Self-stabilizing indulgent zero-degrading binary consensus

Theoretical Computer Science 989 (2024) 114387

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Self-stabilizing indulgent zero-degrading binary consensus [☆]

Oskar Lundström [a], Michel Raynal [b,c], Elad M. Schiller [a,*]

[a] Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, SE-412 96, Sweden
[b] University Rennes IRISA, CNRS, Inria, 35042 Rennes, France
[c] Department of Computing, Polytechnic University, Hong Kong

ARTICLE INFO

Keywords:
Replication systems
Fault-tolerance
Self-stabilization

ABSTRACT

Guerraoui proposed an indulgent solution for the binary consensus problem. Namely, he showed that an arbitrary behavior of the failure detector never violates safety requirements even if it compromises liveness. Consensus implementations are often used in a repeated manner. Dutta and Guerraoui proposed a zero-degrading solution, *i.e.,* during system runs in which the failure detector behaves perfectly, a node failure during one consensus instance has no impact on the performance of future instances.

Our study, which focuses on indulgent zero-degrading binary consensus, aims at the design of an even more robust communication abstraction. We do so through the lenses of *self-stabilization*—a very strong notion of fault-tolerance. In addition to node and communication failures, self-stabilizing algorithms can recover after the occurrence of *arbitrary transient faults*; these faults represent any violation of the assumptions according to which the system was designed to operate (as long as the algorithm code stays intact).

This work proposes the first, to the best of our knowledge, self-stabilizing algorithm for indulgent zero-degrading binary consensus for time-free message-passing systems prone to detectable process failures. The proposed algorithm recovers within a finite time after the occurrence of the last arbitrary transient fault. Since the proposed solution uses an Ω failure detector, we also present the first, to the best of our knowledge, self-stabilizing asynchronous Ω failure detector, which is a variation on the one by Mostéfaoui, Mourgaya, and Raynal.

1. Introduction

We propose a self-stabilizing implementation of *binary consensus* objects for time-free (aka asynchronous) message-passing systems whose nodes may fail-stop. We also show a self-stabilizing asynchronous construction of eventual leader failure detector, Ω.

1.1. Background and motivation

With the information revolution, everything became connected, *e.g.,* banking services, online reservations, e-commerce, IoTs, automated driving systems, to name a few. All of these applications are distributed, use message-passing systems, and require fault-

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.
* Corresponding author.
E-mail addresses: osklunds@student.chalmers.se (O. Lundström), michel.raynal@irisa.fr (M. Raynal), elad@chalmers.se (E.M. Schiller).

https://doi.org/10.1016/j.tcs.2024.114387
Received 30 July 2021; Received in revised form 17 November 2023; Accepted 2 January 2024

Available online 11 January 2024

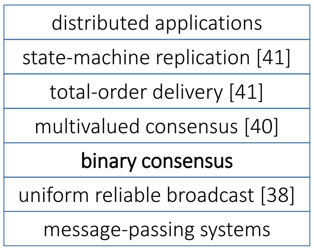
distributed applications
state-machine replication [41]
total-order delivery [41]
multivalued consensus [40]
binary consensus
uniform reliable broadcast [38]
message-passing systems

Fig. 1. The studied problem of Binary consensus (in bold) in the context of a relevant protocol suite. The layers include references for the self-stabilizing algorithms needed for implementing the system components.

tolerant implementations. Designing and verifying these systems is notoriously difficult since the system designers have to cope with their asynchronous nature and the presence of failures. The combined presence of failures and asynchrony creates uncertainties (from the perspective of individual processes) with respect to the application state. Indeed, Fischer, Lynch, and Paterson [1] showed that, in the presence of at least one (undetectable) process crash, there is no deterministic algorithm for determining the state of an asynchronous message-passing system in a way that can be validly agreed on by all non-faulty processes.

This work is motivated by applications whose state is replicated over several processes in a way that emulates a finite-state machine. In order to maintain consistent replicas, each process has to apply the same sequence of state transitions according to different sources of (user) input. To this end, one can divide the problem into two: (i) propagate the user input to all replicas, and (ii) let each replica perform the same sequence of state transitions. The former challenge can be rather simply addressed via uniform reliable broadcast [2,3], whereas the latter one is often considered to be at the problem core since it requires all processes to agree on a common value, *i.e.,* the order in which all replicas apply their state transitions. In other words, the input must be totally ordered before delivering it to the emulated automaton.

It was observed that the agreement problem of item (ii) can be generalized. Namely, the consensus problem requires each process to propose a value, and all non-faulty processes to agree on a single decision, which must be one of the proposed values. The problem of fault-tolerant consensus was studied extensively in the context of time-free message-passing systems. The goal of our work is to broaden the set of failures that such solutions can tolerate.

1.2. Problem definition and scope

Definition 1.1 states the consensus problem. When the set, V, of values that can be proposed, includes just two values, the problem is called binary consensus. Otherwise, it is called multivalued consensus. Existing solutions for multivalued consensus often use binary consensus algorithms. Fig. 1 depicts the relation to other problems in the area, which were mentioned earlier and require self-stabilizing solutions.

Definition 1.1 *(The consensus problem).* Every process p_i has to propose a value $v_i \in V$ via an invocation of the $\mathsf{propose}_i(v_i)$ operation, where V is a finite set of values. Let Alg be an algorithm that solves consensus. Alg has to satisfy *safety* (*i.e.,* validity, integrity, and agreement) and *liveness* (*i.e.,* termination) requirements.

- **Validity.** Suppose that v is decided. Then, $\mathsf{propose}(v)$ was invoked by some process.
- **Integrity.** Suppose a process decides. It does so at most once.
- **Agreement.** No two processes decide different values.
- **Termination.** All non-faulty processes decide.

As mentioned earlier, consensus cannot be solved in asynchronous message-passing systems that are prone to failures, as weak as even the crash of a single process [1]. Unreliable failure detectors [4] are often used to circumvent such impossibilities. For a given failure detector class, Guerraoui [5] proposed an indulgent solution, namely, he showed that an arbitrary behavior of the failure detector never violates safety requirements even if it compromises liveness. Consensus implementations are often used in a repeated manner. Dutta and Guerraoui [6] proposed a zero-degrading solution, *i.e.,* during system runs in which the failure detector behaves perfectly, a failure during one consensus instance has no impact on the performance of future instances. We study solutions for indulgent zero-degrading binary consensus.

1.3. Fault model

We study a time-free message-passing system that has no guarantees on the communication delay and the algorithm cannot explicitly access the local clock. Our fault model includes (i) detectable fail-stop failures of processes, and (ii) communication failures, such as packet omission, duplication, and reordering.

In addition to the failures captured in our model, we also aim to recover from *arbitrary transient faults*, *i.e.,* any temporary violation of assumptions according to which the system and network were designed to operate, *e.g.,* the corruption of control variables, such as the program counter, packet payload, and indices, *e.g.,* sequence numbers, which are responsible for the correct operation of the studied system, as well as operational assumptions, such as that at least a majority of nodes never fail. Since the occurrence of these failures can be arbitrarily combined, we assume that these transient faults can alter the system state in unpredictable ways. In particular, when modeling the system, we assume that these violations bring the system to an arbitrary state from which a *self-stabilizing algorithm* should recover the system.

1.4. *Repeated consensus in the context of self-stabilizing systems*

We consider the case of repeated invocation of the Binary consensus task. The proposed solution is suitable for the (rather common) case in which the $(k+1)$-th invocation can only start after the completion of the k-th instance, as in the case of total order broadcast [7]. Since the studied environment is time-free (asynchronous), there is a need to guarantee the completion of the k-th instance even in the presence of transient faults. We clarify that, due to transient faults, the safety guarantees might be violated with respect to the first instance (that is running after the last transient fault). However, liveness must hold even during the period in which the system recovers from the last transient fault since otherwise, the system might block indefinitely. Moreover, all requirements in Definition 1.1 must hold starting from the second invocation.

1.5. *Related work*

The celebrated Paxos algorithm [8] circumvents the impossibility by Fischer, Lynch, and Paterson [1]. Paxos assumes that failed processes can be detected by unreliable failure detectors [4]. These detectors can eventually notify the algorithm about the set of computers that were recently up and connected. However, there is no bound on the time that it takes the algorithm to receive a correct version of this notification. It is worth mentioning that Paxos has inspired many veins of research, *e.g.,* see [9] and references therein. We, however, follow the family of abstractions by Raynal [2] due to its clear presentation that is easy to grasp.

1.5.1. *Non-self-stabilizing solutions*

The Ω class includes eventual leader failure detectors. Chandra, Hadzilacos, and Toueg [10] defined this class and showed that it is the weakest for solving consensus in asynchronous message-passing systems while assuming that at most a minority of the nodes may fail. In this work, we study the Ω failure detector by Mostéfaoui, Mourgaya, and Raynal [11]. They proposed a solution that does not rely on synchrony assumptions. For example, they allow communication delays to always increase. Their algorithm is based on a query-response mechanism. Their correctness proof assumes that the exchanged of query/response messages obey a defined pattern, which we borrow (and specify Assumption 3.1). We note the existence of a computationally equivalent Ω failure detector by Aguilera et al. [12,13], which explicitly accesses timers. Our study focuses on [11] since it is asynchronous.

Guerraoui [5] presented the design criterion of indulgence. Guerraoui and Lynch [14] studied this criterion formally. In the context of this work, we consider consensus algorithms that are indulgence towards their failure detectors. By [6], it is required that even if this failure detector is totally unreliable and does never provide useful information (say, since no synchrony requirements were ever satisfied), the safety properties of consensus (validity and agreement) are preserved. Raynal [15,16] generalized the indulgence criterion and designed indulgent Ω-based consensus algorithms.

Dutta and Guerraoui [6] introduced the zero-degradation criterion. They define the criterion for consensus algorithms. They say that the algorithm is zero degrading if, and only if, the number of communication rounds needed to achieve a decision is the same in every stable run (irrespective of the identity or the number of the initially crashed processes). They define a stable run as such that if all failures occurred before the start of the run, then the output of the failure detector does not change during the run. We refine their definition in the context of the studied problem, fault model, and design criteria, see Section 3.1.3.

The studied algorithm is by Guerraoui and Raynal [15] who presented an indulgent zero-degrading consensus algorithm for message-passing systems in which the majority of the nodes never fail, and Ω-failure detectors are available. We have selected this algorithm due to its clear presentation and the fact that it matches the “two rounds” lower bound by Keidar and Rajsbaum [17].

Hurfin et al. [18] showed that zero-degradation can be combined with the versatile use of a family of failure detectors for improving the efficiency of round-based consensus algorithms. Wu et al. [19] presented the notion “look-ahead”, which is a technique for addressing challenges related to the fact that, due to asynchrony, at any given time, different processes can be at different stages of the computation. Thus, some processes might receive information related to prospective communication rounds while waiting for the completion of their current communication round. This can lead to the processing of an unbounded amount of stale information. The “look-ahead” technique by Wu et al. offer a significant latency reduction since it avoids waiting for the completion of (some parts of) communication rounds in the presence of stale information (but still, due to safety requirements, has to go through all communication rounds). The “look-ahead” technique used in this paper resembles the one by Wu et al. Unlike the one by Wu et al., the proposed solution does not use dedicated messages when invoking the “look-ahead” technique. As a bibliographical note, we would like to mention that the earlier version of this work [20] did not consider a “look-ahead” technique that can offer the benefits that Wu et al. have.

1.5.2. *Self-stabilizing solutions*

We follow the design criteria of self-stabilization, which Dijkstra [21] proposed. A detailed pretension of self-stabilization was provided by Dolev [22] and Altisen et al. [23].

Blanchard et al. [24] have a self-stabilizing failure detector for partially synchronous systems. They mention the class P of perfect failure detectors. Indeed, there is a self-stabilizing asynchronous failure detector for class P by Beauquier and Kekkonen-Moneta [25] and a self-stabilizing synchronous Ω failure detector by Delporte-Gallet, Devismes, and Fauconnier [26]. We present the first, to the best of our knowledge, asynchronous Ω failure detector. Hutle and Widder [27] present an impossibility result that connects fault detection, self-stabilization, and time-freedom as well as link capacity and local memory bounds. They explain how randomization can circumvent this impossibility for an eventually perfect failure detector [28]. Biely et al. [29] connect between classes of deterministic failure detectors, self-stabilization, and synchrony assumptions. We follow the assumption made by Mostéfaoui, Mourgaya, and Raynal [11] regarding communication patterns, which is another way to circumvent such impossibilities.

The consensus problem was not extensively studied in the context of self-stabilization. The notable exceptions are by Dolev et al. [30] and Blanchard et al. [24], which presented the first practically-self-stabilizing solutions for share-memory and message-passing systems, respectively. We note that practically-self-stabilizing systems, as defined by Alon et al. [31] and clarified by Salem and Schiller [32], do not satisfy Dijkstra's requirements, *i.e.*, practically-self-stabilizing systems do not guarantee recovery within a finite time after the occurrence of transient faults. Moreover, the message size of Blanchard et al. is polynomial in the number of processes, whereas ours is a constant (that depends on the number of bits it takes to represent a process identifier). The origin of the design criteria of practically-self-stabilizing systems can be traced back to Dolev et al. [30], who provided a practically-self-stabilizing solution for the consensus problem in shared memory systems, whereas we study message-passing systems. It is worth mentioning that the work of Blanchard et al. has led to the work of Dolev et al. [33], which considers a practically-self-stabilizing emulation of state-machine replication. *I.e.*, it has the same task of the state-machine replication in Fig. 1. However, Dolev et al.'s solution is based on virtual synchrony by Birman and Joseph [34], where the one in Fig. 1 considers consensus. We also note that earlier self-stabilizing algorithms for state-machine replications were based on group communication systems and assumed execution fairness [35–37].

There are self-stabilizing algorithms that are the result of transformations of non-self-stabilizing yet solutions, such as for atomic snapshots [38], uniform reliable broadcast [39], set-constraint delivery broadcast [40], multivalued consensus [41], total-order reliable broadcast, and state-machine replication [7], as well as coded atomic storage [42]. Our work focuses on the considerations and techniques for transferring the studied algorithm, see Section 9. A stronger design criterion is self-stabilizing Byzantine fault tolerance systems [43,44]. Recent advances in that area includes reliable broadcast [45,46], (Binary and multivalued) consensus [47, 48] consensus object recycling [49], state-machine replication [50,51], and topology discovery [52].

1.6. Our contribution

We present a fundamental module for dependable distributed systems: a self-stabilizing algorithm for indulgent zero-degrading binary consensus for time-free message-passing systems that are prone to detectable node fail-stop failures.

The design criteria of indulgence and zero-degradation are essential for facilitating efficient distributed replication systems and self-stabilization is imperative for significantly advancing the fault-tolerance degree of future replication systems. Indulgence means that the safety properties, *e.g.,* agreement, are never compromised even if the underlying model assumptions are never satisfied. Zero-degrading means that the process failures that occurred before the algorithm starts have no impact on its efficiency, which depends only on the failure pattern that occurs during the system run. To the best of our knowledge, we are the first to provide a solution for binary consensus that is indulgent, zero-degrading, and can tolerate a fault model as broad as ours. Our model includes detectable fail-stop failures, communication failures, such as packet omission, duplication, and reordering as well as arbitrary transient faults. The latter can model any temporary violation of the assumptions according to which the system was designed to operate (as long as the algorithm code stays intact).

