Distributed Algorithms

An Intuitive Approach

Second edition

Wan Fokkink

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in LATEX by the author. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Fokkink, Wan, 1965- author
Title: Distributed algorithms: an intuitive approach / Wan Fokkink
Description: Second edition | Cambridge, MA: The MIT Press, [2018] | Includes bibliographical references and index
Identifiers: LCCN 2017031164 | ISBN 9780262037662 (hardcover: alk. paper)
Subjects: LCSH: Distributed algorithms–Textbooks
Classification: LCC QA76.58.F647 2018 | DDC 004/.36–dc23
LC record available at https://lccn.loc.gov/2017031164

d_r0

Contents

Solutions to exercises and LATEX source files of slides for a course based on this textbook are available for prospective lecturers at

http://mitpress.mit.edu/distalgorithms2ed.

Preface

This textbook is meant to be used in a course on distributed algorithms for senior-level undergraduate or graduate students in computer science or software engineering and as a quick reference for researchers in the field. On the one hand it focuses on fundamental and instructive algorithms and results in distributed computing, while on the other hand it aims to exhibit the versatility of distributed and parallel computer systems. The distributed algorithms treated in this book are largely "classics" that were selected mainly because they are instructive with regard to the algorithmic design of distributed systems or shed light on key issues in distributed computing and concurrent programming. The book also discusses topics that are the focus of exciting new developments, notably transactional memory, blockchains, and quantum cryptography.

There are two very different ways to structure a course on algorithms. One way is to discuss algorithms and their analysis in great detail. The advantage of this approach is that students may gain deep insight into the algorithms and at the same time acquire experience in mathematical reasoning regarding their correctness and performance. Another way is to discuss algorithms and their correctness in an informal manner, and let students get acquainted with an algorithm from different angles by means of examples and exercises, without a need to understand the intricacies of the corresponding model and its underlying assumptions. Mathematical argumentations, which can be a stumbling block for many students, are thus avoided. An additional advantage is that a large number of algorithms can be discussed within a relatively short time, providing students with many different views on, and solutions to, challenges in distributed computing. In 10 years of teaching about distributed algorithms, I have converged on the latter approach, most of all because the students in my lectures tend to have hands-on experience and practical interests with regard to distributed systems. As a result, the learning objective of my course has been algorithmic thought rather than proofs and logic.

This book provides a Cook's tour of distributed algorithms. This phrase, meaning a rapid but extensive survey, refers to Thomas Cook, the visionary tour operator (and not the great explorer James Cook). Accordingly, this book is intended to be a travel guide through the world of distributed algorithms. A notable difference from other books in this area is that it does not emphasize correctness proofs. Algorithms are explained by means of brief, informal descriptions, illuminating examples, and exercises. The exercises have been carefully selected to make students well acquainted with the intricacies of distributed algorithms. Proof sketches, arguing the correctness of an algorithm or explaining the idea behind a fundamental result, are presented at a rather superficial level.

A thorough correctness proof, of course, is important in order to understand an algorithm in full detail. My research area is automated correctness proofs of distributed algorithms and communication protocols, and I have written two textbooks devoted to this topic. In the current textbook, however, intuition prevails. Readers who want to get a more detailed description of distributed algorithms in this book are recommended to consult the original papers, mentioned in the bibliographical notes at the end of each chapter. Moreover, pseudocode descriptions of a considerable number of algorithms are provided as an appendix.

I gratefully acknowledge the support of Helle Hvid Hansen, Jeroen Ketema, and David van Moolenbroek with providing detailed solutions to the exercises in this book, and am thankful for suggested improvements by students regarding earlier versions of these lecture notes. Special thanks go to Stefan Vijzelaar for his feedback on the description of blockchains and to Gerard Tel for his useful comments over the years.

Wan Fokkink

Amsterdam, November 2017

1

Introduction

In this age of the Internet, wide and local area networks, and multicore laptops, the importance of distributed computing is evident. The fact that you have opened this book suggests that no further explanation is needed to convince you of this point. The majority of modern-day system designers, system programmers, and ICT support staff must have a good understanding of distributed and concurrent programming. This, however, is easier said than done. The main aim of this book is to provide students with an algorithmic frame of mind, so that they can recognize and solve fundamental problems in distributed computing.

The two main communication paradigms for distributed systems are message passing, in which nodes send messages to each other via channels, and shared memory, in which different execution threads can read and write to the same memory locations. Both communication paradigms are addressed in this book.

An algorithm is a step-by-step procedure to solve a particular problem on a computer. To become a skilled programmer, it is essential to have good insight into algorithms. Every computer science degree program offers one or more courses on basic algorithms, typically for searching and sorting, pattern recognition, and finding shortest paths in graphs. There, students learn how to detect such subproblems within their computer programs and solve them effectively. Moreover, they are trained to think algorithmically, to reason about the correctness of algorithms, and to perform a simple complexity analysis.

Distributed computing is very different from and considerably more complex than a uniprocessor setting, because executions at the different compute nodes in a distributed system are interleaved. When two such nodes can concurrently perform events, it cannot be predicted which of these events will happen first. This gives rise, for instance, to so-called race conditions: if two messages are traveling to the same node in the network, different behavior may occur depending on which of the messages reaches its destination first. Distributed systems are therefore inherently nondeterministic: running a system twice from the same initial configuration may yield different results. And the number of reachable configurations of a distributed system tends to grow exponentially relative to its number of nodes.

Another important distinction between distributed and uniprocessor computing is the fact that the nodes in a distributed system in general do not have up-to-date knowledge of the global configuration of the system. They are aware of their own local state, but not always of the local states at other nodes or the messages in transit. For example, termination detection becomes an issue. It must be determined that all nodes in the system have become passive, and even if this is the case, there may still be a message in transit that will make the receiving node active again.

This book offers a wide range of basic algorithms for key challenges in distributed systems, such as termination detection, routing in a distributed network, coping with a crash of a compute node, or letting the nodes in a distributed network together build a snapshot of a global system configuration. It offers a bird’s-eye view on such pivotal issues from many different angles, as well as hand-waving correctness arguments and back-of-the-envelope complexity calculations.

2

Preliminaries

We start with some general mathematical notions and notations in section 2.1. Sections 2.2 and 2.3 contain preliminaries that are specific to the two different communication frameworks for distributed systems: message passing and shared memory.

2.1 Mathematical Notions

This section presents some basic definitions regarding sets, complexity measures for algorithms, and special operations on numbers.

Sets and Orders

As usual, $S_1 \cup S_2$, $S_1 \cap S_2$, $S_1 \setminus S_2$, and $S_1 \subseteq S_2$ denote set union, intersection, difference, and inclusion, respectively; $s \in S$ means that s is an element of the set S. Equality is expressed by $s_1 = s_2$; it holds if s_1 and s_2 denote the same element. The sets of natural and real numbers are denoted by $\mathbb{N}$ and $\mathbb{R}$. The Booleans consist of the elements *true* and *false*. A set may be written as $\{\cdots \mid \cdots\}$, where to the left of $\mid$ the elements in the set are described, and to the right the property that they should satisfy is identified. For example, $\{n \in \mathbb{N} \mid n > 5\}$ represents the set of natural numbers greater than 5. The empty set is denoted by $\emptyset$. For any finite set S, $|S|$ denotes its number of elements.

A (strict) order on a set S is a binary relation $<$ between elements of S that is irreflexive, asymmetric, and transitive, meaning that for all $s, s', s'' \in S$: $s < s$ does not hold; if $s < s'$, then $s' < s$ does not hold; and if $s < s'$ and $s' < s''$, then $s < s''$. An order is *total* if for any distinct $s, s' \in S$ either $s < s'$ or $s' < s$; otherwise the order is called *partial*. Given two sets S_1 and S_2 with orders $<_1$ and $<_2$, respectively, the *lexicographical order* $<$ on pairs from $S_1 \times S_2$ is defined by $(s_1, s_2) < (s_1', s_2')$ if either $s_1 <_1 s_1'$, or $s_1 = s_1'$ and $s_2 <_2 s_2'$. If $<_1$ and $<_2$ are total orders, then so is the corresponding lexicographical order $<$.

Complexity Measures

Complexity measures state how resource consumption of an algorithm (in terms of messages, time, or space) grows in relation to input size. We are not interested in the precise utilization of resources on a given network, as this may depend, for instance, on the underlying hardware, but in how quickly resource consumption grows if the input size increases in a linear fashion. Different executions of an algorithm usually give rise to a different utilization of resources. We consider worst-case complexity, which provides an upper bound on the considered resource consumption for all executions, and average-case complexity, which gives the average resource consumption over all executions. For example, if an algorithm has a worst-case message complexity of $O(n^2)$, then for an input of size n, the algorithm in the worst case takes *in the order of* n^2 messages, give or take a constant factor.

Let $f, g : \mathbb{N} \rightarrow \mathbb{R}_{>0}$, meaning that the functions f and g map natural numbers to positive real numbers. We write $f \in O(g)$ if f is bounded from above by g, multiplied by some positive constant:

$$f \in O(g) \text{ if, for some } c \in \mathbb{R}_{>0},\ f(n) \leq c \cdot g(n) \text{ for all } n \in \mathbb{N}.$$

Likewise, $f \in \Omega(g)$ means that f is bounded by g from below, multiplied by some positive constant:

$$f \in \Omega(g) \text{ if, for some } c \in \mathbb{R}_{>0},\ f(n) \geq c \cdot g(n) \text{ for all } n \in \mathbb{N}.$$

Finally, $f \in \Theta(g)$ means that f is bounded by g from above as well as below:

$$f \in \Theta(g) \text{ if } f \in O(g) \text{ and } f \in \Omega(g).$$

So if an algorithm has, say, a worst-case message complexity of $\Theta(n^2)$, then this upper bound is sharp.

Example 2.1 Let $a, b \in \mathbb{R}_{>0}$.

- $n^a \in O(n^b)$ if $a \leq b$.
- $n^a \in O(b^n)$ if $b > 1$.
- $\log_a n \in O(n^b)$ for all a, b.

Divide-and-conquer algorithms, which recursively divide a problem into smaller subproblems until they are simple enough to be solved directly, typically have a logarithm in their time complexity. The reason is that dividing a problem of input size 2^k into subproblems of size 1 takes k steps, and $k = \log_2 2^k$.

One can write $O(\log n)$ instead of $O(\log_a n)$. By definition, $\log_a a^n = n$, which implies $a^{\log_a n} = n$. So

$$a^{\log_a b \cdot \log_b n} = b^{\log_b n} = n.$$

Therefore, $\log_a b{\cdot}\log_b n = \log_a n$. So $\log_b n \in O(\log_a n)$ for all $a, b \in \mathbb{R}_{>0}$.

Resource consumption of an execution of a distributed algorithm can be measured in several ways:

- *Message complexity:* the total number of messages exchanged.
- *Bit complexity:* the total number of bits exchanged.
- *Time complexity:* the amount of time required to complete the execution.
- *Space complexity:* the total amount of space needed for the processes.

In the context of the message-passing framework, we focus mostly on message complexity. Bit complexity is interesting only when messages can be very long. In the analysis of time complexity, we assume that internal computation steps at a process take no time and that a message is received at most one time unit after it is sent. In the context of the shared-memory framework, we focus on time and space complexity.

Special Operations on Numbers

While determining a complexity measure, we will sometimes use the notations $\lfloor a \rfloor$ and $\lceil a \rceil$ with $a \in \mathbb{R}_{>0}$, meaning the largest integer k such that $k \leq a$ and the smallest integer ℓ such that $\ell \geq a$, respectively.

The integer domain *modulo* a positive natural number n is represented by the elements $\{0, ..., n - 1\}$. Each integer k has a representative modulo n, denoted by k mod n, which is the unique $\ell \in \{0, ..., n{-}1\}$ such that $k - \ell$ is divisible by n. In other words, k mod n is the remainder if k is divided by n. This means that when counting upward modulo n, integer values are wrapped around when n is reached: n mod n is represented by 0, $(n + 1)$ mod n by 1, and so on. Addition and multiplication on integers carry over to modulo arithmetic in a straightforward fashion: (j mod n) + (k mod n) = (j

$+ k) \bmod n$ and $(j \bmod n) \cdot (k \bmod n) = (j \cdot k) \bmod n$.

2.2 Message Passing

In a message-passing framework, a distributed system consists of a finite network (or graph) of N processes (or nodes), which are denoted by p, q, r. Each process in the network carries a unique ID. The processes in the network are connected by channels (or edges), through which they can send messages to each other, which are denoted by m. A channel between processes p and q is denoted by pq. There is at most one channel from one process to another, and a process does not have a channel to itself. Sometimes a process may want to communicate a message to itself, but clearly no channel is needed for this. We use E and D to denote the number of channels and the diameter of the network, respectively.

A process can record its own state and the messages it sends and receives; processes do not share memory or a global clock. Each process knows only its (direct) neighbors in the network. The topology of the network is assumed to be *strongly connected*, meaning that there is a path from each process to every other process.

Communication in the network is *asynchronous*, meaning that sending and receiving a message are distinct events at the sending and receiving process, respectively. The delay of a message in a channel is arbitrary but finite. We assume that communication protocols are being used to avoid having messages garbled, duplicated, or lost. Such a protocol defines a set of rules for how data is transmitted through the network. Channels need not be FIFO (first-in, first-out), which means that messages can overtake each other. To mimic a FIFO channel, a simple protocol suffices, which typically provides a sequence number to each message. (Special care is required if these numbers overflow, meaning that they go outside the available range of integers.) However, for the purposes of this book, it is important to study the fully asynchronous nature of distributed algorithms, which comes to light in a setting with non-FIFO channels.

In a *directed* network, messages can travel only in one direction through a channel, while in an *undirected* network, messages can travel either way. If the network is directed, then a channel pq is directed from p to q. For instance, undirected channels are required for distributed algorithms that use an acknowledgment scheme. Acyclic networks will always be undirected, since otherwise the network would not be strongly connected. A network topology is called *complete* if there is an undirected channel (or, equivalently, two directed channels in both directions) between each pair of different processes.

A *spanning tree* of an undirected network is a connected, acyclic graph that consists of all the nodes and a subset of the edges in the network. The edges of the spanning tree are called *tree edges*, and all other edges are called *frond edges*. Often the edges in a spanning tree are given a direction to obtain a *sink tree*, in which all paths lead to the same node, called the *root*; each directed edge leads from a *child* to its *parent*. A *leaf* is a node without children in the tree.

At some places, we will deviate from the assumptions mentioned so far and move to, for instance, synchronous communication in which the sending and receiving of the same message form one atomic step; unreliable processes; channels; FIFO channels; or processes without a unique ID, or processes with a probabilistic specification instead of a deterministic one. Such deviations will be stated explicitly.

Transition Systems

A distributed algorithm, which runs on a distributed system, provides (usually deterministic) specifications for the individual processes. The global state of a distributed algorithm, called a *configuration*, evolves by means of *transitions*. The overall behavior of distributed algorithms running on a distributed system is captured by a *transition system*, which consists of

- a set $\mathcal{C}$ of configurations,
- a binary transition relation $\rightarrow$ on $\mathcal{C}$, and
- a set $\mathcal{I} \subseteq \mathcal{C}$ of *initial* configurations.

The underlying idea is that every run of a distributed algorithm starts in an initial configuration, and each of its transitions adheres to the transition relation. A configuration γ is *terminal* if it has no outgoing transition: $\gamma \rightarrow \delta$ for no $\delta \in \mathcal{C}$. An *execution* of a distributed algorithm is a sequence $\gamma_0\ \gamma_1\ \gamma_2 \cdots$ of configurations that is either infinite or ends in a terminal configuration γ_k, such that

- $\gamma_0 \in \mathcal{I}$, and
- $\gamma_i \rightarrow \gamma_{i+1}$ for all $i \geq 0$ (and in the case of a finite execution, $i < k$).

A configuration δ is *reachable* if it is part of an execution: there is a $\gamma_0 \in \mathcal{I}$ and a sequence $\gamma_0\ \gamma_1 \cdots \gamma_k$ with $\gamma_i \rightarrow \gamma_{i+1}$ for all $0 \leq i < k$ and $\gamma_k = \delta$.

States and Events

The configuration of a distributed system consists of the local *states* of its processes and the messages in transit. A transition between configurations of a distributed system is associated with an *event* (or, in the case of synchronous communication, two events) at one (or two) of its processes. A process can perform *internal*, *send*, and *receive* events. An internal event influences only the state at the process where the event is performed. Typical internal events are reading or writing to a local variable. Assignment of a new value to a variable is written as $\leftarrow$; for example, $n \leftarrow n + 1$ means that the value of variable n, representing a natural number, is increased by 1. A send event, in principle, gives rise to a corresponding receive event of the same message at another process. It is assumed that two different events never happen at the same moment in real time (with the exception of synchronous communication).

A process is an *initiator* if its first event is an internal or send event; that is, an initiator can start performing events without input from another process. An algorithm is *centralized* if there is exactly one initiator. A *decentralized* algorithm can have multiple initiators.

Assertions

An assertion is predicated on the configurations of an algorithm. That is, in each configuration, the assertion is either true or false.

An assertion is a *safety property* if it is supposed to be true in each reachable configuration of the algorithm. A safety property typically expresses that something bad will never happen. Informal examples of safety properties are:

- You can always count on me.
- Never will a deadlock occur.
- Each message reaches its destination within one second after it is sent.

Phrased differently, the first example states that it is never the case that you cannot count on me, and the second example states that never a message fails to reach its destination within one second.

In general, it is undecidable whether a given configuration is reachable. An assertion P is an *invariant* if

- $P(\gamma)$ for all $\gamma \in \mathcal{I}$, and
- if $\gamma \rightarrow \delta$ and $P(\gamma)$, then $P(\delta)$.

In other words, an invariant is true in all initial configurations and is preserved by all transitions. Clearly, each invariant is a safety property. Note that checking whether an assertion is an invariant does not involve reachability, so such a proof can focus on local transitions. The price to pay is that an invariant must also hold for unreachable configurations, although in principle we are only interested in the reachable ones.

An assertion is a *liveness property* if it expresses that each execution contains a configuration in which the assertion holds. A liveness property typically expresses that something good will eventually happen but provides no time bound on how long this may take. Informal examples of liveness properties are:

- What goes up, must come down.
- The program always eventually terminates.
- Each message eventually reaches its destination.

The last informal safety and liveness properties show that the difference between these notions can be subtle. One could say that the safety property regards the foreseeable future and the liveness property the unforeseeable future.

A liveness property sometimes holds only with respect to the *fair* executions of an algorithm. For example, consider a simple algorithm that consists of flipping a coin until the result is tails. Since there is an infinite execution in which the outcome of every coin flip is heads, the liveness property that eventually the outcome will be tails does not hold. However, if the coin is fair, this infinite execution has zero chance of happening; infinitely often we flip the coin with the possible outcome tails, but never is the outcome tails. We say that an execution is *fair* if every event that can happen in infinitely many configurations in the execution is performed infinitely often during the execution. The infinite execution in which the outcome of every coin flip is heads is not fair. Note that any finite execution is trivially fair.

Causal Order

In each configuration of an asynchronous distributed system, events that can occur at different processes are always independent, meaning that they can happen in any order. The *causal order* $\prec$ is a binary relation on events in an execution such that $a \prec b$ if and only if a must happen before b. That is, the events in the execution cannot be reordered in such a way that a happens after b. The causal order for an execution is the smallest relation such that

- if a and b are events at the same process and a occurs before b, then $a \prec b$,
- if a is a send and b the corresponding receive event, then $a \prec b$, and
- if $a \prec b$ and $b \prec c$, then $a \prec c$.

The first two clauses define direct causal order relations between events, and the third clause takes the transitive closure. The causal order relation is irreflexive: $a \prec a$ never holds. We write $a \preceq b$ if either $a \prec b$ or $a = b$. Distinct events in an execution that are not causally related are called *concurrent*. An important challenge in the design of distributed systems is to cope with concurrency (for example, to avoid race conditions).

A permutation of concurrent events in an execution does not affect the result of the execution. These permutations together form a *computation*. All executions of a computation start in the same configuration, and if they are finite, they all end in the same terminal configuration. In general, we will consider computations rather than executions.

Example 2.2 Consider an execution *abc*. If $a \prec b$ is the only causal relationship, then the executions *abc*, *acb*, and *cab* form one computation.

Logical Clocks

A common physical clock, which approximates the global real time, is in general difficult to maintain by the separate processes in a distributed system. For many applications, however, we are not interested in the precise moments in time at which events occur but only in the ordering of these occurrences in time. A *logical clock* C maps occurrences of events in a computation to a partially ordered set such that

$$a \prec b \;\Rightarrow\; C(a) < C(b).$$

Lamport's clock LC assigns to each event a the length k of a longest causality chain $a_1 \prec \cdots \prec a_k = a$ in the computation. Clearly, $a \prec b$ implies $LC(a) < LC(b)$, because $a_1 \prec \cdots \prec a_k \prec b$. So Lamport's clock is a logical clock. The clock values that Lamport's clock assigns to events can be computed at run-time as follows. Consider an event a, and let k be the clock value of the previous event at the same process ($k = 0$ if there is no such previous event).

- If a is an internal or send event, then $LC(a) = k + 1$.
- If a is a receive event and b the send event corresponding to a, then $LC(a) = \max\{k, LC(b)\}+1$.

Example 2.3 Consider the following sequences of events at processes p_0, p_1, and p_2.

$$\begin{array}{llllll}
p_0: & a & s_1 & r_3 & b \\
p_1: & c & r_2 & s_3 & \\
p_2: & r_1 & d & s_2 & e
\end{array}$$

Here s_i and r_i are corresponding send and receive events for $i = 1, 2, 3$, while a, b, c, d, e are internal events. Lamport's clock provides these events with the following values.

$$\begin{array}{lllll}
p_0: & 1 & 2 & 8 & 9 \\
p_1: & 1 & 6 & 7 & \\
p_2: & 3 & 4 & 5 & 6
\end{array}$$

Since the values of Lamport's clock are in the totally ordered set of natural numbers, there is an ordering relation between the clock values of each pair of events, even if they are concurrent. Sometimes it is convenient to use a logical clock for which this is not the case. The *vector clock* VC maps each event in a computation to a unique value such that

$$a \prec b \;\Leftrightarrow\; VC(a) < VC(b).$$

Let the network consist of processes $p_0, ..., p_{N-1}$. The vector clock assigns to events in a computation values in $\mathbb{N}^N$, whereby this set is provided with a partial order defined by

$$(k_0,\ldots,k_{N-1}) \leq (\ell_0,\ldots,\ell_{N-1}) \Leftrightarrow k_i \leq \ell_i \text{ for all } i = 0,\ldots,N-1.$$

We write $(k_0, ..., k_{N-1}) < (\ell_0, ..., \ell_{N-1})$ if, moreover, these sequences are unequal to each other. (We note that in contrast to this partial order, the lexicographical order on $\mathbb{N}^N$ is total.)

The vector clock is defined as $VC(a) = (k_0, ..., k_{N-1})$, where each k_i is the length of a longest causality chain $a^i_1 \prec \cdots \prec a^i_{k_i}$ of events at process p_i with $a^i_{k_i} \preceq a$. On the one hand, $a \prec b$ implies $VC(a) < VC(b)$. This follows from the fact that $c \preceq a$ implies $c \prec b$ for each event c. And at the process where b occurs, there is a longer causality chain for b than for a. Conversely, $VC(a) < VC(b)$ implies $a \prec b$. Consider the longest causality chain $a^i_1 \prec \cdots \prec a^i_k = a$ of events at the process p_i where a occurs. Then $VC(a) < VC(b)$ implies that the ith coefficient of $VC(b)$ is at least k, and so $a \preceq b$. Since clearly a and b must be distinct events, $a \prec b$. The vector clock can also be computed at run-time (see exercise 2.9).

Example 2.4 Consider the sequences of events at processes p_0, p_1, and p_2 from example 2.3. The vector clock provides these events with the following values.

p_0 :	(1 0 0)	(2 0 0)	(3 3 3)	(4 3 3)
p_1 :	(0 1 0)	(2 2 3)	(2 3 3)	
p_2 :	(2 0 1)	(2 0 2)	(2 0 3)	(2 0 4)

Basic and Control Algorithms

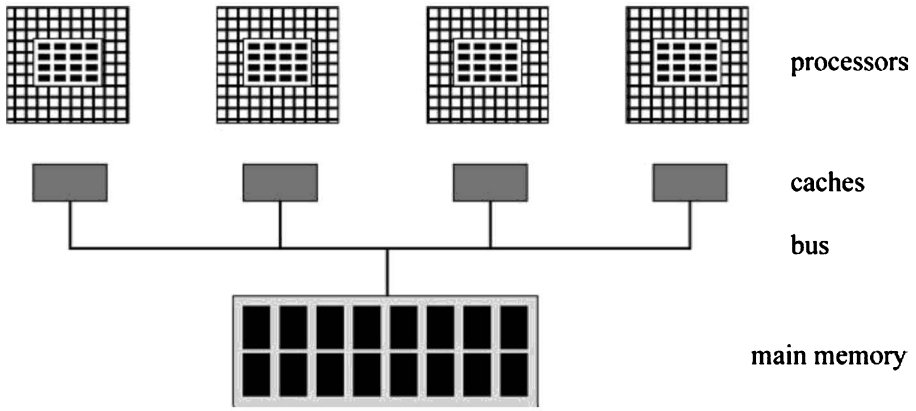
Several chapters will discuss distributed algorithms to provide some service or detect a certain property during the execution of another distributed algorithm. For instance, chapter 3 shows how processes can take a snapshot of a configuration in the ongoing computation, chapter 6 treats how processes can detect termination, and chapter 7 discusses garbage collection to reclaim memory taken by inaccessible objects. Then the underlying distributed algorithm for which we are taking a snapshot, detecting termination, or collecting garbage is called the *basic algorithm*, while the distributed algorithm put on top for executing this specific task is called the *control algorithm*. We refer to a message of the basic or control algorithm as a *basic message* or *control message*, respectively.

2.3 Shared Memory

In a shared-memory framework, a number of (hardware) processors communicate with main memory, and with the other processors, over a bus. The processes (or threads) are sequential programs that can run on a processor; at any time, at most one process is running on each processor. Processes communicate with each other, asynchronously, via variables in main memory, called registers (or fields). We distinguish single-reader registers, which are fields that can be read by only one process, and multi-reader registers, which can be read by all processes. Likewise, we distinguish single-writer and multi-writer registers. Unless stated differently, it will be assumed that registers are multi-reader.

In practice, many processors allow out-of-order execution of read and write operations to increase performance. This means a processor may execute such operations in an order governed by the availability of input data to avoid delays, instead of the order declared by the program. If a programmer wants to disallow out-of-order execution with regard to certain read and write operations, this must be specified explicitly by placing memory barriers; operations before a memory barrier cannot be reordered behind operations after this memory barrier. In the algorithms for shared memory described in this book, we will take the liberty of disregarding out-of-order execution.

Processors and main memory are snooping, meaning that they are listening for messages that are broadcast over the bus. A cache is a relatively small memory unit, local to a processor, that stores copies of data from main memory that may be frequently used. Access to main memory is slow compared to cache access. Changes to data values are therefore accumulated in the cache and written back to main memory when needed to make space in the cache, when another process wants the value, or at some memory barrier which enforces that writes are flushed to main memory. A cache consists of data blocks of fixed size called cache lines; copying data from and to main memory is performed at the level of cache lines. When a process takes a cache miss, meaning that it cannot find some data in its cache, the required data is fetched from main memory or provided by another processor.



In the presence of caches, synchronization primitives are typically needed at times to avoid that processes read stale values from their cache, enforcing that they fetch fresh values from main memory instead. Notably, one can impose memory barriers or declare that for certain variables the reads and writes are always with regard to main memory (for example, in Java, such variables are declared volatile).

A cache coherence protocol maintains the consistency of the data in the local caches. If a process writes a value in a cache line, the cache coherence protocol makes sure that this same cache line in other caches is invalidated, so that the values in these lines become obsolete. For example, the MSI protocol identifies three possible states for a cache line:

- *Modified*: The line has been written to in the cache and must eventually be stored in main memory.
- *Shared*: The line has not been written to in any cache and can be read.
- *Invalid*: The line must be refreshed before it can be read.

When a process writes to a nonmodified cache line, this line becomes modified in the corresponding cache and is invalidated in all other caches. If a process reads a cache line that is modified in some other cache, then in both caches this line becomes shared. When a modified cache line becomes shared or invalid, it is flushed to main memory. Note that if a cache line is modified in some cache, then it can only occur as invalid in other caches.

An event at a process typically consists of one atomic read or write to a register. *Read-modify-write* operations, however, read a memory location and write a new value into it in one atomic step; this new value may be computed on the basis of the value returned by the read, without interruption by an event of another process. Atomicity is achieved through keeping a hardware lock on the bus from the moment the value is read until the moment the new value is written. Typical examples of read-modify-write operations are:

- *test-and-set*, which writes *true* in a Boolean register and returns the previous value of the register; and
- *get-and-increment*, which increases the value of an integer register by 1 and returns the previous value of the register;
- *get-and-set(new)*, which writes the value *new* in a register and returns the previous value of the register;
- *compare-and-set*(*old*, *new*), which checks whether the value of the register equals *old* and, if so, overwrites it with the value *new*. A Boolean value is returned to signal whether the new value was actually written.

Read-modify-write operations tend to impose a memory barrier.

To avoid simultaneous use of a resource, such as a block of memory, mutual exclusion is of the essence in a shared-memory framework. A common way to achieve this is by means of software locks; to become privileged, one must obtain the lock, and at any time at most one process can hold the lock. The mutual exclusion algorithms for shared memory presented in chapter 14 are all based on this principle. In a setting with locks, we distinguish between two kinds of progress properties for mutual exclusion algorithms. Here it is assumed that no process holds a lock forever; in particular, that processes do not crash. *Livelock-freeness* means that some process trying to get a lock always eventually succeeds. *Starvation-freeness* means that every process trying to get a lock eventually succeeds. For example, consider an algorithm in which some process infinitely often wants to obtain and release a certain lock. If in some execution this process obtains the lock only finitely often, then the algorithm is not starvation-free.

A disadvantage of locks is that they create a bottleneck, since many processes may concurrently be waiting to obtain a lock. Moreover, if a process holding a lock crashes, this may bring the execution of the entire system to a standstill. One can aim for stronger, nonblocking progress properties, where locks are disallowed. An algorithm is *lock-free* if, as long as it has not terminated, it is guaranteed that some process progresses (but it is allowed that some active process that has not crashed performs no events); it is *wait-free* if all active, noncrashed processes always progress. For example, consider an algorithm in which some process infinitely often performs the event *a*. If in some execution this process does not crash and performs *a* only finitely often, then the algorithm is not wait-free. Note that if a process crashes while holding a lock, then the lock-freeness (and therefore wait-freeness) property may be violated if another process waits in vain to get this lock. Therefore, this property precludes the use of locks. A software crash is always a consequence of a hardware instruction; for example, a program counter may be set to an incorrect address. This means that processes do not crash during a read-modify-write operation. Therefore, in the absence of a hardware crash, the lock on the bus is always eventually released. Hence, lock- and wait-free algorithms may be achieved with read-modify-write operations.

Locks and read-modify-write operations are usually furnished with automatic synchronization primitives: when a process acquires a lock or performs a read-modify-write operation, it invalidates its cache to ensure that fields are reread from main memory, and when it releases the lock, modified fields in its cache are written back to main memory.

Bibliographical notes

Lamport's clock originates from [54]. The vector clock was proposed independently in [34] and [64].

2.4 Exercises

Exercise 2.1 For each of the following functions f and g, say whether $f \in O(g)$ and/or $g \in O(f)$.

(a) $f(n) = 5 \cdot n^2 + 3 \cdot n + 7$ and $g(n) = n^3$.
(b) $f(n) = \sum_{i=1}^{n} i$ and $g(n) = n^2$.
(c) $f(n) = n^n$ and $g(n) = n!$.
(d) $f(n) = n \cdot \log_2 n$ and $g(n) = n \cdot \sqrt{\frac{n}{2}}$.
(e) $f(n) = n + n \cdot \log_2 n$ and $g(n) = n \cdot \sqrt{n}$.

Exercise 2.2 What is more general?

(a) An algorithm for directed or undirected networks.
(b) A control algorithm for centralized or decentralized basic algorithms.

Exercise 2.3 [92] Give a transition system S and an assertion P such that P is a safety property but not an invariant of S.

Exercise 2.4 Define the union of $S_1 = (\mathcal{C}, \rightarrow_1, \mathcal{I})$ and $S_2 = (\mathcal{C}, \rightarrow_2, \mathcal{I})$ as $S = (\mathcal{C}, \rightarrow_1 \cup \rightarrow_2, \mathcal{I})$.

(a) [92] Prove that if P is an invariant of S_1 and S_2, then P is an invariant of S.
(b) Give an example where P is a safety property of S_1 and S_2 but not of S.

Exercise 2.5 Consider an execution $abcd$. Let $a \prec c$ and $b \prec d$ be the only causal relationships. Which executions are in the same computation as $abcd$?

Exercise 2.6 Consider the following sequences of events at processes p_0, p_1, p_2, and p_3.

$$\begin{array}{lllllll}
p_0: & s_1 & s_2 & r_5 & & & \\
p_1: & r_2 & s_5 & & & & \\
p_2: & r_1 & a & s_4 & r_3 & r_6 & \\
p_3: & s_3 & r_4 & b & s_6 & &
\end{array}$$

Here s_i and r_i are corresponding send and receive events for all i, while a and b are internal events. Use Lamport's logical clock to assign clock values to these events. Do the same for the vector clock.

Exercise 2.7 [92] Define the causal order for the transitions of a system with synchronous communication. Adapt Lamport's logical clock for such systems, and give a distributed algorithm for computing the clock at run-time.

Exercise 2.8 Give an example where $LC(a) < LC(b)$, while a and b are concurrent events.

Exercise 2.9 Give an algorithm to compute the vector clock at run-time.

Exercise 2.10 Propose two adaptations of the MSI cache coherence protocol, in both cases by introducing an additional state for cache lines.

(a) Reduce bus traffic when a process reads a certain variable and then writes to it in its cache, while the corresponding cache line occurs only as invalid in other caches.
(b) Reduce the number of flushes to main memory in the case of a continuous stream of reads and writes by different processes to a certain variable.

Exercise 2.11 For each of the following approaches for handing a free lock to a waiting process, say whether it is starvation-free.

(a) Randomly hand the lock to one of the waiting processes.
(b) Processes that wait for the lock are placed in a FIFO (first-in, first-out) queue. Hand the lock to the process at the head of this queue (and remove it from the queue).
(c) Hand the lock to the waiting process with the highest ID.

3
Snapshots

A *snapshot* of an execution of a distributed algorithm is a configuration of this execution that consists of the local states of the processes and the messages in transit. Snapshots are useful to try to determine offline properties that will remain true as soon as they have become true, such as deadlock (see chapter 5), termination (see chapter 6), or being garbage (see chapter 7). Moreover, snapshots can be used for checkpointing to restart after a failure (see section 3.3) or for debugging.

In a centralized environment, the execution of a program can usually be interrupted at any moment to query the program state, consisting of the values of the program variables. In a distributed setting, this is not the case. Suppose a process that is involved in the execution of a distributed algorithm wants to take a snapshot of a configuration of the ongoing execution. Then it should ask all processes to take a snapshot of their state. Processes, moreover, need to compute channel states of messages that were in transit at the moment of the snapshot. The challenge is to develop a snapshot algorithm that works at run-time; that is, without freezing the execution of the basic algorithm for which the snapshot is taken. Messages of the basic and the snapshot algorithm are called basic and control messages, respectively.

A complication is that processes take local snapshots and compute channel states at different moments in time. Therefore, a snapshot may actually not represent a configuration of the ongoing execution, but a configuration of an execution in the same computation as the actual execution is good enough. Such a snapshot is called *consistent*. One should be careful not to take an inconsistent snapshot. For instance, a process p could take a local snapshot and then send a basic message m to a process q, where q could either take a local snapshot after the receipt of m or include m in the state of the channel pq. This would turn m into an *orphan* message that was, with regard to the snapshot, not sent according to p but was received or in transit according to q. Likewise, p could send a basic message m before taking its local snapshot, while q could receive m after taking its local snapshot and exclude m from the channel state of pq. This would turn m into a *lost* message that was, with regard to the snapshot, sent according to p but neither received nor in transit according to q. Such inconsistent snapshots clearly should be avoided.

An event is called *presnapshot* if it occurs at a process before the local snapshot at this process is taken; otherwise it is called *postsnapshot*. A snapshot is consistent if

(1) no postsnapshot event is causally before a presnapshot event, and
(2) a basic message is included in a channel state if and only if the corresponding send event is presnapshot while the corresponding receive event is postsnapshot.

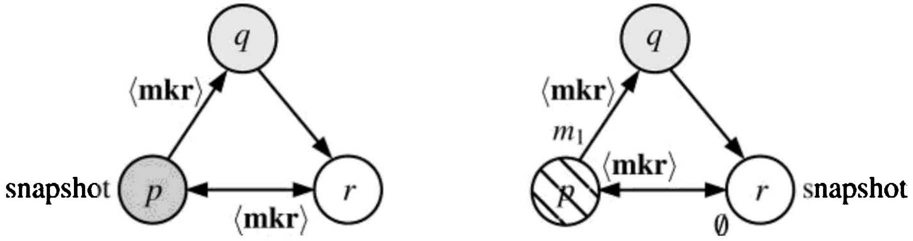
The first property guarantees that all presnapshot events can be placed before the postsnapshot events in the actual execution by means of permutations that do not violate the causal order. The second property guarantees that there are no lost messages. And the two properties together guarantee that there are no orphan messages. This implies that the snapshot is a configuration of an execution that is in the same computation as the actual execution.

We discuss two decentralized snapshot algorithms for directed networks; the first one requires channels to be FIFO. In these algorithms, the individual processes record fragments of the snapshot; the subsequent phase of collecting these fragments to obtain a composite view is omitted here.

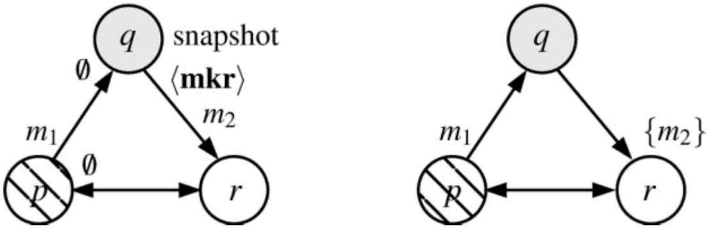
3.1 Chandy-Lamport Algorithm

The Chandy-Lamport snapshot algorithm requires that channels be FIFO. Any process that is designated to be an initiator can decide to take a local snapshot of its state. It then sends a control message ⟨**marker**⟩ through all its outgoing channels to let its neighbors take a snapshot also. When a process that has not yet taken a snapshot receives a ⟨**marker**⟩ message, it takes a local snapshot of its state and sends a ⟨**marker**⟩ message through all its outgoing channels. A process q computes as channel state for an incoming channel pq the (basic) messages that it receives via pq after taking its local snapshot and before receiving a ⟨**marker**⟩ message from p. The Chandy-Lamport algorithm terminates at a process when it has received a ⟨**marker**⟩ message through all its incoming channels.

Example 3.1 We consider one possible computation of the Chandy-Lamport algorithm on the directed network pictured here. First, process p takes a local snapshot of its state (dark gray) and sends ⟨**marker**⟩ into its two outgoing channels pq and pr. Next, p sends a basic message m_1 to process q and changes its state (to striped). Concurrently, process r receives ⟨**marker**⟩ from p and as a result sends ⟨**marker**⟩ into its outgoing channel rp, takes a local snapshot of its state (white), and computes the channel state ∅ for its incoming channel pr.



Next, p receives ⟨**marker**⟩ from r and as a result computes the channel state ∅ for its incoming channel rp. Concurrently, q sends a basic message m_2 to r. Next, q receives ⟨**marker**⟩ from p and as a result sends ⟨**marker**⟩ into its outgoing channel qr, takes a local snapshot of its state (light gray), and computes the channel state ∅ for its incoming channel pq. Finally, r receives m_2 and next ⟨**marker**⟩ from q, and as a result it computes the channel state $\{m_2\}$ for its incoming channel qr.



We note that the computed snapshot (states: dark gray, light gray, white; channels: ∅, ∅, ∅, $\{m_2\}$) is not a configuration of the actual execution. However, both the sending of m_1 and the internal event at p that changes its state from light gray to striped are not causally before the sending of m_2. Therefore, the snapshot is a configuration of an execution in the same computation as the actual execution.

We argue that properties (1) and (2) of a consistent snapshot are satisfied. First, if an event a is causally before a presnapshot event b, then a is also presnapshot. If a and b occur at the same process, then this is trivially the case. The interesting case is where a is a send and b is the corresponding receive event. Suppose that a occurs at process p, and b occurs at process q. Since b is presnapshot, q has not yet received a ⟨**marker**⟩

message at the time it performs b. Since channels are FIFO, this implies that p has not yet sent a ⟨**marker**⟩ message to q at the time it performs a. Hence, a is presnapshot.

Second, a basic message m via a channel pq is included in the channel state of pq if and only if the corresponding send event at p is presnapshot and the corresponding receive event at q is postsnapshot. Suppose m is included in the channel state. Then q receives m before ⟨**marker**⟩ through pq, so since channels are FIFO, p must send m before ⟨**marker**⟩ into pq. Hence, sending m is presnapshot. Furthermore, q starts computing the channel state of pq after taking its local snapshot. So receiving m is postsnapshot. Vice versa, suppose m is excluded from the channel state and its reception is postsnapshot. Then q receives ⟨**marker**⟩ before m through pq, so since channels are FIFO, p must send ⟨**marker**⟩ before m into pq. Hence, sending m is postsnapshot.

The Chandy-Lamport algorithm requires E control messages, one per (directed) channel, and it takes at most $O(D)$ time units to complete a snapshot.

3.2 Lai-Yang Algorithm

The Lai-Yang snapshot algorithm does not require channels to be FIFO. Any initiator can decide to take a local snapshot of its state. As long as a process has not taken a local snapshot, it appends *false* to each outgoing basic message; after its local snapshot, it appends *true* to these messages. When a process that has not yet taken a local snapshot receives a message with the tag *true*, it takes a local snapshot of its state before receiving this message. A process q computes as channel state of an incoming channel pq the basic messages with the tag *false* that it receives through this channel after having taken its local snapshot.

There are two complications. First, if after its local snapshot an initiator would happen not to send any basic messages, other processes might never take a local snapshot. Second, how does a process know when it can stop waiting for basic messages with the tag *false* and compute the state of an incoming channel? Both issues are resolved by a special control message, which each process p sends into all its outgoing channels pq after having taken its local snapshot. This control message informs q how many basic messages with the tag *false* p has sent into the channel pq. If q has not yet taken a local snapshot, it takes one upon receiving this control message.

Example 3.2 Consider a network of two processes p and q, with non-FIFO channels pq and qp. We apply the Lai-Yang algorithm to take a snapshot.

Let p send basic messages ⟨m_1, *false*⟩ and ⟨m_2, *false*⟩ to q. Then it takes a local snapshot of its state and sends a control message to q, reporting that p sent two basic messages with the tag *false* to q. Next, p sends basic messages ⟨m_3, *true*⟩ and ⟨m_4, *true*⟩ to q. Let ⟨m_3, *true*⟩ arrive at q first. Then q takes a local snapshot of its state and sends a control message to p, reporting that q did not send any basic message with the tag *false* to p. Next, q waits until the control message from p, $\langle m_1, \mathit{false}\rangle$, and $\langle m_2, \mathit{false}\rangle$ have arrived and concludes that the channel state of pq consists of m_1 and m_2. Thanks to the tag *true*, q recognizes that m_3 and m_4 are not part of the channel state. When p receives q's control message, it concludes that the channel state of qp is empty.

Similar to the case with the Chandy-Lamport algorithm, we can argue that if an event a is causally before a presnapshot event b, then a is also presnapshot. Again, the interesting case is where a is a send and b is the corresponding receive event. The fact that b is presnapshot implies that the message sent by a carries the tag *false*. Hence, a is presnapshot.

Moreover, a basic message m via a channel pq is included in the channel state of pq if and only if it carries the tag *false*, meaning that its corresponding send event at p is presnapshot and the corresponding receive event at q is postsnapshot.

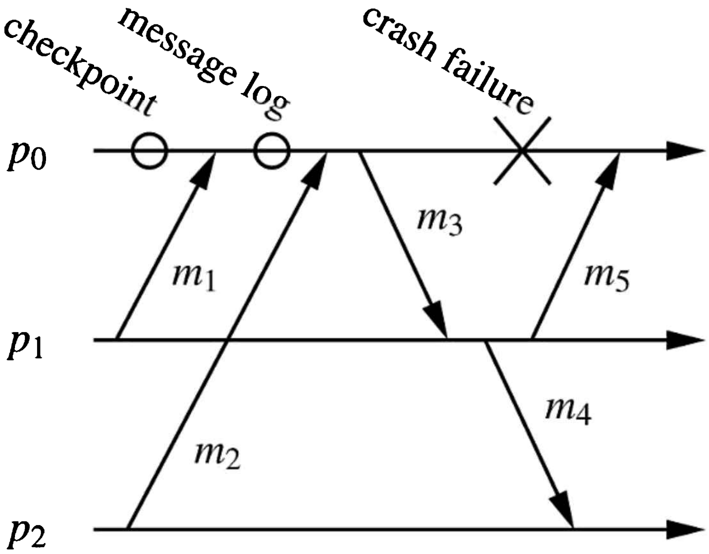
The Lai-Yang algorithm requires E control messages, and it takes at most $O(D)$ time units to complete a snapshot.

3.3 Peterson-Kearns Rollback Recovery Algorithm

Checkpointing and *rollback recovery* aim at coping with crash failures of processes in a network, meaning that they stop executing unexpectedly; this topic will be studied in depth in chapter 12. A standard assumption in this setting is that the network topology is complete, so that strong connectedness is preserved after crash failures. Moreover, we still presume that channels always function properly. Here we suppose that when a process crashes, another process will eventually take over its role and continue its execution. It is assumed that each process owns a failure detector that eventually picks up on every crash failure. If an upper bound on network latency, meaning the time between when a message is sent and when it is received, is known, and each process sends a heartbeat message at regular time intervals, then a failure detector can be implemented using a timeout mechanism. Section 12.3 provides an in-depth treatment of such devices. It is, moreover, assumed that each process p has *stable storage* at its disposal, meaning that this part of p's local memory is preserved in a consistent state and remains accessible to the other processes even after p has crashed. Without special precautions, inconsistent memory could remain if p crashes while it is performing a write operation. Stable storage can be implemented using two disks, where updates in memory on the first disk are faithfully copied to the second disk.

Each process periodically establishes a checkpoint by saving its state in stable storage; moreover, it continuously tracks which basic messages it received at which moments of its execution and periodically logs this information in stable storage. In contrast to the snapshot algorithms discussed in the previous two sections, processes do not coordinate their checkpoints. The underlying idea is that crashes occur rarely, which makes it attractive to only incur message overhead directly after such a crash instead of periodically. The price to be paid for this reduced message overhead is that after a crash a rollback must be performed because the uncoordinated checkpoints in the different processes may give rise to orphan or lost messages. The checkpoints and message logs of the processes in stable storage serve as the basis for rollback recovery, determining a consistent configuration in the past from which execution can be restarted after a crash. In general, an event must be rolled back (i.e., in the restart, it is supposed not to have occurred) if it either happened after the last checkpoint at the crashed process or is causally after such an irrecoverably lost event at the crashed process. Checkpointing and message logging are performed sporadically because they require stable storage, which makes these operations relatively expensive.

Example 3.3 The picture that follows shows the time line of events at three processes p_0, p_1, and p_2 with regard to some basic computation, where real time progresses from left to right.



Process p_0 performs a checkpoint, receives m_1 from p_1, performs a message log, receives m_2 from p_2, sends m_3 to p_1, and crashes. After receiving m_3, p_1 sends m_4 to p_2 and m_5 to p_0. The latter message reaches its destination after the crash has occurred.

When p_0 recovers from its crash, its state is restored to its last checkpoint, and the receipt of m_1 is replayed from its message log. The fact that p_0 received m_2 is irrecoverably lost. The resulting configuration is inconsistent because m_2 is a lost message and m_3 an orphan message. To resolve the latter, p_1 is rolled back to before the receipt of m_3. This, however, turns m_4 and m_5 into orphan messages. To resolve the former, p_2 in turn is rolled back to before the receipt of m_4. Furthermore, p_2 needs to resend m_2. Finally, after its recovery phase, p_0 needs to recognize that m_5 is an orphan message and discard it. This will be explained later.

The Peterson-Kearns rollback recovery algorithm uses the logical vector clock to determine which basic events should be discarded in the rollback. Each basic message contains the vector time of its send event, so that the vector time of the corresponding receive event can be determined; see the computation at run-time of Lamport's logical clock in section 2.2 as well as exercise 2.9. Control events of the rollback procedure are not taken into account by the vector clock. The vector time of a process is the vector time of the last basic event it performed; initially it consists only of zeros. When a checkpoint is performed, not only the state but also the vector time of the process is saved in stable storage. Moreover, vector times of incoming basic messages are kept in the message log. While a process is performing the rollback procedure, its basic computation is stalled.

After a crashed process p_i has restarted (i.e., its execution has been taken over by another process), it retrieves its last checkpoint and message log from stable storage. From this checkpoint, it replays events according to its basic algorithm until its message log is exhausted and as a result the next basic event cannot be reconstructed. Then it initiates a run of the rollback procedure at the other processes by sending them a control message that carries the number of basic events at p_i. To be more precise, if $(k_0, ..., k_{N-1})$ is the vector time of p_i's last reconstructed basic event, then p_i's control message contains (k_i, i). The rollback procedure discards those basic events that are causally after a basic event at p_i that was irrecoverably lost in the crash; that is, all basic events for which the ith coordinate of the vector time is greater than k_i.

When p_i's control message arrives at a p_j, that process checks whether the ith coordinate of its current vector time is greater than k_i. If it is, then a basic event at p_i that is causally before basic events at p_j has been lost in the crash and was not restored by the recovery procedure at p_i. In that case, p_j restarts at its last checkpoint for which the ith coordinate of the vector time is not greater than k_i. From there, it replays events according to its basic algorithm up to (but not including) its first event at which the ith coordinate of the vector time is greater than k_i. Basic messages that were received by p_j beyond this point and at which the ith coordinate of the vector time (of the corresponding send event) is not greater than k_i are kept, since discarding them would needlessly turn them into lost messages; they can be clustered right after the point where the replay at p_j has halted. On the other hand, received basic messages for which the ith coordinate of the vector time is greater than k_i are discarded, since keeping them would turn them into orphan messages.

An orphaned message, for which the corresponding send event was rolled back, may reach its destination after completion of the recovery phase. This is remedied by maintaining a sequence number, which initially is zero at all processes. At each new recovery phase, all processes increase their sequence number by 1. They keep the sequence number paired with the time stamp (k_i, i) of the corresponding recovery phase in stable storage. The sequence number of the sender is attached to basic messages, so that such orphan messages can be recognized and discarded by the receiver.

Basic send events to the crashed process p_i that were not rolled back are repeated by the sender. The reason is that the corresponding receive event may have been irrecoverably lost in the crash. If this is not the case, p_i will recognize from the vector time of the send event that it received this message before, so this second instance of the message will be discarded. To increase efficiency, message logs can contain not only receive events but also send events to support the resending of basic messages.

Example 3.4 We apply the Peterson-Kearns algorithm in the scenario given in example 3.3. Let the sequence number initially be 0; all basic messages carry this number. After p_0 has crashed, it restarts from its last checkpoint with sequence number 1 and replays the receipt of m_1 from its message log. Let this receive event have vector time (k_0, k_1, k_2). Then p_0 sends control messages to p_1 and p_2 with the time stamp $(k_0, 0)$ and sequence number 1. Upon receipt of this message, p_1 and p_2 start a run of the rollback procedure with sequence number 1; they store $(k_0, 0)$ paired with the sequence number 1. Because of the receipt of m_3 and m_4, the vector times at p_1 and p_2 carry a value greater than k_0 at index 0. So both processes restart at their last checkpoint (not shown in the picture) and replay events. These replays halt right before the receipt of m_3 and m_4, respectively. Messages m_1 and m_2 are resent by p_1 and p_2, respectively. At p_0, m_1 is discarded because it is in p_0's message log, while m_2 is treated as a new message. When m_5 reaches p_0, it is discarded as an orphan message, because its sequence number is 0, while the vector time of its send event carries a value greater than k_0 at index 0.

The Peterson-Kearns algorithm cannot cope with multiple concurrent crashes. The underlying idea is that if crashes are rare and a recovery phase takes little time, it is reasonable to neglect the possibility of a crash during a recovery phase.

Bibliographical notes

The Chandy-Lamport algorithm originates from [18], and the Lai-Yang algorithm comes from [52]; the special control message was suggested in [64]. The Peterson-Kearns algorithm stems from [75]. An implementation of stable storage was proposed in [57].

3.4 Exercises

Exercise 3.1 Argue that a consistent snapshot, meaning one that satisfies properties (1) and (2), rules out orphan messages.

Exercise 3.2 Give an example to show that the Chandy-Lamport algorithm is flawed if channels are non-FIFO.

Exercise 3.3 Propose an adaptation of the Chandy-Lamport algorithm in which basic messages may be buffered at the receiving processes and in which the channel states of the snapshot are always empty.

Exercise 3.4 Give an example in which the Lai-Yang algorithm computes a snapshot that is not a configuration of the ongoing execution.

Exercise 3.5 Adapt the Lai-Yang algorithm so that it supports multiple subsequent snapshots.

Exercise 3.6 Give a snapshot algorithm for undirected networks with non-FIFO channels that uses

- marker messages, tagged with the number of basic messages sent into a channel before the marker message,
- acknowledgments, and
- temporary (local) freezing of the basic execution.

Exercise 3.7 Suppose sequence numbers were omitted from the Peterson-Kearns algorithm. What would go wrong in example 3.4?

Exercise 3.8 Give an example in which the rollback procedure of the Peterson-Kearns algorithm would roll back a certain event if Lamport's clock were used, but this event is not rolled back with the vector clock.

4

Waves

In distributed computing, a process often needs to gather information from all other processes in the network. This process then typically sends a request through the network, which incites the other processes to reply with the required information. Notable examples are termination detection (see chapter 6), routing (see chapter 8), and election of a leader in the network (see chapter 9).

This procedure is formalized in the notion of a *wave algorithm*, in which each computation, called a *wave*, satisfies the following three properties:

- It is finite.
- It contains one or more *decide* events.
- For each decide event a and process p, $b \prec a$ for some event b at p.

The idea behind wave algorithms is that each computation gives rise to one or more decisions in which all processes have a say. An important characteristic of a wave algorithm is that it does not complete if any process p refuses to take part in its execution, because no event at p in the wave would be causally before the decide event.

Often a wave is initiated by one process, and in the end one decide event happens, at its initiator. If there can be concurrent calls of a wave algorithm, initiated by different processes, then usually for each wave the messages are marked with the ID of its initiator. In such a setting, if a wave does not complete, because a process refuses to take part, then typically another wave will complete successfully later on. Examples are Rana's termination detection algorithm (see section 6.2) and the echo algorithm with extinction for election (see section 9.3).

4.1 Traversal Algorithms

A *traversal algorithm* is a centralized wave algorithm in which the initiator sends a token through the network. After visiting all other processes, the token returns to the initiator, which then makes a decision. A typical example of a traversal algorithm is the ring algorithm, in which the token makes one trip around the ring.

Traversal algorithms can be used to build a spanning tree of the network, with the initiator as the root. Each noninitiator has as its parent the process from which it received the token for the first time.

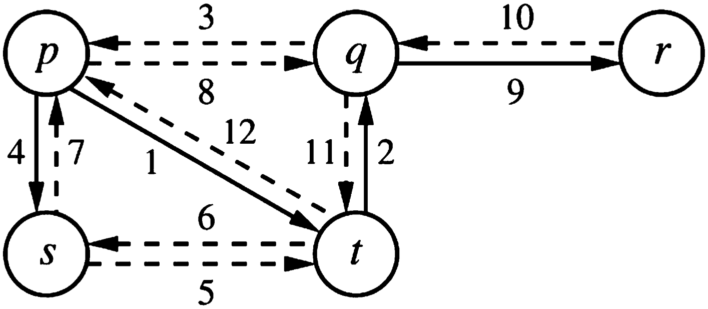
Tarry's Algorithm

Tarry's algorithm is a traversal algorithm for undirected networks. It is based on the following two rules:

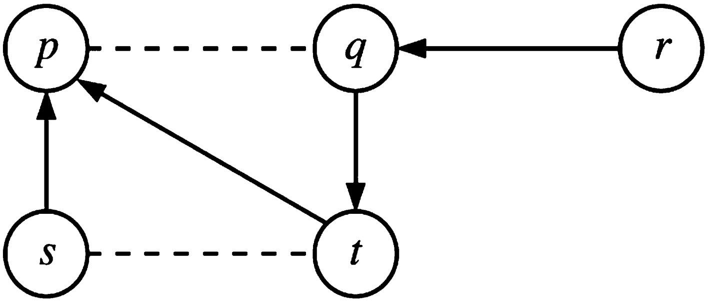
1. A process never forwards the token through the same channel twice.
2. A process only forwards the token to its parent if there is no other option.

By applying these two rules, the token travels through each channel twice and finally ends up at the initiator.

Example 4.1 We apply Tarry's algorithm to the following network; p is the initiator.



The network is undirected (and unweighted); arrows and numbers mark the consecutive steps of one possible path of the token. Solid arrows establish a parent-child relation (in the opposite direction) in the resulting spanning tree. So the resulting spanning tree of this execution is



Tree edges are solid, whereas frond edges are dashed.

We argue that in Tarry's algorithm the token traverses each channel twice, once in each direction, and finally ends up at the initiator. By rule 1, the token is never sent through the same channel in the same direction twice. Each time a noninitiator *p* holds the token, it has received the token one more time than it has sent the token to a neighbor, meaning that there is still a channel into which *p* has not yet sent the token. So, in accordance with rule 1, *p* can send the token into this channel. Hence, when Tarry's algorithm terminates, the token must be at the initiator. Suppose, toward a contradiction, that at the moment of termination some channel *pq* has not been traversed by the token in both directions; let *p* be the earliest visited process for which such a channel exists. The fact that the token started and finished at the initiator implies that the token has traversed all channels of the initiator in both directions. So *p* is a noninitiator. Since by assumption all channels of the parent of *p* have been traversed in both directions, *p* has sent the token to its parent. So, by rule 2, *p* must have sent the token into all its channels. Since *p* has sent and received the token an equal number of times, it must have received the token through all its channels. To conclude, the token has traversed *pq* in both directions, contradicting our assumption. Hence, the token must have traversed all channels in both directions.

Tarry's algorithm requires $2{\cdot}E$ messages (two messages per channel, one in either direction), and it takes at most $2{\cdot}E$ time units to terminate.

Depth-First Search

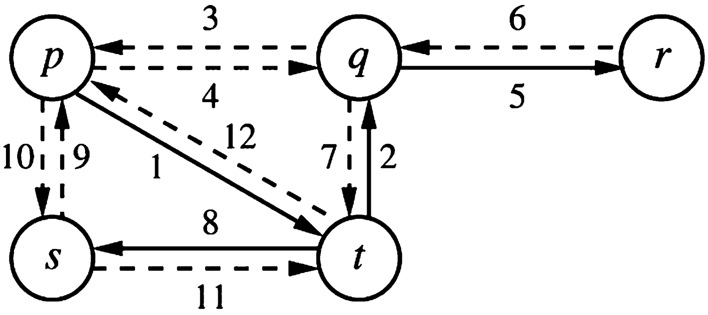
In a depth-first search, starting from the initiator, whenever possible, the token is forwarded to a process that has not yet held the token. If a process holding the token has no unvisited neighbor, then it sends the token back to its parent, being the process from which it received the token for the first time.

The spanning tree in example 4.1 is not a depth-first search tree; that is, it cannot be the result of a depth-first search. Because in a depth-first search, processes *s* and *t* would never both have *p* as their parent. In general, a spanning tree is the result of a depth-first search if all frond edges connect an ancestor with one of its descendants in the spanning tree (unlike the frond edge between *s* and *t*).

A depth-first search is obtained by adding one more rule to Tarry's algorithm:

3. When a process receives the token, it immediately sends it back through the same channel if this is allowed by rules 1 and 2.

Example 4.2 Consider the same network as in example 4.1; *p* is again the initiator. In the following picture one possible depth-first search is charted out.



In example 4.1, when *p* receives the token from *q*, it forwards it to *s*. However, here *p* is forced to send the token back to *q* immediately because of rule 3.

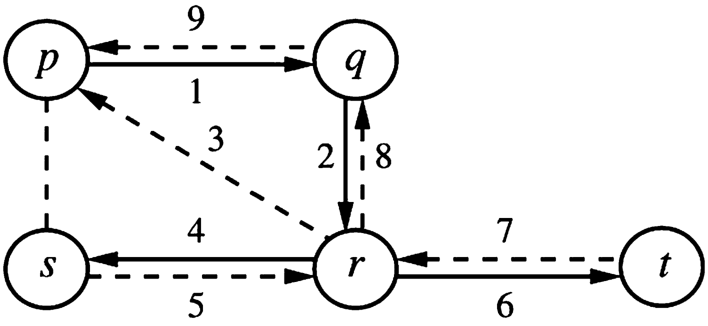
Depth-first search is a special case of Tarry's algorithm, so it also takes $2{\cdot}E$ messages, and at most $2{\cdot}E$ time units to terminate.

Sending the token back and forth through a frond edge, such as in steps 3 and 4 and steps 9 and 10 in example 4.2, constitutes a loss of time. One obvious way to avoid this is by including the IDs of visited processes in the token, so that a process can determine which neighbors have already seen the token. Since the token then travels back and forth only through the $N - 1$ tree edges, the message complexity is reduced from $2{\cdot}E$ to $2{\cdot}N - 2$ messages, and similarly the time complexity is reduced to at most $2{\cdot}N - 2$ time units. The drawback, however, is that the bit complexity goes up from $O(1)$ to $O(N{\cdot}\log N)$ (assuming that $O(\log N)$ bits are needed to represent the ID of a process).

An alternative is to let a process *p* that holds the token for the first time inform its neighbors (except the process that sent the token to *p* and the process to which *p* will send the token) that it has seen the token. In Awerbuch's depth-first search algorithm, *p* waits for acknowledgments from all those neighbors before forwarding the token, to ensure that they cannot receive the token before *p*'s information message. A process marks a channel as a frond edge as soon as it has received an information message through this channel and has received the token through another channel. A process never forwards the token through a frond edge. The worst-case message complexity goes up to $4{\cdot}E$ because frond edges carry two information messages and two acknowledgments, while tree edges carry two tokens and possibly one information and acknowledgment pair. Also, the worst-case time complexity goes up, to 4·N - 2 time units, because tree edges carry two tokens, and each process may wait at most 2 time units for acknowledgments to return.

Cidon's depth-first search algorithm improves on Awerbuch's algorithm by abolishing the wait for acknowledgments. A process p forwards the token without delay and records to which process $forward_p$ it forwarded the token last. If p receives the token back from a process $q \neq forward_p$, it dismisses the token and marks the channel pq as a frond edge. No further action from p is required, because q will eventually receive the information message from p. Then in turn q marks the channel pq as a frond edge and continues to forward the token to another process (if possible).

Example 4.3 In the following undirected network, with p as initiator, one possible computation of Cidon's depth-first search is depicted.



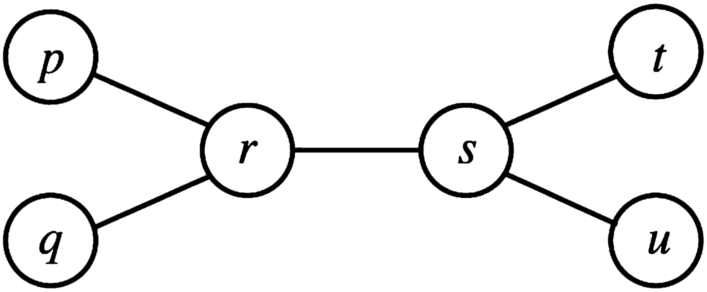
The token is forwarded by r through the frond edge pr before the information message from p reaches r. When the information message from p arrives, r continues to forward the token to s. The information message from p reaches s before the token does, so s does not send the token to p.

In Cidon's algorithm, frond edges may carry two information messages and two tokens (see exercise 4.2), so the worst-case message complexity is still 4·E. But the worst-case time complexity reduces to 2·N - 2 time units, because at least once per time unit the token is forwarded through a tree edge, and the N - 1 tree edges each carry two tokens.

4.2 Tree Algorithm

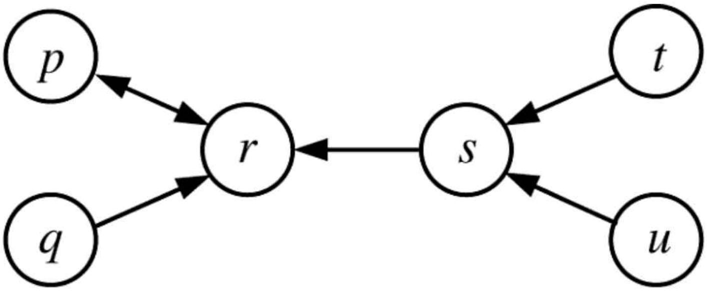
The tree algorithm is a decentralized wave algorithm for undirected, acyclic networks. A process p waits until it has received messages from all its neighbors except one. Then p makes that neighbor its parent and sends a message to it. When p receives a message from its parent, it decides. Always exactly two processes in the network decide, and these two processes consider each other their parent. Note that initially only the processes with a single neighbor can send a message.

Example 4.4 We consider one possible computation of the tree algorithm on the following network.



- p and q both send a message to r and make r their parent. Likewise, t and u both send a message to s and make s their parent.
- When the messages from t and u have arrived, s sends a message to r and makes r its parent.
- When the messages from q and s have arrived, r sends a message to p and makes p its parent.
- When p's message has arrived, r decides. Likewise, when r's message has arrived, p decides.

The parent-child relations in the terminal configuration are as follows.



We argue that in each execution of the tree algorithm, exactly two processes decide. As each process sends at most one message, each execution reaches a terminal configuration γ. Suppose, toward a contradiction, that in γ a process p has not sent any message, meaning that it did not receive a message through two of its channels, say qp

and *rp*. Since γ is terminal, *q* did not send a message to *p*, which implies it did not receive a message through two of its channels, *pq* and say *sq*, and so forth. This argument can be repeated with *s* instead of *q*, and so on. Repeating this argument over and over again, inevitably we eventually establish a cycle of processes that did not receive a message through two of their channels. This contradicts the assumption that the network topology is acyclic. So in γ each process has sent a message, which adds up to N messages in total. Since processes send a message into the only channel through which they have not yet received a message, clearly each channel carries at least one message. An acyclic network has $N - 1$ channels, so exactly one channel carries two messages. Only the two processes *t* and *u* connected by this channel decide. All events, except for the reception of *t*'s message and the decision at *u*, are causally before the decision at *t*, and likewise for the decision at *u*.

The tree algorithm is incorrect for networks that contain a cycle, because in that case the algorithm does not terminate. For instance, consider a ring of three processes. Since each process has two neighbors, it will wait for a message from one of its neighbors. Hence, all three processes wait for input, meaning that no event ever happens.

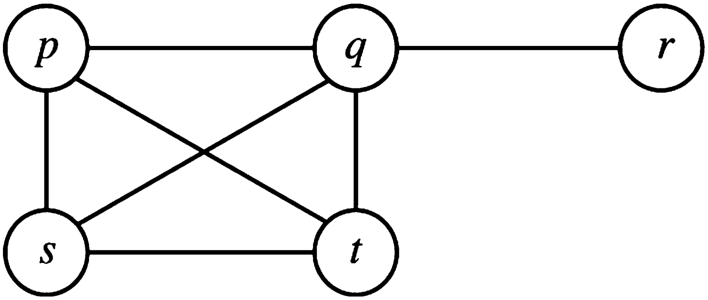
The tree algorithm takes at most $\frac{D}{2}$ time units to terminate if $D > 1$ (see exercise 4.4).

4.3 Echo Algorithm

The echo algorithm is a centralized wave algorithm for undirected networks. It underlies several of the distributed algorithms presented in the following chapters.

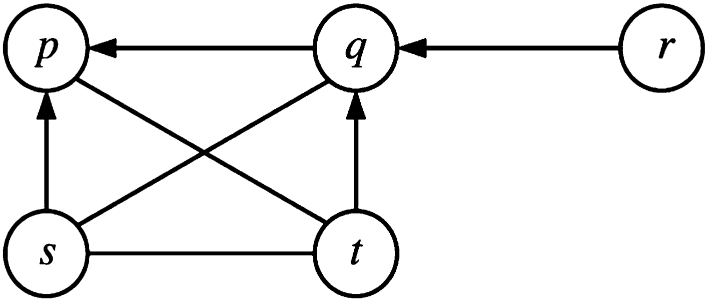
The initiator starts by sending a message to all its neighbors. Intuitively, these messages travel in all directions and bounce back from the corners of the network toward the initiator. This is achieved as follows. When a noninitiator receives a message for the first time, it makes the sender its parent and sends a message to all neighbors except its parent. When a noninitiator has received messages from all its neighbors, it sends a message to its parent. Finally, when the initiator has received messages from all its neighbors, it decides.

Example 4.5 We consider one possible computation of the echo algorithm on the following network, in which *p* is the initiator.



- *p* sends messages to *q*, *s*, and *t*.
- *p*'s message arrives at *q*, which makes *p* its parent and sends messages to *r*, *s*, and *t*.
- *q*'s message arrives at *t*, which makes *q* its parent and sends messages to *p* and *s*.
- *q*'s message arrives at *r*, which makes *q* its parent. Since *r* has no other neighbors, it sends a message to its parent *q* straightaway.
- *p*'s message arrives at *s*, which makes *p* its parent and sends messages to *q* and *t*.
- *p*'s and *s*'s messages arrive at *t*, which sends a message to its parent *q*.
- *r*'s, *s*'s, and *t*'s messages arrive at *q*, which sends a message to its parent *p*.
- *q*'s and *t*'s messages arrive at *s*, which sends a message to its parent *p*.
- *q*'s, *s*'s, and *t*'s messages arrive at *p*, which decides.

The resulting spanning tree is as follows.



We argue that the echo algorithm is a wave algorithm. Clearly, it constructs a spanning tree that covers the entire network. When a noninitiator joins this tree, upon receiving a message from its parent, it sends a message to all its other neighbors. Moreover, the initiator sends a message to all its neighbors. Hence, through each frond edge, one message travels either way. We argue by induction on the size of the network that each noninitiator eventually sends a message to its parent. Consider a leaf *p* in the spanning tree. Eventually, *p* will receive a message from all its neighbors (as only the channel to its parent is a tree edge) and send a message to its parent. When this message arrives, we can consider the network without p, in which by induction each noninitiator eventually sends a message to its parent. We conclude that, through each channel, one message travels either way. So eventually the initiator receives a message from all its neighbors and decides. All messages are causally before this decision.

In total, the echo algorithm takes $2 \cdot E$ messages, and it takes at most $2 \cdot N - 2$ time units to terminate.

Bibliographical notes

Tarry's algorithm originates from [90]. The first distributed depth-first search algorithm was presented in [22]. Awerbuch's algorithm originates from [7], and Cidon's algorithm comes from [24]. The echo algorithm stems from [20]; the presentation here is based on a slightly optimized version from [83].

4.4 Exercises

Exercise 4.1 Give an example of a computation of Awerbuch's algorithm in which an information message and an acknowledgment are communicated through the same tree edge.

Exercise 4.2 Give an example of a computation of Cidon's algorithm in which two information messages and two tokens are communicated through the same channel in the network.

Exercise 4.3 Explain how the tree algorithm can be used to compute the size of an undirected, acyclic network.

Exercise 4.4 Explain how the tree algorithm can be extended with a phase in which the decision is communicated to all processes. Furthermore, argue that the tree algorithm takes at most D time units to terminate if the time needed to communicate the decision to all processes is taken into account.

Exercise 4.5 Consider an undirected network of $N > 3$ processes $p_0, \dots, p_{N-1}$, where $p_1, \dots, p_{N-1}$ form a ring and p_0 has a channel to all other processes. (Note that this network has diameter 2.) Give a computation of the echo algorithm on this network, with p_0 as initiator, that takes (arbitrarily close to) $N + 1$ time units to complete.

Exercise 4.6 Argue that the echo algorithm takes at most $2 \cdot N - 2$ time units to terminate.

Exercise 4.7 [92] Suppose you want to use the echo algorithm in a network where duplication of messages may occur. Which modification to the algorithm should be made?

Exercise 4.8 [92] Let each process initially carry a random integer value. Adapt the echo algorithm to compute the sum of these integer values. Explain why your algorithm is correct.

5 Deadlock Detection

A process may have to wait to perform events until some other processes send or (in a synchronous setting) are ready to receive input, or until some resources have become available. A deadlock occurs if a group of processes is doomed to wait forever. This happens if each of these processes is waiting until some other process in the group either sends a message or releases a resource. The first type of deadlock is called a *communication deadlock*, while the second type is called a *resource deadlock.*

Deadlock detection is a fundamental problem in distributed computing, which requires determining a cyclic dependency within a running system. For this purpose, the global configuration of the distributed system is regularly examined by individual processes to detect whether a deadlock has occurred. That is, snapshots are taken of the global configuration of the system, and these are examined for cycles. If a deadlock is detected, the basic algorithm may be rolled back and processes may be restarted in order to remove the detected deadlock. Here we focus on detection of deadlocks and ignore rollback.

5.1 Wait-for Graphs

A *wait-for graph* depicts dependencies between processes and resources. A node in a wait-for graph can represent either a process or a resource. Both communication and resource deadlocks can be captured by what are called *N-out-of-M requests*, where $N \leq M$. For example, if a process is waiting for one message from a group of M processes, then $N = 1$; or, if a database transaction first needs to lock M data files, then $N = M$.

A nonblocked node u in a wait-for graph can issue an N-out-of-M request, meaning that it sends a request to M other nodes and becomes blocked until N of these requests have been granted. In the wait-for graph, a directed edge is drawn from u to each of the M nodes to which u issues the N-out-of-M request. Only nonblocked nodes can grant a request. Every time a node v grants u's request, the edge uv can be removed from the wait-for graph. When N requests have been granted, u becomes unblocked and informs the remaining $M - N$ nodes that u's request can be dismissed; accordingly, these $M - N$ outgoing edges of u are removed from the wait-for graph.

The following example shows how a wait-for graph can be used to model communication dependencies.

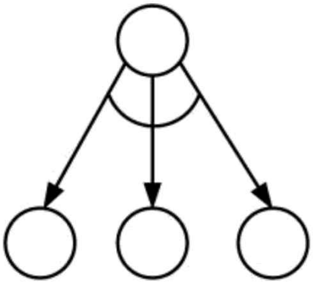
Example 5.1 Suppose process p must wait for a message from process q. In the wait-for graph, p sends a request to q; as a result, the edge pq is created and p becomes blocked. When q sends a message to p, the request from p is granted. Then the edge pq is removed from the wait-for graph, and p becomes unblocked.

The following example shows how a wait-for graph can be used to model resource dependencies.

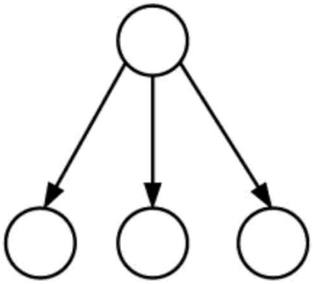
Example 5.2 Suppose two different processes p and q want to claim a resource, while at any time only one process can own the resource.

- Nodes u and v, representing p and q, respectively, send a request to node w, representing the resource. As a result, in the wait-for graph, edges uw and vw are created.
- The resource is free, and w sends a grant to, say, u, so that p can claim the resource. In the wait-for graph, the edge uw is removed.
- The resource must be released by p before q can claim it. Therefore, w sends a request to u, creating the edge wu in the wait-for graph.
- After p releases the resource, u grants the request from w. Then the edge wu is removed from the wait-for graph.
- Now w can grant the request from v, so q can claim the resource. In the wait-for graph, the edge vw is removed and the edge wv is created.

In wait-for graphs, an M-out-of-M request with $M > 1$ (also called an AND request) is drawn with an arc through the M edges, while a 1-out-of-M request (also called an OR request) is drawn without an arc. For example, for $M = 3$,



AND (3-out-of-3) request



OR (1-out-of-3) request

The examples in this chapter do not contain any N-out-of-M requests with $1 < N < M$.

5.2 Bracha-Toueg Algorithm

To try to detect a deadlock with regard to an ongoing execution of a basic algorithm, first a snapshot can be taken of the corresponding wait-for graph. A process that suspects it is deadlocked starts a Lai-Yang snapshot to compute the wait-for graph (see section 3.2). To distinguish between subsequent snapshots, snapshots (and their control messages) are tagged with a sequence number. Each node u takes a local snapshot to determine the requests it sent or received that were not yet granted or dismissed, taking into account the grant and dismiss messages in the channel states of its incoming edges. Then it computes two sets of nodes:

- Out_u: the nodes to which u has sent requests that have not yet been granted or dismissed.
- In_u: the nodes from which u has received requests that have not yet been granted or dismissed.

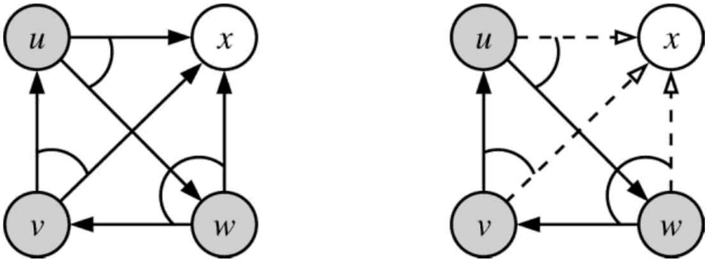
The following analysis on the computed wait-for graph, which cleans out the graph as much as possible, may reveal deadlocks:

- Nonblocked nodes in the wait-for graph can grant requests.
- When a request has been granted, the corresponding edge in the wait-for graph is removed.
- When a node u with an outstanding N-out-of-M request has received N grants, u becomes unblocked. The remaining $M - N$ outgoing edges of u in the wait-for graph are removed.

When no more grants are possible, nodes that are still blocked in the wait-for graph are deadlocked in the snapshot of the basic algorithm.

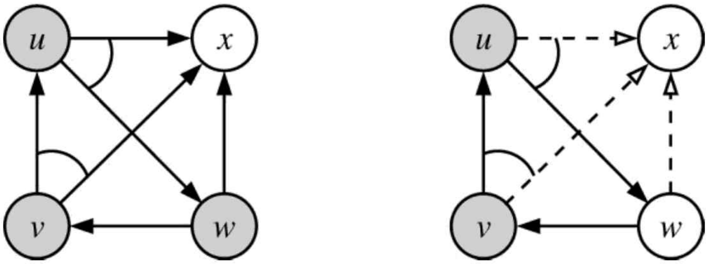
We consider two examples. Granted requests are drawn as dashed arrows (they are no longer part of the wait-for graph).

Example 5.3 The next wait-for graph contains three 2-out-of-2 requests. Blocked nodes are colored gray.

