Self-stabilizing multivalued consensus in asynchronous crash-prone systems

Citation for the original published paper (version of record):
Lundström, O., Raynal, M., Schiller, E. (2024). Self-stabilizing multivalued consensus in asynchronous crash-prone systems. Theoretical Computer Science, 1022.
http://dx.doi.org/10.1016/j.tcs.2024.114886

(article starts on next page)

Theoretical Computer Science 1022 (2024) 114886

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Self-stabilizing multivalued consensus in asynchronous crash-prone systems ☆

Oskar Lundström [a], Michel Raynal [b], Elad Michael Schiller [a,*]

[a] Chalmers University of Technology, Sweden
[b] IRISA, Univ. Rennes 1, France

A B S T R A C T

The multivalued consensus problem is a fundamental issue in fault-tolerant distributed computing. It encompasses a wide range of agreement problems where processes must unanimously decide on a specific value $v \in V$, with $|V| \geq 2$. Existing solutions that handle process crash failures simplify the multivalued consensus problem by reducing it to the binary consensus problem. Examples of such solutions include Mostéfaoui-Raynal-Tronel [IPL 2000] and Zhang-Chen [IPL 2009].

In this work, we aim to design an even more reliable solution by leveraging the concept of *self-stabilization*, which provides a strong form of fault tolerance. Self-stabilizing algorithms can recover from transient faults, which represent any deviation from the system's intended behavior (as long as the algorithm code remains intact) in addition to process and communication failures.

To the best of our knowledge, this work presents the first self-stabilizing algorithm for multivalued consensus in asynchronous message-passing systems susceptible to process failures and transient faults. Our solution uses, at most, n concurrent invocations of binary consensus. This is another way we advance state-of-the-art solutions compared to previous non-self-stabilizing ones. For example, Mostéfaoui-Raynal-Tronel's solution requires an unbounded number of sequential invocations of binary consensus.

1. Introduction

We propose, to the best of our knowledge, the first self-stabilizing, non-blocking, and memory-bounded implementation of *multivalued consensus* objects for asynchronous message-passing systems whose processes may crash.

1.1. Background and motivation

Fault-tolerant distributed systems find applications across diverse domains, such as banking, transportation, tourism, production, and commerce. These applications rely on message-passing systems and demand robustness against failures. Designing and verifying such systems is notoriously challenging due to the combination of failures and asynchrony, which introduces uncertainties regarding the application state from the perspective of individual processes. For instance, Fischer, Lynch, and Paterson [1] demonstrated that, in any asynchronous message-passing system, the occurrence of just one process crash is sufficient to prevent deterministic consensus from being achieved.

Our main application focus is the emulation of finite-state machines [2]. To ensure consistency, all emulating processes must follow identical sequences of state transitions. To achieve this, we divide the problem into two parts: (i) propagating user input to

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

* Corresponding author.
E-mail address: elad@chalmers.se (E.M. Schiller).

https://doi.org/10.1016/j.tcs.2024.114886
Received 20 August 2023; Received in revised form 26 July 2024; Accepted 18 September 2024

Available online 24 September 2024

all emulating processes, and (ii) ensuring that the emulating processes execute identical sequences of state transitions. The issue of propagating user input to all processes can be addressed using uniform reliable broadcast [3–5], which solves Problem (i). However, this work concentrates explicitly on Problem (ii) as the core challenge. In other words, all processes must reach a consensus on a shared value that dictates the execution of state transitions across all emulating processes. The literature on fault-tolerant consensus is extensive. This work advances the state of the art by tolerating a significantly broader set of failures and using a bounded amount of memory.

1.2. Problem definition and scope

The definition of the consensus problem appears in Definition 1.1. This work focuses on studying the multivalued version of the problem, where the set of values that can be proposed includes at least two values. We clarify that the upper bound on the number of these values can be any predefined constant. It is important to note that there is another version of the problem in which the set includes (at most) two values, known as binary consensus. Existing solutions for multivalued consensus (including the proposed one) often rely on binary consensus algorithms. The relationship among the mentioned problems is presented in Fig. 1.

Definition 1.1 *(Consensus).* 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).

- **Validity.** Suppose that v is decided. Then, some process had invoked $\mathsf{propose}(v)$.
- **Termination.** All non-faulty processes decide.
- **Agreement.** No two processes decide different values.
- **Integrity.** No process decides more than once.

Definition 1.1 considers algorithms that can only be invoked once. Definition 1.2 enhances Definition 1.1 by considering repeated invocations. These invocations can occur in batches of δ consensus objects, where δ is a predefined constant. For example, in this work, where we solve the problem of multivalued consensus, we consider $\delta \in \{1, n\}$ when using n binary consensus objects. Also, in [2], where we solve the problem of reliable total-order broadcast, we consider $\delta = 1$ when using 3 multivalued consensus objects. The solution in [2] shows how these three objects are recycled; one object at a time.

Definition 1.2 *(Repeated Consensus).* Let Alg be an algorithm that satisfies the requirements of Definition 1.1 for δ consensus objects, executed concurrently. Moreover, once all these δ objects have completed their task and delivered their decisions, all δ objects are recycled in a way that allows their reuse.

1.3. Fault model

An asynchronous distributed system is one in which components or processes operate independently of each other in terms of timing, meaning they are not synchronized to a common clock signal. In such systems, events or operations are driven by external stimuli or internal conditions. Asynchronous systems are often contrasted with synchronous systems, where all operations are coordinated and synchronized according to a common clock signal or the use of local timers.

We consider an asynchronous message-passing system with no guarantees on communication delays (except that they are finite) since it is well-known that bounded communication delays can serve as the basis for emulating global clocks. We clarify that the studied model does not allow the algorithm to access the local clock or use timers since they can also be used to emulate global clocks. Our fault model encompasses two types of failures: (i) crashes affecting less than half of the processes and (ii) communication failures, including packet omission, duplication, and reordering.

In addition to the failures outlined in our model, we aim to address recovery from *transient faults*. These transient faults encompass any temporary violation of the assumptions under which the system and network were designed to operate. For example, this includes the corruption of critical control variables such as the program counter, packet payload, and indices (*e.g.,* sequence numbers) that are essential for the correct running of the algorithm. Additionally, we consider operational assumptions, such as the assumption that at least a majority of processes never fail. As these transient faults can occur in arbitrary combinations, they may lead to unpredictable alterations in the system state. Our modeling assumes that these violations bring the system to an arbitrary state from which a *self-stabilizing algorithm* should recover the system after the occurrence of the last transient fault. After this recovery period, it is essential to ensure that the system satisfies the task requirements (*e.g.,* Definition 1.1).

1.4. Related work

The celebrated Paxos algorithm [6] circumvents the impossibility, as proven by Fischer, Lynch, and Paterson (FLP) [1], by assuming that failed computers can be detected using unreliable failure detectors [7]. Paxos has served as an inspiration for numerous veins of research, as seen in [8] and the references therein.

In this work, however, we follow Raynal's family of abstractions [3] due to its clear and easily comprehensible presentation. Notably, the studied algorithm does not consider direct access to failure detectors. Instead, it assumes the availability of binary

finite-state machine emulation [2]
total-order delivery [2]
multivalued consensus (Algorithm 2)
binary consensus [21]
uniform reliable broadcast [5]

Fig. 1. An illustration of the hierarchical structure of protocols involved in the studied problem of multivalued consensus (highlighted in bold). The protocol suite above shows how Uniform Reliable Broadcast (URB) is utilized to implement binary consensus, which is used to implement multivalued consensus. This multivalued consensus is then employed to achieve total-order delivery, ultimately enabling finite-state machine emulation. The cited articles refer to relevant self-stabilizing solutions for each of these protocols.

consensus objects. We clarify that the latter utilizes the weakest failure detector, as described by Raynal [3]. Also, our study focuses on deterministic solutions and does not consider probabilistic approaches, such as [9–11].

1.4.1. Non-self-stabilizing solutions

Mostéfaoui, Raynal, and Tronel [12], from now on MRT, reduce multivalued consensus to binary consensus via a crash-tolerant block-free algorithm. MRT's algorithm uses an unbounded number of invocations of binary consensus objects and, at most, one uniform reliable broadcast (URB) per process. Zhang and Chen [13] proposed an algorithm for multivalued consensus that uses only x instances, where x is the number of bits it takes to represent any value in the domain of proposable values.

Our self-stabilizing solution terminates within at most n invocations of binary consensus objects and at most one uniform reliable broadcast [5,14] (URB) operation per process, where n is the number of processes in the system.

Afek et al. [15] showed that binary and multivalued versions of the k-simultaneous consensus task are wait-free equivalent. Here, the k-simultaneous consensus is required to let each process participate at the same time in k-independent consensus instances until it decides in any of them.

