

Loosely-self-stabilizing Byzantine-tolerant Binary Consensus for Signature-free Message-passing Systems

Chryssis Georgiou ^{*} Ioannis Marcoullis ^{*} Michel Raynal [†]
 Elad Michael Schiller [‡]

Abstract

Many distributed applications, such as cloud computing, service replication, load balancing, and distributed ledgers, *e.g.*, Blockchain, require the system to solve *consensus* in which all processes reliably agree on a single value. *Binary consensus*, where the set of values that can be proposed is either zero or one, is a fundamental building block for other “flavors” of consensus, *e.g.*, multivalued, or vector, and of total order broadcast. At PODC 2014, Mostéfaoui, Moumen, and Raynal, in short MMR, presented a randomized signature-free asynchronous binary consensus algorithm. They demonstrated that their solution could deal with up to t Byzantine processes, where $t < n/3$ and n is the number of processes. MMR assumes the availability of a service of random common coins and fair scheduling of message arrivals, which does not depend on the current coin values. It completes within $O(1)$ expected time.

Our study, which focuses on binary consensus, aims at the design of an even more robust consensus protocol. We do so by augmenting MMR with self-stabilization, a powerful notion of fault-tolerance. In addition to tolerating process and communication failures, self-stabilizing systems can automatically recover after the occurrence of *arbitrary transient-faults*; these faults represent any violation of the assumptions on which the system was designed to operate (provided that the algorithm code remains intact).

We present the first loosely-self-stabilizing fault-tolerant asynchronous solution to binary consensus in Byzantine message-passing systems. This is achieved via an instructive transformation of MMR to a self-stabilizing solution that can violate safety requirements with probability $\Pr = \mathcal{O}(2^{-M})$, where $M \in \mathbb{Z}^+$ is a predefined constant that can be set to any positive value at the cost of $3Mn + \log M$ bits of local memory; n is the number of processes. The obtained self-stabilizing version of the MMR algorithm considers a far broader fault-model since it recovers from transient faults. Additionally, the algorithm preserves the MMR’s properties of optimal resilience and termination, *i.e.*, $t < n/3$, and $\mathcal{O}(1)$ expected decision time. Moreover, any instance of the proposed solution requires a bounded amount of memory.

^{*}University of Cyprus, Cyprus. {chryssis,imarco01}@ucy.ac.cy

[†]IRISA, Univ. Rennes 1, France, and Polytechnic Univ., Hong Kong. michel.raynal@irisa.fr

[‡]Chalmers University of Technology, Sweden. elad@chalmers.se

We also offer a recycling mechanism for these asynchronous objects that allows their reuse once each object completes its task and all non-faulty nodes retrieved the decided values. This mechanism itself uses synchrony assumptions and is based on a novel composition of existing techniques as well as a new self-stabilizing Byzantine-tolerant multivalued consensus algorithm for synchronous systems.

1 Introduction

We propose a loosely-self-stabilizing Byzantine fault-tolerant asynchronous implementation of *binary consensus* objects for signature-free message-passing systems.

1.1 Background and motivation

En route to constructing robust distributed systems, rose the need for different (possibly geographically dispersed) computational entities to take common decisions. Of past and recent contexts in which the need for agreement appeared, one can cherry-pick applications, such as service replication, cloud computing, load balancing, and distributed ledgers (most notably Blockchain). In distributed computing, the problem of agreeing on a single value after the proposal of values by computational entities, *processes* (sometimes called *nodes* or *processors*), is called *consensus* [75, 9]. The most basic form of the consensus problem is for processes to decide between two possible values, *e.g.*, zero or one. This version of the problem is called *binary consensus* [96, Ch. 14]. In the absence of faults, solving consensus is straightforward, however, in the presence of faults, even benign ones such as crashes, and in the face of asynchrony, the problem is not solvable deterministically (cf. [62]). This work aims to fortify consensus protocols with fault-tolerance guarantees that are more powerful than any existing known solution. Such solutions are imperative for many distributed systems that run in hostile environments, such as Blockchains.

Over the years, research into the consensus problem has tried to exhaust all the different possible variations of the problem by tweaking synchrony assumptions, the range of possible values to be agreed upon, adversarial and failure models, as well as other parameters. To circumvent known impossibility results, *e.g.*, the celebrated FLP [62], the system models are also equipped with additional capabilities, such as cryptography, oracles, *e.g.*, perfect failure detectors, and randomization [32]. Despite the decades-long research, the consensus problem remains a popular research topic. The most recent spike in interest in consensus was triggered by the Blockchain “rush” of the past decade. Agreement in a common chain of blocks is inherently a consensus problem. “Blockchain consensus” [26, 109] is a highly-researched topic, and all the “proof-of-***” concepts enclose an underlying consensus-solving mechanism.

1.2 Problem definition

The problem of letting all processes to uniformly select a single value among all the values that they propose is called consensus. When the set, V , of values that can be proposed, includes just two values, *i.e.*, $V = \{0, 1\}$, the problem is called binary consensus. Otherwise, it is called multivalued consensus.

Definition 1.1 (Binary Consensus) *Every process p_i has to propose a value $v_i \in V = \{0, 1\}$, via an invocation of the $\text{propose}_i(v_i)$ operation. Let Alg be an algorithm that solves binary consensus. Alg has to satisfy safety, *i.e.*, BC-validity and BC-agreement, and liveness, *i.e.*, BC-completion, requirements.*

- **BC-validity.** *The value $v \in \{0, 1\}$ decided by a non-faulty process is a value proposed by a non-faulty process.*
- **BC-agreement.** *Any two non-faulty processes that decide, do so with identical decided values.*
- **BC-completion.** *All non-faulty processes decide.*

Starting from the algorithm of Mostéfaoui, Moumen, and Raynal [87], from now on MMR, this study proposes an even more fault-tolerant consensus algorithm, which is a variant on MMR. Note that MMR provides randomized liveness guarantees, *i.e.*, with the probability of 1, MMR satisfies the BC-completion requirement within a finite time that is known only by expectation. The proposed solution satisfies BC-completion within a time that depends on a predefined parameter $M \in \mathbb{Z}^+$. However, it provides randomized safety guarantees, *i.e.*, with the probability of $1 - \mathcal{O}(2^{-M})$, the proposed solution satisfies the BC-validity and BC-agreement requirements. Since the number of bits that each process needs to store is $3nM + \lceil \log M \rceil$, we note that the probability for violating safety can be made, in practice, to be extremely small, where n is the number of processes, see Remark 3.1 for details.

We note that the literature often refers to BC-completion property as BC-termination. In Section 1.5, we explain the reason for this deviation.

Also, Definition 1.1 considers a single instance Binary consensus object. Our implementation considers an extended version of *recyclable* Binary consensus objects that can be stored in a δ -size set, where δ is a predefined constant (Section 4). This set can be repeatedly recycled once all objects complete their task and all non-faulty nodes retrieved their results (Section 5). Thus, the proposed solution can be reused an unbounded number of times (and still, use only a bounded amount of memory).

1.3 Fault model

We study solutions for message-passing systems. We model a broad set of failures that can occur to computers and networks, *e.g.*, due to procrastination, equivocation, selfishness, hostile (human) interference, deviation from the program code, etc. Specifically, our fault model includes up to t process failures, *i.e.*, crashed or Byzantine [75]. In detail, a faulty

process runs the algorithm correctly, but the adversary completely controls the messages that the algorithm sends, *i.e.*, it can modify the content of a message, delay the delivery of a message, or omit it altogether. The adversary's control can challenge the algorithm by creating failure patterns in which a fault occurrence appears differently to different system components. Moreover, the adversary is empowered with the unlimited ability to compute and coordinate the most severe failure patterns. We assume a known maximum number, t , of processes that the adversary can capture. We also restrict the adversary from letting a captured process impersonate a non-faulty one. In addition, we limit the adversary's ability to impact the delivery of messages between any two non-faulty processes by assuming fair scheduling of message arrivals *i.e.*, Fair Communication (FC) between non-faulty processes is assumed.

1.4 Hybrid synchronous/asynchronous approach

The proposed solution uses a hybridization of two different fault models, which their notations follow Raynal [96].

- $\text{BAMP}_{n,t}[-\text{FC}, t < n/3, \text{RCCs}]$. The studied asynchronous solutions are for message-passing systems where the algorithm cannot explicitly access the local clock or assume the existence of guarantees on the communication delay. These systems are also prone to communication failures, *e.g.*, packet omission, duplication, and reordering, as long as fair communication (FC) holds. For the sake of solvability [75, 92, 106], we also assume that the number of faulty processes $t < n/3$ is less than one-third of the number of processes in the system. This fault model, $\text{BAMP}_{n,t}[-\text{FC}, t < n/3, \text{RCCs}]$, is called the Byzantine Asynchronous Message-Passing model with at most t (out of n) faulty processes. The array $[-\text{FC}, t < n/3, \text{RCCs}]$ denotes the list of all assumptions, *i.e.*, FC and $t < n/3$ as well as *random common coins* (RCCs).
- $\text{BSMP}_{n,t}[\kappa\text{-SGC}, t < n/3, \text{RCCs}]$. This model is called the Byzantine synchronous message-passing with at most t (out of n) faulty processes, and $t < n/3$. The $\text{BSMP}_{n,t}[\kappa\text{-SGC}, t < n/3, \text{RCCs}]$ model is defined by enriching the $\text{BAMP}_{n,t}[-\text{FC}, t < n/3]$ model with a κ -state global clock, reliable communication, and a service of RCCs. A detailed presentation of $\text{BSMP}_{n,t}[\kappa\text{-SGC}, t < n/3, \text{RCCs}]$ appears in Section 5.1.

1.5 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 and network were 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 [37] assumes that these violations bring the system to an arbitrary state from which a *self-stabilizing system* should recover, see [3, 42] for details. Dijkstra requires recovery after the last occurrence of a transient-fault and once the system has recovered, it must never violate the task specification.

For the case of the studied problem and fault model, there are currently no known ways to meet Dijkstra’s self-stabilizing design criteria. *Loosely-self-stabilizing systems* [101] require that, once the system has recovered, only rarely and briefly can it violate the safety specifications. Although it is a weaker design criterion than the one defined by Dijkstra, the violation occurrence can be made to be so rare, that the risk of breaking the safety requirements of Definition 1.1 becomes negligible.

It is well-known that self-stabilizing systems cannot stop sending messages when the system’s task has so-called “terminated”, see [42, Chapter 2.3] for details. This impossibility is, mistakenly, stated as “self-stabilizing system can never terminate”. However, the system’s task can terminate but the system cannot stop sending messages. In order to avoid this confusion, as mentioned, we refer to BC-termination as BC-completion.

1.6 Related work

In this paper, the design criteria for non-self-stabilizing Byzantine fault-tolerant solutions are called BFT, and the ones for self-stabilizing Byzantine fault-tolerant are called SSBFT. We review the most related BFT and SSBFT solutions for the studied problem.

1.6.1 Impossibilities and lower-bounds

The FLP impossibility result [62] concluded that consensus is impossible to solve deterministically in asynchronous settings in the presence of even a single crash failure. In [61] it was shown that a lower bound of $t + 1$ communication steps are required to solve consensus deterministically in both synchronous and asynchronous environments. The proposed solution is a randomized one. In the presence of asynchrony, transient-faults, and (non-Byzantine) crash failures, there are known problems such as leader election and counting the number of processes in the system, for which there are no (randomized) self-stabilizing solutions [4, 8]. In this work, we consider weaker design criteria than Dijkstra’s self-stabilization.

In the presence of Byzantine faults, the consensus problem is not solvable if a third or more of the processes are faulty [75]. Thus, optimally resilient Byzantine consensus algorithms, such as the one we present, 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 [9]. We assume the absence of spoofing attacks and similar means of impersonation. In the presence of asynchrony, transient-faults, and Byzantine failures, the task of unison is known to be unsolvable (unless the strongest fairness assumptions are made) [55, 56]. As indicated by the above impossibility results, the studied problem remains challenging even under randomization and fairness assumptions during the recovery period.

1.6.2 Non-self-stabilizing non-BFT solutions

Paxos [72] is the best-known solution for the consensus problem. Despite becoming notorious for being complex [74], Paxos was followed by rich literature [108]. Raynal [96] offers a family of abstractions for solving a number of well-known problems including consensus. This line of research is easier to understand and supports well-organized implementations. Protocols implementing total order broadcast are usually built on top of consensus since consensus and total order broadcast are equivalent [30, 97].

1.6.3 Non-self-stabilizing BFT solutions

BFT consensus was tackled by many protocols [86]. Several variants of Paxos consensus tolerate such malicious processes, *e.g.*, [73]. State machine replication protocols, such as PBFT [29] and BFT-SMART [11] incorporate a BFT consensus mechanism.

Randomization can circumvent the FLP impossibility [61], which only entails deterministic algorithms. This line of work started with Ben-Or [9] using a local coin (that generated a required exponential number of communication steps in the general case) and resilience $t < n/5$, and by Rabin [95] in the same year, which assumes the availability of RCCs, allowed for a polynomial number of communication steps and optimal resilience, *i.e.*, $t < n/3$. We later discuss more extensively the notion of RCCs. Bracha [22] constructed a reliable broadcast protocol that allowed optimally-resilient binary agreement, but using a local coin needed an exponential expected number of communication steps. Cachin et al. [25] solve asynchronous binary consensus using RCCs and cryptographic threshold signatures. They achieve optimal resilience ($t < n/3$) and quadratic message-per-round complexity.

In the sequel, we focus on MMR [87] as a signature-free BFT solution for binary consensus. This algorithm is optimal in resilience, uses $O(n^2)$ messages per consensus invocation, and completes within $O(1)$ expected time. MMR can be combined with a reduction of multivalued consensus to binary consensus [89] to attain multivalued consensus with the same fault-tolerance properties.

Binary consensus is a fundamental component of total order reliable broadcast, *e.g.*, [24, 31] (see Section 1.7). In what appears as a revival of the topic, several Blockchain consensus protocols are also using similar approaches. HoneyBadger [85] was the first randomized BFT protocol for Blockchain. They employ MMR as their binary consensus protocol. The BEAT [54] suite of protocols for blockchain consensus also uses MMR.

MMR has PODC 2014 [87] and JACM 2015 [88] variations. The latter variation overcomes an implementation challenge later discussed by Tholoniati and Gramoli [105], which raised concerns regarding the liveness of the PODC 2014 variation when the adversary is allowed to control the schedule of message arrivals. Recently, Cachin and Zanolini [27] modified the MMR variation of PODC 2014 with a couple of simple modifications that cope with the above liveness concern. Specifically, they suggest imposing FIFO message delivery and an extra sampling of the arriving values before accessing the RCC. For the sake of a simple presentation, this work considers the PODC 2014 variation and assumes fair scheduling of message arrival (which does not depend on the current coin value). Thus, our results do not

implement the modifications proposed by Cachin and Zanolini.

Duvignau, Raynal, and Schiller [57, Algorithm 3] explain how to implement the FIFO message ordering in the context of SSBFT. The interested reader is offered to apply the technique for dynamic value reception proposed by Cachin and Zanolini to the proposed solution since it preserves MMR’s key algorithmic features.

Non-self-stabilizing BFT services for RCCs Randomized algorithms employ coin flips to circumvent the FLP impossibility [61], which only entails deterministic algorithms. The two known coin flip constructions are *local coins*, where each process only uses a local random function, and RCCs, where the k -th invocation of the random function by a non-faulty process, returns the same bit as to any other non-faulty process. Ben-Or [9] using a local coin, developed an asynchronous BFT Binary Consensus algorithm with $t < n/5 + 1$ resilience, but (as any local-coin-based algorithm) required an exponential number of communication steps, unless $t = O(\sqrt{n})$ where a polynomial number can be achieved. Rabin [95] was the first to introduce RCCs demonstrating the possibility of designing asynchronous BFT binary consensus algorithms with a polynomial number of communication steps and with constant expected computational rounds. The coin construction is based on Shamir’s secret sharing [100] and digital signatures for authenticating the messages exchanged. Since then, RCCs provision has become an essential tool, and many subsequent works have devised randomized coin-flipping algorithms, *e.g.*, [10, 23, 24, 25, 28, 58, 59, 90] as building blocks for consensus and other related problems, such as clock synchronization. Aspens [7] demonstrates that agreeing on RCCs is a harder problem than solving consensus, in the sense that if we can solve it, then we can solve consensus.

An important feature that RCCs algorithm must provide is *unpredictability*, that is, the outcome of the random bit at a given round should not be predicted by the Byzantine adversary before that round. In this respect, two communication models have been used in devising coin-flipping algorithms. Either *private communication* is assumed, *e.g.*, [58, 59, 28, 10] or digital signatures and other cryptographic tools are employed, *e.g.*, [100, 90, 24, 25]. In the former, the usual assumption is that processes are connected via private channels and the Byzantine adversary can have access to the messages exchanged between faulty and non-faulty processes, but not to the messages exchanged between non-faulty processes, hence providing confidentiality. In the latter, cryptographic tools (signatures) conceal the content of a message and only the intended recipient can view its content. Hence, a subtle difference between the two schemes is that with private channels, a third process does not even know whether two other processes have exchanged a message, whereas, with signatures, the third process might be aware of the message exchange, but not the message’s content. Feldman and Micali [58] show how to compile any protocol assuming private channels to a cryptographic protocol not assuming private channels which runs exactly the same.

Non-self-stabilizing synchronous BFT multivalued consensus As mentioned, self-stabilizing systems are required to use bounded memory, and thus, we are interested in recycling mechanisms for consensus objects (Section 1.2). The proposed recycling mecha-

nism uses an SSBFT multivalued consensus for the $\text{BSMP}_{n,t}[\kappa\text{--SGC}, t < n/3, \text{RCCs}]$ model, which is based on a non-self-stabilizing BFT multivalued consensus. Pease, Shostak, and Lamport [92] were the first to propose a solution that has optimal resilience to $t < n/3$ and optimal worst-case $t + 1$ synchronous rounds with exponential communication costs. Dolev and Strong [41] proposed the first solution that has optimal resilience and polynomial communication costs but not with optimal worst-case rounds. Garay and Moses [63] proposed the first solution for binary-valued Byzantine agreement with optimal resilience, polynomial communication costs, optimal $t + 1$ rounds, and early termination. Kowalski and Mostéfaoui [71] proposed the first multivalued optimal resilience, polynomial communication costs, and optimal $t + 1$ rounds, but without early stopping. Abraham and Dolev [1] advance the state of the art by offering also optimal early stopping. Unlike the above BFT multivalued consensus solutions, our BFT multivalued solution considers self-stabilization, and its implementation is an application of the well-known technique of the recomputation of floating outputs [42, Chapter 2.8].

1.6.4 Self-stabilizing crash-tolerant solutions

Lundström, Raynal, and Schiller [79] presented the first self-stabilizing solution for the problem of binary consensus for message-passing systems where processes may fail by crashing. They ensure a line of self-stabilizing solutions [80, 77, 78, 66, 65]. This line follows the approach proposed by Dolev, Petig, and Schiller [48, 49] 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 restart, see [64, 65], but only during the recovery period. Our work does not assume execution fairness either in the presence or absence of arbitrary transient-faults. As in MMR, our loosely-self-stabilizing BFT solution assumes fair scheduling of message arrivals and the accessibility to an independent service for RCCs.

