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

Chryssis Georgiou Ioannis Marcoullis Michel Raynal
Elad Michael Schiller

May 14, 2021

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 nodes 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 can deal with up to t Byzantine nodes, where $t < n/3$ and n is the number of nodes. MMR assumes the availability of a common coin service and fair scheduling of message arrivals, which does not depend on the current coin values. It terminates 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 node 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 the 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. 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. Furthermore, it only requires a bounded amount of memory.

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, called *nodes* or *processors*, is called *consensus* [57, 7]. The most basic form of the consensus problem is for processors to decide between two possible values, *e.g.,* zero or one. This version of the problem is called *binary consensus* [71, Ch. 14]. 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 Blockchain.

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 [46], the system models are also equipped with additional capabilities, such as cryptography, oracles, *e.g.,* perfect failure detectors, and randomization [27]. 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" [81, 22] 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 processors 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, see Definition 1.1. Otherwise, it is called multivalued consensus.

Definition 1.1 *Every processor p_i has to propose a value $v_i \in V = \{0, 1\}$, via an invocation of the* $\mathsf{propose}_i(v_i)$ *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-termination, requirements.*

- ***BC-validity.*** *The value $v \in \{0, 1\}$ decided by a correct processor is a value proposed by a correct processor.*
- ***BC-agreement.*** *Any two correct processors that decide, do so with identical decided values.*
- ***BC-termination.*** *All correct processors decide.*

Starting from the algorithm of Mostéfaoui, Moumen, and Raynal [65], 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-termination requirement within a finite time that is known only by expectation. The proposed solution satisfies BC-termination 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 node 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, see Remark 3.1 for details.

1.3 Fault model

We study asynchronous solutions for message-passing systems where the algorithm cannot explicitly access the local clock or assume the existence of guarantees on the communication delay. We model a broad set of benign 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 (i) communication failures, such as packet omission, duplication, and reordering, as well as (ii) up to t node failures, *i.e.,* crashed or Byzantine. In detail, a faulty node 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 nodes that the adversary can capture. We also restrict the adversary from letting a captured node 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.

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 nodes 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 [28] assumes that these violations bring the system to an arbitrary state from which a *self-stabilizing system* should recover, see [29, 2] 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* [75] 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.

1.4 Related work

1.4.1 Impossibilities and lower-bounds

The FLP impossibility result [46] concluded that consensus is impossible to solve deterministically in asynchronous settings in the presence of even a single crash failure. In [45] 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 nodes in the system, for which there are no (randomized) self-stabilizing solutions [3, 6]. In this work, we consider weaker design criteria than Dijkstra's self-stabilization.

In the presence of Byzantine faults, consensus is not solvable if a third or more of the nodes are faulty [57]. Thus, optimally resilient Byzantine consensus algorithms, such as the one we present, tolerate $t < n/3$ faulty nodes. The task is also impossible if a node can impersonate some other node in its communication with the other entities [7]. 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) [41, 40]. As indicated by the above impossibility results, the studied problem remains challenging even under randomization and fairness assumptions during the recovery period.

1.4.2 Non-self-stabilizing non-Byzantine fault-tolerant solutions

Paxos [54] is the best-known solution for the consensus problem. Despite becoming notorious for being complex [56], Paxos was followed by rich literature [80]. Raynal [71] 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 [25, 72].

1.4.3 Non-self-stabilizing Byzantine fault-tolerant solutions

Byzantine fault-tolerant consensus was tackled by many protocols [64]. Several variants of Paxos consensus tolerate such malicious processors, *e.g.,* [55]. State machine replication protocols, such as PBFT [24] and BFT-SMART [9] incorporate a Byzantine-tolerant consensus mechanism.

Randomization can circumvent the FLP impossibility [45], which only entails deterministic algorithms. This line of work started with Ben-Or [7] 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 [70] in the same year, which assumes the availability of a common coin, allowed for a polynomial number of communication steps and optimal resilience. We later discuss more extensively the notion of common coins. Bracha [18] 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. [21] solve asynchronous binary consensus using a common coin and cryptographic threshold signatures. They achieve optimal resilience ($t < n/3$) and quadratic message-per-round complexity.

In the sequel, we focus on MMR [65] as a signature-free Byzantine-tolerant solution for binary consensus. This algorithm is optimal in resilience, uses $O(n^2)$ messages per consensus invocation, and terminates within $O(1)$ expected time. The MMR can be combined with a reduction of multivalued consensus to binary consensus [66] to attain multivalued consensus with the same fault-tolerance properties.

Binary consensus is a fundamental component of total order reliable broadcast, *e.g.,* [20, 26] (see Section 1.5). In what appears as a revival of the topic, several Blockchain consensus protocols are also using similar approaches. HoneyBadger [63] was the first randomized BFT protocol for Blockchain. They employ MMR as their binary consensus protocol. The BEAT [39] suite of protocols for blockchain consensus also uses the MMR.

1.4.4 Common coin services

Randomized algorithms employ coin flips to circumvent the FLP impossibility [45], which only entails deterministic algorithms. The two known coin flip constructions are *local coins*, where each processor only uses a local random function, and *common coins*, where the k-th invocation of the random function by a correct (non-Byzantine) processor, returns the same bit as to any other correct processor. Ben-Or [7] using a local coin, developed an asynchronous Byzantine-tolerant 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 [70] was the first to introduce a common coin demonstrating the possibility of designing asynchronous Byzantine-tolerant 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 [74] and digital signatures for authenticating the messages exchanged. Since then, common coin provision has become an essential tool, and many subsequent works have devised randomized coin-flipping algorithms, *e.g.,* [42, 43, 23, 67, 20, 21, 8, 19] as building blocks for consensus and other related problems, such as clock synchronization. Aspens [4] demonstrates that agreeing on a common coin 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 a common coin 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.,* [42, 43, 23, 8] or digital signatures and other cryptographic tools are employed, *e.g.,* [74, 67, 20, 21]. 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 [42] show how to compile any protocol assuming private channels to a cryptographic protocol not assuming private channels which runs exactly the same.

To the best of our knowledge, the only self-stabilizing Byzantine-tolerant common coin construction

is the one by Ben-Or, Dolev, and Hoch [8] for synchronous (pulse-based) systems with private channels. They use a pipeline technique to transform the non-self-stabilizing synchronous Byzantine-tolerant coin-flipping algorithm of Feldman and Micali [43] into a self-stabilizing one; the work in [43] assumes private channels.

1.4.5 Self-stabilizing non-Byzantine fault-tolerant solutions

Lundström, Raynal, and Schiller [61] presented the first self-stabilizing solution for the problem of binary consensus for message-passing systems where nodes may fail by crashing. They ensure a line of self-stabilizing solutions [62, 59, 60, 49, 48]. This line follows the approach proposed by Dolev, Petig, and Schiller [36, 35] for self-stabilization in the presence of seldom fairness. Namely, in the absence of transient-faults, these self-stabilizing solutions are wait-free and no assumptions are made regarding the system's synchrony or fairness of its scheduler. However, the recovery from transient faults does require fair execution, *e.g.,* to perform a global reset, see [47, 48], 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 Byzantine fault-tolerant solution assumes fair scheduling of message arrivals and the accessibility to an independent common coin service.

We note the existence of other approaches for recovering from transient faults without assuming execution fairness during the recovery period [73, 31, 1]. However, none of these results consider both Byzantine fault-tolerance and self-stabilization.

Algorithms for loosely-self-stabilizing systems [76, 77, 78, 51] mainly focus on the task of leader election and population protocols. Recently, Feldmann, Götte, and Scheideler [44] 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 [76, 77, 78, 51, 44] since it requires the loosely-self-stabilizing condition to hold only eventually.

1.4.6 Self-stabilizing Byzantine fault-tolerant solutions

In the context of this dual design criteria, there are solutions for topology discovery [34], storage [16, 15, 14, 13, 12], clock synchronization [38, 58, 53], approximate agreement [17], asynchronous unison [40] to name a few. The most relevant work is the one by Binun *et al.* [10, 11] and Dolev *et al.* [32] for a deterministic Byzantine fault-tolerant 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. We study another problem, which is binary consensus.

