Self-stabilizing multivalued consensus in the presence of Byzantine faults and asynchrony

Theoretical Computer Science 1039 (2025) 115184

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Self-stabilizing multivalued consensus in the presence of Byzantine faults and asynchrony ☆

Romaric Duvignau [a], Michel Raynal [b], Elad Michael Schiller [a,*]

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

ARTICLE INFO

Keywords:
Multivalued consensus
Byzantine fault tolerance
Self-stabilization
Asynchronous message-passing systems

ABSTRACT

Consensus, abstracting myriad problems in which processes must agree on a single value, is one of the most celebrated problems of fault-tolerant distributed computing. Consensus applications include fundamental services for the Cloud and Blockchain environments, and in such challenging environments, malicious behaviors are often modeled as adversarial Byzantine faults.

At OPODIS 2010, Mostéfaoui and Raynal (in short, MR) presented a Byzantine-tolerant solution to consensus in which the decided value cannot be proposed only by Byzantine processes. MR has optimal resilience coping with up to $t < n/3$ Byzantine nodes over n processes. MR provides this multivalued consensus object (which accepts proposals taken from a finite set of values), assuming the availability of a single binary consensus object (which accepts proposals taken from the set $\{0, 1\}$).

This work, which focuses on multivalued consensus, aims to design an even more robust solution than MR. Our proposal expands MR's fault-model with self-stabilization, a vigorous notion of fault-tolerance. In addition to tolerating Byzantine, self-stabilizing systems can automatically recover after *arbitrary transient-faults* occur. These faults represent any violation of the assumptions according to which the system was designed to operate (provided that the algorithm code remains intact).

To the best of our knowledge, we propose the first self-stabilizing solution for multivalued consensus for asynchronous message-passing systems prone to Byzantine failures. Our solution has an $\mathcal{O}(t)$ stabilization time from arbitrary transient faults.

1. Introduction

We present in this work a novel self-stabilizing algorithm for multivalued consensus in signature-free asynchronous message-passing systems that can tolerate Byzantine faults. We provide rigorous correctness proofs to demonstrate that our solution is correct and outperforms all previous approaches in terms of its fault tolerance capabilities. We also analyze its recovery time. Compared to existing solutions, our proposed algorithm represents a significant advancement in the state of the art, as it can effectively handle a broader range of faults, including both benign and malicious failures, as well as arbitrary, transient, and possibly unforeseen violations

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

* Corresponding author.
E-mail addresses: duvignau@chalmers.se (R. Duvignau), michel.raynal@irisa.fr (M. Raynal), elad@chalmers.se (E.M. Schiller).

https://doi.org/10.1016/j.tcs.2025.115184
Received 22 April 2024; Received in revised form 5 January 2025; Accepted 13 March 2025

Available online 17 March 2025

of the assumptions according to which the system was designed to operate. Our proposed solution can hence further facilitate the design of new fault-tolerant building blocks for distributed systems.

Glossary: For the reader's convenience, all abbreviations are listed below. **ACAF**: asynchronous cycles (while assuming execution fairness); **BFT**: (non-stabilizing) Byzantine fault-tolerant; **BRB**: Byzantine-tolerant Reliable Broadcast; **BV-broadcast**: Binary-values broadcast; **CRWF**: communication rounds (without assuming execution fairness); **DRS**: SSBFT BRB by Duvignau, Raynal, and Schiller [1]; **GMRS**: SSBFT MMR by Georgiou, Marcoullis, Raynal, and Schiller [2,3] for binary consensus and BV-broadcast; **MR**: the studied solution by Mostéfaoui and Raynal [4]; **MVC**: multivalued consensus (Section 1.1); **SSBFT**: self-stabilizing Byzantine fault-tolerant; **VBB**: Validated Byzantine Broadcast, *e.g.,* BFT and SSBFT ones in Algorithms 1 and 3, resp.

1.1. Task requirements and fault models

Multivalued consensus (MVC) The consensus problem is one of the most challenging tasks in fault-tolerant distributed computing. The problem definition is relatively simple. It assumes that each non-faulty process advocates for a single value from a given set V. The problem of *Byzantine-tolerant Consensus* (BC) requires *BC-completion* (**R1**), *i.e.,* all non-faulty processes decide a value, *BC-agreement* (**R2**), *i.e.,* no two non-faulty processes can decide different values, and *BC-validity* (**R3**), *i.e.,* if all non-faulty processes propose the same value $v \in V$, only v can be decided. When the set, V, from which the proposed values are taken is $\{0, 1\}$, the problem is called binary consensus and, otherwise, MVC. We study MVC solutions that assume access to a single binary consensus object. In this paper, we define an object as implementing an abstraction—specified by a set of properties—that enables the solution of a problem (see Raynal [10]).

Byzantine fault-tolerance (BFT) Lamport et al. [5] say that a process commits a *Byzantine* failure if it deviates from the algorithm's instructions, for example, by deferring or omitting messages sent by the algorithm or by sending *fake messages*—messages that are forged or fabricated by a node that was maliciously captured and might be, at any time, forced not to follow the proposed solution. Honest nodes, on the other hand, are expected to faithfully follow the proposed algorithm and do not send such fake messages. Such malicious behaviors include crashes resulting from hardware or software malfunctions and coordinated malware attacks. To safeguard against such attacks, Mostéfaoui and Raynal [4], MR, from now on, suggested the *BC-no-intrusion* (**R4**) validity requirement (aka *intrusion-tolerance*). Specifically, the decided value cannot be a value that was proposed *only* by faulty processes. Also, an error symbol is returned instead when deciding on a value is impossible. For the sake of deterministic solvability [6,5,7,8], we assume that there are at most $t < n/3$ Byzantine processes in the system, where n is the total number of processes. It is also well-known that no deterministic (multivalued or binary) consensus solution exists for asynchronous systems in which at least one process may crash (or be Byzantine) [9]. Our self-stabilizing MVC algorithm circumvents this impossibility by assuming that the system is enriched with a binary consensus object, as in the studied (non-self-stabilizing) solution by MR [4], *i.e.,* reducing MVC to binary consensus.

Definition 1.1. The **BFT Multivalued Consensus** (MVC) problem requires BC-completion (R1), BC-agreement (R2), BC-validity (R3), and BC-no-Intrusion (R4).

Self-stabilization We study an asynchronous message-passing system with no guarantees of communication delay, and the algorithm cannot explicitly access the local clock. Our fault model includes (undetectable) Byzantine failures. In addition, we aim to recover from *arbitrary transient-faults*, *i.e.,* any temporary violation of assumptions according to which the system was designed to operate. This includes the corruption of control variables, such as the program counter and message payloads, and operational assumptions, such as that there are more than t faulty processes. We note that non-self-stabilizing BFT systems do not consider recovery after the occurrence of such violations. Since the occurrence of these failures can be arbitrarily combined, we assume these transient-faults can alter the system state in unpredictable ways. In particular, when modeling the system, Dijkstra [11] assumes that these violations bring the system to an arbitrary state from which a *self-stabilizing system* should recover [12,13]. That is, Dijkstra requires (i) recovery after the last occurrence of a transient-fault and (ii) that once the system has recovered, it must never violate the task requirements. Arora and Gouda [14] refer to the former requirement as *Convergence* and the latter as *Closure*.

Definition 1.2. A **Self-Stabilizing Byzantine Fault-Tolerant** (SSBFT) MVC algorithm satisfies the requirements of Definition 1.1 within the execution of a finite number of steps following the last transient fault, leaving the system in an arbitrary state.

1.2. Related work

Since the seminal work of Lamport, Shostak, and Pease [5] four decades ago, BFT consensus has been an active research subject, see [15] and references therein. The recent rise of distributed ledger technologies, *e.g.,* [16], brought phenomenal attention to the subject. We aim to provide a higher degree of dependability than existing solutions.

Ben-Or, Kelmer, and Rabin [17] were the first to show that BFT MVC can be reduced to binary consensus. Correia, Neves, and Veríssimo [18,19] later established the connection between intrusion tolerance and Byzantine resistance. These ideas form the basis of the MR algorithm [4]. MR is a leaderless consensus algorithm [20], and as such, it avoids the critical weakness of leader-based algorithms [21] when the leader is slow and delays termination. The self-stabilizing solutions for MVC are only crash-tolerant [22–26], whereas the existing BFT solutions are not self-stabilizing [10]. For example, the recent self-stabilizing crash-tolerant MVC in [24]

solves a less challenging problem than the SSBFT problem studied here since it does not account for malicious behaviors. Mostéfaoui, Moumen, and Raynal [27] (MMR in short) presented BFT algorithms for solving binary consensus using common coins, of which GMRS [2,3] recently introduced a self-stabilizing variation that satisfies the safety requirements, *i.e.,* agreement and validity, with an exponentially high probability that depends only on a predefined constant, which safeguards safety. The related work also includes SSBFT state-machine replication by Binun et al. [28,29] for synchronous systems and Dolev et al. [30] for practically-self-stabilizing partially-synchronous systems. Both Binun et al. and Dolev et al. study another problem for another system setting. In [31], the problems of SSBFT topology discovery and message delivery were investigated. Self-stabilizing atomic memory under a semi-Byzantine adversary is studied in [32].

This work's extended abstract and technical report versions appear in [33] and [34], respectively.

1.3. A brief overview of the MR algorithm

The MR algorithm assumes that all (non-faulty) processes eventually propose a value. Upon the proposal of value v, the algorithm utilizes a Validated Byzantine Broadcast protocol, known as VBB, to enable each process to reliably deliver v. The VBB-delivered value could be either the message, v, which was VBB-broadcast, or $\perp$ when v could not be validated. For a value v to be valid, it must be VBB-broadcast by at least one non-faulty process.

Following the VBB-delivery from at least $n - t$ different processes, MR undergoes a local test, detailed in Section 3.2. If at least one non-faulty process passes this test, it implies that all non-faulty processes can ultimately agree on a single value proposed by at least one non-faulty process. Therefore, the MR algorithm employs Byzantine-tolerant binary consensus to reach consensus on the outcome of the local test. Suppose the agreed value indicates the existence of at least one non-faulty process that has passed the test. In that case, each non-faulty process waits until it receives at least $n - 2t$ VBB-arrivals with the same value, v, which is the decided value in this instance of multivalued consensus. If the agreed value does not represent such an indication, the MR algorithm reports its inability to decide in this MVC invocation. For further information, please refer to Section 3.

