Self-stabilizing Byzantine Multivalued Consensus: (extended abstract)

Self-stabilizing Byzantine Multivalued Consensus

(extended abstract)

Romaric Duvignau
duvignau@chalmers.se
Chalmers University of Technology
Sweden

Michel Raynal
michel.raynal@irisa.fr
IRISA, University Rennes 1
France

Elad M. Schiller
elad@chalmers.se
Chalmers University of Technology
Sweden

ABSTRACT

Consensus, abstracting a myriad of problems in which processes have to agree on a single value, is one of the most celebrated problems of fault-tolerant distributed computing. Consensus applications include fundamental services for the environments of the Cloud and Blockchain, 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 a value 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 at the design of 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 the occurrence of *arbitrary transient-faults*. 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 intrusion-tolerant multivalued consensus for asynchronous message-passing systems prone to Byzantine failures. Our solution has a $O(t)$ stabilization time from arbitrary transient faults.

ACM Reference Format:
Romaric Duvignau, Michel Raynal, and Elad M. Schiller. 2024. Self-stabilizing Byzantine Multivalued Consensus: (extended abstract). In *25th International Conference on Distributed Computing and Networking (ICDCN '24), January 04–07, 2024, Chennai, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3631461.3631540

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, and further 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 wider range of faults, including both benign and malicious failures, as well as arbitrary, transient, and possibly unforeseen violations 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.

ICDCN '24, January 04–07, 2024, Chennai, India

ACM ISBN 979-8-4007-1673-7/24/01.
https://doi.org/10.1145/3631461.3631540

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 rather 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.

Byzantine fault-tolerance (BFT). Lamport *et al.* [28] say that a process commits a *Byzantine* failure if it deviates from the algorithm instructions, say, by deferring or omitting messages that were sent by the algorithm or sending fake messages, which the algorithm never sent. Such malicious behaviors include crashes and may be the result of hardware or software malfunctions as well as coordinated malware attacks. In order to safeguard against such attacks, Mostéfaoui and Raynal [33], 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, when it is not possible to decide on a value, the error symbol $\lightning$ is returned instead. For the sake of deterministic solvability [23, 28, 35, 36], 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) [24]. 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 [33], *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 that has no guarantees on the 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, as well as 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 [15] assumes that these violations bring the system to an arbitrary state from which a *self-stabilizing system* should recover [3, 16]. *I.e.,* Dijkstra requires (i) recovery after the last occurrence of a transient-fault and (ii) once the system has recovered, it must never violate the task requirements. Arora and Gouda [5] 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, which left the system in an arbitrary state.*

1.2 Related work

Ever since the seminal work of Lamport, Shostak, and Pease [28] four decades ago, BFT consensus has been an active research subject, see [13] and references therein. The recent rise of distributed ledger technologies, *e.g.,* [1], brought phenomenal attention to the subject. We aim to provide a degree of dependability that is higher than existing solutions.

Ben-Or, Kelmer, and Rabin [6] were the first to show that BFT MVC can be reduced to Binary consensus. Correia, Neves, and Veríssimo [12, 34] later established the connection between intrusion tolerance and Byzantine resistance. These ideas form the basis of the MR algorithm [33]. MR is a leaderless consensus algorithm [4] and as such, it avoids the key weakness of leader-based algorithms [11] when the leader is slow and delays termination. There exist self-stabilizing solutions for MVC but only crash-tolerant ones [9, 18, 25, 29, 30], whereas, the existing BFT solutions are not self-stabilizing [37]. For example, the recent self-stabilizing crash-tolerant MVC in [30] solves a less challenging problem than the SSBFT problem studied here since it does not account for malicious behaviors. Mostéfaoui, Moumen, and Raynal [32] (MMR in short) presented BFT algorithms for solving Binary consensus using common coins, of which [26, 27] 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.* [7, 8] for synchronous systems and Dolev *et al.* [17] for practically-self-stabilizing partially-synchronous systems. Note that both Binun *et al.* and Dolev *et al.* study another problem for another kind of system setting. In [19], the problems of SSBFT topology discovery and message delivery were studied. Self-stabilizing atomic memory under semi-Byzantine adversary is studied in [20].

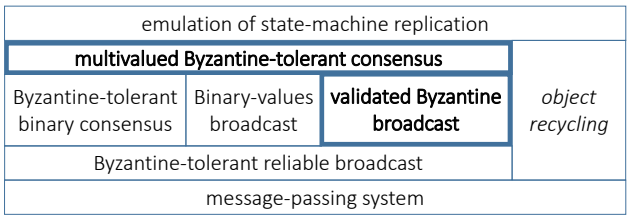

Figure 1: We assume the availability of SSBFT protocols (cf. Definition 1.2) for Binary consensus and object recycling. The studied problems appear in boldface fonts. The other layers, BRB, BV-broadcast, and state machine replication, are in plain font.