1.4.2. Self-stabilizing solutions

We follow the design criteria of self-stabilization, which Dijkstra [16] proposed. Dolev [17] and Altisen et al. [18] provided a detailed pretension of self-stabilization. Consensus was insufficiently explored in the context of self-stabilization. Blanchard et al. [19] presented the first solution in the context of self-stabilization. They presented a practically-self-stabilizing version of Paxos [6]. We note that practically-self-stabilizing systems, as defined by Alon et al. [20], 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.

We base our self-stabilizing multivalued consensus on the self-stabilizing binary consensus [21] and Uniform Reliable Broadcast [5] (URB, in short) both by Lundström, Raynal, and Schiller. These solutions are based on an assumption by Dolev, Petig, and Schiller [22] for self-stabilization in the presence of seldom fairness. The latter is defined by Dolev, Petig, and Schiller as follows. In the absence of transient faults, these self-stabilizing algorithms do not assume system synchrony or fairness of its scheduler. Nevertheless, after the occurrence of the last transient faults, the system recovery assumes that the system execution eventually becomes fair so that a global reset [23] is possible. This new approch presented by Dolev, Petig, and Schiller allows us to relax assumptions related to execution fairness [24–27].

Georgiou, Lundström, and Schiller [28] conducted a study concerning self-stabilizing atomic snapshot objects. We study a different problem. Furthermore, we note the existence of self-stabilizing solutions for consensus within shared-memory systems susceptible to fail-stop failures [29], as well as message-passing systems prone to Byzantine failures [30–32]. These solutions address a similar problem. However, their fault models are more challenging than the ones studied here. Thus, the solution proposed in this paper is simpler than the ones in [30–32].

1.4.3. Applications

The studied solution is part of a protocol suite (Fig. 1) and it facilitates the self-stabilizing emulation of state-machine replication via reliable total-order broadcast [2]. Dolev et al. [33] proposed the first practically-self-stabilizing emulation of state-machine replication, which has a similar task to the one in Fig. 1. However, Dolev et al.'s solution does not guarantee recovery within a finite time since it does not follow Dijkstra's criterion. Moreover, it is based on virtual synchrony by Birman and Joseph [34], where the one in Fig. 1 utilizes consensus.

1.4.4. Bibliographical notes

An earlier version of this work appeared as an extended abstract [35] and technical report [36].

1.5. Our contribution

We present a self-stabilizing algorithm for multivalued consensus in asynchronous message-passing systems susceptible to crash failures. To the best of our knowledge, we are the first to provide a solution for multivalued consensus that tolerates a broad fault model, encompassing crashes, communication failures (*e.g.,* packet omission, duplication, and reordering), as well as transient faults, all while using a bounded amount of resources (unlike earlier work, such as the one by MRT, that uses an unbounded number of binary consensus objects). The latter type of failure models any violation of the assumptions under which the system was designed to operate (as long as the program code, which represents the proposed algorithm, remains intact).

Our solution achieves (multivalued) consensus within n invocations of binary consensus instances that run sequentially or concurrently, where n is the number of processes.[1] Furthermore, the concurrent version of our solution can piggyback the binary consensus messages and terminate within the time it takes to complete one uniform reliable broadcast (URB) and one binary consensus (when running n invocations concurrently). This time frame also represents the system's recovery period after the occurrence of the last transient fault, *i.e.,* the stabilization time.

We note that each URB invocation needs to be repeated until the invoking algorithm deactivates the consensus object. This is due to a well-known impossibility [17, Chapter 2.3], which says that self-stabilizing systems cannot be quiescent, *i.e.,* stop sending messages once the protocol has terminated. It is easy to trade the broadcast repetition rate with the recovery speed from transient faults.

As an application to the proposed solution, we refer to our self-stabilizing total order uniform reliable broadcast and finite-state machine emulation [2]. We note that the abstraction of state-machine replication is widely recognized as one of the most fundamental concepts in the area of distributed computing and systems.

1.6. Document organization

The task specifications and solution organization appear in Section 2. We state our system settings in Section 3. Section 4 includes a brief overview of the studied algorithm by Mostéfaoui, Raynal, and Tronel [12] that has led to the proposed solution. Our self-stabilizing algorithm for multivalued consensus is proposed in Section 5. The correctness proof appears in Section 6. We conclude in Section 7 and explain how to extend the proposed application to serve as an emulator for state-machine replication.

2. Task specifications and solution organization

The proposed solution is tailored for the protocol suite presented in Fig. 1. Therefore, before detailing how these tasks are integrated into a cohesive solution, we first list the external building blocks and define the tasks under study.

2.1. External building-blocks

The proposed solution for multivalued consensus uses binary consensus and reliable broadcast objects.

2.1.1. Binary consensus objects

T_{binCon} denotes the task of binary consensus, which Definition 1.1 specifies. We assume the availability of self-stabilizing binary consensus objects, such as the one by Lundström, Raynal, and Schiller [21]. As in Definition 1.1, the proposed and decided values have to be from the V domain (of proposable values). For clarity's sake, for a given binary consensus object BC, the operation $BC.\mathsf{binPropose}(v)$ invokes the binary consensus on $v \in V = \{\mathsf{True}, \mathsf{False}\}$. (Traditionally, binary consensus results are either 0 or 1, but we rename them.)

2.1.2. Uniform Reliable Broadcast (URB)

The task T_{URB} of Uniform Reliable Broadcast (URB) [4] considers an operation for URB broadcasting of message m and an event of URB delivery of message m. The requirements include URB-validity, *i.e.,* there is no spontaneous creation or alteration of URB messages, URB-integrity, *i.e.,* there is no duplication of URB messages, as well as URB-termination, *i.e.,* if the broadcasting node is non-faulty, or if at least one receiver URB-delivers a message, then all non-failing processes URB-deliver that message. Note that the URB-termination property considers both faulty and non-faulty receivers. This is the reason why this type of reliable broadcast is named *uniform*. We clarify that the set of receivers also includes the sending node.

The proposed solution assumes the availability of a self-stabilizing URB, such as the one by Lundström, Raynal, and Schiller [5]. Their solution allows the operation for URB broadcast of message m to return a transmission descriptor, $txDes$, which is the unique message identifier. Using a failure detector [7], they provide a predicate, $\mathsf{hasTerminated}(txDes)$, that holds whenever the sender knows that all processes (that are not-suspected to be faulty) have delivered m. Their implementation of $\mathsf{hasTerminated}(txDes)$ simply tests that all non-suspected receivers have acknowledged the arrival of the message with identifier $txDes$. This way, their solution implements $\mathsf{hasTerminated}()$ since their self-stabilizing algorithm considers such messages as 'obsolete' messages and lets the garbage collector remove them.

2.2. Task specification: multivalued consensus

T_{mulCon} denotes the task of multivalued consensus, which Definition 1.1 specifies. As in Definition 1.1, the proposed and decided values have to be from the V domain (of proposable values), where $|V| \geq 2$. The operation $\mathsf{propose}(v)$ invokes the multivalued consensus on $v \in V$.

[1] We note the possibility of the following straightforward extension. By running concurrent batches of binary consensus objects while preserving a sequential order between the batches, one can balance the trade-off between message size and the potential speed-up provided by concurrency since the size of each batch can be a predefined fraction of n..

2.3. Solution overview

We consider a multivalued consensus object that uses an array, $BC[]$, of n binary consensus objects, such as the one by [3, Chapter 17], where n is the number of processes in the system. The proposed algorithm considers a single multivalued consensus object, denoted by O.

Definition 1.1 considers the propose(v) operation, but it does not specify how the decided value is retrieved. We clarify that it can be either via the returned value of the propose(v) (or binPropose(v) for the case of binary consensus) operation as the one in MRT's algorithm [12] (as in the algorithm by Guerraoui and Raynal [37], respectively) or via the returned value of the result() operation, as in the proposed solution. But, if processor p_i has yet to access the decided value, $\mathsf{result}_i()$ returns $\perp$. Otherwise, the decided value is returned. Specifically, for the case of propose(v), $\mathsf{result}_i()$ is used and for the case of binPropose(v), also the parameter k should be used when calling $\mathsf{result}_i(k)$, where p_k is a system processor. We clarify that, in the absence of transient faults, $\mathsf{result}_i()$ and $\mathsf{result}_i(k)$ always return either $\perp$ or the decided value for the multivalued and binary consensus objects, respectively.

The algorithm by Guerraoui and Raynal [37], which we use as a starting point in the design of the proposed solution, was not designed to deal with transient faults. As Section 4.2 explains, transient faults can cause the studied algorithm to violate Definition 1.1's requirements without indicating the invoking algorithm. Thus, after a transient fault occurs, the proposed solution allows $\mathsf{result}_i()$ to provide such indication to the invoking algorithm via the return of the *transient error* symbol Ψ. See details in Section 5.2.2.

