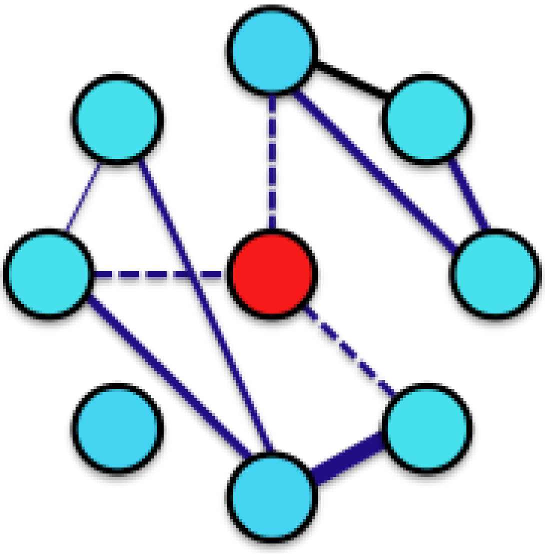
# Distributed Algorithms



**eighth edition**

**D.H.J. Epema**

**Parallel and Distributed Systems Group**

**Faculty of Electrical Engineering, Mathematics, and Computer Science**

**Delft University of Technology**

**the Netherlands**

**January 2017**

## Preface

This is the ninth edition of the lecture notes of the course Distributed Algorithms (IN4150) at Delft University of Technology. In this edition the Paxos consensus algorithm has been added in Chapter 5, and a number of errors has been corrected. Any comments on the contents or presentation are welcome at the email address below.

D.H.J. Epema
d.h.j.epema@tudelft.nl
Delft, the Netherlands
January 2017

# Contents

# Chapter 1

# An Introduction to Distributed Computer Systems and Distributed Algorithms

Distributed computer systems are collections of computer systems that present themselves as single entities to their users. Over the last three decades, such systems have become the rule rather than the exception. Considering the proliferation of networks such as the Internet and distributed systems such as the World-Wide Web, it is probably very difficult to point at two computers anywhere in the world that are not in some way connected and that are not able to communicate. This proliferation has come about thanks to the tremendous progress in the hardware of computer networks—new technologies have been invented, and transmission speeds have increased—in the software managing computer networks—new protocols for data communication and for efficient routing in networks have been developed—and in the software managing distributed systems—high-level protocols for accessing remote data, and middleware enabling multi-component applications have been designed and implemented.

Distributed systems have to be controled by software that enables their components to communicate and to cooperate. Such software executes distributed algorithms that take care of specific control functions in the systems, e.g., for synchronizing or coordinating certain actions occurring in different locations of the systems, or for assuring a certain level of fault tolerance. Distributed algorithms pose many and difficult problems of both a theoretical and a more practical design character. The common cause of all these problems is that in a distributed system, things happen in different locations at the same time, and, as a consequence, no single component of the system has up-to-date knowledge about everything that is going on in the system. In this chapter, we give a general introduction to distributed computer systems and distributed algorithms.

## 1.1 What are Distributed Systems?

We will start with giving two definitions of Distributed Systems (DSs). These definitions are not very precise, but do convey the basic idea of distributed systems. As our first definition, DSs are systems that are characterized by

- *Autonomy*: The components of a DS have a certain power or authority to make their own decisions. Autonomy can be regarded as the distribution of authority;

- *Cooperation*: The components of a DS are working together towards common goals. Cooperation can be regarded as the distribution of functionality;

- *Communication*: The components of a DS exchange information.

An advantage of this definition is that it applies not only to distributed computer systems, but to many other types of systems as well. For instance, organizations—whether they are companies, government agencies, or simply sports clubs or families—and, in fact, complete societies, are also distributed systems in the sense of the definition above, exhibiting all of their essential problems and pointing the way to possible solutions. For example, large companies usually have a hierarchical structure with many departments geographically spread across different countries. Each of these departments has some autonomy to make decisions about its part of the work, but of course, they have to cooperate with other departments, they have to report to a central board, etc. Each company has its own rules (protocols) for dealing with communications (meetings, memos, etc.) and with cooperation. Of course, when any person or department in a company takes a decision, it has to be communicated to other persons and departments, which takes time and may be subject to errors. Whatever means are used for such a communication, whether it is by traditional paper mail or by fast electronic means, there is always a time window in which not everybody is up to date. To cast this in computer terms, people and departments do not have a common, directly accessible memory in which all information relevant to them is stored. We will often draw on real-world situations for examples of the problems in distributed systems and their solutions.

As our second definition, which is much more concrete and specifically deals with computer systems, a distributed computing system consists of multiple autonomous processors that do not share common memory, but cooperate by sending messages over a communications network.

## Examples of Distributed Systems

We now list some examples of distributed systems.

1. Distributed databases, such as airline reservation systems, library catalogue systems, and the World-Wide Web. Much of the discipline of distributed systems originates from the field of distributed database systems. Apparently, the need for access to geographically distributed data was felt very early on in the information age. We will say more about distributed database systems in Section 1.6.

2. Distributed operating systems with such functions as distributed file systems, and shared use of expensive I/O devices such as high-end printers. We will say more about distributed operating systems in Section 1.7.

3. Many types of distributed applications, such as teleconferencing and collaborative environments. Although some operating systems include support for distributed applications, such applications may need functionality that is not offered by these operating systems. Therefore, some distributed applications are like DSs to themselves.

## Essential Properties of Distributed Systems

As an addition to our two definitions above, we list here five properties we assume DSs to have:

1. There is **no regular structure** such as identical processors or a homogeneous network interconnecting the processors with all network links using the same technology or having equal speeds. A DS may be made up of processors of widely different types interconnected by networks of different technologies. Of course, in order to enable processors to communicate, they have to use the same protocols.

2. There is **no directly accessible common state** such as shared variables in a shared memory (so we exclude shared-memory multiprocessors). However, system functions and distributed applications usually need a **logical common state**, which has to be maintained through the exchange of messages.

   As a real-world analogy, the population of a country does not have complete instantaneous knowledge of everything that goes on in the country—only when people read newspapers or watch news broadcasts on television do they get an update on what has happened, and then only in a very limited form. In order to maintain a logical common state, family members and friends make telephone calls, send and read paper-mail and e-mail messages, etc.

3. There is no **common clock**, i.e., a device that is directly accessible by all processors that indicates (real or logical) time. Such a clock would be a part of the common global state, which is supposed not to be present according to property 2. The components of a DS may maintain a notion of a common clock through the exchange of messages, with all of its inherent synchronization problems. If a DS succeeds in doing so, it is said to be *synchronous*, otherwise *asynchronous*.

   In the real world, an exact common global clock is an illusion. Even if a group of people hear somebody say what time it is, or when two computers receive a wireless broadcast message from an atom clock, there is still some inaccuracy caused by the variation in the transmission times of the waves carrying the message, and in the amount of processing time in human or computer brains.

4. There is **nondeterminism** in that different components of the system make progress independently. This means that it is unpredictable in what order operations in different parts of the system are executed, which makes it very difficult to repeat the execution of an application in exactly the same way.

5. There are **independent failure modes**. Non-distributed, single-processor systems are either up and running or down (of course, a component different from the processor may fail, but in computer science we usually think this way). In a DS, some components may fail while others continue running.

## 1.2 Motivations and Requirements for Distributed Systems

In this section we will trace the reasons why DSs have come into existence, and the properties designers and users want such systems to have.

### Motivations for Distributed Systems

The huge investments in research, development and deployment of distributed systems show that there are strong driving forces for such systems. Below we mention what we feel are among the most important ones.

1. **Organizational and application reasons**: People and data are distributed in a natural way, and concentrating all data and applications in a single location or computer system is unnatural, even if questions about the feasibility of such an approach could be solved. It may be much more effective and efficient when the applications and the data that people need are located

close to them. Ideally, the structure of the computer systems of an organization reflects the structure of the organization itself, and perhaps even the way the organization interacts with other organizations. Similarly, an application such as a teleconferencing system dictates a split up across different locations.

2. **Resource sharing**: Resources, whether they are physical such as printers or logical such as information stored in files, have to be accessible to users and application components in different locations.

3. **Extensibility**: Single computer systems, to whatever extent they can be equipped with additional processors, memory, etc, have hard limits to their sizes. Ideally, distributed systems do not have such limits, or they are at least much more flexible.

4. **Performance**: Ideally, DSs should give better performance than centralized systems.

5. **Availability**: When in a distributed system multiple components can perform the same function—that is, when *redundancy* is introduced—it should be possible to continue using the system when one such components fails, albeit perhaps at a lower performance level.

6. **Reliability**: Reliability is a combination of correctness and timeliness—a system (component) is said to be reliable if it produces correct results, and does so in a timely fashion. As DSs usually contain multiple components that can perform certain functions, they should offer an increased reliability.

7. **Security**: With the advent of such applications as e-commerce, and the growth of databases with all kinds of personal information, security is of paramount importance in distributed systems. On the one hand, the manifold interconnections in for instance the Internet may in principle enable access to systems storing classified information from many locations, jeopardizing security. On the other hand, rather than being forced to store many different kinds of information on a single central system, distributed systems offer the option to store smaller amounts of such information on separate, well-shielded subsystems.

Note that the motivations listed above are mainly management and usability aspects of computer systems, not functional aspects. That is, DSs in general do not add functionality, but add to the quality and ease of use.

## Requirements for Distributed Systems

There are some general requirements for distributed systems in order for them to be truly useful.

1. **Transparency**: DSs have to present themselves to their users and the applications running on them as single entities without revealing that they are in fact built from many components. Many different forms of transparency have been identified in the literature. We discuss the following:

   (a) *Location transparency*: It is not necessary to know (for either a user or an application) the location of an entity in order to request an operation on it to be performed. In some early UNIX-based distributed systems, the file system was organized by adding a new root and letting the roots of the file systems of the component machines be subdirectories in this new root with the names of these machines as their names. So full path names of files consisted of first a machine name and then the full local path name. Clearly, such a solution is highly non-transparent.

(b) *Access transparency*: Wherever entities are located, they can be accessed (again by users and applications alike) with the same means. E.g., wherever a file resides, it can be accessed with the same editor (at the user level) and with the same system calls (at the application level).

(c) *Migration transparency*: When an entity is moved from one host to another, neither users nor applications have to be modified. For instance, in a distributed file system, a set of files may be moved from one file server to another in order to balance the load. Location and access transparency are prerequisites for migration transparency.

(d) *Replication transparency*: A user or application does not have to be concerned with which copy of a replicated entity is used. Again, this requires location and access transparency.

(e) *Failure transparency*: It cannot be noticed—except perhaps for degraded performance—when a component fails, and so, neither users nor applications have to take measures to counter failures. This form of transparency is supported by replication transparency.

2. **Scalability**: DSs should be extensible without bottlenecks, and the increase of the capacity of a system should be proportional to the extension. So the architecture should not contain central components that may come to constitute a performance bottleneck under increased load, or a single point of failure.

   For instance, bus-based multiprocessors are not scalable because the bus is a central component that will get congested when too many processors are connected to it. An interconnection structure with better scalability is a two-dimensional grid. Similarly, an ethernet (segment) becomes a bottleneck if too many computers are connected to it.

3. **Consistency**: DSs should be consistent in terms of their performance, user interface, and the global view they offer (e.g., when replication is employed).

4. **Modularity and Openness**: In DSs, there is a strong emphasis on the design and standardization of interfaces, which is in particular helpful when systems are, or evolve to be, heterogeneous.

## 1.3 Distributed Algorithms

Distributed Algorithms (DAs) are algorithms that consist of different components that run on different processors and that communicate through messages. We distinguish two types of DAs:

1. Algorithms that *compute* something, that is, applications which are submitted to a DS to do some work perceived as useful by a user of the system. For instance, algorithms for linear-algebra computations may be partitioned across multiple processors, amongst which partial results are exchanged.

2. *Control* algorithms or *system* algorithms that execute some function to ensure the proper operation of some aspect of a DS. For instance, a typical control problem in DSs that has to be solved by DAs is mutual exclusion, which in a DS can be formulated in exactly the same way as in sequential systems. Such algorithms are sometimes called *protocols*.

In this book, we will only be concerned with algorithms of the latter type. The general structure of DAs will be treated in Sections 2.2.3 and 2.2.4.

A control problem that is specific for DSs is that of determining the global state of a distributed computation, which we explain in the example below.

**Example 1.1** An example of a problem in DSs that has to be solved by means of a DA is the *determination of a global state* of a computation in a DS. In a sequential system, determining the global state of a computation is conceptually easy by simply recording the data segment, the stack, and the processor registers, including the program counter of a running program. In fact, this is what happens when a core dump of a program is made when an error occurs. One of the purposes of determining global states is debugging. Determining the global state of a distributed computation is more complicated because one would like to record the states of the different parts of the computation at the same time, and because at that time messages may be in transit. The first is a synchronization problem, the second a problem of determining the states of communication channels. In Chapter 3 we will deal with DAs for this problem.

As a real-world analogy of this problem, in some countries the income-tax agency tries to do just this, determining global states. All tax payers have to enter the amounts in all their bank accounts on January 1 at 00:00 hours on their tax forms. Of course, the agency does not want tax payers to have money in transit between their accounts at that time. □

## 1.4 Parallel Computing versus Distributed Computing

The distinction between parallel computing and distributed computing is not clear-cut. Still, we have different connotations with these terms. A characteristic common to parallel and distributed computing is of course that a single application is split up into different components that each run on a different processor. In parallel computing,

- The granularity of splitting up an application is usually rather small;
- The frequency of communication between the components is usually high, e.g., on the order of at most milliseconds;
- Applications are usually split up in a homogeneous fashion, with each of the components executing more or less the same function on different parts of the data. In contrast to this so-called data parallelism, a form called task parallelism is distinguished, in which the tasks do have different functions. This occurs for instance when an application is structured as a pipeline, with different stages performing different functions;
- The components of an application usually have to run simultaneously in order for the application to run efficiently;
- Applications are usually executed on "parallel" hardware, that is, on systems with identical processors and with a regular interconnection structure (e.g., bus-based shared-memory multiprocessors, or processors connected by a two-dimensional grid).

Examples of parallel computing can be found in linear algebra.

**Example 1.2** In Cannon's algorithm, two $n \times n$-matrices $A = (a_{ij})$ and $B = (b_{ij})$, $i, j = 0, \ldots, n-1$, are multiplied in a two-dimensional grid of processors with wrap-around connections of size $p \times p$, with $n/p$ an integer. To describe the algorithm, first suppose that $p = n$. Assume that the matrices are distributed across the grid such that processor $(i, j)$ contains matrix elements $a_{ij}$ and $b_{ij}$. In addition, processor $(i, j)$ contains a variable $c_{ij}$, which is initialized to zero. In the first step, the matrices are redistributed across the system such that each processor contains elements of the matrices that have to be multiplied. In particular, row $i$ of matrix $A$ is shifted $i - 1$ processors to the left, and column $j$ of matrix $B$ is shifted $j - 1$ steps up, $i, j = 2, \ldots, n$. Now processor $(i, j)$ contains elements $a_{i,j+i-1}$ and $b_{i+j-1,j}$, which indeed have to be multiplied; the product is assigned to $c_{ij}$. In each of the next $n - 1$ steps, the complete matrix $A$ is shifted one step to the left and the complete matrix $B$ is shifted one step up, the elements of $A$ and $B$ then received by each of the processors are multiplied, and the result is added to the partial sum $c_{ij}$. The final result is that processor $(i, j)$ contains the $(i, j)$-th element of the product matrix in $c_{ij}$. When $n > p$, this algorithm can be applied to the sub-matrices of $A$ and $B$ of size $n/p \times n/p$. Clearly, running this algorithm would be very inefficient when not all processors are allowed to run the algoritm at the same time, when the processors employed would have widely different clock rates, or when the cummunication network would not have low latency. □

In distributed computing,

- The granularity of splitting up applications is usually large;
- The communication between the components of the application may be infrequent, e.g., on the order of minutes or hours;
- Applications are split up in an inhomogeneous fashion, with different components serving different functions;
- The components of an application do not have to run simultaneously, but only need to synchronize or communicate once in a while;
- Applications are executed on "distributed" hardware, that is, on systems with processors of different types, which may be interconnected with different network technologies.

**Example 1.3** An example of a distributed application is a distributed file system. The file servers making up such systems may be of varying sizes, and they have to communicate with each other when a file stored at one location is accessed from another location. Different file servers may have different functionalities, with some being responsible for keeping replicas at different locations consistent, and with others performing periodic back-ups, etc. The time scales for these operations may vary from minutes to days. □

## 1.5 Computer Networks and Distributed Systems

Distributed systems rely on computer networks for their operation. Although the seven-layer OSI-model is not in universal use, it provides a good way to explain the relation between distributed systems and networks. Therefore, we first briefly describe the functions of the lowest four layers in this model; layers 5, 6, and 7 (the session layer, the presentation layer, and the application layer, respectively), are not relevant for our discussion.

1. The physical layer provides the physical means for the transportation of bits in the form of the hardware of the network links and the way in which bits are (de-)modulated in order to be transmitted.

2. The datalink layer provides the illusion of fault-free links across which data can be sent by dealing with errors in the transmission through such techniques as error-detecting and error-correcting codes.

3. The network layer provides such functions as routing and congestion control, and is the highest layer that is present not only in the end points of a communication path between processors, but also in all intermediate computers (and routers) along the way.

4. The transport layer provides end-to-end connections between processes, and is the lowest layer that only exists in these end points.

In DSs, we are concerned with layers 2, 3 and 4. When we discuss problems in DSs in terms of any set of processes that somehow communicate, we base ourselves on the transport layer. However, when we discuss DAs for specific interconnection structures—such as election in unidirectional rings in Chapter 4 and consensus protocols in complete networks in Chapter 5—we base ourselves on the network layer. And when we want distributed algorithms to be resilient to link failures, we in fact assume that layer 2 does not function properly.

## 1.6 Distributed Database Systems

Distributed database systems were among the first distributed systems to be designed and implemented. Apparently, the need to be able to access information remotely was felt early on. Examples of such databases include financial databases with information about accounts of customers, and airline-reservation systems. In such systems, information is stored in some structured way such as records with some number of fields. These records are accessed through sets of queries and updates that specify the values of certain fields called *transactions*. A transaction should be executed in an *atomic* way, with either all or none of its results becoming permanent; in addition, its partial results should be invisible to other transactions. This introduces the problem of *concurrency control*: having multiple transactions in operation simultaneously but avoiding conflicts when transactions try to access the same records, basically forcing the system to operate as if transactions are executed serially. A popular mechanism for concurrency control is to use *locks* that govern the access to database records. In effect, concurrency control amounts to a complicated version of mutual exclusion, with dynamic access to sets of records.

## 1.7 Distributed Operating Systems

The operating system of a computer system has two main functions. First, it has to offer a virtual machine to the users and to the applications running on the computer system in the form of a command interface and a system-call interface, respectively. Second, it has to act as a manager of the resources in the computer system. Here we are only interested in the second function. We will now state some of the special features of distributed operating systems as managers of a computer system's resources, which are usually categorized into the following six groups.

1. Processes. Operating systems have to create, maintain, and terminate processes. There may be circumstances when a process has to be migrated, that is, moved, from one processor to another, for instance, when there is a mismatch in the processor loads. Migrating a process not only involves moving its address space and processor context, but also moving queues of pending messages to its new location, closing open files and re-opening them in the new location, etc. A general process migration mechanism is very complex [41].

2. Communication. Distributed operating systems have to offer message-passing facilities. Often, communication at the application level has a more structured character, such as Remote Procedure Calls (RPCs), with a process on one machine requesting a process on another machine to execute a procedure on its behalf, and Remote Method Invocation (RMI), which is the object-oriented analogue of RPC.

3. Processors. Operating systems have to manage the time on the processors. In addition to allocating time slots to the processes on each of the processors, this involves deciding where to put a new process, and where to migrate a process when some processors are heavily loaded while others are lightly loaded.

4. Memory. Although according to our definition, in a DS every processor has its own memory, a distributed operating system may include such features as Distributed Shared Memory (DSM), which maintains the illusion of a single memory across multiple machines to applications.

5. Data. Distributed file systems offer similar functionality as single-processor file systems. Distributed file systems involve such problems as locating files, distributed naming schemes of files, and the efficieny of read and write operations across a network.

6. I/O. For instance, distributed presentation environments like allow a user to redirect the output of an application from any location to his own computer screen.

Since the mid 1990s, the notion of *middleware* has gained importance. Middleware is a software layer between operating systems and distributed applications that implements all kinds of services in a platform-independent way, such as communication, data storage, transactions, etc. Also since the mid 1990s, *grids* have emerged as multi-organizational collections of clusters and individual machines that act as a utility for compute-intensive and data-intensive applications. A follow-up technology to grids is constituted by *clouds*, which are large-scale computing resources that can be leased in an on-demand fashion from cloud providers. A development that has attracted much interest since 2000 is the emergence of *peer-to-peer* (P2P) systems, which are collections of computer systems in which no system has any authority over another, i.e., in which all systems are truly autonomous; see [42] for an overview.

## 1.8 Important Techniques in Distributed Systems

There has emerged a set of techniques that recur very often as ingredients of solutions to problems in DSs and distributed applications. We list some of these techniques below, along with some examples and real-world analogies.

1. **Replication**, that is, maintaining multiple copies of the same entity, is a very widely used technique for improving the performance and the availability of DSs. For instance, having many copies of some often-used file may decrease the average access time to the data stored in the file. Replication introduces the obvious problem of maintaining consistency, that is, of guaranteeing the replicas to have equal contents, when modifications to the replicated object are made.

   Replication abounds in the real world: Newspapers are printed in large numbers, paper telephone directories are everywhere, etc. These two real-world examples exhibit different ways of dealing with consistency and updates. Copies of newspapers are written only once, and no attempt is made at updating them. Although next day's newspaper is in a sense an update, it is regarded as a completely new thing, so there are no consistency issues. Paper telephone directories are issued in a series of versions, and even the latest version is usually not completely up to date. Then, or when only an older version is at hand, we may consult a human operator or an electronic directory. Even then, there is a time window between changing or adding a telephone number and entering it into the directory.

2. **Caching** is a special form of replication of data in which usually only a fraction of a data set is copied, and possibly modified and integrated back into the complete copies. Caching is for instance used in file systems, and when looking up the IP-number of a computer in a name server. People usually keep a cache of telephone numbers of family members, friends, etc.

3. Using **locality**, that is, first trying to use some entity that is close in some metric rather than one that is remote, is an important technique in DSs. For instance, when accessing a file, it is more efficient to use a copy of it that resides locally instead of accessing a copy on a remote file server. More generally, in DSs there are many tradeoffs to be made in deciding where to perform an operation on some entity. When a simple operation has to be performed on a large data set, it may be more worthwhile to transfer a description of the operation to a location where the data reside, perform the operation, and transfer the result back. This is what is known as *function shipping*. When the data set is relatively small and the operation complex, *data shipping* may be preferred.

4. **Timestamps** are assigned to all kinds of events in computer systems, such as modifying a record in a database, and scheduling an event in a real-time system. In DSs, there is the additional difficulty of synchronizing computers in order to be able to compare timestamps assigned by these systems. One use of timestamps is to check whether some copy of data is still valid or has gone stale.

   A concern when using timestamps is that in principle, they can be unbounded, which is not practical, as they have to be implemented as counters in a finite number of bits. However, this concern is somewhat exaggerated, as the following computation shows. Suppose that timestamps are implemented in 64-bit registers, so the maximal value is $2^{64} - 1 \approx 16 \cdot 10^{18}$. If the counter is incremented every microsecond (which for most applications seems very frequent), then because the number of seconds in a year is approximately equal to $\pi \cdot 10^7$, the counter only overflows after 500,000 years!

5. **Time outs**—requesting some service and when some amount of time has expired taking some action—are widely used in distributed systems. For instance, when a process tries to communicate with another process and does not receive a reply within some amount of time, it may try again, or it may try somewhere else. Similarly, when a transaction in a distributed database

| decade | 1960s | 1970s | 1980s | 1990s |
|---|---|---|---|---|
| keyword | batch | time sharing | desktop | network |
| technology | MSI | LSI | VLSI | ULSI |
| location | computer room | terminal room | desk | mobile |
| users | experts | specialists | individuals | groups |
| data | alphanumeric | text, vector | fonts, graphs | speech, audio |
| goal | computation | access | presentation | communication |
| activity | punch & try | remember & type | see & point | ask & tell |
| operation | process | edit | layout | orchestrate |
| connect | peripherals | terminals | desktops | palmtops |
| languages | Cobol, Fortran | PL/1, Basic | Pascal, C | C++, Java |

Table 1.1: Characterization of computing in the last four decades of the twentieth century (adapted from IEEE ...).

system has been holding a lock for a record for some amount of time and another transaction requests the same lock, it may have to release it. Of course, time outs abound in real life.

6. **Randomization** in algorithms in general is the technique of having processes flip a coin once in a while, usually to achieve a result faster. Randomized algorithms stand in contrast to deterministic algorithms, in which the actions of the processes are exactly determined by their states. In fact, randomization can be used in distributed algorithms in two ways. In the first, processes flip coins in order to achieve better results, or to achieve results faster, usually in a probabilistic sense as randomization leads to non-guaranteed behavior. In the second way, randomization is used to achieve results in cases which it can be proved that no deterministic algorithms exist. We will come across such *impossibility results* in Chapter 5.

## 1.9 Historical Perspective

In this section we give a rough classification of the character of computing in the first seven decades of the existence of computers, that is, in the period 1950–2020 (as far as the current decade can already be characterized). In Table 1.1, a nice overview of many different aspects of computers and their use in the last four decades of the 20-th century is given.

- The 1950s were characterized by huge stand-alone computers, which were difficult to program. People who wanted to use such a thing had to make a reservation. At the appointed hour, they walked into the computer room, and were all by themselves to get and keep the machine going. At best, there was a primitive run-time system, a compiler, and a set of I/O routines.

- In the 1960s, batch-oriented mainframes prevailed. Users could offer jobs to such machines through some form of a job-entry system and card readers! The turn-around times of these jobs were hours, if not days.

- In the 1970s, interactive computing or time sharing became possible. Mainframe computers, and later mini-computers, got equipped with time-sharing operating systems which kept up the illusion for every user that he had the machine to himself.

- In the 1980s, first stand-alone PCs became popular, followed by local-area networks and distributed file systems.

- The 1990s saw the wide acceptance of the Internet as an information and communication infrastructure with the World Wide Web and e-mail, and of mobile systems.

- In the 2000s (the decade 2001-2010, that is!), grids, clouds, datacenters, peer-to-peer systems and online social networks, content-distribution networks, and sensor networks have become popular and feasible.

- In the 2010s, embedded and wearable devices will become commonplace, and the Internet will really become an infrastructure for high-quality, ubiquitous video distribution and communication.

## 1.10 Bibliographic Notes

General books on distributed algorithms include Barbosa [4], the comprehensive book by Lynch [38], which makes important distinctions between synchronous and asynchronous systems, and between shared-memory and message-passing systems, the book by Attiya and Welch [2], which is more theoretical, the book by Reisig [48], in which Petri nets are used as the modeling technique, and the book by Tel [61]. More recent books include the book by Kshemkalyani and Singhal [31], and the book by Raynal [47]. A very accessible book is by Fokkink [22], which presents a light-weight introduction to distributed algorithms. An interesting lecture collection on distributed computing is maintained by Wattenhofer [1].

Two general books on distributed systems are Coulouris et al. [15] and Tanenbaum and van Steen [60]. The book by Chow and Johnson [14] deals with both distributed systems and distributed algorithms. A rather theoretical and now somewhat outdated overview of the field of distributed systems and algorithms is given by Lamport and Lynch [33]. The book by Jalote [29] deals specifically with fault tolerance in distributed systems. An introduction to reliable distributed computing, including topics such as reliable broadcast and consensus is presented in the book by Guerraoui and Rodrigues [25].

The journals *ACM Transactions on Computer Systems*, *ACM Transactions on Database Systems*, *ACM Transactions on Programming Languages and Systems*, *ACM Transactions on Internet Technology*, *ACM Transactions on the Web*, and *IEEE Transactions on Parallel and Distributed Systems* publish rather practically oriented papers on all aspects of distributed systems, while the *Journal of the ACM* and *Distributed Computing* do the same on a much more theoretical level. The following three conferences and workshops on distributed systems and algorithms are held annually:

1. The series of *Workshops on Distributed Algorithms* (WDAG) held between 1985 and 1997, renamed the *International Symposium on Distributed Computing* (DISC) [2] in 1998 and held annually ever since, focuses on distributed algorithms.

2. The *ACM Symposium on Principles of Distributed Computing* (PODC) [3] is a rather theoretically oriented conference on both distributed systems and distributed algorithms.

[1] see `disco.ethz.ch/lectures/podc_allstars`.

[2] see `http://www.disc-conference.org`.

[3] see `http://www.podc.org`.

3. The *International Conference on Distributed Computer Systems* (ICDCS) organized by the IEEE is a general conference on distributed systems.

# Chapter 2

# Modeling Distributed Systems

In order to discuss and reason about distributed systems and algorithms, we need models of such systems, which clearly have to include a notion of cooperating processes communicating through messages. As we will see, one of the most important ingredients of models of DSs concerns the timing in such systems. In asynchronous systems, there are no assumptions about the transfer times of messages other than that these times are finite. In synchronous systems, it is assumed that there is a bound on the message transfer times, known to all processes.

For presenting DAs, in this book we (loosely) define constructs for imperative programming with message passing, which are based on a simple operational model of DSs.

## 2.1 Networks, Processors, and Processes

In our basic model, a distributed system consists of a finite set of processors, a network consisting of a finite set of network links interconnecting these processors, and a set of processes running on the processors that cooperate on some application. Usually, in the problems we will consider, there is only one process running on each processor, and then we will identify processes and processors. We will now first discuss the network structure.

### 2.1.1 The Interconnection Structure

In our model, we assume that network links (or channels) are unidirectional. When two-way communication is possible between two processors, this is modeled by two unidirectional links. We will always assume that the network is connected, that is, that every processor (process) can be reached from every other processor, either directly or indirectly. In this book, we will only use three special interconnection structures, viz. complete networks, rings, and two-dimensional grids. In a *complete network* (also called a *clique*), there is a link from every processor to every other processor. In a *ring*, every processor is connected to two other processors. In a *unidirectional ring*, every processor can only send messages to one of its neighbors (called its *downstream neighbor*) and receive messages from its other neighbor (called its *upstream neighbor*); the downstream and upstream directions of all processors are compatible, respectively. In bidirectional rings, every processor can send to and receive from both its neighbors. In a two-dimensional grid, processors are arranged in a rectangle, with processors internal to the rectangle having four connections (up, down, left, right), processors at the edges having three connections, and the four processors at the corners having two connections. If there are also *wrap-around* connections, every processor has exactly four connections.

### 2.1.2 Synchronous and Asynchronous Systems

Time plays an important role in computer systems in general, and in DSs in particular. In DSs, (at least) the following two forms of synchrony are distinguished:

1. Processors are synchronous when the ratios of their speeds are bounded;

2. Message passing is synchronous when the message delays are bounded.

A DS is said to be *synchronous* when all of its processes have access to a global common clock, which is some magic device that all processors can access instantaneously, and when both processors and message passing are synchronous. A DS is *asynchronous* when there is no global common clock, and either processors or message passing, or both, are asynchronous.

Perhaps the most important difficulty in asynchronous DAs in possibly faulty systems is that when a process expects a message from another process, however long it has already been waiting, it can never be sure as to whether the message is still going to be received or that the process sending it or the link supposed to transfer it has failed.

### 2.1.3 Synchronous and Asynchronous Communication

In asynchronous DSs, the communication between processes can be modeled as synchronous or as asynchronous communication.

In systems with *asynchronous communication*, the events of sending a message and receiving the same message are truly separate events, with the latter occurring after the former. Then, the sending process does not have to be blocked for the receiving process to be ready to receive the message. The receiving process may then still block waiting for the sending process to be ready to send the message, or it may be interrupted when a message becomes available.

In systems with *synchronous communication*, the two events of sending a message and receiving it occur (logically) simultaneously. The send and receive events together can then be regarded as a single *message transfer* event. Sending a message is now blocking, as the sending process has to wait for the receiving process to be ready for the reception of a message. In fact, in systems with synchronous communication, we can do away with channels altogether, as messages sent are received immediately, and so, (logically) there are never messages in transit.