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, which is 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. If the agreed value indicates the existence of at least one non-faulty process that has passed the test, then 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 no such indication is represented by the agreed value, 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 [22] (referred to as DRS) proposed a transformation for converting the Byzantine Reliable Broadcast (referred to as BRB) algorithm, originally introduced by Bracha and Toueg [10], into a Self-Stabilizing BFT (in short, SSBFT) variation. Another transformation, proposed by Georgiou, Marcoullis, Raynal, and Schiller [26, 27] (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 that is proposed by at least one non-faulty process, which corresponds 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, which was not designed to tolerate transient faults, to wait indefinitely for messages that are never sent. 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 correctness proof 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 the occurrence of 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 for 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 is built upon the previous works of DRS [22] and GMRS [26, 27] (which addressed different problems than the one under study), we encounter similar challenges in the transformation of code from non-self-stabilizing to self-stabilizing algorithms. Nevertheless, achieving the desired self-stabilizing properties in our construction necessitates a thoughtful combination of SSBFT building blocks and a meticulous analysis of the transformed algorithms. This combination process cannot be derived from the DRS and GMRS transformations. The self-stabilizing issues to tackle that are inherent to the studied algorithms are further explained in Section 4.1.1 for the VBB algorithm and in 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 obtain this new self-stabilizing algorithm via a transformation of 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 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 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 taking into account also the stabilization time of the underlying communication abstractions, the recycling mechanism stabilizes within $O(t)$ synchronous rounds.

The communication costs of the studied algorithm, *i.e.,* 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 for making sure that the value decided by the SSBFT Binary consensus object remains consistent until the proposed SSBFT solution completes 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.

Due to the page limit, some of the proof details appear in the complementary technical report [21]. For the reader's convenience, all abbreviations are listed below. Glossary: **ACAF**: asynchronous cycles (while assuming fairness); **BFT**: (non-stabilizing) Byzantine fault-tolerant; **BRB**: Byzantine-tolerant Reliable Broadcast; **BV-broadcast**: Binary-values broadcast; **CRWF**: communication rounds (without assuming fairness); **DRS**: SSBFT BRB by Duvignau, Raynal, and Schiller [22]; **GMRS**: SSBFT MMR by Georgiou, Marcoullis, Raynal, and Schiller [26] for Binary consensus and BV-broadcast; **MR**: the studied solution by Mostéfaoui and Raynal [33]; **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.

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 communication channel, $channel_{j,i}$, that, at any time, has at most $\mathsf{channelCapacity} \in Z^+$ packets on transit from p_j to p_i (this assumption is due to a known impossibility [16, Chapter 3.2]).

We use the *interleaving model* [16] for representing the asynchronous execution of the system. The node's program is a sequence

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

2.1 The fault model and self-stabilization

We 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 calculate its state according to the algorithm. However, once a node (that is captured by the adversary) sends a message, the adversary can modify the message in any way, delay it for an arbitrarily long period 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 [28, 35, 38], we limit the number, t, of nodes that can be captured by the adversary, *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 that they can corrupt the system state arbitrarily (while keeping the program code intact). The occurrence of a transient fault is rare. Thus, we assume that the last arbitrary transient fault occurs before the system execution starts [16]. 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.

2.1.3 Dijkstra's self-stabilization. The *legal execution* (*LE*) set refers to all executions in which the problem requirements hold. 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 [15] requires $\forall R : \exists R' : R = R' \circ R_{legal} \wedge R_{legal} \in LE \wedge |R'| \in Z^+$, 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. The time between the invocation of an operation (such as consensus or broadcast) and the occurrence of all required deliveries is called operation latency. 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, once the convergence period is over, no fairness assumption is needed. 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 [16, Chapter 2.3]. Also, the study 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 is able to start and complete at least a constant number, num_b, of round-trips with every correct node. The second asynchronous communication round of R is the first round of the suffix R'', and so forth.

2.2 Building Blocks

Following Raynal [37], 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 [26, 27]) as well as SSBFT BRB (DRS [22]).

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$(k) : p_k \in \mathcal{P}$ operations.

Definition 2.1. *The operations* broadcast(v) *and* deliver(k) *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. All correct nodes BRB-deliver from p_i eventually.*
- ***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 [22], which stabilizes within $O(1)$ asynchronous cycles. Such implementation lets $p_i \in \mathcal{P}$ to use the operation deliver$_i(k)$ for retrieving the current return value, v, of the BRB broadcast from $p_k \in \mathcal{P}$. Before the completion of the task of the deliver$_i(k)$ operation, v's value is $\perp$. This way, whenever deliver$_i(k) \neq \perp$, node p_i knows that the task is completed and the returned value can be used. There are several BRB implementations [2, 14, 31] that satisfy different requirements than the ones in Definition 2.1, which is taken from the textbook [37].

Note that existing non-self-stabilizing BFT BRB implementations, *e.g.,* [37, Ch. 4], consider another kind of 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 the message. The interface proposed in [22] 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() is straightforward — it checks the condition in the if-statement on line 49 of Algorithm 4 in [22]. If the condition is met, it returns False, otherwise, it returns True.

