Self-stabilizing Byzantine fault-tolerant repeated reliable broadcast

Theoretical Computer Science 972 (2023) 114070

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Self-stabilizing Byzantine fault-tolerant repeated reliable broadcast ☆

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

[a] Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, SE-412 96, Sweden
[b] University Rennes IRISA, CNRS, Inria, 35042 Rennes, France

ARTICLE INFO

Article history:
Received 21 February 2023
Received in revised form 7 June 2023
Accepted 10 July 2023
Available online 20 July 2023

Keywords:
Reliable broadcast
Fault-tolerance
Self-stabilization

ABSTRACT

We study a well-known communication abstraction called Byzantine Reliable Broadcast (BRB). This abstraction is central in the design and implementation of fault-tolerant distributed systems, as many fault-tolerant distributed applications require communication with provable guarantees on message deliveries. Our study focuses on fault-tolerant implementations for message-passing systems that are prone to process-failures, such as crashes and malicious behavior.

At PODC 1983, Bracha and Toueg, in short, BT, solved the BRB problem. BT has optimal resilience since it can deal with $t < n/3$ Byzantine processes, where n is the number of processes. The present work aims to design an even more robust solution than BT by expanding its fault-model with self-stabilization, a vigorous notion of fault-tolerance. In addition to tolerating Byzantine and communication failures, self-stabilizing systems can recover after the occurrence of *arbitrary transient-faults*. These transient faults allow the model to represent temporary deviations from the assumptions on which the system was originally designed to operate (provided that the algorithm code remains intact).

We propose, to the best of our knowledge, the first self-stabilizing Byzantine fault-tolerant (BFT) solution for repeated BRB in signature-free message-passing systems (that follows BT's problem specifications). Our contribution includes a self-stabilizing variation on BT that solves a single-instance BRB for asynchronous systems. We also consider the problem of recycling instances of single-instance BRB. Our self-stabilizing BFT recycling for time-free systems facilitates the concurrent handling of a predefined number of BRB invocations and, in this way, can serve as the basis for self-stabilizing BFT consensus.

1. Introduction

Fault-tolerant distributed systems are known to be hard to design and verify. High-level communication primitives can facilitate such complex challenges. These high-level primitives can be based on low-level ones, such as the one that allows processes to send a message to only one other process at a time. Hence, when an algorithm wishes to broadcast message m to all processes, it can send m individually to every other process. Note that if failures occur during this broadcast, it can be the case that only some of the processes have received m. Even in the presence of network-level support for broadcasting or

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

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

https://doi.org/10.1016/j.tcs.2023.114070

Table 1
Glossary.

Notation	Meaning
AMP	a fault model for asynchronous message-passing systems.
ARQ	automatic repeat request.
BAMP	a fault model for Byzantine asynchronous message-passing systems.
BFT	the design criteria of Byzantine fault-tolerant.
BML	a synchrony assumption about bounded message lifetime, λ.
BRB	the problem of Byzantine Reliable Broadcast.
BT	the studied non-self-stabilizing BFT algorithm by Bracha and Toueg [4,5].
FC	the fault model-related assumption about fair communications.
IRC	the problem abstraction of Independent Round Counter, which is used for implementing the proposed BRB-instance recycling for repeated BRB.
RB	the problem of Reliable Broadcast.
SSBFT	self-stabilizing Byzantine fault-tolerant.
n	number of nodes in the system.
t	an upper bound on the number of faulty nodes.
channelCapacity	an upper bound on the number of messages in any given communication channel.
δ	a constant number of concurrent BRB instances.
λ	a bound on the BML lifetime.
$\diamond P_{mute}$	a class of mute failure detectors.

multicasting, node failures can cause similar inconsistencies. To simplify the design of fault-tolerant distributed algorithms, such inconsistencies need to be avoided. Many examples show how fault-tolerant broadcasts can significantly simplify the development of fault-tolerant distributed systems, *e.g.,* State Machine Replication [1] and Set-Constrained Delivery Broadcast [2]. The weakest variant, named *Reliable Broadcast*, lets all non-failing processes agree on the set of delivered messages. This set includes all the messages broadcast by the non-failing processes. Stronger Reliable Broadcast variants specify additional requirements on the delivery order. Such requirements can simplify the design of fault-tolerant distributed consensus, which allows reaching, despite failures, a common decision based on distributed inputs. Reliable broadcast and consensus (as well as message-passing emulation of read/write registers [1]) are closely related problems in the area of distributed computing. This work aims to design a reliable broadcast solution that is more fault-tolerant than the state-of-the-art.

The problem. Lamport, Shostak, and Pease [3] said that a process commits a Byzantine failure if it deviates from the algorithm instructions, say, by deferring (or omitting) messages that were sent by the algorithm or sending fake messages. Such malicious behavior can be the result of hardware malfunctions or software errors as well as coordinated malware attacks. Bracha and Toueg [4,5], BT from now on, proposed the communication abstraction of Byzantine Reliable Broadcast (BRB), which allows every process to invoke the brbBroadcast(v) operation and raise the brbDeliver() event upon message arrival. For convenience, the Glossary is provided in Table 1, see the end of this section. Following Raynal [1, Ch. 4], we consider the (single instance) BRB problem.

Single-instance BRB. Definition 1.1 specifies the task of single-instance BRB.

Definition 1.1. We require brbBroadcast(v) and brbDeliver() to satisfy:

- **BRB-validity.** Suppose a correct process BRB-delivers message m from a correct process p_i, where the term correct node refers to a non-faulty one. Then, p_i had BRB-broadcast m.
- **BRB-integrity.** No correct process BRB-delivers more than once.
- **BRB-no-duplicity.** No two correct processes BRB-deliver different messages from p_i (which might be faulty).
- **BRB-Completion-1.** Suppose p_i is a correct sender. All correct processes BRB-deliver from p_i eventually.
- **BRB-Completion-2.** Suppose a correct process BRB-delivers a message from p_i (who might be faulty). All correct processes BRB-deliver p_i's message eventually.

Note that self-stabilizing systems cannot be quiescent [6, Ch. 2.3]. In that sense, once the completion properties are satisfied, the self-stabilizing system cannot stop sending messages. For this reason, we use the name completion when referring to these properties rather than the common name in the literature, *e.g.,* [1, Ch. 4], which is termination.

Repeated BRB. Distributed systems may use, over time, an unbounded number of BRB instances. We require our solution to use, at any given point in time, a bounded amount of memory. Thus, for the sake of completeness, we also consider the problem of recycling an unbounded sequence of BRB invocations using bounded memory. We require the (single-instance)

BRB object, O, to have an operation, called recycle(), that allows the recycling mechanism to locally reset O, after all non-faulty processes have completed the delivery of O's message. Also, we require the mechanism to inform (the possibly recycled) O regarding its availability to take new missions. Specifically, the txAvailable() operation returns True when the sender can use O for broadcasting, and rxAvailable(k) returns True when O's latest transmission has arrived at the receiver p_k.

One may observe that the problem statement does not depend on the fault model or the design criteria. However, the proposed solution depends on all three. To clarify, we solve the single instance BRB using the requirements presented by Raynal [1, Ch. 4]. Then, we solve an extended version of the problem in which each BRB instance needs to be recycled so that an unbounded number of BRB instances can appear.

Fault models. Recall that our BRB solution may be a component in a system that solves consensus. Thus, we safeguard against Byzantine failures by following the same assumptions that are often used when solving consensus. Specifically, for the sake of deterministic and signature-free solvability [7], we assume there are at most $t < n/3$ crashed or Byzantine processes, where n is the total number of processes. The proposed solutions are for message-passing systems that have no guarantees on the communication delay and without explicit access to the clock. These systems are also prone to communication failures, *e.g.,* packet omission and duplication, as long as fair communication (FC) holds, *i.e.,* if p_i sends a message infinitely often to p_j, then p_j receives that message infinitely often. An extension of our solution can also deal with packet reordering via the use of the automatic repeat request (ARQ) solution presented in [8], cf. § 6.) We use two different fault models with notations following Raynal [1]:

- $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$. This is a Byzantine Asynchronous Message-Passing model with at most t (out of n) faulty nodes. The array $[\mathsf{FC}, t < n/3]$ denotes the list of all assumptions, *i.e.,* FC and $t < n/3$. We use this model for studying the problem of single-instance BRB since it has no synchrony assumptions.
- $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3, \diamond\mathsf{P}_{\mathsf{mute}}, \mathsf{BML}]$. By Doudou et al. [9], processes commit muteness failures when they stop sending specific messages, but they may continue to send "I-am-alive" messages. For studying the problem of BRB instance recycling, we enrich $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$ with a muteness detector of class $\diamond P_{mute}$ and assume bounded message lifetime (BML). That is, in any unbounded sequence of BRB invocations, at the time that immediately follows the x-th invocation, the messages associated with the $(x-\lambda)$-th invocation (or earlier) are either delivered or lost, where λ is a known upper-bound.

Raynal [1] refers to an asynchronous system as *time-free* when it includes synchrony assumptions, *e.g.,* BML. Note that BML does not imply bounded communication delay since an unbounded number of messages can be lost between any two successful transmissions. Our time-free implementation of the class $\diamond P_{mute}$ muteness detector uses an additional synchrony assumption, *i.e.,* there is a known bound, Θ, on the number of messages that some non-faulty processes can exchange without hearing from all non-faulty processes.

Self-stabilization. Dijkstra's seminal work [10] demonstrated recovery within a finite time after the occurrence of the last transient fault, which may corrupt the system state in any manner (as long as the program code stays intact). Dijkstra offered an alternative to traditional fault-tolerance, which aims at assuring that the system, at all times, remains in a correct state under the assumption that the system state changes only due to the algorithmic steps and specified failures. Alas, the latter target is unattainable in the presence of failures that were unforeseen during the algorithm design. To address this concern, self-stabilization considers failures that are transient by nature and hard to be observed. Thus, they cannot be specified by the fault model, such as the one above, which includes process and communication failures. Therefore, self-stabilizing systems are required to recover eventually (in the presence of all foreseen and specified failures) after the occurrence of the last unforeseen and transient failure.

In this paper, in addition to the faults specified above, we also aim to recover after the occurrence of the last *arbitrary transient-fault* [11,6]. These transient faults allow the model to represent temporary deviations from the assumptions on which the system was originally designed to operate. This includes the corruption of control variables, such as the program counter and packet payloads, as well as operational assumptions, such as that at most $t < n/3$ processes are faulty. Since the occurrence of these failures can be arbitrarily combined, we assume these transient-faults can alter the system state in unpredictable ways. When modeling the system, Dijkstra assumed that these violations can bring the system to an arbitrary state from which a *self-stabilizing system* should recover [10]. *I.e.,* Dijkstra requires the correctness proof of a self-stabilizing system to demonstrate recovery within a finite time after the last occurrence of a transient-fault and once the system has recovered, it must never violate the task requirements since no further transient faults occur. Arora and Gouda [12] refer to the former property as Convergence and the latter as Closure. Note that the satisfaction of the task requirements, which is Definition 1.1 in the case of this paper, holds only when Closure is guaranteed. In other words, only after the system has finished recovering from the occurrence of the last transient fault does a self-stabilizing system guarantees the satisfaction of the task requirements, for details see [11,6].

Related work. In the context of Reliable Broadcast, there are (non-self-stabilizing) Byzantine fault-tolerant (BFT) solutions [1, 13,14] and (non-BFT) self-stabilizing solutions [15] (even for total order broadcast [15–18]). We focus on BT [4,5] to which we propose a self-stabilizing variation. BT is the basis for advanced BFT algorithms for solving consensus, such as the one

by Mostéfaoui and Raynal [19]. BT is based on a simpler communication abstraction by Toueg that is called no-duplicity broadcast [20] (ND-broadcast, from now on). The requirements of ND-broadcast include all of Definition 1.1's requirements except for BRB-Completion-2.

Maurer and Tixeuil [21] consider an abstraction that is somewhat simpler than the one of ND-broadcast since they only consider no-duplicity (and none of the other requirements of Definition 1.1). They provide a single-instance synchronous broadcast that is self-stabilizing BFT (or SSBFT in short), whereas we consider an asynchronous repeated BRB that follows Definition 1.1, which is taken from Raynal [1]. Raynal studies the exact power of all the essential communication abstractions in the area of fault-tolerant message-passing systems. We study the more versatile definition provided by Raynal since we wish to connect our solution to all relevant protocols in the area.

In the broader context of SSBFT solutions for message-passing systems, we find topology discovery [22], storage [23–25], clock synchronization [26,27], approximate agreement [28], asynchronous unison [29], communication in dynamic networks [30], and SSBFT state-machine replication [31,32] to name a few. We also propose an SSBFT mechanism for recycling single-instance BRB objects, which uses a muteness detector inspired by Doudou et al. [9]. Observe that Doudou et al. consider the Consensus problem, whereas we consider BRB.

Our contribution. We present SSBRB, a fundamental module for dependable distributed systems: a Self-Stabilizing BFT Reliable Broadcast for asynchronous message-passing systems, *i.e.,* for the model of $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$. We obtain this new self-stabilizing solution via a transformation of the non-self-stabilizing BT algorithm [4,5] while preserving BT's resilience optimality of $t < n/3$. As in BT [4,5], in the absence of transient-faults, our asynchronous solution for single-instance BRB achieves operation completion within a finite time. After the occurrence of the last transient-fault, the system recovers within a bounded time in terms of asynchronous cycles (while assuming execution fairness among the non-faulty processes). The amount of memory used by the proposed algorithm is bounded and the communication costs of the studied and proposed algorithms are similar, *i.e.,* $\mathcal{O}(n^2)$ messages per BRB instance. The main difference is that our solution unifies all the types of messages sent by BT into a single message that is repeatedly sent. This repetition is imperative since self-stabilizing systems cannot be quiescent [6, Ch. 2.3].

Our contribution also includes an SSBFT recycling mechanism for time-free systems that are enriched with muteness detectors, *i.e.,* for the $\mathsf{BAMP}_{n,t}[\mathsf{FC}, \mathsf{BML}, \diamond\mathsf{P}_{\mathsf{mute}}]$ model. The mechanism is based on an algorithm that counts communication rounds. Since individual BRB broadcasters increment the counter independently, the algorithm is named the *Independent Round Counter* (IRC) algorithm. Implementing an SSBFT IRC is a non-trivial challenge since this counter should facilitate an unbounded number of increments, yet it has to use only a constant amount of memory. Using novel techniques for dealing with integer overflow events, the proposed solution recovers from transient faults eventually, uses a bounded amount of memory, and has a communication cost of $\mathcal{O}(n)$ messages per BRB instance.