Systems with synchronous communication can be simulated by systems with asynchronous communication, and vice versa. In order to simulate synchronous communication with asynchronous communication, an explicit acknowledgment scheme can be employed, with the sending process blocking until it has received an acknowledgment from the receiving process. For the simulation the other way around, for each communication link a special process managing a buffer of messages can be introduced to decouple the sender from the receiver. When a sender sends a message, the buffer process receives it immediately and enters it into a buffer. After a certain amount of time, the buffer process retrives the message from the buffer and sends it to the receiver, thus introducing an artificial delay in the message transfer. It should be noted, however, that DAs working correctly in a system with asynchronous communication may exhibit deadlocks when executed in the simulated synchronous system. For more on this subject, see [13].

### 2.1.4 Configurations and States, Transitions and Events

At any point in time, a process in a DS is in a certain *state*, which is defined as the set of values of all its relevant variables. Among the set of possible states of a process we assume that there is a set of *initial states* and a set of *terminal states*. Similarly, every channel has a state, which is the set of messages sent along the channel but not yet received. We assume that the initial state of a channel is the empty set. When a distributed algorithm starts, all processes and channels are in one of their initial states, and the algorithm terminates when all processes are in one of their terminal states. A *configuration* or *global state* of a distributed system is made up of the joint (*local*) *states* of all its processes and channels.

A state change of a DS is called a *transition*. Transitions are caused by *events* in one or more processes. We distinguish the following three types of events:

1. *Internal* events. An internal event in a process only causes its own local state to be modified;

2. *Message send* events. A message send event in a process modifies the state of one of its outgoing channels by adding a message to it;

3. *Message receive* events. A message receive event in a process modifies the state of one of its incoming channels by removing one of the messages from the state of the channel, and may cause a modification of the state of the process.

An *execution* of an asynchronous DS is a finite or infinite sequence of events that starts with all processes in one of their initial states and all channels empty. Clearly, in such a sequence, the order of many pairs of events cannot be interchanged, as they may be related. For instance, the order of a message send event and the corresponding receive event is fixed. In addition, the events in a single process do occur in a particular order which cannot be changed. However, for many pairs of events in an execution, the order can be changed. For instance, two internal events in two different processes that appear next to each other in such a sequence may very well be interchanged. So the order in a sequence of events which constitutes an execution, is only a *partial order*. In Section 3.1 we will further explore these notions.

### 2.1.5 Properties of Network Links

Usually, DAs impose conditions on the communication links in order to function properly. Some of these conditions are:

1. No loss of messages: messages sent are guaranteed to be received;

2. No damage of messages: messages received are guaranteed to be correct;

3. The FIFO property: the messages sent along a single channel are received in the order sent. When a channel has the FIFO property, the set of messages sent along a channel but not yet received, is actually a sequence.

4. Bounded or unbounded buffers: the number of messages sent but not yet received along a link is or is not bounded;

5. Finite delay: messages sent along a link (that are not lost) are always received within a finite (but possibly unbounded) amount of time;

6. Bounded delay: there exists some upper bound on the delay experienced by messages that are not lost.

Note that in systems with synchronous communication, only the first two properties are relevant. In systems with asynchronous communication, all the above properties are relevant. Unless stated otherwise, we will assume that communication links are fault-free (no loss or damage of messages), and in the case of asynchronous communication, that they have unbounded buffers and have finite delays.

## 2.2 Distributed Algorithms

In this section we discuss our way of presenting DAs, both in terms of the different parts in their descriptions and of the conventions in the pseudo-code we use.

### 2.2.1 Presentation of Distributed Algorithms

In this book, we will present DAs in the following four parts:

1. **Idea:** The first part presents the main idea(s) of the algorithm, which is an informal description and explanation of the algorithm.

2. **Implementation:** The second part provides an implementation of the algorithm in imperative pseudocode.

3. **Correctness:** The third part is a correctness proof.

4. **Complexity:** The final part is a discussion of the algorithm's performance or complexity.

When any of these parts is omitted, it is left as an exercise for the reader.

### 2.2.2 Programming Conventions

In this book we will use a rather loosely defined imperative language for presenting algorithms in pseudocode:

1. Unless specified otherwise, all processes have the same implementation. When a reference to a process id is needed, it is indicated with the index $i$. The code is always written from the perspective of a general process $P_i$.

2. We do not explicitly declare the local variables of the processes; rather, they are introduced by their first use. Usually, all processes have the same variables, which we simply refer to by their names without explicitly stating the process to which they belong. When confusion can arise, for instance, when reasoning about an algorithm, we use process ids as subscripts to distinguish the different copies of the same variable in different processes.

3. The assignment of a value v to a variable a is written as

   ```
   a ← v
   ```

4. We specify messages as containing a message type and a comma-separated list of fields, like this:

   ```
   (message type; message fields)
   ```

   If the message type is clear from the context, we omit it. We assume that the message fields can be accessed as read-only local variables in the receiving process. When a new message of the same type is received, the fields of the previous message are lost. When these fields are needed after the message has been received, they have to be stored into local variables. Often, the process id of the sending process is either obvious (e.g., in a unidirectional ring) or not important. If this id is needed by the receiving process, it is explicitly included as one of the message fields.

5. Sending (receiving) a message `m` to (from) some other process is expressed as

   ```
   send(m) to $P_j$
   receive(m) from $P_j$
   ```

6. Broadcasting a message `m` to all processes in the system is expressed as:

   ```
   broadcast(m)
   ```

   We adopt the convention that a broadcast message is also sent to the sending process.

7. In some algorithms we use an ordered *queue* with elements from a totally ordered domain, such as pairs `(T,i)` with `T` a timestamp and `i` a process id, which can be ordered lexicographically. The order in the queue is such that the smallest element is at the head. If `Q` is an ordered queue of variables of some type, and `e` is a variable of the same type, then with the statement

   ```
   enqueue(Q,e)
   ```

   we enter `e` into the queue in its proper location according to the order. For a non-empty queue `Q`, with the statements

   ```
   e ← head(Q)
   dequeue(Q)
   Q′ ← tail(Q)
   ```

   we assign to `e` the element at the head of `Q` (without removing it), we remove the element at the head of `Q`, and we assign to `Q′` the remainder of `Q` after having removed the element at the head, respectively.

### 2.2.3 Distributed Algorithms in Synchronous Systems

A DA in a synchronous DS proceeds in *rounds* consisting of first receiving any messages sent to it in the previous round, then doing some internal computations, and finally sending a finite number of messages. The overall structure of DAs in synchronous systems is:

```
do some number of times
   receive all pending messages
   perform local computations
   send messages
```

One execution of this loop is a round, and for every round, the common clock increases by 1. The number of times the loop is to be executed may not be known ahead of time, but can depend on the messages received. At any time, all processes are always in the same round.

### 2.2.4 Distributed Algorithms in Asynchronous Systems

In asynchronous systems, DAs typically take the form of different pieces of (pseudo)code each of which describes the corresponding actions to be performed when a message is received or when a condition is or becomes true:

I. Pseudocode for message reception

```
upon receipt of (message type; message fields) do
    actions
```

or

II. Pseudocode when a condition is or turns true

```
when (condition) do
    actions
```

We adopt the convention that after the first moment that a condition is true, the process may wait for an unbounded but finite amount of time before executing the actions. For instance, in some algorithms we will encounter, the condition is `true` and the `actions` consist of initiating the algorithm.

The different pieces of code of an asynchronous DA may access the same local variables. If one piece can be interrupted by another, some form of mutual exclusion may be called for. To exclude as much as possible this added complexity, we will adopt the convention that the pieces of code are executed atomically, except when the actions contain a blocking primitive or a statement that implies a delay, such as a critical section.

### 2.2.5 Properties of Distributed Algorithms

Below are some properties of distributed algorithms.

1. The *symmetry* of the algorithm. In general, a solution is considered to be more elegant (and distributed) when the component processes are (more or less) identical. Different forms of symmetry are distinguished. An algorithm is said to be symmetric by state if **to be added** and symmetric by id if the algorithm texts are completely identical, or the algorithm texts are completely identical except for a reference to the processes' ids.

2. An algorithm is said to be *uniform* when the processes do not know the number of processes in the system.

3. We say that a distributed system made up of nodes and connections is *anonymous* when the nodes do not have ids. Even when there are ids, they may not be unique. In actual systems, processors may have unique processor ids stored in ROM.

4. A DA is said to be *randomized* if it includes drawing a random number from some distribution to decide on some of its actions, and *deterministic* otherwise. As we will see, the concept of randomization is very powerful: it may reduce considerably the (expected) performance of DAs, and it makes solutions to problems in asynchronous DSs possible that do not have deterministic solutions.

5. The types of networks for which the algorithm is suitable, such as the level of connectivity.

### 2.2.6 The Performance of Distributed Algorithms

An important aspect of a distributed algorithm is its performance, expressed in some performance metric. These metrics fall into two categories, viz. time and space metrics. In synchronous systems, the time metric we will consider is the number of rounds in a deterministic algorithm or the expected number of rounds in a randomized algorithm to run the algorithm to completion. In asynchronous systems, the time metric of interest is the length of the longest chain of messages in any execution of the algorithm. Here, a chain of messages is a sequence of messages with the sender of every message equal to the reveiver of the previous message, of which no message can be sent before the previous message has been received.

The only space metric we will consider—for any type of algorithm—is the total number of messages sent.

For every metric one can study its worst-case or its average behavior. We sometimes only determine the expected values of these metrics by either averaging over all runs for all possible inputs, or over all possible runs if there is a random element in the algorithm.

## 2.3 Simulations

There exist many different models of DSs, and it may be easier to design a DA for some problem for a certain type of system than for another. For instance, a DA for a synchronous system may be more straightforward than for an asynchronous system. In such a case, rather than design a DA for every model, one may try to design a single DA for an "easy" model, and try to have more complicated (less restricted) models behave like the easy model by means of *simulations*, i.e., by means of a software layer running on top of the "difficult" model.

Suppose we have a distributed algorithm $A$ for some problem running on top of model $M$, and we would like to have a distributed algorithm for the same problem for some model $M'$, which is more difficult to deal with. Then a *simulation* consists of a DA (protocol) $P$ running on $M'$, such that the pair $(M', P)$ behaves as $M$, and so that $A$ can be run on top of $P$.

In real networks and DSs this happens all the time. As an example, in the OSI model of networks (see Section 1.5), $M'$ can be the bare physical layer, $P$ the software of the datalink layer, and $A$ can be the network layer running on top of the datalink layer.

In the field of DAs, a distinction is made between *local simulations* and *global simulations*. In a local simulation, every single process(or) running $A$ cannot distinguish $M$ and $(M', P)$, but an outside observer can see the difference. In a global simulation, even an outside observer cannot distinguish the two systems. In Section 3.2, we will see a local simulation of a synchronous system on top of an asynchronous system. There, from the perspective of every process, the system operates in rounds, but an outside observer may find that different processes are in different rounds at the same time.

## 2.4 Bibliographic Notes

In [13], the implications of synchronous and asynchronous communication in distributed systems are treated in great detail.

# Chapter 3

# Synchronization

Time plays an important role in computer systems. Both for their internal operation and for the applications they run, computers need to be able to keep track of time, to affix timestamps to events, and to compare timestamps. For instance, real-time systems have to set timers all the time to schedule future events such as initiating tasks. In this case, computers need a notion of real time, with a granularity on the order of milliseconds or even microseconds. In electronic banking applications, computers need to be able to affix timestamps to money transfers. Here again, the computers need a notion of real time, but the granularity can be on the order of seconds or minutes. Finally, file systems usually keep track of the times of the last modifications to files. Here, for users it is very important that real time is used, but for the purpose of making incremental backups, some notion of logical time—to determine whether a file has been modified after the previous backup or not—may suffice. Of course, when in each of these examples the computer system is distributed, some form of synchronization among its components is called for. If a network of real-time computers is used in process control, their actions must be synchronized. When performing a money transfer in an electronic banking system, the timestamp assigned to the event of receiving the money should be more or less equal to the timestamp of the event of sending it plus the transmission delay. And when in a distributed file system a file is replicated, that is, multiple copies of the same file are maintained, the modifications to the copies should be performed in the same order.

In this chapter, we first define time concepts in distributed systems in Section 3.1. Then, in Section 3.2 we discuss algorithms for synchronizers, which simulate synchronous systems on top of asynchronous systems. In Section 3.3 we present different ways of ordering the delivery of messages to single processes in a logical order. Section 3.4 deals with the fundamental problem of taking logically consistent snapshots of a distributed system. In Section 3.5 we discuss the problem of finding out whether a distributed program has terminated, which is a non-trivial problem as there may be messages in transit. Finally, in Section 3.6 we show a number of solutions for the problem of distributed deadlock.

## 3.1 Time Concepts in Asynchronous Distributed Systems

In asynchronous distributed systems, the need may arise to reason about events based on their order of occurrence. In a specific situation, an event may be anything from the execution of a machine instruction to the execution of a database transaction. Events within a single process(or) are assumed to have a total order. The difficulty of course lies in ordering events that occur in different processors. When no messages are sent between processors, nothing can nor has to be said about the order of events in different processors, so our definition of ordering events in different processors strongly rests on the exchange of messages.

We assume that we are given an asynchronous system consisting of $n$ processes $P_i, i = 1, \ldots, n$, which we take here to mean that processes do not have access to a common clock, and that messages have arbitrary but finite delays. We distinguish three types of events. First, an event can be completely *internal* to a process. Furthermore, the acts of sending and receiving a message are modeled as (*message*) *send* and (*message*) *receive* events in the sending and receiving process, respectively. Let $E_i$ be the set of events in $P_i$, which we assume to be totally ordered, and let $E = \cup_i E_i$ be the total set of all events in the system.

### 3.1.1 The Happened-Before Relation

The basis of much of the theory of ordering events in distributed systems is the *happened-before relation* [32], which is at the basis of much of the concepts of time in distributed systems.

**Definition 3.1** *The* **happened-before** *(HB) or* **precedes** *relation $\rightarrow$ on $E$ is the smallest relation satisfying*

1. *(Local order) If* $\mathtt{a}, \mathtt{b} \in E_i$ *for some $i$ and* $\mathtt{a}$ *occurred in $P_i$ before* $\mathtt{b}$*, then* $\mathtt{a} \rightarrow \mathtt{b}$*;*
2. *(Message exchange) If* $\mathtt{a} \in E_i$ *is the event in $P_i$ of sending message* $\mathtt{m}$ *and* $\mathtt{b} \in E_j$ *is the event in $P_j$ of receiving message* $\mathtt{m}$ *for some $i, j$ with $i \neq j$, then* $\mathtt{a} \rightarrow \mathtt{b}$*;*
3. *(Transitivity) If for* $\mathtt{a}, \mathtt{b}, \mathtt{c} \in E$*,* $\mathtt{a} \rightarrow \mathtt{b}$ *and* $\mathtt{b} \rightarrow \mathtt{c}$*, then* $\mathtt{a} \rightarrow \mathtt{c}$*.* □

If $\mathtt{a} \rightarrow \mathtt{b}$, we say that $\mathtt{b}$ *happens after* $\mathtt{a}$. Because an event which precedes another event according to the HB relation may have an effect on the latter, the HB relation is also sometimes called the *causality relation* [54], and if $\mathtt{a} \rightarrow \mathtt{b}$, then it is also said that $\mathtt{a}$ *causally affects* $\mathtt{b}$. Intuitively, when $\mathtt{a} \rightarrow \mathtt{b}$, there is a "path" from event $\mathtt{a}$ to event $\mathtt{b}$ consisting of "internal steps" between successive events in the same process and "message steps" from one process to the other.

**Example 3.2** In Figure 3.1, we show the time axes of three processes $P_1, P_2, P_3$, three message transfers denoted by arrows (and so three message send and three message receive events), and four internal events. In this example, we have for instance $\mathtt{a} \rightarrow \mathtt{b}$, $\mathtt{a} \rightarrow \mathtt{c}$, and $\mathtt{a} \rightarrow \mathtt{d}$ as relations within the same process, $\mathtt{b} \rightarrow \mathtt{e}$, $\mathtt{f} \rightarrow \mathtt{i}$, and $\mathtt{j} \rightarrow \mathtt{d}$ as relations between corresponding send and receive events, and $\mathtt{a} \rightarrow \mathtt{e}$, $\mathtt{f} \rightarrow \mathtt{j}$, and $\mathtt{e} \rightarrow \mathtt{d}$ as relations due to transitivity. Of the pairs of events $\mathtt{b}$ and $\mathtt{h}$, and $\mathtt{c}$ and $\mathtt{f}$, neither one happens before the other. □

The occurrence of unrelated events in Example 3.2 shows that the HB relation imposes only a partial order on events, which motivates the following definition of of the *concurrency relation*.

**Definition 3.3** *Two events* $\mathtt{a}, \mathtt{b} \in E$ *are* **concurrent** *(written as* $\mathtt{a}||\mathtt{b}$*) when neither* $\mathtt{a} \rightarrow \mathtt{b}$ *nor* $\mathtt{b} \rightarrow \mathtt{a}$ *holds.* □

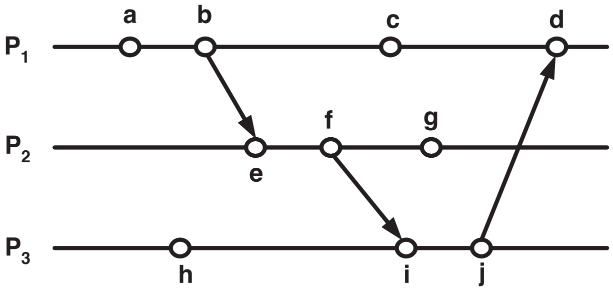


Figure 3.1: An example of the HB relation.

In particular, for every event $\mathtt{a}$, we have $\mathtt{a}||\mathtt{a}$. When we define for any event $\mathtt{a}$ the (*causal*) *past* $P(\mathtt{a})$, the set $C(\mathtt{a})$ of events concurrent with $\mathtt{a}$, and the (*causal*) *future* $F(\mathtt{a})$ by

$$\begin{array}{rcl} P(\mathtt{a}) & = & \{\mathtt{b} \in E \,|\, \mathtt{b} \to \mathtt{a}\}, \\ C(\mathtt{a}) & = & \{\mathtt{b} \in E \,|\, \mathtt{a}||\mathtt{b}\}, \\ F(\mathtt{a}) & = & \{\mathtt{b} \in E \,|\, \mathtt{a} \to \mathtt{b}\}, \end{array}$$

we have $E = P(\mathtt{a}) \cup C(\mathtt{a}) \cup F(\mathtt{a})$. The causal past of an event with the event itself included has also been called the *causal history* of the event [54].

### 3.1.2 Logical Clocks

We will now define *logical clocks*, which are functions that assign an element of some totally ordered set to each event—which we will call its *timestamp*—in a way that respects the HB relation in some way.

**Definition 3.4** *(a) A* **logical clock** *is a function $C : E \to S$, with $S$ a partially ordered set with partial order $\prec$, which is* **consistent** *with the HB relation, which means that $C(\mathtt{a}) \prec C(\mathtt{b})$ for every two events $\mathtt{a}, \mathtt{b} \in E$ with $\mathtt{a} \to \mathtt{b}$.*

*(b) A logical clock* **characterizes** *the HB relation if for any two events $\mathtt{a}, \mathtt{b} \in E$, $C(\mathtt{a}) \prec C(\mathtt{b})$ iff $\mathtt{a} \to \mathtt{b}$.* □

The condition of part (a) of Definition 3.4 is sometimes called the *weak clock condition*, and the condition of part (b) the *strong clock condition.*

For a logical clock $C$, we denote by $C_i$ the restriction $C|_{E_i}$ of $C$ to the events in $P_i$. Clearly, a function $C : E \to S$ is a logical clock if the following two conditions are satisfied:

1. If $\mathtt{a}, \mathtt{b} \in E_i$ and $\mathtt{a} \to \mathtt{b}$, then $C_i(\mathtt{a}) \prec C_i(\mathtt{b})$, for $i = 1, \ldots, n$.

2. If $\mathtt{a} \in E_i$ is the event in $P_i$ of sending message $\mathtt{m}$ and $\mathtt{b} \in E_j$ is the event in $P_j$ of receiving message $\mathtt{m}$ for some $i, j$ with $i \neq j$, then $C_i(\mathtt{a}) \prec C_j(\mathtt{b})$.

A natural choice for $S$ is the set $\mathbb{N}$ of natural numbers with the usual ordering. However, although we will see that such a logical clock may be consistent with the HB relation, it can never characterize it, because on the one hand there may be concurrent events, but on the other hand, the natural numbers are totally ordered. This observation has led to the notion of *vector clocks*, which have $S = \mathbb{N}^k$ for some $k > 1$. Denoting the $i$-th element of a vector $v \in \mathbb{N}^k$ by $v[i]$, we define the following relationships on vectors; for $k \geq 2$ the relations $<$ and $\leq$ are non-total orders on $\mathbb{N}^k$.

**Definition 3.5** *Let $v, w \in \mathbb{N}^k$. Then*

$$\begin{aligned}
v = w &\Leftrightarrow v[i] = w[i],\ i = 1, \ldots, k \\
v \leq w &\Leftrightarrow v[i] \leq w[i],\ i = 1, \ldots, k \\
v < w &\Leftrightarrow v[i] \leq w[i],\ i = 1, \ldots, k,\ \ and\ \ v \neq w \\
v \geq w &\Leftrightarrow v[i] \geq w[i],\ i = 1, \ldots, k \\
v > w &\Leftrightarrow v[i] \geq w[i],\ i = 1, \ldots, k,\ \ and\ \ v \neq w
\end{aligned}$$

For $v, w \in \mathbb{N}^k$ we define the maximum $\max(v, w) \in \mathbb{N}^k$ by $\max(v, w)[i] = \max(v[i], w[i])$, $i = 1, \ldots, k$. The unit vector in dimension $i$ is denoted by $e_i$.

**Definition 3.6** *A* **$k$-dimensional vector logical clock** *is a logical clock with $S = \mathbb{N}^k$ with the order $\leq$ as in Definition 3.5.* □

We will refer to a 1-dimensional vector logical clock as a scalar logical clock. Such a scalar logical clock $C$ can be constructed by having each process $P_i$ maintain an integer counter—by a slight abuse of notation also denoted by $C_i$—with initial value 0 (or any other non-negative value) which is used in the following way:

1. If $\mathtt{a} \in E_i$ and if $\mathtt{a}$ is not a message-receive event, then $P_i$ first increments $C_i$ by 1, and then sets $C(a)$ equal to the new value of $C_i$.

2. If $\mathtt{a} \in E_i$ is the event in $P_i$ of sending message $\mathtt{m}$ and $\mathtt{b} \in E_j$ is the event in $P_j$ of receiving message $\mathtt{m}$ for some $i, j$ with $i \neq j$, then $P_i$ sends $C(\mathtt{a})$ along with message $\mathtt{m}$ to $P_j$. On receipt of $\mathtt{m}$, $P_j$ first assigns $C_j$ the value $\max(C_j + 1, C(\mathtt{a}) + 1)$, and then sets $C(\mathtt{b})$ equal to the new value of $C_j$.

One easily proves the following theorem, which says that indeed this function is consistent with the HB relation.

**Theorem 3.7** *The function C constructed above is a scalar logical clock.* □

The scalar logical clock $C$ induces a partial order on the set of events $E$. However, different events (in different processes) may get the same timestamp. In order to extend this partial order to a total order on events, processor numbers can be used as tie breakers when two events have equal logical-clock values. So then the timestamp of $\mathtt{a} \in E_i$ has the form $(C(\mathtt{a}), i)$, and $(C(\mathtt{a}), i) \prec (C(\mathtt{b}), j)$ iff $C(\mathtt{a}) < C(\mathtt{b})$, or $C(\mathtt{a}) = C(\mathtt{b})$ and $i < j$.

An $n$-dimensional vector logical clock $V$ can be constructed by having each process $P_i$ maintain an $n$-vector $V_i$ of integers with initial values 0, which is used in the following way:

1. If $\mathtt{a} \in E_i$ and if $\mathtt{a}$ is not a message-receive event, then $P_i$ first increments $V_i[i]$ by 1, and then sets $V(\mathtt{a})$ equal to the new value of $V_i$.
2. If $\mathtt{a} \in E_i$ is the event in $P_i$ of sending message $\mathtt{m}$ and $\mathtt{b} \in E_j$ is the event in $P_j$ of receiving message $\mathtt{m}$ for some $i, j$ with $i \neq j$, then $P_i$ sends $V(\mathtt{a})$ along with message $\mathtt{m}$ to $P_j$. On receipt of $m$, $P_j$ first assigns $V_j$ the value $\max(V_j + e_j, V(\mathtt{a}))$, and then sets $V(\mathtt{b})$ equal to the new value of $V_j$.

With this construction of a vector clock, a process $P_i$ simply enumerates all of its own events in $E_i$ in the component $V_i[i]$, while the components $V_i[j]$ with $j \neq i$ indicate the last event in $P_j$ "that $P_i$ has heard of," either directly or indirectly. This interpretation is the reason for the initialization to all zeroes of the vector clocks of all processes.

**Theorem 3.8** *The function $V$ constructed above is a vector logical clock that characterizes the HB relation.*

PROOF. By the construction of the vector clock $V$ above, if $\mathtt{a} \rightarrow \mathtt{b}$ then $V(\mathtt{a}) < V(\mathtt{b})$. The proof of the converse is left as an exercise to the reader. □

As a consequence of Theorem 3.8, two events $\mathtt{a}$ and $\mathtt{b}$ are concurrent when neither $V(\mathtt{a}) < V(\mathtt{b})$ nor $V(\mathtt{b}) < V(\mathtt{a})$ holds.

One can ask the question if the dimension of vector logical clocks has to be (at least) equal to the number of processes in the system, as is the case in the logical clock constructed above. Below we show that this is indeed the case, provided that we require the comparison relations on $\mathbb{N}^k$ as defined in Definition 3.5. If we drop this restriction, we can of course always get away with scalar timestamps, because we can map $\mathbb{N}^k$ bijectively onto $\mathbb{N}$—a vector of dimension $k$ can of course always simply be represented by a natural number.

**Theorem 3.9** *If $V : E \rightarrow \mathbb{N}^k$ is a $k$-dimensional vector logical clock for a system of $n$ processes with the $<$-relation on $\mathbb{N}^k$ as in Definition 3.5 that characterizes the HB relation, then $k \geq n$.*

PROOF. For $n = 2$, take a system of two processes that never communicate, that do have internal events, and that start with scalar logical clock values of 0. Then the event $\mathtt{a}$ with logical time 1 in process 1 has a lower clock value than the event $\mathtt{b}$ with time 2 in process 2, but $\mathtt{a} \rightarrow \mathtt{b}$ does not hold, so $k \geq 2$ is needed.

Now assume that $n > 2$. We devise a scenario of events in which $k \geq n$ is necessary, which is shown in Figure 3.2. In this scenario, each process $P_i, i = 0, \ldots, n-1$, sends a message to each of the processes $P_{i+1}, P_{i+2}, \ldots, P_{i-2}$, in this order (in this proof we assume that the arithmetic on process indices is modulo $n$). Only after process $P_i$ has sent all of these messages it receives the messages sent to it by $P_{i-1}, P_{i-2}, \ldots, P_{i+2}$, again in this order. Let $\mathtt{a}_i, \mathtt{b}_i \in E_i$ be the events in $P_i$ of sending the message to $P_{i+1}$ and of receiving the message from $P_{i+2}$, respectively, which are the first and the last events in $P_i$ in this scenario.

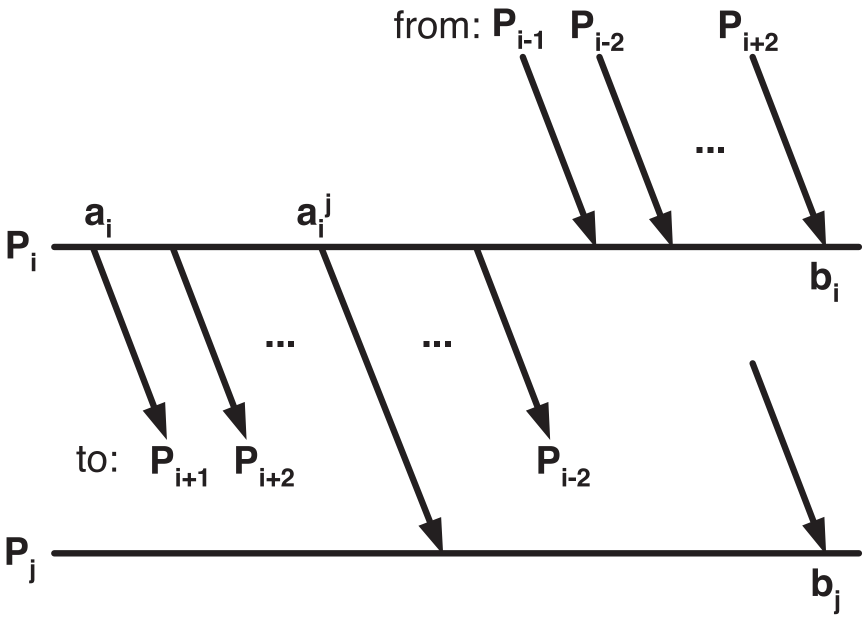


Figure 3.2: The scenario of messages in the proof of Theorem 3.9.

We claim that $\mathtt{a}_i \to \mathtt{b}_j$ iff $j \neq i-1$. Clearly, if $j \neq i-1, i$, and if $\mathtt{a}_i^j$ is the event of sending a message from $P_i$ to $P_j$, we have $\mathtt{a}_i \to \mathtt{a}_i^j \to \mathtt{b}_j$. In addition, when $j = i$, of course $\mathtt{a}_i \to \mathtt{b}_j$. On the other hand, any transitive path of events can only extend across two processes (because all processes first perform all send actions), and as no message is sent from $P_i$ to $P_{i-1}$, this proves our claim.

Let $V$ be a vector clock that characterizes causality. Because of our claim, $\mathtt{a}_{i+1} \to \mathtt{b}_i$ does not hold, so there exists an index $l(i)$, $0 \leq l(i) \leq k-1$, such that $V(\mathtt{b}_i)[l(i)] < V(\mathtt{a}_{i+1})[l(i)]$. If $k < n$, then for some $i, j$, with $i \neq j$, we must have $l(i) = l(j)$. Because by our claim $\mathtt{a}_{i+1} \to \mathtt{b}_j$, we then have

$$V(\mathtt{b}_i)[l(i)] < V(\mathtt{a}_{i+1})[l(i)] \leq V(\mathtt{b}_j)[l(j)] < V(\mathtt{a}_{j+1})[l(j)],$$

which in turn contradicts our claim. □

In large systems with many processes, the overhead of sending time vectors along with messages is large. More efficient schemes have been devised in which only a part of the vectors are sent along with messages, but in which more local data are maintained in order to reconstruct the complete vector logical clocks [54, 57]. two arrays,

Another scheme to reduce the length of vector clocks is constituted by plausible clocks [62]. A *plausible clock* assigns unique timestamps to events and satisfies the weak clock condition, but may erroneously "think" that two events are causally related while they are concurrent. Two plausible clocks can be combined (by taking pairs of timestamps) to yield a plausible clock which is more accurate in that the combination makes fewer of these errors. For more details, see [62].

## 3.2 Synchronizers

In this section we will present (local) simulations of synchronous systems on top of asynchronous systems (see also Section 2.3). The distributed algorithms that achieve this are called *synchronizers*. These synchronizers make it possible to run synchronous algorithms, which proceed in rounds of sending messages, receiving messages, and performing local computations, on asynchronous systems. The basic problem in a synchronizer is when a process can decide when to move to the next round, that is, when the synchonizer can issue a (*clock*) *pulse*. It can only do so when it has received all its messages of the current round, but it has no (direct) way of knowing this fact as message delays are unbounded. The basic solution to this problem is to let the senders rather than the receivers figure out that all of their messages in a certain round have been received through acknowledgments, and then let them notify the receivers.