3. System settings

We consider an asynchronous 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).[2] The system consists of a set, $\mathcal{P} = \{p_0, \ldots, p_{n-1}\}$, of n crash-prone processors (or nodes) with unique identifiers.[3] We assume that any pair of nodes $p_i, p_j \in \mathcal{P}$ have access to bidirectional peer-to-peer communication media, where $channel_{j,i}$ denotes the communication channel that holds protocol message in transit from p_j to p_i. Due to an impossibility [17, Chapter 3.2], we assume that $channel_{j,i}$, at any time, has at most $\mathsf{channelCapacity} \in \mathbb{N}$ protocol messages on transit from p_j to p_i. Specifically, Jayaram and Varghese [38] show that, without such an assumption, crash failures can drive deterministic asynchronous distributed systems to arbitrary states regardless of the presence of transient faults.

3.1. The interleaving model

The *interleaving model* [17] is a framework used to analyze the behavior of distributed systems, particularly in the context of concurrency. It represents the execution of processes by interleaving their actions into a single global sequence, allowing us to visualize and reason about the system's behavior as if the processes were executed one at a time, even though they actually run concurrently. The model considers all possible interleavings of actions from different processors, with each interleaving corresponding to a different possible execution of the system, thereby capturing the non-determinism inherent in concurrent execution.

The node's program is a sequence of *(atomic) steps*. These steps are atomic in the sense that they represent operations that are indivisible and uninterruptible. This means that the step is executed entirely or not at all, with no intermediate states visible to other operations or processors. Specifically, 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 $c = (s_0, s_1, \cdots, s_{n-1})$. We define a *system execution (or run)* $R = c[0], a[0], c[1], a[1], \ldots$ as an alternating sequence of system states $c[x]$ and (atomic) steps $a[x]$, such that each $c[x+1]$, except for the starting one, $c[0]$, is obtained from $c[x]$ by the execution of step $a[x]$ that some processor takes. We clarify that, thus far, the term system execution has only considered the algorithm steps. However, in the presence of failures, the system execution also includes adversarial steps injected by the environment; see Section 3.2 for details.

As mentioned in Section 2.2, the studied distributed systems consider a set of specific tasks (Fig. 1), such as the task of repeated multivalued consensus, which is denoted by T_{mulCon}, see Definition 1.2 for the specified requirements of T_{mulCon}. The set of *legal executions* (LE) refers to all the executions in which the requirements of a given task hold.

3.2. The fault model and self-stabilization

As mentioned, in addition to the steps taken by the algorithm, in the presence of failures, the system execution may include adversarial steps taken by the environment or disable the algorithm from taking specific steps.

[2] Such exclusion of the use of clocks is well-known to simplify the efficient deployment of distributed systems since (unlike synchronous systems) there is no need to wait until it is possible to guarantee that the slowest component in the system is ready to process the next communication round (which synchronous systems require to occur simultaneously)..

[3] For the sake of a simple presentation, we use the processor's index as its identifier. One can simply substitute the use of the index by the algorithm with the function $ID(i)$, which returns a unique processor identifier that is not necessarily between 0 and $n-1$..

3.2.1. Benign failures

The system is prone to *crash failures*, in which faulty nodes stop taking steps forever. We assume that at most $t < n/2$ node may crash. We denote by *Correct* the set of indices of processors that never crash. We consider solutions oriented towards asynchronous message-passing systems in which the notion of time does not play any role. Thus, the algorithm is oblivious to when packets arrive and depart. Also, the communication channels are prone to packet failures, such as omission (in which the environment removes a message from a communication channel), duplication (in which the environment duplicates a message), and reordering (in which the environment reorders messages). 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. We assume any message can reside in a communication channel only for a finite period (before it is delivered or lost).

3.2.2. Transient faults

We consider any violation of the assumptions according to which the system was designed to operate. We refer to these violations and deviations as *transient faults* and assume they can corrupt the system state arbitrarily (as long as the program code, which represents the proposed algorithm, remains intact). We follow the model presented by Dijkstra [16] and detailed by Dolev [17] and Altisen et al. [18]. The model assumes that the last transient fault occurs before the system execution starts, *i.e.,* before the first system state of any system execution. Moreover, the occurrence of the last transient fault leaves the system to begin in an arbitrary state. As we explain in Section 3.2.3, a self-stabilizing system must eventually recover from any arbitrary state. Once the recovery process is completed, the self-stabilizing system must exhibit correct behavior indefinitely.

3.2.3. Dijkstra's self-stabilization criterion

An algorithm is *self-stabilizing* with respect to the task of LE, when every (unbounded) execution R of the algorithm reaches within a finite period a suffix $R_{legal} \in LE$ that is legal. That is, Dijkstra [16] requires that $\forall R : \exists R_{recovery} : R = R_{recovery} \circ R_{legal} \wedge R_{legal} \in LE \wedge |R_{recovery}| \in \mathbb{Z}^+$, where the operator $\circ$ denotes that $R = R' \circ R''$ concatenates R' with R''. In other words, any execution of a self-stabilizing system must reach a *safe state* regardless of the initial state. The literature sometimes refers to these states as *legitimate system states* because they signify the end of the *Convergence*, *i.e.,* recovery phase, and ensure that the *Closure* property is indefinite. Furthermore, starting from a safe state, the system's execution always satisfies the task requirements, thereby making these execution legal (Section 3.1).

The number of steps during recovery, denoted as $|R_{recovery}|$, serves as a complexity measure for self-stabilizing systems. In the literature, this measure is commonly referred to as *stabilization time*, acting as an analytical approximation for recovery time. Recovery time is well-defined for synchronous systems but remains conceptual for asynchronous systems under study, where it may not be directly measurable due to variations in timing and completion of recovery phases. As explained in Section 2.1, the studied and proposed solutions allow nodes to interact and share information via binary consensus objects and uniform reliable broadcast (URB). Thus, we measure the stabilization time as the number of accesses to these primitives plus the number of URB accesses. To do this, one needs to add the stabilization time of these two underlying objects when calculating the overall stabilization time.

4. The studied non-self-stabilizing multivalued consensus

We review in Sections 4.1 and 4.2 a non-self-stabilizing non-blocking algorithm for multivalued consensus by Mostéfaoui, Raynal, and Tronel [12], which uses an unbounded number of binary consensus objects.

4.1. Algorithm 1: non-self-stabilizing multivalued consensus

The non-self-stabilizing solution in Algorithm 1 is the basis for its self-stabilizing variation in Algorithm 2. The program code pertains to node p_i, where for any variable x appearing in Algorithm 1, x_i denotes the value stored by p_i for variable x. The operation propose(v) (line 5) invokes an instance of a multivalued consensus object. Algorithm 1 uses a URB protocol (Section 2.1.2) for letting any $p_i \in \mathcal{P}$ disseminate its proposed value v_i (line 7). Each $p_j \in \mathcal{P}$ that delivers this proposal, stores this value in $proposal_j[i]$. Also, $p_k \in \mathcal{P}$ can concurrently broadcast its proposal, v_k, which p_j stores in $proposal_j[k]$. Therefore, Algorithm 1 must decide which entry in $proposal_j[]$ the propose(v) operation should return. This decision is coordinated via an unbounded global array $BC[0], BC[1], \ldots$, of binary consensus objects.

Algorithm 1 starts by URB-broadcasting p_i's proposed value, v_i (line 7). This broadcast assures that all correct nodes eventually receive identical sets of messages (Section 2.1.2). Also, the set of delivered messages must include every message URB-broadcast by any correct node. The arrival of PROPOSAL(v) from p_j, informs p_i about p_j's proposal, and thus, p_i stores v_j in $proposals_i[j]$ (line 12).

Following the proposal broadcast, Algorithm 1 proceeds in asynchronous communication rounds. That is, each node operates as quickly as possible based on the conditions described in the code rather than waiting for all nodes to complete their previous communication rounds synchronously. The variable k stores the round counter (lines 3 and 8). Once p_i decides, it leaves the loop by returning the value of $proposals_i[x_i]$ (line 11), where $x_i = k_i \bmod n$. In other words, $x_i \in \{0, \ldots, n-1\}$ is the identifier of the node that has broadcast the proposal stored in $proposals_i[x_i]$.

As mentioned, the selection of x_i is facilitated via the unbounded array, $BC[]$, of binary consensus objects. Since all correct nodes eventually receive the same set of broadcasts, p_i proposes $proposals_i[x] \neq \perp$ to the k_i-th object, $BC[k_i]$ (line 9). That is, p_i proposes True on the k_i-th round if, and only if, it received p_{x_i}'s proposal.