In the absence of transient faults, our solution achieves consensus within an optimal number of communication rounds (and without requiring execution fairness). After the occurrence of any finite number of arbitrary transient faults, the system recovers within a finite time. As in Guerraoui and Raynal [15], each node uses a bounded amount of memory. Moreover, the communication costs of our algorithm are similar to the non-self-stabilizing one by Guerraoui and Raynal [15]. The main difference is in the period after a node has decided. Then, it has to broadcast the decided value. At that time, the non-self-stabilizing solution in [15] terminates whereas our self-stabilizing solution repeats the broadcast until the consensus object is deactivated by the invoking algorithm. This is along the lines of a well-known impossibility [22, Chapter 2.3] stating that self-stabilizing systems cannot terminate. Also, it is easy to trade the broadcast repetition rate with the speed of recovery from transient faults.

We also propose the first, to the best of our knowledge, self-stabilizing asynchronous Ω failure detector, which is a variation on Mostéfaoui, Mourgaya, and Raynal [11]. We show transient fault recovery within the time it takes all non-crashed processes to exchange messages among themselves. The use of local memory and communication costs are asymptotically the same as the one of [11]. The key difference is that we deal with the "counting to infinity" scenario, which transient faults can introduce. The proposed self-stabilizing solution uses a trade-off parameter, δ, that can balance between the solution's vulnerability (to elect a crashed node as a leader even in the absence of transient faults) and the time it takes to elect a non-faulty leader (after the occurrence of the last transient fault). Note that $\delta \in \mathbb{Z}^+$ is a predefined constant.

1.7. Organization

We state our system settings in Section 2. Section 3 presents our self-stabilizing asynchronous Ω failure detector. Section 4 includes a brief overview of the earlier algorithm that has led to the proposed solution. An unbounded self-stabilizing algorithm is proposed in Section 5. The correctness proof appears in Section 6. Our bounded self-stabilizing solution appears in Section 7 and correctness proof follows in Section 8. We draw our concludes in Section 9.

2. System settings

We consider a time-free message-passing system that has no guarantees on the communication delay. Moreover, there is no notion of global (or universal) clocks and the algorithm cannot explicitly access the local clock (or timeout mechanisms). The system consists of a set, $\mathcal{P}$, of n fail-prone nodes (or processes) with unique identifiers. Any pair of nodes $p_i, p_j \in \mathcal{P}$ have access to a bidirectional communication channel, $channel_{j,i}$, that, at any time, has at most $\mathsf{channelCapacity} \in \mathbb{N}$ packets on transit from p_j to p_i (this assumption is due to a well-known impossibility [22, Chapter 3.2]).

In the *interleaving model* [22], the node's program is a sequence of *(atomic) steps*. Each step starts with an internal computation and finishes with a single communication operation, *i.e.,* a message $send$ or $receive$. The *state*, s_i, of node $p_i \in \mathcal{P}$ includes all of p_i's variables and $channel_{j,i}$. The term *system state* (or configuration) refers to the tuple $c = (s_1, s_2, \cdots, s_n)$. We define an *execution (or run)* $R = c[0], a[0], c[1], a[1], \ldots$ as an alternating sequence of system states $c[x]$ and steps $a[x]$, such that each $c[x+1]$, except for the starting one, $c[0]$, is obtained from $c[x]$ via the execution of $a[x]$.

We clarify that the above interleaving model [22,23] allows reasoning about the system state in a way that is somewhat simpler than event-based models [53,54].

2.1. Task specification

The set of *legal executions* (LE) refers to all the executions in which the requirements of task T hold. In this work, T_{binCon} denotes the task of binary consensus, which Definition 1.1 specifies, and LE_{binCon} denotes the set of executions in which the system fulfills T_{binCon}'s requirements.

The proposed solution is tailored for the protocol suite presented in Fig. 1. Thus, we consider multivalued consensus objects that use an array, $BC[]$, of n binary consensus objects, such as the one by [2, Chapter 17] and [41], where $n = |\mathcal{P}|$ is the number of nodes in the system. Also, we organize these multivalued consensus objects in an array, $CS[]$, of M elements, where $M \in \mathbb{Z}^+$ is a predefined constant. In [55], we explain how to use a global restart procedure to bound the number of sequence numbers used for the multivalued objects.

We clarify the importance, the context of self-stabilizing systems, of using only bounded variables. Specifically, the implementation of algorithms that were designed to use unbounded counters in systems that have only bounded memory may violate correctness invariants in the presence of transient faults. The reason is that, in practice, computer systems use only a bounded amount of memory. Thus, a single transient fault can cause an integer counter to overflow. From that point in time, basic correctness invariants can be lost. For example, counters are used for ordering events and once they wrap to zero, the ordering logic is broken.

Also, we specify how the decided value is retrieved. We clarify that it can be either via the returned value of the $\mathsf{propose}()$ operation (as in the studied algorithm [15]) or via the returned value of the $\mathsf{result}()$ operation (as in the proposed solution).

2.2. The fault model and self-stabilization

A failure occurrence is a step that the environment takes rather than the algorithm.

2.2.1. Benign failures

When the occurrence of a failure cannot cause the system execution to lose legality, *i.e.,* to leave LE, we refer to that failure as a benign one.

Node failures. The studied consensus algorithms are prone to *fail-stop failures*, in which nodes stop taking steps. We assume that at most $t < |P|/2$ node may fail and that unreliable failure detectors [4] can detect these failures.

The studied failure detector constructions consider *crash failures* that are undetectable without explicit access to timers (as in Aguilera et al. [13]) or other synchrony assumptions (as Mostéfaoui, Mourgaya, and Raynal [11] do, see our review Section 3.3).

Communication failures. We consider solutions for the problem of consensus that are oriented towards time-free message-passing systems and thus they are oblivious to the time in which the packets arrive and depart. We assume that any message can reside in a communication channel only for a finite period. Also, the communication channels are prone to packet failures, such as omission, duplication, reordering. However, if p_i sends a message infinitely often to p_j, node p_j receives that message infinitely often. We refer to the latter as the *fair communication* assumption.

2.2.2. Arbitrary transient faults

We consider any temporary violation of the assumptions according to which the system was designed to operate. We refer to these violations and deviations as *arbitrary transient faults* and assume that they can corrupt the system state arbitrarily (while keeping the program code intact). The occurrence of an arbitrary transient fault is rare. Thus, our model assumes that the last arbitrary transient fault occurs before the system execution starts [22]. Also, it leaves the system to start in an arbitrary state.

2.2.3. Dijkstra's self-stabilization criterion

An algorithm is *self-stabilizing* with respect to the task of LE, if every (unbounded) execution R of the algorithm reaches within a finite period a suffix $R_{legal} \in LE$ that is legal. Namely, Dijkstra [21] requires $\forall R : \exists R' : R = R' \circ R_{legal} \wedge R_{legal} \in LE \wedge |R'| \in \mathbb{Z}^+$, where the operator $\circ$ denotes that $R = R' \circ R''$ concatenates R' with R''.

2.2.4. Complexity measures

The complexity measure of self-stabilizing systems, called *stabilization time*, is the time it takes the system to recover after the occurrence of the last transient fault. Next, we provide the assumptions needed for defining this period.

We *do not assume* execution fairness in the absence of transient faults. We say that a system execution is *fair* when every step that is applicable infinitely often is executed infinitely often and fair communication is kept. After the occurrence of the last transient fault, we assume the system execution is *temporarily* fair until the system reaches a legal execution, as in Georgiou et al. [38].

As mentioned, our solution for the problem of consensus does not consider the notion of time. We do, however, use the term complete iteration (of the do-forever loop). It is well-known that self-stabilizing algorithms cannot terminate and stop sending messages [22, Chapter 2.3]. Moreover, their code includes a do-forever loop. Let N_i be the set of nodes with whom p_i completes a message round trip infinitely often in R. Suppose that immediately after the state c_{begin}, node p_i takes a step that includes the execution of the first line of the do-forever loop, and immediately after the system state c_{end}, it holds that: (i) p_i has completed the iteration of c_{begin} and (ii) every request message m (and its reply) that p_i has sent to any non-failing node $p_j \in \mathcal{P}$ during the iteration (of the do-forever loop) has completed its round trip. In this case, we say that p_i's complete iteration starts at c_{begin} and ends at c_{end}.

3. Ω -class failure detectors

We study a non-self-stabilizing construction of an Ω failure detector (Section 3.1.2) and propose its self-stabilizing variant.

3.1. Unreliable failure detectors

Chandra and Toueg [4] introduced the concepts of failure patterns and unreliable failure detectors. Chandra, Hadzilacos, and Toueg [10] proposed the class Ω of eventual leader failure detectors. It is known to be the weakest failure detector class to solve consensus. A pedagogical presentation of these failure detectors is given in Raynal [2].

3.1.1. Failure patterns

Any execution $R := (c[0], a[0], c[1], a[1], \ldots)$ can have any number of failures during its run. R's failure pattern is a function $F : \mathbb{Z}^+ \to 2^{\mathcal{P}}$, where $\mathbb{Z}^+$ refers to an index of a system state in R, which in some sense represents (progress over) time, and $2^{\mathcal{P}}$ is the power-set of $\mathcal{P}$, which represents the set of failing nodes in a given system state. $F(\tau)$ denotes the set of failing nodes in system state $c_\tau \in R$. Since we consider fail-stop failures, $F(\tau) \subseteq F(\tau + 1)$ holds for any $\tau \in \mathbb{Z}^+$. Denote by $\mathit{Faulty}(F) \subseteq \mathcal{P}$ the set of nodes that eventually fail-stop in the (unbounded) execution R, which has the failure pattern F. Moreover, $\mathit{Correct}(F) = \mathcal{P} \setminus \mathit{Faulty}(F)$. For brevity, we sometimes notate these sets as *Correct* and *Faulty*.

3.1.2. Eventual leader failure detectors

This class allows $p_i \in \mathcal{P}$ to access a read-only local variable $leader_i$, such that $\{leader_i\}_{1 \le i \le n}$ satisfy the Ω-validity and Ω-eventual leadership requirements, where $leader_i^\tau$ denotes $leader_i$'s value in system state $c_\tau \in R$ of system execution R. Ω-validity requires that $\forall i : \forall \tau : leader_i^\tau$ contains a node identity. Ω-eventual leadership requires that $\exists \ell \in \mathit{Correct}(F), \exists c_\tau \in R : \forall \tau' \ge \tau : \forall i \in \mathit{Correct}(F) : leader_i^{\tau'} = \ell$. These requirements imply that a unique and non-faulty leader is eventually elected, however, they do not specify when this occurs and how many leaders might co-exist during an arbitrarily long (yet finite) anarchy period. Moreover, no node can detect the ending of this period of anarchy.

3.1.3. Stable runs

We refine the definition of stable runs (by Dutta and Guerraoui [6]) in order to clarify its application in the context of the studied problem, fault model, and design criteria. Let R be an execution in which the Ω failure detector always (and at all non-faulty nodes) outputs the same non-faulty node, p_ℓ. In this case, we say that R is a stable run with respect to the studied problem, fault model, and design criteria. We note that stable runs are relevant only in the context of legal executions since during the period of recovery from transient faults there are no guarantees regarding the output correctness of the failure detector.

3.2. Non-self-stabilizing Ω failure detector

Algorithm 1 presents the non-self-stabilizing Ω failure detector by Mostéfaoui, Mourgaya, and Raynal [11]; the boxed code lines are irrelevant to [11] since we use them to present our self-stabilizing solution. Note that, in addition to the assumptions described in Section 2.1, Mostéfaoui, Mourgaya, and Raynal make the following operational assumptions (Section 3.3), which are asynchronous by nature.

Algorithm 1: An Ω construction; code for p_i. (Only the self-stabilizing version includes the boxed code lines.)

```
1  local constants, variables, and their initialization: (Initialization is optional in the context of self-stabilization.)
2  $\boxed{\text{const } \delta}$ ;                          /* max gap between the extrema of count values */
3  $r := 0$ ;                                                /* current round number */
4  $recFrom := \mathcal{P}$ ;          /* set of identities of the nodes that replied to the most recent query */
5  $count[0..n\text{-}1] := [0,\ldots,0]$ ;                  /* the number of times each node was suspected */
6  operation leader() {let $(-, x) := \min\{(count[k], k)\}_{p_k \in \mathcal{P}}$; return $(x)$}
7  $\boxed{\text{macro counts()} := \{count[k] : p_k \in \mathcal{P}\}}$;
8  $\boxed{\text{macro check()} := \text{if } \max \text{counts()} - \min \text{counts()} > \delta \text{ then foreach } p_k \in \mathcal{P} \text{ do}}$ $\boxed{count[k] \leftarrow \max\{count[k], (\max \text{counts()} - \delta)\}}$;
9  do-forever begin
10     $r \leftarrow r + 1$;
11     repeat
12         foreach $j \neq i$ do send ALIVE$(r, count)$ to $p_j$;
13     until RESPONSE$(rJ = r,$ $\boxed{-,}$ $recFromJ)$ received from $(n - t)$ nodes;
14     let $prevRecFrom := \cup$ sets of $recFromJ$ received in line 13;
15     foreach $j \notin prevRecFrom$ $\boxed{: count[j] < \delta + \min \text{counts()}}$ do $count[j] \leftarrow count[j] + 1$;
16     $recFrom \leftarrow$ {nodes from which $p_i$ has received RESPONSE$(rJ, \bullet)$ in line 13};
17     $\boxed{\text{check()}}$;
18 upon ALIVE$(rJ, countJ)$ arrival from $p_j$ begin
19     foreach $p_k \in \mathcal{P}$ do $count[k] \leftarrow \max(count[k], countJ[k])$;
20     $\boxed{\text{check()}}$;
21     send RESPONSE$(rJ,$ $\boxed{\text{count,}}$ $recFrom)$ to $p_j$
22 $\boxed{\text{upon RESPONSE}(rJ, countJ, recFromJ) \text{ arrival from } p_j}$ begin
23     $\boxed{\text{foreach } p_k \in \mathcal{P} \text{ do } count[k] \leftarrow \max(count[k], countJ[k]);}$
24     $\boxed{\text{check()}}$;
```

3.3. Operational assumptions

Algorithm 1 follows Assumption 3.1. Let us observe Algorithm 1's communication pattern of queries and responses. Node p_i broadcasts ALIVE() queries repeatedly until the arrival of the corresponding RESPONSE() messages from $(n - t)$ receivers (the maximum number of messages from distinct nodes it can wait for without risking being blocked forever). For the sake of a simple presentation (and without loss of generality), it is assumed that nodes always receive their own responses. We refer to the first $(n-t)$ replies to a query that p_i receives as the winning responses. The others are referred to as the losing since, after a crash, the failing nodes cannot reply.

Assumption 3.1 *(Eventual Message Pattern).* In any execution R, there is a system state $c_\tau \in R$, a non-faulty $p_i \in \mathcal{P}$, and a set Q of $(t+1)$ nodes, such that, after c_τ, each node $p_j \in Q$ always receives a winning response from p_i to each of its queries (until p_j possibly crashes). (Note that the time until the system reaches c_τ, the identity of p_i, and the set Q need not be explicitly known by the nodes.)

3.4. Variables

The local state includes r_i (initialized to 0) and is used for indexing p_i's current round of alive queries and responses. Moreover, the array $count[]$ counts the number of suspicions, *e.g.*, $count_i[j]$ counts from zero the number of times p_i suspected p_j. Also, the $recFrom$ set (initialized to $\mathcal{P}$) has the identities of the nodes that responded to the most recent alive query. When the application layer accesses the variable $leader$, Algorithm 1 returns the identity of the least suspected node (line 6).