We note the existence of other approaches for recovering from transient faults without assuming execution fairness during the recovery period [99, 44, 2]. However, none of these results consider both Byzantine fault-tolerance and self-stabilization.

Algorithms for loosely-self-stabilizing systems [102, 103, 104, 68, 53] mainly focus on the task of leader election and population protocols. Recently, Feldmann, Götte, and Scheider [60] proposed a loosely-self-stabilizing algorithm for congestion control. Considering a message-passing system prone to Byzantine failures, we implement leaderless binary consensus. Our loosely-self-stabilizing design criterion is slightly weaker than the one studied in [102, 103, 104, 68, 60] since it requires the loosely-self-stabilizing condition to hold only eventually.

1.6.5 Self-stabilizing BFT solutions

In the context of this dual design criteria, there are solutions for clock synchronization [111, 93, 81, 39, 110, 34, 38, 10, 67, 51, 76, 70], storage [17, 16, 20, 19, 18, 15, 14],

and gathering of mobile robots [5, 6, 36, 35]. There are also SSBFT solutions for link-coloring [82, 98], topology discovery [47, 91], overlay networks [40], exact agreement [33] approximate agreement [21], asynchronous unison [55], communication in dynamic networks [83], and reliable broadcast [57, 84]. The most relevant work is the one by Binun *et al.* [12, 13] and Dolev *et al.* [45] for a deterministic BFT emulation of state-machine replication. Binun *et al.* present the first self-stabilizing solution for synchronous message-passing systems and Dolev *et al.* present the first practically-self-stabilizing solution for partially-synchronous settings, utilizing failure detectors. We study another problem, which is binary consensus. Note that in practically-self-stabilizing systems there can be a bounded number of possible safety violations during any practically infinite period of the system execution, whereas loosely-self-stabilizing systems recover within a bounded (expected) time with no further safety violations.

To the best of our knowledge, the only SSBFT RCCs construction is the one by Ben-Or, Dolev, and Hoch [10], in short BDH, for synchronous (pulse-based) systems with private channels. They use a pipeline technique to transform the non-self-stabilizing synchronous BFT coin-flipping algorithm of Feldman and Micali [59] into a self-stabilizing one; the work in [59] assumes private channels. In [10], BDH have used their SSBFT RCCs construction as a building block for devising an SSBFT synchronous clock synchronization solution.

Our work borrows several mechanisms from BDH, such as SSBFT RCCs and SSBFT clock synchronization. We also borrow proof techniques from their random algorithm for providing SSBFT digital clock synchronization. We note the existence of an earlier SSBFT algorithm for deterministic digital clock synchronization by Dolev and Welch [52] rather than BDH’s randomized solution. We decided not to base our solution on the one by Dolev and Welch since it has exponential stabilization time.

1.7 The studied architecture of asynchronous and synchronous components

A Blockchain can be seen as a replication service for state-machine emulation in extremely hostile environments. The stacking of reliable broadcast protocols can facilitate this emulation, see Figure 1 and Raynal [96, Ch. 16 and 19]. Specifically, the order of all state transitions of the automaton can be agreed by using total order reliable broadcast. The order of the broadcasts is agreed via multivalued consensus [80]. Whenever multivalued consensus is invoked, the latter calls binary consensus for a finite number of times.

1.7.1 Using both asynchronous and synchronous components

Existing solutions for binary consensus use either randomization techniques or synchrony assumptions in order to circumvent the mentioned impossibilities, *e.g.*, FLP. The system as a whole can avoid communication-related bottlenecks by making design choices that prefer weaker synchrony assumptions for the components that are more communication demanding. Binary consensus protocols are inherently communication-intensive since a number of them can be invoked for every transition of the state-machine and each such invocation has to take

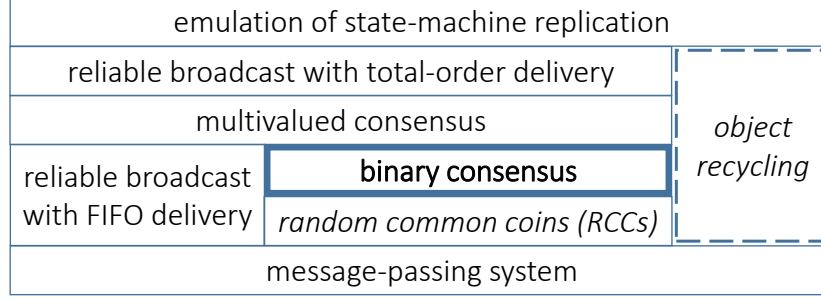


Figure 1: The hybrid architecture of asynchronous and synchronous components. The studied problem (which appears in boldface font and is surrounded by a thick frame) assumes no explicit synchrony, but it requires the availability of a service for RCCs (which appears in italic font) and fair scheduling of message arrival (which does not depend on the current coin value). The object recycling mechanism (which appears in italic font and is surrounded by a dashed frame) assumes synchrony. The other system components mentioned in Section 1.7 are presented in plain font.

at least two communication rounds, due to a lower bound by Keidar and Rajsbaum [69]. Therefore, we select to study the non-self-stabilizing probabilistic MMR algorithm [87] for solving binary consensus (in asynchronous systems) while assuming access to RCCs.

1.7.2 Random common coins (RCCs)

As already mentioned, BDH presented a synchronous SSBFT RCCs solution for synchronous 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 , it holds $rand_j = x$ occurs 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 $1 - \min\{p_0, p_1\}$. Just like MMR's PODC 2014 variation, this work assumes that $p_0 = p_1 = 1/2$.

The correctness of 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.

1.7.3 Recycling and initializing of completed consensus objects

We clarify the advantage of the studied architecture that considers a hybrid model that is composed of asynchronous, *i.e.*, MMR for the model of $\text{BAMP}_{n,t}[-\text{FC}, t < n/3, \text{RCCs}]$, and

synchronous, *i.e.*, BDH for the model of $\text{BSMP}_{n,t}[\kappa\text{--SGC}, t < n/3, \text{RCCs}]$, components.

The proposed solution uses a synchronous mechanism for object recycling, which we propose in Section 5 using a *synchronous* RCCs service, such as BDH. That is, whenever an asynchronous consensus object has completed its task, the synchronous recycling mechanism re-initializes the object’s state together with the associated instance of an RCCs service—this synchronous re-initialization facilitates the use of the single instance object in a self-stabilizing manner. As we explain in sections 2.1.3 and 2.4.1, this simplifies the correctness proof since it implies that recovery from transient-faults depends only on the completion of all operations after the occurrence of the last transient fault.

A straightforward extension can further mitigate the effect of the synchronization imposed by the recycling mechanism via the recycling of a predefined number of asynchronous objects at a time (Section 4.2). This way, the communication-intensive components remain asynchronous and synchronization occurs less often.

We point out another (challenging) extension that can be the subject of future work. Canetti and Rabin [28, Section 8] present an asynchronous (non-self-stabilizing) version (and matching implementation) of the synchronous requirements above. The proposed solution could further increase the degree of asynchrony by using a self-stabilizing variation of Canetti and Rabin. This would allow to assume that each asynchronous consensus object has its own instance of an *asynchronous* RCCs service, such as the one by Canetti and Rabin [28, Section 8].

1.8 Our contribution

We present a fundamental module for dependable distributed systems: a loosely-self-stabilizing asynchronous binary consensus algorithm for message-passing systems that are prone to Byzantine process failures. We obtain this new loosely-self-stabilizing algorithm via a transformation of the non-self-stabilizing probabilistic MMR algorithm by Mostéfaoui, Moumen, and Raynal [87] for the $\text{BAMP}_{n,t}[-\text{FC}, t < n/3, \text{RCCs}]$ model. MMR assumes that $t < n/3$ and completes within $O(1)$ expected time, where t is the number of faulty processes and n is the total number of processes. The proposed algorithm preserves these elegant properties of MMR.

In order to bound the amount of memory required to implement MMR (and our variation of MMR), we use $M \in \mathbb{Z}^+$ as a bound on the number of rounds. This implies that with a probability in $\mathcal{O}(2^{-M})$ the safety requirement of Definition 1.1 can be violated. However, as we clarify (Remark 3.1), by selecting a sufficiently large value of M , the risk of violating the safety requirements becomes negligible at affordable costs.

In the absence of transient-faults, our solution achieves consensus within a constant expected time (without assuming execution fairness). After the occurrence of any finite number of arbitrary transient-faults, the system recovers within a constant time (in terms of asynchronous communication rounds) while assuming execution fairness. Unlike in MMR, each process uses a bounded amount of memory. Moreover, the communication costs of our algorithm are similar to the non-self-stabilizing MMR algorithm. That is, in every

communication round, the proposed solution requires every non-faulty process to complete at least one round-trip with every other non-faulty process.

For the sake of providing a complete solution, this work also provides an SSBFT mechanism for the model of $\text{BSMP}_{n,t}[\kappa\text{--SGC}, t < n/3, \text{RCCs}]$ that recycles distributed objects, such as the proposed MMR solution. The proposed recycling mechanism recovers after the occurrence of the last transient fault within $\mathcal{O}(\max\{\kappa, 2(t+1)\})$ synchronous rounds, where κ is a predefined constant (Section 1.4) and t is an upper bound on the number of Byzantine nodes. We obtain this part of the solution via a novel algorithmic composition of existing solutions, such as recomputation of floating output, SSBFT multivalued consensus, and a modified version of SSBFT clock synchronization. In the context of SSBFT, this composition is of special interest since it can be used not only for object recycling, because it implements SSBFT unison for the $\text{BSMP}_{n,t}[\kappa\text{--SGC}, t < n/3, \text{RCCs}]$ model. We also present, to the best of our knowledge, the *first* SSBFT synchronous multivalued consensus solution, which is needed for the implementation of our SSBFT object recycling mechanism.

To the best of our knowledge, we propose the *first* loosely-self-stabilizing BFT algorithm for solving the problem of binary consensus in the model of $\text{BAMP}_{n,t}[\text{--FC}, t < n/3, \text{RCCs}]$ and the SSBFT recycling of these consensus objects in the model of $\text{BSMP}_{n,t}[\kappa\text{--SGC}, t < n/3, \text{RCCs}]$. As we have explained in Section 1.7.3, the composition of these two parts of the proposed solution has a long line of distributed applications, such as service replication and Blockchain. Thus, our contribution can facilitate solutions that are more fault-tolerant than the existing implementations which they cannot recover after the occurrence of the last transient fault.

1.9 Document structure

The paper proceeds with the system settings (Section 2). Section 3 briefly explains the MMR algorithm. It then presents a non-self-stabilizing interpretation of MMR that embodies the reliability guarantees for broadcast-based communications that the proposed solution uses. This non-self-stabilizing algorithm is a steppingstone to our loosely-self-stabilizing algorithm that is featured (along with its correctness poof) in Section 4. Section 5 presents a SSBFT recycling mechanism for $\text{BSMP}_{n,t}[\kappa\text{--SGC}, t < n/3, \text{RCCs}]$, and Section 6 concludes the paper.

For the reader's convenience, Table 1 (given before the bibliography) includes the Glossary, where all abbreviations are listed.

2 System Settings for $\text{BAMP}_{n,t}[\text{--FC}, t < n/3, \text{RCCs}]$

We consider an asynchronous message-passing system that has no guarantees on the communication delay. Moreover, there is no notion of global (or universal) clocks and the algorithm cannot explicitly access the local clock (or timeout mechanisms). The system consists of a set, \mathcal{P} , of n fail-prone *nodes* (sometimes called *processes* or *processors*) with unique identifiers. Any pair of nodes $p_i, p_j \in \mathcal{P}$ has access to a bidirectional communication channel,

$channel_{j,i}$, that, at any time, has at most $channelCapacity \in \mathbb{N}$ packets on transit from p_j to p_i (this assumption is due to a well-known impossibility [42, Chapter 3.2]).

In the *interleaving model* [42], 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 $c = (s_1, s_2, \dots, s_n)$. We define an *execution* (or *run*) $R = c[0], a[0], c[1], a[1], \dots$ 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 Task specifications

Next, we detail the studied task.

2.1.1 Returning the decided value

Definition 1.1 considers the **propose**(v) operation. We refine the definition of **propose**(v) by specifying how the decided value is retrieved. This value is either returned by the **propose**() operation (as in the studied algorithm [87]) or via the returned value of the **result**() operation (as in the proposed solution). In the latter case, the symbol \perp is returned as long as no value was decided. Also, the symbol Ψ indicates a (transient) error that occurs only when the proposed algorithm exceeds the bound on the number of iterations that it may take.

2.1.2 Randomized guarantees

The studied algorithm has a randomized guarantee with respect to the liveness requirement, *i.e.*, BC-completion. Specifically, MMR states that each non-faulty node decides with probability 1. Also, since MMR is a round-based algorithm, it holds that $\lim_{r \rightarrow +\infty} (\Pr_{\text{MMR}}[p_i \text{ decides by round } r]) = 1$.

In order to bound the amount of memory that the proposed algorithm uses, the proposed solution allows the algorithm to run for a bounded number of rounds. Specifically, there is a predefined constant, $M \in \mathbb{Z}^+$, such that the probability of $\Pr_{\text{proposed}}[p_i \text{ decides by round } M+1] = 1$. Due to this, the proposed algorithm provides a randomized guarantee with respect to the safety requirements, *i.e.*, BC-validity and BC-agreement. Specifically, $\Pr_{\text{proposed}}[p_i \text{ satisfies the safety requirements}] = 1 - \mathcal{O}(2^{-M})$. In other words, the proposed solution has weaker guarantees than the studied algorithm with respect to the safety requirements.

2.1.3 Invocation by algorithms from higher layers

We assume that the studied problem is invoked by algorithms that run at higher layers, such as multivalued consensus, see Figure 1. This means that eventually there is an invocation, I , of the proposed algorithm that starts from a well-initialized system state. That is, immediately before invocation I , all local states of all non-faulty nodes have the (predefined) initial

values in all variables and the communication channels do not include messages related to invocation I .

For the sake of completeness, we illustrate briefly how the assumption above can be covered [94] in the studied hybrid asynchronous/synchronous architecture presented in Figure 1. Suppose that upon the periodic installation of the common seed, the system also initializes the array of binary consensus objects that are going to be used with this new installation. In other words, once all operations of a given common seed installation are done, a new installation occurs, which also initializes the array of binary consensus objects that are going to be used with the new common seed installation. Note that the efficient implementation of a mechanism that covers the above assumption is outside the scope of this work.

2.1.4 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_{binCon} denotes the task of binary consensus, which Definition 1.1 specifies, and LE_{binCon} denotes the set of executions in which the system fulfills T_{binCon} 's requirements.

Due to the BC-completion requirement (Definition 1.1), LE_{binCon} includes only finite executions. In Section 2.4.2, we consider executions $R = R_1 \circ R_2 \circ \dots$ as infinite compositions of finite executions, $R_1, R_2, \dots \in LE_{\text{binCon}}$, such that R_x includes one invocation of task T_{binCon} , which always satisfies the liveness requirement, *i.e.*, BC-completion, but, with an exponentially small probability, it does not necessarily satisfy the safety requirements, *i.e.*, BC-validity and BC-agreement.

2.2 The fault model and self-stabilization

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

2.2.1 Benign Failures

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

Communication failures and fairness We consider solutions that are oriented towards asynchronous message-passing systems and thus they are oblivious to the time at which the packets arrive and depart. We assume that any message can reside in a communication channel only for a finite period. Also, the communication channels are prone to packet failures, such as omission, duplication, and reordering. However, if p_i sends a message infinitely often to p_j , node p_j receives that message infinitely often. We refer to the latter as the *fair communication* assumption. We also follow the assumption of MMR regarding the fair scheduling of message arrivals (also in the absence of transient-faults) that does not depend on the current coin's value. *I.e.*, the adversary does not control the network's ability to deliver messages to non-faulty nodes.

We note that MMR assumes reliable communication channels whereas the proposed solution does not make any assumption regarding reliable communications. Section 3.2 provides further details regarding the reasons why the proposed solution cannot make this assumption.

Arbitrary node failures Byzantine faults model any fault in a node including crashes, arbitrary behavior, and malicious behavior [75]. 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. Note that the adversary has the power to coordinate such actions without any limitation on his computational or communication power.

We also note that the studied algorithm, MMR, assumes the absence of spoofing attacks, and thus authentication is not needed. Also, the adversary cannot change the content of messages sent from a non-faulty node. Since MMR assumes the availability of a RCCs service, and since the only available, to the best of our knowledge, self-stabilizing RCCs algorithm, BDH [10], assumes private channels, we also assume that the communications between any two non-faulty nodes are private. That is, it cannot be read by the adversary.

For the sake of solvability [75, 92, 106], the fault model that we consider 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 in the system, *i.e.*, $3t + 1 \leq n$. The set of non-faulty nodes is denoted by *Correct* and called the set of non-faulty nodes.

2.2.2 Arbitrary transient-faults

We consider any temporary violation of the assumptions according to which the system was designed to operate. We refer to these violations and deviations as *arbitrary transient-faults* and assume that they can corrupt the system state arbitrarily (while keeping the program code intact). The occurrence of an arbitrary transient fault is rare. Thus, our model assumes that the last arbitrary transient fault occurs before the system execution starts [42]. Also, it leaves the system to start in an arbitrary state.

2.2.3 Dijkstra's self-stabilization

An algorithm is *self-stabilizing* with respect to the task of LE , when every (unbounded) execution R of the algorithm reaches within a finite period a suffix $R_{legal} \in LE$ that is legal. Namely, Dijkstra [37] requires $\forall R : \exists R' : R = R' \circ R_{legal} \wedge R_{legal} \in LE \wedge |R'| \in \mathbb{Z}^+$, where the operator \circ denotes that $R = R' \circ R''$ is the concatenation of R' with R'' . The part of the proof that shows the existence of R' is called the *convergence* (or recovery) proof, and the part that shows that $R_{legal} \in LE$ is called the *closure* proof. The main complexity measure of a self-stabilizing system is the length of the recovery period, R' , which is counted by the number of its asynchronous communication rounds during fair executions, as we define in Section 2.4.

2.3 Execution fairness and wait-free guarantees

We say that a system execution is *fair* when every step of a correct node that is applicable infinitely often is executed infinitely often and fair communication is kept. Self-stabilizing algorithms often assume that their executions are fair [42]. Wait-free algorithms guarantee that any operation (that was invoked by non-failing nodes) is always complete in the presence of asynchrony and any number of node failures. This work assumes execution fairness during the period in which the system recovers from the occurrence of the last arbitrary transient fault. In other words, the system is wait-free only during legal executions, which are absent from arbitrary transient-faults. Moreover, the system recovery from arbitrary transient-faults is not wait-free, but this bounded recovery period occurs only once throughout the system execution.

