

Self-stabilizing Byzantine-tolerant Recycling ^{*}

Chryssis Georgiou [†] Michel Raynal [‡] Elad M. Schiller [§]

July 28, 2023

Abstract

Numerous distributed applications, such as cloud computing and distributed ledgers, necessitate the system to invoke asynchronous consensus objects an unbounded number of times, where the completion of one consensus instance is followed by the invocation of another. With only a constant number of objects available, object reuse becomes vital.

We investigate the challenge of object recycling in the presence of *Byzantine* processes, which can deviate from the algorithm code in any manner. Our solution must also be *self-stabilizing*, as it is a powerful notion of fault tolerance. Self-stabilizing systems can recover automatically after the occurrence of *arbitrary transient-faults*, in addition to tolerating communication and (Byzantine or crash) process failures, provided the algorithm code remains intact.

We provide a recycling mechanism for asynchronous objects that enables their reuse once their task has ended, and all non-faulty processes have retrieved the decided values. This mechanism relies on synchrony assumptions and builds on a new self-stabilizing Byzantine-tolerant synchronous multivalued consensus algorithm, along with a novel composition of existing techniques.

1 Introduction

We study the problem of recycling asynchronous consensus objects. We propose a more robust solution than the state-of-the-art solution to achieve this goal.

Fault model. We study solutions for message-passing systems. We model a broad set of failures that can occur to computers and networks. Specifically, our model includes up to t process failures, *i.e.*, crashed or Byzantine [37]. In detail, the adversary completely controls any Byzantine node, *e.g.*, the adversary can send a fake message that the node never sent, modify the payload of its messages, delay the delivery of its messages, or omit any subset

^{*}This document is a complementary technical report to the extended abstract version of this work [33].

[†]Computer Science, Univ. Cyprus, Cyprus. Email: chryssis@cs.ucy.ac.cy

[‡]IRISA, Univ. Rennes 1, France. Email: michel.raynal@irisa.fr

[§]Computer Science and Engineering, Chalmers Univ. Tech., Sweden. Email: elad@chalmers.se

of them. The adversary can challenge the algorithm by creating failure patterns in which a fault occurrence appears differently to different system components. The adversary is empowered with the unlimited ability to coordinate the most severe failure patterns. We assume a known maximum number, t , of Byzantine processes. For solvability's sake, we also restrict the adversary from letting a Byzantine process impersonate a non-faulty one, *i.e.*, as in [51], we assume private channels between any pair of nodes.

Self-stabilization. In addition to the failures captured by our model, we also 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, packet payload, and indices, *e.g.*, sequence numbers, which are responsible for the correct operation of the studied system, as well as operational assumptions, such as that at least a distinguished majority of processes never fail. Since the occurrence of these failures can be arbitrarily combined, we assume that these transient-faults can alter the system state in unpredictable ways. In particular, when modeling the system, Dijkstra [14] assumes that these violations bring the system to an arbitrary state from which a *self-stabilizing system* should recover. Dijkstra requires recovery after the last occurrence of a transient-fault and once the system has recovered, it must never violate the task specifications. *I.e.*, there could be any finite number of transient faults before the last one occurs, which may leave the system in an arbitrary state. Moreover, recovery from an arbitrary system state is demonstrated once all transient faults cease to happen, see [4, 15] for details.

Memory constraints. In the absence of transient faults, one can safely assume that the algorithm variables, such as a counter for the message sequence number, are unbounded. This assumption can be made valid for any practical setting by letting each counter use enough bits, say, 64, because counting (using sequential steps) from zero to the maximum value of the counter will take longer than the time during which the system is required to remain operational. Specifically, if each message transmission requires at least one nanosecond, it would take at least 584 years until the maximum value can be reached. However, in the context of self-stabilization, a single transient fault can set the counter value into one that is close to the maximum value. Thus, any self-stabilizing solution must cope with this challenge and use only bounded memory and communication.

Self-stabilization via algorithmic transformation. This work is dedicated to designing a generic transformer, which takes an algorithm as input and systematically redesigns it into its self-stabilizing variation as output. Existing transformers differ in the range of input algorithms and fault models they can transform, see Dolev [15, 2.8], Katz and Perry [35], Afek *et al.* [2], and Awerbuch *et al.* [5].

Dolev, Petig, and Schiller [21] (DPS in short) proposed a transformer of crash-tolerant algorithms for asynchronous message-passing systems into ones that also recover from transient faults via self-stabilization. Georgiou *et al.* [29] implemented DPS. Lundström *et al.* show DPS's applicability to various communication abstractions, such as atomic snapshot [30], consensus [41, 42], reliable broadcast [40], and state-machine replication [44]. DPS mandates that (DPS.i) the input algorithm guarantees, after the last transient fault occurrence,

the completion of each invocation of the communication abstraction, *i.e.*, it should eventually terminate regardless of the starting state. This condition facilitates the eventual release of resources used by each invocation. Additionally, (DPS.ii) it associates a sequence number with each invocation to differentiate the resources utilized by different invocations. This enables the recycling of resources associated with obsolete invocations through a sliding window technique, along with a global restart once the maximum sequence number is reached.

Recently, DPS was utilized by Duvignau *et al.* [25] for converting Byzantine fault-tolerant (BFT) reliable broadcast proposed by Bracha and Toueg [13] into a Self-Stabilizing BFT (SSBFT) variation. This solution recycles reliable broadcast objects using synchrony assumptions. It also relies on the fact that the process may allocate independent local memory and sequence numbers per sender. However, consensus objects often use shared sequence numbers, and thus, parts of their local memories are codependent. Therefore, we use another approach.

Problem description. This work studies an important building block that is needed for the SSBFT implementation of asynchronous consensus objects. With only a bounded number of consensus objects available, it becomes essential to reuse them robustly. We examine the case in which the repeated invocation of consensus needs to reuse the same memory space and the $(k + 1)$ -th invocation can only start after the completion of the k -th instance. In an asynchronous system that uses only a bounded number of objects, ensuring the termination of the k -th instance before invoking the $(k + 1)$ -th might be crucial, *e.g.*, for total order broadcasting, as in some blockchains. Thus, we require SSBFT consensus objects to eventually terminate regardless of their starting state, as in (DPS.i).

We focus on addressing the challenge of recycling asynchronous consensus objects after they have completed their task and delivered their decision to all non-faulty processes. This task becomes complex due to the presence of asynchrony and Byzantine failures. Utilizing the joint sequence numbers of (DPS.ii) for recycling consensus objects is not straightforward, because it requires ensuring that all non-faulty processes have delivered the decided value (for the k -th consensus invocation) as well as agreeing that such collective delivery occurred before incrementing the sequence number counter (that is going to be used by the $(k + 1)$ -th invocation). To overcome this chicken-and-egg problem, we relax the problem requirements by allowing the recycling mechanism to depend 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 available memory. Thus, our approach facilitates asynchronous networking in communication-intensive components, *i.e.*, the consensus objects, while synchronous recycling actions are performed according to a load parameter, δ .

Our solution in a nutshell. Our solution aims to emulate (DPS.ii) by incorporating synchrony assumptions specifically for the recycling service, while keeping the consensus object asynchronous to handle intensive message exchange.

To begin, we maintain an index that points to the most recently invoked object in a constant-size array. In order to ensure that all non-faulty processes agree on the value of the index, we utilize a novel technique called *simultaneous increment-or-get indexes* (SGI-

index). When recycling an object, we increment the index, but this increment is performed only after an agreement among the non-faulty processes that the relevant object has made its decision and delivered it to all non-faulty processes. Thus, we use a new SSBFT multivalued consensus before each increment, ensuring that consensus is reached before the increment. *I.e.*, all needed deliveries had occurred before the increment.

Additionally, our solution answers how an SSBFT asynchronous consensus object can provide an indication that at least one non-faulty process has made a decision and delivered. We utilize this indication as input to trigger the recycling action, effectively incorporating it into the SSBFT multivalued consensus.

Related work. *Object recycling.* Object recycling was studied mainly in the context of crash-tolerant (non-BFT) systems [49, 58]. There are a few (non-self-stabilizing) implementations of garbage collection in the presence of Byzantine processes, *e.g.*, [47].