3.5. Algorithm description

Algorithm 1 repeatedly executes a do-forever loop (lines 9 to 16), which broadcasts ALIVE$(r, \bullet)$ messages (line 12) and collects their replies, which are the RESPONSE$(rJ, \bullet)$ messages (lines 13 and 21). In this message exchange, every $p_i \in \mathcal{P}$ uses a round number, r_i, to facilitate asynchronous rounds without any coordination linking the rounds of different nodes. Moreover, there is no limit on the number of steps any node takes to complete an asynchronous round.

3.5.1. The do-forever loop

Each iteration of the loop includes actions (1) to (3).

1. Node p_i broadcasts ALIVE$(r_i, count_i)$ queries (line 12), and waits for $(n - t)$ replies, *i.e.*, RESPONSE$(rJ, recFromJ)$ messages from $p_j \in \mathcal{P}$ (line 21), where r_i and rJ are matching round numbers. Moreover, $count_i$ is an array in which, as said before, $count_i[k]$ stores the number of times p_i suspected $p_k \in \mathcal{P}$. Also, $recFromJ$ is a set of the identities of the responders to p_j's most recent query (lines 16 and 21).

2. By aggregating into $prevRecFrom_i$ all the arriving $recFromJ$ sets (line 13), p_i can estimate that any $p_j : j \notin prevRecFrom_i$ that does not appear in any of these sets is faulty. Thus, p_i increment $count_i[j]$ (line 15).
3. The iteration of the do-forever loop ends with a local update to p_i's $recFrom_i$ (line 16).

3.5.2. Processing of arriving queries

Upon $\mathsf{ALIVE}(rJ, countJ)$ arrival from p_j, node p_i merges the arriving data with its own (line 19), and replies with $\mathsf{RESPONSE}(rJ, recFrom_i)$ (line 21). This reply includes p_j's round number, rJ, which is not linked to p_i's round number, r_i.

3.6. Self-stabilizing Ω failure detector

When including the boxed code lines, Algorithm 1 presents an unbounded self-stabilizing variation of the Ω failure detector in [11]. (As mentioned before, Section 5 in [38] explains how to convert such unbounded self-stabilizing algorithms to bounded ones.) Note that in [11], all non-crashed nodes converge to a constant value that is known to all correct nodes whereas the counters of crashed nodes increase forever, see claims C2 and C3 of Theorem 97 in [2]. Thus, the proposed algorithm includes the following differences from [11].

3.6.1. Sharing all $count[]$'s values

Algorithm 1 makes sure that any non-failing node does not “hide” a value that is too high in $count_i[x]$ without sharing it with all correct nodes. In the context of self-stabilization, such a value can appear due to a transient fault. To that end, Algorithm 1 includes the field $count$ in the $\mathsf{RESPONSE}()$ message (line 21) so that the receiver can merge the arriving data with the local one (line 23).

3.6.2. No counting to infinity

Algorithm 1 also avoids “counting to infinity” since, in the context of self-stabilization, a transient fault can set the counters to arbitrary values. For example, suppose that the counter values that non-faulty nodes associate with all crashed nodes are zero. Then suppose that the counters associated with any non-faulty node store an extremely high value, say, $M = 2^{62}$. We must not require the system to count from zero to M before it is guaranteed that a non-crashed leader is elected, because it would take more than 146 years to do (if we assume the rate of one nanosecond per communication round).

Thus, the proposed solution limits the difference between the extrema counter values in any local array to be less than δ, where $\delta \in \mathbb{Z}^+$ is a predefined constant. One can view δ as a trade-off parameter between the solution vulnerability (to elect a crashed node as a leader even in the absence of transient faults) and the time it takes to elect a non-faulty leader (after the occurrence of the last transient fault and after the system has reached c_τ that satisfies the eventual message pattern assumption, see Assumption 3.1). *I.e.*, on the one hand, if the value of δ is set too low, nodes that sporadically slow down might be elected, while on the other hand, for very large values of δ, say, $M = 2^{62}$, the time it takes to recover after the occurrence of the last transient faults can be extremely long.

3.7. Correctness of the self-stabilizing version of Algorithm 1

Definitions 3.1 and 3.2 are needed for showing that Algorithm 1 brings the system to a legal execution (Theorem 3.2).

Definition 3.1 *(Algorithm 1's consistent state).* Suppose $\max \mathsf{counts}_i() - \min \mathsf{counts}_i() \leq \delta$ holds in $c \in R$ for any non-failing $p_i \in \mathcal{P}$. In this case, we say c is consistent.

Definition 3.2 *(Complete execution of Algorithm 1).* Let R be an execution of Algorithm 1. Let $c, c'' \in R = R' \circ R''$ denote the starting system states of R, and respectively, R'', for some suffix R'' of R. We say that message m is *completely delivered* in c if the communication channels do not include $\mathsf{ALIVE}(r, \bullet)$ nor $\mathsf{RESPONSE}(r, \bullet)$ messages (in system state c). Suppose that $R = R' \circ R''$ has a suffix R'', such that for any $\mathsf{ALIVE}(r, \bullet)$ or $\mathsf{RESPONSE}(r, \bullet)$ message m that is not completely delivered in c'', it holds that m does not appear in c. In this case, we say that R'' is complete with respect to R.

Theorem 3.2 (Convergence). *(i) Once every non-failing node completes at least one iteration of the do-forever loop (lines 9 to 17) or receives at least one message (lines 18 or 22), the system reaches a consistent state. (ii) Every infinite execution $R = R' \circ R''$ of Algorithm 1 reaches within a finite number of steps a suffix R'', such that R'' is complete with respect to R (Definition 3.2).*

Proof of Theorem 3.2. Lines 17, 20, and 24 imply invariant (i). Invariant (ii) is implied by the assumption that any message that appears in the starting system state can reside in a communication channel only for a finite time (Section 2.2.1). $\square_{\textit{Theorem 3.2}}$

Theorem 3.3 (Closure). *Let R be an execution of Algorithm 1 that starts in a consistent system state. Suppose that R has an eventual message pattern (Assumption 3.1). Algorithm 1 demonstrates in R a construction of the eventual leader failure detector, Ω.*

Proof of Theorem 3.3. In the context of Algorithm 1, we say that $p_i \in \mathcal{P}$ inhibits the increment of $count_i[x]$ in line 15 when $x \notin prevRecFrom$ holds but $count_i[x] < \delta + \min \mathsf{counts}_i()$ does not. Suppose that, for a given $p_x \in \mathcal{P}$, there is $p_k \in \mathcal{P}$ that, during

R, either increments $count_k[x]$ in line 15 or in inhibits such increments for a finite number of times. In this case, we say that $count_k[x]$ is eventually constant. In all other cases, we say that $count_k[x]$ is unbounded. Given a failure pattern $F()$, we define: $PL = \{x : \exists i \in Correct(F) : count_i[x]$ is eventually constant$\}$, and $\forall i \in Correct(F) : PL_i = \{x : count_i[x]$ is eventually constant$\}$, where the set of node identities, PL, stands for "potential leaders". These definitions imply $\forall i \in Correct(F) : PL_i \subseteq PL$.

The rest of the proof shows that correct nodes share identical sets of potential leaders (PL), which are non-empty (Lemma 3.4), and include only correct nodes (Lemmas 3.5 and 3.6). The proof ends by showing that the nodes in PL can only be suspected, *i.e.,* their counters are incremented (or inhibited from being incremented), a finite number of times, and this number is eventually the same at each non-faulty node (Lemma 3.7). Thus, all correct nodes eventually elect the node that was suspected for the smallest number of times.

Lemma 3.4. $PL \neq \emptyset$

Proof of Lemma 3.4. Since Assumption 3.1 holds, there must be a system state $c_{\tau_0} \in R$, a node p_i and a set Q of $(t+1)$ nodes for which at any state after c_{τ_0}, any non-failing node $p_j \in Q$ receives winning responses from p_i for any of p_j's queries. Due to the assumptions that $|Q| > t$ and that there are at most t faulty nodes, Q includes at least one non-faulty node. Let $\tau \geq \tau_0$ be a time after which no more nodes fail.

Node $p_k \in \mathcal{P} : k \in Correct(F)$ does not stop sending its query (line 12) until it receives RESPONSE() messages from $(n-t)$ nodes. Moreover, after c_τ, at least $(t+1)$ nodes get winning responses from p_i. Therefore, the system eventually reaches a state $c_{\tau_k} \in R : \tau \leq \tau_k$ after which $i \in prevRecFrom_k$ holds (line 21). Thus, p_k stops incrementing (or inhibiting the increment) of $count_k[i]$ at line 15.

Since p_k is any correct nodes, the system reaches $c_{\max\{\tau_x\}_{x \in Correct(F)}} \in R$ eventually, it holds that $\forall x, y \in Correct(F) : count_x[i] = count_y[i] = M_i \in \mathbb{Z}^+$. *I.e.*, due to the repeated exchange of messages between any pair of non-faulty nodes, these nodes have a constant value for $count[i]$ eventually. $\square_{\text{Lemma 3.4}}$

Lemma 3.5. $PL \subseteq Correct(F)$.

Proof of Lemma 3.5. We show that for every $x \notin Correct(F)$, it holds that $p_i : i \in Correct(F)$ increments (or inhibits the increment) of $count_i[x]$ for an unbounded number of times during R. The rest of the proof is implied by the fact that non-faulty nodes never stop exchanging messages among themselves and merge the arriving information upon message arrival (lines 19 and 23).

Suppose that all the faulty nodes have crashed (and their RESPONSE() messages have been received) before $c_\tau \in R$. Let p_i and p_j be two non-faulty nodes, and p_x a faulty one. We observe invariants (i) to (iv), which imply the proof. (i) Since p_x cannot respond to any of p_j's queries, it holds that $x \notin rF_j$, where rF_j is the value of $recFrom_j$ (which is assigned in line 16) in any system state, c'_τ, that appears in R after c_τ. (ii) Due to invariant (i), it holds that $x \notin pRF_j$, where pRF_j is the value of $prevRecFrom_i$ (which is assigned in line 13) in any system state, c''_τ, that appears in R after c'_τ. (iii) Due to invariant (ii), after c''_τ, every execution of line 15 implies an increment of $count_i[x]$ (or the inhibition of an increment). (iv) Since p_i sends an unbounded number of queries, invariant (iii) implies that $count_i[x]$ is incremented (or inhibited from incrementing) for an unbounded number of times during R. $\square_{\text{Lemma 3.5}}$

Lemma 3.6. $(i \in Correct(F)) \Rightarrow (PL_i = PL)$

Proof of Lemma 3.6. Recall that $PL_i \subseteq PL$ (by the definitions of PL and PL_i). Thus, $PL_i \subseteq Correct(F)$ (Lemma 3.5). Also, by showing that $PL \subseteq PL_i$ we have the proof.

Let us assume that $k \in PL$ and show that $k \in PL_i$. That is, we assume that there are $k, j \in Correct(F)$ for which the constant M_k is the highest value stored in $count_j[k]$ throughout R. In order to prove that $k \in PL_i$, we need to show that $count_i[k]$ convergences to a constant eventually. Since $count_j[k] \leq M_k$ throughout R, the repeated exchange of ALIVE() and RESPONSE() messages between the correct nodes p_i and p_j (line 12, lines 18 to 19, and lines 22 to 23), implies $count_i[k] \leq M_k$ throughout R. $\square_{\text{Lemma 3.6}}$

Lemma 3.7. *Let $i, j \in Correct(F)$. Suppose that R has a suffix R'' during which $count_i[k] = M_k$ always hold, where M_k is a constant. Then, $count_j[k] = M_k$ also holds throughout R''.*

Proof of Lemma 3.7. This is due to the repeated exchange of ALIVE() and RESPONSE() messages between p_i and p_j (lines 12, 18 to 19 and 22 to 23). $\square_{\text{Lemma 3.7}}$ $\square_{\text{Theorem 3.3}}$

4. Background: non-self-stabilizing binary consensus

Algorithm 2 is a non-self-stabilizing Ω-based binary consensus algorithm that is indulging and zero-degrading. For the sake of a simpler presentation of the correctness proofs, Algorithm 2's line enumeration continues the one of Algorithm 1.

Algorithm 2: Guerraoui-Raynal [15]'s non-self-stabilizing indulgent zero-degrading binary consensus; code for p_i.

```
25 local variables and their initialization:
26 $r := 0$;                                                    /* current round number */
27 est[0..1] := [$\bot,\bot$];                 /* decision estimates at the start of phases 0 & 1 */
28 operation propose($v$) begin
29     (est[], $r$) ← ([$v,\bot$], 0);                                  /* $\bot$ denotes no value */
30     while True do
31         $r \leftarrow r+1$;
           /* Phase 0: select a value with the help of $\Omega$                                  */
32         let $myLeader$ := leader;                                                  /* read $\Omega$ */
33         repeat
34             broadcast PHASE($0, r$, est[0], $myLeader$)
35         until [PHASE($0,r,\bullet$) received from $n-t$ nodes] $\wedge$ [PHASE($0,r,\bullet$) received from $p_{myLeader}$ $\vee$ $myLeader \neq$ leader];
36         if [PHASE($0,r,\bullet,\ell$) received from more than $n/2$ nodes] $\wedge$ [PHASE($0,r,v,\bullet$) received from $p_\ell$] then est[1] ← $v$ else est[1] ← $\bot$;
           /* Here ((est$_i$[1] $\neq \bot$) $\wedge$ (est$_j$[1] $\neq \bot$)) $\implies$ (est$_i$[1] = est$_j$[1] = $v$)         */
           /* Phase 1: try to decide on an est[1] value                                              */
37         repeat broadcast PHASE($1, r$, est[1]) until [PHASE($1,r,\bullet$) received from $n-t$ nodes];
38         switch {$rec$ : PHASE($1,r,rec$) has been received} do
39             case {$v$} do {broadcast DECIDE($v$); return($v$)};
40             case {$v,\bot$} do est[0] ← $v$;
41             case {$\bot$} do continue;

42 upon DECIDE($v$) arrival from $p_j$ do {broadcast DECIDE($v$); return($v$);}
```

4.1. Algorithm structure

Algorithm 2 proceeds in asynchronous rounds that include two phases. The algorithm aims to have, by the end of phase zero, the same value, which is named the estimated value. This selection is done by a leader, whose election is facilitated by the Ω failure detector. Next, during phase one, the algorithm tests the success of phase zero. The challenging scenario occurs when, due to asynchrony, not all nodes run the same round simultaneously. Therefore, the test considers the agreement on the round number, the leader identity, and the proposed value. Moreover, just before deciding on any value, say v, the deciding node broadcasts a DECIDE(v) message. Upon DECIDE(v) arrival, the receiver repeats the broadcast of the arriving message before deciding. Algorithm 2 executes the "decide" action by returning with v from propose(v)'s invocation. This technique of 'broadcast repetition' basically lets Algorithm 2 invoke a reliable broadcast of the decided value.

4.1.1. The system behavior during phase zero

The objective of phase zero of round r is to let all nodes store in est[1] the same value. Once that happens, a decision can be taken during phase one of round r. As we are about to explain, that objective is guaranteed to be achieved once a single leader is elected.

