Enabling Snap-Stabilization

Alain Cournier\textsuperscript{1} Ajoy K. Datta\textsuperscript{2} Franck Petit\textsuperscript{1} Vincent Villain\textsuperscript{1}
\textsuperscript{1} LaRIA, Université de Picardie Jules Verne, Amiens France
\{cournier,petit,villain\}@laria.u-picardie.fr
\textsuperscript{2} Department of Computer Science, University of Nevada Las Vegas
datta@faculty.egr.unlv.edu

Abstract

A snap-stabilizing protocol guarantees that the system always behaves according to its specification provided some processor initiated the protocol. We present how to snap-stabilize some important protocols, like Leader Election, Reset, Snapshot, and Termination Detection. We use a Snap-stabilizing Propagation of Information with Feedback protocol for arbitrary networks as the key module in the above transformation process. Finally, we design a universal transformer to provide a snap-stabilizing version of any protocol (which can be self-stabilized with the transformer of [15]).

Keywords: Fault-tolerance, leader election, propagation of information with feedback, reset protocols, self-stabilization, snapshot, snap-stabilization.

1 Introduction

The concept of self-stabilization [13] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time.

The concept of Snap-stabilization was introduced in [8]. A snap-stabilizing algorithm guarantees that it always behaves according to its specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps. Obviously, a snap-stabilizing protocol is optimal in stabilization time. This notion of zero stabilization time is a surprising result in the area of self-stabilization. It is important to note that this new paradigm does not guarantee that all components of the system never work in a fuzzy manner. What this powerful tool ensures is if an execution of a protocol is initiated by some processor, then the protocol behaves as expected. Moreover, after the above initialization, the protocol does not introduce additional fuzzy behavior. Consider the problem of mutual exclusion. Starting from an arbitrary configuration, a snap-stabilizing protocol does not guarantee that in this configuration, several processors cannot be in the critical section. But, it guarantees that for any new requests, requesting processors will not enter the critical section before the above problem is solved. The self-stabilizing (but not snap-stabilizing) protocols cannot provide this guarantee.

Contributions. We first design a snap-stabilizing version of four fundamental protocols — Reset, Snapshot, Leader Election, and Termination Detection — using the snap-stabilizing Propagation of Information with Feedback (PIF) protocol of [12]. We then show how to design a universal transformer to provide a snap-stabilizing version of any protocol. Although this method proposed is not optimal, but it demonstrates the possibility to transform almost any protocol (which can be self-stabilized) to a snap-stabilizing version.
Related Work. The concept of PIF (also called wave propagation) [10, 18] is a well-studied problem in distributed computing. The PIF scheme has been used extensively in distributed computing to solve a wide class of problems, e.g., spanning tree construction, distributed infimum function computations, snapshot, termination detection, and synchronization. PIF algorithms have been proposed in the area of self-stabilization, e.g., [7, 8, 14, 16, 20] for tree networks, and [11, 19] for arbitrary networks. The self-stabilizing PIF protocols have also been used in the area of self-stabilizing synchronizers [2, 4, 6]. Reset protocols are also PIF-based algorithms. Several reset protocols exist in the self-stabilizing literature (see [1, 3, 4, 5, 20]). Snapshot algorithms [9] are also based on the PIF scheme. Self-stabilizing snapshot algorithms are provided in [15, 20]. In [15], Katz and Perry used the self-stabilizing snapshot algorithm in designing a protocol that transforms non-self-stabilizing to self-stabilizing algorithms.

Snap-Stabilizing PIF for oriented and un-oriented tree networks are proposed in [7, 8]. Two more snap-stabilizing protocols for tree networks were presented in [17]. The first snap-stabilizing protocol for arbitrary networks is proposed in [12]. The proposed protocol [12] implements the PIF scheme. It does not use a pre-constructed spanning tree. The snap-stabilizing property guarantees that when a processor initiates the broadcast wave, the broadcast message will reach every processor. Moreover, all the feedback messages correspond to the broadcast message and will be received by the initiator the broadcast wave.

Outline of the paper. In the next section (Section 2), we describe the distributed system and the model we consider in this paper. In Section 3, we formally define the concept of snap-stabilization. In the same section, we also clarify some concepts seemingly similar in the domain of self-stabilization and snap-stabilization. In Section 4, we propose five key snap-stabilizing protocols. We first recall the snap-stabilizing PIF algorithm presented in [12]. We then present snap-stabilizing solutions to the Leader Election, Reset, Snapshot, and Termination Detection problems. In Section 5, we show their applications in developing a universal transformer. Finally, we make some concluding remarks in Section 6.

