13

Fault Tolerance in Distributed Systems

The earlier parts of this book studied the coordinated behavior that can be achieved in distributed systems where processes are *reliable*. For several reasons it is attractive to study what coordinated behavior of processes is possible under the assumption that processes may fail; this study is the subject of the last part of the book. Many solutions for fault tolerance are *ad hoc* and, also, in the large collection of impossibility proofs structure and underlying theory are sometimes hard to find. On the other hand, some problems have elegant and well-established theory and simple solutions whose presentation fits well in this introductory textbook.

This chapter serves as a general introduction to the later chapters. We illustrate the reasons for using fault-tolerant algorithms (Section 13.1), and subsequently introduce two main types of fault-tolerant algorithms, namely robust algorithms (Section 13.2) and stabilizing algorithms (Section 13.3).

13.1 Reasons for Using Fault-tolerant Algorithms

Increasing the number of components in a distributed system means increasing the probability that some of these components will be subject to failure during the execution of a distributed algorithm. Computers in a network may fail, processes in a system can be erroneously killed by switching off a workstation, or a machine may produce an incorrect result due to, e.g., memory malfunctioning. Modern computers are becoming more and more reliable, thereby decreasing the chance of the occurrence of such failures in any individual computer. Nonetheless, the chance of a failure occurring at some place in a distributed system may grow arbitrarily large when the number of its components increases. To avoid the necessity of restarting an algorithm each time a failure occurs, algorithms should be designed so as to deal properly with such failures.

Of course, vulnerability to failures is also a concern in sequential computations, in safety-critical applications, or if a computation runs for a long time and produces a non-verifiable result. Internal checks protect against errors of some types (e.g., modular checking against arithmetic miscalculation), but of course no protection can be achieved against the complete loss of the program or erroneous changes in its code. Therefore the possibilities of fault-tolerant computing by sequential algorithms and uni-processor computing systems are limited.

Because of the dispersion of processing resources in a distributed system, these systems have the *partial-failure* property; no matter what kind of failure occurs, it usually affects only a part of the entire system. And, while the growth of the number of components makes it extremely likely that a failure will occur in some component, it becomes extremely unlikely that a failure will occur in all components. Therefore it may be hoped that the tasks of failing processes can be taken over by the remaining components, leading to a graceful degradation rather than an overall malfunctioning. Indeed, as can be seen in the following chapters, the design of distributed algorithms for systems with defective processes is possible, and coordinated behavior can be achieved.

The partial-failure property makes the use of distributed ("replicated") architectures an attractive option for the design of computer applications that are not distributed by nature, but require high reliability. The primary computer system of the Space Shuttle aircraft serves as an example; Spector and Gifford [SG84] describe its development. Controlling the Shuttle is a task well within the capacity of a single off-the-shelf microprocessor, but the possibility of such a processor's breaking down during a flight was a serious concern in its design. The final control system used four identical processors, each performing exactly the same computation[1]; the actuators vote on the outcome, which allows perfect control even if one processor fails. (The physical realization of the actuators allows the system to survive the later failure of a second processor as well.) Although replication is an appealing option for increasing reliability, the design of the algorithms necessary to coordinate a cluster of (unreliable) processors is far from trivial. This is illustrated by the postponement of the first flight of the Shuttle (from April 8 to April 10, 1981), because of an error in the software precisely responsible for that task.

Fortunately, the study of fault-tolerant algorithms has advanced consider-

[1] See [SG84] for an explanation of the fifth ("hot spare") and sixth ("cold spare") processors.

ably since 1981, and reliable applications based on replication are now well within reach.

Two radically different approaches towards fault tolerance are followed in the literature, and both will be studied in this book. In *robust algorithms* each step of each process is taken with sufficient care to ensure that, in spite of failures, correct processes only take correct steps. In *stabilizing algorithms* correct processes can be affected by failures, but the algorithm is guaranteed to recover from any arbitrary configuration when the processes resume correct behavior. We shall briefly introduce robust algorithms in Section 13.2 and stabilizing algorithms in Section 13.3, and give a short comparison at the end of the chapter.

13.2 Robust Algorithms

Robust algorithms are designed to guarantee the continuous correct behavior of correctly operating processes in spite of failures occurring in other processes during their execution. These algorithms rely on strategies such as voting, whereby a process will only accept certain information when sufficiently many other processes have also declared receipt of this information. However, a process should never wait to receive information from all processes, because a deadlock may thus arise, if a process has crashed.

13.2.1 Failure Models

To determine how the correctly operating processes can protect themselves against failed processes, assumptions must be made about *how* a process might fail. In the following chapters it is always assumed that only *processes* can fail; channels are reliable. Thus, if a correct process sends a message to another correct process, receipt of the message within finite time is guaranteed. (A failing channel can be modeled by a failure in one of the incident processes, for example, an omission failure.) As an additional assumption, we always assume that each process can send to each other process.

The following *fault models* are used in this book.

(1) *Initially dead processes.* A process is called initially dead if it does not execute a single step of its local algorithm.

(2) *Crash model.* A process is said to crash if it executes its local algorithm correctly up to some moment, and does not execute any step thereafter.

(3) *Byzantine behavior.* A process is said to be Byzantine if it executes

arbitrary steps that are not in accordance with its local algorithm. In particular, a Byzantine process may send messages with an arbitrary content.

The correctness requirements set forward for robust algorithms always refer to the local state (or output) of *correct* processes only. An initially dead process never produces output and its state always equals the initial state. The output of a crashing process, if it produces any, will be correct, because up to the occurrence of the crash the process behaves correctly. Needless to say, the local state or output of a Byzantine process can be arbitrary, and no non-trivial requirement on it can be satisfied by any algorithm.

Hierarchy of fault models. A hierarchy can be defined among these three fault models. First, an initially dead process can be considered a special case of a crashing process, namely, the case where the crash occurs before the first event in that process. Second, a crashed process can be considered as a special case of a Byzantine process, because the arbitrary behavior assumed for Byzantine processes includes execution of no steps at all. It is therefore more difficult to tolerate crashes than to tolerate initially dead processes, and it is even more difficult to tolerate Byzantine processes. Stated differently, a Byzantine-robust algorithm is also crash-robust, and a crash-robust algorithm is also initially-dead-robust. On the other hand, impossibility of an initially-dead-robust algorithm implies impossibility of a crash-robust algorithm, and impossibility of a crash-robust algorithm implies impossibility of a Byzantine-robust algorithm.

An *omission failure* consists of skipping a step in the algorithm (e.g., the sending or receipt of a message), after which execution continues. Omission failures are a special case of Byzantine behavior, and a crash is a special case of omission (namely, where *all* steps after a certain moment are omitted). Therefore omission failures fit into this hierarchy between crash and Byzantine failures.

Mixed failures and timing errors. Initially dead processes and crashes are called *benign* failure types, while Byzantine failures are called *malign* failures. For several distributed problems it turns out that a collection of N processes can tolerate t benign failures if $2t < N$, while robustness against t malign failures requires $3t < N$ (in an *asynchronous* model of computation). Because malign failures often cannot be excluded in practice but are very rare compared with benign failures, Garay and Perry [GP92] extended some results to a mixed-failure model, where t processes may fail, b out of which in

a malign way. Correct behavior of the remaining processes can be achieved in this model if $2b + t < N$ (in a *synchronous* model of computation).

In synchronous distributed systems there is an additional failure mode, namely, where a process executes correct steps but at the wrong time (due to a slow or fast clock of the process). This type of incorrect process behavior is called a *timing error*.

13.2.2 Decision Problems

The study of robust algorithms is centered around the solution of so-called *decision problems*, where it is required that each of the (correct) processes irreversibly writes a "decision" value to its output. The required decision value usually depends in a quite trivial manner on the input values of the processes, making these problems fairly easy to solve in a fault-free (reliable) environment.

The requirements set for the decisions are usually of three types, namely, *termination*, *consistency*, and *non-triviality*.

The termination requirement states that all correct processes will decide, i.e., write a value to their output, eventually. In deterministic algorithms termination is required in all computations; in probabilistic algorithms it is required to occur with probability one (see Chapter 9). The requirement that all correct processes write ia value to their output excludes any solution in which a process must wait to receive some information from more than $N-t$ processes. Indeed, the stopping of t processes causes an indefinite wait in a correct process, violating the termination requirement.

The consistency requirement imposes a relation between the decisions taken by different processes. In the simplest case, all decisions are required to be equal; we speak of a *consensus problem* in this case (which value is required may depend on the inputs). In more complicated problems a class of output vectors can be defined, and the decisions of correct processes should form a vector in this class. In the election problem (Chapter 7), for example, the requirement is that one process decides that it is *elected* ("1"), and all other processes should decide *defeated* ("0").

A termination and a consistency requirement are usually sufficient for a distributed algorithm to be useful; in cases where a task is shown to be impossible to solve, an additional requirement of a more technical nature is needed. The non-triviality requirement excludes algorithms based on a fixed output for the problem, on which every process decides without any communication. The consensus problem, for example, could be solved by an algorithm in which each process writes a "0" to the output immediately.

The non-triviality requirement states that two essentially different outputs are possible in different executions of the algorithm (i.e., in the case of consensus, that the algorithm has executions that write "0" *and* executions that write "1"; of course, within one execution all decisions are consistent).

Decision problems abstract a large number of common situations in distributed computing, as we now discuss.

(1) *Commit–Abort.* In a distributed database a transaction involving various sites must be executed in *all* involved sites or in *none* of them. Therefore, after announcing the update to these sites, each site determines whether the update can be executed locally and votes either "yes" or "no". Subsequently, all (correct) sites must decide whether to *commit* the transaction, meaning that it will be executed everywhere, or to *abort* it, meaning that it will not be executed. If all processes voted "yes" the processes must decide *commit*, if some processes vote "no" the outcome must be *abort*. Consistency here means that all decisions are equal, and the problem is non-trivial because, depending on the input, both a *commit* and an *abort* may be required.

(2) *Distributed computations: consensus on input.* In systems where a computation is replicated to increase reliability, the outcome of the various processors can be equal only if the computations are based on the same inputs. The effect of a defective sensor, sending different values to the various processors, must be eliminated by executing a consensus algorithm among the processors. The input of each processor is the value received from the sensor, and each (correct) processor must decide on the same value to be used in the subsequent computation. The output is usually required to be the value that is a majority of the inputs or, weaker, to be any value that occurs at least once as an input; in both cases the problem is non-trivial.

(3) *Election.* In the election problem (see also Chapter 7) it is required that one process decides to become a leader and all other (correct) processes decide to become a non-leader. The problem becomes non-trivial if we require that potentially different processes can become leader.

(4) *Approximate agreement.* In cases where consensus among inputs is not achievable, a weaker form of consistency may suffice for some applications. In the approximate-agreement problem each process has an integer input from a given finite range, say $1, \ldots, k$. The decisions of processes are required to differ by at most 1, and to lie

between values that actually occurred as input; see also Section 14.3 for a continuous version of the problem (ϵ-approximate agreement).

13.2.3 Overview of Chapters 14 through 16

Due to the large number of known results concerning robust algorithms it is impossible in this book to discuss completely the state-of-the-art in the field. The material presented in Chapters 14 through 16 has been selected with the following criteria in mind.

(1) Some fundamental results should be included: that there is no deterministic consensus in asynchronous systems; that probabilistic algorithms tolerate benign failures in up to one-half of the processes, or malign failures in up to one-third of the processes.
(2) A demonstration should be given of some of the techniques used to achieve robustness or to prove its impossibility.
(3) A demonstration should be included that synchronous systems can achieve more robustness than asynchronous systems. This contrasts with the results (presented in Chapter 12) that indicate that reliable synchronous systems are not more powerfull than reliable asynchronous systems.
(4) Failure detectors are a promising new paradigm, rapidly making the move to being applied technology, and should be included.

Chapter 14 studies the robustness that can be achieved in *asynchronous* systems. A fundamental result was shown by Fischer, Lynch, and Paterson [FLP85], namely, that no deterministic algorithm for consensus exists that can tolerate even a single crash failure. This result was generalized by Moran and Wolfstahl [MW87] to a wider class of decision problems (tasks) including, e.g., election, but not approximate agreement. The conclusion of these works is that crashes can be tolerated only by probabilistic algorithms (or synchronous ones). The result cannot be strengthened to apply to the weaker fault model of initially dead processes; this is shown by the existence of deterministic consensus and election algorithms for initially dead processes. Randomization does help; a randomized consensus algorithm by Bracha and Toueg [BT85], which is asynchronous and tolerates $t < N/2$ crashing processes or $t < N/3$ Byzantine processes is presented.

Chapter 15 studies the robustness that can be achieved in *synchronous* systems. Unlike in the asynchronous model, the crashing of a process can be detected by the remaining processes, because a bounded response time of a correct process is guaranteed. Consequently, a much higher degree of

robustness is achievable, which is shown in the idealized model of a system operating in pulses (see Chapter 12). Deterministic algorithms can tolerate $t < N/3$ Byzantine processes, $t < N$ crashing processes, or $t < N$ Byzantine processes if message authentication is possible. This and the impossibility results of Chapter 14 imply that robust synchronizers for fully asynchronous networks do not exist. The implementation of the pulsed model on systems with limited asynchrony (physical clocks and bounded message-delay time) requires the clocks to be accurately synchronized, a problem which is the subject of the remainder of Chapter 15.

Failure detection, discussed in Chapter 16, is another way to strengthen the computational model and, like synchrony, is usually available in distributed systems in some form. Just like one can compare various models of synchrony to see how much is necessary to solve a problem, we may study models of failure detection and see how accurate their output must reflect actual failures in order to solve the consensus problem.

13.2.4 Topics not Addressed in this Book

The results in Chapter 14 and 15 give an indication about the degree of robustness achievable in distributed systems. Many problems and results remain for further study; below we indicate a few of these and give pointers to the literature.

(1) *Refinement of synchrony assumptions.* In this book only completely asynchronous and completely synchronous systems are considered, and these systems differ considerably in their achievable robustness. Dolev, Dwork, Lynch, and Stockmeyer [DDS87], [DLS88] studied the fault-tolerance of systems under intermediate assumptions about synchrony.

(2) *Determination of solvable tasks.* In this book the solvability of some tasks and the unsolvability of others is demonstrated; a precise characterization of solvable and unsolvable tasks was given by Biran, Moran, and Zaks [BMZ90].

(3) *Complexity of fault tolerance.* In addition to deciding what tasks are solvable, it is possible to study the amount of computational resources consumed by a protocol for solvable tasks. Several complexity measures can be taken into account: message complexity, bit complexity, and time complexity (often called round complexity in this context). An overview of some results and a consensus protocol meeting several lower bounds simultaneously is found in [BGP92].

(4) *Dynamic systems and group membership.* In this book it is assumed that the collection of processes is fixed (static) and known (though some processes may fail). A static collection of processes is suitable for an application that must run reliably for a fixed, finite amount of time, such as an aircraft control system, where repairs and reconfiguration take place *off-line.* Fault-tolerant operation of a system for an indefinite duration (as in, for example, an operating system or an air-traffic-control system) requires that processes can be repaired and the set of processes reconfigured *on-line*, i.e., without stopping the application. Reconfiguring the set of processes is done by executing a *group membership* protocol, in which the active processes agree on the set of processes in the system. Some results on these protocols and references are found in [RBS92].

(5) *Communication using shared variables.* In this book interprocess communication by means of message passing is considered. Some authors have studied the fault tolerance of distributed systems where communication is based on shared variables; see Taubenfeld and Moran [TM89].

(6) *Wait-free synchronization.* An even more sophisticated communication model is where processes share arbitrary objects, rather than registers. In this model the desired interaction between the processes can be formulated in terms of object methods, and algorithmical problems are formulated as implementation of objects. There is extensive theory about which objects can and which cannot be implemented; see [AW98].

13.3 Stabilizing Algorithms

Robust algorithms continuously show correct coordinated behavior even when failures occur, but the number of failures is limited and the failure model must usually be known precisely. The second approach to fault tolerance, that of stabilizing algorithms, is different. Any number of failures of arbitrary types is allowable, but correct behavior of the algorithm is suspended until some time after the repair of the failures. A *stabilizing* algorithm (sometimes called *self-stabilizing*) can be started in any system configuration, and eventually reaches an allowed state, and behaves according to its specifications from then on. Consequently, the effects of temporal failures die out, and also, there is no need to initialize the system consistently.

Stabilizing algorithms are studied in Chapter 17. The concept of stabi-

lization was introduced as early as in 1974 [Dij74], and applied to algorithms to achieve mutual exclusion on a ring of processors. These algorithms are discussed because of their historical value and because they are elegant and insightful. Stabilizing solutions exist for a number of problems solved earlier in this book, such as data transmission, election, and the computation of routing tables and a depth-first search tree.

Robust versus stabilizing algorithms. Stabilizing algorithms offer protection against so-called *transient failures*, i.e., temporary misbehavior of system components. These failures may occur in large parts of a distributed system when the physical conditions temporarily reach extreme values, inducing erroneous behavior of memories and processors. Examples include the control system of a spacecraft when being hit by large amounts of cosmic rays, and systems in which many components are simultaneously affected by a natural disaster. When conditions return to normal, the processes resume operation according to their programs, but due to their temporal misbehavior the global state (configuration) may be an arbitrary one. The stabilizing property guarantees convergence to the required behavior.

Robust algorithms protect against the *permanent failure* of a limited number of components. The surviving processes maintain correct (though possibly less efficient) behavior during the repair and reconfiguration of the system. Therefore, robust algorithms must be used when the temporal interruption of service is unacceptable.

Publications by Gopal and Perry [GP93] and Anagnostou and Hadzilacos [AH93] study algorithms that are both robust and stabilizing. Gopal and Perry demonstrate how a robust protocol can be modified automatically (compiled) into a protocol that is both robust and stabilizing. Anagnostou and Hadzilacos show that no robust and stabilizing algorithm exists for election and computation of the ring size, and present a (randomized) protocol for assigning distinct names.

14

Fault Tolerance in Asynchronous Systems

This chapter studies the solvability of decision problems in asynchronous distributed systems. The results are arranged around a fundamental result by Fischer, Lynch, and Paterson [FLP85], presented in Section 14.1. Formulated as an impossibility proof for a class of decision algorithms, the result can also be read as a list of assumptions that together exclude solutions for decision problems. Relaxing the assumptions makes it possible to obtain practical solutions for various problems, as is shown in the subsequent sections. See also Subsection 14.1.3 for a further discussion.

14.1 Impossibility of Consensus

In this section the fundamental theorem of Fischer, Lynch, and Paterson [FLP85], stating that there are no asynchronous, deterministic 1-crash robust consensus protocols, is proved. The result is shown by reasoning involving fair execution sequences of the algorithms. We first introduce some notation (in addition to that introduced in Section 2.1) and give elementary results that are useful also in later sections.

14.1.1 Notation, Definitions, Elementary Results

The sequence $\sigma = (e_1, \ldots, e_k)$ of events is *applicable* in configuration γ if e_1 is applicable in γ, e_2 in $e_1(\gamma)$, and so on. If the resulting configuration is δ, we write $\gamma \rightsquigarrow^{\sigma} \delta$ or $\sigma(\gamma) = \delta$, to make the events leading from γ to δ explicit. If $S \subseteq \mathbb{P}$ and σ contains only events in processes of S we also write $\gamma \rightsquigarrow_S \delta$.

Proposition 14.1 *Let sequences σ_1 and σ_2 be applicable in configuration γ, and let no process participate in both σ_1 and σ_2. Then σ_2 is applicable in $\sigma_1(\gamma)$, σ_1 is applicable in $\sigma_2(\gamma)$, and $\sigma_2(\sigma_1(\gamma)) = \sigma_1(\sigma_2(\gamma))$.*

Proof. This follows from repeated application of Theorem 2.19. □

Process p has a read-only input variable x_p and a write-once output register y_p with initial value b. The input configuration is completely determined by the value of x_p for each process p. Process p can *decide* on a value (usually 0 or 1) by writing it in y_p; the initial value b is *not* a decision value. It is assumed that a correct process executes infinitely many events in a fair execution; to this end, a process can always execute a (possibly void) internal event.

Definition 14.2 *A t-crash fair execution is an execution in which at least $N - t$ processes execute infinitely many events, and each message sent to a correct process is received. (A process is correct if it executes infinitely many events.)*

The maximal number of faulty processes that can be handled by an algorithm is called the *resilience* of the algorithm, and is always denoted by t. In this section the impossibility of an asynchronous, deterministic algorithm with resilience one is demonstrated.

Definition 14.3 *A 1-crash-robust consensus algorithm is an algorithm satisfying the following three requirements.*

(1) **Termination.** *In every 1-crash fair execution, all correct processes decide.*

(2) **Agreement.** *If, in a reachable configuration, $y_p \neq b$ and $y_q \neq b$ for correct processes p and q, then $y_p = y_q$.*

(3) **Non-triviality.** *For $v = 0$ and for $v = 1$ there exist reachable configurations in which, for some p, $y_p = v$.*

For $v = 0, 1$ a configuration is called *v-decided* if for some p, $y_p = v$; a configuration is called *decided* if it is 0-decided or 1-decided. In a v-decided configuration, some process has decided on v. A configuration is called *v-valent* if all decided configurations reachable from it are v-decided. A configuration is called *bivalent* if both 0-decided and 1-decided configurations are reachable from it, and *univalent* if it is either 1-valent or 0-valent. In a univalent configuration, although no decision has necessarily been taken by any process the eventual decision is implicitly determined already.

A configuration γ of a t-robust protocol is called a *fork* if there exists a set T of (at most) t processes, and configurations γ_0 and γ_1, such that $\gamma \rightsquigarrow_T \gamma_0$, $\gamma \rightsquigarrow_T \gamma_1$, and γ_v is v-valent. Informally, γ is a fork if a subset of t processes can enforce a 0-decision as well as a 1-decision. The following proposition formalizes that at any moment, the crash of at most t processes must be survived by the remaining processes.

Proposition 14.4 *For each reachable configuration of a t-robust algorithm and each subset S of at least $N - t$ processes, there exists a decided configuration δ such that $\gamma \rightsquigarrow_S \delta$.*

Proof. Let γ and S be as above, and consider an execution that reaches configuration γ and contains infinitely many events in each process of S thereafter (and no steps of processes not in S). This execution is t-crash fair, and the processes in S are correct; hence they reach a decision. □

Lemma 14.5 *There exist no reachable fork.*

Proof. Let γ be a reachable configuration and T a subset of at most t processes.

Let S be the complement of T, i.e., $S = \mathbb{P} \setminus T$; S has at least $N - t$ processes, hence there exists a decided configuration δ such that $\gamma \rightsquigarrow_S \delta$ (Proposition 14.4). Configuration δ is either 0- or 1-decided; assume w.l.o.g. that it is 0-decided.

It will now be shown that $\gamma \rightsquigarrow_T \gamma'$ for no 1-valent γ'; let γ' be any configuration such that $\gamma \rightsquigarrow_T \gamma'$. As steps in T and S commute (Proposition 14.1), there is a configuration δ' that is reachable from both δ and γ'. As δ is 0-decided, so is δ', which shows that γ' is *not* 1-valent. □

14.1.2 The Impossibility Proof

We shall first exploit the non-triviality of the problem to show that there exists a bivalent initial configuration (Lemma 14.6). Subsequently it will be shown that, starting from a bivalent configuration, every enabled step can be executed without moving to a univalent configuration (Lemma 14.7). This suffices to show the impossibility of consensus algorithms (Theorem 14.8). In the sequel, let A be a 1-crash-robust consensus algorithm.

Lemma 14.6 *There exists a bivalent initial configuration for A.*

Proof. As A is non-trivial (Definition 14.3), there are reachable 0- and 1-decided configurations; let δ_0 and δ_1 be initial configurations such that a v-decided configuration is reachable from δ_v.

If $\delta_0 = \delta_1$, this initial configuration is bivalent and the result holds. Otherwise, there are initial configurations γ_0 and γ_1 such that a v-decided configuration is reachable from γ_v, and γ_0 and γ_1 differ in the input of a single process. Indeed, consider a sequence of initial configurations, starting with δ_0 and ending with δ_1, in which each next initial configuration differs from the previous one in a single process. (The sequence is obtained by inverting input bits one by one.) From the first configuration in the sequence, δ_0, a 0-decided configuration is reachable, and from the last one, δ_1, a 1-decided configuration is reachable. Because a decided configuration is reachable from each initial configuration, γ_0 and γ_1 as described can be found as two subsequent configurations in the sequence. Let p be the process in which γ_0 and γ_1 differ.

Consider a fair execution starting in γ_0 in which p takes no steps; this execution is 1-crash fair and hence reaches a decided configuration β. If β is 1-decided, γ_0 is bivalent. If β is 0-decided, observe that γ_1 only differs from γ_0 in p, and p takes no steps in the execution; hence β is reachable from γ_1, showing that γ_1 is bivalent. (More precisely, a configuration β' can be reached from γ_1, where β' differs only from β in the state of p; hence β' is 0-decided.) □

To construct a non-deciding fair execution we must show that every process can take a step, and every message can be received, without enforcing a decision. Let a *step* s denote the receipt and handling of a particular message or a spontaneous move (internal or send) by a particular process. Depending on the state of the process where the step occurs, a different event may result. The receipt of a message is applicable if it is in transit, and a spontaneous step is always applicable.

Lemma 14.7 *Let γ be a reachable bivalent configuration and s an applicable step for process p in γ. There exists a sequence σ of events such that s is applicable in $\sigma(\gamma)$, and $s(\sigma(\gamma))$ is bivalent.*

Proof. Let C be the set of configurations reachable from γ without applying s, i.e., $C = \{\sigma(\gamma) : s \text{ does not occur in } \sigma\}$; s is applicable in every configuration of C (recall that s is a step, not a particular event).

There are configurations α_0 and α_1 in C such that a v-decided configuration is reachable from $s(\alpha_v)$. To see this, observe that by the bivalence of γ, v-decided configurations β_v are reachable from γ for $v = 0, 1$. If $\beta_v \in C$

(i.e., s was not applied to reach a decided configuration), observe that $s(\beta_v)$ is still v-decided, so choose $\alpha_v = \beta_v$. If $\beta_v \notin C$ (i.e., s was applied to reach a decided configuration), choose α_v as the configuration from which s was applied.

If $\alpha_0 = \alpha_1$, $s(\alpha_0)$ is the required bivalent configuration. Assume further that $\alpha_0 \neq \alpha_1$, and consider the configurations on the paths from γ to α_0 and α_1. Two configurations on these paths are called neighbors if one is obtained from the other in a single step. Because a 0-decided configuration is reachable from $s(\alpha_0)$ and a 1-decided configuration from $s(\alpha_1)$, it follows that

(1) there is a configuration γ' on the paths such that $s(\gamma')$ is bivalent; or
(2) there are neighbors γ_0 and γ_1 such that $s(\gamma_0)$ is 0-valent and $s(\gamma_1)$ is 1-valent.

In the first case, $s(\gamma')$ is the required bivalent configuration and we are done. In the second case, one of γ_0 and γ_1 is a fork, which is a contradiction. Indeed, assume (w.l.o.g.) that γ_1 is obtained in a single step from γ_0, i.e., $\gamma_1 = e(\gamma_0)$ for event e in process q. Now $s(e(\gamma_0))$ is $s(\gamma_1)$ and hence 1-valent, but it is not the case that $e(s(\gamma_0))$ is 1-valent because $s(\gamma_0)$ is already 0-valent. So e and s do not commute, which implies (Theorem 2.19) that $p = q$, but then the reachable configuration γ_0 satisfies $\gamma_0 \rightsquigarrow_p s(\gamma_0)$ and $\gamma_0 \rightsquigarrow_p s(e(\gamma_0))$. As the former is 0-valent and the latter 1-valent, γ_0 is a fork, which is a contradiction. □

Theorem 14.8 *There exists no asynchronous, deterministic, 1-crash-robust consensus algorithm.*

Proof. Assuming that such an algorithm exists, a non-deciding fair execution can be constructed, starting from a bivalent initial configuration γ_0.

When the construction is completed up to configuration γ_i, choose for s_i an applicable step that has been applicable for the longest possible number of steps. By the previous lemma, the execution can be extended in such a way that s_i is executed, and a bivalent configuration γ_{i+1} is reached.

The construction gives an infinite fair execution in which all processes are correct but a decision is never taken. □

14.1.3 Discussion

The result states that there are no asynchronous, deterministic, 1-crash-robust decision algorithms for the consensus problem; this excludes algorithms for a class of non-trivial problems (see Subsection 13.2.2).

Fortunately, some assumptions underlying the result of Fischer, Lynch, and Paterson can be made explicit, and the result turns out to be very sensitive to the weakening of any of them. Despite the impossibility result, many non-trivial problems do have solutions, even in asynchronous systems and where processes may fail.

(1) *Weaker fault model.* Section 14.2 considers the fault model of initially dead processes, which is weaker than the crash model, and in this model consensus and election are deterministically achievable.
(2) *Weaker coordination.* Section 14.3 considers problems that require a less close coordination between processes than does consensus, and demonstrates that some of these problems, including renaming, are solvable in the crash model.
(3) *Randomization.* Section 14.4 considers randomized protocols, where the termination requirement is sufficiently relaxed to make solutions possible even in the presence of Byzantine failures.
(4) *Weak termination.* Section 14.5 considers a different relaxation of the termination requirement, namely where termination is required only when a given process is correct; here also Byzantine-robust solutions are possible.
(5) *Synchrony.* The influence of synchrony is further studied in Chapter 15.

Fairly trivial solutions are possible if one of the three requirements of Definition 14.3 is simply omitted; see Exercise 14.1. Omission of the assumption (implicitly used in the proof of Lemma 14.6) that all combinations of inputs are possible is studied in Exercise 14.2.

14.2 Initially Dead Processes

In the model of initially dead processes, no process can fail after having executed an event, hence in a fair execution each process executes either 0 or infinitely many events.

Definition 14.9 *A t-initially-dead fair execution is an execution in which at least $N - t$ processes are active, each active process executes infinitely many events, and each message sent to a correct process is received.*

In a t-initially-dead-robust consensus algorithm, every correct process decides in every t-initially-dead fair execution. Agreement and non-triviality are defined as for the crash model.

```
var Succ_p, Alive_p, Rcvd_p : sets of processes      init ∅ ;

begin shout ⟨ name, p ⟩ ;
      (* that is: forall q ∈ ℙ do send ⟨ name, p ⟩ to q *)
      while #Succ_p < L
            do begin receive ⟨ name, q ⟩ ; Succ_p := Succ_p ∪ {q} end ;
      shout ⟨ pre, p, Succ_p ⟩ ;
      Alive_p := Succ_p ;
      while Alive_p ⊈ Rcvd_p
            do begin receive ⟨ pre, q, Succ ⟩ ;
                     Alive_p := Alive_p ∪ Succ ∪ {q} ;
                     Rcvd_p := Rcvd_p ∪ {q}
               end ;
      Compute a knot in G
end
```

Algorithm 14.1 COMPUTATION OF A KNOT.