To the best of our knowledge, we propose the first SSBFT solutions for the problems of IRC and repeated BRB (that follows BT's problem specifications [1, Ch. 4]). As said, BRB and IRC consider different fault models. § 2 defines $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$ and self-stabilization. The non-self-stabilizing BT algorithm for $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$ is studied in § 3. Our self-stabilization BFT variation on BT for $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$ is proposed in § 4. Our SSBTF IRC for $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3, \diamond\mathsf{P}_{\mathsf{mute}}, \mathsf{BML}]$ is presented in § 6. § 7 presents the correct proof of the proposed SSBTF IRC. § 9 compares the overhead of the studied and proposed solutions when executing δ BRB instances concurrently. This straightforward extension is imperative for the sake of practical deployments.

Recall that our task specifications (Definition 1.1) follow the ones by Raynal [1, Ch. 4], and thus, our SSBRB solution can serve as a building block for multivalued consensus [33].

2. System settings for $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$

This work focuses on two fault models. The asynchronous model for message-passing systems, $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$, is presented in this section. This model has a time-free enrichment, $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3, \diamond\mathsf{P}_{\mathsf{mute}}, \mathsf{BML}]$, which we explain in § 6. The $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$ model requires no guarantees of communication delay. Also, the algorithm cannot explicitly access the (local) clock (or use timeout mechanisms). The system consists of a set, $\mathcal{P}$, of n fail-prone nodes (or processes) with unique identifiers. Any pair of nodes $p_i, p_j \in \mathcal{P}$ have access to a bidirectional communication FIFO-channel, $channel_{j,i}$, that, at any time, has at most $\mathsf{channelCapacity} \in Z^+$ messages on transit from p_j to p_i (this assumption is due to a known impossibility [6, Ch. 3.2]).

In the *interleaving model* [6], 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 all incoming communication FIFO-channels, $channel_{j,i} : p_i, p_j \in \mathcal{P}$. 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.

The fault model and self-stabilization. The *legal executions* (LE) set refers to all the executions in which the requirements of task T hold. In this work, T_{BRB} denotes the task of BFT Reliable Broadcast, which § 1 specifies, and the executions in the set LE_{BRB} fulfill T_{BRB}'s requirements.

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

• *Communication failures and fairness.* We focus on solutions that are oriented towards asynchronous message-passing systems and thus they are oblivious to the time at which the packets depart and arrive. 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 loss and duplication. 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* (FC) assumption. As in [34], we assume that the communication channel from a correct node eventually includes only messages that were transmitted by the sender. The BT algorithm assumes reliable communication channels whereas the proposed solution does not make any assumption regarding reliable communications. § 4 provides further details regarding the reasons why the proposed solution cannot make this assumption.

• *Arbitrary node failures.* Byzantine faults model any failure in a node including crashes, and arbitrary malicious behaviors. Here, all nodes receive the arriving messages and calculate their 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 omit it from the communication channel. The adversary can also send fake messages, *i.e.,* not according to the algorithm. Note that the adversary has the power to coordinate such actions without any computational (or communication) limitation. For the sake of solvability [3,7,20], 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 failures needs to be less than one-third of the number, n, of nodes in the system, *i.e.,* $3t + 1 \leq n$. The set of non-faulty indexes is denoted by *Correct*, such that $i \in Correct$ when p_i is a correct node.

Arbitrary transient-faults. We consider any temporary violation of the assumptions according to which the system was designed to operate. We refer to these violations and deviations as *arbitrary transient-faults* and assume that they can corrupt the system state arbitrarily (while keeping the program code intact). The occurrence of a transient fault is rare. Thus, we assume that the last arbitrary transient fault occurs before the system execution starts [6]. Also, it leaves the system to start in an arbitrary state.

Dijkstra's self-stabilization. An algorithm is *self-stabilizing w.r.t. LE*, when every execution R of the algorithm reaches within a finite period a suffix $R_{legal} \in LE$ that is legal. Namely, Dijkstra [10] requires $\forall R : \exists R' : R = R' \circ R_{legal} \wedge R_{legal} \in LE \wedge |R'| \in Z^+$, where the operator $\circ$ denotes that $R = R' \circ R''$ is the concatenation of R' with R''. By Arora and Gouda [12], 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.

Complexity measures. The time between the invocation of a BRB-broadcast and the occurrence of all required BRB-deliveries is called latency. As in MR [35], we show finite latency without assuming execution fairness, *i.e.,* every correct node that has an applicable step infinitely often, takes a step infinitely often.

Stabilization time refers to the period in which the system recovers after the occurrence of the last transient fault. When estimating the stabilization time, our analysis assumes that all correct nodes complete roundtrips infinitely often with all other correct nodes, where a roundtrip is defined as the successful exchange of request-reply messages. However, once the Convergence period is over, no fairness assumption is needed.

The stabilization time is measured in terms of asynchronous cycles. The definition of these cycles uses the constant number num_b, which is the maximum number of broadcasts per iteration of the 'do forever loop'. Note that all self-stabilizing algorithms have such 'do forever loop' since these systems cannot be quiescent due to a well-known impossibility [6, Ch. 2.3].

The first asynchronous cycle, R', of execution $R = R' \circ R''$ is the shortest prefix of R in which every correct node can start and complete at least num_b roundtrips with every correct node. The second asynchronous communication round of R is the first round of the suffix R'', and so on.

3. The non-self-stabilizing BT algorithm

Recall that the BT algorithm [4,5], is a BRB solution for $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$. BT is based on the simpler communication abstraction, ND-broadcast. The ND-broadcast's task includes all of the BRB requirements (§ 1) except BRB-Completion-2. We review BT after studying the ND-broadcast algorithm [20].

ND-Broadcast. Algorithm 1 brings Toueg's solution [20]. It assumes that every correct node invokes ND-broadcast at most once. Node p_i initiates the ND-broadcasts of m_i by sending INIT(m_i) to all nodes (line 1). Upon the first arrival of this message to p_j, it disseminates the fact that p_i has initiated m's ND-broadcast by sending ECHO(i, m) to all nodes (line 2). In the code of the algorithm, when message INIT(m) arrives from p_j, the handling procedure passes the parameter mJ to allow access to the field mJ. Upon its arrival to p_k from more than $(n+t)/2$ different nodes, p_k ND-deliver $\langle i, m_i \rangle$ (line 3). Please note that in lines 3 to 5, the event handler procedure is defined with two parameters: k and mK. These parameter names adhere to a convention where p_k's message is represented as mK.

Byzantine Reliable Broadcast. As explained, we present the BT solution for BRB as an extension of Toueg's solution for ND-broadcast. Algorithm 2 satisfies the BRB requirements (§ 1) assuming $t < n/3$.

Algorithm 1: Non-self-stabilizing no-duplicity broadcast (ND-broadcast); code for p_i.

```
1 operation ndBroadcast(m) do broadcast INIT(m);
2 upon INIT(mJ) first arrival from p_j do broadcast ECHO(j, mJ);
3 upon ECHO(k, mK) arrival from p_j begin
4     if ECHO(k, mK) received from at least (n+t)/2 nodes then
5         ndDeliver(k, mK)
```

Algorithm 2: Non-self-stabilizing Byzantine Reliable Broadcast (BRB); code for p_i.

```
1 operation brbBroadcast(m) do broadcast INIT(m);
2 upon INIT(mJ) first arrival from p_j do broadcast ECHO(j, mJ);
3 upon ECHO(k, mK) arrival from p_j begin
4     if ECHO(k, mK) received from at least (n+t)/2 nodes ∧ READY(k, mK) not yet broadcast then broadcast READY(k, mK) ;
5 upon READY(k, mK) arrival from p_j begin
6     if READY(k, mK) received from (t+1) nodes ∧ READY(k, mK) not yet broadcast then broadcast READY(k, mK);
7     if READY(k, mK) received from at least (2t+1) nodes ∧⟨k, mK⟩ not yet BRB-Delivered then brbDeliver (k, mK);
```

The first difference between the ND-broadcast and BRB algorithms is in the consequent clause of the if-statement in line 4, where ND-delivery of $\langle j, m\rangle$ is replaced with the broadcast of READY(j, m). This broadcast indicates that p_i is ready to BRB-deliver $\langle j, m\rangle$ as soon as it receives sufficient support, *i.e.,* the arrival of READY(j, m), which tells that $\langle j, m\rangle$ can be BRB-delivered. Note that BRB-no-duplicity protects Algorithm 2 from the case in which p_i broadcasts READY(j, m) while p_j broadcasts READY(j, m'), such that $m \neq m'$.

The new part of the BRB algorithm (lines 5 to 7) includes two if-statements. The first one (line 6) makes sure that every correct node receives READY(j, m) from at least one correct node before BRB-delivering $\langle j, m\rangle$. This is done via the broadcasting of READY(j, m) as soon as p_i received it, for the first time, from at least $(t+1)$ different nodes (since t of them can be Byzantine).

The second if-statement (line 7) makes sure that no two correct nodes BRB-deliver different pairs (in the presence of plausibly fake READY$(j, \text{-})$ messages sent by Byzantine nodes, where the symbol '-' stands for any legal value). That is, the delivery of a BRB-broadcast is done only after the first reception of the pair $\langle j, m\rangle$ from at least $(2t+1)$ nodes (out of which at most t are Byzantine). The receiver then knows that there are at least $t+1$ correct nodes that can make sure that the condition in line 6 holds eventually for all correct nodes whenever reliable communication channels are assumed.

4. Self-stabilizing Byzantine-tolerant single-instance BRB

Before proposing our solution, we review the challenges of transforming the non-self-stabilizing BT into a self-stabilizing one.

Challenges and approaches. We analyze here the behavior of the BT algorithm in the presence of transient-faults. We would like to highlight to the reader that our analysis is relevant only in the context of self-stabilization since Bracha and Toueg [4,5] do not consider transient-faults in their work. In Section 8, we explain how to coordinate the recycling of BRB objects.

Dealing with memory corruption and desynchronized system states. Recall that transient faults can corrupt the system state in any manner (as long as the program code remains intact). For example, memory corruption can cause the local state to indicate that a certain message has already arrived (line 2) or that a certain broadcast was already performed (line 6). This means that some necessary messages will not be broadcast. This will result in indefinite blocking. Our solution avoids such situations by unifying all messages and repeatedly broadcasting the unified message (until the BRB object is recycled).

Datagram-based end-to-end communications. Algorithm 2 assumes reliable communication channels when broadcasting in a quorum-based manner, *i.e.,* sending the same message to all nodes and then waiting for a reply from $n-t$ nodes. Next, we explain why, for the sake of a simpler presentation, we choose not to follow this assumption. Self-stabilizing end-to-end communications require a known bound on the capacity of the communication channels [6, Ch. 3]. In the context of self-stabilization and quorum systems, we must avoid situations in which communicating in a quorum-based manner can lead to a contradiction with the system's assumptions. Dolev, Petig, and Schiller [36] explain that there might be a subset of nodes that can complete many roundtrips with a given sender, while other nodes merely accumulate messages in their communication channels. The channel-bounded capacity implies that the system has to either block or omit messages before their delivery. For this reason, the proposed solution does not assume access to reliable channels. Instead, communications are simply repeated by the algorithm's do-forever loop.

Algorithm 3: SSBFT BRB with instance recycling interface; p_i's code.

```
1  types: brbMSG := {init, echo, ready};
2  variables:
3  msg[P][brbMSG] := [[∅, ..., ∅], ..., [∅, ..., ∅]]; // most recently sent/received message
4  wasDelivered[P] := [False, ..., False]; /* indicates if the message was delivered */
5  required interfaces: txAvailable() and rxAvailable(k);
6  provided interfaces: recycle(k) do {(msg[k], wasDelivered[k]) ← ([∅, ∅, ∅], False)} /* also initialize the communication channel
       with p_k [6, Ch. 3.1] */
7  macros:
8  msg() begin
9      let m[P][brbMSG] := [[∅, ..., ∅], ..., [∅, ..., ∅]];
10     m[i][init] ← msg[i][init];
11     foreach p_k ∈ P \ {p_i} do m[k][init] ← ∅;
12     foreach s ∈ brbMSG \ {init} ∧ p_k ∈ P ∧ (i, m) ∈ msg[k][s] do
13         m[k][s] ← m[k][s] ∪ {(i, m)};
14     return m;
15 mrg(mJ, j) begin
16     msg[j][init] ← mJ[j][init];
17     foreach p_k ∈ P ∧ s ∈ brbMSG \ {init} do
18         msg[k][s] ← (msg[k][s] \ {(j, -)}) ∪ mJ[k][s];
19 operations:
20 brbBroadcast(v) do {if txAvailable() then recycle(i); msg[i][init] ← {v}};
21 brbDeliver(k) begin
22     if ∃m : (n − t) ≤ |{p_ℓ ∈ P : (ℓ, m) ∈ msg[k][ready]}| ∧ rxAvailable(k) then
23         wasDelivered[k] ← m ≠ ⊥; return m;
24     else return ⊥;
25 brbWasDelivered(k) do return wasDelivered[k];
26 do-forever begin
27     foreach p_k ∈ P do
28         if |msg[k][init]| ≠ 1 ∨ ∃s ≠ init : ∃p_ℓ ∈ P:
             ∃(ℓ, m), (ℓ, m′) ∈ msg[k][s] : m ≠ m′ ∨ ∃(ℓ, m) ∈ msg[k][echo] : m ∉ msg[k][init] ∨ ∃(ℓ, m) ∈ msg[k][ready]: ¬((n+t)/2
             < |{p_ℓ′ ∈ P : (ℓ′, m) ∈ msg[k][echo]}| ∨ (t+1) ≤ |{p_ℓ′ ∈ P : (ℓ′, m) ∈ msg[k][ready]}|)∨(wasDelivered[k] ∧ brbDeliver(k) = ⊥) then
29             recycle(k);
30         if ∃m ∈ msg[k][init] then msg[k][echo] ← msg[k][echo] ∪ {(i, m)};
31         if ∃m : (n+t)/2 < |{p_ℓ ∈ P : (ℓ, m) ∈ msg[k][echo]}| then
32             msg[k][ready] ← msg[k][ready] ∪ {(i, m)};
33         if ∃m : (t+1) ≤ |{p_ℓ ∈ P : (ℓ, m) ∈ msg[k][ready]}| then
34             msg[k][ready] ← msg[k][ready] ∪ {(i, m)};
35     broadcast MSG(msg(), txMSG());
36 upon MSG(brbJ, ircJ) arrival from p_j begin
37     rxMSG(brbJ, ircJ, j);
38     mrg(brbJ, j);
```

Self-stabilizing BFT single-instance solution. Algorithm 3 proposes our SSBRB solution for $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3]$. The key ideas are to (i) offer a variance of Algorithm 2, such that its operations always complete even when starting from a corrupted state, (ii) offer interfaces for coordinating the recycling of a given BRB object, as well as (iii) offer interfaces for accessing the delivered value and current status of the broadcast. This way, the recycling coordination mechanism (§ 6) can make sure that no BRB object is recycled before all correct nodes deliver its result. Also, once all correct nodes have delivered a message, the BRB object can eventually be recycled. The boxed code fragments in lines 35 to 36 are irrelevant to our single-instance BRB implementation. Moreover, in line 28 we use the underline font to denote a clause that is only relevant when executing Algorithm 3 as a single instance BRB (and not when running it concurrently with Algorithms 4 and 5). Fig. 1 presents an overview of the solution components.