1.4. Our SSBFT variation on MR

This work considers transformers that take algorithms as input and output their self-stabilizing variations. For example, Duvignau, Raynal, and Schiller [1] (referred to as DRS) proposed a transformation for converting the Byzantine Reliable Broadcast (referred to as BRB) algorithm, originally introduced by Bracha and Toueg [35], into a Self-Stabilizing BFT (in short, SSBFT) variation. Another transformation, proposed by Georgiou, Marcoullis, Raynal, and Schiller [2,3] (referred to as GMRS), presented the SSBFT variation of the BFT binary consensus algorithm of MMR.

Our transformation builds upon the works of DRS and GMRS when transforming the (non-stabilizing) BFT MR algorithm into its self-stabilizing variation. The design of SSBFT solutions requires addressing considerations that BFT solutions do not need to handle, as they do not consider transient faults.

For instance, MR uses a (non-self-stabilizing) BFT binary consensus object, denoted as obj. In MR, obj returns a value proposed by at least one non-faulty process, corresponding to a test result (as mentioned in Section 1.3 and detailed in Section 4.1.1). However, a single transient fault can change obj's value from False, *i.e.,* not passing the test, to True. Such an event would cause MR, not designed to tolerate transient faults, to wait indefinitely for never-sent messages. Our solution addresses this issue by carefully integrating GMRS's SSBFT binary-values broadcast (in short, BV-broadcast). This subroutine ensures that obj's value is proposed by at least one non-faulty node, even in the presence of transient faults.

The vulnerability of consensus objects to corruption by transient faults holds true regardless of whether we consider binary or multivalued consensus (MVC). Thus, our SSBFT MVC solution is required to decide even when starting from an arbitrary state. To achieve this, our proof of correctness demonstrates that our solution always terminates. We borrow from GMRS the concept of *consensus object recycling*, which refers to reusing the object (space in the local memory of all non-faulty processes) for a later MVC invocation. Even when starting from an arbitrary state, the proposed solution decides on a value that is eventually delivered to all non-faulty processes, albeit potentially violating safety due to transient faults. Then, utilizing GMRS's subroutine for recycling consensus objects, the MVC object is recycled. Starting from a post-recycling state, the MVC object guarantees both safety and liveness for an unbounded number of invocations. This is one of the principal arguments behind our correctness proof.

We clarify that GMRS's recycling subroutine relies on synchrony assumptions. To mitigate the impact of these assumptions, a single recycling action can be performed for a batch of δ objects, where δ is a predefined constant determined by the memory available for consensus objects. This approach allows asynchronous networking in communication-intensive components, such as the consensus objects, while the synchronous recycling actions occur according to the predefined load parameter, δ.

We want to emphasize to the reader that although our solution builds upon the prior works of DRS [1] and GMRS [2,3]. While these serve as building blocks, they address problems that differ from those considered in this work. Their constructions rely on code transformations from non-self-stabilizing to self-stabilizing algorithms. However, achieving the required self-stabilizing properties in our setting demands a careful integration of SSBFT components and a rigorous analysis of the resulting transformed algorithms. This integration process is not directly derivable from the DRS and GMRS transformations. The self-stabilization challenges specific to the algorithms we study are detailed in Section 4.1.1 for the VBB algorithm and Section 4.2.1 for the MVC algorithm.

1.5. Our contribution

We present a fundamental module for dependable distributed systems: an SSBFT MVC algorithm for asynchronous message-passing systems. Hence, we advance the state-of-the-art *w.r.t.* the dependability degree. We obtained this new self-stabilizing algorithm by transforming the non-self-stabilizing MR algorithm. MR offers optimal resilience by assuming $t < n/3$, where t is the number of faulty processes, and n is the total number of processes. Our solution preserves this optimality.

In the absence of transient faults, our solution achieves consensus within a constant number of communication rounds during arbitrary executions and without execution fairness assumptions. After the occurrence of any finite number of arbitrary transient faults, the system recovers within a constant number of invocations of the underlying communication abstractions. This implies recovery within a constant time (in terms of asynchronous cycles), assuming execution fairness among the non-faulty processes. We clarify that these execution fairness assumptions are only needed for a bounded time, *i.e.,* during recovery, and not during the period in which the system is required to satisfy the task requirements (Definition 1.1). It is important to note that when also considering the stabilization time of the underlying communication abstractions, the recycling mechanism stabilizes within $\mathcal{O}(t)$ synchronous rounds.

The communication costs of the studied algorithm, MR, and the proposed one are similar in the number of BRB and binary consensus invocations. The main difference is that our SSBFT solution uses BV-broadcast to ensure that the value decided by the SSBFT binary consensus object remains consistent until the proposed SSBFT solution is completed and is ready to be recycled.

To the best of our knowledge, we propose the first self-stabilizing Byzantine-tolerant algorithm for solving MVC in asynchronous message-passing systems, enriched with required primitives. That is, our solution is built on using an SSBFT binary consensus object, a BV-broadcast object, and two SSBFT BRB objects as well as a synchronous recycling mechanism. We believe that our solution can stimulate research for the design of algorithms that can recover after the occurrence of transient faults.

2. System settings

We consider an asynchronous message-passing system that has no guarantees of communication delay. Also, the algorithms do not access the (local) clock (or use timeout mechanisms). The system consists of a set, $\mathcal{P}$, of n nodes (or processes) with unique identifiers. Any (ordered) pair of nodes $p_i, p_j \in \mathcal{P}$ has access to a unidirectional FIFO communication channel, $channel_{j,i}$, that, at any time, has at most $\mathsf{channelCapacity} \in \mathbb{Z}^+$ packets on transit from p_j to p_i (this assumption is due to a known impossibility [13, Chapter 3.2]). If the number of packets exceeds the channel capacity, the adversary may omit any message already in transit, as long as communication fairness is preserved. By communication fairness, we mean that if a packet is sent infinitely often, it will eventually be received infinitely often.

We adopt the *interleaving model* [13] to represent the asynchronous execution of a message-passing system. In this model, only one processor executes a single computation step at any given time. Each step, also referred to as an *atomic step*, comprises two parts: an internal computation and a single communication operation, which can be either a message *send* or *receive*.

To simplify the representation of program execution, the interleaving model assumes that the time taken for internal computations between two communication operations of a processor is instantaneous, *i.e.,* it is completed in zero time. In other words, state transition of any processor is triggered by the execution of a communication step, which encompasses all local computations performed after the previous step and before the communication operation of the current step. Thus, an *atomic step* is an indivisible execution unit, combining computation and communication into a single computation unit. This abstraction captures the fine-grained nature of the interleaving model.

Note that *receive* operations, appearing as 'upon message arrival' in our code, are non-blocking. Specifically, a *receive* operation is triggered when a message is ready to be delivered from the communication channel, allowing the receiving processor to immediately process it as part of its next atomic step.

2.1. The fault model and self-stabilization

We now specify the fault model and design criteria.

2.1.1. Arbitrary node failures

Byzantine faults model any fault in a node, including crashes and arbitrary malicious behaviors. Here, the adversary lets each node receive the arriving messages and calculates its state according to the algorithm. However, once a node (captured by the adversary) sends a message, the adversary can modify the message in any way, delay it arbitrarily long, or even remove it from the communication channel. The adversary can also send fake messages spontaneously. The adversary has the power to coordinate such actions without any limitation. For the sake of solvability [5,7,36], we limit the number, t, of nodes the adversary can capture, *i.e.,* $n \geq 3t + 1$. The set of non-faulty indices is denoted by $Correct$ and called the set of correct nodes.

2.1.2. 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 they can corrupt the system state arbitrarily (while keeping the program code intact). Following Dijkstra [11], we assume that the last arbitrary transient fault occurs before the system execution starts [13]. Also, it leaves the system to start in an arbitrary state. In other words, we assume arbitrary starting states at all correct nodes and the communication channels that lead to them. Moreover, transient faults do not occur during the system execution.

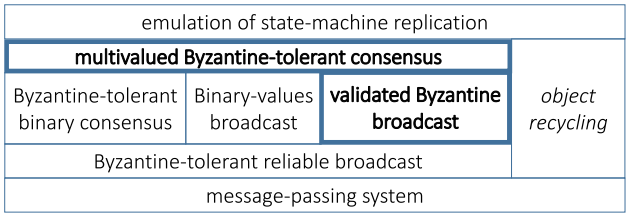

Fig. 1. We assume the availability of SSBFT protocols (cf. Definition 1.2) for binary consensus and object recycling. The studied problems appear in boldface fonts. The other layers, BRB, BV-broadcast, and state machine replication, are in plain font.

2.1.3. Dijkstra's self-stabilization

The set of *legal executions* (LE) consists of all executions that satisfy the problem requirements. (Recall the definition of a system execution from the second paragraph of this section.) A system is *self-stabilizing* with respect to LE, when every execution R of the algorithm reaches within a finite period a suffix $R_{legal} \in LE$ that is legal. Namely, Dijkstra [11] 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 $R = R' \circ R''$ is the concatenation of R' with R''. The part of the proof that shows the existence of R' is called *Convergence* (or recovery), and the part that shows $R_{legal} \in LE$ is called *Closure*.

2.1.4. Complexity measures and execution fairness

We say that execution fairness holds among processes if the scheduler enables any correct process infinitely often, *i.e.,* the scheduler cannot (eventually) halt the execution of non-faulty processes. Operation latency is the time between the invocation of an operation (such as consensus or broadcast) and the occurrence of all required delivery. As in MR, we show that the latency is finite without assuming execution fairness. The term stabilization time refers to the period in which the system recovers after the occurrence of the last transient fault. When estimating the stabilization time, our analysis assumes that all correct nodes complete roundtrips infinitely often with all other correct nodes. However, no execution fairness assumption is needed once the convergence period is over. Then, the stabilization time is measured in terms of *asynchronous cycles*, which we define next. All self-stabilizing algorithms have a do forever loop since these systems cannot be quiescent due to a well-known impossibility [13, Chapter 2.3]. Also, the studied algorithms allow nodes to communicate with each other via broadcast operation. Let num_b be the maximum number of (underlying) broadcasts per iteration of the do forever loop. The first asynchronous cycle, R', of execution $R = R' \circ R''$ is the shortest prefix of R in which every correct node can start and complete at least a constant number, num_b, of roundtrips with every correct node. R's second asynchronous communication round is the first round of the suffix R'', and so forth.