Impossibilities. As mentioned, FLP [28] concluded that consensus is impossible to solve deterministically in asynchronous settings in the presence of even a single crash failure. In [27] it was shown that a lower bound of $t + 1$ communication steps are required to solve consensus deterministically in both synchronous and asynchronous environments. In the presence of Byzantine faults, the consensus problem is not solvable (without signatures) if a third or more of the processes are faulty [37]. Thus, optimally resilient signature-free Byzantine consensus algorithms, tolerate $t < n/3$ faulty processes. The task is also impossible if a process can impersonate some other process in its communication with the other entities [6]. As in [46], we assume the absence of spoofing attacks and similar means of impersonation.

Non-self-stabilizing BFT consensus. Rabin [50] offers a solution to BFT *consensus* (cf. Section 2 for definitions). It assumes the availability of *random common coins* (RCCs), allowing for a polynomial number of communication steps and optimal resilience, *i.e.*, $t < n/3$, where n is the number of participating processes. Mostéfaoui, Moumen, and Raynal [46], or MMR in short, is a signature-free BFT binary consensus solution. MMR is optimal in resilience, uses $O(n^2)$ messages per consensus invocation, and completes within $O(1)$ expected time.

Non-self-stabilizing synchronous BFT multivalued consensus. The proposed recycling mechanism uses an SSBFT multivalued consensus, which is based on a non-self-stabilizing BFT multivalued consensus. Kowalski and Mostéfaoui [36] proposed the first multivalued optimal resilience, polynomial communication cost, and optimal $t + 1$ rounds, but without early stopping. Abraham and Dolev [1] advanced the state of the art by offering also optimal early stopping. Unlike the above BFT multivalued consensus solutions, our SSBFT multivalued solution adds self-stabilization.

SSBFT solutions. In the broader context of SSBFT solutions for message-passing systems, we find topology discovery [18], storage [11, 10, 9], clock synchronization [22, 38], approximate agreement [12], asynchronous unison [24], communication in dynamic networks [45], and SSBFT state-machine replication [8, 17] to name a few.

SSBFT consensus. To the best of our knowledge, the only SSBFT RCCs construction is the one by Ben-Or, Dolev, and Hoch [7], in short BDH, for synchronous systems with private channels. BDH uses its SSBFT RCCs construction as a building block for devising

an SSBFT clock synchronization solution. Our work borrows several mechanisms from BDH, such as SSBFT RCCs and SSBFT clock synchronization. Recently, Georgiou, Marcoullis, Raynal, and Schiller [32], or GMRS in short, presented an SSBFT variation on MMR, which offers a BFT binary consensus solution. GMRS preserves MMR’s optimality properties, and thus, we base our example consensus object on GMRS in this work.

GMRS follows the design criteria of loosely self-stabilizing systems [53], ensuring task completion but with rare safety violation events. In the context of the studied problem, the former guarantee renders the latter one irrelevant. We point out that related work to loosely self-stabilizing systems include randomized congestion control [26] and leader election [54, 55, 56, 34].

Self-stabilizing non-Byzantine fault-tolerant solutions. Lundström, Raynal, and Schiller [41] presented the first self-stabilizing solution for the problem of binary consensus for message-passing systems where nodes may fail by crashing. They provided a line of self-stabilizing solutions [43, 39, 40, 31, 29]. This line follows the approach proposed by Dolev, Petig, and Schiller [20, 19] for self-stabilization in the presence of seldom fairness. Namely, in the absence of transient-faults, these self-stabilizing solutions are wait-free and no assumptions are made regarding the system’s synchrony or fairness of its scheduler. However, the recovery from transient faults does require fair execution, *e.g.*, to perform a global reset, see [29], but only during the recovery period. The studied problem is more challenging than the above due to the presence of Byzantine processes and transient faults. Thus, we consider synchrony assumptions.

Our contribution. We propose an important building block for reliable distributed systems: a new SSBFT mechanism for recycling SSBFT consensus objects. The proposed mechanism stabilizes within expected $\mathcal{O}(\kappa)$ synchronous rounds, where $\kappa \in \mathcal{O}(t)$ is a pre-defined constant (that depends on synchrony assumptions) and t is an upper bound on the number of Byzantine processes. We also present, to the best of our knowledge, the first SSBFT synchronous multivalued consensus solution. The novel composition of (i) SSBFT recycling and (ii) SSBFT recyclable objects has a long line of applications, such as replication and blockchain. Thus, our transformation advances the state of the art by facilitating solutions that are more fault-tolerant than the existing implementations, which cannot recover from transient faults.

For convenience, a Glossary is provided in Table 1.

2 Basic Result: Recyclable SSBFT Consensus Objects

In this section, we define a recyclable variation on the consensus problem, which facilitates the use of an unbounded number of consensus instances via the reuse of a constant number of objects (as presented in Section 4). Then, we sketch Algorithm 1, which presents a recyclable variation on an SSBFT asynchronous consensus algorithm (such as GMRS). Towards the end of this section, Theorem 2.1 demonstrates that Algorithm 1 constructs recyclable SSBFT consensus objects.

Byzantine-tolerant Consensus (BC). This problem requires agreeing on a single value

Notation	Meaning
BC	Byzantine-tolerant Consensus
BDH	Ben-Or, Dolev, and Hoch [7]
BFT	non-self-stabilizing Byzantine fault-tolerant solutions
COR	Consensus Object Recycling (Definition 4.1)
DPS	Dolev, Petig, and Schiller [21]
MMR	Mostéfaoui, Moumen, and Raynal [46]
RCCs	random common coins
SSBFT	self-stabilizing Byzantine fault-tolerant
κ -SGC	κ -state global clock

Table 1: Glossary

from a given set V , which every (non-faulty) node inputs via `propose()`. It requires *BC-validity*, *i.e.*, if all non-faulty nodes propose the same value $v \in V$, only v can be decided, *BC-agreement*, *i.e.*, no two non-faulty nodes can decide different values, and *BC-completion*, *i.e.*, all non-faulty nodes decide a value. When the set, V , from which the proposed values are taken is $\{0, 1\}$, the problem is called *binary consensus*. Otherwise, it is referred to as *multivalued consensus*.

Recyclable Consensus Objects. We study systems that implement consensus objects using storage of constant size allocated at program compilation time. Since these objects can be instantiated an unbounded number of times, it becomes necessary to reuse the storage once consensus is reached and each non-faulty node has received the object result via `result()`. To facilitate this, we assume that the object has two meta-statuses: *used* and *unused*. The *unused* status represents both objects that were never used and those that are no longer in current use, indicating they are available (for reuse). Our definition of recyclable objects assumes that the objects implement an interface function called `wasDelivered()` that must return 1 anytime after the result delivery. Recycling is triggered by the recycling mechanism (Section 4), which invokes `recycle()` at each non-faulty node, thereby setting the meta-status of the corresponding consensus object to *unused*. We specify the task of recyclable object construction as one that requires eventual agreement on the value of `wasDelivered()`. In detail, if a non-faulty node p_i reports delivery (*i.e.*, `wasDeliveredi() = 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()`.