1.5 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 [71, 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

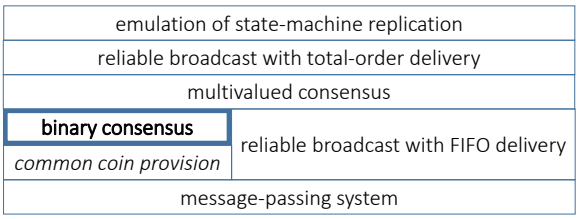

Figure 1: The hybrid architecture of asynchronous and synchronous components. The studied problem (which appears in boldface font and is surrounded by a thick frame) assumes no explicit synchrony but it requires the availability of a common coin service and fair scheduling of message arrival, which does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

via multivalued consensus. Whenever multivalued consensus is called, the latter invokes binary consensus for a finite number of times.

1.5.1 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, such as FLP's one. 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 finite number of them can be invoked for every transition of the state-machine and each such invocation has to take at least two communication rounds, due to a lower bound by Keidar and Rajsbaum [52]. Therefore, we select to study the non-self-stabilizing probabilistic MMR algorithm [65] for solving binary consensus in asynchronous message-passing-systems. MMR assumes access to a common coin service. Ben-Or, Dolev, and Hoch [8], in short BDH, presented a self-stabilizing synchronous solution for common coin provision.

1.5.2 Periodic re-installation of the common seed and initialization of consensus objects

We clarify the advantage of the studied architecture that considers a hybrid model that is composed of asynchronous, *i.e.,* MMR, and synchronous, *i.e.,* BDH, components. The use of the common coin service can aim at devising a random seed that is long enough to support plenty of invocations of binary consensus. In detail, the BDH algorithm can renew the common seed periodically, such that a pseudo-random generator can use the common seed, s_t, that was renewed at time t for generating the sequence $bits_{t,1}, bits_{t,2}, \ldots, bits_{t,x}$ of unique M-bits integers, where $M \in \mathbb{Z}^+$ is a bound on the number of pseudo-random bits each invocation of binary consensus might use and $x \in \mathbb{Z}^+$ is the highest value that guarantees that the sequence $bits_{t,1}, bits_{t,2}, \ldots, bits_{t,x}, bits_{t+1,1}, bits_{t+1,2}, \ldots, bits_{t+1,x}, \ldots$ satisfies the common coin requirements over time.

We note that high x values can mitigate the effect of BDH's synchrony assumption on the benefits that the system is expected to gain from selecting an asynchronous algorithm for solving binary consensus. Moreover, it is imperative to re-install the common seed repeatedly due to the need to

tolerate corruption of the common seed, after the occurrence of a transient fault. This is also imposed by the properties of pseudo-random generator functions, which eventually cannot avoid repeating the same sequences of "random" bits. Note that one can use the events of common seed re-installation also for the initialization of consensus objects. As we explain in sections 2.1.3 and 2.4.1, this can help to simplify the correctness proof since it implies that recovery from transient-faults depends only on the termination of all operations after the occurrence of the last transient fault. Our hybrid architecture and the assumptions made above helps us to circumvent the aforementioned impossibility results.

1.6 Our contribution

We present a fundamental module for dependable distributed systems: a loosely-self-stabilizing asynchronous algorithm for binary consensus for message-passing systems that are prone to Byzantine node 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 [65] for asynchronous message-passing systems. MMR assumes that $t < n/3$ and terminates within $O(1)$ expected time, where t is the number of faulty nodes and n is the total number of nodes. 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 time (without assuming execution fairness). After the occurrence of any finite number of arbitrary transient-faults, the system recovers within a finite time (while assuming execution fairness). Unlike in MMR, each node 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 node to complete at least one round-trip with every other non-faulty node.

To the best of our knowledge, we propose the first loosely-self-stabilizing Byzantine fault-tolerant asynchronous algorithm for solving binary consensus in message-passing systems. As such, there is a long line of distributed applications, such as service replication and Blockchain, that our contribution can facilitate solutions that are more fault-tolerant than the existing implementations since they cannot recover after the occurrence of the last transient fault.

1.7 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 stepping stone to our loosely-self-stabilizing algorithm that is featured in Section 4. Section 5 provides the correctness proof, Section 6 discusses an extension, and Section 7 concludes the paper.

2 System settings

We consider an asynchronous message-passing system that has no guarantees on the communication delay. Moreover, there is no notion of global (or universal) clocks and the algorithm cannot explicitly access the local clock (or timeout mechanisms). The system consists of a set, $\mathcal{P}$, of n fail-prone nodes (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 $\mathsf{channelCapacity} \in \mathbb{N}$ packets on transit from p_j to p_i (this assumption is due to a well-known impossibility [29, Chapter 3.2]).

In the *interleaving model* [29], the node's program is a sequence of *(atomic) steps*. Each step starts with an internal computation and finishes with a single communication operation, *i.e.,* a message *send* or *receive*. The *state*, s_i, of node $p_i \in \mathcal{P}$ includes all of p_i's variables and $channel_{j,i}$. The term *system state* (or configuration) refers to the tuple $c = (s_1, s_2, \cdots, s_n)$. We define an *execution (or run)* $R = c[0], a[0], c[1], a[1], \ldots$ as an alternating sequence of system states $c[x]$ and steps $a[x]$, such that each $c[x+1]$, except for the starting one, $c[0]$, is obtained from $c[x]$ by $a[x]$'s execution.

2.1 Task specifications

2.1.1 Returning the decided value

Definition 1.1 considers the $\mathsf{propose}(v)$ operation. We refine the definition of $\mathsf{propose}(v)$ by specifying how the decided value is retrieved. This value is either returned by the $\mathsf{propose}()$ operation (as in the studied algorithm [65]) or via the returned value of the $\mathsf{result}()$ 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 indicate a (transient) error that occurs only when the proposed algorithm exceed 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-termination. Specifically, MMR states that each non-faulty processor decides with probability 1. Also, since MMR is a round-based algorithm, it holds that $\lim_{r \to +\infty}(\Pr_{\text{MMR}}[p_i \text{ decides by round } r]) = 1$.

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 $\Pr_{proposed}[p_i \text{ decides by round } M{+}1] = 1$. Due to this, the proposed algorithm provides a randomized guarantee with respect to the safety requirements, *i.e.,* BC-validity and BC-agreement. Specifically, $\Pr_{proposed}[p_i \text{ satisfies the safety requirements}] = 1 - \mathcal{O}(2^{-M})$. 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 correct processors 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 [69] 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 the 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-termination requirement (Definition 1.1), LE_{binCon} includes only finite executions. In Section 2.4.2, we consider executions $R = R_1 \circ R_2 \circ, \ldots$ as infinite compositions of finite executions, $R_1, R_2, \ldots \in LE_{\text{binCon}}$, such that R_x includes one invocation of task T_{binCon}, which always satisfies the liveness requirement, *i.e.,* BC-termination, 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 correct processors.

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 processor including crashes, arbitrary behavior, and malicious behaviors. Here the adversary lets each node receive the arriving messages and calculate its state according to the algorithm. However, once a node (that is captured by

the adversary) sends a message, the adversary can modify the message in any way, delay it for an arbitrarily long period or even remove it from the communication channel. Note that the adversary has the power to coordinate such actions without any limitation about his computational or communication power.

We also note that the studied algorithm, MMR, assumes the absence of spoofing attack, 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 common coin service, and since the only available, to the best of our knowledge, self-stabilizing common coin algorithm, BDH [8], assumes private channels, we also assume that the communications between any two non-faulty processors are private. That is, it cannot be read by the adversary.

For the sake of solvability [57, 68, 79], 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 processors in the system, *i.e.,* $3t + 1 \leq n$. The set of non-faulty processors is denoted by *Correct* and called the set of correct processors.

