Resource Efficient Stabilization for Local Tasks despite Unknown Capacity Links

Lélia Blin, Anaïs Durand, Sébastien Tixeuil

Resource Efficient Stabilization for Local Tasks despite Unknown Capacity Links

Lélia Blin[1], Anaïs Durand[2], and Sébastien Tixeuil[3]

1   Sorbonne Université, UPMC Univ Paris 06, CNRS, Université
    d'Evry-Val-d'Essonne, LIP6 UMR 7606, 4 place Jussieu 75005, Paris, France.
2   Université Clermont Auvergne, LIMOS, Clermont-Ferrand, France.
3   Sorbonne Université, CNRS, LIP6, FR-75005, Paris, France.

──── **Abstract** ────

Self-stabilizing protocols enable distributed systems to recover correct behavior starting from any arbitrary configuration. In particular, when processors communicate by message passing, fake messages may be placed in communication links by an adversary. When the number of such fake messages is unknown, self-stabilization may require huge resources:

- generic solutions (*a.k.a.* data link protocols) require unbounded resources, which makes them unrealistic to deploy,
- specific solutions (*e.g.*, census or tree construction) require $O(n \log n)$ or $O(\Delta \log n)$ bits of memory per node, where n denotes the network size and Δ its maximum degree, which may prevent scalability.

We investigate the possibility of resource efficient self-stabilizing protocols in this context. Specifically, we present a self-stabilizing protocol for $(\Delta + 1)$-coloring in any n-node graph, under the asynchronous message-passing model. The problem of $(\Delta + 1)$-coloring is considered a benchmarking problem for local tasks. Our protocol offers many desirable features.

It is deterministic, it converges in $O(k\Delta n^2 \log n)$ message exchanges, where k is the bound of the link capacity in terms of number of messages, and it uses messages on $O(\log \log n + \log \Delta)$ bits with a memory of $O(\Delta \log \Delta + \log \log n)$ bits at each node. The resource consumption of our protocol is thus almost oblivious to the number of nodes, enabling scalability.

Moreover, a striking property of our protocol is that the nodes do not need to know the number, or any bound on the number of messages initially present in each communication link of the initial (potentially corrupted) network configuration. This permits our protocol to handle any future network with unknown message capacity communication links.

A key building block of our coloring scheme is a spanning directed acyclic graph construction, that is of independent interest, and can serve as a useful tool for solving other tasks in this challenging setting.

Keywords and phrases Self-stabilizing algorithm, message passing, unbounded capacity communication, nodes coloring

1 Introduction

Self-stabilization [9, 10, 23] is a versatile technique that enables recovery after arbitrary *transient* faults hit the distributed system, where both the participating processes and the communication medium are subject to be corrupted. Roughly, a self-stabilizing protocol is able to bring the system back to a legal configuration, starting from an arbitrary initial, potentially corrupted, configuration. The core motivation for designing self-stabilizing protocols has been underlined by Varghese and Jarayam [25], who observed that, whenever processes can crash and recover, a message-passing distributed system may reach any arbitrary global state, where the local variables stored at the processes may be inconsistent, and/or the communication links may contain spurious erroneous messages. As the global state is arbitrary, one may even assume that the local variables at the nodes as well as the contents of the messages are adversarially set to prevent recovery.

It is worth pointing out that self-stabilization in the presence of fake messages in the communication links is a challenge. Specifically, it is particularly difficult to ensure recovery when no upper bound is known on the number of (possibly fake) messages in transit in the initial configuration. Conversely, if an upper bound is known, then one can reset the system to a clean configuration by, first, emptying the links, and, second, resetting all local variables using a protocol that can "trust" the messages. Hence, non surprisingly, the vast majority of recent works in self-stabilization assumes a weaker adversary than the one we consider in this paper. In particular, a widely used model of self-stabilization is the *state model*.

State Model, and Data Link Protocols. In the state model, processes atomically read the states of their neighboring processes for updating their state. That is, the state model abstracts away all issues related to corrupted communication media. Indeed, the state model is motivated by the fact that there exist self-stabilizing *data link* protocols [1, 6, 11, 12, 16, 19, 24]. The purpose of such protocols is to ensure reliable communications between neighbors exchanging messages over unreliable communication links. Yet, the use of data link protocols such as the ones in the aforementioned previous work yields important issues. In particular, if the initial number of spurious messages in the communication links is unknown, then it is impossible to design a self-stabilizing data link protocol using bounded memory at each node [1, 6, 12, 16, 19]. Therefore, one has to relax the constraints on the protocol. Such a relaxation may consist in designing *pseudo-stabilizing* data link protocols [6]. However, a pseudo-stabilizing protocol only guarantees that an infinite suffix of the execution satisfies the specification of the system, and hence its stabilization time becomes unbounded. Another relaxation consists of using self-stabilizing data link protocols that are *not bounded* in terms of resources. However, such protocols require either unbounded variables [12, 16, 19], or unbounded code size (*a.k.a.* aperiodic functions) [1], which is undesirable from a practical point of view. A third relaxation consists of using *randomization* [1], but then the correctness of the system is not certain.

As a consequence, in the framework of data link self-stabilization, previous work often assume that the initial number of spurious messages present in the communication links is known to the participating processes. Under this assumption, very efficient self-stabilizing solutions can be obtained (see, *e.g.*, [24]). Actually, assuming that the number of erroneous messages initially present in the links is known to the nodes, even stronger self-stabilizing properties can be guaranteed, such as *snap-stabilization* [11] (a snap-stabilizing data link protocol guarantees that reliable communications between participating processes are immediately available after a failure).

From the above, one can conclude that, while the knowledge of the number of possible initial spurious messages in the communication links enables the design of efficient self-stabilizing data link protocols, the lack of this knowledge precludes the existence of bounded-size self-stabilizing deterministic data link solutions. In other word, when no bound is known, the use of data link protocols does not provide fully practical solutions for the design of efficient self-stabilizing protocols. Therefore, one has to focus on self-stabilization for message-passing systems, without using data links.