Because processes do not fail after sending a message, it is safe for a process to wait for the receipt of a message from p if it knows that p has already sent at least one message. It will be shown that consensus and election are solvable in the initially dead model as long as a minority of the processes can be faulty ($t < N/2$). A larger number of initially dead processes cannot be tolerated (see Exercise 14.3).

Agreement on a subset of correct processes. First an algorithm by Fischer, Lynch, and Paterson [FLP85] is presented by which each of the correct processes computes *the same* collection of correct processes. The resiliency of this algorithm is $\lfloor (N-1)/2 \rfloor$; let L stand for $\lceil (N+1)/2 \rceil$, and observe that there are at least L correct processes. The algorithm works in two stages; see Algorithm 14.1.

Observe that processes send messages to themselves; this is done in many robust algorithms, and facilitates analysis. Here and in the remainder, let the operation "shout $\langle$ **mes** $\rangle$" stand for

$$\textbf{forall } q \in \mathbb{P} \textbf{ do send } \langle \textbf{ mes } \rangle \text{ to } q \ .$$

The processes construct a directed graph G by shouting their identity (in a $\langle$ **name**, p $\rangle$ message) and waiting for the receipt of L messages. As there are at least L correct processes, each correct process receives sufficiently many messages to complete this part. The successors of p in graph G are the nodes q from which p has received a $\langle$ **name**, q $\rangle$ message.

An initially dead process has not sent nor received any message, hence

forms an isolated node in G; a correct process has L successors, hence is not isolated. A knot is a strongly connected component without outgoing edges, containing at least two nodes. There is a knot in G, containing correct processes, and, because each correct process has out-degree L, this knot has size at least L. Consequently, as $2L > N$, there exists exactly one knot; call it K. Finally, as the correct process p has L successors, at least one successor belongs to K, implying that all processes in K are descendants of p.

Therefore, in the second stage of the algorithm, processes construct an induced subgraph of G, containing at least their descendants, by receiving the set of successors from every process they know to be correct. Because processes do not fail after sending a message, no deadlock occurs in this stage. Indeed, p waits to receive a message from q only if in the first stage some process has received a $\langle$ **name**, q $\rangle$ message, showing that q is correct.

Upon termination of Algorithm 14.1 each correct process has received the set of successors of each of its descendants, allowing it to compute the unique knot in G.

Consensus and election. As all correct processes agree on a knot of correct processes, it is now trivial to elect a process; the process with largest identity in K is elected. It is also easy to achieve consensus. Each process broadcasts, together with its successors, its input (x). After computing K, the processes decide on a value that is a function of the collection of inputs in K (for example, the value that occurs the more often, zero in the case of a draw).

The algorithms for knot-agreement, consensus, and election exchange $O(N^2)$ messages, where a message may contain a list of L process names. More efficient election algorithms have been proposed. Itai *et al.* [IKWZ90] gave an algorithm using $O(N(t + \log N))$ messages and demonstrated that this is a lower bound. Masuzawa *et al.* [MNHT89] considered the problem for cliques with sense of direction and proposed an $O(N\,t)$ message algorithm, which is again optimal.

Any election algorithm choosing a correct process as leader also solves the consensus problem; the leader broadcasts its input and all correct processes decide on it. Consequently, the above-mentioned upper bounds hold for the consensus problem for initially dead processes as well. In the crash model, however, the availability of a leader does not help in solving the consensus problem; the leader itself can crash before broadcasting its input. In addition, the election problem is not solvable in the crash model as is demonstrated in the next section.

14.3 Deterministically Achievable Cases

The consensus problem studied so far requires the same value to be decided upon in each process; this section studies the solvability of tasks that require a less close coordination between processes. In Subsection 14.3.1 a solution is presented for a practical problem, namely, the renaming of a collection of processes in a smaller name space. In Subsection 14.3.2 the impossibility results given earlier are extended to cover a larger class of decision problems.

A distributed *task* is described by sets X and D of possible input and output values, and a (possibly partial) mapping

$$T : X^N \longrightarrow \mathcal{P}(D^N).$$

The interpretation of the mapping T is that if the vector $\vec{x} = (x_1, \ldots, x_N)$ describes the input of the processes, then $T(\vec{x})$ is the set of legal outputs of the algorithm, described as a decision vector $\vec{d} = (d_1, \ldots, d_N)$. If T is a partial function, not every combination of input values is allowed.

Definition 14.10 *An algorithm is a t-crash robust solution for task T if it satisfies the following.*

(1) **Termination.** *In every t-crash fair execution, all correct processes decide.*

(2) **Consistency.** *If all processes are correct, the decision vector $\vec{d}$ is in $T(\vec{x})$.*

The consistency condition implies that in executions where a subset of the processes decide, the partial vector of decisions can always be extended to a vector in $T(\vec{x})$. The set D_T denotes the collection of all output vectors, i.e., the range of T.

(1) *Example: consensus.* The consensus problem requires all decisions to be equal, i.e.,

$$D_{\mathbf{cons}} = \{(0, 0, \ldots, 0), (1, 1, \ldots, 1)\}.$$

(2) *Example: election.* The election problem requires one process to decide on 1 and the others on 0, i.e.,

$$D_{\mathbf{elec}} = \{(1, 0, \ldots, 0), (0, 1, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}.$$

(3) *Example: approximate agreement.* In the *ϵ-approximate agreement* problem each process has a real input value and decides on a real output value. The maximal difference between two output values is

at most ϵ, and the outputs are required to be enclosed between two inputs.

$$D_{\mathbf{approx}} = \{(d_1, \ldots, d_N) : \max(d_i) - \min(d_i) \leq \epsilon\}.$$

(4) *Example: renaming.* In the *renaming problem* each process has a distinct identity, which can be taken from an arbitrarily large domain. Each process must decide on a new name, taken from a smaller domain $1, \ldots, K$, such that all new names are different.

$$D_{\mathbf{rename}} = \{(d_1, \ldots, d_N) : i \neq j \Rightarrow d_i \neq d_j\}.$$

In the *order-preserving* version of the renaming problem, the new names are required to preserve the order of the old names, i.e., $x_i < x_j \Rightarrow d_i < d_j$.

14.3.1 A Solvable Problem: Renaming

In this subsection an algorithm for renaming, by Attiya *et al.* [ABND+90], will be presented. The algorithm tolerates up to $t < N/2$ crashes (t is a parameter of the algorithm) and renames in a space of size $K = (N - t/2)(t+1)$.

An upper bound on t. We first show that no renaming algorithm can tolerate $N/2$ or more crashes; in fact, almost all crash-robust algorithms have a limit $t < N/2$ on the number of faults, and the proof below can be adapted to other problems.

Theorem 14.11 *No algorithm for renaming exists if $t \geq N/2$.*

Proof. If $t \geq N/2$, two disjoint groups of processes S and T of size $N - t$ can be formed. Owing to the possibility that t processes fail, a group must be able to decide "on its own", i.e., without interaction with processes outside the group (see Proposition 14.4). But then the groups can *independently* reach a decision in a single execution; the crux of the proof is to show that these decisions can be mutually inconsistent. We proceed with the formal argument for the case of renaming.

By Proposition 14.4, for each initial configuration γ there exists a configuration δ_S such that all processes in S have decided and $\gamma \rightsquigarrow_S \delta_S$; a similar property holds for T. The operation of the algorithm within a group of $N - t$ processes defines a relation from vectors of $N - t$ initial identities to vectors of $N - t$ new names. Because the initial name space is unbounded and the new names come from a bounded range, there are disjoint inputs

```
var V_p   : set of identities ;
    c_p   : integer ;

begin V_p := {x_p} ; c_p := 0 ; shout ⟨ set, V_p ⟩ ;
      while true
      do begin receive ⟨ set, V ⟩ ;
               if V = V_p then
                 begin c_p := c_p + 1 ;
                       if c_p = N − t and y_p = b then
                         (* V_p is stable for the first time: decide *)
                         y_p := ( # V_p , rank(V_p, x_p) )
                 end
               else if V ⊆ V_p then
                 skip    (* Ignore "old" information *)
               else  (* new input; update V_p and restart counting *)
                 begin if V_p ⊂ V then c_p := 1 else c_p := 0 ;
                       V_p := V_p ∪ V ; shout ⟨ set, V_p ⟩
                 end
         end
end
```

Algorithm 14.2 A SIMPLE RENAMING ALGORITHM.

that are mapped onto overlapping outputs. That is, there are input vectors (of length $N - t$) $\vec{u}$ and $\vec{v}$ such that $u_i \neq v_j$ for all i, j, but there are corresponding output vectors $\vec{d}$ and $\vec{e}$ such that $d_i = e_j$ for some i, j.

An incorrect execution is now constructed as follows. The initial configuration γ has inputs $\vec{u}$ in group S and $\vec{v}$ in group T; observe that all initial names are different (the initial names outside both groups can be chosen arbitrarily). Let σ_T be the sequence of steps by which group T reaches, from γ, the configuration δ_T in which the processes in T have decided on the names $\vec{e}$. By Proposition 14.1, this sequence is still applicable in configuration γ_S, in which the processes in S have decided on the names $\vec{d}$. In $\sigma_T(\gamma_S)$, two processes have decided on the same name (because $d_i = e_j$), which shows that the algorithm is not consistent. □

In what follows, $t < N/2$ is assumed.

The renaming algorithm. In the renaming algorithm (Algorithm 14.2), process p maintains a set V_p of process inputs that p has seen; initially, V_p contains just x_p. Every time p receives a set of inputs including ones that are new for p, V_p is extended by the new inputs. Upon starting and every time V_p is extended, p shouts its set. It follows that the set V_p only grows during

the execution, i.e., subsequent values of V_p are totally ordered by inclusion, and moreover, V_p contains at most N names. Consequently, process p shouts its set at most N times, showing that the algorithm terminates and that the message complexity is bounded by $O(N^3)$.

Further, p counts (in the variable c_p) the number of times it has received copies of its current set V_p. Initially c_p is 0, and c_p is incremented each time a message containing V_p is received. The receipt of a message $\langle \mathbf{set}, V \rangle$ may cause V_p to grow, necessitating a reset of c_p. If the new value of V_p equals V (i.e., if V is a strict superset of the old V_p), c_p is set to 1, otherwise to 0.

Process p is said to reach a *stable set* V if c_p becomes $N - t$ when the value of V_p is V. In other words, p has received for the $(N - t)$th time the current value V of V_p.

Lemma 14.12 *Stable sets are totally ordered, i.e., if q reaches the stable set V_1 and r reaches the stable set V_2, then $V_1 \subseteq V_2$ or $V_2 \subseteq V_1$.*

Proof. Assume q that reaches the stable set V_1 and r reaches the stable set V_2. This implies that q has received $\langle \mathbf{set}, V_1 \rangle$ from $N - t$ processes and r has received $\langle \mathbf{set}, V_2 \rangle$ from $N - t$ processes. As $2(N - t) > N$, there is at least one process, say p, from which q has received $\langle \mathbf{set}, V_1 \rangle$ and r has received $\langle \mathbf{set}, V_2 \rangle$. Consequently, V_1 and V_2 are both values of V_p, implying that one is included in the other. □

Lemma 14.13 *Each correct process reaches a stable set at least once in every fair t-crash execution.*

Proof. Let p be a correct process; the set V_p can only expand, and contains at most N input names. Consequently, a maximal value V_0 is reached for V_p. Process p shouts this value, and a $\langle \mathbf{set}, V_0 \rangle$ message is received by every correct process, which shows that every correct process eventually holds a *superset* of V_0.

However, this superset is *not* strict; otherwise, a correct process would send a strict superset of V_0 to p, contradicting the choice of V_0 (as being the largest set ever held by p). Consequently, every correct process q has a value $V_q = V_0$ at least once in the execution, and hence every correct process sends a $\langle \mathbf{set}, V_0 \rangle$ message to p during the execution. All these messages are received in the execution, and as V_p is never increased beyond V_0, they are all counted and cause V_0 to become stable in p. □

Upon reaching a stable set V for the first time, process p decides on the pair (s, r), where s is the size of V and r is the rank of x_p in V. A stable set has been received from $N - t$ processes, and hence contains at least

$N - t$ input names, showing that $N - t \leq s \leq N$. The rank in a set of size s satisfies $1 \leq r \leq s$. The number of possible decisions is therefore $K = \sum_{s=N-t}^{N} s$, which is $(N - t/2)(t + 1)$; if desired, a fixed mapping from the pairs to integers in the range $1, \ldots, K$ can be used (Exercise 14.5).

Theorem 14.14 *Algorithm 14.2 solves the renaming problem with an output name space of size* $K = (N - t/2)(t + 1)$.

Proof. Because, in any fair t-crash execution, each correct process reaches a stable set, each correct process decides on a new name. To show that the new names are all distinct, consider the stable sets V_1 and V_2 reached by processes q and r respectively. If the sets have different sizes, the decisions of q and r are different because the size is included in the decision. If the sets have the same size, then by Lemma 14.12, they are equal; hence q and r have a different rank in the set, again showing that their decisions are different. □

Discussion. Observe that a process does not terminate Algorithm 14.2 after deciding on its name; it continues the algorithm to "help" other processes to decide, too. Attiya *et al.* [ABND$^+$90] show that this is necessary because the algorithm must deal with the situation that some processes are so slow that they execute their first step after some other processes have already decided.

The simple algorithm presented here is not the best in terms of the size of name space used for renaming. Attiya *et al.* [ABND$^+$90] gave a more complicated algorithm that assigns names in the range from 1 to $N + t$. The results of the next subsection imply a lower bound of $N + 1$ on the size of the new name space for crash-robust renaming.

Attiya *et al.* also proposed an algorithm for order-preserving renaming. It renames to integers in the range 1 to $K = 2^t \cdot (N - t + 1) - 1$, which was shown to be the smallest size of name space possible for t-crash-robust order-preserving renaming.

14.3.2 Extended Impossibility Results

The impossibility result for consensus (Theorem 14.8) was generalized by Moran and Wolfstahl [MW87] to more general decision problems. The *decision graph* of task T is the graph $G_T = (V, E)$, where $V = D_T$ and

$$E = \{(\vec{d}_1, \vec{d}_2) : \vec{d}_1 \text{ and } \vec{d}_2 \text{ differ in exactly one component}\}.$$

Task T is called *connected* if G_T is a connected graph, and *disconnected* otherwise. It was assumed by Moran and Wolfstahl that the input graph of T (defined similarly to the decision graph) is connected, i.e., as in the proof of Lemma 14.6 we can move between any two input configurations by changing process inputs one by one. Furthermore, the impossibility result was shown for non-trivial algorithms, i.e., algorithms that satisfy, in addition to (1) termination and (2) consistency,

(3) **Non-triviality.** For each $\vec{d} \in D_T$, there is a reachable configuration in which the processes have decided on $\vec{d}$.

Theorem 14.15 *There exists no non-trivial* 1*-crash-robust decision algorithm for a disconnected task* T.

Proof. Assume, to the contrary, that such an algorithm, A, exists; a consensus algorithm A' may be derived from it, which constitutes a contradiction by Theorem 14.8. To simplify the argument we assume that G_T contains two connected components, "0" and "1".

Algorithm A' first simulates A, but instead of deciding on value d, a process shouts $\langle \mathbf{vote}, d \rangle$ and awaits the receipt of $N-1$ vote messages. No deadlock arises, because all correct processes decide in A; hence at least $N-1$ processes shout a vote message.

After receiving the messages, process p holds $N-1$ components of a vector in D^N. This vector can be extended by a value for the process from which no vote was received, in such a way that the entire vector is in D_T. (Indeed, a consistent decision was taken by this process, or is still possible.)

Now observe that different processes may compute different extensions, but that these extensions belong to the same connected component of G_T. Each process that has received $N-1$ votes decides on the name of the connected component to which the extended vector belongs. It remains to show that A' is a consensus algorithm.

Termination. It has already been argued above that every correct process receives at least $N-1$ votes.

Agreement. We first argue there exists a vector $\vec{d_0} \in D_T$ such that each correct process obtains $N-1$ components of $\vec{d_0}$.

Case 1: *All processes found a decision in* A. Let $\vec{d_0}$ be the vector of decisions reached; each process obtains $N-1$ components of $\vec{d_0}$, though the "missing" component may be different for each process.

Case 2: *All processes except one, say* r*, found a decision in* A. All correct processes receive the same $N-1$ decisions, namely those of

all processes except r. It is possible that r crashed, but because it is also possible that r is only slow, it must still be possible for r to reach a decision, i.e., there exists a vector $\vec{d_0} \in D_T$ which extends the decisions taken so far.

From the existence of $\vec{d_0}$ it follows that each process decides on the connected component of this vector $\vec{d_0}$.

Non-triviality. By the non-triviality of A, decision vectors in both component 0 and component 1 can be reached; by the construction of A', both decisions are possible.

Thus, A' is an asynchronous, deterministic, 1-crash-robust consensus algorithm. With Theorem 14.8 the non-existence of algorithm A follows. □

Discussion. The non-triviality requirement, stating that every decision vector in D_T is reachable, is fairly strong. One may ask whether some algorithms that are trivial in this sense may nonetheless be of interest. As an example, consider Algorithm 14.2 for renaming; it is not immediately clear that it is non-trivial, i.e., *every* vector with distinct names is reachable (it is); even less clear is why non-triviality would be of interest in this case.

Inspection of the proof of Theorem 14.15 reveals that a weaker requirement of non-triviality can be used in the proof, namely, that decision vectors are reachable in at least two different connected components of G_T. This weaker non-triviality may sometimes be induced from the statement of the problem.

Fundamental work concerning the decision tasks that are solvable and unsolvable in the presence of one faulty processor was done by Biran, Moran, and Zaks [BMZ90]. They gave a complete combinatorial characterization of the solvable decision tasks.

14.4 Probabilistic Consensus Algorithms

It was shown in the proof of Theorem 14.8 that every asynchronous consensus algorithm has infinite executions in which no decision is ever taken. Fortunately, for well-chosen algorithms such executions may be sufficiently rare to have probability zero, which makes the algorithms very practical in a probabilistic sense; see Chapter 9. In this section we present two probabilistic consensus algorithms, one for the crash model and one for the Byzantine model; the algorithms were proposed by Bracha and Toueg [BT85]. In both cases an upper bound on the resilience ($t < N/2$ and $t < N/3$, respectively) is proved first, and both algorithms match the respective bound.

In the correctness requirements for these probabilistic consensus algorithms, the termination requirement is made probabilistic, i.e., replaced by the weaker requirement of convergence.

(1) **Convergence.** For every initial configuration,

$$\lim_{k \to \infty} \mathbf{Pr}[\text{ a correct process has not decided after } k \text{ steps }] = 0.$$

Partial correctness (Agreement) must be satisfied in every execution; the resulting probabilistic algorithms are of the Las Vegas class (Subsection 9.1.2).

The probability is taken over all executions starting in a given initial configuration. In order for the probabilities to be meaningful, a probability distribution over these executions must be given. This can be done by using randomization in the processes (as in Chapter 9), but here a probability distribution on message arrivals is defined instead.

The probability distribution on executions starting in a given initial configuration is defined by the assumption of *fair scheduling*. Both algorithms operate in rounds; in a round a process shouts a message and awaits the receipt of $N - t$ messages. Define $R(q, p, k)$ as the event that, in round k, process p receives the (round-k) message of q among the first $N - t$ messages. Fair scheduling means that

(1) $\exists \epsilon > 0 \; \forall p, q, k : \mathbf{Pr}[\, R(p, q, k) \,] \geq \epsilon$.
(2) For all k, and different processes p, q, r, the events $R(q, p, k)$ and $R(q, r, k)$ are independent.

Observe that Proposition 14.4 also holds for probabilistic algorithms when convergence (termination with probability one) is required. Indeed, because a reachable configuration is reached with positive probability, a decided configuration must be reachable from every reachable configuration (albeit not necessarily reached in every execution).

14.4.1 Crash-robust Consensus Protocols

In this subsection the consensus problem is studied in the crash failure model. An upper bound $t < N/2$ on the resiliency is proved first, and subsequently an algorithm with resiliency $t < N/2$ is given.

Theorem 14.16 *There is no t-crash-robust consensus protocol for $t \geq N/2$.*

Proof. The existence of such a protocol, say P, implies the following three claims.

Claim 14.17 *P has a bivalent initial configuration.*

Proof. This is similar to the proof of Lemma 14.6; details are left to the reader. □

For a subset S of processes, configuration γ is said to be S-bivalent if both a 0- and a 1-decided configuration are reachable from γ by taking steps only in S. γ is called S-0-valent if, by taking steps only in S, a 0-decided configuration, but no 1-decided configuration, can be reached, and an S-1-valent configuration is defined similarly.

Partition the processes in two groups, S and T, of size $\lfloor N/2 \rfloor$ and $\lceil N/2 \rceil$.

Claim 14.18 *For a reachable configuration γ, γ is either both S-0-valent and T-0-valent, or both S-1-valent and T-1-valent.*

Proof. Indeed, the high resilience of the protocol implies that both S and T can reach a decision independently; if *different* decisions are possible, an inconsistent configuration can be reached by combining the schedules. □

Claim 14.19 *P has no reachable bivalent configuration.*

Proof. Let a reachable bivalent configuration γ be given and assume, w.l.o.g., that γ is S-1-valent and T-1-valent (use Claim 14.18). However, γ is bivalent, so also (clearly in cooperation between the groups) a 0-decided configuration δ_0 is reachable from γ. In the sequence of configurations from γ to δ_0 there are two subsequent configurations γ_1 and γ_0, where γ_v is both S-v-valent and T-v-valent. Let p be the process causing the transition from γ_1 to γ_0. Now $p \in S$ is impossible because γ_1 is S-1-valent and γ_0 is S-0-valent; similarly $p \in T$ is impossible. This is a contradiction. □

Contradiction to the existence of the protocol P arises from Claims 14.17 and 14.19; thus Theorem 14.16 is proved. □

The crash-robust consensus algorithm of Bracha and Toueg. The crash-robust consensus algorithm proposed by Bracha and Toueg [BT85] operates in *rounds*: in round k, a process sends a message to all processes (including itself) and awaits the receipt of $N - t$ round-k messages. Waiting for this number of messages does not introduce the possibility of deadlock (see Exercise 14.10).

In each round, process p shouts a vote for either 0 or 1, together with a weight. The weight is the number of votes received for that value in the previous round (1 in the first round); a vote with a weight exceeding $N/2$ is called a *witness*. Although different processes may vote differently in a

round, there are never witnesses for different values in one round, as will be shown below. If process p receives a witness in round k, p votes for its value in round $k+1$; otherwise p votes for the majority of the received votes. A decision is taken if more than t witnesses are received in a round; the decided process exits the main loop and shouts witnesses for the next two rounds in order to enable other processes to decide. The protocol is given as Algorithm 14.3.

Votes arriving for later rounds must be processed in the appropriate round; this is modeled in the algorithm by sending this message to the process itself for later processing. Observe that in any round a process receives at most one vote from each process, to a total of $N-t$ votes; because more than $N-t$ processes may shout a vote, processes may take different subsets of the shouted votes into account. We subsequently show several properties of the algorithm that together imply that it is a probabilistic crash-robust consensus protocol (Theorem 14.24).

Lemma 14.20 *In any round, no two processes witness for different values.*

Proof. Assume that in round k, process p witnesses for v and process q witnesses for w; $k > 1$ because in round 1 no process witnesses. The assumption implies that in round $k-1$, p received more than $N/2$ votes for v and q received more than $N/2$ votes for w. Together more than N votes are involved; consequently, the processes from which p and q received votes overlap, i.e., there is an r that sent a v-vote to p and a w-vote to q. This implies $v = w$. □

Lemma 14.21 *If a process decides, then all correct processes decide for the same value, and at most two rounds later.*

Proof. Let k be the first round in which a decision is taken, p a process deciding in round k, and v the value of p's decision. The decision implies that there were v-witnesses in round k; hence by Lemma 14.20 there were no witnesses for other values, and so no different decision is taken in round k.

In round k there were more than t witnesses for v (this follows from p's decision), hence all correct processes receive at least one v-witness in round k. Consequently, all processes that vote in round $k+1$ vote for v (observe also that p still shouts a vote in round $k+1$). This implies that if a decision is taken at all in round $k+1$, it is a decision for v.

In round $k+1$, only v-votes are submitted, hence all processes that vote in round $k+2$ witness for v in that round (p also does so). Consequently, in

```
var value_p          : (0, 1)   init x_p   (* p's vote *)
    round_p          : integer  init 0     (* Round number *)
    weight_p         : integer  init 1     (* Weight of p's vote *)
    msgs_p[0..1]     : integer  init 0     (* Count received votes *)
    witness_p[0..1]  : integer  init 0     (* Count received witnesses *)

begin
  while y_p = b do
    begin witness_p[0], witness_p[1], msgs_p[0], msgs_p[1] :=
                                          0, 0, 0, 0 ;   (* Reset counts *)
          shout ⟨ vote, round_p, value_p, weight_p ⟩ ;
          while msgs_p[0] + msgs_p[1] < N − t do
                begin receive ⟨ vote, r, v, w ⟩ ;
                      if r > round_p then      (* Future round ... *)
                        send ⟨ vote, r, v, w ⟩ to p   (* ... process later *)
                      else if r = round_p then
                        begin msgs_p[v] := msgs_p[v] + 1 ;
                              if w > N/2 then     (* Witness *)
                                witness_p[v] := witness_p[v] + 1
                        end
                      else (* r < round_p, ignore *) skip
                end ;
          (* Choose new value: vote and weight in next round *)
          if witness_p[0] > 0 then value_p := 0
          else if witness_p[1] > 0 then value_p := 1
          else if msgs_p[0] > msgs_p[1] then value_p := 0
          else value_p := 1 ;
          weight_p := msgs_p[value_p] ;
          (* Decide if more than t witnesses *)
          if witness_p[value_p] > t then y_p := value_p ;
          round_p := round_p + 1
    end ;
  (* Help other processes decide *)
  shout ⟨ vote, round_p, value_p, N − t ⟩ ;
  shout ⟨ vote, round_p + 1, value_p, N − t ⟩
end
```

Algorithm 14.3 CRASH-ROBUST CONSENSUS ALGORITHM.

round $k+2$ all correct processes that did not decide in earlier rounds receive $N-t$ v-witnesses, and decide on v. □

Lemma 14.22 $\lim_{k\to\infty}$ **Pr**[No decision is taken in a round $\leq k$] $= 0$.

Proof. Let S be a set of $N-t$ correct processes (such a set exists) and assume that no decision was taken until round k_0. The fair-scheduling assumption

implies that, for some $\rho > 0$, in any round the probability is at least ρ that every process in S receives exactly the votes of the $N - t$ processes in S. With probability at least $\psi = \rho^3$ this happens in three subsequent rounds, k_0, $k_0 + 1$, and $k_0 + 2$.

If this happens, the processes in S receive the same votes in round k_0 and hence choose the same value, say v_0, in round k_0. All processes in S vote for v_0 in round $k_0 + 1$, implying that each process in S receives $N - t$ votes for v_0 in round $k_0 + 1$. This implies that the processes in S witness for v_0 in round $k + 2$; hence they all receive $N - t > t$ witnesses for v_0 in round $k_0 + 2$, and all decide for v_0 in that round. It follows that

$$\begin{aligned}&\mathbf{Pr}[\text{ Processes in } S \text{ have not decided in round } k+2\ ]\\ &\quad \leq \psi \times \mathbf{Pr}[\text{ Processes in } S \text{ have not decided before round } k\ ],\end{aligned}$$

which implies the result. □

Lemma 14.23 *If all processes start the algorithm with input v, then all processes decide for v in round 2.*

Proof. All processes receive only votes for v in round 1, so all processes witness for v in round 2. This implies that they all decide on v in that round. □

Theorem 14.24 *Algorithm 14.3 is a probabilistic, t-crash-robust consensus protocol for $t < N/2$.*

Proof. Convergence was shown in Lemma 14.22 and agreement in Lemma 14.21; non-triviality is implied by Lemma 14.23. □

The dependence of the decision on the input values is further analyzed in Exercise 14.11.

14.4.2 Byzantine-robust Consensus Protocols

The Byzantine failure model is more malicious than the crash model, because Byzantine processes may execute arbitrary state transitions and may send messages that are in disaccordance with the algorithm. In the sequel, we use the notation $\gamma \rightsquigarrow \delta$ (or $\gamma \rightsquigarrow_S \delta$) to denote that there is a sequence of *correct steps*, i.e., transitions of the protocol (in processes of S), leading the system from γ to δ. Similarly, γ is reachable if there is a sequence of correct steps leading from an initial configuration to γ. The maliciousness of the Byzantine model implies a lower maximum on the resilience than for the crash model.

Theorem 14.25 *There is no t-Byzantine-robust consensus protocol for $t \geq N/3$.*

Proof. Assume, to the contrary, that such a protocol exists. Again it is left to the reader to show the existence of a bivalent initial configuration of any such protocol (exploit, as usual, the non-triviality).

The high resilience of the protocol implies that two sets S and T of processes can be chosen such that $|S| \geq N - t$, $|T| \geq N - t$, and $|S \cap T| \leq t$. In words, both S and T are sufficiently large to survive independently, but their intersection can be entirely malicious. This is exploited to show that no bivalent configurations are reachable.

Claim 14.26 *A reachable configuration γ is either both S-0-valent and T-0-valent, or both S-1-valent and T-1-valent.*

Proof. As γ is reached by a sequence of correct steps, all possibilities for choosing a set of t processes that fail are still open. Assume, on the contrary, that *different* decisions can be reached by S and T, i.e., $\gamma \rightsquigarrow_S \delta_v$ and $\gamma \rightsquigarrow_T \delta_{\bar{v}}$, where δ_v ($\delta_{\bar{v}}$) is a configuration where all processes in S (in T) have decided on v ($\bar{v}$). An inconsistent state can be reached by assuming that the processes in $S \cap T$ are malicious and combining schedules as follows. Starting from configuration γ, the processes in $S \cap T$ cooperate with the other processes in S as in the sequence leading to a v-decision in S. When this decision has been taken by the processes in S, the malicious processes restore their state as in configuration γ, and subsequently cooperate with the processes in T as in the sequence leading to an $\bar{v}$ decision in T. This results in a configuration in which correct processes have decided differently, which conflicts with the agreement requirement. □

Claim 14.27 *There is no reachable bivalent configuration.*

Proof. Let a reachable bivalent configuration γ be given and assume, w.l.o.g., that γ is both S-1-valent and T-1-valent (Claim 14.26). However, γ is bivalent, so a 0-decided configuration δ_0 is also reachable (clearly in cooperation between S and T) from γ. In the sequence of configurations from γ to δ_0 there are two subsequent configurations γ_1 and γ_0, where γ_v is both S-v-valent and T-v-valent. Let p be the process causing the transition from γ_1 to γ_0. Now $p \in S$ is impossible because γ_1 is S-1-valent and γ_0 is S-0-valent; similarly $p \in T$ is impossible. This is a contradiction. □

Again, the last claim contradicts the existence of bivalent initial configurations. Thus Theorem 14.25 is proved. □

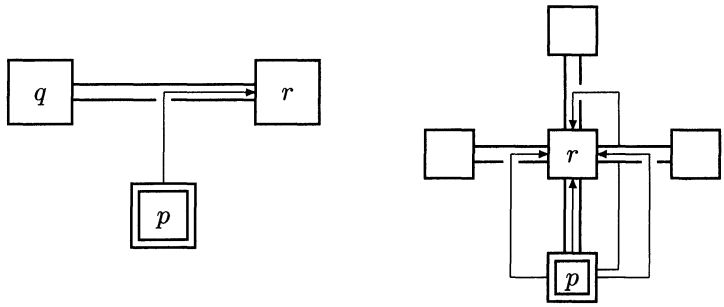

Figure 14.4 BYZANTINE PROCESS SIMULATING OTHER PROCESSES.