Algorithm outline. Algorithm 1’s `boxed` code lines highlight the code lines relevant to recyclability. The set of nodes is denoted by \mathcal{P} . We avoid the restatement of the algorithm, and to focus on the parts that matter in this work, the other parts are given in words (cf. GMRS [32] for full details). The code uses the symbol \bullet to denote any sequence of values. We assume that the object allows the proposal of v via `propose(v)` (line 9). As in `result()`

Algorithm 1: A recyclable variation on GMRS; code for node p_i .

```

1 local variables:
   /* the algorithm's local state is defined here. */
2  $delivered[\mathcal{P}] := [\text{False}, \dots, \text{False}]$  delivery indications;  $delivered[i]$  stores the local
3 indication and  $delivered[j]$  stores the last received indication from  $p_j \in \mathcal{P}$ ;
4 constants:
5  $initState := (\bullet, [\text{False}, \dots, \text{False}]);$ 
6 interfaces:
7 recycle() do (local state,  $delivered$ )  $\leftarrow initState$ ; /* also initialize all
   attached communication channels [15, Ch. 3.1] */
8 wasDelivered() do {if  $\exists S \subseteq \mathcal{P} : n-t \leq |S| : \forall p_k \in S : delivered[k] = \text{True}$  then
   return 1 else return 0;}
9 operations: propose( $v$ ) do {implement the algorithm logic};
10 result() do begin
11   if there is a decided value then  $\{delivered[i] \leftarrow \text{True}; \text{return } v\}$ ;
12   else if an error occurred then  $\{delivered[i] \leftarrow \text{True}; \text{return } \Psi\}$ ;
13   else return  $\perp$ ;
14 do forever begin
15   if result() =  $\perp$  then  $delivered[i] \leftarrow \perp$ ; /* consistency test */
   /* implementation of the algorithm's logic */
16   foreach  $p_j \in \mathcal{P}$  do send  $\text{EST}(\bullet, delivered[i])$  to  $p_j$ ;
17 upon  $\text{EST}(\bullet, deliveredJ)$  arrival from  $p_j$  begin
18    $delivered[j] \leftarrow deliveredJ$ ;
   /* merge arriving information with the local one */

```

(line 10), once the consensus algorithm decides, one of the decided value is returned (line 11). Since the algorithm tolerates transient faults, the object may need to indicate an internal error via the return of the (transient) error symbol, Ψ (line 12). In all other cases, *i.e.*, as long as no value was decided, the \perp -value is returned (line 13). GMRS uses a do-forever loop that broadcasts the protocol messages (line 16). Any node that receives this protocol message, merges the arriving information with the one stored by the local state (line 18).

Recyclable variation. Algorithm 1 uses the array $delivered[\mathcal{P}]$ (initialized to the vector $[\text{False}, \dots, \text{False}]$) for delivery indications, where $delivered_i[i] : p_i \in \mathcal{P}$ stores the local indication and $delivered_i[j] : p_i, p_j \in \mathcal{P}$ stores the indication that was last received from

p_j . This indication is set to **True** whenever **result()** returns a non- \perp value (lines 11 to 12). Algorithm 1 updates $delivered[j]$ according to the arriving values from p_j (lines 16 and 18). The interface function **wasDelivered()** (line 8) returns 1 if at least $n - t$ entries in $delivered[]$ hold **True**. The interface function **recycle()** (line 7) allows the node to restart its local state w.r.t. Algorithm 1.

Theorem 2.1 shows that Algorithm 1 satisfies the requirements for Recyclable Consensus Objects, which we defined above. Following the definition of the BC problem and GMRS, the theorem assumes that every (non-faulty) node invokes **result()** infinitely often.

Theorem 2.1 *Suppose that every (non-faulty) node invokes **result()** infinitely often. Algorithm 1 offers a recyclable asynchronous consensus object.*

Proof of Theorem 2.1 Let R be an unbounded execution of Algorithm 1 in which no (non-faulty) node invokes **recycle()**. Suppose $\exists i \in \text{Correct} : \text{wasDelivered}_i() = 1$ in any system state in R . We show that the system reaches a state $c \in R$ in which $\forall j \in \text{Correct} : \text{wasDelivered}_j() = 1$.

By line 15 and the assumption that $i \in \text{Correct} : \text{wasDelivered}_i() = 1$ holds throughout R , $\text{result}_i() \neq \perp$ holds in every system state R , i.e., p_i 's state encodes completion. By BC-completion and the assumption that Algorithm 1 is an SSBFT implementation of consensus, $\forall j \in \text{Correct} : \text{result}_j() \neq \perp$ eventually. By lines 11 to 12 and the theorem assumption that every (non-faulty) node invokes **result()** infinitely often, $\forall j \in \text{Correct} : \text{wasDelivered}_j() = 1$.

□_{Theorem 2.1}

3 System Settings for the Recycling Mechanism

This model considers a synchronous message-passing system. The system consists of a set, \mathcal{P} , of n nodes (sometimes called *processes* or *processors*) with unique identifiers. At most $t < n/3$, out of the n nodes, are faulty. Any pair of nodes $p_i, p_j \in \mathcal{P}$ has access to a bidirectional reliable communication channel, $channel_{j,i}$. In the *interleaving model* [15], the node's program is a sequence of (*atomic*) *steps*. Each step starts with (i) the communication operation for receiving messages that is followed by (ii) an internal computation, and (iii) finishes with a single *send* operation. 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 $c = (s_1, s_2, \dots, s_n)$. Our model also assumes the availability of a κ -state global clock, reliable communications, and random common coins (RCCs).

A κ -state global clock. We assume that the algorithm takes steps according to a common global pulse (beat) that triggers a simultaneous step of every node in the system. Specifically, we denote synchronous executions by $R = c[0], c[1], \dots$, where $c[x]$ is the system state that immediately precedes the x -th global pulse. And, $a_i[x]$ is the step that node p_i takes between $c[x]$ and $c[x + 1]$ simultaneously with all other nodes. We also assume that each node has access to a κ -state global clock via the function $clock(\kappa)$, which returns an integer between 0 and $\kappa - 1$. Algorithm 3 of BDH [7] offers an SSBFT κ -state global clock that stabilizes within a constant time.

Reliable communication. Recall that we assume the availability of reliable communication. Also, any non-faulty node $p_i \in \mathcal{P}$ starts any step $a_i[x]$ with receiving all pending messages from all nodes. And, if p_i sends any message during $a_i[x]$, it does so only at the end of $a_i[x]$. We require (i) any message that a non-faulty node p_i sends during step $a_i[x]$ to another non-faulty node p_j is received at p_j at the start of step $a_j[x + 1]$, and (ii) any message that p_j received during step $a_j[x + 1]$, was sent at the end of $a_i[x]$.

Random common coins (RCCs). As already mentioned, BDH presented a synchronous SSBFT RCCs solution for message passing systems. Algorithm \mathcal{A} , which has the output of $rand_i \in \{0, 1\}$, is said to provide an RCC if \mathcal{A} satisfies the following:

- **RCC-completion:** \mathcal{A} provides an output within $\Delta_{\mathcal{A}} \in \mathbb{Z}^+$ synchronous rounds.
- **RCC-unpredictability:** Denote by $E_{x \in \{0,1\}}$ the event that for any non-faulty process, p_j , $rand_j = x$ holds with constant probability $p_x > 0$. Suppose either E_0 or E_1 occurs at the end of round $\Delta_{\mathcal{A}}$. We require that the adversary can predict the output of \mathcal{A} by the end of round $\Delta_{\mathcal{A}} - 1$ with a probability that is not greater than $1 - \min\{p_0, p_1\}$. Following [46], we assume that $p_0 = p_1 = 1/2$.

Our solution depends on the existence of a self-stabilizing RCC service, *e.g.*, BDH. BDH considers (*progress*) *enabling* instances of RCCs if there is $x \in \{0, 1\}$ such that for any non-faulty process p_i , we have $rand_i = x$. BDH correctness proof depends on the consecutive existence of two enabling RCCs instances.

Legal executions. The set of *legal executions* (LE) refers to all the executions in which the requirements of task T hold. In this work, T_{recycl} denotes the task of consensus object recycling (specified in Section 4), and LE_{recycl} denotes the set of executions in which the system fulfills T_{recycl} 's requirements.

Arbitrary node failures. As explained in Section 1, Byzantine faults model any fault in a node including crashes, arbitrary behavior, and malicious behavior [37]. For the sake of solvability [37, 48, 57], our fault model limits only the number of nodes that can be captured by the adversary. That is, the number, t , of Byzantine failure needs to be less than one-third of the number, n , of nodes. The set of non-faulty nodes is denoted by *Correct*.

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). Our model assumes that the last arbitrary transient fault occurs before the system execution starts [4, 15]. Also, it leaves the system to start in an arbitrary state.