2 Preliminaries

We consider an asynchronous network of N processors with distinct IDs, connected by bi-directional links according to an arbitrary topology. We assume the local shared memory model of communication. The program of every processor consists of a set of shared variables (henceforth, referred to as variables) and a finite set of actions. A processor can only write to its own variables, and read its own variables and variables owned by the neighboring processors.

Each action is of the following form: < label >:: < guard > →< statement >. The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The state of a processor is defined by the value of its variables. The state of a system is the product of the states of all processors (∈ V). We will refer to the state of a processor and system as a (local) state and (global) configuration, respectively. A processor p is said to be enabled in Configuration γ if there exists at least an action A of p such that the guard of A is true in γ. Similarly, an action A is said to be enabled (in γ) at p if the guard of A is true at p (in γ). We assume a weakly fair and distributed daemon. The weak fairness means that if a processor p is continuously enabled, then p will be eventually chosen by the daemon to execute an action. A computation step is a transition between two configurations where the transition contains at least one action and at most one action per processor. The distributed daemon implies that during a computation step, if one or more processors are enabled, then the daemon chooses at least one (possibly more) of these enabled processors to execute an action.
3 Snap-Stabilization

A self-stabilizing algorithm achieves the convergence to a specified behavior of the system in a finite time. Consider a self-stabilizing version of a finite protocol $P$. After an initialization action (as described below), the first execution of $P$ may not satisfy the specification. The self-stabilization guarantees that by repetition of this protocol, the system eventually satisfies the specification. Solving this particular drawback is the target of the snap-stabilization. The snap-stabilization guarantees that after the first initialization action, the execution of $P$ works as expected (from the specification).

Formal Definition of Snap-Stabilization. We claimed up to this point that a snap-stabilizing protocol guarantees that the system always behaves according to its specification provided at least one initiator initiated the protocol. We now give a more formal definition.

We assume that in a normal execution, at least one processor (called, the initiator) initiates the protocol upon an external (w.r.t. the protocol) request by executing a special type of action, called an initialization action. We define the normal starting state as follows: The initiators are enabled for an initialization action, and the other processors are quiescent, i.e., they are not enabled until they receive the wave initiated by some initiator. In other words, all processors except the initiators are awakened by the receipt of one of the first waves. The initial configuration where all processors except the initiators are in a quiescent state is called a normal starting configuration. Any configuration reachable from a normal starting configuration is called a normal configuration. Any execution starting from a normal configuration is termed as a normal execution. Considering faulty networks, the system may start in a configuration which is unreachable from a normal starting configuration. We call this type of configurations abnormal configurations. Since a configuration is defined by the state of all the processors, we can claim that every processor in a normal configuration is a normal processor. So, in an abnormal configuration, there exists at least one abnormal processor.

As mentioned in [15], not every task admits self- or snap-stabilizing solution. For example, if the task is to print “1” followed by an infinite sequence of “0”s, any self- or snap-stabilizing protocol cannot guarantee better than printing an infinite sequence of “0”s. So, the above task cannot be solved. We can then make the following remark from the above example:

Remark 1 An infinite task $T$ admitting a self- or snap-stabilizing solution must satisfy the following condition: The specification of $T$ is suffix-closed.

Definition 1 (Snap-Stabilization) Let $P$ be a protocol designed to solve a task $T$. $P$ is called snap-stabilizing if and only if any execution $E$ of $P$ satisfies (i) $E$ always satisfies the specification of $T$ and (ii) if $E$ does not include any initialization action, then $E$ includes a normal configuration.

The above definition (Definition 1) may seem to be somewhat different than the informal definition we have presented so far. In the following, we will show how they correspond to each other in practice. Given a specification $SP$ for a task $T$, when we design a protocol $P$ to solve $T$, we implicitly make the following assumption (where “starts” means “at least one initialization action is executed”):

$$\forall E \text{ execution of } P, \text{ } E \text{ starts } \Rightarrow SP$$

The above assumption implies that any protocol which never starts satisfies any specification. More formally the following predicate is always true:

$$(E \text{ does not start}) \Rightarrow (E \text{ starts } \Rightarrow SP)$$
Consider an execution $A$ of a self-stabilizing protocol which includes an initialization action infinitely often (i.e., $A$ starts infinitely often). It follows from the above predicate that before $A$ starts, $A$ satisfies the specification. Let us denote this part of the execution as a default prefix. When $A$ starts, the initialization action can be executed from an abnormal configuration, and $A$ may not satisfy the specification immediately, but will eventually do forever. So, by definition of self-stabilization, Execution $A$ only guarantees that there exists a suffix which satisfies the specification forever. Let us call this suffix a correct suffix. The first step of a correct suffix includes at least one initialization action. So, between the end of the default prefix (the first initialization action of $A$) and the correct suffix, the number of times the protocol starts and does not satisfy the specification is generally unknown. Let us call this part of $A$ the stabilization factor. $A$ is the concatenation of a default prefix, a stabilization factor, and a correct suffix.

Now, consider an execution $B$ of a snap-stabilizing protocol which includes at least an initialization action. By definition, as soon as $B$ starts, even if the initialization action is executed from an abnormal configuration, $B$ satisfies the specification, and the suffix starting from this computation step is a correct suffix. So, $B$ is the concatenation of a default prefix and a correct suffix, and hence, always satisfies the specification. Thus, proving a protocol $P$, which starts at least once, to be snap-stabilizing may involve showing the following two steps: (i) Any execution includes at least one computation step which contains at least one initialization action (i.e., the default prefix is finite) and (ii) Starting from this step, any execution satisfies the specification forever (i.e., the suffix starting from this step is a correct suffix). The above two steps ((i) and (ii)) correspond to both the formal and informal definitions.

Now assume that $B$ never starts. In this case, by definition, $B$ is the sequence of configurations: $\gamma_0, \gamma_1, \ldots, \gamma_{i-1}, \gamma_i, \gamma_{i+1}, \ldots$, where $\forall j < i, \gamma_j$ is an abnormal configuration and $\forall j \geq i, \gamma_j$ is a normal configuration. Since there does not exist any initialization action in $B$, $B$ is an execution of a protocol solving an infinite task. By Remark 1, the suffix of $B$ starting in $\gamma_i$ satisfies the specification as an execution does when it starts with an initialization action.

**Delay vs. Stabilization Time.** First, let us assume an external request for a processor to initiate a protocol. We first observe the behavior of a non-faulty protocol from this point. For instance, one requests to print a file and makes the same request for a second file so that the printing of the second file is delayed until the previous printing has finished. Let us take the second request as the initial point of the observation. So, we observe a delay before the printing protocol starts for this request.

We now observe the execution of a self- or snap-stabilizing protocol. As above, the initiator has to wait for being able to start the protocol, i.e., it is waiting for the end of the default prefix. The time of the execution of the default prefix is called delay. From now, the behavior depends on whether the protocol is self- or snap-stabilizing. In the first case, the stabilization factor may not be empty. Let us call the execution time of this factor the stabilization time. Of course in the self-stabilization literature, authors include the delay in the stabilization time, since the notion of delay appears with the notion of snap-stabilization. We conjecture that in every case, the order of the stabilization time complexity is the same with or without included delay.\footnote{Until now, we did not find any counter example.} In the second case, the stabilization factor is empty, and the self-stabilization time is null.

For executions with no initialization action, the delay is the time needed to reach a normal configuration.

We should emphasize here that the notion of delay and stabilization time are clearly different. The impact of the delay is to slow down the protocol, whereas during the stabilization time, the specification of the protocol may be violated.

**4 Five Key Snap-Stabilizing Protocols**

In this section, we briefly recall the specification of a PIF algorithm and the result proposed in [12]. Then, we present how to design four essential snap-stabilizing protocols to develop a universal transformer (see Section 5).
This four protocols in this section use the snap-stabilizing PIF protocol.

4.1 Snap-Stabilizing PIF

A processor $i$, initiates the first phase (propagation or broadcast phase) of the wave. Every processor, upon receiving the first broadcast message, chooses the sender of this message as its parent in the PIF wave, and forwards the wave to its neighbors except its parent. (Thus, a spanning tree is dynamically built rooted at $i$ during the broadcast phase.) When a processor receives a feedback (acknowledgment) message from all its children with respect to the current PIF wave, it sends a feedback message to its parent. So, eventually, the feedback phase ends at the initiator $i$.

Due to lack of space, the snap-stabilizing PIF protocol, Algorithm $\mathcal{PIF}$, and its informal explanation are presented in the appendix.