Types, constants, variables, message structure, and macros. As mentioned, the message MSG() unifies the messages of Algorithm 2. The array $msg[][]$ stores both the information that is sent and arrived at these messages. *I.e.,* for any $j \neq i$, $msg_i[j][]$ stores the information related to p_j's BRB object. Also, we define the type $\texttt{brbMSG} := \{\texttt{init}, \texttt{echo}, \texttt{ready}\}$ (line 1) for storing information related to BRB-broadcast messages, for example, $msg_i[i][\texttt{init}]$ stores the information that BRB-broadcast disseminates of INIT() messages and the results of the content of READY() messages appear in $msg_i[i][\texttt{ready}]$. Using the macro $msg()$, p_i can send information that is supported by only p_i, *i.e.,* p_i's init message as well as p_i's support

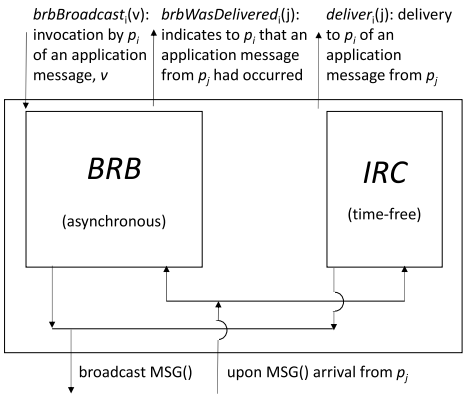

Fig. 1. The solution components.

records (i, m) for phases `echo` and `ready`. The macro $mrg()$ allows the integration of received information with the local one.

The algorithmic transformation. The brbBroadcast(v) operation (line 20) allows Algorithm 3 to invoke BRB-broadcast with message v. The invocation causes Algorithm 3 to follow the logic of the BRB solution in Algorithm 2 in lines 21 to 24 and 30 to 34. Our solution also includes consistency tests at line 28.

Getting delivered values and current status. Algorithms 1 and 2 inform the application about message arrival by raising the events of ndDeliver() (line 5), and respectively, brbDeliver() (line 7). As explained in the context of self-stabilization, a single transient fault can cause the local state to indicate that the message has already been delivered. Our SSBFT BRB solution takes another approach in which the application is pulling information from Algorithm 3 by invoking the brbDeliver() operation (line 21), which returns message m once it has arrived from at least $n - t$ different nodes (line 23). Otherwise, $\perp$ is returned since no message is ready to be delivered (line 24). Observe that once brbDeliver$_i(k) : i, k \in Correct$ returns a non-$\perp$ value, brbDeliver$_i(k)$ returns a non-$\perp$ value in all subsequent invocations.

Satisfying BRB-integrity (Definition 1.1) in a self-stabilizing manner. Line 23 records the fact that the non-$\perp$ message was delivered at least once by storing True in $wasDelivered_i[k]$. The application can access the value stored in $wasDelivered_i[k]$ by invoking brbWasDelivered$_i(k)$ (line 25). In other words, whenever the application at p_i tries to pull the arriving message from p_k, it can first test brbWasDelivered$_i(k)$. If False is returned, brbDeliver$_i(k)$ can be invoked. Otherwise, the message from p_k was already delivered. Note that the use of brbWasDelivered() is needed only for the single-instance BRB implementation presented by Algorithm 2 since the repeated BRB version uses the recycling mechanism (§ 6) for providing a simpler way to satisfy BRB-integrity.

5. Correctness of Algorithm 3

Definition 5.1 defines the terms *active nodes* and *consistency*. Theorem 5.1 shows that *Convergence* is obtained by consistency regaining (Lemma 5.2). Then, we prove completion from any starting state (Lemma 5.4). The rest of the proof is based on the assumption that BRB objects are eventually recycled after their task was completed (§ 1). The *Closure* proof (§ 5) shows that Algorithm 3 satisfies BRB's requirements (Definition 1.1).

Definition 5.1 *(Active nodes and consistent executions of Algorithm 3).* Node $p_i \in \mathcal{P}$ is said to be *active* when $msg_i[i][\texttt{init}] \neq \emptyset$. Let R be an Algorithm 3's execution, $p_i, p_j \in \mathcal{P} : i \in Correct$, and $c \in R$. Suppose in c: **(brb.i)** The if-statement condition in line 28 does not hold *w.r.t.* any correct node. **(brb.ii)** any message MSG($brbJ = mJ$, -) in transit from p_i to p_j is the result of $msg_i()$. In this case, we say that c is consistent *w.r.t.* p_i. Suppose every system state in R is consistent *w.r.t.* p_i. Then we say that R is consistent *w.r.t.* p_i.

Note the term active (Definition 5.1) does not distinguish between the cases in which a node is active due to the occurrence of a transient fault or the invocation of brbBroadcast(v). Our proof uses the term post-recycled system state (Definition 5.2), which is also a consistent one (Definition 5.1).

Definition 5.2 *(Post-recycle system states and complete invocation of operations).* We say that system state c is *post-recycled w.r.t.* $p_i \in \mathcal{P} : i \in Correct$ if $\forall j \in Correct : (msg_j[i], wasDelivered_j[i]) = ([\emptyset, \emptyset, \emptyset], \text{False})$ holds and no communication channel from p_i to p_j includes MSG($brbJ \neq [\emptyset, \ldots, \emptyset]$, -). Suppose that execution R starts in the post-recycled system state c and p_i invokes brbBroadcast$_i(v)$ exactly once. In this case, we say that R includes a *complete BRB invocation w.r.t.* p_i.

Algorithm 3's Convergence. Theorem 5.1 bounds the stabilization time of Algorithm 3 using asynchronous cycles (§ 2) while considering fair executions in which all correct nodes are assumed to complete roundtrips infinitely often with all other correct nodes.

The Convergence proof (Theorem 5.1) shows that, within $\mathcal{O}(1)$ asynchronous cycles, *w.r.t.* any correct p_i, the state of p_i's BRB object reaches its post-recycling state (Definition 5.2). Our recycling mechanism facilitates the satisfaction of this requirement. That is, within $\mathcal{O}(1)$ asynchronous cycles, p_i's BRB object has either reached a post-recycling state or has completed its task, and thus it is ready to be recycled. Specifically, Lemma 5.2 demonstrates consistency within $\mathcal{O}(1)$ asynchronous cycles. Lemma 5.3 shows that if a correct node is not active, its BRB object reaches a post-recycling state within $\mathcal{O}(1)$ asynchronous cycles. Lemma 5.6 shows that if a correct node is active, the object completes its task within $\mathcal{O}(1)$ asynchronous cycles, and thus, it can be recycled.

Theorem 5.1 *(Algorithm 3's Convergence). Let R be a fair execution of Algorithm 3. Within $\mathcal{O}(1)$ asynchronous cycles, Convergence is done.*

Proof of Theorem 5.1.

Lemma 5.2 *(Consistency). Let $i \in$ Correct. Within $\mathcal{O}(1)$ asynchronous cycles, the system reaches a state $c \in R$ that starts a consistent execution* w.r.t. p_i *(Definition 5.1).*

Proof of Lemma 5.2. Suppose that R's starting state is not consistent *w.r.t.* p_i. Specifically, suppose that Invariant (brb.i) does not hold. *I.e.,* the if-statement condition in line 28 holds. Since R is fair, within $\mathcal{O}(1)$ asynchronous cycles, p_i takes a step that includes the execution of line 29, which assures that R becomes consistent *w.r.t.* p_i (brb.i). By Algorithm 3's code, once Invariant (brb.i) holds *w.r.t.* p_i in $c \in R$, it holds in state $c' \in R$ that follows c.

Let m be a message that in R's starting state resides in a channel from p_i. Since the channel preserves the FIFO order, if m is ever delivered during R, then it does so within $\mathcal{O}(1)$ asynchronous cycles. By line 35, within $\mathcal{O}(1)$ asynchronous cycles, (brb.ii) holds since it is sufficient to consider only messages that were sent during R from nodes in which (brb.i) holds. $\square_{\text{Lemma 5.2}}$

Lemma 5.3 *(Completion starting from any system state and no active broadcaster). Suppose $p_i \in \mathcal{P} : i \in$ Correct is not active. Within $\mathcal{O}(1)$ asynchronous cycles, $\forall j \in$ Correct : $(msg_j[i], wasDelivered_j[i]) = ([\emptyset, \emptyset, \emptyset], \mathsf{False})$.*

Proof of Lemma 5.3. The proof follows from the assumption that $msg_i[i][\texttt{init}] = \emptyset$ throughout R, Argument (1), and Lemma 5.2.

Argument (1) *Within $\mathcal{O}(1)$ asynchronous cycles, $msg_i[i][\texttt{init}] = msg_j[i][\texttt{init}]$ holds if $msg_i[i][\texttt{init}]$'s values do not change during R.* Since p_i is correct, it broadcasts MSG($brbJ = msg_i$, -) within $\mathcal{O}(1)$ asynchronous cycles. Since R is fair, every correct $p_j \in \mathcal{P}$ receives MSG($brbJ$, -) within $\mathcal{O}(1)$ asynchronous cycles. Thus, the argument holds (due to line 16). $\square_{\text{Lemma 5.3}}$

Lemma 5.4 *(Completion starting from any system state and an active broadcaster). Suppose $p_i \in \mathcal{P} : i \in$ Correct is active throughout R. Within $\mathcal{O}(1)$ asynchronous cycles, $\forall i, j \in$ Correct : $\mathsf{brbDeliver}_j(i) \neq \perp$.*

Proof of Lemma 5.4. Since p_i is active, $msg_i[i][\texttt{init}] = \{m\}$. By Lemma 5.3's Argument (1), $msg_j[i][\texttt{init}] = \{m\}$ within $\mathcal{O}(1)$ asynchronous cycles. Also, $\forall j \in$ Correct : $msg_j[j][\texttt{echo}] \supseteq \{(i, m)\}$ since node p_j obverses that the if-statement condition in line 30 holds (for the case of $k_j = i$). Then, p_j broadcasts MSG($brbJ = msg_j$, -) at least once in every $\mathcal{O}(1)$ asynchronous cycles. Since R is fair, every correct $p_\ell \in \mathcal{P}$ receives MSG($brbJ$, -) within $\mathcal{O}(1)$ asynchronous cycles. Thus, $\forall j, \ell \in$ Correct : $msg_\ell[j][\texttt{echo}] \supseteq \{(i, m)\}$ (line 17). Since $n - t > (n+t)/2$, p_ℓ observes that $(n+t)/2 < |\{p_x \in \mathcal{P} : (x, m) \in msg_\ell[i][\texttt{echo}]\}|$ holds, *i.e.,* the if-statement condition in line 31 holds, and thus, $msg_\ell[\ell][\texttt{ready}] \supseteq \{(i, m)\}$ holds. Note $t < (n+t)/2$, faulty nodes cannot prevent a correct node from broadcasting MSG($brbJ = mJ$, -) : $mJ[i][\texttt{ready}] \supseteq \{(i, m)\}$ at least once every $\mathcal{O}(1)$ asynchronous cycles, say, by colluding and sending MSG($brbJ = mJ$, -) : $mJ[i][\texttt{ready}] \supseteq \{(i, m')\} \wedge m' \neq m$. Since R is fair, any correct $p_y \in \mathcal{P}$ receives MSG($brbJ = mJ$, -) within $\mathcal{O}(1)$ asynchronous cycles. Thus, $\forall j, y \in$ Correct : $msg_y[j][\texttt{ready}] \supseteq \{(i, m)\}$ holds (line 17). When p_y invokes $\mathsf{brbDeliver}_y(i)$ (line 21), $\exists_m (n - t) \leq |\{p_\ell \in \mathcal{P} : (\ell, m) \in msg_y[k_y = i][\texttt{ready}]\}|$ holds and m is returned. $\square_{\text{Lemma 5.4}}$ $\square_{\text{Theorem 5.1}}$

Closure of BRB-broadcast. The Closure proof considers post-recycled system states and complete invocation of operations (Definition 5.2).

Theorem 5.5 *(BRB Closure). Let R be an Algorithm 3's execution that starts in a post-recycled and in which every correct node eventually invokes a complete BRB-broadcast. R demonstrates a BRB construction.*

Proof of Theorem 5.5. Lemma 5.6 shows operation completion within a finite time. Lemma 5.6's proof is by similar arguments to the ones of Lemma 5.4. The main difference is that fairness assumptions are not considered (§ 2), and thus, we demonstrate that BRB-Completion-1 holds eventually.