2.2. Building blocks

Following Raynal [10], Fig. 1 illustrates a protocol suite for SSBFT state-machine replication using total order broadcast. This order can be defined by instances of MVC objects, which in turn, invoke SSBFT binary consensus, BV-broadcast, and SSBFT recycling subroutine for consensus objects (GMRS [2,3]) as well as SSBFT BRB (DRS [1]). We clarify that the correctness of all these building blocks relies on the assumption of fair communication (see the first paragraph of this section).

2.2.1. SSBFT Byzantine-tolerant reliable broadcast (BRB)

The communication abstraction of (single instance) BRB allows every node to invoke the broadcast(v) : $v \in V$ and deliver() operations. This section presents a single-sender variation on BRB. The multi-sender variation can be derived by maintaining n concurrent BRB instances, each corresponding to a distinct sending node.

Definition 2.1 specifies the semantics of the operations broadcast() and deliver(). When these operations are invoked by nodes p_i and p_j, we use the notation broadcast$_i$() and deliver$_j$() to indicate that p_i is the broadcaster and p_j is any of the receivers. Additionally, the notation deliver$_j() \neq \perp$ indicates that node p_j has successfully delivered p_i's message.

Definition 2.1. The operations broadcast(v) and deliver() should satisfy:

- **BRB-validity.** Suppose a correct node BRB-delivers message m from a correct p_i. Then, p_i had BRB-broadcast m.
- **BRB-integrity.** No correct node BRB-delivers more than once.
- **BRB-no-duplicity.** No two correct nodes BRB-deliver different messages from p_i (which might be faulty).
- **BRB-completion-1.** Suppose p_i is a correct sender. Eventually, all correct nodes BRB-deliver from p_i.
- **BRB-completion-2.** Suppose a correct node BRB-delivers a message from p_i (which might be faulty). All correct nodes BRB-deliver p_i's message eventually.

We assume the availability of an SSBFT BRB implementation, such as the one in [1], which stabilizes within $\mathcal{O}(1)$ asynchronous cycles. Such implementation allows p_j to use the operation deliver$_j$() for retrieving the current return value, v, of the BRB broadcast from p_i. Before completing the task of the deliver$_j$() operation, v's value is $\perp$, which denotes the fact that no value is ready to be delivered. This behavior is akin to non-blocking I/O operations, where if the value is not ready, the operation returns $\perp$. In other

words, whenever $\mathsf{deliver}_j() \neq \perp$, node p_j knows that the broadcast task is completed and the returned value can be used. Several BRB implementations [37–39] satisfy different requirements than the ones in Definition 2.1, which is taken from the textbook [10].

Note that existing non-self-stabilizing BFT BRB implementations, *e.g.,* [10, Ch. 4], consider another interface between BRB and its application. In that interface, BRB notifies the application via the raising of an event whenever a new message is ready to be BRB-delivered. However, in the context of self-stabilization, a single transient fault can corrupt the BRB object to encode in its internal state that the message was already BRB-delivered without ever BRB-delivering it. The interface proposed in [1] addresses this challenge by allowing the application to repeatedly query the status of the SSBFT BRB object without changing its state.

We also assume that BRB objects have the interface function hasTerminated(), which serves as a predicate indicating when the sender knows that all non-faulty nodes have successfully delivered the application message. The implementation of hasTerminated() relies on an acknowledgment mechanism. Specifically, it checks whether a sufficient number of nodes have delivered the message by verifying the condition in the if-statement on line 49 of Algorithm 4 in [1]. If the required number of acknowledgments have been received, the function hasTerminated() returns True; otherwise, it returns False.

2.2.2. SSBFT binary-values broadcast (BV)

This is an all-to-all broadcast operation of binary values. This abstraction uses the operation $\mathsf{bvBroadcast}(v)$, which is assumed to be invoked independently (*i.e.,* not necessarily simultaneously) by all the correct nodes, where $v \in V$. We prefer $V = \{\mathsf{False}, \mathsf{True}\}$ over the traditional $V = \{0, 1\}$ presentation for a more straightforward presentation of our solutions. The set of BV-delivered values to node p_i can be retrieved via the function $binValues_i()$, which returns $\emptyset$ before the arrival of any bvBroadcast() by a correct node. We specify under which conditions values are added to $binValues()$.

- **BV-validity.** Suppose $v \in binValues_i()$ and p_i is correct. It holds that v has been BV-broadcast by a correct node.
- **BV-uniformity.** $v \in binValues_i()$ and p_i is correct. Eventually $\forall j \in Correct : v \in binValues_j()$.
- **BV-completion.** Eventually, $\forall i \in Correct : binValues_i() \neq \emptyset$ holds.

The above requirements imply that eventually $\exists s \subseteq \{\mathsf{False}, \mathsf{True}\} : s \neq \emptyset \wedge \forall i \in Correct : binValues_i() = s$ and the set s does not include values that were BV-broadcast only by Byzantine nodes. The SSBFT BV-broadcast solution in [2] stabilizes within $\mathcal{O}(1)$ asynchronous cycles. This implementation allows the correct nodes to repeat a BV-broadcast using the same BV-broadcast object. As mentioned in Section 1.4, this allows the proposed solution to overcome challenges related to the corruption of the state of the SSBFT binary consensus object; more details in Section 4.2.1.

2.2.3. SSBFT binary consensus

As mentioned, the studied solution reduces MVC to binary consensus by enriching the system model with a BFT object that solves binary consensus (Definition 2.2).

Definition 2.2. Every $p_i \in \mathcal{P}$ has to propose a value $v_i \in V = \{\mathsf{False}, \mathsf{True}\}$ via an invocation of $\mathsf{propose}_i(v_i)$. Let Alg be a binary Consensus (BC) algorithm. Alg has to satisfy *safety*, *i.e.,* BC-validity and BC-agreement, and *liveness*, *i.e.,* BC-completion.

- **BC-validity.** The value $v \in \{\mathsf{False}, \mathsf{True}\}$ decided by a correct node is a value proposed by a correct node.
- **BC-agreement.** Any two correct nodes that decide do so with identical decided values.
- **BC-completion.** All correct nodes decide.

We assume the availability of SSBFT binary consensus, such as the one from GMRS [2], which stabilizes within $\mathcal{O}(1)$ asynchronous cycles. GMRS's solution might fail to decide with negligible probability. In this case, GMRS's solution returns the error symbol, $\lightning$, instead of a legitimate value from the set $\{\mathsf{False}, \mathsf{True}\}$. The proposed SSBFT MVC algorithm returns $\lightning$ whenever the SSBFT binary consensus returns $\lightning$ (cf. line 70 of Algorithm 4).

2.2.4. The recycling mechanism and recyclable objects

To ensure efficient memory usage, our system employs a recycling mechanism for MVC objects, allowing them to be reused after completing their tasks. This mechanism is based on the GMRS framework [2,3], which we outline below.

Key concepts GMRS considers systems that implement consensus objects using constant-size storage allocated at program compilation time. Since the number of MVC object instantiations can be unbounded, efficient recycling of storage is essential after consensus is reached and all correct nodes have received the decided value via result().

Each MVC object has two meta-statuses: *unused* and *used*. The former indicates that the object is available for reuse whereas the latter indicates that the object is currently participating in consensus. The transition from *used* to *unused* is controlled by the recycling mechanism through the recycle() function.

To facilitate recycling, each MVC object implements a wasDelivered() function that: (1) Returns 1 once the result has been delivered, and (2) Ensures eventual agreement: if one non-faulty node reports delivery ($\mathsf{wasDelivered}_i() = 1$), all non-faulty nodes will eventually report delivery. Once a consistent agreement on wasDelivered() occurred, the value of wasDelivered() changes during GMRS's recycling process as the nodes independently mark objects as *unused* by invoking recycle().

Algorithm 1: Non-self-stabilizing BFT VBB-broadcast; code for p_i.

```
1  local variables: rec is a multiset initialized to ∅, storing all values in the messages that were BRB-delivered (including
     self-deliveries) resulting from broadcasts (of type INIT) initiated by any node in line 3 ;
2  operation broadcast(v) of VBB begin
3      BRB-broadcast INIT(i, v);
       /* wait until n−t INIT messages received from different senders */
4      wait |rec| ≥ n−t;
5      BRB-broadcast VALID(i, (equal(v, rec) ≥ n−2t));
6  foreach p_j ∈ P execute concurrently do
7      wait INIT(j, v) and VALID(j, x) BRB-delivered from p_j;
8      if x then
9          {wait (equal(v, rec) ≥ n−2t); d ← v}
10     else
11         {wait (differ(v, rec) ≥ t+1); d ← ↯}
12     raise VBB's event deliver(d) for sender p_j;
```

Interfacing with GMRS The recycling process resets algorithms to a predefined *post-recycling state*. This is achieved by resetting all variables in use, as shown in line 26 in Algorithm 3 and lines 53 and 54 in Algorithm 4. Algorithms 3 and 4 also implement the result() operation, which supports the implementation of wasDelivered(), following GMRS's approach.

Mitigating GMRS's synchrony assumptions GMRS's recycling service operates under synchronous assumptions, enabling non-faulty nodes to reuse objects immediately after invoking recycle(). This synchronous recycling facilitates our transformation of the non-self-stabilizing BFT MR algorithm into a SSBFT one. Once all objects are recycled, the system reaches a *post-recycling state* with no stale information, ensuring convergence. As mentioned in Section 1.4, the effect of these assumptions can be mitigated by recycling batches of δ objects, where δ is a predefined constant that depends on the available memory. This way, the communication-intensive components are asynchronous, and the synchronous recycling actions occur according to a load defined by δ.

3. The studied algorithms

As mentioned, MR is based on a reduction of BFT MVC to BFT binary consensus. MR guarantees that the decided value is not proposed by Byzantine nodes only. Also, if there is a value, $v \in V$, that all correct nodes propose, then v is decided. Otherwise, the decided value is either proposed by the correct nodes or it is the error symbol, $\lightning$. This way, an adversary that commands its captured nodes to propose the same value, say, $v_{byz} \in V$, cannot lead to the selection of v_{byz} without the support of at least one correct node. MR uses the VBB communication abstraction (Fig. 1), which we present (Section 3.1) before we bring the reduction algorithm (Section 3.2).

