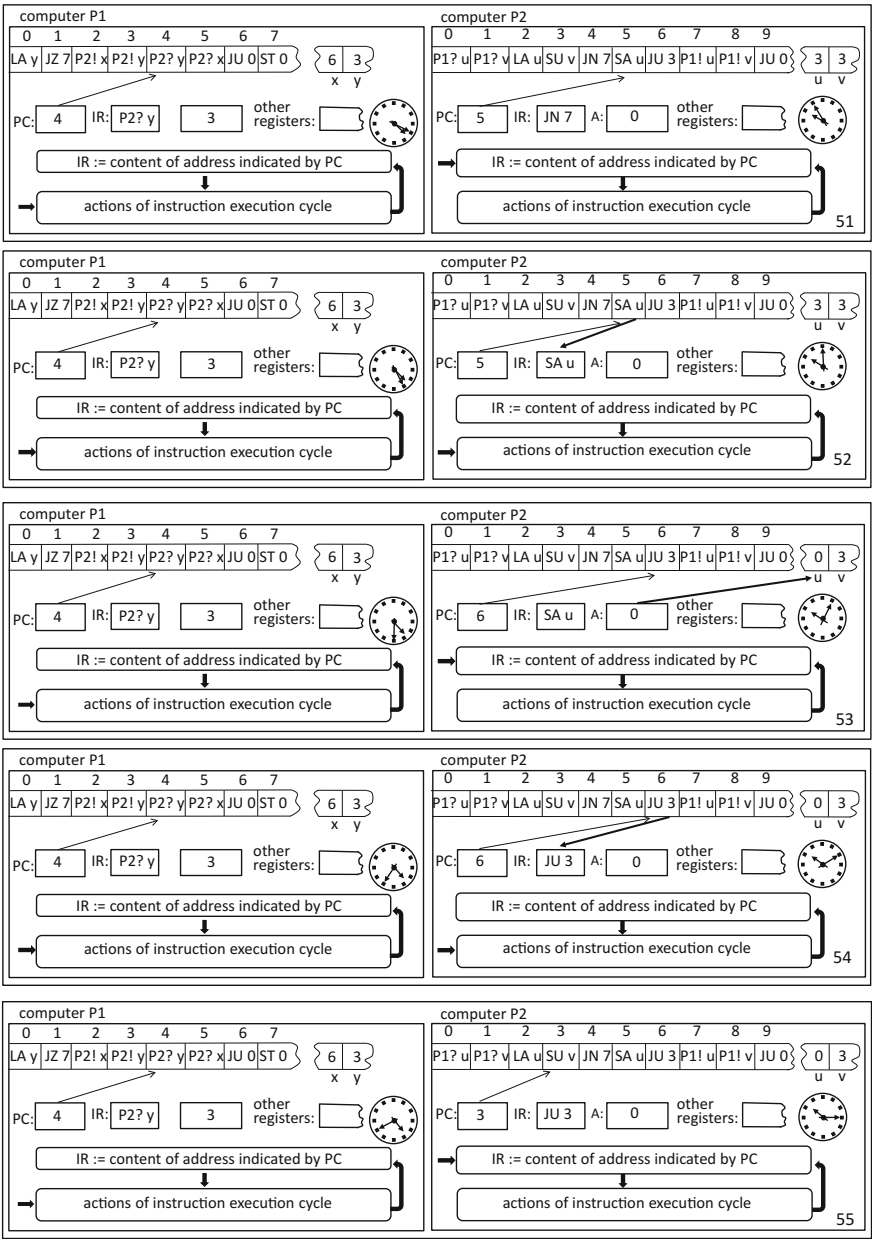
(continued)

Table 1.6 (continued)



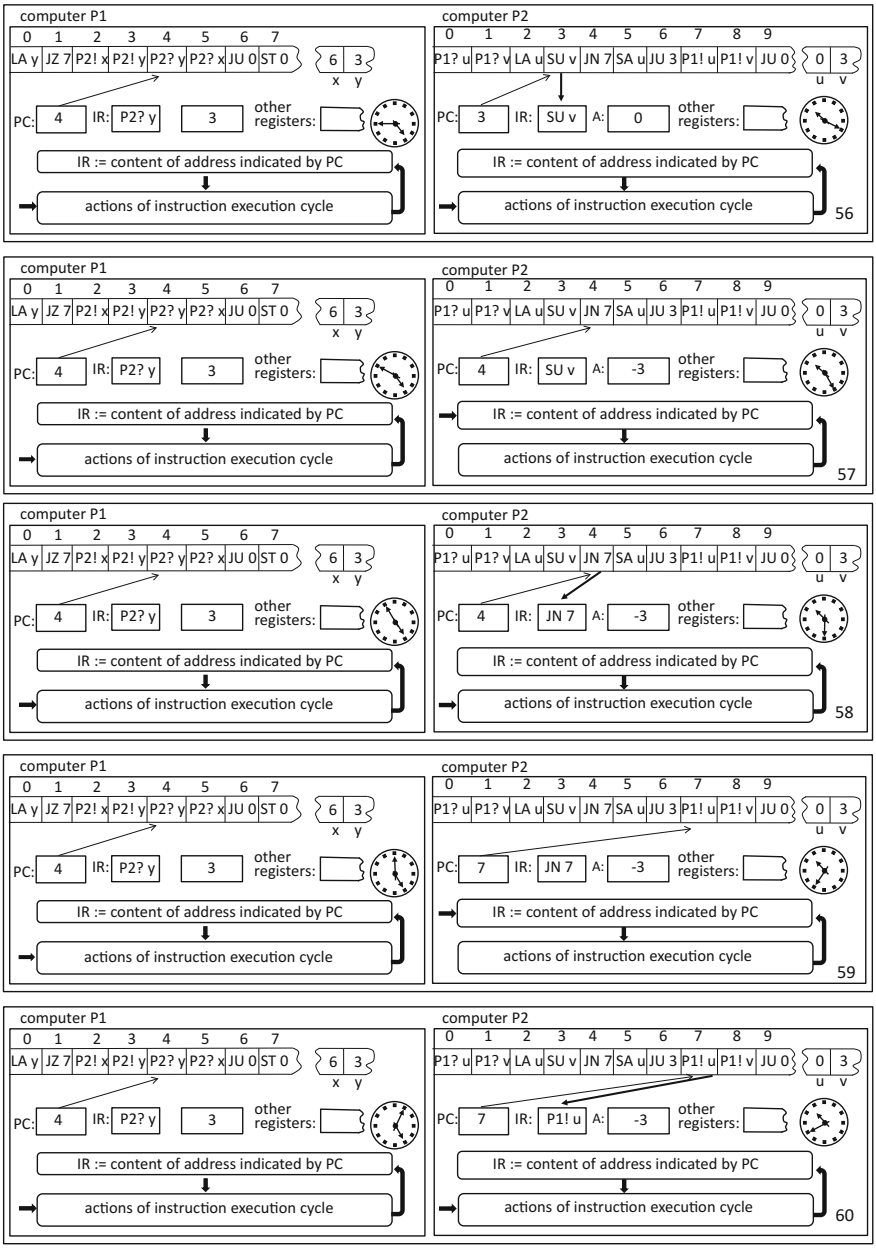
(continued)

Table 1.6 (continued)



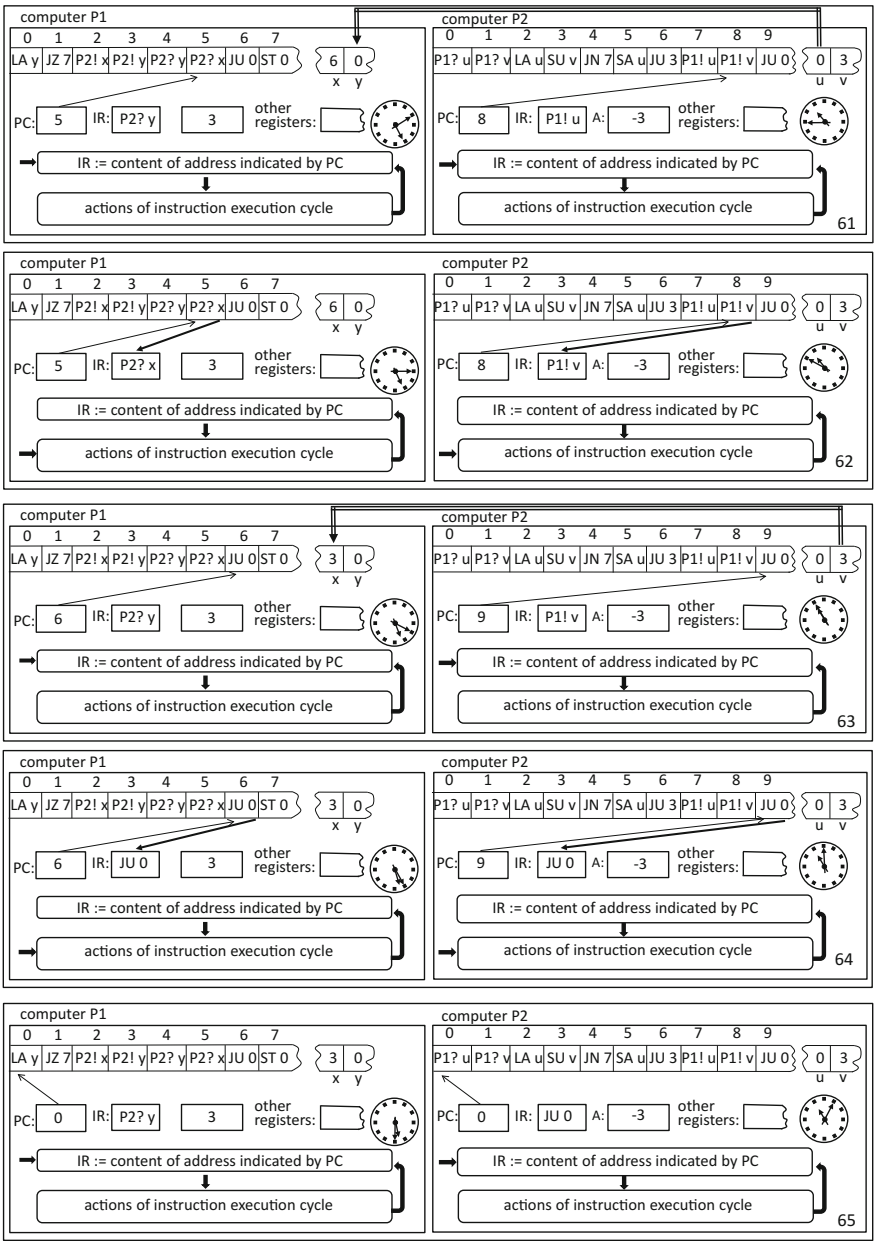
(continued)

Table 1.6 (continued)



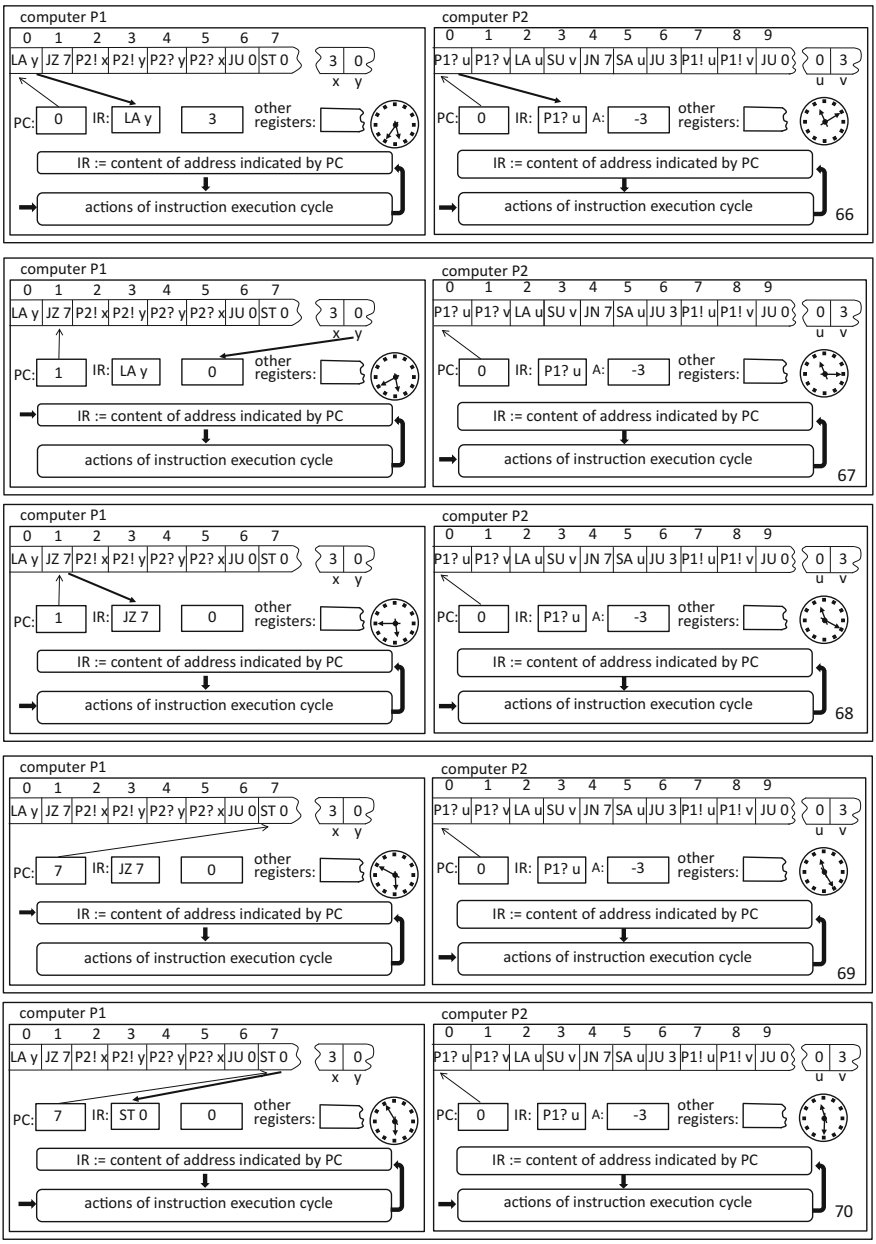
(continued)

Table 1.6 (continued)



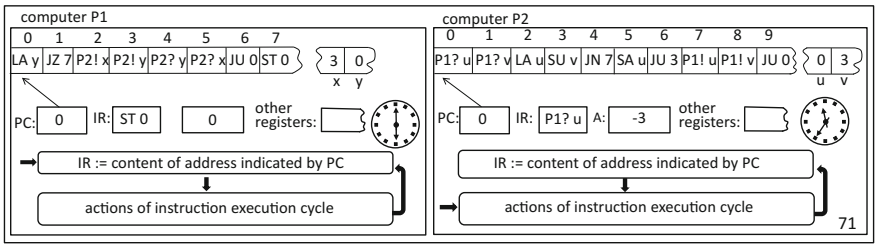
(continued)

Table 1.6 (continued)



(continued)

Table 1.6 (continued)



First, the algoritm is based on the ancient theorem ascribed to Euclid and states the following: if $x > 0$ then gcd of x and y equals x when $y = 0$; gcd of x and y equals gcd of y and r when $y > 0$, where r is the remainder of division x by y. Concisely: $\gcd(x, 0) = x$ and $\gcd(x, y) = \gcd(y, r)$, where $r = x \bmod y$. Therefore, the algorithm may look as follows:

step 1. If $x > 0$ and $y = 0$ then stop—the gcd is x, otherwise go to step 2;
step 2. $r := x \bmod y$; $x := y$; $y := r$; go to step 1;

Notice that the algorithm will terminate activity, because the sequence of remainders in consecutive executing actions in step 2 is decreasing and each decreasing sequence of natural numbers is finite.

Second, the algorithm will be distributed among two processes (here—computers): the actions of step 1 will be committed to computer 1, which sends values of x and y to computer 2 if $y > 0$, while computation of remainder r will be performed by computer 2, which sends values of r and y to computer 1.

Third, the algorithm may be coded concisely as the system of two programs P1 and P2 (shown in Fig. 1.6), written in machine instructions (from Figs. 1.3 and 1.4) and loaded into two communicating computers.

It is assumed that input values of x, y are already stored in memory of computer 1 before actions of P1 and the result $\gcd(x, y)$ is assigned to variable x on termination of this program. The consecutive remainders computed by program P2 are stored in variable u and transmitted to program P1, which stores it in variable y. Notice that the separate variable for remainders is not needed due to synchronous communication between the programs: its role plays communication channel linking instruction P1! u with instruction P2? y. Table 1.6 illustrates two concurrent processes generated during execution of this system with input values $x = 9$, $y = 6$ and the same speed of clocks. Notice, however that the result would be the same if their speeds were different. The result $\gcd(9, 6) = 3$ resides in variable x on termination of the processes. Both processes start simultaneously, but at different local times.

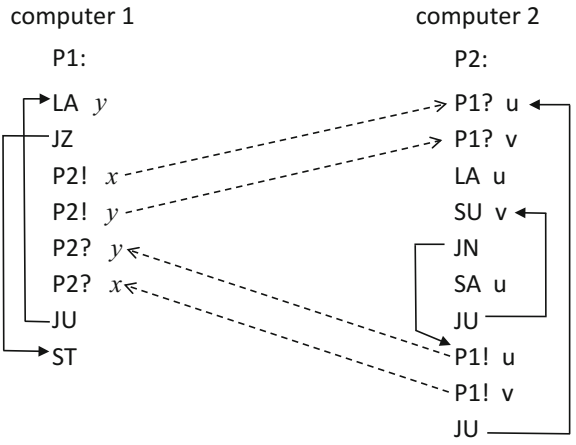


Fig. 1.6 Synchronous interprocess communication when computing the greatest common divisor; the dashed lines show passing messages, the solid–control

Though the gcd algorithm has been used here only as an example of its distributive arrangement with synchronous communication, the importance of the gcd notion, both in theory and many applications, its computational methods and complexity is sketched in the few following, but not exhaustive remarks:

1. The notion of gcd is commonly used for reducing fractions to the lowest numerator and denominator.
2. The gcd for numbers may be extended to gcd for polynomials or to more abstract mathematical notions, like rings, in particular to rational and real numbers.
3. The gcd is used in factorization of integer numbers (in the simplest case) into primes: in 1. representing them as product of prime numbers.
4. The gcd is used in computational geometry.
5. The notion of gcd is used in the cryptographic protocols, to secure network communications and in methods for breaking the cryptosystems by factoring large composite numbers.
6. There are known a number of algorithms computing the gcd, more efficient (of lower computational complexity than the Euclidean algorithm shown in this chapter) but more mathematically involved, some of them for some special numbers only.
7. There are some probabilistic methods to asses if a number (or an element of a ring) is a gcd of given pair of numbers.
8. There are some methods of parallelization of computing of gcd, more complicated than shown in this chapter.

9. A great many investigations on computational complexity of gcd have been carried out and results for particular algorithms obtained. However, so far the lower bound of parallel computing complexity of the gcd problem (roughly speaking—the most efficient algorithm, not discovered hitherto), is an open problem (Karp and Ramachandran 1990).
10. Some investigations on correctness of the gcd algorithms, especially their distributed versions, are important examples belonging to the theory of computation.

1.5 Asynchronous Communication of Processes

In order to explain the principle of asynchronous communication, let us imagine the following organization of its participants. Each program has a mailbox of messages delivered by senders to it. The mailbox is partitioned into pigeon-holes, each assigned to one sender and containing a queue of messages sent by this sender. The receiver of these messages, when needs a message from a certain sender, takes it from a queue assigned to this sender—if the queue is nonempty. Otherwise, the receiver waits until the sender dispatches the message to this queue. So, unlike in synchronous communication, the sender is not suspended until receiver gets the message, but continues activity, and symmetrically for the receiver—unless its respective queue is not empty.

The instruction set is extended with two instructions shown in Fig. 1.7.

The asynchronous communication mechanism has been implemented for some programming languages (in their syntax or libraries), like LINDA Carriero et al. (1986), Carriero and Gelernter (1989), some extensions of COBOL and even the old Fortran and a number of newer, like C#, Visual Basic, JavaScript and some others. The principle of asynchronous communication is depicted in Fig. 1.8.

S(P2) *n* (instruction in program P1)	Send the content of cell with address *n* to the tail of queue assigned to program P1 in the mailbox of program P2 and go to execution of the next instruction;
R(P1) *m* (instruction in program P2)	If the queue assigned to program P1 in the mailbox of program P2 is non-empty, take a message from the head of this queue, store it in the cell with address *m* and go to execution of the next instruction; otherwise wait until the queue becomes non-empty.

Fig. 1.7 Instructions of asynchronous communication; S stands for send, R—for receive

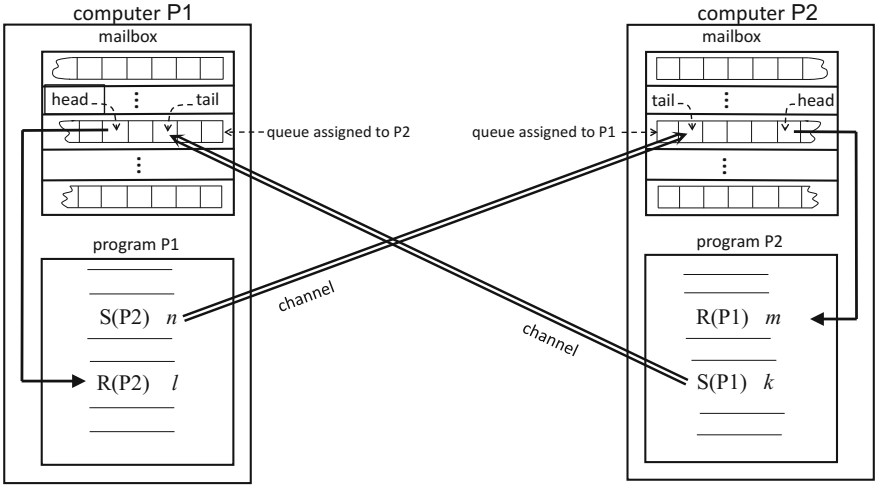


Fig. 1.8 Illustration of the principle of asynchronous communication

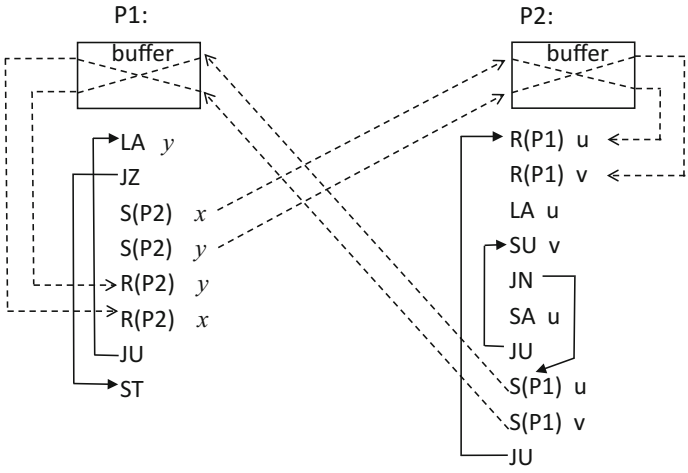
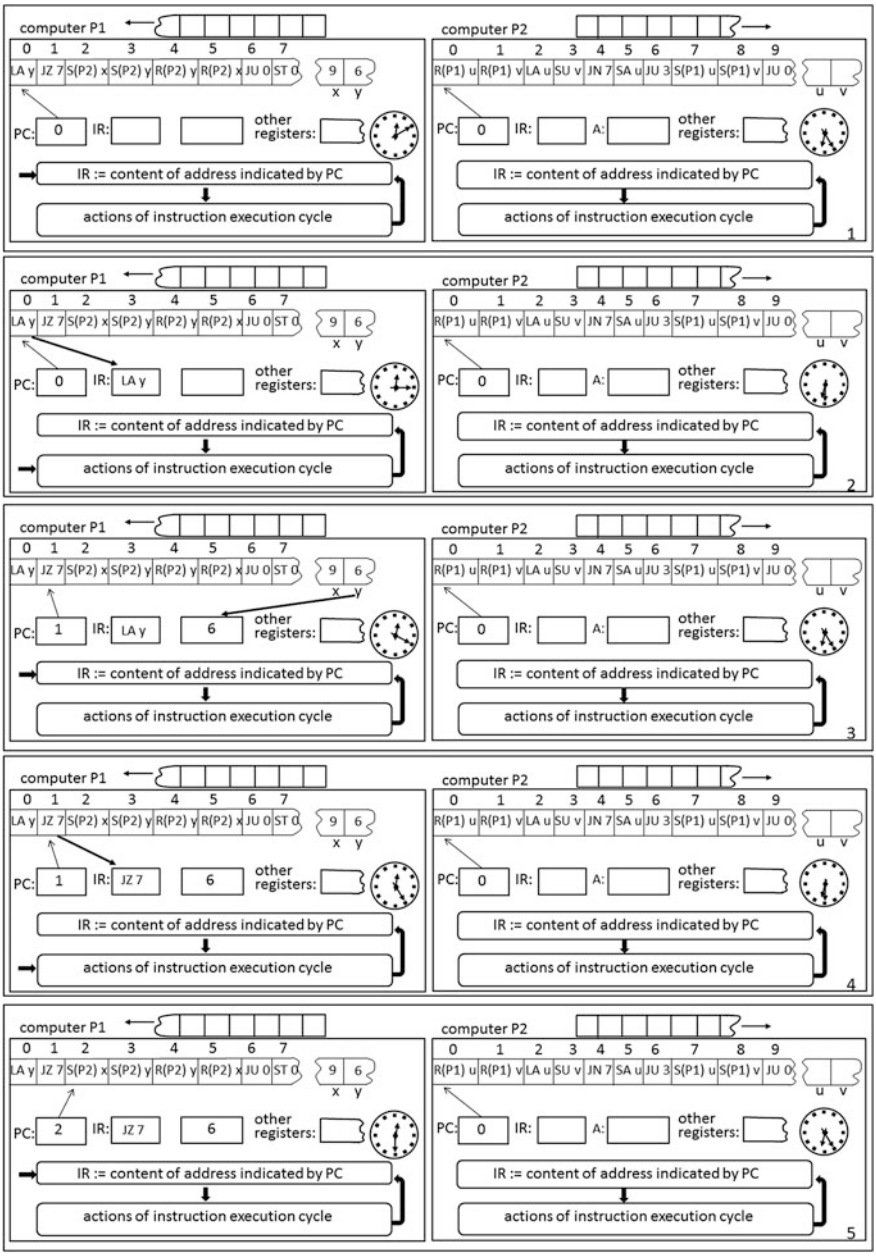


Fig. 1.9 Asynchronous interprocess communication when computing the greatest common divisor; The dashed lines show passing of messages, the solid lines—control

Notice that the program in Fig. 1.6 computing the gcd by means of synchronous communication may be rewritten as in Fig. 1.9, with replacement of synchronous instructions „!", „?" with asynchronous „S", „R" and yields the same result, provided that the order of receiving messages is the same as the order of sending them (i.e. in accordance with the FIFO discipline).

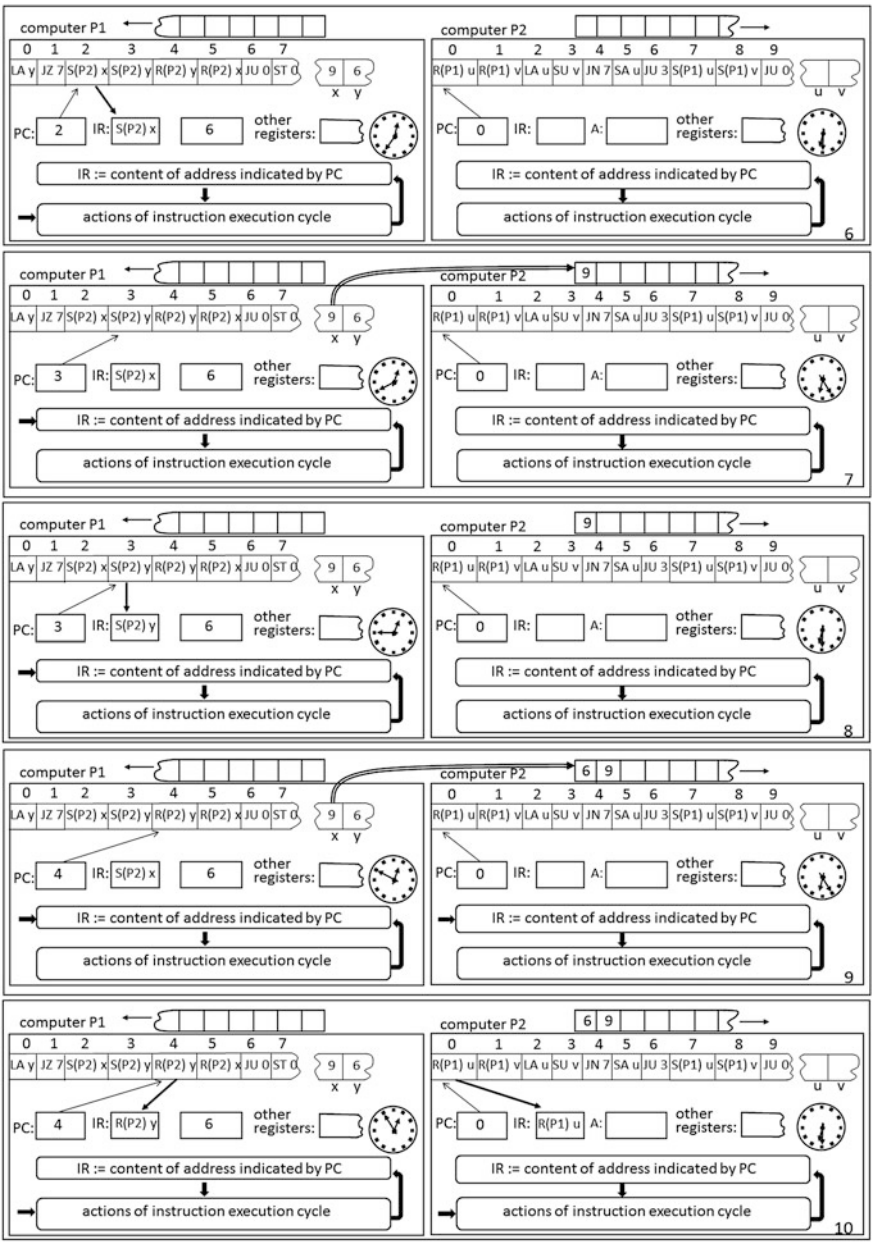
Table 1.7 illustrates a part of two concurrent processes, communicating asynchronously and generated during execution of this system for input values $x = 9$,

Table 1.7 Asynchronous computing gcd (9, 6) = 3



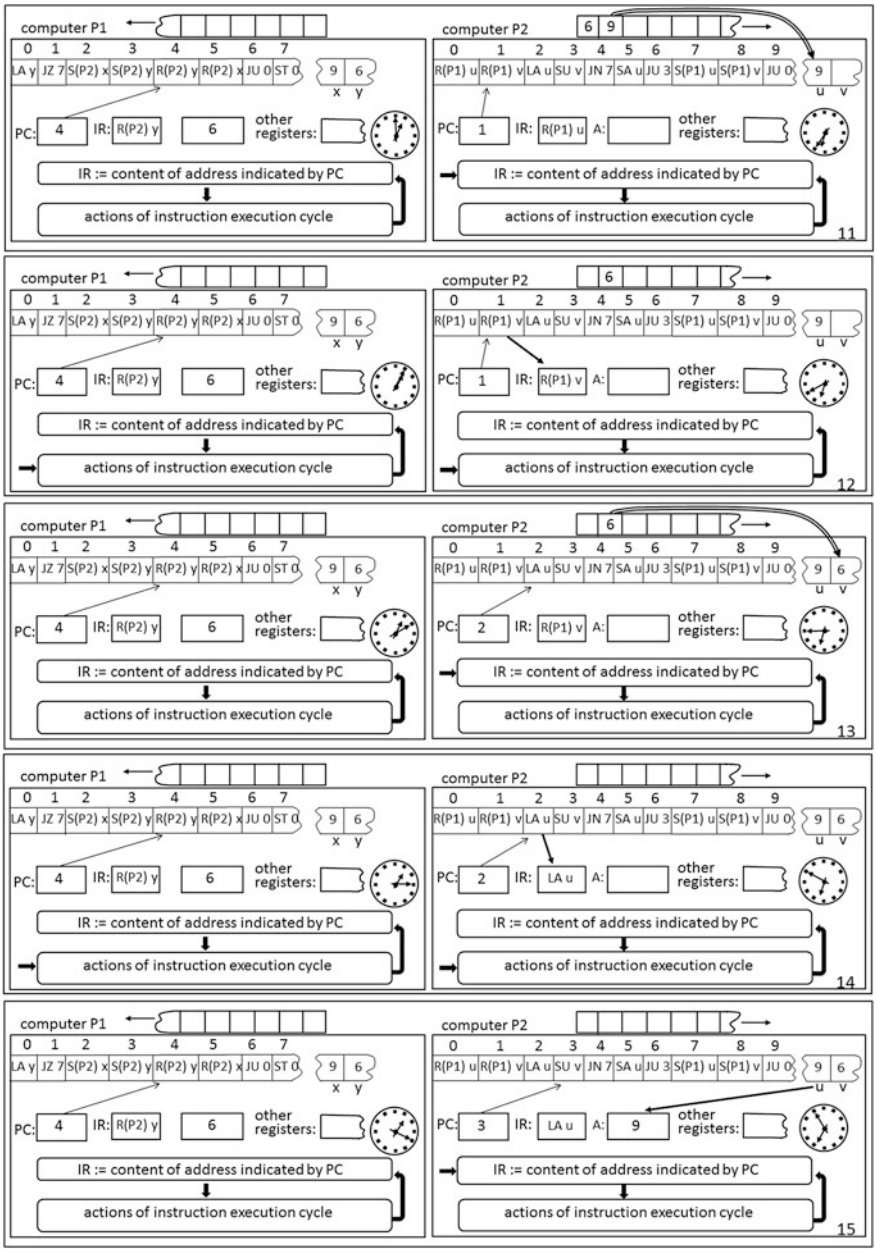
(continued)

Table 1.7 (continued)



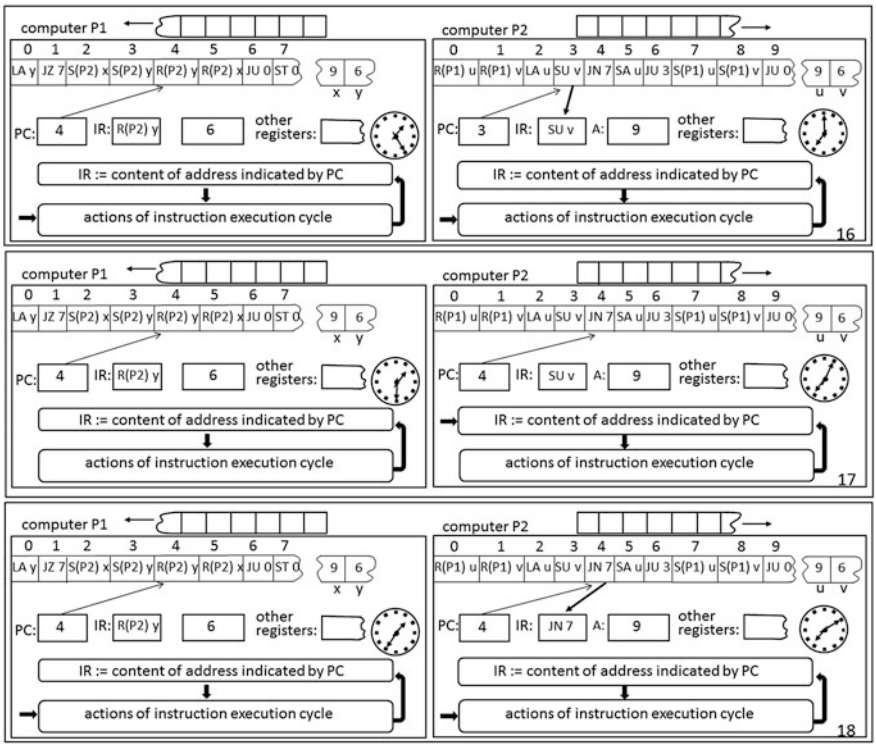
(continued)

Table 1.7 (continued)



(continued)

Table 1.7 (continued)



$y = 6$ and the same speed of clocks' pace. Process P2 is idle until the state 8, whereas process P1 starts in the state 1 and waits, since the state 10, for data from P2 (i.e. the remainders) that will be put in the queue (buffer) located in the mailbox associated with P1, collecting data from P2. The reader may continue the processes until the end, when the variable x gets value 3 as the value of gcd (9, 6). Notice that, every process may be suspended only when its control reached instruction „receive" (R) with empty queue, unlike in case of synchronous communication, when the control reaches also instruction „send" (S).

1.6 Synchronous Vector Systems

Considerations on the level of machine instructions will be concluded with demonstration of action of a synchronous vector system, composed of very simple identical processors. Each of them executes only four instructions and has access to shared memory for data. It is assumed that their clocks, of identical frequencies, are precisely synchronized (or equivalently: the processes use a common clock). Such systems, composed of a great number of very simple processors, are capable of offering large computing power, when performing algorithms, where identical instructions (machine commands) in each processor are being executed simultaneously in one instruction execution cycle. Systems of such architecture are specialized for certain tasks, for instance, in computation with vectors or matrices. To illustrate such system, let us consider a simple example of adding vectors of four components (obviously the gain of such architecture is significant in case of vectors of large number of components). Given vectors:

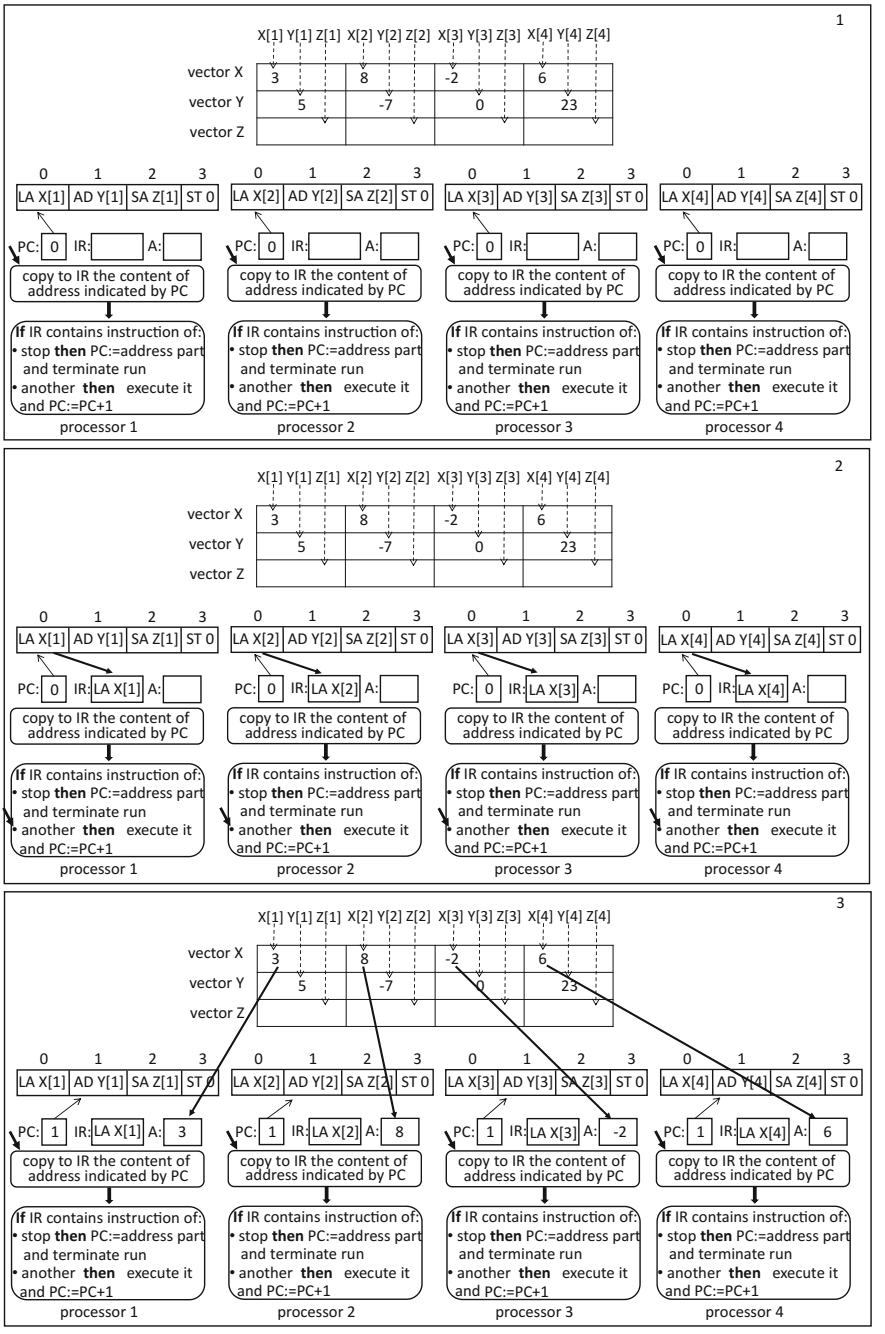
$$X = (X[1], X[2], X[3], X[4]) = (3, 8, -2, 6)$$
$$Y = (Y[1], Y[2], Y[3], Y[4]) = (5, -7, 0, 23)$$

let us compute their sum:

$$Z = X + Y = (Z[1], Z[2], Z[3], Z[4]) = (8, 1, -2, 29)$$

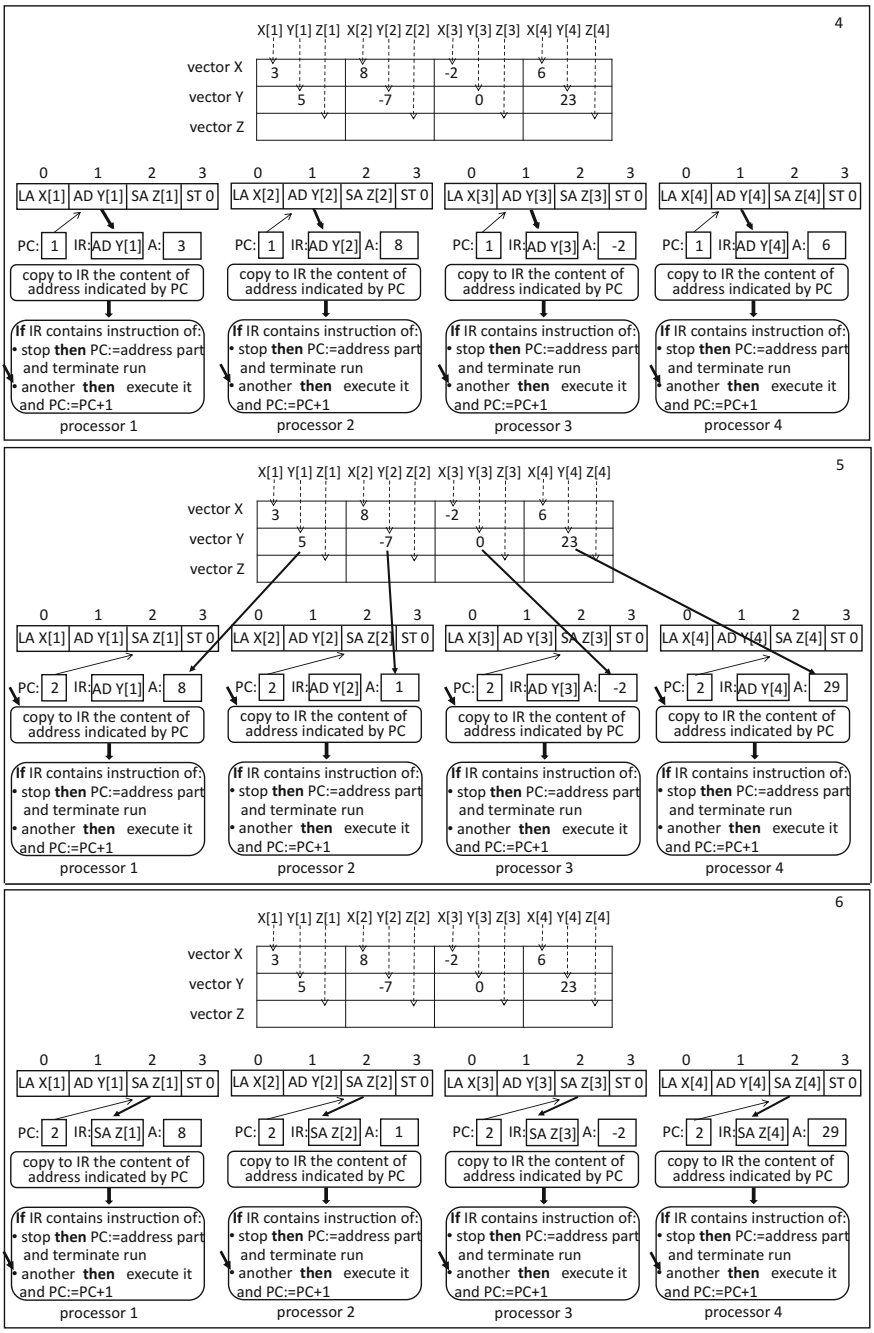
Table 1.8 shows the activity of of 4-processor vector system, computing the sum X + Y and storing result in the vector Z. Note that computation of sum of n-components vectors takes as much time as computing sum of two numbers $X[i] + Y[i]$ ($i = 1,\ldots, n$): only 4 instructions are being executed, instead of at least 4 ($n + 1$), when performed by one processor. Replacing instruction of addition (AD) with multiplication (MU), leads to computing products of respective components, whose sum yields the inner product of vectors (summation of the products may be performed by an algorithm which would sum up all the products; such efficient algorithms are elaborated and easily found in the literature). Note that computation of the inner product of vectors is a basic activity of computation of the product of matrices, which is encountered in a number of problems, like solving linear equations systems, fast Fourier transform (FFT) and others. For this purpose, synchronous matrix architectures are devised, included in the supercomputers, as well as very fast, so-called systolic arrays, of very large integration scale (VLSI), worked out by Kung and Leiserson (1979), Petkov (Petkov 1992), for special tasks. Architectures like vector, matrix or others of regular interconnection structures between simple but numerous processing and memory units, acting synchronously, are sometimes referred to as *massively parallel*.

Table 1.8 Activity of of 4-processor vector system, computing sum of vectors X, Y



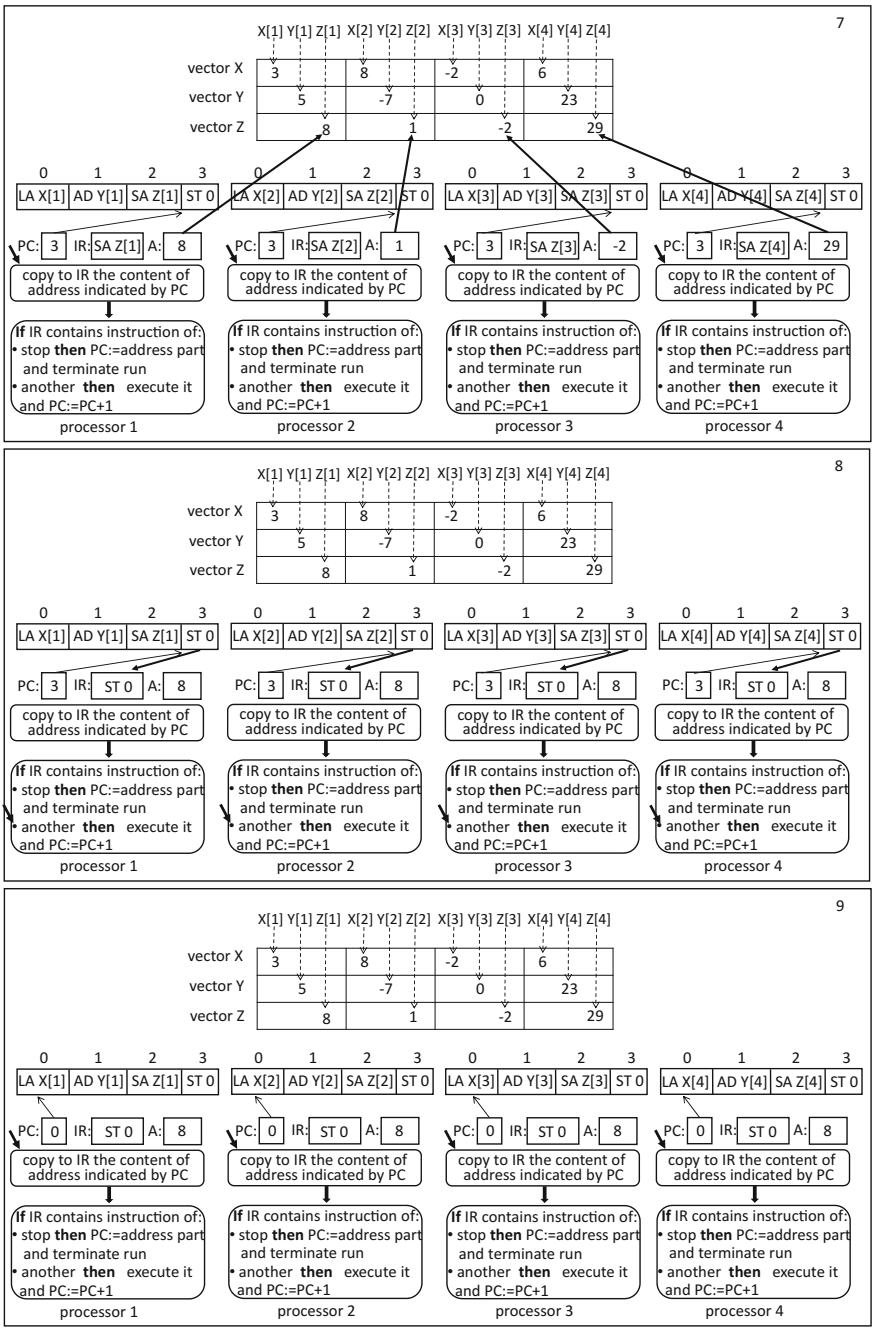
(continued)

Table 1.8 (continued)



(continued)

Table 1.8 (continued)



1.7 Some Classifications of Computer Systems

Before we pass on, to presentation of fundamental features and functions of distributed systems, let us take a look at possible types of computer systems depicted in the following diagram (Fig. 1.10).

The reader will easily ascribe exemplary computer systems outlined in this chapter to some types shown in Fig. 1.10.

A classification based on different principle (i.e. on multiplicity of instruction streams and data streams) is the the so-called Flynn's taxonomy (Flynn 1972):

- **SISD** (**S**ingle **I**nstruction [stream] **S**ingle **D**ata [stream])—traditional computers with one instruction stream
- **SIMD** (**S**ingle **I**nstruction [stream] **M**ultiple **D**ata [stream])—systems with one nstruction stream and more than one stream of data
- **MISD** (**M**ultiple **I**nstruction [stream] **S**ingle **D**ata [stream])—systems with more than one instruction stream and one data stream (do not exist)
- **MIMD** (**M**ultiple **I**nstruction [stream] **M**ultiple **D**ata [stream])—systems with more than one instruction and more than one data stream

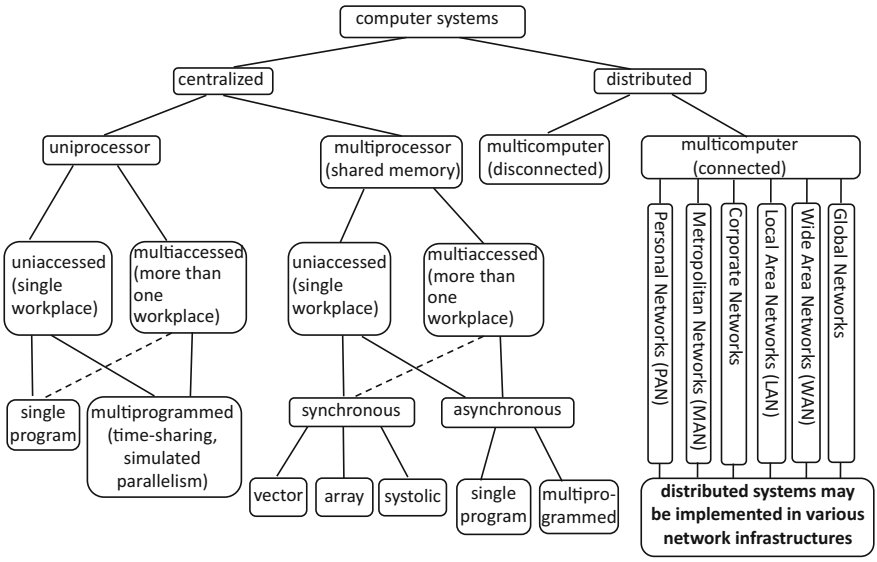


Fig. 1.10 Outline of possible types of computer systems

References

Carriero, N., & Gelernter, D. (1989). Linda in context. *Communication of the ACM, 32*(4), 444–458.

Carriero, N., Gelernter, D., & Leichter, J. (1986). *Distributed data structures in linda, symposium on principles of programming languages ACM*. Proc: ACM.

Dijkstra, E. W. (1965). *Cooperating sequential processes*. Eindhoven, the Netherlands: Technological University.

Dijkstra, E. W. (1968). Cooperating sequential processes. In F. Genuys (Ed.), *The origin of concurrent programming* (pp. 43–112). New York: Academic Press.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, *C–21*(9), 948–960.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM, 21*(8), 666–677.

Hoare, C. A. R. (1985). *Communicating sequential processes*. London: Prentice-Hall International.

Karp, R. M., & Ramachandran, V. (1990). Parallel algorithms for shared-memory machines. In J. Van Leeuwen (Ed.), *Handbook of theoretical computer science* (Vol. A). Elsevier.

Kung, H. T., & Leiserson, C. E. (1979). Algorithms for VLSI processor arrays. In C. Mead, L. Conway (Eds.), *Introduction to VLSI systems.* Addison-Wesley.

Petkov, N. (1992). *Systolic parallel processing*. North Holland Publishing Co.

von Neumann, J. (1945). First draft of a report on the EDVAC.

Chapter 2
Distributed Systems—Objectives, Features, Applications

2.1 What Systems Do We Consider as Distributed?

No generally admitted definition of distributed computer system exists. Before some understandings of this notion encountered in the professional literature will be outlined in this chapter, let us turn attention to its sources. As the first distributed network-based system, the **SAGE** (***S**emi **A**utomatic **G**round **E**nvironment*, USA, 50. of XX century) is recognized. It consisted of some tens of computers (some of them of several hundred of tons!), radars, missile launchers and other military installations, connected by telephone cables and dispersed over territory of USA and Canada. A successive system enjoying distributed features was a network **ARPANET** (***A**dvanced **R**esearch **P**rojects **A**gency **Net**work*, late 60. and 70. of XX century), initiated by the ARPA agency of Defence Departments USA. At first it comprised several academic computers connected into a network, then a fast increase of their number took place. In this network, radio and satelite links have been used for the military purpose of USA and Canada—similarly to the SAGE, but made available also for some academic institutions. It was the first large system, in which the so called packet switching (in contrast to circuit switching—as at that time in telephone systems) were applied as well as the popular protocols FTP and TCP/IP. On the turn of the 80. and 90. of XX century the **ARPANET** was partitioned into a military and civilian part (academic), named then **INTERNET**. The civilian part was gradually becoming a commercial system and in effect of fast development, reached the worldwide range. Thus, INTERNET grew out of ARPANET and took over from its military-academic prototype the basic ideas and techniques. The ***D**efence **A**dvanced **R**esearch **P**rojects **A**gency* (**DARPA**) is until now a mighty organization, launching various, not only military technological projects. To history of large computer networks, having some features of distributed system, belongs also **BITNET** (USA, developed and in operation in 1981–2000). It was an academic network, stretched over a number of countries for the purpose of electronic mail, remote conversation, access to remote scientific archives, etc., as

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_2

well as **EARN** (***E**uropean **A**cademic and **R**esearch **N**etwork*, 1984), a network linking academic institutions in Europe, connected with other networks like BITNET and INTERNET. The Polish branch of the EARN network, called **PLEARN**, has been connected to EARN in 1990 (closed in 2000) in Informatics Center of Warsaw University but its beginning was in operation in mid 1980. Then the Research and Academic Computer Network **NASK** was initiated in 1991 at University of Warsaw by a team of physicists, which played essential role in connecting Poland to INTERNET. Now, NASK has a status of a research institute.

Above mentioned computer networks are global and exhibit some features of distributed systems: the lack of common (shared) memory and clocks. Communication between computers proceeds through data transmission channels. These networks offer some services specific for a distributed software. On the other hand, even groups of terminals, like workstations instaled in many places, often remotely from one another, e.g. in different localities and connected to one main computer (e.g. mainframe) are sometimes thought of as distributed systems. The term „distributed system" will be used in a narrower meaning: not only as a network with a communication software making possible access to common resources, but as a work environment possessing certain properties mentioned further in this chapter. Closer to this notion are the so-called corporate networks (INTRANET), whose computers, apart from communication capability, are endowed with distributed software, devised specially for tasks typical for activity of a given corporation.

Some informal definitions of distributed system emphasising various features, dependently on different perspectives (users of various applications, programmers, designers of programming languages or operating systems, computer architecture, etc.) may be expressed as follows:

- collection of autonomous computers cooperating so that to reach a common aims;
- multicomputer system, where all computers work over the same or similar problem(s);
- set of independent computers linked by a network and managed by a distributed software, that offers various facilities to users;
- a system of independent computers that makes a user's impression of work on a single computer;
- multicomputer system—a set of autonomous computers (possibly of diverse architecture) interconnected by a network of transmission channels and equipped by distributed operating system;
- set of distributed processes and their hardware environment (computers, transmission channels, switches, routers, etc.). Each of these processes may be a set of sequential processes (termed sometimes as threads), acting concurrently on several or one procesor—in a time sharing mode;
- an informatics system where:

– arbitrary number of processes may run simultaneously; the system is modular and open to extension: its components may be attached and removed;

– communication takes place by message passing through transmission channels, not by common memory, which is absent;
– computer clocks are independent with various frequencies, but can be synchronized;
– operating system manages the network of connected computers; some parts of the operating system may be located on various computers; instants of dispatch and reception of a message are separated by unpredictable time period—not possible instantaneous (timeless) data transmission;

Basic difference between centralized and distributed systems

Centralized: processes in the system intercommunicate (interchange data) through shared physical memory space (Fig. 2.1). Hence fast communication. However, multiprocessor computers with such memory space, accessible also to eventually attached workstations, require quite sophisticated switchgear (see Sect. 2.4) to handle complex structure („topology") of wiring, which connects its processors to the memory. Access time to such memory depends on the distance—in the sense of this topology. This is so, because initiations of the access requires passing of the signal through a number of switches between levels of the wiring. This significantly limits a number of processors (working units) in such systems. Their important feature: all cooperating units have the current information about the system global state (content of the shared memory), which is not the case with systems based on message passing.

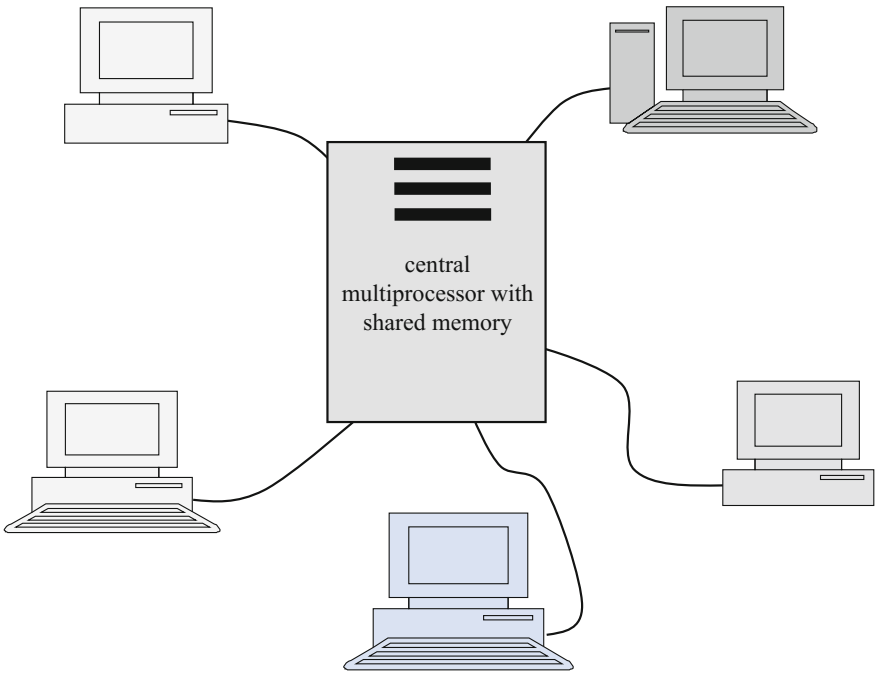


Fig. 2.1 Exemplary structure of a centralized system with terminals and/or workstations

Distributed: neither shared memory nor shared clock exists. A process in a node of the network may communicate with processes in other nodes only by sending data in packets through transmission channels (cables, radio links, waveguides, etc. handled by appropriate devices like transmitters, receivers, amplifiers, etc.). For correct mutual interpretation („understanding") of messages travelling between senders and receivers, the communication protocols are applied. A protocol is a set of rules of data coding (by sender) and decoding (by receiver). In accordance with the rules, the sender encodes (translates) a message into a certain „mutually comprehensible" form and the receiver decodes it for its usage. Such form is a sort of a formal language, whose syntax (grammar) and semantics („mutual understanding") represent the rules. Therefore communication in distributed systems is much slower than in centralized ones. But number of working units (network nodes, like computers, of various architecture and purpose) is practically unlimited. The activity of programs is managed by a software (distributed operating system), whose fragments are located in servers and users' computers (Fig. 2.2).

Such a software, when fulfils requirements after-mentioned in Sect. 2.3, makes communication network a medium for distributed system, i.e. for environment that allows to use resources located outside user's computer in a way not essentially different from using the local ones. Obviously, the same network may become an infrastructure for various distributed systems.

Commonly known corporate information systems are distributed systems for particular applications like computer system of a network of supermarkets, banks and cashpoints, insurance companies, booking of air tickets, etc.

Some examples of operating systems (or their kernels) that make a networked multicomputer to be a distributed system for general purpose, are the following:

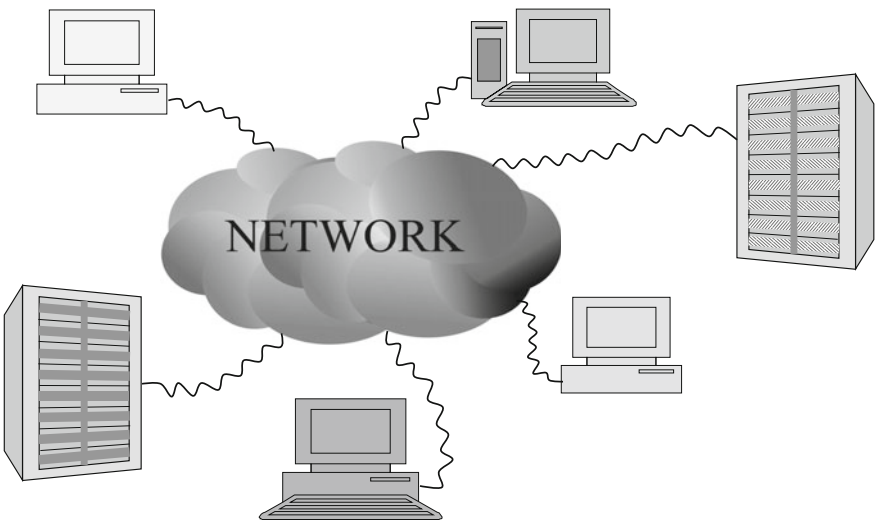


Fig. 2.2 Exemplary structure of a distributed system

- UNIX 4BSD (*Berkeley Software Distribution*, USA end of 1980 and successive years—a series of operating systems made in Berkeley University in California, originating from the UNIX BSD devised in 1978). They are extensions of UNIX (devised in the 70. of XX century in AT&T's Bell Labs.) and belong to its widespread family of operating systems. They were endowed with mechanisms of interprocess communication in computer networks, like TCP/IP protocols, socket interfaces, group communication, Remote Procedure Call, and a number of improvements concerning efficiency, extendibility and reliability. Systems of this series enjoy most of distributed systems' features and became a design pattern for respective projects.
- Chorus (France, INRIA, late 70. of XXcentury); the so-called micro-kernel for distributed operating system, afterwards developed by Chorus Systèmes Co.
- Mach (Carnegie-Mellon Univ. USA, 1980–1990); a kernel of distributed operating system, applied in several commercial systems;
- Amoeba (Vrije Universiteit Amsterdam, 1980-te-1990); distributed operating system for which the Python programming language was devised;
- CLOUDS (Georgia Institute of Technology, USA, 80. of XX century);
- DCE (Distributed Computing Environment, USA, a number of consortiums, early 1990);

2.2 Most Important Objectives of Distributed Systems

2.2.1 Economy

Distribution gives better cost-effective results than a single main (also powerful mainframe) central computer. With a number of cheap computers connected by a network, one may obtain computing power of a mainframe or higher, but the total cost of the system significantly lower. If a powerful computer is connected to the network, the other computers may delegate to it particularly time consuming and exceeding their capabilities fragments of computation, whereas most of activity is being performed locally. Many computer centers, apart from large computers, install in their premises local networks, that connect workstations and clusters of servers, as well as systems of dedicated architectures, e.g. grid systems.

2.2.2 Increase of Computing Power

Sometimes possible only by parallel actions. That is, by execution of an algorithm concurrently on multiple processors, one can overcome the physical barrier of a sequential procesor's power. Even if this barrier is far away, it is advisable (sometimes necessary) to disperse the computation over a number of processes in

case of handling with big data, when none large computer can cope this task e.g. due to bottlenecks in memory access. Examples of distributed applications projects, for which dedicated systems for big data processing were constructed are:

International project SETI@home—(***S****earch for* ***E****xtra-****T****errestrial* ***I****ntelligence,* University of California, Berkeley, USA) allows everybody who has a computer with Internet, to download a program that analyses data collected by largest radiotelescopes. The program fetches successive data portions, analyses them when the computer is not engaged with ordinary everyday activity, or underloaded is its processor. The results are returned to the head office of the project. Average effective speed of these data processing (now—2017) amounts to over 260 TeraFLOPS (one TeraFlop: 10^{12} ***FL****oating point* ***O****perations* ***P****er* ***S****econd*). Another public program of similar purpose is the Planet Hunters (PH). So far, no signals from extraterrestial civilization have been discovered. But amateur astronomers have found (October 2012) a planet PH1 that rotates around two stars (in the system of 4 stars). Now several milion of personal computers are participating in the project.

International project Folding@home—the so-called Blue Gene project. It allows to download a program for analysis of proteins and simulates their folding and misfolding (the latter causes some diseases). The program proceeds similarly to SETI@home: analyses imported data and returns results. Average effective speed of data processing amounts to over 1200 TeraFLOPS and increases as new volunteers join the project. Some sources announce that it reached 5000 TeraFLOPS. The IBM Blue Gene supercomputer used in the project is composed of many cheap processors (applied in personal computers), with access to a large memory. It has reached more that 1000 TeraFLOPS and is used also for other purposes, e.g. to simulation of human brain cortex activity.

Many projects in astronomy, mathematics, physics, chemistry, genetics and other research and application domains, are being elaborated by millions of people working over a joint problem in such remote manner. Some other examples of such distributed projects are: **MilkyWay@home** (design of exact 3D map of our galaxy), **Einstein@home** (search for gravitational waves), **ABC@home** (verification of a certain conjecture in number theory), **QMC@home** (investigation of molecule stuctures, by means of quantum chemistry methods), **DNA@home** (discovery of gene regulators). There exist hundreds of computational projects („dispersed over homes") that use special software like the mentioned above.

An extensive lecture course on distributed algorithms may be found e.g. in Lynch (1996).

2.2.3 Internal Distribution

The only reasonable solution for some applications—examples: a network of supermarkets (everyone should keep in its local computer its accounting of stock, instead to overload the central data base of the company). Most of updates is

performed locally, but sometimes the company management needs a total information about goods in stocks of all of their supermarkets at a time. Another example: a collective work or game of a number of people, video conferences, e-learning or systems using multimedia data like music, painting, photography, etc.

2.2.4 Reliability

In case of failure of one or several computers (hardware or software), the distributed system can assure correct work of the whole. A user, independently of his/her whereabouts, does not know that a failure occurred, because his/her processes run correctly. Distributed operation system, when detects a source of the failure, conveys activities of the faulty machine to those being in full order. For instance, establishes alternative routing in the network and recovers data from the faulty machine. This is called *a fault-tolerance*. The failure may be caused by disappearance of power supply, mechanical damage, viruses, etc. If this is not a total breakdown, the system can remove the faulty entity by means of some error recovery procedures and make a self-reconfiguration. Apart from fault tolerance, the distributed system should assure the so-called data consistency. For some organizations (like banks, flight control), reliability of the system is a paramount necessity. It is worth to mention that fundamental aims of the aforetime military devices based on widespread networks SAGE and ARPANET, was reliability: destruction of one installation, would not paralyse activity of the whole. Enhancement of reliability is also achieved by doubling (at least) certain components of hardware, software or user's programs and data.

2.2.5 Independence from Changes of Environment

Replacement of hardware components (servers, switches of packets, routers, etc.) for more effective as well as adding of newer ones and removing of older, does not entail necessity of modifying the software structure. Also, a change of requirements, e.g. for increasing a service offer should not destroy basic principles of hardware and software construction, after adding of new components of different type. To this end, a modular structure of the software is needed. This makes easier its installation, exploitation and adding and removal of the modules.

2.2.6 *Flexibility*

Possibility to use various computers for a job, depending on their current workload and their computing capability. Example: if in a country A is night, while in B is day, the computers from B may delegate some jobs to the computers from A.

Above mentioned objectives do not exhaust all requirements for distributed systems. Particular systems are being devised for specified classes of tasks and needs. Specific aims the systems should achieve, are determined by these tasks and needs. Examples: widespread public access to informations (www pages), e-mail, public communicators, social portals, etc.

2.3 Main Features of Distributed Systems

The distributed system should possess properties that ensure attainment of its objectives by obeying assumptions on its project. The objectives are specific for tasks the system has been intended for, but also general aims, listed in Sect. 2.2. Typical features, following from these general assumptions are:

- **sharing resources**—a number of users may use the same resource, like printers, files, procedures, databases, etc.
- **openness**—susceptibility to extension: enlargement of the system (hardware and software) with retaining of hitherto existing solutions;
- **concurrency**—capability of performing many jobs or processes simultaneously;
- **communication**—interprocess data transfer by data transmission channels offered by the network;
- **lack of global time**—various clock frequency of different computers, different time indication of these clocks, different speed of processors' activity;
- **fault tolerance**—system's capability to remain active if an error of hardware or software occurs (e.g. by maintenance of redundant hardware or taking over the role of crashed device by a certain components working correctly);
- **transparency**—making possible the user to perceive the system as a whole, not its separate components, by hiding technical details specific to cooperation of multiple computers;
- **scalability**—preserving performance of the system after increasing its scale (e.g. number of processes, computers, etc.).

So, some of these features pertain to advancement of processes, whereas some others—to give the user impression of exclusive usage of resources. The former (let us call them „dynamic") will be illustrated in the successive chapters, by examples of activity of mechanisms for management of concurrency, communication, failures, time, remote services and distributed shared memory. This illustration will consist in presentation of exemplary processes, like in Chap. 1—as sequences of

states of a system during its activity. This chapter continues presentation of features regarding usage of distributed systems—their pragmatics.

2.3.1 Resource Sharing

Every program acting in a computer system uses resources available from its environment:

- *hardware*: processors, memory, input/output devices, other peripheral equipment, communication channels, sensors, clocks, devices whose activity is computer controlled like radiotelescopes mentioned above.
- *software*: application programs, procedures, methods, controllers, user files, www pages, data bases, libraries and any data sets provided for the user's needs.
- *time*: physical or logical magnitude (Chap. 4).

Resource management—assignation to processes, synchronization of their usage, protection, recovery if lost, scheduling of time and other activities of this management—is accomplished by operating system, whose parts are located in servers, the managers of resources. The resources are categorized into types. To every type, specific methods are being applied, but common for entire network are:

– the manner (schema) of resource naming for each type;
– enabled access to resources from each computer in the network—in accordance with defined accessibility rules; an access to a resource may change its state (informally: a set of instantaneous values of its components like memory content, indications of clocks, etc.);
– mapping of resource names into communication addresses;
– coordination of concurrent access to resources—to ensure their consistency and coherence.

For each resource type there exists its *manager*. The users are accessing resources by communicating with the managers as depicted in Fig. 2.3.

More than one manager may be located on one server as depicted in Fig. 2.4.