Lemma 5.6 *(BRB-Completion-1). BRB-Completion-1 holds.*

Proof of Lemma 5.6. Since p_i is correct, it broadcasts MSG($brb\,J = msg_i$, -) infinitely often, where $msg_i[i][\texttt{init}] = \{m\}$. By fair communication, every correct p_j receives MSG($brb\,J$, -) eventually. Thus, $\forall j \in Correct : msg_j[i][\texttt{init}] = \{m\}$ (line 17). Also, $\forall j \in Correct : msg_j[j][\texttt{echo}] \supseteq \{(i,m)\}$ since p_j's if-statement in line 30 holds (for the case of $k_j = i$). Then, p_j broadcasts MSG($brb\,J = msg_j$, -) infinitely often. By fair communication, every correct p_ℓ receives MSG($brb\,J$, -) eventually. Thus, $\forall j, \ell \in Correct : msg_\ell[j][\texttt{echo}] \supseteq \{(i,m)\}$ (line 17). Since $n-t > (n+t)/2$, p_ℓ observes that $(n+t)/2 < |\{p_x \in \mathcal{P} : (x,m) \in msg_\ell[i][\texttt{echo}]\}|$ holds, *i.e.,* the if-statement in line 31 holds, and thus, $msg_\ell[i][\texttt{ready}] \supseteq \{(i,m)\}$ holds. Note that, since $t < (n+t)/2$, faulty nodes cannot prevent a correct node from broadcasting MSG($brb\,J = m\,J$, -) : $mJ[i][\texttt{ready}] \supseteq \{(i,m)\}$ infinitely often, say, by colluding and sending MSG($brb\,J = m\,J$, -) : $mJ[i][\texttt{ready}] \supseteq \{(i,m')\} \wedge m' \neq m$. By fair communication, every correct $p_y \in \mathcal{P}$ receives MSG($brb\,J = m\,J$, -) eventually. Thus, $\forall j, y \in Correct : msg_y[j][\texttt{ready}] \supseteq \{(i,m)\}$ holds (line 17). Therefore, whenever p_y invokes $\mathsf{brbDeliver}_y(i)$ (line 21), the condition $\exists_m (n-t) \leq |\{p_\ell \in \mathcal{P} : (\ell,m) \in msg_y[i][\texttt{ready}]\}|$ holds, and thus, m is returned. □Lemma 5.6

Lemma 5.7 *(BRB-Completion-2). BRB-Completion-2 holds.*

Proof of Lemma 5.7. By line 22, p_i can BRB-deliver m from p_j only once $\exists_m (n-t) \leq |\{p_\ell \in \mathcal{P} : (\ell,m) \in msg_i[j][\texttt{ready}]\}|$ holds. Due to the assumption that R's starting state is post-recycled, only lines 18 and 32 to 34 can add items to $msg_i[j][\texttt{ready}]$. Let MSG(mJ, -) be a message such that $(\ell,m) \in mJ[j][\texttt{ready}]$. Specifically, line 18 adds to $msg_i[j][\texttt{ready}]$ items according to information in MSG(mJ, -) messages coming from, say, p_ℓ. *I.e.,* infinitely often, at least $t+1$ distinct and correct nodes, p_ℓ, broadcast MSG(mJ, -) : $(\ell,m) \in mJ[j][\texttt{ready}]$. By fair communication and line 18, all correct nodes, p_x, eventually receive MSG(mJ, -) : $(\ell,m) \in mJ[j][\texttt{ready}]$ from at least $t+1$ distinct nodes, p_ℓ, and make sure that $(\ell,m) \in msg_x[j][\texttt{ready}]$. By line 34, $msg_x[j][\texttt{ready}] \supseteq \{(x,m)\}$, *i.e.,* every correct node broadcasts MSG(mJ, -) : $(x,m) \in mJ[j][\texttt{ready}]$ infinitely often. By fair communication and line 18, all correct p_y receive MSG(mJ, -) : $(x,m) \in mJ[j][\texttt{ready}]$ from at least $n-t$ distinct nodes, p_x, eventually. *I.e.,* $\exists_m (n-t) \leq |\{p_x \in \mathcal{P} : (x,m) \in msg_y[j][\texttt{ready}]\}|$ holds (due to line 18) and $\forall i \in Correct : \mathsf{brbDeliver}_i(j) \neq \perp$. □Lemma 5.7

Lemma 5.8. *The BRB-integrity property holds.*

Proof of Lemma 5.8. Suppose $\mathsf{brbDeliver}_i(k) = m \neq \perp$ holds in $c \in R$, where $i \in Correct \wedge p_k \in \mathcal{P}$. Also, (towards a contradiction) $\mathsf{brbDeliver}_i(k) = m' \notin \{\perp, m\}$ holds in $c' \in R$, where c' appears after c in R. In other words, $\exists_m (n-t) \leq |\{p_\ell \in \mathcal{P} : (\ell,m) \in msg_i[k][\texttt{ready}]\}|$ in c and $\exists_{m'} (n-t) \leq |\{p_\ell \in \mathcal{P} : (\ell,m') \in msg_i[k][\texttt{ready}]\}|$ in c'. Since $n \geq 3t+1$ and R starts in a post-recycling state (and thus consistent), there is $p_\ell \in \mathcal{P}$, such that $(\ell,m) \in msg_i[k][\texttt{ready}]$ in c and $(\ell,m') \in msg_i[k][\texttt{ready}]$ in c'. Algorithm 3 does not remove elements from any entry $msg[][]$ since R is consistent. This means that $msg_i[k][\texttt{ready}]$ includes both (ℓ,m) and (ℓ,m') in c'. However, this contradicts the fact that c' is consistent (Lemma 5.2). Thus, $\nexists c' \in R$ and BRB-integrity holds. □Lemma 5.8

Lemma 5.9 *(BRB-validity). BRB-validity holds.*

Proof of Lemma 5.9. Let $i, j \in Correct$. Suppose p_j BRB-delivers m from p_i. The proof shows that p_i BRB-broadcasts m. *I.e.,* suppose the adversary, who captures up to t (Byzantine) nodes, sends "fake" messages of $(i,m) \in msg_j[i][\texttt{echo}]$ or $msg_j[i][\texttt{ready}]$, but p_i, who is correct, never invoked $\mathsf{brbBroadcast}(m)$. In this case, our proof shows that no correct node BRB-delivers $\langle i,m\rangle$. This is because there are at most t nodes that can broadcast "fake" messages. Thus, $\mathsf{brbDeliver}(k)$ (line 21) cannot deliver $\langle i,m\rangle$ since $t < n-t$, *i.e.,* $\nexists_m (n-t) \leq |\{p_\ell \in \mathcal{P} : (\ell,m) \in msg_i[i][\texttt{ready}]\}|$. □Lemma 5.9

Lemma 5.10 *(BRB-no-duplicity). BRB-no-duplicity holds.*

Proof of Lemma 5.10. Let $i, j \in Correct$ and $p_k \in \mathcal{P}$. Suppose that p_i and p_j broadcast MSG(mJ, -) : $mJ[k][\texttt{ready}] = \{(i,m)\}$, and respect., MSG($mJ$, -) : $mJ[k][\texttt{ready}] = \{(j,m')\}$. We prove BRB-no-duplicity by showing that $m = m'$. Since R is post-recycling, there must be a step in R in which the element $(x,$ -$)$ is added to $msg_x[k][\texttt{ready}]$ for the first time during R, where $p_x \in \{p_i, p_j\}$. The correctness proof considers the following two cases.

• **Both p_i and p_j add $(x,$ -$)$ due to line 32.** Suppose, towards a contradiction, that $m \neq m'$. Since the if-statement condition in line 32 holds for both p_i and p_j, we know that $\exists_m (n+t)/2 < |\{p_\ell \in \mathcal{P} : (\ell,m) \in msg_i[k][\texttt{echo}]\}|$ and $\exists_{m'} (n+t)/2 < |\{p_\ell \in \mathcal{P} : (\ell,m') \in msg_j[k][\texttt{echo}]\}|$ hold. Since R is post-recycling, this can only happen if p_i and p_j received MSG(mJ, -) : $mJ[k][\texttt{echo}] \supseteq \{(\ell,m)\}$, and respect., MSG($mJ$, -) : $mJ[k][\texttt{echo}] \supseteq \{(\ell,m')\}$ from $(n+t)/2$ distinct nodes, p_ℓ. Note that $\exists p_x \in$

$Q_1 \cap Q_2 : x \in Correct$, where $Q_1, Q_2 \subseteq \mathcal{P} : |Q_1|, |Q_2| \geq 1+(n+t)/2$ (as in Raynal [1], item (c) of Lemma 3). But, any correct node, p_x, sends at most one element in $msg_x[k][\texttt{echo}]$ (lines 8 to 14 and 35) during R. Thus, $m = m'$, which contradicts the case assumption.

• **There is $p_x \in \{p_i, p_j\}$ that adds $(x, \text{-})$ due to line 34.** *I.e.,* $\exists_{m''}(t+1) \leq |\{p_\ell \in \mathcal{P} : (\ell, m'') \in msg_x[k][\texttt{ready}]\}| \wedge m'' \in \{m, m'\}$. Since there are at most t faulty nodes, p_x received $\text{MSG}(mJ) : mJ[k][\texttt{ready}] = \{(\ell, m'')\}$ from at least one correct node, say p_{x_1}, which received $\text{MSG}(mJ) : mJ[k][\texttt{ready}] = \{(k, m'')\}$ from p_{x_2}, and so on. This chain cannot be longer than n and it must be originated by the previous case in which $(x, \text{-})$ is added due to line 32. Thus, $m = m'$. □Lemma 5.10 □Theorem 5.5

6. Self-stabilizing BRB object recycling in message-passing systems

We present the *Independent Round Counter* (IRC) task, which can implement txAvailable() and rxAvailable(k) (Fig. 4 and Algorithm 3) for the model of $\text{BAMP}_{n,t}[\text{FC}, t < n/3, \diamond\text{P}_{\text{mute}}, \text{BML}]$. This section also explains how to enrich the model presented in § 2 with muteness detectors and the BML assumption.

In non-self-stabilizing node-failure-free systems, the operations txAvailable() and rxAvailable(k) can be implemented using prevailing mechanisms for ARQ, which use unbounded counters. These mechanisms are often used for guaranteeing reliable communications by letting the sender collect acknowledgments from all receivers. Each message is associated with a unique message number, which the sender obtains by adding one to the previous message number after all acknowledgments arrived. From that point in time, the previous message number is obsolete and can be recycled.

For the case of self-stabilizing node-failure-free systems, the challenge is to deal with integer overflow events. Specifically, when an algorithm considers the counters to be unbounded but the studied system has bounded memory, transient faults can trigger integer overflow events. The solution presented here shows how to overcome this challenge via a mild synchrony assumption and muteness detection.

Independent Round Counters (IRCs). The IRC task (Definition 6.1) considers n independent counters. Each counter, cnt_i, can be incremented only by a unique node, $p_i \in \mathcal{P}$, via the invocation of the $\text{increment}_i()$ operation, which returns the new round number or $\perp$ when the invocation is (temporarily) disabled. Suppose $p_i, p_j \in \mathcal{P}$ are correct. Every node $p_j \in \mathcal{P}$ can fetch cnt_i's value via the invocation of the $\text{fetch}_j(i)$ operation, which returns the most recent and non-fetched cnt_i's value or $\perp$ when such value is currently unavailable. Algorithms that satisfy Definition 6.1 provide an implementation of txAvailable() and rxAvailable(k) by returning increment() $\neq \perp$ and fetch(k) $\neq \perp$, respectively.

Definition 6.1 *(Independent Round Counters (IRCs)).* We define the IRC task using the following requirements.

- **IRC-validity.** Suppose p_j IRC-fetches s from cnt_i. Then, p_i had IRC-incremented cnt_i to s.
- **IRC-integrity-1.** No correct node IRC-fetches a value more than once from the counter of any other correct node (considering the B most recent IRC-fetches). In detail, let $S_{i,j} = (s_0, \ldots, s_x) : x < B$ be a sequence of p_i's round numbers that p_j fetched—we are only interested in B most recent ones, where B is a predefined constant. It holds that $\forall s_y \in S_{i,j} : y < B - 1 \implies s_y + 1 \bmod (B+1) = s_{y+1}$. In other words, $S_{i,j}$ is a sequence that includes all the integers that s takes when incrementing s modulo $B+1$, starting from s_0 and ending at s_x.
- **IRC-integrity-2.** Correct nodes that IRC-fetch numbers from cnt_i do so in the order in which cnt_i was incremented (considering the B most recent IRC-fetches).
- **IRC-preemption.** Suppose p_i IRC-increments cnt_i to s. IRC-increment is (temporarily) disabled until all correct nodes have fetched s from p_i's counter.
- **IRC-completion.** Suppose all correct nodes IRC-fetch p_i's counter infinitely often. Node p_i's IRC-increment is enabled infinitely often.

Bounded message lifetime (BML). Consider a scenario in which, due to a transient fault, p_i's copy of its round counter is smaller than p_j's copy of p_i's counter, say, by $x \in \mathbb{Z}^+$, thus node p_i will have to complete x rounds before p_j could IRC-fetch a non-$\perp$ value. $\text{BAMP}_{n,t}[\text{FC}, t < n/3, \diamond\text{P}_{\text{mute}}, \text{BML}]$ overcomes this challenge by following Assumption 6.1.

Assumption 6.1 *(Bounded message lifetime, BML).* Let R be an execution in which a correct $p_i \in \mathcal{P}$ repeatedly broadcasts the protocol messages and completes an unbounded number of roundtrips with every correct, $p_j \in \mathcal{P}$. Suppose p_j receives message $m(s)$ from p_i immediately before $c \in R$, where $s \in \mathbb{Z}^+$ is the round number. We assume $(rnd_i - s) \bmod (B+1) \leq \lambda$ in c, where rnd_i is p_i's round number, $\lambda \in \mathbb{Z}^+ : \text{channelCapacity} < \lambda < B/6$ is a known upper-bound, channelCapacity is defined in § 2, and B is a predefined bound on integer size.

We would like to clarify that Assumption 6.1 accounts for the scenario where the value of s can differ from rnd_i during Convergence. As mentioned in Section 2, the latter refers to the recovery period that follows the last transient fault occurrence.

Muteness Detection. $\text{BAMP}_{n,t}[\text{FC}, t < n/3, \diamond\text{P}_{\text{mute}}, \text{BML}]$ also provides an SSBFT detector for muteness failures of class $\diamond P_{mute}$.