3.1. Validated Byzantine broadcast (VBB)

This abstraction sends messages from any node, p_i, to all nodes, p_j. It allows p_i to invoke the VBB's operation, broadcast$_i(v)$ and p_j to raise the VBB's event of deliver$_j(v)$, for VBB-broadcasting, and resp., VBB-delivering. We clarify that VBB is used only as a multi-sender broadcasting abstraction. This is done by maintaining n concurrent (and interdependent) VBB instances, each corresponding to a distinct sending node.

3.1.1. Specifications

We detail VBB-broadcast requirements.

- **VBB-validity.** VBB-delivery of messages needs to relate to VBB-broadcast of messages in the following manner.
  - **VBB-justification.** Suppose $p_i : i \in Correct$ VBB-delivers message $m \neq \lightning$ from some (faulty or correct) node. There is at least one correct node that VBB-broadcasts m.
  - **VBB-obligation.** Suppose all correct nodes VBB-broadcast the same v. All correct nodes VBB-deliver v from each correct node.
- **VBB-uniformity.** Let $p_i : i \in Correct$. Suppose p_i VBB-delivers $m' \in \{m, \lightning\}$ from a (possibly faulty) node p_j. All the correct nodes VBB-deliver the same message m' from p_j.
- **VBB-completion.** Suppose a correct node p_i VBB-broadcasts m. All the correct nodes VBB-deliver from p_i.

We also say that a complete VBB-broadcast instance includes VBB broadcast$_i(m_i)$ invocation by every correct $p_i \in \mathcal{P}$. It also provides VBB deliver() of m' from at least $(n-t)$ distinct nodes, where m' is either p_j's message, m_j, or the error symbol, $\lightning$. The latter value is returned when a message from a given sender cannot be validated. This validation requires m_j to be VBB-broadcast by at least one correct node.

Algorithm 2: Non-self-stabilizing BFT MVC; code for p_i.

13 **local variables:** bcO is a binary consensus object, $\perp$ is the initial state;
14 rec is a multiset initialized to $\emptyset$, storing all values in the messages that were VBB-delivered (including self-deliveries) resulting from broadcasts (of type EST) initiated by any node in line 18;

/* the following macro returns True if, and only if, the set of values VBB-delivered at p_i satisfy the condition of VBB-delivering identical values at least $(n-2t)$ times from different senders (and there is only one non-$\lightning$ value in rec) */

15 **macro** $sameValue()$ **do return** $\exists v \neq \lightning : equal(v, rec) \geq n-2t \wedge |rec \setminus \{\lightning\}| = 1$;

16 **operation** propose(v) **begin**
17 VBB broadcast EST(v);
18 **wait** EST($\bullet$) messages VBB-delivered from $(n-t)$ different nodes, which are stored in rec;
/* the next call invokes the binary consensus object and blocks further program execution until a consensus is reached */
19 bcO.propose($sameValue()$);
20 **if** bcO.result() **then**
21 **wait** $(\exists v \neq \lightning : equal(v, rec) \geq n-2t)$ **then return** v
22 **else**
23 **return** $\lightning$

3.1.2. Implementing VBB-broadcast

Algorithm 1 presents the studied VBB-broadcast.

Notation: Denote by $equal(v, rec)$ and $differ(v, rec)$ the number of items in multiset rec that are equal to and different from v, resp.

Overview: Algorithm 1 invokes BRB-broadcast twice in the first part of the algorithm (lines 2 to 5), and then VBB-delivers messages from nodes in the second part (lines 6 to 12).

Node p_i first BRB-broadcasts INIT(i, v_i) (where v_i is the VBB-broadcast message), and suspends until the arrival of INIT() from at least $(n-t)$ different nodes (lines 3 to 4), which p_i collects in the multiset rec_i. In line 3, node p_i tests whether v_i was BRB-delivered from at least $n-2t \geq t+1$ different nodes. Since this means that v_i was BRB-broadcast by at least one correct node, p_i attests to the validity of v_i (line 5). Recall that each time INIT() arrives at p_i, the message is added to rec_i. Therefore, the fact that $|rec_i| \geq n-t$ holds (line 4) does not keep rec_i from growing.

Algorithm 1's second part (lines 6 to 12) includes n concurrent background tasks. Each task aims at VBB-delivering a message from a different node, say, p_j. It starts by waiting until p_i BRB-delivered both INIT(j, v_j) and VALID(j, x_j) from p_j so that p_i has both p_j's VBB's values, v_j, and the result of its validation test, x_j.

1. **The $x_j = \mathsf{True}$ case (line 8).** Since p_j might be faulty, we cannot be sure that v_j was indeed validated. Thus, p_i re-attests v_j by waiting until $equal(v_j, rec_i) \geq n-2t$ holds. If this happens, p_i VBB-delivers v_j as a message from p_j, which implies $equal(v_j, rec_i) \geq t+1$ since $n-2t \geq t+1$.
2. **The $x_j = \mathsf{False}$ case (line 11).** For similar reasons to the former case, p_i waits until rec_i has at least $t+1$ items that are not v_j. This implies at least one correct node cannot attest to v_j's validity. If this ever happens, p_i VBB-delivers the error symbol, $\lightning$, as the received message from p_j.

3.2. Non-stabilizing BFT multivalued consensus

Algorithm 2 reduces the BFT MVC problem to BFT binary consensus in message-passing systems with up to $t < n/3$ Byzantine nodes. Algorithm 2 uses VBB-broadcast abstraction (Algorithm 1). Note that the line numbers of Algorithm 2 continue the ones of Algorithm 1.

3.2.1. Specifications

Our BFT MVC task includes the requirements of BC-validity, BC-agreement, and BC-completion, as well as the BC-no-Intrusion property (all requirements being defined in Section 1.1).

3.2.2. Implementation

Node p_i waits for EST() messages from $(n-t)$ different nodes after it has VBB-broadcast its own value (lines 17 to 18). It holds all the VBB-delivered values in the multiset rec_i (line 15) before testing whether rec_i includes (1) non-$\lightning$ replies from at least $(n-2t)$ different nodes and (2) precisely one non-$\lightning$ value v (line 15). The test result is proposed to the binary consensus object, bcO (line 19). We clarify that the use of this binary consensus object is imperative since Algorithm 2 solves the BFT MVC problem. We assume that each algorithm uses a distinct namespace for its local variables. Specifically, the name rec (lines 1 and 14) refers to different variables in Algorithms 1 and 2.

Once consensus is reached, p_i decides according to the consensus result, bcO_i.result() (line 19). Note that bcO_i.result() = True (line 20) if, and only if, there is at least one correct node that received non-$\lightning$ replies from at least $(n-2t)$ different nodes, and these replies included exactly one value, which is $v \neq \lightning$. Otherwise, *i.e.,* when bcO_i.result() = False, there are no guarantees that all correct nodes could eventually attest to the validity of their proposed value. Thus, if bcO_i.result() = False, p_i returns the error symbol, $\lightning$ (line 23), since there is no guarantee that any correct node could attest to the validity of the proposed value. Otherwise, p_i waits until it receives EST(v) messages that have identical values from at least $(n-2t)$ different nodes before returning that value v (line 21).

Note that some of these $(n-2t)$ messages were already VBB-delivered at line 18. The proof in [4] shows that, if all correct nodes eventually VBB-deliver identical values at least $(n-2t)$ times, then any correct node, p_i, that invokes bcO_i.propose(True) does so based on its own set of VBB-delivered values. Specifically, for any correct node, p_j, the invariant bcO_j.result() = True indicates the existence of a correct node p_ℓ that has VBB-delivered identical values at least $(n-2t)$ times from different senders. This ensures that p_j can eventually decide on the value v once it also VBB-delivers identical values at least $(n-2t)$ times from different senders. That is, p_j can safely decide on the returned value, v, for the MVC object, as this value matches the one p_ℓ had VBB-delivered at least $(n-2t)$ times from different senders.

4. SSBFT multivalued consensus

Algorithms 3 and 4 present our SSBFT VBB solution and self-stabilizing Byzantine- and intrusion-tolerant solution for MVC. They are obtained from Algorithms 1 and 2 via code transformation and the addition of necessary consistency tests (Sections 4.1.1 and 4.2.1). Note that the line numbers of Algorithms 3 and 4 continue the ones of Algorithms 2, and resp., 3.

4.1. SSBFT VBB-broadcast

The operation VBB broadcast(v) allows node p_i to initiate a VBB-broadcast instance with the value v. Node p_j can retrieve messages VBB-broadcast by p_i by invoking the operation VBB deliver() on its VBB object associated with p_i. As previously mentioned (Section 2.2.1), a message is considered delivered when the operation returns a value that is non-$\perp$. Otherwise, the operation must be invoked repeatedly until a non-$\perp$ value is obtained.

4.1.1. Algorithm 1's invariants that transient faults can violate

Define the phase types, vbbMSG $:=$ {init, valid} (line 24) for BRB dissemination of INIT(), and resp., VALID() messages in Algorithm 1. Transient faults can violate the following invariants, which our SSBFT solution addresses via consistency tests.

1. Node p_i's state must not record the occurrence of BRB execution of phase valid (line 5) without encoding BRB execution of phase init (line 3). Algorithm 3 addresses this concern by informing that the VBB object has an internal error (line 41). This way, the application is prevented from being blocked indefinitely.
2. For a given phase, $phs \in$ vbbMSG, the BRB message format must follow the one of BRB-broadcast of phase phs, as in lines 3 and 5. In order to avoid blocking, the VBB object informs about an internal error (lines 45 and 46).
3. For a given phase, $phs \in$ vbbMSG, if at least $n-t$ different nodes BRB-delivered messages of phase phs, to node p_i, p_i's state must lead to the next phase, *i.e.,* from init to valid, or from valid to operation complete, in which p_i VBB-delivers a non-$\perp$ value. Algorithm 3 addresses this concern by monitoring the conditions in which the nodes should move from phase init to valid (line 36). The case in which the nodes should move from phase valid to operation complete is more challenging since a single transient fault can (undetectably) corrupt the state of the BRB objects. Algorithm 3 makes sure that such inconsistencies are detected eventually. When an inconsistency is discovered, the VBB object informs the application about an internal error (line 50); see Section 4.1.5 for more details.

4.1.2. Local variables

The array brb[vbbMSG][$\mathcal{P}$] holds BRB objects, which disseminate BRB-broadcast messages. Specifically, Algorithm 3 utilizes brb[init][] to send and deliver Algorithm 1's INIT() messages. It also uses brb[valid][] send and deliver VALID() messages. The second dimension of the array brb[][] enables Algorithm 3 to associate each BRB object with a unique sending node, as required by Algorithm 1, which counts the number of BRB messages arriving from different senders (line 4).