Keeping resources on servers specially intended for this purpose is one of two methods of resource sharing. This method, called a *client-server* model, assumes that the server, the resource holder, decides to immediately assign it to the user on his/her request, or to postpone, or to reject. Thus, a server plays a part of decision-maker and provider of services. Figure 2.4 shows that a server may itself become a user of resources located on other servers, that is may become a client, as well as may hold various resource managers. Its typical services are the following:

– access to shared printers
– access to shared big data sets
– e-mail
– application programs

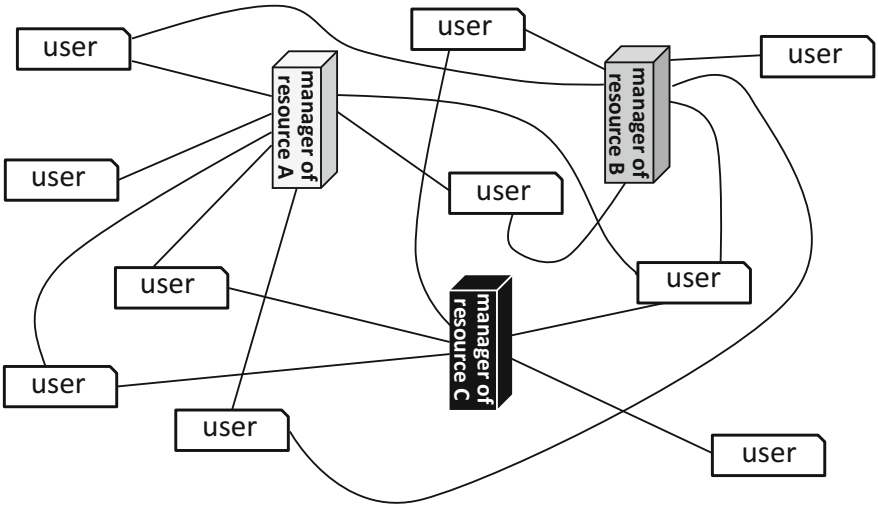


Fig. 2.3 Managers of resources A, B, C and users

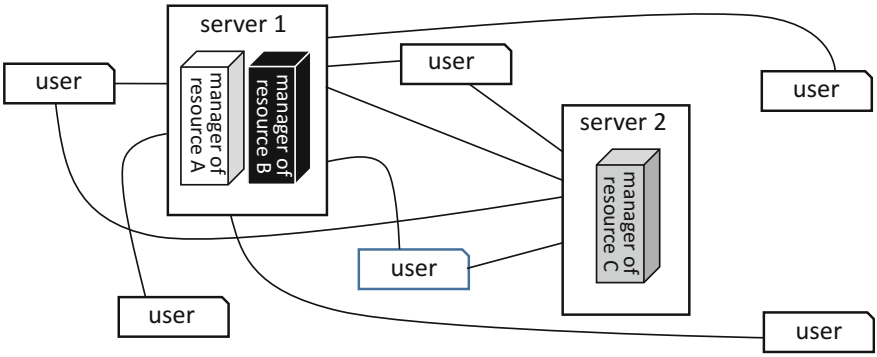


Fig. 2.4 Managers of resources A, B located on server 1 and resource C—on server 2

- www pages
- social portals
- distributed data bases
- file and directory services
- time service
- naming services
- remote procedure call or remote method invocation
- transactions
- synchronization of clocks
- distributed shared memory.

Some types of resources are located on the client's computers, not on servers, to achieve better performance of the former, e.g. local memory (RAM, ROM, cache, disk), some interfaces and controllers, etc. The second resource sharing method, called an *object model*, assumes that the resource is treated as a jointly used object —in the meaning of object programming paradigm. In this approach, the resource is an instance of an element of a class (a set of objects of the same type), with algorithms (procedures or methods) that operate on this resource. A manager of resource is then a collection of operations performed on this resource. There exist systems supporting interprocess communication (thus, also resource sharing), endowed with various systems of object programming. An example is **CORBA** (*Common Object Request Broker Architecture*) that offers communication mechanism of objects residing in computers of different hardware and software architectures, in particular, those endowed with various programming languages and their compilers. The CORBA mechanism offers the so-called *Interface Definition Language* (*IDL*) for defining interfaces for communication between objects. The interfaces are being translated from the IDL (by its compiler) into a programming language used by the programmer for creating classes of objects. Some programming systems, like Java, C++, Python and several others are outfitted with such compilers that convert interfaces (translated by the IDL's compiler) into a form used by communication instructions like *send* and *receive*. A mechanism of *Remote Procedure Call* (Chap. 6) is based on similar idea, but limited to creating interfaces for calling procedures whose bodies are located in other computers (servers). Still another example facilitating distributed objects communication is the CLOUDS operating system (see Sect. 2.1).

2.3.2 Openness

This is capability to easy appending of hardware and software to the system. The easiness means no need of change the system architecture in this case. The user may do it himself, without ordering this job to a professional service. The openness is being achieved by:

- ensuring independence of desired system behaviour, from its architecture (hardware and software); to this end, standard controllers are used and—in case of programming languages—intermediate languages and their compilers or interpreters (e.g. virtual machine of Java);
- applying unified standardized communication mechanisms between processes (like communication protocols);
- using widely offered standard interfaces of access to common resources like libraries, controllers, application programs, etc.

Favourable for openness is modular construction of the system, both hardware and software. Notice that the modularity allows for replacement of system

components for more modern ones of the same functionality, as well as change of universal components (commonly available in the market), into specially devised for the system—in order to enhance its performance. The open, modular architecture makes easier its installation and putting into practice: for instance, appending components when a part of the system has already been tested and is in operation.

2.3.3 Transparency

Hiding (from the user) that various types of computers and devices are in the system: giving the user impression of exclusiveness of using the system. This is a consequence of a definition of distributed system, where the impression of working with a single computer has been stressed. This impression remains, when the system evolves during its usage (when adding and removing or replacing components, etc.). It does not affect the user's work, nor his perception of the system as a whole—as a tool in exclusive usage. In general, the user is not aware of localization of resources, like data stores, processors performing processes, etc. Obviously, in case of some of them like shared printers or projectors, located in remote rooms, etc., such knowledge is needed.

The International Standard Organization (ISO) defines the following types of transparency in distributed systems:

- **of access**—enables getting both local and remote data by means of similar operations;
- **of location**—enables access to data without knowledge of their site in the system;
- **of concurrency**—enables undisturbed execution of many processes in parallel with shared data;
- **of multiplication**—enables using many copies of data—to enhance reliability and performance; sometimes this type of transparency is necessary—as in the Distributed Shared Memory mechanism (Chap. 8);
- **of faults**—enables hiding some breakdowns; the programs continue execution despite breakdowns of hardware or software; this is a consequence of reliability as one of distributed system's objectives;
- **of migration**—enables relocating data over the system with no influence on the user's work, or application programs, for instance naming of files is preserved;
- **of performance**—enables reconfiguration of the system to enhance its efficiency when its workload is changed; this is a consequence of flexibility as one of distributed system's objectives;
- **of scaling**—enables modification of the system size without changing its structure and application algorithms.

2.3.4 Scalability

The smallest scale: two connected computers or two workstations and server(s), larger—local network (e.g. several hundred workstations or personal computers), next—a system composed of a number of connected local networks (e.g. wide area network), next—a number of connected latter systems (e.g. intranets), up to a global network (e.g. internet). Scalability makes a certain alleviation of possible consequences of openness. It aims at not worsening of efficiency after adding new components to the system. Enlargement of the scale retains the system software unchanged. The system adjusts its usage to the new scale. Modified are parameters of attached components only, e.g. size of distributed database after adding larger memory. A modification of scale takes place with change of number of users, size of data, as well as in effect of combining smaller networks into larger structures, e.g. local networks into corporate ones, etc. Obviously, attainment of high degree of scalability depends on the current potential of network technology.

Features like concurrency, communication, lack of global time, fault tolerance, worth of more thorough consideration, are subject matter of successive chapters.

2.4 Exemplary Memory Connection Structures in Centralized Multiprocessors

Since distributed multicomputer systems often take advantage of services delivered by servers being large multiprocessors with shared memory (sometimes of a powerful mainframe supercomputer), expediently will be to sketch main types of structures connecting processors and memories: non-hierarchical (Figs. 2.5, 2.6 and 2.7) and hierarchical Figs. 2.8 and 2.9.

In the type depicted in Fig. 2.5, each processor can access the shared memory with equal time (delay). Simple construction but heavily exposed to bottlenecks with large number of processors.

In the crossbar switch with n processors and m memory banks, the crossbar requires $n \times m$ 4-fold switches (4 connection points of each, $n = m = 8$ in Fig. 2.6). Expensive with large number of processors and memory banks, so, applied in architectures with rather small number of these units, or as a components of more complicated connection structures.

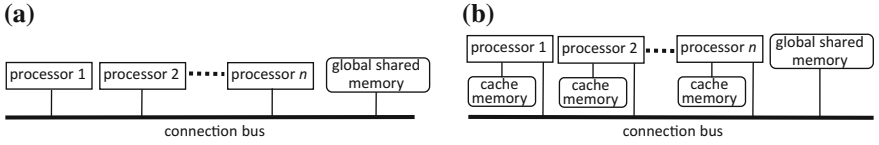


Fig. 2.5 Non-hierarchical connection with single homogeneous global memory space **a** Unibus with one shared physical memory **b** Unibus with one shared physical memory with caches

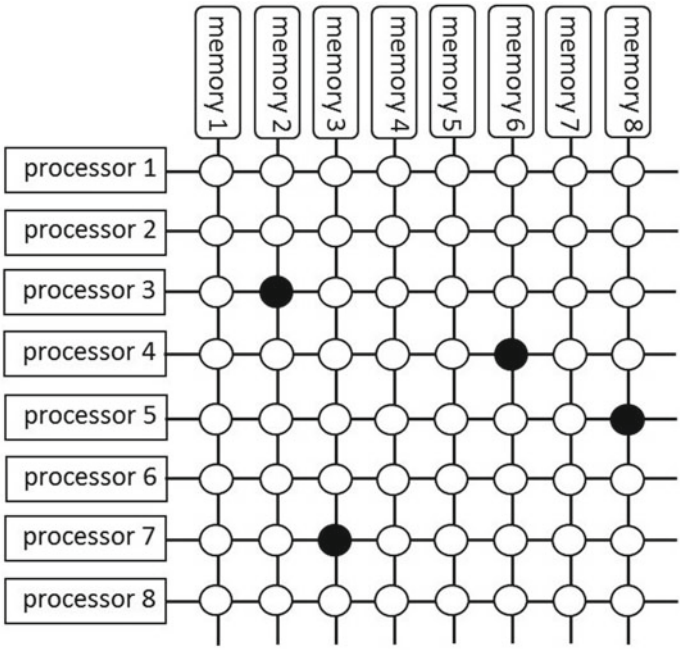


Fig. 2.6 Crossbar switch: non-hierarchical multibus connection with heterogenous (partitioned into „banks") global memory space. Circles represent *4-fold* switches (4 connection points) between processors and memory banks (white—switched off, black—switched on)

In the logarithmic switch (Fig. 2.7) with n processors and n memory banks, this wiring structure requires $\log_2 n$ switching degrees; there are $n/2$ switches on every switching degree, thus the omega-network requires $(n/2)\ \log_2 n$ 4-fold switches—substantial decrease in cost and action time in comparison with the crossbar.

In the binary tree structure with n processors with local memory each (Fig. 2.8), there are $n - 2$ *3-fold* switches and one *2-fold* switch;

A generalization of the binary tree connection structure is the *cluster structure*, exemplified in Fig. 2.9.

In a certain terminology, the non-hierarchical connection structures are refered to as ***U**niform **M**emory **A**ccess* (**UMA**) exemplified by Figs. 2.5, 2.6 and 2.7, whereas hierarchical—as ***N**on **U**niform **M**emory **A**ccess* (**NUMA**) exemplified by Figs. 2.8 and 2.9. However:

Attention: The term UMA is ambiguous: in some technical areas it is used as abbreviation of „Unified Memory Architecture", a synonym of "Integrated Graphics", in some others—as abbreviation of „Unlicensed Mobile Access".

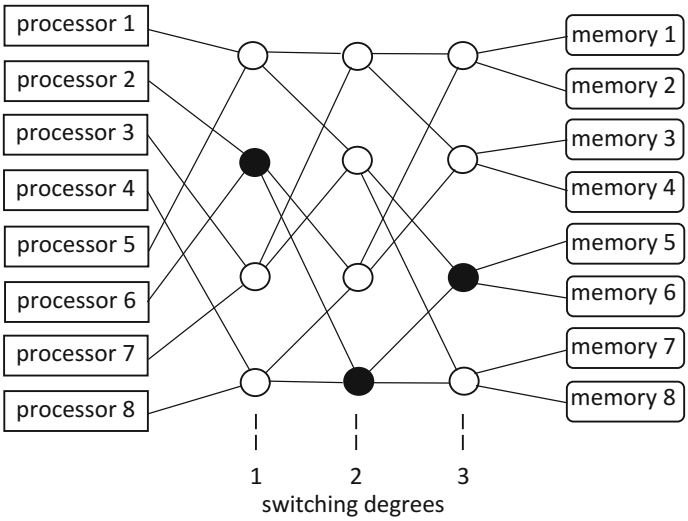


Fig. 2.7 Logarithmic switch, the so-called *omega-network*: non-hierarchical multibus connection with heterogenous (partitioned into banks) global memory space and *4-fold* switches (white switched off, black switched on, so, the route, e.g. from procesor 6 to memory bank 5 has been established by a communication protocol)

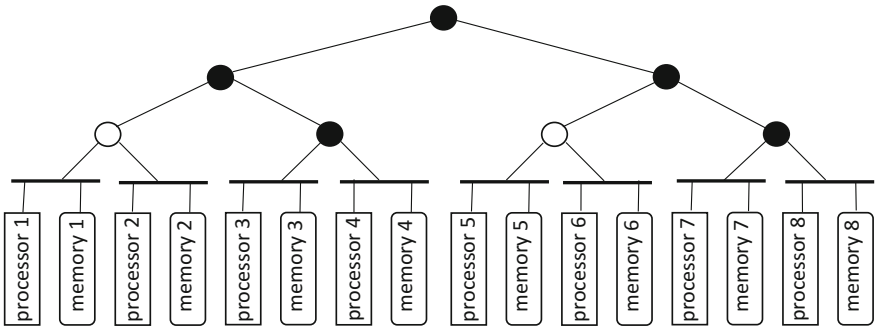


Fig. 2.8 Binary tree—hierarchical multibus connection structure with local memory of processors accessible from other processors; the route e.g. from procesor 3 to memory 8

The above examples show that in the **UMA** type connections the data transfer time (latency) is roughly equal for every pair (processor, memory unit). But number of components is substantially limited. In contrast, in the **NUMA** type, the time of transfer essentially depends on the distance—in the sense of the wiring structure (switchgear „topology")—between a processor and required memory unit. The data transfer time increases with number of switches the data must pass through. But number of components may be larger. The processors in such architectures communicate between themselves by shared memory space. Another kind of

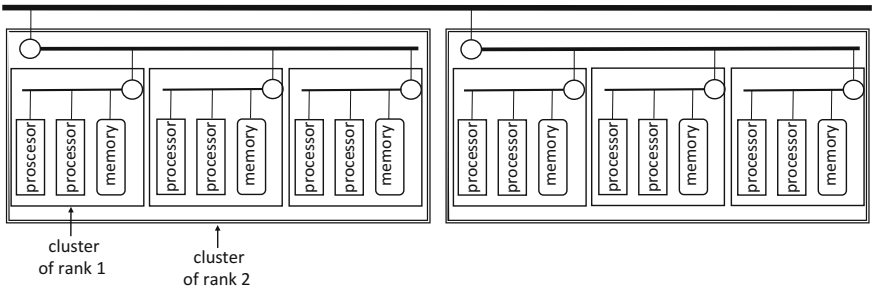


Fig. 2.9 Heterogeneous multibus interconnection of clusters; every cluster's switch connects it to a bus of cluster on the next, higher level, that is of the one rank greater

interprocessor or multicomputer communication in specialized architectures, also within one large central computer, is by means of message passing through networks of various structures, not by common memory space. This kind is sometimes referred to as **NO R**emote **M**emory **A**ccess (**NORMA**), the term somewhat misleading (for instance, the multiprocessor in Fig. 2.5 could hardly be named a „**R**emote **M**emory **A**ccess" architecture, thus as of the **NORMA** kind!). There are a number of such structures („topologies"), usually intended for particular domains of application. Example is a topology of cube and hypercube, star, ring, grid, torus. Their detailed description does not belong to the subject of this book, but is easy available in many books and articles concerning various computer architectures.

Reference

Lynch, N.A. (1996). Distributed algorithms, Morgan Kaufmann Publishers, Inc.

Chapter 3
Concurrency

3.1 Concurrent Execution of Programs

Concurrency is the capability of running a number of processes simultaneously, that is existence of more than one process in the overlapping time periods. It can be real —realized by more than one processor, or simulated—on a single processor with time sharing for a number of processes (Table 1.2 in Chap. 1). Here, we will not distinguish these two technical ways of concurrency realization: their properties in principle („logically") are the same, though their implementations require different solution in hardware or software. The concurrent execution of programs is usually desired or indispensable. For instance when several users (clients) send requests to a server, which initiates respective processes to be run in overlapping time periods, i.e. starts them before others are terminated, as well as for faster execution of programs by executing their parts in many processors simultaneously. Problems with concurrency arise when the processes share resources or transfer data between one another and when some events in a process must precede some events in another process—if required by specification of the program. Event in a process is execution of the active element in a program, like machine instruction or a statement in a programming language, or reading/writing memory—depending on the generality level of considerations. Management of concurrency means coordination of processes, i.e. enforcing a desired ordering of their events. The main problem is synchronization of some activities, that is, assurance of mutual exclusion or determination of temporal order or eventual simultaneity of execution. This management is responsible also for handling with errors and with pathological situations, like deadlock and starvation. This chapter focuses on mutual exclusion in distributed systems, that is, assurance of exclusive access to resources—if necessary (e.g. when writing to a shared file) in the context of some transactions, as well as on the deadlock and starvation that may occur during execution. There are several methods of mutual exclusion realization: by means of „external force" for users' computers and „truly distributed"—solely by exchange of messages between them.

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_3

Two methods are presented in this chapter: with supervisory server and with a token ring. These without „external force", based on the so-called global timestamps, are postponed to Chap. 4, where issues of time will be examined.

Before coming to the mutual exclusion, let us mention some general features of transactions—a typical activity, where management of concurrency is indispensable. They are the following:

- „all or nothing"—a transaction is either committed, i.e. completes successfully with correct results, or is discarded (aborted) and returns to a state from before its initiation (unexisting transaction)—never is being performed only partially;
- isolation—other transactions have no access to the intermediate results (not ultimate);
- cohesion—structure of any data set (like data base), that is, a schema of interrelationship between subsets of this set, is not changed in effect of realization of a transaction;
- serial equivalence—the outcome of concurrent execution of not mutually excluding actions is the same as the outcome of their execution one by one (sequentially);
- exclusivity (indivisibility)—of writing to memory but not reading from memory —thus, the writing is performed in transaction protective sections.

(cf. the extensive book on transactional information systems (Weikum and Vossen 2002)).

Any transaction is being initiated by execution of the ***„open-transaction"*** action and terminated—by the ***„close-transaction"***. A transaction may be committed or aborted, depending on its success or failure. To this end, ***commit*** and ***abort*** actions are performed. For conciseness of exemplary sketch-procedures of bank transactions, from among these four only the ***abort*** will be explicitly used. It makes the transaction interrupt and return to a state from before its initiation. An entire transaction and its sub transactions will be marked by curly brackets. In successive examples, actions belonging to the bank transfer procedures and summation of bank accounts, are executed by a bank's servers and are initiated on the client's request. These actions are being performed in the global (physical) time (Chap. 4) depicted by the time axis directed downwards in the successive figures. The actions' positions on this axis represent their temporal interrelations, that is their succession in a transaction. Execution of bank transfers and summation of accounts are accomplished by sequences of actions presented by the examples. For exclusive access to resources, bank accounts in examples, apart from the mentioned ***abort***, the following operations will be used:

- ***lock*** Z, where Z is a name of resource protected against simultaneous access (reading/writing) by more than one process or a list of such resources. In the distributed (multicomputer) systems, it plays a part similar to operation of semaphore closing (*P* or *wait*) in centralized systems. The ***lock*** Z in a process, attempts entry into system state where resources Z can be accessed, i.e. not currently in use by another process. If impossible, the access to Z is delayed until a state, which enables this, when other processes complete use of Z.

This prevents access to these resources by any other process at a time, that is, serializes accesses to Z.

- ***unlock*** Z, where Z is as above. It plays a part similar to operation of semaphore opening (*V* or *signal*) in centralized systems. Its execution causes the process leave state where resources Z may be used at most by one process at a time, and permits access to them by a process in which the access has been delayed, e.g. the one which waited longest for Z.
- ***read*** Z, where Z is a name of a resource whose value is to be fetched from memory.
- ***write*** Z, where Z is a name of a resource whose value is to be stored in memory.

In the following pictorial examples explaining distributed mutual exclusion, these operations will be shown explicitly (in concrete implementations they are accomplished by protocols of resource management). So, portions of processes where they appear, will be illustrated on the grey framed background and called here *protective zones*. They play a part similar to critical sections (Chap. 1), but belong to tasks of resource management protocols, not user programs. What is more, the identifier of protected resource is explicitly used as the parameter of the ***lock*** operation—that is why the name, "protective zone" instead of "critical section" is used here. The exclusive access to resources Z require passing „request" and „release" messages to the manager of resources or to the processes where usage of Z must be exclusive. Thus, execution of these operations takes more time than that of *P* and *V*, which use a semaphore located in the memory common to all processes. Moreover: execution of the ***lock*** Z in a transaction, delays only access (read/write) to resources Z being currently in use by another transaction, but not other actions in this transaction.

Example 1: bank transfer transactions Figure 3.1 presents performing erroneously programmed bank transactions. Amount of balance of accounts B, C, D are respectively 50, 200, 100 EUR. The gentleman makes a bank transfer of 5 EUR from the account B to C and the lady—3 EUR from account D to C. After the transactions, the balance of account C is 205 EUR, whereas it should amount to 208 EUR. A *lost update* of account C occurred: the gentleman destroyed update of account C, made by the lady before. The conflict of updates will be eliminated if the bank transfers are closed into protective zones for C, which is shown in Fig. 3.2.

The protective zones in Fig. 3.2 are „coarse-grained". It suffices to block account C only, if accounts B and D are accessible to their owners but not to other clients in the system. In this case such zones may be made more „fine-grained", as depicted in Fig. 3.3.

Only the updates of account C must proceed one by one, i.e. sequentially. The bank transfers in Figs. 3.2 and 3.3 are, thus, accomplished partly concurrently. This exemplifies a property called a *sequential equivalence*: the outcome of concurrent execution (with observance of exclusive access to resources—if necassary) must be identical with that of sequential execution.

Transactions may be nested, as depicted in Fig. 3.4. The account owners make bank transfer to C1 and C2, the latter nested inside C1.

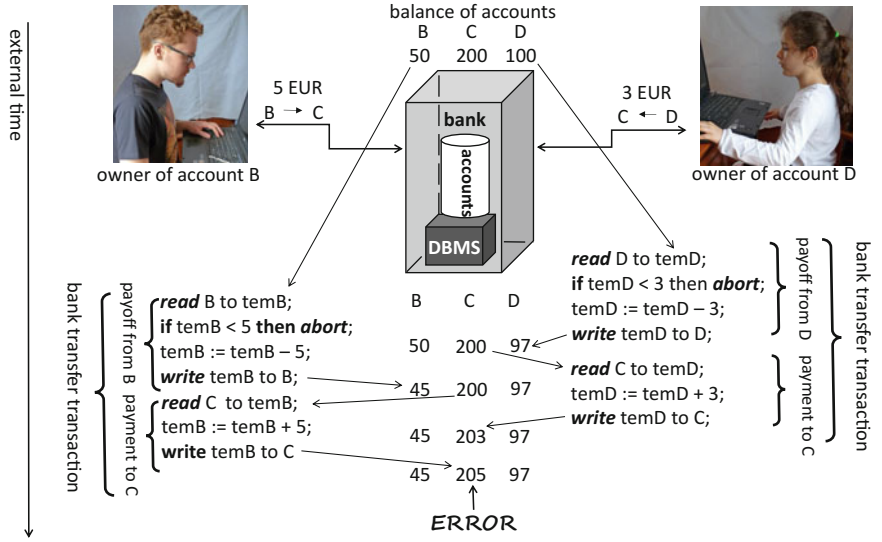


Fig. 3.1 No blocking—lost update of account C; temB and temD are variables for temporary values

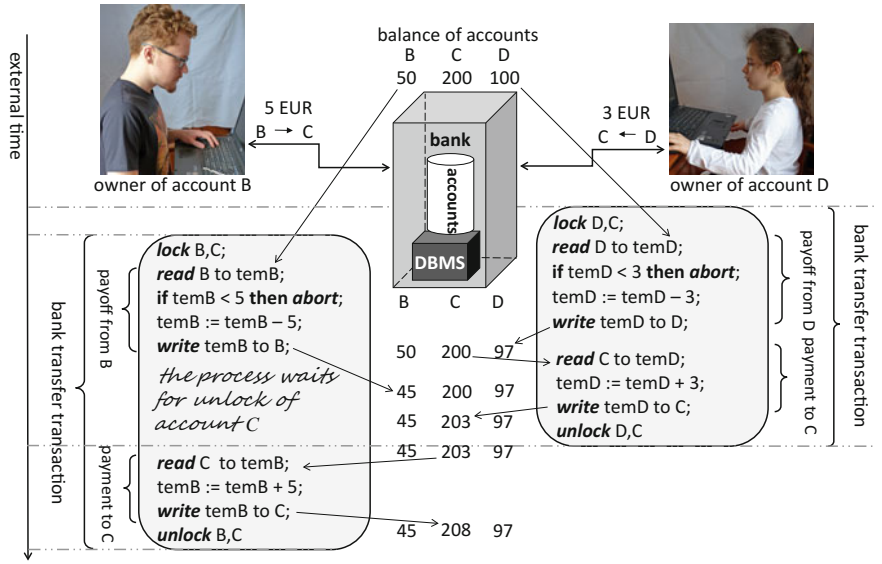


Fig. 3.2 Correct blocking; protective zones are in frames on grey background; accounts B and D are being concurrently read and written by their owners only

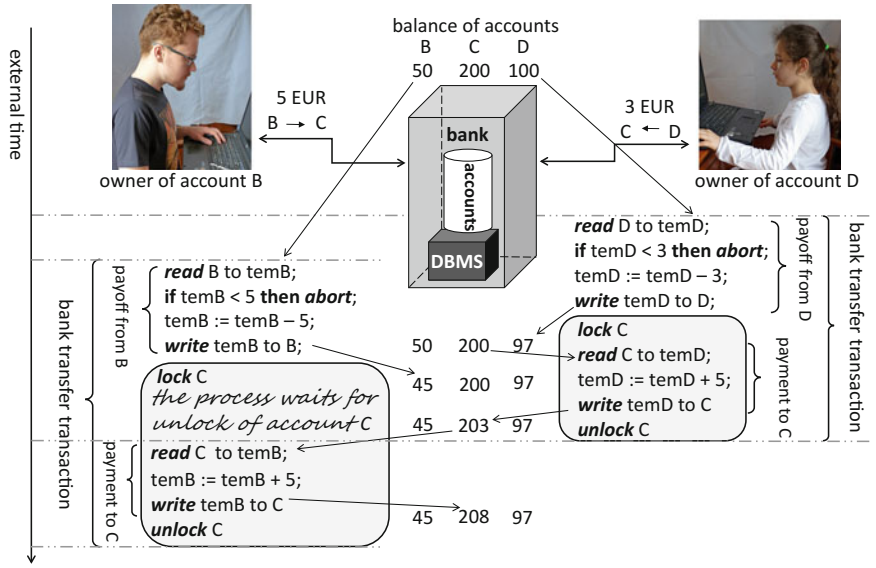


Fig. 3.3 Correct blocking with finer-grained zones protecting account C if access to B and D is restricted to their owners by resource management protocols

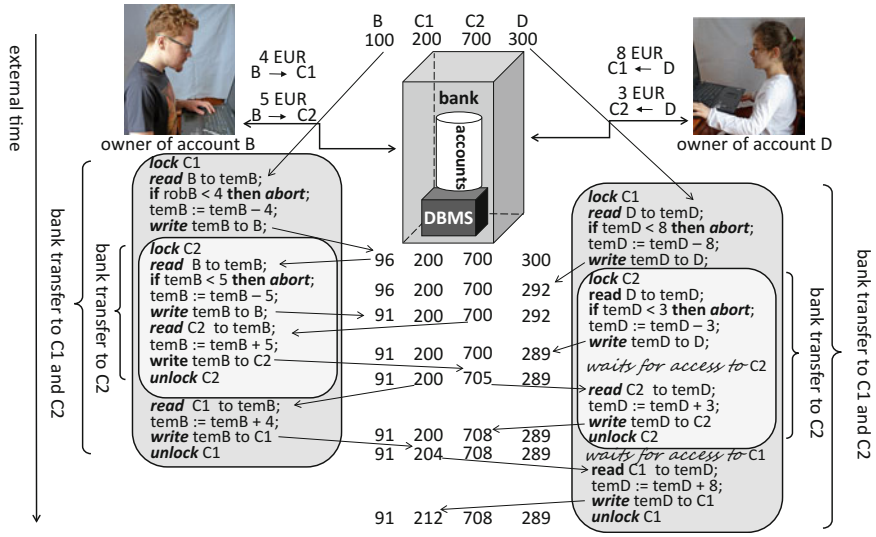


Fig. 3.4 Nested transactions

Nesting of transactions may be accomplished by two servers, as shown in Fig. 3.5. For instance, server 1 that handles account C1, conveys to server 2 the transfer of some amount of money to account C2 (and reciprocally).

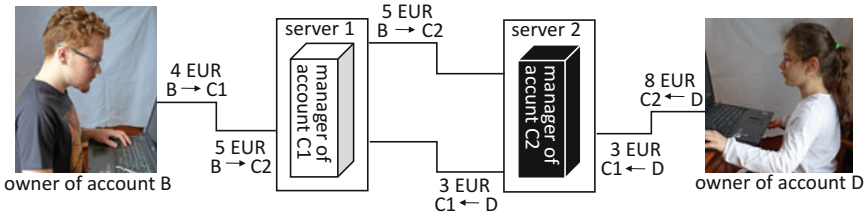


Fig. 3.5 Server 2 is a client for server 1 and conversely

However, nesting of transactions is fairly deadlock-prone, as shown in Fig. 3.6.

The ***lock*** Z entitles to access resources Z, only the process in which this action has been performed and no other process is currently using Z. Thus, it prevents access to Z for other processes, both to reading and writing. However, writing only may change state of Z. So, to preserve consistency of resources, if a process is writing something to Z, no other one may be reading from Z nor writing to Z at the time. Though processes may read resource concurrently, they should prevent from writing to it at the time. That is why before start of reading, delay of writing to Z must be ensured until completion of all readings of Z. To this end the action ***read-lock*** Z is introduced, which allows concurrent reading Z but prevents writing to Z when Z is in use, as well as the action ***read-unlock*** Z which cancels the former. Such actions are called *shared locking*. This mechanism fulfils the principle of exclusive writing and concurrent reading. Yet, if for applications where the reading only should be exclusive, but not the writing, the management system may offer actions „symmetric" to the former: ***write-lock***, ***write-unlock***. In general,

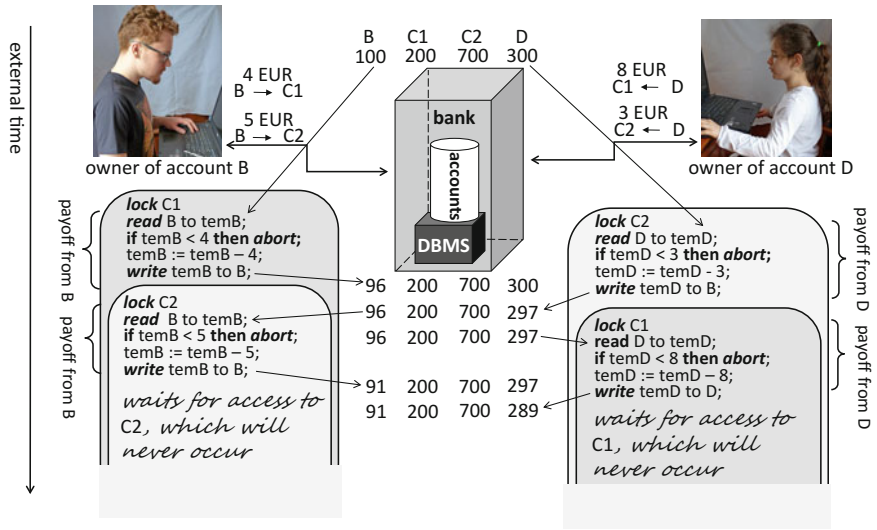


Fig. 3.6 Deadlock: endless mutual waiting for unlock of accounts

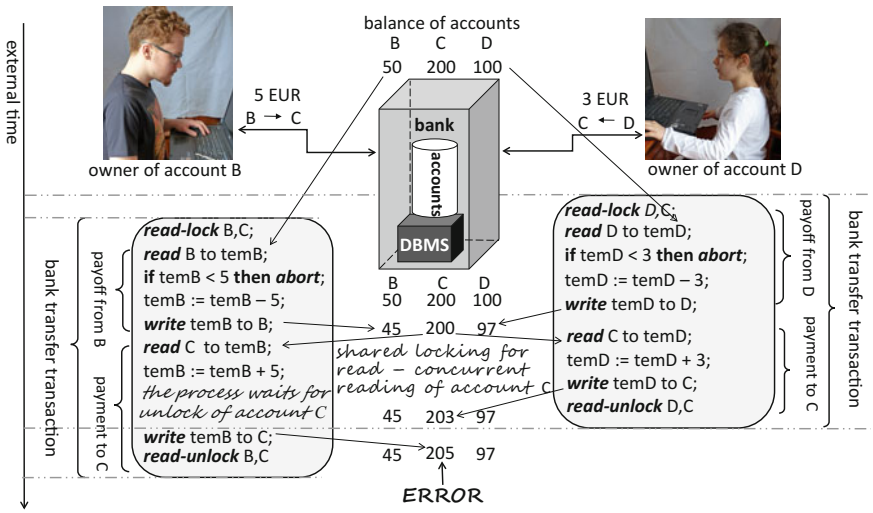


Fig. 3.7 The lady made shared locking of account C for read, which caused the lost update

management protocols may enforce various access strategies. In this way they may enhance degree of concurrency, but also chances of errors—when inattentively applied, as illustrated in Fig. 3.7.

> Obviously, the problem of concurrent reading/writing concerns not only bank transactions, but all transactions where such operations are applied, for instance in data bases. The principle of exclusive writing and concurrent reading, in abstract setting, is referred to as the readers/writers problem, originally described in (Courtois et al. 1971), then many times investigated in various versions (e.g. with priority to the readers or writers). This principle, embodied into access strategy, has been applied in our exemplary bank transactions.

Example 2: check-up of bank assets Another (than the lost update), result of incorrect programming of transaction, is the so-called the *incoherent data recovery*, exemplified in Fig. 3.8. A bank wants to know, from time to time, the total balance of all accounts, so makes their summation. During this summation, the clients make transactions—the lady makes a bank transfer from her account D, to the account C. The total balance of all accounts after her bank transfer and after the summation is 1497 EUR instead of 1500 EUR. Incorrect outcome!

The correct outcome is obtained if the summation and the bank transfer are closed into the protective zones, as in Fig. 3.9.

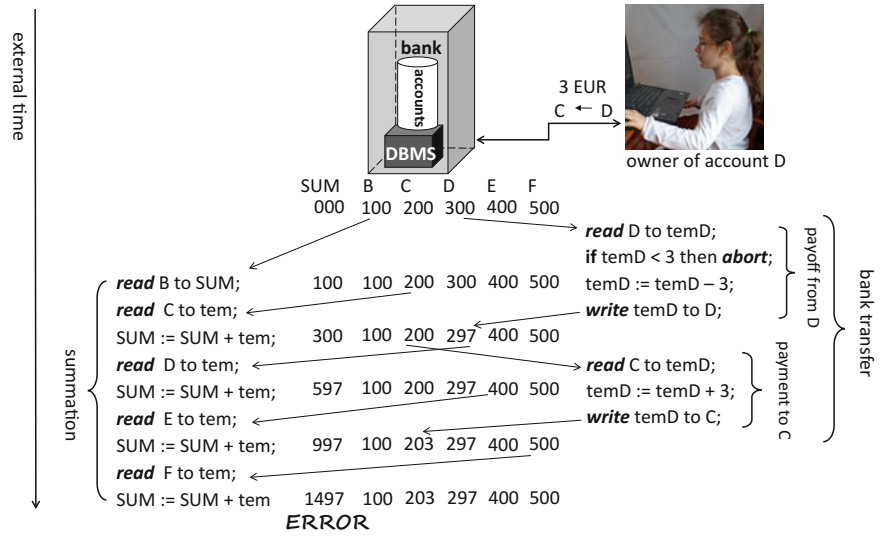


Fig. 3.8 No blocking—incoherent recovery; the total sum should amount to 1500 EUR

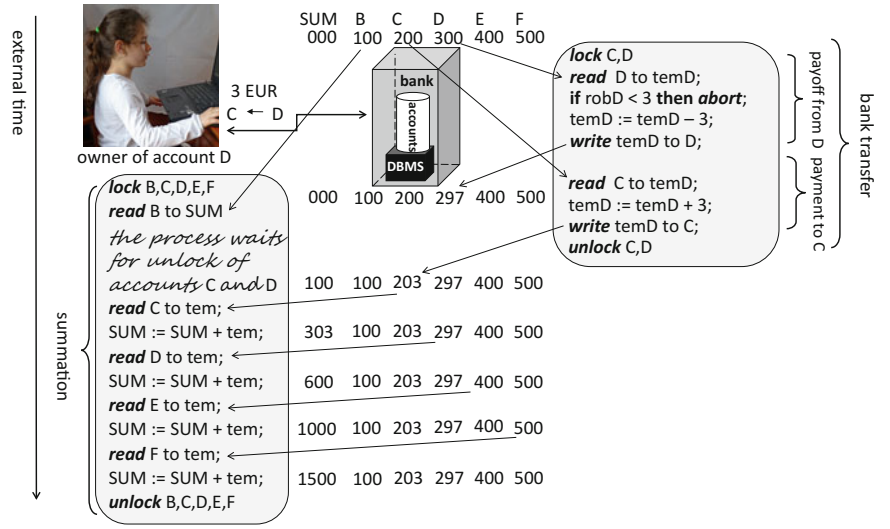


Fig. 3.9 Correct blocking: coherent recovery; protective zones are in frames on grey background; accounts C and D are first blocked by the client, then, all accounts—by the bank manager, whose activity has been delayed until unlock of C, D is executed

3.2 Deadlock

An exemplary pathological situation, occurrent in the parallel processing, is illustrated in Fig. 3.6 for nested transactions. The gentleman blocked access to account C1 before the lady who blocked access to account C2 and both clients have made pay-off from their own accounts B and D. Then the gentleman blocked C2, the lady blocked C1 (the accounts already blocked) and both clients again made pay-off from accounts B and D. The clients are waiting for unblock of accounts C1 and C2 delayed indefinitely. This kind of deadlock is referred to as a *deadly embrace*, illustrated in Fig. 3.10.

Such reciprocal expectation for events is a simplest case of deadlock. Some not so simple is when transactions (processes in general) are expecting some events in a single cycle as in Fig. 3.11 or quite complicated when transactions are interelated creating a graph with cycles, as in Fig. 3.12.

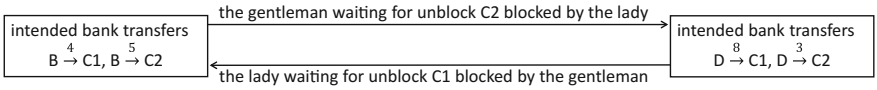


Fig. 3.10 Deadly embrace of two transactions

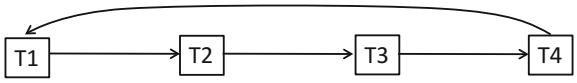


Fig. 3.11 Transaction T1 expects event from T2 and so on, up to situation when T4, expects event from T1. The deadlock is caused by a cycle of expectations (the so-called *wait loop*)

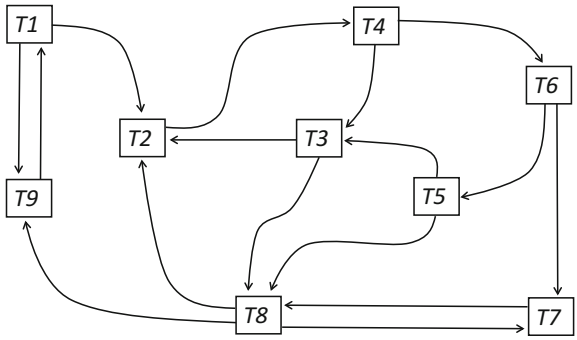


Fig. 3.12 The *wait graph* containing cycles: numerous deadlocks possible

A graph exemplifying current state of waiting for events by nine transactions is in Fig. 3.12. Vertex (node) of the graph stands for a transaction whereas arrow—a waiting for an event from transaction pointed to. Such a graph, called the *wait graph*, characterizes existence of deadlocks in the current state of system execution:

In a set of processes, a deadlock occurs if and only if the respective wait graph contains a cycle.

For any set of processes the system management may build the wait graph. Note that in this notion, there is no requirement of concurrency of processes: in such case, the relation of waiting is then empty. The wait graph is a dynamic structure: it is evolving during the system run, because the nodes (processes) and arrows (expectations) appear and disappear.

Deadlock is a pathological situation. So the designer of a distributed system must take it into account, and should provide a suitable remedy. To this end, the following kinds of salvage operations are distinguished:

Deadlock prevention—may be obtained by:

- blocking of all resources used during a transaction at its initiation and unblocking at its termination, as in Fig. 3.2. However this makes slower access to the shared resources (too coarse granulation), thus, this limits concurrency, and what is more, usually it is not known at the beginning of the transaction, what resources will be used;
- blocking resources in a fixed order; but this also limits concurrency as well as causes their untimely blocking.

Deadlock detection and removal—may be accomplished if:

- a module managing deadlocks creates and maintains the wait graph, in which periodically detects cycles and makes decision, which transaction should be aborted, so that to break the cycles; then makes the system return to a state from before occurrence of the deadlock;
- the system returns to a state from before occurrence of the deadlock, if suspension of activity exceeds predefined time—the *timeout* occurred;

The above examples of deadlocks arise, when processes that compete for resources, are not correctly ordered in time. Thus a source of such situations lies in incorrect synchronization. Another source of deadlocks is incorrect organization of communication between processes (Chap. 5). Notice that a system designer faces some contradictory needs: deadlock control vs system efficiency. Admittance of deadlocks, if are believed infrequent, enhances degree of concurrency—more processes exist in overlapping time periods. On the other hand, prevention, detection and removal procedures indispensable for protection against endless suspension of activity, deteriorate the system performance. So, design policy must resolve what is more favourable in applications the distributed system is intended for. Perhaps a compromise.

3.3 Starvation

Besides the deadlock, a harmful situation is starvation of processes. Consider the following example. Computers in companies A, B, C are performing the following actions:

(1) *check income in company book-keeping*;
(2) *block access to revenue offices accounts*;
(3) *pay tax to tax office*;
(4) *pay fee to social security*;
(5) *unblock access to accounts blocked in* 2;
(6) *repay part of debt*;
(7) *check income and go to* 2.

Actions 2 and 5 concern blocking and unblocking resources of offices in charge of tax revenue and social security. Concurrent execution of this system by computers A, B, C, may result in situation shown in Table 3.1 (the arrow pointing to actions, indicates presence of control in the „programs"). This situation occurs when computer of company B is never allowed to perform actions 3 and 4—computers A and C always forestall it. The states (numbered on grey background) are repeated cyclically: (4,5,6,7,8,9), (10,11,12,13,14,15), (16,17,18, …),… etc., that is, corresponding states in the bracketed groups are identical. Exactly: state(n) = state(n + 6) for $n > 3$, where state(n) denotes the nth state of this infinite sequence. Computer of the company B will never go out of action 2. The situation is called a *starvation* of the company's B process. It may be said metaphorically that a conspiracy of companies A and C against the company B has occurred.

Starvation is caused by faulty („unfair") strategy of allocation of resources by their managers. In general, this situation may be expressed as follows:

Starvation of a program during the run occurs if and only if execution of some its actions that must be executed, will never be executed in finite time during evolution of the system, resulting in the program activity indefinite suspension.

In particular, starving of a process occurs when its request for access to a resource will not be fulfilled in finite time during further evolution of the system.

There is a certain theoretical fact: the precise investigation of the starvation notion requires stronger formal means than investigation of deadlock. In case of starvation, they belong to the second order logic (where variables bound by quantifiers may stand for sets, functions, relations), whereas in case of deadlock—the first order logic suffices (the bound variables stand for simple, indivisible objects like numbers) (Czaja 1980).

Table 3.1 The process continued indefinitely—computers of companies A and C permanently outdistance the computer of company B in effective execution of action 2 (to infinity)

	Company A	Company B	Company C
1	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	➡1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	➡1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
2	1. check income in company book-keeping; 2. block access to revenue offices accounts; ➡3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	➡1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
3	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; ➡4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	➡1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
4	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; ➡5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
5	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; ➡6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; ➡3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
6	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; ➡7. check income and go to 2;	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; ➡4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
7	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; ➡5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
8	1. check income in company book-keeping; 2. block access to revenue offices accounts; ➡3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; ➡2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; ➡6. repay part of debt; 7. check income and go to 2;

(continued)

Table 3.1 (continued)

	Company A	Company B	Company C
9	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; **4. pay fee to social security;** 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; **7. check income and go to 2;**
10	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; **5. unblock access to accounts blocked in 2;** 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
11	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; **6. repay part of debt;** 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; **3. pay tax due to tax office;** 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
12	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; **7. check income and go to 2;**	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; **4. pay fee to social security;** 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;
13	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; **5. unblock access to accounts blocked in 2;** 6. repay part of debt; 7. check income and go to 2;
14	1. check income in company book-keeping; 2. block access to revenue offices accounts; **3. pay tax due to tax office;** 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; **6. repay part of debt;** 7. check income and go to 2;
15	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; **4. pay fee to social security;** 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; **7. check income and go to 2;**
16	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; **5. unblock access to accounts blocked in 2;** 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;	1. check income in company book-keeping; **2. block access to revenue offices accounts;** 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2;

(continued)

Table 3.1 (continued)

17	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2; Company A	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2; Company B	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2; Company C
18	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2; Company A	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2; Company B	1. check income in company book-keeping; 2. block access to revenue offices accounts; 3. pay tax due to tax office; 4. pay fee to social security; 5. unblock access to accounts blocked in 2; 6. repay part of debt; 7. check income and go to 2; Company C

This chapter is concluded with presenting two realizations of mutual exclusion in distributed systems, where interprocess communication takes place by means of message passing through the network only, not by shared memory. Synchronization is, thus, possible only in this way. These two realizations are: by supervisory server and by token ring. Others, based on logical clocks and global timestamps will be presented in Chap. 4.

3.4 Mutual Exclusion by Supervisory Server

Assume there is one protective zone (a counterpart of critical section in centralized systems) and during its execution no failure of the system occurs. The supervisory server receives ***requests*** for protected resources from user processes and makes decision if the resources can be accessed. If yes, it sends a message ***permission*** to the requesting process, otherwise (because another process is using the resource)—sends a message ***refusal*** and puts the process (its name and its current state) in the queue of waiting processes. On completion of executing of protective zone, the process sends a message ***release*** to the server and continues further run—execution of the local section. Then the server removes a waiting process from the head of queue and reactivates it. Notice that in this solution, the protective zone is being performed by a process which requested a resource, not by the server. In the opposite case, the server would send the ***release*** message to the user process. A schema of the supervisory server is in Fig. 3.13 and its exemplary activity in Table 3.2.

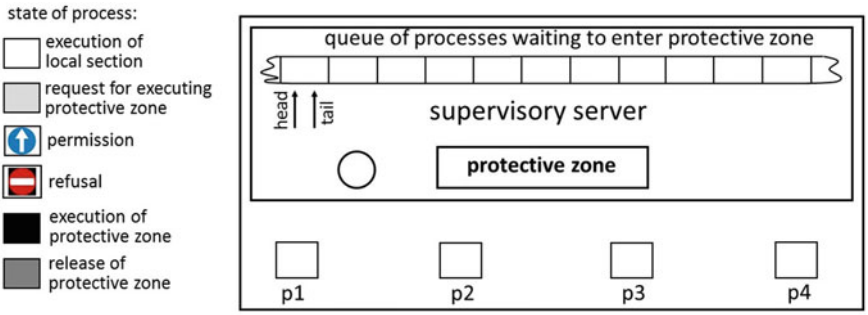


Fig. 3.13 A schematic view of a system of 4 processes and server managing mutual exclusion

3.5 Mutual Exclusion—Token-Ring Algorithm

Processes *p1, p2, p3,...,pn* are connected into a ring (cycle) which determines a route of the token. Their order is arbitrary but fixed, for instance like in Fig. 3.14.

A module of the system, that manages the ring, grants authorization for the processes to enter the protective zone in the ordering determined by succession in the ring. Metaphorically, this authorization is called a „token". Every process knows identifier of its successor in the ring. If a process is not in the protective zone, nor is requesting to execute it and has received a token from its predecessor, then immediately conveys the token to its successor. If it has issued request, then waits for a token, and as soon as receives the token—keeps it, executes the protective zone and on completion of the execution—conveys the token to the successor in the ring. The processes may be in the same states as in case of the method of the supervising server. A schema of the token ring of 4 processes is in Fig. 3.15 and its exemplary activity in Table 3.3.

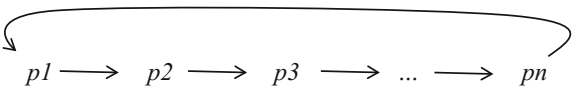
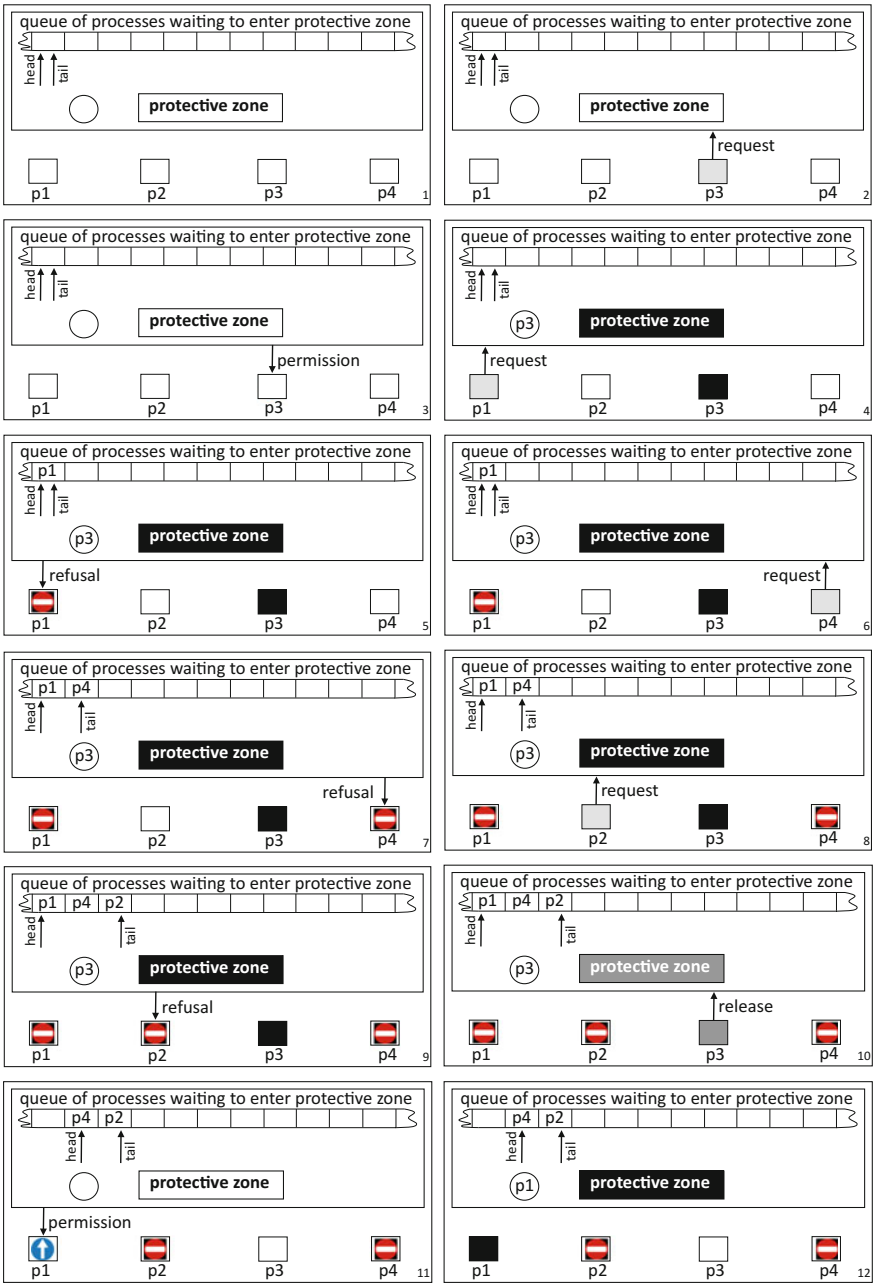


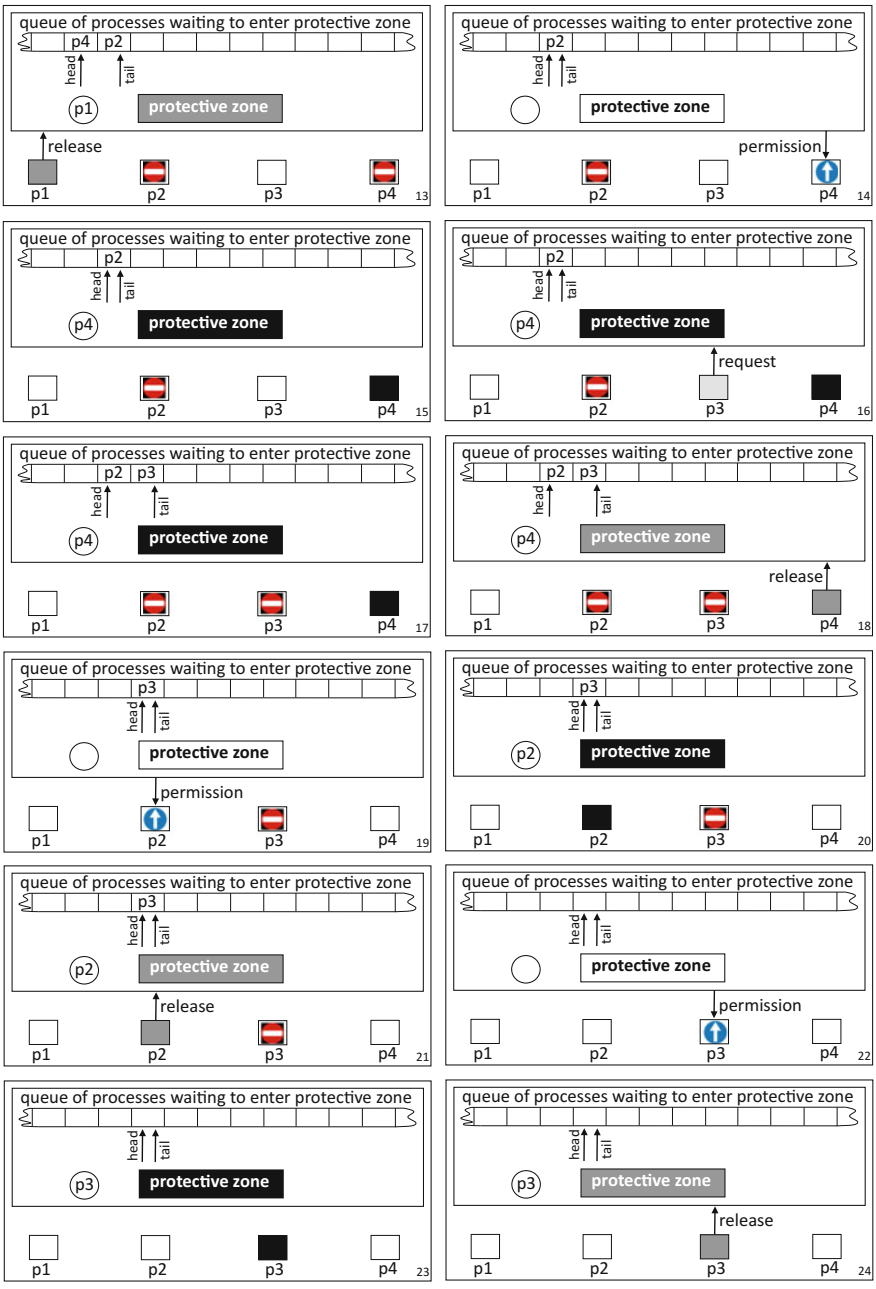
Fig. 3.14 Token ring

Table 3.2 States 1–12 of 4 processes with supervisory server managing mutual exclusion



(continued)

Table 3.2 (continued)



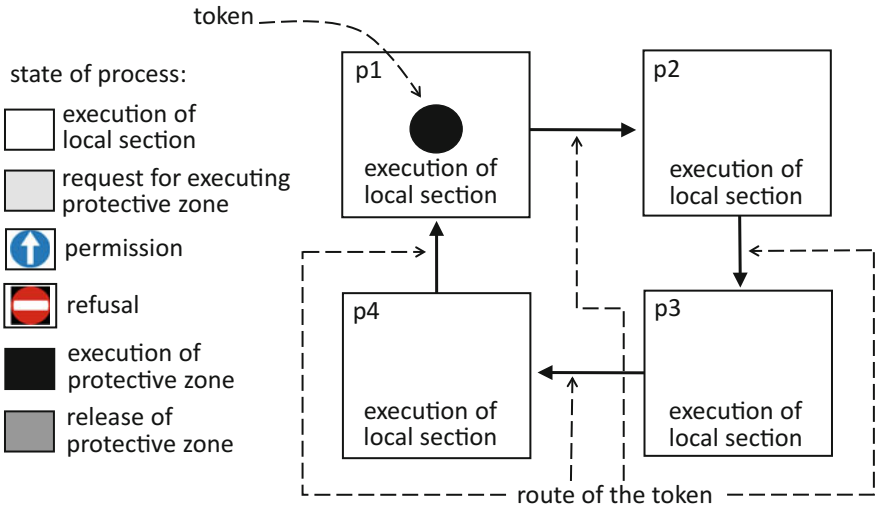
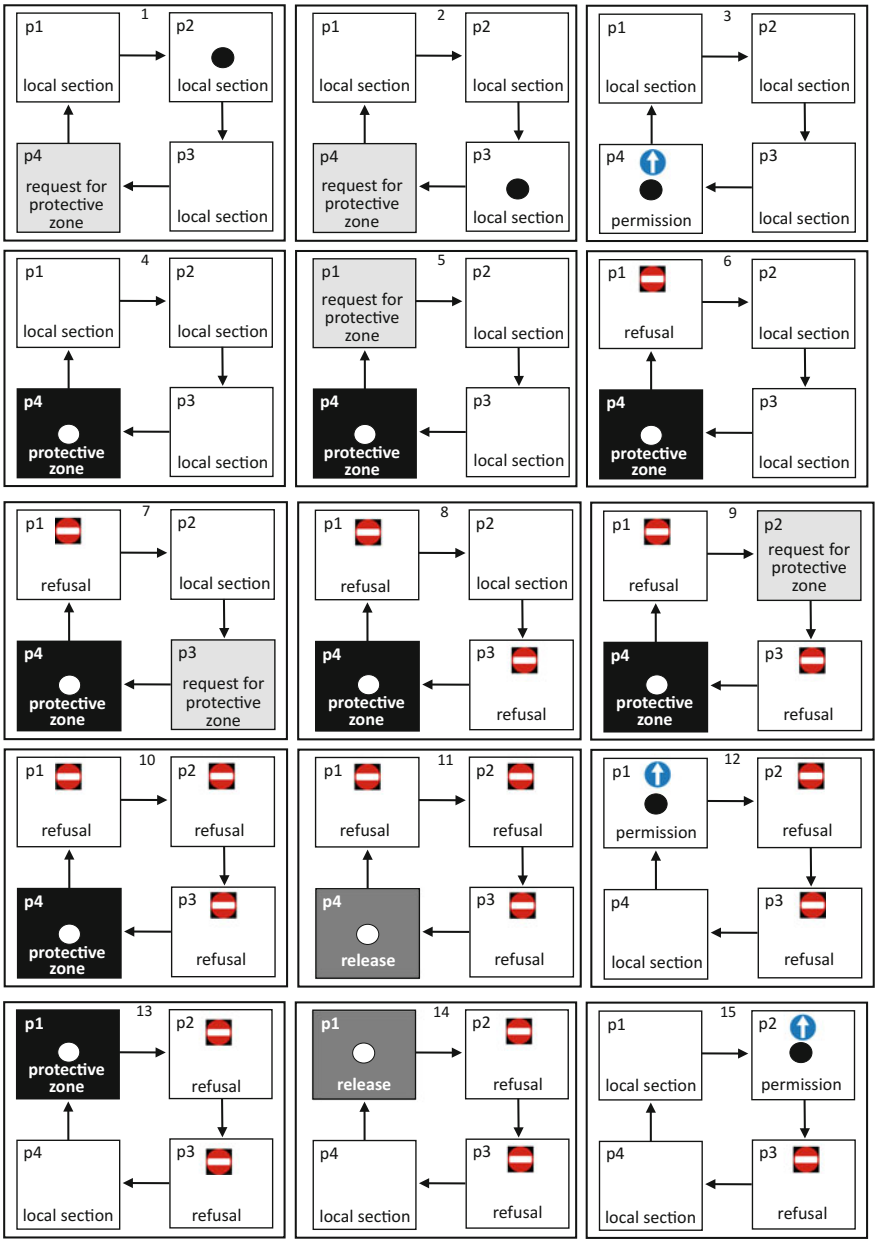


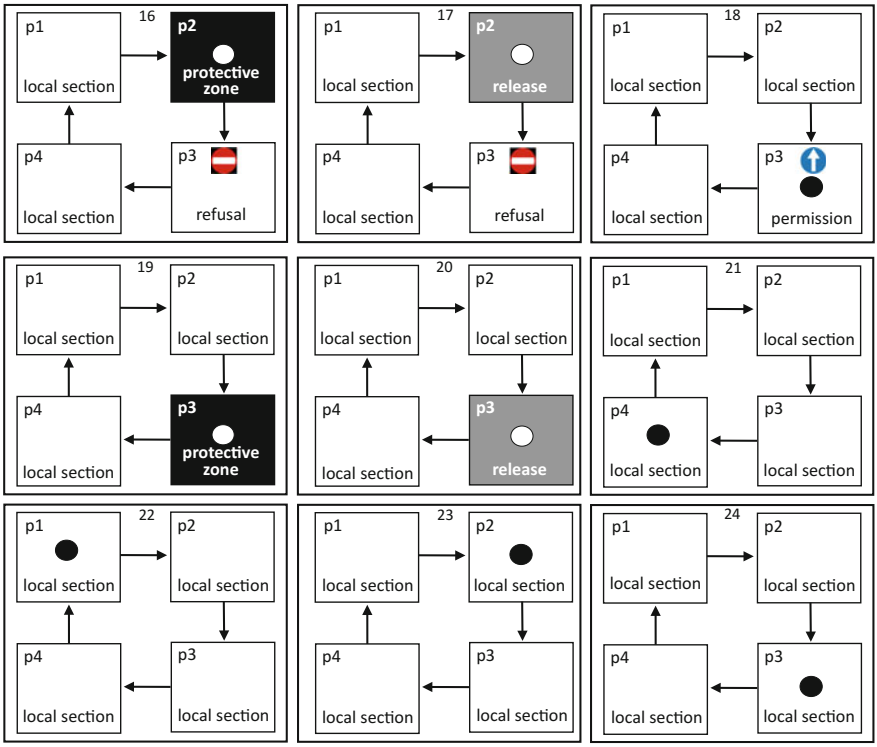
Fig. 3.15 Schema of the token ring with four processes

Table 3.3 Four processes with token ring managing mutual exclusion



(continued)

Table 3.3 (continued)



References

Courtois, J., Heymans, F., & Parnas, D. L. (1971). Concurrent Control with "Readers" and "Writers". *Communication of the ACM, 14*(10), 667–668.

Czaja, L. (1980). Deadlock and fairness in parallel schemas: A set-theoretic characterization and decision problems. *Information Processing Letters*, *10*(4–5), 234–239.

Weikum, G., & Vossen, G. (2002). *Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery*. Elsevier.

Chapter 4
Time, Coordination, Mutual Exclusion Without Supervisory Manager

4.1 Physical Time

In the sequential processor, time is measured by its clock and progresses linearly, i.e. for any two distinct events, one of them precedes the other. Ordering of events, that is lapse of time in the processor, is based on the notion of precedence. As an indivisible (atomic) event, the clock's pulse called a „tick" is admitted. But execution of a processor's instruction (regarded before as an event) takes place usually in several ticks, as well as between consecutive ticks, some electric actions appear, being effect of some phenomena on the nuclear level, etc. That is why the set of real numbers with ordinary ordering is admitted as a model of linear time, because between two distinct real numbers (images of events—abstract objects), there is a different real number (an image of certain event). But why not the set of rational numbers, which enjoys the same characteristic? Because physical magnitudes that depend on time, are often expressed by operations that return real numbers. Because flow of time in a processor is measured by ticks of its clock and clocks in different processors have different ticking frequencies, we say about the local time of processors. In the distributed system, i.e. in a collection of processors, the flow of time cannot be measured by ticks of their clocks: it is impossible to say for any two events, that one event precedes the other: a common clock does not exist. Events are partially ordered, not linearly. However, there is sometimes a need to contemporize some actions, e.g. dispatch of a message must precede its reception. For some applications, a pattern of external (global) time is needed to which the system may refer, when makes a coordination. Such a pattern is a sequence of some events that occur in a chosen physical phenomenon. This is a sequence of occurrences of a periodical phenomenon of a chosen physical reality. Such a choice is prompted by possibly regular periodicity of the phenomenon. Examples:

- Sun-Earth (*solar time, real and mean*). Event: crossing a fixedly agreed meridian by the sun. As the mean solar second, has been admitted 1/86400 of

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_4

the period between two such consecutive events; This magnitude is not exactly fixed, because this period lengthens about 16 μs (in average) annually.

- Quartz crystal oscillator in clocks; chosen effect—deformation of quartz crystal under electric field; frequency spans between tens of thousand to tens of million deformations (depending on workmanship) in the period defined to be one second—according to a certain fixed pattern of time measure.
- Atomic oscillator—now a standard since 1967, one second: 9 192 631 770 periods of transition between two energy levels in the basic state of Caesium-133. Event: change of energy level. On this oscillator is based nowadays the so-called **U**niversal **T**ime **C**lock and **C**oordinated **U**niversal **T**ime (in English) or **T**emps **U**niversel **C**oordonné (in French) of accuracy better than 1 s in 1.4 milion years (compromise acronym: **UTC**). Signals of the UTC time are broadcast by radio broadcasting stations and by Internet (http://www.worldtimeserver.com/current_time_in_UTC.aspx). This is the so-called timed signal service.

Remarks

1. It might be predicted that the race for better and better accuracy of time measuring will never be ended. This is so, because of fast development of technology and science, wherever this is indispensable. A few examples are: GPS technology applied in vehicle navigation, particularly driverless, for new kind of sensors in geological explorations, for precise navigation of aircrafts and spacecrafts, for precise spectrometers in the international space stations, also for observational astrophysics. That is why more accurate time measurements have been invented recently, than that based on Caesium-133. These are, for instance, the so-called mercury-clock, reported of at least 5 times better accuracy than the caesium clock, an optical single-ion clock of 100 times better accuracy than the caesium clock, or quite recently the so-called Quantum Logic Clock. All these clocks, exceeding the accuracy of the standard caesium clock, are experimental devices so far, devised in 10 recent years.
2. The presentation of phenomena appearing in distributed systems will be based on conventional, everyday practice perception of „flow" of time. So, the presentation refrains from taking into account such relativistic facts like different outcomes of seeing order of the same events by two external observers or lack of meaning of commonly experienced notions like „simultaneity", „earlier", „later", etc. On the generality level of presentation adopted here, such notions will be used liberally, whenever prove helpful in explaining distributed systems' properties. Although the aforementioned consequences of finite light velocity will not influence our considerations, the duration of data transmission through the network will be of primary importance in this and next chapters.

In a chosen physical system, the real numbers are univocally assigned to events in a distributed system, in such way that if an event x occurs earlier than y (but remember remark 2 above), then the number assigned to x is less than the number assigned to y. We say that events appear in instants that are their numerical images. Physical time is treated here as the linearly ordered set of instants—images of events—in a given physical system. Therefore, it depends on this system. An event will be treated as a primary notion, an indivisible, timeless object. The notion of a partial and linear order relation will be needed (binary relations—see Sect. 10.1 in Chap. 10):

A set Z is partially ordered iff its elements are related by relation $\sqsubseteq \subseteq Z \times Z$ satisfying: for every $x \in Z$, $y \in Z$, $z \in Z$:

1. $x \sqsubseteq x$ (reflexivity)
2. if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$ (antisymmetry)
3. if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$ (transitivity)

Moreover, if apart from (1), (2), (3):

4. $x \sqsubseteq y$ or $y \sqsubseteq x$ (connectivity)
   then Z is linearly (totally) ordered. As usually, $x \sqsubset y$ iff $x \sqsubseteq y$ and $x \neq y$.

The set $Z \times Z$ (Cartesian product) denotes the set of all ordered pairs of elements of the set Z, thus the order $\sqsubseteq$ and $\sqsubset$ is a subset of $Z \times Z$, satisfying above axioms. In case of partial (but not linear) order, it is not necessary that any two elements be related by $\sqsubseteq$ order. The set of all events during activity of a distributed system, is ordered only *partially* by relation $\sqsubseteq$ (meant as „event x occurs earlier than y or $x = y$"), because not all pairs of events are necessarily related by the $\sqsubseteq$ order. The set of events emited by any sequential processor is *linearly* ordered by $\sqsubseteq$. The order of instants of time $C(x)$, $C(y)$, i.e. real numbers, assigned to events x, y, must be coherent with the order of these events in the following sense: if $x \sqsubseteq y$ then $C(x) \leq C(y)$. It will be seen that the converse implication is not true.

A coordination of local time of processors, that is synchronization of their clocks, may be accomplished:

- by referring to a pattern of physical time, like UTC, for some applications, e.g. when real time of banking transactions should be recorded;
- by periodical computing of arithmetic average of clocks' indications in processors and setting the clocks to this average;
- between clocks of various processors in the network, so that the logical order of events be preserved, necessary for their meaning, for instance, reception of a message can never occur earlier than its dispatch.

The two first ways of clock synchronization are referred to as realization of physical clock, the third one—realization of logical clock. Let us at first consider examples of realization of a physical clock.

4.1.1 The Cristian Method of Clock Synchronization (Cristian 1989)

The standard time UTC, requested by client-computers, is being delivered by a time server, thus, by an external source for the clients. In a moment of reception, the standard time is later than delivered by the server, due to the transmission duration. Moreover, if the client's local time flows faster than time of server and the client announces e.g. time of takeoff of an aircraft, then, after receiving the server's time, announces the landing time, it might happen that the aircraft landed before the takeoff—even in the same time zone. The client's time „stepped back", which is impermissible in a correct synchronization. These problems are being solved by the Cristian method as follows. Let t_0 be a time of sending request to the server by the client, let T_0 be a time when the request reached the server, let T_1 be a time of sending reply T to the client by the server and let t_1 be a time of reception of message T (the reply T is the requested standard time). All these times are external, standard, say, UTC, see Fig. 4.1.

In the simplest variant of the method it is assumed that the reply takes:

$$\frac{(t_1 - t_0) - (T_1 - T_0)}{2} \text{ of time units}$$

and the client sets its clock to time:

$$T + \frac{(t_1 - t_0) - (T_1 - T_0)}{2} = T + \frac{(T_0 - t_0) + (t_1 - T_1)}{2} \approx T + t_1 - T_1$$

assuming that travel period of request ($T_0 - t_0$) and of reply ($t_1 - T_1$) through the network are approximately equal. They are not fixed: they depend on network load, distance between server and client, etc. and may be estimated only [the reasoning behind the estimation is in the original publication (Cristian 1989)]. The period $T_1 - T_0$ of preparing a packet with reply T by the server, is negligible in comparison with transmission time, so is neglected. Thus, value T delivered by the server is increased by $t_1 - T_1$.

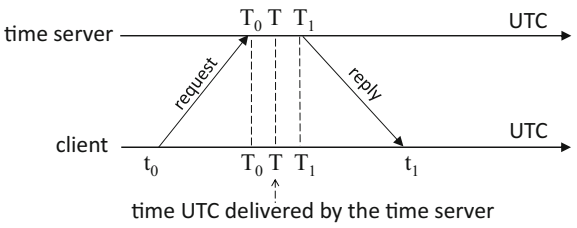


Fig. 4.1 Passing messages between the client and time server

Estimation of the Cristian's method accuracy

If the shortest possible time of sending a client's request and (similar) time of returning response of the server is known as $m = min(T_0 - t_0) \approx min(t_1 - T_1)$, then after sending request in a moment t, the server would get it at the earliest in the moment $t + m$. On the other hand, the server returns response at the latest in the moment $t + (t' - t) - m = t' - m$ where t' is a moment of reception of server's time T, by the client.

Therefore, T satisfies the inequalities: $t + m < T < t' - m$ i.e. T is contained in the period:

$t' - m - (t + m) = (t' - t) - 2m$. Hence, the accuracy of the method:

$$\frac{t' - t - 2m}{2} = \frac{t' - t}{2} - m$$

Value $(t' - t)$ is the period from a request till response.