Algorithm 4: SSBFT IRC; code for p_i.

constants: B: a predefined bound on the integer size, say, $2^{64}-1$;

variables: $cur[\mathcal{P}]=[-1,\ldots,-1]$ and $nxt[\mathcal{P}]=[-1,\ldots,-1]$: a pair of round number arrays. Each entry in each array is associated with a single system node. The entry $cur[i]$ is p_i's current round number. Also, $cur[j]$ stores the most recently received round number from p_j. The array $nxt[]$ stores on the j-th entry the next sequence number to be recycled (once it is delivered to all);

$txLbl[\mathcal{P}]=[0,\ldots,0]$: sender-side round-trip labels that are used in the context of round number $cur[i]$, where $txLbl[j]$ stores the most recently received label from the receiver p_j. Also, the assignment of a new round number to $cur[i]$ leads to $txLbl[j]$'s nullification, cf. line 73.;

$rxLbl[\mathcal{P}]=[0,\ldots,0]$ receiver-side round-trip labels, where $rxLbl[j]$ stores the most recently received label from the sender p_j;

required interfaces: recycle(k), trusted(), invoc(), rtComp(j);

provided interface:

txAvailable() **do** {**return** increment() $\neq \perp$};

rxAvailable(k) **do** {**return** fetch(k) $\neq \perp$};

macro: behind(d,s,c) **do** {**return** $s \in \{x \mod (B+1) : x \in \{c-d\lambda,\ldots,c\}\}$};

operation increment() **begin**

  if $cur[i] \neq -1 \wedge \exists j \in$ trusted()$:txLbl[j] \leq 2($channelCapacity $+1)$ **then return** $\perp$;

  else

    invoc();

    $(cur[i], txLbl[]) \leftarrow ((cur[i]+1) \bmod (B+1), [0,\ldots,0])$;

    recycle(i);

    return $cur[i]$;

operation fetch(k) **begin**

  if behind($1, cur[k], nxt[k]$) **then return** $\perp$;

  else {$nxt[k] \leftarrow cur[k]$; **return** $nxt[k]$};

operation txMSG(j) {**return** $(cur[i], nxt[j], txLbl[j], rxLbl[j])$};

operation rxMSG($brbJ, ircJ = (curJ, nxtJ, tx\ell J, rx\ell J), j$) **begin**

  if behind($2, cur[i], nxtJ$) $\wedge\ txLbl[j] = rx\ell J$ **then**

    rtComp(j);

    $txLbl[j] \leftarrow \min\{B, txLbl[j]+1\}$;

  else

    if $\neg$behind($1, curJ, cur[j]$) **then** {$cur[j] \leftarrow curJ$; recycle(j)};

    $rxLbl[j] \leftarrow tx\ell J$;

do forever foreach $p_j \in \mathcal{P}$ **send** MSG($msg()$, txMSG(j));

upon MSG($brbJ, ircJ$) **arrival from** p_j **begin**

  rxMSG($brbJ, ircJ, j$);

  $mrg(brbJ, j)$;

Muteness Failures. Let us consider an algorithm, *Alg*, that attaches a round number, $seq \in \mathbb{Z}^+$, to every message, $m(seq)$ that it sends. Suppose there is a system state $c_\tau \in R$ after which p_j stops forever replying to p_i's messages, $m(seq)$, where $p_i, p_j \in \mathcal{P}$. In this case, we say that p_j is *mute* to p_i *w.r.t.* message $m(seq)$. We clarify that a Byzantine node is not mute if it forever sends all the messages required by *Alg*. For the sake of a simple presentation, we assume that the syntax of $m(seq)$ corresponds to the syntax of a message generated by *Alg* (since, otherwise, the receiver may simply omit messages with syntax errors). Naturally, the data load of those messages can be wrong. Observe that the set of mute nodes also includes all crashed nodes.

Muteness Detection. The class $\diamond P_{mute}$ of muteness detectors (Definition 6.2) assumes the availability of the interface function, trusted().

Definition 6.2 *(Specifications of eventual muteness failure detector).* The function trusted() returns the set of unsuspected node identifiers, such that:

- **Strong Completeness:** Eventually, every mute node is forever suspected *w.r.t.* round number s by every correct node (or the round number changes).
- **Eventual Strong Accuracy:** Eventually, the system reaches a state $c_\tau \in R$ in which no correct node is suspected.

SSBFT IRC for $\mathsf{BAMP_{n,t}[FC, t < n/3, \diamond P_{mute}, BML]}$**.** Algorithm 4 makes sure that any node that had IRC-incremented its round counter defers any further IRC-increments until all nodes have acknowledged the latest IRC-increment. Algorithm 4's line numbers continue the ones of Algorithm 3. Also, we defer the presentation of the boxed code in lines 51 and 61 until we present the implementation of our muteness detector (Algorithm 5). We clarify that when these boxed code lines are

excluded, Algorithm 4 loses its ability to tolerate node failures, even when considering only crashes. Thus, it is imperative for the correctness proof (§ 7) to consider both the boxed code lines and Algorithm 5 when demonstrating the properties of Algorithm 4. In other words, the division between Algorithms 4 and 5 serves the sole purpose of presentation simplicity.

Constants and variables. All integers used by Algorithm 4 have a maximum value, which we denote by B (line 39), that is required to be large, say, $2^{64}-1$. The arrays $cur[]$ and $nxt[]$ (line 40) store round numbers. The entry $cur[i]$ is p_i's current round number and $cur[j]$ stores the most recently received round number from each other node p_j. The array $nxt[]$ stores on the j-th entry the next sequence number to be recycled (once it is delivered to all). The array $txLbl[]$ holds labels that are used for counting the number of round-trips that p_i is able to complete in the context of the current round number, $cur[i]$, where $txLbl[j]$ is the most recently received label from p_j (line 40). The array $rxLbl[\mathcal{P}] = [0,\ldots,0]$ holds the receiver-side round-trip labels, where $rxLbl[j]$ stores the most recently received label from the sender p_j. These values allow the receiver to acknowledge the arriving labels back to the sender.

Interfaces. We remind the implementation of interface function recycle() (line 43) is provided by Algorithm 3, line 6. Algorithm 4 offers implementations for the interfaces txAvailable() (line 45) and rxAvailable(k) (line 46).

Macros. The macro behind(d,s,c) (line 47) takes three input parameters, *i.e.,* $d \in \{1,2\}$ and two sequence numbers, s and c. It allows Algorithm 4 to test whether s is a member of the set of numbers that are between $c-d\lambda$ and c while taking integer overflow considerations using modulo $B+1$ operations, where λ is a bound on the message lifetime (BML), see Assumption 6.1. Note that the parameter $d \in \{1,2\}$ allows behind(d,s,c) to test whether c is not 'too old' when simple transmissions ($d=1$) or roundtrip exchanges ($d=2$) are considered.

The increment() *operation.* This operation allows the caller to IRC-increment the value of its round number modulo $B+1$ and nullify all labels in $txLbl[]$ (line 52) as well as recycle the relevant BRB object (line 53) and return the new round number (line 54). However, if the previous invocation has not finished, the operation is disabled and the $\perp$ value is returned. Line 49 tests whether the round number is ready to be incremented. In detail, line 49 checks whether this is the first round, *i.e.,* a round number of -1, or the previous round has finished, *i.e.,* the labels indicate that every node has completed at least $2(\mathsf{channelCapacity}+1)$ roundtrips. By exchanging at least $2(\mathsf{channelCapacity}+1)$ labels, the proposed solution overcomes packet loss and duplication over FIFO channels, see [8] for a more efficient variation on this technique that also deals with packet reordering.

The fetch(k) *operation.* This operation returns, exactly once, the most recently received round number. Line 56 tests whether a new round number has arrived. If this is not the case, then $\perp$ is returned. Otherwise, the value of the new round number is returned (line 57). In detail, due to Assumption 6.1, immediately after the arrival of message $m(s)$ to p_j from p_i, the fact that $s \notin \{x \mod (B+1) : x \in \{c-1\lambda,\ldots,c\}\}$ holds implies that s is newer than $cur_j[i]$. Thus, p_i can use behind$_i(1, cur_i[k], nxt_i[k])$ (line 56) for testing the freshness of the round number stored in $cur_i[k]$ *w.r.t.* $nxt_i[k]$. In case the number is indeed fresh, fetch$_i$() updates $nxt_i[k]$ with the returned round number.

The txMSG() *and* rxMSG() *operations.* The operations txMSG() and rxMSG() let the sender, and respectively the receiver, process messages. Algorithm 4 sends via the message $MSG()$ two fields: $brbJ$ and $ircJ$, where the field $brbJ$ corresponds to the one of Algorithm 3. Recall that when a message arrives from p_j, the receiving-side adds the suffix J to the field name, *i.e.,* $brbJ$ and $ircJ$. The field $ircJ$ is composed of the fields *ack*, which indicates whether a reply is required, *seq*, which is the sender's round number, and *lbl*, which, during legal executions, is the corresponding label to *seq* that the sender uses for the receiver. The operation txMSG() (line 58) is used when the sender transmits a message (line 66). It includes the sender's current round number, *i.e.,* $cur[i]$, and the corresponding label that the sender uses for the receiver $p_j \in \mathcal{P}$, *i.e.,* $txLbl[j]$. It also includes the values that are needed for letting the receiver to send an acknowledgment back to the sender, p_j, *i.e.,* $nxt[j]$, and the corresponding label, *i.e.,* $rxLbl[j]$.

The operation rxMSG() processes messages arriving to the sender and the receiver. On the sender-side (lines 60 to 62), when an acknowledgment arrives from the receiver, p_j, the sender checks whether the arriving message has a fresh round number and label (line 60). In this case, the label is incremented (line 62) in order to indicate that at least one roundtrip was completed. In detail, p_i uses behind$_i(2, cur_i[j], sJ)$ for testing whether the arriving round number, sJ, is fresh by asking whether sJ is not a member of the set $\{cur_j[i]-2\lambda,\ldots,cur_j[i]\}$, see Assumption 6.1. As we will see in the next paragraph, there is a need to take into account the receiver's test (line 64), which can cause a non-fresh value to be a member of the set $\{x \mod (B+1) : x \in \{c-2\lambda,\ldots,c\}\}$, but not the set $\{x \mod (B+1) : x \in \{c-\lambda,\ldots,c\}\}$.

On the receiver-side (lines 64 and 65), p_i uses behind$_i(1, sJ, cur_i[j])$ to test whether a new round number arrived, *i.e.,* testing whether the arriving number, sJ, is a member of $\{cur_j[i]-\lambda,\ldots,cur_j[i]\}$. In this case, the local round number is updated (line 64) and the interface function recycle$_i(j)$ is called. Note that whenever the receiver gets a message, it stores the arriving sender's label (line 65) so that it could be acknowledged (line 66). That acknowledgment includes the most recently delivered round number, *i.e.,* $nxt[i]$, and the arriving label, ℓJ's value, which is stored at $rxLbl_i[j]$.

The 'do forever loop' and message arrival. Note that the processing of messages (for sending and receiving) is along the lines of Algorithm 3. The 'do forever loop' broadcasts the message MSG() to every node in the system (line 66). The operation txMSG() is used for setting the value of the $ircJ$ field. Upon message arrival, as in Algorithm 3, the receiver merges the arriving information (line 67) and passes the arriving values to rxMSG() for processing.

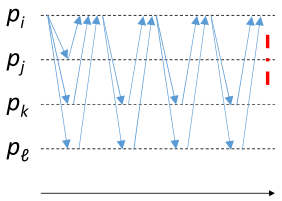

Fig. 2. Blanchard et al.'s implementation with $\Theta = 6$ and no Byzantine failures. In this scenario, node p_j crashes after a single communication round. The dashed line marks the point in time in which p_i suspects p_j to be faulty once nodes p_k and p_ℓ complete three additional rounds. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

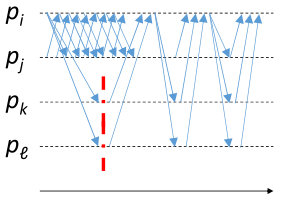

Fig. 3. Blanchard et al.'s implementation with $\Theta = 6$ and a single Byzantine node, p_j. In this scenario, p_j sends six fake replies to messages before their arrival. This leads p_i to falsely suspect the correct nodes p_k and p_ℓ, see the dashed red line.

Our SSBFT implementation of $\diamond P_{mute}$. As mentioned, Algorithm 4 is our SSBFT recycling mechanism for $\mathsf{BAMP}_{n,t}[\mathsf{FC}, t < n/3, \diamond\mathsf{P}_{\mathsf{mute}}, \mathsf{BML}]$, which requires the implementation of $\diamond P_{mute}$ (Definition 6.2). We present our $\diamond P_{mute}$'s implementation (Algorithm 5) and its interface with Algorithm 4, see the boxed code lines. Algorithm 5's line numbers continue the ones of Algorithm 4. Specifically, Algorithm 4 lets p_i restart the local state of the muteness detector via a call to $\mathsf{invoc}_i()$ (line 51). It uses $\mathsf{rtComp}_i(j)$ (line 61) for taking into account the completion of a roundtrip between p_i and p_j.

Related solutions. In the context of self-stabilizing Byzantine-free (crash-prone) systems, Blanchard et al. [37] designed a solution for crash-tolerant systems, rather than BFT one. They implemented perfect failure detectors, specifically class P, by allowing node p_i to suspect any node $p_j \in \mathcal{P}$ when p_i is able to complete Θ roundtrips with other nodes in $\mathcal{P}$ but not with p_j, where Θ is a predefined constant. Fig. 2 illustrates the fact that any node p_j that has crashed is unable to respond to messages from p_i, while any correct node can successfully complete roundtrips with p_i. As p_i completes more and more roundtrips with the correct nodes, it eventually suspects node p_j since no roundtrips are completed with p_j. Blanchard et al.'s solution protects the system against adversarial attacks that cause all message deliveries to be deferred in the network, as it is free from assumptions regarding the time it takes to complete a round trip. Blanchard et al.'s time-free solution simply requires the knowledge of an appropriate value for Θ.

Since the studied fault model includes Byzantine failures, we cannot directly borrow such earlier proposals, such as the one by Blanchard et al. Consider, for example, a Byzantine node that anticipates the sender's messages and transmits acknowledgments before the arrival of prospective messages. Using this attack of speculative acknowledgments, the adversity may accelerate the (false) completion of roundtrips and let the unreliable failure-detector suspect non-faulty nodes. Fig. 3 illustrates such attacks on Blanchard et al., which was designed to be crash-tolerant rather than BFT.

Our solution in a nutshell. As we explain next, our solution relies on Assumption 6.2, which facilitates the defense against the above attacks that use speculative acknowledgments. Specifically, when testing whether the Θ threshold has been exceeded, p_i ignores the roundtrips that were completed with the top t nodes, say *w.l.o.g.*, $p_1, \ldots, p_t$, that had the highest number of roundtrips with p_i. Suppose, *w.l.o.g.*, that nodes $p^{byz}_{n-t}, \ldots, p^{byz}_{n-1}$ are captured by the adversary. On the one hand, the adversary aims at letting $p^{byz}_{n-t}, \ldots, p^{byz}_{n-1}$ to rapidly complete roundtrips with p_i (since it tries to fool the muteness detector and cause it to suspect correct nodes). While on the other hand, if any of the nodes $p^{byz}_{n-t}, \ldots, p^{byz}_{n-1}$ complete roundtrips with p_i faster than any of the nodes $p_1, \ldots, p_t$, the former, p^{byz}_k, is ignored by p_i when testing whether the Θ threshold has been exceeded. In other words, any adversarial strategy that lets any of the nodes $p^{byz}_{n-t}, \ldots, p^{byz}_{n-1}$ to complete more roundtrips with p_i than the nodes $p_1, \ldots, p_t$ cannot cause a 'haste' muteness detection of a correct node. For example in Fig. 3, the proposed muteness detector of node p_i would have ignored p_j's fake replies, and thus, never falsely suspects p_k and p_ℓ.