Self-stabilization. An algorithm is *self-stabilizing* for the task of LE , when every (unbounded) execution R of the algorithm reaches within a finite period a suffix $R_{\text{legal}} \in LE$ that is legal. Namely, Dijkstra [14] requires $\forall R : \exists R' : R = R' \circ R_{\text{legal}} \wedge R_{\text{legal}} \in LE \wedge |R'| \in \mathbb{Z}^+$, 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 the *convergence* (or recovery) proof, and the part that shows that $R_{\text{legal}} \in LE$ is called the *closure* proof. We clarify that once the execution of a self-stabilizing system becomes legal, it stays legal due to the property

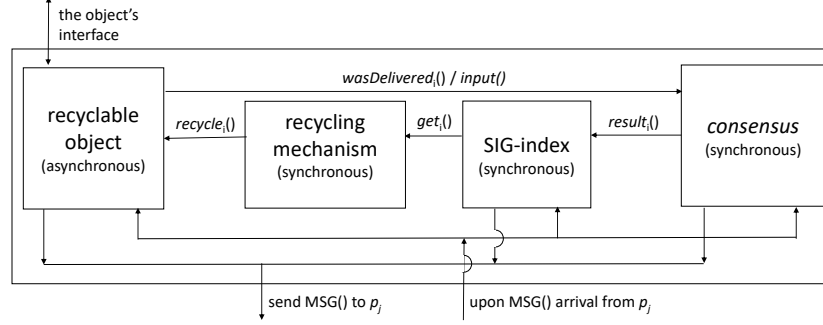


Figure 1: The solution uses recyclable objects (Algorithm 1), a recycling mechanism (Algorithm 2), multivalued consensus (Algorithm 3), and SIG-index (Algorithm 4). Algorithms 1 and 3 solve the BC problem (Section 2) for asynchronous, and resp., synchronous settings. Algorithms 2 and 4 solve the COR, resp., SGI-index problems (Section 4).

of closure. The main complexity measure of a self-stabilizing system is its stabilization time, which is the length of the recovery period, R' , which is counted by the number of its synchronous rounds.

4 SSBFT Recycling Mechanism

We present an SSBFT recycling mechanism for recyclable objects (Section 2). The mechanism is a service that recycles consensus objects via the invocation of `recycle()` by all (non-faulty) nodes. The coordinated invocation of `recycle()` can occur only after the consensus object has terminated and the non-faulty nodes have delivered the result, via `result()`, as indicated by `wasDelivered()`.

Consensus Object Recycling (COR). Definition 4.1 specifies the COR problem for a single object. COR-validity-1 is a safety property requiring that `recycle()` is invoked only if there was at least one reported delivery by a non-faulty node. COR-validity-2 is a liveness property requiring that eventually `recycle()` is invoked. COR-agreement is a safety property requiring that all non-faulty nodes simultaneously set the object's status to unused. This allows any node p_i to reuse the object immediately after the return from `recyclei()`.

Definition 4.1 (Consensus Object Recycling) *The following properties are required:*

- **COR-validity-1:** *If a non-faulty node, p_j , invokes `recyclej()`, then at least one non-faulty node, p_i , reported delivery. I.e., no non-faulty node, p_j , invokes `recyclej()` if only faulty nodes report delivery.*
- **COR-validity-2:** *If all non-faulty nodes report delivery, then at least one non-faulty node, p_j , eventually invokes `recyclej()`.*
- **COR-agreement:** *If a non-faulty node invokes `recycle()`, then all non-faulty nodes, p_i , invoke `recyclei()` simultaneously.*

Algorithm 2: SSBFT synchronous recycling with a predefined log size; code for node p_i

```

19 constants:  $indexNum$  number of indices of recyclable objects;
20  $logSize \in \{0, \dots, indexNum - 2\}$  user-defined bound on the object log size;
21 variables:  $obj[indexNum]$  : array of recyclable objects, e.g., GMRS. Note that
    during legal execution only at most  $(logSize + 1)$  objects are stored at any given
    point of time;
22  $ssbftIndex$  : an SSBFT index of the current object in use (Algorithm 4);
23 upon pulse /* signal from global pulse system */ begin
24   foreach  $x \notin \{y \bmod indexNum : y \in \{z - logSize, \dots, z\}\}$  where
     $z = indexNum + ssbftIndex.getIndex()$  do
25      $obj[x].recycle()$ 

```

Multiple objects. We also specify that the recycling mechanism makes sure that, at any time, there are at most a constant number, $logSize$, of active objects, *i.e.*, objects that have not completed their tasks. Once an object completes its task, the recycling mechanism can allocate a new object by moving to the next array entry as long as some constraints are satisfied. Specifically, the proposed solution is based on a synchrony assumption that guarantees that every (correct) node retrieves (at least once) the result of a completed object, x , within $logSize$ synchronous rounds since the first time in which at least $t + 1$ (correct) nodes have retrieved the result of x , and thus, x can be recycled.

Solution overview. The SSBFT recycling solution is a composition of several algorithms, see Figure 1. Our recycling mechanism is presented in Algorithm 2. It allows every (correct) node to retrieve at least once the result of any object that is stored in a constant-size array and yet over time that array can store an unbounded number of object instances. The proposed service mechanism (Algorithm 2) ensures that every instance of the recyclable object, which is implemented by Algorithm 1, is guaranteed that every (correct) node calls `result()` (line 8) at least once before all (correct) nodes simultaneously invoke `recycle()` (line 7). This aligns with the solution architecture (Figure 1).

We consider the case in which the entity that retrieves the result of object obj might be external (and perhaps, asynchronous) to the proposed solution. The proposed solution does not decide to recycle obj before there is sufficient evidence that, within $logSize$ synchronous cycles, the system is going to reach a state in which obj can be properly recycled. Specifically, Assumption 1 considers an event that can be locally learned about when `wasDelivered()` returns ‘1’ (line 8).

Assumption 1 (A bounded time result retrieval) *Let us consider the system state, $c[r]$, in which the result of object obj was retrieved by at least $t + 1$ (correct) nodes. We assume, within $logSize$ synchronous cycles from $c[r]$, the system reaches a state, $c[r + logSize]$, in which all $n - t$ (correct) nodes have retrieved the result of obj at least once.*

Algorithm 2’s recycling guarantees are facilitated by an SSBFT multivalued consensus object (Algorithm 3). It helps to decide on a single piece of evidence from all collected ones (regarding recyclability) and Algorithm 4 uses the agreed evidence for updating the value of the index that points to the current entry in the object array. We later add details on Algorithm 2 before proving its correctness (Theorem 4.7).

Evidence collection using an SSBFT (multivalued) consensus (Algorithm 3). The SSBFT multivalued consensus protocol returns within $t + 1$ synchronous rounds an agreed non- \perp value as long as at least $t + 1$ nodes proposed that value, *i.e.*, at least one (correct) node proposed that value. As mentioned, `wasDelivered()` (line 8) provides the input to this consensus protocol. Thus, whenever ‘1’ is decided, at least one (correct) node gets an indication from at least $n - t$ nodes that they have retrieved the results of the current object. This implies that by at least $t + 1$ (correct) nodes have retrieved the results, and, by Assumption 1, all $n - t$ (correct) nodes will retrieve the object result within a known number of synchronous rounds. Then, the object could be recycled. We later add details on Algorithm 3 before proving its correctness (Theorem 4.1).

SSBFT simultaneous increment-or-get index (SIG-index). Algorithm 4 allows the proposed solution to keep track of the current object index that is currently used as well as facilitate synchronous increments to the index value. We call this task *simultaneous increment-or-get index* (SIG-index). During legal executions of Algorithm 4, the (correct) nodes assert their agreement on the index value and update the index according to the result of the agreement on `wasDelivered()`’s value. We later add details on Algorithm 4 before proving its correctness (Theorem 4.2).

Scheduling strategy. As mentioned, our SSBFT multivalued consensus requires $t + 1$ synchronous rounds to complete and provide input to Algorithm 4 and $\kappa - (t + 1)$ synchronous rounds after that, any (correct) node can recycle the current object (according to the multivalued consensus result), where $\kappa = \max\{t + 1, \logSize\}$. Thus, Algorithm 4 has to defer its index updates until that time. Figure 2 depicts this scheduling strategy, which considers the schedule cycle of κ . That is, the SIG-index and multivalued consensus starting points are 0 and $\kappa - 4$, respectively. Note that Algorithm 2 does not require scheduling since it accesses the index only via Algorithm 4’s interface of SIG-index, see Figure 1.

