Self-Stabilizing Reconfiguration[⋆]

Shlomi Dolev[1][⋆⋆], Chryssis Georgiou[2], Ioannis Marcoullis[2][⋆⋆⋆], and Elad M. Schiller[3]

[1] Dept. of Computer Science, Ben-Gurion University of the Negev, Israel.
[2] Dept. of Computer Science, University of Cyprus, Cyprus.
[3] Dept. of Engineering and Computer Science, Chalmers University of Technology, Sweden.

Abstract. Current reconfiguration techniques depend on starting the system in a consistent configuration, in which all participating entities are in a predefined state. Starting from that state, the system must preserve consistency as long as a predefined churn rate of processors joins and leaves is not violated, and unbounded storage is available. Many systems cannot control this churn rate and lack access to unbounded storage. System designers that neglect the outcome of violating the above assumptions may doom the system to exhibit illegal behaviors. We present the first automatically recovering reconfiguration scheme that recovers from transient faults, such as temporal violations of the above assumptions. Our self-stabilizing solutions regain safety automatically by assuming temporal access to reliable failure detectors (FDs). Once safety is established, the FD reliability is no longer needed. Still, liveness is conditioned by the FD's unreliable signals. Our self-stabilizing reconfiguration techniques can serve as the basis for the implementation of several dynamic services over message passing systems. Examples include self-stabilizing reconfigurable virtual synchrony, extendable to a self-stabilizing reconfigurable state machine replication.

1 Introduction

Motivation. We consider distributed systems working in dynamic asynchronous environments, such as a shared storage system [17]. Quorum configurations [19], i.e., sets of active processors (servers or replicas), are typically used to provide service to the system's participants. A configuration may gradually lose active participants due to voluntary leaves or stop failures, rendering service provision poor or impossible. It is important to instate a new configuration, i.e., to *reconfigure*, on time, based on a more recent participation set. In recent years, several reconfiguration techniques were proposed, mainly for state machine replication and atomic memory emulation (e.g., [1,2,3,4,13,14,15,16,18]). Such reconfiguration techniques depend on initiating the system in a consistent configuration, with all processors in a predefined state. Continuing from this state, the system must preserve consistency assuming a predefined

[⋆] A full version of this paper can be found in [8].

[⋆⋆] The research was partially supported by the Rita Altura Trust Chair in Computer Sciences; Frankel center for computer science, grant of the Ministry of Science, Technology and Space, Israel, and the National Science Council (NSC) of Taiwan; the Ministry of Foreign Affairs, Italy; the Ministry of Science, Technology and Space, Infrastructure Research in the Field of Advanced Computing and Cyber Security and the Israel National Cyber Bureau.

[⋆⋆⋆] Partially supported by a Doctoral Scholarship program of the University of Cyprus.

churn rate is not violated and unbounded storage availability. Also, they do not claim to tolerate *transient faults* that may arbitrarily alter the system's variables

Many working systems cannot control their churn rate and do not have access to unbounded storage. System designers that neglect the outcome of violating the above assumptions may doom the system to forever exhibit a behavior that does not satisfy the system requirements. Furthermore, the dynamic and difficult-to-predict nature of distributed systems gives rise to many fault-tolerance issues and requires efficient solutions. Large-scale message passing networks are asynchronous and they are subject to transient faults due to hardware or software temporal malfunctions, short-lived violations of the assumed failure rates or violation of correctness invariants, such as the uniform agreement among all current participants about the current configuration. Fault tolerant systems that are *self-stabilizing* [6] can recover after the occurrence of transient faults as long as the program's code is still intact.

Contributions and approach. We present the first automatically recovering reconfiguration scheme that recovers from transient faults, such as temporal violations of the predefined churn rate or the unexpected activities of processors and communication channels. Our blueprint for self-stabilizing reconfigurable distributed systems can withstand a temporal violation of such assumptions, and recover once conditions are resumed, using a bounded amount of local storage and message size. Our self-stabilizing solutions regain safety automatically by assuming temporal access to reliable failure detectors* (FDs). Once safety is re-established, the FDs' reliability is no longer needed; liveness is conditioned by the FDs' unreliable signals. We now overview our approach.

Reconfiguration scheme. Our scheme comprises of two layers that appear as a single "black-box" module to any application that uses the reconfiguration service. The objective is to provide the application with a *conflict-free* configuration, such that no two alive processors consider different configurations.

The first layer, called *Reconfiguration Stability Assurance* (*recSA*) and detailed in Section 3.1, is mainly responsible for detecting configuration conflicts (possibly the result of transient faults). It deploys a *brute-force* technique for converging to a conflict-free new configuration. It also employs a *delicate* configuration replacement technique when a processor notifies that it wishes to replace the current configuration with a new set of participants. For both techniques, processors use an implementable FD (cf. Section 2) to obtain membership information. Configuration convergence is reached when the FDs have temporal reliability. Once a uniform configuration is installed, the FDs' reliability is no longer needed. Liveness conditions thereafter consider unreliable FDs.

The decision for requesting a delicate reconfiguration is controlled by the other layer, called *Reconfiguration Management* or *recMA* for short (detailed in Section 3.2). Specifically, if a processor suspects that the dependability of the current configuration is under jeopardy, it seeks to obtain a majority approval from the alive *members* of the current configuration, and requests a (delicate) reconfiguration from *recSA*. Moreover, in the absence of such a majority (e.g., configuration replacement was not activated "on time" or the churn assumptions were violated), the *recMA* can aim to control the recovery via a *recSA* reconfiguration request. The current participant set can, over time,

* Transient faults pose challenges in managing dynamic membership that justify the use of FDs; see discussion in Related work.

become different than the configuration member set. As new members arrive and others go, changing the configuration based on system membership would imply a high frequency of (delicate) reconfigurations, especially in the presence of high churn. Note that we avoid unnecessary reconfiguration requests by requiring a weak liveness condition: if a majority of the configuration set has not collapsed, then there exists at least one processor that is known to trust this majority in the FD of each alive processor. Such active configuration members can aim to replace the current configuration with a newer one (that would provide an approving majority for prospective reconfigurations) without the use of the brute-force stabilization technique.