Phase zero aims at the provision of the safety property, *i.e.,* no two different decisions are made during Ω's anarchy period (in which there is no single non-faulty elected leader). To that end, phase zero makes sure that the *quasi-agreement* property always holds before anyone enters phase one of round r, where $((\mathrm{est}_i[1] \neq \bot) \wedge (\mathrm{est}_j[1] \neq \bot)) \implies (\mathrm{est}_i[1] = \mathrm{est}_j[1] = v)$ (line 36) is the property definition. This means that, if $\mathrm{est}_i[1] = v \neq \bot$ holds, from the perspective of p_i, it can decide v. Moreover, if $\mathrm{est}_i[1] = \bot$ holds, then from the perspective of p_i, it is not ready to decide any value. Therefore, a system state that satisfies the quasi-agreement property allows the individual nodes to decide during phase one on the same value (when $\mathrm{est}_i[1] = \mathrm{est}_j[1] = v$) or defer the decision to the next round (when $\mathrm{est}_i[1] = \bot$). In order to satisfy the quasi-agreement property by the end of phase zero, each $p_i \in \mathcal{P}$ performs actions (1) and (2), which imply Corollary 4.1.

Corollary 4.1. *The* quasi-agreement *property holds immediately before $p_i \in \mathcal{P}$ enters phase one of any round.*

Action (1): node p_i stores in $myLeader_i$ the value of $leader_i$ (line 32), which is the interface to the Ω failure detector, before broadcasting the message PHASE($0, r, \mathrm{est}_i[0], myLeader_i$) (line 34). It then waits until it hears from $n-t$ nodes on the same round (line 35). Since there are at most t failed nodes, waiting for more than $n-t$ nodes jeopardizes the system's liveness. Moreover, $t < n/2$ and thus any set of $n-t$ nodes is a majority set, which contains at least one correct node. Node p_i may stop broadcasting when it receives a PHASE($0, r, \bullet$) message from its leader, *i.e.,* $p_{myLeader_i}$, or when a new leader is elected, *i.e.,* $myLeader_i \neq leader_i$.

Action (2): after the above broadcast, p_i's assignment to $\mathrm{est}_i[1]$ (line 36) satisfies the quasi-agreement property by making sure that (i) a majority of nodes consider p_ℓ as their leader when they broadcast the PHASE($0, r, \bullet, \ell$), and (ii) p_i received PHASE($0, r, v, \bullet$) from p_ℓ. In other words, if (i) and (ii) hold, p_i can assign v to $\mathrm{est}_i[1]$, which is p_ℓ's value in $\mathrm{est}_\ell[0]$ at the start of round r. Otherwise, $\mathrm{est}_i[1]$ gets $\bot$. Due to the majority intersection property, no two majority sets can have two different unique leaders. Therefore, it cannot be that $est1_i[r] = v \neq \bot$ and $est1_j[r] = v' \neq \bot$ without having $v = v'$.

Corollary 4.1 is implied by the above two actions.

4.1.2. The system behavior during phase one

During this phase, p_i broadcasts PHASE($1, r, \mathrm{est}_i[1]$) until it hears from $n-t$ nodes. By the quasi-agreement property, $\exists v \in V$: $\forall p_j \in \mathcal{P}$: $\mathrm{est}_j[1] = \bot \vee \mathrm{est}_j[1] = v \neq \bot$ holds during round r. Thus, for the set of all received estimated values, $rec_i \in \{\{v\}, \{v, \bot\}, \{\bot\}\}$

Figure 2: Constants, variables, and operations for Algorithm 3.

```
43 constants: $initState$: the initial value (a vector) of a binary consensus object;
44 variables: $rnd[0,..,n{-}1] = [-1,\dots,-1]$ round numbers arriving from all nodes;
45 $\mathsf{phs}[0,..][0,..,n{-}1] = [[0,\dots,0],\dots]$ phase numbers (per round) arriving from all nodes;
46 $\mathsf{est}[0,..][0,..,n{-}1][0,1] = [[[\bot,\bot],\dots],\dots]$ estimated values (per round) arriving from all nodes;
47 $\mathsf{lead}[0,..][0,..,n{-}1] = [\bot,\dots,\bot]$ leader identity (per round) arriving from all nodes;
48 $\mathsf{dec}[0,..,n{-}1] = [\bot,\dots,\bot]$ decide value arriving from all nodes (if does not exist, then $\bot$);

49 operation propose($v \neq \bot$) do $\{(O, \mathsf{rnd}[i], \mathsf{est}[0][i][0]) \leftarrow (initState, 0, v)\}$
50 operation result() if $(O \neq \bot \wedge |\{p_k \in \mathcal{P} : O.\mathsf{dec}[k] \neq \bot\}| \geq t{+}1)$ then return $(O.\mathsf{dec}[i])$ else return $\bot$;
```

Algorithm 3: Unbounded self-stabilizing algorithm for indulgent zero-degrading binary consensus; code for p_i and binary object O.

```
51 Constants, variables, and operation appear in Fig. 2.
52 do-forever {foreach $O \neq \bot$ with $O's$ fields $r, \mathsf{phs}, est, \mathsf{lead}$, and dec do
53     if $(\exists_{x>\mathsf{rnd}[i]} \emptyset \neq (\{\mathsf{est}[x{+}1][i][0], \mathsf{est}[x][i][1]\} \setminus \{\bot\}))$ then $\{O \leftarrow \bot$; continue;$\}$;
54     if $\nexists p_k \in \mathcal{P} : \mathsf{dec}[k] \neq \bot$ then $(\mathsf{rnd}[i], \mathsf{lead}[\mathsf{rnd}[i]][i], \mathsf{phs}[\mathsf{rnd}[i]][i]) \leftarrow (\mathsf{rnd}[i]+1, \mathsf{leader}, 0)$;
55     else if $\mathsf{dec}[i] = \bot$ then $\mathsf{dec}[i] \leftarrow \mathsf{dec}[k]$;
56     if $(\exists_{x \in \{0,\dots,\mathsf{rnd}[i]\}} \bot \in \{\mathsf{est}[x][i][0], \mathsf{lead}[x][i]\})$ then $\{O \leftarrow \bot$; continue;$\}$;
57     repeat
58         if $\mathsf{phs}[\mathsf{rnd}[i]][i] = 0$ then
59             if $(\exists_{p_\ell \in \mathcal{P}} \mathsf{est}[\mathsf{rnd}[i]][\ell][0] \neq \bot \wedge \exists_{S \subseteq \mathcal{P}} |S| \geq n-t \wedge \forall_{p_k \in S} \mathsf{lead}[\mathsf{rnd}[i]][k] = \ell)$ then $(\mathsf{est}[\mathsf{rnd}[i]][i][1], \mathsf{phs}[\mathsf{rnd}[i]][i]) \leftarrow (\mathsf{est}[\mathsf{rnd}[i]][\ell][0], 1)$ ;
60             else if $(\exists_{p_k \in \mathcal{P}} \mathsf{rnd}[k] > \mathsf{rnd}[i] \vee (\mathsf{rnd}[k] = \mathsf{rnd}[i] \wedge \mathsf{phs}[\mathsf{rnd}[i]][k] = 1))$ then
61                 $(\mathsf{est}[\mathsf{rnd}[i]][i][1], \mathsf{phs}[\mathsf{rnd}[i]][i]) \leftarrow (\mathsf{est}[\mathsf{rnd}[k]][k][1], 1)$  /* look ahead */
62             else if $\mathsf{lead}[\mathsf{rnd}[i]][i] \neq \mathsf{leader}$ then $(\mathsf{est}[\mathsf{rnd}[i]][i][1], \mathsf{phs}[\mathsf{rnd}[i]][i]) \leftarrow (\bot, 1)$;
63         broadcast PHASE(True, $\mathsf{rnd}[i], \mathsf{phs}[\mathsf{rnd}[i]][i], \mathsf{est}[\mathsf{rnd}[i]][i], \mathsf{lead}[\mathsf{rnd}[i]][i], \mathsf{dec}[i]$);
64     until $(\exists_{p_k \in \mathcal{P}} \mathsf{dec}[k] \neq \bot) \vee (|\{p_k \in \mathcal{P} : \mathsf{rnd}[k] \geq \mathsf{rnd}[i] \wedge \mathsf{phs}[\mathsf{rnd}[i]][k] \geq 1\}| \geq n{-}t)$;
65     let $rec = \{\mathsf{est}[\mathsf{rnd}[i]][k][1] : p_k \in \mathcal{P} \wedge \mathsf{phs}[\mathsf{rnd}[i]][k] \geq 1\}$;
66     switch $rec$ do
67         case $\{v \neq \bot\}$ do $\{\mathsf{est}[\mathsf{rnd}[i]+1][0] \leftarrow v$; if $\mathsf{dec}[i] = \bot$ then $\mathsf{dec}[i] \leftarrow v\}$;
68         case $\{\bot, v \neq \bot\}$ do $\mathsf{est}[\mathsf{rnd}[i]+1][0] \leftarrow v$;
69         case $\{\bot\}$ do $\mathsf{est}[\mathsf{rnd}[i]+1][0] \leftarrow \mathsf{est}[\mathsf{rnd}[i]][0]$;

70 upon PHASE($aJ, rJ, pJ, eJ, \ell J, dJ$) arrival from $p_j$ do begin
71     if $\bot \in \{eJ[0], \ell J\}$ then return;
72     if $O = \bot$ then $(O, \mathsf{rnd}[i], \mathsf{est}[0][i][0]) \leftarrow (initState, 0, eJ[0])$;
73     $O.\mathsf{phs}[rJ][j] \leftarrow \max\{O.\mathsf{phs}[rJ][j], pJ\}$;
74     foreach $x \in \{0,1\} : O.\mathsf{est}[rJ][j][x] = \bot$ do $O.\mathsf{est}[rJ][j][x] \leftarrow eJ[x]$;
75     if $O.\mathsf{dec}[j] = \bot$ then $O.\mathsf{dec}[j] \leftarrow dJ$;
76     if $\ell J \neq \bot$ then $O.\mathsf{lead}[rJ][j] \leftarrow \ell J$;
77     if $aJ$ then send PHASE(False, $rJ, O.\mathsf{phs}[rJ][i], O.\mathsf{est}[rJ][i], O.\mathsf{lead}[rJ][i], O.\mathsf{dec}[i]$) to $p_j$;
```

(line 38) holds. For the $rec_i = \{v\}$ case, p_i broadcasts DECIDE(v) and then decide v (line 39). For the $rec_i = \{v, \bot\}$ case, p_i uses v during round $r+1$ as the new estimated value $\mathsf{est}_i[0]$ since some other node might have decided v (line 40). For the $rec_i = \{\bot\}$ case, p_i continues to round $r+1$ without modifying $\mathsf{est}_i[0]$ (line 41). Note that, at any round r, it cannot be the case that both $rec_i = \{v\}$ and $rec_j = \{\bot\}$ hold, since p_i's broadcast of DECIDE(v) implies that it had received PHASE($1, r_i, v$) from a majority of nodes. Due to the majority intersection property, there is at least one PHASE($1, r_i, v$) arrival to any $p_j \in \mathcal{P}$ that executes line 38 since it also received PHASE($1, r_i, \bullet$) messages from a majority. Thus, $rec_j = \{\bot\}$ cannot hold.

4.1.3. *The necessity of broadcasting v before deciding on it*

Algorithm 2 has to take into consideration the case in which not all nodes decide during round r. *E.g.,* a majority of nodes might decide on round r, while a minority of them continues to round $r+1$ during which it must not wait in vain to hear from a majority. By broadcasting DECIDE(v) before deciding v, Algorithm 2 allows the system to avoid such bad situations since once p_i decides, it is guaranteed that eventually, all correct nodes decide (because Guerraoui Raynal [15] assumes reliable communication channels).

5. Unbounded self-stabilizing binary consensus

In this section, we present our self-stabilizing variation on Guerraoui and Raynal [15]. Before presenting our solution, we review the challenges that one faces when transforming the solution by Guerraoui and Raynal into a self-stabilizing one. As a first transformation step, we present Algorithm 3, which uses arrays with an unbounded number of entries that grows linearly in the number of communication rounds. After demonstrating the correctness of Algorithm 3 (Section 6), we transform Algorithm 3 into a solution (Section 7) that uses only bounded size arrays and unbounded round numbers. But, using existing techniques [55], the round counters can be bounded as well.

5.1. Challenges and approaches

Wu et al. [19] observed situations in which, due to asynchrony, one node, say, p_i, runs phase zero, while another, say, p_j runs phase one of the same round as of p_i. Wu et al. also note that asynchrony can lead for p_j to execute a round number that is (much) higher than the ones of p_i. In both cases, once p_i receives information about p_j's round and phase, it can proceed to the next phase of its round since the estimated value for phase one of its round is already given via p_j's message.

Wu et al. proposed a "look-ahead" technique for helping to reduce the number of rounds required for reaching consensus. Note that the "look-ahead" technique by Wu et al., as well as the one used by the proposed self-stabilizing solution, still requires p_i to execute phase one. In other words, p_i cannot skip rounds when it sees that p_j is running a higher round number.

Our main motivation for using a "look-ahead" technique (that is different than the one by Wu et al.) is not for the sake of reducing the number of rounds but rather to avoid the need to demonstrate the absence of live-lock situations in the presence of transient-faults. *E.g.,* if p_i ignores the fact that p_j is running another round number, it might wait in vain for p_j's message. Specifically, when exactly half of the nodes execute round r and another half executes round $r' > r$, a self-stabilizing solution needs to guarantee that no live-lock situations can occur. (Note that we are not claiming that Guerraoui and Raynal [15] might run into live-lock situations in the presence of transient faults. We rather wish to simplify the proof that no live-lock can occur.)

Our "look-ahead" technique merges PHASE() messages (of both phases) and the DECIDE() messages into a single message that includes all of the fields of Algorithm 2's messages. Also, we replace the two repeat-until loops of Algorithm 2 with a single repeat-until loop and cascading if-statements that implement our "look-ahead" technique.

Another challenge that our self-stabilizing solution needs to deal with is the fact that Guerraoui and Raynal [15] assume reliable communications between any two nodes. This work does not assume reliable communications, due to reasons explained in [42, Section 2.2]. Thus, the proposed solution needs to store all sent information (in arrays that have one entry per communication round) in order to provide such reliability.

5.2. Variables

Algorithm 3 considers a single binary consensus object, which is denoted by O, which Section 2.1 details how this object can fit in the protocol suite presented in Fig. 1. The object O is composed of $\mathsf{rnd}[0,\ldots,n{-}1]$, $\mathsf{phs}[0,\ldots][0,\ldots,n{-}1]$, $\mathsf{est}[0,\ldots][0,\ldots,n{-}1][0,1]$, $\mathsf{lead}[0,\ldots][0,\ldots,n{-}1]$, and $\mathsf{dec}[0,\ldots,n{-}1]$, where $\mathsf{phs}[0,\ldots][]$, $\mathsf{est}[0,\ldots][]$, and $\mathsf{lead}[0,\ldots][]$ are unbounded since they grow linearly in the number of communication rounds.