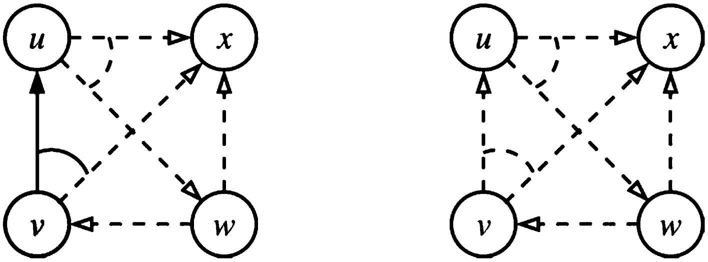


The unblocked node *x* grants the three incoming requests. After that, the three other nodes remain blocked, so no other requests can be granted. Hence, these three nodes are deadlocked.

Example 5.4 The next wait-for graph contains two 2-out-of-2 requests and one 1-out-of-2 request.



The unblocked node *x* grants the three incoming requests. As a result, node *w* becomes unblocked; it dismisses its remaining request to *v* and grants the incoming request from *u*. Next, node *u* becomes unblocked; it grants the last pending request in the graph.



Finally, all nodes have become unblocked, so no nodes are found to be deadlocked.

Let the basic algorithm run on an undirected network, and suppose a wait-for graph has been computed. The Bracha-Toueg deadlock detection algorithm provides a distributed method to perform the analysis whereby the wait-for graph is cleaned out to try to find deadlocks. The nodes in the wait-for graph start to resolve grants, in the manner described earlier. Initially, $requests_v$ is the number of grants node v requires to become unblocked in the wait-for graph. When $requests_v$ is or becomes 0, v sends grant messages to all nodes in In_v. When v receives a grant message, $requests_v \leftarrow requests_v - 1$. If after termination of this deadlock detection run $requests_u > 0$ at the initiator u, then it is deadlocked in the basic algorithm.

A key question is how to determine that deadlock detection has terminated. In principle, nodes could apply a termination detection algorithm from chapter 6. However, the Bracha-Toueg algorithm is designed in such a way that termination detection comes for free. We now explain in detail how the nodes choreograph cleaning out the wait-for graph. Initially, $notified_v = false$ and $free_v = false$ at all nodes v; these two variables ensure that v executes at most once the routines $Notify_v$ and $Grant_v$, respectively, given in the pseudocode that follows. The initiator u of deadlock detection starts the resolution of grants throughout the wait-for graph by executing $Notify_u$. It consists of sending a **notify** message into all outgoing edges, and executing $Grant_u$ if $requests_u = 0$. Noninitiators v that receive a **notify** message for the first time execute $Notify_v$. Moreover, nodes v that are or become unblocked, meaning that $requests_v = 0$, grant all pending requests by executing $Grant_v$. The pseudocode for the procedure $Notify_v$ is as follows.

```
$notified_v \leftarrow true$;
send ⟨notify⟩ to all $w \in Out_v$;
if $requests_v = 0$ then
    perform procedure $Grant_v$;
end if
await ⟨done⟩ from all $w \in Out_v$;
```

And the pseudocode for the procedure $Grant_v$ is the following.

```
$free_v \leftarrow true$;
send ⟨grant⟩ to all $w \in In_u$;
await ⟨ack⟩ from all $w \in In_u$;
```

Note that since $Grant_v$ is a subcall of $Notify_v$, waiting for **ack** messages postpones the sending of **done** messages. The **done** (and **ack**) messages are used for termination detection. That is, when the initiator u has received a **done** message from all nodes in Out_u, it checks the value of $free_u$. If it is still *false*, u concludes that it is deadlocked.

While a node is awaiting **done** or **ack** messages, it can process incoming **notify** and **grant** messages. When a node v receives a **notify** message from a neighbor w, it does the following.

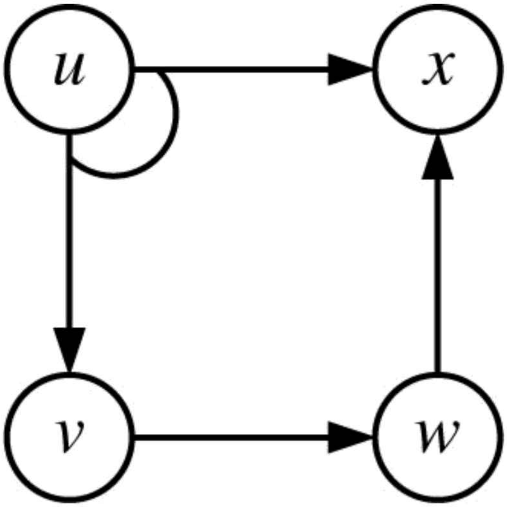
```
if $notified_v = false$ then
    perform procedure $Notify_v$;
end if
send ⟨done⟩ to $w$;
```

When a node v receives a **grant** message from a neighbor w, it does the following.

```
if $requests_v > 0$ then
    $requests_v \leftarrow requests_v - 1$;
    if $requests_v = 0$ then
        perform procedure $Grant_v$;
    end if
end if
send ⟨ack⟩ to $w$;
```

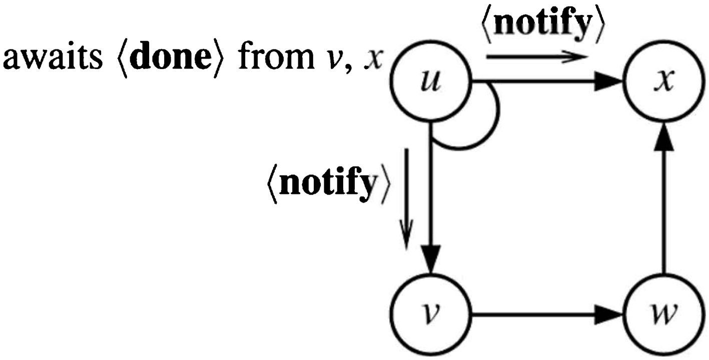
Note that if v receives a **notify** message and $notified_v = true$ (meaning that v already executed $Notify_v$), or a **grant** message and the assignment $requests_v \leftarrow requests_v - 1$ does not set $requests_v$ from 1 to 0 (meaning that v does not become unblocked by the **grant** message), then v immediately sends back a **done** or **ack** message, respectively.

Example 5.5 Suppose the following wait-for graph, consisting of one 2-out-of-2 request and two 1-out-of-1 requests, has been computed in a snapshot. Initially, $requests_u = 2$, $requests_v = requests_w = 1$, and $requests_x = 0$.

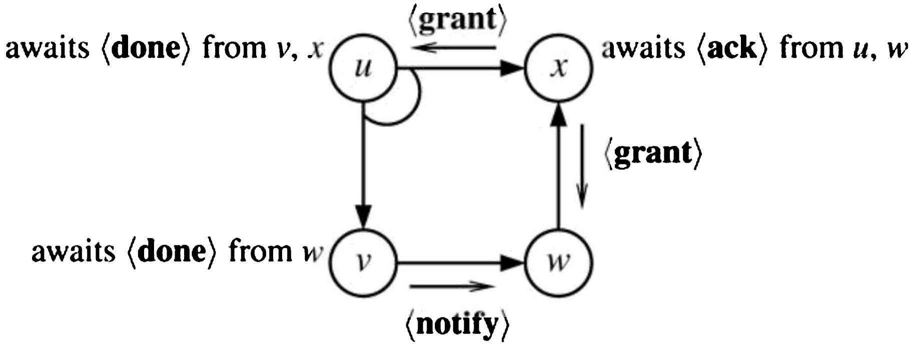


We consider one possible computation of the Bracha-Toueg algorithm.

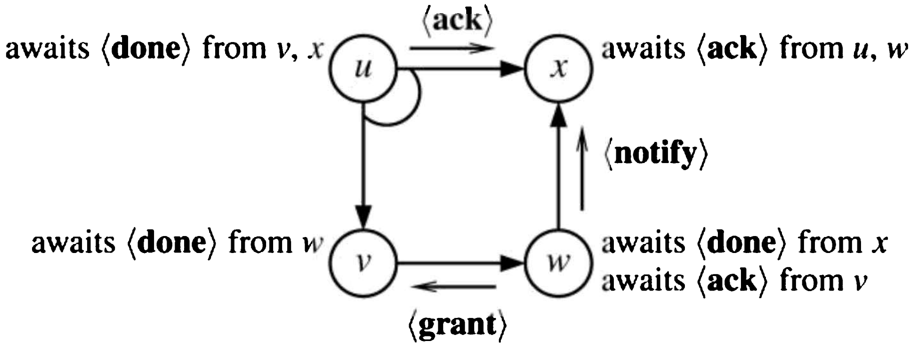
- Initiator u starts by sending a **notify** to v and x, and must now await a **done** from both v and x before it can examine $requests_u$ to see whether it is deadlocked.



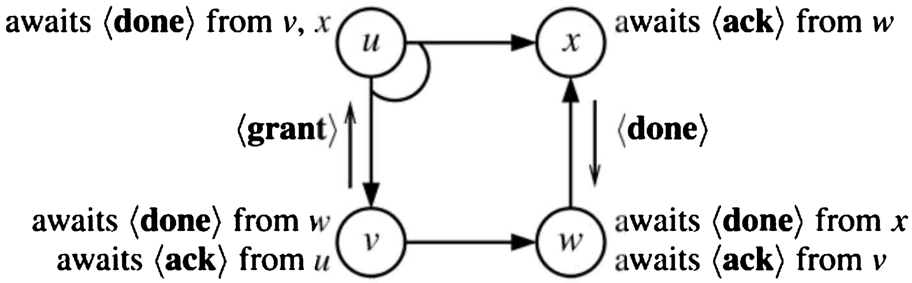
- The **notify** from u is received at v, which sends a **notify** to w and must await a **done** from w before it can send a **done** back to u. Concurrently, the **notify** from u is received at x, which sends a **grant** to u and w, because $requests_x = 0$, and must await an **ack** from both u and w before it can send a **done** back to u.



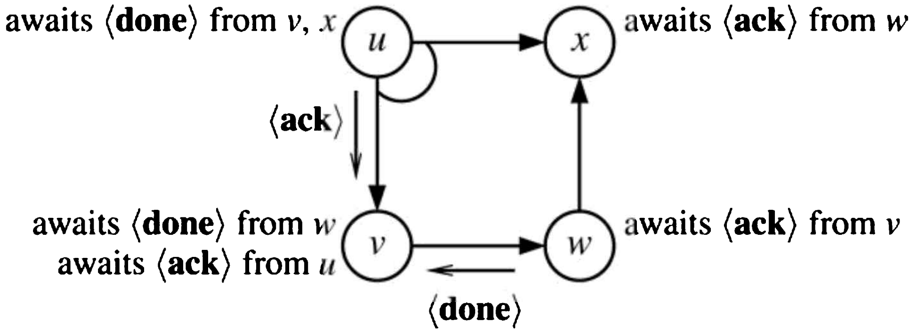
- The **notify** from *v* is received at *w*, which sends a **notify** to *x* and must await a **done** from *x* before it can send a **done** to *v*. Concurrently, the **grant** from *x* is received at *u*, which sends an **ack** back to *x* immediately, because the **grant** decreases $requests_u$ from 2 to 1. Next, the **grant** from *x* is received at *w*, which sends a **grant** to *v*, because $requests_w$ decreases from 1 to 0, and must await an **ack** from *v* before it can send an **ack** back to *x*.



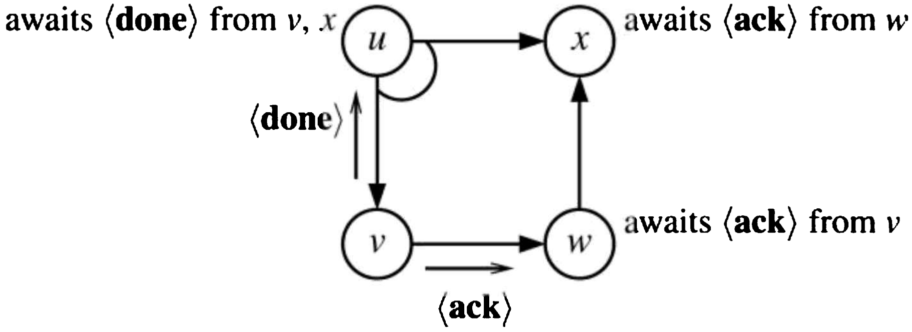
- The **notify** from *w* is received at *x*, which sends a **done** back to *w* immediately. Next, the **ack** from *u* is received at *x*, so that *x* only needs an **ack** from *w* in order to send a **done** to *u*. Concurrently, the **grant** from *w* is received at *v*, which sends a **grant** to *u*, because $requests_v$ decreases from 1 to 0, and must await an **ack** from *u* before it can send an **ack** back to *w*.


awaits ⟨**done**⟩ from v, x
u
x
awaits ⟨**ack**⟩ from w
⟨**grant**⟩
⟨**done**⟩
awaits ⟨**done**⟩ from w
awaits ⟨**ack**⟩ from u
v
w
awaits ⟨**done**⟩ from x
awaits ⟨**ack**⟩ from v

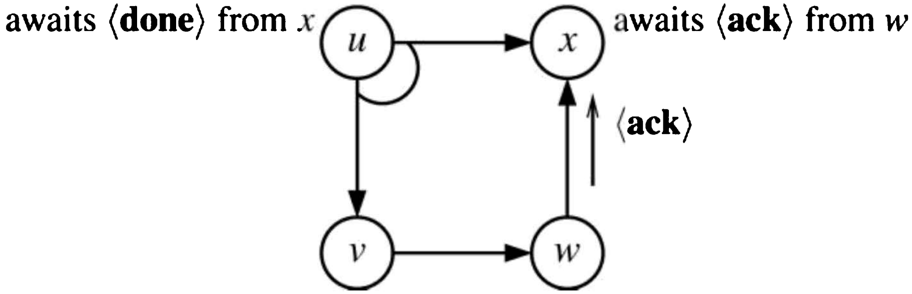
- The **done** from x is received at w, which can now send a **done** to v. Concurrently, the **grant** from v is received at u, which decreases $requests_u$ from 1 to 0 and sends an **ack** back to v immediately, because there are no requests for u to grant.


awaits ⟨**done**⟩ from v, x
u
x
awaits ⟨**ack**⟩ from w
⟨**ack**⟩
awaits ⟨**done**⟩ from w
awaits ⟨**ack**⟩ from u
v
w
awaits ⟨**ack**⟩ from v
⟨**done**⟩

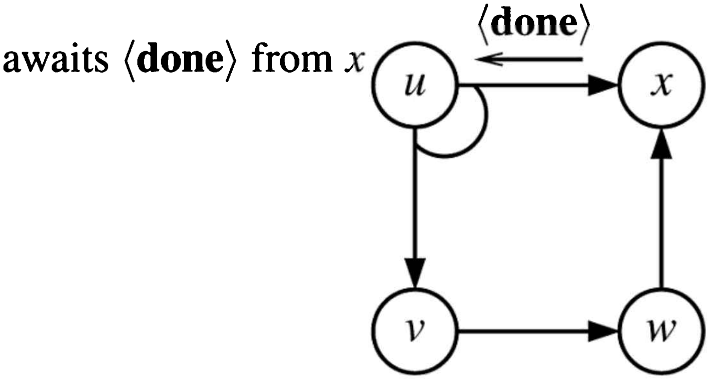
- The **done** from w is received at v, which can now send a **done** to u. Next, the **ack** from u is received at v, which can now send an **ack** to w.


awaits ⟨**done**⟩ from v, x
u
x
awaits ⟨**ack**⟩ from w
⟨**done**⟩
v
w
awaits ⟨**ack**⟩ from v
⟨**ack**⟩

- The **done** from v is received at u, which now only awaits a **done** from x before it can examine $requests_u$ to see whether it is deadlocked. Concurrently, the **ack** from v is received at w, which can now send an **ack** to x.



- The **ack** from w is received at x, which can now send a **done** to u.



- The **done** from x is received at u, which now examines $requests_u$, finds that its value is 0, and concludes that it is not deadlocked.

We argue that when the initiator completes its *Notify* call, the Bracha-Toueg algorithm has terminated. The idea is that we can distinguish between two types of trees. First, by assuming that each node receiving a **notify** message for the first time makes the sender its parent, we obtain a tree T rooted in the initiator. The **notify/done** messages construct T and travel through (part of) the network similarly as in the echo algorithm (see section 4.3). Second, by assuming that each node receiving a **grant** message that sets *requests* to 0 makes the sender its parent, we obtain disjoint trees T_v (that may overlap with T), each rooted in a node v where from the start $requests_v = 0$. Again, the **grant/ack** messages construct T_v and travel through the network similarly as in the echo algorithm. A noninitiator v that is the root of a tree T_v only sends a **done** to its parent in T when all **grant** messages sent by nodes in T_v have been acknowledged. This implies that when the initiator completes its *Notify* call, not only all **notify** messages but also all **grant** messages in the network have been acknowledged.

We argue that the Bracha-Toueg algorithm is deadlock-free. That is, the initiator will eventually complete its *Notify* call. Replying with a **done** (to a **notify**) or **ack** (to a **grant**) is delayed by a node u only if it is executing $Grant_u$ because $requests_u$ is 0 (in the case of **done**) or has been made 0 by the **grant** (in the case of an **ack**), and u is awaiting **ack** messages. We note that there cannot be a cycle of nodes that sent a **grant** to the next node and must wait before sending an **ack** to the previous node in the cycle. Such a cycle would always contain a node v for which $requests_v$ was not set to 0 by a **grant** from a node in this cycle. This implies that some node will always be able to respond to a pending **notify** or **grant**.

As noted, if after resolving the wait-for graph the initiator remains blocked, then it is deadlocked in the snapshot. Because the Bracha-Toueg algorithm cleans out the part of the wait-for graph that is reachable from the initiator as much as possible. So if the initiator remains blocked, this means it is part of a cycle of nodes waiting for each other.

In the case of a communication deadlock (see example 5.1), the other direction also holds: if the initiator is deadlocked when the snapshot is taken, then it will remain blocked in the wait-for graph. In the case of resource deadlock, this only holds if resource requests are granted nondeterministically (see exercise 5.7). Because, as shown in example 5.2, modeling resource deadlock means that removing one edge (in the example, wu) may automatically produce another edge (in the example, wv). The Bracha-Toueg approach to resolving edges in wait-for graphs does not take into account this automatic creation of edges.

Bibliographical notes

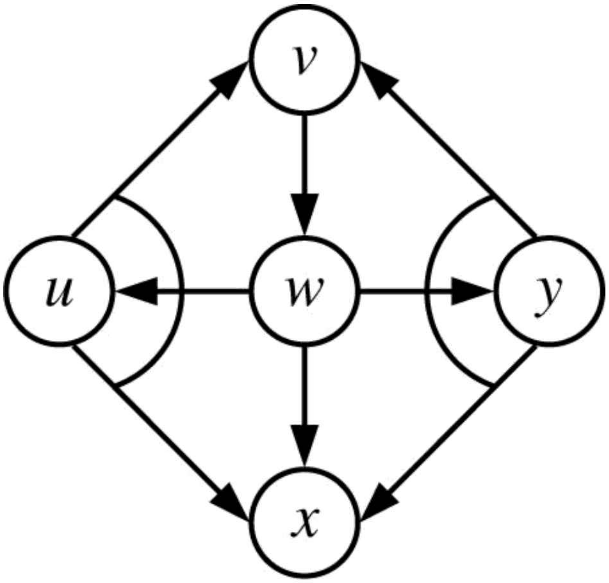
The Bracha-Toueg algorithm originates from [14].

5.3 Exercises

Exercise 5.1 Draw the wait-for graph for the initial configuration of the tree algorithm, applied to an undirected cycle of three nodes.

Exercise 5.2 Give one possible computation of the Bracha-Toueg algorithm on the wait-for graph in example 5.5, with v as initiator.

Exercise 5.3 Let node u initiate a deadlock detection run in which the following wait-for graph is computed.



Give one possible computation of the Bracha-Toueg algorithm on this wait-for graph.

Exercise 5.4 Let node *u* initiate a deadlock detection run in which the wait-for graph from the previous exercise is computed, with the only difference that *w* is waiting for a 2-out-of-3 (instead of a 1-out-of-3) request. Give one possible computation of the Bracha-Toueg algorithm.

Exercise 5.5 Give a computation on a wait-for graph in which $free_u$ remains *false* for some noninitiator *u* after running the Bracha-Toueg algorithm, while *u* is not deadlocked in the basic algorithm.

Exercise 5.6 Suppose node *u* sends a request to node *v*, then dismisses this request, and next sends another request to *v*. Let the dismiss message reach *v* first, then the second request, and finally the first request. How should *v* process these three messages?

Exercise 5.7 Suppose that the order in which resource requests are granted is predetermined. Give an example of a snapshot with a resource deadlock that is not discovered by the Bracha-Toueg algorithm. Show, moreover, that if the selection of which resource request is granted is performed in a nondeterministic fashion, then the deadlock in your example may be avoided.

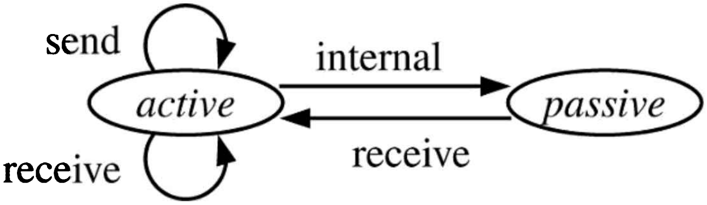
6

Termination Detection

In the previous chapter, we looked at deadlocks, in which some processes are doomed to wait forever. In the current chapter, we turn our attention to the related problem of termination. A distributed algorithm is *terminated* if all processes are passive and no (basic) messages are in transit. Termination detection is a fundamental and challenging problem in distributed computing because no process has complete knowledge of the global configuration of the network. Moreover, a passive process may be reactivated by a message from another process, and the absence of messages in the network must be established.

The basic algorithm is the algorithm for which termination is being detected, and the control algorithm is the termination detection algorithm employed for this task. The control algorithm in general consists of two parts: termination detection and announcement. The announcement part, called *Announce*, is straightforward; we therefore focus on the termination detection part. Ideally, termination detection does not require freezing the basic execution.

From the viewpoint of the control algorithm, a simple description of the process states in the basic algorithm suffices. A process is said to be *passive* if the only basic event it can perform is a receive event, upon which it becomes active. A process is *active* if it is not passive. So initially the initiators of the basic algorithm are active and the noninitiators are passive. We are, moreover, interested in send and receive events, to determine whether there are basic messages under way. As for internal events, we are only interested in those that change a process state from active to passive. So the abstract view of states of processes with regard to the basic algorithm is as follows.



We will consider termination detection techniques based on maintaining trees of active processes (section 6.1), waves tagged with logical clock values (section 6.2), token-based traversal (section 6.3), and dividing a fixed weight between the active processes and basic messages in transit (sections 6.4 and 6.5).

6.1 Dijkstra-Scholten Algorithm

The Dijkstra-Scholten algorithm is a termination detection algorithm for a centralized basic algorithm on an undirected network. The idea is to build a tree, rooted in the initiator of the basic algorithm, that contains all active processes, and passive ones that have active descendants in the tree. If a basic message from a process p makes a process q active, then q becomes a child of p in the tree. A process can quit the tree only if it is passive and has no children left in the tree. In that case, it informs its parent that it is no longer a child. Termination is detected by the initiator when the tree has disappeared.

To be more precise, initially the tree T consists of only the initiator. Each process p maintains a child counter cc_p that estimates from above its number of children in T. Initially, this counter is zero at all processes. The control algorithm works as follows. When a process p sends a basic message m, it increases cc_p by 1, because the receiver q may become active upon receiving m. When the message arrives at q, either q joins T with parent p if q was not yet in T or otherwise q sends an acknowledgment to p that it is not a new child. Upon receiving an acknowledgment, p decreases cc_p by 1. When a process in T is passive and its counter is zero, it quits T. When a noninitiator quits T, it sends an acknowledgment to its parent that it is no longer its child. Finally, when the initiator quits T, it calls *Announce*.

Example 6.1 We consider one possible computation of a basic algorithm supplied with the Dijkstra-Scholten algorithm on an undirected ring of three processes p, q, and r.

- At the start, the initiator p sends basic messages to q and r; it sets cc_p to 2. Upon receipt of these messages, q and r both become active and join T with parent p.

- q sends a basic message to r; it sets cc_q to 1. Upon receipt of this message, r sends back an acknowledgment, which causes q to decrease cc_q to 0.
- p becomes passive. (Since $cc_p = 2$, it remains in T.)
- r becomes passive. Since $cc_r = 0$, it sends an acknowledgment to its parent p, which causes p to decrease cc_p to 1.
- q sends a basic message to r; it sets cc_q to 1.
- q becomes passive. (Since $cc_q = 1$, it remains in T.)
- Note that all three processes are now passive, but there is still a basic message traveling from q to r. Upon receipt of this message, r becomes active again and joins T with parent q.
- r becomes passive. Since $cc_r = 0$, it sends an acknowledgment to its parent q, which causes q to decrease cc_q to 0.
- Since q is passive and $cc_q = 0$, it sends an acknowledgment to its parent p, which causes p to decrease cc_p to 0.
- Since p is passive and $cc_p = 0$, it calls *Announce*.

When all processes have become passive and all basic messages have been acknowledged, clearly the tree T will eventually disappear, after which the initiator will call *Announce*. Conversely, active processes and processes that sent a basic message that is still in transit are guaranteed to be in T, which implies that the initiator will only call *Announce* when the basic algorithm has terminated.

The Shavit-Francez algorithm generalizes the Dijkstra-Scholten algorithm to decentralized basic algorithms. The idea is to maintain not one tree but a forest of (disjoint) trees, one for each initiator. Initially, each initiator constitutes a tree in the forest. A process can only join a tree if it is not yet in a tree in the forest. For the rest, the algorithm proceeds exactly like the Dijkstra-Scholten algorithm. The only distinction is that when an initiator detects that its tree has disappeared, it cannot immediately call *Announce*. Instead, the initiator starts a wave, tagged with its ID, in which only processes that are not in a tree participate and the decide event calls *Announce*. If such a wave does not complete, this is not a problem, because then another initiator for which the tree has not yet disappeared will start a subsequent wave. And the last tree to disappear is guaranteed to start a wave that will complete.

Example 6.2 We consider one possible computation of a basic algorithm supplied with the Shavit-Francez algorithm on an undirected ring of three processes p, q, and r.

- At the start, the initiators p and q both send a basic message to r; they set cc_p and cc_q to 1. Next, p and q become passive.
- Upon receipt of the basic message from p, r becomes active and makes p its parent. Next, r receives the basic message from q and sends back an acknowledgment, which causes q to decrease cc_q to 0.
- Since q became passive as the root of a tree and $cc_q = 0$, it starts a wave. This wave

does not complete, because p and r refuse to participate.

- r sends a basic message to q; it sets cc_r to 1. Next, r becomes passive.
- Upon receipt of the basic message from r, q becomes active and makes r its parent. Next, q becomes passive and sends an acknowledgment to its parent r, which causes r to decrease cc_r to 0.
- Since r is passive and $cc_r = 0$, it sends an acknowledgment to its parent p, which causes p to decrease cc_p to 0.
- Since p became passive as the root of a tree and $cc_p = 0$, it starts a wave. This wave completes, so p calls *Announce*.

6.2 Rana's Algorithm

Rana's algorithm detects termination for a decentralized basic algorithm on an undirected network. It exploits waves that carry a clock value provided by a logical clock (see section 2.2). Each basic message is acknowledged, so that a process can determine whether each basic message it sent has reached its destination.

To understand Rana's algorithm, it is helpful first to consider an incorrect termination detection algorithm, which uses waves without clock values. Let a process become *quiet* if (1) it is passive and (2) all the basic messages it sent have been acknowledged. Then it starts a wave, tagged with its ID. Only quiet processes take part in this wave. If a wave completes, its initiator calls *Announce*. Note that there can be multiple concurrent waves; each process must keep track of its state in each of these waves.

The problem with this termination detection algorithm is expressed by the following scenario. Let a process q that has not yet been visited by a wave send a basic message to a quiet process p that already took part in the wave, making p active again. Next, q receives p's acknowledgment, becomes quiet, and takes part in the wave. Then the wave can complete while p is active.

To avoid this scenario, a logical clock provides each event with a time stamp. The time stamp of a process is the highest time stamp of its events so far (initially it is zero). As noted, each basic message is acknowledged, and a process becomes quiet if (1) it is passive and (2) all the basic messages it sent have been acknowledged. Then it starts a wave, tagged with its ID *and its time stamp t*. Only quiet processes *that have been quiet from some logical time $\leq t$ onward* take part in this wave. If a wave completes, its initiator calls *Announce*.

Let us revisit the problematic scenario, but now for Rana's algorithm. When p is visited by the wave, its clock time is set beyond the time stamp of the wave. Therefore, the acknowledgment from p sets q's clock time beyond the time stamp of the wave. Hence, q will refuse to take part in the wave.

Actually, Rana's algorithm does not require a full-blown logical clock. It suffices that only the control messages (acknowledgments and wave messages) be taken into account. That is, a wave tagged with time stamp t puts the clock of each recipient to t (if its value is not $\geq t$ already), and each acknowledgment is tagged with the time stamp t' of the sender and puts the clock of the receiver to $t' + 1$ (if its value is not $\geq t' + 1$ already).

Example 6.3 We consider one possible computation of a basic algorithm supplied with Rana's algorithm, using logical time stamps, on an undirected ring of three processes p, q, and r.

- Initially, p, q, and r all have logical time 0, and only p is active. It sends basic messages m_1 to q and m_2 to r. The corresponding receive events of these messages make q and r active. Next, they send acknowledgments $\langle a_1, 0\rangle$ and $\langle a_2, 0\rangle$, respectively, to p.
- p and q become passive. Moreover, p receives both acknowledgments, setting its time to 1. Next, p and q both start a wave, tagged with 1 and 0, respectively. The wave of p first visits q, setting its time to 1; q takes part in the wave, because it is quiet from time 0 onward. The wave of q first visits r, which refuses to take part in the wave, because it is active.
- r sends a basic message m_3 to q. Upon receipt of this message, q becomes active and sends back an acknowledgment $\langle a_3, 1\rangle$. When r receives this acknowledgment, its clock value becomes 2.
- q and r become passive. Next, r refuses to take part in p's wave, because r is quiet from time 2 onward, while the wave is tagged with 1.
- q and r both start a wave, tagged with 1 and 2, respectively. The wave of r completes, and r calls *Announce*.

We argue that Rana's algorithm is a correct termination detection algorithm. When the basic algorithm has terminated, Rana's algorithm will eventually call *Announce*. Because each process eventually becomes quiet, when all the basic messages it sent have been acknowledged; and when a process becomes quiet, it starts a wave. Suppose a wave, tagged with some time stamp t, does not complete. Then some process does not take part in the wave, which means that at some time $> t$ it is not quiet. When that process becomes quiet, it will start another wave, tagged with some $t' > t$. This implies that when all processes have become quiet, eventually some wave, with the largest time stamp among all waves, is guaranteed to complete.

Conversely, when Rana's algorithm calls *Announce*, the basic algorithm has terminated. Let a wave complete. Suppose, toward a contradiction, that at that moment the basic algorithm has not terminated. Then some process is active, or some basic message is in transit. Since only quiet processes take part in a wave, such a situation can arise only if a quiet process p was first visited by the wave and then made active again by a basic message m from a process q that was not yet visited by the wave. Note that q can take part in the wave only after it has received an acknowledgment for *m* from *p*. Let the wave be tagged with time stamp *t*. When *p* takes part in the wave, its logical time becomes at least *t*. So the acknowledgment from *p* to *q* sets the logical time of *q* to a value greater than *t*. However, this means that *q* is not quiet from a logical time ≤ *t*, so it cannot take part in the wave. This contradicts the fact that the wave completes. So at the moment the wave completes, the basic algorithm must have terminated.

6.3 Safra's Algorithm

Safra's algorithm is a traversal-based termination detection algorithm. The initiator of the control algorithm sends out a token that visits every process in the network. A process can only forward the token when it is passive. When the token returns to the initiator, it is decided on the basis of information in the token whether the basic algorithm has terminated. If not, the token is sent out again. Although Safra's algorithm is centralized, the basic algorithm can be decentralized. The network can be directed. Note that a traversal of the entire network is always feasible, owing to the fact that networks are assumed to be strongly connected.

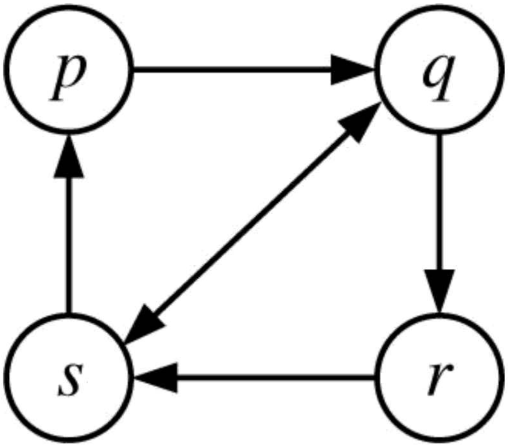
There are two complications. First, it must be determined whether there are basic messages in transit. In the case of a directed network, there is no simple acknowledgment scheme for basic messages. The second complication is that a traversal of passive processes does not guarantee termination. A traversal-based algorithm can give rise to an execution similar to the counterexample to the flawed termination detection algorithm presented at the start of section 6.2. An active process *q* may make a passive process *p* that was already visited by the token active again, after which *q* becomes passive and forwards the token.

To cope with these complications, in Safra's algorithm, every process maintains a counter of type integer; initially it is zero. At each outgoing or incoming basic message, the counter is increased or decreased, respectively. During each round trip, the token carries the sum of the counters of the processes it has traversed.

At any time, the sum of the counters of all processes in the network is nonnegative, and it is zero if and only if no basic message is in transit. Still, the token may compute a negative sum for a round trip if a passive process *p* that already forwarded the token receives a basic message, becomes active, and sends basic messages that are received by a process *q* before this same token arrives. Note that then the receipt of these messages at *q* is accounted for in the token, but not the sending at *p*.

This scenario is dealt with by coloring processes black after receiving a basic message. Initially, all processes are white. When the initiator of the control algorithm becomes passive, it sends a white token, carrying integer value zero, and the initiator becomes white. Suppose a process receives the token. If both the process and the token are white, then the process adds its counter value to the integer value of the token. A noninitiator must wait until it is passive and then forwards the token. A white noninitiator leaves the color of the token unchanged; a black noninitiator colors the token black, while the process itself becomes white. Eventually the token returns to the initiator. If the token and the initiator are both white and the integer value of the token is zero, then the initiator calls *Announce*. Otherwise the initiator waits until it is passive, sends a white token again carrying the integer value zero, and becomes white.

Example 6.4 We consider one possible computation of a basic algorithm supplied with Safra's algorithm on the following directed network.



Initially, the token is at the initiator p, only s is active, all processes are white with counter 0, and there are no messages in transit. First, s sends a basic message m to q, setting the counter of s to 1. Now s becomes passive. The token travels around the network and returns to p, white with sum 1. Next, the token travels on from p via q to r, white with sum 0. The message m travels to q and back to s, making them active and black, with counter 0. Now s becomes passive and the token travels from r via s to p, black with sum 0. Note that s becomes white. Finally, q becomes passive, and after two more round trips of the token (one round trip is needed to color q white), p calls *Announce*.

When there are no basic messages in transit, the counters of the processes sum to zero. When, moreover, all processes have become passive, the token will color them all white. So when the basic algorithm has terminated, the token will eventually return to the initiator colored white and with sum zero, after which the initiator calls *Announce*.

Conversely, suppose a token returns to the initiator colored white and with sum zero. This token was only forwarded by processes while they were passive and white. The fact that the token is white implies that if the receipt of a basic message is included in the integer value of the token, then the sending of this message is also included in this integer value. So, since the token has sum zero, it follows that no basic messages are in transit and no process was made active after the token's visit. Hence, the basic algorithm has terminated.

An optimization of Safra's algorithm does away with black tokens. Instead, when a black process p holds the token, it dismisses the token. As soon as p is passive, it becomes white and sends a fresh token, carrying the integer value zero and tagged with p's ID. Suppose the token completes the round trip, meaning that it reaches p, tagged with p. As soon as p is passive, it checks whether p is white and the token's integer value is zero. If so, p calls *Announce*. Otherwise p dismisses the token, becomes white, and sends a fresh token, as explained earlier.

6.4 Weight Throwing

In weight-throwing termination detection, for a centralized basic algorithm on a directed network, the initiator is given a certain amount of weight. During a computation, this weight is divided among active processes and basic messages in transit. Crucially, the total amount of weight in the entire network must stay unchanged during the computation. Every time a basic message is sent, the sender transfers some (but not all) of its weight to the message. And every time a basic message is received, the receiver adds the weight in this message to its own weight. When a noninitiator becomes passive, it returns its weight to the initiator (possibly via some other processes if there is no direct channel to the initiator). When the initiator is passive and has regained its original weight, it calls *Announce*.

Example 6.5 We consider one possible computation of a basic algorithm supplied with the weight-throwing algorithm on an undirected ring of three processes p, q, and r.

- At the start, the initiator p has weight 12. It sends basic messages to q and r, with weights 6 and 3, respectively, and reduces its own weight to 3. Upon receipt of these messages, q and r become active, with weights 6 and 3.
- q sends a basic message to r, with weight 3, and reduces its own weight to 3. Upon receipt of this message, r increases its weight to 6.
- p becomes passive. (Since it has weight less than 12, it does not yet call *Announce*.)
- r becomes passive and sends a control message to p, returning its weight 6. Upon receipt of this message, p increases its weight to 9.
- q sends a basic message to r, with weight 1.5, and reduces its own weight to 1.5.
- q becomes passive and sends a control message to p, returning its weight 1.5. Upon receipt of this message, p increases its weight to 10.5.
- Note that all three processes are now passive, but there is still a message traveling from q to r. Upon receipt of this message, r becomes active again, with weight 1.5.
- r becomes passive again. It sends a control message to p, returning its weight 1.5.

Upon receipt of this message, p increases its weight to 12. Since p is passive, it calls *Announce*.

When all processes have become passive and there are no basic messages in transit, clearly all weight will eventually be returned to the initiator, which will then call *Announce*. Conversely, since all active processes and basic messages carry some weight, it is guaranteed that the initiator detects termination only when all processes are passive and there are no basic messages in transit.

The Achilles' heel of this simple and effective termination detection scheme is *underflow*: the weight at a process can become too small to be divided further. Two solutions have been proposed for this problem.

The first solution is that a process p where underflow occurs gives itself extra weight. If p is a noninitiator, it must send a control message to the initiator that more weight has been introduced in the system or else the initiator could call *Announce* prematurely. To avoid race conditions, p must wait for an acknowledgment from the initiator before it can continue sending basic messages. Otherwise the initiator could regain its original weight before the control message from p has reached it.

The second solution is that p starts a weight-throwing termination detection subcall. Then p only returns its weight to the initiator when it has become passive and its subcall has terminated. The weights originating from the initiator and from p must be recorded separately at the processes as well as in the basic messages.

6.5 Fault-Tolerant Weight Throwing

As in section 3.3 on rollback recovery, we now turn our attention to the situation where processes may crash, meaning that they stop executing unexpectedly and never resume execution; this topic will be studied in depth in chapter 12. We adapt the weight-throwing algorithm from the previous section such that it can cope with crash failures. As in section 3.3, we assume that the network topology is complete, so that strong connectedness is preserved after crash failures, and that each process owns a failure detector that eventually picks up on every crash failure; see section 12.3 for an in-depth treatment of such devices.

The possibility of crash failures poses three serious problems to the weight-throwing algorithm. The first problem is that when a process crashes, this process or a basic message traveling to it may hold some weight. This weight can then no longer be recovered by the initiator p. To cope with this situation, when p detects a crash failure, it initiates a run of a snapshot algorithm to account for the weight still present at the live processes and basic messages in transit. We employ the Lai-Yang algorithm from section 3.2 for this purpose. Each live process q reports the weight in its local snapshot to p, including the weights of basic messages in the states of its incoming channels rq for which r is alive. The initiator p waits until every live process has reported its weight and then determines the total amount of weight in the network. If during a run of the snapshot algorithm p detects a new crash, it starts a new run of the snapshot algorithm. Either p must wait until the current run has completed or control messages of different runs of the snapshot algorithm must be distinguished by means of a sequence number. If a process q has reported its local snapshot to p while knowing that another process r has crashed and later receives a basic message m from r, then q must ignore m if it is passive and must in any case discard the weight in m. In the case of underflow, a process can attribute extra weight to itself just before reporting its local snapshot to p; this extra weight is included in the snapshot.

The second problem is that, at the start of a snapshot, the initiator p may know that a process r has crashed while a live noninitiator q does not. Then q may report its local snapshot to p and later send a basic message m to r. When eventually q's failure detector reports that r has crashed, q cannot figure out whether the weight of m has been lost. If the weight is indeed lost and no further crashes happen, the initiator p will never realize that the total amount of weight in the network has been reduced. To avoid this situation, p informs all live noninitiators at the start of a snapshot which processes have crashed according to p's failure detector.

The third problem is that the initiator of the basic algorithm, which is responsible for announcing termination, may crash. Therefore, the processes in the network are enumerated: $p_0, \ldots, p_{N-1}$, where p_0 is the initiator of the basic algorithm. When a process p_{i+1} discovers that all processes p_j with $j \leq i$ have crashed, it becomes the leader, meaning that it takes responsibility for initiating snapshots, recovering the weight in the network, and announcing termination. (We note that this is not an election algorithm according to the definition that will be presented at the start of chapter 9, because processes do not have the same local election algorithm, as it depends on their process ID.) Each process q considers as leader the process with the smallest process ID that is still alive according to q's failure detector. When p_{i+1} becomes the new leader, a crash has occurred, so p_{i+1} initiates a snapshot and thus gets an accurate view of the weight in the network.

Example 6.6 Let the fault-tolerant weight-throwing algorithm run on a network of three active processes p_0, p_1, and p_2. Suppose p_1 sends a basic message m_1 to p_0, while p_2 crashes after sending a basic message m_2 to p_1. When the failure detector of the initiator p_0 detects this crash, p_0 starts a snapshot by taking a local snapshot and sending a control message to p_1. When p_1 receives this message, it takes a local snapshot, determines that the incoming channel from p_0 is empty, and sends control messages to p_0 and p_2. Next, the failure detector of p_1 detects that p_2 crashed, so p_1 no longer waits for a control message from p_2, decides that the incoming channel from p_2 is empty, and reports its own weight to p_0. Later m_2 arrives at p_1, which discards the weight in this basic message. When p_0 receives the control message from p_1, it determines that this incoming channel contains a basic message and it waits for m_1 to arrive to determine the snapshot of this channel. Finally, p_0 receives the local snapshot of p_1 and determines as the total weight in the network the sum of the weights at p_0 and p_1, which includes the weight in m_1.

Bibliographical notes

The Dijkstra-Scholten algorithm originates from [29], and the Shavit-Francez algorithm comes from [85]. Rana's algorithm stems from [78]. Safra's algorithm was presented in [28]. Weight throwing was proposed in [63], and its fault-tolerant variant was proposed in [95].

6.6 Exercises

Exercise 6.1 [92] How much time does the Dijkstra-Scholten algorithm need at most to call *Announce* after the basic algorithm has terminated?

Exercise 6.2 Give a computation of the Shavit-Francez algorithm with two initiators, in which one of the initiators becomes active again after having become passive, and both initiators concurrently call *Announce*.

Exercise 6.3 Consider the following computation of a decentralized basic algorithm on an undirected ring of size 3, with processes p, q, and r, where p and q are the initiators. First, p sends a message to q and r and becomes passive, while q sends a message to r. When q receives p's message, it also becomes passive. After receiving the messages first from p and then from q, r sends a message to both p and q and becomes passive. After receiving the message from r, p and q send a message to each other and after receiving these messages become passive.

Add the following termination detection algorithms on top of the basic algorithm, and in each case extend the basic computation with control messages to explain how termination is detected:

(a) The Shavit-Francez algorithm.
(b) Rana's algorithm.
(c) Safra's algorithm, with p as initiator of this control algorithm.

Exercise 6.4 Suppose that Rana's algorithm is adapted as follows: only quiet processes that have been quiet from some logical time $< t$ (instead of $\leq t$) onward can take part in a wave tagged with time stamp t. Give an example of a finite computation for which termination would not be detected.

Exercise 6.5 Give an example to show that, in Safra's algorithm, coloring sending processes black (instead of receiving ones) is incorrect.

Exercise 6.6 In Safra's algorithm, certain messages do not need to color the receiver black. Only messages that are sent after a token visits the sender and that are received before this same token visits the receiver need to be taken into account. Propose an optimization of Safra's algorithm based on this observation.

Exercise 6.7 [92] Safra's algorithm can be viewed as a snapshot algorithm. During every tour of the token, each process takes a local snapshot when it handles the token. In the constructed snapshot, all processes are passive, because the token is handled only by passive processes. Explain how the token's message integer value and color (when the token arrives back at the initiator) capture the consistency and channel states of the snapshot. In particular, argue that the following two claims are true:

1. If the token is white, the snapshot is consistent.
2. If, moreover, the token's integer value is zero, all channels are empty.

Exercise 6.8 Consider weight-throwing termination detection, where in the case of underflow at a process *p*, it gives itself extra weight and informs the initiator. Give an example to show that if *p* does not wait for an acknowledgment from the initiator, then the initiator could prematurely detect termination.

Exercise 6.9 Consider the fault-tolerant weight-throwing termination detection algorithm. Suppose a process *p* reports its local snapshot to the leader while knowing another process *q* has crashed, and later *p* receives a basic message from *q*. Explain what could go wrong if *p* added the weight in this message to its own weight.

Exercise 6.10 Consider the fault-tolerant weight-throwing termination detection algorithm. Suppose processes *p* and *q* report their local snapshot to the leader, where *p* knows that another process *r* has crashed, while *q* does not. Explain why in this case the leader will start a new run of the snapshot algorithm.

7

Garbage Collection

Each process is provided with memory to store, for example, its local state. An *object* in memory can carry references to other objects, possibly in the memory of other processes. A reference to a local object, which is located at the same process, is called a *pointer*, to distinguish it from a *reference* to a remote object, which is located at another process. An object needs to be kept in memory only if it is accessible by navigating from a *root object*. An object is *garbage* if it cannot be accessed from any root object.

Garbage collection aims to automatically detect and reclaim inaccessible memory objects in order to free up memory space. The two main techniques for garbage collection are reference counting and tracing. Reference counting, which counts the number of references to an object, is discussed in section 7.1. Tracing, which marks all objects reachable from the root objects, is discussed in section 7.3. In section 7.2, it is shown that there is a strong link between garbage collection and termination detection, which was discussed in the previous chapter.

7.1 Reference Counting

Reference counting is based on keeping track of the number of references to an object in memory; if it drops to zero and there are no pointers to the object, then the object is garbage.

An advantage of reference counting is that it can be easily performed at run-time, because counting can be done locally. A disadvantage is that cyclic garbage, meaning a cycle of references between garbage objects, is not detected. A separate technique must be added to try and detect such cycles. For instance, a nonroot object that is suspected of being part of cyclic garbage may be virtually deleted. That is, a separate set of trial reference counts is used to propagate the effects of this trial deletion: every outgoing link of the suspected object is trial deleted, and as a result the trial count at the destination object of the link is its current reference count minus 1; if as a result the trial count at an object becomes zero, it is also trial deleted, and so on. If in the end the trial count of the suspected object drops to zero, then this confirms that the object is garbage. In that case, it can be physically deleted.

The owner of an object O, meaning the process where O is located, can easily count the (local) pointers to O. But, in a distributed setting, the challenge is to keep track of the number of (remote) O-references during the execution of a (basic) distributed algorithm. We distinguish three operations in which processes build or delete a reference to an object O:

- *Creation*: The owner of O sends an O-pointer to another process.
- *Duplication*: A process that is not the owner of O sends an O-reference to another process.
- *Deletion*: An O-reference is deleted because it has become obsolete.

Reference counting must take into account basic messages that duplicate a reference. Otherwise an object could be reclaimed prematurely if there are no pointers and references to it but a message is carrying a duplicated reference to this object. One solution is that a process wanting to duplicate a reference must first inform the object owner; the reference is duplicated only after the receipt of the owner's acknowledgment. The drawback of this approach is high synchronization delays. We now discuss two different approaches to avoiding such delays.

Indirect Reference Counting

One method to avoid having to inform the object owner when a reference is duplicated is to maintain a tree for each object, with the object at the root and the references to this object as the other nodes in the tree. Each reference keeps track of where it was duplicated or created from; that is, it stores its parent in the tree. Objects and references are provided with a counter, estimating from above how many children they have in the tree: the counter at an object keeps track of how many references to the object have been created, while the counter at a reference keeps track of how many times the reference has been duplicated.

When a process receives a reference to an object but already holds a reference to or owns this object, it sends back a decrement to decrease the counter at the sender. A deleted reference can be restored if a duplication or creation of this reference is received before its counter has become zero. In other words, a deleted reference that is still part of a tree does not switch to another tree.

When a duplicated (or created) reference has been deleted and its counter is zero, a decrement message is sent to the process from which it was duplicated (or to the object owner). When the counter of the object becomes zero and there are no pointers to it, the object can be reclaimed.

Example 7.1 We consider one possible computation with indirect reference counting on an undirected ring of three processes *p*, *q*, and *r*. Let *p* hold one pointer to the object *O*.

- *p* sends *O*-references to *q* and *r*; it sets its counter of created *O*-references to 2. Upon receipt of these messages, *q* and *r* build an *O*-reference.
- *q* sends an *O*-reference to *r*; it increases its counter of duplicated *O*-references to 1. Upon receipt of this message, *r* sends back a decrement message to *q* because it already holds an *O*-reference. Upon receipt of this message, *q* decreases its counter back to 0.
- *p* deletes its *O*-pointer. (Since its counter is 2, *O* cannot yet be reclaimed by the garbage collector.)
- *r* deletes its *O*-reference. Since its counter is 0, *r* sends a decrement message to *p*, which causes *p* to decrease its counter to 1.
- *q* sends an *O*-reference to *r*; it increases its counter to 1.
- *q* deletes its *O*-reference. (Since its counter is 1, *q* does not yet send a decrement message to *p*.)
- Note that there is no pointer or reference to *O*, but there is still an *O*-reference traveling from *q* to *r*. Upon receipt of this message, *r* builds an *O*-reference.
- *r* deletes its *O*-reference. Since its counter is 0, *r* sends a decrement message to *q*, which causes *q* to decrease its counter to 0.
- Since *q* holds no *O*-reference and its counter is 0, it sends a decrement message to *p*, which causes *p* to decrease its counter to 0.
- Since *p* holds no *O*-pointer and its counter is 0, *O* can be reclaimed by the garbage collector.

Weighted Reference Counting

Another method to avoid having to inform the object owner when a reference is duplicated is to provide each object with a total weight. References are provided with a part of the weight of the object to which they refer. Each object maintains a partial weight that was not yet handed out to references to the object. Initially, the partial weight of an object equals its total weight.

When a reference is created, the partial weight of the object is divided between the object and the reference. That is, the object owner gives some weight to the message responsible for creating this reference, and it deducts this weight from the partial weight of the object. When the message arrives at its destination, either the reference is created with the weight of the message, if the process does not yet hold a reference to this object, or otherwise the weight of the message is added to this reference. Likewise, when a reference is duplicated, the weight of the reference is divided between itself and the copy (except if the reference happens to be duplicated to the object owner, in which case the weight is subtracted from the total weight of the object). When a reference is deleted, the object owner is notified and the weight of the deleted reference is subtracted from the total weight of the object. When the total weight of the object becomes equal to its partial weight and there are no pointers to the object, it can be reclaimed.

Example 7.2 We consider one possible computation with weighted reference counting on an undirected ring of three processes *p*, *q*, and *r*. Let *p* hold one pointer to the object *O*, which has total and partial weights 12.

- *p* sends *O*-references to *q* and *r*, with weights 6 and 3, respectively, and reduces the partial weight of *O* to 3. Upon receipt of these messages, *q* and *r* build an *O*-reference, with weights 6 and 3.
- *q* sends an *O*-reference to *r*, with weight 3, and reduces the weight of its *O*-reference to 3. Upon receipt of this message, *r* increases the weight of its *O*-reference to 6.
- *p* deletes its *O*-pointer. (Since the partial weight of *O* is less than its total weight, *O* cannot yet be reclaimed by the garbage collector.)
- *r* deletes its *O*-reference and sends a control message to *p* with weight 6. Upon receipt of this message, *p* decreases the total weight of *O* to 6.
- *q* sends an *O*-reference to *r* with weight 1.5 and decreases the weight of its *O*-reference to 1.5.
- *q* deletes its *O*-reference and sends a control message to *p* with weight 1.5. Upon receipt of this message, *p* decreases the total weight of *O* to 4.5.
- Note that there is no pointer or reference to *O*, but there is still an *O*-reference traveling from *q* to *r*. Upon receipt of this message, *r* builds an *O*-reference with weight 1.5.
- *r* deletes its *O*-reference and sends a control message to *p* with weight 1.5. Upon receipt of this message, *p* decreases the total weight of *O* to 3. Since the partial and total weights of *O* are now equal, and *p* holds no *O*-pointer, *O* can be reclaimed by the garbage collector.

Just as with weight-throwing termination detection, a drawback of weighted reference counting is the possibility of underflow: when the weight of a reference becomes too small to be divided further, it can no longer be duplicated. There are two possible solutions:

1. The reference increases its weight and tells the object owner to increase its total weight. An acknowledgment from the object owner to the reference is needed before it can be duplicated, to avoid race conditions.
2. The process at which the underflow occurs creates an artificial object, with a reference to the original object. Duplicated references are then referenced to the artificial object, so that references to the original object become indirect.

7.2 Garbage Collection Implies Termination Detection

At first sight, garbage collection has little in common with termination detection, discussed in the previous chapter. On the other hand, the garbage collection algorithms in the previous section may have reminded you of some of the termination detection algorithms discussed earlier. This is not a coincidence. It turns out that garbage collection algorithms can be transformed into (existing and new) termination detection algorithms. This works as follows. Given a basic algorithm, let each process p host one artificial root object O_p. There is also a special nonroot object Z. Initially, only initiators p hold a reference from O_p to Z. Each basic message carries a duplication of the Z-reference. When a process becomes passive, it deletes its Z-reference. When a process becomes active, it immediately duplicates a Z-reference, owing to the fact that all basic messages carry a Z-reference.

The basic algorithm is terminated if and only if Z is garbage. If all processes are passive and there are no messages in transit, then clearly there is no Z-reference. And vice versa, if there is an active process or a message in transit, it holds (a duplication of) the Z-reference.

This transformation turns indirect reference counting into Dijkstra-Scholten termination detection (see exercise 7.6) and weighted reference counting into a slight variation of weight-throwing termination detection (see exercise 7.7). Note that examples 7.1 and 7.2 are basically identical to examples 6.1 and 6.5, respectively, but in the context of garbage collection instead of termination detection.

7.3 Tracing

Tracing (or mark-scan) garbage collection consists of two phases. The first phase consists of a traversal of all accessible objects in memory, starting from the root objects; the accessible objects are marked. In the second phase, all unmarked objects are reclaimed.

An advantage of this approach, compared to reference counting, is that it detects all garbage, including cyclic garbage. A disadvantage is that it tends to require freezing the basic execution, since it performs a global scan of all reachable memory objects. In spite of this drawback, tracing has become more widely used than reference counting, since it has become the method of choice for garbage collection within the programming language Java. A key to this success has been the division of objects into two generations.

There are two standard ways to perform the second phase of tracing, in which unmarked objects are reclaimed:

- *Mark-copy*: Copy all marked objects to contiguous empty memory space.
- *Mark-compact*: Compact all marked objects in the current memory space.

Copying is significantly faster than compaction, because marked objects are copied without changing the memory structure. However, in the long run, mark-copy leads to a fragmentation of the memory space, because objects have a fixed place, while mark-compact clearly resolves such fragmentation.

In practice, most objects either can be reclaimed shortly after their creation or stay accessible for a very long time. This observation is exploited by generational garbage collection, in which objects are divided into two generations. Garbage in the young generation is collected frequently using mark-copy, while garbage in the old generation is collected sporadically using mark-compact. A newly created object starts in the young generation. If it stays accessible for a certain amount of time (or for a certain number of garbage collection runs), it is moved from the young generation to the old generation.

Bibliographical notes

The technique for detecting cyclic garbage that was mentioned at the start of this chapter originates from [96]. Indirect reference counting was put forward in [76], and weighted reference counting was proposed independently in [12] and [97]. The derivation of termination detection algorithms from garbage collection algorithms was observed in [93]. Generational garbage collection stems from [59].

7.4 Exercises

Exercise 7.1 Give an example of cyclic garbage where trial deletion of an object does not help to detect garbage.

Exercise 7.2 Consider the following computation of a basic algorithm on an undirected ring of size 3, with processes p, q, and r, where p owns an object O. Initially, there is one O-pointer. First, p sends a message to q and r, both containing a created O-reference. Next, p deletes the O-pointer. When the message from p arrives, q and r create an O-reference. Now q and r send messages to each other, both of which contain a duplicated O-reference, and delete their O-references. When these messages arrive, q and r create an O-reference again. Finally, q and r delete their O-references.

Explain for each of the following two garbage collection algorithms how it is detected that O has become garbage:

(a) Indirect reference counting.
(b) Weighted reference counting.

Exercise 7.3 Argue the correctness of indirect as well as weighted reference counting.

Exercise 7.4 In weighted reference counting, why is underflow much more likely to happen than overflow of a reference counter?

Exercise 7.5 Consider solution 1 for dealing with underflow in weighted reference counting. Give an example to show that if the process where the weight is increased would not wait for an acknowledgment from the object owner, then the object owner could prematurely mark the object as garbage.

Exercise 7.6 Using the technique from section 7.2, show that indirect reference counting gives rise to Dijkstra-Scholten termination detection.

Exercise 7.7 Using the technique from section 7.2, show that weighted reference counting gives rise to a variation of weight-throwing termination detection in which the initiator cannot reuse weight that was returned to it. Also take into account solution 1 in the case of underflow.

8

Routing

When a process wants to send a message to another process in the network that is not a direct neighbor, the message needs to be routed through the network. Effective routing algorithms are of vital importance, especially for the Internet. Each process q maintains a routing table, which stores for each destination $p \neq q$ the distance of q to p as well as a neighbor r of q: each message with destination p that arrives at q is passed on to r. We assume that each channel pq is provided with a positive weight $weight(pq)$ from $\mathbb{R}_{>0}$, and we will discuss algorithms that route messages via shortest paths, meaning that the sum of the weights of the traversed channels is minimal.

8.1 Chandy-Misra Algorithm

The Chandy-Misra algorithm is a centralized routing algorithm (also called a single-source shortest-path algorithm) for undirected networks. It computes a sink tree consisting of shortest paths to the initiator.

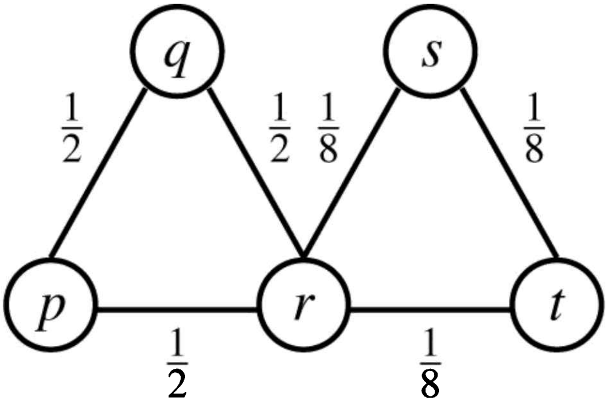
Each process p maintains values $dist_p$ and $parent_p$, where $dist_p$ is the length of the shortest known path from p to the initiator, and $parent_p$ is the process after p on this path. Initially, the variable $dist$ at the initiator has value 0, $dist_p = \infty$ (that is, infinity) for each noninitiator p, and $parent_p = \perp$ (that is, undefined) for all processes p.

The algorithm starts with messages ⟨**dist**, 0⟩, which the initiator sends to all its neighbors, informing them that the initiator knows a path to itself of distance 0.

When a process p receives a message ⟨**dist**, d⟩ from a neighbor q, it checks whether $d + weight(pq) < dist_p$. If this is the case, then p has found a shorter path to the initiator via q, so it changes $dist_p$ into $d + weight(pq)$ and $parent_p$ into q, and communicates the improved estimate to all neighbors except q in the form of a message ⟨**dist**, $dist_p$⟩. If not, then p simply dismisses the incoming message from q.

A termination detection algorithm needs to be used on the side. For instance, one could employ the Dijkstra-Scholten algorithm (see section 6.1).

Example 8.1 We consider the longest possible computation of the Chandy-Misra algorithm on the following network with initiator p.



Initially, $dist_p = 0$ and $dist_q = dist_r = dist_s = dist_t = \infty$, while $parent = \perp$ at all five processes.

- p sends ⟨**dist**, 0⟩ to q and r.
- When p's message arrives at q, $dist_q \leftarrow \frac{1}{2}$ and $parent_q \leftarrow p$, and q sends ⟨**dist**, $\frac{1}{2}$⟩ to r.
- When q's message arrives at r, $dist_r \leftarrow 1$ and $parent_r \leftarrow q$, and r sends ⟨**dist**, 1⟩ to p, s, and t.
- p dismisses r's message.
- When r's message arrives at s, $dist_s \leftarrow \frac{9}{8}$ and $parent_s \leftarrow r$, and s sends ⟨**dist**, $\frac{9}{8}$⟩ to t.
- When s's message arrives at t, $dist_t \leftarrow \frac{5}{4}$ and $parent_t \leftarrow s$, and t sends ⟨**dist**, $\frac{5}{4}$⟩ to r.
- r dismisses t's message.
- When r's message arrives at t, $dist_t \leftarrow \frac{9}{8}$ and $parent_t \leftarrow r$, and t sends ⟨**dist**, $\frac{9}{8}$⟩ to s.
- s dismisses t's message.
- When p's message (finally) arrives at r, $dist_r \leftarrow \frac{1}{2}$ and $parent_r \leftarrow p$, and r sends ⟨**dist**, $\frac{1}{2}$⟩ to q, s, and t.

- q dismisses r's message.
- When r's message arrives at s, $dist_s \leftarrow \frac{5}{8}$ and $parent_s \leftarrow r$, and s sends ⟨**dist**, $\frac{5}{8}$⟩ to t.
- When s's message arrives at t, $dist_t \leftarrow \frac{3}{4}$ and $parent_t \leftarrow s$, and t sends ⟨**dist**, $\frac{3}{4}$⟩ to r.
- r dismisses t's message.
- When r's message arrives at t, $dist_t \leftarrow \frac{5}{8}$ and $parent_t \leftarrow r$, and t sends ⟨**dist**, $\frac{5}{8}$⟩ to s.
- s dismisses t's message.

We argue that the Chandy-Misra algorithm computes shortest paths toward the initiator. A safety property of the algorithm is that any process p with $dist_p < \infty$ has a shortest path to the initiator with weight at most $dist_p$. This follows from the facts that this property holds initially and it is an invariant: if p receives a message ⟨**dist**, d⟩ from a neighbor q, then there is a path from p via q to the initiator with weight at most d + $weight(pq)$, so (even) if p changes $dist_p$ into $d + weight(pq)$, the property is preserved. For each process p, $dist_p$ will eventually attain the weight of a shortest path from p to the initiator, which we argue by induction on the number of channels in such a path. The base case, where p is the initiator, is trivial, because then $dist_p$ is 0 from the start. In the inductive case, let a shortest path from p to the initiator start with the channel pq. By induction, eventually $dist_q$ will attain the weight of a shortest path from q to the initiator; this path cannot go via p, because channels carry positive weights. So q will send ⟨**dist**, $dist_q$⟩ to p, and upon receipt of this message, $dist_p$ will equal the weight of a shortest path from p to the initiator. Finally, if p is a noninitiator, then a shortest path from p to the initiator goes via $parent_p$, because $parent_p$ is updated at each improvement of $dist_p$.

The worst-case message complexity of the Chandy-Misra algorithm is exponential, since there can be exponentially many different cycle-free paths from a process to the initiator, which may be discovered in decreasing order of weight (see exercise 8.3). In the case of *minimum-hop paths* in unweighted networks (in other words, each channel has weight 1), the worst-case message complexity of computing shortest paths to all processes in the network drops down to $O(N^2{\cdot}E)$. For each process, the algorithm requires at most $O(N{\cdot}E)$ messages to compute all shortest paths to this process: the longest cycle-free path has at most length $N - 1$, so each process sends at most $N - 1$ messages to its neighbors. In the case of minimum-hop paths, the sink tree forms what is called a breadth-first search tree.

8.2 Merlin-Segall Algorithm

The Merlin-Segall algorithm is a centralized algorithm to compute all shortest paths to the initiator. The underlying idea is to bring structure to the Chandy-Misra algorithm by letting it proceed in rounds. In each round, distance messages à la Chandy-Misra flow up and down the sink tree, similarly as in the echo algorithm, and distance values are updated. At the end of each round, the sink tree is restructured.

Initially (after round 0), the variable *dist* at the initiator has value 0, $dist_p = \infty$ for each noninitiator p, and the $parent_p$ values form a sink tree with the initiator as root. Such a sink tree can be built by means of a centralized wave algorithm from chapter 4.

Each round > 0 is started by the initiator, which sends the message ⟨**dist**, 0⟩ to its neighbors to inform them that the initiator has a shortest path to itself of length 0.

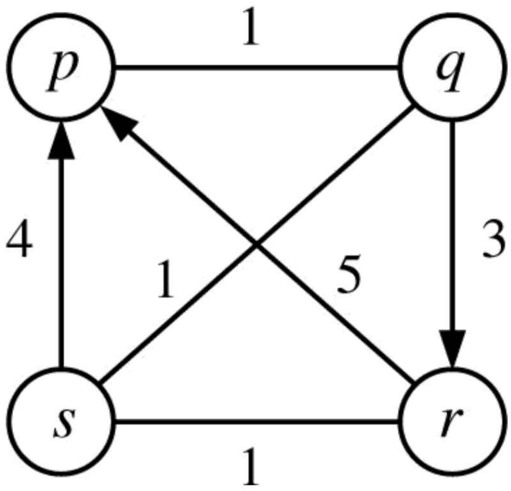
Let a noninitiator p receive a message ⟨**dist**, d⟩ from a neighbor q.

- If $d + weight(pq) < dist_p$, then $dist_p \leftarrow d + weight(pq)$ (and p stores q as a future value for $parent_p$).
- If $q = parent_p$, then p sends ⟨**dist**, $dist_p$⟩ to its neighbors, except q.

When p has received a message from all its neighbors in the current round, it sends ⟨**dist**, $dist_p$⟩ to $parent_p$ and moves to the next round. If p updated $dist_p$ in the last round, then p updates $parent_p$ to the neighbor whose message is responsible for the current value of $dist_p$. The initiator starts a new round after it has received a message from all its neighbors in the current round.

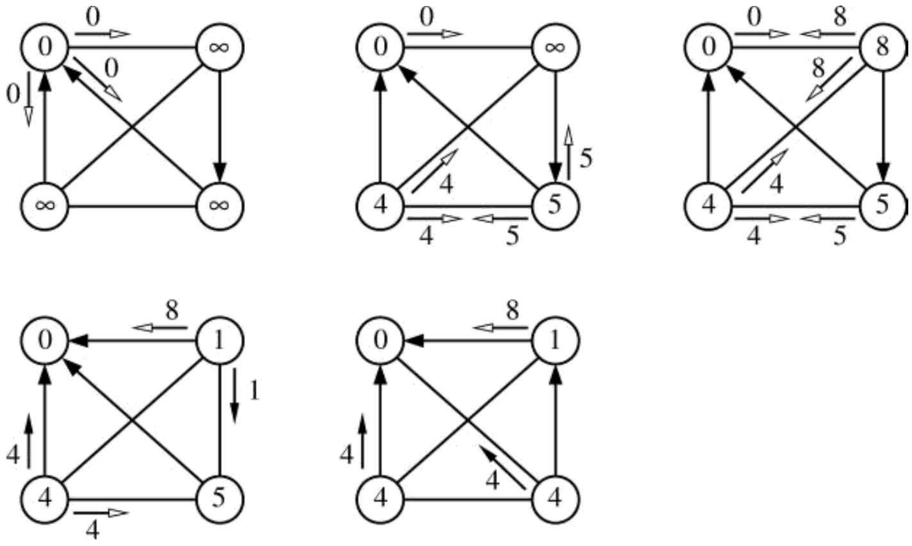
After round $n \geq 0$, for each process p with a shortest path to the initiator that consists of $\leq n$ channels, $dist_p$ and $parent_p$ have achieved their ultimate values. This is easy to see by induction on n. The base case $n = 0$ is trivial, because at the initiator the variables *dist* and *parent* have values 0 and $\perp$, respectively, from the start. Now consider the inductive case: a noninitiator p with a shortest path of $\leq n + 1$ channels to the initiator. Let pq be the first channel in this path. Then process q has a shortest path of $\leq n$ channels to the initiator. By the induction hypothesis, after round n, $dist_q$ has obtained its ultimate value. So in round $n + 1$, p receives the message ⟨**dist**, $dist_q$⟩ from q. Therefore, $dist_p$ and $parent_p$ have achieved their ultimate values at the end of round $n + 1$. Since shortest paths consist of at most $N - 1$ channels, the Merlin-Segall algorithm can terminate after round $N - 1$.

Example 8.2 We consider one possible computation of the Merlin-Segall algorithm on the following undirected network. The original sink tree, after round 0, consists of the edges qr, rp, and sp.

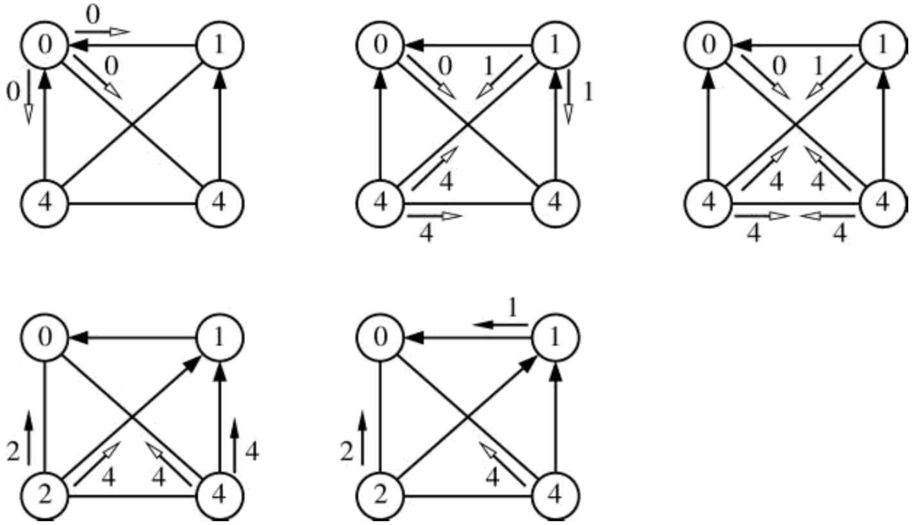


The computation we consider determines the correct sink tree toward the initiator *p* only at the end of round 3, because (1) there is a shortest path of length 3, (2) we start with a sink tree that has nothing in common with the correct sink tree, and (3) in every round we let messages in the opposite direction of the sink tree travel very quickly, so that processes send messages to their neighbors early on.

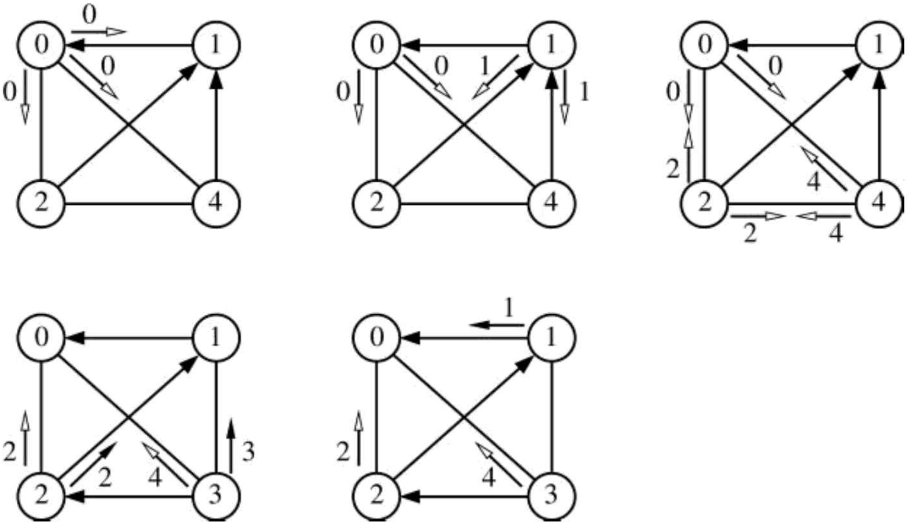
The five pictures that follow show the progression of the computation in round 1. Process names and channel weights have been omitted from these pictures. Messages that are depicted with a solid arrowhead are toward a parent. In the first picture, *p* has sent out messages ⟨**dist**, 0⟩. In the second picture, *r* and *s* have received this message from their parent *p*, computed distances 5 and 4, and sent ⟨**dist**, 5⟩ and ⟨**dist**, 4⟩, respectively, to their other neighbors. In the third picture, *q* has received ⟨**dist**, 5⟩ from its parent *r*, computed distance 8, and sent ⟨**dist**, 8⟩ to its other neighbors. In the fourth picture, *s* has received ⟨**dist**, 8⟩ from *q* and ⟨**dist**, 5⟩ from *r*, sent ⟨**dist**, 4⟩ to its parent *p*, and made *p* its parent (again); moreover, *q* has received ⟨**dist**, 0⟩ from *p* and ⟨**dist**, 4⟩ from *s*, computed an improved distance 1, sent ⟨**dist**, 1⟩ to its parent *r*, and made *p* its new parent. In the fifth picture, *r* has received ⟨**dist**, 1⟩ from *q* and ⟨**dist**, 4⟩ from *s*, computed an improved distance 4, sent ⟨**dist**, 4⟩ to its parent *p*, and made *q* its new parent. When the three messages traveling toward *p* have reached their destinations, round 2 is started.



The depictions of the two other rounds are given without further explanations, as they are similar to those of round 1. The five pictures that follow show the progression of the computation in round 2.



Finally, the five pictures that follow show the progression of the computation in round 3.



Now the computation has terminated. In the terminal configuration, the correct shortest paths toward *p* have been computed, leading from *r* to *s* to *q* to *p*.

The message complexity of the Merlin-Segall algorithm is $\Theta(N^2 \cdot E)$. For every root, the algorithm requires $(N - 1) \cdot 2 \cdot E$ messages, since in each of the $N - 1$ rounds, two messages travel through each channel.

The Merlin-Segall algorithm can be adapted to make it robust against topology changes. When a channel fails or becomes operational, the adjacent processes send a special control message toward the initiator via the sink tree. If the failed channel happens to be a tree edge, then the remaining tree is extended to a complete sink tree toward the initiator again. If the special control message meets a failed tree edge, it is discarded. This is no problem, because the other side of this tree edge already sends a control message toward the initiator. When the initiator receives this control message, it starts a new set of *N* rounds, with a higher number. This number is attached to the messages in this run of the algorithm.

8.3 Toueg's Algorithm

Toueg's algorithm is a decentralized algorithm for undirected networks that generalizes the well-known Floyd-Warshall algorithm to a distributed setting. It is an all-pairs shortest-path algorithm, meaning that it computes a shortest path between any pair of processes. The idea behind the Floyd-Warshall algorithm is to compute, for each set *S* of processes, a distance function $d^S(p, q)$ denoting the length of a shortest path between *p* and *q* with all *intermediate* processes in *S*. Its value is ∞ if there is no such path. The following equations hold for all processes p, q:

$$\begin{aligned}
d^S(p,p) &= 0. \\
d^{\emptyset}(p,q) &= \mathit{weight}(pq) \text{ if } p \neq q \text{ and there is a channel } pq. \\
d^{\emptyset}(p,q) &= \infty \text{ if } p \neq q \text{ and there is no channel } pq. \\
d^{S\cup\{r\}}(p,q) &= \min\{d^S(p,r) + d^S(r,q), d^S(p,q)\} \text{ for each } r \notin S.
\end{aligned}$$

The first equation is obvious. For the second and third equations, note that if $S = \emptyset$, then a path between two distinct processes p and q with all intermediate processes in S can only consist of a channel between p and q. The fourth equation expresses that a shortest path between p and q with all intermediate processes in $S \cup \{r\}$ either visits r or does not; in the first case this path has length $d^S(p, r) + d^S(r, q)$, and in the second case it has length $d^S(p, q)$. Here, $\min\{\infty, a\} = a$ and $\infty + a = \infty$ for all $a \in \mathbb{R}_{\geq 0} \cup \{\infty\}$.

The Floyd-Warshall algorithm starts with $S = \emptyset$, in which case the first three equations completely define d^S. As long as S does not contain all processes, a *pivot* $r \notin S$ is selected and $d^{S\cup\{r\}}$ is computed from d^S using the fourth equation; then r is added to S. Finally, note that if S contains all processes, then d^S is the standard distance function.

Transferring this algorithm to a distributed setting gives rise to two complications. First, all processes must uniformly select the pivots in the same order. Therefore, we make the (strong) assumption that each process knows from the start the IDs of all processes in the network. Second, in each pivot round, the pivot r must broadcast its routing table because a process p may need to know $d^S(r, q)$ in order to compute $d^{S\cup\{r\}}(p, q)$ in view of the fourth equation.

In Toueg's algorithm, each process p starts with $S_p = \emptyset$ and maintains values $\mathit{dist}_p(q)$ and $\mathit{parent}_p(q)$ for each process q, where $\mathit{dist}_p(q)$ is the length of the shortest known path from p to q, and $\mathit{parent}_p(q)$ is the process after p on this path. Initially, $\mathit{dist}_p(p) = 0$ and $\mathit{parent}_p(p) = \perp$, while for each $q \neq p$, either $\mathit{dist}_p(q) = \mathit{weight}(pq)$ and $\mathit{parent}_p(q) = q$ if there is a channel pq, or $\mathit{dist}_p(q) = \infty$ and $\mathit{parent}_p(q) = \perp$ otherwise.

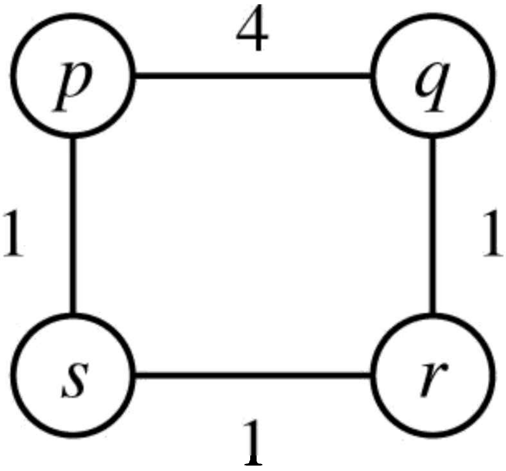
In each successive round, the same pivot r is selected by all processes p and added to the sets S_p. The pivot r broadcasts its values $\mathit{dist}_r(q)$ for all processes q. If $\mathit{parent}_p(r) = \perp$ for a process $p \neq r$ in this pivot round, then $\mathit{dist}_p(r) = \infty$, so $\mathit{dist}_p(r) + \mathit{dist}_r(q) = \infty \geq \mathit{dist}_p(q)$ for all processes q. Therefore, processes that are not in the sink tree toward r do not need the routing table of r. Hence, this sink tree can be used in the opposite direction to broadcast dist_r. To facilitate this use of r's sink tree, in the r-pivot round, each process p sends ⟨**request**, r⟩ to $\mathit{parent}_p(r)$ if it is not $\perp$ to let that process pass on the values $\mathit{dist}_r(q)$ to p. Next, p acts as follows:

- If p is not in the sink tree of r, then p immediately completes the r-pivot round.
- Suppose p is in the sink tree of r (that is, $\mathit{parent}_p(r) \neq \perp$ or $p = r$). If $p \neq r$, then p

must first wait until it has received $dist_r$ from $parent_p(r)$. It forwards $dist_r$ to those neighbors that send ⟨**request**, r⟩ to p. Moreover, for each process q, p checks whether $dist_p(r) + dist_r(q) < dist_p(q)$ and, if so, performs $dist_p(q) \leftarrow dist_p(r){+}dist_r(q)$ and $parent_p(q) \leftarrow parent_p(r)$. This completes the r-pivot round.