Communication piggybacking and multiplexing. We use a piggybacking technique to facilitate the spread of the result (decision) values of the recyclable objects. As Figure 1 illustrates, all communications are piggybacked. Specifically, we consider a meta-message `MSG()` that has a field for each message sent by all algorithms in Figure 1. That is, when any of these algorithms is active, its respective field in `MSG()` includes a non- \perp value. With respect to GMRS’s field, `MSG()` includes the most recent message that GMRS has sent (or currently wishes to send). This piggybacking technique allows the multiplexing of timed and reliable communication (assumed for the recycling mechanism) and fair communication (assumed for the recyclable object).

SSBFT recycling (Algorithm 2). As mentioned, Algorithm 2 has an array, `obj[]` (line 21), of `indexNum` recyclable objects (line 19). The array size needs to be larger than `logSize` (line 20 and Assumption 1). Algorithm 2’s variable set also includes `ssbftIndex`,

Algorithm 3: SSBFT synchronous multivalued consensus; code for node p_i

```

26 variables: currentResult stores the most recent result of co;
27 co a (non-self-stabilizing) BFT (multivalued) consensus object;
28 interface required:
29 input() : source of (the proposed values) of the given consensus protocol;
30 interface provided:
31 result(): do return(currentResult) // decided value of the most recent co's
    invocation;
32 message structure:  $\langle appMsg \rangle$ , where appMsg is the application message, i.e., a
    message sent by the given consensus protocol;
33 upon pulse /* signal from global pulse system */ begin
34   let M be a message that holds at  $M[j]$  the arriving  $\langle appMsg_j \rangle$  messages from  $p_j$ 
    for the current synchronous round and  $M' = [\perp, \dots, \perp]$ ;
35   if  $clock(\kappa) = 0$  then
36     currentResult  $\leftarrow co.result()$ ;
37     co.restart();
38      $M' \leftarrow co.propose(input())$  // for recycling  $input() \equiv wasDelivered()$ 
39   else if  $clock(cycleSize) \in \{1, \dots, t\}$  then  $M' \leftarrow co.process(M)$ ;
40   foreach  $p_j \in \mathcal{P}$  do send  $\langle M'[j] \rangle$  to  $p_j$ ;

```

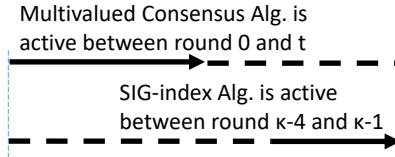


Figure 2: The solution schedule uses a cycle of $\kappa = \max\{t + 1, logSize\}$ synchronous rounds.

which is an integer that holds the entry number of the latest object in use. Algorithm 2 accesses the agreed current index via *ssbftIndex.getIndex()*. This lets the code to nullify any entry in *obj[]* that is not *ssbftIndex.getIndex()* or at most *logSize* older than *ssbftIndex.getIndex()*. Theorem 4.7 shows Algorithm 2's correctness.

SSBFT synchronous multivalued consensus. Algorithm 3 assumes access to a deterministic (non-self-stabilizing) BFT (multivalued) consensus object, *co*, such as the ones proposed by Kowalski and Mostéfaoui [36] or Abraham and Dolev [1], for which completion is guaranteed to occur within $t + 1$ synchronous rounds. We list our assumptions regarding the interface to *co* in Definition 4.2.

Required consensus object interface. Our solution uses the technique of recomputation of *co*'s floating output [15, Ch. 2.8], where *co* is specified in Definition 4.2.

Algorithm 4: SSBFT synchronous SIG-index; p_i 's code

```
41 constants:  $I$  : bound on the number of states an index may have;
42 variables:  $index \in \{0, \dots, I - 1\}$  : a local copy of the global logical object index;
43  $ssbftCO$  : an SSBFT multivalued consensus object (Algorithm 3) that is used for
    agreeing on the recycling state, i.e., 1 when there is a need to recycle (otherwise 0);
44 interfaces provided:  $getIndex()$  do return  $index$ ;
45 message structure:  $\langle index \rangle$ : the logical object index;
46 upon pulse /* signal from global pulse system */ begin
47   let  $M$  be the arriving  $\langle index_j \rangle$  messages from  $p_j$ ;
48   switch  $clock(\kappa)$  /* consider  $clock()$  at the pulse beginning */ do
49     case  $\kappa - 4$  do broadcast  $\langle index = getIndex() \rangle$ ;
50     case  $\kappa - 3$  do
51       let  $propose := \perp$ ;
52       if  $\exists v \neq \perp : |\{ \langle v \rangle \in M \}| \geq n - t$  then  $propose \leftarrow v$ ;
53       broadcast  $\langle propose \rangle$ ;
54     case  $\kappa - 2$  do
55       let  $bit := 0$ ;  $save \leftarrow \perp$ ;
56       if  $\exists s \neq \perp : |\{ \langle s \rangle \in M \}| > n/2$  then  $save \leftarrow s$ ;
57       if  $|\{ \langle save \neq \perp \rangle \in M \}| \geq n - t$  then  $bit \leftarrow 1$ ;
58       if  $save = \perp$  then  $save \leftarrow 0$ ;
59       broadcast  $\langle bit \rangle$ ;
60     case  $\kappa - 1$  do
61       let  $inc := 0$ ;
62       if  $ssbftCO.result()$  then  $inc \leftarrow 1$ ;
63       if  $|\{ \langle 1 \rangle \in M \}| \geq n - t$  then  $index \leftarrow (save + inc) \bmod I$ ;
64       else if  $|\{ \langle 0 \rangle \in M \}| \geq n - t$  then  $index \leftarrow 0$ ;
65       else  $index \leftarrow rand(save + inc) \bmod I$ ;
```

Definition 4.2 (Synchronous BFT Consensus) *Let co be a BFT (non-self-stabilizing) synchronous multivalued consensus that implements the following.*

- $restart()$ sets co to its initial state.
- $propose(v)$ proposes the value v when invoking (or re-invoking) co . This operation is effective only after $restart()$ was invoked. The returned value is a message vector, $M[]$, that includes all the messages, $M[j]$, that co wishes to send to node p_j for the current synchrony round.
- $process(M)$ runs a single step of co . The input vector M includes the arriving messages for the current synchronous round, where $M[j]$ is p_j 's message. The returned value is a message vector that includes all the messages that co wishes to send for the current synchrony round.

round. This operation is guaranteed to work correctly only after all (correct) nodes have simultaneously taken a consecutive sequence of steps that include invocations of either (i) `process()`, or (ii) `restart()` immediately before proposing a non- \perp value via the invocation of `propose()`.

- **result()** returns a non- \perp results after the completion of *co*. The returned value is required to satisfy the consensus specifications only if all (correct) nodes have simultaneously taken a sequence of correct `process()` invocations.

Detailed description. Algorithm 3's set of variables includes *co* itself (line 27) and the current version of the result, *i.e.*, `currentResult` (line 26). This way, the SSBFT version of *co*'s result can be retrieved via a call to **result()** (line 31). Algorithm 3 proceeds in synchronous rounds. At the start of any round, node p_i stores all the arriving messages at the message vector M (line 34).

When the clock value is zero (line 35), it is time to start the re-computation of *co*'s result. Thus, Algorithm 3 first stores the current value of *co*'s result at `currentResulti` (line 36). Then, it restarts *co*'s local state and proposes a new value to *co* (lines 37 and 38). For the recycling solution presented in this paper, the proposed value is retrieved from `wasDelivered()` (line 8). For the case in which the clock value is not zero (line 39), Algorithm 3 simply lets *co* process the arriving messages of the current round. Both for the case in which the clock value is zero and the case it is not, Algorithm 3 broadcasts *co*'s messages for the current round (line 40).

Correctness proof. Theorem 4.1 shows that Algorithm 3 stabilizes within 2κ synchronous rounds.