Because transmission time of the client's request and the server's reply are indefinite and possible are system failures, the Cristian's paper (Cristian 1989) focusses on a probabilistic analysis of synchronization between the client and server.

The avoidance of stepping back of the client's clock

The second problem in synchronizing of clocks is avoidance of stepping back of client's clock if it is faster than the server's clock. Let H be a client's hardware clock and suppose it cannot be set by software. In order to synchronize (approximately—the exact is impossible!) this clock with the server's clock, the so-called *virtual clock* C (sometimes called „logical", but this term is reserved for the Sect. 4.2) is programmed as a special register, and set periodically to the time delivered by the server. This is a „clock" for correcting the hardware clock H indications. Formally, H and C are functions of time:

H: $R \rightarrow R$ (R-the set of real numbers)
C: $R \rightarrow R$

Values $t - H(t)$ and $t - C(t)$ are deviations of the times shown by clocks H and C, from t—the physical time shown by the server. Thus, $d(t) = (t - H(t) - (t - C(t)) = C(t) - H(t)$ is the value, by which the time of H must be corrected so that to obtain the time of the clock C:

$$C(t) = H(t) + d(t)$$

Assume that d is a continuous function (to ensure avoidance of „jumps" of the virtual clock C) and in the simple variant let it be linear:

$$\mathrm{d}(t) = x \cdot \mathrm{H}(t) + y$$

where x and y are constants which can be calculated as follows. First, note that $\mathrm{C}(t) = \mathrm{H}(t) + \mathrm{d}(t) = \mathrm{H}(t) + x \cdot \mathrm{H}(t) + y = (1+x) \cdot \mathrm{H}(t) + y$ is the time shown by clock C when clock H shows H(t). Next, assume that at the moment t, the virtual clock C shows L = C(t) and hardware clock H shows h = H(t) and the server shows physical time M. If M > L, then the time shown by C should be increased and if M < L, it should be decreased, so that after a period Δt (from t), the clock C would show time M + Δt, but not L + Δt. Putting these times to equation $\mathrm{C}(t) = (1+x) \cdot \mathrm{H}(t) + y$ we get that in the moment t the clock C shows time $\mathrm{L} = (1+x) \cdot \mathrm{h} + y$ and in the moment $t + \Delta t$ the clock C shows a compensated time $\mathrm{M} + \Delta t = (1+x) \cdot (\mathrm{h} + \Delta \mathrm{t}) + y$. Therefore, we have two linear equations with two unknowns x and y:

$$\begin{aligned} \mathrm{h} \cdot x + y &= \mathrm{L} - \mathrm{h} \\ (\mathrm{h} + \Delta t) \cdot x + y &= \mathrm{M} - \mathrm{h} \end{aligned}$$

Assuming $\Delta t \neq 0$, after solving the equations, we get:

$$x = \frac{\mathrm{M} - \mathrm{L}}{\Delta t}$$

$$y = \frac{(h + \Delta t) \cdot \mathrm{L} - (\mathrm{M} + \Delta t) \cdot \mathrm{h}}{\Delta t}$$

The Cristian's method is ***passive***, in that the time server is inactive as long as it does not receive a request for time synchronization from a client. Here, the ***external*** source of clocks synchronization is applied. To avoid a system failure, in case of time server crash, several time servers, providing time signals and mutually synchronized, may be used. In this case, the clients send request to all time servers and synchronize their clocks e.g. in accordance of the first response received.

4.1.2 The Berkeley Method of Clock Synchronization (Gusella 1989)

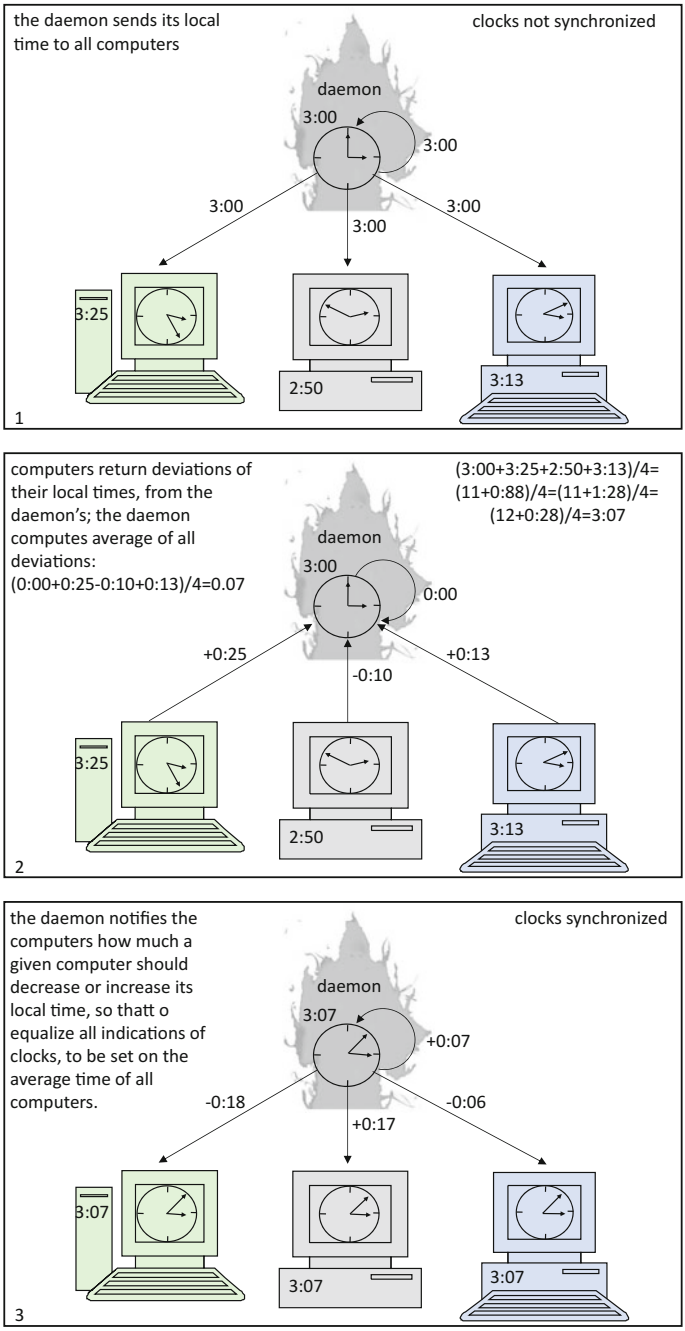
In this method, developed at the University of California, Berkeley in 1989, there is no server delivering an external time, e.g. UTC. Instead, there is a computer, chosen from among all in the system, called a *time daemon*, which periodically inspects the local time of all, computes the average of all differences between its own local time and local times of remaining computers called *clients* and informs every one how many time units it should make a shift forward or backward the indication of its local clock (the time register). Such synchronization is called ***internal***, since does not refer to the time delivered from outside. The two-way communication between

the daemon and clients, takes some time, that must be taken into account in computing the average. Therefore, the daemon must estimate the time of data transmission: *daemon* ⟶ *clients* ⟶ *daemon* ⟶ *clients*, as it was the case in the Cristian's method. The synchronization proceeds as follows. The daemon broadcasts its time to all computers, who send back differences between their local times and the time delivered by the daemon. Then, the daemon computes the average of the differences and broadcasts amendments to the computers, which on this basis set local clocks to the average time of all. Table 4.1 shows an example of two consecutive time corrections. At the states 1, 2, 3 the synchronization of clocks' indications to the time 3:07 takes place and at the states 4, 5, 6—to the time 3:14. The time flow is here expressed in hours and minutes: 3:07 means seven minutes past three o'clock, in reality, the time units are milliseconds or their fractions. Since the end of first correction till begin of the second, three minutes passed on the daemon's clock. It happens that indications of some clients' clocks differ substantially (by a predefined constant) from the remaining. Such clients do not take part in computing the average, but the daemon sends the amendments to such clients too, who set their clocks to the average time of all. The election of a computer to play role of the daemon (e.g. if the current one gets faulty) will be described in Chap. 7. The Berkeley method is ***active*** in that the daemon initiates consecutive cycles of synchronization.

Remarks

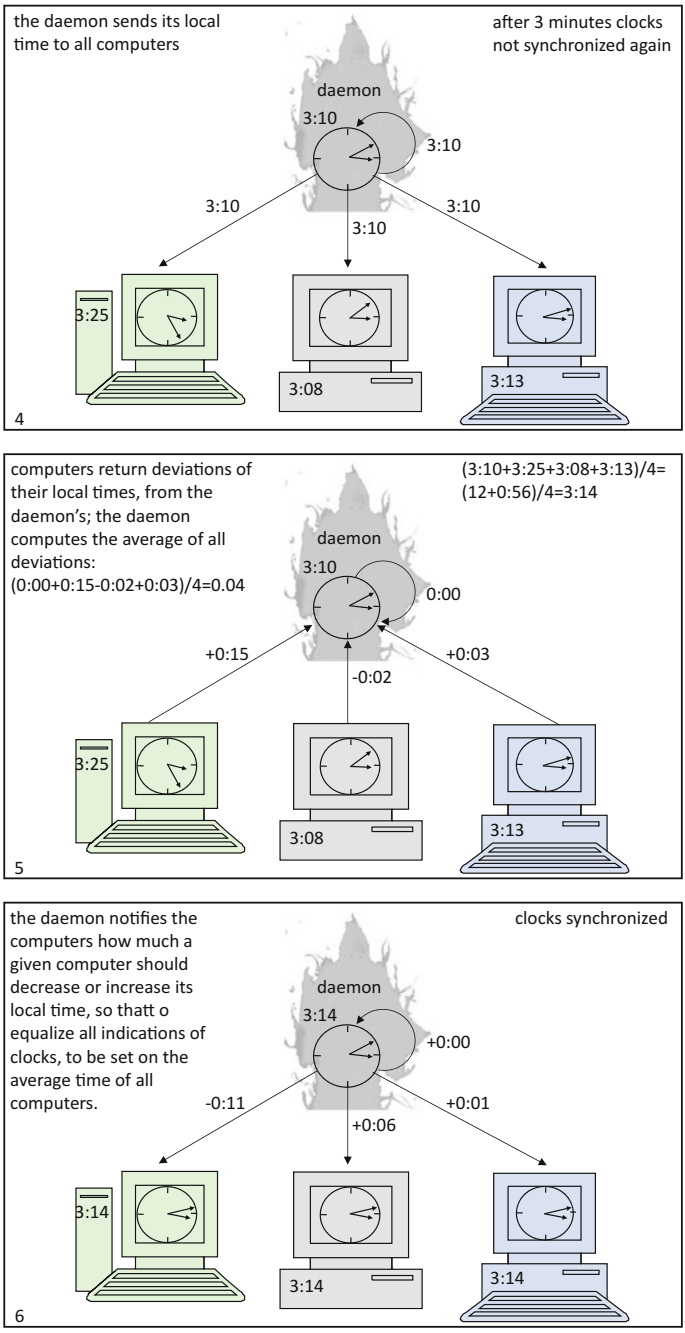
1. In the original paper (Gusella 1989) on the Berkeley method, the terms „master" and „slave" have been used for what we called here „daemon" and „client". The term „daemon" refers, in that paper, to a mechanism responsible for time management inside all computers in question. The detailed analysis of the method's accuracy is avaiable in Gusella (1989).
2. Notice that the same result is obtained if the daemon would compute directly the average time of all computers and broadcast it to all computers. Such computation is shown in the upper right site of states 2 and 5 depicted in the Table 4.1, whereas computation based on deviations between the daemon's and clients' time (for better accuracy)—in the upper left site.

Table 4.1 Synchronization by means of a daemon. States 1–3 synchronization of clocks by the daemon—in action



(continued)

Table 4.1 (continued)



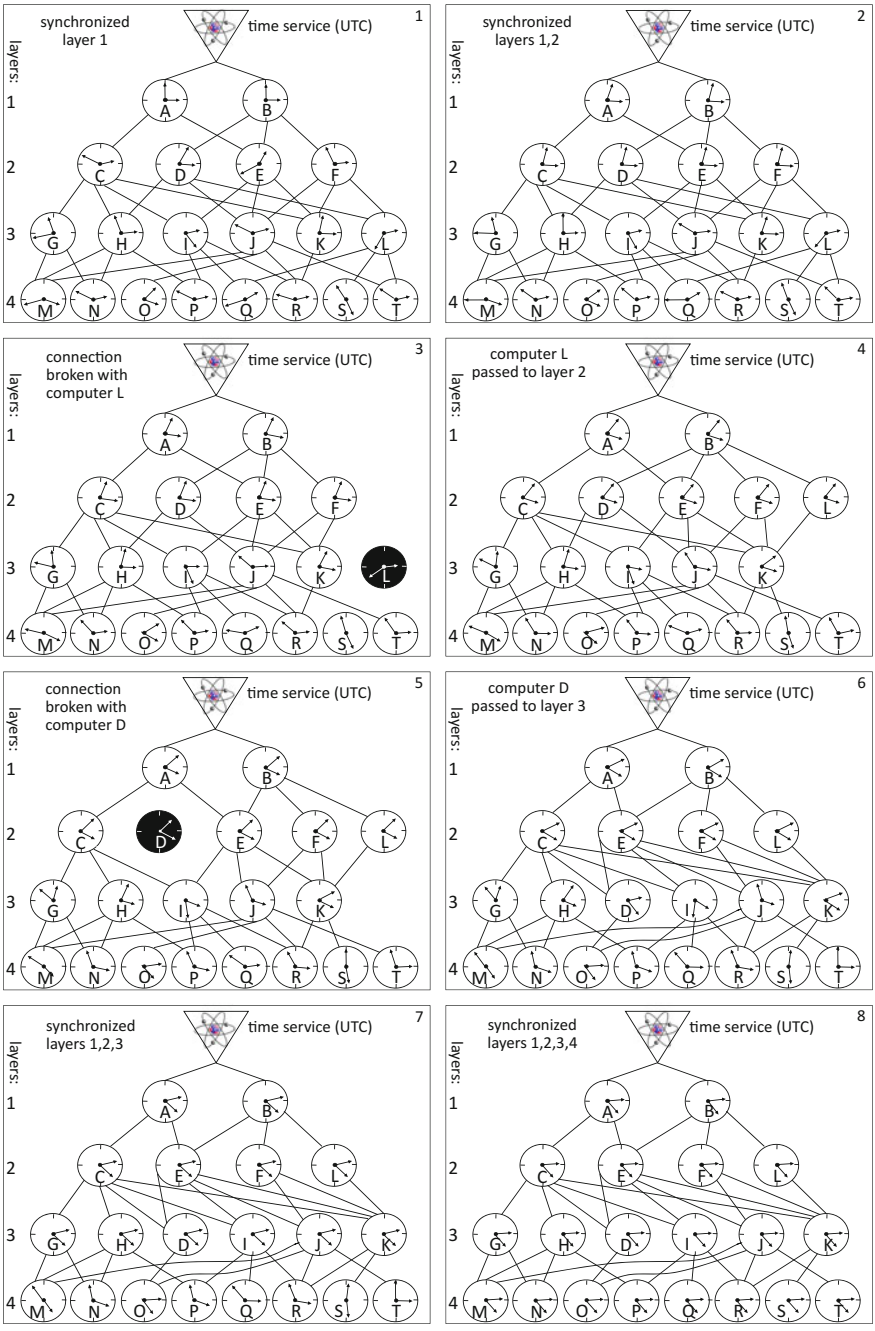
4.1.3 The Network Time Protocol (NTP) Method of Synchronization of Clocks (Mills 1991)

In this method, computers in the system are arranged in a varying layered graph structure. A graph has an n-layered structure (n > 1) if its set of vertices V can be partitioned into a union of subsets numbered from 1 to n: $V = V_1 \cup V_2 \cup \cdots \cup V_n$, $V_i \cap V_j = \varnothing$ ($i, j = 1,\ldots, n, i \neq j$), called layers, so that edges may connect vertices from the neighbouring layers V_i, V_{i+1} only. The structure is called a *synchronization subnet.* Computers in layer 1 receive directly signals of UTC from a time service and are mutually synchronized. Computers of the $i + 1$ ($i < n$) layer are being synchronized by computers of the layer i by means of a certain chosen method, e.g. by the Cristian's method, extended to a group of synchronizing computers. Therefore computers of the layer $i + 1$ play a part of clients of computers of the layer i. That is why all computers in this method are called servers. Synchronization is all the more exact in layers closer to the time service, i.e. those with low numbers. Apart from their function as synchronisers and clients, the computers may intercommunicate regardless of their location in the graph layered structure. Communication links not intended for synchronization, are used for reconfiguration of the graph structure, when a certain computer gets faulty and is removed from the synchronization subnet, or crosses from one layer to another when a disconnection of synchronizing links with it occured. So, the graph structure may fluctuate: the synchronization subnet is subject to reconfiguration. An exemplary synchronization subnet in the course of working is depicted in Table 4.2. Computer L has passed from layer 3 to layer 2 and computer D from layer 2 to layer 3. Note that the synchronization subnet, for n = 2 is a bipartite graph, which, in this case, is the communication structure in the Cristian's method but with more than one time server. The synchronization subnet is a mechanism for time management in various kind of networks. It became a basis for standard protocol of clocks synchronization in the Internet (Network Time Protocol 1989). It takes into account time zones and transmission delay periods. Advanced statistic methods reduce these delays in large networks to milliseconds, whereas in local networks—to fractions of milliseconds. Because of multiplicity of servers and links with clients, the NTP method enjoys a good degree of fault resistance caused by computers' failures and loss of packets in transmission channels. For better efficiency, a simplified version of the NTP protocol (abbreviated to SNTP) resigns from some algorithms of the full protocol, not necessarily required by some distributed systems.

4.2 Logical Time: Precedence of Events, Time Compensation, Timestamps, Logical Clock

Computers measure off time of processes by means of their own clocks, usually of different frequencies. Thus, it is impossible to fix a linear ordering of events occurring in different computers working independently, only on the basis of

Table 4.2 Synchronization of clocks by the NTP method—in action



indication of their local clocks. Events in the system are only partially ordered. The computers cooperate by sending messages. There is no common memory, so the transmission through communication links only makes possible this cooperation. To avoid absurd situations perceived on the basis of indications of local clocks, it is necessary to ensure that sending a message would precede (in physical, external time) its reception. Sending and receiving messages will be the only events (in different computers) requiring to determine the ordering. Obviously events occurring in every computer with sequential processor are linearly ordered—this is determined by the processor structure, its instruction cycle (Chap. 1). The abstract relation of partial and linear order has been defined in Sect. 4.1. Here, let us define these relations between events occurring in the same process as well as between sending and reception of a message, occurring in different processes and let us combine them into a special partial ordering of events occurring in a system S. Let $E(S)$ denote the set of all such events. For events $a, b \in E(S)$ two auxiliary relations are admitted as primary notions:

$\xrightarrow{process}$ and $\xrightarrow{message}$ of the following meaning

(1) If a precedes b in the same process or if $a = b$ then $a \overset{process}{\rightarrow} b$

(2) If a is sending a message in a certain process and b is a reception of this message in another process then $a \overset{message}{\rightarrow} b$

Acc relation of (weak) precedence $\rightsquigarrow \subseteq E(S) \times E(S)$ of events in a system S is a least (with respect to $\subseteq$) relation satisfying conditions:

i. If $a \xrightarrow{process} b$ or $a \xrightarrow{message} b$ then $a \rightsquigarrow b$
ii. If $a \rightsquigarrow b$ and $b \rightsquigarrow c$ then $a \rightsquigarrow c$

This precedence is called „weak", since $a \rightsquigarrow a$. This is a partial order in the set $E(S)$, a modified precedence introduced by Lamport (Lamport 1978), where $\xrightarrow{process}$ is not reflexive.

Events a, b are said *independent* (concurrent) if neither $a \rightsquigarrow b$ nor $b \rightsquigarrow a$, written $a \| b$.

Figure 4.2 shows processes *p1*, *p2*, *p3* where events depicted black denote sending a message, events grey—reception of a message and white—other events. The following relationships take place:

$$a \rightsquigarrow i, b \rightsquigarrow k, h \rightsquigarrow e, c \rightsquigarrow q, j \| d, d \| h, b \| j, j \| d, c \| n.$$

So, $x \rightsquigarrow y$ holds if and only if in the diagram is a path from x to y leading by arrows. Note that more than one path is possible. Partially ordered events are watched from outside of the system as occurring in real (external) time, e.g. UTC, thus $x \rightsquigarrow y$ should hold only if $C(x) \leq C(y)$, where $C(x)$, $C(y)$ denote real numbers corresponding to externally visible times of occurrences of the events x, y. Hence, the implication $x \rightsquigarrow y \Rightarrow C(x) \leq C(y)$ is required. Because of absence of global clock, a problem arises: how to determine time measurement in the distributed system, that is, how to assign

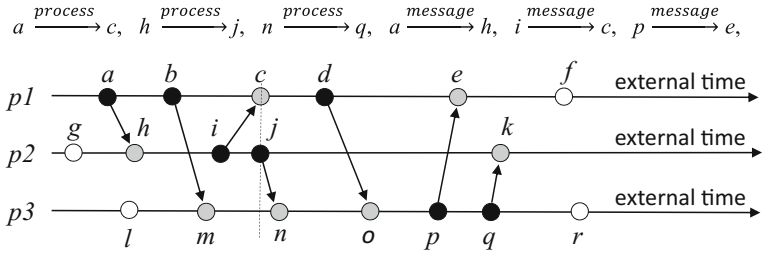


Fig. 4.2 Diagram of relationships between events in processes *p1*, *p2*, *p3* as seen by external observer

a time instant to a given event, so that this implication be true? To this end, an injective mapping C: $E(S) \rightarrow$ R (R—set of real numbers), called an abstract *logical clock*, is defined, satisfying implication $x \rightsquigarrow y \Rightarrow \mathrm{C}(x) \leq \mathrm{C}(y)$, for all events $x, y \in E(S)$. If events x, y occur in the same process, then this implication holds evidently, since relation $\rightsquigarrow$ reduces to $\xrightarrow{process}$ in this case. If however x is a dispatch of a message and y—its reception, then to make the implication algorithmically realizable, this abstract logical clock should behave similarly to its algorithmic representative in the method described in Sect. 4.1 (avoidance of time retreating). That means, if the dispatch x appeared later than reception y—according to indications of local clocks of sender and receiver, then the receiver must shift forth indication of its local clock (time register), so that the local time of y would become somewhat later than the local time of x. Therefore the sender dispatches—along with a message—its current local time, the receiver compares it with its own current local time and if it is earlier than the sender's, then the receiver makes compensation of its local time. The exemplary compensation of time during activity of three processes *p1*, *p2*, *p3* is shown in Table 4.3. The black fragments of „stretching ribbons" depict increase of local time as result of compensation.

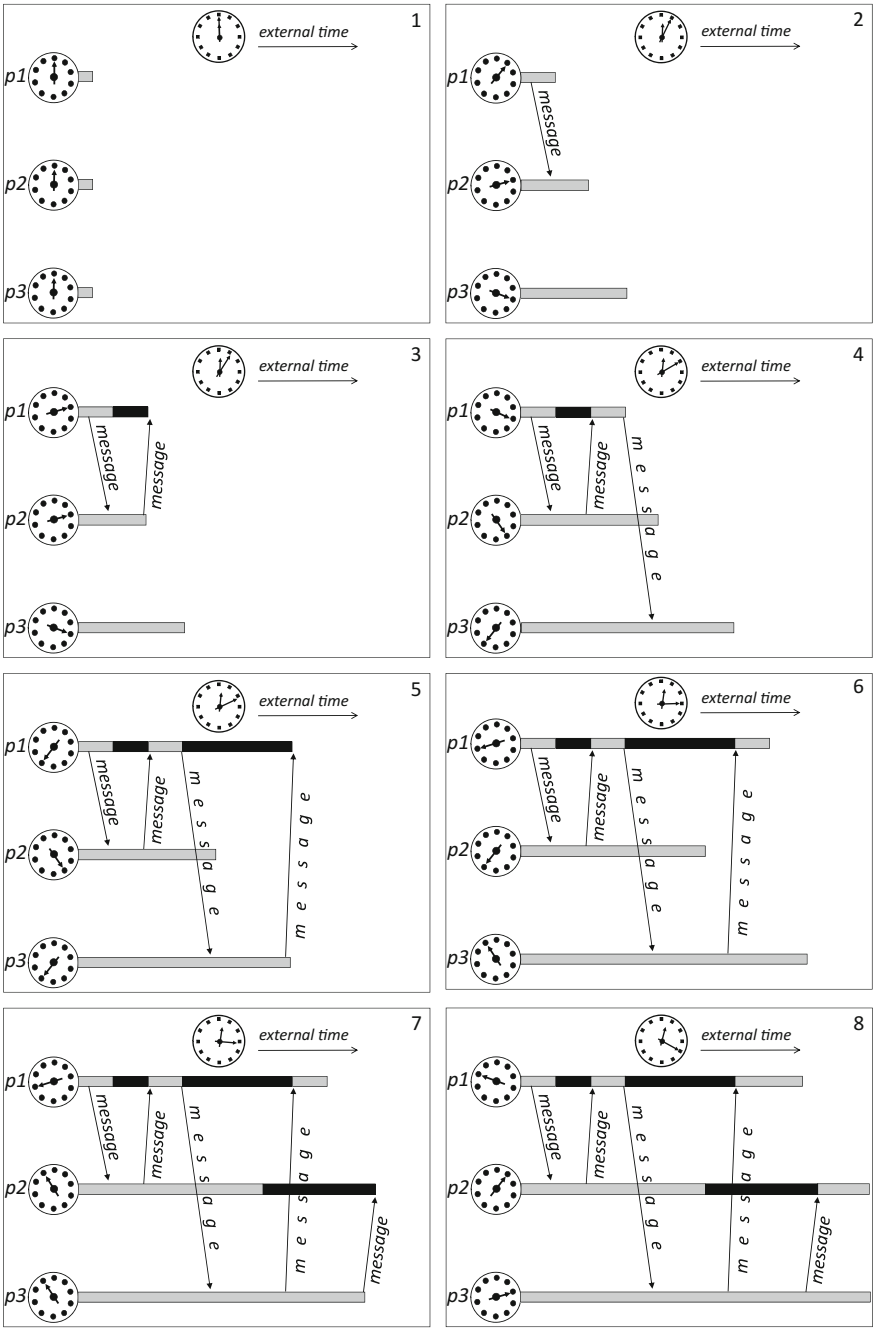
Defining function C, the abstract logical clock and global timestamps of events

For the formal definition of function C, some denotations and assumptions will be needed. If event x precedes y in a process p or if $x = y$ then write $x \xrightarrow{p} y$. By $\mathrm{C}_p(x)$ is denoted indication of the local clock of process p at the moment of occurrence of x in p. Therefore if $x \xrightarrow{p} y$ then

$\mathrm{C}_p(x) \leq \mathrm{C}_p(y)$. If $\xrightarrow{message}$ where x occurs in a process p and y in a process q, then two cases are possible:

1. If $\mathrm{C}_p(x) \geq \mathrm{C}_q(y)$ (local time of dispatch is later or equal to local time of reception) then $\mathrm{C}_q(y) := \mathrm{C}_p(x) + \Delta_{xy}$ where $\Delta_{xy} > 0$ denotes the (estimated) period of time from dispatch to reception of the message (shift forth indication of local clock in the process q, slower than p).
2. If $\mathrm{C}_p(x) < \mathrm{C}_q(y)$ (local time of dispatch is earlier than local time of reception) then no action: $\mathrm{C}_q(y)$ remains unchanged.

Table 4.3 Periods of lapse of time in processes *p1* and *p2* shown as black, represent compensation of local times, when they get messages from a process with faster clock



The shift of the clock indication in process q in the case 1, is a ***time compensation*** in this process.

The logical clock C: $E(S) \rightarrow$ R is defined as follows:

- $C(x) = C_p(x)$ if x is a dispatch of a message in the process p or is not a communication event;
- $C(y) = C_q(y)$ if y is a reception of a message in the process q and $C_q(y)$ is computed in the case 1.

The value $C(x)$ is referred to as a (compensated) ***timestamp of event x***.

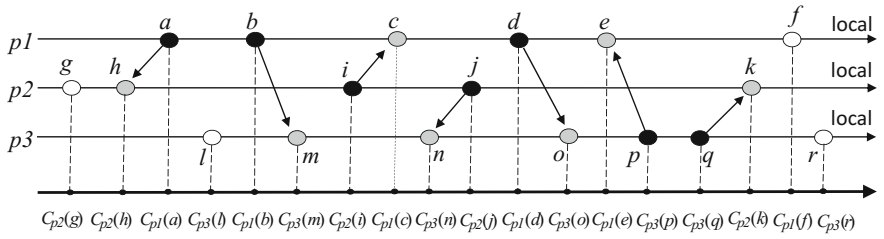


Fig. 4.3 Diagram of events on axes of local times of their occurrence. On the bottom axis projections of local clocks indications are recorded in the ordering of respective events position on axes corresponding to local times of processes *p1*, *p2*, *p3*

Activity of processes *p1*, *p2*, *p3* before time compensations is depicted in Fig. 4.3. Local times of events are projected onto the bottom axis (note that this is not the ordinary geometric image of time). After time compensations, this diagram transforms into the one depicted in Fig. 4.2.

The main fact concerning connection of the precedence relation $\rightsquigarrow$ with the order of the function C values (i.e. compensated time measured by the logical clock) defined above, is the implication $x \rightsquigarrow y \Rightarrow C(x) \leq C(y)$ for all events $x, y \in E(S)$. An inductive proof may be formalized as follows. Write $x \diamond\!\!\xrightarrow{process} y$, if x directly precedes y in the same process or $x = y$. Let x_1, x_2, x_3,… be a finite or infinite sequence of events such that $x_n \diamond\!\!\xrightarrow{process} x_{n+1}$ or $x_n \xrightarrow{message} x_{n+1}$ for each $n \geq 1$. Suppose $x_i \rightsquigarrow x_{i+1} \Rightarrow C(x_i) \leq C(x_{i+1})$ for i less than a certain $n > 1$. If $x_n \diamond\!\!\xrightarrow{process} x_{n+1}$ then $C(x_n) \leq C(x_{n+1})$, because $x \xrightarrow{p} y \Rightarrow C_p(x) \leq C_p(y)$ for any process p and because $C(x) = C_p(x)$ (by definition of function C), for all events x, y. Thus $x_n \rightsquigarrow x_{n+1} \Rightarrow C(x_n) \leq C(x_{n+1})$. If $x_n \xrightarrow{message} x_{n+1}$ then consider two cases:

- If $C_p(x_n) > C_q(x_{n+1})$ then the compensation of time of (slower) receiver process q takes place: $C_q(x_{n+1}) := C_p(x_n) + \Delta x_n x_{n+1}$ for a certain $\Delta x_n x_{n+1} > 0$, in effect of which $C_p(x_n) < C_q(x_{n+1})$ holds. Thus, after the time compensing of the receiver q, implication $x_n \rightsquigarrow x_{n+1} \Rightarrow C_p(x_n) < C_q(x_{n+1})$ holds. Since $C(x_n) = C_p(x_n)$ and $C(x_{n+1}) = C_q(x_{n+1})$ (by definition of function C), we get the implication $x_n \rightsquigarrow x_{n+1} \Rightarrow C(x_n) < C(x_{n+1})$.

- If $C_p(x_n) < C_q(x_{n+1})$ then the sender p is the slower process, so no time compensation takes place and by definition of function C : $C(x_n) < C(x_{n+1})$ we again get the implication $x_n \rightsquigarrow x_{n+1} \Rightarrow C(x_n) < C(x_{n+1})$.

Therefore in any case, the implication $x_n \rightsquigarrow x_{n+1} \Rightarrow C(x_n) < C(x_{n+1})$ is fulfilled, thus by virtue of the induction principle, the implication holds for all n. Because the sequence (a chain) $x_1, x_2, x_3, \ldots$ has been chosen arbitrarily, the implication $x \rightsquigarrow y \Rightarrow C(x) \leq C(y)$ holds for all events $x, y \in E(S)$. This ends the proof.

Now, we are ready to define a ***global timestamp of events***, the main notion for further considerations. Note that in general, reverse implication to $x \rightsquigarrow y \Rightarrow C(x) \leq C(y)$ does not hold (some counterexamples are in Fig. 4.2, where $C(b) < C(j)$ but $b||j$). Moreover, though the logical clock C measures the compensated time, it may happen that $C(x) = C(y)$ and $x \neq y$ for some concurrent x, y, so the mapping C is not one-to-one function (e.g. in Fig. 4.2 $C(c) = C(j)$ and $c \neq j$), thus C does not establish unique representation of events by their timestamps. However if the processes are linearly ordered, e.g. numbered, and the timestamp $C(x)$ is supplemented with a process number in which the event x occurs, then events can be uniquely represented by the richer timestamps, called *global*. So, let $\#(p_x)$ be a unique process number of process p_x, in which the event x occurs (a given event may occur in exactly one process, thus it identifies this process). A pair $\langle C(x), \#(p_x)\rangle$ is called a ***global timestamp of event x*** and let $\preccurlyeq$ denote a relation between global timestamps defined as $\langle C(x), \#(p_x)\rangle \preccurlyeq \langle C(y), \#(p_y)\rangle$ iff $C(x) < C(y)$ or if $C(x) = C(y)$ then $\#(p_x) \leq \#(p_y)$. Obviously $\preccurlyeq$ is a linear order, the so-called lexicographic order and the **one-to-one** injective mapping Γ: $E(S) \rightarrow R \times N$ (N—set of natural numbers) has been established by $\Gamma(x) = \langle C(x), \#(p_x)\rangle$, because $\Gamma(x) = \Gamma(y) \Rightarrow x = y$ for all events x, y. Therefore, Γ establishes a unique representation of events by their global timestamps. This means that events, occurring as partially ordered, may be linearly ordered, using the global timestamp construction. This linear order will be denoted by $\sqsubseteq$ and defined as: $x \sqsubseteq y$ if and only if $\Gamma(x) \preccurlyeq \Gamma(y)$.

As usually $\Gamma(x) \prec \Gamma(y)$ means $\Gamma(x) \preccurlyeq \Gamma(y)$ and $\Gamma(x) \neq \Gamma(y)$.

Evidently, the implication $x \rightsquigarrow y \Rightarrow x \sqsubseteq y$ holds but not the reverse one (see Fig. 4.2).

The representation of events by global timestamps and their linear order will be applied in Sects. 4.3 and 4.4 for implementing distributed mutual exclusion without an „external force" for cooperating computers, that is without a supervisory server.

Remark A widely known theorem states that any partial order relation on a set can be extended to a linear, for some algebraic systems in general, whose basic sets are infinite, also non-enumerable (Kuratowski 1976; Marczewski 1996). The above construction utilizing global timestamps is a very special and straightforward way of the needed extension.

4.3 Distributed Mutual Exclusion Without External Service for Processes—A Method Based on Global Timestamps (Ricart 1981)

As in the former methods with supervisory server and token-ring, it is assumed that there is one protective zone (extension to several of them is straightforward) and during its execution no failure of the system occurs. Processes are cooperating by broadcast of messages among themselves, taking advantage of global timestamps in this cooperation. Figure 4.4 depicts structure of a system with four computers, arbitrarily numbered, that compete for entrance to the protective zone. Similarly to the methods presented in Sects. 3.4 and 3.5, the computers send „***request***", „***permission***" and „***refuse***" messages in the course of the competition, but between themselves, not via any supervisory server.

These messages (of the form of system procedures) are interpreted as follows:

- Process which requests the protecting zone broadcasts the „***request***" message with its current global timestamp as parameter, stores it in its register „*global timestamp of process which has sent request*" and waits for replies from receivers. After having received all the replies, increases the content of its register „*number of granted permissions*" by number of replies „***permission***", and if among them is the „***refuse***" message, passes to the waiting state. The „***refuse***" message, the process receives from process executing protective zone and from processes being in the waiting state, whose global timestamps are smaller (according to the $\preccurlyeq$ order) than its own. If the content of register „*number of granted permissions*" becomes by one smaller than the number of all processes, the process enters the protective zone.

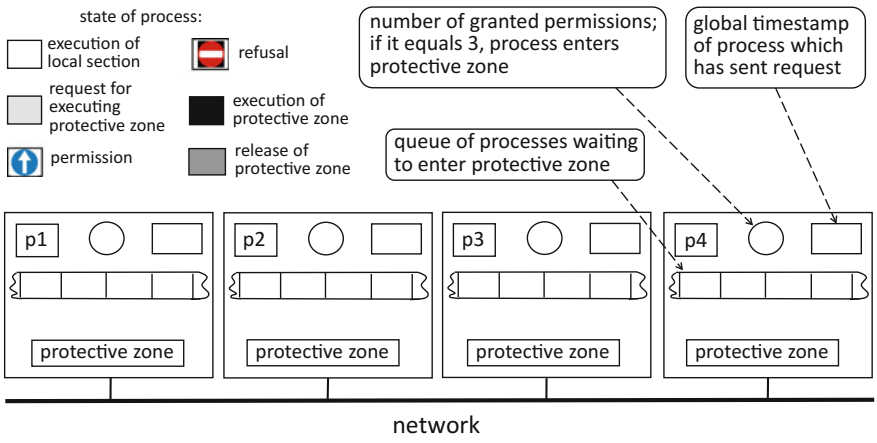


Fig. 4.4 Structure of a system without server managing usage of protective zone; computers are running processes *p1*, *p2*, *p3*, *p4* fixedly numbered as #(*p1*) < #(*p2*) < #(*p3*) < #(*p4*)

- Process which is not executing protective zone and is not in the waiting state , which received the *„**request**"* message, sends back to the sender the *„**permission**"* message.
- Process which is in the waiting state and received the *„**request**"* message, compares its global timestamp with a timestamp in the message received. If its own timestamp is greater, then sends back the *„**permission**"* message, if smaller, then sends back the *„**refuse**"* message and puts the received timestamp into its *queue of processes waiting to enter the protective zone*.
- Process which is executing the protective zone , which received the *„**request**"* message, sends back the *„**refuse**"* message and puts the received timestamp into its queue of processes waiting to enter the protective zone.
- *Process which leaves the protective zone* , sends the *„**permission**"* message to processes whose global timestamps are in its queue of processes waiting to enter the protective zone and deletes content of this queue as well as content of the registers „*number of granted permissions*" and „*global timestamp of process which has sent request*".

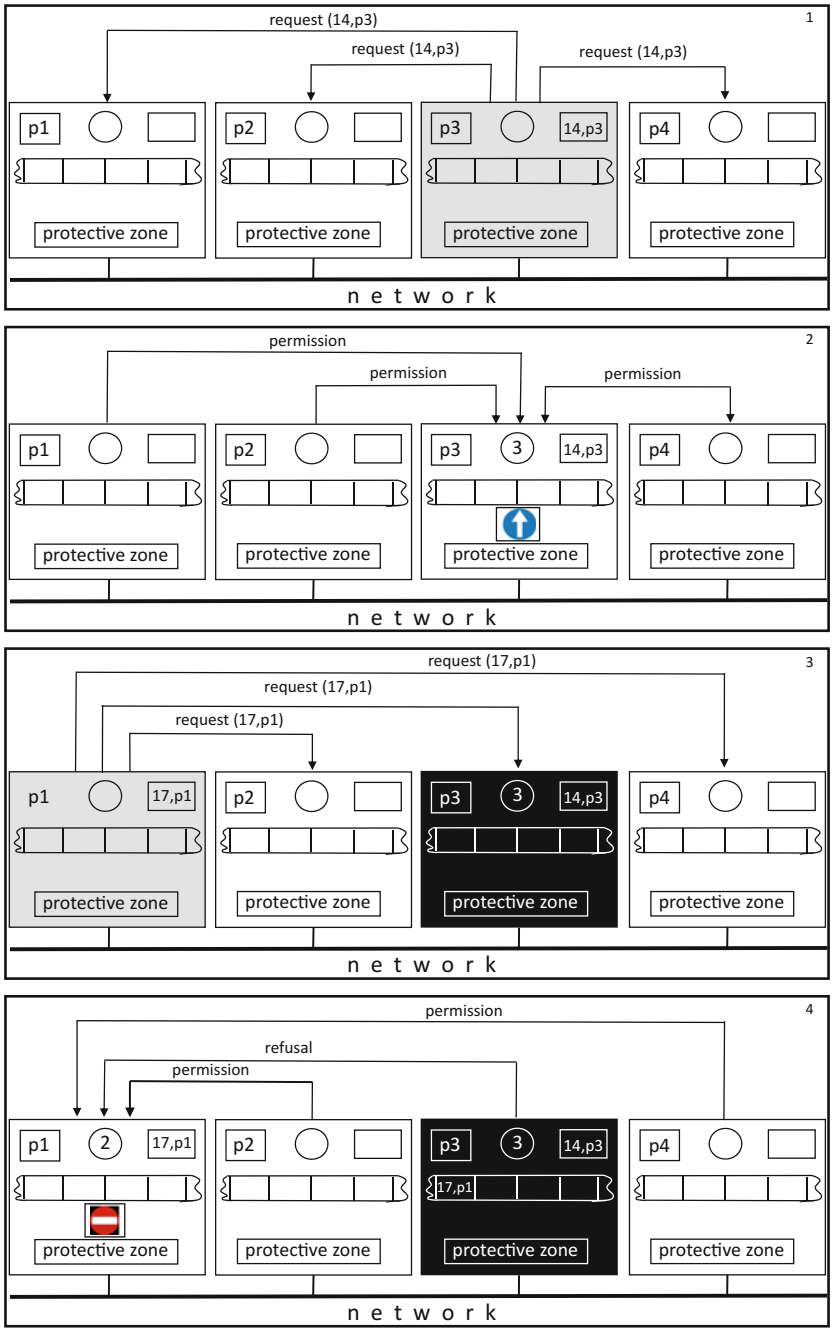
Table 4.4 presents an example of mutual exclusion in operation, accomplished by exchange of messages between processors competing for a resource protected by a protective zone, without service from a supervisory server. For brevity of notation, process' numbers are identified with their names, e.g. „*p1*" will be written instead of „#(*p1*)". The processes are numbered as: #(*p1*) = 1, #(*p2*) = 2, #(*p3*) = 3, #(*p4*) = 4.

Remark Remember that the time management must take into account the time compensation during message passing—when message reception precedes its dispatch according to local clock indications of the sender and receiver. We do not discuss problems of timestamp progress and a mechanism of vector clocks [cf, for instance (Saxena 2003; Kuz et al. 2016)]. For the purpose of the presentation here, it suffices to assume that the value $C_p(x)$ of the current timestamp of the process p increases by a certain value (e.g. a pulse duration of the clock) on every occurrence of an event x—with regard time compensation, if needed.

4.4 Distributed Mutual Exclusion Without External Service for Processes—A Method Based on Vectors of Global Timestamps (Czaja 2012)

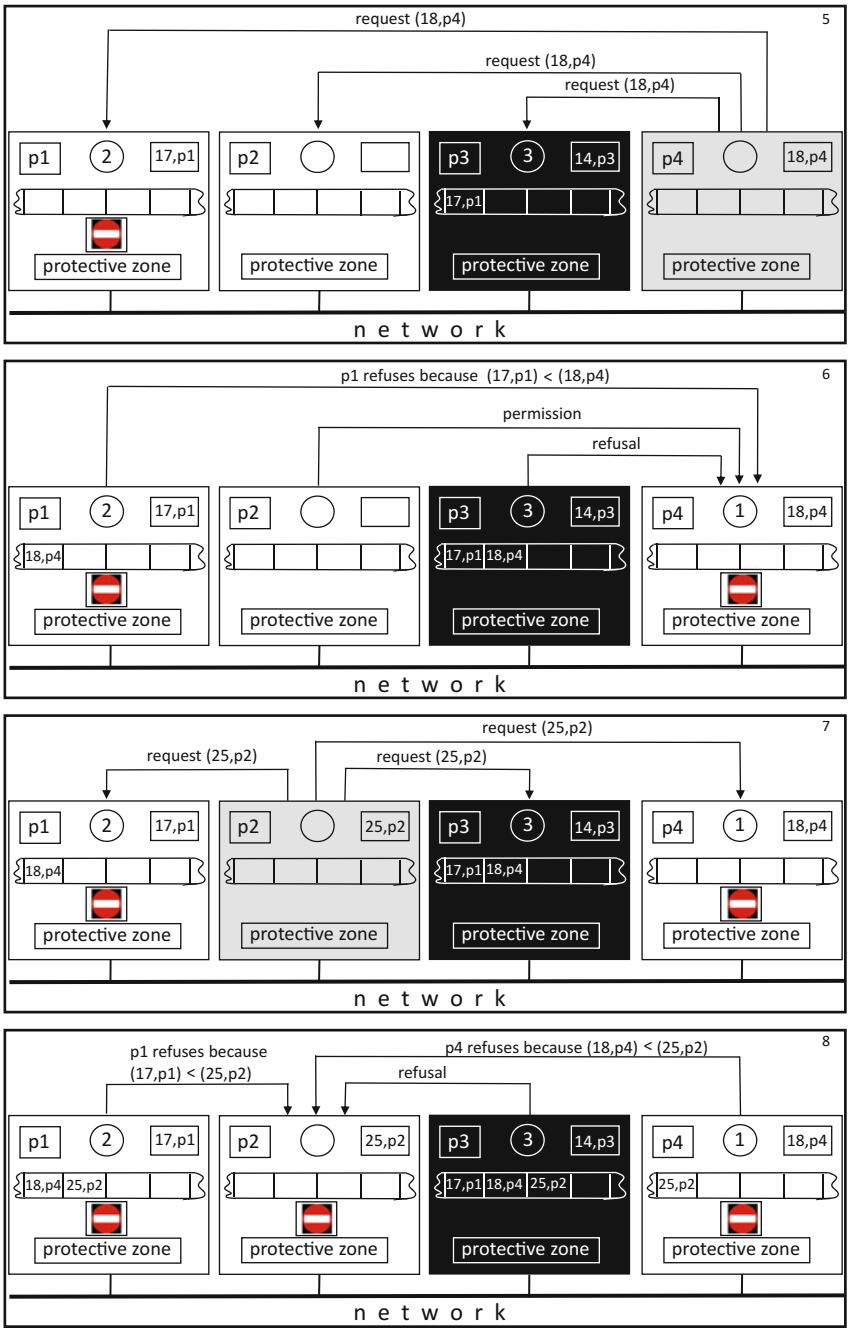
In this protocol, functionally somewhat similar to that presented in Sect. 4.3 but differently designed, the global timestamps are also used. Each computer is equipped with a vector of timestamps corresponding to computers of the system.

Table 4.4 Exemplary run of a system with four computers striving to enter protective zone



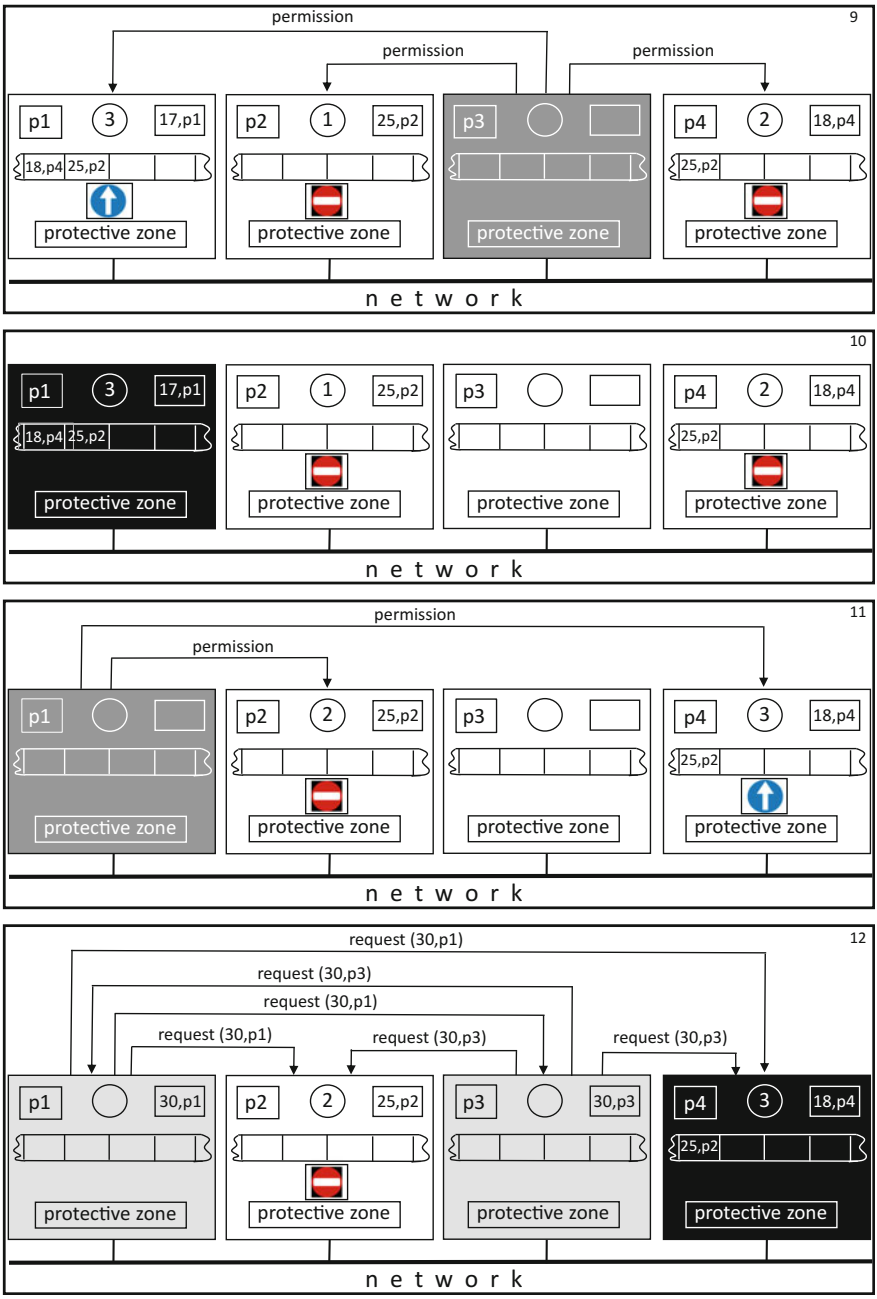
(continued)

Table 4.4 (continued)



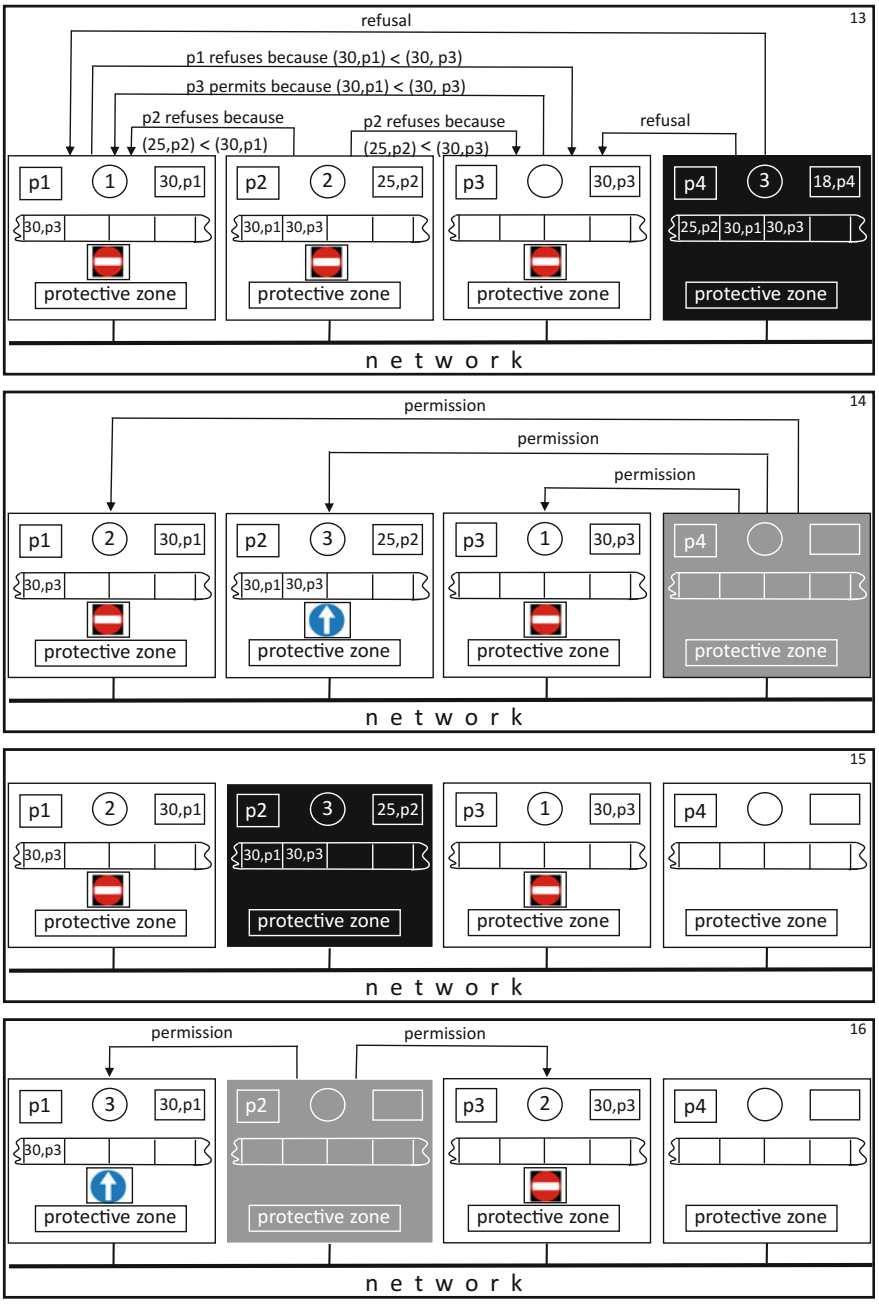
(continued)

Table 4.4 (continued)



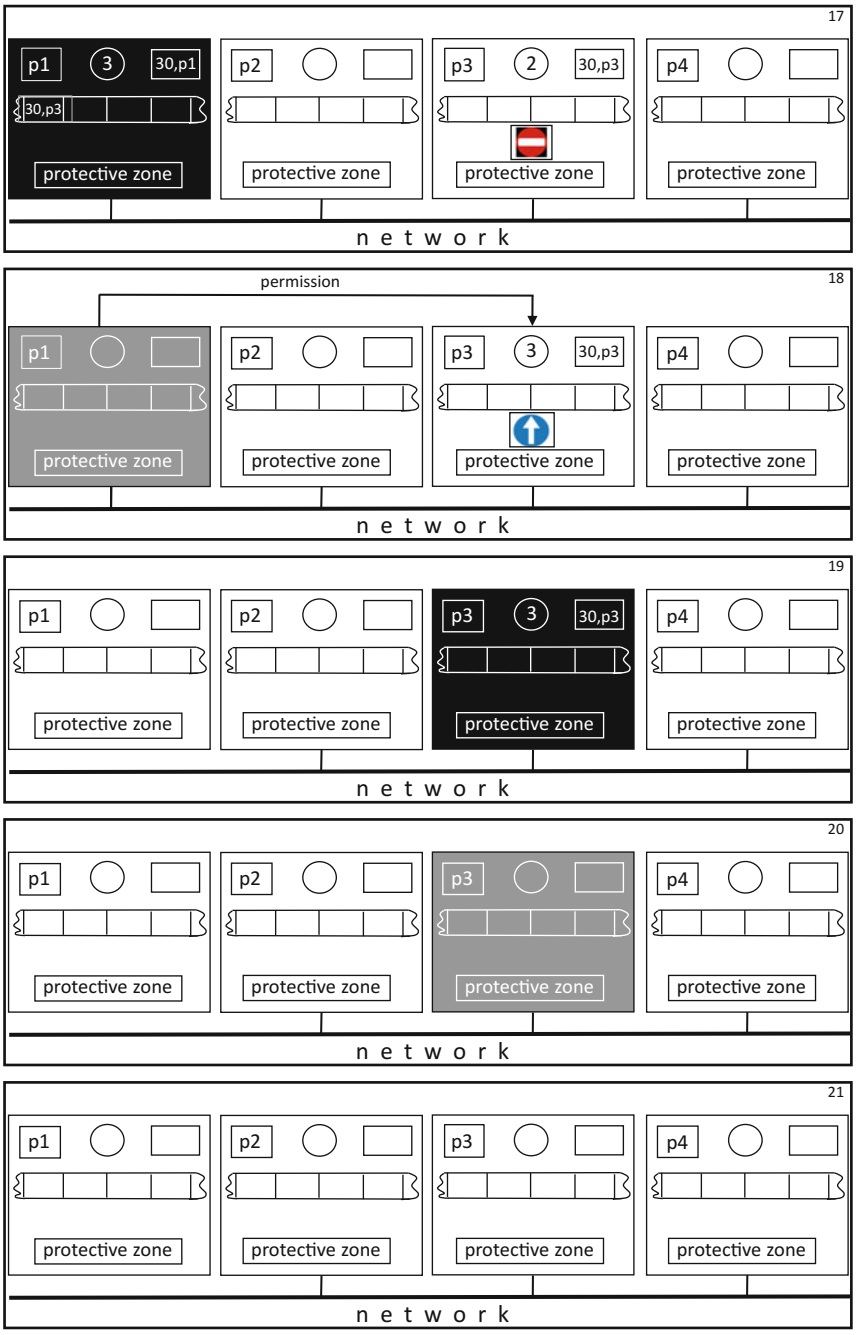
(continued)

Table 4.4 (continued)



(continued)

Table 4.4 (continued)



Processes are updating the vectors in the course of system activity and make decision on the basis of their content, whether to enter into protective zone or wait. Also usage of the artificial value ∞, greater than any possible timestamp, makes this protocol different than that in Sect. 4.3.

The following assumptions are admitted:

1. sequential computers work in parallel asynchronously and are numbered 1,2,…, n;
2. writing and reading to/from memory is governed by the memory manager of each computer;
3. each request to the protocol for the protective zone, delivers a current global timestamp to the requesting process;
4. computer of number i keeps vector $\overrightarrow{ri} = [r_{i1}, r_{i2}, \ldots, r_{in}]$ of variables r_{ik} allocated in its physical memory; it stores its current timestamp in the component r_{ii} when requesting for the protective zone, then fetches values of components r_{kk} $(k \neq i)$ from remaining computers and stores them in variables r_{ik} of its vector $\overrightarrow{ri}$. Figure 4.5 depicts location of vectors of timestamps in the local memories;
5. initially all variables r_{ij} contain ∞ where $\infty > x$ for any number x;
6. by $min(\overrightarrow{ri})$ is denoted the least value of the components in the vector $\overrightarrow{ri}$;

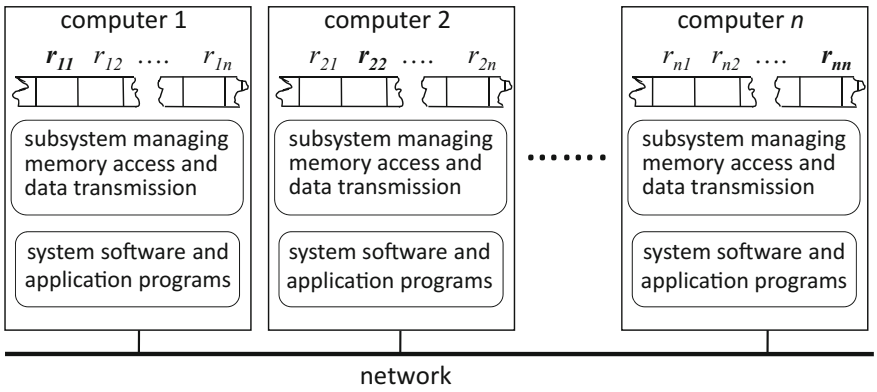


Fig. 4.5 Structure of distributed system of n computers with vectors $\overrightarrow{ri}(i = 1, 2, \ldots, n)$ of timestamps allocated in local memories

Computer of number i = 1, 2,…, n, using the protocol depicted as a transition graph in Fig. 4.6 for exclusive access to a protected resource, passes through the following states:

W_i – execution of local section

B_i – import of current timestamps stored in variables r_{kk} of remaining computers; execution of n -1 assignments $r_{ik} := r_{kk}$ $(k \neq i)$; test of condition $r_{ii} > min(\overrightarrow{ri})$

Y_i – refusal to perform protective zone (waiting state)

R_i – execution of protective zone

G_i – release of protective zone

The set of states of the ith computer: $\Omega_i = \{W_i, B_i, Y_i, R_i, G_i\}$:

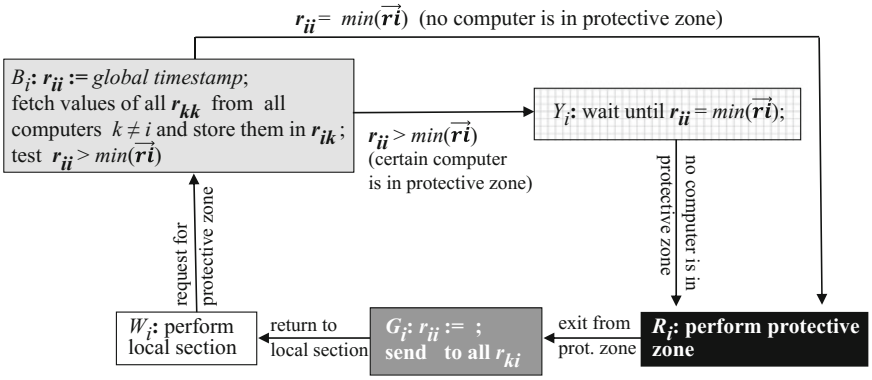


Fig. 4.6 The distributed mutual exclusion protocol performed by computer of number i in the cycle from request for protective zone till its release

Denotations:

- Let $Q_i \rightarrow Q_i'$ mean: computer of number i passes from a state $Q_i \in \Omega_i$ to the next state $Q_i' \in \Omega_i$ in the transition graph in Fig. 4.6. Note that transitions $B_i \rightarrow R_i$ and $Y_i \rightarrow R_i$ are possible if and only if $r_{ii} = min(\overrightarrow{ri})$, thus due to steady growth of global timestamp as a strictly increasing function of time, at most one computer may perform protective zone at a time. A formal proof is given further.
- Set of global states $\mathbf{\Omega} = \Omega_1\Omega_2 \times \cdots \times \Omega_n$ satisfying: if $\vec{Q} = [Q_1, Q_2, \ldots, Q_n] \in \mathbf{\Omega}$ then $\neg\exists i, k : (i \neq k \wedge Q_i = R_i \wedge Q_k = R_k)$.
- Initial state: $\overrightarrow{Qinit} = [W_1, W_2, \ldots, W_n]$ with $r_{ij} = \infty$ for every computer $i, j = 1, 2, \ldots, n$.
- For $\vec{Q} = [Q_1, Q_2, \ldots, Q_n] \in \mathbf{\Omega}$ and $\overrightarrow{Q'} = [Q_1', Q_2', \ldots, Q_n'] \in \mathbf{\Omega}$ let $\vec{Q} \rightrightarrows \overrightarrow{Q'}$ mean: there exists a computer of number i such that $Q_i \rightarrow Q_i'\overrightarrow{Q'}$ is the next global state following $\vec{Q}$.

It follows from the transition graph in Fig. 4.6 that for any computer of number i:

1. Storing a timestamp in register r_{ii} proceeds only at the state B_i of computer of number i; r_{ii} retains this value until the transition $R_i \rightarrow G_i$ takes place.
2. Storing ∞ in register r_{ii} and sending to r_{ki} of remaining computers proceeds only at the state G_i. Thus, from point 1 follows that r_{ii} decreases its value only at the state B_i.
3. Global states are exactly those reachable from the initial state $\overrightarrow{Qinit} = [W_1, W_2, \ldots, W_n]$.
4. Because computation of $min(\overrightarrow{\boldsymbol{ri}})$ at the state B_i of computer of number i takes place on completion of fetching values of r_{kk} from remaining computers, the order of entering computers into the protective zone does not depend of the transmission latency. This is the FCFS order (First Come, First Served) due to the steady growth of the global timestamps.

Table 4.5 presents an exemplary run of a four computer system using the protocol depicted in Fig. 4.6. This is the succession of global states:

$$[W_1, W_2, W_3, W_4] \rightrightarrows [B_1, W_2, B_3, W_4] \rightrightarrows [Y_1, W_2, R_3, W_4] \rightrightarrows [Y_1, W_2, R_3, B_4] \rightrightarrows [Y_1, B_2, R_3, Y_4] \rightrightarrows$$
$$[Y_1, Y_2, G_3, Y_4] \rightrightarrows [R_1, Y_2, W_3, Y_4] \rightrightarrows [G_1, Y_2, W_3, Y_4] \rightrightarrows [W_1, Y_2, W_3, R_4] \rightrightarrows [W_1, Y_2, W_3, G_4] \rightrightarrows$$
$$[W_1, R_2, W_3, W_4] \rightrightarrows [W_1, G_2, W_3, W_4]$$

Global timestamps, i.e. pairs of numbers, are coded by single numbers—for brevity. That is why the order $\leq$ (and $<$) instead of $\preccurlyeq$ will be used when comparing global timestamps.

Correctness of the protocol in Fig. 4.6, which is to assure mutually exclusive execution of protective zone in distributed systems, enjoys almost straightforward formal verification:

Theorem 4.4.1 *At none of global state, two distinct computers using the protocol in Fig. 4.6 can perform the same protective zone.*

Proof Suppose on the contrary, that in a global state $\vec{Q} = [Q_1, Q_2, \ldots, Q_n] \in \boldsymbol{\Omega}$ computers of number i and k perform protective zone. Then $r_{ii} = min(\overrightarrow{\boldsymbol{ri}})$ and $r_{kk} = min(\overrightarrow{\boldsymbol{rk}})$ in the local states Q_i and Q_k of the computers. By definition of the global timestamps: $r_{ii} \neq r_{kk}$ because events of request for protective zone are distinct, so, their global timestamps (i.e. values of r_{ii} and r_{kk}) are also distinct—due to the one-to-one function Γ (Sect. 4.2). But because of actions at the states B_i, B_k of the protocol in Fig. 4.6, equations $r_{ik} = r_{kk}$ and $r_{ki} = r_{ii}$ hold. Since r_{ii} and r_{kk} are minimal in vectors $\overrightarrow{\boldsymbol{ri}}$ and $\overrightarrow{\boldsymbol{rk}}$ respectively, so, $r_{ii} \preccurlyeq r_{ik}$ and $r_{kk} \preccurlyeq r_{ki}$, therefore $r_{ii} \preccurlyeq r_{kk}$ and $r_{kk} \preccurlyeq r_{ii}$ which implies $r_{ii} = r_{kk}$ (by antisymmetry of $\preccurlyeq$—by definition of the order $\preccurlyeq$ between global timestamps—Sect. 4.2)—a contradiction! □

Another important property of this protocol is stated in Theorem 4.4.2, whose proof needs the following:

Lemma *In all computers whose timestamps vectors, at a certain global state* $\vec{Q}$*, contain at least one component less than* ∞*, the minimal components of such vectors are equal.*

Proof Let $min(\overrightarrow{\boldsymbol{ri}}) < \infty$ in a computer of number i. Then at the global state $\vec{Q}$, exactly one computer, say of number k, either is performing protective zone or is ready to do this, therefore $r_{kk} = min(\overrightarrow{\boldsymbol{rk}})$. According to the protocol in Fig. 4.6 $r_{ik} = r_{kk}$ and r_{ik} is the least component of vector $\overrightarrow{\boldsymbol{ri}}$ at the state $\vec{Q}$, thus $min(\overrightarrow{\boldsymbol{ri}}) = min(\overrightarrow{\boldsymbol{rk}})$. □

The next theorem states the fair order (i.e. FCFS) of entering the protective zone.

Theorem 4.4.2 *Computers are entering the critical section in the order of their requests. This order is independent of the data transmission latency.*

Proof Let $\vec{Q} = [Q_1, \ldots, Q_i, \ldots, Q_j, \ldots, Q_n] \in \boldsymbol{\Omega}$ be a global state with local states Q_i, Q_j each of them either B_i, B_j or Y_i, Y_j thus $r_{ii} < \infty$, $r_{jj} < \infty$. It is to be proved that if computer of number i had reached its local state Q_i before the computer of number j has reached its local state Q_j, i.e. if $r_{ii} < r_{jj}$, then state R_i will be reached by computer i before computer j reaches state R_j. By above Lemma, $min(\overrightarrow{\boldsymbol{ri}}) = min(\overrightarrow{\boldsymbol{rj}})$ at the state $\vec{Q}$, and according to the protocol in Fig. 4.6, variables r_{ii}, r_{jj} are not changing their values until computers of number i and of number j have not reached their local states G_i, G_j. Thus, if eventually, a global state $\overrightarrow{Q'} = [Q'_1, \ldots, Q'_i = R'_i, \ldots, Q'_j, \ldots, Q'_n]$ (nearest to $\vec{Q}$) has been reached from $\vec{Q}$ then $r_{ii} = min(\overrightarrow{\boldsymbol{ri}})$ at $\overrightarrow{Q'}$. But if a global state $\overrightarrow{Q''} = [Q''_1, \ldots, Q''_i, \ldots, Q''_j = R_j, \ldots, Q'_n]$ had been reached from $\vec{Q}$ before the state $\overrightarrow{Q'}$, then $r_{jj} = min(\overrightarrow{\boldsymbol{rj}})$ at $\overrightarrow{Q''}$ would hold.

Thus, $min(\overrightarrow{\boldsymbol{ri}}) \leq r_{ii} < r_{jj} = min(\overrightarrow{\boldsymbol{rj}})$ at the state Q'', because values of r_{ii} and of r_{jj} at this state are the same as at $\vec{Q}$ and $min(\overrightarrow{\boldsymbol{ri}}) = min(\overrightarrow{\boldsymbol{rj}})$—a contradiction! Now, note that, in accordance with the protocol in Fig. 4.6, the value of $min(\overrightarrow{\boldsymbol{ri}})$ is established only when values of r_{kk} have been completely transmitted from all computers $k \neq i$ to computer i and stored in r_{ik}. This value does not depend on duration of these transmissions nor on their order. Therefore the order of entering computers into the protective zone is independent of transmission latency but depends only on the order of their requests. □

Figure 4.7a, b illustrate independence of value $min(\overrightarrow{\boldsymbol{ri}})$ from ordering and duration of values transmission, when the system's evolution, shown in Table 4.5, has reached state 2: $[B_1, W_2, B_3, W_4]$. The symbol $i\uparrow(r_{kk})_k$ means: "computer of number k sends value of r_{kk} up to computer of number i" and the symbol $k\downarrow(r_{ik})_i$ means: "computer of number i receives a value sent by computer of number k and stores it in r_{ik}. The transmitted values are annotations on respective arrows.

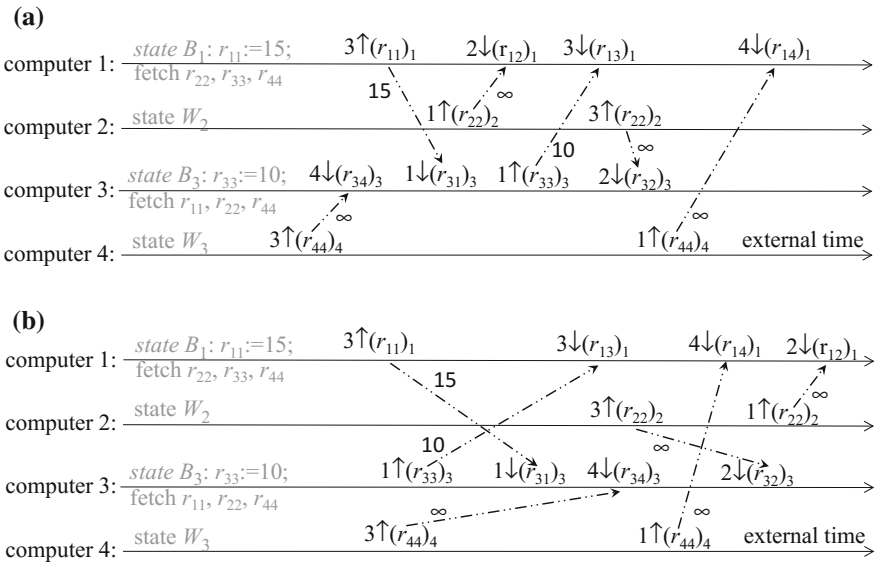
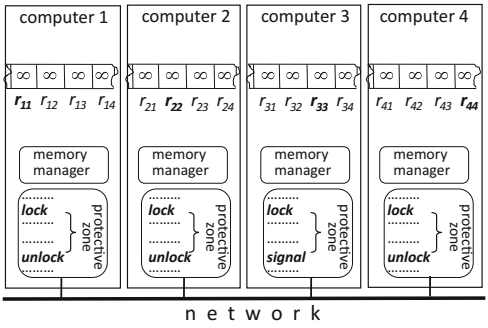


Fig. 4.7 **a** Diagram of global state [B_1,W_2, B_3,W_4] and message transmissions between computers. **b** Diagram of the same global state and $min(\overrightarrow{\boldsymbol{ri}})$ as in **a** but different ordering and duration of values transmissions

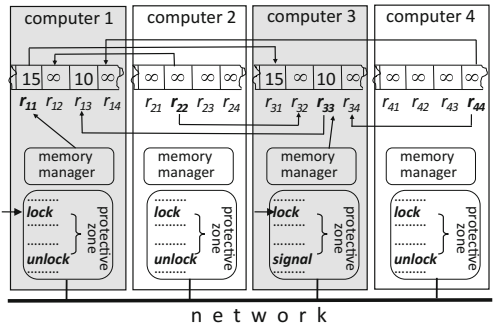
More remarks on the protocol in Fig. 4.6:

1. **Consumption of time**. At the state B_i, computer of number i when requesting for protective zone, sends message „*send me value of* your r_{kk}" to all $n - 1$ remaining $k \neq i$ computers, and waits for delivery. In the worst case the message reaches all destinations one after one as well as responses arrive one after one, which takes $2(n - 1)$ transmissions. Next, at the state G_i, on release of protectibe zone, the computer broadcasts ∞ to all r_{ki} of all $n - 1$ remaining computers, which takes, in the worst case, $n - 1$ transmissions.
2. **Failure**. If a faulty computer of number k permanently delivers incorrect timestamp in r_{kk} to remaining computers that fetch it at the state B_k of the protocol, then their behaviour depends on this value. If, for instance, r_{kk} is small enough to make $min(\overrightarrow{\boldsymbol{ri}})$ smaller than r_{ii}, then computer of number i enters the waiting state Y_i and will remain there forever: a starvation! But if computer of number k delivers to computer of number i value of r_{kk} satisfying equality $r_{ii} = min(\overrightarrow{\boldsymbol{ri}})$, and after a while it delivers to computer of number j value of r_{kk} satisfying equality $r_{jj} = min(\overrightarrow{\boldsymbol{rj}})$, then computer of number j may enter the protective zone before computer of number i leaves it: violation of mutual exclusion! To solve this problem, the protocol in Fig. 4.6 would require suitable supplementation. The failure issues is the subject of Chap. 7.