Joining mechanism. We complement our reconfiguration scheme with a self-stabilizing joining mechanism *JoinMec* (detailed in Section 3.3) that manages and controls the introduction of new processors into the system. It is crucial to ensure that newly joining processors do not carry stale information (due to arbitrary faults) into the system state. To this end, we employ several techniques along with a snap-stabilizing data link protocol (see Section 2). We have designed *JoinMec* to grant the application the control on whether to allow new processors to join the system or not. In this way, the churn (regarding new arrivals) can be "fine-tuned" based on the application requirements; we have modeled this by having joining processors obtaining approval from a majority of the current configuration's members given no reconfiguration is taking place. These, in turn, provide such approval if the application's (among other) criteria are met. We note that in the event of transient faults, such as unavailable approving majority, *recSA* assures recovery via brute-force stabilization that includes *all* alive processors.

Applications. The presented reconfiguration scheme is modular and can be used to extend the capabilities of algorithms designed for more static environments, i.e., for environments where processors are aware of a single set of processors that can fail by crashing. The reconfiguration scheme allows for this set to be renewed and thus service can continue. We have used our reconfiguration scheme to obtain dynamic versions of a multipurpose counter increment algorithm and a self-stabilizing virtual synchrony algorithm that also leads to a self-stabilizing replicated state machine (cf., Section 4).

Related work. Existing solutions for providing reconfiguration in dynamic systems, such as [14] and [1], do not consider transient faults and self-stabilization, as their correctness proofs (implicitly) depend on a coherent start [17] and also assume that fail-stops can never prevent the (quorum) configuration to facilitate configuration updates. They also often use unbounded counters for ordering consensus messages (or for shared memory emulation) and by that facilitate configuration updates, e.g., [14]. Our self-stabilizing solution recovers after the occurrence of transient faults, which we model as an arbitrary starting state, and guarantees a consistent configuration that provides (quorum) services, e.g., allowing reading from and writing to distributed shared memory, and at the same time managing the configuration that provides these services.

Furthermore, in existing non self-stabilizing solutions, dynamic membership is usually maintained by the exchange of "membership sets" (e.g., the set *World* in [14]). But when dealing with transient faults, it is possible that local membership sets may change arbitrarily and result in containing a large number of identifiers of processors that are not present in the system. Given the asynchronous environment, this would result in a deadlock if the processors wait for some majority (or quorum) of these non-existing processors to respond while they have no means for detecting their non-existence. To this respect, our self-stabilizing solution makes use of FDs (cf. Section 2).

There exists a significant amount of research to characterize the fault-tolerance guarantees that different quorum system designs can provided; see [19] for an in depth discussion. In this paper we use majorities, generally regarded as the simplest quorum system (each set composed of a majority of the processors is a quorum). One can modify our reconfiguration scheme to support more complex, quorum systems, as long as processors have access to a mechanism (that is a function) that, given a set of processors, can generate the specific quorum system. The *when* a reconfiguration (delicate in our case) should take place is another important design decision; see related discussion in [17]. A simple approach is to reconfigure when a fraction (e.g., 1/4th) of the members of a configuration appear to have failed. More complex decisions use prediction mechanisms (possibly based on statistics). This issue is beyond the scope of this work; however, we have designed our reconfiguration scheme (specifically the *recMA* layer) to use any decision mechanism imposed by the application (via an application interface).

2 System Settings

Processing entities. We consider an asynchronous message-passing system of processors. Each processor p_i has a unique identifier, i, taken from a totally-ordered set of identifiers P. The number of live and connected processors at any time of the computation is bounded by some integer N such that $N \ll |P|$. We refer to such processors as *active*. We assume that processors have knowledge of the upper bound N, but not of the actual number of active processors. Processors may stop-fail by crashing at any point without warning. A crashed processor takes no further steps and never rejoins the computation. (For readability's sake, we model rejoins as transient faults rather than considering them explicitly. Self-stabilization inherently deals with rejoins by regarding the past join information as possibly corrupted.) New processors may join the system (using a joining procedure) at any point in time with an identifier drawn from P, such that this identifier is only used by this processor forever. A *participant* is an active processor that has joined the computation and sends configuration-related messages. Note that N accounts for all active processors, both the participants and those still joining.
Communication. The network topology is that of a fully connected graph, and links have a bounded capacity cap. Processors exchange low-level messages called *packets* to enable a reliable delivery of high level *messages*. Packets sent may be lost, reordered, or duplicated but not arbitrarily created, although the channels may initially (after transient faults) contain stale packets, which due to the boundedness of the channels are also bounded in a number that is in $O(N^2 cap)$. We assume the availability of self-stabilizing protocols for reliable FIFO end-to-end message delivery (over unreliable channels with bounded capacity), e.g., [9]and that channels provide *fair communication*, i.e., a packet sent infinitely often is received infinitely often.

Using the underlying packet exchange protocol described, a processor p_i that has received a packet from some processor p_j which did not belong to p_i's FD, engages in a two phase protocol with p_j in order to clean their intermediate link. This is done before any messages are delivered to the algorithms that handle reconfiguration, joining and applications. We follow the snap-stabilizing data link protocol detailed in [12]. A *snap-stabilizing* protocol is one which allows the system (after faults cease) to behave according to its specification upon its first invocation. We require that every data-link established between two processors is initialized and cleaned straight after it is established. In contrast to [12] where the protocol runs on a tree and initiates from the root, our case requires that each pair of processors takes the responsibility of cleaning their intermediate link. Snap-stabilizing data links do not ignore signals indicating the existence of new connections (such as physical carrier signal from the port). In fact, when such a connection signal is received by the newly connected parties, they start a communication procedure that uses the bound on the packet in transit (possibly in buffers too) to clean all unknown packets in transit, possibly repeatedly sending the same packet until more than the round trip capacity acknowledgments arrive.