Implementation. As shown in Fig. 4, Algorithm 5 does not send independent messages as it merely provides three interface functions to Algorithm 4, *i.e.,* invoc(), rtComp(j), and trusted(). The algorithm's state is based on the array $rt[][]$ (line 71), which stores the number of roundtrips that node p_i has completed with p_j. Note that $rt[][]$ counts separately the number of roundtrips p_i and p_k can complete during any period in which p_i and p_j are attempting to complete a single roundtrip.

The function invoc() (line 73) nullifies $rt[][]$. We require p_i, upon completion of a roundtrip with p_j, to call rtComp(j) (lines 74 to 76). It increments, for every $p_k \in \mathcal{P} \setminus \{p_i, p_j\}$, the counter in $rt_i[k][j]$ (line 75). Then, rtComp(j) assigns zero

Algorithm 5: Class $\diamond P_{mute}$ detector; code for p_i.

constants: B: a predefined bound on the integer size, say, $2^{64}-1$.
variables: $rt[\mathcal{P}\setminus\{p_i\}][\mathcal{P}\setminus\{p_i\}]$: roundtrip counters, initially all entries are zero;
interface functions:
invoc() **do** $\{rt \leftarrow [[0,\ldots,0],\ldots,[0,\ldots,0]]\}$;
rtComp(j) **begin**
  foreach $p_k \in \mathcal{P}\setminus\{p_i,p_j\}$ **do** $rt[k][j] \leftarrow \min\{B, rt[k][j]+1\}$;
  $rt[j] \leftarrow [0,\ldots,0]$;
trusted() **do return** $\{p_j : \Theta > \sum_{x\in withoutTopItems(t,j)} x\}$ **where** $\{rt[j][\ell]\}_{p_\ell\in\mathcal{P}}$ is a multi-set with all the values in $rt[j][]$ and $withoutTopItems(t,j)$ is the same multi-set after the removal of the top t values;

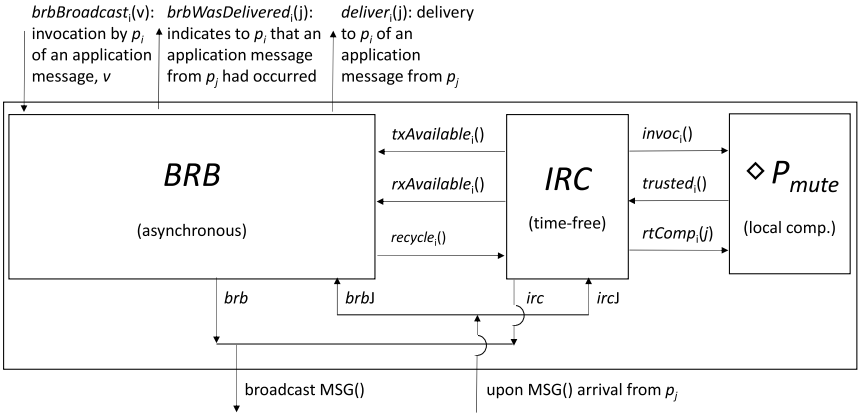

Fig. 4. Integrating BRB (§ 4) as well as IRC and $\diamond P_{mute}$ (§ 6).

to every entry in $rt_i[j]$ (line 76). The function trusted() returns the set of unsuspected nodes. Its implementation relies on Assumption 6.2, which answers the above challenge of speculative acknowledgments. As a defense against the above attacks, p_i ignores the top t roundtrip counters when testing whether the Θ threshold has been exceeded.

Assumption 6.2. Let R be an execution, $p_i \in \mathcal{P}$ a correct node that repeatedly broadcasts $m(s) : s \in \mathbb{Z}^+$ and completes an unbounded number of roundtrips with every correct node. Let $rt_{i,c} : \mathcal{P}\times\mathcal{P} \to Z^+$ be a mapping of $p_j, p_k \in \mathcal{P}$ with the number of roundtrips that p_i has completed with p_k between $c' \in R$ and $c \in R$, where c' is the first system state that immediately follows the last time p_i has completed a roundtrip with p_j, or the start of R (in case p_i has not completed any roundtrip with p_j between R's start and c). Let $\sum_{x\in withoutTopItems_{i,c}(t,j)} x$ be the total number of roundtrips that p_i has completed until c when excluding the top t values of $rt_{i,c}$ of nodes that have completed with p_i the greatest number of roundtrips. We assume that if $\Theta \le \sum_{x\in withoutTopItems_{i,c}(t,j)} x$ then p_j is mute to p_i *w.r.t.* $m(s)$, where Θ is a predefined constant.

7. Correctness of Algorithm 4 and Algorithm 5

Our correctness proof shows Convergence (Theorem 7.1) and Closure (Theorem 7.21). Theorem 7.1 demonstrates that the system completes the first part of the Convergence within $\mathcal{O}(\Theta + \text{channelCapacity}\cdot\lambda)$ asynchronous cycles assuming execution fairness among the correct nodes, as explained in § 2. It does so by showing that the system reaches a consistent system state, which Definition 7.1 specifies. This proof also includes the demonstration, in Claims 7.6 and 7.7, that Algorithm 5 satisfies $\diamond P_{mute}$'s requirements (Definition 6.2). Once Convergence is done, no execution fairness is needed when demonstrating Closure. This is done in Theorem 7.21, which shows that Algorithms 4 and 5 satisfy the requirements of the IRC task (Definition 6.1). Theorem 7.21's proof uses Theorem 7.11, which states that IRC-completion and $\diamond P_{mute}$'s requirements can be demonstrated using similar arguments to the ones in Theorem 7.1 while assuming a consistent starting state (and without assuming execution fairness).

Definition 7.1 *(Consistent system states for the IRC task in Definition 6.1).* Let R be an execution of Algorithm 4, $p_i, p_j \in \mathcal{P}$, and $c \in R$ be a system state of R. We say that c is consistent if the following holds.

1. $txLbl_i[j] = 0 \wedge cur_i[i] = (cur_j[i]+1) \bmod (B+1) \wedge cur_j[i] = (nxt_j[i]+1) \bmod (B+1)$ holds in c.
2. Given $c \in R$ and $s = cur_i[i]$, let $L_{c,i}(s)$ be all pairs (s,ℓ) in MSG(-, ($sJ = s$, -, $\ell J = \ell$, -)) message in transit $channel_{i,j}$ and MSG(-, (-, $sJ = s$, -, $\ell J = \ell$)) in $channel_{j,i}$ as well as p_i's variables, $cur_i[i] = s$ and $txLbl_i[j] = \ell$, and p_j's variables,

$cur_j[i] = s$ and $rxLbl_j[i]$ (line 59). $L_{c,i}(s)$ does not include values that were present in every system state between R's starting state and c. *I.e.,* every value in $L_{c,i}(s)$ was introduced (or reintroduced) between R's start and c.

Theorem 7.1 *(Convergence of Algorithms 4 and 5). Let R be an execution of Algorithm 4 for which we assume fairness among the correct nodes, as explained in § 2. We also assume that Algorithm 4 uses the procedures presented in Algorithm 5 and that R satisfies Assumption 6.2. Let p_i be a correct node. Throughout R, suppose all correct nodes, p_j, IRC-fetch p_i's counter, at least once in every $\mathcal{O}(1)$ asynchronous cycles, and p_i invokes IRC-increment, at least once in every $\mathcal{O}(1)$ asynchronous cycles. Within $\mathcal{O}(\Theta + \mathsf{channelCapacity} \cdot \lambda)$ asynchronous cycles, the system reaches a consistent system state (Definition 7.1).*

Proof of Theorem 7.1. The proof is implied by Lemmas 7.2 to 7.10.

Lemma 7.2. *Within $\mathcal{O}(\Theta + \mathsf{channelCapacity})$ asynchronous cycles, IRC-increment is enabled at least once,* i.e., $\mathsf{increment}_i()$ *returns a non-$\perp$ value.*

Proof of Lemma 7.2. Whenever p_i's IRC-increment is enabled (lines 50 to 54), the return of a non-$\perp$ value implies that the value of $cur_i[i]$ changes. For the case of $cur_i[i] = -1$, IRC-increment is clearly enabled (line 49). Thus, towards a contradiction, assume $cur_i[i] = s \geq 0$ holds in every system state of R', where R' is a prefix of R that has at least $\mathcal{O}(\mathsf{channelCapacity})$ asynchronous cycles.

Claims 7.3 to 7.8 show the needed contradiction since the latter claim implies the execution of line 52, which increments $cur_i[i]$.

Claim 7.3. *Within $\mathcal{O}(1)$ asynchronous cycles, the system reaches a state in which $\mathsf{behind}_j(1, cur_j[i] = s, cur_j[i])$ holds.*

Proof of Claim 7.3. Within $\mathcal{O}(1)$ asynchronous cycles, node p_i broadcasts MSG(-, $(cur_i[i] = s, \bullet)$) (line 66). Thus, p_j receives MSG(-, $(cur_i[i] = s, \bullet)$) within $\mathcal{O}(1)$ asynchronous cycles. In the system state that immediately follows this message arrival (lines 67 to 68), the condition $\mathsf{behind}_j(1, s, cur_j[i])$ holds due to the if-statement and its assignment to $cur_j[i]$ in line 64. □Claim 7.3

Claim 7.4. *Within $\mathcal{O}(1)$ asynchronous cycles, the system reaches a state in which $\mathsf{behind}_j(1, cur_j[i] = s, nxt_j[i])$ holds.*

Proof of Claim 7.4. By the assumption that p_j invokes $\mathsf{fetch}_j(i)$ within $\mathcal{O}(1)$ asynchronous cycles, we know that the if-statement condition in line 56 eventually holds. Since otherwise the if-statement condition in line 56 does not hold and the assignment $nxt_j[i] \leftarrow cur_j[i]$ (line 57) occurs. □Claim 7.4

Claim 7.5. *Within $\mathcal{O}(1)$ asynchronous cycles, the system reaches a state in which the if-statement condition in line 60 holds* w.r.t. p_i.

Proof of Claim 7.5. By the proof of Claim 7.3, within $\mathcal{O}(1)$ asynchronous cycles, p_i sends MSG(-, $(cur_i[i] = s, -, txLbl_i[j] = \ell, -)$) to p_j and then p_j receives MSG(-, $(s, -, \ell, -)$) and assign ℓ to $rxLbl_j[i]$ (line 65) before replying with MSG(-, $(-, nxt_j[i] = s', -, \ell)$) (line 66). By Claims 7.3 and 7.4, we know that both $\mathsf{behind}_j(1, cur_j[i] = s, cur_j[i])$ and $\mathsf{behind}_j(1, cur_j[i] = s, nxt_j[i] = s')$ hold. By the assumption that $cur_i[i] = s$ holds throughout R' and the definition of $\mathsf{behind}_j()$ (line 47), we know that $\mathsf{behind}_j(2, cur_i[i] = s, nxt_j[i] = s')$ holds when p_j's reply, MSG(-, $(-, nxt_j[i] = s', -, \ell)$), arrives at p_i. The rest of the proof is implied from lines 67 to 68 and then lines 59 to 60. □Claim 7.5

Claim 7.6. *Within $\mathcal{O}(\Theta)$ asynchronous cycles, R demonstrates the property of Strong Completeness (Definition 6.2).*

Proof of Claim 7.6. Let us consider the sequence of values of $rt_i[j'][k]$ in the different system states $c \in R$, where $p_i, p_{j'}, p_k \in \mathcal{P}$. Note that this sequence is defined by the function $rt_{i,c}(k, j')$ (Assumption 6.2). Thus, by line 77, we know that $j' \in \mathsf{trusted}_i()$ if, and only if, $\Theta \leq \sum_{x \in withoutTopItems_{i,c}(t,j')} x$.

We demonstrate Strong Completeness in R by showing that, within $\mathcal{O}(\Theta)$ asynchronous cycles, every mute node, $p_{j'}$, is forever suspected *w.r.t.* round number $cur_i[i] = s$ (or the round number in $cur_i[i]$ becomes different than s).

Argument (1) *Within $\mathcal{O}(1)$ asynchronous cycles, p_i calls $\mathsf{rtComp}_i(j')$ (line 61).*

By Claim 7.4, the if-statement condition in line 64 holds within $\mathcal{O}(1)$ asynchronous cycles. By Claim 7.5, line 61 is executed within $\mathcal{O}(1)$ asynchronous cycles.

Argument (2) *Within $\mathcal{O}(\Theta)$ asynchronous cycles, every mute node, $p_m \in \mathcal{P}$, is forever suspected (*w.r.t. *round number s) by every correct node (or the round number changes and stops being s).* By Argument (1), p_i calls $\mathsf{rtComp}_i(j')$ (line 61) within $\mathcal{O}(1)$ asynchronous cycles. *I.e.,* for every correct node $p_k \in \mathcal{P} \setminus \{p_i, p_{j'}\}$, the value of $rt_i[k][j']$ is incremented, every $\mathcal{O}(1)$ asynchronous cycles. Thus, within $\mathcal{O}(\Theta)$ asynchronous cycles $p_{j'} \notin \mathsf{trusted}_i()$ holds. □Claim 7.6

Claim 7.7. *Within $\mathcal{O}(\Theta)$ asynchronous cycles, R demonstrates the property of Eventual Strong Accuracy (Definition 6.2).*

Proof of Claim 7.7. As in the proof of Claim 7.6, consider the sequence $rt_i[j'][k]$, which is defined by $rt_{i,c}(k, j')$ (Assumption 6.2). *I.e.,* $j' \in \text{trusted}_i()$ if, and only if, $\Theta \leq \sum_{x \in withoutTopItems_{i,c}(t,j')} x$. We demonstrate Eventual Strong Accuracy in R by showing that, within $\mathcal{O}(\Theta)$ asynchronous cycles, the system reaches a state $c_\tau \in R$ in which every correct node, $p_{j'}$, is not suspected *w.r.t.* round number $cur_i[i] = s$ (or the round number in $cur_i[i]$ becomes different than s). By Argument (1) of Claim 7.6, since both p_i and $p_{j'}$ are correct, we know that p_i completes a roundtrip (that has a fresh label) with $p_{j'}$ within $\mathcal{O}(\Theta)$ asynchronous cycles. Whenever such a roundtrip is completed, p_i assigns $[0, \ldots, 0]$ to $rt_i[j']$ (due to line 76) and the condition $\Theta > \sum_{x \in withoutTopItems_i(t,j')} x$ (line 77) holds until the next roundtrip completion (Assumption 6.2). □Claim 7.7

Claim 7.8. *Suppose R satisfies $\diamond P_{mute}$'s requirements (Definition 6.2). Within $\mathcal{O}$(channelCapacity) asynchronous cycles, the system reaches a state in which the if-statement condition in line 49 does not hold* w.r.t. p_i.