Message-Passing Model. Self-stabilizing protocols that operate in message passing systems with unknown initial link capacity are the most versatile, since they can directly be executed in newly set up networks whose characteristics were unknown when the protocol was designed. However, previous works that do not rely on a data link layer require large messages, large memory, or both. For instance, the versatile *census protocol* in [8] collects the entire topology at each node, and therefore requires message and memory of $O(n\Delta \log n)$ bits in n-node networks with maximum degree Δ. Similarly, the versatile technique based on so-called *r-operators*, where the algebraic properties of the executed protocol guarantees self-stabilization, has been shown to be quite efficient in general [7, 13, 14]. However, to our knowledge, in the message passing setting, there exists r-operators only for variants of tree construction tasks [7]. Moreover, while the size of the messages used by such protocols remain in $O(\log n)$ bits, the memory at each node might grow as much as $\Omega(\Delta \log n)$ bits.

Local tasks. Vertex coloring One may wonder whether the large amount of memory and communications resources used in the context of unknown capacity links is due to the global nature of the task to be solved (census, tree construction). Hence, an intriguing question arises: do local tasks yield high resource consumption when solved self-stabilizingly with unknown capacity links? A benchmarking local task in the domain of self-stabilization is that of vertex coloring. In vertex coloring, every process in the network must maintain a color variable such that, for every two adjacent nodes, the value of their color variables is distinct. Typically, the number of colors is supposed to be restricted in the range $\{1, 2, \ldots, \Delta + 1\}$ in networks with nodes with maximum degree Δ. Vertex coloring is one of the most studied tasks in distributed network computing in general, and in self-stabilization in particular, as witnessed by numerous contributions: [2, 3, 4, 5, 15, 17, 18, 20, 21, 22]. While most previous work about self-stabilizing vertex coloring considered the state model (see [4, 5, 15, 17, 22]), a few paper considered the message passing model [2, 18, 20, 21]. However, all existing solutions for this latter model provide probabilistic guarantees only [2, 18, 20, 21], and most of them assume some strong or weak forms of synchronous execution model [2, 18, 21]. To our knowledge, there are no deterministic self-stabilizing vertex coloring protocols that operate in the *asynchronous* message passing setting, where the number of initial spurious messages in communication links is *unknown* to the participating processes.

Our contribution. To this paper, it remains unknown whether resource efficient self-stabilizing solutions exist when communication links have unknown initial capacity. We show that, for some local tasks, the answer to this question is positive.

In more details, we establish the following result: without any assumption on the number of messages present initially (and their content), starting from any configuration, we present a protocol that is self-stabilizing for the task of vertex coloring and uses $O(\log \log n + \Delta \log \Delta)$ bits of memory per node, and $O(\log \log n + \log \Delta)$ bits of information per message, where n denotes the number of nodes in the network, and Δ its maximum degree. A key ingredient of

our protocol of independent interest is a symmetry breaking mechanism that locally orients every link in the network so that the overall orientation is acyclic (hence constructing a directed acyclic graph in the network), simplifying the design of higher layer algorithms such as vertex coloring or maximal independent set.

Our work thus paves the way toward resource efficient self-stabilizing protocols for the most challenging communication model, enabling solutions to remain valid when new networks are considered.

2   Model

The communication model consists of a point-to-point communication network described by a connected graph $G = (V, E)$ where the nodes V represent the processes and the set E represent bidirectional communication channels. Processes communicate by message passing: a process sends a message to another by depositing the message in the corresponding channel. We denote by $N(v)$ the set of processes that are *neighbors* with process v, *i.e.*, such that there exists a communication link between them and v. Let denote by n the number of processes and Δ the degree of the graph. Note that we denote by $\delta(v)$ the degree of node v.

Communications.   The communication model is asynchronous message passing with FIFO channels (on each link messages are delivered in the same order as they have been sent). The number of messages per link is bounded by an integer k, however the nodes do not know k. $Q(u, v) = (m_q, m_{q-1}, \ldots, m_1)$ is the queue representing the messages in FIFO order between u and v, where m_1 is the head of the queue and $q \leqslant k$. We assume each node v is *fair* with respect to its input channels: if v receives an infinite number of messages, then a given message m cannot stay in v's input channel forever. We denote by $\mathcal{M}$ the set of all possible messages. A node v has access to locally unique port numbers associated with its adjacent edges. We do not assume any consistency between port numbers of a given edge. In short, port numbers are constant throughout the execution but two neighboring processes can associate different port numbers for the communication link between them. We denote by $\mathsf{Ports}(v)$ the set of port numbers for the adjacent edges of process v. The port number associated by v to the edge (v, u), if it exists, is denoted $\mathsf{pt}(v, u)$.

Execution.   Each process v maintains some *variables*. We denote by var_v the value of variable var at process v. The *state* of a process is the vector of the values of its variables. A *configuration* is the vector of the state of every process and the content of the channels between every two neighboring processes. We denote by $var_v(\gamma)$ the value of variable var_v in configuration γ. Similarly, $var_v[u, \gamma]$ and $Q(\gamma, v, u)$ denote the value of the entry u of array var_v in γ and the content of channel $Q(v, u)$ in γ, respectively. The set of every configuration is denoted Γ.

Let $\mapsto$ be the binary relation between configurations such that $\gamma \mapsto \gamma'$ if the system can reach γ' from γ by executing an *(atomic) step*. During a step, some processes: (a) receive messages (at most one by incoming channel), (b) do some internal computation, and (c) send some messages (at most one by outgoing channel). An *execution* is a maximal sequence of configuration $e = \gamma_0, \gamma_1, \ldots, \gamma_i, \ldots$ such that $\forall i > 0$, $\gamma_{i-1} \mapsto \gamma_i$. Configuration γ_0 is the *initial configuration* of e.

For the purpose of complexity analysis, we define a *round* as the smallest execution fragment such that the two following conditions hold: *(i)* all messages that were in transit

at the beginning of the round are received (and processed) by the nodes, and *(ii)* all nodes that have no message in transit in their input channels trigger a timeout (and process it).