(N, Θ)**-failure detector.** It extends the Θ-FD used in [5]. It allows each processor p_i to order other processors according to how recently they have communicated. To achieve this, p_i maintains an ordered vector $nonCrashed$ where every other communicating processor p_k is ranked according to the message exchanges that it has performed with p_i and relative to the communication it has with some other processor p_j. Specifically, when p_i receives a message from p_j, it sets p_j's corresponding counter to 0, and increments the counters of any other processor p_k by one. Since there are at most N processors in the computation at any given time, we can ignore any processors that rank below the N^{th} vector entry. Each processor p_i uses the $nonCrashed$ vector to get an estimate on the number of processors n_i that p_i believes to be active in the system; $n_i \leq N$. Processor p_i will find that between the last processor that is still communicating with, and the first processor that has not communicated for some time, there is a significant difference in their counter. Thus, the last processor is the ${n_i}^{th}$ processor and provides an estimate on the number of active processors. If, for example, the first 30 processors in the vector have corresponding counters of up to 30, then the 31^{st} will have a counter much greater than that, say 100; so n_i will be estimated at 30. This estimation mechanism is suggested in [10] and in [11]. (For implementation details see [8].)

The interleaving model and self-stabilization. A program is a sequence of *(atomic) steps*. Each atomic step starts with local computations and ends with a communication operation, i.e., packet $send$ or $receive$. We assume the standard interleaving model where at most one step is executed in every given moment. An input event can either be the arrival of a packet or a periodic timer triggering p_i to (re)send. Note that the system is asynchronous and the rate of the timer is totally unknown. The *state*, c_i, consists of p_i's variable values and the content of p_i's incoming communication channels. A step executed by p_i can change p_i's state. The tuple of the form $(c_1, c_2, \cdots, c_n)$ is used to denote the *system state*. An *execution (or run)* $R = c_0, a_0, c_1, a_1, \ldots$ is an alternating sequence of system states c_x and steps a_x, such that each c_{x+1}, except the initial system state c_0, is obtained from c_x by the execution of a_x. A practically infinite execution is an execution with many steps, where many is defined to be proportional to the time it takes to execute a step and the life-span time of a system. The system's task is a set of executions called *legal executions* (LE) in which the task's requirements hold. An algorithm is *self-stabilizing* with respect to LE when every (unbounded) execution of the algorithm has a suffix that is in LE.

3 Self-stabilizing Reconfiguration Scheme

We now describe the reconfiguration scheme and joining mechanism. Figure 1 depicts the interaction between the modules and the application. The Reconfiguration Stability Assurance ($recSA$) layer ensures that participants eventually have a common configuration. It provides information on the current configuration and on whether a reconfiguration is not taking place using interfaces $getConfig()$ and $noReco()$ respectively. This is based on local information. The Reconfiguration Management ($recMA$) layer uses the (application-based) prediction mechanism $evalConf()$ to evaluate if a reconfiguration is required. If a reconfiguration is necessary, $recMA$ initiates it with $estab()$. Joining only proceeds if no reconfiguration is taking place. A joiner becomes a participant via $participate()$ only if $passQuery()$ of a majority of configuration members is reported as True. Arrows directed from module A to B show the transfer of specified information from A to B. We proceed with details.

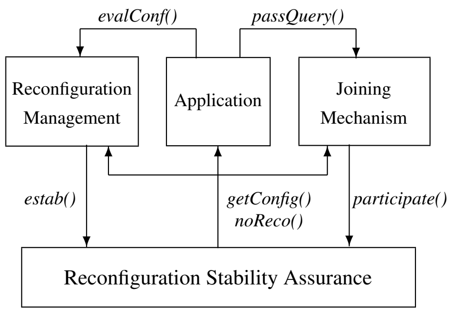

Fig. 1. Module Interaction.

3.1 The Reconfiguration Stability Assurance Layer

This layer uses Algorithm 1 for assuring correct configuration while allowing updates from the $recMA$ layer (next section). Algorithm 1 guarantees that (1) all active processors have eventually identical copies of a single configuration, (2) when participants notify the system that they wish to replace the current configuration with another, the algorithm selects one proposal and replaces the current configuration with it, and (3) joining processors can become participants eventually.

The algorithm structure. The algorithm combines two techniques: one for *brute force stabilization* that recovers from stale information and a complementary technique for *delicate (configuration) replacement*, where participants jointly select a single new configuration that replaces the current one.

Combining the two techniques. As long as a given processor is not aware of ongoing configuration replacements, Algorithm 1 merely monitors the system for stale information, e.g., that the config fields hold the same non-$\perp$ value. During these periods the algorithm allows the invocation of configuration replacement processes (via the $estab(\text{set})$ interface) as well as the acceptance of joining processors as participants (via the $participate()$ interface). During the process of configuration replacement, the algorithm selects a single configuration proposal and replaces the current one with that proposal before returning to monitor for configuration disagreements.

Blocking joins to the participants' set during reconfiguration periods. While the system reconfigures, there is no immediate need to allow joining processors to become participants. By temporarily disabling this functionality, the algorithm can focus on completing the configuration replacement using the current participant set. To that end,