2.4 Asynchronous communication rounds

As explained in Section 1.5, it is well-known that self-stabilizing algorithms cannot (stop their execution and) stop sending messages [42, Chapter 2.3]. Moreover, their code includes a do-forever loop. The proposed algorithm uses M communication round numbers. Let $r \in \{1, \dots, M\}$ be a round number. We define the r -th *asynchronous (communication) round* of an algorithm's execution $R = R' \circ A_r \circ R''$ as the shortest execution fragment, A_r , of R in which *every* correct node $p_i \in \mathcal{P} : i \in \text{Correct}$ starts and ends its r -th iteration, $I_{i,r}$, of the do-forever loop. Moreover, let $m_{i,r,j,\text{ackReq}=\text{True}}$ be a message that p_i sends to p_j during $I_{i,r}$, where the field $\text{ackReq} = \text{True}$ implies that an acknowledgment reply is required. Let $a_{i,r,j,\text{True}}, a_{j,r,i,\text{False}} \in R$ be the steps in which $m_{i,r,j,\text{True}}$ and $m_{j,r,i,\text{False}}$ arrive to p_j and p_i , respectively. We require A_r to also include, for every pair of correct nodes $p_i, p_j \in \mathcal{P} : i, j \in \text{Correct}$, the steps $a_{i,r,j,\text{True}}$ and $a_{j,r,i,\text{False}}$. We say that A_r is *complete* if every correct node $p_i \in \mathcal{P} : i \in \text{Correct}$ starts its r -th iteration, $I_{i,r}$, at the first line of the do-forever loop. The latter definition is needed in the context of arbitrary starting system states.

Remark 2.1 *For the sake of simplifying the presentation of the correctness proof, when considering fair executions, we assume that any message that arrives in R without being transmitted in R does so within $\mathcal{O}(1)$ asynchronous rounds in R .*

2.4.1 Demonstrating recovery of consensus objects invoked by higher layers' algorithms

Note that the assumption made in Section 2.1.3 simplifies the challenge of meeting the design criteria of self-stabilizing systems. Specifically, demonstrating recovery from transient-faults, *i.e.*, convergence proof, can be done by showing completion of all operations in the presence of transient-faults. This is because the assumption made in Section 2.1.3 implies that, as long as the completion requirement is always guaranteed, then eventually the system reaches a state in which only initialized consensus objects exist.

2.4.2 Loosely-self-stabilizing systems

Satisfying the design criteria of Dijkstra’s self-stabilizing systems is non-trivial since it is required to eventually satisfy strictly always the task’s specifications. These severe requirements can lead to some impossibility conditions, as in our case of solving binary consensus without synchrony assumptions [4, 61, 55]

To circumvent such challenges, Sudo *et al.* [101] proposed the design criteria for loosely-self-stabilizing systems, which relaxes Dijkstra’s criteria by requiring that, starting from any system state, the system (i) reaches a legal execution within a relatively short period, and (ii) remains in the set of legal for a relatively long period. The definition of loosely-self-stabilizing systems by Sudo *et al.* considers the task of leader election, which any system state may, or may not, satisfy. This paper focuses on an operation-based task that has both safety and liveness requirements. Only at the end of the task execution, can one observe whether the safety requirements were satisfied. Thus, Definition 2.2 presents a variation of Sudo *et al.*’s definition that is operation-based and requires criterion (i) to hold within a finite time rather than within ‘a short period’.

To that end, Definition 2.1 says what it means for a system \mathcal{S} that implements operation $\text{op}()$ to satisfy task $T_{\text{op}()}$ ’s safety requirements with a probability $p_{\mathcal{S}}$. Definition 2.1 uses the term correct invocation of operation $\text{op}()$. Recall that in Section 2.1 we define what a correct invocation of binary consensus is, *i.e.*, it is required that all correct nodes invoke the $\text{propose}()$ operation exactly once during any execution that is in LE_{binCon} .

Definition 2.1 specifies probabilistic satisfaction of repeated invocations of operation $\text{op}()$.

Definition 2.1 *For a given system \mathcal{S} that aims at satisfying task $T_{\text{op}()}$ in a probabilistic manner, denote by $IE_{\mathcal{S}}(LE_{\text{op}()})$ the set of all infinite executions that system \mathcal{S} can run, such that for any $R \in IE_{\mathcal{S}}(LE_{\text{op}()})$ it holds that $R = R_1 \circ R_2 \circ \dots$ is an infinite composition of finite executions, $R_1, R_2, \dots \in LE_{\text{op}()}$. Moreover, each $R_x : x \in \mathbb{Z}^+$ includes the correct invocation of $\text{op}()$ that always satisfies $T_{\text{op}()}$ ’s liveness requirements.*

We say that R satisfies task $T_{\text{op}()}$ ’s safety requirements with probability \Pr_R if (i) for any $x \in \mathbb{Z}^+$ it holds that $R_x \in LE_{\text{op}()}$ with probability $\Pr_{R_x} \leq \Pr_R$ and (ii) for any $x, y \in \mathbb{Z}^+$ the event of $R_x \in LE_{\text{op}()}$ and $R_y \in LE_{\text{op}()}$ are independent. Furthermore, we say system \mathcal{S} satisfies task $T_{\text{op}()}$ with probability $\Pr_{\mathcal{S}}$ if $\forall R \in IE_{\mathcal{S}}(LE_{\text{op}()}) : \Pr_R \leq \Pr_{\mathcal{S}}$.

Definition 2.2 specifies probabilistic operation-based eventually-loosely-self-stabilizing systems.

Definition 2.2 (Eventually-loosely-self-stabilizing systems) *Let \mathcal{S} be a system that implements a probabilistic solution for task $T_{\text{op}()}$. Let R be any unbounded execution of \mathcal{S} , which includes repeated sequential and correct invocations of $\text{op}()$, such that task $T_{\text{op}()}$ completes within a period of $\ell_{\mathcal{S}}$ steps in R . Suppose that within a finite number of steps in R , the system \mathcal{S} reaches a suffix of R that satisfies $T_{\text{op}()}$ ’s safety requirements with the probability $\Pr_{\mathcal{S}} = 1 - p : p \in o(\ell_{\mathcal{S}})$. In this case, we say that system \mathcal{S} is eventually-loosely-self-stabilizing, where $\ell_{\mathcal{S}}$ is the complexity measure.*

Definition 2.2 says that any eventually-loosely-self-stabilizing system recovers within a finite period. After that period, the probability to violate safety-requirement is exponentially small. This work shows that the studied algorithm has an eventually-loosely-self-stabilizing variation for which the probability to violate safety can be made so low that it becomes negligible (Remark 3.1).

3 Non-self-stabilizing MMR for $\text{BAMP}_{n,t}[-\text{FC}, t < n/3, \text{RCCs}]$

We review the MMR algorithm (Section 3.1). This algorithm considers a communication abstraction named BV-broadcast, which we bring before we present the details of MMR. Then, we present a non-self-stabilizing BFT algorithm (Section 3.2) that serves as a steppingstone to the proposed SSBFT algorithm (Section 4).

3.1 The MMR algorithm

Algorithm 1 presents the MMR algorithm [87], which considers an underlying communication abstraction named BV-broadcast. Recall that the set *Correct* denotes the set of nodes that do not commit failures.

3.1.1 Broadcasting of binary-values

MMR uses an all-to-all broadcast operation of binary values. That is, the operation, $\text{bvBroadcast}(v)$, assumes that all the correct nodes invoke $\text{bvBroadcast}(w)$, where $v, w \in \{0, 1\}$.

Task definition The set of values that are BV-delivered to node p_i are stored in the read-only variable binValues_i , which is initialized to \emptyset . Next, we specify under which conditions values are added to binValues_i .

- **BV-validity.** Suppose that $v \in \text{binValues}_i$ and p_i is correct. It holds that v has been BV-broadcast by a correct node.
- **BV-uniformity.** $v \in \text{binValues}_i$ and p_i is correct. Eventually $\forall j \in \text{Correct} : v \in \text{binValues}_j$.
- **BV-completion.** Eventually $\forall i \in \text{Correct} : \text{binValues}_i \neq \emptyset$ holds.

The above requirements imply that eventually $\exists s \subseteq \{0, 1\} : s \neq \emptyset \wedge \forall i \in \text{Correct} : \text{binValues}_i = s$ and the set s does not include values that were BV-broadcast only by Byzantine nodes.

Algorithm 1: Non-self-stabilizing MMR algorithm for Binary BFT consensus with $t < n/3$, $\mathcal{O}(n^2)$ messages, and $\mathcal{O}(1)$ expected time; code for p_i

```

1 operation bvBroadcast( $v$ ) do broadcast bVAL( $v$ );
2 upon bVAL( $vJ$ ) arrival from  $p_j$  begin
3   if (bVAL( $vJ$ ) received from  $(t + 1)$  different nodes and bVAL( $vJ$ ) not yet
      broadcast) then
4     broadcast bVAL( $vJ$ ) /* a node echoes a value only once          */
5   if (bVAL( $vJ$ ) received from  $(2t + 1)$  different nodes) then
6     binValues  $\leftarrow$  binValues  $\cup \{vJ\}$  /* local delivery of a value */

7 operation propose( $v$ ) begin
8   ( $est, r$ )  $\leftarrow$  ( $v, 0$ );
9   do forever begin
10     $r \leftarrow r + 1$ ;
11    bvBroadcast EST[ $r$ ]( $est$ );
12    wait(binValues[ $r$ ]  $\neq \emptyset$ ) ; /* binValues[ $r$ ] has not necessarily obtained
       its final value when wait returns */
13    broadcast AUX[ $r$ ]( $w$ ) where  $w \in binValues[ $r$ ]$ ;
14    wait  $\exists$  a set of binary values,  $vals$ , and a set of  $(n-t)$  messages AUX[ $r$ ]( $x$ ),
       such that  $vals$  is the set union of the values,  $x$ , carried by these  $(n-t)$ 
       messages  $\wedge vals \subseteq binValues[ $r$ ]$ ;
15     $s[ $r$ ] \leftarrow$  randomBit();
16    if ( $vals = \{v\}$ ) then % i.e.,  $|vals| = 1$  %
17      if ( $v = s[ $r$ ]$ ) then
18        decide( $v$ ) if not yet done
19       $est \leftarrow v$ ;
20    else  $est \leftarrow s[ $r$ ]$ ;

```

Implementation MMR uses the `bvBroadcast(v)` operation (line 1) to reliably deliver a `bVAL(v)` message containing a single binary value, v . Such values are propagated via a straightforward “echo” mechanism that repeats any arriving value at most once per sender. In detail, the mechanism invokes a broadcast of the proposed value v . Upon the arrival of value vJ from at least $t + 1$ distinct nodes, vJ is replayed via broadcast (but only if this was not done earlier). Also, if vJ was received by at least $2t + 1$ different nodes, then vJ is added to a set `binValues`. On round r of MMR’s operation `propose(v)`, the set `binValues` appears as `binValues[r]`.

Note that no correct node can become aware of when its local copy of the set `binValues` has reached its final value. Suppose this would have been possible, consensus can be solved by instructing each node deterministically select a value from the set `binValues` and by that

contradict FLP [62].

3.1.2 MMR's binary randomized consensus algorithm

Variables Algorithm 1 uses variable r (initialized by zero) for counting the number of asynchronous communication rounds. The variable est holds the current estimate of the value to be decided. As mentioned in Section 1.7.2, the operation $\text{randomBit}(r)$ retrieves the value of the RCC on round r . The set $vals \subseteq \{0, 1\}$ holds the value received during the current round. Recall that node $p_i \in \mathcal{P}$ stores the binary values received in a round r via a $\text{bvBroadcast}()$ in the read-only set $\text{binValues}_i[r]$.

Detailed description MMR's main algorithm (appearing as the $\text{propose}(v)$ operation) comprises three phases. After initialization (line 8), Algorithm 1 enters a do forever loop (lines 9–20) that executes endlessly, reflecting the non-deterministic nature of its completion guarantees. Every iteration signifies a new round of the protocol by initiating with a round number increment (line 10) and is performed via the following phases.

- *Query the estimated binary values (lines 11–12):* The estimate est is broadcast via the $\text{bvBroadcast}()$ protocol. Due to the BV-completion property, eventually, the set $\text{binValues}[r]$ is populated with at least one binary value, w . Even though the system might not reach the final value of the set during round r , by BV-validity we know that any value in the set is an estimated value during round r of at least one correct node.
- *Inform about the query results (lines 13–14):* The auxiliary message, $AUX(w)$, carrying the value of $\text{binValues}[r]$ is broadcast. Note that all the correct nodes, p_j , broadcast $w \in \text{values}_j[r]$, i.e., a value that is estimated by at least one correct node. However, arbitrary binary values can be broadcast by the Byzantine nodes.

Processor p_i then waits for the arrival of $AUX(w)$ messages from $n - t$ distinct nodes, and gathers their attached values, w , in the set $vals$. By waiting for $n - t$ arrivals of these $AUX()$ messages, Algorithm 1 can:

- Sift out values that were sent only by Byzantine nodes, cf. $vals_i \subseteq \text{binValues}_i[r]$ at line 14.
- Guarantee that, for a given round r , it holds that $\exists i \in \text{Correct} : vals_i = \{v\} \implies \forall j \in \text{Correct} : v \in vals_j$. Also, $vals_i \subseteq \{0, 1\}$ and any $v \in vals_i$ is an estimated value that was BV-broadcast by at least one correct node.
- *Try-to-decide (lines 16–20):* If there is a single value in $vals$, then this value serves as the estimated value for the next round. This is also the decided value if it coincides with the output of the RCC and the node has not yet decided. If $vals$ contains both of the binary values, the RCC output serves as the estimated value for the next round. Note that deciding on a value does not mean that any node can stop executing Algorithm 1. (The non-self-stabilizing version of MMR can be found in [87].)

We end the description of Algorithm 1 by bringing a couple of examples that illustrate how the try-to-decide phase works. Note that if all correct nodes estimate the same value during round r , then $\exists x \in \{0, 1\} : \forall i \in \text{Correct} : x \in \text{binValues}[r]_i$ holds, which means that $\exists x \in \{0, 1\} : \forall i \in \text{Correct} : \text{vals}_i = \{x\}$ holds during round r . Moreover, the proof of MMR [87] shows that $\exists x \in \{0, 1\} : \forall i \in \text{Correct} : \text{vals}_i = \{x\}$ holds for any round $r' \geq r$. Thus, the decision of x depends only on the value of the RCC. In other words, the RCC has the “correct value” with probability $1/2$ and the algorithm decides.

Now suppose that, for any reason, $\exists x \in \{0, 1\} : \forall i \in \text{Correct} : \text{vals}_i = \{x\}$ does not hold during round r . Then, any node that decides on round r decides the value of the common coin. Also, the ones that do not decide on round r , since $\text{vals} = \{0, 1\}$, estimate for round $r + 1$ the value of the common coin. Therefore, BC-agreement holds in this case. Moreover, all the nodes for which $\text{vals} = \{0, 1\}$ holds during round r select “the correct” estimated value from the set vals with probability $1/2$, and thus, the system reaches a state in which all nodes have the same estimated value. As discussed above, this state leads to agreement with probability $1/2$. More details can be found in [87].

3.2 The non-self-stabilizing yet bounded version of the studied algorithm

After reviewing MMR, we transform the code of Algorithm 1 into Algorithm 2, which has a bound, M , on the number of iterations of the do-forever loop in lines 9 to 20. In this paper, Algorithm 2 serves as a steppingstone towards the proposed solution, which appears in Algorithm 3. We start the presentation of Algorithm 2 by weakening the assumptions that the studied solution has about the communication channels. This will help us later when presenting the proposed solution.

3.2.1 Variables

Algorithm 2 uses variable r (initialized to zero) for counting the number of asynchronous communication rounds. During round r , every node $p_i \in \mathcal{P}$ stores in the set $\text{est}_i[r][i]$ its estimated decision values, where $\text{est}_i[0][i] = \{v\}$ stores its own proposal and $\text{est}_i[M+1][i]$ aims to hold the decided value. Since nodes exchange these estimates, $\text{est}_i[r][j]$ stores the last estimate that p_i received from p_j . Note that $\text{est}_i[r][j] \subseteq \{0, 1\}$ holds a set of values and it is initialized by the empty set, \emptyset . At the end of round r , node $p_i \in \mathcal{P}$ tests whether it is ready to decide after it selects a single value $w \in \text{est}_i[r][i]$ to be exchanged with other nodes. In order to ensure reliable broadcast in the presence of packet loss, there is a need to store w in auxiliary storage, $\text{aux}_i[r][i]$, so that p_i can retransmit w . Note that all entries in $\text{aux}[]$ are initialized to \perp .

3.2.2 Transforming the assumptions about the communication channels

MMR assumes reliable communication channels when broadcasting in a quorum-based manner, *i.e.*, sending the same message to all nodes in the system and then waiting for a reply

Algorithm 2: Non-self-stabilizing BFT binary consensus that uses M iterations and violates safety with a probability that is in $\mathcal{O}(1/2^M)$; code for p_i .

```

21 operations: propose( $v$ ) do  $\{(est[0][i], aux[0][i] \leftarrow (\{v\}, \perp)\}$ ;
22 result() do  $\{\text{if } (est[M+1][i] = \{v\}) \text{ then return } v \text{ else if } (r \geq M \wedge \text{infoResult}() \neq \emptyset) \text{ then}$ 
    return}  $\Psi$  else return  $\perp$ ;  $\}$ 
23 macros: binValues( $r, x$ ) return  $\{y \in \{0, 1\} : \exists s \subseteq \mathcal{P} : |\{p_j \in s : y \in est[r][j]\}| \geq x\}$ ;
24 infoResult() do  $\{\text{if } (\exists s \subseteq \mathcal{P} : n-t \leq |s| \wedge (\forall p_j \in s : aux[r][j] \in binValues(r, 2t+1))) \text{ then return}$ 
     $\{aux[r][j]\}_{p_j \in s}$  else return  $\emptyset$ ;  $\}$ 
25 functions: decide( $x$ ) begin
26   if  $(est[M+1][i] = \emptyset \vee aux[M+1][i] = \perp)$  then  $(est[M+1][i], aux[M+1][i]) \leftarrow (\{x\}, x)$ ;
27 tryToDecide(values) begin
28   if  $(values \neq \{v\})$  then  $est[r][i] \leftarrow \{\text{randomBit}(r)\}$ ;
29   else  $\{est[r][i] \leftarrow \{v\}; \text{if } (v = \text{randomBit}(r)) \text{ then decide}(v)\}$ ;
30 do forever begin
31   if  $(est[0][i] \neq \emptyset)$  then
32      $r \leftarrow \min\{r+1, M\}$ ;
33     repeat
34       foreach  $p_j \in \mathcal{P}$  do send EST(True,  $r, est[r-1][i] \cup binValues(r, t+1)$ ) to  $p_j$ 
35       if  $(\exists w \in binValues(r, 2t+1))$  then  $aux[r][i] \leftarrow w$ ;
36     until  $aux[r][i] \neq \perp$ ;
37     repeat
38       foreach  $p_j \in \mathcal{P}$  do send AUX(True,  $r, aux[r][i]$ ) to  $p_j$ 
39     until infoResult()  $\neq \emptyset$ ;
40     tryToDecide(infoResult());
41 upon EST( $aJ, rJ, vJ$ ) arrival from  $p_j$  do begin
42    $est[rJ][j] \leftarrow est[rJ][j] \cup vJ$ ;
43   if ( $aJ$ ) then send EST(False,  $rJ, est[rJ-1][i]$ ) to  $p_j$ ;
44 upon AUX( $aJ, rJ, vJ$ ) arrival from  $p_j$  do begin
45   if ( $vJ \neq \perp$ ) then  $aux[rJ][j] \leftarrow vJ$ ;
46   if ( $aJ$ ) then send AUX(False,  $rJ, aux[rJ][i]$ ) to  $p_j$ ;

```

from the maximum number of nodes that guarantee never to block forever. After explaining why the proposed algorithm cannot make this assumption, we present how Algorithm 2 provides the needed communication guarantees.