Proof of Claim 7.8. By the proof of Claim 7.3, within $\mathcal{O}(1)$ asynchronous cycles, p_i sends MSG(-, ($cur_i[i] = s$, -, $txLbl_i[j] = \ell$, -)) to every $p_j \in \mathcal{P}$. Then, p_j receives MSG(-, (s, -, ℓ, -)) before assigning ℓ to $rxLbl_j[i]$ (line 65). By Claim 7.14, $nxt_j[i]$ stores the value s' for which $\text{behind}_j(1, s, s')$ holds eventually. Thus, p_j eventually send to p_i the message MSG(-, (-, $nxt_j[i] = s'$, -, ℓ)) (line 66). By Claim 7.5, the arrival of the latter message to p_i implies that p_i increments $txLbl_i[j]$ or that $txLbl_i[j] = B$ (line 62). Therefore, within $\mathcal{O}$(channelCapacity) asynchronous cycles, there is no trusted node $p_j \in \text{trusted}_i()$ for which $txLbl_i[j] \leq 2(\text{channelCapacity} + 1)$ holds. Thus, the if-statement condition in line 49 does not hold *w.r.t.* p_i. □Claim 7.8 □Lemma 7.2

Lemma 7.9. *Within $\mathcal{O}(\Theta + \text{channelCapacity} \cdot \lambda)$ asynchronous cycles, p_i executes line 52, which causes p_j to execute line 64 and leads the system to a state in which for all nodes, $p_j \in \mathcal{P}$, it holds that $cur_j[i] = cur_i[i]$.*

Proof of Lemma 7.9. Let us consider a suffix, R', of execution R, of length $\mathcal{O}(\Theta + \text{channelCapacity} \cdot \lambda)$ asynchronous cycles, during which there is at least one node, $p_j \in \text{trusted}_i()$, for which $cur_j[i] \neq cur_i[i]$ holds throughout R'. From all the possible choices, let p_j to be a node for which, at the starting system state of R', the value of $x_{i,j} = (cur_j[i] - cur_i[i]) \bmod (B + 1)$ is higher (or equal) to the value of $x_{i,k} = (cur_k[i] - cur_i[i]) \bmod (B + 1)$, for any $p_k \in \text{trusted}_i()$.

Argument (1) *Only line 52 can introduce a new p_i's round number value to the system, which p_j may assign to $cur_j[i]$.* By the code of Algorithm 4, only lines 52 and 64 update the value of $cur[i]$. The former is at the sender-side, *i.e.,* $cur_i[i]$, whereas the latter is at the receiver-side, *i.e.,* $cur_j[i]$.

Argument (2) *The assignment $cur_j[i] \leftarrow s$ occurs within $\mathcal{O}(\Theta + \text{channelCapacity} \cdot \lambda)$ asynchronous cycles, where s is the value of $cur_i[i]$ immediately after it was incremented.* By Claim 7.8, within $\mathcal{O}$(channelCapacity) asynchronous cycles, p_i executes line 52, which increments the value of $cur_i[i]$, say, to s. Also, between any consecutive pair of such increments, p_j executes line 64 (see the proof of Claim 7.3), which tests whether $\neg\text{behind}(1, s, cur_j[i])$ holds, where s is a round number that arrived from p_i after it was assigned to $cur_i[i]$ in line 52. If this if-statement condition holds, the assignment $cur_j[i] \leftarrow s$ occurs. By behind()'s definition and the pigeonhole principle, there are at most λ increments to $cur_i[i]$ without reaching a state in which $\neg\text{behind}(1, s, cur_j[i])$ does not hold.

Argument (3) *The claim holds.* By Claim 7.8 and Argument (2), (i) $cur_i[i]$ is incremented to s within $\mathcal{O}(\Theta + \text{channelCapacity} \cdot \lambda)$ asynchronous cycles, (ii) the value s arrives to p_j, (iii) the value s is assigned to $cur_j[i]$. Due to Argument (1) and Assumption 6.1, (iv) $cur_j[i]$'s value does not change before a new assignment to $cur_i[i]$ occurs. Invariants (i) to (iv) imply the lemma. □Lemma 7.9

Lemma 7.10. *Within $\mathcal{O}(\Theta + \text{channelCapacity} \cdot \lambda)$ asynchronous cycles, the system reaches a consistent state (Definition 7.1).*

Proof of Lemma 7.10. We show that, within $\mathcal{O}(\Theta + \text{channelCapacity} \cdot \lambda)$ asynchronous cycles, invariants (1) and (2) hold. Within $\mathcal{O}(\Theta + \text{channelCapacity} \cdot \lambda)$ asynchronous cycles, p_i executes line 52, which causes p_j to execute line 64 and for $cur_j[i] = cur_i[i]$ to hold (Lemma 7.9). Let c be a system state that imminently follows the second time in which p_i executes line 52 in a way that leads to a state in which for all nodes, $p_j \in \mathcal{P}$, it holds that $cur_j[i] = cur_i[i]$.

Argument (1) *Definition 7.1's Invariant (1) holds.* In c, $txLbl_i[j] = 0$ holds due to the assignment in line 52. Also, by the proof of Lemma 7.9, such changes to the value of $cur_i[i]$, via increments by one module $B + 1$, introduce to the system state a round number that line 64 uses to update $cur_j[i]$. Imminently after such a change, $cur_i[i] = (cur_j[i] + 1) \bmod (B + 1)$ holds (Lemma 7.9). By the proof of Claim 7.4, the (first) change also leads to $cur_j[i] = nxt_j[i]$. Since c imminently follows the second change, both $cur_i[i] = (cur_j[i] + 1) \bmod (B + 1)$ and $cur_j[i] = (nxt_j[i] + 1) \bmod (B + 1)$ hold.

Argument (2) *Definition 7.1's Invariant (2) holds.* The proof is based on the one in [6, Ch. 4.2]. Due to Assumption 6.1, our proof is simpler. Let $c' \in R$. By Algorithm 4 and Definition 7.1, $L_{c',i}(s)$ cannot include more than $2(\text{channelCapacity} + 1)$ different pairs since (a) $channel_{i,j}$ and $channel_{j,i}$ include at most channelCapacity pairs each as well as (b) $cur_i[i]$ and $txLbl_i[j]$ are

the only variable in c that may contribute a unique set to $L_{c',i}(s)$, because ℓJ (line 59) is an automatic (temporary) variable. Node p_i cannot change $cnt_i[i]$'s value twice before counting the reception of 2(channelCapacity + 1) different labels (line 49). Invariant (2) holds since during the period in which $\mathsf{increment}_i()$ returns non-$\perp$ values twice, the messages in the FIFO-channels (between p_i and p_j) and p_i's variables, $cur_i[i]$ and $txLbl_i[i]$, do not hold values that have not changed since the starting system state of R'. □Lemma 7.10 □Theorem 7.1

The proof of Theorem 7.11 is written using similar arguments to the ones in Theorem 7.1 without requiring fairness yet assuming that the system starts in a consistent state (Definition 7.1). This is done by letting the word 'eventually' substitute the parts in the proofs that quantify time in terms of asynchronous cycles, e.g., 'within $\mathcal{O}(1)$ asynchronous cycles'. We also note that when a correct node, p_i, sends, infinitely often, message m to another correct node, p_j, we know that m arrives at p_j infinitely often (due to the fair communication assumption, see § 2). Thus, this substitution satisfies the requirement to complete the roundtrips (cf. complete iterations in § 2) without requiring execution fairness. Similarly, it is also possible to show the satisfaction of Definition 6.2's requirements for $\diamond P_{mute}$'s class of muteness detectors. We bring hereafter the entire proof for the sake of completeness.

Theorem 7.11 *(IRC-completion and Algorithm 5's Closure). Let R be an Algorithm 4's execution that starts in a consistent system state (Definition 7.1). We also assume that Algorithm 4 uses the procedures presented in Algorithm 5 and that R satisfies Assumption 6.2. Let p_i be a correct node. Throughout R, suppose all correct nodes, p_j, IRC-fetch p_i's counter, infinitely often, and p_i invokes IRC-increment, infinitely often. Eventually, R demonstrates IRC-completion (Definition 6.1). Also, R satisfies Definition 6.2's requirements for class $\diamond P_{mute}$ muteness detector.*

Proof of Theorem 7.11. The proof is implied by Lemmas 7.12 to 7.20.

Lemma 7.12. *Eventually, IRC-increment is enabled at least once,* i.e., $\mathsf{increment}_i()$ *returns a non-$\perp$ value.*

Proof of Lemma 7.12. Whenever p_i's IRC-increment is enabled (lines 50 to 54), the return of a non-$\perp$ value implies that the value of $cur_i[i]$ changes. For the case of $cur_i[i] = -1$, IRC-increment is clearly enabled (line 49). Thus, towards a contradiction, assume $cur_i[i] = s \geq 0$ holds in every system state of R', where R' is a finite prefix of R. Claims 7.13 to 7.18 show the needed contradiction since the latter claim implies the execution of line 52, which increments $cur_i[i]$.

Claim 7.13. *Eventually* $\mathsf{behind}_j(1, cur_j[i] = s, cur_j[i])$ *holds.*

Proof of Claim 7.13. Eventually, p_i broadcasts MSG(-, $(cur_i[i] = s, \bullet)$) (line 66). Thus, p_j receives MSG(-, $(cur_i[i] = s, \bullet)$) eventually. In the state that immediately follows this message's arrival (lines 67 to 68), the condition $\mathsf{behind}_j(1, s, cur_j[i])$ holds due to the assignment to $cur_j[i]$ in line 64. □Claim 7.13

Claim 7.14. *Eventually,* $\mathsf{behind}_j(1, cur_j[i] = s, nxt_j[i])$ *holds.*

Proof of Claim 7.14. By the assumption that p_j invokes $\mathsf{fetch}_j(i)$ eventually, the if-statement in line 56 eventually holds due to line 57. □Claim 7.14

Claim 7.15. *Eventually, the if-statement condition in line 60 holds* w.r.t. p_i.

Proof of Claim 7.15. By Claim 7.13, eventually, p_i sends MSG(-, $(cur_i[i] = s, -, txLbl_i[j] = \ell, -)$) to p_j and then p_j receives MSG(-, $(s, -, \ell, -)$) before assigning ℓ to $rxLbl_j[i]$ (line 65). By Claim 7.14, $nxt_j[i]$ stores the value s' for which $\mathsf{behind}_j(1, s, s')$ holds eventually. Thus, p_j eventually send to p_i the message MSG(-, $(-, nxt_j[i] = s', -, \ell)$) (line 66). By Claims 7.13 and 7.14, both $\mathsf{behind}_j(1, cur_j[i] = s, cur_j[i])$ and $\mathsf{behind}_j(1, cur_j[i] = s, nxt_j[i] = s')$ hold. By the assumption that $cur_i[i] = s$ holds throughout R' and the definition of $\mathsf{behind}_j()$ (line 47), we know that $\mathsf{behind}_j(2, cur_i[i] = s, nxt_j[i] = s')$ holds when p_j's reply, MSG(-, $(-, nxt_j[i] = s', -, \ell)$), arrives at p_i. The rest of the proof is implied from lines 67 to 68 and then lines 59 to 60. □Claim 7.15

Claim 7.16. *Eventually, Strong Completeness (Definition 6.2) holds.*

Proof of Claim 7.16. As in Claim 7.6, consider the sequence $rt_i[j'][k]$. We demonstrate Strong Completeness in R by showing that, eventually, every mute node, $p_{j'}$, is forever suspected *w.r.t.* round $cur_i[i] = s$ (or $cur_i[i]$ becomes different than s).

Argument (1) *Eventually, p_i calls* $\mathsf{rtComp}_i(j')$ *(line 61).* The if-statement in line 60 holds eventually (Claim 7.15). Thus, line 61 is executed eventually.

Argument (2) *Eventually, every mute node, $p_m \in \mathcal{P}$, is forever suspected* w.r.t. *s by every correct node (or the round number changes).* By Argument (1), p_i calls $\mathsf{rtComp}_i(j')$ (line 61) eventually. *I.e.,* every correct $p_k \in \mathcal{P} \setminus \{p_i, p_{j'}\}$, $rt_i[k][j']$ is incremented, infinitely often. Eventually $p_{j'} \notin \mathsf{trusted}_i()$. □Claim 7.16

Claim 7.17. *Eventually, Eventual Strong Accuracy (Definition 6.2) holds.*

Proof of Claim 7.17. As in Claim 7.6, consider the sequence $rt_i[j'][k]$. We demonstrate Eventual Strong Accuracy in R by showing that, eventually, the system reaches $c_\tau \in R$ in which every correct $p_{j'}$ is not suspected *w.r.t.* $cur_i[i] = s$ (or $cur_i[i]$ becomes different than s). By Argument (1) of Claim 7.16, since both p_i and $p_{j'}$ are correct, we know that p_i completes a roundtrip (that has a fresh label) with $p_{j'}$ eventually. Whenever such a roundtrip is completed, p_i assigns $[0, \ldots, 0]$ to $rt_i[j']$ (due to line 76) and $\Theta > \sum_{x \in withoutTopItems_i(t,j')} x$ (line 77) holds until the next roundtrip completion. □Claim 7.16

Claim 7.18. *Suppose R satisfies $\diamond P_{mute}$'s requirements (Definition 6.2). Eventually, the if-statement condition in line 49 does not hold* w.r.t. p_i.