- The array $\mathsf{rnd}[]$ (round numbers) is initialized by $[-1,\ldots,-1]$. For a given node $p_i \in \mathcal{P}$, the entry $\mathsf{rnd}_i[i]$ holds p_i's round number. For node $p_j \in \mathcal{P} \setminus \{p_i\}$, the entry $\mathsf{rnd}_i[j]$ holds the highest round number that p_i ever received from p_j, where -1 represents the case in which no value was ever received.
- The array $\mathsf{phs}[][]$ (phase numbers) is initialized to an unbounded sequence of the element $[0,\ldots,0]$. For a given round number r, the entry $\mathsf{phs}_i[r][i]$ holds p_i's phase. For the case of $r' \leq \mathsf{rnd}_i[j]$, the entry $\mathsf{phs}_i[r'][j]$ holds the highest phase that p_i ever received from p_j for round r'.
- The array $\mathsf{est}[][][]$ (estimated values) is initialized to an unbounded sequence of the element $[[\bot,\bot],\ldots,[\bot,\bot]]$. For a given round number r and phase $x \in \{0,1\}$, the entry $\mathsf{est}_i[r][i][x]$ holds p_i's estimate for phase x. For the case of $r' \leq \mathsf{rnd}_i[j] \wedge x' \leq \mathsf{phs}_i[r'][j]$, the entry $\mathsf{est}_i[r'][j][x']$ holds p_j's (last received) estimate for round r' and phase x'.
- The array $\mathsf{lead}[][]$ (leaders) is initialized to an unbounded sequence of the element $[\bot,\ldots,\bot]$. For a given round number r, the entry $\mathsf{lead}_i[r][i]$ holds p_i's leader for round r. For the case of $r' \leq \mathsf{rnd}_i[j]$, the entry $\mathsf{lead}_i[r'][j]$ holds p_j's leader for round r'.
- The array $\mathsf{dec}[]$ (decided values) is initialized by $[\bot,\ldots,\bot]$. For a given node $p_i \in \mathcal{P}$, the entry $\mathsf{dec}_i[i]$ holds p_i's decided value. For node $p_j \in \mathcal{P} \setminus \{p_i\}$ and the case of $\mathsf{dec}_i[j] \neq \bot$, the entry $\mathsf{dec}_i[j]$ holds p_j's decided value.

5.3. Message structure

Algorithm 3 uses the PHASE($ackNeed, round, phase, estimate, leader, decide$) message (lines 63 and 70), where the field *ackNeed* (Boolean) indicates whether a reply message is needed (line 77), the field *round* holds the sender's round number, the field *phase* holds the sender's phase, the field *estimate* holds the current estimates for the respective round (for both phases), the field *leader* holds the sender's leader (for the current round), and the field *decide* holds the sender's decide value, if there is such value (otherwise, $\bot$).

5.4. Interface operations

The operation propose(v) (Section 2.1) allows the invoking node to propose value v (line 49). The operation result() returns eventually the decided value, if such a decision occurred (line 50). Otherwise, $\bot$ is returned.

5.5. The do-forever loop (lines 52 to 69)

The loop starts by asserting consistency and managing a new round (Section 5.5.1), iterative communication via a repeat-until loop (Section 5.5.2), and ending the round by attempting to reach a decision (Section 5.5.3).

5.5.1. *Starting a new round (lines 53 to 56)*

Line 53 asserts that est[][][] does not store information about rounds that p_i has yet to reach. Since this can only occur due to a transient fault, the presence of such stale information results in the deactivation of object O_i (line 54).

Line 54 considers the case in which no node, $p_k \in \mathcal{P}$, has informed p_i about any decision. In this case, line 54 starts a new round by incrementing p_i's round number, queries the Ω failure detector, and asserts that the phase is zero. Otherwise, p_i can simply adopt p_k's decided value (line 55).

Line 56 is another consistency assertion in which p_i makes sure that there is no round number (earlier than the current one) that does not store an estimated value or a leader. As in line 54, the failure of this assertion results in the deactivation of object O_i.

5.5.2. *The repeat-until loop (line 58 to 64)*

This loop embeds the processing of phase zero messages in lines 58 to 62. Line 59 tests whether a PHASE() message was received from at least $n-t$ nodes such that their *leader* field points to the same leader, p_ℓ, and p_i also received a PHASE() message from p_ℓ. In this case, line 62 lets p_i use p_ℓ's estimated value (in phase zero) as its phase one estimated value.

Line 60 tests the "look-ahead" conditions. That is, whether p_i is aware of node p_k that has a higher round or phase numbers. In this case, line 60 lets p_i use p_k's estimated value (in phase one) as its phase one estimated value.

Line 62 deals with the case in which the Ω failure detector changes its value while p_i is in phase zero. In this case, the $\perp$-value is used as p_i's phase one estimation since there is no guarantee that this round can succeed.

The repeat-until loop ends with a broadcast of the PHASE() message (line 63). It exits (line 64) when p_i discovers that a decision has already been reached or when at least a majority of nodes have sent PHASE() message for p_i's current round and phase (or a later round number).

5.5.3. *Attempting to reach a decision (line 65 to 69)*

Line 65 defines the set *rec* that includes all the phase-one estimated values (of p_i's round number) that have arrived from nodes that have reached phase one of p_i's round number (or a later round number). According to *rec*'s value, p_i attempts to decide in lines 67 to 69. As Theorem 6.13 shows, it is safe for p_i to decide when $\perp \notin rec$ (line 67). The decided value is then disseminated via lines 55, 63, and 74 until p_i received a decided value from at least t other nodes (line 50). Then, p_i is ready to return the decided value since it knows that it was received by at least one correct node. For the cases in which $\perp \in rec$, node p_i tries to see whether it can use any non-$\perp$ value from *rec* for estimating the decided value for the next round (line 68). Otherwise, the current estimation is used (line 69).

5.6. *The arrival of* PHASE() *messages*

This arrival updates (and even initializes) the local state of the binary consensus, O_i. Prior to this update, p_i runs a consistency test (line 72) and makes sure that O_i is an active object (line 72). Lines 73 to 76 update O_i according to the arriving information. The procedure ends by acknowledging the sender, if needed (line 77).

6. Algorithm 3's correctness

Theorems 6.2, 6.3 and 6.13 show the recovery (from transient faults), termination (when starting from an arbitrary system state), and respectively, the satisfaction of the task requirements (Definition 1.1). Theorem 6.2 uses Definition 6.1, which uses Condition 6.1.

Condition 6.1 *(Consistency condition for Algorithm 3).* $\forall i \in Correct : (\nexists x > \mathsf{rnd}_i[i] : \emptyset \neq (\{\mathsf{est}_i[x+1][i][0], \mathsf{est}_i[x][i][1]\} \setminus \{\perp\})) \wedge (\nexists x \in \{0, \ldots, \mathsf{rnd}_i[i]\} : \perp \in \{\mathsf{est}_i[x][i][0], \mathsf{lead}_i[x][i]\})$.

Definition 6.1. Let R be an execution of Algorithm 3, $c \in R$ be a system state, and $p_i : i \in Correct$ be a correct node. Suppose either $O_i = \perp$ holds in c or Condition 6.1 holds in c. In this case, we say that p_i is consistent in c. Suppose all non-failing nodes are consistent in c and there is no message $\mathsf{PHASE}(ack, rnd, \mathsf{phs} = p, est = e, ldr = \ell, dcs)$ in transient between two non-failing nodes such that $\perp \in \{e[0], \ell\}$. In this case, we say that c is consistent. Suppose every system state in R is consistent. In this case, we say that R is consistent.

As explained in Section 2.2.4, only Theorem 6.2 requires the execution to be fair. All the other parts of the proof of Algorithm 3, do not make this assumption, see Section 2.2.4 for details.

Theorem 6.2 *(Algorithm 3's Convergence). Let R be a fair execution of Algorithm 3. Within finite time, the system reaches a state $c \in R$ that starts a consistent execution (Definition 6.1).*

Proof of Theorem 6.2. Suppose that R's starting state is not consistent, specifically, with respect to node $p_i : i \in Correct$. In other words, the if-statement condition in line 53 or 56 holds. Within a single complete iteration of the do forever loop, every correct node p_i takes a step that includes the execution of lines 53 to 56, which assures that p_i becomes consistent.

We observe from the code of Algorithm 3 that once p_i is consistent in c, node p_i is also consistent in any state $c' \in R$ that follows c, cf. lines 49, 71, and 72 as well as the assignments to $O_i.\mathsf{phs}[r'][i]$ (lines 59 to 62) and $O_i.\mathsf{est}[r'][i][0]$ (lines 67 to 69), for some

$r' \in \mathbb{Z}^+$. Due to the above, the rest of the proof assumes, without the loss of generality, that all non-failing nodes are consistent in any state of R.

Let m be a message that in R's starting system state resides in the communication channels between any pair of correct nodes. Recall that any message that appears in the starting system state can reside in a communication channel only for a finite time (Section 2.2.1). Thus, by the definition of complete iterations, within a finite time, the system reaches a state in which m does not appear in the communication channels. Therefore, without the loss of generality, we can consider only messages, $\mathsf{PHASE}(ackNeed, round, phase = p, estimate = e, leader = \ell, decide)$, that were sent during R by (non-failing) and consistent nodes. Thus, the invariant $\bot \in \{e[0], \ell\}$ holds. $\square_{\textit{Theorem 6.2}}$

Theorem 6.3 *(Algorithm 3's Termination). Let R be a consistent execution of Algorithm 3. Suppose $\exists x \in Correct : O_x \neq \bot$ holds in the starting system state of R. Eventually, the system reaches a state, $c \in R$ after which $\forall x \in Correct : \mathsf{result}_x() \neq \bot$ always hold.*

Proof of Theorem 6.3. Claims 6.4 to 6.12 imply the proof.

Claim 6.4. *Eventually, the system reaches a state in R after which $\forall x \in Correct : O_x \neq \bot \wedge O_x.eJ[0] \neq \bot$ holds in every system state.*

Proof of Claim 6.4. Since R is consistent, the if-statement conditions in lines 53 and 56 cannot hold (Theorem 3) for any node in the system and throughout R. Therefore, no node $p_j \in \mathcal{P}$ assigns $\bot$ to O_j. The fact that R is consistent also means that $O_j \neq \bot \implies O_j.eJ[0] \neq \bot$. By the theorem assumption, $\exists y \in Correct : O_y \neq \bot$ holds in the starting system state of R. Thus, $O_y \neq \bot$ and $O_y.eJ[0] \neq \bot$ hold throughout R (Theorem 6.2). Thus, by lines 63 and 72, $\forall x \in Correct : O_x \neq \bot \wedge O_x.eJ[0] \neq \bot$ holds eventually. $\square_{\textit{Claim 6.4}}$

Claim 6.5. *Suppose either (i) $\exists i \in Correct : O_i \neq \bot \wedge O_i.\mathsf{dec}[i] \neq \bot$, or (ii) $\exists S \subseteq P : |S| \geq t+1 \wedge \forall p_i \in S : O_i \neq \bot \wedge O_i.\mathsf{dec}[i] \neq \bot$ hold in R's starting state. Eventually $\forall x \in Correct : \mathsf{result}_x() \neq \bot$ holds.*

Proof of Claim 6.5. In every iteration of the do-forever loop (lines 52 to 69), node $p_i \in P : i \in Correct$ accesses the binary consensus object O_i (line 52) since $O_i \neq \bot$. Note that $O_i.\mathsf{dec}[i] \neq \bot$ in the starting system state of R implies that $O_i.\mathsf{dec}[i] \neq \bot$ holds throughout R due to the theorem assumptions that R is consistent and the fact that Algorithm 3 never assigns $\bot$ to O_i (lines 53 and 56) or to $O_i.\mathsf{dec}[i]$. Thus, the rest of the proof assumes, without loss of generality, that $O_i.\mathsf{dec}[i] \neq \bot$ holds throughout R.

The following two cases show that $\forall i, j \in Correct : O_j.\mathsf{dec}[i] \neq \bot$ holds eventually. Note that in both cases imply that, for any $x \in Correct : \mathsf{result}_x() \neq \bot$ holds eventually, and thus, the claim is correct.

The case of $\exists i \in Correct : O_i \neq \bot \wedge O_i.\mathsf{dec}[i] \neq \bot$. Since p_i does not crash throughout R, node p_i broadcasts PHASE$(\bullet, decide = \mathsf{dec}[i])$ (line 63) infinitely often. Due to the communication fairness assumption, every correct node, $p_j \in \mathcal{P}$, receives PHASE$(\bullet, decide = \mathsf{dec}[i])$ (line 70) eventually. Thus, $O_j.\mathsf{dec}[i] \neq \bot$ holds eventually (line 75). Moreover, eventually $O_j.\mathsf{dec}[j] \neq \bot$ holds due to line 55. This means that $\forall i, j \in Correct : O_j.\mathsf{dec}[i] \neq \bot$ holds eventually (including the $i = j$ case).

The case of $\exists S \subseteq P : |S| \geq t+1 \wedge \forall p_i \in S : O_i \neq \bot \wedge O_i.\mathsf{dec}[i] \neq \bot$. By the assumption that there are at most t faulty nodes (Section 2.2.1), we know that any set of at least $t+1$ nodes, includes at least one correct node. Thus, by the case above, $\forall i, j \in Correct : O_j.\mathsf{dec}[i] \neq \bot$ holds eventually. $\square_{\textit{Claim 6.5}}$

Claim 6.6. *During consistent executions, only non-$\bot$ values can be assigned to $\mathsf{est}_i[0][0]$ (line 49) and $\mathsf{est}_i[\mathsf{rnd}_i[i]+1][0]$ (lines 67 to 69).*

Proof of Claim 6.6. The assignments in lines 49, 67 and 68 cannot assign the $\bot$ value by the definition of the parameter v. The assignment in line 69 cannot assign the $\bot$ value due to the assumption that R is a consistent execution. In detail, every any system state either $O_i = \bot$ holds or $\nexists x \in \{0, \ldots, \mathsf{rnd}_i[i]\} : \mathsf{est}_i[x][i][0] \neq \bot$ holds. $\square_{\textit{Claim 6.6}}$

Lemma 6.7. *Let $r \geq 0$ be a round number and $p_k \in \mathcal{P} : k \in Correct$ be a correct node. Suppose p_k does not decide before round r,* i.e., *$\exists c \in R : pred_k(r) := (\mathsf{result}_k() = \bot \wedge \mathsf{rnd}_k[k] = r)$ holds. Then, any correct node either decides or continues to round $r+1$ eventually. In other words, eventually after c, $pred(r)$ does not hold.*

Proof of Lemma 6.7. Without loss of generality, let us assume that $p_i : i \in Correct$ does not decide during round r, *i.e.*, $(\mathsf{result}_k() \neq \bot \wedge \mathsf{rnd}_k[k] = r)$ does not hold throughout R. Generality is not lost due to the proof of Claim 6.5 since the case in which $O_k.\mathsf{dec}[k] \neq \bot$ holds, implies that eventually $\forall p_x \in \mathcal{P} : x \in Correct : O_k.\mathsf{dec}[k] \neq \bot$ holds (Claim 6.5). In other words, the termination property holds since $\mathsf{result}_k() \neq \bot$.

Towards a contradiction, suppose that r is the smallest round in which a correct node p_i for which $pred_i(r)$ holds indefinitely. By the choice of r, no correct node can continue to execute forever in round $r' < r$, *i.e.*, $\forall r' \in \{0, \ldots, r-1\} : \forall i \in Correct : \exists c_{r',i} \in R : pred_i(r')$ does not hold in any system state the follow $c_{r',i}$. The proof is implied by Claim 6.10, which uses Claims 6.8 and 6.9.