The challenge Without a known bound on the capacity of the communication channels, self-stabilizing end-to-end communications are not possible [42, Chapter 3]. In the context of self-stabilization and quorum systems, Dolev, Petig, and Schiller [49] explained that one has to avoid situations in which communicating in a quorum-based manner can lead to a contradiction with the system assumptions. Specifically, the asynchronous nature of the system can imply that there is a subset of nodes that are able to complete many round-trips with a given sender, while the other nodes in \mathcal{P} accumulate messages in their communication chan-

nels, which must have bounded capacity. If such a scenario continues, the channel capacity might drive the system either to block or remove messages from the communication channel before their delivery. Therefore, the proposed solution weakens the required properties for FIFO reliable communications when broadcasting in a quorum-based manner.

Self-stabilizing communications One can consider advanced automatic repeat request (ARQ) algorithms for reliable end-to-end communications, such as the ones by Dolev *et al.* [46, 43]. However, our variation of MMR requires only communication fairness. Thus, we can address the above challenge by looking at simple mechanisms for assuring that, for every round r , all correct nodes eventually receive messages from at least $n - t$ nodes (from which at least $n - 2t$ must be correct). For the sake of a simple presentation, we start by reviewing these considerations for the $AUX()$ messages before the ones for the $EST()$ messages.

AUX() messages For a given round number, r , sender p_j , and receiver p_i , the repeat-until loop in lines 38 to 39 makes sure, even in the presence of packet loss, that p_i receives at least $(n - t)$ messages of $AUX(\bullet, rnd = r, aux = w) : aux_j[r][j] = w$ from distinguishable senders. This is because line 38 broadcasts the message $AUX(ack = \text{True}, rnd = r, \bullet)$ and upon its arrival to p_j , line 46 replies with $AUX(ack = \text{False}, rnd = r, \bullet)$. Note that duplication is not a challenge since, for a given round number r , p_j always sends the same $AUX(\bullet, rnd = r, aux = w) : aux_j[r][j] = w$ message. Algorithm 2 deals with packet re-ordering by storing all information arriving via $AUX[]()$ messages in the array $aux[][]$. We observe from the code of Algorithm 2 that FIFO processing is practiced since during the r -th iteration of the do-forever loop in lines 30 to 40, node p_i nodes only the values stored in $aux_i[r][]$.

EST() messages. Recall that Algorithm 1 uses the $bvBroadcast()$ operation for broadcasting $EST[r](est[r])$ messages (line 11). The operation $bvBroadcast()$ sends $bVAL(v)$ messages, where $v = est[r-1]$ and possibly also the complementary value $v' \in \{0, 1\} \setminus \{est[r-1]\}$.

For the sake of a concise presentation, Algorithm 2 embeds the code of operation $bvBroadcast()$ into its own code. Thus, in Algorithm 2, node p_i sends $EST(\bullet, rnd = r, est = e)$ messages, where the value e of the field est is a set that includes p_i 's estimated value, $v : est_i[r-1][i] = \{v\}$, from round number $r - 1$ and perhaps also the complementary value, $v' \in binValues(r, t+1) \setminus \{v\}$, see line 34 for details ($binValues()$ may return any subset of $\{0, 1\}$). Note that once p_i adds the complementary value, v' , to the field est , the value v' remains in the field est in all future broadcasts of $EST(\bullet, rnd = r, est = e)$.

Thus, the repeat-until loop in lines 33 to 36 has at least one value, v , that appears in the field est of every $EST(\bullet, rnd = r, est = e)$ message, and a complementary value, v' , that once it is added, it always appears in e . Thus, eventually, p_i broadcasts the same $EST(\bullet, rnd = r, est = e)$ message. Therefore, packet loss is tolerated due to the broadcast repetition in lines 33 to 36. Duplication is tolerated due to the union operator that p_i uses for storing arriving information from p_j (line 42). Concerning reordering tolerance, the value $est_i[r-1][i]$ always appears in e . Thus, once the value v is added to $est_j[r-1][i]$

due to the arrival of a $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v, \bullet\})$ message from p_i to p_j (line 42), v is always present in $\text{est}_j[r-1][i]$. The same holds for any complementary value, v' , that p_i adds to later on to e due to the union operation (line 42). This means, that reordering of $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v, \bullet\})$ messages that do, and do not, include the complementary value, v' , does not play a role.

3.2.3 Detailed description

As in MMR, Algorithm 2 includes the following three stages.

1. *Invocation.* An invocation of operation **propose**(v) (line 21) initializes $\text{est}_i[0][i]$ with the estimated value v . No communication or decision occurs before such an invocation occurs. These actions are only possible through the lines enclosed in the do forever loop (lines 30 to 40). These lines are not accessible before such an invocation, because of the condition of line 31. Each iteration of the do forever loop is initiated with a round increment (line 32); this line ensures that r is bounded by M .
2. *Communication.* The communication mechanism is detailed in Section 3.2.2. The first communication phase, which queries the estimated binary values, is implemented in the repeat-until loop of lines 33–36. The receiver’s side of this communication is given in the code of lines 41–43. Similarly, the second communication phase, which informs about the query results through the use of auxiliary messages, is given in the repeat-until loop of lines 37–39. Lines 44–46 are the receiver side’s actions for this phase.
3. *Decision.* The decision phase (line 40) is a call to function **tryToDecide**(\cdot). Lines 27 to 29 are the implementation of **tryToDecide**(\cdot). This exactly maps the *Try-to-decide* phase of MMR: (i) If the *values* set that was composed of the auxiliary messages that were received is a single value, then this is the estimate of the next round. (ii) If this is also the output of **randomBit**(\cdot) then this is the value to be decided. (iii) If *values* is not a single value then the estimate for the next round is the **randomBit**(\cdot) output. The actual decision action (line 26) is for both $\text{est}[M+1][i]$ and $\text{aux}[M+1][i]$ to be assigned the decided value.

As specified in Section 2.1, the function **result**(\cdot) (line 22) aims to return the decided value. However, the \perp symbol is returned when no value was decided. Also, it indicates whether r has exceeded the limit M , in which case it returns the error symbol Ψ , laying the ground for the proposed self-stabilizing algorithm presented in Section 4 (Algorithm 3).

3.2.4 Bounding the number of iterations

Algorithm 2 preallocates $\mathcal{O}(M)$ of memory space for every node in the system, where $M \in \mathbb{Z}^+$ is a predefined constant that bounds the maximum number of iterations that Algorithm 2 may take. Lemma 3.1 shows that Algorithm 2 may exceed the limit M with a probability

that is in $\mathcal{O}(2^{-M})$. Once that happens, the safety requirements of Definition 1.1 can be violated. As an indication of this occurrence, the `result()` operation returns the transient error symbol, Ψ , which some nodes might return. Remark 3.1 explains that it is possible to select a value for M , such that the probability for a safety violation is negligible.

Lemma 3.1 *By the end of round r , with probability $\Pr(r) = 1 - (1/2)^r$, we have $\text{result}_i() \in \{0, 1\} : p_i \in \mathcal{P} : i \in \text{Correct}$.*

Proof Sketch of Lemma 3.1 The proof uses Claim 3.2.

Claim 3.2 $\exists v \in \{0, 1\} : \forall i \in \text{Correct} : \text{est}_i[r][i] = \{v\}$ holds with the probability $\Pr(r) = 1 - (1/2)^r$.

Proof of Claim 3.2 Let values_i^r be the parameter that p_i passes to `tryToDecide()` (line 27) on round r . If $\forall k \in \text{Correct} : \text{values}_i^r = \{0, 1\}$ or $\forall k \in \text{Correct} : \text{values}_i^r = \{v_k(r)\}$ hold, p_k assigns the same value to $\text{est}_k[r][k]$, which is $\{\text{randomBit}_k(r)\}$, and resp., $v_k(r)$. The remaining case is when some correct nodes assign $\{v_k(r)\}$ to $\text{est}_k[r][k]$ (line 29), whereas others assigns $\{\text{randomBit}_k(r)\}$ (line 28).

Recall the assumption that the Byzantine nodes have no control over the network or its scheduler. Due to the RCC properties, $\text{randomBit}_k(r)$ and $\text{randomBit}_k(r')$ are independent, where $r \neq r'$. The assignments of $\{v_k(r)\}$ and $\{\text{randomBit}_k(r)\}$ are equal with the probability of $\frac{1}{2}$. Thus, $\Pr(r)$ is the probability that $[\exists r' \leq r : \text{randomBit}(r) = v(r)] = \frac{1}{2} + (1 - \frac{1}{2})\frac{1}{2} + \dots + (1 - \frac{1}{2})^{r-1}\frac{1}{2} = 1 - (\frac{1}{2})^r$. $\square_{\text{Claim 3.2}}$

The complete proof shows that the repeat-until loop in lines 33 to 36 cannot block forever and that all the correct nodes p_i keep their estimated value $\text{est}_i = \{v\}$ and consequently the predicate $(\text{values}_i^{r'} = \{v\})$ at line 28 holds for round r' , where $\text{values}_i^{r'} = \cup_{j \in s} \{\text{aux}_i[r][j]\}$. With probability $\Pr(r) = 1 - (1/2)^r$, by round r , $\text{randomBit}(r) = v$ holds. Then, the if-statement condition of line 28 does not hold and the one in line 29 does hold. Thus, all the correct nodes decide v . $\square_{\text{Lemma 3.1}}$

Remark 3.1 (safety in practical settings) *By Lemma 3.1, it is known that, asymptotically speaking, $\Pr(M)$ becomes exponentially small as M grows linearly. Therefore, for a given system, \mathcal{S} , we can select $M \in \mathbb{Z}^+$ to be, say, 150, so it would take at least $\ell_{\mathcal{S}} = 10^{100}$ invocations of binary consensus to lead to at most one expected instance in which the requirements of Definition 1.1 are violated. Note that for $M = 150$, the arrays `est[]` and `aux[][]` require the allocation of 57 bytes per node, since each node needs only $3nM + \lceil \log M \rceil$ bits of memory. So, \mathcal{S} can be implemented as a practical system. We believe that one expected violation in every $\ell_{\mathcal{S}}$ invocations implies a negligible risk.*

4 Self-stabilizing BFT MMR for BAMP_{n,t}[-FC, t < n/3, RCCs]

Algorithm 3 presents a solution that can recover from transient-faults. We demonstrate the correctness of that solution in Section 4.3. The boxed lines in Algorithm 3 are relevant only to an extension (Section 4.1) that accelerates the notification of the decided value.

Algorithm 3: Recovering from transient-faults

Recall that by Section 2.4.1, the main concern that we have when designing a loosely-self-stabilizing version of MMR is to make sure that no transient fault can cause the algorithm to not complete, *e.g.*, block forever in one of the repeat-until loops in lines 33 to 36 and 37 to 39 of Algorithm 2.

Recall that Algorithm 2 is a code transformation of MMR [87] that runs for M iterations and violates Definition 1.1's safety requirement with a probability that is in $\mathcal{O}(2^{-M})$. The proposed solution appears in Algorithm 3. We obtain this solution via code transformation from Algorithm 2. The latter transformation aims to offer recovery from transient-faults.

Note that a transient fault can corrupt the state of node $p_i \in \mathcal{P}$ by, for example, setting $est_i[i]$ with $\{0, 1\}$. Line 71 addresses this concern. Another case of state corruption is when the round counter, r_i , equals to r , but there is $r' < r$ and entries $est_i[r']$ or $aux_i[r']$ that point to their initial values *i.e.*, $\exists r' \in \{1, \dots, r-1\} : est_i[r'][i] = \emptyset \vee aux_i[r'][i] = \perp$. Line 73 addresses this concern. Since we wish not that the for-each condition in line 72 to hold when a correct node decides, line 62 makes sure that all entries of $est[r']$ and $aux[r']$ store the decided value, where r' is any round number that is between the current round number, r , and $M+1$, which is the entry that stores the decided value.

The last concern that Algorithm 3 needs to address is the fact that the repeat-until loop in lines 37 to 39 of Algorithm 2 depends on the assumption that $aux_i[r][i] \neq \perp$, which is supposed to be fulfilled by the repeat-until loop in lines 33 to 36 of Algorithm 2. However, a transient fault can place the program counter to point at line 38 without ever satisfying the requirement of $aux_i[r][i] \neq \perp$. Therefore, Algorithm 3 combines in lines 70 to 77 the repeat-until loops of lines 33 to 36 and 37 to 39 of Algorithm 2. Similarly, it combines in lines 80 to 82 of the upon events in lines 41 to 43 and lines 44 to 46 of Algorithm 2.

4.1 Extension: eventually silent self-stabilization Byzantine fault-tolerance

Self-stabilizing systems can never stop the exchange of messages until the consensus object is deactivated, see [42, Chapter 2.3] for details. We say that a self-stabilizing system is *eventually silent* if every legal execution has a suffix in which the same messages are repeatedly sent using the same communication pattern. We describe an extension to Algorithm 3 that, once at least $t+1$ nodes have decided, lets all correct nodes decide and reach the M -th round quickly. Once the latter occurs, the system execution becomes silent. This property makes Algorithm 3 a candidate for optimization, as described in [50].

The extension idea is to let node p_i wait until at least $t+1$ nodes have decided. Once that happens, p_i can notify all nodes about this decision because at least one of these $t+1$ nodes is correct. Algorithm 3 (including the boxed code-lines) does this by setting the round number, r , to have the value of $M+1$ when deciding (line 63) and allowing r to have the value of up to $M+1$ (line 69). Also, line 79 decides value w whenever it sees that it was decided by $t+1$ other nodes, since at least one of the must be correct.

Since a transient fault can cause the nodes to exceed their storage limit, there is a need to

indicate that to the invoking algorithm. Therefore, the self-stabilizing version of Algorithm 3 uses the operation `result()` to return the transient error symbol, Ψ . In order to ensure that $\forall p_i \in \mathcal{P} : i \in \text{Correct} \wedge \text{result}_i() = \Psi$, the self-stabilizing version of Algorithm 3 runs a completion procedure, that is based on additional synchronization assumptions, which we define next. We clarify that these additional assumptions impact Algorithm 3 only in the presence of transient-faults. Otherwise, the system is assumed to be asynchronous.

We assume that in the presence of transient-faults, any pair $p_i, p_j \in \mathcal{P}$ of correct nodes is able to complete at least one round-trip of messages exchange whenever p_i is able to exchange at most θ round trips with all other correct nodes $p_k \in \mathcal{P} \setminus \{p_i\} : k \in \text{Correct}$. Note that a faulty node $p_{byz} \in \mathcal{P}$ can attempt to complete rounds trips with p_i much faster than any correct node p_k . For example, p_{byz} can respond to any messages from p_i with all the θ acknowledgments p_i would need to receive for the perspective messages that it is going to send to p_{byz} . By flooding the network with responses, p_{byz} creates scenarios in which p_i believes that it has completed θ round trips without this ever occurring. For this reason, the proposed completion procedure counts the number of round trips each node, p_k , was able to complete with p_i ever since p_j has completed a round trip with p_i . Moreover, when summing up the number of these round-trips, p_i ignores the t ‘fastest’ node since they might be faulty.

By identifying the faulty nodes that are ‘too slow’, *i.e.*, the ones that do not complete round trips with p_i according to the above synchronization assumption, p_i can safely avoid blocking when waiting for all trusted nodes to respond. Specifically, during the execution of the completion procedure, only dedicated messages, `tEST(phs, ct, val)` are to be used, where *phs* is a phase number, *ct* is a round-trip counter, and *val* is the sender’s latest estimated value. The procedure uses three phases. Each phase completes (and the next one starts) when p_i receives an acknowledgment from all trusted nodes that they have entered this phase (or a higher one). This way, when p_i phase number changes from zero to one, we know that all correct nodes were able to share the latest estimation value that they had before starting the completion procedure. Moreover, when p_i phase number changes from one to two, we know that all correct nodes were able to share the estimated values that they received during the first phase. Furthermore, when p_i phase number changes from two to three, we know that all correct nodes are aware that the correct nodes were able to exchange all of their estimated values and the procedure can terminate.

4.1.1 Constants and variables:

Node p_i store the round-trip counters in the `ct[False, True][0, ..., n-1]` array, where `cti[False][j]` and `cti[True][j]` store the sender-side, and resp., receiver-side counters of messages that p_i and p_j exchange. The `rt[0, ..., n-1][0, ..., n-1]` array stores in `rti[j][k]` the number of replies p_i received from p_k ever since p_j has completed its last round-trip with p_i (or since the procedure invocation). Both `ct[][]` and `rt[][]` holds integers of at most $B = 4(\theta + 1)(n + 1)$ states that are initialized with the zero value. The array `phs[0, ..., n-1]` holds the phase numbers, where `phsi[i]` stores p_i ’s phase number and `phsi[j]` stores the highest value received from p_j ever since the invocation of the procedure.

Part A of Algorithm 3: SSBFT MMR, code for p_i .

```

47 constants:  $initState := (0, [[\emptyset, \dots, \emptyset], \dots, [\emptyset, \dots, \emptyset]], [[\perp, \dots, \perp], \dots, [\perp, \dots, \perp]]);$ 
48 operations:  $propose(v)$  begin
49    $(r, est, aux) \leftarrow initState; est[0][i] \leftarrow \{v\};$ 
50 result() begin
51   if  $(est[M+1][i] = \{v\})$  then return  $v$ ;
52   else if  $(r \geq M \wedge infoResult() \neq \emptyset)$  then return  $\Psi$ ;
53   else return  $\perp$ ;
54 macros:  $binValues(r, x)$  begin
55   return  $\{y \in \{0, 1\} : \exists s \subseteq \mathcal{P} : |\{p_j \in s : y \in est[r][j]\}| \geq x\}$ 
56 infoResult() begin
57   if  $(\exists s \subseteq \mathcal{P} : n-t \leq |s| \wedge (\forall p_j \in s : aux[r][j] \in binValues(r, 2t+1)))$  then return
58      $\{aux[r][j]\}_{p_j \in s};$ 
59   else return  $\emptyset;$ 
60 functions:  $decide(x)$  begin
61   foreach  $r' \in \{r, \dots, M+1\}$  do
62     if  $(est[r'][i] = \emptyset \vee aux[r'][i] = \perp)$  then
63        $(est[r'][i], aux[r'][i]) \leftarrow (\{x\}, x)$ 
64    $r \leftarrow M+1;$ 
65 tryToDecide(values) begin
66   if  $(values \neq \{v\})$  then  $est[r][i] \leftarrow \{\mathbf{randomBit}(r)\};$ 
67   else  $\{est[r][i] \leftarrow \{v\};$  if  $(v = \mathbf{randomBit}(r))$  then  $decide(v)\};$ 

```

4.2 A recyclable variation on Algorithm 3

Algorithm 4 presents a recyclable variation on Algorithm 3 that is needed for allowing the system to sequentially instantiate and recycle an unbounded number of Algorithm 3's objects using an SSBFT recycling mechanism, which we propose in Section 5. The boxed code lines highlight the modified code lines with respect to the code of Algorithm 3. Also, as before, the line numbers of the latter continue the one of the former. We clarify that the correctness proof (Section 4.3) focuses on Algorithm 3 rather than the straightforward added details of Algorithm 4.