Algorithm 1: Stabilizing Reconfiguration Stability Assurance; p_i's code

```
1  Variables: Each field is held in an array that stores $p_i$'s own values and $p_j$'s most recently received ones. For
   example, in the case of the config[] field, config$[i]$ is $p_i$'s view on the current configuration and config$[j]$ stores the
   most recently received one. Note that $p_i$ assigns $\bot$ (the empty configuration) after receiving a conflicting (different)
   non-empty configuration value. FD$[i]$ and FD$[i].part$ represent $p_i$'s failure detector, and respectively, an alias to
   $\{p_j \in$ FD$[i]$ : config$[j] \neq \sharp\}$. Note that we consider only the trusted (unsuspected) processors. Namely, crashed
   processors are eventually suspected and the FD field of every message encodes also this participation info. The field
   prp$[i] = \langle phase \in \{0,1,2\}, set \subseteq P\rangle$, where prp$[i]$ refers to $p_i$'s configuration replacement proposal. The
   case of no proposal is denoted by $\langle 0, \bot\rangle$. The field all$[i]$ is true when $p_i$ observes that all trusted nodes notice its
   current (max) proposal and they hold the same value. The variable allSeen stores the set of nodes $p_k$ for which $p_i$
   received the all $= true$ indication.
2  Interfaces: function $participate()$ replaces $p_i$'s configuration (possibly set to $\sharp$) with $chsConfig()$. Only
   allowed when no reconfiguration is taking place.
3  function $chsConfig()$ is the current config value, or $\bot$ when there is no single non-$\sharp$ value.
4  function $getConfig()$ {if $noReco()$ then return($chsConfig()$) else return(config$[i]$)};
5  function $noReco()$ test (locally) whether $p_i$ runs a reconfiguration process.
6  function $estab$(set) $=$ {if ($noReco() \wedge (set \notin \{$config$[i], \emptyset\})$) then prp$[i] \leftarrow \langle 1,$ set$\rangle$};
7  do forever begin
8      if stale info present, e.g., different (non-$\bot$ or-$\sharp$) config values or empty intersection between config and
         participant set then reset, i.e., call $configSet(\bot)$;
9      if there is no proposal for configuration replacement then
10         if $|\{$config$[k]\}_{p_k \in \text{FD}[i]} \setminus \{\bot, \sharp\}| > 1$ then $configSet(\bot)$ // once a trusted processor has sent a
             different (non-$\bot$ or $\sharp$) configuration, $\bot$-nullify the stored one – i.e., nullify the configuration upon
             conflict;
11         if (config$[i] = \bot \wedge |\{$FD$[j] : p_j \in$ FD$[i]\}| = 1$) then $configSet($FD$[i])$ // once all trusted
             nodes trust the same nodes, use this node set as the new configuration;
12     else
13         if all trusted participants report the same proposals and participation sets and they echo back the sent
             values of these fields then all$[i] \leftarrow true$;
14         else if trusted participant $p_k$ reports all$[i] = true$ then
15             add $p_k$ to allSeen;
16             if allSeen includes all trusted participants then run the automaton (Figure 2) and empty
                 allSeen $\leftarrow \emptyset$;
17     if config$[i] \neq \sharp$ then send to $p_j$ the state of $p_i$ (including $p_j$'s recently received info.);
18 upon receive $m$ from $p_j$ do store $m$'s fields as the recently received values from $p_j$;
19 upon interrupt $p_i$'s booting do foreach $p_k$ do (config$[k]$, prp$[k]$, all$[k]$) $\leftarrow (\sharp, \langle 0, \bot\rangle, false)$ // during
   boot, nullify the stored fields and disable message transmissions;
```

only participants broadcast their state when finishing the do forever loop (line 17) and only their messages arrive to the other active processors (line 18). Moreover, we assume that the only way for a joining processor to start executing Algorithm 1 is by responding to an interrupt call (line 19), where the assignment of $\sharp$ to config nullifies the configuration. Thus, joining processors cannot broadcast (line 17) before their safe entry to participant set via the function $participate()$ (line 2), which enables broadcasting. Note that non-participants monitor the intersection between the current configuration and the set of active participants (line 8). In case it is empty, the processors (participants or not) call $configSet(\bot)$ and start a *reset process* that ends with a brute-force stabilization, which we explain below. Thus, the $\sharp$ values are removed from config and there is no more blocking of joining processors to become participants.

Brute-force stabilization. The proposed algorithm detects the presence of stale information and recovers from these transient faults. *Configuration conflicts* are one of several kinds of such stale information and they refer to differences in the field config, which stores the configuration values. Processor p_i can signal to all processors that it has detected stale information by assigning $\bot$ to config$_i$ and by that starts a *reset process* that nullifies all config fields in the system (lines 8 and 10). Algorithm 1 uses the brute-force technique for letting processor p_i to assign to config$_i$ its set of trusted processors (line 11), which the failure detector FD_i provides. Note that brute-force stabilization removes any $\sharp$ value from config and allows all processors (joining or participants) to become a participant at the end of the brute-force process. Theorem 1 together with Lemma 2 show that eventually all active processors share identical (non-$\bot$) config values.

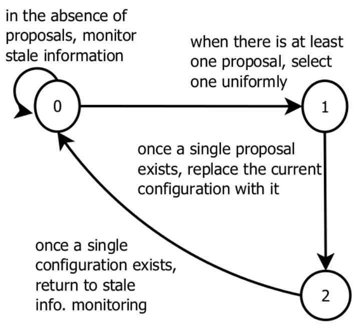
in the absence of proposals, monitor stale information
when there is at least one proposal, select one uniformly
once a single proposal exists, replace the current configuration with it
once a single configuration exists, return to stale info. monitoring
0
1
2

Fig. 2. The automaton

Delicate (configuration) replacement. Participants can propose to replace the current configuration with a new set, via the $estab$(set) interface. This replacement uses the *configuration replacement* automaton (Figure 2) that a self-stabilizing mechanism for *(phase transition) coordination* emulates.

The configuration replacement automaton. When the system is free from stale information, the configuration uniformity invariant (of the config field values) holds. Then, any number of calls to the $estab$(set) interface starts the automaton, which controls the configuration replacement using the following three phases: (1) selecting uniformly a single proposal (while verifying the eventual absence of "unselected" proposals), (2) replacing uniformly all config fields with the jointly selected proposal, and (3) bringing the system back to a state where it merely tests for stale information.

A self-stabilizing mechanism for phase transition coordination. The configuration replacement automaton, requires coordinated phase transition. Algorithm 1 lets processor p_i represent proposals as $\mathrm{prp}_i[j] = (phase, set)$, where p_j is the processor from which p_i received the proposal, $phase \in \{0, 1, 2\}$ and set is a processor set or the null value, $\bot$. The *default proposal*, $\langle 0, \bot\rangle$, refers to the case in which prp encodes "no proposal" (line 1). When p_i calls the function $estab$(set), it changes prp to $\langle 1, set\rangle$ (line 6) as long as p_i is not aware of an ongoing configuration replacement process, i.e., $noReco()$ returns $true$. Upon this change, the algorithm disseminates $\mathrm{prp}_i[i]$ and by that guarantees that eventually $noReco()$ returns $false$ for any processor that calls it. Once this happens, no call to $estab$(set) adds a new proposal for configuration replacement and no call to $participate()$ lets a joining processor to become a participant (line 2). Algorithm 1 can then use the lexical value of the $\mathrm{prp}_i[]$'s tuples to deterministically select one of them (Figure 2). To that end, each participant ensures that all other participants report the same tuples by waiting until they "echo" back the same values as the ones it had sent them. After this, participant p_i makes sure that the communication channels do not include other "unselected" proposals, by raising a flag $\mathrm{all}_i = true$ (line 13) and waiting for the echoed values of these three fields, i.e., participant set, $\mathrm{prp}_i[i]$ and all_i. This waiting continues until the echoed values match the values of any other active participant in the system (while monitoring their well-being). Before this participant proceeds, it makes sure that all active participants have noticed its phase completion (line 15). Each processor maintains the allSeen variable; a set of participants that have noticed its phase completion and has thus added them to the set allSeen.