Identifier. A node v has access to a constant unique identifier ID_v, but can only access its identifier one bit at a time, using the $\text{Bit}_v(i)$ function. The function Bit returns the place of the bit at one. Specifically, $\text{Bit}_v(i)$ returns the position (numbered from right to left) of the ith bit to one (from left to right). Note that since nodes have unique identifiers, they are allowed to execute unique code. For example, suppose node v has identifier 10 (in decimal notation), or 1010 (in binary notation). Then, one can implement $\text{Bit}_v(i)$ as follows for $v = 1010$:

$$\text{Bit}_v(i) := \begin{cases} 4 & \text{if} \quad \text{i=1} \\ 2 & \text{if} \quad \text{i=2} \\ \text{-1} & \text{if} \quad i > 2 \end{cases}$$

Since we assume that all identifiers are $O(\log n)$ bits long, the Bit_v function only returns values with $O(\log \log n)$ bits. Also, when executing Function Bit_v, the program counter only requires $O(\log \log n)$ values. In turn, this position can be encoded with $O(\log \log n)$ bits when identifiers are encoded using $O(\log n)$ bits, as we assume they are.

Self-stabilization. An algorithm $\mathcal{A}$ is *self-stabilizing* for some specification SP if there exists a subset of *legitimate configurations* $\Gamma_\ell \subseteq \Gamma$ such that:
- **Closure:** Γ_ℓ is closed, *i.e.*, $\forall \gamma, \gamma' \in \Gamma$ such that $\gamma \mapsto \gamma'$, if $\gamma \in \Gamma_\ell$ then $\gamma' \in \Gamma_\ell$.
- **Convergence:** $\Gamma \triangleright \Gamma_\ell$, *i.e.*, for any execution $e = \gamma_0, \gamma_1, \ldots, \gamma_i, \ldots$ of $\mathcal{A}$starting from an arbitrary initial configuration $\gamma_0 \in \Gamma$, $\exists i \geqslant 0$ such that $\gamma_i \in \Gamma_\ell$.
- **Correctness:** For any execution $e = \gamma_0, \gamma_1, \ldots, \gamma_i, \ldots$ of $\mathcal{A}$starting from a legitimate configuration $\gamma_0 \in \Gamma_\ell$, e satisfies the specification SP.

3 Algorithm DAG

The first layer of our solution consists in providing a symmetry-breaking mechanism. Our approach is to construct a directed acyclic graph (or DAG) based on the unique identifiers of the nodes: hence an edge is to be oriented from the lower identifier to the higher identifier. Of course, neighboring nodes do *not* know the identifier of the other, and should not communicate them directly to each other as it would break the $o(\log n)$ bits constraint on messages.

Our algorithm is presented as Algorithm 1. For each adjacent link (v, u) of process v, v maintains a binary variable $\mathsf{Ord}[v](p)$, where $p = \mathsf{pt}(v, u)$, as follows. $\mathsf{Ord}[v](p)$ represents the orientation of (v, u). More precisely, $\mathsf{Ord}[v](p)$ equals 0 if v's identifier is greater than the identifier of u and 1 otherwise. To update variable $\mathsf{Ord}[v](p)$, v permanently exchanges its identifier with u in a compact manner using the Bit function and a counter variable $\mathsf{cnt}_v \in \{1, \ldots, \lceil \log Id_v \rceil\}$ of size $O(\log \log n)$ bits.

When the counter equals to 1, v sends a message to every neighbor u requesting the position of their own first bit to one (see Line 10). A neighbor u answers by a message $< 1, B >$, where B is the position of the 1st bit to one of u (see Line 12). Now, if $B > \text{Bit}_v(1)$, u's identifier is greater than that of v, and if $B < \text{Bit}_v(1)$, then u's identifier is smaller than that of v. Finally, if $B = \text{Bit}_v(1)$, the comparison must continue. When v receives answers from every neighbor about the 1st position, v increments its counter (see Line 7. Then, for every neighbor whose status remains unknown (that is, the condition $B = \text{Bit}_v(1)$ was

satisfied), v requests the value of the 2nd bit, and so on until all link orientations are established. This comparison process restarts when the node executes function $Restart(v)$ (see Lines 1-3) that, in particular, sets cnt_v to 1.

Due to the arbitrary initial configuration, where the link can contain an unknown number of corrupter messages, this process repeats indefinitely. A "Do forever" process is used to cope with the case of a deadlock due to an absence of messages (see Lines 21-21). Moreover, messages $< \ell, B >$ that are received by v when its counter cnt_v is different than ℓ are discarded.

In addition to variables $\mathsf{Ord}[v](p)$ and cnt_v, process v maintains the following variables. The variable $\mathsf{tmp}_v(p) \in \{0, 1, \bot\}$ is a temporary variable used during the computation of the ordering of the identifiers. When $\mathsf{tmp}_v(p) = \bot$, the orientation of the link (v, u) (if $p = \mathsf{pt}(v, u)$) has not been computed yet in this step of the comparison. Moreover, the variable wait_v is a set of port numbers used to remember which neighbors have not yet responded during the current step of computation of the ordering.

Algorithm 1: DAG Algorithm

---

1 Function $Restart(v)$ **is**
2 $\quad$ $\mathsf{cnt}_v := 1$; $\mathsf{wait}_v := \mathsf{Ports}(v)$;
3 $\quad$ **forall** $p \in \mathsf{Ports}(v)$*:* $\mathsf{tmp}_v[p] := \bot$;

4 Function $Step(v)$ **is**
5 $\quad$ **if** $\mathsf{wait}_v = \varnothing$ **then**
6 $\quad\quad$ **if** $\mathsf{cnt}_v < \lceil \log id_v \rceil$ **then**
7 $\quad\quad\quad$ $\mathsf{cnt}_v := \mathsf{cnt}_v + 1$;
8 $\quad\quad\quad$ $\mathsf{wait}_v := \{p \in \mathsf{Ports}(v) : \mathsf{tmp}_v[p] = \bot\}$
9 $\quad\quad$ **else** $Restart(v)$;
10 $\quad$ send $< \mathsf{cnt}_v >$ to wait_v ;

11 Upon receipt of $< \ell >$ *from port* p **:**
12 $\quad$ send $<\ell, \mathrm{Bit}_v(\ell)>$ to p;
13 $\quad$ $Step(v)$;