Algorithm 4 uses the array $delivered[\mathcal{P}]$ (initialized to $[\mathbf{False}, \dots, \mathbf{False}]$) 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 a non- \perp value is returned by **result()**, see lines 89 and 90. Algorithm 4 updates

Part B of Algorithm 3: SSBFT MMR, code for p_i .

```

67 do forever begin
68   if  $((r, est, aux) \neq initState)$  then
69      $r \leftarrow \min\{r+1, M \bmod +1\}$ ;
70     repeat
71       if  $(est[0][i] \neq \{v\})$  then  $est[0][i] \leftarrow \{w\} : \exists w \in est[0][i]$ ;
72       foreach  $r' \in \{1, \dots, r-1\} : est[r'][i] = \emptyset \vee aux[r'][i] = \perp$  do
73          $(est[r'][i], aux[r'][i]) \leftarrow (est[0][i], x) : x \in est[0][i]$ ;
74       if  $((\exists w \in binValues(r, 2t+1) \wedge (aux[r][i] = \perp \vee aux[r][i] \notin$ 
75          $binValues(r, 2t+1)))$  then
76          $aux[r][i] \leftarrow w$ ;
77       foreach  $p_j \in \mathcal{P}$  do send  $EST(True, r,$ 
78          $est[r-1][i] \cup binValues(r, t+1), aux[r][i])$  to  $p_j$ ;
79       until  $infoResult() \neq \emptyset$ ;
80       tryToDecide( $infoResult()$ );
81       if  $(\exists w \in binValues(M+1, t+1))$  then  $decide(w)$ ;
82 upon  $EST(aJ, rJ, vJ, uJ)$  arrival from  $p_j$  begin
83    $est[rJ][j] \leftarrow est[rJ][j] \cup vJ$ ;  $aux[rJ][j] \leftarrow uJ$ ;
84   if  $aJ$  then send  $EST(False, rJ, est[rJ-1][i], aux[r][i])$  to  $p_j$ ;

```

$delivered[j]$ according to the arriving values from p_j (lines 112 and 116). The interface function `wasDelivered()` (line 85) returns 1 whenever there is a set of at least $n - t$ entries with the value `True` in $delivered[]$. The interface function `recycle()` (line 86) allows the node to restart its local state w.r.t. Algorithm 4.

The approach studied here considers the instantiation of one object at a time. A straightforward extension is to allow the allocation and recycling of a set of objects. Specifically, one can run δ concurrent MMR instances, where δ is a parameter for balancing the trade-off between fault recovery time and the number of MMR instances that can be used (before the next δ concurrent instances can start).

4.3 Correctness

The correctness proof shows that the solution presented in Section 4 recovers from transient-faults without blocking (Section 4.3.1) and that any consensus operation always satisfies the liveness requirements of Definition 1.1 (Section 4.3.2). Also, it satisfies the safety requirements of Definition 1.1 in the way that loosely-self-stabilizing systems do (Section 4.3.2), *i.e.*, any consensus operation satisfies the requirements of Definition 1.1 with probability $\Pr(r) = 1 - (1/2)^{-M}$.

Part A of Algorithm 4: A recyclable variation of Algorithm 3; code for p_i .

```

83 constants:
84  $initState := (0, [[\emptyset, \dots, \emptyset], \dots, [\emptyset, \dots, \emptyset]], [[\perp, \dots, \perp], \dots, [\perp, \dots, \perp]],$ 
    $[\text{False}, \dots, \text{False}]);$ 
85 provided interfaces:
    $\text{wasDelivered() do \{if } \exists S \subseteq \mathcal{P} : n-t \leq |S| : \forall p_k \in S : \exists r' \in \{0,$ 
    $\dots, r\} : delivered[k] = \text{True then return 1 else return 0;\}}$ 
86  $\text{recycle() do } (r, est, aux, delivered) \leftarrow initState;$ 
87 operations:  $\text{propose}(v) \text{ do } \{\text{recycle()}; est[0][i] \leftarrow \{v\}\};$ 
88 result() do begin
89    $\text{if } (est[M+1][i] = \{v\}) \text{ then } \{\text{delivered}[i] \leftarrow \text{True}; \text{return } v\};$ 
90    $\text{else if } (r \geq M \wedge \text{infoResult() } \neq \emptyset) \text{ then } \{\text{delivered}[i] \leftarrow \text{True}; \text{return } \Psi\};$ 
91    $\text{else return } \perp;$ 
92 macros:  $\text{binValues}(r, x) \text{ return}$ 
    $\{y \in \{0, 1\} : \exists s \subseteq \mathcal{P} : |\{p_j \in s : y \in est[r][j]\}| \geq x\};$ 
93 infoResult() begin
94    $\text{if } (\exists s \subseteq \mathcal{P} : n-t \leq |s| \wedge (\forall p_j \in s : aux[r][j] \in \text{binValues}(r, 2t+1))) \text{ then}$ 
95    $\quad \text{return } \{aux[r][j]\}_{p_j \in s} \text{ else return } \emptyset;$ 
96 functions:  $\text{decide}(x) \text{ begin}$ 
97    $\text{foreach } r' \in \{r, \dots, M+1\} \text{ do}$ 
98    $\quad \text{if } (est[r'][i] = \emptyset \vee aux[r'][i] = \perp) \text{ then } (est[r'][i], aux[r'][i]) \leftarrow (\{x\}, x);$ 
99 tryToDecide(values) begin
100    $\text{if } (values \neq \{v\}) \text{ then } est[r][i] \leftarrow \{\text{randomBit}(r)\};$ 
101    $\text{else } \{est[r][i] \leftarrow \{v\}; \text{if } (v = \text{randomBit}(r)) \text{ then decide}(v)\};$ 

```

4.3.1 Transient fault recovery

We say that a system state c is *resolved* if $\forall i \in \text{Correct} : |est_i[0][i]| \in \{0, 1\} \wedge \nexists r' \in \{1, \dots, r-1\} : est_i[r'][i] = \emptyset \vee aux_i[r'][i] = \perp$ and no communication channel that goes out from $p_i \in \mathcal{P} : i \in \text{Correct}$ to any other correct node includes $\text{EST}(rnd = r, est = W, aux = w) : r_i < r \vee W \not\subseteq est_i[r][i] \vee (w \neq \perp \wedge w \notin W)$ messages. Suppose that during execution R , every correct node $p_i \in \mathcal{P}$ invokes $\text{propose}_i()$ exactly once. In this case, we say that R includes a *complete invocation* of binary consensus. Theorem 4.1 shows recovery to resolved system states and termination during executions that include a complete invocation of binary consensus. The statement of Theorem 4.1 uses the term *active* for node $p_i \in \mathcal{P}$

Part B of Algorithm 4: A recyclable variation of Algorithm 3; code for p_i .

```

102 do forever begin
103   if  $((r, est, aux) \neq initState)$  then
104      $r \leftarrow \min\{r+1, M\}$ ;
105     repeat
106       if  $(est[0][i] \neq \{v\})$  then  $est[0][i] \leftarrow \{w\} : \exists w \in est[0][i]$ ;
107       foreach  $r' \in \{1, \dots, r-1\} : est[r'][i] = \emptyset \vee aux[r'][i] = \perp$  do
108          $(est[r'][i], aux[r'][i]) \leftarrow (est[0][i], x) : x \in est[0][i]$ ;
109       if  $((\exists w \in binValues(r, 2t+1) \wedge (aux[r][i] = \perp \vee aux[r][i] \notin$ 
110          $binValues(r, 2t+1)))$  then
111          $aux[r][i] \leftarrow w$ ;
112       foreach  $p_j \in \mathcal{P}$  do
113         send
114            $EST(True, r, est[r-1][i] \cup binValues(r, t+1), aux[r][i], \boxed{delivered[i]})$  to
115              $p_j$ 
116       until  $infoResult() \neq \emptyset$ ;
117       tryToDecide( $infoResult()$ );
118 upon  $EST(aJ, rJ, vJ, uJ, \boxed{deliveredJ})$  arrival from  $p_j$  begin
119    $delivered[i] \leftarrow delivered[i] \vee deliveredJ$ ;
120    $est[rJ][j] \leftarrow est[rJ][j] \cup vJ$ ;  $aux[rJ][j] \leftarrow uJ$ ;
121   if  $aJ$  then send  $EST(False, rJ, est[rJ-1][i], aux[r][i])$  to  $p_j$ ;

```

when referring to the case of $est_i[0][i] \neq initState$.

Theorem 4.1 (Convergence) *Let R be an execution of Algorithm 3. (i) Within one complete asynchronous (communication) round, the system reaches a resolved state. Moreover, suppose that throughout R all correct nodes are active. (ii) Within $\mathcal{O}(M)$ asynchronous (communication) rounds, for every correct node $p_i \in P$, it holds that the operation $result_i()$ returns $v \in \{0, 1, \Psi\}$, where Ψ is the transient error symbol.*

Proof of Theorem 4.1 Lemmas 4.2 and 4.4 demonstrate the theorem.

Lemma 4.2 *Invariant (i) holds.*

Proof of Lemma 4.2 Let m be a message that in R 's starting system state resides in the communication channels between any pair of correct nodes. By Remark 2.1, within $\mathcal{O}(1)$ asynchronous rounds, the system reaches a state in which m does not appear. Let us look at p_i 's first complete iteration of the do-forever loop (lines 68 to 78) after m has left the system. Once that happens, for any message $EST(rnd = r, est = W, aux = w)$ that appears

in any communication channel that is going out from $p_i \in \mathcal{P} : i \in \text{Correct}$, it holds that $r \leq r_i \wedge W \subseteq \text{est}_i[r][i] \wedge (w = \perp \vee w \in W)$ (due to lines 48 and 76).

Let $I_{i,r}$ be p_i 's first complete iteration in the first complete asynchronous (communication) round of R . Suppose that in the iteration's first system state, it holds that $\forall i \in \text{Correct} : (r_i, \text{est}_i, \text{aux}_i) = \text{initState}$. In this case, Invariant (i) holds by definition. In case $\forall i \in \text{Correct} : (r_i, \text{est}_i, \text{aux}_i) = \text{initState}$ does not hold, lines 71 to 73 imply that Invariant (i) holds. Invariant (i) also holds when the round number r is incremented. Note that regardless of which branch of the if-statement in line 65 node p_i follows, $\text{est}_i[r][i]$ is always assigned a value that is not the empty set at the end of round r , cf. lines 65 and 66. Moreover, the assignment of w to $\text{aux}_i[r][i]$ in line 75 is always of a value that is not the empty set due to the if-statement condition in line 74 and the definition of $\text{binValues}()$ (line 55). $\square_{\text{Lemma 4.2}}$

Lemma 4.3 is needed for the proof of Lemma 4.4.

Lemma 4.3 *Suppose that R 's states are resolved (Lemma 4.2). The repeat-until loop in lines 74 to 77 cannot block forever.*

Proof of Lemma 4.3 The proof is by contradiction; to prove the lemma to be true, we begin by assuming it is false and show that this leads to a contradiction, which implies that the lemma holds. Argument 5 shows the needed contradiction and it uses arguments 1 to 4.

Argument 1: *Eventually $\text{aux}_i[r][i] \in \text{binValues}_i(r, 2t+1)$ holds.* Suppose that in R 's starting state, $(\text{aux}[r][i] \neq \perp \wedge \text{aux}[r][i] \in \text{binValues}(r, 2t+1))$ does not hold, because otherwise the proof of the argument is done. There are at least $n-t \geq 2t+1 = (t+1)+t$ correct nodes and each of them sends $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{w, \bullet\}, \bullet) : w \in \{0, 1\}$ messages to all nodes (line 76). Therefore, we know that there is $v \in \{0, 1\}$, such that at least $(t+1)$ correct nodes send $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v, \bullet\}, \bullet)$ messages to all other nodes.

Since every correct node receives $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v, \bullet\}, \bullet)$ from at least $(t+1)$ nodes (line 82), we know that eventually every correct node relays the value v via the message $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v, \bullet\}, \bullet)$ that line 76 sends due to the fact that $v \in \text{binValues}_i(r, t+1)$.

Since $n-t \geq 2t+1$ holds, the clause $(\exists w \in \text{binValues}(r, 2t+1))$ in the if-statement condition at line 74 is eventually satisfied at each correct node $p_i \in \mathcal{P}$. Thus, if $(\text{aux}[r][i] = \perp \vee \text{aux}[r][i] \notin \text{binValues}(r, 2t+1))$ does not hold, line 75 makes sure it does.

Argument 2: *Eventually the system reaches a state in which $\exists i \in \text{Correct} : w \in \text{binValues}_i(r_i, 2t+1) \implies \exists s \subseteq \text{Correct} : t+1 \leq |s| \wedge \forall k \in s : w \in \text{est}_k[k]$.*

We prove the argument by contradiction; we begin by assuming the argument is false and show that this leads to a contradiction, which implies that the argument holds. Specifically, suppose that $\exists i \in \text{Correct} : w \in \text{binValues}_i(r_i, 2t+1)$ holds in every system state in R and yet $\forall s \subseteq \text{Correct} : t+1 \leq |s|$, it is true that $\exists k \in s : w \notin \text{est}_k[k]$.

By lines 76 and 81, the only way in which $w \in \text{binValues}_i(r_i, 2t+1)$ hold in every system state $c' \in R$, is if there is a system state c that appears in R before c' , such that $\exists s \subseteq \text{Correct} : t+1 \leq |s| : \forall k \in s : w \in \text{est}_k[k]$. Thus, a contradiction is reached (with respect to the assumption made at the start of this argument's proof), which implies that the argument is true.

Argument 3: *Eventually the system reaches a state $c' \in R$ in which $\exists s \subseteq \text{Correct} : t+1 \leq |s| \wedge \forall p_k \in s : w \in \text{est}_k[k] \implies \forall i \in \text{Correct} : w \in \text{binValues}_i(r_i, 2t+1)$.*

By line 76 and the argument's assumption, there are at least $(t+1)$ correct nodes that send $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{w, \bullet\}, \bullet)$ messages to all (correct) nodes. Since every correct node receives w from at least $(t+1)$ nodes (line 81), every correct node eventually reply w via the message $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{w, \bullet\}, \bullet)$ at lines 76 and 82 due to the fact that $w \in \text{binValues}_i(r, t+1)$. Since $n-t \geq 2t+1$ holds, we know that $(\exists w \in \text{binValues}(r, 2t+1))$ holds and the argument is true.

Argument 4: *Suppose that the condition $\text{cond}(i) := \text{infoResult}_i() \neq \emptyset : i \in \text{Correct}$ does not hold in R 's starting system state. Eventually, the system reaches a state $c'' \in R$, in which $\text{cond}(i) : i \in \text{Correct}$ holds.*

We prove the argument by contradiction; we begin by assuming the argument is false and show that this leads to a contradiction, which implies that the argument holds. Specifically, suppose that $\text{cond}(i)$ never holds, i.e., $c'' \in R$ does not exist. We note that $\text{cond}(i)$ must hold if $\text{binValues}_i(r_i, 2t+1) = \{0, 1\}$. The same can be said for the case of $\text{binValues}_i(r_i, 2t+1) = \{v\} \wedge \exists s \subseteq \mathcal{P} : n-t \leq |s| \wedge (\cup_{p_k \in s} \{aux_i[r][k]\}) = \{w\} \wedge w = v$. Therefore, we assume that, for any system state, it holds that $\text{binValues}_i(r_i, 2t+1) = \{v\} \subsetneq \{0, 1\}$ and $\forall s \subseteq \mathcal{P} : n-t \leq |s| \implies w \in (\cup_{p_k \in s} \{aux_i[r][k]\}) : w \neq v$. We demonstrate a contradiction by showing that eventually $w \in \text{binValues}_i(r_i, 2t+1)$.

By lines 76 and 81, the only way in which $w \in (\cup_{p_k \in s} \{aux_i[r][k]\})$ holds in every system state $c' \in R$, is if there is a system state c that appears in R before c' , such that $\exists p_k \in \mathcal{P} : aux_k[r][k] = w$. Note that c' and c can be selected such that the following sequence of statements are true. By Argument 1, $aux_k[r][k] \in \text{binValues}_k(r, 2t+1)$. By Argument 2, $w \in \text{binValues}_k(r_i, 2t+1) \implies \exists s \subseteq \text{Correct} : t+1 \leq |s| \wedge \forall p_k \in s : w \in \text{est}_k[k]$ in c . By Argument 3, $\exists s \subseteq \text{Correct} : t+1 \leq |s| \wedge \forall k \in s : w \in \text{est}_k[k] \implies \forall i \in \text{Correct} : w \in \text{binValues}_i(r_i, 2t+1)$ in c . Thus, a contradiction is reached (with respect to the assumption made at the start of this argument's proof), which implies that the argument is true.

Argument 5: *The lemma is true.* Argument 4 implies that a contradiction (with respect to the assumption made in the start of this lemma's proof) was reached since the exist condition in line 77 eventually holds. $\square_{\text{Lemma 4.3}}$

Lemma 4.4 *Invariant (ii) holds.*

Proof of Lemma 4.4 Lemma 4.2 shows that R 's system states are resolved. Lemma 4.3 says that the repeat-until loop in lines 74 to 77 does not block. By line 69 and the definition of an asynchronous (communication) round (Section 2.4), every iteration of the do-forever loop (lines 68 to 78) can be associated with at most one asynchronous (communication) round. Thus, line 50 and Argument (4) of the proof of Lemma 4.3 imply that $(r_i \geq M \wedge \text{infoResult}_i() \neq \emptyset)$ holds within $\mathcal{O}(M)$ asynchronous (communication) rounds. Therefore, $\text{result}_i()$ returns a non- \perp value within $\mathcal{O}(M)$ asynchronous (communication) rounds. $\square_{\text{Lemma 4.4}}$ $\square_{\text{Theorem 4.1}}$

4.3.2 Satisfying the task specifications

We say that the system state c is *well-initialized* if $\forall i \in \text{Correct} : (r_i, \text{est}_i, \text{aux}_i) := \text{initState}$ holds and no communication channel between two correct nodes includes $\text{EST}()$ messages. Note that a well-initialized system state is also a resolved one (Section 4.3.1). Theorem 4.6 shows that Algorithm 3 satisfies the requirements of Definition 1.1 during legal executions that start from a well-initialized system state and have a complete invocation of binary consensus. The proof of Theorem 4.6 uses Theorem 4.5, which demonstrates that Algorithm 3 satisfies the requirements of Definition 4.1, which adds more details to the one given in Section 3.1.1. Recall that the operation $\text{bvBroadcast}(v)$ of Algorithm 1 is embedded in the code of Algorithm 3.