The above self-stabilizing mechanism for phase transition coordination allows progression in a unison fashion. Namely, no processor starts a new phase before it has seen that all other active participants have completed the current phase and have noticed that all others have done so (because they have identical participant set, prp and all values). This is the basis for emulating every step of the configuration replacement automaton (line 16) and making sure that the phase 2 replacement occurs correctly before returning to phase 0, in which the system simply tests for stale information. Since the FDs monitor the participants' well-being, this process terminates.

Correctness. We here highlight the main steps of the proof, starting with some key definitions. An execution R is *admissible* when throughout R the FD values of active processors are identical, do not change and consist of only themselves (the set of active processors). I.e., $\forall c \in R$, $p_i, p_j \in P$ that are active in R, we have $\text{FD}_i[i] = \text{FD}_j[j]$ and $p_k \in \text{FD}_i[i] \iff p_k$ is active. Furthermore, we say that system state c has *no stale information* when (1) all (quorum) configuration proposals are valid, e.g., the proposal $\langle 0, set \rangle$ is not valid when $set \neq \bot$, (2) all config values are non-$\bot$ and the same, (3) the phase information (including $allSeen$) is in synch, and (4) the config set includes active participants. The correctness proof shows that eventually there is no stale information (Theorem 1), because they are all detected and cleaned eventually (lines 8 and 10), as processors run configuration *reset processes* (by calling $configSet(\bot)$). To guarantee the success of such reset processes (Lemma 2), we assume that the system reaches eventually an admissible execution until the reset process terminates.
Failure Detector Usage: The above assumption implies that Algorithm 1 completes the reset process by having a temporal access to reliable FDs. However, once it completes this process, safety holds forever thereafter because, as Theorem 1 shows, the system cannot introduce stale information (or start another reset process) after the process terminates. In other words, once the reset process establishes safety, the FD reliability is no longer needed, because the success of Algorithm 1 to achieve its task does not require that the system reaches admissible executions, and liveness is conditioned by the FD's unreliable signals. Since Theorem 1 shows that no stale information eventually exists, all the processors p_i for which the field $\text{config}_i[i] \notin \{\bot, \sharp\}$ store the same value in that field. We now give the main result and a proof sketch. (For the full proof see [8]).

Theorem 1 (Convergence). *Let R be an admissible execution of Algorithm 1. R has no stale information eventually.*

Proof Sketch. Lines 8 and 10 detect stale information and start the configuration reset, which by Lemma 2 terminates. The proof uses Claim 5 and Lemma 6 to imply the theorem's correctness, the first assuming that R does not include (notifications about) replacement proposals, and the second that it does.

Lemma 2 *During admissible executions R, reset processes terminate, eventually leading to no configuration conflicts.*

Proof Sketch. Suppose that R's starting system state does include a detection (line 8), does not include a conflict, i.e., $\exists p_i, p_j \in P : (\text{config}_i[i] = \bot) \vee (\text{config}_i[i] \neq \text{config}_i[j]) \vee (\text{config}_i[i] \neq \text{config}_j[j])$ or there is a message, $m_{i,j}$, in the communication channel from p_i to p_j, such that the field $(m_{i,j}.\text{config}[k] = \bot) : p_k \in \text{FD}_i[i] \vee (m_{i,j}.\text{config} \neq \text{config}_i[i])$, where both p_i and p_j are active processors. We use Claims 3 and 4 to show that in all of these cases, eventually $\forall p_i \in P : \text{config}_i[i] \in$

$\{\bot, \text{FD}_i[i]\}$ before using Claim 5 to show that eventually there are no configuration conflicts. Claims 3 and 4 consider the values in the field config that are either held by an active processor $p_i \in P$ or in its outgoing communication channel to another active processor $p_j \in P$. We define the set $S = \{S_i \cup S\_out_i\}_{p_i \in P}$ to be the set of all these values, where $S_i = \{\text{config}_i[j]\}_{p_j \in \text{FD}_i[i]}$ and $S\_out_i = \{m_{i,j}.\text{config}\}_{p_j \in \text{FD}_i[i]}$.

Claim 3 *Suppose that in R's starting system state, there are processors $p_i, p_j \in P$ that are active in R, for which $|S \setminus \{\bot, \sharp\}| > 1$. (1) $\exists S' \subseteq S : S' \in \{\{\text{config}_i[i], \text{config}_i[j]\}, \{\text{config}_i[i], m_{i,j}.\text{config}\}\}$ implies that eventually the system reaches a state in which $\text{config}_i[i] \in \{\bot, \text{FD}_i[i]\}$ holds. (2) $\exists S' \subseteq S : S' \in \{\{\text{config}_i[i], \text{config}_j[j]\}\}$ implies that eventually the system reaches a state in which $\text{config}_i[i] \in \{\bot, \text{FD}_i[i]\}$ or $\text{config}_j[j] \in \{\bot, \text{FD}_i[i]\}$ holds.*

Claim 4 *Suppose that $\text{config}_i[i] \in \{\bot, \text{FD}_i[i]\} : p_i \in P$ in R's starting system state. (1) For any system state $c \in R : \text{config}_i[i] \in \{\bot, \text{FD}_i[i]\}$, and (2) $R = R' \circ R''$ has a suffix, R'', such that $\forall c'' \in R'', \forall p_i, p_j$ that are active in $R : (\{m_{i,j}.\text{config}, \ \text{config}_j[i], \text{config}_j[j]\} \setminus \{\bot, \text{FD}_i[i]\}) = \emptyset$.*