14 Upon receipt of $< \ell, B >$ *from port* p **:**
15 $\quad$ **if** $(p \in \mathsf{wait}_v) \wedge (\ell = \mathsf{cnt}_v)$ **then**
16 $\quad\quad$ $\mathsf{wait}_v := \mathsf{wait}_v \backslash \{p\}$;
17 $\quad\quad$ **if** $B > \mathrm{Bit}_v(\ell)$ **then** $\mathsf{Ord}_v[p] := 1; \mathsf{tmp}_v[p] := 1$;
18 $\quad\quad$ **if** $B < \mathrm{Bit}_v(\ell) \vee (B = \varnothing)$ **then** $\mathsf{Ord}_v[p] := 0; \mathsf{tmp}_v[p] := 0$;
19 $\quad$ $Step(v)$;

20 Do forever :
21 $\quad$ **if** $\exists p \in \mathsf{Ports}(v) : (p = \mathsf{pt}(v, u)) \wedge (Q(u, v) = \varnothing)$ **then** $Step(v)$;

We now state the main result about Algorithm 1:

▸ **Theorem 1.** *Algorithm 1 solves the spanning DAG construction problem in a self-stabilizing manner in n-nodes graphs with maximum degree* Δ*, assuming the message passing model. If the n node identifiers are in* $[1, n^c]$ *for some* $c \geqslant 1$*, then Algorithm 1 uses* $O(\log \log n + \Delta \log \Delta)$ *bits of memory per node and* $O(\log \log n)$ *bits per message. Moreover, it converges after in* $O(k \log n)$ *rounds, and the exchange of* $O(k \Delta \log n)$ *messages, where k (unknown to the algorithm) is the maximum number of (potentially corrupted) messages initially present*

in each communication link.

In our algorithm, we use two types of messages: $< \ell_i >$ and $< \ell_i, B_i >$, where $\ell_i \in \{1, \ldots, \lceil \log n \rceil\}$ and $B_i \in \{-1, \ldots, \lceil \log n \rceil\}$. In the sequel, we may use ℓ_i to denote both $< \ell_i >$ and $< \ell_i, B_i >$.

Let $\lambda_m : \Gamma \times \mathcal{M} \times V \times V \to \mathbb{N}$ be the following function:

$$\lambda_m(\gamma, m, u, v) = \begin{cases} 1 & \text{if } m = < \ell, B > \wedge (B \neq \text{Bit}_u(\ell)) \\ 0 & \text{otherwise} \end{cases}$$

Let $\lambda_Q : \Gamma \times V \times V \to \mathbb{N}$ be the following function:

$$\lambda_Q(\gamma, u, v) = \sum_{m \in Q(\gamma, u, v)} \lambda_m(\gamma, m, u, v)$$

Let $\lambda, \Lambda : \Gamma \times V \to \mathbb{N}$ be the following functions:

$$\lambda(\gamma, v) = \sum_{u \in N(v)} \lambda_Q(\gamma, u, v) \text{ and } \Lambda(\gamma) = \sum_{v \in V} \lambda(\gamma, v)$$

Finally, we define the set of configurations $\Gamma_{\mathcal{B}}$ as follows $\Gamma_{\mathcal{B}} = \{\gamma \in \Gamma : \Lambda(\gamma) = 0\}$

▸ **Lemma 2.** $\Gamma \rhd \Gamma_{\mathcal{B}}$, *and* $\Gamma_{\mathcal{B}}$ *is closed.*

Proof. When node v receives message $< \ell, B >$ from v in configuration γ with $B \neq \text{Bit}_u(\ell)$, v sends in configuration $\gamma' > \gamma$ a message $< cnt_v >$ (see Function $Step(v)$ of Algorithm 1). If node u receives message $< \ell >$, it sends $< \ell, \text{Bit}_u(\ell) >$ (see Line 11 of Algorithm 1) followed by $< cnt_v >$ (see Function $Step(v)$ of Algorithm 1). So, u does *not* send $< \ell, B >$ with $B \neq \text{Bit}_u(\ell)$. As a consequence, $\lambda_Q(\gamma, u, v) < \lambda_Q(\gamma', u, v)$ and $\Lambda(\gamma) < \Lambda(\gamma')$. Note that, we obtain $\lambda_Q(\gamma, u, v) = 0$ after at most k receptions of messages through the port p of node v (p being the port number of v leading to u). ◂

Let define $dif(v, u)$ as the first level at which a node v can determine if the identifier of its neighbor u is lower or greater. More formally:

$$dif(v, u) = \min \{\ell : \ell \in \{1, \ldots, \lceil \log n \rceil\} \wedge \text{Bit}_v(\ell) \neq \text{Bit}_u(\ell)\}$$

We define predicate $Good(\gamma, v, u)$ as follows:
- If $id_v < id_u$:

$$Good(\gamma, v, u) \equiv \big[(\text{cnt}_v(\gamma) \leqslant dif(u, v)) \wedge (\text{tmp}_v[\gamma, u] = \bot)\big] \vee \tag{1}$$
$$\big[(\text{cnt}_v(\gamma) \geqslant dif(u, v)) \wedge (\text{tmp}_v[\gamma, u] = 1) \wedge \text{pt}(v, u) \notin \text{wait}_v(\gamma)\big] \tag{2}$$

- If $id_v > id_u$:

$$Good(\gamma, v, u) \equiv \big[(\text{cnt}_v(\gamma) \leqslant dif(u, v)) \wedge (\text{tmp}_v[\gamma, u] = \bot)\big] \vee \tag{3}$$
$$\big[(\text{cnt}_v(\gamma) \geqslant dif(u, v)) \wedge (\text{tmp}_v[\gamma, u] = 0) \wedge \text{pt}(v, u) \notin \text{wait}_v(\gamma)\big] \tag{4}$$

Let $\phi_p(\gamma, u, v) : \Gamma \times V \times V \to \mathbb{N}$ be the following function:

$$\phi_p(\gamma, u, v) = \begin{cases} 0 & \text{if } (id_u > id_v) \wedge (\text{Ord}_v[\gamma, u] = 1) \wedge Good(\gamma, v, u) \\ 0 & \text{if } (id_u < id_v) \wedge (\text{Ord}_v[\gamma, u] = 0) \wedge Good(\gamma, v, u) \\ 1 & \text{otherwise} \end{cases}$$

Let $\phi : \Gamma \times V \to \mathbb{N}$ and $\Phi : \Gamma \to \mathbb{N}$ be the following potential functions:

$$\phi(\gamma, v) = \sum_{u \in N(v)} \phi_p(\gamma, u, v) \qquad \text{and} \qquad \Phi(\gamma) = \sum_{v \in V} \phi(\gamma, v)$$

Finally, we define the set of configurations Γ_{DAG} as the following $\Gamma_{\text{DAG}} = \{\gamma \in \Gamma_{\mathcal{B}} : \Phi(\gamma) = 0\}$

▶ **Lemma 3.** Γ_{DAG} *is closed.*

Proof. Let $\gamma \in \Gamma_{\text{DAG}}$. Let $\gamma' \in \Gamma$ such that $\gamma \mapsto \gamma'$. Without loss of generality, let consider two neighboring processes u and v such that $id_v < id_u$.