The implementation of this solution, which was called the *alpha-synchronizer* [3], is very time-efficient but not very communication-efficient, as there is communication along every link in the system. Therefore, also the *beta-synchronizer* [3] was proposed, in which a spanning tree is constructed and in which communication only takes place along the links in the tree. This solution is communication-efficient, but not very time-efficient as the whole tree has to be traversed from the root to the leaves and back again. The *gamma-synchronizer* combines the ideas of the *alpha-synchronizer* and *beta-synchronizer* to achieve an overall efficient solution. In order to do so, first the nodes are clustered into separate clusters with a *preferred link* between each pair of clusters, then the beta-synchronizer is applied in every cluster, and finally the alpha-synchronizer is applied among the clusters. The fact that a single cluster can proceed to the next round is communicated via the preferred links.

**Algorithm 3.10** *Awerbuch's alpha-synchronizer* [3].

**Idea:** After a node has received a message of the (synchronous) distributed algorithm that runs on the simulated system, it sends an ACK message back to the sender. When a process has received an ACK for every message it has sent in some round, it is called *safe*. It then sends a SAFE message indicating that it is safe to all its neighbors (those nodes it has a direct connection to). When a node is safe itself and has received a SAFE message from all of its neighbors, it can proceed to the next round.

Alternatively, every node can send to each of its neighbors exactly one message in every round, by sending an empty message if it did not really want to send something, and by packing multiple messages into a single large message. Then, a node can proceed to the next round when it has received exactly one message from each of its neighbors.

**Complexity:** The (additional) communication complexity is twice the number of edges in the system, as an additional ACK and SAFE message are sent across each link. When the number of nodes is $N$, in the worst case the communication complexity is of order $O(N^2)$. The (additional) time complexity is constant. □

We now turn to the more communication-efficient, but less time-efficient beta-synchronizer.

**Algorithm 3.11** *Awerbuch's beta-synchronizer* [3].

**Idea:** The nodes in the system first elect a leader, and then create a spanning tree with the leader as the root. In every round, there is a wave of PULSE messages from the leader downwards in the tree indicating that all nodes can start the next round, and a convergecast of SAFE messages from the leaves of the tree upwards to the root. When a leaf of the tree has received a PULSE message and it knows it is safe (in the same sense as in Algorithm 3.10), it sends a SAFE message to its parent. When a non-leaf node in the tree has received a SAFE message from all of its descendents and is safe itself, it sends a SAFE message to its parent. When the root has received a SAFE message from all its descendents and is safe itself, it generates the next PULSE.

**Complexity:** If $N$ is the number of nodes, both the time complexity and the message complexity is of order $O(N)$, as there are $N-1$ links in the spanning tree, and the maximal depth of the tree is of order $O(N)$. □

Finally, below is the gamma synchronizer, which combines the previous two synchronizers. We do not concern ourselves with how to first cluster the processes.

**Algorithm 3.12** *Awerbuch's gamma-synchronizer* [3].

**Idea:** In the *initilization phase*, the nodes have to be partitioned into clusters, each of the clusters has to elect a leader and has to create a spanning tree with that leader as the root, and a single preferred link has to be created between any two nodes of every pair of clusters. In the algorithm below, we assume that these operations have already been performed. Again, all the messages of the synchronous algorithm are acknowledged; a node keeps track of the number of ACKs still to be received from its neighbors in the `dif` variables (code fragments I-IV).

For every node, the following three sets of node ids are defined:

- the set `D` of descendents in the spanning tree in the cluster the node belongs to;
- the set `N` of neighbors in the whole system;
- the set `P` of the nodes at the other ends of the preferred links the node is connected to.

In addition, every node maintains the id of its parent in the local tree in the variable `parent`; for a root, we assume that `parent` is equal to its own id.

In every round, first in every cluster the beta-synchronizer is applied, with a wave of PULSE messages downwards in the spanning trees (code fragment I), and a wave of SAFE messages upwards to the roots of these trees (code fragment VI). When the root of a tree has received a SAFE message from all its descendents, it sends a wave of CLUSTER_SAFE messages downwards in its tree, including a message to itself (code fragments VI, VIII); this message is also sent along all preferred links to which the tree is connected to communicate to neighboring clusters that the cluster is safe (code fragment VIII). Finally, a node sends a READY message upwards in the tree it belongs to when it has received a READY message from all its descendents, and a CLUSTER_SAFE message across all preferred links it is connected to (code fragment X).

When each node constitutes a cluster by itself, the gamma-synchronizer amounts to the alpha-synchronizer, and when there is only a single cluster, the gamma-synchronizer coincides with the beta-synchronizer.

**Implementation:**

```
I. Reception of a PULSE message
upon receipt of (PULSE) do
    execute next round of synchronous algorithm
    for all (j in D) do
        safe(j) ← 0
        send (PULSE) to j
    for all (j in N) do dif(j) ← 0
    for all (j in P) do cluster_safe(j) ← 0
```

```
II. Sending a message of the synchronous algorithm
send (message) to j
    dif(j) ← dif(j)+1
```

```
III. Receiving a message of the synchronous algorithm
upon receipt of (message) from j do
    send (ACK) to j
```

```
IV. Receiving an ACK
upon receipt of (ACK) from j do
    dif(j) ← dif(j)-1
```

```
V. A round has been completed
when (all actions of a round have been completed) then
    safe_propagation()
```

```
VI. Procedure safe_propagation()
if ((dif(j)=0 for all j in N) and (safe(j)=1 for all j in D)) then
    if (i ≠ root) then send (SAFE) to parent
    else send (CLUSTER_SAFE) to i
```

VII. Reception of a SAFE message from a descendent

```
upon receipt of (SAFE) from j do
    safe(j) ← 1
    safe_propagation()
```

VIII. Reception of a CLUSTER_SAFE message

```
upon receipt of (CLUSTER_SAFE) from j do
    if (j in P) then cluster_safe(j) ← j
    if (j = parent) then
        for all (j in D) do
            send (CLUSTER_SAFE) to j
            ready(j) ← 0
        for all (j in P) do
            send (CLUSTER_SAFE) to j
    ready_propagation()
```

IX. Receiving a READY message

```
upon receipt of (READY) from j do
    ready(j) ← 1
    ready_propagation()
```

X. Procedure ready_propagation()

```
if ((ready(j)=1 for all j in D) and (CLUSTER_SAFE(j) = 1) for all j in P)) then
    if (i ≠ root) then
        send (READY) to parent
    else send (PULSE) to i
```

**Complexity:** Let $E$ be the set of all links in the spanning trees of the clusters and the preferred links, and let $H$ be the maximal height of the spanning trees. Because at most four messages are sent across all links in $E$, and because the four waves of downward and upward messages have to traverse a tree of depth at most $H$, the communication complexity if of order $O(E)$, and the time complexity is of order $O(H)$. Of course, these complexities depend on the clustering. □

Because of the existence of synchronizers, the reader should not be led to conclude that synchronous and asynchronous systems are of equal power. Synchronizers only work in fault-free systems. When faults may occur, synchronous systems are definitely more powerful, as we will see in Chapter 5.

## 3.3 Message Ordering

In asynchronous distributed systems, messages have arbitrary (but finite) delays. However, some applications may impose conditions on the order in which messages are received. For instance, when a user posts an article on a bulletin board to which another user replies, a third user would like to see the original message before he sees the reply. In this section we will use scalar and vector timestamps to enforce different orderings of message-send and message-receive events. As a basis, we assume an asynchronous system which only has capabilities for point-to-point messages, and which does not have the FIFO property, that is, even between two processes, messages may not be received in the order sent.

Some of the orderings that we will study are for systems in which the same message is sent to a group of processes rather than to a single process, which is called *multicasting*. For a message $\mathtt{m}$, the group of processes to which it is multicast is denoted by $Dest(\mathtt{m})$. In particular, for a *broadcast* message $\mathtt{m}$, we have $Dest(\mathtt{m}) = \{P_1, P_2, \ldots, P_n\}$ (a message broadcast by a process is also (logically) sent to and received by the process itself). We consider multicasting a message $\mathtt{m}$ to all of its destinations as a single event denoted by $m(\mathtt{m})$.

In some applications it may be necessary to enforce messages to be used by processes in an order which is consistent with the HB relation, or, in other words, in a *causal order*. An example of such an application is maintaining the consistency of replicas in distributed databases. When in an application the event of sending some message $\mathtt{m}$ happens before the event of sending another message $\mathtt{m}'$ (possibly by another process), the contents of $\mathtt{m}'$ may depend on the contents of $\mathtt{m}$. As a consequence, if a process is supposed to receive both messages, it should receive $\mathtt{m}$ before $\mathtt{m}'$, because otherwise it may not "understand" the contents of $\mathtt{m}'$.

As messages may arrive at a processor in an order which does not obey the required ordering, we make a distinction in this section between *receiving* a message and *delivering* a message. The easiest way to do so is to imagine that there are two processes on each processor. One process receives messages, checks their order, and maintains a buffer with messages that cannot yet be delivered. The other process is the application process proper to which the messages are delivered in the (a) correct order. The separation between receiving and delivering messages is illustrated in Figure 3.3. The event of delivering message $\mathtt{m}$ to process $P_i$ with $i \in Dest(\mathtt{m})$ is denoted by $d_i(\mathtt{m})$.

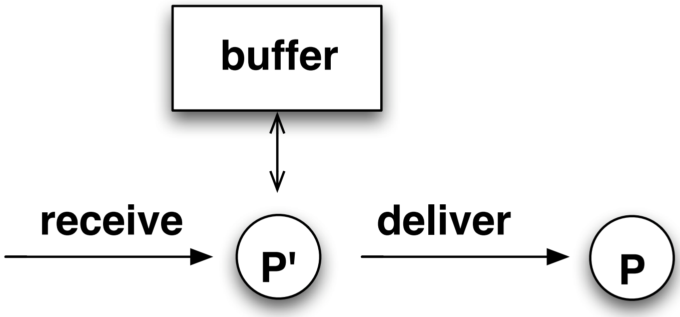


Figure 3.3: The distinction between receiving and delivering a message.

The implementation of some form of ordering of messages in a distributed system on top of a basic messaging system can also be seen as a simulation in the sense of Section 2.3.

**Example 3.13** Suppose in a system with three processes $P_1, P_2, P_3$, first $P_1$ broadcasts a message $\mathtt{m}_1$, and then $P_2$ on receipt of $\mathtt{m}_1$ broadcasts a message $\mathtt{m}_2$. Then $\mathtt{m}_1$ may arrive at $P_3$ later than $\mathtt{m}_2$, and so the delivery of $\mathtt{m}_2$ at $P_3$ should be delayed until $\mathtt{m}_1$ has been received and delivered. See Figure 3.4 for this situation. □

We now first define the causal and total ordering of messages.

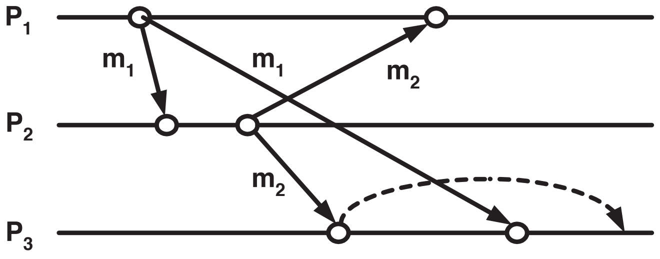


Figure 3.4: An example of the causal message ordering of broadcast messages.

**Definition 3.14** *(a) Message order is* **causal** *when for every two messages* $\mathtt{m}_1$ *and* $\mathtt{m}_2$*, if* $m(\mathtt{m}_1) \rightarrow m(\mathtt{m}_2)$*, then* $d_i(\mathtt{m}_1) \rightarrow d_i(\mathtt{m}_2)$ *for all* $i \in Dest(\mathtt{m}_1) \cap Dest(\mathtt{m}_2)$*.*

*(b) Message order is* **total** *when for every two messages* $\mathtt{m}_1$ *and* $\mathtt{m}_2$*,* $d_i(\mathtt{m}_1) \rightarrow d_i(\mathtt{m}_2)$ *if and only if* $d_j(\mathtt{m}_1) \rightarrow d_j(\mathtt{m}_2)$ *for all* $i, j \in Dest(\mathtt{m}_1) \cap Dest(\mathtt{m}_2)$*.* □

As a consequence of this definition, causal message ordering implies FIFO message ordering along the separate channels. However, total ordering does not imply causal ordering, nor vice versa. An application of message ordering is in replicated databases, where every replica has to process the updates in a causal or total order. For proving an algorithm claiming to implement one of these message orderings to be correct, we have to prove its safety, which means that it satisfies its definition, and its liveness, which means that every message sent is eventually delivered.

**Algorithm 3.15** *The Birman-Schiper-Stephenson algorithm for causal message ordering of broadcast messages* [6].

**Idea:** This algorithm uses vector logical clocks implemented by the variable $V$ in every process, initialized to all zeroes, for the sole purpose of causal message ordering, but not for affixing timestamps to other events in the system. The links do not have to be FIFO. Every process numbers its broadcasts consecutively in its own component of its vector logical clock and sends this vector along with the message to all other processes. If on receipt of a message a process finds that it cannot yet be delivered, it puts it into a buffer $B$ of pending messages (along with their timestamps). We define the following condition for a message m that carries timestamp $V_{\mathtt{m}}$ and is received from process $P_j$ to indicate whether it can be delivered:

$$D_j(\mathtt{m}) = (V + e_j \geq V_{\mathtt{m}}).$$

Condition $D_j(\mathtt{m})$ says that message m is the message expected next from $P_j$, and that the receiving process is at least as up to date with respect to all other processes as $P_j$ was when it sent m.

**Implementation:**

```
I. Broadcasting a message
$V \leftarrow V + e_i$
broadcast(m,$V$)

III. Delivering a message from $P_j$
deliver(m):
deliver(m)
$V \leftarrow V + e_j$
remove $(\mathtt{m}, V_\mathtt{m})$ from $B$
```

```
II. Receiving a message from $P_j$
upon receipt of $(\mathtt{m}, V_\mathtt{m})$ do
    if $D_j(\mathtt{m})$ then
        deliver(m)
        while $(\{(\mathtt{m}, \mathtt{k}, V_\mathtt{m}) \in B | D_k(\mathtt{m})\} \neq \emptyset)$ do
            deliver(m) with $(\mathtt{m}, V_\mathtt{m}) \in B$ such that $D_j(\mathtt{m})$
    else add $(\mathtt{m}, \mathtt{j}, V_\mathtt{m})$ to $B$
```

**Correctness:** First we prove the safety of the algorithm, that is, we prove that if $m(\mathtt{m}_1) \rightarrow m(\mathtt{m}_2)$, then $d_i(\mathtt{m}_1) \rightarrow d_i(\mathtt{m}_2)$ for $i = 1, \ldots, n$. Because predicate $D_j$ already checks the order of the broadcasts from the same process, we can assume that $\mathtt{m}_1$ is broadcast by process $P_j$ and $\mathtt{m}_2$ is broadcast by process $P_k$, with $j \neq k$. Because $m(\mathtt{m}_1) \rightarrow m(\mathtt{m}_2)$, $V(\mathtt{m}_1)[j] \leq V(\mathtt{m}_2)[j]$. By code fragment III, the only modification made to the vector logical time in the receiving process when a message from process $P_j$ is delivered is adding $e_j$ to its vector clock. Now $\mathtt{m}_2$ can only be delivered in $P_i$ when $V_i[j] \geq V(\mathtt{m}_2)[j]$, but $V_i[j]$ can only attain this value after $\mathtt{m}_1$ has been delivered ($V_i$ is here the vector clock in the receiving process $P_i$).

The proof of the liveness of the algorithm is left as an exercise for the reader. □

We now turn to an algorithm for causal ordering of point-to-point messages. This is more difficult than ordering broadcast messages because processes do not know which message next to expect from another process.

**Example 3.16** In Figure 3.5, first $P_1$ sends a message $\mathtt{m}_1$ to $P_3$ that is slow in arriving at $P_3$. Then, $P_1$ send a message $\mathtt{m}_2$ to $P_2$, which then sends a message $\mathtt{m}_3$ to $P_3$. When the latter message arrives first at $P_3$, causal message ordering is violated. □

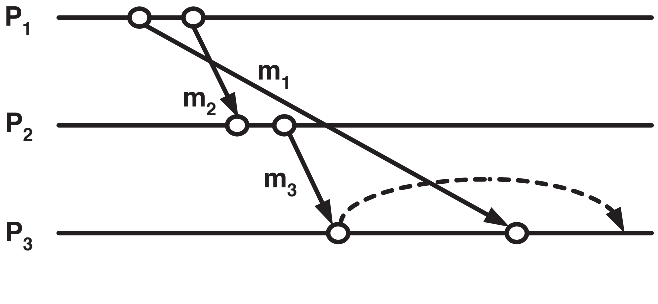


Figure 3.5: An example of the causal message ordering of point-to-point messages.

**Algorithm 3.17** *The Schiper-Eggli-Sandoz algorithm for causal message ordering of point-to-point messages* [52].

**Idea:** The processes use vector logical clocks, initialized to all zeroes, in the ordinary way, that is, for assigning timestamps to all events in the system, as opposed to Algorithm 3.15. For this reason, explict vector clock operations are omitted from the code fragments below. Every process maintains an initially empty local buffer $S$ of ordered pairs, each made up of a process id and a vector time stamp. When a process sends a message, it sends the current contents of $S$ along; only then does it insert into $S$ the pair corresponding to the current message, deleting any pair for the same process that was already contained in $S$. (This is the implied meaning of the insert operation in code fragment I. below.) When a process receives a message which cannot yet be delivered, it is put into a buffer $B$ of pending messages (along with its timestamp and the accompanying buffer). The condition for delivering message m with accompanying buffer $S_{\mathtt{m}}$ to process $P_i$ with vector clock $V$ is

$$D_i(\mathtt{m}) = (\text{there does not exist } (i, V') \in S_{\mathtt{m}}) \textbf{ or } (\text{there exists } (i, V') \in S_{\mathtt{m}} \textbf{ and } V' \leq V),$$

which states that either the process sending m does not know, either directly or indirectly, about messages sent to $P_i$, or it does so but $P_i$ has at least the same amount of knowledge the sending process had when m was sent. When a message with accompanying buffer $S_{\mathtt{m}}$ is delivered, $S_{\mathtt{m}}$ and $S$ are merged. When for some process only one of these buffers contains an element, it is retained in the result. When both contain an element, the pairwise maximum of the components of the vector timestamps are taken. The links do not have to be FIFO.

**Implementation:**

```
I. Sending a message to $P_j$
**send**$(\mathtt{m}, S, V)$ **to** $P_j$
insert $(j, V)$ into $S$
```

```
II. Receiving a message
Code fragment II. of Algorithm 3.15 with $(\mathtt{m}, V_{\mathtt{m}})$
replaced by $(\mathtt{m}, S_{\mathtt{m}}, V_{\mathtt{m}})$.
```

```
III. Delivering a message
deliver$(\mathtt{m}, S_{\mathtt{m}})$:
deliver$(\mathtt{m})$
**for all** $((j, V') \in S_{\mathtt{m}})$ **do**
    **if** (there exists $(j, V'') \in S$) **then**
        $S \leftarrow S - \{(j, V'')\}$
        $V'' \leftarrow \max(V', V'')$
        $S \leftarrow S \cup \{(j, V'')\}$
    **else** $S \leftarrow S \cup \{(j, V')\}$
```

**Correctness:** See [52]. □

We now turn to total message ordering, in which every two processes receive all messages that are sent to both of them in the same order. In particular, when all message are broadcast (also to the sending processes themselves), then all processes receive all messages sent by all processes in the same order.

A very simple but efficient solution with a central component for total ordering [12] uses a special process (that can for instance be elected, see Section 4.3) to which all processes send their broadcast messages. This special process gives each message a sequence number, and then broadcasts it to all processes (including the originator). The order of the broadcasts of a single process can be guaranteed if each process numbers its own broadcasts sequentially before sending them to the special process. The complexity of this algorithm is $O(N)$ if there are $N$ processes. Some form of reliability can be implemented by having each process check whether it receives all broadcasts by simply checking their sequence numbers, and requesting a resend from the special process when it misses messages.

We now present a simple but truly distributed algorithm for totally ordering broadcast messages, which is in effect but a slight variation of Algorithm 4.1 for mutual exclusion.

**Algorithm 3.18** *An algorithm for total ordering of broadcast messages.*

**Idea:** In this algorithm, the links between processes are assumed to be FIFO. Scalar clocks are used with process ids as tie breakers, so events have a total order. Every process maintains a queue of messages that have been received but not yet delivered, ordered according to timestamp. Every process acknowledges every message received to all processes (including itself and the originating process); also acknowledgments carry timestamps. When a process has received an acknowledgment for the message at the head of its message queue from every process, that message can be delivered (and removed from the queue), and the corresponding acknowledgments can then be deleted.

**Correctness:** The acknowledgments for the message at the head of its message queue guarantee a process that it has not missed any older messages, as the channels are FIFO.

**Complexity:** When $N$ is the number of processes, the number of messages sent for every broadcast is of order $O(N^2)$, as every process sends an acknowledgment to every other process. □

## 3.4 Global States

A problem touching at the heart of distributed systems is the determination of the global state of an asynchronous system or an asynchronous computation. Solutions to this problem may for instance be used for debugging or checkpointing distributed applications, and for detecting *stable properties* such as deadlock and termination. A property of a distributed system is called stable when, once it holds, it will hold for ever (at least, if no drastic measures are taken, such as aborting a process to remove a deadlock).

In a central system with one processor, the determination of the global state is trivial (in theory): the processor can simply inspect the contents of its memory, or the application the values of its variables. When trying to determine the global state of an asynchronous system, there are two main problems. First, it won't do sending each of the participating processes a message telling them to record their own local states, as there is no way of synchronizing these recordings. Secondly, there can be messages in transit that should be included in the state: by state we mean the joint states of all processes and all communication channels in the system. The state of a process can be defined as the contents of (a part of) its memory, and the state of a unidirectional FIFO channel must be some subsequence of the sequence of messages sent along it. To see what this channel state should be, consider the following example.

**Example 3.19** Two processes $P_1$ and $P_2$ representing two bank accounts originally contain euro 100 and euro 0, respectively. Process $P_1$ successively sends messages indicating money transfers of 5, 10, 15, 20, and 25 to $P_2$, and records its own local state after it has sent the first four messages as E 50. Process $P_2$ records its local state after it has received the messages with 5 and 10 as 15. See Figure 3.6 for the processes and messages in this example. Then of course, the only logical choice for the state of the channel is the sequence of messages 15, 20. □

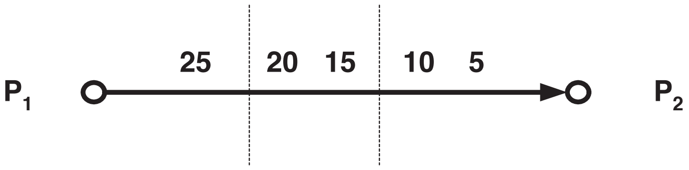


Figure 3.6: An example of recording the state of a channel.

So we see that the state of a channel should be defined as the sequence of messages sent along the channel before the sending process records its state minus the messages that have been received before the receiving process does so. In asynchronous systems, it turns out that it is too ambitious to determine a state the system has actually been in. In this section we present an algorithm to find a global state the system *might* have been in.

Basically, what we are trying to do when determining a global state is find a set of events, one in each process, that is in some sense consistent.

**Definition 3.20** *A consistent cut is a set* $C = \{c_1, c_2, \ldots, c_n\}$ *of internal events,* $c_i \in E_i$*, such that for every* $i, j = 1, 2, \ldots, n, i \neq j$*, there do not exist events* $e_i \in E_i$ *and* $e_j \in E_j$ *with*

$$(e_i \to e_j) \wedge (e_j \to c_j) \wedge (e_i \not\to c_i).$$

□

Note that in Definition 3.20, $e_i \not\to c_i$ implies $c_i \to e_i$ or $c_i = e_i$ because the events occurring in $P_i$ are totally ordered.

**Example 3.21** In Figure 3.7 we show examples of cuts in a system with three processes. □

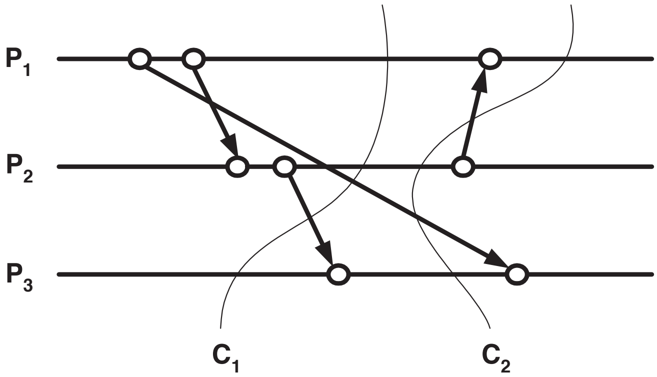


Figure 3.7: An example of a consistent ($C_1$) and an inconsistent cut ($C_2$).

Let's now assume that vector times are maintained, and define the vector time $V(C)$ of a cut by setting $V(C) = \max\{V(c_1), V(c_2), \dots, V(c_n)\}$. We have the following theorem.

**Theorem 3.22** (*a*) *A cut $C = \{c_1, c_2, \dots, c_n\}$ is consistent if and only if $c_i||c_j$ for $i, j = 1, 2, \dots, n$.*

(*b*) *A cut $C = \{c_1, c_2, \dots, c_n\}$ is consistent if and only if $V(C)[i] = V(c_i)[i]$ for $i = 1, 2, \dots, n$.*

PROOF. (a) Suppose that the cut is consistent but that not all the $c_i$ are mutually concurrent. If $c_i \rightarrow c_j$ for some $i, j$ with $i \neq j$, then there is a "path" of events $c_i \rightarrow e_i \rightarrow \cdots \rightarrow e_j \rightarrow c_j$ with $e_i \in E_i$ and $e_j \in E_j$. But then the events $e_i$ and $e_j$ violate the condition of Definition 3.20.

Conversely, suppose all the $c_i$ are mutually concurrent, but that the cut is not consistent. Then for some $i, j, i \neq j$, there exist $e_i \in E_i$ and $e_j \in E_j$ such that the condition of Definition 3.20 holds. Because $e_i \not\rightarrow c_i$, either $c_i = e_i$ or $c_i \rightarrow e_i$. But then we have $c_i \rightarrow e_j \rightarrow c_j$, which contradicts the concurrency of $c_i$ and $c_j$.

(b) Using (a), it is sufficient for a cut $C = \{c_1, c_2, \dots, c_n\}$ to show that $c_i||c_j$ for $i, j = 1, 2, \dots, n$ if and only if $V(c_i)[i] \geq V(c_j)[i]$ for $i, j = 1, 2, \dots, n$. Clearly, if $c_i||c_j$, then $c_i \rightarrow c_j$ does not hold, and so $V(c_i)[i] > V(c_j)[i]$. The converse is left as an exercise for the reader. □

**Algorithm 3.23** *Chandy's and Lamport's algorithm for determining a global state of a distributed system with unidirectional FIFO channels* [10].

**Idea:** We assume that in the graph with processes as nodes and the unidirectional channels as directed edges, there is a path from every process to every other process. Any processor wishing to record the global state of the system first records its own local state, and then sends a special message, a *marker*, along every outgoing channel. Upon the first receipt of a marker along any channel, a process records the state of that channel as the empty state, records its own local state, sends a marker along every outgoing channel, and creates an initially empty FIFO message buffer for each of its incoming channels (except for the one along which the first marker is received). Every message subsequently received along a channel is entered into the corresponding buffer. Upon any later receipt of a marker along a channel, the state of that channel is recorded as the sequence of messages in the corresponding buffer. A process has finished its part of the algorithm when it has received a marker along every incoming channel.

**Implementation:**

I. Spontaneously recording a processor's state

```
record_and_send_markers:
record local state
loc_state_recorded $\leftarrow$ true
for every outgoing channel $c$ do
    send marker along $c$
for every incoming channel $c$ do
    create message buffer $B_c$
```

II. Receiving a marker along a channel

```
upon receipt of marker along channel c do
    if ($\neg$ loc_state_recorded) then
        record state of $c$ as empty
        record_and_send_markers
    else
        record state of $c$ as contents of $B_c$
```

```
III. Receiving a message along a channel
upon receipt of m along channel c do
    append m to $B_c$
```

**Correctness:** First of all, the algorithm terminates because the directed graph of processes is connected. Let $S$ be the sequence of events in the system between the moments the first and the last processors record their local states. In Example 3.24 below the reader can see that the global state recorded may not actually have occurred between any two successive events in $S$. We will now show that there is a sequence $S'$ of events equivalent to $S$ such that the recorded state does occur in $S'$. We call an event local to some process a *pre-recording* (*post-recording*) event if it happened before (after) the process recorded its own state. Of course, for every process $P$, the (non-contiguous) subsequence of $S$ of events local to $P$ first contains all pre-recording, and then all post-recording events local to $P$, but $S$ may contain post-recording events before pre-recording events. If the latter is the case, $S$ must contain successive events $e$ in $P$ and $e'$ in $P'$ with $e$ ($e'$) post-recording (pre-recording). But that means that $e$ and $e'$ cannot be the events of sending and receiving some message m along a channel $c$ from $P$ and $P'$, for then, because $e$ is post-recording, a marker will have been sent along $c$ before m, contradicting the fact that $e'$ is postrecording. Now we can interchange $e$ and $e'$ to obtain an equivalent sequence. Doing this successively, we end up with the desired sequence $S'$.

The only thing to prove is that the sequences of messages on the channels in $S'$ after all pre-recording and before all post-recording events are identical to the states recorded for these channels by the algorithm. But this is trivially true, as the state of a channel recorded by the algorithm consists precisely of the messages received after the receiving process has recorded its own state and before it receives a marker along the channel.

**Example 3.24** Suppose we have two processes $P_1$ and $P_2$, representing bank accounts between which money is transferred through messages containing a number of Euros along two channels $C_1$ and $C_2$ (see Figure 3.8). Both accounts initially contain E 100. Process $P_1$ starts the algorithm by recording its own state as 100, by sending a marker (#) along channel $C_1$ to $P_2$, and by creating a buffer $B_2$ for messages to be received along channel $C_2$ (Figure 3.8.(a)). Then both processes send a message with contents 100 to the other (Figure 3.8.(b)), and process $P_1$ receives the message with 100, which it enters into $B_2$ (Figure 3.8.(c)). Then $P_2$ receives the marker along $C_1$, records its own state as 0, records the state of $C_1$ as empty, and sends a marker along channel $C_2$ (Figure 3.8.(d)). After $P_1$ receives the marker along $C_2$, it records the state of $C_2$ as the contents of $B_2$, which contains one message with contents 100 (Figure 3.8.(e)). Finally, $P_2$ receives the message with 100 and adapts its account to 100. (Figure 3.8.(f)).

From Figure 3.8 it can easily be seen that the state of the system that has been recorded has not actually occurred in the system. □

One may wonder what the use is of recording a state of a system that has not occurred. The point is that the recorded state can still be used for detecting stable properties such as deadlock and termination. To show this, let $T_i$ be the initial state of a system when the execution of Algorithm 3.23 starts, let $T_f$ be the final state after it finishes, and let $T_r$ be the state that has been recorded. Then there is a sequence of events that leads the system from state $T_r$ to $T_f$ (namely, the latter part of the reordered sequence of events as in the correctness proof above), and so any stable property that holds in $T_r$ also holds in $T_f$.

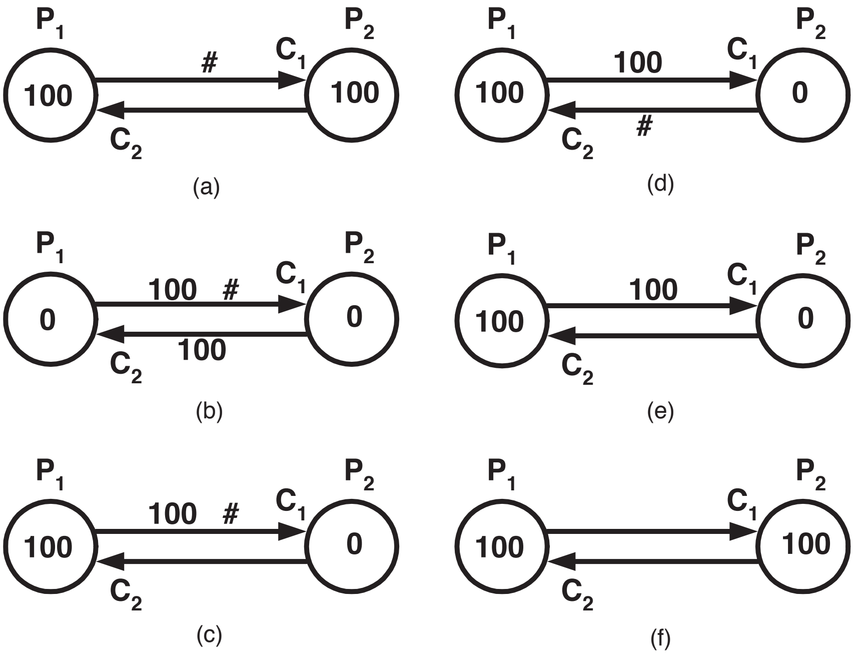


Figure 3.8: An example of recording a global state with Algorithm 3.23 that has not actually occurred in the system.