Algorithm 1: Non-self-stabilizing non-blocking multivalued consensus using an unbounded number of binary consensus instances; code for p_i.

```
1  local variables:
2  proposals[0,..,n-1] ;                          /* array of the received proposals */
3  k ;                                            /* the communication round counter */
4  BC[0,...] ;                                    /* binary consensus objects (unbounded list) */
5  operation propose(v) begin
6      (proposals, BC) ← ([⊥,...,⊥],[⊥,...]);
7      urbBroadcast PROPOSAL(v);
8      while (k ← 0; True; k ← k + 1) do
9          if BC[k].binPropose((proposals[k mod n] ≠ ⊥)) then
10             wait(proposals[k mod n] ≠ ⊥);
11             return (proposals[k mod n]);
12 upon urbDelivered PROPOSAL(v) from p_j do {proposals[j] ← v;}
```

Algorithm 1 continues to the next round whenever $BC[k_i]$ decides False. Otherwise, p_i decides the value, $proposals_i[x_i]$, proposed by p_{x_i}. Due to asynchrony, p_i might need to wait until p_{x_i}'s broadcast was URB-delivered (line 10). However, if any node proposed to decide v_{x_i}, it must be the case that $proposals_i[x_i]$ was delivered to the node that has proposed True at $BC[k_i]$. Therefore, eventually, p_i is guaranteed to URB-deliver v_{x_i} and stores it at $proposals_i[x_i]$. For this reason, Algorithm 1 does not block forever in line 10, and the decided value is eventually returned (line 11).

4.2. *Executing Algorithm 1 in the presence of transient faults*

Before describing Algorithm 2, we review the main challenges one faces when transforming Algorithm 1 into an algorithm that can recover after the concurrence of transient faults. It is important to recall that the studied solution by Mostéfaoui, Raynal, and Tronel [12] did not consider recovery after the occurrence of transient faults.

4.2.1. *Use of an unbounded number of binary consensus objects*

Self-stabilizing systems can only use a bounded amount of memory [17]. This is because, in practice, (autonomous) computer systems use only a predefined amount of memory. However, a single transient fault can set every counter (or data structure) to its maximum value (respectively, exhaust the memory capacity of the data structure). Specifically, the correctness proof for Algorithm 1 does not consider cases where the communication round counter, k, reaches the maximum value that an integer can hold (or when all binary consensus objects are used).

4.2.2. *Corrupted counter for communication round numbers and binary consensus objects*

In the context of self-stabilization, one cannot simply rely on counter k (line 3) to count the number of asynchronous rounds. This is because a single transient fault can set the value of k to zero. It can also alter the state of every $BC[k]_{k\in\{0,\ldots,z\}\wedge z\in\mathbb{Z}^+}$, such that a call to propose() returns False, where z can be practically infinite, say $z = 2^{64} - 1$, which is a standard maximum integer value. In this case, the system must iterate 2^{64} times before reaching a fresh binary consensus object. Assuming each iteration takes just one nanosecond, counting from zero to $2^{64} - 1$ takes more than 584 years, more than any practical system's lifetime. In other words, the system will virtually block forever.

4.2.3. *Corrupted program counter*

A transient fault can set the program counter of every $p_j \in \mathcal{P}$ to skip over the broadcast in line 7 and to point to line 8. If this happens, validity or termination (Definition 1.1) can be violated. Therefore, there is a need to repeat the transmission of v_i in order to ensures that at least one proposal is known to all correct processors.

4.2.4. *A corrupted array of binary consensus objects*

Transient faults can corrupt binary consensus objects in the array $BC[]$. Specifically, since the array $BC[]$ should include only a bounded number of binary consensus objects, a transient fault can change the state of all objects in $BC[]$ to encode 'False was decided'. In this case, Algorithm 1 cannot finish the multivalued consensus.

5. The proposed solution: self-stabilizing wait-free multivalued consensus

This section presents a new self-stabilizing algorithm for multivalued consensus that uses n binary consensus objects (Section 2.1.1) and n self-stabilizing URB objects (Section 2.1.2). The correctness proof appears in Section 6.

(a) Upon propose(v), uniform reliable broadcast $\langle v \rangle$.
(b) By URB-termination, eventually, there is $p_j \in \mathcal{P}$ and round k', such that p_j's message arrived at all non-faulty processors, *i.e.,* $\forall \ell \in Correct \implies proposals_\ell[j] \neq \perp$.
(c) For $k \in \{0, 1, 2, \ldots\}$, p_i invokes $BC[k]$.binPropose($proposals[k \bmod n] \neq \perp$).
(d) By BC-termination and stage (b), eventually, the $k_{\min}$-th binary consensus objects is the first to decide True while all x-th objects decide False, where $x \in \{0, 1, 2, \ldots, k_{\min}-1\}$.
(e) Due to URB-termination, eventually, $proposals[k_{\min} \bmod n]$ includes a non-$\perp$ value.
(f) Then, return $proposals[k_{\min} \bmod n]$ as the decided value for the multivalued consensus.

Fig. 2. High-level stages in the execution of Algorithm 1; high-level code for p_i.

(a) Upon propose(v), uniform reliable broadcast $\langle v \rangle$.
(b) Wait until hasTerminated() says that $\langle v \rangle$ arrived at all non-faulty processors.
(c) For $k \in \{0, \ldots, n-1\}$, p_i invokes $BC[k]$.binPropose($proposals[k \bmod n] \neq \perp$).
(d) By BC-termination and stage (b), eventually, the $k_{\min}$-th binary consensus objects is the first to decide True while all x-th objects decide False, where $x \in \{0, 1, 2, \ldots, k_{\min}-1\}$.
(e) Due to URB-termination, eventually, $proposals[k_{\min} \bmod n]$ includes a non-$\perp$ value.
(f) Then, return $proposals[k_{\min} \bmod n]$ as the decided value for the multivalued consensus.

Fig. 3. An alternative to Fig. 2 that uses a bounded number of binary consensus objects; high-level code for p_i.

5.1. The algorithm idea

We sketch the key notions needed for Algorithm 2 by addressing the challenges raised in Section 4.2. For the sake of a simple presentation, the line numbers of Algorithm 2 continues the ones of Algorithm 1.

5.1.1. Using a bounded number of binary consensus objects

We explain how Algorithm 2 can use only n binary consensus objects at most. Fig. 2 is a high-level description of Algorithm 1's execution and Fig. 3 shows how this process can become more efficient. The key differences between Figs. 2 and 3 appear in the boxed text of Fig. 3. Specifically, Fig. 3 waits until p_i's broadcast has terminated in line (b). At that point in time, p_i knows that all non-faulty processors have received its message. Only then does p_i allow itself to propose values via the array of binary consensus objects. This means that no processor starts proposing any binary value before there is at least one index $k \in \{0, \ldots, n-1\}$ for which $proposals_j[k] \neq \perp$, where p_j is any node that has not failed. In other words, regardless of who is going to invoke the k-th binary consensus object, only the value True can be proposed. For this reason, there is no need to use more than n binary consensus values until at least one of them decides True, cf. line (c) in Fig. 3.

5.1.2. Dealing with corrupted round number counter

Using the object values in $BC[]$, Algorithm 2 calculates k(), which returns the current round number. This way, a transient fault cannot create inconsistencies between k()'s value and $BC[]$. The k()'s calculation uses a notation that considers an object, O. When $O \neq \perp$, we say that using O is enabled; thus, the object is said to be active. Otherwise, *i.e.,* $O = \perp$, its use is disabled and hence inactive.

In detail, for an active multivalued consensus object O, *i.e.,* $O \neq \perp$, we say that the binary consensus object $O.BC[k]$ is active when $O.BC[k] \neq \perp$. Algorithm 2 calculates k() (line 19) by counting the number of active binary consensus objects terminated, and the decided value is False. We restrict this counting to consider only the entries $BC[k]$, such that $k = 0$ or $\forall k' < k : BC[k']$ is an active binary consensus object that has terminated and the decided False. This is defined by the set $\mathsf{K} = (\{k \in S(n\text{-}1) : O.BC[k] \neq \perp \wedge O.BC[k].\mathsf{result}(k) = \mathsf{False}\})$, where $S(x) = \{0, \ldots, x\}$ is the set of all integers between zero and x. This way, the value of k() is $\max(\{\{-1\} \cup \{x \in S(n-1) : (S(x) \cap \mathsf{K}) = S(x)\})$. Note that the value of -1 indicates that no active binary consensus objects in $BC[]$ terminated with a decided value of False, *i.e.,* $\mathsf{K} = \emptyset$.

5.1.3. Dealing with a corrupted program counter

As explained in Section 4.2.3, there is a need to repeat the transmission of v_i to ensure that at least one proposal is known to all correct processors. Specifically, after $\mathsf{propose}_i(v_i)$'s invocation, $p_i \in \mathcal{P}$ needs to store v_i and broadcast v_i repeatedly due to a well-known impossibility [17, Chapter 2.3]. Note that there is a straightforward way to trade the broadcast repetition rate for the recovery speed from transient faults. This is because as the rate becomes slower, it takes more time for information to propagate, which leads to a slower removal of stale information. Also, once the first broadcast has terminated, all correct processors $p_i \in \mathcal{P}$ are ready to decide by proposing $\mathsf{binPropose}_i(k, proposals_i[k] \neq \perp)$ for any $p_k \in \mathcal{P}$, see steps (b) and (c) in Fig. 3.