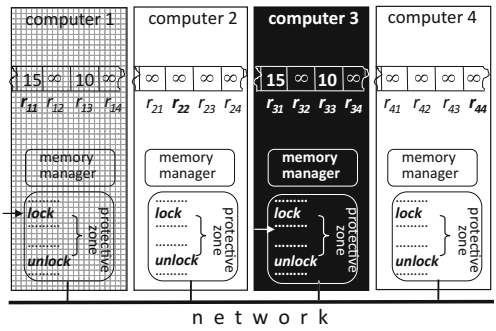
Table 4.5 Exemplary run of a system with four computers using protocol depicted in Fig. 4.2. Background of the computers corresponds to their local states as pictured in the protocol



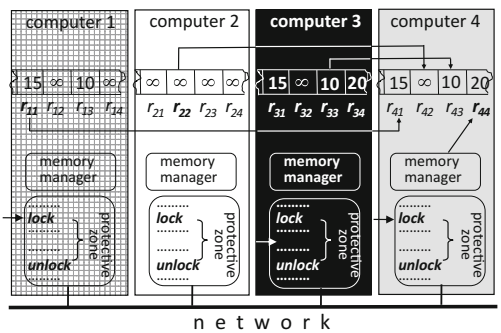
state 1
$[W_1, W_2, W_3, W_4]$



state 2
$[B_1, W_2, B_3, W_4]$



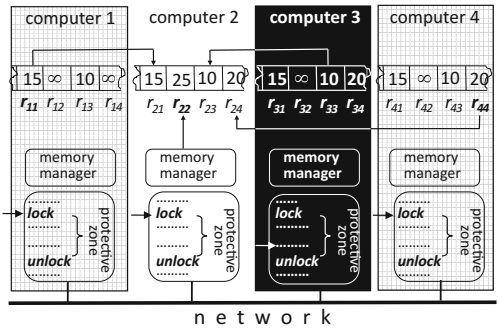
state 3
$[Y_1, W_2, R_3, W_4]$



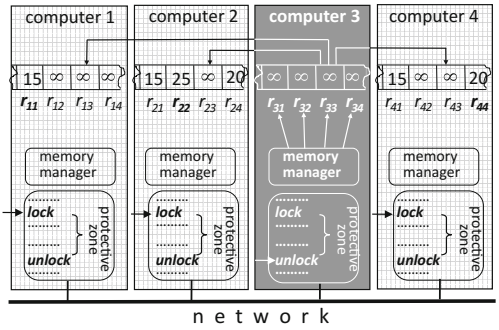
state 4
$[Y_1, W_2, R_3, B_4]$

(continued)

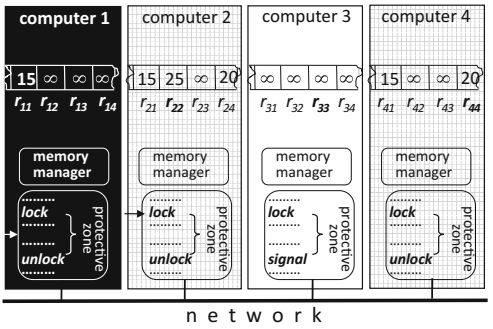
Table 4.5 (continued)



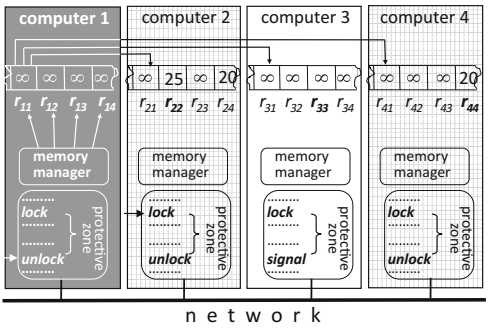
state 5
$[Y_1, B_2, R_3, Y_4]$



state 6
$[Y_1, Y_2, G_3, Y_4]$



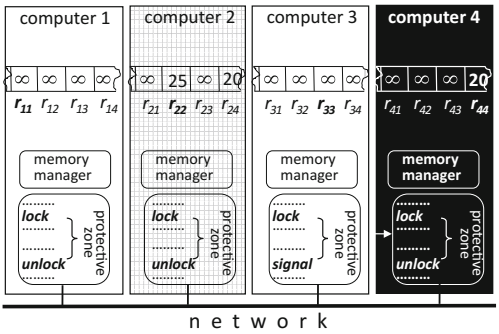
state 7
$[R_1, Y_2, W_3, Y_4]$



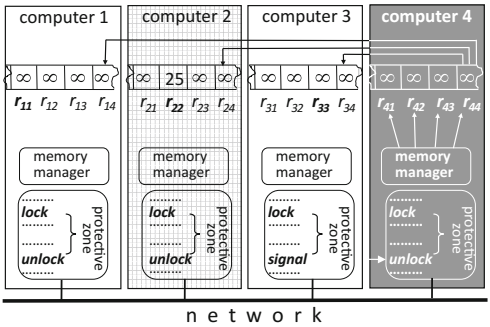
state 8
$[G_1, Y_2, W_3, Y_4]$

(continued)

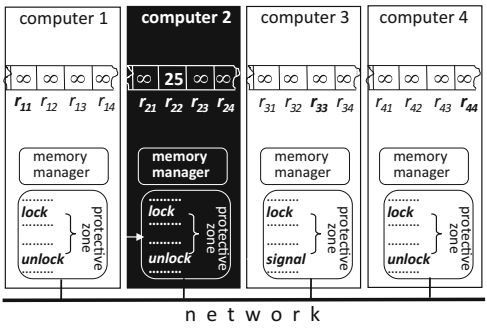
Table 4.5 (continued)



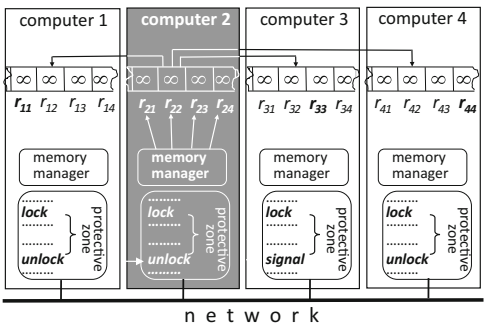
state 9
$[W_1, Y_2, W_3, R_4]$



state 10
$[W_1, Y_2, W_3, G_4]$



state 11
$[W_1, R_2, W_3, W_4]$



state 12
$[W_1, G_2, W_3, W_4]$

3. **Elimination of busy waiting**. To make computers perform some computation instead of useless permanent checking $r_{kk} = min(\overrightarrow{\boldsymbol{rk}})$ at the states Y_k, a computer which exits protective zone (i.e. at the state G_k) might fetch values of r_{kk} from remaining computers and send a permission to execute protective zone, for such computer, whose vector contains the minimal value from among the fetched ones. This would, however, increase number of transmissions.
4. **Representation by Petri nets (for those familiar with this theory)**. The control flow in the protocol depicted in Fig. 4.6, applied by a system of four computers, may be modelled by ordinary Place/Transition Petri net (P/T) (Petri 1966; Reisig 1985) (even by Condition/Event Petri nets), like that in Fig. 4.8. In this case, however, one should notice that temporal aspects, i.e. the essential role of global timestamps, cannot be expressed in the P/T nets formalism. That is why the control (the token) from the state B_i (i = 1, 2, 3, 4) may pass either to the state R_i or Y_i, provided that the shared synchronizer S holds a token, i.e. none computer is performing the protective zone (is not at its state R_k). Moreover, the existence of the shared synchronizer S (playing role of a semaphore) negates the idea of „distributed mutual exclusion without external service for processes". In order to realize this idea when modeling by Petri nets, they should be equipped with inhibitor arcs——o (Peterson 1981; Christensen 1993) leading from places R_i (execution of protective zone) to transitions, from which the ordinary arcs ⟶, go out and lead to places representing protective zone. Such extended Petri net is depicted in Fig. 4.9. Notice that in this figure, some junction points • on inhibitor arcs going out from the same place R_i to two distinct transitions, have been used, so that to avoid excessive entanglement of such arcs (representing communication lines). Also, the net places are here understood as units where some actions are being performed, whereas the transitions—as passing control from actions to successive actions in the protocol. The readers familiar

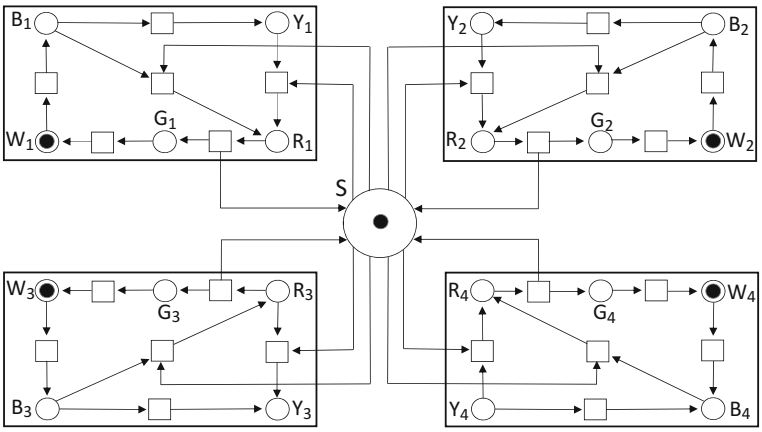


Fig. 4.8 P/T Petri net representing four computers competing for a protective zone R_i; the computers are synchronized by means of external service—a shared synchronizer S

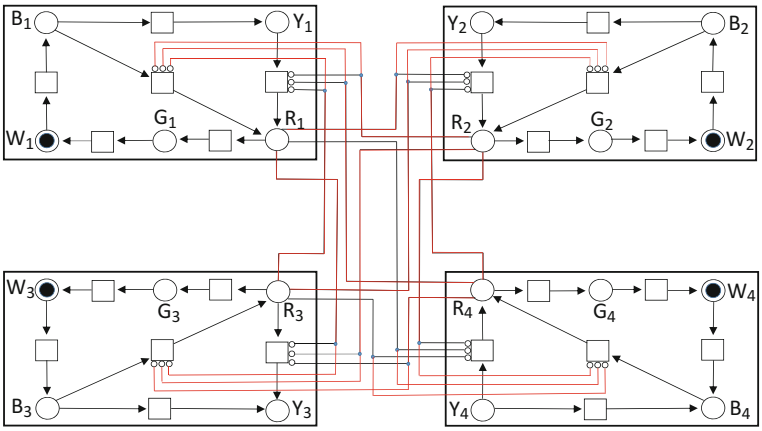


Fig. 4.9 Similar Petri net but without external service—for the price of introducing inhibitor arcs

with Petri net theory know that expressive power of P/T nets with inhibitors (with more than one such arc) is essentially higher than without them: it reaches the Turing power, which is not the case with „pure" P/T nets.

References

Christensen, S., & Hansen N. D. (1993). Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. In Application and theory of Petri Nets, Lecture Notes in Computer Science book series (LNCS, vol 691).

Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing, Springer-Verlag, 3,* 146–158.

Czaja, L. (2012). *Exclusive Access to Resources in Distributed Shared Memory Architecture*Volume. *Fundamenta Informaticae, 119*(3–4), 265–280.

Gusella, R., & Zatti, S. (1989). The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering, 15*(7), 847–853.

Kuratowski, K., & Mostowski, A. (1976). *Set Theory, with an Introduction to Descriptive Set Theory* (2nd ed., Studies in Logic and the Foundations of Mathematics—Vol 86) Hardcover—26 Feb 1976.

Kuz, I., Manuel M. T., Chakravarty & Gernot Heiser. (2016). *A course on distributed systems* COMP9243, 2016.

Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Communications of the ACM, 21*(7), 558–565.

Marczewski (Spilrajn) K. (1930). *Sur l'extension de l'ordre partiel*, FM (Vol. 16, pp. 386–389). in Edward Marczewski, Collected Mathematical Papers, IM PAN 1996.

Mills, D. L. (1991). Internet time synchronization: The network time protocol. *IEEE Transactions on Communications, 39*(10), 1482–1493.

Network Time Protocol (version 2), specification and implementation. (1989). DARPA Network Working Group Report RFC-119, University of Delaware, Sep 1989.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Inc.

Petri, C. A. (1996). *Communication with automata*, Report RADC TR-65-377, Applied Data Research (Vol. 1, Suppl. 1), Contract AF 30 Princeton N.J.

Reisig, W. (1985). *Petri Nets. An Introduction*, Monographs on Theoretical Computer Science, Springer-Verlag.

Ricart, G., & Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM, 24*(1), 9–17.

Saxena, P. C., & Rai, J. (2003). A survey of permission-based distributed mutual exclusion algorithms. *Computer Standards & Interfaces*, *25*(2), 159–181. May 2003.

Chapter 5
Interprocess Communication

5.1 Basic Problems of Communication

So far, some problems specific for distributed systems have been presented, such as correctness of transactions, of banking in particular, resource sharing and protection, pathological phenomena (deadlock, starvation), synchronization of clocks and processes and, in general, the issues of time and coordination. In such problems an essential role plays interprocess communication, performed by computers connected in a network, which ensures hardware infrastructure for distributed system. It had been assumed that the communication mechanisms are already delivered by hardware and software, so, there were no need to get into their implementation details, which might hinder explaining the essence of the communication problems. The processes were just capable of sending and receiving messages. Communication problems belong to science and engineering area wider than distributed systems characterized as in Chap. 2: to the telecommunication and computer networks—the areas using techniques elaborated there. Now, selected problems of communication in networks, essential for computer distributed systems, are presented on a certain generality level, in order to emphasise principles independent of technical solutions in existing implemented systems. For instance, neglected is number of bits for particular fragments of transmitted packets, the methods of naming resources and communication participants, mapping the names onto addresses or routing of messages. So, it is assumed that processes, ports, sockets, channels, are named and addressed. Also, we do not enter into structure details of particular communication protocols. A presentation of such problems is easily available in writings on computer networks, e.g. Comer (2015), Dolińska (2005), Sportack (2004), [Tanenbaum and Wetherall 2011] and more others. Thus, we confine ourselves to presentation on several examples, the role and names of some protocols and their functions, collected in the so-called layer models of arranging message transmission. Such models have been adopted as international standards, independent of their concrete implementation.

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_5

Communication between a pair of processes encompasses:

- transmission of data from the environment of sending process, to the environment of receiving process through a common transmission channel; processes of sender and receiver are often performed by computers of different hardware and software architecture;
- mode of communication: synchronous or asynchronous (like the mechanisms „!", „?" and „S", „R" in Chap. 1), connection oriented or connectionless, peer-to-peer, multicast or broadcast;
- conversion of transmitted data from the format used by sender into format used by receiver.

In contrast to the previous chapters, in this chapter the differences between architecture of computers are not neglected. This requires transforming representation of messages dispatched onto representation which enables correct, i.e. identical „understanding" (interpretation) by both sides: the sender and receiver, differing often in hardware and software. This representation is, usually, a certain standard commonly adopted in distributed systems and more general—in systems based on networks. Successive stages of transforming of the sender's representation of data structures onto a standard („flat") representation for transmission through communication channels and—farther—onto the receiver's representation, are realized by communication protocols. Actions of transformation and packing of data to be dispatched, is called ***marshalling***, whereas unpacking of data to be received—***unmarshalling***.

„marshalling"—the term taken from railway terminology (similarly to the semaphore), which means setting railway cars in a desired order in the marshalling yard.

This is accomplished (depending of implementation) in kernels of operating systems of communication participants, in runtime systems of programming languages equipped with communication mechanisms or in special protocols. Other issues of interprocess communication presented in this chapter are various modes of transmission, like synchronous, asynchronous, connected-oriented, connectionless, as well as layer models of protocol sets, like OSI/RM and ATM standards and broadcast and multicast communication (applied in the distributed mutual exclusion presented in Chap. 4).

For a communication to appear it is necessary:

- to establish a connection between communicating processes;
- to ensure a buffering (queuing) of messages between the sender and receiver,
- to apply a certain communication protocol that ensures marshalling and unmarshalling; to this end:

  - the sender converts a message from its language onto a language defined by rules of the protocol, and so translated sends to the receiver;
  - the receiver converts the delivered message onto its language and, possibly, returns a confirmation;

- to perform transmission of message with marks of its begin and end, signals of synchronization, information of encoding of numbers (the sender's arithmetic) and alphanumeric characters, of data structures necessary for their reproduction on the receiver's side;
- to signal the end of transmission and to release its route.

5.2 Tasks of Communication Protocols—Examples

A communication protocol is a set of rules of coding (by sender) and decoding (by receiver) data traveling through communicating channels. In other words, this is a formal description of a form (syntax) and meaning (semantics) of messages transmitted in computer networks (in general—telecommunication networks) and rules of their sending and reception. The rules determine a kind and mode of communication, like synchronous or asynchronous, connection-oriented or connectionless, with or without confirmations, one-way or two-way. A particular protocol is destined for realization of some functions in interprocess data transmission. So, their suitable collection makes possible communication between devices of various construction, e.g. computers of various architecture. Apart from encoding and decoding messages and communication mode determination, to the tasks of some protocols belong determination of routes of the message, the so-called routing, as well as identification of receiver's process and its site (port, socket), whence the message is to be fetched from and also assurance of transmission correctness. A protocol is usually a component of a "protocol suite" realizing entire activity of message passing. It passes the outcome of its activity to successive protocol of the suite, for realizing next functions, etc. Such multilevel, layered structure of the protocol suite, is presented—in action—further in this chapter. In this section, several examples of "marshalling" function of protocols in communication between devices of different architectures is presented.

Example 1 In a computer *A,* an integer number is represented by a sequence of 40 bits (40 bit word) $e_0e_1...e_{39}$ ($e_i \in \{0, 1\}$, $i = 0, 1, 2, \ldots, 39$) interpreted (by the arithmetic unit of the computer *A*) as the arithmetic value:

$$N_A = -e_0 2^{39} + e_1 2^{38} + e_2 2^{37} + \cdots + e_{38} 2^1 + e_{39} 2^0$$

In a computer *B,* an integer number is represented by a sequence of 17 bits (17 bit word) $e_0e_1...e_{16}$ ($e_i \in \{0, 1\}$, $i = 0, 1, 2, ..., 16$) interpreted (by the arithmetic unit of the computer *B*) as the arithmetic value:

$$N_B = \begin{cases} e_0 2^{16} + e_1 2^{15} + \cdots + e_{15} 2^1 + e_{16} 2^0 & \text{if } e_0 = 0 \\ -\left(e_0 2^{16} + e_1 2^{15} + \cdots + e_{15} 2^1 + e_{16} 2^0\right) & \text{if } e_0 = 1 \end{cases}$$

If the computer A has sent to B a sequence of bits representing number N_A in the *A's* architecture, then the computer *B,* after reception of this sequence would truncate some bits from N_A and remaining 17 bits would interpret as a numeric value N_B—completely different than N_A. To avoid this, computer A transforms the sequence of bits representing N_A onto a certain form „jointly understood" by A and B and sends the transformed sequence to B together with information that the sequence should be interpreted as an integer number. This „jointly understood" intermediate form is determined by a communication protocol between A and B. In computer A the so-called complementary arithmetic is applied, while in computer B the so-called arithmetic of module-sign (computers of such arithmetic had been in use indeed (Czaja 1971; Fiałkowski and Swianiewicz 1962). A little more complicated is representation of ***real*** numbers and translation between them.

Example 2 In a computer C some data structures (tables, trees, stacks, queues, lists, graphs, etc.) are represented in completely different way than in computer D. They must be converted onto a form, properly interpreted („understood") by both computers. In the transmission channel, a data structure is transmitted in a „flattened" form, that is, as a sequence of bits, along with information what type of structure the sequence represents. Transformation of the data onto a „flat" form usually proceeds in several stages: by successive „layers" of the communication system. All these activities are ensured by suitable protocols.

Example 3 In computer *E,* a string of alphanumeric characters, e.g. a text of a natural language, is stored in consecutive memory bytes and being encoded differently than in computer F (and, possibly, in a distinct order). Correct reception of this text sent from E to F requires transforming it onto a „jointly understood" intermediate form, determined by a suitable communication protocol between E and F.

Example 4 In a computer G, bits are represented by electric signals of completely different voltage than in a computer H (for instance in G, bit 0 by a voltage lower than bit 1, while in computer H—conversely). Moreover, the first bit transmitted by G is at the beginning of the message, whereas transmitted by H—at the end. Here, again a certain „jointly understandable" form is needed for the correct message reception.

Some analogies may turn out noticeable:

- The relationship between a protocol and communication in action, is similar to the relationship between an algorithm and computation: this is a finite description of activities;

- The intermediate form of messages defined by a communication protocol, plays a similar role like:
  - a standardized intermediate language common to various computers equipped with interpreters of this language. Different compilers of a higher level programming language, are transforming source programs onto the intermediate form, independent of the individual computer architecture (diverse processors, operating systems). This intermediate language is further interpreted by individual „virtual" machines. An example is the standardized intermediate language for Java, interpreted by Java's virtual machine, that translates each statement of the intermediate language onto the machine code of individual computers (equipped with the Java virtual machines), executes it, translates the next statement, executes it, etc.
  - written correspondence of Mr. Diego with Ms. Krystyna. Diego, the Spaniard, writes a letter to Krystyna in Spanish. However he has a friend Gulmaro, whose native language is Spanish, but with a good command of English. Krystyna, the Pole, writes to Diego in Polish, however she has a Polish friend Joanna with an excellent command of English.

A message transmission may be accomplished without transformation to an intermediate form, by means of protocols using by dispatch module. The message may be transmitted directly, as a sequence of characters written by the sender in a programming language. In this case, an exact information on the transmitted data structure must be attached. Such information—a sequence of control characters—are used by protocols of the receiver's module, for transforming the received sequence onto a sequence representing the data structure in the receiver's language. For instance, in Fig. 5.1, the intermediate English disappears if Joanna knows Spanish language, in which messages travel through the channel. The "marshalling" and „unmarshalling" is performed by one protocol only—Joanna. Such solution is an implementation detail.

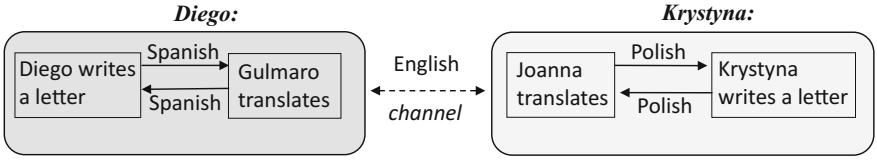


Fig. 5.1 Gulmaro and Joanna play a part of programs—communication protocols; a common intermediate language is English

5.3 Dispatch and Reception

There are various notational forms to initiate and carry out communication, in languages that offer mechanisms for concurrent and distributed programming. Their syntax and semantics concern not only message transmission itself, but a number of some accompanying properties. It conforms to general concept and structure of a language supporting a certain programming paradigm: imperative, object, functional, in logic, event-directed, etc. The detail presentation of such mechanisms exceeds the subject of this chapter and may be found in the respective writings as e.g. Hoare (1985), Milner (1980), Ben-Ari (1990), OCCAM (1984), Szałas and Warpechowska (1991), Brinch Hansen (1975), Arnold et al. (2005), Stevens (1997), [Tanenbaum and Wetherall 2011]. The main task of the mechanisms is to deliver a message to a certain site of reception. This takes place in effect of executing an operation, differently named in various programming languages and operation systems. Let us name them ***send*** and ***receive***. In distributed systems, such operations play a similar part like input/output operations in a stand-alone, autonomous computer with external devices, where suitable protocols are also applied and message delivery sites (ports, sockets) specified. Some low-level (machine) operations of communication: !, ?, S, R, were shown in Sects. 1.4 and 1.5 of Chap. 1. Their parameters were name of receiver's process in the sending process and name of sender's process in the receiving process, as well as respective addresses where the message is being sent from, and delivered to. These low-level operations represent maximally simplified synchronous and asynchronous mechanism, neglecting identifiers of ports, sockets, channels and neglecting reaction to exceptions like damage or loss of messages and delivery receipt, time-out, etc. Some programming and operating systems, offer means for specifying properties of message transmission or include means for such specification in send/receive operations as parameters. Here, we confine ourselves to parameters specifying messages, processes and threads, sites of dispatch and reception like ports or sockets and to media of transmission like channels.

Messages

For the user, data transmitted between computers or processes, are collected into structures being abstract concepts—models of real objects. For instance, a model of city transport is a graph, a model of population census is a list (a text file, a sequence of records), a model of family history on the male side is a tree, etc. The essence of a data structure is a way of grouping of single (elementary) data, i.e. rules determining their position in the structure and a way of accessing them. Whereas in the computer memory, the data structure is stored in sequentially addressed cells or bytes, that is—a linear object. There exist some standards of linear representation of data structures, not being dealt with here. To a data structure, linearly stored, a name and type is assigned (see the frame below). This is done in programs of communication participants, by means of declaration of a variable used as the actual parameter of the send/receive operations. This parameter

represents a message. But allocation of named objects, i.e. assigning addresses to them, is managed by suitable modules of distributed operating system. For the user, such proceeding is similar to commonly known in not distributed programming, where it is hidden (transparent) and accomplished by a compiler, interpreter or runtime system. Addressing of a data structure in distributed systems is, however, much more complicated. This is due to its possible storage in memory of many different computers, as well as its possible relocation and problems of memory consistency when using its multiple copies. Such problems are left to Chap. 8.

Every data structure has a certain type. This is a feature determining a way of grouping of single components and informing about permissible actions on them. For instance, a matrix of numbers has a type of two-dimensional array with equal length of rows, equal lengths of columns and direct access to its entries, and permission of arithmetic operations on them, but not e.g. logical. Therefore, the type determines a schema of placement of single data in the compound structure, rules of access and informs of permissible operations on them. In terms of the set theory, the type is a set of all data structures, of identical manner of grouping and handling their components.

A basic method of message transmission in network is *packet switching*, in contrast to *circuit switching*, applied in the past telephonic systems. A message, before dispatch, is partitioned into parts called packets, along with information how to reach its destination and is being sent serially through one route, or concurrently —via various routes (Sect. 5.4 where connection-oriented and connectionless transmission is presented). A transmission may proceed incorrectly or not to act at all. For instance, the receiver may get a corrupted message, may not get it at all or get it many times, the sender may not get acknowledgment required by the protocol, etc. Notice that the receiver becomes the sender when send the acknowledgment or provides a service requested by the sender. Prevention of message corruption may be partly obtained by usage of the check sum, a technique well known in mono-computer systems.

Processes and threads

In the parallel and distributed programming, processes and threads (the "light processes", with joint address space) are declared, as well as created dynamically—during the program run. They have names assigned and perhaps attributes, i.e. types. Remember that in a simplified version of a model-language CSP (Chap. 1), as parameters of operation "!" (send) and "?" (receive) were names of processes and of location of messages and the communication was synchronous. Similarly, in some high level programming languages (like ADA, Linda, Concurrent Pascal, OCCAM, Modula-2, Loglan, Java, Concurrent Prolog) and operating systems (like UNIX, Amoeba, Mach, Chorus), names of processes (in the ADA, instead of the term "process" the "task" is used) or threads, may be transmitted in messages, thus

may appear among parameters of send/receive operations. In particular, as a parameter may appear an identifier of a remote procedure or method—see Chap. 6. The synchrony or asynchrony of communication may be a specific feature of a programming language (e.g. in ADA and OCCAM—synchronous communication, whereas in Linda—asynchronous), or may be a parameter of send/receive operation. In the programming languages with rich mechanisms of interprocess communication, a class declaration may contain names of processes or threads as formal parameters. The activity is dynamic: processes and messages may be created and may disappear in the course of distributed program run. This is accomplished by services from execution system of a programming language and distributed operating system. Some programming and operating systems admit multicast communication: a parameter of the send operation is then a name of a group of processes.

Ports

A process may have many dispatch/reception points for message passing between computers and various devices. Such points or sites, are referred to as „ports", by analogy to junction points (interfaces) in computers, for connecting external devices, like mouse, camera, keyboard, etc. Throughout a port ("entryway"), messages are collected in a "mailbox", i.e. a buffer. Likewise as processes, destination ports may be named and declared or created dynamically during activity of distributed programs and transmitted in messages directed to these ports. So, in the ***send*** operation, among parameters representing a message and receiving process, identifiers of destination ports may appear. Such mechanism, for example, is offered by the ADA programming language, where the sending process encounters ("shakes hand"—the synchronous communication) with the receiving process in a port determined by the programmer. In the theoretical, model-language CCS (Milner 1980), the port is the primary (not defined) notion, understood as a junction point of communicating processes.

Sockets

This concept originates from extended versions of operating system UNIX, called UNIX 4.x BSD. Transfer of a message takes place from the output socket to input socket. The sockets are created by a process, for communication with another process, and are defining points in this process, whence messages are to be dispatched and the points in other processes where messages are to be received. Creation of a socket by a process (by calling procedure *socket*), assigns a name and properties to this socket. The properties are encoded within parameters of the *socket* procedure or in remotely called procedures. The socket mechanism is described in the extensive book (Stevens 1997).

Channels

An example of distributed programming language where channels for message transmission are explicitly defined, is OCCAM devised for integrated circuits, the so-called transputers (Stakem 2011). This language is a concrete realization,

somewhat modified, of the abstract model-language CSP. The names of processes (in CSP) have been replaced with names of channels declared in a program, which is a parent (in OCCAM) for programs describing processes. So, the process which sends message, delivers it (e.g. arithmetic expression) into a channel, whence it is taken by the receiving process. The sender and receiver are communicating by means of the same channel. The communication is synchronous: the sender waits for readiness of the receiver to take the message, and the receiver waits for readiness of the sender to put the message in the channel. Some distributed programming and operating systems, apart from the ***send*** and ***receive*** operations, contain more structured mechanisms of interprocess communication, like RPC and RMI—Chap. 6.

Message flow from the sending unit to receiving unit (basic for some programming languages) may be presented as follows:

- *process* → *process* (CSP)
- *process* → *port* → *process* (ADA, Modula-2)
- *process* → *channel* → *process* (OCCAM)
- *socket* → *socket* (UNIX 4.x BSD)

5.4 Modes of Communication: Synchronous and Asynchronous, Connection-Oriented and Connectionless, Multicast and Broadcast, Group Communication

Synchronous communication

A model-example where synchronous transmission is the basic communication mode is CSP (Hoare 1978, 1985) and its realization (slightly modified by introducing the channels) as a programming language OCCAM (OCCAM 1984). An example of extensive high-level programming language with explicit commands for such mode of communication is ADA (Barnes 2005). In the synchronous communication, the sending process, starting a ***send*** operation is suspended until the message has been received by a receiver and the acknowledgment of delivery arrives. The process resumes the main activity, when the dispatch and reception of data (in general—communication session) is fully completed. This is called a *synchronous* (or *blocking*) *send* operation. The receiving process, starting a ***receive*** operation, is suspended until an expected message arrives and the acknowledgment of delivery is dispatched to the sender. Then the process resumes activity. This is called a *synchronous* (*or blocking*) *receive* operation.

Metaphorically one may say that a meeting ("randez vous") of the sender with receiver takes place in the period of communication session. A created transmission channel contains, at that time, one message at most. The channel disappears on

completion of the session. Exceeding of predetermined time of waiting for the interlocutor (the timeout) may interrupt the session and, possibly, retransmission of the message by the sender. The synchronous transmission between processes $p1$ and $p2$ is depicted in Fig. 5.2a, b, where events and actions marked black represent sending, marked grey represent reception and white are other events. The wavy arrows denote message flow between processes. Their slant illustrates elapsed time from dispatch to reaching destination by elementary signal, e.g. a single bit. An implementation detail is, which components of the system realize the meeting: the process directly, as in Fig. 5.2a, or the operating system on request from the process, as in Fig. 5.2b. Here, process $p2$ waits for arrival of the ***send*** operation in process $p1$ and during the next meeting, *p1* waits for arrival of the ***receive*** operation in $p2$.

Asynchronous communication

In this mode of transmission (called also *non-blocking*), the sending process puts a message in its output-mailbox and during transmission of a message, is not suspended, but resumes the ordinary activity without waiting for acknowledgment of delivery. Thus, the transmission may proceed in parallel with successive process activity. The receiving process, may either wait for an expected message or resume

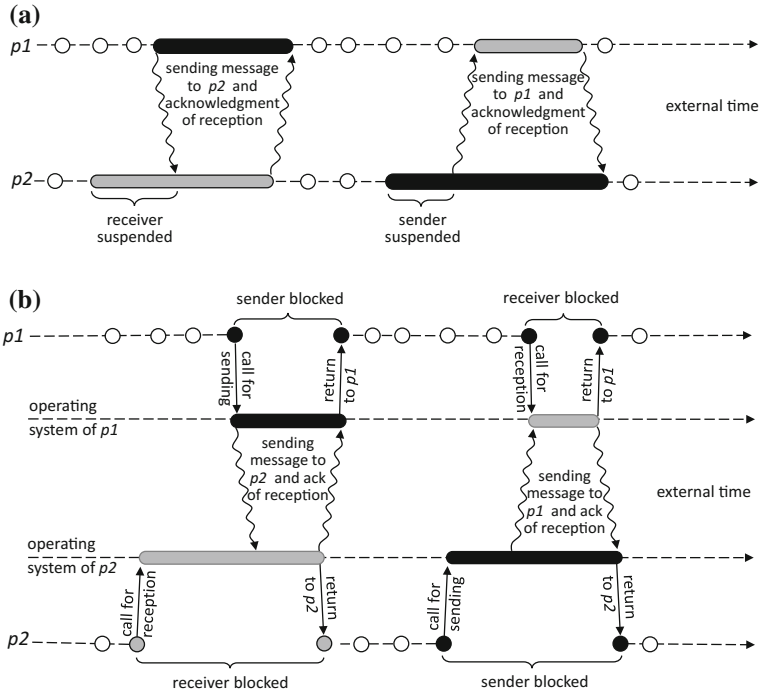


Fig. 5.2 **a** Direct synchronous transmission **b** Synchronous transmission performed by call to operating system

the ordinary activity and fetch the message from its input-mailbox later. Thus, there are possible two ways of asynchronous message reception: with *blocking* or *non-blocking* of the receiver. In the blocking way, the receiver is suspended until it gets complete message directed to it. In the non-blocking way, if the receiver has not ordered (by operating system) a message reception, then continues the ordinary activity, even if another process is sending a message to it. If it has ordered reception of a message, it interrupts its ordinary activity only when fetching the message from the input-mailbox—if not empty. Otherwise, waits for appearance of the complete message. An example of the high-level programming language with explicit commands for asynchronous mode of communication is LINDA (Carriero and Gelernter 1989). A schematic asynchronous transmission with non-blocking of sender and receiver is depicted in Fig. 5.3. The $p2$ process has ordered reception of a message, its input-mailbox is empty, so $p2$ waits for delivery, then fetches the message from the input-mailbox and resumes the ordinary activity. The $p1$ process has ordered reception of a message, its input mailbox is not empty, so it fetches the message and resumes the ordinary activity. A modification of this schema to transmission with blocking of the receiver, is straightforward.

Connection-oriented communication

It takes place when the sender and receiver established a connection channel (logical or physical), called sometimes a "tunnel" and have made arrangement on some details of communication prior to the communication itself, i.e. data transfer. The channel may be used by these interlocutors only, that have established it. Like in the telephone communication, the channel may be established for permanent usage (as e.g. "hot line" between countries) or being created dynamically (as. e.g. switching in telephone exchange)—then it exists only during transmission. The logical ("virtual") channel is created by recording it in memory of each device (computer, router, switch) along the channel's route, the address of the successive

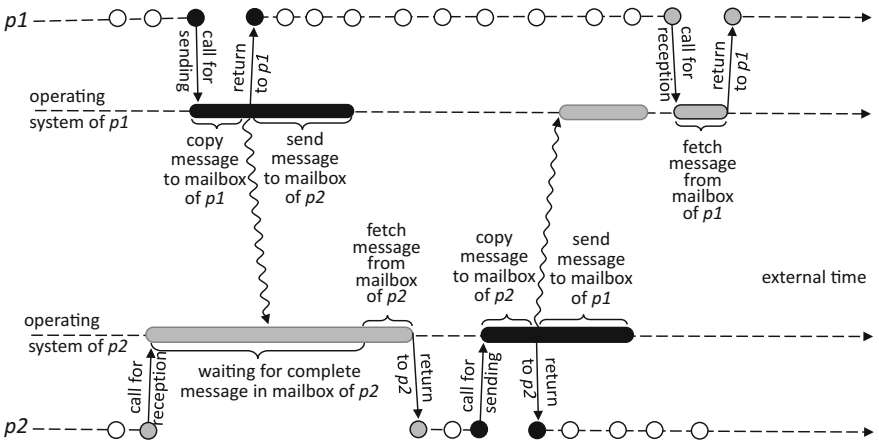


Fig. 5.3 Asynchronous transmission with non-blocking of sender and receiver

device—from the sender to receiver. The data in a message are delivered in the same order as have been sent as a sequence ("stream") of units, like bits or bytes. The receiver sends acknowledgment of reception. The message may be of arbitrary length. The error correction takes place during transmission, thus the connection-oriented is more reliable than connectionless communication. It is realized by various protocols, such as belonging to the popular TCP (Transport Control Protocols) family or ATM (Asynchronous Transfer Mode) family—see Sect. 5.5. An example of connection-oriented asynchronous transmission is depicted in Fig. 5.4. It is, in fact, the same as in Fig. 5.3, but the communication tunnels (established by suitable protocols located in operating systems of computers performing processes $p1$ and $p2$) are explicitly exhibited.

Connectionless communication

It takes place when the sender and receiver do not establish any connection, but a message along with receiver's address is passed to the communication service for dispatch (like in a postal service—the sender puts a letter into a mailbox). On the sender's side, the message is being partitioned into units called *datagrams*, each equipped—independently of other datagrams—with information how to reach destination. They may be transmitted via different routes concurrently and are being assembled in correct order into a complete message on the receiver's side. The order of their delivery may differ from the order of dispatch. Neither reception acknowledgment nor error check, as well as repeated delivery of the same datagram takes place. So, this is not such reliable transmission like the connection-oriented—for a price of higher efficiency (speed). Examples are multimedia broadcast of picture, sound, animated scene, video conference, e-learning, films, etc. where disappearance or corruption of a few datagrams is imperceptible for the receiver.

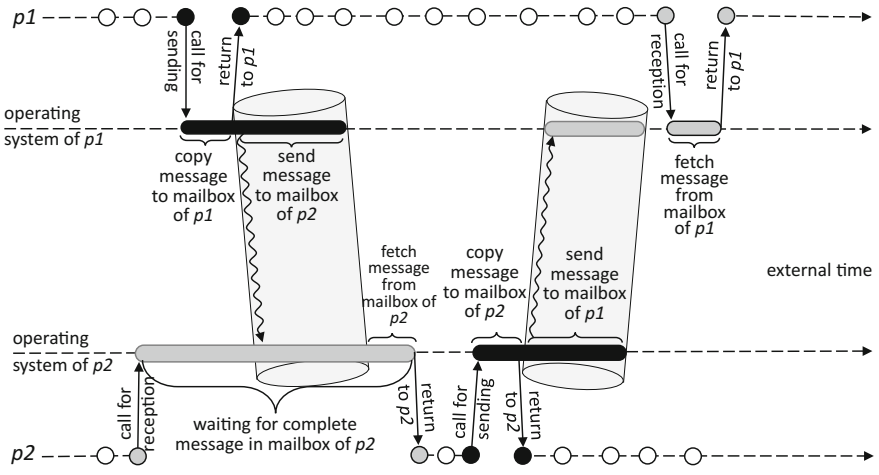


Fig. 5.4 Connection-oriented asynchronous communication; the tunnels established prior to message dispatch

However, for applications where the high speed transmission and reliability is indispensable, it may comply such requirements. Datagrams obtain then the unique identifications, e.g. numbers, by which the receiver assembles the complete message and sends acknowledgment to the sender. In case of a lost datagram (timeout occurred!), its copy is being retransmitted, and if repeated—the duplicate removed. The typical protocols for connectionless communication belong to the UDP (User Datagram Protocol) and IP (Internet Protocol) family. In Sect. 5.5, functions of various protocols making a layer structure of their set is described "in action" of message transmission. The choice of protocols in concrete implementation depends on objectives of a particular distributed system, that is, on its applications.

Remark Notice that the connection-oriented/connectionless and synchronous/asynchronous modes of communication are mutually independent.

Group communication

So far in this chapter was assumed that one sender dispatches a message to one receiver, performing the so-called *point-to-point* communication. Its generalization is when one sender may dispatch a message to many receivers simultaneously. If their set is fixed, the transmission is called a *multicast*, or *point-to points*. If their set contains all the computers in the system, the transmission is called a *broadcast*. The group communication has been used in Sects. 4.1.2 and 4.3 of Chap. 4 and will be used in Chaps. 7 and 8. The representation of a group of communication participants as a graph, where every participant may send messages to everyone, is called a *clique*. A group may evolve during system activity: some participants are leaving the group or entering it, without affecting activity of remaining computers. This feature is the scalability of the group—a general property of distributed systems (Sect. 2.3.4, Chap. 2). Other two properties of the group communications are: *atomicity*—each message must arrive in all members of the group, or into no one of them (it is inadmissible that some processes receive the message and some do not) and a fixed order of message reception (for instance FIFO, called a global FIFO). Figures 5.5 and 5.6 depict communication of processes $p1$, $p2$, $p3$, $p4$ in two groups: $\{p1, p2, p3\}$ and $\{p1, p3, p4\}$. At 18:00, the process $p4$ is sending message A which reaches process $p3$ at 19:00 and $p1$ at 19:30. At 18:30, the process $p2$ is sending message B which reaches process $p3$ at 20:00 and $p1$ at 20:30. The FIFO order is, thus, preserved.

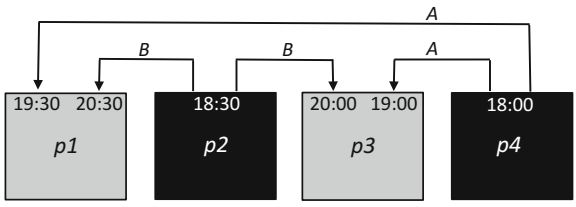


Fig. 5.5 Group transmission: $p2 \rightarrow \{p1, p3\}$ and $p4 \rightarrow \{p1, p3\}$

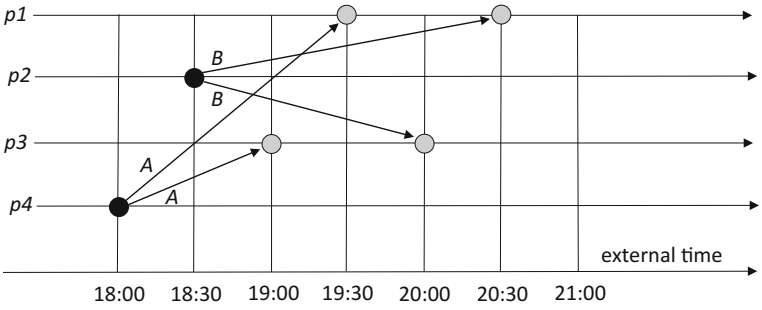


Fig. 5.6 Diagram of the global FIFO order transmission depicted in Fig. 5.5

The same transmission represented on a space-time diagram is depicted in Fig. 5.6. The global FIFO reception order is in this example evidently described by the formula:

$$x \overset{message}{\longrightarrow} y \wedge u \overset{message}{\longrightarrow} v \wedge C(x) < C(u) \Rightarrow C(y) < C(v)$$

for all sending events x, u (black) and reception y, v (grey). $C(x)$, $C(y)$, $C(u)$, $C(v)$ are timestamps of respective events (Sect. 4.2, Chap. 4). Here, $C(x)$ = 18:00, $C(u)$ = 18:30,

$C(y)$ = 19:00—if y is the event in $p3$ and $C(y)$ = 19:30—if y is the event in $p1$;

$C(v)$ = 20:00—if v is the event in $p3$ and $C(v)$ = 20:30—if v is the event in $p1$,

Using notation introduced in Sect. 4.2, the general formula characterising necessary and sufficient condition for the global FIFO order of group communication is expressed as:

$$x \overset{message}{\longrightarrow} y \wedge u \overset{message}{\longrightarrow} v \wedge x \sqsubseteq u \Rightarrow y \sqsubseteq v \tag{5.1}$$

for each pair $\langle x, u \rangle$ of dispatch events and each pair $\langle y, v \rangle$ of reception events (remember: $x \sqsubseteq u$ is equivalent, by definition, to inequality $\langle \mathrm{C}(x), \#(p_x) \rangle \preccurlyeq \langle \mathrm{C}(u), \#(p_u) \rangle$ of global timestamps of events x and u occurring in processes p_x and p_u respectively).

The space-time diagram in Fig. 5.7 depicts transmission in the same groups as in Fig. 5.6, but with reception of messages not in the global FIFO order: message B sent later than A, reaches both receivers sooner than message A reaches its receiver $p3$.

Here, the conjunction $x \overset{message}{\longrightarrow} y \wedge u \overset{message}{\longrightarrow} v \wedge x \sqsubseteq u$ is true and $y \sqsubseteq v$ is false, for dispatch x in $p4$ with its reception y in $p3$ and for dispatch u in $p2$ with each of its receptions v in $p3$ and in $p1$.

Notice that the formula (5.1) is fulfilled also if x and u are in the same process, as well if $x = u$—consider Fig. 5.8.

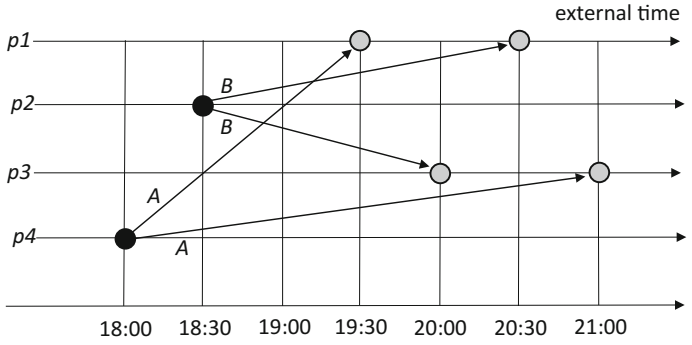


Fig. 5.7 Diagram showing violation of global FIFO order of transmissions

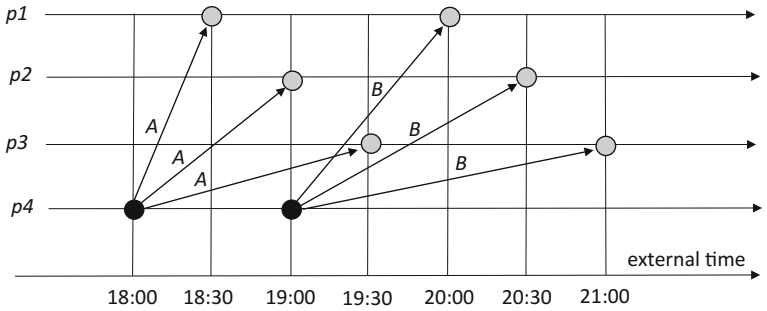


Fig. 5.8 Notice that if $x = u$ then all receptions of u coincide with receptions of x

5.5 Layered Structure of the Set of Communication Protocols: OSI/RM and ATM

Messages travelling in the network are sequences of bits partitioned into fragments called packets, datagrams, cells, etc.—depending on a communication model and mode of transmission. Transforming a message from its syntax in a programming language (user's external form) onto a sequence of bits moving in the network, proceeds usually stage by stage: through successive modules called layers, containing protocols performing certain functions. Protocols from a layer convert the message delivered by the previous layer onto a form for the next layer. Passage from a layer to the next one proceeds via interface using an auxiliary protocol, called sometimes „generic". Such conversion, on the sender's side, is called a marshalling and on the receiver's side—unmarshalling (Sect. 5.1). Each layer deals with a specific aspect of communication, for instance, in the application layer, identification of transmission participants and determination of a mode of transmission takes place. Thus it plays a role similar to compiler's pass (like lexical analysis, syntax analysis, etc.)—see the frame below. The layers are numbered:

0, 1, 2, …, n (in Fig. 5.9 $n = 8$) with user computers in layer 0 and the network in layer n. On the sender's side, the layer $j > 0$ takes from the layer $j - 1$ the user message (a "payload" suitably prepared by layer $j - 1$) equipped with a header containing control information necessary for the layer j (and the receiver) for further processing of the message. The layer j converts ("marshals") the message further, appends a header to the one received from layer $j - 1$ and necessary for layer $j + 1$ ($j < n$). On the receiver's side, the layer $j > 0$ takes from the layer $j + 1$ the message (with the headers) prepared by the sender in the layer j. Then, converts ("unmarshals") it, using information from the header appended (by the sender) in the layer j, deletes this header and passes the converted message with remaining headers to the layer $j - 1$. In Fig. 5.9 the layer structure of protocols is depicted, arranged in accordance to the standard elaborated by *ISO* (***I**nternational **S**tandard **O**rganization)*, referred to as the **OSI/RM** (***O**pen **S**ystems **I**nterconnection* **R**eference **M**odel 1983). Table 5.1 shows this structure in action, where computer 1 is sending a message to computer 2. Various functions of the protocols in the respective layers are given.

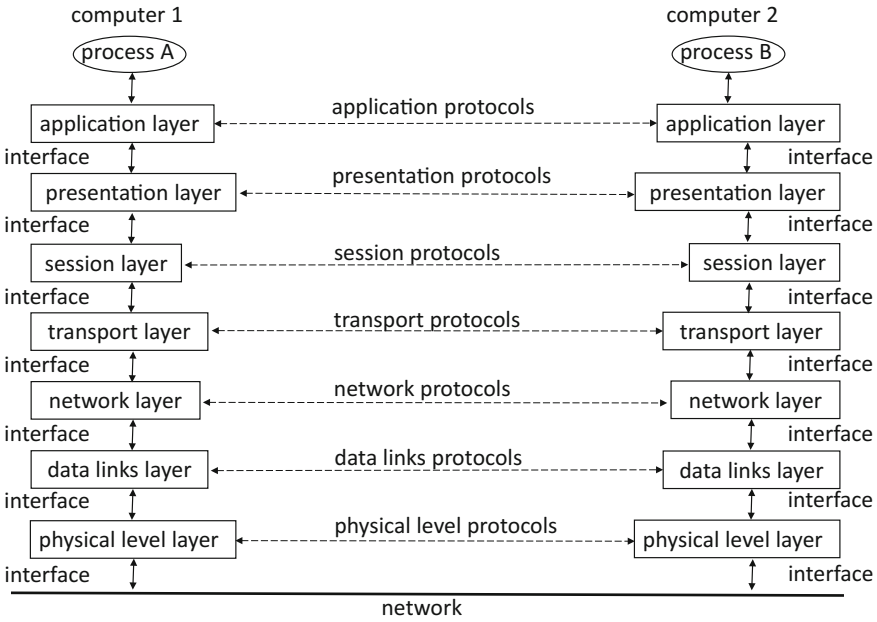


Fig. 5.9 OSI/RM layer structure of protocols; each computer may be sender or receiver

Table 5.1 Transmission of a message from computer 1 to computer 2 through successive layers of protocols; obviously, transmission from computer 2 to computer 1 looks the same

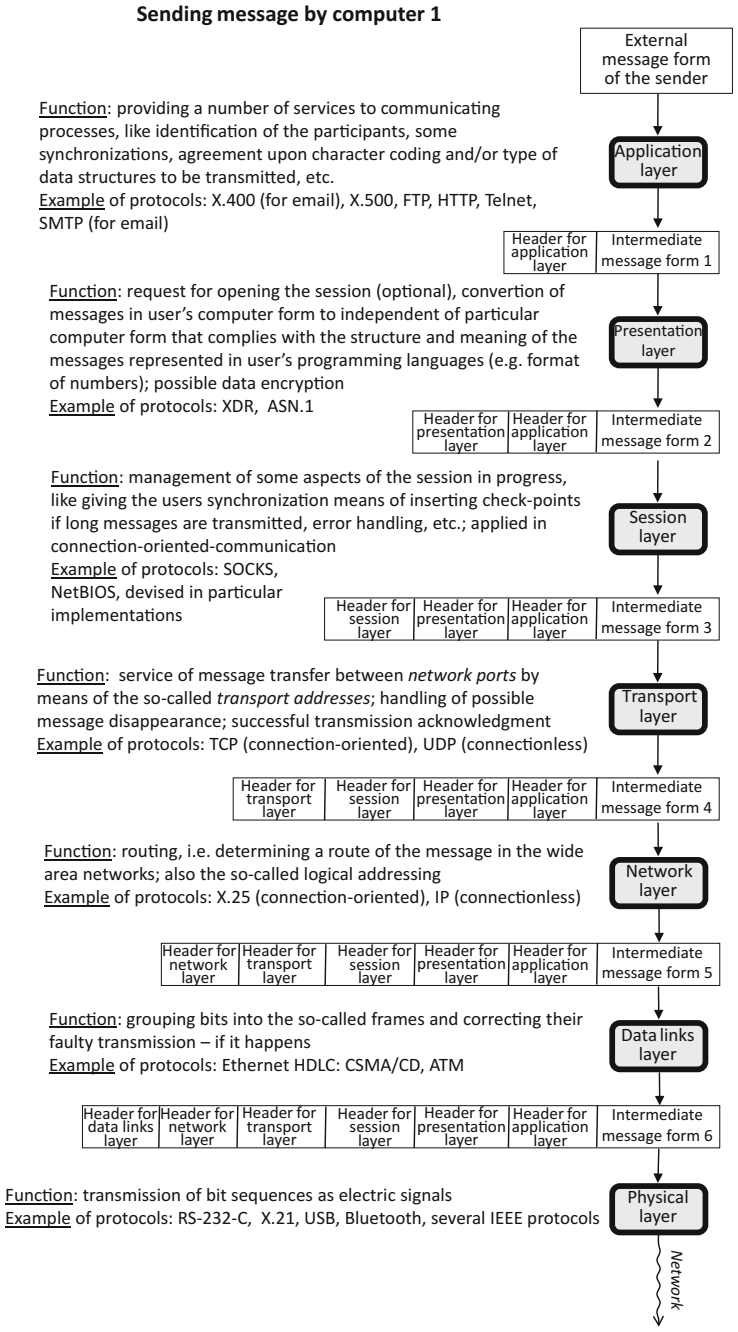
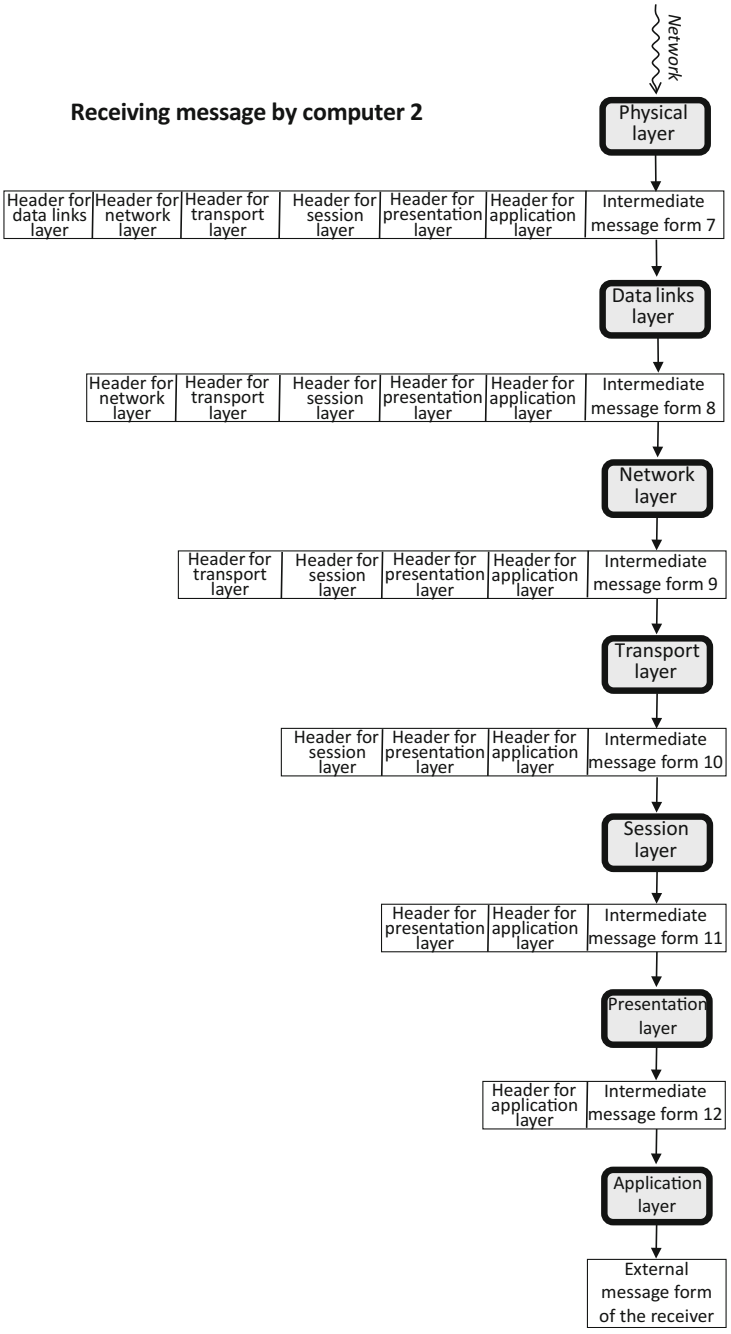


Table 5.1 (continued)



An analogous arrangement of some compilers is partition of the whole compiler into successive passes. They convert—stage-by-stage—the program during compilation, from its external (user's) form onto the form interpreted by runtime system (e.g. machine instructions or an interpreter, like a „virtual machine"). Another example is a technique called "bootstrapping" in translating of one programming language to another. In mathematics, a layer structure represents a composition of functions.

Apart from the OSI/RM model, examples of layered structure of protocols models are:

- TCP/IP early prototypes (1975–early 1980) developed and used by Stanford University (USA), University College (London, UK) and some other institutions, then in 1982 adopted by US Dept. of Defense. In 1989 adopted for UNIX; 4 layers.
- **ATM** (**A**synchronous **T**ransfer **M**ode (1988); 5 layers—see below.

The communication OSI/RM mechanism is a conceptual schema, having various implementation in real systems. It is a proposal of structural arrangement of protocols. It enjoys important property: the layers are mutually independent, which allows for replacement or modification of protocols in a layer, with no affect for other layers. This is because any layer determines only activities that are to be performed of a given stage of transmission, but not which specific protocols are to be used for these activities. That is why in real implementations, various constructs and names of protocols are used for the same function. In the early versions, the OSI/RM model assumed synchronous and connection-oriented mode of communication. In its later versions, this feature has been treated more liberally. For instance, although the original model assumed connection-oriented communication, some protocols at its layers assume connectionless communication (an example is TCP/IP—a combination of connection-oriented with connectionless—and UDP protocols at transport and network layers); this is an extension of the RM/OSI early versions to provide also connectionless service.

In the Table 5.1, activity of successive layers and examples of protocols for this purpose is shown.

Another standard of layered structures of the protocol suit is ATM (Asynchronous Transfer Mode) elaborated in 1988 by ANSI (American National Standards Institute) in cooperation with ITU (International Telecommunication Union). Since 1991, this standard in being promoted by the international industry consortium ATM Forum. The ATM model has been devised to transmit not only texts, but also multimedia data like voice (speech), music, pictures, paintings, films, animated www pages, videoconferences, etc. To this end, indispensable was increased speed of transmission, possible at that time due to development of network techniques. The increased speed was also an effect of applying protocols not requiring acknowledgment of message delivery, as well as partition of the message

onto small units (48 bytes for a "cell" with a piece of message and 5 bytes for a header) transmitted concurrently through different routes. A version of this model, due to high speed of transmission, is also applied for mobile devices. The transmission is mainly connection-oriented, but admits connectionless mode too – when synchronization between sender and receives is not required. The general principle of the ATM technique follows the OSI/RM model: the layered structure of protocol suite and appending headers (on the sender's side) and removing them (on the receiver's side). This layered structure of the ATM model is shown in Fig. 5.10.

The application layer plays a similar part as its counterpart in the OSI/RM model. Functions of the „planes" layer provide various services for transmission in networks applying the ATM technique and differ in its particular implementations. The typical are: check-up (e.g. whether all cells reached destination) and removal of connections, management of resources needed to perform connection, refusal of connection because of absence of resources necessary for setting up this connection (e.g. all the routs are occupied), signalling of timeout of connections, data recovery from before occurrence of transmission error—these are examples of services provided by the „planes". Functions of the ATM adaptation layer are devised for adapting the form of messages coming from previous layers to the form fixed in the ATM technique—as a sequence of (48 + 5)-byte cells. On the sender's side, the protocols of this layer accomplish partition of input packets onto these cells, while on the receiver's side—their assembling into suitable packets. In the ATM layer activities allow to hide some details of transmission, such as creation of headers of packets on the sender's side and their usage on the receiver side, routing of packets, multiplicity of identical packets in switches, etc.

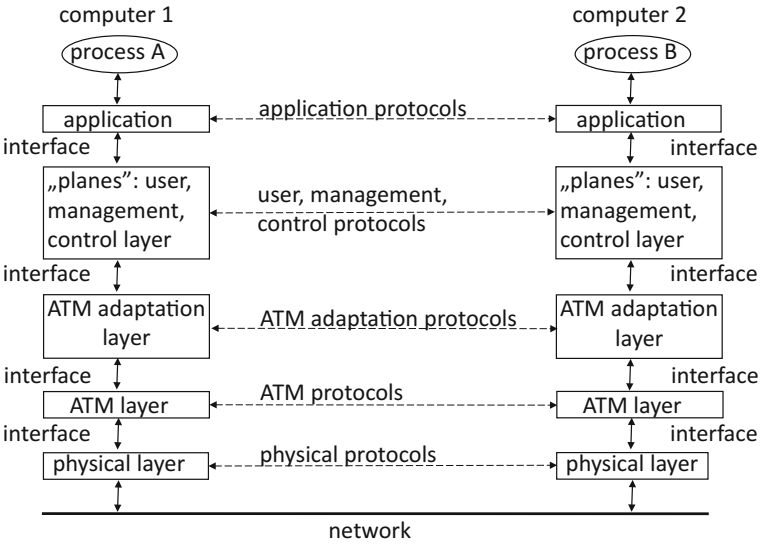


Fig. 5.10 ATM layered model

References

Arnold, K., Gosling, J., & Holmes, D. (2005). *The java programming language* (4th edn.). USA: Addison-Wesley Professional.

Barnes, J. (2005). *Programming in ada 2005*. USA: Addison-Wesley.

Ben-Ari, M. (1990). *Principles of concurrent and distributed programming*. USA: Prentice-Hall.

Brinch Hansen, P. (1975) The programming language concurrent pascal. *IEEE Transactions on Software Engineering, 1*(2), 199–207 (June).

Carriero, N., & Gelernter, D. (1989). Linda in context. *Communication of the ACM, 32*(4), 444–458.

Comer, D. E. (2015). *Computer networks and internets* (6th edn.), UK: Pearson Education Limited.

Czaja, L. (1971). *GIER ALGOL 4*, (in Polish) Wydawnictwa Uniwersytetu Warszawskiego.

Dolińska, I. (2005). *Sieci Komputerowe* (*Computer networks*) (in Polish) Wydawnictwo WSE-I.

Fiałkowski, K., & Swianiewicz, J. (1962). *Maszyna ZAM-2. Opis maszyny. Kompendium programowania w języku SAS*. Prace Zakładu Aparatów Matematycznych Polskiej Akademii Nauk (*ZAM-2 computer. Description of the machine. Manual of programming*) (in Polish).

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM, 21*(8), 666–677.

Hoare, C. A. R. (1985). *Communicating sequential processes*. London: Prentice-Hall International.

Milner, R. (1980). *A calculus of communication systems*, *Lecture Notes in Computer Science* (Vol. 92). Berlin: Springer.

OCCAM. (1984). *OCCAM programming manual*. USA: Prentice-Hall. (C.A.R. Hoare Series Editor).

Sportack, M. (2004). *Sieci komputerowe. Księga eksperta*, Wydawnictwo Helion, Gliwice (Polish translation of [1998]).

Stakem, P. H. (2011). *The hardware and software architecture of the transputer* (Kindle Edition).

Stevens, W. R. (1997). *UNIX network programming* (2nd ed., Vol. 1). USA: Prentice-Hall.

Szałas, A., & Warpechowska, J. (1991). *Loglan*. Warszawa: WNT (in Polish).

Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer networks*. USA: Prentice Hall.

Chapter 6
Remote Procedure Call

6.1 Motivations, Problems, Limitations

An outline of problems the interprocess communication in distributed systems creates, has been presented in Chap. 5. Dependently on services and means of expression provided by operating system and programming language, communication operations require preparing various parameters carrying information necessary for message transmission. These are, for instance: a mode of communication, identifiers of the sender and receiver, form and type of the message, time limitations, a way of reaction on errors and other exceptions, etc. It is, thus, necessary to perform many actions preceding mere dispatch and reception of the message. To this end, some functions realized by appropriate protocols for creating and removing connection, directing messages to specified sockets or ports in network services programming (Stevens 1997), marshalling and unmarshalling of transmitted data, etc. are used (Sects. 5.1 and 5.2 in Chap. 5). Alleviation of such excessive burden of programmers, on the one hand, and fulfilment of methodologic postulate to clearly structuring programs—on the other, was the leading motive to design a mechanism called **RPC** (**R**emote **P**rocedure **C**all) (Birell and Nelson 1984) and its later object version **RMI** (**R**emote **M**ethod **I**nvocation) (Arnold et al. 2005). This mechanism is to assure communication in the form of the ordinary procedure call (invocation). This takes place as in ordinary procedural programming language, where procedure call is a special case of communication, where the invocation statement and the procedure body are localized inside the same environment (the same computer). Because the aim of a distributed system is to provide impression of working on one computer, so, the user should use procedures whose bodies are localized in other computers in the system, in an ordinary way. Thus, the remote procedure call is of the form (syntax) similar to the ordinary one in conventional programming languages, but the user is unaware about location in the network, of the procedure declaration, i.e. the procedure body (transparency). In some systems, the programmer should precede the remote call by creation (e.g. by an application from the

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_6

library) an interface to procedures (or methods) offered by a server. Thence the programmer gets information about presence or absence of the required procedure on this server, a type of parameters, etc. However, facilitation of the user's work in the distributed system, brings about many problems to designers of RPC mechanism, the problems absent in centralized systems. Let us say that a computer or process, in which a procedure call takes place, is a client and the one where procedure body (a declaration) is located—a server, and let us outline main such problems. By the way notice that request for delivery of time in the Cristian's method of clock synchronization (Sect. 4.1.1 in Chap. 4) is a kind of remote call for this service.

6.1.1 *Different Environments of the Client and Server*

Architecture, address space, the client's and server's resources are usually not the same. Transmission of actual parameters from a procedure call to formal parameters in the procedure body requires using of communication mechanism. In conventional procedural languages, various modes of actual to formal parameters passing are encountered: by value, by result and value (ADA, (Barnes 2005; Ben-Ari 1990)), by name (the Algol family) or by reference (Pascal, PL/1, Mudula, Java, C, C++ and others). The methods elaborated for these languages allow for sufficient efficiency (time and space complexity) of parameter passing, in case of one centralized environment, i.e. without their transmission through the network. Passing parameters through the network from one environment to another, often of different data representation, makes the activities of runtime system (a kind of "virtual machine"), which realizes remote procedure call, fairly complicated. This substantially lowers its performance. That is why in its nowadays existing implementations, parameter passing is limited to „call by value" and sometimes to „call by reference". The simplest is call by value: the client sends the value of actual parameter to the server-receiver, into a site assigned to the formal parameter. This requires a number of actions performed by ***send*** and ***receive*** operations: transforming the value from external form onto a sequence of bits in the network - passage through layers of protocols, in particular data marshalling and unmarshalling. Replacement of formal parameters with actual, the execution of procedure body and return of results to the client is accomplished by the server. As in case of the ordinary procedure mechanism, the formal parameter in procedure body becomes its local variable, inaccessible in the client's process. Therefore, procedures in RPC mechanism do not yield the so-called „side effects", i.e. changing values of variables in the calling process by the called procedure (provided that these variables are not parameters for results returned by the procedure). This feature is postulated by the structured programming methodology. Call by reference consists in transmitting to the server, a pointer to a place in the client's program, where the variable is located, instead of its value. This allows for access from procedure body to resource indicated by the pointer. The procedure may, in particular, change the content of this resource. The resource may be a data structure,

e.g. array, tree, graph, etc., which does not have to be transmitted „on the whole", as in case of the call by value. This is a convenient mechanism for the programmer, but more complicated for its designers. It requires network transmissions between a client and server, which makes it of poorer efficiency in terms of time and memory space and more exposed to errors than an ordinary procedure call. It is implemented in compilers and interpreters of the C language family, where scalar types (numbers, characters, truth values) are passed by value, but arrays—by reference. Also in ADA (Barnes 2005; Ben-Ari 1990), where variables of the reference type, may be parameters of procedures and tasks (i.e. processes). Applications offered by library of Java, permit the remote procedure call ("Remote Method Invocation" in Java) by the so-called „object reference", global for the entire distributed system. Some implementations of programming languages make possible deciding about a mode of parameters transfer, by the programmer. Figure 6.1 illustrates activities performed from remote procedure call to return of result and Fig. 6.2—a structure of the RPC mechanism. In Table 6.1 this mechanism is demonstrated in action.

6.1.2 Conflicts When Using Shared Resources

Since a procedure may give access to a resource for a number of processes running simultaneously, the system or user must assure their mutual exclusion, where it is necessary (Sects. 3.2 and 4.3 in Chaps. 3 and 4). Some programming languages, like Concurrent Pascal (Hansen 1975), ADA (Arnold et al. 2005), provide possibility to decide which resources require protection. For this purpose, the so-called monitors were introduced. The monitors are language constructs in a form of a collection of procedures, each protecting a certain resource, therefore executed by at most one process at a time. Thus, they implement critical sections ("protective zones" as named before). Users of programming languages where the language constructs for defining critical sections have not been provided, have to apply appropriate library procedures—if available. Otherwise they must assure the mutual exclusion „manually" by using other language constructs.

If the server does not allow for simultaneous execution of procedures remotely called by different processes, it puts the service requests into a queue to take them one after one for execution—Fig. 6.2. Moreover, if the distributed system with the RPC mechanism uses only a single server working sequentially, then conflicts of access to shared protected resources do not occur. Without such limitations (unobserved in some implementations, e.g. for the transactions—see Sect. 3.1 in Chap. 3), the RPC mechanism must assure realization of critical section, where this is necessary (for instance in the ANSA (1989) system assuring synchronization of threads). Obviously, in case of the ordinary (not remote) procedures, the protection of resources belongs to the user, e.g. by means of monitors or "manually" programmed.

6.1.3 The Stub

As said above, the RPC mechanism allows to use procedures located in environments different than their invocations (calls) in the same way as if were located on the user's local computer. The notation (syntax) of the procedure call is diverse in particular programming languages, but as example notation, let us take the following one:

$$procedure\_identifier\ (actual\_parameters)$$

where the list of actual parameters delivers data for the procedure and, possibly, variables for results returned by the procedure. The ideal implementation hides a location of procedure body: the user is unaware whether the procedure is local or remote or taken from the library. In existing implementations, this property is partly accomplished: some actions preparing a remote procedure call are being performed by the user. The main tool to this end is the so-called ***stub***, a service—subprogram consisting in two modules: transmitting and receiving, both for the client and server. The stub is an interface between the client and server, that transforms the procedure call into the form required by a communication protocol for dispatch to the server, for reproducing it on the server's side and finally for reproducing results on the client's side. The stub may be created in the user's program comprising remote calls of procedures, prior to the mere program run (the so-called *preprocessing*), if there are language facilities for its defining, or may be generated in a special language for defining such interface (e.g. CORBA—see Sect. 2.3.1 in

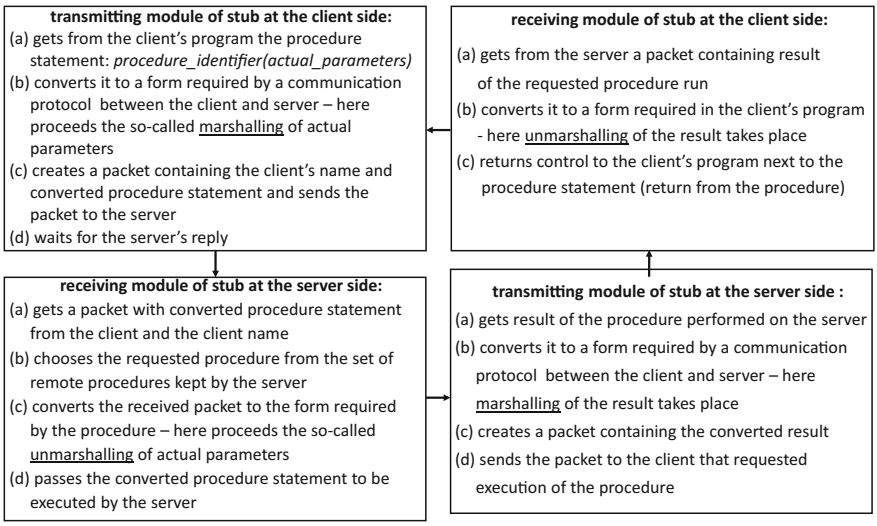


Fig. 6.1 Activity of the client's and server's stub in the cycle from invocation of remote procedure to return of results

Chap. 2) and if compiler of this language is associated with compiler of the user's programming language. After being created, the stub resides in an operating system, or a library, or in a runtime system of the user's program. Actions of the stub are shown in Fig. 6.1, and the "animated" example of three clients cooperating with one server—in Table 6.1.