## 3.5 Termination Detection

An important problem in distributed systems that has played a crucial role in the development and understanding of distributed algorithms is *termination detection.* Concisely stated, termination detection is the problem of determining whether a distributed computation in a distributed system consisting of processes which communicate by means of messages, has terminated. The main difficulty is that no single process has complete information on the state of the computation. This state consists of the state of every process, and of the messages that have been sent but not yet received. A process may be idle at one time, but be activated at a later time because of the reception of a message.

The termination-detection problem can be stated in the following way:

- There is a distributed computation running on a DS with the following properties:
  - A process is either *active* or *passive*.
  - Only active processes can send messages.
  - An active process may become passive spontaneously.
  - A passive process becomes active at the reception of a message.
- Devise a distributed algorithm that enables one or all of the processes to determine that the computation has finished.

So when we have a solution to this problem, we have in effect two DAs running, the original distributed computation and the DA detecting termination. Passive processes are still allowed to participate in the termination-detection algorithm. Below, we present a solution for a unidirecional ring.

**Algorithm 3.25** *Termination detection in an asynchronous unidirectional ring with FIFO communication* [18].

**Idea:** We assume that process $P_0$ is connected to process $P_{n-1}$ in the direction of sending in the ring, and process $P_i$ to process $P_{i-1}, i = 1, 2, \dots, n-1$. Process $P_0$ has the special role of detecting termination. When it wants to do so, it sends a *token* (or probe, a special message that has nothing to do with the original computation) to $P_{n-1}$. In principle, the token is forwarded along the ring by passive processes, so that when $P_0$ receives it and is itself passive, it can conclude termination. The only problem with this naieve solution is that the sending of a message belonging to the original computation from $P_i$ to $P_j$ with $j > i$ activates $P_j$ "behind the token's back." Therefore, this fact has to be recorded in $P_i$, which is done by introducing a color (black or white) in the processes, and turning this color into black when such a message is sent. The token is also given a color, which is white when $P_0$ sends it, but which is changed into black when the token arrives in a black process to indicate that no decision can be reached during the current round of the token. When upon reception of the token in some process, either the process or the token is black, no decision can be reached during the current round, and the token can be relayed immediately. Else the token is sent or held according to the process's state. When the token is forwarded by a black process, the process turns white. When $P_0$ receives a white token while it is passive and white itself, it concludes termination; when it receives a black token, it starts over again.

**Implementation:**

```
I. Spontaneous state change
when (state = active) do
   state ← passive

II. Receiving a message of the computation
upon receipt of (message) do
   state ← active

III. Sending a message to a higher-numbered process
send(message) to $P_j$
if ($j > i$) then
   color_p ← black

IV. (Re-)initiating the token in $P_0$
when (token_present) do
   token_present ← false
   send(token;white)
   color_p ← white
```

```
V. Receiving the token in $P_i, i = 1, 2, \ldots, n-1$
upon receipt of (token;color_t) do
    token_present ← true
    if (color_t = black) or (color_p = black) then
        token_present ← false
        send(token;black)
        color_p ← white
    else if (state = passive) then
        token_present ← false
        send(token;white)
```

```
VI. Sending the token in $P_i, i = 1, 2, \ldots, n-1$
when (token_present and ((state = passive) or (color_p = black)) do
    token_present ← false
    send(token;color_p)
    color_p ← white
```

```
VII. Receiving the token in $P_0$
upon receipt of (token;color_t) do
    token_present ← true
    if ((color_t = white) and (color_p = white) and (state = passive)) then
        decide termination
```

**Correctness:** Let $\texttt{loct} = i$ if token_present = true in $P_i$, or if token_present = false in all processes and $P_i$ is the last process in which token_present = true was true. The color and state of $P_i$ are denoted by color_p_i and state_i in this proof. We define the following predicates:

$$\begin{aligned}\texttt{p} &= \forall\,(i : \texttt{loct} < i < n : \texttt{state\_i} = \texttt{passive})\\ \texttt{q} &= \exists\,(i : 0 \le i \le \texttt{loct} : \texttt{color\_p\_i} = \texttt{black})\\ \texttt{r} &= (\texttt{color\_t} = \texttt{black})\end{aligned}$$

We prove that $\texttt{p} \vee \texttt{q} \vee \texttt{r}$ always holds after initiation of the algorithm (i.e., when loct = n-1) until the token returns in $P_0$. Then from a white token in a white process $P_0$ (color_t = white, loct = 0, and color_p_0 = white) we conclude $\neg$ q and $\neg$ r, so p holds. If in addition $P_0$ is passive, we can conclude termination in VII.

To prove $\texttt{p} \vee \texttt{q} \vee \texttt{r}$, note that when $\texttt{loct} = n-1$, p is trivially true. With every step of the token, we can assume it remains white, otherwise r holds and continues to hold until the token returns in $P_0$, so we can assume that the token is always sent by a passive process (IV). Therefore, p continues to hold, unless a process $P_i$ with $\texttt{loct} < i < n$ turns active, which can only occur when a process $P_j$ with $0 \le j \le \texttt{loct}$ sends a message to a higher-numbered process and turns black, and so q holds. Then $P_j$ remains black (and q true) until the token passes through $P_j$ and q may become false but r turns true, and remains so until the token arrives in $P_0$.

**Algorithm 3.26** *Termination detection in a general network* [28, 40].

**Idea:** There is a special process $P$ that controls the algorithm, but does not participate in the computation proper. Processes and messages carry non-negative weights (or credits). Initially, $P$ has weight 1, and all other processes have weight 0. Then $P$ equally splits its weight over all processes that want to start their computation. When a process wants to send a message, it halves its own weight, and the message carries the other half. When a process receives a message, it adds the weight carried by the message to its own weight. When a process terminates its computation (it may later restart due to the reception of a message), it sends a special message with its remaining weight to $P$, and sets its own weight to $0$. Termination can be detected by $P$ when its weight is equal to $1$.

**Correctness:** Clearly, the sum of all weights in the system (of all processes including $P$ and of all messages in transit) is always equal to $1$. When $P$ has weight equal to $1$, this means that there are no messages in transit, and that all processes have ceased computing. $\Box$

## 3.6 Deadlock Detection

Similarly as in single, sequential systems, in distributed systems deadlocks can occur. The formulation of the problem of deadlocks in distributed systems is exacly the same as in centralized systems: when multiple processes request sets of resources, partial allocations of those resources can lead to a situation in which a number of processes are all waiting for one or more other processes to release resources.

Traditionally, there are three methods for dealing with deadlocks. Deadlocks can be prevented apriori by the system's design. For instance, linearly ordering the resources and allowing processes only to request resources that are lower in the order than those they already possess, makes it impossible for deadlocks to occur. Similarly, requiring processes to request all the resources they will ever need at any point during their execution when they start, and either allocating all of these resources or none at all, prevents deadlocks. Both of these methods are not very realistic in large-scale systems with complex applications. In general, deadlock prevention methods lead to low system utilizations.

As a second method for dealing with deadlocks, they can be avoided: processes are at any time allowed to request (and release) resources, but the system runs an algorithm to decide for each request whether it is safe to grant it, that is, whether granting it will lead to a deadlock or not. A famous algorithm for doing so is the Banker's algorithm.

The third method for dealing with deadlocks is deadlock detection: processes are again allowed to do any resource requests, and periodically, or when a process or the (distributed) operating system or middleware suspects a deadlock, an algorithm is run to figure out whether indeed a deadlock has occurred. The only difference with sequential systems is that now the processes reside on different processors. In the algorithms below, a process is either executing or blocked, waiting for a resource. We assume in either case that each process has a thread running that processes the messages of the deadlock-detection algorithm.

### 3.6.1 Models for deadlock

Similarly as in non-distributed systems, a (directed) Wait-For-Graph (WFG) can be maintained with the processes as nodes and with an edge from process $P$ to process $Q$ when $Q$ is holding a resource that $P$ is requesting.

Usually, two models are distinguished in which deadlock may occur. In the *resource model*, processes can gain access and later release resources, where resources are either hardware resources such as memory segments or printers, or software resources such as database records. In this model, we associate a process with each resource. Nodes in the WFG are either "ordinary processes" in the distributed algorithm, or processes associated with resources, and edges in the WFG only run between a process of the former and a process of the latter kind. When a process $P$ wants access to a resource $a$ with associated process $R_a$, an edge from $P$ to $R_a$ is created in the WFG. When the resource is granted, this edge is removed and replaced by an edge from $R_a$ to $P$, indicating that $R_a$ is waiting for $P$ to release resource $a$. When later $P$ actually does so, the edge from $R_a$ to $P$ is removed. It can be shown that there exists a deadlock in the system when the WFG contains a cycle, and then all the "ordinary" processes in cycles on paths leading to cycles are deadlocked.

In the *communication* model, processes can do a blocking RECEIVE operation to a set of processes, indicating that they want to receive a message from one process in this set, which is called its *dependent set*. After such a RECEIVE operation by a process $P$, the WFG contains an edge from $P$ to every process in its dependent set. When $P$ receives a message from any of these processes, all edges from $P$ to these processes are removed. It can be shown that there exists a deadlcok in the system when the WFG contains a *knot*, which is a set of processes with a (directed) path from every process in the set to every other process in the set, and no edges from any process in the set to any process outside the set.

### 3.6.2 Types of Requests

A very general type of resource requests that processes can do are so-called $N$-out-of-$M$ requests. When doing such a request, a process sends a request to $M$ processes that can satisfy the request, and it can proceed as soon as it has received $N$ REPLY messages. It may then send RELINQUISH messages to the processes from which it has not received a REPLY. When $N = 1$, such a request is called an OR-request, and when $N = M$, it is called an AND-request. An example of an $N$-out-of-$M$ request is quorum-based replication, when for instance more than half of the copies of a record in a database have to be locked before a write operation is allowed. The values of $M$ and $N$ may be different in different requests.

### 3.6.3 Deadlock Detection for AND Requests

We will first consider an algorithm for AND requests. In the model used by this algorithm, a process is said to be *dependent* on another process when there is a path in the WFG from the former to the latter process. In order to record its current dependencies, each process maintains a boolean array `dep`, which is initialized to all `falses`; `dep_i(j)` is `true` if $P_i$ knows that $P_j$ is dependent on it.

**Algorithm 3.27** *The deadlock-detection algorithm of Chandy, Misra, and Haas for AND requests* [9].

**Idea:** When a process is blocked and waiting for resources held by other processes, i.e., when it may suspect to be deadlocked, it initiates a *probe* by sending special PROBE messages to the processes it is waiting for ()code fragment I). These PROBE messages are further propagated throughout the system from blocked processes to the processes they are waiting for. When a PROBE message returns in the process that initiated the probe, a cycle in the WFG has been detected, and the initiating process is deadlocked (code fragment II). The PROBE messages are of the form `probe(i,j,k)` with `i` the id of the process that initiated the probe, `j` the id of the process sending the probe message, and `k` the id of the destination process of the probe message.

**Implementation:**

```
I. Initiating a probe
for all $P_j$ for which $P_i$ is waiting do
    send(probe(i,i,j))

II. Receiving a probe message
upon receipt of (probe(j,k,i)) do
    if (($P_i$ is blocked) and (dep_i(j) = false) and ($P_i$ has not replied to all requests of $P_k$)) then
        dep_i(j) ← true
    if (i=j) then $P_i$ is deadlocked
    else
        for all $P_l$ for which $P_i$ is waiting do
            send(probe(j,i,l))

III. When a process turns executing again
if ($P_i$ turns executing) then
    for j=1 to n do dep_i(j) ← false
```

### 3.6.4 Deadlock Detection for OR Requests

We now turn to an algorithm by the same authors as the previous one for another model of deadlocks. In this model, there are no (explicit) resources and resource requests, but every blocked process has a so-called *dependent set* of processes for which it is waiting to receive a message. The reception of a single message from any of the processes in its dependent set will make a process active again, thus modeling OR-requests.

**Algorithm 3.28** *The deadlock-detection algorithm of Chandy, Misra, and Haas for OR requests* [9].

**Idea:** When process $P_i$ is blocked and waiting for a message from other processes, i.e., when it may suspect to be deadlocked, it initiates a *query* by sending special QUERY messages to all processes in its dependent set $D_i$ (code fragment I). These QUERY messages are further propagated throughout the system from blocked processes to the processes in their dependent sets, in effect propagating the QUERY messages down a tree of processes. In response, REPLY messages are sent up the tree.

The QUERY messages are of the form query(i,m,j,k), with i the id of the process that initiated the query, m the sequence number of the query of process $P_i$, j the id of the process sending the message, and k the id of the destination process. A process receiving a query message propagates it to the members of its dependent set if the query contains a sequence number m it has not yet seen for the initiating process (code fragment II). For every QUERY message query(i,m,j,k) sent, at most one REPLY message reply(i,m,k,j) will be returned—process $P_j$ keeps in the variable num(i) the balance of the numbers of these messages. Every process keeps in the variables latest(k) and engager(k) the largest value m it has seen in QUERY messages that were initiated by $P_k$, and the id of the process (the *engager*) that caused it to set largest(k) to m, respectively. REPLY messages are sent back only to engagers. A process only sends a REPLY message when it has received a REPLY message for every QUERY message it has sent (code fragment III). Finally, a process records in wait(k) whether it has been blocked since it last incremented latest(k). A process only reacts to messages with either m > latest(k) or with m=latest(k) and with wait(k) equal to true. The first condition indicates that a process ignores old queries, and the second says that a process only reacts to a message if it relates to the current query number of ptocess k and it has been blocked since it learnt about this query number—if would would have been blocked during that time, it can be shown that $P_k$ was not deadlocked.

A process is deadlocked if and only if for every QUERY message it has received a corresponding REPLY message.

**Implementation:**

I. Initiating a query

```
latest(i) ← latest(i)+1
wait(i) ← true
for all j ∈ D_i do send(query(i,latest(i),i,j))
num(i) ← |D_i|
```

II. Receiving a QUERY message in $P_i$ while blocked

```
upon receipt of query(j,m,k,i) do
    if (m > latest(j)) then
        latest(j) ← m
        engager(j) ← k
        wait(j) ← true
        for all l ∈ D_i do
            send(query(j,m,i,l))
            num(j) ← |D_i|
    else
        if (wait(j) and (m=latest(j))) then
            send(reply(j,m,i,k)
```

III. Receiving a REPLY message

```
upon receipt of reply(j,m,k,i) do
    if ((m=latest(j)) and wait(j)) then
        num(j) ← num(j)-1
        if (num(j)=0) then
            if (j=i) then P_i is deadlocked
            else
                l ← engager(j)
                send(reply(j,m,i,l))
```

IV. When a process turns executing again

```
for all j do wait(j) ← false
```

### 3.6.5 Deadlock Detection for N-out-of-M Requests

In this section we present the deadlock-detection algorithm of [7]. In the model of this algorithm, after a process has done a REQUEST (to $M$ processes), which it can only do when it is *active*, it becomes *blocked*; it can then do no further requests. When it has received $N$ (positive) REPLY messages, it sends a RELINQUISH message to the processes that have not responded, and becomes *active* again. Only active processes can carry our request actions. The channels are assumed to be FIFO. Note that the REQUEST, REPLY, and RELINQUISH messages are messages of the underlying resource-management algorithm of which deadlock is to be detected, but do not belong to the deadlock-detection algorithm proper.

Below we first present the algorithm of [7] for *static* systems with *instantaneous* communication, which means that we have a complete WFG of the system, which we denote by $W$, and that there are no REQUEST, REPLY, or RELINQUISH messages in transit. The edge $(P, Q)$ exists in $W$ if $P$ has sent a REQUEST to $Q$, $P$ has not received a REPLY from $Q$, and $Q$ has not received a RELINQUISH from

$P$. The only possible transformations of $W$ due to actions in the underlying algorithm are:

1. When a process $P$ does an $N$-out-of-$M$ request, it sets its local variable $n$ indicating the number of REPLYs expected to $N$, and $M$ links are added to $W$;

2. When a process $P$ receives a REPLY from process $Q$, the link $(P, Q)$ is deleted from $W$ and $P$ decrements $n$. If then $n = 0$, all outgoing links from $P$ are deleted from $W$.

**Algorithm 3.29** *The deadlock-detection algorithm of Bracha and Toueg for static systems with instantaneous communication* [7].

**Idea:** Every node $P$ maintains two sets of nodes. The set OUT contains the nodes $Q$ such that $(P, Q)$ is in $W$, and the set IN contains the nodes $Q$ such that $(Q, P)$ is in $W$. The algorithm consists of two phases. In the *notify* phase, which can be started spontaneously by any node by invoking the `notify()` procedure, the other nodes get to know that the algorithm has started (code fragment II). When a node then does not have a request pending (`n=0`), it can immediately set its `free` variable to `true`, indicating that it is not deadlocked (code fragment IV). This phase ends only when a process has received a DONE message from every process in its OUT set (code fragments II and III). In the *simulate* phase, which is nested in the *notify* phase, the granting of resources by active processes is simulated. Here, GRANT messages are used to simulate REPLY messages. When a process receives a sufficient number of GRANT messages, it becomes active again (code fragment V). After this phase has terminated, any process that has been notified but remains blocked (i.e., has `notified=true` and `free=false`), is deadlocked.

**Implementation:**

```
I. Initialization in every node P
OUT ← {Q|(P,Q) ∈ W}
IN ← {Q|(Q,P) ∈ W}
notified ← false
free ← false
num_grants ← 0

II. Procedure notify()
notified ← true
for all (Q ∈ OUT) send (NOTIFY) to Q
if (n=0) then grant()
for all (Q ∈ OUT) await (DONE) from Q

III. Reception of a NOTIFY message
upon receipt of (NOTIFY) from Q do
    if (notified=0) then notify()
    send (DONE) to Q

IV. Procedure grant()
free ← true
for all (Q ∈ IN) send (GRANT) to Q
for all (Q ∈ IN) await (ACK) from Q
```

```
V. Reception of a GRANT message
upon receipt of (GRANT) from Q do
    num_grants ← num_grants+1
    if ((free=0) and (num_grants ≥ n)) then grant()
    send (ACK) to Q
```

**Correctness:** See [7]. □

We now turn to the algorithm of [7] for systems without instantaneous communication, which means that there can be messages in transit in $W$. In this case, for each link $(P, Q)$, a *color* is defined, with the following meaning:

- *grey*, if $P$ has sent a REQUEST to $Q$ which has not yet been received, and $P$ has not sent a RELINQUISH to $Q$;
- *black*, if $Q$ has received a REQUEST from $P$ but has not yet replied, and $P$ has not sent a RELINQUISH to $Q$;
- *white*, if $Q$ has sent a REPLY to $P$ which has not yet been received, and $P$ has not sent a RELINQUISH to $Q$;
- *translucent*, if $P$ has sent a RELINQUISH to $Q$ which $Q$ has not yet received.

Now the following transformations of $W$ are possible due to actions in the underlying algorithm:

1. When a process $P$ does an $N$-out-of-$M$ request, it sets its local variable $n$ indicating the number of REPLYs expected to $N$, and $M$ grey links are added to $W$;
2. When a process receives a REQUEST, the color of the corresponding link is turned from grey into black;
3. When a process sends a REPLY, the color of the corresponding link is turned from black into white;
4. When a process receives a REPLY, it removes the corresponding white edge and decrements $n$. If then $n = 0$, all its outgoing edges are made translucent and RELINQUISH messages are sent along these translucent edges;
5. When a process receives a RELINQUISH message, it removes the translucent edge over which it received this message.

**Algorithm 3.30** *The deadlock-detection algorithm of Bracha and Toueg for static systems without instantaneous communication* [7]

**Idea:** When a link has the color grey, white, or translucent, within finite time it will not have that color anymore as the message on the link inducing the color will be received. Therefore, Algorithm 3.29 is applied to the graph obtained from $W$ by only retaining the black edges. Correspondingly, we assume (possibly falsely) that along grey and white edges a REPLY will be sent. This may not be true, and a deadlock that actually exists may not be detected, but in a later run such a deadlock will be detected (for instance, when a grey link has turned black). We assume that every node knows the colors of the edges in its IN and OUT sets.

**Implementation:** The implementation of this algorithm is as the implementation of Algorithm 3.29 with the following modifications:

- The sets IN and OUT only consist of the black edges in $W$, the other edges are ignored;
- Instead of the number $n$, the number $n$ minus the numbers of grey and white edges in the OUT sets are used;
- We can assume that the nodes know the colors of the edges they are incident upon by having them exchange additional COLOR message. Every process sends a COLOR message to all processes in its IN and OUT sets, making those processes aware of their membership of these sets. Then an edge $(P, Q)$ is grey or white (this is needed in the algorithm) when $Q \in \text{OUT}_P$ and $P \notin \text{IN}_Q$, which means either that a REQUEST message of $P$ to $Q$ has been sent but not yet received (so the link has the color grey), or that a REQUEST message of $P$ to $Q$ has been received, that $Q$ has replied (and removed $P$ from $\text{IN}_Q$), and that the REPLY message has not yet been received by $P$ (so the link has the color white).

**Correctness:** See [7]. □

## 3.7 Bibliographic Notes

The happened-before relation and scalar logical clocks were introduced in [32]. In [62], an extension of logical clocks is presented. The proof of Theorem 3.9 follows the proof in [2] closely. Algorithm 3.17 is adapted from [52]. An extensive survey of totally ordering messages appeared in [16].

Early overviews of deadlock detection in distributed systems can be found in [30] and [56]. Our treatment of Algorithm 3.27 differs from the original paper [9] in that we do not assume local controllers in every process that keep track of the dependencies among the local processes.

## 3.8 Exercises

1. Consider Example 3.2 and Figure 3.1.
   (a) Trace all pairs of events that are related according to the HB relation.
   (b) Trace all pairs of concurrent events.
   (c) Assign Lamport timestamps to all events.
   (d) Assign vector timestamps to all events.
2. Is the concurrency relation transitive? If so, prove this. If not, show a counter example.
3. Prove that
   - if a is not a message-receive event and if b is the event immediately preceding a in the process in which a occurs, then $P(\mathtt{a}) = P(\mathtt{b}) \cup \{\mathtt{b}\}$.
   - if a is the event of receiving the message sent in event b and if c is the event immediately preceding a in the process in which a occurs, then $P(\mathtt{a}) = P(\mathtt{b}) \cup P(\mathtt{c}) \cup \{\mathtt{b}, \mathtt{c}\}$.

4. Prove that for two different events $\mathtt{a}$ and $\mathtt{b}$:
    - $\mathtt{a} \rightarrow \mathtt{b}$ iff $\mathtt{a} \in \mathtt{P}(\mathtt{b})$
    - $\mathtt{a}||\mathtt{b}$ iff $\mathtt{a} \notin \mathtt{P}(\mathtt{b})$ and $\mathtt{b} \notin \mathtt{P}(\mathtt{a})$.

5. Prove that for a vector clock $V$ and an event $\mathtt{a}$,

$$V(\mathtt{a})[k] = \max\{V(\mathtt{b})[k] \mid \mathtt{b} \in P(\mathtt{a}) \cap E_k\}.$$

6. Prove that for $\mathtt{a} \in E_i$ and $\mathtt{b} \in E_j$,
    - $\mathtt{a} \rightarrow \mathtt{b}$ if and only if $V(\mathtt{a})[i] \leq V(\mathtt{b})[i]$
    - $\mathtt{a}||\mathtt{b}$ if and only if $V(\mathtt{a})[i] > V(\mathtt{b})[i]$ and $V(\mathtt{b})[j] > V(\mathtt{a})[j]$.

7. Prove that if for two events $\mathtt{a}$ and $\mathtt{b}$, $V(\mathtt{a}) < V(\mathtt{b})$, then $\mathtt{a} \rightarrow \mathtt{b}$ (cf. Theorem 3.8).

8. Show an example of message exchanges in an asynchronous system in which causal order is obeyed but total order is not, and vice versa.

9. Trace the execution of Algorithm 3.15 for causal broadcast in Example 3.13. In particular, what are vector timestamps sent along with each of the two messages $\mathtt{m}_1$ and $\mathtt{m}_2$, and what is the condition for delivering message $\mathtt{m}_2$ in $P_3$?

10. Prove the liveness of Algorithm 3.15.

11. Trace the execution of Algorithm 3.17 for causal order of point-to-point messages in Example 3.16. In particular, what are the message buffers and vector timestamps sent along with each of the three messages $\mathtt{m}_1$, $\mathtt{m}_2$, and $\mathtt{m}_3$, and what are the conditions checked for delivering the messages?

12. Does Algorithm 3.18 for total order of messages enforce causal order?

13. Give an example that shows that Chandy's and Lamport's algorithm for determining global states can give an incorrect result when the FIFO condition for channels is lifted.

14. Show how the order of the four events of sending and receiving amounts of money in Example 3.24 of the Chandy-Lamport algorithm for determining global states can be modified so that the recorded state has occurred in the system.

15. Consider Algorithm 3.26.
    (a) Show that the FIFO property of the network links is indeed needed.
    (b) In the algorithm, we have assumed asynchronous communication. Is this assumption sufficient when the communication along the ring is only used for forwarding the token, but when the processes can communicate directly?

16. Trace the execution of Algorithm 3.29 in case there is a cycle in the WFG.

# Chapter 4

# Coordination

Two of the most important characteristics of distributed systems mentioned in the introduction of this book are non-determinism and the lack of a common global state. However, cooperating processors sometimes have to coordinate their actions, so they have to overcome the non-determinism, and in a distributed system they have to do so by means of messages. In this chapter, we deal with three coordination problems, viz. mutual exclusion, election, and creating special routing tables in weighted networks, in particular, routing tables that implement shortest paths and routing tables that implement minimum-weight spanning trees.

## 4.1 Mutual Exclusion in Distributed Systems

Mutual exclusion has played an important role in the development of algorithms in central systems, expecially in operating systems. The need for mutual exclusion arises when a resource can only be accessed by one process at a time. Such resources can be hardware components such as printers, or software components such as data structures stored in memory. To access the resource, a process executes a critical section (CS), and so the problem translates into guaranteeing at most one process to be in its CS at a time. Also in distributed systems, mutual-exlusion algorithms have received much attention, both as an interesting theoretical problem for distributed algorithms and for practical applications. In distributed systems, the problem can be posed in the following way:

- There are $n$ processes $P_i, i = 0, \ldots, n-1$, running on multiple processors in a connected network;
- Each process $P_i$ has a CS, which takes a finite amount of time to execute;
- At most one of the $P_i$ is allowed to be executing its CS at the same time (mutual exclusion);
- When a process $P_i$ requests entry to its CS, it is guaranteed to enter it within finite time (no starvation or deadlock).

An obvious solution in distributed systems is a central solution: Assign a single node the task of granting the processes exclusive access to their CSs. However, this solution causes the access-granting process to be a single point of failure and a potential performance bottleneck. In addition, it is not an elegant solution. One rather aims at truly distributed solutions in which each process plays a similar role.

Mutual-exclusion algorithms in distributed systems are divided into token-based and assertion-based algorithms. In *token-based* algorithms there is a single distinguished message, the *token*, the possession of which allows a process to execute its CS. In such algorithms, mutual exclusion is trivially guaranteed, and the main issues are the prevention of starvation and of deadlock. In *assertion-based* or non-token-based algorithms, a process has to request permission from all or part of the other processes, and based on their replies, it may conclude that it is the only one with the right to access its CS. For more on the distinction between these two kinds of mutual-exclusion algorithms, see [46].

In order to prove a mutual-exlusion algorithm correct, we have to prove its safety and its liveness.

### 4.1.1 Assertion-Based Mutual-Exclusion Algorithms

We start with three assertion-based algorithms. The first may be the first such algorithm ever published. The second and especially the third improve on its message complexity.

**Algorithm 4.1** *Lamport's mutual-exclusion algorithm* [32].

**Idea:** In this algorithm, all links have the FIFO property, and all messages are timestamped with a pair consisting of a scalar logical time and the id of the sending processor. A process wishing to enter its CS broadcasts a timestamped REQUEST message to all processes, including itself. When a process receives a REQUEST message, it enters the request into its queue Q of requests ordered according to timestamp, and sends back a REPLY message. A process is allowed to enter its CS when it has received a REPLY message from every process and when its own request is at the head of its request queue. When a process leaves its CS, it sends a RELEASE message to all processes, which then remove the request from their request queues.

**Implementation:**

```
I. Broadcasting a REQUEST message
no_replies ← 0
T ← current timestamp
broadcast(request;T,i)

II. Receiving a REQUEST message
upon receipt of (request;T,j) do
    enqueue(Q,(T,j))
    send(reply) to $P_j$

III. Receiving a REPLY message
upon receipt of (reply) do
    no_replies ← no_replies + 1
    Conditional_CS

IV. Receiving a RELEASE message
upon receipt of (release) do
    Q ← tail(Q)
    Conditional_CS

V. Procedure Conditional_CS
if ((no_replies=n) and (head(Q)=(*,i)))
then
    Critical Section
    broadcast(release)
```

**Correctness:** In order to prove the safety of Algorithm 4.1, we prove that if $\mathtt{a}_i$ and $\mathtt{a}_j$ are the events of requesting access to the CSs in two different processes $P_i$ and $P_j$, respectively, with either $\mathtt{a}_i \rightarrow \mathtt{a}_j$ or $\mathtt{a}_i || \mathtt{a}_j$ and $i < j$, then $P_j$ only gets access to its CS after $P_i$ has broadcast the RELEASE message corresponding to $\mathtt{a}_i$. Let $\mathtt{b}_j$, $\mathtt{b}_i$, $\mathtt{c}_i$, $\mathtt{c}_j$ be the events of receiving $P_i$'s CS request in $P_j$, of receiving $P_j$'s CS request in $P_i$, of sending a reply to $P_j$'s request by $P_i$, and of receiving this reply by $P_j$, respectively. (The reader may want to draw a picture here.) Then we have $\mathtt{a}_i \rightarrow \mathtt{b}_i$ (otherwise $\mathtt{a}_j \rightarrow \mathtt{a}_i$), and of course $\mathtt{b}_i \rightarrow \mathtt{c}_i$, and so, by the transitivity of the HB relation, $\mathtt{a}_i \rightarrow \mathtt{c}_i$. Then by the FIFO property of the links, we also have $\mathtt{b}_j \rightarrow \mathtt{c}_j$. As a result, $P_j$ enqueues $P_i$'s request before is has received $n$ replies to its own request, and so it has to await $P_i$'s RELEASE message before it can enter its CS.

As to the liveness of Algorithm 4.1, if at any point in the execution there are CS requests, one of them is the oldest (in the lexicographic ordering) in the system. The process that generated this request will in due course receive $n$ replies, and will have its own request at the head of its request queue, so it can enter its CS.

**Complexity:** Obviously, for each CS invocation, $n-1$ REQUEST messages, $n-1$ REPLY messages, and $n-1$ RELEASE messages are used (not counting messages sent by a process to itself), for a total of $3(n-1)$ messages. □

In Algorithm 4.1, when some process $P_i$ receives a CS request from $P_j$ while it has an older CS request itself, it first sends a REPLY message to $P_j$, and later, after it has finished its CS, it sends a RELEASE message to $P_j$. These two messages can be combined into a single one, which is the idea of the following algorithm.

**Algorithm 4.2** *Ricart's and Agrawala's mutual-exclusion algorithm* [49].

**Idea:** The idea is similar to that of Algorithm 4.1, but now a process *defers* sending a REPLY message to a request if it is currently having a request of its own that is older, until its own request has been satisfied. If a process does not have such a request, it still sends a REPLY message immediately. RELEASE messages are not needed anymore. The implementation of the algorithm and the question whether the links have to be FIFO are left to the reader as Exercises 2 and 3.