Claim 6.8. *Eventually p_i receives $\mathsf{PHASE}(ackNeed, round = r, phase = p \geq 0, \bullet)$ from at least $(n-t)$ different nodes.*

Proof of Claim 6.8. Since $pred_i(r)$ holds indefinitely, node p_i broadcasts PHASE$(ackNeed = \mathsf{True}, round = r, phase = p \geq 0, \bullet)$ repeatedly (line 63). Thus, PHASE$(ackNeed = \mathsf{True}, round = r, phase = p \geq 0, \bullet)$ is repeatedly received from p_i by at least $(n-t)$ nodes,

$p_j : j \in Correct$ (line 70 to 77). By line 73 and 77, p_j sends PHASE($ackNeed = \mathsf{False}, round = r, phase = p \geq 0, \bullet$) repeatedly. Thus, p_i receives PHASE($ackNeed = \mathsf{False}, round = r, phase = p \geq 0, \bullet$) from at least $(n - t)$ different nodes eventually. $\square_{\text{Claim } 6.8}$

Claim 6.9. *Eventually* $\forall i \in Correct : O_i.\mathsf{phs}[\mathsf{rnd}_i[i]] = 1$ *holds.*

Proof of Claim 6.9. The proof of the claim considers the case in which $\exists i \in Correct : O_i.\mathsf{phs}[\mathsf{rnd}[i]][i] = 0$ holds in the starting system state of R, *i.e.*, the if-statement condition in line 58 holds. (Note that the case in which $\exists i \in Correct : O_i.\mathsf{phs}[\mathsf{rnd}[i]][i] = 0$ does not hold implies that the proof is done.) We prove the claim by showing that, eventually, at least one of the if-statement conditions in lines 59 to 62 holds. This demonstrates the claim since any consequent clause of lines 59 to 62 assigns 1 to $O_i.\mathsf{phs}[\mathsf{rnd}_i[i]]$.

- Suppose $O_i.\mathsf{lead}[\mathsf{rnd}_i[i]][i] \neq \mathsf{leader}_i$ holds eventually. Thus, the if-statement condition in line 62 holds eventually, which means that the proof of the argument is done. For the rest of the proof of the claim, we assume $O_i.\mathsf{lead}[\mathsf{rnd}_i[i]][i] \neq \mathsf{leader}_i$ never holds with respect to p_i and round $\mathsf{rnd}_i[i]$.
- Suppose $\exists p_k \in \mathcal{P} : prd_{i,k}$ holds eventually, where $prd_{i,k} = \mathsf{rnd}_i[k] > \mathsf{rnd}_i[i] \vee (\mathsf{rnd}_i[k] = \mathsf{rnd}[i] \wedge O_i.\mathsf{phs}[\mathsf{rnd}_i[i]][k] = 1)$. Thus, the if-statement condition in line 60 holds eventually and the proof of the claim is done. For the rest of the proof of the claim, we assume that if $O_i.\mathsf{phs}[\mathsf{rnd}_i[i]][i] = 0$ holds, $\forall p_k \in \mathcal{P} : \neg prd_{i,k}$ holds.
- By the Ω-eventual leadership property, the assumption that $pred_i(r)$ holds indefinitely and line 54, for any correct node $p_i \in Correct$, we know that $O_i.\mathsf{lead}[\mathsf{rnd}_i[i]][i]$ eventually refers to a single node, p_{leader_i}, that is a correct node. Thus, eventually, p_i receives PHASE($-, round = r, phase = p, estimate = e, \bullet$) from p_{leader_i}, such that $e[0] \neq \bot$ (Claim 6.6). Also, $|\{p_k \in \mathcal{P} : \mathsf{rnd}_i[k] = \mathsf{rnd}_i[i] \wedge O_i.\mathsf{phs}[\mathsf{rnd}[k]][k] \geq 0\}| \geq n{-}t$ holds eventually (Claim 6.8). Thus, the if-statement condition in line 59 holds eventually. Thus, the proof is done. $\square_{\text{Claim } 6.9}$

Claim 6.10. *Eventually the repeat-until condition in line 64 holds*

Proof of Claim 6.10. We show $(|\{p_k \in \mathcal{P} : \mathsf{rnd}_i[k] \geq \mathsf{rnd}_i[i] \wedge O_i.\mathsf{phs}[\mathsf{rnd}_i[i]][k] \geq 1\}| \geq n{-}t)$ eventually holds. By Claim 6.9, the phase of all correct nodes is 1 eventually. Then, Claim 6.8 says that p_i eventually receives PHASE($-, round = r, phase = 1, \bullet$) from at least $(n - t)$ different nodes. The rest of the proof is implied by line 73. We clarify that the if-statement condition in line 71 holds. This is true since $O_j.\mathsf{est}[\mathsf{rnd}_j[j]][0] \neq \bot$ (Claim 6.6), lines 54 and 56 as well as the fact that p_j executes line 63 infinitely often. $\square_{\text{Claim } 6.10}$
$\square_{\text{Lemma } 6.7}$

Lemma 6.11. *(i) Eventually there is a correct node $p_i \in \mathcal{P}$ for which the if-statement condition in line 59 holds. (ii) Eventually the if-statement conditions in lines 59 or 60 hold with respect to any correct nodes $p_i \in \mathcal{P}$ (but the if-statement condition in line 62, eventually, cannot hold).*

Proof of Lemma 6.11. Arguments (1) and (2) show invariant (i) and Argument (3) shows invariant (ii).

Argument 1: *It is sufficient to show that no node changes its round number before p_i receives a message from p_ℓ.* Recall Ω's eventual leadership (Section 3) and the fact that faulty nodes eventually crash (Section 2.2.1). Thus, using Lemma 6.7, we can state that, as long as no non-failing node ever decides, there existence of a finite round number $r' \in \mathbb{Z}^+$ from which (a) only the correct nodes are alive and connected, and (b) all correct nodes, $p_i, p_j \in \mathcal{P}$, store in $O_i.\mathsf{lead}[r'][j]$ the same correct leader, say, p_ℓ (due to lines 54, 63, and 76). Since there are at most t faulty nodes, there is always a set $S \subseteq P : |S| \geq n - t$ for which $\forall p_k \in S : \mathsf{lead}[\mathsf{rnd}[i]][k] = \ell$ holds. Thus, in order to show that the if-statement condition in line 59 holds eventually for some $r' \in \mathbb{Z}^+$, it is sufficient to show that no (correct) node changes its round number from r' to $r' + 1$ before at least one correct node $p_i \in \mathcal{P}$ receives a message $m = \text{PHASE}(ack, rnd = r', \mathsf{phs} = 0, est = e, \bullet)$ from p_ℓ, *i.e.*, $O_i.\mathsf{est}[r'][\ell][0] \neq \bot$ Claim 6.6 and line 74) while $\mathsf{rnd}[i] = r'$ is the largest round number in the system.

Argument 2: *Showing that the if-statement condition in line 59 holds eventually.* Suppose, towards a contradiction, that no node $p_i \in \mathcal{P}$ (for which $\mathsf{rnd}_i[i] = r'$) ever receives the message m from p_ℓ before changing its round number to $r' + 1$. We know that change occurs due to Lemma 6.7. Among all the choices of p_i from $\mathcal{P}$, let us assume, without the loss of generality, that p_i is the first to change its round number. We note that such changes cannot be due to lines 60 and 62. This is because p_i is the first to increment its round number (*i.e.*, the condition in line 60 cannot hold). Also, all nodes share the same leader, which does not change (*i.e.*, the condition in line 62 cannot hold).

The proof has reached the needed contradiction since p_i must exist and increment its round number (Lemma 6.7). The only way to do so is by executing line 59. *I.e.*, the system has to reach a state in which the if-statement condition in line 59 holds.

Argument 3: *Invariant (ii) holds eventually.* We note that the if-statement condition in line 62 cannot hold since all nodes share the same leader. The if-statement condition in line 59 can hold due to Argument 2. Once that happens, say, with respect to node $p_i \in \mathcal{P}$, other nodes, say $p_j \in \mathcal{P}$, might receive message PHASE($ackNeed = -, round = r, phase = p, \bullet$) from p_i, such that $r > \mathsf{rnd}_j[j]$ or $r = \mathsf{rnd}_j[j] \wedge p > O_j.\mathsf{phs}[r]$, *i.e.*, the if-statement condition in line 60 holds. $\square_{\text{Lemma } 6.11}$

Claim 6.12. *Eventually* $\forall x \in Correct : O_x.\mathsf{dec} \neq \bot$.

Proof of Claim 6.12. Recall that $\exists p_k \in \mathcal{P} : O_i \neq \perp \wedge O_i.\mathsf{dec}[k] \neq \perp$ implies a decision due to invariant (i) of Claim 6.5. Assume, toward a contradiction, that no non-failing node $p_k \in \mathcal{P}$ ever decides with respect to any value $v \in V$ (and that $\nexists p_k \in \mathcal{P} : O_i \neq \perp \wedge O_i.\mathsf{dec}[k] \neq \perp$). We demonstrate a contradiction by showing that eventually there is a round number, r', and a correct node, $p_i \in \mathcal{P}$, for which $(\mathsf{result}_i() \neq \perp \wedge \mathsf{rnd}_i[i] = r')$ holds since the switch-case condition in line 67 holds, see the following Argument (1).

Argument 1: *Eventually the system reaches a state for which it holds that* $\exists rec_i = \{O_i.\mathsf{est}[r'][k][1] : p_k \in \mathcal{P} \wedge O_i.\mathsf{phs}[r'][k] \geq 1\} = \{v : v \neq \perp\}$ *(cf. lines 65 and 67) holds.* By Claim 6.10, the repeat-until condition in line 64 holds for any round eventually. By Invariant (ii) of Lemma 6.11, eventually, there is a round number r', in which the if-statement conditions in lines 59 or 60 hold (but not the if-statement conditions in lines 62) with respect to any non-failing node $p_j \in \mathcal{P}$. Thus, during round r' and before any non-failing node $p_j \in \mathcal{P}$ tests whether repeat-until condition in line 64 holds, p_j assigns a non-$\perp$ value to $\mathsf{est}_i[r'][i][1]$ (due to lines 59 and 60 as well as Claim 6.6). By Claims 6.6 and 6.8, $rec_i = \{v : v \neq \perp\}$ holds for any non-failing node $p_i \in \mathcal{P}$ during r' (before the repeat-until condition in line 64 is tested and rec_i is defined in line 65). $\square_{Claim\ 6.12}$ $\square_{Theorem\ 6.3}$

We say that the system state c is *well-initialized* if $\forall p_i \in Correct : O_i := \perp$ holds and no communication channel between two non-failing nodes includes PHASE() messages. Note that a well-initialized system state is also a consistent one (Definition 6.1). Suppose that during execution R, there is a correct node $p_i \in \mathcal{P}$ that invokes $\mathsf{propose}_i()$, and any node $p_j \in \mathcal{P}$ that invokes $\mathsf{propose}_j()$ does so exactly once. In this case, we say that R includes a *complete invocation* of binary consensus. Theorem 6.13 shows that Algorithm 3 satisfies the requirements of Definition 1.1 during legal executions that start from a well-initialized system state and have a complete invocation of Algorithm 3.

Theorem 6.13. *Let R be a well-initialized consistent execution of Algorithm 3 that has a complete invocation of binary consensus,* $\mathsf{propose}()$. *The system demonstrates in R a construction of a binary consensus object (Definition 1.1).*

Proof of Theorem 6.3. Termination holds due to Theorem 6.3.

Integrity. Recall that $p_i \in \mathcal{P}$ decides when $\mathsf{result}_i() \neq \perp$ holds. We show that once $\mathsf{result}_i() \neq \perp$ holds, $\mathsf{result}_i()$ cannot change. By line 50, for $\mathsf{result}_i() \neq \perp$ to hold (i) $O_i \neq \perp$, (ii) $(|\{p_k \in \mathcal{P} : O_i.\mathsf{dec}[k] \neq \perp\}| \geq t{+}1)$, and (iii) $O_i.\mathsf{dec}[i] \neq \perp$ need to hold as well. By the code of Algorithm 3, we observe that $\perp$ is never assigned to O_i during consistent executions due to lines 53 and 56 as well as Definition 6.1 and Theorem 6.2. This covers case (i) above. For cases (ii) and (iii), we note that p_i can assign a non-$\perp$ value to $O_i.\mathsf{dec}[k] : p_k \in \mathcal{P}$ at most once, cf. lines 55, 67, and 75. Specifically, once $x = |\{p_k \in \mathcal{P} : O.\mathsf{dec}[k] \neq \perp\}|$ holds in system state $c' \in R$, it must be that $x \leq |\{p_k \in \mathcal{P} : O.\mathsf{dec}[k] \neq \perp\}|$ in any system state that follows c'.

Validity. The variable $O_i.\mathsf{dec}[i]$ can only be assigned with a non-$\perp$ value (Claim 6.6). Thus, when p_i receives a $\mathsf{PHASE}(\bullet, decide = v)$ message, line 75 never assigns to $O_i.\mathsf{dec}$ a $\perp$-value. That is, p_i decides on a non-$\perp$ value that comes from $\mathsf{est}[1]$ of some O_j, which in turn comes from $\mathsf{est}[0]$ of some entry O_x, where $p_j, p_x \in \mathcal{P}$. Since R is a well-initialized and consistent execution, $\mathsf{est}[0]$ can contain only proposed values that Algorithm 3 assigns in line 49. Moreover, $\mathsf{est}[1]$ can contain only values that Algorithm 3 copied from $\mathsf{est}[0]$ in lines 59 and 75. Thus, the validity property holds.

Agreement. Claim 6.14 implies agreement since it shows that only a single value can be decided in consistent executions.

Claim 6.14. *Let r be the smallest round during which any $p_i \in \mathcal{P}$ broadcasts* $\mathsf{PHASE}(-, round = r, -, estimate = [v, -], \bullet)$ *(line 63) and then the switch-case condition in line 67 holds. Suppose that $p_j \in \mathcal{P}$ also broadcasts* $\mathsf{PHASE}(-, round = r, -, estimate = [v', -], \bullet)$ *and then the switch-case condition in line 67 holds. (i) It is true that $v' = v$ holds. Let v'' be the local estimate $O_x.\mathsf{est}[0]$ of any $p_x \in \mathcal{P}$ that proceeds to round $r+1$. (ii) It holds that $v = v''$.*

Proof of Claim 6.14. *Invariant (i).* By the code of Algorithm 3 and Claim 6.8, p_i receives during r the message $m = \mathsf{PHASE}(-, round = r, phase = p \geq 0, estimate = [v, -], \bullet)$ from at least $(n - t)$ different nodes. Moreover, p_j has received during round r the message $m' = \mathsf{PHASE}(-, round = r, phase = p \geq 0, estimate = [v', -], \bullet)$ from at least $n - t$ different nodes. During consistent executions, $p_x \in \mathcal{P}$ can only transmit (and perhaps retransmit) one kind of $\mathsf{PHASE}(-, round = r, phase = p \geq 0, estimate = [v, -], \bullet)$ message per round r. Due to the property of majority intersection, p_i and p_j receive during round r the same message $\mathsf{PHASE}(-, round = r, phase = p \geq 0, estimate = [w, -], \bullet)$ from some $p_x \in \mathcal{P}$. Since both p_i and p_j executes line 67 during round r, it must be the case that $w = v = v'$.

Invariant (ii). Suppose that some correct node $p_i \in \mathcal{P}$ assigns v to $O_i.\mathsf{dec}[i]$ during a round r (line 67). Also, $p_j \in \mathcal{P}$ continues to round $r + 1$ (without evaluating to true the switch-case condition in line 67). We have to prove that $O_j.\mathsf{est}[r{+}1][0] = v$ when p_j starts round $r{+}1$.