Proof of Claim 7.18. By the proof of Claim 7.13, p_i sends MSG(-, ($cur_i[i] = s$, -, $txLbl_i[j] = \ell$, -)) to every $p_j \in \mathcal{P}$ eventually. Then, node p_j receives MSG(-, (s, -, ℓ, -)) before assigning ℓ to $rxLbl_j[i]$ (line 65). By Claim 7.14, $nxt_j[i]$ stores the value s' for which $\mathsf{behind}_j(1, s, s')$ holds eventually. Thus, p_j eventually send to p_i the message MSG(-, (-, $nxt_j[i] = s'$, -, ℓ)) (line 66). By Claim 7.15, the arrival of the latter message to p_i implies that p_i increments $txLbl_i[j]$ or that $txLbl_i[j] = B$ (line 62). Therefore, eventually, there is no $p_j \in \mathsf{trusted}_i()$ for which $txLbl_i[j] \leq 2(\mathsf{channelCapacity} + 1)$ holds. Thus, the if-statement condition in line 49 does not hold *w.r.t.* p_i. □Claim 7.18 □Lemma 7.12

Lemma 7.19. *Eventually, p_i executes line 52, which causes p_j to execute line 64 and leads the system to a state in which for all nodes, $p_j \in \mathcal{P}$, it holds that $cur_j[i] = cur_i[i]$.*

Proof of Lemma 7.19. Eventually, p_i executes line 52, which causes p_j to execute line 64 and for $cur_j[i] = cur_i[i]$ to hold (Lemma 7.19). Let c be a system state that imminently follows the second time in which p_i executes line 52 in a way that leads to a state in which for all nodes, $p_j \in \mathcal{P}$, it holds that $cur_j[i] = cur_i[i]$. The rest of the proof demonstrates that Definition 7.1's invariants (1) and (2) hold. The proof is identical to the ones of Claim 7.10's arguments (1) and (2). □Lemma 7.19

Lemma 7.20. *Eventually, the system reaches a consistent state (Definition 7.1).*

Proof of Lemma 7.20. Eventually, p_i executes line 52, which causes p_j to execute line 64 and for $cur_j[i] = cur_i[i]$ to hold (Lemma 7.19). Let c be a system state that imminently follows the second time in which p_i executes line 52 in a way that leads to a state in which for all nodes, $p_j \in \mathcal{P}$, it holds that $cur_j[i] = cur_i[i]$. The rest of the proof is identical to arguments (1) and (2) in Lemma 7.10. □Lemma 7.20 □Theorem 7.11

Theorem 7.21 *(Algorithm 4's Closure). Let $p_i \in \mathcal{P}$ be a correct node and R be an Algorithm 4's execution that starts in a consistent system state (Definition 7.1) as well as Algorithm 5 satisfies the requirements for class $\diamond P_{mute}$ muteness detector (Definition 6.2). Suppose all correct nodes, p_j, IRC-fetch p_i's counter infinitely often and p_i invokes IRC-increment infinitely often. R demonstrates an IRC construction (Definition 6.1).*

Proof of Theorem 7.21. IRC-completion is implied by Theorem 7.11.

IRC-validity. Suppose a correct node, p_j, IRC-fetches s in $a_j \in R$ from p_i's counter. We show that p_i had IRC-incremented cnt_i to s in step a_i, which appears in R before a_j. Suppose, towards a contradiction, $\nexists a_i \in R$, yet a_j returns $s \neq \perp$ from $\mathsf{fetch}_j(i)$ when executing line 57.

By the assumption that a_j returns $s \neq \perp$ from $\mathsf{fetch}_j(i)$, we know that $cur_j[i] = s$ immediately before a_j (line 57). Node p_j assigns s to $cur_j[i] : i \neq j$ only in line 64 when it processes a message coming from the sender p_i. However, p_i can assign s to $cur_i[i]$ only at line 52, *i.e.,* a_i exists.

IRC-preemption. Suppose that there is a correct node, $p_k \in \mathcal{P}$, that does not IRC-fetch s from p_i's round counter during R even after a_i (in which p_i IRC-increment cnt_i to $s \neq -1$). Also, let a'_i be a step that appears in R after a_i and includes an IRC-increment invocation by p_i. We show that a'_i's invocation returns $\perp$, *i.e.,* a'_i's invocation is disabled.

Since there is no step in which $\mathsf{fetch}_k(i)$ returns s, it implies that $nxt_k[i] \neq s$ holds in any system state during R (line 57), when considering the last B p_i's IRC-increments. Therefore, p_k does not send MSG(-, (-, $nxt[j] = s, \bullet$)) to p_i (line 66). This means that, as long as $cur_i[i] = s$, it holds that $txLbl_i[j] = 0$. Also, as long as $cur_i[i] = s$, whenever p_i invokes $\mathsf{increment}_i()$, the if-statement condition in line 49 holds. Thus, $\mathsf{increment}_i()$ returns $\perp$ in step a'_i.

IRC-integrity-1. Lines 56 to 57 and behind()'s definition imply that no correct node, p_i, can IRC-fetch the same value twice from the counter of the same node, say, p_j (when considering the last B IRC-increments done by p_i).

IRC-integrity-2. Suppose p_j IRC-fetches s' from p_i's counter in step a'_j that appears in R after a_j (in which p_j IRC-fetches s). We show that p_i IRC-incremented cnt_i to s and then to s'. Step a'_i appears in R after a_j (IRC-preemption, -validity, and line 52) and a_j after a_i (IRC-validity). Since $s \neq s'$ (IRC-integrity-1), $a_i \neq a'_i$ holds. By line 52, s was IRC-incremented before s' when considering the B IRC-increments preceding a'_i. □Theorem 7.21

8. Coordinating the recycling of BRB objects

We sketch an integration approach between the components proposed in this paper. Fig. 4 presents Algorithm 3's interface for recycling of BRB instances, *i.e.,* recycle(), txAvailable(), and rxAvailable() operations. (The interface between IRC and $\diamond P_{mute}$ is irrelevant to Algorithm 3.) The function recycle$_i(k)$ (line 6) lets the recycling mechanism reset $msg_i[k][]$. The notation $f_i()$ says that p_i executes the function $f()$. For the single-instance BRB (without recycling), define txAvailable() and rxAvailable(k) (line 5) to return True. We redefine these implementations in § 6 as a preliminary integration approach between BRB and IRC via message piggybacking (lines 35 to 36).

The two SSBFT algorithms are integrated via specified interfaces and message piggybacking (Fig. 4). Thus, our SSBFT repeated BRB solution increases BT's message size only by a constant per BRB, but the number of messages per instance stays similar. The integrated solution can run an unbounded number of (concurrent and independent) BRB instances. The advantage is that the more communication-intensive component, *i.e.,* SSBFT BRB, is not associated with any synchrony assumptions. Specifically, one can run δ concurrent BRB instances, where δ is a parameter for balancing the trade-off between fault recovery time and the number of BRB instances that can be used (before the next δ concurrent instances can start). The above extension mitigates the effect of the fact that, for the repeated BRB problem, muteness detectors are used and mild synchrony assumptions are made to circumvent well-known impossibilities, *e.g.,* [38]. Those additional assumptions are required for the entire integrated solution to work. To the best of our knowledge, there is no proposal for a weaker set of assumptions for solving the studied problem in a self-stabilizing manner.

The above extension facilitates the implementation of FIFO-ordered delivery SSBFT repeated BRB. Here, each of the δ instances is associated with a unique label $\ell \in \{0, \ldots, \delta - 1\}$. The implementation makes sure that no node p_i delivers a BRB message with label $\ell > 0$ before all the BRB messages with labels in $\{0, \ldots, \ell - 1\}$. (For the case of $\ell = 0$, the delivery is unconditional.)

9. Conclusion

To the best of our knowledge, this paper presents the first SSBFT algorithms for IRC and repeated BRB (that follows Definition 1.1) for hybrid asynchronous/time-free systems. As in BT, the SSBFT BRB algorithm takes several communication rounds of $\mathcal{O}(n^2)$ messages per instance whereas the IRC algorithm takes $\mathcal{O}(n)$ messages but requires synchrony assumptions. We hope that the proposed solutions, *e.g.,* the proposed recycling mechanism and the hybrid composition of asynchronous/time-free system settings, will facilitate new SSBFT building blocks.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We are grateful for the comments made by anonymous reviewers that helped to improve the presentation of this article. This work was partially support by the Vinnova/FFI project AutoSec (2019-05883).

References

[1] M. Raynal, Fault-Tolerant Message-Passing Distributed Systems - an Algorithmic Approach, Springer, 2018.
[2] A. Auvolat, M. Raynal, F. Taïani, Byzantine-tolerant set-constrained delivery broadcast, in: OPODIS, in: LIPIcs, vol. 153, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 6.
[3] L. Lamport, R.E. Shostak, M.C. Pease, The Byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (3) (1982) 382–401.
[4] G. Bracha, S. Toueg, Resilient consensus protocols, in: PODC, ACM, 1983, pp. 12–26.
[5] G. Bracha, S. Toueg, Asynchronous consensus and broadcast protocols, J. ACM 32 (4) (1985) 824–840.
[6] S. Dolev, Self-Stabilization, MIT Press, 2000.
[7] M.C. Pease, R.E. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234.
[8] S. Dolev, A. Hanemann, E.M. Schiller, S. Sharma, Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks - (extended abstract), in: SSS, in: LNCS, vol. 7596, Springer, 2012, pp. 133–147.
[9] A. Doudou, B. Garbinato, R. Guerraoui, A. Schiper, Muteness failure detectors: specification and implementation, in: EDCC, in: LNCS, vol. 1667, Springer, 1999, pp. 71–87.

[10] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[11] K. Altisen, S. Devismes, S. Dubois, F. Petit, Introduction to Distributed Self-Stabilizing Algorithms, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2019.
[12] A. Arora, M.G. Gouda, Closure and convergence: a formulation of fault-tolerant computing, in: FTCS, IEEE Computer Society, 1992, pp. 396–403.
[13] S. Bonomi, J. Decouchant, G. Farina, V. Rahli, S. Tixeuil, Practical Byzantine reliable broadcast on partially connected networks, in: ICDCS, IEEE, 2021, pp. 506–516.
[14] R. Guerraoui, J. Komatovic, P. Kuznetsov, Y. Pignolet, D. Seredinschi, A. Tonkikh, Dynamic Byzantine reliable broadcast, in: OPODIS, in: LIPIcs, vol. 184, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 23.
[15] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing uniform reliable broadcast, in: NETYS, in: LNCS, vol. 12129, Springer, 2020, pp. 296–313.
[16] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing indulgent zero-degrading binary consensus, in: Distributed Computing and Networking, ICDCN'21, 2021, pp. 106–115.
[17] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing multivalued consensus in asynchronous crash-prone systems, in: 2021 17th European Dependable Computing Conference (EDCC), IEEE, 2021, pp. 111–118.
[18] C. Georgiou, O. Lundström, E.M. Schiller, Self-stabilizing snapshot objects for asynchronous failure-prone networked systems, in: CoRR, 2019.
[19] A. Mostéfaoui, M. Raynal, Intrusion-tolerant broadcast and agreement abstractions in the presence of Byzantine processes, IEEE Trans. Parallel Distrib. Syst. 27 (4) (2016) 1085–1098.
[20] S. Toueg, Randomized Byzantine agreements, in: Proceedings of the Third Annual ACM Principles of Distributed Computing, Vancouver, B.C., Canada, August 27–29, 1984, 1984, pp. 163–178.
[21] A. Maurer, S. Tixeuil, Self-stabilizing Byzantine broadcast, in: 33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6–9, 2014, 2014, pp. 152–160.
[22] S. Dolev, O. Liba, E.M. Schiller, Self-stabilizing Byzantine resilient topology discovery and message delivery, in: Networked Systems NETYS'13, 2013, pp. 42–57.
[23] S. Bonomi, A.D. Pozzo, M. Potop-Butucaru, S. Tixeuil, Optimal self-stabilizing mobile Byzantine-tolerant regular register with bounded timestamps, in: Stabilization, Safety, and Security of Distributed Systems, SSS'18, 2018, pp. 398–403.
[24] S. Bonomi, M. Potop-Butucaru, S. Tixeuil, Stabilizing Byzantine-fault tolerant storage, in: IEEE Parallel and Distributed Processing Symposium, IPDPS'15, 2015, pp. 894–903.
[25] S. Bonomi, S. Dolev, M. Potop-Butucaru, M. Raynal, Stabilizing server-based storage in Byzantine asynchronous message-passing systems, in: ACM Principles of Distributed Computing, PODC'15, 2015, pp. 471–479.
[26] S. Dolev, J.L. Welch, Self-stabilizing clock synchronization in the presence of Byzantine faults, in: ACM Principles of Distributed Computing PODC'95, 1995, p. 256.
[27] C. Lenzen, J. Rybicki, Self-stabilising Byzantine clock synchronisation is almost as easy as consensus, J. ACM 66 (5) (2019) 32.
[28] S. Bonomi, A.D. Pozzo, M. Potop-Butucaru, S. Tixeuil, Approximate agreement under mobile Byzantine faults, Theor. Comput. Sci. 758 (2019) 17–29.
[29] S. Dubois, M. Potop-Butucaru, M. Nesterenko, S. Tixeuil, Self-stabilizing Byzantine asynchronous unison, J. Parallel Distrib. Comput. 72 (7) (2012) 917–923.
[30] A. Maurer, Self-stabilizing Byzantine-resilient communication in dynamic networks, in: OPODIS, in: LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, vol. 184, 2020, 27.
[31] A. Binun, T. Coupaye, S. Dolev, M. Kassi-Lahlou, M. Lacoste, A. Palesandro, R. Yagel, L. Yankulin, Self-stabilizing Byzantine-tolerant distributed replicated state machine, in: Stabilization, Safety, and Security of Distributed Systems SSS'16, 2016, pp. 36–53.
[32] S. Dolev, C. Georgiou, I. Marcoullis, E.M. Schiller, Self-stabilizing Byzantine tolerant replicated state machine based on failure detectors, in: Cyber Security Cryptography and Machine Learning - Second International Symposium CSCML'18, 2018, pp. 84–100.
[33] R. Duvignau, M. Raynal, E.M. Schiller, Self-stabilizing Byzantine- and intrusion-tolerant consensus, CoRR, arXiv:2110.08592 [abs], 2021, arXiv:2110.08592, https://arxiv.org/abs/2110.08592.
[34] C. Georgiou, I. Marcoullis, M. Raynal, E.M. Schiller, Loosely-self-stabilizing Byzantine-tolerant binary consensus for signature-free message-passing systems, in: Networked Systems - 9th International Conference, NETYS, in: LNCS, vol. 12754, Springer, 2021, pp. 36–53.
[35] A. Mostéfaoui, M. Raynal, Signature-free broadcast-based intrusion tolerance: never decide a Byzantine value, in: Principles of Distributed Systems, OPODIS'10, 2010, pp. 143–158.
[36] S. Dolev, T. Petig, E.M. Schiller, Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks, Algorithmica 85 (1) (2023) 216–276.
[37] P. Blanchard, S. Dolev, J. Beauquier, S. Delaët, Practically self-stabilizing Paxos replicated state-machine, in: NETYS, in: LNCS, vol. 8593, Springer, 2014, pp. 99–121.
[38] M.J. Fischer, N.A. Lynch, M. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382.