**Complexity:** In this algorithm, $n-1$ REQUEST messages and $n-1$ REPLY messages are involved in the execution of a CS, for a total of $2(n-1)$ messages. □

In Algorithms 4.1 and 4.2, a process sends its REQUEST messages to every other process, leading to a message complexity of order $n$. In order to reduce the message complexity, one may try to reduce the size of the *request set* of processes to which a process sends its requests and from whom it needs permission to enter its CS. Of course, to guarantee mutual exclusion, we then need the request sets of two processes to have a non-empty intersection. Denoting the request set of $P_i$ by $R_i, i = 0, 1, \ldots, n-1$, we require, for $i, j = 0, 1, \ldots, n-1$:

1. Every two request sets have a non-empty intersection: $R_i \cap R_j \neq \emptyset$.

In addition, the following properties are desirable for $i, j = 0, 1, \ldots, n-1$:

2. Every process is contained in its own request set: $i \in R_i$;
3. Every request set has the same number of elements, so $|R_i| = K$ for some positive integer $K$;
4. Every process appears the same number of times in a request set, so $i \in R_j$ for $D$ values of $j$, for some positive integer $D$.

In order to reduce the message complexity as much as possible, the objective is to choose the sets $R_i$ in such a way that $K$ is minimal.

**Lemma 4.3** *For a set of request sets $R_i$ satisfying the properties 1.–4. above, $K$ is at least of order $O(\sqrt{n})$.*

PROOF. Consider some $R_i$. Any other request set $R_j$ has a nonzero intersection with $R_i$, and each element of $R_i$ is contained in $D - 1$ other request sets, so the number $n$ of request sets $R_i$ satisfies

$$n \leq K(D - 1) + 1.$$

Since there are $n$ request sets, each request set has $K$ elements, and each element appears in $D$ request sets, we have $nK/D = n$, or $K = D$. We conclude that

$$n \leq K(K - 1) + 1,$$

which proves the lemma. □

It can be shown that for every positive power $l$ of every prime number $p$, if $n = p^{2l} + p^l + 1$, one can construct request sets $R_i$ that satisfy the conditions above with $D = K = p^l + 1$. This construction involves finite projective planes, the explanation of which is beyond the scope of this book, and for which we refer the reader to [50].

**Example 4.4** We give a possibility for the request sets for $n = 7$ and $K = 3$ in Table 4.1. □

Table 4.1: An example for the request sets with $n = 7$ and $K = 3$.

| node $i$ | request set $R_i$ |
|---|---|
| 0 | $\{0,1,2\}$ |
| 1 | $\{1,4,6\}$ |
| 2 | $\{2,3,4\}$ |
| 3 | $\{0,3,6\}$ |
| 4 | $\{0,4,5\}$ |
| 5 | $\{1,3,5\}$ |
| 6 | $\{2,5,6\}$ |

**Example 4.5** When $n = m^2$, and the processors are interconnected by an $m \times m$ two-dimensional grid, we can take for the request set of a node the set of all nodes in the same row and the same column. In this case, the request sets contain $2m - 1$ nodes, which is of order $\sqrt{n}$. □

We now present a mutual-exclusion algorithm which employs these request sets. The basic operation of the algorithm is very simple, but there is the potential for deadlock, which is more difficult to handle.

TBA: example of potential deadlock.

**Algorithm 4.6** *Maekawa's mutual-exclusion algorithm* [39].

**Idea:** Every process has a request set R, and the intersection of any two request sets is non-empty. When a process wants to enter its CS, it multicasts a timestamped REQUEST message to the members of R. After a process has received a GRANT from all processes in R, it enters its CS, and after finishing its CS, it sends a RELEASE message to all processes in R.

In order to avoid deadlock, when a process receives a REQUEST message, it replies with a GRANT when it has not sent a GRANT to another process without having received the corresponding RELEASE message. When a process receives a REQUEST message while it has already granted permission to another process, it compares the timestamps of the two requests. When the timestamp of the new request is the later of the two, it queues the new request in a queue Q and sends the requesting process a POSTPONE message; otherwise it inquires with the process to whom it has sent a GRANT message. When a process receives an INQUIRE message, it waits until either it has obtained a GRANT from every process in R, or until it has received a POSTPONED message. In the former case, it completes its CS and replies with a RELEASE message, in the latter case, it gives the permission back with a RELINQUISH message. Upon receiving a RELINQUISH message, a process enqueues the corresponding CS request, and sends a GRANT message to the process with the oldest request it knows of.

**Implementation:**

I. Multicasting a REQUEST message

```
no_grants ← 0
T ← current timestamp
for all $j \in$ R do
    send(request;T,i) to $P_j$
```

II. Receiving a REQUEST message

```
upon receipt of (request;T,j) do
    if (¬granted) then
        current_grant ← (T,j)
        send(grant) to $P_j$
        granted ← true
    else
        insert(Q,(T,j))
        (V,k) ← head(Q)
        if (current_grant < (T,j)) or ((V,k) < (T,j)) then
            send(postponed) to $P_j$
        else
            if (¬inquiring) then
                inquiring ← true
                l ← current_grant.node
                send(inquire;i) to $P_l$
```

III. Receiving a GRANT message

```
upon receipt of (grant) do
    no_grants ← no_grants+1
    if (no_grants = |R|) then
        postponed ← false
        Critical Section
        for all $j \in$ R do
            send (release) to $P_j$
```

IV. Receiving an INQUIRE message

```
upon receipt of (inquire; j) do
    wait until ((postponed) or (no_grants = |R|))
    if (postponed) then
        no_grants ← no_grants-1
        send (relinquish) to $P_j$
```

V. Receiving a RELINQUISH message

```
upon receipt of (relinquish) do
    inquiring ← false
    granted ← false
    insert(Q,current_grant)
    current_grant ← head(Q)
    dequeue(Q)
    j ← current_grant.node
    send (granted) to $P_j$
    granted ← true
```

VI. Receiving a RELEASE message

```
upon receipt of (release) do
    granted ← false
    inquiring ← false
    if (not_empty(Q)) then
        current_grant ← head(Q)
        dequeue(Q)
        j ← current_grant.node
        send (grant) to $P_j$
        granted ← true
```

VII. Receiving a POSTPONED message

```
upon receipt of (postponed) do
    postponed ← true
```

**Correctness:** As to the safety of the algorithm, no two processes can be in their CSs at the same time, because then the processes in the intersection of their request sets would have sent them both a GRANT message. For the proof of the absence of deadlock and starvation, see [39].

**Complexity:** If there is no contention, one CS request takes $K-1$ REQUEST messages, $K-1$ GRANT messages, and $K-1$ RELEASE messages, for a total of $3(K-1)$ messages. If there is high contention, $4(K-1)$ messages per entry can be expected: an additional number of $K-1$ POSTPONED messages may then have to be sent before the GRANT messages. When a process that has not requested access to its CS and has not participated in the algorithm for a certain amount of time does initiate a request, it may be expected to have the oldest request in the system. Then each of the $K-1$ REQUEST messages may trigger an INQUIRE message, which may in turn cause a RELINQUISH message before a GRANT can be sent and later a RELEASE be returned. So then a total of $5(K-1)$ messages are needed for a single access to a CS. □

In Algorithm 4.2 on the one hand and in Algorithm 4.6 on the other, processes that receive a request send REPLY or GRANT messages. However, these messages have very different meanings in these algorithms. The meaning of the REPLY messages in Algorithm 4.2 is that the receiving process can enter its CS as far as the process sending the REPLY is concerned. In particular, a process can send multiple REPLY messages. However, the meaning of a GRANT message in Algorithm 4.6 is that the sending process gives the process to which it sends a GRANT exclusive access, and will only send a GRANT to another process after it has received a RELEASE (or a RELINQUISH) from the first. We will now present a generalized mutual-exclusion algorithm that includes Algorithms 4.2 and 4.6 as special cases.

**Algorithm 4.7** *A generalized mutual-exclusion algorithm* [51].

**Idea:** Every process $P_i$ has a *request set* $R_i$ and an *inform set* $I_i$. In addition, for every process $P_i$, a *status set* $S_i$ is defined by $j \in S_i$ iff $i \in I_j$. It is assumed that $i \in I_i$ (and, as a consequence, $i \in S_i$) for every $i$. The requirements on these sets for a correct algorithm are

1. $I_i \subset R_i$, for $i = 1, 2, \ldots, n$;

2. for $i, j = 1, 2, \ldots, n$, either $I_i \cap I_j \neq \emptyset$ or $i \in R_j$ and $j \in R_i$.

See Exercises 4 and 5 for how to set these sets in Algorithms 4.2 and 4.6.

When a process $P_i$ wants to access its CS, it sends a REQUEST message to all processes in $R_i$, and it awaits a GRANT from all these processes before it enters its CS. When a process $P_i$ leaves its CS, it sends a RELEASE message only to the processes in its inform set $I_i$. Every process $P_i$ maintains a variable p_in_CS, which is the index of the process in its status set $S_i$, if any, to which it has sent a GRANT message without having received the corresponding RELEASE. Otherwise, p_in_CS is NULL. Upon reception in $P_i$ of a REQUEST from some process $P_j$, $P_i$ will return a GRANT as long as p_in_CS is NULL. If $j \in S_i$, $P_i$ will then set p_in_CS to $j$. All messages are timestamped with scalar logical clocks and process ids as tie breakers, and requests that cannot be granted when they are received, are entered into an ordered queue. Upon reception in $P_i$ of a RELEASE message from some process $P_j$ (from a process in its status set), $P_i$ will reset p_in_CS. It will then go through its local queue of pending requests and send GRANTs and remove the requests from the queue as long as p_in_CS is NULL and the queue is not empty. When it encounters a request from some process $P_j$ in $S_i$, it sends $P_j$ a GRANT and sets p_in_CS to $j$.

Similarly as in Algorithm 4.6, deadlocks may have to be handled.

**Correctness:** See [51]. □

### 4.1.2 Token-Based Algorithms

We now present three token-based mutual-exclusion algorithms. In such algorithms, mutual exclusion is trivially satisfied because there is only a single token the possession of which gives a process the privilege to entre its CS. So the correctness proof of these algorithms only deals with liveness.

In the first of the three algorithms below every process notifies every other process of its requests to enter its CS. The second algorithm tries to cut back on the number of messages by having a process only send its requests for entry to the subset of processes it expects may have the token; however, the worst-case message complexity is then not improved upon. The third algorithm does achieve an improvement of the message complexity by "pooling" requests so that processes may announce requests for entry from multiple processes by means of a single message.

**Algorithm 4.8** *Suzuki's and Kasami's broadcast-based mutual-exclusion algorithm* [59].

**Idea:** There is a single token that circulates among the processes, the presence of which in a process signals permission to access its CS. When wishing to enter its CS, a process sends a request with a sequence number to all other processes (including itself). Every process maintains an array `N`, initialized to all zeroes, with for every process the sequence number of the last request it knows about. In addition, in the token an array `TN` is maintained with for every process the sequence number of the last request that was granted. By comparing corresponding elements in `N` and `TN`, processes decide to which process to forward the token. In order to avoid starvation, a process starts looking for another process to send the token to at its own process index. Note that it is possible for a process to receive a request that has already been granted.

**Implementation:**

I. Broadcasting a REQUEST message

```
N[i] ← N[i] + 1
broadcast(request;i,N[i])
```

II. Receiving a REQUEST message

```
upon receipt of (request;j,r) do
    N[j] ← r
    if ((token_present) and (not in CS)) and (N[j]>TN[j]) then
        token_present ← false
        send(token;TN) to $P_j$
```

III. Receiving the token

```
upon receipt of (token;TN) do
    token_present ← true
    Critical Section
TN[i] ← N[i]
for j=i+1 to n, 1 to i-1 do
    if (N[j] > TN[j]) then
        token_present ← false
        send(token;TN) to $P_j$
        break
```

**Correctness:** Because a CS request eventually reaches all processes and because processes search in a circular way among the processes when deciding where to send the token, every request will eventually be satisfied.

**Complexity:** Algorithm 4.8 achieves mutual exclusion with $n-1$ messages and one token transfer for every CS invocation. □

As a matter of fact, in the algorithm as presented in [59], there is an additional data structure, a queue, in the token. Whenever the token is about to be sent by a process, the processes for which the local array `N` and the array `TN` indicate that they have a new CS request are appended to this queue, if they are not already contained in it. Then a process finishing its CS sends the token to the process at the head of the queue if this is not empty, and the receiving process removes itself from the queue.

This queue may speed up the algorithm. For instance, in the algorithm as presented above, if process $P_i$ sends the token to $P_j$ after it has received a new request from $P_k$, and if it takes a long time for $P_k$'s request to reach $P_j$, the token may remain in $P_j$ for a long time. When the queue is used, knowledge of $P_k$'s request is transferred to $P_j$, which can then send the token to $P_k$, even when it has not received $P_k$'s request yet.

In Algorithm 4.8, a process wishing to enter its CS sends a request to all other processes. Similarly as with assertion-based algorithms, we may try to reduce the number of processes to which a request is sent. In this case, the request has to be sent to the subset of processes that may possess the token, or that may receive it in due course.

**Algorithm 4.9** *Singhal's multicast-based mutual-exclusion algorithm* [55].

**Idea:** Processes only send CS requests to the subset of processes who they think may possess the token. A process may be in one of four states: requesting the token (`R`), executing its critical section (`E`), not executing its critical section and holding the token while not aware of any other process wanting to access its critical section (`H`), and other (`O`). Every process maintains an array `N` of integers counting the number of requests of each process, initialized to all zeroes, and an array `S` of the states of all processes, which in process $P_0$ is initialized as

$$\begin{aligned} &\mathtt{S[0]} = \mathtt{H}, \\ &\mathtt{S[j]} = \mathtt{O}, j = 1, \ldots, n-1, \end{aligned}$$

and in process $P_i, i = 1, \ldots, n-1$ as

$$\begin{aligned} &\mathtt{S[j]} = \mathtt{R}, j = 0, \ldots, i-1, \\ &\mathtt{S[j]} = \mathtt{O}, j = i, \ldots, n-1. \end{aligned}$$

The meaning of these initializations is that process $P_i$ thinks that the token is in one of the processes $P_0, P_1, \ldots, P_{i-1}$, for $i = 1, 2, \ldots, n-1$. The token contains an array `TN` with the same function as in Algorithm 4.8, and an array `TS` in which knowledge on the states of processes is transferred. In code fragment II. below, when process $P_i$ which is requesting entry to its CS (indicated by `S[i]=R`) receives a `REQUEST` from a process $P_j$, it sends $P_j$ a `REQUEST` message because $P_j$ may receive the token in the future. Similarly as in Algorithm 4.8, fairness can be introduced by having the processes start at their own index when looking for a process to send the token to in code fragment III. below.

**Implementation:**

```
I. Requesting access to the CS
if (S[i]=H) then
    send(request;i,N[i]) to $P_i$
S[i] ← R; N[i] ← N[i]+1
for j=0 to i-1, i+1 to n-1 do
    if (S[j]=R) then
        send(request;i,N[i]) to $P_j$

II. Receiving a REQUEST message
upon receipt of (request;j,r) do
    N[j] ← r
    case S[i] of
        E,O: S[j] ← R
        R: if (S[j] ≠ R) then
             S[j] ← R
             send(request;i,N[i]) to $P_j$
        H: S[j] ← R; S[i] ← O
           TS[j] ← R; TN[j] ← r
           send(token) to $P_j$
```

```
III. Receiving the token
upon receipt of (token) do
    S[i] ← E
    Critical Section
S[i] ← O; TS[i] ← O
for j:=0 to n-1 do
    if (N[j] > TN[j]) then
        TN[j] ← N[j]; TS[j] ← S[j]
    else
        N[j] ← TN[j]; S[j] ← TS[j]
if ($\wedge_{j=0}^{n-1}$ (S[j]=O)) then S[i] ← H
else send(token) to some $P_j$ with S[j]=R
```

**Correctness:** A proof of the correctness of Algorithm 4.9 can be found in [55]. One can show that if at some point during the execution of the algorithm the system is in a quiescent state in the sense that no process is in its CS or is requesting its CS, the system is, up to a permutation of the processes, in the initial state with respect to the values in the S arrays. A subtle point arises when a process exits its CS in the for-loop in code fragment III, and the process and the token have the same values for the request numbers of some process (N[j]=TN[j]). Exercise 6 deals with this issue.

**Complexity:** When there is a very low contention for entry to the CSs and so the system is often in a state identical to the initial state (up to a permutation of the processes), on average, a CS execution takes $n/2$ REQUEST messages and one token transfer. However, when there is a high contention with almost every process having a request for its CS outstanding, the message complexity approaches $n$. □

We now consider an example of the operation of Algorithm 4.9.

**Example 4.10** Let's trace the execution of Algorithm 4.9 in a system with three processes. In Table 4.2 we show the values of the state vectors in the processes and in the token in five consecutive states. State 1 is the initial state. Then $P_2$ generates a request, which it sends to both $P_0$ and $P_1$. As a response, $P_0$ sends the token with TS=(O,O,R) to $P_2$, which results in state 2. After $P_2$ has finished its CS, the system is in state 3. If then $P_1$ generates a request, it sends it to both $P_0$ and $P_2$, and $P_2$ sends the token with TS=(O,R,O) to $P_1$, resulting in state 4. After $P_1$ has finished its CS, the system is in state 5. States 1 and 5 are, up to a permutation of the processes, identical. □

Finally we turn to an algorithm that spreads CS requests through a spanning tree.

**Algorithm 4.11** *Raymond's token-based mutual-exclusion algorithm in a tree* [45].

Table 4.2: Five successive states in the execution of Algorithm 4.9 in Example 4.10.

| Process: | | $P_0$ | $P_1$ | $P_2$ |
|---|---|---|---|---|
| State: | | | | |
| 1 | S | (H,O,O) | (R,O,O) | (R,R,O) |
| | TS | (O,O,O) | | |
| 2 | S | (O,O,R) | (R,O,R) | (R,R,R) |
| | TS | | | (O,O,R) |
| 3 | S | (O,O,R) | (R,O,R) | (O,O,H) |
| | TS | | | (O,O,O) |
| 4 | S | (O,R,R) | (R,R,R) | (O,R,O) |
| | TS | | (O,R,O) | |
| 5 | S | (O,R,R) | (O,H,O) | (O,R,O) |
| | TS | | (O,O,O) | |

**Idea:** This algorithm assumes that a spanning tree among the processes in the system has been created ahead of time. The root of this tree may change over time: the process that holds the token and can access its CS is the current root of the tree. At any time, every process has a notion of its parent, which is its neighbor that is the first process on the path to the current root. When the algorithm starts, one process is designated as the root and has the token, and it has to notify all other processes of the structure of the tree through a wave of outgoing messages.

Every process maintains a queue with the ids of those of its neighbor processes in whose subtree there are pending CS requests as far as they have been made aware of. A process that wants to access its CS appends itself to its local queue and sends a REQUEST message to its parent. A process that receives a REQUEST message appends the id of the process from which it receives the message to its local queue, and if it hasn't previously already done so, sends a REQUEST message with its *own* id to its parent, thus making a single REQUEST message represent the wish of multiple processes to enter their CS. When the current root is done with its CS, it sends the token to its neighbor that is at the head of its queue, removes it from its queue, and assigns it to be its parent, thus modifying the tree structure. When a process receives the token, it enters its CS when it is at the head of its own queue; otherwise it propagates the token to the process at the head of its local queue, removes it from its queue, and makes that process its parent.

**Implementation:**

The implementation is left as an exercise. It requires four pieces of code as specified below.

I. Requesting access to the CS

II. Receiving a REQUEST message

III. Receiving the token

IV. Leaving the CS

**Correctness:** A proof that this algorithm is deadlock and starvation-free is in [45].

**Complexity:** The worst-case message complexity of this algorithm is $2(N-1)$. In general, when

$D$ is the diameter of the spanning tree, in the worst case a REQUEST message and the corresponding message with the token have to travel the complete distance between the requesting and the token-holding process, which can be a distance $D$ apart. When the tree is in fact linear, $D = N - 1$. □

We now consider an example of the operation of Algorithm 4.11.

**Example 4.12** To be provided. □

## 4.2 Detection of Loss and Regeneration of a Token

Some distributed algorithms, such as Algorithm 3.25 for termination detection and Algorithms 4.8 and 4.9 for mutual exclusion, require the use of a token, a special message which circulates in the system. Often, the correctness of an algorithm critically depends on the presence of exactly one token. When the token gets lost, for instance because of an unreliable line, a single new token has to be generated. In this section we present a distributed algorithm for the detection of the loss and the subsequent regeneration of a token in a unidirectional ring.

**Algorithm 4.13** *Detection of the loss and the regeneration of a token.*

**Idea:** We suppose that in the distributed algorithm which uses the token, the token, say $t_0$, circles around the ring. The solution involves another token $t_1$, by means of which the loss of $t_0$ can be detected. The tokens $t_0$ and $t_1$ play a symmetrical role: one detects the loss of the other. The message carrying a token contains the token number (0 or 1), and plus (in $t_0$) or minus (in $t_1$) the number of times they have been in a process at the same time. Tokens are represented by messages of type `token`, a token number (`j`) and a counter (`c`) which is equal to plus (token $t_0$ with `j=0`) or minus (token $t_1$ with `j=1`) the number of times the tokens have met (i.e., were present in the same node at the same time). Whenever the tokens meet, the latter two numbers are incremented and decremented, respectively. Also, every node records (in `l`) the counter of the token which arrived at it last. When a token, say $t_0$, arrives at a node, the condition `l = c` signals the loss of $t_1$. Then $t_1$ is regenerated by setting `token_present[1]` to 1, and the counters are in/decremented, as if the tokens met.

In the implementation below, `C(i,j)` is some condition in $P_i$ which allows token `j` to leave $P_i$. We suppose that when token `j` is in $P_i$, `C(i,j)` becomes true in finite time. When token $t_0$ passes through a node, say $P_i$, for the second time in a row while in the mean time $t_1$ has not passed through $P_i$ and $t_0$ and $t_1$ have not met in any $P_j$, $j \neq i$, $t_1$ has been lost. The links in the ring are assumed to be FIFO.

**Implementation:**

I. Receiving a token

```
upon receipt of (token;j,c) do
    token_present[j] ← true
    m[j] ← c
    if (token_present[1-j] or (l = c)) then
        m[j] ← c + sign(c)
        m[1-j] ← -m[j]
        token_present[1-j] ← true
```

II. Sending a token

```
when (token_present[j] and C(i,j)) do
    token_present[j] ← false
    send(token;j,m[j])
    l ← m[j]
```

**Correctness:** Suppose a token, say $t_0$, arrives at $P_i$ and finds `l = c`. Because of the sign of `l`, $t_1$ did not pass through $P_i$ after $t_0$ visited $P_i$ for the last time, and because the value of `c` has not changed during $t_0$'s last complete round, it did not meet $t_1$. Because of the FIFO-property of the links, $t_1$ is lost. Furthermore, when $t_1$ is lost and $t_0$ does not get lost, $t_0$ will keep on traveling around the ring, so in due course it will find out that $t_1$ is lost. Finally, a token will not be regenerated twice (before getting lost again), for if a token is regenerated in say $P_i$, both `m[0]` and `m[1]` in $P_i$ are set to values these variables never had before in any process, and consequently, the value of `l` in any other process cannot be equal to the value of `m[0]` or the value of `m[1]` in $P_i$. □

## 4.3 Election in Distributed Systems

An important aim in the design of distributed algorithms and distributed systems is to achieve truly distributed solutions. The processes executing a distributed algorithm should be similar, each having about the same level of responsibility and power to contribute to decisions. Central components, both in hardware and in software, should be avoided. Sometimes, however, the nature of an application may dictate that one processor be endowed with a special privilege. It may be necessary to assign this privilege dynamically, and all processors should then cooperate to *elect* one from among them to get this privilege. Algorithms that achieve this goal are called *election* algorithms. The requirement for such algorithms is that within finite time, exactly one processor is elected to be the leader and is aware of this fact. In addition, one may require that all other processors get to know that they have not been elected, and possibly who the winner is, but this is easily achieved by having the winner broadcast the news of its victory.

To model the election problem, it is often assumed that each processor has a unique integer processor id. If this is the case, the system is said to be *non-anonymous*, and *anonymous* otherwise. In the non-anynomous case, the processor with the highest (or the lowest) id then has to be elected. Such election algorithms also go by the name of *maximum-finding* or *extrema-finding* algorithms.

The following properties of systems have an important effect on the complexity of the election problem, and on whether solutions do exist at all:

1. The topology of the network. Election algorithms have been extensively studied for unidirectional and bidirectional rings, and for complete networks. In real systems, these topologies may either exist physically, or it may be simulated.

2. Whether the system is synchronous or asynchronous.

3. Whether the system is anonymous or not. For instance, it can be shown that in an anonymous ring, no deterministic solution to the election problem exists, whether the system is synchronous or asynchronous. This means that in anymous rings, one has to resort to randomized solutions. Then, nodes first draw a random number with many bits, so that the probability of two nodes choosing the same number is very small, and then execute an election algorithm for non-anonymous systems.

4. Whether or not the size of the network is known ahead of time. Algorithms that can function in rings the size of which is not known are called *uniform*. Of course, in a non-anonymous ring, it is easy for a processor to find out the size of the ring by sending a message around the ring with its id and a counter which is incremented for every hop.

5. Whether or not an algorithm is *comparison based*. This notion can be made precise, but informally, it means that in addition to receiving, copying, and sending processor ids, they can only be mutually compared as a basis for actions. In synchronous systems, non-comparison-based algorithms can be much more efficient, as shown by Algorithm 4.18. However, in such solutions, the values of ids are used for such things as delaying messages. It can be shown [43] that the message complexity of comparison-based algorithms in rings is $\Omega(n \log n)$.

Of course, in case of a non-anonymous system with any topology, there is an obvious solution to the election problem: First every processor sends its id to every other processor, and then every processor compares all ids to see whether its own is the largest or not. When there are $n$ processors, this protocol takes $n(n-1)$ messages (not counting hops). So the aim is to design algorithms with a message complexity smaller than $n^2$. The time complexity of this solution is $O(1)$.

### 4.3.1 Bidirectional Rings

We first present two election algorithms in bidirectional rings, both of which have message complexity $O(n \log n)$.

**Algorithm 4.14** *Hirschberg's and Sinclair's election algorithm in a bidirectional ring* [27].

**Idea:** Let for odd $k > 0$ the $k$-neighborhood of a processor be defined as the contiguous segment of the ring of size $k$ with the processor in the middle. In every subsequent phase $l$, starting with phase 1, every processor tries to establish whether it has the largest id among the processors in its $(2^l + 1)$-neighborhood, after it has first established that it has the largest id among the processors in its $(2^{l-1} + 1)$-neighborhood. In order to do so, it sends a PROBE message with its id and a hop counter, which is initialized to $2^{l-1}$, in each direction. When a processor receives a PROBE message with a smaller id than its own, it discards it. Otherwise, it forwards the message after having decremented the hop counter by one, or, when the hop counter is equal to zero, it sends an OK message with the id of the initiating processor back to that processor. If a processor receives OK messages with its own ide from both sides in some phase, it initiates the next phase. If the ring size $n$ is known, a processor knows that it has been elected when it receives two OK messages in phase $l_0$, with $l_0$ the lowest integer such that $2^{l_0} + 1 \geq n$. If the ring size is not known, a processor knows it has been elected when it receives its own PROBE messages from the "wrong" sides. Alternatively, when some processor receives a PROBE message from either side with the same id which is larger than its own, it can conclude that the corresponding processor is the one to be elected, and it can send this process a special message to that effect.

In the implementation below, `left` and `right` are used to identify the neighbors of a processor, and if `dir` is any of these two values, $\overline{\texttt{dir}}$ is the other.

**Implementation:**

```
I. Initiating the election
l ← 1
counter ← 0
send (id,out,1) to left,right

II. Receiving an outgoing message
upon receipt of (nid,out,h) from dir do
    if (nid>id) then
        h ← h-1
        if (h>0) then
            send (nid,out,h) to $\overline{\texttt{dir}}$
        else
            send (nid,in) to dir
```

```
III. Receiving an ingoing message
upon receipt of (nid,in) from dir do
    if (nid=id) then
        counter ← counter + 1
        if (counter=2) then
            if ($2^{\texttt{l}}$+1 $\geq n$) then
                elected ← true
            else
                l ← l+1
                counter ← 0
                send (id,out,$2^{\texttt{l}-1}$) to left,right
    else
        send (nid,in) to $\overline{\texttt{dir}}$
```

**Correctness:** Clearly, the processor with the maximum id will get elected because its PROBE messages will always return as SUCCESS messages. On the other hand, for any other `pid`, at least one of the PROBE messages will hit a processor with a higher id, and so will be discarded.

**Complexity:** The message complexity of Algorithm 4.14 in a ring of $n$ processors is $O(n \log n)$. To see this, clearly, after phase $l$ at most $n/(2^{l-1}+1)$ processors can still be trying to get elected. At most $4 \cdot 2^{l-1}$ message are generated by such a process in phase $l$, so the total number of messages in phase $l$ is bounded by

$$4 \cdot 2^{l-1} \cdot \frac{n}{2^{l-1}+1} \approx 4n.$$

As the total number of phases is equal to ${}^2\log(n/2)$, the message complexity follows. □

**Example 4.15** In Figure 4.1, the execution of Algorithm 4.14 is shown in a bidirectional ring with 12 processes. After one round, the four processes with ids 7, 10, 11, and 12 are still active, as they have ids that are higher than their neighbors'. In the second round, these processes test whether their ids are higher than those in a neighborhood of size 5 (that is, with 2 processes on either side). Only the process with id 11 then finds a process with a larger id (12) in this neighborhood.

In Algorithm 4.14, PROBE messages are sent in neighborhoods of statically defined increasing sizes. The algorithm below is more dynamic in that the neighborhoods probed in a subsequent phase are bounded by still active processes.

**Algorithm 4.16** *An election algorithm in a bidirectional ring.*

**Idea:** In the first phase, every process exchanges ids with its two neighbors. When a process detects that its own id is larger than those of its two neighbors, it remains active and initiates the second phase. Otherwise, a process becomes passive. In every subsequent phase, only the active processes execute the same algorithm in the *virtual ring* of processes that are still active, with the passive processes

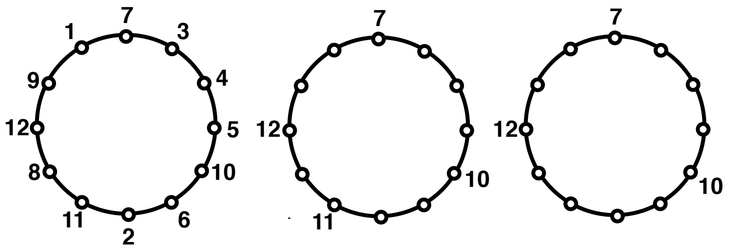


Figure 4.1: An example of the execution of Algorithm 4.14: the original ring (left), and the active processes after one (middle) and two (right) rounds.

simply relaying messages. When a process receives its own id, it has been elected. □

**Example 4.17** In Figure 4.2, the execution of Algorithm 4.16 is shown in a bidirectional ring with 12 processes. After one round, the four processes with ids 7, 10, 11, and 12 are still active, just as in Example 4.15. However, in the second round, all active processes except for the process with id 12 has an active neighbor with a higher id, so only the process with id 12 survices round 2, and then wins the election.

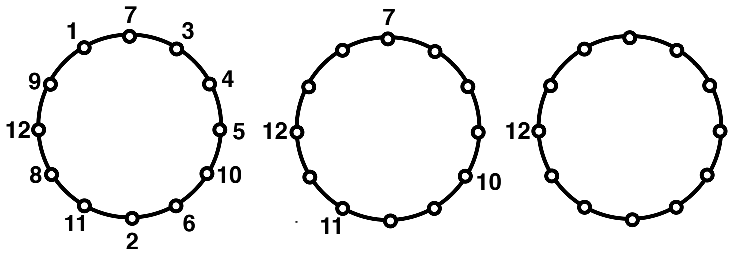


Figure 4.2: An example of the execution of Algorithm 4.16: the original ring (left), and the active processes after one (middle) and two (right) rounds.

### 4.3.2 Unidirectional Rings

We now turn to election algorithms in unidirectional rings. We assume that the neighbor a processor can send to is its right-hand neighbor.