Referring to the analogy of translation between Spanish and Polish language by assistance of two interpreters—Fig. 5.1 in Chap. 5—notice that the role of client and server is symmetric there. Diego and Krystyna play both roles, whereas Gulmaro and Joanna play a part of stubs.

An object-oriented version of the RPC mechanism is RMI (Remote Method Invocation), available in Java, sometimes referred to as a *distributed object application.* The client's and server's parts are called there a "stub" (in some publications just „interface") and "skeleton" respectively. Motivations and principles of RMI are the same as in case of RPC, while technical solutions and a way of usage differ in details. For a programmer, the main difference (apart from „object way of thinking") is existence of an address, available for all computers in the distributed system with RMI, referred to as an *object reference*. Such address may be passed as an actual parameter of a method call, as if the parameter passing took place in the same environment. By means of this parameter the dispatch of data, as well as reception, can be done.

6.1.4 The Binder—Finding a Server

Point (d) in Fig. 6.1 describing the client's stub, encompasses searching for a server where the remotely called procedure has been declared. This is a separate service of the RPC mechanism in concrete implementations, used by every stub. This service, referred to as *binding*, is accomplished by the so-called *binder*, and consists in making available a network address of a server where is the respective procedure, to the client. The binding may be either static, i.e. with keeping this network address during the entire period of the client's and server's presence in the system, or dynamic, when the client's stub requests the binder for a network address of a server (possibly, a point in the server's process, to which the stub should send the procedure call: a port or a socket). Depending on the search outcome, the service is performed or the client is informed about absence of needed server in the system. After having connected to the system and login, a server registers its presence in the binder along with data needed for the clients, then, after termination of its task—performs logoff. The dynamic binding enables greater degree of transparency (Sect. 2.2.3, Chap. 2) than static. In case of the static binding, replacement of a server, change of its localization, etc., requires new compilation of some programs,

while in case of the dynamic binding—only modification of tables comprising mapping of names into network addresses. In the above description of the client's transmitting module, finding a server and sending a packet with the procedure call, has been included into the tasks of stub, to avoid implementation details.

6.1.5 Exceptions

During the call of remote procedure, some events that prevent delivery of results to the client may occur. A server failure (Chap. 7) or its overload with services, failure of communication infrastructure, absence of required procedure declaration, time-out of processing the procedure call, missing packet in the network, not matching up parameters actual and formal (types, size of resources, exceeded range of array indices, etc.), some errors in procedure body—these are examples of events and situations preventing correct completion of the procedure call. They are referred to as *exceptions* and require notifying the client about their cause and a suitable reaction of the RPC mechanism or of the user (programmer). The user of programming languages capable of exception handling (like ADA (Barnes 2005), Modula-2, (Wirth 1987), Loglan (Szałas and Warpechowska 1991; Bartol et al. 1984; Kreczmar et al. 1990), Java (Arnold et al. 2005), C++), reacts on them in a way personally programmed. Some implementations of RPC mechanism, ensure exception handling automatically, implemented by system designers. The programmer, using the RPC mechanism with the automatic handling of exceptions (like Concurrent Pascal (Hansen 1975)), receives from the system appropriate messages about exceptions and, possibly, advices on actions that should be taken.

6.1.6 Lost and Repeated Messages

In some implementations of RPC mechanism, various strategies of exception handling are applied, in particular, strategies concerning disappearance and repetition of messages. Appropriate protocol keeps a message dispatched, until acknowledgment of reception in a defined time (before the time-out) arrives. Otherwise, the protocol repeats dispatch of the same message. The same concerns the acknowledgment. In concrete implementations of the RPC mechanism, various strategies of exception handling are applied, in particular, regarding disappearance and repetition of message. Their review may be found e.g. in Coulouris et al. (1994), Tanenbaum (1995).

6.2 Example of RPC Mechanism Activity

Figure 6.2 illustrates organization of a system of computers-clients p1, p2, p3, which use procedures located on a server. The clients are serviced in the order of procedure invocations and are passing through three states shown in Fig. 6.2 (a state of an exception occurrence has been omitted). It is assumed that after initiation of remote procedure call, the client waits for results, thus suspends its activity until the service is accomplished, whereas the server executes one procedure body during this time. Such mechanism is referred to as a synchronous RPC. Obviously more efficient solution is if the server runs many threads concurrently, but the principle of the RPC mechanism remains the same.

In some distributed systems [e.g. Mercury (Liskov and Shrira 1988), (Davidson et al. 1992)], an asynchronous RPC mechanism has been applied. That means that, after sending to the server request for procedure execution, the client does not wait for results, but continues activity and collects results later, until the server completes execution of the procedure and returns them.

The Table 6.1 on the following pages, demonstrates an example of system depicted in Fig. 6.2 in action is demonstrated.

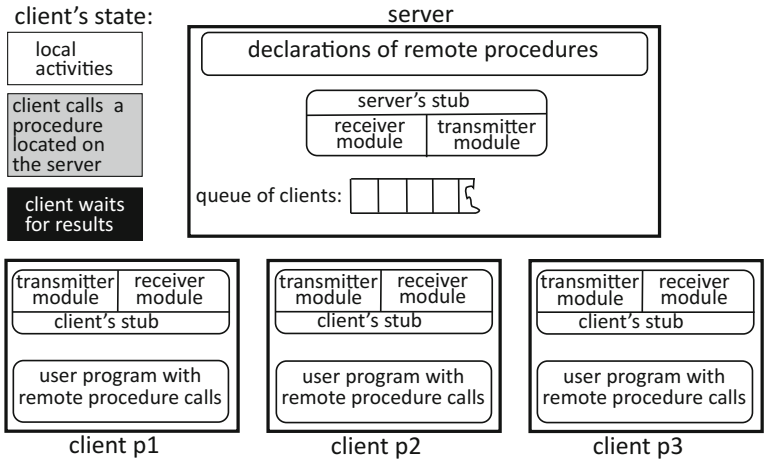
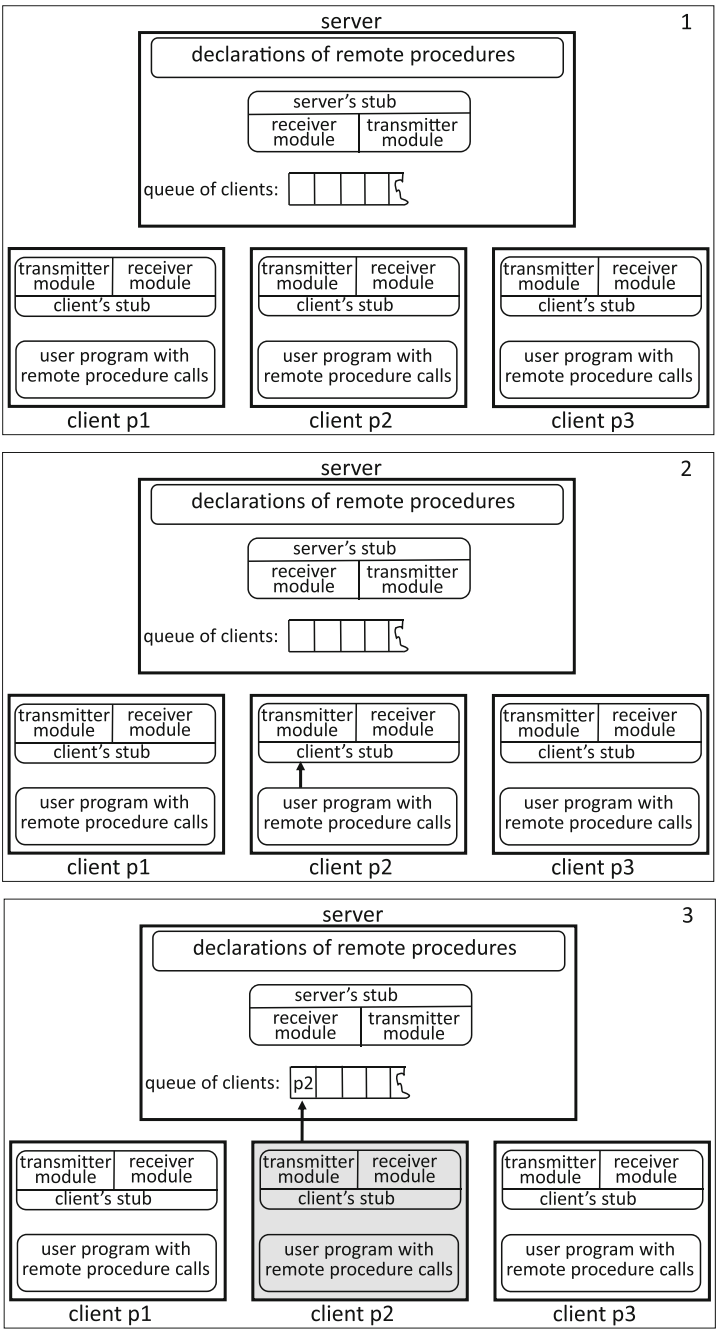


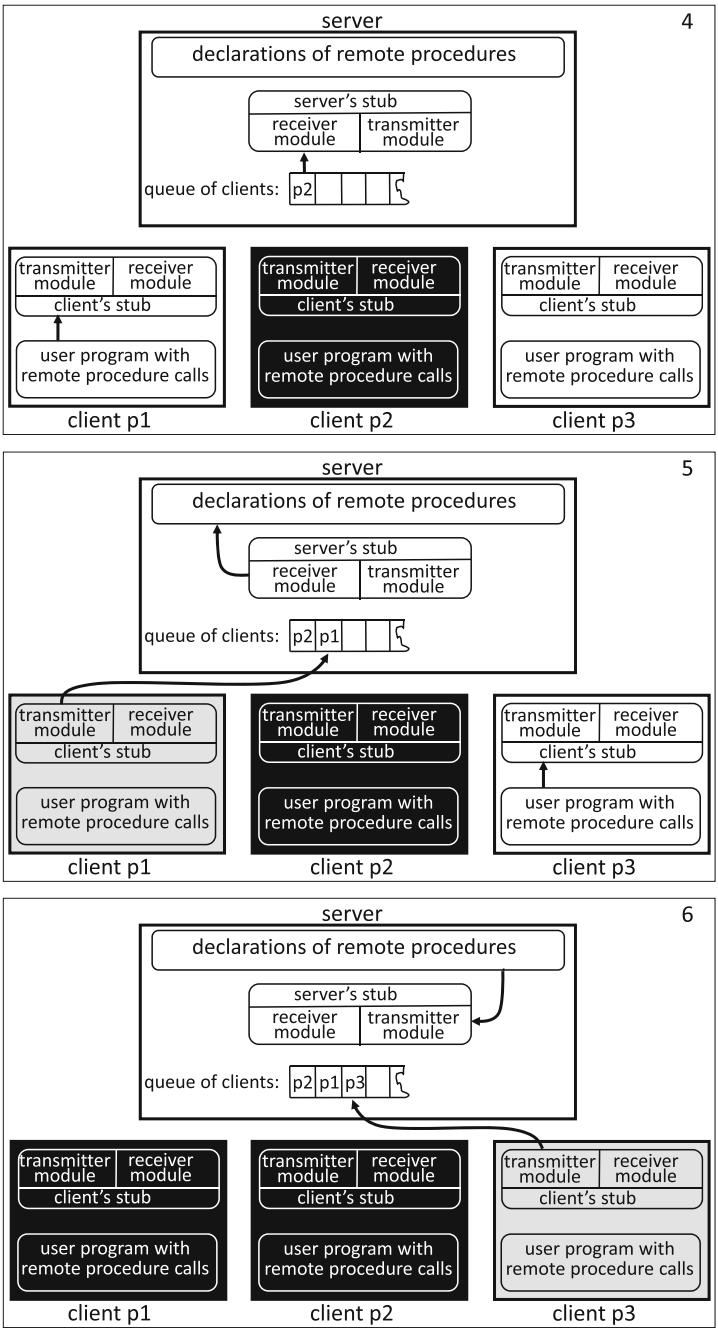
Fig. 6.2 Exemplary structure of the RPC mechanism and states, which the clients pass through from invocation to reception of results, when calling a remote procedure

Table 6.1 RPC in action



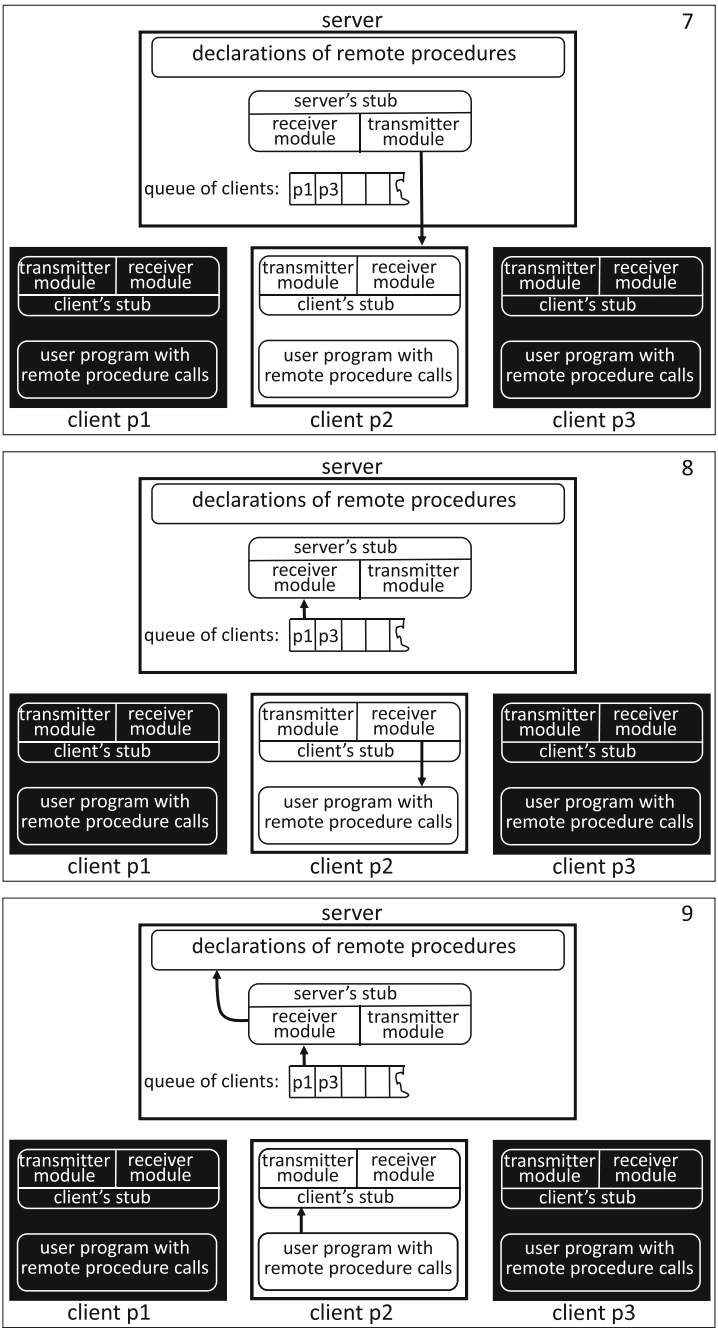
(continued)

Table 6.1 (continued)



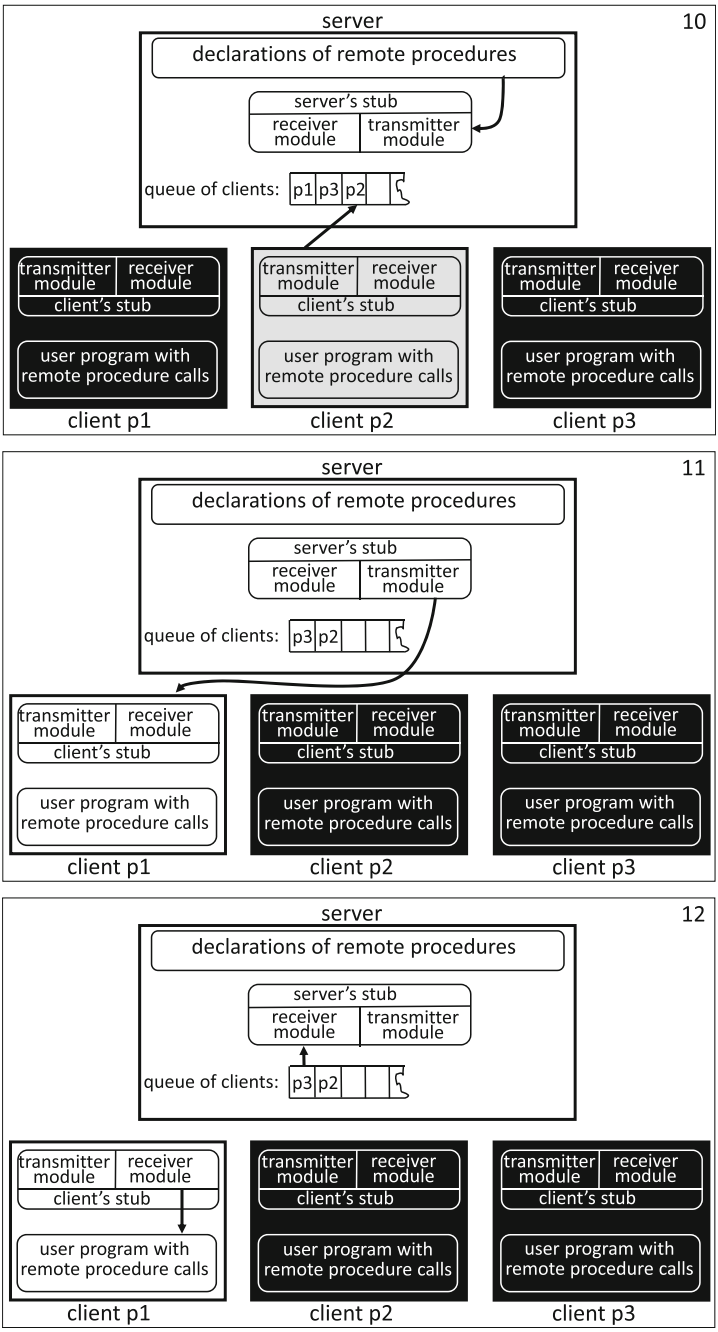
(continued)

Table 6.1 (continued)



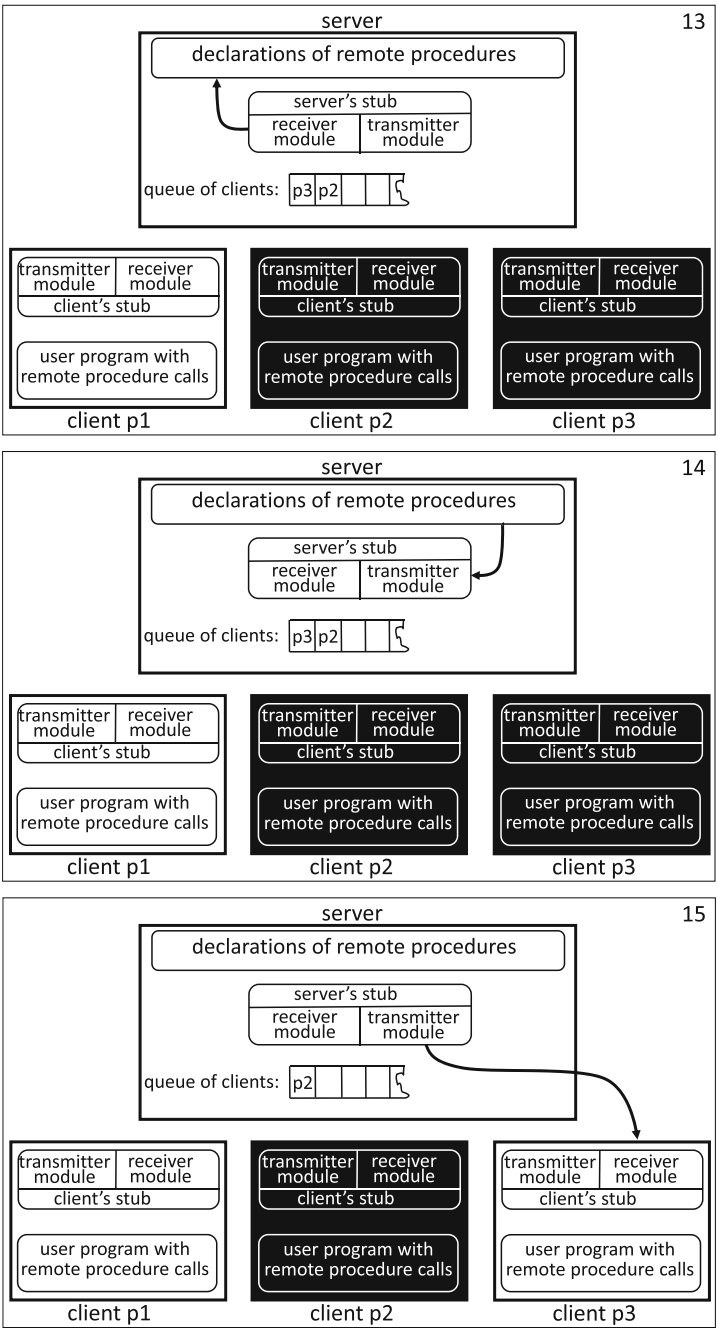
(continued)

Table 6.1 (continued)



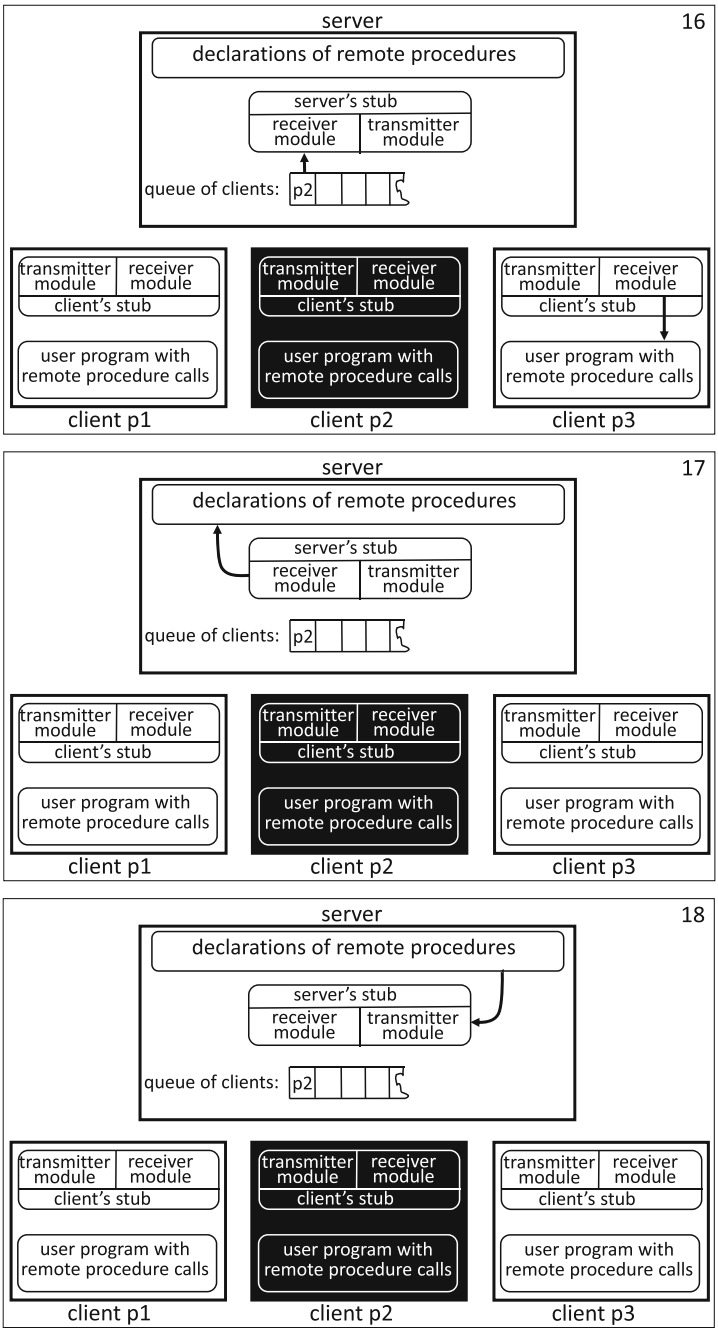
(continued)

Table 6.1 (continued)



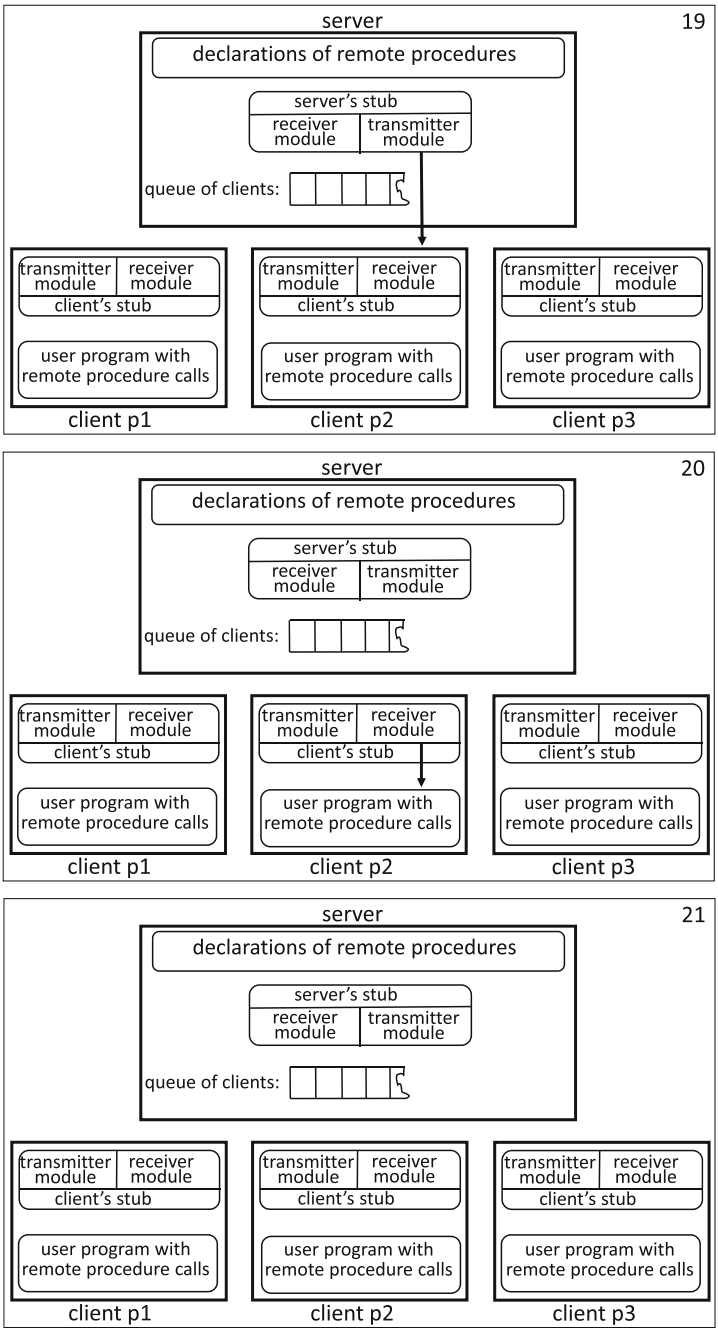
(continued)

Table 6.1 (continued)



(continued)

Table 6.1 (continued)



References

The Advance Network System Architecture (ANSA). (1989). *Reference Manual*. Castle Hill, Cambridge England. Architecture Project Management.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The java programming language*. Addison Wesley Professional.

Barnes, J. (2005). *Programming in Ada*. Addison-Wesley.

Bartol, W. M., et al. (1984). *LOGLAN'82. Report on the Loglan 82 programming language,* Warszawa-Lodz, PWN.

Ben-Ari, M. (1990). *Principles of concurrent and distributed programming*, Prentice-Hall.

Birell, A., & Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems, 2*(2), 39–59.

Hansen, P. B. (1975). The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, *1*(2), 199–207 (June 1975).

Coulouris, G., Dollimore, J., & Kindberg, T. (1994). *Distributed systems*. Addison Wesley Longman Limited: Concepts and Design.

Davidson, A., Drake, K., Roberts, W., Slater, M. (1992). *Distributed windows system, a practical guide to X11 and open windows*, Wokingham Addison-Wesley.

Kreczmar, A., Salwicki, A., Warpechowski, M. (1990). *Loglan'88—Report on the programming language,* Lecture Notes on Computer Science (Vol. 414). Springer.

Liskov, B., Shrira, L. (1988). *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*, Proc. ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, Georgia.

Stevens, W. R. (1997). UNIX Network Programming, Vol. 1, second edition, Prentice-Hall.

Szałas, A. (1991). *Warpechowska J: Loglan*. Warszawa: WNT. (in Polish).

Tanenbaum, A. S. (1995). *Distributed operating systems*, Prentice-Hall International, Inc.

Wirth, N. (1987). *Modula 2*, Wydanictwa Naukowo-Techniczne, Warszawa (Polish translation of author's book *Modula-2*).

Chapter 7
Failures and Damages in Distributed Systems

7.1 Chances and Kinds of Failure, Remedial Measures, Fault Tolerance

A crash or faulty activity of computer system happens in a single stand-alone or centralized installation as well as in a distributed system characterized in Chap. 2, or in arbitrary network. In the first period of information technology, simultaneous running of the same program on several disconnected computers (Fig. 1.7) was a remedy to increase degree of reliability. This concerned especially real time data processing, for instance beginnings of space flights control. With emergence of real distributed systems (end of 1970), the research and realization of more subtle methods to deal with failure situations received a strong stimulus. Numerous cooperating computers and applying multiple replicas of data, made the robustness of such systems more demanded than in case of single, stand-alone machines. Increasing of reliability by simultaneous run of the same program on several disconnected computers has been automated in distributed systems. It was accomplished also by simultaneous run of one program on several computers too but interconnected interconnected, and exchanging data for periodical comparison of intermediate results. So, the group (multicast) communication (Sect. 5.4) has been applied. However the increase of reliability in this way, may reduce expected effect, because of possible erroneous message passing between the members of the group. Moreover, the more components of hardware and software and the longer time of work, the greater chance of faulty behaviour of some of them. Though this is intuitively evident, a quantitative estimation may look as follows (some mathematical notions and facts needed here, are in Chap. 10):

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_7

7.1.1 Probability of System's Defective Activity; Expected Time up to a Breakdown

Suppose that on the basis of a number of tests, an estimation has been made that during a fixed time period Δt, the average probability of one component's damage is p ($0 \leq p \leq 1$). Then the probability of failure-free activity of the system containing n components in the period Δt is $B(n)$ where $B(1) = 1 - p$, $B(n + 1) = B(n)\cdot(1 - p)$, thus $B(n) = (1 - p)^n$ provided that damage of the components are independent events. Probability of the whole system failure in the period Δt is then $A(n) = 1 - B(n)$. Therefore, if $p > 0$ then $\lim_{n\to\infty} A(n) = 1$, which corresponds to intuition: along with increase of the system's scale, the chance of damage of a certain component tends to certainty. If the probabilities of individual components damage are $p_1, p_2, \ldots, p_n$ then

$$p = \frac{\sum_{i=1}^{n} p_i}{n}$$

is the average probability of damage of one component during the period Δt.

The probability of the whole system failure in the period Δt is

$$A(n) = 1 - \prod_{i=1}^{n} (1 - p_i)$$

Now, let us estimate an expected time up to damage of a single component in the system. Suppose as previously, that the average probability of one component's damage in a period Δt is p and consider the chance of its damage during multiple of this period. For conspicuity of reasoning, let us draw the interval Δt as a segment $\vdash\Delta t\dashv$ of the time axis.

So, we have:

the chance of failure-free work in the period $\vdash\Delta t\dashv$ is $1 - p$

the chance of failure-free work in the period $\vdash\Delta t \dashv\vdash\Delta t \dashv$ is $p(1 - p)^1$

the chance of failure-free work in the period $\vdash\Delta t \dashv\vdash\Delta t \dashv\vdash\Delta t \dashv$ is $p(1 - p)^2$

$\vdots$

the chance of failure-free work in the period $\underbrace{\vdash\Delta t \dashv\vdash\Delta t \dashv\vdash\Delta t \dashv\ldots\vdash\Delta t \dashv}_{k+1 \text{ periods}}$ is $p(1 - p)^k$

This is the probability of the component's damage not sooner than after elapse of the period $(k + 1)\Delta t$.

Therefore $\lim_{k\to\infty} p(1 - p)^k = 0$. First, let us calculate the expected value (i.e. average) of time until damage of one component (for necessary mathematical notions see Chap. 10). Elementary events $\omega_1, \omega_2, \omega_3, \ldots$ in this case are correct work of the component in the periods Δt, $2\Delta t$ t, $3\Delta t$, … respectively. To this end, the discrete random variable as the function

$X : \Omega \xrightarrow{onto} \boldsymbol{S}_X \subset \boldsymbol{R}$ where $\Omega = \{\omega_1, \omega_2, \omega_3, \ldots\}$, $\boldsymbol{S}_X = \{\Delta t, 2\Delta t, 3\Delta t, \ldots\}$, ($\boldsymbol{R}$—set of real numbers), is defined as $X(\omega_k) = (k + 1)\Delta t$, $k = 1, 2, 3, \ldots$.

Since probability of correct work of the component in the period $(k + 1)\Delta t$ (i.e. the probability p_k of assuming this value by random variable X) is $p(1 - p)^k$, thus, by definition of expected value $E(X)$, the expected time up to a damage of one component is

$$E(X) = \sum_{k=0}^{\infty} (k+1)\Delta t \cdot p(1-p)^k = p\Delta t \sum_{k=0}^{\infty} (k+1)(1-p)^k$$

Because in case of this problem, the expected value depends only on p and Δt, therefore instead of $E(X)$ we will write $E(p, \Delta t)$. The series $\sum_{k=0}^{\infty} (k+1)(1-p)^k$ is absolutely convergent which follows from the d'Alambert criterion (Chap. 10), because $\frac{(k+1)(1-p)^k}{k(1-p)^{k-1}} = (1 + \frac{1}{k})(1 - p)$ thus $\lim\limits_{k\to\infty} \left(1 + \frac{1}{k}\right)(1 - p) = 1 - p < 1$.

By Mertens theorem (example in Sect. 10.2.2) we get:

$$\sum_{k=0}^{\infty} (k+1)(1-p)^k = \left(\frac{1}{1-(1-p)}\right)^2 = \frac{1}{p^2}$$

Finally, the expected value of time up to a damage of one component is

$$E(p, \Delta t) = \frac{\Delta t}{p}$$

Since the probability of a damage of the system of n-components in the period Δt is $A(n) = 1 - \prod_{i=1}^{n} (1 - p_n)$ and in the considered case $p_i = p$, thus, if the average probability of a damage of one component is p then the expected value of time up to a damage of the n-components system is

$$E(p, \Delta t, n) = \frac{\Delta t}{1 - (1-p)^n}$$

7.1.2 Kinds of Failure and Some Mechanisms of the Fault-Tolerant Systems

The components, like computers (containing many constituents), routers, switches, modems, amplifiers, cables, devices of radio transmission, operating system modules, compilers, applicative programs, etc. may succumb failures (crash or faulty activity) of the following kinds:

- **Permanent**—requires replacement (or removal) of the faulty component from the system,
- **temporary**—a simple repair suffices, e.g. restoring lost contact in some connexions, removing viruses, etc.
- **transient**—disappears itself after a little while, e.g. disturbance of radio waves as result of atmospheric phenomena, etc.
- **byzantine**—requires special algorithms deciding if continuation of correct system run is possible in case of defectively acting components; if yes—the algorithms allow for the continuation (without notification of the user about the failure), otherwise—they notify the user about the situation; this kind of failure is considered in Sect. 7.2.2.

Some failures may be removed by the system itself, if it is endowed with appropriate mechanisms. The user is not notified of this circumstance (transparency). Designing such mechanisms for reliable functioning of the system is indispensable because great number hardware and software components and long lifetime of systems may entail faulty activity of a system as a whole. As always, the designers face decision: the system should enjoy high degree of transparency, that is to automatize this service for a price of lower efficiency, or—on the contrary—more important is efficiency. In case of the latter, a user receives messages from the system about undesirable situations and undertakes some remedial measures e.g. by programming suitable exception handling or using library procedures. However the primary goal is assurance of correct (consistent with specification) system behaviour, also in case of erroneous activity of some hardware or software components. A system making possible correct behaviour when some of its components become faulty, is called the *fault-tolerant*. Such goal not always can be achieved. It happens when the system behaves accidentally or undesirably stops working. No absolute resistance against the damage exists—we say only of errors tolerable up to greater or less degree. In Chap. 6 some exceptional situations during remote procedure calls were described and in Chap. 9, an example of coping with messages lost during transmission or with repeated delivery of the same message, will be shown. To handle such cases, a number of algorithms (protocols) have been devised, which make replication of lost packets, elimination of multiple reception and ensure delivery in the order of dispatch. The so-called Alternating Bit Protocol (Chap. 9) is an example of such algorithms. If a faulty behaviour appears, to hide it from the user, a mechanism removing its outcomes is needed—if possible. Depending on a kind of failure and a strategy of handling it, such mechanism may proceed as follows:

- The mechanism periodically (often!) records the state of system in a permanent memory and represents it by a certain identifier (e.g. a timestamp) called a *restoration point* or a *checkpoint*. After a fault detection, the mechanism restores the state from before its occurrence. This is the so-called *backward recovery*, applied also in operating systems of stand-alone computers, where too subtle algorithms are not necessary. However, it may cause the so-called *domino effect*,

if functioning of the process depends on other processes: withdrawal of a process state entails withdrawal of states of cooperating processes. Restoration of the distributed system state, is much more complicated than in case of stand-alone machines. For this purpose is needed a register keeping restoration points stored by the local operating systems. The algorithms that use this register must ensure the proper ordering of events: in the restored system state, for each event of message sending, an event of this message reception exists and conversely, and the dispatch must precede (in the sense of relation $\rightsquigarrow$ see Sect. 4.2 in Chap. 4) the reception. The sequence of restoration points used to restore the system state is called a (restoration) *line* (Randel et al. 1978). The choice of this line depends on a strategy of failure handling, but good algorithms choose the latter one from before the failure. So, the backward recovery is expensive in terms of process time and memory size.

- The mechanism modifies the state of breakdown so, that the system, starting from the modified state, would work correctly. This requires identification of expected effects of erroneous functioning of components, which may disturb further correct activity of the system as a whole and preclude some decisions of failure prevention. Such proceeding is called a ***forward recovery.*** Appropriate algorithms are complicated and also costly in terms of process time and memory size.
- The mechanism conveys the function of faulty components or completely inactive to reserve (or redundant) components or to components elected by application of special algorithms (described in Sect. 7.3), started by a process requesting service from a faulty provider.
- The mechanism signals a *crash* (*indivertible failure*) if it cannot withdraw defective components from the system and cannot automatically convey their activity onto faultless components.

The mechanism removing failure outcome, should accomplish its work in time admissible by the system users and, moreover, make it transparent. Designers of such mechanism face a difficult task that demand ingenuity in conciliation of often contradictory tendencies. Notice that not every exception is a failure, for instance the result of dividing by zero is an exception, but is not regarded as a failure. Notice also that creation of backup copies in a word processor is not creation of checkpoints, in the above meaning. As said above, damage of hardware or software may be permanent, temporary or transient. Messages may disappear or be corrupted during transmission, computers may get damaged or work randomly (out of control), some processes whose synchronization is required, may proceed without any synchronization. Assurance of high degree of resistance against defective or accidental behaviour of cooperating computers is an important tasks of failure handling. The difficulty of this task results from disability of operating systems to analyse semantic correctness of user programs: no operating system knows whether the computer is working in accordance with the program specification or accidently. That is why more dangerous and hard to detect is accidental (“frantic”) activity of a computer than to detect its crash and work stop. The security threat is more serious

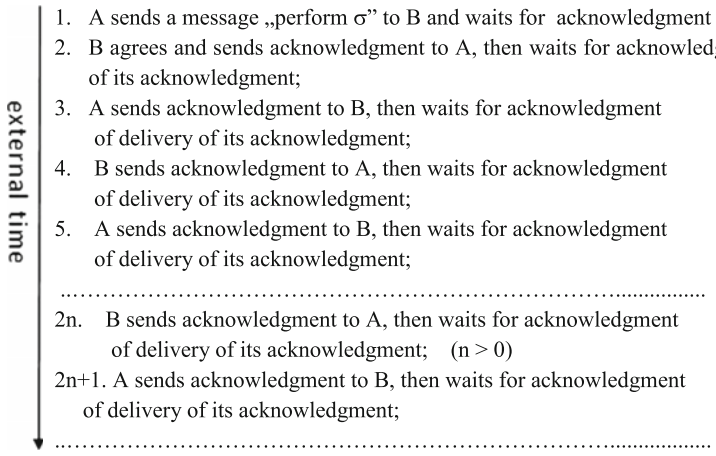
in case when computers get out of control than in the case of their undesired work termination. This is referred to as the *byzantine failure*. Another situation to cope with is when a simultaneity of some actions is required: there is no algorithm assuring the exact simultaneity. It may only be approximated. In the following sections such issues are discussed.

7.2 Some Problems of Activity Coordination

Coordination of processes in order to perform a joint action, to reach certain state or to protect some resources, has been considered in Chaps. 3 and 4, where mutual exclusion, deadlock, starvation and synchronization of clocks were discussed. Some other instances of coordination are when a group of interconnected computers are performing the same program with the same data, so that to enhance reliability by periodical scrutiny sameness of behaviour, when the group is to make simultaneity of some actions, or to reach an agreement on a joint activity and when a group is to elect a new coordinator after a crash of the current one. The next two subsections are concerned with simultaneous execution of some actions and with coming to agreement on a joint activity, whereas the succeeding section—election of a new coordinator after a crash of the current one. Notice that problems of the simultaneity and agreement on a joint activity, are, in a sense reverse to the mutual exclusion, in which more than one process *cannot* execute certain activity simultaneously.

7.2.1 Infinite Cycle of Confirmations—the "Two Army" Problem

Assume that armies A and B can communicate and should coordinate a joint action σ, (e.g. a gunfire), but a moment of this event is not determined. Obviously, the armies are a metaphor of computers, so let us refer to A and B as computers. Action σ should be performed (perhaps simultaneously) by A and B, but transmission is not instantaneous—it takes some time. The computers are working faultlessly, but the transmission channels may be faulty ("the enemy can intercept the messengers"), thus the computers require acknowledgment of delivery of each message. The coordination actions are the following:



moment 1 A and B cannot perform action σ (evident)
moment 2 A cannot perform σ since does not know if B got the message
moment 3 B cannot perform σ since does not know if A got the acknowledgment
moment 4 A cannot perform σ since does not know if B got the acknowledgment
moment 5 B cannot perform σ since does not know if A got the acknowledgment
etc. to infinity if the channel is unreliable and the computers work faultlessly.

Never the agreement on the joint action σ will take place, irrespectively of reliability of the channel (requirement of acknowledgments means that the computers „do not know" whether or not the channel is reliable). This is because a moment of the action ("gunfire") has not been determined and any message cannot reach the receiver instantaneously. The chain of communications A→B→A→B…. might be interrupted by a fault during message transmission (a message was undelivered) and will be interrupted if a certain computer stops working. Then the joint action σ does not take effect either. If the computer A inserts a moment T of execution of σ into the message "execute σ", and time T is later than time of dispatch of this message increased by time of transmission to B and return with acknowledgment of delivery, then the joint action σ will not take effect either, even when:

– the local clocks of computers are well synchronized and indicate the same time during exchange of messages;
– computer A receives acknowledgment from B before the moment T.

Why? Because B does not know if its acknowledgment reached A before T, therefore must wait for confirmation of this, etc. The action σ might have been simultaneously executed under completely unrealistic assumptions: the computers

know the time of transmission and moment of the action, their clocks are exactly synchronized and they know that the mail does not disappear. In this case, the acknowledgments are superfluous.

> The above example is a reformulation of the known "two army" problem, which originally appeared in (Akkoyunlu et al. 1975) (where instead of armies, two gangs were acting) and has been considered in many writings, e.g. (Tanenbaum 1995).

Although the impossibility of simultaneous action σ is pretty obvious, the formal proof of this may be carried out (by contradiction) as follows. Suppose there exists a finite chain of transmissions:

$$A \xrightarrow{\sigma} B \xrightarrow{\pi^1} A \xrightarrow{\pi^2} B \xrightarrow{\pi^3} \cdots \xrightarrow{\pi^{n-1}} A \xrightarrow{\pi^n} B$$

which ensures agreement between computers A and B for simultaneous execution of σ and let this chain be the shortest (i.e. any shorter does not ensure the agreement), where π^i denotes acknowledgment of reception of acknowledgment π^{i-1} ($i > 1$), and π^1 is the acknowledgment of reception of message σ from A by B.

Thus, the computers can start action σ, when computer A, had sent π^n. On the other hand, if π^n has not reached B, then B cannot perform action σ. But if π^n has reached B then B also cannot perform σ, because B knows that computer A after having sent π^n, does not know whether or not π^n has reached B (because B has not sent acknowledgment). Therefore the above chain does not ensure agreement: a contradiction, which ends the proof.

Obviously, if the minimal finite chain ends at A, then the reasoning remains valid. Notice that if there are not two computers involved, but A_1, A_2,…, A_m, this formal reasoning of impossibility of reaching agreement, requires similar reasoning for $m - 1$ of chains that begin from a certain computer. Computers A_2,…, A_m are communicating with A_1 only, but not between themselves.

Although coming to agreement on joint simultaneous action in finite time is theoretically impossible under above assumptions, this is sometimes necessary in some applications, for instance when approval of bank transactions is required. Diverse methods are being then applied, e.g. dispatch of a stream of messages instead of one and waiting for acknowledgment of everyone. In this case, a minimal number of messages in the stream is being estimated in order to obtain high probability of reaching agreement. A certain predictable degree of uncertainty of transmission must be then assumed.

Another task of reaching consensus of simultaneous actions is the so-called *Firing Squad Problem*, discussed in a number of publications, for instance Waksman (1966), Moore and Langdon (1968), Balzer (1967).

7.2.2 *Reaching Consensus with Fault Tolerance—the "Byzantine Generals" Problem*

In the "two army" problem it was assumed that the computers perform correctly, but deceptive may turn out communication channels. In the problem of "Byzantine Generals", communication channels are reliable, thus confirmations are not required, but the computers may perform incorrectly. Their activity may be random (get out of control), which is difficult or impossible to detect. As before, the task is to reach consensus in undertaking a simultaneous action by a group of computers. There are a number of versions and formulations of this problem, for instance Lamport et al. (1982), Fisher et al. (1985), Garg and Bridgman (2011), Tanenbaum (1995), Coulouris et al. (1994), Ben-Ari (1990). Let us begin considerations close to presented in Lamport et al. (1982), the first where the problem has been posed and solved. The computers („generals") are to agree a common strategy leading to coordination of a „military action"- one of several possible. They may also pass information on a manpower of their strike force. In the ideal situation each computer is reliable, so, sends its vote for one and the same strategy to the remaining computers and notes down a number of votes for particular strategies received from them, including its own vote. In this way the computers make records of votes given to particular strategies. Each computer gets identical record of votes, because, as reliable, sends the same vote to the remaining computers. When exactly one strategy obtained majority of votes, the computers came to agreement, if more than one—they failed. In Fig. 7.1a, computers G_1, G_2, G_3, G_4 (the generals) are to agree on one of strategies *a*, *b*, *c*, whereas in Fig. 7.1b—on one of *a*, *b*. In the case (a) the set of votes collected by every computer contains 2 votes for strategy *a*, 1 vote for *b* and 1 vote for *c*, thus strategy *a* has been agreed-upon. In the case (b) the set of votes collected by every computer contains 2 votes for strategy *a* and 2 votes for *b*; a strategy has not been agreed-upon.

But a decision has to be made, so, what is to be done? It depends on the kind of strategy. If the strategies *a*, *b* concern only a strike force of the armies, then nothing

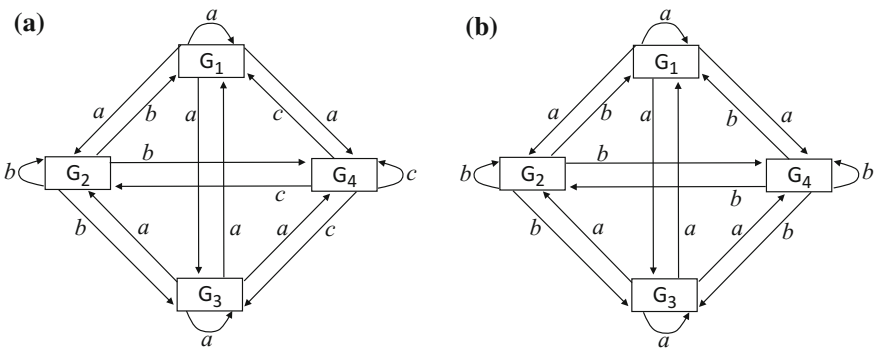


Fig. 7.1 Faultless computers ("loyal generals"); (a) strategy "a" outvoted, (b) no strategy outvoted

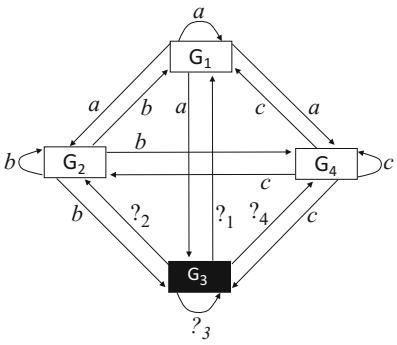
is to be done—such information will be used for fixing a joint strategy. If, however, they concern realization of a certain action then it is possible for instance:

- to repeat the voting;
- to convey decision to a co-ordinator (chief general), who resolves voting result; the co-ordinator's verdict is being broadcast to remaining generals;
- if weights are assigned to strategies (e.g. strength of the armies under command of the generals), then the decision may depend on the weights; in some versions, the weights may also be assigned to computers, as considered in Garg and Bridgman (2011).

If all the computers are reliable, that is if no one behaves unforeseeably and communicates the same vote to all remaining computers, then their decisions lead to a common strategy. Difficulties occur if some computers are unreliable: they are running but chaotically, randomly; in particular they communicate various proposals of strategy to their partners or convey incidental strategies (those are not received from the coordinator). There are a number of this problem versions, depending on various assumptions. Apart from the reliability of communication channels (thus when no delivery confirmation is required), let us consider a version under the following assumptions:

(i) no computer has been appointed as a coordinator—all have equal rights;
(ii) each computer sends messages to all others, as well as to itself;
(iii) some computers are unreliable, so, they may send different messages to different receivers;
(iv) each receiver knows from whom received a message (messages are „signed") —an identifier of the sender is attached to the message;

Let computer G_3 be unreliable ("treacherous general") and let it broadcast to computers G_1, G_2, G_3, G_4 proposals of strategies $?_1$, $?_2$, $?_3$, $?_4$ respectively, as shown in Fig. 7.2.



The graph shows a process of Table A creation:

receiver \ sender	G_1	G_2	G_3	G_4
G_1	*a*	*b*	$?_1$	*c*
G_2	*a*	*b*	$?_2$	*c*
G_3	*a*	*b*	$?_3$	*c*
G_4	*a*	*b*	$?_4$	*c*

Table A

Fig. 7.2 The graph presents voting of computers G_1, G_2, G_3, G_4, with results in the Table A

The rows corresponding to receivers contain proposals of strategy dispatched by senders corresponding to the columns. Every reliable computer sends the same proposal to all receivers. The unreliable, may send different proposals to different receivers. After having created Table A, none computer may infer from this table, which computer is unreliable and which strategies the computers had sent most frequently, because none of them knows what strategies had been received by its partners. So, every reliable computer sends to the remaining computers (and to itself), its row of Table A, that is, the vector of four received proposals of strategy. The unreliable sender may send different (incidental) vectors to different receivers. In this way Table B in Fig. 7.3 is created with identical rows, except for (possibly) entries in columns corresponding to unreliable computers, which may send different vectors. In the example it is the column G_3, what every receiver can see in its own row of the Table B. Therefore:

G_1 sees that G_3 had sent $?_1$, $?_2$, $?_7$, $?_4$ to G_1, G_2, G_3, G_4 respectively
G_2 sees that G_3 had sent $?_1$, $?_2$, $?_{11}$, $?_4$ to G_1, G_2, G_3, G_4 respectively
G_3 sees that G_3 had sent $?_1$, $?_2$, $?_{15}$, $?_4$ to G_1, G_2, G_3, G_4 respectively
G_4 sees that G_3 had sent $?_1$, $?_2$, $?_{19}$, $?_4$ to G_1, G_2, G_3, G_4 respectively

sender

receiver	G_1	G_2	G_3	G_4
G_1	$[a, b, ?_1, c]$	$[a, b, ?_2, c]$	$[?_5, ?_6, ?_7, ?_8]$	$[a, b, ?_4, c]$
G_2	$[a, b, ?_1, c]$	$[a, b, ?_2, c]$	$[?_9, ?_{10}, ?_{11}, ?_{12}]$	$[a, b, ?_4, c]$
G_3	$[a, b, ?_1, c]$	$[a, b, ?_2, c]$	$[?_{13}, ?_{14}, ?_{15}, ?_{16}]$	$[a, b, ?_4, c]$
G_4	$[a, b, ?_1, c]$	$[a, b, ?_2, c]$	$[?_{17}, ?_{18}, ?_{19}, ?_{20}]$	$[a, b, ?_4, c]$

Table B

Fig. 7.3 Rows G_1, G_2, G_4 contain vectors, each being the respective row of the Table A; also row G_3, save for the vector in the column corresponding to unreliable G_3; this vector has incidental components $?_{13}$, $?_{14}$, $?_{15}$, $?_{16}$; every reliable general notices the identical voting result by reliable generals

The graph in Fig. 7.4 shows process of Table B creation, that is the renewed voting, this time for vectors of strategies.

The assumption that the computers know from whom they have received the messages, was necessary for creation Tables A and B. Without this assumption the computers would not be able to arrange the messages as sequences, i.e. vectors, in which a place (component position) corresponds to one sender. Thus, it would result in a set, not a vector.

Remembering assumptions i–iv let us sum up what information every computer has collected in the Table B, what can be inferred from this information and how the computers can utilize this in order to come to an agreement. Therefore:

- Each receiver has access to information contained only in its row of Table B.
- If a certain sender had not sent the same value to all receivers, then each reliable receiver would have noticed this in its row of the Table B; thus the receiver

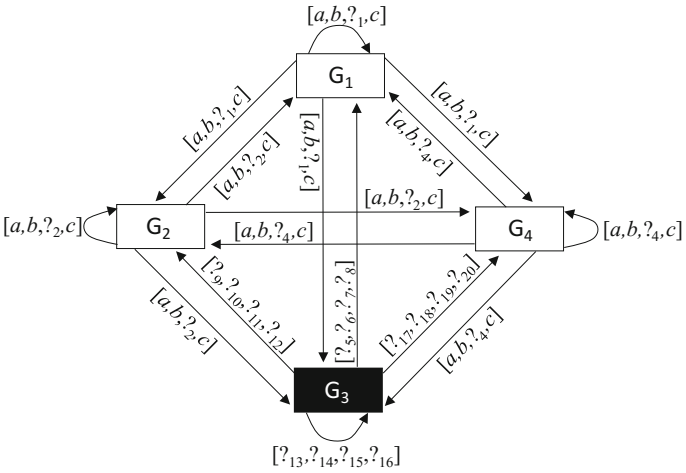


Fig. 7.4 The graph presents voting of computers G_1, G_2, G_3, G_4, with results in the Table B

states that the sender is unreliable. This suffices for the joint knowledge on unreliability of the sender. However, this is not the necessary condition. Even if none of the senders had not sent different messages to different receivers (during creation of Table A), then nonetheless different receivers could have received different vectors (during creation of Table B). If all computers were reliable then all vectors in the Table B would be identical, but not always conversely. This is a global information: no computer can ascertain this situation. Any computer can see its own row of the Table B only. For instance, if $?_1 \neq ?_2$ then everyone sees that G_3 is unreliable. But also, if $?_1 = ?_2 = ?_3 = ?_4$ but e.g. [$?_5$, $?_6$, $?_7$, $?_8$] $\neq$ [$?_9$, $?_{10}$, $?_{11}$, $?_{12}$] (G_1 i G_2 are unable to ascertain this), then G_3 is unreliable. This may happen when during creation of the Table A, the computer G_3 was behaving correctly, but then got spoiled. Thus:

- A group of computers not always can pinpoint the unreliable („generals-traitors") from among themselves and, for instance, exclude them from making decision on the basis of proposed strategies.
- Every computer can ascertain only which values (strategies) have been sent by remaining computers most frequently: which occur most often in the received vectors. Sending a value is understood as „giving a vote" for a certain strategy.
- Every computer sees that values *a*, *b*, *c* occur in the two vectors received from two remaining computers. But every value denoted by "?" sent by G_3 occurs in one vector if differs from *a*, *b*, *c* and if are different among themselves. This means that G_1, G_2, G_4 have dominated („outvoted") computer G_3 in proposing strategies. We say that the three computers „tolerate" G_3, that is, do not exclude it from the system but ignore its behaviour.

- Such proceeding may lead to an agreement in choosing some values, but cannot ensure this, because some values sent by unreliable computers may occur in majority of vectors seen by a certain computer.
- It can be proved that such proceeding makes possible agreement only when in a group of computers, with M unreliable, at least $2M + 1$ are reliable. Thus, this is a necessary condition for possibility of agreement by means of the presented above method of „voting". The seminal article, where several versions of the Byzantine Generals problem was discussed with proofs of results is Lamport et al. (1982). In the above example, three computers are reliable and one is unreliable ($M = 1$ and $2M + 1 = 3$), the agreement is possible: they agreed for strategies a, b, c.

The reader can check how the tables A and B would have look like, if two computers in Fig. 7.2 were unreliable, e.g. also G4. In such case, although the reliable might be able to identify the unreliable, but making decision on the basis of majority of votes is generally impossible: the majority may not exist. So, the reliable computers would not be able to "tolerate" the unreliable ones.

Creating table B (Fig. 7.3) required two voting rounds. Theoretically, one may imagine continuation of the procedure, so that a next table might be similarly created from the table B by means of the third round, etc. Bearing in mind above remarks, the successive rounds (though practically hardly acceptable) might detect a "frantic" behaviour of some computers, if it had not been detected in the previous rounds. This (potentially endless) proceedings may look as in Fig. 7.5.

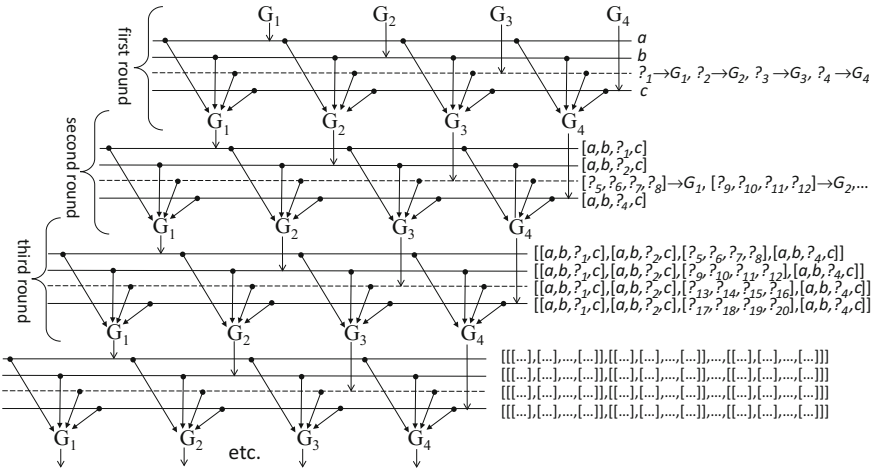


Fig. 7.5 Horizontal lines („buses") correspond to the computers (generals); each computer broadcasts a message (a strategy or vector of strategies) onto its line, from which the computers take it and compose vectors for the next round. The dashed line corresponds to the unreliable computer

So far it has been assumed that each computer knows from whom a message was received. If this is not so, then in the Table B, will occur sets instead of vectors (sequences). For instance, instead of vector $[a, b, ?_1, c]$ the set $\{a, b, ?_1, c\}$ will appear (and if e.g. $?_1 = a$ then the set $\{a, b, c\}$), so, it is not known which computer has sent particular elements of this set. In such case each computer can see which values are in the majority in the sets, but does not know who has sent them. Thus no computer is able to conclude which computer is unreliable. If each computer knows who was a sender of particular message, we say that the messages are „signed".

Let us consider another version of the problem, where one computer is appointed as a coordinator, the „General", while others, the „Lieutenants", depend on him. As previously, some computers are reliable ("loyal"), some—do not ("traitors"). The general starts the election by sending a strategy "attack" or "retreat" to all lieutenants. The loyal general and lieutenants are broadcasting this strategy to all remaining lieutenants without change, but the traitors may convey different strategies to different lieutenants. The lieutenants are unaware who sent them a given strategy unless this was the general. So, the strategy was "signed" only by the general, whereas other senders are anonymous. Voting consists in sending a vote in favor of one strategy. The lieutenants must select one strategy on the basis of majority of votes received. If all lieutenants select the same strategy, then they reach agreement as to the military action, otherwise—they do not. Examples of various cases of this version are shown in Figs. 7.6a, b, 7.7c, d and 7.8e, f. We see that though the computers cannot pinpoint the traitors, sometimes can ignore ("tolerate") the traitors' votes.

The cases (a), (b) in Figs. 7.6 and 7.7 (c) show a system with one loyal General, two loyal Lieutenants and one traitor (black). Every loyal Lieutenant conveys the general's decision unchanged to remaining Lieutenants, the treacherous Lieutenant falsifies it. So, regardless of this falsification, the voting result by all Lieutenants would be the same—consistent with the General's decision. In these cases $M = 1$, thus $2M + 1 = 3$, therefore, similarly to the previous version, the necessary

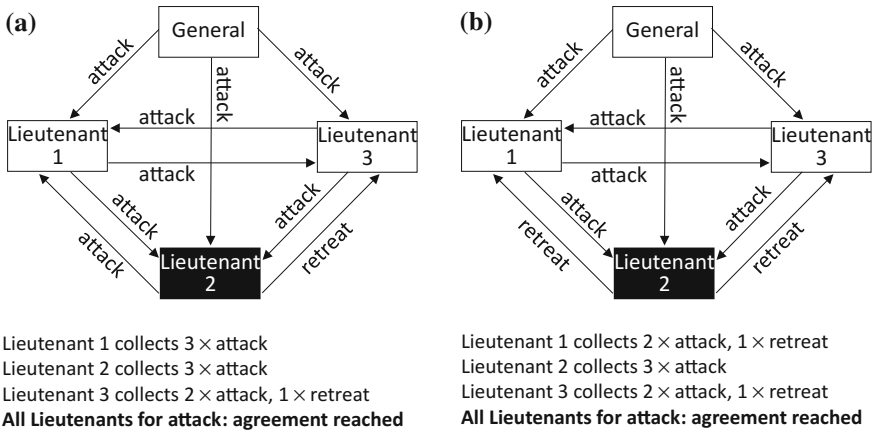
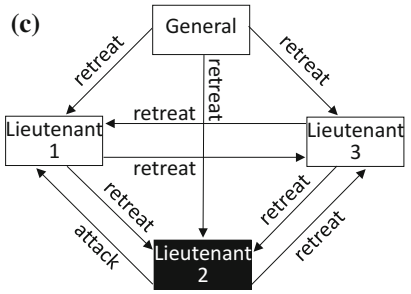


Fig. 7.6 (a), (b) one treacherous Lieutenant: agreement



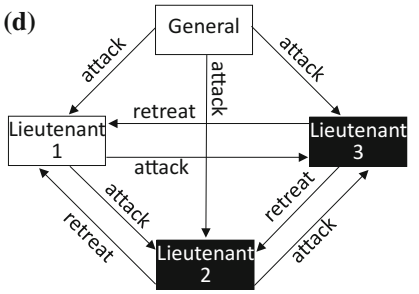
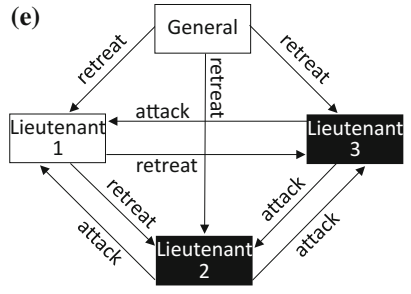


Fig. 7.7 (c) one treacherous Lieutenant: agreement; (d) two treacherous Lieutenants: no agreement



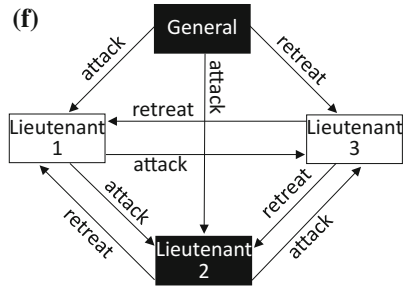


Fig. 7.8 (e) two treacherous Lieutenants: no agreement; (f) treacherous General and one Lieutenant; no agreement

condition for reaching agreement is fulfilled. In the case (d) in Fig. 7.7 the treacherous Lieutenants have collected majority votes for "attack", while the loyal Lieutenant—for "retreat", whereas in the case (e) in Fig. 7.8—conversely. In the case (f) the General and Lieutenant 2 are traitors. The Lieutenants 2 and 3 have collected majority of "attack", while the Lieutenant 1—"retreat". In these cases $M = 2$, thus $2M + 1 = 5$, therefore, the necessary condition for reaching agreement is not fulfilled.

Notice that if the Lieutenants have conveyed result of their voting also to the General and to themselves, the final outcome: agreement or its lack, would be the same. Thus, for the systems in Figs. 7.6, 7.7 and 7.8, the tables A and B may be created (but for two strategies)—as in the previous version for the system in Fig. 7.2. The outcome of voting (thus the agreement) would be the same as in the version with the general and lieutenants. It may be proved that the previous version (in which all the generals enjoy equal rights and there are several possible strategies and all senders are known to the receivers) can be reduced to the version with the general and lieutenants.

This example, generalized to other configurations and any number of "soldiers", reaffirms validity of the general fact: under assumptions admitted above, agreement is possible only if in a group of computers containing M traitors, at least $2M + 1$ are loyal. A formal proof, fairly long (based on a construction and analysis of several recursive algorithms) may be found in Lamport et al. (1982). An extensive presentation of the Byzantine Fault Tolerance problem is in Cachin et al. (2006).

Examples of systems where consensus of some actions is required but some computers may behave chaotically and where BFT (Byzantine Fault Tolerance) algorithms are applied to tolerate their behaviour

- Some systems of flight control (Airplane Information Management System) of Boeing 777 and 787; as the real-time systems to ensure flight security, they require fast reaction to events.
- Some systems of a spacecraft (SpaceX Dragon and NASA Crew Exploration Vehicle—in elaboration); exposed on cosmic radiation, they should assure possibly reliable activity by fast automatic reaction and removal of disturbances.
- A system launched in 2015 for VISA card holders, a network system of peer-to-peer fast cash remittance; requires utmost reliability against possible disturbances, also due to criminal acts.
- Distributed measuring system where tolerance of faulty work of sensors is required.

Summarizing the above considerations one can see that the main purpose of the Byzantine Fault Tolerance (BFT) methods is to automatically ignore computers behaving chaotically if it is possible (remember the necessary condition for this), without wasting time for repairing them or removing from the system. This is especially important in the real-time systems, where the prompt reaction on events is demanded. If the "tolerance" is not possible, then the detected faulty units can be excluded from the system. If this is the coordinator (the "General"), then a new coordinator can be elected, by means of e.g. the *Bully Algorithm* or the *Ring Algorithm* presented in the next two sections.

7.3 Election of a New Coordinator Following Detection of the Damaged

In distributed systems, some services are being provided by one or more computers acting as servers. Typical examples are coordination of access to shared resources, synchronization of clocks, communication service, remote procedure call, etc. When a service provider undergoes a failure, it is necessary to assure further system work, by election of a new provider, which takes over tasks of the damaged one. For the two presented here election methods, the following assumptions are admitted:

- Every computer may take a part in the election of a computer, from among those that are endowed with a program for coordination of the system activity, like initiating some actions, sequencing events, reacting to failures, etc. After being elected, it becomes a ***coordinator*** (customarily, we say „a process" instead of „computer").
- To every process p, a unique identification number is assigned, denoted by $\#[p]$ (it may be e.g. network address of a process p) and everyone knows the ID number of the remaining processes.
- As a coordinator, a faultless process with the greatest ID number is elected. Such arrangement allows for direct reference to the coordinator, by processes requiring coordination of some activities.
- If a process requires a service from the coordinator, but does not receive a response after a predetermined time (time-out) or receives a message of the coordinator's failure, from a ***failure detector***, then it declares elections of a new coordinator.
- The elections are made by algorithms, that try to localize a faultless process of the greatest ID number and to appoint it as a coordinator.
- In the course of election, processes are sending and receiving messages among themselves via ***reliable transmission channels***, i.e. each message reaches a receiver and is not falsified during transmission. However, a process may become faulty—in such case it is not responding to messages or is ignored ("tolerated") if identified as such, by the BFT method (Sect. 7.2.2).
- A result of election is agreed-upon by all active processes.

Two successive subsections present methods of election of a new coordinator.

7.3.1 Bully Algorithm of Election

1. Process p which had discovered a failure of the coordinator, proclaims election. It sends out a message „***election***" to the group of processes bearing greater ID numbers than its own. Then waits for messages „***reply***". The process p is

unaware which receiver of its message is faultless, that is why p sends „***election***" not only directly to the process bearing the greatest ID number.
2. If after a lapse of time fixed in the system, the process p would not receive message „***reply***", then it becomes a coordinator. The coordinator notifies the processes bearing smaller ID number about the election result, by sending message „***coordinator***".
3. If the process p, before a lapse of time fixed in the system, receives the message „***reply***" from all processes to which p had sent the message "***election***", then processes which responded with this message, proclaim the election and the role of process p is over.

Remarks

- outcome of this algorithm, i.e. the winner of elections (new coordinator), has the greatest ID number among all the faultless processes; that is why it is called a „bully"—as a strongest that has conquered the remaining processes;
- the processes, after having sent the message „reply" to the process which discovered a failure of the coordinator, proclaim the election concurrently (by sending message „election" to processes bearing greater ID number). This speeds-up the choice of the new coordinator.