Finally, p either proceeds to the next pivot round, if S_p does not contain all processes, or terminates.

Example 8.3 We give a computation of Toueg's algorithm on the following network, with pivot order $p\ q\ r\ s$.



Initially, $dist_t(t) = 0$ for all four processes t, $dist_t(u) = weight(tu)$ if there is a channel tu, and all other $dist$ values are ∞. And $parent_t(u) = u$ if t is a direct neighbor of u, and all other $parent$ values are $\perp$.

In the p-pivot round, q and s both send ⟨**request**, p⟩ to p. So the distance values of p are sent to q and s, but not to r, which is not yet in the sink tree of p. As a result, q and s discover a path to each other via p, so that $dist_q(s)$ and $dist_s(q)$ are set to 5, and $parent_q(s)$ and $parent_s(q)$ are set to p.

In the q-pivot round, p, r, and s send ⟨**request**, q⟩ to q, q, and p, respectively. So the distance values of q are sent to p, r, and s. As a result, p and r discover a path to each other via q, so that $dist_p(r)$ and $dist_r(p)$ are set to 5, and $parent_p(r)$ and $parent_r(p)$ are set to q.

In the r-pivot round, p, q, and s send ⟨**request**, r⟩ to q, r, and r, respectively. So the distance values of r are sent to p, q, and s. As a result, q and s discover a shorter path to each other via r, so that $dist_q(s)$ and $dist_s(q)$ are set to 2, and $parent_q(s)$ and $parent_s(q)$ are set to r.

In the s-pivot round, p, q, and r send ⟨**request**, s⟩ to s, r, and s, respectively. So the distance values of s are sent to p, q, and r. As a result, p and r discover a shorter path to each other via s, so that $dist_p(r)$ and $dist_r(p)$ are set to 2, and $parent_p(r)$ and $parent_r(p)$ are set to s. Moreover, p and q discover a shorter path to each other via s, so that $dist_p(q)$ and $dist_q(p)$ are set to 3, $parent_p(q)$ to s, and $parent_q(p)$ to r.

The worst-case message complexity of Toueg's algorithm is $O(N^2)$, since there are N pivot rounds, and each round takes at most $O(N)$ messages to forward the distance values of the pivot through the sink tree.

By introducing negative counterparts of the **request** messages, which inform neighbors that no distance values for the current round have to be forwarded, the need for processes to store distance values of pivots from past rounds indefinitely is avoided. This, however, has a negative impact on the message complexity.

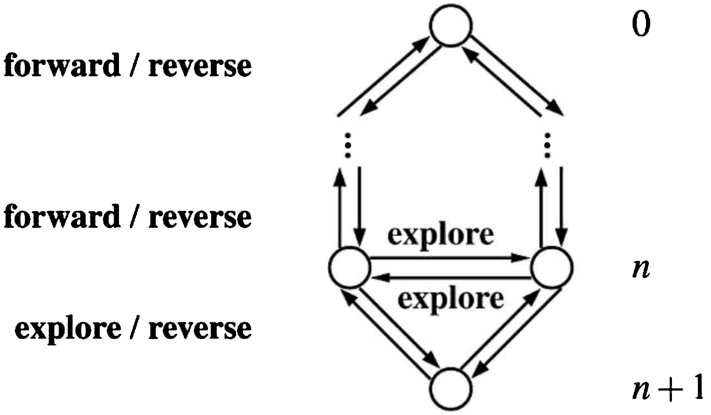
A drawback of Toueg's algorithm (besides uniform selection of pivots) is that all distance values of a pivot are sent through the sink tree of the pivot, which gives rise to a high bit complexity. This overhead can be reduced as follows. When a process p in the sink tree of the pivot r receives the distance values of r, it first performs for each process q the check for whether $dist_p(r) + dist_r(q) < dist_p(q)$. Now p only needs to forward those values $dist_r(q)$ for which this check yields a positive result (see exercise 8.9).

8.4 Frederickson's Algorithm

We now discuss a centralized algorithm to compute a breadth-first search tree toward the initiator in an undirected (unweighted) network. We first consider a simple version of this algorithm, in which the processes at distance n from the initiator are discovered in round n.

Initially, after round 0, the variable $dist$ at the initiator has value 0, $dist_p = \infty$ for each noninitiator p, and $parent_p = \perp$ for all processes p. After each round $n \geq 0$, the breadth-first search tree has been constructed up to depth n: for each process p at a distance $k \leq n$ from the initiator, $dist_p = k$, and p knows which neighbors are at distance $k - 1$; and if p is a noninitiator, then it has a parent in the sink tree toward the initiator.

We explain what happens in round $n + 1$. It can be depicted as follows.



At the start of the round, messages ⟨**forward**, n⟩ travel down the tree, from the initiator to processes at depth n. When a process p at depth n receives this message, it sends ⟨**explore**, $n + 1$⟩ to its neighbors that are not at depth $n - 1$. When such a neighbor q receives this message, it acts as follows, depending on whether $dist_q$ is ∞, $n + 1$, or n:

- If $dist_q = \infty$, then $dist_q \leftarrow n + 1$, $parent_q \leftarrow p$, and q sends back ⟨**reverse**, *true*⟩, informing p that q is a child of p.
- If $dist_q = n + 1$, then q stores that p is at depth n and sends back ⟨**reverse**, *false*⟩, informing p that q is not a child of p.
- If $dist_q = n$, then q interprets ⟨**explore**, $n + 1$⟩ as a negative reply to the message ⟨**explore**, $n + 1$⟩ that q sent (or will send) to p.

A process p at depth n waits until all messages ⟨**explore**, $n + 1$⟩ have been answered with a ⟨**reverse**, _⟩ or ⟨**explore**, $n + 1$⟩. Likewise, a noninitiator p at a depth $< n$ waits until all messages ⟨**forward**, n⟩ have been answered with a ⟨**reverse**, _⟩. In both cases, p sends ⟨**reverse**, b⟩ to its parent, where $b = true$ only if new processes were added to its subtree.

The initiator waits until all messages ⟨**forward**, n⟩ (or, in the case of round 1, ⟨**explore**, 1⟩) have been answered with a ⟨**reverse**, _⟩. If no new processes were added in round $n + 1$, then the initiator terminates, and it may inform all other processes that the breadth-first search has terminated. Otherwise the initiator continues with round $n + 2$; processes in the tree only send a **forward** to children that reported new processes in round $n + 1$.

The worst-case message complexity of this breadth-first search algorithm is $O(N^2)$. There are at most $N + 1$ rounds, and in each round tree edges carry at most one **forward** and one replying **reverse**, adding up to at most $2 \cdot (N - 1) \cdot N$ messages. And, in total, channels carry one **explore** and one replying **reverse** or **explore**, adding up to $2 \cdot E$ messages. The worst-case time complexity is also $O(N^2)$: round n is completed in at most $2 \cdot n$ time units, for $n = 1, ..., N$, and $2 \cdot (1 + 2 + \cdots + N) = N \cdot (N + 1)$.

Frederickson's algorithm is based on the observation that in the simple breadth-first search algorithm just described, **forward** messages often need to travel up and down the tree, since each round only discovers processes that are one level deeper than the ones discovered in the previous round, level meaning the distance to the initiator. Efficiency can be gained by exploring several levels in one round. However, **explore** messages then give a performance penalty because they may travel through the same channel multiple times in one round. Notably, if we abolished **forward** messages and used only **explore** messages to discover all processes in one round, we would be back at the Chandy-Misra algorithm, which we have seen is not efficient. Therefore, the number of levels explored in one round is included as a parameter ℓ. At the end, an optimal value for ℓ will be determined. In the simple breadth-first search algorithm, $\ell = 1$.

In Frederickson's algorithm, initially (after round 0) the variable *dist* at the initiator has value 0, $dist_p = \infty$ for each noninitiator p, and $parent_p = \perp$ for all processes p. After each round $n \geq 0$, the breadth-first search tree has been constructed up to depth $\ell \cdot n$: for each process p at a distance $k \leq \ell \cdot n$ from the initiator, $dist_p = k$, and p knows which neighbors are at distance $k - 1$; and if p is a noninitiator, then it has a parent in the sink tree toward the initiator.

At the start of round $n + 1$, messages ⟨**forward**, $\ell \cdot n$⟩ travel down the tree, from the initiator to processes at depth $\ell \cdot n$. When a process at depth $\ell \cdot n$ receives this message, it sends ⟨**explore**, $\ell \cdot n + 1$⟩ to its neighbors that are not at depth $\ell \cdot n - 1$. The parameter in this message is (possibly an overapproximation of) the depth of the receiving process. This value is increased by 1 every time the **explore** message is forwarded. When this parameter becomes divisible by ℓ, the ℓ levels for the current round have been explored and ⟨**reverse**, b⟩ messages bounce back toward the processes at level $\ell \cdot n$.

Compared to the simple breadth-first search algorithm, two complications occur. First, a process q may receive a **forward** from a neighbor p that is not its parent. This can happen if in the previous round p temporarily was q's parent but q later selected another parent with a shorter path to the initiator and p sent the **forward** to q before being informed that q is no longer its child (see exercise 8.13). Such a **forward** can simply be dismissed by q, because p will eventually receive q's negative acknowledgment. Second, a process may send multiple **explore**s into the same channel, if its distance value is improved several times in a round. Therefore, a **reverse** in reply to an **explore** is supplied with a distance parameter, so that a process can distinguish which **explore** an incoming **reverse** is a reply to.

We now specify Frederickson's algorithm in detail. Each process is supposed to know the value of ℓ. The algorithm is started by the initiator, which in round 1 sends ⟨**explore**, 1⟩ to its neighbors. Let a process q receive a message ⟨**explore**, k⟩ from a neighbor p. We consider two cases:

- If $k < dist_q$, then $dist_q \leftarrow k$ and $parent_q \leftarrow p$. Again, we consider two cases:

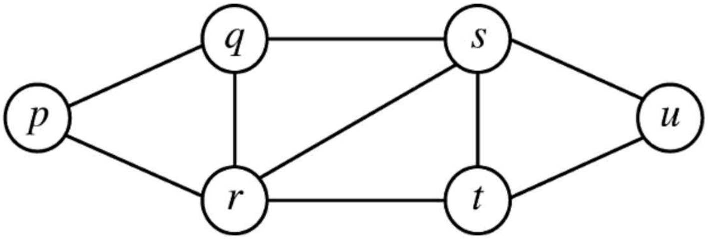
- * If k is not divisible by ℓ, then q sends ⟨**explore**, $k + 1$⟩ to its neighbors, except p. Next, q waits until it has received ⟨**reverse**, $k + 1$, _⟩ or ⟨**explore**, j⟩ with $j \in \{k, k + 1, k + 2\}$ from all these neighbors. Then q sends ⟨**reverse**, k, b⟩ to p, where b = *true* if and only if q received a message ⟨**reverse**, k +1, *true*⟩. Only neighbors that reply with ⟨**reverse**, k +1, *true*⟩ are children of q in the tree (unless q also receives a message ⟨**explore**, k⟩ from such a neighbor, in which case the sender is not a child of q).

  If q later receives a message ⟨**explore**, k' ⟩ with $k' < k$, then q changes its distance to k' , makes the sender its parent, sends messages ⟨**explore**, k' +1⟩, and waits for replies to these messages before sending ⟨**reverse**, k' , b⟩ to its new parent, where b = *true* if and only if q received a message ⟨**reverse**, k' + 1, *true*⟩.
- * If k is divisible by ℓ, then q sends ⟨**reverse**, k, *true*⟩ to p immediately.
- If $k \geq dist_q$, then again we consider two cases:
- * If $k > dist_q$ or k is not divisible by ℓ, then q does not send a reply to p. In this case, q sent ⟨**explore**, $dist_q$ + 1⟩ into the channel qp, which already serves as a negative acknowledgment to the current incoming message.
- * If $k = dist_q$ and k is divisible by ℓ, then q sends ⟨**reverse**, k, *false*⟩ to p.

A process p at depth $\ell{\cdot}n$ waits until all messages ⟨**explore**, $\ell{\cdot}n$ + 1⟩ have been answered with a **reverse** or **explore**. Likewise, a noninitiator p at a depth < $\ell{\cdot}n$ waits until all messages ⟨**forward**, $\ell{\cdot}n$⟩ have been answered with a **reverse**. In both cases, p sends ⟨**reverse**, b⟩ to its parent, where b = *true* only if new processes were added to its subtree.

The initiator waits until all messages ⟨**forward**, $\ell{\cdot}n$⟩ (or, in the case of round 1, ⟨**explore**, 1⟩) have been answered with a **reverse**. If no new processes were added in round $n + 1$, then the initiator terminates and may inform all other processes that the breadth-first search has terminated. Otherwise the initiator continues with round $n + 2$; processes in the tree only send a **forward** to children that reported new processes in round $n + 1$.

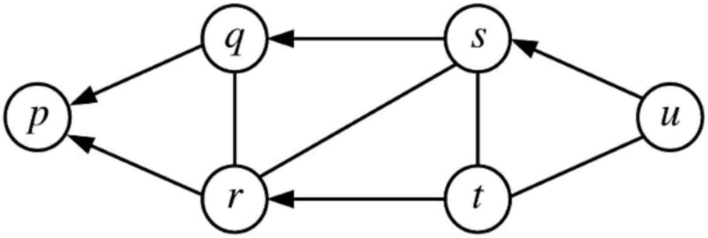
Example 8.4 We consider one possible computation of Frederickson's algorithm on the following network with p as initiator and $\ell = 3$.



After round 0, $dist_p = 0$ and the distance value of all other processes is ∞; nobody has a parent.

- Round 1 is started by p, which sends ⟨**explore**, 1⟩ to q and r.
- p's **explore** arrives at q: $dist_q \leftarrow 1$, $parent_q \leftarrow p$, and q sends ⟨**explore**, 2⟩ to r and s.
- q's **explore** arrives at r: $dist_r \leftarrow 2$, $parent_r \leftarrow q$, and r sends ⟨**explore**, 3⟩ to p, s, and t.
- p receives and dismisses r's **explore**.
- r's **explore** arrives at s: $dist_s \leftarrow 3$, $parent_s \leftarrow r$, and s sends ⟨**reverse**, 3, *true*⟩ in reply.
- r's **explore** arrives at t: $dist_t \leftarrow 3$, $parent_t \leftarrow r$, and t sends ⟨**reverse**, 3, *true*⟩ in reply.
- p's **explore** arrives at r: $dist_r \leftarrow 1$, $parent_r \leftarrow p$, and r sends ⟨**explore**, 2⟩ to q, s, and t.
- r receives and dismisses the **reverse**s from s and t.
- q's **explore** arrives at s: $dist_s \leftarrow 2$, $parent_s \leftarrow q$, and s sends ⟨**explore**, 3⟩ to r, t, and u.
- r's **explore** arrives at t: $dist_t \leftarrow 2$, and t sends ⟨**explore**, 3⟩ to s and u.
- s's **explore** arrives at u: $dist_u \leftarrow 3$, $parent_u \leftarrow s$, and u sends back ⟨**reverse**, 3, *true*⟩.
- t's **explore** arrives at u, which sends back ⟨**reverse**, 3, *false*⟩.
- r's and t's **explores** and u's **reverse** arrive at s, which sends ⟨**reverse**, 2, *true*⟩ to q.
- s's **explore** and u's **reverse** arrive at t, which sends ⟨**reverse**, 2, *true*⟩ to r.
- r's **explore** and s's **reverse** arrive at q, which sends ⟨**reverse**, 1, *true*⟩ to p.
- s's **explore** and t's **reverse** arrive at r, which sends ⟨**reverse**, 1, *true*⟩ to p.
- q's and r's **reverses** arrive at p, which starts round 2. No further processes are discovered in that round, after which the computation terminates.

The resulting spanning tree is as follows.



The worst-case message complexity of Frederickson's algorithm, where ℓ levels are explored in each round, is $O(\frac{N^2}{\ell} + \ell \cdot E)$. There are at most $\lceil \frac{N}{\ell} \rceil + 1$ rounds, and, in each round, tree edges carry at most one **forward** and one replying **reverse**, adding up to at most $2 \cdot \lceil \frac{N}{\ell} \rceil \cdot (N-1)$ messages. And, in total, channels carry at most $2 \cdot \ell$

explores and $2\cdot\ell$ replying **reverses**, and frond edges carry at most one spurious **forward**, adding up to (fewer than) $(4\cdot\ell + 1)\cdot E$ messages. Note that $\frac{N^2}{\ell}$ decreases while $\ell\cdot E$ increases when ℓ is increased. An optimal value for ℓ is determined by equating these two summands of the message complexity: $\frac{N^2}{\ell} = \ell\cdot E$ yields $\ell = \frac{N}{\sqrt{E}}$. Since ℓ must be an integer value, we take $\ell = \lceil\frac{N}{\sqrt{E}}\rceil$. The worst-case message complexity then becomes $O(N\cdot\sqrt{E})$.

The worst-case time complexity of Frederickson's algorithm is $O(\frac{N^2}{\ell})$: round n is completed in at most $2\cdot\ell\cdot n$ time units, for $n = 0,\ldots,\lceil\frac{N}{\ell}\rceil$, and $2\cdot\ell\cdot(1+2+\cdots+\lceil\frac{N}{\ell}\rceil) = \ell\cdot\lceil\frac{N}{\ell}\rceil\cdot(\lceil\frac{N}{\ell}\rceil+1)$. If we take $\ell = \lceil\frac{N}{\sqrt{E}}\rceil$, then the worst-case message time complexity becomes $O(N\cdot\sqrt{E})$.

Concluding, computing a breadth-first search tree toward each process in the network takes $O(N^2\cdot\sqrt{E})$ messages and time in the worst case.

8.5 Packet Switching

Consider a network for which routing tables have been computed, so that all processes know how data packets should be forwarded through the network toward their destinations. On their way, these packets are stored at a buffer slot of a process until that process is certain the packet has arrived safely and has been stored in the buffer of the next process on the packet's route. When a packet reaches its destination, it is consumed; that is, removed from the network.

Even with cycle-free routing tables, a store-and-forward deadlock may occur when a group of packets are all waiting for the use of a buffer slot occupied by a packet in the group. A controller avoids such deadlocks by prescribing whether a new packet can be generated at a process or whether an existing packet can be forwarded to the next process, and possibly prescribing the buffer slot in which it is put. To avoid a trivial deadlock-free controller that disallows any generation of packets, it is generally required that generation of a new packet at a process with an empty buffer should always be allowed.

As noted, a process can eliminate a packet from its buffer only when it is sure the packet has arrived safely at the next process. For simplicity, we assume synchronous communication, which basically means that a process can send a packet only when the receiver is ready to receive it; that is, when it has a suitable buffer slot available. Thus, we abstract away from the communication overhead imposed by fruitless attempts to forward a packet.

Destination and Hops-So-Far Controllers

Consider a directed network of processes $p_0, ..., p_{N-1}$, and let T_i be the sink tree (with respect to the routing tables) with root p_i for $i = 0, ..., N - 1$. We first discuss two simple controllers based on these sink trees.

In the destination controller, each process carries N buffer slots, numbered from 0 to $N - 1$. The ith buffer slots at the processes are used to mimic the sink tree T_i.

- When a packet with destination p_i is generated at a process q, it is placed in the ith buffer slot of q.
- If qr is an edge in T_i, then a packet in the ith buffer slot of q can be forwarded to the ith buffer slot of r.

The destination controller is deadlock-free. This follows from the fact that, for each i, since T_i is acyclic, a packet in the ith buffer slot of any process can always travel to its destination. The clear drawback of this controller is that it requires N buffer slots per process, so it does not scale to large networks.

Let K be the length of a longest path in any T_i. In the hops-so-far controller, each process carries $K + 1$ buffer slots, numbered from 0 to K.

- When a packet is generated at a process q, it is placed in the 0th buffer slot of q.
- If qr is an edge in some T_i, then, for any $j < K$, a packet in the jth buffer slot of q can be forwarded to the $(j + 1)$th buffer slot of r.

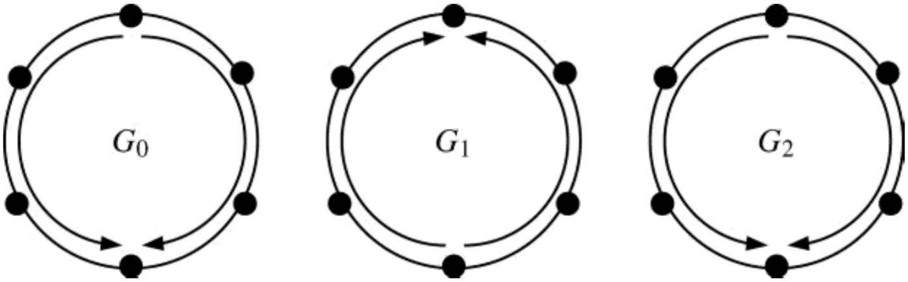
We argue that the hops-so-far controller is deadlock-free. It is easy to see, by induction on j, that no packet can get stuck at a $(K - j)$th buffer slot of any process, for $j = 0, ..., K$. The base case $j = 0$ is trivial, because a packet in a Kth buffer slot is guaranteed to be at its destination and so can be consumed. Now consider the inductive case: a packet in a $(K - (j + 1))$th buffer slot. By induction, packets in a $(K - j)$th buffer slot can be forwarded to their destinations and consumed. Hence, a packet in a $(K - (j + 1))$th buffer slot can either be consumed or forwarded to a $(K - j)$th buffer slot and from there to its destination, where it is consumed.

The hops-so-far controller typically requires between $\frac{D}{2}$ and D buffer slots per process. This is much better than for the destination controller but still not appealing for large networks.

Acyclic Orientation Cover Controller

An *acyclic orientation* of an undirected network G is a directed, acyclic network obtained by directing all the edges of G. Acyclic orientations $G_0, ..., G_{n-1}$ of G form an *acyclic orientation cover* of a set $\mathcal{P}$ of paths in G if each path in $\mathcal{P}$ is the concatenation of paths $P_0, ..., P_{n-1}$ in $G_0, ..., G_{n-1}$, respectively.

Example 8.5 For each undirected ring, there exists a cover, consisting of three acyclic orientations, of the collection of minimum-hop paths. For instance, in the case of a ring of size 6:

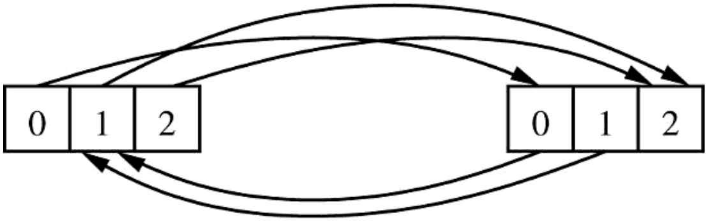


Consider an undirected network G that has been equipped with routing tables; $\mathcal{P}$ denotes the collection of paths in G according to the routing tables. Let $G_0, ..., G_{n-1}$ be an acyclic orientation cover of $\mathcal{P}$. In the acyclic orientation cover controller, each process in G has n buffer slots, numbered from 0 to $n - 1$.

- A packet generated at a process q is placed in the 0th buffer slot of q.
- Let qr be an edge in some G_i. A packet in the ith buffer slot of q can be forwarded to the ith buffer slot of r. Moreover, if $i < n - 1$, then a packet in the ith buffer slot of r can be forwarded to the $(i + 1)$th buffer slot of q.

Since all packets in G are routed via paths in $\mathcal{P}$, the acyclic orientation cover controller is deadlock-free. Consider a reachable configuration γ. Packets are forwarded and consumed until a configuration δ is reached in which forwarding and consumption are no longer possible. We argue by induction on j that in δ each $(n - j)$th buffer slot of any process is empty for $j = 1, ..., n$. In the base case $j = 1$, consider a packet that is being routed via a concatenation of paths $P_0, ..., P_{n-1}$ in $G_0, ..., G_{n-1}$. It is not hard to see, by the definition of the acyclic orientation cover controller, that when this packet is in a kth buffer slot, it is being routed via a P_ℓ with $\ell \geq k$. This implies that any packet in an $(n - 1)$th buffer slot is being routed via G_{n-1}. Since G_{n-1} is acyclic, a packet in an $(n - 1)$th buffer slot can always travel to its destination. Now consider the inductive case: a packet in an $(n - (j + 1))$th buffer slot. By induction, in δ all $(n - j)$th buffer slots are empty. Hence, packets in $(n - (j + 1))$th buffer slots can be consumed, forwarded via $G_{n-(j+1)}$ since it is acyclic, or forwarded to an $(n - j)$th buffer slot since these are empty. To conclude, in δ, all buffer slots are empty.

Example 8.6 For each undirected ring, there exists a deadlock-free controller that uses three buffer slots per process and allows packets to travel via minimum-hop paths. This follows from the fact that, according to example 8.5, undirected rings have a cover of the collection of minimum-hop paths that consists of three acyclic orientations. So the resulting acyclic orientation cover controller requires three buffer slots per process. The buffer slots between two neighboring processes are connected as follows, where the edge between these processes in the ring is directed from left to right in G_0 and G_2 and in the opposite direction in G_1.



8.6 Routing on the Internet

The routing approaches discussed so far were not designed to cope with large-sized and dynamic networks. Link-state routing is a pragmatic routing algorithm that has been geared to the Internet. It takes into account that processes may join the network or may crash temporarily or permanently and therefore not be available.

Each process periodically (and after a network change) sends a *link-state packet* to its neighbors, reporting the channels between the process and its direct neighbors, as well as their weights (typically based on latency or bandwidth). Moreover, it attaches a sequence number to these link-state packets, which is increased every time it broadcasts link-state packets to its neighbors. The link-state packets are flooded through the network; all processes store their content, so they obtain a local view of the entire network. Processes also store the sequence numbers of link-state packets on which their local view is based in order to prevent new information from being overwritten by old information. With its local view, a process can locally compute shortest paths using a uniprocessor algorithm (mostly Dijkstra's algorithm).

The crash failure and subsequent recovery of a process are eventually detected by, and taken into account in the link-state packets broadcast by, its neighbors. When a process recovers from a crash, its sequence number restarts at zero, so the link-state packets it broadcasts after the crash might be ignored by the other processes for a long time. Therefore, link-state packets carry a *time-to-live field*, defining the moment in time after which the packet becomes stale and its information may be discarded in favor of a link-state packet with possibly a lower sequence number but a higher time-to-live field. To reduce the overhead of flooding, each time a link-state packet is forwarded, its time-to-live field is decreased; when it becomes zero, the packet is discarded.

Link-state routing deals well with dynamicity but does not scale up to the size of the Internet, because it uses flooding. Therefore, the Internet is divided into what are called autonomous systems, which are basically different subnetworks that use link-state routing (notably by means of the OSPF Protocol) within their own domain.

Routing between autonomous systems is performed with the Border Gateway Protocol, in which peer routers exchange reachability information, meaning that a router informs its neighbors about updates in its routing table either because it noticed a topology change or as a result of an update in the routing table of one of its neighbors. Thus, each router maintains an up-to-date routing table based on autonomous system connectivity. When a router connects to the network for the first time, other routers provide it with their entire routing table.

The Transmission Control Protocol (TCP) aims to guarantee reliable data delivery over the Internet. In this protocol, to control congestion in the network, every process maintains a congestion window for each of its channels. Packets are acknowledged by the receiver, and the congestion window of a channel provides an upper bound on the number of unacknowledged packets a process is allowed to have sent into this channel. The congestion window grows linearly with each received acknowledgment, up to some threshold. The congestion window may effectively double in size during every round-trip time (that is, the time between sending a packet and receiving the corresponding acknowledgment) if all packets are being acknowledged (see exercise 8.25). The congestion window is reset to the initial size (in TCP Tahoe) or halved (in TCP Reno) with each lost data packet.

Bibliographical notes

The Chandy-Misra algorithm originates from [19], the Merlin-Segall algorithm from [69], Toueg's algorithm from [94], and Frederickson's algorithm from [39] (where the algorithm is only sketched). The destination and hops-so-far controllers were proposed in [68], and the acyclic orientation cover controller was proposed in [92]. The mechanisms underlying link-state routing were put forward in [65], and an earlier version of link-state routing was used in ARPANET [66]. Congestion windows can be traced back to [49].

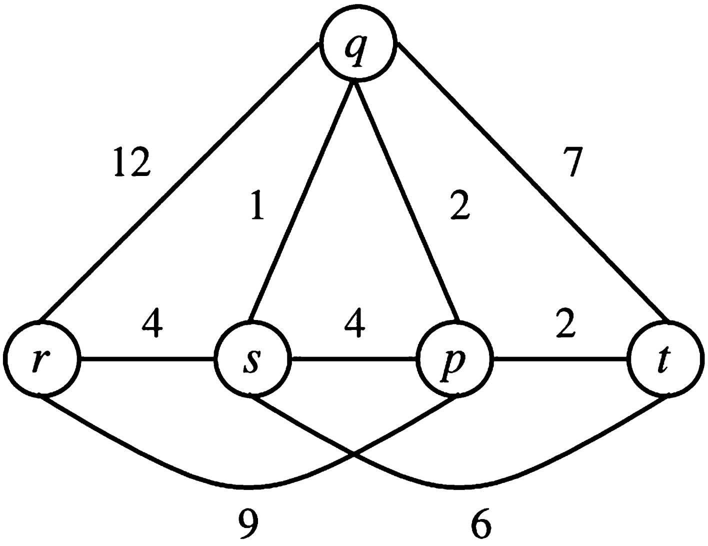
8.7 Exercises

Exercise 8.1 Explain in detail how the Dijkstra-Scholten algorithm detects termination of the Chandy-Misra algorithm.

Exercise 8.2 Adapt the Dijkstra-Scholten algorithm so that termination is detected in the Chandy-Misra algorithm without building two distinct sink trees (that is, no separate sink tree is needed for detecting termination).

Exercise 8.3 Let n range over the natural numbers. Generalize example 8.1 to a network with $2{\cdot}n + 1$ processes and $3{\cdot}n$ weighted channels, for which the number of messages sent by the Chandy-Misra algorithm in the worst case grows exponentially (in n). Explain why this is the case.

Exercise 8.4 Run the Merlin-Segall algorithm on the following undirected weighted network to compute all shortest paths toward process t. Give a computation that takes four rounds before the correct sink tree has been computed.



Exercise 8.5 Suppose that in the Merlin-Segall algorithm a process q updates $parent_q$ each time it updates $dist_q$. Explain what would go wrong.

Exercise 8.6 Run Toueg's algorithm on the network in exercise 8.4. Take as pivot order $p\,q\,r\,s\,t$.

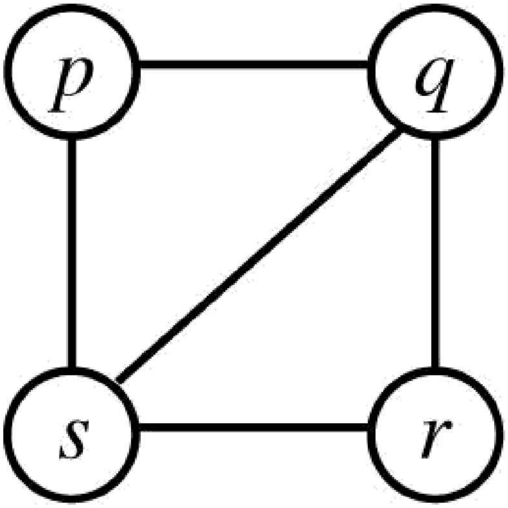
Exercise 8.7 Argue that Toueg's algorithm is an all-pairs shortest-path algorithm.

Exercise 8.8 Analyze the space complexity of Toueg's algorithm.

Exercise 8.9 In Toueg's algorithm, when a process $p \neq r$ in the sink tree of the pivot r receives the distance values of r, let p first perform for each process q the check whether $dist_p(r)+dist_r(q) < dist_p(q)$. Explain why p needs to forward only those values $dist_r(q)$ for which this check yields a positive result.

Exercise 8.10 Suppose that channels can carry negative weights. Explain how the output of Toueg's algorithm can be used to detect the presence of a negative-weight cycle (of at least three channels).

Exercise 8.11 Apply Frederickson's algorithm with $\ell = 1$ (the "simple" algorithm) to the following undirected network in order to find a breadth-first search tree rooted in p. Do the same with $\ell = 2$.



Exercise 8.12 In Frederickson's algorithm, consider a process located k hops from the initiator, with $k \neq \ell \cdot n$ for all n. Argue that this process is guaranteed to receive a message ⟨**reverse**, k +1, _⟩ or ⟨**explore**, j⟩ with $j \in \{k, k+1, k+2\}$ from all its neighbors.

Exercise 8.13 Give a computation of Frederickson's algorithm on an undirected ring of size 3 and with $\ell = 2$ to show that a **forward** can be sent to a process that is not a child of the sender.

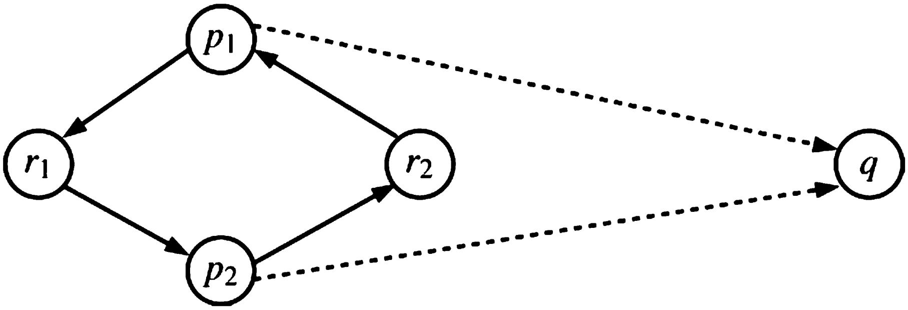
Exercise 8.14 Argue that Frederickson's algorithm establishes a breadth-first search tree toward the initiator.

Exercise 8.15 Analyze the message and time complexity of Frederickson's breadth-first search algorithm, taking into account the network diameter D.

Exercise 8.16 Develop a distributed version of Dijkstra's celebrated single-source shortest-path algorithm for undirected weighted networks. Discuss the worst-case message and time complexity of your algorithm.

Exercise 8.17 [92] Show that there does not exist a deadlock-free controller that uses only one buffer slot per process and allows each process to send packets to at least one other process.

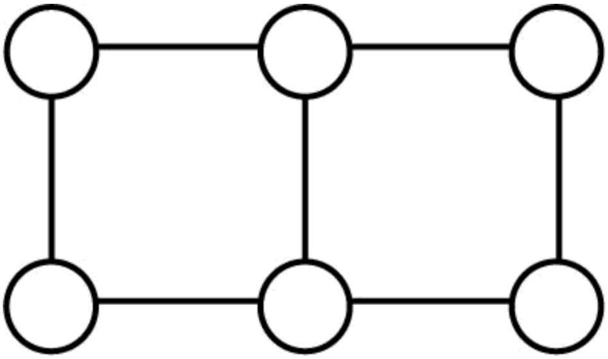
Exercise 8.18 [92] Show that the destination controller is not deadlock-free if packet routing is as follows.



In this picture, packets from p_1 to q are routed via the path $p_1\ r_1\ p_2 \cdots q$, and packets from p_2 to q are routed via the path $p_2\ r_2\ p_1 \cdots q$.

Exercise 8.19 Argue that the acyclic orientation cover of a ring of size 6 in example 8.5 covers all shortest paths in this ring.

Exercise 8.20 Give an acyclic orientation cover G_0, G_1 of a set of paths in the following undirected network that contains for each pair of processes p, q a minimum-hop path from p to q.



Describe in detail how the buffer slots are linked in the corresponding acyclic orientation cover controller.

Exercise 8.21 Given the undirected cube, prove that there is an acyclic orientation cover G_0, G_1 such that between every two processes in the cube there is a minimum-hop path that is the concatenation of paths in G_0 and G_1.

Exercise 8.22 Show that for any acyclic undirected network there exists a deadlock-free controller that uses only two buffer slots at each process.

Exercise 8.23 Give an example to show that a *cyclic* orientation cover controller (in which the orientations G_i are allowed to contain cycles) is not always deadlock-free.

Exercise 8.24 Why does the link-state algorithm become less efficient if processes broadcast their entire routing table instead of only their channels and weights?

Exercise 8.25 Argue why a congestion window may effectively double in size during every round-trip time.

9
Election

In an election algorithm, the processes in the network elect one process among them as their leader. The aim is usually to let the leader act as the organizer of some distributed task, for example in the role of the root of a spanning tree of the network, the initiator of a centralized algorithm, the central decision point, or the assembly point of information. Each computation starts in a configuration in which the processes are unaware which process will serve as the leader and must terminate in a configuration where exactly one process is the leader.

Election algorithms are decentralized: the initiators can be any nonempty set of processes. We require that all processes have the same local algorithm. This disallows the trivial solution where exactly one process has the algorithm "I am the leader." Process IDs are supposed to be unique and from a totally ordered set. In chapter 10, we will see that unique IDs are essential for constructing election algorithms that always terminate.

9.1 Election in Rings

We first consider three election algorithms for ring topologies. In each of these algorithms, the initiators determine among themselves which one has the highest ID. This initiator becomes the leader. Initially, the initiators are active, while all noninitiators are passive. In all three algorithms, passive processes are out of the race to become the leader and simply pass on messages.

Chang-Roberts Algorithm

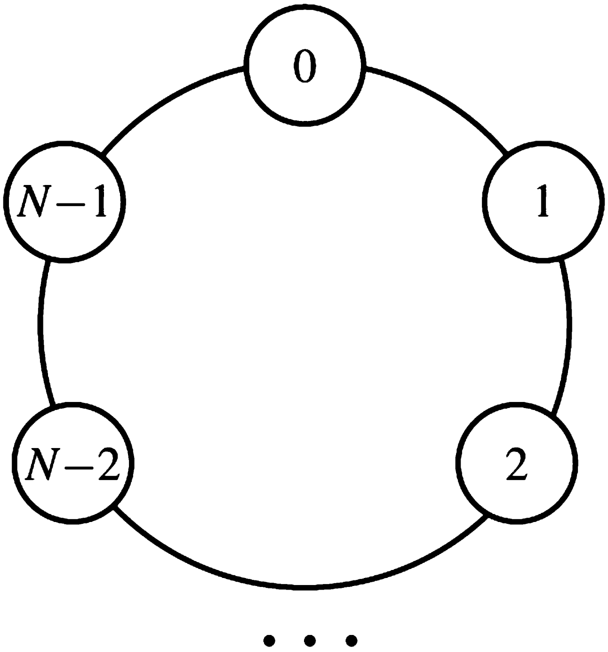
The Chang-Roberts algorithm targets a directed ring. Since networks are assumed to be strongly connected, a directed ring is oriented either in a clockwise or in a counterclockwise fashion.

Initially, the initiators send a message to the next process in the ring, tagged with their ID. When an active process p receives a message tagged with q, there are three cases:

- If $q < p$, then p dismisses the message.
- If $q > p$, then p becomes passive and passes on the message.
- If $q = p$, then p becomes the leader.

The idea behind the Chang-Roberts algorithm is that only the message with the highest ID will complete the round trip, because every other message is stopped, at the latest, when it arrives at the initiator with the highest ID (by the first case). Moreover, initiators that do not have the highest ID are made passive, at the latest, when they receive the message with the highest ID (by the second case). When an initiator receives back its own message, it knows it is the leader (by the third case).

Example 9.1 In the ring that follows, all processes are initiators. If the ring is directed counterclockwise, then it takes $\frac{1}{2}\cdot N\cdot(N+1)$ messages to elect process $N - 1$ as the leader. Each message is stopped at process $N - 1$, so the message from process i travels $i + 1$ hops for $i = 0, \ldots, N - 1$; and $1+2+\cdots+N=\frac{1}{2}\cdot N\cdot(N+1)$.



If the ring is directed clockwise, then it takes only $2 \cdot N - 1$ messages to elect process $N - 1$ as the leader. Each message is stopped after one hop, except for the message from process $N - 1$, which travels N hops.

The preceding example shows that the worst-case message complexity of the Chang-Roberts algorithm is $O(N^2)$. It can be shown, however, that the average-case message complexity is $O(N \cdot \log N)$.

Franklin's Algorithm

Franklin's algorithm, which requires an undirected ring, improves on the worst-case message complexity of the Chang-Roberts algorithm. In an election round, each active process p compares its own ID with the IDs of its nearest active neighbors on both sides. If p's ID is the largest of the three IDs, then p proceeds to the next election round. If one of the other IDs is larger than p's ID, then p becomes passive. And if p receives its own ID from either side, then it becomes the leader, because there are no other active processes left in the ring.

To be more precise, at the start of an election round, each active process sends its ID to its neighbors on either side. When an active process p has received messages tagged with q and r from either side, there are three cases:

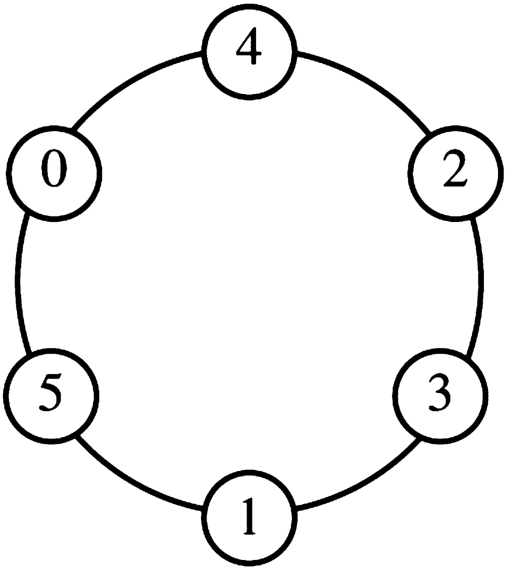
- If $\max\{q, r\} < p$, then p enters another election round by sending its ID in both directions again.

- If $\max\{q, r\} > p$, then p becomes passive.
- If $\max\{q, r\} = p$, then p becomes the leader.

Since a message can overtake another message from the previous round, processes need to keep track of the parity of their current round number, meaning whether the round number is even or odd. They must attach this parity to the message containing their ID; see the pseudocode of the Dolev-Klawe-Rodeh algorithm in the appendix.

Example 9.2 On the undirected version of the ring in example 9.1, Franklin's algorithm terminates in two rounds. In the first round, only process N -1 remains active. In the second round, process $N - 1$ finds that it is the leader.

Example 9.3 We run Franklin's algorithm on the following undirected ring, in which all processes are initiators.



- Processes 3, 4, and 5 progress to the second election round, because their IDs are larger than those of their (active) neighbors. Processes 0, 1, and 2 become passive in the first round.
- Only process 5 progresses to the third round, because in the second round it finds that its ID is larger than those of its nearest active neighbors, 3 and 4. Conversely, the latter two processes become passive in the second round, because they find that their nearest active neighbor, 5, has a larger ID.
- Finally, in the third round, process 5 finds that it is the leader, because the two

messages it sends in either direction complete the round trip.

The worst-case message complexity of Franklin's algorithm is $O(N \cdot \log N)$. In each round with two or more active processes, at least half of the active processes become passive because for each pair of nearest active neighbors at least one becomes passive. In the final round, the remaining active process becomes the leader. So there are at most $\lfloor \log_2 N \rfloor + 1$ rounds. And each round takes $2 \cdot N$ messages (two messages per channel).

Dolev-Klawe-Rodeh Algorithm

The Dolev-Klawe-Rodeh algorithm transposes the idea behind Franklin's algorithm to a directed ring. In that setting, messages cannot travel in both directions, so an active process cannot easily compare its own ID p with the IDs q and r of its nearest active neighbors. This is resolved by performing this comparison not at p but at its next (in the direction of the ring) active neighbor, r. That is, the IDs p and q are collected at r. If p is larger than q and r, then r remains active and progresses to the next election round, in which it assumes the ID p. If p is smaller than q or r, then r becomes passive. And if p equals q and r, then r becomes the leader. If we really want the initiator with the largest ID to become the leader, then in the last case r could send a special leader message tagged with its last ID around the ring in order to inform the initiator that started the election with this ID that it is the leader.

To be more precise, at the start of an election round, each active process sends its ID to its next neighbor, with a 0 attached. When an active process r receives this message $\langle \mathbf{id}, p, 0 \rangle$, it stores the ID p and passes on the message with a 1 attached. And when r receives a message $\langle \mathbf{id}, q, 1 \rangle$, it stores the ID q. The bits 0 and 1 allow r to determine that p is located between q and r. Now there are three cases:

- If $\max\{q, r\} < p$, then r enters another round with the new ID p by sending $\langle \mathbf{id}, p, 0 \rangle$.
- If $\max\{q, r\} > p$, then r becomes passive.
- If $\max\{q, r\} = p$, then r becomes the leader (or sends out a special leader message tagged with r).

Since a message can overtake another message from the previous round, processes need to keep track of the parity of their current round number and must attach this parity to the message containing their ID; see the pseudocode in the appendix.

Example 9.4 Just like Franklin's algorithm, the Dolev-Klawe-Rodeh algorithm takes just two rounds to terminate on the undirected version of the ring in example 9.1.

Example 9.1 shows that an active process r that collects IDs p and q in an election round must really act as if it is the middle process, p. If r would proceed to the next round if its own ID was the largest, then it is easy to see that the message complexity of the Dolev-Klawe-Rodeh algorithm on the ring in example 9.1 would become $O(N^2)$. Because then in each successive round, only the active process with the smallest ID among all active processes would become passive.

Example 9.5 We run the Dolev-Klawe-Rodeh algorithm on the ring in example 9.3, oriented in the clockwise direction.

- Processes 0, 1, and 2 progress to the second election round because they act like processes 5, 3, and 4, respectively. For instance, process 0 collects the IDs 5 (with a 0 attached) and 1 (with a 1 attached), concludes that 5 is the largest ID among 0, 5, and 1, and progresses to the second round with the ID 5, and likewise for processes 1 and 2. Processes 3, 4, and 5 become passive in the first round because they act like processes 2, 0, and 1, respectively.
- Only process 2 (which assumed the ID 4 for the second round) progresses to the third round, because in the second round it collects the IDs 5 (with a 0 attached) and 3 (with a 1 attached) and concludes that 5 is the largest ID among 4, 5, and 3. Processes 0 (which assumed the ID 5) and 1 (which assumed the ID 3) become passive in the second round.
- Finally, in the third round, process 2 (which assumed the ID 5 for the third round) finds that it is the leader, because it receives back its own ID, first with a 0 attached and next with a 1 attached. Alternatively, process 2 can announce to the other processes that 5 is the largest ID in the ring, after which the process that carried the ID 5 at the start of the election becomes the leader.

The worst-case message complexity of the Dolev-Klawe-Rodeh algorithm is the same as that of Franklin's algorithm: $O(N{\cdot}\log N)$. For there are at most $\lfloor \log_2 N \rfloor + 1$ rounds, each taking $2{\cdot}N$ messages. It can be shown that $\Omega(N{\cdot}\log N)$ is a lower bound on the average-case message complexity of any election algorithm for rings.

9.2 Tree Election Algorithm

The tree algorithm from section 4.2 serves as the basis for an election algorithm in acyclic undirected networks. The idea is that each process p collects IDs from its children, computes the maximum of these IDs and its own ID, and sends this maximum to its parent. The two decision events determine the overall maximum among the IDs in the network. All processes are informed of this maximum.

In election algorithms, the initiators can be any nonempty set of processes, while the tree algorithm starts from all the leaves in the network. Therefore, the tree election algorithm is booted by a wake-up phase, driven by the initiators, which send a wake-up message to all their neighbors. These wake-up messages are then flooded through the network: when a noninitiator receives its first wake-up message, it wakes up and sends a wake-up message to its neighbors. Messages from the tree election algorithm that arrive before the receiver has woken up are buffered.

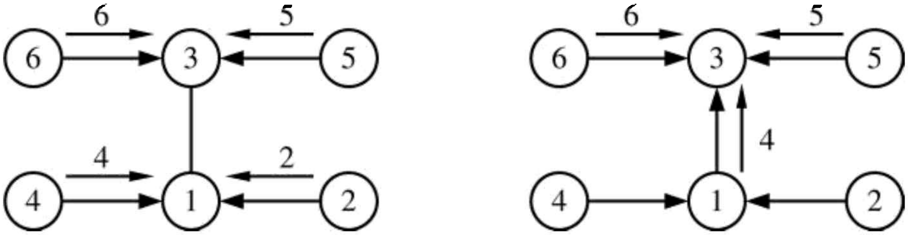
The local algorithm at an awake process p is as follows:

- p waits until it has received IDs from all its neighbors except one, which becomes its parent.
- p computes the largest ID $\max_p$ among the received IDs and its own ID.
- p sends a parent message to its parent, tagged with $\max_p$.
- If p receives a parent message from its parent, tagged with r, then it computes $\max_p'$, being the maximum of r and $\max_p$. Next, p sends an information message to all neighbors except its parent, tagged with $\max_p'$.
- If p receives an information message from its parent, tagged with the largest ID in the network, then it forwards this message to its children.

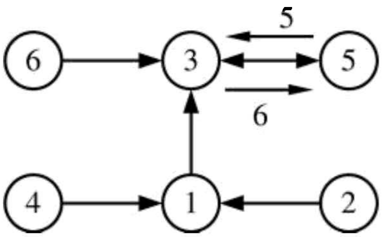
Thus, the information message is forwarded through the entire network, and eventually the process with the largest ID becomes the leader.

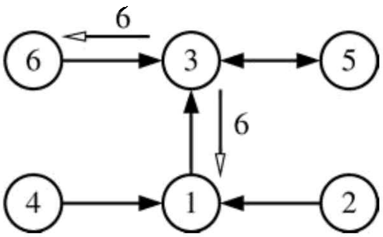
The only tricky part of the algorithm is at the two neighbors p and q where the decide events happen. When p receives a parent message from q, tagged with an ID r, it must compute the maximum of r and the value of $\max_p$ it computed earlier. Likewise, q must compute the maximum of the ID it receives from p and the value of $\max_q$ it computed earlier. The reason is that p and q computed the maximum among all IDs in disjoint parts of the network, which together cover the entire network.

Example 9.6 In the pictures here, we consider one possible computation of the tree election algorithm on the network. The wake-up phase is omitted. Messages that are depicted with a solid arrowhead are toward a parent.

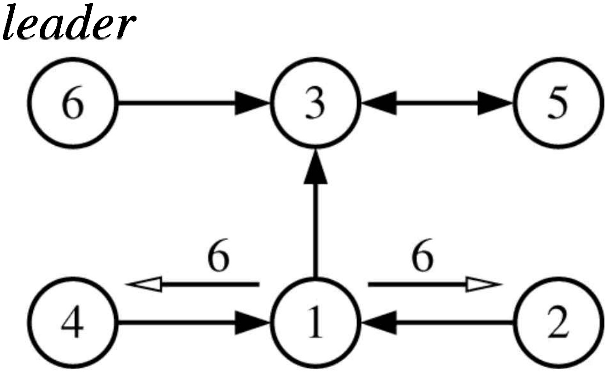


In the first picture, the leaves 2, 4, 5, and 6 have selected their only neighbor as their parent and have sent their ID to their parent. In the second picture, process 1 has received the messages from 2 and 4, calculated the maximum, 4, made its remaining neighbor, 3, its parent, and sent the maximum to its parent.





In the third picture, process 3 has received the messages from 1 and 6, calculated the maximum, 6, made its remaining neighbor, 5, its parent, and sent the maximum to its parent. In the fourth picture, processes 3 and 5 have received each other's messages, and calculated the maximum, 6, concluding that 6 must become the leader; process 3 has, moreover, sent 6 to its two children.



In the fifth picture, processes 1 and 6 have received the message from 3, process 6 has become the leader, and process 1 has sent the ID 6 to its two children. When these two messages have arrived, the algorithm has terminated.

Just like the tree algorithm, the tree election algorithm takes $2 \cdot N - 2$ messages: two messages per channel. The wake-up phase also takes $2 \cdot N - 2$ messages.

9.3 Echo Algorithm with Extinction

We now discuss an election algorithm for undirected networks that works for any topology. The idea is to let each initiator start a run of the echo algorithm from section 4.3, tagged with its ID. Only the wave started by the initiator with the largest ID completes, after which that initiator becomes the leader. Noninitiators join the first wave that hits them. If a process in a wave is hit by another wave with a larger ID, then it switches to that wave; if it is hit by another wave with a smaller ID, it dismisses that wave message.

At any time, each process takes part in at most one wave. Suppose a process *p* that is participating in a wave tagged with *q* is hit by a wave tagged with *r*.

- If $q < r$, then *p* makes the sender its parent, switches to the wave tagged with *r* (it abandons all the wave messages it received earlier), and treats the incoming message accordingly.
- If $q > r$, then *p* continues with the wave tagged with *q* (it dismisses the incoming message).
- If $q = r$, then *p* treats the incoming message according to the echo algorithm of the wave tagged with *q*.

If the wave tagged with *p* completes by executing a decide event at *p*, then *p* becomes the leader.

Waves of initiators that do not have the largest ID among all initiators will not complete (that is, they are extinguished), because the initiator with the largest ID will refuse to take part in those waves. Conversely, the wave of the initiator with the largest ID is guaranteed to complete, because each process will eventually switch to that wave.

Example 9.7 We consider a computation of the echo algorithm with extinction on an undirected ring of three processes, 0, 1, and 2, all of which are initiators.

- The three processes all start a wave and send a wave message to their two neighbors, tagged with their ID.
- The wave messages from 0, tagged with 0, are dismissed by 1 and 2.
- 0 receives the wave message from 1, tagged with 1. As a result, 0 switches to 1’s wave, makes 1 its parent, and sends a wave message to 2, tagged with 1.
- The wave messages from 0 and 1, tagged with 1, are dismissed by 2.
- 0 and 1 receive the wave message from 2, tagged with 2. As a result, they switch to 2’s wave, make 2 their parent, and send wave messages to each other, tagged with 2.
- 0 and 1 receive each other’s wave messages, tagged with 2. Next, they send wave messages tagged with 2 to their parent 2.
- 2 receives the wave messages from 0 and 1, tagged with 2. As a result, 2’s wave decides and 2 becomes the leader.

The worst-case message complexity of this echo algorithm with extinction is $O(N{\cdot}E)$: there are at most *N* waves, and each wave uses at most $2{\cdot}E$ messages.

9.4 Minimum Spanning Trees

We now turn our attention to a topic that at first sight has little to do with election:

minimum spanning trees. However, the distributed algorithm for constructing a minimum spanning tree that is discussed in this section will turn out to yield an efficient election algorithm.

An undirected network is given, in which the channels carry positive weights. A *minimum* spanning tree is a spanning tree of the network for which the sum of the weights of its channels is minimal. Such spanning trees can be employed, for example, to minimize the cost of broadcasting messages through the network or of distributing electricity over an electrical grid.

For convenience, we assume that different channels in the network always have different weights; this guarantees that the minimum spanning tree is unique. Alternatively, we could allow different channels to have the same weight and could impose a total order on channels with the same weight by using the IDs of the end points of a channel.

The Gallager-Humblet-Spira algorithm is a distributed version of Kruskal's famous algorithm for computing minimum spanning trees in a uniprocessor setting. We first briefly discuss Kruskal's algorithm, which uses the notion of a *fragment*, being any connected subgraph of the minimum spanning tree. A channel in the network is said to be an *outgoing edge* for a fragment if exactly one of the processes connected by the channel is in the fragment. Kruskal's algorithm is based on the observation that the lowest-weight outgoing edge c of a fragment F is always in the minimum spanning tree. Suppose, toward a contradiction, that this is not the case. Then the minimum spanning tree extended with c would contain a cycle, which would include c and another outgoing edge d of F. Replacing d by c in the minimum spanning tree would give another spanning tree of the network, and the sum of the weights of its channels would be smaller than for the minimum spanning tree. This is a contradiction.

In Kruskal's algorithm, initially each process forms a separate fragment. In each step, two different (disjoint) fragments are joined into one fragment via a channel between these fragments that is the lowest-weight outgoing edge for at least one of the two fragments. The algorithm terminates when only one fragment remains.

In a distributed setting, it is complicated for a process to decide whether its channels are outgoing edges or not. For each of its channels, it must communicate with the process at the other side to find out whether it is in the same fragment. If one of its channels turns out to be an outgoing edge, the process needs to work together with the other processes in its fragment to determine whether it is the lowest-weight outgoing edge for the fragment. And when finally the lowest-weight outgoing edge for the fragment has been detected, the fragment at the other side of the channel has to be asked to join together so that the two fragments become one.

In the Gallager-Humblet-Spira algorithm, each fragment carries a *name*, which is a nonnegative real number, and a *level*, which is a natural number. Processes keep track of the name and level of their fragment. Each fragment has a unique name, except initially, when each process starts as a fragment with name and level 0. The level of a fragment is the maximum number of joins any process in the fragment has experienced. When two fragments join, there are two scenarios. If the two joining fragments have different levels, the one with the lowest level copies the name and level of the other fragment (in which case the processes in the other fragment do not experience the join). If they have the same level, the new name of the joint fragment is the weight of the *core edge* via which they are joined, and their level is increased by 1.

The core edge of a fragment, which is the last channel via which two of its subfragments were joined at the same level, plays a key role in the algorithm. It is the central computing unit of the fragment, to which the processes in the fragment report the lowest-weight outgoing edge they are aware of, and from which the join via the lowest-weight outgoing edge of the fragment is initiated. Each process has a parent, toward the core edge of its fragment (except initially, when fragments consist of a single process). The end points of a core edge are called the *core nodes*; they have each other as parent.

Processes are in one of the following three states:

- *sleep*: This is a special state for noninitiators, so that this algorithm for computing minimum spanning trees can be easily turned into an election algorithm, where we must allow for any nonempty set of initiators. A process that is asleep wakes up as soon as it receives any message.
- *find*: The process is looking for its lowest-weight outgoing edge and/or waiting for its children to report the lowest-weight outgoing edge they are aware of.
- *found*: The process has reported the lowest-weight outgoing edge it is aware of to its parent and is waiting either for an instruction from the core edge that a join should be performed via that channel or for a message informing it that a join has been completed elsewhere in the fragment.

Moreover, processes maintain a status for each of their channels:

- *basic edge*: It is undecided whether the channel is part of the minimum spanning tree.
- *branch edge*: The channel is part of the minimum spanning tree.
- *rejected*: The channel is not part of the minimum spanning tree.

Each initiator, and each noninitiator after it has woken up, sets its lowest-weight channel to *branch*, its other channels to *basic*, and its state to *found*. It sends the message ⟨**connect**, 0⟩ into the branch edge to inform the fragment at the other side that it wants to join via this channel and that its fragment has level 0.

Let two fragments, one with name *fn* and level ℓ and the other with name *fn*′ and level ℓ', be joined via channel *pq*, where *p* is in the first fragment and *q* is in the second fragment. Let $\ell \leq \ell'$. As explained earlier, there are two possible scenarios:

- If $\ell < \ell'$, then in the past *p* sent ⟨**connect**, ℓ⟩ to *q*, and now *q* sends the message

⟨**initiate**, *fn*′ , ℓ′ , $\frac{\textit{find}}{\textit{found}}$⟩ to *p*. We write $\frac{\textit{find}}{\textit{found}}$ to express that this parameter can be either *find* or *found*, depending on the state *q* is in.

- If ℓ = ℓ′ , then in the past *p* and *q* sent ⟨**connect**, ℓ⟩ to each other, and now they send the message ⟨**initiate**, *weight*(*pq*), ℓ + 1, *find*⟩ to each other.

Upon receipt of a message ⟨**initiate**, f_n, ℓ, $\frac{\textit{find}}{\textit{found}}$⟩, a process stores *fn* and ℓ as the name and level of its fragment, assumes the state *find* or *found* depending on the last parameter in the message, and adopts the sender as its parent. It passes on the message through its other branch edges.

In the first scenario, *q* is in the fragment with the higher level, so its **initiate** message imposes the name and level of its fragment onto *p*. Moreover, *p* makes *q* its parent, toward the core edge of *q*'s fragment. By forwarding the **initiate** message through its other branch edges, *p* ensures that all other processes in its fragment update the name and level of their fragment and select a new parent toward the core edge of *q*'s fragment.

In the second scenario, both fragments have the same level, so the joint fragment gets a new name, level, and core edge. The new name is the weight of the channel *pq*, the new level is the old level plus 1, and the new core edge is *pq*. Since all processes in both fragments must be informed, *p* and *q* send an **initiate** message with the new name and level to each other. As this message is forwarded through the branch edges, the processes in the joint fragment select a new parent toward the core edge *pq*. The parameter *find* in the **initiate** message makes sure that these processes, moreover, start looking for the lowest-weight outgoing edge of the joint fragment.

When a process *p* receives a message ⟨**initiate**, *fn*, ℓ, *find*⟩, it checks in increasing order of weight whether one of its basic edges *pq* is outgoing by sending ⟨**test**, *fn*, ℓ⟩ to *q* and waiting for an answer from *q*. A basic edge *pq* that was found to be outgoing earlier must be tested again because in the meantime the fragments of *p* and *q* may have joined via some other channel. Upon receipt of the **test** message, *q* acts as follows:

- If ℓ is greater than the level of *q*'s fragment, then *q* postpones processing the incoming **test** message until the level of *q*'s fragment has reached or surpassed ℓ. The reason for this postponement is that *p* and *q* might actually be in the same fragment, in which case the message ⟨**initiate**, *fn*, ℓ, *find*⟩ is on its way to *q*.
- If ℓ is not greater than the level of *q*'s fragment, then *q* checks whether the name of *q*'s fragment is *fn*. If so, then *q* replies with a **reject** message to *p* (except if *q* is awaiting a reply to a **test** message to *p*, because then *p* and *q* can interpret each other's messages ⟨**test**, *fn*, ℓ⟩ as a **reject**), and as a result, *p* and *q* both set the status of the channel *pq* to *rejected*. If not, then *q* replies with an **accept** message to *p*.

When a basic edge *pq* is accepted or there are no basic edges at *p* left, *p* stops the search for its lowest-weight outgoing basic edge.

Moreover, p waits for **report** messages through its branch edges, except the one to its parent. Then p sets its state to *found* and computes the minimum λ_p of these reports and the weight of its lowest-weight outgoing basic edge (or ∞, if no such edge was found). If $\lambda_p < \infty$, then p stores the branch edge through which it received λ_p, or its outgoing basic edge of weight λ_p. Finally, p sends ⟨**report**, λ_p⟩ to its parent.

Only the core nodes receive a **report** message from their parent. Thus, the core nodes can determine the weight μ of the lowest-weight outgoing edge of their fragment. If μ is ∞, then the core nodes terminate, because the entire minimum spanning tree has been computed. If μ is not ∞, then the core node that received μ first sends a **changeroot** message via branch edges toward the process p that originally reported a lowest-weight outgoing basic edge pq with weight μ. Here it is utilized that processes stored their branch edge that reported μ. From then on, the core edge becomes a regular tree edge. When p receives the **changeroot** message, it sets the channel pq to *branch* and sends ⟨**connect**, ℓ⟩ into it, where ℓ is the level of p's fragment.

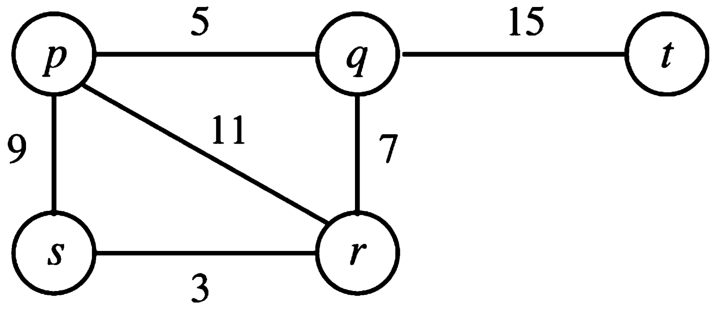
Let a process q receive a message ⟨**connect**, ℓ⟩ from a neighbor p. We note that the level of q's fragment is at least ℓ: either $\ell = 0$, or q earlier sent **accept** to p, which it could only do if its fragment had a level of at least ℓ, and fragment levels never decrease over time. Now q acts as follows:

- As long as q's fragment also has level ℓ and qp is not a branch edge, q postpones processing the **connect** message from p. The reason for this postponement is that q's fragment might be in the process of joining a fragment with a level $\geq \ell$, in which case p's fragment should subsume the name and level of that joint fragment instead of joining q's fragment at an equal level.
- If the level ℓ' of q's fragment is or becomes greater than ℓ, then q sets the channel qp to *branch* and sends ⟨**initiate**, fn' , ℓ' , $\frac{find}{found}$⟩ to p, where fn' is the name of q's fragment and *find* or *found* depends on the state of q.
- If q's fragment has level ℓ and qp is or becomes a branch edge (in which case q sent ⟨**connect**, ℓ⟩ to p), then q sends ⟨**initiate**, *weight*(qp), $\ell + 1$, *find*⟩ to p (and vice versa).

In the last case, pq becomes the core edge of the joint fragment.

This completes the description of the Gallager-Humblet-Spira algorithm; we are back at the situation where two fragments are joined at a different level or the same level by means of one or two **initiate** messages, respectively, which was explained earlier.

Example 9.8 We consider one possible computation of the Gallager-Humblet-Spira algorithm on the following network, in which all processes are initiators.



- *p* and *q* send ⟨**connect**, 0⟩ to each other, and both make *pq* a branch edge. Since these fragments join at the same level, 0, *p* and *q* send ⟨**initiate**, 5, 1, *find*⟩ to each other. Next, *p* and *q* send ⟨**test**, 5, 1⟩ to *s* and *r*, respectively.
- *t* sends ⟨**connect**, 0⟩ to *q* and makes *tq* a branch edge. Since the fragment of *q* is at level 1, *q* replies with ⟨**initiate**, 5, 1, *find*⟩. Then *t* sends ⟨**report**, ∞⟩ to its new parent *q*, since *t* has no other channels except *tq*.
- *r* and *s* send ⟨**connect**, 0⟩ to each other, and both make *rs* a branch edge. Since these fragments join at the same level, 0, *r* and *s* send ⟨**initiate**, 3, 1, *find*⟩ to each other. Next, *r* and *s* send ⟨**test**, 3, 1⟩ to *q* and *p*, respectively.
- Since the fragments of *r* and *s* are at the same level as the fragments of *q* and *p* but have different names, *r* and *s* reply to the **test** messages from *q* and *p*, respectively, with an **accept** message. As a result, *p* sends ⟨**report**, 9⟩ to its parent *q*, while *q* sends ⟨**report**, 7⟩ to its parent *p*. Since 7 is smaller than 9, *q* sends ⟨**connect**, 1⟩ to *r*.
- Since the fragments of *p* and *q* are at the same level as the fragments of *s* and *r* but have different names, *p* and *q* can reply to the **test** messages from *s* and *r*, respectively, with an **accept** message. As a result, *r* sends ⟨**report**, 7⟩ to its parent *s*, while *s* sends ⟨**report**, 9⟩ to its parent *r*. Since 7 is smaller than 9, *r* sends ⟨**connect**, 1⟩ to *q*.
- By the crossing ⟨**connect**, 1⟩ messages between *q* and *r*, the channel between these processes becomes a branch edge as well as the core edge of the new fragment, which has level 2. Messages ⟨**initiate**, 7, 2, *find*⟩ are forwarded through the branch edges. The channels *ps* and *pr* are tested (in this order) from either side, both times leading to a reject. Finally, all processes report ∞ to the core edge *qr*, and the algorithm terminates.

The computed minimum spanning tree consists of the channels *pq*, *qr*, *qt*, and *rs*.

We argue that the Gallager-Humblet-Spira algorithm is deadlock-free. There are two potential causes for deadlock, because a process *q* may postpone an incoming message. The first case is if *q* receives a message ⟨**test**, *fn*, ℓ⟩ while ℓ is greater than the level of *q*'s fragment. This postponement does not lead to a deadlock, because there is always a fragment of minimal level, and **test** messages originating from this fragment will be answered promptly. The second case is if q receives a message ⟨**connect**, ℓ⟩ while the level of q's fragment equals ℓ and qp is not a branch edge. This postponement also does not lead to a deadlock, because different edges have different weights, so there cannot be a cycle of fragments waiting for a reply to a postponed **connect** message (see exercise 9.11).

The worst-case message complexity of the Gallager-Humblet-Spira algorithm is $O(E + N \cdot \log N)$. The summand E is the accumulation of all messages for rejecting channels: each channel outside the minimum spanning tree is rejected by a **test-reject** or **test-test** pair. This adds up to $2 \cdot (E - (N - 1))$ messages in total. Furthermore, each process experiences at most $\lfloor \log_2 N \rfloor$ joins, because each time this happens, the level at the process increases, and a fragment at a level ℓ contains at least 2^{ℓ} processes (see exercise 9.6). Every time a process experiences a join, it receives an **initiate** message and may send the following messages: one **test** that triggers an **accept**, a **report**, and a **changeroot** or **connect**. Including the **accept**, there are five messages every time a process experiences a join, adding up to at most $5 \cdot N \cdot \lfloor \log_2 N \rfloor$ messages in total. This analysis covers all messages in the algorithm.

Finally, as promised at the start of this section, we return to election. By two extra messages at the very end of the Gallager-Humblet-Spira algorithm, the core node with the largest ID can become the leader. This yields an election algorithm for undirected networks.

The lower bound of $\Omega(N \cdot \log N)$ for rings, mentioned earlier, implies a lower bound of $\Omega(E + N \cdot \log N)$ on the average-case message complexity of any election algorithm for general networks. So the message complexity of the Gallager-Humblet-Spira algorithm is optimal.

Bibliographical notes

The Chang-Roberts algorithm originates from [21], and Franklin's algorithm comes from [38]. The Dolev-Klawe-Rodeh algorithm was proposed independently in [30] and in [73]. The lower bound of $\Omega(N \cdot \log N)$ on the average-case message complexity of election algorithms for rings was proved in [71]. The Gallager-Humblet-Spira algorithm stems from [40].

9.5 Exercises

Exercise 9.1 [92] Consider the Chang-Roberts algorithm on a directed ring of size N, under the assumption that every process is an initiator. For which distribution of IDs over the ring is the message complexity minimal, respectively maximal, and exactly how many messages are exchanged in these cases? Argue why no distribution of IDs gives rise to fewer or more messages.

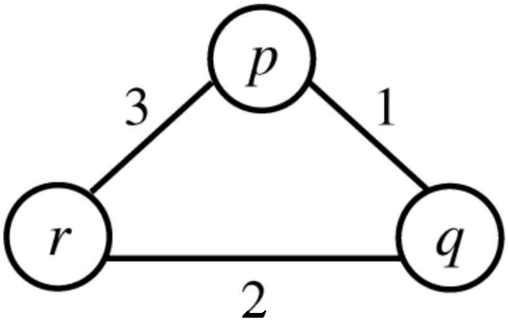
Exercise 9.2 [92] Give an initial configuration of a directed ring of size N, with every process an initiator, for which the Dolev-Klawe-Rodeh algorithm requires only two rounds. Also give an initial configuration for which the algorithm requires $\lfloor \log N \rfloor + 1$ rounds.

Exercise 9.3 Give a computation of the tree election algorithm on the network from example 9.6 in which eventually the processes 1 and 4 have each other as parent.

Exercise 9.4 Consider the tree election algorithm.

(a) [92] Show that for undirected networks with a diameter $D > 1$, the time complexity of this algorithm (including the wake-up phase) is at most $2{\cdot}D$.
(b) For $D = 2$, give an example where this algorithm takes 4 time units to terminate.
(c) Give an example to show that if $D = 1$, this algorithm may take 3 time units to terminate.

Exercise 9.5 Give one possible computation of the Gallager-Humblet-Spira algorithm on the following undirected network in order to determine the minimum spanning tree.



Exercise 9.6 Argue that in the Gallager-Humblet-Spira algorithm, any fragment at a level ℓ always contains at least 2^ℓ processes.

Exercise 9.7 Argue that the Gallager-Humblet-Spira algorithm correctly computes the minimum spanning tree.

Exercise 9.8 Suppose that, at some point in the Gallager-Humblet-Spira algorithm, a process reported a lowest-weight outgoing basic edge and next receives a message ⟨**initiate**, *fn*, ℓ, *find*⟩. Explain by means of a scenario why it must test again whether this basic edge is outgoing.

Exercise 9.9 Suppose that, at some point in the Gallager-Humblet-Spira algorithm, a process p receives a message ⟨**test**, fn, ℓ⟩ through channel pq, where p's fragment has a different name than fn and at least level ℓ. Explain why p can send an **accept** message to q without fear that p and q are in the same fragment.

Exercise 9.10 Consider the following scenario for the Gallager-Humblet-Spira algorithm. In a fragment F with name fn and level ℓ, the core nodes have just determined the lowest-weight outgoing edge of F. Concurrently, another fragment with name fn' and level $\ell' < \ell$ connects to F via a channel qp. Why can we be sure that F has an outgoing edge with a lower weight than pq?

Exercise 9.11 Suppose that the Gallager-Humblet-Spira algorithm is applied to a network in which different channels may have the same weight. (If a fragment has multiple lowest-weight outgoing edges, one of them is chosen to connect to a neighboring fragment, and the unique name of a fragment consists of the process IDs of its core nodes.) Give an example to show that a deadlock can occur. Also argue that the deadlock in your example is avoided if a total order is imposed on channels with the same weight.

10
Anonymous Networks

Sometimes the processes in a network are *anonymous*, meaning that they may lack a unique ID. Typically, this is the case if there are no unique hardware IDs (for example, LEGO Mindstorms). Furthermore, when each process does have a unique ID but cannot reveal it to the other processes, this is similar to having no unique process IDs at all. For instance, processes may not want to reveal their ID because of security concerns, or transmitting or storing IDs may be deemed too expensive, as is the case for the IEEE 1394 serial bus discussed in section 10.7.

In this chapter, it is assumed that processes (and channels) do not have unique IDs.

10.1 Impossibility of Election in Anonymous Rings

We show that no election algorithm for anonymous networks always terminates. The idea is that if the initial configuration is symmetric (typically, a ring in which all processes are in the same state and all channels are empty), then there is always an infinite execution that cannot escape this symmetry. Apparently, unique process IDs are a crucial ingredient for the election algorithms in chapter 9 in order to break symmetry.

Note that, vice versa, if one leader has been elected, all processes can be given a unique ID using a traversal algorithm (see section 4.1) initiated by the leader.

We recall from chapter 9 that election algorithms are decentralized. The initiators can be any nonempty set of processes, and all processes must have the same local algorithm.

Theorem 10.1 *No election algorithm for anonymous rings always terminates.*

Proof. Suppose we have an election algorithm for (directed or undirected) anonymous rings. We run it on an anonymous ring of size $N > 1$.

A configuration of the algorithm on the ring is *symmetric* if all processes are in the same state and all channels carry the same messages. We make three observations:

- There is a symmetric initial configuration: all processes can be provided with the same initial state, and initially all channels are empty.
- If γ_0 is a symmetric configuration and $\gamma_0 \rightarrow \gamma_1$, then there is a sequence of transitions $\gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_N$ where γ_N is symmetric. That is, the transition $\gamma_0 \rightarrow \gamma_1$ is caused by an internal, send, or receive event at some process. Since in γ_0 all processes are in the same state and all channels carry the same messages, all other processes in the ring can also perform this event, and the resulting configuration γ_N is symmetric.
- In a symmetric configuration, there is not one leader, because all processes are in the same state.

These observations together imply that the election algorithm exhibits an infinite execution, which infinitely often visits a symmetric configuration. □

We can even construct an infinite execution that is fair; that is, if an event can happen in infinitely many configurations in the execution, then this event is performed infinitely often during the execution. Given the symmetric configuration γ_0 in the third case just given, it does not matter which event available in γ_0 is used for the transition $\gamma_0 \rightarrow \gamma_1$; we can always build the transition sequence to the symmetric configuration γ_N. This implies that we can make sure no event is ignored infinitely often in the infinite execution.

10.2 Probabilistic Algorithms

In view of the impossibility result in theorem 10.1, we now turn to *probabilistic* algorithms, in which a process may, for example, flip a coin and perform an event based on the outcome of this coin flip. That is, events can happen with a certain probability.

For probabilistic algorithms, one can calculate the probability that an execution from some set of possible executions will occur. Although in the previous section we proved that election algorithms for anonymous rings inevitably contain infinite executions, in the next section we will see that there exists such an algorithm in which the probability that an infinite execution occurs is zero.

Probabilistic algorithms for which all executions terminate in a correct configuration are in general not so interesting, because any deterministic version of such an algorithm (for example, let the coin flip always yield heads) produces a correct nonprobabilistic algorithm. We therefore consider two classes of probabilistic algorithms: ones that always terminate but that may terminate incorrectly and ones that may not terminate but if they do the outcome is always correct.

A probabilistic algorithm is *Las Vegas* if

- the probability that it terminates is greater than zero and
- all terminal configurations are correct.

It is *Monte Carlo* if

- it always terminates and
- the probability that a terminal configuration is correct is greater than zero.

Note that if the probability that a Las Vegas algorithm terminates is one, there may still be infinite executions. But in this case the probability mass of all infinite executions together is zero. For example, consider an algorithm that flips a fair coin as long as the result is heads and that terminates as soon as the result of a coin flip is tails. The algorithm has one infinite execution, in which every time the outcome of the coin flip is heads. But the probability that this execution occurs is zero.

10.3 Itai-Rodeh Election Algorithm for Rings

The Itai-Rodeh election algorithm targets anonymous directed rings. Although in section 10.1 it was shown that there is no terminating election algorithm for this setting, the Itai-Rodeh election algorithm will achieve the next best thing: a Las Vegas algorithm that terminates with probability one.

We adapt the Chang-Roberts algorithm from section 9.1 to anonymous directed rings. Since processes do not have a unique ID, initiators randomly select an ID and send out this ID; only messages with the largest ID complete their round trip. The complication is that different initiators may select the same ID, in which case the election can be inconclusive. The Itai-Rodeh election algorithm therefore progresses in election rounds. If one active process in some round selects a larger ID than any other active process, then it becomes the leader at the end of this round. If, on the other hand, multiple active processes select the largest ID in some round, then there will be a next round, in which only active processes that selected the largest ID participate; all other processes become passive. At the start of every round, the active processes randomly select a new ID. Active processes keep track of their round number, so they can recognize and ignore messages from earlier rounds. We will see that this is essential because otherwise the algorithm could terminate in a configuration where all processes have become passive.

Since different processes may select the same ID, a process cannot readily recognize its own message when it completes the round trip. Therefore, messages carry a hop count, keeping track of how many processes have been visited. A message arrives at its source if and only if its hop count is N. Hence, it is required that processes know the ring size; in section 10.5, we will see that this requirement is crucial.