**Theorem 1** ([12]) Algorithm $\mathcal{PIF}$ proposed in [12] is snap-stabilizing.

The following additional property of Algorithm $\mathcal{PIF}$ will be used to design snap-stabilizing reset protocol:

**Property 1** The feedback phase cannot be initiated before every processor has received the broadcast message.

4.2 Snap-Stabilizing Leader Election Protocol

A leader election protocol ensures that upon termination of the protocol, one and only one processor is elected as the leader. Our purpose here is not to propose a very efficient leader election protocol, rather to show that we can design a snap-stabilizing leader election protocol using Algorithm $\mathcal{PIF}$. We now describe a basic solution.

We first design a snap-stabilizing maximum ID computation (Algorithm $\mathcal{M}$) using Algorithm $\mathcal{PIF}$. Every processor in the network runs Algorithm $\mathcal{M}$ in parallel. Some processors may not initiate their own version of Algorithm $\mathcal{M}$ on their own. However, upon receiving an $\mathcal{M}$ message from some other processor, these processors also initiate the same protocol. When the PIF waves corresponding to Algorithm $\mathcal{M}$ terminate, all processors have elected the (unique) ID of the leader.

In this paper, we use a weaker version of leader election in the following sense: Each processor checks if it is the actual leader (using $\mathcal{M}$) only when it needs to. In the rest of the paper, we refer to this protocol as $\mathcal{LE}$ (instead of $\mathcal{M}$).

4.3 Snap-Stabilizing Reset Protocol

The reset protocol is used in a faulty environment to “reset” a system to a pre-defined “good” global configuration for a task $\mathcal{T}$. A good global configuration can be restricted to the normal initial configuration (see Section 3) to simplify the design process. But, any normal configuration can be used as a good global configuration. Our approach to design the reset protocol is similar to the one in [19].

We restrict the reset protocol to be initiated by one and only one initiator, called $i$. We call this rooted reset protocol Algorithm $\mathcal{R}_i$. $i$ uses Algorithm $\mathcal{PIF}$ as follows: Processor $i$ initiates broadcasting of an “abort” message, and then waits for an acknowledgment. Since Algorithm $\mathcal{PIF}$ is snap-stabilizing, all the processors will receive the abort message. From Property 1, the processors can reset their variables during the feedback phase, since they all aborted the execution of the task. So, when $i$ receives all the acknowledgment messages, the system is guaranteed to be in a normal initial configuration.


4.4 Snap-Stabilizing Snapshot Protocol

The problem of distributed snapshot (to compute a consistent global configuration of the system) is quite challenging in a faulty environment. The consistent snapshot is a collection of local state of all processors such that this collection can correspond to some normal configurations of the task. Before we present our protocol, we briefly describe the main idea of the snapshot algorithm in [9] applied on a task $T$. In this algorithm, a processor initiates the snapshot by first recording the state of $T$ and then broadcasting a marker to all its neighbors. Upon receiving the first marker message, a processor records its own state of $T$ and sends the marker to all its neighbors (including the sender of the marker). A marked processor records every message received from a neighbor which did not send the marker yet. When a processor $i$ receives a marker from all its neighbors, $i$ completes its participation in the snapshot protocol by sending a report (recorded state and messages) to the initiator. In our model (locally shared memory), the messages are modeled by the ability of a processor to read the state of its neighbors. So, the collection of the recorded messages at $i$ from a neighbor $j$ (in the message passing model) is replaced by the history of change of state of $j$ after the marking event of $i$ and before that of $q$ (in our model).

We adapt the algorithm [9] in our model to design Algorithm $S^T$ in the following manner: The markers are propagated during the broadcast phase, and the reports are sent along the path towards the root. At the end of the feedback phase of Algorithm $P_{IF}$, the root is sure to have received all the reports. Since Algorithm $S^T$ ensures that every processor participates in the snapshot protocol (property of Algorithm $P_{IF}$), we obtain a consistent snapshot when Algorithm $S^T$ terminates (property of the algorithm in [9]).

4.5 Snap-Stabilizing Termination Detection Protocol

It is easy to design a snap-stabilizing termination detection protocol for a task $T$ ($TD^T$) in the local shared memory model of communication considered in this paper. The approach is similar to the design of snap-stabilizing snapshot protocol of Section 4.4. When a processor $i$ sends a report towards the root, the report contains only a boolean field indicating if any processor in the subtree rooted at $i$ (built during the broadcast phase) is enabled w.r.t. $T$.