Theorem 4.1 *Algorithm 3 is an SSBFT deterministic (multivalued) consensus solution that stabilizes within 2κ synchronous rounds.*

Proof of Theorem 4.1 Let R be an execution of Algorithm 3. Within κ synchronous rounds, the system reaches a state $c \in R$ in which $clock(\kappa) = 0$ holds. Immediately after c , every (correct) node, p_i , simultaneously restarts *co_i* and proposes the input (lines 37 and 38) before sending the needed messages (line 40). Then, for the $t < \kappa$ synchronous rounds that follow, all (correct) nodes simultaneously process the arriving messages and send their replies (line 39 and 40). Thus, within 2κ synchronous rounds from R 's start, the system reaches a state $c' \in R$ in which $clock(\kappa) = 0$ holds. Also, in the following synchronous round, all (correct) nodes store *co*'s results. These results are guaranteed to be correct due to Definition 4.2's assumptions. $\square_{\text{Theorem 4.1}}$

SSBFT simultaneous increment-or-get index. The task of *simultaneous increment-or-get index* (SGI-index) requires all (correct) nodes to maintain identical index values that all nodes can independently retrieve via `getIndex()`. The task assumes that all increments are performed according to the result of a consensus object, *ssbftCO*, such as Algorithm 3. Algorithm 4 presents an SGI-index solution that recovers from disagreement on the index value using RCCs. That is, whenever a (correct) node receives $n - t$ reports from other nodes that they have each observed $n - t$ identical index values, an agreement on the index value

is assumed and the index is incremented according to the most recent result of *ssbftCO*. Otherwise, a randomized strategy is taken for guaranteeing recovery from a disagreement on the index value. Our strategy is inspired by BDH [7]’s SSBFT clock synchronization algorithm.

Detailed description. Algorithm 4 is active during four clock phases, *i.e.*, $\kappa - 4$ to $\kappa - 1$. Each phase starts with storing all arriving messages (from the previous round) in the array, M (line 47). The first phase broadcasts the local index value (line 49). The second phase lets each node vote on the majority arriving index value, or \perp in case such value was not received (lines 51 to 53). The third phase resolves the case in which there is an arriving non- \perp value, *save*, that received sufficient support when voting during phase two (lines 55 to 58). Specifically, if $save \neq \perp$ exists, then $\langle bit = 1 \rangle$ is broadcast. Otherwise, $\langle bit = 0 \rangle$ is broadcast. On the fourth phase (lines 61 to 65), the (possibly new) index is set either to be the majority-supported index value of phase two plus *inc* (lines 61 to 63), where *inc* is the output of *ssbftCO*, or (if there was insufficient support) to a randomly chosen output of the RCC (lines 64 and 65).

Correctness proof. Theorem 4.2 bounds Algorithm 4’s stabilization time.

Theorem 4.2 *Let R be an execution of algorithms 3 and 4 that is legal w.r.t. Algorithm 3 (Theorem 4.1). Algorithm 4 is an SSBFT SGI-index implementation that stabilizes within expected $\mathcal{O}(1)$ synchronous rounds.*

Proof of Theorem 4.2 Corollaries 4.3 and 4.4 are needed for Lemmas 4.5 and 4.6. The pigeonhole principle implies Corollary 4.3.

Corollary 4.3 *Let $V_{x \in \{a,b\}}$ be two n -length vectors that differ in at most $f < n/3$ entries. For any $x \in \{a,b\}$, suppose V_x contains $n - t$ copies of v_x . Then $v_a = v_b$.*

Corollary 4.4 is implied by Corollary 4.3.

Corollary 4.4 *Let $c[r] \in R$ be a system state in which $clock(\kappa) = \kappa - 3$ and $X = \{x_i : i \in \text{Correct}\}$ be the set of values encoded in the messages $\langle x_i \rangle$ that any (correct) node, $p_i \in \mathcal{P}$, broadcasts in line 53 at the end of $a_i[r]$. The set X includes at most one non- \perp value.*

Lemma 4.5 implies that, within $\mathcal{O}(1)$ of expected rounds, all (correct) nodes have identical index values. Recall that $c[r] \in R$ is (progress) enabling if $\exists x \in \{0,1\} : \forall i \in \text{Correct} : rand_i = x$ holds at $c[r]$ (Section 3).

Lemma 4.5 (Convergence) *Let $r > \kappa$. Suppose $c[r] \in R$ is (progress) enabling system state (Section 3) for which $clock(\kappa) = \kappa - 1$ holds. With probability at least $\min\{p_0, p_1\}$, all (correct) nodes have the same index at $c[r + 1]$.*

Proof of Lemma 4.5 The proof is implied by claims 1 to 4.

Claim 1 Suppose (i) there is no value $x \in \{0, 1\}$ and (ii) there is no (correct) node $p_i \in \mathcal{P}$ that receives at the start of step $a_i[r]$ the message $\langle x \rangle$ from at least $n - t$ different nodes. For any (correct) node, $p_j \in \mathcal{P}$, it holds that step $a_j[r]$ assigns 0 to $index_j$ with probability p_0 .

Proof of Claim 1 The proof is implied directly from lines 62 to 65. $\square_{Claim\ 1}$

Claim 2 Suppose there is a (correct) node $p_i \in \mathcal{P}$ that receives at the start of step $a_i[r]$ the message $\langle 0 \rangle$ from at least $n - t$ different nodes. Also, suppose there is $x \in \{0, 1\}$ and a (correct) node $p_j \in \mathcal{P}$ that receives at the start of step $a_j[r]$ the message $\langle x \rangle$ from at least $n - t$ different nodes, where $i = j$ may or may not hold. The step $a_j[r]$ assigns 0 to $index_j$.

Proof of Claim 2 Line 64 implies the proof since $x = 0$ (Corollary 4.3). $\square_{Claim\ 2}$

Claim 3 Suppose there is a (correct) $p_i \in \mathcal{P}$ that receives at the start of step $a_i[r]$ the message $\langle 1 \rangle$ from at least $n - t$ different nodes. Let $p_j \in \mathcal{P}$ be a (correct) node. At $c[r]$, $ssbftCO_i.result() = ssbftCO_j.result()$ and $save_i = save_j$ hold.

Proof of Claim 3 At $c[r]$, $ssbftCO_i.result() = ssbftCO_j.result()$ holds (Algorithm 3's agreement property). We show that $save_i = save_j$ holds at $c[r]$. Since p_i has received $\langle 1 \rangle$ from at least $n - t$ different nodes at the start of $a_i[r]$, we know that there is a (correct) node, $p_k \in \mathcal{P}$, that has sent $\langle 1 \rangle$ at the end of $a[r - 1]$. By lines 55 to 57, node p_j receives at the start of $a_j[r - 1]$ the message $\langle x \rangle$ from at least $n - t$ different nodes, where $x = save_j \neq \perp$. By Corollary 4.4, any (correct) node broadcasts (line 53) either \perp or x at the end of step $a[r - 2]$. This means that at the start of $a[r - 1]$, (correct) nodes receive at most $f < n - 2f$ messages with values that are neither \perp nor $x \neq \perp$. Therefore, $save_i = save_j$ since, at the start of $a_i[r]$ and $a_j[r]$ both p_i , and resp., p_j receive from at least $n - t$ different nodes the messages $\langle x_i \rangle$, and resp., $\langle x_j \rangle$, where neither x_i nor x_j is \perp . $\square_{Claim\ 3}$

Claim 4 Suppose there is a (correct) node $p_i \in \mathcal{P}$ that receives at the start of step $a_i[r]$ the message $\langle 1 \rangle$ from at least $n - t$ different nodes. Suppose there is $x \in \{0, 1\}$ and a (correct) node $p_j \in \mathcal{P}$ that receives at the start of step $a_j[r]$ the message $\langle x \rangle$ from at least $n - t$ different nodes, where $i = j$ may or may not hold. With a probability of at least $\min\{p_0, p_1\}$, the steps $a_i[r]$ and $a_j[r]$ assign the same value to $index_j$, and resp., $index_j$.

Proof of Claim 4 By Corollary 4.3, $x = 1$. The step $a[r - 1]$ determines x 's value and $rand$ is chosen at the start of step $a[r]$. Due to $rand$'s unpredictability (Section 3), $rand$ and x are two independent values. Thus, with a probability of at least $\min\{p_0, p_1\}$, all (correct) nodes update $index$ in the same manner, i.e., to either 0 or $save + inc$ (Claim 3), where $save$ and inc are values determined by lines 55 to 58, and resp. 61 to 62. $\square_{Claim\ 4} \quad \square_{Lemma\ 4.5}$

Lemma 4.6 shows that all (correct) nodes forever agree on their indexes and simultaneously increment them by one (modulo I) only when $clock(\kappa) = \kappa - 1$ and $ssbftCO_i.result() = 1$. Lemma 4.6 uses the following notation. Let $R = c[0], c[1], \dots, c[r], \dots$ an unbounded synchronous execution of Algorithm 4, where $c[r]$ is the system state that immediately precedes the arrival of the r -th common pulse. Denote by $indices_r^{start}$ and $indices_r^{end}$ the sets of all $index_i : i \in Correct$ at $c[r]$, and resp., $c[r + 1]$, i.e., the beginning, and resp., the end of step $a[r]$. Note that, for all r and $x \in \{start, end\}$, we have $indices_r^x \subseteq \{0, 1, \dots, I - 1\}$.