5.1.4. Dealing with a corrupted array of binary consensus objects

Algorithm 2 uses only n binary consensus objects. Due to the challenge in Section 4.2.4, we explain how to deal with the case in which a transient fault changes the state of all objects in $BC[]$ to encode 'False was decided'. In this case, the algorithm cannot satisfy

the requirements of the multivalued consensus task (Definition 1.1). Therefore, our solution identifies such situations and informs the invoking algorithm via the return of the *transient error* symbol Ψ.

5.2. Algorithm description

We will now delve into a detailed presentation of Algorithm 2. Our adaptation of Algorithm 1 leverages the URB's ability to signal the termination of its broadcast through the hasTerminated() interface. This indication confirms the presence of at least one proposed value that every correct node stores within the array of proposals. Our solution utilizes the binary consensus objects to achieve uniform proposal selection upon this confirmation. Notably, within our variation on Algorithm 1, there exists a minimum of one index within the binary consensus object (alongside its corresponding proposal) for which every node will propose True. An illustrative example of this could be the index of the initial node to activate the binary consensus object, as it takes this step only after verifying the delivery of its proposal to all nodes.

Algorithm 2: Self-stabilizing non-blocking multivalued consensus; p_i's code.

```
13 variables: /* initialization is optional in the context of self-stabilization */
14 v ;                                                          /* local decision estimates */
15 proposals[0,..,n-1] ;                                        /* array of arriving proposals */
16 BC[0,..,n-1] ;                                               /* array of n binary consensus objects */
17 txDes ;                                                      /* URB transmission descriptor for decision sharing */
18 oneTerm ;                                                    /* true once at least one broadcast termination occurred */
19 macro k() = max({{-1} ∪ {x ∈ S(n-1) : (S(x) ∩ K) = S(x)}): where S(x) = {0,...,x} and K = ({k ∈ S(n-1) : O.BC[k] ≠ ⊥ ∧O.BC[k].result(k) = False}) ;
     /* k() is the max consecutive BC[] entry index with the decision False */
20 operation propose(v) do {if v ≠ ⊥ ∧ O = ⊥ then O.(v, proposals, BC, txDes, oneTerm) ← (v, [⊥,...,⊥], [⊥,...,⊥], ⊥, False)};
21 operation result() begin
22     if O = ⊥ then return ⊥;
23     else if O.v = ⊥ ∨ k ≥ n − 1 then return Ψ where k = k();
24     else if BC[k + 1] = ⊥ ∨ BC[k + 1].result(k + 1) ≠ True then return ⊥;
25     else if x = ⊥ then return Ψ else return x where x = O.proposals[k + 1];
26 do forever foreach O ≠ ⊥ with O's fields v, proposals, BC, and txDes do
27     if (v ≠ ⊥ ∧ (txDes = ⊥ ∨ hasTerminated(txDes)) then
28         oneTerm ← oneTerm ∨ (txDes ≠ ⊥ ∧ hasTerminated(txDes));
29         txDes ← urbBroadcast PROPOSAL(v)
       /* use either lines 30 to 31 or lines 32 to 33                                          */
30     if oneTerm ∧ k < n-1 ∧ BC[k+1]=⊥ ∧ (k=-1 ∨ BC[k].result(k) ≠ ⊥) then
31         binPropose(k + 1, proposals[k + 1] ≠ ⊥) where k = k()
32     if [oneTerm ∧ ∃ℓ : BC[ℓ] = ⊥] then /* invoke BC objects concurrently */
33         [for each k ∈ {0,...,n−1} : BC[k] = ⊥ do binPropose(k, proposals[k] ≠ ⊥)]
34 upon PROPOSAL(vJ) urbDelivered from p_j begin
35     if vJ ≠ ⊥ then
36         if O ≠ ⊥ ∧ O.proposals[j] = ⊥ then O.proposals[j] ← vJ;
37         else if O = ⊥ then
38             O.(v, proposals, BC, txDes) ← (vJ, [⊥,...,⊥], [⊥,...,⊥], ⊥);
39             O.proposals[j] ← vJ;
```

5.2.1. The propose(v) operation and variables

The operation propose(v) activates a multivalued object by initializing its fields (line 20). These are the proposed value, v, the array, $proposals[]$, of received proposals, where $proposals[j]$ stores the value received from $p_j \in \mathcal{P}$. Moreover, $BC[]$ is the array of binary consensus objects, where the active object $BC[j]$ determines whether the value in $proposals[j]$ should be the decided value. Also, $txDes$ is the transmission descriptor (initialized with $\perp$),[4] and $oneTerm$ is a boolean that indicates that at least one transmission has completed, which is initialized with False. Note that only v has its (immutable) value initialized in line 20 to its final value. The other fields are initialized to $\perp$ or an array of $\perp$ values; their values can be changed later on.

5.2.2. The result() operation

Algorithm 2 allows retrieving the decided value via result() (line 21). As in Algorithm 1, the array of n binary consensus objects encodes the proposal that will be used as the returned value by the multivalued consensus object. Specifically, the index of the entry in the array in which the first object decides True is the index of the proposal that should be used. As long as the multivalued consensus

[4] Recall that initialization is optional for self-stabilizing systems since we assume an arbitrary starting system state where any variable may hold any value.

object is not active (line 22), or there is no decision yet regarding the value of the multivalued consensus object (line 24), the operation returns $\perp$. As explained in Section 5.1.3, Algorithm 2 might enter an error state. In this case, result() returns Ψ (lines 23 and 25). The only case left (the else clause of line 25) is when there is a binary consensus object $O.BC[k]$ and a matching $O.proposals[k] \neq \perp$, where $k = \mathsf{k}()$. Here, due to the definition of k() (line 19), for any $k' \in \{0, \ldots, k\text{-}1\}$ the decided value of $O.BC[k']$ is False and $O.BC[k]$ decides True since the if-statement condition in line 24 must be false whenever line 25 is reached. Thus, $\mathsf{result}_i()$ returns the value of $O.proposals[k]$.

5.2.3. The do-forever loop

As explained above, Algorithm 2 has to make sure that the proposed value, v, arrives at all processors and records in $oneTerm$ the fact that at least once transmission has arrived. To that end, the if-statement in lines 27 to 29 lets p_i test the predicate ($txDes \neq \perp \wedge$ hasTerminated($txDes$)) and make sure that the transmission descriptor, $txDes$, refers to an active broadcast, *i.e.,* $txDes$ stores a descriptor that has not terminated (cf. hasTerminated()'s definition in Section 2.1.2). In detail, whenever $x.txDes \neq \perp$ holds, $\mathsf{hasTerminated}_i(txDes)$ holds eventually (URB-termination). Thus, the if-statement condition in line 27 holds eventually and p_i URB-broadcasts PROPOSAL(v) (line 29) after checking that $v \neq \perp$ (line 27). Note that p_i records the fact that at least one transmission was completed by assigning True to $oneTerm$ (line 28).

Upon the URB-delivery of p_i's PROPOSAL(vJ) at $p_j \in \mathcal{P}$, processor p_j considers the following two cases. If O is an active object, p_j merely checks whether $O.proposals[i]$ needs to be updated with vJ (line 36). Otherwise, O is initialized with vJ as the proposed value (line 39), similar to line 20.

Returning to the sender side, Algorithm 2 uses either lines 30 to 31, which sequentially access the array, $BC[]$, of binary consensus objects, or lines 32 to 33, which simply access all binary consensus objects concurrently. In both methods, processor p_i makes sure that at least one broadcast was completed, *i.e.,* $oneTerm =$ True (lines 28, 30, and 32). When following the sequential method (lines 30 to 31), the aim is to invoke binary consensus by calling $\mathsf{binPropose}(k+1, proposals[k+1] \neq \perp)$ (line 31), where $k = \mathsf{k}_i()$. This can only happen when the $(k+1)$-th object in $BC[]$ is not active, *i.e.,* $BC[k+1] = \perp$ and $BC[k+1]$ is either the first in $BC[]$, *i.e.,* $k =$ -1 or $BC[k]$ has terminated, *i.e.,* $BC[k].\mathsf{result}_i(k) \neq \perp$ (line 30).

The advantage of the sequential access method over the concurrent one is that it is more conservative with respect to the number of consensus objects being used since once the decision is True, there is no need to use more objects. The concurrent access method, marked in the boxed lines, encourages piggybacking of the messages related to binary concurrent objects. This is most relevant when every message (of binary consensus) can carry the data-loads of n proposals. In this case, the concurrent access method is more straightforward and potentially faster than the sequential one.