Claim 5 *Suppose for any two active $p_i, p_j \in P$, we have that $(\{\text{config}_i[i], \text{config}_j[i], m_{i,j}.\text{config}\} \setminus \{\bot, \text{FD}_i[i]\}) = \emptyset$. Eventually $\text{config}_i[i] = \text{FD}_i[i]$.*

Lemma 6 *Let R be an admissible execution (wrt the participant sets) of Algorithm 1. Let n be a configuration replacement notification in R. Eventually n leaves the system.*

Proof Sketch. We assume, towards a contradiction, that notification n never leaves the system and it has a maximal lexical value among all the notifications in R. We begin by assuming that all of R's notifications appear in its starting state before removing this assumption. We use the fact that only lines 15 to 16 change the notifications and by that we show the non-decrease property of their lexical values. A contradiction is achieved by showing that the following invariants hold. Suppose that $\text{prp}_i[i] = \text{n}$ holds in every system state $c' \in R$. Eventually the system reaches a state $c'' \in R$, such that for any $p_j \in P$ that is an active participant in R, it holds that: (1) $\text{prp}_j[i] = \text{n}$ and $\text{FD}_j[i] = \text{FD}_i$. Moreover, $\text{prp}_j[j] = \text{n}$ and $\text{FD}_j[j] = \text{FD}_i$ in c'' eventually, (2) $\text{echo}_i[j].\text{prp} = \text{n}$, $\text{echo}_i[j].\text{part} = \text{FD}_i[i].part$ and $\text{prp}_i[j] = \text{n}$ in c'', (3) $\text{all}_i[i] = true$ in c''. (4) $\text{all}_j[i] = true$ in c''. (5) $\text{echo}_i[j] = (\text{FD}_i[i].part, \text{prp}_i[i], \text{all}_i[i])$ in c''. (6) $p_i \in allSeen_j$ in c''. (7) the if-statement condition of line 16 holds in c''. Note that there exists a system state $c_{\exists \text{n}} \in R$ in which there are no notifications, because of invariant (7) there is a step a_i that immediately follows c'' and in which p_i for any $\text{n}.phase$ value contradicts the assumption that n is of maximal value or that it never leave the system. We complete the proof by showing that even in executions in which not all of R's notifications appear in its starting state, the above eventually holds. To that end, the proof considers all notifications that appeared in R's starting state and shows that they must leave the system eventually because their (continuous) presence causes $noReco()$ to return false and by that disable the effect of the function $estab(\text{set})$ (line 6). Once this is true for every active processor in the system, the conditions for invariants (1) to (7) hold and all notifications leave the system eventually.

Algorithm 2: Self-stabilizing Reconfiguration Management; code for processor p_i

```
1  Interfaces: $evalConf()$ returns True/False on whether a reconfiguration is required or not by based on a
     user-defined prediction function. The rest of the interfaces are specified in Algorithm 1. $noReco()$ returns True
     if a reconfiguration is not taking place, else False. $estab(set)$ initiates the creation of a new configuration based
     on the processor $set$ provided. $getConfig()$ returns the current local configuration.
2  Variables: $needReconf[]$ is an array of size at most $N$, composed of booleans {True, False}, where
     $needReconf_i[j]$ holds the last value of $needReconf_j[j]$ that $p_i$ received from $p_j$ as a result of exchange
     (lines 16 and 17) and $needReconf$ is an alias to $needReconf_i[i]$ i.e., of $p_i$'s last reading of $evalConf()$.
     Similarly, $noMaj_i[]$ is an array of booleans of size at most $N$ on whether some trusted processor of $p_i$ detects a
     majority of members that are active per the reading of line 11. $noMaj_i[j]$ (for $i \neq j$) holds the last value of
     $noMaj_j[j]$ that $p_i$ received from $p_j$. $prevConfig$ holds $p_i$'s believed previous $config$.
3  Macros: $core() = \bigcap_{p_j \in FD_i[i].part} FD[j].part$;
4  $flushFlags()$ : foreach $p_j \in FD[i]$ do $needReconf[j] \leftarrow (noMaj[j] \leftarrow \text{False})$;
5  Do forever begin
6      if $p_i \in FD[i].part$ then
7          $curConf = getConfig()$; $needReconf[i] \leftarrow (noMaj[i] \leftarrow \text{False})$;
8          if $prevConfig \notin \{curConf, \bot\}$ then $flushFlags()$;
9          if $noReco() = \text{True}$ then
10             $prevConfig \leftarrow curConf$;
11             if $|\{p_j \notin curConf \cap FD[i]\}| < (\frac{|curConf|}{2} + 1)$ then $noMaj[i] \leftarrow \text{True}$;
12             if $(noMaj[i] = \text{True}) \wedge (|core()| > 1) \wedge (\forall p_k \in core() : noMaj[k] = \text{True})$ then
13                 $estab(FD[i].part)$; $flushFlags()$;
14             else if $(needReconf[i] \leftarrow evalConf(curConf)) \wedge$
                 $|\{p_j \in curConf \cap FD[i] : needReconf[j] = \text{True}\}| > \frac{|curConf|}{2}$ then
15                 $estab(FD[i].part)$; $flushFlags()$;
16         foreach $p_j \in FD[i].part$ do $send(\langle noMaj[i], needReconf[i] \rangle)$;
17 Upon receive $m$ from $p_j$ do if $p_i \in FD[i].part$ then $\langle noMaj[j], needReconf[j] \rangle \leftarrow m$;
```

3.2 Reconfiguration Management

The Reconfiguration Management ($recMA$) layer (Algorithm 2), bears the weight of initiating (or "triggering") a reconfiguration when either the majority has been lost, or when the prediction function $evalConf()$ indicates to a majority of processors that a reconfiguration is needed to preserve the majority. To achieve this, it uses the $estab()$ interface of Algorithm 1. In spite of using majorities, the algorithm is generalizable to other (more complex) quorum systems, while the prediction function $evalConf()$ (used as a black box) can be either very simple, e.g., asking for reconfiguration once $1/4^{th}$ of the members appear to have failed, or more complex, based on application criteria or network considerations. More elaborate methods may also be used to define the set of processors that Algorithm 2 proposes as the new configuration. Our current implementation, aiming at simplicity of presentation, defines the set of trusted participants of the proposer as the proposed set for the new configuration.