**Algorithm 4.18** *A non-comparison-based election algorithm in synchronous unidirectional rings.*

**Idea:** This algorithm in a synchronous unidirectional ring with positive ids shows that non-comparison-based algorithms may have a lower complexity than the complexity of $O(n \log n)$ of comparison-based algorithms in rings. In this algorithm, the process with the minumum rather than the maximum id is chosen. Let the ring size be $n$, and assume that this size is known to all processors. When a processor finds that its id is equal to 1, it knows that it will be elected, and it sends in round 1 its id to its neighbor. Every process relays this id in the round immediately after it receives it. If a process does not receive anything in the first $n$ rounds, it knows that id 1 does not occur in the ring. In general, if a processor has id equal to $k$ and it has received nothing in rounds 1 through $(k-1)n$, it knows that it will be elected, and it sends its id along the ring in round $(k-1)n+1$.

**Complexity:** The message complexity of this algorithm is $n$ and the time complexity is $n$ times the minimum of the ids. □

**Algorithm 4.19** *Chang's and Roberts's election algorithm in a unidirectional ring* [11].

**Idea:** At least one processor spontaneously starts the algorithm by sending a message with its id to its neighbor. Upon receipt of a message, depending on whether the id it contains is equal to, is smaller than, or exceeds the local id, the processor is elected, the message is discarded and the processor sends a message with its own id if it has not already done so, or the message is relayed, respectively.

**Implementation:**

```
I. Spontaneously starting the election
id_sent ← true
send(id)
```

```
II. Receiving a message
upon receipt of (nid) do
    if (nid=id) then elected ← true
    if ((nid < id) and (¬id_sent)) then
        id_sent ← true
        send(id)
    if (nid > id) then
        id_sent ← true
        send(nid)
```

**Complexity:** The number of messages sent in Algorithm 4.19 is at least equal to $n$, is at most equal to $n(n+1)/2$, and is on average of order $O(n \log n)$. Obviously, a message with the largest id will travel all the way around the ring, which accounts for the lower bound. As to the upper bound, without loss of generality we can assume that the set of ids is $\{1, 2, \dots, n\}$. The id with value $i$ can travel at most $i$ steps, and when the ids are arranged in decreasing order around the ring, this is indeed possible for all $i$, which accounts for the upper bound.

As to the average message complexity, assume that every process spontaneously starts the algorithm. Let for $i = 1, 2, \dots, n-1$ and $k = 1, 2, \dots, i$, $P(i,k)$ be the probability that the message with id equal to $i$ travels exactly $k$ steps, that is, $i$ exceeds the ids of the first $k-1$ nodes in the direction of sending, but is smaller then the id of the $k$-th node in that direction. We have

$$P(i,k) = \frac{C(i-1,k-1)}{C(n-1,k-1)} \cdot \frac{n-i}{n-k},$$

where $C(p,q)$ is the number of combinations of $p$ out of $q$. The first factor is the probability that when choosing the $k-1$ ids after $i$ out of the remaining $n-1$, they are all smaller than $i$, and the second factor is the probability that the $k$-th is larger than $i$. The expected number $E_i$ of steps of the message with id equal to $k$ is

$$E_i \quad = \quad \sum_{k=1}^{i} k \cdot P(i,k), \qquad i = 1, 2, \dots, n-1,$$

$$E_n \quad = \quad n.$$

Using the equality

$$\sum_{i=k}^{n-1} k \cdot P(i,k) = \frac{n}{k+1}$$

(see Exercise 11), we find that the total expected number $E$ of messages is equal to

$$E = \sum_{i=1}^{n} E_i = n + \sum_{i=1}^{n-1} \sum_{k=1}^{i} k \cdot P(i,k) = n \left( 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \right) = O(n \log n).$$

□

We now turn to a more efficient solution to the election problem in a unidirectional ring.

**Algorithm 4.20** *Peterson's election algorithm in a unidirectional ring* [44].

**Idea:** Algorithm 4.16 is simulated in a unidirectional ring. Every process first sends its id to its right neighbor, and subsequently sends the maximum of its own id and the value received from its left neighbor to its right neighbor. If among the three values a process now possesses, the first one that was received is at least as large as the other two, the process remains active, otherwise it becomes a relay process. This same procedure is then repeated over and over again in the virtual ring only consisting of the active processes, with the relay processes only acting as transmitters of messages. A process is elected when it receives its own id.

**Implementation:**

```
I. Active processes
tid ← id
do forever
    send (tid); receive (ntid)
    if (ntid=id) then elected ← true
    send (max(tid,ntid)); receive (nntid)
    if (nntid=id) then elected ← true
    if ((ntid>=tid) and (ntid>=nntid)) then
        tid ← ntid
    else goto relay
```

```
II. Relay processes
relay:
do forever
    receive (tid)
    if (tid=id) then elected ← true
    send (tid)
```

**Correctness:** We call one execution of the loop in code fragment I of Algorithm 4.20 in some process a round. (But note that the algorithm is asynchronous, even though some form of synchrony is enforced by the message pattern.) Let the id of process $P_i$ be denoted by $\texttt{id}_i$, $i = 0, 1, \ldots, n-1$, and let $\texttt{id}_m$ be their maximum. We say that an id survives round $k$ if it is equal to the `tid` of some active process at the start of round $k+1$. Clearly, $\texttt{id}_m$ always survives, and as it continues to make progress around the ring in each round, it will eventually return to $P_m$, which then concludes it has been elected. So the only potential problem is that another process thinks it has been elected. It is easy to see that if process $P_i$ has $\texttt{tid} = \texttt{id}_j$ at the start of some round, then all of the processes $j, j+1, \ldots, i-1$ (we take the process numbers modulo $n$) are relay processes in that round. As long as $\texttt{id}_l$ with $l \neq m$ survives, it will at the start of successive rounds be equal to the `tid` of processes that are ever closer to $P_m$, until at some point there is no active process left before $P_m$. But then $\texttt{id}_l$

and $\texttt{id}_m$ are in neighboring active processes, with $\texttt{id}_m$ in an active process between $P_m$ and $P_l$, and so $\texttt{id}_l$ will not survive the next round.

**Complexity:** The number of rounds is at most equal to $\log n$, because in every round, the number of active processes is at least cut in half. Because in every round exactly two messages are sent along every link, the number of messages is at most equal to $2n \log n$. In [44], it is proven that the message delay, defined as the longest chain of messages in the algorithm, is at most equal to $2n - 1$. □

**Example 4.21** In Figure 4.3, a part of a unidirectional ring with three processes is shown. In the first part of the first round, process $P_3$ sends its id of 3 to $P_2$, and $P_2$ sends its own id 7 to $P_1$. In the second part of the first round, $P_2$ sends the maximum of its own id (7) and the value it received in the first round (3), so a 7, to $P_1$. Process $P_1$ now has three values 4,7,7, and as the first received is at least as large as the other two, it remains active with that value (7). □

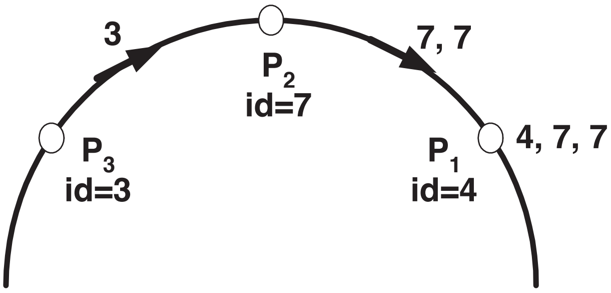


Figure 4.3: An example of the execution of Algorithm 4.20.

### 4.3.3 Complete Networks

We present below algorithms for election in a synchronous and an asynchronous complete network.

**Algorithm 4.22** *Afek's and Gafni's algorithm for election in a synchronous complete network* [1].

**Idea:** A node that wants to be elected (a *candidate*) repeatedly sends its id to ever larger different subsets of other nodes, who only return acknowledgments if the id received exceeds the largest id in the system they currently know about. When the number of acknowledgments a candidate receives is smaller than the number of nodes in the subset, it ceases to be a candidate. The size of the initial subset is equal to 1, and the size of every next subset is twice the size of the previous subset, except for the last one, which just contains the remainder of the nodes. Nodes keep track of the nodes to which they have already sent their own id and and exclude these nodes from any future subset of nodes to send their id to. A node has been elected when it has sent its id to and received acknowledgments from all other nodes.

The algorithm proceeds in rounds, and any number of nodes may spontaneously start the algorithm in possibly different rounds. A node that does so, spawns two processes, a *candidate* process and an *ordinary* process, while a node that does not spontaneously start the algorithm only spawns an ordinary process upon the receipt of the first message. Candidate processes keep track of their *level*, which is the round number since they started. These processes send candidate messages with as contents the pair of the current level of the node and the node id to the ordinary processes in other nodes. In every round, the ordinary processes order the messages they have received according to the lexicographic ordering (level first), and when the maximum is larger than their own current identifier, they assume the maximum as their new identifier and send an acknowledgment back to the corresponding candidate process. In that case, the node of the ordinary process is said to have been *captured* by the node of the candidate process, which in turn is called the *owner* of the former node. Ordinary processes also, and independently, keep track of their level, which is incremented by 1 in every round, or is set to the level in a received message, if that is larger. The node elected is the node with the largest id among all candidates that were the first to start the algorithm (and so, have the highest level). In particular, a node with only an ordinary process can never be elected.

**Implementation:**

```
I. The candidate process
E ← set of all links connected to the node
level ← -1
do forever
    level ← level + 1
    if (level mod 2 = 0) then
        if (E = ∅) then
            elected ← true
        else
            K ← min($2^{\text{level}/2}$, |E|)
            E′ ← any subset of E of K elements
            send(level, id) on all links in E′
            E ← E \ E′
    else
        A ← set of all acks received
        if (|A| < K) then STOP
```

```
II. The ordinary process
link ← nil
level ← -1
do forever
    send(ack) over link
    level ← level + 1
    R ← set of all candidate messages received
    (nlevel, nid) ← lexicographic maximum in R
    if ((nlevel, nid) > (level, id)) then
        (level,id) ← (nlevel,nid)
        link ← link over which (nlevel,nid) is received
    else
        link ← nil
```

**Correctness:** Clearly, only the process with the largest id among those that are the earliest to start will always receive acknowledgments from all processes to which it sends a candidate message.

**Complexity:** The maximum number of (double) rounds the algorithm takes is $\log(n)$, as the final winner will after this number of rounds have captured all other nodes. The maximum number of messages is $2n \cdot \log(n)$, which can be explained as follows. First of all, every node sends at most one acknowledgment in every (double) round, leading to a maximum of $n{\cdot}\log(n)$ such messages. As to the number of candidate messages, note that in any round, the sets of nodes captured by different candidate processes are disjoint, as an ordinary process only sends an acknowledgment to the candidate process with the maximum id among the ones from which he receives a candidate message. As the size of the set to which a candidate process sends a candidate message in (double) round $i$ is $2^{i-1}$, in (double) round $i$, at most $n/2^{i-1}$ candidate process still exist. So the total number of candidate messages does not exceed

$$\sum_{i=1}^{\log(n)} (n/2^{i-1})2^{i-1} = n\log(n).$$

It can be shown that this algorithm is message-optimal for a time-optimal algorithm. □

Algorithm 4.22 cannot be immediately generalized to asynchronous systems, because the arbitrary delays of messages can lead to deadlock. As an example, assume that two candidate processes $C_1$ and $C_2$ each first capture one node, and subsequently proceed to capture the same two nodes $N_1$ and $N_2$ with small ids. Then if the candidate message of $C_1$ arrives first at $N_1$ and the candidate message of $C_2$ arrives first at $N_2$, both candidates will receive one acknowledgment, and deadlock arises.

In order to adapt Algorithm 4.22 to asynchronous systems, note that it does not make sense for a candidate process to wait for all messages that have been sent to it. So it should immediately reply as soon as it receives one candidate message.

**Algorithm 4.23** *Afek's and Gafni's algorithm for election in an asynchronous complete network* [1].

**Idea:** Again, the candidate messages consist of pairs (level, id), but now the level indicates the number of nodes a candidate process has captured. However, this means that if a candidate process captures a node that had already previously been captured (by a node with a lower (level, id) at the time of capturing), the level of the previous owner is not correct anymore. As the previous owner will not be elected anyway, and in order to reduce the number of messages, the node to be captured again tries to kill its previous owner. In order to do so, ordinary captured processes maintain a pointer `owner` and `potential-owner` to their current and potential new owner. Note that because of the asynchronous nature of the system, the previous owner may already have been killed by another node. In addition, when a candidate process with a relatively large, but not the maximum id has captured many nodes, it will be attempted to be killed many times through those nodes by the final winner.

**Implementation:**

```
I. The candidate process
while (untraversed $\neq \emptyset$) do
    link $\leftarrow$ any untraversed link
    send(level,id) on link
  R: receive(level',id') on link'
    if ((id=id') and (killed=false)) then
        level $\leftarrow$ level+1
        untraversed $\leftarrow$ untraversed $\setminus$ link
    else
        if ((level',id') $<$ (level,id)) then goto R
        else
            send(level',id') on link'
            killed $\leftarrow$ true
            goto R
if (killed = false) then elected $\leftarrow$ true

II. The ordinary process
do forever
    receive(level',id') on link'
    case (level',id') of
        (level',id') $<$ (level,owner-id):  ignore
        (level',id') $>$ (level,owner-id):
            potential-owner $\leftarrow$ link'
            (level,owner-id) $\leftarrow$ (level',id')
            if (owner=nil) then owner $\leftarrow$ potential-owner
            send(level',id') on owner-link
        (level',id') = (level,owner-id):
            owner $\leftarrow$ potential-owner
            send(level',id') on owner-link
```

**Complexity:** The time complexity of Algorithm 4.23 is $n$. First of all, candidate processes capture nodes independently from each other, and the final winner has to capture $n-1$ nodes. Secondly, when a candidate process kills another candidate process, the former has to have done at least the same amount of work as the latter as it must have a higher level. The message complexity of the algorithm is $n\log(n)$.

## 4.4 Shortest-Path Spanning Trees

When a node in a weighted directed or undirected network (with positive edge weights) repeatedly sends point-to-point message to many other nodes, it is efficient for it to compute the shortest paths from itself as the source to all the other nodes in the network. Below we discuss a simple but expensive algorithm for computing such a shortest-path tree rooted at a specific node. This algorithm explores all paths in the whole network starting at the root.

**Algorithm 4.24** *The Chandy-Misra algorithm for computing the shortest-paths spanning tree* [8]

In this algorithm, every node $n$ maintains two variables, its current estimate $d_n$ of its distance from the root, initialized at $\infty$, and its parent, which is the last node on the path from the root to itself that it currently thinks is the shortest, initialized at `nil`. Nodes repeatedly send to their neighbors messages with their value of $d_n$ in the following way. The root $r$ starts the algorithm by sending an initial message to all its neighbors with value $d_r = 0$, indicating its distance from the root of the tree (this is the only action of the root). When a node $n$ receives a message from its neighbot $m$ with distance value $d_m$, it checks whether $d_n > d_m + w(m, n)$, where $w(m, n)$ denotes the weight of the edge from $m$ to $n$. If so, it updates the value of its variable $d_n$ to $d_m + w(m, n)$ and the value of its variable $p_n$ to $m$, and sends a message with the new value of $d_n$ to all its neigbors except for node $m$. If not, it discards the message.

For deciding whether the algorithm has completed, a termination detection algorithm is needed.

## 4.5 Minimum-Weight Spanning Trees

When in a connected undirected graph the diameter of which is known, a node wants to broadcast a message, the following *flooding* algorithm can be used. The node sends the message with a hop counter initialized to the diameter of the graph along each of the edges it is connected to. When a node receives the message for the first time, it decrements the hop counter and sends the message with the modified hop counter, if this is still positive, on every edge it is connected to, except the edge along which it received the message. Messages with a hop counter of 0 are discarded. In general, one may assign a (positive) weight to every edge in a graph, which is indicative of the cost associated with sending a message along the edge. An objective can then be to perform a broadcast such that the sum of the weights of the edges along which the message is sent, is minimal. In this section we will consider the problem of minimizing this cost.

We start with some definitions. Let $G = (V_G, E_G)$ be a connected undirected graph with node set (or vertex set) $V_G$ and edge set $E_G$. (If no confusion can arise, we omit the subscript $G$ in the denotation of the sets of nodes and edges.) An undirected graph is a *tree* if it is connected and does not contain cycles. In a tree, the numbers of nodes and edges are related by $|E| = |V| - 1$. A *spanning tree* $T = (V_T, E_T)$ of an undirected connected graph $G$ is a tree with $V_T = V_G$ and $E_T \subset E_G$. A graph $G = (V, E)$ is *weighted* if every edge $e \in E$ has a (real or integer) weight $w(e)$ attached to it. The weight of a spanning tree of a weighted graph is the sum of the weights of the edges in the tree. A Minimum-weight Spanning Tree (MST) of a weighted graph $G$ is a spanning tree of $G$ of minimal weight. MSTs of a weighted graph are in general not unique.

The requirement of different weights in $G$ is not very important if we assume that in $G$ all nodes have different integer identities. If in a graph $G$ edges of equal weights would occur, we can assign every edge $e$ connecting nodes $n_1$ and $n_2$ with identies $i_1, i_2$ with $i_1 < i_2$ the triple $(w(e), i_1, i_2)$ as its weight, and use the lexicographic ordering on these weights.

**Lemma 4.25** *A weighted connected undirected graph in which all weights are different has a unique MST.*

PROOF. Suppose that there exists a weighted connected undirected graph $G$ with different edge weights that has two different MSTs $T$ and $T'$. Let $e \in E_G$ be the edge of minimum weight that does occur in one of these MSTs but not in the other. Without loss of generality, we can assume that $e$ occurs in $T$ but not in $T'$. Of course, $E_{T'} \cup \{e\}$ contains a cycle, and as $T$ does not contain cycles, this cycle contains at least one edge, say $e'$, that does not belong to $T$. Because of the choice of $e$, we have $w(e') > w(e)$. Now $E_{T'} \cup \{e\} - \{e'\}$ is a spanning tree of weight less than the weight of $T'$, which is a contradiction. □

For the remainder of this section we fix a weighted connected undirected graph $G$ with different edge weights. A *fragment* of $G$ is a subtree of its (unique) MST. The edge $e$ of $G$ is the Minimum-weight Outgoing Edge (MOE) of fragment $F$ if $e \notin E_F$, if exactly one of the two nodes connected by $e$ is in $F$, and if $e$ has minimum weight among the edges in $G$ with these two properties.

**Lemma 4.26** *Let $F$ be a fragment of $G$, let $e$ be its MOE, and let $v$ be the node connected by $e$ that is not in $F$. Then $F \cup \{e\}$ is a fragment of $G$.*

PROOF. Suppose $F \cup \{e\}$ is not a fragment. Then adding $e$ to the MST of $G$ creates a cycle consisting of a subset of edges in the MST and $e$. At least one edge $x$ in this cycle that is different from $e$ is then also an outgoing edge of $F$. Because $e$ has the minimum weight of the outgoing edges of $F$, we have $w(x) > w(e)$. Now replacing $x$ by $e$ in the MST yields a spanning tree of weight smaller than the weight of the original MST, which is a contradiction. □

**Example 4.27** Let $G$ be a ring with an even number of nodes, and assume that the weights along the ring are alternatingly low and high. Then if we connect all nodes with their MOEs, we get $|V|/2$ unconnected fragments each consisting of two nodes. □

**Example 4.28** Let $G$ be a ring, assume that the nodes are identified by the integers 0 through $|V|-1$, and let nodes $i$ and $i+1$ (modulo $|V|$) be connected by an edge of weight $i+1$. Then if we have every node connect through its MOE, we end up with the MST of $G$. □

**Algorithm 4.29** *Gallager's, Humblet's, and Spira's algorithm for the MST of an undirected weighted graph with different edge weights* [23].

**Idea:** The general idea of the algorithm is to have fragments connect to each other along their MOEs, starting with single-node fragments and ending when there is only a single fragment left, which is then the MST. Because the algorithm is rather complex, we describe it at a high level in the following three steps, before giving the details.

1. The MST is constructed by repeatedly connecting pairs of fragments along a single edge that is the MOE of at least one of them, starting from the set of single-node fragments. The nodes of a fragment cooperate to find the MOE of the fragment. Each fragment is assigned a *level*, with single-node fragments having level 0. (Due to the asynchronicity, different runs of the algorithm may create different fragments.) Every fragment except for level-0 fragments has a unique edge that is called its *core*. Every fragment has a fragment *name*, which is the weight of its core.

2. The rules for connecting fragment $F$ of level $l$ to fragment $F'$ of level $l'$ are:
    - If $l = l'$ *and* the MOEs of $F$ and $F'$ coincide, then a new fragment of level $l+1$ is created by *merging* $F$ and $F'$ along their common MOE, which is the core of the newly created fragment;
    - If $l < l'$, then fragment $F$ is *absorbed* by $F'$ along the MOE of $F$, which is not necessarily the MOE of $F'$. The resulting fragment retains the level and the core (and so the name) of $F'$;
    - If $l > l'$ or if $l = l'$ but the MOEs of $F$ and $F'$ do not coincide, connecting the fragments is postponed until one of these two conditions is satisfied.
3. Any subset of processes may start the algorithm spontaneously. Processes that do not do so start the algorithm upon receipt of a message pertaining to the algorithm from another process.

We now explain the implementation details of this algorithm. We start with the data structures:

1. For each of its adjacent edges, a node maintains its state, which can have three values:
    (a) `?_in_MST`, which indicates that the node does not know yet whether the edge will be in the MST or not;
    (b) `in_MST`, which indicates that the edge is part of the MST;
    (c) `not_in_MST`, which indicates that the edge is not part of the MST.

    The initial value of all edge state variables is `?_in_MST`, and there is only a single assignment to them in a run of the algorithm.
2. Every node maintains its own state, which can have the following values:
    (a) `sleeping`, which is the initial state;
    (b) `find`, which indicates that the node is participating in finding the MOE of the current fragment it belongs to;
    (c) `found`, which indicates that the node has finished its own part in finding the MOE of the current fragment it belongs to.

    Once a node is not `sleeping` anymore, its state alternates between the values `find` and `found`.
3. Nodes maintain the following additional data structures:
    (a) the name of the current fragment it belongs to;
    (b) the level of the current fragment it belongs to;
    (c) the edge adjacent to it that leads to the core of the current fragment it belongs to;
    (d) the number of `report` messages it still expects (see below);
    (e) the edge adjacent to it that leads towards the best candidate for the MOE it knows about;
    (f) the weight of the best candidate for the MOE it knows about;
    (g) the edge adjacent to it that it is currently testing for being a candidate MOE.

We now give a description of the operation of the algorithm based on the seven message types used (the Roman numerals refer to the code fragments in the implementation):

1. The nodes in a fragment cooperate to find the fragment's MOE. In order to do so, the two nodes connected by the fragment's core broadcast an `initiate` message carrying the fragment's name and level in their "own" parts of the fragment (III). As a node belongs to different, ever larger fragments in the course of the execution of the algorithm, it will be involved in MOE finding multiple times.

2. When a node receives an `initiate` message, it forwards an `initiate` message along every edge in state `in_MST` except for the edge along which the `initiate` message was received, and records the number of such messages forwarded (IV). The node then tries to select a candidate MOE among its own adjacent edges (V). It does so by checking the edges with status `?_MST` in the order of increasing weight. An edge is checked by sending a `test` message along it, containing the fragment's name and level. The purpose of this message is to find out whether the node at the other end of the edge is in the same or a different fragment.

3. When a node receives a `test` message (VI) from a lower-level fragment or from a fragment of the same level with a different fragment name, it replies with an `accept` message. The node receiving an `accept` message records the edge as a potential MOE (VIII).

4. When a node receiving a `test` message (VI) has the same fragment name, it replies with a `reject` message. The node receiving a `reject` message sets the state of the edge to `not_in_MST` (VII).

5. The potential MOEs and their weights are sent back in the direction of the core with `report` messages (VII). Nodes wait until they have received the same number of such messages as the number of `initiate` messages they have forwarded (IX) before reporting the optimal candidate MOEs in the subtrees of which they are the roots.

6. When the core nodes have received all report messages they expect (X), and have exchanged the best candidates in their own subtrees, the MOE of the whole fragment is known to them. Then a `change-root` message is sent from the core to the node that is connected to the MOE (X).

7. When the node that has the MOE of a fragment as one of its edges receives a `change-root` message (XI), it sends a `connect` message over the MOE in order to have its fragment merge with or be absorbed by the fragment at the other end of the MOE. When a node receives a `connect` message, it absorbs the fragment from which it receives this message if this fragment's level is lower than its own level (III). In addition, it sends an `initiate` message back, which causes the absorbed fragment to cooperate in finding the MOE (the last argument of the `initiate` message is `find`), or to get to know the fragment's name (the last argument of the `initiate` message is `found`). When the level of the fragment from which the `connect` message is received is not lower than the level of the receiving fragment, the message is appended to the queue if the edge is not in the MST.

Finally, at three points in the algorithm, a message that is received cannot be handled immediately, but is appended to the message queue until the condition for handling it becomes true. We leave it to the reader to argue why eventually these conditions become true and the messages will be deleted from the queue (see Exercise 21).

**Implementation:**

```
I. Spontaneously starting the algorithm
when (SN=sleeping) do
   wakeup()

II. Procedure wakeup()
j ← adjacent edge of minimum weight
SE(j) ← in_MST; LN ← 0; SN ← found
find-count ← 0
send(connect;0) on edge j

III. Receiving a connect message
upon receipt of (connect;L) on edge j do
   if (SN=sleeping) then wakeup()
   if (L<LN) then
      SE(j) ← in_MST
      send(initiate;LN,FN,SN) on edge j
      if (SN=find) then
         find-count ← find-count + 1
   else
      if (SE(j)=?_in_MST) then
         append message to message queue
      else
         send(initiate;LN+1,w(j),find) on edge j

IV. Receiving an initiate message
upon receipt of (initiate;L,F,S) on edge j do
   LN ← L; FN ← F; SN ← S
   in-branch ← j
   best-edge ← NIL; best-wt ← ∞
   for all (adjacent edges i, i ≠ j, SE(i) = in_MST do
      send(initiate;L,F,S) on edge i
      if (S=find) then
         find-count ← find-count + 1
   if (S=find) then test()

V. Procedure test()
if (there are adjacent edges in state ?_in_MST) then
   test-edge ← edge in state ?_in_MST of minimum weight
   send(test;LN,FN) on test-edge
else
   test-edge ← nil
   report()
```

VI. Receiving a test message

```
upon receipt of (test;L,F) on edge j do
    if (SN=sleeping) then wakeup()
    if (L>LN) then append message to message queue
    else
        if (F $\neq$ FN) then
            send(accept) on edge j
        else
            if (SE(j)=?_in_MST) then
                SE(j) = not_in_MST
            if (test-edge $\neq$ j) then
                send(reject) on edge j
            else test()
```

VII. Receiving a reject message

```
upon receipt of (reject) on edge j do
    if (SE(j)=?_in_MST) then
        SE(j) $\leftarrow$ not_in_MST
    test()
```

VIII. Receiving an accept message

```
upon receipt of (accept) on edge j do
    test-edge $\leftarrow$ NIL
    if (w(j)<best-wt) then
        best-edge $\leftarrow$ j
        best-wt $\leftarrow$ w(j)
    report()
```

IX. Procedure report()

```
if ((find-count=0) and (test-edge=NIL)) then
    SN $\leftarrow$ found
    send(report;best-wt) on in-branch
```

X. Receiving a report message

```
upon receipt of (report;w) on edge j do
    if (j $\neq$ in-branch) then
        find-count $\leftarrow$ find-count-1
        if (w<best-wt) then
            best-wt $\leftarrow$ w
            best-edge $\leftarrow$ j
        report()
    else
        if (SN=find) then
            append message to message queue
        else
            if (w > best-wt) then
                change-root()
            else
                if (w=best-wt=$\infty$) then HALT
```

```
XI. Procedure change-root
if (SE(best-edge) = in_MST) then
    send(change-root) on best-edge
else
    send(connect;LN) on best-edge
    SE(best-edge) = in_MST
```

```
XII. Receiving a change-root message
upon receipt of change-root do
    change-root()
```

**Complexity:** The message complexity of the algorithm is $O(5 \cdot |V| \cdot \log |V| + 2 \cdot |E|)$. □

**Example 4.30** Consider in Figure 4.4 the fragments $F_1, F_2$, and $F_3$, with levels $L_1, L_2$, and $L_3$, respectively. Assume that $L_1 = L_2$, and that $L_3 < L_1$. The core of $F_1$ is the edge between nodes A and B, and the core of $F_3$ is the single edge in $F_3$. When the nodes in $F_1$ start finding the fragment's MOE, nodes A and B initiate an INITIATE in their subtrees of $F_1$. When node $C$ receives an INITIATE message, it starts finding the candidate MOE it is adjacent to. First it will send a TEST message to node D, which will reply with a REJECT message because it is in the same fragment. Then it will send a TEST message to node F, which replies with an ACCEPT. Then node C will send a report back towards the core of $F_1$. When edge CF is indeed the MOE of $F_1$, node B will send a CHANGE-ROOT along the path to C, which will then send a CONNECT along edge CF. If CF is also the MOE of $F_2$, fragments $F_1$ and $F_2$ will merge to form a fragment of level $L_1 + 1$ with as its core the edge CF.

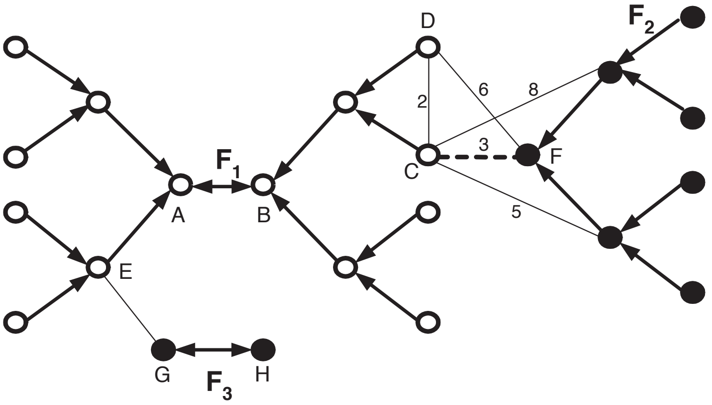


Figure 4.4: An example of merging and absorbing fragments in Algorithm 4.29.

Because fragment $F_3$ is of a lower level, it can simply be absorbed along the edge GE. Then node E sends an INITIATE message to node G. If at the moment that $F_3$ is absorbed, node E had not already finished finding the candidate MOE of the subtree of which it is the root, this message will include $F_3$ in the search for the MOE. Otherwise, this message simply notifies $F_3$ that it has now become part of $F_1$. □

## 4.6 Bibliographic Notes

See for an extensive treatment of election in ring networks Chapter 3 of [2]. A solution to the problem of election in anonymous rings is to use randomization [26]. Descriptions of the Gallager-Humblet-Spira algorithm can also be found in the books [4, 38, 61].

## 4.7 Exercises

1. Show that when the links are not FIFO, Algorithm 4.1 does not function correctly in that multiple processes may be in their CSs simultaneously.

2. Write an implementation of Algorithm 4.2.

3. Are the links necessarily FIFO in Algorithm 4.2?

4. Write an implementation of Algorithm 4.7.

5. What are the sets $R_i, I_i$ and $S_i$ in Algorithm 4.7 in a completely centralized mutual-exclusion algorithm, in Algorithm 4.2, and in Algorithm 4.6?

6. Show that when in code fragment III in Algorithm 4.9 the process holding the token and the token itself have the same request number for process $P_j$ (`N[j]=TN[j]`), the state of $P_j$ has to be copied from the token into the process and not the other way around.

7. If in Algorithm 4.13, the assignment of `m[j]` to `l` would be performed at the end of code fragment I. when a token arrives rather than at the end of code fragment II. when a token leaves, would the algorithm then still function correctly?

8. Algorithm 4.13 is resilient to the loss of only one token. Extend this algorithm in such a way that it is resilient to the loss of $k-1$ tokens, for $k > 2$. (Hint: Use $k$ tokens.)

9. In Algorithm 4.13, `l,n,m[0]` and `m[1]` are unbounded. Adapt the algorithm in such a way that these numbers are bounded. (Hint: Find a suitable `M`, such that these numbers modulo `M` can be used.)