6. Correctness of Algorithm 2

Theorems 6.1 and 6.6 show that Algorithm 2 implements a self-stabilizing multivalued consensus. Definition 6.1 is used by Theorem 6.1. As explained in Section 2.3, for the sake of a simple presentation, we make the following assumptions. Let R be an Algorithm 2's execution, $p_i \in \mathcal{P}$, and O_i a multivalued consensus object.

Definition 6.1 *(Consistent multivalued consensus object).* Let R be an Algorithm 2's execution and O_i a multivalued consensus object, where $p_i \in \mathcal{P}$. Suppose in system state $c \in R$, either (i) $O_i = \perp$ is inactive or that (ii) $O_i \neq \perp$ is active, $O_i.v \neq \perp \wedge (k < n-1) \wedge ((BC[k{+}1] = \perp \vee BC[k{+}1].\mathsf{result}(k{+}1) = \perp \vee (BC[k{+}1].\mathsf{result}(k{+}1) = \mathsf{True} \wedge O_i.proposals[k{+}1] \neq \perp)))$, where $k = \mathsf{k}_i()$. In either case, we say that O_i is consistent in c.

Theorem 6.1 shows recovery from transient faults.

Theorem 6.1 *(Convergence). Let R be an Algorithm 2's execution. Suppose that there exists a correct processor $p_j \in \mathcal{P} : j \in Correct$, such that throughout R it holds that $O_j \neq \perp$ is an active multivalued consensus object. Moreover, suppose that any correct processor $p_i \in \mathcal{P} : i \in Correct$ calls* $\mathsf{result}_i()$ *infinitely often in R. Within n invocations of binary consensus, (i) the system reaches a state $c \in R$ after which* $\mathsf{result}_i() \neq \perp$ *holds. Specifically, (ii) O_i is either consistent (Definition of 6.1) or eventually reports the occurrence of a transient fault,* i.e., $\mathsf{result}_i() = \Psi$.

Proof of Theorem 6.1. Lemmas 6.2 and 6.5 imply the proof.

Lemma 6.2. *Invariant (i) holds,* i.e., $\mathsf{result}_i() \neq \perp$ *holds in c.*

Proof of Lemma 6.2. Suppose, towards a contradiction, that c does not exist. Specifically, let R' be the longest prefix of R that includes no more than n invocations of binary consensus. The proof of Invariant (i) needs to show that the system reaches a contradiction by showing that $c \in R'$. To that end, arguments (1) to (3), as well as Claims 6.3 to 6.4, show the needed contradiction.

Argument (1) implies that it is enough to show that the if-statement in line 24 cannot hold eventually.

Argument (1) *The if-statement conditions in lines 22, 23, and 25 do not hold for p_j throughout R.* By the theorem assumption that $O_j \neq \perp$ is an active multivalued consensus object throughout R, we know that the if-statement condition in line 22 cannot hold. Moreover, by the assumption that c does not exist, we know that the if-statement conditions in lines 23 and 25 do not hold for any (correct) p_i throughout R.

Argument (2) *The invariant $O_j.v \neq \perp$ holds throughout R.* Since the if-statement condition in line 23 does not hold, $O_j.v \neq \perp$ holds in R's starting system state. Moreover, only lines 20, 36, and 39 change the value of $O_j.v$ but this happens only after testing that the assigned value is not $\perp$ (lines 20 and 35).

Argument (3) *R has a suffix in which all correct processors $p_i \in \mathcal{P}$ are active.* Since p_j is active and $O_j.v \neq \perp$ holds throughout R, the if-statement condition in line 27 holds eventually since either $txDes = \perp$ or $\mathsf{hasTerminated}(txDes)$ holds eventually due to the URB-termination property. By line 29, p_j broadcasts the $\langle v \rangle$ message to all correct processors p_i. By the URB-termination property, p_i receives $\langle v \rangle$ and by lines 36 to 39, processor p_i is active.

Argument (4) *$\forall i, j \in Correct : O_i.proposals[j] \neq \perp \wedge O_i.txDes \neq \perp \wedge O_i.oneTerm = \mathsf{True}$ holds eventually.* By URB-termination, $\mathsf{hasTerminated}(O_i.txDes)$ holds eventually. Once that happens, the if-statement condition in line 27 holds (due to arguments (2) and (3)) and $O_i.txDes = \perp$ cannot hold (line 29). By Argument (2), $O_i.v \neq \perp$. Thus, p_i eventually URB-broadcasts PROPOSAL($O_i.v$). Once p_i self-delivers this message, line 36 assigns v to $O_i.proposals[i]$ due to the assumption that $O_i \neq \perp$ throughout R. We can now repeat the reasoning that $\mathsf{hasTerminated}(O_i.txDes)$ holds eventually and thus the if-statement condition in line 27 holds. Thus, $O_i.oneTerm = \mathsf{False}$ does not hold eventually (line 28). By Argument (3), the same holds for p_j. Specifically, p_j eventually URB-broadcasts PROPOSAL($O_j.v$). Once p_i URB-delivers this message from p_j, p_i's state can possibly change, even in the case that $O_i \neq \perp$, cf. lines 36 and 39.

Claim 6.3. *The if-statement condition in lines 30 and 32 can only hold at most n times for any $p_i \in \mathcal{P} : i \in Correct$.*

Proof of Claim 6.3. The if-statement condition in line 32 can only hold at most once due to line 33. Thus, the rest of the proof focuses on the if-statement in lines 30 to 31.

By the proof of Argument (4), eventually, the system reaches a state, $c' \in R$, in which $O_i.txDes \neq \perp \wedge O_i.proposals[j] \neq \perp \wedge O_i.oneTerm = \mathsf{True}$ holds. Note that the if-statement condition in line 30 holds whenever $k = -1$. Arguments (5) and (6) assume that $k > -1$ and consider the cases in which $O_i.BC[k+1] \neq \perp$ holds and does not hold, respectively, where $k = \mathsf{k}()$. Argument (7) shows that the if-statement condition in line 30 can hold at most n times.

Argument (5) *Suppose that $k > -1 \wedge O_i.BC[k+1] \neq \perp$ holds. Eventually, either $k_i() < n-1$ does not hold or the if-statement condition in line 30 holds.* $BC[k+1].\mathsf{result}_i(k+1) \neq \perp$ holds eventually due to the termination property of binary consensus objects.

In case $BC[k+1].\mathsf{result}_i(k+1) = \mathsf{True}$, we know that $\mathsf{result}_i() \neq \perp$ holds due to the definition of $\mathsf{k}()$ (line 19). However, this implies a contradiction with the assumption made at the start of this lemma's proof.

In case $BC[k+1].\mathsf{result}_i(k+1) = \mathsf{False}$ holds, the if-statement condition in line 30 holds in c' if $k+1 < n$-1 and $O_i.BC[k+1] = \perp$. In case the former predicate holds and the latter does not, we can repeat the reasoning above for at most n times until either the former does not hold or both predicates hold. In either case, the proof of the argument is done.

Argument (6) *Suppose that $k > -1 \wedge O_i.BC[k+1] = \perp$ holds. Eventually, either $k_i() < n-1$ does not hold or the if-statement condition in line 30 holds.* The if-statement condition in line 30 holds if $BC[k].\mathsf{result}_i(k) \neq \perp$ holds. Since $k > -1$, the reasoning in the proof of Argument (5), which shows that $BC[k+1].\mathsf{result}_i(k+1) \neq \perp$ holds, can be used for showing that $BC[k].\mathsf{result}_i(k) \neq \perp$ holds eventually.

Argument (7) *Within n invocations of $\mathsf{binPropose}_i()$, the if-statement condition in line 30 does not hold.* Suppose that the if-statement condition in line 30 holds. In line 31, p_i invokes the operation $\mathsf{binPropose}_i(k+1, O_i.proposals[k+1] \neq \perp)$ of the $(k+1)$-*th* binary consensus object. This invocation changes p_i's state, such that $O_i.BC[k+1] = \perp$ does not hold any longer (because the $\mathsf{binPropose}_i()$ operation initializes the state of $O_i.BC[k+1]$). Since $BC[]$ has n entries, there could be at most n such invocations until the system reaches $c'' \in R$, after which the if-statement condition in line 30 cannot hold. □

Claim 6.4. *Once the if-statement condition in line 30 (or 32) does not hold, also the if-statement condition in line 24 does not hold.*

Proof of Claim 6.4. Since if-statement condition in line 30 does not hold, we know that $BC_i[k+1] = \perp$ does not hold, see Argument (5) of Claim 6.3. In the case of line 32, the same holds in a straightforward manner. By BC-termination, $BC[k+1].\mathsf{result}_i(k+1) \neq \perp$ holds eventually. Since $\forall p_x \in \mathcal{P} : O_i.BC[x] \neq \perp \wedge BC[x].\mathsf{result}_i(x) = \mathsf{False}$ implies a contradiction with Argument (1), we know that $BC_i[k+1].\mathsf{result}(k+1) \neq \mathsf{True}$ cannot hold. □

Lemma 6.5. *Invariant (ii) holds,* i.e., *O_i is either consistent or* $\mathsf{result}_i() = \Psi$.