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 [29]. 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 [28] 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 [29]. Wait-free algorithms guarantee that operations (that were invoked by non-failing nodes) always terminate 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

It is well-known that self-stabilizing algorithms cannot terminate their execution and stop sending messages [29, Chapter 2.3]. Moreover, their code includes a do-forever loop. The proposed algorithm uses M communication round numbers. Let $r \in \{1, \ldots, M\}$ 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 processor $p_i \in \mathcal{P} : i \in Correct$ starts and ends its r-th iteration, $I_{i,r}$, of the do-forever loop. Moreover, let $m_{i,r,j,ackReq=\mathsf{True}}$ be a message that p_i sends to p_j during $I_{i,r}$, where the field $ackReq = \mathsf{True}$ implies that an acknowledgment reply is required. Let $a_{i,r,j,\mathsf{True}}, a_{j,r,i,\mathsf{False}} \in R$ be the steps in which $m_{i,r,j,\mathsf{True}}$ and $m_{j,r,i,\mathsf{False}}$ arrive to p_j and p_i, respectively. We require A_r to also include, for every pair of correct processors $p_i, p_j \in \mathcal{P} : i, j \in Correct$, the steps $a_{i,r,j,\mathsf{True}}$ and $a_{j,r,i,\mathsf{False}}$. We say that A_r is *complete* if every correct processor $p_i \in \mathcal{P} : i \in Correct$ 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 simple 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 layer's 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 termination 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 termination 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 [3, 45, 40]

To circumvent such challenges, Sudo *et al.* [75] 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 $\mathsf{op}()$ to satisfy task $T_{\mathsf{op}()}$'s safety requirements with a probability $p_{\mathcal{S}}$. Definition 2.1 uses the term correct invocation of operation $\mathsf{op}()$. Recall that Section 2.1, defines what a correct invocation of binary consensus is, *i.e.,* it is required that all correct processors invoke the $\mathsf{propose}()$ operation exactly once during any execution that is in LE_{binCon}.

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

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

Definition 2.2 (Probabilistic (operation-based) eventually-loosely-self-stabilizing systems) *Let $\mathcal{S}$ be a system that implements a probabilistic solution for task $T_{\mathsf{op}()}$. Let R be any unbounded execution of $\mathcal{S}$, which includes repeated sequential and correct invocations of $\mathsf{op}()$, such that task $T_{\mathsf{op}()}$ terminates 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 $T_{\mathsf{op}()}$'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).

2.5 Enhancing the computation model with common coins

A common coin service (Section 1.4.4) delivers to all processors, via the operation randomBit(r) : $r \in \mathbb{Z}^+$, identical sequences of random bits $b_1, b_2, \ldots, b_r, \ldots : b_r \in \{0, 1\}$. We assume that $\Pr(b_r = 0) = \Pr(b_r = 1) = 1/2$ and that b_r is independent of $b_{r'}$, where $r, r' \in \mathbb{Z}^+$. As mentioned earlier, we follow MMR's assumption regarding the fair scheduling of message arrivals that does not depend on the current coin's value.

3 The MMR Non-self-stabilizing Solution

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 algorithm (Section 3.2) that serves as a stepping stone to the proposed algorithm (Section 4).

Algorithm 1: Non-self-stabilizing MMR algorithm for Binary Byzantine fault-tolerant 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 processors and bVAL(vJ) not yet broadcast)
         then
4          broadcast bVAL(vJ) /* a processor echoes a value only once          */
5      if (bVAL(vJ) received from (2t+1) different processors) then
6          binValues ← binValues ∪ {vJ} /* local delivery of a value           */
7  operation propose(v) begin
8      (est, r) ← (v, 0);
9      do forever begin
10         r ← r + 1;
11         bvBroadcast EST[r](est);
12         wait(binValues[r] ≠ ∅) ;    /* binValues[r] has not necessarily obtained its
             final value when wait returns */
13         broadcast AUX[r](w) where w ∈ binValues[r];
14         wait ∃ 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 ∧
             vals ⊆ binValues[r];
15         s[r] ← 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 ← v;
20         else est ← s[r];
```

3.1 The MMR algorithm

Algorithm 1 presents the MMR algorithm [65], which considers an underlying communication abstraction named BV-broadcast. Recall that the set *Correct* denotes the set of processors 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, bvBroadcast(v), assumes that all the correct processors invoke bvBroadcast(w), where $v, w \in \{0, 1\}$.

Task definition The set of values that are BV-delivered to processor 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 $v \in binValues_i$ and p_i is correct. It holds that v has been BV-broadcast by a correct processor.
- **BV-uniformity.** $v \in binValues_i$ and p_i is correct. Eventually $\forall j \in Correct : v \in binValues_j$.
- **BV-termination.** Eventually $\forall i \in Correct : binValues_i \neq \emptyset$ holds.

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

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 processors, vJ is replayed via broadcast (but only if this was not done earlier). Also, if vJ was received by at least $2t+1$ different processors, 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 processor 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 processor deterministically select a value from the set $binValues$ and by that contradict FLP [46].

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 2.5, the operation randomBit(r) retrieves the value of the common coin on round r. The set $vals \subseteq \{0,1\}$ holds the value received during the current round. Recall that processor $p_i \in \mathcal{P}$ stores the binary values received in a round r via a bvBroadcast() in the read-only set $binValues_i[r]$.

Detailed description MMR's main algorithm (appearing as operation propose(v)) 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 termination 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 bvBroadcast() protocol. Due to the BV-termination property, eventually, the set *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 processor.

- *Inform about the query results (lines 13–14)*: The auxiliary message, $AUX(w)$, carrying the value of $binValues[r]$ is broadcast. Note that all the correct processors, p_j, broadcast $w \in values_j[r]$, *i.e.,* a value that is estimated by at least one correct processor. However, arbitrary binary values can be broadcast by the Byzantine processors.

  Processor p_i then waits for the arrival of $AUX(w)$ messages from $n - t$ distinct processors, 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 processors, cf. $vals_i \subseteq binValues_i[r]$ at line 14.
  - Guarantee that, for a given round r, it holds that $\exists i \in Correct : vals_i = \{v\} \implies \forall j \in Correct : v \in vals_j$. 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 processor.

- *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 common coin and the processor has not yet decided. If $vals$ contains both of the binary values, the common coin output serves as the estimated value for the next round. Note that deciding on a value does not mean that any processor can stop executing Algorithm 1. (The non-self-stabilizing version of MMR can be found in [65].)

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 processors estimate the same value during round r, then $\exists x \in \{0, 1\} : \forall i \in Correct : x \in binValues[r]_i$ holds, which means that $\exists x \in \{0, 1\} : \forall i \in Correct : vals_i = \{x\}$ holds during round r. Moreover, the proof of MMR [65] shows that $\exists x \in \{0, 1\} : \forall i \in Correct : vals_i = \{x\}$ holds for any round $r' \geq r$. Thus, the decision of x depends only on the value of the common coin. In other words, the common coin has the "correct value" with probability 1/2 and the algorithm decides.

Now suppose that, for any reason, $\exists x \in \{0, 1\} : \forall i \in Correct : vals_i = \{x\}$ does not hold during round r. Then, any processor that decides on round r decides the value of the common coin. Also, the ones that do not decide on round r, since $vals = \{0, 1\}$, estimate for round $r + 1$ the value of the common coin. Therefore, BC-agreement holds in this case. Moreover, all the processors for which $vals = \{0, 1\}$ 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 processors have the same estimated value. As discussed above, this state leads to agreement with probability 1/2. More details can be found in [65].

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 stepping stone 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.

Algorithm 2: Non-self-stabilizing Byzantine-tolerant 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] ← ({v}, ⊥)};
22 result() do {if (est[M+1][i] = {v}) then return v else if (r ≥ M ∧ infoResult() ≠ ∅)
     then returnΨelse return ⊥;}
23 macros: binValues(r, x) return {y ∈ {0,1} : ∃s ⊆ P : |{p_j ∈ s : y ∈ est[r][j]}| ≥ x};
24 infoResult() do {if (∃s ⊆ P : n−t ≤ |s| ∧ (∀p_j ∈ s : aux[r][j] ∈ binValues(r, 2t+1))) then
   return {aux[r][j]}_{p_j∈s} else return ∅};
25 functions: decide(x) begin
26   if (est[M+1][i] = ∅ ∨ aux[M+1][i] = ⊥) then (est[M+1][i], aux[M+1][i]) ← ({x}, x);
27 tryToDecide(values) begin
28   if (values ≠ {v}) then est[r][i] ← {randomBit(r)};
29   else {est[r][i] ← {v}; if (v = randomBit(r)) then decide(v)};
30 do forever begin
31   if (est[0][i] ≠ ∅) then
32     r ← min{r+1, M};
33     repeat
34       foreach p_j ∈ P do send EST(True, r, est[r−1][i] ∪ binValues(r, t+1)) to p_j
35       if (∃w ∈ binValues(r, 2t+1)) then aux[r][i] ← w;
36     until aux[r][i] ≠ ⊥;
37     repeat
38       foreach p_j ∈ P do send AUX(True, r, aux[r][i]) to p_j
39     until infoResult() ≠ ∅;
40     tryToDecide(infoResult());
41 upon EST(aJ, rJ, vJ) arrival from p_j do begin
42   est[rJ][j] ← est[rJ][j] ∪ 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 ≠ ⊥) then aux[rJ][j] ← vJ;
46   if (aJ) then send AUX(False, rJ, aux[rJ][i]) to p_j;
```