As specified in Section 2.2.4, after recycling these objects or before they ever become active, each of the $2n$ BRB objects is initialized to $\perp$. For a given $p_i \in \mathcal{P}$ and $phs \in$ vbbMSG, the BRB object $brb_i[phs][i]$ becomes active either through a $brb_i[phs][i]$.broadcast(v) invocation (resulting in $brb_i[phs][i] \neq \perp$) or by receiving BRB protocol messages from sender p_j with phase phs (resulting in $brb_i[phs][j] \neq \perp$). Once a BRB message is delivered from p_ℓ (in the context of phase $phs \in$ vbbMSG), a call to $brb_i[phs][\ell]$.deliver() retrieves the delivered message.

4.1.3. Macros

The macro *vbbWait*(phs) (line 29) serves at if-statement conditions in lines 36 and 50 when the proposed transformation represents the exit conditions of the wait operations in lines 4 and 11. Specifically, given a phase, phs, it tests whether a set S includes at least $n-t$ different nodes from which there is a message ready to be BRB-delivered. We clarify that $brb_i[phs][\ell]$.deliver() (line 29) accesses

Algorithm 3: SSBFT VBB-broadcast; code for p_i.

24 **types:** vbbMSG := {init, valid}; // BRB object phases
25 **local variables:**
26 brb[vbbMSG][$\mathcal{P}$] // brb[init][$\mathcal{P}$] and brb[valid][$\mathcal{P}$] are BRB objects. Upon recycling, [[$\perp,\dots,\perp$], [$\perp,\dots,\perp$]] is assigned;
27 **macros:**
28 // exit conditions of wait operations in lines 4 and 11
29 $vbbWait(phs) := \exists S \subseteq \mathcal{P} : n-t \leq |S| : \forall p_\ell \in S : (brb[phs][\ell].\text{deliver}() \neq \perp)$
30 // detailed version of equal condition in lines 5 and 8
31 $vbbEq(phs, v) := \exists S \subseteq \mathcal{P} : n-2t \leq |S| : \forall p_\ell \in S : ((-, v) = brb[phs][\ell].\text{deliver}())$;
32 // detailed version of the differ condition used in line 11
33 $vbbDiff(phs, v) := \exists S \subseteq \mathcal{P} : t+1 \leq |S| : \forall p_\ell \in S : (v \neq brb[phs][\ell].\text{deliver}())$;
34 **operation:** VBB broadcast(v) **do** brb[init][i].broadcast((i, v)) // cf. line 3;
35 **do-forever begin**
36 **if** $vbbWait(\texttt{init}) \wedge brb[\texttt{init}][i].\text{hasTerminated}()$ **then**
37 **let** $v := brb[\texttt{valid}][i].\text{deliver}()$;
38 $brb[\texttt{valid}][i].\text{broadcast}((i, vbbEq(\texttt{init}, v)))$ // cf. line 5;
39 **operation:** deliver() of p_k's VBB object **begin**
40 // case (I) of the consistency tests (Section 4.1.1)
41 **if** $brb[\texttt{init}][k].\text{deliver}() = \perp \wedge brb[\texttt{valid}][k].\text{deliver}() \neq \perp$ **then return** $\lightning$;
42 // wait until p_j's BRB objects have delivered, cf. line 7
43 **if** $brb[\texttt{init}][k].\text{deliver}() = \perp \vee brb[\texttt{valid}][k].\text{deliver}() = \perp$ **then return** $\perp$;
44 // lines 45 and 46 are case (II) of consistency tests (Section 4.1.1)
45 **if** $\exists phs \in \texttt{vbbMSG} : brb[phs][k].\text{deliver}() = (j, -) \wedge j \neq k$ **then return** $\lightning$;
46 **if** $\neg((brb[\texttt{init}][k].\text{deliver}() = (k, v) \wedge v \in V) \wedge (brb[\texttt{valid}][k].\text{deliver}() = (k, x) \wedge x \in \{\text{False}, \text{True}\}))$ **then return** $\lightning$;
47 **else if** $x \wedge vbbEq(\texttt{valid}, v)$ **then return** v; // cf. line 8
48 **else if** $\neg x \wedge vbbDiff(\texttt{valid}, v)$ **then return** $\lightning$; // cf. line 11
49 // case (III) of the consistency tests (Section 4.1.1)
50 **else if** $vbbWait(\texttt{valid})$ **then return** $\lightning$;
51 **return** $\perp$; // VBB's deliver() is incomplete

the BRB object, $brb[phs][\ell]$, associated with messages of type phs sent from p_ℓ. Node p_i, when executing line 29, invokes deliver() to retrieve BRB messages that need to be delivered from p_ℓ.

The macros $vbbEq(phs, v)$ (line 31) and $vbbDiff(phs, v)$ (line 33) are detailed versions of the equal, and resp., differ conditions used in lines 5 and 8, resp., line 11. They check whether the value v equals to, resp., differs from at least $n-2t$, resp., $t+1$ received BRB messages of phase phs. The symbol '$-$' in line 33 denotes any possible parameter value.

4.1.4. *The* VBB broadcast() *operation (line 34)*

As in line 3 in Algorithm 1, VBB broadcast(v)'s invocation (line 34) leads to the invocation of the BRB object brb[init][i].broadcast((i, v)). Algorithm 4 uses line 36 for implementing the logic of lines 4 and 5 in Algorithm 1 as well as the consistency test of item 3 in Section 4.1.1; that case of moving from phase init to valid. In detail, the macro $vbbWait(phs)$ returns True whenever the BRB object $brb[phs][k]$ has a message to BRB-deliver from at least $n-t$ different nodes. Thus, p_i can "wait" for BRB deliveries from at least $n-t$ distinct nodes by testing $vbbWait_i(\texttt{init}) \wedge brb_i[\texttt{init}][i].\text{hasTerminated}()$, where the second clause indicates that $brb_i[\texttt{init}][i]$ has terminated (Section 2.2.1). Thus, Item 1 in Section 4.1.1 is implemented correctly. Also, the macro $vbbEq()$ is a detailed implementation of the function $equal()$ (Algorithm 1). As mentioned when describing line 29, $brb[\texttt{valid}][i]_i$.deliver() (line 37) accesses the BRB object, $brb[\texttt{valid}][i]_i$, associated with messages of type valid sent from p_i. Node p_i, when executing line 37, invokes deliver() to retrieve BRB messages that need to be delivered from itself, p_i.

4.1.5. *The* VBB deliver() *operation (lines 39 and 51)*

This operation (lines 39 to 51) is based on lines 6 and 12 in Algorithm 1 and a few consistency tests (Section 4.1.1). Note that node p_i executes this part of the code, including the macros it invokes, within the context of the sender p_k.

Line 41 performs a consistency test that matches Item 1 in Section 4.1.1, *i.e.,* for a given sender $p_k \in \mathcal{P}$, if p_k had invoked $brb[\texttt{valid}][k]$ before $brb[\texttt{init}][k]$'s termination, an error is indicated via the return of $\lightning$. Line 43 follows line 7's logic by testing whether this VBB object is ready to complete w.r.t. sender $p_k \in \mathcal{P}$. It does so by checking the state of the two BRB objects in $brb[-][k]$ since they each need to deliver a non-$\perp$ value. If any of them are not ready to be completed, the operation returns $\perp$.

The if-statements in lines 45 and 46 return $\lightning$ when the delivered BRB message is ill-formatted. By that, they fit the consistency test of Item 2 in Section 4.1.1. As mentioned, the symbol '$-$' in line 45 denotes any possible parameter value.

Algorithm 4: Self-stabilizing Byzantine-tolerant multivalued consensus via VBB-broadcast; code for p_i.

52 **local variables:**
53 $bvO := \perp$; // Binary-values object. Recycling assigns $\perp$
54 $bcO := \perp$; // Binary consensus object. Upon recycling, assign $\perp$
55 $vbb[\mathcal{P}]$ // VBB objects, one sender per object. Upon recycling, $[\perp, \ldots, \perp]$ is assigned;
56 **macros:**
57 // exit conditions of the wait operation in line 18 $mcWait() := \exists S \subseteq \mathcal{P} : n-t \leq |S| : \forall p_k \in S : (vbb[k].\mathsf{deliver}() \neq \perp)$;
58 // adapted version of the same macro in line 15
59 $sameValue()$ **do return** $(\exists v \notin \{\perp, \lightning\} : \exists S' \subseteq \mathcal{P} : n-2t \leq |S'| : \forall p_{k'} \in S' : (vbb[k'].\mathsf{deliver}() = v)) \wedge$
$(|\{vbb[k].\mathsf{deliver}() \notin \{\perp, \lightning\} : p_k \in \mathcal{P}\}| = 1)$;
60 **operation:** propose(v) **do** $\{vbb[i].\mathsf{broadcast}(v)\}$;
61 **operation:** result() **begin**
62 // test whether result() is not ready to complete
63 **if** $bcO = \perp \vee bcO.\mathsf{result}() = \perp$ **then**
64 **return** $\perp$;
65 **else if** $\neg bcO.\mathsf{result}()$ **then** // cf. line 23
66 **return** $\lightning$;
67 **else if** $\exists v \notin \{\perp, \lightning\} : \exists S' \subseteq \mathcal{P} : n-2t \leq |S'| : \forall p_{k'} \in S' : (vbb[k'].\mathsf{deliver}() = v)$ **then** // cf. line 21
68 **return** v;
69 // perform a consistency test, cf. Section 4.2.1
70 **else if** $mcWait() \wedge \mathit{True} \notin bvO.binValues()$ **then**
71 **return** $\lightning$;
72 **return** $\perp$; // result() is not ready to complete
73 **do-forever begin**
74 **if** $mcWait()$ **then** // cf. line 18
75 **if** $bcO = \perp$ **then** $bcO.\mathsf{propose}(sameValue())$; // cf. line 20
76 $bvO.\mathsf{broadcast}(sameValue())$; // assist with the consistency test in line 70

The if-statements in lines 47 to 48 implement the logic of lines 8 to 11 in Algorithm 1. The logic of these lines is explained in items 1, and resp., 2 in Section 3.1.1. Similar to line 8 in Algorithm 1, x_i (line 46) is the value that p_i BRB-delivers from p_k via the BRB object $brb_i[\mathtt{valid}]$. As mentioned, the macro $vbbDiff()$ is a detailed implementation of $differ()$ used by Algorithm 1.