Now the Itai-Rodeh election algorithm is presented in more detail. Initially, at election round 0, initiators are active and noninitiators are passive. At the start of an election round $n \geq 0$, each active process p randomly selects an ID id_p from $\{1, \ldots, N\}$ and sends the message $(n, id_p, 1, \mathit{false})$ into its outgoing channel. The first value is the number of the election round in which this message evolved, the second value is the ID of its source, the third value is the hop count, and the fourth value is a Boolean that is set to *true* when an active process different from p with the ID id_p is encountered during the round trip.

Next, p waits for a message (n', i, h, b) to arrive. When this happens, p acts as follows, depending on the parameter values in this message:

- $n' > n$, or $n' = n$ and $i > id_p$:
  p has received a message from a future round, or from the current round with a larger ID. It becomes passive and sends $(n', i, h + 1, b)$.
- $n' < n$, or $n' = n$ and $i < id_p$:
  p has received a message from an earlier round, or from the current round with a smaller ID. It dismisses the message.
- $n' = n$, $i = id_p$, and $h < N$:
  p has received a message from the current round with its own ID but with a hop count smaller than N. Therefore, p is not the source of this message. It sends $(n, id_p, h + 1, \mathit{true})$ into its outgoing channel.
- $n' = n$, $i = id_p$, $h = N$, and $b = \mathit{true}$:
  p has received back its own message. Since the Boolean was set to *true* during the round trip, another active process selected the same ID as p in this round. Therefore, p proceeds to round $n + 1$.
- $n' = n$, $i = id_p$, $h = N$, and $b = \mathit{false}$:
  p has received back its own message. Since the Boolean is still *false*, all other active processes selected a smaller ID in this round or were still in an earlier round and have therefore become passive. Hence, p becomes the leader.

Passive processes simply pass on messages, increasing their hop count by 1.

The Itai-Rodeh election algorithm is a Las Vegas algorithm that terminates with probability one. There are infinite executions, in which active processes keep on selecting the same ID in every election round. However, in each election round with multiple active processes, with a positive probability not all active processes select the same ID. And, in that case, not all active processes will make it to the next round. Therefore, with probability one, eventually one process will become the leader.

The following example shows that without round numbers, the algorithm would be flawed.

Example 10.1 We consider one possible computation of the Itai-Rodeh election algorithm on an anonymous directed ring of size 3. The processes p, q, and r are all initiators and know that the ring size is 3.

- In round 0, p and q both select ID i, while r selects ID j, with $i > j$. The message sent by q makes r passive, its Boolean is set at p, and it returns to q. Likewise, the Boolean in the message sent by p is set at q, and it returns to p. Next, p and q move to the next round.
- In round 1, p and q select IDs k and ℓ, respectively, with $j > k > \ell$. The message sent by r in round 0 is slow, only now it reaches p. If round numbers were omitted, then p and subsequently q would not recognize that r's message is from the previous round, and they would become passive. Owing to round numbers, however, r's message is dismissed by p, and p and q continue to compete for the leadership. Since $k > \ell$, the message from q is dismissed at p, while the message from p makes q passive and returns to p with the Boolean still *false*. So p becomes the leader.

The computation in example 10.1 uses in an essential way that channels are non-FIFO. In case of FIFO channels, round numbers can be omitted.

It can be shown that if all processes are initiators, the average-case message complexity of the Itai-Rodeh election algorithm is $O(N{\cdot}\log N)$: on average, the N messages travel a logarithmic number of hops before they are dismissed, and the average number of election rounds is finite. In view of the lower bound of $\Omega(N{\cdot}\log N)$ on the average-case message complexity of election algorithms for rings mentioned in the previous chapter, one cannot hope to do better in a fully asynchronous setting.

Similar to the way the Chang-Roberts algorithm for election in directed rings was turned into a probabilistic version for anonymous directed rings, Franklin's algorithm for undirected rings can be turned into a probabilistic version for anonymous undirected rings. An advantage of the latter probabilistic algorithm, compared to the Itai-Rodeh algorithm, is that round numbers modulo 2, meaning only 0 and 1, suffice.

10.4 Echo Algorithm with Extinction for Anonymous Networks

The echo algorithm with extinction from section 9.3 can be adapted to anonymous undirected networks in a manner similar to the adaptation of the Chang-Roberts algorithm in the previous section. The resulting election algorithm progresses in rounds: at the start of every round, the active processes randomly select an ID and run the echo algorithm with extinction. Again, round numbers are used to recognize messages from earlier rounds. When a process is hit by a wave with a higher round number than its current wave or with the same round number but a higher ID, the process becomes passive (if it was not already) and moves to that other wave.

Each process again needs to know the network size to be able to determine at completion of its wave whether it has covered the entire network. The reason is that different waves with the same ID in the same round can collide with each other, after which these waves may both complete. These waves cover disjoint parts of the network. Therefore, when a process sends a wave message to its parent, it reports the size of its subgraph in this wave so that at the end the initiator of this wave can determine how many processes participated in the wave. If a wave completes but did not cover the entire network, then the initiator moves to the next election round. If a completed wave covered the entire network, then its initiator becomes the leader.

We now present the echo algorithm with extinction for election in anonymous undirected networks in more detail. Initially, at the start of round 0, initiators are active and noninitiators are passive.

Let process p be active. At the start of an election round $n \geq 0$, p randomly selects an ID $id_p \in \{1, ..., N\}$ and starts a wave, tagged with n and id_p. As third parameter, it adds a 0, meaning that this is not a message to its parent. This third parameter is used to report the subtree size in messages to a parent.

A process p, which is in a wave from round n with ID i, waits for a wave message to arrive, tagged with a round number n' and an ID j. When this happens, p acts as follows, depending on the parameter values in this message:

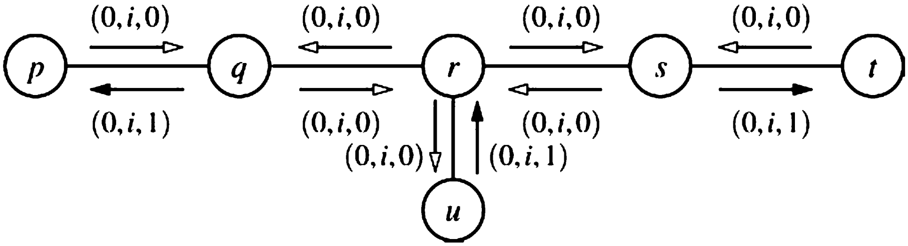
- If $n' > n$, or $n' = n$ and $j > i$, then p makes the sender its parent, changes to the wave in round n' with ID j, and treats the message accordingly.
- If $n' < n$, or $n' = n$ and $j < i$, then p dismisses the message.
- If $n' = n$ and $j = i$, then p treats the message according to the echo algorithm.

As noted, the echo algorithm is adapted by letting each message sent upward in the constructed tree report the size of its subtree; all other wave messages report 0. When a process decides, meaning that its wave completes, it computes the size of the constructed tree. If this equals the network size N, then the process becomes the leader. Otherwise it moves to the next election round, in which it randomly selects a new ID from $\{1, ..., N\}$ and initiates a new wave.

This election algorithm is a Las Vegas algorithm that terminates with probability one. There are infinite executions, in which active processes keep on selecting the same ID in every election round. However, with a positive probability, in election rounds with multiple active processes, not all active processes select the same ID. And, in that case, not all active processes will make it to the next round. Therefore, with probability one, eventually one process will become the leader.

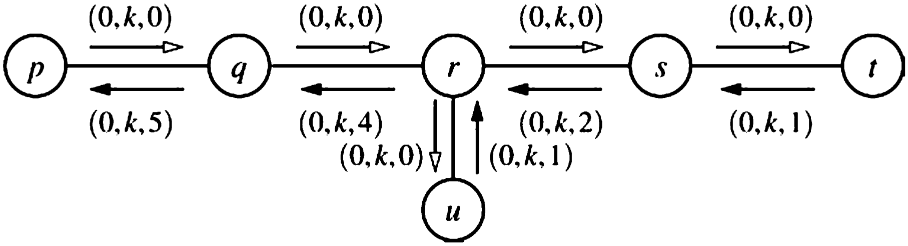
Example 10.2 We consider one possible computation of the echo algorithm with extinction on the anonymous undirected network pictured here. All processes know that the network size is 6.

- In round 0, processes p, r, and t select ID i, while processes q, s, and u select ID j, with $i > j$. In the picture, only the messages carrying ID i are shown; the messages carrying j are all dismissed at reception. Messages that are depicted with a solid arrowhead are toward a parent.



Processes p, r, and t each start their wave by sending (0, i, 0) to their neighbors. Next, q and s receive the messages from p and t, respectively, and send (0, i, 0) to r. When q and s receive (0, i, 0) from r, they interpret this as an answer to their message to r, reporting a subtree of size 0. So q and s report a subtree of size 1 to p and t, respectively. Moreover, u receives (0, i, 0) from r and reports a subtree of size 1 to r. Finally, p, r, and t each compute that their wave covered two processes and move to round 1.

- In round 1, p selects ID k, while r and t select ID ℓ, with $k > \ell$. In the next picture, only the messages carrying the ID k are shown.



The wave of p completes and reports that it covered six processes. So p becomes the leader.

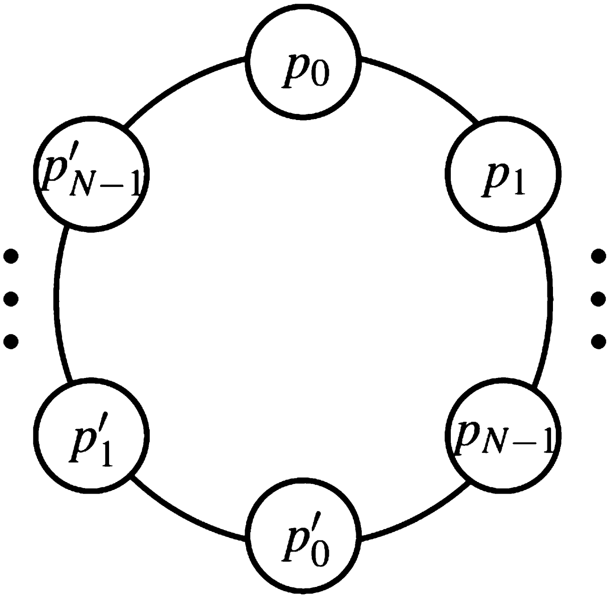
Note that in round 1 another scenario would be possible, in which the wave of t completes and t computes that its wave covered two processes, so that it moves to round 2. However, in the computation we consider, the wave of p in round 1 travels faster than the waves of r and t, so the wave of p completes.

10.5 Computing the Size of an Anonymous Ring Is Impossible

In the previous sections, we discussed Las Vegas algorithms for election in anonymous networks, which require that all processes in the network know the network size. We now prove that this assumption is essential. There is no Las Vegas algorithm to compute the size of anonymous rings; every probabilistic algorithm for computing the size of anonymous rings must allow for incorrect outcomes. This implies that there is no Las Vegas algorithm for election in anonymous rings if processes do not know the ring size, because when there is one leader, network size can be computed using a wave algorithm initiated by the leader.

Theorem 10.2 *There is no Las Vegas algorithm to compute the size of an anonymous ring.*

Proof. Suppose we have an algorithm for computing the size of a (directed or undirected) anonymous ring of size $N > 2$ with processes $p_0, \ldots, p_{N-1}$. Let C be a computation of the algorithm on this ring that terminates with the correct outcome N. We cut open this ring between p_0 and p_{N-1} and glue a copy $p_0', \ldots, p_{N-1}'$ of the ring in between. That is, we consider the following anonymous ring, of size $2{\cdot}N$.



Let the computation C' on this ring consist of replaying C twice: once on the half p_0,

..., p_{N-1} and once on the half p_0', ..., p_{N-1}'. In C', compared to C, p_0 communicates with p_{N-1}' instead of p_{N-1}, and p_{N-1} communicates with p_0' instead of p_0. But since p_0 and p_0' send the same messages in C', and likewise for p_{N-1} and p_{N-1}', and they do not have unique IDs to tell each other apart, none of these four processes can determine this difference. In C', the processes in the ring of size $2{\cdot}N$ terminate with the incorrect outcome N. □

10.6 Itai-Rodeh Ring Size Algorithm

The Itai-Rodeh ring size algorithm targets anonymous directed rings. In section 10.5, it was shown that it must be a Monte Carlo algorithm, meaning that it must allow for incorrect outcomes. However, in the Itai-Rodeh ring size algorithm, the probability of an erroneous outcome can be arbitrarily close to zero by letting the processes randomly select IDs from a sufficiently large domain.

Each process p maintains an estimate est_p of the ring size; initially $est_p = 2$. During any execution of the algorithm, est_p will never exceed the correct estimate N. The algorithm proceeds in estimate rounds. Every time a process finds that its estimate is too conservative, it moves to another round. That is, each process p initiates an estimate round at the start of the algorithm as well as at every update of est_p.

In each round, p randomly selects an ID id_p from $\{1, ..., R\}$ for some positive number R and sends the message $(est_p, id_p, 1)$ to its next neighbor. The third value is a hop count, which is increased by 1 every time the message is forwarded.

Now p waits for a message (est, id, h) to arrive. An invariant of such messages is that always $h \leq est$. When a message arrives, p acts as follows, depending on the parameter values in this message:

- $est < est_p$:
  The estimate of the message is more conservative than p's estimate, so p dismisses the message.
- $est > est_p$:
  The estimate of the message improves on p's estimate, so p increases its estimate. We distinguish between two cases:
  - $h < est$:
    The estimate est may be correct. So p sends $(est, id, h+1)$ to give the message the chance to complete its round trip. Moreover, p performs $est_p \leftarrow est$.
  - $h = est$:
    The estimate est is too conservative because the message traveled est hops but did not complete its round trip. So p performs $est_p \leftarrow est + 1$.
- $est = est_p$:
  The estimate of the message and that of p agree. We distinguish between two cases:
  - $h < est$:

p sends $(est, id, h + 1)$ to give the message the chance to complete its round trip.

- $h = est$:

  Once again, we distinguish between two cases:

  * $id \neq id_p$:

  The estimate *est* is too conservative because the message traveled *est* hops but did not complete its round trip. So p performs $est_p \leftarrow est + 1$.

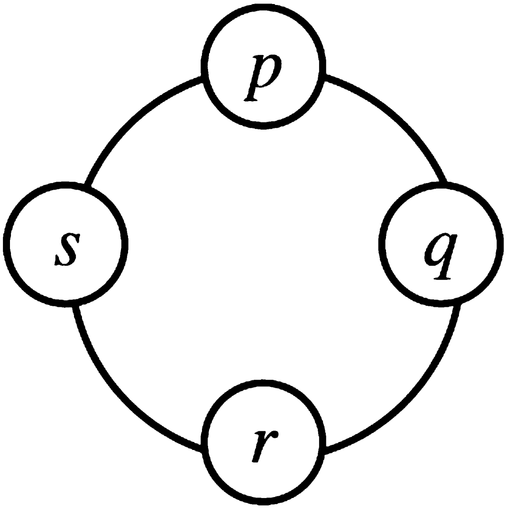
  * $id = id_p$:

  Possibly p's own message returned (or a message originating from another process *est* hops before p that unfortunately happened to select the same ID as p in this estimate round). In this case, p dismisses the message.

When the algorithm terminates, $est_p \leq N$ for all processes p, because a process increases its estimate only when it is certain that its current estimate is too conservative. Furthermore, est_p converges to the same value at all processes p. If this were not the case, clearly there would be processes p and q where p is q's predecessor in the ring and p's final estimate is larger than that of q. But then p's message in its final estimate round would have increased q's estimate to p's estimate.

The Itai-Rodeh ring size algorithm is a Monte Carlo algorithm: it may terminate with an estimate smaller than N. This can happen if in a round with an estimate $est < N$ all processes at distance *est* from each other happen to select the same ID.

Example 10.3 We consider one possible computation of the Itai-Rodeh ring size algorithm on the following anonymous ring, which is directed in a clockwise fashion.



In the initial estimate round with estimate 2, let p and r select ID i, while q and s

select ID j. Then p and r send the message $(2, i, 1)$, which is forwarded by q and s, respectively, as $(2, i, 2)$. Likewise, q and s send $(2, j, 1)$, which is forwarded by r and p, respectively, as $(2, j, 2)$. So each of the four processes receives a message $(2, k, 2)$ with k equal to its own ID. Hence, the algorithm terminates with the wrong estimate 2.

Example 10.4 We give another computation of the Itai-Rodeh ring size algorithm on the anonymous directed ring from example 10.3, which does converge to the correct estimate. In the initial estimate round, let p select ID i, while q and s select ID j and r selects ID $k \neq i$. Then p and r send messages $(2, i, 1)$ and $(2, k, 1)$, which are forwarded by q and s, as $(2, i, 2)$ and $(2, k, 2)$, respectively. Since $i \neq k$, these messages make p and r progress to the next estimate round, in which they both select ID j and send $(3, j, 1)$. These messages make q and s progress to the next estimate round because their estimate is larger than the estimates of q and s. Let q select ID j and s select ID $\ell \neq j$. In this estimate round, the messages $(3, j, 3)$ and $(3, \ell, 3)$ that originate from p and s, respectively, make s and r progress to the next estimate round. The messages from r and s in this last round make p and q progress to the next estimate round also. Finally, all processes terminate with the correct estimate, 4.

The probability that the algorithm terminates with an incorrect outcome becomes smaller when the domain $\{1, ..., R\}$ from which random IDs are drawn is made larger. This probability tends to zero when R tends to infinity for a fixed N (see exercise 10.11).

The worst-case message complexity is $O(N^3)$: each process starts at most $N - 1$ estimate rounds, and during each round it sends out one message, which takes at most N steps.

10.7 Election in IEEE 1394

The IEEE 1394 serial bus interface standard contains protocols for connecting devices in order to carry different forms of digital video and audio. Its architecture is scalable, and devices can be added or removed.

We concentrate on the election algorithm in IEEE 1394, which is employed when devices have been added to or removed from the network. Since it is deemed too expensive to store IDs of other processes, no IDs are attached to messages. This means that election is basically performed within an anonymous network. In view of the dynamic nature of the network, processes are not aware of the network size. We saw in the previous section that no Las Vegas algorithm exists for these types of networks, if cycles can be present. The topology of the undirected network is here assumed to be acyclic, and a variant of the tree algorithm from section 4.2 is employed. While in the tree election algorithm from section 9.2 the process with the largest ID becomes the leader, in IEEE 1394 the leader is selected from among the two processes that send a parent request to each other.

All processes in the network are initiators (so no wake-up phase is needed). When a process has one possible parent, it sends a parent request to this neighbor. If the request is accepted, an acknowledgment is sent back. The last two parentless processes may send parent requests to each other simultaneously; this is called root contention. A process p finds itself in root contention if it receives a parent request instead of an acknowledgment from a neighbor q in reply to its parent request. Then p randomly decides either to immediately send a parent request again or to wait some time for another parent request from q. In the latter case, if p does not receive another parent request from q within the waiting period, then p once again randomly decides either to immediately send a parent request or to wait some more time. Root contention is resolved when one of the processes waits while the other sends immediately. Then the process that is waiting becomes the leader.

Example 10.5 We run the IEEE 1394 election protocol on an acyclic undirected network of three processes p, q, and r, with channels pq and qr.

- p and r have only one possible parent, so they send a parent request to q.
- q receives the parent request of r and sends an acknowledgment back to r.
- q has only one possible parent left, so it sends a parent request to p. Now p and q are in root contention.
- p and q receive each other's parent requests. Now p (randomly) decides to wait some time, while q decides to send a parent request again immediately.
- p receives the parent request of q and sends back an acknowledgment. Thus, p has become the leader.

In practice, the election algorithm in IEEE 1394 is sometimes employed on networks that contain a cycle, which leads to a deadlock. For example, if the network is an undirected ring, no process ever sends a parent request, because every process has two possible parents. Therefore, the election algorithm contains a timeout mechanism, so networks containing a cycle lead to a timeout.

Bibliographical notes

The impossibility results regarding election in and computing the size of an anonymous ring date back to [4]. The Itai-Rodeh election and ring size algorithms originate from [48]. A variant of the Itai-Rodeh election algorithm without round numbers in the case of FIFO channels was proposed in [37], and a probabilistic version of Franklin's algorithm was proposed in [9]. The echo algorithm with extinction for anonymous networks stems from [91]. The IEEE 1394 standard was published in [47].

10.8 Exercises

Exercise 10.1 [92] Give a centralized algorithm for assigning unique IDs to processes that terminates after at most $D + 1$ time units.

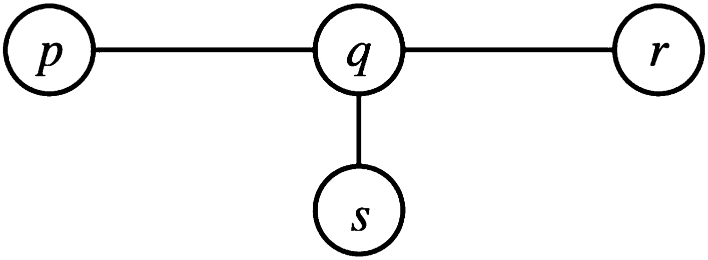
Exercise 10.2 Assume a Monte Carlo algorithm, and assume a (deterministic) algorithm to check whether the Monte Carlo algorithm terminated correctly. Give a Las Vegas algorithm that terminates with probability one.

Suppose the Monte Carlo algorithm gives a correct outcome with some probability π. How many applications of this algorithm does it take on average to come to a correct outcome?

Exercise 10.3 Apply the Itai-Rodeh election algorithm to an anonymous directed ring of size 3, in which all processes know the network size. Initially, let two processes select ID i, and let one process select ID j, with $i > j$. Give one possible computation.

Exercise 10.4 Give a probabilistic version of the Dolev-Klawe-Rodeh algorithm for election in anonymous directed rings of known size. Argue that your algorithm is a Las Vegas algorithm that terminates with probability one.

Exercise 10.5 Apply the echo algorithm with extinction to elect a leader in the anonymous undirected network pictured here. All processes are initiators and know the network size. In election round 0, let p and r select ID i, while q and s select ID j, with $i > j$. Give a computation in which s becomes the leader in round 1. Explain why, in such a computation, p and r will not both progress to round 1.



Exercise 10.6 Argue that there is no Las Vegas algorithm for election in anonymous rings of unknown size.

Exercise 10.7 Give a Monte Carlo algorithm for election in anonymous networks of unknown size. What is the success probability of your algorithm?

Exercise 10.8 Argue that there is no termination detection algorithm that always correctly detects termination on anonymous rings.

Exercise 10.9 Give an (always correctly terminating) algorithm for computing the size of anonymous acyclic networks.

Exercise 10.10 Apply the Itai-Rodeh ring size algorithm to an anonymous directed ring of size 3 in the following two cases:

(a) All three processes initially choose the same ID. Show that the algorithm computes ring size 2.
(b) Only two processes initially choose the same ID. Show that the algorithm computes ring size 3.

Exercise 10.11 Consider an anonymous ring for which the size N is an odd prime number and channels are FIFO. Determine the probability that the Itai-Rodeh ring size algorithm computes the correct ring size N under the assumption that processes never skip a round.

Exercise 10.12 In the case of root contention in the IEEE 1394 election algorithm, is it optimal for the average-case time complexity to give an equal chance of 50 percent to both sending immediately and waiting for some time?

11

Synchronous Networks

A *synchronous system* is a network in which the processes proceed in what is called lockstep. That is, a synchronous system proceeds in pulses and, in each pulse, each process

1. sends messages to its neighbors,
2. receives messages from its neighbors, and
3. performs internal events.

A message that is sent in a pulse must reach its destination before the receiver moves to the next pulse.

A *synchronizer* turns a network with asynchronous communication into a synchronous system. It must make sure that a process only moves to the next pulse when it has received all messages for its current pulse. This allows one to develop a distributed algorithm for synchronous systems, which in some cases is easier than developing it for a setting with fully asynchronous communication, and then use a synchronizer to make this algorithm applicable to general networks.

11.1 Awerbuch's Synchronizer

Awerbuch's synchronizer for undirected networks comprises three classes of synchronizers: α, β, and γ. The α synchronizer has a better time complexity, while the β synchronizer has a better message complexity. The γ synchronizer is a mix of the α and β synchronizers, combining the best of both worlds.

Let a basic algorithm run on the network. A process can become *safe* in a pulse when all basic messages it sent in this pulse have reached their destination. Basic messages are therefore acknowledged, and a process becomes safe in a pulse as soon as all basic messages it sent in this pulse have been acknowledged. A process can move to the next pulse when all its neighbors have become safe in its current pulse.

For simplicity, we take the liberty of ignoring acknowledgments of basic messages in the analysis of the message overhead of Awerbuch's synchronizer. This is not entirely unreasonable, since often acknowledgments of messages come for free if they are part of the underlying transport layer, as is the case in the Transmission Control Protocol.

α Synchronizer

In the α synchronizer, when a process has received acknowledgments for all basic messages it sent in a pulse, it sends a **safe** message to its neighbors. When a process p has received **safe** messages from all its neighbors in a pulse, it can move to the next pulse. Because then all basic messages sent by p's neighbors in this pulse have reached their destination, so in particular, p must have received all basic messages for this pulse.

In every pulse, the α synchronizer requires $2{\cdot}E$ **safe** messages. If the last process to start a pulse does so at time t, then this process is guaranteed to receive acknowledgments for its basic messages in this pulse no later than at time $t + 2$, so each neighbor will receive a **safe** message no later than at time $t + 3$. Hence, the time overhead is at most 3 time units per pulse.

β Synchronizer

The β synchronizer reduces the number of required **safe** messages. The key idea is to include an initialization phase, in which a centralized wave algorithm from chapter 4 is employed to build a spanning tree of the network. The **safe** messages travel up the tree to the root. When the root has received **safe** messages from all its children, all processes have become safe and can move to the next pulse. Then **next** messages travel down the tree to start this next pulse.

To be more precise, when a nonroot has in a pulse received acknowledgments for all basic messages it sent in this pulse as well as **safe** messages from all its children in the tree, it sends a **safe** message to its parent in the tree. When the root has in a pulse received acknowledgments for all basic messages it sent as well as **safe** messages from all its children, or when a nonroot receives a **next** message from its parent, it sends a **next** message to its children and moves to the next pulse.

In comparison to the α synchronizer, where $2{\cdot}E$ **safe** messages per pulse are sent, the β synchronizer uses $N - 1$ **safe** and $N - 1$ **next** messages, since they are sent only through tree edges. The time overhead of the β synchronizer, however, is more severe than that of the α synchronizer. If the last process to start a pulse does so at time t, then this process is guaranteed to receive acknowledgments for its basic messages in this pulse no later than at time $t + 2$. So if the spanning tree has depth k, the root will receive a **safe** message from its children no later than at time $t + k + 2$, and each nonroot will receive a **next** message from its parent no later than at time $t + 2 \cdot k + 2$. Hence, the time overhead is at most $2 \cdot k + 2$ time units per pulse.

γ *Synchronizer*

The γ synchronizer divides the network into clusters, and within each cluster a spanning tree is built. Between each pair of neighboring clusters, meaning distinct clusters that are connected by a channel, one of these connecting channels is selected as a designated channel. Each pulse consists of three phases. First, the β synchronizer is applied in each cluster to determine whether all processes in the cluster have become safe. Next, clusters signal to each other via the designated channels that they contain only safe processes, by means of the α synchronizer. Finally, within each cluster, the β synchronizer is used once more, to conclude that all neighboring clusters are safe so all processes in the cluster can move to the next pulse.

As noted, in each pulse, first the β synchronizer is applied within each cluster. Note that a process must receive acknowledgments for all basic messages it sent, including those to neighbors outside its cluster, before it can become safe. When the **next** messages of the β synchronizer travel down the tree within a cluster, the processes do not immediately move to the next pulse. Instead, they send **cluster-safe** messages into their designated channels. When a nonroot has received **cluster-safe** messages through all its designated channels as well as from all its children within its own cluster, it sends a **cluster-safe** message to its parent. When a root has received **cluster-safe** messages through all its designated channels as well as from all its children, or when a nonroot receives a **cluster-next** message from its parent, it sends a **cluster-next** message to its children and moves to the next pulse.

The message overhead of the γ synchronizer is

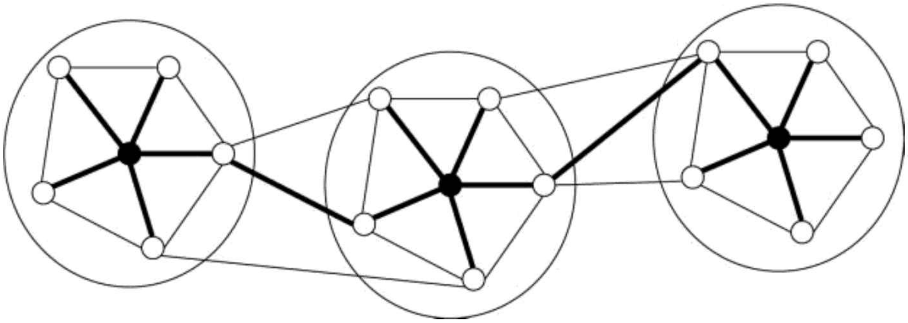
- a **safe**, a **next**, a **cluster-safe**, and a **cluster-next** message through each tree edge in any cluster, and
- two **cluster-safe** messages, one in each direction, through every designated channel.

The message overhead tends to increase with the number of designated channels; that is, with the number of neighboring clusters.

Let the spanning trees of the clusters have depth at most ℓ. If the last process to start a pulse does so at time t, then this process is guaranteed to receive acknowledgments for its basic messages in this pulse no later than time $t + 2$. So each root will receive a **safe** message from its children no later than time $t + \ell + 2$, and each nonroot will receive a **next** message from its parent no later than at time $t + 2\cdot\ell + 2$. Then each **cluster-safe** message through a designated channel will reach its destination no later than at time $t + 2\cdot\ell + 3$. So each root will receive a **cluster-safe** message from its children (and through its designated channels) no later than at time $t + 3\cdot\ell + 3$, and each nonroot will receive a **cluster-next** message from its parent no later than at time $t + 4\cdot\ell + 3$. Hence, the time overhead is at most $4\cdot\ell + 3$ time units per pulse.

In conclusion, on the one hand, we want few clusters to minimize the number of designated channels and thus reduce the message overhead. On the other hand, we want the trees in the clusters to have a small depth to minimize the time overhead.

Example 11.1 The network pictured here has been divided into three clusters. The dark processes depict the roots, while the dark lines depict the tree edges and designated channels. Since there are 15 tree edges in total and 2 designated channels, the message overhead of the γ synchronizer in a pulse consists of 60 messages through the tree edges plus 4 messages through the designated channels. Since the trees all have depth 1, we can take $\ell = 1$, so the time overhead of the γ synchronizer is at most 7 time units per pulse.



Note that because the network has 36 channels, the α synchronizer gives a message overhead of 72 **safe** messages per pulse. Furthermore, since each spanning tree of the entire network has depth at least 3, meaning $k \geq 3$, the β synchronizer may give a time overhead of 8 time units per pulse.

11.2 Bounded Delay Networks with Local Clocks

This section discusses a synchronizer for *bounded delay networks*, meaning that an upper bound $d_{\max}$ on network latency is known; when a message is sent, it is guaranteed to reach its destination within $d_{\max}$ time units.

Each process p is supposed to have a local clock C_p. A process can read as well as adjust the value of its local clock. The time domain is $\mathbb{R}_{\geq 0}$; that is, the nonnegative real numbers. We distinguish real time, meaning time progression in the real world, from local clock time, which tries to estimate real time. At real time τ, the clock at p returns the value $C_p(\tau)$. The local clocks are started at real time 0: for all processes p, $C_p(0) = 0$. Each local clock C_p is assumed to have *ρ-bounded drift*, for some $\rho > 1$, compared to real time. That is, if $\tau_2 \geq \tau_1$, then

$$\frac{1}{\rho}\cdot(\tau_2 - \tau_1) \;\leq\; C_p(\tau_2) - C_p(\tau_1) \;\leq\; \rho\cdot(\tau_2 - \tau_1).$$

To build a synchronous system, the local clocks should, moreover, have *precision δ* for some $\delta > 0$. That is, at any time τ and for any pair of processes p, q,

$$|C_p(\tau) - C_q(\tau)| \;\leq\; \delta.$$

Such precision can be achieved by a regular synchronization of all local clocks (see section 13.2).

Consider a bounded delay network, with local clocks that have ρ-bounded drift and precision δ; we assume that upper bounds for ρ and δ are known. The synchronizer is defined as follows: when a local clock C_p reaches the time

$$(i-1)\cdot(\rho^2\cdot\delta + \rho\cdot d_{\max})$$

for some $i \geq 1$, process p can start pulse i.

Theorem 11.1 *The synchronizer for bounded delay networks, with local clocks that have ρ-bounded drift and precision δ, is correct.*

Proof. We must show that each process is guaranteed to receive all its messages for a pulse $i \geq 1$ before starting pulse $i + 1$. It suffices to prove that for all processes p, q and all τ,

$$C_q^{-1}(\tau) + d_{\max} \;\leq\; C_p^{-1}(\tau + \rho^2\cdot\delta + \rho\cdot d_{\max})$$

(where $C_r^{-1}(\tau)$ is the first moment in real time at which the clock of process r returns the value τ). Because taking $\tau = (i-1)\cdot(\rho^2\cdot\delta + \rho d_{\max})$, this implies that q starts pulse i at least $d_{\max}$ time units before p starts pulse $i + 1$. Since network latency is assumed to be at most $d_{\max}$, this ensures that p receives the messages from q for pulse i in time.

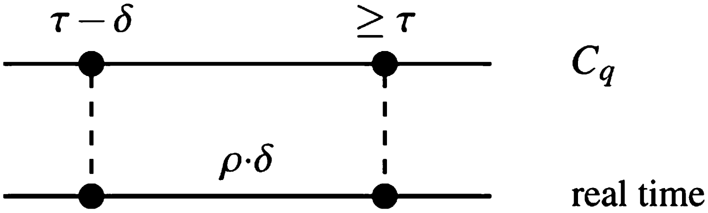
The desired inequality follows immediately from the following two inequalities:

$$C_q^{-1}(\tau) \le C_p^{-1}(\tau+\rho^2\cdot\delta), \tag{11.1}$$

$$C_p^{-1}(\tau)+\upsilon \le C_p^{-1}(\tau+\rho\cdot\upsilon). \tag{11.2}$$

(In the first inequality, add a summand $d_{\max}$ at both sides; in the second inequality, replace τ by $\tau + \rho^2\cdot\delta$ and υ by $d_{\max}$.) We now set out to prove these two inequalities.

Consider the moment in time when q's clock returns some value $\tau - \delta$, and let real time progress for $\rho\delta$ time units. Since q's clock is ρ-bounded from below, in that period q's clock will progress with at least δ time units, so it will return a clock value of at least τ. This can be depicted as follows.



In other words,

$$C_q^{-1}(\tau) \le C_q^{-1}(\tau-\delta)+\rho\cdot\delta.$$

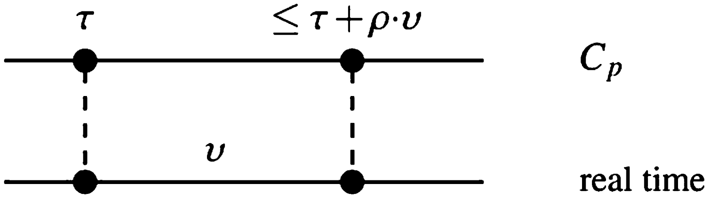
Furthermore, since local clocks have precision δ,

$$C_q^{-1}(\tau-\delta) \le C_p^{-1}(\tau).$$

Combining these two inequalities, we conclude that for all τ,

$$C_q^{-1}(\tau) \le C_p^{-1}(\tau)+\rho\cdot\delta. \tag{11.3}$$

Now consider the moment in time when p's clock returns some value τ, and let real time progress for υ time units. Since p's clock is ρ-bounded from above, in that period p's clock will progress with at most $\rho\upsilon$ time units, so it will return a clock value of at most $\tau + \rho\upsilon$. This can be depicted as follows.



In other words, we have argued that inequality (11.2) holds. Furthermore, inequalities (11.3) and (11.2), with $v = \rho\delta$, together yield inequality (11.1). □

11.3 Election in Anonymous Rings with Bounded Expected Delay

Bounded *expected* delay networks relax the restriction of bounded network latency to a known upper bound on the expected network latency. This means that arbitrarily long message delays are possible, but very long delays are less probable than short delays. In this setting, for anonymous directed rings in which all processes know the ring size N, a Las Vegas election algorithm exists with an average-case message and time complexity of $O(N)$. This shows that even mild synchrony restrictions can be helpful for the development of efficient distributed algorithms. We recall that without restrictions on the expected network latency, there is a lower bound $\Omega(N{\cdot}\log N)$ for the average-case message complexity of election algorithms on rings (see section 9.1).

The election algorithm for anonymous directed rings with bounded expected delay, which we refer to as the resuscitation algorithm, requires that processes have clocks with ρ-bounded drift for some $\rho > 1$. The algorithm starts, just as in the tree election algorithm, with a wake-up phase, in which each initiator, and each noninitiator when it is woken up, sends a wake-up message to its successor in the ring. After sending this message, the process becomes idle. Each (awake) process can be in four states: idle, active, passive, or leader. Initially, initiators are idle and noninitiators are passive. A passive process remains passive forever.

Each process p maintains a counter $np\text{-}hops_p$, which estimates from below the number of hops between p and its first nonpassive predecessor in the ring. Initially, $np\text{-}hops_p = 1$ at all processes p. An idle process p at each clock tick (i.e., at every moment that its clock progresses by one time unit) remains idle with some probability $\pi_p \in \langle 0, 1\rangle$ and becomes active with probability $1 - \pi_p$. When p becomes active, it sends a message ⟨**np-hops**, 1⟩, where the argument provides the receiver with a safe value for its *np-hops* variable.

Let a message ⟨**np-hops**, h⟩ arrive at a process q. Then q performs $np\text{-}hops_q \leftarrow \max\{np\text{-}hops_q, h\}$ and acts as follows, depending on its state:

- If q is idle or passive, it sends ⟨**np-hops**, $np\text{-}hops_q{+}1$⟩. And if q is idle, then it becomes passive.

- If q is active, either it becomes idle, if $h < N$, or it becomes the leader, if $h = N$.

Note that if a process sends a message ⟨**np-hops**, h⟩ with $h > 1$, then it is or becomes passive.

When a process q receives a message ⟨**np-hops**, h⟩ from its predecessor p, at least the first $h - 1$ predecessors of q in the ring are passive. This can be seen by induction on h, where the base case $h = 1$ is trivial. In the inductive case $h > 1$, p is passive and received a message ⟨**np-hops**, $h - 1$⟩ in the past, so by induction, the first $h - 2$ predecessors of p in the ring are also passive. This proves the claim. Since $np\text{-}hops_q$ stores the maximum h-value among all **np-hops** messages q has received, at least the first $np\text{-}hops_q - 1$ predecessors of q in the ring are passive.

We discuss the correctness of the resuscitation algorithm. The number of messages in transit always equals the number of active processes in the ring. This invariant can be argued as follows: initially both numbers are zero; idle processes that become active send a message; idle and passive processes forward messages and remain nonactive; and active processes remain active until a message arrives, at which point they become nonactive and do not forward the message. Since active processes never become passive upon the arrival of a message, there is always a nonpassive process. Furthermore, when an idle process becomes active, there is a positive probability that the message it sends will complete the round trip, after which the process becomes the leader. And we have argued earlier that when a process receives ⟨**np-hops**, N⟩, all other $N - 1$ processes must be passive. To conclude, the resuscitation algorithm is a Las Vegas algorithm that terminates with probability one.

Example 11.2 We consider one possible computation of the resuscitation algorithm on an anonymous directed ring with processes p, q, and r. At the first tick of their clocks all three processes happen to become active and send ⟨**np-hops**, 1⟩. Next, p and q receive these messages from r and p, respectively, and both become idle. At the next tick of its clock, p becomes active again and sends ⟨**np-hops**, 1⟩. This message arrives at q, which becomes passive and sends ⟨**np-hops**, 2⟩. This message arrives at r, which sets $np\text{-}hops_r$ to 2 and becomes idle. Finally, the message ⟨**np-hops**, 1⟩ that q sent at the start of the computation arrives at r. As a result, r becomes passive and sends ⟨**np-hops**, 3⟩ (because $np\text{-}hops_r = 2$). When this message arrives at p, it becomes the leader.

The computation in example 11.2 shows that it is beneficial for the message complexity that an idle or passive process r, upon the arrival of a message ⟨**np-hops**, h⟩, forwards ⟨**np-hops**, $np\text{-}hops_r + 1$⟩ instead of ⟨**np-hops**, $h + 1$⟩. Otherwise, at the end of the computation in the example, r would send not ⟨**np-hops**, 3⟩ but ⟨**np-hops**, 2⟩ to p. Then p would have become idle, instead of the leader.

Key to the favorable average-case message complexity of the resuscitation algorithm mentioned at the start of this section is a smart choice of the probability π_p with which an idle process p remains idle at a clock tick. This choice depends on the number of idle processes in the ring; the more idle processes, the larger π_p should be to maximize the chance that exactly one idle process becomes active and its message completes the round trip without any other idle process becoming active in the meantime. Therefore, the initial value of π_p depends on N, and π_p decreases at every increase of *np-hops*$_p$,

$$\pi_p = \left(\frac{N-1}{N+1}\right)^{\frac{np\text{-}hops_p}{N}} .$$

We argue that this dynamic value of π_p yields an average-case message and time complexity of $O(N)$. For simplicity, we assume that local clocks are perfectly synchronized with real time, that a message on average takes one time unit to reach its destination, and that the values *np-hops*$_p$ at nonpassive processes p always equal the number of hops between p and its first nonpassive predecessor. The forthcoming argumentation is somewhat more involved when clocks have ρ-bounded drift, or *np-hops*$_p$ may temporarily be smaller than the number of hops between p and its first nonpassive predecessor.

Consider a clock tick at which all nonpassive processes are idle. By assumption, the values of the parameters *np-hops*$_p$ of the nonpassive processes p always add up to N. Therefore, the probability that the idle processes all remain idle at this clock tick, meaning the product of the π_p for all idle processes p, is $\frac{N-1}{N+1}$. So the probability that an idle process becomes active and sends a message at this clock tick is $\frac{2}{N+1}$. By assumption, a round trip of this message on average takes N time units. Therefore, the probability that another idle process becomes active during this round trip is at most $1 - (\frac{N-1}{N+1})^N$, which is at most $\frac{8}{9}$ because $N \geq 2$, so this probability has an upper bound < 1 that is independent of the ring size. In conclusion, on average once every $O(N)$ time units an idle process becomes active, and with an average probability of $O(1)$ the message of this active process completes its round trip. These two observations together imply that the average-case message and time complexity of the resuscitation algorithm are $O(N)$.

Bibliographical notes

Awerbuch's synchronizer originates from [6]. Bounded delay networks were introduced in [23], and a synchronizer for bounded delay networks with local clocks that have bounded drift was presented in [92]. Expected bounded delay networks and the resuscitation algorithm stem from [8].

11.4 Exercises

Exercise 11.1 Consider the α synchronizer on a network of processes $p_0, \ldots, p_{N-1}$ on a line (so the network has diameter $N - 1$). Give an example in which at some point p_0 is in pulse 0 while p_{N-2} is in pulse $N - 2$.

Exercise 11.2 Argue that with the β synchronizer a process can never be two pulses ahead of another process in the network.

Exercise 11.3 Argue the correctness of the γ synchronizer.

Exercise 11.4 Suppose that performing internal events at the end of a pulse takes time and that an upper bound ε is known for this processing time. Explain how the synchronizer for bounded delay networks, with local clocks that have ρ-bounded drift and precision δ, needs to be adapted. Argue that your adapted synchronizer is correct.

Exercise 11.5 Explain how the resuscitation algorithm needs to be adapted if we want an initiator to become the leader. Also discuss how the wake-up phase of the resuscitation algorithm could then be used to obtain more accurate values for the *np-hops* variables at the processes.

12

Consensus with Crash Failures

In practice, processes in a distributed system may crash, meaning that they stop executing unexpectedly. The problem of failures can be alleviated by including redundancy and replication in the system and letting processes negotiate before a certain action is taken. A typical example is a database management system in which the processes collectively decide whether to commit or abort a distributed transaction (see section 16.2). In sections 3.3 and 6.5, crashed processes were already considered for the specific challenges of rollback recovery and termination detection. A basic assumption we made in those two sections and also in the current chapter is that the network topology is complete. This guarantees that the network topology always stays strongly connected, even when processes have crashed.

In this chapter, we consider how in general the rest of the system can cope with a crash of one or more processes, if at all possible. The next chapter will consider the more severe type of what is called a Byzantine failure, where a process may show behavior that is not in line with the specification of the distributed algorithm being executed. The challenge we pose to the system is to let its processes agree on a single value, even though some processes may have crashed. The *consensus* problem is that the correct processes, meaning the processes that have not crashed (or, in the next chapter, that are not Byzantine), must eventually uniformly decide for the same value. In a crash consensus algorithm to solve this problem, each process randomly chooses an initial value from some finite data domain. In all executions of a crash consensus algorithm, the following three properties must be satisfied:

- *Termination*: Every correct process eventually decides for a value.
- *Agreement*: All correct processes decide for the same value.
- *Validity*: If all processes choose the same initial value, then all correct processes decide for this value.

The validity requirement rules out trivial solutions where the processes always decide for the same value, for example, 0, irrespective of their initial values.

In the following sections, we will focus on the *binary* consensus problem, in which the input values of the processes are restricted to 0 and 1. So with N processes there are 2^N different initial configurations. By abuse of terminology, we will refer to this problem as consensus.

A reachable configuration of a crash consensus algorithm is called *bivalent* if from this configuration one can reach a terminal configuration where consensus has been reached on the value 0, as well as a terminal configuration where consensus has been reached on the value 1. The validity requirement implies that each crash consensus algorithm has a bivalent initial configuration; see exercise 12.1.

A k-crash consensus algorithm, for a $k > 0$, is a crash consensus algorithm that can cope with up to k crashing processes.

12.1 Impossibility of 1-Crash Consensus

In this section, we assume that processes cannot observe whether some process has crashed. If a process does not send messages for a very long time, this may simply be because the process or its outgoing channels is very slow. It turns out that in this setting one cannot hope to develop a terminating 1-crash consensus algorithm. No matter how large the network is, the prospect of one crashing process suffices to yield infinite executions in which no decision is ever made. The basic idea behind this impossibility result is that a decision for either 0 or 1, in an asynchronous setting, must be enforced by an event at a single process. But what if this process crashes immediately after this event, and in the case of a send event, the message travels through the channel for a very long, indefinite amount of time? Then, at some moment in time, the remaining processes should organize themselves to mimic the decision of the crashed process but without any input from this crashed process. This is shown to be impossible, if crashes cannot be observed.

Theorem 12.1 *There is no (always correctly terminating) algorithm for 1-crash consensus.*

Proof. Consider any 1-crash consensus algorithm. Let γ be a bivalent configuration. Then $\gamma \rightarrow \gamma_0$ and $\gamma \rightarrow \gamma_1$, where γ_0 can lead to decision 0 and γ_1 to decision 1. We argue that γ_0 or γ_1 must be bivalent. We distinguish between two cases:

- The transitions $\gamma \to \gamma_0$ and $\gamma \to \gamma_1$ correspond to events a_0 and a_1 at different processes p_0 and p_1, respectively. Then in γ_0 the event a_1 at p_1 can still be performed, and likewise in γ_1 the event a_0 at p_0 can still be performed. Performing a_1 at p_1 in γ_0 and performing a_0 at p_0 in γ_1 lead to the same configuration δ, because in both cases the resulting configuration is reached from γ by performing the concurrent events a_0 and a_1 at p_0 and p_1. So there are transitions $\gamma_0 \to \delta$ and $\gamma_1 \to \delta$. Hence, γ_0 or γ_1 is bivalent, if δ can lead to a decision 1 or 0, respectively.
- The transitions $\gamma \to \gamma_0$ and $\gamma \to \gamma_1$ correspond to events at the same process p. In γ, p can crash. Moreover, a message sent by p can take indefinitely long to reach its destination. Therefore, in γ the processes other than p must be 0-*potent* or 1-*potent*. In a configuration, a set S of processes is said to be b-potent, for some $b \in \{0, 1\}$, if by executing only events at processes in S, some process in S can decide for b. Since $\gamma \to \gamma_0$ and $\gamma \to \gamma_1$ concern p, in γ_0 and γ_1 the processes other than p are still b-potent. So $\gamma_{1\text{-}b}$ is bivalent.

In conclusion, a bivalent configuration can always make a transition to some bivalent configuration. Since the crash consensus algorithm has a bivalent initial configuration (see exercise 12.1), it follows that there is an infinite execution, visiting only bivalent configurations. □

We can even construct an infinite execution that is fair; see exercise 12.3.

12.2 Bracha-Toueg Crash Consensus Algorithm

In chapter 10, to circumvent the impossibility of a terminating election algorithm for anonymous networks, we moved to probabilistic algorithms. Here we follow the same approach. First, we discuss another impossibility result: there is no Las Vegas algorithm for crash consensus if half of the processes can crash. Then the network can be divided into two disjoint halves, where in each half the processes do not receive messages from the processes in the other half for a very long time, so that eventually they must act under the assumption that the processes in the other half have crashed. As a result, the two halves may act as independent entities that can come to different decisions.

Theorem 12.2 *Let $k \geq \frac{N}{2}$. There is no Las Vegas algorithm for k-crash consensus.*

Proof. Suppose, toward a contradiction, that such an algorithm does exist. Divide the processes into two disjoint sets S and T, which both contain at most $\lceil \frac{N}{2} \rceil$ processes.

Since $k \geq \lceil \frac{N}{2} \rceil$, the processes in S must be able to come to a decision by themselves if all processes in T have crashed. In other words, S is always 0-potent or 1-potent. Likewise, T is always 0-potent or 1-potent. In each reachable configuration, S and T

should either be both 0-potent or both 1-potent. Otherwise they could independently decide for different values, because they are disjoint sets.

Consider a bivalent initial configuration γ. There must be a configuration δ reachable from γ, and a transition $\delta \rightarrow \delta'$, with S and T both only b-potent in δ and only $(1 - b)$-potent in δ' , for some $b \in \{0, 1\}$. Since this transition would correspond to a single event, at a process in either S or T, clearly such a transition cannot exist. □

If $k < \frac{N}{2}$, then a Las Vegas algorithm for k-crash consensus does exist. The Bracha-Toueg k-crash consensus algorithm, for $k < \frac{N}{2}$, progresses in rounds. Initially, at the start of round 0, each process randomly chooses a value 0 or 1. The weight of a process, holding a value $b \in \{0, 1\}$, approximates from below the number of processes that voted b in the previous round. In round 0, each process has weight 1, because it only knows its own vote.

In each round $n \geq 0$, each correct, undecided process p sends its value and weight to all processes and determines a new value and weight, based on the first $N - k$ messages it receives in this round.

- p sends $\langle n, value_p, weight_p\rangle$ to all processes (including itself).
- p waits until $N - k$ messages $\langle n, b, w\rangle$ have arrived. (It dismisses or stores messages from earlier or future rounds, respectively.) With regard to $value_p$, we distinguish between two cases.
  - If $w > \frac{N}{2}$ for an incoming message $\langle n, b, w\rangle$, then $value_p \leftarrow b$.
  - Otherwise, $value_p \leftarrow 0$ if most messages voted 0, or else $value_p \leftarrow 1$.

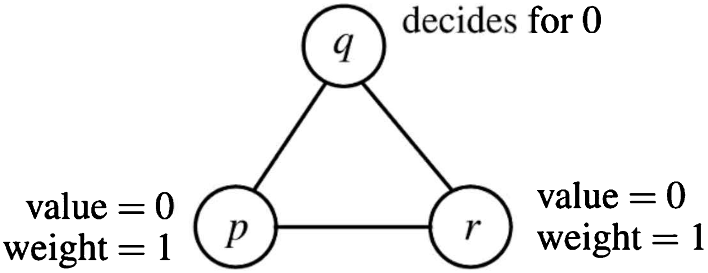
  $weight_p$ is changed into the number of incoming votes for $value_p$ in round n.
- If $w > \frac{N}{2}$ for more than k incoming messages $\langle n, b, w\rangle$, then p decides for b.

If p decides for b in round n, it broadcasts $\langle n + 1, b, N - k\rangle$ and $\langle n + 2, b, N - k\rangle$ and terminates. This suffices because when a process decides, all other correct processes are guaranteed to decide within two rounds; see the proof of theorem 12.3.

In reality, p of course does not send messages to itself but simply includes the message it broadcasts in the collection of messages it receives from other processes in the same round. Note that if p waited for more than $N - k$ messages $\langle n, b, w\rangle$ in a round n, then a deadlock could occur, if k processes have crashed. Note also that if p receives messages $\langle n, b, w\rangle$ and $\langle n, 1 - b, w'\rangle$, then $w + w' \leq N$, so w and w' cannot both be greater than $\frac{N}{2}$. Note, finally, that since p waits for $N - k$ messages $\langle n, b, w\rangle$ and can decide only if more than k of those messages have a weight greater than $\frac{N}{2}$, it is essential for termination that $N - k > k$.

Example 12.1 Consider a network of three processes p, q, and r, with $k = 1$. During each round a process requires two incoming messages to determine a new value and weight, and two b-votes with weight 2 to decide for b. We study one possible computation of the Bracha-Toueg 1-crash consensus algorithm on this network.

- Initially, p and q randomly choose the value 0 and r the value 1, with weight 1.
- In round 0, p takes into account the messages from p and r; it sets its value to 1 and its weight to 1. Moreover, q and r both take into account the messages from p and q; each sets its value to 0 and its weight to 2.
- In round 1, q takes into account the messages from q and r; since both messages carry weight 2, it decides for 0. Moreover, p and r both take into account the messages from p and r; since the message from r carries weight 2, each sets its value to 0 and its weight to 1.



- At the start of round 2, q crashes. So p and r can take into account only the messages from p and r; as a result, each sets its value to 0 and its weight to 2.
- In round 3, p and r can again only take into account the messages from p and r; since both messages carry weight 2, they decide for 0.
- p and r send messages with value 0 and weight 2 for two more rounds and terminate.

Theorem 12.3 *If the scheduling of messages is fair, then the Bracha-Toueg k-crash consensus algorithm, for each* $k < \frac{N}{2}$, *is a Las Vegas algorithm that terminates with probability one.*

Proof. First, we prove that processes cannot decide for different values. Then we prove that the algorithm terminates with probability one, owing to the assumption that the scheduling of messages is fair, meaning that each possible order of delivery of the messages in a round occurs with a positive probability.

Suppose a process p decides for a value b at the end of a round n. Then, at the start of round n, $value_q = b$ and $weight_q > \frac{N}{2}$ for more than k processes q. Since in every round each correct, undecided process ignores messages from only k processes, in round n these processes all take into account a message $\langle q, b, w\rangle$ with $w > \frac{N}{2}$. So in round $n + 1$, all correct processes vote b. So in round $n + 2$, all correct processes vote b

with weight $N - k > \frac{N}{2}$. Hence, after round $n + 2$, all correct processes have decided for b. To conclude, all correct processes decide for the same value.

Because of fair scheduling, in each round there is a positive probability that all processes receive the first $N - k$ messages from the same $N - k$ processes. After such a round n, all correct processes have the same value b. Then, after round $n + 1$, all correct processes have the value b with weight $N - k$. And after round $n + 2$, all correct processes have decided for b. In conclusion, the algorithm terminates with probability one. □

12.3 Failure Detectors

The impossibility result in theorem 12.1, that there is no terminating 1-crash consensus algorithm, assumes that crashes of processes cannot be observed. In the next section, we will see that this is a crucial assumption: when crashes can be detected, there does exist a terminating k-crash consensus algorithm, if $k < \frac{N}{2}$. In the current section, the notion of failure detectors, their properties, and two straightforward implementations with different properties are discussed. We recall that failure detectors already played a crucial role in the rollback recovery and termination detection algorithms in sections 3.3 and 6.5.

As time domain we take $\mathbb{R}_{\geq 0}$. Each execution of a crash consensus algorithm is provided with a failure pattern consisting of sets $F(\tau)$ that contain the crashed processes at time τ. Crashed processes cannot restart: $\tau_0 \leq \tau_1 \Rightarrow F(\tau_0) \subseteq F(\tau_1)$. It is assumed that processes cannot observe $F(\tau)$.

Processes carry a failure detector to try to detect crashed processes. By $H(p, \tau)$ we denote the set of processes that are suspected to have crashed by (the failure detector of) process p at time τ. Each execution of a crash consensus algorithm is provided with a failure detector history H. In general, such suspicions may turn out to be false, typically because at a time τ a process p receives a message from a suspected process in $H(p, \tau)$. However, we require that failure detectors be always *complete*: from some time onward, every crashed process is suspected by every correct process.

A failure detector is called *strongly accurate* if only crashed processes are ever suspected. In bounded delay networks (see section 11.2), a strongly accurate (and complete) failure detector can be implemented as follows. Suppose $d_{\max}$ is a known upper bound on network latency. Let each process broadcast a message **alive** every ν time units. A process from which no message has been received for $\nu + d_{\max}$ time units has crashed. Since crashed processes stop sending **alive** messages, the failure detector is clearly complete. And the bound on network latency ensures that there can never be more than $\nu + d_{\max}$ time units between the arrival at p of subsequent **alive** messages from q, so the failure detector is strongly accurate.

A failure detector is called *eventually strongly accurate* if from some point in time onward, only crashed processes are suspected. In other words, there can be false suspicions, but only up to a certain moment in time. Suppose there is an unknown upper bound on network delay. Again, let each process broadcast a message **alive** every v time units. An eventually strongly accurate (and complete) failure detector can be implemented as follows. Each correct process p initially guesses as network latency $d_p = 1$. If p does not receive a message from a process q for $v + d_p$ time units, then p suspects that q has crashed. This suspicion may be false if the value of d_p is too conservative. When p receives a message from a suspected process q, then q is no longer suspected by p and $d_p \leftarrow d_p + 1$. Since crashed processes stop sending **alive** messages, the failure detector is clearly complete. And since network latency is bounded, each process p can receive a message from a suspected process and as a result increase d_p only finitely many times. This guarantees that the failure detector is eventually strongly accurate.

12.4 Consensus with a Weakly Accurate Failure Detector

A failure detector is called *weakly accurate* if some correct process is never suspected by the other processes. In the presence of a weakly accurate failure detector, there is a simple k-crash consensus algorithm that works for each k.

Let the processes be numbered: $p_0, \ldots, p_{N-1}$. Initially, each process randomly chooses a value 0 or 1. The crash consensus algorithm proceeds in rounds $n = 0, \ldots, N - 1$. In a round n, the process p_n acts as the coordinator.

- p_n (if not crashed) broadcasts its value.
- Each process waits
  - either for a message from p_n, in which case it adopts the value of p_n,
  - or until it suspects that p_n has crashed.

  Next, if $n < N - 1$, the process moves to the next round.

After round $N - 1$, each correct process decides for its value at that time.

This rotating coordinator algorithm is a k-crash consensus algorithm for each $k < N$. Since the failure detector is weakly accurate, some correct process p_i is never suspected by the other processes. This implies that after round i, all correct processes have the same value b. Then clearly in the rounds $i + 1, \ldots, N - 1$, all processes keep this value b. Hence, after round $N - 1$, all correct processes decide for b.

12.5 Chandra-Toueg Algorithm

With an eventually strongly accurate failure detector, the proof of theorem 12.2, that there is no Las Vegas algorithm for k-crash consensus if $k \geq \frac{N}{2}$, still applies. Because it may take a very long, indefinite period of time before the failure detector becomes strongly accurate, and up to that point there can still be false suspicions.

Theorem 12.4 *Let $k \geq \frac{N}{2}$. There is no Las Vegas algorithm for k-crash consensus based on an eventually strongly accurate failure detector.*

Proof. Suppose, toward a contradiction, that such an algorithm does exist. Divide the processes into two disjoint sets S and T, which both contain at most $\lceil \frac{N}{2} \rceil$ processes.

Since $k \geq \lceil \frac{N}{2} \rceil$, the processes in S are able to come to a decision by themselves, as they must be able to cope if all processes in T have crashed. In other words, S is always 0-potent or 1-potent. Likewise, T is always 0-potent or 1-potent. In each reachable configuration, S and T should either be both 0-potent or both 1-potent, for otherwise they could independently decide for different values. *Because, since the failure detector is only eventually strongly accurate, the processes in S (or T) may falsely suspect for a very long period of time that the processes in T (respectively S) have crashed and at some point need to come to a decision by themselves.*

Consider a bivalent initial configuration γ. There must be a configuration δ reachable from γ, and a transition $\delta \rightarrow \delta'$, with S and T both only b-potent in δ and only (1 -b)-potent in δ' , for some $b \in \{0, 1\}$. Since this transition would correspond to a single event, at a process in either S or T, clearly such a transition cannot exist. □

A failure detector is called *eventually weakly accurate* if from some point in time on, some correct process is never suspected by the other processes. The Chandra-Toueg crash consensus algorithm, which uses an eventually weakly accurate failure detector, is an always correctly terminating k-crash consensus algorithm if $k < \frac{N}{2}$.

Let the processes be numbered: $p_0, \ldots, p_{N-1}$. Initially, each process randomly chooses a value 0 or 1. The algorithm proceeds in rounds. Each process q records the number of the last round *last-update*$_q$ in which it updated its value; initially *last-update*$_q$ = −1.

Each round $n \geq 0$ is coordinated by the process p_c with $c = n \bmod N$. Round n progresses as follows.

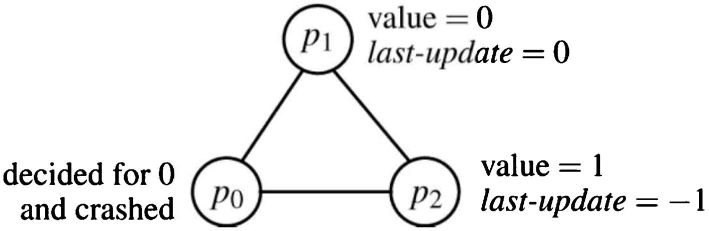
- Every correct, undecided process q (including p_c) sends to p_c the message ⟨**vote**, n, *value*$_q$, *last-update*$_q$⟩.
- p_c (unless crashed or decided) waits until $N - k$ such messages have arrived, selects one, say ⟨**vote**, n, b, ℓ⟩, with ℓ as large as possible, and broadcasts ⟨**value**, n, b⟩.
- Every correct, undecided process q (including p_c) waits until either
  - * ⟨**value**, n, b⟩ arrives; then it performs *value*$_q \leftarrow b$ and *last-update*$_q \leftarrow n$, and sends ⟨**ack**, n⟩ to p_c,
  - * or it suspects that p_c has crashed, at which time it sends ⟨**nack**, n⟩ to p_c.
- p_c waits for N - k acknowledgments. If more than k of them are positive, then p_c decides for b, broadcasts ⟨**decide**, b⟩, and terminates.

A correct, undecided process that receives ⟨**decide**, b⟩ decides for b and terminates.

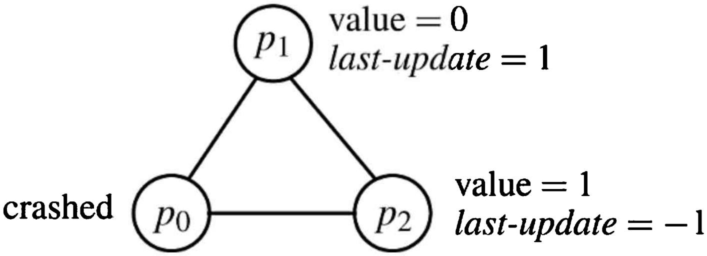
The idea behind the decision criterion for the coordinator in a round n, the receipt of more than k positive acknowledgments, is that then more than k processes have adopted the value b of the coordinator and set their *last-update* parameter to n. Since in each round the coordinator ignores messages of only k processes and adopts the value from a message with a maximal *last-update*, this guarantees that in future rounds coordinators will always adopt the value b; see the proof of theorem 12.5.

Example 12.2 Consider a complete network of three processes p_0, p_1, and p_2, with $k = 1$. During each round the coordinator waits for two incoming votes and needs two positive acknowledgments to decide. We study one possible computation of the Chandra-Toueg 1-crash consensus algorithm on this network.

- Initially, p_0 and p_2 randomly choose the value 1 and p_1 chooses the value 0; *last-update* = −1 at all three processes.
- In round 0, the coordinator p_0 takes into account the messages from p_0 and p_1, selects the message from p_1 to determine its new value, and broadcasts the value 0. When p_0 and p_1 receive this message, each sets its value to 0 and *last-update* to 0 and sends **ack** to p_0; moreover, p_1 moves to round 1. However, p_2 moves to round 1 without waiting for a message from p_0, because its failure detector falsely suspects that p_0 has crashed; p_2 sends **nack** to p_0. The coordinator p_0 receives the **ack**s of p_0 and p_1, decides for 0, and crashes before it can broadcast a **decide** message.



- In round 1, the coordinator p_1 can take into account only the messages from p_1 and p_2. It must select the message from p_1 to determine its new value, because it has the highest *last-update*. So p_1 broadcasts the value 0. When p_1 receives this message, it sets its value to 0 and *last-update* to 1 and sends **ack** to itself. The process p_2 moves to round 2 without waiting for a message from p_1, because its failure detector falsely suspects that p_1 has crashed; p_2 sends **nack** to p_1. After p_1 has received the **ack** and **nack** from p_1 and p_2, respectively, it also moves to round 2.



- In round 2, the coordinator p_2 can take into account only the messages from p_1 and p_2. It must select the message from p_1 to determine its new value, because it has the highest *last-update*. So p_2 broadcasts the value 0. When p_1 and p_2 receive this message, each sets its value to 0 and *last-update* to 2 and sends **ack** to p_2; moreover, p_1 moves to round 3. The coordinator p_2 receives the **ack**s of p_1 and p_2, decides for 0, and broadcasts ⟨**decide**, 0⟩. When p_1 receives this message, it also decides for 0.

Theorem 12.5 *In the presence of an eventually weakly accurate failure detector, the Chandra-Toueg algorithm is an (always correctly terminating) k-crash consensus algorithm for each* $k < \frac{N}{2}$.

Proof. First, we prove that processes cannot decide for different values. Next, we prove that the algorithm always terminates.

Let the coordinator in a round n decide for a value, say b. Then the coordinator received more than k **acks** in this round, so at the end of this round

(1) there are more than k processes q with $\mathit{last\text{-}update}_q \geq n$, and
(2) $\mathit{last\text{-}update}_q \geq n$ implies $\mathit{value}_q = b$.

We argue, by induction on $m - n$, that properties (1) and (2) are preserved in all rounds $m \geq n$. The base case $m = n$ was already argued. For the inductive case, consider a round $m > n$. By induction, (1) and (2) hold at the start of this round. Since the coordinator ignores the votes of only k processes, by (1) it takes into account at least one vote with *last-update* $\geq n$ to determine its new value. Hence, by (2), the coordinator sets its value to b and broadcasts ⟨**value**, m, b⟩. This means that (1) and (2) still hold at the end of round m, which concludes the proof of the claim. It implies that from round n onward, processes can decide only for b.

Now we argue that eventually some correct process will decide. Since the failure detector is eventually weakly accurate, from some round onward, some correct process p will never be suspected. So in the next round where p is the coordinator, all correct processes will wait for a **value** message from p. Therefore, p will receive at least $N - k$ **acks**, and since $N - k > k$, it will decide. All correct processes will eventually receive the **decide** message of p and will also decide. □

12.6 Consensus for Shared Memory

In this last section on consensus in the presence of crashes, we turn to shared memory (see section 2.3). In that setting a wait-free consensus algorithm, meaning an algorithm in which all noncrashed processes are guaranteed to reach consensus, is achieved in a straightforward fashion using the *compare-and-set* operation. Instead of binary consensus, we now consider the general consensus problem that all processes should uniformly decide for a value from some data domain D. Initially, each process p is provided with a random value $value_p$ from D. Let $\perp$ denote a special value that is not in D. Some register that is multi-reader and multi-writer initially carries the value $\perp$. Each process p performs *compare-and-set*($\perp$, $value_p$) on this register. The idea is that all processes decide for the value of the first process to perform this operation. If p gets *true* returned, then it can decide for its own value, since it is the first to perform this read-modify-write operation on the register. If it gets *false* returned, some other process was the first to perform this operation. Then p reads the value in the register and decides for that value. This is a correct consensus algorithm because the first *compare-and-set*($\perp$, v) operation that is performed on the register reads the value $\perp$ and overwrites it with v in one atomic step. All subsequent *compare-and-set* operations on the register will therefore fail.

The use of a read-modify-write operation is essential here because with atomic read and write operations it may be impossible to determine the value of the process that performs its write operation first. A consensus algorithm in which reading $\perp$ and writing $value_p$ are distinct atomic steps would be flawed, because multiple processes could concurrently read the value $\perp$ and write their value in the register.

Remarkably, other standard read-modify-write operations, such as *test-and-set* and *get-and-increment*, do not allow for a wait-free consensus algorithm, if there are three or more processes. The problem is that these operations are either overwriting, meaning that they write a predetermined new value without taking into account the old value of the register, or commuting, meaning that the two orders in which two processes can apply this operation are guaranteed to produce the same value in the register. We now argue this impossibility result in more detail, distinguishing between the two types of read-modify-write operations: overwriting and commuting. For simplicity, we focus on binary consensus.

We say that a configuration is *critical* if it is bivalent and each of its transitions results in a configuration that is not bivalent. In other words, the configuration is still undecided, but each event enforces a decision. Each consensus algorithm contains such a critical configuration because by validity we can start in a bivalent initial configuration and by termination we must after a finite number of transitions reach a configuration that is not bivalent (cf. the proof of theorem 12.1).

Suppose, toward a contradiction, that a binary consensus algorithm exists that is based on *test-and-set* or *get-and-increment*. Consider a critical configuration of the algorithm. Let p, q, and r be different processes, where a read-modify-write operation by p or q enforces the decision 0 or 1, respectively. If these critical operations are performed on different registers, and p and q crash immediately after having performed their operations, then r cannot figure out whether p or q performed its operation first so r does not know whether to decide for 0 or 1. Hence, these operations must be performed on the same register. We distinguish between the two types of read-modify-write operations.

- First, consider *test-and-set*, which is overwriting in nature. As a result, even if these operations by p and q are performed on the same register, r still may not be able to figure out who performed this operation first if p and q crash. Because r cannot discriminate between the situation where first p and then q perform the operation and the situation where only q performs the operation.
- Now consider *get-and-increment*, which is commuting in nature. As a result, again r still may not be able to figure out who performed this operation on a register first if p and q crash. Because r cannot discriminate between the situation where first p and then q perform the operation and the situation where first q and then p perform the operation.

Bibliographical notes

The impossibility of a terminating 1-crash consensus algorithm was proved in [35]. The Bracha-Toueg crash consensus algorithm originates from [13], and the Chandra-Toueg crash consensus algorithm comes from [17]. The strong consensus property of *compare-and-set*, in contrast to overwriting and commuting read-modify-write operations, was observed in [44].

12.7 Exercises

Exercise 12.1 Prove that, if $N > 1$, every 1-crash consensus algorithm has a bivalent initial configuration.

Exercise 12.2 [92] Give terminating 1-crash consensus algorithms for each of the following restrictions on the chosen values in initial configurations.

(a) An even number of processes choose the value 1.
(b) At least $\lceil \frac{N}{2} + 1 \rceil$ processes choose the same value.

Exercise 12.3 Argue that each algorithm for 1-crash consensus exhibits a *fair* infinite execution.

Exercise 12.4 Give a Monte Carlo algorithm for k-crash consensus that works for each $k < N$.

Exercise 12.5 Consider a complete network of five processes. Apply the Bracha-Toueg 2-crash consensus algorithm, where initially three processes choose the value 0 and two processes choose the value 1. Give two computations, one where all correct processes decide for 0 and one where all correct processes decide for 1.

Exercise 12.6 [92] Consider the Bracha-Toueg k-crash consensus algorithm for $k < \frac{N}{2}$.

(a) Let more than $\frac{N+k}{2}$ processes choose the value b in the initial configuration. Prove that then the correct processes always decide for b within three rounds.

(b) Let more than $\frac{N-k}{2}$ processes choose the value b in the initial configuration. Give a computation in which all correct processes decide for b within three rounds.

(c) Let N -k be even. Is a decision for b possible if exactly $\frac{N-k}{2}$ processes choose the value b in the initial configuration?

Exercise 12.7 Give an example to show that if the scheduling of messages is not fair, then the Bracha-Toueg crash consensus algorithm may not terminate with probability one.

Exercise 12.8 Give an example, with $N = 3$ and $k = 1$, to show that in the Bracha-Toueg k-crash consensus algorithm, k incoming messages with a weight greater than $\frac{N}{2}$ are not sufficient to perform a decide event.

Exercise 12.9 [92] The requirement of strong accuracy for failure detectors is stronger than the requirement that processes that never crash never be suspected. Give an example of a failure pattern and a failure detector history that satisfy the latter property but are not allowed in the case of a strongly accurate failure detector.

Exercise 12.10 Suppose that the implementation of the eventually strongly accurate failure detector is adapted as follows. If p receives a message from a suspected process q, then it no longer suspects q. If this message arrives $\nu + d_p + \rho$ time units after the previous message from q, with $\rho > 0$, then $d_p \leftarrow d_p + \rho$. Give an example to show that this failure detector need not be eventually strongly accurate.

Exercise 12.11 Consider a complete network of five processes. Apply the Chandra-Toueg 2-crash consensus algorithm, where initially four processes choose the value 0 and one process chooses the value 1. Give a computation in which all correct processes decide for 1.

Exercise 12.12 Suppose we adapt the Chandra-Toueg algorithm k-crash consensus for $k < \frac{N}{2}$ as follows. If the coordinator p_c receives *at least* (instead of more than) k

acknowledgments **ack**, then p_c decides for its value. Give an example to show that this could lead to inconsistent decisions.

Exercise 12.13 Give examples to show that *compare-and-set* is neither overwriting nor commuting. In other words, this operation does not blindly overwrite the value in a register, and performing two of these operations on a register in a different order may produce different results.

Exercise 12.14 Argue that the consensus algorithm using *compare-and-set* satisfies the termination, agreement, and validity properties.

Exercise 12.15 Give a wait-free consensus algorithm using *test-and-set* that works for two processes.

Exercise 12.16 Argue that a wait-free consensus algorithm for two processes cannot be achieved with only atomic read and write operations.

13

Consensus with Byzantine Failures

In this chapter, we consider Byzantine failures, meaning that a process may start to show behavior that is not in line with the specification of the distributed algorithm that is being executed. The class of Byzantine failures includes crash failures. We assume that processes are either correct or Byzantine from the start; a process does not know which of the other processes are Byzantine. The network topology is still assumed to be complete.

Again, the challenge we pose to the system is to let its correct processes agree on a value. In a (binary) Byzantine consensus algorithm, each correct process randomly chooses an initial value 0 or 1. In all executions of a Byzantine consensus algorithm, each correct process should perform exactly one decide event. The termination and agreement properties mentioned at the start of chapter 12 must be satisfied, together with a slightly adapted version of validity.

- *Validity*: If all *correct* processes choose the same initial value, then all correct processes decide for this value.

A k-Byzantine consensus algorithm, for a $k > 0$, is a Byzantine consensus algorithm that can cope with up to k Byzantine processes. Similar to the case of crash consensus, the validity requirement implies that each Byzantine consensus algorithm has a bivalent initial configuration (cf. exercise 12.1).

13.1 Bracha-Toueg Byzantine Consensus Algorithm

With Byzantine failures, fewer incorrect processes can be allowed than in the case of crashing processes, if one wants to achieve a Las Vegas consensus algorithm. A k-Byzantine consensus algorithm is possible only if $k < \frac{N}{3}$. Intuitively, this is for the following reason. Correct processes must be able to cope with k Byzantine processes, which may not cast a vote. So they can collect votes from at most $N - k$ processes, since otherwise a deadlock could occur. Among these $N - k$ processes, k could be Byzantine. Only if $k < \frac{N}{3}$ is it guaranteed that (in the worst case) the $N - 2 \cdot k$ collected votes from correct processes outnumber the k collected votes from Byzantine processes.

Theorem 13.1 *Let $k \geq \frac{N}{3}$. There is no Las Vegas algorithm for k-Byzantine consensus.*

Proof. Suppose, toward a contradiction, that such an algorithm does exist. Since $k \geq \lceil \frac{N}{3} \rceil$, the processes can be divided into two sets S and T that both contain N -k processes, while $S \cap T$ contains at most k processes.

Since S contains $N - k$ elements, the processes in S must be able to come to a decision by themselves if all processes outside S are Byzantine. In other words, S is always 0-potent or 1-potent. Likewise, T is always 0-potent or 1-potent. In each reachable configuration, S and T should be either both 0-potent or both 1-potent, for otherwise they could independently decide for different values. *Because the processes in $S \cap T$ could all be Byzantine, and could therefore be free to participate in an execution leading to a decision for b with the processes in S, while participating in an execution leading to a decision for* $1 - b$ *with the processes in T.*

Consider a bivalent initial configuration γ. Then there must be a configuration δ reachable from γ, and a transition $\delta \to \delta'$, with S and T both only b-potent in δ and only $(1 - b)$-potent in δ' , for some $b \in \{0, 1\}$. Since this transition would correspond to a single event, at a process in either S or T, clearly such a transition cannot exist. □

If $k < \frac{N}{3}$, then a Las Vegas algorithm for k-Byzantine consensus does exist. The Bracha-Toueg k-Byzantine consensus algorithm works roughly as follows. Just as in the Bracha-Toueg crash consensus algorithm, there are successive rounds in which every correct, undecided process sends its value to all processes and determines a new value, based on the first $N - k$ messages it receives in this round: the new value is 0 if most messages voted 0 and 1 otherwise. A process p decides for b if in a round it receives more than $\frac{N+k}{2}$ b-votes; note that by assumption $\frac{N+k}{2} < N - k$. In that case, p broadcasts ⟨**decide**, b⟩ and terminates. The other processes interpret ⟨**decide**, b⟩ as a b-vote by p for all rounds to come. Note that they cannot simply decide for b at the reception of this message, since p might be Byzantine. This completes the informal description of the algorithm, except for a verification phase of votes.

The following example shows that this algorithm is flawed if in a round two correct processes can accept different votes from a Byzantine process.

Example 13.1 Consider a network of four processes p, q, r, and s, with $k = 1$. Suppose that q is Byzantine. We study one possible computation of the Bracha-Toueg 1-Byzantine consensus algorithm. During each round, a correct process waits for three votes and needs three b-votes to decide for b. Initially, p and s randomly choose the value 1, and r chooses the value 0.

- In round 0, p, r, and s broadcast their values, while q sends a 0-vote to p and r and a 1-vote to s. Next, p and r take into account the 1-vote by p and 0-votes by q and r, and each sets its value to 0. Furthermore, s takes into account 1-votes by p, q, and s, as a result decides for 1, and broadcasts ⟨**decide**, 1⟩.
- In round 1, p, and r broadcast their values, while q broadcasts a 0-vote. Next, p and r take into account 0-votes by p, q, and r and as a result decide for 0. So we have inconsistent decisions.