The Byzantine-robust consensus algorithm of Bracha and Toueg. For $t < N/3$, t-Byzantine-robust consensus protocols do exist. It is necessary that the communication system allows a process to determine by what process a received message was sent. If Byzantine process p can send to correct process r a message and successfully pretend that r receives the message *from* q (see Figure 14.4), the problem becomes unsolvable. Indeed, process p can simulate sufficiently many correct processes to enforce an incorrect decision in process r.

Like the crash-robust protocol, the Byzantine-robust protocol (Algorithm 14.5) operates in rounds. In each round, every process can submit votes, and a decision is taken when sufficiently many processes vote for the same value. The lower resilience ($t < N/3$) eliminates the necessity to distinguish between witnesses and non-witnesses; a process decides upon accepting more than $(N + t)/2$ votes for the same value.

The maliciousness of the fault model necessitates, however, the introduction of a vote-verification mechanism, which is the crux of the protocol. Without such a mechanism a Byzantine process can disturb the voting among correct processes by sending *different* votes to various correct processes. Such malicious behavior is not possible in the crash model. The verification mechanism ensures that, although Byzantine process r can send different votes to correct processes p and q, it cannot fool p and q into *accepting* different votes for r (in some round).

The verification mechanism is based on echoing messages. A process shouts its vote (as *initial*, **in**), and each process, upon receiving the first vote for some process in some round, echoes the vote (as *echo*, **ec**). A process will accept a vote if more than $(N+t)/2$ echo messages for it have been

```
var value_p          : (0, 1)    init x_p ;
    round_p          : integer   init 0 ;
    msgs_p[0..1]     : integer   init 0 ;
    echos_p[ℙ, 0..1] : integer   init 0 ;

while true do
    begin forall v, q do begin msgs_p[v] := 0 ; echos_p[q, v] := 0 end ;
        shout ⟨ vote, in, p, value_p, round_p ⟩ ;
        (* Now accept N − t votes for the current round *)
        while msgs_p[0] + msgs_p[1] < N − t do
            begin receive ⟨ vote, t, r, v, rn ⟩ from q ;
                if ⟨ vote, t, r, *, rn ⟩ has been received from q already
                  then skip    (* q repeats, must be Byzantine *)
                else if t = in and q ≠ r
                  then skip    (* q lies, must be Byzantine *)
                else if rn > round_p
                  then (* Process message in later round *)
                       send ⟨ vote, t, r, v, rn ⟩ to p
                else (* Process or echo vote message *)
                  case t of
                       in : shout ⟨ vote, ec, r, v, rn ⟩
                       ec : if rn = round_p then
                                begin echos_p[r, v] := echos_p[r, v] + 1 ;
                                      if echos_p[r, v] = ⌊(N + t)/2⌋ + 1
                                        then msgs_p[v] := msgs_p[v] + 1
                                end
                            else skip (* Old message *)
                  esac
            end ;
        (* Choose value for next round *)
        if msgs_p[0] > msgs_p[1] then value_p := 0 else value_p := 1 ;
        if msgs_p[value_p] > (N + t)/2 then y_p := value_p ;
        round_p := round_p + 1
    end
```

Algorithm 14.5 BYZANTINE-ROBUST CONSENSUS ALGORITHM.

received. The verification mechanism preserves the (partial) correctness of communication between correct processes (Lemma 14.28), and correct processes never accept different votes for the same process (Lemma 14.29). No deadlocks are introduced (Lemma 14.30).

We say that process p *accepts a v-vote* for process r in round k if p increments $msgs_p[v]$ upon receipt of a vote message $\langle \mathbf{vote}, \mathbf{ec}, r, v, k \rangle$. The algorithm ensures that p passes round k only after acceptance of $N - t$ votes, and also that p accepts at most one vote for each process in each round.

Lemma 14.28 *If correct process p accepts in round k the vote v for correct process r, then r has voted for v in round k.*

Proof. Process p accepts the vote upon receipt of a $\langle\, \mathbf{vote}, \mathbf{ec}, r, v, k \,\rangle$ message from more than $(N+t)/2$ (different) processes; at least one correct process s has sent such a message to p. Process s sends the echo to p upon receipt of a $\langle\, \mathbf{vote}, \mathbf{in}, r, v, k \,\rangle$ from r, which implies, because r is correct, that r votes for v in round k. □

Lemma 14.29 *If correct processes p and q accept a vote for process r in round k, they accept the same vote.*

Proof. Assume that in round k process p accepts a v-vote for r, and process q accepts a w-vote. Thus, p has received a $\langle\, \mathbf{vote}, \mathbf{ec}, r, v, k \,\rangle$ from more than $(N+t)/2$ processes, and q has received a $\langle\, \mathbf{vote}, \mathbf{ec}, r, w, k \,\rangle$ from more than $(N+t)/2$ processes. Because there are only N processes, more than t processes must have sent a $\langle\, \mathbf{vote}, \mathbf{ec}, r, v, k \,\rangle$ to p *and* a $\langle\, \mathbf{vote}, \mathbf{ec}, r, w, k \,\rangle$ to q. This implies that at least one correct process did so, and hence that $v = w$. □

Lemma 14.30 *If all correct processes start round k, then all correct processes accept sufficiently many votes in that round to complete the round.*

Proof. Correct process r starting round k with $value_r = v$ shouts an initial vote for that round, which is echoed by all correct processes. Thus, for correct processes p and r, a $\langle\, \mathbf{vote}, \mathbf{ec}, r, v, k \,\rangle$ is sent to p by at least $N-t$ processes, allowing p to accept the v-vote for r in round k unless $N-t$ other votes are accepted earlier. It follows that process p accepts $N-t$ votes in this round. □

The proof of correctness of the protocol now follows similar lines to the correctness proof of the crash-robust protocol.

Lemma 14.31 *If a correct process decides on v in round k, then all correct processes choose v in round k and all later rounds.*

Proof. Let S be the set of at least $(N+t)/2$ processes for which p accepts a v-vote in round k. Correct process q accepts, in round k, $N-t$ votes, including at least $|S| - t > (N-t)/2$ votes for processes in S. By Lemma 14.29, q accepts more than $(N-t)/2$ v-votes, which implies that q chooses v in round k.

To show that all correct processes choose v in later rounds, assume that all correct processes choose v in some round l; hence, all correct processes vote

for v in round $l+1$. In round $l+1$ each correct process accepts $N-t$ votes, including more than $(N-t)/2$ votes for correct processes. By Lemma 14.28, a correct process accepts at least $(N-t)/2$ v-votes, and hence chooses v again in round $l+1$. □

Lemma 14.32 $\lim_{k\to\infty}$ **Pr**[Correct p has not decided before round k] $= 0$.

Proof. Let S be a set of at least $N-t$ correct processes and assume that p has not decided before round k. With probability $\psi > 0$, the processes in S all accept, in round k, votes for *the same* collection of $N-t$ processes and, in round $k+1$, only votes for processes in S. If this happens, the processes in S vote equally in round $k+1$, and decide in round $k+1$. It follows that

$$\begin{aligned}&\textbf{Pr}[\text{ Correct process } p \text{ has not decided before round } k+2\ ]\\ &\quad \leq \psi \times \textbf{Pr}[\text{ Correct process } p \text{ has not decided before round } k\ ],\end{aligned}$$

which implies the result. □

Lemma 14.33 *If all correct processes start the algorithm with input v, a decision for v is eventually taken.*

Proof. As in the proof of Lemma 14.31 it can be shown that all correct processes choose v again in every round. □

Theorem 14.34 *Algorithm 14.5 is a probabilistic, t-Byzantine-robust consensus protocol for $t < N/3$.*

Proof. Convergence was shown in Lemma 14.32 and agreement in Lemma 14.31; non-triviality is implied by Lemma 14.33. □

The dependence of the decision on the input values is further analyzed in Exercise 14.12. Algorithm 14.5 is described as an infinite loop for ease of presentation; we finally describe how the algorithm can be modified so as to terminate in every deciding process. After deciding on v in round k, process p exits the loop and shouts "multiple" votes $\langle\, \textbf{vote}, \textbf{in}, p, k^+, v \,\rangle$ and echoes $\langle\, \textbf{vote}, \textbf{ec}, *, k^+, v \,\rangle$. These messages are interpreted as initial and echoed votes for all rounds later than k. Indeed, p will vote for v in all later rounds, and so will all correct processes (Lemma 14.31). Hence the multiple messages are those that would be sent by p when continuing the algorithm, with a possible exception for the echoes of malicious initial votes.

14.5 Weak Termination

In this section the problem of an asynchronous Byzantine broadcast is studied. The goal of a broadcast is to make a value that is present in one process g, referred to as the *general*, known to all processes. Formally, the non-triviality requirement for a consensus protocol is strengthened by specifying that the decision value is the input of the general if the general is correct:

(3) **Dependence.** If the general is correct, all correct process decide on its input.

With this specification, however, the general becomes a single point of failure, implying that the problem is not solvable, as expressed in the following theorem.

Theorem 14.35 *There is no* 1*-Byzantine robust algorithm satisfying convergence, agreement, and dependence, even if convergence is required only if the general has sent at least one message.*

Proof. We consider two scenarios. In the first one the general is assumed Byzantine; the scenario serves to define a reachable configuration γ. A contradiction is then derived, in the second scenario.

(1) Assume that the general is Byzantine, and sends a message to initiate a broadcast of "0" to process p_0 and a message to initiate a broadcast of "1" to process p_1. Then the general stops. We call the resulting configuration γ.
By convergence, a decided configuration can be reached even if the general crashes; let $S = \mathbb{P} \setminus \{g\}$ and assume, w.l.o.g., that $\gamma \rightsquigarrow_S \delta_0$, where δ_0 is 0-decided.

(2) For the second scenario, assume that the general is correct and has input 1, and sends messages to initiate a broadcast of 1 to p_0 and p_1, after which its messages are delayed for a very long time. Now assume p_0 is Byzantine, and, after receipt of the message, changes its state to the state in γ, i.e., pretends to have received a 0-message from the general. As $\gamma \rightsquigarrow_S \delta_0$, a 0-decision can now be reached without interaction with the general, which is not allowed because the general is correct and has input 1.

□

The impossibility results from the possibility that the general initiates a broadcast and halts (first scenario) without providing sufficient information

about its input (as exploited in the second scenario). It will now be shown that a (deterministic) solution is possible if termination is required only in the case where the general is correct.

Definition 14.36 *A t-Byzantine-robust broadcast algorithm is an algorithm satisfying the following three requirements.*

(1) **Weak termination.** *All correct processes decide, or no correct process decides. If the general is correct, all correct processes decide.*
(2) **Agreement.** *If the correct processes decide, they decide on the same value.*
(3) **Dependence.** *If the general is correct, all correct processes decide on its input.*

The resilience of an asynchronous Byzantine broadcast algorithm can be shown to be bounded by $t < N/3$ by arguments similar to those used in the proof of Theorem 14.25. The broadcast algorithm of Bracha and Toueg [BT85], given as Algorithm 14.6, uses three types of vote messages: *initial* messages (type **in**), *echo* messages (type **ec**), and *ready* messages (type **re**). Each process counts, for each type and value, how many messages have been received, counting at most one message received from every process.

The general initiates the broadcast by shouting an initial vote. Upon receipt of an initial vote from the general, a process shouts an echo vote containing the same value. When more than $(N + t)/2$ echo messages with value v have been received, a ready message is shouted. The number of required echoes is sufficiently large to guarantee that no correct processes send ready messages for different values (Lemma 14.37). The receipt of more than t ready messages for the same value (implying that at least one correct process has sent such a message) also triggers the shouting of ready messages. The receipt of more than $2t$ ready messages for the same value (implying that more than t correct processes have sent such a message) causes a decision for that value. No provision is taken in Algorithm 14.6 to prevent a correct process from shouting a ready message twice, but the message is ignored by correct processes anyway.

Lemma 14.37 *No two correct processes send ready messages for different values.*

Proof. A correct process accepts at most one initial message (from the general), and hence sends echoes for at most one value.

Let p be the first correct process to send a ready message for v, and q the first correct process to send a ready message for w. Although a ready

```
var msgs_p[(in, ec, re), 0..1]    : integer    init 0 ;

For the general only:
    shout ⟨ vote, in, x_p ⟩

For all processes:
while y_p = b do
    begin receive ⟨ vote, t, v ⟩ from q;
        if a ⟨ vote, t, * ⟩ message has been received from q already
          then skip    (* q repeats, ignore *)
        else if t = in and q ≠ g
          then skip    (* q mimics g, must be Byzantine *)
        else begin msgs_p[t, v] := msgs_p[t, v] + 1 ;
                case t of
                    in : if msgs_p[in, v] = 1
                           then shout ⟨ vote, ec, v ⟩
                    ec : if msgs_p[ec, v] = ⌈(N + t)/2⌉ + 1
                           then shout ⟨ vote, re, v ⟩
                    re : if msgs_p[re, v] = t + 1
                           then shout ⟨ vote, re, v ⟩ ;
                         if msgs_p[re, v] = 2t + 1
                           then y_p := v
                esac
             end
    end
```

Algorithm 14.6 BYZANTINE-ROBUST BROADCAST ALGORITHM.

message can be sent upon receipt of sufficiently many ready messages, this is not the case for the first correct process that sends a ready message. This is so because $t + 1$ ready messages must be received before sending one, implying that a ready message from at least one correct process has been received already. Thus, p has received v-echoes from more than $(N + t)/2$ processes and q has received w-echoes from more than $(N + t)/2$ processes.

Because there are only N processes and $t < N/3$, there are more than t processes, including at least one correct process r, from which p has received a v-echo and q has received a w-echo. Because r is correct, $v = w$ is implied. □

Lemma 14.38 *If a correct process decides, then all correct processes decide, and on the same value.*

Proof. To decide on v, more than $2t$ ready messages must be received for v, which includes more than t ready messages from correct processes; Lemma 14.37 implies that the decisions will agree.

Assume correct process p decides on v; p has received more than $2t$ ready messages, including more than t messages from correct processes. A correct process sending a ready message to p sends this message to all processes, implying that all correct processes receive more than t ready messages. This in turn implies that all correct processes send a ready message, so that every correct process eventually receives $N - t > 2t$ ready messages and decides. □

Lemma 14.39 *If the general is correct, all correct processes decide on its input.*

Proof. If the general is correct it sends no initial messages with values different from its input. Consequently, no correct process will send echoes with values different from the general's input, which implies that at most t processes send these bad echoes. The number of bad echoes is insufficient for correct processes to send ready messages for bad values, which implies that at most t processes send bad ready messages. The number of bad ready messages is insufficient for a correct process to send ready messages or to decide, which implies that no correct process sends a bad ready message or decides incorrectly.

If the general is correct it sends an initial vote with its input to all correct processes, and all correct processes shout an echo with this value. Consequently, all correct processes will receive at least $N - t > (N + t)/2$ correct echo messages, and will shout a ready message with the correct value. Thus, all correct processes will receive at least $N - t > 2t$ correct ready messages, and will decide correctly. □

Theorem 14.40 *Algorithm 14.6 is an asynchronous t-Byzantine-robust broadcast algorithm for $t < N/3$.*

Proof. Weak termination follows from Lemmas 14.39 and 14.38, agreement from Lemma 14.38, and dependence from Lemma 14.39. □

Exercises to Chapter 14

Section 14.1

Exercise 14.1 *Omission of any of the three requirements of Definition 14.3 (termination, agreement, non-triviality) for the consensus problem allows a very simple solution. Show this by presenting the three simple solutions.*

Exercise 14.2 *In the proof of Lemma 14.6 it is assumed that each of the 2^N assignments of a bit to the N processes produces a possible input configuration.*
Give deterministic, 1-crash robust consensus protocols for each of the following restrictions on the input values.

(1) *It is given that the parity of the input is even (i.e., there are an even number of processes with input 1) in each initial configuration.*
(2) *There are two (known) processes r_1 and r_2, and each initial configuration satisfies $x_{r_1} = x_{r_2}$.*
(3) *In each initial configuration there are at least $\lceil (N/2) + 1 \rceil$ processes with the same input.*

Section 14.2

Exercise 14.3 *Show that there is no t-initially-dead-robust election algorithm for $t \geq N/2$.*

Section 14.3

Exercise 14.4 *Show that no algorithm for ϵ-approximate agreement can tolerate $t \geq N/2$ crashes.*

Exercise 14.5 *Give a bijection from the set*

$$\{(s, r) : N - t \leq s \leq N \text{ and } 1 \leq r \leq s\}$$

to integers in the range $[1, \ldots, K]$.

Project 14.6 *Is Algorithm 14.2 non-trivial?*

Exercise 14.7 *Adapt the proof of Theorem 14.15 for the case that G_T consists of k connected components.*

Exercise 14.8 *In this exercise we consider the problem of* $[k, l]$*-election, which generalizes the usual election problem. The problem requires that all correct processes decide on either* 0 *("defeated") or* 1 *("elected"), and that the number of processes that decide* 1 *is between* k *and* l *(inclusive).*

(1) *What are the uses of* $[k, l]$*-election?*
(2) *Demonstrate that no deterministic* 1*-crash robust algorithm for* $[k, k]$*-election exists (if* $0 < k < N$*).*
(3) *Give a deterministic* t*-crash robust algorithm for* $[k, k + 2t]$*-election.*

Section 14.4

Exercise 14.9 *Does the convergence requirement imply that the expected number of steps is bounded?*
Is the expected number of steps bounded in all algorithms of this section?

Exercise 14.10 *Show that if all correct processes start round* k *of the crash-robust consensus algorithm (Algorithm 14.3), then all correct processes will also finish round* k*.*

Exercise 14.11

(1) *Prove, that if more that* $(N + t)/2$ *processes start the crash-robust consensus algorithm (Algorithm 14.3) with input* v*, then a decision for* v *is taken in three rounds.*
(2) *Prove, that if more than* $(N - t)/2$ *processes start the algorithm with input* v*, then a decision for* v *is possible.*
(3) *Is a decision for* v *possible if exactly* $(N - t)/2$ *processes start the algorithm with input* v*?*
(4) *What are the bivalent input configurations of the algorithm?*

Exercise 14.12

(1) *Prove that, if more than* $(N + t)/2$ *correct processes start Algorithm 14.5 with input* v*, a* v*-decision is eventually taken.*
(2) *Prove that, if more than* $(N + t)/2$ *correct processes start Algorithm 14.5 with input* v *and* $t < N/5$*, a* v*-decision is taken within two rounds.*

Section 14.5

Exercise 14.13 *Prove that no asynchronous t-Byzantine-robust broadcast algorithm exists for* $t \geq N/3$.

Exercise 14.14 *Prove that during the execution of Algorithm 14.6 at most* $N(3N + 1)$ *messages are sent by correct processes.*

15

Fault Tolerance in Synchronous Systems

The previous chapter has studied the degree of fault tolerance achievable in completely asynchronous systems. Although a reasonable robustness is attainable, reliable systems in practice are always synchronous in the sense of relying on the use of timers and upper bounds on the message-delivery time. In these systems a higher degree of robustness is attainable, the algorithms are simpler, and the algorithms guarantee an upper bound on the response time in most of the cases.

The synchrony of the system makes it impossible for faulty processes to confuse correct processes by not sending information; indeed, if a process does not receive a message when expected, a default value is used instead, and the sender becomes suspected of being faulty. Thus, crashed processes are detected immediately and pose no difficult problems in synchronous systems; we concentrate on Byzantine failures in this chapter.

In Section 15.1 the problem of performing a broadcast in synchronous networks is studied; we present an upper bound ($t < N/3$) on the resilience, as well as two algorithms with optimal resilience. The algorithms are deterministic and achieve consensus; it is assumed that all processes know at what time the broadcast is initiated. Because consensus is not deterministically achievable in asynchronous systems (Theorem 14.8), it follows that in the presence of failures (even a single crash), synchronous systems exhibit a strictly stronger computational power than asynchronous ones.

Because crashing and not sending information are detected (and hence "harmless") in synchronous systems, Byzantine processes are only able to disturb the computation by sending erroneous information, either about their own state or by incorrectly forwarding information. In Section 15.2 it will be demonstrated that the robustness of synchronous systems can be further enhanced by providing methods for information authentication.

With these mechanisms it becomes impossible for a malicious process to "lie" about information received from other processes. It remains possible, though, to send inconsistent information about the process's own state. It is also shown that implementation of authentication is in practice possible using cryptographic techniques.

The algorithms in Sections 15.1 and 15.2 assume an idealized model of synchronous systems, in which the computation proceeds in pulses (also called *rounds*); see Chapter 12. The fundamentally higher resilience of synchronous systems over asynchronous systems implies the impossibility of any 1-crash-robust deterministic implementation of the pulse model in the asynchronous model. (Such an implementation, called a *synchronizer*, is possible in reliable networks; see Section 12.3).

Implementation of the pulse model is possible, however, in asynchronous bounded-delay networks (Subsection 12.1.3), where processes possess clocks and an upper bound on message delay is known. The implementation is possible even if the clocks drift and up to one-third of the processes may fail maliciously. The most difficult part of the implementation is to synchronize the process clocks reliably, a problem that will be discussed in Section 15.3.

15.1 Synchronous Decision Protocols

In this section we shall present algorithms for Byzantine-robust broadcast in synchronous (pulsed) networks; we start with a brief review of the model of pulsed networks as defined in Section 12.1.1. In a synchronous network the processes operate in pulses numbered 1, 2, 3, and so on; each process can execute an unbounded number of pulses as long as its local algorithm does not terminate. The initial configuration (γ_0) is described by the initial states of the processes, and the configuration after the ith pulse (denoted γ_i) is also described by the states of the processes. In pulse i, each process first sends a finite set of messages, depending on its state in γ_{i-1}. Subsequently each process receives all the messages sent to it in this pulse, and computes the new state from the old one and the collection of messages received in the pulse.

The pulse model is an idealized model of synchronous computations. The synchrony is reflected in

(1) the apparently simultaneous occurrence of the state transitions in the processes; and
(2) the guarantee that messages of a pulse are received before the state transitions of that pulse.

These idealized assumptions can be weakened to more realistic assumptions, namely (1) the availability of hardware clocks and (2) an upper bound on the message-delivery time. The resulting model of *asynchronous bounded delay networks* allows very efficient implementation of the pulse model (see Section 12.1.3). As shown in Chapter 12, the simultaneity of state transitions is only apparent. In an implementation of the model the state transitions may occur at different times, as long as timely receipt of all messages is guaranteed. In addition, the implementation should allow a process an unbounded number of pulses. The latter requirement excludes the implementations of Chapter 12 for use in fault-tolerant applications, because they all suffer from deadlock, most of them even in case of a single message loss. As already mentioned, robust implementation of the pulse model will be treated in Section 15.3.

Because the pulse model guarantees delivery of messages in the same pulse, a process is able to determine that a neighbor *did not send a message* to it. This feature is absent in asynchronous systems and makes a solution to the consensus problem, and even to the reliable-broadcast problem, possible in synchronous systems, as we shall see shortly.

In the Byzantine-broadcast problem, one distinct process g, called the *general*, is given an input x_g taken from a set V (usually $\{0, 1\}$). The processes different from the general are called the *lieutenants*. The following three requirements must be satisfied.

(1) **Termination.** Every correct process p will decide on a value $y_p \in V$.
(2) **Agreement.** All correct processes decide on the same value.
(3) **Dependence.** If the general is correct, all correct processes decide on x_g.

It is possible to require, in addition, **simultaneity**, i.e., that all correct processes decide in the same pulse. All algorithms discussed in this section and the following section satisfy simultaneity; see also Subsection 15.2.6.

15.1.1 A Bound on the Resilience

The resilience of synchronous networks against Byzantine failures is, as in the case of asynchronous networks (Theorem 14.25), bounded by $t < N/3$. The bound was first shown by Pease, Shostak, and Lamport [PSL80], by presenting several scenarios for an algorithm in the presence of $N/3$ or more Byzantine processes. Unlike the scenarios used in the proof of Theorem 14.25, here the correct processes receive contradictory information, allowing the conclusion that some processes are faulty. However, it turns out to be impossible

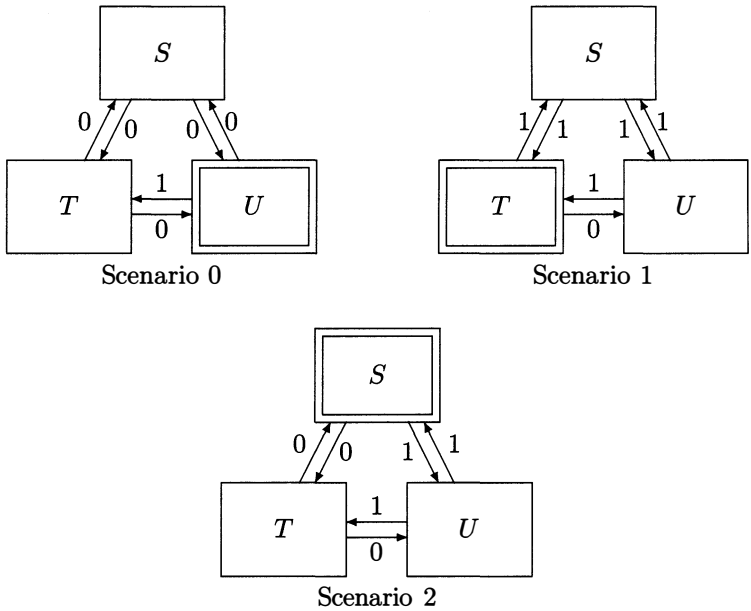

Figure 15.1 SCENARIOS FOR THE PROOF OF THEOREM 15.1.

to determine which processes are unreliable, and an incorrect decision can be enforced by the incorrect processes.

Theorem 15.1 *There is no t-Byzantine-robust broadcast protocol for $t \geq N/3$.*

Proof. As in earlier proofs, a resilience of $N/3$ or higher allows a partition of the processes into three groups (S, T, and U), each of which can be entirely faulty. The group containing the general is called S. A contradiction is derived by considering three scenarios, depicted in Figure 15.1, where the faulty group is indicated by a double box.

In scenario 0 the general broadcasts the value 0 and the processes in group U are faulty; in scenario 1 the general broadcasts a 1 and the processes in T are faulty. In pulse i of scenario 0, the processes of group U send to the processes in group T exactly those messages they would have sent (according to the protocol) in scenario 1. (That is, the messages sent in reaction to the messages received in pulse $i-1$ of scenario 1.) To the processes in S they send the messages directed by the protocol. Processes in S and T of course send the correct messages in all pulses. Observe, that in this scenario only

the processes in group U send incorrect messages, and the specifications of the protocol dictate that all correct processes, including group T, decide on 0.

Scenario 1 is defined similarly, but here the processes in T are faulty and send the messages they should have sent in scenario 0. In this scenario the processes in U decide on 1.

Finally consider scenario 2, where the processes in S are faulty and behave as follows. To the processes in T they send the messages of scenario 0 and to the processes in U they send the messages of scenario 1. It now can be shown by induction on the pulse number that the messages sent by T to U (or, from U to T) are exactly those sent in scenario 0 (or 1, respectively). Consequently, for the processes in T scenario 2 is indistinguishable from scenario 0 and for the processes in U it is indistinguishable from scenario 1. It follows that the processes in T decide on 0 and the processes in U decide on 1, a contradiction. □

It is used in the proof that Byzantine processes can send the messages of a 1-scenario, even if they have only the received messages of a 0-scenario. That is, the processes can "lie" not only about their own state, but also about the messages they have received. It is exactly this possibility that can be eliminated by using authentication, as discussed in Section 15.2; this leads to a resilience of $N - 1$.

15.1.2 Byzantine-broadcast Algorithm

In this subsection it will be shown that the upper bound on the resilience, shown in the previous subsection, is sharp. Moreover, contrasting the situation in asynchronous networks, the maximal resilience is attainable using deterministic algorithms. We present a recursive algorithm, also due to Pease *et al.* [PSL80], that tolerates t Byzantine failures for $t < N/3$. The resilience is a parameter of the algorithm.

The algorithm *Broadcast*$(N, 0)$ is given as Algorithm 15.2; it tolerates no failures ($t = 0$), and if no failures occur all processes decide on the input of the general in pulse 1. If a failure occurs, agreement may be violated, but termination (and simultaneity) is still guaranteed.

The protocol for resilience $t > 0$ (Algorithm 15.3) makes use of recursive calls to the procedure for resilience $t - 1$. The general sends its input to all lieutenants in pulse 1, and in the next pulse each lieutenant starts a broadcast of the received value to the other lieutenants, but this broadcast has resilience $t - 1$. This reduced resilience is a subtle point in the algo-

Pulse

1: The general sends $\langle$ **value**, x_g $\rangle$ to all processes,
the lieutenants do not send.
receive messages of pulse 1.
The general decides on x_g.
Lieutenants decide as follows:
if a message $\langle$ **value**, x $\rangle$ was received from g in pulse 1
then decide on x
else decide on *udef*

Algorithm 15.2 *Broadcast*$(N, 0)$.

rithm, because (if the general is correct) all t Byzantine processes may be found among the lieutenants, so the actual number of faults may exceed the resilience of the nested call of *Broadcast*. To prove the correctness of the resulting algorithm it is necessary to reason using the resilience t *and* the actual number of faulty processes f (see Lemma 15.3). In pulse $t+1$ the nested calls produce a decision, so lieutenant p decides in $N-1$ nested broadcasts. These $N-1$ decisions are stored in the array W_p, from which the decision of p is obtained by majority voting (the value received directly from the general is ignored here!). To this end a deterministic function *ma-*

Pulse

1: The general sends $\langle$ **value**, x_g $\rangle$ to all processes,
the lieutenants do not send.
receive messages of pulse 1.
Lieutenant p acts as follows.
if a message $\langle$ **value**, x $\rangle$ was received from g in pulse 1
then $x_p := x$ **else** $x_p :=$ *udef* ;
Announce x_p to the other lieutenants by acting as a general
in $Broadcast_p(N-1, t-1)$ in the next pulse

$(t+1)$: **receive messages of pulse** $t+1$.
The general decides on x_g.
For lieutenant p:
a decision occurs in $Broadcast_q(N-1, t-1)$ for each lieutenant q.
$W_p[q] :=$ decision in $Broadcast_q(N-1, t-1)$;
$y_p := major(W_p)$

Algorithm 15.3 *Broadcast*(N, t) (FOR $t > 0$).

jor is defined on arrays, with the property that if v has a majority in W, (i.e., more than half the entries equal v), then $major(W) = v$.