The if-statement in line 50 considers the case in which x_i is corrupted. Thus, there is a need to return the error symbol, $\lightning$. This happens when p_i VBB-delivered VALID() messages from at least $n-t$ different nodes, but none of the if-statement conditions in lines 41 to 48 hold. This fits the consistency test of Item 3 in Section 4.1.1, which requires eventual completion in the presence of transient faults.

4.2. SSBFT multivalued consensus

The invocation of propose(v) VBB-broadcasts v. Node p_i VBB-delivers messages from p_k via the result$_i$() operation.

4.2.1. Algorithm 2's invariants that transient faults can violate

As mentioned in Section 1.4, the occurrence of a transient fault can cause the binary consensus object to encode a decided value that was never proposed, which violates BC-validity.

Any SSBFT solution must address this concern since the MVC object can be blocked indefinitely if bcO decides True when $\forall p_j : j \in Correct : sameValue_j() = \mathsf{False}$ holds. As we explain next, our implementation BV-broadcasts (line 76) for testing the consistency of the SSBFT binary consensus object (line 70). This way, indefinite blocking can be avoided by reporting an internal error state.

4.2.2. Local variables

Algorithm 4's state includes the SSBFT BV-broadcast object, bvO, and SSBFT consensus binary object, bcO. As required in Section 2.2.4, each object has the post-recycling value of $\perp$, *i.e.,* when $bvO = \perp$ (or $bcO = \perp$) the object is said to be inactive. They become active upon invocation and complete according to their specifications (which are detailed in Sections 2.2.1 and 2.2.2, resp.).

The array $vbb[]$ holds VBB objects, which disseminate VBB-broadcast messages. Specifically, Algorithm 4 utilizes $vbb[]$'s objects to send and deliver Algorithm 2's VBB messages while associating each VBB object with a unique sending node, as required by Algorithm 2, which counts the number of VBB messages arriving from different senders (line 18).

4.2.3. Macros

The macro *mcWait*() (line 57) serves at the if-statement conditions in lines 70 and 74 when the proposed transformation represents the exit conditions of the wait operations in lines 18 and 21. Specifically, it tests whether a set $S \subseteq \mathcal{P}$ includes at least $n-t$ different nodes from which there a message is ready to be VBB-delivered. The macro *sameValue*() is an adaptation of the macro in line 15, which tests whether there is a value $v \notin \{\perp, \lightning\}$ that a set of at least $n-2t$ different nodes have VBB-delivered and there is only one value $v' \notin \{\perp, \lightning\}$ that was VBB-delivered.

4.2.4. Implementation

The logic of lines 16 and 21 in Algorithm 2 is implemented by lines 60 to 75 in Algorithm 4. *I.e.,* the invocation of propose(v) (line 60) leads to the VBB-broadcast of v.

The logic of lines 18 and 20 in Algorithm 2 is implemented by lines 74 and 75, resp. In detail, recall from Section 4.2.3 that *mcWait*() (line 74) allows waiting until there are at least $n-t$ different nodes from which p_i is ready to VBB-deliver a message. Then, if $bcO = \perp$ (*i.e.,* the binary consensus object was not invoked), line 75 uses bcO to propose the returned value from *sameValue*(). Recall from Section 4.2.3, that the macro *sameValue*() (line 59) implements the one in line 15 (Algorithm 2); see Section 3.2.2 for details.

Line 76 facilitates the implementation of the consistency test (Section 4.2.1) by using BV-broadcasting to disseminate the returned value from *sameValue*(). This way, it is possible to detect the case where all correct nodes BV-broadcast a value that is different from bcO's decided one, due to a transient fault. This is explained when we discuss line 70.

The operation result() (lines 61 to 72) returns the decided value, which lines 23 and 21 implement in Algorithm 2. It is a query-based operation, just as deliver() (cf. text just after Definition 2.1). Thus, line 63 considers the case in which the decision has yet to occur, *i.e.,* it returns $\perp$. Line 65 considers the case that line 20 (Algorithm 2) deals with and returns the error symbol, $\lightning$. Line 67 implements line 21 (Algorithm 2), *i.e.,* returns the decided value. Line 70 performs a consistency test for the case in which the if-statement conditions in lines 63 to 67 hold, there are VBB-deliveries from at least $n-t$ different nodes (*i.e.,* $mcWait_i()$ holds), and yet there is no correct node, say p_j, that reports to p_i, via BV-broadcast, that the predicate $sameValue_j()$ holds. Lemma 5.8 shows that this test addresses the challenge described in Section 4.2.1. Whenever none of the conditions of the if-statements in lines 63 to 70 hold, line 72 returns $\perp$.

5. Correctness

As explained in Section 2.2.4, we demonstrate Convergence (Theorem 5.1) by showing that all operations are eventually completed since this implies their recyclability, and thus, the SSBFT object recycler can restart their state (Section 2.2.4). For every layer, *i.e.,* VBB-broadcast and MVC, we prove the properties of completion and Convergence (Theorems 5.1 and resp., 5.6) before demonstrating the Closure property, cf. Theorems 5.2 and 5.9 resp.

5.1. VBB-completion and convergence

The proof demonstrates Convergence by considering executions that start in arbitrary states. Theorem 5.1 shows that all VBB objects are completed within a bounded time. Specifically, assuming fair execution among the correct nodes (Section 2.1.4), Theorem 5.1 shows that, within a bounded time, for any pair of correct nodes, sender p_i and receiver p_j, a non-$\perp$ value is returned from $vbb_j[i]$.deliver(). As explained in Section 2.2.4, this means that all VBB objects become recyclable, *i.e.,* $\text{wasDelivered}_i()$ returns True. Since we assume the availability of the object recycling mechanism, the system reaches a post-recycling state within a bounded time. Specifically, using the mechanism by GMRS [2,3], Convergence is completed with $\mathcal{O}(t)$ synchronous rounds. We introduce the *CRWF/ACAF* notation since the proof can use the arguments of Theorem 5.1 to demonstrate different properties under different assumptions. Specifically, Theorem 5.1 demonstrates that VBB-completion occurs within $\mathcal{O}(1)$ communication rounds (Section 2.1.4) without assuming execution fairness but assuming execution R starts in a post-recycling system state. For the sake of brevity, when the proof arguments are used for counting the number of Communication Rounds Without assuming execution Fairness (CRWF), we write 'within $\mathcal{O}(1)$ CRWF'. Theorem 5.1 also demonstrates Convergence within $\mathcal{O}(1)$ asynchronous cycles assuming fair execution among the correct nodes (Section 2.1.4). Thus, when the proof arguments can be used for counting the number of Asynchronous Cycles while Assuming execution Fairness (ACAF), we say, in short, 'within $\mathcal{O}(1)$ ACAF'. Moreover, when the same arguments can be used in both cases, we say 'within $\mathcal{O}(1)$ CRWF/ACAF'.

Theorem 5.1 demonstrates that VBB-completion is achieved within a bounded time. The proof follows a sequence of observations.

- Observation 5.1.1 addresses scenarios where the VBB object is not in its post-recycling state, resulting from either a VBB-broadcast invocation or a transient fault. It establishes that the relevant BRB object engages in the process within a defined time frame.
- Recall that line 41 pertains to a situation reachable solely due to a transient fault. Observation 5.1.2 deals with cases where the if-statement condition in line 41 does not hold. It asserts that within a finite duration, one of the conditions leading to completion is met, namely, the BRB object of the `valid` phase delivers a non-$\perp$ value.
- Observation 5.1.3 builds upon Observation 5.1.2 by demonstrating that the same conditions imply either the if-statement condition in line 41 holds or the one in line 43 cannot.
- Observation 5.1.4 utilizes the BRB properties along with Observations 5.1.1 to 5.1.3 to conclude the proof.

Theorem 5.1 *(VBB-completion). Suppose all correct nodes, p_i, invoke $vbb_i[i]$.broadcast() within $\mathcal{O}(1)$ CRWF/ACAF. Within $\mathcal{O}(1)$ CRWF/ACAF, $\forall_{i,j \in Correct} : vbb_j[i]$.deliver() $\neq \perp$.*

Proof of Theorem 5.1. Let $i \in Correct$. Suppose either p_i VBB-broadcasts m in R or $\exists phs \in$ `vbbMSG` $: brb_j[phs][i] \neq \perp$ holds in R's starting state. We demonstrate that all correct nodes VBB-deliver $m' \neq \perp$ from p_i by considering all the if-statements in lines 41 to 50 and showing that, within $\mathcal{O}(1)$ CRWF/ACAF, one of the if-statements (that returns a non-$\perp$) in lines 41 to 50 holds.

Observation 5.1.1. *Suppose $\exists phs \in$ `vbbMSG` $: \exists \ell \in Correct : brb_j[phs][i] \neq \perp$,* i.e., *$brb_j[phs][i]$ is not in its post-recycling state. Within $\mathcal{O}(1)$ CRWF/ACAF, $\forall k \in Correct : brb_k[phs][\ell] \neq \perp$ holds.*

Proof of Observation 5.1.1. The proof is implied by BRB-completion-1, BRB-completion-2, and the $\mathcal{O}(1)$ stabilization time of SSBFT BRB, cf. Section 2.2.1. □Observation 5.1.1

Observation 5.1.2. *Suppose in R, the if-statement condition in line 41 does not hold. Within $\mathcal{O}(1)$ CRWF/ACAF, brb_i*`[valid]`*$[i] \neq \perp$ holds.*

Proof of Observation 5.1.2. By the theorem assumption that all correct nodes, p_i, invoke $vbb_i[i]$.broadcast() within $\mathcal{O}(1)$ CRWF/ACAF, the BRB properties (Definition 2.1), and that there are at least $(n-t)$ correct nodes, the if-statement condition in line 36 holds within $\mathcal{O}(1)$ CRWF/ACAF. Then, p_i invokes the operation brb_i*`[valid]`*$[i]$.broadcast(). □Observation 5.1.2

Observation 5.1.3. *Suppose the condition brb_i*`[valid]`*$[i] \neq \perp$ holds in R's starting state. Within $\mathcal{O}(1)$ CRWF/ACAF, either the if-statement condition in line 41 holds or the one in line 43 cannot hold.*

Proof of Observation 5.1.3. The proof is implied by Algorithm 4's code, BRB-completion, and Observations 5.1.1 and 5.1.2. □Observation 5.1.3

Observation 5.1.4. *Within $\mathcal{O}(1)$ CRWF/ACAF, $vbb_j[i]$.deliver() $\neq \perp$ holds.*

Proof of Observation 5.1.4. Suppose the if-statement conditions in lines 41 to 48 never hold. By *vbbWait*()'s definition (line 29), BRB properties (Definition 2.1), the presence of at least $n-t$ correct and active nodes, and Observations 5.1.1 to 5.1.3, the if-statement condition in line 50 holds within $\mathcal{O}(1)$ CRWF/ACAF. □Observation 5.1.4

□Theorem 5.1

5.2. Closure of VBB-broadcast

Theorem 5.10 demonstrates Closure by considering executions that start from a post-recycling state, which Theorem 5.1 implies that the system reaches, see Section 5.1 for details. Theorem 5.2's proof shows no consistency test causes false error indications. Theorem 5.2 counts communication rounds (without assuming execution fairness) using the CRWF notation presented in Section 5.1.