The cause of the inconsistent decisions in example 13.1 is that in round 0, p and r use a 0-vote from q, while s uses a 1-vote from q, which can happen because q is Byzantine. To avoid this mishap, the Bracha-Toueg Byzantine consensus algorithm includes a verification phase of votes. When a process p receives a b-vote from a process q, p does not accept this vote straightaway. Instead, p echoes this vote to all processes, and it accepts a b-vote by q only if it receives an echo of a b-vote by q in this round from more than $\frac{N+k}{2}$ processes. This guarantees that the correct processes in a round never accept different votes from the same process, even if that process is Byzantine (see the proof of theorem 13.2).

As noted, a correct process decides for b if it accepts more than $\frac{N+k}{2}$ b-votes in a round. Since $\frac{N+k}{2} = \frac{N-k}{2} + k$, each correct process then will find that more than $\frac{N-k}{2}$ of the N -k votes it accepts in this round are b-votes. So, at the end of this round, all correct processes will take on the value b and hence will continue to do so for all future rounds. This guarantees that the correct processes all decide for the same value (again, see the proof of theorem 13.2).

We now give a more precise description of the Bracha-Toueg k-Byzantine consensus algorithm. Initially, each correct process randomly chooses a value 0 or 1. In each round $n \geq 0$, each correct, undecided process p acts as follows:

- p sends ⟨**vote**, n, $value_p$⟩ to all processes (including itself).
- When p receives ⟨**vote**, m, b⟩ from a process q, it sends ⟨**echo**, q, m, b⟩ to all processes (including itself).
- p stores incoming messages ⟨**vote**, m, b⟩ and ⟨**echo**, q, m, b⟩ with $m > n$ for future

rounds.

- p counts incoming messages ⟨**echo**, q, n, b⟩ for each pair q, b. When more than $\frac{N+k}{2}$ such messages have arrived, p accepts a b-vote by q.
- The round is completed when p has accepted votes from N -k processes. If most votes are for 0, then $value_p \leftarrow 0$. Otherwise $value_p \leftarrow 1$.
- If more than $\frac{N+k}{2}$ of the accepted votes are for b, then p decides for b, broadcasts ⟨**decide**, b⟩, and terminates.

The other processes interpret ⟨**decide**, b⟩ as a b-vote by p, and a b-echo by p for each q, for all rounds to come.

Processes keep track of whether multiple messages ⟨**vote**, m, _⟩ or ⟨**echo**, q, m, _⟩ arrive via the same channel; the sender must be Byzantine. To avoid miscounts, only the first of these messages is taken into account.

Theorem 13.2 *If the scheduling of messages is fair, then the Bracha-Toueg k-Byzantine consensus algorithm, for each* $k < \frac{N}{3}$, *is a Las Vegas algorithm that terminates with probability one.*

Proof. First, we prove that correct processes cannot decide for different values. Then we prove that the algorithm terminates with probability one, under the assumption that the scheduling of messages is fair.

During each round, the correct processes eventually accept N - k votes, since there are at least N - k correct processes that faithfully confirm each other's votes, and by assumption $N - k > \frac{N+k}{2}$.

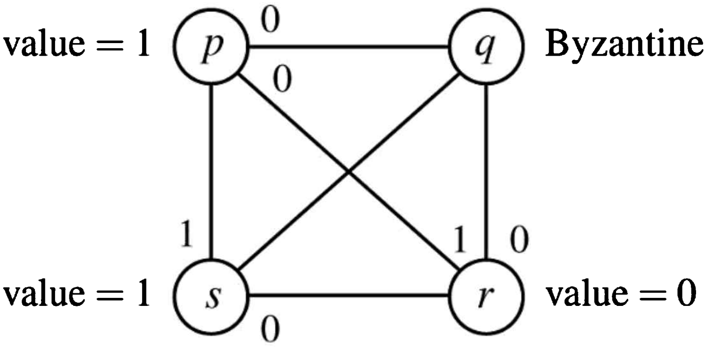
In a round n, let correct processes p and q accept votes for b and b' , respectively, from a (possibly Byzantine) process r. Then p and q received more than $\frac{N+k}{2}$ messages ⟨**echo**, r, n, b⟩ and ⟨**echo**, r, n, b' ⟩, respectively. Since this adds up to more than N +k messages in total, more than k processes, and therefore at least one correct process, sent such messages to both p and q. This implies that $b = b'$.

Suppose a correct process p decides for b at the end of a round n, meaning that p accepted more than $\frac{N+k}{2}$ b-votes. In every round, each correct process ignores votes from only k processes, and we argued that two correct processes never accept different values from the same process. So, since p accepted more than $\frac{N+k}{2}$ b-votes, all correct processes will accept more than $\frac{N+k}{2} - k = \frac{N-k}{2}$ b-votes in round n. Hence, after round n, all correct processes have the value b. If after a round m all correct processes have the value b, then after round m + 1 they still do, because they accept at least $N - 2 \cdot k > \frac{N-k}{2}$ b-votes in round m + 1. This implies that the correct processes stick to the value b from round n onward and hence cannot decide for different values.

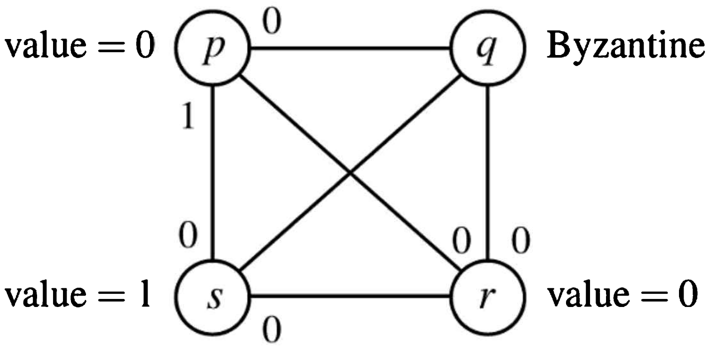
Let S be a set of N - k correct processes. Because of fair scheduling, in each round there is a probability $\rho > 0$ that each process in S accepts $N - k$ votes from the processes in S. With probability ρ^2 this happens in consecutive rounds $n, n + 1$. After round n, all processes in S have the same value b. After round n+1, all processes in S have decided for b, and broadcast ⟨**decide**, b⟩. To conclude, the algorithm terminates with probability one. □

Example 13.2 We revisit example 13.1 but now taking into account the verification phase of votes. It concerns a network of four processes p, q, r, and s, with $k = 1$, where q is Byzantine. We study one possible computation of the Bracha-Toueg 1-Byzantine consensus algorithm on this network. During each round, a correct process needs three confirming echoes to accept a vote, three accepted votes to complete the round, and three accepted b-votes to decide for b. Initially, p and s randomly choose the value 1, and r chooses the value 0. In the pictures of the consecutive rounds, for every process, two accepted votes from other processes and its own value in that round are depicted, which are used to determine the value of that process in the next round.

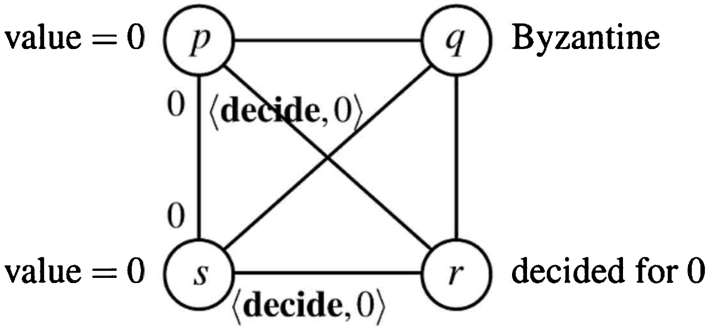
- In round 0, p and r accept the 1-vote by p and 0-votes by q and r, because they get confirmations for these votes from p, q, and r. As a result, each sets its value to 0. Furthermore, s accepts 1-votes by p and s, because it gets confirmations for these votes from p, r, and s. However, s does not accept the 1-vote by q, because it receives only two confirmations, from q and s. In the end, s accepts the 0-vote by r, for which it gets confirmations from p, r, and s, and sets its value to 1.



- In round 1, p and r accept 0-votes by p and q, because they get confirmations for these votes from p, q, and r. Moreover, r accepts the 0-vote by r, decides for 0, and broadcasts ⟨**decide**, 0⟩. On the other hand, p accepts the 1-vote by s, because it gets confirmations for these votes from p, r, and s, and sets its value to 0. Once again, s does not accept the 1-vote by q, because it receives only two confirmations, from q and s. Instead, s accepts 0-votes by p and r and the 1-vote by s, because it gets confirmations for these votes from p, r, and s. As a result, s sets its value to 0.



- Now that all correct processes have the value 0, in all future rounds this will remain the case. A decision for 0 by p and s could be postponed indefinitely if q gets a 1-vote accepted by p and s in successive rounds. However, in order to complete this computation swiftly, we let ⟨**decide**, 0⟩ arrive at p and s, and in round 2 we let p and s accept each other's 0-votes, so they decide for 0.



13.2 Mahaney-Schneider Synchronizer

In this section, we revisit bounded delay networks, in which there is an upper bound $d_{\max}$ on network latency; processes carry a local clock with ρ-bounded drift for some $\rho > 1$ (see section 11.2). It is shown that even in the presence of Byzantine processes, a precision $\delta > 0$ of the local clocks can be achieved: at any time τ and for each pair of correct processes p and q, $|C_p(\tau) - C_q(\tau)| \leq \delta$. Since local clocks may drift from real time, they need to be synchronized regularly to obtain such precision.

First, it is shown that such a synchronization algorithm to keep local clocks in sync is possible only if fewer than one-third of the processes are Byzantine.

Theorem 13.3 *Let $k \geq \frac{N}{3}$. There is no k-Byzantine clock synchronization algorithm.*

Proof. Suppose we want to achieve clock synchronization so that the local clocks have precision $\delta > 0$. Consider a network of three processes p, q, and r, where r is Byzantine, and let $k = 1$. (The following computation can be easily extended to general N and $k \geq \frac{N}{3}$; see exercise 13.4.)

Let the clock of p run faster than the clock of q. Suppose a synchronization round takes place at real time τ, when r sends $C_p(\tau) + \delta$ to p, and $C_q(\tau) - \delta$ to q. Since r reports a clock value to p that is within δ of p's local clock value, and since p can compare it only with the clock value from one other process (i.e., q), p cannot recognize that r is Byzantine. Likewise, q cannot recognize that r is Byzantine. So p and q must stay within range δ of the value reported by r, meaning that p cannot decrease and q cannot increase its clock value.

By repeating this scenario at each synchronization round, the clock values of p and q get farther and farther apart. So eventually p will need to choose whether its clock stays within range δ of q's or r's clock without being able to determine which of the two processes is Byzantine, and likewise for q. □

If at most $k < \frac{N}{3}$ processes are Byzantine, then clock synchronization can be achieved. To simplify the forthcoming presentation of Byzantine clock synchronization, we take the bound $d_{\max}$ on network latency to be zero. (In a sense, we assume that $d_{\max}$ is negligible compared to δ.)

In the Mahaney-Schneider k-Byzantine clock synchronization algorithm, processes regularly communicate their clock values to each other, and each process takes the average of all these clock values, whereby clock values that are provably from Byzantine processes are disregarded. It is assumed that at the start of a clock synchronization round, the local clocks have precision $\delta > 0$. This algorithm achieves that after a clock synchronization round, the local clocks have precision $\frac{2}{3} \cdot \delta$.

The Mahaney-Schneider synchronizer proceeds in synchronization rounds, in which each correct process

- collects the clock values of all processes,
- discards those reported values τ for which fewer than $N - k$ processes report a value in the interval $[\tau - \delta, \tau + \delta]$ (they are from Byzantine processes),
- replaces all discarded and nonreceived values by an accepted value, and
- takes the average of these N values as its new clock value.

Theorem 13.4 *The Mahaney-Schneider synchronizer is a k-Byzantine clock synchronization algorithm for each* $k < \frac{N}{3}$.

Proof. Consider two correct processes p and q in some synchronization round. First, we argue that if in this round p and q accept clock values a_p and a_q, respectively, then

$$|a_p - a_q| \leq 2\cdot\delta.$$

Since p accepted a_p, in this round at least $N - k$ processes reported a value in $[a_p - \delta, a_p + \delta]$ to p. Likewise, at least $N - k$ processes reported a value in $[a_q - \delta, a_q + \delta]$ to q. So at least $N - 2\cdot k$ processes reported a value in $[a_p - \delta, a_p + \delta]$ to p as well as a value in $[a_q - \delta, a_q + \delta]$ to q. Since $N - 2\cdot k > k$, there is a correct process among these processes. This correct process reports the same value to p and q, so $[a_p - \delta, a_p + \delta]$ and $[a_q - \delta, a_q + \delta]$ have a nonempty intersection. This implies $|a_p - a_q| \leq 2\cdot\delta$.

For each process r, let a_{pr} and a_{qr} denote the value that p and q, respectively, accept from (or assign to) r. Then p and q compute as new clock values $\frac{1}{N}\cdot(\sum_{\text{processes } r} a_{pr})$ and $\frac{1}{N}\cdot(\sum_{\text{processes } r} a_{qr})$, respectively. By the preceding argument, $|a_{pr} - a_{qr}| \leq 2\cdot\delta$ for all processes r. And $a_{pr} = a_{qr}$ for all correct processes r. Hence,

$$\left|\frac{1}{N}\cdot\left(\sum_{\text{processes } r} a_{pr}\right) - \frac{1}{N}\cdot\left(\sum_{\text{processes } r} a_{qr}\right)\right| \leq \frac{1}{N}\cdot k\cdot 2\cdot\delta < \frac{2}{3}\cdot\delta. \qquad \square$$

In the proof of theorem 13.4, it was shown that, after a synchronization round, the clocks at correct processes have a precision smaller than $\frac{2}{3}\cdot\delta$. If the local clocks have been synchronized at time τ, then because of ρ-bounded drift of local clocks, if no synchronization takes place in the time interval $[\tau, \tau + R]$,

$$|C_p(\tau+R) - C_q(\tau+R)| < \frac{2}{3}\cdot\delta + \left(\rho - \frac{1}{\rho}\right)\cdot R < \frac{2}{3}\cdot\delta + \rho\cdot R.$$

So, to achieve precision δ, a synchronization of local clocks should be performed every $\frac{\delta}{3\cdot\rho}$ time units.

13.3 Lamport-Shostak-Pease Broadcast Algorithm

To cope with crash failures, failure detectors were introduced in section 12.3 to try to detect crashed processes. Both implementations of failure detectors that we discussed are based on the absence of messages from a crashed process over a certain period of time. Detecting Byzantine processes is much more complicated, since they can keep on sending messages, and in general it is far from easy to determine that a process is performing events that are not in line with the specification of the distributed algorithm that is being executed. Therefore, to obtain an always correctly terminating consensus algorithm in the presence of Byzantine processes, another strategy is followed: the network is transformed into a synchronous system. As discussed in section 11.2, this can be realized in a bounded delay network if all processes have a ρ-bounded clock, using also the Mahaney-Schneider synchronizer from the previous section to achieve some precision δ for these clocks. In a synchronous network, it can be guaranteed that the votes from all correct processes are taken into account in each voting round. This will make it possible to accomplish consensus within a finite number of voting rounds.

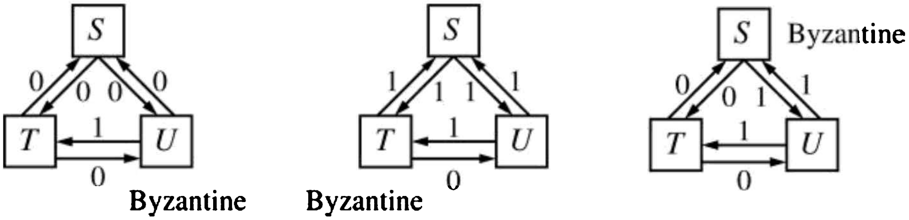
In this section and the next, we consider a variation on the Byzantine consensus problem, called *Byzantine broadcast*, in the setting of synchronous networks. One process, called the general, randomly chooses an initial value 0 or 1. The other processes, called lieutenants, know who the general is. In each execution of a (binary) Byzantine broadcast algorithm, the requirements termination and agreement must be satisfied, together with a variation on validity.

- *Dependence*: If the general is correct, then it decides for its own initial value.

A k-Byzantine broadcast algorithm can cope with at most k Byzantine processes. It will come as no surprise that for $k \geq \frac{N}{3}$ such an algorithm does not exist, even if we have a synchronous network.

Theorem 13.5 *Let $k \geq \frac{N}{3}$. There is no (always correctly terminating) k-Byzantine broadcast algorithm for synchronous networks.*

Proof. Suppose, toward a contradiction, that a k-Byzantine broadcast algorithm for synchronous networks does exist. Divide the processes into three disjoint sets S, T, and U, each with at most k elements. Let the general be in S. We consider three different scenarios, which are depicted as follows.



In the first scenario, at the left, the processes in U are Byzantine and the general starts with the value 0. The processes in S and T propagate 0. Moreover, by dependence and agreement, all processes in $S \cup T$ should decide for 0. The Byzantine processes in U, however, propagate 1 to the processes in T.

In the second scenario, in the middle, the processes in T are Byzantine and the general starts with the value 1. The processes in S and U propagate 1. Moreover, by dependence and agreement, all processes in $S \cup U$ should decide for 1. The Byzantine processes in T, however, propagate 0 to the processes in U.

In the third scenario, at the right, the processes in S, including the general, are Byzantine. The processes in S propagate 0 to T, and as a result the processes in T do so also. Likewise, the processes in S propagate 1 to U, and as a result the processes in U do so also. Note that the inputs of the processes in T agree with the first scenario, so they all decide for 0. By contrast, the inputs of the processes in U agree with the second scenario, so they all decide for 1. This violates agreement. □

The Lamport-Shostak-Pease broadcast algorithm for synchronous networks is a k-Byzantine broadcast algorithm for each $k < \frac{N}{3}$. Since the algorithm contains recursive calls to itself, we denote it with $Broadcast_g(N, k)$, where g denotes the general. Let *majority* be a (deterministic) function that maps each multiset of elements from {0, 1} either to 0 or to 1; in a multiset, the same element can occur multiple times. If more than half of the elements in a multiset M are equal to b, then it is required that $majority(M) = b$.

The algorithm $Broadcast_g(N, k)$ proceeds in $k + 1$ pulses.

- *Pulse* 1: The general g, if correct, broadcasts its value and decides for its value. Each correct lieutenant q acts as follows. If q receives a value b from g, then it sets its value to b; otherwise it sets its value to 0.
  - If $k = 0$, then q decides for its value.
  - If $k > 0$, then for every lieutenant p, q takes part in $Broadcast_p(N - 1, k - 1)$, starting in pulse 2.
- *Pulse* $k + 1$ (if $k > 0$): Each correct lieutenant q has, for every lieutenant p, computed a value in $Broadcast_p(N - 1, k - 1)$; it stores these $N - 1$ values in a multiset $Decisions_q$. Finally, q sets its value to $majority(Decisions_q)$ and decides for this value.

Note that if the general is correct, then it can decide immediately, because by the dependence requirement all correct lieutenants are supposed to decide for the general's value. If $k = 0$, then the general is certainly correct, so the lieutenants can also decide immediately for the received value. However, if $k > 0$, then the lieutenants must take into account that the general may be Byzantine. Therefore, each correct lieutenant p starts a recursive call $Broadcast_p(N - 1, k - 1)$, reminiscent of the echo principle in the Bracha-Toueg Byzantine consensus algorithm; the general is excluded from these calls. This gives rise to a cascade of recursive calls because if $k > 1$, then each of these calls starts another $N - 2$ recursive calls, and so on.

We now argue the correctness of the Lamport-Shostak-Pease broadcast algorithm in an informal fashion; a more formal proof will be given at the end of this section. If the general is correct, each recursive call $Broadcast_p(N - 1, k - 1)$ with p correct is guaranteed to produce the general's value. Since a majority of the lieutenants are correct, by the definition of the *majority* function, all correct lieutenants will decide for the general's value. On the other hand, if the general is Byzantine, by induction on k, all correct lieutenants are guaranteed to compute the same values in the $N - 1$ recursive calls $\mathit{Broadcast}_p(N - 1, k - 1)$. So because of the deterministic nature of the *majority* function, in pulse $k + 1$ all correct lieutenants will compute and decide for the same value in $\mathit{Broadcast}_g(N, k)$.

We consider two examples of the Lamport-Shostak-Pease broadcast algorithm, one where the general is correct and one where the general is Byzantine.

Example 13.3 Consider a complete network of four processes g, p, q, and r, with $k = 1$. Suppose that the general g and lieutenants p and q are correct, while lieutenant r is Byzantine. We study one possible computation of the Lamport-Shostak-Pease broadcast algorithm $\mathit{Broadcast}_g(4, 1)$ on this network.

In pulse 1, g broadcasts and decides for 1. So after pulse 1, p and q carry the value 1. These two correct lieutenants build, by the three recursive calls $\mathit{Broadcast}_s(3, 0)$ with $s \in \{p, q, r\}$, multisets $\{1, 1, b_p\}$ and $\{1, 1, b_q\}$. The values b_p and b_q may be distinct if in $\mathit{Broadcast}_r(3, 0)$ the Byzantine process r sends different values to p and q. Since the majority of values in these two multisets are 1, both p and q decide for 1, irrespective of the values b_p and b_q.

Example 13.4 Consider a complete network with $N = 7$ and $k = 2$. Suppose that the general g and one lieutenant r are Byzantine. We study one possible computation of the Lamport-Shostak-Pease broadcast algorithm $\mathit{Broadcast}_g(7, 2)$ on this network.

In pulse 1, the general sends 0 to three correct lieutenants and 1 to the two other correct lieutenants. The five correct lieutenants all build, using the six recursive calls $\mathit{Broadcast}_s(6, 1)$, the same multiset $M = \{0, 0, 0, 1, 1, b\}$ for some $b \in \{0, 1\}$. That is, even in $\mathit{Broadcast}_r(6, 1)$, all correct lieutenants are guaranteed to compute the same value b, so they all decide for $\mathit{majority}(M)$.

For instance, in $\mathit{Broadcast}_r(6, 1)$, the Byzantine lieutenant r could send 0 to two correct lieutenants and 1 to the three other correct lieutenants. Then the five correct lieutenants all build, using five recursive calls $\mathit{Broadcast}_s(5, 0)$, the same multiset $M = \{0, 0, 1, 1, 1\}$. So in this case, $b = \mathit{majority}(\{0, 0, 1, 1, 1\}) = 1$.

Theorem 13.6 *The Lamport-Shostak-Pease broadcast algorithm is a k-Byzantine broadcast algorithm for each* $k < \frac{N}{3}$.

Proof. First, we prove that if the general g is correct and fewer than $\frac{N-k}{2}$ lieutenants are Byzantine, then in $\mathit{Broadcast}_g(N, k)$ all correct processes decide for g's value. (This holds even if $\frac{N-k}{2} > k$.)

We prove this claim by induction on k. The base case $k = 0$ is trivial because g is assumed to be correct, so all correct processes decide for g's value in pulse 1. We now consider the inductive case $k > 0$. Since g is assumed to be correct, in pulse 1 each correct lieutenant sets its value to g's value. Since $\frac{(N-1)-(k-1)}{2} = \frac{N-k}{2}$, by induction, for all correct lieutenants q, in $Broadcast_q(N - 1, k - 1)$ the value of q, meaning g's value is computed. By assumption, a majority of the lieutenants are correct (because $k > 0$), so in pulse k+1 a majority of the values that a correct lieutenant computes in the recursive calls $Broadcast_q(N -1, k-1)$ equal g's value. Hence, by the definition of the *majority* function, in pulse $k + 1$ each correct lieutenant computes and decides for g's value.

Now we prove that the Lamport-Shostak-Pease broadcast algorithm is a k-Byzantine broadcast algorithm for each $k < \frac{N}{3}$, again by induction on k. The case where g is correct follows immediately from the statement proved earlier, together with the fact that $\frac{N}{3} < \frac{N-k}{2}$. Therefore, we can assume that g is Byzantine (so $k > 0$). Then at most $k - 1$ lieutenants are Byzantine. Since $k < \frac{N}{3}$ implies $k - 1 < \frac{N-1}{3}$, by induction all correct lieutenants compute the same value in $Broadcast_p(N - 1, k - 1)$ for every lieutenant p. Hence, all correct lieutenants compute the same multiset M in pulse k +1. So all correct lieutenants decide for the same value $majority(M)$. □

13.4 Lamport-Shostak-Pease Authentication Algorithm

A *public-key cryptosystem* consists of a finite (but very large) message domain $\mathcal{M}$ and, for each process q, functions $S_q, P_q : \mathcal{M} \to \mathcal{M}$ with $S_q(P_q(m)) = P_q(S_q(m)) = m$ for all $m \in \mathcal{M}$. The key S_q is kept secret by q, while P_q is made public. The underlying assumption is that computing S_q from P_q is very expensive.

If a process p wants to send a secret message m to a process q, then p can send $P_q(m)$ to q, because only q knows the key S_q that is needed to decrypt this message. Furthermore, if p wants to send a signed message m to q such that other processes cannot fraudulently sign their messages with p's signature, then p can send $\langle m, S_p(m)\rangle$ to q. Because only p can compute $S_p(m)$, and q can obtain p's public key P_q, apply this to the signature $S_p(m)$, and check whether the result equals the message m. Public-key cryptosystems and digital signatures will be discussed in more depth in section 18.1.

The Lamport-Shostak-Pease authentication algorithm for synchronous systems uses this signature principle based on a public-key cryptosystem in such a way that the Byzantine processes cannot lie about the values they have received. It is a correct and terminating k-Byzantine broadcast algorithm for each k. Every process p has a private key S_p and a public key P_q. We step away from binary Byzantine broadcast: the correct processes must uniformly pick a value from a large domain of possible values. The reason is that in a small domain (in the case of binary consensus consisting of only two values), computing S_q from P_q is easy. It is assumed that processes $q \neq p$ cannot guess values $S_p(v)$ without acquiring extra knowledge about S_p.

The Lamport-Shostak-Pease authentication algorithm proceeds in $k + 1$ pulses.

- In pulse 1, the general, if correct, broadcasts $\langle value_g, (S_g(value_g), g)\rangle$ and decides for $value_g$.
- If in a pulse $i \leq k + 1$ a correct lieutenant q receives a message $\langle v, (\sigma_1, p_1): \cdots : (\sigma_i, p_i)\rangle$ that is *valid*, meaning that
  * $p_1 = g$,
  * $p_1, ..., p_i, q$ are all distinct, and
  * $P_{p_k}(\sigma_k) = v$ for all $k = 1, ..., i$,

  then q includes v in the set $Values_q$.
  If $i \leq k$, then in pulse $i + 1$, q sends to all other lieutenants the message

$$\langle v, (\sigma_1, p_1) : \cdots : (\sigma_i, p_i) : (S_q(v), q)\rangle.$$

- After pulse $k + 1$, each correct lieutenant q decides for v if $Values_q$ is a singleton $\{v\}$, or 0 otherwise. (In the latter case, the general is Byzantine.)

The second part of a message is a list of signatures. Suppose a correct lieutenant q receives a valid message, meaning that the signatories are distinct, include the general and not the receiver, and each signatory p has added the correct signature $S_p(v)$, where v is the value contained in the first part of the message. Then q takes v on board as a possible value, adds its signature to the list of signatures, and broadcasts the message. A key observation is that each such message in pulse $k + 1$ contains a list of $k + 1$ signatures and thus at least one signature from a correct process. This means that no later than pulse $k + 1$, every lieutenant will have received a valid message with this value. As a result, after pulse $k + 1$, all correct lieutenants have computed the same set *Values*, so they all decide for the same value. And if the general is correct, then these sets are guaranteed to contain only $value_g$ because then Byzantine processes cannot forge signatures by the general for other values. This will be argued more formally in the proof of theorem 13.7.

Example 13.5 Consider a complete network of four processes g, p, q, and r, with $k = 2$. Suppose that the general g and lieutenant r are Byzantine. We study one possible computation of the Lamport-Shostak-Pease authentication algorithm:

- In pulse 1, g sends $\langle 0, (S_g(0), g)\rangle$ to p and q and $\langle 1, (S_g(1), g)\rangle$ to r. Then $Values_p$ and $Values_q$ become $\{0\}$.
- In pulse 2, p and q broadcast $\langle 0, (S_g(0), g): (S_p(0), p)\rangle$ and $\langle 0, (S_g(0), g): (S_q(0), q)\rangle$, respectively. And r sends $\langle 1, (S_g(1), g): (S_r(1), r)\rangle$ to only q. Then $Values_p$ remains $\{0\}$, while $Values_q$ becomes $\{0, 1\}$.
- In pulse 3, q broadcasts $\langle 1, (S_g(1), g): (S_r(1), r): (S_q(1), q)\rangle$. Then $Values_p$ becomes $\{0, 1\}$.
- After pulse 3, p and q both decide for 0 because $Values_p$ and $Values_q$ contain two

elements.

Theorem 13.7 *The authentication algorithm from Lamport-Shostak-Pease is a k-Byzantine broadcast algorithm for each $k < N$.*

Proof. If the general is correct, then Byzantine processes will not get an opportunity to forge a signature for a value different from $value_g$ on behalf of the general. So owing to authentication, correct lieutenants will only add $value_g$ to their set *Values*. Hence, all processes will decide for $value_g$.

Suppose a correct lieutenant q receives a valid message $\langle v, \ell\rangle$ in a pulse $i = 1, ..., k + 1$. We distinguish between two cases:

- $i \le k$: Then, in the next pulse, q broadcasts a message that will make all correct lieutenants add v to their set *Values*.
- $i = k + 1$: Since the list ℓ of signatures has length $k + 1$, it contains a correct process p. Then p received a valid message $\langle v, \ell' \rangle$ in a pulse $j \le k$. In pulse $j + 1$, p has broadcast a message that makes all correct lieutenants add v to their set *Values*.

So after pulse $k + 1$, $Values_q$ is the same for all correct lieutenants q. Hence, after pulse $k + 1$, they all decide for the same value. □

The Dolev-Strong optimization minimizes the number of required messages. This optimization lets each correct lieutenant broadcast at most two messages, with different values. When it has broadcast two different values, all correct lieutenants are certain to compute a set *Values* with at least two values, so all correct lieutenants will decide for 0.

Bibliographical notes

The Bracha-Toueg Byzantine consensus algorithm stems from [13]. The Mahaney-Schneider synchronizer originates from [62]. The Lamport-Shostak-Pease broadcast and authentication algorithms were proposed in [56]. The Dolev-Strong optimization is due to [31].

13.5 Exercises

Exercise 13.1 Consider a complete network of four processes, in which one process is Byzantine. Apply the Bracha-Toueg 1-Byzantine consensus algorithm, where initially one correct process chooses the value 0 and two correct processes choose the value 1. Give a computation in which all correct processes decide for 0.

Exercise 13.2 [92] In the Bracha-Toueg k-Byzantine consensus algorithm, suppose that more than $\frac{N+k}{2}$ correct processes initially choose the value b. Explain why the correct processes will eventually decide for b.

Exercise 13.3 Consider three correct processes p_0, p_1, and p_2 and a Byzantine process q. Let the local clocks of p_0 and p_1 run half as fast as real time, and let the local clock of p_2 run twice as fast as real time. Consider the Mahaney-Schneider synchronizer for some precision $\delta > 0$, whereby it is assumed that $d_{\max} = 0$ (that is, messages are communicated instantaneously).

(a) What is the smallest $\rho > 1$ for which the clocks of p_0, p_1, and p_2 are all ρ-bounded?
(b) Suppose that, at real time τ, the clocks of p_0 and p_1 are at $\tau - \frac{\delta}{3}$, while the clock at p_2 is at $\tau + \frac{\delta}{3}$. Let p_0, p_1, and p_2 synchronize their clocks at real time $\tau + \frac{\delta}{3\cdot\rho}$ (with the ρ from part (a)). Show that whatever input q gives at this synchronization point, the clocks of p_0, p_1, and p_2 are less than $\frac{2\cdot\delta}{3}$ apart.
(c) Repeat the exercise in part (b) but now leaving out p_0. Show that in this case, after synchronization at real time $\tau + \frac{\delta}{3\cdot\rho}$, the clocks of p_1 and p_2 can be more than $\frac{2\cdot\delta}{3}$ apart.

Exercise 13.4 Argue the impossibility of k-Byzantine clock synchronization for general N and $k \geq \frac{N}{3}$.

Exercise 13.5 The k-Byzantine synchronizer of Lamport and Melliar-Smith differs from the Mahaney-Schneider synchronizer in one aspect: a correct process p accepts a local clock value of another process q if at the moment of synchronization it differs by no more than δ from its own clock value. Explain why that synchronizer has precision $\frac{3\cdot k}{N} \cdot \delta$ (versus precision $\frac{2\cdot k}{N} \cdot \delta$ of the Mahaney-Schneider synchronizer).

Exercise 13.6 Explain how the Mahaney-Schneider synchronizer must be adapted if $d_{\max} > 0$.

Exercise 13.7 Apply the Lamport-Shostak-Pease broadcast algorithm to a complete network of five processes, with $k = 1$. Let the general g be Byzantine. Suppose that, in pulse 1, g sends the value 1 to two lieutenants, and the value 0 to the two other lieutenants. Give a computation of $Broadcast_g(5, 1)$ (including a definition of the *majority* function) such that all lieutenants decide for 0.

Exercise 13.8 Apply the Lamport-Shostak-Pease broadcast algorithm to a complete network of seven processes, with $k = 2$. Let the general g and one lieutenant be Byzantine. Give a computation of $Broadcast_g(7, 2)$ (and its subcalls) in which all correct lieutenants decide for *majority*({0, 0, 0, 1, 1, 1}).

Exercise 13.9 Apply the Lamport-Shostak-Pease broadcast algorithm to a complete network of eight processes, with $k = 2$. Let three of the lieutenants be Byzantine. (The general g and the four other lieutenants are correct.) Give a computation of $Broadcast_g(8, 2)$ (and its subcalls) in which a correct lieutenant decides for a value different from g. (This shows that the bound $\frac{N-k}{2}$ at the start of the proof of theorem 13.6 cannot be replaced by $\frac{N-1}{2}$.)

Exercise 13.10 Determine the worst-case message complexity of the correct processes in $Broadcast_g(N, k)$.

Exercise 13.11 Apply the Lamport-Shostak-Pease authentication algorithm to a complete network of five processes. Let three of the processes be Byzantine. Give a computation in which the two correct processes would decide for different values at the end of pulse 3 but decide for the same value in pulse 4.

Exercise 13.12 Let $k \geq N$-1. Explain why the Lamport-Shostak-Pease authentication algorithm can then be adapted by letting it already terminate at the end of pulse N - 1.

Exercise 13.13 Determine the worst-case message complexity of the correct processes in the Lamport-Shostak-Pease authentication algorithm, taking into account the Dolev-Strong optimization.

14
Mutual Exclusion

Mutual exclusion in distributed systems addresses the problem that different processes may concurrently want to perform a related task but should perform these tasks sequentially. Mutual exclusion is of vital importance in a shared-memory setting in order to serialize access to a shared resource. Notably, a thread may want to lock a block of shared memory, to ensure exclusive access to it. For example, purchasing a plane ticket or a specific seat on an airplane over the Internet requires mutual exclusion to avoid having different persons acquire the same seat on a flight. Mutual exclusion is also important in a message-passing framework, for instance to avoid race conditions, such as serializing control signals to an I/O device.

Although multiple processes may want to access the resource concurrently, at any moment in time at most one process should be *privileged*, meaning that it is allowed access. A process that becomes privileged is said to enter its *critical section*, which is a block of source code where the process needs access to the resource. When a process gives up the privilege, it is said to exit its critical section. Mutual exclusion algorithms are supposed to satisfy the following two properties in each execution:

- *Mutual exclusion*: In every configuration, at most one process is privileged.
- *Starvation-freeness*: If a process tries to enter its critical section and no process remains privileged forever, then this process will eventually become privileged.

In a message-passing setting, mutual exclusion algorithms are generally based on one of the following three paradigms:

- *A logical clock*: Requests for entering a critical section are prioritized by means of logical time stamps.
- *Token passing*: The process holding the token is privileged.
- *Quorums*: To become privileged, a process needs the permission from a quorum of processes. Each pair of quorums should have a nonempty intersection.

We will first discuss one mutual exclusion algorithm for each of these categories. Next, we will consider several mutual exclusion algorithms for shared memory, all of which use spinning, meaning that values of registers are read repeatedly until some condition is met.

14.1 Ricart-Agrawala Algorithm

The Ricart-Agrawala mutual exclusion algorithm uses a logical clock (see section 2.2). For simplicity, we assume that the network topology is complete. Processes are indexed: $p_0, ..., p_{N-1}$. When a process p_i wants to enter its critical section, it sends the message ⟨**request**, ts_i, i⟩ to all other processes, with ts_i its logical time stamp. The second argument of this message, the index i, is meant to break ties between competing processes that send concurrent requests with the same logical time stamp. Then the request from the process with the lowest index has the highest priority. When another process p_j receives this request, it sends permission to p_i as soon as

- p_j is not privileged, and
- p_j does not have a pending request with a logical time stamp ts_j, where $(ts_j, j) < (ts_i, i)$ (with respect to the lexicographical order).

Process p_i enters its critical section when it has received permission from all other processes. When p_i exits its critical section, it sends permission in reply to all pending requests.

Actually, the Ricart-Agrawala algorithm does not require a full-blown logical clock. It suffices that only requests be taken into account. That is, if a process receives a request with time stamp t, then it increases its clock value to $t + 1$.

We consider two examples, both with $N = 2$. Let clocks start at time 0.

Example 14.1 p_1 sends ⟨**request**, 0, 1⟩ to p_0. Upon receipt of this request, p_0 sends permission to p_1 and sets its clock value to 2. Next, p_0 sends ⟨**request**, 2, 0⟩ to p_1. When p_1 receives this message, it does not send permission to p_0, because $(0, 1) < (2, 0)$. Finally, p_1 receives permission from p_0 and enters its critical section.

Example 14.2 p_0 and p_1 concurrently exchange requests: p_0 sends $\langle$**request**, 0, 0$\rangle$ to p_1, and p_1 sends $\langle$**request**, 0, 1$\rangle$ to p_0. When p_0 receives the request from p_1, it does not send permission to p_1, because $(0, 0) < (0, 1)$. When p_1 receives the request from p_0, it does send permission to p_0, because $(0, 0) < (0, 1)$. Finally, p_0 receives permission from p_1 and enters its critical section.

We argue that the Ricart-Agrawala algorithm guarantees mutual exclusion. Suppose a process p wants to enter its critical section and sends requests to all other processes. When another process q sends permission to p, q is not privileged. Moreover, either q's pending request is smaller than p's request or the fact that p's request is taken into account in q's logical clock ensures that q's future request will be larger than p's request. Hence, q will not get permission from p to enter its critical section until after p has entered and left its critical section. The Ricart-Agrawala algorithm is also starvation-free, because each request will eventually become the smallest request in the network.

A drawback of the Ricart-Agrawala algorithm is the high message overhead because requests must be sent to all other processes. The Carvalho-Roucairol optimization reduces this message overhead. Suppose a process p_i entered its critical section earlier and wants to enter it again. Then p_i only needs to send requests to the set of processes $Requests_i$ to which p_i has sent permission since the last exit from its critical section. This set initially contains all processes except p_i, and is emptied every time p_i exits its critical section. When p_i sends permission to a process q, it adds q to $Requests_i$. We argue that in all reachable configurations, $p_i \in Requests_j$ or $p_j \in Requests_i$ for each pair of different processes p_i, p_j. In initial configurations, this trivially holds. Now consider a reachable configuration in which $p_i \in Requests_j$. (The case $p_j \in Requests_i$ can be treated likewise.) To remove p_i from $Requests_j$, p_j must (enter and) exit its critical section. When p_j becomes privileged, it must have received permission from p_i. As a result, $p_j \in Requests_i$. This completes the argument. Since in each reachable configuration, for every pair of different processes at least one of them must ask permission from the other, mutual exclusion is guaranteed. If a process $q \notin Requests_i$ sends a request to p_i while p_i is waiting for permissions, and q's request is smaller than p_i's outstanding request, then p_i sends both permission and a request to q.

14.2 Raymond's Algorithm

Raymond's mutual exclusion algorithm is based on token passing; only the process holding the token may be privileged. It assumes an undirected network and starts with building a spanning tree in this network. At any moment, the root of this sink tree is supposed to hold the token. The root of the spanning tree therefore changes over time, meaning that parent-child relations in the tree may be inverted.

Each process maintains a FIFO queue, which can contain IDs of its children in the sink tree, and its own ID. An enqueue operation adds an ID at the tail of the queue, while a dequeue operation removes the ID at the other side of the queue, called the head, if the queue is nonempty. Initially, the queue is empty. A process maintains its queue as follows:

- When a nonroot wants to enter its critical section, it enqueues its ID at the tail of its own queue.
- Each time a nonroot gets a new head at its (nonempty) queue, it sends a request for the token to its parent in the sink tree.
- When a process receives a request for the token from a child, it enqueues the ID of this child at the tail of its own queue.
- When the root has left its critical section and its queue is or becomes nonempty, it sends the token to the process q at the head of its queue, makes q its parent, and dequeues q's ID from the head of its queue.

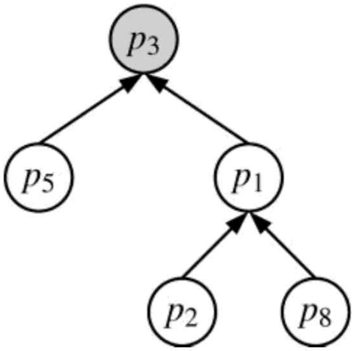
In the special case where the root wants to enter its critical section again and its queue is empty, it can become privileged straightaway.

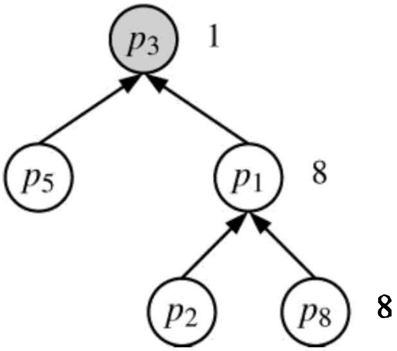
Let a nonroot p get the token from its parent, and let the ID of the process q be at the head of p's queue.

- If $p \neq q$, then p sends the token to q and makes q its parent.
- If $p = q$, then p becomes the root; that is, it has no parent and is privileged.

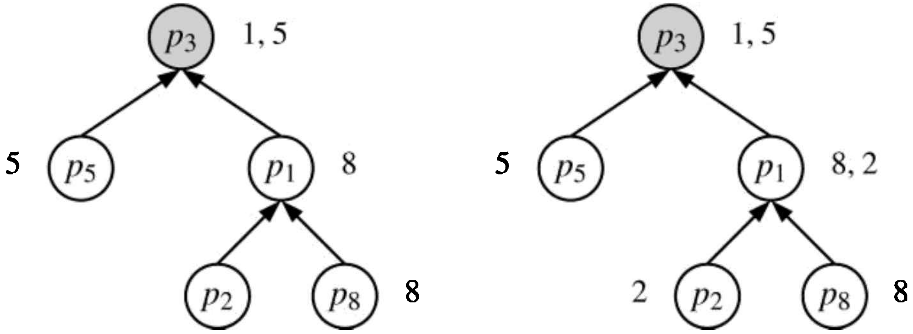
In both cases, p removes q's ID from the head of its queue.

Example 14.3 We consider one possible computation of Raymond's algorithm, on an undirected network of five processes. In every picture, the gray process is the root.

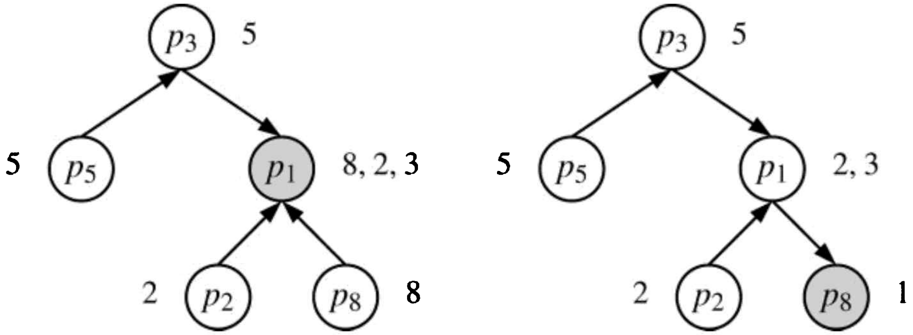




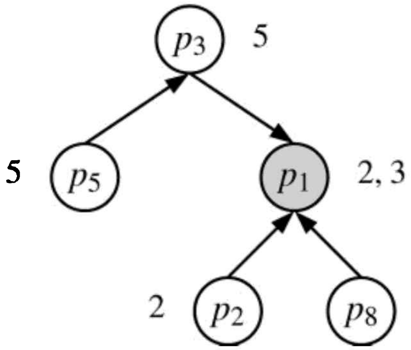
Initially, in the preceding picture at the left, p_3 is the root and is in its critical section. Next, in the picture at the right, the following has happened. Since p_8 wants to enter its critical section, it enqueued its own ID in its queue. Since this was a new head of its queue, p_8 sent a request for the token to its parent p_1. As a result, p_1 also placed the ID 8 in its queue. Since this was a new head of its queue, p_1 sent a request for the token to its parent p_3. As a result, p_3 enqueued the ID 1 in its queue.

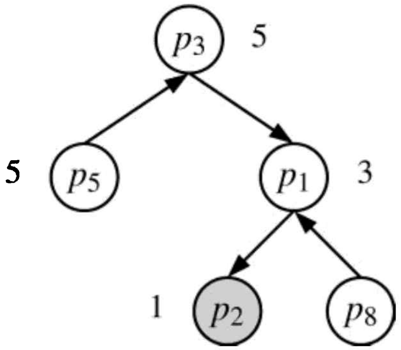


In the preceding picture at the left, p_5 wants to enter its critical section, so it enqueued its own ID in its queue. Since this was a new head of its queue, p_5 sent a request for the token to its parent p_3. As a result, p_3 also enqueued the ID 5 in its queue. In the picture at the right, p_2 wants to enter its critical section, so it enqueued its own ID in its queue. Since this was a new head of its queue, p_2 sent a request for the token to its parent p_1. As a result, p_1 also enqueued the ID 2 in its queue.

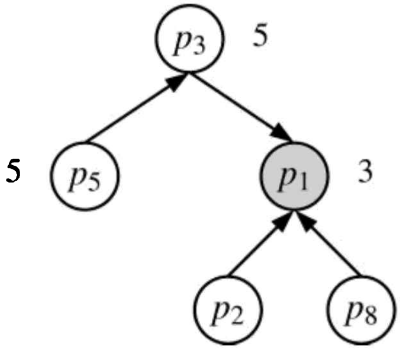


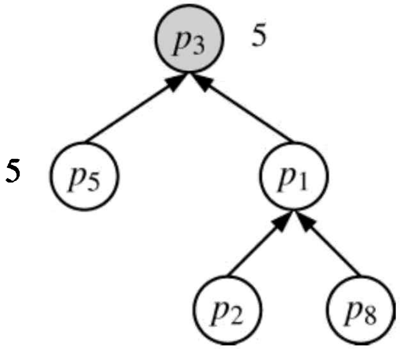
In the preceding picture at the left, p_3 has exited its critical section. Since the ID 1 was the head of its queue, it sent the token to p_1, made p_1 its parent, and dequeued 1 from its queue. Then 5 became the new head of its queue, so p_3 sent a request for the token to its parent p_1. When p_1 received the token, it became the root. Moreover, the request from p_3 made p_1 enqueue the ID 3 in its queue. In the picture at the right, since the ID 8 was the head of its queue, p_1 forwarded the token to p_8, made p_8 its parent, and dequeued 8 from its queue. Then 2 became the new head of its queue, so p_1 sent a request for the token to its parent p_8. When p_8 received the token, it became the root, dequeued 8 from its queue, and entered its critical section. Moreover, the request from p_1 made p_8 enqueue the ID 1 in its queue.



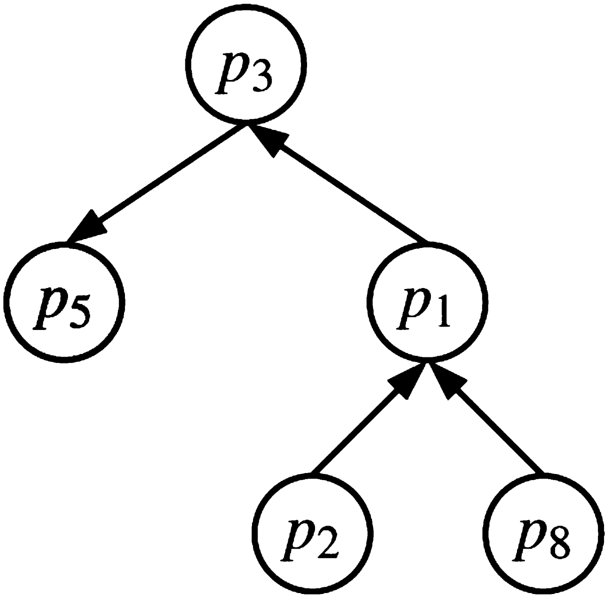


In the preceding picture at the left, p_8 has exited its critical section. Since the ID 1 was the head of its queue, it sent the token to p_1, made p_1 its parent, and dequeued 1 from its queue. When p_1 received the token, it became the root. In the picture at the right, since the ID 2 was the head of its queue, p_1 forwarded the token to p_2, made p_2 its parent, and dequeued 2 from its queue. Then 3 became the new head of its queue, so p_1 sent a request for the token to its parent p_2. When p_2 received the token, it became the root, dequeued 2 from its queue, and entered its critical section. Moreover, the request from p_1 made p_2 enqueue the ID 1 in its queue.





In the preceding picture at the left, p_2 has exited its critical section. Since the ID 1 was the head of its queue, it sent the token to p_1, made p_1 its parent, and dequeued 1 from its queue. When p_1 received the token, it became the root. In the picture at the right, since the ID 3 was the head of its queue, p_1 forwarded the token to p_3, made p_3 its parent, and dequeued 3 from its queue. When p_3 received the token, it became the root.



In the final picture, since the ID 5 was the head of its queue, p_3 sent the token to p_5, made p_5 its parent, and dequeued 5 from its queue. When p_5 received the token, it became the root, dequeued 5 from its queue, and entered its critical section.

Whenever a process holds the token, it is the root of the sink tree. So Raymond's algorithm provides mutual exclusion, since at all times at most one process holds the token. Raymond's algorithm also provides starvation-freeness, because eventually each ID in a queue moves to the head of this queue, and if a process wants the token, then the resulting chain of IDs in queues never contains a cycle.

14.3 Agrawal–El Abbadi Algorithm

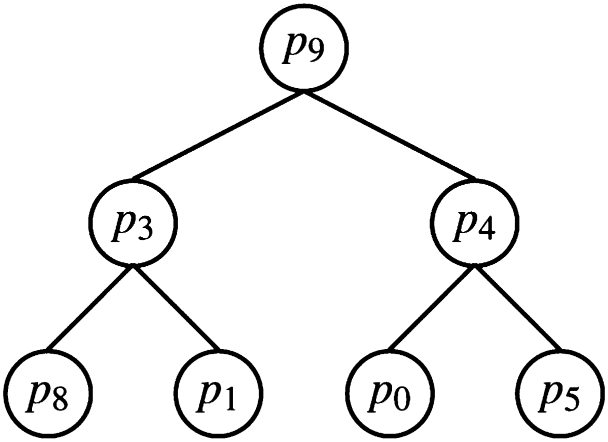
In the Agrawal–El Abbadi mutual exclusion algorithm, a process must obtain permission from a *quorum* of processes to enter its critical section. The crucial property of quorums is that each pair of quorums has a nonempty intersection. This guarantees mutual exclusion because different quorums cannot concurrently make different processes privileged. The crux of the Agrawal–El Abbadi algorithm is how quorums are defined. A strong point of this algorithm is that it can cope with processes that are not responsive or have crashed.

For simplicity, we assume that the network topology is complete and that $N = 2^{k+1} - 1$ for some $k > 0$. The processes are structured in a (full) binary tree of depth k, meaning that nodes at a depth $< k$ have exactly two children and the leaves are at depth k. A quorum consists of all processes on a path in this binary tree from the root to a leaf. If a nonleaf r is unresponsive, then instead of asking permission from r, permission can be asked from all processes on two paths: from each child of r to some leaf. We note that a process may have to ask permission from itself.

To be more precise, a process p that wants to enter its critical section places the root of the tree in a queue. Then it repeatedly tries to get permission from the process r at the head of its queue. If successful, r is removed from p's queue; if r is a nonleaf, one of the two children of r is appended at the end of p's queue. If a nonleaf r is found to be unresponsive, then r is removed from p's queue, and both of its children are appended at the end of the queue, in a fixed order, to avoid deadlocks; for instance, the left child is always placed before the right node. Otherwise two processes p and q could find that a nonleaf r is unresponsive, after which p and q might obtain permission from the left and right child of r, respectively, leading to a deadlock. If a leaf r is found to be unresponsive, p must abort this attempt to become privileged and retry. When p's queue becomes empty, it has received permission from a quorum of processes, so it can enter its critical section. After exiting its critical section, p informs each process in the quorum that its permission to p can be withdrawn.

If processes may crash, a complete and strongly accurate failure detector is required, and a process withdraws its permission if it detects that the process to which it has given permission has crashed.

Example 14.4 Let $N = 7$, and suppose the processes are structured in a binary tree as follows.



Some possible quorums are

- $\{p_9, p_3, p_8\}$, $\{p_9, p_3, p_1\}$, $\{p_9, p_4, p_0\}$, and $\{p_9, p_4, p_5\}$;

- if p_9 is not responding: $\{p_3, p_8, p_4, p_0\}$, $\{p_3, p_1, p_4, p_0\}$, $\{p_3, p_8, p_4, p_5\}$, and $\{p_3, p_1, p_4, p_5\}$;
- if p_3 is not responding: $\{p_9, p_8, p_1\}$;
- if p_4 is not responding: $\{p_9, p_0, p_5\}$.

For more possible quorums in this network, see exercise 14.9.

Suppose that two processes p and q concurrently want to enter their critical section; we consider one possible computation, with regard to the preceding binary tree. First, p obtains permission from p_9, and now it wants to obtain permission from p_4. Next, p_9 crashes, which is observed by q, which now wants to obtain permission from p_3 and p_4. Let q obtain permission from p_3, after which it appends p_8 at the end of its queue. Next, q obtains permission from p_4, after which it appends p_5 at the end of its queue. Next, p_4 crashes, which is observed by p, which now wants to obtain permission from p_0 and p_5. Next, q obtains permission from p_8, and now it wants to obtain permission from p_5. Finally, p obtains permission from both p_0 and p_5 and enters its critical section. It has obtained permission from the quorum $\{p_9, p_0, p_5\}$.

We argue, by induction on the depth of the binary tree, that each pair of quorums Q and Q' has a nonempty intersection. This implies that the Agrawal–El Abbadi algorithm guarantees mutual exclusion. A quorum that includes the root contains a quorum in one of the subtrees below the root, while a quorum without the root contains a quorum in both subtrees below the root. If Q and Q' both contain the root, then we are done, because the root is in their intersection. If both do not contain the root, then by induction they have elements in common in the two subtrees below the root. Finally, if Q contains the root while Q' does not, then Q contains a quorum in one of the subtrees below the root, and Q' also contains a quorum in this subtree. So, by induction, Q and Q' have an element in common in this subtree.

The Agrawal–El Abbadi algorithm is deadlock-free if all leaves are responsive. This property depends crucially on strict queue management. Suppose that in the case of an unresponsive process its left child is placed before its right child in the queue of a process that wants to become privileged. Consider the following total order $<$ on processes, based on their positions in the binary tree: $p < q$ either if p occurs at smaller depth than q in the binary tree or if p and q occur at the same depth and p is placed more to the left than q in the binary tree. Then a process that has obtained permission from a process q never needs permission from a process $p < q$ to enter its critical section. This guarantees that if all leaves are responsive, some process will always become privileged. Starvation may happen if a process fails infinitely often to get permission from a process in the binary tree.

14.4 Peterson's Algorithm

It was explained at the start of this chapter that mutual exclusion is of vital importance in a shared-memory setting in order to serialize access to a shared resource. In the remainder of this chapter, we focus on mutual exclusion for shared memory with processes $p_0, ..., p_{N-1}$. The algorithms we discuss are based on spinning, meaning that values of registers are read repeatedly until some condition is met.

Mutual Exclusion for Two Processes

Peterson's algorithm provides mutual exclusion for two processes, p_0 and p_1. The basic idea is that when a process p_b with $b \in \{0, 1\}$ wants to enter its critical section, it signals its intention to the process p_{1-b} by setting a Boolean flag to *true*. Next, p_b repeatedly checks whether p_{1-b}'s flag is set. As soon as this is not the case, p_b enters its critical section. When p_b exits its critical section, it resets its flag.

This simple mutual exclusion algorithm may suffer from livelock, meaning that the processes perform a continuous stream of events without making progress. This can happen if p_0 and p_1 concurrently set their flag and spin on the other's flag forever. To avoid livelock, Peterson's algorithm exploits a *wait* register. When p_b wants to enter its critical section, it not only sets its flag to *true* but also sets *wait* to b. If p_b now finds that *wait* = 1 - b, it can enter its critical section, even when the flag of p_{1-b} is set. In that case, p_0 and p_1 concurrently set their flag and p_{1-b} wrote to *wait* last, so it must wait.

To be more precise, Peterson's algorithm uses a multi-writer register *wait* with range {0, 1} and multi-reader/single-writer registers *flag*[b] of type Boolean for $b = 0, 1$; only p_b can write to *flag*[b]. Initially, *flag*[b] = *false*. When p_b wants to enter its critical section, it first sets *flag*[b] to *true* and then sets *wait* to b. Next, it spins on *flag*[1 - b] and *wait* until *flag*[1 - b] = *false* or *wait* = 1 - b, and then enters its critical section. When p_b exits its critical section, it sets *flag*[b] to *false*.

Example 14.5 We consider one possible execution of Peterson's algorithm. Initially, *flag*[b] = *false* for $b = 0, 1$.

- p_1 wants to enter its critical section and sets *flag*[1] to *true* and *wait* to 1.
- p_0 wants to enter its critical section and sets *flag*[0] to *true*.
- Since *flag*[0] = *true* and *wait* = 1, p_1 does not yet enter its critical section.
- p_0 sets *wait* to 0. Since *flag*[1] = *true* and *wait* = 0, p_0 does not yet enter its critical section.
- Since *wait* = 0, p_1 enters its critical section.
- p_1 exits its critical section and sets *flag*[1] to *false*.
- p_1 wants to enter its critical section and sets *flag*[1] to *true* and *wait* to 1. Since *flag*[0] = *true* and *wait* = 1, p_1 does not yet enter its critical section.
- Since *wait* = 1, p_0 enters its critical section.

We argue that Peterson's algorithm provides mutual exclusion. Suppose p_b is in its critical section (so it performed *flag*[b] ← *true* and *wait* ← b) and p_{1-b} tries to enter its critical section. There are two possibilities:

1. Before entering its critical section, p_b read *flag*[1-*b*] = *false*. Then p_{1-b} must set *flag*[1-*b*] to *true* and *wait* to 1 - *b* before it can enter its critical section.
2. Before entering its critical section, p_b read *wait* = 1 - *b*.

In both cases, *wait* has the value 1 - *b* by the time p_{1-b} starts spinning on *flag*[*b*] and *wait*. Since, moreover, *flag*[*b*] = *true*, p_{1-b} can enter its critical section only after p_b has set *flag*[*b*] to *false* or *wait* to *b*. Hence, p_{1-b} must wait until p_b is no longer in its critical section.

Peterson's algorithm is starvation-free. Let p_{1-b} try to enter its critical section. Then it sets *flag*[1 -*b*] to *true* and *wait* to 1 - *b*. Now p_b could only starve p_{1-b} by repeatedly trying to enter its critical section, because p_{1-b} should continuously read *flag*[*b*] = *true*. However, before (re)entering, p_b sets *wait* to *b*, after which p_{1-b} can enter its critical section.

Mutual Exclusion for More Than Two Processes

To obtain a mutual exclusion algorithm for $N > 2$ processes, we build a *tournament tree*, a binary tree of depth $k > 0$ in which each node represents an application of Peterson's algorithm for two processes. Initially, at most two processes are assigned to each of the 2^k leaves of the tournament tree; for simplicity, we assume that $N = 2^{k+1}$. A process that wants to enter its critical section performs Peterson's algorithm at its leaf. When a process becomes privileged at a nonroot, it proceeds to the parent of this node in the tournament tree. There it runs Peterson's algorithm again, where it may need to compete with the winner of the competition of the subtree below the other side of this node. A process that becomes privileged at the root in the tournament tree enters its critical section.

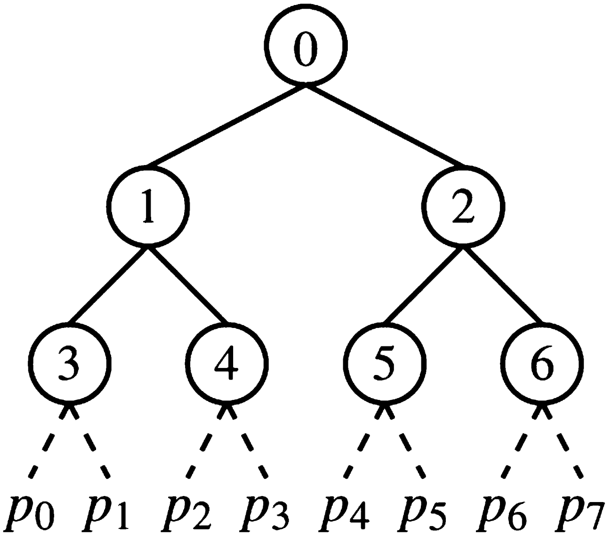
To be more precise, nodes in the tournament tree are numbered as follows: the root carries number 0, and given a node with the number *n*, its left and right children carry the numbers $2{\cdot}n + 1$ and $2{\cdot}n + 2$, respectively. To each node *n* we associate three multi-writer registers: $wait_n$ with range {0, 1} and $flag_n[b]$ of type Boolean for $b = 0, 1$. Initially, $flag_n[b] = false$. Each node has two sides, 0 and 1, and a process p_i that wants to enter its critical section is assigned to the leaf $(2^k - 1) + \lfloor i/2 \rfloor$ at the side *i* mod 2. A process at side *b* of a node *n* performs the following procedure *Peterson*(*n, b*):

```
$flag_n[b] \leftarrow true$;          $wait_n \leftarrow b$;
while $flag_n[1 - b] = true$ and $wait_n = b$ do
    {};
end while
if $n = 0$ then
    enter critical section;
    exit critical section;
else
```

```
        perform procedure $Peterson(\lceil n/2 \rceil - 1, (n + 1) \bmod 2)$;
    end if
    $flag_n[b] \leftarrow false$;
```

The first four lines are simply Peterson's algorithm at node *n*. The next six lines express that, at the root, becoming privileged means entering the critical section, while at a nonroot it means moving to the parent node, where Peterson's algorithm is run once again. The last line makes sure that when a process exits its critical section, its flags at all the nodes it visited are set back to *false*, thus releasing processes that may be waiting at these nodes.

Example 14.6 We consider one possible execution of Peterson's algorithm in a tournament tree. Let $N = 8$, so $k = 2$. Suppose that p_1 and p_6 both want to enter their critical section; p_1 starts at node 3, side 1, and p_6 at node 6, side 0.



Initially, $flag_n[b] = false$ for all *n* and *b*.

- p_6 executes $Peterson(6, 0)$. It sets $flag_6[0]$ to *true* and $wait_6$ to 0, and (since $flag_6[1] = false$) continues executing $Peterson(2, 1)$. It sets $flag_2[1]$ to *true* and $wait_2$ to 1, and (since $flag_2[0] = false$) continues executing $Peterson(0, 1)$. It sets $flag_0[1]$ to *true.*
- p_1 executes $Peterson(3, 1)$. It sets $flag_3[1]$ to *true* and $wait_3$ to 1, and (since $flag_3[0] = false$) continues executing $Peterson(1, 0)$. It sets $flag_1[0]$ to *true* and $wait_1$ to 0, and (since $flag_1[1] = false$) continues executing $Peterson(0, 0)$. It sets $flag_0[0]$ to *true* and $wait_0$ to 0. Since $flag_0[1] = true$ and $wait_0 = 0$, p_1 must wait.
- p_6 finally sets $wait_0$ to 1. Since $flag_0[0] = true$ and $wait_0 = 1$, p_6 must wait.

- Since $wait_0 = 1$, p_1 can enter its critical section.
- When p_1 exits its critical section, it sets $flag_0[0]$, $flag_1[0]$, and $flag_3[1]$ to *false*.
- Since $flag_0[0] = false$, p_6 can enter its critical section.

We argue that the tournament tree provides mutual exclusion. Suppose that a process p has moved from a node n, at a side b, to its critical section (if $n = 0$) or to the parent of n (if $n > 0$). We show by induction on $k - depth(n)$, where $depth(n)$ denotes the depth of n in the tournament tree, that no other process can move away from n until p has left its critical section. For the process at side $1 - b$ of n, this follows from the mutual exclusion property of Peterson's algorithm for two processes. Furthermore, if n is not a leaf, by induction, no process can move from the child of n at side b to n until p has left its critical section. Hence, at any time, and for any $\ell = 0, ..., k$, at most 2^{ℓ} processes can have progressed beyond a node at depth ℓ in the tournament tree. In particular, at any time, at most one process can be in its critical section, because it requires progressing beyond the root.

The tournament tree, moreover, is starvation-free. Suppose that a process p has arrived at a node n in the tournament tree. We argue by induction on the depth of n in the tournament tree that p will eventually enter its critical section. Suppose that p is at the moment stuck at n. Then some other process q entered its critical section (in the base case $n = 0$) or moved to the parent of n (in the inductive case $n > 0$), thereby blocking p; in the case $n > 0$, by induction, q will eventually enter its critical section. When q exits its critical section, it will set the flag at its side of n to *false*, after which p can enter its critical section (if $n = 0$) or move to the parent of n (if $n > 0$); in the latter case, by induction, p will eventually enter its critical section.

The tournament tree does not let processes enter their critical sections in a first-come, first-served manner. That is, we could consider as the doorway of a process wanting to enter its critical section its initial write operations to *flag* and *wait*, the first two assignments in the pseudocode of *Peterson*(n, b), at its starting leaf. If a process p exits its doorway before another process q enters its doorway at another leaf, it remains possible that q will become privileged before p.

14.5 Bakery Algorithm

The bakery algorithm enforces mutual exclusion similar to the way customers in a shop are served in a first-come, first-served fashion. When entering, say, a bakery, each customer gets a ticket with a number, which is increased by 1 with each ticket that follows. The waiting customer with the smallest number is the next to be served.

Tickets could in principle be modeled by means of a multi-writer register of type integer. However, avoiding having multiple processes concurrently read the value of this register and increase it by 1 requires the use of a read-modify-write operation (see section 14.7). In the bakery algorithm, a process instead reads the numbers of all other processes and selects the number that is the maximum of all those numbers plus 1. To break ties between different processes that concurrently select the same number, the ticket of a process p_i that selects a number k consists of the pair (k, i), and tickets are ordered lexicographically: $(k, i) < (\ell, j)$ if either $k < \ell$ or $k = \ell$ and $i < j$. A process can enter its critical section if its ticket is the smallest among all processes that have a number greater than zero. When a process exits its critical section, it sets its number back to zero.

To be more precise, the bakery algorithm uses multi-reader/single-writer registers $choosing_i$ of type Boolean and $number_i$ of type integer, for $i = 0, ..., N - 1$; only p_i can write to $choosing_i$ and $number_i$. Initially, $choosing_i = false$ and $number_i = 0$. A process p_i that wants to enter its critical section sets $choosing_i$ to *true*, reads the values of the registers $number_j$ for all $j \neq i$, writes $\max\{number_j \mid 0 \leq j < N\} + 1$ into $number_i$, and sets $choosing_i$ back to *false*. Next, for each $j \neq i$, p_i first spins on $choosing_j$ until it is *false* and then on $number_j$ until either $number_j = 0$ or $(number_i, i) < (number_j, j)$. After that, p_i can enter its critical section. When p_i exits its critical section, it sets $number_i$ to 0.

Example 14.7 We consider one possible execution of the bakery algorithm. Consider three processes p_0, p_1, and p_2; initially, $choosing_i = false$ and $number_i = 0$ for $i = 0, 1, 2$.

- p_1 wants to enter its critical section, sets $choosing_1$ to *true*, and reads $number_0$ and $number_2$.
- p_0 wants to enter its critical section, sets $choosing_0$ to *true*, and reads $number_1$ and $number_2$.
- p_0 sets $number_0$ to 1 and $choosing_0$ to *false*. It does not yet enter its critical section, because $choosing_1 = true$.
- p_1 sets $number_1$ to 1 and $choosing_1$ to *false*. It does not yet enter its critical section, because $(number_0, 0) < (number_1, 1)$ and $number_0 > 0$.
- Since $choosing_1 = choosing_2 = false$, $(number_0, 0) < (number_1, 1)$, and $number_2 = 0$, p_0 enters its critical section.
- p_0 exits its critical section and sets $number_0$ to 0.
- p_0 wants to enter its critical section, sets $choosing_0$ to *true*, and reads $number_1$ and $number_2$.
- p_0 sets $number_0$ to 2 and $choosing_0$ to *false*. It does not yet enter its critical section, because $(number_1, 1) < (number_0, 0)$ and $number_1 > 0$.
- Since $choosing_0 = choosing_2 = false$, $number_1 < number_0$, and $number_2 = 0$, p_1 enters its critical section.

The bakery algorithm provides mutual exclusion. Suppose that p_i is in its critical section. Then clearly $number_i > 0$. Moreover, let $number_j > 0$ for some $j \neq i$. We argue that then $(number_i, i) < (number_j, j)$. Before p_i entered its critical section, it must have read $choosing_j = false$, and either $number_j = 0$ or $(number_i, i) < (number_j, j)$. If p_j chooses a new ticket while p_i is in its critical section, then p_j is guaranteed to take $number_i$ into account and therefore will choose a larger number. This concludes the argument. Since a privileged process always carries a positive number, and has the smallest ticket among all processes with a positive number, only one process can be in its critical section at any time.

The bakery algorithm is starvation-free. If a process p_i wants to enter its critical section and other processes keep on entering and exiting their critical section, then eventually p_i will have the smallest ticket. That is, eventually each process that wants to enter its critical section will take the current value of $number_i$ into account and will choose a number larger than $number_i$ for its own ticket.

Let us say that the doorway of a process p_i that wants to enter its critical section consists of the sequence of events from setting $choosing_i$ to *true* up to and including setting $choosing_i$ back to *false*. The bakery algorithm treats processes in a first-come, first-served fashion, in the sense that if a process p exits its doorway before another process q enters its doorway, then p will enter its critical section before q.

The values of the *number* fields grow without bound. The bakery algorithm can be adapted such that these values are limited to a finite range; see exercise 14.17.

In the bakery algorithm, a process that wants to enter its critical section reads the values of two registers at every other process. This renders the algorithm impractical in the case of a large number of processes. The following theorem states that in principle it is impossible to solve mutual exclusion for N processes with fewer than N registers by using only atomic read and write operations.

Theorem 14.1 *At least N registers are needed to solve livelock-free mutual exclusion for N processes if only atomic read and write operations are employed.*

Proof. We sketch a proof for the case $N = 2$; the general case is similar. Suppose, toward a contradiction, that a livelock-free mutual exclusion algorithm for two processes p and q uses only one multi-writer register R.

Before p can enter its critical section, it must write to R, for otherwise q would not be able to recognize whether p is in its critical section. Likewise, before q can enter its critical section, it must write to R. Because of livelock-freeness, we can bring p and q into a position where they are both about to write to R, after which each will enter its critical section.

Suppose without loss of generality that p writes to R first and enters its critical section. The subsequent write by q obliterates the value p wrote to R, so q cannot tell that p is in its critical section. Consequently q will also enter its critical section. This contradicts the mutual exclusion property. □

14.6 Fischer's Algorithm

Fischer's algorithm circumvents the impossibility result at the end of the previous section by means of time delays.

The multi-writer register *turn* ranges over $\{-1, 0, ..., N - 1\}$. Initially, it has the value -1. A process p_i that wants to enter its critical section spins on *turn* until its value is - 1. Within one time unit of this read, p_i sets the value of *turn* to i. Next, p_i waits for more than one time unit and then reads *turn*. If p_i still has the value i, then it enters its critical section. Otherwise p_i returns to spinning on *turn* until its value is - 1. When a process exits its critical section, it sets the value of *turn* to - 1.

Example 14.8 We consider a possible execution of Fischer's algorithm. Consider three processes p_0, p_1, and p_2, with p_0 in its critical section, so *turn* = 0. Processes p_1 and p_2 both want to enter their critical section and are spinning on *turn*.

When p_0 exits its critical section, it sets *turn* to - 1. Now p_1 and p_2 concurrently read that *turn* = -1. First, p_1 sets the value of *turn* to 1, and less than one time unit later, p_2 sets its value to 2. More than one time unit after it performed its write, p_1 reads *turn*, finds that its value was changed to 2, and returns to spinning on *turn*. On the other hand, more than one time unit after it performed its write, p_2 reads *turn*, finds that its value is still 2, and enters its critical section.

We argue that Fischer's algorithm guarantees mutual exclusion. When *turn* = -1, clearly no process is in its critical section. And when a process p_i sets the value of *turn* to i, other processes p_j can only set the value of *turn* to $j \neq i$ within one time unit. So if the value of *turn* remains i for more than one time unit, p_i can be certain that no other process can become privileged.

Fischer's algorithm is livelock-free: when the value of *turn* becomes - 1, a process p_i that wants to enter its critical section can freely write the value i in *turn*. The last process to set the value of *turn* within one time unit of the first write will enter its critical section. However, there can be starvation, if a process p_i wants to enter its critical section, infinitely often *turn* is set to - 1 and p_i writes i in *turn*, but every time this value is overwritten by a $j \neq i$ within one time unit.

A strong requirement of this algorithm is the presence of a global clock. Another drawback is that processes all spin on the same register *turn*. Why this can be problematic will be explained in the next section.

14.7 Test-and-Test-and-Set Lock

Test-and-Set Lock

The test-and-set lock circumvents the impossibility result from section 14.5 by using a read-modify-write operation: *test-and-set*. This lock uses one Boolean multi-writer register *locked*, which initially holds the value *false*. A process that wants to acquire the lock repeatedly applies *test-and-set* to *locked*. This operation sets the value of *locked* to *true* and returns the previous value of *locked*. The process obtains the lock (in other words, it becomes privileged) as soon as *false* is returned by a *test-and-set* operation. To unlock, the process sets *locked* to *false*.

The test-and-set lock provides mutual exclusion. When *locked* contains *false*, clearly no process holds the lock. And when a process *p* acquires the lock, meaning that it applies a *test-and-set* that turns the value of *locked* from *false* to *true*, then no other process can acquire the lock until *p* unlocks by setting *locked* to *false*. The test-and-set lock, moreover, is livelock-free but not starvation-free.

The test-and-set lock, although conceptually simple, tends to have a poor performance. The reason is that each *test-and-set* operation on *locked* comes with a memory barrier: it invalidates all cached values of *locked*. As a result, all processes that want to acquire the lock take a cache miss and fetch the (mostly unchanged) value from main memory. Thus, the processes performing *test-and-set* operations on *locked* cause a continuous storm of unnecessary messages over the bus.

Test-and-Test-and-Set Lock

The test-and-test-and-set lock improves on the test-and-set lock by letting processes that want to acquire the lock spin on a cached copy of the Boolean register *locked*. When *false* is returned, the process applies *test-and-set* to *locked* itself. The process obtains the lock if *false* is returned; otherwise it goes back to spinning on its cached copy of *locked*. To unlock, the process sets *locked* to *false*.

The test-and-test-and-set lock provides mutual exclusion and livelock-freeness. It avoids a considerable part of the bus traffic of the test-and-set lock and therefore tends to have a much better performance. Still, the test-and-test-and-set lock generates unnecessary bus traffic but only when the lock is released. Then an instance of what is referred to as the thundering herd problem arises, where all waiting processes simultaneously try to acquire the lock. Since *false* is written in *locked*, they all invalidate the corresponding cache line and then go to the bus to fetch the value of *locked*. Then they concurrently perform *test-and-set* to try to acquire the lock, invalidating the cached copies at other processes and thus leading to another round of cache misses. Finally, the storm subsides and processes return to local spinning on their cached copy of *locked*.

The performance of the test-and-test-and-set lock can be improved by applying *exponential back-off* to reduce contention. The idea is that when a process applies *test-and-set* to *locked* but fails to get the lock, it backs off for a certain amount of time to avoid collisions. Each subsequent failure to get the lock by means of a *test-and-set* is interpreted as a sign that there is a high contention for the lock. Therefore, the waiting time is doubled at each failed attempt, up to some maximum. Two important parameters for the performance of the lock are the initial minimum delay and the maximum delay; optimal values for these parameters are platform dependent. Waiting durations are randomized to avoid having competing processes fall into lockstep.

Example 14.9 Consider three processes p_0, p_1, and p_2 that all want to acquire the test-and-test-and-set lock. Initially, the Boolean register *locked* is *false.*

Processes p_0, p_1, and p_2 concurrently read that (their cached copy of) *locked* is *false.* Let p_1 apply *test-and-set* to *locked* first, setting it to *true.* Since this operation returns *false*, p_1 takes the lock. Next, p_0 and p_2 apply *test-and-set* to *locked.* In both cases, this operation returns *true*, so p_0 and p_1 back off for (a randomization of) the minimum delay. After this delay, p_0 and p_1 start spinning on their cached copy of *locked.*

When p_1 releases the lock, it sets *locked* to *false.* Now p_0 and p_2 concurrently read that the value has changed and apply *test-and-set* to *locked.* Let p_2 be the first to do so, setting its value to *true.* Since this operation returns *false*, p_2 takes the lock. The *test-and-set* by p_0 returns *true*, after which p_0 backs off for twice the minimum delay. After this delay, p_0 returns to spinning on its cached copy of *locked.*

The test-and-test-and-set lock with exponential back-off is easy to implement and can give excellent performance in the case of low contention. However, it may suffer from starvation, processes may be delayed longer than necessary because of back-off, and last but not least, all processes still spin on the same register *locked*, which creates a bottleneck and generates bus traffic, especially in the case of high contention.

14.8 Queue Locks

Queue locks overcome the drawbacks of the test-and-test-and-set lock by placing processes that want to acquire the lock in a queue. A process p in the queue spins on a register to check whether its predecessor in the queue has released the lock. When this is the case, p takes the lock. Key to the success of queue locks is that all processes in the queue spin on a different register. Queue locks provide mutual exclusion because only the head of the queue holds the lock. Moreover, processes are treated in a first-come, first-served manner: the sooner a process is added to the queue, the earlier it is served.

Anderson's Lock

Anderson's lock places processes that want to acquire the lock in a queue by means of a Boolean array of size n. Here n is the maximum number of processes that can concurrently compete for the lock (so $n \leq N$). A counter is used to assign a slot in the array to every process that wants to acquire the lock; this counter is interpreted modulo n. At any moment, at most one process is assigned to each slot in the array, and at most one slot in the array holds *true*; the process that is assigned to this slot holds the lock. The slots in the array and the counter are multi-writer registers. Initially, slot 0 of the array holds *true*, slots $1, \dots, n-1$ hold *false*, and the counter is zero.