Algorithm description. Each processor executing the algorithm maintains two variables, $noMaj$ and $needReconf$. The first stores True/False on whether p_i's FD considers a majority of the configuration members as alive. $needReconf$ stores the outcome of the last call to the prediction function $evalConf()$. These two variables are sent to all participating processors in every iteration of the algorithm and the received variables are stored for every participating processor. All decisions on whether a reconfiguration should take place or not, is based on the received values for the two flags.

Algorithm 2 persistently refrains from triggering a reconfiguration if one is already taking place, by the check of line 9. If a reconfiguration is not taking place, two cases can force the algorithm to reconfigure.

(i) Processor p_i sees that a majority of members suggests a reconfiguration. If a majority of active configuration members exists and locally they see that $evalConfig() =$ True, each propagates $needReconf =$ True. Any such processor, that locally sees a majority of $needReconf =$ True (lines 14–15), will proceed to propose $FD_i[i]$ as the new configuration (line 15). We note that this will be a delicate reconfiguration.
(ii) Processor p_i sees a loss of majority also seen by p_i's core. If a processor p_i suspects that the majority has collapsed, it propagates $noMaj =$ True. Given that FDs are not required to be always perfect (this is only required by Algorithm 1 to converge to a new configuration), local information may inaccurately at times present a loss of majority. In order to prevent unnecessary reconfigurations, we require that a processor considers a "core" of information from the processors that seem to be regarded active by all the processors. We thus introduce the notion of the local *core* as the intersection of the FDs of participating processors in p_i's FD (line 3). If every processor in p_i's core appears to have $noMaj =$ True based on p_i's local information (collected via the exchange of line 17) then a reconfiguration is triggered by p_i with $FD_i[i]$ as the new configuration (lines 12–13). The core is required to have size greater than 1 to prevent p_i from triggering if it is the only processor of its core. Using the notion of the core, we also place the following *liveness* assumption on the FDs.
Majority-supportive core assumption. If a majority (of the configuration) has not collapsed, then in the core of every participant p_i, there exists at least one processor that is known (by p_i) to trust this majority in its FD.

In triggering a reconfiguration, Algorithm 2 uses the $estab(set)$ interface with Algorithm 1. In this perspective the two algorithms display modularity as to their workings. Several processors may trigger reconfiguration simultaneously, but by the correctness of Algorithm 1 this does not affect the delicate reconfiguration, and by the correctness of Algorithm 2, a processor can only trigger once when this is needed.
Correctness. Algorithm 2 achieves correctness based on the ability of delicate reconfiguration in Algorithm 1 to converge to a single configuration even if many proposals are given. We use the term *steady $config$ state* to indicate a system state were $recSA$ has imposed a conflict-free state at least once. We refer to a system state c_{safe} during an execution R_{safe} of Algorithm 2, as one which contains no stale information. We first show that the algorithm eventually cleans stale information from an initial arbitrary state (in variables and program counters) after a bounded number of reconfiguration triggerings that may be the result of this arbitrary state. We then proceed to prove that $recMA$ prevents processors that are already reconfiguring to trigger a new reconfiguration.

Lemma 7 *Starting from an arbitrary initial state in an execution R, where stale information exists, Algorithm 2 eventually converges to a steady $config$ state, where local stale information is removed.*

Lemma 8 *Starting from an R_{safe} execution, any call to $estab()$ (lines 13 and 15) related to a specific event (majority collapse or agreement of majority to change $config$), can only cause a one per participant trigger. After the $config$ has been established, no triggering that relates to this event takes place.*

A *legal execution R'* of Algorithm 2, refers to an execution composed of conflict-free states and delicate configurations triggered due to loss of majority of members, or

Algorithm 3: Self-stabilizing Joining Mechanism (*JoinMec*); code for processor p_i

```
1  Interfaces. The algorithm uses following interfaces from Algorithm 1. $noReco()$ returns True if a reconfiguration
     is not taking place. $participate()$ makes $p_i$ a participant. $getConfig()$ returns the agreed configuration from
     Algorithm 1 or $\perp$ if reconfiguration is taking place. The $passQuery()$ interface to the application, returns a
     True/False in response to granting a permission to a joining processor.
2  Variables. $FD[]$ as defined in Algorithm 1. $state[]$ an array of application states, where $state[i]$ represents $p_i$'s
     local variables and $state[j]$ the state that $p_i$ most recently received from $p_j$. $pass[]$ a boolean array of passes that
     $p_i$ receives from configuration members.
3  Functions. $resetVars()$ initializes all variables related to the application based on default values. $initVars()$
     initializes all variables related to the application based on states exchanged with the configuration members.
4  procedure $join()$ begin
5      foreach $p_j \notin FD$ do $pass[j] \leftarrow$ False;
6      do forever begin
7          if $p_i \in FD[i].part$ then
8              $resetVars()$;
9              repeat
10                 let $conf = getConfig()$;
11                 if $noReco() \wedge (|\{p_j : p_j \in conf \cap FD[i] \wedge pass[j] = \text{True}\}| > \frac{|conf|}{2})$ then
                       $initVars(); participate()$;
12                 foreach $p_j \in FD[i]$ do send("Join");
13             until $p_i \in FD[i].part$;
14 upon receive ("Join") from $p_j \in FD \setminus FD[i].part$ do begin
15     if $p_i \in config \wedge noReco() =$ True then send($\langle passQuery(), state_i \rangle$);
16 upon receive $m = \langle pass, state \rangle$ from $p_j \in FD$ do if $p_i \notin FD[i].part$ then $\langle pass[j], state[j] \rangle \leftarrow m$;
```

due to the need of a majority of the members to reconfigure. Given the above lemmas, the proof concludes that a reconfiguration will take place when required and only when it is necessary, if the majority-supportive core assumption holds. This provides liveness to the application and leads to the following theorem.