Lemma 15.2 (Termination) *If Broadcast*(N, t) *is started in pulse* 1, *every process decides in pulse* $t + 1$.

Proof. Because the protocol is recursive, its properties are proved using recursion on t.

In the algorithm *Broadcast*$(N, 0)$ (Algorithm 15.2), every process decides in pulse 1.

In the algorithm *Broadcast*(N, t), the lieutenants initiate recursive calls of the algorithm, *Broadcast*$(N-1, t-1)$, in pulse 2. When started in pulse 1, this algorithm decides in pulse t (this is the induction hypothesis), hence when started in pulse 2, all nested calls decide in pulse $t + 1$. In the same pulse, the decision in *Broadcast*(N, t) is taken. □

To prove dependence (also by induction) it is assumed that the general is correct, hence all t faulty processes are found among the $N - 1$ lieutenants. As $t < (N - 1)/3$ is not necessarily true, simple induction cannot be used, and we reason using the actual number of faults, denoted f.

Lemma 15.3 (Dependence) *If the general is correct, if there are* f *faulty processes, and if* $N > 2f + t$, *then all correct processes decide on the input of the general.*

Proof. In the algorithm *Broadcast*$(N, 0)$, if the general is correct all correct processes decide on the value of the general's input.

Now assume the lemma holds for *Broadcast*$(N-1, t-1)$. As the general is correct, it sends its input to all lieutenants in pulse 1, so every correct lieutenant q chooses $x_q = x_g$. Now $N > 2f+t$ implies $(N-1) > 2f+(t-1)$, so the induction hypothesis applies to the nested calls, even now all f faulty processes are found among the lieutenants. Thus, for correct lieutenants p and q, the decision of p in *Broadcast*$_q(N-1, t-1)$ equals x_q, i.e., x_g. But, as a strict majority of the lieutenants is correct $(N > 2f + t)$, process p will end with W_p in which a majority of the values equals x_g. Hence the application of *major* by p yields the desired value x_g. □

Lemma 15.4 (Agreement) *All correct processes decide on the same value.*

Proof. Because dependence implies agreement in the executions in which the general is correct, we now concentrate on the case where the general is

faulty. But then at most $t-1$ of the lieutenants are faulty, implying that the nested calls operate within the bounds of their resilience!

Indeed, $t < N/3$ implies $t-1 < (N-1)/3$, hence the nested calls satisfy agreement. So, all *correct* lieutenants will decide on the same value of x_q for *every* lieutenant q in the nested call $Broadcast_q(N-1, t-1)$. Thus, every correct lieutenant computes exactly the same vector W in pulse $t+1$, which implies that the application of *majority* gives the same result in every correct process. □

Theorem 15.5 *The Broadcast(N, t) protocol (Algorithm 15.2/15.3) is a t-Byzantine-robust broadcast protocol for $t < N/3$.*

Proof. Termination was shown in Lemma 15.2, dependence in Lemma 15.3, and agreement in Lemma 15.4. □

The *Broadcast* protocol decides in the $(t+1)$th pulse, which is optimal; see Subsection 15.2.6. Unfortunately, the message complexity is exponential; see Exercise 15.1.

15.1.3 A Polynomial Broadcast Algorithm

In this section we present a Byzantine broadcast algorithm by Dolev *et al.* [DFF+82], which uses only a polynomial number of messages and bits. The time complexity is higher than for the previous protocol; the algorithm requires $2t+3$ pulses to reach a decision. In the following description it will be assumed that $N = 3t+1$, and the case $N > 3t+1$ will be discussed later.

The algorithm uses two thresholds, $L = t+1$ and $H = 2t+1$. These numbers are chosen such that (1) each set of L processes contains at least one correct process, (2) each set of H processes contains at least L correct processes, and (3) there are at least H correct processes. Note that the assumption $N \geq 3t+1$ is necessary and sufficient to make the choice of L and H satisfying these three properties possible.

The algorithm exchanges messages of the type $\langle \mathbf{bm}, v \rangle$, where v is either the value 1, or the name of a process. (**bm** stands for "broadcast message".) Process p maintains a two-dimensional boolean table R, where $R_p[q, v]$ is true if and only if p has received a message $\langle \mathbf{bm}, v \rangle$ from process q. Initially all entries in the table are false, and we assume that the table is updated in the receive phase of every pulse (this is not shown in Algorithm 15.4). Observe that R_p is monotone in the pulses, i.e., if $R_p[q, v]$ becomes true in some pulse, it remains true in later pulses. Furthermore, as correct processes

only shout messages, we have for correct p, q, and r at the end of each pulse: $R_p[r, v] = R_q[r, v]$.

Unlike the *Broadcast* protocol of the previous subsection, the protocol of Dolev *et al.* is asymmetric in the values 0 and 1. A decision on 0 is the default value, and is chosen if insufficiently many messages are exchanged. If the general has input 1 it will shout $\langle \mathbf{bm}, 1 \rangle$ messages, and the receipt of sufficiently many echoing messages, of type $\langle \mathbf{bm}, q \rangle$, causes a process to decide on 1.

Three types of activity are relevant in the algorithm: *initiating*, *supporting*, and *confirming*.

(1) *Supporting.* Process p *supports* process q in pulse i if p has received sufficient evidence in earlier pulses that q has sent $\langle \mathbf{bm}, 1 \rangle$ messages; if this is the case, p will send $\langle \mathbf{bm}, q \rangle$ messages in pulse i. Process p is a *direct supporter* of q if p has received a $\langle \mathbf{bm}, 1 \rangle$ message from q. Process p is an *indirect supporter* of q if p has received a $\langle \mathbf{bm}, q \rangle$ message from at least L processes. The set S_p of processes supported by p is defined implicitly from R_p by

$$\begin{aligned} DS_p &= \{q : R_p[q, 1]\} \\ IS_p &= \{q : \#\{r : R_p[r, q]\} \geq L\} \\ S_p &= DS_p \cup IS_p \end{aligned}$$

The threshold for becoming an indirect supporter implies that if a correct process supports process q, then q has sent at least one $\langle \mathbf{bm}, 1 \rangle$ message. Indeed, assume some correct process supports q; let i be the first pulse in which this happens. As supporting q indirectly requires the receipt of at least one $\langle \mathbf{bm}, q \rangle$ message from a correct process in an earlier pulse, the first support for q by a correct process is direct. The direct support by a correct process implies that this process has received a $\langle \mathbf{bm}, 1 \rangle$ message from q.

(2) *Confirming.* Process p *confirms* process q upon receiving $\langle \mathbf{bm}, q \rangle$ messages from H processes, i.e.,

$$C_p = \{q : \#\{r : R_p[r, q]\} \geq H\}.$$

The choice of thresholds implies that if a correct process p confirms q, then all correct processes confirm q at most one pulse later. Indeed, assume p confirms q after pulse i. Process p has received $\langle \mathbf{bm}, q \rangle$ messages from H processes, including (by the choice of thresholds) at least L correct supporters for q. The correct supporters for q send the message $\langle \mathbf{bm}, q \rangle$ to all processes, implying that in pulse i all correct

processes receive at least L $\langle \mathbf{bm}, q \rangle$ messages, and support q in pulse $i+1$. Thus, in pulse $i+1$ all correct processes send $\langle \mathbf{bm}, q \rangle$, and as the number of correct processes is at least H, each correct process receives sufficient support to confirm q.

(3) *Initiating.* Process p *initiates* when it has sufficient evidence for the final decision value to be 1. After initiation, process p sends $\langle \mathbf{bm}, 1 \rangle$ messages. Initiation can be caused by three types of evidence, namely (1) p is the general and $x_p = 1$, (2) p receives $\langle \mathbf{bm}, 1 \rangle$ from the general in pulse 1, or (3) p has confirmed sufficiently many *lieutenants* at the end of a later pulse. The last possibility in particular requires some attention, because the number of confirmed lieutenants that is "sufficient" increases during the execution, and a confirmed general does not count for this rule. In the first three pulses L lieutenants need be confirmed to initiate, but starting from pulse 4 the threshold is incremented every two pulses. Thus, initiating according to rule (3) requires that by the end of pulse i, $Th(i) = L + \max(0, \lfloor i/2 \rfloor - 1)$ lieutenants are confirmed. The notation $C_p^{\mathbf{L}}$ in the algorithm denotes the set of confirmed lieutenants, i.e., $C_p \setminus \{g\}$. Initiation by p is represented by the boolean variable ini_p.

If a correct lieutenant r initiates at the end of pulse i, all correct processes confirm r at the end of pulse $i+2$. Indeed, r shouts $\langle \mathbf{bm}, 1 \rangle$ in pulse $i+1$, so all correct processes (directly) support r in pulse $i+2$, so every process receives at least H $\langle \mathbf{bm}, q \rangle$ messages in that pulse.

The algorithm continues for $2t+3$ pulses; if process p has confirmed at least H processes (here the general does count) by the end of that pulse, p decides 1, otherwise p decides 0. See Algorithm 15.4.

The general, being the only process that can enforce initiations (in other processes) on its own, holds a powerful position in the algorithm. It is easily seen that if the general initiates correctly, an avalanche of messages starts that causes all correct processes to confirm H processes and to decide on the value 1. Also, if it does not initiate there is no "critical mass" of messages that leads to the initiation by any correct process.

Lemma 15.6 *Algorithm 15.4 satisfies termination (and simultaneity) and dependence.*

Proof. It can be seen from the algorithm that all correct processes decide at the end of pulse $2t+3$, which shows termination and simultaneity. To show dependence, we shall assume that the general is correct.

```
var R_p[..., ...]  : boolean init false ;
    ini_p          : boolean init if p = g ∧ x_p = 1 then true else false ;

Pulse i: (* Sending phase *)
         if ini_p then shout ⟨ bm, 1 ⟩ ;
         forall q ∈ S_p do shout ⟨ bm, q ⟩ ;
         receive all messages of pulse i ;
         (* State update *)
         if i = 1 and R_p[g, 1] then ini_p := true ;
         if #C_p^L ≥ Th(i) then ini_p := true ;
         if i = 2t + 3 then   (* Decide *)
           if #C_p ≥ H then y_p := 1 else y_p := 0
```

Algorithm 15.4 RELIABLE-BROADCAST PROTOCOL.

If the general is correct and has input 1, it shouts a $\langle\, \mathbf{bm}, 1 \,\rangle$ message in pulse 1, causing every correct process q to initiate. Hence every correct process q shouts $\langle\, \mathbf{bm}, 1 \,\rangle$ in pulse 2, so that by the end of pulse 2 every correct process p supports all other correct processes. This implies that in pulse 3 every correct p shouts $\langle\, \mathbf{bm}, q \,\rangle$ for every correct q, so at the end of pulse 3 every correct process receives $\langle\, \mathbf{bm}, q \,\rangle$ from every other correct process, causing it to confirm q. Thus from the end of round 3 every correct process has confirmed H processes, implying that the final decision will be 1. (The general is supported and confirmed by all correct processes one pulse earlier than the other processes.)

If the general is correct and has input 0, it does not shout $\langle\, \mathbf{bm}, 1 \,\rangle$ in pulse 1, and neither does any other correct process. Assume no correct process has initiated in pulses 1 through $i - 1$; then no correct process sends $\langle\, \mathbf{bm}, 1 \,\rangle$ in pulse i. At the end of pulse i no correct process supports or confirms any correct process, because as we have seen earlier, this implies that the latter process has sent a $\langle\, \mathbf{bm}, 1 \,\rangle$ message. Consequently, no correct process initiates at the end of pulse i. It follows that no correct process initiates at all. This implies that no correct process ever confirms a correct process, so no correct process confirms more than t processes, and the decision at the final pulse is 0. □

We continue by proving agreement, and we shall assume in the following lemmas that the general is *faulty*. A sufficient "critical mass" of messages, leading inevitably to a 1-decision, is created by the initiation of L correct processes when there are at least four pulses to go.

Lemma 15.7 *If L correct processes initiate by the end of pulse i, where $i < 2t$, then all correct processes decide on the value* 1.

Proof. Let i be the first pulse at the end of which at least L correct processes initiate, and let A denote the set of correct processes that have initiated at the end of pulse i. All processes in A are lieutenants because the general is faulty. At the end of pulse $i+2$ all correct processes have confirmed the lieutenants in A; we show that at that time all correct processes initiate.

Case $i = 1$**:** All correct processes have confirmed the lieutenants of A by the end of pulse 3, and initiate because $\#A \geq L = Th(3)$.

Case $i \geq 2$**:** At least one process, say r, of A initiated in pulse i because it had confirmed $Th(i)$ lieutenants (initiation by receiving $\langle\, \mathbf{bm}, 1 \,\rangle$ from the general is only possible in pulse 1). Those $Th(i)$ lieutenants are confirmed by *all* correct processes at the end of pulse $i+1$, but r does not belong to these $Th(i)$ confirmed lieutenants, because r sends $\langle\, \mathbf{bm}, 1 \,\rangle$ messages for the first time in pulse $i+1$. However, all correct processes will have confirmed r at the end of pulse $i+2$; thus they have confirmed at least $Th(i)+1$ lieutenants at the end of round $i+2$. Finally, as $Th(i+2) = Th(i)+1$, all correct processes initiate.

Now, as all correct processes initiate by the end of round $i+2$, they are confirmed (by all correct processes) at the end of round $i+4$, hence all correct processes have confirmed at least H lieutenants. As $i < 2t$ was assumed, $i+4 \leq 2t+3$, so all correct processes decide on the value 1. □

For any correct process to decide on 1, an "avalanche" consisting of initiation by at least L correct processes is necessary. Indeed, a 1-decision requires the confirmation of at least H processes, including L correct ones, and these correct processes have initiated. The question is whether a Byzantine conspiracy can postpone the occurrence of an avalanche sufficiently long to trigger a 1-decision in some correct processes, without enforcing it in all according to Lemma 15.7. Of course, the answer is no, because there is a limit on how long a conspiracy can postpone an avalanche, and the number of pulses, $2t+3$, is chosen precisely so as to prevent this from happening. The reason is the growing threshold for the required number of confirmed processes; in the later pulses it becomes so high that already L initiated correct lieutenants have become necessary for a next correct process to initiate.

Lemma 15.8 *Assume at least L correct processes initiate during the algorithm, and let i be the first pulse at the end of which L correct processes initiate. Then $i < 2t$.*

Proof. To initiate in pulse $2t$ or higher, a correct process needs to have confirmed at least $Th(2t) = L + (t-1)$ lieutenants. As the general is faulty, there at most $t-1$ faulty lieutenants, so at least L correct lieutenants must have been confirmed, showing that already, in an earlier pulse, L correct processes must have initiated. □

Theorem 15.9 *The broadcast algorithm by Dolev et al. (Algorithm 15.4) is a t-Byzantine-robust broadcast protocol.*

Proof. Termination (and simultaneity as well) and dependence were shown in Lemma 15.6. To show agreement, assume that there is a correct process that decides on the value 1. We have observed that this implies that at least L correct processes have initiated. By Lemma 15.8, this happened for the first time in pulse $i < 2t$. But then, by Lemma 15.7, all correct processes decide on the value 1. □

To facilitate the presentation of the algorithm it has been assumed that processes repeat in every round the messages they have sent in earlier rounds. As correct processes record the messages received in earlier rounds, this is not necessary, so it suffices to send each message only once. In this way, each correct process sends each of the $N+1$ possible messages at most once to every other process, which bounds the message complexity to $\Theta(N^3)$. As there are only $N+1$ different messages, each message need contain only $O(\log N)$ bits.

If the number of processes exceeds $3t+1$, a collection of $3t$ *active lieutenants* is chosen to execute the algorithm. (The choice is done statically, e.g., by taking the $3t$ processes whose names follow g in the process name ordering.) The general and the active lieutenants inform the passive lieutenants about their decision, and the passive lieutenants decide on a value they receive from more than t processes. The message complexity of this layered approach is $\Theta(t^3 + t \cdot N)$, and the bit complexity is $\Theta(t^3 \log t + t\,N)$.

15.2 Authenticating Protocols

The malicious behavior considered up to now has included the incorrect forwarding of information in addition to the sending of incorrect information about the process's own state. Fortunately, this extremely malicious behavior of Byzantine processes can be restricted using cryptographic means, which would render Theorem 15.1 invalid. Indeed, in the scenarios used in its proof, faulty processes would send messages as in scenario 1, while having received only the messages of scenario 0.

In this section a means for digitally *signing* and *authenticating* messages is assumed. Process p sending message M adds to this message some additional information $S_p(M)$, called the *digital signature* of p for message M. Unlike handwritten signatures, the digital signature depends on M, which makes copying signatures into other messages useless. The signature scheme satisfies the following properties.

(1) If p is correct, only p can feasibly compute $S_p(M)$. This computation is referred to as *signing* message M.
(2) Every process can efficiently verify (given p, M, and S) whether $S = S_p(M)$. This verification is referred to as *authenticating* message M.

Signature schemes are based on *private* and *public keys.* The first assumption does not exclude that Byzantine processes conspire by revealing their secret keys to each other, allowing one Byzantine process to forge the signature of another. Only correct processes are assumed to keep their private keys secret.

We shall study the implementation of signature schemes in Subsections 15.2.2 through 15.2.5. In the next subsection, a message $\langle\, \mathbf{msg} \,\rangle$ signed by process p, i.e., the pair containing $\langle\, \mathbf{msg} \,\rangle$ and $S_p(\langle\, \mathbf{msg} \,\rangle)$, is denoted $\langle\, \mathbf{msg} \,\rangle : p$.

15.2.1 A Highly Resilient Protocol

An efficient Byzantine broadcast algorithm, using polynomially many messages and $t + 1$ pulses, was proposed by Dolev and Strong [DS83]. The authentication used in the protocol allows an unbounded resilience. We observe, though, that no more than N processes (out of N) can fail, and if N processes fail, all requirements are satisfied vacuously; hence let $t < N$. Their protocol is based on an earlier one by Lamport, Shostak, and Pease [LSP82], which is exponential in the number of messages. We present the latter protocol first.

In pulse 1 the general shouts the message $\langle\, \mathbf{value}, x_g \,\rangle : g$, containing its (signed) input.

In pulses 2 through $t+1$, processes sign and forward messages they received in the previous pulse; therefore, a message exchanged in pulse i contains i signatures. A message $\langle\, \mathbf{value}, v \,\rangle : g : p_2 : \cdots : p_i$ is called *valid* for the receiving process p if all the following hold.

(1) All i signatures are correct.

(2) The i signatures are from i different processes.
(3) p does not occur in the list of signatures.

During the algorithm, process p maintains a set W_p of values contained in valid messages received by p; initially this set is empty, and the value of each valid message is inserted in it.

The messages forwarded in pulse i are exactly the valid messages received in the previous pulse. At the end of pulse $t+1$, process p decides based on W_p. If W_p is the singleton $\{v\}$, p decides v; otherwise, p decides a default (e.g., 0). To save on the number of messages, p forwards the message $\langle \mathbf{value}, v \rangle : g : p_2 : \cdots : p_i : p$ only to the processes not occurring in the list $g, p_2, \ldots, p_i$. This modification has no influence on the behavior of the algorithm because for processes on the list the message is not valid.

Theorem 15.10 *The algorithm by Lamport, Shostak, and Pease is a correct Byzantine broadcast algorithm for $t < N$, using $t+1$ pulses.*

Proof. All processes decide in pulse $t+1$, implying both termination and simultaneity of the algorithm.

If the general is correct and has input v, all processes receive its message $\langle \mathbf{value}, x_g \rangle : g$ in pulse 1, so all correct processes include v in W. No other value is inserted in W, because no other value is ever signed by the general. Consequently, in pulse $t+1$ all processes have $W = \{v\}$ and decide on v, which implies dependence.

To show agreement, we shall derive that for correct processes p and q, $W_p = W_q$ at the end of pulse $t+1$. Assume $v \in W_p$ at the end of pulse $t+1$, and let i be the pulse in which p inserted v in W_p, on receipt of the message $\langle \mathbf{value}, v \rangle : g : p_2 : \cdots : p_i$.

Case 1: If q occurs in $g, p_2, \ldots, p_i$, then q has itself seen the value v and inserted it in W_q.

Case 2: If q does not occur in the sequence $g, p_2, \ldots, p_i$ and $i \leq t$, then p forwards the message $\langle \mathbf{value}, v \rangle : g : p_2 : \cdots : p_i : p$ to q in pulse $i+1$, so q validates v at the latest in pulse $i+1$.

Case 3: If q does not occur in $g, p_2, \ldots, p_i$ and $i = t+1$, observe that the message received by p was signed by $t+1$ consecutive processes, including at least one correct process. This process forwarded the message to all other processes, including q, so q sees v.

As $W_p = W_q$ by the end of pulse $t+1$, p and q decide equally. □

It is not possible to terminate the algorithm earlier than in pulse $t+1$. In all pulses up to t, a correct process could receive messages created and

forwarded only by faulty processors, and not sent to other correct processes, which could lead to inconsistent decisions.

The intermediate result in the previous algorithm, namely agreement on a set of values among all correct processes, is stronger than necessary to achieve agreement on a single value. This was observed by Dolev and Strong [DS83], who proposed a more efficient modification. It is in fact sufficient that at the end of pulse $t+1$, either (a) for every correct p the set W_p is *the same* singleton, or (b) for no correct p the set W_p is a singleton. In the first case all processes decide v, in the latter case they all decide 0 (or, if it is desired to modify the algorithm in this way, they decide "general faulty").

The weaker requirement on the sets W is achieved by the algorithm of Dolev and Strong. Instead of relaying every valid message, process p forwards at most two messages, namely one message with the first and one message with the second value accepted by p. A complete description of the algorithm is left to the reader.

Theorem 15.11 *The algorithm of Dolev and Strong as described above is a Byzantine-broadcast protocol using $t+1$ pulses and at most $2N^2$ messages.*

Proof. Termination and simultaneity are as for the previous protocol, because each correct process decides at the end of pulse $t+1$. Dependence also follows as in the previous protocol. If g correctly shouts v in the first pulse, all correct processes accept v in that pulse and no other value is ever accepted; hence all correct processes decide on v. The claimed message complexity follows from the fact that each (correct) process shouts at most two messages.

To show agreement, we shall show that for correct processes p and q, W_p and W_q satisfy at the end of pulse $t+1$ the following.

(1) If $W_p = \{v\}$ then $v \in W_q$.
(2) If $\#W_p > 1$ then $\#W_q > 1$.

For (1): Assume that p accepted the value v upon receipt of the message $\langle \mathbf{value}, v \rangle : g : p_2 : \cdots : p_i$ in pulse i, and reason as in the proof of Theorem 15.10:

Case 1: If q occurs among $g, \ldots, p_i$, q has clearly accepted v.

Case 2: If q does not occur among $g, \ldots, p_i$ and $i \leq t$, then p forwards the value to q, which will accept it in this case.

Case 3: If q does not occur and $i = t+1$, at least one of the processes that signed the message, say r, is correct. Process r has forwarded the value v to q as well, implying that v is in W_q.

For (2): Assume that $\#W_p > 1$ at the end of the algorithm, and let w be the second value accepted by p. Again by similar reasoning, it can be shown that $w \in W_q$, which implies that $\#W_q > 1$. (Equality of W_p and W_q cannot be derived because process p will not forward its third accepted value or later ones.)

Having proved (1) and (2), assume that correct process p decides on $v \in W_p$, that is, $W_p = \{v\}$. Then, by (1), v is contained in all W_q for correct q, but it follows that W_q is not larger than the singleton $\{v\}$; otherwise W_p is not a singleton, by (2). Therefore, every correct process q also decides on v. Further, assume that correct process p decides on a default because W_p is not a singleton. If W_p is empty, every correct q has W_q empty by (1) and if $\#W_p > 1$, then $\#W_q > 1$ by (2); consequently, q also decides on a default. □

Dolev and Strong have further improve the algorithm and obtained an algorithm that solves the Byzantine-broadcast problem in the same number of pulses and only $O(N\,t)$ messages.

15.2.2 Implementation of Digital Signatures

Because p's signature $S_p(M)$ should constitute sufficient evidence that p is the originator of the message, the signature must consist of some form of information, which

(1) can be computed efficiently by p (signed);
(2) cannot be computed efficiently by any other process than p (forged).

We should remark immediately that, for most of the signature schemes in use today, the second requirement is not proved to the extent that the problem of forgery is shown to be exponentially hard. Usually, the problem of forging is shown to be related to (or sometimes equivalent to) some computational problem that has been studied for a long time without being known to be polynomially solvable. For example, forging signatures in the Fiat–Shamir scheme allows to factor large integers; as the latter is (presumably) computationally hard, the former must also be computationally hard.

Signature schemes have been proposed based on various supposedly hard problems, such as computing the discrete logarithm, factoring large numbers, knapsack problems. Requirements (1) and (2) imply that process p must have a computational "advantage" over the other processes; this advantage is some secret information, held by p and referred to as p's *secret* (or private) *key*. So, the computation of $S_p(M)$ is efficient when the secret

key is known, but (presumably) difficult without this information. Clearly, if p succeeds in keeping its key secret, this implies that only p can tractably compute $S_p(M)$.

All processes must be able to verify signatures, that is, given a message M and a signature S, it must be possible to verify efficiently that S has indeed been computed from M using p's secret key. This verification requires that some information is revealed regarding p's secret key; this information is referred to as p's *public key.* The public key should allow verification of the signature, but it should be impossible or at least computationally hard to use it to compute p's secret key or forge signatures.

The most successful signature schemes proposed up to date are based on number-theoretic computations in the arithmetic rings modulo large numbers. The basic arithmetic operations of addition, multiplication, and exponentiation can be performed in these rings in times polynomial in the length (in bits) of the modulus. Division is possible if the denominator and modulus are coprime (i.e., have no common prime factors) and can also be performed in polynomial time. Because signing and verification require computations on the message, M is interpreted as a large number.

15.2.3 The ElGamal Signature Scheme

ElGamal's signature scheme [ElG85] is based on a number-theoretic function called the *discrete logarithm.* For a large prime number P, the multiplicative group modulo P, denoted $\mathbb{Z}_P^*$, contains $P-1$ elements and is *cyclic.* The latter means that an element $g \in \mathbb{Z}_P^*$ can be chosen such that the $P-1$ numbers

$$g^0 = 1,\ g^1,\ g^2,\ \ldots,\ g^{P-3},\ g^{P-2}$$

are all different and hence, enumerate all elements of $\mathbb{Z}_P^*$. Such a g is called a *generator* of $\mathbb{Z}_P^*$, or also a *primitive root* modulo P. The generator is not unique; usually there are many of them. Given fixed P and generator g, for each $x \in \mathbb{Z}_P^*$ there is a *unique* integer i modulo $P-1$ such that $g^i = x$ (equality in $\mathbb{Z}_P^*$). This i is called the *discrete logarithm* (sometimes *index*) of x. Unlike the basic arithmetic operations mentioned above, computation of the discrete log is *not* easy. It is a well-studied problem for which no efficient general solution has been found to date, but neither has the problem been shown to be intractable; see [Odl84] for an overview of results.

The signature scheme of ElGamal [ElG85] is based on the difficulty of computing discrete logarithms. The processes share a large prime number P and a primitive root g of $\mathbb{Z}_P^*$. Process p chooses as its secret key a number

d, randomly between 1 and $P-2$, and the public key of p is the number $e = g^d$; observe that d is the discrete log of e. The signature of p can be computed efficiently knowing the log of e, and therefore forms an implicit proof that the signer knows d.

A valid signature for message M is a pair (r, s) satisfying $g^M = e^r r^s$. Such a pair is easily found by p using the secret key d. Process p selects a random number a, coprime with $P-1$, and computes

$$r = g^a \pmod{P}$$

and

$$s = (M - dr)a^{-1} \pmod{(P-1)}.$$

These numbers indeed satisfy

$$\begin{aligned} e^r \cdot r^s &= e^r (g^a)^{(M-dr)a^{-1}} \\ &= g^{dr} g^{M-dr} = g^M. \end{aligned}$$

(All equalities are in $\mathbb{Z}_P^*$.) The validity of a signature $S = (r, s)$ for message M is easily verified by checking whether $g^M = e^r r^s$.

Algorithms for the discrete logarithm. Because p's secret key, d, equals the discrete logarithm of its public key, e, the scheme is broken if discrete logarithms modulo P can be computed efficiently. To date, no efficient algorithm to do this in the general case or to forge signatures in any other way is known.

A general algorithm for computing discrete logarithms was presented by Odlyzko [Odl84]. Its complexity is of the same order of magnitude as the best-known algorithms for factoring integers as big as P. The algorithm first computes several tables using only P and g, and, in a second phase, computes logarithms for given numbers. If Q is the largest prime factor of $P-1$, the time for the first phase and the size of the tables are of the order of Q; therefore it is desirable to select P such that $P-1$ has a large prime factor. The second phase, computing logarithms, can be performed within seconds even on very small computers. Therefore it is necessary to change P and g sufficiently often, say every month, so that the tables for a particular P are obsolete before their completion.

Randomized signing. The randomization in the signing procedure makes each one of $\phi(P-1)$ different signatures[1] for a given message an equally likely outcome of the signing procedure. Thus, the same document signed

[1] The function ϕ is known as *Euler's phi function*; $\phi(n)$ is the size of $\mathbb{Z}_n^*$.

twice will almost certainly produce two different valid signatures. Randomization is essential in the signing procedure; if p signs two messages using the same value of a, p's secret key can be computed from the signatures; see Exercise 15.6.

15.2.4 The RSA Signature Scheme

If n is a large number, the product of two prime numbers P and Q, it is very hard to compute square and higher-order roots modulo n unless the factorization is known. The ability to compute square roots can be exploited to find the factors of n (see Exercise 15.7), which shows that square rooting is as hard as factoring.

In the signature scheme by Rivest, Shamir and Adleman [RSA78], the public key of p is a large number n, of which p knows the factorization, and an exponent e. The signature of p for message M is the eth root of M modulo n, which is easily verified using exponentiation. This higher-order root is found by p using exponentiation as well; when generating its key, p computes a number d such that $de = 1 \pmod{(\phi(n))}$, which implies that $(M^d)^e = M$, that is, M^d is an eth root of M. The secret key of p consists only of the number d, i.e., p need not memorize the factorization of n.