The Table 7.1 shows processes p1, p2, p3, p4, p5, p6 in the course of several elections, proclaimed as a result of breakdown of the current coordinators. Some processes, not being coordinators, also are breaking down during elections. The ID numbers assigned to processes, are ordered as follows: #(p1) < #(p2) < #(p3) < #(p4) < #(p5) < #(p6). The processes pass through the following states in the course of elections (Fig. 7.9):

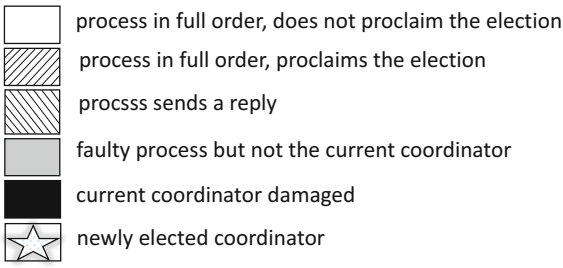
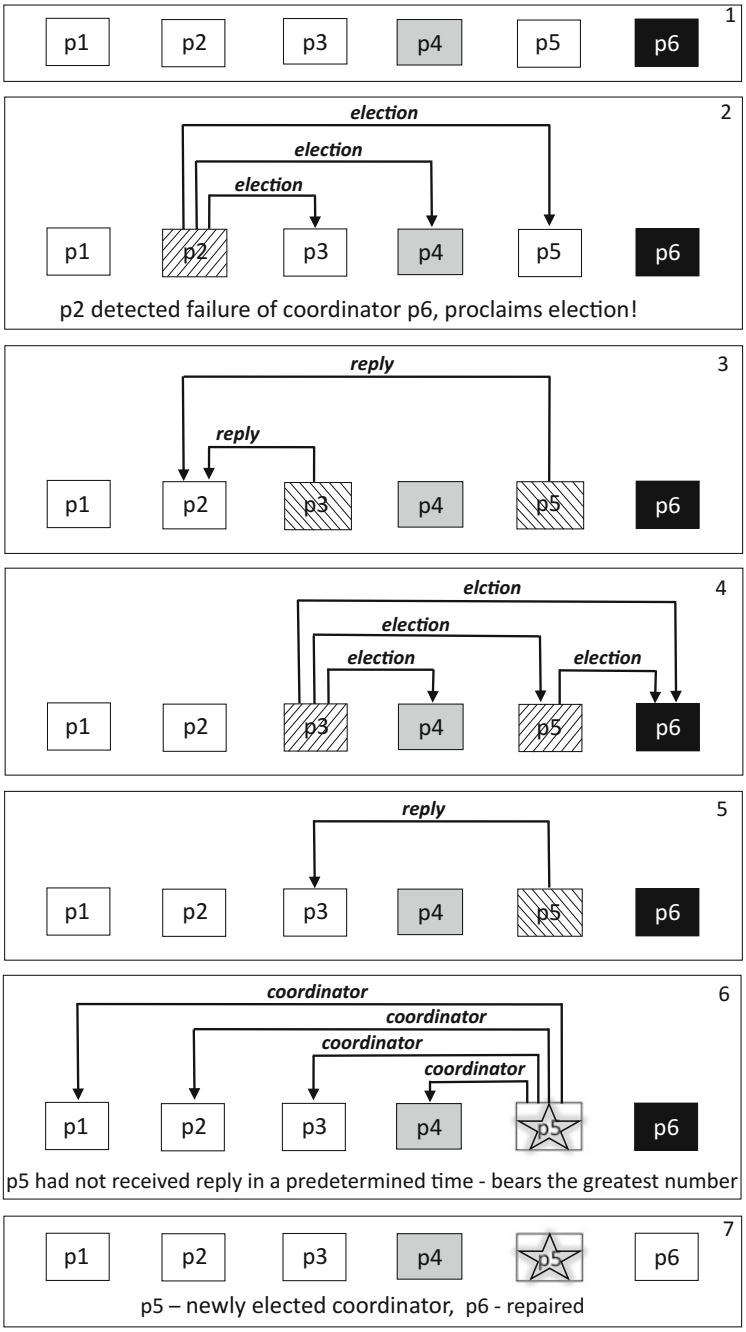


Fig. 7.9 States assumed by processes during election

Table 7.1 Election of new coordinator by the Bully algorithm

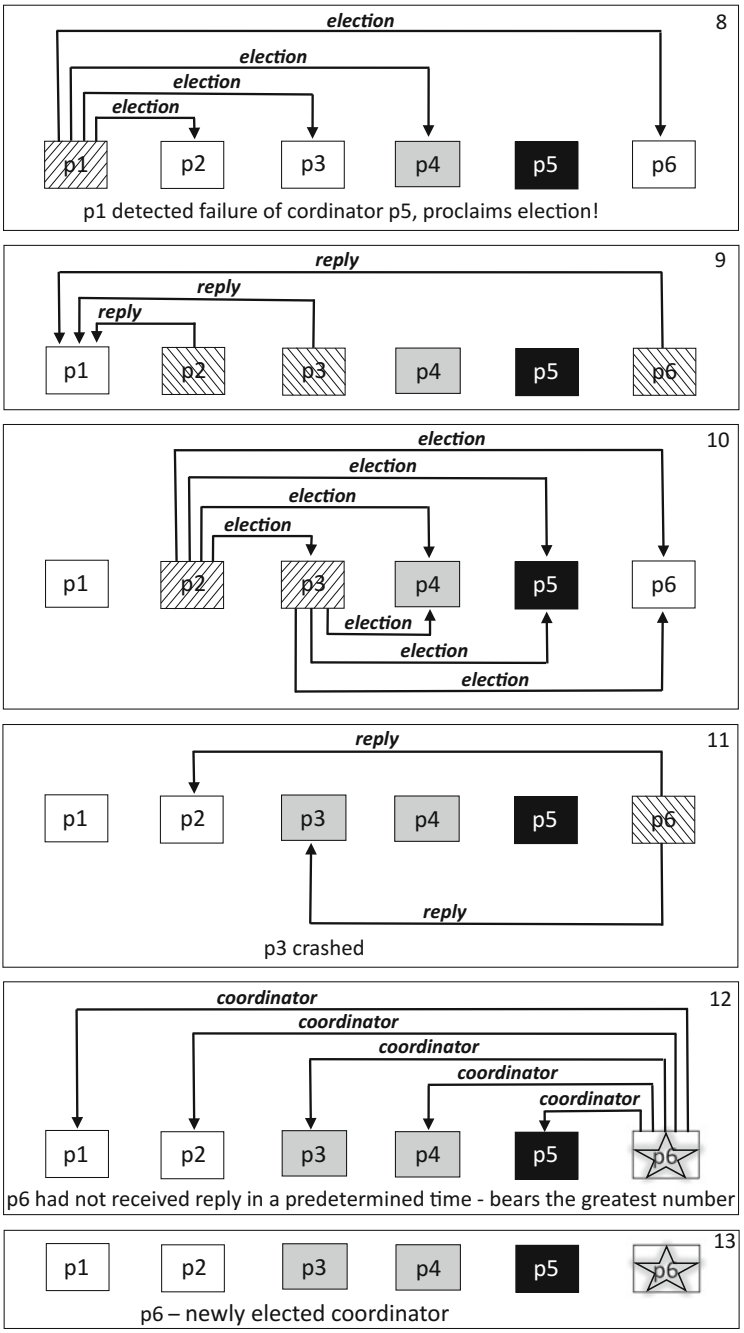
States 1–7



(continued)

Table 7.1 (continued)

States 8–13



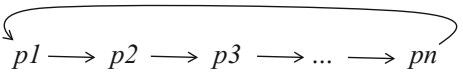


Fig. 7.10 Succession of processes in the ring

7.3.2 Ring Algorithm of Election

The processes are ordered along the ring as the arrows indicate in Fig. 7.10.

The ordering of processes along the ring has nothing common with their numbering in the Bully algorithm: although the processes are named (hold identifiers) and unique ID numbers, their ID numbers are in no relation to the succession in the ring. There is no token in the ring. As in case of the Bully algorithm, this algorithm aims at choosing a faultless process of the greatest ID number as a coordinator. In the course of election every process may find oneself at the status: ***participates***, ***not participates***, ***faulty coordinator***, ***newly elected coordinator***. Two kinds of message are being transmitted between processes:

- ***election***(#(p), q) where #(p) is the ID number of process named p, which detected the faulty coordinator and q is the name (identifier) of the faulty coordinator.
- ***elected***(#(p), q) where #(p) is the ID number of newly elected coordinator named p and q is the name of the hitherto faulty coordinator.

The ring algorithm is outlined as follows:

1. A process which has required a service and has detected a failure of the coordinator, assumes the „***participates***" status and sends the message „***election***" to its successor on the ring, along with its own ID number and identifier of the damaged coordinator.
2. A process, which has received the message "***election***", compares its ID number to ID number in the received message and if:
   (a) the received ID number is greater than its own, then passes this message to its successor on the ring without change and assumes the "***participates***" status unless already it has this status;
   (b) the received ID number is smaller than its own, and if the receiver:
   - has the status "***not_participates***", then sends to its successor the message "***election***" with its own ID number and identifier of the faulty coordinator and assumes the "***participates***" status;
   - has the status "***participates***", then replaces this received smaller ID number with the own one and sends to its successor the message "***election***" with its own ID number and identifier of the faulty coordinator;

(c) the received ID number is equal to its own, then this means it is the greatest, so, the receiver has been elected as a new coordinator; if this receiver has the status "***participates***" then it assumes the "***newly_elected***" status and sends to its successor the message "***elected***" with its own ID number and identifier of the faulty coordinator; otherwise, end of election.

3. A process which received the message "***elected***", proceeds as follows: if it has not been elected as the new coordinator, then passes this message to its successor on the ring and assumes the "***not_participates***" status; otherwise, end of election.

Notice that this version of the ring algorithm allows carry out the elections concurrently: two or more processes may detect failure of the same coordinator at moments close to each other and proclaim election.

Readers who prefer to study algorithms in a flow diagram form, may take a look at Fig. 7.11 Its exemplary work, as a sequence of states, shows Table 7.2.

The Table 7.2 presents a system of processes *p1, p2, p3, p4, p5, p6* in the course of several elections proclaimed after a failure of current coordinators. The ID numbers assigned to processes, are ordered as follows:

$$\#(p5) < \#(p3) < \#(p1) < \#(p6) < \#(p2) < \#(p4)$$

The processes assume the status during elections shown in Fig. 7.12, Structure of the ring is shown Fig. 7.13.

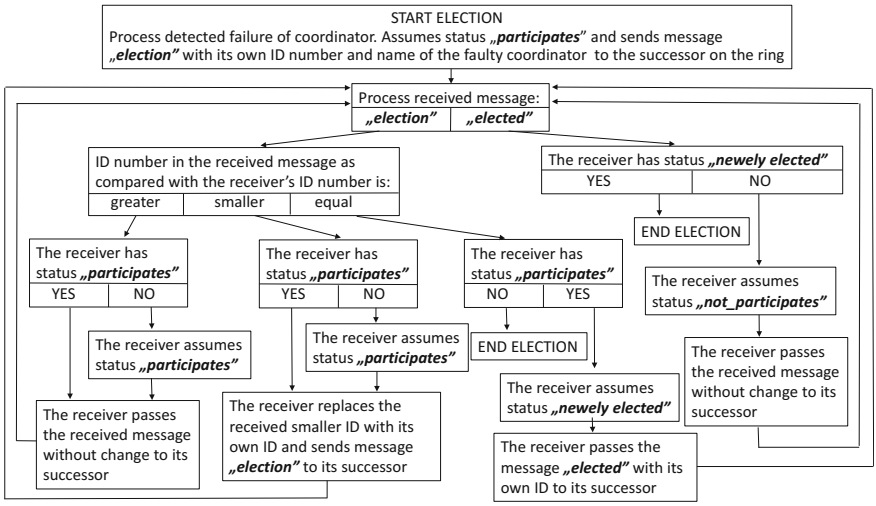


Fig. 7.11 Flow diagram of the ring algorithm of a coordinator election

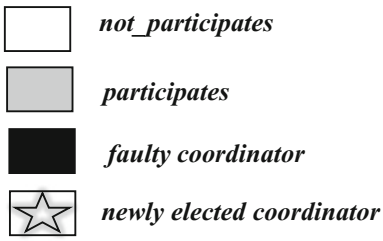


Fig. 7.12 Status assumed by processes during election

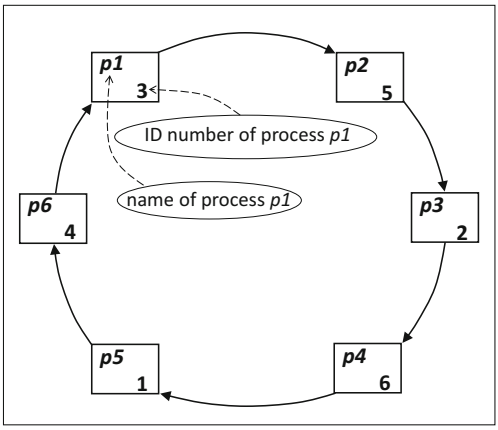
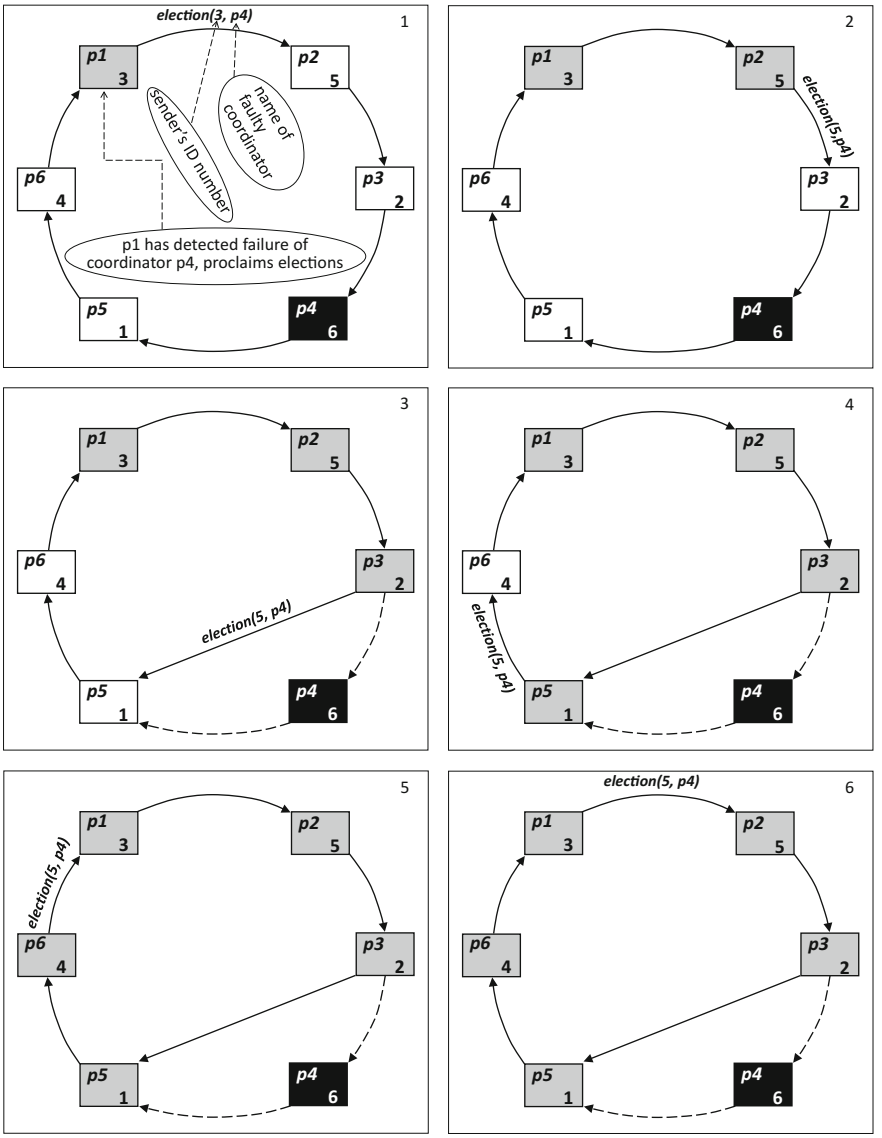


Fig. 7.13 Structure of the ring

Table 7.2 Election of new coordinator by the ring algorithm

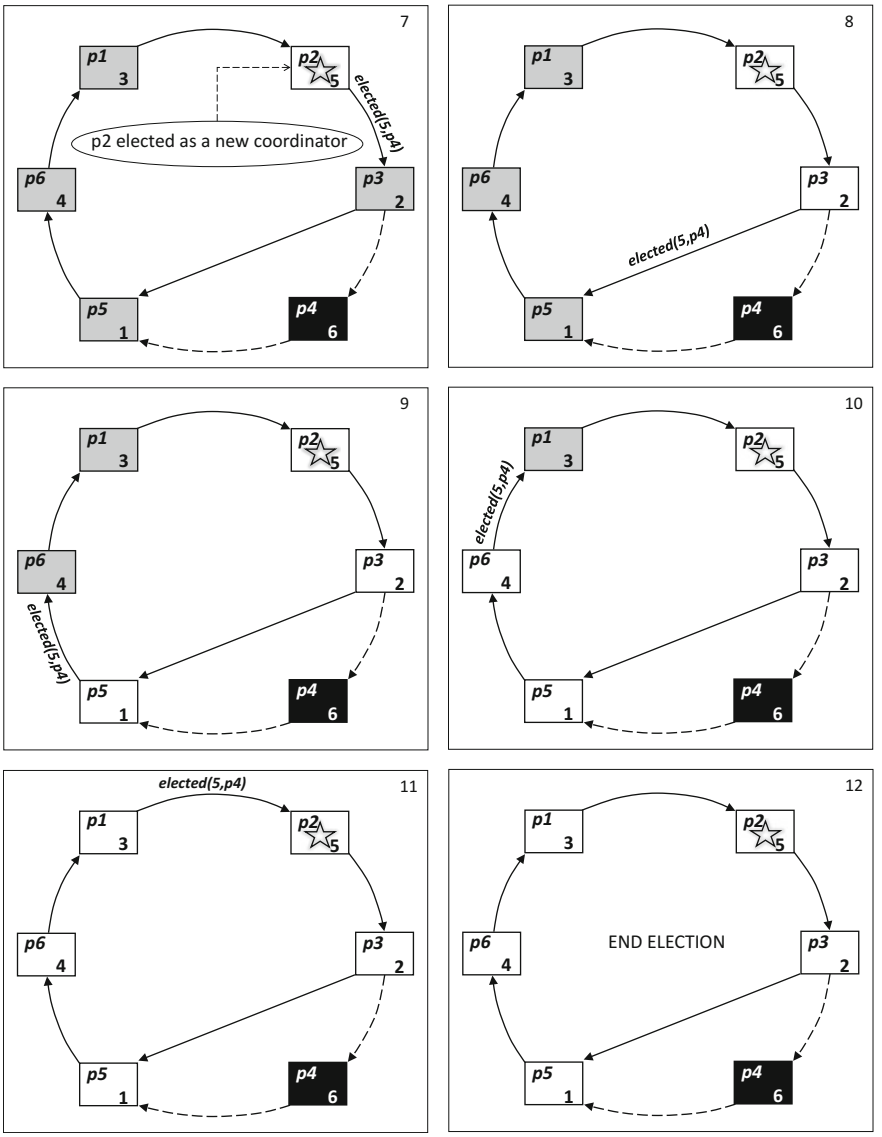
States 1–6



(continued)

Table 7.2 (continued)

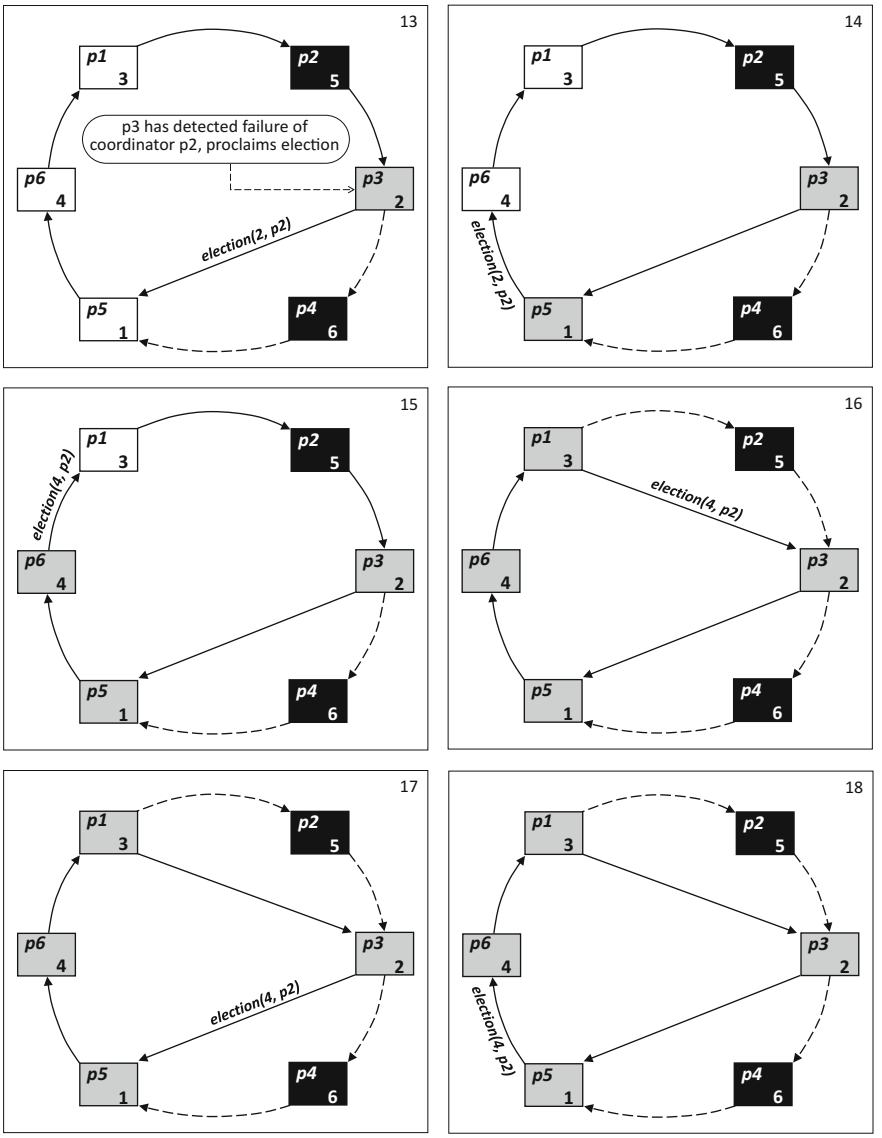
States 7–12



(continued)

Table 7.2 (continued)

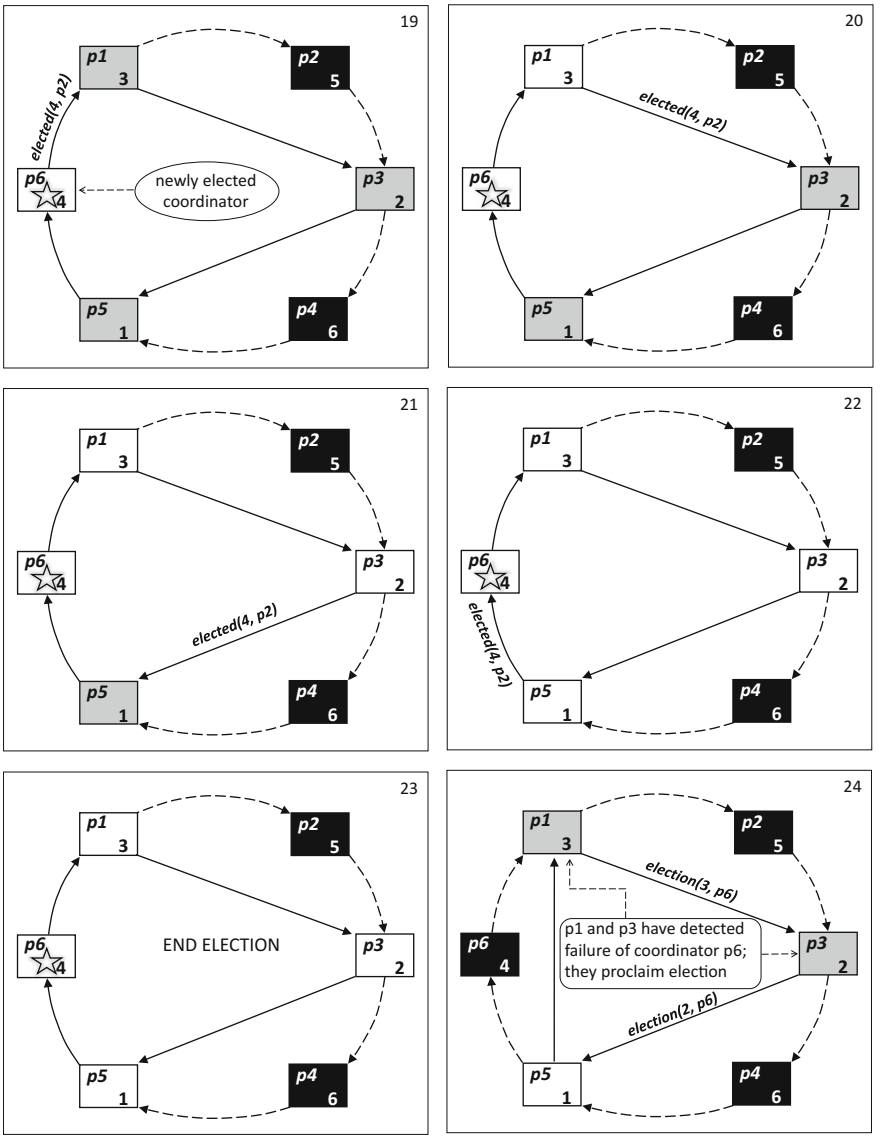
States 13–18



(continued)

Table 7.2 (continued)

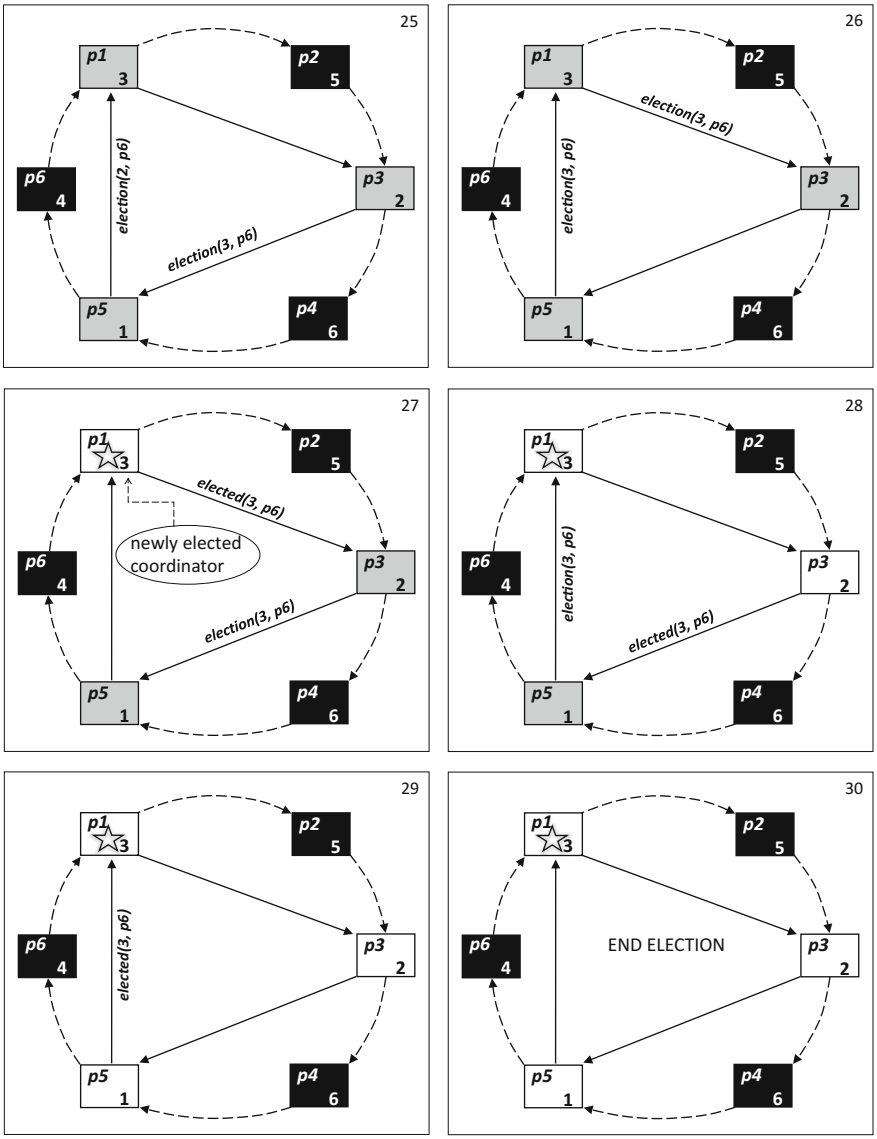
States 19–24



(continued)

Table 7.2 (continued)

States 25–30



References

Akkoyunlu, E. A., Ekanadham, K., Huber, R. V. (1975). Some constraints and trade-offs in the design of network communications. In *Proceedings of 5th ACM Symposium on Operating Systems Principles* (pp. 67–74).

Balzer, R. (1967). An 8-state minimal time solution to the firing squad synchronization problem. *Information and Control, 10*(1), 22–42.

Ben-Ari, M. (1990). *Principles of concurrent and distributed programming*. New Jersey: Prentice-Hall.

Cachin, Ch., Guerraoui, R., Rodrigues, L. (2006). *Introduction to reliable and secure distributed programming* (2nd Ed). Berlin: Springer.

Coulouris, G., Dollimore, J., & Kindberg, T. (1994). *Distributed systems, concepts and design*. Boston: Addison Wesley Longman Limited.

Fisher, M., Lynch, N., & Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM, 32*(4), 374–382.

Garg, V. K., Bridgman, J. (2011). The weighted byzantine agreement problem. In *IEEE International Parallel & Distributed Processing Symposium*, Ancorage, Alaska (pp. 524–531), May 16–20, 2011.

Lamport, L., Shostak, R., & Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems, 4*(3), 382–401.

Moore, F. R., & Langdon, G. G. (1968). A generalized firing squad problem. *Information and Control, 12*(3), 212–220.

Randel, B., et al. (1978). Reliability issues in computing system design. *Computing Surveys*, *10*(2), 123–165.

Tanenbaum, A. S. (1995). *Distributed operating systems*. New Jersey: Prentice-Hall International, Inc.

Waksman, A. (1966). An optimum solution to the firing squad synchronization problem. *Information and Control, 9*(1), 66–78.

Chapter 8
Distributed Shared Memory

8.1 Structure, Motivations, Problems, Advantages, Disadvantages

In accordance with general objectives of distributed systems, Distributed Shared Memory (DSM) aims at making possible usage of local memory of all computers by the programmer, as if constituted jointly a single private local memory in his/her computer. This is to unburden the programmer from details of transmissions of data between his/her own computer and other computers in the system. To achieve this, a mechanism which simulates one huge address space, accessible to all computers has been devised. Such virtual memory is, from a user viewpoint, a union of all local memories of computers. Its construction yields problems much more complicated than construction of virtual memory of a single computer (the simulated extension of RAM by a disc), where it is implemented by means of paging or segmentation. Figure 8.1 depicts a general structure of the distributed system endowed with the DSM mechanism and Fig. 8.2 a view, how such system may be perceived by the user.

DSM is a mechanism for integration of computers in distributed system, similarly as mechanism of Remote Procedure Call (RPC), but on lower level: for the user, access to memory of external computer, does not differ from access to memory of his/her own computer. The main problem is to ensure the system's behaviour in accordance with desired model of *memory consistency*. This means that results of activity of every computer are to be functionally close to the results of its activity in the environment of remaining computers with a common memory for all, unique (with no replicas) location of data and undisturbed access to the data by other computers. A degree of this closeness depends on a model of memory consistency. In other words, the system should behave as if it worked in accordance with its specification, but on a single computer with a huge memory and in the time sharing mode for programs executing the specified common task. That is, as a multiprogrammed centralized computer. Lack of the so-called strict consistency, is a

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_8

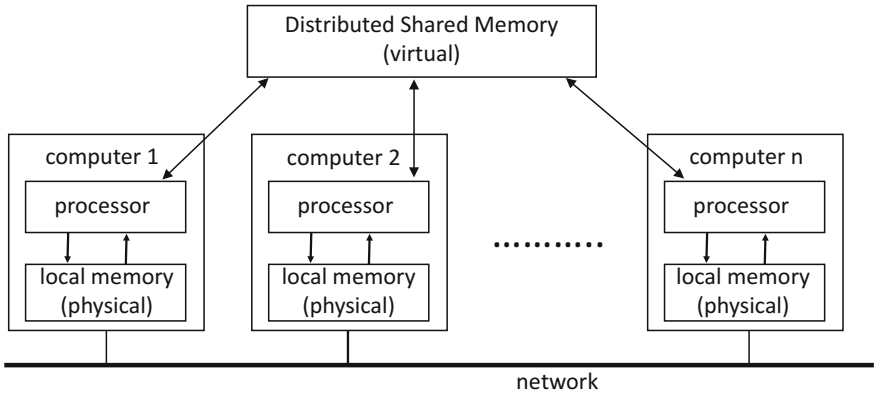


Fig. 8.1 DSM system—general structure

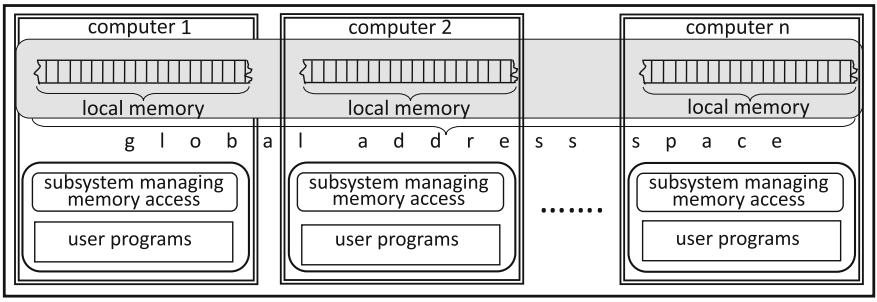


Fig. 8.2 DSM system—user's view (a union of local memories)

consequence of performing access to memory by means of subsystems managing the access, that create and handle multiple copies of data and make their relocation and transmission through the network. This takes considerably longer time than executing a machine instruction. The memory manager may be a fragment of operating system on every computer, or its task may be performed by an interpreter („virtual machine" or runtime system) of a programming language, or by library procedures. The access to DSM, i.e. the reading and writing of data, may also be done by one supervising manager, which receives and performs requests from client-processes and ensures desired memory consistency model. Remember that by "distributed system" is meant here not each possible computer network, where realization of DSM would be not only inexpedient but impossible, for instance for all computers in the internet. All this depends on the DSM system architecture. The DSM mechanism is implemented for a system of autonomous computers connected by a network, without common physical memory, managed by a distributed operating system and, in general, is designed for a certain class of tasks. So, e.g. in some corporate or local networks (see 2.1 in Chap. 2). Application of this

mechanism requires new solution to some problems typical for multicomputer systems without common memory. In particular, devising specific algorithms (protocols) for synchronization of processes and clocks, realization of critical sections (called sometimes protective zones) when using shared resources, Remote Procedure Call, exception handling, etc.

In its early versions (Li and Hudak 1989), the DSM was a virtual memory, implemented analogously to the paging system, where pages had their locations in the memory of individual computers (work stations) connected by local networks (LAN), in contrast to systems comprising only a single computer with disk. If a program in a certain computer attempted to access data located on a page absent in memory of this computer, then the paging manager was making transfer of this page from memory of another computer, where the data is located. As always, the problem of page replacement arises, i.e. choosing a page (or segment) that is to be removed from RAM, if there is not enough space for a page actually needed (Czaja 1968; Madnick and Donovan 1974; Silberschatz et al. 2009; Nutt 2002; Tanenbaum 1995). However in case of distributed systems, more transmissions of pages occurs, than between one computer and disk, since a fixed store of pages does not exist in DSM. It increases a chance of the so-called thrashing or flickering—of pages, that is, waste of time for unnecessary transmissions. In the further development of the DSM techniques, the enhancement of efficiency has been obtained, e.g. by multiplying number of locations (data replication) of shared variables, or by creating of virtual memory only for variables (objects) used in processes performing a certain task. The problems of memory inconsistency then arise, when a certain variable has a location in diverse local (and cache) memory storages, or when the variable has been declared as a *shared*, in programs executed by diverse computers. A review of some memory consistency models may be found in Mosberger (1993). Several versions of DSM have been devised and implemented, e.g. with a singled out DSM manager, or with a distributed manager, that is a collection of fragments of operating systems residing on individual computers.

Advantages

- distributed system with the DSM mechanism is scalable: computers may be attached and removed, thus arbitrary large address space may be obtained;
- the user-programmer is released from communication management, that is from using instructions ***send*** and ***receive***, when writing or reading data located outside of his/her computer, as well as from data marshalling (especially of complex data structures) and from entering into details such as control of communication reliability, familiarity with protocols, etc.;
- simpler interfaces (implemented in software), cheaper than those in multiprocessor hardware;
- the total cost of distributed memory of all interconnected computers is lower than the cost of a specially constructed large global memory of multiprocessor systems, because of massively produced integrated circuits as well as publicly available protocols of memory access are being used;

- no memory buses between the processors and global memory with a complex structure („topology") of their interconnection, hence no bottlenecks in access to memory;
- programs in DSM systems are portable, because of common programming interfaces;
- the DSM system may handle big databases without burden the user with details of this service, e.g. making copies of the databases—the system itself creates each copy.

Disadvantages

- users have no control over traffic of data between computers; this may slow down this traffic in comparison with usage of explicit message passing by means of ***send*** and ***receive*** operations;
- users have less control over reliability of data transmission;
- users must understand (at least some) memory consistency models; the inconsistency arises when replicas of a certain variable, shared by programs in diverse computers, contain different values at the same time; this results from data replication and their transmission, which takes undetermined and unpredictable time, as well as from various frequency of local clocks, thus different speed of program execution by the computers.

8.2 Interleaving Model of System Activity

In Chap. 1, a number of programs running on one computer in a time sharing (i.e. multiprogrammed) mode has been shown (Table 1.2). Some fragments of processes were interleaved, simulating concurrent execution of the programs by separate physical processors. We say that a *true concurrency* (non-interleaving) has been simulated by *indeterminism*: order of the interleaved fragments is incidental—has nothing to do with what the programs are doing! If the programs constitute one system for solving a task, then such interleaving model makes easier a formal analysis of its behaviour, albeit it loses the important property of concurrency: independence of some actions and dependence on some others.

Remark The true concurrency have been presented and investigated in several formal models, like Petri nets (Petri 1966; Reisig 1985), Mazurkiewicz traces (Mazurkiewicz 1987; Diekert and Rozenberg 1995), Dependence graphs (Hoogeboom and Rozenberg 1995), cause-effect structures (Czaja 2002) and in a number of periodicals.

In order to present the DSM peculiarities, first let us consider a system of two computers executing programs (Fig. 8.3).

Fig. 8.3 There are diverse interleavings of statements of these programs

computer 1 performs:	computer 2 performs:
$y := b$; $x := a$; *print*(x,y);	$a := a+1$; $b := b+1$;

For the purpose of this section let us assume:

(1) the statements are atomic (indivisible), so they are the critical sections along with reading and writing memory;
(2) the statements are executed in the order of their appearance in every program;
(3) every execution of the statements in the system is serialized: one execution can be described as an interleaving of execution sequences of both programs;
(4) variables a, b are shared by both programs; initial values: $a = 0$, $b = 0$, x, y—arbitrary;
(5) *print*(x, y) is executed when execution of other statements in both programs is over; its execution causes fetching values of x, y from memory and their printout;

Under these assumptions there are possible only the following interleavings:

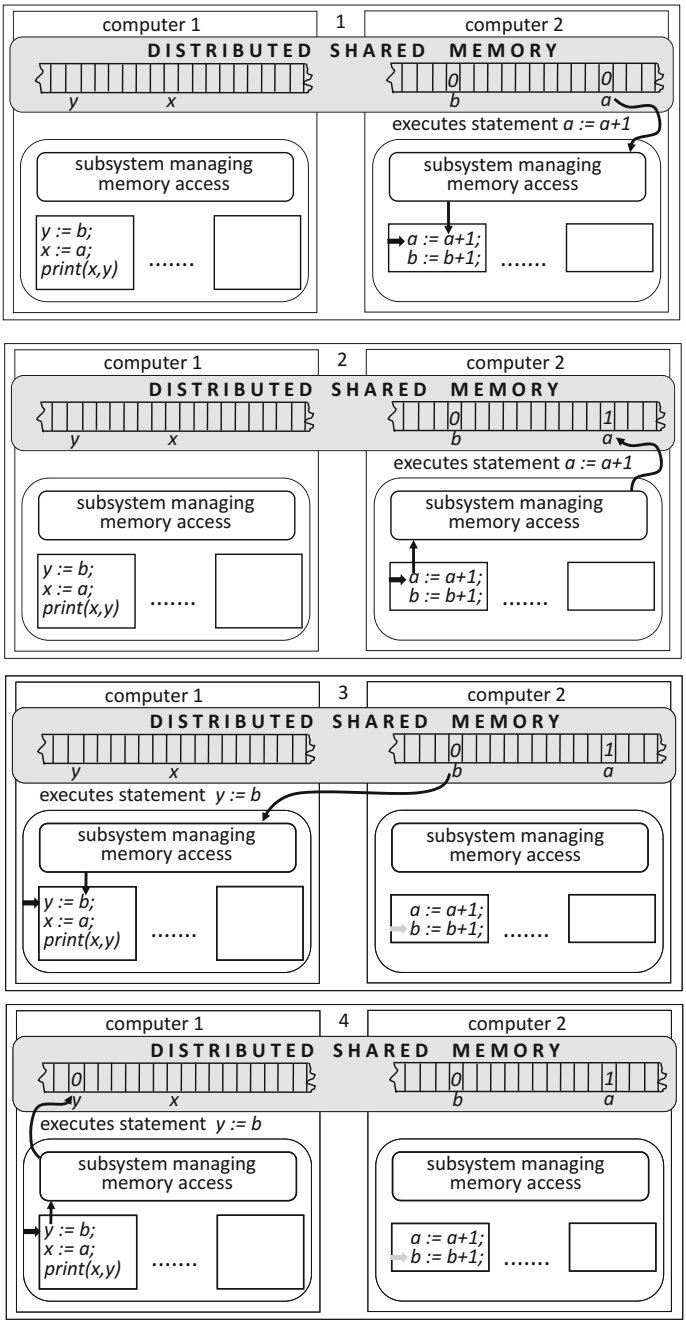
interleaving 1: $a := a + 1$; $b := b + 1$; $y := b$; $x := a$; *print*(x, y)
interleaving 2: $a := a + 1$; $y := b$; $b := b + 1$; $x := a$; *print*(x, y)
interleaving 3: $y := b$; $a := a + 1$; $b := b + 1$; $x := a$; *print*(x, y)
interleaving 4: $a := a + 1$; $y := b$; $x := a$; $b := b + 1$; *print*(x, y)
interleaving 5: $y := b$; $a := a + 1$; $x := a$; $b := b + 1$; *print*(x, y)
interleaving 6: $y := b$; $x := a$; $a := a + 1$; $b := b + 1$; *print*(x, y)

Remarks

1. Without assumptions 2 and 5 there are 5! = 120 permutations of statements of these processes.
2. Different interleavings yield different printouts of values of x, y, but the inequality $x \geq y$ (invariant of the system) is fulfilled for every of the interleaving; Indeed, if $x < y$ then the printout would be $x = 0$, $y = 1$, thus statement $x := a$ should be executed before $a := a + 1$ and statement $y := b$—after $b := b + 1$, which is absent in interleavings 1–6, since it would contradict execution order of statements in the programs.
3. It may be proved that the number of interleavings of n sequences, each of k_1, k_2, ..., k_n elements, amounts to $\frac{(k_1+k_2+\ \ldots\ +k_n)!}{k_1!k_2!\ \ \ldots\ k_n!}$. Thus, there are $\frac{(2+2)!}{2!2!} = 6$ interleavings of two sequences $y := b$; $x := a$ and $a := a + 1$; $b := b + 1$ of statements in above programs.

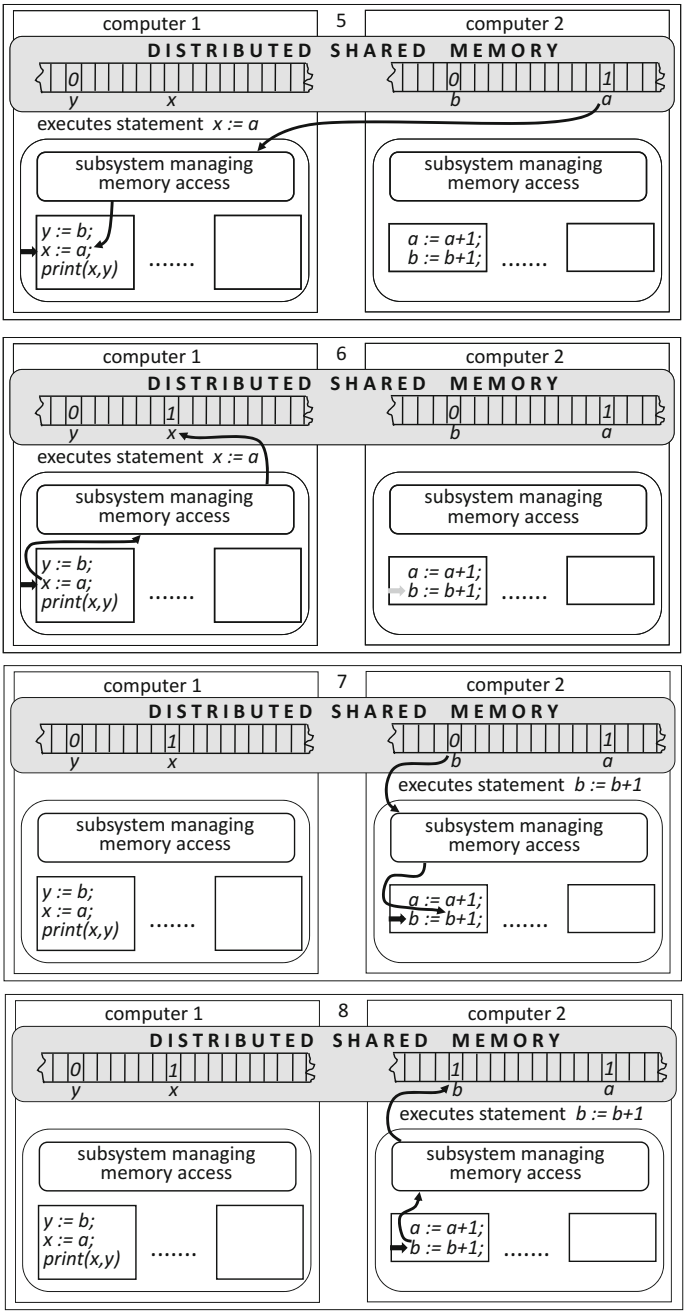
Table 8.1 shows execution of the system depicted in Fig. 8.3 in the interleaving 4. Variables x, y are located (addressed) in the memory of computer 1 and variables a, b in the memory of computer 2. No replication of variables takes place, instead memory manager of computer 1 fetches values of a, b from memory of computer 2.

Table 8.1 State 1—fetching value of a from memory to program, state 2—storing value of expression $a + 1$ in variable a, state 3—transmission of value of b to computer 1, state 4—storing it in variable y. state 5—transmission of value of a to computer 1, state 6—storing it in variable x, state 7—fetching value of b from memory to program, state 8—storing value of expression $b + 1$ in variable b. 1 states 9, 10—fetching values of x and y to printing



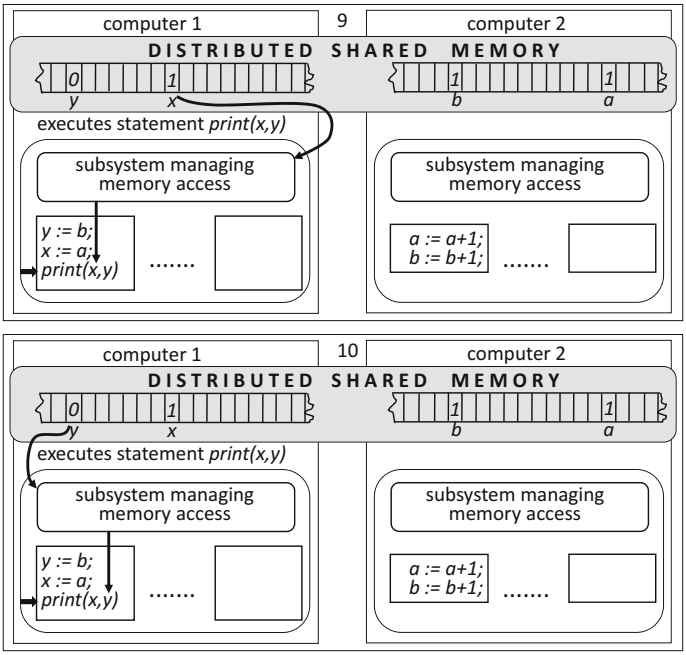
(continued)

Table 8.1 (continued)



(continued)

Table 8.1 (continued)



Notice however, that this is practically untypical realization, since in most (all?) known implementations pages or segment containing variables a, b are being transmitted to computer 1 where these variables become local. A part of paging mechanism is neglected, so that not to overshadow the main issue of this section: simulation of concurrent execution by possible interleavings. In the Table 8.1 the small horizontal arrow points to currently executed statement, the arrow connecting memory with the subsystem managing memory access shows reading or writing data from/to memory or data transmission from computer 2 to computer 1. After the execution, the printout $x = 1$, $y = 0$ takes place. The system is truly distributed, that is without an external manager of the DSM.

Notice that if variables a, b were located in both computers (replication) then the result of computation would be the same, provided that the statements are indivisible, since every statement begins execution only if execution of the previous statement is over. In this case, the local update of a value in memory of computer 2 is accompanied by transmission of this value to computer 1. Notice also that a system with a central manager of DSM would behave in the same way, due to assumptions (1–5). It is seen how the arrows showing data transfer in the Table 8.1 would go over in the case of other interleavings and what printouts would have taken place: interleaving 1: $x = 1$, $y = 1$, interleavings 2–5: $x = 1$, $y = 0$, interleaving 6: $x = 0$, $y = 0$.

In every interleaving 1–6 the following conditions are fulfilled:

(i) In each execution, the readout of a variable from memory fetches value assigned to this variable **before its most latter (prior to this readout) update (write in)**; thus, between these two access operations no other update of this variable takes place; here, the prepositions like „before", "prior to", „between" refer to the global (external) time.

(ii) The order of execution of statements in each interleaving is the same as order of their execution in respective programs; the readout of a variable fetches value assigned to this variable by a write operation, **preceding this readout in this interleaving** (with no other updates of this variable inbetween); so, here the „preceding" and the „inbetween" refer to a position in the interleaving, that is, in the sequence.

Condition *(i)* characterizes the ***strict*** consistency of memory, condition *(ii)*—the ***sequential consistency*** (Lamport 1979). They will be formally defined in Sect. 8.5. The conditions are fulfilled in the previous example due to serializability of entire statements in processes, i.e. ensuring (by the subsystems managing memory access) their execution only in permitted interleavings—due to their indivisibility (atomicity). Resignation of the assumptions (1–3) above, may cause violation of conditions *(i)*, *(ii)* and lack of meaning of the notions „preceding" and „between" with respect to the global time, since a global clock is absent in distributed systems. The global time is a concept external for the system. The notions „preceding" and „between", etc. may refer either to the real (external) time or to a place in a process (i.e. in a sequence), or in the interleavings—this should be made clear when used.

We have seen that the concurrent system may behave variously even with the same initial data: it may yield different executions with different results, but a given execution generates unique results, i.e. memory contents and printouts. Later will be seen that a given concurrent execution may be equivalent (i.e. with identical results) to executions in many interleavings satisfying condition *(ii)*. The user is unaware which execution out of many possible, the system would choose. If he/she is interested in some properties of results (e.g. some invariants), but not necessarily exact values, then he/she selects (or "agrees on") a system that ensures these properties. Thus, the user agrees on a certain kind (model) of memory consistency, more or less "liberal"—if more, then such one that permits for more concurrency. In this sense, one can say metaphorically that the memory, thus in fact the user, "concludes a contract" with the system to apply a certain kind of memory consistency.

The next sections show some consequences of resignation from above assumptions (1–3): the operational units (statements, memory read/write, …) in a system will no more be atomic and not necessarily executed in the order of their occurrence in the programs and not always can be executed sequentially with the same result as a given concurrent execution. It will be seen that only operations of memory access (i.e. read/write) are taken into account in description of the so-called consistency models of DSM. The lack of their atomicity, faulty data replication (not

identical copies of the same data) and latency of their transmission between computers, is a cause of various types of inconsistency. The assurance of desired consistency model is the task of subsystems managing memory access. The problems of memory consistency will be discussed by various examples, then formal definitions of some consistency models will be given.

Notice that by decreasing granularity of operational units, the degree of concurrency is increased, thus performance (speed) of the system but also probability of incorrect behaviour: collision when using shared resources, low accuracy of clocks synchronizing, various kinds of memory inconsistency (accepted for some tasks), etc. Notice also, that a value of a variable may be a certain data structure or object like array, tree, etc. In examples, for simplicity, it is assumed to be a primitive unit like a number.

8.3 Concurrency of Access Operations to Distributed Shared Memory, Examples of Sequential and Strict Consistency, Informal Description

Consider possible behaviours of the system {computer 1, computer 2} in Fig. 8.3, but with the following assumptions:

1. Significant are only operations of memory access—computation of $a + 1$, $b + 1$ is not taken into account: only memory read/write operations decide on various kind of memory consistency. Fetching value of a variable, means a read-out of its value from the local memory or transmission from memory of another computer. Update of a variable, means write-in its value into memory of each computer, where the variable has a location. So, the memory access operations encompass inter-computer transmission of data. The subsystems managing memory access arrange the read/write operations in a partial order. Read-out a value a of variable x by computer $\mathrm{j} = 1, 2$ is denoted by $R(x, a)_\mathrm{j}$ and write-in—by $W(x, a)_\mathrm{j}$.
2. Subsystems managing memory access where suitable protocols are performed, can run concurrently with execution of program statements (e.g. by means of different cores).
3. The initiation of reading or writing value of a variable takes place directly following a request sent to the subsystem managing memory access. Termination is not synchronized with execution of statements—a consequence of point 2.

4. Every computer terminates the reading and writing a variable with the same value as at the initiation—the computer does not change this value during execution of read/write.
5. Variables a, b are shared by both programs; initially $a = 0$, $b = 0$, x, y—arbitrary.

In examples that follow, variables a, b are located in memory of both computers. Thus, it is assumed that the paging mechanism takes care of supplying a page containing the variable to the computer that uses this variable. And for present considerations it does not matter whether the needed variable will be located in the main or cache memory. Various locations of a variable are called its ***replicas***. It is convenient to present execution of the system as a bunch of parallel straight lines, each line showing execution of a sequential process on the time axis. For instance, the diagram in Fig. 8.4 shows one of possible runs of the system from Fig. 8.3, reduced to the memory read/write operations.

This execution (run) in action is presented by Table 8.2, where the double lines stretching out across the successive states from computer 2 to computer 1 show data transmission pending. Notice that the states reflect temporal relationships between operations, preserving property expressed in point 4 above, but not exact instants of begin and end of operations shown in Fig. 8.4.

The system's concurrent execution shown in Fig. 8.4 and in Table 8.2 allows to make several remarks on memory consistency issues as well as the geometrical presentation of some dynamic peculiarities of concurrent systems.

First, the sequence of R/W operations on one time axis, or corresponding line segments, is called a local ***history of access to memory*** during execution of a sequential process. The beginning "(" of a segment is the instant of initiation (invocation) of respective operation. The end ")" of an R-segment is an instant of termination of fetching a variable's value from local memory to processor. The end ")" of a W-segment is an instant of completion updating all replicas (copies) of a variable (we will see later, that the end of a W-segment may be understood differently: as an instant of acknowledgment of broadcast a value to all replicas of a variable).

Second, the R/W-line segments of all processes can be arranged into a sequence, such that executing the operations in the order as in this sequence, yields the same

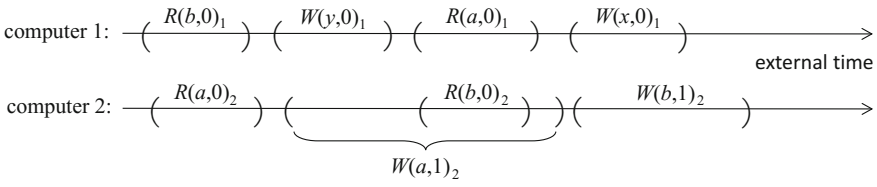
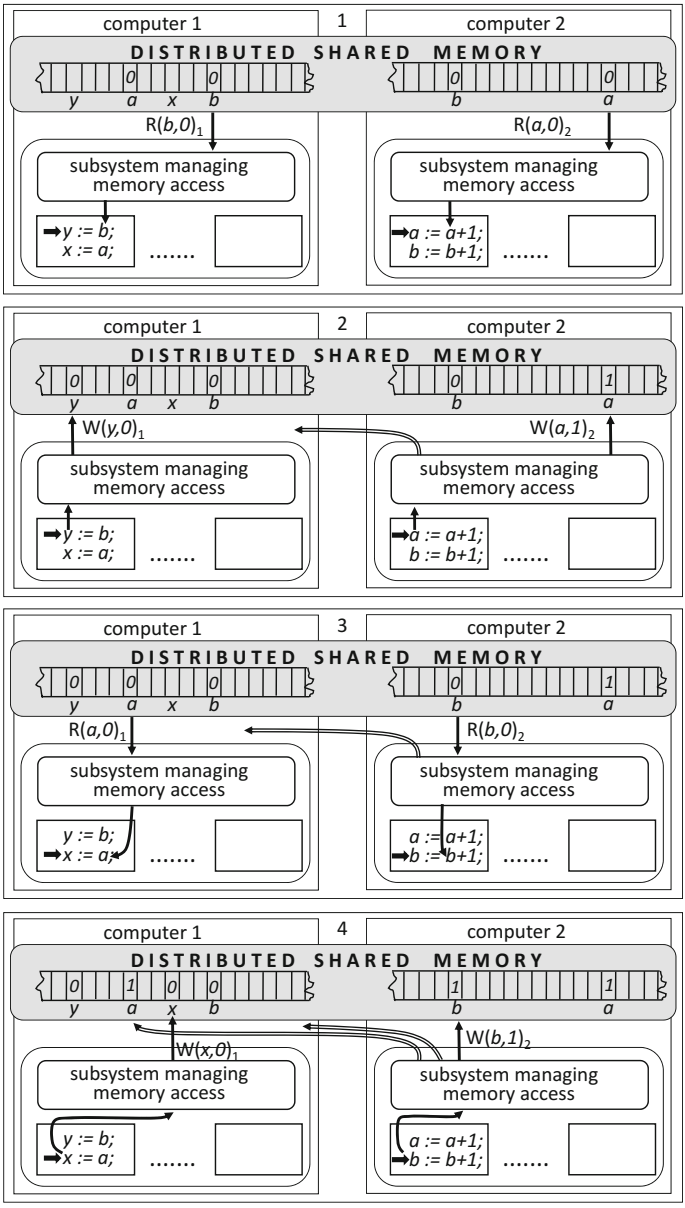


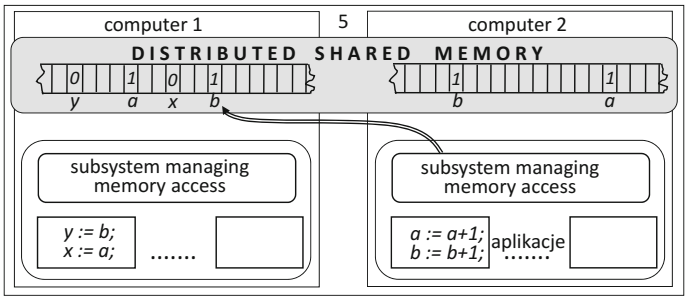
Fig. 8.4 Exemplary run of the system shown in Fig. 8.3, reduced to the R/W operations (save for operations relating to print-out). Allows for sequential consistency of memory

Table 8.2 Sequentially consistent execution. Operations $W(a, 1)_2$, $W(b, 1)_2$ initiate transmission of value 1 to computer 1



(continued)

Table 8.2 (continued)



result as the concurrent execution. For the execution shown in Fig. 8.4 one of such possible sequences is, for instance:

$$R(b,\,0)_1, R(a,\,0)_2, W(y,\,0)_1, R(a,\,0)_1, W(a,\,1)_2, R(b,\,0)_2, W(x,\,0)_1, W(b,\,1)_2$$

This is an admissible interleaving of local histories of memory access, making the **global history** of this execution. It is easily seen what are other admissible interleavings, that is such that yield the same result as the concurrent execution (by the way: all these equivalent—in this sense—interleavings, constitute one Mazurkiewicz trace (Mazurkiewicz 1987; Hoogeboom and Rozenberg 1995), a concept introduced on higher abstraction level than present description of memory consistency). The existence of admissible interleaving of all the histories of memory access characterizes the sequential consistency of the memory. In the seminal paper (Lamport 1979), Leslie Lamport, who had introduced this concept, put it as follows: "*the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by its program*". So, the sequential consistency is characterized by two kinds of preservation (invariance) of some properties: preservation of the same ordering of read/write execution as in the individual programs and preservation of memory coherence (Sect. 8.2, point *(i)*).

Third, to ensure sequential consistency, the subsystems managing memory access must sometimes delay some operations, so, in geometric terms, to make shift of a respective segments along the time axis. This happened in above example: the $W(a,\,1)_2$ operation and its successors have been delayed, so that $R(a,\,0)_1$ precedes the $W(a,\,1)_2$ in the global history, which ensures memory coherence.

Fourth, construction of the global history—if possible for a given concurrent execution, that is serialization of the R/W operations—requires the operations to be indivisible (atomic) in this history. Assurance of this belongs to the subsystems managing memory access. Sequential consistency of memory entails preserving the so-called Readers/Writers principle (essential e.g. for data bases): while a process is updating a variable (all its replicas), no other process can write to or read from this variable, but when it is reading a variable, then other processes can also read this

variable, but not write to it. The „Readers/Writers" problem (Courtois et al. 1971), has been analysed in various versions, in a number of publications. In fact, the sequential consistency of memory, as requiring serialization of R/W operations, is the stronger property than the Readers/Writers principle.

Fifth, the order between some operations in a global history may be inverted (yielding different but equivalent global history) as far as it does not influence operation ordering in individual programs and memory coherence. In particular, the R-operations in different local histories are independent of each other. In the global history they are ordered relative to the respective W-operations only.

Sixth, although different executions of a concurrent system with the same initial data may yield different results, thus not equivalent global histories, some relationships (desirable by the user) between the results should be invariant. An example is the inequality $x \geq y$ fulfilled after some executions of the system shown in Fig. 8.3, e.g. the execution depicted in Fig. 8.4. It may be easily verified that this inequality is ensured by every execution permitting the sequentially consistence of memory. Later, an execution of this system, violating this inequality will be shown.

Seventh, the sequential consistency of memory facilitates programming as well as makes easier reasoning on program correctness, since it simulates, in a sense, programming on a centralized multiprocessor system. However it impairs performance, because serializing the R/W operations lowers degree of concurrency. This is also a price of transparency—the important feature of distributed systems.

Eighth, The special case of the sequential consistency is the strict consistency. An example of execution of the system shown in Fig. 8.3, is given in the diagram in Fig. 8.5 and Table 8.3.

Here, every R-operation returns value of a variable updated by a W-operation execution **preceding—in the external time**—execution of this R-operation (or the initial value) with no update of this variable between these R and W operations. Thus, no delay of W-operations or shifting line segments along the time axis is needed, for the global serialization of the operations, which is always possible. One of possible serialization (a global history) may look as $R(b, 0)_1$, $R(a, 0)_2$, $W(y, 0)_1$, $W(a, 1)_2$, $R(a, 1)_1$, $R(b, 0)_2$, $W(x, 1)_1$, $W(b, 1)_2$. Notice that in case of sequential consistency, every R-operation returns value of a variable updated by a W-operation execution **preceding—in the global history**—execution of this R-operation (or the initial value) with no update of this variable between (in this interleaving) these R and W operations. So, as it has been already stated in Sect. 8.2, the prepositions "preceding", "between", etc. may refer to (external) time —in case of strict consistency or space (place in a sequence) in case of sequential

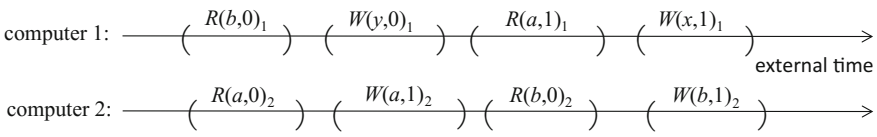
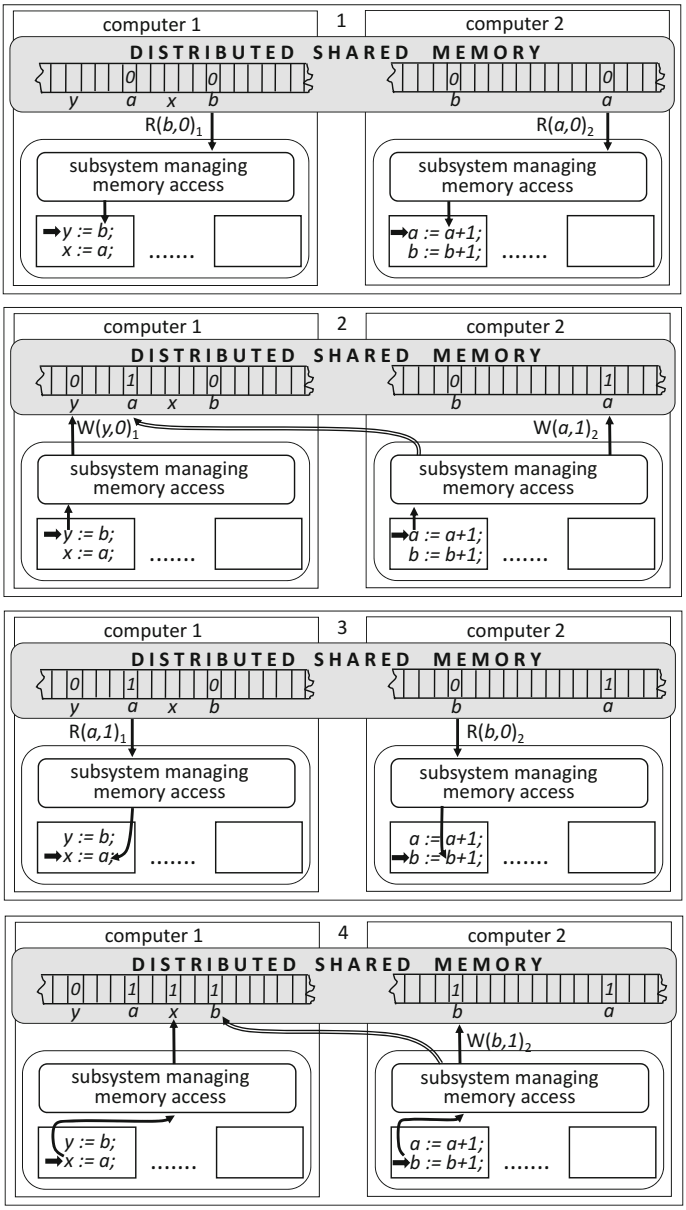


Fig. 8.5 This run of the system allows for strict consistency of memory

Table 8.3 Strictly consistent execution (unrealistic), thus also sequentially consistent



consistency. They are ambiguous without specification of a context in which they are used, since the distributed systems have no common clock. Notice that in the geometrical presentation, the strict consistency is possible if the line segments that represent the R/W operations, can be arranged on the global time axis in the order of their initiations. The effect of strict consistency is as if the update of all replicas of a variable were performed instantly—entirely unrealistic!

In the individual histories of above executions, the W-segments and R-segments are linearly ordered, so disjoint, but between executions in different histories, there is no fixed order—respective operations are concurrent. After the global (linear) arrangement of all these segments in the order consistent with their occurrence in the individual histories, every segment representing the readout of variable's value, must be preceded (in the global history) by a nearest segment representing write of this value to this variable—if such exists. Such a linear arrangement of all the segments, defines an admissible—for sequential consistency—interleaving of individual histories. In the next example (Fig. 8.6 and Table 8.4), such arrangement is not possible. The example demonstrates another execution of the system shown in Fig. 8.3. This execution violates sequential consistency of memory.

The sequential inconsistency results here from violating program order of complete execution of operations by computer 2: termination of $W(a, 1)_2$ follows termination of $W(b, 1)_2$. In the geometric setting—segments of both processes cannot be arranged into an order such that respective operations executed in this order would yield the same result as in Fig. 8.6. Before completion of $W(a, 1)_2$, operations $R(b, 0)_2$ and $W(b, 1)_2$ are being executed (a pipelining). It may happen due to violation of conditions (1–3) in Sect. 8.2—since subsystems managing memory run concurrently with application programs. In such case the system from Fig. 8.3 may behave as shows Fig. 8.6 and respective Table 8.4 where the double line grey arrows represent data transmission. In effect $x = 0$, $y = 1$ holds, what cannot be obtained by whichever execution of the system satisfying sequential consistency. This is so, since the $R(a, 0)_1$ would have to be executed **in external time before** $W(a, 1)_2$ and the $R(b, 1)_1$ **after** $W(b, 1)_2$—like in the sequence, $R(a, 0)_2$ $R(a, 0)_1$ $W(a, 1)_2$ $R(b, 0)_2$ $W(b, 1)_2$ $R(b, 1)_1$ $W(y, 1)_1$ $W(x, 0)_1$, which is not an interleaving of local histories, because $R(a, 0)_1$ cannot precede $R(b, 1)_1$.

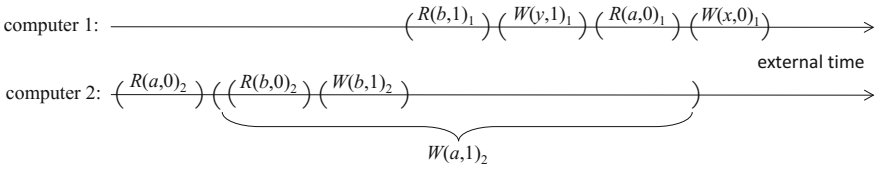
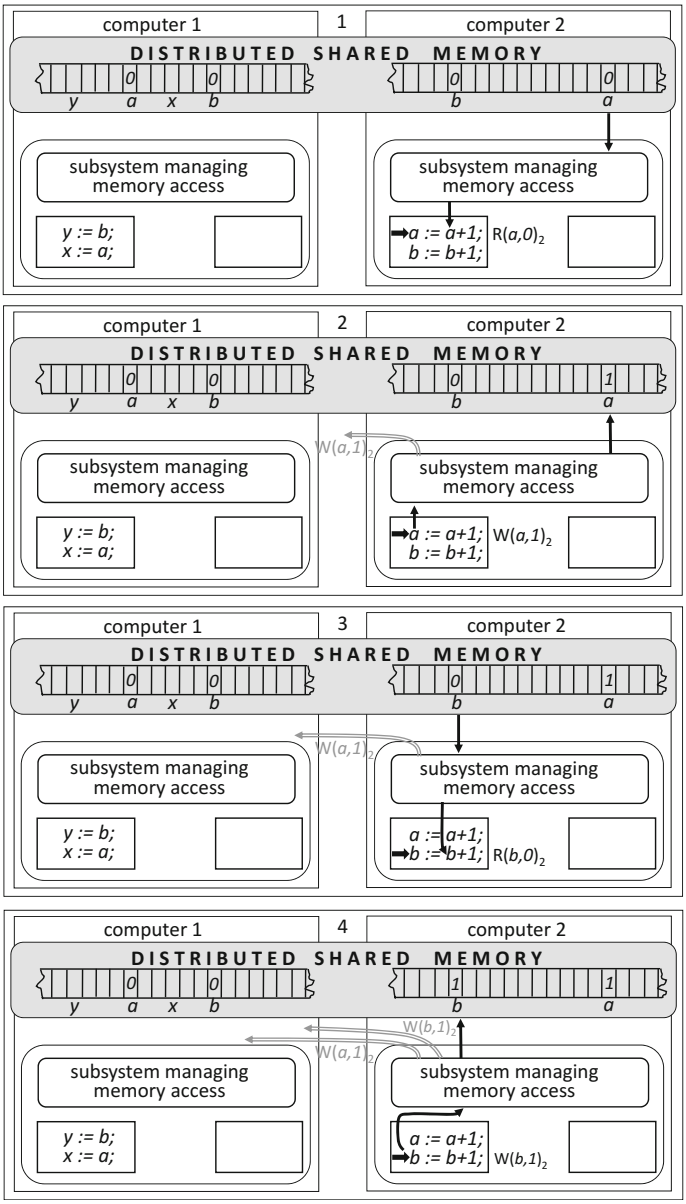


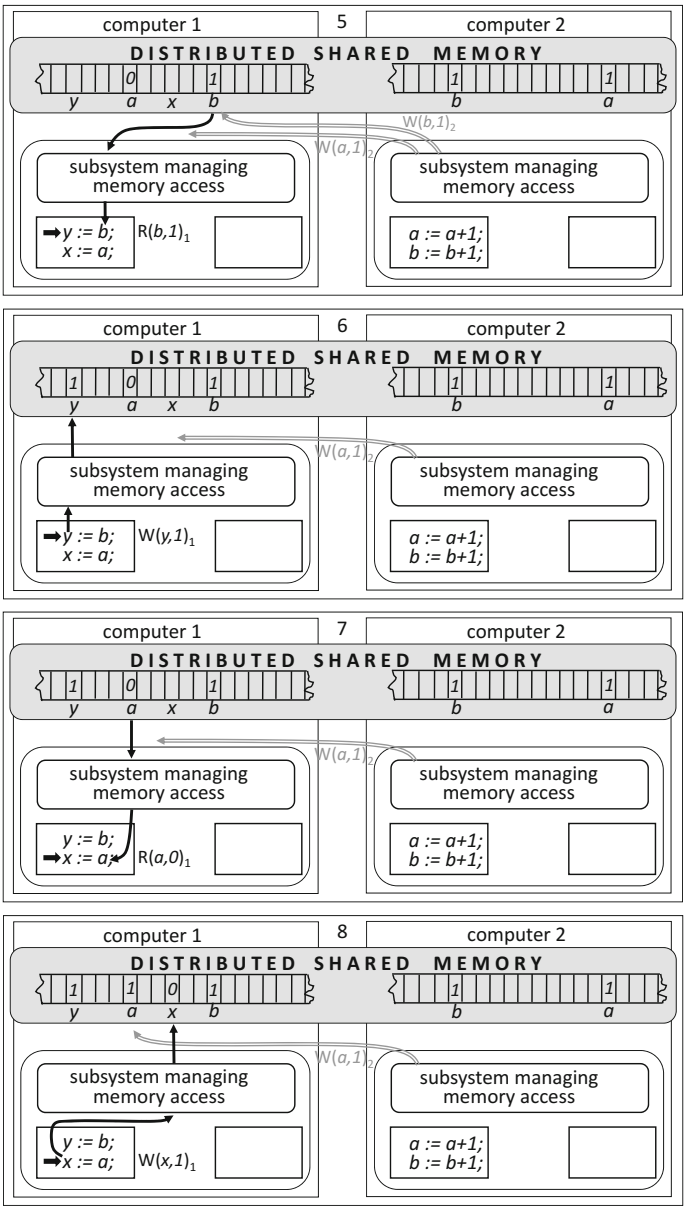
Fig. 8.6 Here, the complete update of variable a by computer 2 is delayed because of transmission duration. Does not allow for sequential consistency

Table 8.4 Computer 2 after computation of $a + 1$ commits the subsystem managing memory access to execute $W(a, 1)_2$ and passes to execution of $b := b + 1$. The subsystem will complete $W(b, 1)_2$ first, then $W(a, 1)_2$, changing ordering of these write operations in the program. Violation of the global FIFO order of message reception took place (Sect. 5.4 in Chap. 5). End of sequentially inconsistent execution of the system from Fig. 8.3, presented geometrically in Fig. 8.6



(continued)

Table 8.4 (continued)



Geometrically, obtainment of sequential consistency, if possible, consists in dislocation of R and W segments along the time axis, with preserving memory coherence so that following this transformation, the overlapping segments become disjoint.