2.2.2 SSBFT Binary-values Broadcast (BV). This is an all-to-all broadcast operation of Binary values. This abstraction uses the operation, bvBroadcast(v), which is assumed to be invoked independently (*i.e.,* not necessarily simultaneously) by all the correct nodes, where $v \in V$. For the sake of a simpler presentation of our solutions, we prefer $V = \{\text{False}, \text{True}\}$ over the traditional $V = \{0, 1\}$ presentation. The set of values that are BV-delivered 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 $\exists s \subseteq \{\text{False}, \text{True}\} : s \neq \emptyset \wedge \forall i \in Correct : binValues_i() = s$ and the set s does not include values that were BV-broadcast only by Byzantine nodes. The SSBFT BV-broadcast solution in [26] stabilizes within $O(1)$ 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 $v_i \in V = \{False, True\}$ via an invocation of* 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 $v \in \{False, True\}$ 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 [26], which stabilizes within $O(1)$ 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 {False, True}. The proposed SSBFT MVC algorithm returns $\lightning$ whenever the SSBFT Binary consensus returns $\lightning$ (cf. line 66 of Algorithm 4).

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

```
1  operation vbbBroadcast(v) begin
2      BRB-broadcast INIT(i, v);
3      wait |rec| ≥ n−t where rec is the multiset of
         BRB-delivered values;
4      BRB-broadcast VALID(i, (equal(v, rec) ≥ n−2t));
5  foreach p_j ∈ P execute concurrently do
6      wait INIT(j, v) and VALID(j, x) BRB-delivered from p_j;
7      if x then
8          {wait (equal(v, rec) ≥ n−2t); d ← v}
9      else
10         {wait (differ(v, rec) ≥ t+1); d ← ↯}
11     vbbDeliver(d) at p_i as the value VBB-broadcast by p_j;
```

2.2.4 The Recycling Mechanism and Recyclable Objects. Just as MR, we do not focus on the management of consensus invocations since we assume the availability of a mechanism for eventually recycling all MVC objects that have completed their tasks. GMRS [26, 27] implement such subroutine. We review their subroutine and explain how they construct recyclable objects.

GMRS implements consensus objects using a storage of constant size allocated at program compilation time. Since these objects can be instantiated an unbounded number of times, it is necessary to reuse the storage once a consensus is reached. This should occur only after each correct node received the decided value via result().

To facilitate this, GMRS assumes that the object has two meta-statuses: *used* and *unused*. The *unused* status indicates the availability of objects that were either never used or are no longer in current use. GMRS specifies that recyclable objects must implement an interface function called wasDelivered(), which returns 1 after the result delivery. Recycling is triggered by the recycling mechanism, which invokes recycle() at each non-faulty node, setting the meta-status of the corresponding consensus object to *unused*.

GMRS defines recyclable object construction as a task that requires eventual agreement on the value of wasDelivered(). In detail, if a non-faulty node p_i reports delivery (*i.e.,* wasDelivered$_i() = 1$), then all non-faulty nodes will eventually report delivery as well. We clarify that during the recycling process, *i.e.,* when at least one non-faulty node invokes recycle(), there is no need to maintain agreement on the values of wasDelivered().

GMRS requires us to implement recycle() by locally setting the algorithm to its predefined post-recycling state, see Sections 4.1.2 and 4.2.2. Also, our solution implements the operation result(), which facilitates the implementation of wasDelivered() using the same construction proposed by GMRS in [26, 27]. By implementing GMRS' interfaces, we borrow GMRS correctness properties since the studied problem and the structure of the proposed algorithms are very similar.

GMRS implements a recycling service using a synchronous SSBFT consensus that allows all non-faulty nodes to reuse the object

immediately after the process returns from recycle(). GMRS's recycling facilitates the transformation of the non-self-stabilizing BFT MR algorithm to an SSBFT one. The transformation concentrates on assuring operation completion since once all objects have been recycled, the system reaches its *post-recycling state*, which has no trace of stale information, *i.e.,* Convergence holds. As mentioned in Section 1.4, the effect of these assumptions can be mitigated by letting 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 that is 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 a value proposed only by Byzantine nodes. Also, if there is a value, $v \in V$, that all correct nodes propose, then v is decided. Otherwise, the decided value is either a value proposed by the correct nodes or 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 all nodes to all nodes. It allows the operation, vbbBroadcast(v) and raises the event vbbDeliver(d), for VBB-broadcasting, and resp., VBB-delivering.

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-broadcast m.
  - **VBB-obligation.** Suppose all correct nodes VBB-broadcast the same v. All correct nodes VBB-delivers v from each correct node.
- **VBB-uniformity.** Let $p_i : i \in$ *Correct*. Suppose 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 vbbBroadcast$_i(m_i)$ invocation by every correct $p_i \in \mathcal{P}$. It also includes vbbDeliver() 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.