5 Transformer

Let $P$ be a protocol to be transformed into snap-stabilizing version.

Remark 2 As in [15], we assume that:

- We can define for $P$ a predicate $CK$ which characterizes the normal configurations of the system. This predicate can be computed by a snapshot.

- If $P$ solves infinite tasks, then it must follow Remark 1.

Let us quickly define some composition definition needed in the remainder. A fair parallel composition of two protocols $A$ and $B$ is a protocol denoted by $A \parallel B$, where actions of $A$ (resp., $B$) occur infinitely often in any execution if an infinite number of configurations contains enabled actions of $A$ (resp., $B$). $A \rightarrow \parallel B$ denotes that $A$ and $B$ cannot be executed in parallel. A sequential composition of two protocols $A$ and $B$ is the protocol denoted by $AB$, where $B$ can start if and only if $A$ has terminated.

Now, we first focus on finite protocols. We present the idea of the transformer by considering two classes of protocols:

(i) The protocols which have a unique initiator (the task is associated to a processor and cannot be merged with a
Consider a protocol $P$ which is of the class (i) with Processor $i$ as initiator. So, to transform $P$ into a snap-stabilizing protocol is simple. Let $H_1$ be Algorithm $\mathcal{R}^P_i$ where $\mathcal{R}^P_i$ is a snap-stabilizing reset initialized by $i$ for protocol $P$. We add $H_1$ as a header of $P$ such that $P$ starts only after $H_1$ terminates, i.e., we consider the protocol $H_1P$. The initialization action of $H_1P$ is the initialization action of $H_1$. Thus, starting $P$ now implies first running $H_1$ and then $P$. As the system has been reset for $P$, obviously, $P$ will behave as in a non-faulty situation. We showed above that if $H_1P$ starts (a processor executes an initialization action) during an execution, then the execution is snap-stabilizing.

Now, assume that no initialization action of $H_1$ was executed after a fault occurred, but $H_1P$ is running. Then, there is no guarantee that $H_1P$ terminates. Hence, we cannot ensure that the execution will contain at least one initialization action of $H_1P$. Since $H_1$ is snap-stabilizing, we know that $H_1$ eventually terminates. So, the reason for $H_1P$ not terminating must be due to the fact that $P$ cannot terminate (recall that $P$ is not even self-stabilizing).

Assume that $i$ regularly requests to run the snapshot algorithm (Algorithm $\mathcal{S}^P_i$ rooted in $i$) to compute the predicate $\delta \mathcal{K}$ related to $P$, i.e., we consider $P||\mathcal{S}^P_i$. If the result of Algorithm $\mathcal{S}^P_i$ is an abnormal configuration ($\neg \delta \mathcal{K}$), then $i$ starts $H_1P$. If Algorithm $\mathcal{S}^P_i$ returns a normal configuration ($\delta \mathcal{K}$), then the system is still running according to the specification of $P$ and no other external actions are invoked before running Algorithm $\mathcal{S}^P_i$ again. The projection upon the actions of $H_1P$ of the execution of the new protocol $P_i = H_1P||\mathcal{S}^P_i$ can be decomposed as follows: The default prefix is a suffix of an execution of $H_1P$. It is followed by a complete execution of $H_1P$ if $\mathcal{S}^P_i$ always returns $\delta \mathcal{K}$. Otherwise, it is any fuzzy behavior of $H_1P$ (without any initialization action) stopped by $\mathcal{S}^P_i$ returning $\neg \delta \mathcal{K}$ followed by a complete execution of $H_1P$. In the second case, we can see that $\mathcal{S}^P_i$ returns $\neg \delta \mathcal{K}$ only once. From the above discussion, we can claim the following theorem:

**Theorem 2** Let $P$ be a protocol which has a unique initiator. Then the transformed protocol $H_1P||\mathcal{S}^P_i$ is snap-stabilizing for the specification of the problem solved by $P$.