In the RSA scheme, p shows its identity by computing roots modulo n, which requires (implicit) knowledge of a factorization of n; only p is supposed to have this knowledge. In this scheme, every process uses a different modulus.

15.2.5 The Fiat–Shamir Signature Scheme

A more subtle use of the difficulty of (square) rooting is made in a scheme by Fiat and Shamir [FS86]. In the RSA scheme, a process signs by showing that it is able to compute roots modulo its public key, and the ability to compute roots presumably requires knowledge of the factorization. In the Fiat–Shamir scheme the processes make use of a *common* modulus n, of which the factorization is known only to a trusted center. Process p is given the square roots of some specific numbers (depending on p's identity), and the signature of p for M provides evidence that the signer knows these square roots, but without revealing what they are.

An advantage of the Fiat–Shamir scheme over the RSA scheme is the lower arithmetic complexity and the absence of a separate public key for every process. A disadvantage is the necessity of a trusted authority that issues the secret keys. As mentioned before, the scheme uses a large integer

n, the product of two large prime numbers known only to the center. In addition there is a one-way pseudo-random function f mapping strings to $\mathbb{Z}_n$; this function is known and can be computed by every process, but its inverse cannot be computed feasibly.

The secret and public keys. As its secret key, p is given the square roots s_1 through s_k of k numbers modulo n, namely $s_j = \sqrt{v_j^{-1}}$, where $v_j = f(p, j)$. The v_j can be considered as the public keys of p, but as they can be computed from p's identity they need not be stored. To avoid some technical nuisance we assume that these k numbers are all quadratic residues modulo n. The square roots can be computed by the center, which knows the factors of n.

Signing messages: first attempt. The signature of p implicitly proves that the signer knows the roots of the v_j, i.e., can provide a number s such that $s^2 v_j = 1$. Such a number is s_j, but sending s_j itself would reveal the secret key; to avoid revealing the key, the scheme uses the following idea. Process p selects a random number r and computes $x = r^2$. Now p is the only process that can provide a number y satisfying $y^2 v_j = x$, namely, $y = r s_j$. Thus, p may demonstrate its knowledge of s_j without revealing it by sending a pair (x, y) satisfying $y^2 v_j = x$. As p does not send the number r, computing s_j from the pair is not possible without computing a square root.

But there are two problems with signatures consisting of such pairs. First, anybody can produce such a pair by cheating in the following way: select y *first* and compute $x = y^2 v$ subsequently. Second, the signature does not depend on the message so a process that has received a signed message from p can copy the signature onto any forged message. The crux of the signature scheme is to have p demonstrate knowledge of the root of *the product of a subset of the* v_j, where the subset depends on the message and the random number. Scrambling the message and the random number through f prevents a forger from selecting y first.

To sign message M, p acts as follows.

(1) p selects a random number r and computes $x = r^2$.
(2) p computes $f(M, x)$; call the first k bits e_1 through e_k.
(3) p computes $y = r \prod_{e_j=1} s_j$.

The signature $S_p(M)$ consists of the tuple $(e_1, \ldots, e_k, y)$.

To verify the signature $(e_1, \ldots, e_k, y)$ of p for message M, act as follows.

(1) Compute the v_j and $z = y^2 \prod_{e_j=1} v_j$.
(2) Compute $f(M, z)$ and verify that the first k bits are e_1 through e_k.

If the signature is computed genuinely, the value of z computed in the first step of verification equals the value of x used in signing and hence the first k bits of $f(M, z)$ equal e_1 through e_k.

Forgery and final solution. We now consider a strategy for a forger to obtain a signature according to the above scheme without knowing the s_j.

(1) Choose k random bits e_1 through e_k.
(2) Choose a random number y and compute $x = y^2 \prod_{e_j=1} v_j$.
(3) Compute $f(M, x)$ and see whether the first k bits equal the values of e_1 through e_k chosen earlier. If so, $(e_1, \ldots, e_k, y)$ is the forged signature for message M.

As the probability of equality in step (3) can be assumed to be 2^{-k}, forgery succeeds after an expected number of 2^k trials.

With $k = 72$ and an assumed time of 10^{-9} seconds to try one choice of the e_j, the expected time for forgery (with this strategy) is $2^{72} \cdot 10^{-9}$ seconds or 1.5 million years, which makes the scheme very safe. However, each process must store k roots, and if k must be limited due to space restrictions, an expected 2^k forgery time may not be satisfactory. We now show how to modify the scheme so as to use k roots, and obtain a 2^{kt} expected forgery time for a chosen integer t. The idea is to use the first kt bits of an f-result to define t subsets of the v_j, and have p demonstrate its knowledge of the t products of these. To sign message M, p acts as follows.

(1) p selects random $r_1, \ldots, r_t$ and computes $x_i = r_i^2$.
(2) p computes $f(M, x_1, \ldots, x_t)$; call the first kt bits e_{ij} $(1 \leq i \leq t$ and $1 \leq j \leq k)$.
(3) p computes $y_i = r_i \prod_{e_{ij}=1} s_j$ for $1 \leq i \leq t$. The signature $S_p(M)$ consists of $(e_{11}, \ldots, e_{tk}, y_1, \ldots, y_t)$.

To verify the signature $(e_{11}, \ldots, e_{tk}, y_1, \ldots, y_t)$ of p for message M, act as follows.

(1) Compute the v_j and $z_i = y_i^2 \prod_{e_{ij}=1} v_j$.
(2) Compute $f(M, z_1, \ldots, z_t)$ and verify that the first kt bits are e_{11} through e_{tk}.

A forger trying to produce a valid signature with the same strategy as above now has a probability of 2^{-kt} of success in its third step, implying an expected number of 2^{kt} of trials. Fiat and Shamir show in their paper that

unless factoring n turns out to be easy, no essentially better forgery algorithm exists, hence the scheme can be made arbitrarily safe by choosing k and t sufficiently large.

15.2.6 Summary and Discussion

In this and the previous section it has been demonstrated that in synchronous systems there exist deterministic solutions to the Byzantine broadcast problem. The maximal resilience of such solutions is $t < N/3$ if no authentication is used (Section 15.1), and unbounded if message authentication is used (this section). In all solutions presented here synchrony has been modeled using the (rather strong) assumptions of the pulse model; fault-tolerant implementation of the pulse model is discussed in Section 15.3.

The firing-squad problem. In addition to assuming the pulse model, a second assumption underlying all solutions presented so far is that the pulse in which the broadcast starts is known to all processes (and numbered 1 for convenience). If this is not the case a priori, the problem arises of starting the algorithm simultaneously, after one or more processes (spontaneously) initiate a request for execution of the broadcast algorithm. A request may come from the general (after computing a result that must be announced to all processes) or from lieutenants (realizing that they all need information stored in the general). This problem is studied in the literature as the *firing-squad* problem. In this problem, one or more processes *initiate* (a request), but not necessarily in the same pulse, and processes may *fire*. The requirements are:

(1) **Validity.** No correct process fires unless some process has initiated.
(2) **Simultaneity.** If any correct process fires then all correct processes fire in the same pulse.
(3) **Termination.** If a correct process initiates, then all correct processes fire within finitely many pulses.

Indeed, given a solution for the firing squad problem, the first pulse of the broadcast need not be agreed upon in advance; processes requesting a broadcast initiate the firing squad algorithm, and the broadcast starts in the pulse following the firing. The techniques used in the solutions for the Byzantine-broadcast problem and the firing-squad problem can be combined to obtain more time-efficient protocols that solve the broadcast problem directly in the absence of a priori agreement about the first pulse.

The time complexity and early-stopping protocols. In this chapter we have presented protocols using $t + 1$ or $2t + 3$ pulses, or rounds of communication. It was shown by Fischer and Lynch [FL82] that $t + 1$ rounds of communication is optimal for t-robust consensus protocols, and the result was extended to cover authenticating protocols by Dolev and Strong [DS83].

A subtle point in these proofs is that in the scenarios used a process must fail in each of the pulses 1 through t, so the lower bounds are worst-case w.r.t. the number of actual failures during the execution. Because in most executions the actual number of failures is much lower than the resilience, the existence has been studied of protocols that can reach agreement earlier in those executions that have only a small number of failures. Broadcast protocols with this property are called *early stopping* protocols. Dolev, Reischuk, and Strong [DRS82] demonstrated a lower bound of $f + 2$ rounds for any protocol in an execution with f failures. A discussion of several early stopping broadcast and consensus protocols is found in [BGP92].

The early stopping protocols decide within a few pulses after correct processes conclude that there has been a pulse without new failures. It cannot be guaranteed, however, that all correct processes reach this conclusion in the same pulse. (Unless, of course, they do so in pulse $t + 1$; as at most t processes fail, there is one round among the first $t + 1$ in which no new failure occurs.) As a consequence, early stopping protocols do not satisfy simultaneity. It was shown by Coan and Dwork [CD91] that to achieve simultaneity, $t + 1$ rounds are also necessary in executions where no failures occur, even for randomized protocols and in the (very weak) crash model. This implies that authenticated protocols also need $t + 1$ pulses to agree simultaneously.

Decision problems and interactive consistency. Using a broadcast protocol as a subroutine, virtually all decision problems can be solved for synchronous systems by reaching *interactive consistency*, that is, agreement on the set of inputs. In the interactive consistency problem, processes decide on a *vector* of inputs, with one entry for each process in the system. Formally, the requirements are:

(1) **Termination.** Each correct process p decides on a vector V_p, with one entry for each process.
(2) **Agreement.** The decision vectors of correct processes are equal.
(3) **Dependence.** If q is correct, then for correct p, $V_p[q] = x_q$.

Interactive consistency can be achieved by multiple broadcasts: each process broadcasts its input, and process p places its decision in q's broadcast in

$V_p[q]$. Termination, agreement, and dependence are immediately inherited from the corresponding properties of the broadcast algorithm.

Because every correct process computes the same vector (agreement), most decision problems are easily solved using a deterministic function on the decision vector (which immediately guarantees agreement). Consensus, for example, is solved by extracting the majority value from the decision vector. Election is solved by selecting the smallest unique identity in the vector (beware; the elected process can be faulty).

15.3 Clock Synchronization

It was shown in the previous sections that (when deterministic algorithms are considered) synchronous systems have a higher resilience than asynchronous systems. The demonstration was made for the idealized model of synchrony, where processes operate in pulses. The higher resilience of the pulse model implies that it is not possible deterministically to synchronize fully asynchronous networks in a robust way. It will be shown in this section that a robust implementation of the pulse model is possible in the model of asynchronous bounded-delay networks (ABD networks).

The ABD model is characterized by the availability of local clocks and an upper bound on message delay. In the description and analysis of algorithms we make use of a *real-time frame*, which is an assignment of a time of occurrence $t \in \mathbb{R}$ to every event. According to relativistic physics, there is no standard or preferred way to make this assignment; we assume in the following that a physically meaningful assignment has been chosen. The real-time frame is not observable for processes in the system, but processes can indirectly observe time using their *clocks*, whose values are related to real time. Process p's clock is denoted C_p and can be read and written to by process p (writing to the clocks is necessary for synchronization). The value of the clock changes continuously in time when the clock is not assigned; we write $C_p(t) = T$ to denote that at real time t the clock reads T.

Capitals (C, T) are used for clock times and lower case letters (c, t) for real times. Clocks can be used for controlling the occurrence of events, as in

$$\textbf{when } C_p = T \textbf{ then } \text{send message}$$

which causes the message to be sent at time $C_p^{-1}(T)$. The function C_p^{-1} is denoted c_p.

The value of a perfect clock increases by Δ in Δ time units, i.e., it satisfies $C(t + \Delta) = C(T = t) + \Delta$. Perfect clocks, once synchronized, never need

adjustment again, but unfortunately they are only a (useful) mathematical abstraction. Clocks used in distributed systems suffer a *drift*, bounded by a small known constant ρ (typically of the order of 10^{-5} or 10^{-6}). The drift of clock C is *ρ-bounded* if, for t_1 and t_2 such that no assignment to C occurs between t_1 and t_2,

$$(t_2 - t_1)(1+\rho)^{-1} \leq C(t_2) - C(t_1) \leq (t_2 - t_1)(1+\rho). \tag{15.1}$$

The various clocks in distributed systems do not show the same clock time at any given real time, i.e., $C_p(t) = C_q(t)$ does not hold necessarily. The clocks are *δ-synchronized* at real time t if $|C_p(t) - C_q(t)| \leq \delta$, and they are δ-synchronized at clock time T if $|c_p(T) - c_q(T)| \leq \delta$. We shall regard these notions as equivalent; see Exercise 15.8. The goal of clock-synchronizing algorithms is to achieve and maintain a global δ-synchronization, i.e., δ-synchronization between each pair of clocks. The parameter δ is the *precision* of synchronization.

The message delay is bounded from below by $\delta_{\min}$ and from above by $\delta_{\max}$, where $0 \leq \delta_{\min} < \delta_{\max}$; formally, if a message is sent at real time σ and received at real time τ, then

$$\delta_{\min} \leq \tau - \sigma \leq \delta_{\max}. \tag{15.2}$$

Because the choice of a real-time frame is free, assumptions (15.1) and (15.2) have regard to the time frame as well as to the clocks and the communication system.

15.3.1 Reading a Remote Clock

In this subsection the degree of precision with which process p can adjust its perfect clock to the perfect clock of a reliable server s will be studied. With a deterministic protocol the best obtainable precision is $\frac{1}{2}(\delta_{\max} - \delta_{\min})$, and this precision can be obtained by a simple protocol that exchanges only one message. Probabilistic protocols may achieve an arbitrary precision, but the message complexity depends on the desired precision and the distribution of message-delivery times.

Theorem 15.12 *There exists a deterministic protocol for synchronizing C_p to C_s with precision $\frac{1}{2}(\delta_{\max} - \delta_{\min})$, which exchanges one message. No deterministic protocol achieves a higher precision.*

Proof. We first present the simple protocol and prove that it achieves the precision claimed in the theorem. To synchronize C_p the server sends one

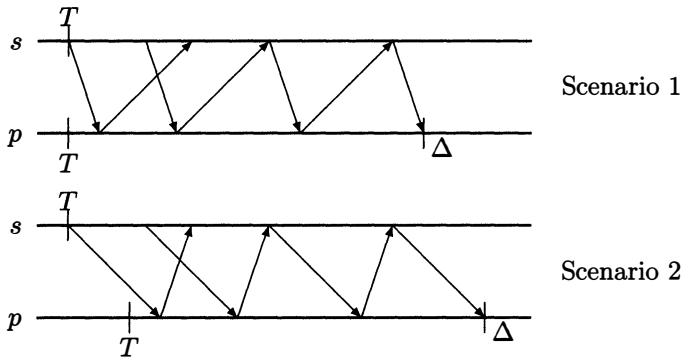

Figure 15.5 SCENARIOS FOR THE DETERMINISTIC PROTOCOL.

message, $\langle \textbf{time}, C_s \rangle$. When p receives $\langle \textbf{time}, T \rangle$ it adjusts its clock to $T + \frac{1}{2}(\delta_{\max} + \delta_{\min})$.

To prove the claimed precision, call the real times of sending and receiving the $\langle \textbf{time}, T \rangle$ message σ and τ, respectively; now $T = C_s(\sigma)$. Because the clocks are perfect, $C_s(\tau) = T + (\tau - \sigma)$. At time τ, p adjusts its clock to read $C_p(\tau) = T + \frac{1}{2}(\delta_{\max} + \delta_{\min})$, so $C_s(\tau) - C_p(\tau) = (\tau - \sigma) - \frac{1}{2}(\delta_{\max} + \delta_{\min})$. Now $\delta_{\min} \leq \tau - \sigma \leq \delta_{\max}$ implies $|C_s(\tau) - C_p(\tau)| \leq \frac{1}{2}(\delta_{\max} - \delta_{\min})$.

To show the lower bound on the precision, let a deterministic protocol be given; in this protocol p and s exchange some messages, after which p adjusts its clock. Two scenarios for the protocol are considered, as depicted in Figure 15.5. In the first scenario, the clocks are equal prior to the execution, all messages from s to p are delivered after $\delta_{\min}$, and all messages from p to s are delivered after $\delta_{\max}$. If the adjustment in this scenario is Δ_1, p's clock is exactly Δ_1 ahead of C_s after the synchronization.

In the second scenario, C_p is $\delta_{\max} - \delta_{\min}$ behind C_s prior to execution, all messages from p to s are delivered after $\delta_{\min}$, and all messages from s to p are delivered after $\delta_{\max}$. Calling the adjustment in this scenario Δ_2, we find that p's clock is exactly $\delta_{\max} - \delta_{\min} - \Delta_2$ behind C_s after the synchronization.

However, neither p nor s observes the difference between the scenarios, because the uncertainty in message delay hides the difference; consequently $\Delta_1 = \Delta_2$. This implies that the worst-case precision is at best

$$\min_{\Delta} \max(|\Delta|,\ |\delta_{\max} - \delta_{\min} - \Delta|).$$

This minimum equals $\frac{1}{2}(\delta_{\max} - \delta_{\min})$ (and occurs for $\Delta = \frac{1}{2}(\delta_{\max} - \delta_{\min})$). □

If two processes p and q both synchronize their clock with the server with this precision, a global $(\delta_{\max} - \delta_{\min})$-synchronization is achieved, which is sufficient for most applications.

A better precision is achievable with the probabilistic synchronization protocol proposed by Cristian [Cri89]. For this protocol it is assumed that the delay of a message is a stochastic variable, distributed according to a (not necessarily known) function $F : [\delta_{\min}, \delta_{\max}] \to [0, 1]$. The probability for any message to arrive *within* x is $F(x)$, and the delay of different messages is independent. An arbitrary precision is only achievable if the lower bound $\delta_{\min}$ is tight, i.e., for all $x > \delta_{\min}$, $F(x) > 0$.

The protocol is simple; p asks s to send the time, and s responds with a $\langle\, \mathbf{time}, T \,\rangle$ message immediately. Process p measures the time I between sending the request and receipt of the response; $I \geq 2\delta_{\min}$ holds. The delay of the response message is at least $\delta_{\min}$ and at most $I - \delta_{\min}$, and hence differs by at most $\frac{1}{2}(I - 2\delta_{\min})$ from $\frac{1}{2}I$. So p can set its clock to $T + \frac{1}{2}I$, and achieves a precision of $\frac{1}{2}(I - 2\delta_{\min})$. Assuming the desired precision is ϵ, p sends a new request if $\frac{1}{2}(I - 2\delta_{\min}) > \epsilon$, and terminates otherwise.

Lemma 15.13 *The probabilistic clock-synchronization protocol achieves precision ϵ with an expected number of messages of at most $F(\delta_{\min} + \epsilon)^{-2}$.*

Proof. The probability that p's request arrives within $\delta_{\min} + \epsilon$ is $F(\delta_{\min} + \epsilon)$ and so is the probability that the response arrives within $\delta_{\min} + \epsilon$. Consequently, the probability that p receives the response within $2\delta_{\min} + 2\epsilon$ is at least $F(\delta_{\min} + \epsilon)^2$, which implies a bound of $F(\delta_{\min} + \epsilon)^{-2}$ on the expected number of trials before a successful message exchange. □

The time complexity of the protocol is reduced if p sends a new request when no response was received $2\delta_{\min} + 2\epsilon$ after sending the request. The expected time is then independent of the expected or maximal delay, namely $2(\delta_{\min} + \epsilon)\, F(\delta_{\min} + \epsilon)^{-2}$, and the protocol is robust against loss of messages. (Sequence numbers must be used to distinguish obsolete responses.)

15.3.2 Distributed Clock Synchronization

This section presents a t-Byzantine-robust (for $t < N/3$) distributed clock-synchronization algorithm due to Mahany and Schneider [MS85]. It was shown by Dolev, Halpern, and Strong [DHS84] that no synchronization is possible if $t \geq N/3$, unless authentication is used.

The core of the synchronization algorithm is a protocol that achieves *inexact agreement* on the average values of the clocks. The processes adjust

```
var x_p, y_p, esti_p   : real ;      (* Input, output, estimator of V *)
    V_p, A_p           : multiset of real ;

begin (* Input collection phase *)
      V_p := ∅ ;
      forall q ∈ ℙ do send ⟨ask⟩ to q ;
      wait 2δ_max ; (* Process ⟨ask⟩ and ⟨val, x⟩ messages *)
      while #V_p < N do insert(V_p, ∞) ;
      (* Now compute acceptable values *)
      A_p := {x ∈ V_p : #{y ∈ V_p : |y − x| ≤ δ} ≥ N − t} ;
      esti_p := estimator(A_p) ;
      while #A_p < N do insert(A_p, esti_p) ;
      y_p := (Σ A_p)/N
end

Upon receipt of ⟨ask⟩ from q:
      send ⟨val, x_p⟩ to q

Upon receipt of ⟨val, x⟩ from q:
      if no such message was received from q before
        then insert(V_p, x)
```

Algorithm 15.6 THE FAST-CONVERGENCE ALGORITHM.

their clocks and achieve a high degree of synchronization. Due to drift, the precision degrades after a while, necessitating a new synchronization round after a certain interval. Assume that at real time t_0 the clocks are δ_0-synchronized; then until time $t_0 + R$, the clocks are $(\delta_0 + 2\rho R)$-synchronized. Thus, if the desired precision is δ and a synchronization round achieves precision δ_0, the rounds are repeated every $(\delta - \delta_0)/2\rho$ time units. As the time, say S, to execute a synchronization round is usually very small compared to R, the simplifying assumption is justified that during the synchronization the drift is negligible, i.e., the clocks are perfect.

Inexact agreement: the fast-convergence algorithm. In the problem of inexact agreement, used by Mahany and Schneider [MS85] to synchronize clocks, process p has a real input value x_p, where for correct p and q, $|x_p - x_q| \leq \delta$. The output of process p is a real value y_p, and the precision of the output is defined as $\max_{p,q} |y_p - y_q|$; the aim of the algorithm is to achieve a very small value of the precision.

The fast-convergence algorithm proposed by Mahany and Schneider is given as Algorithm 15.6. For a finite set $A \subset \mathbb{R}$, define two functions $intvl(A) = [\min(A), \max(A)]$ and $width(A) = \max(A) - \min(A)$. The algo-

rithm has an input collection phase and a computation phase. In the first phase process p requests every other process to send its input (by shouting an $\langle$ **ask** $\rangle$ message) and waits for $2\,\delta_{\max}$ time units. After that time p has received all the inputs from the correct processes, as well as answers from a subset of faulty processes. These answers are padded with (meaningless) values ∞ for processes that did not reply.

The process then applies a filter to the received values, which is guaranteed to pass all values of correct processes and only those faulty values that are sufficiently close to correct values. As correct values differ only by δ and there are at least $N - t$ correct values, each correct value has at least $N - t$ values that differ at most δ from it; A_p stores the received values with this property.

The output is then computed by averaging over the values, where all rejected values are replaced by an estimate computed by applying a deterministic function *estimator* to the surviving values. This function satisfies $estimator(A) \in intvl(A)$, but is otherwise arbitrary; it could be the minimum, maximum, average, or $\frac{1}{2}[\max(A) + \min(A)]$.

Theorem 15.14 *The fast-convergence algorithm achieves precision* $2t\delta/N$.

Proof. Let v_{pr} be the value included in V_p for process r when p times-out (i.e., v_{pr} is either x_r or ∞), and a_{pr} the value in A_p for process r when p computes y_p (i.e., a_{pr} is either v_{pr} or $esti_p$). The precision will be bounded by splitting the summation in the computation of the decision into summations over correct processes (C) and incorrect processes (B). For correct p and q, the difference $|a_{pr} - a_{qr}|$ is bounded by 0 if $r \in C$ and by 2δ if $r \in B$.

The first bound follows because if p and r are correct processes, $a_{pr} = x_r$. Indeed, as r replies to p's $\langle$ **ask** $\rangle$ message promptly, $v_{pr} = x_r$. Similarly, $v_{pr'} = x_{r'}$ for all correct r', and the assumption on the input implies that r's value survives the filtering by p, hence $a_{pr} = v_{pr}$.

The second bound holds because for correct p and q, $width(A_p \cup A_q) \leq 2\delta$ when p and q compute their decisions. Because the added estimates lie between accepted values, it suffices to consider the maximal difference between values a_p and a_q that passed the filters of p and q, respectively. There are at least $N - t$ processes r for which $|v_{pr} - a_p| \leq \delta$, and at least $N - t$ processes r for which $|v_{qr} - a_p| \leq \delta$. This implies that there is a *correct* r such that both $|v_{pr} - a_p| \leq \delta$ and $|v_{qr} - a_p| \leq \delta$; but as r is correct, $v_{pr} = v_{qr}$, hence $|a_p - a_q| \leq 2\delta$.

It now follows that, for correct p and q,

$$
\begin{aligned}
|y_p - y_q| &= |(\textstyle\sum A_p)/N \;-\; (\sum A_q)/N| \\
&= \frac{1}{N} \cdot \left| \left(\sum_{r \in C} a_{pr} + \sum_{r \in B} a_{pr} \right) - \left(\sum_{r \in C} a_{qr} + \sum_{r \in B} a_{qr} \right) \right| \\
&= \frac{1}{N} \cdot \left| \left(\sum_{r \in C} a_{pr} - \sum_{r \in C} a_{qr} \right) + \left(\sum_{r \in B} a_{pr} - \sum_{r \in B} a_{qr} \right) \right| \\
&\leq \frac{1}{N} \cdot \left[\left(\sum_{r \in C} |a_{pr} - a_{qr}| \right) + \left(\sum_{r \in B} |a_{pr} - a_{qr}| \right) \right] \\
&\leq \frac{1}{N} \cdot \left[(0) + \left(\sum_{r \in B} 2\delta \right) \right] \;\leq\; 2t\delta/N.
\end{aligned}
$$

□

An arbitrary precision can be achieved by repeating the algorithm; when iterated i times, the precision becomes $(\frac{2}{3})^i\delta$. The precision is even better if a smaller fraction (than one-third) of the processes is faulty; in the derivation of the precision, t can be understood as the actual number of faulty processes. The (worst-case) output precision of the algorithm cannot be improved by a suitable choice of the function *estimator*; indeed, a Byzantine process r can enforce to p any value $a_{pr} \in \mathit{intvl}(A_p)$, by simply sending this value to p. The function can be chosen suitably so as to achieve a good average precision when something is known about the most likely behavior of faulty processes.

Clock synchronization. To synchronize clocks, the fast-convergence algorithm is used to reach inexact agreement on the new value of the clocks. It is assumed that the clocks are δ-synchronized initially. The algorithm must be adapted because

(1) the message delay is not known exactly so a process cannot know the exact value of another process; and
(2) time elapses during execution of the algorithm, so the clocks do not have constant values but increase in time.

To compensate for the unknown delay a process adds $\frac{1}{2}(\delta_{\max} + \delta_{\min})$ to the received clock values (as in the deterministic protocol of Theorem 15.12),

var C_p, Δ_p, $esti_p$: real ; (* Clock, adaptation, estimator of V *)
D_p, A_p : multiset of real ;

begin (* Input collection phase *)
$D_p := \varnothing$;
forall $q \in \mathbb{P}$ **do** send $\langle$ **ask** $\rangle$ to q ;
wait $2\delta_{\max}$; (* Process $\langle$ **ask** $\rangle$ and $\langle$ **val**, x $\rangle$ messages *)
while $\#D_p < N$ **do** $insert(D_p, \infty)$;
(* Now compute acceptable values *)
$A_p := \{x \in D_p : \#\{y \in D_p : |y - x| \leq \delta + (\delta_{\max} - \delta_{\min})\} \geq N - t\}$;
$esti_p := estimator(A_p)$;
while $\#A_p < N$ **do** $insert(A_p, esti_p)$;
$\Delta_p := (\sum A_p)/N$;
$C_p := C_p + \Delta_p$
end

Upon receipt of $\langle$ **ask** $\rangle$ from q:
send $\langle$ **val**, C_p $\rangle$ to q

Upon receipt of $\langle$ **val**, C $\rangle$ from q:
if no such message was received from q before
then $insert(D_p, (C + \frac{1}{2}(\delta_{\max} + \delta_{\min})) - C_p)$

Algorithm 15.7 FAST CONVERGENCE OF CLOCKS.

introducing an additional $\delta_{\max} - \delta_{\min}$ term in the output precision. To represent the received value as a clock value rather than as a constant, p stores the difference of the received clock value (plus $\frac{1}{2}(\delta_{\max} + \delta_{\min})$) and its own as Δ_{pr}. At time t, p's approximation of r's clock is $C_p(t) + \Delta_{pr}$. The modified algorithm is given as Algorithm 15.7.

Observe that in Algorithm 15.7 the filter has a wider margin, namely $\delta + (\delta_{\max} - \delta_{\min})$, than in Algorithm 15.6, where the margin is δ. The wider margin compensates for the unknown message delay and the threshold is motivated by the following proposition. Let d_{pr} denote the value that p has inserted in D_p for process r after the first phase of p (compare with the value v_{pr} in the previous algorithm).

Proposition 15.15 *For correct p, q, and r, after p's time-out $|d_{pr} - d_{qr}| \leq \delta + (\delta_{\max} - \delta_{\min})$ holds.*

Proof. The exchange of the $\langle$ **val**, C $\rangle$ message from q to p implements the deterministic clock reading algorithm from Theorem 15.12. When p receives this message, $|C_q - [C + \frac{1}{2}(\delta_{\max} + \delta_{\min})]|$ is bounded by $\frac{1}{2}(\delta_{\max} - \delta_{\min})$, so d_{pq} differs by at most $\frac{1}{2}(\delta_{\max} + \delta_{\min})$ from $C_q - C_p$. Similarly, d_{pr} differs by

at most $\frac{1}{2}(\delta_{\max} + \delta_{\min})$ from $C_r - C_p$. As C_q and C_r differ by at most δ, the result follows. □

Theorem 15.16 *After execution of Algorithm 15.7 the clocks are synchronized with precision*

$$(\delta_{\max} - \delta_{\min}) \quad + \quad \frac{2t}{N}[\delta + (\delta_{\max} - \delta_{\min})].$$

Proof. In this proof, write C_p for the unadjusted clock, and C'_p for the adjusted clock, i.e., $C'_p(t) = C_p(t) + \Delta_p$. To bound the precision of the adjusted clocks, fix a real time t that is later than the time-out by all correct processes, and let $w_{pr} = C_p + d_{pr}$. From the proof of the proposition it is seen that (for correct p, q, and r) $|w_{pr} - C_r(t)| \leq \frac{1}{2}(\delta_{\max} - \delta_{\min})$, which implies $|w_{pr} - w_{qr}| \leq (\delta_{\max} - \delta_{\min})$. For incorrect r, the difference $|w_{pr} - w_{qr}|$ is bounded by $2\delta + 3(\delta_{\max} - \delta_{\min})$, which is proved similarly to the corresponding step in the proof of Theorem 15.14.