Theorem 5.2 expands upon the proof provided in Theorem 5.1 by establishing the fulfillment of all VBB properties. As previously mentioned, the proof illustrates the properties of VBB-uniformity, VBB-obligation, and VBB-justification in a manner akin to the proof of the MR algorithm by considering the assumption that the system begins in a post-recycling state. However, adjustments of the proof are necessary since Algorithm 3's structure differs somewhat from MR.

Theorem 5.2 *(VBB-Closure). Let R be an Algorithm 4's execution in which all correct nodes invoke their individual* VBB's broadcast() within $\mathcal{O}(1)$ CRWF. Assume R starts in a post-recycling state. R satisfies the VBB requirements (Section 3.1.1).

Proof of Theorem 5.2. VBB-completion holds (Theorem 5.1).

Lemma 5.3 *(VBB-uniformity). VBB-uniformity holds.*

Proof of Lemma 5.3. Let $i \in Correct$. Suppose p_i VBB-delivers $m' \in \{m, \not\downarrow\}$ from a (possibly faulty) $p_j \in \mathcal{P}$. We show that all the correct nodes VBB-deliver the same message m' from p_j. Since R is post-recycling and p_i VBB-delivers m' from p_j, the condition brb_i*`[init]`*$[j]$.deliver() $= \perp \wedge brb_i$*`[valid]`*$[j]$.deliver() $\neq \perp$ (of the if-statement in line 41) cannot hold. And, within $\mathcal{O}(1)$ CRWF we know that (brb_i*`[init]`*$[j]$.deliver() $= (j, v_{j,i}) \wedge brb_i$*`[valid]`*$[j]$.deliver() $= (j, x_{j,i})$) (line 46) hold (BRB-completion-2 and since all correct nodes invoke their individual VBB's broadcast() within $\mathcal{O}(1)$ CRWF). Also, the condition in line 41 cannot hold *w.r.t.* p_k. And, within $\mathcal{O}(1)$ CRWF, (brb_k*`[init]`*$[j]$.deliver() $= (j, v_{j,k}) \wedge brb_k$*`[valid]`*$[j]$.deliver() $= (j, x_{j,k})$) holds, such that $v_{j,i} = v_{j,k}$ and $x_{j,i} = x_{j,k}$. This is because R starts in a post-recycling system state, BRB-no-duplicity, and BRB-completion-2, which means that every correct node p_k BRB-delivers, within $\mathcal{O}(1)$ CRWF, the same messages that p_i delivers. Due to similar reasons, depending on the value of $x_{j,i} = x_{j,k}$, one of the conditions of the if-statements in lines 47 or 48 must hold. That is, p_k VBB-delivers, within $\mathcal{O}(1)$ CRWF, the same value as p_i does. □Lemma 5.3

Lemma 5.4 *(VBB-obligation). VBB-obligation holds.*

Proof of Lemma 5.4. Suppose all correct nodes, p_j, VBB-broadcast the same value v. We show that every correct node, p_i, VBB-delivers v from p_j. Since every correct node invokes VBB broadcast(v) within $\mathcal{O}(1)$ CRWF, p_j invokes brb_j[init][j].broadcast((j, v)) (line 34). Since $\exists_{S\subseteq\mathcal{P}} n-t \leq |S| : \forall p_k \in S : brb_i$[init][$k$].deliver() $\neq \perp$ holds due BRB-completion-1, the if-statement condition in line 36 holds within $\mathcal{O}(1)$ CRWF. And the multiset $\{brb_j$[init][k].deliver()$\}_{p_k\in\mathcal{P}}$ has at least $(n-2t)$ appearances of $(-, v)$. Thus, p_j BRB-broadcasts the message (valid, (j, True)) (line 38). Therefore, we can conclude that for any $\ell \in$ *Correct*, brb_ℓ[valid][j].deliver() = (j, True) holds within $\mathcal{O}(1)$ CRWF (due to BRB-validity and BRB-completion-1). Thus, within $\mathcal{O}(1)$ CRWF, none of the if-statement conditions at lines 41 to 46 hold. However, the one in line 47 holds, within $\mathcal{O}(1)$ CRWF, and only for the value v. Then, p_i VBB-delivers v as a VBB-broadcast from p_j. □Lemma 5.4

Lemma 5.5 *(VBB-justification). VBB-justification holds.*

Proof of Lemma 5.5. Let $i \in$ *Correct*. Suppose p_i VBB-delivers $m \notin \{\perp, \lightning\}$ in $a_i \in R$. We show that a correct p_j invokes vbb_j[j].broadcast(v) : $m = (j, v)$ in $a_j \in R$, such that a_j appears in R $\mathcal{O}(1)$ CRWF before a_i. Since $m \notin \{\perp, \lightning\}$, ($brb_i$[init][$j$].deliver() = $(j, v) \wedge brb_i$[valid][j].deliver() = (j, x)) (line 46) and $x \wedge vbbEq_i$(valid, v) (line 47) hold, because only line 47 returns (in VBB deliver()) neither $\perp$ nor $\lightning$ and it can only do so when the if-statement condition in line 46 does not hold. Since $vbbEq_i$(valid, v) holds and $n-2t \geq t+1$, at least one correct node, say, p_j had BRB-broadcast v (both for the init and valid phases in lines 34, and resp., 38), because R starts in a post-recycling state and by Theorem 5.1's Argument (2). Thus, a_j is before a_i. □Lemma 5.5

□Theorem 5.2

5.3. Convergence and completion of MVC

As in Section 5.1, Convergence is proven by showing BC-completion in executions that start in any state. Due to the availability of the recycling mechanism, once the task is completed, the system reaches a post-recycling state (by GMRS's mechanism [2,3], which Convergences within $\mathcal{O}(t)$ synchronous rounds). Theorem 5.6 counts communication rounds (without assuming execution fairness) and asynchronous cycles (while assuming execution fairness) using the CRWF/ACAF notation (Section 5.1.)

Theorem 5.6 demonstrates BC-completion via the proof of Lemmas 5.7 and 5.8. Recall Algorithm 4 returns $\perp$ when it needs to indicate that it has not completed its task. This is done by line 63. Lemma 5.7 shows that this cannot happen indefinitely. Recall that the if-statement conditions in lines 65 and 70 indicates the occurance of a traisient fault whereas the one in line 67 returns the decided value. Lemma 5.8 assumes that if-statement conditions in lines 65 and 70 do not hold and show that, within a bounded time, line 67's condition holds.

Theorem 5.6 *(BC-completion). Let R be Algorithm 4's execution in which all correct nodes invoke* propose() *within $\mathcal{O}(1)$ CRWF/ACAF. BC-completion is held during R.*

Proof of Theorem 5.6. We show that every correct node decides within $\mathcal{O}(1)$ CRWF/ACAF, *i.e.,* $\forall i \in$ *Correct* : result$_i() \neq \perp$.

Lemma 5.7. *Within $\mathcal{O}(1)$ CRWF/ACAF,* result$_i()$ *cannot return $\perp$ due to line 63.*

Proof of Lemma 5.7. Any correct node, p_i, asserts $bcO_i \neq \perp$, say, by invoking bcO_i.propose() (line 75). This is due to the assumption that all correct nodes invoke propose() within $\mathcal{O}(1)$ CRWF/ACAF, the definition of propose() (line 60), VBB-completion, and the presence of at least $(n-t)$ correct nodes, which implies that $(\exists S \subseteq \mathcal{P} : n-t \leq |S|) : \forall p_k \in S : vbb_i[k]$.deliver() $\neq \perp$ holds within $\mathcal{O}(1)$ CRWF/ACAF, and the if-statement condition in line 75 holds whenever $bcO_i = \perp$. Eventually, bcO_i.result() $\neq \perp$ (by the completion property of binary consensus). Thus, result$_i()$ cannot return $\perp$ due to the if-statement in line 63. □Lemma 5.7

If result$_i()$ returns due to the if-statement in lines 65 to 70, result$_i() \neq \perp$ is clear. Therefore, the rest of the proof focuses on showing that, within $\mathcal{O}(1)$ CRWF/ACAF, one of these three if-statement conditions must hold; thus, the last return statement (of $\perp$ in line 72) cannot occur, see Lemma 5.8.

Lemma 5.8. *Suppose lines 65 and 70's conditions never hold* w.r.t. *any correct p_i. Within $\mathcal{O}(1)$ CRWF/ACAF, line 67's condition holds.*

Proof of Lemma 5.8. By VBB-completion, $mcWait_i()$ (line 57) must hold within $\mathcal{O}(1)$ CRWF/ACAF since there are $n-t$ correct and active nodes. Thus, by the lemma assumption that the if-statement condition in line 70 never holds in R, we know that, for any correct node p_i, True $\in bvO_i.binValues()$ holds within $\mathcal{O}(1)$ CRWF/ACAF, due to the properties of BV-broadcast (Section 2.2.2). Thus, there is at least one correct node, p_j, for which $sameValue_j()$ = True when BV-broadcasting in line 76. By VBB-uniformity, the if-statement condition in line 67 must hold, within $\mathcal{O}(1)$ CRWF/ACAF, *w.r.t.* any correct node p_i. □Lemma 5.8

□Theorem 5.6

5.4. Closure of MVC

Theorem 5.9's proof shows that no consistency test causes false error indications. Theorem 5.9 counts communication rounds (without assuming execution fairness) using the CRWF notation (Section 5.1). Theorem 5.2's proof expands the proof provided in Theorem 5.6 by establishing the fulfillment of all MVC properties by demonstrating BC-agreement, BC-validity, and BC-no-intrusion in a manner akin to the proof of the MR algorithm by considering the assumption that the system begins in a post-recycling state. However, adjustments the proof are necessary since Algorithm 4's structure differs somewhat from MR.