3.1.2 Implementing VBB-broadcast. Algorithm 1 presents the studied VBB-broadcast.

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

```
12 variables: bcO; // Binary consensus object, ⊥ is the initial state
13 macro sameValue() do return ∃v ≠ ↯ : equal(v, rec) ≥ n−2t ∧ rec = {v' ≠ ↯} where rec is a multiset of the (n−t) values VBB-delivered (line 16);
14 operation propose(v) begin
15     vbbBroadcast EST(v);
16     wait EST(•) messages VBB-delivered from (n−t) different nodes;
17     if ¬bcO.propose(sameValue()) then
18         return ↯
19     else
20         wait (∃v ≠ ⊥: equal(v, rec) ≥ n−2t) then return v
```

Notation: Denote by $equal(v, rec)$ and $differ(v, rec)$ the number of items in multiset rec that are equal to, and resp., different from v.
Overview: Algorithm 1 invokes BRB-broadcast twice in the first part of the algorithm (lines 1 to 4) and then VBB-delivers messages from nodes in the second part (lines 5 to 11).

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 2 to 3), which p_i collects in the multiset rec_i. In line 2, 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 4). 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 3) does not keep rec_i from growing.

Algorithm 1's second part (lines 5 to 11) 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 = True case (line 7).** 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, because this implies $equal(v_j, rec_i) \geq t+1$ since $n-2t \geq t+1$.
(2) **The x_j = False case (line 10).** 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 note cannot attest 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 that have 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 (Section 1.1) includes the requirements of BC-validity, BC-agreement, and BC-completion (Section 1.1) as well as the BC-no-Intrusion property (Section 1.1).

3.2.2 Implementation. Node p_i waits for EST() messages from $(n-t)$ different nodes after it as VBB-broadcast its own value (lines 15 to 16). It holds all the VBB-delivered values in the multiset rec_i (line 13) before testing whether rec_i includes (1) non-$\lightning$ replies from at least $(n-2t)$ different nodes, and (2) exactly one non-$\lightning$ value v (line 13). The test result is proposed to the Binary consensus object, bcO (line 17).

Once consensus is reached, p_i decides according to the consensus result, bcO_i.result(). Specifically, if bcO_i.result() = False, p_i returns the error symbol, $\lightning$, since there is no guarantee that any correct node was able to attest to the validity of the proposed value. Otherwise, p_i waits until it received EST(v) messages that have identical values from at least $(n-2t)$ different nodes (line 20) before returning that value v. Note that some of these $(n-2t)$ messages were already VBB-delivered at line 16. The proof in [33] shows that any correct node that invokes bcO_i.propose(True) does so if all correct nodes eventually VBB-deliver identical values at least $(n-2t)$ times. Then, any correct node can decide on the returned value for the MVC object once it also VBB-delivers identical values at least $(n-2t)$ times.

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 vbbBroadcast(v) allows the invocation of a VBB-broadcast instance with the value v. Node p_i VBB-delivers messages from p_k via vbbDeliver$_i(k)$.

4.1.1 Algorithm 1's invariants that transient faults can violate. Transient faults can violate the following invariants, which our SSBFT solution addresses via consistency tests.

(1) Node p_i's state must not encode the occurrence of BRB execution of phase `valid` (line 4) without encoding BRB execution of phase `init` (line 2). Algorithm 3 addresses this concern by informing that the VBB object has an internal error (line 38). This way, indefinite blocking of the application is avoided.
(2) Define the phase types, vbbMSG := {`init`, `valid`} (line 21) for BRB dissemination of INIT(), and resp., VALID() messages in Algorithm 1. For a given phase, $phs \in$ vbbMSG, the BRB message format must follow the one of BRB-broadcast of phase phs, as in lines 2 and 4. In order to avoid blocking, the VBB object informs about an internal error (lines 42 and 43).
(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 VBB-deliver 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 33). 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 47), see Section 4.1.5 for more details.

4.1.2 Local variables. The array brbvbbMSG[vbbMSG][$\mathcal{P}$] holds BRB object, which disseminate VBB-broadcast messages, *i.e., brb*[`init`][] and *brb*[`valid`][] store the INIT(), and resp., VALID() messages in Algorithm 1. The second dimension of the array *brb*[][] allows us to implement one VBB object per node as this is needed for Algorithm 4. Thus, after the recycling of these objects (Section 2.2.4) or before they ever become active, each of the $2n$ BRB objects has the value $\perp$. For a given $p_i \in \mathcal{P}$, brb_i[-][i] becomes active via the invocation of brb_i[-][i].broadcast(v) (which also leads to brb_i[-][i] $\neq \perp$) or the arrival of BRB protocol messages, say, from p_j (which leads to brb_i[-][j] $\neq \perp$). Once a BRB message is delivered from p_ℓ (in the context of phase $phs \in$ vbbMSG and VBB broadcast from p_k), a call to $brb_i[phs][k]$.deliver(ℓ) retrieves the delivered message.