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 $est_i[r][i]$ its estimated decision values, where $est_i[0][i] = \{v\}$ stores its own proposal and $est_i[M+1][i]$ aims to hold the decided value. Since nodes exchange these estimates, $est_i[r][j]$ stores the last estimate that p_i received from p_j. Note that $est_i[r][j] \subseteq \{0, 1\}$ holds a set of values and it is initialized by the empty set, $\emptyset$. At the end of round r, processor $p_i \in \mathcal{P}$ tests whether it is ready to decide after it selects a single value

$w \in est_i[r][i]$ to be exchanged with other processors. In order to ensure reliable broadcast in the presence of packet loss, there is a need to store w in auxiliary storage, $aux_i[r][i]$, so that p_i can retransmit w. Note that all entries in $aux[][]$ 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 processors in the system and then waiting for a reply from the maximum number of processors 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 [29, Chapter 3]. In the context of self-stabilization and quorum systems, Dolev, Petig, and Schiller [36] 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 processors that are able to complete many round-trips with a given sender, while the other processors in $\mathcal{P}$ accumulate messages in their communication channels, 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.* [33, 30]. 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 processors eventually receive messages from at least $n - t$ processors (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 $\text{AUX}(\bullet, rnd = r, aux = w) : aux_j[r][j] = w$ from distinguishable senders. This is because line 38 broadcasts the message $\text{AUX}(ack = \mathsf{True}, rnd = r, \bullet)$ and upon its arrival to p_j, line 46 replies with $\text{AUX}(ack = \mathsf{False}, rnd = r, \bullet)$. Note that duplication is not a challenge since, for a given round number r, p_j always sends the same $\text{AUX}(\bullet, rnd = r, aux = w) : aux_j[r][j] = w$ message. Algorithm 2 deals with packet reordering 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, processor p_i processes 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, processor 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 EST$(\bullet, rnd = r, est = \{v, \bullet\})$ message from p_i to p_j (line 42), v is always present in $est_j[r-1][i]$. 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 EST$(\bullet, rnd = r, est = \{v, \bullet\})$ 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 $est_i[0][i]$ 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(). Lines 27 to 29 are the implementation of tryToDecide(). 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() 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() output. The actual decision action (line 26) is for both $est[M+1][i]$ and $aux[M+1][i]$ to be assigned the decided value.

As specified in Section 2.1, the function result() (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 processor 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 processors 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 $\mathsf{result}_i() \in \{0,1\} : p_i \in \mathcal{P} : i \in Correct$.*

Proof Sketch of Lemma 3.1 The proof uses Claim 3.2.

Claim 3.2 *$\exists v \in \{0,1\} : \forall i \in Correct : est_i[r][i] = \{v\}$ 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 $\forall k \in Correct : values_i^r = \{0,1\}$ or $\forall k \in Correct : values_i^r = \{v_k(r)\}$ hold, p_k assigns the same value to $est_k[r][k]$, which is $\{\text{randomBit}_k(r)\}$, and respectively, $v_k(r)$. The remaining case is when some correct processors assign $\{v_k(r)\}$ to $est_k[r][k]$ (line 29), whereas others assigns $\{\text{randomBit}_k(r)\}$ (line 28).

Recall the assumption that the Byzantine processors have no control over the network or its scheduler. Due to the common coin 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 $[\exists r' \leq r : \text{randomBit}(r) = v(r)] = \frac{1}{2} + (1-\frac{1}{2})\frac{1}{2} + \cdots + (1-\frac{1}{2})^{r-1}\frac{1}{2} = 1 - (\frac{1}{2})^r$. $\square_{Claim\ 3.2}$

The complete proof shows that the repeat-until loop in lines 33 to 36 cannot block forever and that all the correct processors p_i keep their estimate value $est_i = \{v\}$ and consequently the predicate $(values_i^{r'} = \{v\})$ at line 28 holds for round r', where $values_i^{r'} = \cup_{j \in s}\{aux_i[r][j]\}$. 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 processors decide v. $\square_{Lemma\ 3.1}$

Remark 3.1 (safety in practical settings) *Lemma 3.1 shows 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 for 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 processor, since each processor 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 The Proposed Self-stabilizing Solution

Algorithm 3 presents a solution that can recover from transient-faults. We demonstrate the correctness of that solution in Section 5. The boxed lines in Algorithm 3 are relevant only to an extension (Section 6) 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 terminate, *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 [65] 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 processor $p_i \in \mathcal{P}$ by, for example, setting $est_i[i]$ with $\{0, 1\}$. Line 63 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.,* $\exists r' \in \{1, \ldots, r-1\} : est_i[r'][i] = \emptyset \vee aux_i[r'][i] = \perp$. Line 65 addresses this concern. Since we wish not that the for-each condition in line 64 to hold when a correct processor decides, line 54 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 62 to 69 the repeat-until loops of lines 33 to 36 and 37 to 39 of Algorithm 2. Similarly, it combines in lines 72 to 74 of the upon events in lines 41 to 43 and lines 44 to 46 of Algorithm 2.

5 Correctness

The correctness proof shows that the solution presented in Section 4 recovers from transient-faults without blocking (Section 5.1) and that any consensus operation always satisfies the liveness requirements of Definition 1.1 (Section 5.2). Also, it satisfies the safety requirements of Definition 1.1 in the way that loosely-self-stabilizing systems do (Section 5.2), *i.e.,* any consensus operation satisfies the requirements of Definition 1.1 with probability $\Pr(r) = 1 - (1/2)^{-M}$.

5.1 Transient fault recovery

We say that a system state c is *resolved* if $\forall i \in \mathit{Correct} : \left|est_i[0][i]\right| \in \{0, 1\} \wedge \nexists r' \in \{1, \ldots, r-1\} : est_i[r'][i] = \emptyset \vee aux_i[r'][i] = \perp$ and no communication channel that goes out from $p_i \in \mathcal{P} : i \in \mathit{Correct}$ to any other correct processor includes $\text{EST}(rnd = r, est = W, aux = w) : r_i < r \vee W \nsubseteq est_i[r][i] \vee (w \neq \perp \wedge w \notin W)$ messages. Suppose that during execution R, every correct processor $p_i \in \mathcal{P}$