Definition 4.1 (BV-broadcast) *Let $p_i \in \mathcal{P}$, $r \in \{1, \dots, M\}$, and $v \in \{0, 1\}$. Suppose that $r_i = r \wedge \text{est}_i[r-1][i] = \{v\}$ holds immediately before p_i executes line 76. In this case, we say that p_i BV-broadcast v during round r in line 76. Let $c \in R$ and suppose that $w \in \text{binValues}_i(r, t+1)$ holds in c (for the first time). In this case, we say that p_i BV-delivers w during round r .*

- **BV-validity.** *Suppose that p_i is correct and $v \in \text{binValues}_i(r, t+1)$ holds in system state $c \in R$. Then, before c there is a step in R in which a correct node BV-broadcast v .*
- **BV-uniformity.** *Suppose that p_i is correct and $v \in \text{binValues}_i(r, t+1)$ holds in system state $c \in R$. Then, eventually, the system reaches a state in which $\forall j \in \text{Correct} : v \in \text{binValues}_j(r, t+1)$ holds.*
- **BV-completion.** *Eventually, the system reaches a state in which $\forall i \in \text{Correct} : \text{binValues}_i(r, t+1) \neq \emptyset$ holds.*

Theorem 4.5 (BV-broadcast) *Let R be an execution of Algorithm 3 that starts from a well-initialized system state and includes a complete invocation of binary consensus. Lines 74 to 77 and lines 80 to 82 of Algorithm 3 implement the BV-broadcast task (Definition 4.1).*

Proof of Theorem 4.5 We prove that the requirements of Definition 4.1 hold.

BV-validity. Suppose that, during round r , merely faulty nodes BV-broadcast v . We show that $\nexists c \in R$, such that $(\exists v \in \text{binValues}_i(r, 2t+1))$ holds in c . Since only faulty nodes BV-broadcast v , then no correct node receives $\text{EST}(-, \text{rnd} = r, \text{est} = \{v, \bullet\}, \bullet)$ messages from more than t different senders. Consequently, $v \notin \text{binValues}_i(r, t+1)$ in line 76 at any correct node $p_i \in \mathcal{P}$. Similarly, no correct node $p_i \in \mathcal{P}$ can satisfy the predicate $(\exists w \in \text{binValues}(r, 2t+1))$ at line 77 (via line 56). Thus, the requirement holds.

BV-uniformity. Suppose that $w \in \text{binValues}_i(r, 2t+1)$ holds in c . By lines 55 and 74 we know that p_i stores v in at least $(2t+1)$ entries of $\text{EST}[r][\]$. Since R starts in a well-initialized system state, this can only happen if p_i received $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v, \bullet\}, \bullet)$ messages from at least $(2t+1)$ different nodes (line 80). This means that p_i received this

message from at least $(t+1)$ different correct nodes. Since each of these correct nodes sent the message $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v, \bullet\}, \bullet)$ to any node in \mathcal{P} , we know that $\forall j \in \text{Correct} : \text{binValues}_j(r, t+1) \neq \emptyset$ (line 76) holds eventually. Therefore, every correct node p_j sends $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v, \bullet\}, \bullet) : v \in \text{binValues}_i(r, t+1)$ to all. Since $n-t \geq 2t+1$, we know that $(\exists w \in \text{binValues}_k(r, 2t+1))$ holds eventually at each correct node, $p_k \in \mathcal{P}$.

BV-completion. This requirement is implied by Lemma 4.3. $\square_{\text{Theorem 4.5}}$

Theorem 4.6 (Closure) *Let R be an execution of Algorithm 3 that starts from a well-initialized system state and includes a complete invocation of binary consensus. Within $\mathcal{O}(r) : r \leq M$ asynchronous (communication) rounds, with probability $\Pr(r) = 1 - (1/2)^r$, and for each correct node $p_i \in \mathcal{P}$, the operation $\text{result}_i()$ returns $v \in \{0, 1\}$.*

Proof of Theorem 4.6 Lemmas 4.7 to 4.11 show the proof. Lemma 4.7 shows that once all correct nodes estimate the same value in a round r , they hold on this estimate in all subsequent rounds. Lemma 4.8 shows that correct nodes that pass a singleton to $\text{tryToDecide}()$, pass the same set. Lemma 4.9 shows that correct nodes can only decide a value that has been previously proposed by a correct node. Lemma 4.11 shows that correct nodes have $v \in \{0, 1\}$ as a return value from $\text{result}()$. This occurs by round $r \leq M$ with the probability of $1 - (1/2)^r$. Putting these together, we obtain the proof of Theorem 4.6.

Lemma 4.7 *Suppose that every correct node, $p_i \in \mathcal{P}$, estimates value v upon entering round r , i.e., $\text{est}_i[r-1][i] = \{v\} \wedge r_i = r-1$ immediately before executing line 68. Then, p_i estimates the value v in any round later than r , i.e., $r' \in \{r, \dots, M\} : \text{est}_i[r'][i] = \{v\}$.*

Proof of Lemma 4.7 There are $n-t > t+1$ correct nodes. By the lemma statement, all of them broadcast $\text{EST}(\bullet, \text{rnd} = r, \text{est} = \{v\}, \bullet)$ (line 76). Thus, $\text{binValues}_i(r, 2t+1) = \{v\}$ (BV-completion and BV-validity, Theorem 4.5) and $\text{values}_i^r = \{v\}$ (lines 65), where values_i^r is the parameter that p_i passes to $\text{tryToDecide}()$ (line 64) during round r , cf. $\text{values}_i^r = \cup_{j \in s} \{aux[r][j]\}$ (line 78 via line 56).

Therefore, $\text{est}_i[r][i] = \{v\}$ holds due to the assignment in the start of line 66. Since there are most t Byzantine nodes, and for an estimate to be forwarded (and hence accepted) it needs a “support” of $t+1$ nodes (line 76), it follows that the correct nodes cannot change their estimate in any round $r' \geq r$. $\square_{\text{Lemma 4.7}}$

Lemma 4.8 *Suppose that there is a system state $c \in R$, such that $(\text{values}_i^r = \{v\}) \wedge (\text{values}_j^r = \{w\})$, where $p_i, p_j \in \mathcal{P}$ are two correct nodes and values_i^r is the parameter that p_i passes to $\text{tryToDecide}()$ (line 64) during round r . It holds that $v = w$ in c .*

Proof of Lemma 4.8 Due to the exit condition of the repeat-until loop in lines 74 to 77, p_i had to receive before c identical $\text{EST}(\bullet, \text{rnd} = r, \bullet, aux = v, \bullet)$ messages from at least $(n-t)$ different nodes. Since at most t nodes are faulty, $(n-t) = (n-2t)$, which means that p_i received $\text{EST}(\bullet, \text{rnd} = r, \bullet, aux = v, \bullet)$ messages before c from at least $(t+1)$ different correct nodes, as $n-2t \geq t+1$. Using the symmetrical arguments, we know that p_j had

to receive before c identical $\text{EST}(\bullet, \text{rnd} = r, \bullet, \text{aux} = w, \bullet)$ messages from at least $(n-t)$ different nodes.

Since $(n-t)+(t+1) > n$, the pigeonhole principle implies the existence for at least one correct node, $p_x \in \mathcal{P}$, from which from p_i and p_j have received the messages $\text{EST}(\bullet, \text{rnd} = r, \bullet, \text{aux} = v, \bullet)$ and $\text{EST}(\bullet, \text{rnd} = r, \bullet, \text{aux} = w, \bullet)$, respectively. The fact that p_x is correct implies that it has sent the same $\text{EST}(\bullet, \text{rnd} = r, \bullet)$ message to all the nodes in line 76. Thus $v = w$. $\square_{\text{Lemma 4.8}}$

Lemma 4.9 *Suppose that there is a system state $c \in R$, such that $\text{result}_i() = v \in \{0, 1\}$ in c , where $p_i \in \mathcal{P}$ is a correct node. There is a correct node $p_j \in \mathcal{P}$ and a step $a_j \in R$ (between R 's starting system state and c) in which p_j invokes $\text{propose}_j(v)$.*

Proof of Lemma 4.9 Suppose that $r_i = 1$. Recall (a) the BV-validity property (Theorem 4.5 and line 76), observe (b) the if-statement condition in line 74, which selects the value $w_i \in \text{binValues}(1, 2t+1)$ (line 75) as well as the exit condition in line 77 of the repeat-until loop in lines 74 to 77 in which (c) correct nodes, $p_j \in \mathcal{P}$, broadcast $\text{EST}(\bullet, \text{rnd} = 1, \bullet, \text{aux} = w_j, \bullet) : w_j \in \text{binValues}(1, t+1)$ messages. Thus, the set values_i^1 includes only values arriving from correct nodes, where values_i^r is the parameter that p_i passes to $\text{tryToDecide}()$ (line 64) during round r .

Node p_i can decide v (line 66) when $\text{values}_i^1 = \{v\} \wedge v = \text{randomBit}_i(r_i)$ holds. Regardless of the decision, p_i updates its new estimate (line 66). Processor p_i updates its estimate $\text{est}_i[r_i][i]$ with the value, $\text{randomBit}(r)$, obtained by the RCC (line 65) whenever $\text{values}_i^1 = \{0, 1\}$. This means, that p_i updates the estimated value with a value that a correct node has proposed. Note that the $\text{values}_i^1 = \{0, 1\}$ case occurs when both 0 and 1 were proposed by correct nodes. The same arguments hold also for round numbers $r > 1$, and therefore, a decided value must be a value proposed earlier by a correct node p_j , where $i = j$ can possibly hold. $\square_{\text{Lemma 4.9}}$

Lemma 4.10 *Suppose that there is system state $c \in R$, such that $\text{result}_i()$ and $\text{result}_j()$ are not members of $\{\perp, \Psi\}$ holds in c , where $p_i, p_j \in \mathcal{P}$ are correct nodes. It holds that $\text{result}_i() = \text{result}_j()$.*

Proof of Lemma 4.10 Suppose, without the loss of generality, that node p_i is the first correct node that decides during R and it does so during round r . Suppose that there is another node, p_j , that decides also at round r . We know that both p_i and p_j decide the same value due to the $v_i = \text{randomBit}_i(r)$ condition of the if-statement in line 66 and the properties of the RCC. We also know that p_i and p_j update their estimates in $\text{est}_x[r][x] : x \in \{i, j\}$ to $\text{randomBit}_x(r)$.

Recall that values_i^r denotes the parameter that p_i passes to $\text{tryToDecide}()$ (line 65) during round r . Lemma 4.8 says that $(\text{values}_i^r = \{v\}) \wedge (\text{values}_j^r = \{w\})$ means that $v \neq w$ cannot hold. Moreover, if p_i decides during round r and p_j is not ready to decide, it must be the case that $\text{values}_j^r = \{v, w\} = \{0, 1\}$, see lines 65 to 66 and the proof of Lemma 4.8. Therefore, p_j assigns $\text{randomBit}(r)$ to $\text{est}_j[r][j]$ (line 65). This means that every correct node starts round $(r+1)$ with $\text{est}_j[r][j] = \text{randomBit}(r)$ and $\text{randomBit}(r) = v$. Lemma 4.7 says that this estimate never change, and thus, only v can be decided. $\square_{\text{Lemma 4.10}}$

Lemma 4.11 *By the end of round $r \leq M$, for each correct node $p_i \in \mathcal{P}$, the operation $\text{result}_i()$ returns $v \in \{0, 1\}$ with probability $\Pr(r) = 1 - (1/2)^r$.*

Proof of Lemma 4.11 The proof uses Claim 4.12.

Claim 4.12 *Let $c_r \in R$ be the state that the system reaches at the end of round $r \leq M$. With probability $\Pr(r) = 1 - (1/2)^r$, $\exists v \in \{0, 1\} : \forall i \in \text{Correct} : \text{est}_i[r][i] = \{v\}$ holds in c_r .*

Proof of Claim 4.12 Let values_i^r be the parameter that p_i passes to $\text{tryToDecide}()$ (line 64) on round r .

- **Case 1:** Suppose that the if-statement condition $\text{values}_i^r = \{v_k(r)\}$ (line 65) holds for all correct nodes $p_k \in \mathcal{P}$. Similarly to the proof of Lemma 4.10, any correct node p_k assigns to $\text{est}_k[r][k]$ the same value, $v_k(r)$ (line 29).
- **Case 2:** Suppose that the if-statement condition $\text{values}_i^r = \{v_k(r)\}$ (line 65) does not hold for all correct nodes $p_k \in \mathcal{P}$. By similar arguments as in the previous case, any correct p_k assigns to $\text{est}_k[r][k]$ the same value, $\{\text{randomBit}_k(r)\}$ (line 28).
- **Case 3:** Some correct nodes assign $\{v_k(r)\}$ to $\text{est}_k[r][k]$ (line 66), whereas others assign $\{\text{randomBit}_k(r)\}$ (line 65).

The rest of the proof focuses on Case 3. Recall the assumption that the Byzantine nodes have no control over the network or its scheduler. Thus, the values $\text{randomBit}_k(r)$ and $\text{randomBit}_k(r')$ are independent (due to the RCC properties, see Section 1.7.2), where $r \neq r'$. Therefore, there is probability of $\frac{1}{2}$ that the assignments of the values $\{v_k(r)\}$ and $\{\text{randomBit}_k(r)\}$ are equal. Let $\Pr(r)$ be the probability that $[\exists r' \leq r : \text{randomBit}(r) = v(r)]$. Then, $\Pr(r) = \frac{1}{2} + (1 - \frac{1}{2})\frac{1}{2} + \dots + (1 - \frac{1}{2})^{r-1}\frac{1}{2} = 1 - (\frac{1}{2})^r$. $\square_{\text{Claim 4.12}}$

Recall that Lemma 4.3 says that the repeat-until loop in lines 74 to 77 cannot block forever. It follows from Lemma 4.7 and Claim 4.12 that all the correct nodes p_i keep their estimated value $\text{est}_i = v$ and consequently the predicate $(\text{values}_i^{r'} = \{v\})$ at line 65 holds for round r' , where $\text{values}_i^{r'} = \cup_{j \in s} \{\text{aux}_i[r][j]\}$. With probability $\Pr(r) = 1 - (1/2)^r$, by round r , it holds that $\text{randomBit}(r) = v$ due to the RCC properties. Then, the if-statement condition of line 65 does not hold and the one in line 66 does hold. Thus, all the correct nodes decide v . $\square_{\text{Lemma 4.11}}$ $\square_{\text{Theorem 4.6}}$

We conclude the proof by showing that Algorithm 3 is an eventually loosely-self-stabilizing solution for binary consensus.

Theorem 4.13 *Let R be an execution of Algorithm 3 that starts in a well-initialized system state and during which every correct node $p_i \in \mathcal{P}$ invokes $\text{propose}_i()$ exactly once. Execution R implements a loosely-self-stabilizing and randomized solution for binary consensus that can tolerate up to t Byzantine nodes, where $n \geq 3t + 1$. Moreover, within four asynchronous (communication) rounds, all correct nodes are expected to decide.*

Proof of Theorem 4.13 We divide the proof into four arguments.

Argument 1: *BC-completion is always guaranteed.* Lemma 4.3 and 4.11 demonstrate BC-completion when starting from an arbitrary, and resp., a well-initialized system state.

Argument 2: *Suppose that, for every correct node $p_i \in \mathcal{P}$, operation $\text{result}_i()$ returns $v \in \{0, 1\}$. A complete and well-initialized invocation of binary consensus satisfies the safety requirements of Definition 1.1.* Lemmas 4.9, 4.10, and 4.11 imply BC-validity and BC-agreement as long as $\forall i \in \text{Correct} : \text{result}_i()$ returns $v \in \{0, 1\}$.

Argument 3: *Algorithm 3 satisfies the design criteria of Definition 2.2.* By Theorem 4.1, we know that any complete invocation of binary consensus terminates within a finite number of steps. Once that happens, the next well-initialized invocation of $\text{propose}()$ can succeed independently of previous invocations. Argument 1 and Lemma 4.11 imply that with probability $\Pr(M) = 1 - (1/2)^M$, a complete and well-initialized invocation of binary consensus satisfies the requirements of Definition 1.1.

Argument 4: *All correct nodes are expected to decide within four iterations of Algorithm 3.* The proof of Claim 4.12 considers two stages when demonstrating BC-completion (after starting from a well-initialized system state). That is, all correct nodes need to first use the same value, v , as their estimated one, see the assignment to $\text{est}_i[r][i]$ in lines 66 to 65. Then, each correct node waits until the next round in which the condition, $v_i = \text{randomBit}_i(r_i)$, of the if-statesmen in line 66 holds, where $\text{randomBit}()$ is the interface to the RCC. The rest of the proof is implied via the linearity of expectation and the following arguments regarding the expectation of each stage.

Stage I. The proof of Claim 4.12 reveals the case in which not all correct nodes use the same value (Case 3). This is when the condition, $\text{values} = \{v\}$, of the if-statement in line 66 is true but not for any correct node $p_i \in \mathcal{P}$. We show how to bound by two the number of asynchronous rounds in which this situation can happen. Suppose that $\text{values}_i^r \neq \{v\}$. Note that, with probability $1/2$, the assignment in line 65 sets the value $\{v\}$ to $\text{est}_i[r][i]$. Once that happens, Stage I is finished and Stage II begins. If this does not happen, with probability $1/2$, Stage I needs to be repeated and so does the above arguments. Thus, within two rounds, Stage I is expected to end.

Stage II. By the RCC properties (Section 1.7.2), we know that $\Pr(v_i = \text{randomBit}_i(r_i)) = 1/2$ and $E(\Pr(v_i = \text{randomBit}_i(r_i))) = 2$. $\square_{\text{Theorem 4.13}}$

5 SSBFT Recycling Mechanism for BSMP_{n,t}[κ —SGC, $t < n/3$, RCCs]

We present a SSBFT recycling mechanism that uses a bounded array of recyclable objects. These objects, for example, can be instances of recyclable objects based on Algorithm 3 (with the boxed code lines), which implements the operations $\text{propose}()$ and $\text{result}()$ as well as $\text{wasDelivered}()$ and $\text{recycle}()$.

The mechanism aims at making sure that, at any time, there is 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.

In this section, we refine $\text{BAMP}_{n,t}[-\text{FC}, t < n/3]$ model into the model of $\text{BSMP}_{n,t}[\kappa\text{-SGC}, t < n/3, \text{RCCs}]$ (Section 5.1), which is a synchronous model enriched with a random RCC and κ -state clock. We then present the synchrony assumptions (Assumption 5.1) that we mentioned above and bring an overview of the proposed solution (Section 5.2) before providing the details and correctness proofs (sections 5.4 and 5.5).

5.1 System Settings for $\text{BSMP}_{n,t}[t < n/3, \text{RCCs}]$

We denote the $\text{BSMP}_{n,t}[t < n/3, \kappa\text{-SGC}]$ model, which stands for Byzantine synchronous message-passing with at most t (out of n) faulty nodes, and $t < n/3$. The $\text{BSMP}_{n,t}[\kappa\text{-SGC}, t < n/3, \text{RCCs}]$ model is defined by enriching the model of $\text{BAMP}_{n,t}[-\text{FC}, t < n/3]$ with a κ -state global clock (Section 5.1.1), reliable communications (Section 5.1.2), and RCCs (Section 1.7.2).