4.1.3 Macros. The macro *vbbWait*(k,phs) (line 26) serves at if-statement conditions in lines 47 and 33 when the proposed transformation represents the exit conditions of the wait operations in lines 3 and 10. Specifically, given a phase, phs, it tests whether there is a set S that includes at least $n-t$ different nodes from which there is a message that is ready to be BRB-delivered. The macros *vbbEq*(k,phs, v) and *vbbDiff*(k,phs, v) are detailed versions of the equal, and resp., differ conditions used in lines 4 and 7, resp., line 10. They check whether the value v equals to, resp., differs from at least $n-2t$, resp., $t+1$ received BRB messages of phase phs.

4.1.4 The vbbBroadcast() *operation (line 31).* As in line 2 in Algorithm 1, vbbBroadcast(v)'s invocation (line 31) leads to the invocation of *brb*[`init`][i].broadcast((i,v)). Algorithm 4 uses line 33 for implementing the logic of lines 3 and 4 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*(k,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 $vbbWait_i(i$,`init`$) \wedge brb_i$[`init`][i].hasTerminated(), where the second clause indicates that brb_i[`init`][i] has terminated (Section 2.2.1), and 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).

4.1.5 The vbbDeliver() *operation (lines 36 and 48).* This operation (lines 36 to 48) is based on lines 5 and 11 in Algorithm 1 together with a few of consistency tests (Section 4.1.1).

Line 38 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

Algorithm 3: SSBFT VBB-broadcast; code for p_i

21 **types:** vbbMSG := {init, valid}; // BRB object phases
22 **variables:**
23 brb[vbbMSG][$\mathcal{P}$] // brb[init][$\mathcal{P}$] and brb[valid][$\mathcal{P}$] are the two BRB objects. Upon recycling, [[$\perp,\ldots,\perp$], [$\perp,\ldots,\perp$]] is assigned
24 **macros:**
25 // exit conditions of wait operations in lines 3 and 10
26 $vbbWait(k, phs) := \exists S \subseteq \mathcal{P} : n{-}t \leq |S| : \forall p_\ell \in S :$ ($brb[phs][k]$.deliver(ℓ) $\neq \perp$)
27 // detailed version of equal condition in lines 4 and 7
28 $vbbEq(k, phs, v) := \exists S \subseteq \mathcal{P} : n{-}2t \leq |S| : \forall p_\ell \in S : ((\text{-}, v) =$ $brb[phs][k]$.deliver(ℓ));
29 // detailed version of the differ condition used in line 10
30 $vbbDiff(k, phs, v) := \exists S \subseteq \mathcal{P} : t{+}1 \leq |S| : \forall p_\ell \in S : (v \neq$ $brb[phs][k]$.deliver(ℓ));
31 **operation** vbbBroadcast(v) **do** brb[init][i].broadcast((i, v)) // cf. line 2
32 **do-forever begin**
33 **if** $vbbWait(i, (i,$ init$))\wedge$ brb[init][i].hasTerminated() **then**
34 **let** $v := brb$[valid][i].deliver(i);
35 brb[valid][i].broadcast(($i, vbbEq(i,$ init$, v)$)) // cf. line 4
36 **operation** vbbDeliver(k) **begin**
37 // case (I) of the consistency tests (Section 4.1.1)
38 **if** brb[init][k].deliver(k) $= \perp\wedge$ brb[valid][k].deliver(k) $\neq \perp$ **then return** $\lightning$;
39 // wait until p_j's BRB objects have delivered, cf. line 6
40 **if** brb[init][k].deliver(k) $= \perp\vee$ brb[valid][k].deliver(k) $= \perp$ **then return** $\perp$;
41 // lines 42 and 43 are case (II) of consistency tests (Section 4.1.1)
42 **if** $\exists phs \in$ vbbMSG : $brb[phs][k]$.deliver(k) $= (j, \text{-})\wedge$ $j \neq k$ **then return** $\lightning$;
43 **if** $\neg((brb$[init][k].deliver(k) $= (k, v) \wedge v \in V) \wedge (brb$[valid][$k$].deliver($k$) $= (k, x) \wedge x \in$ {False, True})) **then return** $\lightning$;
44 **else if** $x \wedge vbbEq(k,$ valid$, v)$ **then return** v; // cf. line 7
45 **else if** $\neg x \wedge vbbDiff(k,$ valid$, v)$ **then return** $\lightning$; // cf. line 10
46 // case (III) of the consistency tests (Section 4.1.1)
47 **else if** $vbbWait(k,$ valid$)$ **then return** $\lightning$;
48 **return** $\perp$ // vbbDeliver(k) is incomplete

brb[valid][k] before brb[init][k]'s termination, an error is indicated via the return of $\lightning$. Line 40 follows line 6'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. In case any of them is not ready to complete, the operation returns $\perp$.

The if-statements in lines 42 and 43 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.