Finally, as $C'_p(t) = C_p(t) + \Delta_p = (\sum_r w_{Pr})/N$, the precision is derived as in the proof of Theorem 15.14 by splitting the average into averages over correct processes and over incorrect processes. □

It is implicit in the description of the algorithm that all $\langle\, \mathbf{val}, C \,\rangle$ messages are sent with the *unadjusted* clock values; this can be achieved by deferring the clock adjustment until a reply has been sent to all $\langle\, \mathbf{ask} \,\rangle$ messages.

15.3.3 Implementing the Round-model

The pulse model of synchronous computations can be simulated in systems with weaker synchrony assumptions, provided that

(1) there is an upper bound $\delta_{\max}$ on the message delay;
(2) there is an upper bound γ on the time necessary for the local computation in a pulse (the time for state change plus the time for sending messages); and
(3) the processes have clocks that are δ-synchronized at every clock time T and have ρ-bounded drift.

The simulation takes $(1 + \rho)(\delta_{\max} + \delta + \gamma)$ clock time per pulse.

The simulation algorithm is very elementary; assume that execution of the simulated algorithm is supposed to start at clock time 0 (starting at a later time can be implemented by adding a fixed term T_0 to all clock times). When a process's clock reads 0, it sends its messages of pulse 1. When a process's clock reads $i(1 + \rho)(\delta_{\max} + \delta + \gamma)$, the process executes the state

change of pulse i and subsequently sends the messages for pulse $(i+1)$. As the clock increases to arbitrarily high values, each correct process executes infinitely many pulses with this strategy.

It remains to show that correct process p receives all messages sent to it in pulse i (by correct processes) before it executes the state change for that pulse. Assume process q sends a message to p in pulse i. Process q started executing the state change of the previous pulse and sending messages of pulse i when its clock read $(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)$, that is, at real time $c_q[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)]$. The assumption regarding the bound on local processing time implies that the pulse-i message is sent, at the latest, at real time

$$c_q[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)]+\gamma\ ,$$

which implies (by the bound on message delay) that it is received at the latest at real time

$$c_q[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)]+\gamma+\delta_{\max}\ .$$

Process p starts executing the state change at local clock time $i(1+\rho)(\delta_{\max}+\delta+\gamma)$, which means at real time

$$c_p[i(1+\rho)(\delta_{\max}+\delta+\gamma)].$$

As the clocks of p and q are δ-synchronized at clock time $(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)$,

$$c_q[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)] \leq c_p[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)].$$

The ρ-bounded drift of p's clock implies that

$$c_p[i(1+\rho)(\delta_{\max}+\delta+\gamma)] \geq c_p[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)]+(\delta_{\max}+\delta+\gamma)\ .$$

Combining these equations,

$$c_p[i(1+\rho)(\delta_{\max}+\delta+\gamma)] \geq c_q[(i-1)(1+\rho)(\delta_{\max}+\delta+\gamma)]+\gamma+\delta_{\max}\ ,$$

which implies that p receives the message before executing the state change.

Exercises to Chapter 15

Section 15.1

Exercise 15.1 *How many messages are exchanged in the Broadcast(N, t) protocol (Algorithm 15.2 and 15.3)?*

Exercise 15.2 *What is the highest number of messages sent by correct processes in Algorithm 15.4 in executions that decide on* 0*? Answer both for the case where the general is correct and the case where the general is faulty.*

Section 15.2

Exercise 15.3 *How many messages are exchanged by the broadcast algorithm of Lamport, Shostak, and Pease, described in Subsection 15.2.1?*

Exercise 15.4 *Give an execution of the protocol of Dolev and Strong, described in Subsection 15.2.1, where correct processes* p *and* q *end with* $W_p \neq W_q$.

Exercise 15.5 *Show that order-preserving renaming in the range* 1 *through* N *can be solved when interactive consistency is achieved.*

Exercise 15.6 *Assume that* p *signs two messages* M_1 *and* M_2 *with the ElGamal signature scheme (Subsection 15.2.3), using the same value of* a. *Show how to find* P*'s secret key from the two signed messages.*

Exercise 15.7 *Let* n *be the product of two large prime numbers, and assume that a black box is given that computes square roots. That is, given a quadratic residue* y*, the box outputs an* x *with* $x^2 = y$ *(equation is modulo* n*). Show how the box can be used to factor* n.

Section 15.3

Exercise 15.8 *Show that if clocks* C_p *and* C_q *have* ρ*-bounded drift and are* δ*-synchronized at real time* t*, then they are* $\delta(1+\rho)$*-synchronized at clock time* $T = C_p(t)$.

Exercise 15.9 *The probabilistic clock-synchronization protocol is efficient if many messages have a delay close to* $\delta_{\min}$ *(that is,* $F(\delta_{\min} + \epsilon)$ *moves away from* 0 *quickly even for small* ϵ*).*
Give a "dual" protocol that works efficiently if many messages have a delay close to $\delta_{\max}$ *(that is, if* $F(\delta_{\max} - \epsilon)$ *stays sufficiently far from* 1 *even for small* ϵ*). Show that the expected message complexity is bounded by* $2.[1 - F(\delta_{\max} - \epsilon)]^{-2}$ *and prove a bound on the expected running time.*

Exercise 15.10 *Does the set A_p in Algorithm 15.6 satisfy $width(A_p) \leq \delta$ for all correct p?*

16

Failure Detection

The impossibility of solving consensus in asynchronous systems (Section 14.1) has led to weaker problem formulations and stronger models. Examples of the former include weaker coordinations and randomization, and examples of the latter include introducing synchrony. Failure detectors are now widely recognized as an alternative way to strengthen the computation model.

Studying synchronous models is practically motivated because most distributed programming environments do provide clocks and timers in some way. Theoretical studies reveal for what tasks the use of these primitives is necessary and to what degree they must be used. With failure detectors, the situation is similar: quite often the run-time support system will return error messages upon an attempt to communicate with a crashed process. However, these error messages are not always absolutely reliable. It is therefore useful to study how reliable they must be to allow a solution for the consensus problem (or other problems).

In contrast to synchrony (implemented using physical clocks), failure detectors have no straightforward intuitive implementation. This implies that non-trivial solutions must be found in order to implement them in the run-time system or as a module in the application. The implementations more often than not rely on the use of timers (see Section 16.4) and this has led to some serious critiques of the failure detector approach. It was argued that "failure detectors" as such do not solve any problems, because their implementation requires the same resources (time) that can be used to solve consensus directly.

There certainly is a point here, but the same argument applies to virtually everything that was invented in computer science over the past half century. Why use a higher order programming language if the program must still be compiled to machine language in order to be executed? We could "just

as well" write the whole program in machine language directly! Why use device drivers to access disks and printers from an application if these drivers are just programs themselves as well? We could "just as well" include the driver code in our application and do without the driver. Of course we know all the arguments for using higher order languages and drivers: they make programming easier, improve understandability and portability, and allow studies of the power of certain concepts. And these arguments apply to failure detectors as well.

16.1 Model and Definitions

Failure detection, being available in many practical systems since a long time ago, was first formulated as an abstract mechanism around 1991 by the research goup of Sam Toueg of Cornell University. It took about five years for a foundational paper to appear [CT96], from which most of our definitions and results are taken.

A failure detector is a module that provides to each process a collection of *suspected* processes; the test $j \in D$ in the program returns if process j is suspected at the moment of evaluation. The type of failure considered in this chapter is crashes. The modules in different processes need not agree (that is, correct process p may suspect r while correct q doesn't) and need not be just at any time (correct p may suspect correct r or not suspect failed r). Of course, in order to be able to reason about programs with failure detectors we must first express the properties of the detectors, most notably the relation between the detector output and actual failures.

The actual failures are expressed in a *failure pattern* which we shall first define. To circumvent the use of expressions in temporal logic we use an explicit time variable t in our expressions; this t ranges over the set T of all time instances, which could be the natural numbers for example. Please note that processes can *not* observe the time t, the failure pattern F, the sets $Crash(F)$ and $Corr(F)$, or the failure detector history! Process q *only* "sees" the value $H(q, t)$ when it accesses its failure detector at time t.

16.1.1 Four Basic Detector Classes

The crashes occurring during an execution are modeled by a *failure pattern*, which is a function $F : T \to \mathcal{P}(\mathbb{P})$. Here $F(t)$ is the collection of processes that have crashed at time t, and because we do not assume restarts, $t_1 \leq t_2$ implies $F(t_1) \subseteq F(t_2)$. The collections of defective and correct processes are defined by $Crash(F) = \cup_{t \in T} F(t)$ and $Corr(F) = \mathbb{P} \setminus Crash(F)$.

Suspicions may differ from time to time and from process to process, and are therefore modeled by a function $H : \mathbb{P} \times T \to \mathcal{P}(\mathbb{P})$. Here $H(q, t)$ is the collection of processes suspected by q at time t.

To allow failure detectors to be *non-deterministic* (i.e., with a given failure pattern, different responses are possible), we model a detector $\mathcal{D}$ as a mapping from failure patterns to *collections of* failure detector histories. That is, for a failure pattern F, an element $H \in \mathcal{D}(F)$ is a failure detector history.

Properties of histories. Understandably, in order for a failure detector to be useful, there must be a relation between its output (the failure detector histories) and its input (the failure pattern). The requirements are always of two types, namely completeness (the detector will suspect crashed processes) and accuracy (the detector will not suspect correct processes). Completeness bounds the set of suspected processes from below and accuracy from above, but in neither case is an exact match with the set of crashed processes required (such a requirement could never be met by any practically imaginable system).

We consider completeness in just one form.

Definition 16.1 *Failure detector $\mathcal{D}$ is* complete *if every crashed process will eventually be suspected by every correct process:*

$$\forall F : \forall H \in \mathcal{D}(F) : \exists t : \forall p \in \mathit{Crash}(F) : \forall q \in \mathit{Corr}(F) : \forall t' \geq t : p \in H(q, t').$$

(This property is also called strong completeness in the literature.)

We consider four different forms of accuracy. Remember that accuracy dictates that correct processes are unsuspected; in the strong form this holds for *all*, in the weak form this holds for *at least one* correct process. We further distinguish "eventual" versions, where the property need not hold initially but is required to hold from some time on.

Definition 16.2 *Failure detector $\mathcal{D}$ is* strongly accurate *if no process is ever suspected if it has not crashed:*

$$\forall F : \forall H \in \mathcal{D}(F) : \forall t : \forall p, q \notin F(t) : p \notin H(q, t).$$

Failure detector $\mathcal{D}$ is weakly accurate *if there exists a correct process that is never suspected:*

$$\forall F : \forall H \in \mathcal{D}(F) : \exists p \in \mathit{Corr}(F) : \forall t : \forall q \notin F(t) : p \notin H(q, t).$$

Failure detector $\mathcal{D}$ is eventually strongly accurate *if there exists a time after*

which no correct process is suspected:

$$\forall F : \forall H \in \mathcal{D}(F) : \exists t : \forall t' \geq t : \forall p, q \in Corr(F) : p \notin H(q, t').$$

Failure detector $\mathcal{D}$ is eventually weakly accurate *if there exist a time and a correct process that is not suspected after that time:*

$$\forall F : \forall H \in \mathcal{D}(F) : \exists t : \exists p \in Corr(F) : \forall t' \geq t : \forall q \in Corr(F) : p \notin H(q, t').$$

Observe that the definition only restricts the suspicions by *non-crashed* processes q; the suspicions by crashed processes are irrelevant. Further, the definition of strong accuracy explicitly rules out that a crashing process is suspected *before* it is going to crash. For the other three forms of accuracy it makes no difference to exclude this.

It is very easy to have a failure detector that is complete (a detector that suspects everybody: $H(q, t) = \mathbb{P}$) and also to have a failure etector that is accurate (a detector that suspects nobody: $H(q, t) = \varnothing$). Interesting and useful detectors always combine a completeness and an accuracy property, and four classes are considered.

Definition 16.3 *A failure detector is* perfect *if it is complete and strongly accurate; the class of perfect detectors is denoted by $\mathcal{P}$.*
A failure detector is strong *if it is complete and weakly accurate; the class of strong detectors is denoted by $\mathcal{S}$.*
A failure detector is eventually perfect *if it is complete and eventually strongly accurate; the class of eventually perfect detectors is denoted by $\diamond\mathcal{P}$.*
A failure detector is eventually strong *if it is complete and eventually weakly accurate; the class of eventually strong detectors is denoted by $\diamond\mathcal{S}$.*

16.1.2 Use of Failure Detectors and Pitfalls

It isn't particularly difficult to imagine how failure detectors can be used in designing distributed applications.

Recall the communication operations that we encountered frequently in asynchronous distributed algorithms (Chapter 14):

(1) Each node performs a *shout*, that is, sends a message to each node.
(2) Each node collects $N - t$ of the shouted messages.

A process should never wait for the arrival of more than $N - t$ messages because this risks a deadlock in case of crashes. The construction causes no eternal blocking (see, e.g., Lemma 14.30) because at least $N - t$ processes are alive. The pitfall is that even without any crashes, the set of collected

messages may differ from process to process. The set of messages collected always has the same size $(N - t)$. Waiting for a message to arrive from any *specific* process is absolutely forbidden without detectors, because it may lead to blocking if the desired sender has crashed.

The standard way of communication using failure detectors runs like this:

(1) Each node sends a message to each node.
(2) Each node waits until for each process q, a message from q has arrived or q is suspected.

The completeness of the failure detector makes this construction free of eternal blocking; indeed, each process that doesn't send the message because of having crashed will eventually be suspected. As in the case of the previous construction, there is a pitfall: there are various scenarios leading to different sets of messages collected by different correct processes.

(1) Process q crashes, but prior to its crash sends some or all of the required messages. Process p_1 receives the message before suspecting q, process p_2 doesn't, and ends its receive phase without q's message.
(2) Process q is correct and sends the message, but p_1 suspects q and doesn't collect q's message, while p_2 does not suspect q and collects the message.

Thus, a set of collected messages may include a message from a crashed process, as well as miss one from a correct processes. The size of the set may differ and, even if a bound t on the number of crashes exists, the set can be smaller than $N - t$ messages because of erroneous suspicions. Only a bound on the number of processes that can be *suspected* may bound the size of the set from below; basically this is done in the weakly accurate detectors.

With a failure detector it is possible to receive a message from a specifically mentioned process: waiting for the message will be interrupted when suspicion against the process arises. Again observe that in this way a message from a correct process to a correct process may fail to be received.

The mode of receiving messages sketched is so common that we introduce a programming shorthand for it: the statement "collect $\langle$ **message**, *par* $\rangle$ from q". The instruction waits until either a message $\langle$ **message**, *par* $\rangle$ is received from q, or q is supected; a boolean value indicating the succes of reception is returned.

```
x_i := input ;
for r := 1 to N
    do begin if i = r
                then forall j do send ⟨ value, x_i, r ⟩ to j ;
             if collect ⟨ value, x', r ⟩ from p_r
                then x_i := x'
       end ;
decide x_i
```

Algorithm 16.1 A ROTATING COORDINATOR ALGORITHM USING $\mathcal{S}$ (FOR p_i).

16.2 Solving Consensus with a Weakly Accurate Detector

This section presents a relatively simple consensus algorithm using failure detectors. The correctness specifications are:

(1) **Termination.** Every correct process decides once.
(2) **Agreement.** All decisions (in one execution) are equal.
(3) **Validity.** A decision equals the input of at least one process.

It is easy to see that validity implies non-triviality, and it is thus seen that we obtain a result that is impossible without failure detection (Theorem 14.8).

Algorithm 16.1 solves the problem using a weakly accurate failure detector; there exists a process that is *never* suspected, but of course the processes do not know which one! Only when the processes collect a message from this unknown process, are they all guaranteed to receive the message, hence to receive *the same* information. The algorithm avoids the difficulty of not knowing which node is unsuspected by giving a turn to each process; such an approach is called the *rotating coordinator* paradigm. The algorithm consists of N rounds, each process coordinates one of them.

We call the processes p_1 through p_N and p_j coordinates round j. In a round the coordinator shouts its value, all nodes collect a message from the coordinator, and if a node succeeds it replaces its value by the value received from the coordinator. The correctness proof is just outlined here because the details are fairly obvious.

Theorem 16.4 *Algorithm 16.1 solves consensus.*

Proof. The algorithm satisfies termination because no correct process blocks forever in a round; indeed, the coordinator is correct and sends the message, or is eventually suspected (because of the completeness of the detector).

Formally one proves with induction on i that every correct process reaches round i.

Validity is satisfied because a process can only keep its value or replace it by a value received from the coordinator, which allows us to prove (again by induction) that each process enters each round with a value that was the input of some process.

To show agreement, let p_j be the process that is never suspected. In round j every process (that completes the round) receives the value from p_j, hence all processes complete round j with *the same* value. A round that is entered with one value is also completed with one value (the same), which implies that the entire algorithm is completed with just one value. □

Observe that the resiliency t does *not* occur in the program. Actually the resiliency is $N-1$. It is currently not known how the number of rounds can be reduced if the resiliency is limited. The reason is that correct coordinators can be suspected; in this algorithm it doesn't help if processes are correct, what one needs is that they are unsuspected. The number of rounds can be reduced if the accuracy of the detector is strengthened; see Exercise 16.5.

16.3 Eventually Weakly Accurate Detectors

The weak accuracy assumed in the previous section implies that there is a round with unsuspected coordinator *among the first N rounds* of the algorithm. Thus the algorithm can be designed to terminate after N rounds. However, *eventual* weak accuracy, though it implies that a round with unsuspected coordinator will occur *at some time*, does not provide a bound on which round this will be.

Therefore, in addition to actually reaching agreement, the solution must also *detect* when agreement has been reached. This detection requires that, in addition to a failure detector, there is a bound on the resiliency of $t < N/2$; this will be shown in Subsection 16.3.1. The algorithm we present in Subsection 16.3.2 does not provide the optimal resiliency: a bound of $t < N/3$ is assumed.

16.3.1 Upper Bound on Resiliency

Failure detectors with *eventual* accuracy properties may exhibit arbitrary behavior during an arbitrarily (but finitely) long initial period of time. Put bluntly, this means that they are useless to prevent *finite* erroneous executions. In terms of impossibility proofs: results that can be obtained by constructing *finite* erroneous executions carry over to the model with eventually

accurate failure detectors. This is the case, for example, for Theorem 14.16, and we extend the theorem here to models with failure detection. The theorem is stated for an eventually perfect detector; the result also holds for an eventually strong one, because this class of detectors satisfies weaker specifications.

Theorem 16.5 *There exists no consensus algorithm using $\diamond\mathcal{P}$ that allows $t \geq N/2$ crashes.*

Proof. Assume, to the contrary, that such an algorithm exists. In $\mathbb{P}$ two disjoint sets of processes S and T can be formed, both of size $N - t$. We consider two scenarios.

- In scenario 0, the processes in S all have input 0, those outside S fail immediately, and this is detected right from the start by all processes in S. Because S contains $N-t$ processes, the processes in S decide in finite time, and because all alive processes have input 0, their decision is 0.
- In scenario 1, the processes in T all have input 1, those outside T fail immediately, and this is detected right from the start by all processes in T. Because T contains $N-t$ processes, the processes in T decide in finite time, and because all alive processes have input 1, their decision is 1.

Observe that in both scenarios the failure detector behavior is in accordance with the requirements for $\diamond\mathcal{P}$. We don't expand on why the decision for 0 (or 1) is inevitable in scenario 0 (or 1, respectively).

Because S and T are disjoint, the scenarios can be combined into a new scenario, let us say scenario 2. Assume *only the processes outside $S \cup T$ fail*, the processes in S all have input 0, the processes in T all have input 1, and the failure of processes outside $S \cup T$ is detected immediately by those in S and T. Moreover, the processes in S erroneously suspect those in T right from the start, and those in T erroneously suspect those in S right from the start; the requirement of *eventual* perfectness allows this situation to last for any finite amount of time. The messages sent from S to T and vice versa are delivered very slowly.

Now scenario 2 is like scenario 0 for the processes in S and is like scenario 1 for the processes in T, hence these processes start to execute the same series of steps. Within *finite time* these processes decide on the same value as in the earlier scenario, that is, processes in S and T decide on 0 and 1. After the taking of at least one decision in each group, erroneous suspicions stop and messages from S to T and vice versa arrive.

We have thus constructed a scenario in which fewer than t processes fail, the failure detector behaves according to the definition of eventual perfection, and all messages from correct to correct processes arrive. Yet different processes take different decisions, which shows that Agreement isn't satisfied. □

The proof that consensus is not possible at all (Theorem 14.8) constructs an *infinite* non-deciding execution and we shall see in the next section that this infinite behavior can be avoided with eventually correct detectors.

16.3.2 The Consensus Algorithm

Algorithm 16.1 assumed the failure detector to be weakly accurate from the start, which implies that there is an unsuspected coordinator, and hence agreement on values, within the first N rounds. In Algorithm 16.2 the coordinator must extend its activities to also find out if agreement is already occurring. It first collects the current values of all processes and checks if they are uniform (see phase 2). Now here is a pitfall because due to erroneous suspicions the coordinator could collect effectively *less than half* of the active values and then announce a value as the unanimous outcome while it is actually supported by a minority of the processes. Here is where we use the bound on the number of failed processes: in the collection phase the coordinator does not use its detector, but awaits $N-t$ votes as in the classical approaches. The robustness t of Algorithm 16.2 is bounded by $t < N/3$. Now an observed agreement by the coordinator at least implies that a strict majority of the processes has the computed value; not necessarily all correct processes do, because some conflicting votes may be missed because the coordinator only awaits a fixed number of votes.

A decision is allowed if the coordinator has proclaimed that it chose the value v from unanimous votes, as indicated by the bit d in the message $\langle \textbf{outcome}, d, v, r \rangle$ that announces the result of round r. The processes continue their activity after a decision to help other processes decide too.

Lemma 16.6 *For each round r and correct process p, p finishes round r.*

Proof. Assume all correct processes start a round r. A correct coordinator will complete phase 1 because sufficiently many votes are sent. A correct process will complete phase 3 because the coordinator will send the message if it is correct, and eventually be suspected if it isn't. Thus, all correct processes complete the entire round. Induction completes the proof. □

```
x_i := input ;
r := 0 ;
while true do
    begin (* Start new round and compute coordinator*)
        r := r + 1 ; c := (r mod N) + 1 ;
        (* Phase 1: all processes send value to coordinator *)
        send ⟨ value, x_i, r ⟩ to p_c ;
        (* Phase 2: coordinator evaluates outcome *)
        if i = c then
          begin wait until N − t mesgs. ⟨ value, v_j, r ⟩ have been received;
                v := majority of received values;
                d := (∀j : v_j = v) ;      (* range over received messages *)
                forall j do send ⟨ outcome, d, v, r ⟩ to p_j
          end ;
        (* Phase 3: evaluate the round *)
        if collect ⟨ outcome, d, v, r ⟩ from p_c then
          begin x_i := v ;
                if (d ∧ (y_i = @))
                  then decide(v)
          end
    end
```

Algorithm 16.2 A ROTATING COORDINATOR ALGORITHM WITH $\diamond\mathcal{S}$.

We now proceed to show that a majority, sufficiently large to cause a decision, is persistent.

Lemma 16.7 *If at the beginning of a round $k \geq N - t$ processes have value v, then at least k processes have v at the end of that round.*

Proof. A process can suspect the coordinator and keep the same value; the only way to change value is to receive a message with that value from the coordinator. The coordinator took the majority of $N - t$ values received; at most t processes have a value different from v, so at least $N - 2t$ votes for v were received, and $N - 2t > t$. This shows that the coordinator, if it sent a value in phase 2, sent the value v and thus the number of processes with value v can only grow, not decrease. □

Lemma 16.8 *Each correct process decides.*

Proof. Eventual accuracy implies that eventually there is a round r whose coordinator c is not suspected in round r or later. In that round all correct processes receive from the coordinator and agree on the same value, and continue to do so (Lemma 16.7). Consequently, all messages sent by

var D_p set of processes ;
$r_p[\mathbb{P}]$ A timer for each process

Initialization of the detector:
$D_p := \varnothing$;
forall q **do** $r_p[q] := \sigma + \mu$

The sending process, at each multiple of σ:
forall q **do** send $\langle$ **alive** $\rangle$ to q

The receiving process, upon receipt of $\langle$ **alive** $\rangle$ from q:
$r_p[q] := \sigma + \mu$

Failure detection, when $r_p[q]$ elapses:
$D_p := D_p \cup \{q\}$

Algorithm 16.3 PERFECT FAILURE DETECTOR.

coordinators after round r announce that v was unanimous ($d = true$), so a process will decide v as soon as it collects a message in phase 3. But c is never suspected any more, and is coordinator again in round $r + N$, so at the end of this round all processes have decided. □

Theorem 16.9 *Algorithm 16.2 is a consensus algorithm with* $\diamond\mathcal{S}$.

Proof. It is fairly trivial (and left to the reader) to show by induction on r that if any process holds a value in round r, this value was among the inputs; this shows validity. Lemma 16.7 implies agreement, because from a deciding round there is never sufficient support to decide a different value. Finally termination is expressed in the preceding lemma. □

16.4 Implementation of Failure Detectors

A definite advantage of the failure detector approach is that one has to deal only with the *properties* of the detectors, and not with their implementation. However, we do give some possible implementations to show what happens behind the curtains and as an illustration to the definitions. In most cases the detector is a wrapping with asynchronous interface around the careful use of timers.

var D_p set of processes ;
$r_p[\mathbb{P}]$ A timer for each process ;
me_p μ estimate, initially 1

Initialization of the detector:
$D_p := \varnothing$;
forall q **do** $r_p[q] := \sigma + me_p$

The sending process, at each multiple of σ:
forall q **do** send $\langle$ **alive** $\rangle$ to q

The receiving process, upon receipt of $\langle$ **alive** $\rangle$ from q:
if $j \in D_p$ **then**
begin $D_p := D_p \setminus \{q\}$;
$me_p := me_p + 1$
end ;
$r_p[q] := \sigma + me_p$

Failure detection, when $r[q]$ elapses:
$D_p := D_p \cup \{q\}$

Algorithm 16.4 EVENTUALLY PERFECT FAILURE DETECTOR.

16.4.1 Synchronous Systems: Perfect Detection

Algorithm 16.3 for perfect failure detection assumes an upper bound of μ on the communication delay. Alive processes send $\langle$ **alive** $\rangle$ messages at σ time intervals, and not receiving a message from a process for $\sigma + \mu$ time implies the process has crashed.

16.4.2 Partially Synchronous Systems: Eventually Perfect Detection

A detector with only slightly weaker properties is possible if the upper bound on message delays is unknown or, for some reason, it is undesirable to include it in the program (e.g., portability reasons). The detector (Algorithm 16.4) starts with a small estimated value of μ. It is then possible that early time-outs take place, i.e., a process is timed out, and suspected, while actually an $\langle$ **alive** $\rangle$ message from the process is still in transit. If a message from a suspected process arrives, it is immediately deleted from D, but moreover, the estimate for μ is increased. After a fixed number of such corrections, the estimate exceeds the actual value of μ, and from then on no more erroneous suspicions occur. As the number of erroneous suspicions is finite, they all

occur in a finite initial segment of the execution, which shows that the detector is eventually strongly accurate.

16.4.3 Final Remarks

This chapter has only introduced the subject of failure detectors by presenting a few elementary results that can be easily understood and presented in class. Because the subject has been introduced relatively recently and has just caught the attention of the international research community, it is not possible to make a selection of the results that is indicative of future directions of research and development. We conclude the chapter with some examples of results concerning failure detectors.

A hierarchy of failure detectors is established by the notion of *emulation* of failure detectors [CT96]. Detector $\mathcal{A}$ emulates detector $\mathcal{B}$ if there exists a distributed algorithm that uses the information provided by $\mathcal{A}$ and implements detector $\mathcal{B}$. Detector $\mathcal{B}$ is said to be weaker than $\mathcal{A}$ in this case.

A weaker form of completeness was defined: a detector is weakly complete if every crashed process is eventually permanently suspected by at least one correct process [CT96]. However, it can be easily shown that a weakly complete detector can emulate a strongly complete one (see Definition 16.1) without weakening any of the accuracy properties of Definition 16.2. Therefore, weak completeness is not often considered in the literature (or in this book).

It has been asked whether the consensus problem is solvable with a detector that is weaker than $\diamond\mathcal{S}$ (as used in Section 16.3), but this is not the case. Chandra *et al.* [CHT96] show that any detector that allows one to implement consensus can emulate $\diamond\mathcal{S}$. The work develops a lot of mathematical machinery and proof techniques surrounding failure detection.

Not all failure detectors use time outs in their implementation: a *Heartbeat* [ACT97] detector was proposed that only exchanges asynchronous messages. This detector, however, necessitates a different kind of interaction with the processes from that used in this chapter, and the problem that can be solved with it is slightly contrived. The related issues, such as what can be implemented asynchronously, what problems can be solved with asynchronously implemented detectors, etc., are too complicated and too novel to discuss here.

Exercises to Chapter 16

Section 16.1

Exercise 16.1 *Page 509 lists various possibilities for correct processes to collect different sets of messages in a receive phase. For each of the four types of detector (Definition 16.3), say if this possibility can occur.*

Exercise 16.2 *Let F be a failure pattern; prove there is a t such that $F(t) = Crash(F)$.*
Where does your proof use that $\mathbb{P}$ is finite? Show that the result doesn't hold if $\mathbb{P}$ is an infinite collection of processes.

Exercise 16.3 *The requirement of strong accuracy is* stronger *than the requirement that no correct process is ever suspected:*

$$\forall F : \forall H \in \mathcal{D}(F) : \forall t : \forall p \in Corr(F), q \notin F(t) : p \notin H(q, t).$$

Give an example of a failure patern and failure detector history that satisfy this property, yet are not allowed in a strongly accurate detector.

Exercise 16.4 *Process p sends a message to q and then to r; q and r collect a message from p. Show with an example that it is possible, even if detection is* perfect, *that r receives the message and q doesn't. In your example, q and r must be correct.*

Section 16.2

Exercise 16.5 *A failure detector is called* l-accurate *if in each run at least l correct processes are unsuspected. (The property implies a bound of $N - l$ on the resiliency; weak accuracy is 1-accuracy.)*
Adapt Algorithm 16.1 so as to complete in $N - l + 1$ rounds.

Exercise 16.6 *Modify Algorithm 16.2 as described for Algorithm 14.5 so as to obtain a* finite *algorithm, or show that this is impossible with an eventually accurate detector.*

Section 16.3

Exercise 16.7 *Give an example of an execution of Algorithm 16.2 in which decisions occur in different rounds.*

Section 16.4

Exercise 16.8 *How much time can elapse in Algorithm 16.3 between a crash and its detection?*

Exercise 16.9 *Prove the eventual perfection of Algorithm 16.4. Is the relation* $me_p \geq \mu$ *eventually satisfied in every execution?*

17

Stabilization

The stabilizing algorithms considered in this chapter achieve fault-tolerant behavior in a manner radically different from that of the robust algorithms studied in the previous chapters. Robust algorithms follow a *pessimistic* approach, suspecting all information received, and precede all steps by sufficient checks to guarantee the validity of all steps of correct processes. Validity must be guaranteed in the presence of faulty processes, which necessitates restriction of the number of faults and of the fault model.