Notice that, since $\phi_p(\gamma, u, v) = 0$, $Good(\gamma, v, u)$ holds and cnt_v is incremented at most once every step or reset to 1 if v executes $Restart(v)$. In the latter case, v also set $\mathsf{tmp}_v[u]$ to $\perp$ so the first line of predicate $Good$ holds. Now, there are three cases:

- If $\mathsf{cnt}_v(\gamma) < dif(v, u)$, then $\mathsf{cnt}_v(\gamma') \leqslant dif(v, u)$ and the value of variable $\mathsf{tmp}_v[u]$ is not changed so $\mathsf{tmp}_v[\gamma, u] = \perp$ and the first line of predicate $Good$ holds.
- If $\mathsf{cnt}_v(\gamma) = dif(v, u)$, the value of variable $\mathsf{tmp}_v[u]$ changes only if v executes $Restart(v)$ (see above) or if v receives a message $< \mathsf{cnt}_v, B >$ from u. Since $\gamma \in \Gamma_B$ and by definition of $dif(v, u)$, $B > \mathrm{Bit}_v(\mathsf{cnt}_v)$ and v sets $\mathsf{tmp}_v[u]$ to 1 and removes $\mathsf{pt}(v, u)$ from $\mathsf{wait}(v)$. Thus, the second line of predicate $Good$ holds.
- If $\mathsf{cnt}_v(\gamma) > dif(v, u)$, the value of variable $\mathsf{tmp}_v[u]$ changes only if $Restart(v)$ is executed. Otherwise, the second line of predicate $Good$ remains true.

In those three cases, the value of variable $\mathsf{Ord}_v[u]$ can only change to the value of $\mathsf{tmp}_v[u] \neq \perp$ so $\mathsf{Ord}_v[u]$ can only equal 1.

Thus, $\gamma' \in \Gamma_{\text{DAG}}$ and Γ_{DAG} is closed. ◀

▶ **Lemma 4.** $\Gamma_{\mathcal{B}} \rhd \Gamma_{\text{DAG}}$.

Proof. To prove $\Gamma_{\mathcal{B}} \rhd \Gamma_{\text{DAG}}$, we need to prove that every node v executes at least once the function $Restart(v)$, because this function initiates again the computation of the DAG for a node v. Thanks to the function $Restart(v)$, that puts all the ports number of v in the set wait_v, the comparison of the place of the first bit is restarted for all the neighbors of v. Moreover, since the counter is also restarted at 1, we compute which neighbors have a greater identity and which one has a smaller identity.

Only the function $Step(v)$ of Algorithm 1 can call the function $Restart(v)$ see Line 9 . The function $Step(v)$ is used in three cases:

- By the procedure **Do forever** in order to avoid a deadlock due to an absence of messages in the network (see Line 21).
- When v receives $< \ell >$ from u, to avoid filling queue $Q(v, u)$ with messages $< \ell, B >$, as this would prevent v from updating its own variables (see Line 11).
- When v receives $< \ell, B >$ from u, enabling v to update its own variables (see Line 19).

We denote by $dis(\gamma, v, u)$ the distance in the queue of the first message in $Q(\gamma, u, v)$ that decreases $|wait_v|$, if it exists:

$$dis(\gamma, u, v) = \min\{i : m_i \in Q(\gamma, u, v) \wedge (m_i =< \mathsf{cnt}_v(\gamma), B_i >)\}$$

Otherwise, if such a message does not exist, we handle the last message $< \ell_i >$ in $Q(\gamma, v, u)$ with $\ell_i = \mathsf{cnt}_v(\gamma)$. Note that, if $dis(\gamma, u, v) = 0$ and $Q(\gamma, u, v)$, v must have received all messages in $Q(\gamma, u, v)$ before the reception of messages sent by u after configuration γ. When u receives $< \ell >$ from v, u sends two messages: $< \ell, \mathrm{Bit}_u(\ell) >$, and $< \mathsf{cnt}_u >$ (see

Lines 11 and 10). As a consequence, the number of messages before $< \mathsf{cnt}_v(\gamma) >$ can be doubled when they are sent in $Q(u, v)$.

$$I = \min\{i : m_i \in Q(\gamma, v, u) \wedge (m_i =< \mathsf{cnt}_v(\gamma) >)\}$$

$$dis(\gamma, v, v) = I + |Q(\gamma, u, v)| + 2(|Q(\gamma, v, u)| - I)$$

If $dis(\gamma, u, v) = 0 \wedge dis(\gamma, v, v) = 0 \wedge Q(\gamma, u, v) \cup Q(\gamma, v, u) \neq \varnothing$. In other words, $\forall m \in Q(\gamma, v, u)$, message m is of type $< \ell, B >$, and $\forall m \in Q(\gamma, u, v)$, message m is of type $< \ell >$. As a consequence, when v receives a message $< \ell >$, thanks to function **Step(v)** it sends a message $< \mathsf{cnt}_v >$, so the distance to a message $< \ell, B >$ at destination u of v is $q(\gamma, u, v) = |Q(\gamma, v, u) \cup Q(\gamma, u, v)| + 1$. Last, if $Q(\gamma, u, v) \cup Q(\gamma, v, u) = \varnothing$, then u and v can execute **Do forever** (or only u or only v), the maximum distance 2 is obtained when u and v jointly execute **Do forever**. Let $d(\gamma, u, v) : \Gamma \times V \to \mathbb{N}$ be the following function:

$$d(\gamma, u, v) = \begin{cases} dis(\gamma, u, v) & \text{if } dis(\gamma, u, v) > 0 \\ dis(\gamma, v, v) & \text{if } dis(\gamma, u, v) = 0 \wedge dis(\gamma, v, v) > 0 \\ q(\gamma, v, v) & \text{if } dis(\gamma, u, v) = 0 \wedge dis(\gamma, v, v) = 0 \wedge Q(\gamma, u, v) \cup Q(\gamma, v, u) \neq \varnothing \\ 3 & \text{if } Q(\gamma, u, v) \cup Q(\gamma, v, u) \neq \varnothing \end{cases}$$

$$D(\gamma, v) = \sum_{u \in wait_v} d(\gamma, u, v)$$

Once wait_v is empty (see Line 5), v increases cnt_v, or executes $Restart(v)$ (see Lines 6 and 9). We already proved that in order to reach a configuration where $\phi_p(\gamma, u, v) = 0$, v must execute $Restart(v)$. To achieve that we define the function $\epsilon : \Gamma \times V \to \mathbb{N}$:

$$\epsilon(\gamma, v) = \begin{cases} \lceil \log id_c \rceil - \mathsf{cnt}_v(\gamma) + dif(u, v) & \text{if } \mathsf{cnt}_v > dif(u, v) \\ dif(u, v) - \mathsf{cnt}_v & \text{otherwise} \end{cases}$$

Let $\alpha : \Gamma \times V \to \mathbb{N}$ be the following function:

$$\alpha(\gamma, v) = (\epsilon(\gamma, v), D(\gamma, v))$$

So, when $\alpha(\gamma, v) = 0$, for all configuration $\gamma' \geqslant \gamma$, $\phi(\gamma', v) = 0$. Let us consider a configuration $\gamma' > \gamma$ with $\gamma \in \Gamma_{\mathcal{B}}$, we consider different situations:

- $\mathsf{wait}_v(\gamma) = \varnothing$, and $\forall u \in Q(\gamma, u, v) = \varnothing$, v executes **Do forever**, and thanks to function **Step(v)**, v increases or restarts cnt_v, so $\alpha(\gamma', v) < \alpha(\gamma, v)$.
- $\mathsf{wait}_v(\gamma) = \varnothing$, and $\exists u \in Q(\gamma, u, v) \neq \varnothing$, v receives at least one message, so v executes **Step(v)**, and increases or restarts cnt_v, so $\alpha(\gamma', v) < \alpha(\gamma, v)$.
- Now, if $\mathsf{wait}_v(\gamma) \neq \varnothing$, only the messages $< \ell, B >$ received by a port in the set $\mathsf{wait}_v(\gamma)$ can decrease $|\mathsf{wait}_v(\gamma)|$ in order to reach $\mathsf{wait}_v(\gamma') = \varnothing$ (with $\gamma' > \gamma$), and increases or restarts cnt_v .
  - If $\forall u \in \mathsf{wait}_v(\gamma)$, we have $Q(\gamma, u, v) \cup Q(\gamma, v, u) = \varnothing$, and function $D(\gamma, v)$ returns $3|\mathsf{wait}_v(\gamma)|$, then v executes **Do forever**. The execution of **Do forever** triggers the sending of messages $< \mathsf{cnt}_v >$ through every port $p \in \mathsf{wait}_v(\gamma)$. As a consequence, function $D(\gamma, v)$ returns at most $2|\mathsf{wait}_v(\gamma)|$, so $\alpha(\gamma', v) < \alpha(\gamma, v)$.
  - If $\exists u$ such that $Q(\gamma, u, v) \cup Q(\gamma, v, u) \neq \varnothing$, thanks to function $d(\gamma, u, v)$, we obtain $d(\gamma', u, v) < d(\gamma, u, v)$, so $\alpha(\gamma', v) < \alpha(\gamma, v)$.

To conclude, we obtain a configuration γ where for every $v \in V$, we have $\phi(\gamma, v) = 0$, so $\Phi(\gamma) = 0$. Also, for every configuration $\gamma' > \gamma$ we have proved that $\Phi(\gamma') = 0$. In other words $\Gamma_{\mathcal{B}} \rhd \Gamma_{\mathsf{DAG}}$. ◄

▶ **Lemma 5.** *Algorithm 1 converges in $O(k \log n)$ rounds, and exchanges $O(k\Delta \log n)$ messages before convergence.*

Proof. We use the potential function Φ to compute the number of rounds, remember that $\alpha(\gamma, v) = (\epsilon(\gamma, v), D(\gamma, v))$. Function $\epsilon(\gamma, v)$ is bounded by $\log n$, and Function $D(\gamma, v)$ by $2k$, so in at most $2k \log n$ rounds, each node v computes its set of links directions in the DAG. At each round, v reads $\delta(v)$ messages, so the system converges after $O(k\Delta \log n)$ messages. ◀

4 Vertex Coloring

We now present our self-stabilizing $(\Delta + 1)$ vertex coloring algorithm whose pseudo-code is given in Algorithm 2, that is built on top of Algorithm 1. We assume that each node v knows which neighbor has a lower identifier, and which has a higher one using variables Ord of Algorithm 1. Each node v maintains a color variable, denoted by $\mathsf{c}_v \in \{1, \ldots, \delta(v) + 1\}$. In addition, v maintains an array with all the known colors of its neighbors denoted $\mathsf{c}_v[u] \in \{1, \ldots, \Delta + 1\}$ for each port u. To do so, infinitely often, v send its own color to its neighbors by sending a message $< \mathsf{c}_v >$ (see Lines 4, 5, 9, and 13). If v has a neighbor u whose color is identical (in other words, if v detects a conflict) but identifier is lower, v changes its color for the minimum color not used by its neighbors, by executing function $Conflict(v, u)$ (see Lines 1-5).

Note that, since the maximum degree of the graph is Δ, the size of c_v is bounded by $\Delta \log \Delta$.