The if-statements in lines 44 to 45 implement the logic of lines 7 to 10 in Algorithm 1. The logic of these lines is explained in items 1, and resp., 2 in Section 3.1.1. Similar to line 7 in Algorithm 1, x_i (line 43) is the value that p_i BRB-delivers from p_k via the BRB object brb_i[valid]. As mentioned, the macro *vbbDiff* () is a detailed implementation of *differ*() used by Algorithm 1.

The if-statement in line 47 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 38 to 45 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 let the Binary consensus object to encode a decided value that was never proposed, *i.e.,* this violates BC-validity.

Any SSBFT solution needs to address this concern since the MVC object can block indefinitely if *bcO* decides True when $\forall p_j : j \in Correct : sameValue_j() =$ False holds. As we explain next, our implementation BV-broadcasts (line 72) for testing the consistency of the SSBFT Binary consensus object (line 66). 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*. Each 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 (Sections 2.2.1 and 2.2.2, resp.).

4.2.3 Macros. The macro *mcWait*() (line 53) serves in lines 66 and 70 when the proposed transformation represents the exit conditions of the wait operations in lines 16 and 20. Specifically, it tests whether there is a set $S \subseteq \mathcal{P}$ that includes at least $n{-}t$ different nodes from which there is a message that is ready to be VBB-delivered. The macro *sameValue*() is an adaptation of the macro in line 13, 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 14 and 20 in Algorithm 2 is implemented by lines 56 to 71 in Algorithm 4. *I.e.,* the invocation of propose(v) (line 56) leads to the VBB-broadcast of v.

The logic of lines 16 and 17 in Algorithm 2 is implemented by lines 70 and 71, resp. In detail, recall from Section 4.2.3 that *mcWait*() (line 70) allows waiting until there are at least $n-t$ different nodes from which p_i is ready to VBB-deliver a message. Then, if

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

```
49 variables:
50 bvO := ⊥; // Binary-values object. Recycling assigns ⊥
51 bcO := ⊥; // Binary consensus object. Upon recycling, assign ⊥
52 macros:
53 // exit conditions of the wait operation in line 16
   mcWait() := ∃S ⊆ 𝒫 : n−t ≤ |S| : ∀p_k ∈ S : (vbbDeliver(k) ≠ ⊥);
54 // adapted version of the same macro in line 13
55 sameValue() do return (∃v ∉ {⊥, ↯} : ∃S′ ⊆ 𝒫 : n−2t ≤ |S′| : ∀p_k′ ∈ S′ : (vbbDeliver(k′) = v))∧
   (|{vbbDeliver(k) ∉ {⊥, ↯} : p_k ∈ 𝒫}| = 1);
56 operation: propose(v) do vbbBroadcast(v);
57 operation: result() begin
58     // test whether result() is not ready to complete
59     if bcO = ⊥ ∨ bcO.result() = ⊥ then
60         return ⊥
61     else if ¬bcO.result() then // cf. line 18
62         return ↯
63     else if ∃v ∉ {⊥, ↯} : ∃S′ ⊆ 𝒫 : n−2t ≤ |S′| : ∀p_k′ ∈ S′ : (vbbDeliver(k′) = v) then // cf. line 20
64         return v
65     // perform a consistency test, cf. Section 4.2.1
66     else if mcWait()∧True ∉ bvO.binValues() then
67         return ↯
68     return ⊥; // result() is not ready to complete
69 do-forever begin
70     if mcWait() then // cf. line 16
71         if bcO = ⊥ then bcO.propose(sameValue()); // cf. line 17
72         bvO.broadcast(sameValue()); // assist with the consistency test in line 66
```

$bcO = \perp$ (*i.e.,* the Binary consensus object was not invoked), line 71 uses bcO to propose the returned value from *sameValue*(). Recall from Section 4.2.3, the macro *sameValue*() (line 55) implements the one in line 13 (Algorithm 2), see Section 3.2.2 for details.

Line 72 facilitates the implementation of the consistency test (Section 4.2.1) by BV-broadcasting the returned value *sameValue*(). This way it is possible to detect the case in which all correct nodes BV-broadcast a value that is, due to a transient fault, different than bcO's decided one. This is explained when we discuss line 66.

The operation result() (lines 57 to 68) returns the decided value, which lines 18 and 20 implement in Algorithm 2. It is a query-based operation, just as deliver() (cf. text just after Definition 2.1). Thus, line 59 considers the case in which the decision has yet to occur, *i.e.,* it returns $\perp$. Line 61 considers the case that line 17 (Algorithm 2) deals with and returns the error symbol, $\lightning$. Line 63 implements line 20 (Algorithm 2), *i.e.,* it returns the decided value. Line 66 performs a consistency test for the case in which the if-statment conditions in lines 59 to 63 hold, there are VBB-deliveries from at least $n - t$ different nodes (*i.e.,* $mcWait_i()$ holds), and yet there are no correct node, say p_j, reports to p_i, via BV-broadcast, that the predicate $sameValue_j()$ holds. Whenever none of the conditions of the if-statements in lines 59 to 66 hold, line 68 returns $\perp$.