Stabilizing algorithms are *optimistic*, which may cause correct processes to behave inconsistently, but guarantee a return to correct behavior within finite time after all faulty behavior ceases. That is, stabilizing algorithms protect against *transient* failures; eventual repair is assumed, and this assumption allows us to abandon failure models and a bound on the number of failures. Rather than considering processes to be faulty, it is assumed that all processes operate correctly, but the configuration can be corrupted arbitrarily during a transient failure. Ignoring the history of the computation during the failure, the configuration at which we start the analysis of the algorithm, is considered the initial one of the (correctly operating) algorithm. An algorithm is therefore called stabilizing if it eventually starts to behave correctly (i.e., according to the specification of the algorithm), regardless of the initial configuration.

The concept of stabilization was proposed by Dijkstra [Dij74], but little work on it was done until the late nineteen-eighties; hence the subject can be considered relatively new. Nonetheless, a large number of stabilizing algorithms and related results were proposed in the following years up to the date of the present text, and in this chapter a selection of this work will be presented.

The term "stabilizing" is used throughout, whereas "self-stabilizing" is frequently found in the literature.

17.1 Introduction

17.1.1 Definitions

Stabilizing algorithms are modeled as transition systems without initial configurations (compare this definition with Definition 2.1).

Definition 17.1 *A system is a pair $S = (\mathcal{C}, \rightarrow)$, where $\mathcal{C}$ is a set of configurations and $\rightarrow$ is a binary transition relation on $\mathcal{C}$. An execution of S is a maximal sequence $E = (\gamma_0, \gamma_1, \gamma_2, \ldots)$ such that for all $i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$.*

Unlike in Definition 2.2, each (non-empty) suffix of a computation is now also a computation. The correctness of an algorithm, i.e., the desired "consistent behavior of processes", is expressed as a specification, which is a predicate (usually denoted P) on sequences of configurations.

Definition 17.2 *System S stabilizes to specification P if there exists a subset $\mathcal{L} \subseteq \mathcal{C}$, of legitimate configurations, with the following properties.*

(1) **Correctness.** *Every execution starting in a configuration in $\mathcal{L}$ satisfies P.*
(2) **Convergence.** *Every execution contains a configuration of $\mathcal{L}$.*

The set of legitimate configurations is usually closed, i.e., if $\gamma \in \mathcal{L}$ and $\gamma \rightarrow \delta$ then $\delta \in \mathcal{L}$, but as this is not used in the proof of the following result, we do not include closedness in the definition.

Theorem 17.3 *If system S stabilizes to P, then every execution of S has a non-empty suffix satisfying P.*

Proof. Every execution contains a legitimate configuration by the convergence property, and a suffix starting there satisfies P by the correctness. □

Proving stabilization. The use of legitimate configurations allows us to use the standard verification techniques to show the stabilization of an algorithm. Convergence is shown by a norm function.

Lemma 17.4 *Assume that*

(1) *all terminal configurations belong to $\mathcal{L}$;*

(2) *there exists a function* $f : \mathcal{C} \to W$, *where* W *is a well-founded set, and, for each transition* $\gamma \to \delta$, $f(\gamma) > f(\delta)$ *or* $\delta \in \mathcal{L}$ *holds.*

Then $S = (\mathcal{C}, \to)$ *satisfies convergence.*

To show correctness, one may perform a classical algorithm analysis, considering $\mathcal{L}$ as the set of initial configurations; indeed, only executions starting in $\mathcal{L}$ are considered.

Properties of stabilizing algorithms. Stabilizing algorithms offer the following three fundamental advantages over classical algorithms.

(1) *Fault tolerance.* As was observed in the introduction to this chapter, a stabilizing algorithm offers full and automatic protection against all transient process failures, because the algorithm recovers from any configuration, no matter how much the data has been corrupted by failures.
(2) *Initialization.* The need of proper and consistent initialization of the algorithm is eliminated, because the processes can be started in arbitrary states and yet eventual coordinated behavior is guaranteed.
(3) *Dynamic topology.* A stabilizing algorithm computing a topology-dependent function (routing tables, spanning tree) converges to a new solution after the occurrence of a topological change.

A fourth advantage, namely, the possibility of "sequential" composition without the need for termination detection, is discussed in Subsection 17.3.1. Finally, many of the stabilizing algorithms known to date are simpler then their classical counterparts for the same network problem. This, however, is partly because these algorithms are not yet optimized with respect to their complexity, so this "advantage" may vanish when the study of stabilizing algorithms develops further.

On the other hand, there are the following three disadvantages.

(1) *Initial inconsistencies.* Before a legitimate configuration is reached the algorithm may show an inconsistent output.
(2) *High complexity.* The stabilizing algorithms known to date are usually far less efficient than their classical counterparts for the same problem.
(3) *No detection of stabilization.* It is not possible to observe from within the system that a legitimate configuration has been reached; hence the processes are never aware of when their behavior has become reliable.

Pseudo-stabilization. The property of stabilizing algorithms that was proved in Theorem 17.3 can be taken as an alternative definition: it is then simply required that each execution has a suffix satisfying the specification. This notion, however, is not equivalent to our definition and is referred to as *pseudo-stabilization* by Burns *et al.* [BGM93].

To show the difference, consider the system S with configurations a and b and transitions $a \to a$, $a \to b$, and $b \to b$. Specification P reads "all configurations are the same". Because S can move from a to b at most once, it is easily seen that every execution has a suffix consisting of equal configurations. Configuration a cannot be chosen as legitimate, because the execution $(a, b, b, \ldots)$ starting in it does not satisfy P. Hence the S-execution $(a, a, a, \ldots)$ (although satisfying P itself) does not contain a legitimate configuration.

Burns *et al.* show that the weaker requirement of pseudo-stabilization allows solutions for problems that have no stabilizing solution; one example is data-sequence transmission. On the other hand, in pseudo-stabilizing solutions (that are not stabilizing) there is no upper bound on the number of steps that the system can take before specification P is satisfied, while for stabilizing algorithms such a bound can be given. In this chapter we shall restrict ourselves to the study of stabilizing algorithms.

17.1.2 Communication in Stabilizing Systems

In most of the earlier chapters in this book we have assumed an asynchronous model with communication by message passing, but this model is not adequate for studying stabilizing algorithms.

Indeed, first consider an asynchronous message-passing algorithm in which there is a process p for which every state is a send state (i.e., p can send a message in every local state). This system has an execution consisting only of send actions by p, while all other processes remain frozen in their initial state, and this behavior does not satisfy any meaningful non-trivial specification.

Second, consider an algorithm in which, for every process, there are states in which the process does not send (but can only receive or do an internal step). Every configuration in which every process is in such a state and all channels are empty is a terminal configuration and therefore must satisfy the specification. Again, no non-trivial specification is satisfied by all such configurations.

A stabilizing version of the Netchange algorithm was presented in [Tel91a], using timers in each process and an upper bound on the message-delivery

time. This assumption may work out well in practice, but the analysis of the algorithm is concerned mainly with technical details about timing. Therefore we shall assume the more usual model of communication by shared variables, where one process can write and other processes can read the same variable.

In *state* models a process can (atomically) read the entire state of its neighbor. In such a model every neighbor of p reads the same state, so it is not possible for a process to transfer different information to its various neighbors. In *link-register* models, communication between processes is by two registers shared between the two processes; each process can read one of these and write the other. A process can transfer different information to various neighbors by writing different values in its link registers.

We further distinguish between *read-one* and *read-all* models, in which a process can read in one atomic step the state (or link register) of one neighbor, or of all neighbors, respectively.

17.1.3 Example: Dijkstra's Token Ring

The first stabilizing algorithms were proposed by Dijkstra [Dij74] and these algorithms achieved *mutual exclusion* in a ring of processes. For this problem it is assumed that processes must sometimes execute *critical sections* of their code, but it is required that at most one process executes a critical section at any given time. Each process is provided with a local predicate that, when true, indicates that the process has the *privilege* of executing the critical section. The specification of the problem is then:

(1) In every configuration, at most one process has the privilege.
(2) Every process has the privilege infinitely often.

The problem can be solved by introducing a token that circulates over the ring and grants the privilege to the process holding it. The strength of Dijkstra's solution is that it automatically recovers (within $O(N^2)$ process steps) from situations where no token or more than one token is present in the system.

Description of the solution. The state of process i, denoted σ_i, is an integer in the range $0, ..., K-1$, where K is an integer exceeding N. Process i can read σ_{i-1} (as well as σ_i) and process 0 can read σ_{N-1}. All processes except process 0 are equal. Process $i \neq 0$ has the privilege if $\sigma_i \neq \sigma_{i-1}$, while process 0 has the privilege if $\sigma_0 = \sigma_{N-1}$. A process that has the privilege is also enabled to change its state (presumably it will do so after completing its critical section, or when it does not need to execute the latter at all).

The state change of a process always causes the loss of its privilege. Process $i \neq 0$ may set σ_i equal to σ_{i-1} (when it is enabled, i.e., $\sigma_i \neq \sigma_{i-1}$) by assigning $\sigma_i := \sigma_{i-1}$. Process 0 may set σ_0 unequal to σ_{N-1} (when it is at present equal) by assigning $\sigma_0 := (\sigma_{N-1} + 1) \bmod K$.

Proof of stabilization. The definition of privilege implies that in every configuration at least one process has the privilege (i.e., there are no terminal configurations). Indeed, if no process other than 0 has the privilege, then $\sigma_i = \sigma_{i-1}$ for every $i > 0$, which implies that $\sigma_0 = \sigma_{N-1}$ and process 0 has the privilege. Further, no step increases the number of privileged processes, because a process changing state loses its privilege and the only process that can obtain a privilege in this step is its successor.

To show stabilization, define $\mathcal{L}$ as the set of configurations in which exactly one process has the privilege.

Lemma 17.5 *Executions starting in legitimate configurations satisfy P.*

Proof. In a legitimate configuration exactly one step is possible, namely by the unique privileged process. In this step, this process loses its privilege, but, because there are no configurations without a privilege, its successor necessarily obtains the privilege. Consequently, the privilege circulates over the ring and every process holds the privilege once in every N consecutive configurations. □

Lemma 17.6 *Dijkstra's token ring converges to $\mathcal{L}$.*

Proof. Again, every execution of the system is infinite by the absence of terminal configurations.

An execution contains infinitely many steps of process 0, because at most $\frac{1}{2}N(N-1)$ steps can occur without a step of process 0. Indeed, because a step of i removes i's privilege and may give a privilege only to $i+1$, the norm function

$$F = \sum_{i \in S} (N - i)$$

$$\text{where } S = \{i : i > 1 \text{ and process } i \text{ has the privilege}\}$$

decreases with every step of a process other than process 0.

In the initial configuration γ_0 at most N different states occur so there are at least $K - N$ states that do not occur in γ_0. Because process 0 increments its state (modulo K) in each of its steps, it reaches a state not occurring in γ_0 after at most N steps (of process 0). All processes other than 0 only

copy states, hence the first time process 0 computes such a state, its state is unique in the ring. As process 0's state is unique, it will not get the privilege again before the configuration satisfies $\sigma_1 = \sigma_2 = \ldots = \sigma_{N-1} = \sigma_0$, which describes a legitimate configuration.

As the system reaches a legitimate configuration before the $(N+1)$th step of process 0 and (at most) $\frac{1}{2}N(N-1)$ steps of other processes can occur consecutively, a legitimate configuration is reached after (at most) $(N+1)\frac{1}{2}N(N-1) + N = \mathrm{O}(N^3)$ steps.

With a more precise analysis an $\mathrm{O}(N^2)$ bound on the number of steps can be proved; see Exercise 17.2. □

Corollary 17.7 *Dijkstra's token ring stabilizes to mutual exclusion.*

Uniform solutions. A stabilizing algorithm is called *uniform* if all processes are equal and have no identities to distinguish between them. Finding uniform solutions to mutual exclusion and other problems has received considerable attention in recent years, and this interest can be explained mainly as due to their theoretical interest. Uniform solutions are also attractive, however, in practice, because a process identity is usually stored in memory like any other process variable, so that a corrupted configuration may violate the required uniqueness of identities.

A uniform stabilizing algorithm is also an algorithm for an anonymous network for the same problem; consequently, problems unsolvable for anonymous networks (see Chapter 9) do not have uniform stabilizing solutions. By assuming a symmetric configuration and showing that the symmetry may be preserved indefinitely, Dijkstra [Dij82] showed that no uniform (deterministic) token ring exists if the size of the ring is not prime. A uniform solution, assuming the ring size is prime and known, was proposed by Burns and Pachl [BP88].

The impossibility of deterministic uniform stabilizing algorithms for election, spanning tree, and other symmetry-breaking tasks can be shown similarly, but solutions usually become possible if randomization is allowed. As this chapter is concerned with stabilization, and not with anonymity, we shall assume freely in the sequel that unique names or a leader are available.

17.2 Graph Algorithms

If the purpose of an algorithm is to achieve a postcondition ψ, its specification may be set to "every configuration satisfies ψ". We say S stabilizes to ψ if a set $\mathcal{L}$ exists such that

(1) $\mathcal{L}$ is closed and $\gamma \in \mathcal{L}$ implies $\psi(\gamma)$; and
(2) every computation reaches a configuration of $\mathcal{L}$.

The algorithm may terminate, but this is not always the case. Subsection 17.2.1 presents a ring-orientation algorithm that may eventually go through an infinite sequence of different (but oriented) configurations. Subsection 17.2.2 presents an algorithm for finding a maximal matching, in which the legitimate configurations are terminal.

17.2.1 Ring Orientation

In the ring-orientation problem we consider an undirected ring of N processes, where each process has labeled one of its links by *succ* (successor) and the other by *pred* (predecessor). (A process immediately changes one of its labels if the two labels are equal.) Process p's label of the link pq is called l_{pq}, and no global consistency of the labels is initially assumed. The goal of an orientation algorithm is to compute a consistent sense of direction in the ring.

The orientation problem requires the system to reach the postcondition ψ, defined by "for every edge pq, l_{pq} is *succ* if and only if l_{qp} is *pred*". It was shown by Israeli and Jalfon [IJ90] that no deterministic solution is possible in the state-reading models; we present the Israeli–Jalfon algorithm for the link-register model. The presented algorithm is uniform.

The algorithm, given as Algorithm 17.1, uses, besides the link registers holding *pred* or *succ*, a variable s with values S, R, and I. In one step, process p considers its state, the state of neighbor q and the link registers, and changes state if one of the five guards is *true*. The processes circulate tokens, each process setting its successor to the direction of the last forwarded token; if two tokens meet, one is eliminated. Eventually all remaining tokens will travel in the same direction, which causes the ring to be oriented; the remaining tokens will then keep circulating.

In the S state, a process waits to forward a token (to its current successor), while in the R state a process waits to receive a token; in the I state a process is idle. A process *holds a token* if either its state is S or its state is R and its predecessor is not in state S. A token moves from p to q in action (2) of the algorithm, and is destroyed or created in p in actions (4) or (5), respectively; these steps are called *token steps*. Steps (1) and (3) have no effect on the location of tokens and are called *silent steps*. After step (1), the next step of the same process is step (3), and after step (3) the next step is step (4)

var s_p : (S, R, I) ;

(1) $\{state_p = I \wedge s_q = S \wedge l_{qp} = succ\}$
$s_p := R$; **if** $l_{pq} = succ$ **then** *flip*

(2) $\{s_p = S \wedge l_{pq} = succ \wedge s_q = R \wedge l_{qp} = pred\}$
$s_p := I$

(3) $\{s_p = R \wedge l_{pq} = pred \wedge \neg(s_q = S \wedge l_{qp} = succ)\}$
$s_p := S$

(4) $\{s_p = s_q = S \wedge l_{pq} = l_{qp} = succ\}$
$s_p := R$; *flip*

(5) $\{s_p = s_q = I \wedge l_{pq} = l_{qp} = succ\}$
$s_p := S$

procedure *flip*: reverse *succ* and *pred* for p

Algorithm 17.1 RING-ORIENTATION ALGORITHM.

or step (2), which bounds the number of silent steps in linear relation to the number of token steps.

In action (1), if q wants to forward a token to the idle p, p agrees to accept the token (although it does not move yet) and makes q its predecessor. In action (2), p observes that its successor has agreed to accept the token it wants to transmit and becomes idle, thus moving the token. In action (3), p observes that q, from which it has accepted a token, has become idle, and starts transmitting the token to its successor. The last two actions concern the symmetric situations where (4) two tokens traversing in different directions meet, and (5) two idle processes are inconsistently oriented. In these cases, both processes are enabled, and the first process that takes a step breaks the symmetry[1]. In action (4), p accepts q's token, implicitly destroying its own. In action (5), p generates a token that will overrule q's orientation.

The effect of each action is visualized in Table 17.2, where the state and orientation of each process is compactly represented by an arrow in the direction of its successor.

A configuration is called legitimate if and only if it is oriented, i.e., all arrows point in the same direction.

[1] Israeli and Jalfon consider a finer grain of atomicity of action, which necessitates breaking the symmetry explicitly. In terms of [IJ90], Algorithm 17.1 works for the central demon.

Action no.	q	p		p
(1)	$\overrightarrow{S}$	I	$\longrightarrow$	$\overrightarrow{R}$
(2)	$\overleftarrow{R}$	$\overleftarrow{S}$	$\longrightarrow$	$\overleftarrow{I}$
(3)	$\neg\ \overrightarrow{S}$	$\overrightarrow{R}$	$\longrightarrow$	$\overrightarrow{S}$
(4)	$\overrightarrow{S}$	$\overleftarrow{S}$	$\longrightarrow$	$\overrightarrow{R}$
(5)	$\overrightarrow{I}$	$\overleftarrow{I}$	$\longrightarrow$	$\overleftarrow{S}$

Action (1) is applicable regardless of the original orientation of p.

Table 17.2 EFFECT OF THE ACTIONS OF ALGORITHM 17.1.

Lemma 17.8 *$\mathcal{L}$ is closed and $\gamma \in \mathcal{L}$ implies $\psi(\gamma)$.*

Proof. Legitimate configurations are oriented by the choice of $\mathcal{L}$, which immediately implies the second part. Further, only steps of the types (1), (2), and (3) occur, because steps (4) and (5) are only enabled when there are differently oriented neighbors. Steps (2) and (3) never flip the orientation of a process, and neither does (1) if p is oriented in the same direction as q, which shows the closedness of $\mathcal{L}$. □

Proposition 17.9 *A terminal configuration is legitimate.*

Proof. Let γ be terminal; no process is in state R in γ, because if $s_p = R$ then either p is enabled, by action (3), or its predecessor is enabled, by action (2).

If γ is not oriented there are two processes pointing at each other, i.e., p and q with $l_{pq} = l_{qp} = succ$. If one of p and q is idle action (1) is enabled, if both are idle action (5) is enabled, and if both are sending action (4) is enabled. It follows that a terminal configuration without receiving processes is oriented; hence all terminal configurations are oriented. □

Theorem 17.10 *The protocol converges to a legitimate state.*

Proof. Token creation occurs in a pair $\overrightarrow{I}\,\overleftarrow{I}$, and such a pair is not formed in any of the five actions, so a token is generated only in an initially idle process as the first step of that process. This implies that the overall number of tokens that exists (i.e., initial tokens plus tokens created during execution) in an execution is bounded by N.

If a token exists in a configuration during the execution and this token has already moved k times, then a sequence of $k + 1$ processes (ending in the one holding the token) is consistently oriented. All these processes have forwarded the token and adopted its direction, and no tokens that could

have disturbed the orientation again have been created "behind" the token. So if one token moves $N-1$ times all processes are oriented equally, and the configuration is legitimate.

Because there are no more than N different tokens during the execution, a legitimate configuration is reached after at most N^2 token steps. □

17.2.2 Maximal Matching

A *matching* in a graph is a set of edges such that no node of the graph is incident to more than one of the edges. The matching is *maximal* if it cannot be extended by more edges of the graph, and a maximal matching of a graph can be constructed in a time linear in the number of edges. Each edge is considered in turn, and included in the matching if it is not incident to any edge already in the matching. This algorithm is, however, inherently sequential, and not suited for distributed execution; we shall now present a stabilizing algorithm for constructing maximal matchings, proposed by Hsu and Huang [HH92].

The matching algorithm. Process p has one variable $pref_p$, whose value belongs to $Neigh_p \cup \{nil\}$, representing p's *preferred neighbor*. If $pref_p = q$, p has selected its neighbor q to become matched with, i.e., to include edge pq in the matching; $pref_p = nil$ if p has not selected a matching partner. We distinguish five cases depending on p's preference and that of its neighbors. If p has selected q, then p is waiting (for q) if q has not made a selection yet, matched if q has selected p, and chaining if q has selected a neighbor other than p. If p has not made a selection, p is dead if all neighbors of p are matched and free if there is an unmatched neighbor. Formally,

$$\begin{aligned}
wait(p) &\equiv pref_p = q \in Neigh_p \wedge pref_q = nil \\
match(p) &\equiv pref_p = q \in Neigh_p \wedge pref_q = p \\
chain(p) &\equiv pref_p = q \in Neigh_p \wedge pref_q = r \in Neigh_q \wedge r \neq p \\
dead(p) &\equiv pref_p = nil \wedge \forall q \in Neigh_p : match(q) \\
free(p) &\equiv pref_p = nil \wedge \exists q \in Neigh_p : \neg match(q).
\end{aligned}$$

The required postcondition for the algorithm is

$$\psi \equiv \forall p : (match(p) \vee dead(p)).$$

Proposition 17.11 *If ψ holds, the set $M = \{(p, pref_p) : pref_p \neq nil\}$ is a maximal matching.*

var $pref_p$: $Neigh_p \cup \{nil\}$;

$\mathbf{M}_p$: $\{pref_p = nil \wedge pref_q = p\}$
$pref_p := q$

$\mathbf{S}_p$: $\{pref_p = nil \wedge \forall r \in Neigh_p : pref_r \neq p \wedge pref_q = nil\}$
$pref_p := q$

$\mathbf{U}_p$: $\{pref_p = q \wedge pref_q \neq p \wedge pref_q \neq nil\}$
$pref_p := nil$

Algorithm 17.3 THE MAXIMAL-MATCHING ALGORITHM.

Proof. If there is an edge $pq \in M$, then $pref_p = q$ or $pref_q = p$ by the definition of M; but because q is not waiting or chaining, the latter implies $pref_p = q$ as well. It follows that at most one edge incident to p belongs to M, hence M is a matching.

To show maximality, assume $M \cup \{pq\}$, where $pq \notin M$, is also a matching; then M contains no edge incident to p, which implies $dead(p) \vee free(p)$, and hence, by ψ, $dead(p)$. But $dead(p)$ implies $match(q)$, so M contains an edge incident to q, contradicting the possible extension of M by pq. □

The algorithm is described for the read-all state model and consists of three actions for each process p; see Algorithm 17.3. Process p matches with neighbor q (action $\mathbf{M}_p$) if p is free and q has selected p. If p is free but cannot match, it selects a free neighbor if such is possible (action $\mathbf{S}_p$), and if p is chaining it unchains (action $\mathbf{U}_p$).

Analysis of the algorithm.

Lemma 17.12 *Configuration γ is terminal if and only if $\psi(\gamma)$ holds.*

Proof. Action $\mathbf{M}_p$ is enabled for p only if neighbor q is waiting, action $\mathbf{S}_p$ requires that p is free, and action $\mathbf{U}_p$ that p is chaining, hence $\psi(\gamma)$ implies that no action is enabled in γ.

If ψ does not hold, there is a p such that p is chaining, waiting, or free; in a chaining p, $\mathbf{U}_p$ is enabled, and if p is waiting for q, $\mathbf{M}_q$ is enabled. Finally, assume p is free and q is an unmatched neighbor. Because p is not matched, q is not dead. If q is waiting, the matching action is enabled for one of its neighbors and if q is chaining $\mathbf{U}_q$ is enabled. Finally, if q is free, p and q can select each other. Hence, if ψ does not hold, the configuration is not terminal. □

Lemma 17.13 *Algorithm 17.3 reaches a terminal configuration in* $O(N^2)$ *steps.*

Proof. Define the norm function F by the pair $(c+f+w,\ 2c+f)$ where c, f, and w are the numbers of chaining, free, and waiting processes respectively. We show that F decreases (in the lexicographic order) with every step; first observe that a matched or dead process remains matched or dead forever, so $c+f+w$ never increases.

- $\mathbf{M}_p$: This action applies when p is free and its neighbor q is waiting, and causes both to become matched, decreasing $c+f+w$ by 2. (The sum may decrease even further if some neighbors of p and q become dead.)
- $\mathbf{S}_p$: This action is applicable when p is free, and causes p to become waiting, thus decreasing $2c+f$ by 1. No waiting process becomes free or chaining, because the action is applicable only if no process waits for p, and no free process becomes chaining.
- $\mathbf{U}_p$: This action is applicable when p is chaining, and causes p to become free (if there are unmatched neighbors) or dead (if all neighbors are matched), thus decreasing $2c+f$ by at least 1. Chaining neighbors of p may become waiting, thus further decreasing c.

As $c+f+w$ is bounded by N and $2c+f$ by $2N$, the number of different values of F is at most $(N+1)(2N+1)$, so each execution terminates in $2N^2+3N$ steps. □

Algorithm 17.3 is now easily seen to stabilize to ψ, by taking all configurations satisfying ψ as the legitimate ones.

17.2.3 Election and Spanning-tree Construction

Afek, Kutten, and Yung [AKY90] proposed a stabilizing algorithm for computing a spanning tree on a network, in which the largest process is the root. We describe the algorithm for the read-all state model (although the algorithm can also be used for the read-one model) and assume that the network is connected. In this subsection we shall use capitalized initials for process variables, and lower-case initials for predicates.

Process p maintains the variables $Root_p$, Par_p, and Dis_p to describe the

tree structure; we introduce the following predicates.

$$
\begin{array}{ll}
root(p) & \equiv Root_p = p \wedge Dis_p = 0 \\
child(p, q) & \equiv Root_p = Root_q > p \wedge Par_p = q \in Neigh_p \wedge Dis_p = Dis_q + 1 \\
tree(p) & \equiv root(p) \ \vee \ \exists q : child(p, q) \\
lmax(p) & \equiv \forall q \in Neigh_p : Root_p \geq Root_q \\
sat(p) & \equiv tree(p) \wedge lmax(p)
\end{array}
$$

The intended postcondition of the algorithm is ψ, defined by $\forall p : sat(p)$.

Lemma 17.14 *ψ implies that the edges $\{(p, q) : child(p, q)\}$ form a spanning tree with the largest process as the root.*

Proof. As $lmax(p)$ is satisfied for all p and the network is connected, all processes have the same value of *Root*. The process p_0 with minimal value of *Dis* does not have a neighbor q with $Dis_p = Dis_q + 1$, so $root(p_0)$ holds and the common value of the *Root* variables is p_0. But then, for all $p \neq p_0$, $child(p)$ holds, and, because $child(p, q) \Rightarrow (Dis_p > Dis_q)$, the *child*-relation is acyclic. The result follows. □

Description of the algorithm. In order to establish ψ, process p with $sat(p)$ true never changes those of its variables that describe the tree structure (see Algorithm 17.4). Process p with $sat(p)$ false attempts to establish $sat(p)$ by becoming a child of its neighbor with the highest value of *Root* (action $\mathbf{J}_p$).

Call the value of $Root_q$ a *false root* if there is no process with that identity in the network; false roots may exist initially but are not created during execution. A main problem in the design of the algorithm is to prevent processes from becoming the child of a process with a false root infinitely often. To this end, joining q's tree by p is done in three steps. First, p becomes a root (action $\mathbf{B}_p$), then asks for join permission (action $\mathbf{A}_p$), and finally joins when q grants the permission (action $\mathbf{J}_p$).

The remaining four actions ($\mathbf{C}_p$, $\mathbf{F}_p$, $\mathbf{G}_p$, and $\mathbf{R}_p$) implement the request/reply mechanism and are only executed by processes p with $sat(p)$. Variable Req_p contains the process whose join request p is currently processing; $From_p$ is the neighbor from whom p read the request; To_p is the neighbor to which p forwards it; and Dir_p (values *Ask* and *Grant*) indicates

var $Root_p$, Par_p , Dis_p ; (* Describe tree structure *)
Req_p, $From_p$, To_p, Dir_p ; (* Request forwarding *)

$\mathbf{B}_p$: (* Become root *)
{ $\neg tree(p)$ }
$Root_p := p$; $Dis_p := 0$;
$Req_p := p$; $To_p := q$; $Dir_p := Ask$

$\mathbf{A}_p$: (* Ask permission to join *)
{ $tree(p) \wedge \neg lmax(p)$ }
Select $q \in Neigh_p$ with maximal value of $Root_q$;
$Req_p := p$; $From_p := p$; $To_p := q$; $Dir_p := Ask$

$\mathbf{J}_p$: (* Join tree *)
{ $tree(p) \wedge \neg lmax(p) \wedge grant(To_p, p)$ }
$Par_p := q$; $Root_p := Root_q$; $Dis_p := Dis_q + 1$;
$Req_p := From_p := To_p := Dir_p := udef$

$\mathbf{C}_p$: (* Clear request variables *)
{ $sat(p) \wedge \neg\exists q : forw(p, q) \wedge \neg idle(p)$ }
$Req_p := From_p := To_p := Dir_p := udef$

$\mathbf{F}_p$: (* Forward request *)
{ $sat(p) \wedge idle(p) \wedge asks(q, p)$ }
$Req_p := Req_q$; $From_p := q$; $To_p := Par_p$

$\mathbf{G}_p$: (* Grant join request *)
{ $sat(p) \wedge root(p) \wedge forw(p, q) \wedge Dir_p = Ask$ }
$Dir_p := Grant$

$\mathbf{R}_p$: (* Relay grant *)
{ $sat(p) \wedge grant(Par_p, p) \wedge Dir_p = Ask$ }
$Dir_p := Grant$

Algorithm 17.4 THE SPANNING-TREE ALGORITHM.

whether the request was granted. We define the following predicates.

$$
\begin{aligned}
idle(p) &\equiv Req_p = From_p = To_p = Dir_p = udef \\
asks(p, q) &\equiv ((root(p) \wedge Req_p = p) \vee child(p, q)) \\
&\quad \wedge To_p = q \wedge Dir_p = Ask) \\
forw(p, q) &\equiv Req_p = Req_q \wedge From_p = q \wedge To_q = p \wedge To_p = Par_p \\
grant(p, q) &\equiv forw(p, q) \wedge Dir_p = Grant
\end{aligned}
$$

Process p clears the variables for request processing (action $\mathbf{C}_p$) if it is not currently processing a request and the variables are not undefined already. Then, if p is idle but has a neighbor q with a request for p (a root that wants

to join, or a child of p forwarding a request), process p may start forwarding the request (action $\mathbf{F}_p$). If p is a root and forwards a request, it will grant it (action $\mathbf{G}_p$), and if p forwards the request to its parent and the parent grants it, then p relays the grant (action $\mathbf{R}_p$).