Since p_i decided v during round r, lines 64 and 67 implies that there were at least $(n - t)$ nodes that have sent $\mathsf{PHASE}(-, round = r, phase \geq 1, estimate = [v, -], \bullet)$ to p_i during round r. By the fact that $n - t > n/2$ and the majority intersection property, we know that p_j also had to receive during round r at least one of these $\mathsf{PHASE}(-, round = r, phase \geq 1, estimate = [v, -], \bullet)$ messages. Also, it follows from the quasi-agreement property (Corollary 4.1) that p_j receives both v and $\perp$ (and no other value) in the phase 1 of round r, *i.e.*, $rec_j = \{v, \perp\}$, because $rec_j = \{v\}$ cannot hold since this implies that p_j decides v during r. Thus, p_j assigns v to $O_j.\mathsf{est}[r{+}1][0]$ before the next iteration of the do-forever loop (lines 9 to 16), which starts round $r{+}1$. $\square_{Claim\ 6.14}$ $\square_{Theorem\ 6.3}$

Figure 3: Constants, variables, operation and macros for Algorithm 3.

```
**constants:** $M > 2$ number of rounds until first memory recycling; $initState$ is a vector that holds the initial value of object $O$;
**variables:** $\mathsf{rnd}[0,..,n-1]$, $\mathsf{phs}[0,..,M-1][0,..,n-1]$, $\mathsf{est}[0,..,M-1][0,..,n-1]$, $\mathsf{lead}[0,..,M-1][0,..,n-1]$, and $\mathsf{dec}[0,..,n-1]$ are as defined by Algorithm 3;
**operation** $\mathsf{propose}(v \neq \bot)$ **do** $\{(O, rnd[i], \mathsf{est}[0][i][0]) \leftarrow (initState, 0, v)\}$
**operation** $\mathsf{result}()$ **do** {**if** $(O \neq \bot \wedge |\{p_k \in \mathcal{P} : O.\mathsf{dec}[k] \neq \bot\}| \geq t+1)$ **then return** $O.\mathsf{dec}[i]$ **else return** $\bot$};
**macro** $\mathsf{r}()$ **do return** $\max\{rnd[k] < 0 : k \in \mathsf{trusted}\}$;
**macro** $\mathsf{x}()$ **do return** $\mathsf{r}() \bmod M$;
**macro** $\mathsf{gc}()$ **do return** $\max\{0, \min\{rnd[k] < 0 : k \in \mathsf{trusted}\}, (rnd[i]-(M-2))\}$;
```

Algorithm 4: Bounded self-stabilizing algorithm for indulgent zero-degrading binary consensus; code for p_i and binary object O.

```
Constants, variables, operation and macros appear in Fig. 3.
**do-forever** {**foreach** $O \neq \bot$ **with** $O's$ fields $rnd, \mathsf{phs}, est, \mathsf{lead}$, **and** $\mathsf{dec}$ **do**
    $\boxed{\textbf{if } \mathsf{rnd}[i] < 0 \textbf{ then } \{O \leftarrow \bot; \textbf{continue};\};}$
    **if** $\nexists p_k \in \mathcal{P} : \mathsf{dec}[k] \neq \bot$ **then**
        **if** $\boxed{\neg(\mathsf{r}() - \mathsf{gc}() \geq (M-2) \wedge \mathsf{r}() = \mathsf{rnd}[i])}$ **then**
            $(\mathsf{rnd}[i], \mathsf{phs}[\mathsf{x}()], \mathsf{lead}[\mathsf{x}()][i]) \leftarrow \boxed{(\max\{\mathsf{rnd}[i]+1, \mathsf{gc}()\},}\ 0, \mathsf{leader})$
    **else if** $\mathsf{dec}[i] = \bot$ **then** $\mathsf{dec}[i] \leftarrow \mathsf{dec}[k]$;
    **if** $\boxed{\neg(\exists_{y \in \{\mathsf{gc}(),\ldots,\mathsf{r}()\}} \bot \in \{\mathsf{est}[z][i][0], \mathsf{lead}[z][i]\} \wedge z = y \bmod M)}$ **then**
        $\boxed{\{O \leftarrow \bot; \textbf{continue};\}}$
    **foreach** $\boxed{y \in (\{0,\ldots,M-1\} \setminus \{z \bmod M : z \in \{\mathsf{gc}(),\ldots,\mathsf{r}()\}\})}$ **do**
        $\boxed{(\mathsf{phs}[y], \mathsf{est}[y], \mathsf{lead}[y]) \leftarrow ([\bot,\ldots,\bot], [[\bot,\bot],\ldots,[\bot,\bot]], [\bot,\ldots,\bot])}$
    **repeat**
        **if** $\mathsf{phs}[\mathsf{x}()][i] = 0$ **then**
            **if** $(\exists_{p_\ell \in \mathcal{P}} \mathsf{est}[\mathsf{x}()][\ell][0] \neq \bot \wedge \exists S \subseteq \mathcal{P} : |S| \geq n-t \wedge \forall_{p_k \in S} \mathsf{lead}[\mathsf{x}()][k] = \ell)$ **then** $(\mathsf{est}[\mathsf{x}()][i][1], \mathsf{phs}[\mathsf{x}()][i]) \leftarrow (\mathsf{est}[\mathsf{x}()][\ell][0], 1)$ ;
            **else if** $(\exists_{p_k \in \mathcal{P}} \mathsf{rnd}[k] > \mathsf{rnd}[i] \vee (\mathsf{rnd}[k] = \mathsf{rnd}[i] \wedge \mathsf{phs}[\mathsf{x}()][k] = 1))$ **then**
                $(\mathsf{est}[\mathsf{x}()][i][1], \mathsf{phs}[\mathsf{x}()][i]) \leftarrow (\mathsf{est}[\mathsf{rnd}[k]][k][1], 1)$  /* a look ahead step */
            **else if** $\mathsf{lead}[\mathsf{x}()][i] \neq \mathsf{leader}$ **then** $(\mathsf{est}[\mathsf{x}()][i][1], \mathsf{phs}[\mathsf{x}()][i]) \leftarrow (\bot, 1)$;
        **broadcast** PHASE($\mathsf{True}, \mathsf{rnd}[i], \mathsf{phs}[\mathsf{x}()][i], \mathsf{est}[\mathsf{x}()][i], \mathsf{lead}[\mathsf{x}()][i], \mathsf{dec}[i]$)
    **until** $(\exists_{p_k \in \mathcal{P}} \mathsf{dec}[k] \neq \bot) \vee (|\{p_k \in \mathcal{P} : \mathsf{rnd}[k] \geq r) \wedge \mathsf{phs}[\mathsf{x}()][k] \geq 1\}| \geq n-t)$;
    **let** $rec = \{\mathsf{est}[\mathsf{x}()][k][1] : p_k \in \mathcal{P} \wedge \mathsf{phs}[\mathsf{x}()][k] \geq 1\}$;
    **switch** $rec$ **do**
        **case** $\{v \neq \bot\}$ **do** $\{\mathsf{est}[\mathsf{x}()+1 \bmod M][0] \leftarrow v;$ **if** $\mathsf{dec}[i] = \bot$ **then** $\mathsf{dec}[i] \leftarrow v\}$;
        **case** $\{\bot, v \neq \bot\}$ **do** $\mathsf{est}[\mathsf{x}()+1 \bmod M][0] \leftarrow v$;
        **case** $\{\bot\}$ **do** $\mathsf{est}[\mathsf{x}()+1 \bmod M][0] \leftarrow \mathsf{est}[\mathsf{x}()][0]$;

**upon** PHASE($aJ, rJ, pJ, eJ, \ell J, dJ$) **arrival from** $p_j$ **do begin**
    **if** $\bot \in \{eJ[0], \ell J\}$ **then return**;
    **if** $O = \bot \wedge eJ[0] \neq \bot$ **then** $(O, \mathsf{rnd}[i], \mathsf{est}[0][i][0]) \leftarrow (initState, 0, eJ[0])$;
    $\boxed{\mathsf{rnd}[rJ \bmod M][j] \leftarrow \max\{\mathsf{rnd}[rJ \bmod M][j], rJ\}}$;
    $O.\mathsf{phs}[rJ \bmod M][j] \leftarrow \max\{O.\mathsf{phs}[rJ \bmod M][j], pJ\}$;
    **foreach** $y \in \{0,1\} : O.\mathsf{est}[rJ \bmod M][j][y] = \bot$ **do** $O.\mathsf{est}[rJ \bmod M][j][y] \leftarrow eJ[y]$;
    **if** $O.\mathsf{dec}[j] = \bot$ **then** $O.\mathsf{dec}[j] \leftarrow dJ$;
    **if** $\ell J \neq \bot$ **then** $O.\mathsf{lead}[rJ \bmod M][j] \leftarrow \ell J$;
    **if** $aJ$ **then send** PHASE($\mathsf{False}, rJ, O.\mathsf{phs}[rJ \bmod M][i], O.\mathsf{est}[rJ \bmod M][i], O.\mathsf{lead}[rJ \bmod M][i], O.\mathsf{dec}$) **to** $p_j$;
```

The indulgence criterion states that even when the Ω failure detector always outputs incorrect values, the safety properties of consensus (validity and agreement) remain unviolated. In other words, if an algorithm Alg utilizes Ω and terminates while Ω is misbehaving, the output of Alg remains correct. Algorithm 2 satisfies the indulgence criterion due to the proof of Theorem 6.13. Also, zero-degradation requires the number of communication rounds until a decision is achieved to be the same in every stable run (Section 3.1.3). It is easy to see that, during legal execution that are also stable runs, Algorithm 2 satisfies the zero-degradation criterion since exactly two phases are required until a decision is achieved.

7. Bounded version of Algorithm 3

Algorithm 3 uses three unbounded arrays. These are $\mathsf{phs}[0,\ldots][0,..,n-1]$, $\mathsf{est}[0,\ldots][0,..,n-1]$, and $\mathsf{lead}[0,\ldots][0,..,n-1]$. This limits the applicability of the algorithm since real-world systems have bounded amount of memory. Therefore, we present Algorithm 4 that uses bounded versions of phs[], est[], and lead[]. To that end, Algorithm 4 uses a predefined constant $M \in \mathbb{Z}^+ : M > 2$ that allows us to redefine Algorithm 3's key variables, so that $\mathsf{phs}[0,..,M-1][0,..,n-1]$, $\mathsf{est}[0,..,M-1][0,..,n-1]$, and $\mathsf{lead}[0,..,M-1][0,..,n-1]$ are defined as bounded arrays in Algorithm 4. Recall that the unbounded counters in all of these arrays can be bounded in the way that is explained in [55].

7.1. The challenge

Recall that Algorithm 3 is an Ω-class consensus algorithm. As such, it cannot decide before a leader is elected by at least a majority of the nodes. However, there is no bound on the time that it takes to elect the leader. Therefore, there is no bound on the number of rounds that are required for reaching consensus. Specifically, when the majority of the nodes are in round $r_{\max}$, there could be a minority of nodes that are still in round $r_{\min}$, where $(r_{\max} - r_{\min})$ can be any finite number. In other words, memory-bounded variations on Algorithm 3 need to deal with the possible situation in which there are no more available entries in the arrays $\mathsf{phs}[][]$, $\mathsf{est}[][]$, and $\mathsf{lead}[][]$.

7.2. Our approach

We address the above challenge by recycling array entries that store outdated information. To that end, Algorithm 4 detects such situations (by letting each node locally monitor the latest round number received by any node in the system) and temporarily blocking until $(r_{\max} - r_{\min})$ becomes sufficiently small, *i.e.,* smaller than $M - 2$. Since we consider node failures, we choose to use a self-stabilizing class-P unreliable failure detector [25,24] so that the system will not block indefinitely. Specifically, a class-P failure detector guarantees the completeness and strong accuracy. Completeness requires that if $p_j \in \mathcal{P}$ crashes, it eventually does not appear permanently in the set $i \in Correct : \mathsf{trusted}_i$. Strong accuracy requires that no $p_j \in \mathcal{P}$ stops appearing in a set $\mathsf{trusted}_i$ before crashing.

7.3. Accessing rnd[], phs[][], est[][], and lead[][]

Algorithm 4 stores round numbers in the array $\mathsf{rnd}[]$ and in the arrays $\mathsf{phs}[][]$, $\mathsf{est}[][]$, and $\mathsf{lead}[][]$ it maintains a bounded version of the information that Algorithm 3 stores in them. For a given node $p_i \in \mathcal{P}$, the array $\mathsf{rnd}_i[]$ stores the most recent rounds numbers received from any (non-failing) node in the system. The access to (and maintains of) $\mathsf{phs}[][]$, $\mathsf{est}[][]$, and $\mathsf{lead}[][]$ is facilitated by the following macros. The macro $\mathsf{r}_i()$ returns the largest round number received by node $p_i \in \mathcal{P}$ from any node and the macro $\mathsf{x}_i(k)$ returns the module M value of $\mathsf{r}_i()$. Node p_i uses $\mathsf{x}_i(k)$ for indicating which entry in $\mathsf{phs}_i[][i]$, $\mathsf{est}_i[][i]$, and $\mathsf{lead}_i[][i]$ it should use according to its round number, which is $\mathsf{r}_i()$, cf. lines 90, 97 to 99, 101 to 103, 104, and 106 to 108.

7.4. Entry recycling for phs[][], est[][], and lead[][]

As mentioned above, no node $p_i \in \mathcal{P}$ progresses to the next round if $\mathsf{r}_i() - \mathsf{gc}_i()$ is more than $M - 2$, where $\mathsf{r}_i() - \mathsf{gc}_i()$ refers to the difference between the extrema of round numbers held in $\{\mathsf{rnd}_i[k] : k \in \mathsf{trusted}\}$ node that p_i does not suspect to be faulty. Node p_i also makes sure that its round number, $\mathsf{rnd}_i[i]$, is not smaller by more than $M - 2$ than the highest round number, $\mathsf{r}_i()$, that p_i is aware of. In other words, if a transient fault causes p_i's round value to be smaller by more than $M - 2$ than $\mathsf{r}_i()$, the recovery process would need to "catch up" by raising its round number to be not smaller than $\mathsf{r}_i() - (M - 2)$. This is done via a macro, $\mathsf{gc}_i()$ (which marks the garbage collection line), that returns the largest among the following two values: (i) the smallest round number in $\{\mathsf{rnd}_i[k] : k \in \mathsf{trusted}\}$, and (ii) $((\max\{\mathsf{rnd}_i[k] : k \in \mathsf{trusted}\}) - (M-2))$. This way, p_i can recycle any entry in $\mathsf{phs}[][]$, $\mathsf{est}[][]$, and $\mathsf{lead}[][]$ that refers to a round number smaller than $\mathsf{gc}_i()$.

7.5. Detailed description

We highlight the main differences between Algorithms 3 and 4 in boxed lines codes, cf. lines 87 to 90, 93 to 95, and 112.

Line 87 tests the consistency of the local state of the initialized consensus object, O_i. That is, p_i's round number is at least zero. Line 90 advances p_i's round number while making sure that the round number is not behind the garbage collection line, which is marked by $\mathsf{gc}_i()$. Note that this is done only if the if-statement condition in line 89 holds. In detail, p_i cannot advance its round number if the difference between the extrema of round numbers that p_i stores in $\mathsf{rnd}_i[]$ is at least $(M - 2)$, and p_i holds the maximum round number. (Theorem 6.2 shows that, eventually, p_i either decides or the if-statement condition in line 89 cannot hold.)

Line 93 performs another consistency test for the local state of the initialized consensus object, O_i. That is, p_i has to store non-$\perp$ values as the estimated decision values and leaders. Lines 94 to 95 remove stale information from the arrays $\mathsf{phs}[][]$, $\mathsf{est}[][]$, and $\mathsf{lead}[][]$. That is, the entries that their round numbers are behind the garbage collection line (marked by $\mathsf{gc}_i()$).

Upon the arrival of a PHASE() message (line 109), node p_i updates the $\mathsf{rnd}_i[]$ array with the arriving round number from p_j (line 112).