Lemma 4.6 (Closure) *Let $c[r] \in R$, such that $\text{clock}(\kappa) = \kappa - 1$ at $c[r]$. Suppose $\text{indices}_r^{\text{end}} = \{v \neq \perp\}$. For every $c[r'] \in R : r' \in \{r + 1, r + \kappa\}$ it holds that $\text{indices}_{r'}^{\text{start}} = \{v + x \bmod I\}$ where $x = 1$ when $r' = r + \kappa$ and $\text{ssbftCO.result}() = 1$. Otherwise, $x = 0$, i.e., when $r' \in \{r + 1, \dots, r + \kappa - 1\}$ or $\text{ssbftCO.result}() \neq 1$.*

Proof of Lemma 4.6 For $r' = r + 1$ the lemma holds since, by definition, $\forall r'' : \text{indices}_{r''}^{\text{end}} = \text{indices}_{r''+1}^{\text{start}}$. Also, for any system state $c[r'] : r' \in \{r + 1, \dots, r + \kappa - 1\}$, no (correct) node, $p_i \in \mathcal{P}$, updates index_i during the step, $a_i[r']$, since $\text{clock}(\kappa) \neq \kappa - 1$ at $c[r'] : r' \in \{r + 1, \dots, r + \kappa - 1\}$ and thus lines 63 to 65 are not executed, which are the only lines that update index_i .

It remains to show that all (correct) nodes, $p_i \in \mathcal{P}$, update index_i in the same way during the steps $a_i[r'] : r' = r + \kappa$ that immediately follow $c[r']$. This is due to the agreement property of Algorithm 3, the arguments above about $c[r'] : r' \in \{r + 1, r + \kappa - 1\}$ as well as Claim 5.

Claim 5 $\text{indices}_{r+\kappa}^{\text{start}} = \{v\} : v \neq \perp$.

Proof of Claim 5 By the schedule (Figure 2) and its cycle length, κ , we know that Algorithm 4 is not active between $c[r + 1]$ and $c[r + \kappa - 3]$, but it is active during steps $a[r + \kappa - 3]$, $a[r + \kappa - 2]$, $a[r + \kappa - 1]$, and $a[r + \kappa]$. During steps $a[r + \kappa - 3]$, all (correct) nodes broadcast $\langle v \rangle$ (line 49). Thus, at the start of steps $a[r + \kappa - 3]$, all (correct) nodes receive $\langle v \rangle$ at least $n - t$ times (from different nodes). Thus, during $a[r + \kappa - 2]$, all (correct) nodes assign v to their *propose* variables (line 52) and broadcast $\langle v \rangle$ (line 53). By similar arguments, during $a[r + \kappa - 1]$, all (correct) nodes assign v and 1 to their *save*, and resp., *bit* variables (lines 56 to 57) and broadcast $\langle 1 \rangle$ (line 59). Therefore, all (correct) nodes receive $\langle 1 \rangle$ at least $n - t$ times. This implies that during $a[r + \kappa]$, the if-statement condition in line 63 holds and thus $\text{indices}_{r+\kappa}^{\text{start}} = \{v \neq \perp\}$ holds. $\square_{\text{Claim 5}}$ $\square_{\text{Lemma 4.6}}$ $\square_{\text{Theorem 4.2}}$

Theorem 4.7 *Algorithm 2 is an SSBFT recycling mechanism (Definition 4.1) that stabilizes within expected $\mathcal{O}(\kappa)$ synchronous rounds.*

Proof of Theorem 4.7 COR-validity-1 and COR-validity-2 are implied by arguments 1 and 2, respectively. COR-agreement is implied by Argument 3. The stabilization time is due to the underlying algorithms.

Argument 1 *During legal executions, if the value of index is incremented (line 63), $\exists i \in \text{Correct} : \text{wasDelivered}() = 1$ holds.* By the assumption that $\text{wasDelivered}()$ provides the proposed values used by the SSBFT multivalued consensus. Specifically, Algorithm 3 explicitly requires this, see the comment in line 38. The value decided by this SSBFT consensus is used in line 62 determines whether, during legal executions, the value of *index* is incremented module I (line 63), say, from ind_1 to ind_2 .

Argument 2 *During legal executions, if $\forall i \in \text{Correct} : \text{wasDelivered}_i() = 1$ holds, index is incremented.* Implied by Argument 1 and BC-validity of the consensus protocol.

Argument 3 *During legal executions, the increment of index is followed by the recycling of a single object, $\text{obj}[x]$, the same for all (correct) nodes.* Line 23 (Algorithm 2) uses the

value of $index$ as the returned value from $ssbftIndex.getIndex()$ when calculating the set $S(ind) = \{y \bmod indexNum : y \in \{indexNum + ind - logSize, \dots, indexNum + ind\}\}$, where $ind \in \{ind_1, ind_2\}$. For every $x \notin S(ind)$, $obj[x].recycle()$ is invoked. Since $ind_2 = ind_1 + 1 \bmod I$, during legal executions, there is exactly one index, x , that is in $S(ind_1)$ but not in $S(ind_2)$. I.e., $x = (indexNum + ind_1 - logSize) \bmod indexNum$ and only $obj[x]$ is recycled by all (correct) nodes (BC-agreement of the SSBFT consensus). $\square_{Theorem 4.7}$

5 Conclusion

We have presented an SSBFT algorithm for object recycling. Our proposal can support an unbounded sequence of SSBFT object instances. The expected stabilization time is in $\mathcal{O}(t)$ synchronous rounds. We believe that this work is preparing the groundwork needed to construct SSBFT algorithms for distributed systems, such as Blockchains and the Cloud.

When deploying an asynchronous solution, such as a consensus algorithm, in real-world systems, it is crucial to ensure the solution’s correctness remains independent of any timing bounds, which are assumed to be unknown at the time in which the solution is designed and developed. However, given a specific real-world system, which has bounded computation and communication delays, the consensus algorithm can be expected to terminate and deliver results to all non-faulty nodes within a known bounded time, which we refer to as $logSize$ (Assumption 1). This time corresponds to the duration in which it is required to log consensus objects until their results reach all non-faulty nodes. It is important to note that synchrony assumptions are imperative for any deterministic solution to the studied problem since it is equivalent to consensus. This is because the problem entails deciding both the termination of an asynchronous consensus object and whether the agreed-upon value was received by all non-faulty nodes.

As a potential avenue for future research, one could explore deterministic recycling mechanisms, say by utilizing the Dolev and Welch approach to SSBFT clock synchronization [23], to design an SSBFT SIG-index. However, their solution has exponential stabilization time, making it unfeasible in practice.

Acknowledgments

We express our gratitude to anonymous reviewers for their valuable comments. The work of E. M. Schiller was partially supported by VINNOVA, the Swedish Governmental Agency for Innovation Systems through the CyReV project under Grant 2019-03071.

References

- [1] I. Abraham and D. Dolev. Byzantine agreement with optimal early stopping, optimal resilience and polynomial complexity. In *STOC*, pages 605–614. ACM, 2015.