Consider a protocol $P$ which is of the class (ii). Let $i$ be one of the initiators of $P$. As in the previous case, we add a header $H_2$ to $P$ to get a new combined Algorithm $H_2P$. Again, the initialization of $P$ is the initialization of $H_2$. $H_2$ works as follows: First, Processor $i$ initiates a snap-stabilizing leader election. Let $\mathcal{L}E_i$ be the snap-stabilizing leader election initiated by $i$ to initiate $P$. We need to consider two cases:

1. $i$ is the leader of the network. Then, $i$ executes Protocol $\mathcal{L}P$ (described below).
2. $i$ is not the leader of the network. Then, $i$ requests the leader of the network the permission to initiate $P$. To send the request, we use Algorithm $\mathcal{P}IF$ ($\mathcal{P}IF^P_i$ denotes the PIF initiated by $i$ to initiate $P$). Processor $i$ initiates the broadcast of an "Init $P$" message, and then waits for the end of the feedback phase. Algorithm $\mathcal{P}IF$ being snap-stabilizing, the request will be received by every processor, in particular the actual leader of the network.

This marks the end of $H_2$, followed by the initiation of $P$. More formally:

**PROTOCOL** $H_2$

1. IF $\mathcal{L}E_i$ RETURNS $i /$*$i$ is the actual leader*$/$
2. THEN $\mathcal{L}P_i$
3. ELSE $\mathcal{P}IF^P_i$
4. ENDIF

Let us assume that Processor $i$ asks the leader for the permission (Case 2. above). Upon receipt of the broadcast message, every processor $j$ first extends the broadcast phase and then initiates the snap-stabilizing leader election $\mathcal{L}E_j$. (The execution of Algorithm $\mathcal{L}E_j$ does not prevent the progress of the broadcast phase, i.e., Algorithm $\mathcal{L}E_j$
runs in parallel with the broadcast phase.) If the result of $\mathcal{L}E_j$ indicates that $j$ is the leader of the network, then $j$ executes Protocol $\mathcal{L}P_j$. Next (and after the completion of $\mathcal{L}P_j$ if $j$ is the leader), $j$ just waits to participate in the feedback phase of $\mathcal{P}\mathcal{L}\mathcal{F}P$.

We now present Protocol $\mathcal{L}P$, supposed to be executed by only the leader $l$. The main task of Protocol $\mathcal{L}P$ is to initiate another protocol, called Protocol $\mathcal{C\mathcal{R}P}$ (see Line (2) below). However, before initiating, $l$ must ensure that no other protocol $\mathcal{C\mathcal{R}P}$ is running in the system. So, we need a snap-stabilizing termination detector rooted at $l$ which computes the predicate $TD$ defined below: $TD$ is true if and only if there exists no protocol $\mathcal{C\mathcal{R}P}$ working in the system.

**Protocol $\mathcal{L}P$**

1. WHILE $\mathcal{L}E_l$ RETURNS $l$ AND ($TD\mathcal{C\mathcal{R}P}$ RETURNS $\neg TD$) DO;
2. $\mathcal{C\mathcal{R}P}$

Similarly to the case of a unique initiator, we use a snap-stabilizing snapshot protocol $\mathcal{S}P$ to design $\mathcal{C\mathcal{R}P}$. Protocol $\mathcal{S}P$ computes the predicate $OK$ related to $P$.

**Protocol $\mathcal{C\mathcal{R}P}$**

1. IF $\mathcal{L}E_l$ RETURNS $l$/*$l$ is the actual leader*/
2. THEN $\mathcal{S}P$
3. IF $\neg OK$ THEN $\mathcal{R}P_l$ ENDIF
4. ENDIF

In summary, an initiator executes the protocol $H_2P$, while the other processors execute $\mathcal{L}P|P$. But, as in the previous case, $P$ may never terminate. So, we need to implement a similar control of the correctness of the configurations. This control has to be executed by the leader. But, it has to be implemented at each processor. Thus, the first action is to check if the processor is the actual leader. The control protocol, called $\mathcal{P}\mathcal{C\mathcal{R}P}$, is also executed in parallel with $P$ but not with $H_2$ or $\mathcal{L}P$. Formally:

**Protocol $\mathcal{P}\mathcal{C\mathcal{R}P}$**

1. IF $\mathcal{L}E_i$ RETURNS $i$
2. THEN $\mathcal{C\mathcal{R}P}$
3. ENDIF

So, we denote the transformed protocol (of $P$) as follows:

$$H_2(P|\mathcal{P}\mathcal{C}\mathcal{R}P) \lor (P|(|\mathcal{L}P|\mathcal{P}\mathcal{C}\mathcal{R}P))$$

**Remark 3** $\mathcal{L}E$, $\mathcal{S}P$, $TD\mathcal{P}$, and $\mathcal{R}P$ are snap-stabilizing finite protocols (since they are based on the snap-stabilizing protocol $\mathcal{P}\mathcal{L}\mathcal{F}$).