A process p that wants to acquire the lock applies the read-modify-write operation *get-and-increment* to the counter, which increases the counter by 1 and returns the previous value of the counter in one atomic step. The returned value modulo n is the slot in the array that is assigned to the process. Next, p spins on (a cached copy of) this slot in the array until it holds *true*, at which moment p acquires the lock. To unlock, p first sets its slot in the array to *false* and then the next slot modulo n to *true*, signaling to its successor (if any) that it can take the lock.

Example 14.10 Let $N = n = 3$, and suppose processes p, q, and r all want to acquire Anderson's lock. Initially, only slot 0 in the array holds *true* and the counter is 0.

- q applies *get-and-increment* to the counter, increasing it to 1. Since this operation returns 0 and slot 0 holds *true*, q takes the lock.
- p applies *get-and-increment* to the counter, increasing it to 2. Since this operation returns 1 and slot 1 holds *false*, p starts spinning on this slot.
- r applies *get-and-increment* to the counter, increasing it to 3. Since this operation returns 2 and slot 2 holds *false*, r starts spinning on this slot.
- When q releases the lock, it sets slot 0 to *false* and slot 1 to *true*.
- p reads that the value of slot 1 has changed to *true* and takes the lock.
- q wants to acquire the lock again. It applies *get-and-increment* to the counter, increasing it to 4. Since this operation returns 3 and slot 3 mod 3 = 0 holds *false*, q starts spinning on slot 0.

Anderson's lock resolves the weaknesses of the test-and-test-and-set lock. In particular, different processes spin on different registers. However, a risk is the occurrence of what is called false sharing: different slots in the array may be kept on a single cache line, being the smallest unit of memory to be transferred between main memory and a cache. When a data item in the cache becomes invalid, the entire cache line where the data item is kept is invalidated. So if different slots of the array are kept on the same cache line, releasing the lock still gives rise to unnecessary bus traffic. This is mitigated by padding; the array size is, say, quadrupled, and slots are separated by three unused places in the array.

Another drawback of Anderson's lock is that it requires an array of size n (or more, in the case of padding), even when no process wants the lock. In a setting with a large number of processes and multiple locks, this memory overhead can become costly.

CLH Lock

The CLH lock does not use a fixed array. Instead, the queue of processes that are waiting for the lock is maintained by means of a dynamic list structure. A process that wants to acquire the lock places an element ε in the list, containing a Boolean multi-reader/single-writer register $active_\varepsilon$, which becomes *false* after the process has released the lock. Moreover, a multi-writer register *last* points to the most recently added element in the list. Initially, *last* points to a dummy element in which the *active* field is *false*, indicating that the lock is free.

A process p wanting to acquire the lock creates an element ε with $active_\varepsilon = true$. It applies *get-and-set*(ε) to *last* to make ε the last element in the queue and get a pointer to the element of its predecessor in the queue. Next, p spins on (a cached copy of) the *active* field in its predecessor's element in the list until it becomes *false*. When this happens, p takes the lock. To unlock, p sets $active_\varepsilon$ to *false*, signaling to its successor (if any) that it can take the lock; p can reuse the element of its predecessor for a future lock access (but not its own element ε; see exercise 14.23).

We note that the queue is virtual: a process p that is waiting for the lock knows the element of its predecessor q in the queue, but there is no explicit pointer between the elements of p and q.

Example 14.11 Processes p_0 and p_1 want to acquire the CLH lock; they create elements ε_0 and ε_1, respectively, with $active_{\varepsilon_0} = active_{\varepsilon_1} = true$. Initially, *last* points to a dummy element in which the *active* field is *false*.

- p_1 applies *get-and-set*(ε_1) to *last* to let it point to ε_1. Since this operation returns the dummy element that contains *false*, p_1 takes the lock.
- p_0 applies *get-and-set*(ε_0) to *last* to let it point to ε_0. Since this operation returns ε_1, which contains *true*, p_0 starts spinning on a cached copy of $active_{\varepsilon_1}$.
- When p_1 releases the lock, it sets $active_{\varepsilon_1}$ to *false*.
- p_0 finds that the value of $active_{\varepsilon_1}$ has changed to *false* and takes the lock.

The CLH lock exhibits the same good performance as Anderson's lock and uses space more sparingly. The Achilles' heel of the CLH lock is that because of remote spinning, on the *active* field in the predecessor's element in the list, its performance is heavily dependent on the presence of caches.

MCS Lock

The MCS lock avoids remote spinning; instead, a process q waiting for the lock spins on a Boolean *wait* field in its own element. To achieve this, q must inform its predecessor p in the queue that q is its successor, so that after p releases the lock, it will invert the *wait* field in q's element. The price to pay is a more involved and expensive unlock procedure to deal with the case where q joins the queue before p releases the lock but informs p that q is its successor after p has released the lock.

Again, processes that want the lock place an element ε in the list that contains two multi-writer registers: the Boolean $wait_\varepsilon$ is *true* while the process is waiting in the queue, and $succ_\varepsilon$ points to the successor element in the (this time physical) queue or is a `null` pointer if ε is the last element in the queue. Moreover, the multi-writer register *last* points to the last element in the queue. Initially, *last* = `null`.

A process p that wants to acquire the lock creates an element ε with $wait_\varepsilon = false$ and $succ_\varepsilon$ = `null`. It applies *get-and-set*(ε) to *last* to make ε the last element in the queue and get a pointer to the element of its predecessor in the queue. If *last* contained `null`, then p takes the lock immediately. Otherwise, p first sets $wait_\varepsilon$ to *true* and then lets the *succ* field in the element of its predecessor point to ε. Next, p spins on $wait_\varepsilon$ until it becomes *false*. When this is the case, p can take the lock.

When p releases the lock, it checks whether $succ_\varepsilon$ points to another element. If so, p sets the *wait* field in the latter element to *false*, signaling to its successor that it can take the lock. If, on the other hand, $succ_\varepsilon$ = `null`, then p applies *compare-and-set*(ε, `null`) to *last*, signaling that the queue has become empty. If this operation fails, meaning it returns *false*, then in the meantime another process q that joined the queue has written to *last*. In that case, p starts spinning on $succ_\varepsilon$ until an element is returned, which is the element of q. Finally, p sets the *wait* field in q's element to *false*. After releasing the lock, p can reuse ε for a future lock access.

Example 14.12 Processes p_0 and p_1 want to acquire the MCS lock; they create elements ε_0 and ε_1, respectively, containing *false* and `null`. Initially, *last* = `null`.

- p_1 applies *get-and-set*(ε_1) to *last* to let it point to ε_1. Since this operation returns `null`, p_1 takes the lock.
- p_0 applies *get-and-set*(ε_0) to *last* to let it point to ε_0. Since this operation returns ε_1, p_0 sets $wait_{\varepsilon_0}$ to *true*, lets $succ_{\varepsilon_1}$ point to ε_0, and starts spinning on $wait_{\varepsilon_0}$.
- When p_1 releases the lock, it finds that $succ_{\varepsilon_1}$ points to ε_0. Therefore, p_1 sets $wait_{\varepsilon_0}$ to *false*.
- p_0 reads that the value of $wait_{\varepsilon_0}$ has changed to *false* and takes the lock.

Example 14.13 Processes p_0 and p_1 want to acquire the MCS lock; they create elements ε_0 and ε_1, respectively, containing *false* and `null`. Initially, *last* = `null`.

- p_1 applies *get-and-set*(ε_1) to *last* to let it point to ε_1. Since this operation returns `null`, p_1 takes the lock.
- p_0 applies *get-and-set*(ε_0) to *last* to let it point to ε_0. This operation returns ε_1.
- When p_1 releases the lock, it finds that $succ_{\varepsilon_1}$ = `null`. Therefore, it applies *compare-and-set*(ε_1, `null`) to *last*. Since *last* points to ε_0, this operation returns *false*. Therefore, p_1 starts spinning on $succ_{\varepsilon_1}$.
- p_0 sets $wait_{\varepsilon_0}$ to *true*, lets $succ_{\varepsilon_1}$ point to ε_0, and starts spinning on $wait_{\varepsilon_0}$.
- p_1 finds that $succ_{\varepsilon_1}$ points to ε_0, and it sets $wait_{\varepsilon_0}$ to *false*.
- p_0 reads that the value of $wait_{\varepsilon_0}$ has changed to *false* and takes the lock.

The MCS lock tends to outperform the CLH lock on what are called cacheless NUMA (Non-Uniform Memory Access) architectures, in which each processor is provided with its own memory unit instead of having one shared-memory unit.

Timeouts

With queue locks, a process p in the queue cannot easily give up its attempt to acquire the lock, because its successor in the queue depends on p. We now explain how the CLH lock can be adapted to include such timeouts. The key is that p needs to tell its successor to start spinning on the element of p's predecessor in the queue.

Again, a process p that wants to acquire the lock places an element ε in the list, with a multi-reader/single-writer register $pred_\varepsilon$ that can have the following values:

- `null` if p is waiting in the queue or is in its critical section;
- a pointer to the element of p's predecessor in the queue if p has given up waiting for the lock; or
- a pointer to a special element called `released` if p has left its critical section.

The multi-writer register *last* points to the last element in the queue. Initially, *last* = `null`.

When p wants to acquire the lock, it creates an element ε with $pred_\varepsilon$ = `null`. It applies *get-and-set*(ε) to *last* to make ε the last element in the queue and get a pointer to the element of its predecessor. If *last* contained `null`, then p takes the lock immediately. Otherwise it spins on (a cached copy of) the *pred* field in its predecessor's element until it is not `null`. If that *pred* field then points to the element `released`, p takes the lock. Otherwise it points to the element ε' of the new predecessor of p (meaning that p's original predecessor has timed out). In that case, p continues to spin on $pred_{\varepsilon'}$ until it is not `null`.

If p quits its attempt to acquire the lock, it applies *compare-and-set*(ε, $pred_\varepsilon$) to *last*. If this operation fails, then p has a successor in the queue; in that case, p sets $pred_\varepsilon$ to the element of its predecessor, signaling to the successor of p that it has a new predecessor.

When p releases the lock, it applies *compare-and-set*(ε, `null`) to *last*. If this operation succeeds, then the queue has become empty. If it fails, then p has a successor in the queue. In the latter case, p sets $pred_\varepsilon$ to the element `released`, signaling to its successor that it can take the lock.

Example 14.14 Processes p_0, p_1, and p_2 want to acquire the CLH lock with timeouts; they create elements ε_0, ε_1, and ε_2, respectively, containing `null`. Initially, *last* = `null`.

- p_1 applies *get-and-set*(ε_1) to *last* to let it point to ε_1. Since this operation returns `null`, p_1 takes the lock.
- p_0 applies *get-and-set*(ε_0) to *last* to let it point to ε_0. Since this operation returns ε_1, p_0 starts spinning on $pred_{\varepsilon_1}$.
- p_2 applies *get-and-set*(ε_2) to *last* to let it point to ε_2. Since this operation returns ε_0, p_2 starts spinning on $pred_{\varepsilon_0}$.
- p_0 decides to abort its attempt to acquire the lock. It lets $pred_{\varepsilon_0}$ point to ε_1.
- p_2 finds that $pred_{\varepsilon_0}$ has changed from `null` to ε_1 and starts spinning on $pred_{\varepsilon_1}$.
- When p_1 releases the lock, it applies *compare-and-set*(ε_1, `null`) to *last*. This

operation fails because *last* points to ε_2 instead of ε_1. Therefore, p_1 lets $pred_{\varepsilon_1}$ point to `released`.

- p_2 finds that $pred_{\varepsilon_1}$ has changed from `null` to `released` and takes the lock.

Bibliographical notes

The Ricart-Agrawala algorithm was presented in [80], and the Carvalho-Roucairol optimization was proposed in [16]. Raymond's algorithm originates from [79], and the Agrawal–El Abbadi algorithm comes from [2]. Peterson's mutual exclusion algorithm for two processes stems from [72], and the approach to extend such an algorithm to general N via a tournament tree was proposed in [74]. The bakery algorithm originates from [53]; it was proved in [15] that such an algorithm requires N atomic registers. Fischer's algorithm was proposed in an email by Michael J. Fischer in 1985 and was presented in [55]. The test-and-test-and-set lock is due to [50]. Anderson's lock stems from [3], the CLH lock comes from [25, 61], the MCS lock is taken from [67], and the CLH lock with timeouts comes from [82].

14.9 Exercises

Exercise 14.1 Say for both mutual exclusion and starvation-freeness whether it is a safety or liveness property.

Exercise 14.2 Consider the Ricart-Agrawala algorithm with the Carvalho-Roucairol optimization. Processes p_0 and p_1 enter and exit their critical section and both want to become privileged again. Give two possible computations, one where p_1 does not ask permission from p_0 to enter its critical section and one where p_0 and p_1 ask permission from each other before entering their critical section for the second time.

Exercise 14.3 Give an example to show that with the Carvalho-Roucairol optimization a process can become privileged while its request is larger than some other request in the network.

Exercise 14.4 Show that if processes could apply the Carvalho-Roucairol optimization from the start (instead of after the first entry of their critical section), then the resulting mutual exclusion algorithm would be incorrect.

Exercise 14.5 The logical clock values in the Ricart-Agrawala algorithm are unbounded. Adapt the algorithm so that the range of these values becomes finite (using modulo arithmetic).

Exercise 14.6 Run Raymond's algorithm on the network from example 14.3. Initially, process p_3 holds the token and all buffers are empty. Give a computation (including all messages) in which first p_8, then p_2, and finally p_5 requests the token, but they receive the token in the opposite order.

Exercise 14.7 Argue that in Raymond's algorithm each request to enter a critical section gives rise to at most $2D$ messages.

Exercise 14.8 Explain in detail why Raymond's algorithm is starvation-free.

Exercise 14.9 Consider the Agrawal–El Abbadi algorithm with seven processes structured in a binary tree as in example 14.4. What are the quorums if p_9 and p_4 crashed? If p_9, p_3, and p_4 crashed? If p_9, p_3, and p_8 crashed?

Exercise 14.10 In the Agrawal–El Abbadi algorithm, what are the minimum and maximum sizes of a quorum (in terms of N)?

Exercise 14.11 Prove that for each pair of quorums Q and Q' in the Agrawal–El Abbadi algorithm, $Q \subseteq Q'$ implies $Q = Q'$.

Exercise 14.12 Explain why the following mutual exclusion algorithm is flawed. Let *flag* be a multi-writer Boolean register. A process p wanting to enter its critical section waits until *flag* = *false*. Then p performs *flag* ← *true* and becomes privileged. When p exits its critical section, it performs *flag* ← *false*.

Exercise 14.13 *2-mutual exclusion* is satisfied if at any time at most two processes are in their critical section. Modify the tournament tree (in which the nodes run Peterson's algorithm) to yield a solution for the 2-mutual exclusion problem.

Exercise 14.14 Present a starvation-free 2-mutual exclusion algorithm using one register and a read-modify-write operation.

Exercise 14.15 Suppose that in the bakery algorithm a process could enter its critical section without waiting for all *choosing* registers to become *false*. Give an example to show that then mutual exclusion is no longer guaranteed.

Exercise 14.16 Describe an execution of the bakery algorithm in which the values of *number* registers grow without bound.

Exercise 14.17 Adapt the bakery algorithm such that the range of the *number* registers becomes finite.

Exercise 14.18 Argue the correctness of theorem 14.1 for the case $N = 3$.

Exercise 14.19 Explain why the proof of theorem 14.1 does not apply to Fischer's algorithm.

Exercise 14.20 Give an example of starvation with Fischer's algorithm.

Exercise 14.21 Argue in detail that Anderson's lock provides mutual exclusion and first-come, first-served fairness.

Exercise 14.22 Give an example (with $n = 3$) to show what could go wrong if in Anderson's lock a process that releases the lock first set the next slot modulo n in the array to *true* and only then set its own slot to *false*.

Exercise 14.23 [46] Suppose that in the CLH lock, a process would reuse its own element (instead of the element of its predecessor). Give an execution to show that then the algorithm would be flawed.

Exercise 14.24 For each of the two read-modify-write operations in the MCS lock, replace this operation by read and write operations and give an execution to show that the resulting lock is flawed. In both cases, explain which property is violated.

Exercise 14.25 Argue in detail that (the unlock procedure of) the MCS lock does not suffer from deadlock.

Exercise 14.26 Consider for the MCS lock the situation where a process p wants to acquire the lock and finds that *last* points to an element. Suppose that p would first set the *succ* field of that element to its own element and only then set the *wait* field in its own element to *true*. What could go wrong?

Exercise 14.27 Develop a variant of the MCS lock that includes timeouts, allowing a process to abandon its attempt to obtain the lock.

15
Barriers

Sometimes during a computation processes must collectively wait at some point, called a *barrier*, until all processes have arrived there, after which they can leave the barrier and resume execution. For example, in parallel programs, a barrier is often used to make sure that all threads have completed some parallel loop before the computation proceeds any further. Or a barrier may be employed in a soft real-time application, in which a number of subtasks must be completed by different processes before the overall application can proceed. A barrier generally keeps track of how many processes have reached it and signals to all waiting processes when the last process has reached the barrier. We discuss several barrier algorithms in a shared-memory setting. Waiting at a barrier resembles waiting to enter a critical section. Waiting processes can either spin on local or (locally cached copies of) remote variables or fall asleep when the barrier is reached and be woken up when all processes have reached the barrier.

15.1 Sense-Reversing Barrier

A straightforward way of implementing a barrier is to maintain a counter, a multi-writer register, with initial value 0. Each process that reaches the barrier performs *get-and-increment* on the counter; a read-modify-write operation is needed because otherwise multiple processes could concurrently increase the counter to the same value. When the counter equals the number N of processes that must reach the barrier, all processes can leave the barrier.

If a process reaches the barrier and applies a *get-and-increment* that returns a value smaller than $N - 1$, then it can fall asleep. The last process to reach the barrier, for which *get-and-increment* returns $N - 1$, first resets the value of the counter to 0, so that it can be reused, and then wakes up all other processes. A drawback of this approach is that the waking-up phase can be time-consuming.

A better idea may be to use a global Boolean sense field, a multi-writer register, which initially is *false*. Moreover, each process carries a local Boolean sense field, a single-reader/single-writer register, which initially is *true*. A process p that reaches the barrier applies *get-and-increment* to the counter. If p is not the last to reach the barrier, meaning that *get-and-increment* returns a value smaller than $N - 1$, it starts spinning on the barrier's global sense field until it equals p's local sense, after which p can leave the barrier. On the other hand, suppose p is the last to reach the barrier, meaning that *get-and-increment* returns $N - 1$. Then p first resets the counter to 0, so that it can be reused, and then reverses the value of the global sense field, signaling to the other processes that they can leave the barrier. Processes resume execution with reversed local sense, so that not only the counter but also the (global and local) sense fields can be reused for the next barrier.

Example 15.1 We study one possible execution of the sense-reversing barrier. Consider three processes p, q, and r, initially with local sense *true*. The barrier's counter initially has the value 0, and its global sense is *false*.

- Process q reaches the barrier and applies *get-and-increment* to the counter, which returns 0. Therefore, q starts spinning on the global sense field until it is *true*.
- Process p reaches the barrier and applies *get-and-increment* to the counter, which returns 1. Therefore, p starts spinning on the global sense field until it is *true*.
- Process r reaches the barrier and applies *get-and-increment* to the counter, which returns 2. Therefore, r resets the value of the counter to 0, reverses the global sense field to *true*, and leaves the barrier with reversed local sense *false*.
- Processes p and q notice that the global sense of the barrier has become *true* and also leave the barrier with reversed local sense *false*.

The main drawback of the sense-reversing barrier is, similar to the situation with the test-and-test-and-set lock, that processes that have arrived at the barrier are all spinning on (a cached copy of) the same global sense field.

15.2 Combining Tree Barrier

The combining tree barrier uses a tree structure to reduce contention on the global sense field. Each node represents a sense-reversing barrier; processes that are waiting at the barrier are spinning on the global sense field of a node.

Let the tree have depth k, and let each node at a depth smaller than k have r children. The corresponding combining tree barrier can cope with r^{k+1} processes: to each leaf, we assign at most r processes. For simplicity, we assume that $N = r^{k+1}$. Each node maintains a counter and a global sense field. At a leaf, the counter keeps track of how many of the processes assigned to this leaf have reached the barrier, while at a nonleaf it keeps track of the number of children of this node for which the counter has become r. As soon as the counter of a nonroot becomes r, the counter of the parent of this node is increased by 1. When finally the counter at the root of the tree becomes r, we can be certain that all processes have reached the barrier. Then the counters are reset and the global sense fields are reversed at all the nodes, from top (the root) to bottom (the leaves), after which all processes resume execution.

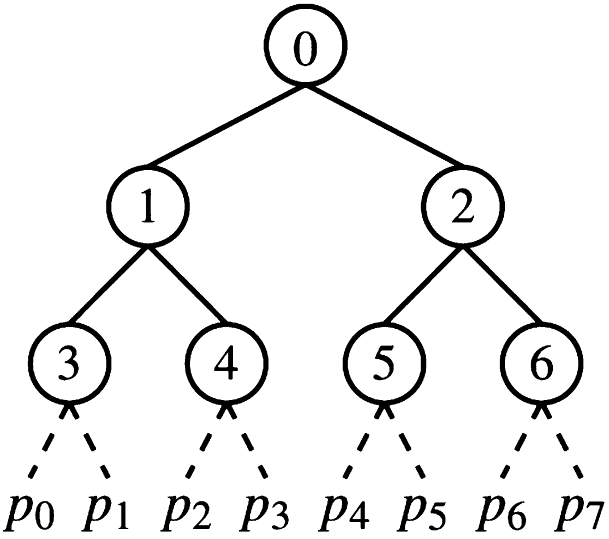
To be more precise, the nodes in the tree are numbered as follows: the root carries number 0, and given a nonleaf with the number n, its children carry the numbers $rn + 1$ up to $rn + r$. To each node n we associate two multi-writer registers: $count_n$ of type integer and $gsense_n$ of type Boolean. We assume that there are processes $p_0, ..., p_{N-1}$. Each process p_i maintains a single-reader/single-writer local sense field $lsense_i$. Initially, $count_n = 0$, $gsense_n = false$, and $lsense_i = true$. Process p_i is assigned to leaf $r^{k-1} + r^{k-2} + \cdots + 1 + \lfloor i/r \rfloor$ in the tree. That is, when p_i reaches the barrier, it performs the following procedure *CombiningTree*(n) with $n = r^{k-1} + r^{k-2} + \cdots + 1 + \lfloor i/r \rfloor$:

```
if $counter_n.get\text{-}and\text{-}increment < r - 1$ then
    while $gsense_n \neq lsense_i$ do
        {};
    end while
else
    if $n > 0$ then
        perform procedure Combining Tree $\left(\left\lceil \frac{n}{r} \right\rceil - 1\right)$;
    end if
    $count_n \leftarrow 0$; $gsense_n \leftarrow lsense_i$;
end if
```

A process p_i performing *CombiningTree*(n) first applies *get-and-increment* to the counter at node n. If the counter is increased to a value smaller than r, then p_i starts spinning on the global sense field of n until it equals p_i's local sense. On the other hand, if p_i increases the counter at n to r, then we distinguish the case where n is a nonroot from the case where n is the root of the tree. If n is a nonroot, then p_i moves to the parent of n in the tree, where it performs *CombiningTree* again. If n is the root of the tree, then all processes have reached the barrier. In this case, p_i resets the counter at the root to 0, so that it can be reused, and next reverses the value of the global sense field at the root, signaling to the $r - 1$ processes spinning on this field that they can leave the barrier.

Processes that find the global sense field they are spinning on reversed, and the process that reverses the global sense at the root, reset the counter and reverse the global sense field at all the nodes they visited earlier, signaling to the $r - 1$ processes spinning on such a field that they can leave the barrier. Processes resume execution with reversed local sense so that the counters and sense fields can be reused for the next barrier.

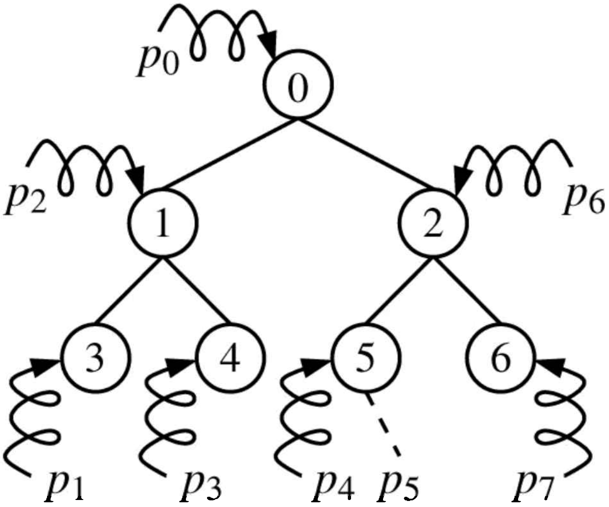
Example 15.2 We consider one possible execution of the combining tree barrier. Let $k = 2$, $r = 2$, and $N = 8$. The processes p_0 to p_7 are assigned to the leaves of the tree as shown in the accompanying picture. Initially, the counters at the nodes are 0, the global sense fields at the nodes are *false*, and all processes have local sense *true*.



- Let p_0, p_2, p_4, and p_7 arrive at the barrier. Each applies *get-and-increment* to the counter of its leaf, increasing it to 1. Next, each starts spinning on the global sense field of its leaf until it becomes *true*.
- Let p_1 and p_6 arrive at the barrier. They apply *get-and-increment* to the counters of leaf 3 and leaf 6, respectively, increasing them to 2. Next, they move to nodes 1 and 2, respectively, where they apply *get-and-increment* to the counters, increasing them to 1. They start spinning on the global sense fields of nodes 1 and 2, respectively, until they become *true*.
- Let p_3 arrive at the barrier. It applies *get-and-increment* to the counter of leaf 4, increasing it to 2. Next, it moves to node 1, where it applies *get-and-increment* to the

counter, increasing it to 2. Next, it moves to the root, where it applies *get-and-increment* to the counter, increasing it to 1. It starts spinning on the global sense field of the root until it becomes *true*.

The resulting situation is as follows.



- Finally, p_5 arrives at the barrier. It applies *get-and-increment* to the counter of leaf 5, increasing it to 2. Next, it proceeds to node 2, where it applies *get-and-increment* to the counter, increasing it to 2. Then it moves to the root, where it applies *get-and-increment* to the counter, increasing it to 2. It reverses the global sense field of the root to *true*.
- p_5 moves to node 2, where it reverses the global sense field to *true*. And p_3, which finds that the global sense field of the root has become *true*, moves to node 1, where it reverses the global sense field to *true*.
- p_5 and p_3 move to leaves 5 and 4, respectively, where they reverse the global sense fields to *true*. And p_1 and p_6, which find that the global sense fields of nodes 1 and 2, respectively, have become *true*, move to leaves 3 and 6, respectively, where they reverse the global sense fields to *true*.
- p_0, p_2, p_4, and p_7 find that the global sense fields they are spinning on have become *true*. Each of the eight processes leaves the barrier and continues its execution with reversed local sense *false*.

We briefly argue that the combining tree barrier is correct. It is easy to see, by induction on depth, that the counter at a node n becomes r if and only if all processes assigned to the leaves below n have reached the barrier: in the base case of the induction, n is a leaf, and in the inductive case, the claim has already been proved for the children of n. So, in particular, the counter at the root becomes r if and only if all processes have reached the barrier. Furthermore, when this happens, it is guaranteed that the global sense fields at all nodes are reversed, so all processes leave the barrier.

15.3 Tournament Barrier

The tournament barrier is an improvement over the combining tree barrier in the sense that it allows processes to spin on local variables and does not use any read-modify-write operations.

A tournament tree is a binary tree, of a depth $k > 0$, in which each node represents a barrier of size 2. The corresponding tournament barrier can cope with 2^{k+1} processes: to each leaf we assign at most two processes. For simplicity, we assume that $N = 2^{k+1}$. Each node is divided into an active and a passive side; both sides of the node carry a global sense field. The active and passive sides of every nonleaf in the tree have one child each, and both the active and the passive sides of every leaf in the tree are assigned one process.

The idea behind the tournament barrier is that a process p at the passive side of a node signals to (the global sense field of) its active partner at this node that p has arrived at the barrier. Next, p starts spinning on the global sense field of its passive side until it has been reversed by its active partner, after which p leaves the barrier. Conversely, a process at the active side of a node waits until it receives a signal that its passive partner at this node has arrived at the barrier and then either moves on to the parent of this node, at a nonroot, or concludes that the barrier has been completed, at the root. In the latter case, the passive global sense fields are reversed at all the nodes, from top (the root) to bottom (the leaves), after which all processes resume execution, with reversed local sense.

To be more precise, nodes in the tree are numbered in the same fashion as the combining tree barrier with $r = 2$ (and the tournament tree for Peterson's mutual exclusion algorithm in section 14.4). To each node n we associate two Boolean multi-writer registers: $asense_n$ and $psense_n$. We assume that there are processes $p_0, ..., p_{N-1}$. Each process p_i maintains a single-reader/single-writer local sense field $lsense_i$. Initially, $asense_n = psense_n = false$ and $lsense_i = true$. Process p_i is assigned to leaf $(2^k - 1) + \lfloor i/2 \rfloor$ in the tree, at the active side if i is even and at the passive side if i is odd. That is, when p_i reaches the barrier, it performs the following procedure *Tournament*(n, b) with $n = (2^k - 1) + \lfloor i/2 \rfloor$ and $b = i \bmod 2$; here, $b = 0$ represents the active side and $b = 1$ the passive side of node n.

```
**if** $b = 1$ **then**
    $asense_n \leftarrow lsense_i$;
    **while** $psense_n \neq lsense_i$ **do**
        {};
        end while
    else
        while $asense_n \neq lsense_i$ do
            {};
        end while
        if $n > 0$ then
            perform procedure Tournament$\left(\lceil \frac{n}{2} \rceil - 1, (n+1) \bmod 2\right)$;
        end if
        $psense_n \leftarrow lsense_i$;
    end if
```

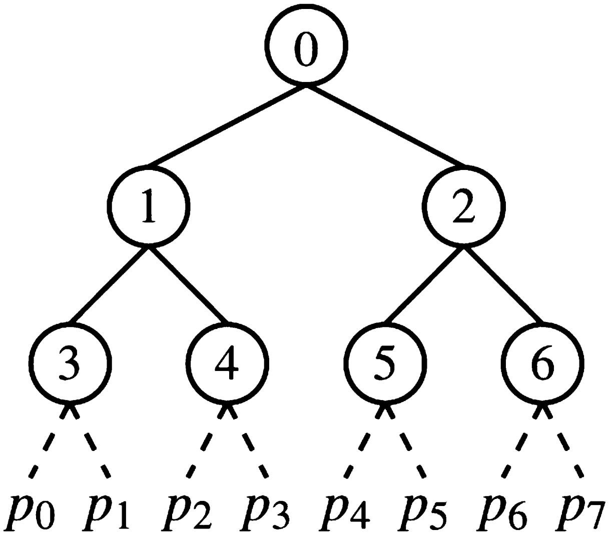
A process p_i that performs *Tournament*(n, b) acts as follows:

- If $b = 1$, then p_i sets the active sense field of n to p_i's local sense and starts spinning on the passive sense field of n until it equals p_i's local sense. When this happens, p_i reverses the passive sense fields of the nodes it visited earlier and leaves the barrier.
- If $b = 0$, then p_i starts spinning on the active sense field of n until it equals p_i's local sense. When this happens, if n is not the root of the tree, p_i moves to the parent of n, where it performs *Tournament* again. On the other hand, if n is the root of the tree, p_i reverses the passive sense fields of the nodes it visited earlier and leaves the barrier.

As noted, processes resume execution with reversed local sense.

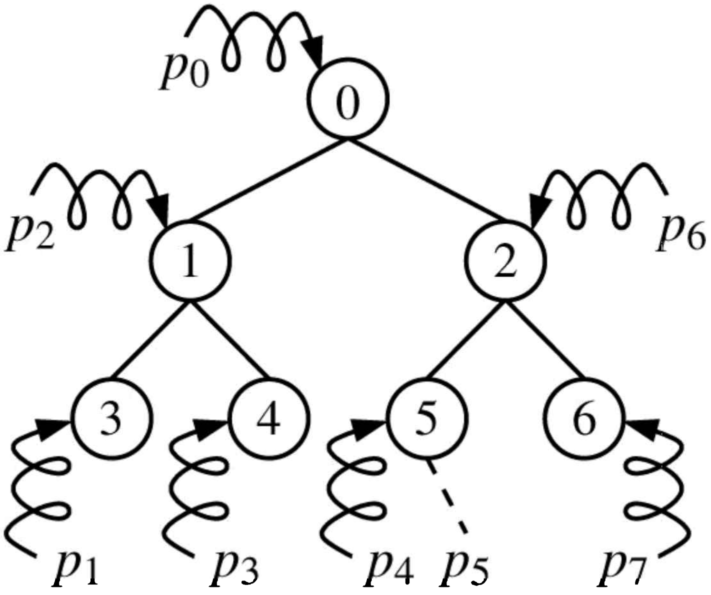
It is determined beforehand which global sense fields a process will spin on while waiting for the barrier to complete. And for each global sense field of each active or passive part of a node there is exactly one process that will spin on this field. Therefore, each of these fields can be kept in the local memory of the process that spins on it.

Example 15.3 We consider one possible execution of the tournament barrier. Let $k = 2$ and $N = 8$. The processes p_0 to p_7 are assigned to the leaves of the tree in the accompanying picture, where p_0, p_2, p_4, and p_6 are assigned to the active sides and p_1, p_3, p_5, and p_7 to the passive sides of their leaves. For each nonleaf, its even and odd children are assigned to its active and passive sides, respectively. Initially, the sense fields at both sides of the nodes are *false* and all processes have local sense *true*.



- Let p_0, p_2, p_4, and p_7 arrive at the barrier. Then p_0, p_2, and p_4 move to their leaves and start spinning on the active sense fields until they equal *true*. And p_7 moves to its leaf 6, sets the active sense field to *true*, and starts spinning on the passive sense field until it equals *true*.
- Let p_1 arrive at the barrier. It moves to its leaf 3, sets the active sense field to *true*, and starts spinning on the passive sense field.
- When p_0 finds that the active sense field of leaf 3 has become *true*, it moves to node 1, where it starts spinning on the active sense field.
- Let p_6 arrive at the barrier. It moves to its leaf 6, where it finds that the active sense field is *true*. Therefore, it moves to node 2, sets the active sense field to *true*, and starts spinning on the passive sense field.
- Let p_3 arrive at the barrier. It moves to its leaf 4, sets the active sense field to *true*, and starts spinning on the passive sense field.
- When p_2 finds that the active sense field of leaf 4 has become *true*, it moves to node 1, where it sets the active sense field to *true* and starts spinning on the passive sense field.
- When p_0 finds that the active sense field of node 1 has become *true*, it moves to the root and starts spinning on the active sense field.

The resulting situation is depicted in the following diagram, where p_0 is spinning on the active sense field of the root, while six other processes are spinning on a passive sense field.



- Finally, p_5 arrives at the barrier. It moves to its leaf 5, sets the active sense field to *true*, and starts spinning on the passive sense field.
- When p_4 finds that the active sense field of leaf 5 has become *true*, it moves to node 2. Since the active sense field of node 2 is *true*, p_4 moves on to the root, where it sets the active sense field to *true* and starts spinning on the passive sense field.
- When p_0 finds that the active sense field of the root has become *true*, it sets the passive sense fields at nodes 0, 1, and 3 to *true*.
- When p_4 finds that the passive sense field of the root has become *true*, it sets the passive sense fields at nodes 2 and 5 to *true*.
- When p_2 and p_6 find that the passive sense fields of nodes 1 and 2, respectively, have become *true*, they set the passive sense fields at leaves 4 and 6, respectively, to *true*.
- p_1, p_3, p_5, and p_7 find that the passive sense field they are spinning on has become *true*. Each of the eight processes leaves the barrier and continues its execution with reversed local sense *false*.

Correctness of the tournament tree can be argued in a similar fashion as for the combining tree barrier.

15.4 Dissemination Barrier

The dissemination barrier progresses in rounds; in each round, every process that has reached the barrier notifies some other process and waits for notification by some other process. Just as in the tournament barrier, no read-modify-write operations are used.

Suppose that N processes $p_0, ..., p_{N-1}$ must reach the barrier. A process that reaches the barrier starts by executing round 0. In a round $n \geq 0$, each process p_i that has reached the barrier

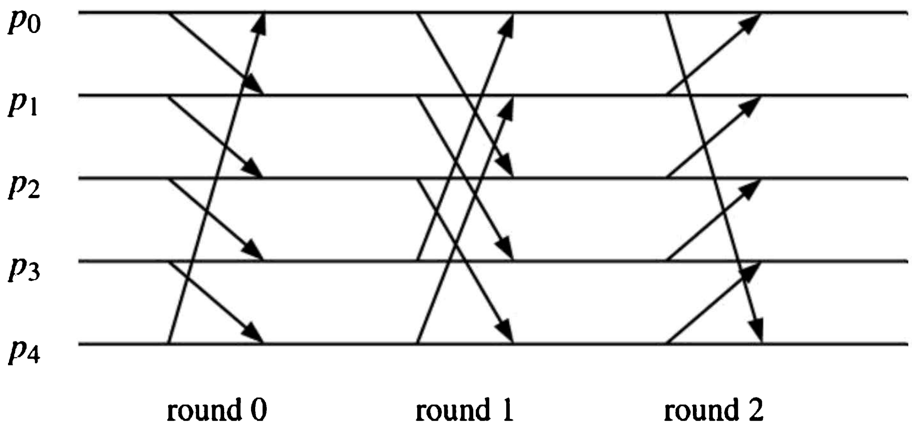
- notifies process $p_{(i+2^n) \bmod N}$,
- waits for notification by process $p_{(i-2^n) \bmod N}$, and
- progresses to round $n + 1$.

When a process completes round $\lceil \log_2 N \rceil - 1$, all N processes have reached the barrier. So then the process can leave the barrier.

Example 15.4 We consider one execution of the dissemination barrier, with $N = 5$. Since $\lceil \log_2 5 \rceil = 3$, the barrier is completed after round 2.

- In round 0, each process p_i notifies process $p_{(i+1) \bmod 5}$ and waits for notification by process $p_{(i-1) \bmod 5}$.
- In round 1, each process p_i notifies process $p_{(i+2) \bmod 5}$ and waits for notification by process $p_{(i-2) \bmod 5}$.
- In round 2, each process p_i notifies process $p_{(i+4) \bmod 5}$ and waits for notification by process $p_{(i-4) \bmod 5}$.

When a process has completed round 2, it can leave the barrier. The notifications in the three successive rounds can be depicted as follows.



Note that if, for instance, p_0 has not yet reached the barrier, then p_1 cannot complete round 0, p_2 and p_3 cannot complete round 1, and p_4 cannot complete round 2. So no process can leave the barrier.

We argue the correctness of the dissemination barrier. When all N processes have reached the barrier, clearly all $\lceil \log_2 N \rceil$ rounds can be completed by all processes. Now suppose that some process p_i has not yet reached the barrier; we argue that then no process can have completed round $\lceil \log_2 N \rceil - 1$. For simplicity, we take $N = 2^k$ for some $k > 0$, so that $\lceil \log_2 N \rceil = k$. In the following explanation, subscripts of processes are to be interpreted modulo 2^k.

- Since p_i has not reached the barrier, p_{i+1} has not completed round 0.
- Since p_i, p_{i+1} have not completed round 0, p_{i+2}, p_{i+3} have not completed round 1.
- Since p_i, p_{i+1}, p_{i+2}, p_{i+3} have not completed round 1, p_{i+4}, p_{i+5}, p_{i+6}, p_{i+7} have not completed round 2.
- ...
- Since p_i, ..., $p_{i+2^{k-1}-1}$ have not completed round $k - 2$, $p_{i+2^{k-1}}$, ..., p_{i+2^k-1} have not completed round $k - 1$.

Since subscripts are interpreted modulo 2^k, p_{i+2^k-1} is p_{i-1}. So no process has left the barrier.

Bibliographical notes

The combining tree barrier originates from [98]. The tournament barrier and the dissemination barrier stem from [43].

15.5 Exercises

Exercise 15.1 Explain under which circumstances it is more favorable to let processes that have arrived at the barrier fall asleep and under which circumstances it is better to use spinning.

Exercise 15.2 Consider the sense-reversing barrier. Suppose that the last process to reach the barrier would first reverse the value of the global sense field and only then reset the counter to 0. What could go wrong?

Exercise 15.3 Continue the application of the sense-reversing barrier in example 15.1 by reusing the counter and the sense fields (with their values reversed) for the next barrier. Give one possible execution.

Exercise 15.4 Argue that the sense-reversing barrier is a correct barrier. Take into account that the counter and sense fields are reused for multiple subsequent barriers.

Exercise 15.5 [46] Argue that the combining tree barrier can employ any barrier algorithm in its nodes (not just the sense-reversing barrier).

Exercise 15.6 Consider the tournament barrier with $k = 2$ and $N = 8$. Give an execution in which at some point only one process has not yet reached the barrier and all other processes are spinning on a passive sense field.

Exercise 15.7 Argue the correctness of the dissemination barrier for any N.

16

Distributed Transactions

A *transaction* is a sequence of (mostly internal) events that are performed as one indivisible unit. It is called a distributed transaction if there are multiple processes involved. A transaction either *commits*, in which case all of its events are performed at once, or *aborts*, meaning that none of its events take effect.

Distributed transactions are especially important in the context of a distributed database that consists of different storage locations that together are supposed to provide a centralized view of the overall database. The database usually has a client-server architecture in which servers hold all the data and clients can access the data. For example, consider a customer who wants to buy a pair of shoes from an online shop by using an electronic banking service. If the bank agrees that the customer's credit is sufficient and the shop determines that the selected shoes can be supplied, then the events of placing the order, transferring the money, and shipping the shoes are all performed, in that order. Otherwise all these events are canceled.

Transactions are required to satisfy the following four *ACID properties*:

- *Atomicity*: Either all events of a transaction take effect or none of them.
- *Consistency*: If a transaction commits, the result is a valid configuration of the system.
- *Isolation*: Intermediate effects of a transaction remain invisible to others as long as it has not committed.

- *Durability*: Once a transaction has committed, its effects are permanent.

These properties are supposed to hold even if some of the processes crash; this is especially relevant for atomicity and durability. When a process crashes, typically a fresh process is started to take over its role. The Peterson-Kearns rollback recovery algorithm from section 3.3 can be used to resume execution after a crash. It is assumed that each process p has stable storage at its disposal, which remains accessible for the other processes in a consistent state even after p has crashed.

Transactions can be nested, meaning that inside a transaction other subtransactions are started, which may be run by different processes. If a subtransaction commits, this is only provisional, in the sense that if the overall transaction aborts, then all its subtransactions are aborted as well. If a subtransaction aborts, the overall transaction is not forced to abort but may, for instance, initiate an alternative subtransaction. Another positive aspect of nested transactions is that they allow for additional concurrency. For simplicity, the techniques for distributed transactions discussed in this chapter focus on so-called flat transactions that do not contain any nesting, but they can all be lifted to nested transactions.

16.1 Serialization

Each execution of multiple concurrent transactions should be *serializable*, which means that the resulting configuration is equal to the outcome of a sequential execution of the committed transactions in which they do not overlap in time. In particular, transactions should always read values of variables that were written last in line with the order in which transactions commit. A key challenge is to avoid synchronization conflicts between concurrent transactions. Such a conflict may occur if one transaction writes to a variable in shared memory while concurrently another transaction reads or also writes to this variable. Ignoring such synchronization conflicts would give rise to two problems, which occur if a transaction determines the value of a write operation on the basis of a stale value:

- *Lost update*: A write by a committed transaction may mistakenly be ignored by a transaction that commits later. Suppose, for example, that one transaction wants to add € 10 and a second transaction € 20 to the same bank account. Both transactions read the current balance of the account, € 50. Next, the first transaction writes € 60 as the new value of the account and commits. Finally, the second transaction writes € 70 as the new value of the account and commits. The final result is that only € 20 has been added to the account instead of € 30.
- *Inconsistent retrievals*: A transaction may read inconsistent values because of writes by a concurrent transaction. For example, suppose one transaction wants to move € 10 from one bank account to another, while a second transaction wants to compute

> the sum of both accounts. The first transaction reads the balance of one of the accounts and subtracts € 10 from it. Next, the second transaction reads the values of both accounts and commits. Finally, the first transaction reads the balance of the second account, adds € 10 to it, and commits. The sum of the two accounts computed by the second transaction misses out on € 10.

Concurrent transactions that perform only reads to the same variable do not constitute a synchronization conflict. Now three different methods are discussed to enforce serializable transactions: locks, time stamps, and an optimistic approach in which a transaction only worries about synchronization conflicts at the end.

A straightforward way to achieve serialization, applied often in practice, is through the use of locks with regard to variables in shared memory on which synchronization conflicts may occur. Basically, locking enforces an ordering on concurrent transactions that may give rise to synchronization conflicts; the transaction that obtains a lock on a contested memory object first is the one that goes first in the serialization order. A distinction can be made between read and write locks. Multiple transactions may concurrently obtain the same read lock, because concurrent read operations do not constitute a synchronization conflict. However, while a write lock is granted to a transaction, no other transaction is allowed to hold this write lock or the corresponding read lock. In particular, a read lock may only be promoted to a write lock after all other granted claims on this read lock have been lifted. To guarantee a correct serialization, it is essential that after a transaction has released a lock, it will no longer claim locks. Therefore, the *two-phase locking* scheme consists of a growing phase, in which locks are accumulated, and a subsequent shrinking phase, in which the acquired locks are released. A transaction can release its read locks early on in the shrinking phase, but its write locks can only be released after the transaction has committed or aborted, because up to that point it remains unclear whether its written values will take effect. When a write lock is released, the corresponding variable carries either the written value if the transaction committed or its original value if the transaction aborted.

The locking approach has several drawbacks. First of all, a deadlock can occur if two concurrent transactions need to obtain the same two locks and request these locks in opposite order. In that case, both transactions may succeed in obtaining one of the locks but will then never obtain the other lock. Such deadlocks can be avoided either by prudent lock management, which inevitably has a negative impact on performance, or by a deadlock detection scheme using wait-for graphs, as explained in chapter 5, and aborting transactions if a deadlock is detected. A second drawback is that locking tends to impose a considerable performance penalty because of the overhead of lock management and the reduction in concurrency, which is aggravated by the fact that in two-phase locking locks can only be released after the growing phase. Note that if a process crashes while it is involved in a transaction holding a lock, then the lock can only be released after the corresponding transaction has been aborted or another process has resumed the execution of the crashed process.

Example 16.1 We revisit the scenario from the explanation of lost updates. If both transactions concurrently obtain the read lock of the bank account, then a deadlock ensues because both transactions are incapable of getting the write lock of the account. Either this deadlock must be resolved after the fact or prudent lock management disallows the transactions concurrent access to the read lock. In the latter case, one of the transactions obtains the read lock, promotes it to a write lock, adds its amount to the account, commits, and releases the write lock. Next, the other transaction can perform this same sequence of events. At the end, the balance of the account has increased from € 50 to € 80 by adding € 10 and € 20 in some sequential order.

We now discuss two alternative approaches for serializing transactions that do not use locks. Next, to lost updates and inconsistent retrievals, two other possible problems related to aborted transactions need to be avoided:

- *Premature write*: A write by a transaction to a variable *x* may be obliterated by a concurrent transaction that wrote to *x* earlier, aborted, and reset the value of *x*. Suppose, for example, that one transaction wants to add € 10 and a second transaction wants to add € 20 to the same bank account. Both transactions read the current balance of the account, € 50. Next, the first transaction writes € 60 as the new value of the account. Now the second transaction writes € 70 as the new value of the account and commits. Finally, the first transaction aborts and resets the balance to € 50. The final result is that no money has been added to the account instead of € 20.
- *Dirty read*: A transaction may read a value that was written by a concurrent transaction that eventually aborts. Suppose once again that one transaction wants to add € 10 and a second transaction wants to add € 20 to the same bank account. The first transaction reads the current balance of the account, € 50, and writes € 60 as the new value of the account. Next, the second transaction reads € 60 as the value of the account. Now the first transaction aborts and resets the balance to € 50. Finally, the second transaction writes € 80 as the new value of the account and commits. The final result is that € 30 has been added to the account instead of € 20.

Premature writes and dirty reads are evaded by letting transactions perform tentative writes on local variables instead of in shared memory. We discuss two lockless approaches to avoid that transactions read stale values.

One approach to achieve a proper serialization is through time stamps. Each transaction is furnished with a unique time stamp, typically the time of a global clock at the moment of its instantiation. Transactions are serialized according to their time stamps. A transaction cannot commit as long as there are ongoing transactions with a lower time stamp. Transactions tentatively perform write operations on local copies of variables, which become definite only when the transaction commits. Each (tentative) read or write operation is immediately visible to all other ongoing transactions. Consider a read operation by a transaction T with time stamp t on variable x. This operation is delayed while there are ongoing transactions with time stamps smaller than t that wrote to x. When there are no such transactions, the read operation returns the last value witten to x either by a committed transaction with a time stamp smaller than t or by T itself.

Now consider a write operation by transaction T on variable x. This operation is performed only if no transaction with a time stamp greater than t read x. The write is therefore first checked with all transactions that have a time stamp greater than t. For this purpose, before a transaction starts, it is first announced at all ongoing transactions. If a transaction with a time stamp greater than t read x, then T must abort. Else the write operation can be performed.

Premature writes cannot occur because transactions perform tentative writes on local copies of variables. Dirty reads cannot occur because reads always return a value written by a committed or the same transaction. Lost updates and inconsistent retrievals cannot occur because reads and writes are performed in accordance with the time stamp order.

Example 16.2 We again revisit the scenario from the explanation of lost updates. Let the first and second transactions have time stamps t_1 and t_2, respectively, with $t_1 < t_2$. The two transactions both read the balance, € 50, of the bank account b. The first transaction is disallowed to add € 10 to the account and aborts, in view of the read by the second transaction and $t_1 < t_2$. The second transaction changes the balance of the account to € 70 and commits.

Another lockless serialization approach is by means of *optimistic concurrency control.* The underlying assumption is that the chance that a synchronization conflict occurs is low. Therefore, a transaction is allowed to perform reads and tentative writes without worrying about such conflicts. Only at the end of the transaction is it validated whether indeed no synchronization conflicts have occurred. If a transaction aborts, it needs to restore the original values of the variables it wrote to.

To avoid dirty reads, a transaction T_1 that has read a value written by another ongoing transaction T_2 cannot commit until T_2 has committed. If T_2 eventually aborts, then T_1 must also abort. This may in turn lead to aborts of other transactions that read values written by T_1. These are called cascading aborts, which can be avoided by requiring that a transaction only be allowed to read values that were written by committed transactions. A transaction that wants to read a value written by an ongoing transaction must then delay this read until the other transaction has committed or aborted.

A more effective approach is to let a transaction copy memory objects to a private workspace at the start. The workspace consists of local memory at the processes involved in the transaction that is only accessible by this transaction. During its working phase, a transaction performs reads and writes on this private workspace. If in the end the transaction commits, then before doing so the updates in its private workspace are copied to shared memory. If the transaction aborts, then these updates remain invisible to the other transactions. Since writes are not rolled back by aborted transactions and all reads are performed with regard to committed values of variables, premature writes and dirty reads are precluded.

To avoid lost updates, after its working phase, a transaction enters a validation phase to determine whether it erroneously used stale values. At the start of this phase, the transaction is assigned a unique sequence number; these numbers are issued in ascending order and represent the order in which transactions are serialized. A transaction can only commit when all transactions with a smaller sequence number have aborted or committed. The advantage of assigning a sequence number to a transaction at the start of its validation instead of before its working phase is that a transaction with a long working phase does not delay transactions that start later but complete much earlier than this transaction.

A transaction T at the start, before copying memory objects to its private workspace, stores the sequence number k of the transaction that was the last to commit. After having been assigned a sequence number $\ell > k$ itself, T can perform its validation phase when all transactions with a smaller sequence number than ℓ have either aborted or committed (and in the latter case, made all written values visible to the other transactions). Then T checks whether variables it read (in its private workspace) during its working phase were written to by other transactions with a sequence number greater than k but smaller than ℓ. If so, then T aborts, because it used stale values of these variables with regard to the serialization order; if not, then T commits. This technique to determine and avoid synchronization conflicts is called backward validation, as opposed to the alternative forward validation technique, in which a transaction checks whether variables it wrote to were read by overlapping active transactions. Note that backward validation requires that a write be archived after it has been overwritten until all transactions concurrent to the transaction that performed the write have completed their validation phases.

Finally, if its validation phase concludes positively with a commit, then in a subsequent update phase a transaction makes all its written values visible to the other transactions by copying them from its private workspace to shared memory.

Example 16.3 We again revisit the scenario from the explanation of lost updates. Let both transactions store the same highest sequence number k of any committed transaction so far. Next, each copies the value € 50 of the bank account to its private workspace. The first and second transactions concurrently read this value and write € 60 and € 70, respectively, in their private workspaces as the new value of the account.

The first and second transactions proceed to their validation phases, where they are assigned sequence numbers k+1 and k+2, respectively. Validation of the first transaction succeeds, so it commits and copies the value € 60 from its private workspace to shared memory. Now validation of the second transaction fails because the first transaction has a sequence number between k and $k + 2$ and it wrote a value to the bank account, while the second transaction read the value of the bank account.

Time stamps and the optimistic approach avoid the reduction in concurrency and overhead of lock management and do not suffer from deadlock. But read operations may still be delayed, and a surge of synchronization conflicts may lead to a massive abortion of transactions.

16.2 Two- and Three-Phase Commit Protocols

A distributed transaction is initiated by a process called the *coordinator* of this transaction, while the other processes involved in the transaction are called *cohorts*. A transaction is started by the coordinator sending a request to all cohorts. In the end, it is up to the coordinator to decide whether the transaction can be committed or should be aborted. First, we discuss the *two-phase commit protocol*. It employs a unanimous voting scheme: the coordinator commits the transaction if and only if all participants in the transaction vote to do so. In the context of databases, next to a synchronization conflict, another ground for a process to vote for (and thus enforce) abortion of a transaction is if it detects that the transaction causes a resource deadlock (see section 5.1).

Each participant in the transaction locally keeps track of updates it makes to data values during the transaction. The old and new values are both kept, so that the process can stick to the old values or move to the new values if the transaction aborts or commits, respectively. In view of the isolation property, as long as the transaction has not been committed, the new values must remain invisible to others.

In the voting phase, the coordinator asks each participant in the transaction (including itself) whether it agrees to commit the transaction. Each participant replies with either a **yes** vote, if all its events during the transaction succeeded, or a **no** vote, if it experienced an obstruction that makes it impossible to commit. Before replying with a **yes** vote, a process must make sure that its tentative changes for the transaction have been copied to stable storage, to be able to cope with a crash of the process while the coordinator decides to commit the transaction. After casting its vote, each process blocks until it receives either a **commit** or an **abort** message from the coordinator. The coordinator waits for votes from all participants to start the completion phase. If all participants vote **yes**, then the coordinator sends a **commit** message to all participants; if, on the other hand, at least one of the participants votes **no**, then the coordinator broadcasts an **abort** message. If a **commit** message is received, participants make the effects of their events during the transaction visible to others, annihilating the old values. If an **abort** message is received, participants annihilate the effects of their events during the transaction, which means they roll back to the old values. In both cases, participants release the resources they held during the transaction and send an acknowledgment to the coordinator. The coordinator considers the transaction finished when acknowledgments have been received from all participants.

To cope with crashed processes, we assume here that there is a complete and strongly accurate failure detector. As explained in section 12.3, such a failure detector can be implemented using a timeout mechanism if there is a known upper bound on network latency and processes regularly send a heartbeat message. In the voting phase, if the coordinator detects that a cohort has crashed, then this counts as a **no** vote. In the completion phase, if the coordinator broadcasts a **commit** message and finds that a cohort has crashed, then it needs to be ensured that the effects of the transaction at the crashed process are made visible and permanent before the transaction can be completed. This is possible because the crashed process must have copied its written values during the transaction to stable storage, clearly marked, before replying with a **yes**.

The Achilles' heel of the two-phase commit protocol is that the coordinator may crash at the end of the voting or at the start of the completion phase, especially if a cohort crashes concurrently. It might be the case that the coordinator sent a **commit** or **abort** message only to the crashed cohort, which before crashing made the effects of the transaction visible or undid these effects irrevocably, respectively. Even after a new coordinator has been put in place, the cohorts must still reach agreement on whether the transaction should be committed. Until this agreement has been reached, the cohorts are blocked.

The *three-phase commit protocol* eliminates this problem by introducing an intermediate precommit phase, which the coordinator enters after receiving a **yes** from all participants. Instead of immediately broadcasting a **commit**, it first sends a **precommit** message to all participants, informing them of the intention to commit the transaction. Upon receipt of this message, each participant replies with an **ack** message. After receiving this reply from all participants, the coordinator finally broadcasts a **commit** and proceeds as in the two-phase commit protocol. If a cohort crashes before sending an **ack**, the coordinator could still decide to commit the transaction. Then the recovery protocol must make sure that the process replacing the crashed cohort will eventually take care of properly completing its part of the transaction.

If the coordinator crashes in the precommit phase, the remaining cohorts can now agree among themselves whether to commit the transaction, because the cohorts have neither rolled back nor made visible their written values during the transaction. In view of the complete and strongly accurate failure detector, the rotating coordinator consensus algorithm from section 12.4 can be employed to reach agreement among the cohorts. For example, we could let a cohort vote 0 if it did not reply with an **ack** to a **precommit** message and vote 1 otherwise. If the cohorts agree on 1, then they can safely commit the transaction, because the coordinator sent out a **precommit** message. The processes replacing the crashed coordinator or crashed cohorts will eventually take care that their parts of the transaction will be made visible and permanent as well. On the other hand, if the cohorts agree on 0, then they can safely abort the transaction, because the coordinator cannot have sent out a **commit** message.

16.3 Transactional Memory

Locks tend to play a key role in coping with concurrency in the presence of shared memory. However, as already discussed in section 16.1, they have some serious disadvantages. They can become a performance bottleneck if multiple processes want to concurrently obtain the same lock. A deadlock may occur if two processes need to claim the same two locks and obtain them in opposite order. And if a process crashes while holding the lock, this needs to be resolved before the lock can be released. Last but not least, it turns out that in the development of large software systems, locks are vulnerable to design errors, because the intended relation between a lock and the data it protects remains implicit. Read-modify-write operations do not really come to the rescue, because they, too, impose a performance penalty and are delicate to use. Furthermore, they operate on a single variable, while designing a concurrent computer program would become much simpler if one could perform multiple read and write operations on different variables in one atomic step.

Transactional memory is a concurrent programming paradigm inspired by the transactions explained in the first part of this chapter. The idea is that a sequence of reads and writes executed by a single process can be declared a transaction, in which case these steps either commit and take effect in one atomic step or are aborted and do not take effect. In the latter case, the transaction is often retried later. A transaction may be aborted by another transaction if a synchronization conflict arises between these two transactions. For example, a process may try to move the head of one queue to the tail of another queue in one atomic step. To do this, it tentatively performs a dequeue to the first queue followed by an enqueue to the second queue, within one transaction. If no conflicting operation by another transaction is performed to these queues in the meantime, these two operations are made visible to the other processes in one atomic step.

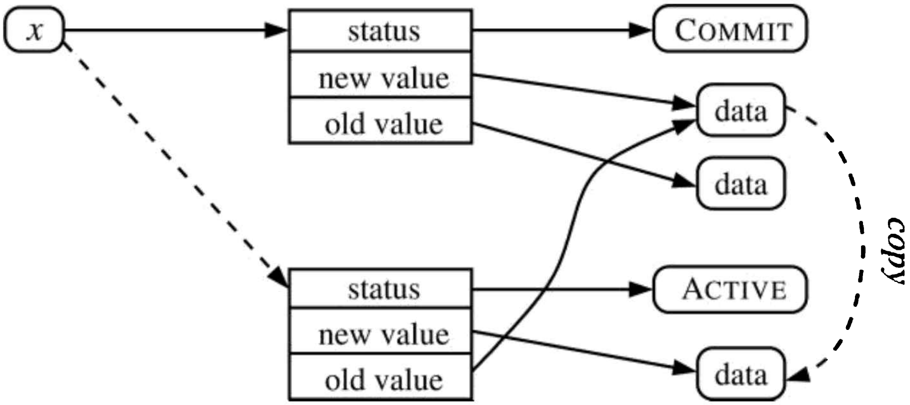
Transactions may be nested in the sense that a transaction may start a subtransaction. However, for simplicity, we again restrict the discussion to flat transactions, disregarding nesting.

As explained in section 2.3, at the hardware level, a cache buffers writes that were performed but still need to be transferred to main memory. A cache coherence protocol, such as the MSI protocol described in section 2.3, takes care of most of the work needed to support transactions: its main purpose is to detect and resolve synchronization conflicts. *Hardware transactional memory* adds a transactional bit to each cache line. If a value is written in a cache line on behalf of a transaction, the transactional bit of this line is set. If such a so-called transactional line is invalidated before the transaction has committed, then the transaction aborts, meaning that all transactional lines in the corresponding cache are invalidated. If, on the other hand, the transaction in the end commits, then simply all its transactional bits are set, so the corresponding transactional lines become permanent.

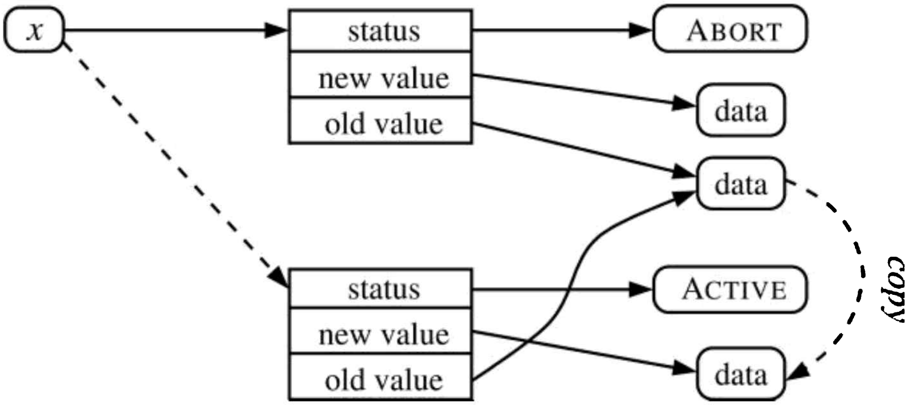
Hardware transactional memory exploits a cache coherence protocol in an effective way but does have some serious drawbacks. The size of a transaction is limited by the size of the cache, which tends to be relatively small. Because of the granularity of cache lines, transactions can suffer from false sharing. In particular, care must be taken that a transaction does not abort itself if it happens to write twice to the same cache line. And transactions can starve each other by repeatedly aborting another process, being aborted by another process, and restarting. In spite of these weak points, Intel's TSX provides widely used hardware transactional memory support.

Software transactional memory implements transactions at the software level. The idea is that the status of a transaction is maintained in a field that initially is Active and ultimately is changed either to Commit or to Abort. A transaction can try to commit by applying *compare-and-set*(Active, Commit) to its own status field, or it can try to abort another transaction by applying *compare-and-set*(Active, Abort) to that transaction's status field. The use of the *compare-and-set* operation is vital in case a transaction tries to commit while concurrently another transaction tries to abort this transaction. Since the check whether the status field is Active and its change to Commit or Abort is performed in one atomic step, only one of these concurrent operations can succeed.

Each memory object x carries a pointer to the transaction T that performed the last write to it. Two values for x are maintained: the new one written by T and the old one before this write. While T is in Active, the value of x is undecided. If T is in Commit, the new value is the current value of x and the old value is obsolete. If, on the other hand, T is in Abort, the old value is the current value of x and the new value is obsolete. When a transaction T' reads x, it can see from T's status which value of x it should consider. If this status is Active, then T' observes a synchronization conflict, in which case it either waits for T to complete or performs *compare-and-set*(Active, Abort) to T's status field. When a transaction T' writes to x, it prepares a new object pointing to the Active status field of T'. If T's status is Commit, then in the object its new value is a copy of the new value of x and its old value points to the new value of x. Next, T' applies a *compare-and-set* to try and swing x's pointer to the object it has prepared, so that all changes take effect in one atomic step. This can be depicted as follows.



Likewise, if *T*'s status is ABORT, then in the object its new value is a copy of the old value of *x* and its old value points to the old value of *x*.



If *T*'s status is ACTIVE, then again T' observes a synchronization conflict and acts accordingly.

There are different policies regarding the circumstances under which a transaction is allowed to abort another conflicting transaction. One could, for instance, provide transactions with a time stamp or keep track of the amount of work they have performed so far, and give precedence to transactions with an older time stamp or that have done more work.

An active transaction must take care that it does not encounter an inconsistent state. For example, let a transaction T_1 read *x*, then another transaction T_2 writes to *x* and *y* and commits, and finally T_1 reads *y*. Then T_1 may enter a state in which the combination of values of *x* and *y* is inconsistent. To avoid such situations, each transaction keeps a log of its reads and must after every read and before it commits check that the values it has read before have not been changed by another transaction. If one of these values did change, it must immediately abort itself. In the aforementioned example, T_1 would find after reading y that another transaction has written to x, causing T_1 to abort.

Example 16.4 We revisit the scenario from the explanation of lost updates in section 16.1 for the last time, in the context of software transactional memory. Both transactions concurrently read the value € 50 of the bank account. The first transaction tentatively writes € 60 as the new value of the account and commits, making this new value permanent. The second transaction tentatively tries to write € 70 as the new value of the account, but this fails because of the earlier write by the first transaction. The second transaction therefore aborts and restarts.

A general drawback of the transactional memory paradigm is that it cannot cope well with output a transaction may want to send to a remote process or the environment, since this cannot be rolled back if the transaction aborts. One approach is to buffer such output until the transaction has completed, which works only if no immediate input is required in reply to the output. Another approach is that at any time at most one privileged transaction is guaranteed to commit and can freely perform output operations. In any case, only a very limited amount of output by transactions is possible in a transactional memory setting.

Bibliographical notes

The ACID properties were put forward in [42], and serializability is due to [33]. The time stamp ordering for transactions was introduced in [11], and optimistic concurrency control stems from [51]. The two-phase commit protocol appeared in [41], and the three-phase commit protocol is found in [87]. Hardware transactional memory was proposed in [45] and software transactional memory in [86].

16.4 Exercises

Exercise 16.1 Let transactions T_1 and T_2 execute the sequences of events $\mathit{read}(x)$; $\mathit{write}(y)$; $\mathit{write}(x)$ and $\mathit{read}(y)$; $\mathit{write}(y)$; $\mathit{write}(x)$, respectively, for some variables x and y. Give three different interleavings of the executions of T_1 and T_2 for which T_1 can be serialized before T_2. How many interleavings exist for which T_2 can be serialized before T_1?

Exercise 16.2 Give examples to show how two-phase locking, the time stamp ordering, optimistic concurrency control, and software transactional memory deal with the scenario regarding a bank account in the explanation of inconsistent retrievals.

Exercise 16.3 Give examples to show that the following relaxations of two-phase locking could lead to a violation of serializability.

(a) A transaction is allowed to claim a read lock after it has released a read lock.
(b) A transaction is allowed to release a write lock before it commits.

Exercise 16.4 Give examples to show how the time stamp ordering, optimistic concurrency control, and software transactional memory deal with the scenario regarding a bank account in the explanation of dirty reads.

Exercise 16.5 Consider a distributed transaction with one coordinator and three cohorts. Give a computation of the two-phase commit protocol in which crashed processes must be restarted before agreement can be reached on whether the transaction commits. Also show how this computation proceeds and concludes with a commit or abort in the case of the three-phase commit protocol.

Exercise 16.6 Explain how the two-phase commit protocol can be extended to nested transactions.

Exercise 16.7 In software transactional memory, why is it unnecessary for an active transaction to check whether the values it has written to are unchanged?

17

Self-Stabilization

A distributed algorithm is *self-stabilizing* if it will always end up in a correct configuration, even if it is initialized in an incorrect (possibly unreachable) configuration. A strong advantage of self-stabilization is that it provides fault tolerance even in circumstances where the system is forced into an incorrect configuration; for example, because of a hardware error or a malicious intruder. Self-stabilization can offer an attractive solution if failures are infrequent and a temporary malfunction is acceptable, as is often the case in operating systems and database systems. An important requirement is that failures be resolved within a relatively short period of time.

Self-stabilizing algorithms are generally presented in a shared-memory framework. The reason is that in a message-passing framework, all processes might be initialized in a state in which they are waiting for a message to arrive, in which case the network would exhibit no behavior at all. In shared memory, processes may take into account the values of variables at their neighbors, in which case such deadlocks can be avoided. We assume that the local variables at the processes are single-writer registers and that processes can read values of variables at their neighbors.

We will discuss a self-stabilizing algorithms for mutual exclusion, where initially multiple processes may be privileged. We will moreover discuss two self-stabilizing algorithms for computing a spanning tree, where initially there may, for instance, be a cycle in the spanning tree. Such a spanning tree construction serves as a cornerstone for several other self-stabilizing algorithms, for example, to compute a snapshot or to elect a leader.

17.1 Dijkstra's Token Ring for Mutual Exclusion

Dijkstra's self-stabilizing token ring for mutual exclusion assumes a directed ring of processes $p_0, ..., p_{N-1}$. Each process p_i holds a single-writer register x_i with values in $\{0, ..., K - 1\}$, where $K \geq N$; process p_i can read the value of the register at its predecessor $p_{(i-1) \bmod N}$ in the ring. The privileged processes are defined as follows:

- p_i for $i = 1, ..., N - 1$ is privileged if $x_i \neq x_{i-1}$.
- p_0 is privileged if $x_0 = x_{N-1}$.

Since Dijkstra's token ring can be initialized in any configuration, there can be multiple privileged processes at the start (if $N \geq 3$).

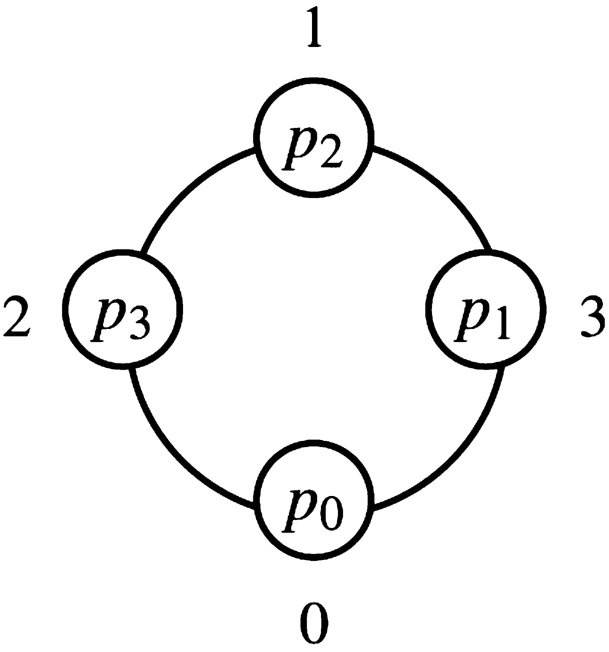
In Dijkstra's token ring, each privileged process is allowed to change its value, causing the loss of its privilege.

- p_i can perform $x_i \leftarrow x_{i-1}$ if $x_i \neq x_{i-1}$, for any $i = 1, ..., N - 1$.
- p_0 can perform $x_0 \leftarrow (x_0 + 1) \bmod K$ if $x_0 = x_{N-1}$.

Example 17.1 Consider a ring of size 3 with $K = 3$. Initially, each process has the value 0, so only process p_0 is privileged. Then p_0 can pass on the privilege to p_1 by setting x_0 to 1. Next, p_1 can pass on the privilege to p_2 by setting x_1 to 1. Now p_2 can pass on the privilege to p_0 by setting x_2 to 1, and so on.

Always at least one process is privileged. That is, if $p_1, ..., p_{N-1}$ are not privileged, then clearly the registers $x_0, ..., x_{N-1}$ all contain the same value. But then p_0 is privileged because $x_0 = x_{N-1}$. Furthermore, an event at a process p_i never increases the number of privileged processes, because p_i loses its privilege, and the event can at most cause p_i's successor $p_{(i+1) \bmod N}$ in the ring to become privileged. So if the initial configuration is correct, in the sense that only one process is privileged, then mutual exclusion is guaranteed.

Example 17.2 Let $N = K = 4$, and consider the following initial configuration.



Initially, p_1, p_2, and p_3 are privileged. Each computation will eventually lead to a configuration in which only one process is privileged. The value at p_0 is different from the values at p_1, p_2, and p_3. In the proof of theorem 17.1, it will be argued that in each infinite computation, p_0 must eventually perform an event. The only way p_0 can perform an event is if the register at p_3 attains the value 0. This can happen only if first the register at p_2 attains this value. And in turn this can happen only if first the register at p_1 attains the value 0. Then the registers at p_1, p_2, and p_3 have attained the value 0, so only p_0 is privileged.

For instance, first $x_3 \leftarrow 1$; next $x_2 \leftarrow 3$ and $x_3 \leftarrow 3$; and finally $x_1 \leftarrow 0$, $x_2 \leftarrow 0$, and $x_3 \leftarrow 0$. Now only p_0 is privileged.

Theorem 17.1 *If $K \geq N$, then Dijkstra's token ring for mutual exclusion always eventually reaches a correct configuration, in which starvation-free mutual exclusion is guaranteed.*

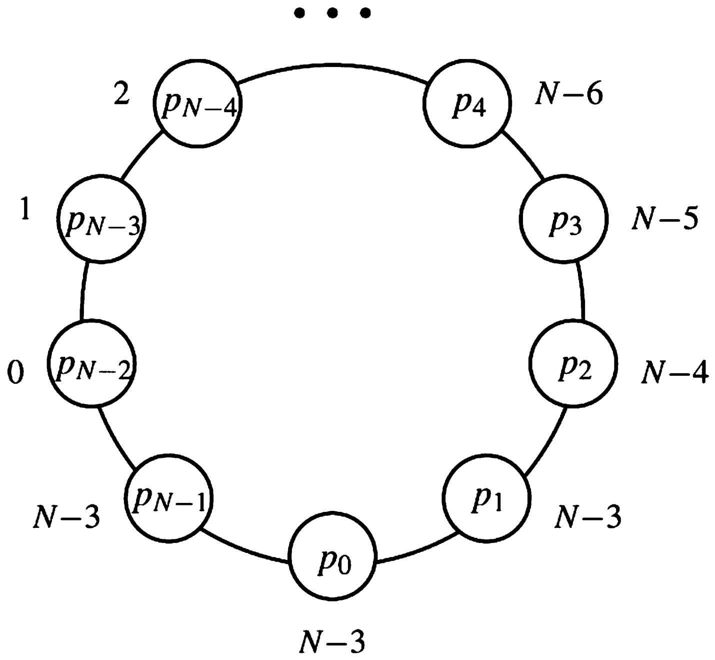
Proof. Consider an infinite computation. We need to argue that eventually a configuration is reached in which only one process is privileged. The longest possible sequence of transitions without an event at p_0 consists of $\frac{1}{2}\cdot(N-1)\cdot N$ events at p_1, ..., p_{N-1}: one event at p_1 (copying p_0's value), two events at p_2 (copying p_1's first and second values), and so on, up to $N - 1$ events at p_{N-1}. So the infinite computation involves infinitely many events at p_0. At each such event, p_0 increases its value by 1 modulo K, so during the computation, x_0 ranges over all values in $\{0, ..., K -1\}$. Initially, at least one of those values is not present in the ring, because $K \geq N$. Since $p_1, ..., p_{N-1}$ only copy values from their predecessor, it follows that in some configuration of the computation, $x_0 \neq x_i$ for all $i = 1, ..., N - 1$. The next time p_0 becomes privileged, when $x_{N-1} = x_0$, clearly $x_i = x_0$ for all $i = 1, ..., N - 1$. Then only p_0 is privileged, so mutual exclusion has been achieved.

Starvation-freeness follows from the fact that in each correct configuration, the privileged process will pass on the privilege to its successor (if $N \geq 2$). □

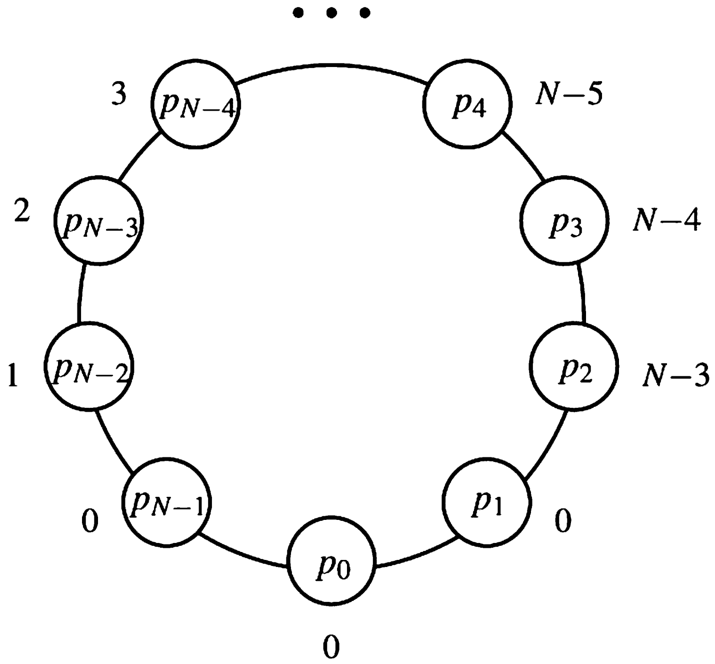
For $N \geq 3$, theorem 17.1 also holds if $K = N - 1$. (If $N = 2$ and $K = 1$, starvation-freeness is violated.) Let us revisit the argumentation in the proof of theorem 17.1. When p_{N-1} copies the value from p_{N-2}, the processes $p_1, ..., p_{N-1}$ hold at most $N - 2$ different values (because $N \geq 3$). Since $p_1, ..., p_{N-1}$ only copy values, they are then restricted to these $N - 2$ values, as long as the value of x_0 is also among these $N - 2$ values. Since $K \geq N - 1$, and in an infinite computation p_0 performs infinitely many events, it follows that in some configuration of the computation, $x_0 \neq x_i$ for all $i = 1, ..., N - 1$. The next time p_0 becomes privileged, $x_i = x_0$ for all $i = 1, ..., N - 1$. Then only p_0 is privileged.

The value $K = N - 1$ is sharp; the next example shows that if $K = N - 2$, then there are infinite computations in which mutual exclusion is never achieved.

Example 17.3 Let $N \geq 4$ and $K = N - 2$, and consider the following initial configuration.



In this configuration, only p_1 is not privileged. We consider one possible computation of Dijkstra's token ring. First, p_0 sets x_0 to $((N - 3) + 1) \bmod (N - 2) = 0$. Next, p_{N-1} sets x_{N-1} to 0. Then, p_{N-2} sets x_{N-2} to 1, and so on. This sequence of events proceeds in a clockwise fashion until finally p_1 sets x_1 to 0. Then we have reached the following configuration.



The only difference with the initial configuration is that the values of the registers have increased by 1, modulo $N - 2$. In particular, in the preceding configuration, again only p_1 is not privileged. This execution pattern can be repeated over and over again, leading to an infinite computation. In each of its configurations, $N - 1$ processes are privileged.