Algorithm 3: Loosely-self-stabilizing Byzantine-tolerant binary consensus that uses M iterations and violates safety with a probability that is in $\mathcal{O}(1/2^M)$; code for p_i.

```
47 constants: initState := (0, [[∅, ..., ∅], ..., [∅, ..., ∅]], [[⊥, ..., ⊥], ..., [⊥, ..., ⊥]]);
48 operations: propose(v) do {(r, est, aux) ← initState; est[0][i] ← {v}};
49 result() do {if (est[M+1][i] = {v}) then return v else if (r ≥ M ∧ infoResult() ≠ ∅)
     then returnΨelse return ⊥;}
50 macros: binValues(r, x) return {y ∈ {0,1} : ∃s ⊆ P : |{p_j ∈ s : y ∈ est[r][j]}| ≥ x};
51 infoResult() do {if (∃s ⊆ P : n−t ≤ |s| ∧ (∀p_j ∈ s : aux[r][j] ∈ binValues(r, 2t+1))) then
   return {aux[r][j]}_{p_j∈s} else return ∅;}
52 functions: decide(x) begin
53   foreach r' ∈ {r, ..., M+1} do
54     if (est[r'][i] = ∅ ∨ aux[r'][i] = ⊥) then (est[r'][i], aux[r'][i]) ← ({x}, x);
55   [r ← M+1;]
56 tryToDecide(values) begin
57   if (values ≠ {v}) then est[r][i] ← {randomBit(r)};
58   else {est[r][i] ← {v}; if (v = randomBit(r)) then decide(v)};
59 do forever begin
60   if ((r, est, aux) ≠ initState) then
61     r ← min{r+1, M[+1]};
62     repeat
63       if (est[0][i] ≠ {v}) then est[0][i] ← {w} : ∃w ∈ est[0][i];
64       foreach r' ∈ {1, ..., r−1} : est[r'][i] = ∅ ∨ aux[r'][i] = ⊥ do
65         (est[r'][i], aux[r'][i]) ← (est[0][i], x) : x ∈ est[0][i];
66       if ((∃w ∈ binValues(r, 2t+1) ∧ (aux[r][i] = ⊥ ∨ aux[r][i] ∉ binValues(r, 2t+1)))
          then
67         aux[r][i] ← w;
68       foreach p_j ∈ P do send EST(True, r, est[r−1][i] ∪ binValues(r, t+1), aux[r][i])
          to p_j
69     until infoResult() ≠ ∅;
70     tryToDecide(infoResult());
71     [if (∃w ∈ binValues(M+1, t+1)) then decide(w);]
72 upon EST(aJ, rJ, vJ, uJ) arrival from p_j begin
73   est[rJ][j] ← est[rJ][j] ∪ vJ; aux[rJ][j] ← uJ;
74   if aJ then send EST(False, rJ, est[rJ−1][i], aux[r][i]) to p_j;
```

invokes $\mathsf{propose}_i()$ exactly once. In this case, we say that R includes a *complete invocation* of binary consensus. Theorem 5.1 shows recovery to resolved system states and termination during executions

that include a complete invocation of binary consensus. The statement of Theorem 5.1 uses the term *active* for processor $p_i \in \mathcal{P}$ when referring to the case of $est_i[0][i] \neq initState$.

Theorem 5.1 (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 processors are active. (ii) Within $\mathcal{O}(M)$ asynchronous (communication) rounds, for every correct processor $p_i \in P$, it holds that the operation $\mathsf{result}_i()$ returns $v \in \{0, 1, \Psi\}$, where Ψ is the transient error symbol.*

Proof of Theorem 5.1 Lemmas 5.2 and 5.4 demonstrate the theorem.

Lemma 5.2 *Invariant (i) holds.*

Proof of Lemma 5.2 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 60 to 70) after m has left the system. Once that happens, for any message $\text{EST}(rnd = r, est = W, aux = w)$ that appears in any communication channel that is going out from $p_i \in \mathcal{P} : i \in Correct$, it holds that $r \leq r_i \wedge W \subseteq est_i[r][i] \wedge (w = \perp \vee w \in W)$ (due to lines 48 and 68).

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 $\forall i \in Correct : (r_i, est_i, aux_i) = initState$. In this case, Invariant (i) holds by definition. In case $\forall i \in Correct : (r_i, est_i, aux_i) = initState$ does not hold, lines 63 to 65 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 57 processor p_i follows, $est_i[r][i]$ is always assigned a value that is not the empty set at the end of round r, cf. lines 57 and 58. Moreover, the assignment of w to $aux_i[r][i]$ in line 67 is always of a value that is not the empty set due to the if-statement condition in line 66 and the definition of $binValues()$ (line 50). $\square_{Lemma\ 5.2}$

Lemma 5.3 is needed for the proof of Lemma 5.4.

Lemma 5.3 *Suppose that R's states are resolved (Lemma 5.2). The repeat-until loop in lines 66 to 69 cannot block forever.*

Proof of Lemma 5.3 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 $aux_i[r][i] \in binValues_i(r, 2t{+}1)$ holds.* Suppose that in R's starting system state, $(aux[r][i] \neq \perp \wedge aux[r][i] \in binValues(r, 2t{+}1))$ 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 processors and each of them sends $\text{EST}(\bullet, rnd = r, est = \{w, \bullet\}, \bullet) : w \in \{0, 1\}$ messages to all processors (line 68). Therefore, we know that there is $v \in \{0, 1\}$, such that at least $(t{+}1)$ correct processors send $\text{EST}(\bullet, rnd = r, est = \{v, \bullet\}, \bullet)$ messages to all other processors.

Since every correct processor receives $\text{EST}(\bullet, rnd = r, est = \{v, \bullet\}, \bullet)$ from at least $(t{+}1)$ processors (line 74), we know that eventually every correct processor relays the value v via the message $\text{EST}(\bullet, rnd = r, est = \{v, \bullet\}, \bullet)$ that line 68 sends due to the fact that $v \in binValues_i(r, t{+}1)$.

Since $n-t \geq 2t+1$ holds, we know that the clause $(\exists w \in binValues(r, 2t+1))$ in the if-statement condition at line 66 is eventually satisfied at each correct processor $p_i \in \mathcal{P}$. Thus, if $(aux[r][i] = \perp \vee aux[r][i] \notin binValues(r, 2t+1))$ does not hold, line 67 makes sure it does.

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

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 $\exists i \in Correct : w \in binValues_i(r_i, 2t+1)$ holds in every system state in R and yet $\forall s \subseteq Correct : t+1 \leq |s|$, it is true that $\exists k \in s : w \notin est_k[k]$.

By lines 68 and 73, the only way in which $w \in binValues_i(r_i, 2t+1)$ hold in every system state $c' \in R$, is if there is a system state c that appears in R before c', such that $\exists s \subseteq Correct : t+1 \leq |s| : \forall k \in s : w \in est_k[k]$. 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* $\exists s \subseteq Correct : t+1 \leq |s| \wedge \forall p_k \in s : w \in est_k[k] \implies \forall i \in Correct : w \in binValues_i(r_i, 2t+1)$.

By line 68 and the argument's assumption, there are at least $(t+1)$ correct processors that send $\text{EST}(\bullet, rnd = r, est = \{w, \bullet\}, \bullet)$ messages to all (correct) processors. Since every correct processor receives w from at least $(t+1)$ processors (line 73), every correct processor eventually reply w via the message $\text{EST}(\bullet, rnd = r, est = \{w, \bullet\}, \bullet)$ at lines 68 and 74 due to the fact that $w \in binValues_i(r, t+1)$. Since $n-t \geq 2t+1$ holds, we know that $(\exists w \in binValues(r, 2t+1))$ holds and the argument is true.

Argument 4: *Suppose that the condition* $cond(i) := \mathsf{infoResult}_i() \neq \emptyset : i \in Correct$ *does not hold in R's starting system state. Eventually, the system reaches a state* $c'' \in R$*, in which* $cond(i) : i \in Correct$ *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 $cond(i)$ never holds, *i.e.,* $c'' \in R$ does not exist. We note that $cond(i)$ must hold if $binValues_i(r_i, 2t+1) = \{0, 1\}$. The same can be said for the case of $binValues_i(r_i, 2t+1) = \{v\} \wedge \exists s \subseteq \mathcal{P} : n-t \leq |s| \wedge (\cup_{p_k \in s}\{aux_i[r][k]\}) = \{w\} \wedge w = v$. Therefore, we assume that, for any system state, it holds that $binValues_i(r_i, 2t+1) = \{v\} \subsetneq \{0, 1\}$ 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 $w \in binValues_i(r_i, 2t+1)$.