A different geometrical presentation of system execution is obtained if the end of W-segment, is understood as acknowledgment of performing respective update by subsystems managing memory access—not as completion of updating all replicas of a variable. So, in the successive examples, the beginning of a segment, means a request (invocation) of respective operation, and the end means acknowledgment of realization—in case of the write, and return (fetch) of value—in case of the read. Between the invocation and the acknowledgment or fetch, the activity of a given process is suspended. Under such understanding, the segments in every history are disjoint. The invocation of write entails the broadcast of a variable's value to all its replicas and this value reaches them usually at different moments. Having ordered the broadcast, a processor continues activity, while the subsystem managing memory access acknowledges this order and is realizing it. For instance, when the end of the write segment is understood as acknowledgment of realization, then the diagram in Fig. 8.6 is replaced with that in Fig. 8.7, but concerns the same behaviour shown in Table 8.4. In this sense both geometric representations are equivalent. Differentiation of the two ways of understanding the end of W-segments changes only geometric representation of behaviour.

Two remarks on different geometric representations of memory usage during the same execution of a system are appropriate.

Remark 1 In the representation with invocation and acknowledgment of performing respective update as in Fig. 8.7, the segments could be reduced ("shrunk") to points labelled with respective operations, giving the same information on memory consistency/inconsistency of a given execution.

Remark 2 The double arrows pointing to instants of the write completion, provide excessive information in temporal diagrams presenting memory consistency/inconsistency of a given execution. For this aspect, unimportant is exact moment of data delivery, but only its temporal relationship to a moment of readout of this data. This is visualized in Fig. 8.7, by a position of value delivery by $W(a, 1)_2$ onto axis of computer 1, relative to position of $R(a, 0)_1$ on this axis.

In examples that follow, diagrams with invocation and acknowledgment will be used and also—for conspicuity—double grey arrows pointing to instants of the write completion.

Now, let us demonstrate behaviours of other (than shown in Fig. 8.3) systems, in the aspect of memory sequential consistency/inconsistency. First, let us consider extremely simple system shown in Fig. 8.8 and its execution in Fig. 8.9 with initial values of variables $a = 0$, $b = 0$; the readout of the variable's value, makes immediate sending it to print.

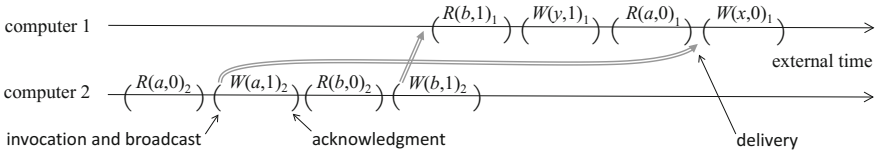


Fig. 8.7 Different meaning of the end of W-segments than in Fig. 8.6. No interleaving of the local histories exists without violation of operations order in processes or memory coherence. The double line grey arrows in Fig. 8.7 point to instants (in external time) of updating replicas of a and b in the memory of computer 1

computer 1 performs:	computer 2 performs:
$a := 1;$	$b := 1;$
print(b);	*print*(a);

Fig. 8.8 Extremely simple system

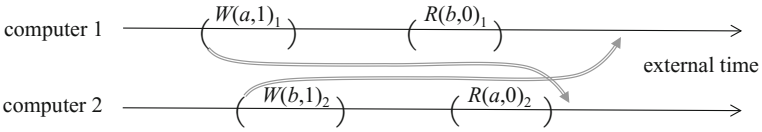


Fig. 8.9 Sequentially inconsistent execution: local histories cannot be interleaved preserving memory coherence

Impossible is global serialization of segments with preserving the order of execution in individual programs and memory coherence, with equivalent execution result, i.e. with $a = 1$, $b = 1$ in memory and printout $a = 0$, $b = 0$. Indeed, the segment $R(b, 0)_1$ in such sequence would have to precede $W(b, 1)_2$, and the segment $R(a, 0)_2$ would have to precede $W(a, 1)_1$ as, for instance, in a sequence: $R(b, 0)_1$ $W(b, 1)_2$ $R(a, 0)_2$ $W(a, 1)_1$ or $R(a, 0)_2$ $W(a, 1)_1$ $R(b, 0)_1$ $W(b, 1)_2$, that violate the order of execution in the local histories. Thus, sequential consistency is impossible. This diagram accounts for execution shown as a sequence of states in the Table 8.5.

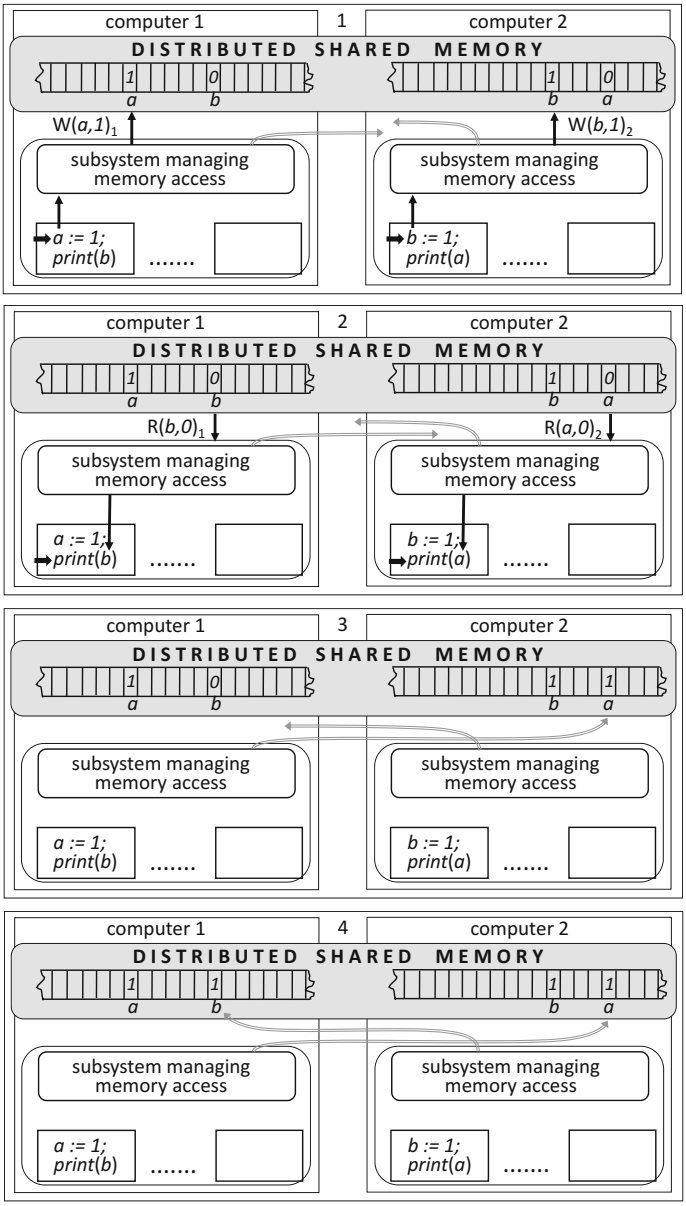
Let us modify the previous execution, replacing $R(b, 0)_1$ with $R(b, 1)_1$ with the same assumptions. So, the execution diagram is as in Fig. 8.10.

The sequential consistency could be ensured: a permissible interleaving exists:

$$W(b, 1)_2 R(a, 0)_2 W(a, 1)_1 R(b, 1)_1$$

This sequence preserves the order of operations in local histories and memory coherence. The result is the same as in the concurrent execution illustrated by the diagram in Fig. 8.10, that is, $a = 1$, $b = 1$ in memory and $a = 0$, $b = 1$ on the printout. The subsystems managing memory access have set all operations

Table 8.5 Sequentially inconsistent execution of the system from Fig. 8.8, presented geometrically in Fig. 8.9



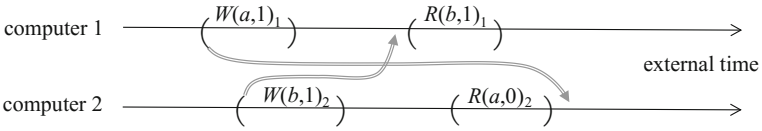


Fig. 8.10 Sequentially consistent execution: local histories can be interleaved preserving memory coherence, but shifting segments of operations is necessary

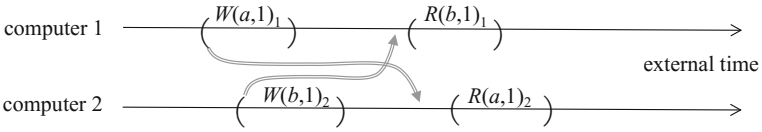


Fig. 8.11 Strictly consistent execution: local histories can be interleaved preserving memory coherence without shifting the segments of operations

sequentially, by delaying operations of computer 1, so that they be performed after operations of computer 2. Geometrically, the segments of computer 1 have been shifted rightwards along the external time axis. The local histories have been interleaved preserving memory coherence. The execution shown in Fig. 8.10 cannot ensure the strict consistency: setting all operations sequentially in the order of their execution in external time, thus $W(a, 1)_1\ W(b, 1)_2\ R(b, 1)_1\ R(a, 0)_2$, violates the memory coherence. After having written value 1 to variable a by computer 1, the computer 2 would return value 0.

Let us modify again the previous execution, replacing $R(a, 0)_2$ with $R(a, 1)_2$. So, the execution diagram is now as in Fig. 8.11.

The strict consistency (thus the sequential too) is possible: every readout of variables returns values previously (in external time) assigned to them and the R/W operations can be arranged into the sequence preserving their order in individual programs without their delay:

$$W(a,\ 1)_1\, W(b,\ 1)_2\, R(b,\ 1)_1\, R(a,\ 1)_2$$

The result is the same as in the concurrent execution illustrated by the diagram, that is, $a = 1$, $b = 1$ in memory and on the printout.

Remark In the Tables presenting execution of exemplary concurrent systems as sequences of states, two ways of access to memory have been applied: the *remote access* (Table 8.1) and *replication of variables*, that is multiplication of their location (Tables 8.2, 8.3, 8.4 and 8.5), perhaps by using cache memory. Another way called a relocation is also being applied: change of location of a variable in one computer into its location in another computer. For instance, when the latter computer uses this variable frequently. In presenting the principles, such details are neglected.

computer 1 performs:	computer 2 performs:	computer 3 performs:	computer 4 performs:
.....	.....	.....	.....
$x := \alpha;$	$x := \beta;$	*print*(x);	*print*(x);
.....	.....	.....	.....
		print(x);	*print*(x);
		.....	.....

Fig. 8.12 System with one shared variable x

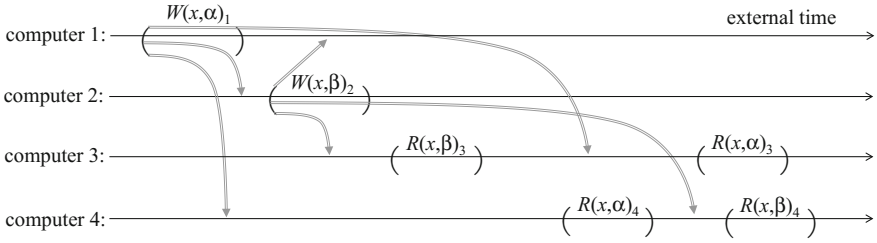


Fig. 8.13 Sequentially inconsistent execution: local histories cannot be interleaved preserving memory coherence

Sequential and strict consistency or their inexistence for some executions of a system, may appear also when access to one variable only takes place. The following examples illustrate such phenomena.

Let us consider possible behaviours of the system in Fig. 8.12 and its execution in Fig. 8.13.

Suppose that the constants α and β are distinct, the initial value of variable x is neither α nor β, and let readout of x make immediate sending its value to print. For instance, execution of $R(x, \beta)_3$ makes immediate printout of β by computer 3. As in the former examples, the beginning of a segment representing an access operation means request (invocation) of its execution, whereas the end represents acknowledgment of the write and return of the readout value. The segments in each history are, then, disjoint. Consider the execution diagram in Fig. 8.13 where, as formerly, the double line grey arrows point to possible instants of update of replicas of x in individual computers. This diagram accounts for execution shown as a sequence of states in the Table 8.6.

It is seen that here impossible is the global serialization of segments by shifting them along the time axis, so that the order of execution in individual programs and memory coherence be retained. The result of executing the system in such order should be the same as in Fig. 8.13, that is, the computer 3 should print βα, while computer 4 – αβ. Notice that the operations on memory are not indivisible (atomic): before complete updating replicas of variable x in all computers by a certain write operation, some other write operations are being executed changing

result of the former update. Therefore, without suspension of W-operations when only one of them is being executed, not all the variable's replicas must assume the same value: the memory incoherence occurs. The execution shown in Fig. 8.13 is presented in the Table 8.6 as a sequence of the system states.

Let us modify the diagram in Fig. 8.13, by replacing $R(x, \beta)_4$ with $R(x, \alpha)_4$, thus changing the instant of the second update of x in computer 4 e.g. as follows (Fig. 8.14).

The sequential consistency could be ensured: a permissible interleaving exists:

$$W(x, \beta)_2 R(x, \beta)_3 W(x, \alpha)_1 R(x, \alpha)_4 R(x, \alpha)_3 R(x, \alpha)_4$$

This interleaving ensures memory coherence, since the order of R/W operations is as in the programs and the same result is (memory content and print-outs) as shown on the diagram in Fig. 8.14. However, the execution as on the diagram, does not allow for the strict consistency, since arranging the segments in the order of their appearing on the time axis, i.e. $W(x, \alpha)_1$ $W(x, \beta)_2$ $R(x, \beta)_3$ $R(x, \alpha)_4$ $R(x, \alpha)_3$ $R(x, \alpha)_4$, violates the memory coherence: between $W(x, \alpha)_1$ and $R(x, \alpha)_4$, the $W(x, \beta)_2$ occurs.

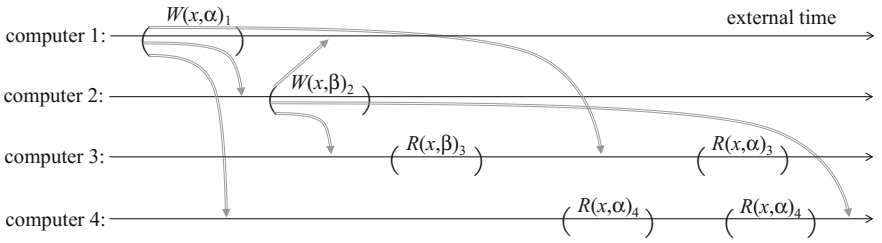
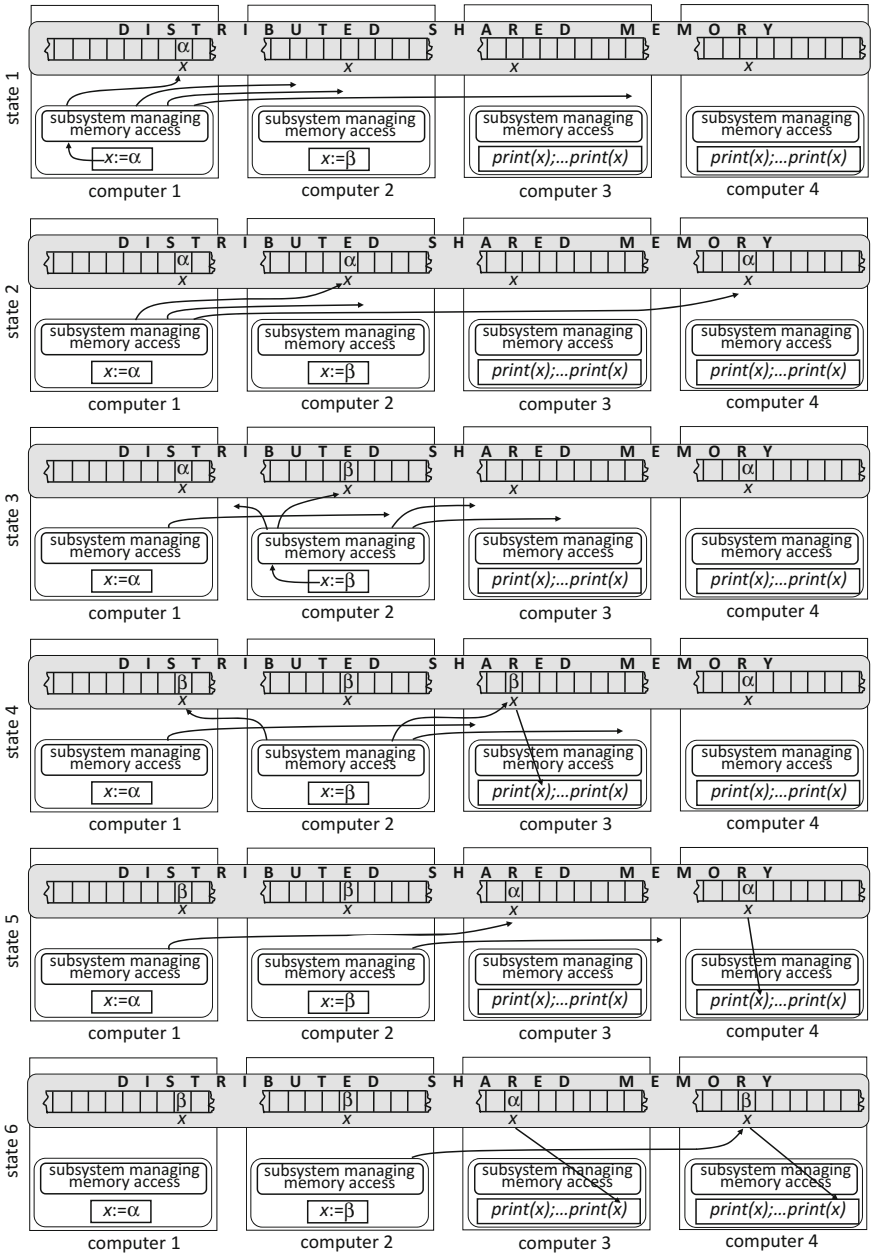


Fig. 8.14 Sequentially consistent execution: local histories can be interleaved preserving memory coherence, but shifting segments of operations is necessary

8.4 Events of Initiations and Terminations of Read/Write Operations

Every access to memory lasts a certain time, which is unknown when nonlocal memory is being accessed, that is when a remote data transmission is required. This indefiniteness results from impossibility of predicting the transmission duration, especially when many data replicas are being updated. Suspension of the data access at each their update, lowers the system efficiency, but fulfils user's expectations, that his/her programs behave as if executed by computer with local memory only. In particular, that the order of read/write memory operations is such as specified by his/her program. Not important for the user is, however, the order between execution of memory operations by different programs run concurrently with his/her program, if memory coherence is preserved. This expectation is

Table 8.6 Sequentially inconsistent execution of the system from Fig. 8.8, presented geometrically in Fig. 8.13



satisfied by the sequential consistency, the most natural feature for the programmer. It was seen in the previous examples that not for each concurrent execution, i.e. the partial order of memory operations, their linear ordering is possible without resignation from memory coherence and change of computation outcome (this is the task of protocols assuring sequential consistency to prevent execution of such kind). Therefore, the indefinite time duration of memory operations (their „coarse granulation") makes impossible exact description of arbitrary concurrent execution by means of interleaving of sequences of these operations, especially description of various kinds of memory consistency. The interleavings not always exist due to the write operations long-drawn-out in time—their overlapping, even in the same process (see Fig. 8.6). That is why the initiations and terminations of read/write operations will be treated as the *elementary events*, i.e. indivisible—atomic („timeless"). With such „fine granularity" of events concerning memory, all such events may be ordered into global sequences, therefore being arranged in the linear (total) order. Each execution is, then, represented by a sequence of events. In distinct sequences representing the same execution, the concurrent (independent) events may occur in reverse order – concurrency is modelled by nondeterminism. Such abstraction allows for describing consistency models without referring to terms like „process sees", so, without metaphoric expressions—intuitive comments to definitions of various types of consistency.

The following assumptions and denotations are admitted:

(1) The readout initiation of value α of variable x by computer j = 1, 2, …, N is denoted by $\overline{R(x, \alpha)_j}$ and termination—by $\underline{R(x, \alpha)_j}$ and similarly for write: $\overline{W(x, \alpha)_j}$ and $\underline{W(x, \alpha)_j}$ Events $\overline{R(x, \alpha)_j}$, $\overline{W(x, \alpha)_j}$ are invocations of the operations. Event $\underline{R(x, \alpha)_j}$ is understood as fetching (return) value α of variable x from memory, and $\underline{W(x, \alpha)_j}$—as end of writing value α, to all replicas of x. Any computer terminates readout or write-in a variable with the same value as has initiated, but another computer may change this value, between begin and end of this operation. For instance, it may happen that in a certain temporal sequence of events, the following fragment appears: $\overline{R(x, \beta)_k}\,\overline{R(x, \alpha)_j}\,\underline{W(x, \alpha)_j}\,\underline{R(x, \beta)_k}$ with $\beta \neq \alpha$, $k \neq j$. Memory coherence is violated: computer k reads different value of variable x than was completely updated by computer j, thus sequential consistency violation took place. The sequential inconsistency may also appear, when in a given program, before termination of an operation, another operation is starting. This is possible, since the subsystems managing memory access may run concurrently with program statements.

(2) Sequencing of initiations and terminations of *read* and *write* (their linearization) is accomplished by subsystems managing memory access. The initiation of an operation must precede its termination in each sequence. Such theoretical model clearly exhibits arrangement of access operations on the external (global) time axis.

As an example, consider the execution of system shown in Fig. 8.12, its execution in the former model shown in Fig. 8.13 and now in the model with events of initiations (invocations) and terminations of memory operations, i.e. where $\underline{W(x,\,\alpha)_1}$ $\underline{W(x,\,\beta)_2}$ denote respectively, complete updates of variable's x value in all of its replicas, what is depicted in Fig. 8.15.

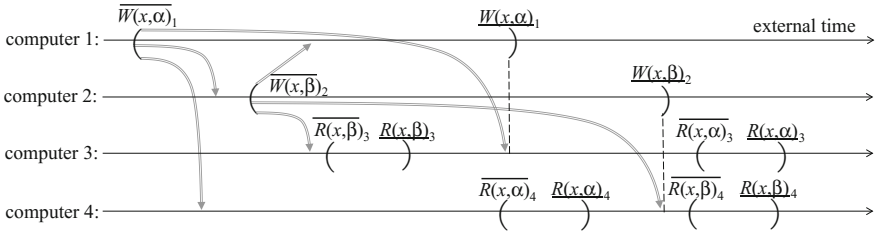


Fig. 8.15 The left and right brackets represent respectively, the instants of initiation and complete termination of the R/W operations in external time. Sequentially inconsistent execution, thus impermissible by the subsystem managing memory access

The events of read/write initiation and termination, have occurred in the following order (in accordance with the external time):

$$\overline{W(x,\alpha)_1}\ \overline{W(x,\beta)_2}\ \overline{R(x,\beta)_3}\ \underline{R(x,\beta)_3}\ \overline{R(x,\alpha)_4}\ \underline{W(x,\alpha)_1}\ \underline{R(x,\alpha)_4}\ \underline{W(x,\beta)_2}\ \overline{R(x,\beta)_4}\ \overline{R(x,\alpha)_3}\ \underline{R(x,\beta)_4}\ \underline{R(x,\alpha)_3}$$

A permutation of this sequence, such that every read/write termination appears next to respective initiation, yields the following sequence:

$$\underbrace{\overline{W(x,\,\alpha)_1}\underline{W(x,\,\alpha)_1}}_{W(x,\alpha)_1}\ \underbrace{\overline{W(x,\,\beta)_2}\underline{W(x,\,\beta)_2}}_{W(x,\beta)_2}\ \underbrace{\overline{R(x,\,\beta)_3}\underline{R(x,\,\beta)_3}}_{R(x,\beta)_3}\ \underbrace{\overline{R(x,\,\alpha)_4}\underline{R(x,\,\alpha)_4}}_{R(x,\alpha)_4}$$
$$\underbrace{\overline{R(x,\,\beta)_4}\underline{R(x,\,\beta)_4}}_{R(x,\beta)_4}\ \underbrace{\overline{R(x,\,\alpha)_3}\underline{R(x,\,\alpha)_3}}_{R(x,\alpha)_3}$$

It is seen that neither this permutation nor any other, satisfies the two requirements of sequential consistency, i.e. memory coherence and the order of events compatible with their order in individual programs. Sequential inconsistency of the execution shown on the diagram in Fig. 8.15 follows from **inexistence** of such permutation. But let us modify the diagram, by replacing events $\overline{R(x,\,\beta)_4}$ and $\underline{R(x,\,\beta)_4}$ with $\overline{R(x,\,\alpha)_4}$ and $\underline{R(x,\,\alpha)_4}$, thus changing the instant of second update of x in computer 4 as on the diagram in Fig. 8.16.

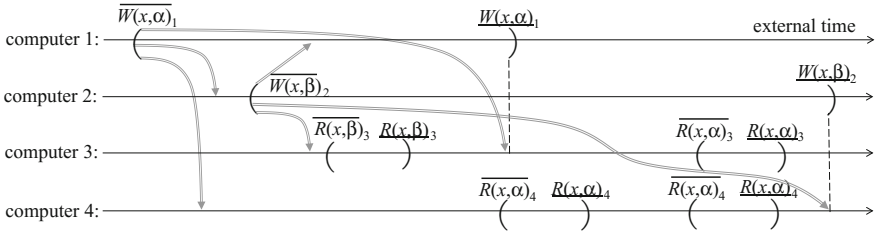


Fig. 8.16 Sequentially consistent execution

Now, the events of read/write initiation and termination, have occurred in the order:

$$\overline{W(x,\alpha)}_1\ \overline{W(x,\beta)}_2\ \overline{R(x,\beta)}_3\ \underline{R(x,\beta)}_3\ \overline{R(x,\alpha)}_4\ \underline{W(x,\alpha)}_1\ \underline{R(x,\alpha)}_4$$
$$\overline{R(x,\alpha)}_4\ \overline{R(x,\alpha)}_3\ \underline{R(x,\alpha)}_4\ \underline{R(x,\alpha)}_3\ \underline{W(x,\beta)}_2$$

A permutation of this sequence, such that every read/write termination appears next to respective initiation, yields the following sequence:

$$\underbrace{\overline{W(x,\beta)}_2\underline{W(x,\beta)}_2}_{W(x,\beta)_2}\ \underbrace{\overline{R(x,\beta)}_3\underline{R(x,\beta)}_3}_{R(x,\beta)_3}\ \underbrace{\overline{W(x,\alpha)}_1\underline{W(x,\alpha)}_1}_{W(x,\alpha)_1}\ \underbrace{\overline{R(x,\alpha)}_4\underline{R(x,\alpha)}_4}_{R(x,\alpha)_4}$$
$$\underbrace{\overline{R(x,\alpha)}_4\underline{R(x,\alpha)}_4}_{R(x,\alpha)_4}\ \underbrace{\overline{R(x,\alpha)}_3\underline{R(x,\alpha)}_3}_{R(x,\alpha)_3}$$

It is seen that this permutation satisfies the two requirements of sequential consistency, i.e. memory coherence and the ordering of events compatible with their order in individual programs. The sequential consistency of the execution shown on the diagram follows from **existence** of such permutation. Therefore, sequential consistency of this execution is equivalent to existence of such permutation. Later it will be shown in general, that existence of a suitable permutation of a sequence of the read/write initiations and terminations generated by a concurrent execution, is the necessary and sufficient condition for sequential consistency of this execution.

Remarks

1. In implementing protocols for sequential consistency, doing permutations as above, entails enforcement of R/W operations to be indivisible.
2. Taking events of initiations and terminations of R/W operations as elements of a certain alphabet, it is seen that their sequences (i.e. as executions of a system) are words in this alphabet, and behaviour (semantics) of the system is a formal language. In such approach, protocols ensuring

various models of memory consistency may be treated as generators of formal languages. The model of parallel systems as formal languages over this alphabet may be used for analysis of some semantic properties of systems. In particular, for decision problems of mutual exclusion, deadlock and starvation. In Czaja (1980, 2012) the problems have been reduced to decision problems of emptiness, and infiniteness of certain formal languages over invocations and terminations of actions.

8.5 Formal Definitions of Sequential and Strict Consistency

Representing of a real (concurrent) execution of a system as a sequence of points (not as segments) on the external time axis, i.e. as events $\overline{R(x, \alpha)_k}$, $\underline{R(x, \alpha)_k}$, $\overline{W(x, \alpha)_k}$, $\underline{W(x, \alpha)_k}$, makes possible selection from among all the sequences, only those that meet requirements of sequential and strict consistency, provided such exist. To this end, for a given sequence representing a concurrent execution, it should be decided whether or not there exists a permutation of this sequence satisfying conditions of a desired consistency model. If it does not exist then the execution is refused, but if it exists—entire (complete) R/W operations are executed in the order of their occurrence in the sequence suitably permuted.

Let us admit the following denotations:

(1) $\mathbf{S} = \{P_1, P_2, \ldots, P_N\}$—a system of sequential programs with DSM, performing in parallel by computers numbered 1, 2, …, N.

(2) **V**—set of variables used by the programs and allocated in DSM.

(3) **D**—set of values, the variables may assume.

(4) **E** is the set of events $\overline{W(x, \alpha)_k}$, $\underline{W(x, \alpha)_k}$, $\overline{R(x, \alpha)_k}$, $\underline{R(x, \alpha)_k}$ with $x \in \mathbf{V}$, $\alpha \in \mathbf{D}$, $k \in \{1, 2, \ldots, N\}$; this is the set of all events of initiations and complete terminations of R/W operations that may occur during activity of the system **S**.

(5) $Q = q_1 q_2 \ldots q_n \in \mathbf{E}^*$ is a sequence simulating parallel activity of the system **S**, if every event $\underline{W(x, \alpha)_k}$ and $\underline{R(x, \alpha)_k}$ occurring in Q is preceded, not necessarily directly, by respective $\overline{W(x, \alpha)_k}$ and $\overline{R(x, \alpha)_k}$ in this sequence called a global history of **S**. If, for every $\overline{W(x, \alpha)_k}$ and $\overline{R(x, \alpha)_k}$ exist respective $\underline{W(x, \alpha)_k}$ and $\underline{R(x, \alpha)_k}$ farther in Q, then Q is called ***closed***, relative to the memory access operations. $\mathbf{E}^*$ denotes the set of all finite sequences of events from **E**.

Now, we are in a position to formally define the principle of sequential and strict consistency:

A global history $Q = q_1\ q_2 \ldots\ q_n$ fulfils the property of sequential consistency if Q is closed and there exists a permutation $\pi(Q) = q_{\pi(1)} q_{\pi(2)} \ldots q_{\pi(n)}$ (precisely, π is a permutation of the sequence 1, 2, …, n) such that:

(i) If event $\underline{W(x, \alpha)_k}$ or $\underline{R(x, \alpha)_k}$ occurs in π(Q), then it is preceded **adjacently** by respective $\overline{W(x, \alpha)_k}$ or $\overline{R(x, \alpha)_k}$. This means that between initiation and termination of access to a variable, no other access operation to this variable occurs.

(ii) Events $\overline{W(x, \alpha)_k}$ and $\overline{R(x, \alpha)_k}$ occur in the sequence π(Q) in the same order as in the local history of program P_k; by virtue of (i), operations $W(x, \alpha)_k$ and $R(x, \alpha)_k$ are performed in the system **S** in the same order as in the program P_k.

(iii) If an event $\overline{R(x, \alpha)_k}$ occurs in π(Q) then α is an initial value or this event is preceded in the sequence π(Q) by a certain event $\overline{W(x, \alpha)_j}$ with no event $\underline{W(x, \beta)_i}$ inbetween, where $\beta \neq \alpha$, $j, i = 1, 2, \ldots, N$.

The system **S** obeys the principle of sequential consistency of DSM if and only if **S** admits only global histories Q fulfilling the property of sequential consistency. In short: the memory managed by **S** is sequentially consistent.

The strict consistency is obtained if (i), (ii), (iii) are satisfied and if events $\overline{W(x, \alpha)_k}$ occur in π(Q) in the same order as in Q.

Remarks

1. The global history $Q = q_1\ q_2 \ldots\ q_n \in \mathbf{E}^*$ uniquely determines the memory state (content) and printouts yielded by Q. The memory state is a set of pairs $\sigma[Q] = \{\langle x, \alpha\rangle : x \in \mathbf{V}, \alpha \in \mathbf{D}\}$, thus a relation $\sigma[Q] \subseteq \mathbf{V} \times \mathbf{D}$ (see Chap. 10). If for each closed Q, the relation σ[Q] is a function $\forall\ x \in \mathbf{V}, \alpha, \beta \in \mathbf{D}$: $(x, \alpha) \in \sigma[Q] \wedge (x, \beta) \in \sigma[Q] \Rightarrow \alpha = \beta$, meaning that values of all replicas of each variable are identical), then the memory is coherent. Sequentially consistent memory is coherent. Memory incoherence (lack of memory integrity) arises e.g. in effect of parallel execution shown in Fig. 8.15. The execution yields the DSM content: in memory of computers 1, 2, 4 the value of x is β and of computer 3 is α (see also Table 8.6). The DSM state is then $\sigma[Q] = \{\langle x, \alpha\rangle, \langle x, \beta\rangle\}$, which is not a function.
2. It is known that in general a decision whether or not a given finite execution fulfils the sequential consistency property is algorithmically very complex (called "intractable"). This is evident in the model presented here: a search for a permutation satisfying (i), (ii), (iii) is required. A good many research tackled this task with the same answer, if not additional information has been supplied about the system behaviour. Some examples of this research are in (Gibbons and Korach 1992; Cantin et al. 2005; Hu et al. 2011).

3. Sequential and especially strict consistency, though lessens performance of the system, brings nearer the DSM mechanism to a multiprocessor system endowed with one physical shared memory of direct access. Applications where efficiency is more crucial than preservation of some execution order in individual programs, may tolerate more liberal models than sequential consistency, the natural model for users. Some of such models will be considered in the next Section.
4. From the formal point of view, the collection of variables in the DSM is a multiset, that is a function *Rep*: V $\rightarrow \mathfrak{N}$ ($\mathfrak{N}$–set of natural numbers), where *Rep*(*x*) is the number of replicas of *x* in the local memories. For instance, in the execution in Fig. 8.5 and Table 8.3 in Sect. 3: $V = \{a, b, x, y\}$, $Rep(a) = 2$, $Rep(b) = 2$, $Rep(x) = 1$, $Rep(y) = 1$.

8.6 Some Other Models of Memory Consistency

Sequential and strict memory consistency models so far considered, permit to use programs written for multiprocessor systems with a physical memory (RAM), common to all processors. Implementation of strict consistency makes system performance of unacceptably low level. Implementation of sequential consistency is, in this respect, more acceptable, however also lowers considerably efficiency, compared with multiprocessor systems. Its important property is preservation of the order of access to memory operations specified in individual programs, the feature natural for the users. Applications in which performance is more important than such preservation, may tolerate weaker, more liberal consistency models in cases where order of some actions, determined by individual programs, is inessential for the main objective and outcome of the whole parallel program. Two examples of such more liberal models are formally defined in the next subsections.

8.6.1 Causal Consistency (Hutto and Ahamad 1990)

In any execution, if effect of an update (write) operation depends on another update (in the same or different process and of the same or different variable), then in every process, the temporal order of readouts of the updated variables should be the same as the temporal order of these updates. This can be formalized as follows (denotation of symbols is as in Sect. 8.5). First, let us define a relation of causal dependency in the set of events: $\rightsquigarrow \subseteq \mathbf{E} \times \mathbf{E}$. For events

$p, q \in \mathbf{E}$ two auxiliary primary relations $\xrightarrow{process}$ and $\xrightarrow{readout}$ are admitted:

1. If p precedes q in the same process (in a local history) then $p \xrightarrow{process} q$ (see Sect. 4.2)
2. If event $q = \underline{R(x, \alpha)_j}$ terminates reading value α of variable x and α was assigned to x by a write operation completed with event $p = \underline{W(x, \alpha)_j}$ then $p \xrightarrow{readout} q$

Second, causal dependency $\rightsquigarrow$ is defined as the least (wrt. $\subseteq$) relation such that:

(i) if $p \xrightarrow{process} q$ or $p \xrightarrow{readout} q$ then $p \rightsquigarrow q$
(ii) if $p \rightsquigarrow q$ and $q \rightsquigarrow r$ then $p \rightsquigarrow r$

If $p \rightsquigarrow q$ then p is a *cause* of q and q is an effect of p. The events are independent if neither $p \rightsquigarrow q$ nor $q \rightsquigarrow p$, written $p \parallel q$.

Let $\underline{\boldsymbol{R}}$ be the set of events $\underline{R(x, \alpha)_j}$ and $\underline{\boldsymbol{W}}$ the set of events $\underline{W(x, \alpha)_j}$, $j \in \{1, 2, \ldots, N\}$, $x \in \boldsymbol{V}$, $\alpha \in \boldsymbol{D}$. Let $Q = q_1\, q_2 \ldots\, q_n \in \mathbf{E}^*$ be a closed history of execution of the system **S**. Q is *causally consistent* iff for any $q_i, q_j \in \underline{\boldsymbol{W}}$ in Q with $q_i \rightsquigarrow q_j$, and for any $q_k, q_l \in \underline{\boldsymbol{R}}$ in Q, the following holds: if $q_k \xrightarrow{process} q_l$ (q_k and q_l are in the same process without specifying their order, i.e. $q_k \xrightarrow{process} q_l \Leftrightarrow q_k \xrightarrow{process} q_l \vee q_l \xrightarrow{process} q_k$) and $q_i \xrightarrow{readout} q_k$ and $q_j \xrightarrow{readout} q_l$ then $q_k \xrightarrow{process} q_l$; but if for some $q_i, q_j \in \underline{\boldsymbol{W}}$, $q_k, q_l \in \underline{\boldsymbol{R}}$ in Q, the relations $q_i \rightsquigarrow q_j$, $q_i \xrightarrow{readout} q_k$ and $q_j \xrightarrow{readout} q_l$, $q_l \xrightarrow{process} q_k$ hold then Q is *causally inconsistent*. In symbols:

$$\forall q_i, q_j \in \underline{W}\,[q_i \rightsquigarrow q_j \Rightarrow \forall q_k, q_l \in \underline{R}\,((q_k \xrightarrow{process} q_l \wedge q_i \xrightarrow{readout} q_k \wedge q_j \xrightarrow{readout} q_l) \Rightarrow q_k \xrightarrow{process} q_l)]$$

The system **S** obeys the principle of causal consistency of DSM iff **S** admits only global causally consistent histories Q. In short: the memory managed by **S** is *causally consistent*.

Example. Let

$$Q = \underbrace{W(x, 9)_1}_{q1} \ldots \underbrace{R(x, 9)_2}_{q2} \ldots \underbrace{W(y, 3)_2}_{q3} \ldots \underbrace{R(x, 9)_3}_{q4} \ldots \underbrace{R(y, 3)_3}_{q5} \ldots \underbrace{R(y, 3)_4}_{q6} \ldots \underbrace{R(x, 9)_4}_{q7}$$

The underlines of symbols denoting terminations of R/W operations are omitted. This causally inconsistent execution is presented graphically in Fig. 8.17. By definition of causality the relation $q_1 \rightsquigarrow q_3$ holds. For instance, suppose that process P_2 after having read value 9 of variable x, computes $\sqrt[2]{9}$ and writes 3 to variable y. Also by definition of causality, the relations $q_1 \xrightarrow{readout} q_7$, $q_3 \xrightarrow{readout} q_6$, $q_6 \xrightarrow{process} q_7$ are fulfilled, which means that execution Q is causally inconsistent.

Conditions for causal consistency/inconsistency of this execution may be schematically presented by diagrams in Fig. 8.18, where the zigzag line denotes the causality relation.

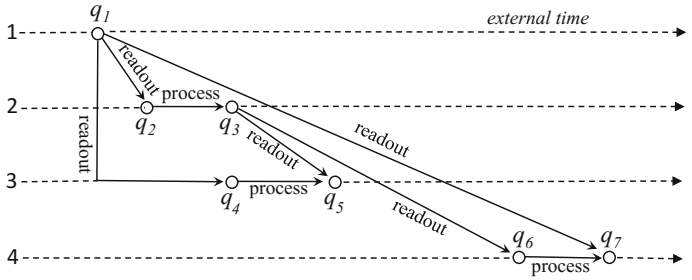


Fig. 8.17 System of four computers with causally consistent memory would not permit for such execution

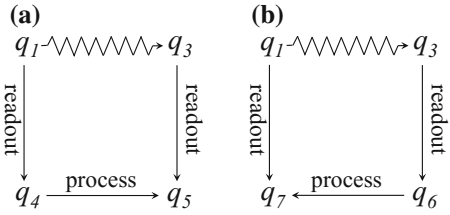


Fig. 8.18 **a** Diagram of causally consistent execution ... q_1... q_2... q_3... q_4... q_5. **b** Diagram of causally inconsistent execution ... q_1... q_2... q_3... q_4... q_5... q_6... q_7

8.6.2 PRAM (Pipelined Random Access Memory) Consistency (Lipton and Sandberg 1988)

In any execution, if two update (W) operations are in the same process, then in every process, the global time order of readouts (R) of these updated variables should be the same as of the updates. This can be formalized as follows. Let Q = q_1 q_2... $q_n \in \mathbf{E}^*$ be a closed history of execution of the system **S**. Q is *PRAM-consistent* iff for any $q_i, q_j \in \underline{\boldsymbol{W}}$ in Q with $q_i \xrightarrow{process} q_j$ and for any $q_k, q_l \in \underline{\boldsymbol{R}}$ in Q, the following holds: if $q_k \xrightarrow{process} q_l$ and $q_i \xrightarrow{readout} q_k$ and $q_j \xrightarrow{readout} q_l$ then $q_k \xrightarrow{process} q_l$; but if for some $q_i, q_j \in \underline{\boldsymbol{W}}$, $q_k, q_l \in \underline{\boldsymbol{R}}$ in Q, the relations $q_i \xrightarrow{process} q_j$, $q_k \xrightarrow{process} q_l$, $q_i \xrightarrow{readout} q_k$, $q_j \xrightarrow{readout} q_l$ are fulfilled, then Q is *PRAM-inconsistent*. In symbols:

$$\forall\, q_i, q_j \in \underline{W}[q_i \xrightarrow{process} q_j \Rightarrow \forall q_k,\ q_l \in \underline{R}\,((q_k \xrightarrow{process} q_l \wedge q_i \xrightarrow{readout} q_k \wedge q_j \xrightarrow{readout} q_l) \Rightarrow q_k \xrightarrow{process} q_l)]$$

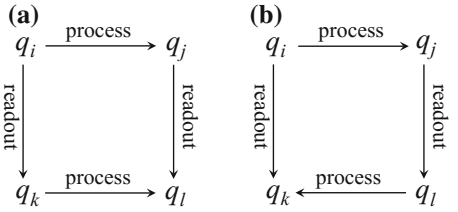


Fig. 8.19 (**a**) PRAM-consistency (**b**) PRAM-inconsistency

The system **S** obeys the principle of PRAM-consistency of DSM iff **S** admits only global causally consistent histories Q. In short: the memory managed by **S** is *PRAM-consistent*.

Conditions for PRAM-consistency/inconsistency of this execution may be schematically presented by diagrams in Fig. 8.19.

Notice that causally consistent memory is PRAM-consistent (obviously the formula defining causal consistency, implies the formula defining the PRAM-consistency), but not conversely (e.g. inconsistent interleaving Q may be PRAM-consistent, as shown in Fig. 8.17).

Remarks

1. The term "PRAM" is being used in the theory of computational complexity in a different meaning than in case of memory consistency investigations. In the theory of computational complexity, it is an abbreviation of "Parallel Random Access Machine", a certain formal model of parallel computation, for making proofs of complexity of algorithms partitioned onto fragments concurrently executed.
2. Notice that the concept of memory consistency depends on adopted model of concurrent processing and what actions have been chosen as basic for this concept. They have been chosen here as events of initiation and complete termination of access to memory. In this model, the concurrency is being modelled by indeterminism: the events may occur in various temporal order, without influence on the outcome of programs' activity, in particular on preserving adopted kind of memory consistence. In such model, closest to reality of distributed processing, the relationships between considered above kinds of consistency models, are easily noticed: a strictly consistent memory is sequentially consistent, which is causally consistent, which is PRAM-consistent.

Apart from the four kinds (models) of memory consistency presented above, a number of others have been created, even less restrictive, some of them not being in the hierarchy with just mentioned. They have been devised for particular applications, where discordance with the "natural" order (ensuing from arrangement of actions in individual programs) of access to memory is permitted, in favour of system better performance. Some of them may be mentioned:

1. weak consistency
2. entry consistency
3. release consistency
4. scope consistency
5. process consistency
6. cache consistency
7. fork consistency

Models 1, 2, 3 require usage of some operations synchronizing access to memory, by the user. This lessens the transparency feature of distributed system and requires additional organization activities in programs, but improves performance. This is not exhaustive list of possible models of DSM consistency, implemented in various research institutions. Their extensive presentation would exceed the scope of this book, as well as other books on distributed systems. This may be found in original papers on the subject of construction of specific systems.

8.7 Exemplary Algorithms Realizing Memory Consistency

Depending on the user needs, the algorithms are divided into two classes called fast-read and fast-write. Computers in the parallel systems, as before, are numbered 1, 2, …, N.

Denotations:

M_i local memory of computer number i
$M_i[x]$ content of a cell assigned to variable x in M_i

Actions ***wait*** and ***signal*** denote suspension and resumption activity of a process.

The grey boxes contain procedures performing operations, respectively, of reading, group communication (broadcast) and storing a value in memory. In the curly brackets are comments.

8.7.1 Algorithms for Sequential Consistency

Fast-read algorithm implementing sequential consistency for computer of number *i*

{*Execution of a read statement* $R(x, M_i[x])_i$ *in computer i*}
$M_i[x] \rightarrow A_i$ {*transfer value* $M_i[x]$ *from memory to a register* A_i}

{*Execution of a write statement* $W(x,)_i$ *in computer i*}

$\alpha \xrightarrow{atomic\_broadcast} M_j[x]$ to all computers of number j where x is located {*atomic, that is indivisible broadcast (exclusive, i.e. with suspended write operations of other computers)*};
wait {*waiting for permission from subsystem managing memory access to resume activity of computer i; the permission is issued when the broadcast is completed*}

{*Storing value received from computer of number j in the cell assigned to variable x in memory* M_i *with conditional permission to continue run of computer i*}
$M_i[x] := \alpha$;
if $j = i$ **then** ***signal*** {*resumes activity of computer i*}

Fast-write algorithm implementing sequential consistency for computer number *i*

Denotation: num_i—counter of pending (unfinished) write operations; initially set to 0. Meaning of this counter is illustrated in Fig. 8.20. The counter is needed when computer of number i repeatedly requests for writing. Any request does not wait for its completion, but may be succeeded by a next request, etc.

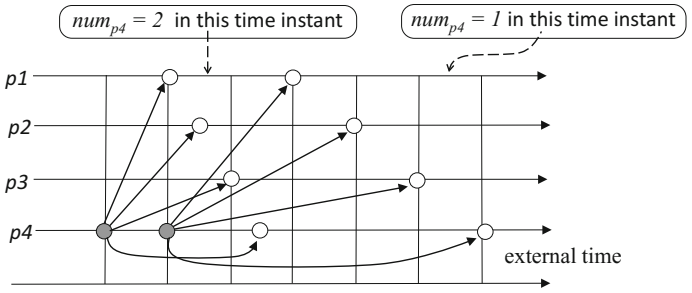


Fig. 8.20 Grey circles represent send operations, blank circles—reception. This is a (global) FIFO broadcast

{*Execution of the read statement* $R(x, M_i[x])_i$ *in computer* i}

if $num_i \neq 0$ **then** ***wait*** {makes computer i suspend until the "***signal***" event comes};

$M_i[x] \rightarrow A_i$ {*transfer value* $M_i[x]$ *from memory to a register* A_i}

{*Execution of the write statement* $W(x, \alpha)_i$ *in computer* i}
$num_i := num_i + 1$;

$\alpha \xrightarrow{FIFO-atomic\_broadcast} M_j[x]$ to all computers of number j where x is located {*for the (global) FIFO and non-FIFO broadcast, see* Figs. 5.6 and 5.7 in Chap. 5}

{*Storing value* α *received from computer of number j, in a cell assigned to variable* x *in memory* M_i *with conditional permission to resume run of computer* i}

$M_i[x] := \alpha$;

if $j = i$ **then begin** $num_i := num_i - 1$;
if $num_i = 0$ **then** ***signal*** {*resumes activity of computer* i}
end;

8.7.2 An Algorithm Implementing Causal Consistency for Computer of Number i

Before presenting this algorithm, let us illustrate pictorially the meaning of causal broadcast and not causal broadcast in Figs. 8.21 and 8.22 respectively.

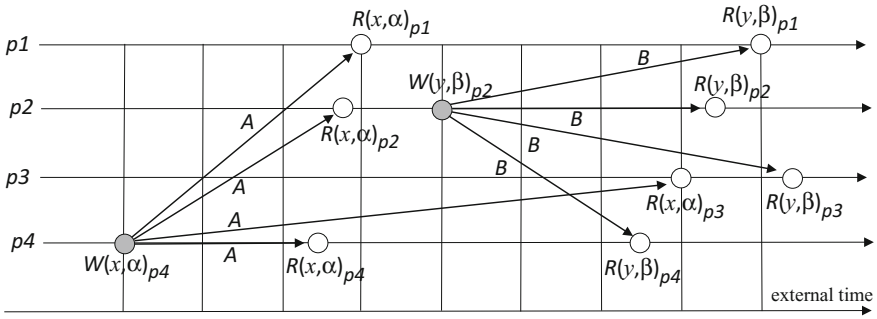


Fig. 8.21 Causal broadcast; message B is causally dependent on message A: relation $W(x, \alpha)_{p4} \rightsquigarrow W(y, \beta)_{p2}$ is fulfilled, $\beta = f(\alpha)$, where f is a certain function

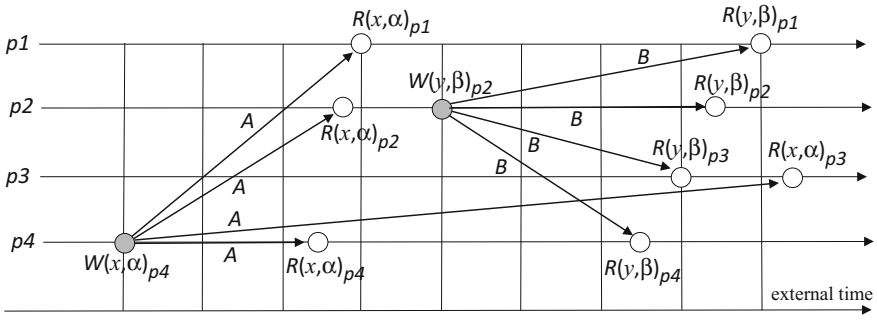


Fig. 8.22 This is not a causal broadcast: relation $W(x, \alpha)_{p4} \rightsquigarrow W(y, \beta)_{p2}$ is fulfilled, but $R(x, \alpha)_{p3}$ occurs later than $R(y, \beta)_{p3}$

{*Execution of a read statement* $R(x, M_i[x])_i$ *in computer* i}
$M_i[x] \rightarrow A_i$ {*transfer value* $M_i[x]$ *from memory to a register* A_i}

{*Execution of a write statement* $W(x, \alpha)_i$ *in computer* i}
$\alpha \xrightarrow{causal\_broadcast} M_j[x]$ to all computers of number j where x is located;

{*Conditional storing value* α *received from computer of number* j *in the cell assigned to variable* x *in memory* M_i}

if $j \neq i$ **then** $M_i[x] := \alpha$

8.7.3 An Algorithm Implementing PRAM Consistency for Computer of Number i

{*Execution of a read statement* $R(x, M_i[x])_i$ *in computer* i}
$M_i[x] \rightarrow A_i$ {*transfer value* $M_i[x]$ *from memory to a register* A_i}

{*Execution of a write statement* $W(x, \alpha)_i$ *in computer* i}
$\alpha \xrightarrow{FIFO\_broadcast} M_j[x]$ to all computers of number j where x is located;

{*Conditional storing value* α *received from computer of number* j *in the cell assigned to variable* x *in memory* M_i}
if $j \neq i$ **then** $M_i[x] := \alpha$

Remark The presented algorithms for sequential, causal and PRAM consistency comprise outlines of activity performed by subsystems managing memory access.

References

Cantin, J. F., Lipasti, M. H., & Smith, J. E. (2005). The complexity of verifying memory coherence and consistency. *IEEE Transactions on Parallel and Distributed Systems, 16*(7), 651–663.

Courtois, J., Heymans, F., & Parnas, D. L. (1971). Concurrent control with "Readers" and "Writers". *Communication of the ACM, 14*(10), 667–668.

Czaja, L. (1968). Organization of segment exchange in ALGOL for ZAM 21 ALFA and ZAM 41 ALFA computers. *Algorytmy, 5*(9), 77–84.

Czaja, L. (1980). Deadlock and fairness in parallel schemas: A set-theoretic characterization and decision problems. *Information Processing Letters, 10*(4–5), 234–239.

Czaja, L. (2002). *Elementary Cause-Effect Structures.* Wydawnictwa Uniwersytetu Warszawskiego (Warsaw University Publisher).

Czaja, L. (2012). Exclusive access to resources in distributed shared memory architecture. *Fundamenta Informaticae, 119*(3–4), 265–280.

Diekert, V., & Rozenberg, G. (Eds.). (1995). *The book of traces.* World Scientific Publishing Co.

Gibbons, P. B., & Korach, E. (1992). The Complexity of Sequential Consistency. In *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium* (pp. 317–325).

Hoogeboom, H. J., & Rozenberg, G. (1995). Dependence graphs. In *The book of traces*. Singapore: World Scientific.

Hu, W., Chen, Y., Chen, T., Qian, C., & Li, L. (2011). Linear time memory consistency verification. *IEEE Transactions on Computers, 61*(4), 502–516.

Hutto, P. W., & Ahamad, M. (1990). Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Distributed Computing Systems, 1990. Proceedings, 10th International Conference IEEE* (pp. 302–311).

Lamport L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers, 28*, 690–691.

Li, K., & Hudak, P. (1989). Memory coherence in shared virtual memory systems. *ACM Transaction On Computer Systems, 7*(4), 321–359.

Lipton, R. J., & Sandberg, J. S. (1998). *Pram: A scalable shared memory*. Technical Report CS-TR-180-88, New Jersey: Princeton University.

Madnick, S. E., & Donovan, J. J. (1974). *Operating systems* (Vol. 197). New York: McGraw-Hill Book Company.

Mazurkiewicz, A. (1987). Trace theory. In W. W. Brauer et al. (Eds.), *Petri Nets, Application and Relationship to other Models of Concurrency: Vol. 255. Lecture notes in computer sciences* (pp. 279–324).

Mosberger, D. (1993). Memory consistency models. *ACM SIGOPS Operating Systems Review, 27* (1), 18–26.

Nutt, G. (2002). *Operating systems: A modern perspective* (2nd ed.). Boston: Addison-Wesley.

Petri, C. A. (1966). *Communication with automata* (Report, Vol. 1 Suppl. 1 RADC TR-65-377), Applied Data Research, Contract AF 30 Princeton N.J.

Reisig, W. (1985). *Petri nets: An introduction*, monographs on theoretical computer science (Vol. 4). Berlin: Springer.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2009). *Operating system concepts* (8th ed.). New Jersey: Wiley.

Tanenbaum, A. S. (1995). *Distributed operating systems.* United States: Prentice-Hall.

Chapter 9
The Control Flow During Execution of the Alternating Bit Protocol Specified by the Cause-Effect Structure

Flow of control during activity of the Alternating Bit Protocol (ABP) (Barlett et al. 1969), will be illustrated as a flow of tokens in a cause/effect (c/e) structure that specifies this control flow. C/E structures (Czaja 1988, 2002) is an algebraic calculus, devised for specification and analysis of parallel processes. Basic notions and properties of the calculus is outlined in Chap. 10. Pictorially, a c/e structure is a graph in which nodes (places) are named. Every name of a node is endowed with a superscript and subscript being terms called the formal polynomials, whose arguments are names of predecessors (in the superscript) and successors (in the subscript) of this node. The operators connecting arguments are „+" and „•", called addition and multiplication, where „+" means nondeterministic choice and „•" simultaneity of receiving (in case of superscript) and sending (in case of subscript) tokens. Figure 9.1 shows possible transformations of an exemplary c/e structure marked with tokens: a flow of token initially residing in the place a. The token moves to b, then it „splits" and moves to c and d simultaneously, then one from c to e then back to a, while the other remains in d forever. A different sequence of transformations occurs, when the token moves from a to b, then to e, then back to a. Polynomial θ (empty) means „no successors or predecessors exist". Tokens flow in accordance with the rules determined by semantics of c/e structures given in Chap. 10.

The c/e structure in Fig. 9.1 may be defined as an expression, called "arrow expression" $(a \rightarrow b) + (b \rightarrow c)\bullet(b \rightarrow d) + (b \rightarrow e) + (c \rightarrow e) + (e \rightarrow d) + (e \rightarrow a)$ in accordance with algebraic composition rules and denotational conventions given in Chap. 10. Although c/e structures and Petri nets are formalisms of equivalent descriptive capability (Raczunas 1993), in the pictorial presentation the c/e structures do not explicitly comprise transitions, thus allow for much more concise graphical form. The counterparts to transitions, the so-called "firing components", are derived from the formal polynomials. That is why the c/e structures are used here for presentation of the ABP protocol behaviour in Table 9.1. The task is to transmit messages from sender to receiver through an unreliable channel, so that: (a) messages reach destinations in the order of their dispatch, (b) messages lost in

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_9

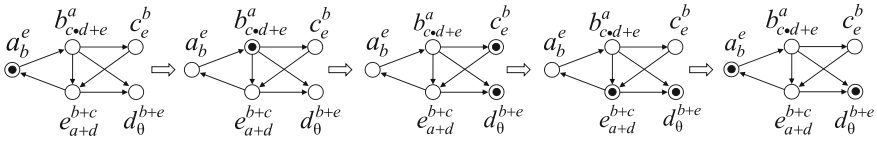


Fig. 9.1 Consecutive transformations of state of exemplary c/e structure

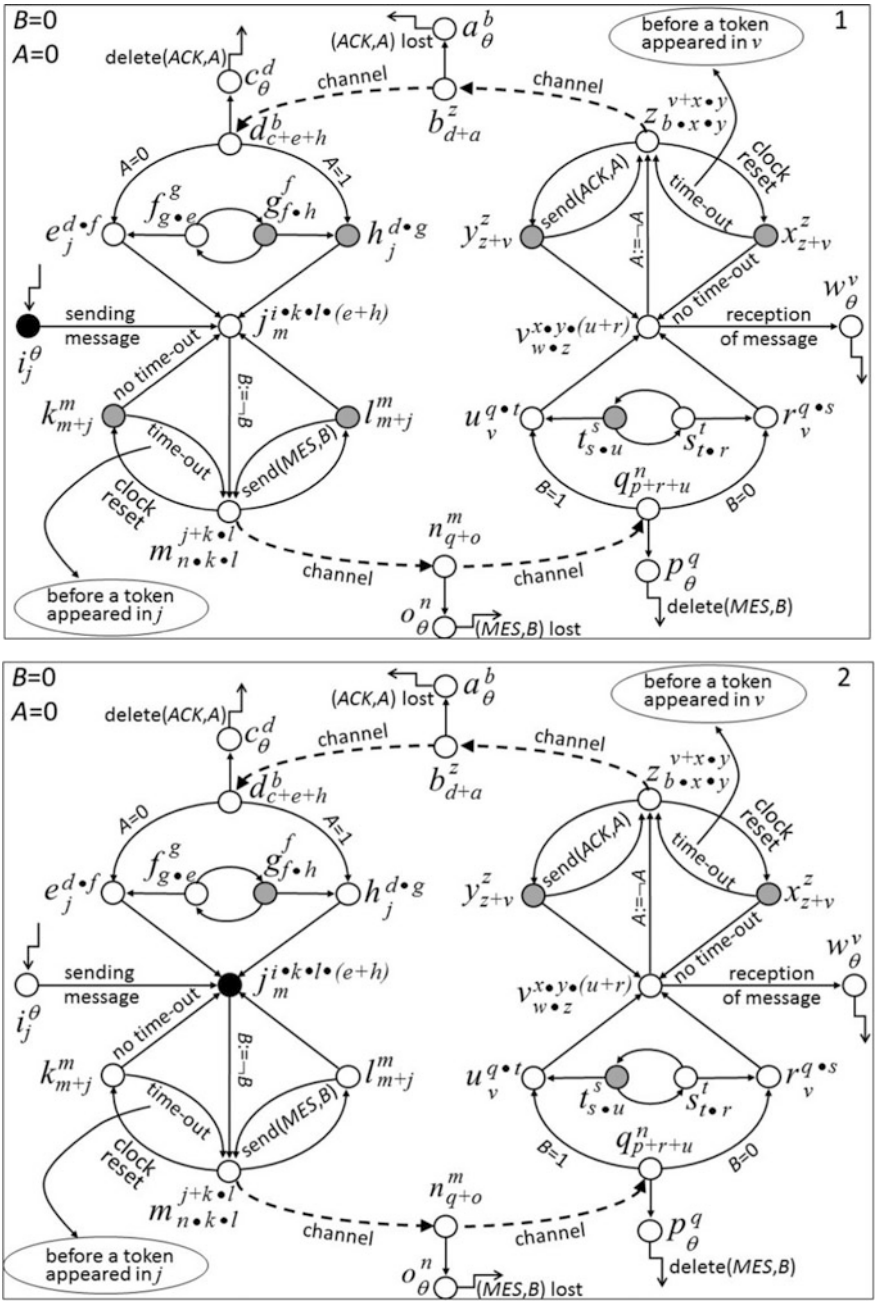
the channel are duplicated and dispatched again, (c) any message reaches its destination exactly once. A message, unless the first, is not taken for dispatch before acknowledgement of reception of the preceding message arrives. In the graphic presentation, the black tokens • in nodes represent message and acknowledgment, which along with grey tokens ● in auxiliary ("technical") nodes—make a current state of the protocol. Some arrows are annotated by conditions and statements, involving truth-valued (1 = *true*, 0 = *false*) variables BIT, ACK and a variable MES for storing messages. They are external to the ABP c-e structure, so, do not belong to the c/e structure definition and play a role of comments only. The left part is the sending module, the right part—the reception module.