From Remark 3, we obtain the following result:

**Lemma 1** Every execution of Protocol $\mathcal{C\mathcal{R}P}$ is finite (even without any starting action).

**Lemma 2** Every processor which is not the actual leader of the network can execute Protocol $\mathcal{C\mathcal{R}P}$ at most once.

**Proof.** Any processor $i$ executes $\mathcal{C\mathcal{R}P}$ inside $\mathcal{L}P$ and $\mathcal{P}\mathcal{C}\mathcal{R}$. Since they cannot be executed in parallel, to execute $\mathcal{C\mathcal{R}P}$ twice, a processor $i$ has to initiate $\mathcal{L}E$. As the result of $\mathcal{L}E$ is different from $i$, $i$ cannot execute $\mathcal{C\mathcal{R}P}$.  

---

**Remark 3** $\mathcal{L}E$, $\mathcal{S}P$, $TD\mathcal{P}$, and $\mathcal{R}P$ are snap-stabilizing finite protocols (since they are based on the snap-stabilizing protocol $\mathcal{P}\mathcal{L}\mathcal{F}$).

From Remark 3, we obtain the following result:

**Lemma 1** Every execution of Protocol $\mathcal{C\mathcal{R}P}$ is finite (even without any starting action).

**Lemma 2** Every processor which is not the actual leader of the network can execute Protocol $\mathcal{C\mathcal{R}P}$ at most once.

**Proof.** Any processor $i$ executes $\mathcal{C\mathcal{R}P}$ inside $\mathcal{L}P$ and $\mathcal{P}\mathcal{C}\mathcal{R}$. Since they cannot be executed in parallel, to execute $\mathcal{C\mathcal{R}P}$ twice, a processor $i$ has to initiate $\mathcal{L}E$. As the result of $\mathcal{L}E$ is different from $i$, $i$ cannot execute $\mathcal{C\mathcal{R}P}$.  

---

**Remark 3** $\mathcal{L}E$, $\mathcal{S}P$, $TD\mathcal{P}$, and $\mathcal{R}P$ are snap-stabilizing finite protocols (since they are based on the snap-stabilizing protocol $\mathcal{P}\mathcal{L}\mathcal{F}$).

From Remark 3, we obtain the following result:

**Lemma 1** Every execution of Protocol $\mathcal{C\mathcal{R}P}$ is finite (even without any starting action).

**Lemma 2** Every processor which is not the actual leader of the network can execute Protocol $\mathcal{C\mathcal{R}P}$ at most once.

**Proof.** Any processor $i$ executes $\mathcal{C\mathcal{R}P}$ inside $\mathcal{L}P$ and $\mathcal{P}\mathcal{C}\mathcal{R}$. Since they cannot be executed in parallel, to execute $\mathcal{C\mathcal{R}P}$ twice, a processor $i$ has to initiate $\mathcal{L}E$. As the result of $\mathcal{L}E$ is different from $i$, $i$ cannot execute $\mathcal{C\mathcal{R}P}$.  

---

8
Lemma 3 Every execution of Protocol $\mathcal{L}^P$ is finite (even without any starting action).

Proof. From Lemma 1, $\mathcal{CR}$ is always finite. Moreover, by Remark 3, $\mathcal{LE}$ and $\mathcal{TD}$ are also finite. So, we just need to show that the execution of the while loop (Line 1 of $\mathcal{L}^P$) is finite. If the processor is not the leader, the loop cannot be executed more than twice (due to the result of $\mathcal{LE}$). Otherwise, by Lemma 2, the processors which are not the leader will execute $\mathcal{CR}$ total $n - 1$ times. So, eventually, $\mathcal{TD}^{CRP}$ will return $\mathcal{TD}$ equal to true.

Lemma 4 At the end of the first execution of Protocol $\mathcal{L}^P$ initiated by the leader, the system is in a normal configuration (w.r.t. $P$) and no more resets will be initiated.