Correctness of the algorithm. Afek *et al.* [AKY90] argue the correctness of the algorithm by showing properties of executions using behavioral reasoning.

The request/reply mechanism ensures that a process can join a tree with a false root only finitely often, because the non-existent root does not grant requests, and only finitely many false grants exist initially. Then, no *Root* variable contains a false root forever; the process with the smallest value of *Dis* containing a false root does not satisfy *tree*, and will reset *Root* to its own identity. It follows that eventually there are no false roots, and also that eventually the process with highest identity will be a root, for this is the only way to satisfy *sat* in the absence of higher *Root* values.

A node that does not belong to the tree rooted at the highest node, but does have a neighbor in that tree, does not satisfy the *sat* predicate and will attempt to join, while nodes in the tree never leave it. As all incorrect requests eventually disappear from the tree, addition of nodes to the tree continues until the tree spans the whole network.

17.3 Methodology for Stabilization

The requirement of stabilization complicates the design of distributed algorithms considerably, but some general paradigms can be used. In Subsection 17.3.1 we discuss how stabilizing computations can be "sequentially composed", even though the stabilization of the first cannot be detected. We focus on Herman's work [Her91], but the techniques have been known and used before that. In Subsection 17.3.2 it is demonstrated that stabilizing algorithms exist for all problems that can be expressed as finding a "minimal path" in a suitably chosen measure.

17.3.1 Protocol Composition

In many cases an algorithm for achieving a desired postcondition ψ is composed of two stages; the first achieves a postcondition θ, while the second has precondition θ and achieves ψ. Examples of composed algorithms are numerous and include some algorithms treated in this book.

(1) *Routing.* Most routing methods start by computing routing tables in every node, after which packets are forwarded using these tables.
(2) *Election.* The reason for electing a leader in a network is usually the desire to subsequently execute a centralized algorithm in the network.
(3) *Deadlock detection.* The global-marking algorithm for deadlock detection (Algorithm 10.8) first computes a snapshot of the basic computation and subsequently analyses the configuration obtained.
(4) *Graph coloring.* A six-coloring of a planar graph can be computed by finding a suitable acyclic orientation of the graph (cf. Definition 5.12), after which the nodes can be colored in an order consistent with this orientation.

The natural way to guide the control switch between two classical distributed algorithms is by means of a termination-detection protocol. Such a protocol (see Chapter 8) is superimposed on the first phase, and when it indicates that the first stage is completed (i.e., θ has been established), the processes start executing the second stage. A similar method cannot be used when the designed algorithm must be stabilizing, because it is not possible to observe (in a stabilizing manner) the termination of the first stage.

Fortunately, a similar composition of stabilizing algorithms is much simpler: execution of the second stage may begin at any time, even when the first stage is not yet complete. If a classical algorithm (for the second stage) is started before its precondition has been established, it may reach a faulty state from which it does not recover; this necessitates detecting the termination of the first stage. However, if the second stage is stabilizing, it eventually establishes its postcondition in spite of erroneous steps made before its precondition has reached the value *true*.

The collateral-composition rule. These observations were formalized by Herman [Her91] in the definition of an algorithm-composition operator. Assume a program is given as a collection of variables and atomic steps on those variables.

Definition 17.15 *Let S_1 and S_2 be programs such that no variable that is written by S_2 occurs in S_1. The collateral composition of S_1 and S_2, denoted $S_1 \boxright S_2$, is the program that has all the variables and all the actions of both S_1 and S_2.*

In this definition, S_1 is the program implementing the first stage of the computation, and S_2 the program for the second stage. The restriction on

S_1 and S_2 means that no results of S_2 are used by S_1, that is, there is no "feedback" from S_2 to S_1; the variables of S_1 occur as constants in S_2.

Now let θ be a predicate over the variables of S_1, and ψ a predicate over the variables of S_2. In the composite algorithm, θ will be established by S_1 and subsequently ψ will be established by S_2; two technical conditions must be kept in mind. First, the execution of the composition must be *fair* w.r.t. both programs; this excludes executions in which all steps occur in one of the stages, preventing progress in the other stage.

Definition 17.16 *An execution of $S_1 \rhd S_2$ is fair w.r.t. S_i if it contains infinitely many steps of S_i, or contains an infinite suffix in which no step of S_i is enabled.*

Second, we assume that the variables read by S_2 are not changed by S_1 when condition θ has been established. This excludes some (far-fetched) situations where S_1 prevents progress in S_2 by presenting with it different settings of the variables, all satisfying θ.

Theorem 17.17 *If the following four conditions hold:*

(1) *program S_1 stabilizes to θ;*
(2) *program S_2 stabilizes to ψ if θ holds;*
(3) *program S_1 does not change variables read by S_2 once θ holds; and*
(4) *all executions are fair w.r.t. both S_1 and S_2,*

then $S_1 \rhd S_2$ stabilizes to ψ.

Proof. We establish the result by reasoning involving the execution sequences of $S_1 \rhd S_2$. For a configuration γ of $S_1 \rhd S_2$, let $\gamma^{(i)}$ denote the projection of γ onto the variables of S_i. For an execution $E = (\gamma_0, \gamma_1, \ldots)$ of $S_1 \rhd S_2$, let $E^{(i)}$ denote the sequence $(\gamma_0^{(i)}, \gamma_1^{(i)}, \ldots)$ from which duplicates have been eliminated.

Let $E = (\gamma_0, \gamma_1, \ldots)$ be an execution of $S_1 \rhd S_2$, and consider the sequence $E^{(1)}$. As S_2 does not write the variables of S_1, all changes in $\gamma^{(1)}$ are by steps of S_1, which implies that $E^{(1)}$ is an execution of S_1. The fairness of E w.r.t. S_1 implies that if $E^{(1)}$ is finite, the last configuration is terminal for S_1. Because S_1 stabilizes to θ, it follows that θ is established, i.e., that there is an i such that $\theta(\gamma_j)$ is true for every $j \geq i$.

We next consider the suffix E_i of E starting in γ_i, i.e., the sequence $(\gamma_i, \gamma_{i+1}, \ldots)$. In this suffix θ is true for every configuration, which implies (by the third assumption in the theorem) that no variable read by S_2 is changed by S_1. Hence, the sequence $(\gamma_i^{(2)}, \gamma_{i+1}^{(2)}, \ldots)$ is an execution of S_2

that, if finite, ends in a terminal configuration of S_2. Because S_2 stabilizes to ψ if θ holds, it follows that ψ is established and remains true. □

If the stabilization of S_1 and S_2 is proved by providing adequate norm functions f_1 and f_2 (with ranges W_1 and W_2 respectively), a norm function for $S_1 \triangleright S_2$ can be given. Define the function $f(\gamma) = (f_1(\gamma^{(1)}), f_2(\gamma^{(2)}))$. When every step of S_1 decreases f_1, and a step of S_2 decreases f_2 and leaves f_1 unchanged, f decreases in the lexicographic order with every step of $S_1 \triangleright S_2$. However, if one of the programs establishes its postcondition without terminating (such as, e.g., the ring orientation program), the fairness condition is still needed to guarantee progress in both programs; see [Fra86] for a discussion of fairness issues.

Application: Six-coloring of planar graphs. As an application of the principle of protocol composition we present an algorithm that stabilizes to a six-coloring of a planar graph. Assume process p has a variable c_p, with values in $\{1, 2, 3, 4, 5, 6\}$; the graph is six-colored if neighbors are colored differently, i.e., if the predicate ψ, defined by

$$\psi \equiv (\forall pq \in E : c_p \neq c_q),$$

is true. The development of the algorithm closely follows the proof that such a coloring exists, which is based on the following fact.

Fact 17.18 *A planar graph has at least one node with degree five or less.*

Lemma 17.19 *A planar graph has an acyclic orientation in which every node has an out-degree of at most five.*

Proof. This is by induction on the number of nodes; the single-node case is trivial, so consider a graph with two or more nodes. There is at least one node, say v, with degree five or less. The graph $G - \{v\}$ is planar and has one node less, hence by induction it has an acyclic orientation in which every node has an out-degree of at most five. In the orientation of G, all edges of $G - \{v\}$ are directed as in this orientation, and the edges incident to v are directed away from v. Clearly, the resulting orientation is acyclic and no node has more than five outgoing edges. □

Theorem 17.20 (Six-color theorem) *Planar graphs are six-colorable.*

Proof. Let G be a planar graph; consider an acyclic orientation of G in which no node has more than five outgoing edges. Because the orientation is acyclic, the nodes can be numbered v_1 through v_n such that if there is

an edge directed from v_j to v_i, then $j > i$. Color the nodes in the order v_1 through v_n; the property of the numbering implies that before node v only out-neighbors of v are colored. This, and the property of the orientation, now imply that at most five neighbors of v are colored before v, hence v can be colored differently from all (already colored) neighbors. □

The proof of the six-color theorem suggests the design of an algorithm for computing a six-coloring; the first stage computes an acyclic orientation as indicated in Lemma 17.19, and the second stage colors the nodes in an order consistent with the orientation. To represent an acyclic orientation, process p is given an integer variable x_p, and we define edge pq to be directed from p to q, denoted $\vec{pq}$, if

$$x_p < x_q \ \vee \ (x_p = x_q \wedge p < q).$$

Clearly, each edge is directed in exactly one way (i.e., either $\vec{pq}$ or $\vec{qp}$ holds), and the orientation is acyclic. Let $out(p)$ denote the number of outgoing edges of p, i.e.,

$$out(p) = \#\{q \in Neigh_p : \vec{pq}\}.$$

The desired postcondition of the first phase is θ, defined by

$$\theta \equiv (\forall p : \ out(p) \leq 5).$$

Program S_1, for establishing θ, consists of one operation for each process, directing all edges of p towards p if p has more than five outgoing edges:

$\mathbf{D}_p$: { $out(p) > 5$ }
$x_p := \max\{x_q : q \in Neigh_p\} + 1$

Theorem 17.21 *Program S_1 stabilizes to θ, and if θ holds then the value of x_p remains constant.*

Proof. A configuration is terminal if and only if θ is true; hence, if the program halts θ is established, and if θ is established the variables x_p remain constant.

To prove the termination of S_1 we use a "reference" acyclic orientation as in Lemma 17.19; this reference orientation need not be, however, the orientation computed by S_1. Lemma 17.19 implies the existence of an enumeration $v_1, v_2, \ldots$ of the nodes such that every node has at most five neighbors with a higher index. An edge v_iv_j is called *wrong* if it is oriented as $\vec{v_iv_j}$ while $i > j$; let $f(\gamma)$ be the list $(n_1, n_2, \ldots)$, where n_i is the number of wrong edges incident to v_i. Each application of action $\mathbf{D}_p$ decreases f (in the lexicographic order); consider its application in node v_j, whose out-neighbor

var x_p : integer ;
c_p : {1, 2, 3, 4, 5, 6} ;

$\vec{pq} \equiv x_p < x_q \ \vee \ (x_p = x_q \wedge p < q)$;
$out(p) \equiv \#\{q \in Neigh_p : \vec{pq}\}$;

$\mathbf{D}_p$: { $out(p) > 5$ }
$x_p := \max\{x_q : q \in Neigh_p\} + 1$

$\mathbf{C}_p$: { $(\exists q : \vec{pq} \wedge c_p = c_q) \wedge (\forall r \text{ s.t. } \vec{pr} : c_r \neq b)$ }
$c_p := b$

Algorithm 17.5 SIX-COLORING ALGORITHM FOR PLANAR GRAPHS.

with the smallest index is v_i. As v_j has at least six out-neighbors but has at most five neighbors with higher index, $i < j$ so the edge $v_i v_j$ was wrong but ceases to be so, thus decrementing n_i, while no n_k is affected for $k < i$. □

Program S_2, for establishing ψ, also consists of a single operation for each process, in which p adopts a color b different from its successors if c_p equals one of those colors and an unused color exists:

$\mathbf{C}_p$: { $(\exists q : \vec{pq} \wedge c_p = c_q) \wedge (\forall r \text{ s.t. } \vec{pr} : c_r \neq b)$ }
$c_p := b$

Theorem 17.22 *Program S_2 terminates, and if θ is true the final configuration satisfies ψ.*

Proof. To show termination, use that the edges are directed as in an acyclic orientation, hence there exists an enumeration of the nodes $v_1, v_2, \ldots$ such that each node has only successors with a smaller index. Define $g(\gamma) = (m_1, m_2, \ldots)$, where m_i is 0 if c_i is different from the colors of each of v_i's successors and 1 otherwise. The application of action $\mathbf{C}_p$ by node v_i changes m_i from 1 to 0, and may only change m_j from 0 to 1 for $j > i$, hence the value of g decreases in the lexicographic order.

Finally, assume θ holds and consider a configuration in which no $\mathbf{C}_p$ action is applicable. Process p has at most five successors, hence there is a color $b \in \{1, 2, 3, 4, 5, 6\}$ not used by any of its successors. As the guard of $\mathbf{C}_p$ is false, it follows that c_p differs from the color of every successor.

Finally, for every edge qr, either r is a successor of q or q is a successor of r, which now implies that $c_r \neq c_q$. □

The collateral composition of S_1 and S_2 is given as Algorithm 17.5.

Theorem 17.23 *Algorithm 17.5 stabilizes to ψ.*

Proof. The collateral composition can be formed because the variables written by S_2 (the c_p) do not occur in S_1. It was demonstrated above that S_1 stabilizes to θ and does not change the x_p after establishing it (Theorem 17.21) and that S_2 stabilizes to ψ if θ holds (Theorem 17.22).

In order to conclude that $S_1 \boxdot S_2$ stabilizes to ψ by Theorem 17.17 it remains to show that the composition is fair. By Theorem 17.22, there are only finitely many steps of S_2 between every two steps of S_1, which shows fairness w.r.t. S_1. By Theorem 17.21 S_1 terminates, which shows fairness w.r.t. S_2. □

17.3.2 Computing Minimal Paths

This subsection presents a stabilizing algorithm for a class of problems that can be formulated as finding a minimal path in a suitably chosen cost function on paths. Examples include not only the (straightforward) computation of routing tables, but also depth-first search trees and, more surprisingly, election (see Exercise 17.12). We shall first present the minimal-path problem in an algebraic setting and present the Update algorithm (Algorithm 17.6), which stabilizes to minimal paths; applications are given at the end of this subsection.

Properties of the cost function and minimal paths. Assume a function D from paths to a totally ordered domain U is given; for a path π, the value $D(\pi)$ will be referred to as the *cost* of π. The cost of the *empty path* (p) of length 0 from p to p, is denoted c_p and assumed to be known to p. We consider cost functions for which the cost of a path can be computed incrementally, which means that the cost of non-empty paths can be computed as

$$D(p_0, \ldots, p_k, p_{k+1}) = f_{p_{k+1}p_k}(D(p_0, \ldots, p_k)).$$

Here f_{pq} is a function $f_{pq} : U \longrightarrow U$, called the *edge function*, given for each edge pq. The cost of a path $\pi = (p_0, p_1, \ldots, p_{k-1}, p_k)$ can now be computed incrementally as $f_{p_kp_{k-1}}(\ldots(f_{p_1p_0}(c_{p_0})\ldots)$. Denoting as $f_\pi(x)$ the value $f_{p_kp_{k-1}}(\ldots(f_{p_1p_0}(x)\ldots)$, we can write $D(\pi)$ as $f_\pi(c_{p_0})$. The cost function (and edge functions) are presumed to satisfy the following axioms.

(1) **Monotonicity.** For every edge pq and $x \in U$, if π is a simple path to q not containing p, then $D(\pi) < x \Rightarrow f_{pq}(D(\pi)) \leq f_{pq}(x)$.
(2) **Cycle-increase.** If π is a cycle, $f_\pi(x) > x$ for all $x \in U$.

(3) **Length-increase.** There exists a number B such that for each path ρ of length B or more, each $x \in U$, and each simple path π of length $N-1$ or less, $f_\rho(x) > D(\pi)$.

The *minimal-path problem* is to compute, for each p, the minimal cost of any path ending in p, and its predecessor q in a path of minimal cost (*nil* if the minimal path is empty). The cost of a minimal path is denoted $\kappa(p)$ and the predecessor is denoted $\phi(p)$. The existence and an important property of minimal paths are expressed in the following theorem.

Theorem 17.24

(1) *For every p there exists a simple path $\pi(p)$ ending in p, such that all paths ending in p have a cost of at least $D(\pi(p))$.*
(2) *There exists a spanning forest (called a minimal-path forest) such that for every p the (unique) path from a root to p has the cost $D(\pi(p))$.*

Proof. For (1): as there are only finitely many simple paths ending in p, we can choose $\pi(p)$ as a minimal simple path ending in p. By the choice of $\pi(p)$, all simple paths ending in p then have a cost of at least $D(\pi(p))$. The cost of a path containing a cycle is (by cycle-increase and monotonicity) higher than the cost of the simple path obtained by removing cycles, hence also bounded by $D(\pi(p))$. This shows that $\pi(p)$ witnesses the first part of the theorem.

For (2): as minimal paths need not be unique, the second part of the theorem requires some care; the graph obtained by taking all edges occurring in minimal paths need not be a forest. Like the optimal sink trees considered in Theorem 4.2, the minimal-path forest is not unique, but it is possible to construct such a forest, as follows.

Choose for every node p a father $\phi(p)$ as follows. If the empty path (p) is optimal, $\phi(p) = nil$, otherwise $\phi(p)$ is a neighbor q such that a minimal path to p with penultimate node q exists. Consider the edges $p, \phi(p)$ for which $\phi(p) \neq nil$.

By monotonicity, for every p with $\phi(p) \neq nil$, $\kappa(p) \geq f_{p\phi(p)}(\kappa(\phi(p)))$. Consequently, if C is a cycle (from p to p) in the formed graph, then $\kappa(p) \geq f_C(\kappa(p))$, which contradicts the cycle-increase property, hence the chosen edges define a forest.

Induction from the roots shows the desired property of this forest. □

Observe that the length-increase property is not needed in the proof; it is only necessary to show that erroneous information in the initial configura-

var K_p : D ;
L_p : $Neigh_p \cup \{nil\}$;

$\mathbf{C}_p$: $K_p := \min(c_p, \min\{f_{pq}(K_q) : q \in Neigh_p\})$;
if $K_p = c_p$ **then** $L_p := nil$
else $L_p := q$ s.t. $K_p = f_{pq}(K_q)$

Algorithm 17.6 THE UPDATE ALGORITHM FOR MINIMAL PATHS.

tion is eventually removed in a stabilizing algorithm for computing minimal paths.

The required postcondition of an algorithm for the minimal-path problem can now be stated:

$$\psi \equiv (\forall p : K_p = \kappa(p) \wedge L_p = \phi(p)).$$

A classical (non-stabilizing) algorithm for establishing ψ consists of a single operation on each edge, called *pushing* the edge:

$\mathbf{P}_{pq}$: $\{ f_{pq}(K_q) < K_p \}$
$K_p := f_{pq}(K_q)$; $L_p := q$

The variable K_p is initialized to c_p. The assertion $K_p \geq \kappa(p)$ can be shown to be invariant. To show liveness we must assume that pushing is fair w.r.t. every edge, that is, in an infinite execution each edge is pushed infinitely often. That $K_p \leq \kappa(p)$ is established is then shown by induction of the nodes in a minimal path leading to p; the reader may compare the Chandy–Misra algorithm for minimal-path computation (Algorithm 4.7).

The Update algorithm. In order to establish ψ in a stabilizing way the information about existing paths may flow through the graph by pushing it through edges. However, it is also necessary to remove from the system erroneous information present in the initial configuration; that is, values of K_p for which no corresponding path to p exists. This is done by moving this information via longer and longer paths such that, by the length-increase property, the erroneous information will eventually be rejected in favor of information concerning an existing path.

The Update algorithm consists of a single operation for each node, in which it computes the smallest cost of the empty path and paths whose cost is stored at its neighbors; see Algorithm 17.6. In the analysis of the algorithm we shall assume, as we did for the pushing algorithm, that execution is

fair w.r.t. every process; that is, in every execution each process computes infinitely often.

Theorem 17.25 *The Update algorithm stabilizes to minimal paths.*

Proof. As $\mathbf{C}_p$ is always enabled, each execution is infinite; the fairness assumption allows us to partition an execution in *rounds* as follows. Round 1 ends when every process has computed at least once. Round $i+1$ ends when every process has computed at least once after the end of round i.

It will first be shown that eventually K_p is a lower bound for $\kappa(p)$. Define $\kappa_i(p)$ as the minimal cost of a path of length i or less to p; we claim that at any time from the end of round i, $K_p \leq \kappa_{i-1}(p)$. We establish this claim by induction on i.

Case $i = 1$**:** The empty path, with cost c_p, is the only path to p of length 0, and it can be seen from the code that after the first $\mathbf{C}_p$ step, $K_p \leq c_p$.

Case $i+1$**:** Let π be a minimal path to p of length at most i; if π has length 0, $K_p \leq D(\pi)$ after every step by p, hence also after the end of round $i+1$. Assume π is a non-empty path, namely the concatenation of path ρ and edge qp; the length of ρ is at most $i-1$. By induction, $K_q \leq D(\rho)$ after the end of round i. It follows that after every step by p taken after the end of round i, $K_p \leq f_{pq}(D(\rho)) = D(\pi)$. At the end of round $i+1$, p has taken at least one such step, showing that $K_p \leq D(\pi)$ from the end of that round.

As a minimal path contains no cycles, its length is bounded by $N-1$, which implies $\kappa(p) = \kappa_{N-1}(p)$, hence $K_p \leq \kappa(p)$ from the end of round N.

To show that eventually K_p is an upper bound for $\kappa(p)$, we shall show that after i rounds the value K_p corresponds to an existing path or to initial information that has traveled via at least i hops. Denote the initial value of K_r by K_r^*. We claim that if $K_p = K$ then there exists a path π to p with $D(\pi) = K$ or there exists a path ρ from r to p such that $K = f_\rho(K_r^*)$. Moreover, if K is computed in round i, the length of ρ is at least i. The claim is established by induction on the steps in an execution.

Base case: Initially, $K_r = f_{(r)}(K_r^*)$, where (r) is a path of length 0; of course K_r may or may not correspond to an existing path.

Induction step: Consider the value K computed in a step by p. If $K = c_p$ it is the cost of an existing path (of length 0) to p, and we are done. Otherwise, $K = f_{pq}(K_q)$; by induction, K_q is (1) the cost of a path π to q, or (2) there exists a path ρ from r to q such that $K_q = f_\rho(K_r^*)$. In the first case, $\pi \cdot qp$ is an existing path with cost $f_{pq}(D(\pi)) = f_{pq}(K_q) = K$, and we are done. In the second case, $K = f_{pq}(f_\rho(K_r^*)) = f_{(\rho \cdot qp)}(K_r^*)$.

Moreover, if K is computed in round i then K_q was computed in round $i-1$ or later, which implies by induction that the length of ρ is at least $i-1$, so that the length of $\rho \cdot qp$ is at least i.

The result is now an application of the length-increase axiom; at the end of round B, $f_\rho(K_r^*)$ exceeds the cost of any existing simple path, which shows that K_p is bounded from below by the cost of some existing path to p. Consequently, at the end of round B, $K_p = \kappa(p)$ for every p. It is easily verified that the value of L_p satisfies the requirement set for $\phi(p)$, which implies that the edges (p, L_p) define a minimal-path forest. □

The length-increase property can be enforced on path cost functions not satisfying it if the network size is known. The cost of paths is modified to include the length as well; when comparing costs, a path of length N or more is always larger than a path of length $N-1$ or less. To describe this modification formally, let a cost function D be given. Define the cost function D' by $D'(\pi) = (D(\pi), |\pi|)$; the cost of an empty path for p is given by $c'_p = (c_p, 0)$ and the edge functions are given by $f'pq(C, k) = (f_{pq}(C), k+1)$. Costs of paths are compared as follows

$$(C_1, k_1) <' (C_2, k_2) \iff (k_1 < N \wedge k_2 \geq N) \vee (C_1 < C_2).$$

All simple paths are ordered by D' exactly as they are by D, which shows that monotonicity is preserved, and the modification also preserves the cycle-increase property. As paths of length at least N have a cost larger than paths of length smaller than N, the length-increase property is satisfied.

Application: routing. A straightforward application of the Update algorithm is found in the computation of routing tables; as with the Netchange algorithm, the computation is done separately per destination. Fix a destination v and define $D(\pi)$ to be the weight of π (i.e., the sum of the weights of the edges in π) if π starts in v, and ∞ otherwise. This function can be computed incrementally by setting c_p to 0 if $p = v$ and to ∞ otherwise, and by setting $f_{pq}(C) = C + \omega_{pq}$ (where ω_{pq} is the weight of edge pq).

Monotonicity follows from the additional nature of the edge functions, and cycle-increase from the assumption that cycles in the network have positive cost. To show the length-increase property, note that there are only finitely many simple cycles and let δ be the smallest weight of a simple cycle and ω be the largest weight of any edge. A path of length $N-1$ or less has weight less than $N\omega$, while a path of length $B = N(N\omega)/\delta$ contains more than $(N\omega)/\delta$ simple cycles, and has weight more than $N\omega$.

Consequently, the distance to v (and a preferred neighbor for v, which is

the first neighbor on a shortest path to v) can be computed with the Update algorithm. To build a complete routing table the algorithm is executed for each destination in parallel.

Application: depth-first search tree. It will now be shown that selecting the lexicographically smallest simple path to each process defines a depth-first search spanning tree, whose root is the smallest process in the network. Consequently, a depth-first search tree can be constructed with the Update algorithm.

Define as the cost of π the list of nodes that occur in the path, i.e., $c_p = (p)$ and $f_{pq}(\sigma) = \sigma \cdot p$. Comparison is lexicographic, but simple paths are always smaller than paths containing cycles:

$$\sigma < \tau \iff (\ \sigma \text{ is simple and } \tau \text{ is not}) \ \vee \ (\sigma <_L \tau).$$

Monotonicity is implied by the properties of lexicographic order, and cycle-increase holds because paths containing cycles are larger than simple paths. Length-increase holds because paths of length N or more contain cycles, which makes them larger than simple paths. Consequently, the Update algorithm can compute the smallest simple path to every node. It remains to analyze the properties of the resulting minimal-path forest.

First, the forest consists of a single tree, rooted at the smallest node; call this smallest node r. Indeed, for every node p there exists a simple path from r to p; a simple path from r is smaller than a path from any other node to p, hence the minimal path to p starts in r.

Second, for neighbors p and q, either $\kappa_p \lhd q$ or $\kappa_q \lhd p$ ($\lhd$ denotes the prefix relation). Indeed, consider $\kappa(p)$ and $\kappa(q)$ and assume, w.l.o.g., that $\kappa(p) < \kappa(q)$; as the minimal paths are simple, $\kappa(p) <_L \kappa(q)$ follows. It is also implied that q does not occur in $\kappa(p)$; otherwise, its prefix up to q is a path to q, and lexicographically smaller than $\kappa(q)$, which contradicts the definition of $\kappa(q)$. But then, $\kappa(p) \cdot q$ is the cost of a *simple* path to q, which implies that $\kappa(q) \leq_L \kappa(p) \cdot q$. So $\kappa(p) <_L \kappa(q) \leq_L \kappa(p) \cdot q$, which implies that $\kappa(p) \lhd \kappa(q)$, or, equivalently, p is an ancestor of q in the minimal-path tree.

These two properties, namely

(1) that the forest consists of a single tree, and
(2) that for neighbors p and q, either $\kappa_p \lhd q$ or $\kappa_q \lhd p$,

imply that the minimal-path forest computed by the Update algorithm is a depth-first search tree. This application of the Update algorithm was proposed by Herman [Her91], who also extended the algorithm in such a

way as to compute separating nodes of the network (nodes whose removal disconnects the network).

17.3.3 Conclusion and Discussion

In this section we have studied two strategies that are useful in the design of stabilizing algorithms. Collateral composition can be used to design algorithms consisting of two (or more) temporally sequential phases, and the Update algorithm can be used to solve any problem that can be formulated as finding a minimal path to every process.

There exist methods for automatically transforming arbitrary algorithms into stabilizing ones (satisfying the same specification). To achieve this goal, Katz and Perry [KP93] have proposed the following mechanism. Concurrently with the execution of a given classical algorithm P the system repeatedly computes a snapshot of the global state of P and verifies whether the constructed configuration is a reachable one for P. If this is not the case, a reset protocol will re-initialize the system to an initial configuration of P. However, implementation of this idea faces many technical problems, owing to the need for stabilizing algorithms for snapshot computation and reset.

Exercises to Chapter 17

Section 17.1

Exercise 17.1 *Prove Lemma 17.4.*

Exercise 17.2 *Prove that Dijkstra's token ring reaches a legitimate configuration in* $O(N^2)$ *steps. Shorten the analysis by giving a single norm function, quadratically bounded in* N*, that decreases with* every *step of the algorithm.*

Exercise 17.3 *Prove that Dijkstra's token ring stabilizes already if* $K \geq N$ *(rather than* $K > N$ *as assumed in the text). (See [Hoe99].)*

Section 17.2

Exercise 17.4 *Modify the ring-orientation algorithm (Algorithm 17.1) so that it will terminate after establishing its postcondition.*

Project 17.5 *Design a stabilizing algorithm to orient a torus, preferably, so that the algorithm terminates.*

Exercise 17.6 *Show that the maximal-matching algorithm can be implemented* uniformly *in the link-register read-all model.*

Exercise 17.7 *Show that the maximal-matching algorithm stabilizes in* $N^2 + O(N)$ *steps and show that there is an* $\Omega(N^2)$ *lower bound on the (worst-case) number of steps).*

Exercise 17.8 *Design a stabilizing algorithm to construct a maximal independent set and compute the maximal number of steps before stabilization.*

Project 17.9 *Give a norm function that proves the termination of Algorithm 17.4.*

Section 17.3

Project 17.10 *Prove that outerplanar graphs are three-colorable. Design a stabilizing algorithm that computes a three-coloring of an outerplanar graph.*

Project 17.11 *Design a stabilizing algorithm that computes a five-coloring of a planar graph. (See [McH90] or another text on graph algorithms.)*

Exercise 17.12 *Show that the Update algorithm can be used for election and computation of a breadth-first search spanning tree by giving an appropriate path-cost function.*

Exercise 17.13 *Give a stabilizing algorithm for computing the network size.*

Exercise 17.14 *Show how to compute the depth of a tree with the Update algorithm.*