10. Write an implementation of Algorithm 4.16.

11. To complete the proof of the average-case complexity of Algorithm 4.19, show that

$$\sum_{i=k}^{n-1} k \cdot P(i,k) = \frac{n}{k+1}.$$

Hint: Use (and prove)

$$\sum_{i=k}^{n-1}(n-i)\left(\begin{array}{c} i-1 \\ k-1 \end{array}\right) = \left(\begin{array}{c} n \\ k+1 \end{array}\right).$$

12. Why does in code fragment I. of Algorithm 4.20 an active process send as its second message `max(tid,ntid)` instead of simply `nid`?

13. Show how Algorithm 4.20 proceeds when the consecutive ids in the direction of sending are $0, 1, \ldots, n-1$ and when they are $n-1, n-2, \ldots, 1$.

14. Devise a distribution of the ids $\{0, 1, \ldots, n-1\}$ along a unidirectional ring with $n = 2^k$, such that Algorithm 4.20 needs $k$ phases. Hint: Use the *bit-reversal ordering* of the integers 0 through $2^k - 1$, which is defined as the ordering obtained by first putting these integers in increasing order, then replacing them by their binary representations in $k$ bits, then reversing the order of the $k$ bits in each of these representations, and finally interpreting the resulting bit strings as integers.

15. Consider the following variation of Algorithm 4.20 for election in a unidirectional ring. First, every process sends its id to its neighbor; a process goes into relay when its own id is smaller than the id received. Then, every remaining active process sends its id to the next active process; a process then remains active—with as new value the id received—only when its own id is smaller than the id received. Write this algorithm in pseudocode, and trace its execution when the ids are ordered along the ring, and when the the number of processes $n$ is a power of 2 and the numbers $0, 1, \ldots, n-1$ are in bit-reversal ordering around the ring.

16. Show that in Algorithm 4.22, the level is not really needed, and nodes can simply only use their integer id to achieve election. What is the disadvantage of doing so (consider the time needed to reach election when the node with the largest id in the system only starts participating in the algorithm at a late stage)?

17. Suupose that in Algorithm 4.29 a fragment $F$ is absorbed by a higher-level fragment $F'$, and that $n$ and $n'$ are the connecting nodes in these fragments, respectively. Then, if $n'$ had already completed reporting about the MOE candidate in the subtree of $F'$ of which it is the root, it does not include $F$ in the search for a candidate MOE, but only sends $n$ an `initiate` message to notify the nodes in $F$ of the current level and name of $F'$ Argue that this is correct, i.e., that the MOE of the combined fragment cannot be adjacent to a node of $F$.

18. Let $G$ be a weighted ring with $2^n$ processors in which the weights of the successive edges along the ring are the numbers in the range 0 through $2^n - 1$ in bit-reversal ordering (see Exercise 14). Devise an execution of Algorithm 4.29 in $G$ in which all fragments created during thr execution of the algorithm is equal to $2^k - 1$ for some $k$.

19. Let $G$ be a weighted ring with $2^n$ processors. What are the possible values for the level of the final fragment when Algorithm 4.29 is executed in $G$.

20. Let $G$ be a weighted complete network. Find an arrangement of the weights on the edges of $G$ such that an execution of Algorithm 4.29 leads to an MST which is a fragment of level 1.

21. Argue why in each of the cases that a message can be appended to the message queue in Algorithm 4.29, the condition for processing the message eventually becomes true.

# Chapter 5

# Consensus and Fault Tolerance

Distributed systems will invariably have to deal with faults caused by software or hardware components not operating according to their specifications. In Chapter 1 we have mentioned that the existence of independent failure modes is one of the characteristic properties of distributed systems. Two important factors that determine if and to what extent faults can be dealt with are the types of faults, and the level of synchronization present in the system.

Faults can be classified as permanent—once a processor exhibits faults it will be considered as faulty for ever—and transient—a processor may exhibit a fault but will return to correct operation again. Usually, permanent failures are further subdivided, with the two most important categories being stopping failures and Byzantine failures.

In the case of permanent faults, we will see that whether the system is synchronous or asynchronous has a strong influence on whether solutions for dealing with faults exist at all. In the case of transient faults, we will deal both with shared-memory models and with distributed-memory models.

## 5.1 Classifying Faults

Faults in distributed systems can be classified as permanent or transient. *Permanent faults* include a processor halting and a malfunctioning sensor giving erroneous results, leading to what are called crash failures and malicious or Byzantine failures. *Transient faults* include short power glitches causing for instance parts of a memory to be momentarily corrupted, and transmission errors.

### 5.1.1 Permanent Faults

Dealing with permanent faults has often been modeled as the need for reaching *agreement* or *consensus* among a set of processors some of which may exhibit faults.

**Example 5.1** As a counter measure against malfunctioning components, a computer system for some critical task may have multiple processors that each perform the same computations, from which the majority result is used. In applications in embedded systems, such a computation may need as its input such data as a temperature, a pressure, an altitude, a speed, or an angle produced by a sensor. Each of the processors must have an interface to the sensor. Hardware errors in the sensor or in such interfaces or slight differences in the times when the sensor is read by the processors may result in different input values to the computations. (These values may be widely different even if their representations differ only in a single bit.) So, before the processors start their computations, they first have to *agree* on the input value. □

**Example 5.2** (Making an appointment) Consider two people, in examples such as these invariably called Alice ($A$) and Bob ($B$), who engage in the process of making an appointment through an asynchronous message-passing system, e.g., the paper or the electronic mail system, in which messages may be lost. Let $P_A, P_B$ be the propositions "A wants an appointment" and "B wants an appointment." Suppose that $A$, after having set $P_A$ to true, sends a message to the effect that she wants to have an appointment, and suppose this message does arrive. So then, $B$ knows that $P_A$ is true, a fact that we denote by $K_B(P_A)$. (The notation $K_i(P)$ means that process $i$ "knows" fact $P$, and is used in the area of modal logic that deals with reasoning about the knowledge of processes [20]). If $B$ sends a positive response which also does arrive, then $P_A, K_B(P_A)$, and $K_A(K_B(P_A))$ are true. But in order to be sure that the appointment will take place, $B$ would now like to receive a confirmation from $A$, which, when it arrives, establishes $K_B(K_A(K_B(P_A)))$. This will go one forever, unless at some point $A$ and $B$ simply trust that their last messages will arrive correctly. □

**Example 5.3** When a transaction in a distributed database system spanning multiple sites wants to commit, the database managers in the different sites have to *agree* on whether this is possible or not. □

**Example 5.4** When parts of a database are replicated and a write action to a database record is initiated, the managers of the copies need to *agree* on the value to be written. □

Processors, and of course other components as well, can exhibit different kinds of failures. In the literature, many different kinds of errors are distinguished, but in this chapter, we will only be concerned with the following two types of permanent faults:

1. In a *stopping (or crash) failure*, a processor simply stops at some point and does not resume at a later time. We assume that a processor does not stop in the middle of sending a message, so a message is sent in its entirety or not at all. If according to its algorithm a processor is supposed to send a number of messages (e.g., all messages in a single round of a synchronous algorithm), it may stop after having sent only a subset of them. Stopping failures model the event of a processor going down.

2. In a *malicious* or *Byzantine failure*, basically anything is allowed. A processor may stop and resume at a later time, omit the sending or receiving of any message, and may send messages with any content. Byzantine failures model a component continuing to operate but exhibiting failures such as a sensor giving values with bits inverted.

In the agreement problems presented in this chapter, we assume that there is a set of values (e.g., $\{0, 1\}$) from which processors may take their initial value (if required by the problem) and on one of which they have to decide. We also assume that this set contains a *default value* (e.g., 0) that processors can resort to in case they do not know what value to use.

### 5.1.2 Transient Failures

Transient faults are faults exhibited by components that will return to normal operation after a while. Transient faults cause the state of a system to be incorrect. Transient faults are used to model the following situations:

1. The incorrect (or the absence of a proper) initialization of a distributed system. (For instance, when a component is added to such a system.)
2. The corruption of a part of the main memory of a number of processors. We will assume that only the variables of processes can be corrupted (including the program counter), but not the programs they execute. This means that the behavior of the process will remain the same (albeit with different data).

   Although the program code is usually stored in the same memory as the program variables, there are reasons to assume that only the latter can be corrupted. For one thing, in embedded systems, programs may be stored in ROM, which is less vulnerable and not susceptible to power failures as RAM is. Second, in non-embedded systems, programs are usually stored on disk, so if the in-memory copy of a program is corrupted (or is suspected to be so), it can be retrieved from disk. Finally, programs are usually immutable, so they can be protected with some form of redundancy which can be checked periodically.

## 5.2 Consensus in Synchronous Systems with Crash Failures

Let's consider the following problem in synchronous systems with only crash failures. Every process starts with a value from the set of possible initial values, and they have to reach agreement subject to the following conditions:

1. *Agreement*: No two processes decide on different values;
2. *Validity*: If all processes start with the same value, then no process decides on a different value;
3. *Termination*: All non-faulty processes decide within finite time.

Below we present an algorithm for this problem.

**Algorithm 5.5** *An algorithm for agreement in a synchronous system with at most $f$ crash failures.*

**Idea:** Every process maintains a set $W$ which initially only contains the value it starts with. Then in each of a succession of $f + 1$ rounds, they all broadcast their sets $W$ and set $W$ to the union of their current set $W$ and all sets they receive. After round $f + 1$, if the set $W$ in a process contains one element, the process decides on that element. If the set $W$ contains more than one element, the process decides on the default value.

**Correctness:** Because the algorithm consists of $f + 1$ rounds and there are at most $f$ processor failures, there is at least one round during which no processor fails. At the end of this round, all still active processes will have identical sets $W$, and these sets will not change anymore after this round.

**Complexity:** If there are $n$ processes in the system, then the compexity of the algorithm is of order $O(f \cdot n^2)$. As an optimization, in order to decide, a process only has to know whether its set $W$ at the end of the algorithm contains one element (then the process decides on its own original value) or more elements (then it decides on the default value). So rather than broadcasting its set $W$ in every round, it might as well broadcast only its initial value in the first round, and the first other value in the any of the sets received from any other process, if any. □

## 5.3 Consensus in Synchronous Systems with Byzantine Failures

Consider the following military problem, which goes by the name of the Byzantine-generals problem. A city is besieged by $n$ armies, each headed by a general. For the armies to conquer the city, they have to attack simultaneously. Each of the generals has an opinion as to whether to attack or not. The generals have to communicate by means of messages in order to reach a decision. Among the generals, there may be traitors, who may send messages with any content to any of the generals. The problem is to devise an algorithm in which all loyal generals reach the same decision. Clearly, it is enough to devise an algorithm in which one general (the commander) transmits his decision to other generals (the lieutenants). For this problem—the Byzantine-generals problem—we can state the following conditions:

- *Agreement*: All loyal lieutenants obey the same order;
- *Validity*: When the commander is loyal, all loyal lieutenants obey the order of the commander;
- *Termination*: All loyal lieutenants decide in finite time.

Obviously, when the commander is loyal, the first condition is a consequence of the second. One can show that in algorithms solving the Byzantine-generals problem in synchronous systems with $n$ processes, the maximal number of faulty processors $f$ has to satisfy $f < n/3$. There are two variations ot the Byzantine-agreement problem, with and without *authentication*, which means that the lieutenents can or cannot forge messages received from loyal generals when passing them along to others.

### 5.3.1 Impossibility for Three Generals

For some consensus problems no solutions exist. For instance, in a purely asynchronous system, there is no possibility to overcome even a single processor failure. Such results are called *impossibility results*. A general technique to prove such an impossibility result is to design two different scenarios that on the one hand cannot be distinguished by at least one process, while on the other hand two processes have to reach different conclusions. We will apply this proof technique now to the (synchronous) case of three generals, one of which is a traitor.

The commander is denoted by $C$, and the lieutenants by $L_1$ and $L_2$. In Scenario 1, $C$ and $L_1$ are loyal. First $C$ sends order 0 to both $L_1$ and $L_2$, and then the lieutenants exchange the orders received from $C$. However, because $L_2$ is a traitor, it sends a 1 to $L_1$ instead of a 0. In Scenario 2, only $C$ is a traitor, and sends order 0 to lieutenant $L_1$ and order 1 to $L_2$. Then the lieutenants exchange their orders. To $L_1$, Scenarios 1 and 2 are identical. As $L_1$ has to decide 0 in Scenario 1 because both $C$ and itself are loyal, it has to decide 0 also in Scenario 2. We conclude that if $L_1$ is loyal, it has to obey the order received directly from $C$. But the same holds for $L_2$ when he is loyal, and so $L_2$ has to decide 1 in Scenario 2, which contradicts the agreement condition.

### 5.3.2 Algorithms for Synchronous Systems

In this section we present the first algorithms ever presented for consensus in synchronous systems, both without and with authentication.

**Algorithm 5.6** *The Lamport-Pease-Shostak algorithm for consensus without authentication in synchronous systems with a completely connected network* [34].

**Idea:** The algorithm can tolerate a maximum number $f$ of faults, and is recursive. As the bottom case, when $f$ equals 0, the commander sends his value to the lieutenants, who simply decide on this value. When $f$ is positive, the commander sends his value to the lieutenants, each of whom then executes the algorithm recursively with parameter $f-1$ with himself as the commander and the remaining lieutenants as lieutenants. Each lieutenant decides on the majority value among the value received directly from the commander, and the values on which he decides as a lieutenant of the other lieutenants when they act as commanders. In order to separate messages from the different executions of the algorithm, each message contains a value and the sequence of ids of the lieutenents through which it has passed. When according to the algorithm a process should receive a message but doesn't do so, which means that the sender exhibits faults, it assumes the default value instead.

In [34] this algorithm is indicated by $\mathtt{OM(f)}$ (for oral messages, as they can be forged). Below we will indicate by $\mathtt{OM}(\mathtt{f},\mathtt{v},\mathtt{i}_1,\mathtt{i}_2,\ldots,\mathtt{i}_\mathtt{k})$ the algorithm with $\mathtt{f}$ as the maximum number of failing processors, $\mathtt{v}$ the initial value in the commander, $\mathtt{i}_1$ the index of the commander, and $\mathtt{i}_2,\ldots,\mathtt{i}_\mathtt{k}$ the indices of the lieutenants.

**Implementation:**

```
I. Code executed by the commander in OM(f)
broadcast(v)

II. Code executed by the lieutenants in OM(0).
receive(v)
order ← v

III. Code executed by lieutenant $L_i$ in OM(f)
if receive(v) then
    $v_i$ ← v
else
    $v_i$ ← default
$OM(f-1, v_i, i, 1, \ldots, i-1, i+1, \ldots, n-1)$
$v'_j$ ← order in $L_i$ of $OM(f-1, v_j, j, 1, \ldots, j-1, j+1, \ldots, n-1)$
order ← $majority(v'_1, \ldots, v'_{i-1}, v_i, v'_{i+1}, , \ldots, v'_{n-1})$
```

**Correctness:** See [34].

**Complexity:** In the course of the execution of OM(f), OM(f-1) is executed $n-1$ times, once for every lieutenant acting as commander to the other lieutenants. As a consequence, in the course of the execution of OM(f), OM(k) is executed $(n-1)(n-2)\cdot\cdots\cdot(n-f+k)$ times. The total number of messages (or rather maximal number when some failing processes do not send some message) sent in this algorithm is $(n-1)+(n-1)(n-2)+\cdots+(n-1)(n-2)\cdots(n-f-1)$, which is of order $O(n^{f+1})$. □

We now turn to the algorithm for consensus in synchronous systems with authentication.

**Algorithm 5.7** *The Lamport-Pease-Shostak algorithm for consensus with authentication in synchronous systems with a completely connected network* [34].

**Idea:** In this algorithm, messages are signed by the original sender, and by each of the lieutenants who receives the message and sends it along. Signatures cannot be forged, and modification of a signed message can be detected. In the algorithm, a message with contents `v` and signed by first the commander (process 0) and subsequently by lieutenants $\mathtt{i}_1, \ldots, \mathtt{i}_k$ is denoted by $[\mathtt{v}, 0, \mathtt{i}_1, \ldots, \mathtt{i}_k]$. In code fragment II below, `s` denotes a string of signatures, and `len(s)` its length. It is assumed that there are at most `f` faulty processors, and lieutenants wait until they have received a message for all possible strings of signatures of length `f+1` (with signature `0` as the first), which is the meaning of the "do long enough" construct. If such a message is not received, the default for its contents is assumed (not shown in the algortithm). The lieutenants maintain an initially empty set $V$ of all orders they receive. We assume the same `choice` function in all lieutenants that picks some value from the final set $V$ (e.g., the majority if it exists and otherwise the default, or the minimum if the set of possible values is ordered).

**Implementation:**

I. Code executed by the commander

```
broadcast([v,0])
```

II. Lieutenant $L_i$:

```
V ← ∅
do long enough
    upon receipt of [v,0,s] do
        if (v ∉ V) then
            V ← V ∪ {v}
            if (len(s) < f) then
                for j=1 to i-1,i+1 to n-1, not in s do
                    send([v,0,s,i]) to $L_j$
order ← choice(V)
```

**Correctness:** See [34]. □

Algorithm 5.6 is very inefficient, it requires an exponential number of messages. We will now present an algorthm that only requires a polynomial number of messages.

**Algorithm 5.8** *The Srikanth-Toueg algorithm for consensus with authenticated broadcast in synchronous systems with a completely connected network* [58].

**Idea:** This algorithm supposes that there is a commander that starts with some binary value $v$ that it wants to transmit to all other processes, which start with there own decision variable $v$ initialized to 0. The algorithm consists of two layers, and uses simulation **??**. The bottom layer simulates digital signatures by implementing what is called authenticated broadcast, while the upper layer does the actual consensus algorithm. In the explanation below the primitives of the lower layer are indicated by `a-broadcast` and `a-receive`.

To start with the upper layer, it proceeds in $f + 1$ rounds In the first round, if the general is correct and it has $v = 0$, it will never `a-broadcast` a message; if in that case it has $v = 1$, it `a-broadcasts` the value 1 in round 1. The other processes send at most one message, namely a message with the value 1 in the first round after they have set their own value to 1. They set their value to 1 in round $r$ if they have `a-received` in all rounds up to and including ropund $r$ the value 1 from at least $r$ processes, including the general.

The most important requirement for the lower layer is that in addition to taking care of all, the message traffic among correct processes, they all have the same information from faulty processes. In particular, a faulty process may send a signed message to some but not all correct processes. The requirements for the authenticated broadcast layer are the following:

1. (correctness) A broadcast by a correct process in a certain round is accepted by all correct processes in the same round.
2. (unforgeability) No correct process ever accepts a message from another correct process that it did not actually send.
3. (relay) If a correct process accepts a broadcast by a process in some round, every other correct process accepts that broadcast at the latest in the next round.

**Correctness:**

## 5.4 Paxos Consensus

In this section we will deal with a form of consensus that has come to be known as Paxos consensus, after the original article [35] in which it was proposed. There are two variations of this problem, the *single-value Paxos* problem, in which a group of processes has to agree on a single value as proposed by one of the participants, and the *multi-value Paxos* problem, in which a group of processes has to agree on a sequence of such values. In both problem statements, the system is asynchronous and processes may fail by crashing and possibly later restarting—which amounts to pausing, potentially forever. Processes are not malicious, their only possible faulty behavior is pausing. In addition, processes have access to stable storage on which they store the variables they need to rejoin the system. Messages may be reordered, duplicated, and lost, but they cannot be corrupted.

Single-value Paxos models straightforward consensus by a set of processes on a value proposed by one of them. In multi-value Paxos, the processes need to agree on a sequence of values. This problem is used to model *state machine replication* (SMR) in which a set of servers maintain replicas of a state (e.g., the contents of a data store) to which clients issue commands (e.g., read and write operations). In order for the states at the differerent replicas to perform the same state transitions and to give the same replies to the clients, they have to execute the same commands of the clients in the same order. That is, the servers have to agree on the commands of the clients and on their order, in the face of the type of failures as stated above. As to the traditional three requirements for solutions to consensus problems, agreement here means that the processes reach consensus on a single value or on each of the values in sequence, and validity in both cases means that each of the values on which consensus is reached was proposed by one of the participants. Termination can of course not be guaranteed in view of the FLP result.

Rather than having a single type of processes, the problem (and its solutions) is sometimes cast in terms of three types of processes. *Proposers* are the processes that each start with a value that they may propose, *acceptors* are processes that accept and decide on a value, and *learners* are processes that learn about the final outcome. Actually, a single process may play all three roles. In the course of the algorithms below, a proposer may be forced to propose a value that is not his own original value. Therefore, sometimes *coordinator* or *leader* processes are introduced that propose values on behalf of the proposers. Below we retain the terminology of "proposers."

**Algorithm 5.9** *Lamport's single-value Paxos consensus algorithm* [35].

**Idea:** A single proposal for a value may not lead to consensus, so multiple proposals (possibly even for the same value, by possibly different proposers) may have to be performed. The algorithm is structured (but does not necessarily exactly proceed) in rounds, each carrying a unique sequence number and each initiated by some proposer. Round numbers can be made to be unique by either preassigning disjoint sets of integers to the proposers (e.g., all round numbers that modulo the number of processes is equal to the id of a proposer) or by using pairs of integers and ids. Rounds can be executed simultaneously, out of order with respect to round numbers, and can even be skipped altogether.

A round consists of two phases. In phase 1, a proposer attempts to build a majority of acceptors willing to accept a proposal on its part *w*ithout already settling on the actual value to be proposed with a REQUEST_TO_PARTICIPATE message to some set of processes (code fragment I). When an acceptor receives such a message and the round number associated with it is new to it, it responds with a PARTICIPATE message that includes its latest vote (if any) and the round number of that vote (code fragment II).

In phase 2, a proposer selects the most recent vote from the votes of a majority of the acceptors, and proposes the corresponding value to (a subset of) the acceptors with a REQUEST_TO_VOTE message (code fragment III). When none of the acceptors in the responding majority has voted sofar, the proposer is free to pick any value to propose. When an acceptor receives a REQUEST_TO_VOTE message, it votes in favor of it by sending a VOTE message to the learners if it has not in the mean time voted for a yet more recent proposal (code fragment IV). A round, and indeed the algorithm, has completed successfully when at least one learner has learnt a value by receiving votes for it from a majority of the acceptors.

As to the data structures of the processes, an acceptor maintains two round numbers, `rnd` of the highest-numbered round in which it has participated and `vrnd` of the highest-numbered round in which it has voted, and the value `vval` it voted for in the latter round. A proposer maintains the highest-numbered round `prnd` it has initiated, and the value `pval` it proposes in phase 2 of that round (or nil). Processes record in stable storage the current values of these variables.

Some remarks on this algorithm are in order.

1. Because of the failure assumptions, whenever a process is supposed to send a message according to the algorithm, it may refrain from doing so. In particular, an acceptor may abstain from voting even when it could do so.

2. All messages pertaining to a certain round are identified by the number of the round to which they belong (not included in the code fragments below). As soon as a process receives a message with a round number larger than any it has seen in sofar, it skips sending any message related to previous rounds.

3. In phases 1 and 2, a proposer exchanges messages with sets of acceptors. These sets may contain all acceptors, or a subset of them, e.g., a subset containing a majority of acceptors. In the end, the only thing that matters about these sets is that a proposer finds a majority of the acceptors willing to participate, and that at least one learner receives a VOTE message from a majority of the acceptors in some round. In fact, a proposer may in phase 2 send a REQUEST_TO_VOTE to an acceptor that has not indicated its willingness to participate in phase 1.

4. The condition in code fragment IV constitutes a promise of an acceptor voting in a certain round that it will not do so anymore in a previous round (i.e., in a round with a lower round number).

5. Livelock is possible in this algorithm (see Exercise ..). To avoid it, an election algorithm may be used to elect a distinguished proposer (or rather, a coordinator or leader) who is the only one initiating proposals. Unless and until this proposer fails, progress is guaranteed. If it fails, a new process has to be elected.

**Implementation:**

I. A proposer does a REQUEST_TO_PARTICIPATE.

```
i ← new round number
prnd ← i
pval ← nil
send(REQUEST_TO_PARTICIPATE,i) to some set of processes
```

II. An acceptor receives a REQUEST_TO_PARTICIPATE message.

```
upon receipt of(request_to_participate,i) do
    if (i>rnd) then
        rnd ← i
        send(participate,vrnd,vval) to proposer
```

III. A proposer does a REQUEST_TO_VOTE.

```
upon receipt of(participate,*,*) from majority Q do
    round ← max{vrnd} in messages from Q
    if (round=0) then pval ← any value
    else
        pval ← vval in messages from Q with vrnd=round
    send(REQUEST_TO_VOTE,i,pval) to processes in Q
```

IV. An acceptor receives a REQUEST_TO_VOTE message in round i.

```
upon receipt of(REQUEST_TO_VOTE,i,v) do
    if ((i ≥ rnd) and (vrnd ≠ i)) then
        rnd ← i
        vrnd ← i
        vval ← v
        send(VOTE,i,v) to the learners
```

**Correctness:** We will not prove correctness of the algorithm here. Correctness proofs can be found elsewhere [35]. The essential element in the correctness of the algorithm is that the selection of the actual value to be proposed in phase 2 in code fragment III guarantees consistency, which we will now informally show. In fact, we will show that in no previous round another value has been or still can be decided (if other rounds are still also active), where we follow the proof in [37].

So suppose pval=v is the value proposed in phase 2 of round $i$ as the most recent vote cast in round $k$ by any the acceptors in the quorum $Q$ of round $i$, with $k < i$. Let $j$ be any round number with $j < i$. There are three cases.

In case $k < j < i$, if a is an acceptor in $Q$, it could not have voted in round $j$ before sending its PARTICIPATE message to the proposer in phase 1 of round $i$. As it promises not to vote in lower-numbered rounds, it will not vote in round $j$ at all, as it votes in round $i$ first. So no acceptor in the majority $Q$ will ever vote in round $j$. As a consequence, in round $j$ no decision can ever be reached.

In case $j = k$, the one and only proposal in that round is v, and so no other value can be decided.

In case $j < k$, the statement of consistency holds by induction, with round $k$ taking the place of round $i$.

## 5.5 Randomized Solutions

For some problems in distributed systems it is useful, or indeed necessary, to introduce a random element, that is, to allow one or more processes to flip a coin once in a while to make progress towards a solution. Randomization may achieve more efficient solutions than deterministic solutions, for instance by reducing the (expected) number of rounds in synchronous Byzantine agreement. Randomization may also achieve a solution for problems for which no deterministic solution exists, such as such as for Byzantine agreement in asynchronous systems, and for election in an anonymous unidirectional ring of a known size.

With randomization in distributed algorithms, something has to be sacrificed. Either the algorithm may always terminate but the result it produces may be correct only with some (high) probability, it may terminate with some (high) probability with a correct result, or it may only both terminate and produce a correct result with some (high) probabitities.

### 5.5.1 Randomized Agreement with Crash Failures

We will first present a randomized distributed algorithm for crash failures, which requires a strict majority of non-faulty processes. This algorithm works in both synchronous and asynchronous systems. The termination condition now specifies that the *expected* number of rounds, which are imposed by the algorithm, is finite.

**Algorithm 5.10** *Ben-Or's randomized algorithm for consensus in synchronous and asynchronous systems with crash failures* [5].

**Idea:** Every process starts with a binary input value `v`. We assume that the number of traitors $f$ satisfies $n > f/2$. The algorithm proceeds in rounds (indicated by `r`) consisting of three phases: a *notification* phase (messages contain the messagwe type `N`), a *proposal* phase (messages contain the message type `P`), and a *decision* phase. When a process expects messages from all processors, it is no use waiting for more than $n - f$ messages, because the $f$ faulty processors may not send the required message. When not enough processors support a possible decision, a process starts the next round with a new, random value `v`. Even though every process goes through a series of rounds, the algorithm is suitable for asynchronous systems—the rounds do impose a level of synchrony.

**Implementation:**

```
r ← 1
decided ← false
    do forever
        broadcast (N;r,v)
        await $n-f$ messages of the form (N;r,$\star$)
        if ($> n/2$ messages (N;r,w) received with w=0 or 1) then
            broadcast (P;r,w)
        else broadcast (P;r,?)
        if decided then STOP
        else await $n-f$ messages of the form (P;r,$\star$)
        if ($>1$ messages (P;r,w) received with w=0 or 1) then
            v ← w
            if ($> f$ messages (P;r,w)) then
                decide w
                decided ← true
        else v ← random(0,1)
        r ← r + 1
```

### 5.5.2 Randomized Byzantine Agreement

In this section we present a randomized algorithm for Byzantine agreement which is valid for both synchronous and asynchronous systems, and which is in fact very similar to Algorithm 5.10.

**Algorithm 5.11** *Ben-Or's randomized algorithm for consensus in synchronous and asynchronous systems with Byzantine failures* [5].

**Idea:** Compared to Algorithm 5.10, this algorithm has the following differences. First, the requirement on the number of traitors $f$ is now $n > 5f$. Secondly, the decision procedure is more complicated.

**Implementation:**

```
r ← 1
decided ← false
    do forever
        broadcast (N;r,v)
        await $n-f$ messages of the form (N;r,$\star$)
        if ($> (n+f)/2$ messages (N;r,w) received with w=0 or 1) then
            broadcast (P;r,w)
        else broadcast (P;r,?)
        if decided then STOP
        else await $n-f$ messages of the form (P,r,$\star$)
        if ($> f$ messages (P;r,w) received with w=0 or 1) then
            v ← w
                if ($> 3f$ messages (P;r,w)) then
                    decide w
                    decided ← true
        else v ← random(0,1)
        r ← r + 1
```

**Correctness:** We prove the correctness of this algorithm with the following lemmas. The correctness hinges on the three conditions for certain numbers of messages received for taking some action in the algorithm. Note that by Lemma 5.13, the algorithm achieves validity in that if all correct processes start with the same value, they all decide on this value. Also, by Lemma 5.14 agreement is achieved in that all correct processes decide on the same value.

**Lemma 5.12** *If a correct process proposes* `v` *in round* `r`*, then no other correct process proposes* `1-v` *in round* `r`*.*

PROOF. For a process to propose the value `v`, the process must have received more than $(n+f)/2$ messages of the form `(N,r,v)`. Of these, more than $(n-f)/2$ are from correct processes, which is a majority of the correct processes. □

**Lemma 5.13** *If at the beginning of round* `r` *all correct processes have the same value* `v`*, then they all decide* `v` *in round* `r`*.*

PROOF. Each correct process will receive at least $n-f$ messages, at least $n-2f$ of which are from correct processes, and so of the form `(N,r,v)`. Because $n > 5f$, we have $n{-}2f = n/2{+}n/2{-}2f > (n+f)/2$, and so, each correct process proposes `v`. So, each correct process receives at least $n-2f$ messages of the form `(P,r,v)`, and so, because $n > 5f$, we have $n-2f > 3f$, and so each correct process decides `v`. □

**Lemma 5.14** *If a correct process decides* `v` *in round* `r`*, then all correct processes decide* `v` *in round* `r+1`*.*

PROOF. It is enough to show that all correct processes propose `v` in round `r+1`. If a correct process decides `v` in round `r`, it must have received more than $3f$ proposals for `v`, $m$ of which are from correct processes for some $m > 2f$. So every other correct processor receives at least $m-f > f$ proposals for `v`, so it starts the next round with this value. Now use Lemma 5.13. □

**Theorem 5.15** *If* $n > 5f$*, Algorithm 5.11 guarantees Agreement and Validity, and terminates with probability 1.*

PROOF. With probability 1, enough correct processes will pick a common value v to have at least one correct process decide. □

The expected number of rounds of this algorithm is exponential in the number of processes: When it is needed that the (correct) processes pick a random number, the probability that they all draw the same number (bit) is equal to $2 \cdot 2^{-n}$. Because it does not matter what value the failing processes pick, and because not all values have to be equal, the time complexity is actually better, but still exponential.

### 5.5.3 Randomized Coordinated Attack

In this section we will discuss a consensus problem in the face of message losses (but correct processes) in which randomization is used to achieve a *probabilistic* form of agreement. We consider a synchronous system and a complete network among the processors, and we fix the number of rounds $r$ that the algorithm will run, thus achieving termination in a trivial way. The validity and agreement conditions are now:

- *Validity*:
    - If all processes start with a 0, then all processes decide 0.
    - If all processes start with a 1 and all messages are received, then all processes decide 1.
- *Agreement*: The probability of disagreement, that is, of two processes deciding on different values, is bounded from above by $\epsilon$, for some $0 < \epsilon < 1$.