5 CORRECTNESS

As explained in Section 2.2.4, we demonstrate Convergence (Theorem 5.1) by showing that all operations eventually complete 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, there is a need to prove the properties of completion and Convergence before demonstrating the Closure property. Due to the page limit, the proof of the Closure properties and the MVC's Convergence properties appear in the complementary technical report [21].

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, p_i, and p_j, a non-$\perp$ value is returned from $\text{vbbDeliver}_j(i)$. 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 [26, 27], Convergence is completed with $O(t)$ synchronous rounds. We introduce the *CRWF/ACAF* notation since the arguments of Theorem 5.1 can be used for demonstrating different properties under different assumptions. Specifically, Theorem 5.1 demonstrates that VBB-completion occurs within $O(1)$ communication rounds (Section 2.1.4) without assuming execution fairness but assuming that 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 Fairness (CRWF), we write 'within $O(1)$ CRWF'. Theorem 5.1 also demonstrates Convergence within $O(1)$ 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 Fairness (ACAF), we say, in short, 'within $O(1)$ ACAF'. Moreover, when the same arguments can be used in both cases, we say 'within $O(1)$ CRWF/ACAF'.

Theorem 5.1 (VBB-completion). *Suppose all correct nodes invoke* vbbBroadcast() *within* $O(1)$ *CRWF/ACAF. Then,* $\forall_{i,j \in Correct}$: $\text{vbbDeliver}_j(i) \neq \perp$ *holds within* $O(1)$ *CRWF/ACAF.*

Proof of Theorem 5.1 Let $i \in Correct$. Suppose either p_i VBB-broadcasts m in R or $\exists phs \in \text{vbbMSG} : brb_j[phs][i] \neq \perp$ 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 38 to 47 and showing that, within $O(1)$ CRWF/ACAF, one of the if-statements (that returns a non-$\perp$) in lines 38 to 47 holds.

Argument 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 $O(1)$ CRWF/ACAF, $\forall k \in Correct : brb_k[phs][\ell] \neq \perp$ holds. The proof is implied by BRB-completion-1, BRB-completion-2, and the $O(1)$ stabilization time of SSBFT BRB, cf. Section 2.2.1.

Argument 2. *Suppose in R, the if-statement condition in line 38 does not hold. Within* $O(1)$ *CRWF/ACAF,* brb_i[valid][i] $\neq \perp$ *holds.* By the theorem assumption that all correct nodes invoke the operation vbbBroadcast() within $O(1)$ CRWF/ACAF, the BRB properties (Definition 2.1), and that there are at least $(n-t)$ correct nodes, the if-statement condition in line 33 holds within $O(1)$ CRWF/ACAF. Then, p_i invokes the operation brb_i[valid][i]broadcast(-).

Argument 3. *Suppose* brb_i[valid][i] $\neq \perp$ *holds in R's starting state. Within* $O(1)$ *CRWF/ACAF, either the if-statement condition in line 38 holds or the one in line 40 cannot hold.* The proof is implied by Algorithm 4's code, BRB-completion, and arguments 1 and 2.

Argument 4. *Within* $O(1)$ *CRWF/ACAF,* vbbDeliver$_j(i) \neq \perp$ *holds.* Suppose the if-statement conditions in lines 38 to 45 never hold. By *vbbWait*()'s definition (line 26), BRB properties (Definition 2.1), the presence of at least $n-t$ correct and active nodes, and arguments (1) to (3), the if-statement condition in line 47 holds within $O(1)$ CRWF/ACAF. □$_{Theorem\ 5.1}$

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 as well as via rigorous proofs. We hope that 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.

Acknowledgments: This work was partically supported by CyReV-2: Cyber Resilience for Vehicles - Cybersecurity for automotive systems in a changing environment (Vinnova 2019-03071).

REFERENCES

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *IEEE Symposium on Security and Privacy, SP'20*. 106–118.

[2] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. 2022. Balanced Byzantine Reliable Broadcast with Near-Optimal Communication and Improved Computation. In *PODC*. ACM, 399–417.

[3] Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. 2019. *Introduction to Distributed Self-Stabilizing Algorithms.* Morgan & Claypool Publishers.

[4] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. 2021. Leaderless Consensus. In *Distributed Computing Systems, ICDCS*. IEEE, 392–402.

[5] Anish Arora and Mohamed G. Gouda. 1993. Closure and Convergence: A Foundation of Fault-Tolerant Computing. *IEEE Trans. Software Eng.* 19, 11 (1993), 1015–1027.

[6] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous Secure Computations with Optimal Resilience. In *ACM PODC*. 183–192.

[7] Alexander Binun, Thierry Coupaye, Shlomi Dolev, Mohammed Kassi-Lahlou, Marc Lacoste, Alex Palesandro, Reuven Yagel, and Leonid Yankulin. 2016. Self-stabilizing Byzantine-Tolerant Distributed Replicated State Machine. In *Stabilization, Safety, and Security of Distributed Systems SSS'16*. 36–53.