By lines 68 and 73, 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, $aux_k[r][k] \in binValues_k(r, 2t+1)$. By Argument 2, $w \in binValues_k(r_i, 2t+1) \implies \exists s \subseteq Correct : t+1 \leq |s| \wedge \forall p_k \in s : w \in est_k[k]$ in c. By Argument 3, $\exists s \subseteq Correct : t+1 \leq |s| \wedge \forall k \in s : w \in est_k[k] \implies \forall i \in Correct : w \in binValues_i(r_i, 2t+1)$ 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 69 eventually holds. $\square_{Lemma\ 5.3}$

Lemma 5.4 *Invariant (ii) holds.*

Proof of Lemma 5.4 Lemma 5.2 shows that R's system states are resolved. Lemma 5.3 says that the repeat-until loop in lines 66 to 69 does not block. By line 61 and the definition of an asynchronous (communication) round (Section 2.4), every iteration of the do-forever loop (lines 60 to 70) can be associated with at most one asynchronous (communication) round. Thus, line 49 and Argument (4) of the proof of Lemma 5.3 imply that $(r_i \geq M \wedge \mathsf{infoResult}_i() \neq \emptyset)$ holds within $\mathcal{O}(M)$ asynchronous (communication) rounds. Therefore, $\mathsf{result}_i()$ returns a non-$\perp$ value within $\mathcal{O}(M)$ asynchronous (communication) rounds. $\square_{Lemma\ 5.4}$ $\square_{Theorem\ 5.1}$

5.2 Satisfying the task specifications

We say that the system state c is *well-initialized* if $\forall i \in Correct : (r_i, est_i, aux_i) := initState$ holds and no communication channel between two correct processors includes EST() messages. Note that a well-initialized system state is also a resolved one (Section 5.1). Theorem 5.6 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 5.6 uses Theorem 5.5, which demonstrates that Algorithm 3 satisfies the requirements of Definition 5.1, which adds more details to the one given in Section 3.1.1. Recall that the operation $\mathsf{bvBroadcast}(v)$ of Algorithm 1 is embedded in the code of Algorithm 3.

Definition 5.1 (BV-broadcast) *Let $p_i \in \mathcal{P}$, $r \in \{1, \ldots, M\}$, and $v \in \{0, 1\}$. Suppose that $r_i = r \wedge est_i[r-1][i] = \{v\}$ holds immediately before p_i executes line 68. In this case, we say that p_i BV-broadcast v during round r in line 68. Let $c \in R$ and suppose that $w \in binValues_i(r, t+1)$ 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 $v \in binValues_i(r, t+1)$ holds in system state $c \in R$. Then, before c there is a step in R in which a correct processor BV-broadcast v.*
- ***BV-uniformity.*** *Suppose that p_i is correct and $v \in binValues_i(r, t+1)$ holds in system state $c \in R$. Then, eventually, the system reaches a state in which $\forall j \in Correct : v \in binValues_j(r, t+1)$ holds.*
- ***BV-termination.*** *Eventually, the system reaches a state in which $\forall i \in Correct : binValues_i(r, t+1) \neq \emptyset$ holds.*

Theorem 5.5 (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 66 to 69 and lines 72 to 74 of Algorithm 3 implement the BV-broadcast task (Definition 5.1).*

Proof of Theorem 5.5 We prove that the requirements of Definition 5.1 hold.
BV-validity. Suppose that, during round r, merely faulty processors BV-broadcast v. We show that $\nexists c \in R$, such that $(\exists v \in binValues_i(r, 2t+1))$ holds in c. Since only faulty processors BV-broadcast v, then no correct processor receive EST(-, $rnd = r, est = \{v, \bullet\}, \bullet$) messages from more than t different senders. Consequently, $v \notin binValues_i(r, t+1)$ in line 68 at any correct processor $p_i \in \mathcal{P}$. Similarly, no correct processor $p_i \in \mathcal{P}$ can satisfy the predicate $(\exists w \in binValues(r, 2t+1))$ at line 69 (via line 51). Thus, the requirement holds.

BV-uniformity. Suppose that $w \in binValues_i(r, 2t{+}1)$ holds in c. By lines 50 and 66 we know that p_i stores v in at least $(2t{+}1)$ entries of $EST[r][]$. Since R starts in a well-initialized system state, this can only happen if p_i received EST$(\bullet, rnd = r, est = \{v, \bullet\}, \bullet)$ messages from at least $(2t{+}1)$ different processors (line 72). This means that p_i received this message from at least $(t{+}1)$ different correct processors. Since each of these correct processors sent the message EST$(\bullet, rnd = r, est = \{v, \bullet\}, \bullet)$ to any processor in $\mathcal{P}$, we know that $\forall j \in Correct : binValues_j(r, t{+}1) \neq \emptyset$ (line 68) holds eventually. Therefore, every correct processor p_j sends EST$(\bullet, rnd = r, est = \{v, \bullet\}, \bullet) : v \in binValues_i(r, t{+}1)$ to all. Since $n{-}t \geq 2t{+}1$, we know that $(\exists w \in binValues_k(r, 2t{+}1))$ holds eventually at each correct processor, $p_k \in \mathcal{P}$.
BV-termination. This requirement is implied by Lemma 5.3. $\square_{Theorem\ 5.5}$

Theorem 5.6 (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 processor $p_i \in \mathcal{P}$, the operation $\mathsf{result}_i()$ returns $v \in \{0, 1\}$.*

Proof of Theorem 5.6 Lemmas 5.7 to 5.11 show the proof. Lemma 5.7 shows that once all correct nodes estimate the same value in a round r, they hold on this estimate in all subsequent rounds. Lemma 5.8 shows that correct nodes that pass a singleton to tryToDecide(), pass the same set. Lemma 5.9 shows that correct nodes can only decide a value that has been previously proposed by a correct node. Lemma 5.11 shows that correct nodes have $v \in \{0, 1\}$ as a return value from 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 5.6.

Lemma 5.7 *Suppose that every correct processor, $p_i \in \mathcal{P}$, estimates value v upon entering round r,* i.e., *$est_i[r{-}1][i] = \{v\} \wedge r_i = r{-}1$ immediately before executing line 60. Then, p_i estimates the value v in any round later than r,* i.e., *$r' \in \{r, \ldots, M\} : est_i[r'][i] = \{v\}$.*

Proof of Lemma 5.7 There are $n - t > t + 1$ correct processors. By the lemma statement, all of them broadcast EST$(\bullet, rnd = r, est = \{v\}, \bullet)$ (line 68). Thus, $binValues_i(r, 2t{+}1) = \{v\}$ (BV-termination and BV-validity, Theorem 5.5) and $values_i^r = \{v\}$ (lines 57), where $values_i^r$ is the parameter that p_i passes to tryToDecide() (line 56) during round r, cf. $values_i^r = \cup_{j \in s}\{aux[r][j]\}$ (line 70 via line 51).

Therefore, $est_i[r][i] = \{v\}$ holds due to the assignment in the start of line 58. Since there are most t Byzantine processors, and for an estimate to be forwarded (and hence accepted) it needs a "support" of $t + 1$ processors (line 68), it follows that the correct processors cannot change their estimate in any round $r' \geq r$. $\square_{Lemma\ 5.7}$

Lemma 5.8 *Suppose that there is a system state $c \in R$, such that $(values_i^r = \{v\}) \wedge (values_j^r = \{w\})$, where $p_i, p_j \in \mathcal{P}$ are two correct processors and $values_i^r$ is the parameter that p_i passes to* tryToDecide() *(line 56) during round r. It holds that $v = w$ in c.*

Proof of Lemma 5.8 Due to the exit condition of the repeat-until loop in lines 66 to 69, p_i had to receive before c identical EST$(\bullet, rnd = r, \bullet, aux = v, \bullet)$ messages from at least $(n{-}t)$ different processors. Since at most t processors are faulty, $(n{-}t) = (n{-}2t)$, which means that p_i received

EST($\bullet, rnd = r, \bullet, aux = v, \bullet$) messages before c from at least $(t{+}1)$ different correct processors, as $n{-}2t \geq t{+}1$. Using the symmetrical arguments, we know that p_j had to receive before c identical EST($\bullet, rnd = r, \bullet, aux = w, \bullet$) messages from at least $(n{-}t)$ different processors.

Since $(n{-}t){+}(t{+}1) > n$, the pigeonhole principle implies the existence for at least one correct processor, $p_x \in \mathcal{P}$, from which from p_i and p_j have received the messages EST($\bullet, rnd = r, \bullet, aux = v, \bullet$) and EST($\bullet, rnd = r, \bullet, aux = w, \bullet$), respectively. The fact that p_x is correct implies that it has sent the same EST($\bullet, rnd = r, \bullet$) message to all the processors in line 68. Thus $v = w$. $\square_{Lemma\ 5.8}$

Lemma 5.9 *Suppose that there is a system state $c \in R$, such that* $\mathsf{result}_i() = v \in \{0, 1\}$ *in c, where $p_i \in \mathcal{P}$ is a correct processor. There is a correct processor $p_j \in \mathcal{P}$ and a step $a_j \in R$ (between R's starting system state and c) in which p_j invokes* $\mathsf{propose}_j(v)$.