17.2 Arora-Gouda Spanning Tree Algorithm

In the Arora-Gouda self-stabilizing spanning tree algorithm for undirected networks, the process with the largest ID eventually becomes the root of a spanning tree of the network. The algorithm requires that all processes know an upper bound K on the network size.

Each process keeps track of its parent in the spanning tree, which process is the root, and the distance to this root via the spanning tree. Because of arbitrary initialization, there are three complications: first, multiple processes may consider themselves the root; second, there may be a cycle in the spanning tree; and third, there may be a "false" root, meaning that processes may consider a process q the root while q is not in the network at all. The idea behind the Arora-Gouda algorithm is that these inconsistencies can be resolved if a process declares itself the root of the spanning tree, and adapts its local variables accordingly, every time it detects an inconsistency in the values of its local variables. Moreover, a process may resolve inconsistencies between the values of its own local variables and those of its neighbors.

Each process p maintains the following variables:

$parent_p$: p's parent in the spanning tree;
$root_p$: the root of the spanning tree;
$dist_p$: p's distance from the root, via the spanning tree.

The value $\perp$ for $parent_p$ means that p's parent is undefined (in particular, when p considers itself the root). A process p declares itself the root by executing

$$parent_p \leftarrow \perp \qquad root_p \leftarrow p \qquad dist_p \leftarrow 0,$$

when it detects an inconsistency in the values of its local variables:

- $root_p < p$; or
- $parent_p = \perp$, and $root_p \neq p$ or $dist_p \neq 0$; or
- $parent_p \neq \perp$ and $parent_p$ is not a neighbor of p; or
- $dist_p \geq K$.

In the first case, $root_p$ is not the largest ID in the network. In the second case, $parent_p$ says p is the root, while $root_p$ or $dist_p$ says it is not. In the third case, $parent_p$ has an improper value. And the fourth case is in contradiction with the fact that K is an upper bound on the network size.

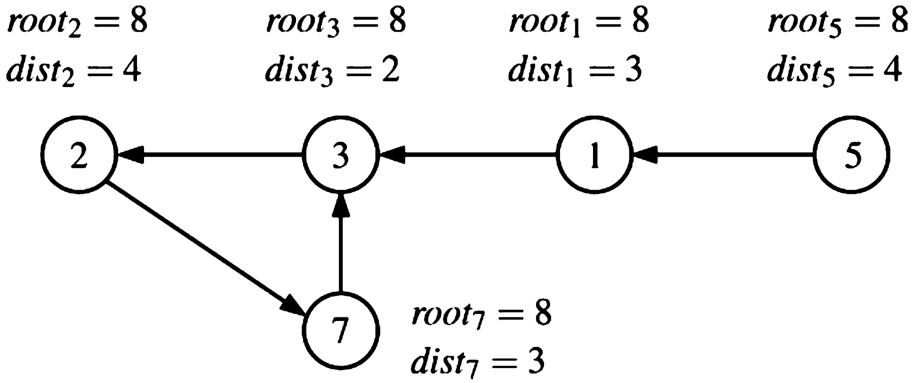
Suppose there is no such inconsistency in the local variables of p. Let q be a neighbor of p with $dist_q < K$. If $q = parent_p$ and p detects an inconsistency between its own variables and q's, then p can bring its root and distance values in line with those of q:

$$root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1.$$

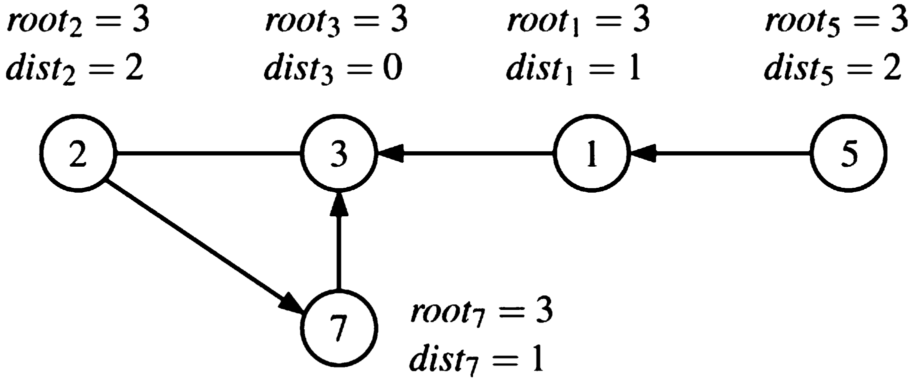
Furthermore, if $q \neq parent_p$ and $root_p < root_q$, then p can make q its parent:

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1.$$

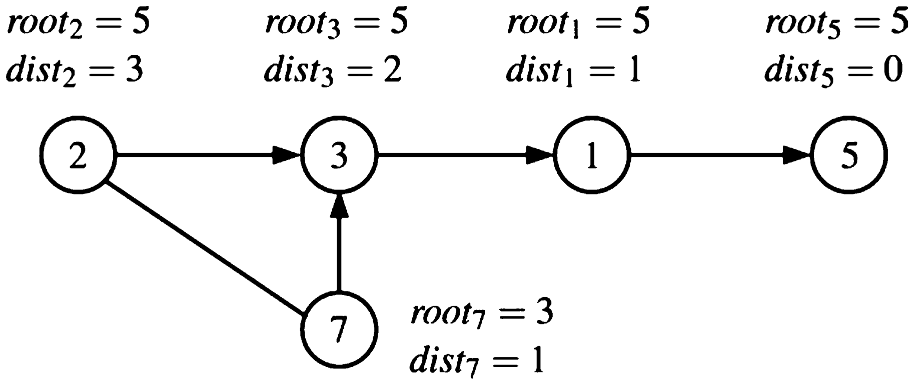
Example 17.4 We consider one possible computation of the Arora-Gouda algorithm on the following undirected network with $K = 5$. Arrows point from a child to its parent. Note that all processes consider process 8 the root, but this is a false root.



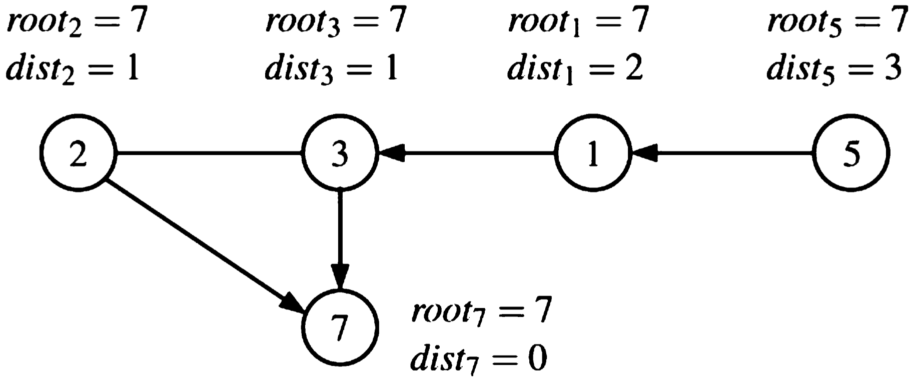
First, process 3 notes that it has distance 2 to the root, while its parent, 2, has distance 4. Therefore, process 3 sets its distance to 5. Then it has a distance equal to $K = 5$, so it declares itself the root: $parent_3 \leftarrow \perp$, $root_3 \leftarrow 3$, and $dist_3 \leftarrow 0$. As a result, processes 7 and 1 set their root to 3 and their distance to 1; next, processes 2 and 5 set their root to 3 and their distance to 2.



Now process 5 finds an inconsistency in its local variables: its root value is smaller than its own ID. Therefore, it declares itself the root: $parent_5 \leftarrow \perp$, $root_5 \leftarrow 5$, and $dist_5 \leftarrow 0$. As a result, first process 1 makes process 5 its parent: $parent_1 \leftarrow 5$, $root_1 \leftarrow 5$, and $dist_1 \leftarrow 1$; next, process 3 makes process 1 its parent: $parent_3 \leftarrow 1$, $root_3 \leftarrow 5$, and $dist_3 \leftarrow 2$; and then, process 2 makes process 3 its parent: $parent_2 \leftarrow 3$, $root_2 \leftarrow 5$, and $dist_2 \leftarrow 3$.



Now process 7 finds an inconsistency in its local variables: its root value is smaller than its own ID. Therefore, it declares itself the root: $parent_7 \leftarrow \perp$, $root_7 \leftarrow 7$, and $dist_7 \leftarrow 0$. As a result, processes 2 and 3 make process 7 their parent; next, process 1 makes process 3 its parent; and finally process 5 makes process 1 its parent.



The resulting configuration, depicted in the preceding diagram, is stable.

We argue that the Arora-Gouda spanning tree algorithm is self-stabilizing if only fair computations are considered (see exercise 17.5). The key is that false root values, which are not an ID of any process in the network, will eventually disappear. Because a false root value can survive only in a cycle of processes that all agree on this root value. Distance values of processes in such a cycle will keep on increasing until one of them gets distance K and declares itself the root. Then the cycle is broken, and by fairness the cycle can be reestablished only a finite number of times. Hence, the false root of the (former) cycle will eventually be eradicated. Since false roots are guaranteed to disappear, the process with the largest ID in the network will eventually declare itself the root. Then the network will converge to a spanning tree with this process as the root.

To obtain a breadth-first search tree, in the Arora-Gouda algorithm, the case where $q \neq parent_p$ is a neighbor of p with $dist_q < K$ has one extra subcase: if $root_p = root_q$ and $dist_p > dist_q + 1$, then $parent_p \leftarrow q$ and $dist_p \leftarrow dist_q + 1$. That is, a process can select a new parent if it offers a shorter path to the root.

17.3 Afek-Kutten-Yung Spanning Tree Algorithm

The Afek-Kutten-Yung self-stabilizing spanning tree algorithm for undirected networks does not require a known upper bound on the network size. Moreover, it does not exhibit any (unfair) infinite computations. Again, the process with the largest ID eventually becomes the root of a spanning tree of the network.

Each process p again maintains the variables $parent_p$, $root_p$, and $dist_p$. A process p declares itself the root by executing

$$parent_p \leftarrow \perp \qquad root_p \leftarrow p \qquad dist_p \leftarrow 0,$$

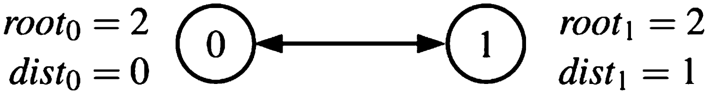
if it does not yet consider itself the root and detects an inconsistency in the values of its local variables or between these values and those of its parent:

- $root_p \leq p$; or
- $parent_p = \perp$; or
- $parent_p \neq \perp$, and $parent_p$ is not a neighbor of p or $root_p \neq root_{parent_p}$ or $dist_p \neq dist_{parent_p}+1$.

Note that if there is a cycle in the spanning tree, then a process in this cycle will always declare itself the root. Because there is always some process p in this cycle with $dist_p \neq dist_{parent_p} + 1$.

A process p that considers itself the root can make a neighbor q its parent if $root_q$ is larger than p. If several neighbors of p have root values greater than $root_p$, p selects a neighbor q with the largest root value among all p's neighbors. Before p can make q its parent, p must wait until q's component of the spanning tree has a proper root, for otherwise processes could infinitely often join a component of the spanning tree with a false root, as shown in the next example.

Example 17.5 Consider the following initial configuration.



Since $dist_0 \neq dist_1 + 1$, process 0 declares itself the root: $parent_0 \leftarrow \perp$, $root_0 \leftarrow 0$, and $dist_0 \leftarrow 0$. Next, since $root_0 < root_1$, process 0 makes process 1 its parent: $parent_0 \leftarrow 1$, $root_0 \leftarrow 2$, and $dist_0 \leftarrow 2$. Next, since $dist_1 \neq dist_0+1$, process 1 declares itself the root: $parent_1 \leftarrow \perp$, $root_1 \leftarrow 1$, and $dist_1 \leftarrow 0$. Next, since $root_1 < root_0$, process 1 makes process 0 its parent: $parent_1 \leftarrow 0$, $root_1 \leftarrow 2$, and $dist_1 \leftarrow 3$. And so on.

Therefore, before p makes q its parent, it first sends a join request to q, which is forwarded through the spanning tree to a root, which sends back an acknowledgment to p via the spanning tree. When p receives this acknowledgment, p makes q its parent by executing

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1.$$

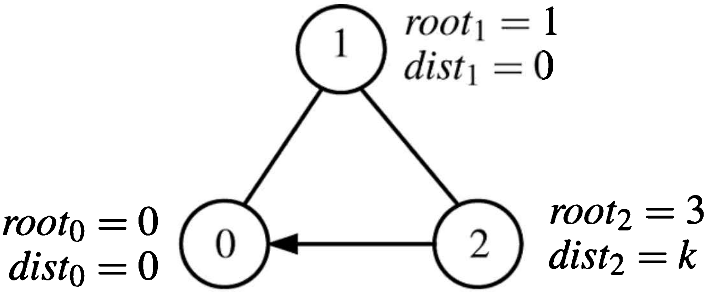
Since we are in a shared-memory framework, join requests and acknowledgments need to be encoded in shared variables; see the pseudocode in the appendix. The path of a join request is remembered in local variables, so the resulting acknowledgment can follow this path in the reverse order. A process can be forwarding and awaiting an acknowledgment for at most one join request at a time. As the encoding of join requests in shared variables is rather involved, they are presented in the examples in a message-passing style.

Example 17.6 We revisit the initial configuration from example 17.5 but now with join requests and acknowledgments. We consider one possible computation of the Afek-Kutten-Yung algorithm.

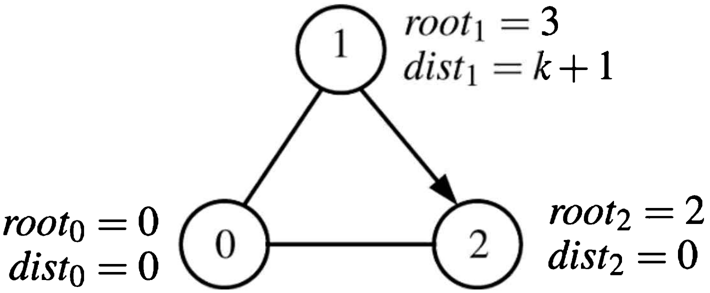
Since $dist_0 \neq dist_1 + 1$, process 0 declares itself the root: $parent_0 \leftarrow \perp$, $root_0 \leftarrow 0$, and $dist_0 \leftarrow 0$. Next, since $root_0 < root_1$, process 0 sends a join request to process 1. Note that process 1 cannot forward this join request to its parent 0, because 0 is awaiting an acknowledgment. Next, since $dist_1 \neq dist_0 + 1$, process 1 declares itself the root: $parent_1 \leftarrow \perp$, $root_1 \leftarrow 1$, and $dist_1 \leftarrow 0$. Since process 1 is now a proper root, it replies to the join request of process 0 with an acknowledgment. As a result, process 0 makes process 1 its parent: $parent_0 \leftarrow 1$, $root_0 \leftarrow 1$, and $dist_0 \leftarrow 1$. The resulting spanning tree, with process 1 as root, is stable.

Join requests are forwarded only between processes for which the local variables have consistent values; otherwise there could be infinite computations, as shown in example 17.7. And processes forward an acknowledgment only if they sent a corresponding join request previously. This check avoids spurious acknowledgments caused by improper initial values of local variables.

Example 17.7 Consider an undirected ring with three processes 0, 1, and 2. Initially, processes 0 and 1 consider themselves the root, while process 2 has process 0 as its parent, considers the (nonexistent) process 3 the root, and has some distance value *k*.



Since $root_2 > root_1$ (and $root_2 > root_0$), process 1 sends a join request to process 2. Without the consistency check, process 2 would forward this join request to process 0. Since process 0 considers itself the root, it would send back an acknowledgment to process 1 (via process 2), and process 1 would make process 2 its parent and consider process 3 the root. Next, since $root_2 \neq root_0$, process 2 could make itself the root.



Now we would have a configuration (nearly) symmetrical to the initial one. This scenario could be repeated to obtain an infinite computation that never reaches a stable configuration.

We briefly argue that the Afek-Kutten-Yung spanning tree algorithm is self-stabilizing. Each component in the network with a false root contains an inconsistency, so a process in this component will declare itself the root. Since join requests are forwarded only between consistent processes, and processes can be involved in only one join request at a time, each join request is eventually acknowledged. Join requests guarantee that processes only finitely often join a component with a false root, each time caused by improper initial values of local variables. These observations together imply that eventually false roots will disappear. Therefore, the process with the largest ID in the network will declare itself the root, and the network will converge to a spanning tree with this process as the root.

Bibliographical notes

Dijkstra's token ring originates from [27]; a proof that the ring is self-stabilizing for $K = N - 1$ is presented in [36]. The Arora-Gouda algorithm stems from [5], and the Afek-Kutten-Yung algorithm was proposed in [1].

17.4 Exercises

Exercise 17.1 Give a computation of Dijkstra's token ring with $N = K = 4$ that takes 13 events to reach a correct configuration.

Exercise 17.2 Argue that from each configuration of Dijkstra's token ring (with $K \geq N$) a correct configuration will be reached in at most $O(N^2)$ transitions.

Exercise 17.3 Consider an undirected ring of three processes with IDs 0, 1, and 2. In the initial configuration, $parent_0 = 1$, $parent_1 = 2$, and $parent_2 = 0$; $root_0 = root_1 = root_2 = 3$; and $dist_0 = 1$, $dist_1 = 0$, and $dist_2 = 2$. Describe one possible computation of the Arora-Gouda algorithm on this network, with $K = 4$.

Exercise 17.4 One part of the Arora-Gouda algorithm considers a neighbor q of process p with $dist_q < K$. Show that if the side condition "with $dist_q < K$" were omitted, then the algorithm might not stabilize.

Exercise 17.5 Give an unfair infinite computation of the Arora-Gouda algorithm that never stabilizes. Let only one process perform events.

Exercise 17.6 Adapt the Arora-Gouda algorithm so that it no longer exhibits (unfair) infinite computations.

Exercise 17.7 Describe one possible computation of the Afek-Kutten-Yung algorithm on the network from exercise 17.3.

Exercise 17.8 Argue that in the Afek-Kutten-Yung algorithm, each join request eventually results in an acknowledgment.

Exercise 17.9 Argue that the Afek-Kutten-Yung algorithm takes at most $O(N^2)$ transitions to stabilize.

18
Security

Distributed computer systems can be vulnerable to hostile attacks. An eavesdropper may secretly listen to a private message exchange, or an unauthorized intruder may steal or modify parts of a database. An information security system aims to prevent such attacks and provide confidentiality, integrity, and availability of messages and data. Confidentiality means that information is never disclosed to unauthorized entities. Integrity means that the accuracy and completeness of information are preserved at all times. Availability means that information is accessible when needed.

A wide range of standard attacks exist to try and violate one of these properties. Confidentiality can, for example, be threatened by a spoofing attack in which the attacker masquerades as another. A particular instance is the man-in-the-middle attack, in which the attacker secretly relays messages between two parties which believe they are communicating directly with each other. Integrity may be violated by an unauthorized intruder who manages to obtain write access in a database. Availability can be threatened by a denial-of-service attack, in which a large flood of simultaneous messages, usually sent from many sources, forces the target system to shut down.

This chapter presents some methods to defend a distributed system against certain kinds of hostile attacks. First, a few standard techniques are discussed, followed by two specific security protocols. A blockchain, which underlies the bitcoin protocol for secure financial transactions in peer-to-peer systems, exemplifies how consensus can be reached in an open environment if a significant majority of peers is trustworthy. Quantum cryptography protects channels from eavesdropping through the use of quantum mechanics.

18.1 Basic Techniques

This section presents some important mechanisms for building secure distributed systems. Cryptographic hash functions, which cannot be inverted by an attacker, are an essential tool for authentication. A public-key cryptosystem serves as a backbone for secure communication. And proof-of-work requires that a simple task, such as sending an email, may be performed only after spending a huge amount of effort solving some meaningless puzzle. The purpose is to deter prospective attackers.

Cryptographic Hash Function

A *hash function* maps elements from a large and possibly heterogeneous data domain D to a usually much smaller data domain E in which all elements have a fixed length. Computing the hash value of an element in D is required to be easy. Originally hash functions were developed to fit elements from a vast data domain into a table of fixed size and perform fast lookups. A *cryptographic* hash function $h : D \to E$ has an important additional property:

- *Collision resistance*: It is very hard to find different $d, d' \in D$ with $h(d) = h(d')$.

As a consequence, cryptographic hash functions also satisfy the following property:

- *Preimage resistance*: Given an $e \in E$, it is very hard to find a $d \in D$ with $h(d) = e$.

One important application of cryptographic hash functions is that the passwords for user authentication are not stored but their hash values are stored instead. When a password is entered by a user, its hash value is compared with the corresponding value in the database. The chance that an erroneous password happens to have the same hash value as the correct one is minimal thanks to collision resistance. The big advantage of storing hashed passwords is in the unfortunate circumstance of a security breach in which an intruder secretly steals the database of passwords. Thanks to preimage resistance, the attacker will still have a very hard time finding an acceptable password. Actually, since passwords tend to be relatively short, hash values of what are called salted passwords are stored, meaning that some (often random) piece of data is attached to a password before the hash value is computed; the salt is stored in combination with the hash value of the salted password. This defends against dictionary attacks in which hash values of large numbers of possible passwords are precomputed by the attacker. Other applications of cryptographic hash functions will be discussed in the remainder of this chapter.

Developing collision-resistant hash functions, and checking that cryptographic hash functions used in practice possess this property, has turned out to be a huge challenge. For example, the widely employed functions SHA-0 and MD5 were in the end shown to have serious vulnerabilities and fell from grace.

The birthday attack aims to break collision resistance by exploiting the birthday paradox: after generating the hash values of approximately $\sqrt{|E|}$ randomly selected elements in D, where $|E|$ denotes the number of elements in E, one can expect to have found some collision. The name of this paradox refers to the fact that as a consequence the probability of two persons in a room sharing the same birthday increases surprisingly quickly with the total number of people in the room. A widely used cryptographic hash function like SHA-256 hashes to bit strings of length 256, meaning that the birthday attack requires roughly 2^{128} hash values before one can expect to find a collision, which is currently considered a safe margin.

In the *Merkle tree* data structure, the leaves of the tree are data blocks, while each nonleaf carries the hash value of its children. The Merkle root at the top of the tree is a fingerprint of the data blocks in the leaves. If a cryptographic hash function is employed, then the Merkle root can be used to verify the integrity of the data blocks in the leaves of the corresponding Merkle tree. Suppose a collection of large data blocks is obtained from an untrusted source in the form of a Merkle tree, and its Merkle root can also be acquired from a trusted source or from a large number of sources, of which the vast majority are honest. If the Merkle root of the tree is found to be correct, the data is uncorrupted. An advantage of the tree structure, in comparison to the linear hash list data structure, is that branches of the tree can be downloaded and verified individually.

Public-Key Cryptography

The field of cryptography develops techniques for secure communication, primarily through the use of encryption techniques that turn sensible data into what appears to be nonsense by means of an encryption key. An eavesdropper should be unable to reconstruct the original data from the resulting encrypted data without knowing the decryption key. Traditionally cryptography was mostly based on symmetric-key encryption schemes, where the encryption and decryption keys are identical or can be turned into each other by a simple transformation. A classical simplistic example is a technique employed by Julius Caesar, who encrypted secret letters by shifting each letter three places upward in the alphabet. A vulnerability of symmetric-key cryptography is that all intended recipients of secret information need to know the decryption key and therefore can derive the encryption key. A hostile intruder may secretly manage to steal the decryption key from a trusted recipient, or such a recipient may have bad intentions and abuse the encryption key.

In public-key cryptography, which was already discussed in section 13.4, an entity makes its encryption (also called public) key openly available to everybody but keeps its decryption (or private) key secret. Vital for the robustness of this approach is that although the public and private keys are each other's inverses, it should be infeasible to reconstruct the private key from the public one. To send a secret message to an entity, the sender applies the public key of that entity to the message beforehand. After reception, the entity applies its private key to obtain the original message. An eavesdropper can do nothing with the encrypted message without knowing the corresponding private key.

New entities in the network create a fresh public and private key pair and publish the public key by means of a trusted public key server. A trusted certificate authority links entities to public/private key pairs and upon request dispenses public keys of entities, encrypted with the private key of the certificate authority. The recipient can decrypt such a message by using the public key of the certificate authority. This prevents an attacker from spreading bogus public keys.

A private key can also be used to provide a message with a digital signature, which serves as an authentication of the sender and makes it impossible for the sender to deny that it has sent the message. The signature consists of the private key of the sender applied to the message, which is tagged to the message itself. The receiver can verify the signature by applying the public key of the sender to the signature and comparing it to the message; if they are equal, the signature is correct. This scheme is, however, vulnerable to what is called a key-only attack. To create a forgery with the aim of impersonating another entity, the attacker picks a random signature and applies the entity's public key to compute the message belonging to that signature. Then the attacker sends the computed message together with the signature. To avoid this type of attack, first a publicly known cryptographic hash function is applied to the message to be signed, and then the private key is applied to the resulting hash value. The receiver applies the sender's public key to the signature, applies the hash function to the message, and compares the two outcomes. The key-only attack is rendered ineffective because the attacker can only determine the hash value of the corresponding message from a signature, while it is required to send the original message along with the signature. A bonus of this adapted digital signature scheme is that the hashed message is usually much shorter than the message itself, so applying the private key tends to require less effort.

The *RSA cryptosystem* yields a public and a private key that are each other's inverses with the desired property that computing the private key from the public key is hard. It exploits a basic equality from number theory. For all prime numbers p and q, positive natural numbers m with $m = 1 \bmod (p-1)\cdot(q-1)$, and integers k,

$$k^m = k \bmod p\cdot q\,.$$

The message domain consists of (or better, is encoded into) numbers modulo $p\cdot q$. To build public and private keys, very large prime numbers p and q are chosen, as well as a positive number e that is relatively prime with $(p - 1)\cdot(q - 1)$, meaning that these two numbers do not have any prime factors in common. Then a positive number d is determined such that $d\cdot e = 1 \bmod (p - 1)\cdot(q - 1)$. The number e and the product $p\cdot q$ are made public; the public key applied to a message n produces $n^e \bmod p\cdot q$. On the other hand, the numbers d, p, and q are kept secret; the private key applied to a message n produces $n^d \bmod p\cdot q$. Note that applying both the public and the private keys to a message n, irrespective of the order in which they are applied, yields $n^{d\cdot e} = n \bmod p\cdot q$ by the preceding equation. So these keys are each other's inverses.

Since p and q are very large, the public information consisting of e and $p\cdot q$ is insufficient to efficiently compute d. For this purpose, one first must find the prime factors p and q from their product. No efficient algorithm for prime factorization is known. The only way found so far to break RSA uses a quantum computer based on quantum mechanics. However, the quantum computers built so far have only very limited processing power and thus do not yet pose a serious threat to RSA.

Elliptic curve cryptography is an alternative method for developing a public-key cryptosystem, based on more advanced mathematical notions, particularly elliptic curves over finite fields. This technique is nowadays widely used in practice, because it requires significantly smaller keys compared to, say, RSA to reach the same level of security.

Proof-of-Work

A *proof-of-work* is a puzzle that takes a considerable amount of processing power to solve, while it is easy to verify that a proposed solution is correct. Ideally, the puzzle has many possible solutions and can only be solved by blindly guessing and trying out one possible solution after the other until a correct solution is found. A solution to such a puzzle is typically not interesting in its own right; the puzzle is only posed to dissuade attackers from performing a vast number of small tasks with bad intent, such as email spam or a denial-of-service attack. The puzzle is generally related to the task being performed. For example, to deter email spammers, for each email sent a proof-of-work could be required on the basis of the email header. If this puzzle takes a few seconds to solve, it will not trouble legitimate emailers, while spammers will have difficulty generating the required solutions on a massive scale because this requires too much time and computational power.

Another important application of proof-of-work, as we will see in the next section, is in an open network in which not all nodes can be trusted and it is easy to spawn new nodes. An attacker could create many alias nodes that are all under its control and in this way, for instance, dominate some majority vote. A defense against such a so-called Sybil attack is to require a significant proof-of-work from each voter. Then an attacker can only bully the network if it possesses a vast amount of processing power compared to the honest nodes in the network. Ideally, the number of correct solutions in the entire space of possible solutions of a puzzle can be adapted easily to increase the difficulty of finding a solution if puzzles are being solved at a faster rate, typically because more processing power has become available in the network.

18.2 Blockchains

In the world of finance, a ledger is a book in which all monetary transactions of an organization are noted. In the context of computer science, this book is a data file. A public (or permissionless) *blockchain* is an openly accessible ledger that is stored as a decentralized distributed database. It can be used to keep track of ownership of assets. A blockchain consists of a continuously growing linked list of blocks, each containing a batch of validated transactions, as well as a link to its predecessor (except for the Genesis block at the start of the list). The blockchain is supposed to contain only legitimate financial transactions, which are temporally ordered by their places in the chain. Anybody can try to add a new block at the end of the list. The main challenge is to prevent a malicious attacker from tampering with the blockchain or adding fraudulent transactions.

The database is distributed over a *peer-to-peer network* in which peers (i.e., processes) can freely join and leave. Some peers keep track of the entire blockchain; updates are shared among those peers. Other peers may be content with a local view that contains sufficient information to validate new transactions without keeping the entire history of the blockchain in memory. When a peer (re)joins the network, it downloads and verifies (newly added) blocks from other peers to obtain (or update) its local copy of the blockchain or its local view. In contrast to the client-server architecture, data is replicated many times, and each change to the blockchain creates communication overhead. Paradoxically, this decentralized and replicated nature of a blockchain is essential not only for its accessibility but also for its security, since there is no single target for an attacker.

The remainder of this section discusses the *bitcoin* protocol for financial transactions over the Internet, which employs a blockchain to achieve secure, tamper-resistant consensus between peers. The possible applications of blockchains stretch far beyond mere financial transactions, as will be discussed briefly at the end of this section. However, the bitcoin protocol provides an insightful illustration of their potential as well as the challenges one is faced with when using blockchain technology.

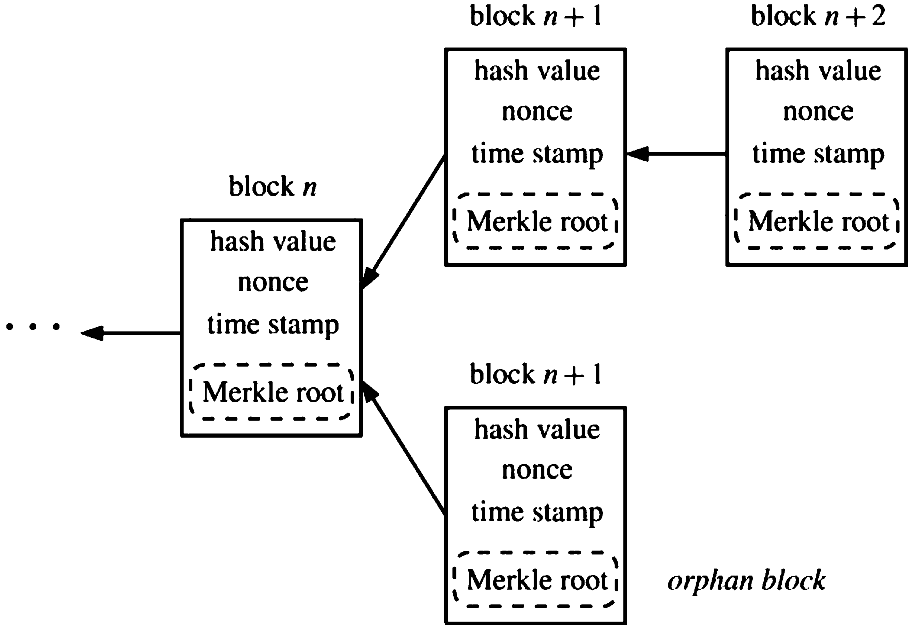
The bitcoin protocol employs cryptographic hash functions (notably SHA-256). A peer controls one or more addresses, each associated with a public/private key pair; the address consists of a cryptographic hash value of the public key. In contrast to traditional bank accounts, reusing the same address for different transactions is considered an undesirable and unnecessary security risk. A transaction collects *cryptocurrency* called bitcoins from one set of addresses and attributes it to (and divides it over) another set of addresses. It takes as inputs the outputs of earlier transactions and protects each of its outputs with a hash of the public key belonging to the destination address, which ensures that the cryptocurrency attributed to this address can be spent only by the peer controlling this address. The output of a transaction can serve as input to a subsequent transaction only once, so to retain part of this cryptocurrency, the payer should in the subsequent transition attribute this part to an address under its own control. Otherwise the unspent bitcoins would be attributed to the miner that adds this transaction to the blockchain; this will be explained in the next two paragraphs.

A transaction is broadcast to all peers in order to be included in a block and then added to the blockchain. Upon receiving a new transaction, a peer validates that all financial contributors to the transaction agree and that their contributions have not been spent previously. The peer verifies the chain of ownership, either directly by using its local view or indirectly through another trusted peer. Published transactions cannot be retracted and are in principle irreversible once they have been included in the blockchain. This eliminates the risk of chargeback fraud, in which a customer claims a refund (typically for a credit card transaction) without returning the purchased goods.

A *miner* collects validated transactions and bundles them in a block. To add this block at the end of the blockchain, a miner must complete a proof-of-work, which will be described later. When a miner succeeds in adding its block to the blockchain, it broadcasts the block to each peer, which accepts the block as the new head of the blockchain only after validating the proof-of-work and that all transactions in the block are valid. This miner is allowed to reward itself with some newly created bitcoins, which also serves the purpose of disseminating fresh bitcoins in a decentralized fashion. Mining requires hard labor and slowly makes new cryptocurrency available, reminiscent of extracting valuable minerals from the earth. The reward for adding a block is included in the block itself as what is called a coinbase transaction, which creates coins from nothing and so has no ancestor. This reward (12.5 bitcoins since October 2016) is halved with each 210,000 blocks, which boils down to roughly once every four years. Eventually (around the year 2140) it will be set to zero, once the maximum number of 21 million bitcoins has been reached. Ownership of bitcoins is defined by sequences of digitally signed transactions originating at their creation as a mining reward in coinbase transactions. A voluntary transaction fee, which consists of the unspent bitcoins in a transaction, is paid to a miner for including the transaction in a block and adding it to the blockchain. Miners naturally tend to select pending transactions with higher fees. When all bitcoins have been mined, transaction fees will become even more important to incentivize miners. Currently the transaction fee tends to be around 0.0005 bitcoin. (The smallest unit is 0.00000001 bitcoin, called a satoshi in honor of the anonymous developer(s) who published the bitcoin protocol under the pseudonym Satoshi Nakamoto.)

A fork in the blockchain may occur when two miners complete the proof-of-work and add a block at almost the same time, as depicted in the diagram that follows.

Miners will continue to append new blocks to the longest chain they are aware of, meaning the one that required the largest amount of effort to produce. As a result, one of the forks will quickly prevail, and the so-called orphan blocks in the unlucky shorter fork will be ignored. A dishonest miner can attempt to deliberately create a long fork of rogue blocks that starts from an older block and thus turn genuine blocks in the blockchain into orphans. This would, for instance, allow the miner to spend the same cryptocurrency twice: first in an orphan block and then in a rogue block. For this reason, mining is intentionally resource-intensive, and the proof-of-work chains a block to its predecessor in the blockchain through cryptographic means, as we will explain. As long as a significant majority of the computing power is controlled by honest miners, a fork of rogue blocks will quickly be outpaced by the fork of honest blocks. Likewise, to modify a past block, an attacker must redo the proof-of-work of that block and all its successors and then surpass the work of the honest miners. The probability of an attacker being able to catch up diminishes exponentially as subsequent blocks are added.



We now dive into the intricacies of the proof-of-work for miners. Next to the body, which consists of its transactions stored in a Merkle tree, a block also carries a header with the following components: the hash value of the header of the previous block in the chain, the Merkle root of its own body, a time stamp based on real time, and a nonce that is allowed to carry any 32-bit value. The first component of the header of a block firmly chains it to the previous block in the blockchain. The second component guarantees that the body of a block cannot be changed after it has been added to the blockchain. The third component, which affirms the ordering of the blocks in the blockchain through increasing time stamps, makes manipulation of the blockchain by an attacker even more difficult. The demanding proof-of-work required from a miner who wants to add its block at the head of the blockchain is to find a nonce for the fourth component such that the hash value of the header of this block starts with many (typically 40) zeros. This results in a guessing game where approximately one out of every 2^{40} nonces yields a solution to this puzzle that allows the miner to add its block. The nonce starts at zero and is increased in a linear fashion until either a solution is found or it overflows, which will happen often because the nonce is small compared to the difficulty of the puzzle. In case of an overflow, an extra nonce field within the coinbase transaction of the block is incremented and the nonce in the block header is restarted at zero. Incrementing the extra nonce field entails recomputing part of the Merkle tree in the body of the block because it changes the coinbase transaction. To be more precise, the difficult puzzle concerns the double hash value of the block header for increased security, although currently no attack on the bitcoin protocol is known even in the case of a single hash.

The difficulty of the puzzle is calibrated based on the speed with which new blocks are being added, to keep the average at approximately once every 10 minutes. If more processing power becomes available so that the time between adding new blocks decreases, then the system increases the required number of leading zeros of the double hash value of a header; the average work required to solve a puzzle grows exponentially with every additional leading zero bit. If, on the other hand, miners decide to quit their operations, typically because they are no longer profitable, then the required number of leading zeros is decreased. In this way, the bitcoin protocol effectively controls the cryptocurrency supply.

Since mining is something of a gamble and requires a lot of processing power to have a serious chance of success, many miners have organized themselves into pools that share resources as well as rewards, usually under some centralized form of coordination. For the security of the bitcoin protocol, it is essential that no pool controls a large part of the hashing power in the network. Currently the vast majority of blocks are mined by a limited number of pools. If these pools join together, the longest chain rule would no longer constitute an effective defense mechanism against adding rogue blocks. Since in the bitcoin protocol mining rewards decrease over time, the incentive to mine may diminish, which makes the threat of a mining monopoly in the future more imminent.

Because of proof-of-work, mining bitcoins consumes huge amounts of power, which is a waste of energy and money. Alternatively, a *proof-of-stake* can be used to decide which new block is added at the head of the blockchain. The basic idea is that a peer's voting power in adding a new block is proportional to the amount of cryptocurrency this peer holds. The cryptocurrency added to the system with each new block is divided among all peers in the same proportional manner, which effectively leads to a steady inflation. An important additional advantage of proof-of-stake is that it mitigates the risk of a monopoly, because an attacker needs to own a majority of the available cryptocurrency instead of the available processing power. A peer is free to vote for both branches of a fork and could maliciously try to spend the same cryptocurrency in both branches. However, such an attack only has a serious chance of success if the peer controls a large amount of cryptocurrency, in which case he actually has a strong interest in maintaining the integrity of the system. A naive implementation of proof-of-stake, however, provides little incentive to validate new blocks and a perverse incentive to stock up on cryptocurrency. Therefore, it needs to be equipped with incentives that counterbalance these factors, possibly based on some form of proof-of-work.

The application of blockchains goes beyond mere financial transactions. A blockchain can be endowed with more complex operations, in particular, instructions from a programming language. A computer program can thus be performed by the blockchain, which ensures that the execution is tamper resistant and recorded permanently in the blockchain. This can, for instance, serve to run a smart contract that automatically verifies and enforces the terms of a binding legal agreement by using cryptographic means. The program can perform a specific action, such as the transfer of cryptocurrency, taking into account the terms of the agreement, actions by the contract partners, and input data provided by trusted peers. For example, a smart contract could regulate car insurance policies, particularly the settlement in the case of an accident.

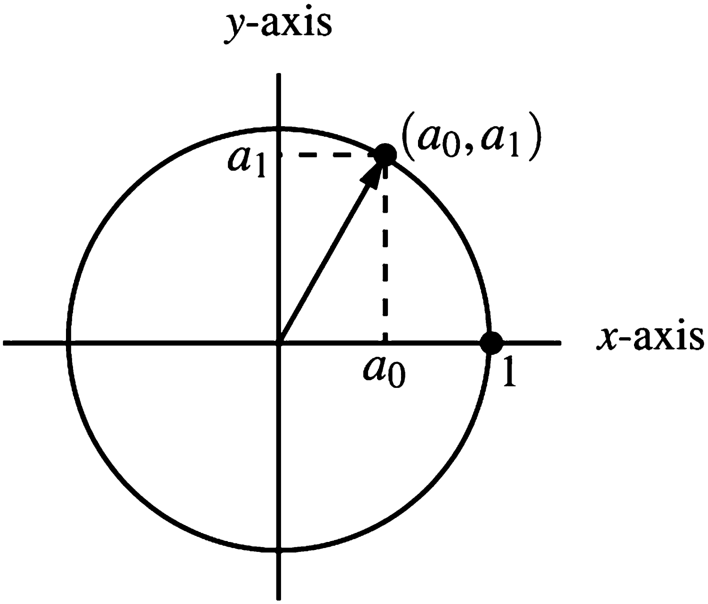
18.3 Quantum Cryptography

The Huygens-Fresnel principle states that every point in space that is reached by a wave becomes a source of a spherical wave. This effect, for example, becomes apparent if a wave in the water encounters a wall with a small hole. At the other side of the wall, the wave continues in a semicircular shape, with the hole at its center. If there are two holes close to each other in the wall, the two resulting semicircular waves may together form a standing wave in which the two subwaves at some places amplify and at other places cancel each other. Christiaan Huygens predicted in 1678 that light behaves like a wave. Thomas Young confirmed this claim in a famous double-slit experiment in 1805 showing that a light beam passing through two thin slits may on the wall behind it form the pattern of a standing wave consisting of an alternating sequence of brightly lit and dark patches. A paradox, known as wave-particle duality, is that light consists of individual elementary particles called photons that according to the double-slit experiment exhibit wave properties by themselves. On the other hand, if one starts to measure which of the two slits a photon passes through, the interference effect of the double-slit experiment disappears and light passes through the slits in a straight line, forming two bright stripes on the wall behind it.

It turns out that an elementary particle can be in a range of possible states at the same time with a certain probability distribution. This is called a *superposition.* Interaction with an observer causes the particle to fall back to one of these possible states. A *qubit*, short for quantum bit, is a binary bit that may be in superposition, basically meaning that it is undecided between 0 and 1. It is represented by the expression

$$\alpha_0 |0\rangle + \alpha_1 |1\rangle.$$

In physics, α_0 and α_1 may be complex numbers (like i with $i^2 = -1$), but for our purpose it suffices to restrict them to real numbers, written as a_0 and a_1. These values indicate what may happen when the qubit falls back from its superposition to a classical deterministic state: with probability a_0^2 it becomes 0, and with probability a_1^2 it becomes 1. Since these two probabilities must sum to 1, the pair (a_0, a_1) constitutes the coordinates of a point on the unit circle in the two-dimensional plane $\mathbb{R}^2$, which we represent as a vector $\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$.



Although the numbers a_0 and a_1 are associated with (nonnegative) probabilities, it is essential that they can be negative, because the operations on qubits that we will now explain range over the entire unit circle.

A *no-cloning theorem* states that it is impossible to create an identical copy of an elementary particle in an unknown superposition. The underlying idea is that to make a copy, the particle needs to be measured, which inevitably influences the state the particle is in. The no-cloning theorem is fundamental to the BB84 key distribution protocol that will be explained in this section. An eavesdropper cannot make identical copies of qubits sent through a quantum channel, but is forced to affect the information flow through the channel while measuring qubits that are in superposition.

Quantum Operations

Controlled quantum operations can be performed on an elementary particle that transform it from some superposition into another superposition. These are linear operations that can be expressed by a 2 × 2 matrix. After the transformation, probabilities must of course still sum to 1; in other words, the linear operation must map the unit circle in two-dimensional space onto itself. Furthermore, quantum operations are always invertible. A quantum operation (restricted to $\mathbb{R}^2$) can be expressed by what is called an orthogonal matrix.

We start with some basic linear algebra. The two-dimensional plane $\mathbb{R}^2$ consists of vectors $\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$ with a_0, $a_1 \in \mathbb{R}$, representing the point in $\mathbb{R}^2$ with *x*-coordinate a_0 and *y*-coordinate a_1. Addition of two vectors is defined by separately adding the *x*-coordinates and the *y*-coordinates: $\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} a_0 + b_0 \\ a_1 + b_1 \end{pmatrix}$. Likewise, multiplication of a vector with a $c \in \mathbb{R}$ is defined at the level of coordinates: $c \cdot \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} c \cdot a_0 \\ c \cdot a_1 \end{pmatrix}$.

A 2 × 2 *matrix* is a linear operator that maps $\mathbb{R}^2$ to itself. By linearity, it suffices to define such a matrix *M* only on the base vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ of $\mathbb{R}^2$: the *M*-image of a general vector $\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$ in $\mathbb{R}^2$ is defined by $a_0 \cdot M\begin{pmatrix} 1 \\ 0 \end{pmatrix} + a_1 \cdot M\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. A 2×2 matrix therefore has the form $\begin{pmatrix} a_0 & b_0 \\ a_1 & b_1 \end{pmatrix}$, where the first column $\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$ is the image of the first base vector and the second column $\begin{pmatrix} b_0 \\ b_1 \end{pmatrix}$ is the image of the second base vector.

Example 18.1 *Consider the* 2 × 2 *matrix* $\begin{pmatrix} 1 & 3 \\ 2 & -1 \end{pmatrix}$. *It maps the first base vector* $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ *to the* *first column* $\begin{pmatrix}1\\2\end{pmatrix}$ *and the second base vector* $\begin{pmatrix}0\\1\end{pmatrix}$ *to the second column* $\begin{pmatrix}3\\-1\end{pmatrix}$ *of the matrix. So, for instance, the vector* $\begin{pmatrix}2\\-1\end{pmatrix}$ *is by this matrix mapped to* $2\cdot\begin{pmatrix}1\\2\end{pmatrix}-1\cdot\begin{pmatrix}3\\-1\end{pmatrix}=\begin{pmatrix}2\\4\end{pmatrix}+\begin{pmatrix}-3\\1\end{pmatrix}=\begin{pmatrix}-1\\5\end{pmatrix}.$

As noted, after performing a quantum operator, the probability mass of all possible states of a superposition must still sum to 1. Therefore, quantum operations (restricted to $\mathbb{R}^2$) are associated with an *orthogonal* matrix, which leaves the length of a vector and the angle between a pair of vectors in $\mathbb{R}^2$ unchanged. In other words, for a 2 × 2 orthogonal matrix, the two base vectors of $\mathbb{R}^2$ are mapped to an orthogonal unit basis: vectors of length 1 that are orthogonal to each other, meaning that they have an angle of (plus or minus) 90 degrees. That is, a matrix $\begin{pmatrix}a_0 & b_0\\a_1 & b_1\end{pmatrix}$ is orthogonal if $a_0^2+a_1^2=1$ (the image of the first base vector has length 1), $b_0^2+b_1^2=1$ (the image of the second base vector has length 1), and $a_0\cdot b_0 + a_1\cdot b_1 = 0$ (these two images are orthogonal to each other). Clearly, the matrix of any rotation around the origin (0, 0) of $\mathbb{R}^2$ is orthogonal, and likewise for a reflection in a line through the origin. Actually, each orthogonal 2 × 2 matrix with coefficients in $\mathbb{R}$ is a composition of such rotations and reflections.

Let us now return to qubits. You need to make a mental leap in considering the states $|0\rangle$ and $|1\rangle$ as the base vectors of a two-dimensional space. A quantum operation associated with an orthogonal 2 × 2 matrix acts on a qubit by considering it a vector on the unit circle in $\mathbb{R}^2$. That is, a quantum operation $\begin{pmatrix}a_0 & b_0\\a_1 & b_1\end{pmatrix}$ transforms $|0\rangle$ to $a_0|0\rangle + a_1|1\rangle$ and $|1\rangle$ to $b_0|0\rangle + b_1|1\rangle$. Since quantum operations are linear, a qubit in a superposition $c_0|0\rangle + c_1|1\rangle$ is transformed to $c_0\cdot(a_0|0\rangle + a_1|1\rangle) + c_1\cdot(b_0|0\rangle + b_1|1\rangle) = (c_0\cdot a_0 + c_1\cdot b_0)|0\rangle + (c_0\cdot a_1 + c_1\cdot b_1)|1\rangle$.

The *Hadamard transform* is the quantum operation defined by the orthogonal matrix $\begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix}$, which is also written as $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$. Note that the minus sign in front of the bottom right coefficient is essential for orthogonality. The Hadamard transform brings $|0\rangle$ as well as $|1\rangle$ in a superposition where the outcomes 0 and 1 are equally likely. Because $|0\rangle$ is transformed into $\frac{1}{\sqrt{2}}|0\rangle+\frac{1}{\sqrt{2}}|1\rangle$ and $|1\rangle$ into $\frac{1}{\sqrt{2}}|0\rangle-\frac{1}{\sqrt{2}}|1\rangle$.

Example 18.2 Let us apply two consecutive Hadamard transforms on a qubit that is in the (deterministic) state $|0\rangle$ or $|1\rangle$.

If it is in state $|0\rangle$, then after the first application it is in the superposition

$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. So the second application results in $\frac{1}{\sqrt{2}} \cdot (\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle) + \frac{1}{\sqrt{2}} \cdot (\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle) = |0\rangle$.

If it is in state $|1\rangle$, then after the first application it is in the superposition $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. So the second application results in $\frac{1}{\sqrt{2}} \cdot (\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle) - \frac{1}{\sqrt{2}} \cdot (\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle) = |1\rangle$.

In either case, after performing the two Hadamard transforms, the qubit has returned to its original state. In other words, the Hadamard transform is its own inverse.

BB84 Key Distribution Protocol

Suppose two peers, called Alice and Bob, want to construct a private key that is known only to them. The BB84 protocol allows them to do this in a secure manner by using qubits. The beauty of this key distribution protocol is that, owing to the nature of superpositions, an eavesdropper cannot avoid disturbing the information flow between Alice and Bob, which will inevitably thwart each attempt to overhear the private key. The price to pay is that an eavesdropper may cause such a serious perturbation of the information flow that the attempt by Alice and Bob to build a private key may fail. Moreover, Bob needs to employ an error correction algorithm at the end, because a casual eavesdropper may have introduced a small aberration in a (successfully computed) private key.

In the BB84 key distribution protocol, Alice starts with a string of n qubits from which the private key will be constructed. Initially, each qubit is either in state $|0\rangle$ or in state $|1\rangle$; for each qubit, this value is chosen randomly. Alice sends each of the n qubits to Bob via a public quantum channel. For each of the qubits, she randomly chooses to send the qubit either unchanged or after performing the Hadamard transform on it. Likewise, for each of the n received qubits, Bob randomly chooses to measure it either in the state in which it was received or after performing the Hadamard transform on it. If neither Alice nor Bob performed the Hadamard transform on a qubit, then clearly Alice and Bob read the same value for this qubit. And if both Alice and Bob performed the Hadamard transform on a qubit, then again they read the same value, as was shown in example 18.2. Since both of them perform their Hadamard transforms in a random fashion, it is expected that on about half of the n qubits they can be certain that their values agree. Alice and Bob inform each other, via a public classical channel, which qubits they performed the Hadamard transform on. The values of the (on average, $\frac{n}{2}$) qubits on which both Alice and Bob either did or did not perform the Hadamard transform constitute the private key.

The underlying idea of the protocol is that if an eavesdropper on the quantum channel overhears a qubit on which Alice performed the Hadamard transform, then it may introduce an error. Since the qubit cannot be cloned, it needs to be measured, which typically forces it into a deterministic state $|0\rangle$ or $|1\rangle$; in view of the definition of the Hadamard transform, each of these outcomes occurs with a 50 percent chance. To obtain a substantial amount of information with regard to the private key, eavesdroppers are bound to cause a significant perturbation in the values of the qubits read by Bob. To recognize such a perturbation, Alice sends half of the private key to Bob via the public classical channel. For each of these values, Bob checks whether it agrees with the value he read. If this is the case for all or almost all values, then (with very high probability) potential eavesdroppers have gained insufficient information for reconstructing the private key, so Alice and Bob can safely use the constructed private key. If this is not the case, then Alice and Bob must start a new attempt to build a private key.

A casual eavesdropper that measures a handful of qubits over the quantum channel may introduce only a few flaws, in which case Alice and Bob can still successfully compute a private key. Therefore, Bob employs an error correction algorithm at the end, which can repair the constructed private key if the error rate is below a certain threshold. The idea behind error correction is that the collection of possible private keys is sparse within the space of all bit strings of length at most n. If the computed private key is not in this collection, the error correction algorithm determines the “nearest” bit string that is a possible private key. An additional advantage of error correction is that the protocol can cope with sporadic errors that may be introduced during transmission because of noise on the quantum channel.

Example 18.3 Let $n = 8$. (In reality, of course, a much larger n must be taken in order to provide a high level of security and allow for error correction.) Let the random initial values of the successive qubits at Alice be 0 0 1 0 1 1 0 0. Alice randomly decides to apply the Hadamard transform on the first, third, and eighth qubits, and Bob on the second, third, seventh, and eighth qubits. After Alice has communicated to Bob which qubits she applied the Hadamard transform on, Bob determines that he is guaranteed to have measured the correct value for the third, fourth, fifth, sixth, and eighth qubits. So these values form the private key: 1 0 1 1 0.

Suppose now that an eavesdropper measures only the third qubit. Since Alice applied a Hadamard transform on this qubit, its superposition before this measurement is $\frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle$. Suppose that by the measurement it falls back to the deterministic state 0, so Bob computes an incorrect private key: 0 0 1 1 0. The first value in this key is corrected by Bob if it is part of the half of the private key that Alice sends to him. If this is not the case, then error correction must repair this flaw.

Bibliographical notes

Public-key cryptosystems and the RSA algorithm are credited to [26] and [81], respectively. (These two notions had actually already been invented some years earlier within the secret service of the United Kingdom but remained classified information until 1997.) Proof-of-work can be traced back to [32]. The bitcoin protocol was proposed in [70], and the BB84 protocol stems from [10].

18.4 Exercises

Exercise 18.1 Let each integer k be mapped to $k \bmod n$ for some positive natural number n. Argue that this hash function is neither collision resistant nor preimage resistant.

Exercise 18.2 Consider the RSA cryptosystem with $p = 5$, $q = 13$, and $e = 5$.

(a) Encrypt 7.
(b) Determine a value for d.
(c) Decrypt the value you computed in (a).

Exercise 18.3 Describe the ways in which cryptographic hash functions play a crucial role in the bitcoin protocol.

Exercise 18.4 Suppose all bitcoin miners, except one dishonest pool, take an extended Christmas break. Explain how this pool can (try to) spend the same cryptocurrency twice.

Exercise 18.5 Show that the Hadamard transform is composed of a rotation around the origin followed by a reflection in a line through the origin in $\mathbb{R}^2$.

Exercise 18.6 Let $n = 10$, and let 0 1 1 0 0 1 0 1 0 0 be the random initial values of the successive qubits at Alice. She randomly decides to apply the Hadamard transform on the second, third, fourth, and eighth qubits, while Bob applies it on the first, second, fourth, seventh, eighth, and ninth qubits.

(a) What is the private key computed by Alice and Bob?
(b) Suppose a malicious peer applies the Hadamard transform on all 10 qubits in the quantum channel before they reach Bob. What is the probability that Bob still determines the correct bit values for the entire private key?

Exercise 18.7 Explain for each of the following 2 × 2 matrices whether it is a possible replacement for the Hadamard transform in the BB84 protocol:

(a) $\frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$;

(b) $\frac{1}{\sqrt{2}}\begin{pmatrix} -1 & 1 \\ 1 & 1 \end{pmatrix}$;

(c) $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$;

(d) $\begin{pmatrix} \cos\frac{\pi}{4} & \sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & -\cos\frac{\pi}{4} \end{pmatrix}$.

19
Online Scheduling

So far, we have mostly ignored timing aspects. Logical clocks were used for rollback recovery, termination detection, and mutual exclusion; local clocks with bounded drift were employed to build a synchronous system in the presence of Byzantine processes; and time stamps helped to serialize distributed transactions. But these were abstract representations of time. In this chapter, we will consider jobs, meaning units of work, that need to be scheduled and executed and are for this purpose divided among the processors. These jobs have time constraints and resource requirements.

One important application of real-time computing is computer graphics in video games, where it is vital to produce and analyze images in real time, and there is very little time available per image. Typically, every image is then decomposed into triangles, and special hardware is employed to generate the pixels inside each of the triangles separately. Another important application is in air traffic control in order to direct planes on the ground and through the air, based on information from different sources such as radars, weather stations, and pilots.

19.1 Jobs

The *arrival time* of a job is the moment in time at which it arrives at a processor, while the *release time* of a job is the moment in time at which it becomes available for execution. In many cases, these two times will coincide, but sometimes it can be useful to postpone the release time of a job, notably to avoid resource competition (see section 19.3). The *execution time* of a job at a processor is the amount of time needed to perform the job (assuming it executes alone and all resources are available).

We disregard the functional behavior of a job and focus on its deadline, meaning the time by which it must have been completed. This can be expressed as an *absolute deadline*, meaning a fixed moment in real time, or as a *relative deadline*, the maximum allowed time between arrival and completion of a job. A deadline can be *hard*, meaning that late completion is not allowed, or *soft*, meaning that late completion is allowed but comes with some penalty.

A scheduler at a processor decides the order in which jobs are performed at this processor, and which resources they can claim. A scheduler aims to meet all hard deadlines, meet soft deadlines as much as possible, and avoid deadlocks. Of course, a job cannot be scheduled before its release time, and the total amount of time assigned to a job should equal its (maximum) execution time.

A task is a set of related jobs. Three types of tasks can be distinguished:

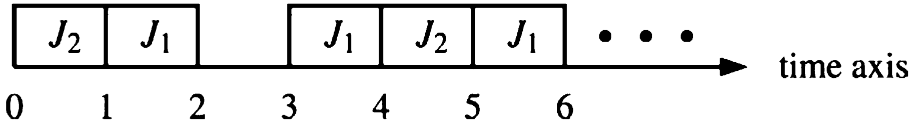
- *Periodic*: Such a task is known at the start of the system; the jobs have hard deadlines.
- *Aperiodic*: Such a task is executed in response to some external event; the jobs have soft deadlines.
- *Sporadic*: Such a task is executed in response to some external event; the jobs have hard deadlines.

A periodic task is defined by the following three parameters:

- the *release time r* of the first periodic job;
- the *period p*, which is a periodic time interval at the start of which a periodic job is released;
- the *execution time e* of each periodic job.

For simplicity, we assume that the relative deadline of each periodic job equals its period. The *utilization* of a periodic task (r, p, e) is $\frac{e}{p}$, representing the relative amount of execution time on a processor that will be consumed by this periodic task. The utilization at a processor is the sum of utilizations of its periodic tasks. Clearly, scheduling of the periodic tasks at a processor is possible only if its utilization does not exceed 1.

Example 19.1 Consider the periodic tasks $T_1 = (1, 2, 1)$ and $T_2 = (0, 3, 1)$ at a processor. The utilization at the processor is $\frac{1}{2} + \frac{1}{3} = \frac{5}{6}$. The periodic jobs can be executed as follows.



The conflict at time 3, when periodic jobs of both T_1 and T_2 are released, must be resolved by some scheduler. In this example, T_1 is given priority over T_2. Different schedulers will be discussed in the next section.

19.2 Schedulers

An offline scheduler determines the order in which jobs will be executed beforehand, typically with an algorithm for an NP-complete graph problem. In such schedulers, time is usually divided into regular time intervals called frames, and in each frame, a predetermined set of periodic tasks is executed. Jobs may be sliced into subjobs to accommodate the frame length. Offline scheduling is conceptually simple but cannot cope well with imprecise release and execution times, extra workload, nondeterminism, and system modifications.

Here we focus on online schedulers, where the schedule is computed at run-time. Scheduling decisions are made when jobs are released, when aperiodic or sporadic tasks arrive, when jobs are completed, or when resources are required or released. Released jobs are placed in priority queues, ordered by, for instance, release time, execution time, period of the task, deadline, or slack. The latter means the available idle time of a job until the next deadline. For example, if at time 2 a job with a deadline at time 6 still needs 3 time units to complete, then its slack at time 2 is $(6 - 2) - 3 = 1$.

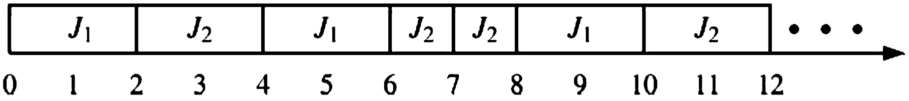
For simplicity, we consider aperiodic and sporadic jobs instead of tasks. Such jobs are offered to processors at run-time. Sporadic jobs are only accepted at a processor if they can be completed in time, without causing the processor to miss hard deadlines of other jobs. We assume that aperiodic jobs are always accepted and are performed such that periodic and accepted sporadic jobs do not miss their deadlines. Sporadic and aperiodic jobs that need to be executed at a processor are placed in job queues. The queueing discipline of aperiodic jobs tries to minimize the penalty associated with missed soft deadlines (for example, minimize the number of missed soft deadlines, or the average amount of time by which an aperiodic job misses its deadline).

Unless stated otherwise, we assume that there is no resource competition and that a job is preemptive, meaning that it can be suspended at any time during its execution.

Scheduling Periodic Jobs

A popular scheduler for periodic tasks is the *rate-monotonic* scheduler, which gives periodic jobs with a shorter period a higher priority. A strong point of this scheduler is that the static priority at the level of tasks makes its schedules relatively easy to compute and predict. The idea behind the rate-monotonic scheduler is that if a periodic job J_1 has a shorter period than a periodic job J_2, then the relative deadline of J_1 is shorter than the relative deadline of J_2. However, it may be the case that J_2 has been released before J_1, in which case J_s has an earlier deadline than J_1. As a result, the rate-monotonic scheduler is not optimal, in the sense that it may cause periodic jobs to miss their deadlines even in cases where the utilization of the periodic tasks at a processor is less than 1.

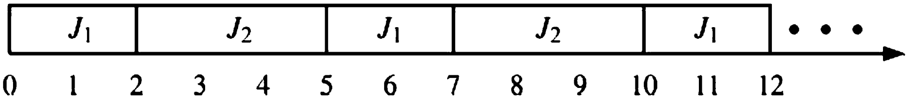
Example 19.2 Consider a single processor, with periodic tasks $T_1 = (0, 4, 2)$ and $T_2 = (0, 6, 3)$. Note that the utilization is $\frac{2}{4} + \frac{3}{6} = 1$. The rate-monotonic scheduler, which gives jobs from T_1 a higher priority than jobs from T_2, schedules the periodic jobs as follows.



Note that T_2 is preempted by T_1 at times 4 and 8. The first periodic job of T_2 misses its deadline at time 6.

The *earliest deadline first* scheduler gives a job a higher priority if its deadline is earlier. In the case of preemptive jobs and no competition for resources, this scheduler is optimal in the sense that if utilization at a processor does not exceed 1, then periodic jobs will be scheduled in such a way that no deadlines are missed.

Example 19.3 Consider the setting of example 19.2, a single processor with periodic tasks $T_1 = (0, 4, 2)$ and $T_2 = (0, 6, 3)$. The earliest deadline first scheduler may schedule the periodic jobs as follows.



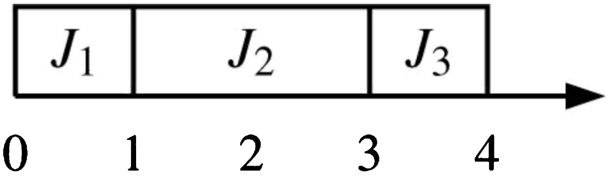
No deadlines are missed. Note that at time 8 it would also be possible to let J_1 preempt J_2, because both jobs have their deadlines at time 12.

The *least slack-time first* scheduler gives a job a higher priority if it has less slack. This scheduler is also optimal: if utilization at a processor does not exceed 1, then periodic jobs will be scheduled in such a way that no deadlines are missed.

The slack of a job gives precise information on how much time can be spent on other jobs without this job missing its deadline. However, continuously keeping track of and comparing the slack of all jobs imposes computational overhead. Another drawback of the least slack-time first scheduler is that priority between jobs is dynamic in the sense that it may change over time. Continuous scheduling decisions would lead to what is called context switch overhead in the case of two jobs with the same amount of slack, because they would interrupt each other repeatedly.

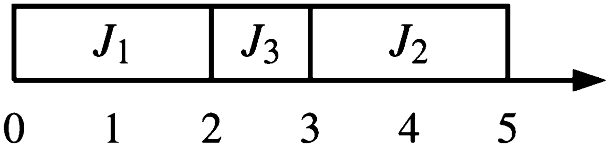
In the case of nonpreemptive jobs or resource competition, it may be impossible to schedule periodic jobs in such a way that no deadlines are missed, even in cases where utilization at a processor is (much smaller than) 1. For instance, suppose there are two nonpreemptive periodic tasks $(0, 1, e)$ and $(0, p, 2)$. No matter how small $e > 0$ and how large p are chosen, as soon as a job of the second periodic task is executed, a job of the first periodic task will miss its deadline. The same holds if these periodic tasks are preemptive but both require the same resource from start to finish. Moreover, nonpreemptive jobs or resource competition can give rise to scheduling anomalies: shorter execution times may lead to violation of deadlines. This is shown in the next example.

Example 19.4 Let three nonpreemptive jobs be executed at the same processor: job J_1 is released at time 0 with a deadline at time 2 and execution time 1; job J_2 is released at time 1 with a deadline at time 5 and execution time 2; and job J_3 is released at time 2 with a deadline at time 3 and execution time 1. The earliest deadline first and least slack-time first schedulers both schedule these three jobs as follows.



Job J_3 misses its deadline at time 3.

If the execution time of J_1 is increased from 1 to 2, then the earliest deadline first and least slack-time first schedulers both schedule these three jobs as follows.



In this case, no deadlines are missed.

Scheduling Aperiodic Jobs

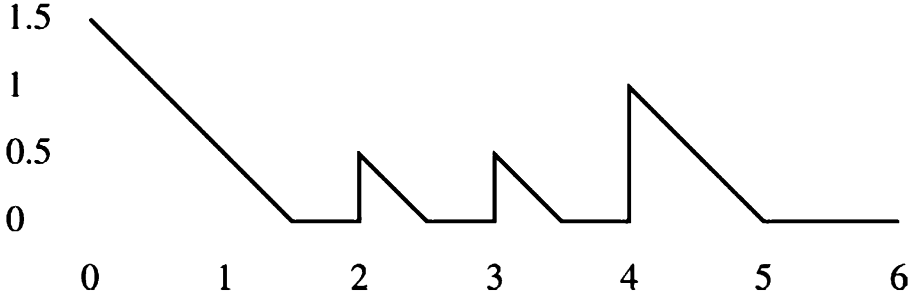
We assume that aperiodic jobs are always accepted for execution at a processor and that they are executed in such a way that periodic and accepted sporadic jobs do not miss their hard deadlines. The challenge is to execute aperiodic jobs in such a way that they adhere to their soft deadlines as much as possible.

A straightforward solution is the *background* server, which schedules aperiodic jobs only in idle time, when no periodic and sporadic jobs are available for execution. The drawback is that this server may needlessly let an aperiodic job miss its deadline.

Example 19.5 Let aperiodic job A have execution time 1 and a deadline 1 time unit away, while sporadic job S has execution time 1 and a deadline 10 time units away. Although A could easily be scheduled before S, the background server would schedule S first, causing A to miss its deadline.

In the *slack stealing* server, an aperiodic job may be executed as long as the processor has slack, meaning that it could idle without causing periodic or sporadic jobs to miss their deadlines. The drawback of this server is that the amount of slack of a processor is difficult to compute because it changes over time, and in practice one would need to take imprecise release and execution times into account.

Example 19.6 Suppose that a processor is executing periodic tasks $T_1 = (0, 2, \frac{1}{2})$ and $T_2 = (0, 3, \frac{1}{2})$, and that aperiodic jobs are available for execution at this processor in the time interval [0, 6]. The following graph depicts how, with the slack stealing server, the amount of slack of the processor changes over time.



In the time intervals $\langle 0, 1\frac{1}{2} \rangle$, $\langle 2, 2\frac{1}{2} \rangle$, $\langle 3, 3\frac{1}{2} \rangle$, and $\langle 4, 5 \rangle$, aperiodic jobs are executed. A periodic job from T_1 is executed in $\langle 1\frac{1}{2}, 2 \rangle$, $\langle 3\frac{1}{2}, 4 \rangle$, and $\langle 5, 5\frac{1}{2} \rangle$ and a periodic job from T_2 is executed in $\langle 2\frac{1}{2}, 3 \rangle$ and $\langle 5\frac{1}{2}, 6 \rangle$.

We discuss three more servers that are based on utilization. Suppose that periodic tasks $T_k = (r_k, p_k, e_k)$ for $k = 1, \ldots, n$ are being executed at the processor under consideration. For simplicity, we ignore sporadic jobs.

The *polling* server carries two parameters, p_s and e_s. In each period of length p_s, *the first e_s time units* can be used to execute aperiodic jobs. The polling server works correctly if

$$\sum_{k-1}^{n} \frac{e_k}{p_k} + \frac{e_s}{p_s} \leq 1. \tag{19.1}$$

For the correctness of the polling server, it is essential that in a period p_s, aperiodic jobs are executed only in the first e_s time units; see example 19.7. A drawback of the polling server is that aperiodic jobs released just after the first e_s time units of a period p_s may be delayed needlessly.

The *deferrable* server is similar to the polling server but allows aperiodic jobs to be executed for e_s time units in the entire period p_s, not only at the start. The following example shows that, for the deferrable server, criterion (19.1) for the values p_s and e_s would be incorrect.

Example 19.7 Consider a processor with one periodic task $T = (2, 5, 3\frac{1}{3})$, and let p_s = 3 and e_s = 1. Note that criterion (19.1) is satisfied.

Let an aperiodic job A with execution time 2 arrive at time 2. The deferrable server would allow A to execute at the end of the first period p_s, from time 2 until time 3, and at the start of the second period p_s, from time 3 until time 4. As a result, the first periodic job, which is released at time 2, can start execution only at time 4 and until time $7\frac{1}{3}$. So it misses its deadline at time 7.

A drawback of the deferrable server is that it is not easy to determine optimal values for p_s and e_s.

The *total bandwidth* server fixes a utilization rate $\tilde{u}_s$ for aperiodic jobs such that

$$\sum_{k=1}^{n} \frac{e_k}{p_k} + \tilde{u}_s \leq 1.$$

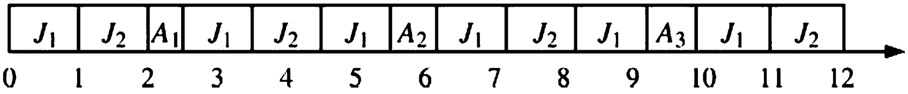
When the queue of aperiodic jobs gets a new head, a deadline d is determined for this head as follows. I, at a time t either a job arrives at the empty aperiodic queue or an aperiodic job completes and the tail of the aperiodic queue is nonempty, then

$$d \leftarrow \max\{d, t\} + \frac{e}{\tilde{u}_s},$$

where e denotes the execution time of the new head of the aperiodic queue. Initially, $d = 0$.

Aperiodic jobs can now be treated in the same way as periodic jobs, by the earliest deadline first scheduler. Periodic jobs are guaranteed to meet their deadlines (in the absence of sporadic jobs), and aperiodic jobs meet the deadlines assigned to them (which may differ from their actual soft deadlines).

Example 19.8 Consider a processor with two periodic tasks $T_1 = (0, 2, 1)$ and $T_2 = (0, 3, 1)$. We fix $\tilde{u}_s = \frac{1}{6}$.



- Aperiodic job A_1, released at time 1 with execution time $\frac{1}{2}$, gets (at 1) deadline $1 + 3 = 4$.
- Aperiodic job A_2, released at time 2 with execution time $\frac{2}{3}$, gets (at $2\frac{1}{2}$) deadline $4 + 4 = 8$.
- Aperiodic job A_3, released at time 3 with execution time $\frac{2}{3}$, gets (at $6\frac{1}{6}$) deadline $8 + 4 = 12$.

Scheduling Sporadic Jobs

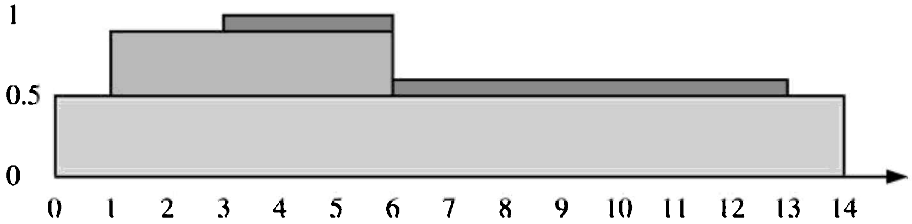
We now present an acceptance test for sporadic jobs. A sporadic job can be accepted at a processor only if it can be executed before its (hard) deadline without causing the violation of deadlines of periodic and accepted sporadic jobs at this processor.

A sporadic job with deadline d and execution time e that is offered to a processor at time t is accepted if utilization of the periodic and accepted sporadic jobs in the time interval $[t, d]$ is never more than $1 - \frac{e}{d-t}$. If accepted, utilization in $[t, d]$ is increased with $\frac{e}{d-t}$. Periodic and accepted sporadic jobs can be scheduled according to the earliest deadline first scheduler.

Example 19.9 Consider a processor with one periodic task $T = (0, 2, 1)$. Utilization of this periodic task over the entire time domain is $\frac{1}{2}$.

- Sporadic job S_1 with execution time 2 and deadline 6 is offered to the processor at time 1. S_1 is accepted, and utilization in $[1, 6]$ is increased to $\frac{1}{2} + \frac{2}{5} = \frac{9}{10}$.
- Sporadic job S_2 with execution time 2 and deadline 20 is offered to the processor at time 2. S_2 is rejected because utilization in the time interval $[2, 6]$ would increase beyond 1.

- Sporadic job S_3 with execution time 1 and deadline 13 is offered to the processor at time 3. S_3 is accepted, utilization in [3, 6] is increased to $\frac{9}{10} + \frac{1}{10} = 1$, and utilization in [6, 13] is increased to $\frac{1}{2} + \frac{1}{10} = \frac{3}{5}$.



The acceptance test may reject schedulable sporadic jobs. In particular, sporadic job S_2 in the previous example is schedulable, but it is nevertheless rejected.

The total bandwidth server can be integrated with the acceptance test for sporadic jobs, for example, by making the allowed utilization rate $\tilde{u}_s$ for the total bandwidth server dynamic.

19.3 Resource Access Control

So far, we have ignored competition for resources, such as a block of memory. In this section, we consider resource units that can be requested by a job during its execution and are allocated to jobs in a mutually exclusive fashion. When a requested resource is refused, the job is preempted.

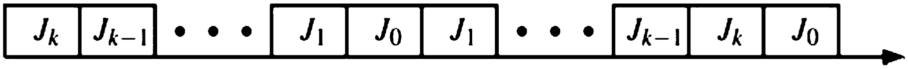
One danger of resource sharing is that it may give rise to deadlock, when two jobs block each other because they hold different resources and these jobs require both resources. A second danger is that a high-priority job J may be blocked by a sequence of low-priority jobs if J requires a resource that is being held by a job with a very low priority. We give examples of these two situations.

Example 19.10 Consider jobs J_1 and J_2, where J_1 has a higher priority than J_2, and resources R and R' .

First, J_2 is executing, and claims R. Then, J_1 arrives at the same processor, preempts J_2, starts executing, and claims R' . Next, J_1 requires R; since this resource is held by J_2, J_1 is preempted and J_2 continues its execution. Finally, J_2 requires R' ; since this resource is held by J_1, J_2 is preempted. Now J_1 is blocked because J_2 holds R, while J_2 is blocked because J_1 holds R' . So J_1 and J_2 are deadlocked.

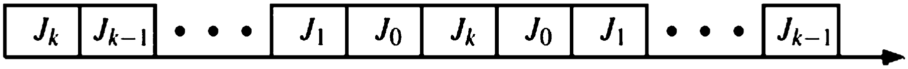
Example 19.11 Consider jobs $J_0, J_1, ..., J_k$, with $J_0 > J_1 > \cdots > J_k$ as priorities. A resource R is required by both J_0 and J_k.

First, J_k is executing and claims R. Then, J_{k-1} arrives at the same processor, preempts J_k, and starts executing. Next, J_{k-2} arrives at the same processor, preempts J_{k-1}, and starts executing. This pattern is repeated, until finally J_0 arrives at the same processor, preempts J_1, starts executing, and requires R. Since this resource is held by J_k, J_0 is preempted, and J_1 (which has the highest priority of the available jobs) continues its execution. When J_1 is completed, J_2 continues its execution; upon completion of J_2, J_3 continues its execution, and so on, until finally J_k continues its execution, completes, and releases R. Only then can J_0 claim R and continue its execution.



Priority inheritance makes blocking of a high-priority job J by a sequence of low-priority jobs less likely. The idea is that when a job J_1 is blocked because it requires a resource that is held by a job J_2, and J_1 has a higher priority than J_2, then J_2 inherits the priority of J_1 as long as it is blocking the execution of J_1.

Example 19.12 We revisit example 19.11 with priority inheritance. When J_0 requires R, J_k inherits the priority of J_k. So instead of J_1, now J_k continues its execution. When J_k completes and releases R, J_0 can claim R and continue its execution.



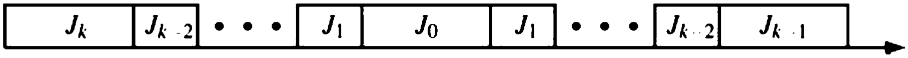
However, with priority inheritance, a deadlock can still occur. For instance, priority inheritance has no effect on the execution discussed in example 19.10. Such deadlocks can be avoided by a *priority ceiling*. The priority ceiling of a resource R at a time t is the highest priority of (known) jobs that will require R at some time $\geq t$. The priority ceiling of a processor at a time t is the highest priority ceiling of resources that are in use at time t. The priority ceiling of a processor has a special bottom value Ω when no resources are in use. In the case of a priority ceiling, from the arrival of a job at a processor, this job is not released until its priority is higher than the priority ceiling of the processor.

The idea behind a priority ceiling is that a job J is released only if all the resources it will require during its execution are not in use, because otherwise, upon the arrival of J, the priority ceiling of the processor increases to the priority of J if this priority ceiling is not beyond this level already. Of course, this approach works properly only if all the resources a job will require during its execution are always known beforehand. If priorities of jobs do not change over time (as is the case for the earliest deadline first scheduler but not for the least slack-time first scheduler), then deadlocks like the one in example 19.10 cannot occur.

Example 19.13 We revisit example 19.10 with a priority ceiling. When J_1 arrives at the processor, the priority ceiling is increased to the priority of J_1, because J_1 will require R, and this resource is in use by J_2. So J_1 is not yet released. When J_2 completes and releases R, the priority ceiling goes down to Ω, so that J_1 is released and starts executing.

A priority ceiling makes it less likely that a high-priority job is blocked by a series of lower-priority jobs.

Example 19.14 We revisit example 19.11 with a priority ceiling, under the assumption that the future arrival of J_0 is known from the start. Since J_0 will require R, and J_k holds R, the priority ceiling becomes the priority of J_0. Therefore, J_{k-1} is not released upon its arrival, and J_k continues its execution. When J_k completes and releases R, the priority ceiling drops to Ω, so jobs are released. When J_0 arrives, it can claim R and start executing straightaway.