Theorem 5.9 *(MVC closure). Let R be an Algorithm 4's execution that starts in a post-recycling state and in which all correct nodes invoke* propose() *within $\mathcal{O}(1)$ CRWF. MVC requirements are held in R.*

Proof of Theorem 5.9. BC-completion holds (Theorem 5.6).

Lemma 5.10. *The BC-agreement property holds.*

Proof of Lemma 5.10. We show that no two correct nodes decide differently. For every correct node, p_i, bcO_i.result() $\neq \perp$ holds within $\mathcal{O}(1)$ CRWF (Theorem 5.6). By the agreement and integrity properties of binary consensus, bcO_i.result() = False implies BC-agreement (line 65). Suppose bcO_i.result() = True. The proof is implied since there is no correct node, p_i, and (faulty or correct) node p_k for which there is a value $w \notin \{\perp, \lightning, v\}$, such that $vbb_i[k]$.deliver() $= w$. This is due to $n-2t \geq t+1$ and *sameValue*()'s second clause (line 59), which requires v to be unique. □Lemma 5.10

Lemma 5.11. *The BC-validity property holds.*

Proof of Lemma 5.11. Suppose that all correct nodes propose the same value, v. The proof shows that v is decided. Since all correct nodes propose v, we know that v is validated (VBB-obligation). Also, all correct nodes VBB-deliver v from at least $n-2t$ different nodes (VBB-completion). Since $n-2t > t$, value v is unique. This is because no value v' can be VBB-broadcast only by faulty nodes and still be validated (VBB-justification). Thus, the non-$\perp$ values that correct nodes can VBB-deliver are v and $\lightning$. This means that $\forall i \in Correct : sameValue_i() =$ True, bcO_i.result() = True (binary consensus validity), and all correct nodes decide v. □Lemma 5.11

Lemma 5.12. *The BC-no-intrusion property holds.*

Proof of Lemma 5.12. Suppose $w \neq \lightning$ is proposed only by faulty nodes. The proof shows that no correct node decides w. By VBB-justification, no $p_i : i \in Correct$ VBB-delivers w.

Suppose that bcO_i.result() $\neq$ True. Thus, w is not decided due to the if-statement line 65. Suppose bcO_i.result() = True. There must be a node p_j for which $sameValue_j() =$ True, *i.e.,* v is decided due to the if-statement in line 67 and since there are at least $n-2t$ VBB-deliveries of v. Note that the if-statement condition in line 70 cannot hold during R since R starts in a post-recycling system state as well as due to lines 75 and 76, which use the same input value from *sameValue*(). This implies that $w \neq v$ cannot be decided since $n-2t > t$. □Lemma 5.12

□Theorem 5.9

6. Discussion

To the best of our knowledge, this paper presents the first SSBFT MVC algorithm for asynchronous message-passing systems. This solution is devised by layering SSBFT broadcast protocols. Our solution is based on a code transformation of existing (non-self-stabilizing) BFT algorithms into an SSBFT one. This transformation is achieved via careful analysis of the effect that arbitrary transient faults can have on the system's state, and via rigorous proofs. We hope the proposed solutions and studied techniques can facilitate the design of new building blocks, such as state-machine replication, for the Cloud and distributed ledgers.

The BC-no-intrusion requirement ensures that the agreed-upon value originates from a correct process, preventing consensus on adversarially crafted values. The proposed solution assumes that at least $t+1$ correct processes propose the same value to guarantee agreement under BC-no-intrusion. In comparison, this assumption is stronger than what is required for SSBFT binary consensus where one of the two values must appear in a distinguished majority of the correct processes when $t < n/3$—it highlights the asymmetry in fault tolerance between the binary and multivalued cases. Specifically, our design avoids reducing multivalued consensus to multiple binary consensus instances (*e.g.,* via bitwise decomposition), which may significantly increase communication costs. We believe that determining whether such an increase in communication cost can be circumvented while still solving SSBFT MVC under the BC-no-intrusion requirement remains an open problem.

CRediT authorship contribution statement

Romaric Duvignau: Writing – review & editing. **Michel Raynal:** Writing – review & editing. **Elad Michael Schiller:** Writing – review & editing, Writing – original draft, Methodology, Investigation, Formal analysis, Conceptualization.

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.

References

[1] R. Duvignau, M. Raynal, E.M. Schiller, Self-stabilizing Byzantine fault-tolerant repeated reliable broadcast, Theor. Comput. Sci. 972 (2023) 114070.
[2] 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: LNCS, vol. 12754, Springer, 2021, pp. 36–53.
[3] C. Georgiou, M. Raynal, E.M. Schiller, Self-stabilizing Byzantine-tolerant recycling, in: SSS, in: LNCS, vol. 14310, Springer, 2023, pp. 518–535.
[4] A. Mostéfaoui, M. Raynal, Signature-free broadcast-based intrusion tolerance: never decide a Byzantine value, in: OPODIS, in: LNCS, vol. 6490, 2010, pp. 143–158, Springer, an extended journal version appears in IEEE Trans. Parallel Distrib. Syst. 27 (4) (2016) 1085–1098.
[5] L. Lamport, R.E. Shostak, M.C. Pease, The Byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (3) (1982) 382–401.
[6] C. Dwork, N.A. Lynch, L.J. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (2) (1988) 288–323.
[7] M.C. Pease, R.E. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234.
[8] K.J. Perry, Randomized Byzantine agreement, in: Fourth Symposium on Reliability in Distributed Software and Database Systems, SRDS, 1984, pp. 107–118.
[9] M.J. Fischer, N.A. Lynch, M. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382.
[10] M. Raynal, Fault-Tolerant Message-Passing Distributed Systems - an Algorithmic Approach, Springer, 2018.
[11] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[12] K. Altisen, S. Devismes, S. Dubois, F. Petit, Introduction to Distributed Self-Stabilizing Algorithms, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2019.
[13] S. Dolev, Self-Stabilization, MIT Press, 2000.
[14] A. Arora, M.G. Gouda, Closure and convergence: a foundation of fault-tolerant computing, IEEE Trans. Softw. Eng. 19 (11) (1993) 1015–1027.
[15] M. Correia, G.S. Veronese, N.F. Neves, P. Veríssimo, Byzantine consensus in asynchronous message-passing systems: a survey, Int. J. Crit. Comput. Based Syst. 2 (2) (2011) 141–161.
[16] I. Abraham, D. Malkhi, K. Nayak, L. Ren, M. Yin, Sync HotStuff: simple and practical synchronous state machine replication, in: IEEE Symposium on Security and Privacy, SP'20, 2020, pp. 106–118.
[17] M. Ben-Or, B. Kelmer, T. Rabin, Asynchronous secure computations with optimal resilience, in: ACM PODC, 1994, pp. 183–192.
[18] M. Correia, N.F. Neves, P. Veríssimo, From consensus to atomic broadcast: time-free Byzantine-resistant protocols without signatures, Comput. J. 49 (1) (2006) 82–96.
[19] N.F. Neves, M. Correia, P. Veríssimo, Solving vector consensus with a wormhole, IEEE Trans. Parallel Distrib. Syst. 16 (12) (2005) 1120–1131.
[20] K. Antoniadis, A. Desjardins, V. Gramoli, R. Guerraoui, I. Zablotchi, Leaderless consensus, in: Distributed Computing Systems, ICDCS, IEEE, 2021, pp. 392–402.
[21] M. Castro, B. Liskov, Practical Byzantine fault tolerance and proactive recovery, ACM Trans. Comput. Syst. 20 (4) (2002) 398–461.
[22] 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.
[23] S. Dolev, R.I. Kat, E.M. Schiller, When consensus meets self-stabilization, J. Comput. Syst. Sci. 76 (8) (2010) 884–900.
[24] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing multivalued consensus in asynchronous crash-prone systems, Theor. Comput. Sci. 1022 (2024) 114886.
[25] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing indulgent zero-degrading binary consensus, Theor. Comput. Sci. 989 (2024) 114387.
[26] 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.
[27] A. Mostéfaoui, M. Hamouma, M. Raynal, Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages, in: ACM Principles of Distributed Computing, PODC '14, 2014, pp. 2–9.
[28] 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: Stabilization, Safety, and Security of Distributed Systems SSS'16, 2016, pp. 36–53.
[29] A. Binun, S. Dolev, T. Hadad, Self-stabilizing Byzantine consensus for blockchain, in: CSCML, 2019, pp. 106–110.
[30] S. Dolev, C. Georgiou, I. Marcoullis, E.M. Schiller, Self-stabilizing Byzantine tolerant replicated state machine based on failure detectors, in: CSCML, 2018, pp. 84–100.
[31] S. Dolev, O. Liba, E.M. Schiller, Self-stabilizing Byzantine resilient topology discovery and message delivery, in: SSS, in: LNCS, vol. 8255, Springer, 2013, pp. 351–353.
[32] 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.
[33] R. Duvignau, M. Raynal, E.M. Schiller, Self-stabilizing Byzantine multivalued consensus: (extended abstract), in: 25th International Conference on Distributed Computing and Networking, ICDCN, ACM, 2024, pp. 12–21.
[34] R. Duvignau, M. Raynal, E.M. Schiller, Self-stabilizing Byzantine multivalued consensus, CoRR, arXiv:2311.09075, 2023.
[35] G. Bracha, S. Toueg, Resilient consensus protocols, in: PODC, ACM, 1983, pp. 12–26.
[36] S. Toueg, Randomized Byzantine agreements, in: Proceedings of the Third Annual ACM Principles of Distributed Computing, 1984, pp. 163–178.
[37] N. Alhaddad, S. Das, S. Duan, L. Ren, M. Varia, Z. Xiang, H. Zhang, Balanced Byzantine reliable broadcast with near-optimal communication and improved computation, in: PODC, ACM, 2022, pp. 399–417.
[38] S. Das, Z. Xiang, L. Ren, Asynchronous data dissemination and its applications, in: CCS, ACM, 2021, pp. 2705–2721.
[39] A. Maurer, S. Tixeuil, Self-stabilizing Byzantine broadcast, in: Reliable Distributed Systems, SRDS, 2014, pp. 152–160.