Proof of Lemma 6.5. Recall that the theorem assumes that O_i is an active object throughout R. The argument is implied by Definition 6.1 and lines 22 to 25.

In detail, line 22 handles the case in which $O_i = \perp$. Suppose that $O_i \neq \perp$ is active, which indicates that an inconsistency was detected. Line 23 handles the case in which $O_i.v \neq \perp \wedge k < n-1$ does not hold by returning Ψ, where $k = \mathsf{k}_i()$, which indicates that an inconsistency was detected. Line 24 allows the case in which $BC[k+1] = \perp \vee BC[k+1].\mathsf{result}(k+1) = \perp$ (note that the case of $BC[k+1].\mathsf{result}(k+1) = \mathsf{False}$ does not exist due to the definition of $\mathsf{k}()$ in line 19). This case is allowed since it is consistent, see Definition 6.1. Line 25 handles the case in which $(BC[k+1].\mathsf{result}(k+1) = \mathsf{True} \wedge O_i.proposals[k+1] \neq \perp)$ does not hold by returning Ψ, which indicates that an inconsistency was detected. □

Theorem 6.6 demonstrates the Closure property and uses the term authentic executions (Definition 6.2).

Definition 6.2 *(Complete execution with respect to* propose() *invocations).* Let R be an execution of Algorithm 2 that starts in $c \in R$. We say that c is *completely free* of PROPOSAL(−) messages if (i) the communication channels do not include PROPOSAL(−) messages, and (ii) for any non-failing $p_i \in \mathcal{P}$, there is no active multivalued consensus object $O_i = \perp$ in c.

Let $c_s \in R$ be the system state that is: (a) completely free of PROPOSAL(−), and (b) it appears in R immediately before a step that includes p_i's invocation of propose(−) (lines 20) in which O_i becomes active (rather than due to the arrival of a PROPOSAL(−) message in lines 35 to 39). In this case, we say that p_i's invocation is *authentic*. Suppose that p_i sends a PROPOSAL(−) message after c_s. In this case, we say that PROPOSAL(−) is an *authentic message transmission*.

An arrival of PROPOSAL(−) to $p_j \in \mathcal{P}$ (lines 20) is said to be authentic if it is due to an authentic message transmission. Suppose that p_j actives $O_j = CS_j[s]$ (line 39) due to an authentic arrival (rather than an invocation of the propose(−) operation). In this case, we also say that p_j's *invocation is authentic*. We complete the definitions of authentic transmissions, arrivals, and invocations by applying the transitive closures of them.

Suppose that any invocation in R of $\text{propose}_k(-) : p_k \in \mathcal{P}$ is authentic as well as the transmission and reception of PROPOSAL(−) messages from or to p_k. In this case, we say that R is an *authentic execution*.

Theorem 6.6 shows that Algorithm 2 satisfies the task requirements (Section 2.2).

Theorem 6.6 *(Closure). Let R be an authentic execution of Algorithm 2. The system demonstrates in R the construction of a multivalued consensus object.*

Proof of Theorem 6.6. Validity holds since only the user input is stored in the field v (line 20), which is then URB-broadcast (line 29), stored in the relevant entry of *proposals* (lines 36 to 39), and returned as the decided value (line 25). Moreover, any value in v can be traced back to an invocation of propose(v) since R is authentic.

Lemma 6.7 demonstrates termination and agreement.

Lemma 6.7. *Let $a_i \in R$ be the first step in R that includes an invocation, say, by $p_i \in \mathcal{P}$ of $\text{propose}_i(v_i)$. Suppose that $v_i \neq \perp$ holds in any system state of R. There exists $v \notin \{\perp, \Psi\}$, such that for every correct $p_j \in \mathcal{P}$ it holds that $\text{result}_j()$ returns v within n invocations of binary consensus.*

Proof of Lemma 6.7. Arguments (1) to (7) imply the proof.

Argument (1) $O_i.(v, proposals, BC, txDes, oneTerm) = (v, [\perp, \ldots, \perp], [\perp, \ldots, \perp], \perp, \text{False})$ *holds immediately after* a_i. We show that the if-statement condition in line 20 holds immediately before a_i. Recall the theorem assumption that $v_i \neq \perp$ holds in R. By the assumption that R is authentic, we know that $O_i = \perp$ holds immediately before a_i. Therefore, p_i assigns $(v_i, [\perp, \ldots, \perp], [\perp, \ldots, \perp], \perp, \text{False})$ to $O_i.(v_i, proposals_i, BC_i, txDes_i, oneTerm_i)$ (line 20).

Argument (2) *Eventually* PROPOSAL(v_i) *messages are URB broadcast and* $oneTerm_i$ *holds.* The proof has to consider the case in which hasTerminated($O_i.txDes$) holds in the starting system state due to a transient fault rather than an actual broadcast during the system execution. By URB-termination, hasTerminated($O_i.txDes$) does not hold eventually. Since $O_i.v \neq \perp$ (by the lemma assumption), the if-statement condition in line 27 holds and p_i URB-broadcasts PROPOSAL($O_i.v$). By applying again the same argument, the assignment in line 28 makes sure that $oneTerm_i = \text{True}$.

Argument (3) *For any $p_x \in \mathcal{P} : x \in Correct$, eventually $O_x.proposals[i] \neq \perp$ and $O_x.proposals[x] \neq \perp$ hold.* By URB-termination, every correct processor, p_x, eventually URB-delivers Argument (2)'s PROPOSAL(v_i) message. By the assumption that $v_i \neq \perp$ holds in any system state of R, the if-statement condition in line 35 holds (even if p_x has invoked $\text{propose}_x(v_x)$ before this URB delivery).

In case there was no earlier invocation of $\text{propose}_x(v_x)$, the assignment $O_x.v \leftarrow v_i$ occurs due to line 39 (otherwise, a similar assignment occurs due to line 36). Moreover, due to the reasons that cause p_i URB broadcasts in Argument (2), also p_x URB broadcasts PROPOSAL($v' \neq \perp$) messages. Upon the URB delivery of p_x message to itself, the $O_x.proposals[x] \leftarrow v' \neq \perp$ assignment occurs (line 36). (Note that this time, the if-statement condition in line 36 must hold since $O_x \neq \perp$.)

Argument (4) *The if-statement condition in lines 30 and 32 hold eventually.* Since $oneTerm_i$ holds eventually (Argument (2)), the if-statement condition in line 32 holds eventually. Also, the fact that $O_x.proposals[x] \neq \perp$ (Argument (3)) and URB-termination imply that eventually, in p_x's do-forever loop, the if-statement condition in line 30 holds. In detail, since R is an authentic execution, $k = \text{-}1 \wedge BC[k+1] = \perp$ holds in R's second state, where $k = k_x()$.

Let $S(z) = \{0, \ldots, z\}$. The proof of Argument (5) shows $\exists y \in S(n\text{-}1), \forall x \in Correct$, p_x invokes $\text{binPropose}_x()$ at most y times and it observes that $O_x.BC[k] \neq \perp : k \in S(y\text{-}1)$.

Argument (5) *The Termination property holds.* By line 33, the if-statement condition in line 32 can hold at most once. The if-statement condition in line 30 cannot hold for more than n times due to the $(k < n\text{-}1)$ clause. Thus, the termination property is implied.

Let $r(j) = [x(0), \ldots, x(n\text{-}1)]$, such that $x(k) = \perp$ if $BC_j[k] = \perp$. Otherwise, $x(k) = BC_j[k].\text{result}(k)$. Moreover, let $S = \{[\perp, \ldots, \perp], [\text{False}, \ldots, \text{False}, \perp, \ldots, \perp], [\text{False}, \ldots, \text{False}, \text{True}, \perp, \ldots, \perp], [\text{False}, \ldots, \text{False}, \text{True}]\}$. For the case of using lines 30 to 31, the proof of Argument (6) shows $\forall p_j \in \mathcal{P} : r(j) \in S$, *i.e.,* sequential invocation of binPropose().

Argument (6) *For the case of using lines 30 to 31, $r(j) \in S$ holds.* Due to lines 19 and 30 as well as the agreement property of binary consensus objects and the fact that R is authentic, we know that eventually, all non-failing nodes must observe the same results from their consensus objects. Specifically for the case of lines 30 to 31, it holds that $|\bigcup_{j \in Correct}\{r(j)\}| = 1$. Also, at any time, $r(j)$ can only

include a finite number (perhaps empty but with no more than n-1) of False values that are followed by at most one True value and the only $\perp$-values (if space is left), *i.e.,* $r(j) \in S$.