Proof of Lemma 5.9 Suppose that $r_i = 1$. Recall (a) the BV-validity property (Theorem 5.5 and line 68), observe (b) the if-statement condition in line 66, which selects the value $w_i \in binValues(1, 2t{+}1)$ (line 67) as well as the exit condition in line 69 of the repeat-until loop in lines 66 to 69 in which (c) correct processors, $p_j \in \mathcal{P}$, broadcast EST($\bullet, rnd = 1, \bullet, aux = w_j, \bullet$) : $w_j \in binValues(1, t{+}1)$ messages. Thus, the set $values_i^1$ includes only values arriving from correct processors, where $values_i^r$ is the parameter that p_i passes to $\mathsf{tryToDecide}()$ (line 56) during round r.

Processor p_i can decide v (line 58) whenever $values_i^1 = \{v\} \wedge v = \text{randomBit}_i(r_i)$ holds. Regardless of the decision, p_i updates its new estimate (line 58). Processor p_i updates its estimate $est_i[r_i][i]$ with the value, randomBit(r)), obtained by the common coin (line 57) whenever $values_i^1 = \{0, 1\}$. This means, that p_i updates the estimated value with a value that a correct processor has proposed. Note that the $values_i^1 = \{0, 1\}$ case occurs when both 0 and 1 were proposed by correct processors. The same arguments hold also for round numbers $r > 1$, and therefore, a decided value must be a value proposed earlier by a correct processor p_j, where $i = j$ can possibly hold. $\square_{Lemma\ 5.9}$

Lemma 5.10 *Suppose that there is system state $c \in R$, such that* $\mathsf{result}_i(), \mathsf{result}_j() \notin \{\perp, \Psi\}$ *holds in c, where $p_i, p_j \in \mathcal{P}$ are correct processors. It holds that* $\mathsf{result}_i() = \mathsf{result}_j()$.

Proof of Lemma 5.10 Suppose, without the loss of generality, that processor p_i is the first correct processor that decides during R and it does so during round r. Suppose that there is another processor, 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 58 and the properties of the common coin. We also know that p_i and p_j update their estimates in $est_x[r][x] : x \in \{i, j\}$ to $\text{randomBit}_x(r)$.

Recall that $values_i^r$ denotes the parameter that p_i passes to $\mathsf{tryToDecide}()$ (line 57) during round r. Lemma 5.8 says that $(values_i^r = \{v\}) \wedge (values_j^r = \{w\})$ 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 $values_j^r = \{v, w\} = \{0, 1\}$, see lines 57 to 58 and the proof of Lemma 5.8. Therefore, p_j assigns randomBit(r) to $est_j[r_j][j]$ (line 57). This means that every correct processor starts round $(r + 1)$ with $est_j[r_j][j] = \text{randomBit}(r)$ and randomBit(r) $= v$. Lemma 5.7 says that this estimate never change, and thus, only v can be decided. $\square_{Lemma\ 5.10}$

Lemma 5.11 *By the end of round $r \leq M$, for each correct processor $p_i \in \mathcal{P}$, the operation* $\mathsf{result}_i()$ *returns $v \in \{0, 1\}$ with probability* $\Pr(r) = 1 - (1/2)^r$.

Proof of Lemma 5.11 The proof uses Claim 5.12.

Claim 5.12 *Let $c_r \in R$ be the state that the system reaches at the end of round $r \leq M$. With probability* $\Pr(r) = 1 - (1/2)^r$, $\exists v \in \{0,1\} : \forall i \in Correct : est_i[r][i] = \{v\}$ *holds in* c_r.

Proof of Claim 5.12 Let $values_i^r$ be the parameter that p_i passes to tryToDecide() (line 56) on round r.

- **Case 1:** Suppose that the if-statement condition $values_i^r = \{v_k(r)\}$ (line 57) holds for all correct processors $p_k \in \mathcal{P}$. Similarly to the proof of Lemma 5.10, any correct processor p_k assigns to $est_k[r][k]$ the same value, $v_k(r)$ (line 29).
- **Case 2:** Suppose that the if-statement condition $values_i^r = \{v_k(r)\}$ (line 57) does not hold for all correct processors $p_k \in \mathcal{P}$. By similar arguments as in the previous case, any correct p_k assigns to $est_k[r][k]$ the same value, $\{\text{randomBit}_k(r)\}$ (line 28).
- **Case 3:** Some correct processors assign $\{v_k(r)\}$ to $est_k[r][k]$ (line 58), whereas others assign $\{\text{randomBit}_k(r)\}$ (line 57).

The rest of the proof focuses on Case 3. Recall the assumption that the Byzantine processors 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 common coin properties, see Section 2.5), 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, $\Pr(r) = \frac{1}{2} + (1 - \frac{1}{2})\frac{1}{2} + \cdots + (1 - \frac{1}{2})^{r-1}\frac{1}{2} = 1 - (\frac{1}{2})^r$. $\square_{Claim\ 5.12}$

Recall that Lemma 5.3 says that the repeat-until loop in lines 66 to 69 cannot block forever. It follows from Lemma 5.7 and Claim 5.12 that all the correct processors p_i keep their estimate value $est_i = v$ and consequently the predicate $(values_i^{r'} = \{v\})$ at line 57 holds for round r', where $values_i^{r'} = \cup_{j \in s}\{aux_i[r][j]\}$. With probability $\Pr(r) = 1 - (1/2)^r$, by round r, $\text{randomBit}(r) = v$ holds due to the common coin properties. Then, the if-statement condition of line 57 does not hold and the one in line 58 does hold. Thus, all the correct processors decide v. $\square_{Lemma\ 5.11}$ $\square_{Theorem\ 5.6}$

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

Theorem 5.13 *Let R be an execution of Algorithm 3 that starts in a well-initialized system state and during which every correct processor $p_i \in \mathcal{P}$ 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 processors are expected to decide.*

Proof of Theorem 5.13 We divide the proof into four arguments.
Argument 1: *BC-termination is always guaranteed.* Lemma 5.3 and 5.11 demonstrate BC-termination when starting from an arbitrary, and respectively, a well-initialized system state.
Argument 2: *Suppose that, for every correct processor $p_i \in \mathcal{P}$, 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 5.9, 5.10, and 5.11 imply BC-validity and BC-agreement as long as $\forall i \in Correct$: $\text{result}_i()$ returns $v \in \{0,1\}$.

Argument 3: *Algorithm 3 satisfies the design criteria of Definition 2.2.* Theorem 5.1 demonstrates that any complete invocation of binary consensus terminates within a finite number of steps. Once that happens, the next well-initialized invocation of propose() can succeed independently of previous invocations. Argument 1 and Lemma 5.11 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 processors are expected to decide within four iterations of Algorithm 3.* The proof of Claim 5.12 considers two stages when demonstrating BC-termination (after starting from a well-initialized system state). That is, all correct processors need to first use the same value, v, as their estimated one, see the assignment to $est_i[r][i]$ in lines 58 to 57. Then, each correct processor waits until the next round in which the condition, $v_i = \text{randomBit}_i(r_i)$, of the if-statesmen in line 58 holds, where randomBit() is the interface to the common coin. 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 5.12 reveals the case in which not all correct processors use the same value (Case 3). This is when the condition, $values = \{v\}$, of the if-statement in line 58 is true but not for any correct processor $p_i \in \mathcal{P}$. We show how to bound by two the number of asynchronous rounds in which this situation can happen. Suppose that $values_i^r \neq \{v\}$. Note that, with probability 1/2, the assignment in line 57 sets the value $\{v\}$ to $est_i[r][i]$. 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 common coin properties (Section 2.5), $\Pr(v_i = \text{randomBit}_i(r_i)) = 1/2$ and $E(\Pr(v_i = \text{randomBit}_i(r_i))) = 2$. $\square_{Theorem\ 5.13}$

6 Extension: Eventually Silent Loosely-Self-stabilization

Self-stabilizing systems can never stop the exchange of messages until the consensus object is deactivated, see [29, 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 $t+1$ processors have decided, lets all correct processors 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 [37].