- [2] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In *WDAG*, volume 486 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 1990.
- [3] N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *J. Comput. Syst. Sci.*, 81(4):692–701, 2015.
- [4] K. Altisen, S. Devismes, S. Dubois, and F. Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019.
- [5] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilization by local checking and global reset (extended abstract). In *WDAG*, volume 857 of *Lecture Notes in Computer Science*, pages 326–339. Springer, 1994.
- [6] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 27–30, 1983.
- [7] M. Ben-Or, D. Dolev, and E. N. Hoch. Fast self-stabilizing Byzantine tolerant digital clock synchronization. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 385–394. ACM, 2008.
- [8] A. Binun, T. Coupaye, S. Dolev, M. Kassi-Lahlou, M. Lacoste, A. Palesandro, R. Yagel, and L. Yankulin. Self-stabilizing Byzantine-tolerant distributed replicated state machine. In *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS*, pages 36–53, 2016.
- [9] S. Bonomi, S. Dolev, M. Potop-Butucaru, and M. Raynal. Stabilizing server-based storage in Byzantine asynchronous message-passing systems: Extended abstract. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*, pages 471–479, 2015.
- [10] S. Bonomi, M. Potop-Butucaru, and S. Tixeuil. Stabilizing Byzantine-fault tolerant storage. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 894–903, 2015.
- [11] S. Bonomi, A. D. Pozzo, M. Potop-Butucaru, and S. Tixeuil. Brief announcement: Optimal self-stabilizing mobile Byzantine-tolerant regular register with bounded timestamps. In *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS*, pages 398–403, 2018.
- [12] S. Bonomi, A. D. Pozzo, M. Potop-Butucaru, and S. Tixeuil. Approximate agreement under mobile Byzantine faults. *Theor. Comput. Sci.*, 758:17–29, 2019.

- [13] G. Bracha and S. Toueg. Resilient consensus protocols. In *PODC*, pages 12–26. ACM, 1983.
- [14] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
- [15] S. Dolev. *Self-Stabilization*. MIT Press, 2000.
- [16] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Practically-self-stabilizing virtual synchrony. *J. Comput. Syst. Sci.*, 96:50–73, 2018.
- [17] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing Byzantine tolerant replicated state machine based on failure detectors. In *Cyber Security Cryptography and Machine Learning - Second International Symposium, CSCML*, pages 84–100, 2018.
- [18] S. Dolev, O. Liba, and E. M. Schiller. Self-stabilizing Byzantine resilient topology discovery and message delivery. In *Networked Systems - First International Conference, NETYS*, pages 42–57, 2013.
- [19] S. Dolev, T. Petig, and E. M. Schiller. Brief announcement: Robust and private distributed shared atomic memory in message passing networks. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*, pages 311–313, 2015.
- [20] S. Dolev, T. Petig, and E. M. Schiller. Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks. *CoRR*, abs/1806.03498, 2018. Also to appear in Springer’s *Algorithmica*.
- [21] S. Dolev, T. Petig, and E. M. Schiller. Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks. *Algorithmica*, 85(1):216–276, 2023.
- [22] S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults (abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, page 256, 1995.
- [23] S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *J. ACM*, 51(5):780–799, 2004.
- [24] S. Dubois, M. Potop-Butucaru, M. Nesterenko, and S. Tixeuil. Self-stabilizing Byzantine asynchronous unison. *J. Parallel Distributed Comput.*, 72(7):917–923, 2012.
- [25] R. Duvignau, M. Raynal, and E. M. Schiller. Self-stabilizing Byzantine fault-tolerant repeated reliable broadcast. In *SSS*, volume 13751 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2022.

- [26] M. Feldmann, T. Götze, and C. Scheideler. A loosely self-stabilizing protocol for randomized congestion control with logarithmic memory. In *Stabilization, Safety, and Security of Distributed Systems - 21st International Symposium, SSS*, volume 11914 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2019.
- [27] M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.
- [28] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [29] C. Georgiou, R. Gustafsson, A. Lindhé, and E. M. Schiller. Self-stabilization overhead: A case study on coded atomic storage. In *NETYS*, volume 11704 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2019.
- [30] C. Georgiou, O. Lundström, and E. M. Schiller. Self-stabilizing snapshot objects for asynchronous failure-prone networked systems. In P. Robinson and F. Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019.*, pages 209–211. ACM, 2019.
- [31] C. Georgiou, O. Lundström, and E. M. Schiller. Self-stabilizing snapshot objects for asynchronous failure-prone networked systems. In *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, pages 113–130, 2019.
- [32] C. Georgiou, I. Marcoullis, M. Raynal, and E. M. Schiller. Loosely-self-stabilizing Byzantine-tolerant binary consensus for signature-free message-passing systems. In *NETYS*, volume 12754 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2021.
- [33] C. Georgiou, M. Raynal, and E. M. Schiller. Self-stabilizing byzantine-tolerant recycling. *To appear in SSS*, 2023.
- [34] T. Izumi. On space and time complexity of loosely-stabilizing leader election. In C. Scheideler, editor, *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO*, volume 9439 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2015.
- [35] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. In *PODC*, pages 91–101. ACM, 1990.
- [36] D. R. Kowalski and A. Mostéfaoui. Synchronous Byzantine agreement with nearly a cubic number of communication bits: synchronous byzantine agreement with nearly a cubic number of communication bits. In *PODC*, pages 84–91. ACM, 2013.
- [37] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

- [38] C. Lenzen and J. Rybicki. Self-stabilising Byzantine clock synchronisation is almost as easy as consensus. *J. ACM*, 66(5):32:1–32:56, 2019.
- [39] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing set-constrained delivery broadcast (extended abstract). In *40th IEEE International Conference on Distributed Computing Systems, ICDCS*, pages 617–627, 2020.
- [40] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing uniform reliable broadcast. In *Networked Systems - 8th International Conference, NETYS*, pages 296–313, 2020.
- [41] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing indulgent zero-degrading binary consensus. In *ICDCN '21: International Conference on Distributed Computing and Networking*, pages 106–115, 2021.
- [42] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing multivalued consensus in asynchronous crash-prone systems. In *EDCC*, pages 111–118. IEEE, 2021.
- [43] O. Lundström, M. Raynal, and E. M. Schiller. Self-stabilizing multivalued consensus in asynchronous crash-prone systems. *CoRR*, abs/2104.03129, 2021.
- [44] O. Lundström, M. Raynal, and E. M. Schiller. Brief announcement: Self-stabilizing total-order broadcast. In *SSS*, volume 13751 of *Lecture Notes in Computer Science*, pages 358–363. Springer, 2022.
- [45] A. Maurer. Self-stabilizing Byzantine-resilient communication in dynamic networks. In *OPODIS*, volume 184 of *LIPIcs*, pages 27:1–27:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [46] A. Mostéfaoui, H. Moumen, and M. Raynal. Signature-free asynchronous Byzantine consensus with $t < n/3$, $O(n^2)$ messages. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 2–9, 2014.
- [47] T. Oliveira, R. Mendes, and A. N. Bessani. Exploring key-value stores in multi-writer Byzantine-resilient register emulations. In *OPODIS*, volume 70 of *LIPIcs*, pages 30:1–30:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.
- [48] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
- [49] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *IWMM*, volume 986 of *Lecture Notes in Computer Science*, pages 211–249. Springer, 1995.
- [50] M. O. Rabin. Randomized Byzantine generals. In *24th Annual Symposium on Foundations of Computer Science*, pages 403–409, 1983.

- [51] M. Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018.
- [52] I. Salem and E. M. Schiller. Practically-self-stabilizing vector clocks in the absence of execution fairness. In *Networked Systems - 6th International Conference, NETYS*, pages 318–333, 2018.
- [53] Y. Sudo, J. Nakamura, Y. Yamauchi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theor. Comput. Sci.*, 444:100–112, 2012.
- [54] Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely stabilizing leader election on arbitrary graphs in population protocols without identifiers or random numbers. *IEICE Trans. Inf. Syst.*, 103-D(3):489–499, 2020.
- [55] Y. Sudo, F. Ooshita, H. Kakugawa, T. Masuzawa, A. K. Datta, and L. L. Larmore. Loosely-stabilizing leader election for arbitrary graphs in population protocol model. *IEEE Trans. Parallel Distributed Syst.*, 30(6):1359–1373, 2019.
- [56] Y. Sudo, F. Ooshita, H. Kakugawa, T. Masuzawa, A. K. Datta, and L. L. Larmore. Loosely-stabilizing leader election with polylogarithmic convergence time. *Theor. Comput. Sci.*, 806:617–631, 2020.
- [57] S. Toueg. Randomized Byzantine agreements. In T. Kameda, J. Misra, J. G. Peters, and N. Santoro, editors, *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 163–178. ACM, 1984.
- [58] L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In *IPDPS*. IEEE Computer Society, 2005.