Argument (7) *The Agreement property holds.* Since there is no system node $p_j \in \mathcal{P}$ invoking $\mathsf{binPropose}_j(k_j+1, O_x.proposals[k_j+1] \neq \perp)$ (lines 30 and 32), before it had assured the safe URB delivery of $O_x.txDes$'s transmission, we know that eventually, at least one element of $r(j)$ is True. Thus, by the agreement property of binary consensus, every p_x eventually calculates the same value of $k_j()$, such that $BC[k_j()].\mathsf{result}_x(k_j()+1) = \mathsf{True}$. This implies the agreement property since $\mathsf{result}_x()$ returns $O_x.proposals[k_j()+1]$ for any non-failing $p_x \in \mathcal{P}$ (line 25). □

Lemma 6.8 demonstrates the property of integrity.

Lemma 6.8. *Suppose that* $\exists v' \notin \{\perp, \Psi\} : \exists p_j \in \mathcal{P} : \exists c' \in R : \mathsf{result}_j() = v'$ *in* c'. $\nexists c'' \in R : \mathsf{result}_j() = v''$ *in* c'', *such that* $v' \neq v''$.

Proof of Lemma 6.8. The proof is by contradiction. Suppose that $c'' \in R$ exists and, without the loss of generality, c' appears before c'' in R. Since R is authentic and $c' \in R$ exists. Then, there is a $p_k \in \mathcal{P} : k \in S(n\text{-}1)$, such that for any $p_j \in \mathcal{P}$, there is a system state c'_j that appears in R not after c' in which for any $k' \in S(n\text{-}1)$ it holds that $BC[k'].\mathsf{result}_j(k') = \mathsf{False}$ for the case of $k' < k$ and $BC[k].\mathsf{result}_j(k') = \mathsf{True}$ for the case of $k' = k$. This is due to the definition of k() (line 19). Note that in any system state that follows c'_j, the value of $k = \mathsf{k}_j()$ does not change due to the integrity of binary consensus objects. Therefore, $\mathsf{result}_j()$ must return the value of $O_j.proposal[k]$ in any system state that follows c'_j. Since line 36 does not allow any change in the value of $O_j.proposal[k]$ between c' and c'', it holds that $v' = v''$. Thus, the proof reached a contradiction, and the lemma is true. □

7. Conclusions

We have demonstrated how a non-self-stabilizing algorithm for multivalued consensus, proposed by Mostéfaoui, Raynal, and Tronel [12], can be transformed into a self-stabilizing one that efficiently recovers from transient faults. Notably, our solution incurs only a bounded number of binary consensus invocations, distinguishing it from earlier approaches that either utilize an unbounded number of binary consensus objects [12] or rely on blocking mechanisms [13]. As a result, our proposed transformation method offers a more appealing and practical solution than the previously studied algorithm, regardless of the presence or absence of transient faults. As applications, we have found our self-stabilizing total-order message delivery and self-stabilizing emulator for state-machine replication [2] to benefit significantly from our solution as well as the proposed transformation method. We encourage researchers and practitioners to adopt our approaches when designing distributed systems that require efficient recovery from transient faults.

CRediT authorship contribution statement

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

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgements

We thank Daniel Kem, Daniel Karlberg, and Amanda Sjöö for valuable 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] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing total-order broadcast, in: Stabilization, Safety, and Security of Distributed Systems - 24th International Symposium, SSS, in: Lecture Notes in Computer Science, vol. 13751, Springer, 2022, pp. 358–363.
[3] M. Raynal, Fault-Tolerant Message-Passing Distributed Systems - an Algorithmic Approach, Springer, 2018.
[4] V. Hadzilacos, S. Toueg, A modular approach to fault-tolerant broadcasts and related problems, Tech. Rep., Cornell Univ, Ithaca, NY, 1994.
[5] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing uniform reliable broadcast, in: Networked Systems NETYS, in: Lecture Notes in Computer Science, vol. 12129, Springer, 2020, pp. 296–313.
[6] L. Lamport, The part-time Parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.
[7] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.
[8] R. van Renesse, D. Altinbuken, Paxos made moderately complex, ACM Comput. Surv. 47 (3) (2015) 42:1–42:36.

[9] P.D. Ezhilchelvan, A. Mostéfaoui, M. Raynal, Randomized multivalued consensus, in: Object-Oriented Real-Time Distributed Computing (ISORC 2001), 2001, pp. 195–200.
[10] G. Liang, N.H. Vaidya, Error-free multi-valued consensus with Byzantine failures, in: ACM Principles of Distributed Computing, PODC, 2011, pp. 11–20.
[11] A. Babaee, M. Draief, Distributed multivalued consensus, Comput. J. 57 (8) (2014) 1132–1140.
[12] A. Mostéfaoui, M. Raynal, F. Tronel, From binary consensus to multivalued consensus in asynchronous message-passing systems, Inf. Process. Lett. 73 (5–6) (2000) 207–212.
[13] J. Zhang, W. Chen, Bounded cost algorithms for multivalued consensus using binary consensus instances, Inf. Process. Lett. 109 (17) (2009) 1005–1009.
[14] T. Albouy, D. Frey, R. Gelles, C. Hazay, M. Raynal, E.M. Schiller, F. Taïani, V. Zikas, Towards optimal communication Byzantine reliable broadcast under a message adversary, in: DISC 2024, 2023, CoRR, arXiv:2312.16253 [abs].
[15] Y. Afek, E. Gafni, S. Rajsbaum, M. Raynal, C. Travers, The k-simultaneous consensus problem, Distrib. Comput. 22 (3) (2010) 185–195.
[16] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[17] S. Dolev, Self-Stabilization, MIT Press, 2000.
[18] K. Altisen, S. Devismes, S. Dubois, F. Petit, Introduction to Distributed Self-Stabilizing Algorithms, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2019.
[19] P. Blanchard, S. Dolev, J. Beauquier, S. Delaët, Practically self-stabilizing Paxos replicated state-machine, in: NETYS, in: LNCS, vol. 8593, Springer, 2014, pp. 99–121.
[20] 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.
[21] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing indulgent zero-degrading binary consensus, in: 22nd Distributed Computing and Networking ICDCN, 2021, pp. 106–115.
[22] S. Dolev, T. Petig, E.M. Schiller, Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks, Algorithmica 85 (1) (2023) 216–276.
[23] C. Georgiou, R. Gustafsson, A. Lindhé, E.M. Schiller, Self-stabilization overhead: a case study on coded atomic storage, in: NETYS, in: Lecture Notes in Computer Science, vol. 11704, Springer, 2019, pp. 131–147.
[24] S. Dolev, E. Schiller, Communication adaptive self-stabilizing group membership service, IEEE Trans. Parallel Distrib. Syst. 14 (7) (2003) 709–720.
[25] S. Dolev, E. Schiller, Self-stabilizing group communication in directed networks, Acta Inform. 40 (9) (2004) 609–636.
[26] 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.
[27] M. Canini, I. Salem, L. Schiff, E.M. Schiller, S. Schmid, *Renaissance*: a self-stabilizing distributed SDN control plane using in-band communications, J. Comput. Syst. Sci. 127 (2022) 91–121.
[28] C. Georgiou, O. Lundström, E.M. Schiller, Self-stabilizing snapshot objects for asynchronous failure-prone networked systems, in: Networked Systems, NETYS, 2019, pp. 113–130.
[29] S. Dolev, R.I. Kat, E.M. Schiller, When consensus meets self-stabilization, J. Comput. Syst. Sci. 76 (8) (2010) 884–900.
[30] 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.
[31] C. Georgiou, M. Raynal, E.M. Schiller, Self-stabilizing byzantine-tolerant recycling, in: Stabilization, Safety, and Security of Distributed Systems - 25th International Symposium, SSS, in: Lecture Notes in Computer Science, vol. 14310, Springer, 2023, pp. 518–535.
[32] R. Duvignau, M. Raynal, E.M. Schiller, Self-stabilizing byzantine multivalued consensus, in: Proceedings of the 25th International Conference on Distributed Computing and Networking, ICDCN, ACM, 2024, pp. 12–21.
[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] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing multivalued consensus in asynchronous crash-prone systems, in: EDCC, IEEE, 2021, pp. 111–118.
[36] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing multivalued consensus in asynchronous crash-prone systems, CoRR, arXiv:2104.03129 [abs], 2021.
[37] R. Guerraoui, M. Raynal, The information structure of indulgent consensus, IEEE Trans. Comput. 53 (4) (2004) 453–466.
[38] M. Jayaram, G. Varghese, Crash failures can drive protocols to arbitrary states, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, ACM, 1996, pp. 247–256.