[8] Alexander Binun, Shlomi Dolev, and Tal Hadad. 2019. Self-stabilizing Byzantine Consensus for Blockchain. In *CSCML*. 106–110.

[9] Peva Blanchard, Shlomi Dolev, Joffroy Beauquier, and Sylvie Delaët. 2014. Practically Self-stabilizing Paxos Replicated State-Machine. In *NETYS (LNCS, Vol. 8593)*. Springer, 99–121.

[10] Gabriel Bracha and Sam Toueg. 1983. Resilient Consensus Protocols. In *PODC*. ACM, 12–26.

[11] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461.

[12] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2006. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *Comput. J.* 49, 1 (2006), 82–96.

[13] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Veríssimo. 2011. Byzantine consensus in asynchronous message-passing systems: a survey. *Int. J. Crit. Comput. Based Syst.* 2, 2 (2011), 141–161.

[14] Sourav Das, Zhuolun Xiang, and Ling Ren. 2021. Asynchronous Data Dissemination and its Applications. In *CCS*. ACM, 2705–2721.

[15] Edsger W. Dijkstra. 1974. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM* 17, 11 (1974), 643–644.

[16] Shlomi Dolev. 2000. *Self-Stabilization.* MIT Press.

[17] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. 2018. Self-stabilizing Byzantine Tolerant Replicated State Machine Based on Failure Detectors. In *CSCML*. 84–100.

[18] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. 2010. When consensus meets self-stabilization. *J. Comput. Syst. Sci.* 76, 8 (2010), 884–900.

[19] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. 2013. Self-stabilizing Byzantine Resilient Topology Discovery and Message Delivery. In *SSS (LNCS, Vol. 8255)*. Springer, 351–353.

[20] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. 2023. Self-Stabilizing and Private Distributed Shared Atomic Memory in Seldomly Fair Message Passing Networks. *Algorithmica* 85, 1 (2023), 216–276.

[21] Romaric Duvignau, Michel Raynal, and Elad M. Schiller. 2021. Self-stabilizing Byzantine- and Intrusion-tolerant Consensus. *CoRR* abs/2110.08592 (2021).

[22] Romaric Duvignau, Michel Raynal, and Elad Michael Schiller. 2022. Self-stabilizing Byzantine-tolerant Broadcast. *Symposium on Stabilization, Safety, and Security of Distributed Systems SSS. Also in CoRR* abs/2201.12880 (2022).

[23] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323.

[24] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382.

[25] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. 2019. Self-stabilizing Snapshot Objects for Asynchronous Failure-Prone Networked Systems. In *Networked Systems NETYS*. 113–130.

[26] Chryssis Georgiou, Ioannis Marcoullis, Michel Raynal, and Elad Michael Schiller. 2021. Loosely-self-stabilizing Byzantine-Tolerant Binary Consensus for Signature-Free Message-Passing Systems. In *NETYS (LNCS, Vol. 12754)*. Springer, 36–53.

[27] Chryssis Georgiou, Michel Raynal, and Elad Michael Schiller. 2023. Self-stabilizing Byzantine-Tolerant Recycling. In *SSS (LNCS, Vol. 14310)*. Springer, 518–535.

[28] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401.

[29] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. [n. d.]. Self-Stabilizing Indulgent Zero-degrading Binary Consensus. In *ICDCN*.

[30] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. 2021. Self-stabilizing Multivalued Consensus in Asynchronous Crash-prone Systems. In *EDCC*. IEEE, 111–118. Also in CoRR abs/2104.03129.

[31] Alexandre Maurer and Sébastien Tixeuil. 2014. Self-Stabilizing Byzantine Broadcast. In *Reliable Distributed Systems, SRDS*. 152–160.

[32] Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. 2014. Signature-free asynchronous Byzantine consensus with t<n/3 and O(n^2) messages. In *ACM Principles of Distributed Computing, PODC '14*. 2–9.

[33] Achour Mostéfaoui and Michel Raynal. 2010. Signature-Free Broadcast-Based Intrusion Tolerance: Never Decide a Byzantine Value. In *OPODIS (LNCS, Vol. 6490)*. Springer, 143–158. An extended journal version appears in IEEE Trans. Parallel Distributed Syst., v. 27, n. 4, pp. 1085–1098, 2016.

[34] Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. 2005. Solving Vector Consensus with a Wormhole. *IEEE Trans. Parallel Distributed Syst.* 16, 12 (2005), 1120–1131.

[35] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (1980), 228–234.

[36] Kenneth J. Perry. 1984. Randomized Byzantine Agreement. In *Fourth Symposium on Reliability in Distributed Software and Database Systems, SRDS*. 107–118.

[37] Michel Raynal. 2018. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach.* Springer.

[38] Sam Toueg. 1984. Randomized Byzantine Agreements. In *Proceedings of the Third Annual ACM Principles of Distributed Computing*. 163–178.