Algorithm 2: Vertex coloring

1 Function $Conflict(v,u)$ **is**   /* $\mathsf{c}_v[u] = \mathsf{c}_v$ */
2 $\quad$ **if** $(\forall w \in \mathsf{Ports}(v) : (\mathsf{c}_v[w] \neq \bot) \wedge ((\mathsf{c}_v[w] = \mathsf{c}_v) \Rightarrow (\mathsf{Ord}_v[w] = 1))$ **then**
3 $\quad\quad$ $\mathsf{c}_v := \min\{1, \ldots, \delta(v)\}\backslash\{\mathsf{c}_v[w] : w \in \mathsf{Ports}(v)\}$;
4 $\quad\quad$ **forall** $w \in \mathsf{Ports}(v) :$ send $< \mathsf{c}_v >$ to w ;
5 $\quad$ **else** send $< \mathsf{c}_v >$ to u;

6 Upon receipt of $< c >$ ***from port*** u **:**
7 $\quad$ $\mathsf{c}_v[u] := c$;
8 $\quad$ **if** $\mathsf{c}_v[u] = \mathsf{c}_v$ **then** $Conflict(v, u)$;
9 $\quad$ **else** send $< \mathsf{c}_v >$ to u;

10 Do forever :
11 $\quad$ **forall** $u \in \mathsf{Ports}(v) : Q(u, v) = \varnothing$
12 $\quad\quad$ **if** $\mathsf{c}_v[u] = \mathsf{c}_v$ **then** $Conflict(v, u)$;
13 $\quad\quad$ **else** send $< \mathsf{c}_v >$ to u ;

▶ **Theorem 6.** *Algorithm 2 solves the vertex coloring problem in a self-stabilizing manner in n-nodes graph with maximum degree Δ, assuming the message passing model and a spanning DAG. It uses $O(\Delta \log \Delta)$ bits of memory per node, $O(\log \Delta)$ bits per message. Moreover, it converges after $O(n)$ rounds and exchanges $O(\Delta\ k\ n)$ messages.*

In the following proof, we assume that the oriented graph described by variables Ord is a spanning DAG.

▶ **Lemma 7.** *Every time process v executes a step, it sends $< c_v >$ to every neighbor $u \in N(v)$.*

Proof. Each time v executes a step, for every neighbor $u \in N(v)$, there are two cases:

- v receives a message from u, so v executes $Conflict(v, u)$, or sends back $< c_v >$ to u.
- v executes **Do forever**, so it calls $Conflict(v, u)$, or sends back $< c_v >$ to u.

If v calls $Conflict(v)$, either v sends back $< c_v >$ to u, or v changes its color and sends the new one to every neighbor. ◀

Let $\alpha_1 : \Gamma \times V \to \mathbb{N}$ and $\alpha_2 : \Gamma \times V \to \mathbb{N}$ be the following functions:

$$\alpha_1(\gamma, v) = |\{u \in N(v) : c_u = c_v\}| \qquad \text{and} \qquad \alpha_2(\gamma, v) = |\{u \in N(v) : c_v[u] \neq c_u\}|$$

Let $\mu : \Gamma \times V \times \{1, \ldots, \Delta\} \to \mathbb{N}$ be the following function:

$$\mu(\gamma, v, c) = \begin{cases} 1 & \text{if } c_v \neq c \\ 0 & \text{otherwise} \end{cases}$$

Let $\alpha_3 : \Gamma \times V \to \mathbb{N}$ be the following function:

$$\alpha_3(\gamma, v) = \sum_{u \in N(v)} \left(\sum_{<c> \in Q(u,v)} \mu(\gamma, u, c) \right)$$

Let $A : \Gamma \to \mathbb{N}$ be the following potential function:

$$A(\gamma) = \sum_{v \in V} \big(\alpha_1(\gamma, v) + \alpha_2(\gamma, v) + \alpha_3(\gamma, v) \big)$$

We define $\Gamma_\alpha = \{\gamma \in \Gamma : A(\gamma) = 0\}$.

▶ **Lemma 8.** $\Gamma \rhd \Gamma_\alpha$.

Proof. Let $d_s(v)$ be the minimum distance from process v to a source of the DAG (*i.e.*, a process with only outgoing edges). First, we show that in finite time, every process v stops changing its color by induction on $d_s(v)$.

- **Base case:** If $d_s(v) = 0$, then v is a source of the DAG and $\forall u \in N(v)$, $\mathsf{Ord}_v[u] = 0$. Thus, v never changes its color c_v (see Line 2).
- **Induction step:** If $d_s(v) = x + 1$, then $\forall u \in N(v)$, either $\mathsf{Ord}_v[u] = 0$, or $d_s(u) < d_s(v) = x + 1$. By induction hypothesis, in finite time, $\forall u \in N(v)$ such that $d_s(u) \leqslant x$, the value of c_u eventually stops changing. Then, by Lemma 7, u sends $< c_u >$ to v infinitely often, so eventually $c_v[u] = c_u$. Now, either v never changes its color, or it gets a new color different from c_u. In the second case, v is no longer in conflict with any neighbor $u \in N(v)$ such that $d_s(u) < d_s(v)$. Node v can only be in conflict with neighbors $w \in N(v)$ such that $\mathsf{Ord}_v[w] = 0$, and thus v does not change its color anymore (see Line 2). Hence, v can change its color at most once.

Once every process stops changing its color forever, a process v cannot send a color different from its own. Thus, there exists a configuration γ_a such that, in every subsequent configuration γ, for every process v, v does not change its color and $\alpha_3(\gamma, v) = 0$. Moreover, in finite time, processes update their local knowledge about their neighbors' colors (see Lemma 7 and Line 7). Thus, there is a configuration γ_b after γ_a such that, in every configuration γ after γ_b, for every process v, $\alpha_2(\gamma, v) = 0$.

Finally, we show that in every configuration after γ_b, there is no color conflict. Assume by contradiction that in some configuration γ after γ_b, there is a color conflict. Let consider the process v in conflict that has the lowest ID among all such processes. For every $u \in N(v)$ such that $\mathsf{c}_v = \mathsf{c}_u$, $\mathsf{c}_v[u] = \mathsf{c}_u = \mathsf{c}_v$, and $\mathsf{c}_u[v] = \mathsf{c}_v = \mathsf{c}_u$. Moreover, by assumption, $\mathsf{Ord}_v[u] = 1$ and $\mathsf{Ord}_u[v] = 0$. Thus, in finite time, v calls the function $Conflict$, and changes its color (see Line 2), a contradiction. So, in every configuration γ after γ_b, and for every process $v \in V$, $\alpha_1(\gamma, v) = 0$. Hence, $A(\gamma) = 0$, and $\Gamma \rhd \Gamma_\alpha$. ◄

▸ **Lemma 9.** Γ_α *is closed.*

Proof. Let $\gamma \mapsto \gamma'$ such that $\gamma \in \Gamma_\alpha$. Let $v \in V$ and $u \in N(v)$.