The extension idea is to let processor p_i to wait until at least $t+1$ processors have decided. Once that happens, p_i can notify all processors about this decision because at least one of these $t+1$ processors 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 55) and allowing r to have the value of up to $M+1$ (line 61). Also, line 71 decides value w whenever it sees that it was decided by $t+1$ other processors since at least one of them must be correct.

7 Discussion

We have presented a new loosely-self-stabilizing variation on the MMR algorithm [65] for solving binary consensus in the presence of Byzantine failures in message-passing systems. 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 termination is the same as the studied algorithm. It also to some extent preserves the expected communication costs. 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 safety property 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. We believe that this work is preparing the groundwork needed to construct self-stabilizing (Byzantine fault-tolerant) algorithms for distributed systems, such as Blockchain, that need to run in an externally hostile environment.

References

[1] 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.

[2] 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.

[3] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In André Schiper, editor, *Distributed Algorithms, 7th International Workshop, WDAG '93, Lausanne, Switzerland, September 27-29, 1993, Proceedings*, volume 725 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 1993.

[4] James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *J. ACM*, 45(3):415–450, 1998.

[5] Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors. *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*. Springer, 2019.

[6] 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.

[7] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30. ACM, 1983.

[8] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing Byzantine tolerant digital clock synchronization. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *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.

[9] 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 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 355–362. IEEE Computer Society, 2014.

[10] 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 Borzoo Bonakdarpour and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings*, volume 10083 of *Lecture Notes in Computer Science*, pages 36–53, 2016.

[11] Alexander Binun, Shlomi Dolev, and Tal Hadad. Self-stabilizing Byzantine consensus for blockchain - (brief announcement). In Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung, editors, *Cyber Security Cryptography and Machine Learning - Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27-28, 2019, Proceedings*, volume 11527 of *Lecture Notes in Computer Science*, pages 106–110. Springer, 2019.

[12] Silvia Bonomi, Shlomi Dolev, Maria Potop-Butucaru, and Michel Raynal. Stabilizing server-based storage in Byzantine asynchronous message-passing systems: Extended abstract. In Georgiou and Spirakis [50], pages 471–479.

[13] Silvia Bonomi, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing Byzantine-fault tolerant storage. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 894–903. IEEE Computer Society, 2015.

[14] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal mobile Byzantine fault tolerant distributed storage: Extended abstract. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 269–278. ACM, 2016.

[15] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal storage under unsynchronized mobile Byzantine faults. In *36th IEEE Symposium on Reliable Distributed Systems, SRDS 2017, Hong Kong, Hong Kong, September 26-29, 2017*, pages 154–163. IEEE Computer Society, 2017.

[16] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Brief announcement: Optimal self-stabilizing mobile Byzantine-tolerant regular register with bounded timestamps. In Taisuke Izumi and Petr Kuznetsov, editors, *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, volume 11201 of *Lecture Notes in Computer Science*, pages 398–403. Springer, 2018.

[17] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Approximate agreement under mobile Byzantine faults. *Theor. Comput. Sci.*, 758:17–29, 2019.

[18] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[19] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[20] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. *IACR Cryptol. ePrint Arch.*, 2001:6, 2001.

[21] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005.

[22] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 1:1–1:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[23] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 42–51. ACM, 1993.

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

[25] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[26] 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.

[27] 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.

[28] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

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

[30] 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.

[31] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Practically-self-stabilizing virtual synchrony. *J. Comput. Syst. Sci.*, 96:50–73, 2018.

[32] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Self-stabilizing Byzantine tolerant replicated state machine based on failure detectors. In Itai Dinur, Shlomi Dolev, and Sachin Lodha, editors, *Cyber Security Cryptography and Machine Learning - Second International Symposium, CSCML 2018, Beer Sheva, Israel, June 21-22, 2018, Proceedings*, volume 10879 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2018.

[33] 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.

[34] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. Self-stabilizing Byzantine resilient topology discovery and message delivery - (extended abstract). In Vincent Gramoli and Rachid Guerraoui, editors, *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*, volume 7853 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2013.

[35] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. Brief announcement: Robust and private distributed shared atomic memory in message passing networks. In Georgiou and Spirakis [50], pages 311–313.

[36] 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.

[37] Shlomi Dolev and Elad Schiller. Communication adaptive self-stabilizing group membership service. *IEEE Trans. Parallel Distributed Syst.*, 14(7):709–720, 2003.

[38] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults (abstract). In James H. Anderson, editor, *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, page 256. ACM, 1995.

[39] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: asynchronous BFT made practical. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2028–2041. ACM, 2018.

[40] Swan Dubois, Maria Potop-Butucaru, Mikhail Nesterenko, and Sébastien Tixeuil. Self-stabilizing Byzantine asynchronous unison. *J. Parallel Distributed Comput.*, 72(7):917–923, 2012.

[41] 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.

[42] 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.

[43] Paul Feldman and Silvio Micali. An optimal probabilistic algorithm for synchronous byzantine agreement. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*, volume 372 of *Lecture Notes in Computer Science*, pages 341–378. Springer, 1989.

[44] Michael Feldmann, Thorsten Götte, and Christian Scheideler. A loosely self-stabilizing protocol for randomized congestion control with logarithmic memory. In Mohsen Ghaffari, Mikhail Nesterenko, Sébastien Tixeuil, Sara Tucci, and Yukiko Yamauchi, editors, *Stabilization, Safety, and Security of Distributed Systems - 21st International Symposium, SSS 2019, Pisa, Italy, October 22-25, 2019, Proceedings*, volume 11914 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2019.

[45] 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.

[46] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[47] 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.

[48] Chryssis Georgiou, Robert Gustafsson, Andreas Lindhé, and Elad Michael Schiller. Self-stabilization overhead: A case study on coded atomic storage. In Atig and Schwarzmann [5], pages 131–147.

[49] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. Self-stabilizing snapshot objects for asynchronous failure-prone networked systems. In Atig and Schwarzmann [5], pages 113–130.

[50] Chryssis Georgiou and Paul G. Spirakis, editors. *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*. ACM, 2015.

[51] 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 2015, Montserrat, Spain, July 14-16, 2015, Post-Proceedings*, volume 9439 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2015.

[52] Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Inf. Process. Lett.*, 85(1):47–52, 2003.

[53] Pankaj Khanchandani and Christoph Lenzen. Self-stabilizing Byzantine clock synchronization with optimal precision. *Theory Comput. Syst.*, 63(2):261–305, 2019.

[54] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[55] Leslie Lamport. Byzantizing paxos by refinement. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2011.

[56] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

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

[58] 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.

[59] 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 2020, Singapore, November 29 - December 1, 2020*, pages 617–627. IEEE, 2020.

[60] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. Self-stabilizing uniform reliable broadcast. In Chryssis Georgiou and Rupak Majumdar, editors, *Networked Systems - 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3-5, 2020, Proceedings*, volume 12129 of *Lecture Notes in Computer Science*, pages 296–313. Springer, 2020.

[61] 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, Virtual Event, Nara, Japan, January 5-8, 2021*, pages 106–115. ACM, 2021.

[62] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. Self-stabilizing multivalued consensus in asynchronous crash-prone systems. *CoRR*, abs/2104.03129, 2021.

[63] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, New York, NY, USA, 2016. ACM.

[64] 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.

[65] Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous Byzantine consensus with t $2<n/3$ and $o(n^2)$ messages. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 2–9. ACM, 2014.

[66] 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.

[67] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and kdcs. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference*

on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346. Springer, 1999.

[68] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[69] David Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers: FTCS-22, The Twenty-Second Annual International Symposium on Fault-Tolerant Computing, Boston, Massachusetts, USA, July 8-10, 1992*, pages 386–395. IEEE Computer Society, 1992.

[70] Michael O. Rabin. Randomized Byzantine generals. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 403–409. IEEE Computer Society, 1983.

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

[72] 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.

[73] Iosif Salem and Elad Michael Schiller. Practically-self-stabilizing vector clocks in the absence of execution fairness. In Andreas Podelski and François Taïani, editors, *Networked Systems - 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers*, volume 11028 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 2018.

[74] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[75] Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theor. Comput. Sci.*, 444:100–112, 2012.

[76] 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.

[77] 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.

[78] 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.

[79] 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, Vancouver, B. C., Canada, August 27-29, 1984*, pages 163–178. ACM, 1984.

[80] Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, 2015.

[81] 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.