8. Correctness of Algorithm 4

The correctness proof for Algorithm 4 is based on the one for Algorithm 3. Specifically, we rewrite Condition 6.1, Definition 6.1, Theorem 6.2, and respectively, Theorem 6.3 as Condition 8.1, Definition 8.1, Theorem 8.2, and Theorem 8.3. We note that Theorem 6.13 holds as is also for Algorithm 4, and thus, not repeated here.

Condition 8.1 *(Consistency condition for Algorithm 4).* $\forall i \in Correct : \mathsf{rnd}_i[i] \geq 0 \wedge (\nexists y \in \{\mathsf{gc}_i(), \dots, \mathsf{r}_i()\}\} : \perp \in \{\mathsf{est}_i[z][i][0], \mathsf{lead}_i[z][i]\} \wedge z = y \bmod M)$.

Definition 8.1 rewrites Definition 6.1 by replacing Condition 6.1 with Condition 8.1.

Definition 8.1. Let R be an execution of Algorithm 4, $c \in R$ be a system state, and $p_i \in \mathcal{P} : i \in Correct$ be a correct node. Suppose that in c either $O_i = \perp$ holds or Condition 8.1 holds. In this case, we say that p_i is consistent in c. The definitions of consistent system states and executions are the same as in Definition 6.1.

Theorem 8.2 *(Algorithm 3's convergence). Let R be an execution of Algorithm 4. Within finite time, the system reaches a state $c \in R$ that starts a consistent execution (Definition 8.1).*

Proof of Theorem 8.2. Suppose that R's starting state is not consistent, specifically, with respect to node $i \in Correct$. *I.e.,* the if-statement condition in line 87 or 92 holds or the for-each condition in line 94 holds with respect to at least one element. Within a complete iteration of the do forever loop, every correct node p_i takes a step that includes the execution of lines 87 to 95, which assures that p_i becomes consistent. We observe from the code of Algorithm 4 that once p_i is consistent in c, node p_i is also consistent in any state $c' \in R$ that follows c, cf. lines 87 to 95 as well as lines 104 to 108. The rest of the proof is followed by the same reasons given in the proof of Theorem 6.2. $\square_{\textit{Theorem 8.2}}$

Theorem 8.3 *(Algorithm 3's termination). Let R be a consistent execution of Algorithm 4. Suppose $\exists x \in Correct : O_x \neq \perp$ holds in the starting system state of R. Eventually the system reaches a state, $c \in R$ after which $\forall x \in Correct : \mathsf{result}_x() \neq \perp$ always hold.*

Proof of Theorem 8.3. The proof of Theorem 6.3 addresses a number of concerns that are related to the plausibility that Algorithm 3 cannot terminate, say, by not exiting the repeat-until loop in lines 58 to 64. We observe from the code of Algorithm 4 that, in addition to the concerns addressed by the proof of Theorem 6.3, Algorithm 4 has one more concern that is related to termination. Specifically, if, eventually, the if-statement condition in line 89 always holds with respect to all non-failing nodes. Because then, none of these nodes can ever increase its round number. Therefore, the rest of the proof focuses on showing that at least one correct node, p_i, decides or the if-statement condition in line 89 does not hold, infinitely often, with respect to at least one non-failing node.

Suppose, toward a contradiction, that no non-failing node decides during R and the if-statement condition in line 89 eventually does not hold for any non-failing node.

Argument 1: *eventually, there is $z \in \mathbb{Z}^+$, such that for any non-failing node p_i, it holds that $z = \mathsf{rnd}_i[i] = \mathsf{r}_i()$.* By the assumption that there is a system state $c \in R$ for which the if-statement condition in line 89 forever does not hold with respect to any non-failing node, we know that no node p_i increments its round number $\mathsf{rnd}_i[i]$ after c. Also, by line 102 for every non-failing node $p_j \in \mathcal{P}$ it holds that $\mathsf{rnd}_j[i]$ stores the value of $\mathsf{rnd}_i[i]$ eventually. The rest of the argument proof is by the definition of $\mathsf{r}()$.

Argument 2: *for any non-failing node p_i, eventually $\mathsf{r}_i() - \mathsf{gc}_i() \geq (M-2)$ cannot hold.* For Argument (1), all non-failing nodes p_i share the same value, z, in $\mathsf{rnd}_i[i]$ and $\mathsf{r}_i()$. Since eventually all faulty nodes are suspected, then $\mathsf{r}() - \mathsf{gc}() = 0$. By the assumption that $M > 2$, we know that $\mathsf{r}_i() - \mathsf{gc}_i() \geq (M-2)$ cannot hold. $\square_{\textit{Theorem 8.3}}$

As mentioned, the indulgence criterion requires that even if the Ω failure detector always outputs an incorrect value, the safety properties of consensus (validity and agreement) are never violated. Also, the criterion of zero-degradation requires the number of communication rounds until a decision is achieved to be the same in every stable run (Section 3.1.3). We note that the proposed solution satisfies indulgence and zero-degradation criteria, as they are inherited from Algorithm 2. For details, please see the end of Section 6.

9. Conclusions

We showed how a non-self-stabilizing algorithm for indulgent zero-degrading binary consensus by Guerraoui and Raynal [15] can be transformed into one that can recover after the occurrence of transient faults. We also obtained a self-stabilizing asynchronous Ω failure detector from the non-self-stabilizing construction by Mostéfaoui, Mourgaya, and Raynal [11]. Our transformation techniques are based on a number of considerations, such as (i) avoiding lengthy periods of recovery from transient faults (due to counting to infinity), (ii) circumventing the need to demonstrate the absence of live-locks after the occurrence of transient faults (by unifying messages and repeat-until loops), and (iii) bounding the size of variables via (iii.a) recycling of entries in arrays using failure detection, and (iii.b) recycling of round numbers using a global restart procedure. We encourage the reader to take these techniques into account when designing distributed systems that can recover from transient faults.

CRediT authorship contribution statement

Oskar Lundström: Conceptualization, Formal analysis, Methodology, Writing – original draft, Writing – review & editing. **Michel Raynal:** Conceptualization, Formal analysis, Methodology, Writing – review & editing. **Elad M. Schiller:** Conceptualization, Formal analysis, Methodology, Project administration, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgements

We thank Daniel Kem, Daniel Karlberg, and Amanda Sjöö for useful discussions.

References

[1] M.J. Fischer, N.A. Lynch, M. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382.
[2] M. Raynal, Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach, Springer, 2018.
[3] V. Hadzilacos, S. Toueg, A modular approach to fault-tolerant broadcasts and related problems, Tech. rep., Cornell Univ., Ithaca, NY, 1994.
[4] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.
[5] R. Guerraoui, Indulgent algorithms, in: Principles of Distributed Computing, PODC, ACM, 2000, pp. 289–297.
[6] P. Dutta, R. Guerraoui, Fast indulgent consensus with zero degradation, in: Dependable Computing EDCC, in: LNCS, vol. 2485, Springer, 2002, pp. 191–208.
[7] O. Lundström, M. Raynal, E.M. Schiller, Brief announcement: self-stabilizing total-order broadcast, in: SSS, in: Lecture Notes in Computer Science, vol. 13751, Springer, 2022, pp. 358–363.
[8] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.
[9] R. van Renesse, D. Altinbuken, Paxos made moderately complex, ACM Comput. Surv. 47 (3) (2015) 42.
[10] T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, J. ACM 43 (4) (1996) 685–722.
[11] A. Mostéfaoui, E. Mourgaya, M. Raynal, Asynchronous implementation of failure detectors, in: Dependable Systems and Networks DSN, IEEE Computer Society, 2003, pp. 351–360.
[12] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, On implementing omega with weak reliability and synchrony assumptions, in: PODC, ACM, 2003, pp. 306–314.
[13] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, Communication-efficient leader election and consensus with limited link synchrony, in: Principles of Distributed Computing, PODC, ACM, 2004, pp. 328–337.
[14] R. Guerraoui, N.A. Lynch, A general characterization of indulgence, in: Stabilization, Safety, and Security of Distributed Systems, SSS, in: LNCS, vol. 4280, Springer, 2006, pp. 16–34.
[15] R. Guerraoui, M. Raynal, The information structure of indulgent consensus, IEEE Trans. Comput. 53 (4) (2004) 453–466.
[16] R. Guerraoui, M. Raynal, The alpha of indulgent consensus, Comput. J. 50 (1) (2007) 53–67.
[17] I. Keidar, S. Rajsbaum, A simple proof of the uniform consensus synchronous lower bound, Inf. Process. Lett. 85 (1) (2003) 47–52.
[18] M. Hurfin, A. Mostéfaoui, M. Raynal, A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors, IEEE Trans. Comput. 51 (4) (2002) 395–408.
[19] W. Wu, J. Cao, J. Yang, M. Raynal, Using asynchrony and zero degradation to speed up indulgent consensus protocols, J. Parallel Distrib. Comput. 68 (7) (2008) 984–996.
[20] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing indulgent zero-degrading binary consensus, in: 22nd Distributed Computing and Networking, ICDCN, ACM, 2021, pp. 106–115.
[21] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[22] S. Dolev, Self-Stabilization, MIT Press, 2000.
[23] K. Altisen, S. Devismes, S. Dubois, F. Petit, Introduction to Distributed Self-Stabilizing Algorithms, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2019.
[24] P. Blanchard, S. Dolev, J. Beauquier, S. Delaët, Practically self-stabilizing Paxos replicated state-machine, in: 2nd Networked Systems (NETYS'14), in: LNCS, vol. 8593, Springer, 2014, pp. 99–121.
[25] J. Beauquier, S. Kekkonen-Moneta, Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors, Int. J. Syst. Sci. 28 (11) (1997) 1177–1187.
[26] C. Delporte-Gallet, S. Devismes, H. Fauconnier, Robust stabilizing leader election, in: Stabilization, Safety, and Security of Distributed Systems, SSS, in: LNCS, vol. 4838, Springer, 2007, pp. 219–233.
[27] M. Hutle, J. Widder, On the possibility and the impossibility of message-driven self-stabilizing failure detection, in: Self-Stabilizing Systems, SSS, in: LNCS, vol. 3764, Springer, 2005, pp. 153–170.
[28] M. Hutle, J. Widder, Self-stabilizing failure detector algorithms, in: T. Fahringer, M.H. Hamza (Eds.), Parallel and Distributed Computing and Networks, IASTED, IASTED/ACTA Press, 2005, pp. 485–490.
[29] M. Biely, M. Hutle, L.D. Penso, J. Widder, Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency, in: Stabilization, Safety, and Security of Distributed Systems, SSS, in: LNCS, vol. 4838, Springer, 2007, pp. 4–20.
[30] S. Dolev, R.I. Kat, E.M. Schiller, When consensus meets self-stabilization, J. Comput. Syst. Sci. 76 (8) (2010) 884–900.
[31] N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, S. Tixeuil, Practically stabilizing SWMR atomic memory in message-passing systems, J. Comput. Syst. Sci. 81 (4) (2015) 692–701.
[32] I. Salem, E.M. Schiller, Practically-self-stabilizing vector clocks in the absence of execution fairness, in: 6th Networked Systems NETYS, in: LNCS, vol. 11028, Springer, 2018, pp. 318–333.
[33] S. Dolev, C. Georgiou, I. Marcoullis, E.M. Schiller, Practically-self-stabilizing virtual synchrony, J. Comput. Syst. Sci. 96 (2018) 50–73.
[34] K.P. Birman, T.A. Joseph, Reliable communication in the presence of failures, ACM Trans. Comput. Syst. 5 (1) (1987) 47–76.
[35] S. Dolev, E. Schiller, Communication adaptive self-stabilizing group membership service, IEEE Trans. Parallel Distrib. Syst. 14 (7) (2003) 709–720.
[36] S. Dolev, E. Schiller, Self-stabilizing group communication in directed networks, Acta Inform. 40 (9) (2004) 609–636.
[37] S. Dolev, E. Schiller, J.L. Welch, Random walk for self-stabilizing group communication in ad hoc networks, IEEE Trans. Mob. Comput. 5 (7) (2006) 893–905.
[38] C. Georgiou, O. Lundström, E.M. Schiller, Self-stabilizing snapshot objects for asynchronous failure-prone networked systems, in: 7th Networked Systems NETYS, in: LNCS, vol. 11704, Springer, 2019, pp. 113–130.
[39] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing uniform reliable broadcast, in: 8th Networked Systems NETYS, in: Lecture Notes in Computer Science, vol. 12129, Springer, 2020, pp. 296–313, CoRR, arXiv:2001.03244 [abs].
[40] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing set-constrained delivery broadcast, in: IEEE 40th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2020, pp. 617–627.
[41] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing multivalued consensus in asynchronous crash-prone systems, in: EDCC, IEEE, 2021, pp. 111–118.

[42] S. Dolev, T. Petig, E.M. Schiller, Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks, CoRR, arXiv:1806.03498 [abs], 2018.
[43] S. Dolev, J.L. Welch, Self-stabilizing clock synchronization in the presence of byzantine faults, J. ACM 51 (5) (2004) 780–799.
[44] M. Ben-Or, D. Dolev, E.N. Hoch, Fast self-stabilizing byzantine tolerant digital clock synchronization, in: PODC, ACM, 2008, pp. 385–394.
[45] A. Maurer, S. Tixeuil, Self-stabilizing byzantine broadcast, in: 33rd IEEE International Symposium on Reliable Distributed Systems, SRDS, 2014, pp. 152–160.
[46] R. Duvignau, M. Raynal, E.M. Schiller, Self-stabilizing byzantine fault-tolerant repeated reliable broadcast, Theor. Comput. Sci. 972 (2023) 114070.
[47] C. Georgiou, I. Marcoullis, M. Raynal, E.M. Schiller, Loosely-self-stabilizing byzantine-tolerant binary consensus for signature-free message-passing systems, in: NETYS, in: Lecture Notes in Computer Science, vol. 12754, Springer, 2021, pp. 36–53.
[48] R. Duvignau, M. Raynal, E.M. Schiller, Self-stabilizing byzantine multivalued consensus, ICDCN 2024 and CoRR, arXiv:2311.09075 [abs], 2023.
[49] C. Georgiou, M. Raynal, E.M. Schiller, Self-stabilizing byzantine-tolerant recycling, in: SSS, in: Lecture Notes in Computer Science, vol. 14310, Springer, 2023, pp. 518–535.
[50] A. Binun, T. Coupaye, S. Dolev, M. Kassi-Lahlou, M. Lacoste, A. Palesandro, R. Yagel, L. Yankulin, Self-stabilizing byzantine-tolerant distributed replicated state machine, in: SSS, in: Lecture Notes in Computer Science, vol. 10083, 2016, pp. 36–53.
[51] S. Dolev, C. Georgiou, I. Marcoullis, E.M. Schiller, Self-stabilizing byzantine tolerant replicated state machine based on failure detectors, in: CSCML, in: Lecture Notes in Computer Science, vol. 10879, Springer, 2018, pp. 84–100.
[52] S. Dolev, O. Liba, E.M. Schiller, Self-stabilizing byzantine resilient topology discovery and message delivery, in: NETYS, in: Lecture Notes in Computer Science, vol. 7853, Springer, 2013, pp. 42–57.
[53] G. Tel, Introduction to Distributed Algorithms, Cambridge University Press, 2000.
[54] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann, 1996.
[55] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing total-order broadcast, in: SSS, in: Lecture Notes in Computer Science, vol. 13751, Springer, 2022, pp. 358–363.