5.1.1 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. Also, $a_i[x]$ 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 local function $\text{clock}(\kappa)$, which returns an integer between 0 and $\kappa - 1$. Algorithm 3 of BDH [10] offers an SSBFT κ -state global clock.

5.1.2 Reliable communications

We assume the availability of reliable communications. We assume that any correct node $p_i \in \mathcal{P}$ starts any step $a_i[x]$ with receiving all pending messages from all nodes. Also, 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 correct node p_i sends during step $a_i[x]$ to another correct 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 before the end of $a_i[x]$.

5.2 Solution overview

The SSBFT recycling solution is a composition of several algorithms, see Figure 2. Our recycling mechanism is presented in Algorithm 5. It allows every correct node to retrieve

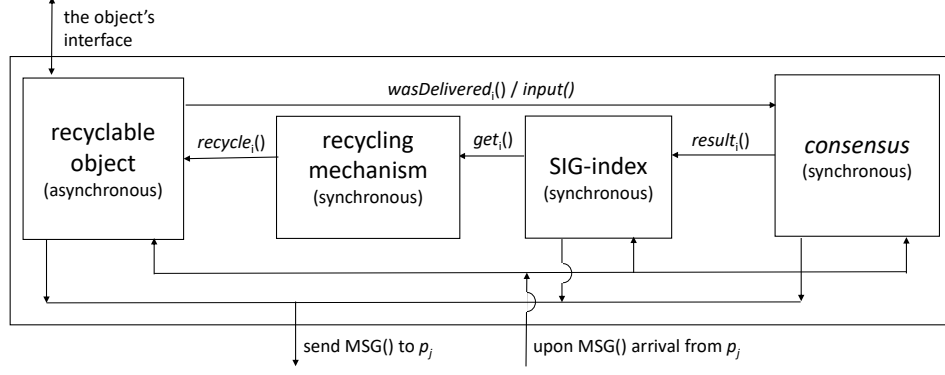


Figure 2: The proposed solution using a recyclable object (Algorithm 3), the recycling mechanism (Algorithm 5), a consensus protocol (Algorithm 6), and SIG-index (Algorithm 7).

Algorithm 5: SSBFT object recycling with a predefined log size; code for p_i

```

119 constants: indexNum number of indices of recyclable objects;
120 logSize  $\in \{0, \dots, indexNum - 2\}$  user-defined bound on the object log size;
121 variables:
122 ssbftIndex : an SSBFT index that marks the current object in use (Algorithm 7);
123 obj[indexNum] : array of recyclable objects, e.g., Algorithm 3. Note that during
    legal execution only at most  $(logSize + 1)$  objects are stores at any given point of
    time;
124 upon pulse /* signal from global pulse system */ begin
125   foreach  $x \notin \{y \bmod indexNum : y \in \{z - logSize, \dots, z\}\}$  where
     $z = indexNum + ssbftIndex.getIndex()$  do
126      $obj[x].recycle()$ 

```

at least once the result of any object that is stored in a bounded array and yet over time that array can store an unbounded number of object instances. Algorithm 5 guarantees that for every instance of the recyclable object, every correct node calls **result()** (line 85) at least once before every correct node simultaneously invokes **recycle()** (line 86).

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 legitimately recycled. Specifically, Assumption 5.1 considers an event that can be locally learned about when **wasDelivered()** returns '1' (line 85).

Assumption 5.1 (Result retrieval within a bounded time) *Let us consider the system*

Algorithm 6: SSBFT multivalued consensus in $\text{BSMP}_{n,t}[t < n/3, (t+1)\text{-SGC}]$;
code for node p_i

```

127 variables: currentResult stores the most recent result of co;
128 co a (non-self-stabilizing) BFT (multivalued) consensus object;
129 interface required:
130 input() : defines the input value to be provided to the given consensus protocol;
131 interface provided:
132 result() do return (currentResult) // the decided value of the most recent co's
    invocation;
133 message structure:  $\langle appMsg \rangle$ , where appMsg is the application message, i.e., a
    message sent by the given consensus protocol;
134 upon pulse /* signal from global pulse system */ begin
135   let M be message that holds at  $M[j]$  the arriving  $\langle appMsg_j \rangle$  messages from  $p_j$ 
    for the current synchronous round and  $M' = [\perp, \dots, \perp]$ ;
136   if  $clock(\kappa) = 0$  then
137     currentResult  $\leftarrow co.result()$ ;
138     co.restart();
139      $M' \leftarrow co.propose(input())$  /* for recycling input() is wasDelivered() */
140   else if  $clock(cycleSize) \in \{1, \dots, t\}$  then  $M' \leftarrow co.process(M)$ ;
141   foreach  $p_j \in \mathcal{P}$  do send  $\langle M'[j] \rangle$  to  $p_j$ ;

```

state, $c[r]$, in which the result of object obj was retrieved by at least $t + 1$ correct nodes. We assume that, 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 5's recycling guarantees are facilitated by Algorithm 6, which agrees on a single evidence from all collected ones, and Algorithm 7, which uses the agreed evidence for updating the value of the index that points to the current entry in the array. Algorithm 5's detailed presentation and correctness proof appear in Section 5.3.

5.2.1 Evidence collection using an SSBFT (multivalued) consensus (Algorithm 6)

Algorithm 6 offers an SSBFT multivalued consensus protocol that 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, we use *wasDelivered()* (line 85) for providing input to Algorithm 6. Thus, whenever '1' is decided, at least one correct node got 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

Algorithm 7: SSBFT SIG-index in $\text{BSMP}_{n,t}[t < n/3, 4\text{-SGC}]$; code for node p_i

```

142 constants:  $I$  : bound on the number of states an index may have;
143 variables:  $index \in \{0, \dots, I - 1\}$  : a local copy of the global logical object index;
144  $ssbftCO$  : an SSBFT consensus object (Algorithm 6) that is used for agreeing on the
    garbage collector state, i.e., 1 when there is a need to recycle (otherwise 0);
145 interfaces provided:  $getIndex()$  do return  $index$ ;
146 message structure:  $\langle index \rangle$ : the logical object index;
147 upon pulse /* signal from global pulse system */ begin
148   let  $M$  be the arriving  $\langle index_j \rangle$  messages from  $p_j$ ;
149   switch  $clock(\kappa)$  /* consider  $clock()$  at the beginning of the pulse */
    do
150     case  $\kappa - 4$  do broadcast  $\langle index = getIndex() \rangle$ ;
151     case  $\kappa - 3$  do
152       let  $propose := \perp$ ;
153       if  $\exists v \neq \perp : |\{ \langle v \rangle \in M \}| \geq n - f$  then  $propose \leftarrow v$ ;
154       broadcast  $\langle propose \rangle$ ;
155     case  $\kappa - 2$  do
156       let  $bit := 0$ ;  $save \leftarrow \perp$ ;
157       if  $\exists s \neq \perp : |\{ \langle s \rangle \in M \}| > n/2$  then  $save \leftarrow s$ ;
158       if  $|\{ \langle save \neq \perp \rangle \in M \}| \geq n - f$  then  $bit \leftarrow 1$ ;
159       if  $save = \perp$  then  $save \leftarrow 0$ ;
160       broadcast  $\langle bit \rangle$ ;
161     case  $\kappa - 1$  do
162       let  $inc := 0$ ;
163       if  $ssbftCO.result()$  then  $inc \leftarrow 1$ ;
164       if  $|\{ \langle 1 \rangle \in M \}| \geq n - f$  then  $index \leftarrow (save + inc) \bmod I$ ;
165       else if  $|\{ \langle 0 \rangle \in M \}| \geq n - f$  then  $index \leftarrow 0$ ;
166       else  $index \leftarrow rand(save + inc) \bmod I$ ;

```

results and, by Assumption 5.1, all $n - t$ correct nodes will retrieve the object result within a known number of synchronous rounds. Then, the object could be recycled. Algorithm 6's detailed presentation and correctness proof appear in Section 5.4.

5.2.2 SSBFT simultaneous increment-or-get index (Algorithm 7)

Algorithm 7 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

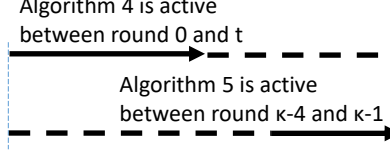


Figure 3: The schedule for algorithms 6 to 7 uses a cycle of $\kappa = \max\{t + 1, \logSize\}$ synchronous rounds.

this task *simultaneous increment-or-get index* (SIG-index). During legal executions of Algorithm 7, the correct nodes assert their agreement on the index value and update the index according to the result of Algorithm 6, which is an agreement on the value of `wasDelivered()`. Algorithm 7's detailed presentation and correctness proof appear in Section 5.5.

5.2.3 Scheduling strategy for algorithms 6 to 7

As mentioned, Algorithm 6 requires $t + 1$ synchronous rounds to complete and provide input to Algorithm 7 and $\kappa - (t + 1)$ synchronous rounds after that, any correct node can recycle the current object (according to Algorithm 6's result), where $\kappa = \max\{t + 1, \logSize\}$. Thus, Algorithm 7 has to defer its index updates until that time. Figure 3 presents this scheduling strategy, which considers the schedule cycle s of κ . That is, algorithms 6 and 7 starting points are 0 and $\kappa - 4$, respectively. Note that Algorithm 5 does not require scheduling since it accesses the index only via Algorithm 7's interface of SIG-index, see Figure 2.

5.2.4 Communication piggybacking and multiplexing

We use a piggybacking technique in order to facilitate the spread of the result (decision) values of the recyclable objects. As Figure 2 illustrates, all communications are piggybacked. Specifically, we consider a meta-message $MSG()$ that has a field for each message sent by algorithms 3, 6, and 7. That is, when any of the algorithms 6 and 7 are active, its respective field in $MSG()$ includes a non- \perp value. With respect to Algorithm 3's field, $MSG()$ includes the most recent message that Algorithm 3 has sent (or currently wishes to send). We note that this piggybacking technique allows the multiplexing of timed and reliable communications (assumed for $\text{BSMP}_{n,t}[\kappa\text{--SGC}, t < n/3, \text{RCCs}]$) and fair communication (assumed for $\text{BAMP}_{n,t}[\text{--FC}, t < n/3]$).

5.3 SSBFT recycling in $\text{BSMP}_{n,t}[t < n/3, (t + 1)\text{--SGC}]$ (Algorithm 5)

As mentioned, Algorithm 5 considers an array, $obj[]$ (line 123), of $indexNum$ recyclable objects (line 119). We require the array size to be larger than \logSize (line 120 and Assumption 5.1). In addition to the array $obj[]$, Algorithm 5's variable set includes $ssbftIndex$, which is an integer that holds the entry number of the latest object in use. Algorithm 5 accesses the agreed current index by calling $ssbftIndex.getIndex()$. This lets the algorithm's

code to nullify any entry in $obj[]$ that is not $ssbftIndex.getIndex()$ or at most $logSize$ older than $ssbftIndex.getIndex()$. Corollary 5.2 is directly implied by Assumption 5.1 and the properties of algorithms 6 to 7, which we show in sections 5.4.3 and 5.5.2, respectively.

Corollary 5.2 *Algorithm 5 is an SSBFT recycling mechanism that stabilizes within expected $\mathcal{O}(\kappa)$ synchronous rounds.*

5.4 SSBFT multivalued consensus in $\text{BSMP}_{n,t}[t < n/3, (t + 1)\text{--SGC}]$

Algorithm 6 assumes access to a deterministic (non-self-stabilizing) BFT (multivalued) consensus object, co , such as the ones proposed by Kowalski and Mostéfaoui [71] 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 the consensus object in Section 5.4.1.

5.4.1 Required interface to the consensus object

The proposed SSBFT solution uses the technique of recomputation of co 's floating output [42, Chapter 2.8]. In order to provide this, we assume that co has the following interface:

- $restart()$ sets co to its initial state.
- $propuse(v)$ 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. 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 $propuse()$.
- $result()$ 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 simultaneity taken a sequence of correct $process()$ invocations.

5.4.2 Detailed description

Algorithm 6's set of variables includes co itself (line 128) and the current version of the result, *i.e.*, $currentResult$ (line 127). This way, the SSBFT version of co 's result can be retrieved via a call to $result()$ (line 132). Algorithm 6 proceeds in periodic rounds. At the start of any round, node p_i stores all the arriving messages at the message vector M (line 135).

When the clock value is zero (line 136), it is time to start the re-computation of co 's result. Thus, Algorithm 6 first stores the current value of co 's result at $currentResult_i$ (line 137). Then, it restarts co 's local state and proposes a new value to co (lines 138 and 139). For the recycling solution presented in this paper, the proposed value is retrieved from $wasDelivered()$ (line 85). For the case in which the clock value is not zero (line 140), Algorithm 6 simply lets co to 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 6 broadcasts co 's messages for the current round (line 141).

5.4.3 Correctness proof

Theorem 5.3 shows that Algorithm 6 stabilizes within $2(t + 1)$ synchronous rounds.

Theorem 5.3 *Algorithm 6 is an SSBFT deterministic (multivalued) consensus solution for $BSMP_{n,t}[t < n/3, (t + 1)\text{--}SGC]$ that recovers after the occurrence of the last transient-faults within $\max\{\kappa, 2(t + 1)\}$ synchronous rounds.*

Proof of Theorem 5.3 Let R be an execution of Algorithm 6. 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 138 and 139) before sending the needed messages (line 141). Then, for the t synchronous rounds that follows, all correct nodes simultaneously process the arriving messages and send their replies (line 140 and 141). Thus, after $\max\{\kappa, 2(t + 1)\}$ synchronous rounds from c , 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. That results in guaranteed to be correct due to Section 5.4.1's assumptions. $\square_{Theorem\ 5.3}$

5.5 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 $getIndex()$. We use the $BSMP_{n,t}[t < n/3, RCC, 4\text{--}SGC]$ model. The task assumes that all increments are performed according to the result of a consensus object, $ssbftCO$, such as Algorithm 6. Algorithm 7 presents an SGI-index solution that recovers from disagreement on the index value using an RCC. That is, whenever a correct node receives $n - f$ reports from other nodes that they have each observed $n - f$ 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 [10]'s SSBFT clock synchronization algorithm, which is in turn derived from non-self-stabilizing BFT solutions by Rabin [95] as well as Turpin and Coan [107].

5.5.1 Detailed description

Algorithm 7 is active during four clock phases of a common pulse, *i.e.*, $\kappa - 4, \kappa - 3, \kappa - 2$, and $\kappa - 1$. Each phase starts with storing all arriving messages (from the previous synchronous round) in the array, M (line 148). The first phase broadcasts the local index value (line 150). The second phase lets each node vote on the majority arriving index value, or \perp in case such value was not received (lines 152 to 154). 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 156 to 159). Specifically, if *save* $\neq \perp$ exists, then $\langle bit = 1 \rangle$ is broadcast. Otherwise, $\langle bit = 0 \rangle$ is broadcast. On the fourth phase (line 162 to 166), the (possibly new) index is set either to be the majority-supported index value of phase two plus *inc* (line 162 to 164), where *inc* is the output of *ssbftCO*, or (if there was insufficient support) to a randomly chosen output of the RCC (lines 165 and 166).

5.5.2 Correctness proof

Theorem 5.4 shows that Algorithm 7 stabilizes within expected $\mathcal{O}(\kappa)$ synchronous rounds.

Theorem 5.4 *Let R be an execution of algorithms 6 and 7 that is legal w.r.t. Algorithm 6 (Theorem 5.3). Algorithm 7 is an SSBFT SGI-index implementation that stabilizes within expected $\mathcal{O}(\kappa)$ synchronous rounds.*

Proof of Theorem 5.4 Corollaries 5.5 and 5.6 are needed for the proof of lemmas 5.7 and 5.12. The pigeonhole principle implies Corollary 5.5 and Corollary 5.6 is implied by Corollary 5.5.

Corollary 5.5 *Let $V_x : x \in \{a, b\}$ be two n -length vectors that differ in at most $f < n/3$ entries. For any $x \in \{a, b\}$, suppose V_x contains $n - f$ copies of v_x . Then $v_a = v_b$.*

Corollary 5.6 is implied by Corollary 5.5.

Corollary 5.6 *Let $c[r] \in R$ be a system state in which $\text{clock}(\kappa) = \kappa - 3$ 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 154 at the end of $a_i[r]$. The set X includes at most one non- \perp value.*

Lemma 5.7 implies that, within $\mathcal{O}(\kappa)$ of expected rounds, all correct nodes have the identical *index* values. Recall that $c[r] \in R$ is (*progress*) *enabling* if $\exists x \in \{0, 1\} : \forall i \in \text{Correct} : \text{rand}_i = x$ holds at $c[r]$ (Section 1.7.2).

Lemma 5.7 (Convergence) *Let $r > \kappa$. Suppose $c[r] \in R$ is (*progress*) *enabling* system state (Section 1.7.2) for which $\text{clock}(\kappa) = \kappa - 1$ holds. With probability at least $\min\{p_0, p_1\}$, all correct nodes have the same index at $c[r + 1]$.*

Proof of Lemma 5.7 The proof is implied by claims 5.8 to 5.11.

Claim 5.8 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 $n - f$ 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 5.8 The proof is implied directly from lines 163 to 166. $\square_{\text{Claim 5.8}}$

Claim 5.9 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 $n - f$ different nodes. Also, suppose there is $x \in \{0, 1\}$ and a correct node $p_j \in \mathcal{P}$ that receive at the start of step $a_j[r]$ the message $\langle x \rangle$ from at least $n - f$ different nodes, where $i = j$ may or may not hold. The step $a_j[r]$ assigns 0 to $index_j$.

Proof of Claim 5.9 Line 165 implies the proof since $x = 0$ (Corollary 5.5). $\square_{\text{Claim 5.9}}$

Claim 5.10 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 $n - f$ different nodes. Let $p_j \in \mathcal{P}$ be a correct node. At $c[r]$, $ssbftCO_i.\text{result}() = ssbftCO_j.\text{result}()$ and $save_i = save_j$ hold.

Proof of Claim 5.10 At $c[r]$, $ssbftCO_i.\text{result}() = ssbftCO_j.\text{result}()$ holds (Algorithm 6's agreement property). The rest of the proof shows that $save_i = save_j$ holds at $c[r]$.

Since p_i has received $\langle 1 \rangle$ from at least $n - f$ 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 156 to 158, node p_j receives at the start of $a_j[r - 1]$ the message $\langle x \rangle$ from at least $n - f$ nodes, where $x = save_j \neq \perp$. By Corollary 5.6, any correct node broadcasts (line 154) 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 $n - f$ different nodes the messages $\langle x_i \rangle$, and resp., $\langle x_j \rangle$, where neither x_i nor x_j are \perp .