Theorem 9. *Let R be an execution of Algorithm 2 starting from an arbitrary system state. R has an execution suffix R' which is a legal execution.*

3.3 Joining Mechanism (*JoinMec*)

Every processor that wants to become a participant, uses the snap-stabilizing data-link protocol (cf. Section 2) so as to avoid introducing stale information before establishing a connection with the system's processors. Algorithm 1 enables a joiner to obtain the agreed $config$ when no reconfiguration is taking place. In spite of eventually acquiring knowledge of this $config$ via $recSA$, a processor should only be able to participate in the computation if the application allows it. In order to sustain the self-stabilization property, it is also important that a new processor initializes its application-related local variables to either default values or to the latest values that a majority of the configuration members suggest. The joining protocol, Algorithm 3, illustrates the above and introduces joiners to the system as *participants* and not as $config$ *members*.

Algorithm description. Both non-participants and participants execute the algorithm.
The joiner's side. Upon a call to the $join()$ procedure, a joiner sets all the entries of its $pass[]$ array to False (line 5) and resets application-related variables to default values, (lines 8). The processor then enters a do-forever loop, the contents of which it executes only while it is not a participant (line 7). Joiners enter an inner loop in which they try to gather enough support from a majority of configuration members in order to become

participants. In every iteration, the joiner sends a "Join" request (line 12) and stores the responses by any configuration member p_j in $pass[j]$, along with the latest application $state$ that p_j had. If a majority of active members has granted a $pass = \mathsf{True}$ and there is no reconfiguration taking place, then application-related variables are initialized and $participate()$ is called to allow the joining processor to become a participant (line 11).
The participant's side. A participant only executes the do–forever loop (line 6) but none of its contents since it always fails the condition of line 7. Participants however respond to join requests, by checking whether a joining processor has the correct configuration, and whether a reconfiguration is not taking place, as well as if the application can accept a new processor (line 15). If the above are satisfied, then the participant sends a $pass = \mathsf{True}$ and its application $state$, otherwise it responds with False.
Correctness. The proof first considers safety, by establishing that a processor becomes a participant through *JoinMec* only while a reconfiguration is not taking place. In the case of a pending delicate reconfiguration, joining processors running Algorithm 3 can only wait. In case of brute force reconfiguration, $recSA$ was shown to bypass the *Join-Mec* in order to introduce more processors to the configuration. The proof proceeds to show that eventually a processor will become a participant if the application permits it, unless it crashes. Theorem 10 summarizes the correctness.

Theorem 10. *Given an execution R of Algorithm 3 with an arbitrary initial state, R has a suffix in which every joining processor p eventually becomes a participant if the application grants permission. Moreover, p respects the installed configuration and does not affect a LE as defined by Theorem 9.*

4 Applications of the Reconfiguration Scheme

Our self-stabilizing reconfiguration scheme allows applications built for static crash-prone systems to endure more adverse system dynamics. When a configuration exists and no reconfiguration is running, applications work in the same way as in their static version, since they run their service on the configuration set as in the original static setting. A main consideration, however, is functionality *during* and *after* reconfiguration.

A general framework for adapting the static algorithm to form a reconfigurable one, involves developing an interface between the application and the reconfiguration scheme to adapt the applications structures and data to the new configuration set. We note that using this framework, the algorithms are *suspending*, i.e., they do not provide service *during* reconfiguration, albeit we believe that it is possible with more elaborate frameworks and under certain conditions to sustain service even during reconfiguration. It is an interesting open question whether this is achievable, but in the meanwhile we refer the reader to [4] for tradeoffs between suspending and non-suspending services.

Due to space limitations (and to focus on presenting the reconfiguration mechanism) we omit details of how this adaptation is performed and refer the reader to [8]. There, we show how the self-stabilizing algorithms of [7] can be adapted to be reconfigurable and prove that the algorithms remain correct and extend their capabilities after this adaptation. Specifically, we present a *self-stabilizing counter* algorithm that is multipurpose (e.g., for Paxos ballot numbers, or view identifiers in group communication services). This forms the basis for *virtually synchronous state machine replication* (SMR).

5 Conclusion

We presented the first self-stabilizing reconfiguration scheme that recovers automatically from transient faults, such as temporal violations of the predefined churn rate or the unexpected activities of processors and communication channels, using a bounded amount of local storage and message size. We use a number of bootstrapping techniques for allowing the system to always recover from arbitrary transient faults, even in cases where the current configuration includes no active processors. We believe that the presented techniques provide a generic blueprint for different solutions that are needed in the area of self-stabilizing high-level communication and synchronization primitives, which need to deal with processor joins and leaves as well as transient faults.

References

1. M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7, 2011.
2. N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, S. Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *J. Comp. Syst. Sci.*, 81(4):692–701.
3. H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch. Simulating a shared register in an asynchronous system that never stops changing. In *Proc. of DISC 2015*, pp. 75–91.
4. K. Birman, D. Malkhi, and R. van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.
5. P. Blanchard, S. Dolev, J. Beauquier, and S. Delaët. Practically self-stabilizing paxos replicated state-machine. In *Proc. of NETYS 2014*, pp. 99–121.
6. S. Dolev. *Self-stabilization*. The MIT press, 2000.
7. S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing virtual synchrony. In *Proc. of SSS 2015*, pp. 248–264.
8. S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing reconfiguration. *CoRR*, abs/1606.00195, 2016.
9. S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma. Self-stabilizing end-to-end communication in dynamic networks. In *Proc. of SSS 2012*, pp. 133–147.
10. S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997.
11. S. Dolev, E. Schiller, and J. L. Welch. Random walk for self-stabilizing group communication in ad hoc networks. *IEEE Trans. Mob. Comput.*, 5(7):893–905, 2006.
12. S. Dolev and N. Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theor. Comput. Sci.*, 410(6-7):514–532, 2009.
13. E. Gafni and D. Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proc. of DISC 2015*, pp. 140–153.
14. S. Gilbert, N. A. Lynch, and A. A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.
15. L. Jehl, R. Vitenberg, and H. Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *Proc. of DISC 2015*, pp. 154–169.
16. L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.
17. P. M. Musial, N. C. Nicolaou, and A. A. Shvartsman. Implementing distributed shared memory for dynamic networks. *Commun. ACM*, 57(6):88–98, 2014.
18. A. Spiegelman, I. Keidar, and D. Malkhi. Dynamic reconfiguration: A tutorial. *OPODIS'15*.
19. M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.