The c/e structure named ALTERNATINGBITPROTOCOL may be defined by the "arrow expression" as combination of its substructures by means of addition and multiplication operations of c/e structures (Chap. 10). It may be done by introducing the following denotations with mnemonic names of the substructures:

MESCHAN = $m \rightarrow n$ (channel for message)
ACKCHAN = $z \rightarrow b$ (channel for acknowledgment)
TAKEMES = $(i \rightarrow j)\bullet(k \rightarrow j)\bullet(l \rightarrow j)\bullet[(e \rightarrow j) + (h \rightarrow j)]$
SENDMES = $(m \rightarrow l)\bullet(m \rightarrow k)\bullet(l \rightarrow m)\bullet(k \rightarrow m)$
MESALT = $j \rightarrow m$
ACKFLIPFLOP = $(f \rightarrow e)\bullet(f \rightarrow g)\bullet(g \rightarrow f)\bullet(g \rightarrow h)$
ACKSKIP = $d \rightarrow c$
ACKLOST = $b \rightarrow a$
RECEIVEACK = $(b \rightarrow d)\bullet[(d \rightarrow e) + (d \rightarrow h)]$•ACKFLIPFLOP + ACKSKIP + ACKLOST
TRANSMIT = SENDMES•MESCHAN + MESALT + RECEIVEACK• TAKEMES
DELIVERMES = $(v \rightarrow w)\bullet(x \rightarrow v)\bullet(y \rightarrow v)\bullet[(r \rightarrow v) + (u \rightarrow v)]$
SENDACK = $(z \rightarrow y)\bullet(z \rightarrow x)\bullet(y \rightarrow z)\bullet(x \rightarrow z)$
ACKALT = $v \rightarrow z$
MESFLIPFLOP = $(t \rightarrow u)\bullet(t \rightarrow s)\bullet(s \rightarrow t)\bullet(s \rightarrow r)$
MESSKIP = $q \rightarrow p$
MESLOST = $n \rightarrow o$
RECEIVEMES = $(n \rightarrow q)\bullet[(q \rightarrow r) + (q \rightarrow u)]$•MESFLIPFLOP + MESSKIP + MESLOST
RECEIVE = SENDACK•ACKCHAN + ACKALT + RECEIVEMES•DELIVERMES
ALTERNATINGBITPROTOCOL = TRANSMIT + RECEIVE

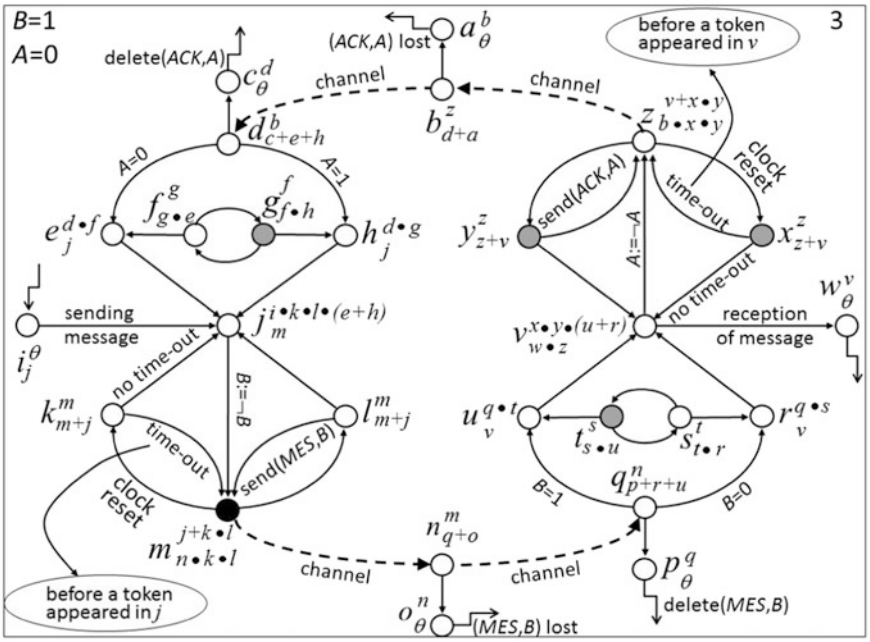
Table 9.1 **State 1**: a message has arrived in node *i* from the sender for dispatch, the time-out since the clock reset has not elapsed, so, the control tokens in nodes *h*, *k*, *l* indicate readiness to its dispatch. **State 2**: the message can be passed to *m*, the start node to the channel, nodes *h*, *k*, *l* are deactivated—lose tokens. **State 3**: the message has passed to node *m* and waits for transmission, value 0 of *B* changed into 1—the opposite value. **State 4**: the message along with value of *B* has been passed to the node *n* in the channel, the control nodes *k*, *l* have resumed activity (tokens), the sender's clock is reset. **State 5**: the current message successfully passed to the node *q* in the reception part, a new message has arrived in node *i* from the sender for dispatch, but cannot be passed to node *j*, because neither *e* nor *h* is active. **State 6**: the current message has been directed to node *u* by means of the flip-flop $t \longleftrightarrow s$ (which changes its state) and value 1 of *B*. **State 7**: the current message passes to node *v* making nodes *x*, *y* inactive (at the state 6, nodes *x*, *y* indicated that the time-out since the clock reset had not been exceeded). **State 8**: the current message reaches the receiver, the acknowledgment of delivery is passed to node *z* and value of *A* has been changed to 1; time-out of the sender's clock elapsed—deactivation of nodes *k*, *l* takes place, and the current message in node *m* waits for retransmission. **State 9**: the acknowledgment along with value of *A* has been passed to the node *b* in the channel, the sender's and receiver's clocks are reset (activated nodes *k*, *l* and *x*, *y*), the retransmitted current message is passed to node *n* in the channel. **State 10**: the acknowledgment successfully passed to the node *d* in the sending part, the retransmitted current message reaches node *q*. **State 11**: The acknowledgment has been directed to node *h* by means of the flip-flop $f \longleftrightarrow g$ (which changes its state) and value 1 of *A*, the retransmitted current message has passed from node *q* to *p* because $B = 1$ and node *t* of the flip-flop is not active. **State 12**: the new message passes from node *i* to *j*—action as at the state 2, the retransmitted current message has been deleted; time-out of the receiver's clock elapsed—deactivation of nodes *x*, *y* takes place, and the acknowledgment in node *z* waits for retransmission. **State 13**: the new message has passed to node *m* and waits for transmission, value 1 of *B* has been changed to 0—the opposite value. **State 14**: the new message along with value of *B* has been passed to the node *n* in the channel, the control nodes *k*, *l* have resumed activity (tokens), the sender's clock is reset. **State 15**: the new message successfully passed to the node *q* in the reception part. **State 16**: the new message has been directed to node *r* by means of the flip-flop $t \longleftrightarrow s$ (which changes its state) and value 0 of *B*. **State 17**: the new message passes to node *v* making nodes *x*, *y* inactive (at the state 16, nodes *x*, *y* indicated that the time-out since the clock reset had not been exceeded). **State 18**: the new message reaches the receiver, the acknowledgment of its delivery is passed to node *z* and value of bit *A* changed to 0

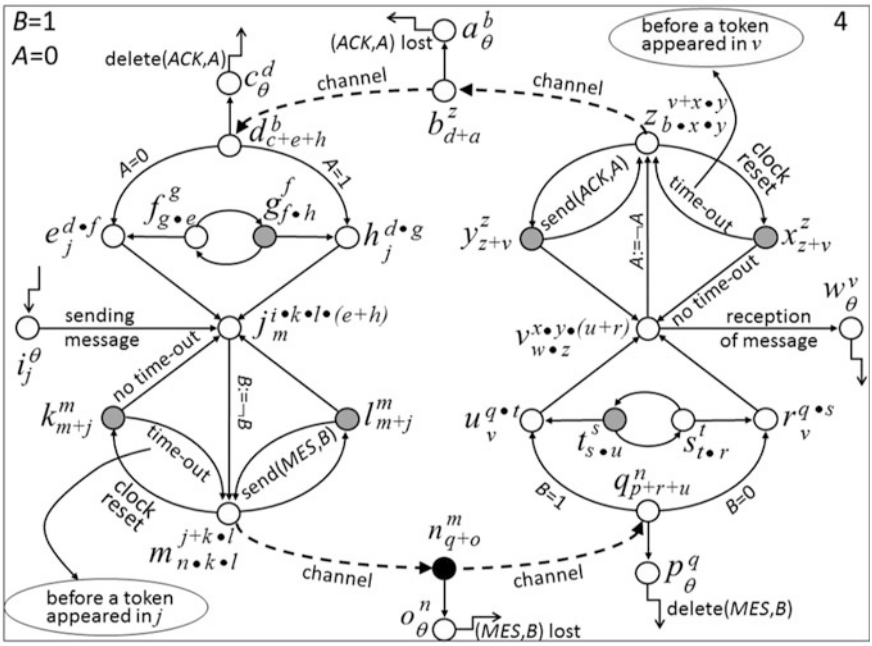
Table 9.1 (continued)



(continued)

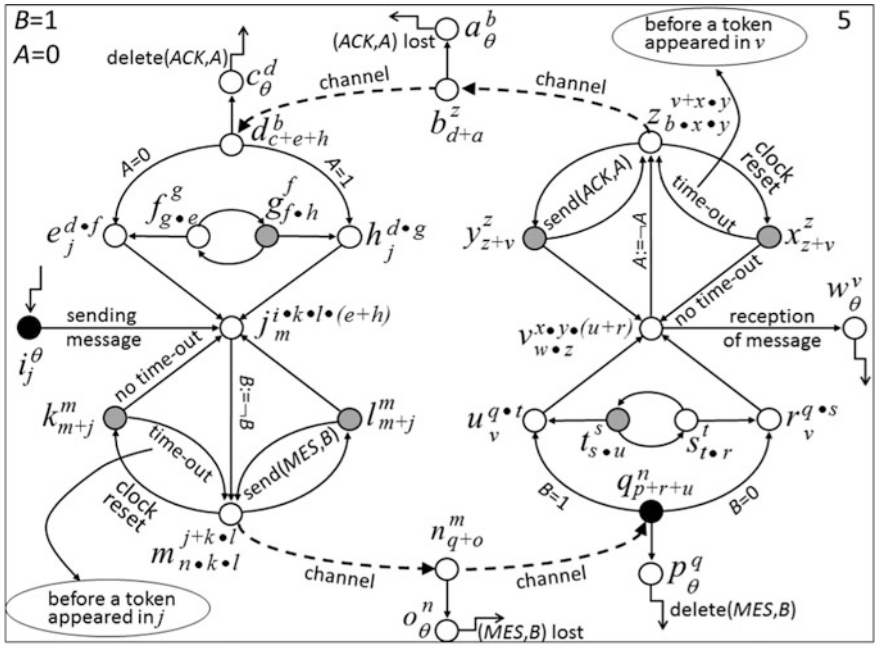
Table 9.1 (continued)

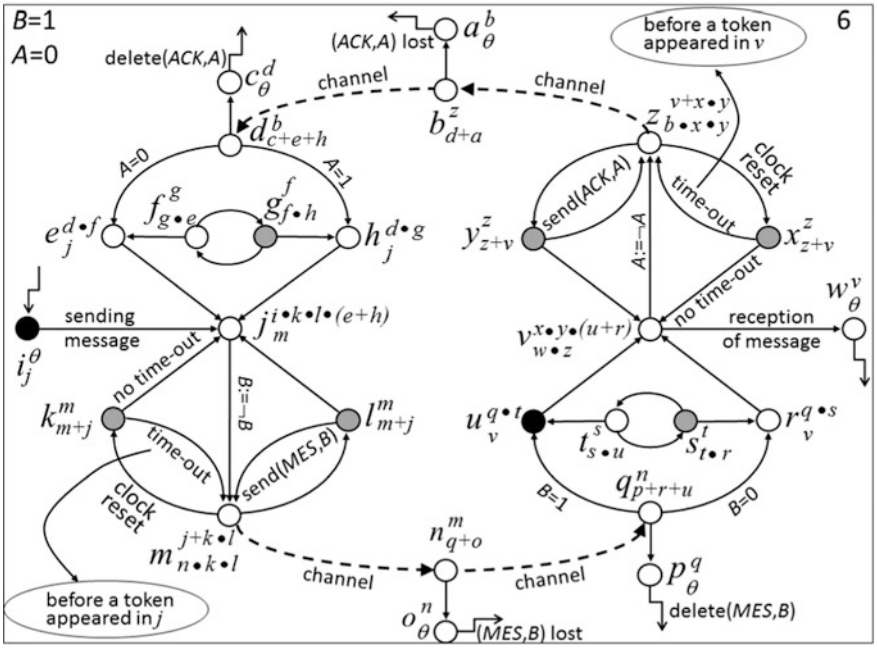




(continued)

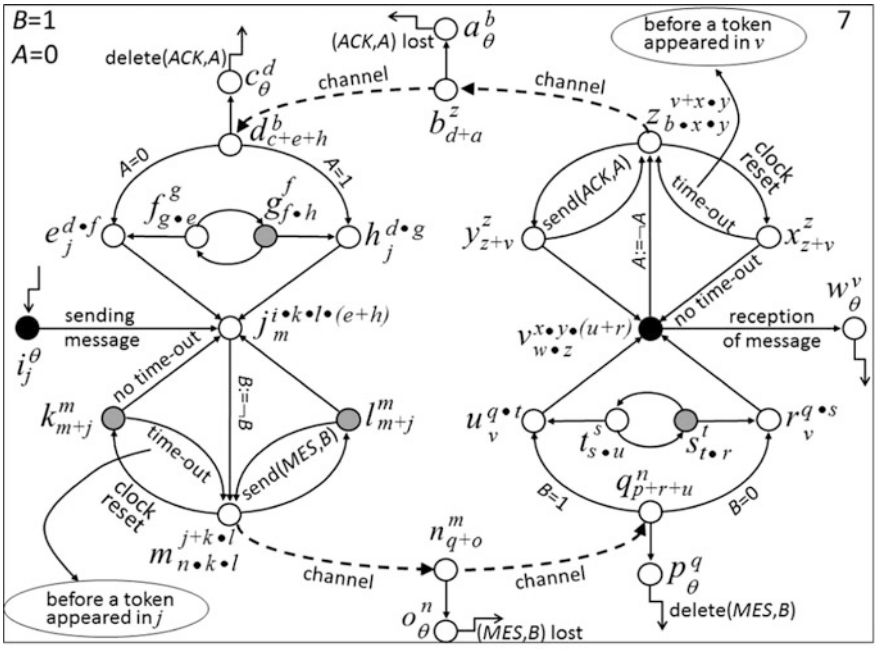
Table 9.1 (continued)

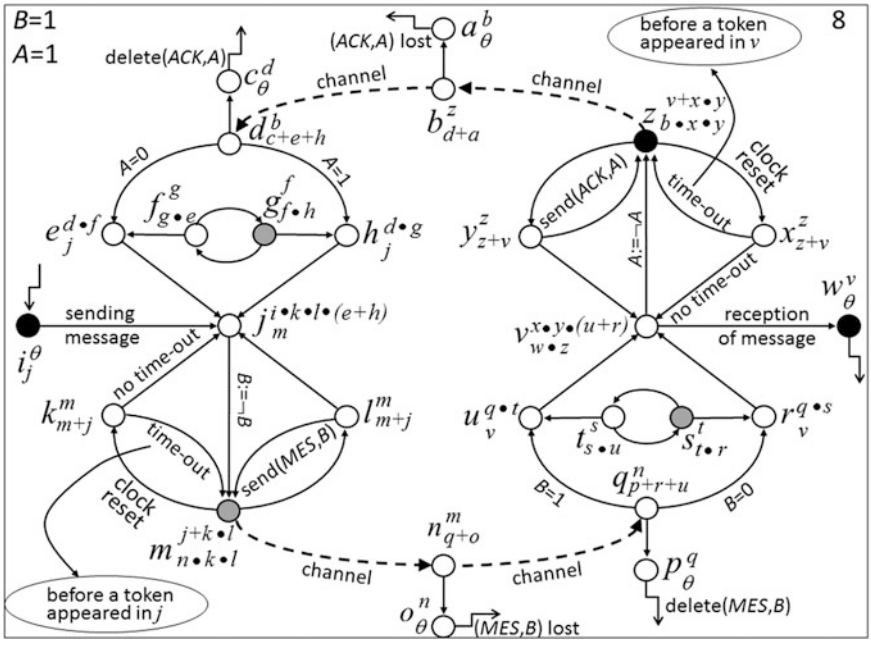




(continued)

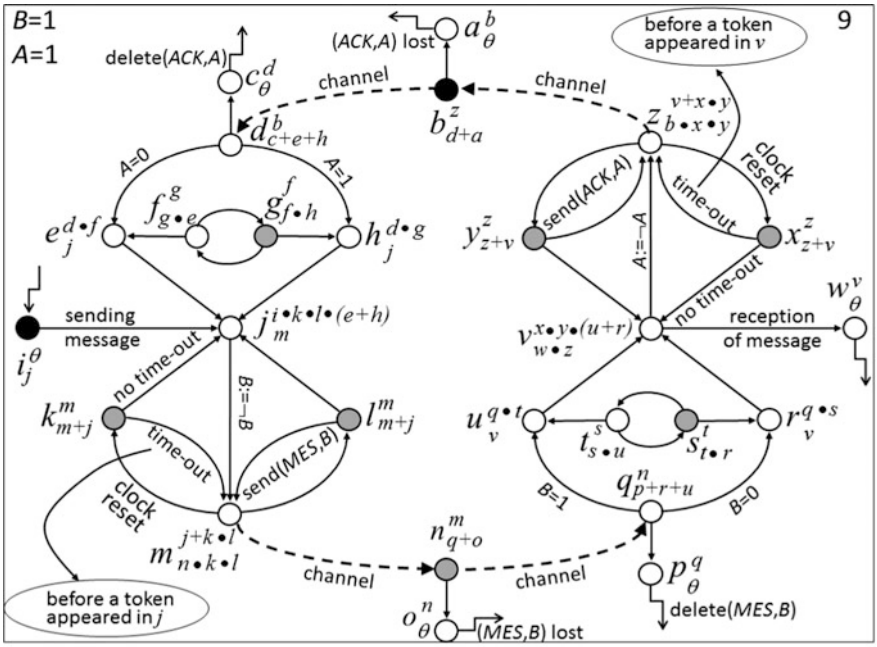
Table 9.1 (continued)

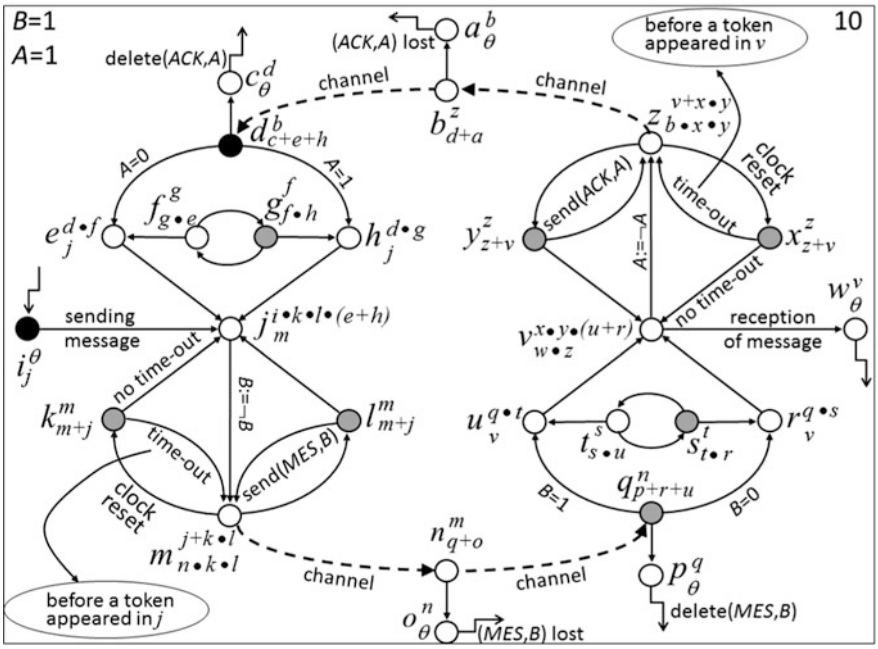




(continued)

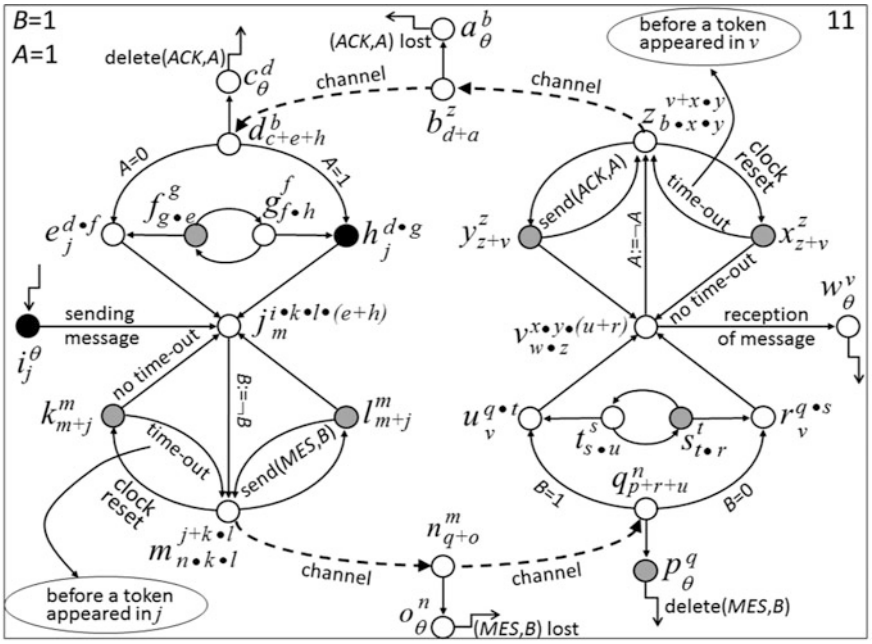
Table 9.1 (continued)

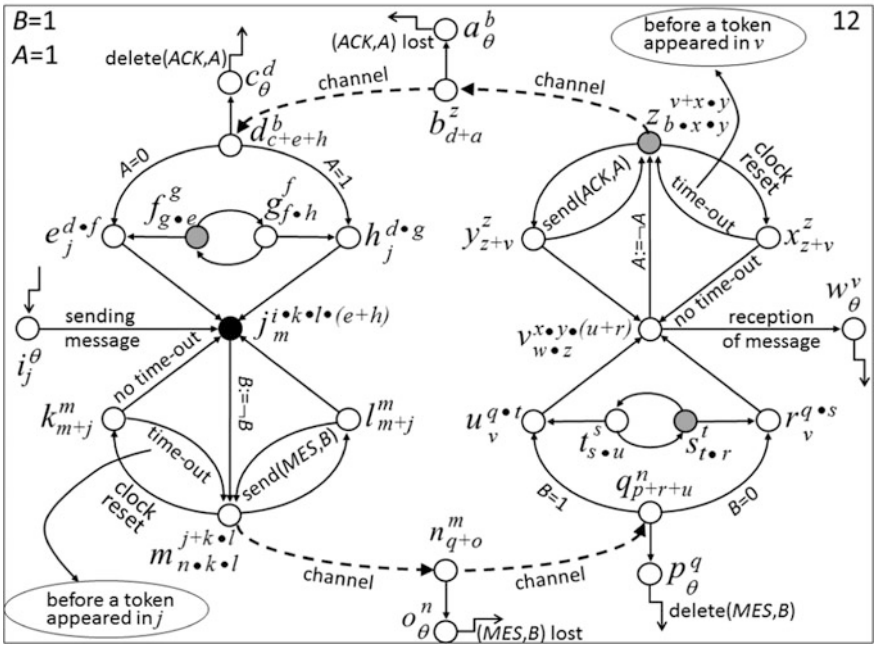




(continued)

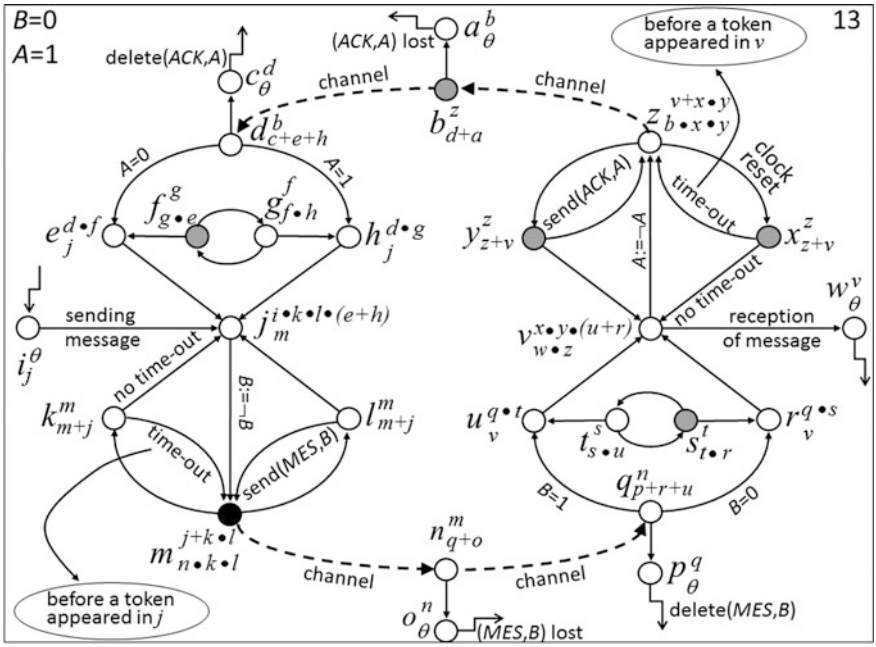
Table 9.1 (continued)

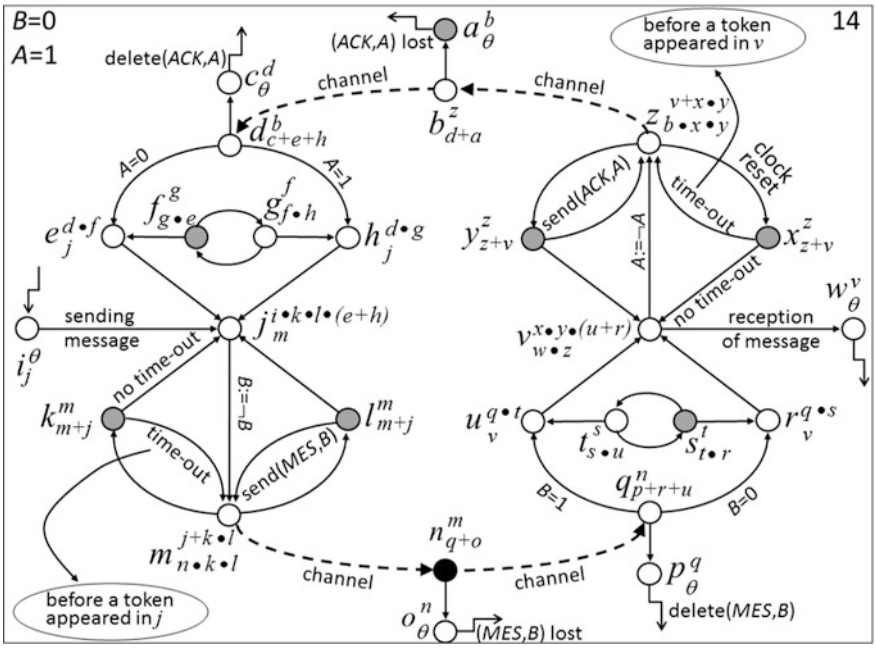




(continued)

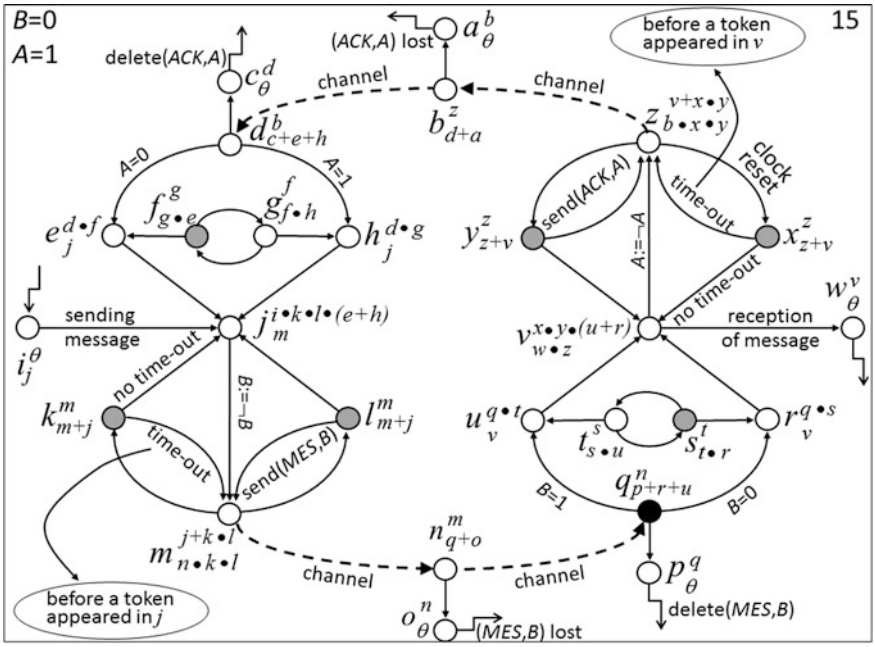
Table 9.1 (continued)

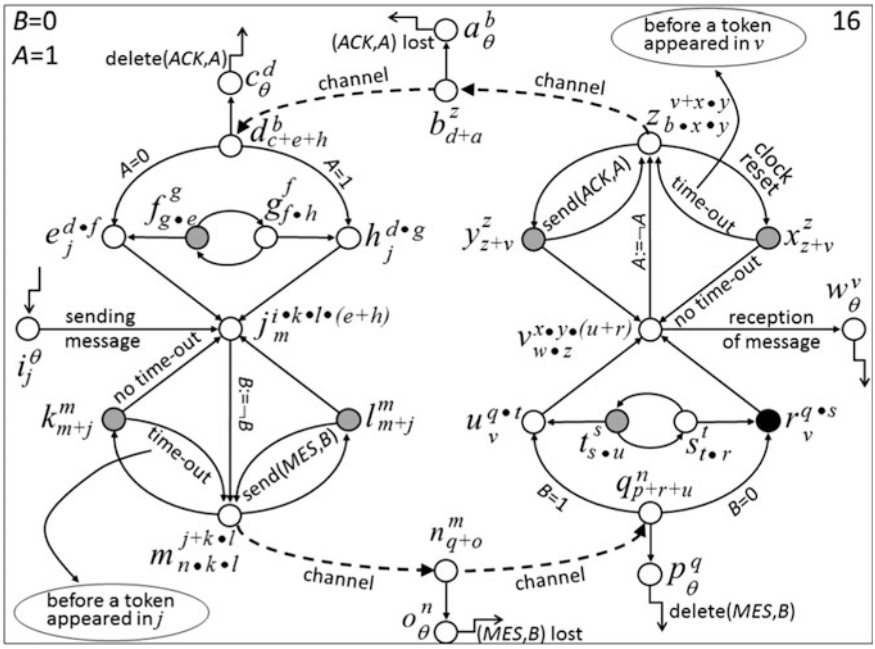




(continued)

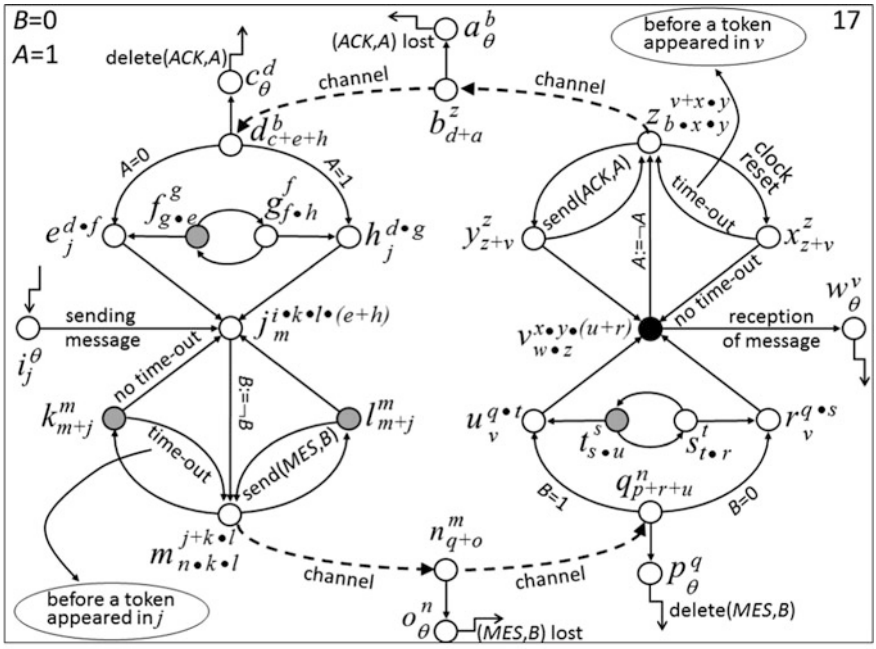
Table 9.1 (continued)

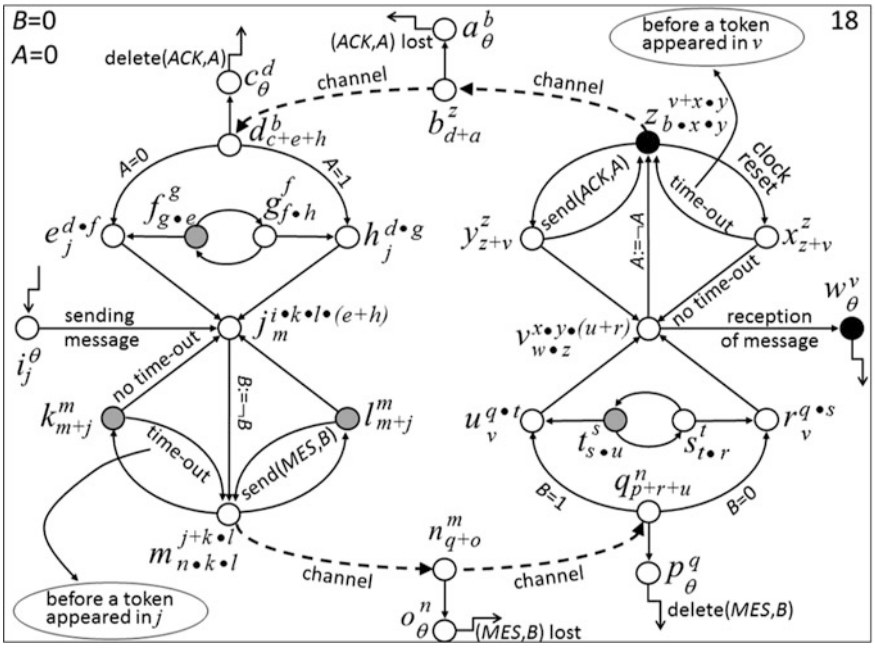




(continued)

Table 9.1 (continued)





References

Barlett, K. A., Sclantlebury, R. A., & Wilkinson, P. T. (1969). A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM, 12*(5), 260–261.

Czaja, L. (1988). Cause-effect structures. *Information Processing Letters, 26,* 313–319.

Czaja, L. (2002). *Elementary cause-effect structures*. Wydawnictwa Uniwersytetu Warszawskiego.

Raczunas, M. (1993). Remarks on the equivalence of c-e structures and Petri nets. *Information Processing Letters, 45,* 165–169.

Chapter 10
Some Mathematical Notions Used in the Previous Chapters

10.1 Binary Relations

A binary relation, which connects elements of a set $\mathbb{X}$ with elements of a set $\mathbb{Y}$, is any subset of the Cartesian product $\mathbb{X} \times \mathbb{Y}$. For a relation $\mathbb{R} \subseteq \mathbb{X} \times \mathbb{Y}$, its domain is the set $dom(\mathbb{R}) = \{x | \exists y : (x, y) \in \mathbb{R}\}$ and its codomain (or range) is the set $cod(\mathbb{R}) = \{y | \exists x : (x,\ y) \in \mathbb{R}\}$.

A relation reverse to $\mathbb{R}$ denoted by $\mathbb{R}^{-1}$, is defined by $(x, y) \in \mathbb{R}^{-1} \Leftrightarrow (y,\ x) \in \mathbb{R}$.

A composition of relations $\mathbb{P}$, $\mathbb{R}$ is a relation denoted by $\mathbb{P} \circ \mathbb{R}$ defined by $(x,\ y) \in \mathbb{P} \circ \mathbb{R} \Leftrightarrow \exists z : (x,\ z) \in \mathbb{P} \wedge (z,\ y) \in \mathbb{R}$. This is associative operation: $\mathbb{P} \circ (\mathbb{R} \circ \mathbb{S}) = (\mathbb{P} \circ \mathbb{R}) \circ \mathbb{S}$, thus the parentheses may be omitted. The infinite iteration of $\mathbb{R}$ is $\mathbb{R}^+ = \bigcup_{k=1}^{\infty} \mathbb{R}^k]] >$ where $\mathbb{R}^k = \mathbb{R} \circ \mathbb{R} \circ \cdots \circ \mathbb{R}$ (k-times). Identity relation in a set $\mathbb{X}$ is $id_{\mathbb{X}} = \{(x,\ x) | x \in \mathbb{X}\}$.

Relation $\mathbb{R}$ is a function $\mathbb{R} : dom(\mathbb{R}) \rightarrow cod(\mathbb{R})$ iff for every x, y, z the following holds: $(x,\ y) \in \mathbb{R} \wedge (x,\ z) \in \mathbb{R} \Rightarrow y = z$. Some elementary properties of relations are:

(a) $dom(\mathbb{R}) = \varnothing \Leftrightarrow cod(\mathbb{R}) = \varnothing \Leftrightarrow \mathbb{R} = \varnothing$ (the empty relation)
(b) $(\mathbb{R}^{-1})^{-1} = \mathbb{R}$
(c) $(\mathbb{P} \circ \mathbb{R})^{-1} = \mathbb{R}^{-1} \circ \mathbb{P}^{-1}$
(d) $(\mathbb{P} \cup \mathbb{R})^{-1} = \mathbb{P}^{-1} \cup \mathbb{R}^{-1}$
(e) $(\mathbb{P} \cap \mathbb{R})^{-1} = \mathbb{P}^{-1} \cap \mathbb{R}^{-1}$
(f) $dom\left(\mathbb{R}^{-1}\right) = cod(\mathbb{R})$
(g) $cod\left(\mathbb{R}^{-1}\right) = dom(\mathbb{R})$
(h) $dom(\mathbb{P} \circ \mathbb{R}) \subseteq dom(\mathbb{P})$
(i) $cod(\mathbb{P} \circ \mathbb{R}) \subseteq cod(\mathbb{R})$
(j) $cod(\mathbb{P}) \cap dom(\mathbb{R}) = \varnothing \Leftrightarrow \mathbb{P} \circ \mathbb{R} = \varnothing$
(k) $dom(\mathbb{R}) = cod(\mathbb{R}) \Rightarrow dom(\mathbb{R}) = dom(\mathbb{R}^+) = cod(\mathbb{R}^+)$

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4_10

(l) $id_{\mathbb{X}}^{-1} = id_{\mathbb{X}}$
(m) $\varnothing^{-1} = \varnothing$
(n) $id_{\mathbb{X}} \circ \mathbb{R} = \mathbb{R} \circ id_{\mathbb{X}} = \mathbb{R}$
(o) $\varnothing \circ \mathbb{R} = \mathbb{R} \circ \varnothing = \varnothing$.

10.2 Infinite Series of Real Numbers

For a given infinite sequence of real numbers $a_0, a_1, a_2, \ldots, a_n, \ldots$ let us define a sequence called a partial sums sequence:

$s_0 = a_0$, $s_{n_{+1}} = s_n + a_{n+1}$, for all $n \geq 0$. The infinite sequence $s_0, s_1, s_2, \ldots, s_n \ldots$ is denoted by the symbol $\sum_{n=0}^{\infty} a_n$ and is called an *infinite series* of elements $a_0, a_1, a_2, \ldots, a_n, \ldots$

The series *is convergent to a sum s* iff $\lim_{n\to\infty} s_n = s$, which is written $\sum_{n=0}^{\infty} a_n = s$.

The series is *divergent* iff the limit of the sequence $s_0, s_1, s_2, \ldots, s_n \ldots$ does not exist. This series is *absolutely convergent* iff the series $\sum_{n=0}^{\infty} |a_n|$ is convergent. The necessary condition of convergence of the series is $\lim_{n\to\infty} a_n = 0$. That is: if $\sum_{n=0}^{\infty} a_n$ is convergent then $\lim_{n\to\infty} a_n = 0$. This is not sufficient condition, for instance, the harmonic series $\sum_{n=0}^{\infty} \frac{1}{n}$ is divergent although $\lim_{n\to\infty} \frac{1}{n} = 0$.

10.2.1 D'Alambert (Sufficient) Criterion of Convergence

The series of real numbers $\sum_{n=0}^{\infty} a_n$ where $a_n > 0$, is convergent if $\lim_{k\to\infty} \frac{a_{k+1}}{a_k} < 1$ and divergent if $\lim_{k\to\infty} \frac{a_{k+1}}{a_k} > 1$. If $\lim_{k\to\infty} \frac{a_{k+1}}{a_k} = 1$ then the series may be either convergent or divergent.

An easy proof of this criterion may be find in most books, where the subject concerning series is included.

10.2.2 Mertens Theorem on Multiplication of Series

If the series $\sum_{n=0}^{\infty} a_n$ and $\sum_{n=0}^{\infty} b_n$ are absolutely convergent then their Cauchy product $\sum_{n=0}^{\infty} c_n$ where

$$c_n = \sum_{k=0}^{n} a_k b_{n-k} = a_0 b_n + a_1 b_{n-1} + a_2 b_{n-2} + \cdots + a_{n-1} b_1 + a_n b_0$$

is convergent and $\sum_{n=0}^{\infty} c_n = \left(\sum_{n=0}^{\infty} a_n\right)\left(\sum_{n=0}^{\infty} b_n\right)$

Proof

$$\begin{array}{lllll} & & & \text{column 0} & \\ & & & \downarrow & \\ \text{for } n = 0: & c_0 & = & a_0 b_0 & \text{column 1} \\ & & & \downarrow & \\ \text{for } n = 1: & c_1 & = & a_0 b_1 + a_1 b_0 & \text{column 2} \\ & & & \downarrow & \\ \text{for } n = 2: & c_2 & = & a_0 b_2 + a_1 b_1 + a_2 b_0 & \text{column 3} \\ & & & \downarrow & \\ \text{for } n = 3: & c_3 & = & a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 & \end{array}$$

……..

etc.

First, adding up all elements of each infinite column and taking all a_i in front of brackets, then adding up all of these products, we get:

$$a_0(b_0 + b_1 + b_2 + \cdots) + a_1(b_0 + b_1 + b_2 + \cdots) + a_2(b_0 + b_1 + b_2 + \cdots) + \cdots =$$
$$(a_0 + a_1 + a_2 + \cdots)(b_0 + b_1 + b_2 + \cdots) = \left(\sum_{n=0}^{\infty} a_n\right)\left(\sum_{n=0}^{\infty} b_n\right)$$

□

Remarks

1. Why do the „horizontal" and „vertical" infinite summations yield the same value? This follows from a theorem that the order of adding elements of absolutely convergent series, has no influence on its value (the infinite sum).
2. The original formulation of the Mertens theorem contains somewhat weaker assumption: it suffices one of the series be absolutely convergent. However the above proof, for its straightforwardness, justifies slightly stronger assumption.

Example If $\sum_{n=0}^{\infty} a_n = \sum_{n=0}^{\infty} b_n = \sum_{n=0}^{\infty} x^n$ then $\left(\sum_{n=0}^{\infty} a_n\right)\left(\sum_{n=0}^{\infty} b_n\right) = \sum_{n=0}^{\infty} (n+1)x^n$ because

$$c_n = x^0 x^n + x^1 x^{n-1} + x^2 x^{n-2} + \cdots + x^{n-1} x^1 + x^n x^0 = (n+1)x^n$$

therefore $\sum_{n=0}^{\infty} c_n = \sum_{n=0}^{\infty} (n+1)x^n$. By virtue of the Mertens theorem:

$$\sum_{n=0}^{\infty}(n+1)x^n = \left(\sum_{n=0}^{\infty}x^n\right)\left(\sum_{n=0}^{\infty}x^n\right)$$

If $|x| < 1$ then $\sum_{n=0}^{\infty}x^n = \frac{1}{1-x}$ thus $\sum_{n=0}^{\infty}(n+1)x^n = \left(\sum_{n=0}^{\infty}x^n\right)^2 = \left(\frac{1}{1-x}\right)^2$

10.3 Probability, Independent Events, Random Variable and Its Expected Value

Let Ω be a set of m possible results of a certain experiment and let A be a subset of Ω, called an event and containing $k \leq m$ results of this experiment. The event A is said to occur if and only if the experiment yields a result that belongs to A. The classic concept of probability (introduced by Laplace in 1812) as a chance of event A to occur in this experiment, is the quotient $\frac{k}{m}$. Such understanding of probability is justified by the following common everyday observation. If in the n-times repeated experiment (under unchanged conditions), event A occurred $f_n(A)$-times, then the frequency $\frac{f_n(A)}{n}$ of occurrences of A gets nearer to $\frac{k}{m}$, the greater is number n. Hence the limit $\lim_{n\to\infty}\frac{f_n(A)}{n}$ has also been taken as a measure of the chance of A to occur (Mises 1931). Since both definitions are mathematically obscure (referring to vague terms, methodological incorrect assumptions and infinite repetition of experiment), the axiomatic, precise formal definition is nowadays adopted (introduced by Kolmogorow in 1933). It may be expressed as follows. Let Ω be a set of elements called the *elementary events* of a certain experiment. The elements will be denoted by ω with subscripts possibly. As in the classic case, a subset of Ω is called *an event* and we say that the event $A \subseteq \Omega$ *occurs*, if a certain element $\omega \in A$ has been chosen in effect of a random choice of elements from Ω (note that this pronouncement is merely a comment supporting intuition, not a formal definition). The entire set Ω is the *certain event* (certainty), the empty set $\varnothing$ is the *impossible event*. Here we confine ourselves to the enumerable (in particular finite) sets Ω and define a *probability* of events as a function assigning to each event A a real number $P(A)$ satisfying the following conditions:

(1) $0 \leq P(A) \leq 1$
(2) $P(\Omega) = 1$
(3) $P(A_1 \cup A_2 \cup \ldots) = P(A_1) + P(A_2) + \cdots$ for arbitrary finite or infinite sequence of events

$A_1, A_2, \ldots$ pairwise disjoint, i.e. such that $A_i \cap A_j = \varnothing$ for each i and j with $i \neq j$. The pair (Ω, P) is called a *probabilistic space.*

Remarks

(a) $P(\varnothing) = 0$, which follows immediately from (2) and (3).
(b) If the set Ω were non-enumerable, then not to all of its subsets could be assigned a probability which satisfies (1), (2), (3), but only to subsets belonging to the so-called σ-family $\mathfrak{M}$ of sets, such that $\Omega \in \mathfrak{M}$, if $A, B \in \mathfrak{M}$ then $A - B \in \mathfrak{M}$ and $A_1 \cup A_2 \cup \ldots \in \mathfrak{M}$, for countably many sets A_1, A_2, … belonging to $\mathfrak{M}$. In this case, the probabilistic space is $(\Omega, \mathfrak{M}, P)$. Obviously, for the enumerable sets Ω, the σ-family $\mathfrak{M}$ is the powerset of Ω, i.e. the family of all its subsets. For some considerations, apart from infinite union of events, their infinite intersection $A_1 \cap A_2 \cap \ldots$ is required to belong to $\mathfrak{M}$ (called then the $\sigma\delta$-family).
(c) Evidently, the classically defined probability as the quotient $\frac{k}{m}$, satisfies axioms (1), (2), (3). The axiomatic definition is a definitional schema. So, it does not enable calculation of a concrete probability, but imposes conditions to be fulfilled by probability of a given event in a concrete experiment.
(d) The events are sets, thus the calculus of sets and combinatorial analysis (for finite events) is applicable to probability theory.

Two events A_1, A_2 in the probabilistic space (Ω, P) are called *independent* iff $P(A_1 \cap A_2) = P(A_1) \cdot P(A_2)$. Generalization onto a finite set of events $\mathfrak{A} = \{A_1, A_2, \ldots, A_n\}$ may be obtained as follows. Events belonging to $\mathfrak{A}$ are *independent* if for every subset $\boldsymbol{S}$ of $\mathfrak{A}$ the equality $P\left(\bigcap_{A\in\boldsymbol{S}} A\right) = \prod_{A\in\boldsymbol{S}} P(A)$ holds, where $\bigcap_{A\in\boldsymbol{S}} A$ is the intersection of all events from $\boldsymbol{S}$ and $\prod_{A\in\boldsymbol{S}} P(A)$ is the product of probabilities of all events from $\boldsymbol{S}$.

Random variable is any function $X : \Omega \xrightarrow{into} R$; (injective function). This is a function „into the set $\boldsymbol{R}$", but not „onto the set $\boldsymbol{R}$", that is, not the whole set $\boldsymbol{R}$ is its range, but a certain proper subset: $\boldsymbol{S}_X \subset \boldsymbol{R}$: X: $\Omega \xrightarrow{onto} \boldsymbol{S}_X$ (surjective function). In general, the range $\boldsymbol{S}_X$ must fulfil some conditions assumed in Chap. 7, where the set Ω was naturally enumerable. The random variable is ***discrete*** if $\boldsymbol{S}_X$ is finite or infinite but enumerable.

Expected value of discrete random variable X is a number $E(X)$, defined as

$E(X) = x_1p_1 + x_2p_2 + x_3p_3 + \cdots = \sum_i x_ip_i$, where summation expands onto all values $x_1, x_2, x_3, \ldots$ of random variable X and p_i is a probability that X assumes value x_i; this probability is denoted by $P(X = x_i) = p_i$. If there are infinitely many of addends in this sum then we assume that the series is absolutely convergent: $\sum_i |x_ip_i| < \infty$.

Example In performing of n tests, $n_i (i = 1, 2, \ldots, k)$ times a result x_i has been obtained, thus $n_1 + n_2 + \cdots + n_k = n$, $\mathbf{S}_X = \{x_1, x_2, \ldots, x_k\}$ (range of random

value X). Then $P(X = x_i) = p_i = \frac{n_i}{n}$ (classic definition of probability as frequency), $E(X) = \frac{n_1}{n}x_1 + \frac{n_2}{n}x_2 + \cdots + \frac{n_k}{n}x_k = \frac{n_1x_1 + n_2x_2 + \cdots + n_kx_k}{n}$.

Thus, this is a weighted average of values $x_1, x_2, \ldots, x_k$ with weights $\frac{n_1}{n}, \frac{n_2}{n}, \ldots, \frac{n_k}{n}$.

If $n_1 = n_2 = \cdots = n_k = 1$, thus when $n = k$ then the expected value $E(X)$ is the arithmetic average of $x_1, x_2, \ldots, x_k$.

10.3.1 Basic Notions of Cause-Effect Structures

Let **X** be a non-empty enumerable set. Its elements, called nodes, are counterparts of places in Petri nets [Petri 1962], (Reisig 1985). Let $\theta \notin \mathbf{X}$ be a symbol called neutral. It will play part of neutral element for operations on formal polynomials (terms). The nodes, symbol θ, operators +, ·, called the addition and multiplication respectively, and parentheses are symbols out of which polynomials are formed as follows. Each node and symbol θ is a polynomial; if K and L are polynomials then $(K + L)$ and $(K \cdot L)$ are too; no other polynomials exist. Let us say "polynomials over **X**". Their set is denoted by **F[X]**. Assume stronger binding of · than +; this allows for dropping some parentheses. Addition and multiplication of polynomials is defined as follows: $K \oplus L = (K + L)$, $K \otimes L = (K \cdot L)$. Let us use + and · instead of ⊕ and ⊗ . It is required that the system $\langle \mathbf{F}[\mathbf{X}], +, \cdot, \theta \rangle$ obeys the following equality axioms for all $K, L, M \in \mathbf{F[X]}$, $x \in \mathbf{X}$:

$(+)$	$\theta + K = K + \theta = K$	$(\cdot)$	$\theta \cdot K = K \cdot \theta = K$
$(++)$	$K + K = K$	$(\cdot\cdot)$	$x \cdot x = x$
$(+++)$	$K + L = L + K$	$(\cdot\cdot\cdot)$	$K \cdot L = L \cdot K$
$(++++)$	$K + (L + M) = (K + L) + M$	$(\cdot\cdot\cdot\cdot)$	$K \cdot (L \cdot M) = (K \cdot L) \cdot M$
$(+\cdot)$	If $L \neq \theta \Leftrightarrow M \neq \theta$ then $K \cdot (L + M) = K \cdot L + K \cdot M$		

Algebraic system which obeys these axioms will be referred to as a *near semi ring of formal polynomials.*

A *cause-effect structure* (c/e structure) over **X** is a pair $U = (C, E)$ of functions:

C: $\mathbf{X} \to \mathbf{F[X]}$ (the cause function; nodes occurring in $C(x)$ are *causes* of x)

E: $\mathbf{X} \to \mathbf{F[X]}$ (the effect function; nodes occurring in E(x) are *effects* of x)

such that x occurs in the polynomial $C(y)$ iff y occurs in $E(x)$. A carrier of U is the set $car(U) = \{x \in \mathbf{X} | C(x) \neq \theta \vee E(x) \neq \theta\}$. U is finite iff $|car(U)| < \infty$ (|…| means cardinality). The set of all c/e structures over **X** is denoted by **CE[X]**. Since **X** is a fixed set, we write simply **CE**—wherever this makes no confusion.

A representation of a c/e structure $U = (C, E)$ as a set of annotated nodes is $\left\{ x_{E(x)}^{C(x)} \middle| x \in car(U) \right\}$. U is also presented as a directed graph with $car(U)$ as set of nodes labelled with objects of the form $x_{E(x)}^{C(x)} (x \in car(U))$ and there is an edge

(arrow) from x to y iff y occurs in the polynomial $E(x)$. Note that in this representation, edges, although useful for the appearance of system models, are redundant: interconnection of nodes may be inferred from polynomials $C(x)$, $E(x)$. Since C,E are total functions (defined on the entire set **X**), any c/e structure comprises all the nodes from **X**, also the isolated ones (with $C(x) = E(x) = \theta$), invisible in the graphical representation. The isolated nodes make the distributivity law $(+\ \cdot)$ to be conditional.

Addition and multiplication of c/e structures

For c/e structures $U = (C_U, E_U)$, $V = (C_V, E_V)$ define:

$U + V = (C_{U+V}, E_{U+V}) = (C_U + C_V, E_U + E_V)$ where

$(C_U + C_V)(x) = C_U(x) + C_V(x)$ and $(E_U + E_V)(x) = E_U(x) + E_V(x)$

$$U \cdot V = (C_{U\cdot V}, E_{U\cdot V}) = (C_U \cdot C_V, E_U \cdot E_V) \text{ where}$$

$$(C_U \cdot C_V)(x) = C_U(x) \cdot C_V(x) \text{ and } (E_U \cdot E_V)(x) = E_U(x) \cdot E_V(x)$$

U is a *monomial* c/e structure if polynomials C_U(x) and $E_U(x)$ are a monomials, i.e. do not comprise " + ". C/e structure $\left\{x_{y}^{\theta}, y_{\theta}^{x}\right\}$ is an arrow, denoted as $x \to y$. The pair (θ, θ) is a c/e structure if θ is understood as a constant function $\theta(x) = \theta$ for each $x \in \mathbf{X}$. From definition of addition and multiplication of c/e structures follows that (θ, θ) is neutral for + and $\cdot$. For brevity let us write θ instead of (θ, θ).

Evidently, $U + V \in \mathbf{CE}$ and $U \cdot V \in \mathbf{CE}$ that is, in the resulting c/e structures, x occurs in $C_{U+V(y)}$ iff y occurs in $E_{U+V(x)}$ and the same for $U \cdot V$. Thus, addition and multiplication of c/e structures yield correct c/e structures. The algebraic system $\langle \mathbf{CE[X]}, +, \cdot, \theta \rangle$ is a near semi ring similar to $\langle \mathbf{F[X]}, +, \cdot, \theta \rangle$, as states the following fact:

Proposition *For all U, V, W* $\in$ **CE[X]**, *x, y* $\in$ **X** *the following properties hold in the algebraic system* $\langle \mathbf{CE}[\mathbf{X}], +, \cdot, \theta \rangle$:

$(+)$	$\theta + U = U + \theta = U$	$(\cdot)$	$\theta \cdot U = U \cdot \theta = U$
$(++)$	$U + U = U$	$(\cdot\cdot)$	$(x \to y) \cdot (x \to y) = x \to y$
$(+++)$	$U + V = V + U$	$(\cdot\cdot\cdot)$	$U \cdot V = V \cdot U$
$(++++)$	$U + (V + W) = (U + V) + W$	$(\cdot\cdot\cdot\cdot)$	$U \cdot (V \cdot W) = (U \cdot V) \cdot W$
$(+\cdot)$	If $C_V(x) \neq \theta \Leftrightarrow C_W(x) \neq \theta$ and $E_V(x) \neq \theta \Leftrightarrow E_W(x) \neq \theta$ then $U \cdot (V + W) = U \cdot V + U \cdot W$		

The equations follow directly from definition of c/e structures and definitions of adding and multiplying c/e structures.

Notice that the operations on c/e structures make possible to combine small c/e structures into large parallel system models.

For $U \in \mathbf{CE}$, define a partial order in **CE** by $U \leq V \Leftrightarrow V = U + V$. If $U \leq V$ then U is a substructure of V; **SUB[*V*]** = $\{U \mid U \leq V\}$ is the set of all substructures of V. For $A \subseteq \mathbf{CE}$:

$V \in A$ is minimal (w.r.t. $\leq$) in A iff $\forall W \in$ A: $(W \leq V \Rightarrow W = V)$.

The crucial notion for behaviour of c/e structures is *firing component*, a counterpart of transition in Petri nets. It is, however, not a primitive notion but derived from the definition of c/e structures, and is introduced regardless of any particular c/e structure:

A minimal in $\mathbf{CE}\backslash\{\theta\}$ c/e structure $Q = (C_Q, E_Q)$ is a ***firing component*** iff Q is a monomial c/e structure and $C_Q(x) = \theta \Leftrightarrow E_Q(x) \neq \theta$ for any $x \in car(Q)$. The set of all firing components is denoted by **FC**, thus the set of all firing components of $U \in \mathbf{CE}$ is $\mathbf{FC}[U] = \mathbf{SUB}[U] \cap \mathbf{FC}$.

Following the standard Petri nets notation, let for $Q \in \mathbf{FC}$:

$$\begin{aligned} {}^\bullet Q &= \{x \in car(Q) | C_Q(x) = \theta\} \quad (\text{pre} - \text{set of } Q) \\ Q^\bullet &= \{x \in car(Q) | E_Q(x) = \theta\} \quad (\text{post} - \text{set of } Q) \end{aligned}$$

The state of c/e structure is a counterpart of marking in 1-safe (elementary) Petri nets. Note however that it is not bound up to any c/e structure:

A ***state*** is a subset of the set of nodes: $s \subseteq \mathbf{X}$. The set of all states: $\mathbf{S} = 2^{\mathbf{X}}$, the powerset of **X**. A node x is *active* in the state s if and only if $x \in s$ and *passive* otherwise. After Petri nets phrasing we say "x holds a token" when x is active.

Semantics of c/e structures is a counterpart of simple the firing rule in 1-safe Petri nets:

For $Q \in \mathbf{FC}[U]$ and $s, t \in \mathbf{S}$, let $[\![Q]\!] \subseteq \mathbf{S} \times \mathbf{S}$ be a binary relation defined as: $(s, t) \in [\![Q]\!]$ iff ${}^\bullet Q \subseteq s$ and $Q^\bullet \cap s = \varnothing$ and $t = (s \backslash {}^\bullet Q) \cup Q^\bullet$

(say: Q transforms state s into t). Semantics $[[U]]$ of $U \in \mathbf{CE}$ is: $[[U]] = \bigcup_{Q \in \mathbf{FC}[U]} [[Q]]$

$[\![U]\!]^*$ is its reflexive and transitive closure, that is, $(s, t) \in [\![U]\!]^*$ iff $s = t$ or there exists a sequence of states $s_0, s_1, \ldots, s_n$ with $s = s_0$, $t = s_n$ and $(s_j, s_{j+1}) \in [\![U]\!]$ for $j = 0, 1, \ldots, n - 1$. We say that t is reachable from s in semantics $[\![\cdot]\!]$. The sequence $s_0, s_1, \ldots, s_n$ is called a *computation* in U.

Note that $[\![U]\!]^* = \varnothing$ iff $\mathbf{FC}[U] = \varnothing$. Behaviour of c/e structures in accordance with this semantics may be imagined as a token game: if each node in a certain firing component's pre-set holds a token and none in its post-set does, then remove tokens from the pre-set and put them in the post-set.

A few immediate conclusions of above definitions are:

1. $U_1 \leq V_1 \wedge U_2 \leq V_2 \Rightarrow U_1 + U_2 \leq V_1 + V_2$ (monotonicity of +)
2. $U_1 \leq V_1 \wedge U_2 \leq V_2 \Rightarrow U_1 \cdot U_2 \leq V_1 \cdot V_2$ provided that $(U_1 + V_1) \cdot (U_2 + V_2) = U_1 \cdot U_2 + V_1 \cdot V_2 + U_1 \cdot V_2 + V_1 \cdot U_2$ (conditional monotonicity of $\cdot$)
3. $U \cdot (V + W) \leq U \cdot V + U \cdot W$ but relation $\leq$ not always may be replaced by equality
4. If $U \cdot (V + W) = U \cdot V + U \cdot W$ then $V \leq W \Rightarrow U \cdot V \leq U \cdot W$
5. $U \leq V \Rightarrow \mathbf{FC}[U] \subseteq \mathbf{FC}[V]$ but converse implication not always holds
6. $\mathbf{FC}[U] \cup \mathbf{FC}[V] \subseteq \mathbf{FC}[U + V]$ but the inclusion not always may be replaced by equality

7. $\langle \mathbf{CE}, \leq \rangle$, i.e. the set of all c/e structures partially ordered by relation $\leq$ is a non-distributive lattice with the least element θ and with no greatest element.
8. $\mathbf{FC}[U] \subseteq \mathbf{FC}[V] \Rightarrow [\![U]\!] \subseteq [\![V]\!]$ but converse implication not always holds
9. $[\![U]\!] \cup [\![V]\!] \subseteq [\![U+V]\!]$ but the inclusion not always may be replaced by equality
10. $\mathbf{FC}[U] \cup \mathbf{FC}[V] = \mathbf{FC}[U+V] \Rightarrow [\![U]\!] \cup [\![V]\!] = [\![U+V]\!]$ but converse implication not always holds. Note that equation $[\![U]\!] \cup [\![V]\!] = [\![U+V]\!]$ expresses compositionality of summation for c/e structures U and V; equation $\mathbf{FC}[U] \cup \mathbf{FC}[V] = \mathbf{FC}[U+V]$ states that no new firing components (except for those in U and V) are created in their sum.

Remark The definition of c/e structures along with some basic facts on them, have been presented here for better understanding the ABP protocol, whose activity, as a token game in a c/e structure specifying the protocol, is given in Table 9.1 in Chap. 9. For this purpose it was sufficient to present elementary c/e structures (counterparts of elementary Petri nets, i.e. with at most one token in any place). More general versions, like place/transitions or colour nets were investigated e.g. in Ustimenko (1996, 1998). Equivalence of c/e structures and Petri nets has been proved in Raczunas (1993). The definition of c/e structures quoted in this chapter, comes from Czaja (1998, 2002). In the first paper on the subject (Czaja 1988), an equivalent definition but based on fix-point, has been adopted.

References

Czaja, L. (January 1988). Cause-effect structures. *Information Processing Letters, 26,* 313–319.

Czaja, L. (2002). *Elementary cause-effect structures*. Wydawnictwa: Uniwersytetu Warszawskiego.

Czaja, L. (1998). Cause-effect structures—structural and semantic properties revisited. *Fundamenta Informaticae* 33, 17–42, IOS Press.

Petri, C. A. (1962). Kommunikation mit Automaten. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

Raczunas, M. (1993). Remarks on the equivalence of c-e structures and Petri nets. *Information Processing Letters, 45,* 165–169.

Reisig, W. (1985). *Petri nets: An introduction, monographs on theoretical computer science*. Berlin: Springer.

Ustimenko, A. P. (1996). Algebra of two-level cause-effect structures. *Information Processing Letters, 59,* 325–330.

Ustimenko, A. P. (1998). Coloured cause-effect structures. *Information Processing Letters, 68*(5), 219–225.

Final Remarks

The book concerns main principles and features of distributed systems, composed of computers internally controlled, that is, the so-called von Neuman's architectures (von Neumann 1945). Presenting basic idea behind activity of such machine in Chap. 1, details of its physical construction have been omitted. All the nowadays realizations of the instruction execution cycle are electronic, but one may imagine, for instance, mechanical realization, involving cogwheels, levers, etc., with program and data supplied on punched cards as memory—as in analytical machine built by Babbage (1864). The principle is similar, but execution duration of arithmetic and other operations in case of present-day electronic processors is several billions times shorter. Along with physical realizations, the von Neumann's machine is the human concept. However this is not the only principle of activity of devices performing computation operations—arithmetical, logical and control. Inspiration for searching for different principles, called computational paradigms, are natural phenomena. Nowadays, the intensive research is being carried out on mechanisms of biological particles activity, in particular information flow among molecular structures and on making usage of their multitude for mass of simple operations, executed in parallel. Some computational architectures have already been created, that operate on such principle, though not yet carried into widespread, public usage. Such constructs are capable of solving tasks exceeding (owing to time complexity) capability of classical sequential processors, even interconnected into distributed systems. Computations in such experimental architectures, variously named: molecular, biomolecular, biochemical or DNA computing, have so far efficiently applied to solving some combinatorial problems, the so-called computationally hard, like finding Hamiltonian cycle in a graph (closed path crossing every vertex exactly ones), verification of satisfiability of propositional formuli of very many variables, modelling of the so-called self-organizing systems (e.g. evolution of organisms, weather phenomena, etc.) and many more problems of high complexity. Another natural phenomenon inspiring non classic work principle of computational devices comes from quantum mechanics. The information unit is there the so-called quantum bit, *qubit* in short, mathematically represented as a linear combination of two states of polarization of a quantum object, e.g. photon, where the coefficients of the combination are the so-called amplitudes of probability

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4

of a certain state occurrence. Thus, the qubit is, in a sense, a superposition (or composition) of two binary states, whose sequence may be simultaneously in many states, due to the so-called quantum entanglement. This phenomenon has been used for simultaneous execution of mass computations. Each computation is a sequence of quantum states and a state is a sequence of n qubits, where n is their number which the so-called quantum computer can contain. The computer may be **simultaneously** in 2^n states, whereas its classic, sequential counterpart—in one only from among 2^n states **at a time**. The quantum architecture (Deutsch 1985), still in experimental phase, alike the molecular one, is capable of overcoming the complexity barriers of classic processors, also in distributed systems. Typical problems of high complexity come from number theory, especially their application in cryptography. Both aforesaid non classic paradigms of computation inspired by natural phenomena, along with usage of natural objects—biochemical and quantum—in their physical realization (hardware), are expected to be applied in the massively parallel computing. The principles belong to a research domain called natural computation. This domain encompasses also imitation of phenomena and processes encountered in the nature as problem solving techniques, but by creating algorithmic models executed in the classic computer system, or in grids of many simple and cheap sequential processors. Examples are neural computation, cellular automata, swarm intelligence (imitating behaviour of a "swarm" of objects striving to a joint objective), evolutionary computation (imitating Darwinian evolution), membrane computing (imitating of cell membrane functions), cf. Rozenberg et al. (2012). These and other models of computation—a result of observation of natural phenomena—are being devised for special tasks, but there are also research attempts to create the Universal Quantum Turing Machine (Turing 1937; Deutsch 1985; Penrose 1989), a system for all the effective computations, like a classic computer with arbitrarily large memory needed for a given class of problems. One may also mention other systems, where parallelization and distribution of great number of cooperating elements takes place, like the cyber-physical systems (Lamnabhi-Lagarrigue et al. 2014) (for control of technical devices) or multi agent systems (Russell and Norvig 2010). All them are dynamically developing scientific and engineering explorations. This book was concerned with distributed systems composed of computers of the classic von Neuman's architecture, leaving to the interested reader a closer familiarity with endeavours leading to practical application of the above mentioned contemporary computational principles.

References

Babbage, C. (1864). *The life of a philosopher*. London.

Deutsch, D. (1985). Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London, A400*, 97–117.

Lamnabhi-Lagarrigue, F., Di Benedetto, M. D., & Schoitsch, E. (2014). Introduction to the special theme cyber-physical systems. *Ercim News, 94*, 6–7.

Penrose, R. (1989). *The emperor's new mind*. UK: Oxford University Press.

Rozenberg, G., Baeck, T., & Kok, J. (2012). *Handbook of natural computing*. Heidelberg.

Russell, S., & Norvig, P. (2010). *Artificial intelligence. A modern approach*. Englewood Cliffs, NJ: Prentice Hall.

Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society Series 2, 42*, 230–265.

von Neumann, J. (1945). *First draft of a report on the EDVAC*.

Bibliography

Akl, S. G. (1997). *Parallel computation: Models and methods*. USA: Prentice Hall.

Bedrouni, A., Mittu, R., Abdeslem Boukhtouta, A., & Berger, J. (2009). *Distributed intelligent systems. A coordination perspective*. Berlin: Springer.

Belapurkar, A., Chakrabarti, A., Ponnapalli, H., Varadarajan, N., Padmanabhuni, S., & Sundarrajan, S. (2009). *Distributed systems security. Issues, processes and solutions*. USA: Wiley.

Ben-Ari, M. (1996). *Podstawy programowania współbieżnego i rozproszonego*. Warszawie: Wydawnictwa Naukowo-Techniczne 1996 (Polish translation of 1990).

Boykin, J., Kirschen, D., Langerman, A., & Loverso, S. (1993). *Programming under Mach*. Reading, MA: Addison-Wesley.

Broy, M., & Stølen, K. (2001). *Abracadabra protocol*. In: *Specification and development of interactive systems. Monograps in Computer Science*. NY: Springer.

Comer, D. E. (1995). *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture*. USA: Prentice Hall.

Coulouris, G., Dollimore, J., & Kindberg, T. (1998). *Systemy rozproszone, podstawy i projektowanie*. Warszawa: WNT (Polish translation of 1994).

Czaja, L. (1973). Niektóre aspekty implementacji ALGOL-u 60 dla maszyn ZAM/21 ALFA. In *Materiały Sympozjum XV-rocznicy Instytutu Maszyn Matematycznych i Roku Nauki Polskiej*, Warszawa (Some aspects of implementation of ALGOL 60, in Polish).

Czaja, L., & Szorc, P. (1967). Implementation of ALGOL for ZAM computers. *Algorytmy, 4*(7), 91–111.

Dahl, O.-J., Dijkstra, E. W., & Hoare, C. A. R. (1972). *Structured programming*. London and New York: Academic Press.

Dasgupta, P., LeBlanc, R. J., Jr., Ahamad, M., & Ramachandran, U. (1991). The clouds distributed operating systems. *IEEE Computer, 24*(11), 34–44.

Davies, D. W., Holler, E., Jensen, E. D., Kimbleton, S. R., Lampson, B. W., LeLann, G., et al. (1983). In B. W. Lampson, M. Paul, & H. J. Siegert (Eds.), *Distributed systems—Architecture and implementation, an advanced course*. Berlin: Springer.

Gabassi, M., & Dupouy, B. (1995). *Przetwarzanie rozproszone w systemie UNIX*. Warszawa: Lupus (Distributed processing in UNIX, in Polish).

Gien, M. (1995). Evolution of the CHORUS open microkernel architecture: The STREAM project. In *FTDCS '95 Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*. USA: IEEE Computer Society.

Gościński, A. (1991). *Distributed operating systems, the logical design*. USA: Addison Wesley.

Haddad, S., Kordon, F., Pautet, L., & Petrucci, L. (2011). *Distributed systems, design and algorithms*. USA: Wiley.

Holenderski, L., & Szałas, A. (1988). Propositional description of finite cause-effect structures. *Information Processing Letters, 27,* 111–117.

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4

Kshemkalyani, A. D., & Singhal, M. (2011). *Distributed computing: Principles, algorithms, and systems*. Cambridge: Cambridge University Press.

Milner, R. (1989). *Communication and concurrency*. In C. A. R. Hoare (Series Ed.). USA: Prentice-Hall.

Shapiro, E. Y. (1987). *Concurrent PROLOG: Collected papers*. Cambridge MA, USA: MIT Press.

Sinha, P. K. (1997). *Distributed operating systems—Concepts and design*. Piscataway: IEEE Press.

Sorin, D. J., Hill, M. D., & Wood, D. A. (2011). A premier on memory consistency and cache coherence. In M. D. Hill (Ed.), *Synthesis lectures on computer architecture*. USA: Morgan and Calypool Publishers.

Spector, A., & Gifford, D. (1984). The space shuttle primary computer system. *Communications of the ACM 27*(i), 874–900.

Sportack, M. (1998). *Networking essentials unleashed* (1st ed.). USA: Sams Publishing.

Starke, P. (1987). *Sieci petri.* Warsaw: PWN (Polish translation of the author's book in German: *Petri Netze*, 1980).

Stevens, W. R. (1999). *UNIX Programowanie usług sieciowych*. Warszawa: WNT.

SYSTEM AUTOMATYCZNEGO KODOWANIA SAKO. (1961). *Prace Zakładu Aparatów Matematycznych PAN*. Warszawa: PAN (SAKO—A system of automatic coding, in Polish).

Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G. J., Mullender, S. J., Jansen, J., et al. (1990). Experience with the amoeba distributed operating system. *Communication of the ACM, 33*(12), 46–63.

Tanenbaum, A. S. (1997). *Rozproszone systemy operacyjne*. Warsaw: PWN (Polish translation of 1995).

Tanenbaum, A. S. (2004). *Sieci komputerowe*. Gliwice: Wydawnictwo Helion (Polish translation of 2003).

Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer networks*. USA: Prentice Hall.

Tanenbaum, A. S., & van Steen, M. (2002). *Distributed systems: Principles and paradigms* (2nd ed.) UK: Pearson.

Tanenbaum, A. S., & van Steen, M. (2003). *Computer networks*. USA: Prentice Hall.

Tanenbaum, A. S., & van Steen, M. (2006). *Systemy rozproszone, zasady i paradygmaty*. Warszawa: WNT (Polish translation of 2002).

Wierzbicki, A. (2010). *Trust and fairness in open, distributed systems*. Berlin: Springer.

Wirth, N. (1988). *Programming in modula-2* (4th ed.), *Texts and Monographs in Computer Science*. Berlin: Springer.

Wulf, W. A., & Bell, C. G. (1972). *C.mmp—A multi-mini processor*. In *Proceedings of AFIPS, FJCC* (Vol. 41, pp. 765–777). Montvale, N.J.: AFIPS-Press.

Index

A
Accumulator, 1, 3
ADA, 20, 125–127, 142, 143, 146
Algol, 142
Algorithm, 2, 8, 35–37, 43, 53, 79, 122, 162, 172–175, 177–179, 222, 223, 225
  bully, 172–174, 177
  ring, 79, 172, 177–179
Alternating Bit Protocol, 160, 227
ALU, 121, 122
Amoeba, 53, 125
Architecture, 1, 43, 50, 52, 59, 60, 62, 64, 120–123, 142, 188
  von Neuman, 1, 2
Arpanet, 49, 55
Asynchronous Transfer Mode (ATM), 120, 130, 133, 137, 138

B
Berkeley method, 90, 91
Binary relations, 87, 241
Binder, 145
BITNET, 49, 50
Blocking, 68, 69, 72, 74, 75, 127–129
Broadcast, 86, 91, 101, 120, 127, 130, 131, 166, 197, 205, 221–223
Busy waiting, 116
Byzantine generals, 165, 169

C
Causal broadcast, 223, 224
Causal consistency, 217, 218, 220, 223
Causal order, 217, 219, 220, 225
Centralized system, 51
c/e structures, 227, 228, 246–249
Channel, 4, 6, 12, 20, 35, 120, 122, 123, 127, 129, 163, 227–229
Checkpoint, 160
Chorus, 53, 125
Circuit switching, 49, 125
Clock, 3, 43, 52, 56, 85–91, 96–99, 102, 142, 195, 202, 229
Clouds, 53, 59
Common Object Request Broker Architecture (CORBA), 59, 144
Communicating Sequential Processes (CSP), 19, 125, 127
Communication, 2, 4, 12, 19, 20, 35–38, 42, 50–53, 56, 57, 59, 61, 63, 64, 74, 78, 90, 94, 96, 99, 116, 119–133, 137, 141, 142, 146, 157, 165, 166, 173, 189, 221
  asynchronous, 19, 37–39, 120, 121, 124, 126–131, 137
  connection oriented, 120
  connectionless, 120, 121, 127, 130, 131, 137, 138
  group, 19, 53, 94, 126, 127, 131, 132, 157, 165, 221
  synchronous, 19, 20, 35–38, 42, 120, 121, 124–128, 131, 137
Computer network, 2, 50, 188
Computer systems classification, 47
Concurrency, 56, 60, 61, 65, 66, 71, 74, 190, 195, 196, 200, 212, 220
Concurrent Pascal, 125, 143, 146
Concurrent Prolog, 125
Consensus, 164, 165, 172
Coordinator, 162, 166, 170, 172–175, 177–179
Cristian's method, 89–91, 94, 142
Critical section, 8, 11, 12, 67, 78, 111, 143
Crossbar, 61, 62

L. Czaja, *Introduction to Distributed Computer Systems*, Lecture Notes in Networks and Systems 27, https://doi.org/10.1007/978-3-319-72023-4

D
D'Alambert criterion, 159
Deadlock, 65, 70, 73–75, 119, 162, 215
Discrete random variable, 158, 245
Distributed Computing Environment (DCE), 53
Distributed Shared Memory (DSM), 60, 187–190, 194–196, 215–218, 220, 221
Distributed system, 1, 2, 49, 50, 52, 55, 56, 60, 74, 85, 87, 96, 108, 119, 131, 141–143, 145, 157, 161, 187–189, 221

E
Election, 91, 162, 170, 173–175, 177–179, 184
European Academicand Research Network (EARN), 50
Event, 65, 73, 74, 85–87, 96, 97, 99, 100, 102, 116, 124, 132, 161, 162, 212, 215, 216, 218, 223, 244, 245
Exceptions, 124, 141, 146
Expected value, 158, 159, 244–246

F
Failure, 55, 66, 78, 90, 101, 112, 146, 157–162, 173, 174, 177, 178
Fast-read algorithm, 222
Fast-write algorithm, 221, 222
Fault tolerance, 55, 56, 61, 157, 165, 172
FIFO, 38, 131–133, 203, 222, 223
Flynn's taxonomy, 47

G
Global timestamp, 100–102, 108, 109
Greatest common divisor, 20, 38

I
Independent events, 158, 244
Inhibitor arcs, 116, 117
Instruction execution cycle, 1–3, 43
Instruction register, 1, 3
Instruction set, 3, 19, 37
Interleaving, 4, 190, 191, 194, 195, 199, 200, 202, 206, 210, 212, 220
Internal control, 1
Internet, 49, 50, 54, 61, 86, 94, 131, 188

J
Java, 20, 59, 123, 125, 142, 143, 145, 146

L
LINDA, 37, 125, 126, 129
Linear order, 87, 96, 100
Lock, 66, 67, 70
Logarithmic switch, 62, 63
Logical clock, 87, 94, 97, 99, 100
Logical time, 94
LOGLAN, 125, 146

M
Mach, 53, 125
Marshalling, 120, 121, 123, 133, 141, 142, 189
Memory coherence, 199, 200, 205, 206, 208–210, 212–214
Memory consistency, 125, 187–190, 195–197, 199, 205, 212, 215, 217, 220, 221
  causal, 217, 219, 220, 225
  Pipelined Random Access Memory (PRAM), 219, 220, 225
  sequential, 195, 197, 199, 200, 202, 205, 212, 217, 225
  strict, 187, 195, 200, 217, 220
Mertens theorem, 159, 242–244
Meta-program, 2
Modula-2, 125, 127, 146
Multicast, 120, 126, 127, 131, 157
Multiprogramming, 4
Mutual exclusion, 8, 19, 65–67, 75, 78–80, 83, 100–102, 109, 112, 116, 120, 143, 144, 162, 215

N
NASK, 50
Network Time Protocol (NTP), 91, 94, 95
Non Uniform Memory Access (NORMA), 64
NO Remote Memory Access (NUMA), 62, 63

O
OCCAM, 20, 124–127
Openness, 56, 59, 61
OSI/RM, 120, 133, 134, 137, 138

P
Partial order, 96, 100, 196, 212, 247
Petri nets, 116, 190, 227, 246, 248, 249
Physical time, 85, 87, 89, 90
PLEARN, 50
Port, 121, 126, 127, 145
Precedence, 85, 94, 96, 99
Probability, 158, 159, 164, 196, 244–246
Process, 3, 5, 8, 12, 19, 42, 52, 65–67, 70, 75, 76, 78, 79, 96–102, 108, 120, 121, 124–129, 131, 132, 142, 143, 145, 161, 162, 167, 173, 174, 177, 195, 197, 199, 205, 212, 217–219, 221
Processor, 1–4, 43, 44, 54, 61, 63, 65, 85, 87, 96, 161, 197, 199, 205
Protective zone, 67, 78, 79, 101–103, 108–112, 116

Protocol, 52, 63, 94, 102, 108–113, 116, 120–123, 125, 131, 133, 137, 138, 144, 146, 227, 228, 249

R
Random variable, 159, 244, 245
Remote Method Invocation (RMI), 58, 127, 141, 143, 145
Remote Procedure Call (RPC), 53, 59, 127, 141–147, 187, 189
Resource, 1, 8, 11, 56–59, 66, 67, 69, 70, 75, 78, 102, 108, 119, 142–144

S
Scalability, 56, 61, 131
Semaphore, 8, 11, 12, 66, 67, 116, 120
Sequential machine, 2
Series of real numbers, 242
Socket, 53, 121, 126, 127, 145
Starvation, 65, 75, 112, 119, 162, 215
Stub, 144–146
Synchronization, 2, 19, 57, 58, 65, 74, 78, 87–91, 94, 95, 119, 121, 138, 142, 161, 162, 173, 189

T
Thread, 50, 124–126, 147
Time compensation, 94, 99, 100, 102
Timeout, 74, 128, 131, 138, 146
Timesharing, 7
Timestamp, 99, 100, 102, 108, 110, 112, 160
Timestamps vectors, 111
Token ring, 66, 78, 79, 82, 83
Transactions, 58, 65–67, 69–71, 73, 74, 87, 119, 143, 164
Transparency, 56, 60, 141, 145, 160, 200, 221
Two Army problem, 162, 164, 165

U
Uniform Memory Access (UMA), 62, 63
UNIX 4BSD, 53
Unlock, 67, 70, 72
Unmarshalling, 120, 123, 133, 141, 142

V
Vector systems, 43

W
Wait graph, 73, 74
Weak precedence, 96