First, in γ, $\mathsf{c}_v[u] = \mathsf{c}_u \neq \mathsf{c}_v$. Moreover, every message in $Q(u, v)$ contains color $\mathsf{c}_u \neq \mathsf{c}_v$. Thus, v does not change its color during step $\gamma \mapsto \gamma'$. Hence, $\alpha_1(\gamma', v) = 0$.

Since u and v do not change their color between γ and γ', and since every message in $Q(u, v)$ in γ equals c_u, then the value of $\mathsf{c}_v[u]$ does not change, and remains equal to c_u in γ'. Hence, $\alpha_2(\gamma', v) = 0$.

Finally, u does not change its color between γ and γ', and every message sent by u during step $\gamma \mapsto \gamma'$ equals $< \mathsf{c}_u >$ such as every other message in $Q(u, v)$ in γ that u did not treat during step $\gamma \mapsto \gamma'$. Thus, $\alpha_3(\gamma', v) = 0$ ◄

▸ **Lemma 10.** *Algorithm 2 converges in $O(n)$ rounds and exchanges $O(\Delta\ k\ n)$ messages before convergence.*

Proof. After one round, the system is purged of all possibly erroneous messages contained in the initial configuration. Moreover, by Lemma 7, at each round a process v sends its color $< \mathsf{c}_v >$ to all its neighbors, neighbors that will receive this message during the next round.

Now, consider a process v that changes its color during some round. The last time v changes its color during the round, it knows the real colors of its neighbors, *i.e.*, $\forall u \in N(v)$, $\mathsf{c}_v[u] = \mathsf{c}_u$. So by changing its color, v solves its color conflicts with its neighbors having a greater identity and does not create any new conflict with them. Since a process can change its color only if it is in conflict with a node of greater identity, v will not change its color anymore after this round. Hence, in the worst case, processes stop changing their colors after $O(n)$ rounds and there is no color conflict anymore.

In the worst case, at each round, a process replies to every message sent by its neighbors during the previous round. Thus, $O(\Delta\ k\ n)$ messages are exchanged.

◄

To avoid starvation of one of the algorithms, we compose them using a *fair composition* [10], *i.e.*, each process alternatively executes a step of Algorithm 1 and a step of Algorithm 2. As demonstrated in previous work [10], fair composition preserves the property of self-stabilization.

5   Concluding Remarks

We presented the first deterministic self-stabilizing solution in asynchronous message passing networks with unknown capacity links that only requires sublogarithmic (in n) memory and message size, when Δ is itself sublogarithmic.

Our approach is constructive and modular. In particular, our **DAG** algorithm layer solves a fundamental difficulty in many settings with respect to self-stabilization: avoidance of cyclic behaviors. We believe this can be a valuable asset when solving other problems in the same setting.

While our **Vertex coloring** algorithm layer does not guarantee a locally minimal coloring (if an initial configuration is a coloring that is not locally minimal, no conflict is found and thus recoloring occurs), a simple modification of the protocol permits to achieve this result: a node v that does not see a conflict but would like a smaller color asks its higher identity neighbors the authorization to take a new color c, any such neighbor u grants the authorization unless it has color c (but v didn't know it), or it itself wants to take a new color and is waiting for authorization. This mechanism does not create new conflicts, and any authorization chain length is bounded by the height of our constructed DAG. As a result, a minimal coloring is obtained, which can then be used for solving the maximal independent set problem: as it provides a locally minimal coloring, any node with color 0 can be seen as a member of the maximal independent set.

We believe the **DAG** and **Vertex Coloring** can prove useful for other local tasks, such as minimal dominating set, link coloring, or maximal matching. It would also be interesting to investigate their utility for solving global tasks (such as tree construction or leader election), and whether resource efficiency remains in this setting.

One property we retained about communication links is their FIFO behavior. Relieving this hypothesis in the context of unknown capacity links is likely to generate many impossibility results, that are left for future work.

References

1 Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

2 Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed (Δ + 1)-coloring below Szegedy-Vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. *CoRR*, 2017. URL: http://arxiv.org/abs/1712.00285.

3 Samuel Bernard, Stéphane Devismes, Katy Paroux, Maria Potop-Butucaru, and Sébastien Tixeuil. Probabilistic self-stabilizing vertex coloring in unidirectional anonymous networks. In *ICDCN'10*, volume 5935, pages 167–177, 2010.

4 Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *IPDPS'09*, pages 1–8, Rome, Italy, 2009.

5 Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. In *LATIN'18*, volume 10807, pages 161–173, 2018.

6 James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

7 Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10):498–514, 2006.

8 Sylvie Delaët and Sébastien Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing (JPDC)*, 62(5):961–981, 2002.

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

10 Shlomi Dolev. *Self-stabilization*. MIT Press, March 2000.

11 Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-FIFO channels with optimal fault-resilience. *Inf. Process. Lett.*, 111(18):912–920, 2011.

12 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997.

13   Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing (DC)*, 14(3):147–162, 2001.

14   Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science (TCS)*, 293(1):219–236, February 2003.

15   Sukumar Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, 1993.

16   Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.

17   Maria Gradinariu and Sébastien Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *OPODIS'00*, pages 55–70, 2000.

18   Ted Herman and Sébastien Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. In *AlgoSensors'04*, number 3121, pages 45–58, 2004.

19   Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

20   Toshimitsu Masuzawa and Sébastien Tixeuil. On bootstrapping topology knowledge in anonymous networks. *ACM Transactions on Adaptive and Autonomous Systems (TAAS)*, 4(1), 2009.

21   Nathalie Mitton, Bruno Séricola, Sébastien Tixeuil, Eric Fleury, and Isabelle Guérin-Lassous. Self-stabilization in self-organized multihop wireless networks. *Ad Hoc and Sensor Wireless Networks*, 11(1-2):1–34, 2011.

22   Sumit Sur and Pradip K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Inf. Sci.*, 69(3):219–227, 1993.

23   Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.

24   George Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2):486–510, 2000.

25   George Varghese and Mahesh Jayaram. The fault span of crash failures. *J. ACM*, 47(2):244–293, 2000.