The priority ceiling has no effect on jobs that have been released. For instance, if in example 19.11 the arrival of J_0 is known only upon its arrival, a priority ceiling does not help. Therefore, priority inheritance tends to be imposed on top of a priority ceiling.

The priority ceiling can be extended to a setting with multiple units of the same resource type. Then the definition of a priority ceiling is adapted as follows. The priority ceiling of a resource R with k free units at a time t is the highest priority level of known jobs that require more than k units of R at some time $\geq t$.

Bibliographical notes

The slack stealing server originates from [58], the deferrable server from [89], and the total bandwidth server in [88]. Priority inheritance was introduced in [84], and the priority ceiling comes from [77].

19.4 Exercises

In these exercises, all jobs are assumed to be preemptive.

Exercise 19.1 [60] Consider a system with two processors. Suppose jobs J_1, J_2, and J_3 are released at time 0, with execution times 1, 1, and 2 and deadlines at times 1, 2, and 3, respectively.

(a) Let jobs J_4 and J_5 be released at time 2, both with execution time 1 and deadline 3.
(b) Let job J_4 be released at time 1, with execution time 2 and deadline 3. (There is no J_5.)

In both cases, give a schedule such that all deadlines are met.

Exercise 19.2 [60] Which of the following collections of periodic tasks are schedulable on one processor by the earliest deadline first scheduler? And which ones are schedulable by the rate-monotonic scheduler?

(a) (0, 8, 4), (0, 10, 2), (0, 12, 3).
(b) (0, 8, 4), (0, 12, 4), (0, 20, 4).
(c) (0, 8, 3), (0, 9, 3), (0, 15, 3).

Exercise 19.3 Consider a processor with one periodic task $(0, 5, 3\frac{1}{3})$ and the earliest deadline first scheduler.

(a) Given a polling server with $p_s = 3$, what is the maximum value for e_s?
(b) Given a deferrable server with $p_s = 3$, what is the maximum value for e_s?
(c) Given a total bandwidth server, what is the maximum utilization rate $\tilde{u}_s$?
(d) Suppose aperiodic jobs A_1, A_2, and A_3 arrive at times 3, 5, and 13, with execution times 1, 2, and 1, respectively. Explain how these aperiodic jobs are executed in the case of the deferrable server (with e_s maximal) and the total bandwidth server (with $\tilde{u}_s$ maximal).

Exercise 19.4 Suppose that the total bandwidth server is adapted as follows. When at time t an aperiodic job (with execution time e) arrives at the aperiodic queue while it is empty, $d \leftarrow t + \frac{e}{\tilde{u}_s}$. Give an example to show that then, with the earliest deadline first scheduler, periodic jobs may miss their deadlines.

Exercise 19.5 Consider a processor with one periodic task (0, 3, 1). Suppose sporadic jobs S_1, S_2, S_3, and S_4 arrive at times 0, 1, 3, and 6, with execution times 1, 3, 1, and 2, and with deadlines at times 1, 12, 7, and 14, respectively. Explain which of these jobs pass the acceptance test.

Exercise 19.6 Give an example where the acceptance test for sporadic jobs rejects a sporadic job at a time t, while it accepts this same job at a time $t' > t$.

Exercise 19.7 Suggest an adaptation of the acceptance test for sporadic jobs that accepts more sporadic jobs (without computing slack). Give an example of a sporadic job that is accepted by your test but not by the original test. Does your test accept all schedulable sporadic jobs?

Exercise 19.8 Jobs J_1 and J_2 arrive at times 1 and 0, with execution times 1 and 2, respectively. Let J_1 and J_2 use resource R for their entire executions, and J_2 use resource R' for the last time unit of its execution.

(a) Job J_3 arrives at time 1, with execution time 100. Let $J_1 > J_3 > J_2$. Explain how J_1, J_2, and J_3 are executed with and without priority inheritance.

(b) Job J_4 arrives at time 1, with execution time 2. Let J_4 use resource R' for its entire execution and use resource R for the last time unit of its execution. Let $J_1 > J_4 > J_2$. Explain how J_1, J_2, and J_4 are executed with and without priority inheritance.

Exercise 19.9 Let preemptive jobs J_1, J_2, and J_3 arrive at times 2, 1, and 0, respectively, with execution time 2. Let the priorities be $J_1 > J_2 > J_3$. Let J_1 and J_3 use resource R for their entire executions. The jobs are executed using a priority ceiling.

(a) Explain how the three jobs are executed if the arrival of J_1 is known from the start.

(b) Explain how the three jobs are executed if the arrival of J_1 is not known before time 2. Consider the cases with and without priority inheritance.

Exercise 19.10 Give an example to show that with a priority ceiling a job can still be blocked by a sequence of lower-priority jobs, even if there is priority inheritance and the arrivals of all jobs are known from the start.

Exercise 19.11 Give an example to show that a deadlock can occur if a priority ceiling is applied in combination with the least slack-time first scheduler. (The resources a job requires during its execution are assumed to be known beforehand.)

A

Appendix: Pseudocode Descriptions

Pseudocode descriptions are presented for a considerable number of distributed algorithms discussed in the main body of this book. Several algorithms are excluded here either because their pseudocode descriptions are trivial or very similar to another algorithm that is included or because the main body contains a description that resembles the pseudocode.

Each piece of pseudocode is presented for a process p or p_i; its local variables are subscripted with p or i, respectively. We use $Neighbors_p$ to denote the set of neighbors of process p in the network and use *Processes* for the set of processes in the network.

Each pseudocode description starts with a variable declaration section. Let **bool**, **nat**, **int**, and **real** denote the data type of Booleans, natural numbers, integers, and reals, with default initial values *false* for the first data type and 0 for the latter three. The operations $\wedge$, $\vee$, and $\neg$ on Booleans denote conjunction, disjunction, and negation, respectively. The data type **dist**, representing distance, consists of the natural numbers extended with infinity ∞, where $\infty + d = d + \infty = \infty$ for all distance values d, and $d < \infty$ for all $d \neq \infty$; its default initial value is ∞.

The data type of processes in the network, **proc**, has default initial value $\perp$ (i.e., undefined). The data types **mess-queue** and **proc-queue** represent FIFO queues of basic messages and processes, respectively. Likewise, **mess-set**, **proc-set**, **proc-nat-set**, **proc-real-nat-set**, and **proc-dist-set** represent sets of basic messages; processes; pairs of a process and a natural number; triples of a process, a natural number, and a real value; and pairs of a process and a distance value, respectively. Variables containing queues or sets have as default initial value ∅; that is, empty. There are three operations on queues: *head* produces the head and *tail* the tail of the queue (on the empty queue, these operations are undefined), while *append*(Q, e) appends element e at the end of queue Q.

We recall that assignment of a new value to a variable is written as ←. Equality between two data elements, $d_1 = d_2$ (or between two sets, $S_1 = S_2$), represents a Boolean value, which is *true* if and only if the two elements (or sets) are equal. We also recall that the network topology is supposed to be strongly connected. In the pseudocode, it is assumed that the network size N is greater than one.

A process is supposed to interrupt the execution of a procedure call (under a boxed text, such as "If p receives [...]") only if it needs to wait for an incoming message, or in the case of a **while** b **do** *statement* **end while** construct, or when it enters its critical section.

In general, pseudocode tends to be error-prone because on the one hand it is condensed and intricate, while on the other hand it has never been executed. I welcome any comments on the pseudocode descriptions, as well as on the main body of the book.

A.1 Chandy-Lamport Snapshot Algorithm

The Boolean variable $recorded_p$ in the following pseudocode is set (to *true*) when p takes a local snapshot of its state. For each incoming channel c of p, the Boolean variable $marker_p[c]$ is set when a **marker** message arrives at p through c, and the queue $state_p[c]$ keeps track of the basic messages that arrive through channel c after p has taken its local snapshot and before a **marker** message arrives through c.

```
bool $recorded_p$, $marker_p[c]$ for all incoming channels $c$ of $p$;
mess-queue $state_p[c]$ for all incoming channels $c$ of $p$;

If $p$ wants to initiate a snapshot
perform procedure $TakeSnapshot_p$;

If $p$ receives a basic message $m$ through an incoming channel $c_0$
if $recorded_p = true$ and $marker_p[c_0] = false$ then
    $state_p[c_0] \leftarrow append(state_p[c_0], m)$;
end if

If $p$ receives $\langle$marker$\rangle$ through an incoming channel $c_0$
perform procedure $TakeSnapshot_p$;
$marker_p[c_0] \leftarrow true$;
```

```
if $marker_p[c] = true$ for all incoming channels $c$ of $p$ then
    terminate;
end if
```

```
Procedure $TakeSnapshot_p$

if $recorded_p = false$ then
    $recorded_p \leftarrow true$;
    send ⟨marker⟩ into each outgoing channel of $p$;
    take a local snapshot of the state of $p$;
end if
```

A.2 Lai-Yang Snapshot Algorithm

The variable $recorded_p$ is set when p takes a local snapshot of its state. The set $State_p[qp]$ keeps track of the basic messages that arrive at p through its incoming channel qp after p has taken its local snapshot and that were sent by q before it took its local snapshot. The variable $counter_q[qp]$ counts how many basic messages process q sent into its outgoing channel qp before taking its local snapshot. Right before taking its local snapshot, q sends the control message ⟨**presnap**, $counter_q[qp] + 1$⟩ to p (the + 1 is present because the control message itself is also counted), and p stores the value within this message in the variable $counter_p[qp]$. Finally, p terminates when it has received a control message ⟨**presnap**, k⟩ and $k - 1$ basic messages with the tag *false* through each incoming channel qp.

```
bool $recorded_p$;
nat $counter_p[c]$ for all channels $c$ of $p$;
mess-set $State_p[c]$ for all incoming channels $c$ of $p$;

If $p$ wants to initiate a snapshot

perform procedure $TakeSnapshot_p$;

If $p$ sends a basic message $m$ into an outgoing channel $c_0$

send ⟨$m$, $recorded_p$⟩ into $c_0$;
if $recorded_p = false$ then
    $counter_p[c_0] \leftarrow counter_p[c_0] + 1$;
end if

If $p$ receives ⟨$m$, $b$⟩ through an incoming channel $c_0$

if $b = true$ then
    perform procedure $TakeSnapshot_p$;
else
```

```
        $counter_p[c_0] \leftarrow counter_p[c_0] - 1$;
        if $recorded_p = true$ then
            $State_p[c_0] \leftarrow State_p[c_0] \cup \{m\}$;
            if $|State_p[c]| + 1 = counter_p[c]$ for all incoming channels $c$ of $p$ then
                terminate;
            end if
        end if
    end if

    If $p$ receives ⟨presnap, $\ell$⟩ through an incoming channel $c_0$

    $counter_p[c_0] \leftarrow counter_p[c_0] + \ell$;
    perform procedure $TakeSnapshot_p$;
    if $|State_p[c]| + 1 = counter_p[c]$ for all incoming channels $c$ of $p$ then
        terminate;
    end if

    Procedure $TakeSnapshot_p$

    if $recorded_p = false$ then
        $recorded_p \leftarrow true$;
        send ⟨presnap, $counter_p[c] + 1$⟩ into each outgoing channel $c$;
        take a local snapshot of the state of $p$;
    end if
```

A.3 Cidon's Depth-First Search Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator (or $\perp$ if p has no parent); $info_p$ is set when p sends the token for the first time, and $token_p[q]$ is set when p is certain that neighbor q will receive or has received the token. In the variable $forward_p$, the neighbor is stored to which p forwarded the token last.

```
    bool $info_p$, $token_p[r]$ for all $r \in Neighbors_p$;
    proc $parent_p$, $forward_p$;

    If $p$ is the initiator

    perform procedure $ForwardToken_p$;

    If $p$ receives ⟨info⟩ from a neighbor $q$

    if $forward_p \neq q$ then
        $token_p[q] \leftarrow true$;
    else
    perform procedure $ForwardToken_p$;
end if

If $p$ receives ⟨token⟩ from a neighbor $q$

if $forward_p = \perp$ then
    $parent_p \leftarrow q$;   $token_p[q] \leftarrow true$;
    perform procedure $ForwardToken_p$;
else if $forward_p = q$ then
    perform procedure $ForwardToken_p$;
else
    $token_p[q] \leftarrow true$;
end if

Procedure $ForwardToken_p$

if $\{r \in Neighbors_p \mid token_p[r] = false\} \neq \emptyset$ then
    choose a $q$ from this set, and send ⟨token⟩ to $q$;
    $forward_p \leftarrow q$;   $token_p[q] \leftarrow true$;
    if $info_p = false$ then
        send ⟨info⟩ to each $r \in Neighbors_p \setminus \{q, parent_p\}$;
        $info_p \leftarrow true$;
    end if
else if $parent_p \neq \perp$ then
    send ⟨token⟩ to $parent_p$;
else
    $decide$;
end if
```

A.4 Tree Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree. The variable $received_p[q]$ is set when p receives a wave message from neighbor q. Messages are included to inform all processes of the decision.

```
bool $received_p[r]$ for all $r \in Neighbors_p$;
proc $parent_p$;

Initialization of $p$

perform procedure $SendWave_p$;

If $p$ receives ⟨wave⟩ from a neighbor $q$

$received_p[q] \leftarrow true$;
```

```
perform procedure $SendWave_p$;
Procedure $SendWave_p$
if $|\{r \in Neighbors_p \mid received_p[r] = false\}| = 1$ then
    send ⟨wave⟩ to the only $q \in Neighbors_p$ with $received_p[q] = false$;
    $parent_p \leftarrow q$;
else if $|\{r \in Neighbors_p \mid received_p[r] = false\}| = 0$ then
    decide;
    send ⟨info⟩ to each $r \in Neighbors_p \setminus \{parent_p\}$;
end if
If $p$ receives ⟨info⟩ from $parent_p$
send ⟨info⟩ to each $r \in Neighbors_p \setminus \{parent_p\}$;
```

A.5 Echo Algorithm

The variable *parent*$_p$ contains the parent of p in the spanning tree rooted at the initiator. The variable *received*$_p$ counts how many wave messages have arrived at p.

```
nat $received_p$;
proc $parent_p$;
If $p$ is the initiator
send ⟨wave⟩ to each $r \in Neighbors_p$;
If $p$ receives ⟨wave⟩ from a neighbor $q$
$received_p \leftarrow received_p + 1$;
if $parent_p = \perp$ and $p$ is a noninitiator then
    $parent_p \leftarrow q$;
    if $|Neighbors_p| > 1$ then
        send ⟨wave⟩ to each $r \in Neighbors_p \setminus \{q\}$;
    else
        send ⟨wave⟩ to $q$;
    end if
else if $received_p = |Neighbors_p|$ then
    if $parent_p \neq \perp$ then
        send ⟨wave⟩ to $parent_p$;
    else
        decide;
    end if
end if
```

A.6 Shavit-Francez Termination Detection Algorithm

The variable $parent_p$ contains the parent of p in a tree in the forest, and cc_p keeps track of (or better, estimates from above) the number of children of p in its tree. The variable $active_p$ is set when p becomes active, and it is reset when p becomes passive.

```
bool $active_p$;
nat $cc_p$;
proc $parent_p$;

If $p$ is an initiator
$active_p \leftarrow true$;

If $p$ sends a basic message
$cc_p \leftarrow cc_p + 1$;

If $p$ receives a basic message from a neighbor $q$
if $active_p = false$ then
    $active_p \leftarrow true$; $parent_p \leftarrow q$;
else
    send ⟨ack⟩ to $q$;
end if

If $p$ receives ⟨ack⟩
$cc_p \leftarrow cc_p - 1$;
perform procedure $LeaveTree_p$;

If $p$ becomes passive
$active_p \leftarrow false$;
perform procedure $LeaveTree_p$;

Procedure $LeaveTree_p$
if $active_p = false$ and $cc_p = 0$ then
    if $parent_p \neq \bot$ then
        send ⟨ack⟩ to $parent_p$;
        $parent_p \leftarrow \bot$;
    else
        start a wave, tagged with $p$;
    end if
end if

If $p$ receives a wave message
if $active_p = false$ and $cc_p = 0$ then
```

```
    act according to the wave algorithm;
    in the case of a decide event, call Announce;
end if
```

A.7 Rana’s Termination Detection Algorithm

The variable $active_p$ is set when p becomes active, and it is reset when p becomes passive. The variable $clock_p$ represents the clock value at p, and $unack_p$ keeps track of the number of unacknowledged basic messages sent by p.

```
bool $active_p$;
nat $clock_p$, $unack_p$;

$\boxed{\text{If } p \text{ is an initiator}}$
$active_p \leftarrow true$;

$\boxed{\text{If } p \text{ sends a basic message}}$
$unack_p \leftarrow unack_p + 1$;

$\boxed{\text{If } p \text{ receives a basic message from a neighbor } q}$
$active_p \leftarrow true$;
send $\langle \mathbf{ack}, clock_p \rangle$ to $q$;

$\boxed{\text{If } p \text{ receives } \langle \mathbf{ack}, t \rangle}$
$clock_p \leftarrow \max\{clock_p, t + 1\}$;    $unack_p \leftarrow unack_p - 1$;
if $active_p = false$ and $unack_p = 0$ then
    start a wave, tagged with $p$ and $clock_p$;
end if

$\boxed{\text{If } p \text{ becomes passive}}$
$active_p \leftarrow false$;
if $unack_p = 0$ then
    start a wave, tagged with $p$ and $clock_p$;
end if

$\boxed{\text{If } p \text{ receives a wave message tagged with } q \text{ and } t}$
if $active_p = false$ and $unack_p = 0$ and $clock_p \leq t$ then
    act according to the wave algorithm, for the wave tagged with $q$ and $t$;
    in the case of a decide event, call Announce;
end if
$clock_p \leftarrow \max\{clock_p, t\}$;
```

A.8 Safra's Termination Detection Algorithm

The variable $active_p$ is set when p becomes active, and it is reset when p becomes passive. The variable $black_p$ is set when p receives a basic message, and it is reset when p forwards the token. Moreover, the initiator of the control algorithm at the start sets this variable to make sure it sends out the token when it becomes passive for the first time. As long as p is holding the token, $token_p$ is set. When p sends or receives a basic message, $mess\text{-}counter_p$ is increased or decreased by 1, respectively. The variable $token\text{-}counter_p$ is used to store the counter value of the token. For simplicity, we assume that the initiator of the control algorithm is also an initiator of the basic algorithm.

```
bool $active_p$, $token_p$, $black_p$;
int $mess\text{-}counter_p$, $token\text{-}counter_p$;

If $p$ is the initiator of the control algorithm
$token_p \leftarrow true$;   $black_p \leftarrow true$;

If $p$ is an initiator of the basic algorithm
$active_p \leftarrow true$;

If $p$ sends a basic message
$mess\text{-}counter_p \leftarrow mess\text{-}counter_p + 1$;

If $p$ receives a basic message
$active_p \leftarrow true$;   $black_p \leftarrow true$;   $mess\text{-}counter_p \leftarrow mess\text{-}counter_p - 1$;

If $p$ becomes passive
$active_p \leftarrow false$;
perform procedure $TreatToken_p$;

If $p$ receives $\langle \mathbf{token}, b, k \rangle$
$token_p \leftarrow true$;   $black_p \leftarrow black_p \vee b$;   $token\text{-}counter_p \leftarrow k$;
perform procedure $TreatToken_p$;

Procedure $TreatToken_p$
if $active_p = false$ and $token_p = true$ then
    if $black_p = false$ then
        $mess\text{-}counter_p \leftarrow mess\text{-}counter_p + token\text{-}counter_p$
    end if
    if $p$ is a noninitiator then
        forward $\langle \mathbf{token}, black_p, mess\text{-}counter_p \rangle$;
        $token_p \leftarrow false$;   $black_p \leftarrow false$;
    else if $black_p = true$ or $mess\text{-}counter_p > 0$ then
```

```
        send ⟨token, $false$, 0⟩ on a round trip through the network;
        $token_p \leftarrow false$;    $black_p \leftarrow false$;
    else
        call $Announce$;
    end if
end if
```

A.9 Weight-Throwing Termination Detection Algorithm

The variable $active_p$ is set when p becomes active, and it is reset when p becomes passive. The variable $weight_p$ contains the weight at p, and $total$ contains the total amount of weight in the network. The constant $minimum$, a real value between 0 and $\frac{1}{2}$, represents the minimum allowed weight at a process. In the case of underflow, a noninitiator informs the initiator that it has added one extra unit of weight to the system, and it waits for an acknowledgment from the initiator. For simplicity, we assume that there is an undirected channel between the initiator and every other process in the network.

```
bool $active_p$;
real $weight_p$, $total$ only at the initiator;

| If $p$ is the initiator |
$active_p \leftarrow true$;    $weight_p \leftarrow 1$;    $total \leftarrow 1$;

| If $p$ sends a basic message $m$ to a neighbor $q$ |
if $\frac{1}{2} \cdot weight_p < minimum$ then
    if $p$ is a noninitiator then
        send ⟨more-weight⟩ to the initiator to ask for extra weight;
        wait for an acknowledgment from the initiator to arrive;
    else
        $total \leftarrow total + 1$;
    end if
    $weight_p \leftarrow weight_p + 1$;
end if
send ⟨$m, \frac{1}{2} \cdot weight_p$⟩ to $q$;
$weight_p \leftarrow \frac{1}{2} \cdot weight_p$;

| If $p$ receives a basic message ⟨$m, w$⟩ |
$active_p \leftarrow true$;    $weight_p \leftarrow weight_p + w$;
```

```
If $p$ becomes passive

$active_p \leftarrow false$;
if $p$ is a noninitiator then
    send ⟨return-weight, $weight_p$⟩ to the initiator;
    $weight_p \leftarrow 0$;
else if $total = weight_p$ then
    call $Announce$;
end if

If initiator $p$ receives ⟨more-weight⟩ from a process $q$

$total \leftarrow total + 1$;
send an acknowledgment to $q$;

If initiator $p$ receives ⟨return-weight, $w$⟩

$weight_p \leftarrow weight_p + w$;
if $active_p = false$ and $total = weight_p$ then
    call $Announce$;
end if
```

A.10 Chandy-Misra Routing Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator, and $dist_p$ is the distance value of p toward the initiator.

```
dist $dist_p$;
proc $parent_p$;

If $p$ is the initiator

$dist_p \leftarrow 0$;
send ⟨dist, 0⟩ to each $r \in Neighbors_p$;

If $p$ receives ⟨dist, $d$⟩ from a neighbor $q$

if $d + weight(pq) < dist_p$ then
    $dist_p \leftarrow d + weight(pq)$;    $parent_p \leftarrow q$;
    send ⟨dist, $dist_p$⟩ to each $r \in Neighbors_p \setminus \{q\}$;
end if
```

A.11 Merlin-Segall Routing Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator, and $dist_p$ is the distance value of p toward the initiator. In $new\text{-}parent_p$, the process is stored that sent the message to p on which the current value of $dist_p$ is based; at the end of a round, the value of $new\text{-}parent_p$ is passed on to $parent_p$. In $counter_p$, p keeps track of how many messages it has received in the current round.

```
nat $counter_p$;
dist $dist_p$;
proc $parent_p$, $new\text{-}parent_p$;
If $p$ is the initiator
$dist_p \leftarrow 0$;
initiate a wave that determines a spanning tree of the network, captured by values
of $parent_r$ for all $r \in Processes$, with $p$ as root;
wait until this wave has terminated;
for $k = 1$ to $N - 1$ do
    send ⟨dist, 0⟩ to each $r \in Neighbors_p$;
    while $counter_p < |Neighbors_p|$ do
        wait for a message ⟨dist, $d$⟩ to arrive;
        $counter_p \leftarrow counter_p + 1$;
    end while
    $counter_p \leftarrow 0$;
end for
If $p$ is a noninitiator
take part in the wave, and provide $parent_p$ with the resulting parent value;
for $k = 1$ to $N - 1$ do
    while $counter_p < |Neighbors_p|$ do
        wait for a message ⟨dist, $d$⟩ from a $q \in Neighbors_p$;
        $counter_p \leftarrow counter_p + 1$;
        if $d + weight(pq) < dist_p$ then
            $dist_p \leftarrow d + weight(pq)$;    $new\text{-}parent_p \leftarrow q$;
        end if
        if $q = parent_p$ then
            send ⟨dist, $dist_p$⟩ to each $r \in Neighbors_p \setminus \{parent_p\}$;
        end if
    end while
    send ⟨dist, $dist_p$⟩ to $parent_p$;
    $parent_p \leftarrow new\text{-}parent_p$;    $counter_p \leftarrow 0$;
end for
```

A.12 Toueg’s Routing Algorithm

The variable $parent_p[q]$ contains the parent of p in the spanning tree rooted at process q, and $dist_p[q]$ is the distance value of p toward destination q. In $round_p$, p keeps track of its round number. The distance values of the pivot in round k are stored in $Distances_p[k]$. Each process that sends a request to p for the distance values of the pivot in the current or a future round k is stored in $Forward_p[k]$. We assume that p only treats incoming requests when it is idle, to avoid having a request stored in $Forward_p[k]$ after p forwarded the distance values of the pivot in round k. The pivot in round k is denoted by $pivot(k)$. We include the optimization that a process, upon receiving distance values from the pivot, first checks which of its distance values are improved and then forwards only those elements of the set that gave rise to an improved distance value.

```
nat $round_p$;
dist $dist_p[r]$ for all $r \in Processes$;
proc $parent_p[r]$ for all $r \in Processes$;
proc-set $Forward_p[k]$ for all $k \in \{0,\ldots,N-1\}$;
proc-dist-set $Distances_p[k]$ for all $k \in \{0,\ldots,N-1\}$;

Initialization of $p$

$dist_p[p] \leftarrow 0$; $parent_p[r] \leftarrow r$ and $dist_p[r] \leftarrow weight(pr)$ for all $r \in Neighbors_p$;
perform procedure $Request_p$;

Procedure $Request_p$

if $p = pivot(round_p)$ then
    send ⟨dist-set, $\{(r, dist_p[r]) \mid r \in Processes$ and $dist_p[r] < \infty\}$⟩
    to each $q \in Forward_p[round_p]$;
    perform procedure $NextRound_p$;
else if $parent_p[pivot(round_p)] \neq \perp$ then
    send ⟨request, $round_p$⟩ to $parent_p[pivot(round_p)]$;
else
    perform procedure $NextRound_p$;
end if

If $p$ receives ⟨request, $k$⟩ from a neighbor $q$

if $k < round_p$ then
    send ⟨dist-set, $Distances_p[k]$⟩ to $q$;
else
    $Forward_p[k] \leftarrow Forward_p[k] \cup \{q\}$;
end if

If $p$ receives ⟨dist-set, $Distances$⟩ from $parent_p[pivot(round_p)]$

for each $s \in Processes$ do
    if there is a pair $(s, d)$ in $Distances$ then
```

```
        if $d + dist_p[pivot(round_p)] < dist_p[s]$ then
            $parent_p[s] \leftarrow parent_p[pivot(round_p)]$;
            $dist_p[s] \leftarrow d + dist_p[pivot(round_p)]$;
        else
            remove entry $(s, d)$ from $Distances$;
        end if
    end if
end for
send ⟨dist-set, $Distances$⟩ to each $r \in Forward_p[round_p]$;
$Distances_p[round_p] \leftarrow Distances$;
perform procedure $NextRound_p$;
```

```
Procedure $NextRound_p$

if $round_p < N - 1$ then
    $round_p \leftarrow round_p + 1$;
    perform procedure $Request_p$;
else
    terminate;
end if
```

A.13 Frederickson's Breadth-First Search Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator, and $dist_p$ is the distance value of p toward the initiator. In $dist_p[r]$, p stores the best-known distance value of neighbor r. After p has sent **forward** or **explore** messages, it uses Ack_p to keep track of the neighbors that should still send a (positive or negative) reply. In $Reported_p$, p keeps track of the neighbors to which it must send a **forward** message in the next round. The initiator maintains the round number in *counter*. During each round, ℓ levels are explored. For uniformity, messages ⟨**reverse**, b⟩ are always supplied with the distance value of the sender.

```
nat counter only at the initiator;
dist $dist_p$, $dist_p[r]$ for all $r \in Neighbors_p$;
proc $parent_p$;
proc-set $Ack_p$, $Reported_p$;
```

```
If $p$ is the initiator

send ⟨explore, 1⟩ to each $r \in Neighbors_p$;
$dist_p \leftarrow 0$;    $Ack_p \leftarrow Neighbors_p$;    $counter \leftarrow 1$;
```

```
If $p$ receives ⟨explore, $k$⟩ from a neighbor $q$
```

```
$dist_p[q] \leftarrow \min\{dist_p[q], k - 1\}$;
if $k < dist_p$ then
    $parent_p \leftarrow q$; $dist_p \leftarrow k$; $Reported_p \leftarrow \emptyset$;
    if $k \bmod \ell \neq 0$ then
        send ⟨explore, $k + 1$⟩ to each $r \in Neighbors_p \setminus \{q\}$;
        $Ack_p \leftarrow \{r \in Neighbors_p \mid dist_p[r] > k + 1\}$;
        if $Ack_p = \emptyset$ then
            send ⟨reverse, $k$, $false$⟩ to $q$;
        end if
    else
        send ⟨reverse, $k$, $true$⟩ to $q$;
    end if
else if $k > dist_p$ or $k \bmod \ell \neq 0$ then
    if $k \leq dist_p + 2$ and $q \in Ack_p$ then
        $Ack_p \leftarrow Ack_p \setminus \{q\}$;
        perform procedure $ReceivedAck_p$;
    else if $k = dist_p$ then
        $Reported_p \leftarrow Reported_p \setminus \{q\}$;
    end if
else
    send ⟨reverse, $k$, $false$⟩ to $q$;
end if
```

If p receives ⟨**reverse**, k, b⟩ from a neighbor q

```
$dist_p[q] \leftarrow \min\{dist_p[q], k\}$;
if $k = dist_p + 1$ then
    if $b = true$ and $dist_p[q] = k$ then
        $Reported_p \leftarrow Reported_p \cup \{q\}$;
    end if
    if $q \in Ack_p$ then
        $Ack_p \leftarrow Ack_p \setminus \{q\}$;
        perform procedure $ReceivedAck_p$;
    end if
end if
```

Procedure $ReceivedAck_p$

```
if $Ack_p = \emptyset$ then
    if $parent_p \neq \perp$ then
        send ⟨reverse, $dist_p$, $Reported_p \neq \emptyset$⟩ to $parent_p$;
    else if $Reported_p \neq \emptyset$ then
        send ⟨forward, $\ell \cdot counter$⟩ to each $r \in Reported_p$;
        $Ack_p \leftarrow Reported_p$; $Reported_p \leftarrow \emptyset$; $counter \leftarrow counter + 1$;
    else
        terminate;
    end if
end if

If $p$ receives ⟨forward, $k$⟩ from a neighbor $q$

if $q = parent_p$ then
    if $k < depth_p$ then
        send ⟨forward, $k$⟩ to each $r \in Reported_p$;
        $Ack_p \leftarrow Reported_p$; $Reported_p \leftarrow \emptyset$;
    else
        $Ack_p \leftarrow \{r \in Neighbors_p \mid dist_p[r] = \infty\}$;
        if $Ack_p \neq \emptyset$ then
            send ⟨explore, $k + 1$⟩ to each $r \in Ack_p$;
        else
            send ⟨reverse, $k$, $false$⟩ to $q$;
        end if
    end if
end if
```

A.14 Dolev-Klawe-Rodeh Election Algorithm

Initially, $active_p$ is set if p is an initiator, and it is reset when p becomes passive. If p terminates as the leader, it sets $leader_p$. Since messages of two consecutive rounds can overtake each other, p keeps track of the parity of its round number in $parity_p$ and attaches this Boolean value to its message. In $election\text{-}id_p$, p stores the ID it assumes for the current election round. In $neighb\text{-}id_p[0, b]$ and $neighb\text{-}id_p[1, b]$, p stores the process IDs of its two nearest active predecessors in the directed ring, with b the parity of the corresponding election round. We assume a total order $<$ on process IDs.

```
bool $active_p$, $leader_p$, $parity_p$;
proc $election\text{-}id_p$, $neighb\text{-}id_p[n, b]$ for $n = 0, 1$ and Booleans $b$;

If $p$ is an initiator

$active_p \leftarrow true$;    $election\text{-}id_p \leftarrow p$;
send ⟨id, $p$, 0, $b$⟩;

If $p$ receives ⟨id, $q$, $n$, $b$⟩

if $active_p = true$ then
    if $n = 0$ then
```

```
        send ⟨id, $q$, 1, $b$⟩;
    end if
    $neighb\text{-}id_p[n, b] \leftarrow q$;
    if $neighb\text{-}id_p[n, parity_p] \neq \bot$ for $n = 0$ and $n = 1$ then
        perform procedure $CompareIds_p$;
    end if
else
    send ⟨id, $q$, $n$, $b$⟩;
end if

Procedure $CompareIds_p$
if $\max\{election\text{-}id_p, neighb\text{-}id_p[1, parity_p]\} < neighb\text{-}id_p[0, parity_p]$ then
    $election\text{-}id_p \leftarrow neighb\text{-}id_p[0, parity_p]$;
    $neighb\text{-}id_p[n, parity_p] \leftarrow \bot$ for $n = 0$ and $n = 1$;    $parity_p \leftarrow \neg parity_p$;
    send ⟨id, $election\text{-}id_p$, 0, $parity_p$⟩;
    if $neighb\text{-}id_p[n, parity_p] \neq \bot$ for $n = 0$ and $n = 1$ then
        perform procedure $CompareIds_p$;
    end if
else if $neighb\text{-}id_p[0, parity_p] < election\text{-}id_p$ then
    $active_p \leftarrow false$;
else
    $leader_p \leftarrow true$;
end if
```

A.15 Gallager-Humblet-Spira Minimum Spanning Tree Algorithm

The variable $parent_p$ contains p's parent toward the core edge of p's fragment. The name and level of p's fragment are stored in $name_p$ and $level_p$. Initially, $state_p$ has the value *find*; for simplicity, the state *sleep* and the corresponding wake-up phase are omitted. The channel states, $state_p[q]$ for each $q \in Neighbors_p$, initially are *basic*. While looking for a lowest-weight outgoing edge, p stores the optimal intermediate result in $best\text{-}weight_p$. If the optimal result was reported through the basic or branch edge pq, then $best\text{-}edge_p$ has the value q. While p is testing whether basic edge pq is outgoing, $test\text{-}edge_p$ has the value q. In $counter_p$, p keeps track of how many branch edges have reported their minimum values; it starts at 1 to account for the fact that p's parent in general does not report a value (except for the core nodes). In $parent\text{-}report_p$, a core node p can keep the value reported by its parent; if there is no report yet, its value is 0, while the value ∞ means that p's parent has reported no outgoing edges at its side. In $Connects_p$ and $Tests_p$, p stores incoming **connect** and **test** messages to which a reply is being delayed until the level of p's fragment is high enough.

```
$\{find, found\} state_p$;
$\{basic, branch, rejected\} state_p[r]$ for all $r \in Neighbors_p$;
real $name_p$;
nat $level_p$, $counter_p$;
dist $best\text{-}weight_p$, $parent\text{-}report_p$;
proc $parent_p$, $test\text{-}edge_p$, $best\text{-}edge_p$;
proc-nat-set $Connects_p$;
proc-real-nat-set $Tests_p$;

Initialization of $p$

determine the lowest-weight channel $pq$;
$state_p \leftarrow found$;   $state_p[q] \leftarrow branch$;   $counter_p \leftarrow 1$;   $parent\text{-}report_p \leftarrow 0$;
send ⟨connect, 0⟩ to $q$;

If $p$ receives ⟨connect, $\ell$⟩ from a neighbor $q$

if $\ell < level_p$ then
    send ⟨initiate, $name_p$, $level_p$, $state_p$⟩ to $q$;
    $state_p[q] \leftarrow branch$;
else if $state_p[q] = branch$ then
    send ⟨initiate, $weight(pq)$, $level_p + 1$, $find$⟩ to $q$;
else
    $Connects_p = Connects_p \cup \{(q, \ell)\}$;
end if

If $p$ receives ⟨initiate, $fn$, $\ell$, $st$⟩ from a neighbor $q$

$name_p \leftarrow fn$;   $level_p \leftarrow \ell$;   $state_p \leftarrow st$;   $parent_p \leftarrow q$;
$best\text{-}edge_p \leftarrow \perp$;   $best\text{-}weight_p \leftarrow \infty$;   $counter_p \leftarrow 1$;   $parent\text{-}report_p \leftarrow 0$;
for each $(q_0, \ell_0) \in Connects_p$ do
    if $\ell_0 < level_p$ then
        $state_p[q_0] \leftarrow branch$;   $Connects_p \leftarrow Connects_p \setminus \{(q_0, \ell_0)\}$;
    end if
end for
send ⟨initiate, $fn$, $\ell$, $st$⟩ to each $r \in Neighbors_p \setminus \{q\}$ with $state_p[r] = branch$;
for each $(q_1, fn_1, \ell_1) \in Tests_p$ do
    if $\ell_1 \leq level_p$ then
        perform procedure $ReplyTest_p(q_1)$;
        $Tests_p \leftarrow Tests_p \setminus \{(q_1, fn_1, \ell_1)\}$;
    end if
end for
if $st = find$ then
    perform procedure $FindMinimalOutgoing_p$;
```

```
end if
Procedure $FindMinimalOutgoing_p$
if $\{pr \mid r \in Neighbors_p$ and $state_p(pr) = basic\} \neq \emptyset$ then
    send ⟨test, $name_p$, $level_p$⟩ into the lowest-weight channel $pq$ in this collection;
    $test\text{-}edge_p \leftarrow q$;
else
    $test\text{-}edge_p \leftarrow \perp$;
    if $counter_p = |\{r \in Neighbors_p \mid state_p[r] = branch\}|$ then
        perform procedure $SendReport_p$
    end if
end if
If $p$ receives ⟨test, $fn$, $\ell$⟩ from a neighbor $q$
if $\ell \leq level_p$ then
    perform procedure $ReplyTest_p(q)$;
else
    $Tests_p = Tests_p \cup \{(q, fn, \ell)\}$;
end if
Procedure $ReplyTest_p(q)$
if $name_p \neq fn$ then
    send ⟨accept⟩ to $q$;
else
    $state_p[pq] \leftarrow rejected$;
    if $test\text{-}edge_p \neq q$ then
        send ⟨reject⟩ to $q$;
    else
        perform procedure $FindMinimalOutgoing_p$;
    end if
end if
If $p$ receives ⟨reject⟩ from a neighbor $q$
$state_p[q] \leftarrow rejected$;
perform procedure $FindMinimalOutgoing_p$;
If $p$ receives ⟨accept⟩ from a neighbor $q$
$test\text{-}edge_p \leftarrow \perp$;
if $weight(pq) < best\text{-}weight_p$ then
    $best\text{-}edge_p \leftarrow q$;    $best\text{-}weight_p \leftarrow weight(pq)$;
end if
if $counter_p = |\{r \in Neighbors_p \mid state_p[r] = branch\}|$ then
```

```
    perform procedure $SendReport_p$
end if
```

Procedure $SendReport_p$

```
$state_p \leftarrow found$;
send ⟨report, $best\text{-}weight_p$⟩ to $parent_p$;
if $parent\text{-}report_p > 0$ and $best\text{-}weight_p < parent\text{-}report_p$ then
    perform procedure $ChangeRoot_p$;
end if
```

If p receives ⟨**report**, λ⟩ from a neighbor q

```
if $q \neq parent_p$ then
    $counter_p \leftarrow counter_p + 1$;
    if $\lambda < best\text{-}weight_p$ then
        $best\text{-}edge_p \leftarrow q$;    $best\text{-}weight_p \leftarrow \lambda$;
    end if
    if $counter_p = |\{r \in Neighbors_p \mid state_p[r] = branch\}|$ and $test\text{-}edge_p = \perp$ then
        perform procedure $SendReport_p$
    end if
else if $state_p = find$ then
    $parent\text{-}report_p \leftarrow \lambda$;
else
    if $best\text{-}weight_p < \lambda$ then
        perform procedure $ChangeRoot_p$;
    else if $\lambda = \infty$ then
        terminate;
    end if
end if
```

Procedure $ChangeRoot_p$

```
if $state_p[best\text{-}edge_p] = branch$ then
    send ⟨changeroot⟩ to $best\text{-}edge_p$;
else
    $state_p[best\text{-}edge_p] \leftarrow branch$;
    send ⟨connect, $level_p$⟩ to $best\text{-}edge_p$;
    if $(best\text{-}edge_p, level_p) \in Connects_p$ then
        send ⟨initiate, $best\text{-}weight_p$, $level_p + 1$, $find$⟩ to $best\text{-}edge_p$;
        $Connects_p \leftarrow Connects_p \setminus \{(best\text{-}edge_p, level_p)\}$;
    end if
end if
```

If p receives ⟨**changeroot**⟩

```
perform procedure $ChangeRoot_p$;
```

A.16 IEEE 1394 Election Algorithm

The variable *parent*$_p$ contains the parent of p in the spanning tree. The variable *received*$_p[q]$ is set when p receives a parent request from a neighbor q to which p has not sent a parent request. If p gets into root contention and chooses to start a timer, it sets *waiting*$_p$. If p terminates as the leader, it sets *leader*$_p$.

```
bool $leader_p$, $waiting_p$, $received_p[r]$ for all $r \in Neighbors_p$;
proc $parent_p$;

Initialization of $p$
perform procedure $SendRequest_p$;

Procedure $SendRequest_p$
if $|\{r \in Neighbors_p \mid received_p[r] = false\}| = 1$ then
    send ⟨parent-req⟩ to the only $q \in Neighbors_p$ with $received_p[q] = false$;
    $parent_p \leftarrow q$;
end if

If $p$ receives ⟨parent-req⟩ from a neighbor $q$
if $q \neq parent_p$ then
    $received_p[q] \leftarrow true$;
    send ⟨ack⟩ to $q$;
    perform procedure $SendRequest_p$;
else if $waiting_p = false$ then
    perform procedure $RootContention_p$;
else
    $leader_p \leftarrow true$;
end if

If $p$ receives ⟨ack⟩ from $parent_p$
terminate;

Procedure $RootContention_p$
either send ⟨parent-req⟩ to $q$ and $waiting_p \leftarrow false$,
or start a timer and $waiting_p \leftarrow true$;

If a timeout occurs at $p$
perform procedure $RootContention_p$;
```

A.17 Awerbuch's Synchronizer

The variable $parent_p$ is the parent of p in the spanning tree within its cluster, $Children_p$ contains the children of p in this spanning tree, and $Designated_p$ contains the processes q for which there is a designated channel pq. Note that these three values are fixed after the initialization phase. In $1st\text{-}counter_p$ and $2nd\text{-}counter_p$, p keeps track of how many messages still need to be received in the first and second phases of this synchronizer, respectively.

```
**nat** $1st\text{-}counter_p$, $2nd\text{-}counter_p$;
**proc** $parent_p$;
**proc-set** $Children_p$, $Designated_p$;

Initialization
The network is divided into clusters, and within each cluster a spanning tree is
built. Between each pair of distinct clusters that are connected by a channel, one of
these connecting channels is labeled as designated. Furthermore, a wake-up phase
makes sure that each process starts its first pulse, meaning that it performs the
procedure NewPulse.

Procedure $NewPulse_p$
send $k \geq 0$ basic messages;
$1st\text{-}counter_p \leftarrow k + |Children_p|$;    $2nd\text{-}counter_p \leftarrow |Children_p| + |Designated_p|$;
perform procedure $FirstReport_p$;

If $p$ receives a basic message from a neighbor $q$
send ⟨**ack**⟩ to $q$;

If $p$ receives ⟨**ack**⟩ or ⟨**safe**⟩
$1st\text{-}counter_p \leftarrow 1st\text{-}counter_p - 1$;
perform procedure $FirstReport_p$;

Procedure $FirstReport_p$
**if** $1st\text{-}counter_p = 0$ **then**
    **if** $parent_p \neq \perp$ **then**
        send ⟨**safe**⟩ to $parent_p$;
    **else**
        perform procedure $SendNext_p$;
    **end if**
**end if**

Procedure $SendNext_p$
send ⟨**next**⟩ to each $q \in Children_p$;
send ⟨cluster-safe⟩ to each $r \in Designated_p$;
perform procedure $SecondReport_p$;

If $p$ receives ⟨next⟩

perform procedure $SendNext_p$;

If $p$ receives ⟨cluster-safe⟩

$2nd\text{-}counter_p \leftarrow 2nd\text{-}counter_p - 1$;
perform procedure $SecondReport_p$;

Procedure $SecondReport_p$

if $2nd\text{-}counter_p = 0$ then
    if $parent_p \neq \perp$ then
        send ⟨cluster-safe⟩ to $parent_p$;
    else
        perform procedure $SendClusterNext_p$;
    end if
end if

Procedure $SendClusterNext_p$

send ⟨cluster-next⟩ to each $q \in Children_p$;
perform procedure $NewPulse_p$;

If $p$ receives ⟨cluster-next⟩

perform procedure $SendClusterNext_p$;
```

A.18 Ricart-Agrawala Mutual Exclusion Algorithm

We use the lexicographical order on pairs (t, i) with t a time stamp and i a process index. The variable $clock_i$ represents the clock value at p_i; it starts at 1. In $req\text{-}stamp_i$, p_i stores the time stamp of its current request; if there is none, $req\text{-}stamp_i = 0$. The number of permissions that p_i has received for its current request is maintained in $counter_i$. In $Pending_i$, p_i keeps track of the processes it has received a request from but to which it has not yet sent permission. The Carvalho-Roucairol optimization is taken into account. In $Requests_i$, p_i keeps track of the processes to which it must send (or has sent) its next (or current) request.

```
nat $clock_i$, $req\text{-}stamp_i$, $counter_i$;
proc-set $Pending_i$, $Requests_i$;

Initialization of $p_i$

$Requests_i \leftarrow Neighbors_i$;    $clock_i \leftarrow 1$;
```

```
If $p_i$ wants to enter its critical section
if $Requests_i \neq \emptyset$ then
    send ⟨request, $clock_i$, $i$⟩ to each $q \in Requests_i$;
    $req\text{-}stamp_i \leftarrow clock_i$;   $counter_i \leftarrow 0$;
else
    perform procedure $CriticalSection_p$;
end if

If $p_i$ receives ⟨permission⟩
$counter_i \leftarrow counter_i + 1$;
if $counter_i = |Requests_i|$ then
    perform procedure $CriticalSection_p$;
end if

Procedure $CriticalSection_p$
enter critical section;
exit critical section;
send ⟨permission⟩ to each $q \in Pending_i$;
$req\text{-}stamp_i \leftarrow 0$;   $Requests_i \leftarrow Pending_i$;   $Pending_i \leftarrow \emptyset$;

If $p_i$ receives ⟨request, $t, j$⟩ from a $p_j$
$clock_i \leftarrow \max\{clock_i, t + 1\}$;
if $req\text{-}stamp_i = 0$ or $(t, j) < (req\text{-}stamp_i, i)$ then
    send ⟨permission⟩ to $p_j$;
    if $p_j \notin Requests_i$ then
        $Requests_i \leftarrow Requests_i \cup \{p_j\}$;
        if $req\text{-}stamp_i > 0$ then
            send ⟨request, $req\text{-}stamp_i$, $i$⟩ to $p_j$;
        end if
    end if
else
    $Pending_i \leftarrow Pending_i \cup \{p_j\}$;
end if
```

A.19 Raymond's Mutual Exclusion Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree. The queue $pending_p$ contains the children of p in the tree that have asked for the token and possibly p itself.

```
proc $parent_p$;
```

```
proc-queue $pending_p$;

If $p$ is the initiator
Initiate a wave that determines a spanning tree of the network, captured by values
of $parent_r$ for all $r \in Processes$, with $p$ as root;

If $p$ wants to enter its critical section
if $parent_p \neq \perp$ then
    $pending_p \leftarrow append(pending_p, p)$;
    if $head(pending_p) = p$ then
        send ⟨request⟩ to $parent_p$;
    end if
else
    perform procedure $CriticalSection_p$;
end if

If $p$ receives ⟨request⟩ from a neighbor $q$
$pending_p \leftarrow append(pending_p, q)$;
if $head(pending_p) = q$ then
    if $parent_p \neq \perp$ then
        send ⟨request⟩ to $parent_p$;
    else if $p$ is not in its critical section then
        perform procedure $SendToken_p$;
    end if
end if

If $p$ receives ⟨token⟩
if $head(pending_p) \neq p$ then
    perform procedure $SendToken_p$;
else
    $parent_p \leftarrow \perp$;    $pending_p \leftarrow tail(pending_p)$;
    perform procedure $CriticalSection_p$;
end if

Procedure $SendToken_p$
$parent_p \leftarrow head(pending_p)$;    $pending_p \leftarrow tail(pending_p)$;
send ⟨token⟩ to $parent_p$;
if $pending_p \neq \emptyset$ then
    send ⟨request⟩ to $parent_p$;
end if

Procedure $CriticalSection_p$
```

```
enter critical section;
exit critical section;
if $pending_p \neq \emptyset$ then
    perform procedure $SendToken_p$;
end if
```

A.20 Agrawal–El Abbadi Mutual Exclusion Algorithm

The variable $requests_p$ contains a queue of processes from which p must still obtain permission to enter its critical section. The set $Permissions_p$ contains the processes from which p has received permission during its current attempt to become privileged. The queue $pending_p$ contains the processes from which p has received a request; it has only sent permission to the head of this queue. (We recall that p may have to ask permission from itself.) We assume that $N = 2^k - 1$ for some $k > 1$, so that the binary tree has depth k - 1. The constant *root* denotes the root node of the binary tree, and for any nonleaf q in the tree, *left-child*(q) and *right-child*(q) denote its children at the left and right, respectively. Processes may crash, and they are provided with a complete and strongly accurate failure detector. When p detects that another process has crashed, it puts the corresponding process ID in the set $Crashed_p$.

```
proc-queue $requests_p$, $pending_p$;
proc-set $Permissions_p$, $Crashed_p$;

If p wants to enter its critical section
$requests_p \leftarrow append(\emptyset, root)$;
perform procedure $SendRequest_p$;

Procedure $SendRequest_p$
if $head(requests_p) \notin Crashed_p$ then
    send ⟨request⟩ to $head(requests_p)$;
else
    perform procedure $HeadRequestsCrashed_p$;
end if

Procedure $HeadRequestsCrashed_p$
if $head(requests_p)$ is not a leaf of the binary tree then
    $requests_p \leftarrow append(append(tail(requests_p), left\text{-}child(head(requests_p))),$
                  $right\text{-}child(head(requests_p)))$;
    perform procedure $SendRequest_p$;
else
    $Permissions_p \leftarrow \emptyset$;
    start a new attempt to enter the critical section;
end if

If $p$ receives ⟨request⟩ from a process $q$
$pending_p \leftarrow append(pending_p, q)$;
if $head(pending_p) = q$ then
    perform procedure $SendPermission_p$;
end if

Procedure $SendPermission_p$
if $pending_p \neq \emptyset$ then
    if $head(pending_p) \notin Crashed_p$ then
        send ⟨permission⟩ to $head(pending_p)$;
    else
        $pending_p \leftarrow tail(pending_p)$;
        perform procedure $SendPermission_p$;
    end if
end if

If $p$ receives ⟨permission⟩ from process $q$
$Permissions_p \leftarrow Permissions_p \cup \{q\}$;   $requests_p \leftarrow tail(requests_p)$;
if $q$ is not a leaf of the binary tree then
    either $requests_p \leftarrow append(requests_p, left\text{-}child(q))$
    or $requests_p \leftarrow append(requests_p, right\text{-}child(q))$;
end if
if $requests_p \neq \emptyset$ then
    perform procedure $SendRequest_p$;
else
    enter critical section;
    exit critical section;
    send ⟨released⟩ to each $r \in Permissions_p$;
    $Permissions_p \leftarrow \emptyset$;
end if

If $p$ receives ⟨released⟩
$pending_p \leftarrow tail(pending_p)$;
perform procedure $SendPermission_p$;

If $p$ detects that a process $q$ has crashed
$Crashed_p \leftarrow Crashed_p \cup \{q\}$;
if $head(requests_p) = q$ then
    perform procedure $HeadRequestsCrashed_p$;
```

```
end if
if $head(pending_p) = q$ then
    $pending_p \leftarrow tail(pending_p)$;
    perform procedure $SendPermission_p$;
end if
```

A.21 MCS Queue Lock

Since processes can use the same element for each lock access, we take the liberty of representing the queue of waiting processes by using process IDs instead of elements. The multi-writer register $wait_p$ is *true* as long as p must wait to get the lock. The multi-writer register $succ_p$ points to the successor of p in the queue of waiting processes. The single-writer register $pred_p$ points to the predecessor of p in the queue of waiting processes. When a process arrives at this queue, it assigns its process ID to the multi-writer register *last*. The operation *last.get-and-set*(p) assigns the value p to *last* and returns the previous value of *last*, all in one atomic step. And *last.compare-and-set*($p, \perp$) in one atomic operation reads the value of *last* and either assigns the value $\perp$ to *last*, if its current value is p, or leaves the value of *last* unchanged otherwise. In the first case, this operation returns *true*, while in the second case it returns *false*.

```
bool $wait_p$;
proc $last, succ_p, pred_p$;

If $p$ wants to enter its critical section
$pred_p \leftarrow last.get\text{-}and\text{-}set(p)$;
if $pred_p \neq \perp$ then
    $wait_p \leftarrow true$;   $succ_{pred_p} \leftarrow p$;
    while $wait_p = true$ do
        {};
    end while
end if
enter critical section;
exit critical section;
if $succ_p \neq \perp$ then
    $wait_{succ_p} \leftarrow false$;
else if $last.compare\text{-}and\text{-}set(p, \perp)$ returns $false$ then
    while $succ_p = \perp$ do
        {};
    end while
    $wait_{succ_p} \leftarrow false$;
end if
```

A.22 CLH Queue Lock with Timeouts

The data type **pointer** consists of pointers to an element, with `null` as default initial value. If p wants to enter its critical section, it creates an element ε containing a pointer $pred_\varepsilon$. Let element ε' denote the nearest nonabandoned predecessor of ε in the queue. The pointer $pred_p$ points to ε'. In $pred\text{-}pred_p$, p repeatedly stores the value of $pred_{\varepsilon'}$. If p decides to abandon its attempt to get the lock and has a successor in the queue, then it lets $pred_\varepsilon$ point to ε'. The multi-writer register *last* points to the last element in the queue.

```
pointer $last$, $pred_p$, $pred\text{-}pred_p$, $pred_\varepsilon$ for all elements $\varepsilon$;

If $p$ wants to enter its critical section
create an element $\varepsilon$;
$pred_p \leftarrow last.get\text{-}and\text{-}set(\varepsilon)$;
if $pred_p$ = null then
    perform procedure $CriticalSection_p(\varepsilon)$;
else
    while no $timeout$ occurs do
        $pred\text{-}pred_p \leftarrow pred_{pred_p}$;
        if $pred\text{-}pred_p$ = released then
            perform procedure $CriticalSection_p(\varepsilon)$;
        else if $pred\text{-}pred_p \neq$ null then
            $pred_p \leftarrow pred\text{-}pred_p$;
        end if
    end while
    if $last.compare\text{-}and\text{-}set(\varepsilon, pred_p)$ returns $false$ then
        $pred_\varepsilon \leftarrow pred_p$;
    end if
    abandon the attempt to take the lock;
end if

Procedure $CriticalSection_p(\varepsilon)$
enter critical section;
exit critical section;
if $last.compare\text{-}and\text{-}set(\varepsilon$, null) returns $false$ then
    $pred_\varepsilon \leftarrow$ released;
end if
$terminate$;
```

A.23 Afek-Kutten-Yung Spanning Tree Algorithm

Self-stabilizing algorithms are always defined in a shared-memory framework. Therefore, the message-passing description of the Afek-Kutten-Yung spanning tree algorithm in section 17.3 is here cast in shared variables. We recall that a variable can be initialized with any value in its domain.

The variable $root_p$ is the root of the spanning tree according to p, $parent_p$ represents the parent of p in the spanning tree, and $dist_p$ is the distance value of p toward the root. The variables req_p, $from_p$, to_p, and $direction_p$ deal with join requests and corresponding grant messages. The process ID of the process that originally issued the request is stored in req_p, the neighbor from which p received the request is stored in $from_p$, the neighbor to which p forwarded the request is stored in to_p, and whether a request is being forwarded to the root of the fragment or a grant message is being forwarded to the process that originally issued the request is remembered in $direction_p$. The variable $toggle_p$ makes sure that p performs an event only when all its neighbors have copied the current values of p's local variables; $toggle_q(p)$ represents the copy at neighbor q of the value of $toggle_p$. It is assumed that a process copies the values of all local variables of a neighbor in one atomic step.

We use the following abbreviations. $AmRoot_p$ states that p considers itself the root:

$$parent_p = \perp \ \wedge \ root_p = p \ \wedge \ dist_p = 0.$$

$NotRoot_p$ states that p does not consider itself the root and that the values of p's local variables are in line with those of its parent:

$$\begin{aligned} &parent_p \in Neighbors_p \ \wedge \ root_p > p \\ \wedge \ &root_p = root_{parent_p} \ \wedge \ dist_p = dist_{parent_p} + 1. \end{aligned}$$

$MaxRoot_p$ states that no neighbor of p has a root value greater than $root_p$:

$$root_p \geq root_r \text{ for all } r \in Neighbors_p.$$

The network is stable, with the process with the largest ID as root, if at each process p either $AmRoot_p$ or $NotRoot_p$ holds, as well as $MaxRoot_p$.

In the following pseudocode, p repeatedly copies the values of the local variables of its neighbors, checks whether all its neighbors have copied the current values of p's local variables, and, if so, tries to perform one of several possible events. First of all, if $NotRoot_p \wedge MaxRoot_p$ does not hold and p does not yet consider itself the root, then p makes itself the root. The second kind of event arises if $MaxRoot_p$ does not hold (and so, since p skipped the first case, $AmRoot_p$ does hold). Then p asks a neighbor with a maximum root value to become its parent, if p is not already making such a request to a neighbor q, expressed by (the negation of) the predicate $Asking_p(q)$:

$$\begin{aligned} & root_q \geq root_r \text{ for all } r \in Neighbors_p \\ \wedge\ & req_p = from_p = p \wedge to_p = q \wedge direction_p = \texttt{ask}. \end{aligned}$$

Or such a request by p may be granted by a neighbor q, expressed by the predicate $Granted_p(q)$:

$$req_q = req_p \wedge from_q = from_p \wedge direction_q = \texttt{grant} \wedge direction_p = \texttt{ask},$$

where we take q to be to_p. It is, moreover, required that p issued a request to to_p, expressed by the predicate $Requestor_p$:

$$to_p \in Neighbors_p \wedge root_{to_p} > p \wedge req_p = from_p = p.$$

In this case, to_p becomes p's parent. The third kind of event arises when p is not yet handling a request from a neighbor q, expressed by (the negation of) the predicate $Handling_p(q)$:

$$req_q = req_p \wedge from_p = q \wedge to_q = p \wedge to_p = parent_p \wedge direction_q = \texttt{ask}.$$

Here q should be either a root that issued a join request or a child of p in the spanning tree, expressed by the predicate $Request_p(q)$:

$$(AmRoot_q \wedge req_q = from_q = q) \vee (parent_q = p \wedge req_q \notin \{q, \perp\}).$$

If the four variables that capture requests are not all undefined, expressed by (the negation of) the predicate $NotHandling_p$,

$$req_p = \perp \wedge from_p = \perp \wedge to_p = \perp \wedge direction_p = \perp,$$

then p sets the values of these four variables to $\perp$. Otherwise, p forwards a request, but only if $from_{parent_p} \neq p$ (allowing $parent_p$ to first reset its join request variables). The fourth kind of event is a root p that is handling a request of a neighbor setting $direction_p$ to *grant*. Finally, the fifth kind of event is a nonroot p that finds that its request has been granted by its parent setting $direction_p$ to *grant*. Note that for the third, fourth, and fifth kinds of event, $AmRoot_p \vee NotRoot_p$ (because p skipped the first case) and $MaxRoot_p$ (because p skipped the second case). And for the fourth and fifth kinds of event, $Request_p(q) \wedge Handling_p(q)$ for some $q \in Neighbors_p$ (because p skipped the third case).

```
bool $toggle_p$, $toggle_p(r)$ for all $r \in Neighbors_p$;
dist $dist_p$;
proc $parent_p$, $root_p$, $req_p$, $from_p$, $to_p$;
{ask, grant, $\perp$}$direction_p$;

while $true$ do
    copy the values of variables of all neighbors into a local copy;
    if $toggle_r(p) = toggle_p$ for all $r \in Neighbors_p$ then
        if $\neg(NotRoot_p \wedge MaxRoot_p) \wedge \neg AmRoot_p$ then
            $parent_p \leftarrow \perp$;   $root_p \leftarrow p$;   $dist_p \leftarrow 0$;
        else if $\neg MaxRoot_p$ then
            if $\neg Asking_p(r)$ for all $r \in Neighbors_p$ then
                $req_p \leftarrow p$;   $from_p \leftarrow p$;   $direction_p \leftarrow$ ask;
                $to_p \leftarrow q$ for a $q \in Neighbors_p$ with $root_q$ as large as possible;
            else if $Requestor_p \wedge Granted_p(to_p)$ then
                $parent_p \leftarrow to_p$;   $root_p \leftarrow root_{to_p}$   $dist_p \leftarrow dist_{to_p} + 1$;
                $req_p \leftarrow \perp$;   $from_p \leftarrow \perp$;   $to_p \leftarrow \perp$;   $direction_p \leftarrow \perp$;
            end if
        else if $\neg(Request_p(r) \wedge Handling_p(r))$ for all $r \in Neighbors_p$ then
            if $\neg NotHandling_p$ then
                $req_p \leftarrow \perp$;   $from_p \leftarrow \perp$;   $to_p \leftarrow \perp$;   $direction_p \leftarrow \perp$;
            else if $from_{parent_p} \neq p \wedge Request_p(q)$ for some $q \in Neighbors_p$ then
                $req_p \leftarrow req_q$;   $from_p \leftarrow q$;   $to_p \leftarrow parent_p$;   $direction_p \leftarrow$ ask;
            end if
        else if $AmRoot_p \wedge direction_p =$ ask then
            $direction_p \leftarrow$ grant;
        else if $Granted_p(parent_p)$ then
            $direction_p \leftarrow$ grant;
        end if
        $toggle_p \leftarrow \neg toggle_p$;
    end if
end while
```

References

1. Y. Afek, S. Kutten, M. Yung (1997), *The local detection paradigm and its applications to self-stabilization*, Theoretical Computer Science, 186, pp. 199–229.
2. D. Agrawal, A. El Abbadi (1991), *An efficient and fault-tolerant solution for distributed mutual exclusion*, ACM Transactions on Computer Systems, 9, pp. 1–20.
3. T. E. Anderson (1990), *The performance of spin lock alternatives for shared-memory multiprocessors*, IEEE Transactions on Parallel and Distributed Systems, 1, pp. 6–16.
4. D. Angluin (1980), *Local and global properties in networks of processors*, in (R. E. Miller, S. Ginsburg, W. A. Burkhard, R. J. Lipton, eds.) Proceedings of the 12th Symposium on Theory of Computing, pp. 82–93, ACM.
5. A. Arora, M. G. Gouda (1994), *Distributed reset*, IEEE Transactions on Computers, 43, pp. 1026–1038.
6. B. Awerbuch (1985), *Complexity of network synchronization*, Journal of the ACM, 32, pp. 804–823.
7. B. Awerbuch (1985), *A new distributed depth-first-search algorithm*, Information Processing Letters, 20, pp. 147–150.
8. R. Bakhshi, J. Endrullis, W. J. Fokkink, J. Pang (2011), *Fast leader election in anonymous rings with bounded expected delay*, Information Processing Letters, 111, pp. 864–870.
9. R. Bakhshi, W. J. Fokkink, J. Pang, J. C. van de Pol (2008), *Leader election in anonymous rings: Franklin goes probabilistic*, in (G. Ausiello, J. Karhumäki, G. Mauri, C.-H. L. Ong, eds.) Proceedings of the 5th IFIP Conference on Theoretical Computer Science, pp. 57–72, Springer.
10. C. H. Bennett, G. Brassard (1984), *Quantum cryptography: Public key distribution and coin tossing*, in Proceedings of the Conference on Computers, Systems and Signal Processing, pp. 175–179, IEEE.

11. P. A. Bernstein, D. W. Shipman, J. B. Rothnie Jr. (1980), *Concurrency control in a system for distributed databases (SDD-1)*, ACM Transactions on Database Systems, 5, pp. 18–51.

12. D. I. Bevan (1987), *Distributed garbage collection using reference counting*, in (J. W. de Bakker, A. J. Nijman, P. C. Treleaven, eds.) Proceedings of the 1st Conference on Parallel Architectures and Languages Europe, vol. 259 of Lecture Notes in Computer Science, pp. 176–187, Springer.

13. G. Bracha, S. Toueg (1985), *Asynchronous consensus and broadcast protocols*, Journal of the ACM, 32, pp. 824–840.

14. G. Bracha, S. Toueg (1987), *Distributed deadlock detection*, Distributed Computing, 2, pp. 127–138.

15. J. E. Burns, N. A. Lynch (1993), *Bounds on shared memory for mutual exclusion*, Information and Computation, 107, pp. 171–184.

16. O. S. F. Carvalho, G. Roucairol (1983), *On mutual exclusion in computer networks*, Communications of the ACM, 26, pp. 146–147.

17. T. D. Chandra, S. Toueg (1996), *Unreliable failure detectors for reliable distributed systems*, Journal of the ACM, 43, pp. 225–267.

18. K. M. Chandy, L. Lamport (1985), *Distributed snapshots: Determining global states of distributed systems*, ACM Transactions on Computer Systems, 3, pp. 63–75.

19. K. M. Chandy, J. Misra (1982), *Distributed computation on graphs: Shortest path algorithms*, Communications of the ACM, 25, pp. 833–837.

20. E. J. H. Chang (1982), *Echo algorithms: Depth parallel operations on general graphs*, IEEE Transactions on Software Engineering, 8, pp. 391–401.

21. E. J. H. Chang, R. Roberts (1979), *An improved algorithm for decentralized extrema-finding in circular configurations of processes*, Communications of the ACM, 22, pp. 281–283.

22. T.-Y. Cheung (1983), *Graph traversal techniques and the maximum flow problem in distributed computation*, IEEE Transactions on Software Engineering, 9, pp. 504–512.

23. C.-T. Chou, I. Cidon, I. S. Gopal, S. Zaks (1990), *Synchronizing asynchronous bounded delay networks*, IEEE Transactions on Communications, 38, pp. 144–147.

24. I. Cidon (1988), *Yet another distributed depth-first-search algorithm*, Information Processing Letters, 26, pp. 301–305.

25. T. S. Craig (1993), *Building FIFO and priority-queueing spin locks from atomic swap*, Technical Report TR 93-02-02, University of Washington.

26. B. W. Diffie, M. E. Hellman (1976), *New directions in cryptography*, IEEE Transactions on Information Theory, 22, pp. 644–654.

27. E. W. Dijkstra (1974), *Self-stabilizing systems in spite of distributed control*, Communications of the ACM, 17, pp. 643–644.

28. E. W. Dijkstra (1987), *Shmuel Safra's version of termination detection*, vol. 998 of EWD manuscripts, The University of Texas at Austin.

29. E. W. Dijkstra, C. S. Scholten (1980), *Termination detection for diffusing computations*, Information Processing Letters, 11, pp. 1–4.

30. D. Dolev, M. M. Klawe, M. Rodeh (1982), *An O(n log n) unidirectional distributed algorithm for extrema finding in a circle*, Journal of Algorithms, 3, pp. 245–260.

31. D. Dolev, H. R. Strong (1983), *Authenticated algorithms for Byzantine agreement*, SIAM Journal on Computing, 12, pp. 656–666.

32. C. Dwork, M. Naor (1992), *Pricing via processing or combatting junk mail*, in (E. F. Brickell, ed.) Proceedings of the 12th Cryptology Conference, vol. 740 of Lecture Notes in Computer Science, pp. 139-âĂŞ147, Springer.

33. K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger (1976), *The notions of consistency and predicate locks in a database system*, Communications of the ACM, 19, pp. 624–633.

34. C. J. Fidge (1988), *Timestamps in message-passing systems that preserve the partial ordering*, in (K. Raymond, ed.) Proceedings of the 11th Australian Computer Science Conference, pp. 56–66.

35. M. J. Fischer, N. A. Lynch, M. Paterson (1985), *Impossibility of distributed consensus with one faulty process*, Journal of the ACM, 32, pp. 374–382.

36. W. J. Fokkink, J.-H. Hoepman, J. Pang (2005), *A note on K-state self-stabilization in a ring with K = N*, Nordic Journal of Computing, 12, pp. 18–26.

37. W. J. Fokkink, J. Pang (2006), *Variations on Itai-Rodeh leader election for anonymous rings and their analysis in PRISM*, Journal of Universal Computer Science, 12, pp. 981–1006.

38. W. R. Franklin (1982), *On an improved algorithm for decentralized extrema-finding in circular configurations of processes*, Communications of the ACM, 25, pp. 336–337.

39. G. N. Frederickson (1985), *A single source shortest path algorithm for a planar distributed network*, in (K. Mehlhorn, ed.) Proceedings of the 2nd Symposium on Theoretical Aspects of Computer Science, vol. 182 of Lecture Notes in Computer Science, pp. 143–150, Springer.

40. R. G. Gallager, P. A. Humblet, P. M. Spira (1983), *A distributed algorithm for minimum-weight spanning trees*, ACM Transactions on Programming Languages and Systems, 5, pp. 66–77.

41. J. N. Gray (1978), *Notes on data base operating systems*, in (M. J. Flynn, J. Gray, A. K. Jones, K. Lagally, H. Opderbeck, G. J. Popek, B. Randell, J. H. Saltzer, H.-R. Wiehle, eds.) Operating Systems, An Advanced Course, vol. 60 of Lecture Notes in Computer Science, pp. 393–481, Springer.

42. T. Härder, A. Reuter (1983), *Principles of transaction-oriented database recovery*, ACM Computing Surveys, 15, pp. 287-âĂŞ317.

43. D. Hensgen, R. A. Finkel, U. Manber (1988), *Two algorithms for barrier synchronization*, International Journal of Parallel Programming, 17, pp. 1–17.

44. M. Herlihy (1991), *Wait-free synchronization*, ACM Transactions on Programming Languages and Systems, 13, pp. 124–149.

45. M. Herlihy, J. E. B. Moss (1993), *Transactional memory: Architectural support for lock-free data structures*, in (A. J. Smith, ed.) Proceedings of the 20th Symposium on Computer Architecture, pp. 289–300, ACM.

46. M. Herlihy, N. Shavit (2008), *The Art of Multiprocessor Programming*, Morgan Kaufmann.

47. IEEE Computer Society (1996), *IEEE standard for a high performance serial bus*, Technical Report Std. 1394-1995, IEEE.

48. A. Itai, M. Rodeh (1990), *Symmetry breaking in distributed networks*, Information and Computation, 88, pp. 60–87.

49. V. Jacobson (1988), *Congestion avoidance and control*, in (V. G. Cerf, ed.) Proceedings of the 3rd Symposium on Communications Architectures and Protocols, pp. 314–329, ACM.

50. C. P. Kruskal, L. Rudolph, M. Snir (1988), *Efficient synchronization on multiprocessors with shared memory*, ACM Transactions on Programming Languages and Systems, 10, pp. 579–601.

51. H. T. Kung, J. T. Robinson (1981), *On optimistic methods for concurrency control*, ACM Transactions on Databases, 6, pp. 213–226.

52. T. H. Lai, T. H. Yang (1987), *On distributed snapshots*, Information Processing Letters, 25, pp. 153–158.

53. L. Lamport (1974), *A new solution of Dijkstra's concurrent programming problem*, Communications of the ACM, 17, pp. 453–455.

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

55. L. Lamport (1987), *A fast mutual exclusion algorithm*, ACM Transactions on Computer Systems, 5, pp. 1–11.

56. L. Lamport, R. E. Shostak, M. C. Pease (1982), *The Byzantine generals problem*, ACM Transactions on Programming Languages and Systems, 4, pp. 382–401.

57. B. W. Lampson (1981), *Atomic transactions*, in (B. W. Lampson, M. Paul, H.-J. Siegert, eds.) Distributed Systems–Architecture and Implementation, An Advanced Course, vol. 105 of Lecture Notes in Computer Science, pp. 246–265, Springer.

58. J. P. Lehoczky, S. Ramos-Thuel (1992), *An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems*, in Proceedings of the 13th Real-Time Systems Symposium, pp. 110–123, IEEE.

59. H. Lieberman, C. Hewitt (1983), *A real-time garbage collector based on the lifetimes of objects*, Communications of the ACM, 26, pp. 419–429.

60. J. W. Liu (2000), *Real-Time Systems*, Prentice-Hall.

61. P. S. Magnusson, A. Landin, E. Hagersten (1994), *Queue locks on cache coherent multiprocessors*, in (H. J. Siegel, ed.) Proceedings of the 8th Symposium on Parallel Processing, pp. 165–171, IEEE.

62. S. R. Mahaney, F. B. Schneider (1985), *Inexact agreement: Accuracy, precision, and graceful degradation*, in (M. A. Malcolm, H. R. Strong, eds.) Proceedings of the 4th Symposium on Principles of Distributed Computing, pp. 237–249, ACM.

63. F. Mattern (1989), *Global quiescence detection based on credit distribution and recovery*, Information Processing Letters, 30, pp. 195–200.

64. F. Mattern (1989), *Virtual time and global states of distributed systems*, in (M. Corsnard, ed.) Proceedings of the Workshop on Parallel and Distributed Algorithms, pp. 215–226, North-Holland/Elsevier.

65. J. M. McQuillan (1974), *Adaptive Routing for Distributed Computer Networks*, PhD thesis, Harvard University.

66. J. M. McQuillan, I. Richer, E. C. Rosen (1980), *The new routing algorithm for ARPANET*, IEEE Transactions on Communications, 28, pp. 711–719.

67. J. M. Mellor-Crummey, M. L. Scott (1991), *Algorithms for scalable synchronization on shared-memory multiprocessors*, ACM Transactions on Computer Systems, 9, pp. 21–65.

68. P. M. Merlin, P. J. Schweitzer (1980), *Deadlock avoidance in store-and-forward networks I: Store-and-forward deadlock*, IEEE Transactions on Communications, 28, pp. 345–354.

69. P. M. Merlin, A. Segall (1979), *A failsafe distributed routing protocol*, IEEE Transactions on Communications, 27, pp. 1280–1287.

70. Satoshi Nakamoto (2008), *Bitcoin: A peer-to-peer electronic cash system*, Cryptography mailing list

at metzdowd.com.

71. J. K. Pachl, E. Korach, D. Rotem (1984), *Lower bounds for distributed maximum-finding algorithms*, Journal of the ACM, 31, pp. 905–918.

72. G. L. Peterson (1981), *Myths about the mutual exclusion problem*, Information Processing Letters, 12, pp. 115–116.

73. G. L. Peterson (1982), *An O(n* log *n) unidirectional algorithm for the circular extrema problem*, ACM Transactions on Programming Languages and Systems, 4, pp. 758–762.

74. G. L. Peterson, M. J. Fischer (1977), *Economical solutions for the critical section problem in a distributed system*, in (J. E. Hopcroft, E. P. Friedman, M. A. Harrison, eds.) Proceedings of the 9th Symposium on Theory of Computing, pp. 91–97, ACM.

75. S. L. Peterson, P. Kearns (1993), *Rollback based on vector time*, in Proceedings of the 12th Symposium on Reliable Distributed Systems, pp. 68–77, IEEE.

76. J. M. Piquer (1991), *Indirect reference counting: A distributed garbage collection algorithm*, in (E. H. L. Aarts, J. van Leeuwen, M. Rem, eds.) Proceedings of the 3rd Conference on Parallel Architectures and Languages Europe, vol. 505 of Lecture Notes in Computer Science, pp. 150–165, Springer.

77. R. Rajkumar, L. Sha, J. P. Lehoczky (1988), *Real-time synchronization protocols for multiprocessors*, in Proceedings of the 9th Real-Time Systems Symposium, pp. 259–269, IEEE.

78. S. P. Rana (1983), *A distributed solution of the distributed termination problem*, Information Processing Letters, 17, pp. 43–46.

79. K. Raymond (1989), *A tree-based algorithm for distributed mutual exclusion*, ACM Transactions on Computer Systems, 7, pp. 61–77.

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

81. R. L. Rivest, A. Shamir, L. M. Adleman (1978), *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM, 21, pp. 120–126.

82. M. L. Scott, W. N. Scherer III (2001), *Scalable queue-based spin locks with timeout*, in (M. T. Heath, A. Lumsdaine, eds.) Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming, pp. 44–52, ACM.

83. A. Segall (1983), *Distributed network protocols*, IEEE Transactions on Information Theory, 29, pp. 23–34.

84. L. Sha, R. Rajkumar, J. P. Lehoczky (1990), *Priority inheritance protocols: An approach to real-time synchronization*, IEEE Transactions on Computers, 39, pp. 1175–1185.

85. N. Shavit, N. Francez (1986), *A new approach to detection of locally indicative stability*, in (L. Kott, ed.) Proceedings of the 13th Colloquium on Automata, Languages and Programming, vol. 226 of Lecture Notes in Computer Science, pp. 344–358, Springer.

86. N. Shavit, D. Touitou (1997), *Software transactional memory*, Distributed Computing, 10, pp. 99–116.

87. D. Skeen (1981), *Nonblocking commit protocols*, in (Y. E. Lien, ed.) Proceedings of the SIGMOD Conference on Management of Data, pp. 133–142, ACM.

88. M. Spuri, G. C. Buttazzo (1996), *Scheduling aperiodic tasks in dynamic priority systems*, Real-Time Systems, 10, pp. 179–210.

89. J. K. Strosnider, J. P. Lehoczky, L. Sha (1995), *The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments*, IEEE Transactions on Computers, 44, pp. 73–91.

90. G. Tarry (1895), *Le problème des labyrinthes*, Nouvelles Annales de Mathématiques, 14, pp. 187–190.

91. G. Tel (1994), *Network orientation*, International Journal of Foundations of Computer Science, 5, pp. 23–57.

92. G. Tel (2000), *Introduction to Distributed Algorithms*, 2nd edition, Cambridge University Press.

93. G. Tel, F. Mattern (1993), *The derivation of distributed termination detection algorithms from garbage collection schemes*, ACM Transactions on Programming Languages and Systems, 15, pp. 1–35.

94. S. Toueg (1980), *An all-pairs shortest-path distributed algorithm*, Technical Report RC-8397, IBM Thomas J. Watson Research Center.

95. Y.-C. Tseng (1995), *Detecting termination by weight-throwing in a faulty distributed system*, Journal of Parallel and Distributed Computing, 25, pp. 7–15.

96. S. C. Vestal (1987), *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*, PhD thesis, University of Washington.

97. P. Watson, I. Watson (1987), *An efficient garbage collection scheme for parallel computer architectures*, in (J. W. de Bakker, A. J. Nijman, P. C. Treleaven, eds.) Proceedings of the 1st Conference on Parallel Architectures and Languages Europe, vol. 259 of Lecture Notes in Computer Science, pp. 432–443, Springer.

98. P.-C. Yew, N.-F. Tzeng, D. H. Lawrie (1987), *Distributing hot-spot addressing in large-scale multiprocessors*, IEEE Transactions on Computers, 36, pp. 388–395.

Index