Proof. Since the execution of $\mathcal{L}^P$ is finite, during the last execution of $\mathcal{TD}^{CRP}$, no processor was running $\mathcal{CR}$ while executing the part of $\mathcal{TD}^{CRP}$. So, if later, some of them execute $\mathcal{CR}$, they must initiate it. Since they are not the leader, they cannot execute anything other than the leader election part ($\mathcal{LE}$). So, at the end of the while loop, no processor except the leader can execute a reset. When the leader ends $\mathcal{CR}$, either the system has been reset to a normal starting configuration (for $P$), or the system is in a normal configuration (not necessarily a starting configuration) because the result of $\mathcal{S}_i^P$ was a normal configuration ($\mathcal{OK}$ was true).

Corollary 1 $\mathcal{L}^P$ is a snap-stabilizing finite protocol for the following specification: if $i$ is the leader, then at the end of $\mathcal{L}_i^P$, the system is normal (w.r.t. $P$) and no more resets will be initiated.

Theorem 3 The transformed protocol $H_2(P|\mathcal{PCR}^P) \lor (P|(\mathcal{L}^P-||\mathcal{PCR}^P))$ is snap-stabilizing w.r.t. the specification of the problem solved by $P$.

Proof. Any deadlock (for $P$) is eventually reset by $\mathcal{PCR}$. So, at least one initiator will be able to execute the first action of $H_2$. Let $i$ be the initiator of the $\mathcal{PIF}$ which reaches the leader first. Since $\mathcal{L}_j^P$ ($\forall j \neq i$) and $\mathcal{PIF}_i^P$ are snap-stabilizing, $i$ will eventually receive the feedback from all the processors. Before sending the acknowledgment, the leader $i$ has executed $\mathcal{L}_i^P$. So, by Lemma 4, the system is in a normal configuration and will not be reset any more. So, $i$ will start $P$ from a normal configuration.

We now focus on protocols for infinite tasks. An example of such a task is the well-known token circulation protocol. The initialization action (the creation of the token) is performed by a unique processor and the infinite task is now to circulate the token in the system forever. Contrary to the problem of printing “1” followed by an infinite sequence of “0”’s mentioned in Section 3, we do not need that the initialization action occurs at least once from the first configuration we consider. A unique token must always exist and fairly circulate in the system.

We use one of the two protocol transformations described above depending on the number of possible initiators. Let $P'$ be the new protocol. The unique initiator or the leader regularly requests to execute Algorithms $\mathcal{S}^P$ or $\mathcal{PCR}^P$, respectively. If these repeated executions never detect any problem (in $P$), then from the first execution we are sure that the configuration is normal and, from that point onwards, we can claim that the projection upon the actions of $H_1P$ (resp., $H_2P$) of the $P'$ execution is a suffix of an execution of $H_1P$ (resp., $H_2P$). This execution is infinite since $P$ solves an infinite task. If $\mathcal{S}^P$ (resp., $\mathcal{PCR}^P$) detects a problem during the default prefix, then $H_1P$ (resp., $H_2P$) starts and runs according to the specification forever.

We summarize all the above discussion in the following definition of the transformer:

Definition 2 (Transformer) Let $P$ be a protocol. The transformed protocol $T_P$ is one of the two following protocols:
- $H_1(P||\mathcal{S}_i^P)$ if $P$ has a unique initiator,
- $H_2(P||\mathcal{PCR}^P) \lor (P|(\mathcal{L}^P-||\mathcal{PCR}^P))$ if $P$ has several initiators.
From the above discussion, it is obvious that any execution of $T_P$ has a finite default prefix, and then the protocol starts $H_1P$ (resp. $H_2P$) or has reached a normal configuration (in the case of infinite task). So, we can claim the following result:

**Theorem 4** Let $P$ be a protocol satisfying Remark 2. Then the transformed protocol $T_P$ is snap-stabilizing w.r.t. the specification of the problem solved by $P$.

### 6 Conclusions

We presented a transformer similar to the one in [15], but in a weaker model, the local shared memory model of communication. In this model, we assumed the same hypothesis as in [15], i.e., any protocol that can be semi-automatically (modulo a predicate on the configurations) transformed into a self-stabilizing protocol can be transformed into a snap-stabilizing one also.

Our transformer is based on the snap-stabilizing PIF protocol on asynchronous arbitrary networks in [12]. This protocol is a key module in the sense that it can be used to design numerous snap-stabilizing protocols, e.g., reset, snapshot, leader election, and termination detection. We showed in this paper that the PIF protocol is also a critical tool to transform almost any protocol into a snap-stabilizing version semi-automatically.

An open question related to the transformer is “Can we design such a transformer in the model used in [15]?”

### References