In the algorithm below, every process sends a message (with all current information it has) to every other process in every round. However, in every round a random subset of the links may fail, and this subset may be different in every round. (In [66] this is modeled with an *adversary*.) In the algorithm, processes keep track of their *level*. All processes start at level 0. If a process has received a message from all other processes indicating that they are at say level $l$, the process advances to level $l + 1$. In particular, if no links exhibit failures, then all processes will reach level $l$ at the end of round $l$.

**Algorithm 5.16** *The randomized coordinated-attack algorithm of Varghese and Lynch* [66].

**Idea:** First of all, a special process, say $P_1$, picks a random number $K$ uniformly distributed on $\{1, 2, \ldots., r\}$ before the algorithm starts. Intuitively, $K$ is a guess as to the level the processes will reach when the algorithm terminates. Every process $P_i$ sends every other process in every round a message of the form $(L, V, k)$, with

- $L$ a vector of length $N$ (the number of processes in the system) indicating the levels of all processes as far as known by $P_i$ when it sends the message; $L$ is initialized to all zeroes;
- $V$ a vector of length $N$ indicating the initial values of all processes as far as known by $P_i$; $V_i[i]$ is initialized to the initial value of $P_i$, the other elements of $V_i$ are initially undefined;

- $k$ is the value of $K$ picked by $P_1$ if it is known by $P_i$; it is initially undefined.

In every round, every process receives all messages from all processes in that round that are sent along links that do not fail in that round. It then updates its vector $V$ with any values from the received vectors $V$ for which it still had the value undefined, and it does the same for $k$ (each of these updates has to be done only once). In addition, it sets the level of each other process to the maximum of the levels of that process as reported by the other processes in their $L$ arrays and its own current value of the level of that process. It then computes its new level based on the levels of the other processes as $1 + \min_{j \neq i}\{L_i[j]\}$, where $i$ is the process index.

At the end of round $r$ a process $P_i$ uses the following decision rule. If the value of $k$ is not undefined anymore in $P_i$, the level of $P_i$ is at least equal to $k$, and the initial values of all processes are known in $P_i$ and are all equal to 1, then $P_i$ decides 1. In all other cases, $P_i$ decides 0.

**Complexity:** First of all, it can be shown that at the end of any round, the levels of the processes can differ by at most 1 (see [66]). Let $l_i$ be the level of $P_i$ at the end of round $r$, let $m = \min_i\{l_i\}$, and $n = \max_i\{l_i\}$. Then either $n = m$ or $n = m + 1$, and according to the decision rule used by the processes, disagreement is only possible when $K = n$. Because the value of $K$ is picked at random by $P_1$ and $n$ can have any value between 0 and $r$, this has probability $1/r$. □

## 5.6 Stabilization

Components in distributed systems may only fail temporarily instead of permanently. For instance, a single message may not be received correctly, or a short power outage may cause a processor to go down and to come up soon after. Whereas with permanent faults, the components of a computer system that are functioning correctly have to "program around the faulty components," with transient faults the aim is to have the system re-enter a consistent configuration with all components functioning correctly within finite time. For dealing with transient faults, *stabilizing* algorithms have been devised which as a side effect of their operation bring a distributed system back into a correct state from any incorrect state it may be in.

### 5.6.1 Definitions

Stabilizing algorithms deal with faults implicitly, in the sense that by simply running them, the system returns to its correct behavior. In order to say what "functioning correctly" means, we define legal and illegal configurations. The former are defined as the states in which some predicate holds. These predicates give a property of the system in terms of the values of variables in the processes. We first give two examples of these predicates.

**Example 5.17** In Algorithm 4.8 for mutual exclusion, we can, for any $t$ with $0 \leq t \leq n$, consider the predicates

$$P(t) = (\#\{i \mid \texttt{token\_present}_i = \texttt{true}\} = t).$$

In token-based mutual-exclusion algorithms, we want the system to reach a state in which $P(0) \vee P(1)$ holds. □

**Example 5.18** Consider a mutual-exclusion algorithm. We can add a local variable `in_CS` in each process, which is set to `true` at the beginning of the execution of the CS of each process, and which is reset to `false` at the end of the CS. We can then consider the predicate $P$ defined as

$$P = \#\{i | \texttt{in\_CS}_i = \texttt{true}\} \leq 1.$$

□

In general, the predicate defining the legal states may be very complicated to state formally.

A predicate on system configurations (a "property") is called stable if once it holds and no faults occur, it continues to hold. Stable predicates have also been called closed predicates.

**Definition 5.19** *Let $P$ and $Q$ be two stable predicates on the configurations of a system $S$. An algorithm* $Q$**-stabilizes from** $P$ **in** $S$ *if from any configuration in which $P$ holds, within a finite number of steps, $S$ is in a configuration in which $Q$ holds.* □

As a special case, if $P \equiv \texttt{true}$, that is, if $P$ is the predicate that holds in any system configuration, then the system has to reach a configuration in which $Q$ holds starting from any configuration. In this case we simply say that the algorithm $Q$-stabilizes, or stabilizes if $Q$ is clear from the context. By choosing other predicates $P$ than `true`, one can restrict the types of faults that the stabilizing algorithm has to deal with. Instead of saying that an algorithm $Q$-stabilizes from $P$, it is sometimes said that the algorithm **converges** from $P$ to $Q$. In fact, it is not quite necessary to require $P$ and $Q$ to be stable, but because we will often assume that $P \equiv \texttt{true}$ and because the notion of stabilization seems to imply that $Q$ is stable, this requirement is not much of a restriction.

Stabilizing algorithms are very sensitive to the assumptions made about the distributed systems in which they must operate. We now discuss the two of these.

### Communication

We will consider two models of communication among processors, the shared-memory model and the message-passing model.

In the shared-memory model, in addition to having *local* registers, processors communicate through sets of *non-local* registers. Each register has associated sets of processors that can read and that can write it. Processors execute *steps*, which consist of computations and of read and write operations. We say that we have *read/write atomicity* when a step consists of a local computation preceded by a single read operation or followed by a single write operation. In *composite atomicity*, a step consists of reading a set of registers, doing some local computations, and writing a set of registers.

In the message-passing model, if processor $p$ is connected by a datalink to processor $q$, there is a message buffer $B_{pq}$ containing the set of messages sent by $p$ but not yet received by $q$. If the datalink from $p$ to $q$ has the FIFO property, then $B_{pq}$ is actually a queue.

In general, it is easier to design stabilizing algorithms for the shared-memory model than for the message-passing model. The reason is that in the latter, we have to assume that corruption of the queue of messages from one processor to another can cause the queue to have arbitrary contents. That is, such a queue may contain messages whose contents are corrupted—which is similar to the corruption of registers in the shared-memory model—but also messages that have not been sent, and it may not contain messages that have been sent. Indeed, if a message queue is implemented as a buffer with pointers to the start and the end of the message queue, corruption of one or both of these pointers does have this effect. Rather than design stabilizing algorithms for all kinds of tasks, it is better to design a stabilizing algorithm for a single link in the message-passing model, and use it as a building block for other algorithms (see Section 5.6.4).

#### Demons

Some stabilizing algorithms only function correctly when only one process at a time takes a step, while in others multiple processes may do so simultaneously. This leads to the notion of demons. In a system (algorithm) with a *central demon*, this demon picks one process at a time to make a step. Only when this process has completed its step, the demon picks another process to make a step. The demon may pick a process at random with equal probabilities, or it may pick processes in a specific order, although we assume that the demon is fair in the sense that in every infinite execution of the system, every process is picked infinitely often.

In a system (algorithm) with a *distributed demon*, the demon picks a set of processes to make a step. Then, all processes in this set make their step in a synchronous fashion. That is, with composite atomicity, they first all read registers, then they perform their local computations, and finally they write registers. Only after all of the processes in the set have completed their step does the demon pick another set. Here, the demon is fair if in every infinite execution of the system, every process belongs to the set of activated processes infinitely often.

### 5.6.2 Stabilizing Mutual-Exclusion Algorithms

The first stabilizing algorithms ever to be published are a set of three algorithms for mutual exclusion in a ring network by Dijkstra [17]. These algorithms can also be regarded as algorithms for passing a single token along the ring in which both the loss of the token and the existence of multiple tokens are corrected (but not detected!). We start with the algorithm that needs (about) as many possible states as there are processors.

**Algorithm 5.20** *Dijkstra's stabilizing mutual-exclusion algorithm for a unidirectional ring in the shared-memory model with a central or a distributed demon* [17].

**Idea:** The network structure is a unidirectional ring with $N$ processes $P_0, P_1, \ldots, P_{N-1}$, in which a link exists from $P_n$ to $P_{n+1}$ for $n = 0, 1, \ldots, N-1$ (process indices are taken modulo $N$). Process $P_i$ maintains only one variable $\mathtt{v_i}$, which is an integer modulo $K$ for some positive value $K$. In a single step, a process $P_n$ reads $\mathtt{v}_{n-1}$ and $\mathtt{v}_n$, compares them, and possibly assigns a new value to $\mathtt{v}_n$. When a process $P_n$ with $n \neq 0$ finds $\mathtt{v}_n \neq \mathtt{v}_{n-1}$, it sets $\mathtt{v}_n$ to $\mathtt{v}_{n-1}$. When process $P_0$ finds $\mathtt{v}_0$ to be equal to $\mathtt{v}_{N-1}$, it assigns to $\mathtt{v}_0$ the value $\mathtt{v}_0 + 1 \bmod K$. In case of a central (distributed) demon, we require $K \geq N-1$ ($K \geq N$).

The legal configurations of the system are defined by the predicate

$$(\mathtt{v}_0 = \mathtt{v}_{N-1}) \quad \mathbf{XOR} \quad (\#\{n \mid 0 < n < N, \mathtt{v}_n \neq \mathtt{v}_{n-1}\} = 1).$$

In the usual formulation of the mutual-exclusion problem, processes are allowed to remain outside of their CS as long as they wish, and are not required to request access to their CS at all. In this algorithm however, processes do have to request their CS over and over again, otherwise they will block the other processes.

**Implementation:**

```
I. A step in $P_0$
if $(\mathtt{v}_0 = \mathtt{v}_{N-1})$ then
   Critical Section
   $\mathtt{v}_0 \leftarrow \mathtt{v}_0 + 1 \bmod K$
```

```
II. A step in $P_n, n = 1, \ldots, N-1$:
if $(\mathtt{v_n} \neq \mathtt{v_{n-1}})$ then
   Critical Section
   $\mathtt{v_n} \leftarrow \mathtt{v}_{n-1}$
```

**Correctness:** To show stabilization, consider an execution of the algorithm, starting from any state, in which no corruption of the state of the processes occurs. Let $V$ be the (time-varying) set of the values of the variables $\mathtt{v}_n, n = 0, 1, \ldots, N-1$.

Let's first consider the case with a central demon. Between any two subsequent steps of any process $P_n$, process $P_{n+i}, i = 1, 2, \ldots, N-1$ can take at most $i$ steps, for at most a total of $N(N-1)/2$ steps. This means that in an infinite execution of the algorithm, every process takes infinitely many steps. Because $P_0$ can only do a step when $\mathtt{v}_0 = \mathtt{v}_{N-1}$, immediately before that step $|V| \leq N-1 \leq K$. In addition, $P_0$ is the only process capable of introducing a new element in $V$, as the other processes only copy existing $\mathtt{v}$-values. As $P_0$ goes through the values $0, 1, \ldots, K-1$ cyclicly, we conclude that within $K$ steps of $P_0$, it does introduce a new value $\mathtt{v}'$ into $V$. Now $P_0$ can only do its next step when $\mathtt{v}_{N-1} = \mathtt{v}'$, which is only possible when $\mathtt{v}_n = \mathtt{v}'$ for $n = 1, 2, \ldots, N-1$. But this means that the system is in a legal state.

For the proof in case of a distributed demon, we refer the reader to [19]. □

The technique used above, where periodically the special process introduces a new value in the system, has been called the *missing-label* technique or *counter flushing* [65].

**Example 5.21** (Central demon) We will show that Algorithm 5.20 stabilizes when $N = 4$ and $K = 3$ in the case of a central demon. Because in every infinite execution, every process has to make an infinite number of steps, we can consider the system at a point in its execution when process $P_0$ is about to make a step. Without loss of generality, we can assume that then $\mathtt{v}_0 = \mathtt{v}_3 = 0$. Because process $P_0$ goes cyclically through states $0, 1, 2$, we have to assume that all possible states occur in the system, for otherwise, at some future point, $P_0$ will introduce a new state, and the system will certainly stabilize. So either $\mathtt{v}_2 = 1$ and $\mathtt{v}_1 = 2$, or $\mathtt{v}_2 = 2$ and $\mathtt{v}_1 = 1$, that is, neither $\mathtt{v}_1$ nor $\mathtt{v}_2$ is equal to 0. When now $P_0$ does a step, it will set $\mathtt{v}_0$ to 1, leaving $\mathtt{v}_3$ as the only 0 in the system. Process $P_0$ can only do its next step when $\mathtt{v}_3 = 1$. But this means that in a few steps, the value 0 will have disappeared from the system. Then when $P_0$ re-introduces 0 in the system, this will be a unique value, and the system will stabilize. □

**Example 5.22** (Central demon) We will show that Algorithm 5.20 may not stabilize when $N = 5$ and $K = 3$ in the case of a central demon. Suppose that the system is initially in State 1 as in Table 5.1. Then, when the processes take steps in the order as in the last column, the resulting state 6 is equivalent modulo 3 to state 1. So always repeating this order of processes that take a step will keep the system in an illegal state forever. □

| state | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | step by |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 2 | 1 | 0 | $P_0$ |
| 2 | 1 | 0 | 2 | 1 | 0 | $P_4$ |
| 3 | 1 | 0 | 2 | 1 | 1 | $P_3$ |
| 4 | 1 | 0 | 2 | 2 | 1 | $P_2$ |
| 5 | 1 | 0 | 0 | 2 | 1 | $P_1$ |
| 6 | 1 | 1 | 0 | 2 | 1 | $P_0$ |

Table 5.1: Successive states in the system in Example 5.22.

| state | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $\ldots$ | $P_{N-2}$ | $P_{N-1}$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | $N-2$ | $N-3$ | $N-4$ | $\ldots$ | 1 | 0 |
| 2 | 1 | 0 | $N-2$ | $N-3$ | $\ldots$ | 2 | 1 |
| 2 | 2 | 1 | 0 | $N-2$ | $\ldots$ | 3 | 2 |
| ............ | | | | | | | |
| $N-1$ | $N-2$ | $N-3$ | $N-4$ | $N-5$ | $\ldots$ | 0 | $N-2$ |
| $N$ | 0 | $N-2$ | $N-3$ | $N-4$ | $\ldots$ | 1 | 0 |

Table 5.2: Successive states in the system in Example 5.23.

**Example 5.23** (Counter-example with a distributed demon) We now show an execution of Algorithm 5.20 with $K = N - 1$ and with a distributed demon that does not stabilize. Let $\mathrm{v}_0 = \mathrm{v}_{N-1} = 0$, and let $\mathrm{v}_n = N - n - 1$, for $n = 1, 2, \ldots, N - 2$ (see Table 5.2, this is state 1). In this state, every process can do a step. If all processes do so simultaneously, the system ends up in state 2 in Table 5.2, in which again all processes can do a step. This procedure can be continued until after $N - 1$ steps of every process, state 1 recurs. □

### 5.6.3 Fair algorithm composition

When for a complicated problem a stabilizing algorithm has to be devised, one can try to split up the problem in parts, write stabilizing algorithms for those parts, and try to compose those algorithms to achieve a stabilizing algorithm for the original problem. *Fair algorithm composition* does precisely that in a sense that is reminiscent of the layering approach in data networks. So consider two tasks $T_1$ and $T_2$, and suppose that $P_1$ is a stabilizing protocol for $T_1$ and $P_2$ is a stabilizing protocol for $T_2$ given $T_1$. The fair combination of protocols $P_1$ and $P_2$ is a protocol that alternatingly executes steps of both protocols. This fair combination is now stabilizing for $T_2$.

### 5.6.4 Stabilizing datalink algorithms

In order to derive stabilizing algorithms for message-passing systems, one can use the results of Section 5.6.3 to first design a stabilizing algorithm for message passing along a datalink, and then use a shared-memory stabilizing algorithm for the original problem (mutual exclusion, election, etc). In this section we will consider two stabilizing datalink algorithms, viz. a stop-and-wait algorithm, and an adapted version of a sliding-window protocol.

**Algorithm 5.24** *A stabilizing stop-and-wait datalink algorithm.*

**Idea:** The straightforward stop-and-wait algorithm, in which for every message the sender waits for an acknowledgment before sending the next message, is stabilizing.

**Implementation:**

```
I. Sending a message
upon reception of (r_counter) do
    if (r_counter ≥ s_counter) then
        s_counter ← s_counter + 1
    send(message[s_counter]; s_counter)

II. Receiving a message
upon receipt of (message; s_counter) do
    if (s_counter ≠ r_counter) then
        process(message)
        r_counter ← s_counter
    send(r_counter)

III. Timeout in the sender
upon timeout do
    send(message[s_counter]; s_counter)
    □
```

We now present a two sliding-window protocols for datalinks, the first of which does not stabilize.

**Algorithm 5.25** *A non-stabilizing sliding-window datalink algorithm* [24].

**Idea:** Let the window size in the protocol be $w$. The sender maintains counters ns and na for the number of the next message to be sent and for the number of the next message to be acknowledged, respectively. The receiver maintains a counter nr for the number of the next message to be received. Messages sent by the sender carry the message number (code fragment I below). When the receiver sends an acknowledgment (code fragment IV below), it sends its current value of nr, which means that it acknowledges the correct receipt of messages up to, but not including, message number nr. Upon reception of an acknowledgment by the sender with a higher value than na, the sender sets na to this higher value (code fragment II below). Upon a time out in the sender, it simply resends any messages sent but not yet acknowledged (code fragment III).

The predicate $P$ to which the algorithm should stabilize is (S and R denote the sender and receiver):

$$((\mathtt{na} \leq \mathtt{nr}) \textbf{ and } (\mathtt{nr} \leq \mathtt{ns}) \textbf{ and } (\mathtt{ns} \leq \mathtt{na} + \mathtt{w}))$$

$$\textbf{and}$$

$$\textbf{for each } (\mathtt{message}; \mathtt{i}) \text{ in channel SR } (\mathtt{i} < \mathtt{ns})$$

$$\textbf{and}$$

$$\textbf{for each } (\mathtt{ack}; \mathtt{i}) \text{ in channel RS } (\mathtt{i} \leq \mathtt{nr})$$

Here the first clause means that the number of the next message to be acknowledged cannot exceed the number of the next message to be received, that the number of the next message to be sent has to be at least equal to the number of the next message to be received, and that the numbers of the next message to be sent and acknowledged cannot differ by more than the window size. The second clause states that the number of the next message to be sent exceeds the message numbers of the messages on the channel from the sender to the receiver. Finally, the third clause says that no acknowledgment on the channel from the receiver to the sender carries a higher number than the number of the next message to be received.

**Implementation:**

```
I. Sending a message
if ($ns < na + w$)) then
    send(message[ns];ns)
    ns ← ns+1

II. Receiving an acknowledgment
upon receipt of(ack;i) do
    if ($i > na$) then na ← i

III. Timeout in the sender
upon timeout do
    if ($ns > na$) then
        for $i = na, na + 1, \ldots, ns - 1$ do
            send(message[i]; i)

IV. Receiving a message
upon reception of (message; i) do
    if ($i = nr$) then
        nr ← nr + 1
    send(ack; nr)
```

**Correctness:** A problem that can occur with this algorithm is when the sender thinks that the receiver has received more messages than it in reality has. For instance, suppose that $w = 1$, and that $ns = 5$, $na = 4$, and $nr = 3$. Then the only possible sequence of actions is that at a timeout, the sender resends message number 4, in reaction to which the receiver, who is still waiting for message number 3, resends an acknowledgment with number 3 in it, acknowledging all messages up to and including message number 2. Upon receiving this acknowledgment, the sender does not adapt na, and the same sequence of actions will be repeated, and the system is stuck. □

Below is the stabilizing version, in which the sender and receiver exchange more information. In particular, in order to prevent the problem explained above, the sender includes the value of na as a second counter in the messages for the receiver to catch up.

**Algorithm 5.26** *A stabilizing sliding-window datalink algorithm* [24].

**Idea:** The sender and receiver maintain the same counters as in Algorithm 5.25. Reception of an acknowledgment by the sender gives it the opportunity to catch up with the receiver when the acknowledgment number is out of bounds (code fragment II below). Messages sent by the sender now carry two integer counters (code fragment I below). The first of these indicates the message number, and the second has the value of na at the time of sending the message. The second counter can be used by the receiver to catch up with the sender if the sender is running ahead (code fragment IV below). Upon a time out in the sender, the counters ns and na are made consistent if they are not in a way that indicates that the channel is empty. Otherwise, any unacknowledged messages are resent (code fragment III).

The predicate $P$ to which the algorithm stabilizes is now:

$$((\mathtt{na} \leq \mathtt{nr}) \textbf{ and } (\mathtt{nr} \leq \mathtt{ns}) \textbf{ and } (\mathtt{ns} \leq \mathtt{na} + \mathtt{w}))$$
$$\textbf{and}$$
$$\textbf{for each } (\mathtt{message}; \mathtt{i}, \mathtt{j}) \text{ in channel } \mathtt{SR}$$
$$((\mathtt{i} < \mathtt{ns}) \textbf{ and } (\mathtt{j} \leq \mathtt{nr}) \textbf{ and } (\mathtt{j} < \mathtt{ns}))$$
$$\textbf{and}$$
$$\textbf{for each } (\mathtt{ack}; \mathtt{i}) \text{ in channel } \mathtt{RS} \ (\mathtt{i} \leq \mathtt{nr})$$

Compared to the predicate of Algorithm 5.25, only the second clause has been extended with two parts that say that the number of the next message to be acknowledged in the messages sent but not yet received cannot exceed the number of the next message to be received, and is strictly smaller than the number of the next message to be sent.

**Implementation:**

```
I. Sending a message
if ((na ≤ ns) and (ns < na + w)) then
    send(message[ns]; ns, na)
    ns ← ns + 1

II. Receiving an acknowledgment
upon receipt of(ack; i) do
    if ((i > na) and (i ≤ ns)) then na ← i
    else
        if ((i > na) and (i > ns)) then
            na ← i
            ns ← i

III. Timeout in the sender
upon timeout do
    if (na ≠ ns) then
        if (na > ns) then ns ← na
        if (ns > na + w) then na ← ns
        for i = na, na + 1, ..., ns − 1 do
            send(message[i]; i, na)

IV. Receiving a message
upon reception of (message; i, j) do
    if ((i = nr) and (j ≤ nr)) then
        nr ← nr + 1
    else
        if (j > nr) then nr ← j
    send(ack; nr)
```

□

### 5.6.5 Discussion

There are some problems with Algorithm 5.26. First, if the variable containing the window size $w$ gets corrupted, the sender has no way to reconstruct the original value. Of course, it can then pick a random value as the receiver does not (have to) know the value anyway. Secondly, and more importantly, in a distributed system one would like processes to communicate with each other through a transportation layer. But that means that for a communication, a processor has to keep track of the identifier of the machine and of the port (or process) number of destination process. In Algorithm 5.25, there are no provisions for the corruption of these data. Indeed, there the communication is tied to a single link, and so the processor on the other side of the link does not have to be identified.

## 5.7 Bibliographic Notes

In [63], a gentle and clear introduction to fault tolerance and impossibility results is presented. Surveys of stabilization include [53] and [21]. Vol. 7(1), 1993, of *Distributed Computing* is a special issue on stabilization. The book by Dolev [19] deals exclusively with stabilization.

The original version of the Paxos consensus problem [35] introduced it as the problem of the parliament of the fictitious island of Paxos to agree on laws (“decrees”). Members of parliament (“priests”) were allowed to leave and return at will. A much more informal explanation is presented in [36]. The single-value (sometimes calles “single-decree”) Paxos problem is solved by the “synod” algorithm. Our treatment closely follows a gentle derivation of the algorihm [37]. An invariant-based explanation of multi-value Paxos consensus is in [64].

A concise account of consensus algorithms in distributed systems leading up to an explanation of the Blockchain in by Wattenhofer [67].

## 5.8 Exercises

1. Trace the execution of Algorithm 5.6 for seven processes, two of which are faulty.

2. Show that in Algorithm 5.20, $K \geq N$ is necessary. Hint: Consider a ring of $N$ processes with initially $\mathrm{v}_0 = \mathrm{v}_1 = 0, \mathrm{v}_k = k - 1$ for $k = 1, \ldots, K$, and $\mathrm{v}_k = 0$ for $k = K, \ldots, N - 1$. Show that when the processes along the ring take steps sequentially, starting with $P_0$, then $P_1$, etc, the ring will return to its initial state (modulo $K$).

3. Generalize Example 5.22 to any values $N, K$ with $K = N - 2$.

4. Show that the problem in Algorithm 5.25 with the receiver waiting for a message which the sender does not resend does not occur in Algorithm 5.26.

5. Discuss the scenario that unfolds in Algorithm 5.9 when the systems starts with only one proposer and there are no failures.

6. Show an example with two proposers that causes livelock in Algorithm 5.9.

# Bibliography

[1] Y. Afek and E. Gafni. Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks. *SIAM J. of Comput.*, 20:376–394, 1991.

[2] H. Attiya and J. Welch. *Distributed Computing—Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.

[3] B. Awerbuch. Complexity of Network Synchronization. *J. of the ACM*, 32:804–823, 1985.

[4] V.C. Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, 1996.

[5] M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *PODC 83*, pages 27–30, 1983.

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Comp. Syst.*, 9:272–314, 1991.

[7] G. Bracha and S. Toueg. Distributed Deadlock Detection. *Distributed Computing*, 2:127–138, 1987.

[8] K. Mani Chandy and J. Misra. Distributed Computation on Graphs: Shortest Path Algorithms. *Commun. ACM*, 25:833–837, 1982.

[9] K. Mani Chandy and J. Misra. Distributed Deadlock Detection. *ACM Trans. Comp. Syst.*, 1:144–156, 1983.

[10] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Comp. Syst.*, 3:63–75, 1985.

[11] E. Chang and R. Roberts. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Comm. of the ACM*, 22:281–283, 1979.

[12] J.-M. Chang and N.F. Maxemchuck. Reliable Broadcast Protocols. *ACM Trans. Comp. Syst.*, 2:251–273, 1984.

[13] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, Asynchronous, and Causally Ordered Communication. *Distributed Computing*, 9:173–191, 1996.

[14] R. Chow and T. Johnson. *Distributed Operating Systems & Algorithms*. Addison-Wesley, 1997.

[15] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems—Concepts and Design*. Addison-Wesley, fourth edition, 2005.

[16] X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms. Taxonomy and Survey. *ACM Computing Surveys*, 36:372–421, 2004.

[17] E.W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Comm. of the ACM*, 17:643–644, 1974.

[18] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters*, 16:217–219, 1983.

[19] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[20] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about Knowledge*. The MIT Press, 1995.

[21] M. Flatebo, A.K. Datta, and S. Ghosh. Self-stabilization in Distributed Systems. In T.L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 100–114. IEEE Computer Society Press, 1994.

[22] W.J. Fokkink. *Distributed Algorithms—An Intuitive Approach*. MIT Press, 2013.

[23] R.G. Gallager, P.A. Humblet, and P.M. Spira. A Distributed Algorithm for Minimum-Weights Spanning Trees. *ACM Trans. on Progr. Lang. and Syst.*, 5:66–77, 1983.

[24] M.G. Gouda and N.J. Multari. Stabilizing Communication Protocols. *IEEE Trans. on Computers*, 40:448–458, 1991.

[25] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[26] R. Gupta, S. A. Smolka, and S. Bhaskar. On Randomization in Sequential and Distributed Algorithms. *ACM Computing Surveys*, 26:7–86, 1994.

[27] D. Hirschberg and J. Sinclair. Decentralized Extrema-Finding in Circular Configurations of Processes. *Comm. of the ACM*, 23:627–628, 1980.

[28] S-T. Huang. Detecting Termination of Distributed Computations by External Agents. In *9th Int'l Conf. on Distributed Computing Systems*, pages 79–84, 1989.

[29] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[30] E. Knapp. Deadlock Detection in Distributed Databases. *ACM Comput. Surv.*, 19:303–328, 1987.

[31] A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[32] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21:558–565, 1978.

[33] L. Lamport and N.A. Lynch. Distributed Computing: Models and Methods. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 1157–1199. 1990.

[34] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[35] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[36] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[37] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[38] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[39] M. Maekawa. A $\sqrt{n}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Trans. on Comp. Syst.*, 3:145–159, 1985.

[40] F. Mattern. Global Quiescence Detection Based on Credit Distribution and Recovery. *Inf. Proc. Lett.*, 30:195–200, 1989.

[41] D.S. Miloji, F. Douglis, Y. Paindeveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32:241–299, 2000.

[42] A. Oram. *Peer-to-Peer, Harnassing the Power of Disruptive Technologies*. O'Reilly, 2001.

[43] J. Pachl, E. Korach, and D. Rotem. Lower Bounds for Distributed Maximum-Finding Algorithms. *J. of the ACM*, 31:905–918, 1984.

[44] G.L. Peterson. An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem. *ACM Trans. on Progr. Lang. and Syst.*, 4:758–762, 1982.

[45] K. Raymond. A Tree-based Algorithm for Distributed Mutual Exclusion. *ACM Trans. Comp. Syst.*, 7:61–77, 1989.

[46] M. Raynal. A Simple Taxonomy for Distributed Mutual Exclusion Algorithms. *Oper. Syst. Rev.*, 25:47–50, 1991.

[47] M. Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer-Verlag, 2013.

[48] W. Reisig. *Elements of Distributed Algorithms—Modeling and Analysis with Petri Nets*. Springer-Verlag, 1998.

[49] G. Ricart and A.K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Comm. of the ACM*, 24:9–17, 1981 (corrigendum in Comm. of the ACM 24 (578)).

[50] P. Samuel. *Projective Geometry*. Springer-Verlag, 1988.

[51] B.A. Sanders. The Information Structure of Distributed Mutual Exclusion Algorithms. *ACM Trans. on Comp. Syst.*, 5:284–299, 1987.

[52] A. Schiper, J. Eggli, and A. Sandoz. A New Algorithm to Implement Causal Ordering. In *3rd Workshop on Distributed Algorithms*, volume 392 of *Lect. Notes Comp. Sc.*, pages 219–232. Springer-Verlag, 1989.

[53] M. Schneider. Self-Stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[54] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: in Search of the Holy Grail. *Distr. Comp.*, 7:149–174, 1994.

[55] M. Singhal. A Heuristically Aided Algorithm for Mutual Exclusion in Distributed Systems. *IEEE Trans. on Softw. Eng.*, 38:651–662, 1989.

[56] M. Singhal. Deadlock Detection in Distributed Systems. *IEEE Computer*, 22:37–48, 1989.

[57] M. Singhal and A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Inf. Process. Lett.*, 43:47–52, 1992.

[58] T.K. Srikanth and S. Toueg. Simulating Authenticated Broadcasts to Derive Simple Fault-tolerant Algorithms. *Distributed Computing*, 2:80–94, 1987.

[59] I. Suzuki and T. Kasami. A Distributed Mutual Exclusion Algorithm. *ACM Trans. on Comp. Syst.*, 3:344–349, 1985.

[60] A.S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2006.

[61] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, second edition, 2000.

[62] F.J. Torres-Rojas and M. Ahamad. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. *Distr. Comp.*, 12:179–195, 1999.

[63] J. Turek and D. Shasha. The Many Faces of Consensus in Distributed Systems. In T.L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 83–99. IEEE Computer Society Press, 1994.

[64] Robbert van Renesse and Deniz Altinbuken. Paxos Made Moderately Complex. *ACM Comput. Surv.*, 47(3):5:1–5:36, 2015.

[65] G. Varghese. Self-stabilization by Counter Flushing. In *PODC 94*, pages 244–253, 1994.

[66] G. Varghese and N. Lynch. A Trade-off between Safety and Lifeness for randomized Coordinated Atttack Protocols. In *PODC 92*, pages 241–259, 1992.

[67] Roger Wattenhofer. *The Science of the Blockchain*. Inverted Forest Publishing, 2016.