$\square_{\text{Claim 5.10}}$

Claim 5.11 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 $n - f$ 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 $n - f$ 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 5.9 By Corollary 5.5, we know that $x = 1$. Note that x 's value is determined during step $a[r - 1]$ and $rand$ is chosen at the start of step $a[r]$. Due to $rand$'s unpredictability (Section 1.7.2), $rand$ 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 5.10), where $save$ and inc are values determined by lines 162, 163 and 156, and resp., lines 157 and 159. $\square_{\text{Claim 5.9}}$ $\square_{\text{Lemma 5.7}}$

Lemma 5.12 shows that all correct nodes forever agree on their index values and simultaneously increment the index by one (modulo I) only when $clock(\kappa) = \kappa - 1$ and

$ssbftCO_i.\text{result}() = 1$. Lemma 5.12 uses the following notation. Let $R = c[0], c[1], \dots, c[r], \dots$ an unbounded synchronous execution of Algorithm 7, 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 $index_i : i \in \text{Correct}$ values of correct nodes 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 5.12 (Closure) *Let $c[r] \in R$, such that $clock(\kappa) = \kappa - 1$ at $c[r]$. Suppose $indices_r^{end} = \{v \neq \perp\}$. For every $c[r'] \in R : r' \in \{r+1, r+\kappa\}$ it holds that $indices_{r'}^{start} = \{v + x \bmod I\}$ where x is 1 when $r' = r + \kappa$ and $ssbftCO.\text{result}() = 1$ and 0 when $r' \in \{r+1, r+\kappa-1\}$ or $ssbftCO.\text{result}() \neq 1$.*

Proof of Lemma 5.12 For $r' = r+1$ the lemma holds since, by definition, $\forall r'' : indices_{r''}^{end} = indices_{r''+1}^{start}$. Also, for any system state between $c[r'] : r' \in \{r+1, r+\kappa-1\}$, no correct node, $p_i \in \mathcal{P}$, updates $index_i$ during the step, $a_i[r']$, since $clock(\kappa) \neq \kappa - 1$ at $c[r'] : r' \in \{r+1, r+\kappa-1\}$ and thus lines 164 to 166 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 6, the arguments above about $c[r'] : r' \in \{r+1, r+\kappa-1\}$ as well as Claim 5.13.

Claim 5.13 $indices_{r+\kappa}^{start} = \{v\} : v \neq \perp$.

Proof of Claim 5.13 By the schedule (Figure 3) and the length of the scheduling cycle, κ , we know that Algorithm 7 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 the steps $a[r+\kappa-3]$, all correct nodes broadcast $\langle v \rangle$ (line 150). Thus, at the start of steps $a[r+\kappa-3]$, all correct nodes receive $\langle v \rangle$ at least $n - f$ times. Thus, during $a[r+\kappa-2]$, all correct nodes assign v to their *propose* variables (line 153) and broadcast $\langle v \rangle$ (line 154). By similar arguments, during $a[r+\kappa-1]$, all correct nodes assign v and 1 to their *save*, and resp., *bit* variables (lines 157 to 158) and broadcast $\langle 1 \rangle$ (line 160). Therefore, all correct nodes receive $\langle 1 \rangle$ at least $n - f$ times. This implies that during $a[r+\kappa]$, the if-statement condition in line 164 holds and thus $indices_{r+\kappa}^{start} = \{v \neq \perp\}$ holds. $\square_{\text{Claim 5.13}}$ $\square_{\text{Lemma 5.12}}$ $\square_{\text{Theorem 5.4}}$

6 Discussion

We have presented a new loosely-self-stabilizing variation of the MMR algorithm [87] for solving binary consensus for the $\text{BAMP}_{n,t}[-\text{FC}, t < n/3, \text{RCCs}]$ model. The proposed solution preserves the following properties of the studied algorithm: it does not require signatures, it offers optimal fault-tolerance, and the expected time until completion is the same as the studied algorithm. The proposed solution is able to achieve this using a new application of the design criteria of loosely-self-stabilizing systems, which requires the satisfaction of

Notation	Meaning
MMR	Mostéfaoui, Moumen, and Raynal [87]
BDH	Ben-Or, Dolev, and Hoch [10]
BFT	non-self-stabilizing Byzantine fault-tolerant solutions
SSBFT	self-stabilizing Byzantine fault-tolerant
BAMP _{n,t}	Byzantine Asynchronous Message-Passing model
BSMP _{n,t}	Byzantine synchronous message-passing model
<i>RCCs</i>	random common coins
<i>FC</i>	fair communication assumption
κ -SGC	κ -state global clock

Table 1: Glossary

safety properties with a probability in $\mathcal{O}(1 - 2^{-M})$. For any practical purposes and in the absence of transient-faults, one can select M to be sufficiently large so that the risk of violating safety is negligible. An SSBFT solution for recycling distributed objects and BSMP_{n,t}[κ -SGC, $t < n/3$, RCCs] is proposed in order to support an unbounded number of instances of our SSBFT MMR solution. We believe that this work is preparing the ground-work needed to construct self-stabilizing (BFT) algorithms for distributed systems, such as Blockchains, that need to run in hostile environments.

Acknowledgments

The work of M. Raynal was partially supported by the French ANR project DESCARTES (16-CE40-0023-03). The work of I. Marcoullis was funded by the ONISILLOS postdoctoral funding scheme of the University of Cyprus.

References

- [1] Ittai Abraham and Danny Dolev. Byzantine agreement with optimal early stopping, optimal resilience and polynomial complexity. In *STOC*, pages 605–614. ACM, 2015.
- [2] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *J. Comput. Syst. Sci.*, 81(4):692–701, 2015.
- [3] Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019.
- [4] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In *Distributed Algorithms, 7th International Workshop*,

- WDAG, volume 725 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 1993.
- [5] Yotam Ashkenazi, Shlomi Dolev, Sayaka Kamei, Yoshiaki Katayama, Fukuhito Ooshita, and Koichi Wada. Location functions for self-stabilizing byzantine tolerant swarms. In *SSS*, volume 13046 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 2021.
 - [6] Yotam Ashkenazi, Shlomi Dolev, Sayaka Kamei, Fukuhito Ooshita, and Koichi Wada. Forgive & forget: Self-stabilizing swarms in spite of byzantine robots. In *CANDAR Workshops*, pages 188–194. IEEE, 2019.
 - [7] James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *J. ACM*, 45(3):415–450, 1998.
 - [8] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *Int. J. Systems Science*, 28(11):1177–1187, 1997.
 - [9] Michael 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.
 - [10] Michael Ben-Or, Danny Dolev, and Ezra 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.
 - [11] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, pages 355–362. IEEE Computer Society, 2014.
 - [12] Alexander Binun, Thierry Coupaye, Shlomi Dolev, Mohammed Kassi-Lahlou, Marc Lacoste, Alex Palesandro, Reuven Yagel, and Leonid 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.
 - [13] Alexander Binun, Shlomi Dolev, and Tal Hadad. Self-stabilizing Byzantine consensus for blockchain - (brief announcement). In *Cyber Security Cryptography and Machine Learning - Third International Symposium, CSCML*, pages 106–110, 2019.
 - [14] Silvia Bonomi, Shlomi Dolev, Maria Potop-Butucaru, and Michel 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.

- [15] Silvia Bonomi, Maria Potop-Butucaru, and Sébastien Tixeul. Stabilizing Byzantine-fault tolerant storage. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 894–903, 2015.
- [16] Silvia Bonomi, Antonella Del Pozzo, and Maria Potop-Butucaru. Tight self-stabilizing mobile byzantine-tolerant atomic register. In *ICDCN*, pages 6:1–6:10. ACM, 2016.
- [17] Silvia Bonomi, Antonella Del Pozzo, and Maria Potop-Butucaru. Optimal self-stabilizing synchronous mobile byzantine-tolerant atomic register. *Theor. Comput. Sci.*, 709:64–79, 2018.
- [18] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeul. Optimal mobile Byzantine fault tolerant distributed storage: Extended abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC*, pages 269–278, 2016.
- [19] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeul. Optimal storage under unsynchronized mobile Byzantine faults. In *36th IEEE Symposium on Reliable Distributed Systems, SRDS*, pages 154–163, 2017.
- [20] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeul. 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.
- [21] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeul. Approximate agreement under mobile Byzantine faults. *Theor. Comput. Sci.*, 758:17–29, 2019.
- [22] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.
- [23] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.
- [24] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. *IACR Cryptol. ePrint Arch.*, 2001:6, 2001.
- [25] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantino-ple: Practical asynchronous Byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005.
- [26] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In *31st International Symposium on Distributed Computing, DISC*, volume 91 of *LIPIcs*, pages 1:1–1:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

- [27] Christian Cachin and Luca Zanolini. Brief announcement: Revisiting signature-free asynchronous byzantine consensus. In *DISC*, volume 209 of *LIPICs*, pages 51:1–51:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [28] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 42–51. ACM, 1993.
- [29] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
- [30] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
- [31] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Comput. J.*, 49(1):82–96, 2006.
- [32] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Veríssimo. Byzantine consensus in asynchronous message-passing systems: a survey. *Int. J. Crit. Comput. Based Syst.*, 2(2):141–161, 2011.
- [33] Ariel Daliot and Danny Dolev. Self-stabilizing byzantine agreement. In *PODC*, pages 143–152. ACM, 2006.
- [34] Ariel Daliot, Danny Dolev, and Hanna Parnas. Brief announcement: linear time byzantine self-stabilizing clock synchronization. In *PODC*, page 379. ACM, 2004.
- [35] Xavier Défago, Maria Potop-Butucaru, and Philippe Raipin Parvédy. Self-stabilizing gathering of mobile robots under crash or byzantine faults. *Distributed Comput.*, 33(5):393–421, 2020.
- [36] Xavier Défago, Maria Gradinariu Potop-Butucaru, Julien Clément, Stéphane Messika, and Philippe Raipin Parvédy. Fault and byzantine tolerant self-stabilizing mobile robots gathering - feasibility study -. *CoRR*, abs/1602.05546, 2016.
- [37] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
- [38] Danny Dolev and Ezra N. Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In *SSS*, volume 4838 of *Lecture Notes in Computer Science*, pages 234–252. Springer, 2007.
- [39] Danny Dolev and Ezra N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In *DISC*, volume 4731 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2007.

- [40] Danny Dolev, Ezra N. Hoch, and Robbert van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *OPODIS*, volume 4878 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2007.
- [41] Danny Dolev and H. Raymond Strong. Polynomial algorithms for multiple processor agreement. In *STOC*, pages 401–407. ACM, 1982.
- [42] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
- [43] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *Inf. Process. Lett.*, 111(18):912–920, 2011.
- [44] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Practically-self-stabilizing virtual synchrony. *J. Comput. Syst. Sci.*, 96:50–73, 2018.
- [45] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael 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.
- [46] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks - (extended abstract). In *SSS*, volume 7596 of *LNCS*, pages 133–147. Springer, 2012.
- [47] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. Self-stabilizing Byzantine resilient topology discovery and message delivery - (extended abstract). In *Networked Systems - First International Conference, NETYS*, pages 42–57, 2013.
- [48] Shlomi Dolev, Thomas Petig, and Elad Michael 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.
- [49] Shlomi Dolev, Thomas Petig, and Elad Michael 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*.
- [50] Shlomi Dolev and Elad Schiller. Communication adaptive self-stabilizing group membership service. *IEEE Trans. Parallel Distributed Syst.*, 14(7):709–720, 2003.
- [51] Shlomi Dolev and Jennifer 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.

- [52] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, 2004.
- [53] Rongcheng Dong, Yuichi Sudo, Taisuke Izumi, and Toshimitsu Masuzawa. Loosely-stabilizing maximal independent set algorithms with unreliable communications. In *SSS*, volume 13046 of *Lecture Notes in Computer Science*, pages 335–349. Springer, 2021.
- [54] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 2028–2041. ACM, 2018.
- [55] Swan Dubois, Maria Potop-Butucaru, Mikhail Nesterenko, and Sébastien Tixeuil. Self-stabilizing Byzantine asynchronous unison. *J. Parallel Distributed Comput.*, 72(7):917–923, 2012.
- [56] Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Dynamic FTSS in asynchronous systems: The case of unison. *Theor. Comput. Sci.*, 412(29):3418–3439, 2011.
- [57] Romaric Duvignau, Michel Raynal, and Elad Michael Schiller. Self-stabilizing byzantine-tolerant broadcast. *CoRR*, abs/2201.12880, 2022.
- [58] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 148–161. ACM, 1988.
- [59] Paul Feldman and Silvio Micali. An optimal probabilistic algorithm for synchronous byzantine agreement. In *Automata, Languages and Programming, 16th International Colloquium, ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 341–378. Springer, 1989.
- [60] Michael Feldmann, Thorsten Götte, and Christian 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.
- [61] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.
- [62] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [63] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM J. Comput.*, 27(1):247–290, 1998.

- [64] Chryssis Georgiou, Robert Gustafsson, Andreas Lindhe, and Elad Michael Schiller. Self-stabilization overhead: an experimental case study on coded atomic storage. *CoRR*, abs/1807.07901, 2018.
- [65] Chryssis Georgiou, Robert Gustafsson, Andreas Lindhé, and Elad Michael Schiller. Self-stabilization overhead: A case study on coded atomic storage. In *Networked Systems - 7th International Conference, NETYS*, pages 131–147, 2019.
- [66] Chryssis Georgiou, Oskar Lundström, and Elad Michael 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.
- [67] Ezra N. Hoch, Danny Dolev, and Ariel Daliot. Self-stabilizing byzantine digital clock synchronization. In *SSS*, volume 4280 of *Lecture Notes in Computer Science*, pages 350–362. Springer, 2006.
- [68] Taisuke Izumi. On space and time complexity of loosely-stabilizing leader election. In Christian Scheideler, editor, *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO*, volume 9439 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2015.
- [69] Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Inf. Process. Lett.*, 85(1):47–52, 2003.
- [70] Pankaj Khanchandani and Christoph Lenzen. Self-stabilizing Byzantine clock synchronization with optimal precision. *Theory Comput. Syst.*, 63(2):261–305, 2019.
- [71] Dariusz R. Kowalski and Achour 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.
- [72] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [73] Leslie Lamport. Byzantizing paxos by refinement. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2011.
- [74] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- [75] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
- [76] Christoph Lenzen and Joel Rybicki. Self-stabilising Byzantine clock synchronisation is almost as easy as consensus. *J. ACM*, 66(5):32:1–32:56, 2019.

- [77] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. Self-stabilizing set-constrained delivery broadcast (extended abstract). In *40th IEEE International Conference on Distributed Computing Systems, ICDCS*, pages 617–627, 2020.
- [78] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. Self-stabilizing uniform reliable broadcast. In *Networked Systems - 8th International Conference, NETYS*, pages 296–313, 2020.
- [79] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. Self-stabilizing indulgent zero-degrading binary consensus. In *ICDCN '21: International Conference on Distributed Computing and Networking*, pages 106–115, 2021.
- [80] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. Self-stabilizing multivalued consensus in asynchronous crash-prone systems. *CoRR*, abs/2104.03129, 2021.
- [81] Mahyar R. Malekpour. A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. In *SSS*, volume 4280 of *Lecture Notes in Computer Science*, pages 411–427. Springer, 2006.
- [82] Toshimitsu Masuzawa and Sébastien Tixeuil. A self-stabilizing link-coloring protocol resilient to unbounded byzantine faults in arbitrary networks. In *OPODIS*, volume 3974 of *Lecture Notes in Computer Science*, pages 118–129. Springer, 2005.
- [83] Alexandre 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.
- [84] Alexandre Maurer and Sébastien Tixeuil. Self-stabilizing Byzantine broadcast. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014*, pages 152–160. IEEE Computer Society, 2014.
- [85] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *CCS*, pages 31–42. ACM, 2016.
- [86] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. RITAS: services for randomized intrusion tolerance. *IEEE Trans. Dependable Secur. Comput.*, 8(1):122–136, 2011.
- [87] Achour Mostéfaoui, Moumen Hamouma, and Michel 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.
- [88] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ expected time. *J. ACM*, 62(4):31:1–31:21, 2015.

- [89] Achour Mostéfaoui and Michel Raynal. Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Acta Informatica*, 54(5):501–520, 2017.
- [90] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and kdc. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346. Springer, 1999.
- [91] Mikhail Nesterenko and Sébastien Tixeuil. Discovering network topology in the presence of Byzantine faults. *IEEE Trans. Parallel Distributed Syst.*, 20(12):1777–1789, 2009.
- [92] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
- [93] Martin Perner, Martin Sigl, Ulrich Schmid, and Christoph Lenzen. Byzantine self-stabilizing clock distribution with hex: Implementation, simulation, clock multiplication. In *6th Conference on Dependability (DEPEND)*. Citeseer, 2013.
- [94] David Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers: FTCS-22, The Twenty-Second Annual International Symposium on Fault-Tolerant Computing*, pages 386–395, 1992.
- [95] Michael O. Rabin. Randomized Byzantine generals. In *24th Annual Symposium on Foundations of Computer Science*, pages 403–409, 1983.
- [96] Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018.
- [97] Luís E. T. Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Trans. Knowl. Data Eng.*, 15(5):1206–1217, 2003.
- [98] Yusuke Sakurai, Fukuhito Ooshita, and Toshimitsu Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *OPODIS*, volume 3544 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2004.
- [99] Iosif Salem and Elad Michael Schiller. Practically-self-stabilizing vector clocks in the absence of execution fairness. In *Networked Systems - 6th International Conference, NETYS*, pages 318–333, 2018.
- [100] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
- [101] Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotugu Kaku-gawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theor. Comput. Sci.*, 444:100–112, 2012.

- [102] Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu 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.
- [103] Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K. Datta, and Lawrence L. Larmore. Loosely-stabilizing leader election for arbitrary graphs in population protocol model. *IEEE Trans. Parallel Distributed Syst.*, 30(6):1359–1373, 2019.
- [104] Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K. Datta, and Lawrence L. Larmore. Loosely-stabilizing leader election with polylogarithmic convergence time. *Theor. Comput. Sci.*, 806:617–631, 2020.
- [105] Pierre Tholoniati and Vincent Gramoli. Certifying blockchain byzantine fault tolerance. *CoRR*, abs/1909.07453, 2019.
- [106] Sam Toueg. Randomized Byzantine agreements. In Tiko Kameda, Jayadev Misra, Joseph G. Peters, and Nicola Santoro, editors, *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 163–178. ACM, 1984.
- [107] Russell Turpin and Brian A. Coan. Extending binary Byzantine agreement to multi-valued Byzantine agreement. *Inf. Process. Lett.*, 18(2):73–76, 1984.
- [108] Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, 2015.
- [109] Yang Xiao, Ning Zhang, Wenjing Lou, and Y. Thomas Hou. A survey of distributed consensus protocols for blockchain networks. *IEEE Commun. Surv. Tutorials*, 22(2):1432–1465, 2020.
- [110] Shaolin Yu, Jihong Zhu, and Jiali Yang. Efficient two-dimensional self-stabilizing byzantine clock synchronization in WALDEN. In *ICPADS*, pages 723–730. IEEE, 2021.
- [111] Shaolin Yu, Jihong Zhu, Jiali Yang, and Wei Lu. Expected constant time self-stabilizing byzantine pulse resynchronization. *CoRR*, abs/2203.14016, 2022.